OI Wiki (Beta)

OI Wiki 项目组

2024 年 3 月 24 日

目录

第 1 章

简介

1.1 Getting Started

欢迎来到 OI Wiki!

图 1.1 GitHub watchers

[1]

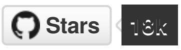


图 1.2 GitHub stars

[1]

图 1.3 Word Art

[1]

OI（Olympiad in Informatics，信息学奥林匹克竞赛）在中国起源于 1984 年，是五大高中学科竞赛之一。

ICPC（International Collegiate Programming Contest，国际大学生程序设计竞赛）由 ICPC 基金会（ICPC Foundation）举办，是最具影响力的大学生计算机竞赛。由于以前 ACM 赞助这个竞赛，也有很多人习惯叫它 ACM 竞赛。

OI Wiki 致力于成为一个免费开放且持续更新的**编程竞赛（competitive programming）**知识整合站点，大家可以在这里获取与竞赛相关的、有趣又实用的知识。我们为大家准备了竞赛中的基础知识、常见题型、解题思路以

及常用工具等内容，帮助大家更快速深入地学习编程竞赛中涉及到的知识。

本项目受 CTF Wiki[2] 的启发，在编写过程中参考了诸多资料，在此一并致谢。

参考资料与注释

[1]

图 1.4 GitHub watchers

[1-1] [1-2] [1-3]

[2] CTF Wiki

1.2 关于本项目

关于本项目

OI Wiki 致力于成为一个免费开放且持续更新的**编程竞赛 (competitive programming)** 知识整合站点。

交流方式

本项目主要使用 Issues[1]/QQ[2]/Telegram[3] 进行交流沟通。

Telegram 群组链接为 @OI_wiki[3]，QQ 群号码为 588793226[2]，欢迎加入。

note

原则上来说，上述群组是 **OI Wiki 讨论群**，所以请尽量不要在群组中发表过多与 **OI Wiki** 无关的内容。

项目方针

- OI Wiki 不是什么

参考资料与注释

[1] Issues

[2] QQ [2-1] [2-2]

[3] Telegram [3-1] [3-2]

1.3 如何参与

在文章开始之前，**OI Wiki** 项目组全体成员十分欢迎您为本项目贡献页面。正因为有了上百位像您一样的人，才有了 **OI Wiki** 的今天！

这篇文章将主要叙述参与 **OI Wiki** 编写的写作过程。请您在撰稿或者修正 Wiki 页面以前，仔细阅读以下内容，以帮助您完成更高质量的内容。

贡献指南

请您在编辑前查看 OI Wiki 贡献指南[3] 和 项目方针，以更好地和社区贡献者进行合作、交流。

参与协作

warning

在开始编写一段内容之前，请查阅 Issues[4]，确认没有别人在做相同的工作之后，开个 新 issue[5] 记录待编写的内容。

在这里引用维基百科的一句话：

> 不要害怕编辑，勇于更新页面！[1]

在 GitHub 上编辑

参与 **OI Wiki** 的编写**需要**一个 GitHub 账号（可以前往 GitHub 的账号注册页面[6] 页面注册），但**不需要**高超的 GitHub 技巧，即使你是一名新手，只要按照下面所述的步骤操作，也能够**非常出色**地完成编辑。

tip

在你的更改被合并到 **OI Wiki** 的主仓库之前，你对 **OI Wiki** 的内容所作出的修改均不会出现在 **OI Wiki** 的主站上，所以无需担心你的修改会破坏 **OI Wiki** 上正在显示的内容。

如果还是不放心，可以查看 GitHub 的官方教程[7]。

编辑单个页面内的内容

1. 在 **OI Wiki** 上找到对应页面；
2. 点击正文右上方（目录左侧）的「**编辑此页**」（edit）按钮，在确认您已经阅读了本页面和 格式手册 后点击按钮根据提示跳转到 GitHub 进行编辑；
3. 在编辑框内编写你想修改的内容。请注意，在修改和接下来的提交过程中，请**关闭您的自动翻译软件**，因为它可能产生不必要的麻烦（例如您修改的文件有时会被其错误改名，从而影响目录结构）；
4. 编写完成后滚动到页面下方，按照本文中 commit 信息格式规范填写 commit 信息，之后点击 **Propose changes** 按钮提交修改。点击按钮后，GitHub 会自动帮你创建一份 **OI Wiki** 仓库的分支，并将你的提交添加到这个分支仓库。
5. GitHub 会自动跳转到你的分支仓库的页面，此时页面上方会显示一个绿色的 **Create pull request** 按钮，点击后 GitHub 会跳转到一个创建 Pull Request 页面。向下滚动检查自己所作出的修改没有错误后，按照本文中 Pull Request 信息格式规范一节中的规范书写 Pull Request 信息，然后点击页面上的绿色的 **Create pull request** 按钮创建 Pull Request。
6. 不出意外的话，你的 Pull Request 就顺利提交到仓库，等待管理员审核并合并到主仓库中即可。

在等待合并的时间里，你可以给他人的 Pull Request 提意见、点赞或者点踩。如果有新消息，会在网页右上角出现提示，并附有邮件提醒（取决于个人设置中配置的通知方式）。

编辑多个页面内的内容

如果你需要同时编辑互相无关联的多个页面的内容，请按照上方的编辑单个页面内的内容一节一次修改所有页面。

1. 打开 OI-Wiki/OI-Wiki[8] 仓库，点击键盘上的 . 按钮（或者将 URL 中的 `github.com` 更改为 `github.dev`）[2-1]，进入 GitHub 的网页版 VS Code 编辑器；

2. 在编辑器中作出对页面源文件的更改，可以使用页面右上方的预览按钮（或按下 Ctrl+KV 快捷键）在右侧打开预览界面；
3. 修改完成后使用左侧的 Source Control 选项卡，并按照本文中 commit 信息格式规范填写 commit 信息并提交，提交时会提示是否创建此仓库的分支，点击绿色的 **Fork Repository** 按钮即可。
4. 提交后会在网页上方的中央弹出一个提示框，在第一次的提示框内填写标题，第二次的提示框内填写此提交要提交到的仓库内分支名称，之后右下角会弹出一个提示框，内容类似于 `Created Pull Request #1 for OI-Wiki/OI-Wiki.`，点击蓝字链接即可查看该 Pull Request。

向 Pull Request 追加更改

1. 打开 OI-Wiki 的 Pull Request 列表[9]，找到您提交的 Pull Request 并点击。
2. Pull Request 页面的标题下方将会有一段例如 `< 您的 ID> wants to merge x commits into OI-wiki:master from < 您的 ID>:patch-1` 的文字，点击 `< 您的 ID>:patch-1` 部分。
3. 您应该会被重定向到您的分支仓库中，而且文件列表左上角的分支名称是你提交 Pull Request 的分支名称（在本示例中应为 `patch-1`）。
4. 进行您需要的更改。
    - 如果您需要编辑单个文件或多个互相无关联的页面的内容，请直接找到你要的文件并进行更改，更改完成后滚动到页面下方，按照本文中 commit 信息格式规范填写 commit 信息，之后点击 **Commit changes** 按钮提交修改。
    - 如果您需要编辑多个文件，点击键盘上的. 按钮（或者将 URL 中的 `github.com` 更改为 `github.dev`）[2-2]，进入 GitHub 的网页版 VS Code 编辑器并作出更改。然后使用左侧的 Source Control 选项卡，并按照本文中 commit 信息格式规范填写 commit 信息并提交修改。
5. 这时你的更改会被自动追加在您的 Pull Request 中。

使用 Git 在本地进行编辑

warning

对于一般用户，我们更推荐使用上方所述的 GitHub 的 Web 编辑器进行编辑。

虽然大多数情况下您可以直接在 GitHub 上进行编辑，但对于一些较为特殊的情况（如需要使用 GPG 签名），我们更推荐使用 Git 在本地进行编辑。

大致流程如下：

1. 将主仓库 Fork 到自己的仓库中；
2. 将 Fork 后的分支仓库克隆（clone）到本地；
3. 在本地进行修改后提交（commit）这些更改；
4. 将这些更改推送（push）到你克隆的分支仓库；
5. 提交 Pull Request 至主仓库。

详细的操作方式可以参考 Git 页面。

向 Pull Request 追加更改

在 clone 下来的本地分支仓库中继续进行修改，并提交（commit）以及推送（push）这些更改即可。你的更改会被自动追加在 Pull Request 中。

在构建的网页中预览变更

在 Pull Request 页面下方可以找到测试页面，点击 netlify/oi-wiki/deploy-preview 一项的 Details 链接（如下图），可以进入自动构建的，由您变更后的页面供您预览。

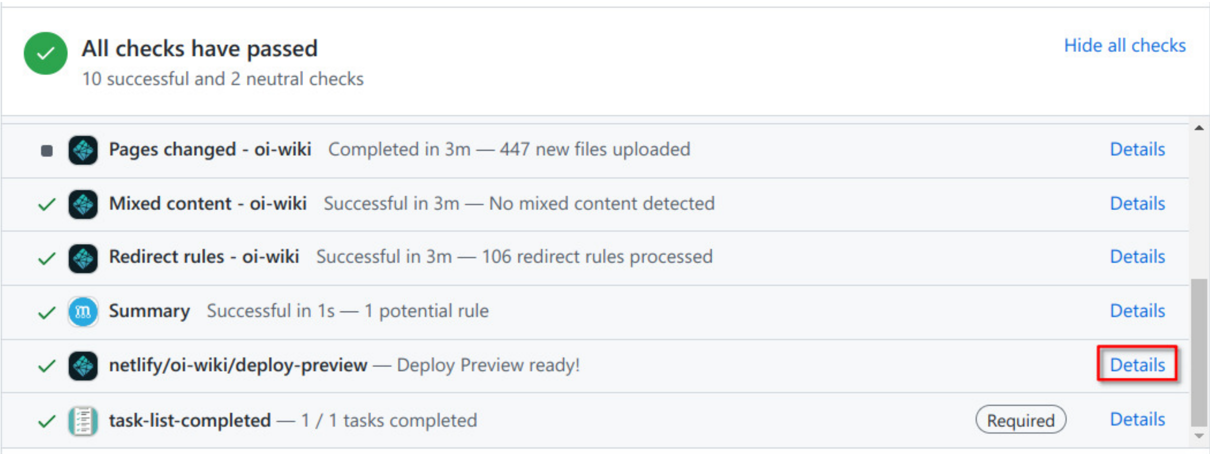


图 1.5 deploy_preview

对于目录和引用的变更

通常情况下，如果您需要添加一个新页面，或者修改已有页面在目录中的链接，您就需要对 `mkdocs.yml`[10] 文件作出改动。

添加新页面可以参考既有的格式。但除非是进行重构或修正名词，否则**我们不建议对既有页面的引用链接进行修改**，Pull Requests 中不必要的修改也将被驳回。

如果您坚持要修改链接，请注意更新 author 字段和重定向文件。

author 字段

GitHub API 在文件目录变更后不能跟踪统计，所以我们在文件头手动维护了一个作者列表来解决这个问题。author 字段位于整个 Markdown 文件的开头，形如 `author: Ir1d, cjsoft`，相邻两个 ID 之间用逗号加空格隔开。这里的 ID 是 GitHub 的用户名，即 GitHub profile 的地址（例如 https://github.com/Ir1d[11] 中的 `Ir1d`）。

修改链接时，需要将当前页面中的 contributors 逐一填入 author 字段。

重定向文件

在修改链接时，为了避免在站外引用时出现死链，需要修改重定向文件。

`_redirects`[12] 文件用于生成 netlify 的配置[13] 和 用于跳转的文件[14]。

每一行表示一个重定向规则，分别写跳转的起点和终点的 url（不包含域名）：

```
/path/to/src /path/to/desc
```

注：所有跳转均为 301 跳转，只有在修改目录中 url 造成死链的时候需要修改。

Commit 信息格式规范

对于提交时需要填写的 commit 信息，请遵守以下几点基本要求：

1. commit 摘要请简要描述这一次 commit 改动的内容。注意 commit 摘要的长度不要超过 50 字符，超出的部分会自动置于正文中。

2. 如果需要进一步描述本次 commit 内容，请在正文中详细说明。

对于 commit 摘要，推荐按照如下格式书写：

```
<修改类型>(<文件名>): <修改的内容>
```

修改类型分为如下几类：

- `feat`：用于添加内容的情况。
- `fix`：用于修正现有内容错误的情况。
- `refactor`：用于对一个页面进行重构（较大规模的更改）的情况。
- `revert`：用于回退之前更改的情况。

Pull Request 信息格式规范

对于 Pull Request，请遵守以下几点要求：

1. 标题请写明本次 PR 的目的（做了**什么**工作，修复了**什么**问题）。
2. 内容请简要叙述修改的内容。如果修复了一个 issue 的问题，请在内容中添加 `fix #xxxx` 字段，其中 `xxxx` 代表 issue 的编号。
3. 请您仔细阅读 贡献指南[3] 和 社区公约[15]，并在同意后勾选 PR 模板中的框，表示您同意了以上指南和公约。

对于 Pull Request 的标题，推荐使用如下格式书写：

```
<修改类型>(<文件名>): <修改的内容> (<对应 issue 的编号>)
```

修改类型分为如下几类：

- `feat`：用于添加内容的情况。
- `fix`：用于修正现有内容错误的情况。
- `refactor`：用于对一个页面进行重构（较大规模的更改）的情况。
- `revert`：用于回退之前更改的情况。

示例：

- `fix(ds/persistent-seg): 修改代码注释使描述更清晰`
- `fix: tools/judger/index 不在目录中 (#3709)`
- `feat(math/poly/fft): better proof`
- `refactor(ds/stack): 整理页面内容`

协作流程

1. 在收到一个新的 Pull Request 之后，GitHub 会给 reviewer 发送邮件；
2. 与此同时，在 GitHub Actions[16] 和 Netlify[17] 上会运行两组测试，它们会把进度同步在 PR 页面的下方。GitHub Actions 主要用来确认 PR 中内容的修改不会影响到网站构建的进程；Netlify 用来把 PR 中的更新构建出来，方便 reviewer 审核（在测试完成后点击 Details 可以了解更多）；
3. reviewer 可能会发现问题，并提出 `review` 或 `suggested changes`（建议更改，显示为灰色图标）/`requested changes`（强制更改，显示为红色图标，只会在 reviewer 拥有 repo 写权限时出现）。一般来说，reviewer 也会附上建议和需要进行的更改，在这时，您将会需要继续向 Pull Request 追加其他更改。更改的方法可以参考在 `GitHub 上编辑或者使用 Git 在本地进行编辑部分的向 Pull Request 追加更改部分`。
4. 在足够多 reviewer 投票通过一个 PR 之后，这个 PR 才可以合并到 master 分支中；
5. 在合并到 master 分支之后，GitHub Actions 会重新构建一遍网站内容，并更新到 gh-pages 分支；

6. 这时服务器才会拉取 gh-pages 分支的更新，并重新部署最新版本的内容。

参考资料与注释

[1] 维基百科： 新手入门 / 编辑

[2] Web-based editor - GitHub Codespaces - GitHub Docs [2-1] [2-2]

[3] OI Wiki 贡献指南 [3-1] [3-2]

[4] Issues

[5] 新 issue

[6] GitHub 的账号注册页面

[7] GitHub 的官方教程

[8] OI-Wiki/OI-Wiki

[9] OI-Wiki 的 Pull Request 列表

[10] `mkdocs.yml`

[11] https://github.com/Ir1d

[12] `_redirects`

[13] netlify 的配置

[14] 用于跳转的文件

[15] 社区公约

[16] GitHub Actions

[17] Netlify

1.4 OI Wiki 不是什么

Authors: abc1763613206, HeRaNO, NachtgeistW, r-value, Tiphereth-A, wlbksy, YZircon, 0zu-cc

" 注意"

作为项目方针的一部分，本页面十分重要，每个贡献者都应确保您的贡献满足如下条件。

OI Wiki 不是发表原创研究的场所

作为一个 Wiki，**OI Wiki** 不是发表 原创研究[1]（如**新理论及解法**、**原创观点**、**自创定义或词语**等）的场所。例如：

- 您发现了某题目的非常规做法，若您不能证明该做法已经被应用于其他题目中，则**不应**在 **OI Wiki** 中开设单独的界面。
- 您提出了新的算法或数据结构，若您不能证明该内容已经被用于解决编程竞赛中的某一类问题，则**不应**将其提交至 **OI Wiki**。

OI Wiki 不是新闻的收集处

作为泛中文为主语境之下、以编程竞赛相关内容为主的知识整合站点，**OI Wiki** 侧重于提供**稳定沉淀并已取得广泛认可**的信息。

换言之，除非是权威机构发布的信息（如中国计算机协会发布的赛制更新），您所贡献的内容应当是已经经过检验沉淀，获得了广泛认可，且在一段时间内不会产生时效性问题的信息。例如：

- 您发现 X 博士在某个位置发布了新的算法，若您想将其加入 **OI Wiki** 中，此时您应该观察其是否能获得广泛认可（如可作为正式比赛中的泛用优秀解法），且与 **OI Wiki** 中现存的算法有一定的区分性。
- 您发现 **OI Wiki** 中提到的某软件或人物出现了舆情问题，此时请您关注评论区等的相关公告，切勿重复开 issue 等说明问题。**OI Wiki** 的目的在于记载**可长久流传的信息**，而不是**带有时效性的临时信息**。

OI Wiki 不是档案馆

OI Wiki 主站不会收录各类**文献**、**课件**、**讲义**、**说明书**等资料，若您想提交有关编程竞赛的资料，请移步至 OI-wiki/libs[2]。

OI Wiki 不是宣传工具

OI Wiki 是一个知识整合站点，**不是**演讲台、论坛、宣传工具等，因此在添加或修改条目时，请勿：

- 发表观点或评论：**OI Wiki** 的内容必须**客观中立**，如果您想要发表与 **OI Wiki** 有关的观点与评论，请移步至评论区或 issue 页面。如果您想要发表的观点与评论与 **OI Wiki** 无关，请移步至个人博客或论坛；
- 作任何形式的宣传行为：出于知识收录需要，**OI Wiki** 可以接受对「在编程竞赛领域已经**广为人知**的网站或软件」的介绍。除此之外，**OI Wiki** 不会接受任何非赞助商提供的任何宣传性质的内容。

OI Wiki 不是权威机构

作为一个主要依靠用户贡献的社区项目，**OI Wiki** 不具有权威性，不应也没有能力作为一个权威机构。**OI Wiki** 可以作为学习编程竞赛相关知识的参考，而不是标准教科书。作为一个社区维护的参考站点，您不应将 **OI Wiki** 用作最终的权威标准（如作为编程竞赛的「考纲」），也不应盲目采信所有 **OI Wiki** 的内容。**OI Wiki** 不对由使用 **OI Wiki** 的内容而产生的任何后果负责。

OI Wiki 不是个人博客

OI Wiki 基于「Wiki」一词，本质上是依靠用户**协同编辑内容**的社区。这就意味着，您所提供的内容应当**尽量减轻个人色彩**，服务于整个 Wiki，关注您所贡献的相关词条的结构。切勿以撰写个人博客文章的思维对待 Wiki 中的条目。例如：

- 您**不应**将 **OI Wiki** 作为您个人博客的导航站点；

- 您**不应**在词条的描述中加入条目历史、个人吐槽、冷笑话等无关内容。

以上例子的反面均为撰写个人博客时的常见思维，而这些在提倡客观中立的 **OI Wiki** 中是不受欢迎的。

OI Wiki 不是译名标准委员会

OI Wiki 的目标群体不是历史学家及语言学家，因此在外文名词的翻译中，**OI Wiki** 倾向于使用中文语境下**已经获得了广泛认同**的译文，即使它们可能是有瑕疵的。

在译文已经获得了广泛熟知认同的基础上，盲目因「正确性」而修改现有的译文只会引发更大的混淆与混乱，与之引发的文字游戏也会影响沟通效率。

OI Wiki 不是百科全书

OI Wiki 收录的内容应限定在「已经被应用于编程竞赛」的计算机科学、数学等领域的部分知识。其他与编程竞赛无关的领域或知识均不适合收录于 **OI Wiki**。

例如，如下的领域或知识**不适合**收录于 **OI Wiki**：

- 历史，艺术等无关领域；
- TBN 矩阵[3] 等虽然从属于有关领域但目前不能应用于编程竞赛的知识；
- PID 控制[4]、有限元法[5] 等目前不能应用于编程竞赛的领域算法；
- 深度学习[6]、强化学习[7] 等目前不能应用于编程竞赛的通用算法。

OI Wiki 不是编程语言的文档和学习指南

OI Wiki 收录的算法相较于代码实现，应更关心算法本身。算法的具体实现仅作为一种更加细致的理解或者实现提示，而不是给具体的语言学习者以方便。如果您想要学习某种语言，您应该阅读该语言的官方文档等资料。

出于知识收录需要，**OI Wiki** 可以收录编程竞赛常用语言的简单使用指南。除此之外，**OI Wiki** 不会收录诸如「某编程语言的某标准库里某函数的实现细节等」与编程竞赛和算法关系不大的内容。

参考资料与注释

[1] 原创研究

[2] OI-wiki/libs

[3] TBN 矩阵

[4] PID 控制

[5] 有限元法

[6] 深度学习

[7] 强化学习

1.5 格式手册

在文章开始之前，**OI Wiki** 项目组全体成员十分欢迎您为本项目贡献页面。正因为有了上百位像您一样的人，才有了 **OI Wiki** 的今天！

本页面将列出在 **OI Wiki** 编写过程时推荐使用的格式规范与编辑方针。请您在撰稿或者修正 Wiki 页面以前，仔细阅读以下内容，以帮助您完成更高质量的内容。

如果您已迫不及待，想要快速上手，建议先阅读图片举例的章节。

贡献文档要求

当你打算贡献某部分的内容时，你应该尽量熟悉以下三部分：

- 文档存储的格式
- 文档的合理性
- remark-lint 和 $\LaTeX$ 公式的格式要求

文档引用与存储的格式

- **文件名请务必都小写，以 `-` 分割**。例如：`file-name.md`。
- 请务必确保文档中引用的**外链**图片已经全部转存到了**本库内**对应的 `images` 文件夹中（防止触发某些网站的防盗链），建议处理成 `MD 文档名称 + 编号`的形式（可参考已有文档中图片的处理方式）。例如：本篇文档的文件名称为 format，则文档中引用的第一张图片的名字为 `format1.png`。
- 推荐使用 SVG 格式的图片[6]，以获取较好的清晰度和缩放效果。
- 动图如果无法或者不会制作 SVG 格式的，则推荐使用 APNG 格式[8] 的文件。Windows 用户可使用 ScreenToGif[10] 录制，Linux 用户可使用 Peek[11] 录制，注意需要在设置里调整为录制 APNG。其他情况则推荐先制作为 MP4 等视频文件再转换为 APNG，如果使用 ffmpeg 则可以使用 `ffmpeg -i filename.mp4 -f apng filename.apng -plays 0` 转换。[9]
- 同时具有源文件和导出图像的图片（例如 JPG 文件与 PSD 文件或者 SVG 图像与 TikZ TeX 源代码），建议将源文件以与图片相同的文件名保存于同一目录下。
- 请确保您的文档中的引用链接的稳定性。**不推荐**引用**自建**服务中的资源（如自建 OJ 里的题目）。建议在添加时同时将该外链存于互联网档案馆[7]，以防无法替代的链接失效。
- 站内链接请去掉网站域名，并且使用相对路径链接对应 `.md` 文件。例如，在本页面（`intro/format`）中链接杂项简介（`misc`），应使用 `[ 杂项简介 ](../misc/index.md)`。可以在链接中添加 hash 来链接到某一节，例如 `[Pull Request 信息格式规范 ](./htc.md#pull-request- 信息格式规范 )`，hash 的值可以通过位于每个标题右侧的按钮或者位于网页右侧的目录中的链接得到。

文档的合理性

合理性，指所编写的**内容**必须具有如下的特性：

- 由浅入深，内容的难度应该具有渐进性。
- 逻辑性。
  - 对于算法或数学概念类内容的撰写应该尽量包含以下的内容：
    1. 原理：说明该内容对应的原理；
    2. 例子：给出 1 ~ 2 个典型的例子；
    3. 题目：在该标题下，**只需要给出题目名字和题目链接**。对于算法类题目，题目链接 OJ 的优先级为：原 OJ（国外 OJ 要求国内可流畅访问）> UOJ > LOJ > 洛谷。

示例页面：IDA*

– 对于工具类内容的撰写应该尽量包含以下的内容：

1. 简介：阐明该工具的背景与用途。
2. 配置方式：详细给出配置环境与使用的过程，下载与安装方法建议尽量引用官方文档。

示例页面：WSL (Windows 10)

除现有内容质量较低的情况外，建议尽量从**补充**的角度来做贡献，而非采取直接覆盖的方式。如果拿不准主意，可以参考 关于本项目的交流方式 一节，与 **OI Wiki** 项目组联系。

文档的基本格式要求

Remark-lint 的格式要求

remark-lint[12] 可以自动给项目内文件统一风格。**OI Wiki** 现在启用的配置文件托管在 .remarkrc[13]。

在配置过程中 **OI Wiki** 项目组也遇到了一些 remark-lint 不能很好处理的问题，所以请严格按照下列要求编辑文档：

- 不要使用如 `<h1>` 或者 `#  标题` 的一级标题。
- 标题要空一个英文半角空格，例如：`##  简介`。
- 由于 remark-lint 不能很好地处理删除线，因此请不要使用删除线语法（不使用删除线语法的另外一个原因是，删除线划去的内容大多为「抖机灵」性质，对读者理解帮助不大，不符合下面的「文本内容的格式要求」中对内容表述的要求）。
- 列表：
  – 列表前要有空行，新开一段。
  – 使用有序列表（如 `1.  例子`）时，点号后要有空格。
- 行间公式前后各要有一行空行，否则会被当做是行内公式。
- 使用 `???` 或 `!!!` 开头的 Details 语法时，每一行要包括在 Details 语法的文本框的文本，开头必须至少有 4 个空格。

  即使是空行，也必须保持与其他行一致的缩进。请不要使用编辑器的自动裁剪行末空格功能。

  " 示例"

  ```
  ???+ warning
      请记得在文本前面添加 4 个空格。其他的语法还是与 Markdown 语法一致。

      不添加 4 个空格的话，文本就不会出现在 Details 文本框里了。

      这个`???`是什么的问题会在下文解答。
  ```

  warning

  请记得在文本前面添加 4 个空格。其他的语法还是与 Markdown 语法一致。

  不添加 4 个空格的话，文本就不会出现在 Details 文本框里了。

  这个 `???` 是什么的问题会在下文解答。

- 代码样式的纯文本块请使用 ` ```text`。直接使用 ` ``` ` 而不指定纯文本块里的语言，可能会导致内容被错误地缩进。

标点符号的使用

- 请在每句话的末尾添加**句号**。

- 请正确使用**全角**标点符号与**半角**标点符号。汉语请使用全角符号，英语请使用半角符号。中文中夹用英文时，请参考 中文出版物夹用英文的编辑规范[14]。
- 由于 “……” 未区分全半角，请使用 「……」 作为全角引号，"..." 作为半角引号。
- 注意区分**顿号**与**逗号**的使用。
- 注意**括号**的位置。句内括号与句外括号的位置不同。
- 通常使用**分号**来表示列表环境中各复句之间的关系。
- 对于有序列表，推荐在每一项的后面添加**分号**，在列表最后一项的后面添加**句号**；对于无序列表，推荐在每一项的后面添加**句号**。
- 注意区分各种不同的连接号，如 hyphen（一般使用 U+002D hyphen-minus（-），即键盘上的「减号」代替），U+2013 en dash（–）和 U+2014 em dash（—）。（英文中连接多个人名时，须用 en dash，但是极常误用为 hyphen。其他误用较为罕见，基本上只需记住这一点即可。）详见 连接号 - 维基百科[15]。

" 示例"

– 中学生学科竞赛主要包括信息学奥林匹克竞赛、信息学奥林匹克竞赛、信息学奥林匹克竞赛、信息学奥林匹克竞赛和信息学奥林匹克竞赛（谁写的这个示例，建议抬走）。

–「你吃了吗?」李四问张三。

– 我想对你说:「我真是太喜欢你了。」

–「苟利国家生死以，岂因祸福避趋之!」

– 张华考上了大学；李萍进了技校；我当了工人：我们都有美好的前途。[1]

– 以下是这个算法的基本流程:

1. 初始化到各点的距离为无穷大，将所有点设置为未被访问过，初始化一个队列；
2. 将起点放入队列，将起点设置为已被访问过，更新到起点的距离为 0；
3. 取出队首元素，将该元素设置为未被访问过；
4. 遍历所有与此元素相连的边，若到这个点存在更短的距离，则进行松弛操作；
5. 若这个点未被访问过，则将这个点放入队列，且设置这个点为已经访问过；
6. 回到第三步，直到队列为空。

– KMP 算法（Knuth–Morris–Pratt algorithm, KMP algorithm）由 Knuth、Pratt 和 Morris 在 1977 年共同发布。[2]

Markdown 格式与主题扩展格式要求

- 表示强调时请使用 `**SOMETHING**` 和 「」，而非某级标题，因为使用标题会导致文章结构层次混乱和（或）目录出现问题。
- 请正确使用 Markdown 的区块功能。插入行内代码请使用一对反引号包围代码区块；行间代码请使用一对 ``` 包围代码区块，其中反引号就是键盘左上角波浪线下面那个符号，行间代码请在第一个 ``` 的后面加上语言名称（如： ```cpp）。

" 示例"

````
```cpp
// #include<stdio.h>     //不好的写法
#include <cstdio>   //好的写法
```
````

```cpp
// #include<stdio.h>     //不好的写法
#include <cstdio>   //好的写法
```

- 「参考资料与注释」使用 Markdown 的脚注功能进行编写。格式为：

```
文本内容。[^ 脚注名]
[^ 脚注名]: 参考资料内容。注意：冒号是英文冒号，冒号后面跟着一个空格。
```

脚注名既可以使用数字也可以使用文本。脚注名摆放的位置与括号的用法一致。为美观起见，建议同一个页面内的脚注名遵循统一的命名规律，如：ref1、ref2、note1……

脚注的内容统一放在 `##  参考资料与注释`二级标题下。

" 示例"

```
当 `#include <cxxxx>` 可以替代 `#include <xxxx.h>` 时，应使用前者。[^ref1]

2020 年 1 月 21 日，CCF 宣布恢复 NOIP。[^ref2]

## 参考资料与注释

[^ref1]: [cstdio stdio.h namespace](https://stackoverflow.com/questions/10460250
/cstdio-stdio-h-namespace)

[^ref2]: [CCF 关于恢复 NOIP 竞赛的公告-中国计算机学会](https://www.ccf.org.cn/c/202
0-01-21/694716.shtml)
```

当 `#include <cxxxx>` 可以替代 `#include <xxxx.h>` 时，应使用前者。[3]

2020 年 1 月 21 日，CCF 宣布恢复 NOIP。[4]

- 建议使用主题扩展的 `???+note` 格式（即 Collapsible Blocks[16]）来描述题面和参考代码。也可以用这种格式来展示其他需要补充介绍的内容。

示例代码：

```
??? note " 标题"
    这个文本框会被默认折叠。

    推荐将 ** 解题代码 ** 放在折叠文本框内。

???+note "[HDOJ 的「A + B Problem」](https://vjudge.net/problem/HDU-1000)"
    标题也可以使用 Markdown 的超链接。这里的超链接是 HDOJ 的「A + B Problem」。

    而且推荐以这种方式 ** 标注原题链接 **。

    注意双引号的位置。
```

效果：

" 标题"

这个文本框会被默认折叠。

推荐将**解题代码**放在折叠文本框内。

"HDOJ 的「A + B Problem」[17]"

标题也可以使用 Markdown 的超链接。这里的超链接是 HDOJ 的「A + B Problem」。

而且推荐以这种方式**标注原题链接**。

注意双引号的位置。

两种格式的区别是，带 `+` 的会默认保持展开，而不带 `+` 的会默认保持折叠。

折叠框的标题，即 `???+note` 中 `note` 后的内容应以 `"` 包裹起来。其中的内容支持 Markdown 语法。详见 Admonition - Changing the title[18]。（不具备折叠功能的为一般的 Admonitions，参考 Admonitions - Material for MkDocs[19]）

- 当需要添加不同语言的代码时，推荐使用 Content tabs，可以实现不同语言代码的切换。Content tabs 还有其他的用法，详见 Content tabs[20]。其使用方法和效果如下。

" 示例"

```
注意需要在文本前面添加 4 个空格。其他的语法还是与 Markdown 语法一致。

=== "C"

    ``` c
    #include <stdio.h>

    int main(void) {
      printf("Hello world!\n");
      return 0;
    }
    ```

=== "C++"

    ``` c++
    #include <iostream>

    int main(void) {
      std::cout << "Hello world!" << std::endl;
      return 0;
    }
    ```
```

```c
#include <stdio.h>

int main(void) {
  printf("Hello world!\n");
  return 0;
}
```

```cpp
#include <iostream>

int main(void) {
  std::cout << "Hello world!" << std::endl;
  return 0;
}
```

如果对 mkdocs-material（我们使用的这个主题）还有什么问题，还可以查阅 MkDocs 使用说明[21]，其介绍了 mkdocs-material 主题的插件使用方式。

文本内容的格式要求

- 所有的 **OI Wiki** 文本都应使用粗体标记。

- 在页面的开头应有一段简短的文字（如「本页面将介绍⋯⋯」），用于概述页面内容。

" 示例"

本页面将列出在 **OI Wiki** 编写过程时推荐使用的格式规范与编辑方针。

- 涉及到「前置知识」的页面，请在开头添加一行**前置知识：⋯⋯**，放在页面概述前。格式如下：

`前置知识：[站内页面 1](url1)、[站内页面 2](url2) 和 [站内页面 3](url3)`

" 示例"

前置知识：时间复杂度

```
本页面将介绍基础的计算理论的知识。
```

- 请注意文档结构。文档结构应当十分条理，层次清晰。请不要让诸如「五级标题」这种事情再次发生了，一篇正常的文章是用不到如此复杂的结构层次的。
- 请注意内容的表述。作为一个百科网站，**OI Wiki** 使用的语言应该是书面的，客观的。诸如「抖机灵」性质的，对读者理解帮助不大的内容，不应该出现在 **OI Wiki** 当中。
- 请尽量为链接提供完整的标题、或者可被识别的提示，避免使用裸地址和「这」、「此」之类的模糊不清的描述。每一个超链接都应尽量对其加以清楚明确的描述，方便读者明白该超链接将指向何处。

  建议使用源文章或者标签页的标题。

fail " **不推荐的写法**"

```
请参考 [这个页面](https://docs.github.com/en/github/collaborating-with-issues-and
-pull-requests/syncing-a-fork)

请参考 <https://docs.github.com/en/github/collaborating-with-issues-and-pull-req
uests/syncing-a-fork>
```

请参考 这个页面[22]

请参考 https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/syncing-a-fork[22]

" **推荐的写法**"

```
请参考 GitHub 官方的帮助页面 [Syncing a fork - GitHub Docs](https://docs.github.
com/en/github/collaborating-with-issues-and-pull-requests/syncing-a-fork)
```

```
请参考 GitHub 官方的帮助页面 [Syncing a fork - GitHub Docs](https://docs.github.
com/en/github/collaborating-with-issues-and-pull-requests/syncing-a-fork)
```

- 受 Markdown 格式限制，`## 参考资料与注释`二级标题必须放在文末。
- 所有用作序号的数字建议使用中文。示例：
  - 数列的第一项。
  - 输入文件的第一行。
- 请尽量避免在标题中使用 MathJax 公式，无论是几级标题。在标题中使用公式有可能会导致目录显示错误。[5]
- 请注意代码的可读性。
  - 代码应拥有清晰的逻辑。
  - 建议在参考代码中添加适当注释以方便读者理解。
  - 尽量避免出现影响阅读的预编译指令和宏定义。

LaTeX 公式的格式要求

LaTeX 作为公式排版的首选，我们应当正确地使用它。因此对于 LaTeX 的使用我们有严格的要求。如果您想要快速上手，可以阅读本章节末给出的表格。

- 您使用的符号不应与 数学符号表 规定的符号冲突。
- 使用 Roman 体表示数字、常量、算子和函数。使用 Italic 体表示变量、下标。LaTeX 已经预先定义好了一些常见的常量、函数、运算符等，我们可以直接调用，包括但不限于：

  ```
  \log, \ln, \lg, \sin, \cos, \tan, \sec, \csc, \cot, \gcd, \min, \max, \exp, \inf
  , \mod, \bmod, \pmod
  ```

  所以在输入常量、函数名、运算符等时，请先检查一下是否应该使用 Roman 体或其它字体。LaTeX 符号的书写可参考 KaTeX 的 Supported Functions 页面[23]（不是全部），也可以搜索求解。

  由于 LaTeX 书写 Roman 体小写希腊字母较为困难，故小写希腊字母常量、算子和函数可以使用 Italic 体，如 π 以及 δx 中的 δ.

  如果遇到没有预先定义好的需要使用 Roman 体的**函数名**，我们可以使用 `$\operatorname{something}$` 来产生，如我们可以使用 `$\operatorname{lcm}$` 产生正体的最小公倍数（函数）符号。同理，产生 Roman 体的**常量**应用 `$\mathrm{}$`；产生 Roman 体粗体符号应用 `$\mathbf{}$`；产生 Italic 体粗体符号应用 `$\boldsymbol{}$`（如向量 $\boldsymbol{a}$）。对于多字母的变量，应当使用 `$\textit{}$`。其他非数学内容，包括英文、特殊符号等，一律使用 `$\text{}$`。中文我们则建议不放在 LaTeX 公式中。
- 如果表达式须折行（常见于较长的行间公式中），则应遵循如下换行规则：
  - 将换行符放在 $=$，$+$，$-$，$\pm$，$\mp$ 之前，如果有必要，也可放在 $\times$，$\cdot$，$/$ 之前，如：

    $$\begin{aligned} \mathrm{e}^x &= \sum_{n=0}^{\infty} \frac{x^n}{n!} \\ &= 1 + x + \frac{x^2}{2} \\ &\quad + \frac{x^3}{6} + \frac{x^4}{24} + \ldots \end{aligned}$$

  - 同一运算符不应在换行符前后同时出现，
  - 换行符尽量不要出现在括号内的表达式中。
- 在行内使用分数的时候，请使用 `$\dfrac{}{}$`。比如 `$\dfrac{1}{2}$`，效果 $\dfrac{1}{2}$，而不是 `$\frac{1}{2}$`，效果 $\frac{1}{2}$。
- 组合数请使用 `\dbinom{n}{m}`，效果 $\dbinom{n}{m}$，而不是 `{n \choose m}`（在 LaTeX 中这种写法已不推荐）；与上一条关于分数的约定相似，请不要使用 `\binom{n}{m}`，效果 $\binom{n}{m}$。
- 尽可能避免在行内使用巨运算符（如 $\sum$，$\prod$，$\int$ 等）。
- 在不会引起歧义的情况下，请用 `$\times$` 代替星号，叉乘请使用 `$\times$`，点乘请使用 `$\cdot$`。如 $a \times b$，$a \cdot b$，而不是 $a * b$。
- 请用 `$\cdots$`（居于排版基线与顶线中间），`$\ldots$`（居于排版基线的位置），`$\vdots$`（竖着的省略号）代替 `$...$`。如 $a_1, a_2, \cdots a_n$，而不是 $a_1, a_2, ...a_n$。
- 请注意，不要在非代码区域使用任何程序设计语言的表示方式，而是使用 LaTeX 公式。例如，使用 `$=$` 而不是 `$==$`（如 $a = b$，而不是 $a == b$）、使用 `` `a<<1` `` 或者 `$a\times 2$` 而不是 `$a<<1$`、使用 `$a\bmod b$` 代替 `$a\%b$`（如 $a \bmod b$，而不是 $a\%b$）等。
- 公式中不要使用中括号连缀（即 C++ 高维数组的表示方式）而多使用下标。即 $a_{i,j,k}$ 而不是 $a[i][j][k]$。在公式中下标较复杂的情况下建议改用多元函数（$f(i,j,k)$）或内联代码格式。对于一元简单函数使用 `$f_i$`、`$f(i)$` 或 `$f[i]$` 均可。
- 为了统一且书写方便，复杂度分析时大 O 记号请直接使用 `$O()$` 而不是 `$\mathcal O()$`。
- 在表示等价关系时，请使用 `$\iff$`，效果 $\iff$，而不是 `$\Leftrightarrow$`，效果 $\Leftrightarrow$。

- 分段函数环境 `cases` **只能有两列**（即一个 `&` 分隔符）。
- 请不要滥用 LaTeX 公式。这不仅会造成页面加载缓慢（因为 MathJax 的效率低是出了名的），同时也会导致页面的排版混乱。我们通常使用 LaTeX 公式字体表示变量名称。我们的建议是，如非必要，尽量减少公式与普通正文字体的**大量**混合使用，如非必要，尽量不要使用公式，如：

  ```
  我们将要学习 $Network-flow$ 中的 $SPFA$ 最小费用流，需要使用 $Edmonds-Karp$ 算法进
  行增广。
  ```

  就是一个典型的**滥用公式字体**的例子。（在页面中使用斜体请用 `* 文本 *` 表示。）
- 请正确使用对应的 LaTeX 符号，尤其是公式中的希腊字母等特殊符号。如欧拉函数请使用 `$\varphi$`，圆的直径请使用 `$\Phi$`，黄金分割请使用 `$\phi$`。这些符号虽然同样表示希腊字母 Phi，但是在不同的环境下有不同的含义。切记**不要使用输入法的插入特殊符号**来插入这种符号。

  另外，由于 LaTeX 历史原因，空集的符号应为 `$\varnothing$` 而不是 `$\emptyset$`；其他的符号应参照 数学符号表 书写。

我们可以使用一个表格来总结一下上述内容。注意本表格没有举出所有符号的用法，只给出常见的错误。类似的情况类比即可。

不符合规定的用法	渲染效果	符合规定的用法	渲染效果
`$log，ln，lg$`	log, ln, lg	`$\log$，$\ln$，$\lg$`	$\log$，$\ln$，$\lg$
`$sin，cos，tan$`	sin, cos, tan	`$\sin$，$\cos$，$\tan$`	$\sin$，$\cos$，$\tan$
`$gcd，lcm$`	gcd, lcm	`$\gcd$，$\operatorname{lcm}$`	$\gcd$，lcm
`$e$`，`$\text{e}$`，`e`（自然对数的底）	e，e，e	`$\mathrm{e}$`	e
`$i$`，`$\text{i}$`，`i`（虚数单位）	i，i，i	`$\mathrm{i}$`	i
`$ 小于 a 的质数 $`	a	`小于 $a$ 的质数`	小于 a 的质数
`$...$`	$\cdots$	`$\cdots$`，`$\ldots$`，`$\vdots$`，`$\ddots$`	$\cdots$，$\ldots$，$\vdots$，$\ddots$
`$a*b$`（两个数相乘）	$a*b$	`$a\times b$`，`$a\cdot b$`	$a\times b$，$a\cdot b$
`$SPFA$`（英文名称）	$SPFA$	`SPFA`	SPFA
`$a==b$`	$a==b$	`$a=b$`	$a=b$
`$f[i][j][k]$`	$f[i][j][k]$	`$f_{i,j,k}$`，`$f(i,j,k)$`	$f_{i,j,k}$，$f(i,j,k)$
`$R,N^*$`（集合）	R, N^*	`$\mathbf{R}$`，`$\mathbf{N}^*$`	$\mathbf{R}$，$\mathbf{N}^*$
`$\emptyset$`	$\emptyset$	`$\varnothing$`	$\varnothing$
`$size$`	$size$	`$\textit{size}$`	*size*

对数学公式的附加格式要求

请注意，尽管上述输入公式的语法和真正的 LaTeX 排版系统非常相似，但 **MathJax 和 LaTeX 是两个完全没有关系的东西**，MathJax 仅仅使用了一部分与 LaTeX 非常相似的语法而已。实际上，二者之间有不少细节差别，而这些差别经常导致写出来的公式在二者之间不通用。

由于 **OI Wiki** 使用 LaTeX 排版引擎开发了 PDF 导出工具，因此有必要强调公式在 MathJax 和 LaTeX 之间的兼容性。**请各位在 Wiki 中书写数学公式时注意以下几点。**

这些规则已经向 MathJax 做了尽可能多的妥协。导出工具兼容了一部分原本仅能在 MathJax 中正常输出的写法。

- 请使用 `\begin{aligned} ... \end{aligned}` 表示多行对齐的公式；
- 如果这些多行对齐的公式需要**编号**，请用 `align` 或 `equation` 环境；
- 不要使用 `split`、`eqnarray` 环境；
- 不要使用 `\lt`,`\gt` 来表示大于号和小于号，请直接使用 `<`，`>`；
- 不要直接用 `\\` 换行（需要换行的公式，请套在 `aligned` 或其他多行环境下）；
- 若要输出 LaTeX 符号 $\LaTeX$，请用 `$\rm{\LaTeX}$`，而不是 `mathrm`；（`\LaTeX` 在 TeX 排版系统中是一个不能用于数学模式下的命令，而 `\mathrm` 又不能在普通模式下使用；另外，`\text` 命令虽然在 TeX 上正常输出，但是在 MathJax 中 `\text` 命令的参数会被原样输出，而不是按命令转义）；
- 数学公式中的中文文字**必须置于 `\text{}` 命令之中**，而变量、数字、运算符、函数名称则必须置于 `\text{}` 命令之外。**请不要在 `\text{}` 命令中嵌套数学公式**；
- 使用 `array` 环境时请注意**实际列数与对齐符号的数量保持一致**。例如下面的公式中，数据实际有 3 列（`&` 是列分隔符），因此需要 3 个对齐符号（`l`/`r`/`c` 分别表示左、右、居中对齐）。

```
$$
\begin{array}{lll}
F_1=\{\frac{0}{1},&&\frac{1}{1}\}\\
F_2=\{\frac{0}{1},&\frac{1}{2},&\frac{1}{1}\}\\
\end{array}
$$
```

伪代码格式

伪码具体格式没有严格要求，请参考算法导论或学术论文。注意不要写成 Python。

Wiki 内使用 LaTeX 书写伪码，整体处于 array 环境中，缩进使用 `$\qquad$`，文字描述使用 `$\text$`，关键字使用 `$\textbf$`，赋值使用 `$\gets$`。

参考示例：

$$
\begin{array}{ll}
1 & \textbf{Input. } \text{The edges of the graph } e , \text{ where each element in } e \text{ is } (u, v, w) \\
 & \text{ denoting that there is an edge between } u \text{ and } v \text{ weighted } w . \\
2 & \textbf{Output. } \text{The edges of the MST of the input graph}.\\
3 & \textbf{Method. } \\
4 & result \gets \varnothing \\
5 & \text{sort } e \text{ into nondecreasing order by weight } w \\
6 & \textbf{for} \text{ each } (u, v, w) \text{ in the sorted } e \\
7 & \qquad \textbf{if } u \text{ and } v \text{ are not connected in the union-find set } \\
8 & \qquad\qquad \text{connect } u \text{ and } v \text{ in the union-find set} \\
9 & \qquad\qquad result \gets result\;\bigcup\ \{(u, v, w)\} \\
10 & \textbf{return } result
\end{array}
$$

```
$$
\begin{array}{ll}
1 &  \textbf{Input. } \text{The edges of the graph } e , \text{ where each element in } e \text{ is } (u, v, w) \\
  &  \text{ denoting that there is an edge between } u \text{ and } v \text{ weighted } w . \\
2 &  \textbf{Output. } \text{The edges of the MST of the input graph}.\\
3 &  \textbf{Method. } \\
4 &  result \gets \varnothing \\
```

```
5 &  \text{sort } e \text{ into nondecreasing order by weight } w \\
6 &  \textbf{for} \text{ each } (u, v, w) \text{ in the sorted } e \\
7 &  \qquad \textbf{if } u \text{ and } v \text{ are not connected in the union-find set } \\
8 &  \qquad\qquad \text{connect } u \text{ and } v \text{ in the union-find set}
 \\
9 &  \qquad\qquad  result \gets result\;\bigcup\ \{(u, v, w)\} \\
10 &  \textbf{return }  result
\end{array}
$$
```

代码块的格式要求

代码块目前分为两种：片段和例题。

关于片段代码：

- 片段的代码内容请直接在 Markdown 文档中修改。

关于例题代码：

- 例题代码的表示形式为 `--8<-- "path"`，代码均存储在 `path` 中。路径通常为 `docs/ 主题 /code/ 内容 / 内容 _ 编号 .cpp`。
- 修改例题代码时，请保证你的代码是正确的。例题代码均拥有一组测试数据，存储在 `/docs/ 主题 /examples/ 内容 / 内容 _ 编号 .in/ans` 中。

如果你需要添加例题：

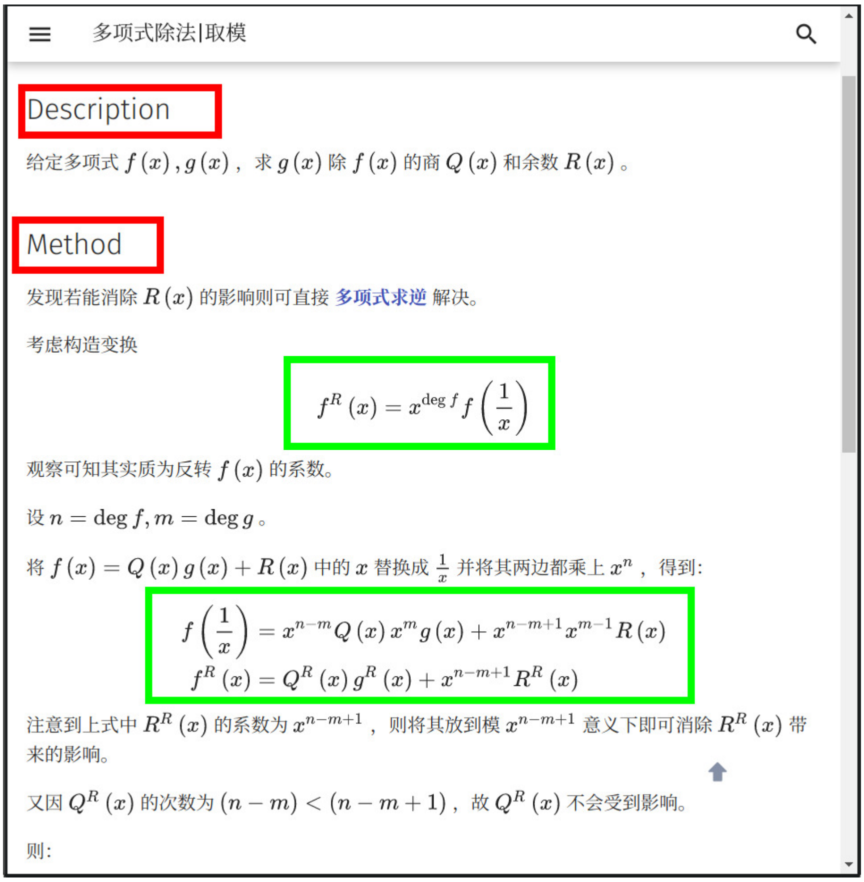
- 请在 `docs/ 主题 /code/ 内容`中添加你的例题代码，并编号。通常，该内容文件夹中已经有了一个或者多个代码。例子：如果需要修改 `dag.md` 的代码，那么路径为 `docs/dp/code/dag`，其中 `dp` 为主题，而 `dag` 为内容。
- 如果需要在所有例题的最后添加一个例题代码，请顺延目前的编号。比如已经存在了 `code/prefix-sum/prefix-sum_3.cpp`，如果需要在最后一个例题后继续添加一个例题，请将你的代码命名为 `prefix-sum_4.cpp` 并添加到 `docs/basic/code/prefix-sum` 中。
- 如果需要在文章中间添加一个例题代码，请插入并改变原先的编号。比如已经存在了 `prefix-sum_2.cpp` 和 `prefix-sum_3.cpp`，如果你需要在第二个例题和第三个例题中间再添加一个例题，请将你的代码命名为 `prefix-sum_3.cpp` 并将原先的 `prefix-sum_3.cpp` 改名为 `prefix-sum_4.cpp` 同时**在 Markdown 文档和测试数据存放的文件夹中同步修改编号**。
- **别忘记，你还要对你的代码添加一组测试数据，以保证这个代码是可以成功运行的**。你需要在 `docs/ 主题 /examples/ 内容`文件夹中添加一组测试数据，将输入数据存储为`内容 _ 编号 .in`，将标准答案存储为`内容 _ 编号 .ans`。
- 最后，可以将代码添加到文档中了。请直接在文档中用添加代码块的格式，并将代码块内部直接写成 `--8<-- " 你的代码路径 "` 的格式就可以了。

图解

可能上述要求把握起来有些困难，接下来我们给出一些图片来具体分析哪种格式应该使用，哪种不该使用：

例 1

将复杂的 LaTeX 公式使用行间格式，可以使得页面错落有致。但 **OI Wiki** 作为一个以中文为主体的站点，我们希望大部分纲领性的信息（如标题）尽量使用中文（除英文专有名词）。



多项式除法|取模

Description

给定多项式 $f(x), g(x)$，求 $g(x)$ 除 $f(x)$ 的商 $Q(x)$ 和余数 $R(x)$。

Method

发现若能消除 $R(x)$ 的影响则可直接 多项式求逆 解决。

考虑构造变换

$$f^R(x) = x^{\deg f} f\left(\frac{1}{x}\right)$$

观察可知其实质为反转 $f(x)$ 的系数。

设 $n = \deg f, m = \deg g$。

将 $f(x) = Q(x) g(x) + R(x)$ 中的 x 替换成 $\frac{1}{x}$ 并将其两边都乘上 x^n，得到：

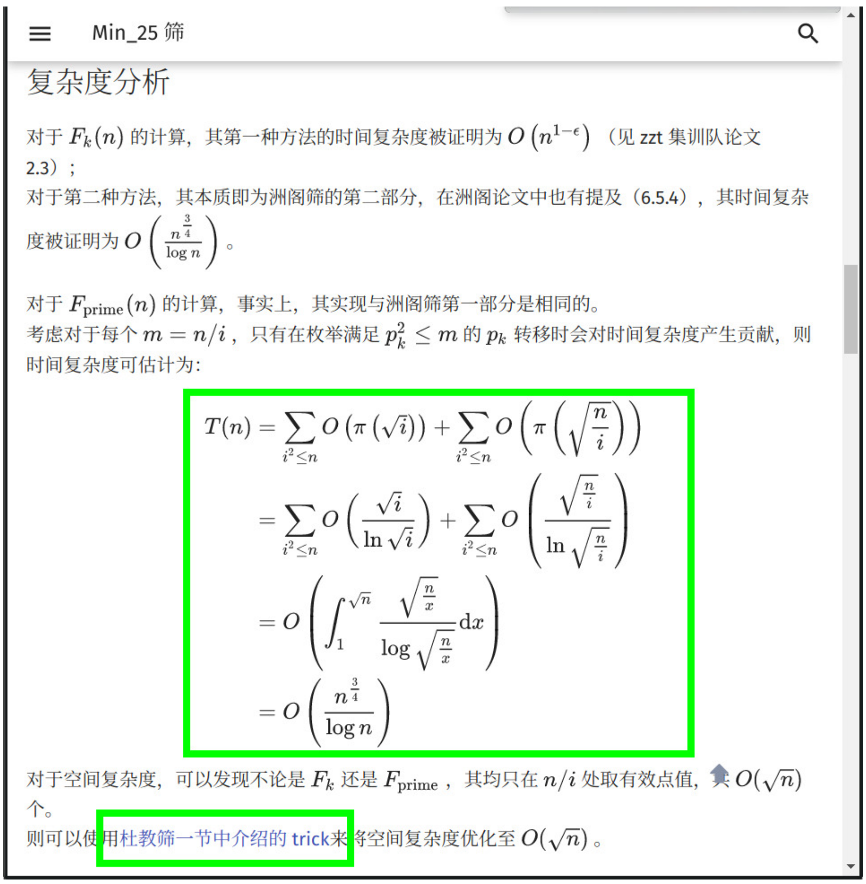
$$\begin{aligned} f\left(\frac{1}{x}\right) &= x^{n-m} Q(x) x^m g(x) + x^{n-m+1} x^{m-1} R(x) \\ f^R(x) &= Q^R(x) g^R(x) + x^{n-m+1} R^R(x) \end{aligned}$$

注意到上式中 $R^R(x)$ 的系数为 x^{n-m+1}，则将其放到模 x^{n-m+1} 意义下即可消除 $R^R(x)$ 带来的影响。

又因 $Q^R(x)$ 的次数为 $(n-m) < (n-m+1)$，故 $Q^R(x)$ 不会受到影响。

则：

图 1.6



Min_25 筛

复杂度分析

对于 $F_k(n)$ 的计算，其第一种方法的时间复杂度被证明为 $O\left(n^{1-\epsilon}\right)$（见 zzt 集训队论文 2.3）；
对于第二种方法，其本质即为洲阁筛的第二部分，在洲阁论文中也有提及（6.5.4），其时间复杂度被证明为 $O\left(\frac{n^{\frac{3}{4}}}{\log n}\right)$。

对于 $F_{\text{prime}}(n)$ 的计算，事实上，其实现与洲阁筛第一部分是相同的。
考虑对于每个 $m = n/i$，只有在枚举满足 $p_k^2 \le m$ 的 p_k 转移时会对时间复杂度产生贡献，则时间复杂度可估计为：

$$\begin{aligned} T(n) &= \sum_{i^2 \le n} O\left(\pi\left(\sqrt{i}\right)\right) + \sum_{i^2 \le n} O\left(\pi\left(\sqrt{\frac{n}{i}}\right)\right) \\ &= \sum_{i^2 \le n} O\left(\frac{\sqrt{i}}{\ln \sqrt{i}}\right) + \sum_{i^2 \le n} O\left(\frac{\sqrt{\frac{n}{i}}}{\ln \sqrt{\frac{n}{i}}}\right) \\ &= O\left(\int_1^{\sqrt{n}} \frac{\sqrt{\frac{n}{x}}}{\log \sqrt{\frac{n}{x}}} \mathrm{d}x\right) \\ &= O\left(\frac{n^{\frac{3}{4}}}{\log n}\right) \end{aligned}$$

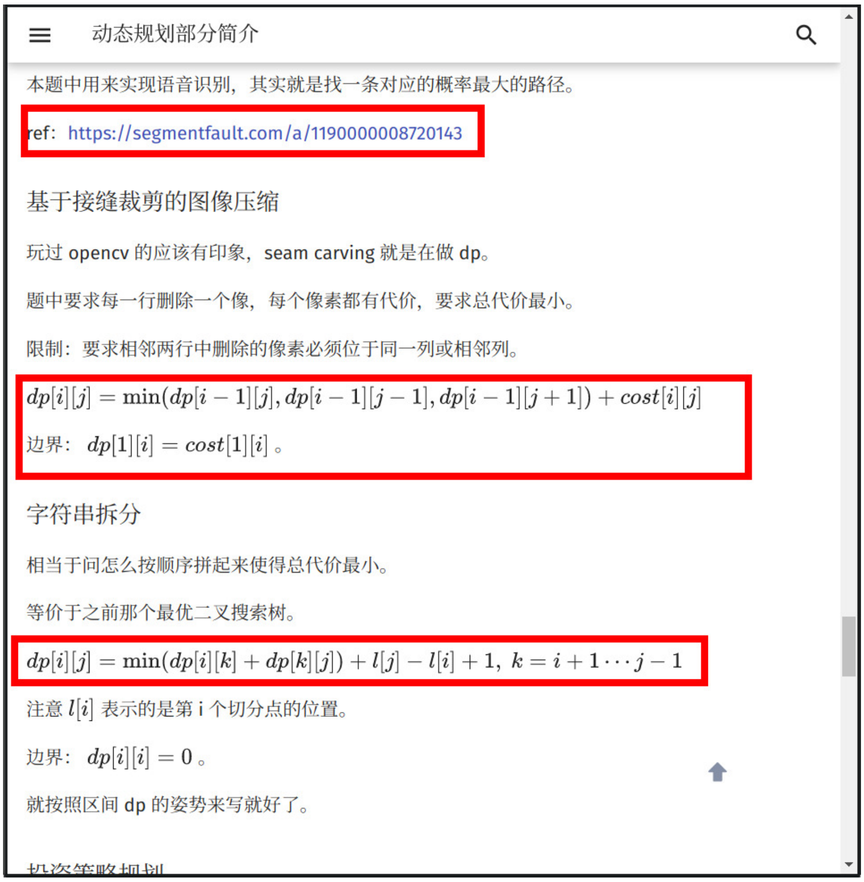
对于空间复杂度，可以发现不论是 F_k 还是 F_{prime}，其均只在 n/i 处取有效点值，共 $O(\sqrt{n})$ 个。
则可以使用杜教筛一节中介绍的 trick 来将空间复杂度优化至 $O(\sqrt{n})$。

图 1.7

例 2

较复杂度的 LaTeX 公式请注意等号的对齐，同时可以适当引用 Wiki 的页面**链接**来完善内容。

例 3



动态规划部分简介

本题中用来实现语音识别，其实就是找一条对应的概率最大的路径。

ref: https://segmentfault.com/a/1190000008720143

基于接缝裁剪的图像压缩

玩过 opencv 的应该有印象，seam carving 就是在做 dp。

题中要求每一行删除一个像，每个像素都有代价，要求总代价最小。

限制：要求相邻两行中删除的像素必须位于同一列或相邻列。

$dp[i][j] = \min(dp[i-1][j], dp[i-1][j-1], dp[i-1][j+1]) + cost[i][j]$

边界：$dp[1][i] = cost[1][i]$ 。

字符串拆分

相当于问怎么按顺序拼起来使得总代价最小。

等价于之前那个最优二叉搜索树。

$dp[i][j] = \min(dp[i][k] + dp[k][j]) + l[j] - l[i] + 1,\ k = i+1 \cdots j-1$

注意 $l[i]$ 表示的是第 i 个切分点的位置。

边界：$dp[i][i] = 0$ 。

就按照区间 dp 的姿势来写就好了。

图 1.8

一般情况下，我们建议将引用的资料列在文末的 `##` 参考资料与注释一节，并在原句后面加上脚注，而不是直接给出链接。同时一定要避免使用 LaTeX 公式表达代码，上图中两个中括号就是不规范的写法。我们建议使用 `dp(i,j)` 或者 `dp_{i,j}`。

例 4

注意我们描述**乘法**的时候一般使用 `\times` 或者 `\cdot`，特殊情况（如卷积）下会使用 `*`（也可以写成 `\ast`）。标题是简洁的词组，但我们不希望正文部分由词组拼凑而成。上图中「两个要素」，建议更改为「动态规划的原理具有以下两个要素」，上下文保持连贯。可取的地方是，适当使用**有序**列表可以更有条理地表述内容。再次提醒，在使用列表的时候，每一项如果是一句话，需要在末位添加**标点符号**。有序列表通常添加分号，在最后一项末位添加句号；无序列表统一添加句号。

例 5

适当引用**图片**可以增强文章易读性。使用**伪代码**的方式表达算法过程可以方便又简洁地描述算法过程，相比于直接贴模板代码更加好懂。

动态规划部分简介

矩阵链乘法

给出 n 个矩阵的序列，希望计算他们的乘积，问最少需要多少次乘法运算？

（认为 $p \times q$ 的矩阵与 $q \times r$ 的矩阵相乘代价是 $p \times q \times r$。）

完全括号化方案是指要给出谁先和谁乘。

动态规划原理

两个要素：

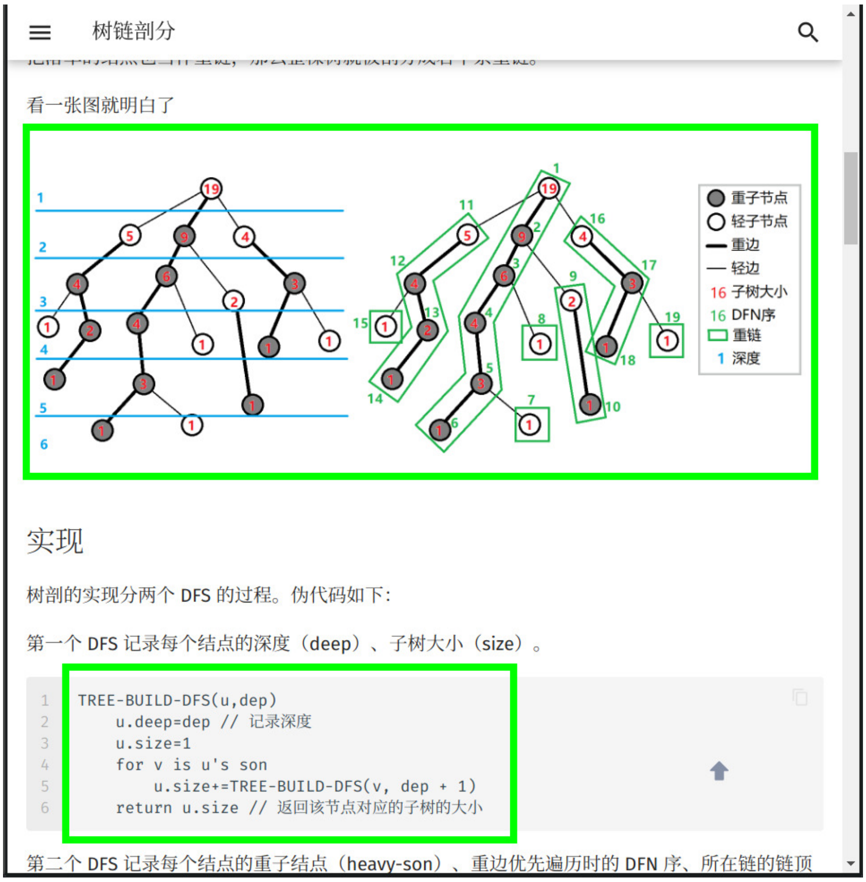
最优子结构

具有最优子结构也可能是适合用贪心的方法求解。

注意要确保我们考察了最优解中用到的所有子问题。

1. 证明问题最优解的第一个组成部分是做出一个选择；
2. 对于一个给定问题，在其可能的第一步选择中，你界定已经知道哪种选择才会得到最优解。你现在并不关心这种选择具体是如何得到的，只是假定已经知道了这种选择；
3. 给定可获得的最优解的选择后，确定这次选择会产生哪些子问题，以及如何最好地刻画子问题空间；
4. 证明作为构成原问题最优解的组成部分，每个子问题的解就是它本身的最优解。方法是反证法，考虑加入某个子问题的解不是其自身的最优解，那么就可以从原问题的解中用该子问题

图 1.9



树链剖分

看一张图就明白了

实现

树剖的实现分两个 DFS 的过程。伪代码如下：

第一个 DFS 记录每个结点的深度（deep）、子树大小（size）。

```
TREE-BUILD-DFS(u,dep)
    u.deep=dep // 记录深度
    u.size=1
    for v is u's son
        u.size+=TREE-BUILD-DFS(v, dep + 1)
    return u.size // 返回该节点对应的子树的大小
```

第二个 DFS 记录每个结点的重子结点（heavy-son）、重边优先遍历时的 DFN 序、所在链的链顶

图 1.10

图 1.11

例 6

同样的问题，标题使用英文。并且在使用完括号后没有句号。另外，上图中的行间公式虽然没有使用艾弗森括号，但是由于下标嵌套过多，使得最底层的下标字体很小，整个公式也并不美观。建议将 `son_{now,i}` 更换为 `son(now,i)`，或者把 `f_{now}` 替换为 `f(now)`。我们希望尽量控制下标嵌套在两层以内（上标的运用主要是数学表达式，因此可以允许多次嵌套，如 $2^{2^{2^{\cdots}}}$，《上帝造题的七分钟》）。

例 7

使用 MkDocs 扩展语法，让例题题面与算法描述区分开。将代码折叠，可以让文章更紧凑。（毕竟看 Wiki 的大多数是了解思路，除了模板代码需要阅读外，习题的代码大多可以折叠。）在描述函数操作时，使用行内代码和 LaTeX 公式都是不错的选择。

例 8

在文末罗列出参考文献，可以使页面的内容更严谨，真实可信。

外部链接

- 标点符号用法（GB/T 15834—2011）[24]
- 维基百科： 格式手册 / 标点符号[25]
- 中文文案排版指北（ 简体中文版 ）[26]
- 中文文案风格指南 - PDFE GUIDELINE[27]
- 一份（ 不太 ） 简短的 LATEX2 介绍或 106 分钟了解 LATEX2 [28]

Link Cut Tree

信息的修改和统计转化为平衡树上的操作，这使得 LCT 在维护树链信息上具有优势。此外，借助 LCT 实现的在树链上二分比树链剖分少一个 $O(\log_2 n)$ 的复杂度。

例题luogu P1501[国家集训队]Tree II

给出一棵有 n 个结点的树，每个点的初始权值为 1 。q 次操作，每次操作均为以下四种之一：

1. `- u1 v1 u2 v2` ：将树上 u_1, v_1 两点之间的边删除，连接 u_2, v_2 两点，保证操作合法且连边后仍是一棵树。
2. `+ u v c` ：将树上 u, v 两点之间的路径上的点权都增加 c 。
3. `* u v c` ：将树上 u, v 两点之间的路径上的点权都乘以 c 。
4. `/ u v` ：输出树上 u, v 两点之间的路径上的点权之和对 51061 取模后的值。

$1 \le n, q \le 10^5, 0 \le c \le 10^4$

`-` 操作可以直接 `Cut(u1,v1),Link(u2,v2)` 。

对树上 u, v 两点之间的路径进行修改时，先 `Split(u,v)` 。

此题要求进行在辅助树上的子树加，子树乘，子树求和操作，所以我们除了一般 LCT 需要维护的子树翻转标记，还要维护子树加法标记和子树乘法标记。处理标记的方法和在 Splay 上是一样的。

在打上和下传加法标记时，子树权值和的变化量和子树中的结点数有关，所以我们还要维护子树的大小 `siz` 。

在下传标记时，需要注意顺序，先下传乘法标记再下传加法标记。子树翻转和子树加乘两种标记没有冲突。

参考代码

图 1.12

并查集

「NOI2001」食物链

「NOI2002」银河英雄传说

其他应用

最小生成树算法中的 Kruskal 是基于并查集的算法。

References

- [1]Tarjan, R. E., & Van Leeuwen, J. (1984). Worst-case analysis of set union algorithms. Journal of the ACM (JACM), 31(2), 245-281.ResearchGate PDF
- [2]Yao, A. C. (1985). On the expected performance of path compression algorithms.SIAM Journal on Computing, 14(1), 129-133.
- [3]知乎回答：是否在并查集中真的有二分路径压缩优化？

本页面最近更新：2019/7/24 02:06:49，更新历史
发现错误？想一起完善？ 在 GitHub 上编辑此页！
本页面贡献者：Ir1d, TrisolarisHD, stevebraveman, HeRaNO, JuicyMio, Xeonacid, sailordiary, sshwy, abc1763613206, symkube, Chrogeek
©本页面的全部内容在 CC BY-SA 4.0 和 SATA 协议之条款下提供，附加条款亦可能应用

评论

图 1.13

- 中文出版物夹用英文的编辑规范[14]

参考资料与注释

[1] （冒号）表示总结上文。

[2] 科学技术名称的英文全称与其缩略形式间，应使用英文逗号。中文句子内夹用了用以注释、补充或说明的英文句子或语段，该英文句子或语段用中文圆括号标示。

[3] cstdio stdio.h namespace

[4] CCF 关于恢复 NOIP 竞赛的公告 - 中国计算机学会

[5] 我的公式为什么在目录里没有正常显示？ 好像双倍了

[6] SVG|MDN

[7] Save Page in Internet Archive

[8] APNG

[9] OI-wiki/OI-wiki#3422

[10] ScreenToGif

[11] Peek

[12] remark-lint

[13] .remarkrc

[14] 中文出版物夹用英文的编辑规范 [14-1] [14-2]

[15] 连接号 - 维基百科

[16] Collapsible Blocks

[17] HDOJ 的「A + B Problem」

[18] Admonition - Changing the title

[19] Admonitions - Material for MkDocs

[20] Content tabs

[21] MkDocs 使用说明

[22] 这个页面 [22-1] [22-2]

[23] KaTeX 的 Supported Functions 页面

[24] 标点符号用法（GB/T 15834—2011）

[25] 维基百科：格式手册/标点符号

[26] 中文文案排版指北（简体中文版）

[27] 中文文案风格指南 - PDFE GUIDELINE

[28] 一份（不太）简短的 LATEX2 介绍或 106 分钟了解 LATEX2

1.6 数学符号表

本文规定了 **OI Wiki** 中数学符号的推荐写法，并给出了一些应用范例。

本文参考了 GB/T 3102.11-1993[4] 和 ISO 80000-2:2019[5] 修订，故基本与国内通行教材的符号体系兼容。

符号的 LaTeX 写法请参考 本文章的源代码[6]

数理逻辑

编号	符号，表达式	意义，等同表述	备注与示例
n1.1	$p \wedge q$	p 和 q 的合取	p 与 q.
n1.2	$p \vee q$	p 和 q 的析取	p 或 q; 此处的” 或” 是包含的，即若 p，q 中有一个为真陈述，则 $p \vee q$ 为真。
n1.3	$\neg p$	p 的否定	非 p.
n1.4	$p \implies q$	p 蕴含 q; 若 p 为真，则 q 为真	$q \impliedby p$ 和 $p \implies q$ 同义。
n1.5	$p \iff q$	p 等价于 q	$(p \implies q) \wedge (q \implies p)$ 和 $p \iff q$ 同义。
n1.6	$(\forall\ x \in A)\ \ p(x)$	对 A 中所有的 x, 命题 $p(x)$ 均为真	如果从上下文中可以得知考虑的是哪个集合 A, 可以使用记号 $(\forall\ x)\ \ p(x)$.$\forall$ 称为全称量词。$x \in A$ 的含义见 n2.1.
n1.7	$(\exists\ x \in A)\ \ p(x)$	存在一个属于 A 的 x 使得 $p(x)$ 为真	如果从上下文中可以得知考虑的是哪个集合 A, 可以使用记号 $(\exists\ x)\ \ p(x)$.$\exists$ 称为存在量词。$x \in A$ 的含义见 n2.1.$(\exists!\ x)\ \ p(x)$（唯一量词）用来表示恰有一个 x 使得 $p(x)$ 为真。$\exists!$ 也可以写作 $\exists^1$.

集合论

编号	符号，表达式	意义，等同表述	备注与示例
n2.1	$x \in A$	x 属于 A，x 是集合 A 中的元素	$A \ni x$ 和 $x \in A$ 同义。
n2.2	$y \notin A$	y 不属于 A，y 不是集合 A 中的元素	

编号	符号，表达式	意义，等同表述	备注与示例
n2.3	$\{x_1, x_2, \dots, x_n\}$	含元素 $x_1, x_2, \dots, x_n$ 的集合	也可写作 $\{x_i \mid i \in I\}$, 其中 I 表示指标集。
n2.4	$\{x \in A \mid p(x)\}$	A 中使命题 $p(x)$ 为真的所有元素组成的集合	例如 $\{x \in \mathbf{R} \mid x \geq 5\}$; 如果从上下文中可以得知考虑的是哪个集合 A，可以使用符号 $\{x \mid p(x)\}$（如在只考虑实数集时可使用 $\{x \mid x \geq 5\}$）| 也可以使用冒号替代，如 $\{x \in A : p(x)\}$.
n2.5	$\operatorname{card} A; \|A\|$	A 中的元素个数，A 的基数	
n2.6	$\emptyset$	空集	不应使用 $\emptyset$.
n2.7	$B \subseteq A$	B 包含于 A 中，B 是 A 的子集	B 的每个元素都属于 A.$\subset$ 也可用于该含义，但请参阅 n2.8 的说明。$A \supseteq B$ 和 $B \subseteq A$ 同义。
n2.8	$B \subset A$	B 真包含于 A 中，B 是 A 的真子集	B 的每个元素都属于 A, 且 A 中至少有一个元素不属于 B. 若 $\subset$ 的含义取 n2.7, 则 n2.8 对应的符号应使用 $\subsetneq$.$A \supset B$ 与 $B \subset A$ 同义。
n2.9	$A \cup B$	A 和 B 的并集	$A \cup B := \{x \mid x \in A \vee x \in B\}$;$:=$ 的定义参见 n4.3
n2.10	$A \cap B$	A 和 B 的交集	$A \cap B := \{x \mid x \in A \wedge x \in B\}$;$:=$ 的定义参见 n4.3
n2.11	$\bigcup\limits_{i=1}^{n} A_i$	集合 $A_1, A_2, \dots, A_n$ 的并集	$\bigcup\limits_{i=1}^{n} A_i = A_1 \cup A_2 \cup \dots \cup A_n$; 也可使用 $\bigcup_{i=1}^{n}$, $\bigcup\limits_{i \in I}$, $\bigcup_{i \in I}$, 其中 I 表示指标集
n2.12	$\bigcap\limits_{i=1}^{n} A_i$	集合 $A_1, A_2, \dots, A_n$ 的交集	$\bigcap\limits_{i=1}^{n} A_i = A_1 \cap A_2 \cap \dots \cap A_n$; 也可使用 $\bigcap_{i=1}^{n}$, $\bigcap\limits_{i \in I}$, $\bigcap_{i \in I}$, 其中 I 表示指标集
n2.13	$A \ \ B$	A 和 B 的差集	$A \ \ B = \{x \mid x \in A \wedge x \notin B\}$; 不应使用 $A - B$; 当 B 是 A 的子集时也可使用 $\complement_A B$, 如果从上下文中可以得知考虑的是哪个集合 A，则 A 可以省略。不引起歧义的情况下也可使用 $\overline{B}$ 表示集合 B 的补集。
n2.14	(a, b)	有序数对 a，b; 有序偶 a，b	$(a, b) = (c, d)$ 当且仅当 $a = c$ 且 $b = d$.
n2.15	$(a_1, a_2, \dots, a_n)$	有序 n 元组	参见 n2.14.
n2.16	$A \times B$	集合 A 和 B 的笛卡尔积	$A \times B = \{(x, y) \mid x \in A \wedge y \in B\}$.
n2.17	$\prod\limits_{i=1}^{n} A_i$	集合 $A_1, A_2, \dots, A_n$ 的笛卡尔积	$\prod\limits_{i=1}^{n} A_i = \{(x_1, x_2, \dots, x_n) \mid x_1 \in A_1, x_2 \in A_2, \dots, x_n \in A_n\}$;$A \times A \times \dots \times A$ 记为 A^n, 其中 n 是乘积中的因子数。

编号	符号，表达式	意义，等同表述	备注与示例
n2.18	id_A	$A \times A$ 的对角集	$\mathrm{id}_A = \{(x, x) \mid x \in A\}$; 如果从上下文中可以得知考虑的是哪个集合 A, 则 A 可以省略。

标准数集和区间

编号	符号，表达式	意义，等同表述	备注与示例
n3.1	$\mathbf{N}$	自然数集	$\mathbf{N} = \{0, 1, 2, 3, \dots\}$;$\mathbf{N}^* = \mathbf{N}_+ = \{1, 2, 3, \dots\}$; 可用如下方式添加其他限制：$\mathbf{N}_{>5} = \{n \in \mathbf{N} \mid n > 5\}$; 也可使用 $\mathbb{N}$.
n3.2	$\mathbf{Z}$	整数集	$\mathbf{Z}^* = \mathbf{Z}_+ = \{n \in \mathbf{Z} \mid n \neq 0\}$; 可用如下方式添加其他限制：$\mathbf{Z}_{>-3} = \{n \in \mathbf{Z} \mid n > -3\}$; 也可使用 $\mathbb{Z}$.
n3.3	$\mathbf{Q}$	有理数集	$\mathbf{Q}^* = \mathbf{Q}_+ = \{r \in \mathbf{Q} \mid r \neq 0\}$; 可用如下方式添加其他限制：$\mathbf{Q}_{<0} = \{r \in \mathbf{Q} \mid r < 0\}$; 也可使用 $\mathbb{Q}$.
n3.4	$\mathbf{R}$	实数集	$\mathbf{R}^* = \mathbf{R}_+ = \{x \in \mathbf{R} \mid x \neq 0\}$; 可用如下方式添加其他限制：$\mathbf{R}_{>0} = \{x \in \mathbf{R} \mid x > 0\}$; 也可使用 $\mathbb{R}$.
n3.5	$\mathbf{C}$	复数集	$\mathbf{C}^* = \mathbf{C}_+ = \{z \in \mathbf{C} \mid z \neq 0\}$; 也可使用 $\mathbb{C}$.
n3.6	$\mathbf{P}$	（正）素数集	$\mathbf{P} = \{2, 3, 5, 7, 11, 13, 17, \dots\}$; 也可使用 $\mathbb{P}$.
n3.7	$[a, b]$	a 到 b 的闭区间	$[a, b] = \{x \in \mathbf{R} \mid a \leq x \leq b\}$.
n3.8	$(a, b]$	a 到 b 的左开右闭区间	$(a, b] = \{x \in \mathbf{R} \mid a < x \leq b\}$;$(-\infty, b] = \{x \in \mathbf{R} \mid x \leq b\}$.
n3.9	$[a, b)$	a 到 b 的左闭右开区间	$[a, b) = \{x \in \mathbf{R} \mid a \leq x < b\}$;$[a, +\infty) = \{x \in \mathbf{R} \mid a \leq x\}$.
n3.10	(a, b)	a 到 b 的开区间	$(a, b) = \{x \in \mathbf{R} \mid a < x < b\}$;$(-\infty, b) = \{x \in \mathbf{R} \mid x < b\}$;$(a, +\infty) = \{x \in \mathbf{R} \mid a < x\}$.

关系

编号	符号，表达式	意义，等同表述	备注与示例
n4.1	$a = b$	a 等于 b	$\equiv$ 用于强调某等式是恒等式该符号的另一个含义参见 n4.18.
n4.2	$a \neq b$	a 不等于 b	
n4.3	$a := b$	a 定义为 b	参见 n2.9, n2.10
n4.4	$a \approx b$	a 约等于 b	不排除相等。
n4.5	$a \simeq b$	a 渐进等于 b	例如：当 $x \to a$ 时，$\dfrac{1}{\sin(x-a)} \simeq \dfrac{1}{x-a}$;$x \to a$ 的含义参见 n4.15.
n4.6	$a \propto b$	a 与 b 成正比	也可使用 $a \sim b$.$\sim$ 也用于表示等价关系。
n4.7	$M \cong N$	M 与 N 全等	当 M 和 N 是点集（几何图形）时。该符号也用于表示代数结构的同构。
n4.8	$a < b$	a 小于 b	

编号	符号，表达式	意义，等同表述	备注与示例
n4.9	$b > a$	b 大于 a	
n4.10	$a \leq b$	a 小于等于 b	
n4.11	$b \geq a$	b 大于等于 a	
n4.12	$a \ll b$	a 远小于 b	
n4.13	$b \gg a$	b 远大于 a	
n4.14	∞	无穷大	该符号**不**是数字。也可以使用 $+\infty$，$-\infty$.
n4.15	$x \to a$	x 趋近于 a	一般出现在极限表达式中。a 也可以为 ∞，$+\infty$，$-\infty$.
n4.16	$m \mid n$	m 整除 n	对整数 m，n:$(\exists\ k \in \mathbf{Z})\quad m \cdot k = n.$
n4.17	$m \perp n$	m 与 n 互质	对整数 m，n:$(\nexists\ k \in \mathbf{Z}_{>1})\quad (k \mid m) \land (k \mid n)$; 该符号的另一种用法参见 n5.2
n4.18	$n \equiv k \pmod{m}$	n 模 m 与 k 同余	对整数 n，k，m:$m \mid (n-k)$; 不要与 n4.1 中提到的相混淆。

初等几何学

编号	符号，表达式	意义，等同表述	备注与示例
n5.1	$\parallel$	平行	
n5.2	$\perp$	垂直	该符号的另一种用法参见 n4.17
n5.3	$\angle$	（平面）角	
n5.4	$\overline{\mathrm{AB}}$	线段 AB	
n5.5	$\overrightarrow{\mathrm{AB}}$	有向线段 AB	
n5.6	$d(\mathrm{A}, \mathrm{B})$	点 A 和 B 之间的距离	即 $\overline{\mathrm{AB}}$ 的长度。

运算符

编号	符号，表达式	意义，等同表述	备注与示例
n6.1	$a + b$	a 加 b	
n6.2	$a - b$	a 减 b	
n6.3	$a \pm b$	a 加或减 b	
n6.4	$a \mp b$	a 减或加 b	$-(a \pm b) = -a \mp b.$
n6.5	$a \cdot b$;$a \times b$;ab	a 乘 b	若出现小数点，则应只使用 $\times$; 部分用例参见 n2.16, n2.17, n14.11, n14.12

编号	符号，表达式	意义，等同表述	备注与示例
n6.6	$\frac{a}{b}$;a/b;$a:b$	a 除以 b	$\frac{a}{b}=a\cdot b^{-1}$; 可用 : 表示同一量纲的数值的比率。不应使用 ÷.
n6.7	$\sum_{i=1}^{n} a_i$	$a_1+a_2+\cdots+a_n$	也可使用 $\sum\nolimits_{i=1}^{n} a_i$，$\sum_{i} a_i$，$\sum\nolimits_{i} a_i$，$\sum a_i$.
n6.8	$\prod_{i=1}^{n} a_i$	$a_1\cdot a_2\cdot\cdots\cdot a_n$	也可使用 $\prod\nolimits_{i=1}^{n} a_i$，$\prod_{i} a_i$，$\prod\nolimits_{i} a_i$，$\prod a_i$.
n6.9	a^p	a 的 p 次幂	
n6.10	$a^{1/2}$;$\sqrt{a}$	a 的 $1/2$ 次方，a 的平方根	应避免使用 $\surd a$.
n6.11	$a^{1/n}$; $\sqrt[n]{a}$	a 的 $1/n$ 次幂，a 的 n 次根	应避免使用 $\sqrt[n]{}a$.
n6.12	$\bar{x}$;$\bar{x}_a$	x 的算数均值	其他均值有：调和均值 $\bar{x}_h$; 几何均值 $\bar{x}_g$; 二次均值/均方根 $\bar{x}_q$ 或 $\bar{x}_{rms}$.$\bar{x}$ 也用于表示复数 x 的共轭，参见 n11.6.
n6.13	$\operatorname{sgn} a$	a 的符号函数	对实数 a:$\operatorname{sgn} a=1\quad(a>0)$;$\operatorname{sgn} a=-1\quad(a<0)$;$\operatorname{sgn} 0=0$; 参见 n11.7.
n6.14	$\inf M$	M 的下确界	小于等于非空集合 M 中元素的最大上界。
n6.15	$\sup M$	M 的上确界	大于等于非空集合 M 中元素的最小下界。
n6.16	$\|a\|$	a 的绝对值	也可使用 $\operatorname{abs} a$.
n6.17	$\lfloor a\rfloor$	向下取整小于等于实数 a 的最大整数	例如：$\lfloor 2.4\rfloor=2$;$\lfloor -2.4\rfloor=-3$.
n6.18	$\lceil a\rceil$	向上取整大于等于实数 a 的最小整数	例如：$\lceil 2.4\rceil=3$;$\lceil -2.4\rceil=-2$.
n6.19	$\min(a,b)$;$\min\{a,b\}$	a 和 b 的最小值	可推广到有限集中。要表示无限集中的最小值建议使用 inf, 参见 n6.14
n6.20	$\max(a,b)$;$\max\{a,b\}$	a 和 b 的最大值	可推广到有限集中。要表示无限集中的最大值建议使用 sup, 参见 n6.15
n6.21	$n \bmod m$	n 模 m 的余数	对正整数 n, m:$(\exists\, q\in\mathbf{N}, r\in[0,m))\quad n=qm+r$; 其中 $r=n \bmod m$.
n6.22	$\gcd(a,b)$;$\gcd\{a,b\}$	整数 a 和 b 的最大公因数	可推广到有限集中。不引起歧义的情况下可写为 (a,b).
n6.23	$\operatorname{lcm}(a,b)$;$\operatorname{lcm}\{a,b\}$	整数 a 和 b 的最小公倍数	可推广到有限集中。不引起歧义的情况下可写为 $[a,b]$;$(a,b)[a,b]=\|ab\|$.

组合数学

本节中的 n 和 k 是自然数，a 是复数，且 $k\le n$.

编号	符号，表达式	意义，等同表述	备注与示例
n7.1	$n!$	阶乘	$n! = \prod_{k=1}^{n} k = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n \quad (n > 0); 0! = 1.$
n7.2	$a^{\underline{k}}$	下降阶乘幂	$a^{\underline{k}} = a \cdot (a-1) \cdot \cdots \cdot (a-k+1) \quad (k > 0); a^{\underline{0}} = 1; n^{\underline{k}} = \dfrac{n!}{(n-k)!}.$
n7.3	$a^{\overline{k}}$	上升阶乘幂	$a^{\overline{k}} = a \cdot (a+1) \cdot \cdots \cdot (a+k-1) \quad (k > 0); a^{\overline{0}} = 1; n^{\overline{k}} = \dfrac{(n+k-1)!}{(n-1)!}.$
n7.4	$\dbinom{n}{k}$	组合数	$\dbinom{n}{k} = \dfrac{n!}{k!(n-k)!}.$
n7.5	$\left[{n \atop k}\right]$	第一类 Stirling 数	$\left[{n+1 \atop k}\right] = n\left[{n \atop k}\right] + \left[{n \atop k-1}\right]; x^{\overline{n}} = \sum_{k=0}^{n} \left[{n \atop k}\right] x^k.$
n7.6	$\left\{{n \atop k}\right\}$	第二类 Stirling 数	$\left\{{n \atop k}\right\} = \dfrac{1}{k!} \sum_{i=0}^{k} (-1)^i \dbinom{k}{i} (k-i)^n; \sum_{k=0}^{n} \left\{{n \atop k}\right\} x^{\underline{k}} = x^n.$

函数

编号	符号，表达式	意义，等同表述	备注与示例
n8.1	f	函数	
n8.2	$f(x)$，$f(x_1, \dots, x_n)$	函数 f 在 x 处的值函数 f 在 $(x_1, \dots, x_n)$ 处的值	
n8.3	$\operatorname{dom} f$	f 的定义域	也可使用 $\mathrm{D}(f)$.
n8.4	$\operatorname{ran} f$	f 的值域	也可使用 $\mathrm{R}(f)$.
n8.5	$f : A \to B$	f 是 A 到 B 的映射	$\operatorname{dom} f = A$ 且 $(\forall\ x \in \operatorname{dom} f) \quad f(x) \in B.$
n8.6	$x \mapsto T(x), x \in A$	将所有 $x \in A$ 映射到 $T(x)$ 的函数	$T(x)$ 仅用于定义，用来表示某个参数为 $x \in A$ 的某个函数值。若这个函数为 f, 则对所有 $x \in A$ 均有 $f(x) = T(x)$. 因此 $T(x)$ 通常用来定义函数 f. 例如：$x \mapsto 3x^2y, x \in [0, 2]$; 这是由 $3x^2y$ 定义的一个关于 x 的二次函数。若未引入函数符号，则用 $3x^2y$ 表示该函数
n8.7	f^{-1}	f 的反函数	函数 f 的反函数 f^{-1} 有定义当且仅当 f 是单射。若 f 是单射，则 $\operatorname{dom}(f^{-1}) = \operatorname{ran} f$, $\operatorname{ran}(f^{-1}) = \operatorname{dom} f$, 且 $(\forall\ x \in \operatorname{dom} f) \quad f^{-1}(f(x)) = x$. 不要与函数的倒数 $f(x)^{-1}$ 混淆。
n8.8	$g \circ f$	f 和 g 的复合函数	$(g \circ f)(x) = g(f(x)).$
n8.9	$f : x \mapsto y$	$f(x) = y$，f 将 x 映射到 y	

编号	符号，表达式	意义，等同表述	备注与示例
n8.10	$f\|_a^b$;$f(\dots,u,\dots)\|_{u=a}^{u=b}$	$f(b)-f(a)$;$f(\dots,b,\dots)-f(\dots,a,\dots)$	主要用于定积分的计算中。
n8.11	$\lim\limits_{x\to a} f(x)$;$\lim_{x\to a} f(x)$	当 x 趋近于 a 时 $f(x)$ 的极限	$\lim_{x\to a} f(x)=b$ 可以写成 $f(x)\to b \quad (x\to a)$. 右极限和左极限的符号分别为 $\lim_{x\to a+} f(x)$ 和 $\lim_{x\to a-} f(x)$.
n8.12	$f(x)=O(g(x))$	$\|f(x)/g(x)\|$ 在上下文隐含的限制中有上界，$f(x)$ 的阶不高于 $g(x)$	当 f/g 与 g/f 均有界时称 f 与 g 是同阶的。使用符号"=" 是出于历史原因，其在此处不表示等价，因为不满足传递性。例如：$\sin x = O(x) \quad (x\to 0)$.
n8.13	$f(x)=o(g(x))$	在上下文隐含的限制中有 $f(x)/g(x)\to 0$，$f(x)$ 的阶高于 $g(x)$	使用符号"=" 是出于历史原因，其在此处不表示等价，因为不满足传递性。例如：$\cos x = 1+o(x) \quad (x\to 0)$.
n8.14	Δf	f 的有限增量	上下文隐含的两函数值的差分。例如：$\Delta x = x_2 - x_1$;$\Delta f(x) = f(x_2)-f(x_1)$.
n8.15	$\frac{\mathrm{d}f}{\mathrm{d}x}$;$f'$	f 对 x 的导（函）数	仅用于一元函数。可以显式指明自变量，如 $\frac{\mathrm{d}f(x)}{\mathrm{d}x}$，$f'(x)$.
n8.16	$\left(\frac{\mathrm{d}f}{\mathrm{d}x}\right)_{x=a}$;$f'(a)$	f 在 a 处的导（函）数值	参见 n8.15
n8.17	$\frac{\mathrm{d}^n f}{\mathrm{d}x^n}$;$f^{(n)}$	f 对 x 的 n 阶导（函）数	仅用于一元函数。可以显式指明自变量，如 $\frac{\mathrm{d}^n f(x)}{\mathrm{d}x^n}$，$f^{(n)}(x)$. 可用 f'' 和 f''' 分别表示 $f^{(2)}$ 和 $f^{(3)}$.
n8.18	$\frac{\partial f}{\partial x}$;$f_x$	f 对 x 的偏导数	仅用于多元函数。可以显式指明自变量，如 $\frac{\partial f(x,y,\dots)}{\partial x}$，$f_x(x,y,\dots)$. 可以扩展到高阶，如 $f_{xx}=\frac{\partial^2 f}{\partial x^2}=\frac{\partial}{\partial x}\left(\frac{\partial f}{\partial x}\right)$;$f_{xy}=\frac{\partial^2 f}{\partial y\partial x}=\frac{\partial}{\partial y}\left(\frac{\partial f}{\partial x}\right)$.
n8.19	$\frac{\partial(f_1,\dots,f_m)}{\partial(x_1,\dots,x_n)}$	Jacobi 矩阵	参见[1]
n8.20	$\mathrm{d}f$	f 的全微分	$\mathrm{d}f(x,y,\dots)=\frac{\partial f}{\partial x}\mathrm{d}x+\frac{\partial f}{\partial y}\mathrm{d}y+\dots$.
n8.21	δf	f 的（无穷小）变分	
n8.22	$\int f(x)\mathrm{d}x$	f 的不定积分	

编号	符号，表达式	意义，等同表述	备注与示例
n8.23	$\int\limits_a^b f(x)\mathrm{d}x$	f 从 a 到 b 的定积分	也可使用 $\int_a^b f(x)\mathrm{d}x$; 定积分还可以定义在更一般的域上。如 $\int\limits_C$，$\int\limits_S$，$\int\limits_V$，$\oint$, 分别表示在曲线 C, 曲面 S, 三维区域 V, 和闭曲线或曲面上的定积分。多重积分可写成 $\iint$，$\iiint$ 等。
n8.24	$f * g$	函数 f 和 g 的卷积	$(f*g)(x)=\int\limits_{-\infty}^{\infty} f(y)g(x-y)\mathrm{d}y.$

指数和对数函数

x 可以是复数。

编号	符号，表达式	意义，等同表述	备注与示例
n9.1	e	自然对数的底	$\mathrm{e}=\lim\limits_{n\to\infty}\left(1+\frac{1}{n}\right)^n=2.718\ 281\ 8\ldots$; 不要写成 e.
n9.2	a^x	x 的指数函数（以 a 为底）	参见 n6.9.
n9.3	e^x;$\exp x$	x 的指数函数（以 e 为底）	
n9.4	$\log_a x$	x 的以 a 为底的对数	当底数不需要指定的时候可以使用 $\log x$. 不应用 $\log x$ 替换 $\ln x$，$\lg x$，$\mathrm{lb}\, x$ 中的任意一个。
n9.5	$\ln x$	x 的自然对数	$\ln x=\log_{\mathrm{e}} x$; 参见 n9.4.
n9.6	$\lg x$	x 的常用对数	$\lg x=\log_{10} x$; 参见 n9.4.
n9.7	$\mathrm{lb}\, x$	x 的以 2 为底的对数	$\mathrm{lb}\, x=\log_2 x$; 参见 n9.4.

三角函数和双曲函数

编号	符号，表达式	意义，等同表述	备注与示例
n10.1	π	圆周率	$\pi=3.141\ 592\ 6\ldots$.
n10.2	$\sin x$	x 的正弦	$\sin x=\frac{\mathrm{e}^{\mathrm{i}x}-\mathrm{e}^{-\mathrm{i}x}}{2\mathrm{i}}$;$(\sin x)^n$，$(\cos x)^n (n\geq 2)$ 等通常写为 $\sin^n x$，$\cos^n x$ 等。
n10.3	$\cos x$	x 的余弦	$\cos x=\sin(x+\pi/2)$.
n10.4	$\tan x$	x 的正切	$\tan x=\sin x/\cos x$; 不可使用 $\mathrm{tg}\, x$.
n10.5	$\cot x$	x 的余切	$\cot x=1/\tan x$; 不可使用 $\mathrm{ctg}\, x$.
n10.6	$\sec x$	x 的正割	$\sec x=1/\cos x$.
n10.7	$\csc x$	x 的余割	$\csc x=1/\sin x$; 不可使用 $\mathrm{cosec}\, x$.

编号	符号，表达式	意义，等同表述	备注与示例
n10.8	$\arcsin x$	x 的反正弦	$y = \arcsin x \iff x = \sin y \quad (-\pi/2 \leq y \leq \pi/2)$.
n10.9	$\arccos x$	x 的反余弦	$y = \arccos x \iff x = \cos y \quad (0 \leq y \leq \pi)$.
n10.10	$\arctan x$	x 反正切	$y = \arctan x \iff x = \tan y \quad (-\pi/2 \leq y \leq \pi/2)$; 不可使用 $\mathrm{arctg}\, x$.
n10.11	$\operatorname{arccot} x$	x 反余切	$y = \operatorname{arccot} x \iff x = \cot y \quad (0 \leq y \leq \pi)$; 不可使用 $\mathrm{arcctg}\, x$.
n10.12	$\operatorname{arcsec} x$	x 反正割	$y = \operatorname{arcsec} x \iff x = \sec y \quad (0 \leq y \leq \pi, y \neq \pi/2)$.
n10.13	$\operatorname{arccsc} x$	x 的反余割	$y = \operatorname{arccsc} x \iff x = \csc y \quad (-\pi/2 \leq y \leq \pi/2, y \neq 0)$; 不可使用 $\operatorname{arccosec} x$.
n10.14	$\sinh x$	x 的双曲正弦	$\sinh x = \dfrac{\mathrm{e}^x - \mathrm{e}^{-x}}{2}$; 不可使用 $\operatorname{sh} x$.
n10.15	$\cosh x$	x 的双曲余弦	$\cosh^2 x = \sinh^2 x + 1$; 不可使用 $\operatorname{ch} x$.
n10.16	$\tanh x$	x 的双曲正切	$\tanh x = \sinh x / \cosh x$; 不可使用 $\operatorname{th} x$.
n10.17	$\coth x$	x 的双曲余切	$\coth x = 1/\tanh x$.
n10.18	$\operatorname{sech} x$	x 的双曲正割	$\operatorname{sech} x = 1/\cosh x$.
n10.19	$\operatorname{csch} x$	x 的双曲余割	$\operatorname{csch} x = 1/\sinh x$; 不可使用 $\operatorname{cosech} x$.
n10.20	$\operatorname{arsinh} x$	x 的反双曲正弦	$y = \operatorname{arsinh} x \iff x = \sinh y$; 不可使用 $\operatorname{arsh} x$.
n10.21	$\operatorname{arcosh} x$	x 的反双曲余弦	$y = \operatorname{arcosh} x \iff x = \cosh y \quad (y \geq 0)$; 不可使用 $\operatorname{arch} x$.
n10.22	$\operatorname{artanh} x$	x 的反双曲正切	$y = \operatorname{artanh} x \iff x = \tanh y$; 不可使用 $\operatorname{arth} x$.
n10.23	$\operatorname{arcoth} x$	x 的反双曲余切	$y = \operatorname{arcoth} x \iff x = \coth y \quad (y \neq 0)$.
n10.24	$\operatorname{arsech} x$	x 的反双曲正割	$y = \operatorname{arsech} x \iff x = \operatorname{sech} y \quad (y \geq 0)$.
n10.25	$\operatorname{arcsch} x$	x 的反双曲余割	$y = \operatorname{arcsch} x \iff x = \operatorname{csch} y \quad (y \geq 0)$; 不可使用 $\operatorname{arcosech} x$.

复数

编号	符号，表达式	意义，等同表述	备注与示例
n11.1	i	虚数单位	$\mathrm{i}^2 = -1$; 不可使用 i 或 `i`
n11.2	$\operatorname{Re} z$	z 的实部	参见 n11.3.
n11.3	$\operatorname{Im} z$	z 的虚部	若 $z = x + \mathrm{i}y \quad (x, y \in \mathbf{R})$, 则 $x = \operatorname{Re} z$，$y = \operatorname{Im} z$.
n11.4	$\lvert z \rvert$	z 的模	$\lvert z \rvert = \sqrt{(\operatorname{Re} z)^2 + (\operatorname{Im} z)^2}$.

编号	符号，表达式	意义，等同表述	备注与示例
n11.5	$\arg z$	z 的辐角	若 $z = re^{i\varphi}$, 其中 $r = \|z\|$ 且 $-\pi < \varphi \le \pi$, 则 $\varphi = \arg z.\mathrm{Re}\, z = r\cos\varphi$，$\mathrm{Im}\, z = r\sin\varphi$.
n11.6	$\bar{z}$;z^*	z 的复共轭	$\bar{z} = \mathrm{Re}\, z - \mathrm{i}\,\mathrm{Im}\, z$.
n11.7	$\mathrm{sgn}\, z$	z 的单位模函数	$\mathrm{sgn}\, z = z/\|z\| = \exp(\mathrm{i}\arg z) \quad (z \ne 0)$;$\mathrm{sgn}\, 0 = 0$; 参见 n6.13.

矩阵

编号	符号，表达式	意义，等同表述	备注与示例
n12.1	A; 参见[2]	$m \times n$ 型矩阵 A	$a_{ij} = (A)_{ij}$; 也可使用 $A = (a_{ij})$. 其中 m 为行数，n 为列数 $m = n$ 时称为方阵可用方括号替代圆括号。
n12.2	$A + B$	矩阵 A 和 B 的和	$(A+B)_{ij} = (A)_{ij} + (B)_{ij}$; 矩阵 A 和 B 的行数和列数必须分别相同。
n12.3	xA	标量 x 和矩阵 A 的乘积	$(xA)_{ij} = x(A)_{ij}$.
n12.4	AB	矩阵 A 和 B 的乘积	$(AB)_{ik} = \sum_j (A)_{ij}(B)_{jk}$; 矩阵 A 的列数必须等于矩阵 B 的行数。
n12.5	I;E	单位矩阵	$(I)_{ik} = \delta_{ik}$;δ_{ik} 的定义参见 n14.9.
n12.6	A^{-1}	方阵 A 的逆	$AA^{-1} = A^{-1}A = I \quad (\det A \ne 0)$.$\det A$ 的定义参见 n12.10.
n12.7	A^{T};A'	A 的转置矩阵	$(A^{\mathrm{T}})_{ik} = (A)_{ki}$.
n12.8	$\overline{A}$;A^*	A 的复共轭矩阵	$\left(\overline{A}\right)_{ik} = \overline{(A)_{ik}}$.
n12.9	A^{H};$A^\dagger$	A 的 Hermite 共轭矩阵	$A^{\mathrm{H}} = \left(\overline{A}\right)^{\mathrm{T}}$.
n12.10	$\det A$; 参见[3]	方阵 A 的行列式	也可使用 $\|A\|$.
n12.11	$\mathrm{rank}\, A$	矩阵 A 的秩	
n12.12	$\mathrm{tr}\, A$	方阵 A 的迹	$\mathrm{tr}\, A = \sum_i (A)_{ii}$.
n12.13	$\\|A\\|$	矩阵 A 的范数	满足三角不等式：若 $A + B = C$, 则 $\\|A\\| + \\|B\\| \ge \\|C\\|$.

坐标系

本节考虑三维空间中的一些坐标系。点 O 为坐标系的**原点**。任意点 P 均由从原点 O 到点 P 的**位置向量**确定。

编号	坐标	位置向量和微分	坐标名	备注
n13.1	x，y，z	$r = xe_x + ye_y + ze_z$;$\mathrm{d}r = \mathrm{d}x\ e_x + \mathrm{d}y\ e_y + \mathrm{d}z\ e_z$	笛卡尔坐标	基向量 e_x，e_y，e_z 构成右手正交系，见图 1 和图 4。基向量也可用 e_1，e_2，e_3 或 i，j，k 表示，坐标也可用 x_1，x_2，x_3 或 i，j，k 表示。

编号	坐标	位置向量和微分	坐标名	备注
n13.2	ρ，φ，z	$r = \rho\, e_\rho + z\, e_z$;$\mathrm{d}r = \mathrm{d}\rho\, e_\rho + \rho\, \mathrm{d}\varphi\, e_\varphi + \mathrm{d}z\, e_z$	柱坐标	$e_\rho(\varphi)$，$e_\varphi(\varphi)$，e_z 组成右手正交系，见图 2。若 $z = 0$, 则 ρ 和 φ 是平面上的极坐标。
n13.3	r，ϑ，φ	$r = re_r$;$\mathrm{d}r = \mathrm{d}r\ e_r + r\ \mathrm{d}\vartheta\ e_\vartheta + r\ \sin\vartheta\ \mathrm{d}\varphi\ e_\varphi$	球坐标	$e_r(\vartheta,\varphi)$，$e_\vartheta(\vartheta,\varphi)$，$e_\varphi(\varphi)$ 组成右手正交系，见图 3。

如果不使用右手坐标系（见图 4），而使用左手坐标系（见图 5），则应在之前明确强调，以免符号误用。

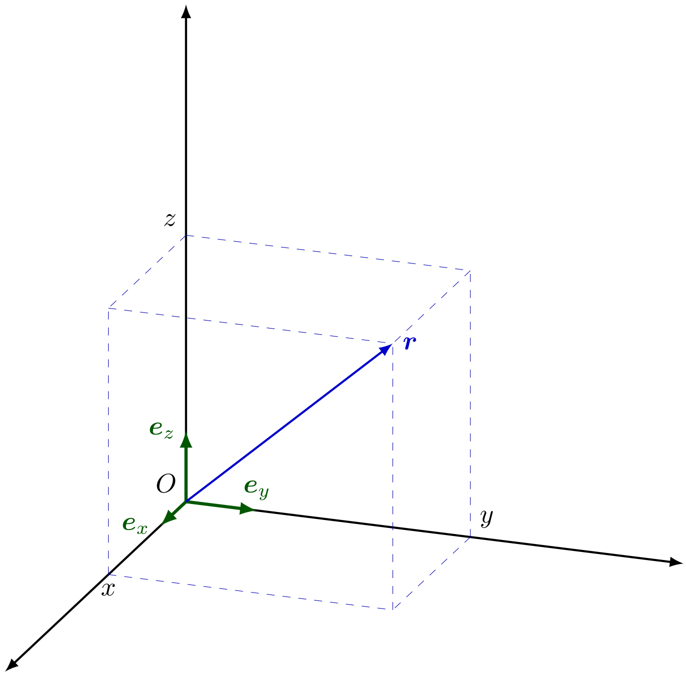


图 1.14

图 1 右手笛卡尔坐标系

图 2 右手柱坐标系

图 3 右手球坐标系

图 4 右手坐标系

图 5 左手坐标系

标量和向量

本节中，基向量用 e_1，e_2，e_3 表示。本节中的许多概念都可以推广到 n 维空间。

标量和向量本身与坐标系的选择无关，而向量的每个标量分量与坐标系的选择有关。

对于基向量 e_1，e_2，e_3, 每个向量 a 都可以表示为 $a = a_1e_1 + a_2e_2 + a_3e_3$, 其中 a_1，a_2 和 a_3 是唯一确定的标量值，将其称为向量相对于该组基向量的” 坐标”，a_1e_1，a_2e_2 和 a_3e_3 称为向量相对于该组基向量的分向量。

在本节中，只考虑普通空间的笛卡尔（正交）坐标。笛卡尔坐标用 x，y，z 或 a_1，a_2，a_3 或 x_1，x_2，x_3 表示。

本节所有下标 i，j，k 的范围均为 1 到 3.

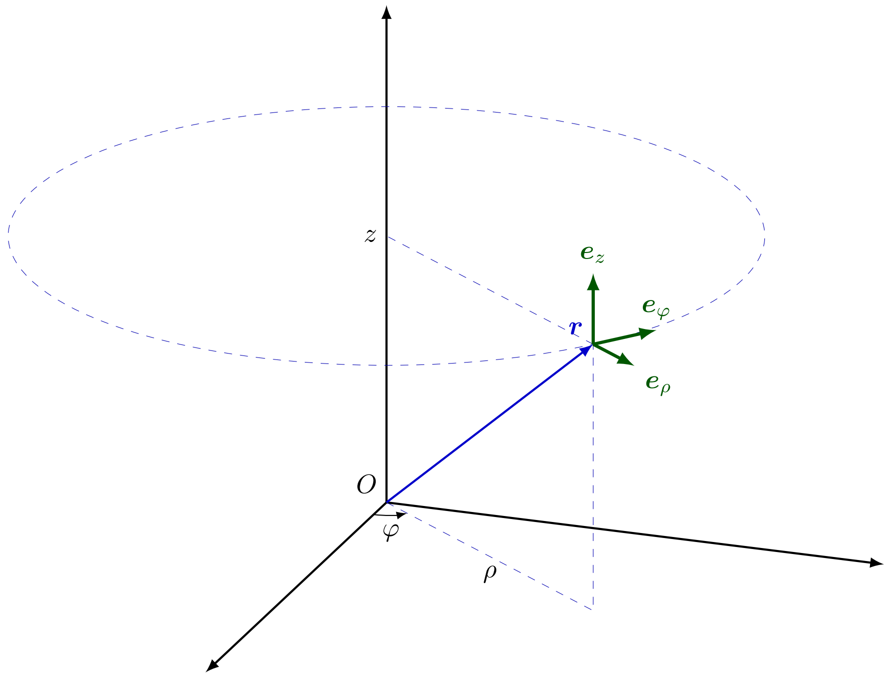

z
$\boldsymbol{e}_z$
$\boldsymbol{e}_\varphi$
$\boldsymbol{r}$
$\boldsymbol{e}_\rho$
O
φ
ρ

图 1.15

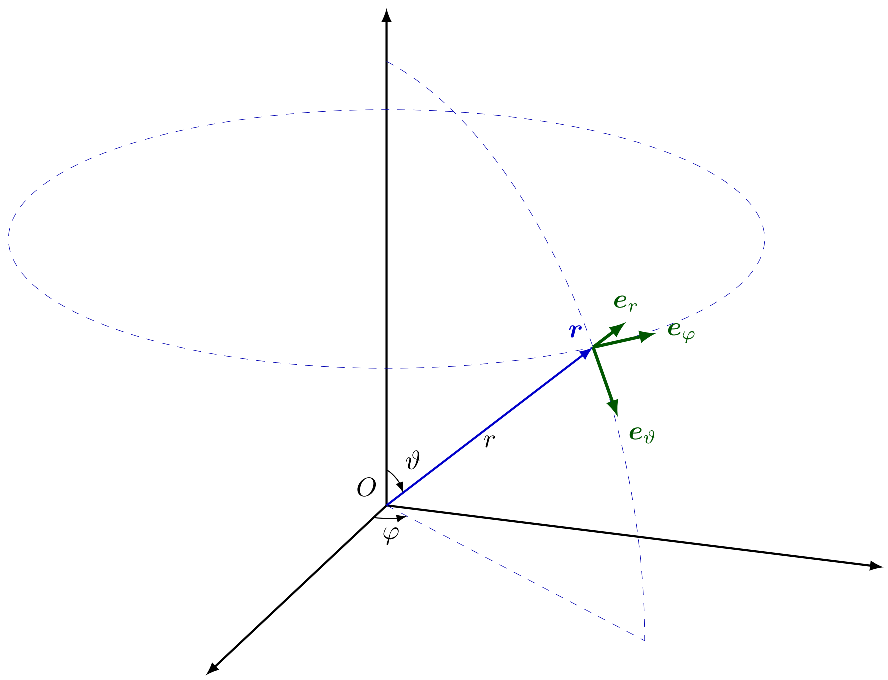

$\boldsymbol{e}_r$
$\boldsymbol{r}$
$\boldsymbol{e}_\varphi$
$\boldsymbol{e}_\vartheta$
r
ϑ
O
φ

图 1.16

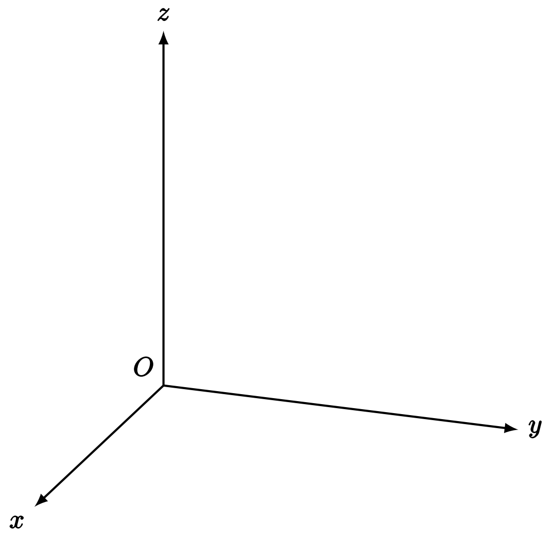


图 1.17

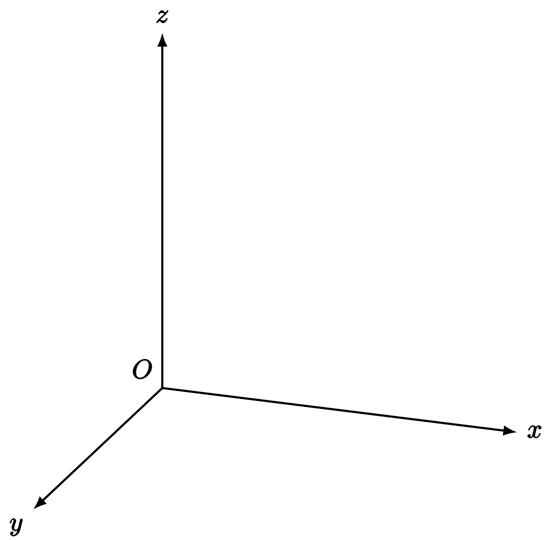


图 1.18

编号	符号，表达式	意义，等同表述	备注与示例
n14.1	a;$\vec{a}$	向量 a	
n14.2	$a+b$	向量 a 和 b 的和	$(a+b)_i = a_i + b_i$.
n14.3	xa	标量 x 与向量 a 的乘积	$(xa)_i = xa_i$.
n14.4	$\|a\|$	向量 a 的大小，向量 a 的范数	$\|a\| = \sqrt{a_x^2 + a_y^2 + a_z^2}$; 也可使用 $\\|a\\|$.
n14.5	0;$\vec{0}$	零向量	零向量的大小为 0.
n14.6	e_a	a 方向的单位向量	$e_a = a/\|a\| \quad (a \neq 0)$.
n14.7	e_x，e_y，e_z;e_1，e_2，e_3	笛卡尔坐标轴方向的单位向量	也可使用 i，j，k.
n14.8	a_x，a_y，a_z;a_i	向量 a 的笛卡尔分量	$a = a_x e_x + a_y e_y + a_z e_z$; 如果上下文确定了基向量，则向量可以写为 $a = (a_x, a_y, a_z)$.$a_x = a \cdot e_x$，$a_y = a \cdot e_y$，$a_z = a \cdot e_z$;$r = xe_x + ye_y + ze_z$ 是坐标为 x，y，z 的位置向量。
n14.9	δ_{ik}	Kronecker delta 符号	$\delta_{ik} = 1 \quad (i = k)$;$\delta_{ik} = 0 \quad (i \neq k)$.
n14.10	ε_{ijk}	Levi-Civita 符号	$\varepsilon_{123} = \varepsilon_{231} = \varepsilon_{312} = 1$;$\varepsilon_{132} = \varepsilon_{321} = \varepsilon_{213} = -1$; 其余的 ε_{ijk} 均为 0.
n14.11	$a \cdot b$	向量 a 和 b 的标量积/内积	$a \cdot b = \sum_i a_i b_i$.
n14.12	$a \times b$	向量 a 和 b 的向量积/外积	右手笛卡尔坐标系中，$(a \times b)_i = \sum_j \sum_k \varepsilon_{ijk} a_j b_k$;$\varepsilon_{ijk}$ 的定义参见 n14.10.
n14.13	⊠	nabla 算子	$⊠ = e_x \frac{\partial}{\partial x} + e_y \frac{\partial}{\partial y} + e_z \frac{\partial}{\partial z} = \sum_i e_i \frac{\partial}{\partial x_i}$.
n14.14	⊠φ;$\mathbf{grad}\,\varphi$	φ 的梯度	⊠$\varphi = \sum_i e_i \frac{\partial \varphi}{\partial x_i}$;**grad** 应使用 `\operatorname{\mathbf{grad}}`.
n14.15	⊠$\cdot a$;$\mathbf{div}\,a$	a 的散度	⊠$\cdot a = \sum_i \frac{\partial a_i}{\partial x_i}$;**div** 应使用 `\operatorname{\mathbf{div}}`.
n14.16	⊠$\times a$;$\mathbf{rot}\,a$	a 的旋度	(⊠$\times a)_i = \sum_j \sum_k \varepsilon_{ijk} \frac{\partial a_k}{\partial x_j}$;**rot** 应使用 `\operatorname{\mathbf{rot}}`. 不应使用 **curl**.ε_{ijk} 的定义参见 n14.10.

编号	符号，表达式	意义，等同表述	备注与示例
n14.17	$\boxtimes^2;\Delta$	Laplace 算子	$\boxtimes^2=\dfrac{\partial^2}{\partial x^2}+\dfrac{\partial^2}{\partial y^2}+\dfrac{\partial^2}{\partial z^2}.$

特殊函数

本节中的 z，w 是复数，k，n 是自然数，且 $k\leq n$。

编号	符号，表达式	意义，等同表述	备注与示例
n15.1	γ	Euler–Mascheroni 常数	$\gamma=\lim\limits_{n\to\infty}\left(\sum\limits_{k=1}^{n}\dfrac{1}{k}-\ln n\right)=0.577\ 215\ 6\dots.$
n15.2	$\Gamma(z)$	gamma 函数	$\Gamma(z)=\int\limits_0^\infty t^{z-1}\mathrm{e}^{-t}\mathrm{d}t\quad(\mathrm{Re}\,z>0);\Gamma(n+1)=n!.$
n15.3	$\zeta(z)$	Riemann zeta 函数	$\zeta(z)=\sum\limits_{n=1}^{\infty}\dfrac{1}{n^z}\quad(\mathrm{Re}\,z>1).$
n15.4	$\mathrm{B}(z,w)$	beta 函数	$\mathrm{B}(z,w)=\int\limits_0^1 t^{z-1}(1-t)^{w-1}\mathrm{d}t\quad(\mathrm{Re}\,z>0，\mathrm{Re}\,w>0);\mathrm{B}(z,w)=\dfrac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)};\dfrac{1}{(n+1)\,\mathrm{B}(k+1,n-k+1)}=\dbinom{n}{k}.$

[1] $\dfrac{\partial(f_1,\dots,f_m)}{\partial(x_1,\dots,x_n)}=\begin{pmatrix}\dfrac{\partial f_1}{\partial x_1} & \cdots & \dfrac{\partial f_1}{\partial x_n}\\ \vdots & \ddots & \vdots\\ \dfrac{\partial f_m}{\partial x_1} & \cdots & \dfrac{\partial f_m}{\partial x_n}\end{pmatrix}$；矩阵的定义参见 n12.1

[2] $\begin{pmatrix}a_{11} & \cdots & a_{1n}\\ \vdots & \ddots & \vdots\\ a_{m1} & \cdots & a_{mn}\end{pmatrix}$

[3] $\begin{vmatrix}a_{11} & \cdots & a_{1n}\\ \vdots & & \vdots\\ a_{n1} & \cdots & a_{nn}\end{vmatrix}$

[4] GB/T 3102.11-1993

[5] ISO 80000-2:2019

[6] 本文章的源代码

1.7 F.A.Q.

本页面主要解答一些常见的问题。

我想问点与这个 Wiki 相关的问题

Q：你们是为什么想要做这个 Wiki 的呢？

A：不知道你在学 **OI** 的时候，面对庞大的知识体系，有没有感到过迷茫无助的时候？**OI Wiki** 想要做的事情可能类似于「让更多竞赛资源不充裕的同学能方便地接触到训练资源」。当然这么表述也不完全，做 Wiki 的动机可能也很纯粹，只是简单地想要对 **OI** 的发展做出一点点微小的贡献吧。XD

Q：我很感兴趣，怎么参与？

A：**OI Wiki** 现在托管在 GitHub 上，你可以直接访问这个 repo[6] 来查看最新进展。参与的途径包括在 GitHub 上面开 Issue[7]、Pull Request[8]，或者在交流群中分享你的想法、直接向管理员投稿。目前，我们使用的框架是用 Python 开发的 MkDocs[9]，支持 Markdown 格式（也支持插入数学公式）。

Q：可是我比较弱……不知道我能做点什么。

A：一切源于热爱。你可以协助其他人审核修改稿件，帮助我们宣传 **OI Wiki**，为社区营造良好学习交流氛围！

Q：现在主要是谁在做这件事啊？感觉这是个大坑，真的能做好吗？

A：最开始主要是一些退役老年选手在做这件事，后来遇到了很多志同道合的小伙伴：有现役选手，退役玩家，也有从未参加过 **OI** 的朋友。目前，这个项目主要是由 **OI Wiki** 项目组来维护（下面是一张合影）。

当然，这个项目只靠我们的力量是很难做得十全十美的，我们诚挚地邀请你一起来完善 **OI Wiki**。

Q：你们怎么保证我们添加的内容不会突然消失？

A：我们把内容托管在 GitHub[6] 上面，即使我们的服务器翻车了，内容也不会丢失。另外，我们也会定期备份大家的心血，即使有一天 GitHub 倒闭了（?），我们的内容也不会丢失。

Q：**OI Wiki** 好像有空的页面啊！

A：是的。受限于项目组成员的水平和时间，我们暂时无法完成这些空页面。所以我们在这里进行征稿和招募，希望可以遇到有同样想法的朋友，我们一起把 **OI Wiki** 完善起来。

Q：为什么不直接去写 中文维基百科[10] 呢？

A：因为我们希望可以真正帮到更多的选手或者对这些内容感兴趣的人。而且由于众所周知的原因，中文维基上的内容并不是无门槛就可以获取到的。

我想参与进来！

Q：我要怎么与项目组交流？

A：可以通过 关于本项目里的交流方式 联系我们。

Q：我要怎么贡献代码或者内容？

请参考 如何参与 页面。

Q：目录在哪？

A：目录在项目根目录下的 mkdocs.yml[11] 文件中。

Q：如何修改一个 topic 的内容？

A：在对应页面右上方有一个编辑按钮 edit，点击并确认阅读了 如何贡献 之后会跳转到 GitHub 上对应文件的位置。

或者也可以自行阅读目录 (mkdocs.yml)[12] 查找文件位置。

Q：如何添加一个 topic？

A：有两种选择：

- 可以开一个 Issue，注明希望能添加的内容。
- 可以开一个 Pull Request，在目录 (mkdocs.yml)[12] 中加上新的 topic，并在 docs[13] 文件夹下对应位置创建一个空的 `.md` 文件。文档的格式细节请参考 格式手册。

Q：我尝试访问 GitHub 的时候遇到了困难。

A：推荐在 hosts 文件中加入如下几行[1]：

```
# GitHub Start
140.82.114.25                 alive.github.com
140.82.113.5                  api.github.com
185.199.110.153               assets-cdn.github.com
185.199.111.133               avatars.githubusercontent.com
185.199.111.133               avatars0.githubusercontent.com
185.199.111.133               avatars1.githubusercontent.com
185.199.111.133               avatars2.githubusercontent.com
185.199.111.133               avatars3.githubusercontent.com
185.199.111.133               avatars4.githubusercontent.com
185.199.111.133               avatars5.githubusercontent.com
185.199.111.133               camo.githubusercontent.com
140.82.112.22                 central.github.com
185.199.111.133               cloud.githubusercontent.com
140.82.114.9                  codeload.github.com
140.82.113.22                 collector.github.com
185.199.111.133               desktop.githubusercontent.com
185.199.111.133               favicons.githubusercontent.com
140.82.112.3                  gist.github.com
52.216.163.147                github-cloud.s3.amazonaws.com
52.217.124.1                  github-com.s3.amazonaws.com
52.216.144.83                 github-production-release-asset-2e65be.s3.amazonaw
s.com
52.217.121.249                github-production-repository-file-5c1aeb.s3.amazon
aws.com
52.217.206.57                 github-production-user-asset-6210df.s3.amazonaws.c
om
192.0.66.2                    github.blog
140.82.114.4                  github.com
140.82.113.18                 github.community
185.199.110.154               github.githubassets.com
151.101.1.194                 github.global.ssl.fastly.net
185.199.110.153               github.io
185.199.111.133               github.map.fastly.net
185.199.110.153               githubstatus.com
140.82.112.25                 live.github.com
185.199.111.133               media.githubusercontent.com
185.199.111.133               objects.githubusercontent.com
13.107.42.16                  pipelines.actions.githubusercontent.com
185.199.111.133               raw.githubusercontent.com
185.199.111.133               user-images.githubusercontent.com
13.107.253.40                 vscode.dev
140.82.112.21                 education.github.com
# GitHub End
```

可以在 GitHub520[3] 上了解到最新内容和更多信息。

Linux 和 macOS 用户可以尝试使用 依云[14] 的 gh-check 脚本[15] 获取访问最快的 IP，使用 `--hosts` 参数可以直接更新 hosts 文件。使用 `--help` 参数可以获取使用帮助。使用先需要安装 Python3 和 aiohttp（`pip install aiohttp -i https://pypi.tuna.tsinghua.edu.cn/simple/`）。依云博客的介绍：寻找最快的 GitHub IP[16]。

同时，您可以使用 Gitclone[17] 服务加速 Clone，可以阅读其首页上的说明。

如果您仅仅是想 Clone **OI Wiki** 的仓库，那么：

```
git clone https://gitclone.com/github.com/OI-wiki/OI-wiki
```

如果您需要向 **OI Wiki** 贡献，那么首先 fork **OI Wiki** 的仓库，然后（将 `username` 替换为您的用户名），需要注意的是提供的示例将使您使用 SSH 连接到 GitHub[5]：

```
git clone https://gitclone.com/github.com/username/OI-wiki
git remote set-url origin git@github.com:username/OI-wiki.git
```

Q：我这里 pip 也太慢了！

A：可以选择更换国内源[2]，或者：

```
pip install -U -r requirements.txt -i https://pypi.tuna.tsinghua.edu.cn/simple/
```

Q：我在客户端 clone 了这个项目，速度太慢。

A：如果有安装 `git bash`，可以加几个限制来减少下载量。[3]

```
git clone https://github.com/OI-wiki/OI-wiki.git --depth=1 -b master
```

Q：我没装过 Python 3。

A：可以访问 Python 官网[18] 了解更多信息。

Q：好像提示我 pip 版本过低。

A：进入 cmd/shell 之后，执行以下命令：

```
python -m pip install --upgrade pip
```

Q：我安装依赖失败了。

A：检查一下：网络？权限？查看错误信息？

Q：我已经 clone 下来了，为什么部署不了？

A：检查一下是否安装好了依赖？

Q：我 clone 了很久之前的 repo，怎么更新到新版本呢？

A：请参考 GitHub 官方的帮助页面 Syncing a fork - GitHub Docs[19]。

Q：如果是装了之前的依赖怎么更新？

A：请输入以下命令：

```
pip install -U -r requirements.txt
```

Q：为什么我的 markdown 格式乱了？

A：可以查阅 cyent 的笔记[20]，或者 MkDocs 使用说明[21]。

我们目前使用 remark-lint[22] 来自动化修正格式，可能还有一些 配置[23] 不够好的地方，欢迎指出。

Q：GitHub 是不是不显示我的数学公式？

A：是的，GitHub 的预览不显示数学公式。但是请放心，MkDocs 是支持数学公式的，可以正常使用，只要是 MathJax 支持的句式都可以使用。

Q：我的数学公式怎么乱码了？

A：如果是行间公式（用的 `$$`），目前已知的问题是需要在 `$$` 两侧留有空行，且 `$$` 要**单独**放在一行里（且不要在前加空格）。格式如下：

```
// 空行
$$
a_i
$$
// 空行
```

Q：我的公式为什么在目录里没有正常显示？好像双倍了。

A：是的，这个是 python-markdown 的一个 bug，可能近期会修复。

如果想要避免目录中出现双倍公式，可以参考 string 分类下 SAM 的目录写法[24]。

```
结束位置 <script type="math/tex">endpos</script>
```

在目录中会变成

```
结束位置 endpos
```

注：现在请尽量避免在目录中引入 MathJax 公式。

Q：如何给一个页面单独声明版权信息？

A：在页面开头加一行即可。[4]

比如：

```
copyright: SATA
```

注：默认的是 CC BY-SA 4.0 和 SATA。

Q：为什么作者信息统计处没有我的名字？

A：如果你发现自己写过一个页面中的部分内容，但是你没有被记录进作者列表，可以把自己的 GitHub ID 加入到文件头的 author 字段。

感谢你看到了最后，我们现在亟需的，就是你的帮助。

OI Wiki 项目组

2018.8

参考资料与注释

[1] GitHub520

[2] 更改 pip 源至国内镜像 - L 瑜 - CSDN 博客

[3] GIT--- 看我一步步入门（Windows Git Bash）

[4] Metadata - Material for MkDocs

[5] GitHub 弃用了基于密码身份验证的 HTTPS 协议，连接必须使用 SSH 或者 Personal Access Token，参见 我应使用哪个远程 URL？，创建个人访问令牌 和 使用 SSH 连接到 GitHub。

[6] repo [6-1] [6-2]

[7] Issue

[8] Pull Request

[9] MkDocs

[10] 中文维基百科

[11] mkdocs.yml

[12] (mkdocs.yml) [12-1] [12-2]

[13] docs

[14] 依云

[15] gh-check 脚本

[16] 寻找最快的 GitHub IP

[17] Gitclone

[18] Python 官网

[19] Syncing a fork - GitHub Docs

[20] cyent 的笔记

[21] MkDocs 使用说明

[22] remark-lint

[23] 配置

[24] string 分类下 SAM 的目录写法

1.8 用 Docker 部署 OI Wiki

本页面将介绍使用 Docker 部署 **OI Wiki** 环境的方式。

> warning
>
> 以下步骤须在 root 用户下或 docker 组用户下执行。

拉取 OI Wiki 镜像

```
# 以下命令在主机中运行其中一个即可
# Docker Hub 镜像（官方镜像仓库）
docker pull 24oi/oi-wiki
# DaoCloud Hub 镜像（国内镜像仓库）
docker pull daocloud.io/sirius/oi-wiki
# Tencent Hub 镜像（国内镜像仓库）
docker pull ccr.ccs.tencentyun.com/oi-wiki/oi-wiki
```

自行构建镜像

```
# 以下命令在主机中运行
# 克隆 Git 仓库
git clone https://github.com/OI-wiki/OI-wiki.git
cd OI-wiki/
# 构建镜像
docker build -t [name][:tag] . --build-arg [variable1]=[value1] [variable2]=[val
ue2]...
```

- （必须）设置 `[name]` 以设置镜像名，（可选）设置 `[tag]` 以设置镜像标签（若设置，则运行时镜像名由两部分构成）。
- 可以通过 `--build-arg` 参数设置环境变量。

可以使用的环境变量：

- 可以设置 `WIKI_REPO` 来使用 Wiki 仓库的镜像站点（当未设置时自动使用 GitHub）
- 可以设置 `PYPI_MIRROR` 来使用 PyPI 仓库的镜像站点（当未设置时自动使用官方 PyPI）
  - 在国内建议使用 TUNA 镜像站 `https://pypi.tuna.tsinghua.edu.cn/simple/`
- 可以设置 `LISTEN_IP` 来更改监听 IP（当未设置时为 `0.0.0.0`，即监听所有 IP 的访问）
- 可以设置 `LISTEN_PORT` 来更改监听端口（当未设置时为 `8000`）

示例：

```
docker build -t OI_Wiki . --build-arg WIKI_REPO=https://hub.fastgit.xyz/OI-wiki/
OI-wiki.git PYPI_MIRROR=https://pypi.tuna.tsinghua.edu.cn/simple/
# 构建一个名为 OI_Wiki （标签默认）的镜像，使用 FastGit 服务加速克隆，使用 TUNA 镜像
站。
```

运行容器

```
# 以下命令在主机中运行
docker run -d -it [image]
```

- （必须）设置 `[image]` 以设置镜像。例如，从 Docker Hub 拉取的为 `24oi/oi-wiki`；DaoCloud Hub 拉取的则为 `daocloud.io/sirius/oi-wiki`。
- （必须）设置 `-p [port]:8000` 以映射容器端口至主机端口（不写该语句则默认为不暴露端口。设置时请替换 `[port]` 为主机端口）。设置后可以在主机使用 `http://127.0.0.1:[port]` 访问 **OI Wiki**。
- 设置 `--name [name]` 以设置容器名字。（默认空。设置时请替换 `[name]` 为自定义的容器名字。若想查看容器 id，则输入 `docker ps`）

使用容器

note

示例基于 Ubuntu latest 部署。

进入容器：

```
# 以下命令在主机中运行
docker exec -it [name] /bin/bash
```

若在上述运行容器中去掉 `-d`，则可以直接进入容器 bash，退出后容器停止，加上 `-d` 则后台运行，请手动停止。上述进入容器针对加上 `-d` 的方法运行。

特殊用法：

```
# 以下命令在容器中运行
# 更新 git 仓库
wiki-upd

# 使用我们的自定义主题
wiki-theme

# 构建 mkdocs ，会在 site 文件夹下得到静态页面
wiki-bld

# 构建 mkdocs 并渲染 MathJax ，会在 site 文件夹下得到静态页面
wiki-bld-math

# 运行一个服务器，访问容器中 http://127.0.0.1:8000 或访问主机中 http://127.0.0.1:
[port] 可以查看效果
wiki-svr

# 修正 Markdown
wiki-o
```

退出容器：

```
# 以下命令在容器中运行
# 退出
exit
```

停止容器

```
# 以下命令在主机中运行
docker stop [name]
```

启动容器

```
# 以下命令在主机中运行
docker start [name]
```

重启容器

```
# 以下命令在主机中运行
docker restart [name]
```

删除容器

```
# 以下命令在主机中运行
# 删除前请先停止容器
docker rm [name]
```

更新镜像

重新再 `pull` 一次即可，通常不会更新。

删除镜像

```
# 以下命令在主机中运行
# 删除前请先删除使用 oi-wiki 镜像构建的容器
docker rmi [image]
```

疑问

如果您有疑问，欢迎提出 issue[1]！

参考资料与注释

[1] issue

1.9 镜像站列表

OI Wiki 部署在国外服务器上，有时可能会因为各种原因，出现访问不通畅的情况。

我们搭建了一个状态页：https://status.oi-wiki.org[1]，用于监控 **OI Wiki** 站点的在线情况。如果你遇到了无法访问的问题，可以打开状态页，寻找可以连接的镜像站。

以下是一个 **OI Wiki** 的镜像站列表，可供选用：

- **OI Wiki** 主站，线路：DMIT
  - https://oi-wiki.org[2]
- 维护者：**OI Wiki**，线路：阿里云，同步频率：与主站相同
  - http://oi-wiki.com[3]
- 维护者：**OI Wiki**，线路：Netlify，同步频率：与主站相同
  - https://demo.oi-wiki.org[4]
- 维护者：琴春（vx.st[5]），线路：AWS，同步频率：与主站相同
  - https://oi-wiki.net[6]
  - https://oi-wiki.wiki[7]

    - https://oi-wiki.win[8]
    - https://oi-wiki.xyz[9]
    - https://oiwiki.moe[10]
    - https://oiwiki.net[11]
    - https://oiwiki.org[12]
    - https://oiwiki.vx.st[13]
    - https://oiwiki.wiki[14]
    - https://oiwiki.win[15]
    - https://oiwiki.com[16]
- 维护者：Menci（men.ci[17]），线路：Azure + 阿里云 CDN，同步频率：与主站相同
    - https://oi.wiki[18]

参考资料与注释

[1] https://status.oi-wiki.org

[2] https://oi-wiki.org

[3] http://oi-wiki.com

[4] https://demo.oi-wiki.org

[5] vx.st

[6] https://oi-wiki.net

[7] https://oi-wiki.wiki

[8] https://oi-wiki.win

[9] https://oi-wiki.xyz

[10] https://oiwiki.moe

[11] https://oiwiki.net

[12] https://oiwiki.org

[13] https://oiwiki.vx.st

[14] https://oiwiki.wiki

[15] https://oiwiki.win

[16] https://oiwiki.com

[17] men.ci

[18] https://oi.wiki

1.10 致谢

本项目目前接受捐赠，扫描下方二维码可以投食（请务必备注 「 捐赠 」+ 自己的信息）。

所有款项将被用于 **OI Wiki** 的域名、服务器、运维等必需支出。

大额捐赠将会记录在本页面下方或日后更合适的位置来表示感谢。

id	amount	date
匿名捐赠者	10 元	2021.6.20
匿名捐赠者	100 元	2021.6.10
wood3	256 元	2021.6.4
海外兔	102 元	2021.5.12
逐梦之人	80.7 元	2021.5.3
十寸雨 zsg	10.24 元	2021.2.27
匿名捐赠者	2 元	2020.12.10
吾有一數名之曰🄵	10.24 元	2020.10.19
Huah	66 元	2020.9.7
icedream61	66 元	2020.9.2
三鸽酸鸽可爱	2.33 元	2020.8.31
Apoi2333	23.33 元	2020.8.31
草莓熊	50 元	2020.7.4
匿名捐赠者	200 元	2020.6.29
Hiroid	30 元	2020.5.13
dcmfqw	20.48 元	2020.5.4
akira	66 元	2020.3.30
Hermione	10 元	2020.3.26
Siyuan	10 元	2020.2.28
hsh	15 元	2020.2.21
陌陌	10 元	2020.2.6
匿名捐赠者	10 元	2020.2.5

id	amount	date
Yisin	10 元	2020.2.4
GinRyan	50 元	2019.11.30
JuicyMio	10 元	2019.11.30
匿名捐赠者	10 元	2019.11.14
QQ 联系 12	5 元	2019.10.28
匿名捐赠者	5 元	2019.10.27
增肥中的小肥	50 元	2019.10.24
ianahao	10 元	2019.10.12
Sundy	10 元	2019.10.11
三鸽最可爱	23.33 元	2019.8.17
Fburan	10.24 元	2019.8.17
匿名捐赠者	30 元	2019.8.8
Billchenchina	100 元	2019.8.7
贷款捐头的匿名入土	30 元	2019.8.4
sshwy	50 元	2019.8.4
匿名捐赠者	10 元	2018.9.9
匿名捐赠者	20 元	2018.8.31
Xeonacid	30 元	2018.8.30
匿名捐赠者	240 元	2018.8.29
Anguei	5 元	2018.8.29

第 2 章

比赛相关

2.1 比赛相关简介

本章主要介绍计算机编程比赛直接相关的知识，包括各种赛事、赛制、题型，以及赛场上常见的坑点与技巧。

学习路线，与常用的学习资源也可以在本章找到。

本章亦设出题板块，介绍出竞赛题的相关知识。

2.2 赛事

2.2.1 OI 赛事与赛制

> **Authors:** Ir1d, Planet6174, abc1763613206, StudyingFather, cjsoft, Marcythm, luoguyuntianming, ChungZH, Xeonacid, YZircon, i-Yirannn, H-J-Granger, NachtgeistW, YuzhenQin, Andycode3759

赛事简介

信息学奥林匹克竞赛（英语：Olympiad in Informatics，简称：OI）是一门在中学生中广泛开展的学科竞赛，和物理、数学等竞赛性质相同。OI 考察的内容是参赛者运用算法、数据结构和数学知识，通过编写计算机程序解决实际问题的能力。

OI 竞赛种类繁多，仅中国就包括：

- 全国青少年信息学奥林匹克联赛（NOIP）
- 全国青少年信息学奥林匹克竞赛（NOI）
- 全国青少年信息学奥林匹克竞赛冬令营（WC）
- 国际信息学奥林匹克竞赛中国队选拔赛（CTSC）

国际性的 OI 竞赛包括：

- 国际信息学奥林匹克（IOI）
- 美国计算机奥林匹克竞赛（USACO）
- 日本信息学奥林匹克（JOI）
- 亚太地区信息学奥林匹克（APIO）

……

对于大部分选手而言，每年的新赛季从 9 月的 CSP-J/S 第一轮开始。

在中国，OI 竞赛允许使用的语言只有 C++（曾经也开放过 C 和 Pascal 语言，但都已停止支持）。其中，不同的竞赛对 C++ 的版本有不同的规定。考试题目一般为算法或者数据结构相关的内容，题目形式包括传统题（最常见的规定输入和输出到文件的题目）和非传统题（提交答案题、交互题、补全代码题……等等）。

赛制介绍

OI 赛制

选手仅有一次提交机会。比赛时无法看到评测结果，评分会在赛后公布。每道题都有多个测试点，根据每道题通过的测试点的数量获得相应的分数；每个测试点还可能会有部分分，即使只有部分数据通过也能拿到分数。

CSP-J/S 第二轮、NOIP、省选、NOI 都是 OI 赛制。

IOI 赛制

选手在比赛时有多次提交机会。比赛实时评测并返回结果，如果提交的结果是错误的，不会有任何惩罚。每道题都有多个测试点，根据每道题通过的测试点的数量获得相应的分数。

APIO、IOI 都是 IOI 赛制。目前国内比赛也在逐渐向 IOI 赛制靠拢。

Codeforces (CF) 赛制

Codeforces[1] 是一个在线评测系统，会定期举办比赛。

它的比赛特点是在比赛过程中只测试一部分数据（Pretests），而在比赛结束后返回完整的所有测试点的测试结果（System Tests）。比赛时可以多次提交，允许 Hack 别人的代码（此处 Hack 的意思是提交一个测试数据，使得别人的代码无法给出正确答案）。如果想要 Hack，选手必须要锁定自己的代码（换言之，比赛时无法重新提交该题）。Hack 时不允许将选手程序拷贝到本地进行测试，源代码会被转换成图片。

Codeforces 同时提供另外一种赛制，称作扩展 ICPC（Extended ICPC 或 ICPC+）。在这一赛制中，在比赛过程中会测试全部数据，但比赛结束以后会有 12 小时的全网 Hack 时间。Hack 时允许将选手程序拷贝到本地进行测试。

主要比赛

CSP-J/S

CSP-J/S（英文：Certified Software Professional Junior/Senior）是 NOIP 在 2019 年被取消之后，CCF 开设的非专业级软件能力认证测试，面向全年龄段。

CSP-J/S 分为入门级（Junior，简写为 CSP-J）与提高级（Senior，简写为 CSP-S）两组，赛程分为第一轮（一般在每年 9 月）和第二轮（一般在每年 10 月）两场。第一轮为笔试，考察计算机理论和操作常识和基本的算法与数学知识；第二轮为上机考试，入门组与提高组都为 4 题，其中入门组考试时间 3.5 个小时，提高组 4 个小时（CSP-S 2019 除外，该场比赛使用旧 NOIP 提高组赛制，赛程分为两天，一天 3 题 3.5 小时）。第一轮面向社会全体学生报名，经过一定的排名筛选后成绩优秀者有机会参加第二轮。

报名参加第一/二轮、第二轮后进行题目申诉等都需要向 CCF 缴费。

两轮测试都会以省为单位按照排名对选手成绩进行评级认证，分为一、二、三等。

NOIP

NOIP（英语：National Olympiad in Informatics in Provinces，中文：全国青少年信息学奥林匹克联赛）是中华人民共和国组织的、面向中国（含港澳）中学生的信息学竞赛。

2018 年及以前的旧赛制：NOIP 按参赛对象分为普及组和提高组，2018 年于上海试点入门组；按阶段分为初赛和复赛两个阶段。初赛会考察一些计算机基础知识和算法基础，复赛为上机考试。时间上一般是 11 月的第二个周末，周六上午提高组一试 8:30-12:00（3.5 小时，共 3 题），下午 14:30-18:00 普及组（3.5 小时，共 4 题），周日上午提高组二试 8:30-12:00（3.5 小时，共 3 题）。全国使用同一套试卷，但是评奖规则按照省内情况由 CCF（中国计算机学会）统一指定，并于赛后在 NOI 官方网站[2] 上公布。各省的一等奖分数线略有不同。

NOIP 于 2019 年 8 月 16 日 被 CCF 暂停[3]，于 2020 年 1 月 21 日 被宣布恢复[4]。2020 年起的 NOIP 赛制与以往有所不同，具体如下：

- 取消初赛，由 CSP-J/S 第一轮替代；
- 取消普及组，由 CSP-J 替代，此后 NOIP 仅有一个组别，面向提高组水平选手；
- 赛程由以往的两天共 6 题、每天 3.5 个小时，缩减为一天 4 题、共 4.5 个小时。
- 选手需要在 CSP-S 第二轮中取得一定名次才能获得 NOIP 参赛资格，具体名额各省有所差异。NOIP 省级参赛资格由该省在去年赛季中的参赛人数和成绩等有关。

报名参加 NOIP 和进行题目申诉不需要额外缴费。

NOIP 以省为单位排名评奖。截至 2019 年，大部分高校的选手获得提高组省一等奖可以得到自主招生资格。

> 2020 年 1 月，中华人民共和国教育部发布 关于在部分高校开展基础学科招生改革试点工作的意见[5]。意见指出，2020 年起，不再组织开展高校自主招生工作，并在部分一流大学建设高校开展基础学科招生改革试点（强基计划）。

省队选拔赛

省队选拔赛（简称：省选）用于选拔各省参加全国赛的代表队，一般举行于每年的 1~4 月。赛程上一般分为两天，每天 3 题 4.5 小时。

省选题目由各个省自行决定，目前的趋势是很多省份选择联合命题。

各个省队的名额有复杂的计算公式，一般和之前的成绩和参赛人数有关。通常来讲，NOIP 分数需要在省选的指标中占一定比例。根据规则，初中选手只能被选拔为 E 类选手，不能参加 A、B 类选拔。A 类选手有 5 人（4 男 1 女），其他选手根据给定名额和所得分数依次进入 B 队。一个学校参加 NOI 的名额不超过本省 A、B 名额总数的三分之一（四舍五入），得分最高且入选 A 队的女选手不占该比例（简称 1/3 限制或 1/3 淘汰）。

自 2020 年起，NOI 省队选拔由 CCF 统一命题和评测，有能力命题的省可自行命题，但选拔方式需得到 CCF 的批准。自 2024 年起，NOI 省队选拔恢复各省自主命题，有需求的省份可组织联考或使用他省试题，但具体方案需要得到 CCF 的批准。

NOI

NOI（英文：National Olympiad in Informatics，中文：全国信息学奥林匹克竞赛）是国内包括港澳在内的省级代表队最高水平的大赛。

NOI 一般在七月份举行，选手分为正式选手与夏令营选手两类。正式选手又分为三类，其中 A、B 类为省队正式选手，C 类选手为邀请赛选手。A、B 类对应省队的 A、B 类选手（其中 A 类在计算成绩时会有 5 分加分）；C 类名义上是学校对 CCF 做出突出贡献后的奖励名额。夏令营选手分为 D、E 类，分别对应以非正式选手身份参赛的高中组与初中组选手。夏令营选手如果成绩超过分数线的话，只有成绩证明而没有奖牌（同等分数含金量要低一些）。排名前 60 的正式选手组成国家集训队，获得保送资格。

在国际平台上，为了与其他同样称作 NOI 的比赛区分，有时会被称作 CNOI。

WC

WC（英文：Winter Camp，中文：全国青少年信息学奥林匹克竞赛冬令营）是每年冬天在当年 NOI 举办地进行的一项活动。

WC 的内容包括若干天的培训和一天的考试。这项考试主要用于从国家集训队（50 人）选拔国家候选队（15 人），但是前一年 NOIP 与 CSP-S 第二轮取得较好成绩的选手也可以参加（不参与选拔）。

APIO

APIO（英文：Asia-Pacific Informatics Olympiad，中文：亚太地区信息学奥林匹克竞赛）是一个面向亚太地区在校中学生的信息学学科竞赛。CCF 每年会在五月初举办中国赛区镜像赛。在比赛日前后会有培训活动。

CTS

CTS（旧称：CTSC, 英文：China Team Selection Competition，中文：国际信息学奥林匹克竞赛中国队选拔赛）用来从国家候选队（15 人）中选拔国家队（6 人）准备参加当年夏天的 IOI 比赛，其中正式选手 4 人，替补选手 2 人。与 WC 一样，前一年 NOIP 取得较好成绩的选手也可以参加（不参与选拔）。

APIO 和 CTS 都以省为单位报名，一般按照 NOIP 的成绩排序来确定参加 APIO 和 CTS 的人员（二者一般时间上非常接近）。

IOI

IOI（英文：International Olympiad in Informatics，中文：国际信息学奥林匹克竞赛）是一年一度的面向全球中学生的信息学科竞赛。每个国家有四人参赛，比赛一般会有直播。IOI 赛制中每个题目会有 Subtask（子任务），每个子任务对应一定的分数。

学科营

北京大学（PKU）

- 北京大学信息学冬季体验营（PKUWC）：在冬令营前后举行。
- 北京大学信息学体验营（PKUSC）：一般在六月份在校内举行。由于在学校机房比赛，机房环境是 Windows，比赛系统是 OpenJudge。
- 北京大学中学生暑期课堂（信息学）：在暑假举行，面向高二年级理科学生。

清华大学（THU）

- 计算机系”大中衔接”冬季研讨与教学活动：相当于信息学冬令营，有时也会用英文简写为 THUWC。一般共两天，上午为竞赛（第一天是标准 OI 竞赛，第二天为清华独创的”工程题”竞赛），下午为课程培训。

其他国家和地区的 OI 竞赛

美国：USACO

官网地址：http://www.usaco.org/[6]

USACO 或许是国内选手最熟悉的外国 OI 竞赛（可能也是中文题解最多的外国 OI 竞赛）。

每年冬季到初春，USACO 会每月举办一场网络赛。一场比赛持续 3~5 个小时。

根据官网的介绍，USACO 的比赛分成这 4 档难度（2015~2016 学年之前为 3 档）：

- 铜牌组，适合编程初学者，尤其是只学了最最基础的算法（如：排序，二分查找）的学生；
- 银牌组，适合开始学习基本的算法技巧（如：递归，搜索，贪心算法）和基础数据结构的学生；
- 金牌组，学生会遇到更复杂的算法（如：最短路径，DP）和更高级的数据结构；
- 铂金组，适合有着扎实的算法设计能力的选手，铂金组可以帮助他们以复杂且更开放的问题来挑战自我。

在国内，目前 USACO 题目最齐全的 OJ 平台是洛谷。

波兰：POI

官网地址：https://oi.edu.pl/[7]

官方提交地址：https://szkopul.edu.pl/p/default/problemset/[8]

POI 是不少省选选手最常刷的外国 OI 比赛。

根据 http://main.edu.pl/en/[9] 的描述，POI 的流程如下：

- 第一轮：五题，网络赛，公开赛；
- 第二轮：包含一场练习赛，和两场正式比赛；
- 第三轮：赛制同上。

- ONTAK：POI 训练营（类似国内的集训队）。

另有 PA，大意为「算法大战」。

目前在国内 OJ 中，POI 题目最全的是 BZOJ。

克罗地亚：COCI

官网地址（英文）：http://www.hsin.hr/coci/[10]

官网地址（克罗地亚语）：http://www.hsin.hr/honi/[11]

难度跨度很大的比赛，大约是从普及 - 到省选 -。

以往 COCI 所有的题目均提供题目、数据、题解和标程。2017 年底起，COCI 的题解和标程停止了更新。2019-2020 赛季重新开始更新题解和标程。

洛谷、BZOJ 和 LibreOJ 都有少量的 COCI 题目。

日本：JOI

官网地址：https://www.ioi-jp.org/[12]

JOI（日文：日本情報オリンピック，中文：日本信息学奥赛）所有的题目都提供题目、数据、题解和标程。近两年的 JOI 决赛和春训营提供了英语题面，但并没有英语题解。历年的 JOI Open 都提供了英语版题面和题解。

JOI 的流程：

- 预赛（予選）
- 决赛（本選/JOI Final）
- 春训营（春季トレ🄵ニング合宿/JOI Spring Camp/JOISC）
- 公开赛（通信教育/JOI Open Contest）

预赛难度较低，自 2019/2020 赛季起，预赛分为多轮。JOI Final 的难度从提高 - 到提高 + 左右。JOISC 和 JOI Open 的题目的难度从提高到 NOI - 不等。

绝大部分 JOI 题可以前往 AtCoder[13] 提交。你可以在 JOI 官网或者 AtCoder 上找到更多的 JOI 题（日文题面）。

目前 LibreOJ 和 BZOJ 有近些年的 JOI Final、JOISC 和 JOI Open 的题目。

俄罗斯：ROI

官网地址：http://neerc.ifmo.ru/school/archive/index.html[14]

在线提交地址：https://contest.yandex.ru/roiarchive/[15] 和 Codeforces（部分）。

ROI（俄文： ，中文：俄罗斯信息学奥赛）是俄罗斯的信息学竞赛。

流程：

- 市级比赛（Municipal Stage/ ）
- 州级比赛（Regional Stage/ ）
- 决赛（Final Stage/ ）

目前 LibreOJ 有近几年的 ROI 决赛题的译文。

除此之外，俄罗斯较大型的、面向中学生的比赛还有：

- 信息学网络奥赛（俄文： - ）
  - 官网地址：http://neerc.ifmo.ru/school/io/index.html[16]
  - 该比赛由 ROI 出题人举办。
- 全国中学生团队信息学竞赛（俄文： ）
  - 官网地址：http://neerc.ifmo.ru/school/russia-team/index.html[17]

  - 该比赛的预选赛 Moscow Team Olympiad 可以在 Codeforces 上提交。
- Innopolis Open
  - 官网地址 https://olymp.innopolis.ru/en/ooui/information/[18]
- 中学生编程公开赛（   ）
  - 官网地址：https://olympiads.ru/zaoch/[19]
  - 官网称该比赛对标 ROI。

加拿大：CCC & CCO

CCC（英文：Canadian Computing Competition），CCO（英文：Canadian Computing Olympiad），可在其 官网[20] 查询历届的信息和试题等。

在 DMOJ 上可以提交 CCC[21] 和 CCO[22]，该 OJ 上还有 CCC 题解。

CCC Junior/Senior 贴近 NOIP 普及组/提高组难度。CCO 想要拿到金牌可能得有 NOI 银牌的水平。

新加坡：NOI SG

官网地址：https://noisg.comp.nus.edu.sg/noi/[23]

全称 Singapore National Olympiad in Informatics，在新加坡国内语境且不引起歧义的情况下也作 NOI。赛制上分为 Online Qualification Contest（在线资格赛）和 Final Contest（全国决赛）。在线资格赛以学校为单位报名参加，选手在本校参赛，通过网络进行远程提交。资格赛成绩只在校内排名，前 5 名且非零分选手有资格作为校代表队参加全国决赛。

目前国内 OJ 对于 NOI SG 的题目收录比较匮乏，可以在 官方的 GitHub 帐号[24] 上找到历年题面、测试数据和官方标准程序。

台湾地区：資訊□林匹亞競賽

台湾地区把 OI 中的 informatics 翻译成「資訊」而非大陆通用的翻译「信息」。

台湾地区的选手如果想参加 IOI，需要经过这几轮比赛：

- 區域資訊學科能力競賽
- 全國資訊學科能力競賽
- 資訊研習營（TOI）

其他国家

- 法国与澳大利亚：FARIO：http://orac.amt.edu.au/cgi-bin/train/hub.pl[25]
  - 难度与 NOI 类似。
- 英国：British Informatics Olympiad：https://www.olympiad.org.uk/[26]
  - 难度太低。
- 捷克：Matematická olympiáda–kategorie P：http://mo.mff.cuni.cz/p/archiv.html[27]
- 罗马尼亚：Olimpiada Nationala de Informatica：http://olimpiada.info/[28]
  - 题面、测试数据、题解请在含有 Subiecte 字样的标签页中寻找。

其它国际 OI 竞赛

BalticOI

BalticOI 面向的是波罗的海周边各国。BalticOI 2018 的参赛国有立陶宛、波兰、爱沙尼亚、芬兰等 9 国。题目难度大。

除了 2017 年，BalticOI 每年都公开题面、测试数据和题解。BalticOI 没有一个固定的官网，每年的主办方都会新建一个网站。历年的官网地址见 帖子[29]。

目前 LibreOJ 有近十年的 BalticOI 题。

BalkanOI

BalkanOI 面向巴尔干地区周边各国。BalkanOI 2018 的参赛国有罗马尼亚、希腊、保加利亚、塞尔维亚等 12 国。题目难度大。

BalkanOI 只有某几年公开题面、测试数据和题解，官网地址见 帖子[29]。

CEOI

CEOI 2018 的参赛国与上面两个比赛有部分重叠，包括波兰、罗马尼亚、格鲁吉亚、克罗地亚等国。题目难度大。

CEOI 每年都公开题面、测试数据和题解，官网地址见 帖子[29]。

在国内 OJ 中，BZOJ 的 CEOI 题相对最齐。

eJOI

eJOI 全名 European Junior Olympiad in Informatics。参赛国包含俄罗斯、亚美尼亚、保加利亚、波兰等国。题目难度较大。

eJOI 每年都公开题面、测试数据和题解，官网地址见 帖子[29]。

ISIJ

ISIJ 全名 International School in Informatics ”Junior”，中文名「国际初中生信息学竞赛」。

官网地址：http://isi-junior.com/[30]

NOI

warning

此处介绍的不是「全国信息学奥林匹克竞赛」。

NOI 全名 Nordic Olympiads in Informatics。

官网地址：http://nordic.progolymp.se[31]

近两年才开始举办的比赛，面向北欧各国。

参考资料

- ICPC/CCPC 赛事与赛制
- 「 翻译组 」 一些大洲级 OI 比赛的地址[29]

参考资料与注释

[1] Codeforces

[2] NOI 官方网站

[3] 被 CCF 暂停

[4] 被宣布恢复

[5] 关于在部分高校开展基础学科招生改革试点工作的意见

[6] http://www.usaco.org/

[7] https://oi.edu.pl/

[8] https://szkopul.edu.pl/p/default/problemset/

[9] http://main.edu.pl/en/

[10] http://www.hsin.hr/coci/

[11] http://www.hsin.hr/honi/

[12] https://www.ioi-jp.org/

[13] AtCoder

[14] http://neerc.ifmo.ru/school/archive/index.html

[15] https://contest.yandex.ru/roiarchive/

[16] http://neerc.ifmo.ru/school/io/index.html

[17] http://neerc.ifmo.ru/school/russia-team/index.html

[18] https://olymp.innopolis.ru/en/ooui/information/

[19] https://olympiads.ru/zaoch/

[20] 官网

[21] CCC

[22] CCO

[23] https://noisg.comp.nus.edu.sg/noi/

[24] 官方的 GitHub 帐号

[25] http://orac.amt.edu.au/cgi-bin/train/hub.pl

[26] https://www.olympiad.org.uk/

[27] http://mo.mff.cuni.cz/p/archiv.html

[28] http://olimpiada.info/

[29] 帖子 [29-1] [29-2] [29-3] [29-4] [29-5]

[30] http://isi-junior.com/

[31] http://nordic.progolymp.se

2.2.2 ICPC/CCPC 赛事与赛制

Authors: NachtgeistW, Ir1d, Xeonacid, H-J-Granger, abc1763613206, YuzhenQin

赛事介绍

ICPC

ICPC（英文：International Collegiate Programming Contest，中文：国际大学生程序设计竞赛）由 ICPC 基金会（英文：ICPC Foundation）举办，是最具影响力的大学生计算机竞赛。由于以前 ACM 赞助这个竞赛，也有很多人习惯叫它 ACM 竞赛。

ICPC 主要分为区域赛（Regionals）和总决赛（World Finals）两部分。

官网地址：https://icpc.global[1]

CCPC

官网地址：https://ccpc.io[2]

中国大学生程序设计竞赛。

和 ICPC 显著的区别是很多学校是不报销的。

赛制介绍

一般是三个人组成一队使用一台机器，在比赛时有多次提交机会。比赛实时评测并返回结果，如果提交的结果错误会有 20 分钟的罚时，错误次数越多，加罚的时间也越长。每个题目只有在所有数据点全部正确后才能得到分数。比赛排名根据做题数来评判，做题数相同的，根据总用时来评判。总用时是每题用时的和。每题的用时是从比赛开始到做出该题的分钟数与该题的罚时之和。

一些 ICPC 相关赛事中，比赛结束前一小时进行封榜，封榜后的提交和排名将无法被其他选手看见。

在 ICPC 相关赛事中，选手允许带一定量的纸质资料。

除 ICPC 和 CCPC 外，众多比赛也采用该赛制，如 LeetCode 周赛及全国编程大赛、牛客小白赛练习赛挑战赛等。

赛季赛程

- ICPC/CCPC 网络赛（8 月底至 9 月初）
- ICPC/CCPC 区域赛（9 月底至 11 月底）
- ICPC EC Final/CCPC Final（12 月中旬）
- ICPC World Finals（次年 4 月至 6 月）

训练指南

多校联合训练

暑期在 HDU OJ[3] 举行的训练赛。有奖金，题目质量高，历经多年积累已有丰富资源。

OJ 里查询用的关键词：`Multi-University Training Contest`。

国内区域赛

在 Virtual Judge[4] 里可以搜到精选题集。

训练营

- 寒假的时候头条/清华/CCPC (Wannafly Camp) 举办的 Camp
- Wannafly Camp

训练资源

- **OI Wiki：https://oi-wiki.org**[5]
- Codeforces Gym：https://codeforces.com/gyms[6]

参考资料与注释

[1] https://icpc.global

[2] https://ccpc.io

[3] HDU OJ

[4] Virtual Judge

[5] https://oi-wiki.org

[6] https://codeforces.com/gyms

2.3 题型

2.3.1 题型概述

Authors: StudyingFather, NachtgeistW, countercurrent-time, Ir1d, H-J-Granger, Chrogeek, sshwy, Suyun514, hsfzLZH1, CBW2007, Xeonacid, kawa-yoiko, Konano

在算法竞赛中，有多种多样的问题类型。

传统题

传统题是目前算法竞赛中较为常见的题型。

选手需要提交源代码，评测系统会使用事先准备好一些输入数据和相应的输出数据作为测试点[1]，将选手提交的源代码编译后[2]，让选手程序读入输入数据，通过将选手输出与事先准备好的输出比较，来判断选手程序是否正确。这种评测方式被称之为**黑盒评测**[3]。

对于一个测试点，往往还会设置时间限制和空间限制。

时间限制，指的是程序运行时间的限制[4]。选手程序在一个测试点上的运行时间不能超过给定的时间限制。

空间限制，指的是程序使用的内存量的限制。选手程序在运行时占用的最大空间不能超过给定的空间限制。

在程序正常运行结束后，选手的输出会和测试点输出进行比对。这种比对一般采用过滤文末换行和行末空格之后，进行全文比对的方式。对于某些特殊的题目，会使用 Special Judge 来进行比对。

这一过程结束后，评测系统会根据程序的运行状态，给出不同的**评测结果**[5]：

- Accepted（AC）：选手程序被接受。
- Compile Error（CE）：选手程序无法正常编译。
- Wrong Answer（WA）：选手程序正常结束，但是选手程序的输出与测试点输出不符。
- Presentation Error（PE）：选手程序正常结束，但是格式不符合要求[6]。
- Runtime Error（RE）：选手程序非正常结束（选手程序结束时的返回值不为零）。
- Time Limit Exceeded（TLE）：选手程序运行的时间超过了给定的时间限制。
- Memory Limit Exceeded（MLE）：选手程序占用的最大空间超过了给定的空间限制。
- Output Limit Exceeded（OLE）：选手程序输出的内容的量超过了最大限制。

在 ICPC 赛事中，你的程序需要在一道题目的所有测试点上都取得 AC 状态，才能视为通过相应的题目。在 OI 赛事中，在一个测试点中取得 AC 状态，即可拿到该测试点的分数[7]。

提交答案题

提交答案题是直接提交答案的题目。该种题目一般会给出输入文件，要求提交包含有 `XXX1.out`、`XXX2.out`、`XXX3.out`⋯`XXXn.out` 的压缩包、文件夹或纯文件。

提交答案后，评测系统会比较答案文件与标准答案，根据选手答案的优劣情况和任务完成度，给予一定的分数。

因为提交答案题不需要运行源程序，故提交答案题不存在时间和空间限制。

做这种题目一般有两种方法：

- 手玩。这种方法简单粗暴，但是遇到较大的数据就没辙了。
- 编写一个程序来获得答案文件。

交互题

交互题是需要选手程序与测评程序交互来完成任务的题目。一类常见的情形是，选手程序向测评程序发出询问，并得到其反馈。测评程序可能对选手的询问作出限制，或调整应答策略来尽可能增加询问次数，这也给题目带来了更多变化。

更详细的交互题讲解可以看 交互题。

交互方式主要有如下两种。虽然技术上有不小的差异，但在考察算法的本质上它们并没有实际区别。

STDIO 交互

STDIO 交互（标准 I/O 交互）是 Codeforces、AtCoder 等在线平台的交互手段，也是 ICPC 系列赛事中的标准。Codeforces 提供了一个更加简要的 说明 （ 英文 ）[8]。

例题"ZQC 的迷宫"

LOJ #559.「LibreOJ Round #9」ZQC 的迷宫[9]

请注意最下方添加内容。

本题是一道交互题。

位于 $n \times m$ 个方格组成的黑暗迷宫的你，需要走到这个迷宫的终点，以完成迷宫挑战。

最开始，你位于迷宫的起点即 $(1,1)$ 处，且面向右侧，终点位于 (n,m) 处。迷宫中任意两个方格之间均连通，且仅有唯一的一条路径，两个相邻（即上、下、左、右四连通）方格间长度为一个单位长度。两个相邻方格之间可能会有墙壁，墙壁厚度相对于方格而言非常小，粗略不计。迷宫的边界均有墙壁，且每一堵墙壁均与边界连通。迷宫是完全黑暗的，这意味着，你无法得到除 (n,m) 以外的任何信息。

为了在黑暗条件下尽量不迷路，每次前进时你只能从当前格子出发，沿着左侧或右侧墙壁，左手或右手扶着墙壁前进，并且使扶着墙壁的手移动距离恰好为一个单位长度。需要注意的是，若左侧或右侧墙壁不存在，则沿该侧方向无法前进。

在黑暗中过久的你会感到恐惧，因此你需要在你尽早走出迷宫。如果你没有在限定步数内走出迷宫，挑战将会失败。

对于这类题目，选手只需像往常一样将询问写到标准输出，**刷新输出缓冲**后从标准输入读取结果。选手程序刷新输出缓冲后，通过管道连接它的测评程序（称为交互器）才能立刻接收到这些数据。在 C/C++ 中，`fflush(stdout)` 和 `std::cout << std::flush` 可以实现这个操作（使用 `std::cout << std::endl` 换行时也会自动刷新缓冲区，但是 `std::cout << '\n'` 不会）；Pascal 则是 `flush(output)`。

Grader 交互

Grader 交互方式常见于 IOI、APIO 等国际 OI 赛事（特别是 CMS 平台的竞赛）。

例题"Gap"

UOJ #206.【APIO2016】Gap[10]

有 N 个严格递增的非负整数 $a_1, a_2, \cdots, a_N (0 \le a_1 < a2 < \cdots < a_N \le 10^{18})$。你需要找出 $a_{i+1} - a_i (0 \le i \le N-1)$ 里的最大的值。

你的程序不能直接读入这个整数序列，但是你可以通过给定的函数来查询该序列的信息。关于查询函数的细节，请根据你所使用的语言，参考下面的实现细节部分。

你需要实现一个函数，该函数返回 $a_{i+1} - a_i (0 \le i \le N-1)$ 中的最大值。

对于这类题目，选手只需编写一个特定的函数完成某项任务，它通过调用给定的若干辅助函数来进行交互。为了便于选手在本地测试，题目会下发一个头文件与一个参考测评程序 `grader.cpp`（对于 Pascal 语言是一个库 `graderlib`），选手将自己的程序与 `grader.cpp` 一同编译方可得到可执行文件。

```
g++ grader.cpp my_solution.cpp -o my_solution -Wall -O2
./my_solution    # 执行程序
```

编译得到的程序表现与传统题程序类似。它会打开固定的文件，以固定的格式读取数据，调用选手编写的函数，并将结果和若干信息（例如询问的次数、答案正确性）显示在标准输出上。

实际测评时，选手的程序会与一个不同的 `grader.cpp` 编译。这个 `grader.cpp` 将以类似的方式调用选手编写的函数，并记录其得分。一般来说，这个版本的 `grader.cpp` 所有全局符号都会设为 `static`，也即不能通过冲突命名的方式破解它，但是任何尝试突破 grader 限制的行为都会被判失格 (disqualification)。

差别

STDIO 交互的一个明显优势在于它可以支持任何编程语言，但是输入输出的耗时容易成为问题设计的瓶颈，导致有时无法区分程序的时间效率差别；Grader 交互则恰好相反，由于函数调用的开销不大，常常可以允许 10^6 数量级的询问次数，但是语言的限制是其短板。

如果自己设计题目或举办比赛，需要对二者认真权衡和比较。

通信题

通信题是需要两个选手程序进行通信，合作完成某项任务的题目。第一个程序接收问题的输入，并产生某些输出；第二个程序的输入会与第一个的输出相关（有时是原封不动地作为一个参数，有时会由评测端处理得到），它需要产生问题的解。

通信题的例子有：UOJ #178. 新年的贺电[11]，#454.【UER #8】打雪仗[12] 等。

本地测试的方法由于题目设定的不同而多种多样，常用的形式如：

- 手工输入
- 编写一个辅助程序，转换第一个程序的输出到第二个程序的输入
- 用双向管道将两个程序的标准输入/输出连接起来

由于评测平台对于通信题的支持有限，因而目前为止，通信题只常见于 IOI 系列赛和 UOJ 等少数在线平台举办的比赛。它仍是一个有待探索的领域。

函数补全题

函数补全题是需要选手补全程序的题目。可以理解为在一道交互题中，题目给定了选手代码，要求编写辅助函数。

通常有以下几种形式：

- 给定一个程序，并告知要求补全的代码块将被嵌入在哪里。
- 不给出程序，而将输入信息作为待提交函数的参数。

这种题在 LeetCode[13] 和 PTA - 拼题 A[14] 上比较多见。

其他类型

例题"Quine"

Quine[15]

写一个程序，使其能输出自己的源代码。

代码中必须至少包含十个可见字符。

题目很经典，但是在绝大多数 OJ 上都很难实现。

" 参考代码"

注意：源代码不包含下方第一行（即 `// clang-format off`）。

```
// clang-format off
#include<cstdio>

char *s={"#include<cstdio>%cchar *s={%c%s%c};%cint main(){printf(s,10,34,s,34,10
);return 0;}"};

int main(){printf(s,10,34,s,34,10);return 0;}
```

参考资料与注释

[1] 因为技术上和资源上的限制，一道题目的测试点大多数情况下不能覆盖满足数据范围的全部数据。

[2] 对于 Python 这样的解释性语言则直接由解释器解释运行程序。

[3] 事实上评测系统的实现远比这个复杂，这里只是大概介绍了评测系统的评测过程。

[4] 准确来说，一般是程序的用户态时间。

[5] 这里的评测结果大多也适用于其他类型题目。

[6] 大多数评测系统会将 PE 状态归到 WA 状态当中。

[7] 一些测试点可能会有部分分，选手在完成一个测试点的部分任务，或者选手的输出正确但不够优的情况下，可以获得一定比例的分数。

[8] 说明（英文）

[9] LOJ #559.「LibreOJ Round #9」ZQC 的迷宫

[10] UOJ #206.【APIO2016】Gap

[11] UOJ #178. 新年的贺电

[12] #454.【UER #8】打雪仗

[13] LeetCode

[14] PTA - 拼题 A

[15] Quine

2.3.2 交互题

Authors: countercurrent-time, StudyingFather

上个世纪的 IOI 就已涉及交互题。虽然交互题近年来没有在省选以下的比赛中出现，不过 2019 年里 NOI 系列比赛中连续出现《P5208[WC2019]I 君的商店》、《P5473[NOI2019]I 君的探险》两道交互题，这可能代表着交互题重新回到 NOI 系列比赛中。

交互题没有很高的前置算法要求，一般也没有严格的时间限制，程序的优秀程度往往仅取决于交互次数限制。所以学习交互题时，建议按照难度循序渐进。要是有意锻炼算法思维而不只是单纯地学习算法，那么完成交互题是很不错的方法。虽然交互题对选手已掌握算法的要求通常较低，但仍建议掌握一定提高和省选算法后再尝试做交互题，因为此时自己的算法思维水平和知识面已经达到了一定水平。基础的交互题介绍可以参考 **OI Wiki** 的 题型介绍 - 交互题。

交互题的特殊错误：

- 选手每一次输出后都需要刷新缓冲区，否则会引起 Idleness limit exceeded 错误。另外，如果题目含多组数据并且程序可以在未读入所有数据前就知道答案，也仍然要读入所有数据，否则同样会因为读入混乱引起 ILE（可以一次提出多次询问，一次接收所有询问的回答）。同时尽量不要使用快读。
- 如果程序查询次数过多，则在 Codeforces 上会给出 Wrong Answer 的评测结果（不过评测系统会说明 Wrong Answer 的原因），而 UVa 会给出 Protocol Limit Exceeded (PLE) 的评测结果。
- 如果程序交互格式错误，UVa 会给出 Protocol Violation (PV) 的评测结果。

由于交互题输入输出较为繁琐，所以建议分别封装输入和输出函数。

比赛时如果出题人给出了 grader 头文件（用于 grader 交互题的调试）或者 checker 程序（用于 stdio 交互题的调试），则交互题的调试比较简单，因为交互题的对拍会比普通题目的对拍困难很多。没有 `testlib.h` 的情况下。交互细节较多的题目的 stdio 交互库会一般有 3k 代码量，再加上 3k 长度的对拍器，至少需要一小时实现。但是，无论是否有调试程序，调试交互题的代码都往往需要选手模拟与程序的交互过程，因此交互题需要选手能设计出高质量的程序，尽量保证一遍做对，同时拥有较强的静态查错能力。

例题：

- CF679A Bear and Prime 100[1]
- CF843B Interactive LowerBound[2]
- UOJ206[APIO2016]Gap[3]

- CF750F New Year and Finding Roots[4]
- UVa12731 太空站之谜 Mysterious Space Station[5]

CF679A Bear and Prime 100

每个质数都有且只有两个因数，所以直接枚举要猜的数的因数。由于限制最多询问 20 次，并且对于较大的数（如 92）尝试分解质因数时发现需要最多枚举到 $\lfloor \frac{n}{2} \rfloor$ 的质数。所以我们先筛出 50 以内的质数，每次把所有这些数都询问一遍。

由于本题对拍比较容易，可以直接把值域内的数都尝试一遍。我们会发现程序无法有效处理质数的平方。所以我们要把 2,3,5,7 的平方 4,9,25,49 都放进去，总共 19 个数字，符合题意。

参考代码

```
#include <cstdio>
const int prime[] = {2,  3,  4,  5,  7,  9,  11, 13, 17, 19,
                     23, 25, 29, 31, 37, 41, 43, 47, 49};
int cnt = 0;
char res[5];

int main() {
  for (int i : prime) {
    printf("%d\n", i);
    fflush(stdout);
    scanf("%s", res);
    if (res[0] == 'y' && ++cnt == 2) return printf("composite"), 0;
  }
  printf("prime");
  return 0;
}
```

CF843B Interactive LowerBound

链表最多有 5×10^4 个元素，但我们只能询问 1999 次，并且只能获取元素的后一个元素，所以普通的遍历整个链表的方法不可用。直接设法逼近目标元素的位置只有一种方法：随机撒点。

对于 $n < 2000$ 的情况直接枚举，$n \geq 2000$ 时，我们直接撒 1000 个点，这时这些点之间的期望距离很小，我们可以直接从小于 x 的最大值开始向后遍历，可以证明在到达下一个点之前我们就已得到答案。遍历的过程中一旦找到大于等于 x 的元素，就可以直接推出。

虽然整体思路简单，但实际情况下，如果没有学习过模拟退火等非完美随机算法，思考起来很可能会困难一些。

同时由于 Codeforces 具有 hack 机制，很多人会刻意卡掉没有初始化随机种子的代码，所以在 `random_shuffle()` 函数前需要 `srand((size_t)new char)`。

参考代码

```
#include <algorithm>
#include <cstdio>
#include <cstdlib>
const int N = 50005;
int n, start, x;
int a[N];

int main() {
```

```
  scanf("%d%d%d", &n, &start, &x);
  if (n < 2000) {
    int ans = 2e9;
    for (int i = 1; i <= n; i++) {
      printf("? %d\n", i), fflush(stdout);
      int val, next;
      scanf("%d%d", &val, &next);
      if (val >= x) ans = std::min(ans, val);
    }
    if (ans == 2e9) ans = -1;
    printf("! %d", ans), fflush(stdout);
  } else {
    srand((size_t) new char);
    int p = start, ans = 0;
    for (int i = 1; i <= n; i++) a[i] = i;
    std::random_shuffle(a + 1, a + n + 1);
    for (int i = 1; i <= 1000; i++) {
      printf("? %d\n", a[i]), fflush(stdout);
      int val, next;
      scanf("%d%d", &val, &next);
      if (val < x && val > ans) p = a[i], ans = val;
    }
    while (p != -1 && ans < x) {
      printf("? %d\n", p), fflush(stdout);
      int val, next;
      scanf("%d%d", &val, &next);
      ans = val;
      p = next;
    }
    if (ans < x) ans = -1;
    printf("! %d", ans), fflush(stdout);
  }
  return 0;
}
```

UOJ206[APIO2016]Gap

分两个子任务讨论：

1. 查询次数限制。

   我们考虑第一次查询。因为我们一开始不知道任何数，所以我们需要询问范围 $[1, 10^{18}]$，获得最大最小值。

   由于查询次数限制刚好为 $\frac{N+1}{2}$，所以考虑怎么每一次都能获取之前没有获取过的值，这样能大概在次数范围内获取序列内的所有数。方法也很简单：每次查询 $[s,t]$ 后，设获得的值为 mn, mx，则下一次查询 $[mn+1, mx-1]$。

2. 询问区间大小限制。

   由于题目要求询问区间内的数的数量之和不能超过 $3N$，所以考虑最小化询问区间。上面的方法不再可用，因为其询问区间内的数数量之和规模为 $O(N^2)$。我们可以考虑二分值域，但这种方法并不可靠，最坏可能会被卡到 $O(N^2)$。所以我们需要更有效的划分值域的方法，避免查询区间内的点重复查询，浪费机会。

   考虑到答案不会小于 $\lfloor \frac{a_n - a_1}{N-1} \rfloor$，所以我们可以考虑按这个值划分值域，设 i 初始为 0，ans 初始为上述值，每次询问 $[i, i+ans]$ 并且更新 ans，之后再以 ans 为步长让 i 自增。

   不过这种方法也不能很好地适用于子任务 1，因为最坏可能很多询问的值域内一个数都没有。

参考代码

```cpp
#include <algorithm>
#include <cstdio>

#include "gap.h"

long long findGap(int T, int N) {
  static long long a[100005] = {}, ans = 0;
  long long s = 0, t = 1e18, s1, t1;
  if (T == 1) {
    int l = 1, r = N;
    while (l <= r) {
      MinMax(s, t, &s1, &t1);
      a[l++] = s1, a[r--] = t1;
      s = s1 + 1, t = t1 - 1;
    }
    for (int i = 2; i <= N; i++) ans = std::max(ans, a[i] - a[i - 1]);
  } else if (T == 2) {
    MinMax(s, t, &s1, &t1);
    ans = (t1 - s1) / (N - 1);
    long long l = s1 + 1, r = t1, last = s1;
    for (long long i = l; i <= r;) {
      MinMax(i, i + ans, &s1, &t1);
      i += ans + 1;
      if (s1 != -1) ans = std::max(ans, s1 - last), last = t1;
    }
  }
  return ans;
}
```

CF750F New Year and Finding Roots

看到 $h \leq 7$，询问次数 ≤ 16 的严格要求，我们需要非常严格地最大化利用访问获得的信息。

$h \leq 4$ 时可以直接暴力枚举。然而 $h > 4$ 时需要很高效的遍历算法。

随机撒点不是好方法，因为随机撒点无法确定自己是否足够接近根节点了，并且单纯随机撒点，至少有一次碰到根节点的概率为 $1 - (\frac{2^h-2}{2^h-1})$，即使排除重复撒点的情况后，碰到根节点的概率仍然非常小。

由于 $1 \leq k \leq 3$，并且我们并不知道哪一边更接近根节点，所以我们考虑最坏的情况，即如果 $k = 3$ 时，前两次我们的遍历方向都是远离根节点的，第三次遍历方向是接近根节点的。所以我们必须往三个方向都遍历。

考虑 bfs 和 dfs 两种遍历方法。由于 bfs 搜索树可能很大，所以我们优先考虑 dfs。当然，如果我们知道当前的深度，并且当前深度小到深度范围内的搜索树规模小于等于剩余次数，我们就可以直接 bfs。

知道当前节点的深度，以及当前遍历的方向会获得很大优势。然而知道当前在往根节点还是在往叶子节点遍历是非常困难的事情。如果使用 dfs，只有当遍历到根节点（$k = 2$）或者叶子节点（$k = 1$）时才知道当前方向。所以我们需要尽可能知道当前节点深度，并且不能采用类似迭代加深搜索的方法，遍历中途停下来。

考虑随机一个初始节点，从初始节点出发可能碰到上面的最坏情况。

如果 $k = 1$，我们就可以直接知道当前节点的深度。

如果 $k = 2$，那当前节点即根节点。

如果 $k = 3$，我们直接考虑往三个方向 dfs。考虑到其中两个方向是直接往叶子节点的方向，遍历路径长度相同；另一个方向是往根节点的方向，不过可能中途不小心往叶子节点的方向走了，遍历路径长度会较大。此时我们就可以计算出当前节点的深度。

当 $k = 1$ 或者 $k = 3$ 时，我们需要考虑较长的遍历路径。我们可以知道路径上深度最小的点（必定比初始节点深度小）。如果我们为访问过的节点打标记，不再遍历，此时从该节点开始就只有一条遍历路径。虽然这条路径可能还

是会走向叶子节点，但是这条路径上同样必然存在深度比起点小的节点，我们就可以从这个节点开始继续重复上面的步骤。

当然，我们考虑 $h=7$ 的最坏情况时（每次只往根节点走一步，就直接往叶子节点走），会发现如果只 dfs，最坏需要 $\frac{(1+7)\times 7}{2}=28$ 次询问。不过我们已经知道初始节点的深度，所以我们可以算出所有已遍历节点的深度，并且根据我们开始时对 bfs 的讨论，判断是否可以从深度最小的点直接 bfs。

这时，我们可以算出最坏需要 17 次。所以我们考虑从搜索树上去掉一个节点（根据 dfs 只能盲目遍历的性质，我们考虑 bfs）：即当进行深度为 k 的 bfs 时，搜索树节点最坏有 2^k-1 个，可能需要 2^k-1 次询问才能确定哪个节点的邻居恰有 2 个。不过我们如果已经对其中 2^k-2 个节点询问后，可以知道最后一个节点肯定是根节点。

此时最坏情况下的最优解为：$h=7$ 时，从叶子节点 dfs，每次都是只往根节点走一步，就直接往叶子节点走，询问 10 次后，当前已知最小深度的节点深度为 4，由于已知其父亲，直接从其父亲开始 bfs（搜索树深度为 3，节点数为 $2^3-1=7$）。在 bfs 时询问了 $2^3-2=6$ 次后，确定 bfs 搜索树上最后一个节点为根节点。

此时我们的算法可以刚好卡到最坏 16 次。

参考代码

```cpp
#include <algorithm>
#include <cstdio>
#include <queue>
#include <vector>
using namespace std;
const int N = 256 + 5;
int T, h, chance;
bool ok;
vector<int> to[N], path;

bool read(int x) {
  if (to[x].empty()) {
    printf("? %d\n", x), fflush(stdout);
    int k, t;
    scanf("%d", &k);
    if (k == 0) exit(0);
    for (int i = 0; i < k; i++) {
      scanf("%d", &t);
      to[x].push_back(t);
    }
    if (k == 2) {
      printf("! %d\n", x), fflush(stdout);
      return ok = true;
    }
    chance--;
  }
  return false;
}

bool dfs(int x) {
  if (to[x].empty()) path.push_back(x);
  if (read(x)) return true;
  for (int i : to[x])
    if (to[i].empty()) return dfs(i);
  return false;
}
```

```cpp
void bfs(int s, int k) {
  queue<int> q;
  for (int i : to[s])
    if (to[i].empty()) q.push(i);
  for (int i = 1; i < k; i++) {
    int x = q.front();
    q.pop();
    if (read(x)) return;
    for (int j : to[x])
      if (to[j].empty()) q.push(j);
  }
  for (int i = 1; i < k; i++) {
    int x = q.front();
    q.pop();
    if (read(x)) return;
  }
  printf("! %d\n", q.front()), fflush(stdout);
}

int main() {
  for (scanf("%d", &T); T--;) {
    ok = false;
    for (int i = 0; i < N; i++) to[i].clear();
    chance = 16;
    scanf("%d", &h);
    if (h == 0) exit(0);
    vector<int> long_path;
    if (read(1)) continue;
    int root, dep;
    if (to[1].size() == 1)
      root = 1, dep = h;
    else {
      for (int i : to[1]) {
        path.clear();
        if (dfs(i)) break;
        if (path.size() > long_path.size()) swap(path, long_path);
      }
      if (ok) continue;
      dep = h - (path.size() + long_path.size()) / 2;
      root = long_path.at((long_path.size() - (h - dep)) - 1);
    }
    while ((1 << (dep - 1)) - 2 > chance) {
      path.clear();
      if (dfs(root)) break;
      dep = h - (h - dep + path.size()) / 2;
      root = path.at((path.size() - (h - dep)) - 1);
    }
    if (ok == false) bfs(root, 1 << (dep - 2));
  }
  return 0;
}
```

UVa12731 太空站之谜 Mysterious Space Station

由于唯一的反馈是移动时是否撞墙，所以我们应该考虑在机器人不走丢的情况下，尽量接近墙边走路，这样有几个好处：

- 靠近墙边走路时，很容易知道自己会不会撞墙，获取到尽量多的信息。
- 墙边都是不会出现传送门的格子，可以避免机器人走丢。

所以，我们如果已知机器人可能在墙边的某个位置，要确定机器人是不是真的在这个位置，就可以通过「单手扶墙法」[6] 确定自己是不是真的在这个位置。根据拓扑学原理，在两边都是墙的迷宫中，如果从入口进入，并且总是用一只手扶着同一边墙，就可以保证找到出口。由于本题中的墙是闭合的，所以只需要沿着墙边的道路走，就可以保证可以回到原点而不会撞墙。另外，由于墙边的道路是地图上的最大闭合回路，所以实际代码中并不需要特意撞墙以保证机器人在墙边，可以使用标记在地图中标明墙边道路。而且一旦撞了墙，就需要赶快沿着原路返回，可以在避免机器人走丢的同时减少步数。

由上，可以推断出确定机器人是否在特定格子的试错法：将机器人在不走到未知格子或已知传送门的情况下走到墙边的道路上，然后绕着墙边道路走一圈。这个过程中如果没有撞墙，就可以确定机器人确实是在特定格子。

我们可以采用上面的方法，一开始标出图中所有未知格子，然后从上到下，从左到右依次判断每个未知格子是否是传送门。可以先走到未知格子上方，然后向下、向左走。再用上面的方法判断机器人是不是在未知格子的左侧。如果不是，说明机器人不在应该在的位置，即未知格子是传送门。

找出未知格子后就需要判断 2k 个未知格子的配对关系，实际方法也很简单：只需要暴力配对就可以了。由于 $k \le 5$，所以最多只需要 $9+7+5+3$ 次试错法。作为对比，判断图中全部未知格子的情况最多需要 $121-40$ 次试错法。

由于目前下面这一份代码只能通过 UOJ 的镜像题：#247.【Rujia Liu's Present 7】Mysterious Space Station[7]，而无法通过 UVa 原题。修改了 UOJ 上刘汝佳的标程后还是无法通过，并且暂时无法联系到刘汝佳。所以下面的代码以 UOJ 为准。

不过刘汝佳的标程质量还是比下面这份代码质量高很多的，可以在 UOJ 上查看到 通过了 UOJ 镜像题的标程[8]。同一份数据下，标程使用的移动次数非常少。

参考代码

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <queue>
#include <stack>

#define Wall 0
#define Unknown 1
#define Space 2
#define Gate 3
#define Path 4

const int N = 20;
const int dir[8][2] = {{0, 1},  {1, 0}, {0, -1}, {-1, 0},
                       {-1, 1}, {1, 1}, {1, -1}, {-1, -1}};
const char dirs[5] = "ESWN";
int n, m, k;
int a[N][N], id[N][N];

struct point {
  int x, y;
```

```
  point(int x = 0, int y = 0) : x(x), y(y) {}

  bool operator==(const point& tmp) const { return x == tmp.x && y == tmp.y; }

  bool operator!=(const point& tmp) const { return !(*this == tmp); }

  point side(int d) const { return point(x + dir[d][0], y + dir[d][1]); }

  int check(int d) { return a[x + dir[d][0]][y + dir[d][1]]; }

  int id() { return ::id[x][y]; }
} start;

std::vector<std::pair<point, int>> path;
std::pair<point, point> ans[N];
std::pair<point, bool> vis[N];

bool walk(int d) {
  printf("MoveRobot %c\n", dirs[d]);
  fflush(stdout);
  int ret;
  scanf("%d", &ret);
  return ret;
}

bool walk(int d, std::stack<int>& st) {
  if (walk(d)) {
    st.push(d);
    return true;
  }
  return false;
}

bool read() {
  if (scanf("%d%d%d", &n, &m, &k) != 3) return false;
  if (n == 0) return false;
  memset(a, 0, sizeof(a));
  for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++) {
      char c;
      std::cin >> c;
      if (c == 'S') start = point(i, j);
      if (c == '*')
        a[i][j] = Wall;
      else
        a[i][j] = Unknown;
    }
  return true;
}

void answer() {
  for (int i = 0; i < k; i++)
    printf("Answer %d %d\n", ans[i].first.id(), ans[i].second.id());
```

```
  fflush(stdout);
}

// 单手扶墙法，因为靠墙的 Path 是极大闭合环，所以只需要在沿着 Path
// 走的过程中没有碰到障碍就可以了
void wall_follower_init(point x, int last, int wallside, point s) {
  if (x == s && !path.empty()) return;
  if (x.check(wallside) == Path) {
    path.push_back(std::make_pair(x, wallside));
    wall_follower_init(x.side(wallside), wallside, last ^ 2, s);
  } else if (x.check(last) == Wall) {
    for (int i = 0; i < 4; i++)
      if (i != (last ^ 2) && x.check(i) != Wall) {
        path.push_back(std::make_pair(x, i));
        wall_follower_init(x.side(i), i, last, s);
        return;
      }
  } else {
    path.push_back(std::make_pair(x, last));
    wall_follower_init(x.side(last), last, wallside, s);
  }
}

void init() {
  int cnt = 1;
  for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++) {
      if (a[i][j] == Unknown) {
        id[i][j] = cnt++;
        for (int k = 0; k < 8; k++)
          if (point(i, j).check(k) == Wall) {
            a[i][j] = Path;
            break;
          }
      } else
        id[i][j] = 0;
    }
  path.clear();
  int wallside = 0, last = 0;
  for (int i = 0; i < 4; i++)
    if (start.check(i) == Wall) {
      wallside = i;
      break;
    }
  for (int i = 0; i < 4; i++)
    if (start.check(i) == Path && i != (wallside ^ 2)) {
      last = i;
      break;
    }
  wall_follower_init(start, last, wallside, start);
}

void undo(std::stack<int>& st) {
  while (!st.empty()) walk(st.top() ^ 2), st.pop();
```

```
}

bool wall_follower(point x) {
  std::stack<int> st;
  bool ok = true;
  int i = 0;
  while (i < path.size() && path[i].first != x) i++;
  for (int j = i; ok && j < path.size(); j++) {
    if (walk(path[j].second))
      st.push(path[j].second);
    else
      ok = false;
  }
  for (int j = 0; ok && j < i; j++) {
    if (walk(path[j].second))
      st.push(path[j].second);
    else
      ok = false;
  }
  if (ok == false) undo(st);
  return ok;
}

// 确定自己当前在
// x，使用「摸着石头过河」的方法，只需要沿着可以避开障碍、未知格子和传送门的方向走到
// Path 就行。在找传送门和配对传送门时使用
void bfs(point s, point t, std::vector<int>& v) {
  static int map[N][N] = {};
  memset(map, -1, sizeof(map));
  std::queue<point> q;
  map[s.x][s.y] = 4;
  q.push(s);
  while (!q.empty()) {
    point x = q.front();
    q.pop();
    if (x == t) break;
    for (int i = 0; i < 4; i++) {
      point y = x.side(i);
      if ((x.check(i) == Path || x.check(i) == Space) && map[y.x][y.y] == -1) {
        map[y.x][y.y] = i;
        q.push(y);
      }
    }
  }
  for (point x = t; x != s; x = x.side(map[x.x][x.y] ^ 2)) {
    v.push_back(map[x.x][x.y]);
  }
  std::reverse(v.begin(), v.end());
}

bool move(point s, point t, std::stack<int>& st) {  // 在靠近传送门时使用
  static std::vector<int> v;
  v.clear();
  bfs(s, t, v);
```

```
  for (int i : v)
    if (walk(i, st) == false) return false;
  return true;
}

// 尽可能快地向墙边移动
bool make_sure(point x, int last) {
  if (a[x.x][x.y] == Path) return wall_follower(x);
  for (int i = 0; i < 4; i++)
    if ((x.check(i) == Path || x.check(i) == Space) && i != (last ^ 2)) {
      if (!walk(i)) return false;
      bool ret = make_sure(x.side(i), i);
      walk(i ^ 2);
      return ret;
    }
  return false;
}

void find_gate() {
  int cnt = 0;
  std::stack<int> st;
  for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
      if (cnt == k * 2 && a[i][j] == Unknown)
        a[i][j] = Space;
      else if (a[i][j] == Unknown) {
        bool ok = true;
        if (!move(start, point(i - 1, j), st))
          ok = false;
        else if (!walk(1, st))
          ok = false;
        else if (!walk(2, st))
          ok = false;
        else if (!make_sure(point(i, j - 1), -1))
          ok = false;
        if (ok == false) {
          vis[cnt++] = std::make_pair(point(i, j), false);
          a[i][j] = Gate;
          for (int k = 0; k < 8; k++) {
            point y = point(i, j).side(k);
            if (point(i, j).check(k) == Unknown) a[y.x][y.y] = Space;
          }
        } else
          a[i][j] = Space;
        undo(st);
      }
}

void make_gate_pair() {
  int cnt = 0;
  std::stack<int> st;
  for (int i = 0; i < k * 2; i++)
    if (vis[i].second == false)
      for (int j = 0; vis[i].second == false && j < k * 2; j++)
```

```
        if (j != i && vis[j].second == false) {
          bool ok = true;
          if (!move(start, vis[i].first.side(2), st))
            ok = false;
          else if (!walk(0, st))
            ok = false;
          else if (!make_sure(vis[j].first.side(0), -1))
            ok = false;
          if (ok == true) {
            ans[cnt++] = std::make_pair(vis[i].first, vis[j].first);
            vis[i].second = vis[j].second = true;
          }
          undo(st);
        }
}

int main() {
  while (read()) {
    init();
    find_gate();
    make_gate_pair();
    answer();
  }
  return 0;
}
```

习题

- 刘汝佳的交互题专场比赛 Rujia Liu's Present 7 质量非常高，推荐一做。[9]
- P5473[NOI2019]I 君的探险[10]
- P5208[WC2019]I 君的商店[11]

参考资料与拓展阅读

- 用 Linux 管道实现 online judge 的交互题功能[12]

参考资料与注释

[1] CF679A Bear and Prime 100

[2] CF843B Interactive LowerBound

[3] UOJ206[APIO2016]Gap

[4] CF750F New Year and Finding Roots

[5] UVa12731 太空站之谜 Mysterious Space Station

[6] 「单手扶墙法」

[7] #247.【Rujia Liu's Present 7】Mysterious Space Station

[8] 通过了 UOJ 镜像题的标程

[9] 刘汝佳的交互题专场比赛 Rujia Liu's Present 7 质量非常高，推荐一做。

[10] P5473[NOI2019]I 君的探险

[11] P5208[WC2019]I 君的商店

[12] 用 Linux 管道实现 online judge 的交互题功能

2.4 学习路线

提示

本文章正在编辑讨论中，欢迎补充更进一步的学习路线或在评论区提出你的想法！

本文将会介绍算法竞赛的学习路线。

该学习路线既是新手学习算法竞赛知识的指南，也是一份复习清单。

1 C++ 语言基础

先从 C++ 语法学起，一步一步来。

1.1 Hello, World!

以一句 `Hello, World!`，开始算法竞赛之旅吧！

同时了解一下 C++ 的源程序的大致框架是什么样子的。

- Hello, World!
- C++ 语法基础

1.2 变量与运算

计算机出现的最初目的就是计算。因此我们先学习如何完成一些简单的运算任务吧。

- 变量
- 运算

1.3 流程控制

1.3.1 分支结构

有的时候，我们需要在不同的条件下，选择执行不同的语句，这时候我们就需要借助分支语句。

- 分支

分支语句包括下面几种：

- if 语句
- if-else 语句
- if-elif-else 语句
- switch 语句

1.3.2 循环结构

将若干条语句重复执行多次，就需要用到循环语句。

- 循环

循环语句包括下面几种：

- for 语句
- while 语句
- do-while 语句

1.4 数组与结构体

数组用于存储大量相同类型的数据。而结构体则可以将若干变量捆绑起来。

- 数组
- 结构体

1.5 函数与递归

使用函数来让程序变得模块化，降低实现成本。

递归则是新手入门的一道坎，「自己调用自己」听起来并不是那么容易理解，不过仔细深究根本，就会发现「自己调用自己」和「自己调用别人」并没有本质差别。

- 函数
- 递归 & 分治

2 CSP-J 入门级

2.1 枚举与模拟

从现在开始，你已经会使用 C++ 语言完成一些简单的任务了，但是这远远不够。

为了做对一些简单的题目，你需要学会通过枚举或模拟脑海中的逻辑，来实现代码。这看起来并不是很高效，但有的时候很管用。

- 枚举
- 模拟

2.2 递归与分治

递归是指函数定义中不断调用自己的方法；而分治则是不断将这一个问题分解为若干子问题，求解后合并的操作。

- 递归 & 分治

2.3 字符串

在做信息学题目时，经常会碰到的一个数据类型就是字符串，你需要学习一些用于操作字符串的 STL 函数。当然，模拟也是解决字符串问题的好方法。

- 字符串基础
- STL 函数

2.4 排序

当你获得了一组数据时，如何将他们从无序变成有序也是个很重要的问题。在你没有思路的时候，不妨考虑一下将数组排个序吧。这也是接下来的很多算法的基础。

排序的方法有点多，但理解后记住它们并不难。

- 排序简介
- 选择排序
- 冒泡排序
- 插入排序
- 计数排序
- 基数排序
- 快速排序
- 归并排序
- 堆排序
- 桶排序
- 排序相关 STL

NOI 大纲中入门级只要求学习选择、冒泡、插入排序，共三个排序算法，但是其余的难度也并不大，且初赛中可能涉及，故一并列出。

2.5 二分与倍增

二分查找，本质上是运用分治的思想，不断减少查找范围的大小，直至找到答案。但是需要注意，这个查找方式必须应用在有序的数据结构中。

- 二分

而倍增则不同，它是不断翻倍，以把线性范畴内的处理转化为对数级，大大优化时间复杂度。（这个知识点需要一点数学基础，暂时跳过也问题不大）

- 倍增

2.6 搜索

在入门组，搜索的题目常常会在迷宫类题目中出现，一般会有地图类的数据；此外，搜索也十分常用于高效地枚举构造合法解的情况，亦可用于骗分。

2.6.1 深度优先搜索（DFS）

深度优先搜索指利用递归函数方便地实现暴力枚举的算法，与图论中的 DFS 算法有一定相似之处，但并不完全相同。

- DFS（搜索）

2.6.2 广度优先搜索（BFS）

将每一个状态设计为图中的一个点，可以展开地毯式搜索。

- BFS（搜索）

2.6.3 搜索优化

很多题目都可以用 DFS 来解决，而这个算法的复杂度显然是无法通过的。因此，需要一些优化使它跑得更快。这样的优化能够减少不可能成功的尝试，称为「剪枝」。BFS 相关的优化就要更加灵活了，但是基本思路和这里是一样的。

- DFS 剪枝优化

2.7 数据结构入门

2.7.1 线性数据结构

数组，链表，队列，栈，都是线性结构。巧用这些结构可以做出不少方便的事情。

- 栈
- 队列
- 链表

2.7.2 复杂数据结构

- 树及二叉树
- 图的概念
- 图的存储

2.8 动态规划入门

动态规划（Dynamic Programming, DP）是一种通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法。

由于动态规划并不是某种具体的算法，而是一种解决特定问题的方法，因此它会出现在各式各样的数据结构中，与之相关的题目种类也更为繁杂。

- 动态规划简介

2.8.1 背包问题

即给出一个有限制容量的背包，选择放入若干有容量和价值的物品，求解如何放置能使得价值总和最大。这是阻挡很多 OIer 的第一道坎，从这里开始，算法就有些难以理解。

- 背包 DP

2.8.2 线性动态规划

在动态规划中，最难的部分之一就是设计状态，需要用到构造相关技巧。当你写出了状态和状态转移方程之后，完成一道动态规划的题目就不难了。

- 构造
- 动态规划基础

记忆化搜索是一种通过记录已经遍历过的状态的信息，从而避免对同一状态重复遍历的搜索实现方式。有的题目也可以使用记忆化搜索来降低思维难度。

因为记忆化搜索确保了每个状态只访问一次，它也是一种常见的动态规划实现方式。

- 记忆化搜索

2.8.3 复杂动态规划

区间类动态规划是线性动态规划的扩展，它在分阶段地划分问题时，与阶段中元素出现的顺序和由前一阶段的哪些元素合并而来有很大的关系。

- 区间 DP

2.9 数学

2.9.1 高精度算法

就算是 long long（或 int64）还不够怎么办？用高精度算法。本质上就是模拟了四则运算。

- 高精度计算

2.9.2 进制转换

在计算机中，除了二进制，比较常用的还有八进制和十六进制。有的时候学会运用正确的进制对解题也有很大帮助。

- 进位制

2.9.3 位运算

位运算就是基于整数的二进制表示进行的运算。由于计算机内部就是以二进制来存储数据，位运算是相当快的。

基本的位运算共 6 种，分别为按位与、按位或、按位异或、按位取反、左移和右移。

- 位运算

2.9.4 数论

- 数论基础
- 素数
- 筛法
- 最大公因数
- 欧拉函数
- 分解质因数

2.9.5 组合计数

- 排列组合
- 抽屉原理
- 容斥原理

至此，你就学习完了入门组范畴内的所有算法，但是想要掌握它们，你需要继续进行足够数量的刷题，以巩固你所学到的知识点。

2.5 学习资源

Authors: Suyun514, ChungZH, Enter-tainer, StudyingFather, Konano, JulieSigtuna, GldHkkowo, SukkaW, Rapiz1, Henry-ZHR, H-J-Granger, countercurrent-time, fouzhe, Ir1d, abc1763613206, EndlessCheng, Plaaant6, LUTLJS, ZsgsDesign

本页面主要列举了一些与算法竞赛有关的在线评测网站、题目合集、书籍、工具等资源。

在线评测平台

在线评测平台（英语：Online Judging System，简称：OJ），一般用于刷题训练，参与和组织比赛，以及用户之间的交流分享。

国内

- 51Nod[1]：有许多值得尝试的数学题和思维题。
- Comet OJ[2]：始于 2018 年，旨在为广大算法爱好者提供一个竞技、练习、交流的平台，经常举办原创性的高质量比赛，有丰富的题库。
- FZUOJ[3] 始于 2008 年，福州大学在线评测系统。
- HDU Online Judge[4] 始于 2005 年，杭州电子科技大学在线评测系统，有多校训练的题目。
- hihoCoder[5] 始于 2012 年，面向企业招聘，有些题目来自于每周一题，涉及知识点的学习。（登录后方可查看题面）
- HydroOJ[6]：始于 2021 年，为开源项目 Hydro[7] 的官方站。用户可以创建自己的 域[8]，域中可以使用题库、比赛、讨论等主站可以使用的功能。
- 计蒜客[9] 北京矩道优达网络科技有限公司旗下的核心产品，提供按知识点和难度筛选的信息学题库和 ICPC 题库。
- Judge Duck Online[10] 基于 松松松[11] 开发的开源项目 JudgeDuck[12]，可以将评测程序的运行时间精确到微秒。（题目较少）
- LibreOJ[13]：始于 2017 年。基于开源项目 Lyrio[14]，Libre 取自由之意。题目所有测试数据以及提交的代码均对所有用户开放。目前由 Menci[15] 维护。
- Lutece[16]：电子科技大学在线评测系统，始于 2018 年，项目开源[17]。
- 洛谷[18]：始于 2013 年，社区群体庞大，各类 OI 的真题和习题较全。提供有偿教育服务。
- 牛客网[19]：始于 2014 年，提供技术类求职备考、社群交流、企业招聘等服务。
- NOJ[20]：南京邮电大学在线评测系统，始于 2008 年，项目开源[21]。自身拥有题目两千余道，同时支持对多个国内外 OJ 的提交，可以直接在 NOJ 提交别的 OJ 的题。
- NTUOJ[22]：台湾大学在线评测系统，始于 2007 年，基于开源项目 Judge Girl[23]。
- OpenJudge[24]：始于 2005 年，由 POJ 团队开发的小组评测平台。
- POJ[25]：北京大学在线评测系统，始于 2003 年，国内历史最悠久的 OJ 之一。内有很多英文题，既有基础题，也有值得一试的好题。
- PTA（拼题 A）[26]：始于 2016 年，浙江大学衍生的杭州百腾教育科技有限公司产品。
- 清澄[27]：始于 2005 年，由 胡伟栋[28] 开发。自 2019 年 9 月 1 日起不再对外提供服务。
- Universal Online Judge[29]：始于 2014 年，Universal 取通用之意，项目开源[30]；VFK[31] 的 OJ：多原创比赛题和 CCF/THU 题，难度较高。
- Vijos[32]：始于 2005 年。服务端[33] 和 评测机[34] 等项目开源。
- WZOI[35]：始于 2017 年，由浙江省温州中学维护的 开源[36] 评测系统。
- ZOJ[37]：浙江大学在线评测系统，始于 2001 年。

国外

- AizuOJ[38]：日本会津大学在线评测系统，始于 2004 年。包含日本若干高中和大学编程比赛的题目，自带编程/数据结构/算法的入门课程。
- AtCoder[39]：日本 OJ，日文版里会有日本高校的比赛，英文内不会显示。题目有趣，质量较高。
- CodeChef[40]：印度 OJ，周期举办比赛。系统基于 SPOJ 的 Sphere Engine。
- Codeforces[41]：俄罗斯 OJ，始于 2010 年，创始人是 Mike Mirzayanov[42]。有多种系列的比赛，并支持个人出题、申请组织比赛。题目质量较高。
- CSES[43](Code Submission Evaluation System)，按专题划分的题库，旨在[44] 成为综合的高质量题库，目前只有 200 题，主要由 Competitive Programmer' s Handbook[45] 作者 Antti Laaksonen 开发，始于 2013。
- CS Academy[46]
- DMOJ[47] 加拿大开源的 OJ，语言支持广；题库是各大比赛的存档，也有定期自行举办的比赛。
- HackerRank[48] 有很多比赛
- ICPC Live Archive[49] 存档了 1990 年至今的 ICPC 区域赛和总决赛题目；但部分比赛的评测数据仅为样例数据，且对 Special Judge 的支持不完善。
- ICPC Problem Archive[50] 基于 Kattis 系统；存档了 2012 年至今的 ICPC 全球总决赛题目，并且会在总决赛开赛时同步发放题目（但不会有同步赛）。
- Kattis[51] 题库主要包含类似 ICPC 比赛的题目；根据用户解题情况评定用户等级，推荐适合该用户水平的 trivial/easy/medium/hard 四类难度的题目，其中题目难度采用类 ELO 等级分[52] 系统来评估。
- LeetCode[53] 码农面试刷题网站，有中文分站：LeetCode China[54]。
- Light OJ[55] 一个快挂了的 OJ，`www` 域名无法访问，请使用 根域名[55] 访问
- opentrains[56] 俄罗斯 Open Cup[57] 比赛的训练平台，基于 ejudge[58] 开源系统搭建，支持虚拟比赛；题库包含历年 Open Cup 赛题以及 Petrozavodsk 训练营的题目。
- SPOJ[59] 始于 2003 年，其后台系统 Sphere Engine[60] 于 2008 年商业化；支持题目点赞和标签功能。
- TopCoder[61] 始于 2001 年，其 竞技编程社区[62] 有很多比赛；目前主营业务是技术众包。
- TimusOJ[63] 始于 2000 年，由 Ural Federal University 开发，拥有俄罗斯最大的在线评测题库，题目主要来自乌拉尔联邦大学校赛、乌拉尔锦标赛、ICPC 乌拉尔区域赛、以及 Petrozavodsk 训练营。
- Online Judge（前 UVaOJ[64]）始于 1995 年，国际成名最早的 OJ，创始人是西班牙 University of Valladolid (UVa) 的 Miguel Ángel Revilla 教授；由于 Revilla 教授于 2018 年不幸离世[65]，且 Valladolid 大学终止维护，UVaOJ 自 2019 年 7 月起更名为 Online Judge。现在该平台的维护者 正在 GitHub 上构建新的评测平台[66]。
- Yandex[67] 存档了近几年的全俄罗斯信息学奥赛。

教程资料

- **OI Wiki**[68]
- Codeforces 上网友整理的一份教程合集[69]
- 英文版 E-Maxx 算法教程[70]
- 演算法笔记[71]：台湾师范大学总结的教程
- 如何为 ACM-ICPC 做准备 ? - geeksforgeeks[72]
- Topcoder 整理的教程[73]
- 校招面试指南[74]
- 由 hzwer 收集整理自互联网的课件[75]
- Trinkle23897 的课件[76]
- huzecong 的课件[77]
- Open Data Structure[78]：内含众多数据结构讲稿

- IOI Syllabus (2020)[79]

书籍

本列表内注明了书籍作者，译者未列其中。因无重名书籍且易于寻找，故不标明 ISBN。

- 刘汝佳系列
  - 《算法竞赛入门经典》（紫）
    - 第一版 配套资源仓库 （ 镜像 ）[80]
    - 第二版 配套资源仓库[81]
    - 第二版 习题选解[82]
  - 《算法竞赛入门经典 - 训练指南》（白/蓝）- 陈锋合著
  - 《算法艺术与信息学竞赛》（蓝/黑）
- 《算法竞赛进阶指南》- 李煜东
  - 配套资源仓库[83]
- 《啊哈算法》- 纪磊
  - 面向初学者或有初步兴趣的人群，有幽默配图。
- CCF 中学生计算机程序设计系列
  - 《CCF 中学生计算机程序设计 - 入门篇》- 陈颖，邱桂香，朱全民
    - 建议配合勘误使用 。[84]
  - 《CCF 中学生计算机程序设计 - 基础篇》- 江涛，宋新波，朱全民
  - 《CCF 中学生计算机程序设计 - 提高篇》- 徐先友，朱全民
  - 《CCF 中学生计算机程序设计 - 专业篇》（未出）
- 深入浅出系列
  - 《深入浅出程序设计竞赛 - 基础篇》- 洛谷网校教研组
- 一本通系列
  - 《信息学奥赛一本通》- 董永建
  - 《信息学奥赛一本通 - 提高篇》- 黄新军，董永建
    - 建议选择性阅读 。[85]
  - 《信息学奥赛一本通 - 高手训练》- 黄新军，董永建
- 其他由国内著名 OI 教练写的教材
  - 《信息学奥赛课课通》- 林厚从
  - 《聪明人的游戏：信息学探秘 - 提高篇》- 江涛，陈茂贤
  - 《计算概论：C++ 编程与信息学竞赛入门》- 金靖
  - 《算法竞赛宝典》- 张新华
- ACM 国际大学生程序设计竞赛系列
  - 《ACM 国际大学生程序设计竞赛系列知识与入门》- 俞勇
  - 《ACM 国际大学生程序设计竞赛系列算法与实现》- 俞勇
  - 《ACM 国际大学生程序设计竞赛系列题目与解读》- 俞勇
- 《算法竞赛入门到进阶》- 罗勇军，郭卫斌
- 《算法导论》第三版 - Thomas H.Cormen/Charles E.Leiserson/Ronald L.Rivest/Clifford Stein
  黑书，大学经典教材。英文版原名 *Introduction to Algorithms*
  - 答案解析 (English)[86]

- 《具体数学》第二版 - Ronald L. Graham/Donald E. Knuth/Oren Patashnik
  英文版原名 *Concrete Mathematics*
- 《组合数学》第五版 - Richard A.Brualdi
  英文版原名 *Introductory Combinatorics*
- 《挑战程序设计竞赛》全套 - 秋叶拓哉，岩田阳一，北川宜稔通俗易懂。
  - 译者博客的介绍页[87]
- 《算法概论》- Sanjoy Dasgupta/Christos Papadimitriou/Umesh Vazirani
  - 提纲挈领，但内容较少。
- Legend-K 的数据结构与算法的笔记[88]
- acm-cheat-sheet[89]
- Competitive Programmer’s Handbook[45]- Antti Laaksonen
  - 作者花了三年个人时间完成。面向算法竞赛，覆盖面广，详略得当。
- 《挑战编程：程序设计竞赛训练手册》[90]- Steven S. Skiena/Miguel A. Revilla
  - 由西班牙 University of Valladolid 的两位教授编写。
  - 阅读 经过翻译的在线电子版图书[91]
  - 购买 纸质版图书[92]
- 《C++，挑战编程——程序设计竞赛进阶训练指南》- 邱秋
  - 作者博客的介绍页[93]
- 《数据结构（C++ 语言版 第 3 版）》- 邓俊辉[94]
  - 建议随配套课程、配套课件和习题解析一起使用。
- 《计算几何：算法与应用》- 伯格（Berg,M.D.）著，邓俊辉译
  英文版原名 *Computational Geometry: Algorithms and Applications*
- 《Handbook of Data Structures and Applications, 2nd Edition》[95]
  - 由许多著名教授如 Sartaj Sahni、Hanan Samet、Weiss 等合著，内容较多，建议有一定基础的数据结构爱好者阅读。
- 算法详解 系列[96]
  - 面向有语言基础的初学者的教材，建议同配套课程一起使用
  - 《Algorithms Illuminated, Part 1: The Basics》- Tim Roughgarden
  - 《算法详解，卷 1：算法基础》- 徐波译
  - 《Algorithms Illuminated, Part 2: Graph Algorithms and Data Structures》- Tim Roughgarden
  - 《算法详解，卷 2：图算法和数据结构》- 徐波译
  - 《Algorithms Illuminated, Part 3: Greedy Algorithms and Dynamic Programming》- Tim Roughgarden
  - 《Algorithms Illuminated, Part 4: Algorithms for NP-Hard Problems》- Tim Roughgarden

课程

- Baylor: CSI 3144 (2006)[97]
- CMU 15-295 (2021)[98]
- Georgia Tech: CS 4540 (2020)[99]
- Georgia Tech: CS 6550 (2021)[100]
- LSU: CSC 2700 (2021)[101]
- NUS: CS 3233 (2021)[102]
- Reykjavik: T-414-ÁFLV (2016)[103]

- SPSU: Coursera (2019)[104]
- Stanford: CS 97SI (2015)[105]
- Stonybrook: CSE 392 (2012)[106]
- TAMU: CSCE 430 (2021)[107]
- UBC: CPSC 490 (2021)[108]
- UCF: COP 4516 (2021)[109]
- VT: CS 2984/4984 (2020)[110]
- THU：数据结构[111]
- THU：计算几何[112]
- StanfordOnline: Algorithms: Design and Analysis[96]

工具

- 《100 个 gdb 小技巧》[113]
- Algorithm Visualizer[114]
- cppreference[115]：一个全面的 C 和 C++ 语言及其标准库的在线参考资料
- Compiler Explorer[116]：在线查看编译后代码块对应的汇编语句，支持选择不同的编译器
- C++ Insights[117]：以编译器的视角去查看你的 C++ 源码
- Inverse Symbolic Calculator[118]：实数反查表达式，适用于反推常数
- LaTeX 手写符号识别[119]
- LaTeX 数学公式参考[120]
- Mathpix[121]：截图转 LaTeX
- OEIS[122]：整数数列搜索引擎
- Python Tutor[123]: 代码执行过程可视化
- Quick C++ Benchmark[124]：在线比较两个及以上函数的运行速度
- Try It Online[125]：在线运行 600+ 种语言的代码，支持 IO 交互，超时 60s，可以分享代码
- 图论画板[126] 与 GraphViz[127]
- Ubuntu Pastebin[128]：可用于分享代码
- uDebug[129]：提供一些 OJ 题目的调试辅助
- USF[130] 与 VisuAlgo[131]：算法可视化
- Wandbox[132]: 在线代码运行，支持 30+ 种语言，可以分享代码，支持不同编译器版本
- Wolfram Alpha[133]：可以计算包括数学、科学技术、社会文化⋯⋯等多个主题的问题

题集和资源

- POJ 训练计划[134]
- USACO[135]
- 洛谷题单[136]
- -Morass- 贴在 Codeforces 上的一份题单[137]
- Codeforces 社区高质量算法文章合集 之一[138] 之二[139]
- 北京大学 ICPC 暑期课课件例题[140]
- 北京大学 ICPC 暑期课课件[141]
- GitHub.com:OI-wiki/libs[142]

- 多校联合训练[143] 关键词：`Multi-University Training Contest`
- Vjudge[144]
- Project Euler[145]
- Junior Training Sheet[146]：对新手友好的训练计划
- USACO Guide[147]：针对 USACO 的各个级别分类的训练资源

参考资料与注释

[1] 51Nod

[2] Comet OJ

[3] FZUOJ

[4] HDU Online Judge

[5] hihoCoder

[6] HydroOJ

[7] Hydro

[8] 域

[9] 计蒜客

[10] Judge Duck Online

[11] 松松松

[12] JudgeDuck

[13] LibreOJ

[14] Lyrio

[15] Menci

[16] Lutece

[17] 项目开源

[18] 洛谷

[19] 牛客网

[20] NOJ

[21] 项目开源

[22] NTUOJ

[23] Judge Girl

[24] OpenJudge

[25] POJ

[26] PTA（拼题 A）

[27] 清澄

[28] 胡伟栋

[29] Universal Online Judge

[30] 项目开源

[31] VFK

[32] Vijos

[33] 服务端

[34] 评测机

[35] WZOI

[36] 开源

[37] ZOJ

[38] AizuOJ

[39] AtCoder

[40] CodeChef

[41] Codeforces

[42] Mike Mirzayanov

[43] CSES

[44] 旨在

[45] Competitive Programmer’s Handbook [45-1] [45-2]

[46] CS Academy

[47] DMOJ

[48] HackerRank

[49] ICPC Live Archive

[50] ICPC Problem Archive

[51] Kattis

[52] ELO 等级分

[53] LeetCode

[54] LeetCode China

[55] Light OJ [55-1] [55-2]

[56] opentrains

[57] Open Cup

[58] ejudge

[59] SPOJ

[60] Sphere Engine

[61] TopCoder

[62] 竞技编程社区

[63] TimusOJ

[64] UVaOJ

[65] Revilla 教授于 2018 年不幸离世

[66] 正在 GitHub 上构建新的评测平台

[67] Yandex

[68] **OI Wiki**

[69] Codeforces 上网友整理的一份教程合集

[70] 英文版 E-Maxx 算法教程

[71] 演算法笔记

[72] 如何为 ACM-ICPC 做准备？ - geeksforgeeks

[73] Topcoder 整理的教程

[74] 校招面试指南

[75] 由 hzwer 收集整理自互联网的课件

[76] Trinkle23897 的课件

[77] huzecong 的课件

[78] Open Data Structure

[79] IOI Syllabus (2020)

[80] 第一版配套资源仓库（镜像）

[81] 第二版配套资源仓库

[82] 第二版习题选解

[83] 配套资源仓库

[84] 建议配合勘误使用。

[85] 建议选择性阅读。

[86] 答案解析 (English)

[87] 译者博客的介绍页

[88] Legend-K 的数据结构与算法的笔记

[89] acm-cheat-sheet

[90] 《挑战编程：程序设计竞赛训练手册》

[91] 经过翻译的在线电子版图书

[92] 纸质版图书

[93] 作者博客的介绍页

[94] 《数据结构（C++ 语言版第 3 版)》- 邓俊辉

[95] 《Handbook of Data Structures and Applications, 2nd Edition》

[96] 算法详解系列 [96-1] [96-2]

[97] Baylor: CSI 3144 (2006)

[98] CMU 15-295 (2021)

[99] Georgia Tech: CS 4540 (2020)

[100] Georgia Tech: CS 6550 (2021)

[101] LSU: CSC 2700 (2021)

[102] NUS: CS 3233 (2021)

[103] Reykjavik: T-414-ÁFLV (2016)

[104] SPSU: Coursera (2019)

[105] Stanford: CS 97SI (2015)

[106] Stonybrook: CSE 392 (2012)

[107] TAMU: CSCE 430 (2021)

[108] UBC: CPSC 490 (2021)

[109] UCF: COP 4516 (2021)

[110] VT: CS 2984/4984 (2020)

[111] THU: 数据结构

[112] THU: 计算几何

[113] 《100 个 gdb 小技巧》

[114] Algorithm Visualizer

[115] cppreference

[116] Compiler Explorer

[117] C++ Insights

[118] Inverse Symbolic Calculator

[119] LaTeX 手写符号识别

[120] LaTeX 数学公式参考

[121] Mathpix

[122] OEIS

[123] Python Tutor

[124] Quick C++ Benchmark

[125] Try It Online

[126] 图论画板

[127] GraphViz

[128] Ubuntu Pastebin

[129] uDebug

[130] USF

[131] VisuAlgo

[132] Wandbox

[133] Wolfram Alpha

[134] POJ 训练计划

[135] USACO

[136] 洛谷题单

[137] -Morass- 贴在 Codeforces 上的一份题单

[138] 之一

[139] 之二

[140] 北京大学 ICPC 暑期课课件例题

[141] 北京大学 ICPC 暑期课课件

[142] GitHub.com:OI-wiki/libs

[143] 多校联合训练

[144] Vjudge

[145] Project Euler

[146] Junior Training Sheet

[147] USACO Guide

2.6 技巧

2.6.1 读入、输出优化

> **Authors:** Marcythm, YZircon, Chaigidel, Tiger3018, voidge, H-J-Granger, ouuan, Enter-tainer, lcfsih, Xeonacid, Ir1d

在默认情况下，`std::cin/std::cout` 是极为迟缓的读入/输出方式，而 `scanf/printf` 比 `std::cin/std::cout`

快得多。

" 注意"

cin/cout 与 scanf/printf 的实际速度差会随编译器和操作系统的不同发生一定的改变。如果想要进行详细对比，请以实际测试结果为准。

下文将详细介绍读入输出的优化方法。

关闭同步/解除绑定

std::ios::sync_with_stdio(false)

这个函数是一个「是否兼容 stdio」的开关，C++ 为了兼容 C，保证程序在使用了 printf 和 std::cout 的时候不发生混乱，将输出流绑到了一起。同步的输出流是线程安全的。

这其实是 C++ 为了兼容而采取的保守措施，也是使 cin/cout 速度较慢的主要原因。我们可以在进行 IO 操作之前将 stdio 解除绑定，但是在这样做之后要注意不能同时使用 std::cin 和 scanf，也不能同时使用 std::cout 和 printf，但是可以同时使用 std::cin 和 printf，也可以同时使用 scanf 和 std::cout。

tie

tie 是将两个 stream 绑定的函数，空参数的话返回当前的输出流指针。

在默认的情况下 std::cin 绑定的是 std::cout，每次执行 << 操作符的时候都要调用 flush() 来清理 stream buffer，这样会增加 IO 负担。可以通过 std::cin.tie(0)（0 表示 NULL）来解除 std::cin 与 std::cout 的绑定，进一步加快执行效率。

但需要注意的是，在解除了 std::cin 和 std::cout 的绑定后，程序中必须手动 flush 才能确保每次 std::cout 展现的内容可以在 std::cin 前出现。这是因为 std::cout 被 buffer 为默认设置。例如：

```
std::cout
    << "Please input your name: "
    << std::flush;  // 或者: std::endl;
                    // 因为每次调用 std::endl 都会 flush 输出缓冲区，而 \n 则不会。
// 但请谨慎使用，过多的 flush 会影响程序效率
std::cin >> name;
```

代码实现

```
std::ios::sync_with_stdio(false);
std::cin.tie(0);
// 如果编译开启了 C++11 或更高版本，建议使用 std::cin.tie(nullptr);
```

读入优化

scanf 和 printf 依然有优化的空间，这就是本章所介绍的内容——读入和输出优化。

- 注意，本页面中介绍的读入和输出优化均针对整型数据，若要支持其他类型的数据（如浮点数），可自行按照本页面介绍的优化原理来编写代码。

原理

众所周知，getchar 是用来读入 1 byte 的数据并将其转换为 char 类型的函数，且速度很快，故可以用「读入字符——转换为整型」来代替缓慢的读入

每个整数由两部分组成——符号和数字

整数的'+' 通常是省略的，且不会对后面数字所代表的值产生影响，而'-' 不可省略，因此要进行判定

10 进制整数中是不含空格或除 0~9 和正负号外的其他字符的，因此在读入不应存在于整数中的字符（通常为空格）时，就可以判定已经读入结束

C 和 C++ 语言分别在 ctype.h 和 cctype 头文件中，提供了函数 `isdigit`, 这个函数会检查传入的参数是否为十进制数字字符，是则返回 **true**，否则返回 **false**。对应的，在下面的代码中，可以使用 `isdigit(ch)` 代替 `ch >= '0' && ch <= '9'`，也可以使用 `!isdigit(ch)` 代替 `ch <'0' || ch> '9'`

代码实现

```
int read() {
  int x = 0, w = 1;
  char ch = 0;
  while (ch < '0' || ch > '9') {  // ch 不是数字时
    if (ch == '-') w = -1;        // 判断是否为负
    ch = getchar();               // 继续读入
  }
  while (ch >= '0' && ch <= '9') {  // ch 是数字时
    x = x * 10 + (ch - '0');  // 将新读入的数字「加」在 x 的后面
    // x 是 int 类型，char 类型的 ch 和 '0' 会被自动转为其对应的
    // ASCII 码，相当于将 ch 转化为对应数字
    // 此处也可以使用 (x<<3)+(x<<1) 的写法来代替 x*10
    ch = getchar();  // 继续读入
  }
  return x * w;  // 数字 * 正负号 = 实际数值
}
```

- 举例

读入 num 可写为 `num=read();`

输出优化

原理

同样是众所周知，`putchar` 是用来输出单个字符的函数

因此将数字的每一位转化为字符输出以加速

要注意的是，负号要单独判断输出，并且每次%（mod）取出的是数字末位，因此要倒序输出

代码实现

```
void write(int x) {
  if (x < 0) {  // 判负 + 输出负号 + 变原数为正数
    x = -x;
    putchar('-');
  }
  if (x > 9) write(x / 10);  // 递归，将除最后一位外的其他部分放到递归中输出
  putchar(x % 10 + '0');  // 已经输出（递归）完 x 末位前的所有数字，输出末位
}
```

但是递归实现常数是较大的，我们可以写一个栈来实现这个过程

```
void write(int x) {
  static int sta[35];
  int top = 0;
  do {
```

```
    sta[top++] = x % 10, x /= 10;
  } while (x);
  while (top) putchar(sta[--top] + 48);  // 48 是 '0'
}
```

- 举例

  输出 num 可写为 `write(num);`

更快的读入/输出优化

通过 `fread` 或者 `mmap` 可以实现更快的读入。

`fread` 能将需要的文件部分读入内存缓冲区。`mmap` 则会调度内核级函数，将文件一次性地映射到内存中，类似于可以指针引用的内存区域。所以在日常程序读写时，只需要重复读取部分文件可以使用 `fread`，因为如果用 `mmap` 反复读取一小块文件，做一次性内存映射并且内核处理 page fault 的花费会远比使用 `fread` 的内核级函数调度大。

一次性读入缓冲区的操作比逐个字符读入（`getchar`,`putchar`）要快的多。因为硬盘的多次读写速度是要慢于直接读取内存的，所以先一次性读到缓存区里再从缓存区读入要快的多。并且 `mmap` 确保了进程间自动共享，存储区如果可以也会与内核缓存分享信息，确保了更少的拷贝操作。

更通用的是 `fread`，因为 `mmap` 不能在 Windows 环境下使用（例如 CodeForces 的 tester）。

`fread` 类似于参数为 `"%s"` 的 `scanf`，不过它更为快速，而且可以一次性读入若干个字符（包括空格换行等制表符），如果缓存区足够大，甚至可以一次性读入整个文件。

对于输出，我们还有对应的 `fwrite` 函数

```
std::size_t fread(void* buffer, std::size_t size, std::size_t count,
                  std::FILE* stream);
std::size_t fwrite(const void* buffer, std::size_t size, std::size_t count,
                   std::FILE* stream);
```

使用示例：`fread(Buf, 1, SIZE, stdin)`，表示从 stdin 文件流中读入 SIZE 个大小为 1 byte 的数据块到 Buf 中。

读入之后的使用就跟普通的读入优化相似了，只需要重定义一下 getchar。它原来是从文件中读入一个 char，现在变成从 Buf 中读入一个 char，也就是头指针向后移动一位。

```
char buf[1 << 20], *p1, *p2;
#define gc()                                                        \
  (p1 == p2 && (p2 = (p1 = buf) + fread(buf, 1, 1 << 20, stdin), p1 == p2) \
       ? EOF                                                        \
       : *p1++)
```

`fwrite` 也是类似的，先放入一个 `OutBuf[MAXSIZE]` 中，最后通过 `fwrite` 一次性将 `OutBuf` 输出。

参考代码：

```
namespace IO {
const int MAXSIZE = 1 << 20;
char buf[MAXSIZE], *p1, *p2;
#define gc()                                                        \
  (p1 == p2 && (p2 = (p1 = buf) + fread(buf, 1, MAXSIZE, stdin), p1 == p2) \
       ? EOF                                                        \
       : *p1++)

int rd() {
  int x = 0, f = 1;
  char c = gc();
```

```
  while (!isdigit(c)) {
    if (c == '-') f = -1;
    c = gc();
  }
  while (isdigit(c)) x = x * 10 + (c ^ 48), c = gc();
  return x * f;
}

char pbuf[1 << 20], *pp = pbuf;

void push(const char &c) {
  if (pp - pbuf == 1 << 20) fwrite(pbuf, 1, 1 << 20, stdout), pp = pbuf;
  *pp++ = c;
}

void write(int x) {
  static int sta[35];
  int top = 0;
  do {
    sta[top++] = x % 10, x /= 10;
  } while (x);
  while (top) push(sta[--top] + '0');
}
}  // namespace IO
```

输入输出的缓冲

printf 和 scanf 是有缓冲区的。这也就是为什么，如果输入函数紧跟在输出函数之后/输出函数紧跟在输入函数之后可能导致错误。

刷新缓冲区

1. 程序结束
2. 关闭文件
3. printf 输出 \r 或者 \n 到终端的时候（注：如果是输出到文件，则不会刷新缓冲区）
4. 手动 fflush()
5. 缓冲区满自动刷新
6. cout 输出 endl

使输入输出优化更为通用

如果你的程序使用多个类型的变量，那么可能需要写多个输入输出优化的函数。下面给出的代码使用 C++ 中的 template[1] 实现了对于所有整数类型的输入输出优化。

```
// 声明 template 类，要求提供输入的类型 T，并以此类型定义内联函数 read()
template <typename T>
T read() {
  T sum = 0, fl = 1;  // 将 sum,fl 和 ch 以输入的类型定义
  int ch = getchar();
  for (; !isdigit(ch); ch = getchar())
    if (ch == '-') fl = -1;
  for (; isdigit(ch); ch = getchar()) sum = sum * 10 + ch - '0';
  return sum * fl;
```

```
}
```

如果要分别输入 `int` 类型的变量 a，`long long` 类型的变量 b 和 `__int128` 类型的变量 c，那么可以写成

```
a = read<int>();
b = read<long long>();
c = read<__int128>();
```

完整带调试版

关闭调试开关时使用 `fread()`,`fwrite()`，退出时自动析构执行 `fwrite()`。

开启调试开关时使用 `getchar()`,`putchar()`，便于调试。

若要开启文件读写时，请在所有读写之前加入 `freopen()`。

```
// #define DEBUG 1  // 调试开关
struct IO {
#define MAXSIZE (1 << 20)
#define isdigit(x) (x >= '0' && x <= '9')
  char buf[MAXSIZE], *p1, *p2;
  char pbuf[MAXSIZE], *pp;
#if DEBUG
#else
  IO() : p1(buf), p2(buf), pp(pbuf) {}

  ~IO() { fwrite(pbuf, 1, pp - pbuf, stdout); }
#endif
  char gc() {
#if DEBUG  // 调试，可显示字符
    return getchar();
#endif
    if (p1 == p2) p2 = (p1 = buf) + fread(buf, 1, MAXSIZE, stdin);
    return p1 == p2 ? ' ' : *p1++;
  }

  bool blank(char ch) {
    return ch == ' ' || ch == '\n' || ch == '\r' || ch == '\t';
  }

  template <class T>
  void read(T &x) {
    double tmp = 1;
    bool sign = 0;
    x = 0;
    char ch = gc();
    for (; !isdigit(ch); ch = gc())
      if (ch == '-') sign = 1;
    for (; isdigit(ch); ch = gc()) x = x * 10 + (ch - '0');
    if (ch == '.')
      for (ch = gc(); isdigit(ch); ch = gc())
        tmp /= 10.0, x += tmp * (ch - '0');
    if (sign) x = -x;
  }

  void read(char *s) {
```

```
    char ch = gc();
    for (; blank(ch); ch = gc())
      ;
    for (; !blank(ch); ch = gc()) *s++ = ch;
    *s = 0;
  }

  void read(char &c) {
    for (c = gc(); blank(c); c = gc())
      ;
  }

  void push(const char &c) {
#if DEBUG  // 调试，可显示字符
    putchar(c);
#else
    if (pp - pbuf == MAXSIZE) fwrite(pbuf, 1, MAXSIZE, stdout), pp = pbuf;
    *pp++ = c;
#endif
  }

  template <class T>
  void write(T x) {
    if (x < 0) x = -x, push('-');  // 负数输出
    static T sta[35];
    T top = 0;
    do {
      sta[top++] = x % 10, x /= 10;
    } while (x);
    while (top) push(sta[--top] + '0');
  }

  template <class T>
  void write(T x, char lastChar) {
    write(x), push(lastChar);
  }
} io;
```

参考

http://www.hankcs.com/program/cpp/cin-tie-with-sync_with_stdio-acceleration-input-and-output.html[2]

http://meme.biology.tohoku.ac.jp/students/iwasaki/cxx/speed.html[3]

https://marc.info/? l=linux-kernel&m=95496636207616&w=2[4]

参考资料与注释

[1] C++ 中的 `template`

[2] http://www.hankcs.com/program/cpp/cin-tie-with-sync_with_stdio-acceleration-input-and-output.html

[3] http://meme.biology.tohoku.ac.jp/students/iwasaki/cxx/speed.html

[4] https://marc.info/? l=linux-kernel&m=95496636207616&w=2

2.6.2 分段打表

前置知识：分块。

朴素的打表，指的是在比赛时把所有可能的输入对应的答案都计算出来并保存下来，然后在代码里开个数组把答案放里面，直接输出即可。

注意这个技巧只适用于输入的值域不大（如，输入只有一个数，而且范围很小）的问题，否则可能会导致代码过长、MLE、打表需要的时间过长等问题。

" 例题"

规定 $f(x)$ 为整数 x 的二进制表示中 1 的个数。输入一个正整数 $n(n \le 10^9)$，输出 $\sum_{i=1}^{n} f^2(i)$。

如果对于每一个 n，都输出 $f(n)$ 的话，除了可能会 MLE 外，还有可能代码超过最大代码长度限制，导致编译不通过。

我们考虑优化这个答案表。采用 分块 的思想，我们设置一个合理的步长 m（这个步长一般视代码长度而定），对于第 i 块，计算出：

$$\sum_{k=\frac{n}{m}(i-1)+1}^{\frac{ni}{m}} f^2(k)$$

的值。

然后输出答案时采用分块思想处理即可。即，整块的答案用预处理的值计算，非整块的答案暴力计算。

一般来说，这样的问题对于处理单个函数值 $f(x)$ 很快，但是需要大量函数值求和（求积或某些可以快速合并的操作），枚举会超出时间限制，在找不到标准做法的情况下，分段打表是一个不错的选择。

" 注意事项"

当上题中指数不是定值，但是范围较小，也可以考虑打表。

例题

「BZOJ 3798」 特殊的质数[1]：求 $[l, r]$ 区间内有多少个质数可以分解为两个正整数的平方和。

「Luogu P1822」 魔法指纹[2]

参考资料与注释

[1] 「BZOJ 3798」特殊的质数

[2] 「Luogu P1822」魔法指纹

2.6.3 常见错误

Authors: H-J-Granger, orzAtalod, ksyx, Ir1d, Chrogeek, Enter-tainer, yiyangit, shuzhouliu, broken-paint

本页面主要列举一些竞赛中很多人经常会出现的错误。

因环境不同导致的错误

- `scanf` 或 `printf` 使用 `%I64d` 格式指示符在 Linux 下可能导致输出格式错误。

会引起 CE 的错误

这类错误多为词法、语法和语义错误，引发的原因较为简单，修复难度较低。

例：

- `int main()` 写为 `int mian()` 之类的拼写错误。
- 写完 `struct` 或 `class` 忘记写分号。
- 数组开太大，（在 OJ 上）使用了不合法的函数（例如多线程），或者函数声明但未定义，会引起链接错误。
- 函数参数类型不匹配。
  - 示例：如使用 `<algorithm>` 头文件中的 `max` 函数时，传入了一个 `int` 类型参数和一个 `long long` 类型参数。

```
// query 为返回 long long 类型的自定义函数
printf("%lld\n", max(0, query(1, 1, n, l, r));

//错误    没有与参数列表匹配的重载函数 "std::max" 实例
```

- 使用 `goto` 和 `switch-case` 的时候跳过了一些局部变量的初始化。

不会引起 CE 但会引起 Warning 的错误

犯这类错误时写下的程序虽然能通过编译，但大概率会得到错误的程序运行结果。这类错误会在使用 `-W{warningtype}` 参数编译时被编译器指出。

- 赋值运算符 `=` 和比较运算符 `==` 不分。
  - 示例：

```
std::srand(std::time(nullptr));
int n = std::rand();
if (n = 1)
  printf("Yes");
else
  printf("No");

// 无论 n 的随机所得值为多少，输出肯定是 Yes
// 警告    运算符不正确：在 Boolean 上下文中执行了常量赋值。应考虑改用「==」。
```

  - 如果确实想在原应使用 `==` 的语句里使用 `=`（比如 `while (foo = bar)`），又不想收到 Warning，可以使用**双括号**：`while ((foo = bar))`。
- 由于运算符优先级产生的错误。
  - 示例：

```
// 错误
// std::cout << (1 << 1 + 1);
// 正确
std::cout << ((1 << 1) + 1);

// 警告    「<<」：检查运算符优先级是否有可能的错误；使用括号阐明优先级
```

- 不正确地使用 `static` 修饰符。
- 使用 `scanf` 读入的时候没加取地址符 `&`。
- 使用 `scanf` 或 `printf` 的时候参数类型与格式指定符不符。
- 同时使用位运算和逻辑运算符 `==` 并且未加括号。

  - 示例：`(x >> j) & 3 == 2`
- `int` 字面量溢出。
  - 示例：`long long x = 0x7f7f7f7f7f7f7f7f`，`1<<62`。
- 未初始化局部变量。

" 未初始化变量会发生什么"

原文：https://loj.ac/d/3679[1] by @hly1204

例如我们在 C++ 中声明一个 `int a;` 但不初始化，可能有时候会认为 `a` 是一个「随机」（其实可能不是真的随机）的值，但是可能将其认为是一个固定的值，但实际上并非如此。

我们在简单的测试代码中

https://wandbox.org/permlink/T2uiVe4n9Hg4EyWT[2]

代码是：

```cpp
#include <iostream>

int main() {
  int a;
  std::cout << std::boolalpha << (a < 0 || a == 0 || a > 0);
  return 0;
}
```

在一些编译器和环境上开启优化后，其输出为 false。

有兴趣的话可以看 https://www.ralfj.de/blog/2019/07/14/uninit.html[3]，尽管其是用 Rust 做的实验，但是本质是一样的。

- 局部变量与全局变量重名，导致全局变量被意外覆盖。（开 `-Wshadow` 就可检查此类错误。）
- 运算符重载后引发的输出错误。
  - 示例：

```cpp
// 本意：前一个 << 为重载后的运算符，表示输出；后一个 << 为移位运算符，表示将 1
// 左移 1 位。但由于忘记加括号，导致编译器将后一个 <<
// 也判作输出运算符，而导致输出的结果与预期不同。错误 std::cout << 1 << 1; 正确
std::cout << (1 << 1);
```

既不会引起 CE 也不会引发 Warning 的错误

这类错误无法被编译器发现，仅能自行查明。

会导致 WA 的错误

- 上一组数据处理完毕，读入下一组数据前，未清空数组。
- 读入优化未判断负数。
- 所用数据类型位宽不足，导致溢出。
  - 如习语「三年 OI 一场空，不开 `long long` 见祖宗」所描述的场景。选手因为没有在正确的地方开 `long long`（将整数定义为 `long long` 类型），导致得出错误的答案而失分。
- 存图时，节点编号 0 开始，而题目给的边中两个端点的编号从 1 开始，读入的时候忘记 -1。
- 大/小于号打错或打反。
- 在执行 `ios::sync_with_stdio(false);` 后混用 `scanf/printf` 和 `std::cin/std::cout` 两种 IO，导致输入/输出错乱。

– 示例：

```cpp
// 这个例子将说明关闭与 stdio 的同步后，混用两种 IO 方式的后果
// 建议单步运行来观察效果
#include <cstdio>
#include <iostream>

int main() {
  // 关闭同步后，cin/cout 将使用独立缓冲区，而不是将输出同步至 scanf/printf
  // 的缓冲区，从而减少 IO 耗时
  std::ios::sync_with_stdio(false);
  // cout 下，使用'\n'换行时，内容会被缓冲而不会被立刻输出
  std::cout << "a\n";
  // printf 的 '\n' 会刷新 printf 的缓冲区，导致输出错位
  printf("b\n");
  std::cout << "c\n";
  // 程序结束时，cout 的缓冲区才会被输出
  return 0;
}
```

– 特别的，也不能在执行 `ios::sync_with_stdio(false);` 后使用 `freopen`。

- 由于宏的展开，且未加括号导致的错误。
  – 示例：该宏返回的值并非 $4^2 = 16$ 而是 $2 + 2 \times 2 + 2 = 8$。

```cpp
#define square(x) x* x
printf("%d", square(2 + 2));
```

- 哈希的时候没有使用 `unsigned` 导致的运算错误。
  – 对负数的右移运算会在最高位补 1。参见：位运算
- 没有删除或注释掉调试输出语句。
- 误加了 `;`。
  – 示例：

```cpp
/* clang-format off */
while (1);
    printf("OI Wiki!\n");
```

- 哨兵值设置错误。例如，平衡树的 `0` 节点。
- 在类或结构体的构造函数中使用 `:` 初始化变量时，变量声明顺序不符合初始化时候的依赖关系。
  – 成员变量的初始化顺序与它们在类中声明的顺序有关，而与初始化列表中的顺序无关。参见：构造函数与成员初始化器列表[4] 的「初始化顺序」
  – 示例：

```cpp
#include <iostream>

class Foo {
 public:
  int a, b;

  // a 将在 b 前初始化，其值不确定
  Foo(int x) : b(x), a(b + 1) {}
};
```

```cpp
int main() {
  Foo bar(1, 2);
  std::cout << bar.a << ' ' << bar.b;
}

// 可能的输出结果：-858993459 1
```

- 并查集合并集合时没有把两个元素的祖先合并。
  - 示例：

```cpp
f[a] = b;              // 错误
f[find(a)] = find(b);  // 正确
```

换行符不同

warning

在正式比赛中会尽量保证选手答题的环境和最终测试的环境相同。

本节内容仅适用于模拟赛等情况，而我们也建议出题人尽量让数据符合数据格式。

不同的操作系统使用不同的符号来标记换行，以下为几种常用系统的换行符：

- LF（用 `\n` 表示）：`Unix` 或 `Unix` 兼容系统
- CR+LF（用 `\r\n` 表示）：`Windows`
- CR（用 `\r` 表示）：`Mac OS` 至版本 9

而 C/C++ 利用转义序列 `\n` 来换行，这可能会导致我们认为输入中的换行符也一定是由 `\n` 来表示，而只读入了一个字符来代表换行符，这就会导致我们没有完全读入输入文件。

以下为解决方案：

- 多次 `getchar()`，直到读到想要的字符为止。
- 使用 `cin` 读入，**这可能会增大代码常数**。
- 使用 `scanf("%s",str)` 读入一个字符串，然后取 `str[0]` 作为读入的字符。
- 使用 `scanf(" %c",&c)` 过滤掉所有空白字符。

会导致未知的结果

未定义行为会导致未知的结果，可能是 WA，RE 等。编译器通常会假定你的程序不会出现未定义行为，因此出现开 O2 与不开 O2 代码行为不一致的情况。

- 除以 0（求 0 的逆元）

  " 示例"

```cpp
cout << x / 0 << endl;
```

- 数组（下标）越界

  例如：

  - 未正确设置循环的初值导致访问了下标为 -1 的值。
  - 无向图边表未开 2 倍。
  - 线段树未开 4 倍空间。
  - 看错数据范围，少打一个零。
  - 错误预估了算法的空间复杂度。

  - 写线段树的时候，pushup 或 pushdown 叶节点。

    正确的做法：不要越界，记得检查自己的代码，使得下标访问数 x，在定义的下标中。

- 除 main 外有返回值函数执行至结尾未执行任何 return 语句

  即使有一个分支有返回值，但是其他分支却没有，结果也是未定义的。

  可以向编译选项中追加 -Wall，检查编译器是否给出有关于函数未 return 的警告。

- 尝试修改字符串字面量

  " 示例"

  ```
  char *p = "OI-wiki";
  p[0] = 'o';
  p[1] = 'i';
  ```

  这样试图修改字符串字面量会导致**未定义行为**，应当使用其他**合适**的数据类型，例如 std::string 和 char[]。

- 多次释放/非法解引用一片内存

  例如：

  - 未初始化就解引用指针。
  - 指针指向的内存区域已经释放。

    使用 erase 或 delete 或 free 操作应注意不要对同一地址/对象多次使用。

- 尝试释放由 new [] 分配的整块内存的一部分

  例如：

  ```
  object *pool = new object[POOL_SIZE];

  object *pointer = pool + 10;

  // 报错!
  delete pointer;
  ```

  常见于使用内存池提前分配整块内存后，试图使用 delete 或 free() 释放从内存池中获取的单个对象。

- 解引用空指针/野指针

  对于空指针：先应该判断空指针，可以用 p == nullptr 或 !p。

  对于野指针：可以释放指针的时候将其置为 nullptr 以规避。

- 有符号数溢出

  例如我们有一个表达式 x+1 > x。

  正常输出应当是 true，但是在 INT_MAX 作为 x 时输出 false，这时称为 signed integer overflow。

  可以使用更大的数据类型（例如 long long 或 __int128），或判断溢出。若保证无负数，亦可使用无符号整型。

  有符号整数溢出可能影响编译优化，例如代码：

  ```
  int foo(int x) {
    if (x > x + 1) return 1;
    return 0;
  }
  ```

  可能被编译器直接优化为：

  ```
  int foo(int x) { return 0; }
  ```

  因为编译器可以假定有符号整数永远不会溢出，因此 x > x + 1 恒成立。

- 使用未初始化的变量

" 示例"

```
int foo(int a) {
  int t; /* 没有初始化 */
  if (/* 使用 */ t > 3) return a;
  return 0;
}
```

会导致 RE

- 没删文件操作（某些 OJ）。
- 排序时比较函数的错误 `std::sort` 要求比较函数是严格弱序：`a<a` 为 `false`；若 `a<b` 为 `true`，则 `b<a` 为 `false`；若 `a<b` 为 `true` 且 `b<c` 为 `true`，则 `a<c` 为 `true`。其中要特别注意第二点。如果不满足上述要求，排序时很可能会 RE。例如，编写莫队的奇偶性排序时，这样写是错误的：

```
bool operator<(const int a, const int b) {
  if (block[a.l] == block[b.l])
    return (block[a.l] & 1) ^ (a.r < b.r);
  else
    return block[a.l] < block[b.l];
}
```

上述代码中 `(block[a.l]&1)^(a.r<b.r)` 不满足上述要求的第二点。改成这样就正确了：

```
bool operator<(const int a, const int b) {
  if (block[a.l] == block[b.l])
    // 错误：不满足严格弱序的要求
    // return (block[a.l] & 1) ^ (a.r < b.r);
    // 正确
    return (block[a.l] & 1) ? (a.r < b.r) : (a.r > b.r);
  else
    return block[a.l] < block[b.l];
}
```

- Windows 下栈空间不足，导致栈空间溢出，Windows 向程序发出 SIGSEGV 信号，程序终止并返回 3221225725（即 0xC00000FD, NTSTATUS 定义为 `STATUS_STACK_OVERFLOW`）。

  若使用 gcc 编译器，可在编译时加入命令 `-Wl,--stack=SIZE` 以扩展栈空间，其中 `SIZE` 为栈空间大小字节数。

会导致 TLE

- 分治未判边界导致死递归。
- 死循环。
  - 循环变量重名。
  - 循环方向反了。
- BFS 时不标记某个状态是否已访问过。
- 使用宏展开编写 min/max

  这种错误会大大增加程序的运行时间，甚至直接影响代码的时间复杂度。在初学者写线段树时尤为多见。

  常见的错误写法是这样的：

```
#define Min(x, y) ((x) < (y) ? (x) : (y))
#define Max(x, y) ((x) > (y) ? (x) : (y))
```

这样写虽然在正确性上没有问题，但是如果直接对函数的返回值取 max，如 `a = Max(func1(), func2())`，而这个函数的运行时间较长，则会大大影响程序的性能，因为宏展开后是 `a = func1() > func2() ? func1() : func2()` 的形式，调用了三次函数，比正常的 max 函数多调用了一次。注意，如果 `func1()` 每次返回的答案不一样，还会导致这种 `max` 的写法出现错误。例如 `func1()` 为 `return ++a;` 而 `a` 为全局变量的情况。

示例：如下代码会被卡到单次查询 $\Theta(n)$ 导致 TLE。

```cpp
#define max(x, y) ((x) > (y) ? (x) : (y))

int query(int t, int l, int r, int ql, int qr) {
  if (ql <= l && qr >= r) {
    ++ti[t];  // 记录结点访问次数方便调试
    return vi[t];
  }

  int mid = (l + r) >> 1;
  if (mid >= qr) return query(lt(t), l, mid, ql, qr);
  if (mid < ql) return query(rt(t), mid + 1, r, ql, qr);
  return max(query(lt(t), l, mid, ql, qr), query(rt(t), mid + 1, r, ql, qr));
}
```

- 使用 + 运算符向 `std::string` 类字符串追加字符

  这种错误会创建一个临时 `string` 变量，修改完成后再赋值给原变量。这种错误无法被编译器优化，在数据量大的情况下可能会导致时间复杂度退化。

  常见错误写法：

```cpp
std::string a;
char b = 'c';
a = a + b;
```

  当执行这段代码时，程序首先会创建一个临时 `string` 变量，随后将 `a` 的值存入临时变量，然后在末尾添加 `b` 的值，最后再存入 `a`。

  从 汇编结果[5] 可以看出，`a = a + b` 调用了三次 `std::__cxx11::basic_string` 中的功能，分别为 `operator+`、`operator=` 和创建变量。

  正确写法应该是：

```cpp
std::string a;
char b = 'c';
a += b;
```

  这种写法[6] 会直接将字符 `b` 附加到字符串 `a` 中，仅调用了一次 `operator+=`。更详细的性能比较可参考 Benchmark[7]。

- 没删文件操作（某些 OJ）。
- 在 `for/while` 循环中重复执行复杂度非 $O(1)$ 的函数。严格来说，这可能会引起时间复杂度的改变。

会导致 MLE

- 数组过大。
- STL 容器中插入了过多的元素。
  - 经常是在一个会向 STL 插入元素的循环中死循环了。
  - 也有可能被卡了。

会导致常数过大

- 定义模数的时候，未定义为常量。

  - 示例：

```
// int mod = 998244353;      // 错误
const int mod = 998244353;  // 正确，方便编译器按常量处理
```

- 使用了不必要的递归（尾递归不在此列）。
- 将递归转化成迭代的时候，引入了大量额外运算。

只在程序在本地运行的时候造成影响的错误

- 文件操作有可能会发生的错误：
  - 对拍时未关闭文件指针 `fclose(fp)` 就又令 `fp = fopen()`。这会使得进程出现大量的文件野指针。
  - `freopen()` 中的文件名未加 `.in`/`.out`。
- 使用堆空间后忘记 `delete` 或 `free`。

参考资料与注释

[1] https://loj.ac/d/3679

[2] https://wandbox.org/permlink/T2uiVe4n9Hg4EyWT

[3] https://www.ralfj.de/blog/2019/07/14/uninit.html

[4] 构造函数与成员初始化器列表

[5] 汇编结果

[6] 这种写法

[7] Benchmark

2.6.4 常见技巧

Authors: H-J-Granger, Ir1d, ChungZH, Marcythm, StudyingFather, billchenchina, Suyun514, Psycho7, greyqz, Xeonacid, partychicken

本页面主要列举一些竞赛中的小技巧。

利用局部性

局部性是指程序倾向于引用邻近于其他最近引用过的数据项的数据项，或者最近引用过的数据项本身。局部性分为时间局部性和空间局部性。

具体可参见 循环展开（Loop Unroll）、代码布局优化（Code Layout Optimizations）等内容

循环宏定义

如下代码可使用宏定义简化：

```
for (int i = 0; i < N; i++) {
  // 循环内容略
}

// 使用宏简化
#define f(x, y, z) for (int x = (y), __ = (z); x < __; ++x)

// 这样写循环代码时，就可以简化成 `f(i, 0, N)` 。例如：
// a is a STL container
f(i, 0, a.size()) { ... }
```

另外推荐一个比较有用的宏定义：

```
#define _rep(i, a, b) for (int i = (a); i <= (b); ++i)
```

善用 namespace

使用 namespace 能使程序可读性更好，便于调试。

" **例题**：NOI 2018 **屠龙勇士**"

```
// NOI 2018 屠龙勇士 40 分部分分代码
#include <algorithm>
#include <cmath>
#include <cstring>
#include <iostream>
using namespace std;
long long n, m, a[100005], p[100005], aw[100005], atk[100005];

namespace one_game {
// 其实 namespace 里也可以声明变量
void solve() {
  for (int y = 0;; y++)
    if ((a[1] + p[1] * y) % atk[1] == 0) {
      cout << (a[1] + p[1] * y) / atk[1] << endl;
      return;
    }
}
}  // namespace one_game

namespace p_1 {
void solve() {
  if (atk[1] == 1) {  // solve 1-2
    sort(a + 1, a + n + 1);
    cout << a[n] << endl;
    return;
  } else if (m == 1) {  // solve 3-4
    long long k = atk[1], kt = ceil(a[1] * 1.0 / k);
    for (int i = 2; i <= n; i++)
      k = aw[i - 1], kt = max(kt, (long long)ceil(a[i] * 1.0 / k));
    cout << k << endl;
  }
}
}  // namespace p_1
```

```cpp
int main() {
  int T;
  cin >> T;
  while (T--) {
    memset(a, 0, sizeof(a));
    memset(p, 0, sizeof(p));
    memset(aw, 0, sizeof(aw));
    memset(atk, 0, sizeof(atk));
    cin >> n >> m;
    for (int i = 1; i <= n; i++) cin >> a[i];
    for (int i = 1; i <= n; i++) cin >> p[i];
    for (int i = 1; i <= n; i++) cin >> aw[i];
    for (int i = 1; i <= m; i++) cin >> atk[i];
    if (n == 1 && m == 1)
      one_game::solve();  // solve 8-13
    else if (p[1] == 1)
      p_1::solve();  // solve 1-4 or 14-15
    else
      cout << -1 << endl;
  }
  return 0;
}
```

使用宏进行调试

编程者在本地测试的时候，往往要加入一些调试语句。而在需要提交到 OJ 时，为了不使调试语句的输出影响到系统对程序输出结果的判断，就要把它们全部删除，耗时较多。这种情况下，可以通过定义宏的方式来节省时间。大致的程序框架是这样的：

```cpp
#define DEBUG
#ifdef DEBUG
// do something when DEBUG is defined
#endif
// or
#ifndef DEBUG
// do something when DEBUG isn't defined
#endif
```

`#ifdef` 会检查程序中是否有 `#define` 定义的对应标识符，如果有定义，就会执行后面的语句。而 `#ifndef` 会在没有定义相应标识符的情况下执行后面的语句。

这样，只需在 `#ifdef DEBUG` 里写好调试用代码，`#ifndef DEBUG` 里写好真正提交的代码，就能方便地进行本地测试。提交程序的时候，只需要将 `#define DEBUG` 一行注释掉即可。也可以不在程序中定义标识符，而是通过 `-DDEBUG` 的编译选项在编译的时候定义 `DEBUG` 标识符。这样就可以在提交的时候不用修改程序了。

不少 OJ 都开启了 `-DONLINE_JUDGE` 这一编译选项，善用这一特性可以节约不少时间。

对拍

对拍是一种进行检验或调试的方法，通过对比两个程序的输出来检验程序的正确性。可以将自己程序的输出与其他程序的输出进行对比，从而判断自己的程序是否正确。

对拍过程要多次进行，因此需要通过批处理的方法来实现对拍的自动化。

具体而言，对拍需要一个 数据生成器 和两个要进行输出结果比对的程序。

每运行一次数据生成器都将生成的数据写入输入文件，通过重定向的方法使两个程序读入数据，并将输出写入指

定文件，最后利用 Windows 下的 `fc` 命令比对文件（Linux 下为 `diff` 命令）来检验程序的正确性。如果发现程序出错，可以直接利用刚刚生成的数据进行调试。

对拍程序的大致框架如下：

```cpp
#include <stdio.h>
#include <stdlib.h>

int main() {
  // For Windows
  // 对拍时不开文件输入输出
  // 当然，这段程序也可以改写成批处理的形式
  while (true) {
    system("gen > test.in");  // 数据生成器将生成数据写入输入文件
    system("test1.exe < test.in > a.out");  // 获取程序 1 输出
    system("test2.exe < test.in > b.out");  // 获取程序 2 输出
    if (system("fc a.out b.out")) {
      // 该行语句比对输入输出
      // fc 返回 0 时表示输出一致，否则表示有不同处
      system("pause");  // 方便查看不同处
      return 0;
      // 该输入数据已经存放在 test.in 文件中，可以直接利用进行调试
    }
  }
}
```

内存池

当动态分配内存时，频繁使用 `new/malloc` 会占用大量的时间和空间，甚至生成大量的内存碎片从而降低程序的性能，可能会使原本正确的程序 TLE/MLE。

这时候需要使用到「内存池」这种技巧：在真正使用内存之前，先申请分配一定大小的内存作为备用。当需要动态分配时直接从备用内存中分配一块即可。

在大多数 OI 题当中，可以预先算出需要使用到的最大内存并一次性申请分配。

示例：

```cpp
// 申请动态分配 32 位有符号整数数组：
int* newarr(int sz) {
  static int pool[maxn], *allocp = pool;
  return allocp += sz, allocp - sz;
}

// 线段树动态开点的代码：
Node* newnode() {
  static Node pool[maxn << 1], *allocp = pool - 1;
  return ++allocp;
}
```

参考资料

洛谷日报 #86[1]

《算法竞赛入门经典习题与解答》

参考资料与注释

[1] 洛谷日报 #86

2.7 出题

> **Authors:** ouuan, Henry-ZHR, StudyingFather, ChungZH, xyf007

出题前的准备

具备一定的水平

一方面，一个人自己出题，很难出出难度大于自身水平的题目，一定的 OI 水平有助于想到更加优质的 idea 并想出优秀的做法；另一方面，OI 水平在一定程度上代表着 OI 资历，见识过更多的题目的选手也会对「好题」拥有自己的见解。

抱有认真负责的态度

出题是给别人做的，比起展示自己，更多是为了是服务他人。算法竞赛是选手之间的竞赛，而不是出题人与做题人之间的较量。因此，出题不应以考倒选手为目标（当然，适当的防 AK 与良好的区分度也是非常重要的），而应当让选手能在比赛中有所收获。花费足够的时间精力去学习如何出题并认真负责地出题非常重要。

做好耗费大量时间的准备

如果想要认真地出题，就必然要花费大量的时间。如果不做好心理准备，可能导致比赛准备匆忙，质量不过关，也可能在事后由于没有将时间花费在学习上而懊悔。但出题也可以带来很多美好的回忆，如果真的对出题抱有兴趣，并做好了充分的心理准备，出题带来的收获也能够弥补那些花费的时间。

认真阅读本文的内容

本文从如何出题、如何把题出好两个方面对整个出题流程进行了介绍。对于想要出题的人来说，认真阅读本文一定能够受益匪浅。

题目内容

出一道题，idea，即题目本质的内容，是题目的灵魂，也是出题的第一步。

idea 的来源

1. 受到已有题目的启发（但不能照搬或无意义地加强，如：序列题目搬到仙人掌上）。
2. 受到学过的知识点的启发（但不能毫无联系地拼凑知识点）。
3. 从生活/游戏中受到启发（但注意不要把游戏出成大模拟）。
4. 不知道为什么，就是想到了一道题。

什么样的 idea 是不好的

关于原题

原题大致可分为完全一致、几乎一致和做法一致三种。

- 完全一致：使用一题的 AC 代码可以 AC 另一题。
- 几乎一致：由一题的 AC 代码改动至另一题的 AC 代码可以由一个不会该题的人完成。
- 做法一致：核心思路、做法一致，但代码实现上、不那么关键的细节上有差异。

这三种原题自下而上为包含关系。

以下情况不应出现：

1. 在明知有「几乎一致」的原题的情况下出原题。
2. 由于未使用搜索引擎查找导致自己不清楚有原题，从而出了「几乎一致」的原题。
3. 在「做法一致」的原题广为人知（如：NOIP、NOI 原题）时出原题。
4. 在带有选拔性的考试的非送分题中出现「做法一致」的原题。

以下情况最好不要出现：

1. 在明知有至少为「做法一致」的原题的情况下出原题。
2. 由于未使用搜索引擎查找导致自己不清楚有原题，从而出了「做法一致」的原题。
3. 在任何情况下出「几乎一致」的原题。

可以放宽要求的例外情况：

1. 校内模拟赛。
2. 以专题训练为目的的模拟赛。
3. 难度较低的比赛，或是定位为送分题的题目。

关于毒瘤题

「毒瘤题」是一个非常模糊而主观的观念，在这只是引用一些前人关于此的探讨，加以自己的一些理解。这个话题是非常开放的，欢迎大家来发表自己的观点。

> 一道好题不应该是两道题拼在一起，一道好题会有自己的 idea——而它应该不加过多包装地突出这个 idea。
>
> 一道好题应该新颖。真正的好题，应该是能让人脑洞出新的好题的好题。
>
> ——vfk《UOJ 精神之源流》

例子：「XR-1」柯南家族[1]，做法的前后两部分完全割裂，前半部分为「模板」树上后缀排序[2]，后半部分是经典树上问题。就算是随意输入树的点权，依然可以做第二部分，前后部分没有联系。

> 一类 OI 题以数学为主，无论是题目描述还是做法都是数学题的特征，并且解法中不含算法相关的知识点，这类 OI 题目统称为纯数学题。
>
> ——王天懿 《论偏题的危害》

经典例子：NOIP2017 小凯的疑惑[3]

OI 中的数学题与其它数学题的区别，也是体现 OI 本质的一个特点，是 OI 中的数学题往往重点不在答案**是什么**，而在如何**加快**答案的计算。如果一道题考察的重点是「怎么算」而非「怎么快速计算」，这样的数学题一般都是不适合出在 OI 中的。

> 一部分偏题中牵涉到了大学物理的内容，导致选手在面对这些从未接触过物理知识点时变得不知所措，造成了知识上的隔膜。
>
> ——王天懿 《论偏题的危害》

经典例子：「清华集训 2015」多边形下海[4]

不止是物理，OI 题目中不应过多涉及到其它学科的知识，如果涉及应当给予详细的解释，不应使其它学科的知

识作为解题的重大障碍。

> 一道好题无论难度如何，都应该具有自己的思维难度，需要选手去思考并发现一些性质。
>
> 一道好题的代码可以长，但一定不是通过强行嵌套或者增加条件而让代码变长，而是长得自然，让人感觉这个题的代码就应该是这么长。
>
> ——王天懿 《论偏题的危害》

经典例子：「SDOI2010」 猪国杀[5]，「集训队互测 2015」 未来程序 · 改[6]

在一般的 OI 比赛中，思维难度应占主要部分。当然，如 THUWC/THUSC 的 Day 2+ 那样的工程题也有其存在的道理——毕竟体验营的目的除了考察选手的算法设计能力，还有和大学学习对接的工程代码以及文档学习能力。但在一般的 OI 比赛中，考察更多的应当还是算法设计与思维能力。

题面

使用 LaTeX 书写公式

网上有很多 LaTeX 的教程，如：

- LaTeX 入门
- LaTeX 数学公式大全[7]
- LaTeX 各种命令，符号[8]

使用时请注意 LaTeX 公式的格式要求。

题目背景

题目背景最好尽量简短。在题目背景较长时，应当与题目描述分开。

需要绝对避免题目背景严重影响题意的理解。

必要时，可以提供与背景结合的题目描述与简洁的题目描述两个版本。

题目描述

简而言之，题目描述需要**清晰易懂**。

题面中的每个可能不被理解的定义都应得到解释，不应凭空冒出未加定义的概念。例如：在 CF1172D Nauuo and Portals[9] 中，你必须在题面中解释什么是「传送门」。

题面中涉及到的每个概念应当使用单一的词汇来描述。例如：不应一会儿说「费用」，一会儿说「代价」。

不应不加说明地使用与原义、常见义不同的词汇。例如：不应不加说明地用「路径」代指一条边。

你需要保证你的题面不会自相矛盾。例如：在 CF1173A Nauuo and Votes[10] 中，没有把”?” 作为一种”result”，是因为”?” 的含义是”there are more than one possible results”。

你需要保证你的题面不能被错误理解而自圆其说，即使这种理解是反常识、没有人会这么去想的。例如：在 CF1172D Nauuo and Portals[9] 中，之所以要繁琐地定义”walk into” 并与”teleport” 区分，是为了防止这种理解：通过传送门可以到另一个传送门，而到了传送门会传送，因此会反复横跳。

顺着读题目描述应当能看懂每一句话，并理解题目的任务与要求。至少在紧接着的下一段话中疑惑能够得到解释，而不是需要在若干段后才能得到解释，或者要看了输入输出格式才能明白题意，甚至需要根据样例来猜题意。例如：在「GuOJ Round #1」 琪露诺的冰雪宴会[11] 中，在输出格式才第一次出现了题目的目标「雾之湖最终能接收到的最大水量」，再加上「灵梦当然能很快算出来清理完全部小溪的总费用是多少」这句带有误解性质的话，更容易使人读错题意，这是不可取的，应当在题目描述中就对题目的目标进行说明。（在这个例子中还存在题目背景严重影响题意理解的问题。）相同的错误还出现在 CF1423(4)N Bubblesquare Tokens[12] 中，在输出格式才第一次出现了题目的目标”friend pairs and number of tokens each of them gets on behalf of their friendship”。

输入输出格式

输入输出格式清晰**完整**即可，没有死板的要求，个人建议参照 CF 的题目来写输入输出格式，具体可以参考CF 出题人须知。

为了方便选手做题，输入输出格式中最好说明每个变量的具体含义，除非变量的意义非常长，没法一句话说清楚（这时可以说「意义见题目描述」）。

需要特别注意的是，如果输出中含有小数，请尽量使用 SPJ 来对误差的大小进行限制，而非要求「保留 x 位小数」。

「保留 x 位小数」对精度的要求可能是无限的。例如：要求保留三位小数，实际答案为 0.0015，此时只要有任意大小的误差导致计算出的答案小于 0.0015，即使计算出的答案是 $0.00149999\cdots$ 也会输出错误的答案。

如果无法使用 SPJ，请保证对精度的要求是有限的，例如：请输出答案四舍五入后保留小数点后三位的结果。令标准答案为 ans，数据保证对于任意满足 $\frac{|x-ans|}{\max(1,ans)}<10^{-9}$ 的 x，四舍五入后结果与 ans 四舍五入后相同。

可以参考的一些句子：

```
输入的第一行包含三个正整数 $n$, $m$, $k$ ($1\le n,m\le 2\cdot 10^5$, $1\le k\le
100$) — $n$ 表示数列的长度，$m$ 表示操作个数，$k$ 的意义见题目描述。
```

```
输入的第二行包含 $n$ 个非负整数 $a_1,a_2,\ldots,a_n$ ($1\le a_i\le 10^9$) —题目
给出的数列。
```

```
接下来的 $m$ 行中的第 $i$ 行包含两个正整数 $l_i$ 和 $r_i$ ($1\le l_i\le r_i\le n
$)，表示第 $i$ 次操作在区间 $[l_i,r_i]$ 上进行。
```

```
接下来的 $n-1$ 行，每行包含两个正整数 $u$ 和 $v$ ($1\le u,v\le n$)，表示 $u$ 和 $v
$ 之间由一条边相连。

数据保证给出的边能构成一棵树。
```

```
输入的唯一一行包含一个由小写英文字母构成的非空字符串，其长度不超过 $10^6$。
```

```
输入的第二行包含一个小数点后不超过三位的实数 $x$ ($-10^6\le x\le 10^6$)，意义见题
目描述。
```

```
输出包含一个实数，当你的输出与标准答案之间的绝对误差或相对误差小于 $10^{-6}$ 时视
作正确。
```

```
输出的第二行包含 $n$ 个正整数，表示你构造的一组方案 —其中第 $i$ 个数表示你打出的
第 $i$ 张牌的编号。

如果有多组合法的答案，可以任意输出其中一组。
```

" 在选手代码内由随机数生成器生成输入数据"

有的题目会因为输入数据过大，为了防止读入用时过长，而要求选手在代码内通过给定的数据生成器生成数据，代替通过标准输入或文件输入来读入数据。

采用这种做法需要谨慎考虑，因为它有很多缺点：

- 可能引入了正解所不需要的数据随机性，或者使得构造数据变得困难
- 可能增大了理解输入格式的难度
- 如果随机数生成器封装的不好，可能理解数据生成器本身的使用方法就有难度
- 如果选手没有使用出题者推荐的语言，可能需要自己写一个数据生成器

采用这种做法一般是为了防止读入数据用时过长，所以一个可能的替代方案是下发一个性能足够好的 读入、输出优化 模板，以尽量保证所有人的读入用时一致，这样的话即使读入用时很久也不会影响不同选手用时的差异。另一个解决方案是将题目包装成函数调用式（而非 IO 式）交互题，即使算法过程中没有交互，交互题也可以起到统一读入用时的作用，IOI 就采用了所有题目都是交互题的方案。但是，这两种方案都对选手使用的语言有限制，需要出题者手动支持每种允许选手使用的语言。

回到问题的本源，还可以考虑一下过大的输入数据是否是必要的，有没有可能使用较小的输入数据达到目的，以及比正解复杂度稍劣的做法是否有卡掉的必要。

数据范围

按照 CF 的要求，数据范围要写在输入格式里，但在国内，数据范围往往是写在题目的最后的。

数据范围中最容易犯的错误就是不完整。输入中的每一个数、每一个字符串都应该有清晰的界定。在上文所给出的输入输出格式示例中就有一些数据范围的正确写法。

数据范围的常见遗漏：

1. 「整数」中的「整」。
2. 题面中只说了是「整数」没说是「正整数」，并且数据范围中只有上限没有下限。
3. 字符串没说字符集。
4. 实数没说小数点后位数。
5. 某些变量没有给范围。

你需要保证标程可以通过满足题面所述数据范围的**任何一组数据**。

" 关于「保证数据随机生成」"

有的题目中会「保证数据随机生成」，很多时候这样的限制并不是最优的解决方案，因为「随机生成」对数据的限制并不明确，会给判断具体数据范围、提供 hack 数据带来困难。

一般来说，「保证数据随机生成」可以换成解法所需要的数据性质。例如，随机生成一棵树往往可以换成限制树的高度。

如果一定要保证数据随机生成，应当指定随机生成的具体操作。例如，生成一棵树是随机选择父亲节点还是随机生成 Prüfer 序列。

需要注意的是，非确定性算法和依赖于数据随机性的算法是不同的。前者可以对于任意数据都有很高的概率得到正解，而后者是对于大部分的数据能得到正解，对于某些特定的数据则不可能得到正解。

样例

样例应当有一定的强度，能够查出一些简单的错误。读错题意的人应当能够通过样例发现自己读错了题意。

有多种操作的题，每种操作都应在样例中出现。

有多种输出的题（如 CF1173A Nauuo and Votes[10]），每种输出都应在样例中出现。例外：实际上不可能无解，但要求判断是否有解的题目。

样例解释

题目描述越复杂、越不易理解就越应当有详细的样例解释。

题目难度越简单就越应当有详细的样例解释。

详细的样例解释可以选择配上图片。

较大的样例可以没有样例解释。

为了照顾色觉障碍者，最好不要使颜色成为理解样例解释所必备的。可以用彩色图片来美化样例解释，但如果一定要用颜色传递一些必要的信息，最好不要同时出现红黄或者红绿。

时限、空间限制与部分分

时限与空间限制的目的是卡掉复杂度错误的做法。（当然，也是为了防止评测用时过长，如：只对交互次数有限制而对时间复杂度没有限制的交互题也有时间限制。）

因此，原则上时间限制应当选取不使错误做法通过的尽量大的值。

一般地，时限应满足以下要求：

1. 至少为 std 在最坏情况下用时的两倍。
2. 如果比赛允许使用 Java，应使 Java 能够通过。
3. 不应使错误做法通过（实在卡不掉、想放某种错解过除外）。

为了更好地在放大常数做法过的同时卡掉错解，一般可以采用同时增大数据范围和时限的方法。但要注意，有时正解（由于缓存等玄学问题）会在数据范围增大时有极大的常数增加，此时增大数据范围不一定能够增大正解与错解之间用时的差距。

在有部分分的赛制中，还可以通过设置有梯度的数据、数据范围稍小的数据来使较为优秀的错解和大常数正解不能通过，同时使其获得较高的部分分。

需要注意的是，在数据范围小于 $5 \cdot 10^5$ 时，应当考虑是否能使用 指令集[13] 通过。

一般情况下空间限制应当设置的足够大，除非空间复杂度更优的做法的确十分巧妙，值得卡掉空间复杂度大的做法。这种情况下可以考虑设置空间限制较松的部分分。值得注意的是，如果不想卡掉空间消耗较大的做法，数据结构题一般需要设置较大的空间限制。

> 一道好题应该具有它的选拔性质，具有足够的区分度。应该至少 4 档部分分，让新手可以拿到分，让高手能够展示自己的实力。
>
> ——vfk《UOJ 精神之源流》

部分分一般分为较小数据范围与特殊性质两种。

较小数据范围一般要设置多档，即使你想不到某种复杂度的做法，也可以考虑给这种复杂度一档分。一般来说，为了避免卡常，可以设置一档极限数据除以二的部分分。

「数据有梯度」最好用多档部分分替代。

特殊性质部分分的设置要依具体题目而定。理想的特殊性质部分分应当是能够引导选手思考正解的。与较小数据范围部分分不同，在你不会针对某种特殊性质的做法时，最好不要给这种特殊性质一档分。例如：「CTS2019」 随机立方体[14] 的 $k = 1$ 这档部分分在讲题时就被很多人吐槽，称这档部分分妨碍了思考正解。

如果题目给分方式与默认方式不同（如：在一般的 OI 赛制比赛中绑 subtask 测试），一定要在题面中说明。

不推荐使用「百分之 XX 的数据满足 XX」的说法，尤其是数据范围有多个变量时。例如，「30% 的数据满足 $n \le 1000$」和「40% 的数据满足 $m \le 100$」可能描述了 70% 的数据的性质，也可能只描述了 40% 数据的性质。一般来说，subtask 或数据范围表格是更好的选择。

造数据

数据生成是出题过程中必要的一步，也是对拍时所必需的，掌握一些生成数据的技巧，就能使造数据的过程更加轻松，造出来的数据强度更高。

生成随机数据

生成随机数

请参考 随机函数 页面。

需要特别提醒的是，在生成值域比随机函数返回值更大的数时，请**不要**使用 `rand() * rand()` 之类的写法，这样的写法生成的随机数非常不均匀。

另外，出题时推荐使用 testlib 来造数据，可以保证在不同平台上同一个种子生成的随机数相同，并且种子会依据命令行参数自动生成。

生成随机排列

可以使用 STL 中的 `std::shuffle` 函数，形如 `std::shuffle(a, a + n, rng)`，这里 `rng` 是一个随机数生成器，比如 `std::mt19937 rng(std::chrono::steady_clock::now().time_since_epoch().count())`。

请**不要**使用 `std::random_shuffle`，它在 C++14 中弃用，C++17 中被移除。

生成随机区间

常见错误方法：在 $[1,n]$ 中随机生成左端点 l，再在 $[l,n]$ 中随机生成右端点 r。这样的话生成的区间会比较靠右。

较为正确的方法（推荐做法）：在 $[1,n]$ 中随机生成两个数，取较小的作为左端点，较大的作为右端点。

真正均匀随机的方法：在 $[0,n]$ 中生成一个随机数 x，若 $x=0$，再在 $[1,n]$ 中生成一个随机数 y，区间为 $[y,y]$；否则按「较为正确的方法」生成。

生成随机树

常用方法是为 $2 \sim n$ 的每个节点 i 从 $[1,i-1]$ 中随机选择一个父亲。这样做的话生成的树不是均匀随机的，期望高度为 $O(\log n)$。

还有一种随机方法：从 $[i \cdot low, i \cdot high]$ 中随机选择 i 的父亲。若 low 和 $high$ 设置得当，可以造出强度较高的树。

真正均匀随机的方法是利用 Prüfer 序列，先生成一个随机 Prüfer 序列，再通过序列生成树。这样做的话，树的期望高度为 $O(\sqrt{n})$。

除此之外，可以随机一个排列来给节点重编号/打乱边的顺序。

构造数据

区间相关的题目

常用构造：长度特别小（特殊地，全部为单点）、长度特别大（特殊地，全部为整个序列）。

需要分解因数的题目

可重质因数个数尽量多：2 的幂。

去重后质因数个数尽量多：最小的若干个质数相乘。

约数尽量多：可以参考 OEIS 上的 A002182[15] 数列。

树上问题

常用构造：

- 链
- 菊花
- 完全二叉树
- 将完全二叉树的每个节点替换为一条长为 $\sqrt{n}$ 的链
- 菊花上挂一条链
- 链上挂一些单点
- 一棵高度为 d 且 $d>1$ 的树的根节点有两个儿子，左子树是一条长为 $d-1$ 的链，右子树是一棵高度为 $d-1$ 的这样的树。

如果不是在考场上，还可以使用 Tree-Generator[16] 来生成各种各样的树。

批量生成数据

笔者推荐使用命令行参数 + bat/sh 的方法。

例如：

gen.cpp:

```cpp
#include "testlib.h"

using namespace std;

int n, m, k;
vector<int> p;

int main(int argc, char* argv[]) {
  registerGen(argc, argv, 1);

  int i;

  n = atoi(argv[1]);
  m = atoi(argv[2]);
  k = rnd.next(1, n);

  for (i = 1; i <= n; ++i) p.push_back(i);

  shuffle(p.begin(), p.end());
  // 使用 rnd.next() 进行 shuffle

  printf("%d %d %d\n", n, m, k);
  for (i = 0; i < n; ++i) {
    printf("%d%c", p[i], " \n"[i == n - 1]);
    // 把字符串当作数组用，中间空格，末尾换行，是一个造数据时常用的技巧
  }

  return 0;
}
```

gen_scripts.bat:

```
gen 10 10 > 1.in
gen 1 1 > 2.in
gen 100 200 > 3.in
gen 2000 1000 > 4.in
gen 100000 100000 > 5.in
```

这样做的好处是，对于不同的数据只需要写一个 generator，并且可以方便地修改某个测试点的参数。

造数据的要求

数据应当包含各个参数的最小值和最大值。

数据应当包含各种边角情况。

在使用 subtask 时，数据（包括输入、输出）最好覆盖到值域中的各个范围，而不是只有数据范围的最大值。

为了防止针对特殊构造的特判过掉，可以将不同的构造结合在一个测试点中，或者数据的大部分是构造，掺杂小部分的随机。

数据中应当包含各种各样的构造，即使你不知道什么错解会挂在这组构造上。（在按测试点给分的赛制中需要酌

情处理。)

当然，如果你已知一个（正常人能想的到、写的出的）正确性有问题的错解，要尽量卡掉它。

需要特别提醒的是，如果有整型溢出的可能，一定要卡掉会溢出的做法。在有部分分的赛制中，不应使不开 long long 的人得到和暴力一样甚至更低的分数。

如果有 pretests，pretests 应尽量强，(同时尽量少)。换言之，你需要在 pretests 中（用尽量少的数据组数）包含该题的所有已知叉点。

如果你希望出现少量而非没有 FST，仍然应当保证 pretests 的强度，因为实际比赛中很可能出现你意想不到的错误，导致远远高出预期的 FST 数量。

数据的格式

这里引用 CodeChef 对题目输入数据的格式要求，可作为一般情况下的参考：

> 1. 不使用 Windows 格式的换行符，即 `\r\n`。
> 2. 最后一行的末尾有换行符，即整个文件的最后一个字符需要是 `\n`。
> 3. 没有空行。
> 4. 任何一行的开头和末尾都没有空白字符。
> 5. 连续的空格不超过 1 个。

Special Judge

SPJ 编写教程

输出方案题和输出浮点数题是两种较为常见的需要使用 SPJ 的题型，其它题目视情况也需要使用 SPJ。在 CF 上，所有题目都必须使用基于 testlib 的 checker，例如：题目要求输出若干个整数时，使用 testlib 自带的 ncmp checker，选手可以任意输出空白字符（既可以空格也可以换行）。

checker 一般使用 testlib 编写。由于 checker 要应对各种各样的不合法输出，需要极强的鲁棒性，不使用 testlib 是很难写好 checker 的。

编写 checker 需要注意以下两点：

1. 你需要应对各种不合法的输出，因此，请检查读入的每个变量是否在合法范围中（`readInt(minvalue, maxvalue)`）。例如：读入一个在 check 过程中会作为数组下标的变量时必须检查其范围，否则可能引发数组越界，有时这会导致 RE，有时则可能判为 AC。
2. 原则上 checker 中不应检查空白字符（即，不应使用 `readSpace()`、`readEoln()`、`readEof()`，值得一提的是，testlib 会自动检查是否有多余的输出）。

题解

题解的目标是让预计会来参加比赛的人都能看懂。所以官方题解详细程度的要求会比一般的题解高。

关于部分分

在有部分分的题目中，题解里可以考虑写一写部分分的做法。

关于知识点

解题中用到的知识点应当明确指出。对于一些难度和题目难度相当的知识点，最好给出学习该知识点的资料（比如一篇博客的地址）。

关于定义

题解中不要凭空冒出来一些概念。

例如：dp 的题解要解释清楚状态的定义。

关于细节

具体的实现细节如果比较巧妙最好写出来，否则的话「详见代码」也是可以的。如果「详见代码」的话，最好在代码中加上一定的注释。

标程

标程中最好去掉冗余部分。比如，有的题解中保留了完整的 define 模板（为了提高做题速度，包含大量 define 与常用函数，常用于 CF 等在线比赛），并且其中很大一部分都没有用到，这是不好的。

如果涉及到一些题解中没有详细说明的实现细节，最好加上适量的注释。

比赛

比赛通知中的题目难度需真实

> Remember that authors tend to underestimate the difficulty of their problems.
>
> ——Codeforces PROPOSE A PROBLEM 页面的提醒

出题人很可能错误估计题目的难度，因此，如果要在比赛通知中写上比赛难度，需要谨慎考虑，最好提前请人来验题并进行评估。

题目难度的分配

在类国内 OI 的模拟赛中，往往是三道题的整体难度与比赛难度相当即可。

在类 CF/ATC 这种线上赛的比赛中，需要尽量保证难度的递增（虽然由于对难度的误估很多时候都并不能真正做到），并且尽量避免出现大的 difficulty gap。可以通过把一题分为难易两题（两个 subtask）来减少 difficulty gap，但是分 subtask 需要谨慎考虑，也有很多人不喜欢 CF 赛制中的 subtask（Are subtasks evil?[17]），原因包括但不限于：

- 由于赛制原因，可能先做 easy version 再做 hard version 罚时更少而总分更高
- subtask 的赋分往往与题目难度不成正比
- 很多时候 easy version 的题目并不是一道合格的题目（不有趣）
- 很多时候 easy version 的解法对于思考 hard version 的正解没有帮助

题目知识点的分配

一场比赛应尽量涵盖较广的知识点（专题训练赛当然除外）。

经典反例：涵盖了动态规划、期望、组合计数、容斥原理、多项式等多种知识点的 CTS2019。

> 我要从五道题里选六道，我也很无奈啊。
>
> ——CTS2019 组题人给出的理由，没有收到足够多的题目投稿

出题平台

Polygon

Polygon 是一个功能非常强大的多人合作出题平台，可以作为在任何网站（使用 package 功能导出到不支持 Polygon 的网站）多人合作出题的首选方案，单人出题（尤其是在不同设备上出题）时也是很不错的选择，使用方法参见 Polygon 简介。

Codeforces

Codeforces 是全球最著名的算法竞赛网站之一，题目质量较高，非常适合有一定出题经验并且想进一步提升出题水平、想要出一套高质量题目的出题人。不足之处是审核速度较慢（一般要几个月），但你也可以在审核期间就开始题目的准备（虽然有题目被否掉导致准备白费了的风险）。

出题资格

- 蓝名且参加过至少 25 场 rated 比赛；
- 紫名且参加过至少 15 场 rated 比赛；
- 橙名且参加过至少 5 场 rated 比赛；
- 红名或黑红名。

提交比赛申请

有了出题资格后，在侧边栏可以看到 Propose a contest/problems[18] 按钮。

点进去之后，先写一份 contest proposal（在 PROPOSE A CONTEST 里写），然后再写 problem proposal 并添加进比赛里。

题目决定好之后，就可以将 contest proposal open to review（提交审核）了。

在 Polygon 上准备题目

参考 Polygon 简介。

与管理之间的联系

与管理联系有两个作用：

1. 加快审核速度。
2. 进入准备阶段后管理会提供建议和帮助。

正规的联系方式是在 proposal system 中以 proposal 的形式提交申请，管理开始审核之后以 comment 的形式在 proposal 的下方进行讨论。

实际上，如果 proposal 长时间没有过审，可以考虑私信联系管理（其实 CF 上写了"Don't send private messages or emails to coordinators"，但 300iq 在 评论[19] 中表示可以私信他）。

Comet OJ

Comet OJ 链接[20]

已经不再活跃（截至 2021 年 11 月，最后一场比赛是 2020 年 1 月的）。

出题申请：https://info.cometoj.com/contests/Questionnaire_IssuerInfo/[21]

CodeChef

印度的算法竞赛平台，有三种赛制：10 天且带 challenge 的 Long Challenge，2.5h 类 ICPC 的 Cook-Off，3h 类 IOI 的 LunchTime。

出题 FAQ：https://www.codechef.com/wiki/faq-problem-setters[22]

出题指南：https://www.codechef.com/problemsetting[23]

AtCoder

日本的算法竞赛平台，出题联系方式：contest@atcoder.jp[24]。

UOJ & LOJ

比赛不多的国内 OJ。

洛谷

个人公开赛

在「我的题库」中出题并提交比赛申请。

团队公开赛

在团队页面中出题并提交比赛申请。

参考资料

1. vfk《UOJ 精神之源流 》
2. 王天懿 《 论偏题的危害 》
3. CF 出题人须知（国内可访问的图片版[25]）
4. CF 出题人的自我修养

本文由作者本人自 ouuan 的出题规范[26] 搬运而来并有所修改、补充。

参考资料与注释

[1] 「XR-1」柯南家族

[2] 「模板」树上后缀排序

[3] NOIP2017 小凯的疑惑

[4] 「清华集训 2015」多边形下海

[5] 「SDOI2010」猪国杀

[6] 「集训队互测 2015」未来程序 · 改

[7] LaTeX 数学公式大全

[8] LaTeX 各种命令，符号

[9] CF1172D Nauuo and Portals [9-1] [9-2]

[10] CF1173A Nauuo and Votes [10-1] [10-2]

[11] 「GuOJ Round #1」琪露诺的冰雪宴会

[12] CF1423(4)N Bubblesquare Tokens

[13] 指令集

[14] 「CTS2019」随机立方体

[15] A002182

[16] Tree-Generator

[17] Are subtasks evil?

[18] Propose a contest/problems

[19] 评论

[20] Comet OJ 链接

[21] https://info.cometoj.com/contests/Questionnaire_IssuerInfo/

[22] https://www.codechef.com/wiki/faq-problem-setters

[23] https://www.codechef.com/problemsetting

[24] contest@atcoder.jp

[25] 国内可访问的图片版

[26] ouuan 的出题规范

第 3 章

工具软件

3.1 工具软件简介

本章主要介绍与竞赛有关的工具软件，包括一些代码编辑器的介绍和 OJ 相关工具。

程序是解决 OI 问题的工具，熟练运用代码编辑器是学习 OI 的前提。

了解 OJ 相关的工具能让 OI 之旅更加舒适、便捷。

3.2 代码编辑工具

3.2.1 Vim

Authors: Enter-tainer, ouuan, Xeonacid, Ir1d, partychicken, ChungZH, LuoshuiTianyi, Kewth, s0cks5, Doveqise, StudyingFather, SukkaW, SodaCris, SkyeYoung, 383494, danielqfmai

Vim - 无处不在的文本编辑器。

简介

Vim 是从 vi 发展出来的一个文本编辑器。其代码补完、编译及错误跳转等方便编程的功能特别丰富，在程序员群体中被广泛使用。

安装

Linux 系统通常自带 Vim，打开终端输入 `vim` 即可启用。

若需手动安装，Vim 的 官方网站[1] 提供了下载的 说明文档[2]，按照需求编译安装即可。

Vim 的模式与常用键位

Vim 的基础操作在 Vim 自带的教程里将会讲述。打开终端输入 `vimtutor` 即可进入教程。

这些操作通常需要二三十分钟来大致熟悉。

命令模式 (Command Mode)

进入 Vim 后的默认模式。

此状态下敲击键盘动作会被 Vim 识别为命令，而非输入字符，比如我们此时按下 i，并不会输入一个字符，i 被当作了一个命令。

Vim 的方向键是↑、↓、←、→，或者 h、j、k、l。

```
        ↑ (k)
        ^
(h) ← <     > → (l)
        v
         (j)
```

以下是命令模式常用的命令：

- `i` 切换到输入模式，在光标当前位置开始输入文本。按 Esc 键可回到普通模式。
- `x` 用于删除光标后的一个字符。
- `:` 切换到底线命令模式，以在最底一行输入命令。
- `a` 切换到输入模式，在光标后开始输入文本。
- `o` 切换到输入模式，在光标下插入新的一行；`O` 切换到输入模式，在光标上插入新的一行。
- `p` 粘贴剪贴板内容到光标下方；`P` 粘贴剪贴板内容到光标上方。
- `dd` 删除光标所在的一整行。
- `d` 命令也是删除，通常配合其他键使用。
- `u` 撤销上一次对文本的更改。
- `y` 命令可以复制被选中的区域。需要按 `v` 进入可视模式操作。
- `yy` 复制当前行。
- `Ctrl + r` 重做上次撤销的操作。
- `:w` 保存文件，常配合 q 保存退出。
- `:q` 退出 Vim。
- `:q!` 强制退出 Vim，不保存修改。

部分其他命令：

- `c` 命令用于修改，相当于 `di`。
- `=` 命令可以以默认格式对选中行应用自动缩进。
- `==` 自动缩进当前行。
- `.` 命令可以重复上次执行的命令。
- `gg` 命令可跳至代码的开头；`G` 命令可跳至代码最后一行的开头；`G` 命令前加数字可跳至指定行。
- `w` 可以跳到下个单词的开头；`e` 可以跳到当前单词或下一单词的结尾；`b` 可以跳到当前单词或上一单词的的开头；`0` 可以跳至行首；`$` 可以跳至行尾。`w`、`e`、`-0`、`$` 还可以与其他命令组合，比如 `de`、`dw`、`d0` 和 `d$` 分别对应删至单词尾、删至下个单词头、删至行首和删至行尾。

命令模式下按/，下方即会出现查找框，输入需要查找的字符，按回车后就能查看搜索结果。如果有多个查找结果，按 n 即可跳至下一个查找结果；按 N 可跳至上一个。

命令模式下按 * 可以查找当前光标下的单词。

在输入某个命令前，输入一个数字 n 的话，命令就会重复 n 次。

输入模式 (Insert Mode)

在命令模式下按下 i 就进入了输入模式，按 Esc 键可以返回到命令模式。

在输入模式中，可以使用以下按键：

- 字符按键以及 Shift 组合，输入字符

- ENTER，回车键，换行
- BACK SPACE，退格键，删除光标前一个字符
- DEL，删除键，删除光标后一个字符
- 方向键，在文本中移动光标
- HOME/END，移动光标到行首/行尾
- Page Up/Page Down，上/下翻页
- Insert，切换光标为输入/替换模式，光标将变成竖线/下划线
- ESC，退出输入模式，切换到命令模式

在输入模式下按 Ctrl+o 即可进入「输入 - 命令模式」，执行完一次操作后又会自动回到输入模式。

底线命令行模式

命令模式下按:，进入底线命令模式。

底线命令模式可以输入单个或多个字符的命令，可用的命令非常多。

在底线命令模式中，基本的命令有：

- `:help`/`:h` 查看英文版 Vim 在线帮助文档。
- `:w` 保存文件。
- `:q` 退出 Vim。
- `:wq` 保存文件，退出 Vim。
- `:q!`/`:!q` 强制退出 Vim，不保存修改。
- `:e filename` 可以打开当前目录下的指定文件。
- `:s` 命令是替换。

```
" 把当前行第一个匹配的 str1 替换成 str2
:s/str1/str2/
" 把当前行所有的 str1 替换成 str2
:s/str1/str2/g
" 把当前行所有的 str1 替换成 str2，在替换前询问
:s/str1/str2/gc
" 把第 x1 行至 x2 行中，每一行第一个匹配的 str1 替换成 str2
:x1,x2 s/str1/str2/
" 把第 x1 行至 x2 行中所有的 str1 替换成 str2
:x1,x2 s/str1/str2/g
" 第 x1 行至 x2 行中所有的 str1 替换成 str2，在替换前询问
:x1,x2 s/str1/str2/gc
" 把所有行第一个匹配的 str1 替换成 str2
:%s/str1/str2/
" 把全文件所有的 str1 替换成 str2
:%s/str1/str2/g
" 把全文件所有的 str1 替换成 str2，在替换前询问
:%s/str1/str2/gc
```

如果命令形式是 `:! command`，则命令将在 bash 终端执行。

按 Esc 键可以退出底线命令模式。

可视模式 (Visual mode)

按 `v` 进入可视模式，多用于选中区域。按 `V`（`Shift+v`）进入行可视模式，用于选中行。

按 Ctrl+v 或 Ctrl+q 进入块可视模式 (visual block)。

进入块可视模式后，按 I 或 A 进入插入模式（相当于 `i` 和 `a`），退出插入模式后对本行所做的改动将被应用到选中的每一行同一位置。常用于批量添加注释。

选中后输入 `y` 或 `d` 亦可执行相应命令。

三种可视模式可以通过按键相互转化。

Vim 的快捷键

可参考 史上最全 Vim 快捷键键位图 — 入门到进阶[3]

进阶知识

. 命令

Vim 的使用者不可避免地会抗拒重复的文本修改，因为 Vim 注定比其他编辑器会多出两次按键——Esc 与 i。但是，Vim 其实提供了重复命令 `.`，它适用于重复的添加、修改、删除文本操作。

`.` 命令可以重复上次执行的命令。但是这个「命令」并不只限于单一的命令，它也可以是数字 + 命令的组合；进入插入模式 + 输入文本 + Esc 也是命令的一种。所以，适当使用 `.` 命令才能达到最高的效率。

例如，如下代码的每一行末尾都少了分号：

```
int a, b
cin >> a >> b
cout << a + b
return 0
```

将 `.` 与搭配移动到行尾插入命令 `A` 使用，就能高效地补上末尾的分号。

```
A;<Esc>
" 重复下面的命令
j.
```

再例如，如下代码中，后面五个赋值语句的数组名全部写错了：

```
int check() {
  book[1] = 1, book[2] = 1, book[3] = 1, bok[1] = 1, bok[2] = 1, bok[3] = 1,
  bok[4] = 1, bok[5] = 1;
  return 0;
}
```

一个个改过于麻烦，而命令行模式的 `s` 命令又会全部改掉。

第一种改法是搭配普通模式下的 `s` 命令（删除光标处字符并进入插入模式）使用。来到第一个错误的数组名首字母处，按下 `3s/cw`，输入正确的数组名并退出。之后把光标一个个移过去，再使用 `.` 命令。

第二种比较节省时间的改法是利用查找模式修改。键入 `/bok`，接着按下回车，并使用 `n` 键来到第一个错误的数组名首字母处，键入 `3s 新数组名 <Esc>`，最后重复 `n.`。

第三种改法是简易查找命令 `f`。在一行中普通模式下，`f + 单个字符`即可查找此行中出现的这个字符并将光标移至字符处；按 `;` 查找下一个，`,` 查找上一个。所以对于上面的代码，只需键入 `fb;;;` 之后进入插入模式修改，然后 `;.` 即可。这种改法适用于只需行内移动的情况。

宏

Vim 的宏功能可以重复任意长的命令。

使用宏之前要先「录制」，即把一串按键操作录下来再回放，这样就达到了重复的效果。录制的方法很简单，普通模式下键入 `q` 开始录制。下一步，为录制的宏指定一个执行的命令键，可以按下 26 个字母中的任意一个来指定。这时左下方会显示`记录中 @ 刚刚选择的字母`。然后就可以开始录制命令了。同理，普通模式下按 `q` 暂停录制。

使用方法为按下 `:` 进入命令行模式，键入 `@` 选择的记录字母，然后之前录制的命令就被调用了。

将 `.` 和宏组合，即录制宏→调用宏→ `.` 重复命令→数字 +.，可以达到非常高的效率。

normal 命令

该命令与普通模式有关，效果是在指定行重复命令。

按 `:` 进入命令行模式，输入如下命令：

```
:a,b normal command
```

或者：

```
:a,b norm command
```

以上命令的意思是在普通模式下，对 a~b 行执行 `command` 命令。

由于 `normal` 命令可以被 `.` 命令重复调用，且其易于理解，它的使用频率甚至更高于宏。

数字 +.+ 宏 + normal

以上三种命令可以组合使用。例如：

> 我下载了一本书，我需要它的每一个章节都变成「标题」，以方便转换成 mobi 之类的格式，或者方便生成 TOC 目录跳转，怎么办呢？

以下是用 Vim 处理的过程：

1. 按下/调出查找框，输入正则表达式进行查找；
2. 用 `q` 命令开始录制宏；
3. 键入 `I#` 命令，然后按下 ESC；
4. 用 `q` 进行修改操作并结束宏录制；
5. 键入 `normal n@` 字母转到下一处并重复上一步操作；
6. 键入数字 `+ .` 多次重复。

外部链接

- Vim 官网[1]
- 原作者提供的配置[4]
- Vim 调试：termdebug 入门[5]
- Vim scripting cheatsheet[6]
- Learn Vimscript the Hard Way[7]
- Linux vi/vim | 菜鸟教程[8]

参考资料与注释

[1] 官方网站 [1-1] [1-2]

[2] 说明文档

[3] 史上最全 Vim 快捷键键位图—入门到进阶

[4] 原作者提供的配置

[5] Vim 调试：termdebug 入门

[6] Vim scripting cheatsheet

[7] Learn Vimscript the Hard Way

[8] Linux vi/vim | 菜鸟教程

3.2.2 Emacs

Authors: ouuan, akakw1, Ir1d, partychicken, Xeonacid

本页面为 Emacs 的入门教程。

> 15 分钟入门 Emacs。

简介

Emacs 是一款非常容易上手的编辑器，只需要简短的几行配置就能使用，但是想要非常熟练地使用 Emacs 进行各项工作还是需要一定的时间。

作为入门教程，这里仅介绍 Emacs 的基本功能，以及较方便地用 Emacs 编写、调试代码的方法。

入门

命令

命令在 Emacs 中有很大的作用。

使用 Application 键[1]（Windows 系统下 Emacs 未指定这个键，需要手动设置）或者快捷键 M-x（Alt+x）可以打开命令输入，输入完按下回车可以执行命令。

通常使用 `es` 或者 `eshell` 命令来打开 Eshell（类似一个终端）。

输入命令通常可以用快捷键代替。

缓冲 (buffer)

缓冲即打开的文件和进程，在不保存的情况下，在缓冲中修改并不会修改到文件。

在缓冲区的底部点击缓冲的名字或者使用快捷键可以切换缓冲。

编译、调试和运行

编译和调试功能的入口在顶部菜单栏的 Tools 下拉栏。使用者也可以通过命令或者自定义快捷键使用编译和调试功能。

可以使用终端或 Eshell 运行程序。

按下 Tools 中的调试 (gud-gdb) 后，输入程序名（一般会自动输好，但如果中途将程序另存为或者打开了两个需要调试的程序，**自动输好的文件名可能会有误**）即可开始调试。

分屏

这个功能能让使用者同时查看各个缓冲的内容，而不需要来回切换缓冲，方便测试、调试代码。

分屏功能可以同时显示多个窗口，用鼠标拖动窗口的边缘可以缩放窗口。

几个快捷键：

- 删除分屏”C-x 0”：将这个分屏删去
- 横向分屏”C-x 3”：将这个分屏横向分成两半
- 纵向分屏”C-x 2”：将这个分屏纵向分成两半

推荐的窗口布局为将窗口分为四块：先横向分，调整一块的宽度约为 3/4 屏，作为编辑窗口。将另一块横向分，一块作为调试和编译信息显示的窗口，另一块再纵向分，一块打开输入文件，一块打开输出文件。

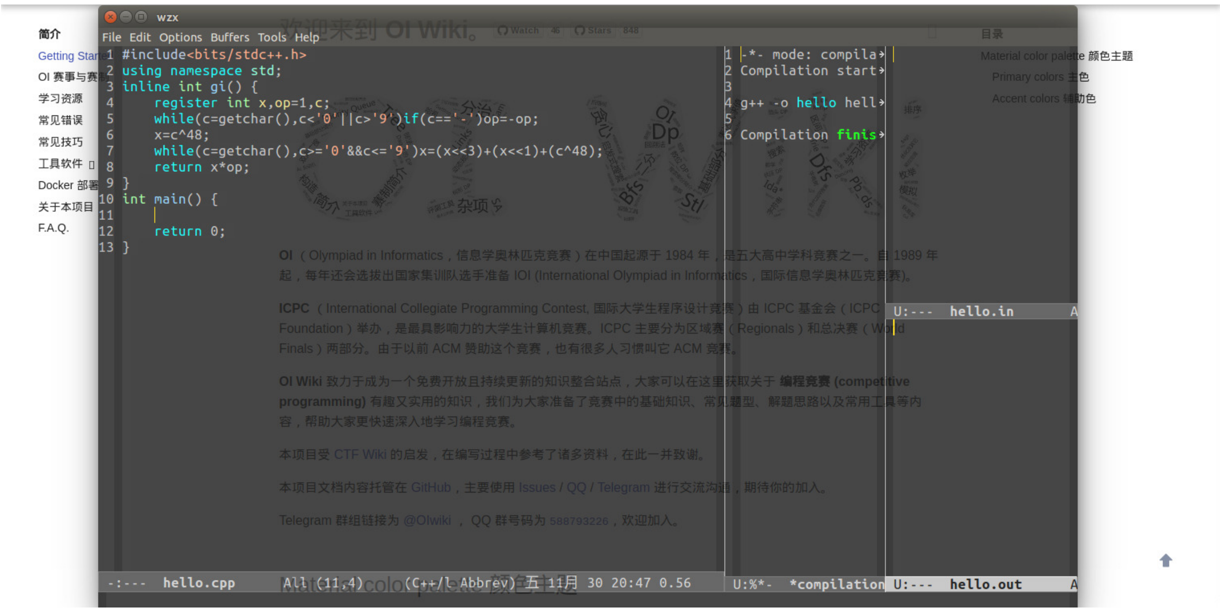


图 3.1

快捷键

Emacs 拥有极为丰富的快捷键，可以大幅提高工作的效率。使用者可以在配置中自定义快捷键或者设置快捷键的映射。

由于快捷键过多，所以 Emacs 快捷键的使用与操作系统不同。

为了方便描述，做如下约定：

字符	键位
C	Ctrl
M	Alt
?	任意键位

一般有以下三种：

- `F?`、`ESC`：直接按下对应的功能键。
- `M-?`、`C-?`、`C-M-?`：按下 Alt 或者 Ctrl 的同时按下 `?`。
- `? ?`：先按下第一个 `?` 代表的键，松开再按下第二个 `?` 代表的键。

下面是一些常用的快捷键：

- `C-x h`：全选

- `C-x left`、`C-x right`：切换到上/下一个缓冲
- `C-x d`：打开一个目录
- `C-x C-f`：打开一个文件（如果不存在文件则新建文件）

个性化

刚安装好的 Emacs 外观难看且不好使用，因此需要对其进行个性化设置。

由于配置不好记，所以部分可以直接设置的部分建议不要记配置。

直接设置

- Options：`Highlight Matching Parentheses` 高亮匹配括号
- Options：`Blink Cursor` 设置光标闪烁
- Options Show/Hide：`Tool Bar` 显示/不显示工具栏（默认显示，建议不显示）
- Options：`Use CUA Keys` 勾选后可以使用 Ctrl + C,Ctrl + V 等快捷键进行复制粘贴
- Options Customize-Emacs：`Custom Theme` 选择配色方案，选择完后需要点击保存
- Options：`Save Options` **保存配置**

配置

在 home 目录下显示隐藏文件（Windows 系统在**用户目录**的 `AppData\Roaming` 目录下），".emacs" 就是配置文件（如果没有说明之前没保存），打开修改即可。如果 Emacs 已打开，则需要重启 Emacs，配置才能生效。

考场推荐的配置如下。

```
;; 设置一键编译可以自行添加参数难背考场不建议使用不建议依赖一键编译
(defun compile-file ()(interactive)(compile (format "g++ -o %s %s -g -lm -Wall"
(file-name-sans-extension (buffer-name))(buffer-name))))
(global-set-key [f9] 'compile-file)
;;;; 设置编译快捷键（如果设置了一键编译不要与一键编译冲突）
;;(global-set-key [f9] 'compile)

(global-set-key (kbd "C-a") 'mark-whole-buffer) ;; 全选快捷键
(global-set-key (kbd "C-z") 'undo) ;; 撤销快捷键
(global-set-key [f10] 'gud-gdb) ;;GDB 调试快捷键
(global-set-key (kbd "RET") 'newline-and-indent) ;; 换行自动缩进
(global-set-key (kbd "C-s") 'save-buffer) ;; 设置保存快捷键
(setq-default kill-ring-max 65535) ;; 扩大可撤销记录

;;C++ 代码风格一般控制缩进规则
;;;"bsd" 所有大括号换行
;;;"java" 所有大括号不换行。else 接在右大括号后面
;;;"awk" 只有命名空间旁、定义类、定义函数时的大括号换行。else 接在右大括号后面
;;;"linux" 只有命名空间旁、定义类、定义函数时的大括号换行。else 接在右大括号后面。一
般来说，这个风格应该有 8 格的空格缩进
(setq-default c-default-style "awk")
```

" 完整配置"

```
;; 设置一键编译
(defun compile-file ()(interactive)(compile (format "g++ -o %s %s -g -lm -Wall"
(file-name-sans-extension (buffer-name))(buffer-name))))
(global-set-key [f9] 'compile-file)
```

```
;;;; 设置编译快捷键 (如果设置了一键编译不要与一键编译冲突)
;;(global-set-key [f9] 'compile)

;; 考场必备
(global-set-key (kbd "C-a") 'mark-whole-buffer) ;; 全选快捷键
(global-set-key (kbd "C-z") 'undo) ;; 撤销快捷键
(global-set-key [f10] 'gud-gdb) ;;GDB 调试快捷键
(global-set-key (kbd "RET") 'newline-and-indent) ;; 换行自动缩进
(global-set-key (kbd "C-s") 'save-buffer) ;; 设置保存快捷键
(setq-default kill-ring-max 65535) ;; 扩大可撤销记录
;;(define-key key-translation-map [apps] (kbd "M-x")) ;; windows 系统下设置命令快
捷键

;; 设置缩进
;;;C++ 代码缩进长度。
(setq-default c-basic-offset 4)
;;; 使用 tab 缩进
(setq-default indent-tabs-mode t)
;;;tab 的长度。务必和缩进长度一致
(setq-default default-tab-width 4)
(setq-default tab-width 4)

;; 设置默认编码环境
(set-language-environment "UTF-8")
(set-default-coding-systems 'utf-8)

;; 不显示欢迎页面
(setq-default inhibit-startup-screen t)

;; 设置标题
(setq-default frame-title-format "")

;; 显示行号
(global-linum-mode t)

;; 高亮
(global-hl-line-mode 1);; 高亮当前行
(show-paren-mode t);; 高亮匹配括号
(global-font-lock-mode t);; 语法高亮

;; 允许 emacs 和外部其他程序的粘贴好像默认允许
(setq-default x-select-enable-clipboard t)

;; 设置字体是 Ubuntu Mono 的 16 号，如果字体不存在会报错
(set-default-font "Ubuntu Mono-16")
;(set-default-font "Consolas-16") ;; windows 系统请用这条

;; 鼠标滚轮支持
(mouse-wheel-mode t)

;; 设置光标形状为竖线 (默认为方块)
(setq-default cursor-type 'bar)

;; 回答 yes/no 改成回答 y/n
```

```
(fset 'yes-or-no-p 'y-or-n-p)

;; 透明度
(set-frame-parameter (selected-frame) 'alpha (list 85 60))
(add-to-list 'default-frame-alist (cons 'alpha (list 85 60)))

;; 减少页面滚动的行数，防止整页地滚动
(setq-default scroll-margin 3 scroll-conservatively 10000)

;; 优化文件树结构
(ido-mode t)

;; 配色方案
(setq default-frame-alist
         '((vertical-scroll-bars)
           (top . 25)
           (left . 45)
           (width . 120)
           (height . 40)
           (background-color . "grey15")
           (foreground-color . "grey")
           (cursor-color . "gold1")
           (mouse-color . "gold1")
           (tool-bar-lines . 0)
           (menu-bar-lines . 1)
           (scroll-bar-lines . 0)
           (right-fringe)
           (left-fringe)))

(set-face-background 'highlight "gray5")
(set-face-foreground 'region "cyan")
(set-face-background 'region "blue")
(set-face-foreground 'secondary-selection "skyblue")
(set-face-background 'secondary-selection "darkblue")
(set-cursor-color "wheat")
(set-mouse-color "wheat")

(custom-set-variables
 '(ansi-color-faces-vector
   [default default default italic underline success warning error])
;; 启动 Ctrl-x Ctrl-c Ctrl-v = 剪切复制粘贴
 '(cua-mode t nil (cua-base))
 '(show-paren-mode t)
;; 隐藏工具栏
 '(tool-bar-mode nil))
;; 关闭光标闪烁
 '(blink-cursor-mode nil)
(custom-set-faces)
```

拓展阅读

要以终端模式启动 Emacs，在启动时添加参数 `-nw`。Emacs 有多种变体，如采用 native-comp 来减少延迟的 GCC Emacs[2] 及其纯 GTK 版本变体、针对 macOS 优化的 Emacs Macport。

Emacs 有中心化的软件仓库，配置后可通过 `M-x package-install` 来安装插件。使用 镜像站[3] 可以加快下载速度。

Emacs 可以使用语言服务器（Language Server Protocol）来提高编辑体验，目前推荐的 C++ 后端是 Clangd[4]。前端可以采用 Eglot[5] 或 Emacs LSP[6]，参阅 此条目[7] 可能对选择前端有所帮助。

拓展名为 `.org` 的 Org Mode 文档可以通过 Pandoc[8] 转换为 Markdown 文档。

参考资料与注释

[1] 该键的作用是调出鼠标右键菜单，一般为右 Ctrl 左边的第一个键。

[2] GCC Emacs

[3] 镜像站

[4] Clangd

[5] Eglot

[6] Emacs LSP

[7] 此条目

[8] Pandoc

3.2.3 VS Code

Authors: NachtgeistW, Ir1d, ouuan, Enter-tainer, Xeonacid, ChungZH, keepthethink, abc1763613206, partychicken, Chrogeek, xkww3n, HeliumOI, Pinghigh, xiaofu-15191

简介

Visual Studio Code（以下简称 VS Code）是一个由微软开发，同时支持 Windows、Linux 和 macOS 等操作系统且开放源代码的代码编辑器。它是用 TypeScript 编写的，并且采用 Electron 架构。它带有对 JavaScript、TypeScript 和 Node.js 的内置支持，并为其他语言（如 C、C++、Java、Python、PHP、Go）提供了丰富的扩展生态系统。

官网：Visual Studio Code - Code Editing. Redefined[3]

使用 Code Runner 扩展运行代码

VS Code 安装并配置扩展后可实现对 C/C++ 的支持，但配置过程比较复杂。一个简单的编译与运行 C++ 程序的方案是安装 Code Runner 扩展。

Code Runner 是一个可以一键运行代码的扩展，在工程上一般用来验证代码片段，支持 Node.js、Python、C、C++、Java、PHP、Perl、Ruby、Go 等 40 多种语言。

安装的方式是在扩展商店搜索 Code Runner 并点击 Install；或者前往 Marketplace[4] 并点击 Install，浏览器会自动打开 VS Code 并进行安装。

安装完成后，打开需要运行的文件，点击右上角的小三角图标即可运行代码；按下快捷键 Ctrl+Alt+N（在 macOS 下是 Control+Option+N）也可以得到同样的效果。

> warning
>
> 如果安装了 VS Code 与 Code Runner 后，代码仍然无法运行，很有可能是因为系统尚未安装 C/C++ 的运

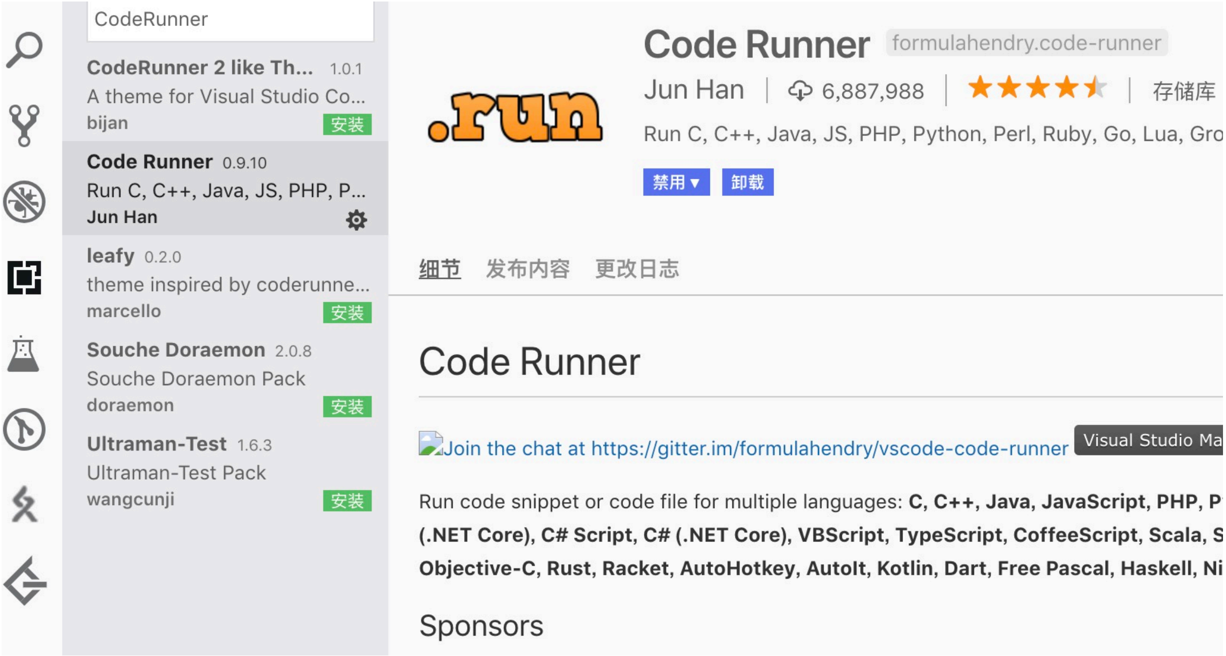


图 3.2

行环境，参考 Hello, World! 页面 以安装。

记得勾选设置中的 Run In Terminal 选项，如图：

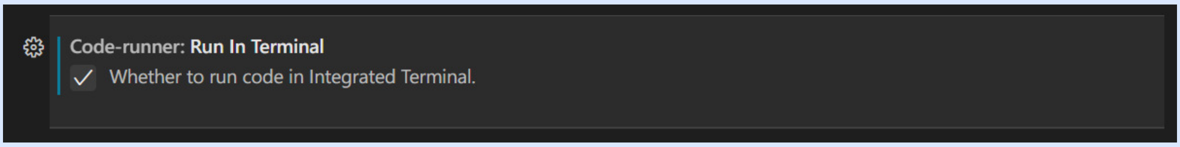


图 3.3

使用 C/C++ 扩展编译并调试/智能补全代码

安装扩展

在 VS Code 中打开扩展商店，在搜索栏中输入 `C++` 或者 `@category:"programming languages"`，然后找到 C/C++，点击 Install 安装扩展。

warning

在配置前，请确保系统已经安装了 G++ 或 Clang，并已添加到了 `PATH` 中。请使用 CMD 或者 PowerShell，而不是 Git Bash 作为集成终端。

配置 GDB/LLDB 调试器

GDB

在 VS Code 中新建一份 C++ 代码文件，按照 C++ 语法写入一些内容（如 `int main(){}`），保存并按下 F5，进入调试模式。如果出现了「选择调试器」的提示，选择 `C++ (GDB/LLDB)`。在「选择配置」中，G++ 用户选择 `g++.exe -` 生成和调试活动文件；Clang 用户选择 `clang++ -` 生成和调试活动文件。

warning

配置名称并非固定，而是可以自定义的。不同的操作系统可能具有不同的配置名称。

完成后，VS Code 将自动完成初始化操作在下方的集成终端中启动调试。至此，GDB 所有的配置流程已经完毕。

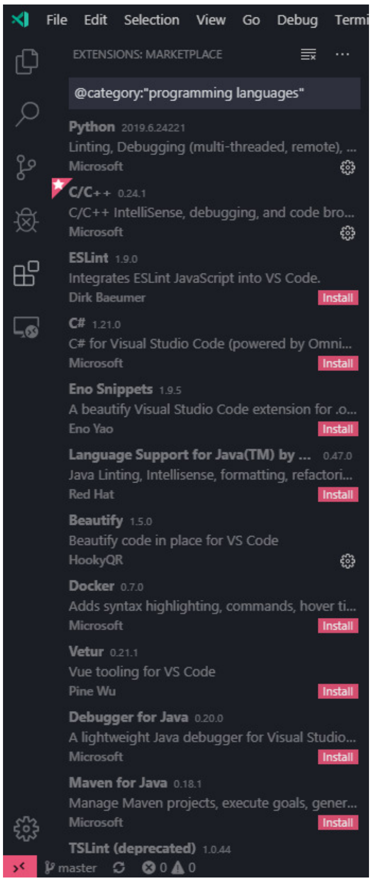


图 3.4

LLDB

如果需要采用 LLDB，需要安装另外一款扩展[1]——CodeLLDB[5]。从该项目的 Release 页面下载.vsix 文件后[2]，从 VS Code 的扩展页面安装。

先按照上文 GDB 的配置过程操作一遍，然后删除 `.vscode/launch.json`，按下 F5，选择 `LLDB`，再把 `launch.json` 中的 `${workspaceFolder}/<executable file>` 更改为 `${fileDirname}/${fileBasenameNoExtension}.exe` 即可。

至此，LLDB 配置完成。再次按下 F5 即可看到软件下方的调试信息。

若要在以后使用 VS Code 编译并调试代码，所有的源代码都需要保存至这个文件夹内。若要编译并调试其他文件夹中存放的代码，需要重新执行上述步骤（或将旧文件夹内的 `.vscode` 子文件夹复制到新文件夹内）。

开始调试代码

使用 VS Code 打开一份代码，将鼠标悬停在行数左侧的空白区域，并单击出现的红点即可为该行代码设置断点。再次单击可取消设置断点。

按下 F5 进入调试模式，编辑器上方会出现一个调试工具栏，四个蓝色按钮从左至右分别代表 GDB 中的 `continue`,`next`,`step` 和 `until`：

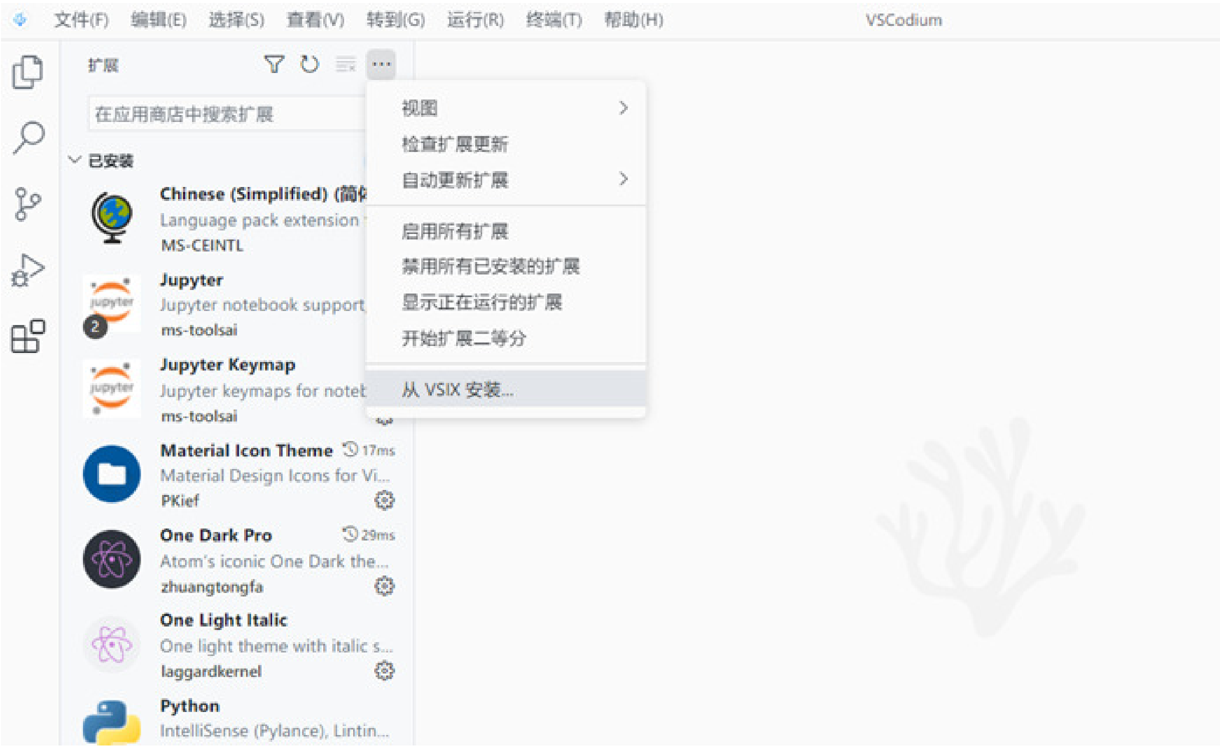
如果编辑器未自动跳转，点击左侧工具栏中的「调试」图标进入调试窗口，即可在左侧看到变量的值。

在调试模式中，编辑器将以黄色底色显示下一步将要执行的代码。

配置 IntelliSense

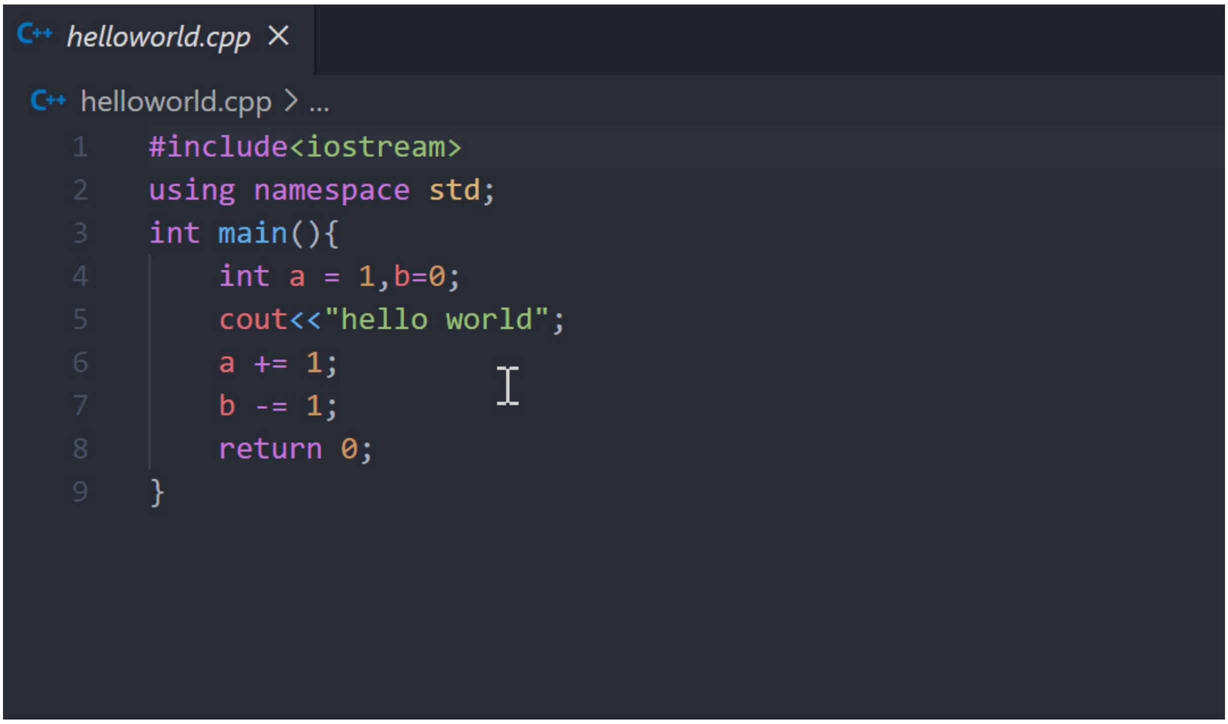
用于调整 VS Code 的智能补全。

如果你使用 Clang 编译器，在「IntelliSense 模式」中选择 `clang-x64` 而非默认的 `msvc-x64`；如果你使用 G++ 编译器，选择 `gcc-x64` 以使用自动补全等功能。否则会得到「IntelliSense 模式 msvc-x64 与编译器路径不兼容。」的错误。



文件(F)
编辑(E)
选择(S)
查看(V)
转到(G)
运行(R)
终端(T)
帮助(H)
VSCodium
扩展
在应用商店中搜索扩展
已安装
Chinese (Simplified) (简体
Language pack extension
MS-CEINTL
Jupyter
Jupyter notebook support
ms-toolsai
Jupyter Keymap
Jupyter keymaps for noteb
ms-toolsai
Material Icon Theme
17ms
Material Design Icons for Vi...
PKief
One Dark Pro
29ms
Atom's iconic One Dark the...
zhuangtongfa
One Light Italic
One light theme with italic s...
laggardkernel
Python
IntelliSense (Pylance), Lintin...
视图
检查扩展更新
自动更新扩展
启用所有扩展
禁用所有已安装的扩展
显示正在运行的扩展
开始扩展二等分
从 VSIX 安装...

图 3.5



helloworld.cpp
helloworld.cpp > ...
#include<iostream>
using namespace std;
int main(){
int a = 1,b=0;
cout<<"hello world";
a += 1;
b -= 1;
return 0;
}

图 3.6

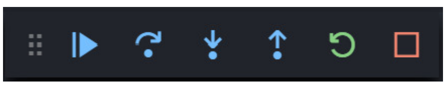


图 3.7

图 3.8

配置 clangd

warning

由于功能冲突，安装 clangd 扩展后 C/C++ 扩展的 IntelliSense 功能将被禁用（调试等功能仍然使用 C/C++ 扩展）。如果 clangd 扩展的功能出现问题，可以查看是否禁用了 C/C++ 扩展的 IntelliSense 功能。

clangd 简介

LLVM 官网上对 clangd 的介绍是这样的：

> Clangd is an implementation of the Language Server Protocol leveraging Clang. Clangd’s goal is to provide language ”smartness” features like code completion, find references, etc. for clients such as C/C++ Editors.

简单来说，clangd 是 Clang 对语言服务器协定（Language Server Protocol）的实现，提供了一些智能的特性，例如全项目索引、代码跳转、变量重命名、更快的代码补全、提示信息、格式化代码等，并且能利用 LSP 与 Vim、Emacs、VSCode 等编辑器协作。虽然官方给出的定义是 LSP 的实现，但 clangd 的功能更接近语言服务器（Language Server）而不仅仅只是协议本身。

VS Code 的 C/C++ 扩展也有自动补全等功能，但在提示信息的易读程度的准确度等方面与 clangd 相比稍逊一筹，所以我们有时会使用 clangd 代替 C/C++ 扩展来实现代码自动补全等功能。

安装

参见 Getting started[6]。

VS Code 扩展

打开 VS Code 扩展商店，在搜索栏中输入 `clangd` 找到 clangd 扩展并安装

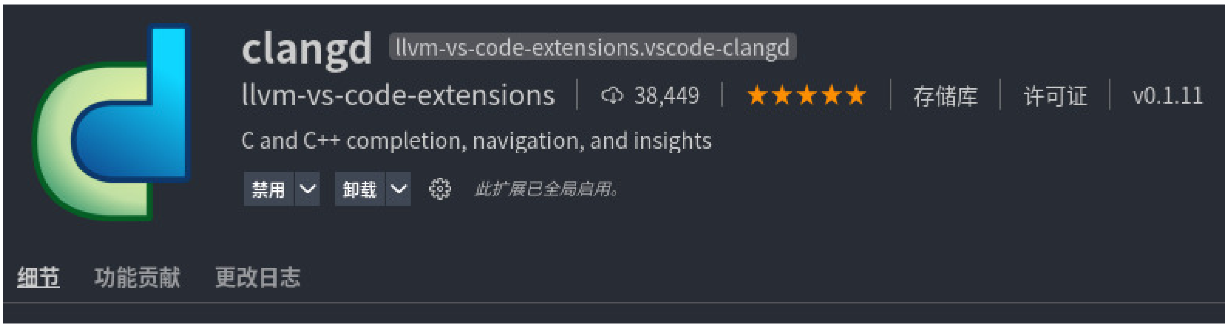


图 3.9

如果下方弹出 clangd 要求关闭 Intellisense 的对话框，点击"Disable Intellisense"，重新加载工作区，就可以享受 clangd 的自动补全等功能了。

编辑

语法设置

在新打开的编辑器中点击「选择语言」，即可打开对应的语法高亮，如图：

图 3.10

快捷键

部分快捷键：

按键	操作
Ctrl+C/X	复制/剪切当前行（当没有选择内容时）
Ctrl+Shift+K	删除当前行
Alt+Up/Down	行上移/下移
Alt+Shift+Up/Down	行向上/向下复制
Ctrl+/	切换行注释
Ctrl+[/]	行向左/右缩进
Ctrl+Shift+[/]	行折叠/展开
Ctrl+P	打开最近打开的文件
Alt+Z	切换自动折行
Alt+F12	速览定义（如函数的定义）
Ctrl+Shift+\	跳转到匹配括号
Ctrl+T	在工作区中查找符号（在文件夹中查找指定名称函数等）

多光标

按住 Alt 并单击即可在编辑器中添加光标，多数编辑操作都可同时进行；按住鼠标中键并在编辑器中移动也可添加多行光标，如图：

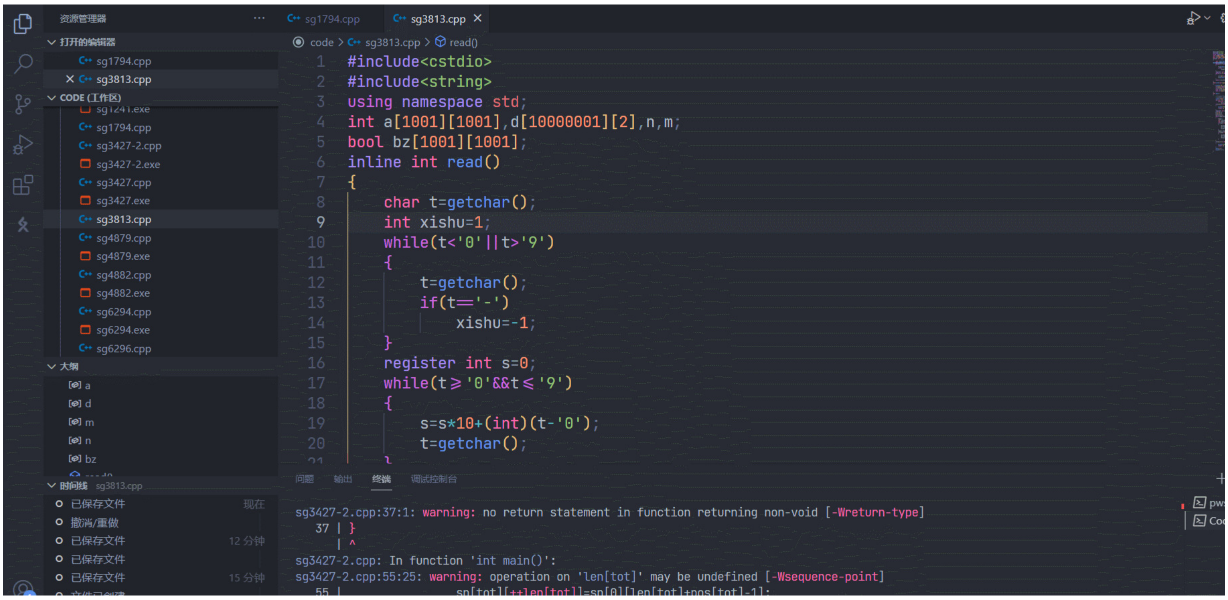


图 3.11

按 Ctrl+F2 可在编辑器中同时更改所有匹配项，如图：

注意此时在右上角会有一个工具栏，可在其中开启查找匹配项时是否开启大小写匹配、全字匹配等。

参考资料与注释

[1] VS Code 的 C/C++ 扩展如果选择 lldb 作调试器，则会默认采用 lldb-mi 程序，而它已经被 LLVM 开发团队从项目中分离出来，需要自己编译该程序。而它本身就有一些 bug，使用体验和方便程度都不如 CodeLLDB 扩展。

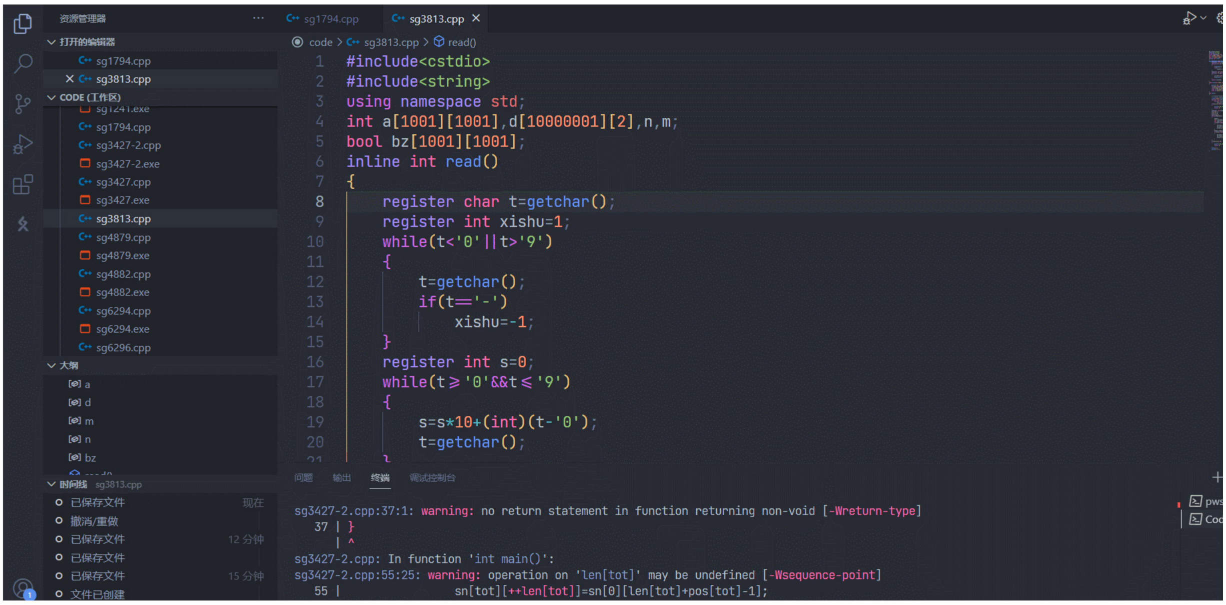


图 3.12

[2] 从扩展商店安装 CodeLLDB 后它会再从 GitHub 下载本体，下载速度奇慢，有时下载出错，所以最好直接下载本体然后安装。更新也可直接按照以上步骤下载安装。

[3] Visual Studio Code - Code Editing. Redefined

[4] Marketplace

[5] CodeLLDB

[6] Getting started

3.2.4 Atom

Authors: ouuan, ChungZH, partychicken, Xeonacid, Find-NICK

Atom，GitHub 家的编辑器。

简介

Atom 是一个免费、开源、跨平台的文本编辑器，由 GitHub 开发。它是用 JavaScript 编写的，并且采用 Electron 架构。它的一个较大缺点就是性能差。

warning

在 2022 年 6 月 8 日，Github 宣布将放弃 Atom 编辑器[2]，并于 2022 年 12 月 15 日对 Atom 编辑器存档。如果你仍然喜欢 Atom 的界面，可以使用 Pulsar[3] 作为替代品，它的用户界面与 Atom 基本相似，目标是让他们最喜欢的编辑器保持活力"keep their favorite editor alive"[1-1]。

外部链接

- Atom 官网[4]
- Pulsar 官网[3]

参考资料

[1] 来源：https://pulsar-edit.dev/about.html#the-team [1-1] [1-2]

[2] Github 宣布将放弃 Atom 编辑器

[3] Pulsar [3-1] [3-2]

[4] Atom 官网

3.2.5 Eclipse

Authors: ouuan, Doveqise, partychicken, Xeonacid, StudyingFather

介绍

Eclipse 是著名的跨平台开源集成开发环境（IDE）。最初主要用来 Java 语言开发，当前亦有人通过插件使其作为 C++、Python、PHP 等其他语言的开发工具。

Eclipse 的本身只是一个框架平台，但是众多插件的支持，使得 Eclipse 拥有较佳的灵活性，所以许多软件开发商以 Eclipse 为框架开发自己的 IDE。

Eclipse 最初是由 IBM 公司开发的替代商业软件 Visual Age for Java 的下一代 IDE 开发环境，2001 年 11 月贡献给开源社区，现在它由非营利软件供应商联盟 Eclipse 基金会（Eclipse Foundation）管理。[1]

缺点：

- 实测这个 IDE 打开速度比 Visual Studio 慢
- 更新速度玄学，插件更新速度跟不上 IDE 的更新速度，对于经常更新的同学很不友好。

优点：

- 使用体验较好
- 能够快速上手，所以比较推荐 OIer 用这个 IDE。

安装 & 配置指南

安装可参照 Eclipse/Installation - Eclipsepedia[3]。

安装后如图填写目录信息以建造项目：

拓展

这个软件的帮助手册很详细，建议刚接触的同学多看帮助手册，多百度，并且这个 IDE 的使用手感与 Visual Studio 相近。

和 VS Code 类似，Eclipse 中也提供了很多插件，这些插件可以让 Eclipse 变得更加易用。[2]

参考资料与注释

[1] Eclipse - 维基百科

[2] 曾经的 Java IDE 王者 Eclipse 真的没落了？21 款插件让它强大起来！

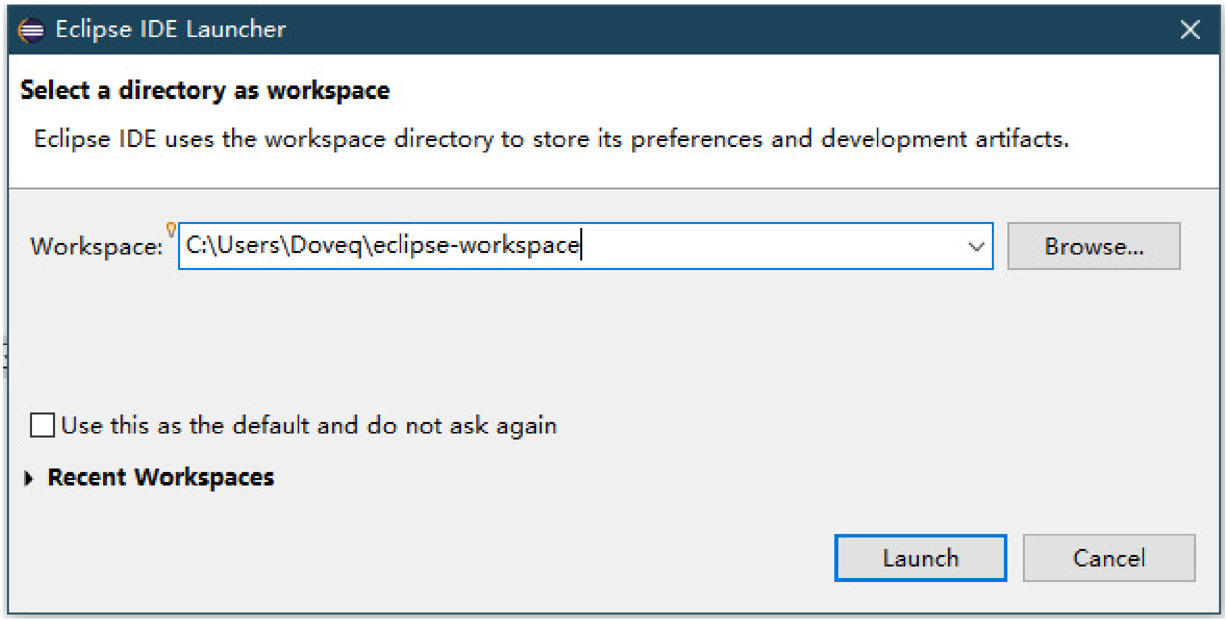

Eclipse IDE Launcher
Select a directory as workspace
Eclipse IDE uses the workspace directory to store its preferences and development artifacts.
Workspace:
C:\Users\Doveq\eclipse-workspace
Browse...
Use this as the default and do not ask again
Recent Workspaces
Launch
Cancel

图 3.13

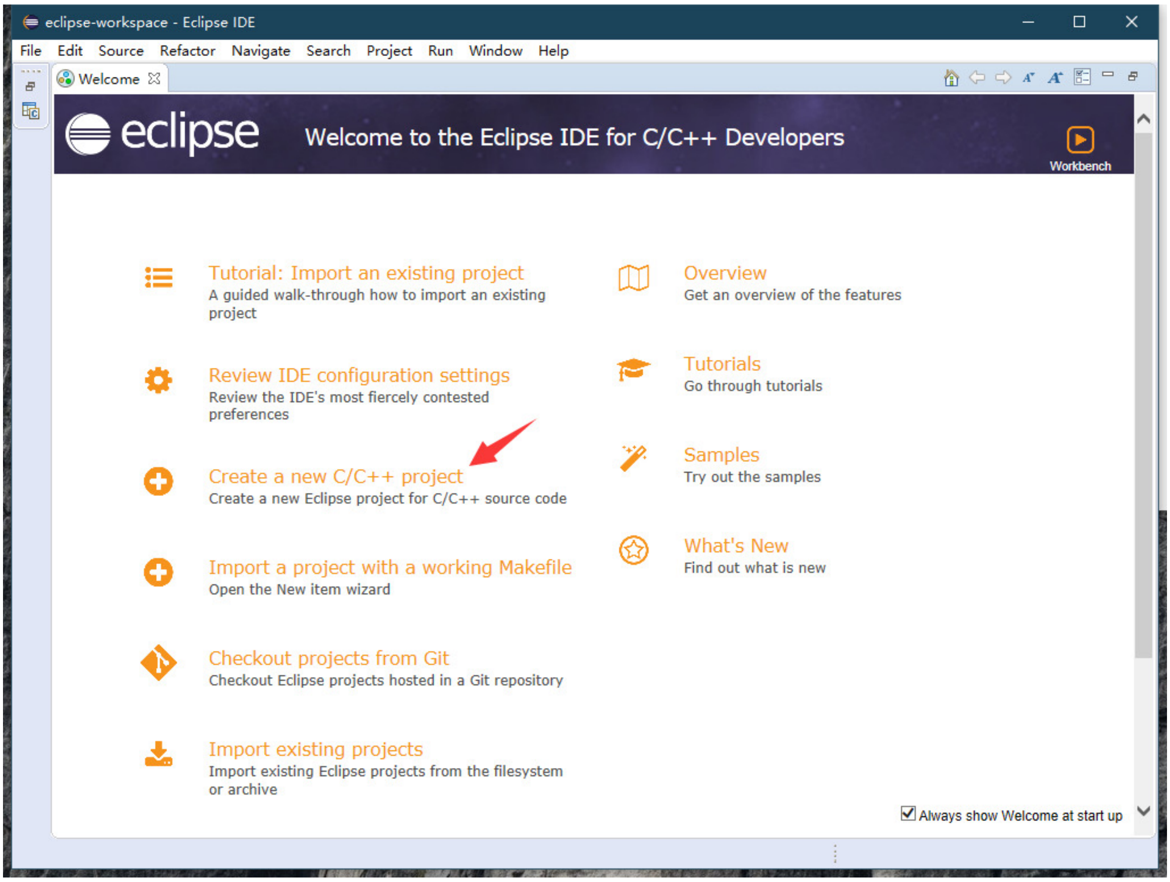

eclipse-workspace - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Welcome
eclipse
Welcome to the Eclipse IDE for C/C++ Developers
Workbench
Tutorial: Import an existing project
A guided walk-through how to import an existing project
Overview
Get an overview of the features
Review IDE configuration settings
Review the IDE's most fiercely contested preferences
Tutorials
Go through tutorials
Create a new C/C++ project
Create a new Eclipse project for C/C++ source code
Samples
Try out the samples
Import a project with a working Makefile
Open the New item wizard
What's New
Find out what is new
Checkout projects from Git
Checkout Eclipse projects hosted in a Git repository
Import existing projects
Import existing Eclipse projects from the filesystem or archive
Always show Welcome at start up

图 3.14

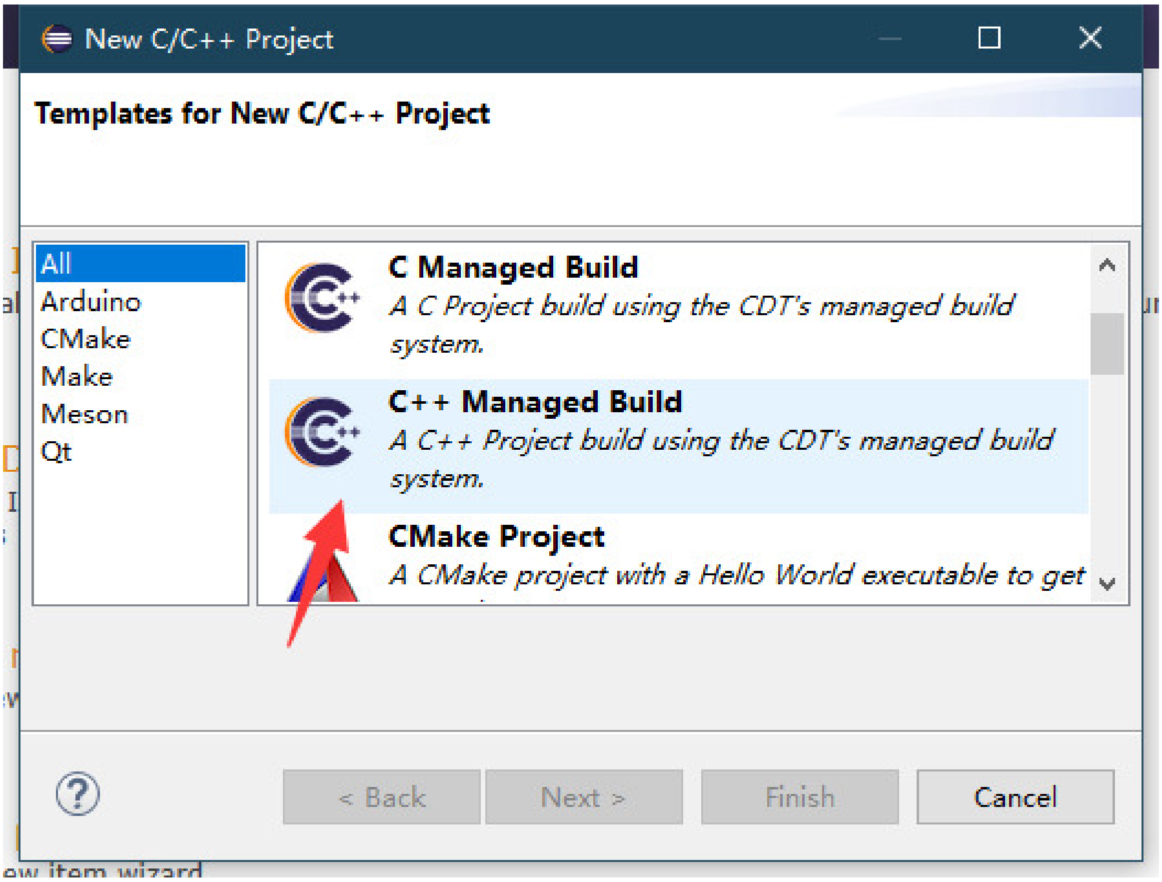

New C/C++ Project
Templates for New C/C++ Project
All
Arduino
CMake
Make
Meson
Qt
C Managed Build
A C Project build using the CDT's managed build system.
C++ Managed Build
A C++ Project build using the CDT's managed build system.
CMake Project
A CMake project with a Hello World executable to get
< Back
Next >
Finish
Cancel

图 3.15

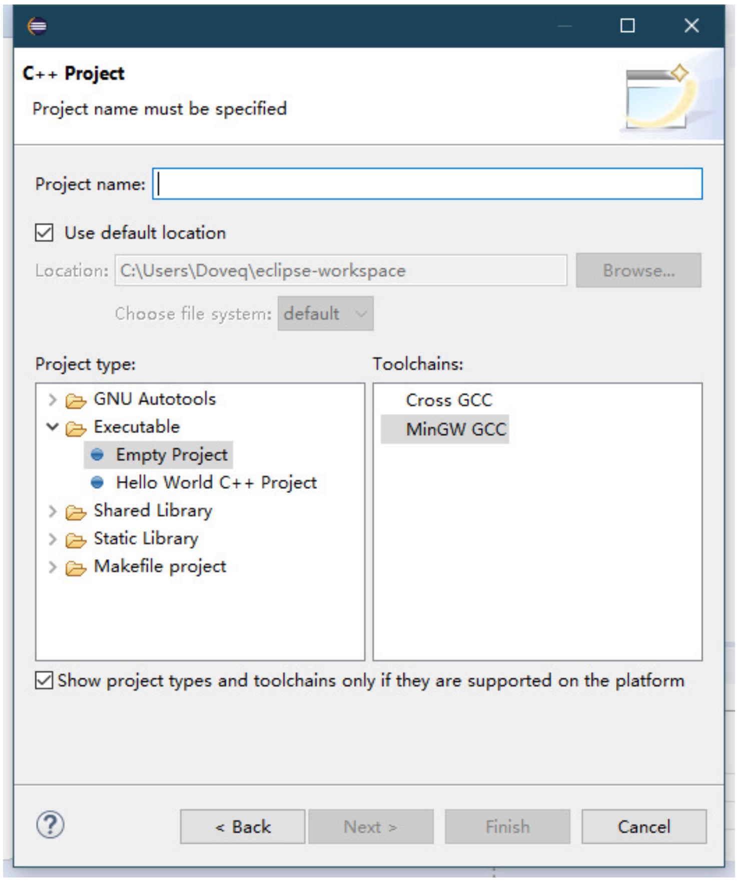

C++ Project
Project name must be specified
Project name:
Use default location
Location: C:\Users\Doveq\eclipse-workspace
Browse...
Choose file system: default
Project type:
Toolchains:
GNU Autotools
Executable
Empty Project
Hello World C++ Project
Shared Library
Static Library
Makefile project
Cross GCC
MinGW GCC
Show project types and toolchains only if they are supported on the platform
< Back
Next >
Finish
Cancel

图 3.16

[3] Eclipse/Installation - Eclipsepedia

3.2.6 Notepad++

Authors: ouuan, CBW2007, partychicken, StudyingFather, Xeonacid, Henry-ZHR

软件简介

Notepad++ 是 Windows 操作系统下的文本编辑器，支持多国语言、多种编码、多种编程语言的高亮和补全。它的 logo 也十分可爱，是一只变色龙（

图 3.17 npp-logo

）

其功能比其他许多编辑软件强大许多，打开大文件时更加稳定，不断撤销不会出问题。关闭时也不需要保存，它会自动为你保存在缓冲区中。（可能需要配置）而且，它十分小巧，只有 10MB+，甚至可以放在 U 盘中随身携带。

下载与安装

参见 Getting started | Notepad++ User Manual[1]。

更改界面语言

语言改完了，就可以随心所欲地魔改编辑器啦！

初级玩法

这里主要讲一些基础和特色功能。

查找与替换

依次单击「（菜单栏）搜索」->「查找」（快捷键 `CTRL+F`）即可打开「查找」页面（如下图）。

依次单击「（菜单栏）搜索」->「替换」（快捷键 `CTRL+H`）即可打开「替换」页面（如下图）。

查找、替换之间其实是一个窗口，单击上面的标签页就可以完成切换。

其功能与普通编辑器大同小异，但是支持更多，如：

1. 严格匹配或大小写匹配等
2. 跨文档匹配
3. 转义字符，如'\r'，'\n'。
4. 正则表达式
5. 计数

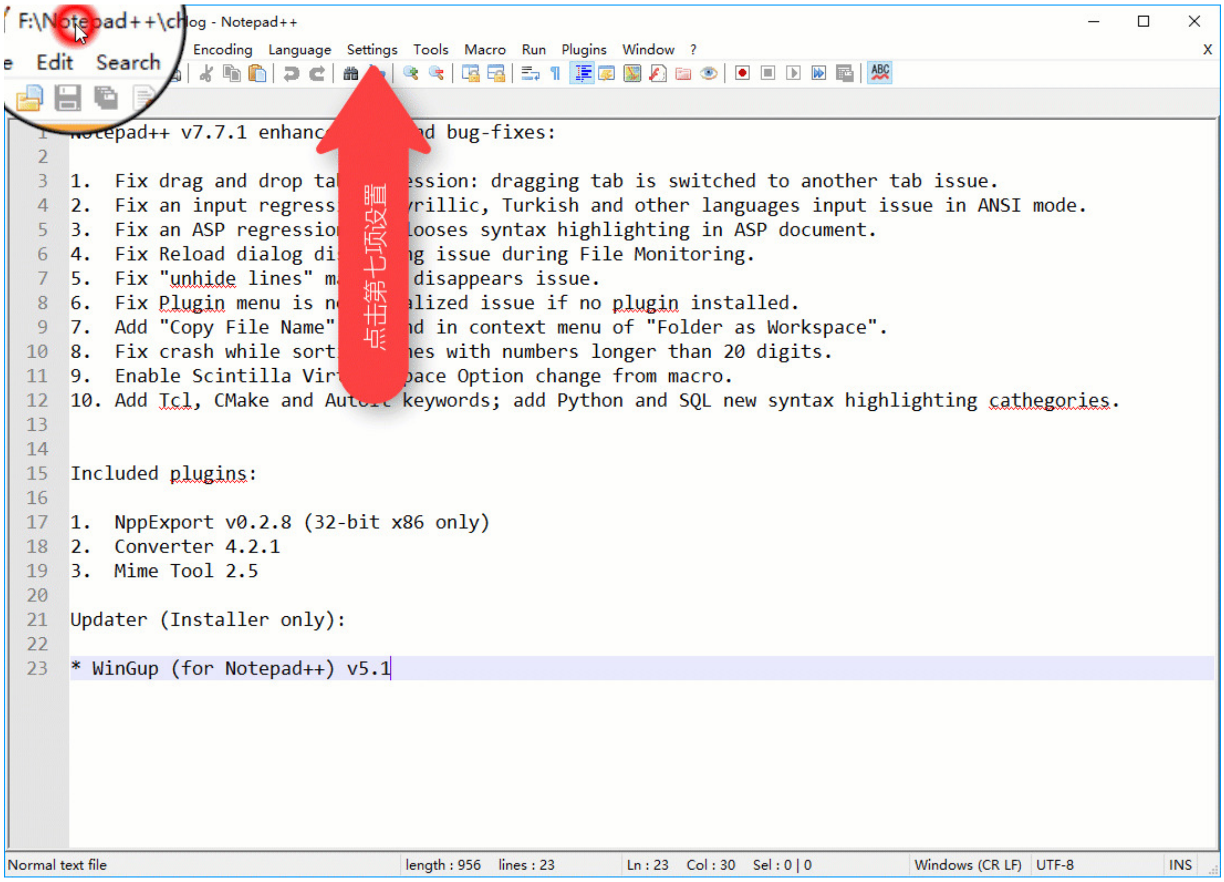


图 3.18 npp-lang

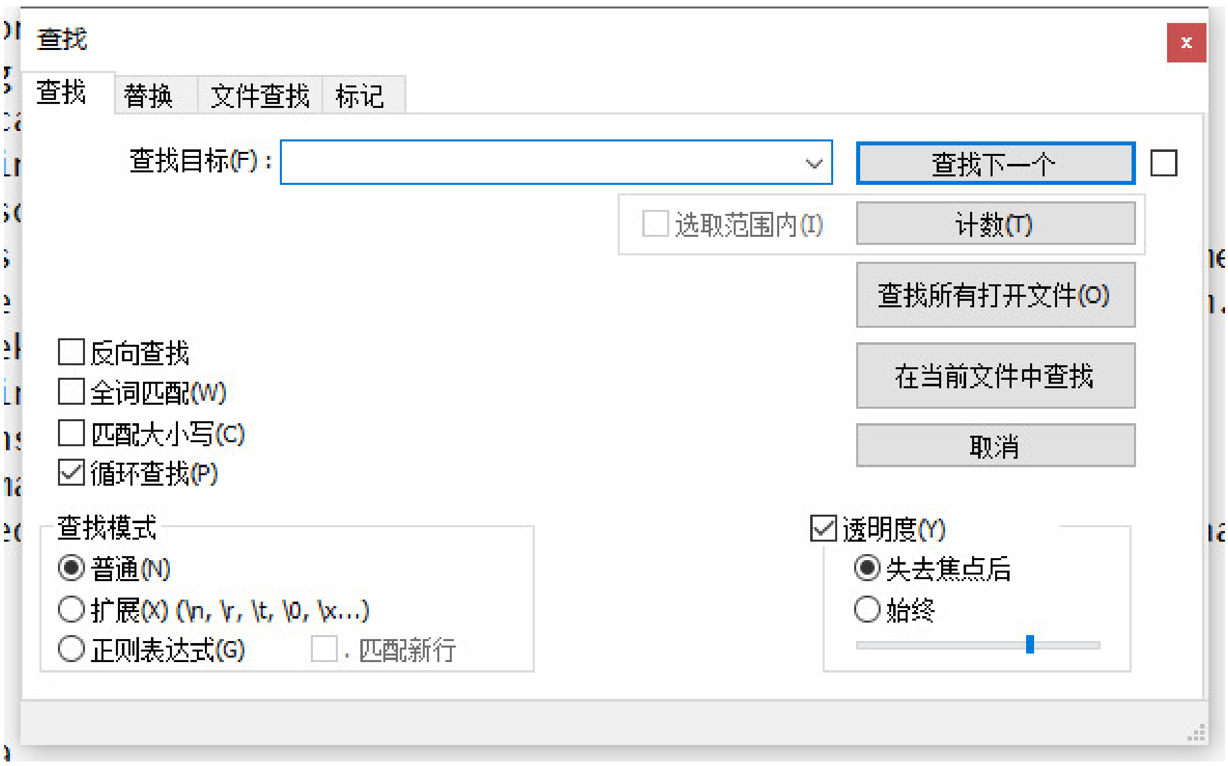


图 3.19 npp-search

图 3.20 npp-replace

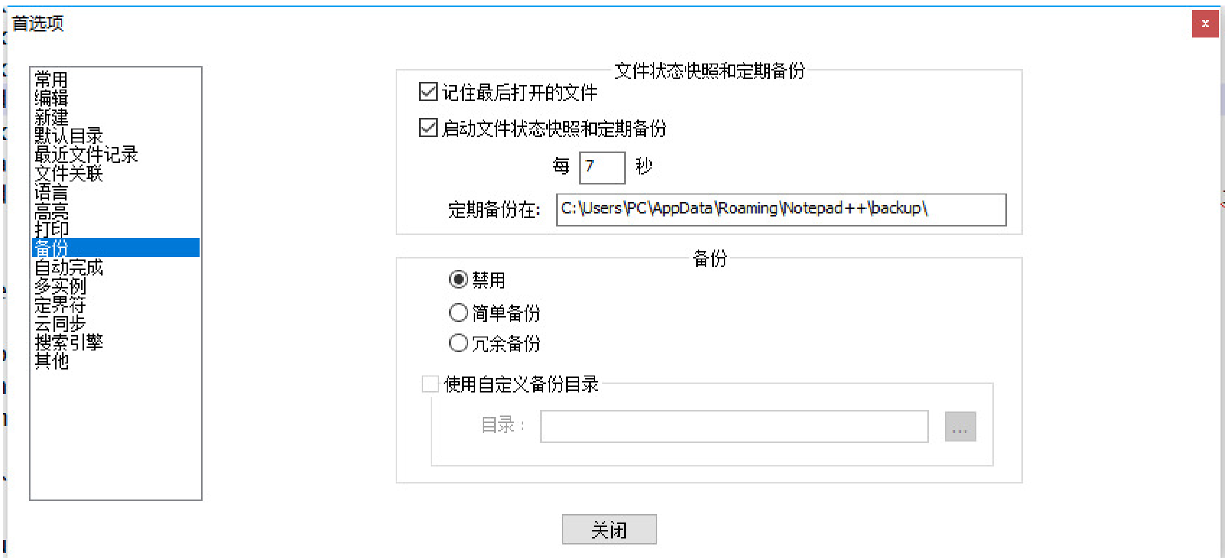


图 3.21 npp-settings-1

定期备份

有了这个功能，就可以不用费心地担心意外情况代码丢失啦！

但是，这个功能只是为你的文件拍了一个快照，并没有真正保存，所以还是建议要有良好的保存习惯。或者说可以去自带插件商店安装"Auto Save" 插件（详见高级玩法 -> 插件，下同）

书签功能

在你需要的行按 `Ctrl+F2` 即可设放置/取消书签，放置过书签的行前段有一个蓝色圆点。

按 `F2` 可以定位到下一个书签。

如果你抱怨不方便，可以去自带插件商店安装"Bookmarks" 插件

代码高亮

右击左下角的"XXX file"，可以选择许多种语言高亮，C、C++、PASCAL、Markdown 等任你挑选。你甚至可以自己定义高亮！

如果你认为每一次打开文件都要更改高亮很麻烦，可以在「设置 -> 首选项 -> 新建 -> 默认语言」中修改默认高亮。

需要渲染 Markdown 的，可以去插件商店安装"Markdown Viewer"，还有更多类似插件等着你！

显示所有字符

图 3.22 npp-settings-2

点击红框所圈的按钮，就可以非（za）常（luan）清（wu）晰（zhang）地显示出「空格」、「TAB」、「换行」等原来不可见字符。

自动识别文件编码与换行符

Notepad++ 可以自动识别当前文件编码是 `UTF-8` 还是 `GB2312` 甚至其他。再也不用担心被锟斤拷抡死或被烫烫烫烫死了。

如果要使用不同的编码浏览文章，请依次单击「（菜单顶栏）编码」->「使用 XXX 编码」。如果想给文件换一个字符编码，请依次单击「（菜单顶栏）编码」->「转为 XXX 编码」。

它还可以自动识别换行符是 `CR`、`LF` 或 `CRLF`。不用担心下载下来的数据被吞换行。

在底部信息栏，你可以看到"Windows(CR LF)" 等字样，这就是当前文件的换行符。右击它，可以改变当前文件换行符。此操作配合「显示所有字符」更直观哟！

高级玩法

这个就适用于需求较高的用户。

宏

宏可以帮助你完成许多重复的工作，例如，将奇数行的「abcde」改为「afce」，需要两步。

录制宏

使用宏

大量处理，重复使用

如果是更多行呢？操作就需要一点改变。

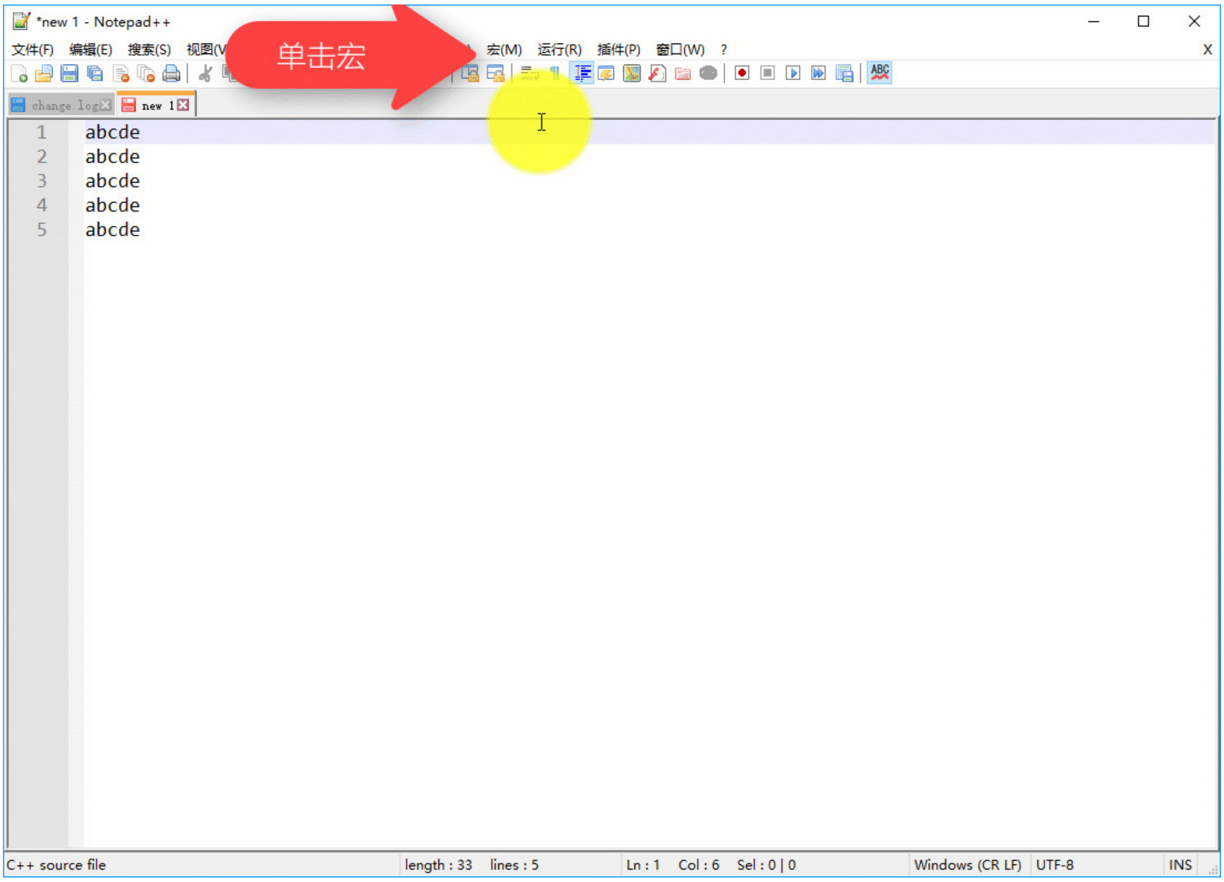


图 3.23 npp-macro-rec

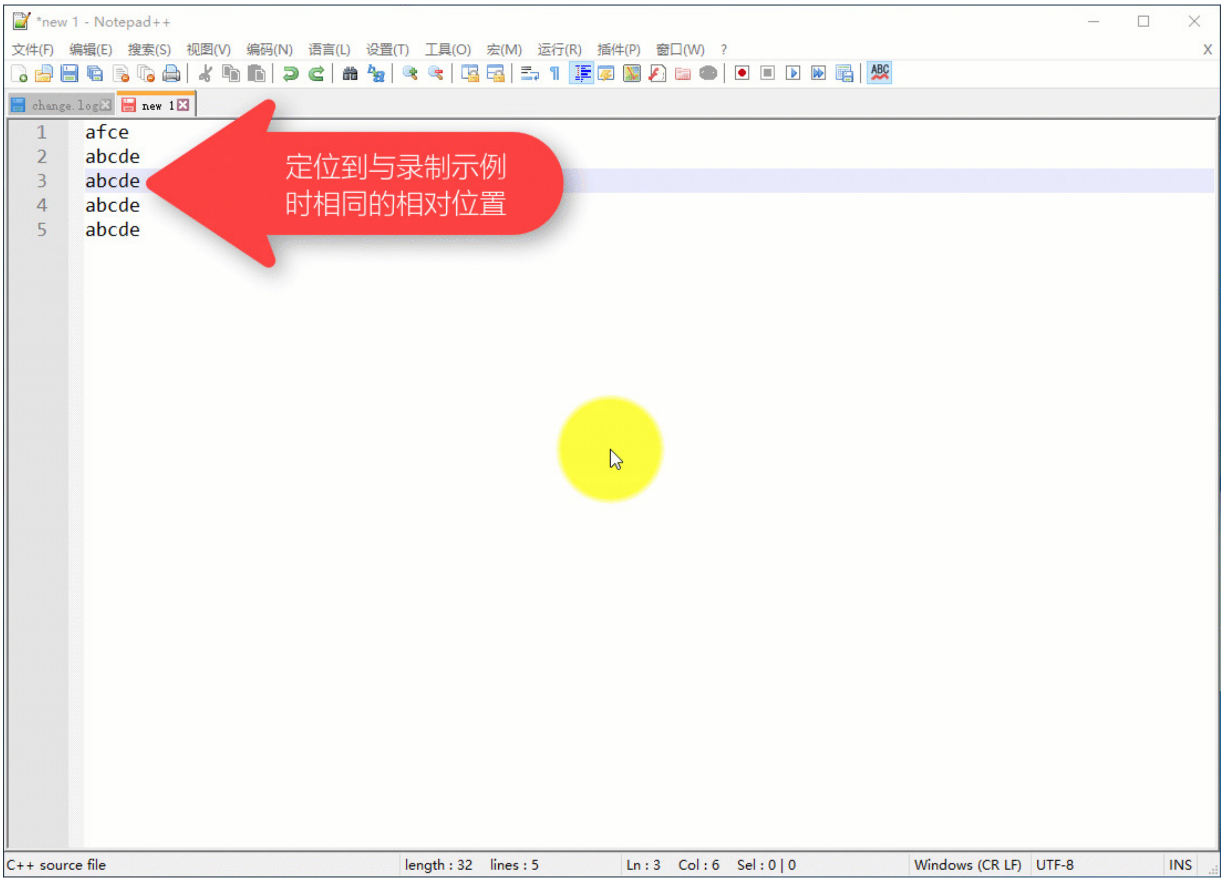


图 3.24 npp-macro-use

首先是录制，一定要先按键盘上的 `HOME` 或 `END` 键将光标移动到行首或行尾，然后用方向键调整横向位置，再进行更改。最后一定要用方向键将光标移动到下一个要处理的行。

比如刚刚的例子，可以先按 `END` 键，然后依次按 ← ，`Backspace`，← ,`Backspace`,`F`，最后按两下 ，最后停止录制。

然后是重播，先定位到第一个要处理的行（第 3 行），然后点击「宏」->「重复运行宏」。在弹出窗口设置要运行的宏（刚录制的一般是第一个），设置运行次数（或者直接运行到文件尾），点确定即可。

保存宏

点击「宏」->「保存录制宏」，并设置名称和快捷键，即可保存，方便后续使用。

插件

插件管理

打开功能栏的「插件」按钮，列表中会显示所有你安装过的插件。

再选择「插件管理」选项，即可管理你的插件。

安装插件（商店）

1. 打开「可用」选项卡，在列表中勾选你所要的插件
2. 点击右上角的「安装」按钮，按照提示重启软件即可。

安装插件（手动）

1. 下载插件（由第三方托管的官方地址：https://sourceforge.net/projects/npp-plugins/[2]）注意一定要选择**与安装 Notepad++ 时处理器架构相同**的插件。
2. 找到一个名为"XXX.dll" 的文件（通常以插件名命名）。
3. 在 Notepad++ 中的功能栏点插件，并在列表中点「打开插件文件夹」。
4. 将刚才找到的 DLL 文件放入文件夹中，重启 Notepad++。
5. 【可选】删除刚才拷贝的文件，**但不要删除生成的文件夹！**

Tips: 如果多次不成功，可以尝试新建一个与插件名相同的文件夹在将".dll" 文件放入创建的文件夹中

更新插件

在插件管理中，选择「更新」选项卡，并勾选要更新的插件，然后点右上角的「更新」按钮。

移除插件

同样在插件管理中，选择「已安装」选项卡，并勾选要移除的插件，然后点右上角的「移除」按钮。

搭建开发环境

不只是编辑器！"Notepad++" 可谓神一样的存在，它可以通过傻瓜式地编译代码，甚至代替 IDE！这里以 C++ 为例

1. 安装编译器并将其必要的文件目录添加到 PATH 环境变量中。（C++ 需要添加%APPPATH%\bin）当你在 cmd 中输入 g++ 时不再提示'g++' 不是内部或外部命令⋯⋯即可（中间可能需要重启电脑）。推荐 下载 ConsolePauser[3] 然后随便放并将其目录添加到环境变量（此为 Dev-C++ 的插件，在 Dev-C++ 软件根目录也有）。
2. 在菜单栏中选择「运行」->「运行⋯⋯」，打开「运行」窗口。
3. 分别输入以下命令

```
# 编译命令:
cmd /c g++.exe -o $(CURRENT_DIRECTORY)\$(NAME_PART).exe $(FULL_CURRENT_PATH)
```

```
# 运行命令:
cmd /c $(CURRENT_DIRECTORY)\$(NAME_PART).exe $(FULL_CURRENT_PATH) & pause
# 调试命令:
cmd /c gdb $(CURRENT_DIRECTORY)\$(NAME_PART).exe

# 如果下载了 ConsolePauser 可以使用下列代码获得更好的程序运行体验!(注意添加环境变量
!)

# 编译命令:
cmd /c (start ConsolePauser "g++.exe -o $(CURRENT_DIRECTORY)\$(NAME_PART).exe $(
FULL_CURRENT_PATH)")
# 运行命令:
cmd /c (start ConsolePauser "$(CURRENT_DIRECTORY)\$(NAME_PART).exe")
# 调试命令:
cmd /c (start ConsolePauser "gdb $(CURRENT_DIRECTORY)\$(NAME_PART).exe")
```

4. 单击「保存」，名字可以自己取，如”Compile”,”Run” 等，然后设定好你想要的快捷键（捡好记的来，如 Dev-C++ 就分别是 `F9` 和 `F10`）。
5. Enjoy it!

小彩蛋

1. 在运行安装程序时你会在下方看到这样一句话：

   > ”The best things in life are free. Notepad++ is free. So Notepad++ is the best(.)”
   > （生活中最好的事情都是免费的。Notepad++ 是免费的。所以 Notepad++ 是最好的。）

   这牛吹的，不得不说，很有底气。
2. 在一个新开的页面中输入”random” 并选中，再按 `F1` 就会得到一句很有意思的话。

参考资料与注释

[1] Getting started | Notepad++ User Manual

[2] https://sourceforge.net/projects/npp-plugins/

[3] 下载 ConsolePauser

3.2.7 Kate

> **Authors:** CoelacanthusHex

软件简介

Kate 是一个具有众多功能的跨平台文本编辑器。Kate 还附带了多种插件，包括一个嵌入式终端，可以让你直接从 Kate 中启动控制台命令，强大的搜索和替换插件，以及一个预览插件，可以渲染 MD、HTML 甚至 SVG 文件。支持通过交换文件在系统崩溃时恢复数据，带参数提示的自动补全，同时支持 LSP (Language Server Protocol)[4] 以获得更为强大的补全。

下载与安装

可打开 Kate 官网[5]，然后进入 获取页面[6]。随后，根据你使用的系统和喜欢的安装方式进行安装。

用法与功能

交换文件防止数据丢失

与 Vim 类似，Kate 会将未保存的更改写入一个交换文件（一般是原文件名前面加点后面加 `.kate-swp`），如果遭遇断电或程序崩溃等意外，下次启动时不会丢失未保存的更改。

代码高亮

Kate 支持三百余种语言的语法高亮。一般来说，Kate 可以自动地选择对应的语言进行语法高亮，不过偶尔也有错误的时候，这时候可以点击最右下角的按钮，选择正确的语言。

自己编写语法高亮文件

尽管 Kate 支持超过三百种语言的语法高亮，但是仍不免有语言未被覆盖到，此时可以自己动手编写语法高亮文件。Kate 自身自带的文件位于 Syntax Highlighting Powered By KSyntaxHighlighting Framework[7]，语法可参照 Working with Syntax Highlighting[8]，编写好的文件根据 Syntax definition files[9] 放置。CoelacanthusHex/dotfiles@80a913c/pam_env.xml[10] 有笔者编写的一个配置文件可供参考。

切换语言

点击上方工具栏里的设置/`Setting`，然后点击配置语言/`Configure Language`，随后选择语言即可，注意可以选择备选语言。

编码与行尾符

Kate 可以自动识别当前文件使用的是什么编码，如果识别错误，可以点击右下角倒数第二个按钮，选择正确的编码。

同时，Kate 也可以自动识别当前文件使用的行尾符，如果识别错误，可以点击工具→行尾/`Tool → End of line` 选择正确的行尾符。

查找与替换

依次单击编辑→查找（快捷键 Ctrl+F）即可打开「查找」页面。依次单击编辑→替换（快捷键 Ctrl+R）即可打开「查找与替换」页面。同时，点击左下角搜索与替换也可打开「查找与替换」页面。

具体操作和其他编辑器并无太大差别，但是支持一些额外的特性，例如：

1. 是否区分大小写
2. 支持正则表达式（包括捕获组）
3. 从当前文件到多文件再到当前工程不等的范围
4. 对查找的结果进行选择替换

Language Server Protocol

Kate 自 19.12 起支持 LSP Client，最初仅支持 C/C++、D、Fortran、Go、Latex/BibTeX、OCaml、Python、Rust，现如今支持如下表中的语言：

语言	LSP Server
Bash	bash-language-server[11]
LaTeX	texlab[12]

语言	LSP Server
BibTeX	texlab[12]
C	clangd[13]
C++	clangd[13]
D	serve-d[14]
Fortran	fortls[15]
Go	gopls[16]
Haskell	haskell-language-server-wrapper[17]
JavaScript	typescript-language-server[18]
OCaml	ocamllsp[19]
Perl	Perl-LanguageServer[20]
Python	pyls[21]
Rust	rls[22]
TypeScript	typescript-language-server[18]
R	RLanguageServer[23]
zig	zls[24]

要启用 LSP 相关特性，需要前往菜单栏中设置→配置 `Kate` 然后选择插件中 `LSP` 客户端以启用相关特性。当打开对应语言的文件时，Kate 会自动拉起对应的 LSP Server。

增加配置

此外，用户还可以手动编写配置，具体格式为：

```
{
    "servers": {
        "bibtex": {
            "use": "latex",
            "highlightingModeRegex": "^BibTeX$"
        },
        "c": {
            "command": ["clangd", "-log=error", "--background-index"],
            "commandDebug": ["clangd", "-log=verbose", "--background-index"],
            "url": "https://clang.llvm.org/extra/clangd/",
            "highlightingModeRegex": "^(C|ANSI C89|Objective-C)$"
        },
        "cpp": {
            "use": "c",
            "highlightingModeRegex": "^(C\\+\\+|ISO C\\+\\+|Objective-C\\+\\+)$"
        },
        "haskell": {
            "command": ["haskell-language-server-wrapper", "--lsp"],
            "rootIndicationFileNames": ["*.cabal", "stack.yaml", "cabal.project"
, "package.yaml"],
```

```
                "url": "https://github.com/haskell/haskell-language-server",
                "highlightingModeRegex": "^Haskell$"
            },
            "latex": {
                "command": ["texlab"],
                "url": "https://texlab.netlify.com/",
                "highlightingModeRegex": "^LaTeX$"
            },
            "rust": {
                "command": ["rls"],
                "rootIndicationFileNames": ["Cargo.lock", "Cargo.toml"],
                "url": "https://github.com/rust-lang/rls",
                "highlightingModeRegex": "^Rust$"
            }
        }
}
```

其中 `server` 里的每一项代表一种语言，在这个语言里，`command` 代表启动 LSP Server 所使用的命令，`command` 是一个数组，是所需要执行的命令以空格分词的结果；`url` 是 LSP 的网址；`rootIndicationFileNames` 是用于确定项目根目录的文件；`highlightingModeRegex` 则匹配某种语法高亮的名字，以确定使用哪个 LSP；如果存在 `use` 项，则代表使用 `use` 项对应的语言的配置。

该配置项位于设置→配置 `Kate → LSP 客户端`→用户服务器设置，其中 `LSP 客户端`部分要在插件中启用 `LSP 客户端`插件后才可见。

内置终端

" 注意"

内置终端依赖了 KDE 的 Konsole[1]，而 Konsole 为 * nix 独有包。也就是说，Windows 下该特性不可用。

按 F4 可打开或关闭内置终端，也可点击左下角终端按钮打开，内置终端的当前目录会自动与当前文件保持一致，并随着你选择的文件而改变。其余与一般终端并无太大不同。

外部工具

点击工具→外部工具可执行。

点击工具→外部工具→配置可以配置外部工具。

添加外部工具

从预置配置中添加

进入配置页面后，点击左下角添加→从默认工具添加，然后点击对应工具即可。

手写配置添加

进入配置页面后，点击左下角添加→添加工具，然后按提示填写即可。可以参照 此文档 （ 英文 ）[25] 来编写自己的外部工具配置。注意可点击如下标志查看可使用的变量。

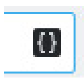


图 3.25

常用的外部工具

编译并执行单个 C++ 文件

在 * nix 系统下，打开任意 C++ 源文件，在外部工具里找到编译执行 `cpp`，点击即可。

" 对于 Windows 用户"

在默认情况下，由于该工具的可执行文件为 `sh`，使得该工具在 Windows 下不可用。然而，用户可以对该工具进行修改，使其可用于 Windows 系统。

要进行修改，请先确保你的系统内有一个可用的 C++ 编译器。然后从默认工具添加编译运行 `cpp`，将其中可执行文件从 `sh` 改为 `powershell`，参数改为 `-ExecutionPolicy Bypass -Command "g++ %{Document:FilePath} -o %{Document:FileBaseName}.exe;./%{Document:FileBaseName}.exe"` [2][3] 即可。

Git Blame

打开任意文件，在外部工具里找到 `git blame`，点击之后，会打开一个窗口，展示 git blame 的结果。

格式化

格式化功能要求对应包或应用程序可用，例如，C/C++ 的格式化要求 `clang-format` 可用。对于其他语言，用户可以前往外部工具配置中查看其默认可执行文件作为参考。

打开任意源文件，在外部工具里找到用 `xxx` 格式化，点击即可。另外，对于 C/C++ 语言的源文件，`clang-format` 可格式化选中的文本。

Git Blame

要启用该特性，需要前往菜单栏中设置→配置 `Kate` 然后选择插件中 `Git Blame`。

启用该特性后，Kate 会在每一行后面以较浅字体显示在 Git 中最后于什么时间被谁修改，将鼠标移动到文字上会出现一个悬浮窗显示 commit 的具体信息。

相关外部链接

- The Kate Handbook[26]
- 关于如何手写自己的 LSP 客户端配置（英文）[27]
- 关于如何手写自己的外部工具配置（英文）[25]

参考资料与脚注

[1] Arch Linux 中对该包的描述 中，其可选依赖了 `konsole`，描述为 `open a terminal in Kate`（在 Kate 中打开一个终端）。

[2] 若 `g++` 不在 `PATH` 环境变量中，则将其改为编译器的绝对路径

[3] 或者，如果使用 Clang，则将 `g++` 改为 `clang++`。

[4] LSP (Language Server Protocol)

[5] Kate 官网

[6] 获取页面

[7] Syntax Highlighting Powered By KSyntaxHighlighting Framework

[8] Working with Syntax Highlighting

[9] Syntax definition files

[10] CoelacanthusHex/dotfiles@80a913c/pam_env.xml

[11] bash-language-server

[12] texlab [12-1] [12-2]

[13] clangd [13-1] [13-2]

[14] serve-d

[15] fortls

[16] gopls

[17] haskell-language-server-wrapper

[18] typescript-language-server [18-1] [18-2]

[19] ocamllsp

[20] Perl-LanguageServer

[21] pyls

[22] rls

[23] RLanguageServer

[24] zls

[25] 此文档（英文） [25-1] [25-2]

[26] The Kate Handbook

[27] 关于如何手写自己的 LSP 客户端配置（英文）

3.2.8 Dev-C++

Authors: ksyx, ouuan, Doveqise, hsfzLZH1, wangqingshiyu, sshwy, NanoApe, DawnMagnet, CamberLoid, royqh1979

介绍

Dev-C++ 是一套用于开发 C/C++ 程序的自由的集成开发环境（IDE），并以 GPL 作为分发许可，使用 MinGW 及 GDB 作为编译系统与调试系统。Dev-C++ 运行在 Microsoft Windows 下。

Dev-C++ 的优点在于界面简洁友好，安装便捷，支持单文件编译，因此成为了许多入门 OI 选手以及 C++ 语言

初学者的首选。在 NOIP 中，提供 Windows 作为比赛系统的省份一般预置 Dev-C++。

Dev-C++ 起源于 Colin Laplace 编写的 Bloodshed Dev-C++。该版本自 2005 年 2 月 22 日停止更新。2006 年，Dev-C++ 主要开发者 Colin Laplace 曾经对此作出了解释：「因忙于现实生活的事务，没有时间继续 Dev-C++ 的开发。」

Orwell Dev-C++ 是 Dev-C++ 的一个衍生版本，由独立程序员 Orwell (Johan Mes) 开发并维护。其对原版 Dev-C++ 进行了错误修正，并更新了编译器版本。一般而言，Dev-C++ 5.x 均为 Orwell Dev-C++。其最后一次更新于 2015 年，版本为 5.11。

Embarcadero Dev-C++[1] 是 Bloodshed Dev-C++ 和 Orwell Dev-C++ 的继任者。2020 年，Embarcadero 赞助并接手了原有的 Dev-C++ 项目，继续开发。Embarcadero Dev-C++ 加入了对高 DPI 的支持，更新了编译器以加入更新版本的 C++ 标准支持，以及暗色模式。

以上的 Dev-C++ 分发都被认为是「官方的」。此外，在 2015 年 Orwell Dev-C++ 停止更新后，因为教学需要，一位来自中国的个人开发者 royqh1979[4] 决定继续开发他的 Dev-C++ 个人分支，命名为小熊猫 Dev-C++[2]，集成了智能提示和高版本的 MinGW64，非常便于国内的个人使用和学习。

小熊猫 Dev-C++ 6.7.5 版本发布后，作者使用 qt5 开发了全新的小熊猫 C++[3]，可在 windows、linux 和 macos 等系统下原生运行。小熊猫 C++ 的界面与 Dev-C++ 相似，除了提供和 Dev-C++ 相似但更加完善的单文件编译、调试、语法高亮、搜索/替换等功能外，还提供了诸如**暗色主题、代码智能提示、变量/函数重命名、切换/自动识别文件编码**等现代 IDE 常见的基本功能。此外小熊猫 C++ 还具备与 CP Editor 类似的试题集功能，可以自行编写或**从常见的 OJ 竞赛网站上下载试题样例，自动运行和测试程序**。

使用教程

常用快捷键

文件部分

- `Ctrl + N`: 创建源代码
- `Ctrl + O`: 打开文件
- `Ctrl + W`: 关闭文件
- `Ctrl + P`: 打印文件

格式部分

- `Ctrl + /`: 注释和取消注释
- `Tab`: 缩进
- `Shift + Tab`: 取消缩进

行操作

- `Ctrl + E`: 复制行
- `Ctrl + D`: 删除行
- `Ctrl + Shift + Up`: 向上移动
- `Ctrl + Shift + Down`: 向下移动

跳转部分

- `Ctrl + F`: 搜索
- `Ctrl + R`: 替换
- `F3`: 搜索下一个
- `Shift + F3`: 搜索上一个
- `Ctrl + G`: 到指定行号
- `Shift + Ctrl + G`: 到指定函数

- `Ctrl + [1 ~ 9]`: 设置书签
- `Alt + [1 ~ 9]`: 跳转书签

显示部分

- `Ctrl +` 滚轮：字号放大或缩小
- `Ctrl + F11`: 全屏或恢复

运行部分

- `F9`: 只编译
- `F10`: 只运行
- `F11`: 编译并运行
- `F12`: 全部重新编译

调试部分

- `F2`: 转到断点
- `F4`: 设置断点或取消
- `F5`: 调试运行
- `F6`: 停止
- `F7`: 逐步调试

调试流程

1. 将编译器配置设定为 `TDM-GCC 4.9.2 64-bit Debug`
2. 按 `F4` 设置或取消调试断点
3. 将光标放置在变量上，按 `Alt + A` 向调试窗口添加监控变量
4. 按 `F5` 启动调试
5. 按 `F7` 或 `Alt + N` 逐步调试
6. 按 `Alt + S` 跳至下一个调试断点
7. 按 `F6` 停止调试

扩展

增加编译选项

点击工具 -> 编译选项，然后选择”代码生成/优化”选项卡，下面介绍笔者常用的几个编译选项。

开启优化

优化代码运行时间或占用空间。

选择”代码生成”子选项卡中的”优化级别（-Ox）”选项标签。

更换语言标准

使用新语言特性或试图让代码在旧标准下编译。

选择”代码生成”子选项卡中的”语言标准（-std）”选项标签。

显示最多警告信息

查错小助手。

选择”代码警告”子选项卡中的”显示最多警告信息（-Wall）”选项标签。


编译器选项
设定编译器配置
TDM-GCC 4.9.2 64-bit Release
编译器
代码生成/优化
目录
程序
C 编译器:
代码生成:
代码警告
代码性能:
连接器
输出
生成特定机器的专用指令
最少优化, 保持全部兼容性 (-mtune)
使用处理器内建函数
优化级别 (-Ox)
使用下列位宽编译 (-mx)
语言标准 (-std)
Low
Med
High
Highest (fast)
Size (s)
Debug (g)
获得更多关于GCC的选项, 请参阅
http://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html
确定[O]
取消[C]
帮助[H]

图 3.26

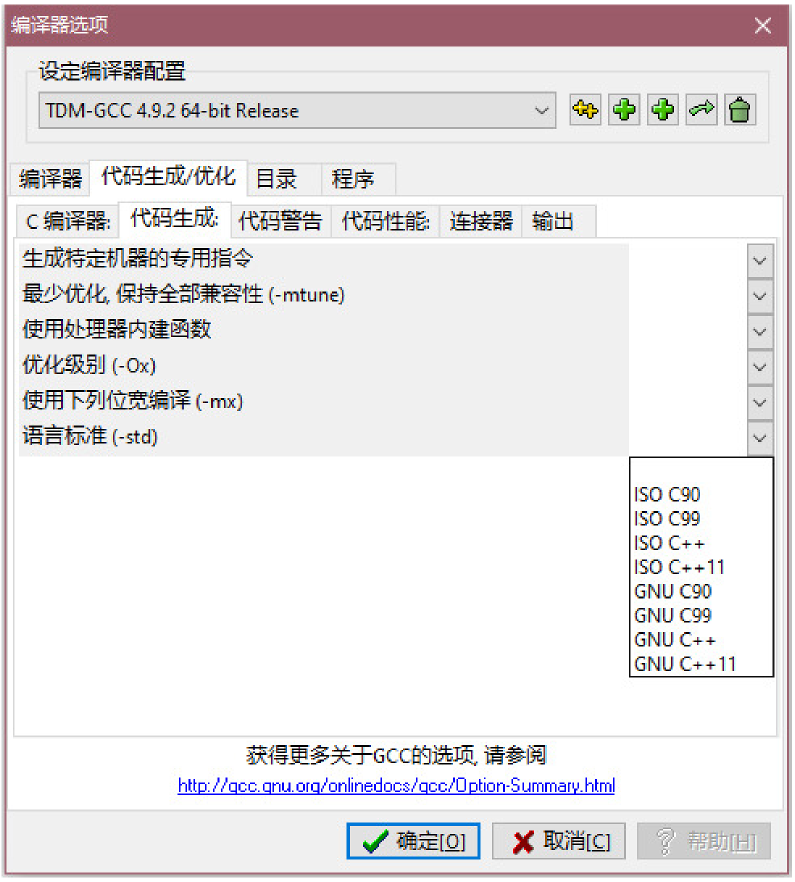

编译器选项
设定编译器配置
TDM-GCC 4.9.2 64-bit Release
编译器
代码生成/优化
目录
程序
C 编译器:
代码生成:
代码警告
代码性能:
连接器
输出
生成特定机器的专用指令
最少优化, 保持全部兼容性 (-mtune)
使用处理器内建函数
优化级别 (-Ox)
使用下列位宽编译 (-mx)
语言标准 (-std)
ISO C90
ISO C99
ISO C++
ISO C++11
GNU C90
GNU C99
GNU C++
GNU C++11
获得更多关于GCC的选项, 请参阅
http://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html
确定[O]
取消[C]
帮助[H]

图 3.27

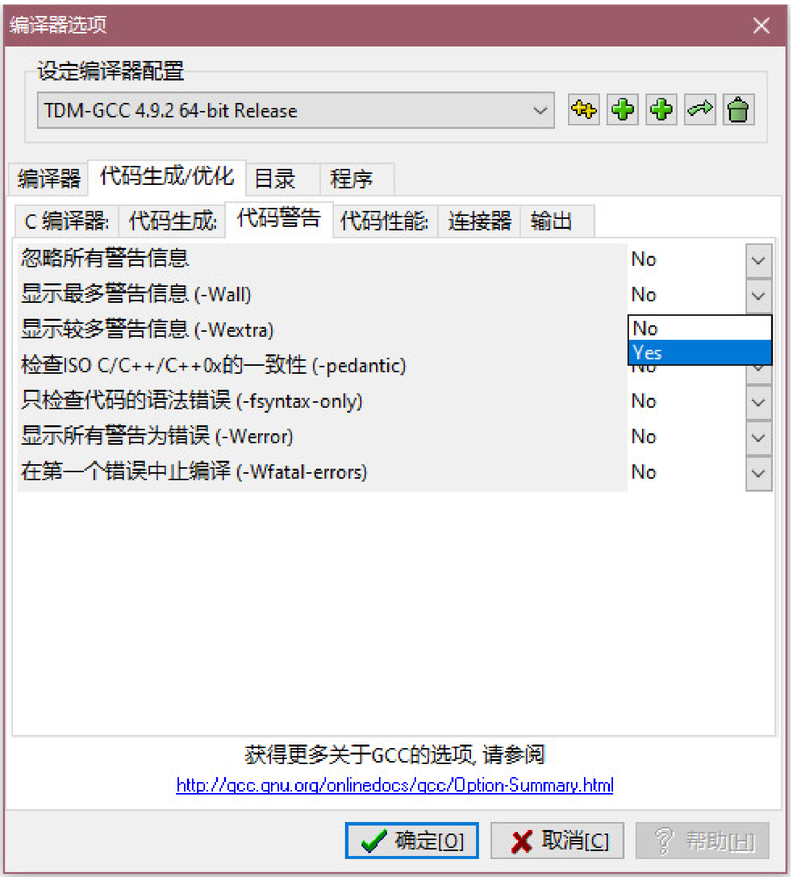


图 3.28

生成调试信息

当显示”项目没有调试信息，您想打开项目调试选项并重新生成吗？”点击后闪退或想使用调试功能时需开启此功能。

选择”连接器”子选项卡中的”产生调试信息”选项标签。

编译小 trick

点击工具 -> 编译选项，然后选择”编译器”选项卡，接下来介绍几个常用 trick。

开大栈

防止 DFS 爆系统栈之类的情况出现。

在”连接器命令行加入以下命令”中加入 `-Wl,--stack=128000000` 命令。

此命令将栈开到了约 128MB 的大小，有需要可以自行增加。

定义宏

方便本地评测使用文件输入输出或作其他用途。

在”连接器命令行加入以下命令”中加入 `-D[String]` 命令。

其中 `[String]` 改为你需要的宏名。

如图，当开启编译选项后便可将以下代码从 `test.in` 文件读入数据并在 `test.out` 文件中输出。

```
#ifdef LOCAL
freopen("test.in", "r", stdin);
freopen("test.out", "w", stdout);
#endif
```

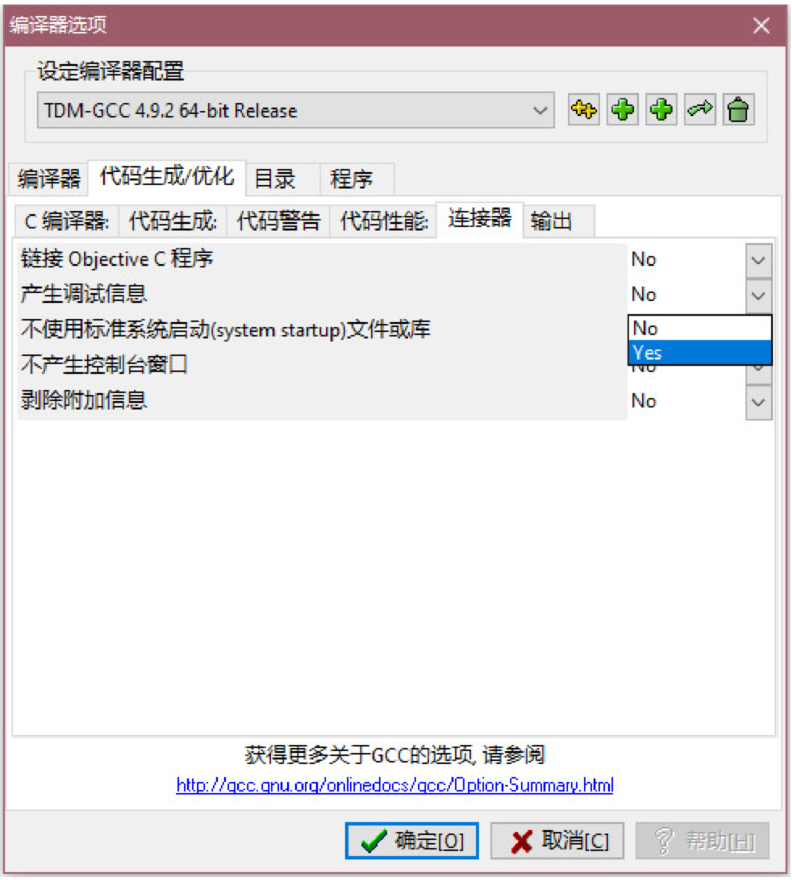

编译器选项
设定编译器配置
TDM-GCC 4.9.2 64-bit Release
编译器
代码生成/优化
目录
程序
C 编译器:
代码生成:
代码警告
代码性能:
连接器
输出
链接 Objective C 程序
No
产生调试信息
No
不使用标准系统启动(system startup)文件或库
No
Yes
不产生控制台窗口
剥除附加信息
No
获得更多关于GCC的选项, 请参阅
http://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html
确定[O]
取消[C]
帮助[H]

图 3.29

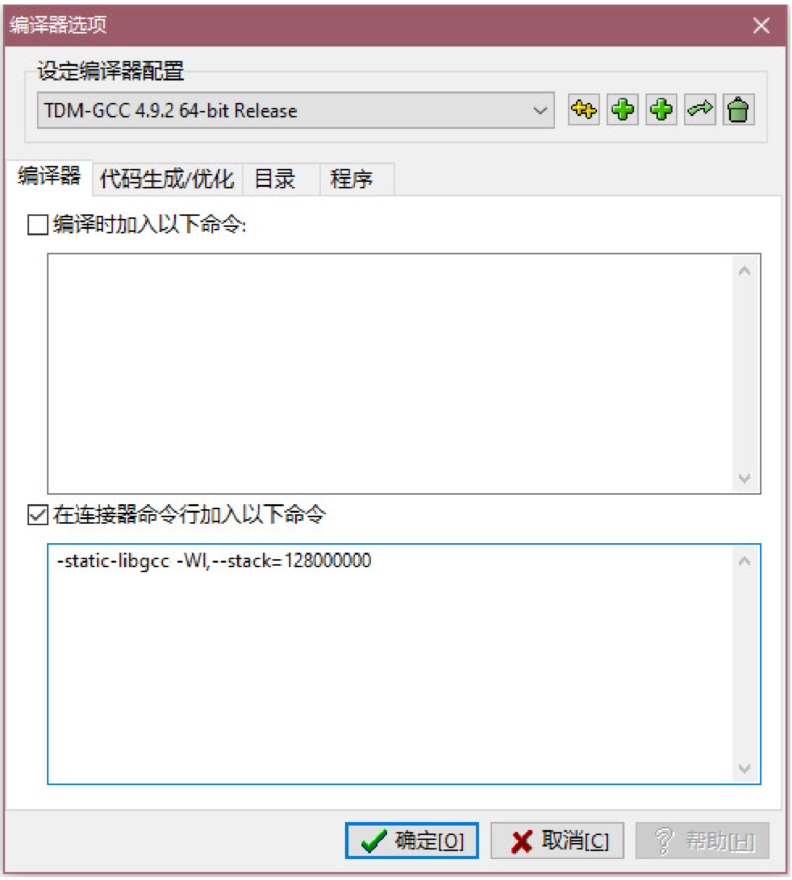

编译器选项
设定编译器配置
TDM-GCC 4.9.2 64-bit Release
编译器
代码生成/优化
目录
程序
编译时加入以下命令:
在连接器命令行加入以下命令
-static-libgcc -Wl,--stack=128000000
确定[O]
取消[C]
帮助[H]

图 3.30

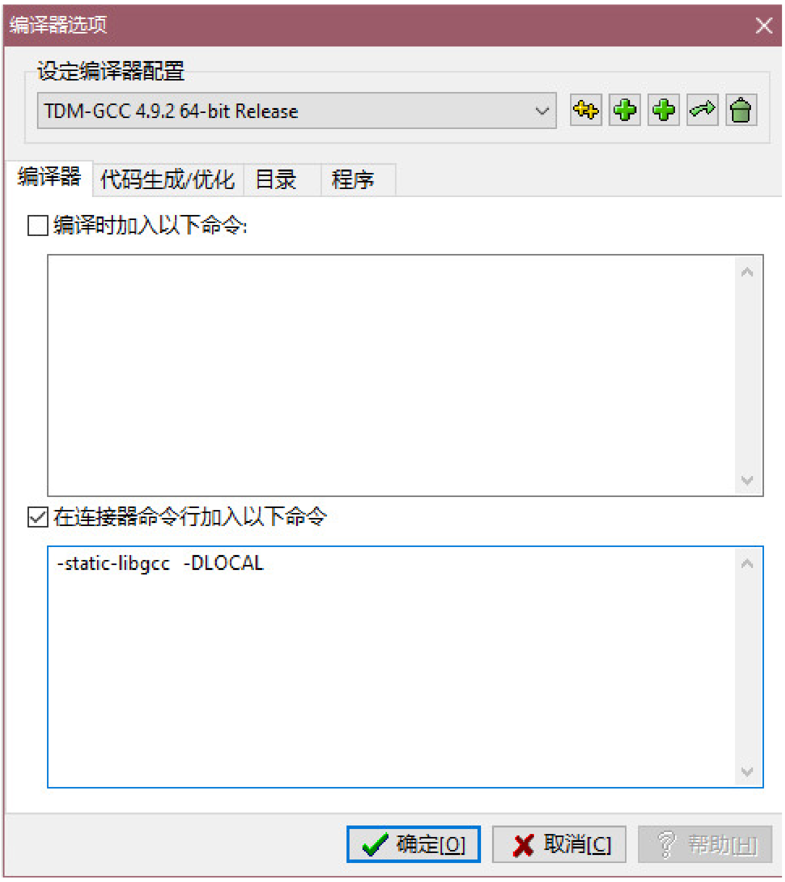


图 3.31

代码格式化

点击 Astyle-> 格式化当前文件或按 Ctrl+Shift+A 进行代码格式化。

图 3.32

美化

字体

点击工具 -> 编辑器选项，然后选择” 显示” 选项卡。

主题

点击工具 -> 编辑器选项，然后选择” 语法” 选项卡，可以使用预设主题，也可以自行调整。

参考资料

[1] 项目源代码托管于 GitHub 和 SourceForge.

[2] 源代码托管于 Github


编辑器属性
基本
显示
语法
代码
代码补全
自动保存
编辑器字体:
字体
Consolas
大小
10
装订线
可见
自动大小
使用自定义字体
行号
从0开始
显示前缀0
栏距
1
字体
Consolas
大小
10
也可以通过Control+滚动修改文本大小，就像在浏览器中。
确定[O]
取消[C]
帮助[H]

图 3.33

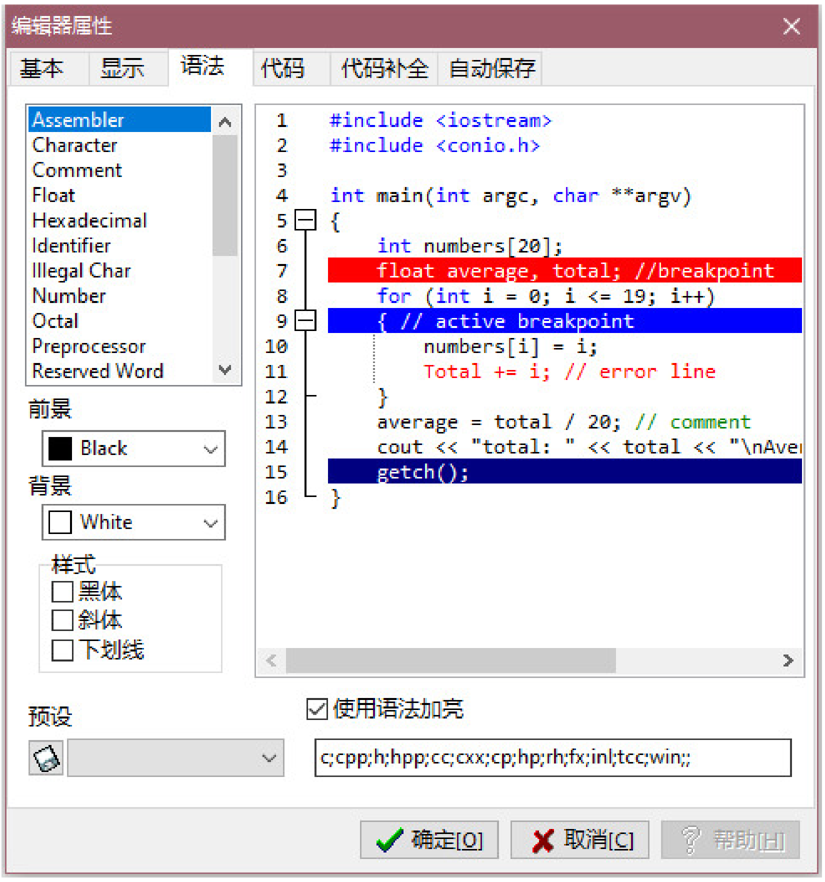

编辑器属性
基本
显示
语法
代码
代码补全
自动保存
Assembler
Character
Comment
Float
Hexadecimal
Identifier
Illegal Char
Number
Octal
Preprocessor
Reserved Word
#include <iostream>
#include <conio.h>
int main(int argc, char **argv)
{
int numbers[20];
float average, total; //breakpoint
for (int i = 0; i <= 19; i++)
{ // active breakpoint
numbers[i] = i;
Total += i; // error line
}
average = total / 20; // comment
cout << "total: " << total << "\nAve
getch();
}
前景
Black
背景
White
样式
黑体
斜体
下划线
预设
使用语法加亮
c;cpp;h;hpp;cc;cxx;cp;hp;rh;fx;inl;tcc;win;;
确定[O]
取消[C]
帮助[H]

图 3.34

[3] 项目官网位于 小熊猫 C++，源代码托管于 Github

[4] royqh1979

3.2.9 CLion

简介

CLion 是一款由 JetBrains 公司开发的功能丰富且强大的跨平台 C/C++ 集成开发环境（IDE）。

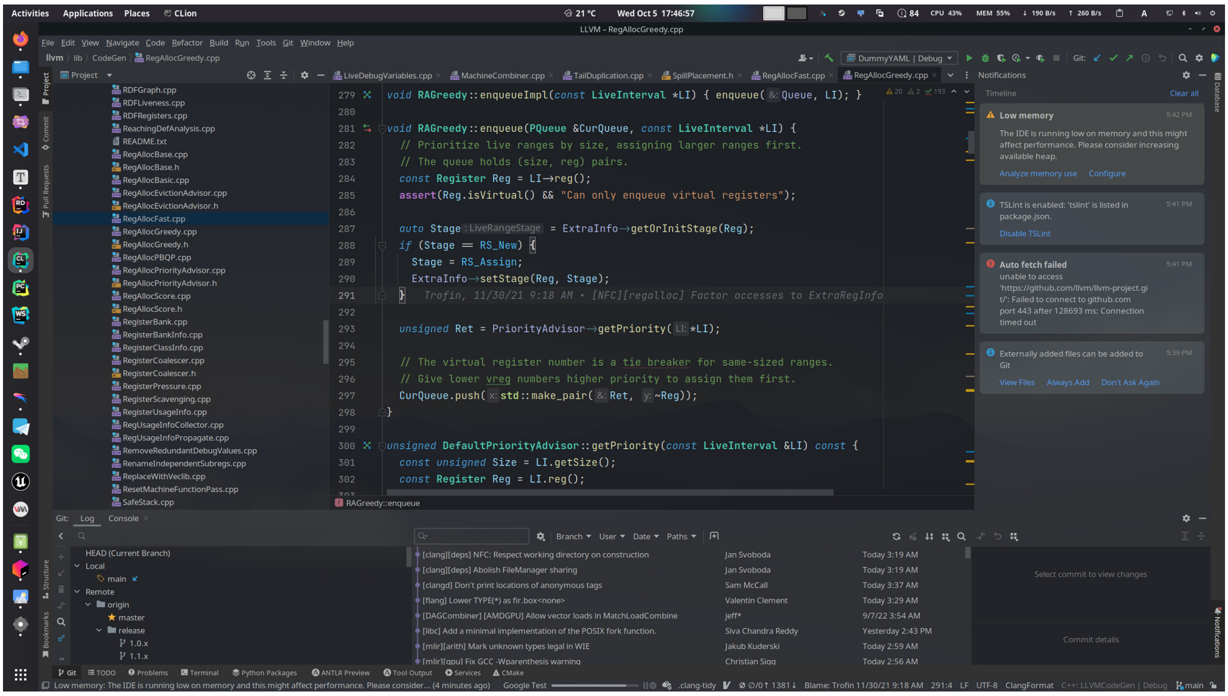


图 3.35 Clion

官方教程

在官方网站中给出了 学习 Clion[1] 的教程。

安装

参见 Download CLion[2]。

配置

工具链安装

CLion 默认不带编译器，构建工具和调试工具，需要手动进行安装。

Windows

参见 Tutorial: Configure CLion on Windows | CLion Documentation[3]

值得一提的是 CLion 的 Windows 版本中自带了 MinGW，所以可以不用额外安装 MinGW 工具链。

Linux

Debian/Ubuntu 及其衍生发行版

```
sudo apt install make cmake # build tools
sudo apt install gcc g++ gdb # compiler and debugger
sudo apt install clang clang++ llvm lldb # you can also choose to use clang tool
chain
```

Arch Linux 及其衍生发行版

```
sudo pacman -S make cmake # build tools
sudo pacman -S gcc g++ gdb # compiler and debugger
sudo pacman -S clang clang++ llvm lldb # you can also choose to use clang toolch
ain
```

Fedora/RHEL/CentOS/Rocky Linux

```
sudo dnf install make cmake # build tools
sudo dnf install gcc g++ gdb # compiler and debugger
sudo dnf install clang clang++ llvm lldb # you can also choose to use clang tool
chain
```

MacOS

参见 Tutorial: Configure CLion on macOS | CLion Documentation[4]

工具链设置

手动设置工具链

新安装的 CLion 会自动检测系统中的 C/C++ 开发工具链，如果已安装的工具链无法自动检测到，可在 `Settings` 中找到 `Build, Execution, Deployment>Toolchains` 进行手动配置。

CMake

设置

CLion 默认使用 CMake[5] 作为构建工具，关于 CMake 的设置可以在 `Build, Execution, Deployment>Toolchains>CMake` 中修改。

编译选项

CMake 默认使用项目根目录下的 `CMakeList.txt` 作为构建项目的配置文件，可以使用 `add_compile_options` 命令来增加编译选项，例如：

```
add_compile_options(-std=c++17 -DDEBUG)
```

其他 CMake 的功能请参考 CMake 官方文档[6]

免费获取 CLion IDE 许可证

CLion 为付费产品，但是可以通过教育邮箱或开源项目申请特殊许可证。申请之后不仅可以免费使用正版 CLion IDE, 还可以免费使用 JetBrains 公司开发的其他付费产品。

使用教育邮箱获取

进入官网的 Free Educational Licenses 页面[7]，点击 `Apply` 按钮，填写相关信息即可申请。

注意：在注册时于邮箱选项请填如 @edu.cn 后缀的教育邮箱，特殊许可证需要邮箱验证后方可拿到。

你可以到所在高校的教务中心官网去申请教育邮箱，如果申请不到需要使用 学信网[8] 进行认证（仅中国大陆）。

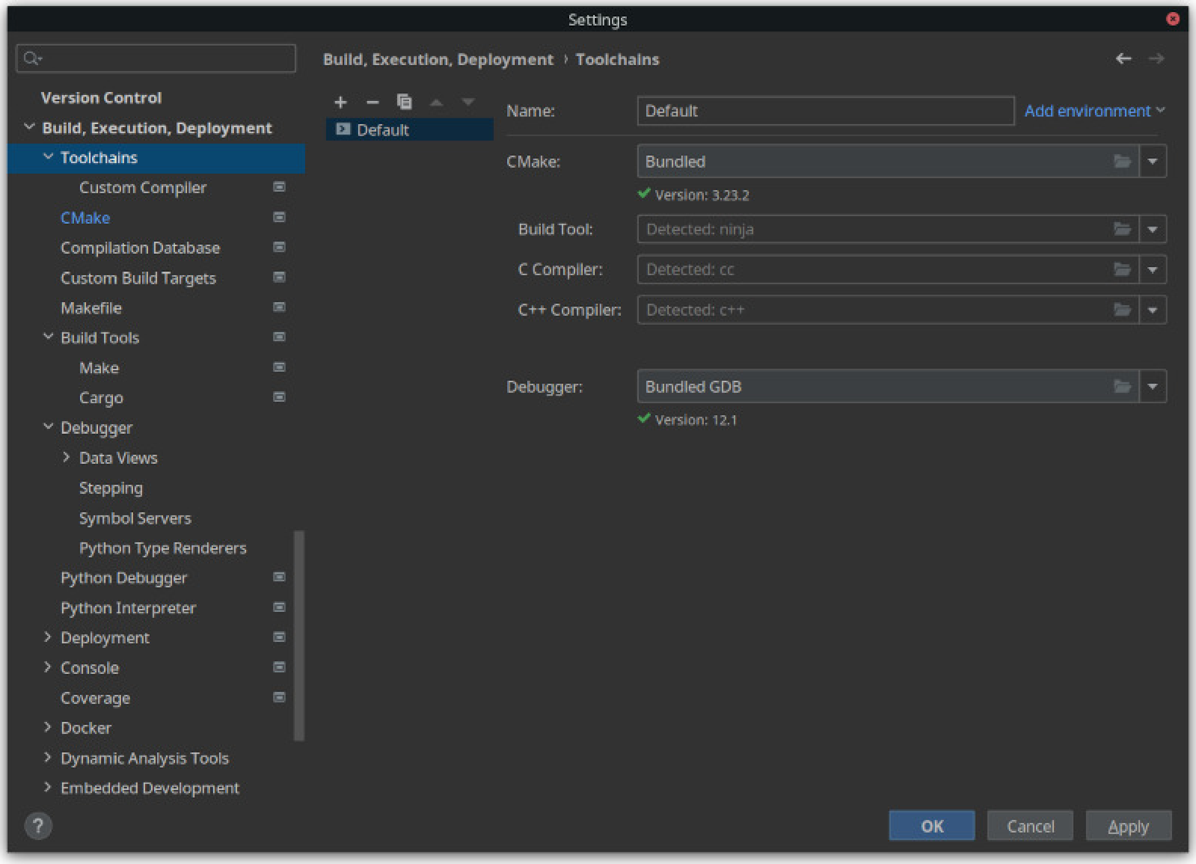


图 3.36 Config Toolchains

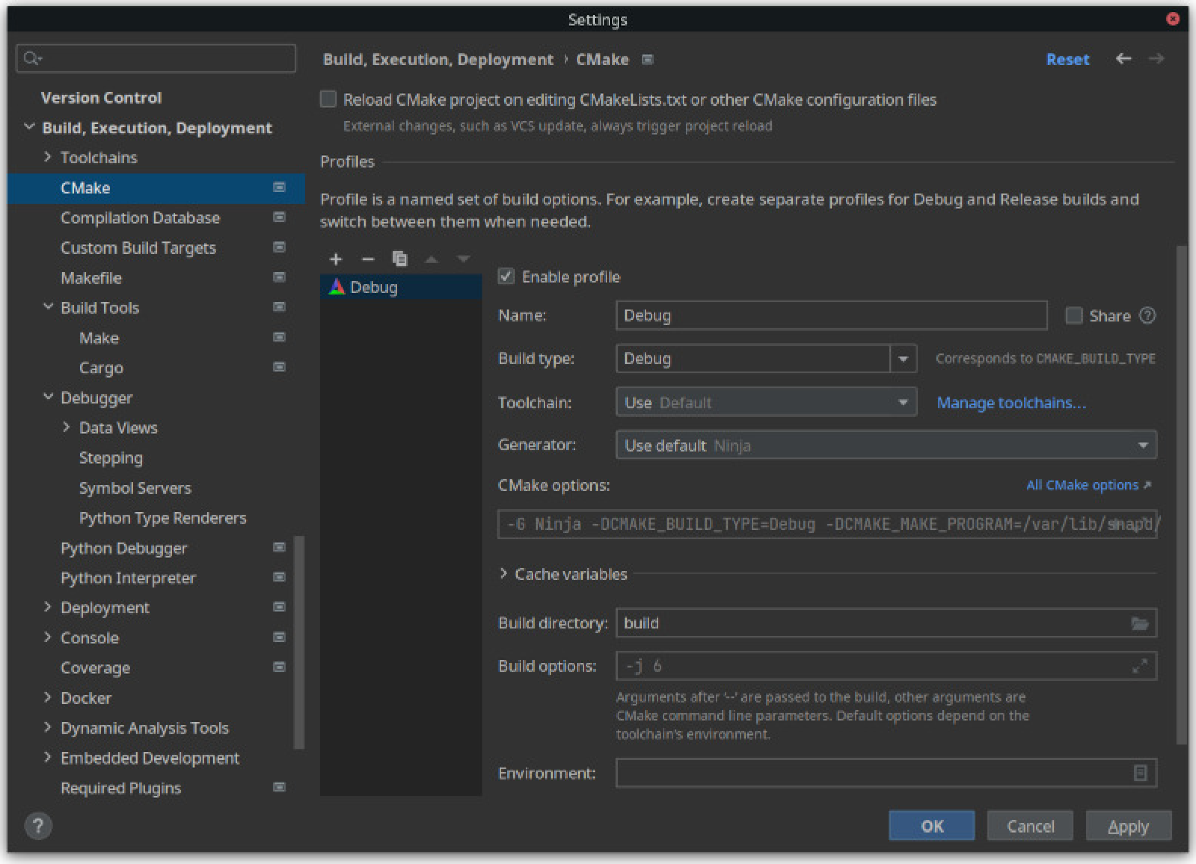


图 3.37 CMake Settings

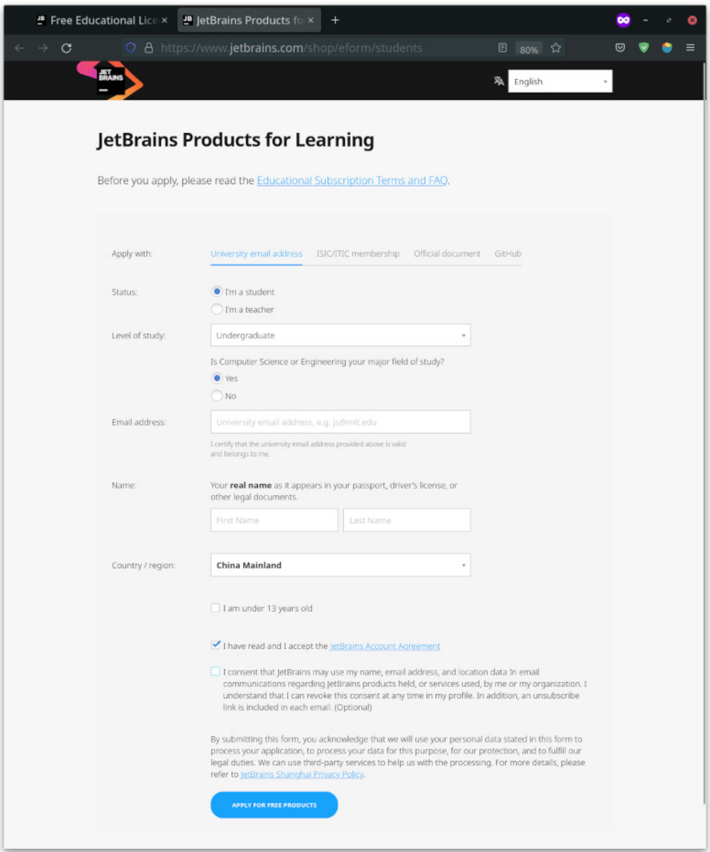


图 3.38 Educational Licenses

使用开源项目获取

如果您是某个开源项目的核心开发者或维护者之一，您可以尝试申请开源开发许可证 (Open Source Development License). 申请流程与教育许可证类似，但需要填写开源项目的仓库地址。

参考资料与注释

[1] 学习 Clion

[2] Download CLion

[3] Tutorial: Configure CLion on Windows | CLion Documentation

[4] Tutorial: Configure CLion on macOS | CLion Documentation

[5] CMake

[6] CMake 官方文档

[7] Free Educational Licenses 页面

[8] 学信网

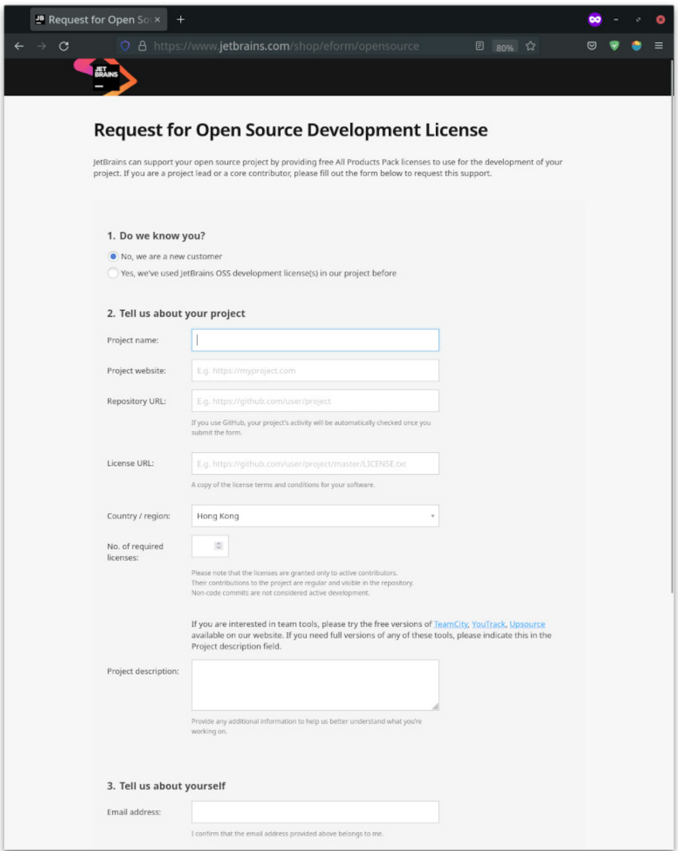


图 3.39 Open Source Development License

3.2.10 Geany

Authors: xingjiapeng, MingqiHuang

Geany 是一个轻量、便捷的编辑器，对于 Linux 环境下的初学者较为友好。

与 Dev-C++ 一样，它可以编译运行单个文件。

不过，它可以在 Linux/Windows/macOS 下运行。

其官网为：https://geany.org/[4]

优缺点

优点

1. 轻量；
2. 可以编译运行单个文件；
3. 不需要太多配置；
4. 跨平台。

缺点

1. 没有太多人使用；
2. 在 macOS Catalina 下有一些权限问题[1-1]；
3. 新建文件时，默认不会有语法高亮，需要手动切换文件类型。

安装

参见 Download | Geany[5]

使用技巧

切换文件类型

在文档 -> 设置文件类型中进行切换。

如 C++ 语言，点击文档 -> 设置文件类型 -> 编程语言 -> *C++* 源文件，即可看到文件已被转换为 C++ 语言的语法高亮了。

设置文件模板

在配置文件目录下建立 templates/files 文件夹，建立在其中的文件即为模板文件，再次打开 Geany，就可以在文件 -> 从模板新建中找到它了。

配置文件目录可以通过帮助 -> 调试信息的第二、三行找出。

这里给出 macOS 和 Linux 下的默认模板配置文件目录：

- 系统目录：`/usr/share/geany/templates/files/`
- 用户目录：`~/.config/geany/templates/files/`[2-1]

常见问题

兼容深度终端

在首选项 -> 工具 -> 虚拟终端，修改终端的命令为：

```
deepin-terminal -x "/bin/sh" %c
```

点击「应用」按钮即可。[3-1]

参考资料与注释

[1] 详见：https://github.com/geany/geany/issues/2344 [1-1] [1-2]

[2] 来源：https://wiki.geany.org/config/templates [2-1] [2-2]

[3] 来源：Deepin Wiki https://wiki.deepin.org/ [3-1] [3-2]

[4] https://geany.org/

[5] Download | Geany

3.2.11 Xcode

Authors: shenyouran, Xeonacid, StudyingFather, CoelacanthusHex

简介

Xcode 是一个运行在 macOS 上的集成开发工具（IDE），由 Apple Inc. 开发。

安装

方法一

打开苹果电脑自带的 App Store（或者尝试 快捷链接[1]）下载 Xcode。点击获取，然后输入苹果账号密码开始下载安装。

图 3.40

方法二

访问 苹果开发者下载页面[2]，用苹果账号登录，然后找到 Xcode 最新的稳定版本安装包（即不含 Beta 的最新版本，此处为 11.6）：

点击弹出框内蓝色的文件名即可下载。得到压缩包之后，用系统自带的工具进行解压，然后得到文件 Xcode.app。把这个文件移动到【应用程序】文件夹后即可使用。

基础配置

首次打开 Xcode 时，可能会遇到下列弹出窗口：

这个窗口是 Xcode 元件的安装引导。点击 `Install` 并输入当前用户密码即可。

安装完毕后，界面左侧显示：

点击 `Create a new Xcode project`（创建一个新的 Xcode 项目），然后选择上方 `macOS` 中的 `Command Line Tool`（命令行工具），并点击右下角的 `Next`。

接下来，我们可以给项目命名，但最重要的是选择项目的语言。我们可以根据自己的需求，在最下方 `Language` 处选择 C 或者 C++：

项目的目录可以根据需要选择。创建完毕后，Xcode 会自动打开这个项目，并自动创建一个 `main` 文件（C 语言的后缀为 `.c`，C++ 语言的后缀为 `.cpp`）。

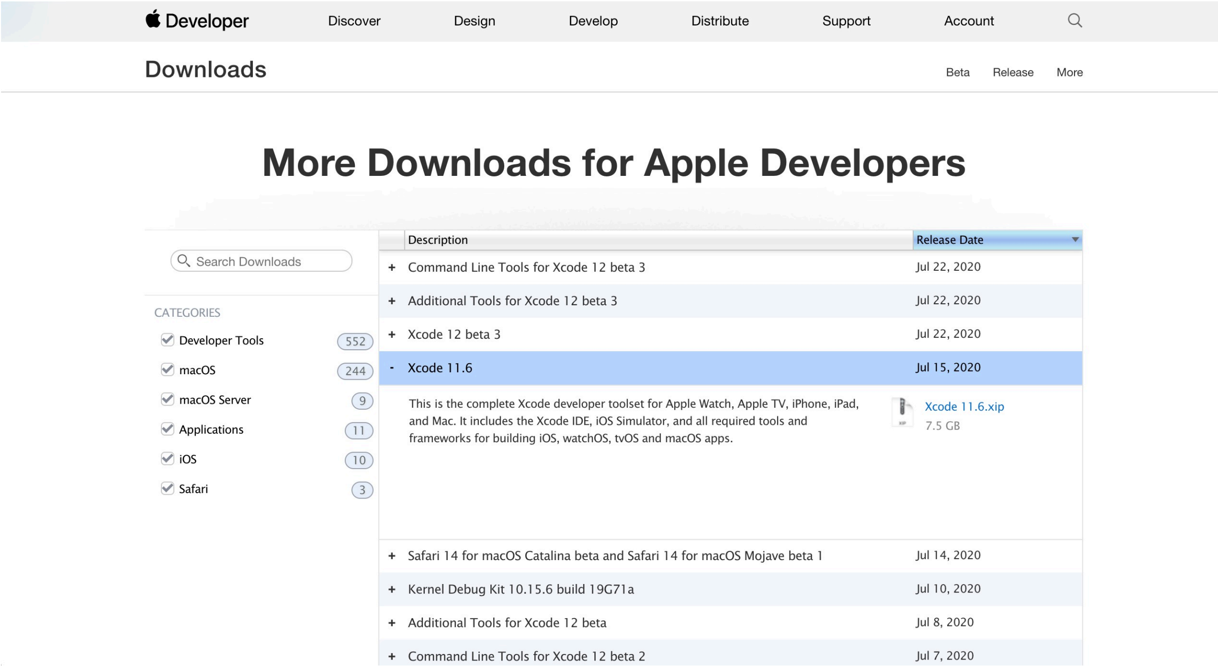

Developer
Discover
Design
Develop
Distribute
Support
Account
Downloads
Beta
Release
More
More Downloads for Apple Developers
Search Downloads
CATEGORIES
Developer Tools 552
macOS 244
macOS Server 9
Applications 11
iOS 10
Safari 3
Description
Release Date
+ Command Line Tools for Xcode 12 beta 3 Jul 22, 2020
+ Additional Tools for Xcode 12 beta 3 Jul 22, 2020
+ Xcode 12 beta 3 Jul 22, 2020
- Xcode 11.6 Jul 15, 2020
This is the complete Xcode developer toolset for Apple Watch, Apple TV, iPhone, iPad, and Mac. It includes the Xcode IDE, iOS Simulator, and all required tools and frameworks for building iOS, watchOS, tvOS and macOS apps.
Xcode 11.6.xip
7.5 GB
+ Safari 14 for macOS Catalina beta and Safari 14 for macOS Mojave beta 1 Jul 14, 2020
+ Kernel Debug Kit 10.15.6 build 19G71a Jul 10, 2020
+ Additional Tools for Xcode 12 beta Jul 8, 2020
+ Command Line Tools for Xcode 12 beta 2 Jul 7, 2020

图 3.41

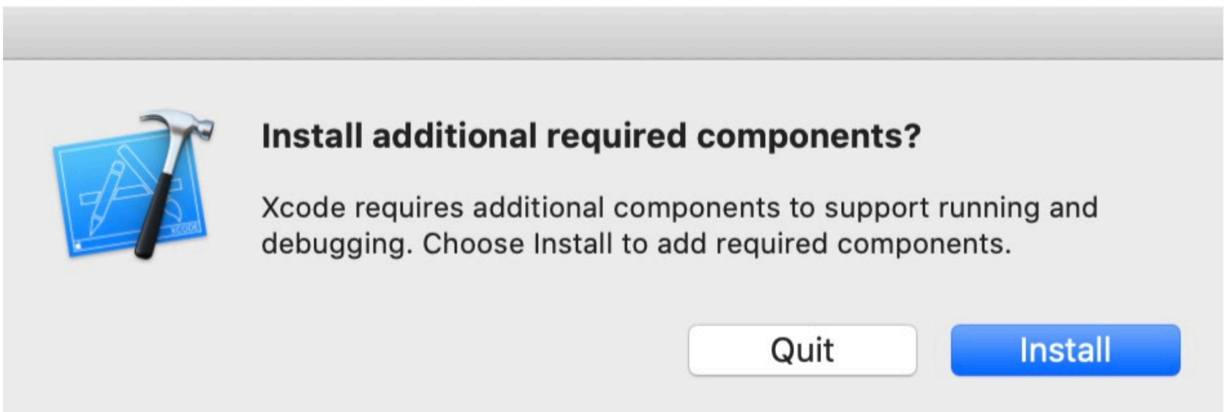

Install additional required components?
Xcode requires additional components to support running and debugging. Choose Install to add required components.
Quit
Install

图 3.42


XCODE
Welcome to Xcode
Version 11.6 (11E708)
Get started with a playground
Explore new ideas quickly and easily.
Create a new Xcode project
Create an app for iPhone, iPad, Mac, Apple Watch, or Apple TV.
Clone an existing project
Start working on something from a Git repository.

图 3.43

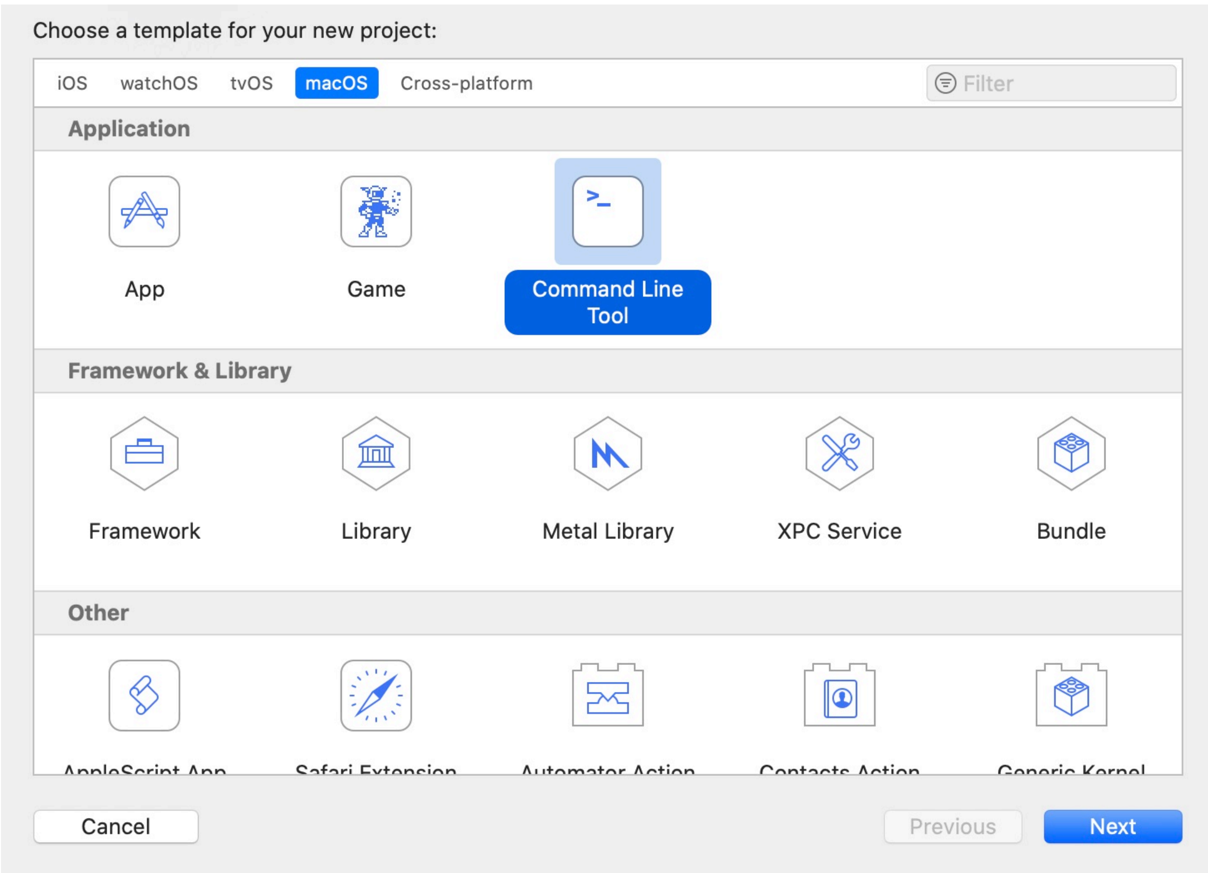

Choose a template for your new project:
iOS
watchOS
tvOS
macOS
Cross-platform
Filter
Application
App
Game
Command Line Tool
Framework & Library
Framework
Library
Metal Library
XPC Service
Bundle
Other
Cancel
Previous
Next

图 3.44

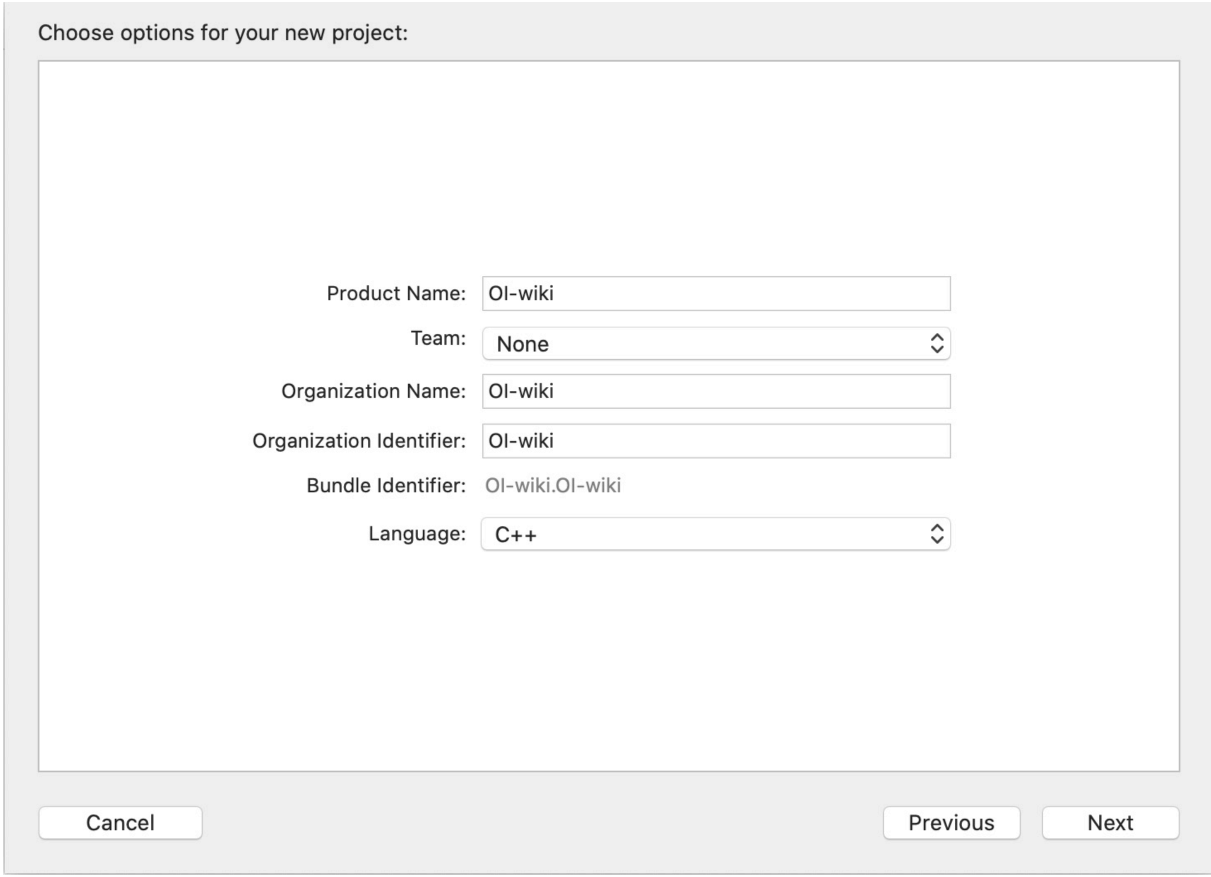


图 3.45

点击这个文件，就可以打开编辑区域：

编写代码后，可以按 B 编译（Build）， R 运行（Run）。运行后拖动，得到三个部分：

一般来说我们只使用【编辑区】和【运行区】。若程序有输入，那么在【运行区】中进行输入之后，就可以得到输出。界面呈现效果：

仿照这种方式，我们就可以运行任何的单个 C/C++ 程序。

万能头文件的使用

在编写代码过程中，我们可能会使用到很多头文件。常用的解决方法是使用万能头文件。

我们在源代码第一行引入万能头文件，然而编译过程中却提示：`'bits/stdc++.h' file not found`。即该头文件未找到。

这是因为在 macOS 上默认使用 libc++[3] 作为 C++ 标准库实现，而万能头 `bits/stdc++.h` 是 GNU libstdc++[4] 所独有的。

不过，我们可以手动编写一个万能头文件来使用。

步骤 1

打开终端（Terminal.app），前往 Xcode 存储头文件的文件夹，即：

```
cd /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/include/c++/v1
```

如果 Xcode 版本大于等于 12.5，那么

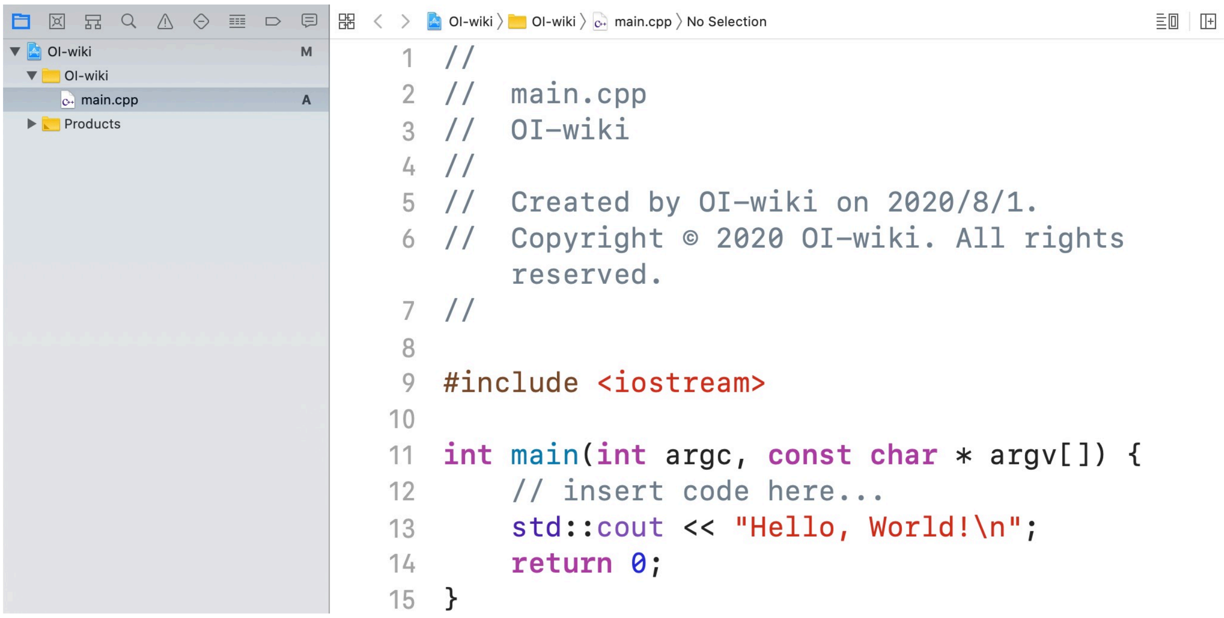

OI-wiki
OI-wiki
main.cpp
Products
OI-wiki 〉 OI-wiki 〉 main.cpp 〉 No Selection
//
// main.cpp
// OI-wiki
//
// Created by OI-wiki on 2020/8/1.
// Copyright © 2020 OI-wiki. All rights reserved.
//
#include <iostream>
int main(int argc, const char * argv[]) {
// insert code here...
std::cout << "Hello, World!\n";
return 0;
}

图 3.46

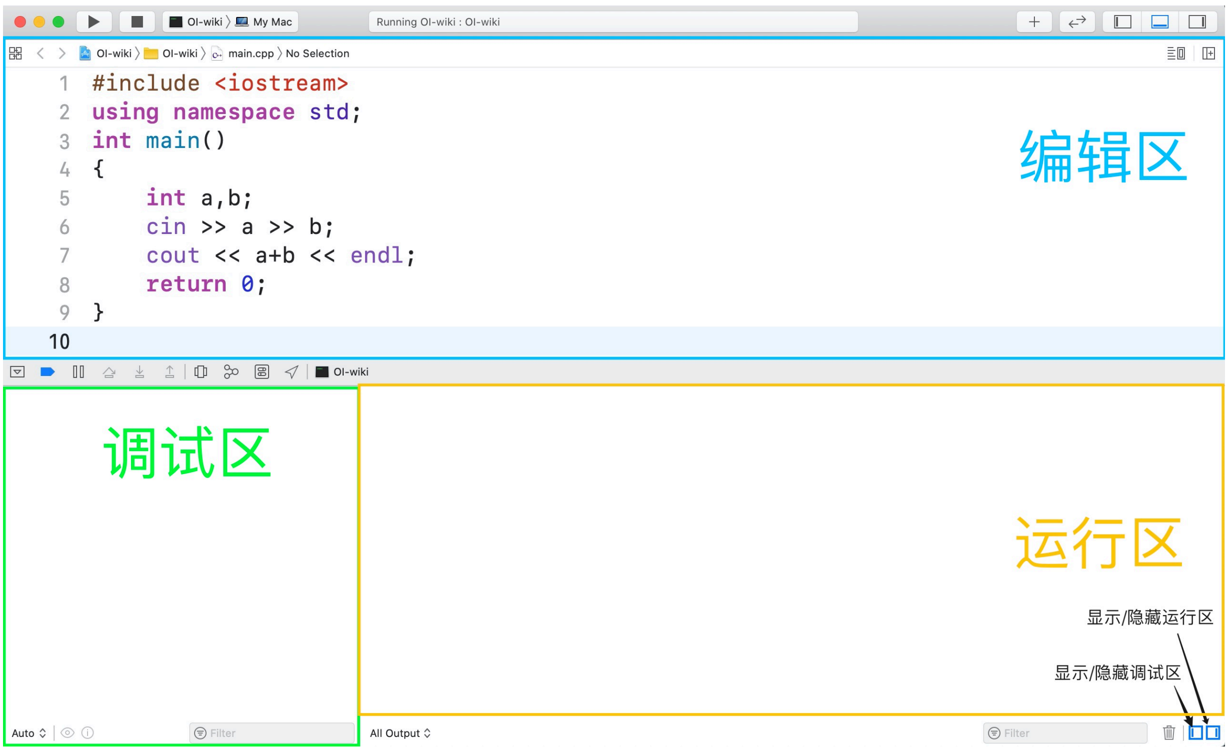

OI-wiki 〉 My Mac
Running OI-wiki : OI-wiki
OI-wiki 〉 OI-wiki 〉 main.cpp 〉 No Selection
#include <iostream>
using namespace std;
int main()
{
int a,b;
cin >> a >> b;
cout << a+b << endl;
return 0;
}
编辑区
调试区
运行区
显示/隐藏运行区
显示/隐藏调试区
Auto
Filter
All Output
Filter

图 3.47

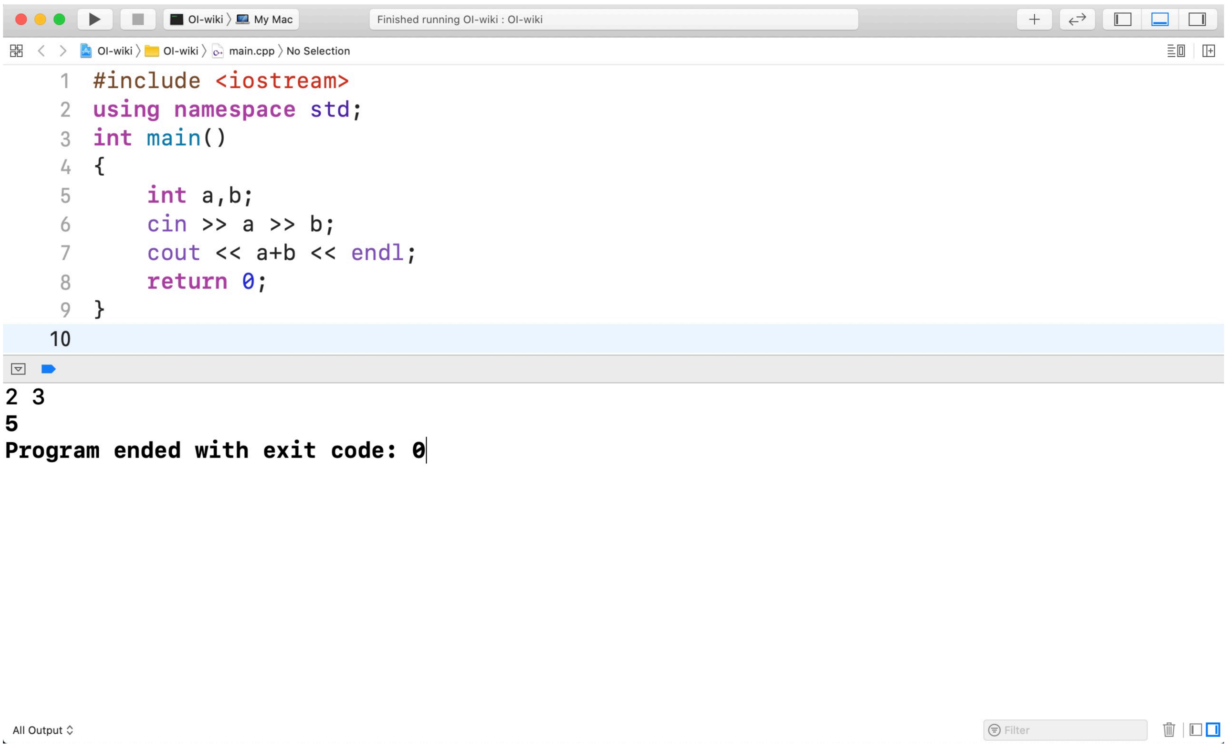


图 3.48

图 3.49

```
cd /Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Develope
r/SDKs/MacOSX.sdk/usr/include/c++/v1/
```

步骤 2

创建 `bits` 文件夹并进入：

```
mkdir bits
cd bits
```

用 vim 创建 stdc++.h 文件：

```
vim stdc++.h
```

界面如下：

接着，我们需要通过 vim 编辑文件。敲击 i（insert）键盘即可进入插入/编辑模式（下方出现 `-- INSERT --`）：

将下面这段代码块复制并粘贴到终端中：

" 万能头文件代码块"

```
// C++ includes used for precompiling -*- C++ -*-

// Copyright (C) 2003-2020 Free Software Foundation, Inc.
//
// This file is part of the GNU ISO C++ Library.  This library is free
// software; you can redistribute it and/or modify it under the
// terms of the GNU General Public License as published by the
```


bits — vim stdc++.h — 80×24
"stdc++.h" [New File]

图 3.50


bits — vim stdc++.h — 80×24
-- INSERT --

图 3.51

```
// Free Software Foundation; either version 3, or (at your option)
// any later version.

// This library is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.

// Under Section 7 of GPL version 3, you are granted additional
// permissions described in the GCC Runtime Library Exception, version
// 3.1, as published by the Free Software Foundation.

// You should have received a copy of the GNU General Public License and
// a copy of the GCC Runtime Library Exception along with this program;
// see the files COPYING3 and COPYING.RUNTIME respectively.  If not, see
// <http://www.gnu.org/licenses/>.

/** @file stdc++.h
 *  This is an implementation file for a precompiled header.
 */

// 17.4.1.2 Headers

// C
#ifndef _GLIBCXX_NO_ASSERT
#include <cassert>
#endif
#include <cctype>
#include <cerrno>
#include <cfloat>
#include <ciso646>
#include <climits>
#include <clocale>
#include <cmath>
#include <csetjmp>
#include <csignal>
#include <cstdarg>
#include <cstddef>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <ctime>
#include <cwchar>
#include <cwctype>

#if __cplusplus >= 201103L
#include <ccomplex>
#include <cfenv>
#include <cinttypes>
#include <cstdbool>
#include <cstdint>
#include <ctgmath>
/* https://stackoverflow.com/a/25892335/15125422 */
#if defined(__GLIBCXX__) || defined(__GLIBCPP__)
```

```
#include <cstdalign>
#include <cuchar>
#endif
#endif

// C++
#include <algorithm>
#include <bitset>
#include <complex>
#include <deque>
#include <exception>
#include <fstream>
#include <functional>
#include <iomanip>
#include <ios>
#include <iosfwd>
#include <iostream>
#include <istream>
#include <iterator>
#include <limits>
#include <list>
#include <locale>
#include <map>
#include <memory>
#include <new>
#include <numeric>
#include <ostream>
#include <queue>
#include <set>
#include <sstream>
#include <stack>
#include <stdexcept>
#include <streambuf>
#include <string>
#include <typeinfo>
#include <utility>
#include <valarray>
#include <vector>

#if __cplusplus >= 201103L
#include <array>
#include <atomic>
#include <chrono>
#include <codecvt>
#include <condition_variable>
#include <forward_list>
#include <future>
#include <initializer_list>
#include <mutex>
#include <random>
#include <ratio>
#include <regex>
#include <scoped_allocator>
#include <system_error>
```

```
#include <thread>
#include <tuple>
#include <type_traits>
#include <typeindex>
#include <unordered_map>
#include <unordered_set>
#endif

#if __cplusplus >= 201402L
#include <shared_mutex>
#endif

#if __cplusplus >= 201703L
#include <any>
#include <charconv>
// #include <execution>
#include <filesystem>
#include <memory_resource>
#include <optional>
#include <string_view>
#include <variant>
#endif

#if __cplusplus > 201703L
#include <bit>
#include <compare>
#include <concepts>
#include <numbers>
#include <ranges>
#include <span>
#include <stop_token>
// #include <syncstream>
#include <version>
#endif
```

该文件来源于 10.2.0 版本的 libstdc++[5] 并经少许修改以兼容 libc++。

按键盘左上角的 Esc 退出编辑模式，然后直接输入 `:wq` 并换行即可保存文件。

步骤 3

关闭终端，回到 Xcode。重新按下 B/ R 进行编译，发现编译成功：

优缺点

优点：由苹果开发，适合 Mac 用户，界面齐全、美观。

缺点：Xcode 主要用来苹果程序的开发，对于竞赛来说功能冗余，安装包大小较大，而且仅能在 Mac 端上使用。

参考资料与注释

[1] 快捷链接

[2] 苹果开发者下载页面

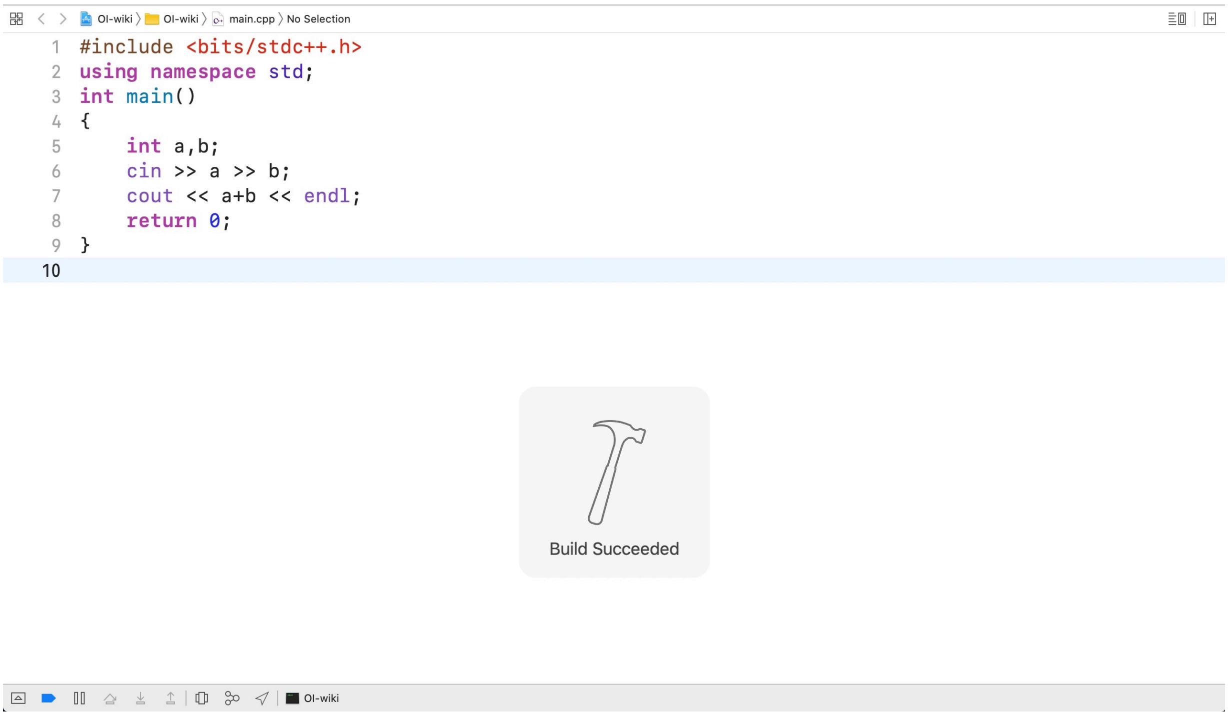


图 3.52

[3] libc++

[4] GNU libstdc++

[5] 10.2.0 版本的 libstdc++

3.2.12 GUIDE

GUIDE（GAIT Universal IDE）是由北航 GAIT 研究组开发的、专门为 NOI 选手设计的、支持 C/C++/Pascal 三种程序设计语言的小型集成开发环境。

note

自 2021 年 9 月 1 日起启用的 NOI Linux 2.0 不再包含 GUIDE。[1]

安装

Windows

参见 https://www.noi.cn/xw/2009-03-23/714714.shtml[2]。

Linux

参见 https://www.noi.cn/xw/2009-03-23/714714.shtml[2] 或按照如下步骤安装。

需要的动态库文件及包名

动态库	Arch 包名	Debian 包名	Fedora 包名	openSUSE x86 包名	openSUSE x86_64 包名
libpng12.so.0	lib32-libpng12	libpng12	libpng12	libpng12-0	libpng12-0-32bit
libSM.so.6	lib32-libsm	libsm6	libSM	libSM6	libSM6-32bit
libICE.so.6	lib32-libice	libice6	libICE	libICE6	libICE6-32bit
libXi.so.6	lib32-libxi	libxi6	libXi	libXi6	libXi6-32bit
libXrender.so.1	lib32-libxrender	libxrender1	libXrender	libXrender1	libXrender1-32bit
libXrandr.so.2	lib32-libxrandr	libxrandr	libXrandr	libXrandr2	libXrandr2-32bit
libfreetype.so.6	lib32-freetype2	libfreetype6	freetype	libfreetype6	libfreetype6-32bit
libfontconfig.so.1	lib32-fontconfig	libfontconfig1	fontconfig	libfontconfig1	libfontconfig1-32bit
libXext.so.6	lib32-libxext	libxext6	libXext	libXext6	libXext6-32bit
libX11.so.6	lib32-libx11	libx11-6	libX11	libX11-6	libX11-6-32bit
libz.so.1	lib32-zlib	zlib1g	zlib	libz1	libz1-32bit
libgthread-2.0.so.0	lib32-glib2	libglib2.0-0	glib2	libgthread-2_0-0	libgthread-2_0-0-32bit
libglib-2.0.so.0	lib32-glib2	libglib2.0-0	glib2	libglib2_0-0	libglib2_0-0-32bit
libstdc++.so.6	lib32-gcc-libs	libstdc++6	libstdc++	libstdc++6	libstdc++6-32bit
libgcc_s.so.1	lib32-gcc-libs	lib32gcc1	libgcc	libgcc_s1	libgcc_s1
librt.so.1	lib32-glibc	libc6	glibc	glibc	glibc-32bit
libpthread.so.0	lib32-glibc	libc6	glibc	glibc	glibc-32bit
libdl.so.2	lib32-glibc	libc6	glibc	glibc	glibc-32bit
libm.so.6	lib32-glibc	libc6	glibc	glibc	glibc-32bit
libc.so.6	lib32-glibc	libc6	glibc	glibc	glibc-32bit

在 Debian 或 Ubuntu 安装

```
sudo apt install -y libpng12 libsm6 libice6 libxi6 libxrender1 libxrandr libfree
type6 libfontconfig1 libxext6 libx11-6 zlib1g libglib2.0-0 libglib2.0-0 libstdc+
+6 lib32gcc1 libc6
wget -c http://download.noi.cn/T/noi/GUIDE-1.0.2-ubuntu.tar
tar -xvf GUIDE-1.0.2-ubuntu.tar
cd GUIDE-1.0.2-ubuntu
echo "install:\n\tinstall -Dm755 -t /usr/bin GUIDE\n\tinstall -Dm644 -t /usr/sha
re/ lang_en.qm\n\tmkdir -p /usr/share/apis/ && cp -r apis/* /usr/share/apis/\n\t
mkdir -p /usr/share/doc/GUIDE/ && mkdir -p /usr/share/doc/GUIDE/html/ && cp -r d
oc/*  /usr/share/doc/GUIDE/html/" > Makefile
sudo apt install -y checkinstall
sudo checkinstall --pkgname "GUIDE" --pkgversion "1.0.2" -y
```

在 openSUSE 安装

按照 openSUSE/opi[3] 给出的方式安装 opi。

然后：（32 位用户自行删去 -32bit）

```
sudo opi checkinstall
sudo zypper install -n {libpng12-0,libSM6,libICE6,libXi6,libXrender1,libXrandr2,
libfreetype6,libfontconfig1,libXext6,libX11-6,libz1,libgthread-2_0-0,libglib2_0-
0,libstdc++6,libgcc_s1,glibc}-32bit
wget -c http://download.noi.cn/T/noi/GUIDE-1.0.2-ubuntu.tar
tar -xvf GUIDE-1.0.2-ubuntu.tar
cd GUIDE-1.0.2-ubuntu
echo "install:\n\tinstall -Dm755 -t /usr/bin GUIDE\n\tinstall -Dm644 -t /usr/sha
re/ lang_en.qm\n\tmkdir -p /usr/share/apis/ && cp -r apis/* /usr/share/apis/\n\t
mkdir -p /usr/share/doc/GUIDE/ && mkdir -p /usr/share/doc/GUIDE/html/ && cp -r d
oc/*  /usr/share/doc/GUIDE/html/" > Makefile
sudo checkinstall --pkgname "GUIDE" --pkgversion "1.0.2" -y -rpmi
```

编辑文件

点击页面上方工具栏的「新文件」按钮（或者使用 Ctrl+N 快捷键）来创建一个新文件。

在默认情况下，GUIDE 的代码字体并非等宽字体，看上去非常不美观，因此需要在设置中更改字体。

在编辑 -> 选项 -> 语法高亮设置中，点击「全部字体」按钮，即可切换编辑器字体。

需要注意的是，对于未保存的新文件，字体仍然是默认字体。因此建议在开始编辑前先保存文件（点击工具栏的「保存」按钮，或按下 Ctrl+S 快捷键），再进行编辑。

编译与运行

在编辑完源代码后，点击工具栏的「编译」按钮（或 F7 快捷键）进行编译。

> " 更改编译选项"
>
> GUIDE 没有设置默认编译选项的功能，用户只能更改对某个文件的编译选项。
>
> 右键点击想要更改编译选项的文件的标签，选择**设置编译命令**选项，即可更改该文件的编译选项。

如果源代码正常编译，点击工具栏的「运行」按钮（或 Ctrl+F5 快捷键）即可运行程序。

调试

GUIDE 自带的调试功能存在很多 bug（如程序中途发生崩溃等），因此不推荐直接使用 GUIDE 的调试功能。

建议直接在 终端 下使用 gdb 来进行调试。

[1] NOI Linux 2.0 发布， 将于 9 月 1 日起正式启用！

[2] https://www.noi.cn/xw/2009-03-23/714714.shtml [2-1] [2-2]

[3] openSUSE/opi

3.2.13 Sublime Text

简介

Sublime Text（以下简称 ST，后附数字作为版本区分，如无则各版本都适用）是一款轻量级的文本编辑器，支持多种语言的语法高亮及代码补全。具有高度的可拓展性以及 Vim 模式，特别的热启动模式大幅减小了文件丢失的可

能。

新版 NOI Linux 中支持版本为 ST3 最后一个版本 3.2.2[1]，故这里以 ST3 为主。目前 ST4 正式版已经发布[2]，现在如仍使用 ST3 会提示更新。

ST4 与 ST3 的重要差别会有额外补充，在介绍中如果提及某项在 ST3 已有翻译则使用中文，否则为 ST4 中的英文。

安装

ST4 的安装方法参见 Sublime Text 4 的下载页面[5]。

ST3 的安装方法参见 Sublime Text 3 的下载页面[6]。

> **" 提示购买"**
>
> ST 是收费软件，但有一个无限期的试用，试用并不会带来功能上的缺失，但会不时弹出弹窗提示激活。

插件与自定义

汉化

ST 并不支持中文，如需中文需要安装汉化插件。

安装插件管理器

打开 ST 后键入 Ctrl+Shift+P 唤出命令框，输入 `Install` 后回车（完整命令是 `Install Package Control`，不区分大小写），此时应该会看到左下角有一个 `=` 在不停的左右移动。Package Control 安装完成（或失败）后会有弹窗提示，具体的加载时间取决于网络。

如果完成的弹窗显示安装失败（与网络有关），则需要手动下载 Package Control[7] 并将下载好的文件放到 ST 的数据目录下的 `\Installed Packages` 文件夹中。稍作等待，ST 会自动识别该插件（有时需要重启 ST）。

> **"ST 数据目录的路径"**
>
> Windows 下，如果在 ST 的**安装目录**下存在 `\data` 文件夹，会自动使用（或初始化）该文件夹作为数据目录。
>
> ST3 的路径一般为 `C:\Users\用户名\AppData\Roaming\Sublime Text 3`，ST4 一般为 `C:\Users\用户名\AppData\Roaming\Sublime Text`，ST 会先寻找对应版本的路径，如不存在则寻找更低版本的路径，如都不存在则新建并初始化。
>
> 在以 NOI Linux 所使用的 Ubuntu 20.04.1 中，ST3 的数据目录为 `$HOME/.config/sublime-text-3`，ST4 的数据目录为 `$HOME/.config/sublime-text`，使用的具体规则同 Windows 环境。
>
> 可以使用 `首选项 -> 浏览插件目录 ...` 快速查看数据文件夹路径下的 `\Packages` 文件夹。

安装汉化插件

再次按下 Ctrl+Shift+P 输入 `Install` 后回车（完整命令是 `Package Contrl:Install Package`），等待加载完成，接下来应该是这个界面：

输入 `Chinese` 选择 ChineseLocalizations 并回车，等待安装完毕，完成后界面会自动切换为中文（如是 ST4，因为汉化插件未更新，会少一些新增的菜单项，但一般对编辑无影响）。

调整字体

进入 `首选项 -> 设置`，在右边的用户设置中的花括号中添加一行 `"font_face": "字体名",`，ST 的设置使用 JSON 格式储存。修改完成后保存，如果系统安装了对应字体会自动切换。

一般而言，如果单论对中文的显示的话，Microsoft Yahei Consolas 和 Microsoft YaHei Mono 是比较好的选择。

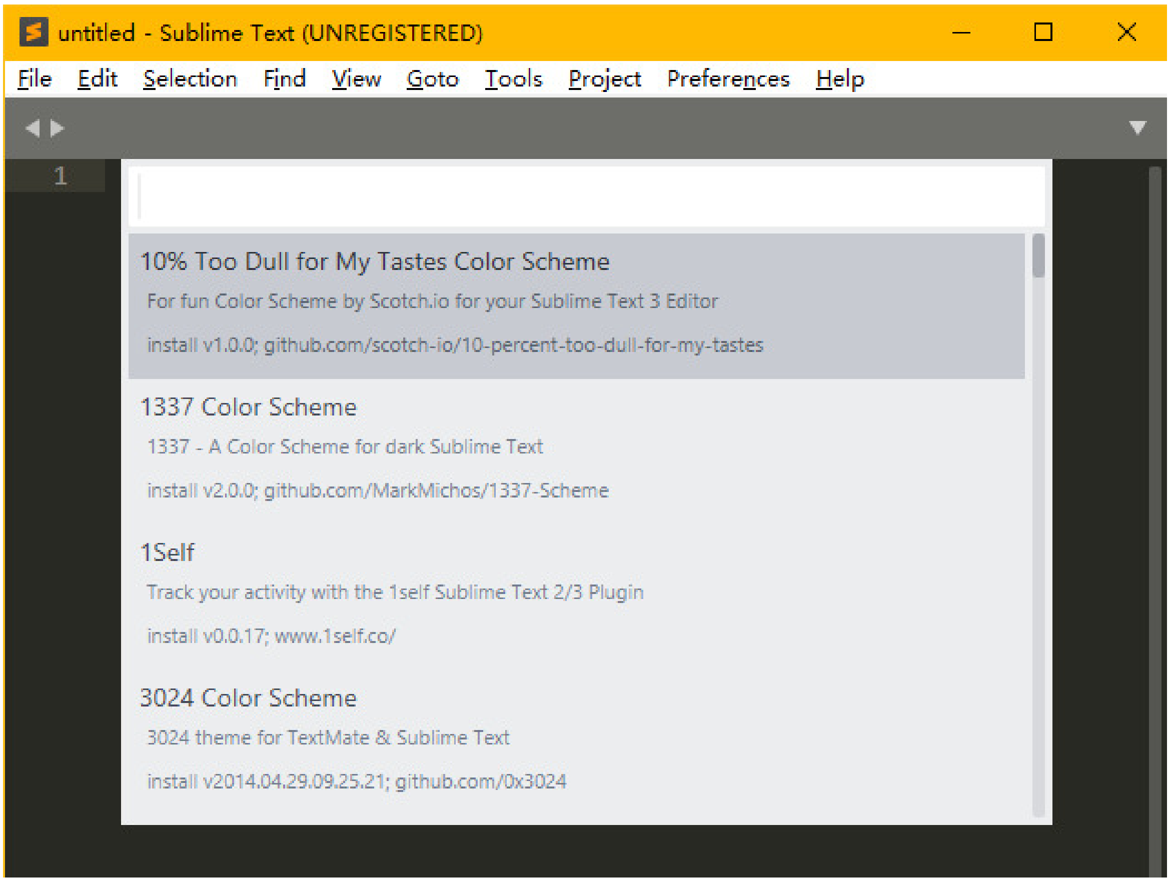


图 3.53

warning

注意任何设置（包括插件设置）即使能也不要在左边的的默认设置中修改，这是不被推荐的，如果软件（或插件）更新，默认设置会被重置。

安装插件与主题

安装插件与主题的方法与安装汉化插件一致。

输入 Ctrl+Shift+P 输入 `Install` 后回车，然后搜索插件/主题/配色即可。

插件推荐：

- BracketHighlighter：对原版的括号高亮进行了增强，必备。
- C++ Snippets：ST 自带有 C++ 代码补全，格式为大括号不换行。如果不习惯自带大括号换行的码风可以安装这个插件，同时增加了一部分补全。
- C++ 11：支持 C++ 11 标准高亮（ST4 中不需要）。
- SublimeAstyleFormatter：Astyle，用于格式化代码。
- Diffy：按 Ctrl+K&Ctrl+D 即可快速比较第一视窗与第二视窗打开的文件的不同，比较方式为逐行比较。
- ConvertToUTF8：自动识别文件编码，并支持编码转换。
- SideBarEnhancements：侧边栏增强，较为推荐。
- Transparency：窗口透明化。

有其他需要可以尝试搜索。

一些主题：[3]

1337（单配色方案）、3024（单配色方案）、Grandson-of-Obsidian（单配色方案）、Seti_UI（单主题，额外包含

git 等格式的高亮，较为推荐)、Material Theme、Predawn、Agila、Materialize。

如果要编辑自己的配色方案，可以访问 tmTheme Editor[8]。

如是 ST4 则可以在 `Preferences->Customize Color Scheme` 中调整配色方案或 `Preferences->Customize Scheme` 中调整主题。

开启 Vi Mode

ST 的开发者为 Vi 使用者提供了一个可选插件 Vintage，可模拟 Vi 的大部分功能。

开启方式

Vintage 插件默认是禁用的。可以通过首选项 -> 设置在用户设置中，将 `"ignored_packages"` 一项中的 Vintage 删除并保存（不要整个删除，只删除 Vintage)，ST 的状态栏左边就会出现 `INSERT MODE`，此时 Vintage 插件已开启。

或者按 Ctrl+Shift+P，然后输入 `Enable` 选择 `Package Control: Enable Package` 并回车，选择 Vintage 即可，该方法需要 Package Control。

相关配置

如果想让上下左右键失效，可以在首选项 -> 快捷键设置中添加：

```
{ "keys": ["left"], "command": ""},
{ "keys": ["right"], "command": ""},
{ "keys": ["up"], "command": ""},
{ "keys": ["down"], "command": ""},
```

如要使 ST 以命令模式启动，则可以在首选项 -> 设置中添加：

```
"vintage_start_in_command_mode": true,
```

也可以通过快捷键设置将进入命令模式设置成任意键（具体详见设置快捷键）。

与 Vi 的不同

ST 的 Vintage 插件与 Vi 有一定不同，部分列于此处：

- 在插入模式下用选中不会进入可视模式，这时输入不会被识别为命令而是直接替换文本。可视模式只有命令模式下才能进入。
- `r`、`R`、`zA`、`za`、`zi`、`z=`、`@` 与使用 [、] 或” 键的命令不存在。
- 使用 Ctrl、Shift 和 Alt 键的快捷键会保留为 ST 设置的快捷键，如 Ctrl+V 不会进入可视模式而是正常粘贴。
- 命令行模式只保留了 `:e`、`:0`、`:$`、`:s`。
- 使用 `\` 与 `?` 命令会自动唤出搜索框，而不是直接在命令中键入单词进行搜索。同时，数字将无法生效。
- `q` 宏录制命令会启动 ST 自带的宏录制，按 Q 后需要再按一个键启动录制，但录制的宏不会与该键绑定，需要按 Ctrl+Shift+Q 才能启动。如果需要保存，需要工具 -> 保存宏，快捷键需要设置。
- 无法使用数字 `+.` 的组合。

设置

设置 ST

在首选项 -> 设置中设置，这里列举部分较有用的选项：

```
{
    //字体大小
        "font_size": 11,

        //字体，可以不设置，默认为 Consolas
        "font_face": "",
```

```
        //Tab 自动转换为空格
        "translate_tabs_to_spaces": true,

        //缩进宽度
        "tab_size": 4,

        //行高亮
        "highlight_line": true,

        //保存时自动在文件尾增加换行
        "trim_trailing_white_space_on_save": true,

        //在选择时查找自动只查找选择范围
        "auto_find_in_selection": true,

        //禁用了 OI 中不太可能用到的插件，可以自己调整
        "ignored_packages": [
                "ActionScript", "AppleScript", "ASP", "Batch File", "C#",
                "Clojure", "CSS", "D", "Diff", "Erlang", "Git Formats",
                "Go", "Graphviz", "Groovy", "Haskell", "HTML", "Java",
                "LaTeX", "Lisp", "Lua", "Makefile", "Matlab",
                "Objective-C", "OCaml", "Perl", "PHP", "Python",
                "R","Rails", "RestructuredText", "Ruby", "Rust",
                "Scala", "ShellScript", "SQL", "TCL", "Textile", "XML",
        ],

        //相对行号，可配合 Vintage 插件快速跳转
        "relative_line_numbers": false,
}
```

设置快捷键

在首选项 -> 快捷键设置中设置，在左边找到需要修改的功能，然后复制到右边并修改按键即可。

例如，如果要把 Ctrl+B 的编译改为 F9（如果不令原有的快捷键失效，实际是增加一个触发方式），则可以在首选项 -> 快捷键设置中添加：

```
//将 build 命令改为f9
{ "keys": ["f9"], "command": "build" },

//将原有的f9对应的行排序功能的快捷键改为 shift+f9，由于大部分时候这个功能用不到，这
一行也可以不添加
{ "keys": ["shift+f9"], "command": "sort_lines", "args": {"case_sensitive": fals
e} },
```

设置插件

插件的设置可以在首选项 ->Package Setting-> 插件名中找到，做修改时请注意不要修改默认设置。

例如，这里给出 BracketHighlighter 的一些设置，在首选项 ->Package Setting->BracketHighlighter->Bracket Setting 中修改：

```
{
    //在匹配的括号之间行的行首显示一条线，可以快速找到括号的范围
        "content_highlight_bar": true,
```

```
        //在小地图中显示匹配的括号
        "show_in_minimap": true,

        //忽略匹配范围限制
        "ignore_threshold": true,

        //style 高亮样式，bold 为块高亮，underline为加粗下划线，outline为外围一圈
        //color 为颜色，默认设置中已经包含了所有支持的颜色
        //icon为在侧边栏显示的标志
        "bracket_styles": {
                "default": {"icon": "dot", "color": "region.yellowish", "style": "bold",},
                "unmatched": {"icon": "question", "color": "region.redish", "style": "outline"
,},
                "curly": {"icon": "curly_bracket", "color": "region.purplish",},
                "round": {"icon": "round_bracket", "color": "region.yellowish",},
                "square": {"icon": "square_bracket", "color": "region.bluish",},
                "angle": {"icon": "angle_bracket", "color": "region.orangish",},
                "tag": {"icon": "tag", "color": "region.orangish",},
                "c_define": {"icon": "hash", "color": "region.yellowish",},
                "single_quote": {"icon": "single_quote", "color": "region.greenish",},
                "double_quote": {"icon": "double_quote", "color": "region.greenish",},
                "regex": {"icon": "star", "color": "region.greenish",}
        }
}
```

修改与添加

有时候，插件的某些地方可能并不尽如人意，或想对插件进行汉化，这时就需要对插件做一些修改。

插件存放的位置是数据目录下的 \Installed Packages 文件夹。

里面的文件以 .sublime-package 为后缀，实际上为 .zip 格式，可以使用解压缩软件打开。

例如，如果要修改自动补全，可以打开 ST 的**安装目录** \Packages\C++ 插件中的 \Snippets*.sublime-snippet 文件修改，如要**增添**自动补全，可以安装 C++ Snippets 并在其中修改或添加文件（或新建一个插件，但不能直接添加进自带的 C++ 插件，否则无法被识别）。保存任何修改时**必须**关闭 ST，且请提前做好备份，否则可能出现文件丢失。

当然，任何增添都可以放在数据目录路径下的 \Packages\User\ 下，这总是被支持的。

例如，一个文件模板的补全可以这么写：

```
<snippet>
    <description>template_code</description> <!-- 这里的内容是补全内容的预览 -->
        <content><![CDATA[#include <cstdio>
using namespace std;

int main() {
  freopen("${1:file name}.in", "r", stdin);
  freopen("$1.out", "w", stdout);
  ${0:/* code */}
  fclose(stdin);
  fclose(stdout);
  return 0;
}]]></content>
        <tabTrigger>code</tabTrigger> <!-- 这里的内容是补全的触发文本 -->
        <scope>source.c++</scope> <!-- 这里的内容是补全适用语言 -->
</snippet>
```

以下列出部分文件后缀以及其用途，具体的插件开发教程详见 社区文档[9] 和 官方文档[10]：

后缀名	用途
.sublime-build	编译系统文件
.sublime-completions	文件名补全列表（一般为头文件）
.sublime-syntax	语法高亮文件
.sublime-settings	设置文件
.tmPreferences	首选项中的列表文件
.sublime-keymap	快捷键设置文件
.sublime-snippet	代码补全文件
.sublime-commands	命令定义文件
.sublime-menu	ST UI 文件，包括侧边栏以及顶部菜单栏（汉化的主要对象）

由于插件更新会直接覆盖原文件，所以建议备份更改的文件。

编辑

设置语法

按 Ctrl+Shift+P 后输入语法即可，或者按右下角的 `Plain Text` 然后修改为需要的语言，同时在视图 `->` 语法中也可以设置。

快捷键

ST 有复合快捷键，如 Ctrl+K&Ctrl+Backspace 表示先按 Ctrl+K 再按 Ctrl+Backspace。

部分快捷键：

按键	命令
Ctrl+X	剪切当前行
Ctrl+Shift+K	删除行
Ctrl+Enter	在下方插入行
Ctrl+Shift+Enter	在上方插入行
Ctrl+Shift+Up	行上移
Ctrl+Shift+Down	行下移
Ctrl+L	选择行，重复以向下选择多行
Ctrl+D	选择词，重复以选择多个相同词，并进入多重选择模式（用于快速批量更改）
Ctrl+M	跳转到匹配的括号
Ctrl+Shift+M	选择括号内的内容（不包括括号），重复以包括括号
Ctrl+K&Ctrl+K	删至行尾（复合快捷键，建议使用 Vim 模式代替）

按键	命令
Ctrl+K&Ctrl+Backspace	删至行首（复合快捷键，建议使用 Vim 模式代替）
Ctrl+]	缩进当前（选择的）行
Ctrl+[	取消缩进当前（选择的）行
Ctrl+Shift+D	复制当前行，并插入在下一行
Ctrl+J	合并下一行与当前行
Ctrl+Shift+V	粘贴并缩进（用于整段粘贴代码）
Ctrl+K&Ctrl+Shift+V	从历史粘贴（复合快捷键，建议修改为 Ctrl+Alt+V）
Ctrl+Alt+Down	光标下移，并保留当前行光标（进入多重选择模式）
Ctrl+Alt+Up	光标上移，并保留当前行光标（进入多重选择模式）
Ctrl+R	跳至文件中的任意符号（函数或类型定义）
Ctrl+Shift+R	跳至项目中的任意符号（函数或类型定义）
Ctrl+P	跳至任意文件（曾经打开过或在项目中且存在的文件）
~	转换选择内容的大小写

自动补全

ST 有丰富的补全功能，可能的补全内容会在光标下方显示，按 Tab 或 Enter 进行补全（ST4 中，如进行一个非 Snippet 类型的补全，接下来再按 Tab 可继续选择为以该补全为子串的补全）。

Snippet 类型的补全一般会有一些编辑块，补全后会自动选择为替换文本，如果是 `for` 等含有多个编辑块的复杂补全，编辑完成后再次按 Tab 完成下一个编辑块，此时要在编辑块中触发补全需要使用 Enter（在 ST4 中可继续使用 Tab）。

如果没有自动补全，请如下修复：

1. 检查是否切换了语言，ST 默认新建文件为 Plain Text。
2. 进入首选项 -> 设置然后添上两行：

```
"auto_complete": true,
"auto_match_enabled": true,
```

多重选择

按住 Ctrl 并用鼠标单击即可在屏幕上增加光标，Ctrl+Alt+Up 或 Ctrl+Alt+Down 可以在相邻两行直接增加光标，任何编辑性质的操作会同时应用至所有光标。

查找与替换

Ctrl+F 为查找，F3 为查找下一个 Shift+F3 为查找上一个，Ctrl+H 为替换。

五个查找选项分别为正则表达式匹配、大小写敏感、全字匹配、循环查找、在选段中查找。

建议在首选项中将 `"auto_find_in_selection"` 设置为 `true`。这样在选择超过一个词时使用查找会自动只在选段中查找。

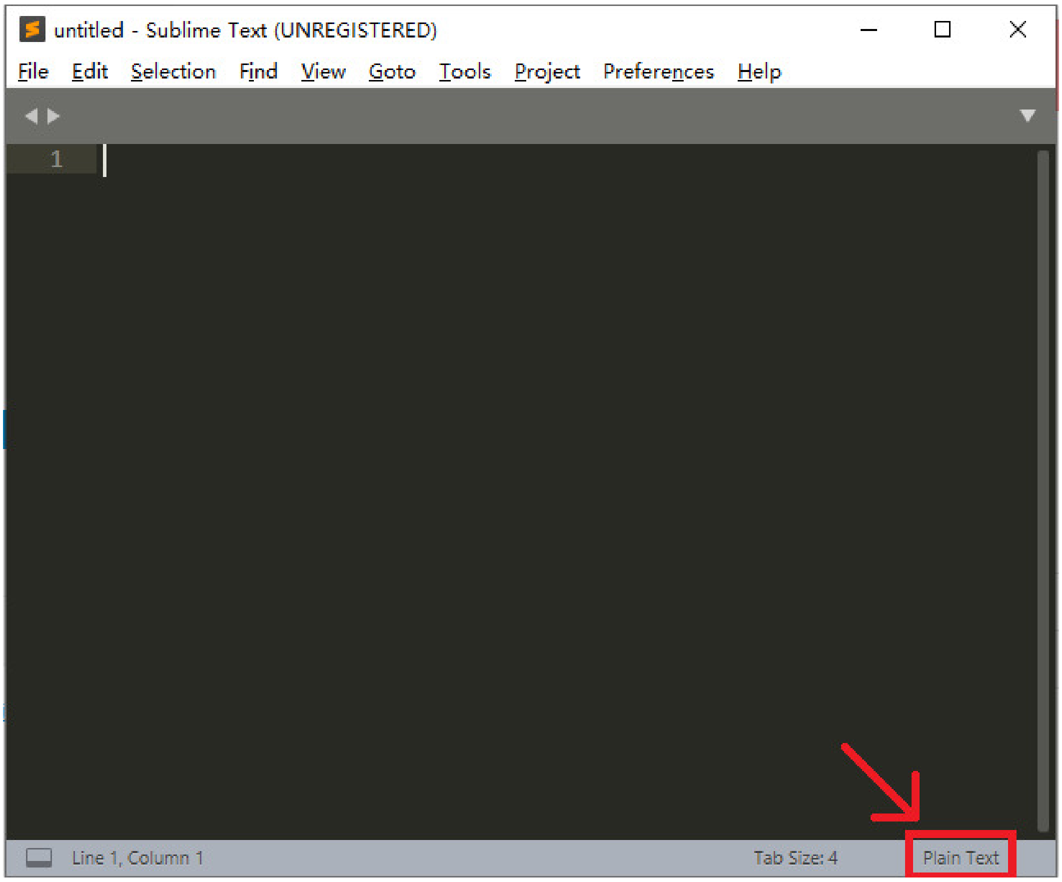


图 3.54

演示

热启动

尝试在 ST 中键入一些内容，并直接把整个 ST 关闭，ST 会直接关闭且没有任何提示，再打开 ST 时只要不对电脑进行数据还原就不会丢失任何数据。

多重选择

如果要把以下代码中的所有 bok 改为 book，只需将光标放置在任意一个 bok 中，长按 Ctrl+D 即可快速选中。

```
int check() {
  book[1] = 1, book[2] = 1, book[3] = 1, bok[1] = 1, bok[2] = 1, bok[3] = 1,
  bok[4] = 1, bok[5] = 1;
}
```

如果要将下列文件中的所有等号后面改成 "good"，只需在将光标放置于第一行的 aaa 前，并按五次 Ctrl+Alt+Down，再然后按下 Ctrl+D 并键入 good 即可。

或选中 "a 并按住 Ctrl+D，然后按 Right、Ctrl+D，之后键入即可。

```
s[1] = "aaa";
s[2] = "aab";
s[3] = "aac";
s[4] = "good";
s[5] = "aae";
s[6] = "aaf";
```

如要为下列所有 a + b 加上括号，只需选择一个 a + b，按住 Ctrl+D 并键入 (即可（如选择一定区域，则任意左括号键入会为该区域两边添加匹配的括号）。

```
a + b*a + b*a + b
```

查找与替换

如果要将下列文件中的所有等号后面改成 "good"，也可以用 Ctrl+H 使用替换，打开正则，输入 ".*"，并替换成 "good" 即可。

```
s[1] = "aaa";
s[2] = "aab";
s[3] = "aac";
s[4] = "good";
s[5] = "aae";
s[6] = "aaf";
```

如要为以下代码添加分号，只需使用选择区域替换，打开正则，输入 \n，并替换成 ;\n 即可。

```
int main() {  int a, b  cin >> a >> b  cout << a + b  return 0}
```

宏录制

如要为以下代码添加分号，可以按 Ctrl+q 启动宏录制接下来依次按 End、;、Down 再按 Ctrl+q 结束宏录制（中途左下角不会全程显示正在录制，但确实在录制），接下来重复 Ctrl+Shift+q 即可。

```
int main() {
  int a, b
  cin >> a >> b
  cout << a + b
  return 0
}
```

" 如已开启 Vintage 插件"

执行一次 Ctrl+Shift+q 后，可以 Esc 进入命令模式，输入 .. 即可（. 命令可以重复 ST3 命令）

关于宏的保存与绑定按键详见 社区文档[11]。

编译与运行

ST 的编译环境已经配置好了，可以直接使用。

Windows 环境下需要将 g++ 所在目录添加到环境变量中，并重启 ST。

编译

直接按 Ctrl+B 编译，第一次使用会需要选择编译系统，选择 C++ Single File（编译）或 C++ Single File - Run（编译并运行）。

修改编译选项

ST 默认的编译选项为 g++ "${file}" -o "${file_path}/${file_base_name}"，如果要修改编译选项，可以新建一个编译系统。

进入系统 -> 编译系统 -> 新建编译系统 … 然后在大括号中间输入：

```
//编译选项可以自己调整
"cmd": ["cmd","/c","g++","-Wall","${file}","-o","${file_path}/${file_base_name}"
    ,"&&","cmd","/c","${file_path}/${file_base_name}"],  //这部分为运行
```

```
//这一行可以让 ST3图形化显示报错，如果习惯了看 g++ 返回的信息可以去掉
"file_regex": "^(..[^:]*):([0-9]+):?([0-9]+)?:? (.*)$",
```

保存后按 Ctrl+Shift+B 切换编译系统就可以使用了，这里的配置是编译并在外部 CMD 运行。

保存的文件为数据目录路径下的 `\Packages\User\ 编译系统名 .sublime-build` 可以反复修改。

运行

如果编译时选择 `C++ Single File - Run`（即编译后运行）或配置了自动运行，那么在下方弹出的编译信息窗口应该不会有任何显示（除非编译错误），因为 ST 的编译信息窗口实际上是一个终端，可以直接输入数据。

运行结束后会提示程序的运行时间，其计时为从按下 Ctrl+B 到全部 CMD 命令运行结束的时间，也就是说包括编译的时间和输入的时间，以及如果在外部 CMD 运行还包括 CMD 开启关闭的时间。

warning

这个窗口无法输入 F6 或 Ctrl+Z，如果运行读入到文件末尾的程序请使用文件输入，或配置在外部 CMD 运行。

调试

可以安装插件使 ST 支持图形化 gdb 调试，但不建议依赖插件进行 gdb 调试。

更好的做法是在配置编译系统时加上相关命令启动 gdb，在外部进行命令行调试。

杂项

- 把文件夹拖进 ST 中并按 Ctrl+K&Ctrl+B 开启侧边栏，从而快速切换文件。
- 善用跳转功能，尤其是 Ctrl+P 进行文件间跳转与 Ctrl+R 进行函数跳转。
- ST 支持 git[4]。
- ST 的所有配置储存在数据目录下，可以随意打包，但注册信息无法在多台电脑上使用。

外部链接

- 使用命令行调试
- Sublime Text 3 官方文档[10]
- Sublime Text 社区文档[12]

参考资料与注释

[1] NOI Linux 2.0 发布

[2] Sublime Text 4 发布

[3] 便捷清新的文本编辑器 sublime

[4] Sublime Text Git 集成

[5] Sublime Text 4 的下载页面

[6] Sublime Text 3 的下载页面

[7] Package Control

[8] tmTheme Editor

[9] 社区文档

[10] 官方文档 [10-1] [10-2]

[11] 社区文档

[12] Sublime Text 社区文档

3.2.14 CP Editor

Authors: zarrtic

简介

CP Editor[2] 专为算法竞赛设计，不像其它 IDE 主要是为了开发设计的。它可以帮助你自动化编译、运行、测试，从而让你专注于算法设计。它甚至可以从各种算法竞赛网站上获取样例，将代码提交到 Codeforces 上！

下载与安装

参见 安装 ｜CP Editor[3]。

基础配置

> CP Editor 内部没有集成编译器，需要自己安装配置编译器，如有需要请参考本站关于编译器安装相关的文章[1]。

- 设置默认语言
  编辑器默认的语言为 `C++`。
- 设置 `C++` 命令
  需要设置一些必要的编译命令，这个要根据编译器来设定。
- 设置模板
  新建文件的时候会自动初始化的模板，需要注意的是 CP Editor 需要的是一个 `xxx.cpp` 的文件作为模板文件。

> 完成了以上的基本操作你就可以使用最基本的功能了。

参考资料

[1] 编译器 - OI Wiki

[2] CP Editor

[3] 安装 | CP Editor

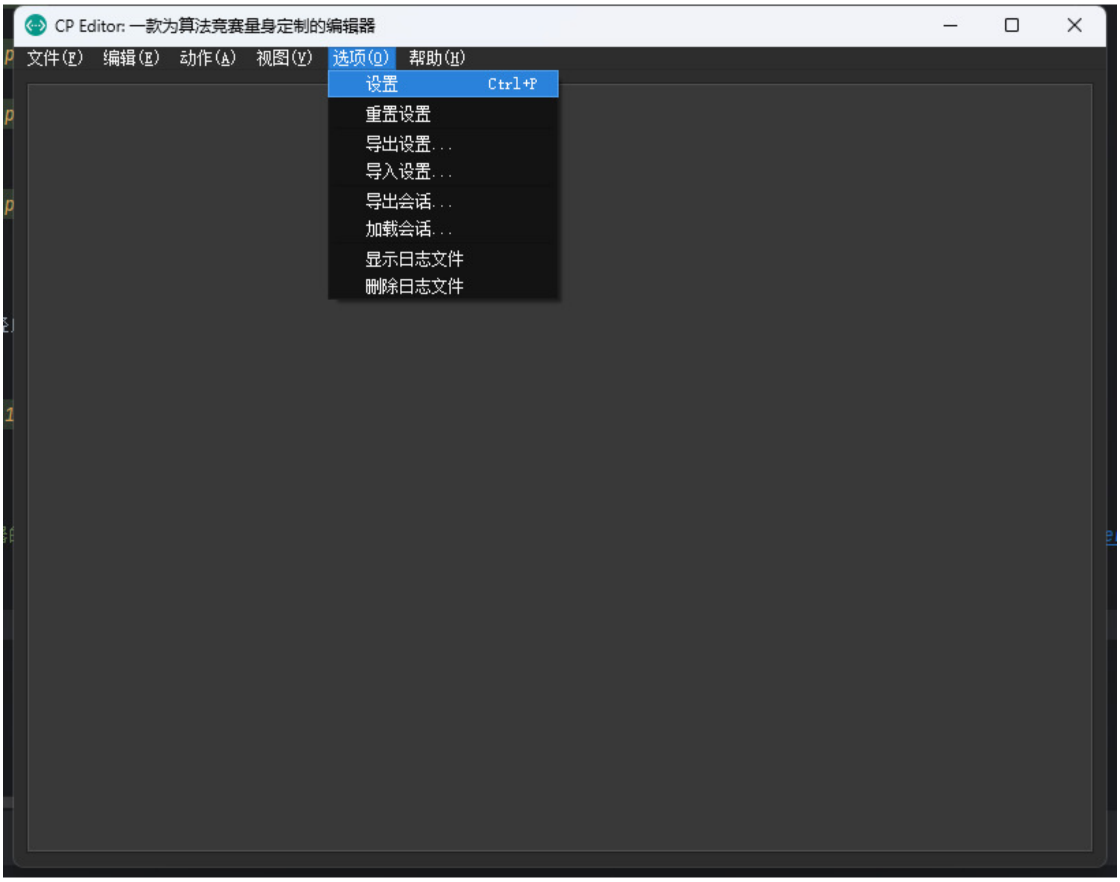


图 3.55 cp-setting-lang-1

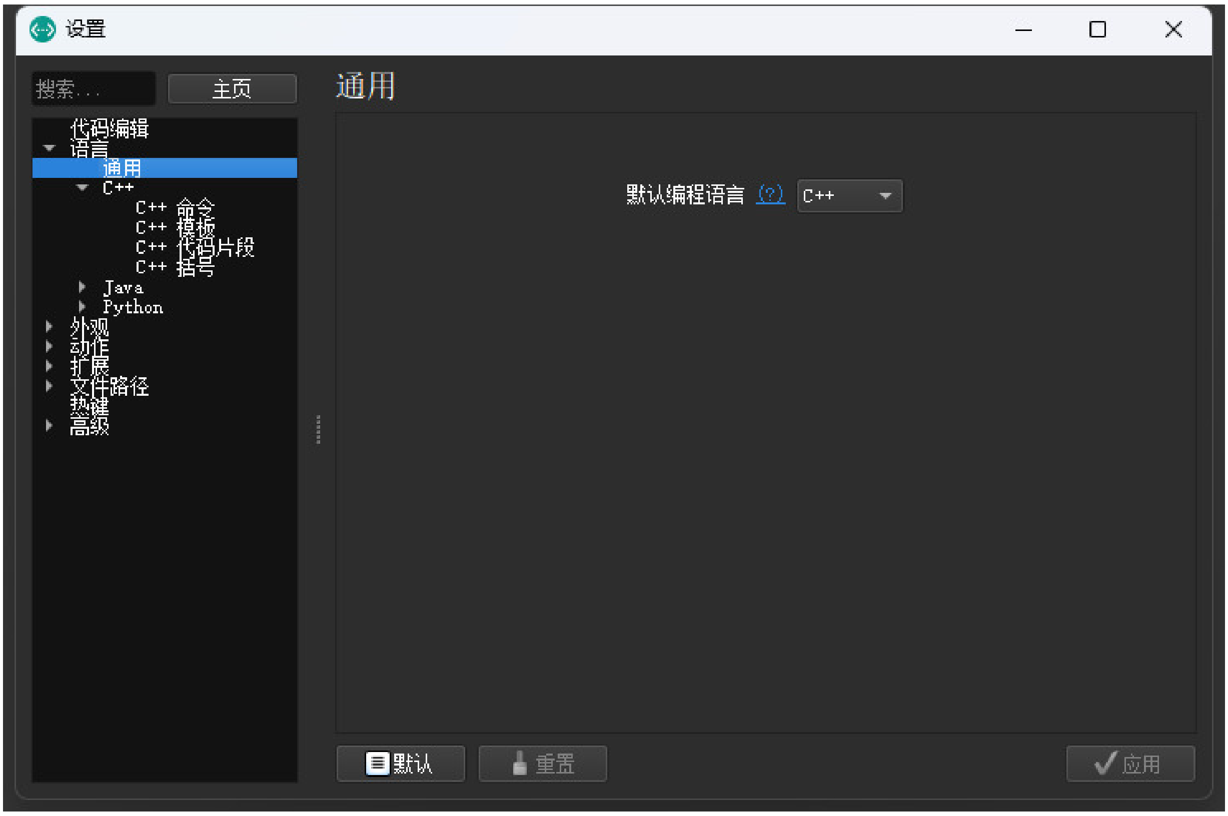


图 3.56 cp-setting-lang-2

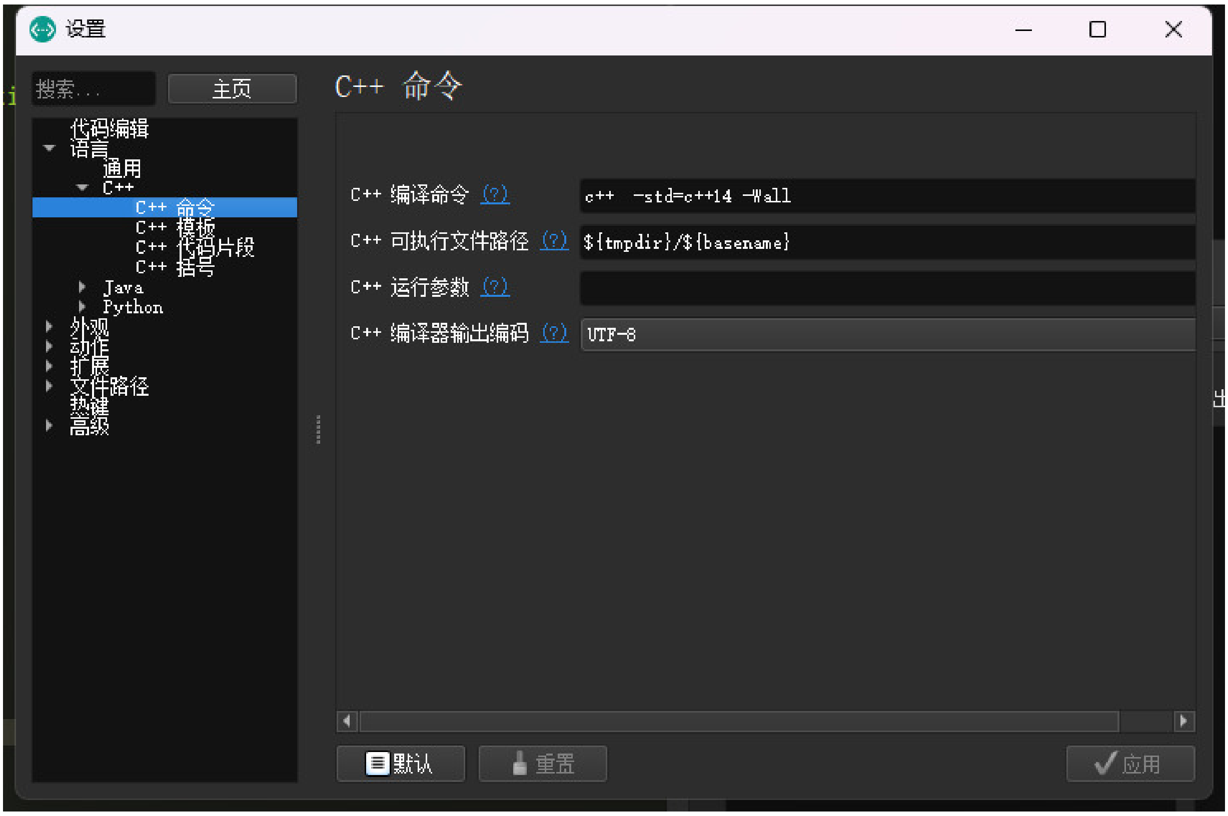


图 3.57 cp-setting-lang-3

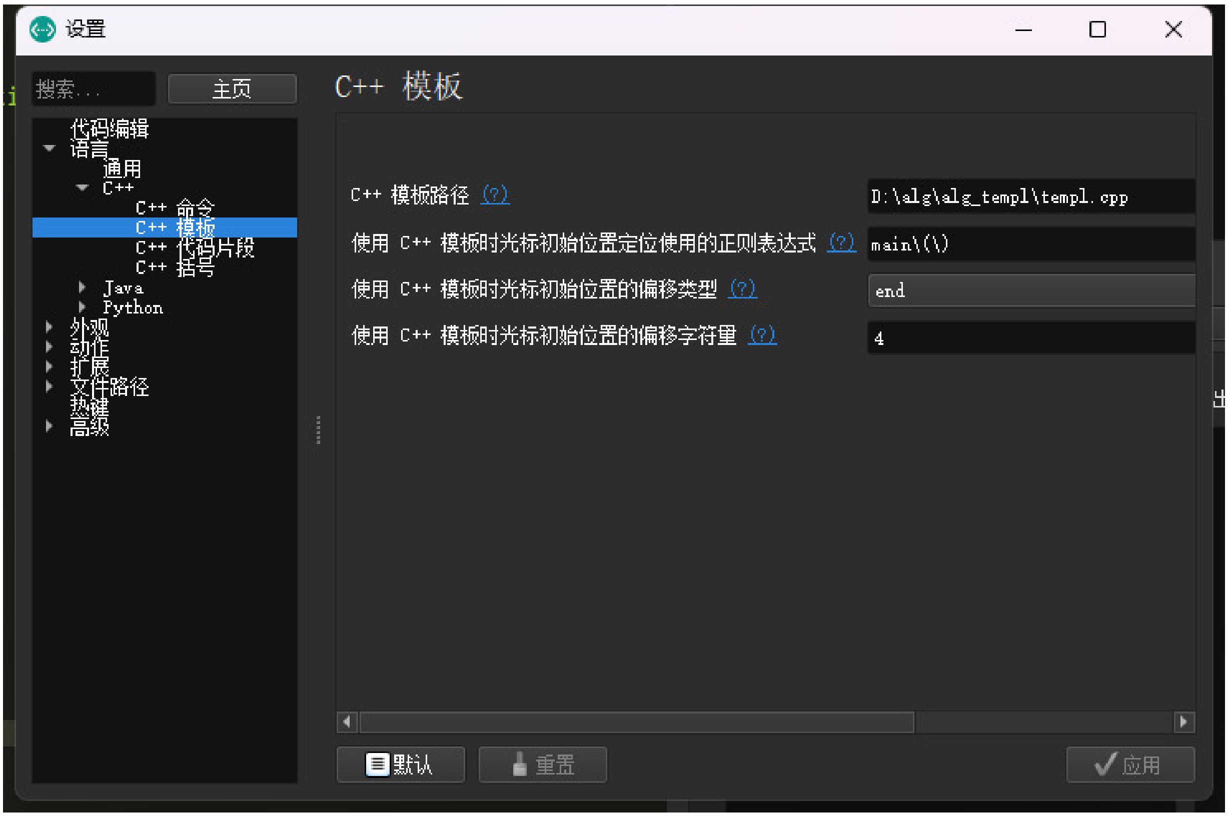


图 3.58 cp-setting-lang-4

3.3 评测工具

3.3.1 评测工具简介

Authors: Ir1d, HeRaNO, NachtgeistW, i-Yirannn, bear-good, ranwen, CoelacanthusHex, billchenchina, Tiger3018, Xeonacid, renbaoshuo

评测软件是用于本地测试分数的软件。使用者在将代码提交到 OJ 前，可以使用评测软件对自己的程序估分。

本部分介绍了以下几种评测软件：

- Arbiter
- CCR Plus
- Cena
- Lemon

3.3.2 Arbiter

Authors: Ir1d, HeRaNO, NachtgeistW, i-Yirannn, bear-good, ranwen, CoelacanthusHex, billchenchina, Tiger3018, Xeonacid

Arbiter

Arbiter 为北京航空航天大学为 NOI Linux 开发的评测工具，现已用于各大 NOI 系列程序设计竞赛的评测。据吕凯风在 2016 年冬令营上的讲稿《下一代测评系统》，Arbiter 是由北京航空航天大学的团队（GAIT）在尹宝林老师的带领下开发完成的。

在 NOI Linux 更新到 2.0 版本后，Arbiter 也用 Qt 5.12.8 重新编译，并发布为 Arbiter 2.0。因为之后的测评环境均使用 NOI Linux 2.0，因此以下介绍使用的 Arbiter 版本均为 NOI Linux 2.0 中自带的 Arbiter 2.0。

此测评软件仅能在 NOI Linux 下找到。二进制文件位置为 `/usr/local/arbiter/local/arbiter_local`。

使用方法

配置程序

配置选手源程序文件夹和选手名单。选手文件夹如 NOIP 格式创建：

```
players/
| -- day1
|     | -- <contestant_1's ID>
|     |      | -- <problem_1>
|     |      |    `-- <problem_1>.c/cpp/pas
|     |      | -- <problem_2>
|     |      |    `-- <problem_2>.c/cpp/pas
|     |      | ...
|     |      | -- <problem_x>
|     |          `-- <problem_x>.c/cpp/pas
|     | -- <contestant_2's ID>
|     |      | -- <problem_1>
|     |      ...
|     ...
| -- day2
|     | -- <contestant_1's ID>
|     |      | -- <problem_1>
```

```
|    |     |   `-- <problem_1>.c/cpp/pas
|    |     | -- <problem_2>
|    |     |   `-- <problem_2>.c/cpp/pas
|    |     | ...
|    |     | -- <problem_x>
|    |         `-- <problem_x>.c/cpp/pas
|    | -- <contestant_2's ID>
|    |     | -- <problem_1>
|    |     ...
|    ...
...
```

其中，`day<x>` 中的 `<x>` 是场次编号，`<contestant_x's ID>` 指的是选手编号，形如 `<` 省份 `>-<` 编号 `>`，例如 HL-001，JL-125 等等；`<problem_x>` 指的是题目名称。在自测时可以使用字母、短线（即 `-`）和数字的组合作为选手编号。

选手名单格式如下：

```
<contestant_1's ID>,<contestant_1's name>
<contestant_2's ID>,<contestant_2's name>
...
```

其中，`<contestant_x's name>` 表示选手姓名。保存这个文件为纯文本文件或 csv 文件，可以使用 `UTF-8` 编码。

选手名单也可以在启动 Arbiter 后手动添加。

接下来配置测试数据。每组数据的命名格式如下：

```
<problem_x><y>.in <problem_x><y>.ans
```

其中，`<y>` 是数据编号，编号从 1 开始。默认测试数据后缀名是 `.ans`，选手输出的后缀名是 `.out`，不能混淆。

不用将每题的测试数据放置在各题的文件夹里，只需要放在一起即可。

然后开始测评文件夹的配置。

左下角「显示应用程序」-「全部」-「Arbiter_local」，启动 Arbiter。

点击 OPEN 可以打开已经建立的比赛，之后需选择对应比赛文件夹下的 `setup.cfg` 文件；点击 NEW 可以新建一个竞赛，并设置名称和比赛目录。注意，需要在用户**主目录下**新建一个文件夹，然后选择其为比赛目录，如果在桌面上建立比赛目录的话无法测评。出现这种问题很有可能是因为比赛文件夹路径中不能包含中文。

在左边试题概要里「右键」-「添加考试」，再在考试标签上「右键」-「添加试题」，新建出试题即可。

单击考试左边的向下箭头即可全部显示，单击试题标签对试题名称进行修改，改为题目的英文名称，同时修改题目时间与空间限制和比较方式。比较方式十分不推荐用「全文完全直接比较」，对于 Windows 下制作的数据十分不友好。可以根据题目自主选择比较器，但是需要注意必须选择一个比较器，否则测评结果将是 `No Score.`。

点击「文件」-「保存」。该操作不可省略，否则程序将不会生成题目配置文件。注意每一次对题目配置的修改都要保存。

此时，打开考试文件夹，会发现有如下内容。

```
<name>/
| -- data
| -- evaldata
| -- filter
| -- final
| -- players
| -- result
| -- tmp
`-- day1.info
`-- player.info
`-- setup.cfg
```

图 3.59 Arbiter_Home

图 3.60 add_problem

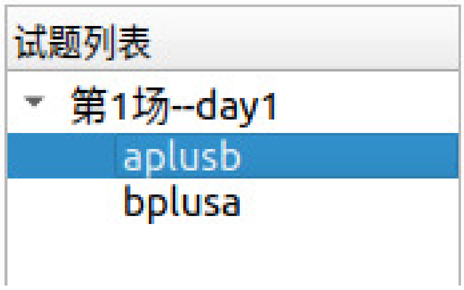


图 3.61 problem_list

```
`-- task1_1.info
`-- task1_2.info
`-- team.info
```

filter 文件夹放置了一些比较器；result 文件夹存放选手的测评结果；tmp 文件夹是测评时的缓存文件夹。其中 day<x>.info 为场次配置文件，<x> 为场次编号；task<x>_<y>.info 文件为题目配置文件，<x> 为场次编号，<y> 为题目序号。

把已经建好的选手程序文件夹放在 players/ 目录下，注意最外层应按照考试日建立相应的 day<x> 文件夹。将所有测试数据（不放在文件夹里）放在 evaldata 中。如果使用了自定义校验器，则需要将自定义校验器放在 filter 中。

正式测评

点开「试题评测」标签，会出现如下页面：

图 3.62 Pretest

如果选手名单已经建立了，直接选择右边的「导入名单」进行导入。如果人数较少，可以选择右边的「添加选手」进行导入。

导入后的页面如图。

图 3.63 Test

示例中的编号是 `HL-001`，程序会自动识别出「所属」一栏。如果不是 NOIP 规范的编号是识别不出来的。

把测评第 0 场变为测评第 1 场（或者其他场次）。然后选择右边的全选（或选择指定的选手），再选择下面的评测选定选手，选择要测评的题目（或全部试题），最后等待测评结束即可。

测试点详细信息需要在 `result` 文件夹下查看，文件夹下会有选手的结果文件夹，结果文件的后缀名为 `.result`，用纯文本方式查看即可。如果出现 `No score file.` 的错误，可以检查测评时是否生成了 `/tmp/_eval.score` 文件。

自定义校验器的编写

反编译其他校验器，可以知道运行自定义校验器的命令是 `<problem>_e <in> <out> <ans>`。后三个参数分别代表输入，选手输出和答案文件。最终的评分结果需写入 `/tmp/_eval.score` 文件中，第一行是测评信息，第二行是分数，10 分为满分。

编译后自定义校验器的名称必须为 `<problem>_e`，其中 `<problem>` 为题目名称。在配置题目时选择自定义校验器，然后选择需要的自定义校验器即可。

在试题管理中题目配置的地方将提交方式由源代码改为答案文件，然后选择自定义校验器，可以测试提交答案题。

注意事项

- 据说很容易死机，如大量测评时移动鼠标会导致死机。
- 据说不定时闪退，需要注意及时保存比赛。可能有些闪退原因是没有保存比赛。
- 据说配置时需要注意权限问题（但是笔者并未遇到）。
- 由于 Linux 运行时栈限制，如果要开无限栈，应在终端先输入 `ulimit -s unlimited` 后执行 `arbiter_local` 打开测评器，否则可能出现 `Exceeding memory limit` 的问题。
- 对于正式测评，在题目准备时需要让所有题目空间限制一致。测评时将命令中的 `unlimited` 换为题目空间限制

的千字节数（KiB），如：题目空间限制为 512 MiB，则命令为 `ulimit -s 524288`。
- 导致这一问题的主要原因是直接启动 Arbiter，其父进程为 GNOME，子进程继承了父进程的栈空间限制。

漏洞

由于长期缺乏维护，系统存在一些漏洞，如可以使用 `#pragma G++ optimize("O2")` 和 `__attribute__((__optimize__("-O2")))` 等。可以使用 gcc-plugins-for-oi[1] 在编译期实现对这些命令的检测。

评价

Arbiter 1.0.2 在开发完成后就一直没有实质性更新，导致测评体验极差，UI 脱离现代审美。在 NOI Linux 1.4.1 中，它和 NOI Linux 自带的 GUIDE 一样沦为选手与教练疯狂吐槽的对象。在 NOI Linux 2.0 中，除了比较器移除了源代码和软件整体使用 Qt 5 重新编译外，并没有很大的变化，一些稳定性问题仍未得到解决。

参考资料与注释

[1] gcc-plugins-for-oi

3.3.3 Cena

Authors: Ir1d, HeRaNO, NachtgeistW, i-Yirannn, bear-good, ranwen, CoelacanthusHex, billchenchina, Tiger3018, Xeonacid

Cena

Cena 是由刘其帅和李子星使用 Pascal 语言编写的开源评测工具，是流传最广泛的本地评测工具。Cena 最初开源于 Google Code 平台，由于不明原因 Google 删除了 Cena 项目。目前可以在 Web Archive[1] 上找到 Cena 的官网。

Cena 对权限的限制不是很明确，测试的时候可以读测点 AC。

Cena 的源代码托管于 oi-archive/cena[2]。

参考资料与注释

[1] Web Archive

[2] oi-archive/cena

3.3.4 CCR Plus

Authors: Ir1d, HeRaNO, NachtgeistW, i-Yirannn, bear-good, ranwen, CoelacanthusHex, billchenchina, Tiger3018, Xeonacid

CCR Plus

CCR Plus 是一款适用于 NOI 系列比赛的开源的跨平台测评环境，使用 Qt 编写，目前支持 Windows 和 Linux。

源代码托管于 sxyzccr/CCR-Plus[1]。

参考资料与注释

[1] sxyzccr/CCR-Plus

3.3.5 Lemon

Authors: Ir1d, HeRaNO, NachtgeistW, i-Yirannn, bear-good, ranwen, CoelacanthusHex, billchenchina, Tiger3018, Xeonacid

Lemon

warning

macOS 下 Lemon 可能会出现内存测试不准确的情况，因为 macOS 缺少部分 Linux 的监测工具，且 Lemon-Linux 也没有针对 macOS 进行优化。

Lemon 是 zhipeng-jia 编写的开源评测工具，源代码托管于 zhipeng-jia/project-lemon[1]。

可直接运行的版本

- Ir1d 提供了一份 Linux 下编译好的版本，源代码托管于 FreestyleOJ/Project_lemon[2]。
- （已停止维护）Menci 提供了一份更新的版本，源代码托管于 Menci/Lemon[3]。
- （已停止维护）Dust1404 维护了一份支持子文件夹和单题测试等功能的版本，源代码托管于 Dust1404/Project_LemonPlus[4]。
- iotang 和 Coelacanthus 维护了一份支持子文件夹和单题测试等功能的版本，源代码托管于 Project-LemonLime/Project_LemonLime[5]。

自行编译

Ubuntu：

```
sudo apt update
sudo apt install qt5-default build-essential git -y
git clone --depth=1 https://github.com/Menci/Lemon.git
cd lemon
# 可以修改 -j 后面的数字来调整 make job 的线程数
./make -j2
sudo install -Dm755 -t /usr/bin/ Lemon
```

如要编译 LemonLime，请参阅 LemonLime 的 编译手册[6]。

数据格式

首先打开 lemon 选择「新建试题」，然后打开新建试题的文件夹。

题目和数据应该如以下格式所示：

```
    data
        gendata.py
        product
            product100.in
            product100.out
            product10.in
            product10.out
            product11.in
...
```

当所有试题添加完成后，回到 lemon 选择「自动添加试题」。此时题目和数据点将显示在 lemon 当中。

参考资料与注释

[1] zhipeng-jia/project-lemon

[2] FreestyleOJ/Project_lemon

[3] Menci/Lemon

[4] Dust1404/Project_LemonPlus

[5] Project-LemonLime/Project_LemonLime

[6] 编译手册

3.4 命令行

Authors: StudyingFather, ayalhw, qinyihao, CoderOJ, mcendu

虽然图形界面能做的事情越来越多，但有很多高阶操作仍然需要使用命令行来解决。

本页面将简要介绍命令行的一些使用方法。

基础

Windows 自带的命令行界面有两个。「命令提示符」(`cmd`) 是其中较为古老的一个，功能也相对简单。PowerShell 是较新的一个命令行界面，自带的功能丰富，但相对臃肿。两个界面都可以在开始菜单中找到。

类 Unix 系统（包含 macOS 和 Linux，以下称为 Unix）分为有图形界面和无图形界面两种情况。如果系统有图形界面（如使用 macOS 或者在 Linux 下安装了 GNOME、KDE 等图形界面），则命令行一般可以通过名为「终端」(Terminal 或 Console) 的程序打开。没有图形界面的系统会在启动完成后自动进入命令行。

Windows 下的命令行长这样：

```
C:\Users\chtholly>
```

在命令行上输入的指令会显示在 `>` 以后。

```
C:\Users\chtholly>echo "Hello World!"
```

Unix 下的命令行长这样（以 Debian/Ubuntu 为例，其它系统的命令行大体类似）：

```
chtholly@seniorious:~$
```

在命令行上输入的指令会显示在 `$` 以后。

```
chtholly@seniorious:~$ echo "Hello World!"
```

如果在 Unix 下使用 `root` 登录命令行，那么 `$` 会被替换成 `#`：

```
root@seniorious:~# apt-get install gcc
```

命令行的 `>`，`$` 或 `#` 之前会显示一个路径，这个路径就是工作目录（working directory），或者当前目录。在 Unix 下当前目录有时会显示成类似 `~/folder` 的形式，最开头的 `~` 就是当前登录的用户的主目录。用户 `chtholly` 的主目录在不同系统下的位置是不同的；在 Linux 下，其主目录位于 `/home/chtholly`，而在 macOS 下，其主目录位于 `/Users/chtholly`。

语法和常用命令[1]

文件系统相关

先介绍文件系统里描述位置的两种方式，相对路径和绝对路径。

- 相对路径：用相对当前路径的位置关系来描述位置。例如当前路径为 `~/folder`，则 `./a.cpp` 实际上指的就是 `~/folder/a.cpp` 这个文件。**随着当前路径的变化，相对路径描述的位置也可能发生改变**。
- 绝对路径：用完整的路径来描述位置。例如 `~/folder/a.cpp` 就是一个绝对路径的例子。**绝对路径描述的位置不随当前路径的变化而改变**。

  Windows/Unix 用 `.` 代表当前目录，`..` 代表当前目录的父目录。特别地，在 Unix 下，用 `~` 表示用户主目录（注意 `~` 由 shell 展开，因此在其他地方可能不可用）。

在 Windows/Unix 下，使用 `pwd` 命令可以打印当前的目录，`cd < 目录 >` 命令可以切换当前的目录。例如，`cd folder` 会切换到当前目录的 `folder` 子目录；`cd ..` 会切换到当前目录的父目录。

在 Windows 下，使用 `dir` 命令可以列出当前目录的文件列表。在 Unix 下，列出文件列表的命令是 `ls`。特别的，在 PowerShell 下，可以使用与 Unix 相同的 `ls` 命令。

在 Windows 下，使用 `md < 目录 >` 或者 `mkdir < 目录 >` 命令创建一个新目录，使用 `rd < 目录 >` 命令删除一个目录。在 Unix 下，这两个命令分别是 `mkdir` 和 `rmdir`。需要注意的是，**使用 `rd` 或是 `rmdir` 删除一个目录前，这个目录必须是空的**。如果想要删除非空目录（和该目录下的所有文件）的话，Unix 下可以执行 `rm -r < 目录 >` 命令，Windows 下可以执行 `rd /s < 目录 >` 命令。

重定向机制

> 我编译了一个程序，它从标准输入读入，并输出到标准输出。然而输入文件和输出文件都很大，这时候能不能想办法把输入重定向到指定的输入文件，输出重定向到指定的输出文件呢？

使用如下命令即可实现。

```
$ command < input > output
```

例如，`./prog < 1.in > 1.out` 这个命令就将让 `prog` 这个程序从当前目录下的 `1.in` 中读入数据，并将程序输出覆盖写入到 `1.out`。

warning

`1.out` 原本的内容会被覆盖，如果想要在原输出文件末尾追加写入，请使用 `>>`，即 `./prog >> 1.out` 的方式做输出重定向

注意，PowerShell 只支持输出重定向，不支持输入重定向。

事实上，大多数 OJ 都采用了这样的重定向机制。选手提交的程序采用标准输入输出，通过重定向机制，就可以让选手的程序从给定的输入文件读入数据，输出到指定的输出文件，再进行文件比较就可以评测了。

执行程序

对于一个可执行程序或是批处理脚本，只需在命令行里直接输入它的文件名即可执行它。

当然，执行一个文件时，命令行并不会把所有目录下的文件都找一遍。环境变量 `PATH` 描述了命令行搜索路径的范围，命令行会在 `PATH` 中的路径寻找目标文件。

对于 Windows 系统，**当前目录也在命令行的默认搜索范围内**。例如 Windows 系统中，输入 `hello` 命令就可以执行当前目录下的 `hello.exe`。但是在 PowerShell 中，PowerShell 默认不会从当前目录寻找可执行文件（这与在 Unix 的行为一致），因而在 PowerShell 中需要使用相对路径或绝对路径调用当前目录下的可执行文件，例如 `.\hello.exe`，否则，你将看到以下报错：

```
PS> hello
hello: The term 'hello' is not recognized as a name of a cmdlet,
function, script file, or executable program.
Check the spelling of the name, or if a path was included, verify that
the path is correct and try again.

Suggestion [3,General]: The command hello was not found, but does exist
in the current location. PowerShell does not load commands from the
current location by default. If you trust this command, instead type:
".\hello". See "get-help about_Command_Precedence" for more details.
```

在 Unix 系统中，**当前目录并不在命令行的默认搜索范围内**，所以执行当前目录下的 `hello` 程序的命令就变成了 `./hello`：

```
$ hello
hello: command not found
$ ./hello
Hello World!
```

总结

上面介绍的用法只是命令行命令的一小部分，还有很多命令没有涉及到。在命令行里输入帮助命令 `help`，可以查询所有基本命令以及它们的用途。

下面给出 Windows 系统和 Unix 系统的命令对照表，以供参考。

分类	Windows 系统	Unix 系统
文件列表	`dir`	`ls`
切换目录	`cd`	`cd`
建立目录	`md`	`mkdir`
删除目录	`rd`	`rmdir`
比较文件	`fc`	`diff`
复制文件	`copy`	`cp`
移动文件	`move`	`mv`
文件改名	`ren`	`mv`
删除文件	`del`	`rm`

使用命令行编译/调试

命令行编译

手动编译

在命令行下输入 `g++ a.cpp` 就可以编译 `a.cpp` 这个文件了（Windows 系统需提前把编译器所在目录加入到 `PATH` 中）。

编译过程中可以加入一些编译选项：

- `-o <文件名>`：指定编译器输出可执行文件的文件名。

- `-g`：在编译时添加调试信息（使用 gdb 调试时需要）。
- `-Wall`：显示所有编译警告信息。
- `-O1`，`-O2`，`-O3`：对编译的程序进行优化，数字越大表示采用的优化手段越多（开启优化会影响使用 gdb 调试）。
- `-DDEBUG`：在编译时定义 `DEBUG` 符号（符号可以随意更换，例如 `-DONLINE_JUDGE` 定义了 `ONLINE_JUDGE` 符号）。
- `-UDEBUG`：在编译时取消定义 `DEBUG` 符号。
- `-lm`，`-lgmp`：链接某个库（此处是 math 和 gmp，具体使用的名字需查阅库文档，但一般与库名相同）。

> note
>
> 在 Unix 下，如使用了标准 C 库里的 math 库（`math.h`），则需在编译时添加 `-lm` 参数。[2]

使用 GNU Make 的内置规则[9]

对于名为 `qwq.c/cpp/p` 的 C，C++，Pascal 程序源代码，可以使用 `make qwq` 自动编译成对应名为 `qwq` 的程序。

如需添加额外的编译选项，请使用 `export CFLAGS="xxx"` 或者 `export CPPFLAGS="xxx"` 定义。

Sanitizers

介绍

`sanitizers` 是一种集成于编译器中，用于调试 `C/C++` 代码的工具，通过在编译过程中插入检查代码来检查代码运行时出现的内存访问越界、未定义行为等错误。

它分为以下几种：

- AddressSanitizer[5]：检测对堆、栈、全局变量的越界访问，无效的释放内存、内存泄漏（实验性）。
- ThreadSanitizer[6]：检测多线程的数据竞争。
- MemorySanitizer[7]：检测对未初始化内存的读取。
- UndefinedBehaviorSanitizer[8]：检测未定义行为。

使用方式

最新版本的 `clang++`、`g++` 以及 `MSVC`（部分支持）均已内置 `sanitizers`，但功能和使用方法有所不同，这里以 `clang++` 为例，它的使用方法如下：

```
$ clang++ -fsanitize=<name> test.cc
```

其中 `<name>` 即为要启用的功能（一个 `sanitizer` 可理解为一些功能的集合），例如：

```
$ clang++ -fsanitize=memory test.cc # 启用 MemorySanitizer
$ clang++ -fsanitize=signed-integer-overflow test.cc # 启用有符号整型溢出检测
```

之后直接像平常一样运行可执行文件即可，如果 sanitizer 检测到错误，则会输出到 `stderr` 流，例如：

```
$ ./a.out
test.cc:3:5: runtime error: signed integer overflow: 2147483647 + 1 cannot be re
presented in type 'int'
```

时间/内存代价

显而易见，这些调试工具会严重拖慢代码的运行时间和增大所用内存，以下为使用它们的时间/内存代价：

名称	所增大内存倍数	所增大时间倍数
AddressSanitizer	N/A	2
ThreadSanitizer	5~15	5~10
MemorySanitizer	N/A	3
UndefinedBehaviorSanitizer	N/A	N/A

命令行调试

在命令行下，最常用的调试工具是 gdb。

执行 `gdb a` 就可以调试 `a` 程序。

以下是几个 gdb 调试的常用命令（大多数命令可以缩写，用命令开头的若干个字母就可以代表该命令）：

- `list`（`l`）：列出程序源代码，如 `l main` 指定列出 `main` 函数附近的若干行代码。
- `break`（`b`）：设置断点，如 `b main` 表示在 `main` 函数处设置断点。
- `run`（`r`）：运行程序直到程序结束运行或遇到断点。
- `continue`（`c`）：在程序遇到断点后继续执行，直到程序结束运行或到达下一个断点。
- `next`（`n`）：执行当前行语句，如果当前行有函数调用，则将其视为一个整体执行。
- `step`（`s`）：执行当前行语句，如果当前行有函数调用，则进入该函数内部。
- `finish`（`fin`）：继续执行至当前函数返回。
- `call`：调用某个函数，例如：`call f(2)`（以参数 2 调用函数 f）。
- `quit`（`q`）：退出 gdb。
- `display`（`disp`）：指定程序暂停时显示的表达式。
- `print`（`p`）：打印表达式的值。

  `display` 和 `print` 指令都支持控制输出格式，其方法是在命令后紧跟 `/` 与格式字符，例如 `p/d test`（按照十进制打印变量 test 的值），支持的格式字符有：

格式字符	对应格式
d	按十进制格式显示变量
x	按十六进制格式显示变量
a	按十六进制格式显示变量
t	按二进制格式显示变量
c	按字符格式显示变量
f	按浮点数格式显示变量
u	按十进制格式显示无符号整型
o	按八进制格式显示变量

命令行使用技巧

自动补全

补全是 Shell 提供的基本功能之一，主要用于减少命令行使用中的输入量和 typo 概率。

一般情况下，使用补全的快捷键一般是 Tab，按下后 Shell 会根据已输入的字符补全信息。

不同的 Shell 提供了能力不尽相同的补全能力。

以下是常见 Shell 的补全能力[3]：

Shell	补全能力（补全范围）
cmd（Windows 的传统控制台）	文件路径
PowerShell	文件路径、PATH 中的命令名、内建命令名、函数名、命令参数，支持模糊匹配，自动纠错
Bash	文件路径、PATH 中的命令名、内建命令名、函数名、命令参数
Zsh	文件路径、PATH 中的命令名、内建命令名、函数名、命令参数，支持模糊匹配，自动纠错和建议
Fish	文件路径、PATH 中的命令名、内建命令名、函数名、命令参数，支持模糊匹配，补全时可显示参数功能，自动纠错和建议

note

PowerShell 的部分功能需要 PSReadline Module 载入或者位于 PowerShell ISE 中。

Bash 的补全功能一般需要一个名为 `bash-completions` 的包才能获得完整功能，部分软件的补全文件由软件包自带。

Zsh 完整的补全功能需要配合用户预定义的文件（一般随 Zsh 包或对应软件包安装）。

Fish 在默认配置下提供良好完整的补全功能，但仍有部分官方未覆盖到的软件的补全文件由软件自行提供。

帮助文档

一般来说，命令行下的程序都附有「帮助」，Windows 下一般使用 `command /?` 或者 `command -?` 获取，Unix-like（例如 Linux）上一般使用 `command --help` 或者 `command -h` 获取（但是 BSD 下的「帮助」往往过分简略而难以使用）。

此外，在 Unix-like 系统上，还有可通过 `man command` 获取的「手册」(manual)，相比「帮助」一般更为详细。

built-in time 和 GNU time

测试程序运行时间时，我们通常可以使用 `time` 命令。

但是这个命令实际上在系统中有两个对应的命令：一个是部分 Shell（例如 Bash）内建的命令，一个是 GNU time（是一个单独的软件）。这两个之间存在一些差异。

一般在 Bash 中直接使用 `time` 调用的是 Bash 内建的版本，我们可以使用 `TIMEFORMAT` 环境变量控制其输出格式，例如将其设为 `%3lR` 即可输出三位精度的实际运行时间，`%3lU` 即可输出三位精度的用户空间运行时间。[4]

如果想要调用 GNU 版本的 time，则需使用 `\time` 或者 `/usr/bin/time` 调用，但是它的输出格式并不易读，我们可以附加 `-p` 参数（即为 `\time -p`）来获得易读的输出。

管道

假设我们现在有两个程序 A 和 B，都用标准输入输出，如何让 A 的输出重定向到 B 的输入？

我们可以使用上文中提到的重定向的方式，先把 A 的输出重定向到一个临时文件，在把 B 的输入重定向到这个临时文件上。

但这个方法很低效，不仅需要创建新的文件，磁盘 IO 的操作也可能成为瓶颈，而且两个程序不能同时运行，必须等 A 跑完了才能开始跑 B。有没有更好的方法？

有，那就是**管道**，使用起来也非常简单，如下操作即可：

```
$ A | B
```

这会在内存创建一个管道，然后两个程序被同时启动。程序 A 每次要输出被重定向到这个管道中，而这个管道本身不会存储数据（其实有一个很小的缓冲区）。在 B 读取之前，A 的输出操作会被阻塞，等到 B 把数据读入以后，A 的输出才能继续进行。这样优美地解决了上述的问题，没有磁盘 IO 操作，两份代码同时运行，也没有额外消耗很多的内存储存中间结果。

命名管道

有时候我们不只是要把一个程序的输出重定向到另一个的输入。比如在做 IO 交互题的时候，经常需要将 A 的输出重定向到 B 的输入，B 的输出重定向到 A 的输出，这个时候用上文提到的普通管道就无能为力了。而重定向到文件，有无法让两个程序同时运行。这个时候就需要一个长得像文件的管道——命名管道。

在 Unix 系统中，可以使用如下命令创建命名管道（以命名为 `my_pipe` 举例）：

```
$ mkfifo my_pipe
```

这个时候使用 `ls` 命令列出当前目录下的文件，会发现多了一个 `my_pipe|` 的文件。这就创建了一个命名管道，文件名后的 `|` 代表这是一个管道文件。然后就可以像文件的重定向一样向这个管道中读写了。

通过命名管道，我们可以通过这样的方式让两个程序交互：

```
$ mkfifo input output
$ ./checker > input < output # 这里一定要把 > input 写在前面，不然 shell 会先打开
 output 管道，而这个管道现在并没有东西，会阻塞 checker 的运行。
$ ./my_code < input > output
```

使用完后，可以像普通文件一样用 `rm` 命令删除命名管道。

参考资料与注释

[1] 刘汝佳《算法竞赛入门经典（第 2 版）》附录 A 开发环境与方法

[2] Why do you have to link the math library in C?

[3] Comparison_of_command_shells#Interactive_features

[4] https://unix.stackexchange.com/a/70655

[5] https://clang.llvm.org/docs/AddressSanitizer.html

[6] https://clang.llvm.org/docs/ThreadSanitizer.html

[7] https://clang.llvm.org/docs/MemorySanitizer.html

[8] https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[9] Catalogue of Built-In Rules

3.5 编译器

本页面主要介绍了各系统下各类编译器/解释器的安装步骤。

GCC

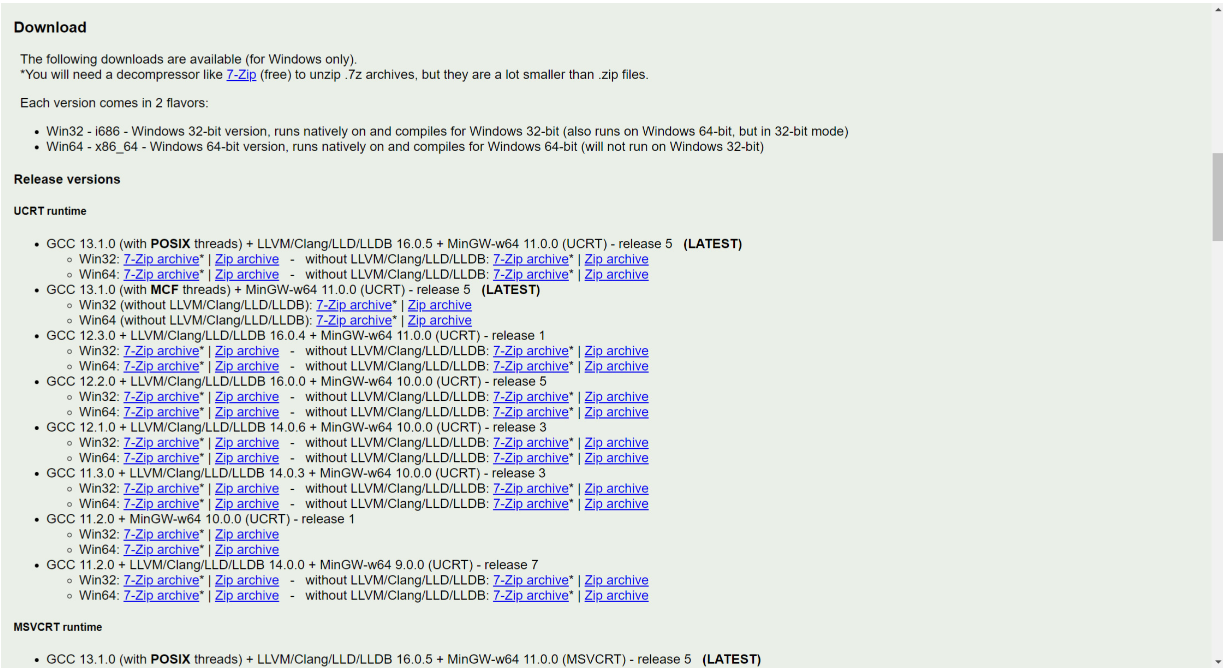
Windows

手动下载安装

访问 MinGW-w64[3] 的下载页面，有多个构建版本。方便起见，我们使用由 WinLibs 提供的构建版本。

首先前往 WinLibs[4] 下载最新的安装包，选择合适的版本，本文选择了 GCC 12.3.0 + LLVM/Clang/LLD/LLDB 16.0.4 + MinGW-w64 11.0.0 (UCRT)：

默认会附带安装 LLVM Clang，如果不想安装，你也可以选择右边的 without LLVM/Clang/LLD/LLDB。


Download

The following downloads are available (for Windows only).
*You will need a decompressor like 7-Zip (free) to unzip .7z archives, but they are a lot smaller than .zip files.

Each version comes in 2 flavors:

- Win32 - i686 - Windows 32-bit version, runs natively on and compiles for Windows 32-bit (also runs on Windows 64-bit, but in 32-bit mode)
- Win64 - x86_64 - Windows 64-bit version, runs natively on and compiles for Windows 64-bit (will not run on Windows 32-bit)

Release versions

UCRT runtime

- GCC 13.1.0 (with **POSIX** threads) + LLVM/Clang/LLD/LLDB 16.0.5 + MinGW-w64 11.0.0 (UCRT) - release 5 **(LATEST)**
  - Win32: 7-Zip archive* | Zip archive - without LLVM/Clang/LLD/LLDB: 7-Zip archive* | Zip archive
  - Win64: 7-Zip archive* | Zip archive - without LLVM/Clang/LLD/LLDB: 7-Zip archive* | Zip archive
- GCC 13.1.0 (with **MCF** threads) + MinGW-w64 11.0.0 (UCRT) - release 5 **(LATEST)**
  - Win32 (without LLVM/Clang/LLD/LLDB): 7-Zip archive* | Zip archive
  - Win64 (without LLVM/Clang/LLD/LLDB): 7-Zip archive* | Zip archive
- GCC 12.3.0 + LLVM/Clang/LLD/LLDB 16.0.4 + MinGW-w64 11.0.0 (UCRT) - release 1
  - Win32: 7-Zip archive* | Zip archive - without LLVM/Clang/LLD/LLDB: 7-Zip archive* | Zip archive
  - Win64: 7-Zip archive* | Zip archive - without LLVM/Clang/LLD/LLDB: 7-Zip archive* | Zip archive
- GCC 12.2.0 + LLVM/Clang/LLD/LLDB 16.0.0 + MinGW-w64 10.0.0 (UCRT) - release 5
  - Win32: 7-Zip archive* | Zip archive - without LLVM/Clang/LLD/LLDB: 7-Zip archive* | Zip archive
  - Win64: 7-Zip archive* | Zip archive - without LLVM/Clang/LLD/LLDB: 7-Zip archive* | Zip archive
- GCC 12.1.0 + LLVM/Clang/LLD/LLDB 14.0.6 + MinGW-w64 10.0.0 (UCRT) - release 3
  - Win32: 7-Zip archive* | Zip archive - without LLVM/Clang/LLD/LLDB: 7-Zip archive* | Zip archive
  - Win64: 7-Zip archive* | Zip archive - without LLVM/Clang/LLD/LLDB: 7-Zip archive* | Zip archive
- GCC 11.3.0 + LLVM/Clang/LLD/LLDB 14.0.3 + MinGW-w64 10.0.0 (UCRT) - release 3
  - Win32: 7-Zip archive* | Zip archive - without LLVM/Clang/LLD/LLDB: 7-Zip archive* | Zip archive
  - Win64: 7-Zip archive* | Zip archive - without LLVM/Clang/LLD/LLDB: 7-Zip archive* | Zip archive
- GCC 11.2.0 + MinGW-w64 10.0.0 (UCRT) - release 1
  - Win32: 7-Zip archive* | Zip archive
  - Win64: 7-Zip archive* | Zip archive
- GCC 11.2.0 + LLVM/Clang/LLD/LLDB 14.0.0 + MinGW-w64 9.0.0 (UCRT) - release 7
  - Win32: 7-Zip archive* | Zip archive - without LLVM/Clang/LLD/LLDB: 7-Zip archive* | Zip archive
  - Win64: 7-Zip archive* | Zip archive - without LLVM/Clang/LLD/LLDB: 7-Zip archive* | Zip archive

MSVCRT runtime

- GCC 13.1.0 (with **POSIX** threads) + LLVM/Clang/LLD/LLDB 16.0.5 + MinGW-w64 11.0.0 (MSVCRT) - release 5 **(LATEST)**

图 3.64

下载好后将其解压到电脑中的某个位置，教程中将其解压到了 C 盘的根目录。目录名中最好不要包含非英文字符和空格，否则可能会在后期导致一些问题。

接下来我们需要将编译器的可执行文件目录添加到系统环境变量中，这样在编译时就不需要指定编译器的路径了，方便使用。上方我们将 MinGW 解压到了 `C:\mingw64` 目录中，那么可执行文件所在的目录就是 `C:\mingw64\bin`。

按下 Windows 徽标 + R 组合键，输入 `rundll32.exe sysdm.cpl,EditEnvironmentVariables`，打开系统环境变量设置窗口，并在「系统变量」一节中选中名为「Path」的变量，然后点击「编辑」按钮：

在编辑窗口中点击右侧的「新建」按钮，为「Path」变量新建一个条目，并填入上文中记录下的可执行文件所在的目录（教程中为 `C:\mingw64\bin`）。

" 对部分老版本系统的提示"

部分老版本系统只能手动修改变量的文本值，那么需要在变量的值的末尾插入一个**半角分号**，再将可执行文

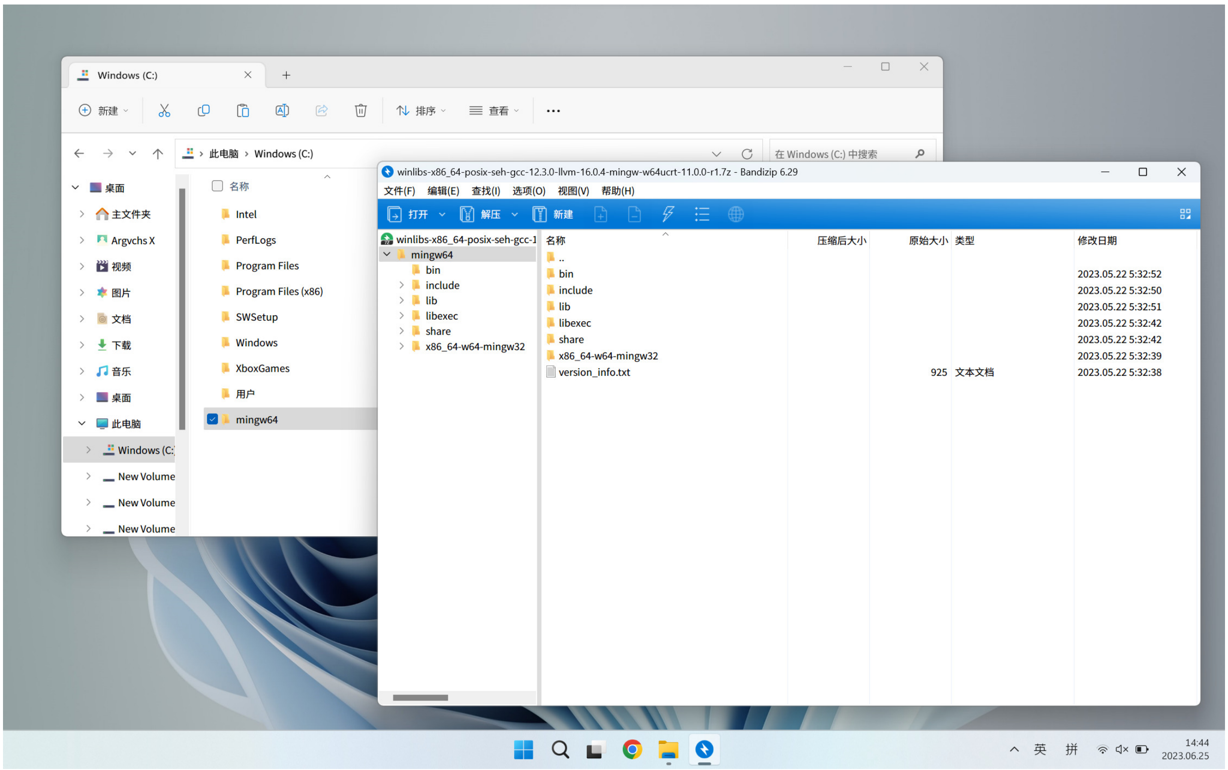

Windows (C:)
新建
排序
查看
此电脑
Windows (C:)
在 Windows (C:) 中搜索
桌面
主文件夹
视频
图片
文档
下载
音乐
此电脑
New Volume
名称
Intel
PerfLogs
Program Files
Program Files (x86)
SWSetup
Windows
XboxGames
用户
mingw64
winlibs-x86_64-posix-seh-gcc-12.3.0-llvm-16.0.4-mingw-w64ucrt-11.0.0-r1.7z - Bandizip 6.29
文件(F) 编辑(E) 查找(I) 选项(O) 视图(V) 帮助(H)
打开
解压
新建
bin
include
lib
libexec
share
x86_64-w64-mingw32
version_info.txt
压缩后大小
原始大小
类型
修改日期
925
文本文档
2023.05.22 5:32:52
2023.05.22 5:32:50
2023.05.22 5:32:51
2023.05.22 5:32:42
2023.05.22 5:32:42
2023.05.22 5:32:39
2023.05.22 5:32:38
14:44
2023.06.25

图 3.65

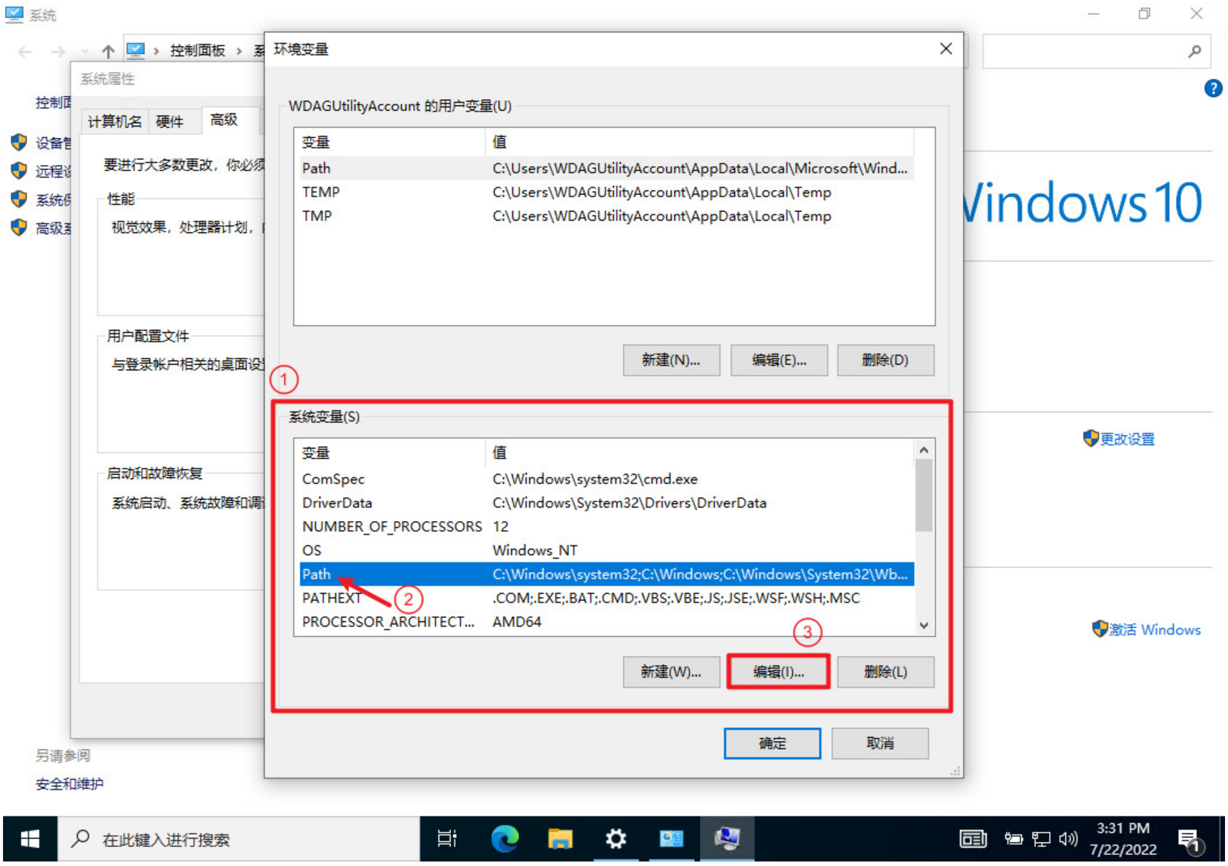

系统
控制面板
系统属性
计算机名
硬件
高级
要进行大多数更改，你必须
性能
视觉效果，处理器计划，
用户配置文件
与登录帐户相关的桌面设
启动和故障恢复
系统启动、系统故障和调
另请参阅
安全和维护
环境变量
WDAGUtilityAccount 的用户变量(U)
变量
值
Path
C:\Users\WDAGUtilityAccount\AppData\Local\Microsoft\Wind...
TEMP
C:\Users\WDAGUtilityAccount\AppData\Local\Temp
TMP
C:\Users\WDAGUtilityAccount\AppData\Local\Temp
新建(N)...
编辑(E)...
删除(D)
1
系统变量(S)
ComSpec
C:\Windows\system32\cmd.exe
DriverData
C:\Windows\System32\Drivers\DriverData
NUMBER_OF_PROCESSORS
12
OS
Windows_NT
Path
C:\Windows\system32;C:\Windows;C:\Windows\System32\Wb...
PATHEXT
2
.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
PROCESSOR_ARCHITECT...
AMD64
3
新建(W)...
编辑(I)...
删除(L)
确定
取消
Windows 10
更改设置
激活 Windows
在此键入进行搜索
3:31 PM
7/22/2022

图 3.66

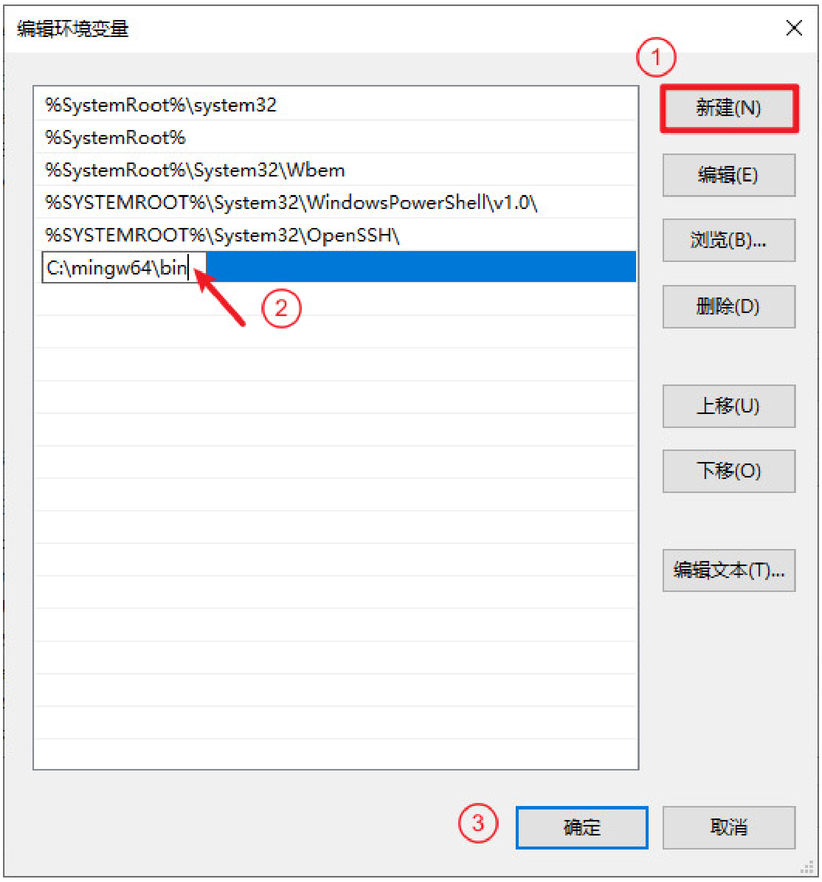


图 3.67

件所在的目录粘贴到这个半角分号的后面，如图所示：

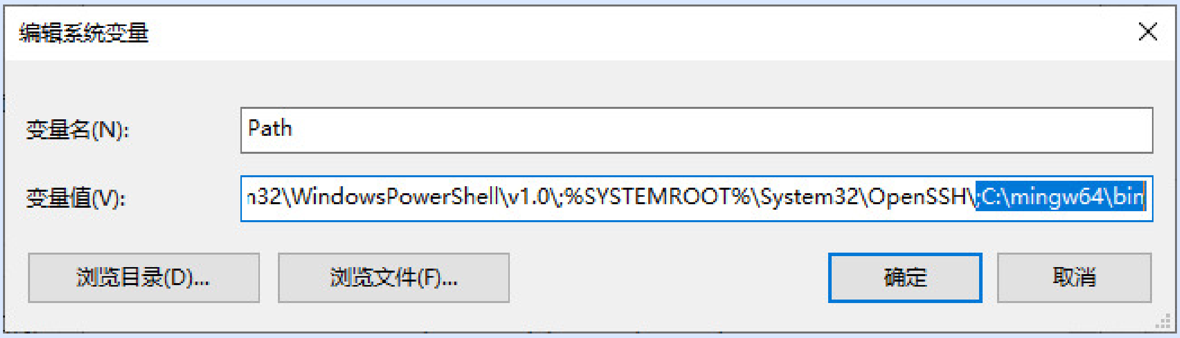


图 3.68

完成后一路点击「确定」按钮退出即可。

接下来打开终端，输入 `g++ --version` 并按下回车，如果出现如图所示的提示则代表安装成功。

Scoop 安装

打开 PowerShell，运行以下脚本：

```
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
irm get.scoop.sh | iex
scoop install mingw-winlibs
```

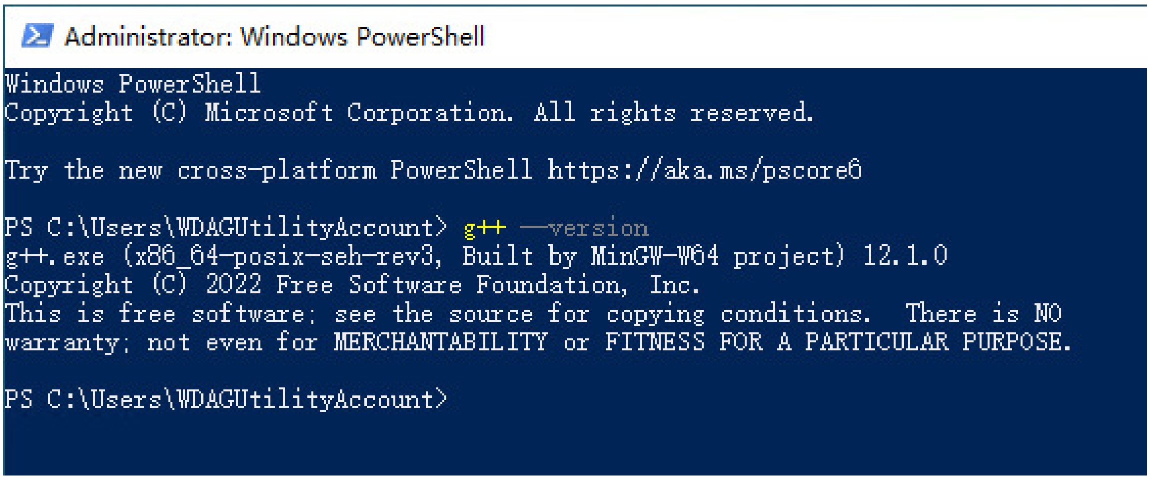


图 3.69

Linux

Debian/Ubuntu

首先先更新软件包列表：

```
sudo apt update
```

再使用命令直接安装即可：

```
sudo apt install g++
```

Arch Linux

使用命令直接安装即可：

```
sudo pacman -Syu gcc
```

openSUSE

使用命令直接安装即可：

```
sudo zypper in gcc-c++
```

macOS

首先更新包管理器：

```
brew upgrade
brew update
```

再使用命令直接安装即可：

```
brew install gcc
```

JDK

JDK 的发行版有很多，以下介绍两种：

- OpenJDK 中的 Eclipse Temurin[5][1]：参见 Install Eclipse Temurin™ | Adoptium[6]。
- Oracle JDK：可参见 JDK Installation Guide（JDK 17）[7]。

Python 3

Python 的实现也有很多[2]，以 CPython 3 为例，参见 Download Python | Python.org[8]。

LLVM

Windows

"LLVM 在 Windows 上的坑"

由于 LLVM 在 Windows 上缺失标准库，所以你仍需安装 MSVC 或 GCC。

直接安装

访问 LLVM[9] 的下载页面，选择 LLVM-*-win64.exe 下载。

如果你的网络质量不佳，你也可以选择访问 清华大学开源软件镜像站[10] 进行下载。

打开.exe 文件，安装时勾选 Add LLVM to system PATH for current user，随后一直点击下一步即可安装完成。

打开终端，输入 `clang++ --version` 并回车，出现

```
clang version 15.0.1
Target: x86_64-pc-windows-msvc
Thread model: posix
InstalledDir: <omitted>
```

类似物即代表成功。

Scoop 安装

打开 PowerShell，运行以下脚本：

```
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
irm get.scoop.sh | iex
scoop install llvm
```

Linux

openSUSE

使用命令直接安装即可：

```
sudo zypper in llvm clang
```

MSVC（Visual Studio）

访问 Visual Studio[11] 的下载页面，将光标移动到下载 Visual Studio，在下来菜单中点击 Community 2022 下载。

打开安装器，选择 Community 2022 安装。

在随后弹出来的窗口中仅选择使用 C++ 的桌面开发，然后单击安装。

[1] Eclipse Temurin 即为原 AdoptOpenJDK，后者已于 2021 年 7 月移交至 Eclipse 基金会。具体可见 本声明。

[2] Alternative Python Implementations | Python.org

[3] MinGW-w64

[4] WinLibs

[5] Eclipse Temurin

[6] Install Eclipse Temurin™ | Adoptium

[7] JDK Installation Guide（JDK 17）

[8] Download Python | Python.org

[9] LLVM

[10] 清华大学开源软件镜像站

[11] Visual Studio

3.6 WSL（Windows 10）

> **Authors:** GoodCoder666, Ir1d, H-J-Granger, NachtgeistW, StudyingFather, Enter-tainer, abc1763613206, Anti-Li, shenyouran, Chrogeek, SukkaW, Henry-ZHR, Early0v0, andylizf, tootal, Marcythm, CoelacanthusHex, indevn, qinyihao, peasoft

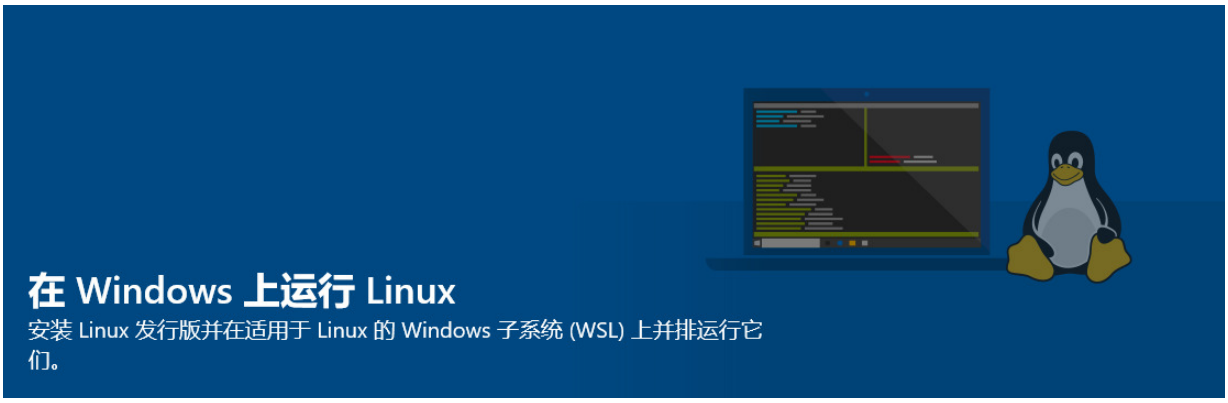


图 3.70 头图

本章主要介绍了在 Windows 系统下使用 Windows Subsystem for Linux 运行 Linux 环境的方法。

引言[1]

现在大部分学校的竞赛练习环境都是构建在 Windows 系操作系统上，但是在 NOI 系列赛中，已经用上了 NOI Linux 这个 Ubuntu 操作系统的修改版。

NOI 竞赛（自 2021 年 9 月 1 日）的环境要求如下。[2]

类别	软件或模块	版本	备注说明
系统	Linux 内核	`5.4.0-42-generic`	64 位 x86 (AMD64)
语言环境	GCC（`gcc` 和 `g++`）	`9.3.0`	C 和 C++ 编译器
	FPC	`3.0.4`	Pascal 编译器（注：自 2022 年起，NOI 系列竞赛不再支持 Pascal 语言）
	Python 2	`2.7`	非竞赛语言
	Python 3	`3.8`	非竞赛语言
调试工具	GDB	`9.1`	
	DDD	`3.3.12`	GDB 的 GUI 前端
集成开发环境（IDE）	Code::Blocks	`20.03`	C/C++ IDE
	Lazarus	`2.0.6`	Pascal IDE
	Geany	`1.36`	C/C++/Pascal（轻量级）IDE
文本编辑工具	Visual Studio Code	`1.54.3`	
	GNU Emacs	`26.3`	
	gedit	`3.36.2`	
	Vim	`8.1`	
	Joe	`4.6`	
	nano	`4.8`	
	Sublime Text	`3.2.2`	
其它软件	Firefox	`79.0`	浏览器
	Midnight Commander (`mc`)	`4.8.24`	文件管理器
	xterm (uxterm)	`3.5.3`	终端
	Arbiter-local	`1.02`	程序评测工具单机版

考场环境与一般环境会有一系列差异：

- 命令行上的操作和图形界面上的操作会有差异。
- Linux 和 Windows 的差异，如对于大小写的敏感性差异。
- 不同编译器的行为（MSVC 和 GCC）和不同版本的编译器（Windows 上和 Linux 上的 GCC，32 位和 64 位的 Linux GCC，GCC 7 和 GCC 8 等）的行为，如对变量初始化和对数组下标越界的处理会有差异。
- 不同评测系统（洛谷和 Arbiter）的超时检查和内存限制检查会有差异。

这有可能导致一系列的尴尬情况：

- 想用 Ctrl+C 复制，结果退出了程序。
- 平时 AC 的程序模板到了 Linux 上就 WA。

为了防止考场上出现此类尴尬情况，我们必须要提前熟悉 Linux 系统的操作方法。

虽然 NOI 的官网已经放出了 NOI Linux 的 ISO 镜像，虚拟机的配置较为麻烦。且由于 NOI Linux 默认自带图形界面，无法保证在低配系统上流畅运行。

Windows 10 在一周年更新时推出了 Linux 子系统（WSL），在 2020 年 5 月更新中升级到了 WSL 2。截至 2020 年 6 月 1 日，WSL 已支持安装 Ubuntu、openSUSE Leap、Kali、Debian 等主流 Linux 分发版。但 WSL 并不支持 NOI 评测用的 Arbiter。

"什么是 Linux 子系统（WSL）[8]"

适用于 Linux 的 Windows 子系统（英语：Windows Subsystem for Linux，简称 WSL）是一个为在 Windows 10、Windows 11 与 Windows Server 2019 上能够原生运行 Linux 二进制可执行文件（ELF 格式）的兼容层。

WSL 可让开发人员按原样运行 GNU/Linux 环境 - 包括大多数命令行工具、实用工具和应用程序 - 且不会产生虚拟机开销。

WSL 仅在 64 位 Windows 10 版本 1607 及以上、Windows 11 和 Windows Server 2019/2022 中可用。

启用 WSL[3]

自动安装

warning

本部分适用于 Windows 10 版本 2004 及更高版本（内部版本 19041 及更高版本）或 Windows 11。

如果你正在使用 2004 以下版本或你的电脑不支持虚拟化，请阅读下面的手动安装一节。

如果你正在使用 1607 以下版本的 Windows 10，你的系统不支持 WSL。

1. 以管理员身份打开 Windows PowerShell（右击「开始」按钮，选择 Windows PowerShell（管理员）或 Windows 终端（管理员））
2. 输入 `wsl --install`，并等待所有组件自动安装完成。期间你可能需要重启你的计算机来启用必要的 Windows 功能。
3. 安装完成后，你可以在「开始」菜单或 Windows 终端的标签页中找到你安装的发行版。
4. 接下来，请转到下面「配置分发版」一节完成其他设置。

手动安装[4]

warning

下面介绍手动安装 WSL 的步骤。如果你已经完成了自动安装，请跳过此部分。

启用适用于 Linux 的 Windows 子系统

在安装适用于 WSL 的任何 Linux 分发版之前，必须在下述两种方法中选择一种，以确保启用「适用于 Linux 的 Windows 子系统」可选功能：

使用命令行：

1. 以管理员身份打开 PowerShell 并运行：

```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-L
inux
# 如果你只想要使用 WSL 1 请跳过此步骤
Enable-WindowsOptionalFeature -Online -FeatureName VirtualMachinePlatform
```

2. 出现提示时，重启计算机。

使用图形界面：

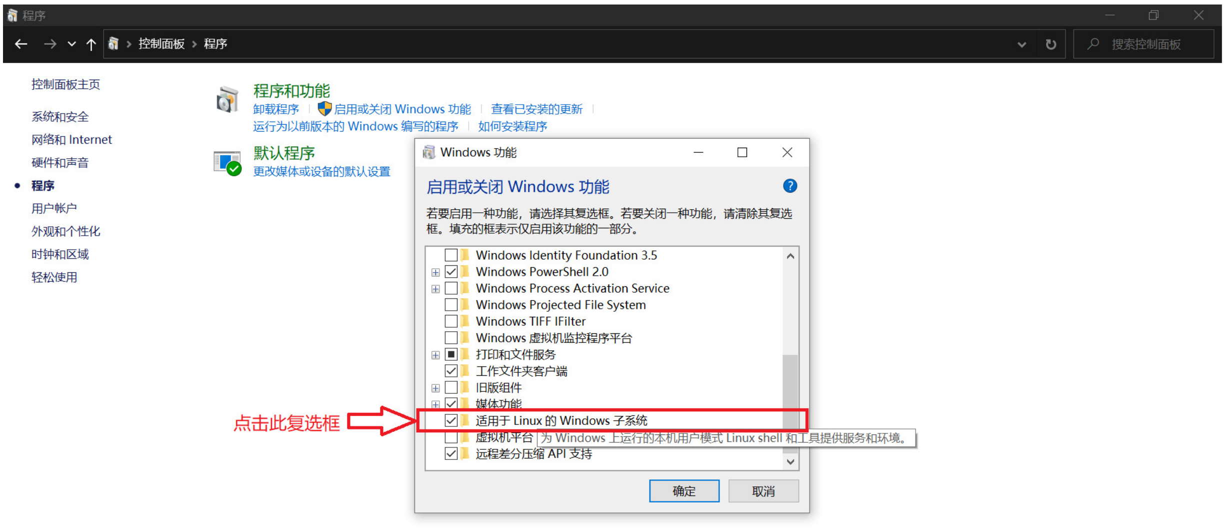


图 3.71 Windows 功能

1. 打开「控制面板」
2. 访问「程序和功能」子菜单「打开或关闭 Windows 功能」
3. 选择「适用于 Linux 的 Windows 子系统」与「虚拟机平台」
4. 点击确定
5. 重启

安装内核更新包

如果你想要使用 WSL 1, 请跳过此步骤。

下载 适用于 x64 计算机的 WSL2 Linux 内核更新包[9] 并安装。

设置 WSL 默认版本

绝大部分情况下，建议使用 WSL 2。WSL 1 与 WSL 2 的区别，请见 比较 WSL 2 和 WSL 1[10]

" 关于 systemd"

WSL 1 完全不支持 systemd（这意味着一些需要 systemd 的功能无法实现或需要其他替代方案）。WSL 2 已经内建对 systemd 的支持。如果需要使用 systemd，而当前运行的发行版没有配置为启用 systemd，可参考 WSL 中的高级设置配置[11]。

```
# 将 WSL 默认版本设置为 WSL 2
wsl --set-default-version 2
```

安装 WSL 分发版

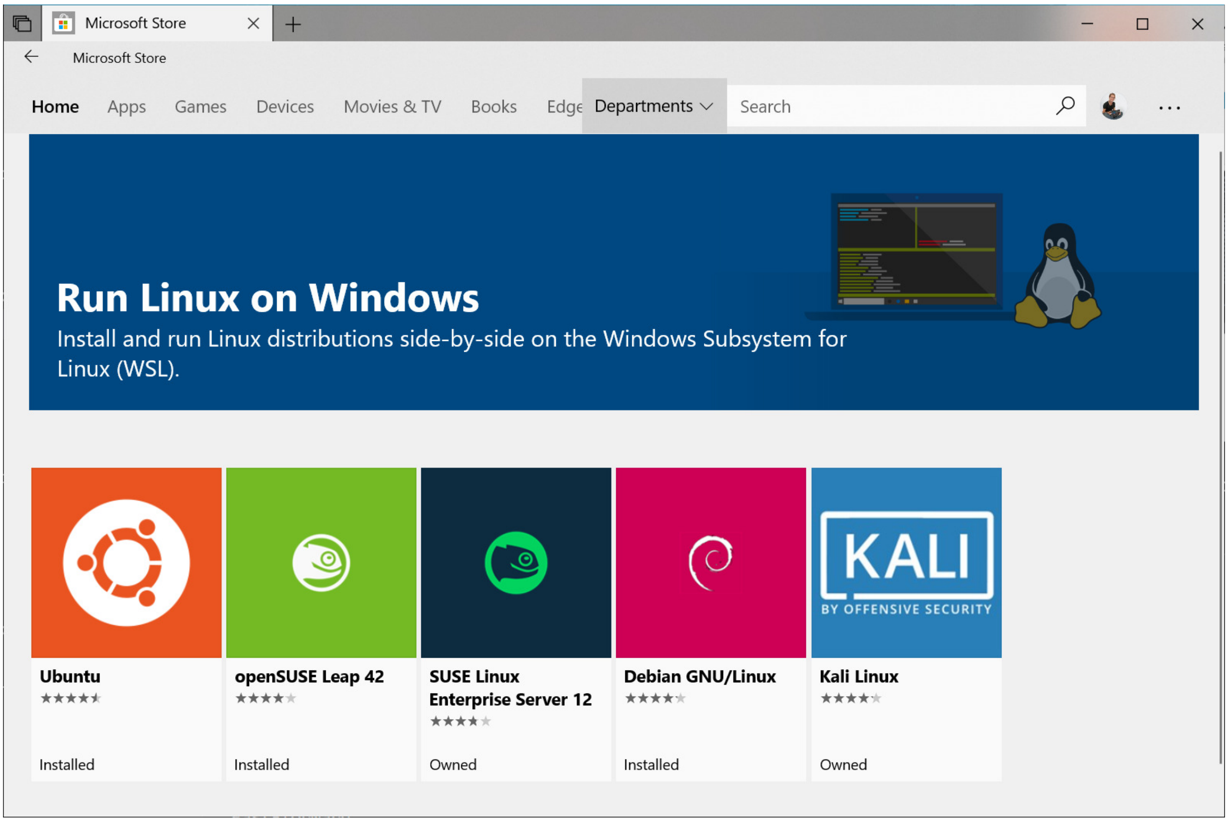


图 3.72 搜索页

进入 Microsoft Store，搜索「Ubuntu」，然后选择「Ubuntu」，点击「安装」进行安装。也可打开 Ubuntu 的商店页面[12]。

> warning
>
> Microsoft Store 的 Ubuntu 随着 Ubuntu 的更新而更新，因此内容可能会有所改变。如果想获取稳定的 Ubuntu 长期支持版，可以在 Microsoft Store 安装 Ubuntu 的 LTS 版本。

配置分发版[5]

本章以 Windows 自动安装的 Ubuntu 为例。

运行 Ubuntu

打开「开始」菜单找到 Ubuntu 并启动，或使用 `wsl` 命令从 Windows 命令行启动。

可以为 Ubuntu 创建应用程序磁贴或固定至任务栏，以在下次方便地打开。

初始化

第一次运行 Ubuntu，需要完成初始化。

```
Installing, this may take a few minutes...
```

等待一两分钟时间，系统会提示创建新的用户帐户。

```
Please create a default UNIX user account. The username does not need to match y
our Windows username.
For more information visit: https://aka.ms/wslusers
Enter new UNIX username: chtholly
```

输入完用户名以后会提示输入密码。在 Linux 中，输入密码时屏幕上不显示文字属于正常现象。

```
Enter new UNIX password:
```

设置好帐户名和密码后，WSL 就安装完成了。

```
Installation successful!
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

chtholly@SENIORIOUS:~$
```

基础配置

初次安装好的系统不附带任何 C/C++ 编译器，需要手动配置环境。

```
$ gcc
The program 'gcc' is currently not installed. You can install it by typing:
sudo apt install gcc
$ g++
The program 'g++' is currently not installed. You can install it by typing:
sudo apt install g++
```

更换为国内软件源

Ubuntu 默认的软件源在国外。可以换成国内的软件源以加快速度，如 清华 TUNA 的软件源[13]。

" 使用与自己系统版本匹配的软件源"

请在页面中寻找与自己系统版本相配的源（可使用 `sudo lsb_release -a` 查看 Ubuntu 版本）。

除非你知道你在做什么，否则不要使用与自己的系统版本不匹配的源！

使用以下命令更新软件和软件源：

```
$ sudo su # 执行这行指令后，终端提示符会从 $ 变成 #，执行下文的命令前注意关注提示符
[sudo] xxx 的密码:
# cp /etc/apt/sources.list /etc/apt/sources.list.bak
# vim /etc/apt/sources.list
...（按 i 之后将上文的源右键粘贴进去，编辑完后按 Esc，再输入 :wq 和回车）
# apt update
# apt upgrade -y
# exit
exit
$
```

安装中文环境

```
# apt install language-pack-zh-hans -y
# apt install fontconfig -y
# apt install fonts-noto-cjk fonts-wqy-microhei fonts-wqy-zenhei -y # 中文字体
# dpkg-reconfigure locales
```

此时会进入一个设置菜单，不用管，直接回车。

下一个菜单中选择 zh_CN.UTF-8 回车。

```
Default locale for the system environment:

             None
             C.UTF-8
             en_US.UTF-8
            [zh_CN.UTF-8]

        <Ok>            <Cancel>
```

之后关闭 WSL 并重启，系统就会变成中文。

再依次输入下列命令，把 man 帮助页替换为中文。[6]

```
# apt install manpages-zh
# sed -i 's|/usr/share/man|/usr/share/man/zh_CN|g' /etc/manpath.config
```

可以用 man help 测试。

安装编译环境[7]

```
# apt install -y build-essential vim ddd gdb fpc emacs gedit anjuta lazarus
```

GUIDE 的安装请参考 Debian 或 Ubuntu 下 GUIDE 的安装。

这里安装的是基础 + NOI 官方要求的环境，如有需要可以用 sudo apt install < 程序名 > 来安装其它软件包。若想安装其他版本可以参考 Debian 官方的 包管理手册[14]。

以下为一个示例程序：

```
$ vim cpuid.cpp
...
$ g++ -Wall cpuid.cpp -o cpuid
$ ./cpuid
AMD Ryzen 5 1400 Quad-Core Processor
```

note

Linux 环境下可执行文件可不带扩展名，运行方式参见上方命令。

进阶操作

使用 WSLg 运行 GUI 程序

如果你使用 Windows 10 19044 及以上版本或 Windows 11，则可以使用 WSL 2 提供的集成的桌面体验。该功能允许你直接安装并启动 Linux 桌面程序而无须其他配置。

参见 在适用于 Linux 的 Windows 子系统上运行 Linux GUI 应用[15]

安装图形环境，并使用远程桌面连接

如果你使用的版本尚不支持 WSLg, 可以尝试使用以下指南开启图形界面功能。

以下以 Xfce 为例。

如果只想安装 Xfce，可以执行以下命令：

```
$ sudo apt install xfce4 tightvncserver -y
```

如果除 Xfce 外想要更多的软件，可以执行以下命令：

```
$ sudo apt install xubuntu-desktop -y
```

图形环境文件较大，下载解包需要一定时间。

配置 xrdp：

```
$ sudo apt install xrdp -y
$ echo "xfce4-session" >~/.xsession
$ sudo service xrdp restart
```

为了防止和计算机原有的远程桌面冲突，需要更换默认端口。

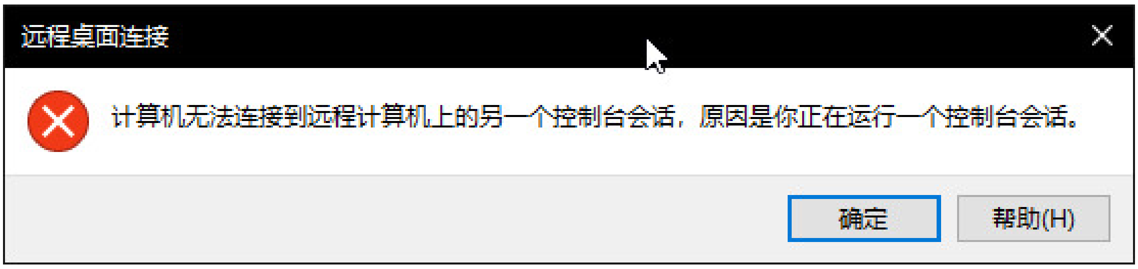


图 3.73 不换端口的结果

运行命令 `sudo sed 's/port=[0-9]{1,5}/port=otherport/' /etc/xrdp/xrdp.ini`，其中 `otherport` 为其他端口（如 `3390`）。

```
[globals]
...
port=3390
```

运行 `sudo service xrdp restart`，然后去开始菜单，用 `localhost:otherport` 来访问。

使用 Xming 连接

进入 Ubuntu 环境，安装 xterm：

```
# apt install xterm -y
```

退出 Ubuntu。

从 Xming X Server 下载地址[16] 下载最新的 Xming Server，然后安装：

如果安装完后忘记勾选 Launch Xming，需在开始菜单里打开 Xming：

之后再回到 Ubuntu，键入如下指令：

```
$ DISPLAY=:0 xterm
```

如果使用了 xfce4，可以在弹出的窗口中使用如下命令激活 xfce4：

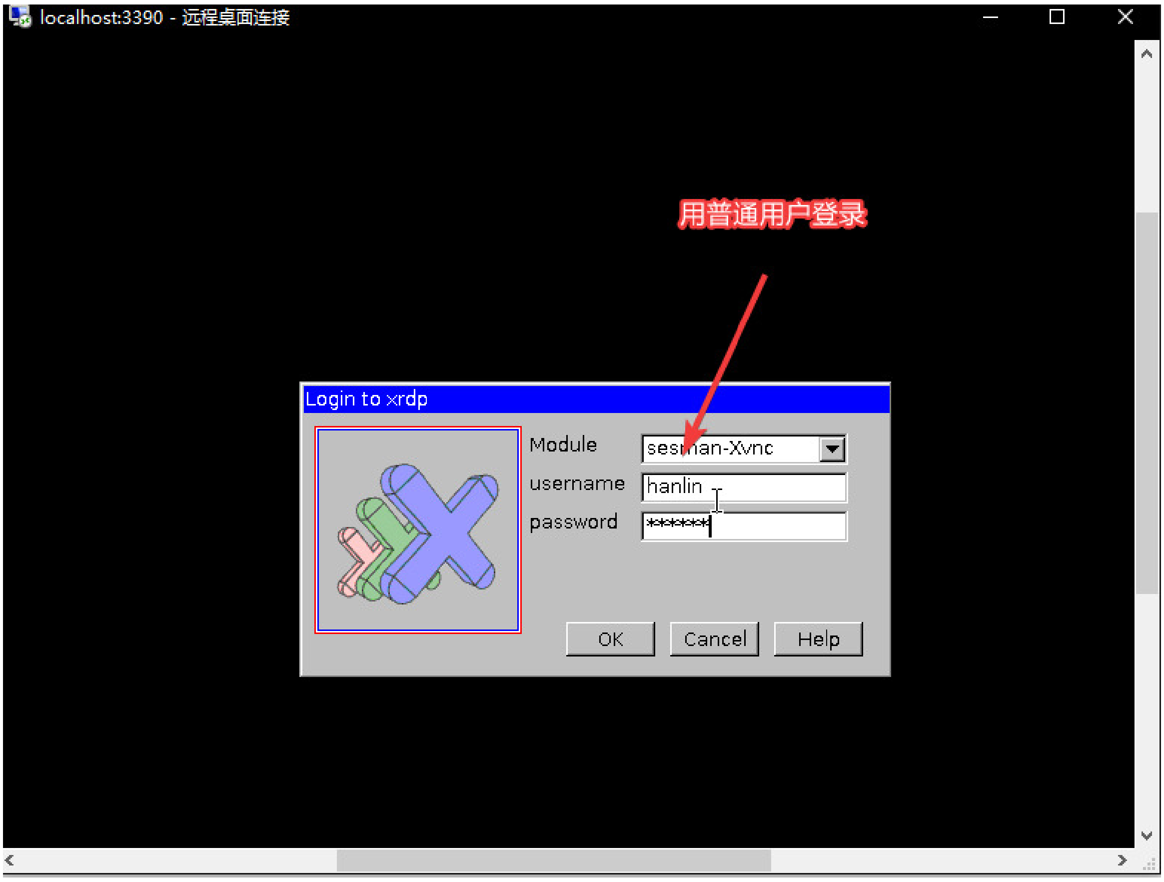

localhost:3390 - 远程桌面连接
用普通用户登录
Login to xrdp
Module
sesman-Xvnc
username
hanlin
password
OK
Cancel
Help

图 3.74

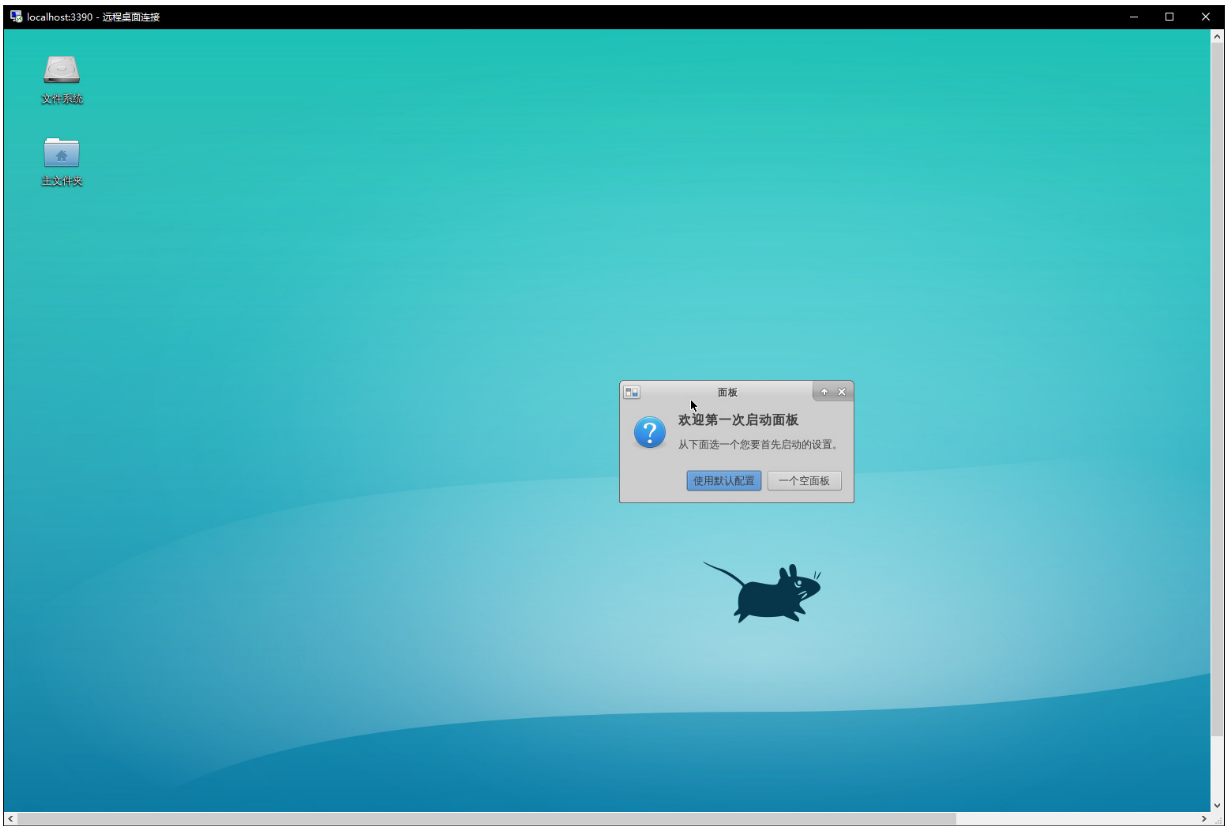

localhost:3390 - 远程桌面连接
文件系统
主文件夹
面板
欢迎第一次启动面板
从下面选一个您要首先启动的设置。
使用默认配置
一个空面板

图 3.75

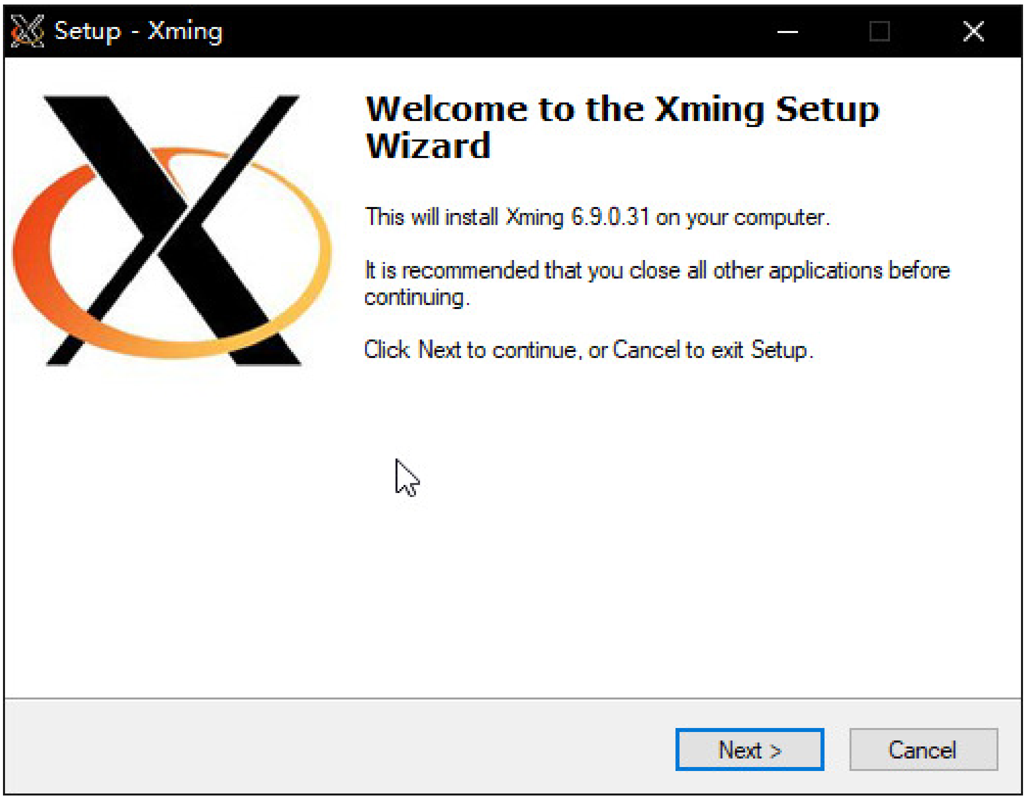


图 3.76

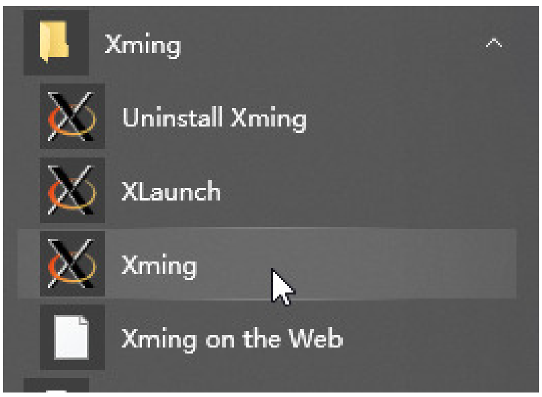


图 3.77 别忘了!

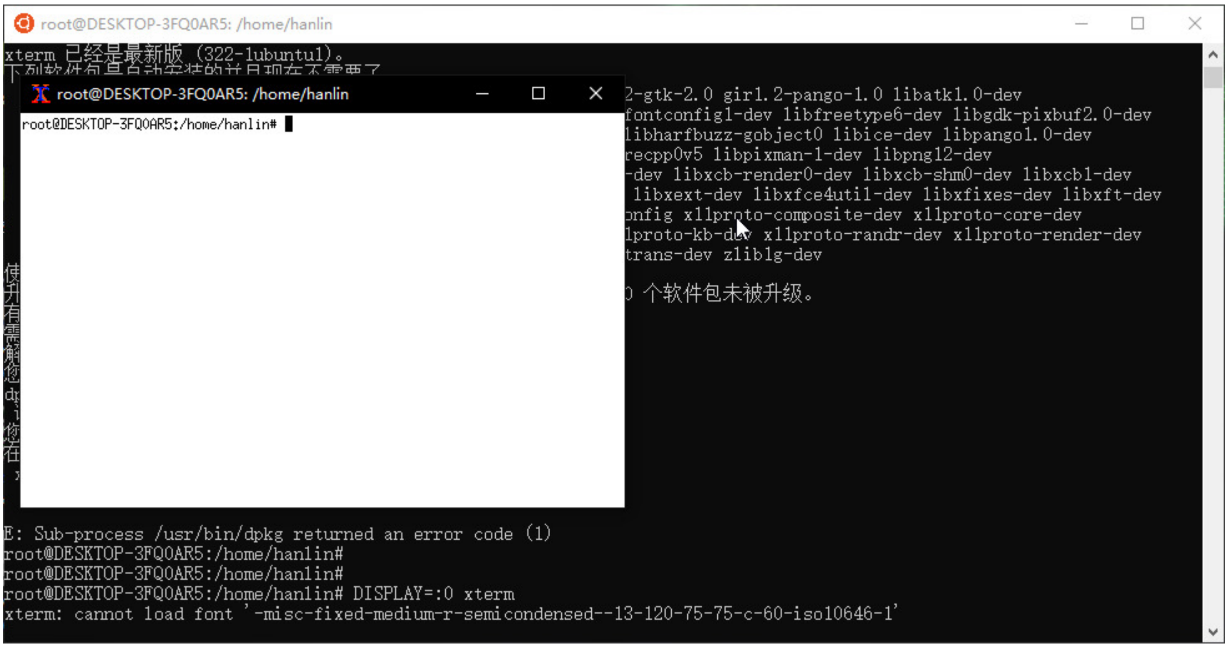


图 3.78

```
$ xfce4-session
```

运行结果如图。(在 Xming 中使用 Ctrl+C 就可以退出该界面。)

WSL 与 Windows 文件的互访问

Windows 下的硬盘被自动挂载至 Linux 环境下的 `/mnt` 文件夹下。如 C 盘在 WSL 下的路径为 `/mnt/c`

```
PS C:\Users\chtholly> bash
/mnt/c/Users/chtholly$ echo "Hello world!" > hello
/mnt/c/Users/chtholly$ exit
PS C:\Users\chtholly> cat hello
Hello world!
PS C:\Users\chtholly> echo "Welcome!" > welcome
PS C:\Users\chtholly> bash
/mnt/c/Users/chtholly$ cat welcome
Welcome!
```

另外，也可以从文件管理器访问 WSL 目录。在安装 WSL 后，可以在资源管理器的侧边栏中发现 Linux 项，在其中可以访问所有安装的发行版中的文件。

同样，也可以在资源管理器的路径或运行（Win+R）中直接输入 `\\wsl$` 来转到 WSL 的目录。

也可以直接使用诸如 `\\wsl$\Ubuntu\home\` 的路径访问其子文件夹。

配合 Visual Sudio Code 进行编辑

如果习惯在 Windows 环境下使用 Visual Studio Code 进行代码编辑，可以安装 VS Code 中的 `Remote - WSL` 插件，更方便地对 WSL 系统中的文件进行编辑。

通过 `Remote - WSL`，可以在 Windows 下的 VS Code 界面中直接对 WSL 子系统进行操作，更加方便地编辑子系统目录下的文件、更方便地使用终端进行调试。

通过在 WSL 中直接键入 `code .`，可以在该目录下直接唤出 Visual Studio Code，对于该目录下的文件进行编辑。

同时，可以通过类似 `code filename` 的命令，对于指定文件进行编辑。

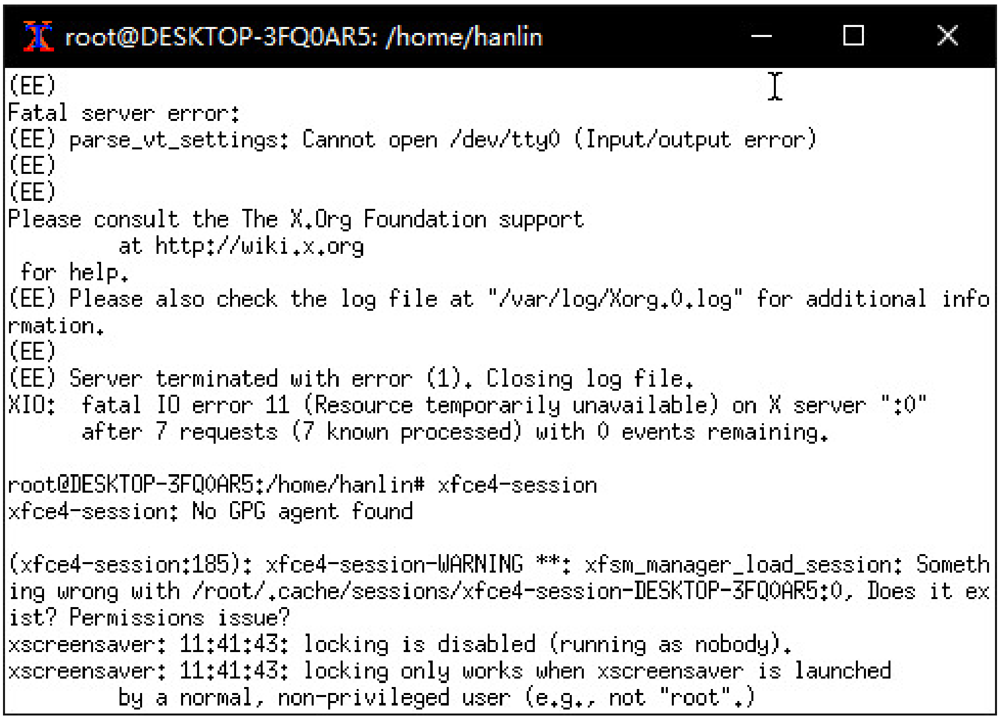

root@DESKTOP-3FQ0AR5: /home/hanlin
(EE)
Fatal server error:
(EE) parse_vt_settings: Cannot open /dev/tty0 (Input/output error)
(EE)
(EE)
Please consult the The X.Org Foundation support
at http://wiki.x.org
for help.
(EE) Please also check the log file at "/var/log/Xorg.0.log" for additional info
rmation.
(EE)
(EE) Server terminated with error (1). Closing log file.
XIO: fatal IO error 11 (Resource temporarily unavailable) on X server ":0"
after 7 requests (7 known processed) with 0 events remaining.
root@DESKTOP-3FQ0AR5:/home/hanlin# xfce4-session
xfce4-session: No GPG agent found
(xfce4-session:185): xfce4-session-WARNING **: xfsm_manager_load_session: Someth
ing wrong with /root/.cache/sessions/xfce4-session-DESKTOP-3FQ0AR5:0. Does it ex
ist? Permissions issue?
xscreensaver: 11:41:43: locking is disabled (running as nobody).
xscreensaver: 11:41:43: locking only works when xscreensaver is launched
by a normal, non-privileged user (e.g., not "root".)

图 3.79

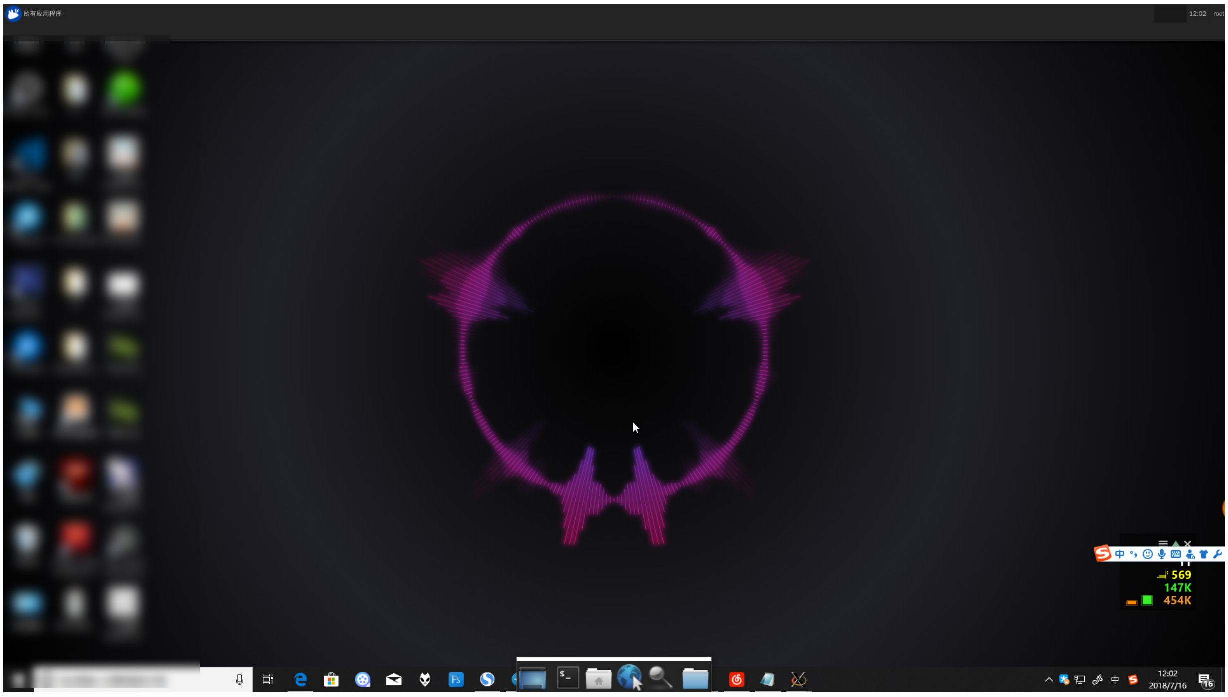


图 3.80

在插件 `Remote - WSL` 的 Getting Started 页面，包含对于编辑操作的详细简介。

同时，也可以参考 Visual Studio Code 的官方文档中关于 WSL 的内容（Remote development in WSL[17]），这篇文章包含从 WSL 安装到配合插件使用的全流程的更详细的介绍。

WSL1 升级为 WSL2

> warning
>
> 请确认已经完成前面 WSL1 的安装步骤。

执行命令 `wsl -l -v` 可以看到 WSL 版本号是 1，需要执行升级，才能到 2。

1. 启用「虚拟机平台」功能

   使用 PowerShell 以管理员身份运行：

   ```
   dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /nores
   tart
   ```

   然后**重启电脑**。

2. 下载 Linux 内核更新包

   - x64[9] 的内核更新包。
   - ARM64/AArch64[18] 的内核更新包。

3. 设置分发版版本

   执行命令：`wsl --set-version < 分发版名称 > < 版本号 >`

   如：将 Ubuntu 18.04 设置为 WSL 2 的命令为 `wsl --set-version Ubuntu-18.04 2`

   这一步比较耗时，执行完成后通过命令 `wsl -l -v` 来检查升级是否成功。

FAQ

参见：常见问题[19]，WSL 2 常见问题解答[20]

- 如何在子系统下进行 xxx?

  可以用自带命令行，或者使用图形界面。比如说 vim，在命令行中键入 `man vim`，会给出一份详尽的使用方法。亦可使用 `vim --help`。

  关于命令行，可阅读 命令行

- 对系统资源的占用量?

  这个系统和 Windows 10 共用 Host，所以理论上是比虚拟机占用小的。

外部链接

- 关于适用于 Linux 的 Windows 子系统[21]
- Ubuntu 镜像使用帮助，清华 TUNA[13]
- Dev on Windows with WSL（在 Windows 上用 WSL 优雅开发）[22]
- GitHub 上的 Awesome-WSL[23]
- 排查适用于 Linux 的 Windows 子系统问题[24]
- WSL1 升级为 WSL2[25]

参考资料与注释

[1] 洛谷日报 #6

[2] NOI Linux 2.0 发布，将于 9 月 1 日起正式启用！

[3] 安装 WSL, Microsoft Docs

[4] 旧版 WSL 的手动安装步骤

[5] WSL-Ubuntu 维基，ubuntu wiki

[6] Ubuntu 的 man 命令帮助如何设置中文版，Frank 看庐山，2017-06-09

[7] Run Bash on Ubuntu on Windows, Mike Harsh, 2016-05-30, Windows Blog

[8] 什么是 Linux 子系统（WSL）

[9] 适用于 x64 计算机的 WSL2 Linux 内核更新包 [9-1] [9-2]

[10] 比较 WSL 2 和 WSL 1

[11] WSL 中的高级设置配置

[12] Ubuntu 的商店页面

[13] 清华 TUNA 的软件源 [13-1] [13-2]

[14] 包管理手册

[15] 在适用于 Linux 的 Windows 子系统上运行 Linux GUI 应用

[16] Xming X Server 下载地址

[17] Remote development in WSL

[18] ARM64/AArch64

[19] 常见问题

[20] WSL 2 常见问题解答

[21] 关于适用于 Linux 的 Windows 子系统

[22] Dev on Windows with WSL（在 Windows 上用 WSL 优雅开发）

[23] GitHub 上的 Awesome-WSL

[24] 排查适用于 Linux 的 Windows 子系统问题

[25] WSL1 升级为 WSL2

3.7 Special Judge

Authors: Xeonacid, NachtgeistW, 2014CAIS01, sshwy, Chrogeek, Menci

本页面主要介绍部分评测工具/OJ 的 spj 编写方法。

简介

Special Judge（简称：spj，别名：checker）是当一道题有多组解时，用来判断答案合法性的程序。

warning

spj 还应当判断文件尾是否有多余内容，及输出格式是否正确（如题目要求数字间用一个空格隔开，而选手却使用了换行）。但是，目前前者只有 Testlib 可以方便地做到这一点，而后者几乎无人去特意进行这种判断。

判断浮点数时应注意 NaN。不合理的判断方式会导致输出 NaN 即可 AC 的情况。

在对选手文件进行读入操作时应该要检查是否正确读入了所需的内容，防止造成 spj 的运行错误。（部分 OJ 会将 spj 的运行错误作为系统错误处理）

note

以下均以 C++ 作为编程语言，以「要求标准答案与选手答案差值小于 1e-3，文件名为 num，单个测试点满分为 10 分」为例。

Testlib

参见：Testlib/ 简介，Testlib/Checker

Testlib 是一个 C++ 的库，用于辅助出题人使用 C++ 编写算法竞赛题。

必须使用 Testlib 作为 spj 的评测工具/OJ：Codeforces、洛谷、UOJ 等。

可以使用 Testlib 作为 spj 的评测工具/OJ：LibreOJ (Lyrio[2])、Lemon、牛客网等。

SYZOJ 2 所需的修改版 Testlib 托管于 pastebin[3][1-1]，但此修改版并未修改交互模式。syzoj/testlib[4] 处托管了一份可以在 SYZOJ 2 上使用交互模式的 Testlib。

Lemon 所需的修改版 Testlib 托管于 GitHub - GitPinkRabbit/Testlib-for-Lemons[5]。注意此版本 Testlib 注册 checker 时应使用 `registerLemonChecker()`，而非 `registerTestlibCmd()`。此版本继承自 matthew99 的旧版[6]，添加了一些 Testlib 的新功能。

DOMJudge 所需的修改版 Testlib 托管于 cn-xcpc-tools/testlib-for-domjudge[7]。此版本 Testlib 同时可作为 Special Judge 的 checker 和交互题的 interactor。

Arbiter 所需的修改版 Testlib 托管于 testlib-for-arbiter[8]。

其他评测工具/OJ 大部分需要按照其 spj 编写格式修改 Testlib，并将 testlib.h 与 spj 一同上传；或将 testlib.h 置于 include 目录。

```cpp
// clang-format off

#include "testlib.h"
#include <cmath>

int main(int argc, char *argv[]) {
  /*
   * inf：输入
   * ouf：选手输出
   * ans：标准输出
   */
  registerTestlibCmd(argc, argv);

  double pans = ouf.readDouble(), jans = ans.readDouble();

  if (abs(pans - jans) < 1e-3)
    quitf(_ok, "Good job\n");
  else
    quitf(_wa, "Too big or too small, expected %f, found %f\n", jans, pans);
}
```

Lemon

note

Lemon 有现成的修改版 Testlib，建议使用 Testlib。

```cpp
#include <cmath>
#include <cstdio>

int main(int argc, char* argv[]) {
  /*
   * argv[1]：输入
   * argv[2]：选手输出
   * argv[3]：标准输出
   * argv[4]：单个测试点分值
   * argv[5]：输出最终得分 (0 ~ argv[4])
   * argv[6]：输出错误报告
   */
  FILE* fin = fopen(argv[1], "r");
  FILE* fout = fopen(argv[2], "r");
  FILE* fstd = fopen(argv[3], "r");
  FILE* fscore = fopen(argv[5], "w");
  FILE* freport = fopen(argv[6], "w");

  double pans, jans;
  fscanf(fout, "%lf", &pans);
  fscanf(fstd, "%lf", &jans);

  if (abs(pans - jans) < 1e-3) {
    fprintf(fscore, "%s", argv[4]);
    fprintf(freport, "Good job\n");
  } else {
```

```
    fprintf(fscore, "%d", 0);
    fprintf(freport, "Too big or too small, expected %f, found %f\n", jans,
            pans);
  }
}
```

Cena

```
#include <cmath>
#include <cstdio>

int main(int argc, char* argv[]) {
  /*
   * FILENAME.in: 输入
   * FILENAME.out: 选手输出
   * argv[1]: 单个测试点分值
   * argv[2]: 标准输出
   * score.log: 输出最终得分 (0 ~ argv[1])
   * report.log: 输出错误报告
   */
  FILE* fin = fopen("num.in", "r");
  FILE* fout = fopen("num.out", "r");
  FILE* fstd = fopen(argv[2], "r");
  FILE* fscore = fopen("score.log", "w");
  FILE* freport = fopen("report.log", "w");

  double pans, jans;
  fscanf(fout, "%lf", &pans);
  fscanf(fstd, "%lf", &jans);

  if (abs(pans - jans) < 1e-3) {
    fprintf(fscore, "%s", argv[1]);
    fprintf(freport, "Good job\n");
  } else {
    fprintf(fscore, "%d", 0);
    fprintf(freport, "Too big or too small, expected %f, found %f\n", jans,
            pans);
  }
}
```

CCR

```
#include <cmath>
#include <cstdio>

int main(int argc, char* argv[]) {
  /*
   * stdin: 输入
   * argv[2]: 标准输出
   * argv[3]: 选手输出
   * stdout:L1: 输出最终得分比率 (0 ~ 1)
   * stdout:L2: 输出错误报告
   */
```

```
  FILE* fout = fopen(argv[3], "r");
  FILE* fstd = fopen(argv[2], "r");

  double pans, jans;
  fscanf(fout, "%lf", &pans);
  fscanf(fstd, "%lf", &jans);

  if (abs(pans - jans) < 1e-3) {
    printf("%d\n", 1);
    printf("Good job\n");
  } else {
    printf("%d\n", 0);
    printf("Too big or too small, expected %f, found %f\n", jans, pans);
  }
}
```

Arbiter

```
#include <cmath>
#include <cstdio>

int main(int argc, char* argv[]) {
  /*
   * argv[1]：输入
   * argv[2]：选手输出
   * argv[3]：标准输出
   * /tmp/_eval.score:L1：输出错误报告
   * /tmp/_eval.score:L2：输出最终得分
   */
  FILE* fout = fopen(argv[2], "r");
  FILE* fstd = fopen(argv[3], "r");
  FILE* fscore = fopen("/tmp/_eval.score", "w");

  double pans, jans;
  fscanf(fout, "%lf", &pans);
  fscanf(fstd, "%lf", &jans);

  if (abs(pans - jans) < 1e-3) {
    fprintf(fscore, "Good job\n");
    fprintf(fscore, "%d", 10);
  } else {
    fprintf(fscore, "Too big or too small, expected %f, found %f\n", jans,
            pans);
    fprintf(fscore, "%d", 0);
  }
}
```

HUSTOJ

```
#include <cmath>
#include <cstdio>

#define AC 0
```

```
#define WA 1

int main(int argc, char* argv[]) {
  /*
   * argv[1]: 输入
   * argv[2]: 标准输出
   * argv[3]: 选手输出
   * exit code: 返回判断结果
   */
  FILE* fin = fopen(argv[1], "r");
  FILE* fout = fopen(argv[3], "r");
  FILE* fstd = fopen(argv[2], "r");

  double pans, jans;
  fscanf(fout, "%lf", &pans);
  fscanf(fstd, "%lf", &jans);

  if (abs(pans - jans) < 1e-3)
    return AC;
  else
    return WA;
}
```

QDUOJ

相较之下，QDUOJ 略为麻烦。它带 spj 的题目没有标准输出，只能把 std 写进 spj，待跑出标准输出后再判断。

```
#include <cmath>
#include <cstdio>

#define AC 0
#define WA 1
#define ERROR -1

double solve(...) {
  // std
}

int main(int argc, char* argv[]) {
  /*
   * argv[1]: 输入
   * argv[2]: 选手输出
   * exit code: 返回判断结果
   */
  FILE* fin = fopen(argv[1], "r");
  FILE* fout = fopen(argv[2], "r");

  double pans, jans;
  fscanf(fout, "%lf", &pans);

  jans = solve(...);
  if (abs(pans - jans) < 1e-3)
    return AC;
  else
    return WA;
```

```
}
```

LibreOJ (SYZOJ 2)

note

LibreOJ (SYZOJ 2) 有现成的修改版 Testlib，建议使用 Testlib。

```
#include <cmath>
#include <cstdio>

int main(int argc, char* argv[]) {
  /*
   * in: 输入
   * user_out: 选手输出
   * answer: 标准输出
   * code: 选手代码
   * stdout: 输出最终得分 (0 ~ 100)
   * stderr: 输出错误报告
   */
  FILE* fin = fopen("in", "r");
  FILE* fout = fopen("user_out", "r");
  FILE* fstd = fopen("answer", "r");
  FILE* fcode = fopen("code", "r");

  double pans, jans;
  fscanf(fout, "%lf", &pans);
  fscanf(fstd, "%lf", &jans);

  if (abs(pans - jans) < 1e-3) {
    printf("%d", 100);
    fprintf(stderr, "Good job\n");
  } else {
    printf("%d", 0);
    fprintf(stderr, "Too big or too small, expected %f, found %f\n", jans,
            pans);
  }
}
```

牛客网

note

牛客网有现成的修改版 Testlib，建议使用 Testlib。

参见：如何在牛客网出 Special Judge 的编程题[9]

```
#include <cmath>
#include <cstdio>

#define AC 0
#define WA 1
```

```
int main(int argc, char* argv[]) {
  /*
   * input: 输入
   * user_output: 选手输出
   * output: 标准输出
   * exit code: 返回判断结果
   */
  FILE* fin = fopen("input", "r");
  FILE* fout = fopen("user_output", "r");
  FILE* fstd = fopen("output", "r");

  double pans, jans;
  fscanf(fout, "%lf", &pans);
  fscanf(fstd, "%lf", &jans);

  if (abs(pans - jans) < 1e-3)
    return AC;
  else
    return WA;
}
```

DOMJudge

note

DOMJudge 支持任何语言编写的 spj，参见：problemarchive.org output validator 格式[10]。

DOMJudge 有现成的修改版 Testlib，建议使用 Testlib。

DOMJudge 使用的 Testlib 及导入 Polygon 题目包方式的文档：https://github.com/cn-xcpc-tools/testlib-for-domjudge[7]

DOMJudge 的 默认比较器[11] 自带了浮点数带精度比较，只需要在题目配置的 `validator_flags` 中添加 `float_tolerance 1e-3` 即可。

```
#include <cmath>
#include <cstdio>

#define AC 42
#define WA 43
char reportfile[50];

int main(int argc, char* argv[]) {
  /*
   * argv[1]: 输入
   * argv[2]: 标准输出
   * argv[3]: 评测信息输出的文件夹
   * stdin: 选手输出
   */
  FILE* fin = fopen(argv[1], "r");
  FILE* fstd = fopen(argv[2], "r");
  sprintf(reportfile, "%s/judgemessage.txt", argv[3]);
  FILE* freport = fopen(reportfile, "w");

  double pans, jans;
  scanf("%lf", &pans);
```

```
  fscanf(fstd, "%lf", &jans);

  if (abs(pans - jans) < 1e-3) {
    fprintf(freport, "Good job\n");
    return AC;
  } else {
    fprintf(freport, "Too big or too small, expected %f, found %f\n", jans,
            pans);
    return WA;
  }
}
```

也可以使用 Kattis Problem Tools 提供的头文件 validate.h[12] 编写，以实现更加复杂的功能。

参考资料

[1] LibreOJ 支持 testlib 检查器啦！ [1-1] [1-2]

[2] Lyrio

[3] pastebin

[4] syzoj/testlib

[5] GitHub - GitPinkRabbit/Testlib-for-Lemons

[6] matthew99 的旧版

[7] cn-xcpc-tools/testlib-for-domjudge [7-1] [7-2]

[8] testlib-for-arbiter

[9] 如何在牛客网出 Special Judge 的编程题

[10] problemarchive.org output validator 格式

[11] 默认比较器

[12] validate.h

3.8 Testlib

3.8.1 Testlib 简介

Authors: Xeonacid, sshwy

如果你正在使用 C++ 出一道算法竞赛题目，Testlib 是编写相关程序（generator, validator, checker, interactor）时的优秀辅助工具。它是俄罗斯和其他一些国家的出题人的必备工具，许多比赛也都在用它：ROI、ICPC 区域赛、所

有 Codeforces round……

Testlib 库仅有 `testlib.h` 一个文件，使用时仅需在所编写的程序开头添加 `#include "testlib.h"` 即可。

Testlib 的具体用途：

- 编写 Generator，即数据生成器。
- 编写 Validator，即数据校验器，判断生成数据是否符合题目要求，如数据范围、格式等。
- 编写 Interactor，即交互器，用于交互题。
- 编写 Checker，即 Special Judge。

Testlib 与 Codeforces 开发的 Polygon[1] 出题平台完全兼容。

`testlib.h` 在 2005 年移植自 `testlib.pas`，并一直在更新。Testlib 与绝大多数编译器兼容，如 VC++ 和 GCC g++，并兼容 C++11。

本文主要翻译自 Testlib - Codeforces[2]。`testlib.h` 的 GitHub 存储库为 MikeMirzayanov/testlib[3]。

参考资料与注释

[1] Polygon

[2] Testlib - Codeforces

[3] MikeMirzayanov/testlib

3.8.2 通用

本页面介绍 Testlib checker/interactor/validator 的一些通用状态/对象/函数、一些用法及注意事项。请在阅读其他页面前完整阅读本页面的内容。

通用状态

结果	Testlib 别名	含义
Ok	`_ok`	答案正确。
Wrong Answer	`_wa`	答案错误。
Presentation Error	`_pe`	答案格式错误。注意包括 Codeforces 在内的许多 OJ 并不区分 PE 和 WA。
Partially Correct	`_pc(score)`	答案部分正确。仅限于有部分分的测试点，其中 `score` 为一个正整数，从 0（没分）到 100（可能的最大分数）。
Fail	`_fail`	validator 中表示输入不合法，不通过校验。checker 中表示程序内部错误、标准输出有误或选手输出比标准输出更优，需要裁判/出题人关注。（也就是题目锅了）

通常用程序的返回值表明结果，但是也有一些其他方法：创建一个输出 xml 文件、输出信息到 stdout 或其他位置……这些都通过下方函数表中的 `quitf` 函数来完成。

通用对象

对象	含义
inf	输入文件流
ouf	选手输出流
ans	参考输出流

通用函数

非成员函数：

调用	含义
void registerTestlibCmd(int argc, char* argv[])	注册程序为 checker
void registerInteraction(int argc, char* argv[])	注册程序为 interactor
void registerValidation()/void registerValidation(int argc, char* argv[])	注册程序为 validator
void registerGen(int argc, char* argv[], int randomGeneratorVersion)	注册程序为 generatorrandomGeneratorVersion 推荐为 1
void quit(TResult verdict, string message)/void quitf(TResult verdict, string message, ...)	结束程序，返回 verdict，输出 message
void quitif(bool condition, TResult verdict, string message, ...)	如果 condition 成立，调用 quitf(verdict, message, ...)

流成员函数：

调用	含义
char readChar()	读入一个字符
char readChar(char c)	读入一个字符，必须为 c
char readSpace()	等同于 readChar(' ')
string readToken()/string readWord()	读入一个串，到空白字符（空格、Tab、EOLN 等）停止
string readToken(string regex)/string readWord(string regex)	读入一个串，必须与 regex 匹配
long long readLong()	读入一个 64 位整数
long long readLong(long long L, long long R)	读入一个 64 位整数，必须在 $[L, R]$ 之间
vector<long long> readLongs(int n, long long L, long long R)	读入 N 个 64 位整数，必须均在 $[L, R]$ 之间
int readInt()/int readInteger()	读入一个 32 位整数
int readInt(int L, int R)/int readInteger(L, R)	读入一个 32 位整数，必须在 $[L, R]$ 之间
vector<int> readInts(int n, int L, int R)/vector<int> readIntegers(int n, int L, int R)	读入 N 个 32 位整数，必须均在 $[L, R]$ 之间
double readReal()/double readDouble()	读入一个双精度浮点数

调用	含义
`double readReal(double L, double R)/double readDouble(double L, double R)`	读入一个双精度浮点数，必须在 $[L, R]$ 之间
`double readStrictReal(double L, double R, int minPrecision, int maxPrecision)/double readStrictDouble(double L, double R, int minPrecision, int maxPrecision)`	读入一个双精度浮点数，必须在 $[L, R]$ 之间，小数位数必须在 $[minPrecision, maxPrecision]$ 之间，不得使用指数计数法等非正常格式
`string readString()/string readLine()`	读入一行（包括换行符），同时将流指针指向下一行的开头
`string readString(string regex)/string readLine(string regex)`	读入一行，必须与 `regex` 匹配
`void readEoln()`	读入 EOLN（在 Linux 环境下读入 `LF`，在 Windows 环境下读入 `CR LF`）
`void readEof()`	读入 EOF
`void quit(TResult verdict, string message)/void quitf(TResult verdict, string message, ...)`	结束程序，若 `Stream` 为 `ouf` 返回 `verdict`，否则返回 `_fail`；输出 `message`
`void quitif(bool condition, TResult verdict, string message, ...)`	如果 `condition` 成立，调用 `quitf(verdict, message, ...)`

未完待续……

极简正则表达式

上面的输入函数中的一部分允许使用「极简正则表达式」特性，如下所示：

- 字符集。如 `[a-z]` 表示所有小写英文字母，`[^a-z]` 表示除小写英文字母外任何字符。
- 范围。如 `[a-z]{1,5}` 表示一个长度在 $[1, 5]$ 范围内且只包含小写英文字母的串。
- 「或」标识符。如 `mike|john` 表示 `mike` 或 `john` 其一。
- 「可选」标识符。如 `-?[1-9][0-9]{0,3}` 表示 $[-9999, 9999]$ 范围内的非零整数（注意那个可选的负号）。
- 「重复」标识符。如 `[0-9]*` 表示零个或更多数字，`[0-9]+` 表示一个或更多数字。
- 注意这里的正则表达式是「贪婪」的（「重复」会尽可能匹配）。如 `[0-9]?1` 将不会匹配 `1`（因为 `[0-9]?` 将 `1` 匹配上，导致模板串剩余的那个 `1` 无法匹配）。

首先 include testlib.h

请确保 testlib.h 是你 include 的**第一个**头文件，Testlib 会重写/禁用（通过名字冲突的方式）一些与随机有关的函数（如 `random()`），保证随机结果与环境无关，这对于 generator 非常重要，generator 页面 会详细说明这一点。

使用项别名

推荐给 `readInt/readInteger/readLong/readDouble/readWord/readToken/readString/readLine` 等的有限制调用最后多传入一个 `string` 参数，即当前读入的项的别名，使报错易读。例如使用 `inf.readInt(1, 100, "n")` 而非 `inf.readInt(1, 100)`，报错信息将为 `FAIL Integer parameter [name=n] equals to 0, violates the range [1, 100]`。

使用 `ensuref/ensure()`

这两个函数用于检查条件是否成立（类似于 `assert()`）。例如检查 $x_i \neq y_i$，我们可以使用

```
ensuref(x[i] != y[i], "Graph can't contain loops");
```

还可以使用 C 风格占位符如

```
ensuref(s.length() % 2 == 0,
        "String 's' should have even length, but s.length()=%d",
        int(s.length()));
```

它有一个简化版 `ensure()`，我们可以直接使用 `ensure(x> y)` 而不添加说明内容（也不支持添加说明内容），如果条件不满足报错将为 `FAIL Condition failed: "x > y"`。很多情况下不加额外的说明的这种报错很不友好，所以我们通常使用 `ensuref()` 并加以说明内容，而非使用 `ensure()`。

warning

注意全局与成员 `ensuref/ensure()` 的区别

全局函数 `::ensuref/ensure()` 多用于 generator 和 validator 中，如果检查失败将统一返回 `_fail`。

成员函数 `InStream::ensuref/ensure()` 一般用于判断选手和参考程序的输出是否合法。当 `InStream` 为 `ouf` 时，返回 `_wa`；为 `inf`（一般不在 checker 中检查输入数据，这应当在 validator 中完成）或 `ans` 时，返回 `_fail`。详见 Checker - 编写 readAns 函数。

本文主要翻译并综合自 Testlib - Codeforces[1] 系列。`testlib.h` 的 GitHub 存储库为 MikeMirzayanov/testlib[2]。

参考资料与注释

[1] Testlib - Codeforces

[2] MikeMirzayanov/testlib

3.8.3 Generator

Generator，即数据生成器。当数据很大，手造会累死的时候，我们就需要它来帮助我们自动造数据。

简单的例子

生成两个 $[1, n]$ 范围内的整数：

```
// clang-format off

#include "testlib.h"
#include <iostream>

using namespace std;

int main(int argc, char* argv[]) {
  registerGen(argc, argv, 1);
  int n = atoi(argv[1]);
  cout << rnd.next(1, n) << " ";
  cout << rnd.next(1, n) << endl;
}
```

为什么要使用 Testlib？

有人说写 generator 不需要用 Testlib，它在这没什么用。实际上这是个不正确的想法。一个好的 generator 应该满足这一点：**在任何环境下对于相同输入它给出相同输出** 。写 generator 就避免不了生成随机值，平时我们用的 `rand()` 或 C++11 的 `mt19937/uniform_int_distribution` ，当操作系统不同、使用不同编译器编译、不同时间运行等，它们的输出都可能不同（对于非常常用的 `srand(time(0))` ，这是显然的），而这就会给生成数据带来不确定性。

需要注意的是，一旦使用了 Testlib，就不能再使用标准库中的 `srand()` ，`rand()` 等随机数函数，否则在编译时会报错。因此，**请确保所有与随机相关的函数均使用 Testlib 而非标准库提供的**。

而 Testlib 中的随机值生成函数则保证了相同调用会输出相同值，与 generator 本身或平台均无关。另外。它给生成各种要求的随机值提供了很大便利，如 `rnd.next("[a-z]{1,10}")` 会生成一个长度在 $[1,10]$ 范围内的串，每个字符为 `a` 到 `z` ，很方便吧！

Testlib 能做什么？

在一切之前，先执行 `registerGen(argc, argv, 1)` 初始化 Testlib（其中 `1` 是使用的 generator 版本，通常保持不变），然后我们就可以使用 `rnd` 对象来生成随机值。随机数种子取自命令行参数的哈希值，对于某 generator `g.cpp` ，`g 100` (Unix-Like) 和 `g.exe "100"` (Windows) 将会有相同的输出，而 `g 100 0` 则与它们不同。

`rnd` 对象的类型为 `random_t` ，你可以建立一个新的随机值生成对象，不过通常你不需要这么做。

该对象有许多有用的成员函数，下面是一些例子：

调用	含义
`rnd.next(4)`	等概率生成一个 $[0,4)$ 范围内的整数
`rnd.next(4, 100)`	等概率生成一个 $[4,100]$ 范围内的整数
`rnd.next(10.0)`	等概率生成一个 $[0,10.0)$ 范围内的浮点数
‘rnd.next(”one	two
`rnd.wnext(4, t)`	`wnext()` 是一个生成不等分布（具有偏移期望）的函数，t 表示调用 `next()` 的次数，并取生成值的最大值。例如 `rnd.wnext(3, 1)` 等同于 `max({rnd.next(3), rnd.next(3)})` ；`rnd.wnext(4, 2)` 等同于 `max({rnd.next(4), rnd.next(4), rnd.next(4)})` 。如果 $t<0$ ，则为调用 $-t$ 次，取最小值；如果 $t=0$ ，等同于 `next()` 。
`rnd.any(container)`	等概率返回一个具有随机访问迭代器（如 `std::vector` 和 `std::string` ）的容器内的某一元素的引用

附：关于 `rnd.wnext(i,t)` 的形式化定义：

$$\operatorname{wnext}(i,t)=\begin{cases}\operatorname{next}(i) & t=0\\ \max(\operatorname{next}(i),\operatorname{wnext}(i,t-1)) & t>0\\ \min(\operatorname{next}(i),\operatorname{wnext}(i,t+1)) & t<0\end{cases}$$

另外，不要使用 `std::random_shuffle()` ，请使用 Testlib 中的 `shuffle()` ，它同样接受一对迭代器。它使用 `rnd` 来打乱序列，即满足如上「好的 generator」的要求。

示例：生成一棵树

下面是生成一棵树的主要代码，它接受两个参数——顶点数和伸展度。例如，当 $n=10,t=1000$ 时，可能会生成链；当 $n=10,t=-1000$ 时，可能会生成菊花。

```
#define forn(i, n) for (int i = 0; i < int(n); i++)

registerGen(argc, argv, 1);

int n = atoi(argv[1]);
int t = atoi(argv[2]);

vector<int> p(n);

/* 为节点 1..n-1 设置父亲 */
forn(i, n) if (i > 0) p[i] = rnd.wnext(i, t);

printf("%d\n", n);

/* 打乱节点 1..n-1 */
vector<int> perm(n);
forn(i, n) perm[i] = i;
shuffle(perm.begin() + 1, perm.end());

/* 根据打乱的节点顺序加边 */
vector<pair<int, int> > edges;
for (int i = 1; i < n; i++)
  if (rnd.next(2))
    edges.push_back(make_pair(perm[i], perm[p[i]]));
  else
    edges.push_back(make_pair(perm[p[i]], perm[i]));

/* 打乱边 */
shuffle(edges.begin(), edges.end());

for (int i = 0; i + 1 < n; i++)
  printf("%d %d\n", edges[i].first + 1, edges[i].second + 1);
```

一次性生成多组数据

跟不使用 Testlib 编写的时候一样，每次输出前重定向输出流就好，不过 Testlib 提供了一个辅助函数 `startTest(test_index)` ，它帮助你将输出流重定向到 `test_index` 文件。

一些注意事项

- 严格遵循题目的格式要求，如空格和换行，注意文件的末尾应有一个换行。
- 对于大数据首选 `printf` 而非 `cout` ，以提高性能。（不建议在使用 Testlib 时关闭流同步）
- 不使用 UB（Undefined Behavior，未定义行为），如本文开头的那个示例，输出如果写成 `cout << rnd.next(1, n) << " " << rnd.next(1, n) << endl;` ，则 `rnd.next()` 的调用顺序没有定义。

新特性：解析命令行参数

在之前，我们通常使用类似 `int n = atoi(argv[3]);` 的代码，但是这样并不好。有以下几点原因：

- 不存在第三个命令行参数的时候是不安全的；
- 第三个命令行参数可能不是有效的 32 位整数。

现在，你可以这样写：`int n = opt<int>(3)` 。与此同时，你也可以使用 `int64_t m = opt<int64_t>(1);` ， `bool t = opt<bool>(2);` 和 `string s = opt(4);` 等。

另外，testlib 同时也支持命名参数。如果有很多参数，这样 `g 10 20000 a true` 的可读性就会比 `g -n10 -m200000 -t=a -increment` 差。

在这种情况下，现在你可以在 generator 中使用以下代码：

```
int n = opt<int>("n");
long long n = opt<long long>("m");
string t = opt("t");
bool increment = opt<bool>("increment");
```

你可以自由地混合使用按下标和按名称读取参数的方式。

支持的用于编写命名参数的方案有以下几种：

- `--key=value` 或 `-key=value` ；
- `--key value` 或 `-key value` ——如果 `value` 不是新参数的开头（不以连字符 `-` 开头或一个/两个连字符后没有跟随字母）；
- `--k12345` 或 `-k12345` ——如果 key `k` 是一个字母，且后面是一个数字；
- `-prop` 或 `--prop` ——启用 bool 属性。

下面是一些例子：

```
g1 -n1
g2 --len=4 --s=oops
g3 -inc -shuffle -n=5
g4 --length 5 --total 21 -ord
```

更多示例

可以在 GitHub[1] 中找到。

本文主要翻译自 testlib.h - Codeforces[2] 。新特性翻译自 Testlib: Opts—parsing command line options[3] 。`testlib.h` 的 GitHub 存储库为 MikeMirzayanov/testlib[4] 。

参考资料与注释

[1] GitHub

[2] testlib.h - Codeforces

[3] Testlib: Opts—parsing command line options

[4] MikeMirzayanov/testlib

3.8.4 Validator

前置知识：通用

本页面将简要介绍 validator 的概念与用法。

概述

Validator（中文：校验器）用于检验造好的数据的合法性。当造好一道题的数据，又担心数据不合法（不符合题目的限制条件：上溢、图不连通、不是树……）时，出题者通常会借助 validator 来检查。[1]

因为 Coderforces 支持 hack 功能，所以所有 Codeforces 上的题目都必须要有 validator。UOJ 也如此。Polygon 内建了对 validator 的支持。

使用方法

直接在命令行输入 `./val` 即可。数据通过 stdin 输入。如果想从文件输入可 `./val < a.in`。

若数据没有问题，则什么都不会输出且返回 0；否则会输出错误信息并返回一个非 0 值。

提示

- 写 validator 时，不能对被 validate 的数据做任何假设，因为它可能包含任何内容。因此，出题者要对各种不合法的情况进行判断（使用 Testlib 会大大简化这一流程）。
  - 例如，输入一个点数为 n 的树，主要工作是判断 n 是否符合范围和判断输入的是树与否。但是切不可在判断过 n 范围之后就不对接下来输入的边的起点与终点的范围进行判断，否则可能会导致 validator RE。
  - 即使不会 RE 也不应该不判断，这会导致你的报错不正确。如上例，如果不判断，报错可能会是「不是一棵树」，但是正确的报错应当是「边起点/终点不在 $[1, n]$ 之间」。
- 不能对选手的读入方式做任何假设。因此，必须保证能通过 validate 的数据完全符合输入格式。
  - 例如，选手可能逐字符地读入数字，在数字与数字之间只读入一个空格。所以在编写 validator 时，数据中的每一个空白字符都要在 validator 中显式地读入（如空格和换行）。
- 结束时不要忘记 `inf.readEof()`。
- 如果题目开放 hack（或者说，validator 的错误信息会给别人看），请使报错信息尽量友好。
  - 读入变量时使用「项别名」。
  - 在判断使用的表达式不那么易懂时，使用 ensuref 而非 ensure。

示例

以下是 CF Gym 100541A - Stock Market[2] 的 validator：

```cpp
#include "testlib.h"

int main(int argc, char* argv[]) {
  registerValidation(argc, argv);
  int testCount = inf.readInt(1, 10, "testCount");
  inf.readEoln();

  for (int i = 0; i < testCount; i++) {
    int n = inf.readInt(1, 100, "n");
    inf.readSpace();
    inf.readInt(1, 1000000, "w");
    inf.readEoln();

    for (int i = 0; i < n; ++i) {
      inf.readInt(1, 1000, "p_i");
      if (i < n - 1) inf.readSpace();
    }
    inf.readEoln();
  }
```

```
  inf.readEof();
}
```

外部链接

- Validator 的更多示例[3]
- `testlib.h` 的 GitHub 存储库 MikeMirzayanov/testlib[4]

参考资料与注释

[1] Validators with testlib.h - Codeforces

[2] CF Gym 100541A - Stock Market

[3] Validator 的更多示例

[4] `testlib.h` 的 GitHub 存储库 MikeMirzayanov/testlib

3.8.5 Interactor

Interactor，即交互器，用于交互题与选手程序交互。交互题的介绍见 题型介绍 - 交互题。

> note
>
> Testlib 仅支持 Codeforces 形式交互题，即两程序交互。不支持 NOI 形式的选手编写函数与其他函数交互。

请在阅读下文前先阅读 通用。

Testlib 为 interactor 提供了一个特殊的流 `std::fstream tout`，它是一个 log 流，你可以在 interactor 中向它写入，并在 checker 中用 `ouf` 读取。

在 interactor 中，我们从 `inf` 读取题目测试数据，将选手程序（和标程）的标准输入写入 `stdout`（在线），从 `ouf` 读选手输出（在线），从 `ans` 读标准输出（在线）。

如果 interactor 返回了 ok 状态，checker（如果有的话）将接管工作，检查答案合法性。

用法

Windows:

```
interactor.exe <Input_File> <Output_File> [<Answer_File> [<Result_File> [-appes]
]],
```

Linux:

```
./interactor.out <Input_File> <Output_File> [<Answer_File> [<Result_File> [-appe
s]]],
```

简单的例子

> **题目**
>
> interactor 随机选择一个 $[1, 10^9]$ 范围内的整数，你要写一个程序来猜它，你最多可以询问 50 次一个 $[1, 10^9]$

范围内的整数。

interactor 将返回：

1：询问与答案相同，你的程序应当停止询问。

0：询问比答案小。

2：询问比答案大。

注意在此题中我们不需要 ans，因为我们不需要将标准输出与其比较；而在其他题中可能需要这么做。

```cpp
int main(int argc, char** argv) {
  registerInteraction(argc, argv);
  int n = inf.readInt();  // 选数
  cout.flush();           // 刷新缓冲区
  int left = 50;
  bool found = false;
  while (left > 0 && !found) {
    left--;
    int a = ouf.readInt(1, 1000000000);  // 询问
    if (a < n)
      cout << 0 << endl;
    else if (a > n)
      cout << 2 << endl;
    else
      cout << 1 << endl, found = true;
    cout.flush();
  }
  if (!found) quitf(_wa, "couldn't guess the number with 50 questions");
  ouf.readEof();
  quitf(_ok, "guessed the number with %d questions!", 50 - left);
}
```

本文主要翻译自 Interactors with testlib.h - Codeforces[1]。testlib.h 的 GitHub 存储库为 MikeMirzayanov/testlib[2]。

参考资料与注释

[1] Interactors with testlib.h - Codeforces

[2] MikeMirzayanov/testlib

3.8.6 Checker

Checker，即 Special Judge，用于检验答案是否合法。使用 Testlib 可以让我们免去检验许多东西，使编写简单许多。

Checker 从命令行参数读取到输入文件名、选手输出文件名、标准输出文件名，并确定选手输出是否正确，并返回一个预定义的结果：

请在阅读下文前先阅读 通用。

简单的例子

题目

给定两个整数 a, b（$-1000 \le a, b \le 1000$），输出它们的和。

这题显然不需要 checker 对吧，但是如果一定要的话也可以写一个：

```
#include "testlib.h"

int main(int argc, char* argv[]) {
  registerTestlibCmd(argc, argv);

  int pans = ouf.readInt(-2000, 2000, "sum of numbers");

  // 假定标准输出是正确的，不检查其范围
  // 之后我们会看到这并不合理
  int jans = ans.readInt();

  if (pans == jans)
    quitf(_ok, "The sum is correct.");
  else
    quitf(_wa, "The sum is wrong: expected = %d, found = %d", jans, pans);
}
```

编写 readAns 函数

假设你有一道题输入输出均有很多数，如：给定一张 DAG，求 s 到 t 的最长路并输出路径（可能有多条，输出任一）。

下面是一个**不好**的 checker 的例子。

不好的实现

```
// clang-format off

#include "testlib.h"
#include <map>
#include <vector>
using namespace std;

map<pair<int, int>, int> edges;

int main(int argc, char* argv[]) {
  registerTestlibCmd(argc, argv);
  int n = inf.readInt();  // 不需要 readSpace() 或 readEoln()
  int m = inf.readInt();  // 因为不需要在 checker 中检查标准输入合法性
                          // (有 validator)
  for (int i = 0; i < m; i++) {
    int a = inf.readInt();
    int b = inf.readInt();
    int w = inf.readInt();
    edges[make_pair(a, b)] = edges[make_pair(b, a)] = w;
  }
  int s = inf.readInt();
  int t = inf.readInt();
```

```
  // 读入标准输出
  int jvalue = 0;
  vector<int> jpath;
  int jlen = ans.readInt();
  for (int i = 0; i < jlen; i++) {
    jpath.push_back(ans.readInt());
  }
  for (int i = 0; i < jlen - 1; i++) {
    jvalue += edges[make_pair(jpath[i], jpath[i + 1])];
  }

  // 读入选手输出
  int pvalue = 0;
  vector<int> ppath;
  vector<bool> used(n, false);
  int plen = ouf.readInt(2, n, "number of vertices");  // 至少包含 s 和 t 两个点
  for (int i = 0; i < plen; i++) {
    int v = ouf.readInt(1, n, format("path[%d]", i + 1).c_str());
    if (used[v - 1])  // 检查每条边是否只用到一次
      quitf(_wa, "vertex %d was used twice", v);
    used[v - 1] = true;
    ppath.push_back(v);
  }
  // 检查起点终点合法性
  if (ppath.front() != s)
    quitf(_wa, "path doesn't start in s: expected s = %d, found %d", s,
          ppath.front());
  if (ppath.back() != t)
    quitf(_wa, "path doesn't finish in t: expected t = %d, found %d", t,
          ppath.back());
  // 检查相邻点间是否有边
  for (int i = 0; i < plen - 1; i++) {
    if (edges.find(make_pair(ppath[i], ppath[i + 1])) == edges.end())
      quitf(_wa, "there is no edge (%d, %d) in the graph", ppath[i],
            ppath[i + 1]);
    pvalue += edges[make_pair(ppath[i], ppath[i + 1])];
  }

  if (jvalue != pvalue)
    quitf(_wa, "jury has answer %d, participant has answer %d", jvalue, pvalue);
  else
    quitf(_ok, "answer = %d", pvalue);
}
```

这个 checker 主要有两个问题：

1. 它确信标准输出是正确的。如果选手输出比标准输出更优，它会被判成 WA，这不太妙。同时，如果标准输出不合法，也会产生 WA。对于这两种情况，正确的操作都是返回 Fail 状态。
2. 读入标准输出和选手输出的代码是重复的。在这道题中写两遍读入问题不大，只需要一个 `for` 循环；但是如果有一道题输出很复杂，就会导致你的 checker 结构混乱。重复代码会大大降低可维护性，让你在 debug 或修改格式时变得困难。

读入标准输出和选手输出的方式实际上是完全相同的，这就是我们通常编写一个用流作为参数的读入函数的原因。

好的实现

```cpp
// clang-format off

#include "testlib.h"
#include <map>
#include <vector>
using namespace std;

map<pair<int, int>, int> edges;
int n, m, s, t;

// 这个函数接受一个流，从其中读入
// 检查路径的合法性并返回路径长度
// 当 stream 为 ans 时，所有 stream.quitf(_wa, ...)
// 和失败的 readXxx() 均会返回 _fail 而非 _wa
// 也就是说，如果输出非法，对于选手输出流它将返回 _wa，
// 对于标准输出流它将返回 _fail
int readAns(InStream& stream) {
  // 读入输出
  int value = 0;
  vector<int> path;
  vector<bool> used(n, false);
  int len = stream.readInt(2, n, "number of vertices");
  for (int i = 0; i < len; i++) {
    int v = stream.readInt(1, n, format("path[%d]", i + 1).c_str());
    if (used[v - 1]) {
      stream.quitf(_wa, "vertex %d was used twice", v);
    }
    used[v - 1] = true;
    path.push_back(v);
  }
  if (path.front() != s)
    stream.quitf(_wa, "path doesn't start in s: expected s = %d, found %d", s,
                 path.front());
  if (path.back() != t)
    stream.quitf(_wa, "path doesn't finish in t: expected t = %d, found %d", t,
                 path.back());
  for (int i = 0; i < len - 1; i++) {
    if (edges.find(make_pair(path[i], path[i + 1])) == edges.end())
      stream.quitf(_wa, "there is no edge (%d, %d) in the graph", path[i],
                   path[i + 1]);
    value += edges[make_pair(path[i], path[i + 1])];
  }
  return value;
}

int main(int argc, char* argv[]) {
  registerTestlibCmd(argc, argv);
  n = inf.readInt();
  m = inf.readInt();
  for (int i = 0; i < m; i++) {
    int a = inf.readInt();
    int b = inf.readInt();
```

```
    int w = inf.readInt();
    edges[make_pair(a, b)] = edges[make_pair(b, a)] = w;
  }
  int s = inf.readInt();
  int t = inf.readInt();

  int jans = readAns(ans);
  int pans = readAns(ouf);
  if (jans > pans)
    quitf(_wa, "jury has the better answer: jans = %d, pans = %d\n", jans,
          pans);
  else if (jans == pans)
    quitf(_ok, "answer = %d\n", pans);
  else  // (jans < pans)
    quitf(_fail, ":( participant has the better answer: jans = %d, pans = %d\n",
          jans, pans);
}
```

注意到这种写法我们同时也检查了标准输出是否合法，这样写 checker 让程序更短，且易于理解和 debug。此种写法也适用于输出 YES（并输出方案什么的），或 NO 的题目。

note

对于某些限制的检查可以用 `InStream::ensure/ensuref()` 函数更简洁地实现。如上例第 23 至 25 行也可以等价地写成如下形式：

```
stream.ensuref(!used[v - 1], "vertex %d was used twice", v);
```

warning

请在 `readAns` 中避免调用**全局**函数 `::ensure/ensuref()`，这会导致在某些应判为 WA 的选手输出下返回 `_fail`，产生错误。

建议与常见错误

- 编写 `readAns` 函数，它真的可以让你的 checker 变得很棒。
- 读入选手输出时永远限定好范围，如果某些变量忘记了限定且被用于某些参数，你的 checker 可能会判定错误或 RE 等。
  - 反面教材

```
// ....
int k = ouf.readInt();
vector<int> lst;
for (int i = 0; i < k; i++)  // k = 0 和 k = -5 在这里作用相同（不会进入循环体）
  lst.push_back(ouf.readInt());
// 但是我们并不想接受一个长度为 -5 的 list，不是吗？
// ....
int pos = ouf.readInt();
int x = A[pos];
// 可能会有人输出 -42, 2147483456 或其他一些非法数字导致 checker RE
```

  - 正面教材

```
// ....
int k = ouf.readInt(0, n);  // 长度不合法会立刻判 WA 而不会继续 check 导致 RE
```

```cpp
vector<int> lst;
for (int i = 0; i < k; i++) lst.push_back(ouf.readInt());
// ....
int pos = ouf.readInt(0, (int)A.size() - 1);  // 防止 out of range
int x = A[pos];
// ....
```

- 使用项别名。
- 和 validator 不同，checker 不用特意检查非空字符。例如对于一个按顺序比较整数的 checker，我们只需判断选手输出的整数和答案整数是否对应相等，而选手是每行输出一个整数，还是在一行中输出所有整数等格式问题，我们的 checker 不必关心。

使用方法

通常我们不需要本地运行它，评测工具/OJ 会帮我们做好这一切。但是如果需要的话，以以下格式在命令行运行：

```
./checker <input-file> <output-file> <answer-file> [<report-file> [<-appes>]]
```

一些预设的 checker

很多时候我们的 checker 完成的工作很简单（如判断输出的整数是否正确，输出的浮点数是否满足精度要求），Testlib[1] 已经为我们给出了这些 checker 的实现，我们可以直接使用。

一些常用的 checker 有：

- ncmp：按顺序比较 64 位整数。
- rcmp4：按顺序比较浮点数，最大可接受误差（绝对误差或相对误差）不超过 10^{-4}（还有 rcmp6，rcmp9 等对精度要求不同的 checker，用法和 rcmp4 类似）。
- wcmp：按顺序比较字符串（不带空格，换行符等非空字符）。
- yesno：比较 YES 和 NO，大小写不敏感。

  本文主要翻译自 Checkers with testlib.h - Codeforces[2]。`testlib.h` 的 GitHub 存储库为 MikeMirzayanov/testlib[3]。

参考资料与注释

[1] Testlib

[2] Checkers with testlib.h - Codeforces

[3] MikeMirzayanov/testlib

3.9 Polygon

> **Authors:** ouuan, NachtgeistW

本页面将简要介绍多人协作出题平台 Polygon。

简介

什么是 Polygon

网址：Index Page - Polygon[1]

Polygon 是一个支持多人协作的出题平台，功能非常完善。官网描述为「Polygon 的使命是为创建编程竞赛题目提供平台。」

在 Codeforces (CF) 出题必须使用 Polygon。在其它地方出题，尤其是多人合作出题时，使用 Polygon 也是不错的选择。

优点

- 有版本管理系统，多人合作时不会乱成一团，也不需要互相传文件。
- 出题系统完善，validator、generator、checker、solutions 环环相扣，输出自动生成。
- 可以为 solutions 设置标签，错解 AC、正解未 AC 都会警告，方便地逐一卡掉错解。
- 可以方便地对拍，拍出来的数据可以直接添加到题目数据中。
- 发现问题可以提 issue，而不会被消息刷屏却一直没有 fix。
- 为日后出 CF 做准备。
- ……

题目列表

题目列表中会显示一道题目的基本信息，如题面、题解撰写情况、数据生成情况以及 std、validator 和 checker 的设置。

可以双击题目列表的"Name" 这一栏来写上 note，比如需要提醒自己做的事（need to add more tests/need to write tutorial），或者是这道题预订的 score distribution，可以根据自己的需要随意填写，当然也可以空着。

"Rev." 中的"x/y" 的 x 指当前题目版本，y 指 package 的版本。如果两者不一样 y 会显示为红色。

"Edit session" 中的"Start" 是指你的账号还没有看过这道题，"Continue (x) Discard" 是指你的账号处于这个题目的第 x 个版本，点击"Start" 或"Continue (x)" 就会进入题目管理界面，点击"Discard" 会**不可恢复地**撤销你的所有更改，回到没有看过这题的状态。

如果你的账号上有一道题的更改没有提交，题目列表中这一整行就会变红。

题目管理

Polygon 的大部分功能都不需要学，能看懂英文就基本能用了。

> warning
>
> 题面不能使用 Markdown，只能用 TeX。

- Invocation 是用来测试 solution 的。
- Stress 是用来对拍的。
- 数据在 Tests 中用 generator 造，generator 在 Files 中上传。

General Info

在这个页面中可以设置题目的时间限制、空间限制、题目类型。

需要注意，"Statement description" 和"Problem tutorial" 并不是用来写题面和题解的，这两个输入框可能是历史遗留原因。

Statement

这个页面是用来写题面和题解的。还可以通过"Review" 按钮来查看题面、validator 与 checker，一般用于审核。

题面和题解都需要使用 TeX 的语法，不能使用 Markdown。例如，需要使用 `\textbf{text}` 而不是 `**text**`。但 Polygon 支持的实际上是 TeX 的一个非常小的子集，具体可以自己尝试。

可以通过最上方的"In HTML" 链接查看渲染后的题面，通过"Tutorial in HTML" 查看渲染后的题解。

如果需要在题面中添加图片，需要先在下面的"Statement Resource Files" 中上传图片，然后在题面中加上 `\includegraphics{filename.png}`。

Files

"Source Files" 是用来存放**除了 solutions 外**的其它代码的，如 validator、checker、generator，如果是 IO 式交互题还有 interactor。

如果这些代码需要 include 其它文件，例如 Tree-Generator[2]，需要放在"Resource Files" 中。

grader 式交互参见 官方教程[3]。

Checker

testlib.h 提供了一些内置的 checker，在选择框中有简要介绍，也可以选择后再点"View source" 查看源码。

如果需要自己编写 checker，请参考 checker 教程。

下面的"Checker tests" 是通过"Add test" 添加若干组输出以及对应的期望评测结果，然后点击"Run tests" 就可以测试 checker 是否正确返回了评测结果。

Interactor

仅 IO 式交互题需要，请参考 interactor 教程。

Validator

validator 用来检测数据合法性，编写请参考 validator 教程。

下面的"Validator tests" 类似于"Checker tests"，需要提供输入和期望是否合法，用来测试 validator。

Tests

这个页面是用来管理数据的。

在 Polygon 上，推荐的做法是使用少量**带命令行参数**的 generator 来生成数据，而不是写一堆 generator 或者每生成一组数据都修改 generator。并且，只需要生成输入，输出会自动生成。

"Testset" 就是一个测试集，如果是给 CF 出题需要手动添加"pretests" 这个 Testset，并且"pretests" 需要是"tests" 的子集。

"Add Test" 是手动添加一组数据，一般用于手动输入样例或较小的数据。虽然可以通过文件上传数据，但这是**不推荐的**，数据应该要么是手动输入的要么是使用 generator 在某个参数下生成的。

如果勾选了"Use in statements"，这组数据就会成为样例，自动加在题面里。如果需要题面里显示的不是样例的输入输出（一般用于交互题），就可以点"If you want to specify custom content of input or output data for statements click here"，然后输入你想显示在题面中的输入输出。

Tests 页面的下方是用来输入生成数据的脚本的，如 `generator-name [params] > test-index`。可以使用 `generator-name [params] > $`，就不用手动指定测试点编号了。

可以参考 Polygon 提供的教程[4] 使用 Freemarker 来批量生成脚本。

"Preview Tests" 可以预览生成的数据。

Stresses

这个页面是用来对拍的。

点击"Add Stress" 就可以添加一组对拍，"Script pattern" 是一个生成数据的脚本，其中可以使用"[10..100]" 之类的来表示在一个范围内随机选择。

然后运行对拍，如果拍出错就会显示"Crashed"，并且可以一键把这组数据加到 Tests 中。

Solution Files

这个页面是用来放解这道题的代码的，可以是正解也可以是错解。将错解传上来可以便捷地卡掉它们，也可以提醒自己需要卡掉它们。

Invocations

这个页面是用来运行 solutions 的。

选择代码和测试点就可以运行了，之后可以在列表里点进去（"View"）查看详细信息。

评测状态"FL" 表示评测出错了，一般是数据没有过 validate 或者 validator/checker/interactor 之类的 RE 了。"RJ" 有两种情况，一种是出现了"FL"，另一种是这份代码第一个测试点就没有通过。

如果用时在时限的一半到两倍之间，会用黄色标识出来。

如果数据中存在变量没有达到最小值或最大值，会在最下方提醒。

Issues

用来提 Issue 的地方。

Packages

Package 包含了一道题的全部信息，在出 CF 时是 CF 评测的依据（例如，如果赛时要修锅，更新了 package 才会影响到 CF），其它时候可以用来导出。

"Verify" 是测试所有 solution 都符合标签（AC、WA、TLE），并且 checker 通过 checker tests，validator 通过 validator tests。

Manage access

管理题目权限。

侧边栏

第一栏会显示一些基本信息，如果有哪里不符合规范（如 tests 没有包含 pretests、有重复的测试点）就会显示为黄色，鼠标移上去会显示具体信息。

"View changes" 可以看修改的历史记录。需要注意的是"switch" 不能用来回退到某一个版本，只能在某个版本的基础上进行不产生冲突的修改，而这实际上是没有意义的，所以 switch 相当于是只读的。

"Update Working Copy" 是获取（他人的）更新。

"Commit Changes" 是提交你的更新。

commit 时如果有不合规范、需要警告的地方会列出来。

比赛管理

如果要出一场比赛，可以通过"New Contest" 来创建比赛，就可以更加方便地管理题目。

比赛管理页面的题目列表右上角的"Add problems?" 是把一道已有的题目加到比赛里。

侧边栏的"New problem" 是新建一道题目加到比赛里。

上面的"Manage problem access" 是查看每道题的权限，下面的"Manage developers list" 是管理有这场比赛的权限的人。通过"New problem" 创建一道题以及添加一个新的 developer 时会自动添加权限，但通过"Add problems?" 加进来的题不会给已有的 developer 权限。

侧边栏还可以预览所有题面、所有题解、所有 validator & checker，下载整个比赛的 package，给题目重新编号。

注意事项

Polygon 虽然拥有版本管理系统，但是并没有冲突解决系统，一旦发生冲突就无法进入题目管理界面，只能撤销修改后手动重做。并且，只要修改了同一个文件，即使不是同一行也会发生冲突。

所以，使用 Polygon 时请与合作者保持沟通，commit 前保证没有其他人在修改。

参考资料与注释

[1] Index Page - Polygon

[2] Tree-Generator

[3] 官方教程

[4] Polygon 提供的教程

3.10 OJ 工具

本页面将介绍一些 OJ 工具。

cf-tool

cf-tool 是 Codeforces 的命令行界面的跨平台（支持 Windows、Linux、macOS）工具，支持很多常用操作。

源码托管在 xalanq/cf-tool[1] 上。

特点

- 支持 Codeforces 中的所有编程语言。
- 支持 Contests 和 Gym。
- 提交代码。
- 动态刷新提交后的情况。
- 拉取问题的样例。
- 本地编译和测试样例。
- 拉取某人的所有代码。
- 从指定模板生成代码（包括时间戳，作者等信息）。
- 列出某场比赛的所有题目的整体信息。
- 用默认的网页浏览器打开题目页面、榜单、提交页面等。
- 丰富多彩的命令行。

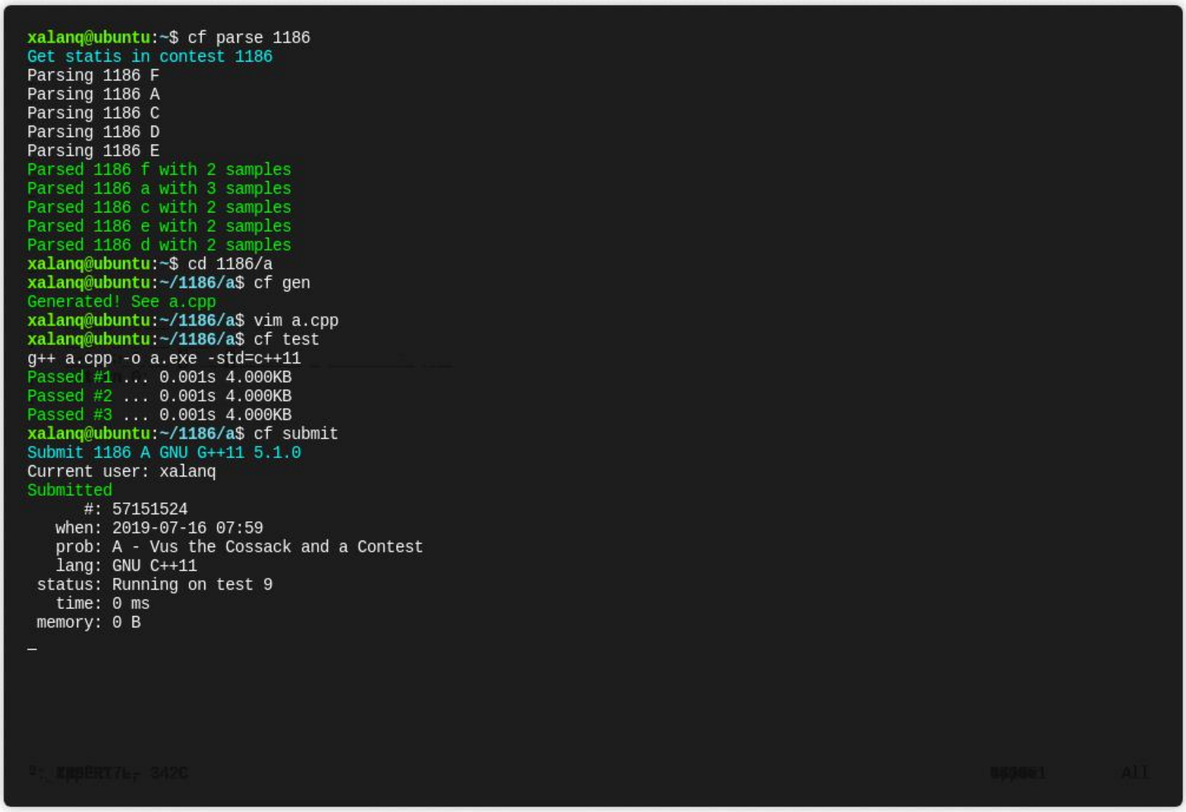


图 3.81 cf-tool 使用截图 1

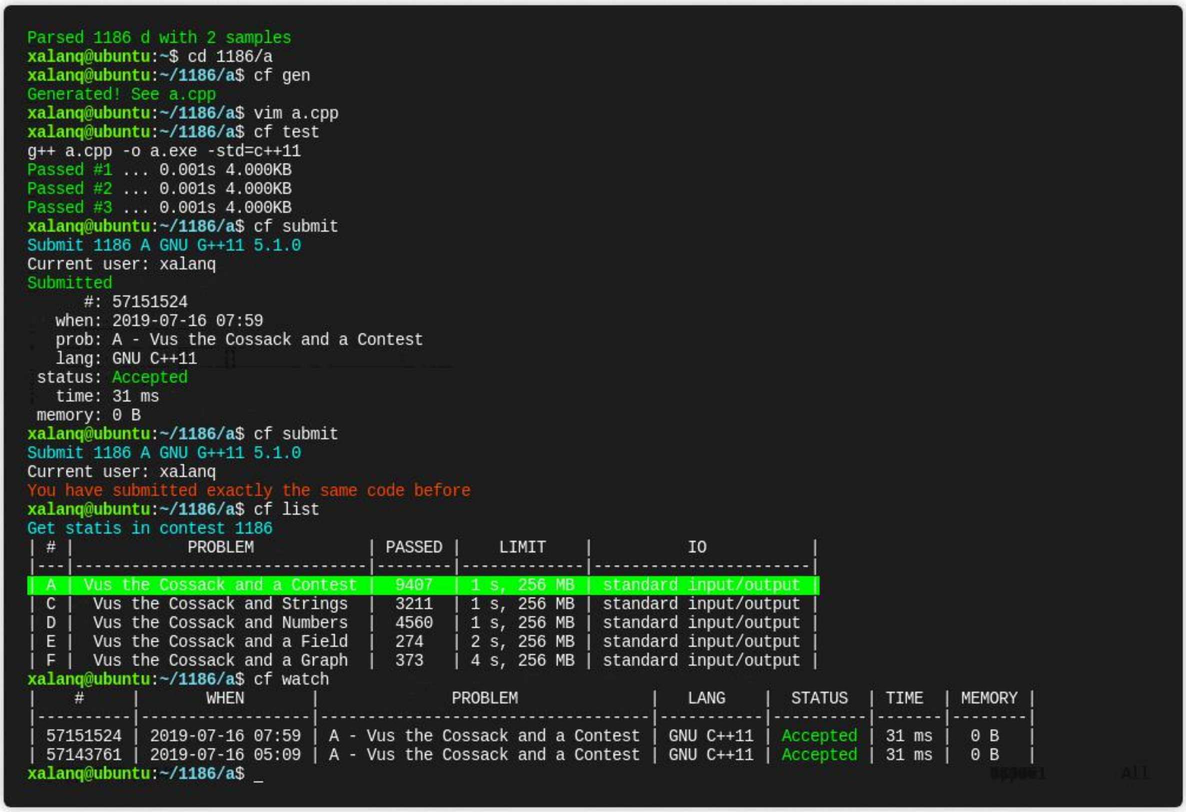


图 3.82 cf-tool 使用截图 2

下载

前往 cf-tool/releases[2] 下载最新版。

之后的更新可以直接使用 `upgrade` 命令获取。

使用

将下载好的可执行文件 `cf`（或者 `cf.exe`）放置到合适的位置后（见常见问题的第二条），然后打开命令行，用 `cf config` 命令来配置一下用户名、密码和代码模板。

使用举例

以下简单模拟一场比赛的流程。

`cf race 1136`

要开始打 1136 这场比赛了！其中 1136 可以从比赛的链接获取，比方说这个例子的比赛链接就为 https://codeforces.com/contest/1136[3]。

如果比赛还未开始，则该命令会进行倒计时。比赛已开始或倒计时完后，工具会自动用默认浏览器打开比赛的所有题目页面，并拉取样例到本地。

`cd 1136/a`

进入 A 题的目录，此时该目录下会包含该题的样例。

`cf gen`

用默认模板生成一份代码，在这里不妨设为 `a.cpp`。

`vim a.cpp`

用 Vim 写代码（或者用其他的编辑器或 IDE 进行）。

`cf test`

编译并测试样例。

`cf submit`

提交代码。

`cf list`

查看当前比赛各个题目的信息。

`cf stand`

用浏览器打开榜单，查看排名。

常见问题

1. 我双击了这个程序但是没啥效果

   cf-tool 是命令行界面的工具，你应该在终端里运行这个工具。

2. 我无法使用 `cf` 这个命令

   你应该将 `cf` 这个程序放到一个已经加入到系统变量 PATH 的路径里（比如说 Linux 里的 `/usr/bin/`）。

   不明白的话请直接搜索「PATH 添加路径」。

3. 如何加一个新的测试数据

   新建两个额外的测试数据文件 `inK.txt` 和 `ansK.txt`（K 是包含 0~9 的字符串）。

4. 怎样在终端里启用 tab 补全命令

   使用这个工具 Infinidat/infi.docopt_completion[4] 即可。

   注意：如果有一个新版本发布（尤其是添加了新命令），你应该重新运行 `docopt-completion cf`。

Codeforces Visualizer

官网：Codeforces Visualizer[5]

源码托管在 sjsakib/cfviz[6] 上。

这个网站有三个功能：

- 用炫酷的图表来可视化某个用户的各种信息（比如通过题目的难度分布）。
- 对比两个用户。
- 计算一场比赛的 Rating 预测。

Competitive Companion

这个工具是一个浏览器插件，用于解析网页里面的测例数据。它支持解析几乎所有的主流 oj 平台（比如 Codeforces、AtCoder）。使用这个插件后，再也不用手动复制任何的测例数据。

源码托管在 jmerle/competitive-companion[7] 上。

使用方法：

- 在谷歌或者火狐浏览器上安装插件。该工具会将解析到的测例数据以 JSON 格式的形式发到指定的端口。
- 在本地安装任何可以从端口监听读取数据的工具即可，可以参考 官方给出的示例[8]。

图片演示：

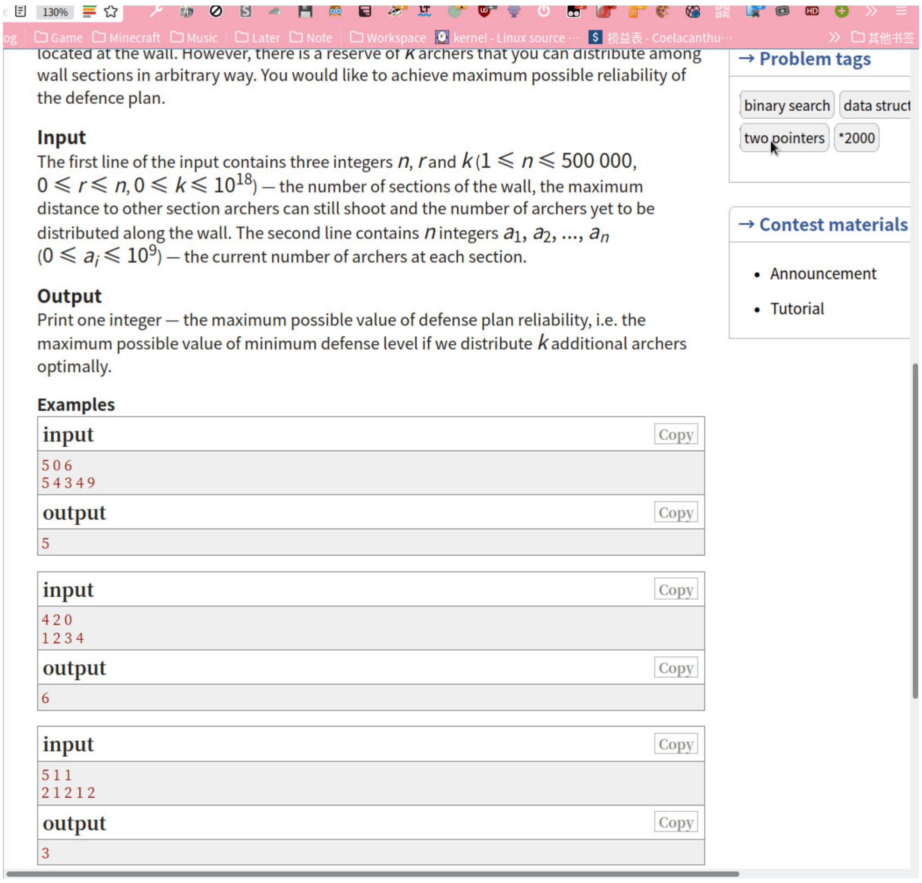


图 3.83 Competitive Companion 使用演示

使用 zqxyz73[9] 同学的 bytetools[10] 完成演示。

ac-predictor

ac-predictor 是一个在 atcoder rating 更新前提前知道比赛 rating 变化的插件。

这个工具是一个 tampermonkey 脚本，所以你需要首先安装 tampermonkey[11]。

完成后来到 greasyfork[12]，点击安装即可。

安装完成后，比赛的排行榜界面会显示每个用户的 rating 变化预测。

这个工具有一个经由 GoodCoder666[13] 汉化的版本，点击 这里[14] 以安装。

参考资料与注释

[1] xalanq/cf-tool

[2] cf-tool/releases

[3] https://codeforces.com/contest/1136

[4] Infinidat/infi.docopt_completion

[5] Codeforces Visualizer

[6] sjsakib/cfviz

[7] jmerle/competitive-companion

[8] 官方给出的示例

[9] zqxyz73

[10] bytetools

[11] tampermonkey

[12] greasyfork

[13] GoodCoder666

[14] 这里

3.11 LaTeX 入门

介绍

什么是 LaTeX

LaTeX（读作/ l t x/或/ le t x/）是一个让你的文档看起来更专业的排版系统，而不是文字处理器。它尤其适合处理篇幅较长、结构严谨的文档，并且十分擅长处理公式表达。它是免费的软件，对大多数操作系统都适用。

LaTeX 基于 TeX（Donald Knuth 在 1978 年为数字化排版设计的排版系统）。TeX 是一种电脑能够处理的低级语言，但大多数人发现它很难使用。LaTeX 正是为了让它变得更加易用而设计的。目前 LaTeX 的版本是 LaTeX 2e。

如果你习惯于使用微软的 Office Word 处理文档，那么你会觉得 LaTeX 的工作方式让你很不习惯。Word 是典型的「所见即所得」的编辑器，你可以在编排文档的时侯查看到最终的排版效果。但使用 LaTeX 时你并不能方便地查看最终效果，这使得你专注于内容而不是外观的调整。

一个 LaTeX 文档是一个以 `.tex` 结尾的文本文件，可以使用任意的文本编辑器编辑，比如 Notepad，但对于大多数人而言，使用一个合适的 LaTeX 编辑器会使得编辑的过程容易很多。在编辑的过程中你可以标记文档的结构。完成后你可以进行编译——这意味着将它转化为另一种格式的文档。它支持多种格式，但最常用的是 PDF 文档格式。

在开始之前

下面列出在本文中使用到的记号：

- 希望你实施的操作会被打上一个箭头 →；
- 你输入的字符会被装进代码块中；
- 菜单命令与按钮的名称会被标记为**粗体**。

一些概念

如果需要编写 LaTeX 文档，你需要安装一个「发行版」，常用的发行版有 TeX Live[1]、MikTeX[2] 和适用于 macOS 用户的 MacTeX（实际上是 TeX Live 的 macOS 版本），至于 CTeX[3] 则现在不推荐使用。TeX Live 和 MacTeX 带有几乎所有的 LaTeX 宏包；而 MikTeX 只带有少量必须的宏包，其他宏包将在需要时安装。

TeX Live 和 MikTeX 都带有 TeXworks 编辑器，你也可以安装功能更多的 TeXstudio 编辑器，或者自行配置 Visual Studio Code 或 Notepad++ 等编辑器。下文所使用的编辑器是运行在 Windows 7 上的 TeXworks。

大部分发行版都带有多个引擎，如 pdfTeX 和 XeTeX。对于中文用户，推荐使用 XeTeX 以获得 Unicode 支持。

TeX 有多种格式，如 Plain TeX 和 LaTeX。现在一般使用 LaTeX 格式。所以，你需要使用与你所使用的格式打包在一起的引擎。如对于 pdfTeX，你需要使用 pdfLaTeX，对于 XeTeX 则是 XeLaTeX。

扩展阅读：TeX 引擎 、 格式 、 发行版之介绍[4]。

环境配置

对于 Windows 用户，你需要下载 TeX Live 或 MikTeX。国内用户可以使用 清华大学 TUNA 镜像站[5]，请点击页面右侧的「获取下载链接」按钮，并选择「应用软件」标签下的「TeX 排版系统」即可下载 TeX Live 或 MikTeX 的安装包，其中 TeX Live 的安装包是一个 ISO 文件，需要挂载后以管理员权限执行 `install-tl-advanced.bat`。

对于 macOS 用户，清华大学 TUNA 镜像站同样提供 MacTeX 和 macOS 版 MikTeX 的下载。

对于 Linux 用户，如果使用 TeX Live，则同样下载 ISO 文件，执行 `install-tl` 脚本；如果使用 MikTeX，则按照 官方文档[6] 进行安装。

文档结构

基本要素

→ 打开 TeXworks。

一个新的文档会被自动打开。

→ 进入 **Format** 菜单，选择 **Line Numbers**。

行号并不是要素，但它可以帮助你比较代码与屏幕信息，找到错误。

→ 进入 **Format** 菜单，选择 **Syntax Coloring**，然后选择 **LaTeX**。

语法色彩会高亮代码，使得代码更加易读。

→ 输入以下文字:

```
\documentclass[a4paper,12pt]{article}

\begin{document}

A sentence of text.

\end{document}
```

`\documentclass` 命令必须出现在每个 LaTeX 文档的开头。花括号内的文本指定了文档的类型。**article** 文档类型适合较短的文章，比如期刊文章和短篇报告。其他文档类型包括 **report**（适用于更长的多章节的文档，比如博士生论文），**proc**（会议论文集），**book** 和 **beamer**。方括号内的文本指定了一些选项——示例中它设置纸张大小为 A4，主要文字大小为 12pt。

`\begin{document}` 和 `\end{document}` 命令将你的文本内容包裹起来。任何在 `\begin{documnet}` 之前的文本都被视为前导命令，会影响整个文档。任何在 `\end{document}` 之后的文本都会被忽视。

空行不是必要的，但它可以让长的文档更易读。

→ 按下 **Save** 按扭；→ 在 **Libraries>Documents** 中新建一个名为 **LaTeX course** 文件夹；→ 将你的文档命名为 **Doc1** 并将其保存为 **TeX document** 放在这个文件夹中。

将不同的 LaTeX 文档放在不同的目录下，在编译的时候组合多个文件是一个很好的想法。

→ 确保 typeset 菜单设置为了 **xeLaTeX**。→ 点击 **Typeset** 按扭。

这时你的源文件会被转换为 PDF 文档，这需要花费一定的时间。在编译结束后，TeXworks 的 PDF 查看器会打开并预览生成的文件。PDF 文件会被自动地保存在与 TeX 文档相同的目录下。

处理问题

如果在你的文档中存在错误，TeXworks 无法创建 PDF 文档时，**Typeset** 按扭会变成一个红叉，并且底部的终端输出会保持展开。这时:

→ 点击 **Abort typesetting** 按扭。→ 阅读终端输出的内容，最后一行可能会给出行号表示出现错误的位置。→ 找到文档中对应的行并修复错误。→ 再次点击 **Typeset** 按扭尝试编译源文件。

添加文档标题

`\maketitle` 命令可以给文档创建标题。你需要指定文档的标题。如果没有指定日期，就会使用现在的时间，作者是可选的。

→ 在 `\begin{document}` 和命令后紧跟着输入以下文本:

```
\title{My First Document}
\author{My Name}
\date{\today}
\maketitle
```

你的文档现在长成了这样:

```
\documentclass[a4paper,12pt]{article}

\begin{document}

\title{My First Document}
\author{My Name}
\date{\today}
\maketitle
```

```
A sentence of text.

\end{document}
```

→ 点击 **Typeset** 按扭，核对生成的 PDF 文档。

要点笔记：

- `\today` 是插入当前时间的命令。你也可以输入一个不同的时间，比如 `\date{November 2013}`。
- **article** 文档的正文会紧跟着标题之后在同一页上排版。**report** 会将标题置为单独的一页。

章节

如果需要的话，你可能想将你的文档分为章（Chatpers）、节（Sections）和小节（Subsections）。下列分节命令适用于 **article** 类型的文档：

- `\section{...}`
- `\subsection{...}`
- `\subsubsection{...}`
- `\paragraph{...}`
- `\subparagraph{...}`

花括号内的文本表示章节的标题。对于 **report** 和 **book** 类型的文档我们还支持 `\chapter{...}` 的命令。

→ 将”A sentence of text.” 替换为以下文本：

```latex
\section{Introduction}
This is the introduction.

\section{Methods}

\subsection{Stage 1}
The first part of the methods.

\subsection{Stage 2}
The second part of the methods.

\section{Results}
Here are my results.
```

你的文档会变成

```latex
\documentclass[a4paper,12pt]{article}

\begin{document}

\title{My First Document}
\author{My Name}
\date{\today}
\maketitle

\section{Introduction}
This is the introduction.

\section{Methods}
```

```
\subsection{Stage 1}
The first part of the methods.

\subsection{Stage 2}
The second part of the methods.

\section{Results}
Here are my results.

\end{document}
```

→ 点击 **Typeset** 按扭，核对 PDF 文档。应该是长这样的：

My First Document

My Name

August 16, 2019

1 Introduction

This is the introduction.

2 Methods

2.1 Stage 1

The first part of the methods.

2.2 Stage 2

The second part of the methods.

3 Results

Here are my results.

图 3.84 p1

创建标签

你可以对任意章节命令创建标签，这样他们可以在文档的其他部分被引用。使用 `\label{labelname}` 对章节创建标签。然后输入 `\ref{labelname}` 或者 `\pageref{labelname}` 来引用对应的章节。

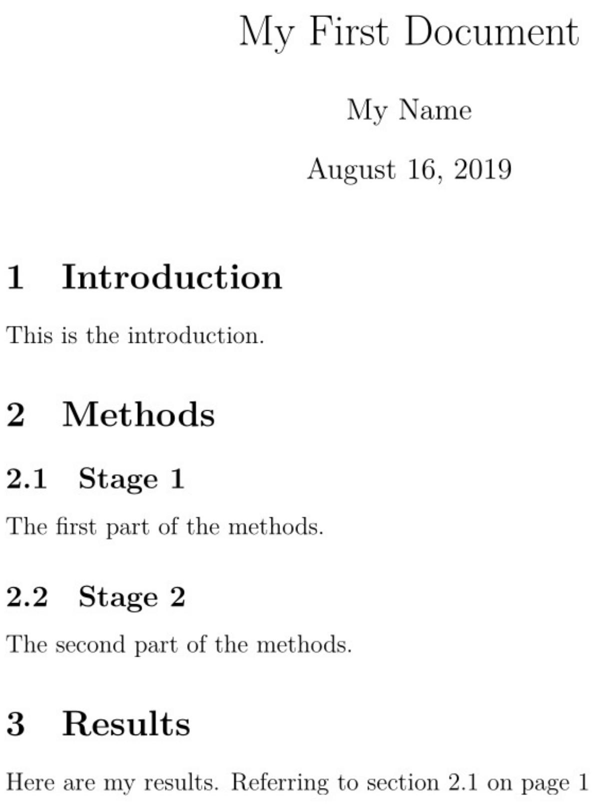
→ 在 `\subsection{Stage 1}` 下面另起一行，输入 `\label{sec1}`。→ 在 **Results** 章节输入 `Referring to section \ref{sec1} on page \pageref{sec1}`。

你的文档会变成这样：

```
\documentclass[a4paper,12pt]{article}

\begin{document}

\title{My First Document}
```

```
\author{My Name}
\date{\today}
\maketitle

\section{Introduction}
This is the introduction.

\section{Methods}

\subsection{Stage 1}
\label{sec1}
The first part of the methods.

\subsection{Stage 2}
The second part of the methods.

\section{Results}
Here are my results. Referring to section \ref{sec1} on page \pageref{sec1}

\end{document}
```

→ 编译并检查 PDF 文档（你可能需要连续编译两次）:



My First Document

My Name

August 16, 2019

1 Introduction

This is the introduction.

2 Methods

2.1 Stage 1

The first part of the methods.

2.2 Stage 2

The second part of the methods.

3 Results

Here are my results. Referring to section 2.1 on page 1

图 3.85 p2

生成目录（TOC）

如果你使用分节命令，那么可以容易地生成一个目录。使用 `\tableofcontents` 在文档中创建目录。通常我们会在标题的后面建立目录。

你可能也想也想更改页码为罗马数字（i,ii,iii）。这会确保文档的正文从第 1 页开始。页码可以使用 `\pagenumber` ing{...} 在阿拉伯数字和罗马数字见切换。

→ 在 \maketitle 之后输入以下内容:

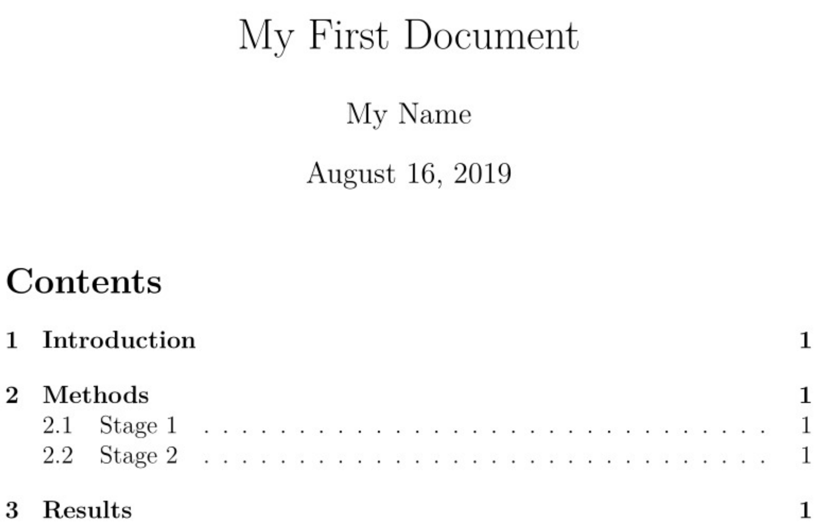
```
\pagenumbering{roman}
\tableofcontents
\newpage
\pagenumbering{arabic}
```

\newpage 命令会另起一个页面，这样我们就可以看到 \pagenumbering 命令带来的影响了。你的文档的前 14 行长这样:

```
\documentclass[a4paper,12pt]{article}

\begin{document}

\title{My First Document}
\author{My Name}
\date{\today}
\maketitle

\pagenumbering{roman}
\tableofcontents
\newpage
\pagenumbering{arabic}
```

→ 编译并核对文档 (可能需要多次编译，下文不赘述)。

文档的第一页长这样:



My First Document

My Name

August 16, 2019

Contents

1 Introduction 1

2 Methods 1

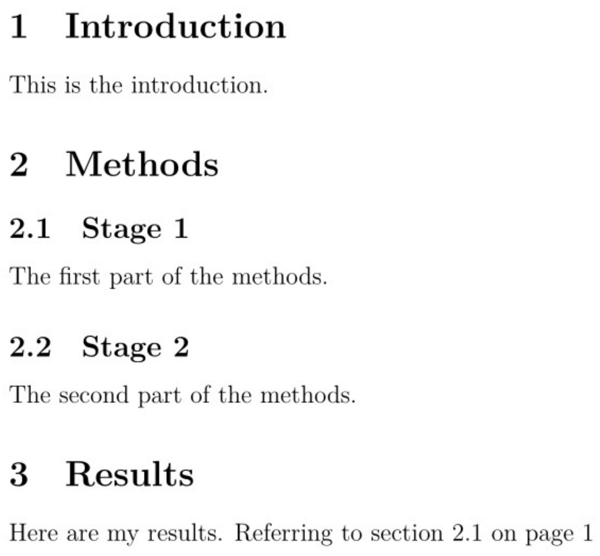
2.1 Stage 1 1

2.2 Stage 2 1

3 Results 1

图 3.86 p3

第二页:



1 Introduction

This is the introduction.

2 Methods

2.1 Stage 1

The first part of the methods.

2.2 Stage 2

The second part of the methods.

3 Results

Here are my results. Referring to section 2.1 on page 1

图 3.87 p4

文字处理

中文字体支持

阅读本文学习 LaTeX 的人，首要学会的自然是 LaTeX 的中文字体支持。事实上，让 LaTeX 支持中文字体有许多方法。在此我们仅给出最**简洁**的解决方案：使用 CTeX 宏包。只需要在文档的前导命令部分添加：

```
\usepackage[UTF8]{ctex}
```

就可以了。在编译文档的时侯使用 `xelatex` 命令，因为它是支持中文字体的。

字体效果

LaTeX 有多种不同的字体效果，在此列举一部分：

```
\textit{words in italics}
\textsl{words slanted}
\textsc{words in smallcaps}
\textbf{words in bold}
\texttt{words in teletype}
\textsf{sans serif words}
\textrm{roman words}
\underline{underlined words}
```

效果如下：

→ 在你的文档中添加更多的文本并尝试各种字体效果。

彩色字体

为了让你的文档支持彩色字体，你需要使用包 (package)。你可以引用很多包来增强 LaTeX 的排版效果。包引用的命令放置在文档的前导命令的位置 (即放在 `\begin{document}` 命令之前)。使用 `\usepackage[options]{package}` 来引用包。其中 **package** 是包的名称，而 **options** 是指定包的特征的一些参数。

words in italics.
words slanted
WORDS IN SMALLCAPS
words in bold
`words in teletype`
sans serif words
roman words
underlined words

图 3.88 p5

使用 `\usepackage{color}` 后，我们可以调用常见的颜色：

red, green, blue, cyan, magenta, yellow, white

图 3.89 p6

使用彩色字体的代码为

```
{\color{colorname}text}
```

其中 **colorname** 是你想要的颜色的名字，**text** 是你的彩色文本内容。注意到示例效果中的黄色与白色是有文字背景色的，这个我们同样可以使用 Color 包中的 `\colorbox` 命令来达到。用法如下：

```
\colorbox{colorname}{text}
```

→ 在 `\begin{document}` 前输入 `\usepackage{color}`。→ 在文档内容中输入 `{\color{red}fire}`。→ 编译并核对 PDF 文档内容。

单词 fire 应该是红色的。

你也可以添加一些参数来调用更多的颜色，甚至自定义你需要的颜色。但这部分超出了本书的内容。如果想要获取更多关于彩色文本的内容请阅读 LaTeX Wikibook 的 Colors 章节[7]。

字体大小

接下来我们列举一些 LaTeX 的字体大小设定命令：

```
normal size words
{\tiny tiny words}
{\scriptsize scriptsize words}
{\footnotesize footnotesize words}
{\small small words}
{\large large words}
{\Large Large words}
{\LARGE LARGE words}
{\huge huge words}
```

效果如下：

→ 尝试为你的文本调整字体大小。

段落缩进

LaTeX 默认每个章节第一段首行顶格，之后的段落首行缩进。如果想要段落顶格，在要顶格的段落前加 `\noindent` 命令即可。如果希望全局所有段落都顶格，在文档的某一位置使用 `\setlength{\parindent}{0pt}` 命令，之

图 3.90 p7

后的所有段落都会顶格。

列表

LaTeX 支持两种类型的列表：有序列表（enumerate）和无序列表（itemize）。列表中的元素定义为 `\item`。列表可以有子列表。

→ 输入下面的内容来生成一个有序列表套无序列表：

```
\begin{enumerate}
\item First thing
\item Second thing
\begin{itemize}
\item A sub-thing
\item Another sub-thing
\end{itemize}
\item Third thing
\end{enumerate}
```

→ 编译并核对 PDF 文档。

列表长这样：

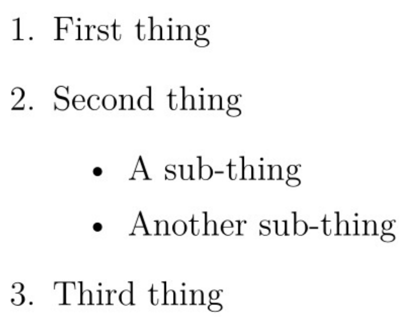


图 3.91 p8

可以使用方括号参数来修改无序列表头的标志。例如，`\item[-]` 会使用一个杠作为标志，你甚至可以使用一个单词，比如 `\item[One]`。

下面的代码：

```
\begin{itemize}
\item[-] First thing
\item[+] Second thing
\begin{itemize}
\item[Fish] A sub-thing
\item[Plants] Another sub-thing
\end{itemize}
\item[Q] Third thing
\end{itemize}
```

生成的效果为

- First thing

+ Second thing

  Fish A sub-thing

  Plants Another sub-thing

Q Third thing

图 3.92 p9

注释和空格

我们使用% 创建一个单行注释，在这个字符之后的该行上的内容都会被忽略，直到下一行开始。

下面的代码：

```
It is a truth universally acknowledged% Note comic irony
in the very first sentence
, that a single man in possession of a good fortune, must
be in want of a wife.
```

生成的结果为

It is a truth universally acknowledgedin the very first sentence , that a single man in possession of a good fortune, must be in want of a wife.

图 3.93 p10

多个连续空格在 LaTeX 中被视为一个空格。多个连续空行被视为一个空行。空行的主要功能是开始一个新的段落。通常来说，LaTeX 忽略空行和其他空白字符，两个反斜杠（\\）可以被用来换行。

→ 尝试在你的文档中添加注释和空行。

如果你想要在你的文档中添加空格，你可以使用 `\vaspace{...}` 的命令。这样可以添加竖着的空格，高度可以指定。如 `\vspace{12pt}` 会产生一个空格，高度等于 12pt 的文字的高度。

特殊字符

下列字符在 LaTeX 中属于特殊字符：

```
# $ % ^ & _ { } ~ \
```

为了使用这些字符，我们需要在他们前面添加反斜杠进行转义：

```
\# \$ \% \^{} \& \_ \{ \} \~{}
```

注意在使用 `^` 和 `~` 字符的时候需要在后面紧跟一对闭合的花括号，否则他们就会被解释为字母的上标，就像 `\^e` 会变成 ê。上面的代码生成的效果如下：

\# \$ % ^ & _ { } ~

图 3.94 p11

注意，反斜杠不能通过反斜杠转义（不然就变成了换行了），使用 `\textbackslash` 命令代替。

→ 输入代码来在你的文档中生成下面内容：

Item #A\642 costs $8 & is sold at a ^ 10% profit.

图 3.95 p12

询问专家或者查看本页面的 源代码[8] 获取帮助。

表格

表格（tabular）命令用于排版表格。LaTeX 默认表格是没有横向和竖向的分割线的——如果你需要，你得手动设定。LaTeX 会根据内容自动设置表格的宽度。下面的代码可以创一个表格：

```
\begin{tabular}{...}
```

省略号会由定义表格的列的代码替换：

- `l` 表示一个左对齐的列；
- `r` 表示一个右对齐的列；
- `c` 表示一个向中对齐的列；
- `|` 表示一个列的竖线；

例如，`{lll}` 会生成一个三列的表格，并且保存向左对齐，没有显式的竖线；`{|l|l|r|}` 会生成一个三列表格，前两列左对齐，最后一列右对齐，并且相邻两列之间有显式的竖线。

表格的数据在 `\begin{tabular}` 后输入：

- `&` 用于分割列；
- `\\` 用于换行；
- `\hline` 表示插入一个贯穿所有列的横着的分割线；
- `\cline{1-2}` 会在第一列和第二列插入一个横着的分割线。

最后使用 `\end{tabular}` 结束表格。举一些例子：

```
\begin{tabular}{|l|l|}
Apples & Green \\
Strawberries & Red \\
Orange & Orange\\
\end{tabular}
```

```
\begin{tabular}{rc}
Apples & Green\\
\hline
Strawberries & Red \\
\cline{1-1}
Oranges & Orange \\
\end{tabular}

\begin{tabular}{|r|l|}
\hline
8 & here's \\
\cline{2-2}
86 & stuff\\
\hline \hline
2008 & now \\
\hline
\end{tabular}
```

效果如下：

Apples	Green
Strawberries	Red
Orange	Orange

Apples	Green
Strawberries	Red
Oranges	Orange

8	here's
86	stuff
2008	now

图 3.96 p13

实践

尝试画出下列表格：

图表

本章介绍如何在 LaTeX 文档中插入图表。这里我们需要引入 **graphicx** 包。图片应当是 PDF，PNG，JPEG 或者 GIF 文件。下面的代码会插入一个名为 myimage 的图片：

```
\begin{figure}[h]
\centering
\includegraphics[width=1\textwidth]{myimage}
\caption{Here is my image}
\label{image-myimage}
```

Item	Quantity	Price($)
Nails	500	0.34
Wooden boards	100	4.00
Bricks	240	11.50

	Year		
City	2006	2007	2008
London	45789	46551	51298
Berlin	34549	32543	29870
Paris	49835	51009	51970

图 3.97 p14

```
\end{figure}
```

`[h]` 是位置参数，**h** 表示把图表近似地放置在这里（如果能放得下）。有其他的选项：**t** 表示放在在页面顶端；**b** 表示放在在页面的底端；**p** 表示另起一页放置图表。你也可以添加一个**!** 参数来强制放在参数指定的位置（尽管这样排版的效果可能不太好）。

`\centering` 将图片放置在页面的中央。如果没有该命令会默认左对齐。使用它的效果是很好的，因为图表的标题也是居中对齐的。

`\includegraphics{...}` 命令可以自动将图放置到你的文档中，图片文件应当与 TeX 文件放在同一目录下。

`[width=1\textwidth]` 是一个可选的参数，它指定图片的宽度——与文本的宽度相同。宽度也可以以厘米为单位。你也可以使用 `[scale=0.5]` 将图片按比例缩小（示例相当于缩小一半）。

`\caption{...}` 定义了图表的标题。如果使用了它，LaTeX 会给你的图表添加「Figure」开头的序号。你可以使用 `\listoffigures` 来生成一个图表的目录。

`\label{...}` 创建了一个可以供你引用的标签。

实践

→ 在你文档的前导命令中添加 `\usepackage{graphicx}`。→ 找到一张图片，放置在你的 **LaTeX course** 文件夹下。→ 在你想要添加图片的地方输入以下内容：

```
\begin{figure}[h!]
\centering
\includegraphics[width=1\textwidth]{ImageFilename}
\caption{My test image}
\end{figure}
```

将 **ImageFilename** 替换为你的文件的名字（不包括后缀）。如果你的文件名有空格，就使用双引号包裹，比如 `"screen 20"`。

→ 编译并核对文件。

公式

使用 LaTeX 的主要原因之一是它可以方便地排版公式。我们使用数学模式来排版公式。

插入公式

你可以使用一对 `$` 来启用数学模式，这可以用于撰写行内数学公式。例如 `$1+2=3$` 的生成效果是 $1+2=3$。

如果你想要行间的公式，可以使用 `$$...$$`（现在我们推荐使用 `\[...\]`，因为前者可能产生不良间距）。例如，`$$1+2=3$$` 的生产效果为

$$1+2=3$$

如果是生成带标号的公式，可以使用 `\begin{equation}...\end{equation}`。例如：

```
\begin{equation}
  1+2=3
\end{equation}
```

生成的效果为：

$$\mathbf{1+2=3} \tag{1}$$

图 3.98 equation

数字 6 代表的是章节的编号，仅当你的文档有设置章节时才会出现，比如 **report** 类型的文档。

使用 `\begin{eqnarray}...\end{eqnarray}` 来撰写一组带标号的公式。例如：

```
\begin{eqnarray}
  a & = & b + c \\
    & = & y - z
\end{eqnarray}
```

生成的效果为

$$\begin{array}{rcll} a & = & b+c & (1) \\ & = & y-z & (2) \end{array}$$

图 3.99 eqnarray

要撰写不标号的公式就在环境标志的后面添加 `*` 字符，如 `{equation*}`，`{eqnarray*}`。

warning

可以发现，使用 `eqnarray` 时，会出现等号周围的空隙过大之类的问题。

可以使用 `amsmath` 宏包中的 `align` 环境：

```
\usepackage{amsmath}
...
\begin{align}
  a & = b + c \\
    & = y - z
\end{align}
```

或在行间公式中使用 `aligned` 环境。它们的名字后面加上星号后，公式就不带标号了。

详见更多阅读中第一篇资料的「4.4 多行公式」。

数学符号

尽管一些基础的符号可以直接键入，但大多数特殊符号需要使用命令来显示。

本书只是数学符号使用的入门教程，LaTeX Wikibook 的数学符号章节是另一个更好更完整的教程。如果想要了解更多关于数学符号的内容请移步。如果你想找到一个特定的符号，可以使用 Detexfiy[9]，它可以识别手写字符。

上标和下标

上标（Powers）使用 `^` 来表示，比如 `$n^2$` 生成的效果为 n^2。

下标（Indices）使用 `_` 表示，比如 `$2_a$` 生成的效果为 2_a。

如果上标或下标的内容包含多个字符，请使用花括号包裹起来。比如 `$b_{a-2}$` 的效果为 b_{a-2}。

分数

分数使用 `\frac{numerator}{denominator}` 命令插入。比如 `$$\frac{a}{3}$$` 的生成效果为

$$\frac{a}{3}$$

分数可以嵌套。比如 `$$\frac{y}{\frac{3}{x}+b}$$` 的生成效果为

$$\frac{y}{\frac{3}{x}+b}$$

根号

我们使用 `\sqrt{...}` 命令插入根号。省略号的内容由被开根的内容替代。如果需要添加开根的次数，使用方括号括起来即可。

例如 `$$\sqrt{y^2}$$` 的生成效果为

$$\sqrt{y^2}$$

而 `$$\sqrt[x]{y^2}$$` 的生成效果为

$$\sqrt[x]{y^2}$$

求和与积分

使用 `\sum` 和 `\int` 来插入求和式与积分式。对于两种符号，上限使用 `^` 来表示，而下限使用 `_` 表示。

`$$\sum_{x=1}^5 y^z$$` 的生成效果为

$$\sum_{x=1}^5 y^z$$

而 `$$\int_a^b f(x)$$` 的生成效果为

$$\int_a^b f(x)$$

希腊字母

我们可以使用反斜杠加希腊字母的名称来表示一个希腊字母。名称的首字母的大小写决定希腊字母的形态。例如

- `$\alpha$`=α
- `$\beta$`=β
- `$\delta, \Delta$`=δ, Δ
- `$\pi, \Pi$`=π, Π
- `$\sigma, \Sigma$`=σ, Σ
- `$\phi, \Phi, \varphi$`=ϕ, Φ, φ
- `$\psi, \Psi$`=ψ, Ψ
- `$\omega, \Omega$`=ω, Ω

实践

→ 撰写代码来生成下列公式：

$$
\begin{aligned}
e &= mc^2 && (1)\\
\pi &= \frac{c}{d} && (2)\\
\frac{d}{dx}e^x &= e^x && (3)\\
\frac{d}{dx}\int_0^\infty f(s)ds &= f(x) && (4)\\
f(x) &= \sum_i 0^\infty \frac{f^{(i)}(0)}{i!}x^i && (5)\\
x &= \sqrt{\frac{x_i}{z}y} && (6)
\end{aligned}
$$

图 3.100 p15

如果需要帮助，可以查看本页面的 源代码[8]。

参考文献

介绍

LaTeX 可以轻松插入参考文献以及目录。本文会介绍如何使用另一个 BibTeX 文件来存储参考文献。

BibTeX 文件类型

BibTeX 文件包含了所有你想要在你文档中引用的文献。它的文件后缀名为 `.bib`。它的名字应设置为你的 TeX 文档的名字。`.bib` 文件是文本文件。你需要将你的参考文献按照下列格式输入：

```
@article{
    Birdetal2001,
    Author = {Bird, R. B. and Smith, E. A. and Bird, D. W.},
    Title = {The hunting handicap: costly signaling in human foraging strategies
},
    Journal = {Behavioral Ecology and Sociobiology},
    Volume = {50},
    Pages = {9-19},
    Year = {2001}
}
```

每一个参考文献先声名它的文献类型（reference type）。示例中使用的是 @article，其他的类型包括 @book，@incollection 用于引用一本书的中的章节，@inproceedings 用于引用会议论文。可以 在此[10] 查看更多支持的类型。

接下来的花括号内首先要列出一个引用键值（citation key）。必须保证你引用的文献的引用键值是不同的。你可以自定义键值串，不过使用第一作者名字加上年分会是一个表义清晰的选择。

接下来的若干行包括文献的若干信息，格式如下：

```
Field name = {field contents},
```

你可以使用 LaTeX 命令来生成特殊的文字效果。比如意大利斜体可以使用 `\emph{Rattus norvegicus}`。

对于需要大写的字母，请用花括号包裹起来。BibTeX 会自动把标题中除第一个字母外所有大写字母替换为小写。比如 `Dispersal in the contemporary United States` 的生成效果为 Dispersal in the contemporary united states，而 `Dispersal in the contemporary {U}nited {S}tates` 的生成效果为 Dispersal in the contemporary United States。

你可以手写 BibTeX 文件，也可以使用软件来生成。

插入文献列表

使用下列命令在文档当前位置插入文献列表：

```
\bibliographystyle{plain}
\bibliography{references}
```

参考文献写在 `references.bib` 里。

参考文献标注

使用 `\cite{citationkey}` 来在你想要引用文献的地方插入一个标注。如果你不希望在正文中插入一个引用标注，但仍想要在文献列表中显示这次引用，使用 `\nocite{citationkey}` 命令。

想要在引用中插入页码信息，使用方括号：`\cite[p. 215]{citationkay}`。

要引用多个文献，使用逗号分隔：`\cite{citation01,citation02,citation03}`。

引用格式

数字标号引用

LaTeX 包含了多种行内数字标号引用的格式：

Plain 方括号包裹数字的形式，如 [1]。文献列表按照第一作者的字母表顺序排列。每一个作者的名字是全称。

Abbrv 与 **plain** 是相同的，但作者的名字是缩写。

Unsrt 与 **plain** 是相同的，但文献列表的排序按照在文中引用的先后顺序排列。

Alpha 与 **plain** 一样，但引用的标注是作者的名字与年份组合在一起，不是数字，如 [Kop10]。

作者日期引用

如果你想使用作者日期的引用，使用 **natbib** 包。它使用 `\citep{...}` 命令来生成一个方括号标注，如 [$Koppe$, 2010]，使用 `\citet{...}` 来生成一个标注，只把年份放到方括号里，如 $Koppe$[2010]。在此[11] 查看它的更多用法。

Natbib 包也有三种格式：**plainnat**，**abbrvnat** 和 **unsrtnat**，他们与 **plain**，**abbrv** 和 **unsrt** 的效果是一样的。

其他引用格式

如果你需要使用不同的格式，你需要在同一个文件夹下创建一个格式文件（`.bst` 文件），引用这个格式的时候使用它的文件名调用 `\bibliographystyle{...}` 命令实现。

实践

→ 在同一文件夹下新建一个同名的 BibTeX 文件，用正确的格式输入参考文献的信息。→ 切换到 TeX 文档，并使用 `\cite`，`\bibliographystyle` 和 `\bibliograph` 命令来引用文献。→ 编译 TeX 文件。→ 切换到 BibTeX 文件，并编译（点击 **Typeset** 按扭）→ 切换到 TeX 文件并编译它**两次**，然后核对 PDF 文档。

更多阅读

- 一份（不太）简短的 LATEX 2 介绍 https://github.com/CTeX-org/lshort-zh-cn/releases/download/v6.02/lshort-zh-cn.pdf[12] 或 112 分钟了解 LaTeX 2 .
- LaTeX Project http://www.latex-project.org/[13] Official website - has links to documentation, information about installing LATEX on your own computer, and information about where to look for help.

- LaTeX Wikibook http://en.wikibooks.org/wiki/LaTeX/[14] Comprehensive and clearly written, although still a work in progress. A downloadable PDF is also available.
- Comparison of TeX Editors on Wikipedia http://en.wikipedia.org/wiki/Comparison_of_TeX_editors[15] Infor mation to help you to choose which L A TEX editor to install on your own computer.
- TeX Live http://www.tug.org/texlive/[16]"An easy way to get up and running with the TeX document produc tion system". Available for Unix and Windows (links to MacTeX for MacOSX users). Includes the TeXworks editor.
- Workbook Source Files http://edin.ac/17EQPM1[17] Download the .tex file and other files needed to compile this workbook.

本文译自 http://www.docs.is.ed.ac.uk/skills/documents/3722/3722-2014.pdf[18], 依据其他文献略有修改。

参考资料与注释

[1] TeX Live

[2] MikTeX

[3] CTeX

[4] TeX 引擎、格式、发行版之介绍

[5] 清华大学 TUNA 镜像站

[6] 官方文档

[7] Colors 章节

[8] 源代码 [8-1] [8-2]

[9] Detexfiy

[10] 在此

[11] 在此

[12] https://github.com/CTeX-org/lshort-zh-cn/releases/download/v6.02/lshort-zh-cn.pdf

[13] http://www.latex-project.org/

[14] http://en.wikibooks.org/wiki/LaTeX/

[15] http://en.wikipedia.org/wiki/Comparison_of_TeX_editors

[16] http://www.tug.org/texlive/

[17] http://edin.ac/17EQPM1

[18] http://www.docs.is.ed.ac.uk/skills/documents/3722/3722-2014.pdf

3.12 Git

note

本页面将着重介绍 Git 这一版本控制系统，与 GitHub 相关的内容，请参考 GitHub 帮助[5] 和 如何参与 - OI Wiki。

Git 是目前使用最广泛的版本控制系统之一。**OI Wiki** 也使用了 Git 作为版本控制系统。

安装

参见 Git - Downloads[6]。

配置

Git 根据配置文件的应用范围，将配置文件分为不同的等级，其中较常用的有两个级别[1]：

1. 适用于当前用户的全局配置文件，该用户操作本系统上的所有仓库时都会查询该配置文件。
2. 适用于当前仓库的配置文件。

当多个配置文件对同一个选项作出设置的时候，局部设置会自动覆盖全局设置。因此如果需要在某个仓库应用特定的设置的话，只需更改该仓库下的特定设置即可，不会对全局设置造成影响。

修改配置文件需要用到 `git config` 命令。

设置用户信息

安装 Git 后，第一件事情就是设置你的用户名和邮箱。这些信息在每次提交时都会用到。

```
$ git config --global user.name "OI Wiki"
$ git config --global user.email oi-wiki@example.com
```

note

这里给出的用户名和邮箱仅供演示。您在根据本页面的内容配置时，请记得将这里的用户名和邮箱改成自己的信息。

这里的 `--global` 表示修改的是全局配置，即该设置对当前用户下的所有仓库均有效。如果不添加 `--global` 选项，则会默认修改当前仓库下的配置文件。

如果想要修改某个仓库的特定设置，只需在该仓库下执行不带 `--global` 的命令即可。

配置编辑器

```
$ git config --global core.editor emacs
```

执行如上命令可以将编辑器更改为 Emacs。

在 Windows 下，Git 的默认编辑器可以在安装 Git 时选择（见前文）。之后若要修改，在 Git Bash 里输入如上命令，将编辑器名换成编辑器的绝对路径，运行命令即可。

显示配置

可以通过 `git config -l` 列出当前已经设置的所有配置参数。使用 `git config --global -l` 可以列出所有全局配置。

仓库操作基础

新建 Git 仓库

新建一个 Git 仓库非常简单，只需在想要建立仓库的文件夹输入如下命令：

```
$ git init
```

Git 将在当前文件夹新建一个 `.git` 文件夹，一个仓库就这样建好了。

如果想把一个仓库克隆到自己的电脑上（比如将 **OI Wiki** 的代码拷贝到本地上进行编辑），采用 `git clone` 命令即可。

```
$ git clone https://github.com/OI-wiki/OI-wiki
```

" 远程仓库的链接"

这里给出的仓库链接是 HTTP(S) 链接，也即我们采用了 HTTP(S) 方式连接到远程仓库。

事实上，连接到远程仓库的方式还有多种。其中使用 ssh 连接到远程仓库的方法更为方便和安全，在「远程仓库的管理」部分我们会简单介绍使用 ssh 连接到远程仓库的方法。

这样，被克隆的仓库的内容就会被储存到当前文件夹下一个与仓库同名的新文件夹。在本例中，当前文件夹下会出现一个名为 `OI-wiki` 的新文件夹。

跟踪文件

在对仓库的文件做出了一些更改后，这些更改需要被纳入到版本管理当中去。

使用 `git status` 命令可以查看当前仓库文件的状态。

举个例子，在一个空仓库中新增了一个 `README.md` 文件后，执行 `git status` 命令的效果如下：

```
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        README.md

nothing added to commit but untracked files present (use "git add" to track)
```

这里的 Untracked files 指的是 Git 之前没有纳入版本跟踪的文件。如果文件没有纳入版本跟踪，对该文件的更改不会被 Git 记录。

执行 `git add < 文件 >` 命令可以将指定的文件纳入到版本跟踪中。

```
$ git add README.md # 将这个文件纳入到版本跟踪中
$ git status
On branch master
```

```
No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   README.md
```

这时 README.md 已经纳入了版本跟踪，放入了暂存区。接下来只需执行 git commit 命令就可以提交这次更改了。

但在进行这一工作之前，让我们先对 README.md 做点小更改。

```
$ vim README.md # 随便更改点东西
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   README.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore -- <file>..." to discard changes in working directory)

        modified:   README.md
```

你会发现 README.md 同时处于暂存区和非暂存区。实际上，是否处于暂存区是对于更改而言的，而不是对于文件而言的，所以对 README.md 的前一次更改已被纳入暂存区，而后一次更改还没有。如果这时候执行 git commit 命令，只有处于暂存区的更改会被提交，而非暂存区的更改，则不会被提交。

Git 给了一条提示，执行 git add README.md 就可以将非暂存区的更改放入暂存区了。

" 一次性将所有更改放入暂存区"

git add 命令会将对指定的文件的更改放入暂存区中。

在多数情况下，用户更期望一次性将所有更改都放入暂存区中，这时候可以应用 git add -A 命令。该命令会将所有更改（包括未被纳入版本跟踪的文件，不包括被忽略的文件）放入暂存区。

如果只需更新已被纳入版本跟踪的文件，而不将未纳入版本跟踪的文件加入暂存区，可以使用 git add -u。

" 忽略文件"

有些时候我们并不希望将一些文件（如可执行文件等）纳入到版本跟踪中。这时候可以在仓库根目录下创建 .gitignore 文件，在该文件里写下想要忽略的文件。Git 将不会将这些文件纳入到版本跟踪中。

例如，*.exe 将自动忽略仓库里的所有扩展名为 .exe 的文件。

现在将非暂存区的文件加入暂存区，将所有更改一并提交（commit）。

```
$ git add README.md
$ git commit # 接下来会弹出编辑器页面，你需要写下 commit 信息
[master (root-commit) f992763] initial commit
 1 file changed, 2 insertions(+)
 create mode 100644 README.md
```

现在重点观察一下这一次 commit 的信息。

`master` 表示当前位于 `master` 分支（关于分支的问题，下文将会详细介绍），`b13c84e` 表示本次提交的 SHA-1 校验和的前几位，后面则是本次提交的信息。

需要特别关注的是这里的 SHA-1 校验码，每个校验码都与某个时刻仓库的一个快照相对应。利用这一特性我们可以访问历史某个时刻的仓库快照，并在该快照上进行更改。

接下来两行则详细说明了本次更新涉及的文件更改。

另外，commit 过程中可以利用几个参数来简化提交过程：

- `-a`：在提交前将所有已跟踪的文件的更改放入暂存区。需要注意的是未被跟踪的文件（新创建的文件）不会被自动加入暂存区，需要用 `git add` 命令手动添加。
- `-m`：该参数后跟提交信息，表示以该提交信息提交本次更改。例如 `git commit -m "fix: typo"` 会创建一条标题为 `fix: typo` 的 commit。

查看提交记录

使用 `git log` 命令可以查看仓库的提交历史记录。

可以看到，提交历史里记录了每次提交时的 SHA-1 校验和，提交的作者，提交时间和 commit 信息。

```
$ git log
commit ae9dd3768a405b348bc6170c7acb8b6cb5fe333e (HEAD -> master)
Author: OI Wiki <oi-wiki@example.com>
Date:   Sun Sep 13 00:30:18 2020 +0800

    feat: update README.md

commit f99276362a3c260d439364c505a7a06859f34bf9
Author: OI Wiki <oi-wiki@example.com>
Date:   Sun Sep 13 00:06:07 2020 +0800

    initial commit
```

分支管理

为什么版本管理中需要分支管理呢？答案主要有两点：

1. 直接更改主分支不仅会使历史记录混乱，也可能会造成一些危险的后果。
2. 通过分支，我们可以专注于当前的工作。如果我们需要完成两个不同的工作，只需开两个分支即可，两个分支间的工作互不干扰。

在 Git 中，简单来说，分支就是指向某个快照的指针。每次提交时，Git 都会为这次提交创建一个快照，并将当前分支的指针移动到该快照。

另外还有一个 HEAD 指针，它指向当前所在的分支。

切换分支的过程，简单来说就是将 HEAD 指针，从指向当前所在的分支，改为指向另外一个分支。在这一过程中，Git 会自动完成文件的更新，使得切换分支后仓库的状态与目标分支指向的快照一致。

分支的创建

利用 `git branch` 命令可以创建分支，`git switch` 命令可以切换分支，`git switch -c` 命令可以创建分支并切换到这个新分支。

```
$ git switch -c dev # 创建一个叫做 dev 的新分支并切换当前分支到 dev
Switched to branch 'dev'
```

```
$ git branch # 查看分支列表
  master
* dev
```

dev 前面的星号代表该仓库的当前分支为 dev，接下来对这个仓库的更改都将记录在这个分支上。

试着创建一个新文件 aplusb.cpp。

```
$ vim aplusb.cpp
$ git add aplusb.cpp
$ git commit -m "feat: add A+B Problem code"
[dev 5da093b] feat: add A+B Problem code
 1 file changed, 7 insertions(+)
 create mode 100644 aplusb.cpp
```

现在切换回 master 分支，这时候文件夹中没有了 aplusb.cpp，一切都回到了刚刚创建 dev 分支时的状态。这时候可以在 master 分支上继续完成其他的工作。

```
$ git switch master
Switched to branch 'master'
$ vim README.md # 对 README 做些小改动
$ git commit -a -m "feat: update README.md"
[master 5ca15f0] feat: update README.md
 1 file changed, 1 insertion(+), 1 deletion(-)
```

下面用一张图来解释刚才的操作过程。

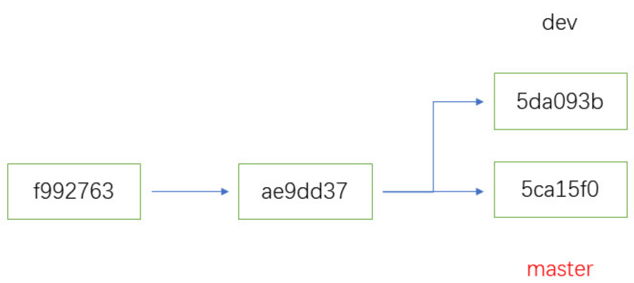


图 3.101

master 分支被标红，表明在这几次操作后，它是当前分支（即 HEAD 指向的位置）。

- 最开始时 master 指向 ae9dd37 这一快照。
- 接下来在 master 所在的位置创建了一个新的 dev 分支，该分支一开始和 master 指向相同位置。
- 在 dev 分支上作了一些更改（创建了 aplusb.cpp），进行了一次提交，本次提交后，dev 分支指向 5da093b 这一快照。
- 切换回 master 分支后，因为 master 分支还指向 ae9dd37，还没有创建 aplusb.cpp，因此仓库中没有这一文件。

- 接下来在 master 分支上进行更改（更新了 README.md），进行了一次提交，master 分支指向了 5ca15f0 这一快照。

分支的合并

当一个分支上的工作已经完成，就可以将这些工作合并到另外一个分支上去。

还是接着上面这个例子，dev 分支的工作已经完成，通过 git merge 命令可以将该分支合并到当前分支（master）上：

```
$ git merge dev
Merge made by the 'recursive' strategy.
 aplusb.cpp | 7 +++++++
 1 file changed, 7 insertions(+)
 create mode 100644 aplusb.cpp
```

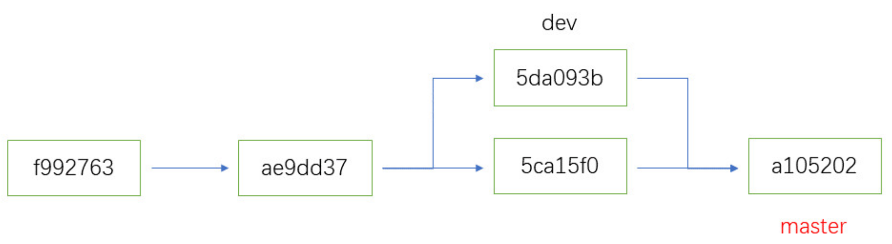


图 3.102

这次合并具体是怎么执行的呢？

在合并之前，master 指向 5ca15f0，而 dev 指向 5da093b，这两个状态并不在一条链上。

Git 会找到这两个状态的最近公共祖先（在上图中是 ae9dd37），并对这三个快照进行一次合并。三个快照合并的结果作为一个新的快照，并将当前分支指向这一快照。

合并过程本身也是一次提交，不过与常规提交不同的是，合并提交有不止一个前驱提交，它是多个提交状态合并后的结果。

在合并完成后，dev 分支就完成了它的使命，这时候可以利用下面的命令删除 dev 分支：

```
$ git branch -d dev # 对于未合并的分支，可以使用 -D 参数强制删除
```

不过合并过程并非总是这么顺利，在某些情况下，合并过程可能会出现冲突，这个问题接下来会讲到。

解决合并冲突

如果在两个分支中，对同一个文件的同一部分进行了不同的更改，Git 就无法自动合并这两个分支，也就是发生了合并冲突。

接着上面的例子，假如你在合并后的 master 分支的基础上，新开了一个 readme-refactor 分支，准备重写一份自述文件。但因为一些疏忽，你同时更改了 readme-refactor 和 master 分支的自述文件。

刚开始自述文件是这样的：

```
# This is a test repo.

This repo includes some c++ codes.
```

在 readme-refactor 分支下的自述文件是这样的：

```
# Code Library

This repo includes some c++ codes.
```

在 master 分支下的自述文件是这样的：

```
# This is a code library.

This repo includes some c++ codes.
```

这时候运行 git merge readme-refactor 命令，Git 提示出现了合并冲突。

执行一下 git status 命令，可以查看是哪些文件引发了冲突。

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      README.md

no changes added to commit (use "git add" and/or "git commit -a")
```

如何解决冲突？对于每个发生了合并冲突的文件，Git 都会在这些文件中加入标准的冲突解决标记。比如这个例子中的 README.md 文件，打开后它长这个样子：

```
<<<<<< HEAD
# This is a code library.
======
# Code Library
>>>>>> readme-refactor

This repo includes some c++ codes.
```

====== 作为分界线将两个分支的内容隔开，<<<<<< HEAD 标记和 ====== 之间的部分是 HEAD 指针（master 分支）的内容，而 ====== 和 >>>>>> readme-refactor 标记之间的部分是 readme-refactor 分支的内容。

通过编辑文本来处理冲突，删除这些冲突标记，保存文件，将这些文件纳入暂存区后提交，就可以解决合并冲突了。

```
$ git add README.md # 将发生冲突的文件纳入暂存区
$ git commit
[master fe92c6b] Merge branch readme-refactor into master
```

其他合并方式

默认情况下，Git 采用 Merge（合并）的方式合并两个分支。使用该方法将分支 B 并入分支 A 时，会将 B 分支的所有 commit 并入 A 分支的提交历史中。

除此以外，Git 还提供了两种合并分支的方式：Squash（压缩）和 Rebase（变基）。

Squash（压缩）

使用 Squash 方式将分支 B 并入分支 A 时，在 B 分支上的所有更改会被合并为一次 commit 提交到 A 分支。

在 `git merge` 中加入 `--squash` 参数即可使用 Squash 方式进行分支合并。

```
git merge <branch> --squash
```

需要注意的是，在执行上述命令后，Git 只会将 B 分支的所有更改存入 A 分支的缓冲区内，接下来还需要执行一次 `git commit` 命令完成合并工作。

使用 Squash 方式合并可以简化 commit 记录，但是会丢失具体到每一次 commit 的信息（每次 commit 的提交者，每次 commit 的更改等等），只留下合并为一个整体的信息（每次 commit 的提交者会以"Co-authored-by" 的形式在提交信息中列出）。但如果是在 GitHub 上进行 Squash and Merge，原有的信息都可以在 Pull Request 中查看。

Rebase（变基）

使用 Rebase 方式将分支 B 并入分支 A 时，在 B 分支上的每一次 commit 都会单独添加到 A 分支，而不再像 Merge 方式那样创建一个合并 commit 来合并两个分支的内容[2]。

首先，切换到 B 分支，接下来将 B 分支变基到 A 分支：

```
git checkout B
git rebase A
```

现在切回到 A 分支，再执行一次 `git merge` 命令，即可完成将 B 分支的内容合并到 A 分支的工作。

```
git checkout A
git merge B
```

使用 Rebase 完成合并可以让提交历史线性化，在适当的场景下正确地使用 Rebase 可以达到比 Merge 更好的效果。但是这样做会改变提交历史，在进行 Rebase 时和 Rebase 后再进行相关合并操作时都会增加出现冲突的可能，如果操作不当可能反而会使提交历史变得杂乱。因此，如果对 Rebase 操作没有充分的了解，不建议使用。

管理远程仓库

在本地完成更改后，你可能会需要将这些更改推送到 GitHub 等 Git 仓库托管平台上。托管在这些平台上的仓库就归属于远程仓库的范畴——你可以从这些仓库中获取信息，也可以将你作出的更改推送到远程仓库上。与其他人的协作往往离不开远程仓库，因此学会管理远程仓库很有必要。

远程仓库的查看

使用 `git remote` 命令可以查看当前仓库的远程仓库列表。

如果当前仓库是克隆来的，那么应该会有一个叫做 origin 的远程仓库，它的链接就是克隆时用的链接。

```
$ git remote
origin
```

如果要查看某个远程仓库的详细信息的话，可以这样操作：

```
$ git remote show origin
* remote origin
  Fetch URL: git@github.com:OI-wiki/OI-wiki.git
  Push  URL: git@github.com:OI-wiki/OI-wiki.git
  HEAD branch: master
  Remote branches:
    git              tracked
    master           tracked
  ...
```

远程仓库的配置

执行 `git remote add <name> <url>` 命令可以添加一个名字为 `name`，链接为 `url` 的远程仓库。

执行 `git remote rename <oldname> <newname>` 可以将名字为 `oldname` 的远程仓库改名为 `newname`。

执行 `git remote rm <name>` 可以删除名字为 `name` 的远程仓库。

执行 `git remote get-url <name>` 可以查看名字为 `name` 的远程仓库的链接。

执行 `git remote set-url <name> <newurl>` 可以将名字为 `name` 的远程仓库的链接更改为 `newurl`。

从远程仓库获取更改

在远程仓库中，其他人可能会推送一些更改，执行 `git fetch` 命令可以将这些更改获取到本地。

```
$ git fetch <remote-name> # 获取 <remote-name> 的更改
```

需要注意的是，`git fetch` 命令只会获取远程仓库的更改，而不会将这些更改合并到本地仓库中。如果需要将这些更改进行合并，可以使用 `git pull` 命令。在默认情况下，`git pull` 相当于 `git fetch` 后 `git merge FETCH_HEAD`。

```
$ git pull <remote-name> <branch> # 获取 <remote-name> 的更改，然后将这些更改合并
到 HEAD
```

将更改推送到远程仓库

当你完成了一些更改之后，使用 `git push` 命令可以将这些更改推送到远程仓库。

```
$ git push <remote> <from>:<to> # 将本地 <from> 分支的更改推送至 <remote> 的 <to
> 分支
```

根据远程仓库的要求，可能会要求你输入远程仓库账户的用户名和密码。

需要注意的是，你的更改能成功推送，需要满足两个条件：你拥有向这个仓库（分支）的写入权限，且你的这个分支比远程仓库的相应分支新（可以理解为没有人在你进行更改的这段时间进行了推送）。当远程分支有当前分支没有的新更改时，可以执行 `git pull` 命令完成合并再提交。

如果你需要强制将本地分支的更改推送到远程仓库的话，可以加入 `-f` 参数。此时**远程仓库的提交历史会被本地的提交历史覆盖**，因此该命令应谨慎使用。更好的选择是使用 `--force-with-lease` 参数，该参数仅在远程仓库没有更新时才会进行覆盖。需要注意的是，此处「更新」是相对于上一次 fetch 而言的，如果使用了 VS Code 提供的 Auto Fetch 功能，可能会没有注意到更新而使 `--force-with-lease` 和 `-f` 一样危险。

追踪远程分支

通过将一个本地分支设定为追踪远程分支，可以方便地查看本地分支与远程分支的差别，并能简化与远程分支交互时的操作。

在开始追踪前，你需要先执行 `git fetch <remote-name>` 将远程仓库的信息抓取到本地。

接下来执行 `git switch <remote-branch>`，会在本地自动创建名字为 `<remote-branch>` 的新分支，并设定该分支自动追踪相应的远程分支。

> note
>
> 需要注意，只有当本地不存在该分支，且恰好只有一个远程分支的名字与该分支匹配时，Git 才会自动创建该分支且设定其追踪相应的远程分支。

这时候执行 `git status` 命令，会提示当前分支与远程分支之间的差别。

因为设定了本地分支追踪的远程分支，向远程分支推送的命令也被简化了。只需要执行 `git push` 命令，在本地分支上作出的更改就能被推送至其追踪的远程分支。

对于本地已有的分支，设定其对应的远程追踪分支也很容易。只需在当前分支下执行 `git branch -u <remote-name>/<remote-branch>`，就可以设定当前的本地分支追踪 `<remote-name>/<remote-branch>` 这一远程分支。

使用 ssh 连接

与 HTTP(S) 相比，使用 ssh 连接到远程仓库更为方便安全。

在使用 ssh 连接到远程仓库之前，需要先在本地添加 ssh 密钥。接下来需要将本地添加的 ssh 密钥的**公钥**上传到远程仓库账户。

考虑到本文主要是给 **OI Wiki** 的贡献者提供一个使用 Git 的教程，这里直接给出 GitHub Docs 提供的教程[7]，供各位读者参考。

完成以上步骤后，你就可以通过 ssh 连接到远程仓库了。下面就是一条通过 ssh 连接 clone **OI Wiki** 仓库的命令：

```
$ git clone git@github.com:OI-wiki/OI-wiki.git
```

将更改推送至远程仓库的过程与使用 HTTP(S) 连接类似。但使用 ssh 连接可以免去验证远程仓库帐号密码的过程。

Git GUI Tools

对于不熟悉命令行的同学，纯命令行的 Git 的上手难度可能会偏高，而借助 GUI 工具可以一定程度上降低 Git 的上手难度。此外，相比于命令行，GUI 工具在查看 diff 以及 log 时在体验上有一定程度的提高。

Git 本身自带有 GUI，市面上也有很多优秀的 Git GUI 工具，例如针对 Windows 用户的 TortoiseGit[3]，支持 Windows 和 Mac 的 Sourcetree[4] 等。

这里简单介绍一下 TortoiseGit 的使用。下载并安装好 TortoiseGit 之后，在本地仓库的目录下，单击鼠标右键，在右键菜单中就可以看到 Git 的各个功能。

详细的使用方法这里不再赘述，可以参考官网里的使用文档或者通过搜索引擎学习，例如 TortoiseGit Manual[8]。

很多 GUI 工具都有官方中文支持，例如 Git Desktop 以及 TortoiseGit。但是还是会有部分翻译看起来较为变扭，推荐使用英文版本。

外部链接

- Git Reference[9]
- Pro Git Book[10]
- Learn Git Branching[11]

图 3.103 TortoiseGit Example

参考资料与注释

[1] 事实上 Git 还有一个针对系统上每一个用户及系统上所有仓库的通用配置文件，该配置文件覆盖范围最广，等级在用户配置文件之上。因为该配置实践中较少使用，这里不再展开。

[2] Pro Git Book 中提供了可视化的 Rebase 过程图，借助图片读者可以更好地理解 Rebase 的机制。

[3] TortoiseGit

[4] Sourcetree

[5] GitHub 帮助

[6] Git - Downloads

[7] GitHub Docs 提供的教程

[8] TortoiseGit Manual

[9] Git Reference

[10] Pro Git Book

[11] Learn Git Branching

第 4 章

语言基础

4.1 语言基础简介

本章将会介绍编程相关的知识，包括 C++ 从入门到进阶教程和一些其它语言的简介。

程序是算法与数据结构的载体，是解决 OI 问题的工具。

在 OI 中，最常用的编程语言是 C++。

学习编程是学习 OI 最基础的部分。

4.2 C++ 基础

4.2.1 Hello, World!

环境配置

工欲善其事，必先利其器。

集成开发环境

IDE 操作较为简单，一般入门玩家会选用 IDE 来编写代码。在竞赛中最常见的是 Dev-C++（如果考试环境是 Windows 系统，一般也会提供这一 IDE）。

编译器

Windows

推荐使用 GNU 编译器。需要去 MinGW Distro[1] 下载 MinGW 并安装。此外 Windows 下也可以选择 Microsoft Visual C++ 编译器[2]，需要去 Visual Studio 页面[3] 下载安装。

macOS

在终端中执行：

```
xcode-select --install
```

Linux

使用 `g++ -v` 来检查是否安装过 `g++`。

使用如下命令可以安装：

```
sudo apt update && sudo apt install g++
```

在命令行中编译代码

熟练之后也有玩家会使用更灵活的命令行来编译代码，这样就不依赖 IDE 了，而是使用自己熟悉的文本编辑器编写代码。

```
g++ test.cpp -o test -lm
```

`g++` 是 C++ 语言的编译器（C 语言的编译器为 `gcc`），`-o` 用于指定可执行文件的文件名，编译选项 `-lm` 用于链接数学库 `libm`，从而使得使用 `math.h` 的代码可以正常编译运行。

注：C++ 程序不需要 `-lm` 即可正常编译运行。历年 NOI/NOIP 试题的 C++ 编译选项中都带着 `-lm`，故这里也一并加上。

第一份代码

通过这样一个示例程序来展开 C++ 入门之旅吧 ~

注：请在编写前注意开启英文输入法。

C++ 语言

```cpp
#include <iostream>  // 引用头文件

using namespace std;

// 引入命名空间（相关阅读 https://oi-wiki.org/lang/namespace/#using ）

int main() {                  // 定义 main 函数
  cout << "Hello, world!";  // 输出 Hello, world!
  return 0;                 // 返回 0，结束 main 函数
}
```

C 语言

```c
#include <stdio.h>  // 引用头文件

int main() {                  // 定义 main 函数
  printf("Hello, world!");  // 输出 Hello, world!
  return 0;                 // 返回 0，结束 main 函数
}
```

注意：C 语言在这里仅做参考，C++ 基本兼容 C 语言，并且拥有许多新的功能，可以让选手在赛场上事半功倍。具体请见 C++ 与其他常用语言区别

参考资料与注释

[1] MinGW Distro

[2] Microsoft Visual C++ 编译器

[3] Visual Studio 页面

4.2.2 C++ 语法基础

代码框架

如果你不想深究背后的原理，初学时可以直接将这个「框架」背下来：

```cpp
#include <cstdio>
#include <iostream>

int main() {
  // do something...
  return 0;
}
```

" 什么是 include？"

`#include` 其实是一个预处理命令，意思为将一个文件「放」在这条语句处，被「放」的文件被称为头文件。也就是说，在编译时，编译器会「复制」头文件 `iostream` 中的内容，「粘贴」到 `#include <iostream>` 这条语句处。这样，你就可以使用 `iostream` 中提供的 `std::cin`、`std::cout`、`std::endl` 等对象了。

如果你学过 C 语言，你会发现目前我们接触的 C++ 中的头文件一般都不带 `.h` 后缀，而那些 C 语言中的头文件 `xx.h` 都变成了 `cxx`，如 `stdio.h` 变成了 `cstdio`。因为 C++ 为了和 C 保持兼容，都直接使用了 C 语言中的头文件，为了区分 C++ 的头文件和 C 的头文件，使用了 `c` 前缀。

一般来说，应当根据你需要编写的 C++ 程序的需要来确定你要 `#include` 哪些头文件。但如果你 `#include` 了多余的头文件，只会增加编译时间，几乎不会对运行时间造成影响。目前我们只接触到了 `iostream` 和 `cstdio` 两个头文件，如果你只需要 `scanf` 和 `printf`，就可以不用 `#include <iostream>`。

可以 `#include` 自己写的头文件吗？答案是，可以。

你可以自己写一个头文件，如：`myheader.h`。然后，将其放到和你的代码相同的目录里，再 `#include "myheader.h"` 即可。需要注意的是，自定义的头文件需要使用引号而非尖括号。当然，你也可以使用编译命令 `-I <header_file_path>` 来告诉编译器在哪找头文件，就不需要将头文件放到和代码相同的目录里了。

" 什么是 `main()`？"

可以理解为程序运行时就会执行 `main()` 中的代码。

实际上，`main` 函数是由系统或外部程序调用的。如，你在命令行中调用了你的程序，也就是调用了你程序中的 `main` 函数（在此之前先完成了全局 变量 的构造）。

最后的 `return 0;` 表示程序运行成功。默认情况下，程序结束时返回 0 表示一切正常，否则返回值表示错误代码（在 Windows 下这个错误代码的十六进制可以通过 Windows Error Codes 网站[1] 进行查询）。这个值返回给谁呢？其实就是调用你写的程序的系统或外部程序，它会在你的程序结束时接收到这个返回值。如果不写 `return` 语句的话，程序正常结束默认返回值也是 0。

在 C 或 C++ 中，程序的返回值不为 0 会导致运行时错误（RE）。

注释

在 C++ 代码中，注释有两种写法：

1. 行内注释

   以 `//` 开头，行内位于其后的内容全部为注释。
2. 注释块

   以 `/*` 开头，`*/` 结尾，中间的内容全部为注释，可以跨行。

注释对程序运行没有影响，可以用来解释程序的意思，还可以在让某段代码不执行（但是依然保留在源文件里）。

在工程开发中，注释可以便于日后维护、他人阅读。

在 OI 中，很少有人写许多注释，但注释可以便于在写代码的时候理清思路，或者便于日后复习。而且，如果要写题解、教程的话，适量的注释可以便于读者阅读，理解代码的意图。希望各位同学能养成写注释的好习惯。

输入与输出

cin 与 cout

```
#include <iostream>

int main() {
  int x, y;                          // 声明变量
  std::cin >> x >> y;                // 读入 x 和 y
  std::cout << y << std::endl << x;  // 输出 y，换行，再输出 x
  return 0;                          // 结束主函数
}
```

"什么是变量？"

可以参考 变量 页面。

"什么是 std？"

std 是 C++ 标准库所使用的**命名空间**。使用命名空间是为了避免重名。

关于命名空间的详细知识，可以参考 命名空间 页面。

scanf 与 printf

`scanf` 与 `printf` 其实是 C 语言提供的函数。大多数情况下，它们的速度比 `cin` 和 `cout` 更快，并且能够方便地控制输入输出格式。

"读入输出优化"

`cin`/`cout` 和 `scanf`/`prinf` 的具体差别和读入输出优化，请参考 读入 、 输出优化 页面。

```
#include <cstdio>

int main() {
  int x, y;
  scanf("%d%d", &x, &y);   // 读入 x 和 y
  printf("%d\n%d", y, x);  // 输出 y，换行，再输出 x
  return 0;
}
```

其中，`%d` 表示读入/输出的变量是一个有符号整型（`int` 型）的变量。

类似地：

1. `%s` 表示字符串。
2. `%c` 表示字符。
3. `%lf` 表示双精度浮点数 (`double`)。
4. `%lld` 表示长整型 (`long long`)。根据系统不同，也可能是 `%I64d`。
5. `%u` 表示无符号整型 (`unsigned int`)。
6. `%llu` 表示无符号长整型 (`unsigned long long`)，也可能是 `%I64u`。

除了类型标识符以外，还有一些控制格式的方式。许多都不常用，选取两个常用的列举如下：

1. `%1d` 表示长度为 1 的整型。在读入时，即使没有空格也可以逐位读入数字。在输出时，若指定的长度大于数字的位数，就会在数字前用空格填充。若指定的长度小于数字的位数，就没有效果。
2. `%.6lf`，用于输出，保留六位小数。

这两种运算符的相应地方都可以填入其他数字，例如 `%.3lf` 表示保留三位小数。

"「双精度浮点数」,「长整型」是什么"

这些表示变量的类型。和上面一样，会留到 变量 中统一讲解。

" 为什么 `scanf` 中有 `&` 运算符？"

在这里，`&` 实际上是取址运算符，返回的是变量在内存中的地址。而 scanf 接收的参数就是变量的地址。具体可能要在 指针 才能完全清楚地说明，现在只需要记下来就好了。

" 什么是 `\n`？"

`\n` 是一种**转义字符**，表示换行。

转义字符用来表示一些无法直接输入的字符，如由于字符串字面量中无法换行而无法直接输入的换行符，由于有特殊含义而无法输入的引号，由于表示转义字符而无法输入的反斜杠。

常用的转义字符有：

1. `\t` 表示制表符。
2. `\\` 表示 `\`。
3. `\"` 表示 `"`。
4. `\0` 表示空字符，用来表示 C 风格字符串的结尾。
5. `\r` 表示回车。Linux 中换行符为 `\n`，Windows 中换行符为 `\r\n`。在 OI 中，如果输出需要换行，使用 `\n` 即可。但读入时，如果使用逐字符读入，可能会由于换行符造成一些问题，需要注意。例如，`gets` 将 `\n` 作为字符串结尾，这时候如果换行符是 `\r\n`，`\r` 就会留在字符串结尾。
6. 特殊地，`%%` 表示 `%`，只能用在 `printf` 或 `scanf` 中，在其他字符串字面量中只需要简单使用 `%` 就好了。

" 什么是字面量？"

「字面量」是在代码里直接作为一个值的程序段，例如 `3` 就是一个 `int` 字面量，`'c'` 就是一个 char 字面量。我们上面写的程序中的 `"hello world"` 也是一个字符串字面量。

不加解释、毫无来由的字面量又被称为「魔术数」（magic number），如果代码需要被人阅读的话，这是一种十分不被推荐的行为。

一些扩展内容

C++ 中的空白字符

在 C++ 中，所有空白字符（空格、制表符、换行），多个或是单个，都被视作是一样的。（当然，引号中视作字符串的一部分的不算。）

因此，你可以自由地使用任何代码风格（除了行内注释、字符串字面量与预处理命令必须在单行内），例如：

```
/* clang-format off */

#include <iostream>

 int
```

```
    main(){
int/**/x, y;  std::cin
>> x >>y;
                std::cout <<
          y  <<std::endl
     << x


          ;

    return      0;     }
```

当然，这么做是不被推荐的。

一种也被广泛使用但与 **OI Wiki** 要求的码风不同的代码风格：

```
/* clang-format off */

#include <iostream>

int main()
{
    int x, y;

    std::cin >> x >> y;
    std::cout << y << std::endl << x;

    return 0;
}
```

#define 命令

#define 是一种预处理命令，用于定义宏，本质上是文本替换。例如：

```
#include <iostream>
#define n 233

// n 不是变量，而是编译器会将代码中所有 n 文本替换为 233，但是作为标识符一部分的
// n 的就不会被替换，如 fn 不会被替换成 f233，同样，字符串内的也不会被替换

int main() {
  std::cout << n;  // 输出 233
  return 0;
}
```

" 什么是标识符？"

标识符就是可以用作变量名的一组字符。例如，abcd 和 abc1 都是合法的标识符，而 1a 和 c+b 都不是合法的标识符。

标识符由英文字母、下划线开头，中间只允许出现英文字母、下划线和数字。值得注意的是，关键字（如 int,for,if）不能用作标识符。

" 什么是预处理命令？"

预处理命令就是预处理器所接受的命令，用于对代码进行初步的文本变换，比如文件包含操作 #include 和处理宏 #define 等，对 GCC 而言，默认不会保留预处理阶段的输出 .i 文件。可以用 -E 选项保留输出文件。

宏可以带参数，带参数的宏可以像函数一样使用：

```cpp
#include <iostream>
#define sum(x, y) ((x) + (y))
#define square(x) ((x) * (x))

int main() {
  std::cout << sum(1, 2) << ' ' << 2 * sum(3, 5) << std::endl;  // 输出 3 16
}
```

但是带参数的宏和函数有区别。因为宏是文本替换，所以会引发许多问题。如：

```cpp
#include <iostream>
#define sum(x, y) x + y
// 这里应当为 #define sum(x, y) ((x) + (y))
#define square(x) ((x) * (x))

int main() {
  std::cout << sum(1, 2) << ' ' << 2 * sum(3, 5) << std::endl;
  // 输出为 3 11，因为 #define 是文本替换，后面的语句被替换为了 2 * 3 + 5
  int i = 1;
  std::cout << square(++i) << ' ' << i;
  // 输出未定义，因为 ++i 被执行了两遍
  // 而同一个语句中多次修改同一个变量是未定义行为（有例外）
}
```

使用 `#define` 是有风险的（由于 `#define` 作用域是整个程序，因此可能导致文本被意外地替换，需要使用 `#undef` 及时取消定义），因此应谨慎使用。较为推荐的做法是：使用 `const` 限定符声明常量，使用函数代替宏。

但是，在 OI 中，`#define` 依然有用武之处（以下两种是不被推荐的用法，会降低代码的规范性）：

1. `#define int long long`+`signed main()`。通常用于避免忘记开 long long 导致的错误，或是调试时排除忘开 long long 导致错误的可能性。（也可能导致增大常数甚至 TLE，或者因为爆空间而 MLE）
2. `#define For(i, l, r) for (int i = (l); i <= (r); ++i)`、`#define pb push_back`、`#define mid ((l + r) / 2)`，用于减短代码长度。

不过，`#define` 也有优点，比如结合 `#ifdef` 等预处理指令有奇效，比如：

```cpp
#ifdef LINUX
// code for linux
#else
// code for other OS
#endif
```

可以在编译的时候通过 `-DLINUX` 来控制编译出的代码，而无需修改源文件。这还有一个优点：通过 `-DLINUX` 编译出的可执行文件里并没有其他操作系统的代码，那些代码在预处理的时候就已经被删除了。

`#define` 还能使用 `#`、`##` 运算符，极大地方便调试。

参考资料与注释

[1] Windows Error Codes 网站

4.2.3 变量

数据类型

C++ 的类型系统由如下几部分组成：

1. 基础类型（括号内为代表关键词/代表类型）
   (a) 无类型/void 型（void）
   (b)（C++11 起）空指针类型（std::nullptr_t）
   (c) 算术类型
      i. 整数类型（int）
      ii. 布尔类型/bool 型（bool）
      iii. 字符类型（char）
      iv. 浮点类型（float,double）
2. 复合类型[2]

布尔类型

一个 bool 类型的变量取值只可能为两种：true 和 false。

一般情况下，一个 bool 类型变量占有 1 字节（一般情况下，1 字节 =8 位）的空间。

"C 语言的布尔类型"

C 语言最初是没有布尔类型的，直到 C99 时才引入 _Bool 关键词作为布尔类型，其被视作无符号整数类型。

note

C 语言的 bool 类型从 C23 起不再使用整型的零与非零值定义，而是定义为足够储存 true 和 false 两个常量的类型。

为方便使用，stdbool.h 中提供了 bool,true,false 三个宏，定义如下：

```c
#define bool _Bool
#define true 1
#define false 0
```

这些宏于 C23 中移除，并且 C23 起引入 true,false 和 bool 作为关键字，同时保留 _Bool 作为替代拼写形式[1]。

整数类型

用于存储整数。最基础的整数类型是 int.

" 注意"

由于历史原因，C++ 中布尔类型和字符类型会被视作特殊的整型。

在几乎所有的情况下都**不应该**将除 signed char 和 unsigned char 之外的字符类型作为整型使用。

整数类型一般按位宽有 5 个梯度：char,short,int,long,long long.

C++ 标准保证 1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)

由于历史原因，整数类型的位宽有多种流行模型，为解决这一问题，C99/C++11 引入了定宽整数类型。

"int 类型的大小"

在 C++ 标准中，规定 int 的位数**至少**为 16 位。

事实上在现在的绝大多数平台，int 的位数均为 32 位。

对于 int 关键字，可以使用如下修饰关键字进行修饰：

符号性：

- `signed`：表示带符号整数（默认）；
- `unsigned`：表示无符号整数。

大小：

- `short`：表示**至少** 16 位整数；
- `long`：表示**至少** 32 位整数；
- （C++11 起）`long long`：表示**至少** 64 位整数。

下表给出在**一般情况下**，各整数类型的位宽和表示范围大小（少数平台上一些类型的表示范围可能与下表不同）：

类型名	等价类型	位宽（C++ 标准）	位宽（常见）	位宽（较罕见）
`signed char`	`signed char`	8	-	-
`unsigned char`	`unsigned char`	8	-	-
`short`,`short int`,`signed short`,`signed short int`	`short int`	≥ 16	16	-
`unsigned short`,`unsigned short int`	`unsigned short int`	≥ 16	16	-
`int`,`signed`,`signed int`	`int`	≥ 16	32	16（常见于 Win16 API）
`unsigned`,`unsigned int`	`unsigned int`	≥ 16	32	16（常见于 Win16 API）
`long`,`long int`,`signed long`,`signed long int`	`long int`	≥ 32	32	64（常见于 64 位 Linux、macOS）
`unsigned long`,`unsigned long int`	`unsigned long int`	≥ 32	32	64（常见于 64 位 Linux、macOS）
`long long`,`long long int`,`signed long long`,`signed long long int`	`long long int`	≥ 64	64	-
`unsigned long long`,`unsigned long long int`	`unsigned long long int`	≥ 64	64	-

当位宽为 x 时，有符号类型的表示范围为 $-2^{x-1} \sim 2^{x-1}-1$，无符号类型的表示范围为 $0 \sim 2^x-1$. 具体而言，有下表：

位宽	表示范围
8	有符号：$-2^7 \sim 2^7-1$，无符号：$0 \sim 2^8-1$
16	有符号：$-2^{15} \sim 2^{15}-1$，无符号：$0 \sim 2^{16}-1$
32	有符号：$-2^{31} \sim 2^{31}-1$，无符号：$0 \sim 2^{32}-1$
64	有符号：$-2^{63} \sim 2^{63}-1$，无符号：$0 \sim 2^{64}-1$

" 等价的类型表述"

在不引发歧义的情况下，允许省略部分修饰关键字，或调整修饰关键字的顺序。这意味着同一类型会存在多种等价表述。

例如 `int`，`signed`，`int signed`，`signed int` 表示同一类型，而 `unsigned long` 和 `unsigned long int`

表示同一类型。

另外，一些编译器实现了扩展整数类型，如 GCC 实现了 128 位整数：有符号版的 `__int128_t` 和无符号版的 `__uint128_t`，如果您在比赛时想使用这些类型，**请仔细阅读比赛规则**以确定是否允许或支持使用扩展整数类型。

" 注意"

STL 不一定对扩展整数类型有足够的支持，故使用扩展整数类型时需格外小心。

" 示例代码"

```
#include <cmath>
#include <iostream>

int f1(int n) {
  return abs(n);  // Good
}

int f2(int n) {
  return std::abs(n);  // Good
}

__int128_t f3(__int128_t n) {
  return abs(n);  // Bad
}

// Wrong
// __int128_t f4(__int128_t n) {
//   return std::abs(n);
// }

int main() {
  std::cout << "f1: " << f1(-42) << std::endl;
  std::cout << "f2: " << f2(-42) << std::endl;
  // std::cout << "f3: " << f3(-42) << std::endl; // Wrong
  // std::cout << "f4: " << f4(-42) << std::endl; // Wrong
  return 0;
}
```

以上示例代码存在如下问题：

1. `__int128_t f3(__int128_t)` 中使用的是 C 风格的绝对值函数，其签名为 `int abs(int)`，故 `n` 首先会强制转换为 `int`，然后才会调用 `abs` 函数。
2. `__int128_t f4(__int128_t)` 中使用的是 C++ 风格的绝对值函数，其并没有签名为 `__int128_t std::abs(__int128_t)` 的函数重载，所以无法通过编译。
3. C++ 的流式输出不支持 `__int128_t` 与 `__uint128_t`。

以下是一种解决方案：

" 修正后的代码"

```
#include <cmath>
#include <iostream>

__int128_t abs(__int128_t n) { return n < 0 ? -n : n; }
```

```
std::ostream &operator<<(std::ostream &os, __uint128_t n) {
  if (n > 9) os << n / 10;
  os << (int)(n % 10);
  return os;
}

std::ostream &operator<<(std::ostream &os, __int128_t n) {
  if (n < 0) {
    os << '-';
    n = -n;
  }
  return os << (__uint128_t)n;
}

int f1(int n) { return abs(n); }

int f2(int n) { return std::abs(n); }

__int128_t f3(__int128_t n) { return abs(n); }

int main() {
  std::cout << "f1: " << f1(-42) << std::endl;
  std::cout << "f2: " << f2(-42) << std::endl;
  std::cout << "f3: " << f3(-42) << std::endl;
}
```

字符类型

分为「窄字符类型」和「宽字符类型」，由于算法竞赛几乎不会用到宽字符类型，故此处仅介绍窄字符类型。

窄字符型位数一般为 8 位，实际上底层存储方式仍然是整数，一般通过 ASCII 编码[7] 实现字符与整数的一一对应，有如下三种：

- `signed char`：有符号字符表示的类型，表示范围在 $-128 \sim 127$ 之间。
- `unsigned char`：无符号字符表示的类型，表示范围在 $0 \sim 255$ 之间。
- `char` 拥有与 `signed char` 或 `unsigned char` 之一相同的表示和对齐，但始终是独立的类型。

  `char` 的符号性取决于编译器和目标平台：ARM 和 PowerPC 的默认设置通常没有符号，而 x86 与 x64 的默认设置通常有符号。

  GCC 可以在编译参数中添加 `-fsigned-char` 或 `-funsigned-char` 指定将 `char` 视作 `signed char` 或 `unsigned char`，其他编译器请参照文档。需要注意指定与架构默认值不同的符号有可能会破坏 ABI，造成程序无法正常工作。

" 注意"

与其他整型不同，`char`、`signed char`、`unsigned char` 是**三种不同的类型**。

一般来说 `signed char,unsigned char` 不应用来存储字符，绝大多数情况下，这两种类型均被视作整数类型。

浮点类型

用于存储「实数」（注意并不是严格意义上的实数，而是实数在一定规则下的近似），包括以下三种：

- `float`：单精度浮点类型。如果支持就会匹配 IEEE-754 binary32 格式。
- `double`：双精度浮点类型。如果支持就会匹配 IEEE-754 binary64 格式。

- `long double`：扩展精度浮点类型。如果支持就会匹配 IEEE-754 binary128 格式，否则如果支持就会匹配 IEEE-754 binary64 扩展格式，否则匹配某种精度优于 binary64 而值域至少和 binary64 一样好的非 IEEE-754 扩展浮点格式，否则匹配 IEEE-754 binary64 格式。

浮点格式	位宽	最大正数	精度位数
IEEE-754 binary32 格式	32	3.4×10^{38}	$6 \sim 9$
IEEE-754 binary64 格式	64	1.8×10^{308}	$15 \sim 17$
IEEE-754 binary64 扩展格式	≥ 80	$\geq 1.2 \times 10^{4932}$	$\geq 18 \sim 21$
IEEE-754 binary128 格式	128	1.2×10^{4932}	$33 \sim 36$

> IEEE-754 浮点格式的最小负数是最大正数的相反数。

因为 `float` 类型表示范围较小，且精度不高，实际应用中常使用 `double` 类型表示浮点数。

另外，浮点类型可以支持一些特殊值：

- 无穷（正或负）：`INFINITY`.
- 负零：`-0.0`，例如 `1.0 / 0.0 == INFINITY`,`1.0 / -0.0 == -INFINITY`.
- 非数（NaN）：`std::nan`,`NAN`，一般可以由 `0.0 / 0.0` 之类的运算产生。它与任何值（包括自身）比较都不相等，C++11 后可以使用 `std::isnan` 判断一个浮点数是不是 NaN.

无类型

`void` 类型为无类型，与上面几种类型不同的是，不能将一个变量声明为 `void` 类型。但是函数的返回值允许为 `void` 类型，表示该函数无返回值。

空指针类型

请参阅指针的 对应章节

定宽整数类型

C++11 起提供了定宽整数的支持，具体如下：

- `<cstdint>`：提供了若干定宽整数的类型和各定宽整数类型最大值、最小值等的宏常量。
- `<cinttypes>`：为定宽整数类型提供了用于 `std::fprintf` 系列函数和 `std::fscanf` 系列函数的格式宏常量。

定宽整数有如下几种：

- `intN_t`: 宽度**恰为** N 位的有符号整数类型，如 `int32_t`.
- `int_fastN_t`: 宽度**至少**有 N 位的**最快的**有符号整数类型，如 `int_fast32_t`.
- `int_leastN_t`: 宽度**至少**有 N 位的**最小的**有符号整数类型，如 `int_least32_t`.

无符号版本只需在有符号版本前加一个字母 u 即可，如 `uint32_t`,`uint_least8_t`.

标准规定必须实现如下 16 种类型：

`int_fast8_t`,`int_fast16_t`,`int_fast32_t`,`int_fast64_t`,

`int_least8_t`,`int_least16_t`,`int_least32_t`,`int_least64_t`,

`uint_fast8_t`,`uint_fast16_t`,`uint_fast32_t`,`uint_fast64_t`,

`uint_least8_t`,`uint_least16_t`,`uint_least32_t`,`uint_least64_t`.

绝大多数编译器在此基础上都实现了如下 8 种类型：

int8_t,int16_t,int32_t,int64_t,

uint8_t,uint16_t,uint32_t,uint64_t.

在实现了对应类型的情况下，C++ 标准规定必须实现表示对应类型的最大值、最小值、位宽的宏常量，格式为将类型名末尾的 _t 去掉后转大写并添加后缀：

- _MAX 表示最大值，如 INT32_MAX 即为 int32_t 的最大值。
- _MIN 表示最小值，如 INT32_MIN 即为 int32_t 的最小值。

" 注意"

定宽整数类型本质上是普通整数类型的类型别名，所以混用定宽整数类型和普通整数类型可能会影响跨平台编译，例如：

" 示例代码"

```cpp
#include <algorithm>
#include <cstdint>
#include <iostream>

int main() {
  long long a;
  int64_t b;
  std::cin >> a >> b;
  std::cout << std::max(a, b) << std::endl;
  return 0;
}
```

int64_t 在 64 位 Windows 下一般为 long long int, 而在 64 位 Linux 下一般为 long int, 所以这段代码在使用 64 位 Linux 下的 GCC 时不能通过编译，而使用 64 位 Windows 下的 MSVC 时可以通过编译，因为 std::max 要求输入的两个参数类型必须相同。

此外，C++17 起在 <limits> 中提供了 std::numeric_limits 类模板，用于查询各种算数类型的属性，如最大值、最小值、是否是整形、是否有符号等。

```cpp
#include <cstdint>
#include <limits>

std::numeric_limits<int32_t>::max();  // int32_t 的最大值, 2'147'483'647
std::numeric_limits<int32_t>::min();  // int32_t 的最小值, -2'147'483'648

std::numeric_limits<double>::min();  // double 的最小值, 约为 2.22507e-308
std::numeric_limits<double>::epsilon();  // 1.0 与 double 的下个可表示值的差,
                                         // 约为 2.22045e-16
```

类型转换

在一些时候（比如某个函数接受 int 类型的参数，但传入了 double 类型的变量），我们需要将某种类型，转换成另外一种类型。

C++ 中类型的转换机制较为复杂，这里主要介绍对于基础数据类型的两种转换：数值提升和数值转换。

数值提升

数值提升过程中，值本身保持不变。

note

C 风格的可变参数域在传值过程中会进行默认参数提升。如:

" 示例代码"

```
#include <stdarg.h>
#include <stdio.h>

void test(int tot, ...) {
  va_list valist;
  int i;

  // 初始化可变参数列表
  va_start(valist, tot);

  for (i = 0; i < tot; ++i) {
    // 获取第 i 个变量的值
    double xx = va_arg(valist, double);  // Correct
    // float xx = va_arg(valist, float); // Wrong

    // 输出第 i 个变量的底层存储内容
    printf("i = %d, value = 0x%016llx\n", i, *(long long *)(&xx));
  }

  // 清理可变参数列表的内存
  va_end(valist);
}

int main() {
  float f;
  double fd, d;
  f = 123.;   // 0x42f60000
  fd = 123.;  // 0x405ec00000000000
  d = 456.;   // 0x407c800000000000
  test(3, f, fd, d);
}
```

在调用 `test` 时，`f` 提升为 `double`，从而底层存储内容和 `fd` 相同，输出为

```
i = 0, value = 0x405ec00000000000
i = 1, value = 0x405ec00000000000
i = 2, value = 0x407c800000000000
```

若将 `double xx = va_arg(valist, double);` 改为 `float xx = va_arg(valist, float);`，GCC 应该给出一条类似下文的警告：

```
In file included from test.c:2:
test.c: In function ‘test’:
test.c:14:35: warning: ‘float’ is promoted to ‘double’ when passed through ‘...’
  14 |          float xx = va_arg(valist, float);
     |                                   ^
test.c:14:35: note: (so you should pass ‘double’ not ‘float’ to ‘va_arg’)
test.c:14:35: note: if this code is reached, the program will abort
```

此时的程序将会在输出前终止。

这一点也能解释为什么 `printf` 的 `%f` 既能匹配 `float` 也能匹配 `double`。

整数提升

小整数类型（如 `char`）的纯右值可转换成较大整数类型（如 `int`）的纯右值。

具体而言，算术运算符不接受小于 `int` 的类型作为它的实参，而在左值到右值转换后，如果适用就会自动实施整数提升。

具体地，有如下规则：

- 源类型为 `signed char`、`signed short / short` 时，可提升为 `int`。
- 源类型为 `unsigned char`、`unsigned short` 时，若 `int` 能保有源类型的值范围，则可提升为 `int`，否则可提升为 `unsigned int`。(`C++20` 起 `char8_t` 也适用本规则)
- `char` 的提升规则取决于其底层类型是 `signed char` 还是 `unsigned char`。
- `bool` 类型可转换到 `int`：`false` 变为 `0`，`true` 变为 `1`。
- 若目标类型的值范围包含源类型，且源类型的值范围不能被 `int` 和 `unsigned int` 包含，则源类型可提升为目标类型。[3]

" 注意"

`char->short` 不是数值提升，因为 `char` 要优先提升为 `int / unsigned int`，之后是 `int / unsigned int->short`，不满足数值提升的条件。

如（以下假定 `int` 为 32 位，`unsigned short` 为 16 位，`signed char` 和 `unsigned char` 为 8 位，`bool` 为 1 位）

- `(signed char)'\0' - (signed char)'\xff'` 会先将 `(signed char)'\0'` 提升为 `(int)0`、将 `(signed char)'\xff'` 提升为 `(int)-1`，再进行 `int` 间的运算，最终结果为 `(int)1`。
- `(unsigned char)'\0' - (unsigned char)'\xff'` 会先将 `(unsigned char)'\0'` 提升为 `(int)0`、将 `(unsigned char)'\xff'` 提升为 `(int)255`，再进行 `int` 间的运算，最终结果为 `(int)-255`。
- `false - (unsigned short)12` 会先将 `false` 提升为 `(int)0`、将 `(unsigned short)12` 提升为 `(int)12`，再进行 `int` 间的运算，最终结果为 `(int)-12`。

浮点提升

位宽较小的浮点数可以提升为位宽较大的浮点数（例如 `float` 类型的变量和 `double` 类型的变量进行算术运算时，会将 `float` 类型变量提升为 `double` 类型变量），其值不变。

数值转换

数值转换过程中，值可能会发生改变。

" 注意"

数值提升优先于数值转换。如 `bool->int` 时是数值提升而非数值转换。

整数转换

- 如果目标类型为位宽为 x 的无符号整数类型，则转换结果是原值 $\mod 2^x$ 后的结果。
  - 若目标类型位宽大于源类型位宽：
    * 若源类型为有符号类型，一般情况下需先进行符号位扩展再转换。

      如

      · 将 `(short)-1`（`(short)0b1111'1111'1111'1111`）转换为 `unsigned int` 类型时，先进行符号位扩展，得到 `0b1111'1111'1111'1111'1111'1111'1111'1111`，再进行整数转换，结果为 `(unsigned int)4'294'967'295`（`(unsigned int)0b1111'1111'1111'1111'1111'1111'1111'1111`）。

  - 将 `(short)32'767`（`(short)0b0111'1111'1111'1111`）转换为 `unsigned int` 类型时，先进行符号位扩展，得到 `0b0000'0000'0000'0000'0111'1111'1111'1111`，再进行整数转换，结果为 `(unsigned int)32'767`（`(unsigned int)0b0000'0000'0000'0000'0111'1111'1111'1111`）。

  * 若源类型为无符号类型，则需先进行零扩展再转换。

    如将 `(unsigned short)65'535`（`(unsigned short)0b1111'1111'1111'1111`）转换为 `unsigned int` 类型时，先进行零扩展，得到 `0b0000'0000'0000'0000'1111'1111'1111'1111`，再进行整数转换，结果为 `(unsigned int)65'535`（`(unsigned int)0b0000'0000'0000'0000'1111'1111'1111'1111`）。

  - 若目标类型位宽不大于源类型位宽，则需先截断再转换。

    如将 `(unsigned int)4'294'967'295`（`(unsigned int)0b1111'1111'1111'1111'1111'1111'1111'1111`）转换为 `unsigned short` 类型时，先进行截断，得到 `0b1111'1111'1111'1111`，再进行整数转换，结果为 `(unsigned short)65'535`（`(unsigned short)0b1111'1111'1111'1111`）。

- 如果目标类型为位宽为 x 的带符号整数类型，则**一般情况下**，转换结果可以认为是原值 $\bmod 2^x$ 后的结果。[4]

  例如将 `(unsigned int)4'294'967'295`（`(unsigned int)0b1111'1111'1111'1111'1111'1111'1111'1111`）转换为 `short` 类型时，结果为 `(short)-1`（`(short)0b1111'1111'1111'1111`）。

- 如果目标类型是 `bool`，则是布尔转换。
- 如果源类型是 `bool`，则 `false` 转为对应类型的 0，`true` 转为对应类型的 1。

浮点转换

位宽较大的浮点数转换为位宽较小的浮点数，会将该数舍入到目标类型下最接近的值。

浮点整数转换

- 浮点数转换为整数时，会舍弃浮点数的全部小数部分。

  如果目标类型是 `bool`，则是布尔转换。

- 整数转换为浮点数时，会舍入到目标类型下最接近的值。

  如果该值不能适应到目标类型中，那么行为未定义。

  如果源类型是 `bool`，那么 `false` 转换为零，而 `true` 转换为一。

布尔转换

将其他类型转换为 `bool` 类型时，零值转换为 `false`，非零值转换为 `true`。

定义变量

简单地说[5]，定义一个变量，需要包含类型说明符（指明变量的类型），以及要定义的变量名。

例如，下面这几条语句都是变量定义语句。

```
int oi;
double wiki;
char org = 'c';
```

在目前我们所接触到的程序段中，定义在花括号包裹的地方的变量是局部变量，而定义在没有花括号包裹的地方的变量是全局变量。实际有例外，但是现在不必了解。

定义时没有初始化值的全局变量会被初始化为 0。而局部变量没有这种特性，需要手动赋初始值，否则可能引起难以发现的 bug。

变量作用域

作用域是变量可以发挥作用的代码块。

全局变量的作用域，自其定义之处开始[6]，至文件结束位置为止。

局部变量的作用域，自其定义之处开始，至代码块结束位置为止。

由一对大括号括起来的若干语句构成一个代码块。

```
int g = 20;  // 定义全局变量

int main() {
  int g = 10;         // 定义局部变量
  printf("%d\n", g);  // 输出 g
  return 0;
}
```

如果一个代码块的内嵌块中定义了相同变量名的变量，则内层块中将无法访问外层块中相同变量名的变量。

例如上面的代码中，输出的 g 的值将是 10。因此为了防止出现意料之外的错误，请尽量避免局部变量与全局变量重名的情况。

常量

常量是固定值，在程序执行期间不会改变。

常量的值在定义后不能被修改。定义时加一个 `const` 关键字即可。

```
const int a = 2;
a = 3;
```

如果修改了常量的值，在编译环节就会报错：`error: assignment of read-only variable ‘a’`。

参考资料与注释

1. Working Draft, Standard for Programming Language C++[8]
2. 类型 - cppreference.com[3]
3. C 语言的 算术类型 - cppreference.com[9]
4. 基础类型 - cppreference.com[10]
5. 定宽整数类型 （C++11 起） - cppreference.com[11]
6. William Kahan (1 October 1997).”Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic”[12].
7. 隐式转换 - cppreference.com[13]
8. 声明 - cppreference[3]
9. 作用域 - cppreference.com[14]

[1] 参见 https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3054.pdf

[2] 包括数组类型、引用类型、指针类型、类类型、函数类型等。由于本篇文章是面向初学者的，故不在本文做具体介绍。具体请参阅 类型 - cppreference.com

[3] 不包含宽字符类型、位域和枚举类型，详见 整型转换 - cppreference。

[4] 自 C++20 起生效。C++20 前结果是实现定义的。详见 整型转换 - cppreference。

[5] 定义一个变量时，除了类型说明符之外，还可以包含其他说明符。详见 声明 - cppreference。

[6] 更准确的说法是 声明点。

[7] ASCII 编码

[8] Working Draft, Standard for Programming Language C++

[9] 算术类型 - cppreference.com

[10] 基础类型 - cppreference.com

[11] 定宽整数类型（C++11 起）- cppreference.com

[12] ”Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic”

[13] 隐式转换 - cppreference.com

[14] 作用域 - cppreference.com

4.2.4 运算

Authors: Ir1d, aofall, Marcythm

算术运算符

运算符	功能
`+`（单目）	正
`-`（单目）	负
`*`（双目）	乘法
`/`	除法
`%`	取模
`+`（双目）	加法
`-`（双目）	减法

” 单目与双目运算符”

单目运算符（又称一元运算符）指被操作对象只有一个的运算符，而双目运算符（又称二元运算符）的被操作对象有两个。例如 `1 + 2` 中加号就是双目运算符，它有 `1` 和 `2` 两个被操作数。此外 C++ 中还有唯一的一个三目运算符 `?:` 。

算术运算符中有两个单目运算符（正、负）以及五个双目运算符（乘法、除法、取模、加法、减法），其中单目运算符的优先级最高。

其中取模运算符 `%` 意为计算两个整数相除得到的余数，即求余数。

而 `-` 为双目运算符时做减法运算符，如 `2-1` ；为单目运算符时做负值运算符，如 `-1` 。

使用方法如下

op=x-y*z

得到的 op 的运算值遵循数学中加减乘除的优先规律，首先进行优先级高的运算，同优先级按运算的结合性运算，括号提高优先级。

算术运算中的类型转换

对于双目算术运算符，当参与运算的两个变量类型相同时，不发生 类型转换，运算结果将会用参与运算的变量的类型容纳，否则会发生类型转换，以使两个变量的类型一致。转换的规则参见 类型转换。

例如，对于一个整型（int）变量 x 和另一个双精度浮点型（double）类型变量 y：

- x/3 的结果将会是整型；
- x/3.0 的结果将会是双精度浮点型；
- x/y 的结果将会是双精度浮点型；
- x*1/3 的结果将会是整型；
- x*1.0/3 的结果将会是双精度浮点型；

位运算符

运算符	功能
~	逐位非
&（双目）	逐位与
‘	‘
^	逐位异或
<<	逐位左移
>>	逐位右移

位操作的意义请参考 位运算 页面。需要注意的是，位运算符的优先级低于普通的算数运算符。

自增/自减运算符

有时我们需要让变量进行增加 1（自增）或者减少 1（自减），这时自增运算符 ++ 和自减运算符 -- 就派上用场了。

自增/自减运算符可放在变量前或变量后面，在变量前称为前缀，在变量后称为后缀，单独使用时前缀后缀无需特别区别，如果需要用到表达式的值则需注意，具体可看下面的例子。详细情况可参考 引用 介绍的例子部分。

```
i = 100;

op1 = i++;  // op1 = 100，先 op1 = i，然后 i = i + 1

i = 100;

op2 = ++i;  // op2 = 101，先 i = i + 1，然后赋值 op2

i = 100;

op3 = i--;  // op3 = 100，先赋值 op3，然后 i = i - 1
```

```
i = 100;

op4 = --i;  // op4 = 99，先 i = i - 1，然后赋值 op4
```

复合赋值运算符

复合赋值运算符实际上是表达式的缩写形式。可分为复合算术运算符 `+=`、`-=`、`*=`、`/=`、`%=` 和复合位运算符 `&=`、`|=`、`^=`、`<<=`、`>>=`。

例如，`op = op + 2` 可写为 `op += 2`，`op = op - 2` 可写为 `op -= 2`，`op= op * 2` 可写为 `op *= 2`。

条件运算符

条件运算符可以看作 `if` 语句的简写，`a ? b : c` 中如果表达式 `a` 成立，那么这个条件表达式的结果是 `b`，否则条件表达式的结果是 `c`。

比较运算符

运算符	功能
`>`	大于
`>=`	大于等于
`<`	小于
`<=`	小于等于
`==`	等于
`!=`	不等于

其中特别需要注意的是要将等于运算符 `==` 和赋值运算符 `=` 区分开来，这在判断语句中尤为重要。

`if (op=1)` 与 `if (op==1)` 看起来类似，但实际功能却相差甚远。第一条语句是在对 op 进行赋值，若赋值为非 0 时为真值，表达式的条件始终是满足的，无法达到判断的作用；而第二条语句才是对 `op` 的值进行判断。

逻辑运算符

运算符	功能
`&&`	逻辑与
‘	
`!`	逻辑非

```
Result = op1 && op2;  // 当 op1 与 op2 都为真时则 Result 为真

Result = op1 || op2;  // 当 op1 或 op2 其中一个为真时则 Result 为真

Result = !op1;  // 当 op1 为假时则 Result 为真
```

内建的运算符 `&&` 和 `||` 进行短路求值（若在求值第一个操作数后结果已知，则不求值第二个），重载的运算符无此特性，并始终对两个操作数都进行求值。

逗号运算符

逗号运算符可将多个表达式分隔开来，被分隔开的表达式按从左至右的顺序依次计算，整个表达式的值是最后的表达式的值。逗号表达式的优先级在所有运算符中的优先级是**最低**的。

```
exp1, exp2, exp3;  // 最后的值为 exp3 的运算结果。

Result = 1 + 2, 3 + 4, 5 + 6;
//得到 Result 的值为 3 而不是 11，因为赋值运算符 "="
//的优先级比逗号运算符高，先进行了赋值运算才进行逗号运算。

Result = (1 + 2, 3 + 4, 5 + 6);

// 若要让 Result 的值得到逗号运算的结果则应将整个表达式用括号提高优先级，此时
// Result 的值才为 11。
```

成员访问运算符

运算符	功能
`[]`	数组下标
`.`	对象成员
`&`（单目）	取地址/获取引用
`*`（单目）	间接寻址/解引用
`->`	指针成员

这些运算符用来访问对象的成员或者内存，除了最后一个运算符外上述运算符都可被重载。与 `&`，`*` 和 `->` 相关的内容请阅读 指针 和 引用 教程。这里还省略了两个很少用到的运算符 `.*` 和 `->*`，其具体用法可以参见 C++ 语言手册[1]。

```
auto result1 = v[1];  // 获取 v 中下标为 2 的对象
auto result2 = p.q;   // 获取 p 对象的 q 成员
auto result3 = p -> q;  // 获取 p 指针指向的对象的 q 成员，等价于 (*p).q
auto result4 = &v;      // 获取指向 v 的指针
auto result5 = *v;      // 获取 v 指针指向的对象
```

C++ 运算符优先级总表

来自 C++ 运算符优先级 - cppreference[2]，有修改。

运算符	描述	例子	可重载性
第一级别			
`::`	作用域解析符	`Class::age = 2;`	不可重载
第二级别			
`++`	后自增运算符	`for (int i = 0; i < 10; i++) cout << i;`	可重载

运算符	描述	例子	可重载性
`--`	后自减运算符	`for (int i = 10; i > 0; i--) cout << i;`	可重载
`type()` `type{}`	强制类型转换	`unsigned int a = unsigned(3.14);`	可重载
`()`	函数调用	`isdigit('1')`	可重载
`[]`	数组数据获取	`array[4] = 2;`	可重载
`.`	对象型成员调用	`obj.age = 34;`	不可重载
`->`	指针型成员调用	`ptr->age = 34;`	可重载
第三级别 （从右向左结合）			
`++`	前自增运算符	`for (i = 0; i < 10; ++i) cout << i;`	可重载
`--`	前自减运算符	`for (i = 10; i > 0; --i) cout << i;`	可重载
`+`	正号	`int i = +1;`	可重载
`-`	负号	`int i = -1;`	可重载
`!`	逻辑取反	`if (!done)` …	可重载
`~`	按位取反	`flags = ~flags;`	可重载
`(type)`	C 风格强制类型转换	`int i = (int) floatNum;`	可重载
`*`	指针取值	`int data = *intPtr;`	可重载
`&`	值取指针	`int *intPtr = &data;`	可重载
`sizeof`	返回类型内存	`int size = sizeof floatNum; int size = sizeof(float);`	不可重载
`new`	动态元素内存分配	`long *pVar = new long; MyClass *ptr = new MyClass(args);`	可重载
`new []`	动态数组内存分配	`long *array = new long[n];`	可重载
`delete`	动态析构元素内存	`delete pVar;`	可重载
`delete []`	动态析构数组内存	`delete [] array;`	可重载
第四级别			
`.*`	类对象成员引用	`obj.*var = 24;`	不可重载
`->*`	类指针成员引用	`ptr->*var = 24;`	可重载
第五级别			
`*`	乘法	`int i = 2 * 4;`	可重载
`/`	除法	`float f = 10.0 / 3.0;`	可重载
`%`	取余数（模运算）	`int rem = 4 % 3;`	可重载
第六级别			
`+`	加法	`int i = 2 + 3;`	可重载

运算符	描述	例子	可重载性
`-`	减法	`int i = 5 - 1;`	可重载
第七级别			
`<<`	位左移	`int flags = 33 << 1;`	可重载
`>>`	位右移	`int flags = 33 >> 1;`	可重载
第八级别			
`<=>`	三路比较运算符	`if ((i <=> 42) < 0) ...`	可重载
第九级别			
`<`	小于	`if (i < 42) ...`	可重载
`<=`	小于等于	`if (i <= 42) ...`	可重载
`>`	大于	`if (i > 42) ...`	可重载
`>=`	大于等于	`if (i >= 42) ...`	可重载
第十级别			
`==`	等于	`if (i == 42) ...`	可重载
`!=`	不等于	`if (i != 42) ...`	可重载
第十一级别			
`&`	位与运算	`flags = flags & 42;`	可重载
第十二级别			
`^`	位异或运算	`flags = flags ^ 42;`	可重载
第十三级别			
‘	‘	位或运算	‘flags = flags
第十四级别			
`&&`	逻辑与运算	`if (conditionA && conditionB) ...`	可重载
第十五级别			
‘		‘	逻辑或运算
第十六级别 （从右向左结合）			
`? :`	条件运算符	`int i = a > b ? a : b;`	不可重载
`throw`	异常抛出	`throw EClass("Message");`	不可重载
`=`	赋值	`int a = b;`	可重载
`+=`	加赋值运算	`a += 3;`	可重载
`-=`	减赋值运算	`b -= 4;`	可重载
`*=`	乘赋值运算	`a *= 5;`	可重载

运算符	描述	例子	可重载性
`/=`	除赋值运算	`a /= 2;`	可重载
`%=`	模赋值运算	`a %= 3;`	可重载
`<<=`	位左移赋值运算	`flags <<= 2;`	可重载
`>>=`	位右移赋值运算	`flags >>= 2;`	可重载
`&=`	位与赋值运算	`flags &= new_flags;`	可重载
`^=`	位异或赋值运算	`flags ^= new_flags;`	可重载
‘	=‘	位或赋值运算	‘flags
第十七级别			
`,`	逗号分隔符	`for (i = 0, j = 0; i < 10; i++, j++) ...`	可重载

需要注意的是，表中并未列出 `const_cast`、`static_cast`、`dynamic_cast`、`reinterpret_cast`、`typeid`、`sizeof...`、`noexcept` 及 `alignof` 等运算符，因为它们的使用形式与函数调用相同，不会出现歧义。

参考资料与注释

[1] C++ 语言手册

[2] C++ 运算符优先级 - cppreference

4.2.5 流程控制语句

分支

一个程序默认是按照代码的顺序执行下来的，有时我们需要选择性的执行某些语句，这时候就需要分支的功能来实现。选择合适的分支语句可以提高程序的效率。

if 语句

基本 if 语句

以下是基本 if 语句的结构。

```
if (条件) {
  主体;
}
```

if 语句通过对条件进行求值，若结果为真（非 0），执行语句，否则不执行。

如果主体中只有单个语句的话，花括号可以省略。

if...else 语句

```
if (条件) {
  主体 1;
} else {
  主体 2;
```

```
}
```

if...else 语句和 if 语句类似，else 不需要再写条件。当 if 语句的条件满足时会执行 if 里的语句，if 语句的条件不满足时会执行 else 里的语句。同样，当主体只有一条语句时，可以省略花括号。

else if 语句

```
if (条件 1) {
  主体 1;
} else if (条件 2) {
  主体 2;
} else if (条件 3) {
  主体 3;
} else {
  主体 4;
}
```

else if 语句是 if 和 else 的组合，对多个条件进行判断并选择不同的语句分支。在最后一条的 else 语句不需要再写条件。例如，若条件 1 为真，执行主体 1，条件 3 为真而条件 1 和条件 2 都为假，执行主体 3，所有的条件都为假才执行主体 4。

实际上，这一个语句相当于第一个 if 的 else 分句只有一个 if 语句，就将花括号省略之后放在一起了。如果条件相互之间是并列关系，这样写可以让代码的逻辑更清晰。

在逻辑上，大约相当于这一段话：

> 解一元二次方程的时候，方程的根与判别式的关系：
>
> - 如果 $(\Delta < 0)$ 方程无解；
> - 否则，如果 $(\Delta = 0)$ 方程有两个相同的实数解；
> - 否则方程有两个不相同的实数解；

switch 语句

```
switch (选择句) {
  case 标签 1:
    主体 1;
  case 标签 2:
    主体 2;
  default:
    主体 3;
}
```

switch 语句执行时，先求出选择句的值，然后根据选择句的值选择相应的标签，从标签处开始执行。其中，选择句必须是一个整数类型表达式，而标签都必须是整数类型的常量。例如：

```
int i = 1;  // 这里的 i 的数据类型是整型 ，满足整数类型的表达式的要求

switch (i) {
  case 1:
    cout << "OI WIKI" << endl;
}
```

```
char i = 'A';

// 这里的 i 的数据类型是字符型 ，但 char
```

```cpp
// 也是属于整数的类型，满足整数类型的表达式的要求
switch (i) {
  case 'A':
    cout << "OI WIKI" << endl;
}
```

switch 语句中还要根据需求加入 break 语句进行中断，否则在对应的 case 被选择之后接下来的所有 case 里的语句和 default 里的语句都会被运行。具体例子可看下面的示例。

```cpp
char i = 'B';

switch (i) {
  case 'A':
    cout << "OI" << endl;
    break;

  case 'B':
    cout << "WIKI" << endl;

  default:
    cout << "Hello World" << endl;
}
```

以上代码运行后输出的结果为 `WIKI` 和 `Hello World`，如果不想让下面分支的语句被运行就需要 break 了，具体例子可看下面的示例。

```cpp
char i = 'B';

switch (i) {
  case 'A':
    cout << "OI" << endl;
    break;

  case 'B':
    cout << "WIKI" << endl;
    break;

  default:
    cout << "Hello World" << endl;
}
```

以上代码运行后输出的结果为 WIKI，因为 break 的存在，接下来的语句就不会继续被执行了。最后一个语句不需要 break，因为下面没有语句了。

处理入口编号不能重复，但可以颠倒。也就是说，入口编号的顺序不重要。各个 case（包括 default）的出现次序可任意。例如：

```cpp
char i = 'B';

switch (i) {
  case 'B':
    cout << "WIKI" << endl;
    break;

  default:
    cout << "Hello World" << endl;
    break;
```

```cpp
  case 'A':
    cout << "OI" << endl;
}
```

switch 的 case 分句中也可以选择性的加花括号。不过要注意的是，如果需要在 switch 语句中定义变量，花括号是必须要加的。例如：

```cpp
char i = 'B';

switch (i) {
  case 'A': {
    int i = 1, j = 2;
    cout << "OI" << endl;
    ans = i + j;
    break;
  }

  case 'B': {
    int qwq = 3;
    cout << "WIKI" << endl;
    ans = qwq * qwq;
    break;
  }

  default: {
    cout << "Hello World" << endl;
  }
}
```

" **如何理解** switch"

在上文中，用了大量「case 分句」，「case 子句」等用语，实际上，在底层实现中，switch 相当于一组跳转语句。也因此，有 Duff's Device 这种奇技淫巧，希望了解的人可以自行学习。

循环

有时，我们需要做一件事很多遍，为了不写过多重复的代码，我们需要循环。

有时，循环的次数不是一个常量，那么我们无法将代码重复多遍，必须使用循环。

for 语句

以下是 for 语句的结构：

```cpp
for (初始化; 判断条件; 更新) {
  循环体;
}
```

执行顺序：

e.g. 读入 n 个数：

```cpp
for (int i = 1; i <= n; ++i) {
  cin >> a[i];
}
```

for 语句的三个部分中，任何一个部分都可以省略。其中，若省略了判断条件，相当于判断条件永远为真。

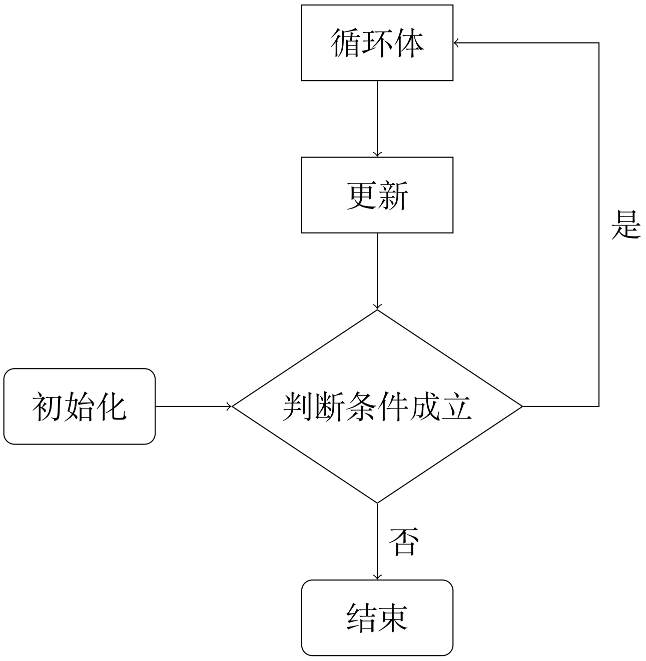


图 4.1

while 语句

以下是 while 语句的结构：

```
while (判断条件) {
  循环体;
}
```

执行顺序：

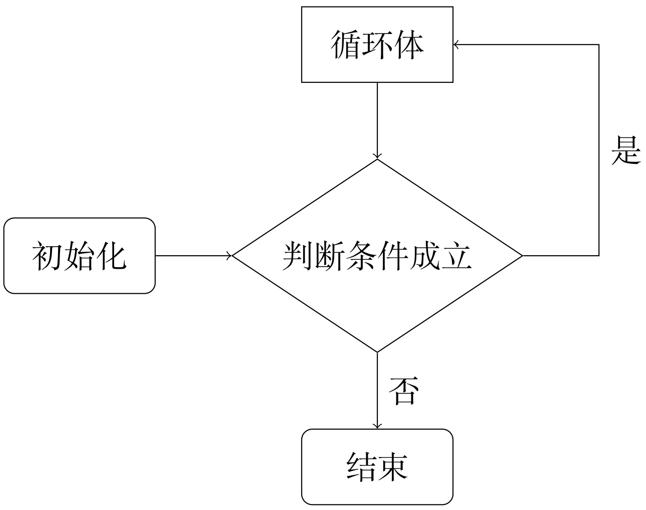


图 4.2

e.g. 验证 3x+1 猜想：

```
while (x > 1) {
  if (x % 2 == 1) {
    x = 3 * x + 1;
  } else {
    x = x / 2;
```

```
  }
}
```

do...while 语句

以下是 do...while 语句的结构：

```
do {
  循环体;
} while (判断条件);
```

执行顺序：

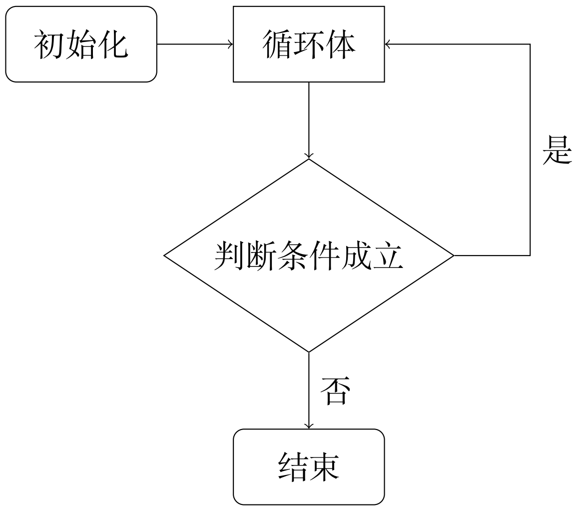


图 4.3

与 while 语句的区别在于，do...while 语句是先执行循环体再进行判断的。

e.g. 枚举排列：

```
do {
  // do someting...
} while (next_permutation(a + 1, a + n + 1));
```

三种语句的联系

```
// for 语句

for (statement1; statement2; statement3) {
  statement4;
}

// while 语句

statement1;
while (statement2) {
  statement4;
  statement3;
}
```

在 statement4 中没有 `continue` 语句（见下文）的时候是等价的，但是下面一种方法很少用到。

```cpp
// while 语句

statement1;
while (statement2) {
  statement1;
}

// do...while 语句

do {
  statement1;
} while (statement2);
```

在 statement1 中没有 `continue` 语句的时候这两种方式也也是等价的。

```cpp
while (1) {
  // do something...
}

for (;;) {
  // do something...
}
```

这两种方式都是永远循环下去。（可以使用 `break`（见下文）退出。）

可以看出，三种语句可以彼此代替，但一般来说，语句的选用遵守以下原则：

1. 循环过程中有个固定的增加步骤（最常见的是枚举）时，使用 for 语句；
2. 只确定循环的终止条件时，使用 while 语句；
3. 使用 while 语句时，若要先执行循环体再进行判断，使用 do...while 语句。一般很少用到，常用场景是用户输入。

break 与 continue 语句

break 语句的作用是退出循环。

continue 语句的作用是跳过循环体的余下部分。下面以 continue 语句在 do...while 语句中的使用为例：

```cpp
do {
  // do something...
  continue;  // 等价于 goto END;
// do something...
END:;
} while (statement);
```

break 与 continue 语句均可在三种循环语句的循环体中使用。

一般来说，break 与 continue 语句用于让代码的逻辑更加清晰，例如：

```cpp
// 逻辑较为不清晰，大括号层次复杂

for (int i = 1; i <= n; ++i) {
  if (i != x) {
    for (int j = 1; j <= n; ++j) {
      if (j != x) {
        // do something...
      }
    }
```

```
  }
}

// 逻辑更加清晰，大括号层次简单明了

for (int i = 1; i <= n; ++i) {
  if (i == x) continue;
  for (int j = 1; j <= n; ++j) {
    if (j == x) continue;
    // do something...
  }
}
```

```
// for 语句判断条件复杂，没有体现「枚举」的本质

for (int i = l; i <= r && i % 10 != 0; ++i) {
  // do something...
}

// for 语句用于枚举，break 用于「到何时为止」

for (int i = l; i <= r; ++i) {
  if (i % 10 == 0) break;
  // do something...
}
```

```
// 语句重复，顺序不自然

statement1;
while (statement3) {
  statement2;
  statement1;
}

// 没有重复语句，顺序自然

while (1) {
  statement1;
  if (!statement3) break;
  statement2;
}
```

4.2.6 高级数据类型

数组

数组是存放相同类型对象的容器，数组中存放的对象没有名字，而是要通过其所在的位置访问。数组的大小是固定的，不能随意改变数组的长度。

定义数组

数组的声明形如 `a[d]`，其中，`a` 是数组的名字，`d` 是数组中元素的个数。在编译时，`d` 应该是已知的，也就是说，`d` 应该是一个整型的常量表达式。

```cpp
unsigned int d1 = 42;
const int d2 = 42;
int arr1[d1];  // 错误：d1 不是常量表达式
int arr2[d2];  // 正确：arr2 是一个长度为 42 的数组
```

不能将一个数组直接赋值给另一个数组：

```cpp
int arr1[3];
int arr2 = arr1;  // 错误
arr2 = arr1;      // 错误
```

应该尽量将较大的数组定义为全局变量。因为局部变量会被创建在栈区中，过大（大于栈的大小）的数组会爆栈，进而导致 RE。如果将数组声明在全局作用域中，就会在静态区中创建数组。

访问数组元素

可以通过下标运算符 `[]` 来访问数组内元素，数组的索引（即方括号中的值）从 0 开始。以一个包含 10 个元素的数组为例，它的索引为 0 到 9，而非 1 到 10。但在 OI 中，为了使用方便，我们通常会将数组开大一点，不使用数组的第一个元素，从下标 1 开始访问数组元素。

例 1：从标准输入中读取一个整数 n，再读取 n 个数，存入数组中。其中，$n \leq 1000$。

```cpp
#include <iostream>
using namespace std;

int arr[1001];  // 数组 arr 的下标范围是 [0, 1001)

int main() {
  int n;
  cin >> n;
  for (int i = 1; i <= n; ++i) {
    cin >> arr[i];
  }
}
```

例 2：（接例 1）求和数组 `arr` 中的元素，并输出和。满足数组中所有元素的和小于等于 $2^{31}-1$

```cpp
#include <iostream>
using namespace std;

int arr[1001];

int main() {
  int n;
  cin >> n;
  for (int i = 1; i <= n; ++i) {
    cin >> arr[i];
  }

  int sum = 0;
  for (int i = 1; i <= n; ++i) {
    sum += arr[i];
  }

  printf("%d\n", sum);
```

```
  return 0;
}
```

越界访问下标

数组的下标 idx 应当满足 $0 \le idx < size$，如果下标越界，则会产生不可预料的后果，如段错误（Segmentation Fault），或者修改预期以外的变量。

多维数组

多维数组的实质是「数组的数组」，即外层数组的元素是数组。一个二维数组需要两个维度来定义：数组的长度和数组内元素的长度。访问二维数组时需要写出两个索引：

```
int arr[3][4];  // 一个长度为 3 的数组，它的元素是「元素为 int 的长度为的 4
                // 的数组」
arr[2][1] = 1;  // 访问二维数组
```

我们经常使用嵌套的 for 循环来处理二维数组。

例：从标准输入中读取两个数 n 和 m，分别表示黑白图片的高与宽，满足 $n, m \le 1000$。对于接下来的 n 行数据，每行有用空格分隔开的 m 个数，代表这一位置的亮度值。现在我们读取这张图片，并将其存入二维数组中。

```
const int maxn = 1001;
int pic[maxn][maxn];
int n, m;

cin >> n >> m;
for (int i = 1; i <= n; ++i)
  for (int j = 1; j <= m; ++j) cin >> pic[i][j];
```

同样地，你可以定义三维、四维，以及更高维的数组。

结构体

> **Authors:** Ir1d, cjsoft, Lans1ot

结构体（struct），可以看做是一系列称为成员元素的组合体。

可以看做是自定义的数据类型。

> note
>
> 本页描述的 `struct` 不同于 C 中 `struct`，在 C++ 中 `struct` 被扩展为类似 `class` 的类说明符。

定义结构体

```
struct Object {
  int weight;
  int value;
} e[array_length];

const Object a;
Object b, B[array_length], tmp;
Object *c;
```

上例中定义了一个名为 `Object` 的结构体，两个成员元素 `value`，`weight`，类型都为 `int`。

在 `}` 后，定义了数据类型为 `Object` 的常量 `a`，变量 `b`，变量 `tmp`，数组 `B`，指针 `c`。对于某种已经存在的类型，都可以使用这里的方法进行定义常量、变量、指针、数组等。

关于指针：不必强求掌握。

定义指针

如果是定义内置类型的指针，则与平常定义指针一样。

如果是定义结构体指针，在定义中使用 `StructName*` 进行定义。

```
struct Edge {
  /*
  ...
  */
  Edge* nxt;
};
```

上例仅作举例，不必纠结实际意义。

访问/修改成员元素

可以使用变量名 . 成员元素名进行访问。例如可以使用 `cout << var.v` 来输出 `var` 的 `v` 成员。

也可以使用指针名 -> 成员元素名或者使用 (* 指针名). 成员元素名进行访问。例如使用 `(*ptr).v = tmp` 或者 `ptr->v = tmp` 可以将结构体指针 `ptr` 指向的结构体的成员元素 `v` 赋值为 `tmp`:。

为什么需要结构体？

首先，条条大路通罗马，可以不使用结构体达到相同的效果。但是结构体能够显式地将成员元素（在算法竞赛中通常是变量）捆绑在一起，如本例中的 `Object` 结构体，便将 `value,weight` 放在了一起（定义这个结构体的实际意义是表示一件物品的重量与价值）。这样的好处边是限制了成员元素的使用。

想象一下，如果不使用结构体而且有两个数组 `value[]`, `Value[]`，很容易写混淆。但如果使用结构体，能够减轻出现使用变量错误的几率。

并且不同的结构体（结构体类型，如 `Object` 这个结构体）或者不同的结构体变量（结构体的实例，如上方的 `e` 数组）可以拥有相同名字的成员元素（如 `tmp.value,b.value`），同名的成员元素相互独立（拥有独自的内存，比如说修改 `tmp.value` 不会影响 `b.value` 的值）。

这样的好处是可以使用尽可能相同或者相近的变量去描述一个物品。比如说 `Object` 里有 `value` 这个成员变量；我们还可以定义一个 `Car` 结构体，同时也拥有 `value` 这个成员；如果不使用结构体，或许我们就需要定义 `valueOfObject[],valueOfCar[]` 等不同名称的数组来区分。

如果想要更详细的描述一种事物，还可以定义成员函数。请参考 类 获取详细内容。

更多的操作？

详见 类。

注意事项

为了访问内存的效率更高，编译器在处理结构中成员的实际存储情况时，可能会将成员对齐在一定的字节位置，也就意味着结构中有空余的地方。因此，该结构所占用的空间可能大于其中所有成员所占空间的总和。

参考资料

1. Class - zh.cppreference.com[1]
2. Data structures - cplusplus.com[2]
3. 对齐方式 - Microsoft Docs[3]

参考资料与注释

[1] Class - zh.cppreference.com

[2] Data structures - cplusplus.com

[3] 对齐方式 - Microsoft Docs

联合体

联合体（union）是特殊的类类型，它在一个时刻只能保有其一个非静态数据成员。

联合体在 2023 年正式被加入 NOI 大纲入门级中。

定义联合体

联合体声明的类说明符与类或 结构体 的声明相似：

```
union MyUnion {
  int x;
  long long y;
} x;
```

联合体的定义与结构体类似。按照上述定义，`MyUnion` 同样可以当作一种自定义类型使用。名称 `MyUnion` 可以省略。

访问/修改成员元素

与结构体类似，同样可以使用变量名 . 成员名进行访问。

联合体所占用的内存空间大小**不小于**其最大的成员的大小，所有成员**共用内存空间与地址**。当一个成员被赋值，由于内存共享，该联合体中的其他成员都会被覆盖。即同一时刻联合体中只能保存一个成员的值。

联合体的更多用法可以参见 cppreference：联合体声明[1]。

参考资料与注释

[1] cppreference：联合体声明

指针

Authors: tsagaanbar, Enter-tainer, Xeonacid

变量的地址、指针

在程序中，我们的数据都有其存储的地址。在程序每次的实际运行过程中，变量在物理内存中的存储位置不尽相同。不过，我们仍能够在编程时，通过一定的语句，来取得数据在内存中的地址。

地址也是数据。存放地址所用的变量类型有一个特殊的名字，叫做「指针变量」，有时也简称做「指针」。

" 指针变量的大小"

指针变量的大小在不同环境下有差异。在 32 位机上，地址用 32 位二进制整数表示，因此一个指针的大小为 4 字节。而 64 位机上，地址用 64 位二进制整数表示，因此一个指针的大小就变成了 8 字节。

地址只是一个刻度一般的数据，为了针对不同类型的数据，「指针变量」也有不同的类型，比如，可以有 `int` 类型的指针变量，其中存储的地址（即指针变量存储的数值）对应一块大小为 32 位的空间的起始地址；有 `char` 类型的指针变量，其中存储的地址对应一块 8 位的空间的起始地址。

事实上，用户也可以声明指向指针变量的指针变量。

假如用户自定义了一个结构体：

```
struct ThreeInt {
  int a;
  int b;
  int c;
};
```

则 `ThreeInt` 类型的指针变量，对应着一块 $3 \times 32 = 96$ bit 的空间。

指针的声明与使用

C/C++ 中，指针变量的类型为类型名后加上一个星号 `*`。比如，`int` 类型的指针变量的类型名即为 `int*`。

我们可以使用 `&` 符号取得一个变量的地址。

要想访问指针变量地址所对应的空间（又称指针所**指向**的空间），需要对指针变量进行**解引用**（dereference），使用 `*` 符号。

```
int main() {
  int a = 123;  // a: 123
  int* pa = &a;
  *pa = 321;  // a: 321
}
```

对结构体变量也是类似。如果要访问指针指向的结构中的成员，需要先对指针进行解引用，再使用 `.` 成员关系运算符。不过，更推荐使用「箭头」运算符 `->` 这一更简便的写法。

```
struct ThreeInt {
  int a;
  int b;
  int c;
};

int main() {
  ThreeInt x{1, 2, 3}, y{6, 7, 8};
  ThreeInt* px = &x;
  (*px) = y;    // x: {6,7,8}
  (*px).a = 4;  // x: {4,7,8}
  px->b = 5;    // x: {4,5,8}
}
```

指针的偏移

指针变量也可以**和整数**进行加减操作。对于 `int` 型指针，每加 1（递增 1），其指向的地址偏移 32 位（即 4 个字节）；若加 2，则指向的地址偏移 $2 \times 32 = 64$ 位。同理，对于 `char` 型指针，每次递增，其指向的地址偏移 8 位（即 1 个字节）。

使用指针偏移访问数组

我们前面说过，数组是一块连续的存储空间。而在 C/C++ 中，直接使用数组名，得到的是数组的起始地址。

```
int main() {
  int a[3] = {1, 2, 3};
  int* p = a;  // p 指向 a[0]
  *p = 4;      // a: [4, 2, 3]
  p = p + 1;   // p 指向 a[1]
  *p = 5;      // a: [4, 5, 3]
  p++;         // p 指向 a[2]
  *p = 6;      // a: [4, 5, 6]
}
```

当通过指针访问数组中的元素时，往往需要用到「指针的偏移」，换句话说，即通过一个基地址（数组起始的地址）加上偏移量来访问。

我们常用 `[]` 运算符来访问数组中某一指定偏移量处的元素。比如 `a[3]` 或者 `p[4]`。这种写法和对指针进行运算后再引用是等价的，即 `p[4]` 和 `*(p + 4)` 是等价的两种写法。

空指针

在 C++11 之前，C++ 和 C 一样使用 `NULL` 宏表示空指针常量，C++ 中 `NULL` 的实现一般如下：

```
// C++11 前
#define NULL 0
```

"C 语言对 NULL 的定义"

C 语言在 C23 前有两个 `NULL` 的定义，只有类型不同：一个是整型常量表达式，一个是转换为 `void *` 类型的常量表达式，但其值都为 0，编译器可任选一个实现。

空指针和整数 `0` 的混用在 C++ 中会导致许多问题，比如：

```
int f(int x);
int f(int* p);
```

在调用 `f(NULL)` 时，实际调用的函数的类型是 `int(int)` 而不是 `int(int *)`.

"NULL 在 C 语言中造成的问题"

比起在 C++ 中，因为有两个定义，在 C 语言中 `NULL` 造成的问题更为严重：如果在一个传递可变参数的函数中，函数编写者想要接受一个指针，但是函数调用者传递了一个定义为整型的 `NULL`，则会造成未定义行为，因在函数内使用传入的可变参数时，要进行类型转换，而从整型到指针类型的转换是未定义行为。[1-1]

为了解决这些问题，C++11 引入了 `nullptr` 关键字作为空指针常量。

C++ 规定 `nullptr` 可以隐式转换为任何指针类型，这种转换结果是该类型的空指针值。

`nullptr` 的类型为 `std::nullptr_t`, 称作空指针类型，可能的实现如下：

```
namespace std {
typedef decltype(nullptr) nullptr_t;
}
```

另外，C++11 起 `NULL` 宏的实现也被修改为了：

```
// C++11 起
#define NULL nullptr
```

"C 语言对空指针常量的改进"

基于类似的原因，C23 也引入了 `nullptr` 作为空指针常量，同时引入了 `nullptr_t` 作为其类型[1-2]。

指针的进阶使用

使用指针，使得程序编写者可以操作程序运行时中各处的数据，而不必局限于作用域。

指针类型参数的使用

在 C/C++ 中，调用函数（过程）时使用的参数，均以拷贝的形式传入子过程中（引用除外，会在后续介绍）。默认情况下，函数仅能通过返回值，将结果返回到调用处。但是，如果某个函数希望修改其外部的数据，或者某个结构体/类的数据量较为庞大、不宜进行拷贝，这时，则可以通过向其传入外部数据的地址，便得以在其中访问甚至修改外部数据。

下面的 `my_swap` 方法，通过接收两个 `int` 型的指针，在函数中使用中间变量，完成对两个 `int` 型变量值的交换。

```
void my_swap(int *a, int *b) {
  int t;
  t = *a;
  *a = *b;
  *b = t;
}

int main() {
  int a = 6, b = 10;
  my_swap(&a, &b);
  // 调用后，main 函数中 a 变量的值变为 10，b 变量的值变为 6
}
```

C++ 中引入了引用的概念，相对于指针来说，更易用，也更安全。详情可以参见 C++：引用 以及 C 与 C++ 的区别： 指针与引用。

动态实例化

除此之外，程序编写时往往会涉及到动态内存分配，即，程序会在运行时，向操作系统动态地申请或归还存放数据所需的内存。当程序通过调用操作系统接口申请内存时，操作系统将返回程序所申请空间的地址。要使用这块空间，我们需要将这块空间的地址存储在指针变量中。

在 C++ 中，我们使用 `new` 运算符来获取一块内存，使用 `delete` 运算符释放某指针所指向的空间。

```
int* p = new int(1234);
/* ... */
delete p;
```

上面的语句使用 `new` 运算符向操作系统申请了一块 `int` 大小的空间，将其中的值初始化为 1234，并声明了一个 `int` 型的指针 `p` 指向这块空间。

同理，也可以使用 `new` 开辟新的对象：

```
class A {
  int a;

 public:
  A(int a_) : a(a_) {}
};

int main() {
  A* p = new A(1234);
  /* ... */
  delete p;
}
```

如上，「new 表达式」将尝试开辟一块对应大小的空间，并尝试在这块空间上构造这一对象，并返回这一空间的地址。

```
struct ThreeInt {
  int a;
  int b;
  int c;
};

int main() {
  ThreeInt* p = new ThreeInt{1, 2, 3};
  /* ... */
  delete p;
}
```

" 列表初始化"

{} 运算符可以用来初始化没有构造函数的结构。除此之外，使用 {} 运算符可以使得变量的初始化形式变得统一。详见「list initialization (since C++11)[2]」。

需要注意，当使用 `new` 申请的内存不再使用时，需要使用 `delete` 释放这块空间。不能对一块内存释放两次或以上。而对空指针 `nullptr` 使用 `delete` 操作是合法的。

动态创建数组

也可以使用 `new[]` 运算符创建数组，这时 `new[]` 运算符会返回数组的首地址，也就是数组第一个元素的地址，我们可以用对应类型的指针存储这个地址。释放时，则需要使用 `delete[]` 运算符。

```
size_t element_cnt = 5;
int *p = new int[element_cnt];
delete[] p;
```

数组中元素的存储是连续的，即 `p + 1` 指向的是 `p` 的后继元素。

二维数组

在存放矩阵形式的数据时，可能会用到「二维数组」这样的数据类型。从语义上来讲，二维数组是一个数组的数组。而计算机内存可以视作一个很长的一维数组。要在计算机内存中存放一个二维数组，便有「连续」与否的说法。

所谓「连续」，即二维数组的任意一行（row）的末尾与下一行的起始，在物理地址上是毗邻的，换言之，整个二维数组可以视作一个一维数组；反之，则二者在物理上不一定相邻。

对于「连续」的二维数组，可以仅使用一个循环，借由一个不断递增的指针即可遍历数组中的所有数据。而对于非连续的二维数组，由于每一行不连续，则需要先取得某一行首的地址，再访问这一行中的元素。

" 二维数组的存储方式"

这种按照「行（row）」存储数据的方式，称为行优先存储；相对的，也可以按照列（column）存储数据。由于计算机内存访问的特性，一般来说，访问连续的数据会得到更高的效率。因此，需要按照数据可能的使用方式，选择「行优先」或「列优先」的存储方式。

动态创建二维数组

在 C/C++ 中，我们可以使用类似下面这样的语句声明一个 N 行（row）M 列（column）的二维数组，其空间在物理上是连续的。

" 描述数组的维度"

更通用的方式是使用第 n 维（dimension）的说法。对于「行优先」的存储形式，数组的第一维长度为 N，第二维长度为 M。

```
int a[N][M];
```

这种声明方式要求 N 和 M 为在编译期即可确定的常量表达式。

在 C/C++ 中，数组的第一个元素下标为 0，因此 `a[r][c]` 这样的式子代表二维数组 a 中第 $r+1$ 行的第 $c+1$ 个元素，我们也称这个元素的下标为 `(r,c)`。

不过，实际使用中，（二维）数组的大小可能不是固定的，需要动态内存分配。

常见的方式是声明一个长度为 $N \times M$ 的**一维数组**，并通过下标 `r * M + c` 访问二维数组中下标为 `(r, c)` 的元素。

```
int* a = new int[N * M];
```

这种方法可以保证二维数组是**连续的**。

" 数组在物理层面上的线性存储"

实际上，数据在内存中都可以视作线性存放的，因此在一定的规则下，通过动态开辟一维数组的空间，即可在其上存储 n 维的数组。

此外，亦可以根据「数组的数组」这一概念来进行内存的获取与使用。对于一个存放的若干数组的数组，实际上为一个存放的若干数组的首地址的数组，也就是一个存放若干指针变量的数组。

我们需要一个变量来存放这个「数组的数组」的首地址——也就是一个指针的地址。这个变量便是一个「指向指针的指针」，有时也称作「二重指针」，如：

```
int** a = new int*[5];
```

接着，我们需要为每一个数组申请空间：

```
for (int i = 0; i < 5; i++) {
  a[i] = new int[5];
}
```

至此，我们便完成了内存的获取。而对于这样获得的内存的释放，则需要进行一个逆向的操作：即先释放每一个数组，再释放存储这些数组首地址的数组，如：

```
for (int i = 0; i < 5; i++) {
  delete[] a[i];
}
delete[] a;
```

需要注意，这样获得的二维数组，不能保证其空间是连续的。

还有一种方式，需要使用到「指向数组的指针」。

" 数组名和数组首元素地址的区别"

我们之前说到，在 C/C++ 中，直接使用数组名，值等于数组首元素的地址。但是数组名表示的这一变量的类型实际上是整个数组，而非单个元素。

```
int main() { int a[5] = {1, 2, 3, 4, 5}; }
```

从概念上说，代码中标识符 `a` 的类型是 `int[5]`；从实际上来说，`a + 1` 所指向的地址相较于 `a` 指向的地址的偏移量为 5 个 `int` 型变量的长度。

```
int main() {
  int(*a)[5] = new int[5][5];
  int* p = a[2];
  a[2][1] = 1;
  delete[] a;
}
```

这种方式获得到的也是连续的内存，但是可以直接使用 `a[n]` 的形式获得到数组的第 $n+1$ 行（row）的首地址，因此，使用 `a[r][c]` 的形式即可访问到下标为 `(r, c)` 的元素。

由于指向数组的指针也是一种确定的数据类型，因此除数组的第一维外，其他维度的长度均须为一个能在编译器确定的常量。不然，编译器将无法翻译如 `a[n]` 这样的表达式（`a` 为指向数组的指针）。

指向函数的指针

关于函数的介绍请参见 C++ 函数 章节。

简单地说，要调用一个函数，需要知晓该函数的参数类型、个数以及返回值类型，这些也统一称作接口类型。

可以通过函数指针调用函数。有时候，若干个函数的接口类型是相同的，使用函数指针可以根据程序的运行**动态地**选择需要调用的函数。换句话说，可以在不修改一个函数的情况下，仅通过修改向其传入的参数（函数指针），使得该函数的行为发生变化。

假设我们有若干针对 `int` 类型的二元运算函数，则函数的参数为 2 个 `int`，返回值亦为 `int`。下边是一个使用了函数指针的例子：

```
#include <iostream>

int (*binary_int_op)(int, int);

int foo1(int a, int b) { return a * b + b; }

int foo2(int a, int b) { return (a + b) * b; }

int main() {
  int choice;
  std::cin >> choice;
  if (choice == 1) {
    binary_int_op = foo1;
  } else {
    binary_int_op = foo2;
  }

  int m, n;
  std::cin >> m >> n;
  std::cout << binary_int_op(m, n);
}
```

"&、* 和函数指针"

在 C 语言中，诸如 `void (*p)() = foo;`、`void (*p)() = &foo;`、`void (*p)() = *foo;`、`void (*p)() = ***foo` 等写法的结果是一样的。

因为函数（如 `foo`）是能够被隐式转换为指向函数的指针的，因此 `void (*p)() = foo;` 的写法能够成立。

使用 `&` 运算符可以取得到对象的地址，这对函数也是成立的，因此 `void (*p)() = &foo;` 的写法仍然成立。

对函数指针使用 `*` 运算符可以取得指针指向的函数，而对于 `**foo` 这样的写法来说，`*foo` 得到的是 `foo` 这个函数，紧接着又被隐式转换为指向 `foo` 的指针。如此类推，`**foo` 得到的最终还是指向 `foo` 的函数指针；

用户尽可以使用任意多的 `*`，结果也是一样的。

同理，在调用时使用类似 `(*p)()` 和 `p()` 的语句是一样的，可以省去 `*` 运算符。

参考资料：Why do function pointer definitions work with any number of ampersands '&' or asterisks '*'? - stackoverflow.com[3]

可以使用 `typedef` 关键字声明函数指针的类型。

```
typedef int (*p_bi_int_op)(int, int);
```

这样我们就可以在之后使用 `p_bi_int_op` 这种类型，即指向「参数为 2 个 `int`，返回值亦为 `int`」的函数的指针。

可以通过使用 `std::function` 来更方便的引用函数。(未完待续)

使用函数指针，可以实现「回调函数」。(未完待续)

参考资料与注释

[1] 参见 Introduce the nullptr constant [1-1] [1-2]

[2] list initialization (since C++11)

[3] Why do function pointer definitions work with any number of ampersands '&' or asterisks '*'? - stackoverflow.com

4.2.7 函数

Authors: Ir1d, tsagaanbar, yang-lile

函数的声明

编程中的函数（function）一般是若干语句的集合。我们也可以将其称作「**子过程**（subroutine）」。在编程中，如果有一些重复的过程，我们可以将其提取出来，形成一个函数。函数可以接收若干值，这叫做函数的参数。函数也可以返回某个值，这叫做函数的返回值。

声明一个函数，我们需要返回值类型、函数的名称，以及参数列表。

```
// 返回值类型 int
// 函数的名称 some_function
// 参数列表 int, int
int some_function(int, int);
```

如上图，我们声明了一个名为 `some_function` 的函数，它需要接收两个 `int` 类型的参数，返回值类型也为 `int`。可以认为，这个函数将会对传入的两个整数进行一些操作，并且返回一个同样类型的结果。

实现函数：编写函数的定义

只有函数的声明（declaration）还不够，他只能让我们在调用时能够得知函数的**接口**类型（即接收什么数据、返回什么数据），但其缺乏具体的内部实现，也就是函数的**定义**（definition）。我们可以在**声明之后的其他地方**编写代码**实现**（implement）这个函数（也可以在另外的文件中实现，但是需要将分别编译后的文件在链接时一并给出）。

如果函数有返回值，则需要通过 `return` 语句，将值返回给调用方。函数一旦执行到 `return` 语句，则直接结束当前函数，不再执行后续的语句。

```cpp
int some_function(int, int);  // 声明

/* some other code here... */

int some_function(int x, int y) {  // 定义
  int result = 2 * x + y;
  return result;
  result = 3;  // 这条语句不会被执行
}
```

在定义时，我们给函数的参数列表的变量起了名字。这样，我们便可以在函数定义中使用这些变量了。

如果是同一个文件中，我们也可以直接将**声明和定义合并在一起**，换句话说，也就是在声明时就完成定义。

```cpp
int some_function(int x, int y) { return 2 * x + y; }
```

如果函数不需要有返回值，则将函数的返回值类型标为 `void`；如果函数不需要参数，则可以将参数列表置空。同样，无返回值的函数执行到 `return;` 语句也会结束执行。

```cpp
void say_hello() {
  cout << "hello!\n";
  cout << "hello!\n";
  cout << "hello!\n";
  return;
  cout << "hello!\n";  // 这条语句不会被执行
}
```

函数的调用

和变量一样，函数需要先被声明，才能使用。使用函数的行为，叫做「调用（call）」。我们可以在任何函数内部调用其他函数，包括这个函数自身。函数调用自身的行为，称为**递归**（recursion）。

在大多数语言中，调用函数的写法，是**函数名称加上一对括号** `()`，如 `foo()`。如果函数需要参数，则我们将其需要的参数按顺序填写在括号中，以逗号间隔，如 `foo(1, 2)`。函数的调用也是一个表达式，**函数的返回值**就是**表达式的值**。

函数声明时候写出的参数，可以理解为在函数**当前次调用的内部**可以使用的变量，这些变量的值由调用处传入的值初始化。看下面这个例子：

```cpp
void foo(int, int);

/* ... */

void foo(int x, int y) {
  x = x * 2;
  y = y + 3;
}

/* ... */

a = 1;
b = 1;
// 调用前：a = 1, b = 1
foo(a, b);  // 调用 foo
            // 调用后：a = 1, b = 1
```

在上面的例子中，foo(a, b) 是一次对 foo 的调用。调用时，foo 中的 x 和 y 变量，分别由调用处 a 和 b 的值初始化。因此，在 foo 中对变量 x 和 y 的修改，**并不会影响到调用处的变量的值**。

如果我们需要在函数（子过程）中修改变量的值，则需要采用「传引用」的方式。

```
void foo(int& x, int& y) {
  x = x * 2;
  y = y + 3;
}

/* ... */

a = 1;
b = 1;
// 调用前：a = 1, b = 1
foo(a, b);  // 调用 foo
            // 调用后：a = 2, b = 4
```

上述代码中，我们看到函数参数列表中的「int」后面添加了一个「&（and 符号）」，这表示对于 int 类型的**引用**（reference）。在调用 foo 时，调用处 a 和 b 变量分别初始化了 foo 中两个对 int 类型的引用 x 和 y。在 foo 中的 x 和 y，可以理解为调用处 a 和 b 变量的「别名」，即 foo 中对 x 和 y 的操作，就是对调用处 a 和 b 的操作。

main 函数

特别的，每个 C/C++ 程序都需要有一个名为 main 的函数。任何程序都将从 main 函数开始运行。

> main 函数也可以有参数，通过 main 函数的参数，我们可以获得外界传给这个程序的指令（也就是「命令行参数」），以便做出不同的反应。

下面是一段调用了函数（子过程）的代码：

```
// hello_subroutine.cpp

#include <iostream>

void say_hello() {
  std::cout << "hello!\n";
  std::cout << "hello!\n";
  std::cout << "hello!\n";
}

int main() {
  say_hello();
  say_hello();
}
```

4.2.8 文件操作

Authors: Ir1d, cqnuljs, akakw1, MingqiHuang, Chrogeek, henrytbtrue, Planet6174, StudyingFather

文件的概念

文件是根据特定的目的而收集在一起的有关数据的集合。C/C++ 把每一个文件都看成是一个有序的字节流，每个文件都是以**文件结束标志**（EOF）结束，如果要操作某个文件，程序应该首先打开该文件，每当一个文件被打开后

（请记得关闭打开的文件），该文件就和一个流关联起来，这里的流实际上是一个字节序列。

C/C++ 将文件分为文本文件和二进制文件。文本文件就是简单的文本文件（重点），另外二进制文件就是特殊格式的文件或者可执行代码文件等。

文件的操作步骤

1、打开文件，将文件指针指向文件，决定打开文件类型；

2、对文件进行读、写操作（比赛中主要用到的操作，其他一些操作暂时不写）；

3、在使用完文件后，关闭文件。

freopen 函数

函数简介

函数用于将指定输入输出流以指定方式重定向到文件，包含于头文件 stdio.h (cstdio) 中，该函数可以在不改变代码原貌的情况下改变输入输出环境，但使用时应当保证流是可靠的。

函数主要有三种方式：读、写和附加。

命令格式

```
FILE* freopen(const char* filename, const char* mode, FILE* stream);
```

参数说明

- filename: 要打开的文件名
- mode: 文件打开的模式，表示文件访问的权限
- stream: 文件指针，通常使用标准文件流（stdin/stdout）或标准错误输出流（stderr）
- 返回值：文件指针，指向被打开文件

文件打开格式（选读）

- r：以只读方式打开文件，文件必须存在，只允许读入数据**（常用）**
- r+：以读/写方式打开文件，文件必须存在，允许读/写数据
- rb：以只读方式打开二进制文件，文件必须存在，只允许读入数据
- rb+：以读/写方式打开二进制文件，文件必须存在，允许读/写数据
- rt+：以读/写方式打开文本文件，允许读/写数据
- w：以只写方式打开文件，文件不存在会新建文件，否则清空内容，只允许写入数据**（常用）**
- w+：以读/写方式打开文件，文件不存在将新建文件，否则清空内容，允许读/写数据
- wb：以只写方式打开二进制文件，文件不存在将会新建文件，否则清空内容，只允许写入数据
- wb+：以读/写方式打开二进制文件，文件不存在将新建文件，否则清空内容，允许读/写数据
- a：以只写方式打开文件，文件不存在将新建文件，写入数据将被附加在文件末尾（保留 EOF 符）
- a+：以读/写方式打开文件，文件不存在将新建文件，写入数据将被附加在文件末尾（不保留 EOF 符）
- at+：以读/写方式打开文本文件，写入数据将被附加在文件末尾
- ab+：以读/写方式打开二进制文件，写入数据将被附加在文件末尾

使用方法

读入文件内容：

```
freopen("data.in", "r", stdin);
// data.in 就是读取的文件名，要和可执行文件放在同一目录下
```

输出到文件：

```
freopen("data.out", "w", stdout);
// data.out 就是输出文件的文件名，和可执行文件在同一目录下
```

关闭标准输入/输出流

```
fclose(stdin);
fclose(stdout);
```

注

printf/scanf/cin/cout 等函数默认使用 stdin/stdout，将 stdin/stdout 重定向后，这些函数将输入/输出到被定向的文件

模板

```
#include <cstdio>
#include <iostream>

int main(void) {
  freopen("data.in", "r", stdin);
  freopen("data.out", "w", stdout);
  /*
  中间的代码不需要改变，直接使用 cin 和 cout 即可
  */
  fclose(stdin);
  fclose(stdout);
  return 0;
}
```

fopen 函数（选读）

函数大致与 freopen 相同，函数将打开指定文件并返回打开文件的指针

函数原型

```
FILE* fopen(const char* path, const char* mode)
```

各项参数含义同 freopen

可用读写函数（基本）

- fread/fwrite
- fgetc/fputc
- fscanf/fprintf
- fgets/fputs

使用方式

```
FILE *in, *out;  // 定义文件指针
in = fopen("data.in", "r");
out = fopen("data.out", "w");
/*
do what you want to do
*/
fclose(in);
fclose(out);
```

C++ 的 ifstream/ofstream 文件输入输出流

使用方法

读入文件内容:

```
ifstream fin("data.in");
// data.in 就是读取文件的相对位置或绝对位置
```

输出到文件:

```
ofstream fout("data.out");
// data.out 就是输出文件的相对位置或绝对位置
```

关闭标准输入/输出流

```
fin.close();
fout.close();
```

模板

```
#include <fstream>
using namespace std;  // 两个类型都在 std 命名空间里

ifstream fin("data.in");
ofstream fout("data.out");

int main(void) {
  /*
  中间的代码改变 cin 为 fin ，cout 为 fout 即可
  */
  fin.close();
  fout.close();
  return 0;
}
```

参考资料

1. 信息学奥赛一本通

4.3 C++ 标准库

4.3.1 C++ 标准库简介

C++ 标准

首先需要介绍的是 C++ 本身的版本。由于 C++ 本身只是一门语言，而不同的编译器对 C++ 的实现方法各不一致，因此需要标准化来约束编译器的实现，使得 C++ 代码在不同的编译器下表现一致。C++ 自 1985 年诞生以来，一共由国际标准化组织（ISO）发布了 5 个正式的 C++ 标准，依次为 C++98、C++03、C++11（亦称 C++0x）、C++14（亦称 C++1y）、C++17（亦称 C++1z）、C++20（亦称 C++2a）。C++ 标准草案在 open-std[1] 网站上，最新的标准 C++23（亦称 C++2b）仍在制定中。此外还有一些补充标准，例如 C++ TR1。

每一个版本的 C++ 标准不仅规定了 C++ 的语法、语言特性，还规定了一套 C++ 内置库的实现规范，这个库便是 C++ 标准库。C++ 标准库中包含大量常用代码的实现，如输入输出、基本数据结构、内存管理、多线程支持等。掌握 C++ 标准库是编写更现代的 C++ 代码必要的一步。C++ 标准库的详细文档在 cppreference[2] 网站上，文档对标准库中的类型函数的用法、效率、注意事项等都有介绍，请善用。

需要指出的是，不同的 OJ 平台对 C++ 版本均不相同，例如 最新的 ICPC 比赛规则[3] 支持 C++20 标准。根据 NOI 科学委员会决议，自 2021 年 9 月 1 日起 NOI Linux 2.0[4] 作为 NOI 系列比赛和 CSP-J/S 等活动的标准环境使用。NOI Linux 2.0 中指定的 g++ 9.3.0 默认支持标准[5] 为 C++14，并支持 C++17 标准，可以满足绝大部分竞赛选手的需求。因此在学习 C++ 时要注意比赛支持的标准，避免在赛场上时编译报错。

标准模板库（STL）

STL 即标准模板库（Standard Template Library），是 C++ 标准库的一部分，里面包含了一些模板化的通用的数据结构和算法。由于其模板化的特点，它能够兼容自定义的数据类型，避免大量的造轮子工作。NOI 和 ICPC 赛事都支持 STL 库的使用，因此合理利用 STL 可以避免编写无用算法，并且充分利用编译器对模板库优化提高效率。STL 库的详细介绍请参见对应的页面：STL 容器 和 STL 算法。

" 什么是造轮子"

造轮子（Reinventing_the_wheel[6]）指的是重复发明已有的算法，或者重复编写现成优化过的代码。造轮子通常耗时耗力，同时效果还没有别人好。但若是为了学习或者练习，造轮子则是必要的。

Boost 库

Boost[7] 是除了标准库外，另一个久副盛名的开源 C++ 工具库，其代码具有可移植、高质量、高性能、高可靠性等特点。Boost 中的模块数量非常之大，功能全面，并且拥有完备的跨平台支持，因此被看作 C++ 的准标准库。C++ 标准中的不少特性也都来自于 Boost，如智能指针、元编程、日期和时间等。尽管在 OI 中无法使用 Boost，但是 Boost 中有不少轮子可以用来验证算法或者对拍，如 Boost.Geometry 有 R 树的实现，Boost.Graph 有图的相关算法，Boost.Intrusive 则提供了一套与 STL 容器用法相似的侵入式容器。有兴趣的读者可以自行在网络搜索教程。

参考资料

1. C++ reference[8]
2. C++ 参考手册[2]
3. 维基百科 - C++[9]
4. Boost 官方网站[7]
5. Boost 教程网站[10]

参考资料与注释

[1] open-std

[2] cppreference [2-1] [2-2]

[3] 最新的 ICPC 比赛规则

[4] NOI Linux 2.0

[5] 默认支持标准

[6] Reinventing_the_wheel

[7] Boost [7-1] [7-2]

[8] C++ reference

[9] 维基百科 - C++

[10] Boost 教程网站

4.3.2 STL 容器

STL 容器简介

分类

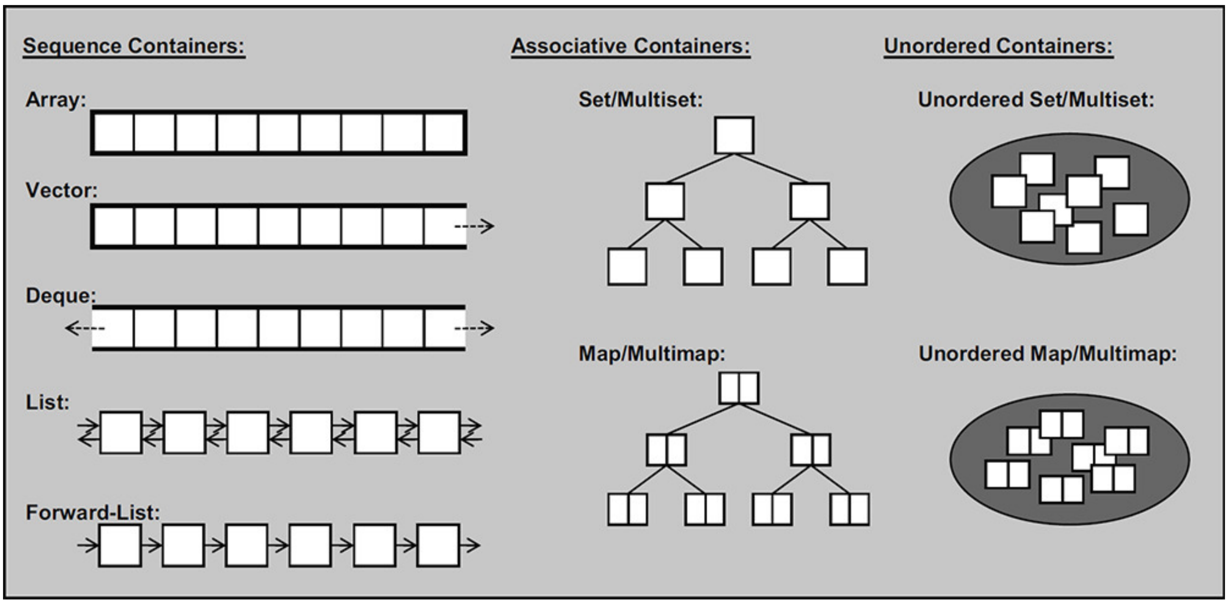


图 4.4

序列式容器

- **向量** (`vector`) 后端可高效增加元素的顺序表。
- **数组** (`array`)**C++11**，定长的顺序表，C 风格数组的简单包装。
- **双端队列** (`deque`) 双端都可高效增加元素的顺序表。
- **列表** (`list`) 可以沿双向遍历的链表。
- **单向列表** (`forward_list`) 只能沿一个方向遍历的链表。

关联式容器

- **集合** (`set`) 用以有序地存储**互异**元素的容器。其实现是由节点组成的红黑树，每个节点都包含着一个元素，节点之间以某种比较元素大小的谓词进行排列。
- **多重集合** (`multiset`) 用以有序地存储元素的容器。允许存在相等的元素。
- **映射** (`map`) 由 {键，值} 对组成的集合，以某种比较键大小关系的谓词进行排列。
- **多重映射** (`multimap`) 由 {键，值} 对组成的多重集合，亦即允许键有相等情况的映射。

> **" 什么是谓词 (Predicate[1]) ? "**
>
> 谓词就是返回值为真或者假的函数。STL 容器中经常会使用到谓词，用于模板参数。

无序（关联式）容器

- **无序（多重）集合** (`unordered_set`/`unordered_multiset`)**C++11**，与 `set`/`multiset` 的区别在于元素无序，只关心「元素是否存在」，使用哈希实现。
- **无序（多重）映射** (`unordered_map`/`unordered_multimap`)**C++11**，与 `map`/`multimap` 的区别在于键 (key) 无序，只关心" 键与值的对应关系"，使用哈希实现。

容器适配器

容器适配器其实并不是容器。它们不具有容器的某些特点（如：有迭代器、有 `clear()` 函数……）。

> 「适配器是使一种事物的行为类似于另外一种事物行为的一种机制」，适配器对容器进行包装，使其表现出另外一种行为。

- **栈** (`stack`) 后进先出 (LIFO) 的容器，默认是对双端队列（`deque`）的包装。
- **队列** (`queue`) 先进先出 (FIFO) 的容器，默认是对双端队列（`deque`）的包装。
- **优先队列** (`priority_queue`) 元素的次序是由作用于所存储的值对上的某种谓词决定的的一种队列，默认是对向量（`vector`）的包装。

共同点

容器声明

都是 `containerName<typeName,...> name` 的形式，但模板参数（`<>` 内的参数）的个数、形式会根据具体容器而变。

本质原因：STL 就是「标准模板库」，所以容器都是模板类。

迭代器

请参考 迭代器。

共有函数

`=`：有赋值运算符以及复制构造函数。

`begin()`：返回指向开头元素的迭代器。

`end()`：返回指向末尾的下一个元素的迭代器。`end()` 不指向某个元素，但它是末尾元素的后继。

`size()`：返回容器内的元素个数。

`max_size()`：返回容器**理论上**能存储的最大元素个数。依容器类型和所存储变量的类型而变。

`empty()`：返回容器是否为空。

`swap()`：交换两个容器。

`clear()`：清空容器。

`==/!=/</>/<=/>=`：按**字典序**比较两个容器的大小。（比较元素大小时 `map` 的每个元素相当于 `set<pair<key, value> >`，无序容器不支持 `</>/<=/>=`。）

参考资料与注释

[1] **Predicate**

迭代器

在 STL 中，迭代器（Iterator）用来访问和检查 STL 容器中元素的对象，它的行为模式和指针类似，但是它封装了一些有效性检查，并且提供了统一的访问格式。类似的概念在其他很多高级语言中都存在，如 Python 的 `__iter__` 函数，C# 的 `IEnumerator`。

基础使用

迭代器听起来比较晦涩，其实迭代器本身可以看作一个数据指针。迭代器主要支持两个运算符：自增 (`++`) 和解引用（单目 `*` 运算符），其中自增用来移动迭代器，解引用可以获取或修改它指向的元素。

指向某个 STL 容器 `container` 中元素的迭代器的类型一般为 `container::iterator`。

迭代器可以用来遍历容器，例如，下面两个 for 循环的效果是一样的：

```
vector<int> data(10);

for (int i = 0; i < data.size(); i++)
  cout << data[i] << endl;  // 使用下标访问元素

for (vector<int>::iterator iter = data.begin(); iter != data.end(); iter++)
  cout << *iter << endl;  // 使用迭代器访问元素
// 在 C++11 后可以使用 auto iter = data.begin() 来简化上述代码
```

"auto 在竞赛中的使用"

大部分选手都喜欢使用 `auto` 来代替繁琐的迭代器声明。根据 2021 年 9 月发布的 关于 NOI 系列活动中编程语言使用限制的补充说明[1]，NOI 系列比赛（包括 CSP J/S）在评测时将使用 **C++14**，这个版本已经支持了 `auto` 关键字。

分类

在 STL 的定义中，迭代器根据其支持的操作依次分为以下几类：

- InputIterator（输入迭代器）：只要求支持拷贝、自增和解引访问。
- OutputIterator（输出迭代器）：只要求支持拷贝、自增和解引赋值。
- ForwardIterator（向前迭代器）：同时满足 InputIterator 和 OutputIterator 的要求。
- BidirectionalIterator（双向迭代器）：在 ForwardIterator 的基础上支持自减（即反向访问）。
- RandomAccessIterator（随机访问迭代器）：在 BidirectionalIterator 的基础上支持加减运算和比较运算（即随机访问）。

- ContiguousIterator（连续迭代器）：在 RandomAccessIterator 的基础上要求对可解引用的迭代器 `a + n` 满足 `*(a + n)` 与 `*(std::address_of(*a) + n)` 等价（即连续存储，其中 `a` 为连续迭代器、`n` 为整型值）。

  ContiguousIterator 于 C++17 中正式引入。

> **" 为什么输入迭代器叫输入迭代器？"**
>
> 「输入」指的是「可以从迭代器中获取输入」，而「输出」指的是「可以输出到迭代器」。
>
> 「输入」和「输出」的施动者是程序的其它部分，而不是迭代器自身。

其实这个「分类」并不互斥——一个「类别」是可以包含另一个「类别」的。例如，在要求使用向前迭代器的地方，同样可以使用双向迭代器。

不同的 STL 容器 支持的迭代器类型不同，在使用时需要留意。

指针满足随机访问迭代器的所有要求，可以当作随机访问迭代器使用。

相关函数

很多 STL 函数 都使用迭代器作为参数。

可以使用 `std::advance(it, n)` 将迭代器 `it` 向后移动 `n` 步；若 `n` 为负数，则对应向前移动。迭代器必须满足双向迭代器，否则行为未定义。

在 C++11 以后可以使用 `std::next(it)` 获得向前迭代器 `it` 的后继（此时迭代器 `it` 不变），`std::next(it, n)` 获得向前迭代器 `it` 的第 `n` 个后继。

在 C++11 以后可以使用 `std::prev(it)` 获得双向迭代器 `it` 的前驱（此时迭代器 `it` 不变），`std::prev(it, n)` 获得双向迭代器 `it` 的第 `n` 个前驱。

STL 容器 一般支持从一端或两端开始的访问，以及对 const 修饰符 的支持。例如容器的 `begin()` 函数可以获得指向容器第一个元素的迭代器，`rbegin()` 函数可以获得指向容器最后一个元素的反向迭代器，`cbegin()` 函数可以获得指向容器第一个元素的 const 迭代器，`end()` 函数可以获得指向容器尾端（「尾端」并不是最后一个元素，可以看作是最后一个元素的后继；「尾端」的前驱是容器里的最后一个元素，其本身不指向任何一个元素）的迭代器。

可在 Iterator library - cppreference.com[2] 查看更多用法。

参考资料与注释

[1] 关于 NOI 系列活动中编程语言使用限制的补充说明

[2] Iterator library - cppreference.com

序列式容器

Authors: MingqiHuang, Xeonacid, greyqz, i-Yirannn, ChenZ01

vector

`std::vector` 是 STL 提供的**内存连续的**、**可变长度**的数组（亦称列表）数据结构。能够提供线性复杂度的插入和删除，以及常数复杂度的随机访问。

为什么要使用 `vector`

作为 OIer，对程序效率的追求远比对工程级别的稳定性要高得多，而 `vector` 由于其对内存的动态处理，时间效率在部分情况下低于静态数组，并且在 OJ 服务器不一定开全优化的情况下更加糟糕。所以在正常存储数据的时候，通常不选择 `vector`。下面给出几个 `vector` 优秀的特性，在需要用到这些特性的情况下，`vector` 能给我们带来很大的帮助。

vector 可以动态分配内存

很多时候我们不能提前开好那么大的空间（eg：预处理 1~n 中所有数的约数）。尽管我们能知道数据总量在空间允许的级别，但是单份数据还可能非常大，这种时候我们就需要 vector 来把内存占用量控制在合适的范围内。vector 还支持动态扩容，在内存非常紧张的时候这个特性就能派上用场了。

vector 重写了比较运算符及赋值运算符

vector 重载了六个比较运算符，以字典序实现，这使得我们可以方便的判断两个容器是否相等（复杂度与容器大小成线性关系）。例如可以利用 vector<char> 实现字符串比较（当然，还是用 std::string 会更快更方便）。另外 vector 也重载了赋值运算符，使得数组拷贝更加方便。

vector 便利的初始化

由于 vector 重载了 = 运算符，所以我们可以方便的初始化。此外从 C++11 起 vector 还支持 列表初始化[1]，例如 vector<int> data {1, 2, 3};。

vector 的使用方法

以下介绍常用用法，详细内容 请参见 C++ 文档[2]。

构造函数

用例参见如下代码（假设你已经 using 了 std 命名空间相关类型）：

```
// 1. 创建空 vector; 常数复杂度
vector<int> v0;
// 1+. 这句代码可以使得向 vector 中插入前 3 个元素时，保证常数时间复杂度
v0.reserve(3);
// 2. 创建一个初始空间为 3 的 vector，其元素的默认值是 0; 线性复杂度
vector<int> v1(3);
// 3. 创建一个初始空间为 3 的 vector，其元素的默认值是 2; 线性复杂度
vector<int> v2(3, 2);
// 4. 创建一个初始空间为 3 的 vector，其元素的默认值是 1，
// 并且使用 v2 的空间配置器; 线性复杂度
vector<int> v3(3, 1, v2.get_allocator());
// 5. 创建一个 v2 的拷贝 vector v4，其内容元素和 v2 一样; 线性复杂度
vector<int> v4(v2);
// 6. 创建一个 v4 的拷贝 vector v5，其内容是 {v4[1], v4[2]}; 线性复杂度
vector<int> v5(v4.begin() + 1, v4.begin() + 3);
// 7. 移动 v2 到新创建的 vector v6，不发生拷贝; 常数复杂度; 需要 C++11
vector<int> v6(std::move(v2));  // 或者 v6 = std::move(v2);
```

" 测试代码"

```
// 以下是测试代码，有兴趣的同学可以自己编译运行一下本代码。
cout << "v1 = ";
copy(v1.begin(), v1.end(), ostream_iterator<int>(cout, " "));
cout << endl;
cout << "v2 = ";
copy(v2.begin(), v2.end(), ostream_iterator<int>(cout, " "));
cout << endl;
cout << "v3 = ";
copy(v3.begin(), v3.end(), ostream_iterator<int>(cout, " "));
cout << endl;
cout << "v4 = ";
copy(v4.begin(), v4.end(), ostream_iterator<int>(cout, " "));
cout << endl;
```

```
cout << "v5 = ";
copy(v5.begin(), v5.end(), ostream_iterator<int>(cout, " "));
cout << endl;
cout << "v6 = ";
copy(v6.begin(), v6.end(), ostream_iterator<int>(cout, " "));
cout << endl;
```

可以利用上述的方法构造一个 `vector`，足够我们使用了。

元素访问

`vector` 提供了如下几种方法进行元素访问

1. `at()`

   `v.at(pos)` 返回容器中下标为 `pos` 的引用。如果数组越界抛出 `std::out_of_range` 类型的异常。
2. `operator[]`

   `v[pos]` 返回容器中下标为 `pos` 的引用。不执行越界检查。
3. `front()`

   `v.front()` 返回首元素的引用。
4. `back()`

   `v.back()` 返回末尾元素的引用。
5. `data()`

   `v.data()` 返回指向数组第一个元素的指针。

迭代器

vector 提供了如下几种 迭代器

1. `begin()/cbegin()`

   返回指向首元素的迭代器，其中 `*begin = front`。
2. `end()/cend()`

   返回指向数组尾端占位符的迭代器，注意是没有元素的。
3. `rbegin()/crbegin()`

   返回指向逆向数组的首元素的逆向迭代器，可以理解为正向容器的末元素。
4. `rend()/crend()`

   返回指向逆向数组末元素后一位置的迭代器，对应容器首的前一个位置，没有元素。

以上列出的迭代器中，含有字符 `c` 的为只读迭代器，你不能通过只读迭代器去修改 `vector` 中的元素的值。如果一个 `vector` 本身就是只读的，那么它的一般迭代器和只读迭代器完全等价。只读迭代器自 C++11 开始支持。

长度和容量

`vector` 有以下几个与容器长度和容量相关的函数。注意，`vector` 的长度（size）指有效元素数量，而容量（capacity）指其实际分配的内存长度，相关细节请参见后文的实现细节介绍。

与长度相关：

- `empty()` 返回一个 `bool` 值，即 `v.begin() == v.end()`，`true` 为空，`false` 为非空。
- `size()` 返回容器长度（元素数量），即 `std::distance(v.begin(), v.end())`。
- `resize()` 改变 `vector` 的长度，多退少补。补充元素可以由参数指定。
- `max_size()` 返回容器的最大可能长度。

与容量相关：

- `reserve()` 使得 `vector` 预留一定的内存空间，避免不必要的内存拷贝。

- capacity() 返回容器的容量，即不发生拷贝的情况下容器的长度上限。
- shrink_to_fit() 使得 vector 的容量与长度一致，多退但不会少。

元素增删及修改

- clear() 清除所有元素
- insert() 支持在某个迭代器位置插入元素、可以插入多个。**复杂度与 pos 距离末尾长度成线性而非常数的**
- erase() 删除某个迭代器或者区间的元素，返回最后被删除的迭代器。复杂度与 insert 一致。
- push_back() 在末尾插入一个元素，均摊复杂度为**常数**，最坏为线性复杂度。
- pop_back() 删除末尾元素，常数复杂度。
- swap() 与另一个容器进行交换，此操作是**常数复杂度**而非线性的。

vector 的实现细节

vector 的底层其实仍然是定长数组，它能够实现动态扩容的原因是增加了避免数量溢出的操作。首先需要指明的是 vector 中元素的数量（长度）n 与它已分配内存最多能包含元素的数量（容量）N 是不一致的，vector 会分开存储这两个量。当向 vector 中添加元素时，如发现 $n > N$，那么容器会分配一个尺寸为 $2N$ 的数组，然后将旧数据从原本的位置拷贝到新的数组中，再将原来的内存释放。尽管这个操作的渐进复杂度是 $O(n)$，但是可以证明其均摊复杂度为 $O(1)$。而在末尾删除元素和访问元素则都仍然是 $O(1)$ 的开销。因此，只要对 vector 的尺寸估计得当并善用 resize() 和 reserve()，就能使得 vector 的效率与定长数组不会有太大差距。

vector<bool>

标准库特别提供了对 bool 的 vector 特化，每个「bool」只占 1 bit，且支持动态增长。但是其 operator[] 的返回值的类型不是 bool& 而是 vector<bool>::reference。因此，使用 vector<bool> 使需谨慎，可以考虑使用 deque<bool> 或 vector<char> 替代。而如果你需要节省空间，请直接使用 bitset。

array(C++11)

std::array 是 STL 提供的**内存连续的**、**固定长度**的数组数据结构。其本质是对原生数组的直接封装。

为什么要用 array

array 实际上是 STL 对数组的封装。它相比 vector 牺牲了动态扩容的特性，但是换来了与原生数组几乎一致的性能（在开满优化的前提下）。因此如果能使用 C++11 特性的情况下，能够使用原生数组的地方几乎都可以直接把定长数组都换成 array，而动态分配的数组可以替换为 vector。

成员函数

隐式定义的成员函数

函数	作用
operator=	以来自另一 array 的每个元素重写 array 的对应元素

元素访问

函数	作用
at	访问指定的元素，同时进行越界检查
operator[]	访问指定的元素，**不**进行越界检查
front	访问第一个元素
back	访问最后一个元素
data	返回指向内存中数组第一个元素的指针

at 若遇 pos >= size() 的情况会抛出 std::out_of_range。

容量

函数	作用
empty	检查容器是否为空
size	返回容纳的元素数
max_size	返回可容纳的最大元素数

由于每个 array 都是固定大小容器，size() 返回的值等于 max_size() 返回的值。

操作

函数	作用
fill	以指定值填充容器
swap	交换内容

注意，交换两个 array 是 $\Theta(\text{size})$ 的，而非与常规 STL 容器一样为 $O(1)$。

非成员函数

函数	作用
operator== 等	按照字典序比较 array 中的值
std::get	访问 array 的一个元素
std::swap	特化的 std::swap 算法

下面是一个 array 的使用示例：

```cpp
// 1. 创建空 array，长度为 3；常数复杂度
std::array<int, 3> v0;
// 2. 用指定常数创建 array；常数复杂度
std::array<int, 3> v1{1, 2, 3};

v0.fill(1);  // 填充数组

// 访问数组
for (int i = 0; i != arr.size(); ++i) cout << arr[i] << " ";
```

deque

std::deque 是 STL 提供的 双端队列 数据结构。能够提供线性复杂度的插入和删除，以及常数复杂度的随机访问。

deque 的使用方法

以下介绍常用用法，详细内容 请参见 C++ 文档[3]。deque 的迭代器函数与 vector 相同，因此不作详细介绍。

构造函数

参见如下代码（假设你已经 using 了 std 命名空间相关类型）：

```
// 1. 定义一个 int 类型的空双端队列 v0
deque<int> v0;
// 2. 定义一个 int 类型的双端队列 v1，并设置初始大小为 10; 线性复杂度
deque<int> v1(10);
// 3. 定义一个 int 类型的双端队列 v2，并初始化为 10 个 1; 线性复杂度
deque<int> v2(10, 1);
// 4. 复制已有的双端队列 v1; 线性复杂度
deque<int> v3(v1);
// 5. 创建一个 v2 的拷贝 deque v4，其内容是 v4[0] 至 v4[2]; 线性复杂度
deque<int> v4(v2.begin(), v2.begin() + 3);
// 6. 移动 v2 到新创建的 deque v5，不发生拷贝; 常数复杂度; 需要 C++11
deque<int> v5(std::move(v2));
```

元素访问

与 `vector` 一致，但无法访问底层内存。其高效的元素访问速度可参考实现细节部分。

- `at()` 返回容器中指定位置元素的引用，执行越界检查，**常数复杂度**。
- `operator[]` 返回容器中指定位置元素的引用。不执行越界检查，**常数复杂度**。
- `front()` 返回首元素的引用。
- `back()` 返回末尾元素的引用。

迭代器

与 `vector` 一致。

长度

与 `vector` 一致，但是没有 `reserve()` 和 `capacity()` 函数。（仍然有 `shrink_to_fit()` 函数）

元素增删及修改

与 `vector` 一致，并额外有向队列头部增加元素的函数。

- `clear()` 清除所有元素
- `insert()` 支持在某个迭代器位置插入元素、可以插入多个。**复杂度与 `pos` 与两端距离较小者成线性**。
- `erase()` 删除某个迭代器或者区间的元素，返回最后被删除的迭代器。复杂度与 `insert` 一致。
- `push_front()` 在头部插入一个元素，**常数复杂度**。
- `pop_front()` 删除头部元素，**常数复杂度**。
- `push_back()` 在末尾插入一个元素，**常数复杂度**。
- `pop_back()` 删除末尾元素，**常数复杂度**。
- `swap()` 与另一个容器进行交换，此操作是**常数复杂度**而非线性的。

deque 的实现细节

`deque` 通常的底层实现是多个不连续的缓冲区，而缓冲区中的内存是连续的。而每个缓冲区还会记录首指针和尾指针，用来标记有效数据的区间。当一个缓冲区填满之后便会在之前或者之后分配新的缓冲区来存储更多的数据。更详细的说明可以参考 STL 源码剖析 ——deque 的实现原理和使用方法详解[4]。

list

`std::list` 是 STL 提供的 双向链表 数据结构。能够提供线性复杂度的随机访问，以及常数复杂度的插入和删除。

list 的使用方法

list 的使用方法与 deque 基本相同，但是增删操作和访问的复杂度不同。详细内容 请参见 C++ 文档[5]。list 的迭代器、长度、元素增删及修改相关的函数与 deque 相同，因此不作详细介绍。

元素访问

由于 list 的实现是链表，因此它不提供随机访问的接口。若需要访问中间元素，则需要使用迭代器。

- front() 返回首元素的引用。
- back() 返回末尾元素的引用。

操作

list 类型还提供了一些针对其特性实现的 STL 算法函数。由于这些算法需要 随机访问迭代器，因此 list 提供了特别的实现以便于使用。这些算法有 splice()、remove()、sort()、unique()、merge() 等。

forward_list (C++11)

std::forward_list 是 STL 提供的 单向链表 数据结构，相比于 std::list 减小了空间开销。

forward_list 的使用方法

forward_list 的使用方法与 list 几乎一致，但是迭代器只有单向的，因此其具体用法不作详细介绍。详细内容 请参见 C++ 文档[6]

参考资料与注释

[1] 列表初始化

[2] 请参见 C++ 文档

[3] 请参见 C++ 文档

[4] STL 源码剖析——deque 的实现原理和使用方法详解

[5] 请参见 C++ 文档

[6] 请参见 C++ 文档

关联式容器

set

set 是关联容器，含有键值类型对象的已排序集，搜索、移除和插入拥有对数复杂度。set 内部通常采用 红黑树 实现。平衡二叉树 的特性使得 set 非常适合处理需要同时兼顾查找、插入与删除的情况。

和数学中的集合相似，set 中不会出现值相同的元素。如果需要有相同元素的集合，需要使用 multiset。multiset 的使用方法与 set 的使用方法基本相同。

插入与删除操作

- insert(x) 当容器中没有等价元素的时候，将元素 x 插入到 set 中。
- erase(x) 删除值为 x 的**所有**元素，返回删除元素的个数。
- erase(pos) 删除迭代器为 pos 的元素，要求迭代器必须合法。

- `erase(first,last)` 删除迭代器在 $[first, last)$ 范围内的所有元素。
- `clear()` 清空 `set`。

"insert 函数的返回值"

insert 函数的返回值类型为 `pair<iterator, bool>`，其中 iterator 是一个指向所插入元素（或者是指向等于所插入值的原本就在容器中的元素）的迭代器，而 bool 则代表元素是否插入成功，由于 `set` 中的元素具有唯一性质，所以如果在 `set` 中已有等值元素，则插入会失败，返回 false，否则插入成功，返回 true；`map` 中的 insert 也是如此。

迭代器

`set` 提供了以下几种迭代器：

1. `begin()/cbegin()`

   返回指向首元素的迭代器，其中 `*begin = front`。
2. `end()/cend()`

   返回指向数组尾端占位符的迭代器，注意是没有元素的。
3. `rbegin()/crbegin()`

   返回指向逆向数组的首元素的逆向迭代器，可以理解为正向容器的末元素。
4. `rend()/crend()`

   返回指向逆向数组末元素后一位置的迭代器，对应容器首的前一个位置，没有元素。

以上列出的迭代器中，含有字符 `c` 的为只读迭代器，你不能通过只读迭代器去修改 `set` 中的元素的值。如果一个 `set` 本身就是只读的，那么它的一般迭代器和只读迭代器完全等价。只读迭代器自 C++11 开始支持。

查找操作

- `count(x)` 返回 `set` 内键为 x 的元素数量。
- `find(x)` 在 `set` 内存在键为 x 的元素时会返回该元素的迭代器，否则返回 `end()`。
- `lower_bound(x)` 返回指向首个不小于给定键的元素的迭代器。如果不存在这样的元素，返回 `end()`。
- `upper_bound(x)` 返回指向首个大于给定键的元素的迭代器。如果不存在这样的元素，返回 `end()`。
- `empty()` 返回容器是否为空。
- `size()` 返回容器内元素个数。

"`lower_bound` 和 `upper_bound` 的时间复杂度"

`set` 自带的 `lower_bound` 和 `upper_bound` 的时间复杂度为 $O(\log n)$。

但使用 `algorithm` 库中的 `lower_bound` 和 `upper_bound` 函数对 `set` 中的元素进行查询，时间复杂度为 $O(n)$。

"`nth_element` 的时间复杂度"

`set` 没有提供自带的 `nth_element`。使用 `algorithm` 库中的 `nth_element` 查找第 k 大的元素时间复杂度为 $O(n)$。

如果需要实现平衡二叉树所具备的 $O(\log n)$ 查找第 k 大元素的功能，需要自己手写平衡二叉树或权值线段树，或者选择使用 pb_ds 库中的平衡二叉树。

使用样例

set 在贪心中的使用

在贪心算法中经常会需要出现类似**找出并删除最小的大于等于某个值的元素**。这种操作能轻松地通过 set 来完成。

```
// 现存可用的元素
set<int> available;
// 需要大于等于的值
int x;

// 查找最小的大于等于 x 的元素
set<int>::iterator it = available.lower_bound(x);
if (it == available.end()) {
  // 不存在这样的元素，则进行相应操作……
} else {
  // 找到了这样的元素，将其从现存可用元素中移除
  available.erase(it);
  // 进行相应操作……
}
```

map

map 是有序键值对容器，它的元素的键是唯一的。搜索、移除和插入操作拥有对数复杂度。map 通常实现为 红黑树。

设想如下场景：现在需要存储一些键值对，例如存储学生姓名对应的分数：Tom 0，Bob 100，Alan 100。但是由于数组下标只能为非负整数，所以无法用姓名作为下标来存储，这个时候最简单的办法就是使用 STL 中的 map。

map 重载了 operator[]，可以用任意定义了 operator < 的类型作为下标（在 map 中叫做 key，也就是索引）：

```
map<Key, T> yourMap;
```

其中，Key 是键的类型，T 是值的类型，下面是使用 map 的实例：

```
map<string, int> mp;
```

map 中不会存在键相同的元素，multimap 中允许多个元素拥有同一键。multimap 的使用方法与 map 的使用方法基本相同。

> warning
>
> 正是因为 multimap 允许多个元素拥有同一键的特点，multimap 并没有提供给出键访问其对应值的方法。

插入与删除操作

- 可以直接通过下标访问来进行查询或插入操作。例如 mp["Alan"]=100。
- 通过向 map 中插入一个类型为 pair<Key, T> 的值可以达到插入元素的目的，例如 mp.insert(pair<string,int>("Alan",100));；
- erase(key) 函数会删除键为 key 的**所有**元素。返回值为删除元素的数量。
- erase(pos): 删除迭代器为 pos 的元素，要求迭代器必须合法。
- erase(first,last): 删除迭代器在 $[first, last)$ 范围内的所有元素。
- clear() 函数会清空整个容器。

> " 下标访问中的注意事项"
>
> 在利用下标访问 map 中的某个元素时，如果 map 中不存在相应键的元素，会自动在 map 中插入一个新元

素，并将其值设置为默认值（对于整数，值为零；对于有默认构造函数的类型，会调用默认构造函数进行初始化）。

当下标访问操作过于频繁时，容器中会出现大量无意义元素，影响 `map` 的效率。因此一般情况下推荐使用 `find()` 函数来寻找特定键的元素。

查询操作

- `count(x)`: 返回容器内键为 x 的元素数量。复杂度为 $O(\log(size)+ans)$（关于容器大小对数复杂度，加上匹配个数）。
- `find(x)`: 若容器内存在键为 x 的元素，会返回该元素的迭代器；否则返回 `end()`。
- `lower_bound(x)`: 返回指向首个不小于给定键的元素的迭代器。
- `upper_bound(x)`: 返回指向首个大于给定键的元素的迭代器。若容器内所有元素均小于或等于给定键，返回 `end()`。
- `empty()`: 返回容器是否为空。
- `size()`: 返回容器内元素个数。

使用样例

`map` 用于存储复杂状态

在搜索中，我们有时需要存储一些较为复杂的状态（如坐标，无法离散化的数值，字符串等）以及与之有关的答案（如到达此状态的最小步数）。`map` 可以用来实现此功能。其中的键是状态，而值是与之相关的答案。下面的示例展示了如何使用 `map` 存储以 `string` 表示的状态。

```
// 存储状态与对应的答案
map<string, int> record;

// 新搜索到的状态与对应答案
string status;
int ans;
// 查找对应的状态是否出现过
map<string, int>::iterator it = record.find(status);
if (it == record.end()) {
  // 尚未搜索过该状态，将其加入状态记录中
  record[status] = ans;
  // 进行相应操作……
} else {
  // 已经搜索过该状态，进行相应操作……
}
```

遍历容器

可以利用迭代器来遍历关联式容器的所有元素。

```
set<int> s;
typedef set<int>::iterator si;
for (si it = s.begin(); it != s.end(); it++) cout << *it << endl;
```

需要注意的是，对 `map` 的迭代器解引用后，得到的是类型为 `pair<Key, T>` 的键值对。

在 C++11 中，使用范围 for 循环会让代码简洁很多：

```
set<int> s;
for (auto x : s) cout << x << endl;
```

对于任意关联式容器，使用迭代器遍历容器的时间复杂度均为 $O(n)$。

自定义比较方式

`set` 在默认情况下的比较函数为 `<`（如果是非内置类型需要 重载 `<` 运算符）。然而在某些特殊情况下，我们希望能自定义 `set` 内部的比较方式。

这时候可以通过传入自定义比较器来解决问题。

具体来说，我们需要定义一个类，并在这个类中 重载 `()` 运算符。

例如，我们想要维护一个存储整数，且较大值靠前的 `set`，可以这样实现：

```cpp
struct cmp {
  bool operator()(int a, int b) { return a > b; }
};

set<int, cmp> s;
```

对于其他关联式容器，可以用类似的方式实现自定义比较，这里不再赘述。

无序关联式容器

概述

自 C++11 标准起，四种基于 哈希 实现的无序关联式容器正式纳入了 C++ 的标准模板库中，分别是：`unordered_set`，`unordered_multiset`，`unordered_map`，`unordered_multimap`。

> " 编译器不支持 C++11 的使用方法"
>
> 在 C++11 之前，无序关联式容器属于 C++ 的 TR1 扩展。所以，如果编译器不支持 C++11，在使用时需要在头文件的名称中加入 `tr1/` 前缀，并且使用 `std::tr1` 命名空间。如 `#include <unordered_map>` 需要改成 `#include <tr1/unordered_map>`；`std::unordered_map` 需要改为 `std::tr1::unordered_map`（如果使用 `using namespace std;`，则为 `tr1::unordered_map`）。

它们与相应的关联式容器在功能，函数等方面有诸多共同点，而最大的不同点则体现在普通的关联式容器一般采用红黑树实现，内部元素按特定顺序进行排序；而这几种无序关联式容器则采用哈希方式存储元素，内部元素不以任何特定顺序进行排序，所以访问无序关联式容器中的元素时，访问顺序也没有任何保证。

采用哈希存储的特点使得无序关联式容器**在平均情况下**大多数操作（包括查找，插入，删除）都能在常数时间复杂度内完成，相较于关联式容器与容器大小成对数的时间复杂度更加优秀。

> warning
>
> 在最坏情况下，对无序关联式容器进行插入、删除、查找等操作的时间复杂度会**与容器大小成线性关系**！这一情况往往在容器内出现大量哈希冲突时产生。
>
> 同时，由于无序关联式容器的操作时通常存在较大的常数，其效率有时并不比普通的关联式容器好太多。
>
> 因此应谨慎使用无序关联式容器，尽量避免滥用（例如懒得离散化，直接将 `unordered_map<int, int>` 当作空间无限的普通数组使用）。

由于无序关联式容器与相应的关联式容器在用途和操作中有很多共同点，这里不再介绍无序关联式容器的各种操作，这些内容读者可以参考 关联式容器。

制造哈希冲突

上文中提到了，在最坏情况下，对无序关联式容器进行一些操作的时间复杂度会与容器大小成线性关系。

在哈希函数确定的情况下，可以构造出数据使得容器内产生大量哈希冲突，导致复杂度达到上界。

在标准库实现里，每个元素的散列值是将值对一个质数取模得到的，更具体地说，是 这个列表[1] 中的质数（g++ 6 及以前版本的编译器，这个质数一般是 126271，g++ 7 及之后版本的编译器，这个质数一般是 107897）。

因此可以通过向容器中插入这些模数的倍数来达到制造大量哈希冲突的目的。

自定义哈希函数

使用自定义哈希函数可以有效避免构造数据产生的大量哈希冲突。

要想使用自定义哈希函数，需要定义一个结构体，并在结构体中重载 `()` 运算符，像这样：

```
struct my_hash {
  size_t operator()(int x) const { return x; }
};
```

当然，为了确保哈希函数不会被迅速破解（例如 Codeforces 中对使用无序关联式容器的提交进行 hack），可以试着在哈希函数中加入一些随机化函数（如时间）来增加破解的难度。

例如，在 这篇博客[2] 中给出了如下哈希函数：

```
struct my_hash {
  static uint64_t splitmix64(uint64_t x) {
    x += 0x9e3779b97f4a7c15;
    x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
    x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
    return x ^ (x >> 31);
  }

  size_t operator()(uint64_t x) const {
    static const uint64_t FIXED_RANDOM =
        chrono::steady_clock::now().time_since_epoch().count();
    return splitmix64(x + FIXED_RANDOM);
  }

  // 针对 std::pair<int, int> 作为主键类型的哈希函数
  size_t operator()(pair<uint64_t, uint64_t> x) const {
    static const uint64_t FIXED_RANDOM =
        chrono::steady_clock::now().time_since_epoch().count();
    return splitmix64(x.first + FIXED_RANDOM) ^
           (splitmix64(x.second + FIXED_RANDOM) >> 1);
  }
};
```

写完自定义的哈希函数后，就可以通过 `unordered_map<int, int, my_hash> my_map;` 或者 `unordered_map<pair<int, int>, int, my_hash> my_pair_map;` 的定义方式将自定义的哈希函数传入容器了。

参考资料与注释

[1] 这个列表

[2] 这篇博客

容器适配器

Authors: Xeonacid, ksyx, Early0v0

栈

STL 栈(`std::stack`) 是一种后进先出 (Last In, First Out) 的容器适配器，仅支持查询或删除最后一个加入的元素（栈顶元素），不支持随机访问，且为了保证数据的严格有序性，不支持迭代器。

头文件

```
#include <stack>
```

定义

```
std::stack<TypeName> s;  // 使用默认底层容器 deque，数据类型为 TypeName
std::stack<TypeName, Container> s;  // 使用 Container 作为底层容器
std::stack<TypeName> s2(s1);        // 将 s1 复制一份用于构造 s2
```

成员函数

以下所有函数均为常数复杂度

- `top()` 访问栈顶元素（如果栈为空，此处会出错）
- `push(x)` 向栈中插入元素 x
- `pop()` 删除栈顶元素
- `size()` 查询容器中的元素数量
- `empty()` 询问容器是否为空

简单示例

```
std::stack<int> s1;
s1.push(2);
s1.push(1);
std::stack<int> s2(s1);
s1.pop();
std::cout << s1.size() << " " << s2.size() << std::endl;  // 1 2
std::cout << s1.top() << " " << s2.top() << std::endl;    // 2 1
s1.pop();
std::cout << s1.empty() << " " << s2.empty() << std::endl;  // 1 0
```

队列

STL 队列(`std::queue`) 是一种先进先出 (First In, First Out) 的容器适配器，仅支持查询或删除第一个加入的元素（队首元素），不支持随机访问，且为了保证数据的严格有序性，不支持迭代器。

头文件

```
#include <queue>
```

定义

```
std::queue<TypeName> q;  // 使用默认底层容器 deque，数据类型为 TypeName
std::queue<TypeName, Container> q;  // 使用 Container 作为底层容器

std::queue<TypeName> q2(q1);  // 将 s1 复制一份用于构造 q2
```

成员函数

以下所有函数均为常数复杂度

- `front()` 访问队首元素（如果队列为空，此处会出错）
- `push(x)` 向队列中插入元素 x

- pop() 删除队首元素
- size() 查询容器中的元素数量
- empty() 询问容器是否为空

简单示例

```
std::queue<int> q1;
q1.push(2);
q1.push(1);
std::queue<int> q2(q1);
q1.pop();
std::cout << q1.size() << " " << q2.size() << std::endl;    // 1 2
std::cout << q1.front() << " " << q2.front() << std::endl;  // 1 2
q1.pop();
std::cout << q1.empty() << " " << q2.empty() << std::endl;  // 1 0
```

优先队列

优先队列 std::priority_queue 是一种 堆，一般为 二叉堆。

头文件

```
#include <queue>
```

定义

```
std::priority_queue<TypeName> q;             // 数据类型为 TypeName
std::priority_queue<TypeName, Container> q;  // 使用 Container 作为底层容器
std::priority_queue<TypeName, Container, Compare> q;
// 使用 Container 作为底层容器，使用 Compare 作为比较类型

// 默认使用底层容器 vector
// 比较类型 less<TypeName>（此时为它的 top() 返回为最大值）
// 若希望 top() 返回最小值，可令比较类型为 greater<TypeName>
// 注意：不可跳过 Container 直接传入 Compare

// 从 C++11 开始，如果使用 lambda 函数自定义 Compare
// 则需要将其作为构造函数的参数代入，如：
auto cmp = [](const std::pair<int, int> &l, const std::pair<int, int> &r) {
  return l.second < r.second;
};
std::priority_queue<std::pair<int, int>, std::vector<std::pair<int, int> >,
                    decltype(cmp)>
    pq(cmp);
```

成员函数

以下所有函数均为常数复杂度

- top() 访问堆顶元素（此时优先队列不能为空）
- empty() 询问容器是否为空
- size() 查询容器中的元素数量

以下所有函数均为对数复杂度

- push(x) 插入元素，并对底层容器排序
- pop() 删除堆顶元素（此时优先队列不能为空）

简单示例

```
std::priority_queue<int> q1;
std::priority_queue<int, std::vector<int> > q2;
// C++11 后空格可省略
std::priority_queue<int, std::deque<int>, std::greater<int> > q3;
// q3 为小根堆
for (int i = 1; i <= 5; i++) q1.push(i);
// q1 中元素 :  [1, 2, 3, 4, 5]
std::cout << q1.top() << std::endl;
// 输出结果 : 5
q1.pop();
// 堆中元素 : [1, 2, 3, 4]
std::cout << q1.size() << std::endl;
// 输出结果 : 4
for (int i = 1; i <= 5; i++) q3.push(i);
// q3 中元素 :  [1, 2, 3, 4, 5]
std::cout << q3.top() << std::endl;
// 输出结果 : 1
```

4.3.3 STL 算法

STL 提供了大约 100 个实现算法的模版函数，基本都包含在 `<algorithm>` 之中，还有一部分包含在 `<numeric>` 和 `<functional>`。完备的函数列表请 参见参考手册[1]，排序相关的可以参考 排序内容的对应页面。

- find：顺序查找。find(v.begin(), v.end(), value)，其中 value 为需要查找的值。
- reverse：翻转数组、字符串。reverse(v.begin(), v.end()) 或 reverse(a + begin, a + end)。
- unique：去除容器中相邻的重复元素。unique(ForwardIterator first, ForwardIterator last)，返回值为指向**去重后**容器结尾的迭代器，原容器大小不变。与 sort 结合使用可以实现完整容器去重。
- random_shuffle：随机地打乱数组。random_shuffle(v.begin(), v.end()) 或 random_shuffle(v + begin, v + end)。

  "random_shuffle 函数在最新 C++ 标准中已被移除"

  random_shuffle 自 C++14 起被弃用，C++17 起被移除。

  在 C++11 以及更新的标准中，您可以使用 shuffle 函数代替原来的 random_shuffle。使用方法为 shuffle(v.begin(), v.end(), rng)（最后一个参数传入的是使用的随机数生成器，一般情况使用以真随机数生成器 random_device[2] 播种的梅森旋转伪随机数生成器 mt19937[3]）。

  ```
  // #include <random>
  std::mt19937 rng(std::random_device{}());
  std::shuffle(v.begin(), v.end(), rng);
  ```

- sort：排序。sort(v.begin(), v.end(), cmp) 或 sort(a + begin, a + end, cmp)，其中 end 是排序的数组最后一个元素的后一位，cmp 为自定义的比较函数。
- stable_sort：稳定排序，用法同 sort()。
- nth_element：按指定范围进行分类，即找出序列中第 n 大的元素，使其左边均为小于它的数，右边均为大于它的数。nth_element(v.begin(), v.begin() + mid, v.end(), cmp) 或 nth_element(a + begin, a + begin + mid, a + end, cmp)。

- binary_search：二分查找。binary_search(v.begin(), v.end(), value)，其中 value 为需要查找的值。
- merge：将两个（已排序的）序列**有序合并**到第三个序列的**插入迭代器**上。merge(v1.begin(), v1.end(), v2.begin(), v2.end() ,back_inserter(v3))。
- inplace_merge：将两个（已按小于运算符排序的）：[first,middle), [middle,last) 范围**原地合并为一个有序序列**。inplace_merge(v.begin(), v.begin() + middle, v.end())。
- lower_bound：在一个有序序列中进行二分查找，返回指向第一个**大于等于** x 的元素的位置的迭代器。如果不存在这样的元素，则返回尾迭代器。lower_bound(v.begin(),v.end(),x)。
- upper_bound：在一个有序序列中进行二分查找，返回指向第一个**大于** x 的元素的位置的迭代器。如果不存在这样的元素，则返回尾迭代器。upper_bound(v.begin(),v.end(),x)。

"lower_bound 和 upper_bound 的时间复杂度"

在一般的数组里，这两个函数的时间复杂度均为 $O(\log n)$，但在 set 等关联式容器中，直接调用 lower_bound(s.begin(),s.end(),val) 的时间复杂度是 $O(n)$ 的。

set 等关联式容器中已经封装了 lower_bound 等函数（像 s.lower_bound(val) 这样），这样调用的时间复杂度是 $O(\log n)$ 的。

- next_permutation：将当前排列更改为**全排列中的下一个排列**。如果当前排列已经是**全排列中的最后一个排列**（元素完全从大到小排列），函数返回 false 并将排列更改为**全排列中的第一个排列**（元素完全从小到大排列）；否则，函数返回 true。next_permutation(v.begin(), v.end()) 或 next_permutation(v + begin, v + end)。
- prev_permutation：将当前排列更改为**全排列中的上一个排列**。用法同 next_permutation。
- partial_sum：求前缀和。设源容器为 x，目标容器为 y，则令 $y[i]=x[0]+x[1]+\cdots+x[i]$。partial_sum(src.begin(), src.end(), back_inserter(dst))。

使用样例

- 使用 next_permutation 生成 1 到 9 的全排列。例题：Luogu P1706 全排列问题[4]

" 实现"

```
int N = 9, a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
do {
  for (int i = 0; i < N; i++) cout << a[i] << " ";
  cout << endl;
} while (next_permutation(a, a + N));
```

- 使用 lower_bound 与 upper_bound 查找有序数组 a 中小于 x，等于 x，大于 x 元素的分界线。

" 实现"

```
int N = 10, a[] = {1, 1, 2, 4, 5, 5, 7, 7, 9, 9}, x = 5;
int i = lower_bound(a, a + N, x) - a, j = upper_bound(a, a + N, x) - a;
// a[0] ~ a[i - 1] 为小于 x 的元素， a[i] ~ a[j - 1] 为等于 x 的元素，
// a[j] ~ a[N - 1] 为大于 x 的元素
cout << i << " " << j << endl;
```

- 使用 partial_sum 求解 src 中元素的前缀和，并存储于 dst 中。

" 实现"

```
vector<int> src = {1, 2, 3, 4, 5}, dst;
```

```
// 求解 src 中元素的前缀和，dst[i] = src[0] + ... + src[i]
// back_inserter 函数作用在 dst 容器上，提供一个迭代器
partial_sum(src.begin(), src.end(), back_inserter(dst));
for (unsigned int i = 0; i < dst.size(); i++) cout << dst[i] << " ";
```

- 使用 `lower_bound` 查找有序数组 a 中最接近 x 的元素。例题：UVa10487 Closest Sums[5]

" 实现"

```
int N = 10, a[] = {1, 1, 2, 4, 5, 5, 8, 8, 9, 9}, x = 6;
// lower_bound 将返回 a 中第一个大于等于 x 的元素的地址，计算出的 i 为其下标
int i = lower_bound(a, a + N, x) - a;
// 在以下两种情况下，a[i] (a 中第一个大于等于 x 的元素) 即为答案：
// 1. a 中最小的元素都大于等于 x;
// 2. a 中存在大于等于 x 的元素，且第一个大于等于 x 的元素 (a[i])
// 相比于第一个小于 x 的元素 (a[i - 1]) 更接近 x;
// 否则，a[i - 1] (a 中第一个小于 x 的元素) 即为答案
if (i == 0 || (i < N && a[i] - x < x - a[i - 1]))
  cout << a[i];
else
  cout << a[i - 1];
```

- 使用 `sort` 与 `unique` 查找数组 a 中**第 k 小的值**（注意：重复出现的值仅算一次，因此本题不是求解第 k 小的元素）。例题：Luogu P1138 第 k 小整数[6]

" 实现"

```
int N = 10, a[] = {1, 3, 3, 7, 2, 5, 1, 2, 4, 6}, k = 3;
sort(a, a + N);
// unique 将返回去重之后数组最后一个元素之后的地址，计算出的 cnt 为去重后数组的长度
int cnt = unique(a, a + N) - a;
cout << a[k - 1];
```

参考资料与注释

[1] 参见参考手册

[2] random_device

[3] mt19937

[4] Luogu P1706 全排列问题

[5] UVa10487 Closest Sums

[6] Luogu P1138 第 k 小整数

4.3.4 bitset

Authors: i-Yirannn, Xeonacid, ouuan

介绍

`std::bitset` 是标准库中的一个存储 `0/1` 的大小不可变容器。严格来讲，它并不属于 STL。

"bitset 与 STL"

"bitset 与 STL"

The C++ standard library provides some special container classes, the so-called container adapters (stack, queue, priority queue). In addition, a few classes provide a container-like interface (for example, strings, bitsets, and valarrays). All these classes are covered separately.1 Container adapters and bitsets are covered in Chapter 12.

The C++ standard library provides not only the containers for the STL framework but also some containers that fit some special needs and provide simple, almost self-explanatory, interfaces. You can group these containers into either the so-called container adapters, which adapt standard STL containers to fit special needs, or a bitset, which is a containers for bits or Boolean values. There are three standard container adapters: stacks, queues, and priority queues. In priority queues, the elements are sorted automatically according to a sorting criterion. Thus, the "next" element of a priority queue is the element with the "highest" value. A bitset is a bitfield with an arbitrary but fixed number of bits. Note that the C++ standard library also provides a special container with a variable size for Boolean values: vector.

——摘自《The C++ Standard Library 2nd Edition》

由此看来，`bitset` 并不属于 STL，而是一种标准库中的"Special Container"。事实上，它作为一种容器，也并不满足 STL 容器的要求。说它是适配器，它也并不依赖于其它 STL 容器作为底层实现。

由于内存地址是按字节即 `byte` 寻址，而非比特 `bit`，一个 `bool` 类型的变量，虽然只能表示 `0/1`, 但是也占了 1 byte 的内存。

`bitset` 就是通过固定的优化，使得一个字节的八个比特能分别储存 8 位的 `0/1`。

对于一个 4 字节的 `int` 变量，在只存 `0/1` 的意义下，`bitset` 占用空间只是其 $\frac{1}{32}$，计算一些信息时，所需时间也是其 $\frac{1}{32}$。

在某些情况下通过 `bitset` 可以优化程序的运行效率。至于其优化的是复杂度还是常数，要看计算复杂度的角度。一般 `bitset` 的复杂度有以下几种记法：（设原复杂度为 $O(n)$）

1. $O(n)$，这种记法认为 `bitset` 完全没有优化复杂度。
2. $O(\frac{n}{32})$，这种记法不太严谨（复杂度中不应出现常数），但体现了 `bitset` 能将所需时间优化至 $\frac{1}{32}$。
3. $O(\frac{n}{w})$，其中 $w = 32$（计算机的位数），这种记法较为普遍接受。
4. $O(\frac{n}{\log w})$，其中 w 为计算机一个整型变量的大小。

当然，`vector` 的一个特化 `vector<bool>` 的储存方式同 `bitset` 一样，区别在于其支持动态开空间，`bitset` 则和我们一般的静态数组一样，是在编译时就开好了的。

然而，`bitset` 有一些好用的库函数，不仅方便，而且有时可以避免使用 for 循环而没有实质的速度优化。因此，一般不使用 `vector<bool>`。

使用

头文件

```
#include <bitset>
```

指定大小

```
bitset<1000> bs;  // a bitset with 1000 bits
```

构造函数

- `bitset()`: 每一位都是 `false`。
- `bitset(unsigned long val)`: 设为 `val` 的二进制形式。
- `bitset(const string& str)`: 设为 01 串 `str`。

运算符

- `operator []`: 访问其特定的一位。
- `operator ==/!=`: 比较两个 `bitset` 内容是否完全一样。
- `operator &/&=/|/| =/^/^=/~`: 进行按位与/或/异或/取反操作。**`bitset` 只能与 `bitset` 进行位运算**，若要和整型进行位运算，要先将整型转换为 `bitset`。
- `operator <</>>/<<=/>>=`: 进行二进制左移/右移。
- `operator <</>>`: 流运算符，这意味着你可以通过 `cin/cout` 进行输入输出。

成员函数

- `count()`: 返回 `true` 的数量。
- `size()`: 返回 `bitset` 的大小。
- `test(pos)`: 它和 `vector` 中的 `at()` 的作用是一样的，和 `[]` 运算符的区别就是越界检查。
- `any()`: 若存在某一位是 `true` 则返回 `true`，否则返回 `false`。
- `none()`: 若所有位都是 `false` 则返回 `true`，否则返回 `false`。
- `all()`:**C++11**，若所有位都是 `true` 则返回 `true`，否则返回 `false`。
- 1. `set()`: 将整个 `bitset` 设置成 `true`。
  2. `set(pos, val = true)`: 将某一位设置成 `true/false`。
- 1. `reset()`: 将整个 `bitset` 设置成 `false`。
  2. `reset(pos)`: 将某一位设置成 `false`。相当于 `set(pos, false)`。
- 1. `flip()`: 翻转每一位。（$0 \leftrightarrow 1$，相当于异或一个全是 1 的 `bitset`）
  2. `flip(pos)`: 翻转某一位。
- `to_string()`: 返回转换成的字符串表达。
- `to_ulong()`: 返回转换成的 `unsigned long` 表达（`long` 在 NT 及 32 位 POSIX 系统下与 `int` 一样，在 64 位 POSIX 下与 `long long` 一样）。
- `to_ullong()`:**C++11**，返回转换成的 `unsigned long long` 表达。

一些文档中没有的成员函数：

- `_Find_first()`: 返回 `bitset` 第一个 `true` 的下标，若没有 `true` 则返回 `bitset` 的大小。
- `_Find_next(pos)`: 返回 `pos` 后面（下标严格大于 `pos` 的位置）第一个 `true` 的下标，若 `pos` 后面没有 `true` 则返回 `bitset` 的大小。

应用

「LibreOJ Round #2」贪心只能过样例[1]

这题可以用 dp 做，转移方程很简单：

$f(i,j)$ 表示前 i 个数的平方和能否为 j，那么 $f(i,j)=\bigvee\limits_{k=a}^{b} f(i-1,j-k^2)$（或起来）。

但如果直接做的话是 $O(n^5)$ 的，（看起来）过不了。

发现可以用 `bitset` 优化，左移再或起来就好了：std::bitset[2]

然后发现……被加了几个剪枝的暴力艹了：加了几个剪枝的暴力[3]

然而，可以手写 `bitset`（只需要支持左移后或起来这一种操作）压 64 位（`unsigned long long`）来艹掉暴力：手写 bitset[4]

CF1097F Alex and a TV Show[5]

题意

给你 n 个可重集，四种操作：

1. 把某个可重集设为一个数。
2. 把某个可重集设为另外两个可重集加起来。
3. 把某个可重集设为从另外两个可重集中各选一个数的 gcd。即：$A = \{\gcd(x, y) | x \in B, y \in C\}$。
4. 询问某个可重集中某个数的个数，**在模 2 意义下**。

可重集个数 10^5，操作个数 10^6，值域 7000。

做法

看到「在模 2 意义下」，可以想到用 `bitset` 维护每个可重集。

这样的话，操作 1 直接设，操作 2 就是异或（因为模 2），操作 4 就是直接查，但.. 操作 3 怎么办？

我们可以尝试维护每个可重集的所有约数构成的可重集，这样的话，操作 3 就是直接按位与。

我们可以把值域内每个数的约数构成的 `bitset` 预处理出来，这样操作 1 就解决了。操作 2 仍然是异或。

现在的问题是，如何通过一个可重集的约数构成的可重集得到该可重集中某个数的个数。

令原可重集为 A，其约数构成的可重集为 A'，我们要求 A 中 x 的个数，用 莫比乌斯反演 推一推：

$$
\begin{aligned}
&\sum_{i \in A} [\frac{i}{x} = 1] \\
=&\sum_{i \in A} \sum_{d | \frac{i}{x}} \mu(d) \\
=&\sum_{d \in A', x | d} \mu(\frac{d}{x})
\end{aligned}
$$

由于是模 2 意义下，-1 和 1 是一样的，只用看 $\frac{d}{x}$ 有没有平方因子即可。所以，可以对值域内每个数预处理出其倍数中除以它不含平方因子的位置构成的 `bitset`，求答案的时候先按位与再 `count()` 就好了。

这样的话，单次询问复杂度就是 $O(\frac{v}{w})$（$v = 7000$, $w = 32$）。

至于预处理的部分，$O(v\sqrt{v})$ 或者 $O(v^2)$ 预处理比较简单，log 预处理就如下面代码所示，复杂度为调和级数，所以是 $O(v \log v)$。

" 参考代码"

```cpp
#include <bitset>
#include <cctype>
#include <cmath>
#include <cstdio>
#include <iostream>

using namespace std;

int read() {
  int out = 0;
```

```
  char c;
  while (!isdigit(c = getchar()))
    ;
  for (; isdigit(c); c = getchar()) out = out * 10 + c - '0';
  return out;
}

const int N = 100005;
const int M = 1000005;
const int V = 7005;

bitset<V> pre[V], pre2[V], a[N], mu;
int n, m, tot;
char ans[M];

int main() {
  int i, j, x, y, z;

  n = read();
  m = read();

  mu.set();
  for (i = 2; i * i < V; ++i) {
    for (j = 1; i * i * j < V; ++j) {
      mu[i * i * j] = 0;
    }
  }
  for (i = 1; i < V; ++i) {
    for (j = 1; i * j < V; ++j) {
      pre[i * j][i] = 1;
      pre2[i][i * j] = mu[j];
    }
  }

  while (m--) {
    switch (read()) {
      case 1:
        x = read();
        y = read();
        a[x] = pre[y];
        break;
      case 2:
        x = read();
        y = read();
        z = read();
        a[x] = a[y] ^ a[z];
        break;
      case 3:
        x = read();
        y = read();
        z = read();
        a[x] = a[y] & a[z];
        break;
      case 4:
```

```
      x = read();
      y = read();
      ans[tot++] = ((a[x] & pre2[y]).count() & 1) + '0';
      break;
    }
  }

  printf("%s", ans);

  return 0;
}
```

与埃氏筛结合

由于 bitset 快速的连续读写效率，使得它非常适合用于与 埃氏筛 结合打质数表。

使用的方式也很简单，只需要将埃氏筛中的布尔数组替换成 bitset 即可。

" 速度测试"

以下代码均使用 g++-4.8 code.cpp -O2 命令行编译，CPU 使用 Intel i5-8259U 进行测试。测试结果取十次平均值。

算法	5×10^7	10^8	5×10^8
埃氏筛 + 布尔数组	386ms	773ms	4.41s
欧拉筛 + 布尔数组	257ms	521ms	2.70s
埃氏筛 +bitset	219ms	492ms	2.66s
欧拉筛 +bitset	332ms	661ms	3.21s

从测试结果中可知，时间复杂度 $O(n \log \log n)$ 的埃氏筛在使用 bitset 优化后速度甚至超过时间复杂度 $O(n)$ 的欧拉筛，而欧拉筛在使用 bitset 后会出现「负优化」的情况。

" 参考代码"

```
bitset<N> vis;

void Prime(int n) {
  vis.set();
  vis[0] = vis[1] = 0;
  for (int i = 2; i * i <= n; i++) {
    if (vis[i]) {
      for (int j = i << 1; j <= n; j += i) vis[j] = 0;
    }
  }
}
```

与树分块结合

bitset 与树分块结合可以解决一类求树上多条路径信息并的问题，详见 数据结构 / 树分块。

与莫队结合

详见 杂项 / 莫队配合 bitset。

计算高维偏序

详见 FHR 课件[6]。

参考资料与注释

[1] 「LibreOJ Round #2」贪心只能过样例

[2] std::bitset

[3] 加了几个剪枝的暴力

[4] 手写 bitset

[5] CF1097F Alex and a TV Show

[6] FHR 课件

4.3.5 string

Authors: johnvp22, Ir1d

string 是什么

`std::string` 是在标准库 `<string>`（注意不是 C 语言中的 `<string.h>` 库）中提供的一个类，本质上是 `std::basic_string<char>` 的别称。

为什么要使用 string

在 C 语言中，提供了字符串的操作，但只能通过字符数组的方式来实现字符串。而 `string` 则是一个简单的类，使用简单，在 OI 竞赛中被广泛使用。并且相较于其他 STL 容器，`string` 的常数可以算是非常优秀的，基本与字符数组不相上下。

string 可以动态分配空间

和许多 STL 容器相同，`string` 能动态分配空间，这使得我们可以直接使用 `std::cin` 来输入，但其速度则同样较慢。这一点也同样让我们不必为内存而烦恼。

string 重载了加法运算符和比较运算符

`string` 的加法运算符可以直接拼接两个字符串或一个字符串和一个字符。和 `std::vector` 类似，`string` 重载了比较运算符，同样是按字典序比较的，所以我们可以直接调用 `std::sort` 对若干字符串进行排序。

使用方法

下面介绍 `string` 的基本操作，具体可看 C++ 文档[1]。

声明

```
std::string s;
```

转 char 数组

在 C 语言里，也有很多字符串的函数，但是它们的参数都是 char 指针类型的，为了方便使用，`string` 有两个成员函数能够将自己转换为 char 指针——`data()`/`c_str()`（它们几乎是一样的，但最好使用 `c_str()`，因为 `c_str()` 保证末尾有空字符，而 `data()` 则不保证），如：

```
printf("%s", s);          // 编译错误
printf("%s", s.data());   // 编译通过，但是是 undefined behavior
printf("%s", s.c_str());  // 一定能够正确输出
```

获取长度

很多函数都可以返回 string 的长度：

```
printf("s 的长度为 %lu", s.size());
printf("s 的长度为 %lu", s.length());
printf("s 的长度为 %lu", strlen(s.c_str()));
```

" 这些函数的复杂度"

`strlen()` 的复杂度一定是与字符串长度线性相关的。

`size()` 和 `length()` 的复杂度在 C++98 中没有指定，在 C++11 中被指定为常数复杂度。但在常见的编译器上，即便是 C++98，这两个函数的复杂度也是常数。

warning

这三个函数（以及下面将要提到的 `find` 函数）的返回值类型都是 `size_t`（`unsigned long`）。因此，这些返回值不支持直接与负数比较或运算，建议在需要时进行强制转换。

寻找某字符（串）第一次出现的位置

`find(str,pos)` 函数可以用来查找字符串中一个字符/字符串在 `pos`（含）之后第一次出现的位置（若不传参给 `pos` 则默认为 `0`）。如果没有出现，则返回 `string::npos`（被定义为 `-1`，但类型仍为 `size_t/unsigned long`）。

示例：

```
string s = "OI Wiki", t = "OI", u = "i";
int pos = 5;
printf("字符 I 在 s 的 %lu 位置第一次出现\n", s.find('I'));
printf("字符 a 在 s 的 %lu 位置第一次出现\n", s.find('a'));
printf("字符 a 在 s 的 %d 位置第一次出现\n", s.find('a'));
printf("字符串 t 在 s 的 %lu 位置第一次出现\n", s.find(t));
printf("在 s 中自 pos 位置起字符串 u 第一次出现在 %lu 位置", s.find(u, pos));
```

输出：

```
字符 I 在 s 的 1 位置第一次出现
字符 a 在 s 的 18446744073709551615 位置第一次出现 // 即为 size_t(-1)，具体数值与
平台有关。
字符 a 在 s 的 -1 位置第一次出现 // 强制转换为 int 类型则正常输出 -1
字符串 t 在 s 的 0 位置第一次出现
在 s 中自 pos 位置起字符串 u 第一次出现在 6 位置
```

截取子串

substr(pos, len) 函数的参数返回从 pos 位置开始截取最多 len 个字符组成的字符串（如果从 pos 开始的后缀长度不足 len 则截取这个后缀）。

示例：

```
string s = "OI Wiki", t = "OI";
printf(" 从字符串 s 的第四位开始的最多三个字符构成的子串是 %s\n",
       s.substr(3, 3).c_str());
printf(" 从字符串 t 的第二位开始的最多三个字符构成的子串是 %s",
       t.substr(1, 3).c_str());
```

输出：

```
从字符串 s 的第二位开始的最多三个字符构成的子串是 Wik
从字符串 t 的第二位开始的最多三个字符构成的子串是 I
```

插入/删除字符（串）

insert(index,count,ch) 和 insert(index,str) 是比较常见的插入函数。它们分别表示在 index 处连续插入 count 次字符串 ch 和插入字符串 str。

erase(index,count) 函数将字符串 index 位置开始（含）的 count 个字符删除（若不传参给 count 则表示删去 index 位置及以后的所有字符）。

示例：

```
string s = "OI Wiki", t = " Wiki";
char u = '!';
s.erase(2);
printf(" 从字符串 s 的第三位开始删去所有字符后得到的字符串是 %s\n", s.c_str());
s.insert(2, t);
printf(" 在字符串 s 的第三位处插入字符串 t 后得到的字符串是 %s\n", s.c_str());
s.insert(7, 3, u);
printf(" 在字符串 s 的第八位处连续插入 3 次字符串 u 后得到的字符串是 %s",
       s.c_str());
```

输出：

```
从字符串 s 的第三位开始删去所有字符后得到的字符串是 OI
在字符串 s 的第三位处插入字符串 t 后得到的字符串是 OI Wiki
在字符串 s 的第八位处连续插入 3 次字符串 u 后得到的字符串是 OI Wiki!!!
```

替换字符（串）

replace(pos,count,str) 和 replace(first,last,str) 是比较常见的替换函数。它们分别表示将从 pos 位置开始 count 个字符的子串替换为 str 以及将以 first 开始（含）、last 结束（不含）的子串替换为 str，其中 first 和 last 均为迭代器。

示例：

```
string s = "OI Wiki";
s.replace(2, 5, "");
printf(" 将字符串 s 的第 3~7 位替换为空串后得到的字符串是 %s\n", s.c_str());
s.replace(s.begin(), s.begin() + 2, "NOI");
printf(" 将字符串 s 的前两位替换为 NOI 后得到的字符串是 %s", s.c_str());
```

输出：

```
将字符串 s 的第 3~7 位替换为空串后得到的字符串是 OI
将字符串 s 的前两位替换为 NOI 后得到的字符串是 NOI
```

参考资料与注释

[1] C++ 文档

4.3.6 pair

Authors: sbofgayschool

std::pair 是标准库中定义的一个类模板。用于将两个变量关联在一起，组成一个「对」，而且两个变量的数据类型可以是不同的。

类模板

类模板（class template）本身不是一个类，而是可以根据**不同数据类型**产生**不同类**的「模板」。

在使用时，编译器会根据传入的数据类型产生对应的类，再创建对应实例。

模板属于 C++ 较为高级的语言特性，在信息学竞赛中几乎不会出现。如果对此感兴趣，可以进一步阅读《C++ Primer》以学习更深层次的 C++ 知识。

通过灵活使用 pair，可以轻松应对**需要将关联数据捆绑存储、处理**的场景。

pair 与自定义 struct

与自定义的 struct 相比，pair 不需要额外定义结构与重载运算符，因此使用起来更加简便。

然而，自定义 struct 的变量命名往往更加清晰（pair 只能使用 first 与 second 访问包含的两个变量）。同时，如果需要将两个以上的变量进行关联，自定义 struct 会更加合适。

使用

初始化

可以在定义时直接完成 pair 的初始化。

```cpp
pair<int, double> p0(1, 2.0);
```

也可以使用先定义，后赋值的方法完成 pair 的初始化。

```cpp
pair<int, double> p1;
p1.first = 1;
p1.second = 2.0;
```

还可以使用 std::make_pair 函数。该函数接受两个变量，并返回由这两个变量组成的 pair。

```cpp
pair<int, double> p2 = make_pair(1, 2.0);
```

一种常用的方法是使用宏定义 #define mp make_pair，将有些冗长的 make_pair 化简为 mp。

在 C++11 以及之后的版本中，make_pair 可以配合 auto 使用，以避免显式声明数据类型。

```cpp
auto p3 = make_pair(1, 2.0);
```

关于 auto 的在信息学竞赛中的使用，参见 迭代器 部分的说明。

访问

通过成员函数 first 与 second，可以访问 pair 中包含的两个变量。

```
int i = p0.first;
double d = p0.second;
```

也可以对其进行修改。

```
p1.first++;
```

比较

pair 已经预先定义了所有的比较运算符，包括 <、>、<=、>=、==、!=。当然，这需要组成 pair 的两个变量所属的数据类型定义了 == 和/或 < 运算符。

其中，<、>、<=、>= 四个运算符会先比较两个 pair 中的第一个变量，在第一个变量相等的情况下再比较第二个变量。

```
if (p2 >= p3) {
  cout << "do something here" << endl;
}
```

由于 pair 定义了 STL 中常用的 < 与 ==，使得其能够很好的与其他 STL 函数或数据结构配合。比如，pair 可以作为 priority_queue 的数据类型。

```
priority_queue<pair<int, double> > q;
```

赋值与交换

可以将 pair 的值赋给另一个类型一致的 pair。

```
p0 = p1;
```

也可以使用 swap 函数交换 pair 的值。

```
swap(p0, p1);
p2.swap(p3);
```

应用举例

离散化

pair 可以轻松实现离散化。

我们可以创建一个 pair 数组，将原始数据的值作为每个 pair 第一个变量，将原始数据的位置作为第二个变量。在排序后，将原始数据值的排名（该值排序后所在的位置）赋给该值原本所在的位置即可。

```
// a 为原始数据
pair<int, int> a[MAXN];
// ai 为离散化后的数据
int ai[MAXN];
for (int i = 0; i < n; i++) {
  // first 为原始数据的值，second 为原始数据的位置
  scanf("%d", &a[i].first);
  a[i].second = i;
}
// 排序
```

```
sort(a, a + n);
for (int i = 0; i < n; i++) {
  // 将该值的排名赋给该值原本所在的位置
  ai[a[i].second] = i;
}
```

Dijkstra

如前所述，pair 可以作为 priority_queue 的数据类型。

那么，在 Dijkstra 算法的堆优化中，可以使用 pair 与 priority_queue 维护节点，将节点当前到起点的距离作为第一个变量，将节点编号作为第二个变量。

```
priority_queue<pair<int, int>, std::vector<pair<int, int> >,
               std::greater<pair<int, int> > >
    q;
... while (!q.empty()) {
  // dis 为入堆时节点到起点的距离，i 为节点编号
  int dis = q.top().first, i = q.top().second;
  q.pop();
  ...
}
```

pair 与 map

map 的是 C++ 中存储键值对的数据结构。很多情况下，map 中存储的键值对通过 pair 向外暴露。

```
map<int, double> m;
m.insert(make_pair(1, 2.0));
```

关于 map 更多的内容，请见 关联式容器 与 无序关联式容器 中相关部分。

4.4 C++ 进阶

4.4.1 类

> **Authors:** Ir1d, cjsoft, Lans1ot, JasonkayZK 类（class）是结构体的拓展，不仅能够拥有成员元素，还拥有成员函数。

在面向对象编程（OOP）中，对象就是类的实例，也就是变量。

C++ 中 struct 关键字定义的也是类，上文中的**结构体**的定义来自 C。因为某些历史原因，C++ 保留并拓展了 struct。

定义类

类使用关键字 class 或者 struct 定义，下文以 class 举例。

```
class ClassName {
  ...
};

// Example:
class Object {
 public:
  int weight;
```

```
  int value;
} e[array_length];

const Object a;
Object b, B[array_length];
Object *c;
```

与使用 struct 大同小异。该例定义了一个名为 Object 的类。该类拥有两个成员元素，分别为 weight,value；并在 } 后使用该类型定义了一个数组 e。

定义类的指针形同 struct。

访问说明符

不同于 struct 中的举例，本例中出现了 public，这属于访问说明符。

- public：该访问说明符之后的各个成员都可以被公开访问，简单来说就是无论**类内**还是**类外**都可以访问。
- protected：该访问说明符之后的各个成员可以被**类内**、派生类或者友元的成员访问，但类外**不能访问**。
- private：该访问说明符之后的各个成员**只能**被**类内**成员或者友元的成员访问，**不能**被从类外或者派生类中访问。

对于 struct，它的所有成员都是默认 public。对于 class，它的所有成员都是默认 private。

关于"友元"和"派生类"，可以参考下方折叠框，或者查询网络资料进行详细了解。

对于算法竞赛来说，友元和派生类并不是必须要掌握的知识点。

" 关于友元以及派生类的基本概念"

友元（friend）：使用 friend 关键字修饰某个函数或者类。可以使得在**被修饰者**在不成为成员函数或者成员类的情况下，访问该类的私有（private）或者受保护（protected）成员。简单来说就是只要带有这个类的 friend 标记，就可以访问私有或受保护的成员元素。

派生类（derived class）：C++ 允许使用一个类作为**基类**，并通过基类**派生**出**派生类**。其中派生类（根据特定规则）继承基类中的成员变量和成员函数。可以提高代码的复用率。

派生类似"is" 的关系。如猫（派生类）"is" 哺乳动物（基类）。

对于上面 private 和 protected 的区别，可以看做派生类可以访问基类的 protected 的元素（public 同），但不能访问 private 元素。

访问与修改成员元素的值

方法形同 struct

- 对于变量，使用 . 符号。
- 对于指针，使用 -> 符号。

成员函数

成员函数，顾名思义。就是类中所包含的函数。

" 常见成员函数举例"

```
vector.push_back();
set.insert();
queue.empty();
```

```
class Class_Name {
  ... type Function_Name(...) { ... }
};

// Example:
class Object {
 public:
  int weight;
  int value;

  void print() {
    cout << weight << endl;
    return;
  }

  void change_w(int);
};

void Object::change_w(int _weight) { weight = _weight; }

Object var;
```

该类有一个打印 `Object` 成员元素的函数，以及更改成员元素 `weight` 的函数。

和函数类似，对于成员函数，也可以先声明，在定义，如第十四行（声明处）以及十七行后（定义处）。

如果想要调用 `var` 的 `print` 成员函数，可以使用 `var.print()` 进行调用。

重载运算符

" 何为重载"

C++ 允许编写者为名称相同的函数或者运算符指定不同的定义。这称为**重载**（overload）。

如果同名函数的参数种类、数量中的一者或多者两两不相同，则这些同名函数被看做是不同的。

需要注意的是：如果两个同名函数的区别仅仅是返回值的类型不同则无法进行重载，此时编译器会拒绝编译！

如果在调用时不会出现混淆（指调用某些同名函数时，无法根据所填参数种类和数量唯一地判断出被调用函数。常发生在具有默认参数的函数中），则编译器会根据调用时所填参数判断应调用函数。

而上述过程被称作重载解析。

重载运算符，可以部分程度上代替函数，简化代码。

下面给出重载运算符的例子。

```
class Vector {
 public:
  int x, y;

  Vector() : x(0), y(0) {}

  Vector(int _x, int _y) : x(_x), y(_y) {}

  int operator*(const Vector& other) { return x * other.x + y * other.y; }

  Vector operator+(const Vector&);
  Vector operator-(const Vector&);
};
```

```
Vector Vector::operator+(const Vector& other) {
  return Vector(x + other.x, y + other.y);
}

Vector Vector::operator-(const Vector& other) {
  return Vector(x - other.x, y - other.y);
}

// 关于 4,5 行表示为 x,y 赋值，具体实现参见后文。
```

该例定义了一个向量类，并重载了 `*` `+` `-` 运算符，并分别代表向量内积，向量加，向量减。

重载运算符的模板大致可分为下面几部分。

```
/* 类定义内重载 */ 返回类型 operator 符号 (参数){...}

/* 类定义内声明，在外部定义 */ 返回类型类名称::operator 符号 (参数){...}
```

对于自定义的类，如果重载了某些运算符（一般来说只需要重载 `<` 这个比较运算符），便可以使用相应的 STL 容器或算法，如 `sort`。

如要了解更多，可参见「参考资料」第四条。

"可以被重载的运算符"

```
+       -       *       /       %       ^       &
|       ~       !       =       <       >       +=
-=      *=      /=      %=      ^=      &=      |=
<<      >>      >>=     <<=     ==      !=      <=
>=      &&      ||      ++      --      ,       ->*
->      ()      []      new     new []  delete  delete []
```

在实例化变量时设定初始值

为完成这种操作，需要定义**默认构造函数** (Default constructor)。

```
class ClassName {
  ... ClassName(...)... { ... }
};

// Example:
class Object {
 public:
  int weight;
  int value;

  Object() {
    weight = 0;
    value = 0;
  }
};
```

该例定义了 `Object` 的默认构造函数，该函数能够在我们实例化 `Object` 类型变量时，将所有的成员元素初始化为 `0`。

若无显式的构造函数，则编译器认为该类有隐式的默认构造函数。换言之，若无定义任何构造函数，则编译器会自动生成一个默认构造函数，并会根据成员元素的类型进行初始化（与定义内置类型变量相同）。

在这种情况下，成员元素都是未初始化的，访问未初始化的变量的结果是未定义的（也就是说并不知道会返回和值）。

如果需要自定义初始化的值，可以再定义（或重载）构造函数。

" 关于定义（或重载）构造函数"

一般来说，默认构造函数是不带参数的，这区别于构造函数。构造函数和默认构造函数的定义大同小异，只是参数数量上的不同。

构造函数可以被重载（当然首次被叫做定义）。需要注意的是，如果已经定义了构造函数，那么编译器便不会再生成无参数的默认构造函数。这会可能会使试图以默认方法构造变量的行为编译失败（指不填入初始化参数）。

使用 C++11 或以上时，可以使用 `{}` 进行变量的初始化。

" 关于 {}"

使用 `{}` 进行初始化，会用到 std::initializer_list 这一个轻量代理对象进行初始化。

初始化步骤大概如下

1. 尝试寻找参数中有 `std::initializer_list` 的默认构造函数，如果有则调用（调用完后不再进行下面的查找，下同）。
2. 尝试将 `{}` 中的元素填入其他构造参数，如果能将参数按照顺序填满（默认参数也算在内），则调用该默认构造函数。
3. 若无 `private` 成员元素，则尝试在**类外**按照元素定义顺序或者下标顺序依次赋值。

上述过程只是完整过程的简化版本，详细内容参见"参考资料九"

```
class Object {
 public:
  int weight;
  int value;

  Object() {
    weight = 0;
    value = 0;
  }

  Object(int _weight = 0, int _value = 0) {
    weight = _weight;
    value = _value;
  }

  // the same as
  // Object(int _weight,int _value):weight(_weight),value(_value) {}
};

// the same as
// Object::Object(int _weight,int _value){
//   weight = _weight;
//   value = _value;
// }
//}

Object A;         // ok
Object B(1, 2);   // ok
```

```
Object C{1, 2};  // ok,(C++11)
```

" 关于隐式类型转换"

有时候会写出如下的代码

```
class Node {
 public:
  int var;

  Node(int _var) : var(_var) {}
};

Node a = 1;
```

看上去十分不符合逻辑，一个 `int` 类型不可能转化为 `node` 类型。但是编译器不会进行 `error` 提示。

原因是在进行赋值时，首先会将 `1` 作为参数调用 `node::node(int)`，然后调用默认的复制函数进行赋值。

但大多数情况下，编写者会希望编译器进行报错。这时便可以在构造函数前追加 `explicit` 关键字。这会告诉编译器必须显式进行调用。

```
class Node {
 public:
  int var;

  explicit Node(int _var) : var(_var) {}
};
```

也就是说 `node a=1` 将会报错，但 `node a=node(1)` 不会。因为后者显式调用了构造函数。当然大多数人不会写出后者的代码，但此例足以说明 explicit 的作用。

不过在算法竞赛中，为了避免此类情况常用的是"加强对代码的规范程度"，从源头上避免

销毁

这是不可避免的问题。每一个变量都将在作用范围结束走向销毁。

但对于已经指向了动态申请的内存的指针来说，该指针在销毁时不会自动释放所指向的内存，需要手动释放动态内存。

如果结构体的成员元素包含指针，同样会遇到这种问题。需要用到析构函数来手动释放动态内存。

析构函数（Destructor）将会在该变量被销毁时被调用。重载的方法形同构造函数，但需要在前加 `~`

默认定义的析构函数通常对于算法竞赛已经足够使用，通常我们只有在成员元素包含指针时才会重载析构函数。

```
class Object {
 public:
  int weight;
  int value;
  int* ned;

  Object() {
    weight = 0;
    value = 0;
  }

  ~Object() { delete ned; }
};
```

为类变量赋值

默认情况下，赋值时会按照对应成员元素赋值的规则进行。也可以使用类名称 () 或类名称 {} 作为临时变量来进行赋值。

前者只是调用了复制构造函数（copy constructor），而后者在调用复制构造函数前会调用默认构造函数。

另外默认情况下，进行的赋值都是对应元素间进行**浅拷贝**，如果成员元素中有指针，则在赋值完成后，两个变量的成员指针具有相同的地址。

```
// A,tmp1,tmp2,tmp3 类型为 Object
tmp1 = A;
tmp2 = Object(...);
tmp3 = {...};
```

如需解决指针问题或更多操作，需要重载相应的构造函数。

更多构造函数（constructor）内容，参见「参考资料」第六条。

参考资料

1. cppreference class[1]
2. cppreference access[2]
3. cppreference default_constructor[3]
4. cppreference operator[4]
5. cplusplus Data structures[5]
6. cplusplus Special members[6]
7. C++11 FAQ[7]
8. cppreference Friendship and inheritance[8]
9. cppreference value initialization[9]

参考资料与注释

[1] cppreference class

[2] cppreference access

[3] cppreference default_constructor

[4] cppreference operator

[5] cplusplus Data structures

[6] cplusplus Special members

[7] C++11 FAQ

[8] cppreference Friendship and inheritance

[9] cppreference value initialization

4.4.2 命名空间

概述

C++ 的**命名空间**机制可以用来解决复杂项目中名字冲突的问题。

举个例子：C++ 标准库的所有内容均定义在 std 命名空间中，如果你定义了一个叫 cin 的变量，则可以通过 cin 来访问你定义的 cin 变量，通过 std::cin 访问标准库的 cin 对象，而不用担心产生冲突。

声明

下面的代码声明了一个名字叫 A 的命名空间：

```
namespace A {
int cnt;

void f(int x) { cnt = x; }
}  // namespace A
```

声明之后，在这个命名空间外部，你可以通过 A::f(x) 来访问命名空间 A 内部的 f 函数。

命名空间的声明是可以嵌套的，因此下面这段代码也是允许的：

```
namespace A {
namespace B {
void f() { ... }
}  // namespace B

void f() {
  B::f();  // 实际访问的是 A::B::f()，由于当前位于命名空间 A
           // 内，所以可以省略前面的 A::
}
}  // namespace A

void f()  // 这里定义的是全局命名空间的 f 函数，与 A::f 和 A::B::f
          // 都不会产生冲突
{
  A::f();
  A::B::f();
}
```

using 指令

声明了命名空间之后，如果在命名空间外部访问命名空间内部的成员，需要在成员名前面加上命名空间 ::。

有没有什么比较方便的方法能让我们直接通过成员名访问命名空间内的成员呢？答案是肯定的。我们可以使用 using 指令。

using 指令有如下两种形式：

1. using 命名空间 :: 成员名 ;：这条指令可以让我们省略某个成员名前的命名空间，直接通过成员名访问成员，相当于将这个成员导入了当前的作用域。
2. using namespace 命名空间 ;：这条指令可以直接通过成员名访问命名空间中的**任何**成员，相当于将这个命名空间的所有成员导入了当前的作用域。

因此，如果执行了 using namespace std;，就会将 std 中的所有名字引入到全局命名空间当中。这样，我们就可以用 cin 代替 std::cin，用 cout 代替 std::cout。

"using 指令可能会导致命名冲突！"

由于 `using namespace std;` 会将 `std` 中的**所有名字**引入，因此如果声明了与 `std` 重名的变量或函数，就可能会因为命名冲突而导致编译错误。

因此在工程中，并不推荐使用 `using namespace 命名空间;` 的指令。

有了 `using` 指令，C++ 语法基础 中的代码可以有这两种等价写法：

```cpp
#include <iostream>

using std::cin;
using std::cout;
using std::endl;

int main() {
  int x, y;
  cin >> x >> y;
  cout << y << endl << x;
  return 0;
}
```

```cpp
#include <iostream>

using namespace std;

int main() {
  int x, y;
  cin >> x >> y;
  cout << y << endl << x;
  return 0;
}
```

应用

在一些具有多个子任务的问题中，我们可以对每个子任务各开一个命名空间，在其中定义我们解决该子任务所需要的变量与函数，这样各个子任务间互不干扰，会在一定程度上方便调试，也会改善程序的可读性。

4.4.3 值类别

注意：这部分的内容很可能对算法竞赛无用，但如果你希望更深入地理解 C++，写出更高效的代码，那么本文的内容也许会对你有所帮助。

每个 C++ 表达式都有两个属性：类型 (type) 和值类别 (value category)。前者是大家都熟悉的，但作为算法竞赛选手，很可能完全不知道后者是什么。不管你在不在意，值类别是 C++ 中非常重要的一个概念。

"关于名词的翻译"

type 和 category 都可以翻译为「类型」或「类别」，但为了区分两者，下文中统一将 type 翻译为「类型」，category 翻译为「类别」。

从 CPL 语言的定义说起

左值与右值的概念最早出现在 C 语言的祖先语言：CPL。

在 CPL 的定义中，lvalue 意为 left-hand side value，即能够出现在赋值运算符（等号）左侧的值，右值的定义亦然。

C 和 C++11 以前

C 语言沿用了相似的分类方法，但左右值的判断标准已经与赋值运算符无关。在新的定义中，lvalue 意为 locate value，即能进行取地址运算 (&) 的值。

可以这么理解：左值是有内存地址的对象，而右值只是一个中间计算结果（虽然编译器往往需要在内存中分配地址来储存这个值，但这个内存地址是无法被程序员感知的，所以可以认为它不存在）。中间计算结果就意味着这个值马上就没用了，以后不会再访问它。

比如在 `int a = 0;` 这段代码中，`a` 就是一个左值，而 `0` 是一个右值。

" 常见的关于左右值的误解"

以下几种类型是经常被误认为右值的左值：

- **字符串字面量**：由于 C++ 兼容 C 风格的字符串，需要能对一个字符串字面量取地址（即头指针）来传参。但是其他的字面量，包括自定义字面量，都是右值。
- **数组**：数组名就是数组首个元素的指针这种说法似乎误导了很多人，但这个说法显然是错误的，对数组进行取地址是可以编译的。数组名可以隐式的退化成首个元素的指针，这才是右值。

C++11 开始

从 C++11 开始，为了配合移动语义，值的类别就不是左值右值这么简单了。

考虑一个简单的场景：

```
std::vector<int> a{...};
std::vector<int> b;
b = a;
```

我们知道第三行的赋值运算复杂度是正比于 `a` 的长度的，复制的开销很大。但有些情况下，比如 `a` 在以后的代码中不会再使用，那么我们完全可以把 `a` 所持有的内存「转移」到 `b` 上，这就是移动语义干的事情。

我们姑且不管移动是怎么实现的，先来考虑一下我们如何标记 `a` 是可以移动的。显然不管能否移动，这个表达式的类型都是 `vector` 不变，所以只能对值类别下手。不可移动的 `a` 是左值，如果要在原有的体系下标记可以移动的 `a`，我们只能把它标记为右值。但标记为右值又是不合理的，因为这个 `a` 实际上拥有自己的内存地址，与其他右值有有根本上的不同。所以 C++11 引入了亡值 (xvalue) 这一值类别来标记这一种表达式。

于是我们现在有了三种类别：左值 (lvalue)、纯右值 (prvalue)、亡值 (xvalue)（纯右值就是原先的右值）。

然后我们发现亡值同时具有一些左值和纯右值的性质，比如它可以像左值一样取地址，又像右值一样不会再被访问。

所以又有了两种组合类别：泛左值 (glvalue)（左值和亡值）、右值 (rvalue)（纯右值和亡值）。

有一个初步的感性理解后，来看一下标准委员会对它们的定义：

- A **glvalue**(generalized lvalue) is an expression whose evaluation determines the identity of an object, bit-field, or function.
- A **prvalue**(pure rvalue) is an expression whose evaluation initializes an object or a bit-field, or computes the value of an operand of an operator, as specified by the context in which it appears, or an expression that has type cv void.
- An **xvalue**(eXpiring value) is a glvalue that denotes an object or bit-field whose resources can be reused (usually because it is near the end of its lifetime)。
- An **lvalue** is a glvalue that is not an xvalue.
- An **rvalue** is a prvalue or an xvalue.

上述定义中提到了一个叫位域 (bit-field) 的东西。如果你不知道位域是什么，忽略它即可，后文也不会提及。

其中关键的两个概念：

- 是否拥有身份 (identity)：可以确定表达式是否与另一表达式指代同一实体，例如比较它们所标识的对象或函数的（直接或间接获得的）地址
- 是否可以被移动 (resources can be reused)：对象的资源可以移动到别的对象中

这 5 种类型无非就是根据上面两种属性的是与否区分的，所以用下面的这张表格可以帮助理解：

	拥有身份（glvalue）	不拥有身份
可移动（rvalue）	xvalue	prvalue
不可移动	lvalue	不存在

注意不拥有身份就意味着这个对象以后无法被访问，这样的对象显然是可以被移动的，所以不存在不拥有身份不可移动的值。

C++17 带来的新变化

从拷贝到移动提升了不少速度，那么我们是否能够优化的更彻底一点，把移动的开销都省去呢？

考虑这样的代码：

```
std::vector<int> make_vector(...) {
  std::vector<int> result;
  // ...
  return result;
}

std::vector<int> a = make_vector(...);
```

`make_vector` 函数根据一输入生成一个 `vector`。这个 `vector` 一开始在 `make_vector` 的栈上被构造，随后又被移动到调用者的栈上，需要一次移动操作，这显然很浪费，能不能省略这次移动？

答案是肯定的，这就是 RVO 优化，即省略拷贝。通常的方法是编译器让 `make_vector` 返回的对象直接在调用者的栈上构造，然后 `make_vector` 在上面进行修改。这相当与这样的代码：

```
void make_vector(std::vector<int>& result, ...) {
  // ... (对 result 进行操作)
}

std::vecctor<int> a;
make_vector(a, ...);
```

在 C++17 以前，尽管标准未做出规定，但主流编译器都实现了这种优化。在 C++17 以后，这种优化成为标准的硬性规定。

回到和移动语义刚被提出时的问题，如何确定一个移动赋值是可以省略的？再引入一种新的值类别？

不，C++11 的值类别已经够复杂了。我们意识到在 C++11 的标准下，亡值和纯右值都是可以移动的，那么就可以在这两种类别上做文章。

C++17 以后，纯右值不再能移动，但可以隐式地转变为亡值。对于纯右值用于初始化的情况下，可以省略拷贝，而其他不能省略的情况下，隐式转换为亡值进行移动。

所以在 C++17 之后的值类别，被更为整齐的划分为泛左值与纯右值两大块，右值存在的意义被削弱。这样的改变某种程度上简化了整个值类别体系。

参考文献与推荐阅读

1. Value categories[1]
2. Wording for guaranteed copy elision through simplified value categories[2]
3. C++ 中的值类别[3]

参考资料与注释

[1] Value categories

[2] Wording for guaranteed copy elision through simplified value categories

[3] C++ 中的值类别

4.4.4 重载运算符

重载运算符是通过对运算符的重新定义，使得其支持特定数据类型的运算操作。重载运算符是重载函数的特殊情况。

C++ 自带的运算符，最初只定义了一些基本类型的运算规则。当我们要在用户自定义的数据类型上使用这些运算符时，就需要定义运算符在这些特定类型上的运算方式。

限制

重载运算符存在如下限制：

- 只能对现有的运算符进行重载，不能自行定义新的运算符。
- 以下运算符不能被重载：`::`（作用域解析），`.`（成员访问），`.*`（通过成员指针的成员访问），`?:`（三目运算符）。
- 重载后的运算符，其运算优先级，运算操作数，结合方向不得改变。
- 对 `&&`（逻辑与）和 `||`（逻辑或）的重载失去短路求值。

实现

重载运算符分为两种情况，重载为成员函数或非成员函数。

当重载为成员函数时，因为隐含一个指向当前成员的 `this` 指针作为参数，此时函数的参数个数与运算操作数相比少一个。

而当重载为非成员函数时，函数的参数个数与运算操作数相同。

下面将给出几个重载运算符的示例。

函数调用运算符

函数调用运算符 `()` 只能重载为成员函数。通过对一个类重载 `()` 运算符，可以使该类的对象能像函数一样调用。

重载 `()` 运算符的一个常见应用是，将重载了 `()` 运算符的结构体作为自定义比较函数传入优先队列等 STL 容器中。

下面就是一个例子：给出 n 个学生的姓名和分数，按分数降序排序，分数相同者按姓名字典序升序排序，输出排名最靠前的人的姓名和分数。

```
#include <iostream>
#include <queue>
using namespace std;
```

```cpp
struct student {
  string name;
  int score;
};

struct cmp {
  bool operator()(const student& a, const student& b) const {
    return a.score < b.score || (a.score == b.score && a.name > b.name);
  }
};

priority_queue<student, vector<student>, cmp> pq;

int main() {
  int n;
  cin >> n;
  for (int i = 1; i <= n; i++) {
    string name;
    int score;
    cin >> name >> score;
    pq.push({name, score});
  }
  student rk1 = pq.top();
  cout << rk1.name << ' ' << rk1.score << endl;
  return 0;
}
```

自增自减运算符

自增自减运算符分为两类，前置和后置。为了能将两类运算符区别开来，对于后置自增自减运算符，重载的时候需要添加一个类型为 `int` 的空置形参。

另外一点是，内置的自增自减运算符中，前置的运算符返回的是引用，而后置的运算符返回的是值。虽然重载后的运算符不必遵循这一限制，不过在语义上，仍然期望重载的运算符与内置的运算符在返回值的类型上保持一致。

因此，对于类型 T，典型的重载自增运算符的定义如下：

重载定义（以 ++ 为例）	成员函数	非成员函数
前置	`T& T::operator++();`	`T& operator++(T& a);`
后置	`T T::operator++(int);`	`T operator++(T& a, int);`

比较运算符

在 `std::sort` 和一些 STL 容器中，需要用到 `<` 运算符。在使用自定义类型时，我们需要手动重载。

还是以讲函数调用运算符时举的例子说起，如果我们重载比较运算符，实现代码是这样的（主函数因为没有改动就略去了）：

```cpp
struct student {
  string name;
  int score;

  bool operator<(const student& a) const {
    return score < a.score || (score == a.score && name > a.name);
```

```
    // 上面省略了 this 指针，完整表达式如下：
    // this->score<a.score||(this->score==a.score&&this->name>a.name);
  }
};

priority_queue<student> pq;
```

上面的代码将小于号重载为了成员函数，当然重载为非成员函数也是可以的。

```
struct student {
  string name;
  int score;
};

bool operator<(const student& a, const student& b) {
  return a.score < b.score || (a.score == b.score && a.name > b.name);
}

priority_queue<student> pq;
```

事实上，只要有了 `<` 运算符，则其他五个比较运算符的重载也可以很容易实现。

```
/* clang-format off */

// 下面的几种实现均将小于号重载为非成员函数

bool operator<(const T& lhs, const T& rhs) { /* 这里重载小于运算符 */ }
bool operator>(const T& lhs, const T& rhs) { return rhs < lhs; }
bool operator<=(const T& lhs, const T& rhs) { return !(lhs > rhs); }
bool operator>=(const T& lhs, const T& rhs) { return !(lhs < rhs); }
bool operator==(const T& lhs, const T& rhs) { return !(lhs < rhs) && !(lhs > rhs
); }
bool operator!=(const T& lhs, const T& rhs) { return !(lhs == rhs); }
```

参考资料与注释：

- 运算符重载 - cppreference[1]

参考资料与注释

[1] 运算符重载 - cppreference

4.4.5 引用

引用可以看成是 C++ 封装的指针，用来传递它所指向的对象。在 C++ 代码中实际上会经常和引用打交道，但是通常不会显式地表现出来。引用的基本原则是在声明时必须指向对象，以及对引用的一切操作都相当于对原对象操作。另外，引用不是对象，因此不存在引用的数组、无法获取引用的指针，也不存在引用的引用。

> 注意引用类型不属于对象类型，所以才需要 `reference_wrapper`[1] 这种设施。

引用主要分为两种，左值引用和右值引用。此外还有两种特殊的引用：转发引用和垂悬引用，不作详细介绍。另外，本文还牵涉到一部分常值的内容，请用 常值 一文辅助阅读。

左值引用

" 左值和右值"

如果你不知道什么是左值和右值，可以参考 值类别 页面。

" 左值表达式"

如果一个表达式返回的是左值，那么这个表达式被称为左值表达式。右值表达式亦然。

通常我们会接触到的引用为左值引用，即绑定到左值的引用，但 `const` 的左值引用可以绑定到右值。以下是来自 参考手册[2] 的一段示例代码。

```cpp
#include <iostream>
#include <string>

int main() {
  std::string s = "Ex";
  std::string& r1 = s;
  const std::string& r2 = s;

  r1 += "ample";  // 修改 r1，即修改了 s
  // r2 += "!";               // 错误：不能通过到 const 的引用修改
  std::cout << r2 << '\n';  // 打印 r2，访问了 s，输出 "Example"
}
```

左值引用最常用的地方是函数参数，通过左值引用传参可以起到与通过指针传参相同的效果。

```cpp
#include <iostream>
#include <string>

// 参数中的 s 是引用，在调用函数时不会发生拷贝
char& char_number(std::string& s, std::size_t n) {
  s += s;           // 's' 与 main() 的 'str' 是同一对象
                    // 此处还说明左值也是可以放在等号右侧的
  return s.at(n);  // string::at() 返回 char 的引用
}

int main() {
  std::string str = "Test";
  char_number(str, 1) = 'a';  // 函数返回是左值，可被赋值
  std::cout << str << '\n';   // 此处输出 "TastTest"
}
```

右值引用 (C++ 11)

右值引用是绑定到右值的引用。右值**可以在内存里也可以在 CPU 寄存器中**。另外，右值引用可以被看作一种**延长临时对象生存期的方式**。

```cpp
#include <iostream>
#include <string>

int main() {
  std::string s1 = "Test";
  // std::string&& r1 = s1;           // 错误：不能绑定到左值

  const std::string& r2 = s1 + s1;  // 可行：到常值的左值引用延长生存期
```

```cpp
  // r2 += "Test";              // 错误：不能通过到常值的引用修改

  std::string&& r3 = s1 + s1;  // 可行：右值引用延长生存期
  r3 += "Test";  // 可行：能通过到非常值的右值引用修改
  std::cout << r3 << '\n';
}
```

在上述代码中，r3 是一个右值引用，引用的是右值 s1 + s1。r2 是一个左值引用，可以发现**右值引用可以转为 const 修饰的左值引用**。

一些例子

++i 和 i++

++i 和 i++ 是典型的左值和右值。++i 的实现是直接给 i 变量加一，然后返回 i 本身。因为 i 是内存中的变量，因此可以是左值。实际上前自增的函数签名是 T& T::operator++();。而 i++ 则不一样，它的实现是用临时变量存下 i，然后再对 i 加一，返回的是临时变量，因此是右值。后自增的函数签名是 T T::operator++(int);。

```cpp
int n1 = 1;
int n2 = ++n1;
int n3 = ++ ++n1;  // 因为是左值，所以可以继续操作
int n4 = n1++;
// int n5 = n1++ ++;   // 错误，无法操作右值
// int n6 = n1 + ++n1; // 未定义行为
int&& n7 = n1++;  // 利用右值引用延长生命期
int n8 = n7++;    // n8 = 5
```

移动语义和 std::move(C++11)

在 C++11 之后，C++ 利用右值引用新增了移动语义的支持，用来避免对象在堆空间的复制（但是无法避免栈空间复制），STL 容器对该特性有完整支持。具体特性有 移动构造函数[3]、移动赋值[4] 和具有移动能力的函数（参数里含有右值引用）。另外，std::move 函数可以用来产生右值引用，需要包含 <utility> 头文件。

注意：一个对象被移动后不应对其进行任何操作，无论是修改还是访问。被移动的对象处于有效但未指定的状态，具体内容依赖于 stl 的实现。如果需要访问（即指定一种状态），可以使用该对象的 swap 成员函数或者偏特化的 std::swap 交换两个对象（同样可以避免堆空间的复制）。

```cpp
// 移动构造函数
std::vector<int> v{1, 2, 3, 4, 5};
std::vector<int> v2(std::move(v));  // 移动 v 到 v2, 不发生拷贝

// 移动赋值函数
std::vector<int> v3;
v3 = std::move(v2);

// 有移动能力的函数
std::string s = "def";
std::vector<std::string> numbers;
numbers.push_back(std::move(s));
```

注意上述代码仅在 C++11 之后可用。

函数返回引用

让函数返回引用值可以避免函数在返回时对返回值进行拷贝，如

```
char &get_val(std::string &str, int index) { return str[index]; }
```

你不能返回在函数中的局部变量的引用，如果一定要在函数内的变量。请使用动态内存。例如如下两个函数都会产生悬垂引用，导致未定义行为。

```
std::vector<int>& getLVector() {  // 错误：返回局部变量的左值引用
  std::vector<int> x{1};
  return x;
}

std::vector<int>&& getRVector() {  // 错误：返回局部变量的右值引用
  std::vector<int> x{1};
  return std::move(x);
}
```

当右值引用指向的空间在进入函数前已经分配时，右值引用可以避免返回值拷贝。

```
struct Beta {
  Beta_ab ab;

  Beta_ab const& getAB() const& { return ab; }

  Beta_ab&& getAB() && { return std::move(ab); }
};

Beta_ab ab = Beta().getAB();  // 这里是移动语义，而非拷贝
```

参考内容

1. C++ 语言文档 —— 引用声明[2]
2. C++ 语言文档 —— 值类别[5]
3. Is returning by rvalue reference more efficient?[6]
4. 浅谈值类别及其历史[7]

参考资料与注释

[1] reference_wrapper

[2] 参考手册 [2-1] [2-2]

[3] 移动构造函数

[4] 移动赋值

[5] C++ 语言文档——值类别

[6] Is returning by rvalue reference more efficient?

[7] 浅谈值类别及其历史

4.4.6 常值

C++ 定义了一套完善的只读量定义方法，被常量修饰符 `const` 修饰的对象或类型都是只读量，只读量的内存存储与一般变量没有任何区别，但是编译器会在编译期进行冲突检查，避免对只读量的修改。因此合理使用 `const` 修饰符可以增加代码健壮性。

常类型

在类型的名字前增加 const 修饰会将该类型的变量标记为不可变的。具体使用情况有常量和常引用（指针）两种。

常量

这里的常量即常变量，指的是 const 类型的变量（而不是标题里泛指的只读量）。常类型量在声明之后便不可重新赋值，也不可访问其可变成员，只能访问其常成员。常成员的定义见后文。

" 类型限定符"

C++ 中类型限定符一共有三种：常量（const）、可变（mutable）和易变（volatile），其中默认情况下是可变变量，声明易变变量的情形是为了刻意避免编译器优化。

```
const int a = 0;  // a 的类型为 const int

// a = 1; // 报错，不能修改常量
```

常引用、常指针

常引用和常指针也与常量类似，但区别在于他们是限制了访问，而没有更改原变量的类型。

```
int a = 0;
const int b = 0;

int *p1 = &a;
*p1 = 1;
const int *p2 = &a;
// *p2 = 2; // 报错，不能通过常指针修改变量
// int *p3 = &b; // 报错，不能用普通指针指向 const 变量
const int *p4 = &b;

int &r1 = a;
r1 = 1;
const int &r2 = a;
// r2 = 2; // 报错，不能通过常引用修改变量
// int &p3 = b; // 报错，不能用普通引用指向 const 变量
const int &r4 = b;
```

另外需要区分开的是「常类型指针」和「常指针变量」(即常指针、指针常量)，例如下列声明

```
int* const p1;        // 类型为 int 的常指针，需要初始化
const int* p2;        // 类型为 const int 的指针
const int* const p3;  // 类型为 const int 的常指针

int (*f1)(int);  // 普通的函数指针
// int (const *f2)(int); // 指向常函数的指针，不可行
int (*const f3)(int) = some_func;  // 指向函数的常指针，需要初始化
int const* (*f4)(int);             // 指向返回常指针的函数指针
```

```cpp
int const* (*const f5)(int) = some_func;  // 指向返回常指针的函数的常指针
```

我们把常类型指针又称**底层指针**、常指针变量又称**顶层指针**。

另外，C++ 中还提供了 `const_cast` 运算符来强行去掉或者增加引用或指针类型的 const 限定，不到万不得已的时候请不要使用这个关键字。

常参数

在函数参数里限定参数为常类型可以避免在函数里意外修改参数，该方法通常用于引用参数。此外，在类型参数中添加 const 修饰符还能增加代码可读性，能区分输入和输出参数。

```cpp
void sum(const std::vector<int> &data, int &total) {
  for (auto iter = data.begin(); iter != data.end(); ++iter)
    total += *iter;  // iter 是 const 迭代器，解引用后的类型是 const int
}
```

常成员

常成员指的是类型中被 const 修饰的成员，常成员可以用来限制对常对象的修改。其中，常成员变量与常量声明相同，而常成员函数声明方法为在成员函数声明的**末尾**（参数列表的右括号的右边）添加 const 修饰符。

```cpp
#include <iostream>

struct ConstMember {
  const int *p;  // 常类型指针成员
  int *const q;  // 常指针变量成员
  int s;

  void func() { std::cout << "General Function" << std::endl; }

  void constFunc1() const { std::cout << "Const Function 1" << std::endl; }

  void constFunc2(int ss) const {
    // func(); // 常成员函数不能调用普通成员函数
    constFunc1();  // 常成员函数可以调用常成员函数

    // s = ss; // 常成员函数不能改变普通成员变量
    // p = &ss; // 常成员函数不能改变常成员变量
  }
};

int main() {
  const int a = 1;
  int b = 1;
  struct ConstMember c = {.p = &a, .q = &b};  // 指派初始化器是 C++20 中的一种语法
  // *(c.p) = 2; // 常类型指针无法改变指向的值
  c.p = &b;    // 常类型指针可以改变指针指向
  *(c.q) = 2;  // 常指针变量可以改变指向的值
  // c.q = &a; // 常指针变量无法改变指针指向
  const struct ConstMember d = c;
  // d.func(); // 常对象不能调用普通成员函数
  d.constFunc2(b);  // 常对象可以调用常成员函数
  return 0;
}
```

常表达式 constexpr（C++11）

> 另请参阅 常值： 常表达式 constexpr（C++11）

`constexpr` 说明符的作用是声明可以在编译时求得函数或变量的值，它的行为和 C 语言中的 `const` 关键字是一致的，会将变量结果直接编译到栈空间中。`constexpr` 还可以用来替换宏定义的常量，规避 宏定义的风险。`constexpr` 修饰的是变量和函数，而 `const` 修饰的是类型。

> note
>
> 实际上把 `const` 理解成”**readonly**”，而把 constexpr 理解成”**const**” 更加直观。

```cpp
constexpr int a = 10;  // 直接定义常量

constexpr int FivePlus(int x) { return 5 + x; }

void test(const int x) {
  std::array<x> c1;            // 错误，x 在编译期不可知
  std::array<FivePlus(6)> c2;  // 可行，FivePlus 编译期可以推断
}
```

参考资料

- C++ 关键字 ——const[1]
- C++ 关键字 ——constexpr[2]

参考资料与注释

[1] C++ 关键字——const

[2] C++ 关键字——constexpr

4.4.7 新版 C++ 特性

注意：考虑到算法竞赛的实际情况，本文将不会全面研究语法，只会讲述在算法竞赛中可能会应用到的部分。

本文语法参照 **C++11** 标准。语义不同的将以 **C++11** 作为标准，C++14、C++17 的语法视情况提及并会特别标注。

auto 类型说明符

`auto` 类型说明符用于自动推导变量等的类型。例如：

```cpp
auto a = 1;        // a 是 int 类型
auto b = a + 0.1;  // b 是 double 类型
```

基于范围的 for 循环

下面是 **C++20 前**基于范围的 `for` 循环的语法：

```
for (range_declaration : range_expression) loop_statement
```

上述语法产生的代码等价于下列代码（__range、__begin 和 __end 仅用于阐释）：

```
auto&& __range = range_expression;
for (auto __begin = begin_expr, __end = end_expr; __begin != __end; ++__begin) {
  range_declaration = *__begin;
  loop_statement
}
```

range_declaration 范围声明

范围声明是一个具名变量的声明，其类型是由范围表达式所表示的序列的元素的类型，或该类型的引用。通常用 auto 说明符进行自动类型推导。

range_expression 范围表达式

范围表达式是任何可以表示一个合适的序列（数组，或定义了 begin 和 end 成员函数或自由函数的对象）的表达式，或一个花括号初始化器列表。正因此，我们不应在循环体中修改范围表达式使其任何尚未被遍历到的「迭代器」（包括「尾后迭代器」）非法化。

这里有一个例子：

```
for (int i : {1, 1, 4, 5, 1, 4}) std::cout << i;
```

loop_statement 循环语句

循环语句可以是任何语句，常为一条复合语句，它是循环体。

这里有一个例子：

```
#include <iostream>

struct C {
  int a, b, c, d;

  C(int a = 0, int b = 0, int c = 0, int d = 0) : a(a), b(b), c(c), d(d) {}
};

int* begin(C& p) { return &p.a; }

int* end(C& p) { return &p.d + 1; }

int main() {
  C n = C(1, 9, 2, 6);
  for (auto i : n) std::cout << i << " ";
  std::cout << std::endl;
  // 下面的循环与上面的循环等价
  auto&& __range = n;
  for (auto __begin = begin(n), __end = end(n); __begin != __end; ++__begin) {
    auto ind = *__begin;
    std::cout << ind << " ";
  }
  std::cout << std::endl;
  return 0;
}
```

Lambda 表达式

详见 Lambda 表达式页面。

decltype 说明符

`decltype` 说明符可以推断表达式的类型。

```
#include <iostream>
#include <vector>

int main() {
  int a = 1926;
  decltype(a) b = a / 2 - 146;         // b 是 int 类型
  std::vector<decltype(b)> vec = {0};  // vec 是 std::vector <int> 类型
  std::cout << a << vec[0] << b << std::endl;
  return 0;
}
```

constexpr

> 另请参阅 新版 C++ 特性：constexpr

`constexpr` 说明符声明可以在编译时求得函数或变量的值。其与 `const` 的主要区别是一定会在编译时进行初始化。用于对象声明的 `constexpr` 说明符蕴含 `const`，用于函数声明的 `constexpr` 蕴含 `inline`。来看一个例子

```
int fact(int x) { return x ? x * fact(x - 1) : 1; }

int main() {
  constexpr int a = fact(5);  // ERROR: 函数调用在常量表达式中必须具有常量值
  return 0;
}
```

在 `int fact(int x)` 之前加上 `constexpr` 则编译通过。

std::tuple

`std::tuple` 定义于头文件 `<tuple>`，是固定大小的异类值汇集（在确定初始元素后不能更改，但是初始元素能有任意多个）。它是 `std::pair` 的推广。来看一个例子：

```
#include <iostream>
#include <tuple>
#include <vector>

constexpr auto expr = 1 + 1 * 4 - 5 - 1 + 4;

int main() {
  std::vector<int> vec = {1, 9, 2, 6, 0};
  std::tuple<int, int, std::string, std::vector<int> > tup =
      std::make_tuple(817, 114, "514", vec);
  std::cout << std::tuple_size<decltype(tup)>::value << std::endl;

  for (auto i : std::get<expr>(tup)) std::cout << i << " ";
  // std::get<> 中尖括号里面的必须是整型常量表达式
```

```
  // expr 常量的值是 3，注意 std::tuple 的首元素编号为 0，
  // 故我们 std::get 到了一个 std::vector<int>
  return 0;
}
```

成员函数

函数	作用
operator=	赋值一个 tuple 的内容给另一个
swap	交换二个 tuple 的内容

例子

```
constexpr std::tuple<int, int> tup = {1, 2};
std::tuple<int, int> tupA = {2, 3}, tupB;
tupB = tup;
tupB.swap(tupA);
```

非成员函数

函数	作用
make_tuple	创建一个 tuple 对象，其类型根据各实参类型定义
std::get	元组式访问指定的元素
operator== 等	按字典顺序比较 tuple 中的值
std::swap	特化的 std::swap 算法

例子

```
std::tuple<int, int> tupA = {2, 3}, tupB;
tupB = std::make_tuple(1, 2);
std::swap(tupA, tupB);
std::cout << std::get<1>(tupA) << std::endl;
```

std::function

类模板 std::function 是通用多态函数封装器，定义于头文件 <functional>。std::function 的实例能存储、复制及调用任何可调用（*Callable*）目标——函数、Lambda 表达式或其他函数对象，还有指向成员函数指针和指向数据成员指针。

存储的可调用对象被称为 std::function 的**目标**。若 std::function 不含目标，则称它为**空**。调用空 std::function 的目标将导致抛出 std::bad_function_call 异常。

来看例子

```
#include <functional>
#include <iostream>

struct Foo {
  Foo(int num) : num_(num) {}
```

```
  void print_add(int i) const { std::cout << num_ + i << '\n'; }

  int num_;
};

void print_num(int i) { std::cout << i << '\n'; }

struct PrintNum {
  void operator()(int i) const { std::cout << i << '\n'; }
};

int main() {
  // 存储自由函数
  std::function<void(int)> f_display = print_num;
  f_display(-9);

  // 存储 Lambda
  std::function<void()> f_display_42 = []() { print_num(42); };
  f_display_42();

  // 存储到成员函数的调用
  std::function<void(const Foo&, int)> f_add_display = &Foo::print_add;
  const Foo foo(314159);
  f_add_display(foo, 1);
  f_add_display(314159, 1);

  // 存储到数据成员访问器的调用
  std::function<int(Foo const&)> f_num = &Foo::num_;
  std::cout << "num_: " << f_num(foo) << '\n';

  // 存储到函数对象的调用
  std::function<void(int)> f_display_obj = PrintNum();
  f_display_obj(18);
}
```

可变参数宏

可变参数宏是 C99 引入的一个特性，C++ 从 C++11 开始支持这一特性。可变参数宏允许宏定义可以拥有可变参数，例如：

```
#define def_name(...) def_body(__VA_ARGS__)
```

其中，`...` 是缺省符号，`__VA_ARGS__` 在调用时会替换成实际的参数列表，`def_body` 应为可变参数模板函数。

现在就可以这么调用 `def_name`：

```
def_name();
def_name(1);
def_name(1, 2, 3);
def_name(1, 0.0, "abc");
```

可变参数模板

在 C++11 之前，类模板和函数模板都只能接受固定数目的模板参数。C++11 允许**任意个数、任意类型**的模板参数。

可变参数模板类

例如，下列代码声明的模板类 tuple 的对象可以接受任意个数、任意类型的模板参数作为它的模板形参。

```
template <typename... Values>
class Tuple {};
```

其中，Values 是一个模板参数包，表示 0 个或多个额外的类型参数。模板类只能含有一个模板参数包，且模板参数包必须位于所有模板参数的最右侧。

所以，可以这么声明 tuple 的对象：

```
Tuple<> test0;
Tuple<int> test1;
Tuple<int, int, int> test2;
Tuple<int, std::vector<int>, std::map<std::string, std::vector<int>>> test3;
```

如果要限制至少有一个模板参数，可以这么定义模板类 tuple：

```
template <typename First, typename... Rest>
class Tuple {};
```

可变参数模板函数

同样的，下列代码声明的模板函数 fun 可以接受任意个数、任意类型的模板参数作为它的模板形参。

```
template <typename... Values>
void fun(Values... values) {}
```

其中，Values 是一个模板参数包，values 是一个函数参数包，表示 0 个或多个函数参数。模板函数只能含有一个模板参数包，且模板参数包必须位于所有模板参数的最右侧。

所以，可以这么调用 fun 函数：

```
fun();
fun(1);
fun(1, 2, 3);
fun(1, 0.0, "abc");
```

参数包展开

之前说面了如何声明模板类或者模板函数，但是具体怎么使用传进来的参数呢？这个时候就需要参数包展开。

对于模板函数而言，参数包展开的方式有递归函数方式展开以及逗号表达式和参数列表方式展开。

对于模板类而言，参数包展开的方式有模板递归方式展开和继承方式展开。

递归函数方式展开参数包

递归函数方式展开参数包需要提供展开参数包的递归函数和参数包展开的终止函数。

举个例子，下面这个代码段使用了递归函数方式展开参数包，实现了可接受大于等于 2 个参数的取最大值函数。

```
// 递归终止函数，可以是 0 或多个参数。
template <typename T>
T MAX(T a, T b) {
  return a > b ? a : b;
}

// 展开参数包的递归函数
template <typename First, typename... Rest>
```

```cpp
First MAX(First first, Rest... rest) {
  return MAX(first, MAX(rest...));
}

// int a = MAX(1); // 编译不通过，但是对 1 个参数取最大值本身也没有意义
// int b = MAX(1, "abc"); //
// 编译不通过，但是在整数和字符串间取最大值本身也没有意义
int c = MAX(1, 233);              // 233
int d = MAX(1, 233, 666, 10086);  // 10086
```

可变参数模板的应用

举个应用的例子，有的人在 debug 的时候可能不喜欢用 IDE 的调试功能，而是喜欢输出中间变量。但是，有时候要输出的中间变量数量有点多，写输出中间变量的代码的时候可能会比较烦躁，这时候就可以用上可变参数模板和可变参数宏。

```cpp
// Author: Backl1ght
#include <bits/stdc++.h>
using namespace std;

namespace DEBUG {
template <typename T>
void _debug(const char* format, T t) {
  cerr << format << '=' << t << endl;
}

template <class First, class... Rest>
void _debug(const char* format, First first, Rest... rest) {
  while (*format != ',') cerr << *format++;
  cerr << '=' << first << ",";
  _debug(format + 1, rest...);
}

template <typename T>
ostream& operator<<(ostream& os, const vector<T>& V) {
  os << "[ ";
  for (const auto& vv : V) os << vv << ", ";
  os << "]";
  return os;
}

#define debug(...) _debug(#__VA_ARGS__, __VA_ARGS__)
}  // namespace DEBUG

using namespace DEBUG;

int main(int argc, char* argv[]) {
  int a = 666;
  vector<int> b({1, 2, 3});
  string c = "hello world";

  // before
  cout << "a=" << a << ", b=" << b << ", c=" << c
       << endl;  // a=666, b=[ 1, 2, 3, ], c=hello world
  // 如果用 printf 的话，在只有基本数据类型的时候是比较方便的，然是如果要输出 vector 等
```

```
的内容的话，就会比较麻烦

  // after
  debug(a, b, c);  // a=666, b=[ 1, 2, 3, ], c=hello world

  return 0;
}
```

这样一来，如果事先在代码模板里写好 DEBUG 的相关代码，后续输出中间变量的时候就会方便许多。

参考

1. C++ reference[1]
2. C++ 参考手册[2]
3. C++ in Visual Studio[3]
4. Variadic template[4]
5. Variadic macros[5]

参考资料与注释

[1] C++ reference

[2] C++ 参考手册

[3] C++ in Visual Studio

[4] Variadic template

[5] Variadic macros

4.4.8 Lambda 表达式

注意：考虑到算法竞赛的实际情况，本文将不会全面研究语法，只会讲述在算法竞赛中可能会应用到的部分。

本文语法参照 **C++11** 标准。语义不同的将以 **C++11** 作为标准，C++14、C++17 的语法视情况提及并会特别标注。

Lambda 表达式

Lambda 表达式因数学中的 λ 演算得名，直接对应于其中的 lambda 抽象。Lambda 表达式能够捕获作用域中的变量的无名函数对象。我们可以将其理解为一个匿名的内联函数，可以用来替换独立函数或者函数对象，从而使代码更可读。但是从本质上来讲，Lambda 表达式只是一种语法糖，因为它能完成的工作也可以用其他复杂的 C++ 语法来实现。

下面是 Lambda 表达式的语法：

```
[capture] (parameters) mutable -> return-type {statement}
```

下面我们分别对其中的 capture, parameters, mutable, return-type, statement 进行介绍。

capture 捕获子句

Lambda 表达式以 capture 子句开头，它指定哪些变量被捕获，以及捕获是通过值还是引用：有 & 符号前缀的变量通过引用访问，没有该前缀的变量通过值访问。空的 capture 子句 [] 指示 Lambda 表达式的主体不访问封闭范围中的变量。

我们也可以使用默认捕获模式：& 表示捕获到的所有变量都通过引用访问，= 表示捕获到的所有变量都通过值访问。之后我们可以为特定的变量**显式**指定相反的模式。

例如 Lambda 体要通过引用访问外部变量 a 并通过值访问外部变量 b，则以下子句等效：

- [&a, b]
- [b, &a]
- [&, b]
- [b, &]
- [=, &a]

默认捕获时，会捕获 Lambda 中提及的变量。获的变量成为 Lambda 的一部分；与函数参数相比，调用 Lambda 时不必传递它们。

以下是一些常见的例子：

```
int a = 0;
auto f = []() { return a * 9; };   // Error, 无法访问 'a'
auto f = [a]() { return a * 9; };  // OK, 'a' 被值「捕获」
auto f = [&a]() { return a++; };   // OK, 'a' 被引用「捕获」
                                   // 注意：请保证 Lambda 被调用时 a 没有被销毁
auto b = f();  // f 从捕获列表里获得 a 的值，无需通过参数传入 a
```

parameters 参数列表

大多数情况下类似于函数的参数列表，例如：

```
auto lam = [](int a, int b) { return a + b; };
std::cout << lam(1, 9) << " " << lam(2, 6) << std::endl;
```

C++14 中，若参数类型是泛型，则可以使用 auto 声明类型：

```
auto lam = [](auto a, auto b)
```

一个例子：

```
int x[] = {5, 1, 7, 6, 1, 4, 2};
std::sort(x, x + 7, [](int a, int b) { return (a > b); });
for (auto i : x) std::cout << i << " ";
```

这将打印出 x 数组从大到小排序后的结果。

由于 **parameters 参数列表**是可选的，如果不将参数传递给 Lambda 表达式，并且其 Lambda 声明器不包含 mutable，且没有后置返回值类型，则可以省略空括号。

Lambda 表达式也可以将另一个 Lambda 表达式作为其参数。

一个例子：

```
#include <functional>
#include <iostream>

int main() {
  using namespace std;
```

```cpp
  // 返回另一个计算两数之和 Lambda 表达式
  auto addtwointegers = [](int x) -> function<int(int)> {
    return [=](int y) { return x + y; };
  };

  // 接受另外一个函数 f 作为参数，返回 f(z) 的两倍
  auto higherorder = [](const function<int(int)>& f, int z) {
    return f(z) * 2;
  };

  // 调用绑定到 higherorder 的 Lambda 表达式
  auto answer = higherorder(addtwointegers(7), 8);

  // 答案为 (7 + 8) * 2 = 15
  cout << answer << endl;
}
```

mutable 可变规范

利用可变规范，Lambda 表达式的主体可以修改通过值捕获的变量。若使用此关键字，则 parameters **不可省略**（即使为空）。

一个例子，使用 **capture 捕获字句**中的例子，来观察 a 的值的变化：

```cpp
int a = 0;
auto func = [a]() mutable { ++a; };
```

此时 lambda 中的 a 的值改变为 1，lambda 外的 a 保持不变。

return-type 返回类型

用于指定 Lambda 表达式的返回类型。若没有指定返回类型，则返回类型将被自动推断（行为与用 `auto` 声明返回值的普通函数一致）。具体的，如果函数体中没有 `return` 语句，返回类型将被推导为 `void`，否则根据返回值推导。若有多个 `return` 语句且返回值类型不同，将产生编译错误。

例如，上文的 `lam` 也可以写作：

```cpp
auto lam = [](int a, int b) -> int
```

再举两个例子：

```cpp
auto x1 = [](int i) { return i; };  // OK
auto x2 = [] { return {1, 2}; };    // Error, 返回类型被推导为 void
```

statement Lambda 主体

Lambda 主体可包含任何函数可包含的部分。普通函数和 Lambda 表达式主体均可访问以下变量类型：

- 从封闭范围捕获变量
- 参数
- 本地声明的变量
- 在一个 `class` 中声明时，若捕获 `this`，则可以访问该对象的成员
- 具有静态存储时间的任何变量，如全局变量

下面是一个例子

```cpp
#include <iostream>

int main() {
  int m = 0, n = 0;
  [&, n](int a) mutable { m = (++n) + a; }(4);
  std::cout << m << " " << n << std::endl;
  return 0;
}
```

最后我们得到输出 5 0。这是由于 n 是通过值捕获的，在调用 Lambda 表达式后仍保持原来的值 0 不变。mutable 规范允许 n 在 Lambda 主体中被修改，将 mutable 删去则编译不通过。

使用类完成更复杂的操作

在 C++11 前没有 Lambda 表达式，但可以使用稍复杂的方法替代，尽管看上去更复杂却更易理解及扩展。

首先我们已经知道 Lambda 本质是一个可调用的对象，那么直接定义一个类并构造一个对象，重载其 operator() 运算符就可以完成和 Lambda 一样的操作，下面看一个简单的例子，我们将使用 C++17 的语法：

```cpp
#include <iostream>

struct AbstractCallable {
  AbstractCallable(int v) : v_(v) {}

  virtual ~AbstractCallable() = default;
  virtual int operator()(int) const = 0;  // 纯虚函数需要继承后实现才可以实例化
  int v_;
};

AbstractCallable *get_CallableObject() {
  struct Callable : public AbstractCallable {
    Callable() : AbstractCallable(10) {}

    int operator()(int k) const override {
      std::cout << v_ + k << std::endl;
      return v_ - 10;
    }
  };

  return new Callable;
}

int main() {
  auto t = get_CallableObject();
  std::cout << t->operator()(5);  // 或者等价的 `(*t)(5);`
  delete t;
  return 0;
}
```

在写 Lambda 表达式时，我们几乎都可以将其等价的映射为上面这种形式。

Lambda 表达式相关语法	类的语法
capture 捕获子句	构造函数
-	析构函数

Lambda 表达式相关语法	类的语法
使用 `std::function` 包装传递	基类指针/引用传递
拷贝多个 Lambda 的函数对象	自定义的拷贝函数
`mutable`	`operator()` 函数是否为 `const`

在 Lambda 的捕获子句中分为引用捕获和按值捕获（暂不考虑比较特殊的捕获 `this` 等），而在类的构造函数中我们可以更精细的控制这一点，另外自定义的析构函数的存在也方便我们更好的扩展，缺点是不够「匿名」，因为仍需要类名。

假设我们有一个函数

```
void func(std::function<int(int)>);
```

那么上述用例中就可以改为

```
void func(AbstractCallable *);
```

并且既然 `Callable` 是一个可调用对象，我们也可以通过 `std::bind(&AbstractCallable::operator(), t, std::placeholders::_1)` 来将其转换再转换为 `std::function<int(int)>`。

如果不需要实现类似 `std::function` 的包装，那么也无需使用抽象基类，这样便和一般的 Lambda 表达式一样不会产生额外的虚拟函数表的开销。

参考文献

- https://en.cppreference.com/w/cpp/language/lambda[1]

参考资料与注释

[1] https://en.cppreference.com/w/cpp/language/lambda

4.4.9 pb_ds

pb_ds 简介

> **Authors:** HeRaNO, Xeonacid, saffahyjp

pb_ds 库全称 Policy-Based Data Structures。

pb_ds 库封装了很多数据结构，比如哈希（Hash）表，平衡二叉树，字典树（Trie 树），堆（优先队列）等。

就像 `vector`、`set`、`map` 一样，其组件均符合 STL 的相关接口规范。部分（如优先队列）包含 STL 内对应组件的所有功能，但比 STL 功能更多。

pb_ds 只在使用 libstdc++ 为标准库的编译器下可以用。

可以使用 `begin()` 和 `end()` 来获取 `iterator` 从而遍历

可以 `increase_key,decrease_key` 以及删除单个元素

由于 pb_ds 库的主要内容在以下划线开头的 `__gnu_pbds` 命名空间中，在 NOI 系列活动中的合规性一直没有确定。2021 年 9 月 1 日，根据 《 关于 NOI 系列活动中编程语言使用限制的补充说明 》[1]，允许使用以下划线开头的库函数或宏（但具有明确禁止操作的库函数和宏除外），在 NOI 系列活动中使用 pb_ds 库的合规性有了文件上的依据。

参考资料：《C++ 的 pb_ds 库在 OI 中的应用 》[2]

参考资料与注释

[1] 《关于 NOI 系列活动中编程语言使用限制的补充说明》

[2] 《C++ 的 pb_ds 库在 OI 中的应用》

堆

Authors: Xeonacid, ouuan, Ir1d, WAAutoMaton, Chrogeek, abc1763613206, Planet6174, i-Yirannn, opsiff, GoodCoder666

`__gnu_pbds :: priority_queue`

附：官方文档地址 —— 复杂度及常数测试[1]

```
#include <ext/pb_ds/priority_queue.hpp>
using namespace __gnu_pbds;
__gnu_pbds ::priority_queue<T, Compare, Tag, Allocator>
```

模板形参

- `T`: 储存的元素类型
- `Compare`: 提供严格的弱序比较类型
- `Tag`: 是 `__gnu_pbds` 提供的不同的五种堆，Tag 参数默认是 `pairing_heap_tag` 五种分别是：
  - `pairing_heap_tag`：配对堆官方文档认为在非原生元素（如自定义结构体/`std :: string`/`pair`）中，配对堆表现最好
  - `binary_heap_tag`：二叉堆官方文档认为在原生元素中二叉堆表现最好，不过笔者测试的表现并没有那么好
  - `binomial_heap_tag`：二项堆二项堆在合并操作的表现要优于二叉堆，但是其取堆顶元素操作的复杂度比二叉堆高
  - `rc_binomial_heap_tag`：冗余计数二项堆
  - `thin_heap_tag`：除了合并的复杂度都和 Fibonacci 堆一样的一个 tag
- `Allocator`：空间配置器，由于 OI 中很少出现，故这里不做讲解

由于本篇文章只是提供给学习算法竞赛的同学们，故对于后四个 tag 只会简单的介绍复杂度，第一个会介绍成员函数和使用方法。

经作者本机 Core i5 @3.1 GHz On macOS 测试堆的基础操作，结合 GNU 官方的复杂度测试，Dijkstra 测试，都表明：至少对于 OIer 来讲，除了配对堆的其他四个 tag 都是鸡肋，要么没用，要么常数大到不如 `std` 的，且有可能造成 MLE，故这里只推荐用默认的配对堆。同样，配对堆也优于 `algorithm` 库中的 `make_heap()`。

构造方式

要注明命名空间因为和 `std` 的类名称重复。

```
__gnu_pbds ::priority_queue<int> __gnu_pbds::priority_queue<int, greater<int> >
__gnu_pbds ::priority_queue<int, greater<int>, pairing_heap_tag>
__gnu_pbds ::priority_queue<int>::point_iterator id; // 点类型迭代器
// 在 modify 和 push 的时候都会返回一个 point_iterator，下文会详细的讲使用方法
id = q.push(1);
```

成员函数

- `push()`: 向堆中压入一个元素，返回该元素位置的迭代器。
- `pop()`: 将堆顶元素弹出。
- `top()`: 返回堆顶元素。
- `size()` 返回元素个数。
- `empty()` 返回是否非空。
- `modify(point_iterator, const key)`: 把迭代器位置的 `key` 修改为传入的 `key`，并对底层储存结构进行排序。
- `erase(point_iterator)`: 把迭代器位置的键值从堆中擦除。
- `join(__gnu_pbds :: priority_queue &other)`: 把 `other` 合并到 `*this` 并把 `other` 清空。

使用的 tag 决定了每个操作的时间复杂度：

	push	pop	modify	erase	Join
`pairing_heap_tag`	$O(1)$	最坏 $\Theta(n)$ 均摊 $\Theta(\log(n))$	最坏 $\Theta(n)$ 均摊 $\Theta(\log(n))$	最坏 $\Theta(n)$ 均摊 $\Theta(\log(n))$	$O(1)$
`binary_heap_tag`	最坏 $\Theta(n)$ 均摊 $\Theta(\log(n))$	最坏 $\Theta(n)$ 均摊 $\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
`binomial_heap_tag`	最坏 $\Theta(\log(n))$ 均摊 $O(1)$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$
`rc_binomial_heap_tag`	$O(1)$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$
`thin_heap_tag`	$O(1)$	最坏 $\Theta(n)$ 均摊 $\Theta(\log(n))$	最坏 $\Theta(\log(n))$ 均摊 $O(1)$	最坏 $\Theta(n)$ 均摊 $\Theta(\log(n))$	$\Theta(n)$

示例

```cpp
#include <algorithm>
#include <cstdio>
#include <ext/pb_ds/priority_queue.hpp>
#include <iostream>
using namespace __gnu_pbds;
// 由于面向 OIer, 本文以常用堆 : pairing_heap_tag 作为范例
// 为了更好的阅读体验，定义宏如下 :
#define pair_heap __gnu_pbds ::priority_queue<int>
pair_heap q1;  // 大根堆, 配对堆
pair_heap q2;
pair_heap ::point_iterator id;  // 一个迭代器

int main() {
  id = q1.push(1);
  // 堆中元素 :  [1];
  for (int i = 2; i <= 5; i++) q1.push(i);
  // 堆中元素 :  [1, 2, 3, 4, 5];
  std ::cout << q1.top() << std ::endl;
  // 输出结果 : 5;
  q1.pop();
  // 堆中元素 : [1, 2, 3, 4];
```

```
  id = q1.push(10);
  // 堆中元素 : [1, 2, 3, 4, 10];
  q1.modify(id, 1);
  // 堆中元素 :  [1, 1, 2, 3, 4];
  std ::cout << q1.top() << std ::endl;
  // 输出结果 : 4;
  q1.pop();
  // 堆中元素 : [1, 1, 2, 3];
  id = q1.push(7);
  // 堆中元素 : [1, 1, 2, 3, 7];
  q1.erase(id);
  // 堆中元素 : [1, 1, 2, 3];
  q2.push(1), q2.push(3), q2.push(5);
  // q1 中元素 : [1, 1, 2, 3], q2 中元素 : [1, 3, 5];
  q2.join(q1);
  // q1 中无元素，q2 中元素 : [1, 1, 1, 2, 3, 3, 5];
}
```

__gnu_pbds 迭代器的失效保证（invalidation_guarantee）

在上述示例以及一些实践中（如使用本章的 pb-ds 堆来编写单源最短路等算法），常常需要保存并使用堆的迭代器（如 `__gnu_pbds::priority_queue<int>::point_iterator` 等）。

可是例如对于 `__gnu_pbds::priority_queue` 中不同的 Tag 参数，其底层实现并不相同，迭代器的失效条件也不一样，根据 __gnu_pbds 库的设计，以下三种由上至下派生的情况：

1. 基本失效保证（basic_invalidation_guarantee）：即不修改容器时，点类型迭代器（point_iterator）、指针和引用（key/value）**保持**有效。
2. 点失效保证（point_invalidation_guarantee）：即**修改**容器后，点类型迭代器（point_iterator）、指针和引用（key/value）只要对应在容器中没被删除**保持**有效。
3. 范围失效保证（range_invalidation_guarantee）：即**修改**容器后，除（2）的特性以外，任何范围类型的迭代器（包括 `begin()` 和 `end()` 的返回值）是正确的，具有范围失效保证的 Tag 有 `rb_tree_tag` 和适用于 `__gnu_pbds::tree` 的 `splay_tree_tag`，以及适用于 `__gnu_pbds::trie` 的 `pat_trie_tag`。

从运行下述代码中看出，除了 `binary_heap_tag` 为 `basic_invalidation_guarantee` 在修改后迭代器会失效，其余的均为 `point_invalidation_guarantee` 可以实现修改后点类型迭代器（point_iterator）不失效的需求。

```cpp
#include <bits/stdc++.h>
using namespace std;
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/priority_queue.hpp>
using namespace __gnu_pbds;
#include <cxxabi.h>

template <typename T>
void print_invalidation_guarantee() {
  typedef typename __gnu_pbds::container_traits<T>::invalidation_guarantee gute;
  cout << abi::__cxa_demangle(typeid(gute).name(), 0, 0, 0) << endl;
}

int main() {
  typedef
      typename __gnu_pbds::priority_queue<int, greater<int>, pairing_heap_tag>
          pairing;
  typedef
```

```
        typename __gnu_pbds::priority_queue<int, greater<int>, binary_heap_tag>
            binary;
    typedef
        typename __gnu_pbds::priority_queue<int, greater<int>, binomial_heap_tag>
            binomial;
    typedef typename __gnu_pbds::priority_queue<int, greater<int>,
                                                rc_binomial_heap_tag>
        rc_binomial;
    typedef typename __gnu_pbds::priority_queue<int, greater<int>, thin_heap_tag>
        thin;
    print_invalidation_guarantee<pairing>();
    print_invalidation_guarantee<binary>();
    print_invalidation_guarantee<binomial>();
    print_invalidation_guarantee<rc_binomial>();
    print_invalidation_guarantee<thin>();
    return 0;
}
```

参考资料与注释

[1] 官方文档地址——复杂度及常数测试

平衡树

__gnu_pbds :: tree

附：官方文档地址[1]

```
#include <ext/pb_ds/assoc_container.hpp>  // 因为 tree 定义在这里所以需要包含这个头文件
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
__gnu_pbds ::tree<Key, Mapped, Cmp_Fn = std::less<Key>, Tag = rb_tree_tag,
                  Node_Update = null_tree_node_update,
                  Allocator = std::allocator<char> >
```

模板形参

- `Key`: 储存的元素类型，如果想要存储多个相同的 `Key` 元素，则需要使用类似于 `std::pair` 和 `struct` 的方法，并配合使用 `lower_bound` 和 `upper_bound` 成员函数进行查找
- `Mapped`: 映射规则（Mapped-Policy）类型，如果要指示关联容器是**集合**，类似于存储元素在 `std::set` 中，此处填入 `null_type`，低版本 `g++` 此处为 `null_mapped_type`；如果要指示关联容器是**带值的集合**，类似于存储元素在 `std::map` 中，此处填入类似于 `std::map<Key, Value>` 的 `Value` 类型
- `Cmp_Fn`: 关键字比较函子，例如 `std::less<Key>`
- `Tag`: 选择使用何种底层数据结构类型，默认是 `rb_tree_tag`。`__gnu_pbds` 提供不同的三种平衡树，分别是：
  - `rb_tree_tag`：红黑树，一般使用这个，后两者的性能一般不如红黑树
  - `splay_tree_tag`：splay 树
  - `ov_tree_tag`：有序向量树，只是一个由 `vector` 实现的有序结构，类似于排序的 `vector` 来实现平衡树，性能取决于数据想不想卡你
- `Node_Update`：用于更新节点的策略，默认使用 `null_node_update`，若要使用 `order_of_key` 和 `find_by_order` 方法，需要使用 `tree_order_statistics_node_update`
- `Allocator`：空间分配器类型

构造方式

```
__gnu_pbds::tree<std::pair<int, int>, __gnu_pbds::null_type,
                 std::less<std::pair<int, int> >, __gnu_pbds::rb_tree_tag,
                 __gnu_pbds::tree_order_statistics_node_update>
    trr;
```

成员函数

- `insert(x)`：向树中插入一个元素 x，返回 `std::pair<point_iterator, bool>`。
- `erase(x)`：从树中删除一个元素/迭代器 x，返回一个 `bool` 表明是否删除成功。
- `order_of_key(x)`：返回 x 以 `Cmp_Fn` 比较的排名。
- `find_by_order(x)`：返回 `Cmp_Fn` 比较的排名所对应元素的迭代器。
- `lower_bound(x)`：以 `Cmp_Fn` 比较做 `lower_bound`，返回迭代器。
- `upper_bound(x)`：以 `Cmp_Fn` 比较做 `upper_bound`，返回迭代器。
- `join(x)`：将 x 树并入当前树，前提是两棵树的类型一样，x 树被删除。
- `split(x,b)`：以 `Cmp_Fn` 比较，小于等于 x 的属于当前树，其余的属于 b 树。
- `empty()`：返回是否为空。
- `size()`：返回大小。

示例

```cpp
// Common Header Simple over C++11
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef unsigned long long ull;
typedef long double ld;
typedef pair<int, int> pii;
#define pb push_back
#define mp make_pair
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
__gnu_pbds ::tree<pair<int, int>, __gnu_pbds::null_type, less<pair<int, int> >,
                  __gnu_pbds::rb_tree_tag,
                  __gnu_pbds::tree_order_statistics_node_update>
    trr;

int main() {
  int cnt = 0;
  trr.insert(mp(1, cnt++));
  trr.insert(mp(5, cnt++));
  trr.insert(mp(4, cnt++));
  trr.insert(mp(3, cnt++));
  trr.insert(mp(2, cnt++));
  // 树上元素 {{1,0},{2,4},{3,3},{4,2},{5,1}}
  auto it = trr.lower_bound(mp(2, 0));
  trr.erase(it);
  // 树上元素 {{1,0},{3,3},{4,2},{5,1}}
  auto it2 = trr.find_by_order(1);
  cout << (*it2).first << endl;
  // 输出排名 0 1 2 3 中的排名 1 的元素的 first:1
```

```cpp
  int pos = trr.order_of_key(*it2);
  cout << pos << endl;
  // 输出排名
  decltype(trr) newtr;
  trr.split(*it2, newtr);
  for (auto i = newtr.begin(); i != newtr.end(); ++i) {
    cout << (*i).first << ' ';
  }
  cout << endl;
  // {4,2},{5,1} 被放入新树
  trr.join(newtr);
  for (auto i = trr.begin(); i != trr.end(); ++i) {
    cout << (*i).first << ' ';
  }
  cout << endl;
  cout << newtr.size() << endl;
  // 将 newtr 树并入 trr 树, newtr 树被删除。
  return 0;
}
```

参考资料与注释

[1] 官方文档地址

4.4.10 编译优化

Authors: inclyc

OI 界的常用编程语言是 C++。既然使用了这门语言，就注定要和编译器、语言标准打交道了。众所周知，C++ 非常混乱邪恶，本文旨在给出实用的编译器相关知识，足够竞赛使用。

编译器优化简介

什么是优化 (Optimization)

根据 如同规则[2]（The as-if Rule），在保持语义不变的情况下，对程序运行速度、程序可执行文件大小作出改进。

常见的编译器优化

常量折叠 (Constant Folding)

常量折叠，又称常量传播 (Constant Propagation)，如果一个表达式可以确定为常量，在他的下一个定义 (Definition) 前，可以进行常量传播。

```cpp
int x = 1;
int y = x;  // x = 1, => y = 1
x = 3;
int z = 2 * y;   // z => 2 * y = 2 * 1 = 2
int y2 = x * 2;  // x = 3, => y2 = 6
```

这段代码在编译期间即可被转换为：

```cpp
int x = 1;
int y = 1;
```

```
x = 3;
int z = 2;
int y2 = 6;
```

实例：https://godbolt.org/z/oEfY35TTd[3]

死代码消除 (Deadcode Elimination)

故名思义，就是一段代码没用上就会被删去。

```
int test() {
  int a = 233;
  int b = a * 2;
  int c = 234;
  return c;
}
```

将被转换为

```
int test() { return 234; }
```

注意，这个代码首先进行了常量折叠，使得返回值可以确定为 234，a, b 为不活跃变量，因此删除。

循环旋转 (Loop Rotate)

将循环从"for" 形式，转换为"do-while" 形式，前面再多加一个条件判断。这个变换主要为其他变换做准备。

```
for (int i = 0; i < n; ++i) {
  auto v = *p;
  use(v);
}
```

变换为

```
if (0 < n) {
  do {
    auto v = *p;
    use(v);
    ++i;
  } while (i < n);
}
```

循环不变量外提 (Loop Invariant Code Motion)

基于别名分析 (Alias Analysis)，将循环中被证明是不变量（可能包含内存访问，load/store，因此依赖别名分析）的代码外提出循环体，这样可以让循环体内部少一些代码。

```
for (int i = 0; i < n; ++i) {
  auto v = *p;
  use(v);
}
```

这个代码直观来看可以外提为：

```
auto v = *p;
for (int i = 0; i < n; ++i) {
```

```
  use(v);
}
```

但实际上，如果 `n <= 0`，这个循环永远不会被进入，但我们又执行了一条多的指令（可能有副作用!）。因此，循环通常被 Rotate 为 do-while 形式，这样可以方便插入一个"loop guard"。之后再进行循环不变量外提。

```
if (0 < n) {  // loop guard
  auto v = *p;
  do {
    use(v);
    ++i;
  } while (i < n);
}
```

循环展开 (Loop Unroll)

循环包含循环体和各类分支语句，需要现代 CPU 进行一定的分支预测。直接把循环展开，用一定的代码大小来换取运行时间。

```
for (int i = 0; i < 3; i++) {
  a[i] = i;
}
```

变换为：

```
a[0] = 0;
a[1] = 1;
a[2] = 2;
```

循环判断外提 (Loop Unswitching)

循环判断外提将循环中的条件式移到循环之外，然后在外部的两个条件各放置两个循环，这样可以增加循环向量化、并行化的可能性（通常简单循环更容易被向量化）。

```
// clang-format off
void before(int x) {
  for(;/* i in some range */;) {
    /* A */;
    if (/* condition */ x % 2) {
      /* B */;
    }
    /* C */;
  }
}

void after(int x) {
  if (/* condition */ x % 2) {
    for(;/* i in some range */;) {
      /* A */;
      /* B */; // 直接执行 B ，不进行循环判断
      /* C */;
    }
  } else {
     for(;/* i in some range */;) {
      /* A */;
               // 不执行 B
```

```
      /* C */;
    }
  }
}
```

代码布局优化 (Code Layout Optimizations)

程序在执行时，可以将执行的路径分为冷热路径 (cold/hot path)。CPU 跳转执行，绝大多数情况下没有直接顺序执行快，后者通常被编译器作者称为"fallthrough"。与之对应的，经常被执行到的代码成为热代码，与之相对的成为冷代码。OI 代码中，如果有一段是循环中的特判边界条件，或者异常处理，类似的逻辑，则此部分代码为冷代码。

基本块 (Basic Block)，是控制流的基本结构，一个过程 (Procedure) 由若干个基本块组成，形成一个有向图。生成可执行文件的过程中，编译器需要安排一个放置基本块的布局 (Layout)，而如何编排布局，是此优化的重点。

原则上，应该更偏好与将热代码放在一起，而将冷代码隔开。原因是这样能够更好地利用指令缓存，热代码的局部性会更好。

```cpp
// clang-format off
int hotpath; // <-- 热!
if (/* 边界条件 */ false) {
    // <-- 冷!
}
int hotpath_again;  // <-- 热!
```

基本块放置 (Basic Block Placement)

我们用 label 来表达一种「伪机器码」，这个 C++ 程序有两种翻译方法：

" 布局 1"

```
// clang-format off
hotblock1:
    Stmts; // <-- 热!
    if (/* 边界条件不成立 */ true)
        goto hotblock2; // 经常发生!  ------+
coldblock:                           /*   |   */
    Stmt; // <- 冷                        |
    Stmt; // <- 冷                        |
    Stmt; // <- 冷                        |  跨越了大量指令，代价高昂!
    Stmt; // <- 冷                        |
    Stmt; // <- 冷                        |
    Stmt; // <- 冷                        |
    Stmt; // <- 冷                        |
hotblock2:                          /*    |   */
    Stmts; // <- 热!           <----------+
```

另一种布局为：

" 布局 2"

```
// clang-format off
hotblock1:
    Stmts; // <-- 热!
    if (/* 边界条件 */ false)
        goto coldblock; // 很少发生
hotblock2:                         /*   |  低代价!   */
```

```
    Stmts; // <- 热!   <-----------------+
coldblock:
    Stmt; // <- 冷
    Stmt; // <- 冷
    Stmt; // <- 冷
    Stmt; // <- 冷
    Stmt; // <- 冷
```

我们看到后一种布局中，两个热代码块被放到了一起，执行效率更优秀。

为了告诉编译器分支是否容易被执行，可以使用 C++20 `[[likely]]` 和 `[[unlikely]]`:https://en.cppreference.com/w/cpp/language/attributes/likely[4]

如果比赛没有采用 C++20 以上标准，则可以利用 `__builtin_expect`(GNU Extension)。

```
#define likely(x) __builtin_expect(!!(x), 1)
#define unlikely(x) __builtin_expect(!!(x), 0)

if (unlikely(/* 一些边界条件检查 */ false)) {
  // 冷代码
}
```

冷热代码分离 (Hot Cold Splitting)

一个过程 (Procedure) 包含同时包含冷热路径，而冷代码较长，更好的做法是让冷代码作为函数调用，而不是阻断热路径。这同时也提示我们不要自作聪明的让所有函数 `inline`。冷代码对执行速度的阻碍比函数调用要多得多。

" 不好的代码布局"

```
// clang-format off
void foo() {
      // clang-format off
hotblock1:
    Stmts; // <-- 热!
    if (/* 边界条件不成立 */ true)
        goto hotblock2; // 经常发生!  ------+
coldblock:                             /*   |   */
    Stmt; // <- 冷                          |
    Stmt; // <- 冷                          |
    Stmt; // <- 冷                          |  跨越了大量指令，代价高昂!
    Stmt; // <- 冷                          |
    Stmt; // <- 冷                          |
    Stmt; // <- 冷                          |
    Stmt; // <- 冷                          |
hotblock2:                            /*    |   */
    Stmts; // <- 热!            <----------+
}
```

" 好的代码布局"

```
// clang-format off
void foo() {
hotblock1:
  Stmts;  // <-- 热!
  if (/* 边界条件 */ false)
    codeBlock();  // 将冷代码分离出，使得热路径对 cache 更友好
```

```
hotblock2:
  Stmts;  // <- 热!
}

void coldBlock() {
  Stmt;  // <- 冷
  Stmt;  // <- 冷
  Stmt;  // <- 冷
  Stmt;  // <- 冷
  Stmt;  // <- 冷
  Stmt;  // <- 冷
  Stmt;  // <- 冷
}
```

冷热代码分离，其实就是函数内联 (Function Inlining) 的反向操作，这一优化的存在启示我们，函数内联不一定会让程序跑的更快。甚至如果内联代码是冷代码，反而会让程序跑的更慢！一些编译器存在强制内联的编译选项，但不推荐使用。编译器内部有一个静态分析过程，计算每个基本块、分支的概率，以及一个函数调用相关的代价模型，以此决定是否内联，自己决定是否内联不一定比编译器的决策好。

事实上，在没有额外信息的情况下，编译器通常会假设分支跳转与不跳转的概率一致，以此为依据传播各个控制流路径的冷热程度。PGO (Profile Guided Optimization) 的一部分便是通过若干次性能测试与实验得出真正环境下的程序分支概率，这些信息可以让代码布局更加优秀。

函数内联 (Function Inlining)

函数调用通常需要寄存器和栈传递参数，调用者 (caller) 和被调用者 (callee) 都需要保存一定的寄存器状态，这个过程通常被叫做调用约定 (calling convention)。一个函数调用因此会引起一些时间损耗，而内联函数就是指将函数直接写在调用方过程中，不进行真正的函数调用。

```
int add(int x) { return x + 1; }

int foo() {
  int a = 1;
  a = add(a);
}
```

`add()` 可以被内联到 `foo()` 当中：

```
int foo() {
  int a = 1;
  a = a + 1;  // <-- add() 的函数体，未经过传参
}
```

`always_inline`，`__force_inline`

https://clang.llvm.org/docs/AttributeReference.html#always-inline-force-inline[5]

一些编译器提供了手动内联函数调用的方法，在函数前加 `__attribute__((always_inline))`。这样使用不一定会比函数调用快，编译器在这个时候相信程序员有足够好的判断能力。

尾调用优化 (Tail Call Optimization)

当一个函数调用位于函数体尾部的位置时，这种函数调用被成为尾调用 (Tail Call)。对于这种特殊形式的调用，可以进行一些特别的优化。绝大多数体系结构拥有 Frame Pointer (a.k.a FP) 和 Stack Pointer (a.k.a SP)，维护者函数的调用帧 (Frame)，而如果调用位于函数尾部，则我们可以不保留外层函数的调用记录，直接用内层函数取代。

用跳转指令代替函数调用

函数调用在绝大多数体系结构下，需要保存当前程序计数器 `$pc` 的位置，保存若干 caller saved register，以便回到现场。而尾调用不需要此过程，将被直接翻译为跳转指令，因为尾递归永远不会返回到函数运行的位置。

一个简单的例子：https://godbolt.org/z/e7b1safaW[6]

```
int test(int a);

int tailCall(int x) { return test(x); }
```

```
int test(int a);

int tailCall(int x) { return test(x); }
```

自动尾递归改写

如果一个函数的尾调用是自身，则此函数是尾递归的。广义来讲，间接递归（由两个函数以上共同形成递归）形成递归，且都是尾调用的，也属于尾递归的范畴。尾递归可以被编译器优化为非递归的形式，减小额外的栈开销和函数调用代价。许多算法竞赛选手热衷于写非递归的代码，在不开优化下这样可以极大优化代码的常数，然而如果开优化，递归代码生成的二进制质量和手写的代码没有什么区别。

```
int fac(int n) {
  if (n < 2) return 1;
  return /* 使用 */ n * fac(n - 1); /* 使用了变量 n ，无法直接做尾递归优化! */
}
```

注意到这个函数并不是尾递归的，但可以改写为：

```
int fac(int acc, int n) {
  if (n < 2) return acc;
  return fac(acc * n, n - 1);
}
```

新的代码即是尾递归的。

现代编译器可以自动帮你完成这个过程，如果你的代码有机会被改写为尾递归，则编译器可以识别出这种形式，然后完成改写。

尾递归消除 -Rpass=tailcallelim

既然函数已经尾递归，那就可以直接删除递归语句，通过一定的静态分析，将函数直接转换为非递归的形式。我们此处并不去深究编译器作者如何做到这一点，从实际体验来看，绝大多数 OI 代码，如果存在递归版本和非递归版本，则此代码一般可自动优化为非递归版本。这里给读者一些具体的例子：

"GCD[7]"

```
int gcd(int a, int b) { return b ? gcd(b, a % b) : a; }
```

"斐波那契数列[8]"

```
// 展开 fib(n - 2) 这一项
// fib(n - 1) 不能变换为非递归，优化后的代码依然是指数级别的
int fib(int n) {
  if (n < 2) return 1;
  return fib(n - 1) + fib(n - 2);
}
```

"阶乘[9]"

```
// 展开成标量循环，然后执行自动向量化，生成的代码是 SIMD 的
unsigned fac(unsigned n) {
  if (n < 2) return 1;
  return n * fac(n - 1);
}
```

这些函数被优化后的汇编和非递归版完全相同，递归将被直接消除。对于 OI 选手而言，可以在开 O2 的情况下放心写递归版本的各种算法，和非递归版不会有什么区别。如果你写的函数本身无法被改写成非递归的形式，那么编译器也无能为力。

强度削减 (Strength Reduction)

常见的编译优化。最简单的例子是 `x * 2` 变为 `x << 1`，第二种写法在 OI 中相当常见。编译器会自动做类似的优化，在打开优化开关的情况下，`x * 2` 和 `x << 1` 是完全等价的。强度削减 (Strength Reduction) 将高开销的指令转换为低开销的指令。

标量运算符变换

位运算代替乘法

```
int a;
a = x * 2;   // bad!
a = x << 1;  // good!
```

需要注意的是有符号数和无符号数在移位 (shifting) 和类型提升 (promotion) 层面有明显的差异。符号位在移位时有着特别的处理，包括算术移位和逻辑移位两种类型。这在编写二分查找/线段树等含有大量除二操作的时候表现突出，有符号整数除法不能直接优化为一步右移位运算。

```
int l, r;
/* codes */
int mid = (l + r) / 2; /* 如果编译器不能假定 l, r 非负，则会生成较差的代码 */
                       // 不能优化为
                       // mid = (l + r) >> 1
                       // 反例：
                       // mid = -127
                       // mid / 2 = -63
                       // mid >> 1 = -64
```

```
int mid = (l + r);
int sign = mid >> 31; /* 逻辑右移，得到符号位 */
mid += sign;
mid >>= 1; /* 算术右移 */
```

可行的解决方案：

- 用 `unsigned l, r;`，下标本来就应该是无符号的
- 在源代码中使用移位

乘法代替除法

```
int x = a / 3;
```

此过程可以被变换为 `x = a * 0x55555556 >> 32`，具体可以看 这篇知乎回答[10] 或者 原始论文[11]。

索引变量强度削减 (IndVars)

编译器自动识别出循环中的索引变量，并将相关的高开销过程转换为低开销

```
int a = 0;
for (int i = 1; i < 10; i++) {
  a = 3 * i;  // bad!
  a = a + 3;  // good!
}
```

此处如果直接使用 `a = 3 * i` 在 OI 中很常见，而编译器可以自动分析出，等价的变换为 `a = a + 3`，用代价更低的加法代替乘法。分析循环变量的迭代过程，被称为 SCEV (Scalar Evolution)。

SCEV 还可以做到优化一些循环：

```
int test(int n) {
  int ans = 1;
  for (int i = 0; i < n; i++) {
    ans += i * (i + 1);
  }
  return ans;
}
```

此函数会被优化为 $O(1)$ 公式求和，参考 https://godbolt.org/z/ET8d89vvK[12]。这个行为目前仅有基于 LLVM 的编译器会出现，GCC 编译器更加保守。

```
int test(int n) {
  int ans = 1;
  for (int i = 0; i < n; i++) {
    ans += i * (i + 1);
  }
  return ans;
}
```

自动向量化 (Auto-Vectorization)

单指令流多数据流是很好的提供单核并行的方法。使用这种指令，可以利用 CPU 的 SIMD 寄存器，比通用寄存器更宽，例如一次放 4 个整数然后计算。OI 选手不需要了解自动向量化的细节，通常而言，Clang 编译器会做比 GCC 更激进的自动向量化：

```
// https://godbolt.org/z/h1hx5sWoE
void test(int *a, int *b, int n) {
  for (int i = 0; i < n; i++) {
    a[i] += b[i];
  }
}
```

`__restrict` type specifier (GNU, MSVC)

两个任意指针对应的区域可能出现重叠 (overlap)，此时需要特判是否可以使用向量代码。下图展示了一个指针重叠的例子：

`__restrict` 作为一种约定使编译器假定两个指针所指向的内存区域永远不会重叠。

```
void test(int* __restrict a, int* __restrict b, int n) {
  for (int i = 0; i < n; i++) {
    a[i] += b[i];
  }
```

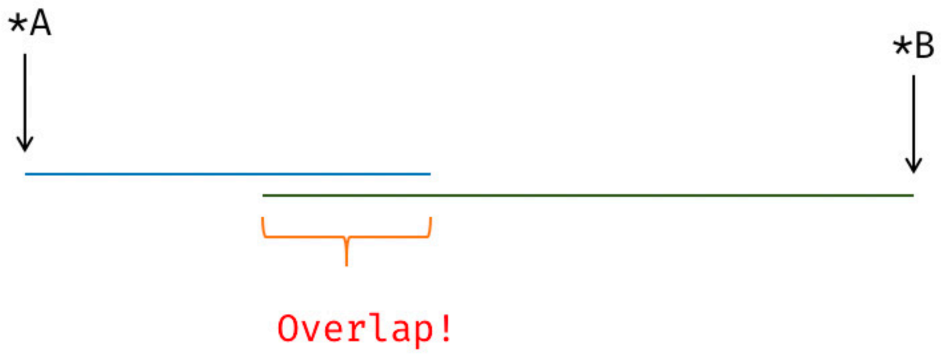


图 4.5

```
}
```

`__restrict` 并非 C++ 标准的一部分，但各大编译器都可以使用。此关键字影响自动向量化的代码生成质量，极端卡常的情况下可以使用。

和编译优化相关的常见语言误用

inline - 内联

函数内联在开 O2 的情况下通常由编译器自动完成。结构体定义中的 `inline` 完全是多余的，如果准备的比赛开 O2 优化，则完全不必声明为内联。如果不开 O2 则使用 `inline` 也不会让编译器真正内联。

`inline` 关键字在现代 C++ 被当作是一种链接、与导出符号的语义行为，而不是做函数内联。

register - 虚假的寄存器建议

现代编译器会直接忽略你的 `register` 关键字，你自己认为的寄存器分配一般没有编译器直接跑寄存器分配算法来的聪明。此关键字于 C++11 被弃用，于 C++17 被删除[1]。

https://en.cppreference.com/w/cpp/keyword/register[13]

未定义行为（Undefined Behavior）与编译优化

编译器可以认为 C++ 程序不存在 未定义行为[14]（undefined behavior，UB），因此在编译存在 UB 的程序时，编译器可能会产生意想不到的结果。同时，编译器也可以在假定不存在 UB 的情况下进行更加激进而自由的优化。

常见的 UB 有：

1. 有符号溢出[15]；
2. 使用未初始化的变量；
3. 访问越界；
4. 空指针解引用；
5. 无副作用的无限循环。

其他 UB 和示例等可通过扩展阅读详细了解。

有符号溢出

```cpp
int f(int x) { return x * 2 / 2; }
```

编译器可以假定程序不存在有符号溢出的行为，进而此函数可能被优化为

```
int f(int x) { return x; }
```

示例：https://godbolt.org/z/WKv3W5hvM[16]、https://godbolt.org/z/qqE9nxP1j[17]。

可通过 `-fwrapv`[18] 选项禁用该假设。示例：https://godbolt.org/z/5x3K5KGnr[19]、https://godbolt.org/z/4r4a4EzMW[20]。

使用未初始化的变量

```
int f(int x) {
  int a;
  if (x)  // either x nonzero or UB
    a = 42;
  return a;
}
```

编译器可以假定程序不存在使用未初始化变量的行为，所以 `a` 一定会被初始化，进而此函数可能被优化为

```
int f(int) { return 42; }
```

示例：https://godbolt.org/z/8WYMYYjdG[21]、https://godbolt.org/z/qvGd1nvv9[22]。

访问越界

```
int table[4] = {};

bool exists_in_table(int v) {
  // return true in one of the first 4 iterations or UB due to out-of-bounds
  // access
  for (int i = 0; i <= 4; i++)
    if (table[i] == v) return true;
  return false;
}
```

编译器可以假定程序不存在访问越界的行为，所以该函数一定会在发生访问越界之前返回，进而此函数可能被优化为

```
bool exists_in_table(int) { return true; }
```

示例：https://godbolt.org/z/xfePeYsE3[23]。

空指针解引用

```
int f(int* p) {
  int x = *p;
  if (!p)
    return x;  // Either UB above or this branch is never taken
  else
    return 0;
}
```

编译器可以假定程序不存在空指针解引用的行为，从而 `!p` 恒为 `false`，进而此函数可能被优化为

```
int f(int*) { return 0; }
```

示例：https://godbolt.org/z/GY1jvsrb5[24]、https://godbolt.org/z/4ronPsnxf[25]。

无副作用的无限循环

" 验证 Fermat 大定理"

由 Fermat 大定理[26] 可知，不定方程 $a^3 = b^3 + c^3$ 没有正整数解。下面的程序试图枚举 $[1, 1000]$ 内的整数验证该方程是否成立，若返回 `true` 则说明在 $[1, 1000]$ 范围内找到了一组整数解，从而 Fermat 大定理不成立。

```cpp
#include <iostream>

bool fermat() {
  const int max_value = 1000;

  // Endless loop with no side effects is UB
  for (int a = 1, b = 1, c = 1; true;) {
    if (((a * a * a) == ((b * b * b) + (c * c * c))))
      return true;  // disproved :()
    a++;
    if (a > max_value) {
      a = 1;
      b++;
    }
    if (b > max_value) {
      b = 1;
      c++;
    }
    if (c > max_value) c = 1;
  }

  return false;  // not disproved
}

int main() {
  std::cout << "Fermat's Last Theorem ";
  fermat() ? std::cout << "has been disproved!\n"
           : std::cout << "has not been disproved.\n";
}
```

编译器可以假定程序不存在无副作用的无限循环，从而认为 `fermat()` 函数中的 for 循环一定会在某一时刻终止并返回 `true`，最终程序可能输出：

```
Fermat's Last Theorem has been disproved!
```

示例：https://godbolt.org/z/d834MK7bz[27]、https://godbolt.org/z/Eov9nsKqf[28]。

Sanitizer

理智保证器。在运行时检查你的程序是否有未定义行为、数组越界、空指针，等等功能。在本地调试模式下，建议开启一些 sanitizer，可以极大缩短你的 Debug 时间。这些 sanitizer 由 Google 开发，绝大多数可以在 GCC 和 Clang 中使用。sanitizer 在 LLVM 中更加成熟，因此推荐选手本地使用 Clang 编译器进行相关除错。

Address Sanitizer -fsanitize=address

https://clang.llvm.org/docs/AddressSanitizer.html[29]

GCC 和 Clang 都支持这个 Sanitizer。包括如下检查项：

- 越界

- 释放后使用 (use-after-free)
- 返回后使用 (use-after-return)
- 重复释放 (double-free)
- 内存泄漏 (memory-leaks)
- 离开作用域后使用 (use-after-scope)

应用这项检查会让你的程序慢 2x 左右。

Undefined Behavior Sanitizer -fsanitize=undefined

https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html[30]

Undefined Behavior Sanitizer (a.k.a UBSan) 用于检查代码中的未定义行为。GCC 和 Clang 都支持这个 Sanitizer。自动检查你的程序有无未定义行为。UBSan 的检查项目包括：

- 位运算溢出，例如 32 位整数左移 72 位
- 有符号整数溢出
- 浮点数转换到整数数据溢出

UBSan 的检查项可选，对程序的影响参考提供的网页地址。

杂项

Compiler Explorer

在这里观察各个编译器的行为和汇编代码：https://godbolt.org[31]

参考资料与注释

扩展阅读

1. The LLVM Project Blog: What Every C Programmer Should Know About Undefined Behavior #1/3[32]
2. The LLVM Project Blog: What Every C Programmer Should Know About Undefined Behavior #2/3[33]
3. The LLVM Project Blog: What Every C Programmer Should Know About Undefined Behavior #3/3[34]

[1] Remove Deprecated Use of the register Keyword (open-std.org)

[2] 如同规则

[3] https://godbolt.org/z/oEfY35TTd

[4] https://en.cppreference.com/w/cpp/language/attributes/likely

[5] https://clang.llvm.org/docs/AttributeReference.html#always-inline-force-inline

[6] https://godbolt.org/z/e7b1safaW

[7] GCD

[8] 斐波那契数列

[9] 阶乘

[10] 这篇知乎回答

[11] 原始论文

[12] https://godbolt.org/z/ET8d89vvK

[13] https://en.cppreference.com/w/cpp/keyword/register

[14] 未定义行为

[15] 有符号溢出

[16] https://godbolt.org/z/WKv3W5hvM

[17] https://godbolt.org/z/qqE9nxP1j

[18] `-fwrapv`

[19] https://godbolt.org/z/5x3K5KGnr

[20] https://godbolt.org/z/4r4a4EzMW

[21] https://godbolt.org/z/8WYMYYjdG

[22] https://godbolt.org/z/qvGd1nvv9

[23] https://godbolt.org/z/xfePeYsE3

[24] https://godbolt.org/z/GY1jvsrb5

[25] https://godbolt.org/z/4ronPsnxf

[26] Fermat 大定理

[27] https://godbolt.org/z/d834MK7bz

[28] https://godbolt.org/z/Eov9nsKqf

[29] https://clang.llvm.org/docs/AddressSanitizer.html

[30] https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[31] https://godbolt.org

[32] The LLVM Project Blog: What Every C Programmer Should Know About Undefined Behavior #1/3

[33] The LLVM Project Blog: What Every C Programmer Should Know About Undefined Behavior #2/3

[34] The LLVM Project Blog: What Every C Programmer Should Know About Undefined Behavior #3/3

4.5 C++ 与其他常用语言的区别

本文介绍 C++ 与其他常用语言的区别，重点介绍 C 与 C++ 之间重要的或者容易忽略的区别。尽管 C++ 几乎是 C 的超集，C/C++ 代码混用一般也没什么问题，但是了解 C/C++ 间比较重要的区别可以避免碰到一些奇怪的 bug。如果你是以 C 为主力语言的 OIer，那么本文也能让你更顺利地上手 C++。C++ 相比 C 增加的独特特性可以阅读 C++ 进阶 部分的教程。此外，本文也简要介绍了 Python, Java 和 C++ 的区别。

C 与 C++ 的区别

宏与模板

C++ 的模板在设计之初的一个用途就是用来替换宏定义。学会模板编程是从 C 迈向 C++ 的重要一步。模板不同于宏的文字替换，在编译时会得到更全面的编译器检查，便于编写更健全的代码。模板特性在 C++11 后支持了可变长度的模板参数表，可以用来替代 C 中的可变长度函数并保证类型安全。

指针与引用

C++ 中你仍然可以使用 C 风格的指针，但是对于变量传递而言，更推荐使用 C++ 的 引用 特性来实现类似的功能。由于引用指向的对象不能为空，因此可以避免一些空地址访问的问题。不过指针由于其灵活性，也仍然有其用武之地。值得一提的是，C 中的 `NULL` 空指针在 C++11 起有类型安全的替代品 `nullptr`。引用和指针之间可以通过 `*` 和 `&` 运算符 相互转换。

bool

与 C++ 不同的是，C 语言最初并没有 布尔类型。

C99 标准加入了 `_Bool` 关键字（以及等效的 `bool` 宏）以及 `true` 和 `false` 两个宏。如果需要使用 `bool`，`true`，`false` 这三个宏，需要在程序中引入 `stdbool.h` 头文件。而使用 `_Bool` 则不需要引入任何额外头文件。

```c
bool x = true;  // 需要引入 stdbool.h
_Bool x = 1;    // 不需要引入 stdbool.h
```

C23 起，`true` 和 `false` 成为 C 语言中的关键字，使用它们不需要再引入 `stdbool.h` 头文件[2]。

下表展示了 C 语言不同标准下，bool 类型支持的变化情况（作为对照，加入了 C++ 的支持情况）：

语言标准	`bool`	`true`/`false`	`_Bool`
C89	/	/	保留[3-1]
C99 起，C23 以前	宏，与 `_Bool` 等价，需要 `stdbool.h` 头文件	宏，`true` 与 `1` 等价，`false` 与 `0` 等价，需要 `stdbool.h` 头文件	关键字

语言标准	bool	true/false	_Bool
C23 起	宏，与 `_Bool` 等价，需要 `stdbool.h` 头文件	关键字	关键字
C++	关键字	关键字	保留[3-2]

struct

尽管在 C 和 C++ 中都有 struct 的概念，但是他们对应的东西是不能混用的！C 中的 struct 用来描述一种固定的内存组织结构，而 C++ 中的 struct 就是一种类，**它与类唯一的区别就是它的成员和继承行为默认是 public 的**，而一般类的默认成员是 private 的。这一点在写 C/C++ 混合代码时尤其致命。

另外，声明 struct 时 C++ 也不需要像 C 那么繁琐，C 版本：

```
typedef struct Node_t {
  struct Node_t *next;
  int key;
} Node;
```

C++ 版本

```
struct Node {
  Node *next;
  int key;
};
```

const

const 在 C 中只有限定变量不能修改的功能，而在 C++ 中，由于大量新特性的出现，const 也被赋予的更多用法。C 中的 const 在 C++ 中的继任者是 constexpr，而 C++ 中的 const 的用法请参见 常值 页面的说明。

内存分配

C++ 中新增了 `new` 和 `delete` 关键字用来在「自由存储区」上分配空间，这个自由存储区可以是堆也可以是静态存储区，他们是为了配合「类」而出现的。其中 `delete[]` 还能够直接释放动态数组的内存，非常方便。`new` 和 `delete` 关键字会调用类型的构造函数和析构函数，相比 C 中的 `malloc()`、`realloc()`、`free()` 函数，他们对类型有更完善的支持，但是效率不如 C 中的这些函数。

简而言之，如果你需要动态分配内存的对象是基础类型或他们的数组，那么你可以使用 `malloc()` 进行更高效的内存分配；但如果你新建的对象是非基础的类型，那么建议使用 `new` 以获得安全性检查。值得注意的是尽管 `new` 和 `malloc()` 都是返回指针，但是 `new` 出来的指针**只能**用 `delete` 回收，而 `malloc()` 出来的指针也只能用 `free()` 回收，否则会有内存泄漏的风险。

变量声明

C99 前，C 的变量声明必须位于语句块开头，C++ 和 C99 后无此限制。

可变长数组

C99 后 C 语言支持 VLA（可变长数组），C++ 始终不支持。

结构体初始化

C99 后 C 语言支持结构体的 指派符初始化[4]（但是在 C11 中为可选特性），C++ 直到 C++20 才支持有顺序要求的指派符初始化，且 C 语言支持的乱序、嵌套、与普通初始化器混用、数组的指派符初始化特性 C++ 都不支持[1]。

注释语法

C++ 风格单行注释 `//`，C 于 C99 前不支持。

Python 与 C++ 的区别

Python 是目前机器学习界最常用的语言。相比于 C++，Python 的优势在于易于学习，易于实践。Python 有着更加简单直接的语法，比如在定义变量时，不需要提前声明变量类型。但是，这样的简单也是有代价的。Python 相比于 C++ 牺牲了性能。C++ 几乎适用于包括嵌入式系统的所有平台，并且有着更快的执行速度，但是 Python 只可以在某些支持高级语言的平台上使用。C++ 更接近底层，所以可以用来进行编写操作系统。

Java 与 C++ 的区别

Java 与 C++ 都是面向对象的语言，都使用了面向对象的思想（封装、继承、多态），由于面向对象有许多非常好的特性（继承、组合等），因此二者有很好的可重用性。所以相比于 Python，Java 和 C++ 更加类似。

二者最大的区别在于 Java 有 JVM 的机制。JVM 全称是 Java Virtual Machine，中文意为 Java 虚拟机。Java 语言的一个非常重要的特点就是与平台的无关性。而使用 Java 虚拟机是实现这一特点的关键。一般的高级语言如果要在不同的平台上运行，至少需要编译成不同的目标代码。而引入 Java 语言虚拟机后，Java 语言在不同平台上运行时不需要重新编译。Java 语言使用 Java 虚拟机屏蔽了与具体平台相关的信息，使得 Java 语言编译程序只需生成在 Java 虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。

因为这个特点，Java 经常被用于需要移植到不同平台程序的开发。但是也由于编译 Java 程序时需要从字节码开始，所以 Java 的性能没有 C++ 好。

参考资料

[1] https://en.cppreference.com/w/cpp/language/aggregate_initialization

[2] https://en.cppreference.com/w/c/23。

[3] C 和 C++ 均规定，以一个下划线跟着一个大写字母开头的标识符是被保留的，详见 https://en.cppreference.com/w/c/language/identifier。[3-1] [3-2]

[4] 指派符初始化

4.6 Pascal 转 C++ 急救

Authors: kexploring, Ir1d, lvneg1

> 用来急救，不多废话。

" 药方食用提示"

本急救贴可以让您充分了解以下内容（对应 C++ 语法快速提要）:

- 基本语法（块语句、注释、导入库、简单输入输出、声明变量、赋值……）
- C++ 的 Hello World 与 A+B Problem 写法与解释

  对应语法部分较为紧凑，正式食用可能需要额外参考资料（已给出）。此部分不包括指针与 C 风格数组的介绍，也没有结构体、运算符重载等等。

  重要不同之处部分为 C++ 的语法特点，也是 Pascal 转 C++ 时会碰到的坑。

如要快速查找，请见附录：

- 附 A：Pascal 与 C++ 运算符与数学函数语法对比表
- 附 B：文章检索 - 按 C++ 语句语法索引

C++ 快速安装与环境配置

> 注意：这里假设使用的系统是 Windows。

方式一：使用 IDE

以下 IDE 选择一个即可：

- Dev-C++
- Code::Blocks[1]

方式二：使用代码编辑器 + 编译器 + 调试器

- VS Code

  Visual Studio Code 官方网站上有文档解释如何进行 C++ 的配置。一般而言 VS Code 搭配插件使用更方便，见 VS Code 的官方网站[2]。

C++ 语法快速提要 Start Here

C++ 程序都是从 `main` 这个部分开始运行的。

大括号表示块语句的开始与结束：`{` 就相当于 Pascal 里面的 `begin`，而 `}` 就相当于 `end`。

注意，和 Pascal 一样，C++ 每句话结束要加分号 `;`，不过大括号结尾不需要有分号，而且程序结束末尾不用打句号 `.`。

`//` 表示行内注释，`/* */` 表示块注释。

按照惯例，看看 Hello World 吧。

Hello World：第一个 C++ 程序

```cpp
#include <iostream>  // 导入 iostream 库

int main()  // main 部分
{
  std::cout << "Hello World!" << std::endl;

  return 0;
}
```

然后编译运行一下，看看结果。

简要解释

第一行，`#include <iostream>` 的意思是，导入 `iostream` 这个库。

> **"Pascal 的库文件"**
>
> Pascal 其实是有库文件的，只不过，很多同学从来都没有用过⋯⋯

看到第三行的 `main` 吗？程序从 `main` 开始执行。

接下来最重要的一句话是

```
std::cout << "Hello World!" << std::endl;
```

`std::cout` 是输出（`cout` 即 C-out）的命令。你可能看过有些 C++ 程序中直接写的是 `cout` 。

> **" 有关 std:: 前缀"**
>
> 有关 `std::` 这个前缀的问题，请见这节底下的注释「什么是 std?」。

中间的 `<<` 很形象地表示流动，其实它就是表示输出怎么「流动」的。这句代码的意思就是，`"Hello World!"` 会先被推到输出流，之后 `std::endl` 再被推到输出流。

而 `std::endl` 是**输出**换行（`endl` 即 end-line）命令，这与 Pascal 的 `writeln` 类似，不过 C++ 里面可没有 `coutln` 。Pascal 与 C++ 的区别在于，`write('Hello World!')` 等价于 `std::cout << "Hello World!"` ，而 `writeln('Hello World!')` 等价于 `std::cout << "Hello World!" << std::endl` 。

此处 `"Hello World!"` 是字符串，Pascal 中字符串都是用单引号 `'` 不能用双引号，而 C++ 的字符串必须用双引号。C++ 中单引号包围的字符会有别的含义，后面会再提及的。

好了，到这里 Hello World 应该解释的差不多了。

可能有同学会问，后面那个 `return 0` 是什么意思？那个 `int main()` 是啥意思？**先别管它**，一开始写程序的时候先把它当作模板来写吧（这里也是用模板写的）。因为入门时并不会用到 `main` 中参数，所以不需要写成 `int main(int argc, char const *argv[])` 。

简单练习

1. 试着换个字符串输出。
2. 试着了解转义字符。

A+B Problem：第二个 C++ 程序

经典的 A+B Problem。

```cpp
#include <iostream>

int main() {
  int a, b, c;

  std::cin >> a >> b;

  c = a + b;

  std::cout << c << std::endl;

  return 0;
}
```

注：代码空行较多，若不习惯可去掉空行。

简要解释

`std::cin` 是读入（`cin` 即 C-in），`>>` 也与输出语法的类似。

这里多出来的语句中最重要的是两个，一个是变量声明语句

```
int a, b, c;
```

你可能习惯于 Pascal 里面的声明变量

```
var
a, b, c: integer;
```

C++ 的声明是直接以数据类型名开头的，在这里，`int` （整型）开头表示接下来要声明变量。

接着一个最重要的语句就是赋值语句

```
c = a + b;
```

这是 Pascal 与 C++ 语法较大的不同，**这是个坑**：Pascal 是 `:=` ，C++ 是 `=` ；而 C++ 判断相等是 `==` 。

C++ 也可直接在声明时进行变量初始化赋值

```
int a = 0, b = 0, c = 0;
```

简单练习

1. 重写一遍代码，提交到 OJ 上，并且 AC。
2. 更多的输入输出语法参考这节内容 ，并试着了解 C++ 的格式化输出。

结束语与下一步

好了，到现在为止，你已经掌握了一些最基本的东西了，剩下就是找 Pascal 和 C++ 里面对应的语法和不同的特征。

不过在此之前，强烈建议先看变量作用域： 全局变量与局部变量 ，也可使用附 B： 文章检索查阅阅读。

请善用 Alt+ ←与 Alt+ →返回跳转。

对应语法 Syntax

变量 Variable

基本数据类型 Fundamental types

C++ 与 Pascal 基本上差不多，常见的有

- `bool` Boolean 类型
- `int` 整型
- 浮点型
  - `float`
  - `double`
- `char` 字符型
- `void` 无类型

C++ 的单引号是专门用于表示单个字符的（字符型），比如 `'a'` ，而字符串（字符型数组）必须要用双引号。

C++ 还要很多额外的数据类型，请参考更多资料。

扩展阅读：

- 基础类型 - cppreference.com[3]

常量声明 Constant

```
const double PI = 3.1415926;
```

若不清楚有关宏展开的问题，建议使用常量，而不用宏定义。

运算符 Operator

请直接参考

- 附 A：Pascal 与 C++ 运算符与数学函数语法对比表
- 运算 - OI Wiki

条件

if 语句

```
if (a = b) and (a > 0) and (b > 0) then
    begin
        b := a;
    end
else
    begin
        a := b;
    end;
```

```
if (a == b && a > 0 && b > 0) {
  b = a;
} else {
  a = b;
}
```

布尔运算与比较

- `and` -> `&&`
- `or` -> `||`
- `not` -> `!`
- `=` -> `==`
- `<>` -> `!=`

注释：

1. Pascal 中 `and` 与 C++ 中 `&&` 优先级不同，C++ 不需要给判断条件加括号。
2. Pascal 中判断相等是 `=` ，赋值是 `:=` ；C++ 中判断相等是 `==` ，赋值是 `=` 。
3. 如果在 `if` 语句的括号内写了 `a = b` 而不是 `a == b` ，程序不会报错，而且会把 `b` 赋值给 `a` ，`a = b` 这个语句的返回结果是 `true` 。
4. C++ 不需要思考到底要不要在 `end` 后面加分号。
5. C++ 布尔运算中，非布尔值可以自动转化为布尔值。

" 易错提醒"

特别注意：**不要把 `==` 写成 `=`！**

由于 C/C++ 比 Pascal 语法灵活，如果在判断语句中写了 `if (a=b) {`，那么程序会顺利运行下去，因为 C++ 中 `a=b` 是有返回值的。

case 与 switch

用到得不多，此处不详细展开。

需要注意：C++ 没有 `1..n`，也没有连续不等式（比如 `1 < x < 2`）。

循环 Loop

以下三种循环、六份代码实现的功能是一样的。

while 循环

`while` 很相似。（C++ 此处并非完整程序，省略一些框架模板，后同）

```pascal
var i: integer;

begin
    i := 1;
    while i <= 10 do
        begin
            write(i,' ');
            inc(i); // 或者 i := i + 1;
        end;
end.
```

```cpp
int i = 1;
while (i <= 10) {
  std::cout << i << " ";
  i++;
}
```

for 循环

C++ 的 `for` 语句非常不同。

```pascal
var i: integer;

begin
    for i:= 1 to 10 do
        begin
            write(i, ' ');
        end;
end.
```

```cpp
for (int i = 1; i <= 10; i++) {
  std::cout << i << " ";
}
```

注释：

1. `for (int i = 1; i <= 10; i++){` 这一行语句很多，`for` 中有三个语句。

2. 第一个语句 `int i = 1;` 此时声明一局部变量 `i` 并初始化。(这个设计比 Pascal 要合理得多。)
3. 第二个语句 `i <= 10;` 作为判断循环是否继续的标准。
4. 第三个语句 `i++`，在每次循环结尾执行，意思大约就是 Pascal 中的 `inc(i)`，此处写成 `++i` 也是一样的。`i++` 与 `++i` 的区别请参考其他资料。

repeat until 与 do while 循环

注意，`repeat until` 与 `do while` 是不同的，请对比以下代码

```pascal
var i: integer;

begin
    i := 1;
    repeat
        write(i, ' ');
        inc(i);
    until i = 11;
end.
```

```cpp
int i = 1;
do {
  std::cout << i << " ";
  i++;
} while (i <= 10);
```

循环控制 Loop Control

C++ 中 `break` 的作用与 Pascal 是一样的，退出循环。

而 `continue` 也是一样的，跳过当前循环，进入下一次循环（回到开头）。

数组与字符串 Array and String

不定长数组：标准库类型 Vector

C++ 标准库中提供了 `vector`，相当于不定长数组，调用前需导入库文件。

```cpp
#include <iostream>
#include <vector>  // 导入 vector 库

int main() {
  std::vector<int> a;  // 声明 vector a 并定义 a 为空 vector 对象
  int n;

  std::cin >> n;
  // 读取 a
  for (int i = 0; i < n; i++) {
    int t;
    std::cin >> t;
    a.push_back(t);  // 将读入的数字 t，放到 vector a 的末尾；该操作复杂度 O(1)
    /* 这里不能使用下标访问来赋值，因为声明时，a 大小依然为空，
    此处使用 `a[i] = t;` 是错误做法。
    */
  }

  // 将读入到 a 中的所有数打印出
```

```
  for (int i = 0; i < n; i++) {
    std::cout << a[i] << ", ";  // ! 注意，a 中第一个数是 a[0];
    // 如果下标越界，它会返回一个未知的值（溢出），而不会报错
  }
  std::cout << std::endl;

  return 0;
}
```

C++ 访问数组成员，与 Pascal 类似，不过有很重要的区别：数组的第一项是 `a[0]`，而 Pascal 中是可以自行指定的。

扩展阅读：

- 序列式容器 - OI Wiki

字符串：标准库类型 String

C++ 标准库中提供了 `string`，与 `vector` 可以进行的操作有些相同，同样需要导入库文件。

```
#include <iostream>
#include <string>

int main() {
  std::string s;  // 声明 string s

  std::cin >> s;  // 读入 s;
  // 读入时会忽略开头所有空格符（空格、换行符、制表符），读入的字串直到下一个空格符为
止。

  std::cout << s << std::endl;

  return 0;
}
```

扩展阅读：

- string - OI Wiki

C 风格数组 Array

如果要用不定长的数组请用 Vector，不要用 C 风格的数组。

C 风格的数组与指针有密切关系，所以此处不多展开。

扩展阅读：

- 数组 - OI Wiki

重要不同之处 Differences

变量作用域 Scope：全局变量与局部变量

C++ 几乎可以在**任何地方**声明变量。

以下对于 C++ 的变量作用域的介绍摘自变量作用域 - OI Wiki：

> 作用域是变量可以发挥作用的代码块。
>
> 变量分为全局变量和局部变量。在所有函数外部声明的变量，称为全局变量。在函数或一个代码块内部声

明的变量，称为局部变量。

全局变量的作用域是整个文件，全局变量一旦声明，在整个程序中都是可用的。

局部变量的作用域是声明语句所在的代码块，局部变量只能被函数内部或者代码块内部的语句使用。

由一对大括号括起来的若干语句构成一个代码块。

```
int g = 20;  // 声明全局变量
int main() {
  int g = 10;          // 声明局部变量
  printf("%d\n", g);  // 输出 g
  return 0;
}
```

在一个代码块中，局部变量会覆盖掉同名的全局变量，比如上面的代码输出的 `g` 就是 `10` 而不是 `20` 。为了防止出现意料之外的错误，请尽量避免局部变量与全局变量重名的情况。

在写 Pascal 过程/函数时，容易忘记声明局部变量 `i` 或者 `j` ，而一般主程序里会有循环，于是大部分情况下 `i` 与 `j` 都是全局变量，于是，在这种情况下，过程/函数中对 `i` 操作极易出错。更要命的是，如果忘记声明这种局部变量，编译器编译不报错，程序可以运行。（有很多难找的 bug 就是这么来的。）

所以，在使用 C++ 时，声明变量，比如循环中使用的 `i` ，**不要用全局变量，能用局部变量就用局部变量** 。如果这么做，不用担心函数中变量名（比如 `i` ）冲突。

" 额外注"

Pascal 可在某种程度上避免这个问题，仿照 C++ 的方法，主程序只有调用过程/函数，不声明 `i` `j` 这类极易名称冲突的全局变量，如果需要循环，另写一个过程进行调用。

C++ 可以自动转换类型

```
int i = 2;
if (i) {  // i = 0 会返回 false，其余返回 true
  std::cout << "true";
} else {
  std::cout << "false";
}
```

不光是 `int` 转成 `bool` ，还有 `int` 与 `float` 相互转换。在 Pascal 中可以把整型赋给浮点型，但不能反过来。C++ 没有这个问题。

```
int a;
a = 3.2;       // 此时 a = 3
float b = a;  // 此时 b = 3.0
```

区分 `/` 是整除还是浮点除法，是通过除数与被除数的类型判断的

```
float a = 32 / 10;     // 32/10 的结果是 3 (整除); a = 3.0
float b = 32.0 / 10;  // 32.0/10 的结果是 3.2; b = 3.2
```

`pow(a, b)` 计算 a^b ，该函数返回的是浮点型，如果直接用来计算整数的幂，由于有自动转换，不需要担心它会报错

```
int a = pow(2, 3);  // 计算 2^3
```

还有 `char` 与 `int` 之间相互转换。

```cpp
char a = 48;              // ASCII 48 是 '0'
int b = a + 1;            // b = 49
std::cout << (a == '0');  // true 输出 1
```

其实 C++ 中的 `char` 与 `bool` 本质上是整型。

扩展阅读：

- 隐式转换 - cppreference.com[4] 注意内容可能过于专业

C++ 很多语句有返回值：以如何实现读取数量不定数据为例

有些时候需要读取到数据结束，比如，求一组不定数量的数之和（数据可以多行），直到文件末尾，实现方式是

" 文件末尾 EOF"

EOF，文件末尾标识符，在命令行中 Windows 上以 Ctrl+Z 输入（还需按 Enter），*unix 系统以 Ctrl+D 输入。

```cpp
#include <iostream>

int main() {
  int sum = 0, a = 0;

  while (std::cin >> a) {
    sum += a;
  }
  std::cout << sum << std::endl;

  return 0;
}
```

实现原理：`while (std::cin >> a)` 中 `std::cin >> a` 若在输入有问题或遇到文件结尾时，会返回 false，使得循环中断。

函数 Function：C++ 只有函数没有过程但有 `void`，没有函数值变量但有 `return`。

Pascal 函数与 C++ 函数对比示例：

```pascal
function abs(x:integer):integer;
begin
    if x < 0 then
        begin
            abs := -x;
        end
    else
        begin
            abs := x;
        end;
end;
```

```cpp
int abs(int x) {
  if (x < 0) {
    return -x;
```

```
  } else {
    return x;
  }
}
```

C++ 中函数声明 `int abs` ，就定义了 `abs()` 函数且返回值为 `int` 型（整型），函数的返回值就是 `return` 语句给出的值。

如果不想有返回值（即 Pascal 的「过程」），就用 `void` 。`void` 即「空」，什么都不返回。

```
var ans: integer;

procedure printAns(ans:integer);
begin
    writeln(ans);
end;

begin
    ans := 10;
    printAns(ans);
end.
```

```
#include <iostream>

void printAns(int ans) {
  std::cout << ans << std::endl;

  return;
}

int main() {
  int ans = 10;
  printAns(ans);

  return 0;
}
```

C++ 的 `return` 与 Pascal 中给函数变量赋值有一点非常大的不同。C++ 的 `return` 即返回一个值，执行完这个语句，函数就执行结束了；而 Pascal 中给函数变量赋值并不会跳出函数本身，而是继续执行。于是，如果 Pascal 需要某处中断函数/过程，就需要一个额外的命令，即 `exit` 。而 C++ 则不需要，如果需要在某处中断，可以直接使用 `return` 。比如（由于实在想不出来简短且实用的代码，所以就先这样）

```
#include <iostream>

void printWarning(int x) {
  if (x >= 0) {
    return;  // 该语句在此处相当于 Pascal 中的 `exit;`
  }
  std::cout << "Warning: input a negative number.";
}

int main() {
  int a;

  std::cin >> a;
  printWarning(a);
```

```
  return 0;
}
```

而在某种意义上，前面的 abs 函数，这样才是严格等效的

```
function abs(x:integer):integer;
begin
    if x < 0 then
        begin
            abs := -x; exit; // ! 注意此处
        end
    else
        begin
            abs := x;  exit; // ! 注意此处
        end;
end;
```

```
int abs(int x) {
  if (x < 0) {
    return -x;
  } else {
    return x;
  }
}
```

" 特别提醒"

C++ 中 exit 是退出程序；不要顺手把 exit 打上去，要用 return！

C++ 把函数和过程统统视作函数，连 main 都不放过，比如写 int main ，C++ 视 main 为一个整型的函数，这里返回值是 0 。它是一种习惯约定，返回 0 代表程序正常退出。

也许你已经猜到了，main(int argc, char const *argv[]) 中的参数就是 int argc 与 char const *argv[] ，不过意义请参考其他资料。

在函数中传递参数 Passing Parameters to Functions

C++ 中没有 Pascal 的 var 关键字可以改变传递的参数，但是 C++ 可以使用引用和指针达到同样的效果。

```
var a, b: integer;

procedure swap(var x,y:integer);
var temp:integer;
begin
    temp := x;
    x := y;
    y := temp;
end;

begin
    a := 10; b:= 20;
    swap(a, b);
    writeln(a, ' ', b);
end.
```

```cpp
// 使用指针的代码
#include <iostream>

void swap(int* x, int* y) {
  int temp;
  temp = *x;
  *x = *y;
  *y = temp;
}

int main() {
  int a = 10, b = 20;
  swap(&a, &b);
  std::cout << a << " " << b;

  return 0;
}
```

注意，此处 C++ 代码**涉及指针问题** 。指针问题还是很麻烦的，建议去阅读相关资料。

```cpp
// 使用引用的代码
#include <iostream>

void swap(int& x, int& y) {
  int temp;
  temp = x;
  x = y;
  y = temp;
}

int main(int argc, char const* argv[]) {
  int a = 10, b = 20;
  swap(a, b);
  std::cout << a << " " << b;

  return 0;
}
```

注意，此处 C++ 代码涉及**引用相关类型问题** 。在用引用调用一些 STL 库、模板库的时候可能会遇到一些问题，这时候需要手动声明别类型。具体资料可以在《C++ Primer》第五版或者网络资料中自行查阅。

C++ 中函数传递参数还有其他方法，其中一种是**直接使用全局变量传递参数** ，如果不会用指针，可以先用这种方法。但是这种方法的缺陷是没有栈保存数据，**没有办法在递归函数中传参** 。（除非手写栈，注意，手写栈也是一种突破系统栈限制的方法。）

C++ 标准库与参考资料 Reference

千万不要重复造轮子（除非为了练习），想要自己动手写一个功能出来之前，先去看看有没有这个函数或者数据结构。

C++ 标准库

C++ 标准库中 `<algorithm>` 有很多有用的函数比如快排、二分查找等，可以直接调用。请参考这个页面：STL 算法 - OI Wiki 。

还有 STL 容器，比如数组、向量（可变大小的数组）、队列、栈等，附带很多函数。请参考这个页面：STL 容器简介 - OI Wiki 。

如果要找关于字符串操作的函数见

- std::basic_string - cppreference.com[5]
- `<string>` - C++ Reference[6]

C/C++ 的指针是很灵活的东西，如果想要彻底理解指针，建议找本书或者参考手册仔细阅读。

- 指针 - OI Wiki

错误排查与技巧

- 常见错误 - OI Wiki
- 常见技巧 - OI Wiki

C++ 语言资料

- 学习资源 - OI Wiki
- cppreference.com[7] - 最重要的 C/C++ 参考资料
- C++ 教程 - 菜鸟教程[8]
- C++ Language - C++ Tutorials[9]
- Reference - C++ Reference[10]
- C++ Standard Library - Wikipedia[11]
- The Ultimate Question of Programming, Refactoring, and Everything[12]
- Google C++ Style Guide[13]

后记

写到这里，很多同学会觉得这一点都不急救啊，有很多东西没有提到啊。那也是没办法的事情。

虽然是为了急救，但很多东西像怎么把字符串转化为数字，怎么搜索字符串中的字符，这些东西也不适合一篇精悍短小的急救帖，如果把这些都写出来，那就是 C++ 入门教程，所以请充分利用本 Wiki、参考手册与搜索引擎。

需要指出的一点是，上面说 C++ 的语法，其实有很多语法是从 C 语言来的，标题这么写比较好——《Pascal 转 C/C++ 急救帖》。

Pascal 在上个世纪后半叶是门很流行的语言，它早于 C 语言，不过随着 UNIX 系统的普及，微软使用 C 语言，现在 Pascal 已经成为历史了。Pascal 后期发展也是有的，比如 Free Pascal 这个开源编译器项目，增加面向对象的特性（Delphi 语言）。Pascal 目前的用处除了在信息竞赛外，有一个特点是其他语言没有的——编译支持非常非常多老旧机器，比如 Gameboy 这种上个世纪的任天堂游戏机，还有一个用处就是以伪代码的形式（Pascal 风格的伪代码）出现在各种教科书中。

最后，Pascal 的圈子其实很小，C/C++ 的圈子很大，帮助手册与教程很多很全，一定要掌握好英语。世界上还有很多很多编程语言，而计算机这门学科与技术不光是信息竞赛和编程语言。

本文 Pascal 语言的参考文献

- Lazarus wiki[14]
- Free Pascal Reference guide[15]

附 A：Pascal 与 C++ 运算符与数学函数语法对比表 Pascal vs C++ Operator Syntax Table

> 仅包括最常用的运算符与函数。

基本算术

	Pascal	C++
加法	`a + b`	`a + b`
减法	`a - b`	`a - b`
乘法	`a * b`	`a * b`
整除	`a div b`	`a / b`
浮点除法	`a / b`	`a / b`
取模	`a mod b`	`a % b`

逻辑

	Pascal	C++
非	`not(a)`	`!a`
且	`a and b`	`a && b`
或	`a or b`	‘a

比较

	Pascal	C++
相等	`a = b`	`a == b`
不等	`a <> b`	`a != b`
大于	`a > b`	`a > b`
小于	`a < b`	`a < b`
大于等于	`a >= b`	`a >= b`
小于等于	`a <= b`	`a <= b`

赋值

Pascal	C++
`a := b`	`a = b`
`a := a + b`	`a += b`
`a := a - b`	`a -= b`
`a := a * b`	`a *= b`
`a := a div b` 或 `a := a / b`	`a /= b`
`a := a mod b`	`a %= b`

自增/自减

	Pascal	C++
自增	`inc(a)`	`a++`
自增	`inc(a)`	`++a`
自减	`dec(a)`	`a--`
自减	`dec(a)`	`--a`

数学函数

> 使用需要导入 `<cmath>` 库。

	Pascal	C++
绝对值	`abs(a)`	`abs(a)` （整数）
绝对值	`abs(a)`	`fabs(a)` （浮点数）
a^b	N/A (*)	`pow(a, b)`
截断取整	`trunc(a)`	`trunc(a)`
近似取整	`round(a)`	`round(a)`

*Extended Pascal 中有 `a**b` 不过需要导入 `Math` 库。

其他函数请参考：

- 常用数学函数 - cppreference.com[16]

附 B：文章检索 Index

> 按 C++ 语句语法索引。

- 基本语法
- 变量 - 数据类型 - 常量声明 - 作用域

- 运算符
- if 语句 - if - else
- 循环语句 - for 语句 - while 语句 - do while 语句 - break, continue
- 函数 - 函数定义 ，return - 函数传参
- 数组与字符串 - 不定长数组 Vector - C 风格数组 - 字符串 String
- 资料

参考资料与注释

[1] Code::Blocks

[2] VS Code 的官方网站

[3] 基础类型 - cppreference.com

[4] 隐式转换 - cppreference.com

[5] std::basic_string - cppreference.com

[6] `<string>` - C++ Reference

[7] cppreference.com

[8] C++ 教程 - 菜鸟教程

[9] C++ Language - C++ Tutorials

[10] Reference - C++ Reference

[11] C++ Standard Library - Wikipedia

[12] The Ultimate Question of Programming, Refactoring, and Everything

[13] Google C++ Style Guide

[14] Lazarus wiki

[15] Free Pascal Reference guide

[16] 常用数学函数 - cppreference.com

4.7 Python 速成

关于 Python

Python 是一门已在世界上广泛使用的解释型语言。它提供了高效的高级数据结构，还能简单有效地面向对象编程，也可以在算法竞赛。

Python 的优点

- Python 是一门**解释型**语言：Python 不需要编译和链接，可以在一定程度上减少操作步骤。
- Python 是一门**交互式**语言：Python 解释器实现了交互式操作，可以直接在终端输入并执行指令。
- Python **易学易用**：Python 提供了大量的数据结构，也支持开发大型程序。
- Python **兼容性强**：Python 同时支持 Windows、macOS 和 Unix 操作系统。
- Python **实用性强**：从简单的输入输出到科学计算甚至于大型 WEB 应用，都可以写出适合的 Python 程序。
- Python **程序简洁、易读**：Python 代码通常比实现同种功能的其他语言的代码短。
- Python **支持拓展**：Python 会开发 C 语言程序（即 CPython），支持把 Python 解释器和用 C 语言开发的应用链接，用 Python 扩展和控制该应用。

学习 Python 的注意事项

- 目前主要使用的 Python 版本是 Python 3.7 及以上的版本，Python 2 和 Python 3.6 及以前的 Python 3 已经不被支持[3]，但仍被一些老旧系统与代码所使用。本文将**介绍较新版本的 Python**。如果遇到 Python 2 代码，可以尝试 `2to3`[4] 程序将 Python 2 代码转换为 Python 3 代码。
- Python 的设计理念和语法结构**与一些其他语言的差异较大**，隐藏了许多底层细节，所以呈现出实用而优雅的风格。
- Python 是高度动态的解释型语言，因此其**程序运行速度相对较慢**，尤其在使用其内置的 `for` 循环语句时。在使用 Python 时，应尽量使用 `filter`、`map` 等内置函数，或使用 列表生成[5] 语法的手段来提高程序性能。

环境搭建

参见 Python 3。或者：

- Windows：也可以在 Microsoft Store 中免费而快捷地获取 Python。
- macOS/Linux：通常情况下，大部分的 Linux 发行版中已经自带了 Python。如果只打算学习 Python 语法，并无其它开发需求，不必另外安装 Python。

> " 注意"
>
> 在一些默认安装（指使用软件包管理器安装）Python 的系统（如 Unix 系统）中，应在终端中运行 `python3` 打开 Python 3 解释器。[1]

此外，也可以通过 venv、conda、Nix 等工具管理 Python 工具链和 Python 软件包，创建隔离的虚拟环境，避免出现依赖问题。

作为一种解释型语言，Python 的执行方式和 C++ 有所不同，这种差异在使用 IDE 编程时往往得不到体现，因此这里需要强调一下运行程序的不同方式。

当在命令行中键入 `python3` 或刚刚打开 IDLE 时，你实际进入了一种交互式的编程环境，也称「REPL」（「读取 - 求值 - 输出」循环），初学者可以在这里输入语句并立即看到结果，这让验证一些语法变得极为容易，我们也将在后文中大量使用这种形式。

但若要编写完整的程序，你最好还是新建一个文本文件（通常后缀为 `.py`），然后在命令行中执行 `python3` filename.py，就能够运行代码看到结果了。

通过镜像下载安装文件

目前国内关于**源码**的镜像缓存主要是 北京交通大学自由与开源软件镜像站[6] 和 华为开源镜像站[7]，可以到那里尝试下载 Python 安装文件。

使用 pip 安装第三方库

Python 的生命力很大程度上来自于丰富的第三方库，编写一些实用程序时「调库」是常规操作，pip 是首选的安装第三方库的程序。自 Python 3.4 版本起，它被默认包含在 Python 二进制安装程序中。

pip 中的第三方库主要存储在 Python 包索引（PyPI）[8] 上，用户也可以指定其它第三方库的托管平台。使用方法可参照 pypi 镜像使用帮助 - 清华大学开源软件镜像站[9] 等使用帮助。你可以在 MirrorZ[10] 上获取更多 PyPI 镜像源。

基本语法

Python 的语法简洁而易懂，也有许多官方和第三方文档与教程。这里仅介绍一些对 OIer 比较实用的语言特性，你可以在 Python 文档[11] 和 Python Wiki[12] 等网页上了解更多关于 Python 的教程。

注释

加入注释并不会对代码的运行产生影响，但加入注释可以使代码更加易懂易用。

```
# 用 # 字符开头的是单行注释

"""
跨多行字符串会用三引号
（即三个单引号或三个双引号）
包裹，但也通常被用于注释
"""
```

加入注释代码并不会对代码产生影响。我们鼓励加入注释来使代码更加易懂易用。

基本数据类型

一切皆对象

在 Python 中，你无需事先声明变量名及其类型，直接赋值即可创建各种类型的变量：

```
>>> x = -3  # 语句结尾不用加分号
>>> x
-3
>>> f = 3.1415926535897932384626; f  # 实在想加分号也可以，这里节省了一行
3.141592653589793
>>> s1 = "O"
>>> s1  # 怎么显示成单引号了？有区别吗？
'O'
>>> b = 'A' == 65  # 明明在 C/C++ 中是成立的
>>> b  # 与众不同的是 True, False 首字母均大写，可能与内置常量的命名约定有关
False
>>> True + 1 == 2 and not False != 0  # Python 可能喜欢单词胜过符号
True
```

但这不代表 Python 没有类型的概念，实际上解释器会根据赋值或运算自动推断变量类型，你可以使用内置函数 `type()` 查看这些变量的类型：

```
>>> type(x)
<class 'int'>
>>> type(f)
<class 'float'>
>>> type(s1)  # 请注意，不要给字符串起名为 str，不信试试看是否真的可以这么做
<class 'str'>
>>> type(b)
<class 'bool'>
```

"内置函数[13] 是什么？"

在 C/C++ 中，很多常用函数都分散在不同的头文件中，但 Python 的解释器内置了许多实用且通用的函数，你可以直接使用而无需注意它们的存在，但这也带来了小问题，这些内置函数的名称多为常见单词，你需要注意避免给自己的变量起相同的名字，否则可能会产生奇怪的结果。

正如我们所看到的，Python 内置有整数、浮点数、字符串和布尔类型，可以类比为 C++ 中的 `int`，`float`，`string` 和 `bool`。但有一些明显的不同之处，比如没有 `char` 字符类型，也没有 `double` 类型（但 `float` 其实对应 C 中的双精度），如果需要更精确的浮点运算，可以使用标准库中的 decimal[14] 模块，如果需要用到复数，Python 还内置了 `complex` 类型（而这也意味着最好不要给变量起名为 `complex`）。可以看到这些类型都以 `class` 开头，而这正是 Python 不同于 C++ 的关键之处，Python 程序中的所有数据都是由对象或对象间关系来表示的，函数是对象，类型本身也是对象：

```
>>> type(int)
<class 'type'>
>>> type(pow)  # 求幂次的内置函数，后文会介绍
<class 'builtin_function_or_method'>
>>> type(type)  # type() 也是内置函数，但有些特殊，感兴趣可自行查阅
<class 'type'>
```

你或许会觉得这些概念一时难以理解且没有用处，所以我们暂时不再深入，在后文的示例中你或许能慢慢体会到，Python 的对象提供了强大的方法，我们在编程时应当优先考虑围绕对象而不是过程进行操作，这会让我们的代码显得更加紧凑明晰。

数字运算

有人说，你可以把你系统里装的 Python 当作一个多用计算器，这是事实。

在交互模式下，你可以在提示符 `>>>` 后面输入一个表达式，就像其他大部分语言（如 C++）一样使用运算符 `+`、`-`、`*`、`/`、`%` 来对数字进行运算，也可以使用 `()` 来进行符合结合律的分组，读者可以自行试验，在这里我们仅展示与 C++ 差异较大的部分：

```
>>> 5.0 * 6  # 浮点数的运算结果是浮点数
30.0
>>> 15 / 3  # 与 C/C++ 不同，除法永远返回浮点 float 类型
5.0
>>> 5 / 100000  # 位数太多，结果显示成科学计数法形式
5e-05
>>> 5 // 3 # 使用整数除法（地板除）则会向下取整，输出整数类型
1
>>> -5 // 3 # 符合向下取整原则，注意这与 C/C++ 不同
-2
>>> 5 % 3 # 取模
2
```

```
>>> -5 % 3 # 负数取模结果一定是非负数，这点也与 C/C++ 不同，不过都满足 (a//b)*b+(a
%b)==a
1
>>> x = abs(-1e4)  # 求绝对值的内置函数
>>> x += 1  # 没有自增/自减运算符
>>> x  # 科学计数法默认为 float
10001.0
```

在上面的实践中可以发现，除法运算（/）永远返回浮点类型（在 Python 2 中返回整数）。如果你想要整数或向下取整的结果的话，可以使用整数除法（//）。同样的，你也可以像 C++ 中一样，使用模（%）来计算余数，科学计数法的形式也相同。

特别地，Python 用 `**` 即可进行幂运算，还通过内置的 `pow(a, b, mod)` 提供了 快速幂 的高效实现。

Python 的字符串类型包含 Unicode 字符，这意味着任何字符串都会存储为 Unicode。[2] 在 Python 中，可以对一个 Unicode 字符使用内置函数 `ord()` 将其转换为对应的 Unicode 编码，逆向的转换使用内置函数 `chr()`。

如果想把数转换为对应的字符串，可使用 Python 内置函数 `str()`，也可以使用 f-string 实现；反之，可以使用 `int()` 和 `float()` 两个函数。

Python 的字符串类型还有 许多方便的功能[15]。由于本文篇幅有限，这里不一一介绍。

数据类型判断

对于一个变量，可以使用 `type(object)` 返回变量的类型，例如 `type(8)` 和 `type('a')` 的值分别为 `<class 'int'>` 和 `<class 'str'>`。

输出和输入

输出

对于一个变量，可以使用 `type(object)` 返回变量的类型，例如 `type(8)` 和 `type('a')` 的值分别为 `<class 'int'>` 和 `<class 'str'>`。

Python 中，还可以使用 `**` 运算符和内置的 `pow(base, exp, mod=None)` 函数进行幂运算，使用 `abs(x)` 求数的绝对值。

```
>>> 3 ** 4 # 幂运算
81
>>> 2 ** 512
13407807929942597099574024998205846127479365820592393377723561443721764030073546
976801874298166903427690031858186486050853753882811946569946433649006084096
>>> pow(2, 512, int(1e4)) # 即 2**512 % 10000 的快速实现，1e4 是 float 所以要转
int
4096
>>> 2048 ** 2048 # 在 IDLE 里试试大整数？
>>> 0.1 + 0.1 + 0.1 - 0.3 == 0.  # 和 C/C++ 一样需要注意浮点数不能直接判相等
False
```

字符串

Python 3 提供了强大的基于 Unicode[16] 的字符串类型，使用起来和 C++ 中的 string 类似，一些概念如转义字符也都相通，除了加号拼接和索引访问，还额外支持数乘 `*` 重复字符串，和 `in` 操作符。

```
>>> s1 = "O"  # 单引号和双引号都能包起字符串，有时可节省转义字符
>>> s1 += 'I-Wiki'  # 为和 C++ 同步建议使用双引号
>>> 'OI' in s1  # 检测子串很方便
True
>>> len(s1)  # 类似 C++ 的 s.length()，但更通用
```

```
7
>>> s2 = """ 感谢你的阅读
... 欢迎参与贡献!
"""  # 使用三重引号的字符串可以跨越多行
>>> s1 + s2
'OI-Wiki 感谢你的阅读\n 欢迎参与贡献!'
>>> print(s1 + s2)  # 这里使用了 print() 函数打印字符串
OI-Wiki 感谢你的阅读
欢迎参与贡献!
>>> s2[2] * 2 + s2[3] + s2[-1]  # 负数索引从右开始计数，加上 len(s)，相当于模 n 的剩余
类环
'谢谢你!'
>>> s1[0] = 'o'  # str 是不可变类型，不能原地修改，其实 += 也是创建了新的对象
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Python 支持多种复合数据类型，可将不同值组合在一起。最常用的 `list`，类型是用方括号标注、逗号分隔的一组值。例如，`[1, 2, 3]` 和 `['a','b','c']` 都是列表。

除了索引，字符串还支持切片，它的设计非常精妙又符合直觉，格式为 `s[ 左闭索引 : 右开索引 : 步长 ]`：

```
>>> s = 'OI-Wiki 感谢你的阅读\n欢迎参与贡献!'
>>> s[:8]  # 省略左闭索引则从头开始
'OI-Wiki '
>>> s[8:14]  # 左闭右开设计的妙处，长度恰好为 14-8=6，还和上一个字符串无缝衔接
'感谢你的阅读'
>>> s[-4:]  # 省略右开索引则直到结尾
'与贡献!'
>>> s[8:14:2]  # 步长为 2
'感你阅'
>>> s[::-1]  # 步长为 -1 时，获得了反转的字符串
'! 献贡与参迎欢\n 读阅的你谢感 ikiW-IO'
>>> s  # 但原来的字符串并未改变
'OI-Wiki 感谢你的阅读\n 欢迎参与贡献!'
```

C/C++ 中 `char` 类型可以和对应的 ASCII 码互转，而在 Python 中你可以对一个 Unicode 字符使用内置函数 `ord()` 将其转换为对应的 Unicode 编码，逆向的转换使用内置函数 `chr()`。

如果想把数字转换成对应的字符串，可以使用内置函数 `str()`，反之可以使用 `int()` 和 `float()`，你可以类比为 C/C++ 中的强制类型转换，但括号不是加在类型上而是作为函数的一部分括住参数。

Python 的字符串类型提供了许多强大的方法，包括计算某字符的索引与出现次数，转换大小写等等，这里就不一一列举，强烈建议查看 官方文档[15] 熟悉常用方法，遇到字符串操作应当首先考虑使用这些方法而非自力更生。

开数组

从 C++ 转过来的同学可能很迷惑怎么在 Python 中开数组，这里就介绍在 Python 开「数组」的语法，需要强调我们介绍的其实是几种 序列类型[17]，和 C 的数组有着本质区别，而更接近 C++ 中的 `vector`。

使用 `list`

列表（`list`）大概是 Python 中最常用也最强大的序列类型，列表中可以存放任意类型的元素，包括嵌套的列表，这符合数据结构中「广义表」的定义。请注意不要将其与 C++ STL 中的双向链表 `list` 混淆，故本文将使用「列表」而非 `list` 以免造成误解。

```
>>> []  # 创建空列表，注意列表使用方括号
[]
>>> nums = [0, 1, 2, 3, 5, 8, 13]; nums  # 初始化列表，注意整个列表可以直接打印
[0, 1, 2, 3, 5, 8, 13]
>>> nums[0] = 1; nums  # 支持索引访问，还支持修改元素
[1, 1, 2, 3, 5, 8, 13]
>>> nums.append(nums[-2]+nums[-1]); nums  # append() 同 vector 的 push_back()，也
都没有返回值
[1, 1, 2, 3, 5, 8, 13, 21]
>>> nums.pop()  # 弹出并返回末尾元素，可以当栈使用；其实还可指定位置，默认是末尾
21
>>> nums.insert(0, 1); nums  # 同 vector 的 insert(position, val)
[1, 1, 1, 2, 3, 5, 8, 13]
>>> nums.remove(1); nums  # 按值移除元素（只删第一个出现的），若不存在则抛出错误
[1, 1, 2, 3, 5, 8, 13]
>>> len(nums)  # 求列表长度，类似 vector 的 size()，但 len() 是内置函数
7
>>> nums.reverse(); nums  # 原地逆置
[13, 8, 5, 3, 2, 1, 1]
>>> sorted(nums)  # 获得排序后的列表
[1, 1, 2, 3, 5, 8, 13]
>>> nums  # 但原来的列表并未排序
[13, 8, 5, 3, 2, 1, 1]
>>> nums.sort(); nums  # 原地排序，可以指定参数 key 作为排序标准
[1, 1, 2, 3, 5, 8, 13]
>>> nums.count(1)  # 类似 std::count()
2
>>> nums.index(1)  # 返回值首次出现项的索引号，若不存在则抛出错误
0
>>> nums.clear(); nums  # 同 vector 的 clear()
```

以上示例展现了列表与 `vector` 的相似之处，`vector` 中常用的操作一般也都能在列表中找到对应方法，不过某些方法如 `len()`,`sorted()` 会以内置函数的面目出现，而 STL 算法中的函数如 `find()`,`count()`,`max_element()`,`sort()`,`reverse()` 在 Python 中又成了对象的方法，使用时需要注意区分，更多方法请参见官方文档的 列表详解[18]。下面将展示列表作为 Python 的基本序列类型的一些强大功能：

Python 支持多种复合数据类型，可将不同值组合在一起。最常用的 `list`，类型是用方括号标注、逗号分隔的一组值。例如，`[1, 2, 3]` 和 `['a','b','c']` 都是列表。

```
>>> lst = [1, '1'] + ["2", 3.0]  # 列表直接相加生成一个新列表
>>> lst  # 这里存放不同的类型只是想说明可以这么做，但这不是好的做法
[1, '1', '2', 3.0]
>>> 3 in lst  # 实用的成员检测操作，字符串也有该操作且还支持子串检测
True
>>> [1, '1'] in lst  # 仅支持单个成员检测，不会发现「子序列」
False
>>> lst[1:3] = [2, 3]; lst  # 切片并赋值，原列表被修改
[1, 2, 3, 3.0]
>>> lst[::-1]  # 获得反转后的新列表
[3.0, 3, 2, 1]
>>> lst *= 2; lst  # 数乘拼接
[1, 2, 3, 3.0, 1, 2, 3, 3.0]
>>> del lst[4:]; lst  # 也可写 lst[4:] = []，del 语句不止可以用于删除序列中元素
[1, 2, 3, 3.0]
```

以上示例展现了列表作为序列的一些常用操作，可以看出许多操作如切片是与字符串相通的，但字符串是「不可

变序列」而列表是「可变序列」，故可以通过切片灵活地修改列表。在 C/C++ 中我们往往会通过循环处理字符数组，下面将展示如何使用 「 列表推导式 」[19] 在字符串和列表之间转换：

```
>>> # 建立一个 [65, 70) 区间上的整数数组，range 也是一种类型，可看作左闭右开区间，第
三个参数为步长可省略
>>> nums = list(range(65,70))  # 记得 range 外面还要套一层 list()
[65, 66, 67, 68, 69]
>>> lst = [chr(x) for x in nums]  # 列表推导式的典型结构，[exp for var in iterabl
e if cond]
>>> lst  # 上两句可以合并成 [chr(x) for x in range(65,70)]
['A', 'B', 'C', 'D', 'E']
>>> s = ''.join(lst); s # 用空字符串 '' 拼接列表中的元素生成新字符串
'ABCDE'
>> list(s)  # 字符串生成字符列表
['A', 'B', 'C', 'D', 'E']
>>> # 如果你不知道有 s.lower() 方法就可能写出下面这样新瓶装旧酒的表达式
>>> ''.join([chr(ord(ch) - 65 + 97) for ch in s if ch >= 'A' and ch <= 'Z'])
'abcde'
```

下面演示一些在 OI 中更常见的场景，比如二维「数组」：

```
>>> vis = [[0] * 3] * 3  # 开一个 3*3 的全 0 数组
>>> vis
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> vis[0][0] = 1; vis  # 怎么会把其他行也修改了？
[[1, 0, 0], [1, 0, 0], [1, 0, 0]]
>>> # 先来看下一维列表的赋值
>>> a1 = [0, 0, 0]; a2 = a1; a3 = a1[:]  # 列表也可以直接被赋给新的变量
>>> a1[0] = 1; a1  # 修改列表 a1，似乎正常
[1, 0, 0]
>>> a2  # 怎么 a2 也被改变了
[1, 0, 0]
>>> a3  # a3 没有变化
[0, 0, 0]
>>> id(a1) == id(a2) and id(a1) != id(a3)  # 内置函数 id() 给出对象的「标识值」，可
类比为地址，地址相同说明是一个对象
True
>>> vis2 = vis[:];  # 拷贝一份二维列表看看
>>> vis[0][1] = 2; vis  # vis 肯定还是被批量修改
>>> [[1, 2, 0], [1, 2, 0], [1, 2, 0]]
>>> vis2  # 但 vis2 是切片拷贝的怎么还是被改了
>>> [[1, 2, 0], [1, 2, 0], [1, 2, 0]]
>>> id(vis) != id(vis2)  # vis 和 vis2 确实不是一个对象啊
True
>>> # 谜底揭晓，vis2 虽然不是 vis 的引用，但其中对应行都指向相同的对象
>>> [[id(vis[i]) == id(vis2[i]) for i in range(3)]
[True, True, True]
>>> # 回看二维列表自身
>>> [id(x) for x in vis]  # 具体数字和这里不一样但三个值一定相同，说明是三个相同对
象
[139760373248192, 139760373248192, 139760373248192]
```

其实我们一直隐瞒了一个重要事实，Python 中赋值只传递了引用而非创建新值，你可以创建不同类型的变量并赋给新变量，验证发现二者的标识值是相同的，只不过直到现在我们才介绍了列表这一种可变类型，而给数字、字符串这样的不可变类型赋新值时实际上创建了新的对象，故而前后两个变量互不干扰。但列表是可变类型，所以我们修改一个列表的元素时，另一个列表由于指向同一个对象所以也被修改了。创建二维数组也是类似的情况，示例中用乘

法创建二维列表相当于把 `[0]*3` 这个一维列表重复了 3 遍，所以涉及其中一个列表的操作会同时影响其他两个列表。更不幸的是，在将二维列表赋给其他变量的时候，就算用切片来拷贝，也只是「浅拷贝」，其中的元素仍然指向相同的对象，解决这个问题需要使用标准库中的 deepcopy[20]，或者尽量避免整个赋值二维列表。不过还好，创建二维列表时避免创建重复的列表还是比较简单，只需使用「列表推导式」:

```
>>> vis1 = [[0] * 3 for _ in range(3)]  # 把用不到的循环计数变量设为下划线 _ 是一
种惯例
>>> # 但在 REPL 中 _ 默认指代上一个表达式输出的结果，故也可使用双下划线
>>> vis1
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> [id(x) for x in vis1]  # 具体数字和这里不一样但三个值一定不同，说明是三个不同
对象
[139685508981248, 139685508981568, 139685508981184]
>>> vis1[0][0] = 1
[[1, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> a2[0][0] = 10  # 访问和赋值二维数组
```

我们未讲循环的用法就先介绍了列表推导式，这是由于 Python 是高度动态的解释型语言，因此其程序运行有大量的额外开销。尤其是 **for 循环在 Python 中运行的奇慢无比**。因此在使用 Python 时若想获得高性能，尽量使用使用列表推导式，或者 `filter`,`map` 等内置函数直接操作整个序列来避免循环，当然这还是要根据具体问题而定。

使用 NumPy

" 什么是 NumPy"

NumPy[21] 是著名的 Python 科学计算库，提供高性能的数值及矩阵运算。在测试算法原型时可以利用 NumPy 避免手写排序、求最值等算法。NumPy 的核心数据结构是 `ndarray`，即 n 维数组，它在内存中连续存储，是定长的。此外 NumPy 核心是用 C 编写的，运算效率很高。不过需要注意，它不是标准库的一部分，可以使用 `pip install numpy` 安装，但不保证 OI 考场环境中可用。

下面的代码将介绍如何利用 NumPy 建立多维数组并进行访问。

```
>>> import numpy as np  # 请自行搜索 import 的意义和用法
>>> np.empty(3) # 开容量为 3 的空数组，注意没有初始化为 0
array([0.00000000e+000, 0.00000000e+000, 2.01191014e+180])
>>> np.zeros((3, 3)) # 开 3*3 的数组，并初始化为 0
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
>>> a1 = np.zeros((3, 3), dtype=int) # 开 3×3 的整数数组
>>> a1[0][0] = 1 # 访问和赋值
>>> a1[0, 0] = 1 # 更友好的语法
>>> a1.shape # 数组的形状
(3, 3)

>>> a1[:2, :2] # 取前两行、前两列构成的子阵，无拷贝
array([[1, 0],
       [0, 0]])

>>> a1[0, 2] # 获取第 1、3 列，无拷贝
array([[1, 0],
       [0, 0],
       [0, 0]])
>>> np.max(a1) # 获取数组最大值
1
```

```
>>> a1.flatten() # 将数组展平
array([1, 0, 0, 0, 0, 0, 0, 0, 0])

>>> np.sort(a1, axis = 1) # 沿行方向对数组进行排序，返回排序结果
array([[0, 0, 1],
       [0, 0, 0],
       [0, 0, 0]])
>>> a1.sort(axis = 1) # 沿行方向对数组进行原地排序
```

使用 array

array[22] 是 Python 标准库提供的一种高效数值数组，可以紧凑地表示基本类型值的数组，但不支持数组嵌套，也很少见到有人使用它，这里只是顺便提一下。

若无特殊说明，后文出现「数组」一般指「列表」。

输入输出[23]

Python 中的输入输出主要通过内置函数 input() 和 print() 完成，print() 的用法十分符合直觉：

```
>>> a = [1,2,3]; print(a[-1])  # 打印时默认末尾换行
3
>>> print(ans[0], ans[1])  # 可以输出任意多个变量，默认以空格间隔
1 2
>>> print(a[0], a[1], end='')  # 令 end='', 使末尾不换行
1 2>>>
>>> print(a[0], a[1], sep=', ')  # 令 sep=', '，改变间隔样式
1, 2
>>> print(str(a[0]) + ', ' + str(a[1]))  # 输出同上，但是手动拼接成一整个字符串
```

算法竞赛中通常只涉及到基本的数值和字符串输出，以上用法基本足够，只有当涉及到浮点数位数时需要用到格式化字符串输出。格式化有三种方法，第一种也是最老旧的方法是使用 printf() 风格的 % 操作符；另一种是利用 format 函数[24]，写起来比较长；第三种是 Python 3.6 新增的 f-string[25]，最为简洁，但不保证考场中的 Python 版本足够新。详细丰富的说明可以参考 这个网页[26]，尽管更推荐使用 format() 方法，但为了获得与 C 接近的体验，下面仅演示与 printf() 类似的老式方法：

```
>>> pi = 3.1415926; print('%.4f' % pi)   # 格式为 %[flags][width][.precision]typ
e
3.1416
>>> '%.4f - %8f = %d' % (pi, 0.1416, 3)  # 右边多个参数用 () 括住，后面会看到其实
是「元组」
'3.1416 - 0.141600 = 3'
```

input() 函数的行为接近 C++ 中的 getline()，即将一整行作为字符串读入，且末尾没有换行符，但在算法竞赛中，常见的输入形式是一行输入多个数值，因此就需要使用字符串的 split() 方法并搭配列表推导式得到存放数值类型的列表，下面以输入 n 个数求平均值为例演示输入 n 个数得到「数组」的方法：

```
>>> s = input('请输入一串数字: '); s  # 自己调试时可以向 input() 传入字符串作为提
示
请输入一串数字: 1 2 3 4 5 6
'1 2 3 4 5 6'
>>> a = s.split(); a
['1', '2', '3', '4', '5', '6']
>>> a = [int(x) for x in a]; a
[1, 2, 3, 4, 5, 6]
>>> # 以上输入过程可写成一行 a = [int(x) for x in input().split()]
```

```
>>> sum(a) / len(a)  # sum() 是内置函数
3.5
```

有时题目会在每行输入固定几个数，比如边的起点、终点、权重，如果只用上面提到的方法就只能每次读入数组然后根据下标赋值，这时可以使用 Python 的「拆包」特性一次赋值多个变量：

```
>>> u, v, w = [int(x) for x in input().split()]
1 2 4
>>> print(u,v,w)
1 2 4
```

题目中经常遇到输入 N 行的情况，可我们还没有讲最基本的循环语句，但 Python 强大的序列操作能在不使用循环的情况下应对多行输入，下面假设将各条边的起点、终点、权值分别读入三个数组：

```
>>> N = 4; mat = [[int(x) for x in input().split()] for i in range(N)]
1 3 3
1 4 1
2 3 4
3 4 1
>>> mat  # 先按行读入二维数组
[[1, 3, 3], [1, 4, 1], [2, 3, 4], [3, 4, 1]]
>>> u, v, w = map(list, zip(*mat))
# * 将 mat 解包得到里层的多个列表
# zip() 将多个列表中对应元素聚合成元组，得到一个迭代器
# map(list, iterable) 将序列中的元素（这里为元组）转成列表
>>> print(u, v, w)  # 直接将 map() 得到的迭代器拆包，分别赋值给 u, v, w
[1, 1, 2, 3] [3, 4, 3, 4] [3, 1, 4, 1]
```

上述程序实际上相当于先读入一个 N 行 3 列的矩阵，然后将其转置成 3 行 N 列的矩阵，也就是外层列表中嵌套了 3 个列表，最后将代表这起点、终点、权值的 3 个列表分别赋值给 u, v, w。内置函数 `zip()`[27] 可以将多个等长序列中的对应元素拼接在「元组」内，得到新序列。而 `map()` 其实是函数式编程的一种操作，它将一个给定函数作用于 `zip()` 所产生序列的元素，这里就是用 `list()` 将元组变成列表。你可以自行练习使用 `*` 和 `zip()`[27]，`map()`[28] 以理解其含义。需要注意的是 Python 3 中 `zip()` 和 `map()` 创建的不再返回列表而是返回迭代器，这里暂不解释它们之间的异同，你可以认为迭代器可以产生列表中的各个元素，用 `list()` 套住迭代器就能生成列表。

控制流程[29]

尽管我们已经学习了 Python 的许多特性，但到目前为止我们展示的 Python 代码都是单行语句，这掩盖了 Python 和 C 在代码风格上的重大差异：首先，Python 中不用 `{}` 而是用缩进表示块结构，如果缩进没有对齐会直接报错，如果 tab 和空格混用也会报错；其次，块结构开始的地方比如 `if` 和 `for` 语句的行末要有冒号 `:`。这有助于代码的可读性，但你也可能怀念 C 那种自由的体验，毕竟如果复制粘贴时因为丢失缩进而不得不手动对齐是很恼人的。

循环结构

列表推导式能在一行内高效地完成批量操作，但有时为了压行我们已经显得过分刻意，许多场景下还是只能使用循环结构，所以我们再以读入多行数据为例展示 Python 中的循环是如何编写的：

```
# 请注意从现在开始我们不再使用 REPL，请自行复制多行数据
u, v, w = ([] for i in range(3))  # 多变量赋值
for i in range(4):  # 这里假设输入 4 行数据
    _u, _v, _w = [int(x) for x in input().split()]
    u.append(_u), v.append(_v), w.append(_w)
    # 不可进行类似 cin >> u[i] >> v[i] >> w[i] 的操作，因为必定超出列表当前的长度
    # 当然你可以选择初始化长度为 MAXN 的全 0 列表，不过需要记住真实长度并删掉多余
```

```
元素
print(u, v, w)
```

需要注意，Python 中的 for 循环和 C/C++ 有较大的差别，其作用类似 C++ 11 引入的「基于范围的循环」，实质是迭代序列中的元素，比如编写循环遍历数组下标需要迭代 `range(len(lst))`，而非真正定义起始和终止条件，所以使用起来并没有 C/C++ 灵活。

下面再用 while 循环展示行数不定的情况下如何输入：

```
u, v, w = [], [], []  # 多变量赋值，其实同上
s = input()  # 注意 Python 中赋值语句不能放在条件表达式中
while s:  # 不能像 C 那样 while(!scanf())
    # 用切片拼接避免了 append()，注意列表推导式中又嵌套了列表
    u[len(u):], v[len(v):], w[len(w):] = [[int(x)] for x in s.split()]
    s = input()
# Python 3.8 引入了 walrus operator 海象运算符后，你可以节省两行，但考场环境很可能
不支持
while s := input():
    u[len(u):], v[len(v):], w[len(w):] = [[int(x)] for x in s.split()]
print(u, v, w)
```

选择结构

和 C/C++ 大同小异，一些形式上的差别都在下面的示例中有所展示，此外还需注意条件表达式中不允许使用赋值运算符（Python 3.8 以上可用 `:=`[30]），以及 没有 switch 语句[31]。

```
# 条件表达式两侧无括号
if 4 >= 3 > 2 and 3 != 5 == 5 != 7:
    print(" 关系运算符可以连续使用")
    x = None or [] or -2
    print("&&  ||  !", " 与  或  非", "and or not", sep='\n')
    print(" 善用 and/or 可节省行数")
    if not x:
        print(" 负数也是 True，不执行本句")
    elif x & 1:
        print(" 用 elif 而不是 else if\n"
        " 位运算符与 C 相近，偶数 &1 得 0，不执行本句")
    else:
        print(" 也有三目运算符") if x else print(" 注意结构")
```

异常处理

尽管 C++ 中有 try 块[32] 用于异常处理，但竞赛中一般从不使用，而 Python 中常见的是 EAFP[33] 风格，故而代码中可能大量使用 `try-except`[34] 语句，在后文介绍 `dict` 这一结构时还会用到，这里展示：

```
s = "OI-wiki"
pat = "NOIP"
x = s.find(pat)  # find() 找不到返回 -1
try:
    y = s.index(pat)  # index() 找不到则抛出错误
    print(y)  # 这句被跳过
except ValueError:
    print(" 没找到")
    try:
        print(y)  # 此时 y 并没有定义，故又会抛出错误
    except NameError as e:
```

```
        print(" 无法输出 y")
        print(" 原因:", e)
```

文件读写[35]

Python 内置函数 `open()`[36] 用于文件读写，为了防止读写过程中出错导致文件未被正常关闭，这里只介绍使用 `with`[35] 语句的安全读写方法：

```python
a = []
with open('in.txt') as f:
    N = int(f.readline())  # 读入第一行的 N
    a[len(a):] = [[int(x) for x in f.readline().split()] for i in range(N)]

with open('out.txt', 'w') as f:
    f.write('1\n')
```

关于文件读写的函数有很多，分别适用于不同的场景，由于 OI 赛事尚不支持使用 Python，这里从略。

内置容器

Python 内置了许多强大的容器类型，只有熟练使用并了解其特点才能真正让 Python 在算法竞赛中有用武之地，除了上面详细介绍的 `list`（列表），还有 `tuple`（元组）、`dict`[37]（字典）和 `set`（集合）这几种类型。

元组可以简单理解成不可变的列表，不过还需注意「不可变」的内涵，如果元组中的某元素是可变类型比如列表，那么仍可以修改该列表的值，元组中存放的是对列表的引用所以元组本身并没有改变。元组的优点是开销较小且「可哈希[38]」，后者在创建字典和集合时非常有用。

```python
tup = tuple([[1,2], 4])  # 由列表得到元组
# 等同于 tup = ([1,2], 4)
tup[0].append(3)
print(tup)
a, b = 0, "I-Wiki"  # 多变量赋值其实是元组拆包
print(id(a), id(b))
b, a = a, b
print(id(a), id(b))  # 你应该会看到 a, b 的 id 值现在互换了
# 这更说明 Python 中，变量更像是名字，赋值只是让其指代对象
```

字典就像 C++ STL 中的 map（请注意和 Python 中内置函数 `map()`[28] 区分）用于存储键值对，形式类似 JSON[39]，但 JSON 中键必须是字符串且以双引号括住，字典则更加灵活强大，可哈希的对象都可作为字典的键。需要注意 Python 几次版本更新后字典的特性有了较多变化，包括其中元素的顺序等，请自行探索。

```python
dic = {'key': "value"}  # 基本形式
dic = {chr(i): i for i in range(65, 91)}  # 大写字母到对应 ASCII 码的映射，注意断
句
dic = dict(zip([chr(i) for i in range(65, 91)], range(65,91)))  # 效果同上
dic = {dic[k]: k for k in dic}  # 将键值对逆转，for k in dic 迭代其键
dic = {v: k for k, v in dic.items()}  # 和上行作用相同，dic.items() 以元组存放单个
键值对
dic = {k: v for k, v in sorted(dic.items(), key=lambda x:-x[1])}  # 字典按值逆排
序，用到了 lambda 表达式

print(dic['A'])  # 返回 dic 中以 'A' 为键的项，这里值为 65
dic['a'] = 97  # 将 d[key] 设为 value，字典中原无 key 就是直接插入
if 'b' in dic:  # LBYL(Look Before You Leap) 风格
    print(dic['b'])  # 若字典中无该键则会出错，故先检查
else:
```

```python
    dic['b'] = 98

# 经典场景统计出现次数
# 新键不存在于原字典，需要额外处理
try:  # EAFP (Easier to Ask for Forgiveness than Permission) 风格
    cnter[key] += 1
except KeyError:
    cnter[key] = 1
```

集合就像 C++ STL 中的 `set`，不会保存重复的元素，可以看成只保存键的字典。需要注意集合和字典都用 `{}` 括住，不过单用 `{}` 会创建空字典而不是空集合，这里就不再给出示例。

编写函数

Python 中定义函数无需指定参数类型和返回值类型，无形中为 OI 选手减少了代码量

```python
def add(a, b):
    return a + b  # 动态类型的优势，a 和 b 也可以是字符串


def add_no_swap(a, b):
    print('in func #1:', id(a), id(b))
    a += b
    b, a = a, b
    print('in func #2:', id(a), id(b))  # a, b 已交换
    return a, b  # 返回多个值，其实就是返回元组，可以拆包接收


lst1 = [1, 2]; lst2 = [3, 4]
print('outside func #1:', id(lst1), id(lst2))
add_no_swap(lst1, lst2)
# 函数外 lst1, lst2 并未交换
print('outside func #2:', id(lst1), id(lst2))
# 不过值确实已经改变
print(lst1, lst2)
```

默认参数

Python 中函数的参数非常灵活，有关键字参数、可变参数等，但在算法竞赛中这些特性的用处并不是很大，这里只介绍一下默认参数，因为 C++ 中也有默认参数，且在 Python 中使用默认参数很有可能遇到坑。

```python
def append_to(element, to=[]):
    to.append(element)
    return to

lst1 = append_to(12)
lst2 = append_to(42)
print(lst1, lst2)

# 你可能以为输出是 [12] [42]
# 但运行结果其实是 [12] [12, 42]

# 这是因为默认参数的值仅仅在函数定义的时候赋值一次
# 默认参数的值应该是不可变对象，使用 None 占位是一种最佳实践
def append_to(element, to=None):
```

```
    if to is None:
        to = []
    to.append(element)
    return to
```

类型标注

Python 是一个动态类型检查的语言，以灵活但隐式的方式处理类型，Python 解释器仅仅在运行时检查类型是否正确，并且允许在运行时改变变量类型，俗话说「动态类型一时爽，代码重构火葬场」，程序中的一些错误可能在运行时才会暴露：

```
>>> if False:
...     1 + "two"  # This line never runs, so no TypeError is raised
... else:
...     1 + 2
...
3

>>> 1 + "two"  # Now this is type checked, and a TypeError is raised
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Python 3.5 后引入了类型标注，允许设置函数参数和返回值的类型，但只是作为提示，并没有实际的限制作用，需要静态检查工具才能排除这类错误（例如 PyCharm[40] 和 Mypy[41]），所以显得有些鸡肋，对于 OIer 来说更是只需了解，可按如下方式对函数的参数和返回值设置类型标注：

```
def headline(
    text,           # type: str
    width = 80,       # type: int
    fill_char = "-",  # type: str
):                  # type: (...) -> str
    return f"{text.title()}".center(width, fill_char)

print(headline("type comments work", width = 40))
```

除了函数参数，变量也是可以类型标注的，你可以通过调用 `__annotations__` 来查看函数中所有的类型标注。变量类型标注赋予了 Python 静态语言的性质，即声明与赋值分离：

```
>>> nothing: str
>>> nothing
NameError: name 'nothing' is not defined

>>> __annotations__
{'nothing': <class 'str'>}
```

装饰器

装饰器是一个函数，接受一个函数或方法作为其唯一的参数，并返回一个新函数或方法，其中整合了修饰后的函数或方法，并附带了一些额外的功能。简而言之，可以在不修改函数代码的情况下，增加函数的功能。相关知识可以参考 官方文档[42]。

部分装饰器在竞赛中非常实用，比如 `lru_cache`[43]，可以为函数自动增加记忆化的能力，在递归算法中非常实用：

`@lru_cache(maxsize=128,typed=False)`

- 传入的参数有 2 个：`maxsize` 和 `typed`，如果不传则 `maxsize` 的默认值为 128，`typed` 的默认值为 `False`。

- 其中 maxsize 参数表示的是 LRU 缓存的容量，即被装饰的方法的最大可缓存结果的数量。如果该参数值为 128，则表示被装饰方法最多可缓存 128 个返回结果；如果 maxsize 传入为 None 则表示可以缓存无限个结果。
- 如果 typed 设置为 True，不同类型的函数参数将被分别缓存，例如，f(3) 和 f(3.0) 会缓存两次。

以下是使用 lru_cache 优化计算斐波那契数列的例子：

```python
@lru_cache(maxsize = None)
def fib(n):
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)
```

常用内置库

在这里介绍一些写算法可能用得到的内置库，具体用法可以自行搜索或者阅读 官方文档[44]。

库名	用途
array[45]	定长数组
argparse[46]	命令行参数处理
bisect[47]	二分查找
collections[48]	有序字典、双端队列等数据结构
fractions[49]	有理数
heapq[50]	基于堆的优先级队列
io[51]	文件流、内存流
itertools[52]	迭代器
math[53]	数学函数
os.path[54]	系统路径等
random[55]	随机数
re[56]	正则表达式
struct[57]	转换结构体和二进制数据
sys[58]	系统信息

从例题对比 C++ 与 Python

"例题 洛谷 P4779【模板】单源最短路径（标准版）[59]"

给定一个 $n(1 \le n \le 10^5)$ 个点、$m(1 \le m \le 2 \times 10^5)$ 条有向边的带非负权图，请你计算从 s 出发，到每个点的距离。数据保证能从 s 出发到任意点。

声明常量

```cpp
#include <bits/stdc++.h>
using namespace std;
const int N = 1e5 + 5, M = 2e5 + 5;
```

```python
try: # 引入优先队列模块
    import Queue as pq #python version < 3.0
except ImportError:
    import queue as pq #python3.*

N = int(1e5 + 5)
M = int(2e5 + 5)
INF = 0x3f3f3f3f
```

声明前向星结构体和其它变量

```cpp
struct qxx {
  int nex, t, v;
};

qxx e[M];
int h[N], cnt;

void add_path(int f, int t, int v) { e[++cnt] = (qxx){h[f], t, v}, h[f] = cnt; }

typedef pair<int, int> pii;
priority_queue<pii, vector<pii>, greater<pii>> q;
int dist[N];
```

```python
class qxx:  # 前向星类（结构体）
    def __init__(self):
        self.nex = 0
        self.t = 0
        self.v = 0

e = [qxx() for i in range(M)]  # 链表
h = [0 for i in range(N)]
cnt = 0

dist = [INF for i in range(N)]
q = pq.PriorityQueue()  # 定义优先队列，默认第一元小根堆

def add_path(f, t, v):  # 在前向星中加边
    # 如果要修改全局变量，要使用 global 来声明
    global cnt, e, h
    # 调试时的输出语句，多个变量使用元组
    # print("add_path(%d,%d,%d)" % (f,t,v))
    cnt += 1
    e[cnt].nex = h[f]
    e[cnt].t = t
    e[cnt].v = v
    h[f] = cnt
```

Dijkstra 算法

```cpp
void dijkstra(int s) {
  memset(dist, 0x3f, sizeof(dist));
  dist[s] = 0, q.push(make_pair(0, s));
  while (q.size()) {
    pii u = q.top();
    q.pop();
    if (dist[u.second] < u.first) continue;
    for (int i = h[u.second]; i; i = e[i].nex) {
      const int &v = e[i].t, &w = e[i].v;
      if (dist[v] <= dist[u.second] + w) continue;
      dist[v] = dist[u.second] + w;
      q.push(make_pair(dist[v], v));
    }
  }
}
```

```python
def nextedgeid(u):  # 生成器，可以用在 for 循环里
    i = h[u]
    while i:
        yield i
        i = e[i].nex


def dijkstra(s):
    dist[s] = 0
    q.put((0, s))
    while not q.empty():
        u = q.get()  # get 函数会顺便删除堆中对应的元素
        if dist[u[1]] < u[0]:
            continue
        for i in nextedgeid(u[1]):
            v = e[i].t
            w = e[i].v
            if dist[v] <= dist[u[1]]+w:
                continue
            dist[v] = dist[u[1]]+w
            q.put((dist[v], v))
```

主函数

```cpp
int n, m, s;

int main() {
  scanf("%d%d%d", &n, &m, &s);
  for (int i = 1; i <= m; i++) {
    int u, v, w;
    scanf("%d%d%d", &u, &v, &w);
    add_path(u, v, w);
  }
  dijkstra(s);
  for (int i = 1; i <= n; i++) printf("%d ", dist[i]);
```

```
  return 0;
}
```

```python
if __name__ == '__main__':
    # 一行读入多个整数。注意它会把整行都读进来
    n, m, s = map(int, input().split())
    for i in range(m):
        u, v, w = map(int, input().split())
        add_path(u, v, w)

    dijkstra(s)

    for i in range(1, n + 1):
        print(dist[i], end = ' ')

    print()
```

完整代码

```cpp
#include <bits/stdc++.h>
using namespace std;
const int N = 1e5 + 5, M = 2e5 + 5;

struct qxx {
  int nex, t, v;
};

qxx e[M];
int h[N], cnt;

void add_path(int f, int t, int v) { e[++cnt] = (qxx){h[f], t, v}, h[f] = cnt; }

typedef pair<int, int> pii;
priority_queue<pii, vector<pii>, greater<pii>> q;
int dist[N];

void dijkstra(int s) {
  memset(dist, 0x3f, sizeof(dist));
  dist[s] = 0, q.push(make_pair(0, s));
  while (q.size()) {
    pii u = q.top();
    q.pop();
    if (dist[u.second] < u.first) continue;
    for (int i = h[u.second]; i; i = e[i].nex) {
      const int &v = e[i].t, &w = e[i].v;
      if (dist[v] <= dist[u.second] + w) continue;
      dist[v] = dist[u.second] + w;
      q.push(make_pair(dist[v], v));
    }
  }
}

int n, m, s;
```

```cpp
int main() {
  scanf("%d%d%d", &n, &m, &s);
  for (int i = 1; i <= m; i++) {
    int u, v, w;
    scanf("%d%d%d", &u, &v, &w);
    add_path(u, v, w);
  }
  dijkstra(s);
  for (int i = 1; i <= n; i++) printf("%d ", dist[i]);
  return 0;
}
```

```python
try:  # 引入优先队列模块
    import Queue as pq  # python version < 3.0
except ImportError:
    import queue as pq  # python3.*

N = int(1e5+5)
M = int(2e5+5)
INF = 0x3f3f3f3f

class qxx:  # 前向星类（结构体）
    def __init__(self):
        self.nex = 0
        self.t = 0
        self.v = 0

e = [qxx() for i in range(M)]  # 链表
h = [0 for i in range(N)]
cnt = 0

dist = [INF for i in range(N)]
q = pq.PriorityQueue()  # 定义优先队列，默认第一元小根堆

def add_path(f, t, v):  # 在前向星中加边
    # 如果要修改全局变量，要使用 global 来声名
    global cnt, e, h
    # 调试时的输出语句，多个变量使用元组
    # print("add_path(%d,%d,%d)" % (f,t,v))
    cnt += 1
    e[cnt].nex = h[f]
    e[cnt].t = t
    e[cnt].v = v
    h[f] = cnt

def nextedgeid(u):  # 生成器，可以用在 for 循环里
    i = h[u]
    while i:
        yield i
        i = e[i].nex

def dijkstra(s):
    dist[s] = 0
```

```
    q.put((0, s))
    while not q.empty():
        u = q.get()
        if dist[u[1]] < u[0]:
            continue
        for i in nextedgeid(u[1]):
            v = e[i].t
            w = e[i].v
            if dist[v] <= dist[u[1]]+w:
                continue
            dist[v] = dist[u[1]]+w
            q.put((dist[v], v))


# 如果你直接运行这个 python 代码（不是模块调用什么的）就执行命令
if __name__ == '__main__':
    # 一行读入多个整数。注意它会把整行都读进来
    n, m, s = map(int, input().split())
    for i in range(m):
        u, v, w = map(int, input().split())
        add_path(u, v, w)

    dijkstra(s)

    for i in range(1, n + 1):
        # 两种输出语法都是可以用的
        print("{}".format(dist[i]), end=' ')
        # print("%d" % dist[i],end=' ')

    print()  # 结尾换行
```

参考文档

1. Python Documentation，https://www.python.org/doc/[60]
2. Python 官方中文教程，https://docs.python.org/zh-cn/3/tutorial/[61]
3. Learn Python3 In Y Minutes，https://learnxinyminutes.com/docs/python3/[62]
4. Real Python Tutorials，https://realpython.com/[63]
5. 廖雪峰的 Python 教程，https://www.liaoxuefeng.com/wiki/1016959663602400/[64]
6. GeeksforGeeks: Python Tutorials,https://www.geeksforgeeks.org/python-programming-language/[65]

参考资料和注释

[1] 2. Python 解释器 —Python 3 文档

[2] Unicode 指南 —Python 3 文档

[3] 不被支持

[4] 2to3

[5] 列表生成

[6] 北京交通大学自由与开源软件镜像站

[7] 华为开源镜像站

[8] Python 包索引（PyPI）

[9] pypi 镜像使用帮助 - 清华大学开源软件镜像站

[10] MirrorZ

[11] Python 文档

[12] Python Wiki

[13] **内置函数**

[14] decimal

[15] 许多方便的功能 [15-1] [15-2]

[16] Unicode

[17] 序列类型

[18] 列表详解

[19] 「列表推导式」

[20] `deepcopy`

[21] NumPy

[22] `array`

[23] 输入输出

[24] `format` 函数

[25] f-string

[26] 这个网页

[27] `zip()` [27-1] [27-2]

[28] `map()` [28-1] [28-2]

[29] 控制流程

[30] `:=`

[31] 没有 switch 语句

[32] try 块

[33] EAFP

[34] `try-except`

[35] 文件读写 [35-1] [35-2]

[36] `open()`

[37] `dict`

[38] 可哈希

[39] JSON

[40] PyCharm

[41] Mypy

[42] 官方文档

[43] `lru_cache`

[44] 官方文档

[45] `array`

[46] `argparse`

[47] `bisect`

[48] `collections`

[49] fractions

[50] heapq

[51] io

[52] itertools

[53] math

[54] os.path

[55] random

[56] re

[57] struct

[58] sys

[59] 例题洛谷 P4779【模板】单源最短路径（标准版）

[60] https://www.python.org/doc/

[61] https://docs.python.org/zh-cn/3/tutorial/

[62] https://learnxinyminutes.com/docs/python3/

[63] https://realpython.com/

[64] https://www.liaoxuefeng.com/wiki/1016959663602400/

[65] https://www.geeksforgeeks.org/python-programming-language/

4.8 Java 速成

关于 Java

Java 是一种广泛使用的计算机编程语言，拥有**跨平台**、**面向对象**、**泛型编程**的特性，广泛应用于企业级 Web 应用开发和移动应用开发。

环境安装

参见 JDK。

基本语法

主函数

Java 类似 C/C++ 语言，需要一个函数（在面向对象中，这被称为方法）作为程序执行的入口点。

Java 的主函数的格式是固定的，形如：

```
class Test {
    public static void main(String[] args) {
        // 程序的代码
    }
}
```

一个打包的 Java 程序（名称一般是 `*.jar`）中可以有很多个类似的函数，但是当运行这个程序的时候，只有其中一个函数会被运行，这是定义在 `Jar` 的 `Manifest` 文件中的，在 OI 比赛中一般用不到关于它的知识。

注释

和 C/C++ 一样，Java 使用 `//` 和 `/* */` 分别注释单行和多行。

基本数据类型

类型名	意义
boolean	布尔类型
byte	字节类型
char	字符型
double	双精度浮点
float	单精度浮点
int	整型
long	长整型
short	短整型
null	空

声明变量

```
int a = 12; // 设置 a 为整数类型，并给 a 赋值为 12
String str = "Hello, OI-wiki"; // 声明字符串变量 str
char ch = 'W';
double PI = 3.1415926;
```

final 关键字

`final` 含义是这是最终的、不可更改的结果，被 `final` 修饰的变量只能被赋值一次，赋值后不再改变。

```java
final double PI = 3.1415926;
```

数组

```java
// 有十个元素的整数类型数组
// 其语法格式为数据类型 [] 变量名 = new 数据类型 [数组大小]
int[] ary = new int[10];
```

字符串

- 字符串是 Java 一个内置的类。

```java
// 最为简单的构造一个字符串变量的方法如下
String a = "Hello";

// 还可以使用字符数组构造一个字符串变量
char[] stringArray = { 'H', 'e', 'l', 'l', 'o' };
String s = new String(stringArray);
```

包和导入包

Java 中的类（Class）都被放在一个个包（package）里面。在一个包里面不允许有同名的类。在类的第一行通常要说明这个类是属于哪个包的。例如：

```java
package org.oi-wiki.tutorial;
```

包的命名规范一般是：项目所有者的顶级域 . 项目所有者的二级域 . 项目名称。

通过 import 关键字来导入不在本类所属的包下面的类。例如下面要用到的 Scanner：

```java
import java.util.Scanner;
```

如果想要导入某包下面所有的类，只需要把这个语句最后的分号前的类名换成 *。

输入

可以通过 Scanner 类来处理命令行输入。

```java
package org.oiwiki.tutorial;

import java.util.Scanner;

class Test {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in); // System.in 是输入流
        int a = scan.nextInt();
        double b = scan.nextDouble();
        String c = scan.nextLine();
    }
}
```

输出

可以对变量进行格式化输出。

符号	意义
%f	浮点类型
%s	字符串类型
%d	整数类型
%c	字符类型

```java
class Test {
    public static void main(String[] args) {
        int a = 12;
        char b = 'A';
        double s = 3.14;
        String str = "Hello world";
        System.out.printf("%f\n", s);
        System.out.printf("%d\n", a);
        System.out.printf("%c\n", b);
        System.out.printf("%s\n", str);
    }
}
```

控制语句

Java 的流程控制语句与 C++ 是基本相同的。

选择

- if

```java
class Test {
    public static void main(String[] args) {
        if ( /* 判断条件 */ ){
            // 条件成立时执行这里面的代码
        }
    }
}
```

- if...else

```java
class Test {
    public static void main(String[] args) {
        if ( /* 判断条件 */ ) {
            // 条件成立时执行这里面的代码
        } else {
            // 条件不成立时执行这里面的代码
        }
    }
}
```

- if...else if...else

```java
class Test {
    public static void main(String[] args) {
        if ( /* 判断条件 */ ) {
            //判断条件成立执行这里面的代码
        } else if ( /* 判断条件 2 */ ) {
            // 判断条件 2 成立执行这里面的代码
        } else {
          // 上述条件都不成立执行这里面的代码
        }
    }
}
```

- switch...case

```java
class Test {
    public static void main(String[] args) {
        switch ( /* 表达式 */ ){
          case /* 值 1 */:
              // 当表达式取得的值符合值 1 执行此段代码
              break; // 如果不加上 break 语句，会让程序按顺序往下执行直到 break
          case /* 值 2 */:
              // 当表达式取得的值符合值 2 执行此段代码
              break;
          default:
              // 当表达式不符合上面列举的值的时候执行这里面的代码
        }
    }
}
```

循环

- for

for 关键字有两种使用方法，其中第一种是普通的 for 循环，形式如下：

```java
class Test {
    public static void main(String[] args) {
        for ( /* 初始化 */; /* 循环的判断条件 */; /* 每次循环后执行的步骤 */ ) {
            // 当循环的条件成立执行循环体内代码
        }
    }
}
```

第二种是类似 C++ 的 foreach 使用方法，用于循环数组或者集合中的数据，相当于把上一种方式中的循环变量隐藏起来了，形式如下：

```java
class Test {
    public static void main(String[] args) {
        for   ( /* 元素类型 X */ /* 元素名 Y */ : /* 集合 Z */ ) {
            // 这个语句块的每一次循环时，元素 Y 分别是集合 Z 中的一个元素。
        }
    }
}
```

- while

```java
class Test {
    public static void main(String[] args) {
        while ( /* 判定条件 */ ) {
            // 条件成立时执行循环体内代码
        }
    }
}
```

- do...while

```java
class Test {
    public static void main(String[] args) {
        do {
          // 需要执行的代码
        } while ( /* 循环判断条件 */ );
    }
}
```

注意事项

类名与文件名一致

创建 Java 源程序需要类名和文件名一致才能编译通过，否则编译器会提示找不到类。通常该文件名会在具体 OJ 中指定。

例：

`Add.java`

```java
class Add {
    public static void main(String[] args) {
        // ...
    }
}
```

在该文件中需使用 `Add` 为类名方可编译通过。

4.9 Java 进阶

> " 注意"
>
> 以下内容均基于 Java JDK 8 版本编写，不排除在更高版本中有部分改动的可能性。

更高速的输入输出

`Scanner` 和 `System.out.print` 在最开始会工作得很好，但是在处理更大的输入的时候会降低效率，因此我们会需要使用一些方法来提高 IO 速度。

使用 Kattio + StringTokenizer 作为输入

最常用的方法之一是使用来自 Kattis 的 Kattio.java[2] 来提高 IO 效率。[1] 这个方法会将 `StringTokenizer` 与 `PrintWriter` 包装在一个类中方便使用。而在具体进行解题的时候（假如赛会/组织方允许）可以直接使用这个模板。

下方即为应包含在代码中的 IO 模板，由于 Kattis 的原 Kattio 包含一些并不常用的功能，下方的模板经过了一些调整（原 Kattio 使用 MIT 作为协议）。

```
class Kattio extends PrintWriter {
    private BufferedReader r;
    private StringTokenizer st;
    // 标准 IO
    public Kattio() { this(System.in, System.out); }
    public Kattio(InputStream i, OutputStream o) {
        super(o);
        r = new BufferedReader(new InputStreamReader(i));
    }
    // 文件 IO
    public Kattio(String intput, String output) throws IOException {
        super(output);
        r = new BufferedReader(new FileReader(intput));
    }
    // 在没有其他输入时返回 null
    public String next() {
        try {
            while (st == null || !st.hasMoreTokens())
                st = new StringTokenizer(r.readLine());
            return st.nextToken();
        } catch (Exception e) {}
        return null;
    }
    public int nextInt() { return Integer.parseInt(next()); }
    public double nextDouble() { return Double.parseDouble(next()); }
    public long nextLong() { return Long.parseLong(next()); }
}
```

而下方代码简单展示了 Kattio 的使用：

```
class Test {
    public static void main(String[] args) {
        Kattio io = new Kattio();
        // 字符串输入
        String str = io.next();
        // int 输入
        int num = io.nextInt();
        // 输出
        io.println("Result");
        // 请确保关闭 IO 流以确保输出被正确写入
        io.close();
    }
}
```

使用 StreamTokenizer 作为输入

在某些情况使用 `StringTokenizer` 会导致 MLE（Memory Limit Exceeded，超过内存上限），此时我们需要使用 `StreamTokenizer` 作为输入。

```
import java.io.*;
public class Main {
    // IO 代码
```

```java
    public static StreamTokenizer in = new StreamTokenizer(new BufferedReader(new InputStreamReader(System.in), 32768));
    public static PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    public static double nextDouble() throws IOException { in.nextToken(); return in.nval; }
    public static float nextFloat() throws IOException { in.nextToken(); return (float)in.nval; }
    public static int nextInt() throws IOException { in.nextToken(); return (int)in.nval; }
    public static String next() throws IOException { in.nextToken(); return in.sval; }
    public static long nextLong() throws Exception { in.nextToken(); return (long)in.nval;}

    // 使用示例
    public static void main(String[] args) throws Exception {
        int n = nextInt();
        out.println(n);
        out.close();
    }
}
```

Kattio + StringTokenizer 的方法与 StreamTokenizer 的方法之间的分析与对比

1. StreamTokenizer 相较于 StringTokenizer 使用的内存较少，当 Java 标程 MLE 时可以尝试使用 StreamTokenizer，但是 StreamTokenizer 会丢失精度，读入部分数据时会出现问题；
    - StreamTokenizer 源码存在 Type，该 Type 根据你输入内容来决定类型，倘若你输入类似于 123oi 以**数字开头**的字符串，他会强制认为你的类型是 double 类型，因此在读入中以 double 类型去读 String 类型便会抛出异常；
    - StreamTokenizer 在读入 1e14 以上大小的数字会丢失精度；
2. 在使用 PrintWriter 情况下，需注意在程序结束最后 close() 关闭输出流或在需要输出的时候使用 flush() 清除缓冲区，否则内容将不会被写入到控制台/文件中。
3. Kattio 是继承自 PrintWriter 类，自身对象具有了 PrintWriter 的功能，因此可以直接调用 PrintWriter 类的函数输出，同时将 StringTokenizer 作为了自身的成员变量来修改。而第二种 Main 是同时将 StreamTokenizer 与 PrintWriter 作为了自身的成员变量，因此在使用上有些许差距。

综上所述，在大部分情况下，StringTokenizer 的使用处境要优越于 StreamTokenizer，在极端 MLE 的情况下可以尝试 StreamTokenizer，同时 int 范围以上的数据 StreamTokenizer 处理是无能为力的。

BigInteger 与数论

BigInteger 是 Java 提供的高精度计算类，可以很方便地解决高精度问题。

初始化

BigInteger 常用创建方式有如下二种：

```java
import java.io.PrintWriter;
import java.math.BigInteger;

```

```
class Main {
    static PrintWriter out = new PrintWriter(System.out);
    public static void main(String[] args) {
        BigInteger a = new BigInteger("12345678910");  // 将字符串以十进制的形式创建 BigInteger 对象
        out.println(a);  // a 的值为 12345678910
        BigInteger b = new BigInteger("1E", 16);  // 将字符串以指定进制的形式创建 BigInteger 对象
        out.println(b);  // b 的值为 30
        out.close();
    }
}
```

基本运算

以下均用 `this` 代替当前 `BigIntger`：

函数名	功能
`abs()`	返回 this 的绝对值
`negate()`	返回 - this
`add(BigInteger val)`	返回 this + val
`subtract(BigInteger val)`	返回 this - val
`multiply(BigInteger val)`	返回 this * val
`divide(BigInteger val)`	返回 this / val
`remainder(BigInteger val)`	返回 this % val
`mod(BigInteger val)`	返回 this mod val
`pow(int e)`	返回 $this^e$
`and(BigInteger val)`	返回 this & val
`or(BigInteger val)`	返回 this ‘
`not()`	返回 ~ this
`xor(BigInteger val)`	返回 this ^ val
`shiftLeft(int n)`	返回 this << n
`shiftRight(int n)`	返回 this >> n
`max(BigInteger val)`	返回 this 与 val 的较大值
`min(BigInteger val)`	返回 this 与 val 的较小值
`bitCount()`	返回 this 的二进制中不包括符号位的 1 的个数
`bitLength()`	返回 this 的二进制中不包括符号位的长度
`getLowestSetBit()`	返回 this 的二进制中最右边的位置
`compareTo(BigInteger val)`	比较 this 和 val 值大小

函数名	功能
toString()	返回 this 的 10 进制字符串表示形式
toString(int radix)。	返回 this 的 raidx 进制字符串表示形式

使用案例如下：

```java
import java.io.PrintWriter;
import java.math.BigInteger;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);
    static BigInteger a, b;

    static void abs() {
        out.println("abs:");
        a = new BigInteger("-123");
        out.println(a.abs());  // 输出 123
        a = new BigInteger("123");
        out.println(a.abs());  // 输出 123
    }

    static void negate() {
        out.println("negate:");
        a = new BigInteger("-123");
        out.println(a.negate());  // 输出 123
        a = new BigInteger("123");
        out.println(a.negate());  // 输出 -123
    }

    static void add() {
        out.println("add:");
        a = new BigInteger("123");
        b = new BigInteger("123");
        out.println(a.add(b));  // 输出 246
    }

    static void subtract() {
        out.println("subtract:");
        a = new BigInteger("123");
        b = new BigInteger("123");
        out.println(a.subtract(b));  // 输出 0
    }

    static void multiply() {
        out.println("multiply:");
        a = new BigInteger("12");
        b = new BigInteger("12");
        out.println(a.multiply(b));  // 输出 144
    }

    static void divide() {
        out.println("divide:");
```

```
        a = new BigInteger("12");
        b = new BigInteger("11");
        out.println(a.divide(b));  // 输出 1
    }

    static void remainder() {
        out.println("remainder:");
        a = new BigInteger("12");
        b = new BigInteger("10");
        out.println(a.remainder(b));  // 输出 2
        a = new BigInteger("-12");
        b = new BigInteger("10");
        out.println(a.remainder(b));  // 输出 -2
    }

    static void mod() {
        out.println("mod:");
        a = new BigInteger("12");
        b = new BigInteger("10");
        out.println(a.mod(b));  // 输出 2
        a = new BigInteger("-12");
        b = new BigInteger("10");
        out.println(a.mod(b));  // 输出 8
    }

    static void pow() {
        out.println("pow:");
        a = new BigInteger("2");
        out.println(a.pow(10));  // 输出 1024
    }

    static void and() {
        out.println("and:");
        a = new BigInteger("3");  // 11
        b = new BigInteger("5");  // 101
        out.println(a.and(b));  // 输出 1
    }

    static void or() {
        out.println("or:");
        a = new BigInteger("2");  // 10
        b = new BigInteger("5");  // 101
        out.println(a.or(b));  // 输出 7
    }

    static void not() {
        out.println("not:");
        a = new BigInteger("2147483647");  // 01111111 11111111 11111111 1111111
1
        out.println(a.not());  // 输出 -2147483648 二进制为: 10000000 00000000 00
000000 00000000
    }

    static void xor() {
```

```java
        out.println("xor:");
        a = new BigInteger("6");  // 110
        b = new BigInteger("5");  // 101
        out.println(a.xor(b));  // 011 输出 3
    }

    static void shiftLeft() {
        out.println("shiftLeft:");
        a = new BigInteger("1");
        out.println(a.shiftLeft(10));  // 输出 1024
    }

    static void shiftRight() {
        out.println("shiftRight:");
        a = new BigInteger("1024");
        out.println(a.shiftRight(8));  // 输出 4
    }

    static void max() {
        out.println("max:");
        a = new BigInteger("6");
        b = new BigInteger("5");
        out.println(a.max(b));  // 输出 6
    }

    static void min() {
        out.println("min:");
        a = new BigInteger("6");
        b = new BigInteger("5");
        out.println(a.min(b));  // 输出 5
    }

    static void bitCount() {
        out.println("bitCount:");
        a = new BigInteger("6");  // 110
        out.println(a.bitCount());  // 输出 2
    }

    static void bitLength() {
        out.println("bitLength:");
        a = new BigInteger("6");  // 110
        out.println(a.bitLength());  // 输出 3
    }

    static void getLowestSetBit() {
        out.println("getLowestSetBit:");
        a = new BigInteger("8");  // 1000
        out.println(a.getLowestSetBit());  // 输出 3
    }

    static void compareTo() {
        out.println("compareTo:");
        a = new BigInteger("8");
        b = new BigInteger("9");
```

```java
        out.println(a.compareTo(b));  // 输出 -1
        a = new BigInteger("8");
        b = new BigInteger("8");
        out.println(a.compareTo(b));  // 输出 0
        a = new BigInteger("8");
        b = new BigInteger("7");
        out.println(a.compareTo(b));  // 输出 1
    }

    static void toStringTest() {
        out.println("toString:");
        a = new BigInteger("15");
        out.println(a.toString());  // 输出 15
        out.println(a.toString(16));  // 输出 f
    }

    public static void main(String[] args) {
        abs();
        negate();
        add();
        subtract();
        multiply();
        divide();
        remainder();
        mod();
        pow();
        and();
        or();
        not();
        xor();
        shiftLeft();
        shiftRight();
        max();
        min();
        bitCount();
        bitLength();
        getLowestSetBit();
        compareTo();
        toStringTest();
        out.close();
    }
}
```

数学运算

以下均用 `this` 代替当前 `BigIntger`：

函数名	功能
`gcd(BigInteger val)`	返回 this 的绝对值与 val 的绝对值的最大公约数
`isProbablePrime(int val)`	返回一个表示 this 是否是素数的布尔值
`nextProbablePrime()`	返回第一个大于 this 的素数
`modPow(BigInteger b, BigInteger p)`	返回 this `^` b `mod` p

函数名	功能
modInverse(BigInteger p)	返回 a mod p 的乘法逆元

使用案例如下：

```
import java.io.PrintWriter;
import java.math.BigInteger;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);
    static BigInteger a, b, p;

    static void gcd() {  // 最大公约数
        a = new BigInteger("120032414321432144212100");
        b = new BigInteger("240231431243123412432140");
        out.println(String.format("gcd(%s,%s)=%s", a.toString(), b.toString(), a
.gcd(b).toString()));  // gcd(120032414321432144212100,240231431243123412432140)
=20
    }

    static void isPrime() {  // 基于米勒罗宾判定该数是否是素数，参数越大准确性越高
，复杂度越高。准确性为 (1-1/(val*2))
        a = new BigInteger("1200324143214321442127");
        out.println("a:" + a.toString());
        out.println(a.isProbablePrime(10) ? "a is prime" : "a is not prime");  /
/ a is not prime
    }

    static void nextPrime() {  // 找出该数的下一个素数
        a = new BigInteger("1200324143214321442127");
        out.println("a:" + a.toString());
        out.println(String.format("a nextPrime is %s", a.nextProbablePrime().toS
tring()));  // a nextPrime is 1200324143214321442199
    }

    static void modPow() {  // 快速幂，比正常版本要快，内部有数学优化
        a = new BigInteger("2");
        b = new BigInteger("10");
        p = new BigInteger("1000");
        out.println(String.format("a:%s b:%s p:%s", a, b, p));
        out.println(String.format("a^b mod p:%s", a.modPow(b, p).toString()));//
  24
    }

    static void modInverse() {  // 逆元
        a = new BigInteger("10");
        b = new BigInteger("3");
        out.println(a.modInverse(b));  // a ^ (p-2) mod p = 1
    }

    public static void main(String[] args) {
        gcd();
        isPrime();
```

```
        nextPrime();
        modPow();
        modInverse();
        out.close();
    }
}
```

关于米勒罗宾相关知识可以查阅Miller–Rabin 素性测试。

基本数据类型与包装数据类型

简介

由于基本类型没有面向对象的特征，为了他们参加到面向对象的开发中，Java 为八个基本类型提供了对应的包装类，分别是 `Byte`、`Double`、`Float`、`Integer`、`Long`、`Short`、`Character` 和 `Boolean`。两者之间的对应关系如下：

基本数据类型	包装数据类型
`byte`	`Byte`
`short`	`Short`
`boolean`	`Boolean`
`char`	`Character`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

区别

此处以 `int` 与 `Integer` 举例：

1. `Integer` 是 `int` 的包装类，`int` 则是 Java 的一种基本类型数据。
2. `Integer` 类型实例后才能使用，而 `int` 类型不需要。
3. `Integer` 实际对应的引用，当 `new` 一个 `Integer` 时，实际上生成了一个对象，而 `int` 则是直接存储数据。
4. `Integer` 的默认值是 `null`，可接受 `null` 和 `int` 类型的数据，`int` 默认值是 0，不能接受 `null` 类型的数据。
5. `Integer` 判定二个变量是否相同使用 `==` 可能会导致不正确的结果，只能使用 `equals()`，而 `int` 可以直接使用 `==`。

装箱与拆箱

此处以 `int` 与 `Integer` 举例：

`Integer` 的本质是对象，`int` 是基本类型，两个类型之间是不能直接赋值的。需要转换时，应将基础类型转换为包装类型，这种做法称为装箱，反过来则称为拆箱。

```
// 基本类型
int value1 = 1;
// 装箱转换为包装类型
Integer integer = Integer.valueOf(value1);
```

```java
// 拆箱转换为基本类型
int value2 = integer.intValue();
```

Java 5 引入了自动装箱拆箱机制：

```java
Integer integer = 1;
int value = integer;
```

" 注意"

虽然 JDK 增加了自动装箱拆箱的机制，但在声明变量时请选择合适的类型，因为包装类型 `Integer` 可以接受 `null`，而基本类型 `int` 不能接受 `null`。因此，对使用 `null` 值的包装类型进行拆箱操作时，会抛出异常。

```java
Integer integer = Integer.valueOf(null);
integer.intValue();  // 抛出 java.lang.NumberFormatException 异常

Integer integer = null;
integer.intValue();  // 抛出 java.lang.NullPointerException 异常
```

继承

基于已有的设计创造新的设计，就是面向对象程序设计中的继承。在继承中，新的类不是凭空产生的，而是基于一个已经存在的类而定义出来的。通过继承，新的类自动获得了基础类中所有的成员，包括成员变量和方法，包括各种访问属性的成员，无论是 `public` 还是 `private` 。显然，通过继承来定义新的类，远比从头开始写一个新的类要简单快捷和方便。继承是支持代码重用的重要手段之一。

在 Java 中，继承的关键字为 `extends`，且 Java 只支持单继承，但可以实现多接口。

在 Java 中，所有类都是 `Object` 类的子类。

子类继承父类，所有的父类的成员，包括变量和方法，都成为了子类的成员，除了构造方法。构造方法是父类所独有的，因为它们的名字就是类的名字，所以父类的构造方法在子类中不存在。除此之外，子类继承得到了父类所有的成员。

每个成员有不同的访问属性，子类继承得到了父类所有的成员，但是不同的访问属性使得子类在使用这些成员时有所不同：有些父类的成员直接成为子类的对外的界面，有些则被深深地隐藏起来，即使子类自己也不能直接访问。

下表列出了不同访问属性的父类成员在子类中的访问属性：

父类成员访问属性	在父类中的含义	在子类中的含义
`public`	对所有类开放	对所有类开放
`protected`	只有包内其它类、自己和子类可以访问	只有包内其它类、自己和子类可以访问
缺省（`default`）	只有包内其它类可以访问	如果子类与父类在同一个包内，只有包内其它类可以访问；否则相当于 `private`，不能访问
`private`	只有自己可以访问	不能访问

多态

在 Java 中当把一个对象赋值给一个变量时，对象的类型必须与变量的类型相匹配。但由于 Java 有继承的概念，便可重新定义为**一个变量可以保存其所声明的类型或该类型的任何子类型**。

如果一个类型实现了接口，也可以称之为该接口的子类型。

Java 中保存对象类型的变量是多态变量。「多态」这个术语（字面意思是许多形态）是指一个变量可以保存不同类型（即其声明的类型或任何子类型）的对象。

多态变量：

1. Java 的对象变量是多态的，它们能保存不止一种类型的对象。
2. 它们可以保存的是声明类型的对象，或声明类型子类的对象。
3. 当把子类的对象赋给父类的变量的时候，就发生了向上转型。

泛型

泛型指在类定义时不设置类中的属性或方法参数的具体类型，而是在使用（或创建对象）时再进行类型的定义。泛型本质是参数化类型，即所操作的数据类型被指定为一个参数。

泛型提供了编译时类型安全检测的机制，该机制允许编译时检测非法类型。

接口

简介

接口（英文：Interface）在 Java 中是一个抽象类型，是抽象方法的集合，通常以 `interface` 来声明。一个类通过实现接口的方式，从而来继承接口的抽象方法。

接口并不是类，编写接口的方式和类很相似，但是它们属于不同的概念。类描述对象的属性和方法。接口则包含类要实现的方法。

除非实现接口的类是抽象类，否则该类要定义接口中的所有方法。

接口无法被实例化，但是可以被实现。一个实现接口的类，必须实现接口内所描述的所有方法，否则就必须声明为抽象类。另外，在 Java 中，接口类型可用来声明一个变量，他们可以成为一个空指针，或是被绑定在一个以此接口实现的对象。

与类的区别

1. 接口不能用于实例化对象。
2. 接口没有构造方法。
3. 接口中所有的方法必须是抽象方法，Java 8 之后接口中可以使用 `default` 关键字修饰的非抽象方法。
4. 接口不能包含成员变量，除了 static 和 final 变量。
5. 接口不是被类继承了，而是要被类实现。
6. 接口支持多继承，类不支持多继承。

声明

```
[可见度] interface 接口名称 [extends 其他的接口名] {
        // 声明变量
        // 抽象方法
}
```

实现

```
...implements 接口名称[, 其他接口名称, 其他接口名称..., ...] ...
```

Lambda 表达式

简介

lambda 表达式也可称为闭包，是 Java 8 的最重要的新特性。

lambda 表达式允许把函数作为一个方法的参数（函数作为参数传递进方法中）。

使用 lambda 表达式可以使代码变的更加简洁紧凑。

语法

可选类型声明：不需要声明参数类型，编译器可以统一识别参数值。

可选的参数圆括号：一个参数无需定义圆括号，但多个参数需要定义圆括号。

可选的大括号：如果主体包含了一个语句，就不需要使用大括号。

可选的返回关键字：如果主体只有一个表达式返回值则编译器会自动返回值，大括号需要指定表达式返回了一个数值。

lambda 表达式声明方式如下：

以字符串数组按长度排序的自定义比较器为例：

1. 参数，箭头，一个表达式。

```
import java.util.Arrays;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);

    public static void main(String[] args) {
        String[] plants = {"Mercury", "venus", "Earth", "Mars", "Jupiter", "Satu
rn", "Uranus", "Neptune"};
        Arrays.sort(plants, (String first, String second) -> (first.length() - s
econd.length()));
        for (String word : plants) {
            out.print(word + " ");
        }
        out.close();
    }
}
```

2. 参数，箭头，多条语句。

```
import java.io.PrintWriter;
import java.util.Arrays;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);

    public static void main(String[] args) {
        String[] plants = {"Mercury", "venus", "Earth", "Mars", "Jupiter", "Satu
rn", "Uranus", "Neptune"};
        Arrays.sort(plants, (first, second) ->
        {
            // 形参不写类型，可以从上下文判断出
            int result = first.length() - second.length();
```

```
            return result;
        });
        for (String word : plants) {
            out.print(word + " ");
        }
        out.close();
    }
}
```

-> 是一个推导符号，表示前面的括号接收到参数，推导后面的返回值（其实就是传递了方法）。

3. 常用形式：

```
// 1. 不需要参数，返回值为 5
() -> 5

// 2. 接收一个参数（数字类型），返回其 2 倍的值
x -> 2 * x

// 3. 接受 2 个参数（数字）并返回他们的差值
(x, y) -> x - y

// 4. 接收 2 个 int 类型整数并返回他们的和
(int x, int y) -> x + y

// 5. 接受一个 String 对象并在控制台打印，不返回任何值（看起来像是返回 void）
(String s) -> System.out.print(s)
```

函数式接口

1. 是一个接口，符合 Java 接口定义。
2. 只包含一个抽象方法的接口。
3. 因为只有一个未实现的方法，所以 lambda 表达式可以自动填上去。

函数式接口使用方式如下：

1. 输出长度为 2 的倍数的字符串。

```
import java.io.PrintWriter;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);

    public static void main(String[] args) {
        String[] plants = {"Mercury", "venus", "Earth", "Mars", "Jupiter", "Satu
rn", "Uranus", "Neptune"};
        Test test = s -> {  // lambda 表达式作为函数式接口的实例
            if (s.length() % 2 == 0) {
                return true;
            }
            return false;
        };
        for (String word : plants) {
            if (test.check(word)) {
```

```java
                out.print(word + " ");
            }
        }
        out.close();
    }
}

interface Test {
    public boolean check(String s);
}
```

2. 实现加减乘除四则运算。

```java
import java.io.PrintWriter;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);

    public static double calc(double a, double b, Calculator util) {
        return util.operation(a, b);
    }

    public static void main(String[] args) {
        Calculator util[] = new Calculator[4];  // 定义函数式接口数组
        util[0] = (a, b) -> a + b;
        util[1] = (a, b) -> a - b;
        util[2] = (a, b) -> a * b;
        util[3] = (a, b) -> a / b;
        double a = 20, b = 15;
        for (Calculator c : util) {
            System.out.println(calc(a, b, c));
        }
        out.close();
    }
}

interface Calculator {
    public double operation(double a, double b);
}
```

Collection

Collection 是 Java 中的接口，被多个泛型容器接口所实现。在这里，Collection 是指代存放对象类型的数据结构。

Java 中的 Collection 元素类型定义时必须为对象，不能为基本数据类型。

以下内容用法均基于 Java 里多态的性质，均是以实现接口的形式出现。

常用的接口包括 List、Queue、Set 和 Map。

容器定义

1. 当定义泛型容器类时，需要在定义时指定数据类型。

   例如：

```
List<Integer> list1 = new LinkedList<>();
```

2. 倘若不指定数据类型，而当成 `Object` 类型随意添加数据，在 Java 8 中虽能编译通过，但会有很多警告风险。

   例如:

```
List list = new ArrayList<>();
list.add(1);
list.add(true);
list.add(1.01);
list.add(1L);
list.add("I am String");
```

因此，如果没有特殊需求的话不推荐第 2 种行为，编译器无法帮忙检查存入的数据是否安全。`list.get(index)` 取值时无法明确数据的类型（取到的数据类型都为 `Object`），需要手动转回原来的类型，稍有不慎可能出现误转型异常。

如果是明确了类型如 `List<Integer>`，此时编译器会检查放入的数据类型，只能放入整数的数据。声明集合变量时只能使用包装类型 `List<Integer>` 或者自定义的 `Class`，而不能是基本类型如 `List<int>`。

List

ArrayList

`ArrayList` 是支持可以根据需求动态生长的数组，初始长度默认为 10。如果超出当前长度便扩容 $\frac{3}{2}$。

初始化

```
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.List;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);

    public static void main(String[] args) {
        List<Integer> list1 = new ArrayList<>();  // 创建一个名字为 list1 的可自
增数组, 初始长度为默认值 (10)
        List<Integer> list2 = new ArrayList<>(30);  // 创建一个名字为 list2 的可自
增数组, 初始长度为 30
        List<Integer> list3 = new ArrayList<>(list2);  // 创建一个名字为 list3 的
可自增数组, 使用 list2 里的元素和 size 作为自己的初始值
    }
}
```

LinkedList

`LinkedList` 是双链表。

初始化

```
import java.io.PrintWriter;
import java.util.LinkedList;
import java.util.List;

public class Main {
```

```java
    static PrintWriter out = new PrintWriter(System.out);

    public static void main(String[] args) {
        List<Integer> list1 = new LinkedList<>();  // 创建一个名字为 list1 的双链
表
        List<Integer> list2 = new LinkedList<>(list1);  // 创建一个名字为 list2
的双链表，将 list1 内所有元素加入进来
    }
}
```

常用方法

以下均用 `this` 代替当前 `List<Integer>`：

函数名	功能
`size()`	返回 this 的长度
`add(Integer val)`	在 this 尾部插入一个元素
`add(int idx, Integer e)`	在 this 指定位置插入一个元素
`get(int idx)`	返回 this 中第 idx 位置的值，若越界则抛出异常
`set(int idx, Integer e)`	修改 this 中第 idx 位置的值

使用案例及区别对比：

```java
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);
    static List<Integer> array = new ArrayList<>();
    static List<Integer> linked = new LinkedList<>();

    static void add() {
        array.add(1);  // 时间复杂度为 O(1)
        linked.add(1);  // 时间复杂度为 O(1)
    }

    static void get() {
        array.get(10);  // 时间复杂度为 O(1)
        linked.get(10);  // 时间复杂度为 O(11)
    }

    static void addIdx() {
        array.add(0, 2);  // 最坏情况下时间复杂度为 O(n)
        linked.add(0, 2);  // 最坏情况下时间复杂度为 O(n)
    }

    static void size() {
        array.size();  // 时间复杂度为 O(1)
        linked.size();  // 时间复杂度为 O(1)
```

```java
    }

    static void set() {  // 该方法返回值为原本该位置元素的值
        array.set(0, 1);  // 时间复杂度为 O(1)
        linked.set(0, 1);  // 最坏时间复杂度为 O(n)
    }

}
```

遍历

```java
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);
    static List<Integer> array = new ArrayList<>();
    static List<Integer> linked = new LinkedList<>();

    static void function1() {  // 朴素遍历
        for (int i = 0; i < array.size(); i++) {
            out.println(array.get(i));  // 遍历自增数组，复杂度为 O(n)
        }
        for (int i = 0; i < linked.size(); i++) {
            out.println(linked.get(i));  // 遍历双链表，复杂度为 O(n^2)，因为 Link
edList 的 get(i) 复杂度是 O(i)
        }
    }

    static void function2() {  // 增强 for 循环遍历
        for (int e : array) {
            out.println(e);
        }
        for (int e : linked) {
            out.println(e);  // 复杂度均为 O(n)
        }
    }

    static void function3() {  // 迭代器遍历
        Iterator<Integer> iterator1 = array.iterator();
        Iterator<Integer> iterator2 = linked.iterator();
        while (iterator1.hasNext()) {
            out.println(iterator1.next());
        }
        while (iterator2.hasNext()) {
            out.println(iterator2.next());
        }  // 复杂度均为 O(n)
    }

}
```

> " 注意"
>
> 不要在 `for/foreach` 遍历 `List` 的过程中删除其中的元素，否则会抛出异常。
>
> 原因也很简单，`list.size()` 改变了，但在循环中已循环的次数却是没有随之变化。原来预计在下一个 `index` 的数据因为删除的操作变成了当前 `index` 的数据，运行下一个循环时操作的会变为原来预计在下下个 `index` 的数据，最终会导致操作的数据不符合预期。

Queue

LinkedList

可以使用 `LinkedList` 实现普通队列，底层是链表模拟队列。

初始化

```java
Queue<Integer> q = new LinkedList<>();
```

`LinkedList` 底层实现了 `List` 接口与 `Deque` 接口，而 `Deque` 接口继承自 `Queue` 接口，所以 `LinkedList` 可以同时实现 `List` 与 `Queue` 。

ArrayDeque

可以使用 `ArrayDeque` 实现普通队列，底层是数组模拟队列。

初始化

```java
Queue<Integer> q = new ArrayDeque<>();
```

`ArrayDeque` 底层实现了 `Deque` 接口，而 `Deque` 接口继承自 `Queue` 接口，所以 `ArrayDeque` 可以实现 `Queue`。

LinkedList 与 ArrayDeque 在实现 Queue 接口上的区别

1. **数据结构**：在数据结构上，`ArrayDeque` 和 `LinkedList` 都实现了 Java Deque 双端队列接口。但 `ArrayDeque` 没有实现了 Java List 列表接口，所以不具备根据索引位置操作的行为。
2. **线程安全**：`ArrayDeque` 和 `LinkedList` 都不考虑线程同步，不保证线程安全。
3. **底层实现**：在底层实现上，`ArrayDeque` 是基于动态数组的，而 `LinkedList` 是基于双向链表的。
4. **在遍历速度上**：`ArrayDeque` 是一块连续内存空间，基于局部性原理能够更好地命中 CPU 缓存行，而 `LinkedList` 是离散的内存空间对缓存行不友好。
5. **在操作速度上**：`ArrayDeque` 和 `LinkedList` 的栈和队列行为都是 $O(1)$ 时间复杂度，`ArrayDeque` 的入栈和入队有可能会触发扩容，但从均摊分析上看依然是 $O(1)$ 时间复杂度。
6. **额外内存消耗上**：`ArrayDeque` 在数组的头指针和尾指针外部有闲置空间，而 `LinkedList` 在节点上增加了前驱和后继指针。

PriorityQueue

`PriorityQueue` 是优先队列，默认是小根堆。

初始化

```java
Queue<Integer> q1 = new PriorityQueue<>();  // 小根堆
Queue<Integer> q2 = new PriorityQueue<>((x, y) -> {return y - x;});  // 大根堆
```

常用方法

以下均用 `this` 代替当前 `Queue<Integer>` :

函数名	功能
size()	返回 this 的长度
add(Integer val)	入队
offer(Integer val)	入队
isEmpty()	判断队列是否为空，为空则返回 true
peek()	返回队头元素
poll()	返回队头元素并删除

使用案例及区别对比：

```java
import java.io.PrintWriter;
import java.util.LinkedList;
import java.util.PriorityQueue;
import java.util.Queue;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);
    static Queue<Integer> q1 = new LinkedList<>();
    static Queue<Integer> q2 = new PriorityQueue<>();

    static void add() {  // add 和 offer 功能上没有差距，区别是是否会抛出异常
        q1.add(1);  // 时间复杂度为 O(1)
        q2.add(1);  // 时间复杂度为 O(logn)
    }

    static void isEmpty() {
        q1.isEmpty();  // 时间复杂度为 O(1)
        q2.isEmpty();  // 空间复杂度为 O(1)
    }

    static void size() {
        q1.size();  // 时间复杂度为 O(1)
        q2.size();  // 返回 q2 的长度
    }

    static void peek() {
        q1.peek();  // 时间复杂度为 O(1)
        q2.peek();  // 时间复杂度为 O(logn)
    }

    static void poll() {
        q1.poll();  // 时间复杂度为 O(1)
        q2.poll();  // 时间复杂度为 O(logn)
    }
}
```

遍历

```java
import java.io.PrintWriter;
import java.util.LinkedList;
import java.util.PriorityQueue;
import java.util.Queue;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);
    static Queue<Integer> q1 = new LinkedList<>();
    static Queue<Integer> q2 = new PriorityQueue<>();

    static void test() {
        while (!q1.isEmpty()) {  // 复杂度为 O(n)
            out.println(q1.poll());
        }
        while (!q2.isEmpty()) {  // 复杂度为 O(nlogn)
            out.println(q2.poll());
        }
    }

}
```

Deque

Deque 是 Java 中的双端队列，我们通常用其进行队列的操作以及栈的操作。

主要函数

以下均用 this 代替当前 Deque<Integer>：

函数名	功能
push(Integer val)	将一个元素从队头加入 this，等效于 addFirst
pop()	将队头元素删除，等效于 removeFirst
addFirst(Integer val)	将一个元素从队头加入 this
removeFirst()	将队头元素删除，并返回该元素
addLast(Integer val)	将一个元素从队尾加入 this
removeLast()	将队尾元素删除，并返回该元素
offerFirst(Integer val)	将一个元素从队头加入 this
pollFirst()	将队头元素删除，并返回该元素
offerLast(Integer val)	将一个元素从队尾加入 this
pollLast()	将队尾元素删除，并返回该元素
add(Integer val)	将一个元素从队尾加入 this
offer(Integer val)	将一个元素从队尾加入 this
poll()	将队头元素删除，并返回该元素
remove()	将队头元素删除，并返回该元素

函数名	功能
peekFirst()	返回队头元素
peekLast()	返回队尾元素

add、remove 操作在遇到异常时会抛出异常，而 offer、poll 不会抛出异常。

栈的操作

```java
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {
    static Deque<Integer> stack = new ArrayDeque<>();
    static int[] a = {1, 2, 3, 4, 5};

    public static void main(String[] args) {
        for (int v : a) {
            stack.push(v);
        }
        while (!stack.isEmpty()) { //输出 5 4 3 2 1
            System.out.println(stack.pop());
        }
    }
}
```

双端队列的操作

```java
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {
    static Deque<Integer> deque = new ArrayDeque<>();

    static void insert() {
        deque.addFirst(1);
        deque.addFirst(2);
        deque.addLast(3);
        deque.addLast(4);
    }

    public static void main(String[] args) {
        insert();
        while (!deque.isEmpty()) { //输出 2 1 3 4
            System.out.println(deque.poll());
        }
        insert();
        while (!deque.isEmpty()) { //输出 4 3 1 2
            System.out.println(deque.pollLast());
        }
```

```
    }
}
```

Set

Set 是保持容器中的元素不重复的一种数据结构。

HashSet

随机位置插入的 Set。

初始化

```java
Set<Integer> s1 = new HashSet<>();
```

LinkedHashSet

保持插入顺序的 Set。

初始化

```java
Set<Integer> s2 = new LinkedHashSet<>();
```

TreeSet

保持容器中元素有序的 Set，默认为升序。

初始化

```java
Set<Integer> s3 = new TreeSet<>();
Set<Integer> s4 = new TreeSet<>((x, y) -> {return y - x;});  // 降序
```

TreeSet 的更多使用

这些方法是 TreeSet 新创建并实现的，我们无法使用 Set 接口调用以下方法，因此我们创建方式如下:

```java
TreeSet<Integer> s3 = new TreeSet<>();
TreeSet<Integer> s4 = new TreeSet<>((x, y) -> {return y - x;});  // 降序
```

以下均用 this 代替当前 TreeSet<Integer> :

函数名	功能
first()	返回 this 中第一个元素，无则返回 null
last()	返回 this 中最后一个元素，无则返回 null
floor(Integer val)	返回集合中 <=val 的第一个元素，无则返回 null
ceiling(Integer val)	返回集合中 >=val 的第一个元素，无则返回 null
higher(Integer val)	返回集合中 >val 的第一个元素，无则返回 null
lower(Integer val)	返回集合中 <val 的第一个元素，无则返回 null
pollFirst()	返回并删除 this 中第一个元素，无则返回 null
pollLast()	返回并删除 this 中最后一个元素，无则返回 null

代码示例:

```
import java.util.TreeSet;

public class Main {
    static int[] a = {4,7,1,2,3,6};

    public static void main(String[] args) {
        TreeSet<Integer> set = new TreeSet<>();
        for(int v:a) {
            set.add(v);
        }
        Integer a2 = set.first();
        System.out.println(a2); //返回 1
        Integer a3 = set.last();
        System.out.println(a3); //返回 7
        Integer a4 = set.floor(5);
        System.out.println(a4); //返回 4
        Integer a5 = set.ceiling(6);
        System.out.println(a5); //返回 6
        Integer a6 = set.higher(7);
        System.out.println(a6); //返回 null
        Integer a7 = set.lower(2);
        System.out.println(a7); //返回 1
        Integer a8 = set.pollFirst();
        System.out.println(a8); //返回 1
        Integer a9 = set.pollLast();
        System.out.println(a9); //返回 7
    }
}
```

Set 常用方法

以下均用 `this` 代替当前 `Set<Integer>`：

函数名	功能
`size()`	返回 this 的长度
`add(Integer val)`	插入一个元素进 this
`contains(Integer val)`	判断 this 中是否有元素 val
`addAll(Collection e)`	将一个容器里的所有元素添加进 this
`retainAll(Collection e)`	将 this 改为两个容器内相同的元素
`removeAll(Collection e)`	将 this 中与 e 相同的元素删除

使用案例：求并集、交集、差集。

```
import java.io.PrintWriter;
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.Set;
```

```
public class Main {
    static PrintWriter out = new PrintWriter(System.out);
    static Set<Integer> s1 = new HashSet<>();
    static Set<Integer> s2 = new LinkedHashSet<>();

    static void add() {
        s1.add(1);
    }

    static void contains() {  // 判断 set 中是否有元素值为 2，有则返回 true，否则返
回 false
        s1.contains(2);
    }

    static void test1() {  // s1 与 s2 的并集
        Set<Integer> res = new HashSet<>();
        res.addAll(s1);
        res.addAll(s2);
    }

    static void test2() {  // s1 与 s2 的交集
        Set<Integer> res = new HashSet<>();
        res.addAll(s1);
        res.retainAll(s2);
    }

    static void test3() {  // 差集：s1 - s2
        Set<Integer> res = new HashSet<>();
        res.addAll(s1);
        res.removeAll(s2);
    }
}
```

遍历

```
import java.io.PrintWriter;
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.Set;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);
    static Set<Integer> s1 = new HashSet<>();
    static Set<Integer> s2 = new LinkedHashSet<>();

    static void test() {
        for (int key : s1) {
            out.println(key);
        }
        out.close();
    }
}
```

Map

Map 是维护键值对 <Key, Value> 的一种数据结构，其中 Key 唯一。

HashMap

随机位置插入的 Map。

初始化

```
Map<Integer, Integer> map1 = new HashMap<>();
```

LinkedHashMap

保持插入顺序的 Map。

初始化

```
Map<Integer, Integer> map2 = new LinkedHashMap<>();
```

TreeMap

保持 key 有序的 Map，默认升序。

初始化

```
Map<Integer, Integer> map3 = new TreeMap<>();
Map<Integer, Integer> map4 = new TreeMap<>((x, y) -> {return y - x;});  // 降序
```

常用方法

以下均用 this 代替当前 Map<Integer, Integer>:

函数名	功能
put(Integer key, Integer value)	插入一个元素进 this
size()	返回 this 的长度
containsKey(Integer val)	判断 this 中是否有元素 key 为 val
get(Integer key)	将 this 中对应的 key 的 value 返回
keySet	将 this 中所有元素的 key 作为集合返回

使用案例:

```
import java.io.PrintWriter;
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.TreeMap;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);
```

```
    static Map<Integer, Integer> map1 = new HashMap<>();
    static Map<Integer, Integer> map2 = new LinkedHashMap<>();
    static Map<Integer, Integer> map3 = new TreeMap<>();
    static Map<Integer, Integer> map4 = new TreeMap<>((x,y)->{return y-x;});

    static void put(){  // 将 key 为 1、value 为 1 的元素返回
        map1.put(1, 1);
    }
    static void get(){  // 将 key 为 1 的 value 返回
        map1.get(1);
    }
    static void containsKey(){  // 判断是否有 key 为 1 的键值对
        map1.containsKey(1);
    }
    static void KeySet(){
        map1.keySet();
    }
}
```

遍历

```
import java.io.PrintWriter;
import java.util.HashMap;
import java.util.Map;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);

    static Map<Integer, Integer> map1 = new HashMap<>();

    static void print() {
        for (int key : map1.keySet()) {
            out.println(key + " " + map1.get(key));
        }
    }
}
```

当然，在面向对象的世界里，你的参数是什么都可以，包括 `Collection` 与自定义类型。

例如 `Map` 也可以定义为：

```
Map<String, Set<Integer>> map = new HashMap<>();
```

Arrays

`Arrays` 是 `java.util` 中对数组操作的一个工具类。方法均为静态方法，可使用类名直接调用。

Arrays.sort()

`Arrays.sort()` 是对数组进行的排序的方法，主要重载方法如下：

```
import java.util.Arrays;
import java.util.Comparator;

public class Main {
```

```java
    static int[] a = new int[10];
    static Integer[] b = new Integer[10];
    static int firstIdx, lastIdx;

    public static void main(String[] args) {
        Arrays.sort(a);  // 1
        Arrays.sort(a, firstIdx, lastIdx);  // 2
        Arrays.sort(b, new Comparator<Integer>() {  // 3
            @Override
            public int compare(Integer o1, Integer o2) {
                return o2 - o1;
            }
        });
        Arrays.sort(b, firstIdx, lastIdx, new Comparator<Integer>() {  // 4
            @Override
            public int compare(Integer o1, Integer o2) {
                return o2 - o1;
            }
        });
        // 由于 Java 8 后有 Lambda 表达式，第三个重载及第四个重载亦可写为
        Arrays.sort(b, (x, y) -> {  // 5
            return y - x;
        });
        Arrays.sort(b, (x, y) -> {  // 6
            return y - x;
        });
    }
}
```

序号所对应的重载方法含义：

1. 对数组 a 进行排序，默认升序。
2. 对数组 a 的指定位置进行排序，默认升序，排序区间为左闭右开 `[firstIdx, lastIdx)`。
3. 对数组 a 以自定义的形式排序，第二个参数 - 第一个参数为降序，第一个参数 - 第二个参数为升序，当自定义排序比较器时，数组元素类型必须为对象类型。
4. 对数组 a 的指定位置进行自定义排序，排序区间为左闭右开 `[firstIdx, lastIdx)`，当自定义排序比较器时，数组元素类型必须为对象类型。
5. 和 3 同理，用 Lambda 表达式优化了代码长度。
6. 和 4 同理，用 Lambda 表达式优化了代码长度。

`Arrays.sort()` 底层函数：

1. 当你 `Arrays.sort` 的参数数组元素类型为基本数据类型（`byte`、`short`、`char`、`int`、`long`、`double`、`float`）时，默认为 `DualPivotQuicksort`（双轴快排），复杂度最坏可以达到 $O(n^2)$。
2. 当你 `Arrays.sort` 的参数数组元素类型为非基本数据类型时，则默认为 `legacyMergeSort` 和 `TimSort`（归并排序），复杂度为 $O(n \log n)$。

可以通过如下代码验证：

"Codeforces 1646B - Quality vs Quantity[3]"

题意概要：有 n 个数，你需要将其分为 2 组，是否能存在 1 组的长度小于另 1 组的同时和大于它。

" 例题代码"

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.Arrays;
import java.util.StringTokenizer;

public class Main {
    static class FastReader {
        StringTokenizer st;
        BufferedReader br;

        public FastReader() {
            br = new BufferedReader(new InputStreamReader(System.in));
        }

        String next() {
            while (st == null || !st.hasMoreElements()) {
                try {
                    st = new StringTokenizer(br.readLine());
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            return st.nextToken();
        }

        int nextInt() {
            return Integer.parseInt(next());
        }

        long nextLong() {
            return Long.parseLong(next());
        }

        double nextDouble() {
            return Double.parseDouble(next());
        }

        String nextLine() {
            String str = "";
            try {
                str = br.readLine();
            } catch (IOException e) {
                e.printStackTrace();
            }
            return str;
        }
    }

    static PrintWriter out = new PrintWriter(System.out);
    static FastReader in = new FastReader();
```

```java
    static void solve() {
        int n = in.nextInt();
        Integer[] a = new Integer[n + 10];
        for (int i = 1; i <= n; i++) {
            a[i] = in.nextInt();
        }
        Arrays.sort(a, 1, n + 1);
        long left = a[1];
        long right = 0;
        int x = n;
        for (int i = 2; i < x; i++, x--) {
            left = left + a[i];
            right = right + a[x];
            if (right > left) {
                out.println("YES");
                return;
            }
        }
        out.println("NO");
    }

    public static void main(String[] args) {
        int t = in.nextInt();
        while (t-- > 0) {
            solve();
        }
        out.close();
    }
}
```

如果你将以上代码的 a 数组类型由 `Integer` 修改为 `int`，会导致 TLE。

Arrays.binarySearch()

`Arrays.binarySearch()` 是对数组连续区间进行二分搜索的方法，前提是数组必须有序，时间复杂度为 $O(\log_n)$，主要重载方法如下：

```java
import java.util.Arrays;

public class Main {
    static int[] a = new int[10];
    static Integer[] b = new Integer[10];
    static int firstIdx, lastIdx;
    static int key;

    public static void main(String[] args) {
        Arrays.binarySearch(a, key);  // 1
        Arrays.binarySearch(a, firstIdx, lastIdx, key);  // 2
    }
}
```

源码如下：

```java
 private static int binarySearch0(int[] a, int fromIndex, int toIndex, int key)
{
```

```java
        int low = fromIndex;
        int high = toIndex - 1;

        while (low <= high) {
            int mid = (low + high) >>> 1;
            int midVal = a[mid];

            if (midVal < key)
                low = mid + 1;
            else if (midVal > key)
                high = mid - 1;
            else
                return mid; // key found
        }
        return -(low + 1);  // key not found.
    }
```

序号所对应的重载方法含义：

1. 从数组 a 中二分查找是否存在 key，如果存在，便返回其下标。若不存在，则返回一个负数。
2. 从数组 a 中二分查找是否存在 key，如果存在，便返回其下标，搜索区间为左闭右开 [firstIdx,lastIdx)。若不存在，则返回一个负数。

Arrays.fill()

Arrays.fill() 方法将数组中连续位置的元素赋值为统一元素。其接受的参数为数组、fromIndex、toIndex 和需要填充的数。方法执行后，数组左闭右开区间 [firstIdx,lastIdx) 内的所有元素的值均为需要填充的数。

Collections

Collections 是 java.util 中对集合操作的一个工具类。方法均为静态方法，可使用类名直接调用。

Collections.sort()

Collections.sort() 底层原理为将其中所有元素转化为数组调用 Arrays.sort()，完成排序后再赋值给原本的集合。又因为 Java 中 Collection 的元素类型均为对象类型，所以始终是归并排序去处理。

该方法无法对集合指定区间排序。

底层源码：

```java
    default void sort(Comparator<? super E> c) {
          Object[] a = this.toArray();
          Arrays.sort(a, (Comparator) c);
          ListIterator<E> i = this.listIterator();
          for (Object e : a) {
              i.next();
              i.set((E) e);
          }
    }
```

Collections.binarySearch()

Collections.binarySearch() 是对集合中指定区间进行二分搜索，功能与 Arrays.binarySearch() 相同。

```
Collections.binarySearch(list, key);
```

该方法无法对指定区间进行搜索。

Collections.swap()

`Collections.swap()` 的功能是交换集合中指定二个位置的元素。

```
Collections.swap(list, i, j);
```

其他

1. -0.0 != 0.0

在 Java 中，如果单纯是数值类型，-0.0 = 0.0 。若是对象类型，则 -0.0 != 0.0 。倘若你尝试用 `Set` 统计斜率数量时，这个问题就会带来麻烦。提供的解决方式是在所有的斜率加入 `Set` 前将值增加 0.0。

```java
import java.io.PrintWriter;

public class Main {
    static PrintWriter out = new PrintWriter(System.out);

    static void A() {
        Double a = 0.0;
        Double b = -0.0;
        out.println(a.equals(b));  // false
    }

    static void B() {
        Double a = 0.0;
        Double b = -0.0 + 0.0;
        out.println(a.equals(b));  // true
    }

    static void C() {
        double a = 0.0;
        double b = -0.0;
        out.println(a == b);  // true
    }


    public static void main(String[] args) {
        A();
        B();
        C();
        out.close();
    }
}
```

参考资料

[1] Input & Output - USACO Guide

[2] Kattio.java

[3] Codeforces 1646B - Quality vs Quantity

第 5 章

算法基础

5.1 算法基础简介

本章主要介绍一些基础算法，为之后的进阶内容做铺垫。

一方面，这些内容可以让初学者对 OI 的一些思想有初步的认识；另一方面，本章介绍的大部分算法还会在以后的进阶内容中得到运用。

5.2 复杂度

> **Authors:** linehk, persdre

时间复杂度和空间复杂度是衡量一个算法效率的重要标准。

基本操作数

同一个算法在不同的计算机上运行的速度会有一定的差别，并且实际运行速度难以在理论上进行计算，实际去测量又比较麻烦，所以我们通常考虑的不是算法运行的实际用时，而是算法运行所需要进行的基本操作的数量。

在普通的计算机上，加减乘除、访问变量（基本数据类型的变量，下同）、给变量赋值等都可以看作基本操作。

对基本操作的计数或是估测可以作为评判算法用时的指标。

时间复杂度

定义

衡量一个算法的快慢，一定要考虑数据规模的大小。所谓数据规模，一般指输入的数字个数、输入中给出的图的点数与边数等等。一般来说，数据规模越大，算法的用时就越长。而在算法竞赛中，我们衡量一个算法的效率时，最重要的不是看它在某个数据规模下的用时，而是看它的用时随数据规模而增长的趋势，即**时间复杂度**。

引入

考虑用时随数据规模变化的趋势的主要原因有以下几点：

1. 现代计算机每秒可以处理数亿乃至更多次基本运算，因此我们处理的数据规模通常很大。如果算法 A 在规模为 n 的数据上用时为 $100n$ 而算法 B 在规模为 n 的数据上用时为 n^2，在数据规模小于 100 时算法 B 用时更短，

但在一秒钟内算法 A 可以处理数百万规模的数据，而算法 B 只能处理数万规模的数据。在允许算法执行时间更久时，时间复杂度对可处理数据规模的影响就会更加明显，远大于同一数据规模下用时的影响。

2. 我们采用基本操作数来表示算法的用时，而不同的基本操作实际用时是不同的，例如加减法的用时远小于除法的用时。计算时间复杂度而忽略不同基本操作之间的区别以及一次基本操作与十次基本操作之间的区别，可以消除基本操作间用时不同的影响。

当然，算法的运行用时并非完全由输入规模决定，而是也与输入的内容相关。所以，时间复杂度又分为几种，例如：

1. 最坏时间复杂度，即每个输入规模下用时最长的输入对应的时间复杂度。在算法竞赛中，由于输入可以在给定的数据范围内任意给定，我们为保证算法能够通过某个数据范围内的任何数据，一般考虑最坏时间复杂度。
2. 平均（期望）时间复杂度，即每个输入规模下所有可能输入对应用时的平均值的复杂度（随机输入下期望用时的复杂度）。

所谓「用时随数据规模而增长的趋势」是一个模糊的概念，我们需要借助下文所介绍的**渐进符号**来形式化地表示时间复杂度。

渐进符号的定义

渐进符号是函数的阶的规范描述。简单来说，渐进符号忽略了一个函数中增长较慢的部分以及各项的系数（在时间复杂度相关分析中，系数一般被称作「常数」），而保留了可以用来表明该函数增长趋势的重要部分。

一个简单的记忆方法是，含等于（非严格）用大写，不含等于（严格）用小写，相等是 Θ，小于是 O，大于是 Ω。大 O 和小 o 原本是希腊字母 Omicron，由于字形相同，也可以理解为拉丁字母的大 O 和小 o。

在英文中，词根「-micro-」和「-mega-」常用于表示 10 的负六次方（百万分之一）和六次方（百万），也表示「小」和「大」。小和大也是希腊字母 Omicron 和 Omega 常表示的含义。

大 Θ 符号

对于函数 $f(n)$ 和 $g(n)$，$f(n)=\Theta(g(n))$，当且仅当 $\exists c_1,c_2,n_0>0$，使得 $\forall n\geq n_0, 0\leq c_1\cdot g(n)\leq f(n)\leq c_2\cdot g(n)$。

也就是说，如果函数 $f(n)=\Theta(g(n))$，那么我们能找到两个正数 c_1,c_2 使得 $f(n)$ 被 $c_1\cdot g(n)$ 和 $c_2\cdot g(n)$ 夹在中间。

例如，$3n^2+5n-3=\Theta(n^2)$，这里的 c_1,c_2,n_0 可以分别是 $2,4,100$。$n\sqrt{n}+n\log^5 n+m\log m+nm=\Theta(n\sqrt{n}+m\log m+nm)$，这里的 c_1,c_2,n_0 可以分别是 $1,2,100$。

大 O 符号

Θ 符号同时给了我们一个函数的上下界，如果只知道一个函数的渐进上界而不知道其渐进下界，可以使用 O 符号。$f(n)=O(g(n))$，当且仅当 $\exists c,n_0$，使得 $\forall n\geq n_0, 0\leq f(n)\leq c\cdot g(n)$。

研究时间复杂度时通常会使用 O 符号，因为我们关注的通常是程序用时的上界，而不关心其用时的下界。

需要注意的是，这里的「上界」和「下界」是对于函数的变化趋势而言的，而不是对算法而言的。算法用时的上界对应的是「最坏时间复杂度」而非大 O 记号。所以，使用 Θ 记号表示最坏时间复杂度是完全可行的，甚至可以说 Θ 比 O 更加精确，而使用 O 记号的主要原因，一是我们有时只能证明时间复杂度的上界而无法证明其下界（这种情况一般出现在较为复杂的算法以及复杂度分析），二是 O 在电脑上输入更方便一些。

大 Ω 符号

同样的，我们使用 Ω 符号来描述一个函数的渐进下界。$f(n)=\Omega(g(n))$，当且仅当 $\exists c,n_0$，使得 $\forall n\geq n_0, 0\leq c\cdot g(n)\leq f(n)$。

小 o 符号

如果说 O 符号相当于小于等于号，那么 o 符号就相当于小于号。

小 o 符号大量应用于数学分析中，函数在某点处的泰勒展开式拥有皮亚诺余项，使用小 o 符号表示严格小于，从而进行等价无穷小的渐进分析。

$f(n) = o(g(n))$，当且仅当对于任意给定的正数 c，$\exists n_0$，使得 $\forall n \geq n_0, 0 \leq f(n) < c \cdot g(n)$。

小 符号

如果说 Ω 符号相当于大于等于号，那么 ω 符号就相当于大于号。

$f(n) = \omega(g(n))$，当且仅当对于任意给定的正数 c，$\exists n_0$，使得 $\forall n \geq n_0, 0 \leq c \cdot g(n) < f(n)$。

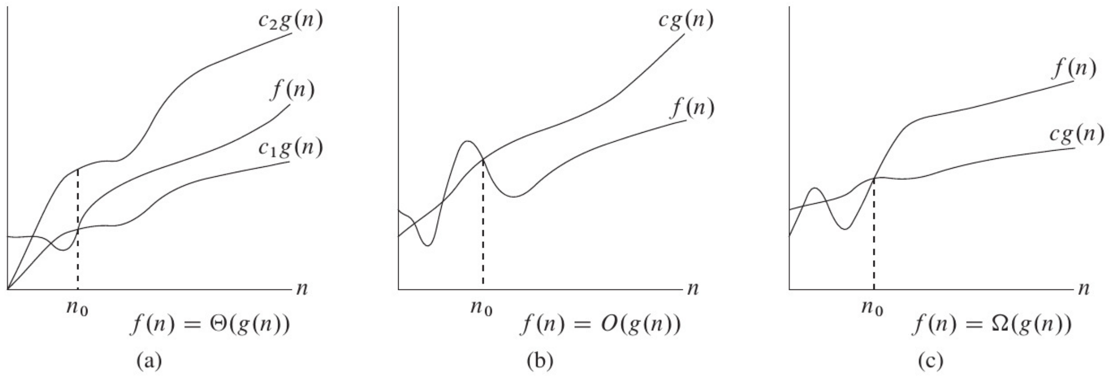


图 5.1

常见性质

- $f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \land f(n) = \Omega(g(n))$
- $f_1(n) + f_2(n) = O(\max(f_1(n), f_2(n)))$
- $f_1(n) \times f_2(n) = O(f_1(n) \times f_2(n))$
- $\forall a \neq 1, \log_a n = O(\log_2 n)$。由换底公式可以得知，任何对数函数无论底数为何，都具有相同的增长率，因此渐进时间复杂度中对数的底数一般省略不写。

简单的时间复杂度计算的例子

for 循环

```cpp
int n, m;
std::cin >> n >> m;
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < n; ++j) {
    for (int k = 0; k < m; ++k) {
      std::cout << "hello world\n";
    }
  }
}
```

```
n = int(input())
m = int(input())
for i in range(0, n):
    for j in range(0, n):
        for k in range(0, m):
            print("hello world")
```

如果以输入的数值 n 和 m 的大小作为数据规模，则上面这段代码的时间复杂度为 $\Theta(n^2m)$。

DFS

在对一张 n 个点 m 条边的图进行 DFS 时，由于每个节点和每条边都只会被访问常数次，复杂度为 $\Theta(n+m)$。

哪些量是常量？

当我们要进行若干次操作时，如何判断这若干次操作是否影响时间复杂度呢？例如：

```
const int N = 100000;
for (int i = 0; i < N; ++i) {
  std::cout << "hello world\n";
}
```

```
N = 100000
for i in range(0, N):
    print("hello world")
```

如果 N 的大小不被看作输入规模，那么这段代码的时间复杂度就是 $O(1)$。

进行时间复杂度计算时，哪些变量被视作输入规模是很重要的，而所有和输入规模无关的量都被视作常量，计算复杂度时可当作 1 来处理。

需要注意的是，在进行时间复杂度相关的理论性讨论时，「算法能够解决任何规模的问题」是一个基本假设（当然，在实际中，由于时间和存储空间有限，无法解决规模过大的问题）。因此，能在常量时间内解决数据规模有限的问题（例如，对于数据范围内的每个可能输入预先计算出答案）并不能使一个算法的时间复杂度变为 $O(1)$。

主定理 (Master Theorem)

我们可以使用 Master Theorem 来快速求得关于递归算法的复杂度。Master Theorem 递推关系式如下

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \qquad \forall n > b$$

那么

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & f(n) = O(n^{\log_b (a)-\epsilon}), \epsilon > 0 \\ \Theta(f(n)) & f(n) = \Omega(n^{\log_b (a)+\epsilon}), \epsilon \geq 0 \\ \Theta(n^{\log_b a} \log^{k+1} n) & f(n) = \Theta(n^{\log_b a} \log^k n), k \geq 0 \end{cases}$$

需要注意的是，这里的第二种情况还需要满足 regularity condition, 即 $af(n/b) \leq cf(n)$，for some constant $c < 1$ and sufficiently large n。

证明思路是是将规模为 n 的问题，分解为 a 个规模为 $(\frac{n}{b})$ 的问题，然后依次合并，直到合并到最高层。每一次合并子问题，都需要花费 $f(n)$ 的时间。

" 证明"

依据上文提到的证明思路，具体证明过程如下

对于第 0 层（最高层），合并子问题需要花费 $f(n)$ 的时间

对于第 1 层（第一次划分出来的子问题），共有 a 个子问题，每个子问题合并需要花费 $f\left(\frac{n}{b}\right)$ 的时间，所以合并总共要花费 $af\left(\frac{n}{b}\right)$ 的时间。

层层递推，我们可以写出类推树如下：

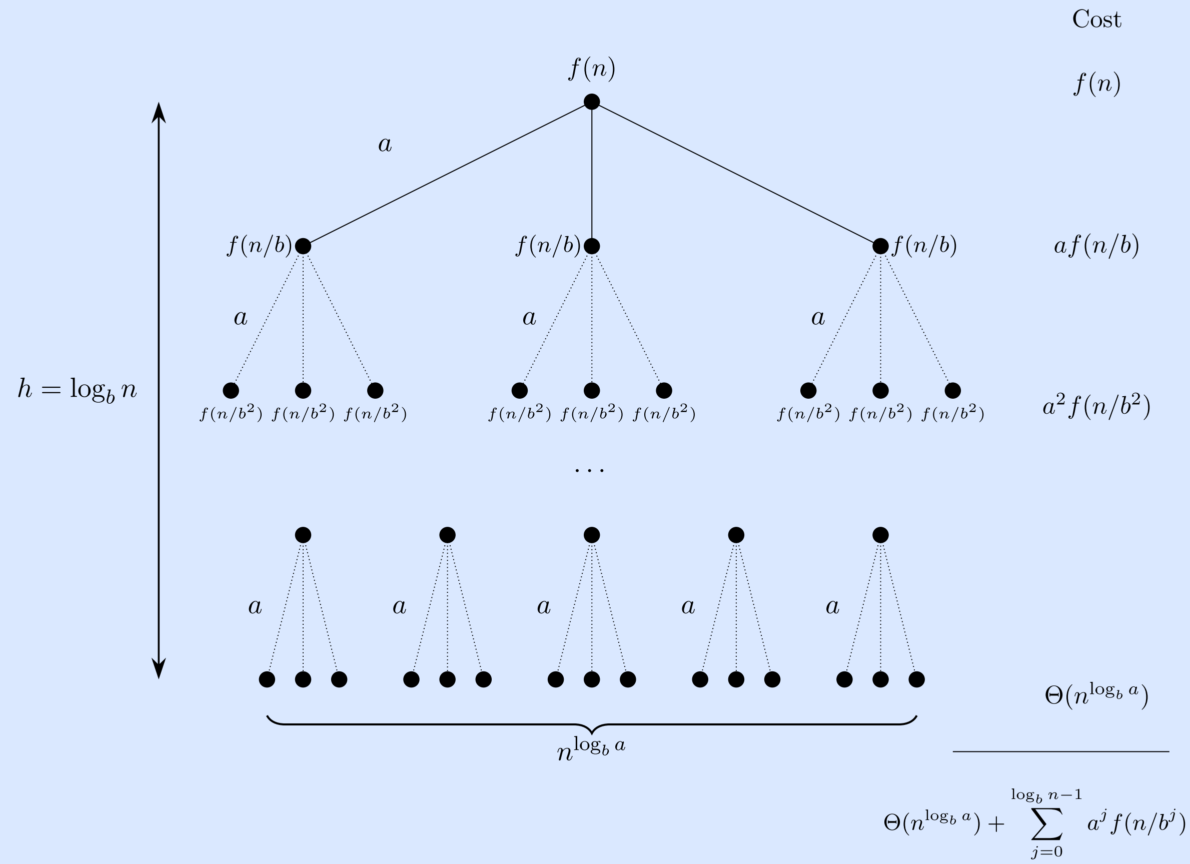


图 5.2

这棵树的高度为 $\log_b n$，共有 $n^{\log_b a}$ 个叶子，从而 $T(n)=\Theta(n^{\log_b a})+g(n)$，其中 $g(n)=\sum_{j=0}^{\log_b n-1} a^j f(n/b^j)$。

针对于第一种情况：$f(n)=O(n^{\log_b a-\epsilon})$，因此 $g(n)=O(n^{\log_b a})$。

对于第二种情况而言：首先 $g(n)=\Omega(f(n))$，又因为 $af(\dfrac{n}{b})\leq cf(n)$，只要 c 的取值是一个足够小的正数，且 n 的取值足够大，因此可以推导出：$g(n)=O(f(n))$。两侧夹逼可以得出，$g(n)=\Theta(f(n))$。

而对于第三种情况：$f(n)=\Theta(n^{\log_b a})$，因此 $g(n)=O(n^{\log_b a}\log n)$。$T(n)$ 的结果可在 $g(n)$ 得出后显然得到。

下面举几个例子来说明主定理如何使用。

1. $T(n)=2T\left(\frac{n}{2}\right)+1$，那么 $a=2,b=2,\log_2 2=1$，那么 ϵ 可以取值在 $(0,1]$ 之间，从而满足第一种情况，所以 $T(n)=\Theta(n)$。
2. $T(n)=T\left(\frac{n}{2}\right)+n$，那么 $a=1,b=2,\log_2 1=0$，那么 ϵ 可以取值在 $(0,1]$ 之间，从而满足第二种情况，所以 $T(n)=\Theta(n)$。
3. $T(n)=T\left(\frac{n}{2}\right)+\log n$，那么 $a=1,b=2,\log_2 1=0$，那么 k 可以取值为 1，从而满足第三种情况，所以 $T(n)=\Theta(\log^2 n)$。
4. $T(n)=T\left(\frac{n}{2}\right)+1$，那么 $a=1,b=2,\log_2 1=0$，那么 k 可以取值为 0，从而满足第三种情况，所以 $T(n)=\Theta(\log n)$。

均摊复杂度

算法往往是会对内存中的数据进行修改的，而同一个算法的多次执行，就会通过对数据的修改而互相影响。

例如快速排序中的「按大小分类」操作，单次执行的最坏时间复杂度，看似是 $O(n)$ 的，但是由于快排的分治过程，先前的「分类」操作每次都减小了数组长度，所以实际的总复杂度 $O(n\log n)$，分摊在每一次「分类」操作上，是 $O(\log n)$。

多次操作的总复杂度除以操作次数，就是这种操作的**均摊复杂度**。

势能分析

势能分析，是一种求均摊复杂度上界的方法。

求均摊复杂度，关键是表达出先前操作对当前操作的影响。势能分析用一个函数来表达此种影响。

定义「状态」S：即某一时刻的所有数据。在快排的例子中，一个「状态」就是当前过程需要排序的下标区间

定义「初始状态」S_0：即未进行任何操作时的状态。在快排的例子中，「初始状态」就是整个数组

假设存在从状态到数的函数 F，且对于任何状态 S，$F(S) \geq F(S_0)$，则有以下推论：

设 $S_1, S_2, \cdots, S_m$ 为从 S_0 开始连续做 m 次操作所得的状态序列，c_i 为第 i 次操作的时间开销。

记 $p_i = c_i + F(S_i) - F(S_{i-1})$，则 m 次操作的总时间花销为

$$\sum_{i=1}^{m} p_i + F(S_0) - F(S_m)$$

（正负相消，证明显然）

又因为 $F(S) \geq F(S_0)$，所以有

$$\sum_{i=1}^{m} p_i \geq \sum_{i=1}^{m} c_i$$

因此，若 $p_i = O(T(n))$，则 $O(T(n))$ 是均摊复杂度的一个上界。

势能分析在实际应用中有很多技巧，在此不详细展开。

空间复杂度

类似地，算法所使用的空间随输入规模变化的趋势可以用**空间复杂度**来衡量。

计算复杂性

本文主要从算法分析的角度对复杂度进行了介绍，如果有兴趣的话可以在 计算复杂性 进行更深入的了解。

5.3 枚举

> **Authors:** Early0v0, frank-xjh, Great-designer, ksyx, qiqistyle, Tiphereth-A , Saisyc, shuzhouliu, Xeonacid, xyf007

本页面将简要介绍枚举算法。

简介

枚举（英语：Enumerate）是基于已有知识来猜测答案的一种问题求解策略。

枚举的思想是不断地猜测，从可能的集合中一一尝试，然后再判断题目的条件是否成立。

要点

给出解空间

建立简洁的数学模型。

枚举的时候要想清楚：可能的情况是什么？要枚举哪些要素？

减少枚举的空间

枚举的范围是什么？是所有的内容都需要枚举吗？

在用枚举法解决问题的时候，一定要想清楚这两件事，否则会带来不必要的时间开销。

选择合适的枚举顺序

根据题目判断。比如例题中要求的是最大的符合条件的素数，那自然是从大到小枚举比较合适。

例题

以下是一个使用枚举解题与优化枚举范围的例子。

例题

一个数组中的数互不相同，求其中和为 0 的数对的个数。

" 解题思路"

枚举两个数的代码很容易就可以写出来。

```cpp
for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j)
    if (a[i] + a[j] == 0) ++ans;
```

```python
for i in range(n):
    for j in range(n):
        if a[i] + a[j] == 0:
            ans += 1
```

来看看枚举的范围如何优化。由于题中没要求数对是有序的，答案就是有序的情况的两倍（考虑如果 `(a, b)` 是答案，那么 `(b, a)` 也是答案）。对于这种情况，只需统计人为要求有顺序之后的答案，最后再乘上 2 就好了。

不妨要求第一个数要出现在靠前的位置。代码如下：

```cpp
for (int i = 0; i < n; ++i)
  for (int j = 0; j < i; ++j)
    if (a[i] + a[j] == 0) ++ans;
```

```python
for i in range(n):
    for j in range(i):
        if a[i] + a[j] == 0:
            ans += 1
```

不难发现这里已经减少了 j 的枚举范围，减少了这段代码的时间开销。

我们可以在此之上进一步优化。

两个数是否都一定要枚举出来呢？枚举其中一个数之后，题目的条件已经确定了其他的要素（另一个数）的条件，如果能找到一种方法直接判断题目要求的那个数是否存在，就可以省掉枚举后一个数的时间了。较为进阶地，在数据范围允许的情况下，我们可以使用桶[1-1] 记录遍历过的数。

```cpp
bool met[MAXN * 2 + 1];
memset(met, 0, sizeof(met));
for (int i = 0; i < n; ++i) {
  if (met[MAXN - a[i]]) ++ans;
  met[MAXN + a[i]] = true;
}
```

```python
met = [False] * (MAXN * 2 + 1)
for i in range(n):
    if met[MAXN - a[i]]:
        ans += 1
    met[a[i] + MAXN] = True
```

复杂度分析

- 时间复杂度分析：对 a 数组遍历了一遍就能完成题目要求，当 n 足够大的时候时间复杂度为 $O(n)$。
- 空间复杂度分析：$O(n + \max\{|x| : x \in a\})$。

习题

- 2811：熄灯问题 - OpenJudge[2]

脚注

[1] 桶排序 以及 主元素问题 以及 Stack Overflow 上对桶数据结构的讲解（英文）[1-1] [1-2]

[2] 2811: 熄灯问题 - OpenJudge

5.4 模拟

本页面将简要介绍模拟算法。

简介

模拟就是用计算机来模拟题目中要求的操作。

模拟题目通常具有码量大、操作多、思路繁复的特点。由于它码量大，经常会出现难以查错的情况，如果在考试中写错是相当浪费时间的。

技巧

写模拟题时，遵循以下的建议有可能会提升做题速度：

- 在动手写代码之前，在草纸上尽可能地写好要实现的流程。
- 在代码中，尽量把每个部分模块化，写成函数、结构体或类。

- 对于一些可能重复用到的概念，可以统一转化，方便处理：如，某题给你"YY-MM-DD 时：分" 把它抽取到一个函数，处理成秒，会减少概念混淆。
- 调试时分块调试。模块化的好处就是可以方便的单独调某一部分。
- 写代码的时候一定要思路清晰，不要想到什么写什么，要按照落在纸上的步骤写。

实际上，上述步骤在解决其它类型的题目时也是很有帮助的。

例题详解

"Climbing Worm[1]"

一只长度不计的蠕虫位于 n 英寸深的井的底部。它每次向上爬 u 英寸，但是必须休息一次才能再次向上爬。在休息的时候，它滑落了 d 英寸。之后它将重复向上爬和休息的过程。蠕虫爬出井口需要至少爬多少次？如果蠕虫爬完后刚好到达井的顶部，我们也设作蠕虫已经爬出井口。

" 解题思路"

直接使用程序模拟蠕虫爬井的过程就可以了。用一个循环重复蠕虫的爬井过程，当攀爬的长度超过或者等于井的深度时跳出。

" 参考代码"

```cpp
#include <cstdio>

int main(void) {
  int n = 0, u = 0, d = 0;
  std::scanf("%d%d%d", &u, &d, &n);
  int time = 0, dist = 0;
  while (true) {  // 用死循环来枚举
    dist += u;
    time++;
    if (dist >= n) break;  // 满足条件则退出死循环
    dist -= d;
  }
  printf("%d\n", time);  // 输出得到的结果
  return 0;
}
```

```python
u, d, n = map(int, input().split())
time = dist = 0
while True:  # 用死循环来枚举
    dist += u
    time += 1
    if dist >= n:  # 满足条件则退出死循环
        break
    dist -= d
print(time)  # 输出得到的结果
```

```
=== "Java"
    ```java
    import java.util.Scanner;
```
```

````java
public class Main {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        int n = input.nextInt();
        int u = input.nextInt();
        int d = input.nextInt();
        int time = 0, dist = 0;
        while (true) {  // 用死循环来枚举
            dist += u;
            time++;
            if (dist >= n) {
                break;  // 满足条件则退出死循环
            }
            dist -= d;
        }
        System.out.println(time);   // 输出得到的结果
        input.close();
    }
}

```
````

习题

- 「NOIP2014」 生活大爆炸版石头剪刀布 - Universal Online Judge[2]
- 「OpenJudge 3750」 魔兽世界[3]
- 「SDOI2010」 猪国杀 - LibreOJ[4]

参考资料与注释

[1] Climbing Worm

[2] 「NOIP2014」生活大爆炸版石头剪刀布 - Universal Online Judge

[3] 「OpenJudge 3750」魔兽世界

[4] 「SDOI2010」猪国杀 - LibreOJ

5.5　递归 & 分治

Authors: fudonglai, AngelKitty, labuladong

本页面将介绍递归与分治算法的区别与结合运用。

递归

定义

递归（英语：Recursion），在数学和计算机科学中是指在函数的定义中使用函数自身的方法，在计算机科学中还额外指一种通过重复将问题分解为同类的子问题而解决问题的方法。
````

引入

> 要理解递归，就得先理解什么是递归。

递归的基本思想是某个函数直接或者间接地调用自身，这样原问题的求解就转换为了许多性质相同但是规模更小的子问题。求解时只需要关注如何把原问题划分成符合条件的子问题，而不需要过分关注这个子问题是如何被解决的。

以下是一些有助于理解递归的例子：

1. 什么是递归？
2. 如何给一堆数字排序？答：分成两半，先排左半边再排右半边，最后合并就行了，至于怎么排左边和右边，请重新阅读这句话。
3. 你今年几岁？答：去年的岁数加一岁，1999 年我出生。

图 5.3 一个用于理解递归的例子

4.

递归在数学中非常常见。例如，集合论对自然数的正式定义是：1 是一个自然数，每个自然数都有一个后继，这一个后继也是自然数。

递归代码最重要的两个特征：结束条件和自我调用。自我调用是在解决子问题，而结束条件定义了最简子问题的答案。

```
int func(传入数值) {
  if (终止条件) return 最小子问题解;
  return func(缩小规模);
}
```

为什么要写递归

1. 结构清晰，可读性强。例如，分别用不同的方法实现 归并排序：

```
// 不使用递归的归并排序算法
template <typename T>
void merge_sort(vector<T> a) {
  int n = a.size();
  for (int seg = 1; seg < n; seg = seg + seg)
    for (int start = 0; start < n - seg; start += seg + seg)
      merge(a, start, start + seg - 1, std::min(start + seg + seg - 1, n - 1));
}

// 使用递归的归并排序算法
template <typename T>
void merge_sort(vector<T> a, int front, int end) {
  if (front >= end) return;
```

```
  int mid = front + (end - front) / 2;
  merge_sort(a, front, mid);
  merge_sort(a, mid + 1, end);
  merge(a, front, mid, end);
}
```

```
# 不使用递归的归并排序算法
def merge_sort(a):
  n = len(a)
  seg, start = 1, 0
  while seg < n:
      while start < n - seg:
          merge(a, start, start + seg - 1, min(start + seg + seg - 1, n - 1))
          start = start + seg + seg
      seg = seg + seg

# 使用递归的归并排序算法
def merge_sort(a, front, end):
  if front >= end:
      return
  mid = front + (end - front) / 2
  merge_sort(a, front, mid)
  merge_sort(a, mid + 1, end)
  merge(a, front, mid, end)
```

显然，递归版本比非递归版本更易理解。递归版本的做法一目了然：把左半边排序，把右半边排序，最后合并两边。而非递归版本看起来不知所云，充斥着各种难以理解的边界计算细节，特别容易出 bug，且难以调试。

2. 练习分析问题的结构。当发现问题可以被分解成相同结构的小问题时，递归写多了就能敏锐发现这个特点，进而高效解决问题。

递归的缺点

在程序执行中，递归是利用堆栈来实现的。每当进入一个函数调用，栈就会增加一层栈帧，每次函数返回，栈就会减少一层栈帧。而栈不是无限大的，当递归层数过多时，就会造成**栈溢出**的后果。

显然有时候递归处理是高效的，比如归并排序；**有时候是低效的**，比如数孙悟空身上的毛，因为堆栈会消耗额外空间，而简单的递推不会消耗空间。比如这个例子，给一个链表头，计算它的长度：

```
// 典型的递推遍历框架
int size(Node *head) {
  int size = 0;
  for (Node *p = head; p != nullptr; p = p->next) size++;
  return size;
}

// 我就是要写递归，递归天下第一
int size_recursion(Node *head) {
  if (head == nullptr) return 0;
  return size_recursion(head->next) + 1;
}
```

递归的优化

主页面：搜索优化 和 记忆化搜索

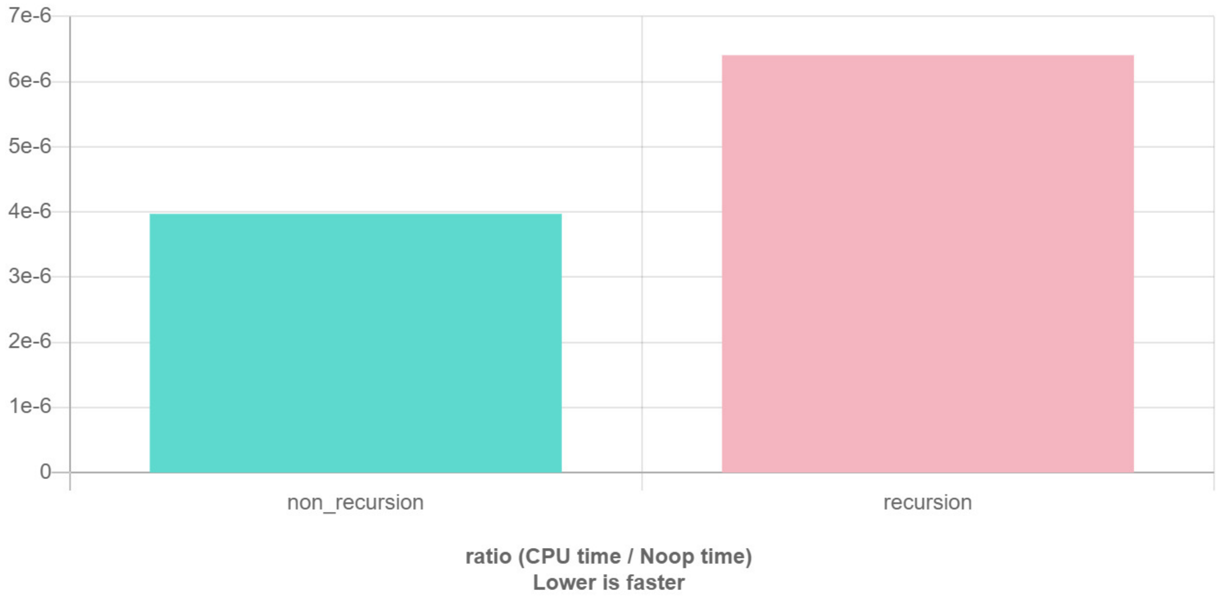


图 5.4 [二者的对比，compiler 设为 Clang 10.0，优化设为 O1](https://quick-bench.com/q/rZ7jWPmSdltparOO5ndLgmS9BVc)

比较初级的递归实现可能递归次数太多，容易超时。这时需要对递归进行优化。[1]

分治

定义

分治（英语：Divide and Conquer），字面上的解释是「分而治之」，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。

过程

分治算法的核心思想就是「分而治之」。

大概的流程可以分为三步：分解 -> 解决 -> 合并。

1. 分解原问题为结构相同的子问题。
2. 分解到某个容易求解的边界之后，进行递归求解。
3. 将子问题的解合并成原问题的解。

分治法能解决的问题一般有如下特征：

- 该问题的规模缩小到一定的程度就可以容易地解决。
- 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质，利用该问题分解出的子问题的解可以合并为该问题的解。
- 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题。

> **" 注意"**
>
> 如果各子问题是不独立的，则分治法要重复地解公共的子问题，也就做了许多不必要的工作。此时虽然也可用分治法，但一般用 动态规划 较好。

以归并排序为例。假设实现归并排序的函数名为 `merge_sort`。明确该函数的职责，即**对传入的一个数组排序**。这个问题显然可以分解。给一个数组排序等于给该数组的左右两半分别排序，然后合并成一个数组。

```
void merge_sort(一个数组) {
  if (可以很容易处理) return;
  merge_sort(左半个数组);
  merge_sort(右半个数组);
  merge(左半个数组, 右半个数组);
}
```

传给它半个数组，那么处理完后这半个数组就已经被排好了。注意到，`merge_sort` 与二叉树的后序遍历模板极其相似。因为分治算法的套路是**分解 -> 解决（触底）-> 合并（回溯）**，先左右分解，再处理合并，回溯就是在退栈，即相当于后序遍历。

`merge` 函数的实现方式与两个有序链表的合并一致。

要点

写递归的要点

明白一个函数的作用并相信它能完成这个任务，千万不要跳进这个函数里面企图探究更多细节，否则就会陷入无穷的细节无法自拔，人脑能压几个栈啊。

以遍历二叉树为例。

```
void traverse(TreeNode* root) {
  if (root == nullptr) return;
  traverse(root->left);
  traverse(root->right);
}
```

这几行代码就足以遍历任何一棵二叉树了。对于递归函数 `traverse(root)`，只要相信给它一个根节点 `root`，它就能遍历这棵树。所以只需要把这个节点的左右节点再传给这个函数就行了。

同样扩展到遍历一棵 N 叉树。与二叉树的写法一模一样。不过，对于 N 叉树，显然没有中序遍历。

```
void traverse(TreeNode* root) {
  if (root == nullptr) return;
  for (auto child : root->children) traverse(child);
}
```

区别

递归与枚举的区别

递归和枚举的区别在于：枚举是横向地把问题划分，然后依次求解子问题；而递归是把问题逐级分解，是纵向的拆分。

递归与分治的区别

递归是一种编程技巧，一种解决问题的思维方式；分治算法很大程度上是基于递归的，解决更具体问题的算法思想。

例题详解

"437. 路径总和 III[2]"

给定一个二叉树，它的每个结点都存放着一个整数值。

找出路径和等于给定数值的路径总数。

路径不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只能从父节点到子节点）。

二叉树不超过 1000 个节点，且节点数值范围是 [-1000000,1000000] 的整数。

示例：

```
root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8

      10
     /  \
    5   -3
   / \    \
  3   2   11
 / \   \
3  -2   1

返回 3。和等于 8 的路径有：

1.  5 -> 3
2.  5 -> 2 -> 1
3. -3 -> 11

/**
 * 二叉树结点的定义
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
```

" 参考代码"

```
int pathSum(TreeNode *root, int sum) {
  if (root == nullptr) return 0;
  return count(root, sum) + pathSum(root->left, sum) +
         pathSum(root->right, sum);
}

int count(TreeNode *node, int sum) {
  if (node == nullptr) return 0;
  return (node->val == sum) + count(node->left, sum - node->val) +
         count(node->right, sum - node->val);
}
```

" 题目解析"

题目看起来很复杂，不过代码却极其简洁。

首先明确，递归求解树的问题必然是要遍历整棵树的，所以二叉树的遍历框架（分别对左右子树递归调用函数本身）必然要出现在主函数 pathSum 中。那么对于每个节点，它们应该干什么呢？它们应该看看，自己和它们的子树包含多少条符合条件的路径。好了，这道题就结束了。

按照前面说的技巧，根据刚才的分析来定义清楚每个递归函数应该做的事：

PathSum 函数：给定一个节点和一个目标值，返回以这个节点为根的树中，和为目标值的路径总数。

count 函数：给定一个节点和一个目标值，返回以这个节点为根的树中，能凑出几个以该节点为路径开头，和为目标值的路径总数。

" 参考代码（附注释）"

```
int pathSum(TreeNode *root, int sum) {
  if (root == nullptr) return 0;
  int pathImLeading = count(root, sum);  // 自己为开头的路径数
  int leftPathSum = pathSum(root->left, sum);  // 左边路径总数（相信它能算出来）
  int rightPathSum =
      pathSum(root->right, sum);  // 右边路径总数（相信它能算出来）
  return leftPathSum + rightPathSum + pathImLeading;
}

int count(TreeNode *node, int sum) {
  if (node == nullptr) return 0;
  // 能不能作为一条单独的路径呢？
  int isMe = (node->val == sum) ? 1 : 0;
  // 左边的，你那边能凑几个 sum - node.val ？
  int leftNode = count(node->left, sum - node->val);
  // 右边的，你那边能凑几个 sum - node.val ？
  int rightNode = count(node->right, sum - node->val);
  return isMe + leftNode + rightNode;  // 我这能凑这么多个
}
```

还是那句话，**明白每个函数能做的事，并相信它们能够完成。**

总结下，PathSum 函数提供了二叉树遍历框架，在遍历中对每个节点调用 count 函数（这里用的是先序遍历，不过中序遍历和后序遍历也可以）。count 函数也是一个二叉树遍历，用于寻找以该节点开头的目标值路径。

习题

- LeetCode 上的递归专题练习[3]
- LeetCode 上的分治算法专项练习[4]

参考资料与注释

[1] labuladong 的算法小抄 - 递归详解

[2] 437. 路径总和 III

[3] LeetCode 上的递归专题练习

[4] LeetCode 上的分治算法专项练习

5.6 贪心

本页面将简要介绍贪心算法。

引入

贪心算法（英语：greedy algorithm），是用计算机来模拟一个「贪心」的人做出决策的过程。这个人十分贪婪，每一步行动总是按某种指标选取最优的操作。而且他目光短浅，总是只看眼前，并不考虑以后可能造成的影响。

可想而知，并不是所有的时候贪心法都能获得最优解，所以一般使用贪心法的时候，都要确保自己能证明其正确性。

解释

适用范围

贪心算法在有最优子结构的问题中尤为有效。最优子结构的意思是问题能够分解成子问题来解决，子问题的最优解能递推到最终问题的最优解。[1]

证明

贪心算法有两种证明方法：反证法和归纳法。一般情况下，一道题只会用到其中的一种方法来证明。

1. 反证法：如果交换方案中任意两个元素/相邻的两个元素后，答案不会变得更好，那么可以推定目前的解已经是最优解了。
2. 归纳法：先算得出边界情况（例如 $n = 1$）的最优解 F_1，然后再证明：对于每个 n，F_{n+1} 都可以由 F_n 推导出结果。

要点

常见题型

在提高组难度以下的题目中，最常见的贪心有两种。

- 「我们将 XXX 按照某某顺序排序，然后按某种顺序（例如从小到大）选择。」。
- 「我们每次都取 XXX 中最大/小的东西，并更新 XXX。」（有时「XXX 中最大/小的东西」可以优化，比如用优先队列维护）

二者的区别在于一种是离线的，先处理后选择；一种是在线的，边处理边选择。

排序解法

用排序法常见的情况是输入一个包含几个（一般一到两个）权值的数组，通过排序然后遍历模拟计算的方法求出最优值。

后悔解法

思路是无论当前的选项是否最优都接受，然后进行比较，如果选择之后不是最优了，则反悔，舍弃掉这个选项；否则，正式接受。如此往复。

区别

与动态规划的区别

贪心算法与动态规划的不同在于它对每个子问题的解决方案都做出选择，不能回退。动态规划则会保存以前的运算结果，并根据以前的结果对当前进行选择，有回退功能。

例题详解

邻项交换法的例题

"NOIP 2012 **国王游戏**[2]"

恰逢 H 国国庆，国王邀请 n 位大臣来玩一个有奖游戏。首先，他让每个大臣在左、右手上面分别写下一个整数，国王自己也在左、右手上各写一个整数。然后，让这 n 位大臣排成一排，国王站在队伍的最前面。排好队后，所有的大臣都会获得国王奖赏的若干金币，每位大臣获得的金币数分别是：排在该大臣前面的所有人的左手上的数的乘积除以他自己右手上的数，然后向下取整得到的结果。

国王不希望某一个大臣获得特别多的奖赏，所以他想请你帮他重新安排一下队伍的顺序，使得获得奖赏最多的大臣，所获奖赏尽可能的少。注意，国王的位置始终在队伍的最前面。

" **解题思路**"

设排序后第 i 个大臣左右手上的数分别为 a_i, b_i。考虑通过邻项交换法推导贪心策略。

用 s 表示第 i 个大臣前面所有人的 a_i 的乘积，那么第 i 个大臣得到的奖赏就是 $\dfrac{s}{b_i}$，第 $i+1$ 个大臣得到的奖赏就是 $\dfrac{s \cdot a_i}{b_{i+1}}$。

如果我们交换第 i 个大臣与第 $i+1$ 个大臣，那么此时的第 i 个大臣得到的奖赏就是 $\dfrac{s}{b_{i+1}}$，第 $i+1$ 个大臣得到的奖赏就是 $\dfrac{s \cdot a_{i+1}}{b_i}$。

如果交换前更优当且仅当

$$\max\left(\frac{s}{b_i}, \frac{s \cdot a_i}{b_{i+1}}\right) < \max\left(\frac{s}{b_{i+1}}, \frac{s \cdot a_{i+1}}{b_i}\right)$$

提取出相同的 s 并约分得到

$$\max\left(\frac{1}{b_i}, \frac{a_i}{b_{i+1}}\right) < \max\left(\frac{1}{b_{i+1}}, \frac{a_{i+1}}{b_i}\right)$$

然后分式化成整式得到

$$\max(b_{i+1}, a_i \cdot b_i) < \max(b_i, a_{i+1} \cdot b_{i+1})$$

实现的时候我们将输入的两个数用一个结构体来保存并重载运算符：

```
struct uv {
  int a, b;

  bool operator<(const uv &x) const {
    return max(x.b, a * b) < max(b, x.a * x.b);
  }
};
```

后悔法的例题

"「USACO09OPEN」 工作调度 Work Scheduling[3]"

约翰的工作日从 0 时刻开始，有 10^9 个单位时间。在任一单位时间，他都可以选择编号 1 到 N 的 $N(1 \le N \le 10^5)$ 项工作中的任意一项工作来完成。工作 i 的截止时间是 $D_i(1 \le D_i \le 10^9)$，完成后获利是 $P_i(1 \le P_i \le 10^9)$。在给定的工作利润和截止时间下，求约翰能够获得的利润最大为多少。

" 解题思路"

1. 先假设每一项工作都做，将各项工作按截止时间排序后入队；
2. 在判断第 `i` 项工作做与不做时，若其截至时间符合条件，则将其与队中报酬最小的元素比较，若第 `i` 项工作报酬较高（后悔），则 `ans += a[i].p - q.top()`。

   用优先队列（小根堆）来维护队首元素最小。
3. 当 `a[i].d<=q.size()` 可以这么理解从 0 开始到 `a[i].d` 这个时间段只能做 `a[i].d` 个任务，而若 `q.size()>=a[i].d` 说明完成 `q.size()` 个任务时间大于等于 `a[i].d` 的时间，所以当第 `i` 个任务获利比较大的时候应该把最小的任务从优先级队列中换出。

" 参考代码"

```cpp
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <queue>
using namespace std;

struct f {
  long long d;
  long long p;
} a[100005];

bool cmp(f A, f B) { return A.d < B.d; }

priority_queue<long long, vector<long long>, greater<long long> >
    q;  // 小根堆维护最小值

int main() {
  long long n, i;
  cin >> n;
  for (i = 1; i <= n; i++) {
    scanf("%lld%lld", &a[i].d, &a[i].p);
  }
  sort(a + 1, a + n + 1, cmp);
  long long ans = 0;
  for (i = 1; i <= n; i++) {
    if (a[i].d <= (int)q.size()) {  // 超过截止时间
      if (q.top() < a[i].p) {       // 后悔
        ans += a[i].p - q.top();
        q.pop();
        q.push(a[i].p);
      }
    } else {  // 直接加入队列
```

```
      ans += a[i].p;
      q.push(a[i].p);
    }
  }
  cout << ans << endl;
  return 0;
}
```

```python
from collections import defaultdict
from heapq import heappush, heappop

a = defaultdict(list)
for _ in range(int(input())):
    d, p = map(int, input().split())
    a[d].append(p)  # 存放对应时间的收益

ans = 0  # 记录总收益
q = []  # 小根堆维护最小值
l = sorted(a.keys(), reverse=True)
for i, j in zip(l, l[1:] + [0]):
    for k in a.pop(i):
        heappush(q, ~k)
    for _ in range(i - j):
        if q:  # 从堆中取出收益最多的工作
            ans += ~heappop(q)
        else:  # 堆为空时退出循环
            break
print(ans)
```

复杂度分析

- 空间复杂度：当输入 n 个任务时使用 n 个 a 数组元素，优先队列中最差情况下会储存 n 个元素，则空间复杂度为 $O(n)$。
- 时间复杂度：`std::sort` 的时间复杂度为 $O(n\log n)$，维护优先队列的时间复杂度为 $O(n\log n)$，综上所述，时间复杂度为 $O(n\log n)$。

习题

- P1209[USACO1.3] 修理牛棚 Barn Repair - 洛谷[4]
- P2123 皇后游戏 - 洛谷[5]
- LeetCode 上标签为贪心算法的题目[6]

参考资料与注释

[1] 贪心算法 - 维基百科， 自由的百科全书

[2] NOIP 2012 国王游戏

[3] 「USACO09OPEN」工作调度 Work Scheduling

[4] P1209[USACO1.3] 修理牛棚 Barn Repair - 洛谷

[5] P2123 皇后游戏 - 洛谷

[6] LeetCode 上标签为贪心算法的题目

5.7 排序

5.7.1 排序简介

本页面将简要介绍排序算法。

定义

排序算法（英语：Sorting algorithm）是一种将一组特定的数据按某种顺序进行排列的算法。排序算法多种多样，性质也大多不同。

性质

稳定性

稳定性是指相等的元素经过排序之后相对顺序是否发生了改变。

拥有稳定性这一特性的算法会让原本有相等键值的纪录维持相对次序，即如果一个排序算法是稳定的，当有两个相等键值的纪录 R 和 S，且在原本的列表中 R 出现在 S 之前，在排序过的列表中 R 也将会是在 S 之前。

基数排序、计数排序、插入排序、冒泡排序、归并排序是稳定排序。

选择排序、堆排序、快速排序、希尔排序不是稳定排序。

时间复杂度

主页面：复杂度

时间复杂度用来衡量一个算法的运行时间和输入规模的关系，通常用 O 表示。

简单计算复杂度的方法一般是统计「简单操作」的执行次数，有时候也可以直接数循环的层数来近似估计。

时间复杂度分为最优时间复杂度、平均时间复杂度和最坏时间复杂度。OI 竞赛中要考虑的一般是最坏时间复杂度，因为它代表的是算法运行水平的下界，在评测中不会出现更差的结果了。

基于比较的排序算法的时间复杂度下限是 $O(n\log n)$ 的。

当然也有不是 $O(n\log n)$ 的。例如，计数排序 的时间复杂度是 $O(n+w)$，其中 w 代表输入数据的值域大小。

以下是几种排序算法的比较。

空间复杂度

与时间复杂度类似，空间复杂度用来描述算法空间消耗的规模。一般来说，空间复杂度越小，算法越好。

外部链接

- 排序算法 - 维基百科， 自由的百科全书[1]

图 5.5 几种排序算法的比较

参考资料与注释

[1] 排序算法 - 维基百科，自由的百科全书

5.7.2 选择排序

本页面将简要介绍选择排序。

定义

选择排序（英语：Selection sort）是一种简单直观的排序算法。它的工作原理是每次找出第 i 小的元素（也就是 $A_{i..n}$ 中最小的元素），然后将这个元素与数组第 i 个位置上的元素交换。

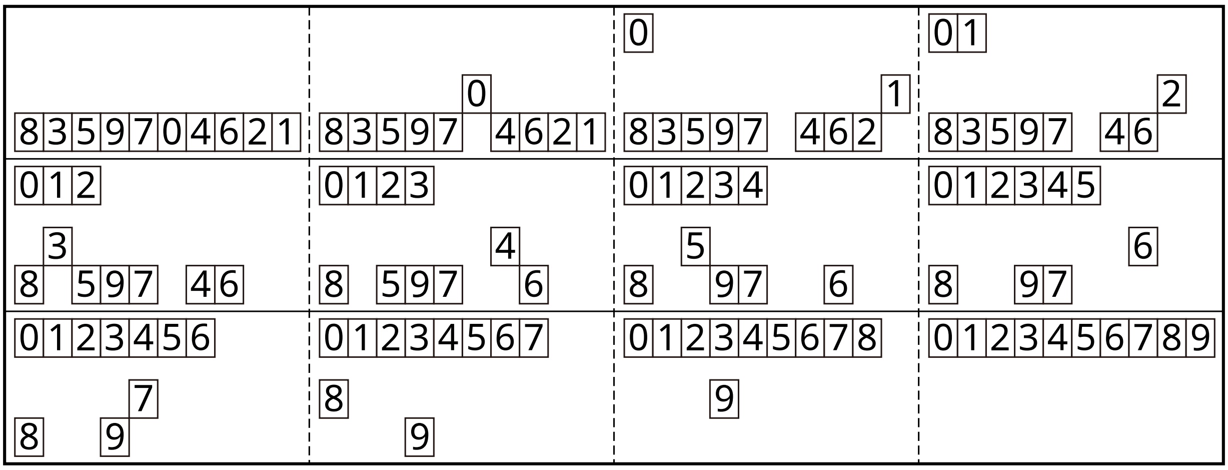


图 5.6 selection sort animate example

性质

稳定性

由于 swap（交换两个元素）操作的存在，选择排序是一种不稳定的排序算法。

时间复杂度

选择排序的最优时间复杂度、平均时间复杂度和最坏时间复杂度均为 $O(n^2)$。

代码实现

伪代码

Input. An array A consisting of n elements.
Output. A will be sorted in nondecreasing order.
Method.
for $i \leftarrow 1$ **to** $n - 1$
  $ith \leftarrow i$
  for $j \leftarrow i + 1$ **to** n
    if $A[j] < A[ith]$
      $ith \leftarrow j$
  swap $A[i]$ and $A[ith]$

```cpp
void selection_sort(int* a, int n) {
  for (int i = 1; i < n; ++i) {
    int ith = i;
    for (int j = i + 1; j <= n; ++j) {
      if (a[j] < a[ith]) {
        ith = j;
      }
    }
    std::swap(a[i], a[ith]);
  }
}
```

```python
def selection_sort(a, n):
    for i in range(1, n):
        ith = i
        for j in range(i + 1, n + 1):
            if a[j] < a[ith]:
                ith = j
        a[i], a[ith] = a[ith], a[i]
```

```java
// arr 代码下标从 1 开始索引
static void selection_sort(int[] arr, int n) {
    for (int i = 1; i < n; i++) {
        int ith = i;
        for (int j = i + 1; j <= n; j++) {
            if (arr[j] < arr[ith]) {
                ith = j;
            }
        }
        // swap
```

```
        int temp = arr[i];
        arr[i] = arr[ith];
        arr[ith] = temp;
    }
}
```

5.7.3 冒泡排序

本页面将简要介绍冒泡排序。

定义

冒泡排序（英语：Bubble sort）是一种简单的排序算法。由于在算法的执行过程中，较小的元素像是气泡般慢慢「浮」到数列的顶端，故叫做冒泡排序。

过程

它的工作原理是每次检查相邻两个元素，如果前面的元素与后面的元素满足给定的排序条件，就将相邻两个元素交换。当没有相邻的元素需要交换时，排序就完成了。

经过 i 次扫描后，数列的末尾 i 项必然是最大的 i 项，因此冒泡排序最多需要扫描 $n-1$ 遍数组就能完成排序。

性质

稳定性

冒泡排序是一种稳定的排序算法。

时间复杂度

在序列完全有序时，冒泡排序只需遍历一遍数组，不用执行任何交换操作，时间复杂度为 $O(n)$。

在最坏情况下，冒泡排序要执行 $\frac{(n-1)n}{2}$ 次交换操作，时间复杂度为 $O(n^2)$。

冒泡排序的平均时间复杂度为 $O(n^2)$。

代码实现

伪代码

Input. An array A consisting of n elements.
Output. A will be sorted in nondecreasing order stably.
Method.
$flag \leftarrow True$
while $flag$
    $flag \leftarrow False$
    for $i \leftarrow 1$ **to** $n-1$
        if $A[i] > A[i+1]$
            $flag \leftarrow True$
            Swap $A[i]$ and $A[i+1]$

```
// 假设数组的大小是 n + 1，冒泡排序从数组下标 1 开始
void bubble_sort(int *a, int n) {
  bool flag = true;
  while (flag) {
```

```
    flag = false;
    for (int i = 1; i < n; ++i) {
      if (a[i] > a[i + 1]) {
        flag = true;
        int t = a[i];
        a[i] = a[i + 1];
        a[i + 1] = t;
      }
    }
  }
}
```

```python
def bubble_sort(a, n):
    flag = True
    while flag:
        flag = False
        for i in range(1, n):
            if a[i] > a[i + 1]:
                flag = True
                a[i], a[i + 1] = a[i + 1], a[i]
```

```java
// 假设数组的大小是 n + 1，冒泡排序从数组下标 1 开始
static void bubble_sort(int[] a, int n) {
    boolean flag = true;
    while (flag) {
        flag = false;
        for (int i = 1; i < n; i++) {
            if (a[i] > a[i + 1]) {
                flag = true;
                int t = a[i];
                a[i] = a[i + 1];
                a[i + 1] = t;
            }
        }
    }
}
```

5.7.4 插入排序

本页面将简要介绍插入排序。

定义

插入排序（英语：Insertion sort）是一种简单直观的排序算法。它的工作原理为将待排列元素划分为「已排序」和「未排序」两部分，每次从「未排序的」元素中选择一个插入到「已排序的」元素中的正确位置。

一个与插入排序相同的操作是打扑克牌时，从牌桌上抓一张牌，按牌面大小插到手牌后，再抓下一张牌。

性质

稳定性

插入排序是一种稳定的排序算法。

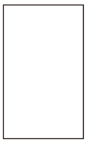


图 5.7 insertion sort animate example

时间复杂度

插入排序的最优时间复杂度为 $O(n)$，在数列几乎有序时效率很高。

插入排序的最坏时间复杂度和平均时间复杂度都为 $O(n^2)$。

代码实现

伪代码

Input. An array A consisting of n elements.
Output. A will be sorted in nondecreasing order stably.
Method.
for $i \leftarrow 2$ **to** n
    $key \leftarrow A[i]$
    $j \leftarrow i - 1$
    while $j > 0$ **and** $A[j] > key$
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j - 1$
    $A[j+1] \leftarrow key$

```
void insertion_sort(int arr[], int len) {
  for (int i = 1; i < len; ++i) {
    int key = arr[i];
    int j = i - 1;
    while (j >= 0 && arr[j] > key) {
      arr[j + 1] = arr[j];
      j--;
    }
    arr[j + 1] = key;
  }
}
```

```python
def insertion_sort(arr, n):
    for i in range(1, n):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j = j - 1
        arr[j + 1] = key
```

折半插入排序

插入排序还可以通过二分算法优化性能，在排序元素数量较多时优化的效果比较明显。

时间复杂度

折半插入排序与直接插入排序的基本思想是一致的，折半插入排序仅对插入排序时间复杂度中的常数进行了优化，所以优化后的时间复杂度仍然不变。

代码实现

```cpp
void insertion_sort(int arr[], int len) {
  if (len < 2) return;
  for (int i = 1; i != len; ++i) {
    int key = arr[i];
    auto index = upper_bound(arr, arr + i, key) - arr;
    // 使用 memmove 移动元素，比使用 for 循环速度更快，时间复杂度仍为 O(n)
    memmove(arr + index + 1, arr + index, (i - index) * sizeof(int));
    arr[index] = key;
  }
}
```

5.7.5 计数排序

" 提醒"

本页面要介绍的不是 基数排序。

本页面将简要介绍计数排序。

定义

计数排序（英语：Counting sort）是一种线性时间的排序算法。

过程

计数排序的工作原理是使用一个额外的数组 C，其中第 i 个元素是待排序数组 A 中值等于 i 的元素的个数，然后根据数组 C 来将 A 中的元素排到正确的位置。[1]

它的工作过程分为三个步骤：

1. 计算每个数出现了几次；
2. 求出每个数出现次数的 前缀和；
3. 利用出现次数的前缀和，从右至左计算每个数的排名。

计算前缀和的原因

阅读本章内容只需要了解前缀和概念即可

直接将 C 中正数对应的元素依次放入 A 中不能解决元素重复的情形。

我们通过为额外数组 C 中的每一项计算前缀和，结合每一项的数值，就可以为重复元素确定一个唯一排名：

额外数组 C 中每一项的数值即是该 key 值下重复元素的个数，而该项的前缀和即是排在最后一个的重复元素的排名。

如果按照 A 的逆序进行排列，那么显然排序后的数组将保持 A 的原序（相同 key 值情况下），也即得到一种稳定的排序算法。

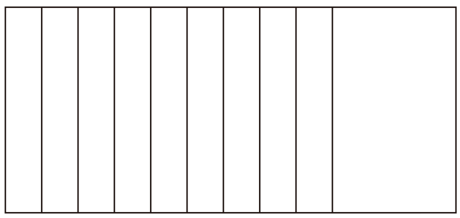


图 5.8 counting sort animate example

性质

稳定性

计数排序是一种稳定的排序算法。

时间复杂度

计数排序的时间复杂度为 $O(n+w)$，其中 w 代表待排序数据的值域大小。

代码实现

伪代码

Input. An array A consisting of n positive integers no greater than w.
Output. Array A after sorting in nondecreasing order stably.
Method.
for $i \leftarrow 0$ **to** w
  $cnt[i] \leftarrow 0$
for $i \leftarrow 1$ **to** n
  $cnt[A[i]] \leftarrow cnt[A[i]] + 1$
for $i \leftarrow 1$ **to** w
  $cnt[i] \leftarrow cnt[i] + cnt[i-1]$
for $i \leftarrow n$ **downto** 1
  $B[cnt[A[i]]] \leftarrow A[i]$
  $cnt[A[i]] \leftarrow cnt[A[i]] - 1$
return B

```cpp
const int N = 100010;
const int W = 100010;

int n, w, a[N], cnt[W], b[N];

void counting_sort() {
  memset(cnt, 0, sizeof(cnt));
  for (int i = 1; i <= n; ++i) ++cnt[a[i]];
  for (int i = 1; i <= w; ++i) cnt[i] += cnt[i - 1];
  for (int i = n; i >= 1; --i) b[cnt[a[i]]--] = a[i];
}
```

```python
N = W = 100010
n = w = 0
a = b = [0] * N
cnt = [0] * W

def counting_sort():
    for i in range(1, n + 1):
        cnt[a[i]] += 1
    for i in range(1, w + 1):
        cnt[i] += cnt[i - 1]
    for i in range(n, 0, -1):
        b[cnt[a[i]] - 1] = a[i]
        cnt[a[i]] -= 1
```

参考资料与注释

[1] 计数排序 - 维基百科 ， 自由的百科全书

5.7.6 基数排序

" 提醒"

本页面要介绍的不是 **计数排序**。

本页面将简要介绍基数排序。

定义

基数排序（英语：Radix sort）是一种非比较型的排序算法，最早用于解决卡片排序的问题。基数排序将待排序的元素拆分为 k 个关键字，逐一对各个关键字排序后完成对所有元素的排序。

如果是从第 1 关键字到第 k 关键字顺序进行比较，则该基数排序称为 MSD（Most Significant Digit first）基数排序；

如果是从第 k 关键字到第 1 关键字顺序进行比较，则该基数排序称为 LSD（Least Significant Digit first）基数排序。

k - 关键字元素的比较

下面用 a_i 表示元素 a 的第 i 关键字。

假如元素有 k 个关键字，对于两个元素 a 和 b，默认的比较方法是：

- 比较两个元素的第 1 关键字 a_1 和 b_1，如果 $a_1 < b_1$ 则 $a < b$，如果 $a_1 > b_1$ 则 $a > b$，如果 $a_1 = b_1$ 则进行下一步；
- 比较两个元素的第 2 关键字 a_2 和 b_2，如果 $a_2 < b_2$ 则 $a < b$，如果 $a_2 > b_2$ 则 $a > b$，如果 $a_2 = b_2$ 则进行下一步；
- ……
- 比较两个元素的第 k 关键字 a_k 和 b_k，如果 $a_k < b_k$ 则 $a < b$，如果 $a_k > b_k$ 则 $a > b$，如果 $a_k = b_k$ 则 $a = b$。

例子：

- 如果对自然数进行比较，将自然数按个位对齐后往高位补齐 0，则一个数字从左往右数第 i 位数就可以作为第 i 关键字；
- 如果对字符串基于字典序进行比较，一个字符串从左往右数第 i 个字符就可以作为第 i 关键字；
- C++ 自带的 `std::pair` 与 `std::tuple` 的默认比较方法与上述的相同。

MSD 基数排序

基于 k - 关键字元素的比较方法，可以想到：先比较所有元素的第 1 关键字，就可以确定出各元素大致的大小关系；然后对**具有相同第 1 关键字的元素**，再比较它们的第 2 关键字……以此类推。

由于是从第 1 关键字到第 k 关键字顺序进行比较，由上述思想导出的排序算法称为 MSD（Most Significant Digit first）基数排序。

算法流程

将待排序的元素拆分为 k 个关键字，先对第 1 关键字进行稳定排序，然后对于每组**具有相同关键字的元素**再对第 2 关键字进行稳定排序（递归执行）……最后对于每组**具有相同关键字的元素**再对第 k 关键字进行稳定排序。

一般而言，我们默认基数排序是稳定的，所以在 MSD 基数排序中，我们也仅仅考虑借助**稳定算法**（通常使用计数排序）完成内层对关键字的排序。

正确性参考上文 k - 关键字元素的比较。

参考代码

对自然数排序

下面是使用迭代式 MSD 基数排序对 `unsigned int` 范围内元素进行排序的 C++ 参考代码，可调整 W 和 $\log_2 W$ 的值（建议将 $\log_2 W$ 设为 2^k 以便位运算优化）。

```cpp
#include <algorithm>
#include <stack>
#include <tuple>
#include <vector>

using std::copy;  // from <algorithm>
using std::make_tuple;
using std::stack;
using std::tie;
using std::tuple;
using std::vector;

typedef unsigned int u32;
typedef unsigned int* u32ptr;

void MSD_radix_sort(u32ptr first, u32ptr last) {
  const size_t maxW = 0x100000000llu;
  const u32 maxlogW = 32;  // = log_2 W

  const u32 W = 256;  // 计数排序的值域
  const u32 logW = 8;
  const u32 mask = W - 1;  // 用位运算替代取模，详见下面的 key 函数

  u32ptr tmp =
      (u32ptr)calloc(last - first, sizeof(u32));  // 计数排序用的输出空间

  typedef tuple<u32ptr, u32ptr, u32> node;
  stack<node, vector<node>> s;
  s.push(make_tuple(first, last, maxlogW - logW));

  while (!s.empty()) {
    u32ptr begin, end;
    size_t shift, length;

    tie(begin, end, shift) = s.top();
    length = end - begin;
    s.pop();

    if (begin + 1 >= end) continue;  // elements <= 1

    // 计数排序
    u32 cnt[W] = {};
    auto key = [](const u32 x, const u32 shift) { return (x >> shift) & mask; };

    for (u32ptr it = begin; it != end; ++it) ++cnt[key(*it, shift)];
    for (u32 value = 1; value < W; ++value) cnt[value] += cnt[value - 1];

    // 求完前缀和后，计算相同关键字的元素范围
    if (shift >= logW) {
```

```
      s.push(make_tuple(begin, begin + cnt[0], shift - logW));
      for (u32 value = 1; value < W; ++value)
        s.push(make_tuple(begin + cnt[value - 1], begin + cnt[value],
                          shift - logW));
    }

    u32ptr it = end;
    do {
      --it;
      --cnt[key(*it, shift)];
      tmp[cnt[key(*it, shift)]] = *it;
    } while (it != begin);

    copy(tmp, tmp + length, begin);
  }
}
```

对字符串排序

下面是使用迭代式 MSD 基数排序对 空终止字节字符串[2] 基于字典序进行排序的 C++ 参考代码：

```
#include <algorithm>
#include <stack>
#include <tuple>
#include <vector>

using std::copy;  // from <algorithm>
using std::make_tuple;
using std::stack;
using std::tie;
using std::tuple;
using std::vector;

typedef char* NTBS;  // 空终止字节字符串
typedef NTBS* NTBSptr;

void MSD_radix_sort(NTBSptr first, NTBSptr last) {
  const size_t W = 128;
  const size_t logW = 7;
  const size_t mask = W - 1;

  NTBSptr tmp = (NTBSptr)calloc(last - first, sizeof(NTBS));

  typedef tuple<NTBSptr, NTBSptr, size_t> node;
  stack<node, vector<node>> s;
  s.push(make_tuple(first, last, 0));

  while (!s.empty()) {
    NTBSptr begin, end;
    size_t index, length;

    tie(begin, end, index) = s.top();
    length = end - begin;
    s.pop();
```

```
    if (begin + 1 >= end) continue;  // elements <= 1

    // 计数排序
    size_t cnt[W] = {};
    auto key = [](const NTBS str, const size_t index) { return str[index]; };

    for (NTBSptr it = begin; it != end; ++it) ++cnt[key(*it, index)];
    for (char ch = 1; value < W; ++value) cnt[ch] += cnt[ch - 1];

    // 求完前缀和后，计算相同关键字的元素范围
    // 对于 NTBS，如果此刻末尾的字符是 \0 则说明这两个字符串相等，不必继续迭代
    for (char ch = 1; ch < W; ++ch)
      s.push(make_tuple(begin + cnt[ch - 1], begin + cnt[ch], index + 1));

    NTBSptr it = end;
    do {
      --it;
      --cnt[key(*it, index)];
      tmp[cnt[key(*it, index)]] = *it;
    } while (it != begin);

    copy(tmp, tmp + length, begin);
  }

  free(tmp);
}
```

由于两个字符串的比较很容易冲上 $O(n)$ 的线性复杂度，因此在字符串排序这件事情上，MSD 基数排序比大多数基于比较的排序算法在时间复杂度和实际用时上都更加优秀。

与桶排序的关系

前置知识：桶排序

桶排序需要其它的排序算法来完成对每个桶内部元素的排序。但实际上，完全可以对每个桶继续执行桶排序，直至某一步桶的元素数量 ≤ 1。

因此 MSD 基数排序的另一种理解方式是：使用桶排序实现的桶排序。

也因此，可以提出 MSD 基数排序在时间常数上的一种优化方法：假如到某一步桶的元素数量 $\leq B$（B 是自己选的常数），则直接执行插入排序然后返回，降低递归次数。

LSD 基数排序

MSD 基数排序从第 1 关键字到第 k 关键字顺序进行比较，为此需要借助递归或迭代来实现，时间常数还是较大，而且在比较自然数上还是略显不便。

而将递归的操作反过来：从第 k 关键字到第 1 关键字顺序进行比较，就可以得到 LSD（Least Significant Digit first）基数排序，不使用递归就可以完成的排序算法。

算法流程

将待排序的元素拆分为 k 个关键字，然后先对**所有元素**的第 k 关键字进行稳定排序，再对**所有元素**的第 $k-1$ 关键字进行稳定排序，再对**所有元素**的第 $k-2$ 关键字进行稳定排序……最后对**所有元素**的第 1 关键字进行稳定排序，这样就完成了对整个待排序序列的稳定排序。

LSD 基数排序也需要借助一种**稳定算法**完成内层对关键字的排序。同样的，通常使用计数排序来完成。

LSD 基数排序的正确性可以参考《算法导论（第三版）》第 8.3-3 题的解法[3] 或参考下面的解释：

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

图 5.9 一个 LSD 基数排序全流程的例子

正确性

回顾一下 k - 关键字元素的比较方法，

- 假如想通过 a_1 和 b_1 就比较出两个元素 a 和 b 的大小，则需要提前知道通过比较 a_2 和 b_2 得到的结论，以便于应对 $a_1 = b_1$ 的情况；
- 而想通过 a_2 和 b_2 就比较出两个元素 a 和 b 的大小，则需要提前知道通过比较 a_3 和 b_3 得到的结论，以便于应对 $a_2 = b_2$ 的情况；
- ……
- 而想通过 a_{k-1} 和 b_{k-1} 就比较出两个元素 a 和 b 的大小，则需要提前知道通过比较 a_k 和 b_k 得到的结论，以便于应对 $a_{k-1} = b_{k-1}$ 的情况；
- a_k 和 b_k 可以直接比较。

现在，将顺序反过来：

- a_k 和 b_k 可以直接比较；
- 而知道通过比较 a_k 和 b_k 得到的结论后，就可以得到比较 a_{k-1} 和 b_{k-1} 的结论；
- ……
- 而知道通过比较 a_2 和 b_2 得到的结论后，就可以得到比较 a_1 和 b_1 的结论；
- 而知道通过比较 a_1 和 b_1 得到的结论后，就最终得到了比较 a 和 b 的结论。

在这个过程中，对每个关键字边比较边重排元素的顺序，就得到了 LSD 基数排序。

伪代码

Input. An array A consisting of n elements, where each element has k keys.
Output. Array A will be sorted in nondecreasing order stably.
Method.
for $i \leftarrow k$ **down to** 1
    sort A into nondecreasing order by the i-th key stably

参考代码

下面是使用 LSD 基数排序实现的对 k - 关键字元素的排序。

```
const int N = 100010;
const int W = 100010;
const int K = 100;

int n, w[K], k, cnt[W];

struct Element {
  int key[K];
```

```
  bool operator<(const Element& y) const {
    // 两个元素的比较流程
    for (int i = 1; i <= k; ++i) {
      if (key[i] == y.key[i]) continue;
      return key[i] < y.key[i];
    }
    return false;
  }
} a[N], b[N];

void counting_sort(int p) {
  memset(cnt, 0, sizeof(cnt));
  for (int i = 1; i <= n; ++i) ++cnt[a[i].key[p]];
  for (int i = 1; i <= w[p]; ++i) cnt[i] += cnt[i - 1];
  // 为保证排序的稳定性，此处循环 i 应从 n 到 1
  // 即当两元素关键字的值相同时，原先排在后面的元素在排序后仍应排在后面
  for (int i = n; i >= 1; --i) b[cnt[a[i].key[p]]--] = a[i];
  memcpy(a, b, sizeof(a));
}

void radix_sort() {
  for (int i = k; i >= 1; --i) {
    // 借助计数排序完成对关键字的排序
    counting_sort(i);
  }
}
```

实际上并非必须从后往前枚举才是稳定排序，只需对 `cnt` 数组进行等价于 `std::exclusive_scan` 的操作即可。

" 例题 洛谷 P1177【模板】快速排序[4]"

给出 n 个正整数，从小到大输出。

```
#include <algorithm>
#include <iostream>
#include <utility>

void radix_sort(int n, int a[]) {
  int *b = new int[n];  // 临时空间
  int *cnt = new int[1 << 8];
  int mask = (1 << 8) - 1;
  int *x = a, *y = b;
  for (int i = 0; i < 32; i += 8) {
    for (int j = 0; j != (1 << 8); ++j) cnt[j] = 0;
    for (int j = 0; j != n; ++j) ++cnt[x[j] >> i & mask];
    for (int sum = 0, j = 0; j != (1 << 8); ++j) {
      // 等价于 std::exclusive_scan(cnt, cnt + (1 << 8), cnt, 0);
      sum += cnt[j], cnt[j] = sum - cnt[j];
    }
    for (int j = 0; j != n; ++j) y[cnt[x[j] >> i & mask]++] = x[j];
    std::swap(x, y);
  }
  delete[] cnt;
  delete[] b;
}
```

```
int main() {
  std::ios::sync_with_stdio(false);
  std::cin.tie(0);
  int n;
  std::cin >> n;
  int *a = new int[n];
  for (int i = 0; i < n; ++i) std::cin >> a[i];
  radix_sort(n, a);
  for (int i = 0; i < n; ++i) std::cout << a[i] << ' ';
  delete[] a;
  return 0;
}
```

性质

稳定性

如果对内层关键字的排序是稳定的，则 MSD 基数排序和 LSD 基数排序都是稳定的排序算法。

时间复杂度

通常而言，基数排序比基于比较的排序算法（比如快速排序）要快。但由于需要额外的内存空间，因此当内存空间稀缺时，原地置换算法（比如快速排序）或许是个更好的选择。[1]

一般来说，如果每个关键字的值域都不大，就可以使用 计数排序 作为内层排序，此时的复杂度为 $O(kn+\sum\limits_{i=1}^{k} w_i)$，其中 w_i 为第 i 关键字的值域大小。如果关键字值域很大，就可以直接使用基于比较的 $O(nk\log n)$ 排序而无需使用基数排序了。

空间复杂度

MSD 基数排序和 LSD 基数排序的空间复杂度都为 $O(k+n)$。

参考资料与注释

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.*Introduction to Algorithms*(3rd ed.). MIT Press and McGraw-Hill, 2009. ISBN 978-0-262-03384-8. ”8.3 Radix sort”, pp. 199.

[2] 空终止字节字符串

[3] 《算法导论（第三版）》第 8.3-3 题的解法

[4] 洛谷 P1177【模板】快速排序

5.7.7 快速排序

本页面将简要介绍快速排序。

定义

快速排序（英语：Quicksort），又称分区交换排序（英语：partition-exchange sort），简称「快排」，是一种被广泛运用的排序算法。

基本原理与实现

过程

快速排序的工作原理是通过 分治 的方式来将一个数组排序。

快速排序分为三个过程：

1. 将数列划分为两部分（要求保证相对大小关系）；
2. 递归到两个子序列中分别进行快速排序；
3. 不用合并，因为此时数列已经完全有序。

和归并排序不同，第一步并不是直接分成前后两个序列，而是在分的过程中要保证相对大小关系。具体来说，第一步要是要把数列分成两个部分，然后保证前一个子数列中的数都小于后一个子数列中的数。为了保证平均时间复杂度，一般是随机选择一个数 m 来当做两个子数列的分界。

之后，维护一前一后两个指针 p 和 q，依次考虑当前的数是否放在了应该放的位置（前还是后）。如果当前的数没放对，比如说如果后面的指针 q 遇到了一个比 m 小的数，那么可以交换 p 和 q 位置上的数，再把 p 向后移一位。当前的数的位置全放对后，再移动指针继续处理，直到两个指针相遇。

其实，快速排序没有指定应如何具体实现第一步，不论是选择 m 的过程还是划分的过程，都有不止一种实现方法。

第三步中的序列已经分别有序且第一个序列中的数都小于第二个数，所以直接拼接起来就好了。

```
struct Range {
  int start, end;

  Range(int s = 0, int e = 0) { start = s, end = e; }
};

template <typename T>
void quick_sort(T arr[], const int len) {
  if (len <= 0) return;
  Range r[len];
  int p = 0;
  r[p++] = Range(0, len - 1);
  while (p) {
    Range range = r[--p];
    if (range.start >= range.end) continue;
    T mid = arr[range.end];
    int left = range.start, right = range.end - 1;
    while (left < right) {
      while (arr[left] < mid && left < right) left++;
      while (arr[right] >= mid && left < right) right--;
      std::swap(arr[left], arr[right]);
    }
    if (arr[left] >= arr[range.end])
      std::swap(arr[left], arr[range.end]);
    else
      left++;
    r[p++] = Range(range.start, left - 1);
    r[p++] = Range(left + 1, range.end);
  }
}
```

```python
def quick_sort(alist, first, last):
    if first >= last:
        return
    mid_value = alist[first]
    low = first
    high = last
    while low < high:
        while low < high and alist[high] >= mid_value:
            high -= 1
        alist[low] = alist[high]
        while low < high and alist[low] < mid_value:
            low += 1
        alist[high] = alist[low]
    alist[low] = mid_value
    quick_sort(alist, first, low - 1)
    quick_sort(alist, low + 1, last)
```

性质

稳定性

快速排序是一种不稳定的排序算法。

时间复杂度

快速排序的最优时间复杂度和平均时间复杂度为 $O(n\log n)$，最坏时间复杂度为 $O(n^2)$。

对于最优情况，每一次选择的分界值都是序列的中位数，此时算法时间复杂度满足的递推式为 $T(n) = 2T(\frac{n}{2}) + \Theta(n)$，由主定理，$T(n) = \Theta(n\log n)$。

对于最坏情况，每一次选择的分界值都是序列的最值，此时算法时间复杂度满足的递推式为 $T(n) = T(n-1) + \Theta(n)$，累加可得 $T(n) = \Theta(n^2)$。

对于平均情况，每一次选择的分界值可以看作是等概率随机的。

" 证明"

下面我们来证明这种情况下算法的时间复杂度是 $O(n\log n)$。

引理 1： 当对 n 个元素的数组进行快速排序时，假设在划分元素时总共的比较次数为 X，则快速排序的时间复杂度是 $O(n+X)$。

由于在每次划分元素的过程中，都会选择一个元素作为分界，所以划分元素的过程至多发生 n 次。又由于划分元素的过程中比较的次数和其他基础操作的次数在一个数量级，所以总时间复杂度是 $O(n+X)$ 的。

设 a_i 为原数组中第 i 小的数，定义 $A_{i,j}$ 为 $\{a_i, a_{i+1}, \dots, a_j\}$，$X_{i,j}$ 是一个取值为 0 或者 1 的离散随机变量表示在排序过程中 a_i 是否和 a_j 发生比较。

显然每次选取的分界值是不同的，而元素只会和分界值比较，所以总比较次数

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j}$$

由期望的线性性，

$$
\begin{aligned}
E[X] &= E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{i,j}\right] \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{i,j}] \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} P(a_i \text{ 和 } a_j \text{ 比较})
\end{aligned}
$$

引理 2： a_i 和 a_j 比较的充要条件是 a_i 或 a_j 是集合 $A_{i,j}$ 中第一个被选中的分界值。

先证必要性，即若 a_i 和 a_j 都不是集合 $A_{i,j}$ 中第一个被选中的分界值，则 a_i 不和 a_j 比较。

若 a_i 和 a_j 都不是集合 $A_{i,j}$ 中第一个被选中的分界值，则一定存在一个 x 满足 $i < x < j$，使得 a_x 是 $A_{i,j}$ 中第一个被选中的分界值。在以 a_x 为分界值的划分中，a_i 和 a_j 被划分到数组的两个不同的子序列中，所以之后 a_i 和 a_j 一定不会比较。又因为元素只和分界值比较，所以 a_i 和 a_j 在此次划分前和划分中没有比较。所以 a_i 不和 a_j 比较。

再证充分性，即若 a_i 或 a_j 是集合 $A_{i,j}$ 中第一个被选中的分界值，则 a_i 和 a_j 比较。

不失一般地，假设 a_i 是集合 $A_{i,j}$ 中第一个被选中的分界值。由于 $A_{i,j}$ 中没有其他数选为分界值，所以 $A_{i,j}$ 中的元素都在数组的同一子序列中。在以 a_i 为分界值的划分中，a_i 和当前子序列中所有元素都进行了比较，所以 a_i 和 a_j 进行了比较。

考虑计算 $P(a_i$ 和 a_j 比较$)$。在 $A_{i,j}$ 中某个元素被选为分界值之前，$A_{i,j}$ 中的元素都在数组的同一子序列中。所以 $A_{i,j}$ 中每个元素都会被等可能地第一个被选为分界值。由于 $A_{i,j}$ 中有 $j-i+1$ 个元素，由引理 2，

$$
P(a_i\text{和}a_j\text{比较}) = P(a_i\text{或}a_j\text{是集合}A_{i,j}\text{中第一个被选中的分界值}) = \frac{2}{j-i+1}
$$

所以

$$
\begin{aligned}
E[X] &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} P(a_i \text{ 和 } a_j \text{ 比较}) \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1}\sum_{k=2}^{n-i+1} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\log n) \\
&= O(n\log n)
\end{aligned}
$$

由此，快速排序的期望时间复杂度为 $O(n\log n)$。

在实践中，几乎不可能达到最坏情况，而快速排序的内存访问遵循局部性原理，所以多数情况下快速排序的表现大幅优于堆排序等其他复杂度为 $O(n\log n)$ 的排序算法。[1]

优化

朴素优化思想

如果仅按照上文所述的基本思想来实现快速排序（或者是直接照抄模板）的话，那大概率是通不过 P1177【模板】快速排序[5] 这道模板的。因为有毒瘤数据能够把朴素的快速排序卡成 $O(n^2)$。

所以，我们需要对朴素快速排序思想加以优化。较为常见的优化思路有以下三种[3]。

- 通过**三数取中（即选取第一个、最后一个以及中间的元素中的中位数）**的方法来选择两个子序列的分界元素（即比较基准）。这样可以避免极端数据（如升序序列或降序序列）带来的退化；

- 当序列较短时，使用**插入排序**的效率更高；
- 每趟排序后，**将与分界元素相等的元素聚集在分界元素周围**，这样可以避免极端数据（如序列中大部分元素都相等）带来的退化。

下面列举了几种较为成熟的快速排序优化方式。

三路快速排序

定义

三路快速排序（英语：3-way Radix Quicksort）是快速排序和 基数排序 的混合。它的算法思想基于 荷兰国旗问题[6] 的解法。

过程

与原始的快速排序不同，三路快速排序在随机选取分界点 m 后，将待排数列划分为三个部分：小于 m、等于 m 以及大于 m。这样做即实现了将与分界元素相等的元素聚集在分界元素周围这一效果。

性质

三路快速排序在处理含有多个重复值的数组时，效率远高于原始快速排序。其最佳时间复杂度为 $O(n)$。

实现

三路快速排序实现起来非常简单，下面给出了一种三路快排的 C++ 实现。

```cpp
// 模板的 T 参数表示元素的类型，此类型需要定义小于（<）运算
template <typename T>
// arr 为需要被排序的数组，len 为数组长度
void quick_sort(T arr[], const int len) {
  if (len <= 1) return;
  // 随机选择基准（pivot）
  const T pivot = arr[rand() % len];
  // i：当前操作的元素下标
  // arr[0, j)：存储小于 pivot 的元素
  // arr[k, len)：存储大于 pivot 的元素
  int i = 0, j = 0, k = len;
  // 完成一趟三路快排，将序列分为：
  // 小于 pivot 的元素 | 等于 pivot 的元素 | 大于 pivot 的元素
  while (i < k) {
    if (arr[i] < pivot)
      swap(arr[i++], arr[j++]);
    else if (pivot < arr[i])
      swap(arr[i], arr[--k]);
    else
      i++;
  }
  // 递归完成对于两个子序列的快速排序
  quick_sort(arr, j);
  quick_sort(arr + k, len - k);
}
```

```python
def quick_sort(arr, l, r):
    if l >= r:
        return
    random_index = random.randint(l, r)
    pivot = arr[random_index]
    arr[l], arr[random_index] = arr[random_index], arr[l]
```

```
    i = l + 1
    j = l
    k = r + 1
    while i < k:
        if arr[i] < pivot:
            arr[i], arr[j + 1] = arr[j + 1], arr[i]
            j += 1
            i += 1
        elif arr[i] > pivot:
            arr[i], arr[k - 1] = arr[k - 1], arr[i]
            k -= 1
        else:
            i += 1
    arr[l], arr[j] = arr[j], arr[l]
    quick_sort(arr, l, j - 1)
    quick_sort(arr, k, r)
```

内省排序

定义

内省排序（英语：Introsort 或 Introspective sort）[4] 是快速排序和 堆排序 的结合，由 David Musser 于 1997 年发明。内省排序其实是对快速排序的一种优化，保证了最差时间复杂度为 $O(n \log n)$。

性质

内省排序将快速排序的最大递归深度限制为 $\lfloor \log_2 n \rfloor$，超过限制时就转换为堆排序。这样既保留了快速排序内存访问的局部性，又可以防止快速排序在某些情况下性能退化为 $O(n^2)$。

实现

从 2000 年 6 月起，SGI C++ STL 的 `stl_algo.h` 中 `sort()` 函数的实现采用了内省排序算法。

线性找第 k 大的数

在下面的代码示例中，第 k 大的数被定义为序列排成升序时，第 k 个位置上的数（编号从 0 开始）。

找第 k 大的数（K-th order statistic），最简单的方法是先排序，然后直接找到第 k 大的位置的元素。这样做的时间复杂度是 $O(n \log n)$，对于这个问题来说很不划算。

我们可以借助快速排序的思想解决这个问题。考虑快速排序的划分过程，在快速排序的「划分」结束后，数列 $A_p \cdots A_r$ 被分成了 $A_p \cdots A_q$ 和 $A_{q+1} \cdots A_r$，此时可以按照左边元素的个数（$q - p + 1$）和 k 的大小关系来判断是只在左边还是只在右边递归地求解。

和快速排序一样，该方法的时间复杂度依赖于每次划分时选择的分界值。如果采用随机选取分界值的方式，可以证明在期望意义下，程序的时间复杂度为 $O(n)$。

实现（C++）

```
// 模板的 T 参数表示元素的类型，此类型需要定义小于（<）运算
template <typename T>
// arr 为查找范围数组，rk 为需要查找的排名（从 0 开始），len 为数组长度
T find_kth_element(T arr[], int rk, const int len) {
  if (len <= 1) return arr[0];
  // 随机选择基准（pivot）
  const T pivot = arr[rand() % len];
  // i：当前操作的元素
  // j：第一个等于 pivot 的元素
```

```
  // k: 第一个大于 pivot 的元素
  int i = 0, j = 0, k = len;
  // 完成一趟三路快排，将序列分为：
  // 小于 pivot 的元素 | 等于 pivot 的元素 | 大于 pivot 的元素
  while (i < k) {
    if (arr[i] < pivot)
      swap(arr[i++], arr[j++]);
    else if (pivot < arr[i])
      swap(arr[i], arr[--k]);
    else
      i++;
  }
  // 根据要找的排名与两条分界线的位置，去不同的区间递归查找第 k 大的数
  // 如果小于 pivot 的元素个数比 k 多，则第 k 大的元素一定是一个小于 pivot 的元素
  if (rk < j) return find_kth_element(arr, rk, j);
  // 否则，如果小于 pivot 和等于 pivot 的元素加起来也没有 k 多，
  // 则第 k 大的元素一定是一个大于 pivot 的元素
  else if (rk >= k)
    return find_kth_element(arr + k, rk - k, len - k);
  // 否则，pivot 就是第 k 大的元素
  return pivot;
}
```

改进：中位数中的中位数

中位数中的中位数（英文：Median of medians），提供了一种确定性的选择划分过程中分界值的方法，从而能够让找第 k 大的数算法在最坏情况下也能实现线性时间复杂度。

该算法的流程如下：

1. 将整个序列划分为 $\left\lfloor \frac{n}{5} \right\rfloor$ 组，每组元素数不超过 5 个；
2. 寻找每组元素的中位数（因为元素个数较少，可以直接使用 插入排序 等算法）。
3. 找出这 $\left\lfloor \frac{n}{5} \right\rfloor$ 组元素中位数中的中位数。将该元素作为前述算法中每次划分时的分界值即可。

时间复杂度证明

下面将证明，该算法在最坏情况下的时间复杂度为 $O(n)$。设 $T(n)$ 为问题规模为 n 时，解决问题需要的计算量。

先分析前两步——划分与寻找中位数。由于划分后每组内的元素数量非常少，可以认为寻找一组元素的中位数的时间复杂度为 $O(1)$。因此找出所有 $\left\lfloor \frac{n}{5} \right\rfloor$ 组元素中位数的时间复杂度为 $O(n)$。

接下来分析第三步——递归过程。这一步进行了两次递归调用：第一次是寻找各组中位数中的中位数，需要的开销显然为 $T(\frac{n}{5})$，第二次是进入分界值的左侧部分或右侧部分。根据我们选取的划分元素，有 $\frac{1}{2} \times \left\lfloor \frac{n}{5} \right\rfloor = \left\lfloor \frac{n}{10} \right\rfloor$ 组元素的中位数小于分界值，这几组元素中，比中位数还小的元素也一定比分界值要小，从而整个序列中小于分界值的元素至少有 $3 \times \left\lfloor \frac{n}{10} \right\rfloor = \left\lfloor \frac{3n}{10} \right\rfloor$ 个。同理，整个序列中大于分界值的元素也至少有 $\left\lfloor \frac{3n}{10} \right\rfloor$ 个。因此，分界值的左边或右边至多有 $\frac{7n}{10}$ 个元素，这次递归的时间开销的上界为 $T(\frac{7n}{10})$。

综上，我们可以列出这样的不等式：

$$T(n) \leq T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n)$$

假设 $T(n) = O(n)$ 在问题规模足够小时成立。根据定义，此时有 $T(n) \leq cn$，其中 c 为一正常数。将不等式右边

的所有 $T(n)$ 进行代换：

$$
\begin{aligned}
T(n) &\leq T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n) \\
&\leq \frac{cn}{5} + \frac{7cn}{10} + O(n) \\
&\leq \frac{9cn}{10} + O(n) \\
&= O(n)
\end{aligned}
$$

到这里我们就证明了，该算法在最坏情况下也具有 $O(n)$ 的时间复杂度。

参考资料与注释

[1] C++ 性能榨汁机之局部性原理 - I'm Root lee !

[2] 算法实现 / 排序 / 快速排序 - 维基教科书， 自由的教学读本 [2-1] [2-2] [2-3]

[3] 三种快速排序以及快速排序的优化

[4] introsort

[5] P1177【模板】快速排序

[6] 荷兰国旗问题

5.7.8 归并排序

定义

归并排序（merge sort[1]）是高效的基于比较的稳定排序算法。

性质

归并排序基于分治思想将数组分段排序后合并，时间复杂度在最优、最坏与平均情况下均为 $\Theta(n \log n)$，空间复杂度为 $\Theta(n)$。

归并排序可以只使用 $\Theta(1)$ 的辅助空间，但为便捷通常使用与原数组等长的辅助数组。

过程

合并

归并排序最核心的部分是合并（merge）过程：将两个有序的数组 `a[i]` 和 `b[j]` 合并为一个有序数组 `c[k]`。

从左往右枚举 `a[i]` 和 `b[j]`，找出最小的值并放入数组 `c[k]`；重复上述过程直到 `a[i]` 和 `b[j]` 有一个为空时，将另一个数组剩下的元素放入 `c[k]`。

为保证排序的稳定性，前段首元素小于或等于后段首元素时（`a[i] <= b[j]`）而非小于时（`a[i] < b[j]`）就要作为最小值放入 `c[k]`。

实现

```cpp
void merge(const int *a, size_t aLen, const int *b, size_t bLen, int *c) {
  size_t i = 0, j = 0, k = 0;
  while (i < aLen && j < bLen) {
    if (b[j] < a[i]) {  // <!> 先判断 b[j] < a[i]，保证稳定性
      c[k] = b[j];
      ++j;
    } else {
      c[k] = a[i];
      ++i;
    }
    ++k;
  }
  // 此时一个数组已空，另一个数组非空，将非空的数组并入 c 中
  for (; i < aLen; ++i, ++k) c[k] = a[i];
  for (; j < bLen; ++j, ++k) c[k] = b[j];
}
```

```cpp
void merge(const int *aBegin, const int *aEnd, const int *bBegin,
           const int *bEnd, int *c) {
  while (aBegin != aEnd && bBegin != bEnd) {
    if (*bBegin < *aBegin) {
      *c = *bBegin;
      ++bBegin;
    } else {
      *c = *aBegin;
      ++aBegin;
    }
    ++c;
  }
  for (; aBegin != aEnd; ++aBegin, ++c) *c = *aBegin;
  for (; bBegin != bEnd; ++bBegin, ++c) *c = *bBegin;
}
```

```
也可使用 `<algorithm>` 库的 `merge` 函数，用法与上述指针式写法的相同。
```

```python
def merge(a, b):
    i, j = 0, 0
    c = []
    while(i < len(a) and j < len(b)):
        # <!> 先判断 b[j] < a[i]，保证稳定性
        if(b[j] < a[i]):
            c.append(b[j])
            j += 1
        else:
            c.append(a[i])
            i += 1
    # 此时一个数组已空，另一个数组非空，将非空的数组并入 c 中
    c.extend(a[i:])
    c.extend(b[j:])
    return c
```

分治法实现归并排序

1. 当数组长度为 1 时，该数组就已经是有序的，不用再分解。

2. 当数组长度大于 1 时，该数组很可能不是有序的。此时将该数组分为两段，再分别检查两个数组是否有序（用第 1 条）。如果有序，则将它们合并为一个有序数组；否则对不有序的数组重复第 2 条，再合并。

用数学归纳法可以证明该流程可以将一个数组转变为有序数组。

为保证排序的复杂度，通常将数组分为尽量等长的两段（$mid = \left\lfloor \dfrac{l+r}{2} \right\rfloor$）。

实现

注意下面的代码所表示的区间分别是 $[l,r)$，$[l,mid)$，$[mid,r)$。

```
void merge_sort(int *a, int l, int r) {
  if (r - l <= 1) return;
  // 分解
  int mid = l + ((r - l) >> 1);
  merge_sort(a, l, mid), merge_sort(a, mid, r);
  // 合并
  int tmp[1024] = {};  // 请结合实际情况设置 tmp 数组的长度 (与 a 相同), 或使用
                       // vector; 先将合并的结果放在 tmp 里, 再返回到数组 a
  merge(a + l, a + mid, a + mid, a + r, tmp + l);  // pointer-style merge
  for (int i = l; i < r; ++i) a[i] = tmp[i];
}
```

```
def merge_sort(a, ll, rr):
    if rr - ll <= 1:
        return
    # 分解
    mid = (rr + ll) // 2
    merge_sort(a, ll, mid)
    merge_sort(a, mid, rr)
    # 合并
    a[ll:rr] = merge(a[ll:mid], a[mid:rr])
```

倍增法实现归并排序

已知当数组长度为 1 时，该数组就已经是有序的。

将数组全部切成长度为 1 的段。

从左往右依次合并两个长度为 1 的有序段，得到一系列长度 ≤ 2 的有序段；

从左往右依次合并两个长度 ≤ 2 的有序段，得到一系列长度 ≤ 4 的有序段；

从左往右依次合并两个长度 ≤ 4 的有序段，得到一系列长度 ≤ 8 的有序段；

……

重复上述过程直至数组只剩一个有序段，该段就是排好序的原数组。

" 为什么是 $\le n$ 而不是 $= n$"

数组的长度很可能不是 2^x，此时在最后就可能出现长度不完整的段，可能出现最后一个段是独立的情况。

实现

```
void merge_sort(int *a, size_t n) {
  int tmp[1024] = {};  // 请结合实际情况设置 tmp 数组的长度 (与 a 相同), 或使用
                       // vector; 先将合并的结果放在 tmp 里, 再返回到数组 a
  for (size_t seg = 1; seg < n; seg <<= 1) {
    for (size_t left1 = 0; left1 < n - seg;
         left1 += seg + seg) {  // n - seg: 如果最后只有一个段就不用合并
```

```
      size_t right1 = left1 + seg;
      size_t left2 = right1;
      size_t right2 = std::min(left2 + seg, n);  // <!> 注意最后一个段的边界
      merge(a + left1, a + right1, a + left2, a + right2,
            tmp + left1);  // pointer-style merge
      for (size_t i = left1; i < right2; ++i) a[i] = tmp[i];
    }
  }
}
```

```python
def merge_sort(a):
    seg = 1
    while seg < len(a):
        for l1 in range(0, len(a) - seg, seg + seg):
            r1 = l1 + seg
            l2 = r1
            r2 = l2 + seg
            a[l1:r2] = merge(a[l1:r1], a[l2:r2])
    seg <<= 1
```

逆序对

逆序对是 $i < j$ 且 $a_i > a_j$ 的有序数对 (i, j)。

排序后的数组无逆序对，归并排序的合并操作中，每次后段首元素被作为当前最小值取出时，前段剩余元素个数之和即是合并操作减少的逆序对数量；故归并排序计算逆序对数量的额外时间复杂度为 $\Theta(n \log n)$，对于 C/C++ 代码将 `merge` 过程的 `if(b[j] < a[i])` 部分加上 `cnt += aLen - i` 或 `cnt += aEnd - aBegin` 即可，对于 Python 代码将 `merge` 过程的 `if(b[j] < a[i]):` 部分加上 `cnt += len(a) - i` 即可。

此外，逆序对计数即是将元素依次加入数组时统计当前大于其的元素数量，将数组离散化后即是区间求和问题，使用树状数组或线段树解决的时间复杂度为 $O(n \log n)$ 且空间复杂度为 $\Theta(n)$。

外部链接

- Merge Sort - GeeksforGeeks[2]
- 归并排序 - 维基百科，自由的百科全书[3]
- 逆序对 - 维基百科，自由的百科全书[4]

参考资料与注释

[1] merge sort

[2] Merge Sort - GeeksforGeeks

[3] 归并排序 - 维基百科，自由的百科全书

[4] 逆序对 - 维基百科，自由的百科全书

5.7.9 堆排序

本页面将简要介绍堆排序。

定义

堆排序（英语：Heapsort）是指利用 二叉堆 这种数据结构所设计的一种排序算法。堆排序的适用数据结构为数组。

过程

堆排序的本质是建立在堆上的选择排序。

排序

首先建立大顶堆，然后将堆顶的元素取出，作为最大值，与数组尾部的元素交换，并维持残余堆的性质；

之后将堆顶的元素取出，作为次大值，与数组倒数第二位元素交换，并维持残余堆的性质；

以此类推，在第 $n-1$ 次操作后，整个数组就完成了排序。

在数组上建立二叉堆

从根节点开始，依次将每一层的节点排列在数组里。

于是有数组中下标为 `i` 的节点，对应的父结点、左子结点和右子结点如下：

```
iParent(i) = (i - 1) / 2;
iLeftChild(i) = 2 * i + 1;
iRightChild(i) = 2 * i + 2;
```

性质

稳定性

同选择排序一样，由于其中交换位置的操作，所以是不稳定的排序算法。

时间复杂度

堆排序的最优时间复杂度、平均时间复杂度、最坏时间复杂度均为 $O(n\log n)$。

空间复杂度

由于可以在输入数组上建立堆，所以这是一个原地算法。

实现

```
void sift_down(int arr[], int start, int end) {
  // 计算父结点和子结点的下标
  int parent = start;
  int child = parent * 2 + 1;
  while (child <= end) {  // 子结点下标在范围内才做比较
    // 先比较两个子结点大小，选择最大的
    if (child + 1 <= end && arr[child] < arr[child + 1]) child++;
    // 如果父结点比子结点大，代表调整完毕，直接跳出函数
    if (arr[parent] >= arr[child])
      return;
    else {  // 否则交换父子内容，子结点再和孙结点比较
      swap(arr[parent], arr[child]);
      parent = child;
      child = parent * 2 + 1;
```

```
    }
  }
}

void heap_sort(int arr[], int len) {
  // 从最后一个节点的父节点开始 sift down 以完成堆化 (heapify)
  for (int i = (len - 1 - 1) / 2; i >= 0; i--) sift_down(arr, i, len - 1);
  // 先将第一个元素和已经排好的元素前一位做交换，再重新调整（刚调整的元素之前的元素
), 直到排序完毕
  for (int i = len - 1; i > 0; i--) {
    swap(arr[0], arr[i]);
    sift_down(arr, 0, i - 1);
  }
}
```

```
def sift_down(arr, start, end):
    # 计算父结点和子结点的下标
    parent = int(start)
    child = int(parent * 2 + 1)
    while child <= end: # 子结点下标在范围内才做比较
        # 先比较两个子结点大小，选择最大的
        if child + 1 <= end and arr[child] < arr[child + 1]:
            child += 1
        # 如果父结点比子结点大，代表调整完毕，直接跳出函数
        if arr[parent] >= arr[child]:
            return
        else: # 否则交换父子内容，子结点再和孙结点比较
            arr[parent], arr[child] = arr[child], arr[parent]
            parent = child
            child = int(parent * 2 + 1)

def heap_sort(arr, len):
  # 从最后一个节点的父节点开始 sift down 以完成堆化 (heapify)
    i = (len - 1 - 1) / 2
    while(i >= 0):
        sift_down(arr, i, len - 1)
        i -= 1
  # 先将第一个元素和已经排好的元素前一位做交换，再重新调整（刚调整的元素之前的元素
), 直到排序完毕
    i = len - 1
    while(i > 0):
        arr[0], arr[i] = arr[i], arr[0]
        sift_down(arr, 0, i - 1)
        i -= 1
```

外部链接

- 堆排序 - 维基百科， 自由的百科全书[1]

参考资料与注释

[1] 堆排序 - 维基百科，自由的百科全书

5.7.10 桶排序

本页面将简要介绍桶排序。

定义

桶排序（英文：Bucket sort）是排序算法的一种，适用于待排序数据值域较大但分布比较均匀的情况。

过程

桶排序按下列步骤进行：

1. 设置一个定量的数组当作空桶；
2. 遍历序列，并将元素一个个放到对应的桶中；
3. 对每个不是空的桶进行排序；
4. 从不是空的桶里把元素再放回原来的序列中。

性质

稳定性

如果使用稳定的内层排序，并且将元素插入桶中时不改变元素间的相对顺序，那么桶排序就是一种稳定的排序算法。

由于每块元素不多，一般使用插入排序。此时桶排序是一种稳定的排序算法。

时间复杂度

桶排序的平均时间复杂度为 $O(n + n^2/k + k)$（将值域平均分成 n 块 + 排序 + 重新合并元素），当 $k \approx n$ 时为 $O(n)$。[1]

桶排序的最坏时间复杂度为 $O(n^2)$。

实现

```
const int N = 100010;

int n, w, a[N];
vector<int> bucket[N];

void insertion_sort(vector<int>& A) {
  for (int i = 1; i < A.size(); ++i) {
    int key = A[i];
    int j = i - 1;
    while (j >= 0 && A[j] > key) {
      A[j + 1] = A[j];
      --j;
    }
    A[j + 1] = key;
  }
}

void bucket_sort() {
  int bucket_size = w / n + 1;
```

```
  for (int i = 0; i < n; ++i) {
    bucket[i].clear();
  }
  for (int i = 1; i <= n; ++i) {
    bucket[a[i] / bucket_size].push_back(a[i]);
  }
  int p = 0;
  for (int i = 0; i < n; ++i) {
    insertion_sort(bucket[i]);
    for (int j = 0; j < bucket[i].size(); ++j) {
      a[++p] = bucket[i][j];
    }
  }
}
```

```python
N = 100010
w = n = 0
a = [0] * N
bucket = [[] for i in range(N)]

def insertion_sort(A):
    for i in range(1, len(A)):
        key = A[i]
        j = i - 1
        while j >= 0 and A[j] > key:
            A[j + 1] = A[j]
            j -= 1
        A[j + 1] = key

def bucket_sort():
    bucket_size = int(w / n + 1)
    for i in range(0, n):
        bucket[i].clear()
    for i in range(1, n + 1):
        bucket[int(a[i] / bucket_size)].append(a[i])
    p = 0
    for i in range(0, n):
        insertion_sort(bucket[i])
        for j in range(0, len(bucket[i])):
            a[p] = bucket[i][j]
            p += 1
```

参考资料与注释

[1] （英文）Bucket sort - Wikipedia

5.7.11 希尔排序

本页面将简要介绍希尔排序。

定义

希尔排序（英语：Shell sort），也称为缩小增量排序法，是 插入排序 的一种改进版本。希尔排序以它的发明者希尔（英语：Donald Shell）命名。

过程

排序对不相邻的记录进行比较和移动：

1. 将待排序序列分为若干子序列（每个子序列的元素在原始数组中间距相同）；
2. 对这些子序列进行插入排序；
3. 减小每个子序列中元素之间的间距，重复上述过程直至间距减少为 1。

性质

稳定性

希尔排序是一种不稳定的排序算法。

时间复杂度

希尔排序的最优时间复杂度为 $O(n)$。

希尔排序的平均时间复杂度和最坏时间复杂度与间距序列的选取有关。设间距序列为 H，下面给出 H 的两种经典选取方式，这两种选取方式均使得排序算法的复杂度降为 $o(n^2)$ 级别。

命题 1：若间距序列为 $H=\{2^k-1\mid k=1,2,\dots,\lfloor\log_2 n\rfloor\}$（从大到小），则希尔排序算法的时间复杂度为 $O(n^{3/2})$。

命题 2：若间距序列为 $H=\{k=2^p\cdot 3^q\mid p,q\in\mathbb{N},k\le n\}$（从大到小），则希尔排序算法的时间复杂度为 $O(n\log^2 n)$。

为证明这两个命题，我们先给出一个重要的定理并证明它，这个定理反应了希尔排序的最主要特征。

定理 1：只要程序执行了一次 InsertionSort(h)，不管之后怎样调用 InsertionSort 函数，A 数组怎样变换，下列性质均会被一直保持：

$$
\begin{gathered}
A_1,A_{1+h},A_{1+2h},\dots\\
A_2,A_{2+h},A_{2+2h},\dots\\
\vdots\\
A_h,A_{h+h},A_{h+2h},\dots
\end{gathered}
$$

证明：

我们先证明一个引理：

引理 1：对于整数 n,m、正整数 l 与两个数组 $X(x_1,x_2,\dots,x_{n+l}),Y(y_1,y_2,\dots,y_{m+l})$，满足如下要求：

$$y_1\le x_{n+1},y_2\le x_{n+2},\dots,y_l\le x_{n+l}$$

则我们将两个数组分别升序排序后，上述要求依然成立。

证明：

设数组 X 排序完为数组 $X'(x_1',\dots,x_{n+l}')$，数组 Y 排序完为数组 $Y'(y_1',\dots,y_{m+l}')$。

对于任何 $1\le i\le l$，x_{n+i}' 小等于数组 X' 中的 $l-i$ 个元素，也小等于数组 X 中的 $l-i$ 个元素（这是因为 X 与 X' 的元素可重集合是相同的）。

那么在可重集合 $\{x_{n+1},\dots,x_{n+l}\}\subset X$ 中，大等于 x_{n+i}' 的元素个数不超过 $l-i$ 个。

进而小于 x_{n+i}' 的元素个数至少有 i 个，取出其中的 i 个，设它们为 $x_{n+k_1},x_{n+k_2},\dots,x_{n+k_i}$。于是有：

$$y_{k_1}\le x_{n+k_1}\le x_{n+i}',y_{k_2}\le x_{n+k_2}\le x_{n+i}',\dots,y_{k_i}\le x_{n+k_i}\le x_{n+i}'$$

> 所以 x'_{n+i} 至少大等于 Y 也即 Y' 中的 i 个元素，那么自然有 $y'_i \le x'_{n+i}\,(1 \le i \le l)$。
>
> **证毕**

再回到原命题的证明：

我们实际上只需要证明调用完 InsertionSort(h) 的紧接着下一次调用 InsertionSort(k) 后，h 个子列仍有序即可，之后容易用归纳法得出。下面只考虑下一个调用：

执行完 InsertionSort(h) 后，如下组已经完成排序：

$$\begin{gathered} A_1, A_{1+h}, A_{1+2h}, \dots \\ A_2, A_{2+h}, A_{2+2h}, \dots \\ \vdots \\ A_h, A_{h+h}, A_{h+2h}, \dots \end{gathered}$$

而之后执行 InsertionSort(k)，则会将如下组排序：

$$\begin{gathered} A_1, A_{1+k}, A_{1+2k}, \dots \\ A_2, A_{2+k}, A_{2+2k}, \dots \\ \vdots \\ A_k, A_{k+k}, A_{k+2k}, \dots \end{gathered}$$

对于每个 $i\ (1 \le i \le \min(h,k))$，考虑如下两个组：

$$\begin{gathered} A_i, A_{i+k}, A_{i+2k}, \dots \\ \dots, A_{i+h}, A_{i+h+k}, A_{i+h+2k}, \dots \end{gathered}$$

第二个组前面也加上“...”的原因是可能 $i+h \ge k$ 从而前面也有元素。

则第二个组就是引理 1 中的 X 数组，第一个组就是 Y 数组，l 就是第二个组从 $i+h$ 之后顶到末尾的长度，n 是第二个组中前面那个“...”的长度，m 是第一个组去掉前 l 个后剩下的个数。

又因为有：

$$A_i \le A_{i+h}, A_{i+k} \le A_{i+h+k}, \dots$$

所以由引理 1 可得执行 InsertionSort(k) 将两个组分别排序后，这个关系依然满足，即依然有 $A_i \le A_{i+h}\,(1 \le i \le \min(h,k))$。

若有 $i > \min(h,k)$，容易发现取正整数 $w\ (1 \le w \le \min(h,k))$ 再加上若干个 k 即可得到 i，则之前的情况已经蕴含了此情况的证明。

综合以上论述便有：执行完 InsertionSort(k) 依然有 $A_i \le A_{i+h}\,(1 \le i \le n-h)$。

得证。

证毕

这个定理揭示了希尔排序在特定集合 H 下可以优化复杂度的关键，因为在整个过程中，它可以一致保持前面的成果不被摧毁（即 h 个子列分别有序），从而使后面的调用中，指针 i 的移动次数大大减少。

接下来我们单拎出来一个数论引理进行证明。这个定理在 OI 界因 小凯的疑惑[2] 一题而大为出名。而在希尔排序复杂度的证明中，它也使得定理 1 得到了很大的扩展。

> **引理** 2：若 a,b 均为正整数且互素，则不在集合 $\{ax+by \mid x,y \in \mathbb{N}\}$ 中的最大正整数为 $ab-a-b$。
>
> **证明**：
>
> 分两步证明：
>
> - 先证明方程 $ax+by=ab-a-b$ 没有 x,y 均为非负整数的解：
>
>   若无非负整数的限制，容易得到两组解 $(b-1,-1),(-1,a-1)$。
>
>   通过其通解形式 $x=x_0+tb, y=y_0-ta$，容易得到上面两组解是“相邻”的（因为 $b-1-b=-1$）。
>
>   当 t 递增时，x 递增，y 递减，所以如果方程有非负整数解，必然会夹在这两组解中间，但这两组解“相邻”，中间没有别的解。

故不可能有非负整数解。

- 再证明对任意整数 $c > ab - a - b$，方程 $ax + by = c$ 有非负整数解：

  我们找一组解 (x_0, y_0) 满足 $0 \le x_0 < b$（由通解的表达式，这可以做到）。

  则有：

$$by_0 = c - ax_0 \ge c - a(b-1) > ab - a - b - ab + a = -b$$

  所以 $b(y_0 + 1) > 0$，又因为 $b > 0$，所以 $y_0 + 1 > 0$，所以 $y_0 \ge 0$。

  所以 (x_0, y_0) 为一组非负整数解。

综上得证。

证毕

而下面这个定理则揭示了引理 2 是如何扩展定理 1 的。

定理 2：如果 $\gcd(h_{t+1}, h_t) = 1$，则程序先执行完 InsertionSort(h_{t+1}) 与 InsertionSort(h_t) 后，执行 InsertionSort(h_{t-1}) 的时间复杂度为 $O\left(\dfrac{nh_{t+1}h_t}{h_{t-1}}\right)$，且对于每个 j，其 i 的移动次数是 $O\left(\dfrac{h_{t+1}h_t}{h_{t-1}}\right)$ 级别的。

证明：

对于 $j \le h_{t+1}h_t$ 的部分，i 的移动次数显然是是 $O\left(\dfrac{h_{t+1}h_t}{h_{t-1}}\right)$ 级别的。

故以下假设 $j > h_{t+1}h_t$。

对于任意的正整数 k 满足 $1 \le k \le j - h_{t+1}h_t$，注意到：$h_{t+1}h_t - h_{t+1} - h_t < h_{t+1}h_t \le j - k \le j - 1$

又因为 $\gcd(h_{t+1}, h_t) = 1$，故由引理 2，得存在非负整数 a, b，使得：$ah_{t+1} + bh_t = j - k$。

即得：

$$k = j - ah_{t+1} - bh_t$$

由定理 1，得：

$$A_{j-bh_t} \le A_{j-(b-1)h_t} \le \ldots \le A_{j-h_t} \le A_j$$

与

$$A_{j-bh_t-ah_{t+1}} \le A_{j-bh_t-(a-1)h_{t+1}} \le \ldots \le A_{j-bh_t-h_{t+1}} \le A_{j-bh_t}$$

综合以上既有：$A_k = A_{j-ah_{t+1}-bh_t} \le A_j$。

所以对于任何 $1 \le k \le j - h_{t+1}h_t$，有 $A_k \le A_j$。

在 Shell-Sort 伪代码中 i 指针每次减 h_{t-1}，减 $O\left(\dfrac{h_{t+1}h_t}{h_{t-1}}\right)$ 次，即可使得 $i \le j - h_{t+1}h_t$，进而有 $A_i \le A_j$，不满足 while 循环的条件退出。

证明完对于每个 j 的移动复杂度后，即可得到总的时间复杂度：

$$\sum_{j=h_{t-1}+1}^{n} O\left(\frac{h_{t+1}h_t}{h_{t-1}}\right) = O\left(\frac{nh_{t+1}h_t}{h_{t-1}}\right)$$

得证。

证毕

认真观察定理 2 的证明过程，可以发现：定理 1 可以进行“线性组合”，即 A 以 h 为间隔有序，以 k 为间隔亦有序，则以 h 和 k 的非负系数线性组合仍是有序的。而这种“线性性”即是由引理 2 保证的。

有了这两个定理，我们可以命题 1 与 2。

先证明命题 1：

证明：

将 H 写为序列的形式：

$$H(h_1 = 1, h_2 = 3, h_3 = 7, \ldots, h_{\lfloor \log_2 n \rfloor} = 2^{\lfloor \log_2 n \rfloor} - 1)$$

Shell-Sort 执行顺序为：InsertionSort$(h_{\lfloor \log_2 n \rfloor})$, InsertionSort$(h_{\lfloor \log_2 n \rfloor - 1})$, $\ldots$, InsertionSort(h_2), InsertionSort(h_1).

分两部分去分析复杂度：

- 对于前面的若干个满足 $h_t \geq \sqrt{n}$ 的 h_t，显然有 InsertionSort(h_t) 的时间复杂度为 $O\left(\dfrac{n^2}{h_t}\right)$。

  考虑对最接近 $\sqrt{n}$ 的项 h_k，有：

$$O\left(\frac{n^2}{h_t}\right) = O(n^{3/2})$$

  而对于 $i > k$ 的 h_i，因为有 $2h_i < h_{i+1}$，所以可得：

$$O\left(\frac{n^2}{h_i}\right) = O(n^{3/2}/2^{i-k})\,(i > k)$$

  所以大等于 $\sqrt{n}$ 部分的总时间复杂度为：

$$\sum_{i=k}^{\lfloor \log_2 n \rfloor} O(n^{3/2}/2^{i-k}) = O(n^{3/2})$$

- 对于后面剩下的满足 $h_t < \sqrt{n}$ 的项，前两项的复杂度还是 $O(n^{3/2})$，而对于后面的项 h_t，有定理 2 可得时间复杂度为：

$$O\left(\frac{nh_{t+2}h_{t+1}}{h_t}\right) = O\left(\frac{nh_{t+2}\cdot h_{t+2}/2}{h_{t+2}/4}\right) = O(nh_{t+2})$$

  再次利用 $2h_i < h_{i+1}$ 性质可得此部分总时间复杂度为（下式中 k 沿用了上一种情况中的含义）：

$$2O(n^{3/2}) + \sum_{i=1}^{k-3} O(nh_{i+1}) = O(n^{3/2}) + \sum_{i=1}^{k-3} O(nh_{k-1}/2^{k-i-3}) = O(n^{3/2}) + O(nh_{k-1}) = O(n^{3/2})$$

综上可得总时间复杂度即为 $O(n^{3/2})$。

证毕

再证明命题 2：

证明：

注意到一个事实：如果已经执行过了 InsertionSort(2) 与 InsertionSort(3)，那么因为 $2 \cdot 3 - 2 - 3 = 1$，所以由定理 2，每个元素只有与它相邻的前一个元素可能大于它，之前的元素全部都小于它。于是 i 指针只需要最多两次就可以退出 while 循环。也就是说，此时再执行 InsertionSort(1)，复杂度降为 $O(n)$。

更进一步：如果已经执行过了 InsertionSort(4) 与 InsertionSort(6)，我们考虑所有的下标为奇数的元素组成的子列与下标为偶数的元素组成的子列。则这相当于把这两个子列分别执行 InsertionSort(2) 与 InsertionSort(3)。那么也是一样，这时候再执行 InsertionSort(2)，相当于对两个子列分别执行 InsertionSort(1)，也只需要两个序列和的级别，即 $O(n)$ 的复杂度就可以将数组变为 2 间隔有序。

不断归纳，就可以得到：如果已经执行过了 InsertionSort($2h$) 与 InsertionSort($3h$)，则执行 InsertionSort(h) 的复杂度也只有 $O(n)$。

接下来分为两部分分析复杂度：

- 对于 $h_t > n/3$ 的部分，则执行每个 InsertionSort(h_t) 的复杂度为 $O(n^2/h_t)$。

  而 $n^2/h_t < 3n$，所以单词插入排序复杂度为 $O(n)$。

  而这一部分元素个数是 $O(\log^2 n)$ 级别的，所以这一部分时间复杂度为 $O(n \log^2 n)$。

- 对于 $h_t \leq n/3$ 的部分，因为 $3h_t \leq n$，所以这之前已经执行了 InsertionSort($2h_t$) 与 InsertionSort($3h_t$)，于是执行 InsertionSort(h_t) 的时间复杂度是 $O(n)$。

  还是一样的，这一部分元素个数也是 $O(\log^2 n)$ 级别的，所以这一部分时间复杂度为 $O(n \log^2 n)$。

综上可得总时间复杂度即为 $O(n \log^2 n)$。

证毕

空间复杂度

希尔排序的空间复杂度为 $O(1)$。

实现

```cpp
template <typename T>
void shell_sort(T array[], int length) {
  int h = 1;
  while (h < length / 3) {
    h = 3 * h + 1;
  }
  while (h >= 1) {
    for (int i = h; i < length; i++) {
      for (int j = i; j >= h && array[j] < array[j - h]; j -= h) {
        std::swap(array[j], array[j - h]);
      }
    }
    h = h / 3;
  }
}
```

```python
def shell_sort(array, length):
    h = 1
    while h < length / 3:
        h = int(3 * h + 1)
    while h >= 1:
        for i in range(h, length):
            j = i
            while j >= h and array[j] < array[j - h]:
                array[j], array[j - h] = array[j - h], array[j]
                j -= h
        h = int(h / 3)
```

参考资料与注释

[1] 希尔排序 - 维基百科， 自由的百科全书

[2] 小凯的疑惑

5.7.12 锦标赛排序

本页面将简要介绍锦标赛排序。

定义

锦标赛排序（英文：Tournament sort），又被称为树形选择排序，是 选择排序 的优化版本，堆排序 的一种变体（均采用完全二叉树）。它在选择排序的基础上使用优先队列查找下一个该选择的元素。

引入

锦标赛排序的名字来源于单败淘汰制的竞赛形式。在这种赛制中有许多选手参与比赛，他们两两比较，胜者进入下一轮比赛。这种淘汰方式能够决定最好的选手，但是在最后一轮比赛中被淘汰的选手不一定是第二好的——他可能不如先前被淘汰的选手。

过程

以**最小锦标赛排序树**为例：

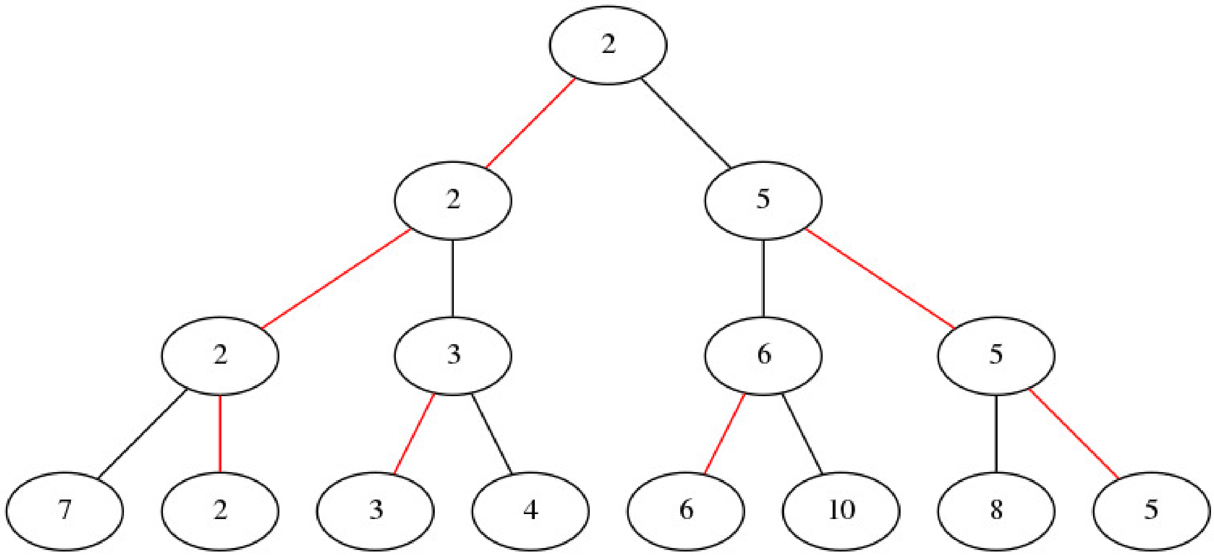


图 5.10 tournament-sort1

待排序元素是叶子节点显示的元素。红色边显示的是每一轮比较中较小的元素的胜出路径。显然，完成一次 " 锦标赛 " 可以选出一组元素中最小的那一个。

每一轮对 n 个元素进行比较后可以得到 $\frac{n}{2}$ 个「优胜者」，每一对中较小的元素进入下一轮比较。如果无法凑齐一对元素，那么这个元素直接进入下一轮的比较。

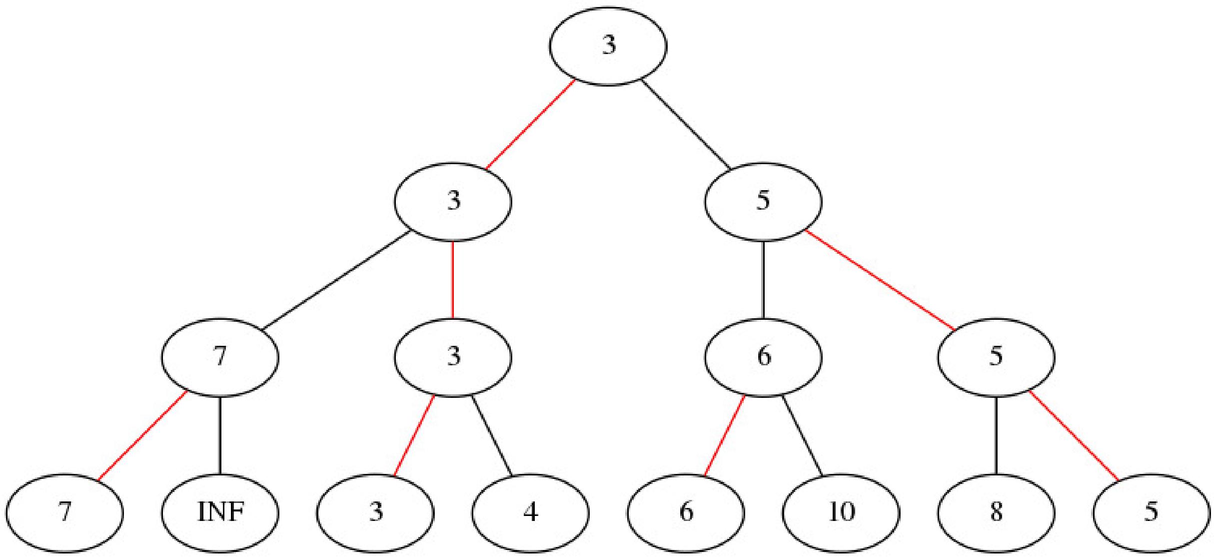


图 5.11 tournament-sort2

完成一次「锦标赛」后需要将被选出的元素去除。直接将其设置为 ∞（这个操作类似 堆排序），然后再次举行「锦标赛」选出次小的元素。

之后一直重复这个操作，直至所有元素有序。

性质

稳定性

锦标赛排序是一种不稳定的排序算法。

时间复杂度

锦标赛排序的最优时间复杂度、平均时间复杂度和最坏时间复杂度均为 $O(n\log n)$。它用 $O(n)$ 的时间初始化「锦标赛」，然后用 $O(\log n)$ 的时间从 n 个元素中选取一个元素。

空间复杂度

锦标赛排序的空间复杂度为 $O(n)$。

实现

```
int n, a[maxn], tmp[maxn << 1];

int winner(int pos1, int pos2) {
  int u = pos1 >= n ? pos1 : tmp[pos1];
  int v = pos2 >= n ? pos2 : tmp[pos2];
  if (tmp[u] <= tmp[v]) return u;
  return v;
}

void creat_tree(int &value) {
  for (int i = 0; i < n; i++) tmp[n + i] = a[i];
  for (int i = 2 * n - 1; i > 1; i -= 2) {
    int k = i / 2;
    int j = i - 1;
    tmp[k] = winner(i, j);
  }
  value = tmp[tmp[1]];
  tmp[tmp[1]] = INF;
}

void recreat(int &value) {
  int i = tmp[1];
  while (i > 1) {
    int j, k = i / 2;
    if (i % 2 == 0 && i < 2 * n - 1)
      j = i + 1;
    else
      j = i - 1;
    tmp[k] = winner(i, j);
    i = k;
  }
  value = tmp[tmp[1]];
  tmp[tmp[1]] = INF;
}
```

```cpp
void tournament_sort() {
  int value;
  creat_tree(value);
  for (int i = 0; i < n; i++) {
    a[i] = value;
    recreat(value);
  }
}
```

```python
n = 0
a = [0] * maxn
tmp = [0] * maxn * 2

def winner(pos1, pos2):
    u = pos1 if pos1 >= n else tmp[pos1]
    v = pos2 if pos2 >= n else tmp[pos2]
    if tmp[u] <= tmp[v]:
        return u
    return v

def creat_tree():
    for i in range(0, n):
        tmp[n + i] = a[i]
    for i in range(2 * n -1, 1, -2):
        k = int(i / 2)
        j = i - 1
        tmp[k] = winner(i, j)
    value = tmp[tmp[1]]
    tmp[tmp[1]] = INF
    return value

def recreat():
    i = tmp[1]
    while i > 1:
        j = k = int(i / 2)
        if i % 2 == 0 and i < 2 * n - 1:
            j = i + 1
        else:
            j = i - 1
        tmp[k] = winner(i, j)
        i = k
    value = tmp[tmp[1]]
    tmp[tmp[1]] = INF
    return value

def tournament_sort():
    value = creat_tree()
    for i in range(0, n):
        a[i] = value
        value = recreat()
```

外部链接

- Tournament sort - Wikipedia[1]

参考资料与注释

[1] Tournament sort - Wikipedia

5.7.13 tim 排序

Author:Backl1ght

tim 排序是归并排序和插入排序的结合，是一个**稳定**的排序算法，由 Tim Peters 于 2002 年用 Python 实现。现在，tim 排序是 Python 的标准排序算法，且被 Java SE7 用于对非原始类型的数组排序。

tim 排序在最好情况下的时间复杂度为 $O(n)$，最差情况下的时间复杂度为 $O(n\log n)$，期望时间复杂度为 $O(n\log n)$。tim 排序在最坏情况下的空间复杂度为 $O(n)$。

算法

众所周知，归并排序是先将数组划分为两部分，然后递归地对两个子数组进行归并排序，最后合并两个子数组。这样一来，归并排序合并操作的最小单位就是单个元素。但是，数组中可能原本就存在连续且有序的子数组，归并排序无法利用这个特性。

tim 排序为了利用数组中本身就存在的连续且有序的子数组，以 RUN 作为合并操作的最小单位。其中，RUN 是一个满足以下性质的子数组：

- 一个 RUN 要么是非降序的，要么是严格升序的。
- 一个 RUN 存在一个长度的下限。

tim 排序的过程就是一个类似归并排序的过程，将数组划分为多个 RUN，然后以某种规则不断地合并两个 RUN，直到数组有序。具体过程如下：

令 $nRemaining$ 初始化为数组的大小，$minRun$ 初始化为 $getMinRunLength(nRemaining)$。

do
  确定run的起点
  ifrun比$minRun$短
    延长run直至 $\min(minRun, nRemaining)$
  pushrun放到$pending-runstack$上
  if$pending-runstack$最顶上的 2 个run长度相近
    合并$pending-runstack$最顶上的 2 个run
  $\text{start index} \leftarrow \text{start index} + run$的长度
  $nRemaining \leftarrow nRemaining - run$的长度
while $nRemaining \neq 0$

其中，$getMinRunLength$ 函数是根据当前数组长度确定 $minRun$ 具体值的函数，natural run 的意思是原本就非降序或者严格升序的 run，扩展长度不够的 run 就是用插入排序往 run 中添加元素。

复杂度证明

最好情况下，数组本身就有序，即数组本身就是一个 RUN，这个时候 tim 排序就遍历了一遍数组，找到了唯一的 RUN，就结束了。所以，最好的情况下，tim 排序的时间复杂度为 $O(n)$。

写在后面

本文只是简单介绍了 tim 排序的原理，实际上 tim 排序在实现的时候还有一些其他的优化，这里不再一一列举。tim 排序在 java 中的实现写得非常好，要想知道真正的 tim 排序推荐去看 java 中 tim 排序的实现。

参考资料

1. Timsort[1]
2. On the Worst-Case Complexity of TimSort[2]
3. original explanation by Tim Peters[3]
4. java 实现[4]
5. c 语言实现[5]

参考资料与注释

[1] Timsort

[2] On the Worst-Case Complexity of TimSort

[3] original explanation by Tim Peters

[4] java 实现

[5] c 语言实现

5.7.14 排序相关 STL

本页面将简要介绍 C 和 C++ 标准库中实现的排序算法。

除已说明的函数外，本页所列函数默认定义于头文件 `<algorithm>` 中。

qsort

参见：`qsort`[2]，`std::qsort`[3]

该函数为 C 标准库实现的 快速排序，定义在 `<stdlib.h>` 中。在 C++ 标准库里，该函数定义在 `<cstdlib>` 中。

qsort 与 bsearch 的比较函数

qsort 函数有四个参数：数组名、元素个数、元素大小、比较规则。其中，比较规则通过指定比较函数来实现，指定不同的比较函数可以实现不同的排序规则。

比较函数的参数限定为两个 const void 类型的指针。返回值规定为正数、负数和 0。

比较函数的一种示例写法为：

```
int compare(const void *p1, const void *p2)  // int 类型数组的比较函数
{
  int *a = (int *)p1;
  int *b = (int *)p2;
  if (*a > *b)
    return 1;  // 返回正数表示 a 大于 b
  else if (*a < *b)
    return -1;  // 返回负数表示 a 小于 b
  else
    return 0;  // 返回 0 表示 a 与 b 等价
}
```

注意：返回值用两个元素相减代替正负数是一种典型的错误写法，因为这样可能会导致溢出错误。

以下是排序结构体的一个示例：

```
struct eg  // 示例结构体
{
  int e;
  int g;
};

int compare(const void *p1,
            const void *p2)  // struct eg 类型数组的比较函数：按成员 e 排序
{
  struct eg *a = (struct eg *)p1;
  struct eg *b = (struct eg *)p2;
  if (a->e > b->e)
    return 1;  // 返回正数表示 a 大于 b
  else if (a->e < b->e)
    return -1;  // 返回负数表示 a 小于 b
  else
    return 0;  // 返回 0 表示 a 与 b 等价
}
```

这里也可以看出，等价不代表相等，只代表在此比较规则下两元素等价。

std::sort

参见：`std::sort`[4]

用法：

```
// a[0] .. a[n - 1] 为需要排序的数列
// 对 a 原地排序，将其按从小到大的顺序排列
std::sort(a, a + n);

// cmp 为自定义的比较函数
std::sort(a, a + n, cmp);
```

注意：sort 的比较函数的返回值是 true 和 false，用 true 和 false 表示两个元素的大小（先后）关系，这与 qsort 的三值比较函数的语义完全不同。具体内容详见上方给出的 sort 的文档。

如果要将 sort 简单改写为 qsort，维持排序顺序整体上不变（不考虑等价的元素），需要将返回 true 改为 - 1，返回 false 改为 1。

`std::sort` 函数是更常用的 C++ 库比较函数。该函数的最后一个参数为二元比较函数，未指定 `cmp` 函数时，默认按从小到大的顺序排序。

旧版 C++ 标准中仅要求它的**平均**时间复杂度达到 $O(n \log n)$。C++11 标准以及后续标准要求它的**最坏**时间复杂度达到 $O(n \log n)$。

C++ 标准并未严格要求此函数的实现算法，具体实现取决于编译器。libstdc++[5] 和 libc++[6] 中的实现都使用了 内省排序。

std::nth_element

参见：`std::nth_element`[7]

用法：

```
std::nth_element(first, nth, last);
std::nth_element(first, nth, last, cmp);
```

它重排 [first, last) 中的元素，使得 nth 所指向的元素被更改为 [first, last) 排好序后该位置会出现的元素。这个新的 nth 元素前的所有元素小于或等于新的 nth 元素后的所有元素。

实现算法是未完成的内省排序。

对于以上两种用法，C++ 标准要求它的平均时间复杂度为 $O(n)$，其中 n 为 std::distance(first, last)。

它常用于构建 K-D Tree。

std::stable_sort

参见：std::stable_sort[8]

用法：

```
std::stable_sort(first, last);
std::stable_sort(first, last, cmp);
```

稳定排序，保证相等元素排序后的相对位置与原序列相同。

时间复杂度为 $O(n\log(n)^2)$，当额外内存可用时，复杂度为 $O(n\log n)$。

std::partial_sort

参见：std::partial_sort[9]

用法：

```
// mid = first + k
std::partial_sort(first, mid, last);
std::partial_sort(first, mid, last, cmp);
```

将序列中前 k 元素按 cmp 给定的顺序进行原地排序，后面的元素不保证顺序。未指定 cmp 函数时，默认按从小到大的顺序排序。

复杂度：约 $(last - first)\log(mid - first)$ 次应用 cmp。

原理：

std::partial_sort 的思想是：对原始容器内区间为 [first, mid) 的元素执行 make_heap() 操作，构造一个大根堆，然后将 [mid, last) 中的每个元素和 first 进行比较，保证 first 内的元素为堆内的最大值。如果小于该最大值，则互换元素位置，并对 [first, mid) 内的元素进行调整，使其保持最大堆序。比较完之后，再对 [first, mid) 内的元素做一次堆排序 sort_heap() 操作，使其按增序排列。注意，堆序和增序是不同的。

自定义比较

参见：运算符重载[10]

内置类型（如 int）和用户定义的结构体允许定制调用 STL 排序函数时使用的比较函数。可以在调用该函数时，在最后一个参数中传入一个实现二元比较的函数。

对于用户定义的结构体，对其使用 STL 排序函数前必须定义至少一种关系运算符，或是在使用函数时提供二元比较函数。通常推荐定义 operator<。[1]

示例：

```
int a[1009], n = 10;
// ...
```

```cpp
std::sort(a + 1, a + 1 + n);                // 从小到大排序
std::sort(a + 1, a + 1 + n, greater<int>()); // 从大到小排序
```

```cpp
struct data {
  int a, b;

  bool operator<(const data rhs) const {
    return (a == rhs.a) ? (b < rhs.b) : (a < rhs.a);
  }
} da[1009];

bool cmp(const data u1, const data u2) {
  return (u1.a == u2.a) ? (u1.b > u2.b) : (u1.a > u2.a);
}

// ...
std::sort(da + 1, da + 1 + 10);  // 使用结构体中定义的 < 运算符，从小到大排序
std::sort(da + 1, da + 1 + 10, cmp);  // 使用 cmp 函数进行比较，从大到小排序
```

严格弱序

另请参阅：C++ 中的应用 - 序理论

进行排序的运算符必须满足 严格弱序，否则会出现不可预料的情况（如运行时错误、无法正确排序）。

常见的错误做法：

- 使用 `<=` 来定义排序中的小于运算符。
- 在调用排序运算符时，读取外部数值可能会改变的数组（常见于最短路算法）。
- 将多个数的最大最小值进行比较的结果作为排序运算符（如皇后游戏/加工生产调度中的经典错误）。

外部链接

- 浅谈邻项交换排序的应用以及需要注意的问题[11]

参考资料与注释

[1] 因为大部分标准算法默认使用 `operator<` 进行比较。

[2] `qsort`

[3] `std::qsort`

[4] `std::sort`

[5] libstdc++

[6] libc++

[7] `std::nth_element`

[8] `std::stable_sort`

[9] std::partial_sort

[10] 运算符重载

[11] 浅谈邻项交换排序的应用以及需要注意的问题

5.7.15 排序应用

本页面将简要介绍排序的用法。

理解数据的特点

使用排序处理数据有利于理解数据的特点，方便我们之后的分析与视觉化。像一些生活中的例子比如词典，菜单，如果不是按照一定顺序排列的话，人们想要找到自己需要的东西的时间就会大大增加。

计算机需要处理大规模的数据，排序后，人们可以根据数据的特点和需求来设计计算机的后续处理流程。

降低时间复杂度

使用排序预处理可以降低求解问题所需要的时间复杂度，通常是一个以空间换取时间的平衡。如果一个排序好的列表需要被多次分析的话，只需要耗费一次排序所需要的资源是很划算的，因为之后的每次分析都可以减少很多时间。

" 示例：检查给定数列中是否有相等的元素"

考虑一个数列，你需要检查其中是否有元素相等。

一个朴素的做法是检查每一个数对，并判断这一对数是否相等。时间复杂度是 $O(n^2)$。

我们不妨先对这一列数排序，之后不难发现：如果有相等的两个数，它们一定在新数列中处于相邻的位置上。这时，只需要 $O(n)$ 地扫一遍新数列了。

总的时间复杂度是排序的复杂度 $O(n\log n)$。

作为查找的预处理

排序是 二分查找 所要做的预处理工作。在排序后使用二分查找，可以以 $O(\log n)$ 的时间在序列中查找指定的元素。

5.8 前缀和 & 差分

前缀和

定义

前缀和可以简单理解为「数列的前 n 项的和」，是一种重要的预处理方式，能大大降低查询的时间复杂度。[1]

C++ 标准库中实现了前缀和函数 std::partial_sum[2]，定义于头文件 <numeric> 中。

例题

例题

有 N 个的正整数放到数组 A 里，现在要求一个新的数组 B，新数组的第 i 个数 $B[i]$ 是原数组 A 第 0 到第

i 个数的和。

输入：

```
5
1 2 3 4 5
```

输出：

```
1 3 6 10 15
```

" 解题思路"

递推：`B[0] = A[0]`，对于 $i \geq 1$ 则 `B[i] = B[i-1] + A[i]`。

" 参考代码"

```cpp
#include <iostream>
using namespace std;

int N, A[10000], B[10000];

int main() {
  cin >> N;
  for (int i = 0; i < N; i++) {
    cin >> A[i];
  }

  // 前缀和数组的第一项和原数组的第一项是相等的。
  B[0] = A[0];

  for (int i = 1; i < N; i++) {
    // 前缀和数组的第 i 项 = 原数组的 0 到 i-1 项的和 + 原数组的第 i 项。
    B[i] = B[i - 1] + A[i];
  }

  for (int i = 0; i < N; i++) {
    cout << B[i] << " ";
  }

  return 0;
}
```

```python
from itertools import accumulate
input()
print(*accumulate(map(int, input().split())))
```

二维/多维前缀和

多维前缀和的普通求解方法几乎都是基于容斥原理。

" 示例：一维前缀和扩展到二维前缀和"

比如我们有这样一个矩阵 a，可以视为二维数组：

```
1 2 4 3
5 1 2 4
6 3 5 9
```

我们定义一个矩阵 sum 使得 $sum_{x,y} = \sum\limits_{i=1}^{x}\sum\limits_{j=1}^{y} a_{i,j}$，

那么这个矩阵长这样：

```
1  3  7  10
6  9  15 22
12 18 29 45
```

第一个问题就是递推求 sum 的过程，$sum_{i,j} = sum_{i-1,j} + sum_{i,j-1} - sum_{i-1,j-1} + a_{i,j}$。

因为同时加了 $sum_{i-1,j}$ 和 $sum_{i,j-1}$，故重复了 $sum_{i-1,j-1}$，减去。

第二个问题就是如何应用，譬如求 $(x_1, y_1) - (x_2, y_2)$ 子矩阵的和。

那么，根据类似的思考过程，易得答案为 $sum_{x_2,y_2} - sum_{x_1-1,y_2} - sum_{x_2,y_1-1} + sum_{x_1-1,y_1-1}$。

例题

"**洛谷 P1387 最大正方形**[3]"

在一个 $n \times m$ 的只包含 0 和 1 的矩阵里找出一个不包含 0 的最大正方形，输出边长。

" **参考代码**"

```cpp
#include <algorithm>
#include <iostream>
using namespace std;
int a[103][103];
int b[103][103];  // 前缀和数组，相当于上文的 sum[]

int main() {
  int n, m;
  cin >> n >> m;

  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
      cin >> a[i][j];
      b[i][j] =
          b[i][j - 1] + b[i - 1][j] - b[i - 1][j - 1] + a[i][j];  // 求前缀和
    }
  }

  int ans = 1;

  int l = 2;
  while (l <= min(n, m)) {  // 判断条件
    for (int i = l; i <= n; i++) {
      for (int j = l; j <= m; j++) {
        if (b[i][j] - b[i - l][j] - b[i][j - l] + b[i - l][j - l] == l * l) {
          ans = max(ans, l);  // 在这里统计答案
        }
      }
    }
```

```
    l++;
  }

  cout << ans << endl;
  return 0;
}
```

```python
n, m = map(int, input().split())
a = [list(map(int, input().split())) for _ in range(n)]
b = [a[0]] + [[i[0]] + [0] * (m - 1) for i in a[1:]]
ans = 0
for i in range(1, n):
    for j in range(1, m):
        if a[i][j]:
            b[i][j] = min(b[i-1][j], b[i][j-1], b[i-1][j-1]) + 1
            ans = max(ans, b[i][j])
print(ans)
```

基于 DP 计算高维前缀和

基于容斥原理来计算高维前缀和的方法，其优点在于形式较为简单，无需特别记忆，但当维数升高时，其复杂度较高。这里介绍一种基于 DP 计算高维前缀和的方法。该方法即通常语境中所称的**高维前缀和**。

设高维空间 U 共有 D 维，需要对 $f[\cdot]$ 求高维前缀和 $\mathrm{sum}[\cdot]$。令 $\mathrm{sum}[i][\mathrm{state}]$ 表示同 state 后 $D-i$ 维相同的所有点对于 state 点高维前缀和的贡献。由定义可知 $\mathrm{sum}[0][\mathrm{state}] = f[\mathrm{state}]$，以及 $\mathrm{sum}[\mathrm{state}] = \mathrm{sum}[D][\mathrm{state}]$。

其递推关系为 $\mathrm{sum}[i][\mathrm{state}] = \mathrm{sum}[i-1][\mathrm{state}] + \mathrm{sum}[i][\mathrm{state}']$，其中 state' 为第 i 维恰好比 state 少 1 的点。该方法的复杂度为 $O(D \times |U|)$，其中 $|U|$ 为高维空间 U 的大小。

一种实现的伪代码如下：

$$
\begin{aligned}
&\textbf{for } state\\
&\quad sum[state] \leftarrow f[state]\\
&\textbf{for } i \leftarrow 0 \textbf{ to } D\\
&\quad \textbf{for } state' \textbf{ in } \text{lexicographical order}\\
&\quad\quad sum[state] \leftarrow sum[state] + sum[state']
\end{aligned}
$$

树上前缀和

设 sum_i 表示结点 i 到根节点的权值总和。

然后：

- 若是点权，x, y 路径上的和为 $sum_x + sum_y - sum_{lca} - sum_{fa_{lca}}$。
- 若是边权，x, y 路径上的和为 $sum_x + sum_y - 2 \cdot sum_{lca}$。

  LCA 的求法参见 最近公共祖先。

差分

解释

差分是一种和前缀和相对的策略，可以当做是求和的逆运算。

这种策略的定义是令 $b_i = \begin{cases} a_i - a_{i-1} & i \in [2, n] \\ a_1 & i = 1 \end{cases}$

性质

- a_i 的值是 b_i 的前缀和，即 $a_n = \sum\limits_{i=1}^{n} b_i$
- 计算 a_i 的前缀和 $sum = \sum\limits_{i=1}^{n} a_i = \sum\limits_{i=1}^{n}\sum\limits_{j=1}^{i} b_j = \sum\limits_{i}^{n}(n-i+1)b_i$

它可以维护多次对序列的一个区间加上一个数，并在最后询问某一位的数或是多次询问某一位的数。注意修改操作一定要在查询操作之前。

" 示例"

譬如使 $[l,r]$ 中的每个数加上一个 k，即

$$b_l \leftarrow b_l + k, b_{r+1} \leftarrow b_{r+1} - k$$

其中 $b_l + k = a_l + k - a_{l-1}$，$b_{r+1} - k = a_{r+1} - (a_r + k)$

最后做一遍前缀和就好了。

C++ 标准库中实现了差分函数 `std::adjacent_difference`[4]，定义于头文件 `<numeric>` 中。

树上差分

树上差分可以理解为对树上的某一段路径进行差分操作，这里的路径可以类比一维数组的区间进行理解。例如在对树上的一些路径进行频繁操作，并且询问某条边或者某个点在经过操作后的值的时候，就可以运用树上差分思想了。

树上差分通常会结合 树基础 和 最近公共祖先 来进行考察。树上差分又分为**点差分**与**边差分**，在实现上会稍有不同。

点差分

举例：对树上的一些路径 $\delta(s_1,t_1),\delta(s_2,t_2),\delta(s_3,t_3)\dots$ 进行访问，问一条路径 $\delta(s,t)$ 上的点被访问的次数。

对于一次 $\delta(s,t)$ 的访问，需要找到 s 与 t 的公共祖先，然后对这条路径上的点进行访问（点的权值加一），若采用 DFS 算法对每个点进行访问，由于有太多的路径需要访问，时间上承受不了。这里进行差分操作：

$$\begin{aligned}
&d_s \leftarrow d_s + 1\\
&d_{lca} \leftarrow d_{lca} - 1\\
&d_t \leftarrow d_t + 1\\
&d_{f(lca)} \leftarrow d_{f(lca)} - 1
\end{aligned}$$

其中 $f(x)$ 表示 x 的父亲节点，d_i 为点权 a_i 的差分数组。

可以认为公式中的前两条是对蓝色方框内的路径进行操作，后两条是对红色方框内的路径进行操作。不妨令 lca 左侧的直系子节点为 $left$。那么有 $d_{lca} - 1 = a_{lca} - (a_{left} + 1)$，$d_{f(lca)} - 1 = a_{f(lca)} - (a_{lca} + 1)$。可以发现实际上点差分的操作和上文一维数组的差分操作是类似的。

边差分

若是对路径中的边进行访问，就需要采用边差分策略了，使用以下公式：

$$\begin{aligned}
&d_s \leftarrow d_s + 1\\
&d_t \leftarrow d_t + 1\\
&d_{lca} \leftarrow d_{lca} - 2
\end{aligned}$$

由于在边上直接进行差分比较困难，所以将本来应当累加到红色边上的值向下移动到附近的点里，那么操作起来也就方便了。对于公式，有了点差分的理解基础后也不难推导，同样是对两段区间进行差分。

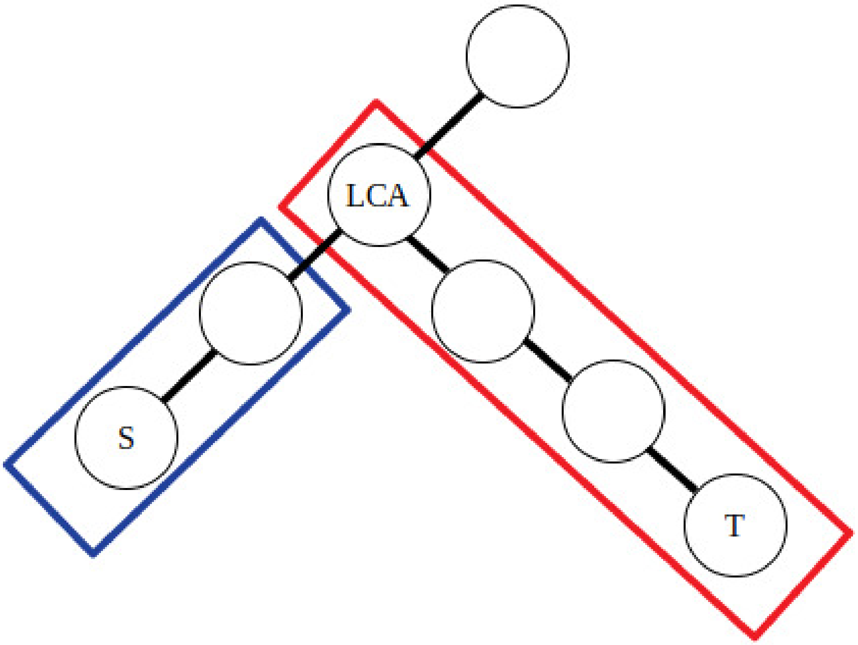


图 5.12

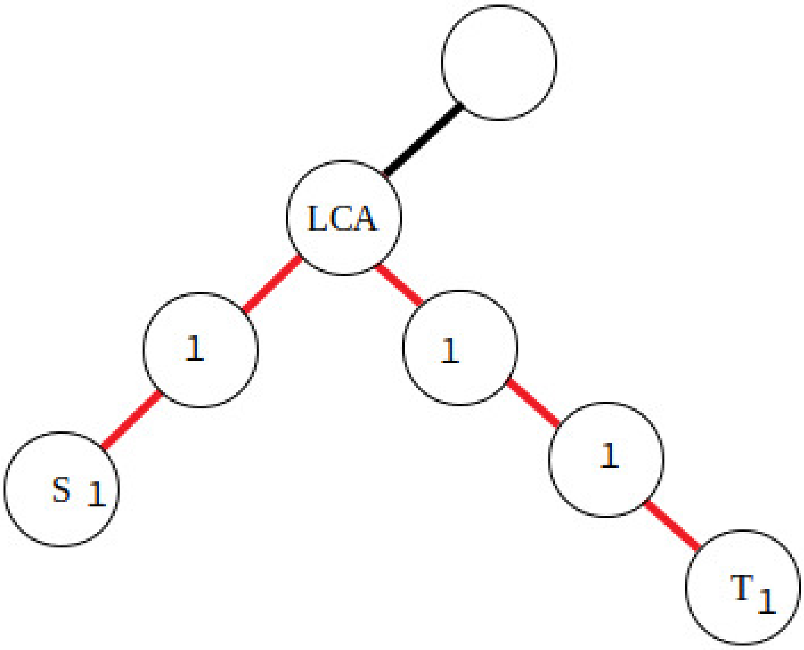


图 5.13

例题

"洛谷 3128 最大流[5]"

FJ 给他的牛棚的 $N(2 \leq N \leq 50,000)$ 个隔间之间安装了 $N-1$ 根管道，隔间编号从 1 到 N。所有隔间都被管道连通了。

FJ 有 $K(1 \leq K \leq 100,000)$ 条运输牛奶的路线，第 i 条路线从隔间 s_i 运输到隔间 t_i。一条运输路线会给它的两个端点处的隔间以及中间途径的所有隔间带来一个单位的运输压力，你需要计算压力最大的隔间的压力是多少。

" 解题思路"

需要统计每个点经过了多少次，那么就用树上差分将每一次的路径上的点加一，可以很快得到每个点经过的次数。这里采用倍增法计算 LCA，最后对 DFS 遍历整棵树，在回溯时对差分数组求和就能求得答案了。

" 参考代码"

```cpp
#include <bits/stdc++.h>

using namespace std;
#define maxn 50010

struct node {
  int to, next;
} edge[maxn << 1];

int fa[maxn][30], head[maxn << 1];
int power[maxn];
int depth[maxn], lg[maxn];
int n, k, ans = 0, tot = 0;

void add(int x, int y) {  // 加边
  edge[++tot].to = y;
  edge[tot].next = head[x];
  head[x] = tot;
}

void dfs(int now, int father) {  // dfs 求最大压力
  fa[now][0] = father;
  depth[now] = depth[father] + 1;
  for (int i = 1; i <= lg[depth[now]]; ++i)
    fa[now][i] = fa[fa[now][i - 1]][i - 1];
  for (int i = head[now]; i; i = edge[i].next)
    if (edge[i].to != father) dfs(edge[i].to, now);
}

int lca(int x, int y) {  // 求 LCA，最近公共祖先
  if (depth[x] < depth[y]) swap(x, y);
  while (depth[x] > depth[y]) x = fa[x][lg[depth[x] - depth[y]] - 1];
  if (x == y) return x;
  for (int k = lg[depth[x]] - 1; k >= 0; k--) {
    if (fa[x][k] != fa[y][k]) x = fa[x][k], y = fa[y][k];
  }
```

```
  return fa[x][0];
}

// 用 dfs 求最大压力，回溯时将子树的权值加上
void get_ans(int u, int father) {
  for (int i = head[u]; i; i = edge[i].next) {
    int to = edge[i].to;
    if (to == father) continue;
    get_ans(to, u);
    power[u] += power[to];
  }
  ans = max(ans, power[u]);
}

int main() {
  scanf("%d %d", &n, &k);
  int x, y;
  for (int i = 1; i <= n; i++) {
    lg[i] = lg[i - 1] + (1 << lg[i - 1] == i);
  }
  for (int i = 1; i <= n - 1; i++) {  // 建图
    scanf("%d %d", &x, &y);
    add(x, y);
    add(y, x);
  }
  dfs(1, 0);
  int s, t;
  for (int i = 1; i <= k; i++) {
    scanf("%d %d", &s, &t);
    int ancestor = lca(s, t);
    // 树上差分
    power[s]++;
    power[t]++;
    power[ancestor]--;
    power[fa[ancestor][0]]--;
  }
  get_ans(1, 0);
  printf("%d\n", ans);
  return 0;
}
```

习题

前缀和：

- 洛谷 B3612【深进 1. 例 1】求区间和[6]
- 洛谷 U69096 前缀和的逆[7]
- AT2412 最大 の 和[8]
- 「USACO16JAN」子共七 Subsequences Summing to Sevens[9]
- 「USACO05JAN」Moo Volume S[10]

二维/多维前缀和：

- HDU 6514 Monitor[11]

- 洛谷 P1387 最大正方形[3]
- 「HNOI2003」 激光炸弹[12]

基于 DP 计算高维前缀和：

- CF 165E Compatible Numbers[13]
- CF 383E Vowels[14]
- ARC 100C Or Plus Max[15]

树上前缀和：

- LOJ 10134.Dis[16]
- LOJ 2491. 求和[17]

差分：

- 树状数组 3： 区间修改 ， 区间查询[18]
- P3397 地毯[19]
- 「Poetize6」IncDec Sequence[20]

树上差分：

- 洛谷 3128 最大流[5]
- JLOI2014 松鼠的新家[21]
- NOIP2015 运输计划[22]
- NOIP2016 天天爱跑步[23]

参考资料与注释

[1] 南海区青少年信息学奥林匹克内部训练教材

[2] `std::partial_sum`

[3] 洛谷 P1387 最大正方形 [3-1] [3-2]

[4] `std::adjacent_difference`

[5] 洛谷 3128 最大流 [5-1] [5-2]

[6] 洛谷 B3612【深进 1. 例 1】求区间和

[7] 洛谷 U69096 前缀和的逆

[8] AT2412 最大の和

[9] 「USACO16JAN」子共七 Subsequences Summing to Sevens

[10] 「USACO05JAN」Moo Volume S

[11] HDU 6514 Monitor

[12] 「HNOI2003」激光炸弹

[13] CF 165E Compatible Numbers

[14] CF 383E Vowels

[15] ARC 100C Or Plus Max

[16] LOJ 10134.Dis

[17] LOJ 2491. 求和

[18] 树状数组 3：区间修改，区间查询

[19] P3397 地毯

[20] 「Poetize6」IncDec Sequence

[21] JLOI2014 松鼠的新家

[22] NOIP2015 运输计划

[23] NOIP2016 天天爱跑步

5.9 二分

本页面将简要介绍二分查找，由二分法衍生的三分法以及二分答案。

二分法

定义

二分查找（英语：binary search），也称折半搜索（英语：half-interval search）、对数搜索（英语：logarithmic search），是用来在一个有序数组中查找某一元素的算法。

过程

以在一个升序数组中查找一个数为例。

它每次考察数组当前部分的中间元素，如果中间元素刚好是要找的，就结束搜索过程；如果中间元素小于所查找的值，那么左侧的只会更小，不会有所查找的元素，只需到右侧查找；如果中间元素大于所查找的值同理，只需到左侧查找。

性质

时间复杂度

二分查找的最优时间复杂度为 $O(1)$。

二分查找的平均时间复杂度和最坏时间复杂度均为 $O(\log n)$。因为在二分搜索过程中，算法每次都把查询的区间减半，所以对于一个长度为 n 的数组，至多会进行 $O(\log n)$ 次查找。

空间复杂度

迭代版本的二分查找的空间复杂度为 $O(1)$。

递归（无尾调用消除）版本的二分查找的空间复杂度为 $O(\log n)$。

实现

```cpp
int binary_search(int start, int end, int key) {
  int ret = -1;  // 未搜索到数据返回-1 下标
  int mid;
  while (start <= end) {
    mid = start + ((end - start) >> 1);  // 直接平均可能会溢出，所以用这个算法
    if (arr[mid] < key)
      start = mid + 1;
    else if (arr[mid] > key)
      end = mid - 1;
    else {  // 最后检测相等是因为多数搜索情况不是大于就是小于
      ret = mid;
      break;
    }
  }
  return ret;  // 单一出口
}
```

note

参考 编译优化 # 位运算代替乘法，对于 n 是有符号数的情况，当你可以保证 $n \geq 0$ 时，`n >> 1` 比 `n / 2` 指令数更少。

最大值最小化

注意，这里的有序是广义的有序，如果一个数组中的左侧或者右侧都满足某一种条件，而另一侧都不满足这种条件，也可以看作是一种有序（如果把满足条件看做 1，不满足看做 0，至少对于这个条件的这一维度是有序的）。换言之，二分搜索法可以用来查找满足某种条件的最大（最小）的值。

要求满足某种条件的最大值的最小可能情况（最大值最小化），首先的想法是从小到大枚举这个作为答案的「最大值」，然后去判断是否合法。若答案单调，就可以使用二分搜索法来更快地找到答案。因此，要想使用二分搜索法来解这种「最大值最小化」的题目，需要满足以下三个条件：

1. 答案在一个固定区间内；
2. 可能查找一个符合条件的值不是很容易，但是要求能比较容易地判断某个值是否是符合条件的；
3. 可行解对于区间满足一定的单调性。换言之，如果 x 是符合条件的，那么有 $x+1$ 或者 $x-1$ 也符合条件。（这样下来就满足了上面提到的单调性）

当然，最小值最大化是同理的。

STL 的二分查找

C++ 标准库中实现了查找首个不小于给定值的元素的函数 `std::lower_bound`[1] 和查找首个大于给定值的元素的函数 `std::upper_bound`[2]，二者均定义于头文件 `<algorithm>` 中。

二者均采用二分实现，所以调用前必须保证元素有序。

bsearch

bsearch 函数为 C 标准库实现的二分查找，定义在 `<stdlib.h>` 中。在 C++ 标准库里，该函数定义在 `<cstdlib>` 中。qsort 和 bsearch 是 C 语言中唯二的两个算法类函数。

bsearch 函数相比 qsort（排序相关 STL）的四个参数，在最左边增加了参数「待查元素的地址」。之所以按照地址的形式传入，是为了方便直接套用与 qsort 相同的比较函数，从而实现排序后的立即查找。因此这个参数不能直接传入具体值，而是要先将待查值用一个变量存储，再传入该变量地址。

于是 bsearch 函数总共有五个参数：待查元素的地址、数组名、元素个数、元素大小、比较规则。比较规则仍然通过指定比较函数实现，详见 排序相关 STL。

bsearch 函数的返回值是查找到的元素的地址，该地址为 void 类型。

注意：bsearch 与上文的 lower_bound 和 upper_bound 有两点不同：

- 当符合条件的元素有重复多个的时候，会返回执行二分查找时第一个符合条件的元素，从而这个元素可能位于重复多个元素的中间部分。
- 当查找不到相应的元素时，会返回 NULL。

用 lower_bound 可以实现与 bsearch 完全相同的功能，所以可以使用 bsearch 通过的题目，直接改写成 lower_bound 同样可以实现。但是鉴于上述不同之处的第二点，例如，在序列 1、2、4、5、6 中查找 3，bsearch 实现 lower_bound 的功能会变得困难。

利用 bsearch 实现 lower_bound 的功能比较困难，是否一定就不能实现？答案是否定的，存在比较 tricky 的技巧。借助编译器处理比较函数的特性：总是将第一个参数指向待查元素，将第二个参数指向待查数组中的元素，也可以用 bsearch 实现 lower_bound 和 upper_bound，如下文示例。只是，这要求待查数组必须是全局数组，从而可以直接传入首地址。

```cpp
int A[100005];  // 示例全局数组

// 查找首个不小于待查元素的元素的地址
int lower(const void *p1, const void *p2) {
  int *a = (int *)p1;
  int *b = (int *)p2;
  if ((b == A || compare(a, b - 1) > 0) && compare(a, b) > 0)
    return 1;
  else if (b != A && compare(a, b - 1) <= 0)
    return -1;  // 用到地址的减法，因此必须指定元素类型
  else
    return 0;
}

// 查找首个大于待查元素的元素的地址
int upper(const void *p1, const void *p2) {
  int *a = (int *)p1;
  int *b = (int *)p2;
  if ((b == A || compare(a, b - 1) >= 0) && compare(a, b) >= 0)
    return 1;
  else if (b != A && compare(a, b - 1) < 0)
    return -1;  // 用到地址的减法，因此必须指定元素类型
```

```
  else
    return 0;
}
```

因为现在的 OI 选手很少写纯 C，并且此方法作用有限，所以不是重点。对于新手而言，建议老老实实地使用 C++ 中的 lower_bound 和 upper_bound 函数。

二分答案

解题的时候往往会考虑枚举答案然后检验枚举的值是否正确。若满足单调性，则满足使用二分法的条件。把这里的枚举换成二分，就变成了「二分答案」。

"Luogu P1873 **砍树**[3]"

伐木工人米尔科需要砍倒 M 米长的木材。这是一个对米尔科来说很容易的工作，因为他有一个漂亮的新伐木机，可以像野火一样砍倒森林。不过，米尔科只被允许砍倒单行树木。

米尔科的伐木机工作过程如下：米尔科设置一个高度参数 H（米），伐木机升起一个巨大的锯片到高度 H，并锯掉所有的树比 H 高的部分（当然，树木不高于 H 米的部分保持不变）。米尔科就得到树木被锯下的部分。

例如，如果一行树的高度分别为 20，15，10，17，米尔科把锯片升到 15 米的高度，切割后树木剩下的高度将是 15，15，10，15，而米尔科将从第 1 棵树得到 5 米木材，从第 4 棵树得到 2 米木材，共 7 米木材。

米尔科非常关注生态保护，所以他不会砍掉过多的木材。这正是他尽可能高地设定伐木机锯片的原因。你的任务是帮助米尔科找到伐木机锯片的最大的整数高度 H，使得他能得到木材至少为 M 米。即，如果再升高 1 米锯片，则他将得不到 M 米木材。

" **解题思路**"

我们可以在 1 到 10^9 中枚举答案，但是这种朴素写法肯定拿不到满分，因为从 1 枚举到 10^9 太耗时间。我们可以在 $[1, 10^9]$ 的区间上进行二分作为答案，然后检查各个答案的可行性（一般使用贪心法）。**这就是二分答案**。

" **参考代码**"

```
int a[1000005];
int n, m;

bool check(int k) {  // 检查可行性，k 为锯片高度
  long long sum = 0;
  for (int i = 1; i <= n; i++)       // 检查每一棵树
    if (a[i] > k)                    // 如果树高于锯片高度
      sum += (long long)(a[i] - k);  // 累加树木长度
  return sum >= m;                   // 如果满足最少长度代表可行
}

int find() {
  int l = 1, r = 1e9 + 1;   // 因为是左闭右开的，所以 10^9 要加 1
  while (l + 1 < r) {       // 如果两点不相邻
    int mid = (l + r) / 2;  // 取中间值
    if (check(mid))         // 如果可行
      l = mid;              // 升高锯片高度
    else
      r = mid;  // 否则降低锯片高度
  }
  return l;  // 返回左边值
}
```

```cpp
int main() {
  cin >> n >> m;
  for (int i = 1; i <= n; i++) cin >> a[i];
  cout << find();
  return 0;
}
```

看完了上面的代码，你肯定会有两个疑问：

1. 为何搜索区间是左闭右开的？

   因为搜到最后，会这样（以合法的最大值为例）：

图 5.14

然后会

图 5.15

合法的最小值恰恰相反。

2. 为何返回左边值？

   同上。

三分法

引入

如果需要求出单峰函数的极值点，通常使用二分法衍生出的三分法求单峰函数的极值点。

" 为什么不通过求导函数的零点来求极值点？"

客观上，求出导数后，通过二分法求出导数的零点（由于函数是单峰函数，其导数在同一范围内的零点是唯一的）得到单峰函数的极值点是可行的。

但首先，对于一些函数，求导的过程和结果比较复杂。

其次，某些题中需要求极值点的单峰函数并非一个单独的函数，而是多个函数进行特殊运算得到的函数（如求多个单调性不完全相同的一次函数的最小值的最大值）。此时函数的导函数可能是分段函数，且在函数某些点上可能不可导。

" 注意"

只要函数是单峰函数，三分法既可以求出其最大值，也可以求出其最小值。为行文方便，除特殊说明外，下文中均以求单峰函数的最小值为例。

三分法与二分法的基本思想类似，但每次操作需在当前区间 $[l, r]$（下图中除去虚线范围内的部分）内任取两点 $lmid, rmid(lmid < rmid)$（下图中的两蓝点）。如下图，如果 $f(lmid) < f(rmid)$，则在 $[rmid, r]$（下图中的红色部分）中函数必然单调递增，最小值所在点（下图中的绿点）必然不在这一区间内，可舍去这一区间。反之亦然。

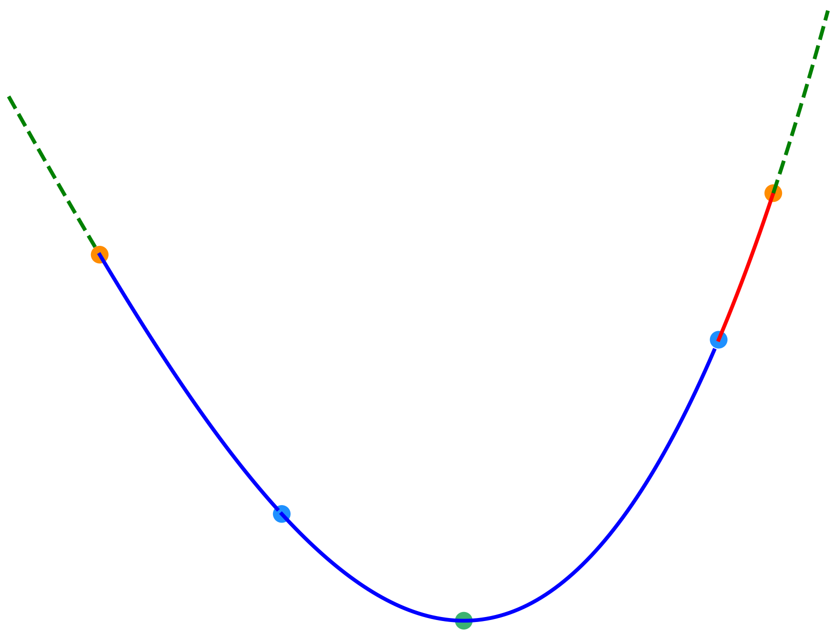


图 5.16

" 注意"

在计算 $lmid$ 和 $rmid$ 时，需要防止数据溢出的现象出现。

三分法每次操作会舍去两侧区间中的其中一个。为减少三分法的操作次数，应使两侧区间尽可能大。因此，每一次操作时的 $lmid$ 和 $rmid$ 分别取 $mid-\varepsilon$ 和 $mid+\varepsilon$ 是一个不错的选择。

实现

伪代码

Input. A range $[l, r]$ meaning that the domain of $f(x)$.
Output. The maximum value of $f(x)$ and the value of x at that time .
Method.
while $r - l > \varepsilon$
  $mid \leftarrow \frac{l+r}{2}$
  $lmid \leftarrow mid - \varepsilon$
  $rmid \leftarrow mid + \varepsilon$
  if $f(lmid) < f(rmid)$
    $r \leftarrow mid$
  else
    $l \leftarrow mid$

C++

```
while (r - l > eps) {
  mid = (l + r) / 2;
  lmid = mid - eps;
  rmid = mid + eps;
  if (f(lmid) < f(rmid))
    r = mid;
  else
    l = mid;
}
```

例题

"洛谷 P3382 -【模板】三分法[4]"

给定一个 N 次函数和范围 $[l, r]$，求出使函数在 $[l, x]$ 上单调递增且在 $[x, r]$ 上单调递减的唯一的 x 的值。

"解题思路"

本题要求求 N 次函数在 $[l, r]$ 取最大值时自变量的值，显然可以使用三分法。

"参考代码"

```
#include <cmath>
#include <cstdio>
using namespace std;

const double eps = 0.0000001;
int N;
double l, r, A[20], mid, lmid, rmid;

double f(double x) {
```

```
  double res = (double)0;
  for (int i = N; i >= 0; i--) res += A[i] * pow(x, i);
  return res;
}

int main() {
  scanf("%d%lf%lf", &N, &l, &r);
  for (int i = N; i >= 0; i--) scanf("%lf", &A[i]);
  while (r - l > eps) {
    mid = (l + r) / 2;
    lmid = mid - eps;
    rmid = mid + eps;
    if (f(lmid) > f(rmid))
      r = mid;
    else
      l = mid;
  }
  printf("%6lf", l);
  return 0;
}
```

```
eps = 1e-6
n, l, r = map(float, input().split())
a = tuple(map(float, input().split()))[::-1]

def f(x):
    return sum(x ** i * j for i, j in enumerate(a))

while r - l > eps:
    mid = (l + r) / 2
    if f(mid - eps) > f(mid + eps):
        r = mid
    else:
        l = mid
print(l)
```

习题

- Uva 1476 - Error Curves[5]
- Uva 10385 - Duathlon[6]
- UOJ 162 -【清华集训 2015】灯泡测试[7]
- 洛谷 P7579 -「RdOI R2」称重（weigh）[8]

分数规划

参见：分数规划

分数规划通常描述为下列问题：每个物品有两个属性 c_i，d_i，要求通过某种方式选出若干个，使得 $\frac{\sum c_i}{\sum d_i}$ 最大或最小。

经典的例子有最优比率环、最优比率生成树等等。

分数规划可以用二分法来解决。

参考资料与注释

[1] std::lower_bound

[2] std::upper_bound

[3] Luogu P1873 砍树

[4] 洛谷 P3382 -【模板】三分法

[5] Uva 1476 - Error Curves

[6] Uva 10385 - Duathlon

[7] UOJ 162 -【清华集训 2015】灯泡测试

[8] 洛谷 P7579 -「RdOI R2」称重（weigh）

5.10 倍增

Authors: Ir1d, ShadowsEpic, Fomalhauthmj, siger-young, MingqiHuang, Xeonacid, hsfzLZH1, orzAtalod, NachtgeistW

本页面将简要介绍倍增法。

定义

倍增法（英语：binary lifting），顾名思义就是翻倍。它能够使线性的处理转化为对数级的处理，大大地优化时间复杂度。

这个方法在很多算法中均有应用，其中最常用的是 RMQ 问题和求 LCA（最近公共祖先）。

应用

RMQ 问题

参见：RMQ 专题

RMQ 是 Range Maximum/Minimum Query 的缩写，表示区间最大（最小）值。使用倍增思想解决 RMQ 问题的方法是 ST 表。

树上倍增求 LCA

参见：最近公共祖先

例题

题 1

" 例题"

如何用尽可能少的砝码称量出 $[0,31]$ 之间的所有重量?（只能在天平的一端放砝码）

" 解题思路"

答案是使用 1 2 4 8 16 这五个砝码，可以称量出 $[0,31]$ 之间的所有重量。同样，如果要称量 $[0,127]$ 之间的所有重量，可以使用 1 2 4 8 16 32 64 这七个砝码。每次我们都选择 2 的整次幂作砝码的重量，就可以使用极少的砝码个数量出任意我们所需要的重量。

为什么说是极少呢? 因为如果我们要量出 $[0,1023]$ 之间的所有重量，只需要 10 个砝码，需要量出 $[0,1048575]$ 之间的所有重量，只需要 20 个。如果我们的目标重量翻倍，砝码个数只需要增加 1。这叫「对数级」的增长速度，因为砝码的所需个数与目标重量的范围的对数成正比。

题 2

" 例题"

给出一个长度为 n 的环和一个常数 k，每次会从第 i 个点跳到第 $(i+k) \bmod n+1$ 个点，总共跳了 m 次。每个点都有一个权值，记为 a_i，求 m 次跳跃的起点的权值之和对 10^9+7 取模的结果。

数据范围：$1 \le n \le 10^6$，$1 \le m \le 10^{18}$，$1 \le k \le n$，$0 \le a_i \le 10^9$。

" 解题思路"

这里显然不能暴力模拟跳 m 次。因为 m 最大可到 10^{18} 级别，如果暴力模拟的话，时间承受不住。

所以就需要进行一些预处理，提前整合一些信息，以便于在查询的时候更快得出结果。如果记录下来每一个可能的跳跃次数的结果的话，不论是时间还是空间都难以承受。

那么应该如何预处理呢? 看看第一道例题。有思路了吗?

回到本题。我们要预处理一些信息，然后用预处理的信息尽量快的整合出答案。同时预处理的信息也不能太多。所以可以预处理出以 2 的整次幂为单位的信息，这样的话在预处理的时候只需要处理少量信息，在整合的时候也不需要大费周章。

在这题上，就是我们预处理出从每个点开始跳 1、2、4、8 等等步之后的结果（所处点和点权和），然后如果要跳 13 步，只需要跳 1+4+8 步就好了。也就是说先在起始点跳 1 步，然后再在跳了之后的终点跳 4 步，再接着跳 8 步，同时统计一下预先处理好的点权和，就可以知道跳 13 步的点权和了。

对于每一个点开始的 2^i 步，记录一个 `go[i][x]` 表示第 x 个点跳 2^i 步之后的终点，而 `sum[i][x]` 表示第 x 个点跳 2^i 步之后能获得的点权和。预处理的时候，开两重循环，对于跳 2^i 步的信息，我们可以看作是先跳了 2^{i-1} 步，再跳 2^{i-1} 步，因为显然有 $2^{i-1}+2^{i-1}=2^i$。即我们有 `sum[i][x] = sum[i-1][x]+sum[i-1][go[i-1][x]]`，且 `go[i][x] = go[i-1][go[i-1][x]]`。

当然还有一些实现细节需要注意。为了保证统计的时候不重不漏，我们一般预处理出「左闭右开」的点权和。亦即，对于跳 1 步的情况，我们只记录该点的点权和；对于跳 2 步的情况，我们只记录该点及其下一个点的点权和。相当于总是不将终点的点权和计入 sum。这样在预处理的时候，只需要将两部分的点权和直接相加就可以了，不需要担心第一段的终点和第二段的起点会被重复计算。

这题的 $m \le 10^{18}$，虽然看似恐怖，但是实际上只需要预处理出 65 以内的 i，就可以轻松解决，比起暴力枚举快了很多。用行话讲，这个做法的 时间复杂度 是预处理 $\Theta(n\log m)$，查询每次 $\Theta(\log m)$。

" 参考代码"

```
#include <cstdio>
using namespace std;
```

```cpp
const int mod = 1000000007;

int modadd(int a, int b) {
  if (a + b >= mod) return a + b - mod;  // 减法代替取模，加快运算
  return a + b;
}

int vi[1000005];

int go[75][1000005];  // 将数组稍微开大以避免越界，小的一维尽量定义在前面
int sum[75][1000005];

int main() {
  int n, k;
  scanf("%d%d", &n, &k);
  for (int i = 1; i <= n; ++i) {
    scanf("%d", vi + i);
  }

  for (int i = 1; i <= n; ++i) {
    go[0][i] = (i + k) % n + 1;
    sum[0][i] = vi[i];
  }

  int logn = 31 - __builtin_clz(n);  // 一个快捷的取对数的方法
  for (int i = 1; i <= logn; ++i) {
    for (int j = 1; j <= n; ++j) {
      go[i][j] = go[i - 1][go[i - 1][j]];
      sum[i][j] = modadd(sum[i - 1][j], sum[i - 1][go[i - 1][j]]);
    }
  }

  long long m;
  scanf("%lld", &m);

  int ans = 0;
  int curx = 1;
  for (int i = 0; m; ++i) {
    if (m & (1ll << i)) {  // 参见位运算的相关内容，意为 m 的第 i 位是否为 1
      ans = modadd(ans, sum[i][curx]);
      curx = go[i][curx];
      m ^= 1ll << i;  // 将第 i 位置零
    }
  }

  printf("%d\n", ans);
}
```

5.11 构造

Authors: leoleoasd, yzxoi

本页面将简要介绍构造题这类题型。

引入

构造题是比赛中常见的一类题型。

从形式上来看，问题的答案往往具有某种规律性，使得在问题规模迅速增大的时候，仍然有机会比较容易地得到答案。

这要求解题时要思考问题规模增长对答案的影响，这种影响是否可以推广。例如，在设计动态规划方法的时候，要考虑从一个状态到后继状态的转移会造成什么影响。

特点

构造题一个很显著的特点就是高自由度，也就是说一道题的构造方式可能有很多种，但是会有一种较为简单的构造方式满足题意。看起来是放宽了要求，让题目变的简单了，但很多时候，正是这种高自由度导致题目没有明确思路而无从下手。

构造题另一个特点就是形式灵活，变化多样。并不存在一个通用解法或套路可以解决所有构造题，甚至很难找出解题思路的共性。

例题

下面将列举一些例题帮助读者体会构造题的一些思想内涵，给予思路上的启发。建议大家深入思考后再查看题解，也欢迎大家参与分享有趣的构造题。

例题 1

"Codeforces Round #384 (Div. 2) C.Vladik and fractions[1]"

构造一组 x, y, z，使得对于给定的 n，满足 $\dfrac{1}{x}+\dfrac{1}{y}+\dfrac{1}{z}=\dfrac{2}{n}$

" **解题思路**"

从样例二可以看出本题的构造方法。

显然 $n, n+1, n(n+1)$ 为一组合法解。特殊地，当 $n=1$ 时，无解，这是因为 $n+1$ 与 $n(n+1)$ 此时相等。

至于构造思路是怎么产生的，大概就是观察样例加上一点点数感了吧。此题对于数学直觉较强的人来说并不难。

例题 2

"Luogu P3599 Koishi Loves Construction[2]"

Task1：试判断能否构造并构造一个长度为 n 的 $1 \dots n$ 的排列，满足其 n 个前缀和在模 n 的意义下互不相同

Task2：试判断能否构造并构造一个长度为 n 的 $1 \dots n$ 的排列，满足其 n 个前缀积在模 n 的意义下互不相同

" **解题思路**"

对于 task1：

当 n 为奇数时，无法构造出合法解；

当 n 为偶数时，可以构造一个形如 $n, 1, n-2, 3, \cdots$ 这样的数列。

首先，我们可以发现 n 必定出现在数列的第一位，否则 n 出现前后的两个前缀和必然会陷入模意义下相等的尴尬境地；

然后，我们考虑构造出整个序列的方式：

考虑通过构造前缀和序列的方式来获得原数列，可以发现前缀和序列两两之间的差在模意义下不能相等，因为前缀和序列的差分序列对应着原来的排列。

因此我们尝试以前缀和数列在模意义下为

$$0, 1, -1, 2, -2, \cdots$$

这样的形式来构造这个序列，不难发现它完美地满足所有限制条件。

对于 task2：

当 n 为除 4 以外的合数时，无法构造出合法解

当 n 为质数或 4 时，可以构造一个形如 $1, \frac{2}{1}, \frac{3}{2}, \cdots, \frac{n-1}{n-2}, n$ 这样的数列

先考虑什么时候有解：

显然，当 n 为合数时无解。因为对于一个合数来说，存在两个比它小的数 p, q 使得 $p \times q \equiv 0 \pmod n$，如 $(3 \times 6)\%9 = 0$。那么，当 p, q 均出现过后，数列的前缀积将一直为 0，故合数时无解。特殊地，我们可以发现 $4 = 2 \times 2$，无满足条件的 p, q，因此存在合法解。

我们考虑如何构造这个数列：

和 task1 同样的思路，我们发现 1 必定出现在数列的第一位，否则 1 出现前后的两个前缀积必然相等；而 n 必定出现在数列的最后一位，因为 n 出现位置后的所有前缀积在模意义下都为 0。分析题目给出的几组样例以后发现，所有样例中均有一组合法解满足前缀积在模意义下为 $1, 2, 3, \cdots, n$，因此我们可以构造出上文所述的数列来满足这个条件。那么我们只需证明这 n 个数互不相同即可。

我们发现这些数均为 $1 \cdots n-2$ 的逆元 $+1$，因此各不相同，此题得解。

例题 3

"AtCoder Grand Contest 032 B[3]"

给定一个整数 N，试构造一个节点数为 N 无向图。令节点编号为 $1 \ldots N$，要求其满足以下条件：

- 这是一个简单连通图。
- 存在一个整数 S 使得对于任意节点，与其相邻节点的下标和为 S。

保证输入数据有解。

" **解题思路**"

通过分析 $n = 3, 4, 5$ 的情况，我们可以找到一个构造思路。

构造一个完全 k 分图，保证这 k 部分和相等。则每个点的 S 均相等，为 $\frac{(k-1)\sum_{i=1}^{n} i}{k}$。

如果 n 为偶数，那么我们可以前后两两配对，即 $\{1, n\}, \{2, n-1\} \cdots$

如果 n 为奇数，那么我们可以把 n 单拿出来作为一组，剩余的 $n-1$ 个两两配对，即 $\{n\}, \{1, n-1\}, \{2, n-2\} \cdots$

这样构造出的图在 $n \geq 3$ 时连通性易证，在此不加赘述。

此题得解。

例题 4

"BZOJ 4971「Lydsy1708 **月赛**」**记忆中的背包**"

经过一天辛苦的工作，小 Q 进入了梦乡。他脑海中浮现出了刚进大学时学 01 背包的情景，那时还是大一萌新的小 Q 解决了一道简单的 01 背包问题。这个问题是这样的：

给定 n 个物品，每个物品的体积分别为 $v_1, v_2, ..., v_n$，请计算从中选择一些物品（也可以不选），使得总体积恰好为 w 的方案数。因为答案可能非常大，你只需要输出答案对 P 取模的结果。

因为长期熬夜刷题，他只看到样例输入中的 w 和 P，以及样例输出是 k，看不清到底有几个物品，也看不清每个物品的体积是多少。直到梦醒，小 Q 也没有看清 n 和 v，请写一个程序，帮助小 Q 一起回忆曾经的样例输入。

" 解题思路"

这道题是自由度最高的构造题之一了。这就导致了没有头绪，难以入手的情况。

首先，不难发现模数是假的。由于我们自由构造数据，我们一定可以让方案数不超过模数。

通过奇怪的方式，我们想到可以通过构造 n 个代价为 1 的小物品和几个代价大于 $\frac{w}{2}$ 的大物品。

由于大物品只能取一件，所以每个代价为 x 的大物品对方案数的贡献为 $\binom{n}{w-x}$。

令 $f_{i,j}$ 表示有 i 个 1，方案数为 j 的最小大物品数。

用 dp 预处理出 f，通过计算可知只需预处理 $i \leq 20$ 的所有值即可。

此题得解。

参考资料与注释

[1] Codeforces Round #384 (Div. 2) C.Vladik and fractions

[2] Luogu P3599 Koishi Loves Construction

[3] AtCoder Grand Contest 032 B

第 6 章

搜索

6.1　搜索部分简介

搜索，也就是对状态空间进行枚举，通过穷尽所有的可能来找到最优解，或者统计合法解的个数。

搜索有很多优化方式，如减小状态空间，更改搜索顺序，剪枝等。

搜索是一些高级算法的基础。在 OI 中，纯粹的搜索往往也是得到部分分的手段，但可以通过纯粹的搜索拿到满分的题目非常少。

习题

- 「kuangbin 带你飞」专题一 简单搜索[1]
- 「kuangbin 带你飞」专题二 搜索进阶[2]
- 洛谷搜索题单[3]
- openjudge 搜索题单[4]

参考资料与注释

[1] 「kuangbin 带你飞」专题一简单搜索

[2] 「kuangbin 带你飞」专题二搜索进阶

[3] 洛谷搜索题单

[4] openjudge 搜索题单

6.2　DFS（搜索）

引入

DFS 为图论中的概念，详见 DFS（图论）页面。在**搜索算法**中，该词常常指利用递归函数方便地实现暴力枚举的算法，与图论中的 DFS 算法有一定相似之处，但并不完全相同。

解释

考虑这个例子：

" 例题"

把正整数 n 分解为 3 个不同的正整数，如 $6=1+2+3$，排在后面的数必须大于等于前面的数，输出所有方案。

对于这个问题，如果不知道搜索，应该怎么办呢？

当然是三重循环，参考代码如下：

" 实现"

```
for (int i = 1; i <= n; ++i)
  for (int j = i; j <= n; ++j)
    for (int k = j; k <= n; ++k)
      if (i + j + k == n) printf("%d = %d + %d + %d\n", n, i, j, k);
```

```
for i in range(1, n + 1):
    for j in range(i, n + 1):
        for k in range(j, n + 1):
            if i + j + k == n:
                print("%d = %d + %d + %d" % (n, i, j, k))
```

```
for (int i = 1; i < n + 1; i++) {
    for (int j = i; j < n + 1; j++) {
        for (int k = j; k < n + 1; k++) {
            if (i + j + k == n) System.out.printf("%d = %d + %d + %d%n", n, i, j
, k);
        }
    }
}
```

那如果是分解成四个整数呢？再加一重循环？

那分解成小于等于 m 个整数呢？

这时候就需要用到递归搜索了。

该类搜索算法的特点在于，将要搜索的目标分成若干「层」，每层基于前几层的状态进行决策，直到达到目标状态。

考虑上述问题，即将正整数 n 分解成小于等于 m 个正整数之和，且排在后面的数必须大于等于前面的数，并输出所有方案。

设一组方案将正整数 n 分解成 k 个正整数 $a_1,a_2,\ldots,a_k$ 的和。

我们将问题分层，第 i 层决定 a_i。则为了进行第 i 层决策，我们需要记录三个状态变量：$n-\sum_{j=1}^{i}a_j$，表示后面所有正整数的和；以及 a_{i-1}，表示前一层的正整数，以确保正整数递增；以及 i，确保我们最多输出 m 个正整数。

为了记录方案，我们用 arr 数组，第 i 项表示 a_i. 注意到 arr 实际上是一个长度为 i 的栈。

代码如下：

" 实现"

```
int m, arr[103];  // arr 用于记录方案

void dfs(int n, int i, int a) {
```

```
  if (n == 0) {
    for (int j = 1; j <= i - 1; ++j) printf("%d ", arr[j]);
    printf("\n");
  }
  if (i <= m) {
    for (int j = a; j <= n; ++j) {
      arr[i] = j;
      dfs(n - j, i + 1, j);  // 请仔细思考该行含义。
    }
  }
}

// 主函数
scanf("%d%d", &n, &m);
dfs(n, 1, 1);
```

```
arr = [0] * 103  # arr 用于记录方案

def dfs(n, i, a):
    if n == 0:
        print(arr[1:i])
    if i <= m:
        for j in range(a, n + 1):
            arr[i] = j
            dfs(n - j, i + 1, j)  # 请仔细思考该行含义。

# 主函数
n, m = map(int, input().split())
dfs(n, 1, 1)
```

```
static int m;

// arr 用于记录方案
static int[] arr = new int[103];

public static void dfs(int n, int i, int a) {
    if (n == 0) {
        for (int j = 1; j <= i - 1; j++) System.out.printf("%d ", arr[j]);
        System.out.println();
    }
    if (i <= m) {
        for (int j = a; j <= n; ++j) {
            arr[i] = j;
            dfs(n - j, i + 1, j); // 请仔细思考该行含义。
        }
    }
}

// 主函数
final int N = new Scanner(System.in).nextInt();
m = new Scanner(System.in).nextInt();
dfs(N, 1, 1);
```

例题

"Luogu P1706 全排列问题[1]"

```cpp
#include <iomanip>
#include <iostream>
using namespace std;
int n;
bool vis[50];  // 访问标记数组
int a[50];     // 排列数组，按顺序储存当前搜索结果

void dfs(int step) {
  if (step == n + 1) {  // 边界
    for (int i = 1; i <= n; i++) {
      cout << setw(5) << a[i];  // 保留 5 个场宽
    }
    cout << endl;
    return;
  }
  for (int i = 1; i <= n; i++) {
    if (vis[i] == 0) {  // 判断数字 i 是否在正在进行的全排列中
      vis[i] = 1;
      a[step] = i;
      dfs(step + 1);
      vis[i] = 0;  // 这一步不使用该数置 0 后允许下一步使用
    }
  }
  return;
}

int main() {
  cin >> n;
  dfs(1);
  return 0;
}
```

参考资料与注释

[1] Luogu P1706 全排列问题

6.3 BFS（搜索）

BFS 是图论中的一种遍历算法，详见 BFS。

BFS 在搜索中也很常用，将每个状态对应为图中的一个点即可。

6.4 双向搜索

Authors: FFjet, ChungZH, frank-xjh, hsfzLZH1, Xarfa, AndrewWayne

本页面将简要介绍两种双向搜索算法：「双向同时搜索」和「Meet in the middle」。

双向同时搜索

定义

双向同时搜索的基本思路是从状态图上的起点和终点同时开始进行 广搜 或 深搜。

如果发现搜索的两端相遇了，那么可以认为是获得了可行解。

过程

双向广搜的步骤：

```
将开始结点和目标结点加入队列 q
标记开始结点为 1
标记目标结点为 2
while (队列 q 不为空)
{
  从 q.front() 扩展出新的 s 个结点

  如果新扩展出的结点已经被其他数字标记过
    那么表示搜索的两端碰撞
    那么循环结束

  如果新的 s 个结点是从开始结点扩展来的
    那么将这个 s 个结点标记为 1 并且入队 q

  如果新的 s 个结点是从目标结点扩展来的
    那么将这个 s 个结点标记为 2 并且入队 q
}
```

Meet in the middle

warning

本节要介绍的不是 **二分搜索**（二分搜索的另外一个译名为「折半搜索」）。

引入

Meet in the middle 算法没有正式译名，常见的翻译为「折半搜索」、「双向搜索」或「中途相遇」。

它适用于输入数据较小，但还没小到能直接使用暴力搜索的情况。

过程

Meet in the middle 算法的主要思想是将整个搜索过程分成两半，分别搜索，最后将两半的结果合并。

性质

暴力搜索的复杂度往往是指数级的，而改用 meet in the middle 算法后复杂度的指数可以减半，即让复杂度从 $O(a^b)$ 降到 $O(a^{b/2})$。

例题

"例题 「USACO09NOV」 灯 Lights[1]"

有 n 盏灯，每盏灯与若干盏灯相连，每盏灯上都有一个开关，如果按下一盏灯上的开关，这盏灯以及与之相连的所有灯的开关状态都会改变。一开始所有灯都是关着的，你需要将所有灯打开，求最小的按开关次数。

$1 \le n \le 35$。

"解题思路"

如果这道题暴力 DFS 找开关灯的状态，时间复杂度就是 $O(2^n)$，显然超时。不过，如果我们用 meet in middle 的话，时间复杂度可以优化至 $O(n2^{n/2})$。meet in middle 就是让我们先找一半的状态，也就是找出只使用编号为 1 到 mid 的开关能够到达的状态，再找出只使用另一半开关能到达的状态。如果前半段和后半段开启的灯互补，将这两段合并起来就得到了一种将所有灯打开的方案。具体实现时，可以把前半段的状态以及达到每种状态的最少按开关次数存储在 map 里面，搜索后半段时，每搜出一种方案，就把它与互补的第一段方案合并来更新答案。

"参考代码"

```cpp
#include <algorithm>
#include <cstdio>
#include <iostream>
#include <map>

using namespace std;

int n, m, ans = 0x7fffffff;
map<long long, int> f;
long long a[40];

int main() {
  cin >> n >> m;
  a[0] = 1;
  for (int i = 1; i < n; ++i) a[i] = a[i - 1] * 2;  // 进行预处理

  for (int i = 1; i <= m; ++i) {  // 对输入的边的情况进行处理
    int u, v;
    cin >> u >> v;
    --u;
    --v;
    a[u] |= ((long long)1 << v);
    a[v] |= ((long long)1 << u);
  }

  for (int i = 0; i < (1 << (n / 2)); ++i) {  // 对前一半进行搜索
    long long t = 0;
    int cnt = 0;
    for (int j = 0; j < n / 2; ++j) {
      if ((i >> j) & 1) {
        t ^= a[j];
        ++cnt;
      }
    }
    if (!f.count(t))
```

```
      f[t] = cnt;
    else
      f[t] = min(f[t], cnt);
  }

  for (int i = 0; i < (1 << (n - n / 2)); ++i) {  // 对后一半进行搜索
    long long t = 0;
    int cnt = 0;
    for (int j = 0; j < (n - n / 2); ++j) {
      if ((i >> j) & 1) {
        t ^= a[n / 2 + j];
        ++cnt;
      }
    }
    if (f.count((((long long)1 << n) - 1) ^ t))
      ans = min(ans, cnt + f[(((long long)1 << n) - 1) ^ t]);
  }

  cout << ans;

  return 0;
}
```

外部链接

- What is meet in the middle algorithm w.r.t. competitive programming? - Quora[2]
- Meet in the Middle Algorithm - YouTube[3]

参考资料与注释

[1] 「USACO09NOV」灯 Lights

[2] What is meet in the middle algorithm w.r.t. competitive programming? - Quora

[3] Meet in the Middle Algorithm - YouTube

6.5 启发式搜索

本页面将简要介绍启发式搜索及其用法。

定义

启发式搜索（英文：heuristic search）是一种在普通搜索算法的基础上引入了启发式函数的搜索算法。

启发式函数的作用是基于已有的信息对搜索的每一个分支选择都做估价，进而选择分支。简单来说，启发式搜索就是对取和不取都做分析，从中选取更优解或删去无效解。

例题

由于概念过于抽象，这里使用例题讲解。

"「NOIP2005 普及组」 采药[1]"

题目大意：有 N 种物品和一个容量为 W 的背包，每种物品有重量 w_i 和价值 v_i 两种属性，要求选若干个物品（每种物品只能选一次）放入背包，使背包中物品的总价值最大，且背包中物品的总重量不超过背包的容量。

" 解题思路"

我们写一个估价函数 f，可以剪掉所有无效的 0 枝条（就是剪去大量无用不选枝条）。

估价函数 f 的运行过程如下：

我们在取的时候判断一下是不是超过了规定体积（可行性剪枝）；在不取的时候判断一下不取这个时，剩下的药所有的价值 + 现有的价值是否大于目前找到的最优解（最优性剪枝）。

" 示例代码"

```cpp
#include <algorithm>
#include <cstdio>
using namespace std;
const int N = 105;
int n, m, ans;

struct Node {
  int a, b;  // a 代表时间，b 代表价值
  double f;
} node[N];

bool operator<(Node p, Node q) { return p.f > q.f; }

int f(int t, int v) {  // 计算在当前时间下，剩余物品的最大价值
  int tot = 0;
  for (int i = 1; t + i <= n; i++)
    if (v >= node[t + i].a) {
      v -= node[t + i].a;
      tot += node[t + i].b;
    } else
      return (int)(tot + v * node[t + i].f);
  return tot;
}

void work(int t, int p, int v) {
  ans = max(ans, v);
  if (t > n) return;                          // 边界条件：只有 n 种物品
  if (f(t, p) + v > ans) work(t + 1, p, v);  // 最优性剪枝
  if (node[t].a <= p) work(t + 1, p - node[t].a, v + node[t].b);  // 可行性剪枝
}

int main() {
  scanf("%d %d", &m, &n);
  for (int i = 1; i <= n; i++) {
    scanf("%d %d", &node[i].a, &node[i].b);
    node[i].f = 1.0 * node[i].b / node[i].a;  // f 为性价比
  }
  sort(node + 1, node + n + 1);  // 根据性价比排序
  work(1, m, 0);
```

```
    printf("%d\n", ans);
    return 0;
}
```

参考资料与注释

[1] 「NOIP2005 普及组」采药

6.6 A*

本页面将简要介绍 A * 算法。

定义

A * 搜索算法（英文：A*search algorithm，A * 读作 A-star），简称 A * 算法，是一种在图形平面上，对于有多个节点的路径求出最低通过成本的算法。它属于图遍历（英文：Graph traversal）和最佳优先搜索算法（英文：Best-first search），亦是 BFS 的改进。

过程

定义起点 s，终点 t，从起点（初始状态）开始的距离函数 $g(x)$，到终点（最终状态）的距离函数 $h(x)$，$h^*(x)$[1]，以及每个点的估价函数 $f(x)=g(x)+h(x)$。

A * 算法每次从优先队列中取出一个 f 最小的元素，然后更新相邻的状态。

如果 $h\leq h*$，则 A * 算法能找到最优解。

上述条件下，如果 h 满足三角形不等式，则 A * 算法不会将重复结点加入队列。

当 $h=0$ 时，A * 算法变为 Dijkstra；当 $h=0$ 并且边权为 1 时变为 BFS。

例题

"八数码[2]"

题目大意：在 3×3 的棋盘上，摆有八个棋子，每个棋子上标有 1 至 8 的某一数字。棋盘中留有一个空格，空格用 0 来表示。空格周围的棋子可以移到空格中，这样原来的位置就会变成空格。给出一种初始布局和目标布局（为了使题目简单，设目标状态如下），找到一种从初始布局到目标布局最少步骤的移动方法。

```
123
804
765
```

" 解题思路"

h 函数可以定义为，不在应该在的位置的数字个数。

容易发现 h 满足以上两个性质，此题可以使用 A * 算法求解。

" 参考代码"

```
#include <algorithm>
#include <cstdio>
```

```cpp
#include <cstring>
#include <queue>
#include <set>
using namespace std;
const int dx[4] = {1, -1, 0, 0}, dy[4] = {0, 0, 1, -1};
int fx, fy;
char ch;

struct matrix {
  int a[5][5];

  bool operator<(matrix x) const {
    for (int i = 1; i <= 3; i++)
      for (int j = 1; j <= 3; j++)
        if (a[i][j] != x.a[i][j]) return a[i][j] < x.a[i][j];
    return false;
  }
} f, st;

int h(matrix a) {
  int ret = 0;
  for (int i = 1; i <= 3; i++)
    for (int j = 1; j <= 3; j++)
      if (a.a[i][j] != st.a[i][j]) ret++;
  return ret;
}

struct node {
  matrix a;
  int t;

  bool operator<(node x) const { return t + h(a) > x.t + h(x.a); }
} x;

priority_queue<node> q;  // 搜索队列
set<matrix> s;           // 防止搜索队列重复

int main() {
  st.a[1][1] = 1;  // 定义标准表
  st.a[1][2] = 2;
  st.a[1][3] = 3;
  st.a[2][1] = 8;
  st.a[2][2] = 0;
  st.a[2][3] = 4;
  st.a[3][1] = 7;
  st.a[3][2] = 6;
  st.a[3][3] = 5;
  for (int i = 1; i <= 3; i++)  // 输入
    for (int j = 1; j <= 3; j++) {
      scanf(" %c", &ch);
      f.a[i][j] = ch - '0';
    }
  q.push({f, 0});
  while (!q.empty()) {
```

```
    x = q.top();
    q.pop();
    if (!h(x.a)) {  // 判断是否与标准矩阵一致
      printf("%d\n", x.t);
      return 0;
    }
    for (int i = 1; i <= 3; i++)
      for (int j = 1; j <= 3; j++)
        if (!x.a.a[i][j]) fx = i, fy = j;  // 查找空格子（0 号点）的位置
    for (int i = 0; i < 4; i++) {  // 对四种移动方式分别进行搜索
      int xx = fx + dx[i], yy = fy + dy[i];
      if (1 <= xx && xx <= 3 && 1 <= yy && yy <= 3) {
        swap(x.a.a[fx][fy], x.a.a[xx][yy]);
        if (!s.count(x.a))
          s.insert(x.a),
              q.push({x.a, x.t + 1});  // 这样移动后，将新的情况放入搜索队列中
        swap(x.a.a[fx][fy], x.a.a[xx][yy]);  // 如果不这样移动的情况
      }
    }
  }
  return 0;
}
```

注：对于 k 短路问题，原题已经可以构造出数据使得 A* 算法无法通过，故本题思路仅供参考，A* 算法非正解，正解为可持久化可并堆做法，请移步 k 短路问题

"k 短路[3]"

按顺序求一个有向图上从结点 s 到结点 t 的所有路径最小的前任意多（不妨设为 k）个。

" 解题思路"

很容易发现，这个问题很容易转化成用 A * 算法解决问题的标准程式。

初始状态为处于结点 s，最终状态为处于结点 t，距离函数为从 s 到当前结点已经走过的距离，估价函数为从当前结点到结点 t 至少要走过的距离，也就是当前结点到结点 t 的最短路。

就这样，我们在预处理的时候反向建图，计算出结点 t 到所有点的最短路，然后将初始状态塞入优先队列，每次取出 $f(x)=g(x)+h(x)$ 最小的一项，计算出其所连结点的信息并将其也塞入队列。当你第 k 次走到结点 t 时，也就算出了结点 s 到结点 t 的 k 短路。

由于设计的距离函数和估价函数，每个状态需要存储两个参数，当前结点 x 和已经走过的距离 v。

我们可以在此基础上加一点小优化：由于只需要求出第 k 短路，所以当我们第 $k+1$ 次或以上走到该结点时，直接跳过该状态。因为前面的 k 次走到这个点的时候肯定能因此构造出 k 条路径，所以之后再加边更无必要。

" 参考代码"

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <queue>
using namespace std;
const int maxn = 5010;
const int maxm = 400010;
const double inf = 2e9;
```

```cpp
int n, m, k, u, v, cur, h[maxn], nxt[maxm], p[maxm], cnt[maxn], ans;
int cur1, h1[maxn], nxt1[maxm], p1[maxm];
double e, ww, w[maxm], f[maxn];
double w1[maxm];
bool tf[maxn];

void add_edge(int x, int y, double z) {  // 正向建图函数
  cur++;
  nxt[cur] = h[x];
  h[x] = cur;
  p[cur] = y;
  w[cur] = z;
}

void add_edge1(int x, int y, double z) {  // 反向建图函数
  cur1++;
  nxt1[cur1] = h1[x];
  h1[x] = cur1;
  p1[cur1] = y;
  w1[cur1] = z;
}

struct node {  // 使用 A* 时所需的结构体
  int x;
  double v;

  bool operator<(node a) const { return v + f[x] > a.v + f[a.x]; }
};

priority_queue<node> q;

struct node2 {  // 计算 t 到所有结点最短路时所需的结构体
  int x;
  double v;

  bool operator<(node2 a) const { return v > a.v; }
} x;

priority_queue<node2> Q;

int main() {
  scanf("%d%d%lf", &n, &m, &e);
  while (m--) {
    scanf("%d%d%lf", &u, &v, &ww);
    add_edge(u, v, ww);   // 正向建图
    add_edge1(v, u, ww);  // 反向建图
  }
  for (int i = 1; i < n; i++) f[i] = inf;
  Q.push({n, 0});
  while (!Q.empty()) {  // 计算 t 到所有结点的最短路
    x = Q.top();
    Q.pop();
    if (tf[x.x]) continue;
    tf[x.x] = true;
```

```
    f[x.x] = x.v;
    for (int j = h1[x.x]; j; j = nxt1[j]) Q.push({p1[j], x.v + w1[j]});
  }
  k = (int)e / f[1];
  q.push({1, 0});
  while (!q.empty()) {  // 使用 A* 算法
    node x = q.top();
    q.pop();
    cnt[x.x]++;
    if (x.x == n) {
      e -= x.v;
      if (e < 0) {
        printf("%d\n", ans);
        return 0;
      }
      ans++;
    }
    for (int j = h[x.x]; j; j = nxt[j])
      if (cnt[p[j]] <= k && x.v + w[j] <= e) q.push({p[j], x.v + w[j]});
  }
  printf("%d\n", ans);
  return 0;
}
```

参考资料与注释

[1] 此处的 h 意为 heuristic。详见 启发式搜索 - 维基百科 和 A*search algorithm - Wikipedia 的 Bounded relaxation 一节。

[2] 八数码

[3] k 短路

6.7 迭代加深搜索

定义

迭代加深是一种**每次限制搜索深度的**深度优先搜索。

解释

迭代加深搜索的本质还是深度优先搜索，只不过在搜索的同时带上了一个深度 d，当 d 达到设定的深度时就返回，一般用于找最优解。如果一次搜索没有找到合法的解，就让设定的深度加一，重新从根开始。

既然是为了找最优解，为什么不用 BFS 呢？我们知道 BFS 的基础是一个队列，队列的空间复杂度很大，当状态比较多或者单个状态比较大时，使用队列的 BFS 就显出了劣势。事实上，迭代加深就类似于用 DFS 方式实现的 BFS，它的空间复杂度相对较小。

当搜索树的分支比较多时，每增加一层的搜索复杂度会出现指数级爆炸式增长，这时前面重复进行的部分所带来的复杂度几乎可以忽略，这也就是为什么迭代加深是可以近似看成 BFS 的。

过程

首先设定一个较小的深度作为全局变量，进行 DFS。每进入一次 DFS，将当前深度加一，当发现 d 大于设定的深度 $limit$ 就返回。如果在搜索的途中发现了答案就可以回溯，同时在回溯的过程中可以记录路径。如果没有发现答案，就返回到函数入口，增加设定深度，继续搜索。

" 实现（伪代码）"

```
IDDFS(u,d)
    if d>limit
        return
    else
        for each edge (u,v)
            IDDFS(v,d+1)
return
```

注意事项

在大多数的题目中，广度优先搜索还是比较方便的，而且容易判重。当发现广度优先搜索在空间上不够优秀，而且要找最优解的问题时，就应该考虑迭代加深。

6.8 IDA*

前置知识：A* 算法、迭代加深搜索。

本页面将简要介绍 IDA * 算法。

定义

IDA * 为采用了迭代加深算法的 A * 算法。

优点

由于 IDA * 改成了深度优先的方式，相对于 A * 算法，它的优点如下：

1. 不需要判重，不需要排序，利于深度剪枝。
2. 空间需求减少：每个深度下实际上是一个深度优先搜索，不过深度有限制，使用 DFS 可以减小空间消耗。

缺点

1. 重复搜索：即使前后两次搜索相差微小，回溯过程中每次深度变大都要再次从头搜索。

实现（伪代码）

```
Procedure IDA_STAR(StartState)
Begin
  PathLimit := H(StartState) - 1;
  Succes := False;
  Repeat
    inc(PathLimit);
```

```
    StartState.g = 0;
    Push(OpenStack, StartState);
    Repeat
      CurrentState := Pop(OpenStack);
      If Solution(CurrentState) then
        Success = True
      Elseif PathLimit >= CurrentState.g + H(CurrentState) then
        For each Child(CurrentState) do
          Push(OpenStack, Child(CurrentState));
    until Success or empty(OpenStack);
  until Success or ResourceLimtsReached;
end;
```

例题

"埃及分数[1]"

在古埃及，人们使用单位分数的和（即 $\frac{1}{a}$，$a \in \mathbb{N}^*$）表示一切有理数。例如，$\frac{2}{3}=\frac{1}{2}+\frac{1}{6}$，但不允许 $\frac{2}{3}=\frac{1}{3}+\frac{1}{3}$，因为在加数中不允许有相同的。

对于一个分数 $\frac{a}{b}$，表示方法有很多种，其中加数少的比加数多的好，如果加数个数相同，则最小的分数越大越好。例如，$\frac{19}{45}=\frac{1}{5}+\frac{1}{6}+\frac{1}{18}$ 是最优方案。

输入整数 a,b（$0<a<b<500$），试编程计算最佳表达式。

样例输入：

```
495 499
```

样例输出：

```
Case 1: 495/499=1/2+1/5+1/6+1/8+1/3992+1/14970
```

解题思路

这道题目理论上可以用回溯法求解，但是解答树会非常「恐怖」——不仅深度没有明显的上界，而且加数的选择理论上也是无限的。换句话说，如果用宽度优先遍历，连一层都扩展不完，因为每一层都是**无限大**的。

解决方案是采用迭代加深搜索：从小到大枚举深度上限 $maxd$，每次执行只考虑深度不超过 $maxd$ 的节点。这样，只要解的深度有限，则一定可以在有限时间内枚举到。

深度上限 $maxd$ 还可以用来**剪枝**。按照分母递增的顺序来进行扩展，如果扩展到 i 层时，前 i 个分数之和为 $\frac{c}{d}$，而第 i 个分数为 $\frac{1}{e}$，则接下来至少还需要 $\frac{\frac{a}{b}-\frac{c}{d}}{\frac{1}{e}}$ 个分数，总和才能达到 $\frac{a}{b}$。例如，当前搜索到 $\frac{19}{45}=\frac{1}{5}+\frac{1}{100}+\cdots$，则后面的分数每个最大为 $\frac{1}{101}$，至少需要 $\frac{\frac{19}{45}-\frac{1}{5}}{\frac{1}{101}}=23$ 项总和才能达到 $\frac{19}{45}$，因此前 22 次迭代是根本不会考虑这棵子树的。这里的关键在于：可以估计至少还要多少步才能出解。

注意，这里使用**至少**一词表示估计都是乐观的。形式化地，设深度上限为 $maxd$，当前结点 n 的深度为 $g(n)$，乐观估价函数为 $h(n)$，则当 $g(n)+h(n)>maxd$ 时应该剪枝。这样的算法就是 IDA*。当然，在实战中不需要严格地在代码里写出 $g(n)$ 和 $h(n)$，只需要像刚才那样设计出乐观估价函数，想清楚在什么情况下不可能在当前的深度限制下出解即可。

解题思路

> 如果可以设计出一个乐观估价函数，预测从当前结点至少还需要扩展几层结点才有可能得到解，则迭代加深搜索变成了 IDA * 算法。

示例代码

```
#include <bits/stdc++.h>

using namespace std;

const int MAX_E = 1e7;
int a, b;
vector<int> ans;
vector<int> current;

inline bool better() { return ans.empty() || current.back() < ans.back(); }

bool dfs(int d, long a, long b, int e) {
  if (d == 0) {
    if (a == 0 && better()) ans = current;
    return a == 0;
  }

  long _gcd = gcd(a, b);
  a /= _gcd;
  b /= _gcd;

  bool solved = false;
  // the min value of e:
  // a/b - 1/e >= 0
  // e >= b/a
  int e1 = max(e + 1, int((b + a - 1) / a));
  // b/a <= e <= MAX_E
  // b/a <= MAX_E
  if (b > a * MAX_E) {
    return false;
  }
  for (;; e1++) {
    // the max value of e:
    // d * (1/e) >= a/b
    // d/e >= a/b
    if (d * b < a * e1) {
      return solved;
    }
    current.push_back(e1);
    // a/b - 1/e
    solved |= dfs(d - 1, a * e1 - b, b * e1, e1);
    current.pop_back();
  }
  return solved;
}

int solve() {
  ans.clear();
  current.clear();
  for (int maxd = 1; maxd <= 100; maxd++)
    if (dfs(maxd, a, b, 1)) return maxd;
  return -1;
```

```
}

int main() {
  int kase = 0;
  while (cin >> a >> b) {
    int maxd = solve();
    cout << "Case " << ++kase << ": ";
    if (maxd > 0) {
      cout << a << "/" << b << "=";
      for (int i = 0; i < maxd - 1; i++) cout << "1/" << ans[i] << "+";
      cout << "1/" << ans[maxd - 1] << "\n";
    } else
      cout << "No solution.\n";
  }
  return 0;
}
```

习题

- UVa1343 旋转游戏[2]

参考资料与注释

[1] 埃及分数

[2] UVa1343 旋转游戏

6.9 回溯法

本页面将简要介绍回溯法的概念和应用。

简介

回溯法是一种经常被用在 深度优先搜索 （DFS） 和 广度优先搜索 （BFS） 的技巧。

其本质是：走不通就回头。

过程

1. 构造空间树；
2. 进行遍历；
3. 如遇到边界条件，即不再向下搜索，转而搜索另一条链；
4. 达到目标条件，输出结果。

例题

"USACO 1.5.4 Checker Challenge[1]"

现在有一个如下的 6×6 的跳棋棋盘，有六个棋子被放置在棋盘上，使得每行，每列，每条对角线（包括两

条主对角线的所有对角线）上都至多有一个棋子。

```
0   1   2   3   4   5   6
  -------------------------
1 |   | O |   |   |   |   |
  -------------------------
2 |   |   |   | O |   |   |
  -------------------------
3 |   |   |   |   |   | O |
  -------------------------
4 | O |   |   |   |   |   |
  -------------------------
5 |   |   | O |   |   |   |
  -------------------------
6 |   |   |   |   | O |   |
  -------------------------
```

上面的布局可以用序列 $\{2,4,6,1,3,5\}$ 来描述，第 i 个数字表示在第 i 行的第 a_i 列有一个棋子，如下所示

行号 i：$\{1,2,3,4,5,6\}$

列号 a_i：$\{2,4,6,1,3,5\}$

这只是跳棋放置的一个方案。请编一个程序找出所有方案并把它们以上面的序列化方法输出，按字典顺序排列。你只需输出前 3 个解并在最后一行输出解的总个数。特别注意：你需要优化你的程序以保证在更大棋盘尺寸下的程序效率。

" 参考代码"

```cpp
// 该代码为回溯法的 DFS 实现
#include <cstdio>
int ans[14], check[3][28] = {0}, sum = 0, n;

void eq(int line) {
  if (line > n) {  // 如果已经搜索完 n 行
    sum++;
    if (sum > 3)
      return;
    else {
      for (int i = 1; i <= n; i++) printf("%d ", ans[i]);
      printf("\n");
      return;
    }
  }
  for (int i = 1; i <= n; i++) {
    if ((!check[0][i]) && (!check[1][line + i]) &&
        (!check[2][line - i + n])) {  // 判断在某位置放置是否合法
      ans[line] = i;
      check[0][i] = 1;
      check[1][line + i] = 1;
      check[2][line - i + n] = 1;
      eq(line + 1);
      // 向下递归后进行回溯，方便下一轮递归
      check[0][i] = 0;
      check[1][line + i] = 0;
      check[2][line - i + n] = 0;
```

```
    }
  }
}

int main() {
  scanf("%d", &n);
  eq(1);
  printf("%d", sum);
  return 0;
}
```

"迷宫[2]"

现有一个尺寸为 $N \times M$ 的迷宫，迷宫里有 T 处障碍，障碍处不可通过。给定起点坐标和终点坐标，且每个方格最多经过一次，问有多少种从起点坐标到终点坐标的方案。在迷宫中移动有上、下、左、右四种移动方式，每次只能移动一个方格。数据保证起点上没有障碍。

" 参考代码"

```
// 该代码为回溯法的 BFS 实现
#include <cstdio>
#include <cstring>
#include <queue>
using namespace std;
int n, m, k, x, y, a, b, ans;
int dx[4] = {0, 0, 1, -1}, dy[4] = {1, -1, 0, 0};  // 四个方向
bool vis[6][6];

struct oo {
  int x, y, used[6][6];
};

oo sa;

void bfs() {
  queue<oo> q;
  sa.x = x;
  sa.y = y;
  sa.used[x][y] = 1;
  q.push(sa);
  while (!q.empty()) {  // BFS 队列
    oo now = q.front();
    q.pop();
    for (int i = 0; i < 4; i++) {  // 枚举向四个方向走
      int sx = now.x + dx[i];
      int sy = now.y + dy[i];
      if (now.used[sx][sy] || vis[sx][sy] || sx == 0 || sy == 0 || sx > n ||
          sy > m)
        continue;
      if (sx == a && sy == b) {
        ans++;
        continue;
```

```
      }
      sa.x = sx;
      sa.y = sy;
      memcpy(sa.used, now.used, sizeof(now.used));
      sa.used[sx][sy] = 1;
      q.push(sa);  // 假设向此方向走，放入 BFS 队列
    }
  }
}

int main() {
  scanf("%d%d%d", &n, &m, &k);
  scanf("%d%d%d%d", &x, &y, &a, &b);
  for (int i = 1, aa, bb; i <= k; i++) {
    scanf("%d%d", &aa, &bb);
    vis[aa][bb] = 1;  // 障碍位置不可通过
  }
  bfs();
  printf("%d", ans);
  return 0;
}
```

参考资料与注释

[1] USACO 1.5.4 Checker Challenge

[2] 迷宫

6.10 Dancing Links

Authors: LeverImmy, 383494

本页面将介绍精确覆盖问题、重复覆盖问题，解决这两个问题的算法「X 算法」，以及用来优化 X 算法的双向十字链表 Dancing Link。本页也将介绍如何在建模的配合下使用 DLX 解决一些搜索题。

精确覆盖问题

定义

精确覆盖问题（英文：Exact Cover Problem）是指给定许多集合 $S_i(1 \le i \le n)$ 以及一个集合 X，求满足以下条件的无序多元组 $(T_1, T_2, \cdots, T_m)$：

1. $\forall i, j \in [1, m], T_i \bigcap T_j = \emptyset (i \neq j)$
2. $X = \bigcup\limits_{i=1}^{m} T_i$
3. $\forall i \in [1, m], T_i \in \{S_1, S_2, \cdots, S_n\}$

解释

例如，若给出

$$
\begin{aligned}
S_1 &= \{5,9,17\}\\
S_2 &= \{1,8,119\}\\
S_3 &= \{3,5,17\}\\
S_4 &= \{1,8\}\\
S_5 &= \{3,119\}\\
S_6 &= \{8,9,119\}\\
X &= \{1,3,5,8,9,17,119\}
\end{aligned}
$$

则 (S_1,S_4,S_5) 为一组合法解。

问题转化

将 $\bigcup\limits_{i=1}^{n} S_i$ 中的所有数离散化，可以得到这么一个模型：

> 给定一个 01 矩阵，你可以选择一些行（row），使得最终每列（column）[1] 都恰好有一个 1。举个例子，我们对上文中的例子进行建模，可以得到这么一个矩阵：

$$
\begin{pmatrix}
0 & 0 & 1 & 0 & 1 & 1 & 0\\
1 & 0 & 0 & 1 & 0 & 0 & 1\\
0 & 1 & 1 & 0 & 0 & 1 & 0\\
1 & 0 & 0 & 1 & 0 & 0 & 0\\
0 & 1 & 0 & 0 & 0 & 0 & 1\\
0 & 0 & 0 & 1 & 1 & 0 & 1
\end{pmatrix}
$$

> 其中第 i 行表示着 S_i，而这一行的每个数依次表示 $[1\in S_i],[3\in S_i],[5\in S_i],\cdots,[119\in S_i]$。

实现

暴力 1

一种方法是枚举选择哪些行，最后检查这个方案是否合法。

因为每一行都有选或者不选两种状态，所以枚举行的时间复杂度是 $O(2^n)$ 的；

而每次检查都需要 $O(nm)$ 的时间复杂度。所以总的复杂度是 $O(nm\cdot 2^n)$。

" 实现"

```
int ok = 0;
for (int state = 0; state < 1 << n; ++state) {  // 枚举每行是否被选
  for (int i = 1; i <= n; ++i)
    if ((1 << i - 1) & state)
      for (int j = 1; j <= m; ++j) a[i][j] = 1;
  int flag = 1;
  for (int j = 1; j <= m; ++j)
    for (int i = 1, bo = 0; i <= n; ++i)
      if (a[i][j]) {
        if (bo)
          flag = 0;
```

```
        else
          bo = 1;
      }
  if (!flag)
    continue;
  else {
    ok = 1;
    for (int i = 1; i <= n; ++i)
      if ((1 << i - 1) & state) printf("%d ", i);
    puts("");
  }
  memset(a, 0, sizeof(a));
}
if (!ok) puts("No solution.");
```

暴力 2

考虑到 01 矩阵的特殊性质，每一行都可以看做一个 m 位二进制数。

因此原问题转化为

> 给定 n 个 m 位二进制数，要求选择一些数，使得任意两个数的与都为 0，且所有数的或为 2^m-1。`tmp` 表示的是截至目前被选中的二进制数的或。

因为每一行都有选或者不选两种状态，所以枚举行的时间复杂度为 $O(2^n)$；

而每次计算 `tmp` 都需要 $O(n)$ 的时间复杂度。所以总的复杂度为 $O(n\cdot 2^n)$。

" 实现"

```
int ok = 0;
for (int i = 1; i <= n; ++i)
  for (int j = m; j >= 1; --j) num[i] = num[i] << 1 | a[i][j];
for (int state = 0; state < 1 << n; ++state) {
  int tmp = 0;
  for (int i = 1; i <= n; ++i)
    if ((1 << i - 1) & state) {
      if (tmp & num[i]) break;
      tmp |= num[i];
    }
  if (tmp == (1 << m) - 1) {
    ok = 1;
    for (int i = 1; i <= n; ++i)
      if ((1 << i - 1) & state) printf("%d ", i);
    puts("");
  }
}
if (!ok) puts("No solution.");
```

重复覆盖问题

重复覆盖问题与精确覆盖问题类似，但没有对元素相似性的限制。下文介绍的 X 算法原本针对精确覆盖问题，但经过一些修改和优化（已标注在其中）同样可以高效地解决重复覆盖问题。

X 算法

Donald E. Knuth 提出了 X 算法 (Algorithm X)，其思想与刚才的暴力差不多，但是方便优化。

过程

继续以上文中中提到的例子为载体，得到一个这样的 01 矩阵：

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

1. 此时第一行有 3 个 1，第二行有 3 个 1，第三行有 3 个 1，第四行有 2 个 1，第五行有 2 个 1，第六行有 3 个 1。选择第一行，将它删除，并将所有 1 所在的列打上标记；

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

2. 选择所有被标记的列，将它们删除，并将这些列中含 1 的行打上标记（重复覆盖问题无需打标记）；

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

3. 选择所有被标记的行，将它们删除；

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

   这表示这一行已被选择，且这一行的所有 1 所在的列不能有其他 1 了。

   于是得到一个新的小 01 矩阵：

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

4. 此时第一行（原来的第二行）有 3 个 1，第二行（原来的第四行）有 2 个 1，第三行（原来的第五行）有 2 个 1。选择第一行（原来的第二行），将它删除，并将所有 1 所在的列打上标记；

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

5. 选择所有被标记的列，将它们删除，并将这些列中含 1 的行打上标记；

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

6. 选择所有被标记的行，将它们删除；

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

这样就得到了一个空矩阵。但是上次删除的行 `1 0 1 1` 不是全 1 的，说明选择有误；

$$)\,($$

7. 回溯到步骤 4，考虑选择第二行（原来的第四行），将它删除，并将所有 1 所在的列打上标记；

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

8. 选择所有被标记的列，将它们删除，并将这些列中含 1 的行打上标记；

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

9. 选择所有被标记的行，将它们删除；

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

于是我们得到了这样的一个矩阵：

$$\begin{pmatrix} 1 & 1 \end{pmatrix}$$

10. 此时第一行（原来的第五行）有 2 个 1，将它们全部删除，得到一个空矩阵：

$$)\,($$

11. 上一次删除的时候，删除的是全 1 的行，因此成功，算法结束。
答案即为被删除的三行：$1, 4, 5$。

强烈建议自己模拟一遍矩阵删除、还原与回溯的过程后，再接着阅读下文。

通过上述步骤，可将 X 算法的流程概括如下：

1. 对于现在的矩阵 M，选择并标记一行 r，将 r 添加至 S 中；
2. 如果尝试了所有的 r 却无解，则算法结束，输出无解；
3. 标记与 r 相关的行 r_i 和 c_i（相关的行和列与 X 算法中第 2 步定义相同，下同）；
4. 删除所有标记的行和列，得到新矩阵 M'；
5. 如果 M' 为空，且 r 为全 1，则算法结束，输出被删除的行组成的集合 S；
如果 M' 为空，且 r 不全为 1，则恢复与 r 相关的行 r_i 以及列 c_i，跳转至步骤 1；
如果 M' 不为空，则跳转至步骤 1。

不难看出，X 算法需要大量的「删除行」、「删除列」和「恢复行」、「恢复列」的操作。

一个朴素的想法是，使用一个二维数组存放矩阵，再用四个数组分别存放每一行与之相邻的行编号，每次删除和恢复仅需更新四个数组中的元素。但由于一般问题的矩阵中 0 的数量远多于 1 的数量，这样做的空间复杂度难以接受。

Donald E. Knuth 想到了用双向十字链表来维护这些操作。

而在双向十字链表上不断跳跃的过程被形象地比喻成「跳跃」，因此被用来优化 X 算法的双向十字链表也被称为「Dancing Links」。

Dancing Links 优化的 X 算法

预编译命令

```
#define IT(i, A, x) for (i = A[x]; i != x; i = A[i])
```

定义

双向十字链表中存在四个指针域，分别指向上、下、左、右的元素；且每个元素 i 在整个双向十字链表系中都对应着一个格子，因此还要表示 i 所在的列和所在的行，如图所示：

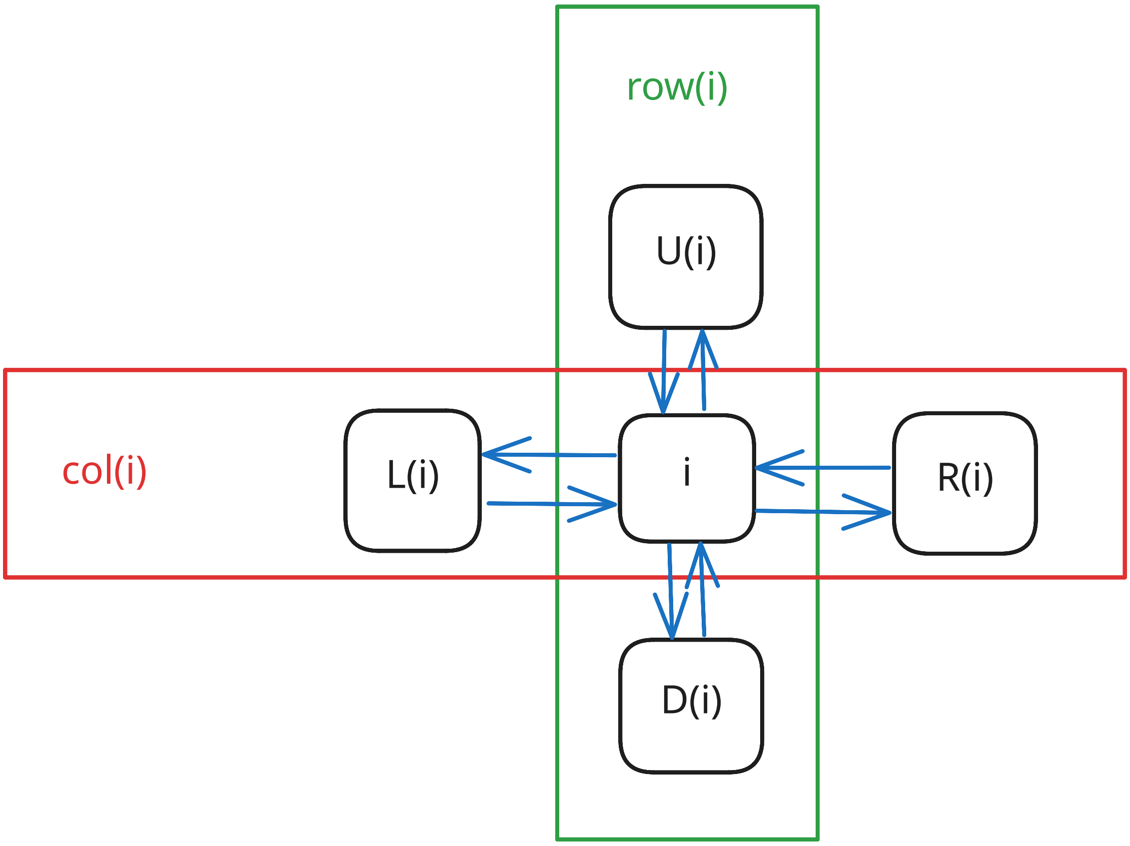


图 6.1 dlx-1.svg

大型的双向链表则更为复杂：

每一行都有一个行首指示，每一列都有一个列指示。

行首指示为 `first[]`，列指示是我们新建的 $c+1$ 个哨兵结点。值得注意的是，**行首指示并非是链表中的哨兵结点**。它是虚拟的，类似于邻接表中的 `first[]` 数组，**直接指向**这一行中的首元素。

同时，每一列都有一个 `siz[]` 表示这一列的元素个数。

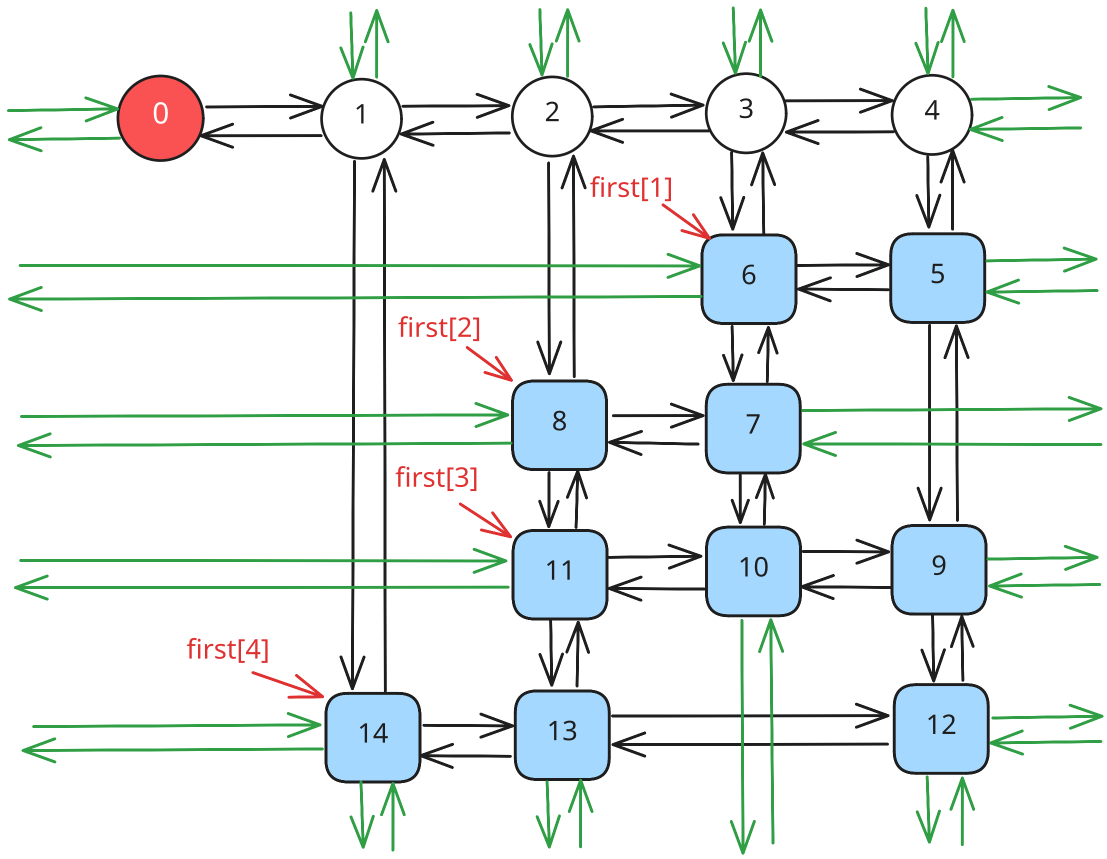


图 6.2 dlx-2.svg

特殊地，0 号结点无右结点等价于这个 Dancing Links 为空。

```
constexpr int MS = 1e5 + 5;
int n, m, idx, first[MS], siz[MS];
int L[MS], R[MS], U[MS], D[MS];
int col[MS], row[MS];
```

过程

remove 操作

`remove(c)` 表示在 Dancing Links 中删除第 c 列以及与其相关的行和列。

先将 c 删除，此时：

- c 左侧的结点的右结点应为 c 的右结点。
- c 右侧的结点的左结点应为 c 的左结点。

即 `L[R[c]] = L[c], R[L[c]] = R[c];`。

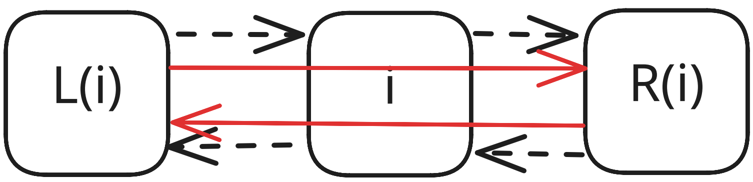


图 6.3 dlx-3.svg

然后顺着这一列往下走，把走过的每一行都删掉。

如何删掉每一行呢？枚举当前行的指针 j，此时：

- j 上方的结点的下结点应为 j 的下结点。
- j 下方的结点的上结点应为 j 的上结点。

注意要修改每一列的元素个数。

即 `U[D[j]] = U[j], D[U[j]] = D[j], --siz[col[j]];`。

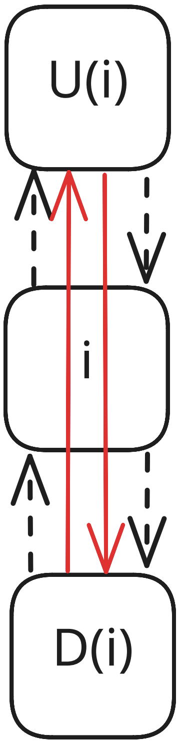


图 6.4 dlx-4.svg

`remove` 函数的代码实现如下：

" 实现"

```
void remove(const int &c) {
  int i, j;
  L[R[c]] = L[c], R[L[c]] = R[c];
  // 顺着这一列从上往下遍历
  IT(i, D, c)
  // 顺着这一行从左往右遍历
  IT(j, R, i)
  U[D[j]] = U[j], D[U[j]] = D[j], --siz[col[j]];
}
```

recover 操作

`recover(c)` 表示在 Dancing Links 中还原第 c 列以及与其相关的行和列。

`recover(c)` 即 `remove(c)` 的逆操作，这里不再赘述。

值得注意的是， `recover(c)` 的所有操作的顺序与 `remove(c)` 的操作恰好相反。

`recover(c)` 的代码实现如下：

" 实现"

```
void recover(const int &c) {
  int i, j;
  IT(i, U, c) IT(j, L, i) U[D[j]] = D[U[j]] = j, ++siz[col[j]];
  L[R[c]] = R[L[c]] = c;
}
```

build 操作

`build(r, c)` 表示新建一个大小为 $r \times c$，即有 r 行，c 列的 Dancing Links。

新建 $c+1$ 个结点作为列指示。

第 i 个点的左结点为 $i-1$，右结点为 $i+1$，上结点为 i，下结点为 i。特殊地，0 结点的左结点为 c，c 结点的右结点为 0。

于是我们得到了一个环状双向链表：

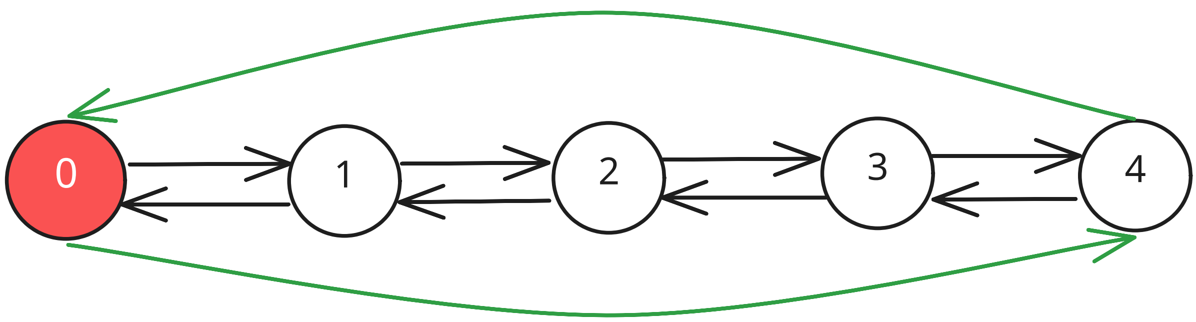


图 6.5 dlx-5.svg

这样就初始化了一个 Dancing Links。

`build(r, c)` 的代码实现如下：

" 实现"

```
void build(const int &r, const int &c) {
  n = r, m = c;
  for (int i = 0; i <= c; ++i) {
    L[i] = i - 1, R[i] = i + 1;
    U[i] = D[i] = i;
  }
  L[0] = c, R[c] = 0, idx = c;
  memset(first, 0, sizeof(first));
  memset(siz, 0, sizeof(siz));
}
```

insert 操作

`insert(r, c)` 表示在第 r 行，第 c 列插入一个结点。

插入操作分为两种情况：

- 如果第 r 行没有元素，那么直接插入一个元素，并使 `first[r]` 指向这个元素。

  这可以通过 `first[r] = L[idx] = R[idx] = idx;` 来实现。
- 如果第 r 行有元素，那么将这个新元素用一种特殊的方式与 c 和 $first(r)$ 连接起来。

设这个新元素为 idx，然后：

- 把 idx 插入到 c 的正下方，此时：
  * idx 下方的结点为原来 c 的下结点；
  * idx 下方的结点（即原来 c 的下结点）的上结点为 idx;
  * idx 的上结点为 c；
  * c 的下结点为 idx。

  注意记录 idx 的所在列和所在行，以及更新这一列的元素个数。

```
col[++idx] = c, row[idx] = r, ++siz[c];
U[idx] = c, D[idx] = D[c], U[D[c]] = idx, D[c] = idx;
```

  强烈建议读者完全掌握这几步的顺序后再继续阅读本文。

- 把 idx 插入到 $first(r)$ 的正右方，此时：
  * idx 右侧的结点为原来 $first(r)$ 的右结点；
  * 原来 $first(r)$ 右侧的结点的左结点为 idx；
  * idx 的左结点为 $first(r)$；
  * $first(r)$ 的右结点为 idx。

```
L[idx] = first[r], R[idx] = R[first[r]];
L[R[first[r]]] = idx, R[first[r]] = idx;
```

  强烈建议读者完全掌握这几步的顺序后再继续阅读本文。

`insert(r, c)` 这个操作可以通过图片来辅助理解：

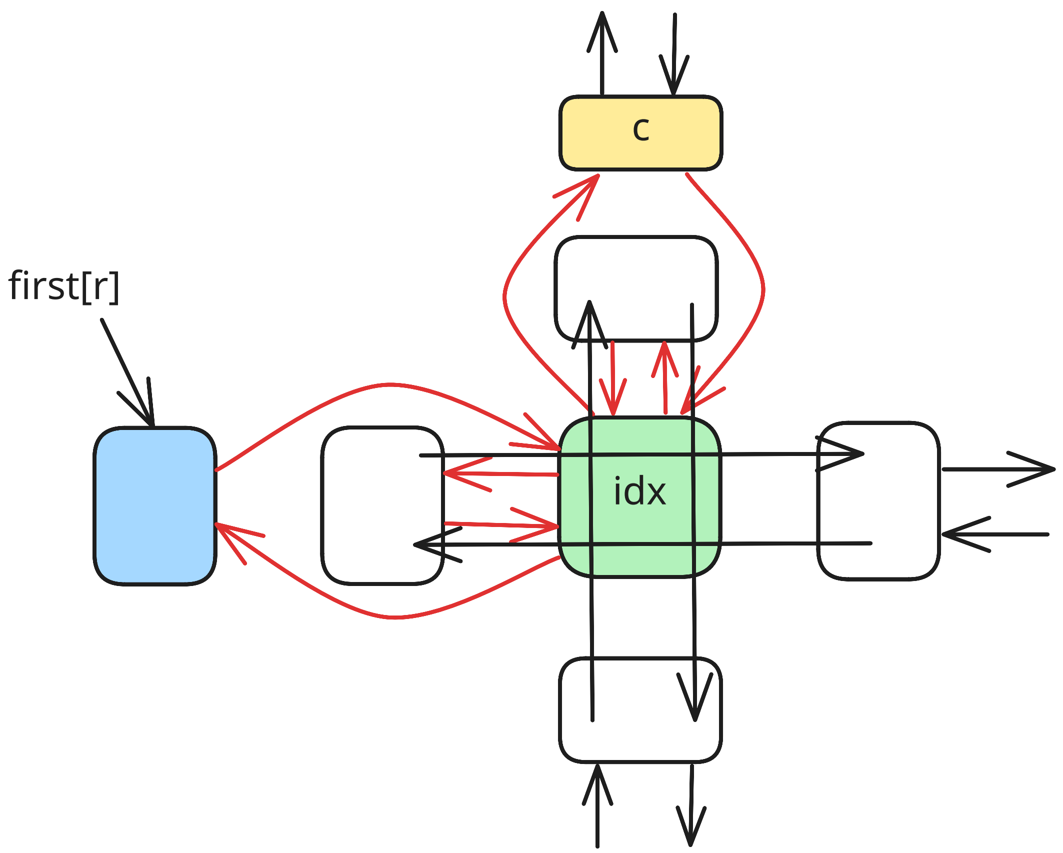


图 6.6 dlx-6.svg

留心曲线箭头的方向。

insert(r, c) 的代码实现如下：

" 实现"

```
void insert(const int &r, const int &c) {
  row[++idx] = r, col[idx] = c, ++siz[c];
  U[idx] = c, D[idx] = D[c], U[D[c]] = idx, D[c] = idx;
  if (!first[r])
    first[r] = L[idx] = R[idx] = idx;
  else {
    L[idx] = first[r], R[idx] = R[first[r]];
    L[R[first[r]]] = idx, R[first[r]] = idx;
  }
}
```

dance 操作

dance() 即为递归地删除以及还原各个行列的过程。

1. 如果 0 号结点没有右结点，那么矩阵为空，记录答案并返回；
2. 选择列元素个数最少的一列，并删掉这一列；
3. 遍历这一列所有有 1 的行，枚举它是否被选择；
4. 递归调用 dance()，如果可行，则返回；如果不可行，则恢复被选择的行；
5. 如果无解，则返回。

dance() 的代码实现如下：

" 实现"

```
bool dance(int dep) {
  int i, j, c = R[0];
  if (!R[0]) {
    ans = dep;
    return 1;
  }
  IT(i, R, 0) if (siz[i] < siz[c]) c = i;
  remove(c);
  IT(i, D, c) {
    stk[dep] = row[i];
    IT(j, R, i) remove(col[j]);
    if (dance(dep + 1)) return 1;
    IT(j, L, i) recover(col[j]);
  }
  recover(c);
  return 0;
}
```

其中 stk[] 用来记录答案。

注意我们每次优先选择列元素个数最少的一列进行删除，这样能保证程序具有一定的启发性，使搜索树分支最少。

对于重复覆盖问题，在搜索时可以用估价函数（与 A* 中类似）进行剪枝：若当前最好情况下所选行数超过目前最优解，则可以直接返回。

模板

"模板代码[2]"

```cpp
#include <bits/stdc++.h>
const int N = 500 + 10;
int n, m, idx, ans;
int first[N], siz[N], stk[N];

int read() {  // 快读
  int x = 0, f = 0, ch;
  while (!isdigit(ch = getchar())) f |= ch == '-';
  while (isdigit(ch)) x = (x << 1) + (x << 3) + (ch ^ 48), ch = getchar();
  return f ? -x : x;
}

struct DLX {
  static const int MAXSIZE = 1e5 + 10;
  int n, m, tot, first[MAXSIZE + 10], siz[MAXSIZE + 10];
  int L[MAXSIZE + 10], R[MAXSIZE + 10], U[MAXSIZE + 10], D[MAXSIZE + 10];
  int col[MAXSIZE + 10], row[MAXSIZE + 10];

  void build(const int &r, const int &c) {  // 进行 build 操作
    n = r, m = c;
    for (int i = 0; i <= c; ++i) {
      L[i] = i - 1, R[i] = i + 1;
      U[i] = D[i] = i;
    }
    L[0] = c, R[c] = 0, tot = c;
    memset(first, 0, sizeof(first));
    memset(siz, 0, sizeof(siz));
  }

  void insert(const int &r, const int &c) {  // 进行 insert 操作
    col[++tot] = c, row[tot] = r, ++siz[c];
    D[tot] = D[c], U[D[c]] = tot, U[tot] = c, D[c] = tot;
    if (!first[r])
      first[r] = L[tot] = R[tot] = tot;
    else {
      R[tot] = R[first[r]], L[R[first[r]]] = tot;
      L[tot] = first[r], R[first[r]] = tot;
    }
  }

  void remove(const int &c) {  // 进行 remove 操作
    int i, j;
    L[R[c]] = L[c], R[L[c]] = R[c];
    for (i = D[c]; i != c; i = D[i])
      for (j = R[i]; j != i; j = R[j])
        U[D[j]] = U[j], D[U[j]] = D[j], --siz[col[j]];
  }

  void recover(const int &c) {  // 进行 recover 操作
    int i, j;
```

```
    for (i = U[c]; i != c; i = U[i])
      for (j = L[i]; j != i; j = L[j]) U[D[j]] = D[U[j]] = j, ++siz[col[j]];
    L[R[c]] = R[L[c]] = c;
  }

  bool dance(int dep) {  // dance
    if (!R[0]) {
      ans = dep;
      return 1;
    }
    int i, j, c = R[0];
    for (i = R[0]; i != 0; i = R[i])
      if (siz[i] < siz[c]) c = i;
    remove(c);
    for (i = D[c]; i != c; i = D[i]) {
      stk[dep] = row[i];
      for (j = R[i]; j != i; j = R[j]) remove(col[j]);
      if (dance(dep + 1)) return 1;
      for (j = L[i]; j != i; j = L[j]) recover(col[j]);
    }
    recover(c);
    return 0;
  }
} solver;

int main() {
  n = read(), m = read();
  solver.build(n, m);
  for (int i = 1; i <= n; ++i)
    for (int j = 1; j <= m; ++j) {
      int x = read();
      if (x) solver.insert(i, j);
    }
  solver.dance(1);
  if (ans)
    for (int i = 1; i < ans; ++i) printf("%d ", stk[i]);
  else
    puts("No Solution!");
  return 0;
}
```

性质

DLX 递归及回溯的次数与矩阵中 1 的个数有关，与矩阵的 r, c 等参数无关。因此，它的时间复杂度是**指数级**的，理论复杂度大概在 $O(c^n)$ 左右，其中 c 为某个非常接近于 1 的常数，n 为矩阵中 1 的个数。

但实际情况下 DLX 表现良好，一般能解决大部分的问题。

建模

DLX 的难点，不全在于链表的建立，而在于建模。

请确保已经完全掌握 DLX 模板后再继续阅读本文。

我们每拿到一个题，应该考虑行和列所表示的意义：

- 行表示决策，因为每行对应着一个集合，也就对应着选/不选；
- 列表示状态，因为第 i 列对应着某个条件 P_i。

对于某一行而言，由于不同的列的值不尽相同，我们**由不同的状态，定义了一个决策**。

例题 1 P1784 数独[3]

" 解题思路"

先考虑决策是什么。

在这一题中，每一个决策可以用形如 (r, c, w) 的有序三元组表示。

注意到「宫」并不是决策的参数，因为它**可以被每个确定的** (r, c) **表示**。

因此有 $9 \times 9 \times 9 = 729$ 行。

再考虑状态是什么。

我们思考一下 (r, c, w) 这个决将会造成什么影响。记 (r, c) 所在的宫为 b。

1. 第 r 行用了一个 w（用 $9 \times 9 = 81$ 列表示）；
2. 第 c 列用了一个 w（用 $9 \times 9 = 81$ 列表示）；
3. 第 b 宫用了一个 w（用 $9 \times 9 = 81$ 列表示）；
4. (r, c) 中填入了一个数（用 $9 \times 9 = 81$ 列表示）。

因此有 $81 \times 4 = 324$ 列，共 $729 \times 4 = 2916$ 个 1。

至此，我们成功地将 9×9 的数独问题转化成了一个**有** 729 **行，**324 **列，共** 2916 **个** 1 的精确覆盖问题。

" 参考代码"

```cpp
#include <bits/stdc++.h>
const int N = 1e6 + 10;
int ans[10][10], stk[N];

int read() {
  int x = 0, f = 0, ch;
  while (!isdigit(ch = getchar())) f |= ch == '-';
  while (isdigit(ch)) x = (x << 1) + (x << 3) + (ch ^ 48), ch = getchar();
  return f ? -x : x;
}  // 快读

struct DLX {
  static const int MAXSIZE = 1e5 + 10;
  int n, m, tot, first[MAXSIZE + 10], siz[MAXSIZE + 10];
  int L[MAXSIZE + 10], R[MAXSIZE + 10], U[MAXSIZE + 10], D[MAXSIZE + 10];
  int col[MAXSIZE + 10], row[MAXSIZE + 10];

  void build(const int &r, const int &c) {  // 进行 build 操作
    n = r, m = c;
    for (int i = 0; i <= c; ++i) {
      L[i] = i - 1, R[i] = i + 1;
      U[i] = D[i] = i;
    }
    L[0] = c, R[c] = 0, tot = c;
    memset(first, 0, sizeof(first));
```

```
    memset(siz, 0, sizeof(siz));
  }

  void insert(const int &r, const int &c) {  // 进行 insert 操作
    col[++tot] = c, row[tot] = r, ++siz[c];
    D[tot] = D[c], U[D[c]] = tot, U[tot] = c, D[c] = tot;
    if (!first[r])
      first[r] = L[tot] = R[tot] = tot;
    else {
      R[tot] = R[first[r]], L[R[first[r]]] = tot;
      L[tot] = first[r], R[first[r]] = tot;
    }
  }

  void remove(const int &c) {  // 进行 remove 操作
    int i, j;
    L[R[c]] = L[c], R[L[c]] = R[c];
    for (i = D[c]; i != c; i = D[i])
      for (j = R[i]; j != i; j = R[j])
        U[D[j]] = U[j], D[U[j]] = D[j], --siz[col[j]];
  }

  void recover(const int &c) {  // 进行 recover 操作
    int i, j;
    for (i = U[c]; i != c; i = U[i])
      for (j = L[i]; j != i; j = L[j]) U[D[j]] = D[U[j]] = j, ++siz[col[j]];
    L[R[c]] = R[L[c]] = c;
  }

  bool dance(int dep) {  // dance
    int i, j, c = R[0];
    if (!R[0]) {
      for (i = 1; i < dep; ++i) {
        int x = (stk[i] - 1) / 9 / 9 + 1;
        int y = (stk[i] - 1) / 9 % 9 + 1;
        int v = (stk[i] - 1) % 9 + 1;
        ans[x][y] = v;
      }
      return 1;
    }
    for (i = R[0]; i != 0; i = R[i])
      if (siz[i] < siz[c]) c = i;
    remove(c);
    for (i = D[c]; i != c; i = D[i]) {
      stk[dep] = row[i];
      for (j = R[i]; j != i; j = R[j]) remove(col[j]);
      if (dance(dep + 1)) return 1;
      for (j = L[i]; j != i; j = L[j]) recover(col[j]);
    }
    recover(c);
    return 0;
  }
} solver;
```

```
int GetId(int row, int col, int num) {
  return (row - 1) * 9 * 9 + (col - 1) * 9 + num;
}

void Insert(int row, int col, int num) {
  int dx = (row - 1) / 3 + 1;
  int dy = (col - 1) / 3 + 1;
  int room = (dx - 1) * 3 + dy;
  int id = GetId(row, col, num);
  int f1 = (row - 1) * 9 + num;           // task 1
  int f2 = 81 + (col - 1) * 9 + num;      // task 2
  int f3 = 81 * 2 + (room - 1) * 9 + num; // task 3
  int f4 = 81 * 3 + (row - 1) * 9 + col;  // task 4
  solver.insert(id, f1);
  solver.insert(id, f2);
  solver.insert(id, f3);
  solver.insert(id, f4);
}

int main() {
  solver.build(729, 324);
  for (int i = 1; i <= 9; ++i)
    for (int j = 1; j <= 9; ++j) {
      ans[i][j] = read();
      for (int v = 1; v <= 9; ++v) {
        if (ans[i][j] && ans[i][j] != v) continue;
        Insert(i, j, v);
      }
    }
  solver.dance(1);
  for (int i = 1; i <= 9; ++i, putchar('\n'))
    for (int j = 1; j <= 9; ++j, putchar(' ')) printf("%d", ans[i][j]);
  return 0;
}
```

例题 2 靶形数独[4]

" 解题思路"

这一题与 数独[3] 的模型构建**一模一样**，主要区别在于答案的更新。

这一题可以开一个权值数组，每次找到一组数独的解时，

每个位置上的数乘上对应的权值计入答案即可。

" 参考代码"

```
#include <bits/stdc++.h>
const int oo = 0x3f3f3f3f;
const int N = 1e5 + 10;
const int e[] = {6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7, 6, 6, 7,  8,
                 8, 8, 8, 8, 7, 6, 6, 7, 8, 9, 9, 9, 8, 7, 6, 6, 7, 8, 9, 10, 9,
                 8, 7, 6, 6, 7, 8, 9, 9, 9, 8, 7, 6, 6, 7, 8, 8, 8, 8, 8, 7,  6,
                 6, 7, 7, 7, 7, 7, 7, 7, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6};
```

```cpp
int ans = -oo, a[10][10], stk[N];

int read() {
  int x = 0, f = 0, ch;
  while (!isdigit(ch = getchar())) f |= ch == '-';
  while (isdigit(ch)) x = (x << 1) + (x << 3) + (ch ^ 48), ch = getchar();
  return f ? -x : x;
}

int GetWeight(int row, int col, int num) {  // 求数乘上对应的权值
  return num * e[(row - 1) * 9 + (col - 1)];
}

struct DLX {
  static const int MAXSIZE = 1e5 + 10;
  int n, m, tot, first[MAXSIZE + 10], siz[MAXSIZE + 10];
  int L[MAXSIZE + 10], R[MAXSIZE + 10], U[MAXSIZE + 10], D[MAXSIZE + 10];
  int col[MAXSIZE + 10], row[MAXSIZE + 10];

  void build(const int &r, const int &c) {  // 进行 build 操作
    n = r, m = c;
    for (int i = 0; i <= c; ++i) {
      L[i] = i - 1, R[i] = i + 1;
      U[i] = D[i] = i;
    }
    L[0] = c, R[c] = 0, tot = c;
    memset(first, 0, sizeof(first));
    memset(siz, 0, sizeof(siz));
  }

  void insert(const int &r, const int &c) {  // 进行 insert 操作
    col[++tot] = c, row[tot] = r, ++siz[c];
    D[tot] = D[c], U[D[c]] = tot, U[tot] = c, D[c] = tot;
    if (!first[r])
      first[r] = L[tot] = R[tot] = tot;
    else {
      R[tot] = R[first[r]], L[R[first[r]]] = tot;
      L[tot] = first[r], R[first[r]] = tot;
    }
  }

  void remove(const int &c) {  // 进行 remove 操作
    int i, j;
    L[R[c]] = L[c], R[L[c]] = R[c];
    for (i = D[c]; i != c; i = D[i])
      for (j = R[i]; j != i; j = R[j])
        U[D[j]] = U[j], D[U[j]] = D[j], --siz[col[j]];
  }

  void recover(const int &c) {  // 进行 recover 操作
    int i, j;
    for (i = U[c]; i != c; i = U[i])
      for (j = L[i]; j != i; j = L[j]) U[D[j]] = D[U[j]] = j, ++siz[col[j]];
    L[R[c]] = R[L[c]] = c;
```

```
  }

  void dance(int dep) {  // dance
    int i, j, c = R[0];
    if (!R[0]) {
      int cur_ans = 0;
      for (i = 1; i < dep; ++i) {
        int cur_row = (stk[i] - 1) / 9 / 9 + 1;
        int cur_col = (stk[i] - 1) / 9 % 9 + 1;
        int cur_num = (stk[i] - 1) % 9 + 1;
        cur_ans += GetWeight(cur_row, cur_col, cur_num);
      }
      ans = std::max(ans, cur_ans);
      return;
    }
    for (i = R[0]; i != 0; i = R[i])
      if (siz[i] < siz[c]) c = i;
    remove(c);
    for (i = D[c]; i != c; i = D[i]) {
      stk[dep] = row[i];
      for (j = R[i]; j != i; j = R[j]) remove(col[j]);
      dance(dep + 1);
      for (j = L[i]; j != i; j = L[j]) recover(col[j]);
    }
    recover(c);
  }
} solver;

int GetId(int row, int col, int num) {
  return (row - 1) * 9 * 9 + (col - 1) * 9 + num;
}

void Insert(int row, int col, int num) {
  int dx = (row - 1) / 3 + 1;    // r
  int dy = (col - 1) / 3 + 1;    // c
  int room = (dx - 1) * 3 + dy;  // room
  int id = GetId(row, col, num);
  int f1 = (row - 1) * 9 + num;             // task 1
  int f2 = 81 + (col - 1) * 9 + num;        // task 2
  int f3 = 81 * 2 + (room - 1) * 9 + num;   // task 3
  int f4 = 81 * 3 + (row - 1) * 9 + col;    // task 4
  solver.insert(id, f1);
  solver.insert(id, f2);
  solver.insert(id, f3);
  solver.insert(id, f4);
}

int main() {
  solver.build(729, 324);
  for (int i = 1; i <= 9; ++i)
    for (int j = 1; j <= 9; ++j) {
      a[i][j] = read();
      for (int v = 1; v <= 9; ++v) {
        if (a[i][j] && v != a[i][j]) continue;
```

```
        Insert(i, j, v);
      }
    }
  solver.dance(1);
  printf("%d", ans == -oo ? -1 : ans);
  return 0;
}
```

例题 3 「NOI2005」 智慧珠游戏[5]

" 解题思路"

定义：题中给我们的智慧珠的形态，称为这个智慧珠的标准形态。

显然，我们可以通过改变两个参数 d（表示顺时针旋转 90° 的次数）和 f（是否水平翻转）来改变这个智慧珠的形态。

仍然，我们先考虑决策是什么。

在这一题中，每一个决策可以用形如 (v, d, f, i) 的有序五元组表示。

表示第 i 个智慧珠的标准形态的左上角的位置，序号为 v，经过了 d 次顺时针转 90°。

巧合的是，我们可以令 $f = 1$ 时不水平翻转，$f = -1$ 时水平翻转，从而达到简化代码的目的。

因此有 $55 \times 4 \times 2 \times 12 = 5280$ 行。

需要注意的是，因为一些不合法的填充，如 $(1, 0, 1, 4)$，

所以**在实际操作中，空的智慧珠棋盘也只需要建出 2730 行**。

再考虑状态是什么。

这一题的状态比较简单。

我们思考一下，(v, d, f, i) 这个决策会造成什么影响。

1. 某些格子被占了（用 55 列表示）；
2. 第 i 个智慧珠被用了（用 12 列表示）。

因此有 $55 + 12 = 67$ 列，共 $5280 \times (5 + 1) = 31680$ 个 1。

至此，我们成功地将智慧珠游戏转化成了一个**有 5280 行，67 列，共 31680 个** 1 的精确覆盖问题。

" 参考代码"

```
#include <bits/stdc++.h>
int numcol, numrow;
int dfn[3000], tx[2], nxt[2], num[50][50], vis[50];
char ans[50][50];
const int f[2] = {-1, 1};
const int table[12][5][2] = {
    // directions of shapes
    {{0, 0}, {1, 0}, {0, 1}},                     // A
    {{0, 0}, {0, 1}, {0, 2}, {0, 3}},             // B
    {{0, 0}, {1, 0}, {0, 1}, {0, 2}},             // C
    {{0, 0}, {1, 0}, {0, 1}, {1, 1}},             // D
    {{0, 0}, {1, 0}, {2, 0}, {2, 1}, {2, 2}},     // E
    {{0, 0}, {0, 1}, {1, 1}, {0, 2}, {0, 3}},     // F
    {{0, 0}, {1, 0}, {0, 1}, {0, 2}, {1, 2}},     // G
    {{0, 0}, {1, 0}, {0, 1}, {1, 1}, {0, 2}},     // H
```

```
    {{0, 0}, {0, 1}, {0, 2}, {1, 2}, {1, 3}},   // I
    {{0, 0}, {-1, 1}, {0, 1}, {1, 1}, {0, 2}},  // J
    {{0, 0}, {1, 0}, {1, 1}, {2, 1}, {2, 2}},   // K
    {{0, 0}, {1, 0}, {0, 1}, {0, 2}, {0, 3}},   // L
};
const int len[12] = {3, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5};
const int getx[] = {0,  1,  2,  2,  3,  3,  3,  4,  4,  4,  4,  5,  5,  5,  5,
                    5,  6,  6,  6,  6,  6,  6,  7,  7,  7,  7,  7,  7,  7,  8,
                    8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9,  9,  9,  9,  9,
                    9,  10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11,
                    11, 11, 11, 11, 11, 11, 11, 12, 12, 12, 12, 12, 12, 12, 12,
                    12, 12, 12, 12, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13,
                    13, 13, 14, 14, 14, 14, 14, 14, 14, 14, 14};
const int gety[] = {0, 1, 1, 2,  1,  2,  3,  1, 2,  3,  4,  1, 2, 3, 4,  5,  1,
                    2, 3, 4, 5,  6,  1,  2,  3, 4,  5,  6,  7, 1, 2, 3,  4,  5,
                    6, 7, 8, 1,  2,  3,  4,  5, 6,  7,  8,  9, 1, 2, 3,  4,  5,
                    6, 7, 8, 9,  10, 1,  2,  3, 4,  5,  6,  7, 8, 9, 10, 11, 1,
                    2, 3, 4, 5,  6,  7,  8,  9, 10, 11, 12, 1, 2, 3, 4,  5,  6,
                    7, 8, 9, 10, 11, 12, 13, 1, 2,  3,  4,  5, 6, 7, 8,  9};

struct DLX {
  static const int MS = 1e5 + 10;
  int n, m, tot, first[MS], siz[MS];
  int L[MS], R[MS], U[MS], D[MS];
  int col[MS], row[MS];

  void build(const int &r, const int &c) {
    n = r, m = c;
    for (int i = 0; i <= c; ++i) {
      L[i] = i - 1, R[i] = i + 1;
      U[i] = D[i] = i;
    }
    L[0] = c, R[c] = 0, tot = c;
    memset(first, 0, sizeof(first));
    memset(siz, 0, sizeof(siz));
  }

  void insert(const int &r, const int &c) {  // insert
    col[++tot] = c, row[tot] = r, ++siz[c];
    D[tot] = D[c], U[D[c]] = tot, U[tot] = c, D[c] = tot;
    if (!first[r])
      first[r] = L[tot] = R[tot] = tot;
    else
      R[tot] = R[first[r]], L[R[first[r]]] = tot, L[tot] = first[r],
      R[first[r]] = tot;  // !
  }

  void remove(const int &c) {  // remove
    int i, j;
    L[R[c]] = L[c], R[L[c]] = R[c];
    for (i = D[c]; i != c; i = D[i])
      for (j = R[i]; j != i; j = R[j])
        U[D[j]] = U[j], D[U[j]] = D[j], --siz[col[j]];
  }
```

```
  void recover(const int &c) {  // recover
    int i, j;
    for (i = U[c]; i != c; i = U[i])
      for (j = L[i]; j != i; j = L[j]) U[D[j]] = D[U[j]] = j, ++siz[col[j]];
    L[R[c]] = R[L[c]] = c;
  }

  bool dance() {  // dance
    if (!R[0]) return 1;
    int i, j, c = R[0];
    for (i = R[0]; i != 0; i = R[i])
      if (siz[i] < siz[c]) c = i;
    remove(c);
    for (i = D[c]; i != c; i = D[i]) {
      if (col[i] <= 55) ans[getx[col[i]]][gety[col[i]]] = dfn[row[i]] + 'A';
      for (j = R[i]; j != i; j = R[j]) {
        remove(col[j]);
        if (col[j] <= 55) ans[getx[col[j]]][gety[col[j]]] = dfn[row[j]] + 'A';
      }
      if (dance()) return 1;
      for (j = L[i]; j != i; j = L[j]) recover(col[j]);
    }
    recover(c);
    return 0;
  }
} solver;

int main() {
  for (int i = 1; i <= 10; ++i) scanf("%s", ans[i] + 1);
  for (int i = 1; i <= 10; ++i)
    for (int j = 1; j <= i; ++j) {
      if (ans[i][j] != '.') vis[ans[i][j] - 'A'] = 1;
      num[i][j] = ++numcol;
    }
  solver.build(2730, numcol + 12);
  /*******build*******/
  for (int id = 0, op; id < 12; ++id) {  // every block
    for (++numcol, op = 0; op <= 1; ++op) {
      for (int dx = 0; dx <= 1; ++dx) {
        for (int dy = 0; dy <= 1; ++dy) {
          for (tx[0] = 1; tx[0] <= 10; ++tx[0]) {
            for (tx[1] = 1; tx[1] <= tx[0]; ++tx[1]) {
              bool flag = 1;
              for (int k = 0; k < len[id]; ++k) {
                nxt[op] = tx[op] + f[dx] * table[id][k][0];
                nxt[op ^ 1] = tx[op ^ 1] + f[dy] * table[id][k][1];
                if (vis[id]) {
                  if (ans[nxt[0]][nxt[1]] != id + 'A') {
                    flag = 0;
                    break;
                  }
                } else if (ans[nxt[0]][nxt[1]] != '.') {
                  flag = 0;
```

```
                  break;
                }
              }
              if (!flag) continue;
              dfn[++numrow] = id;
              solver.insert(numrow, numcol);
              for (int k = 0; k < len[id]; ++k) {
                nxt[op] = tx[op] + f[dx] * table[id][k][0];
                nxt[op ^ 1] = tx[op ^ 1] + f[dy] * table[id][k][1];
                solver.insert(numrow, num[nxt[0]][nxt[1]]);
              }
            }
          }
        }
      }
    }
  }
  /********end********/
  if (!solver.dance())
    puts("No solution");
  else
    for (int i = 1; i <= 10; ++i, puts(""))
      for (int j = 1; j <= i; ++j) putchar(ans[i][j]);
  return 0;
}
```

习题

- SUDOKU - Sudoku[6]
- 「kuangbin 带你飞」 专题三 Dancing Links[7]

外部链接

- 夜深人静写算法 （九） - Dancing Links X（跳舞链）_WhereIsHeroFrom 的博客》[8]
- 跳跃的舞者，舞蹈链（Dancing Links）算法 —— 求解精确覆盖问题 - 万仓一黍[9]
- DLX 算法一览 - zhangjianjunab[10]
- 搜索：DLX 算法 - 静听风吟。[11]
- 《算法竞赛入门经典 - 训练指南》[12]

注释

[1] （两岸用语差异）台灣：直行（column）、F列（row）

[2] 模板代码

[3] P1784 数独 [3-1] [3-2]

[4] 靶形数独

[5] 「NOI2005」智慧珠游戏

[6] SUDOKU - Sudoku

[7] 「kuangbin 带你飞」专题三 Dancing Links

[8] 夜深人静写算法（九）- Dancing Links X（跳舞链）_WhereIsHeroFrom 的博客»

[9] 跳跃的舞者，舞蹈链（Dancing Links）算法——求解精确覆盖问题 - 万仓一黍

[10] DLX 算法一览 - zhangjianjunab

[11] 搜索：DLX 算法 - 静听风吟。

[12] 《算法竞赛入门经典 - 训练指南》

6.11 Alpha-Beta 剪枝

此页面将简要介绍 minimax 算法和 $\alpha - \beta$ 剪枝。

Minimax 算法

定义

Minimax 算法又叫极小化极大算法，是一种找出失败的最大可能性中的最小值的算法。[1]

在局面确定的双人对弈里，常进行对抗搜索，构建一棵每个节点都为一个确定状态的搜索树。奇数层为己方先手，偶数层为对方先手。搜索树上每个叶子节点都会被赋予一个估值，估值越大代表我方赢面越大。我方追求更大的赢面，而对方会设法降低我方的赢面，体现在搜索树上就是，奇数层节点（我方节点）总是会选择赢面最大的子节点状态，而偶数层（对方节点）总是会选择我方赢面最小的的子节点状态。

过程

Minimax 算法的整个过程，会从上到下遍历搜索树，回溯时利用子树信息更新答案，最后得到根节点的值，意义就是我方在双方都采取最优策略下能获得的最大分数。

解释

来看一个简单的例子。

称我方为 MAX，对方为 MIN，图示如下：

图 6.7

例如，对于如下的局势，假设从左往右搜索，根节点的数值为我方赢面：

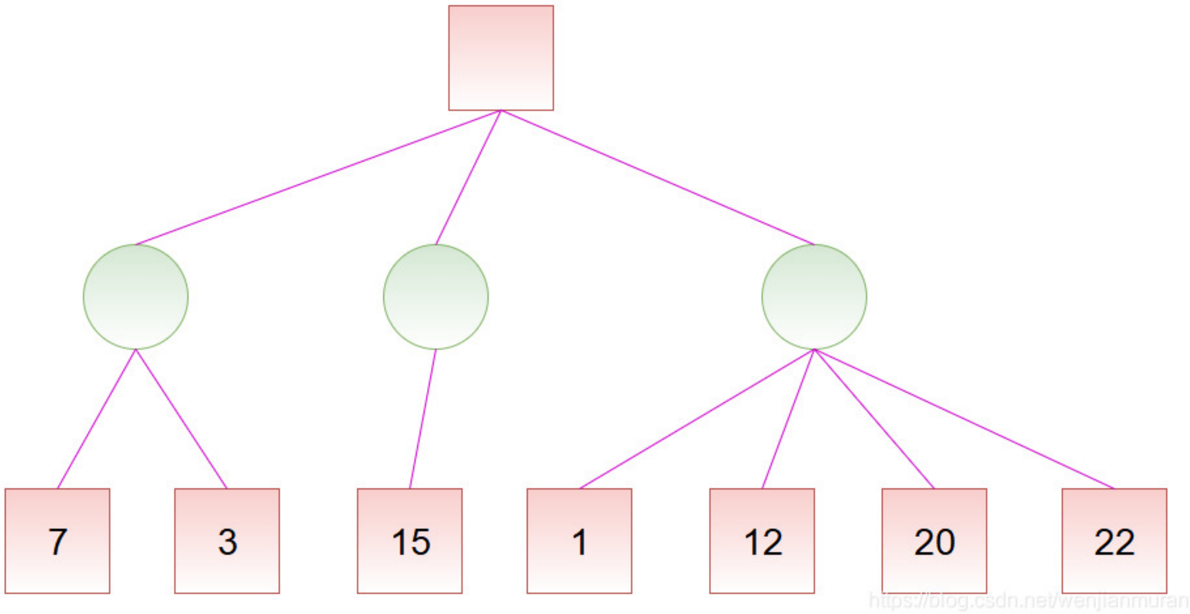


图 6.8

我方应选择中间的路线。因为，如果选择左边的路线，最差的赢面是 3；如果选择中间的路线，最差的赢面是 15；如果选择右边的路线，最差的赢面是 1。虽然选择右边的路线可能有 22 的赢面，但对方也可能使我方只有 1 的赢面，假设对方会选择使得我方赢面最小的方向走，那么经过权衡，显然选择中间的路线更为稳妥。

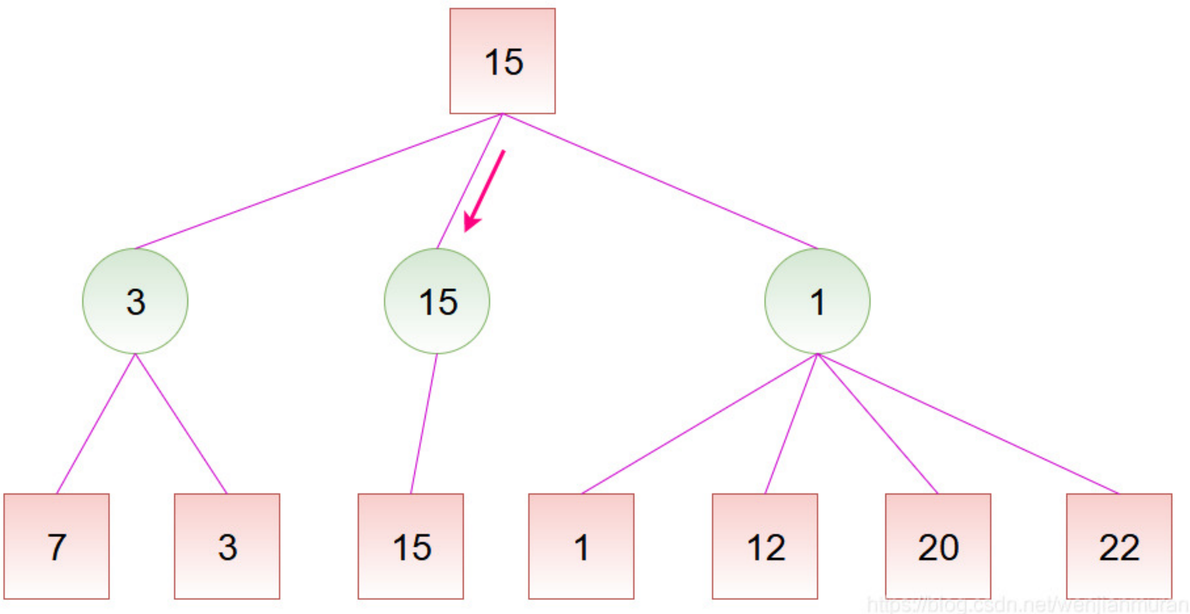


图 6.9

实际上，在看右边的路线时，当发现赢面可能为 1 就不必再去看赢面为 12、20、22 的分支了，因为已经可以确定右边的路线不是最好的。

朴素的 Minimax 算法常常需要构建一棵庞大的搜索树，时间和空间复杂度都将不能承受。而 $\alpha-\beta$ 剪枝就是利用搜索树每个节点取值的上下界来对 Minimax 进行剪枝优化的一种方法。

需要注意的是，对于不同的问题，搜索树每个节点上的值有着不同的含义，它可以是估值、分数、赢的概率等等，为方便起见，我们下面统一用分数来称呼。

alpha-beta 剪枝

过程

对于如下的局势，假设从左往右搜索：

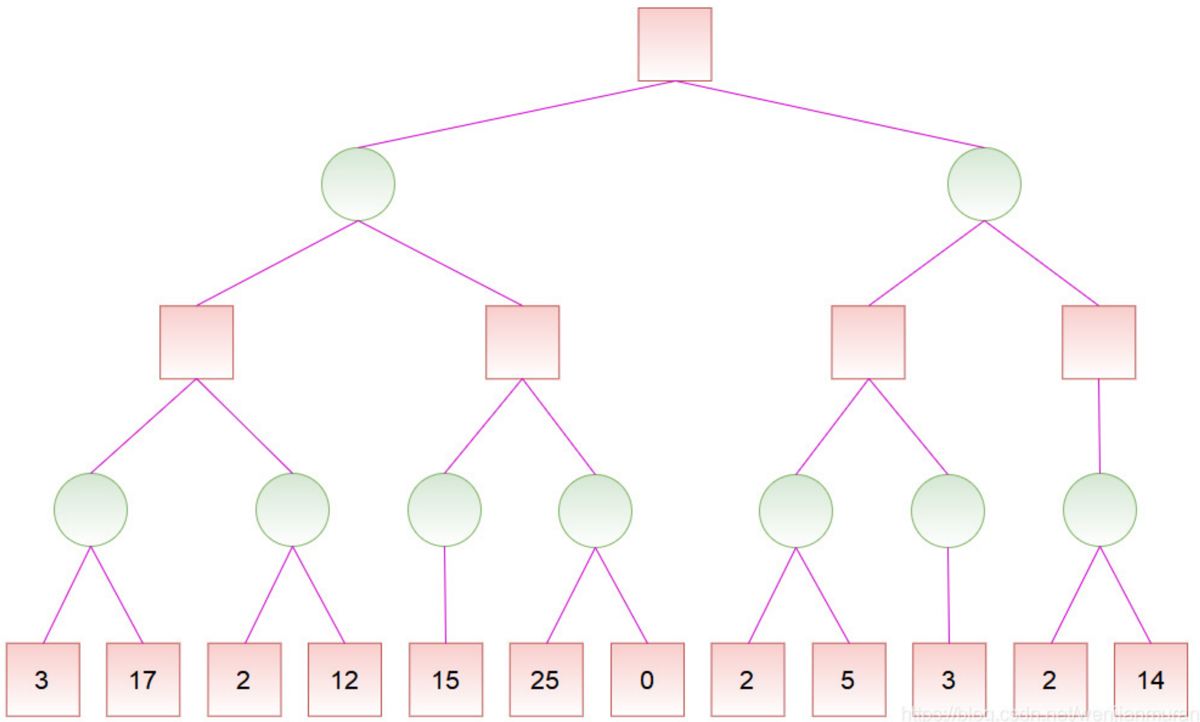


图 6.10

若已知某节点的所有子节点的分数，则可以算出该节点的分数：对于 MAX 节点，取最大分数；对于 MIN 节点，取最小分数。

若已知某节点的部分子节点的分数，虽然不能算出该节点的分数，但可以算出该节点的分数的取值范围。同时，利用该节点的分数的取值范围，在搜索其子节点时，如果已经确定没有更好的走法，就不必再搜索剩余的子节点了。

记 v 为节点的分数，且 $\alpha \le v \le \beta$，即 α 为最大下界，β 为最小上界。当 $\alpha \ge \beta$ 时，该节点剩余的分支就不必继续搜索了（也就是可以进行剪枝了）。注意，当 $\alpha = \beta$ 时，也可以剪枝，这是因为不会有更好的结果了，但可能有更差的结果。

初始化时，令 $\alpha = -\infty, \beta = +\infty$，也就是 $-\infty \le v \le +\infty$。到节点 A 时，由于左子节点的分数为 3，而节点 A 是 MIN 节点，试图找分数小的走法，于是将 β 值修改为 3，这是因为 3 小于当前的 β 值（$\beta = +\infty$）。然后节点 A 的右子节点的分数为 17，此时不修改节点 A 的 β 值，这是因为 17 大于当前的 β 值（$\beta = 3$）。之后，节点 A 的所有子节点搜索完毕，即可计算出节点 A 的分数为 3。

节点 A 是节点 B 的子节点，计算出节点 A 的分数后，可以更新节点 B 的分数范围。由于节点 B 是 MAX 节点，试图找分数大的走法，于是将 α 值修改为 3，这是因为 3 大于当前的 α 值（$\alpha = -\infty$）。之后搜索节点 B 的右子节点 C，并将节点 B 的 α 和 β 值传递给节点 C。

对于节点 C，由于左子节点的分数为 2，而节点 C 是 MIN 节点，于是将 β 值修改为 2。此时 $\alpha \ge \beta$，故节点 C 的剩余子节点就不必搜索了，因为可以确定，通过节点 C 并没有更好的走法。然后，节点 C 是 MIN 节点，将节点 C 的分数设为 β，也就是 2。由于节点 B 的所有子节点搜索完毕，即可计算出节点 B 的分数为 3。

计算出节点 B 的分数后，节点 B 是节点 D 的一个子节点，故可以更新节点 D 的分数范围。由于节点 D 是 MIN 节点，于是将 β 值修改为 3。然后节点 D 将 α 和 β 值传递给节点 E，节点 E 又传递给节点 F。对于节点 F，它只有一个分数为 15 的子节点，由于 15 大于当前的 β 值，而节点 F 为 MIN 节点，所以不更新其 β 值，然后可以计算出节点 F 的分数为 15。

计算出节点 F 的分数后，节点 F 是节点 E 的一个子节点，故可以更新节点 E 的分数范围。节点 E 是 MAX 节

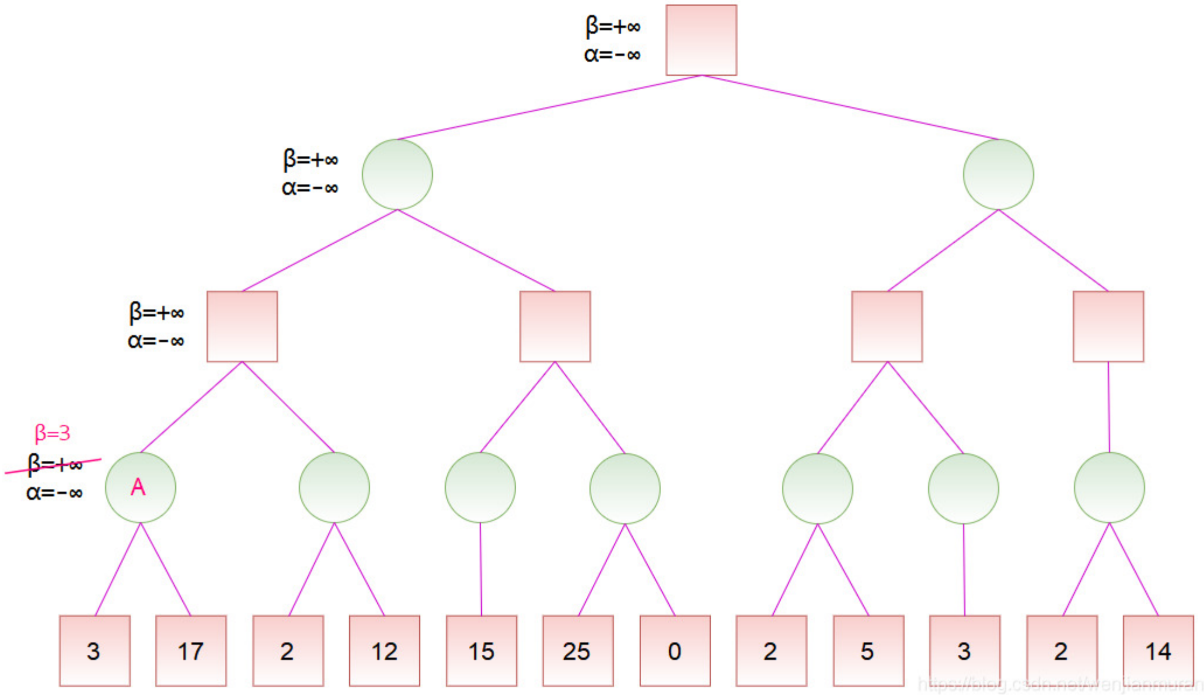

β=+∞
α=-∞
β=+∞
α=-∞
β=+∞
α=-∞
β=3
β=+∞
α=-∞
A
3
17
2
12
15
25
0
2
5
3
2
14
https://blog.csdn.net/wenjianmuran

图 6.11

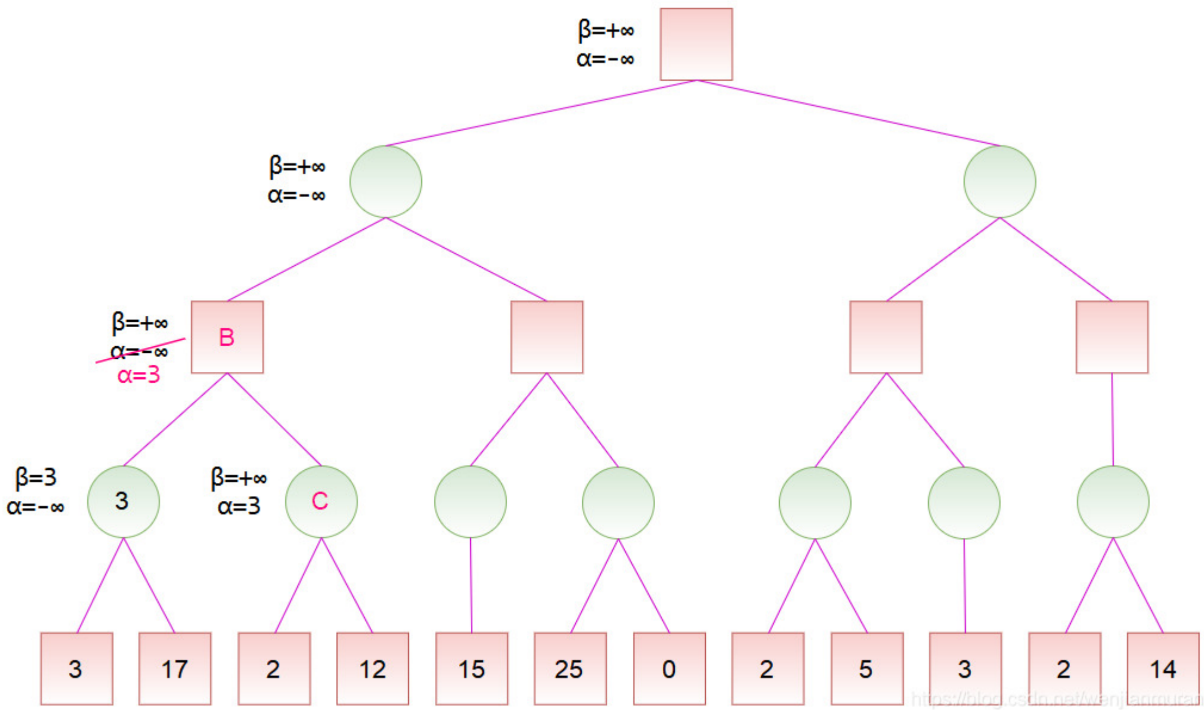

β=+∞
α=-∞
β=+∞
α=-∞
β=+∞
α=-∞
α=3
B
β=3
α=-∞
3
β=+∞
α=3
C
3
17
2
12
15
25
0
2
5
3
2
14
https://blog.csdn.net/wenjianmuran

图 6.12

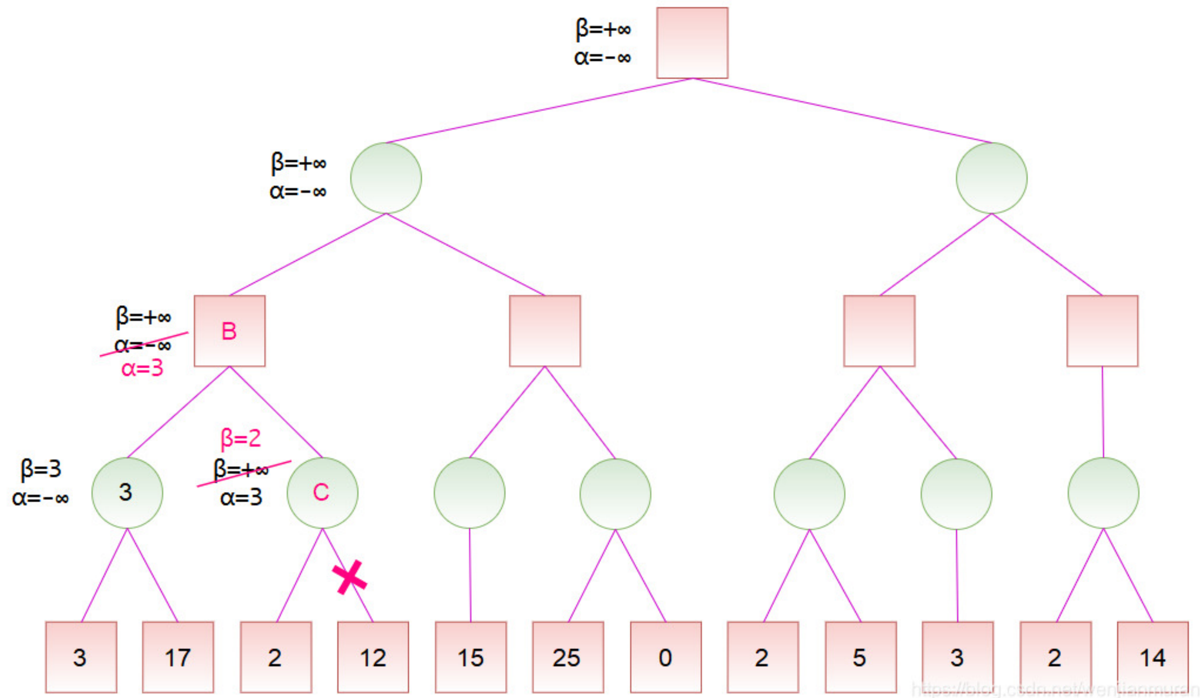

β=+∞
α=-∞
β=+∞
α=-∞
β=+∞
~~α=-∞~~
α=3
B
β=3
α=-∞
3
β=2
~~β=+∞~~
α=3
C
3
17
2
12
15
25
0
2
5
3
2
14

图 6.13

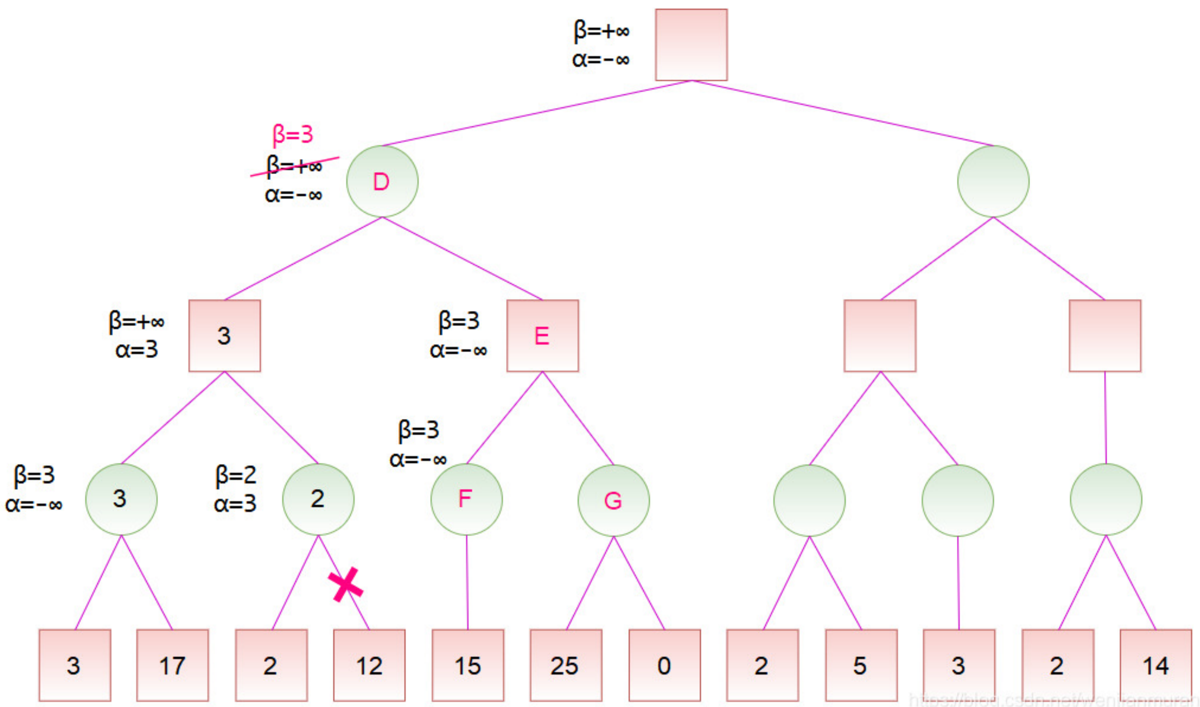

β=+∞
α=-∞
β=3
~~β=+∞~~
α=-∞
D
β=+∞
α=3
3
β=3
α=-∞
E
β=3
α=-∞
3
β=2
α=3
2
β=3
α=-∞
F
G
3
17
2
12
15
25
0
2
5
3
2
14

图 6.14

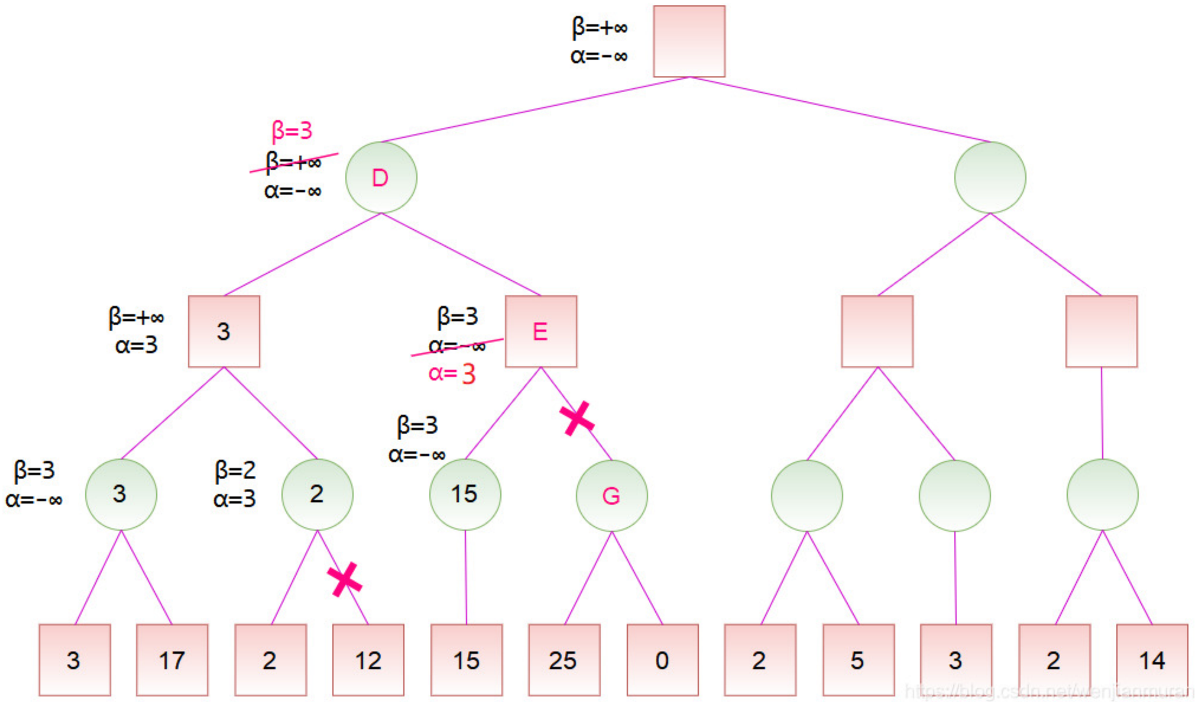


图 6.15

点，更新 α，此时 $\alpha \geq \beta$，故可以剪去节点 E 的余下分支。然后，节点 E 是 MAX 节点，将节点 E 的分数设为 α，也就是 15。此时，节点 D 的所有子节点搜索完毕，即可计算出节点 D 的分数为 3。

计算出节点 D 的分数后，节点 D 是节点 H 的一个子节点，故可以更新节点 H 的分数范围。节点 H 是 MAX 节点，更新 α。然后，按搜索顺序，将节点 H 的 α 和 β 值依次传递给节点 I、J、K。对于节点 K，其左子节点的分数为 2，而节点 K 是 MIN 节点，更新 β，此时 $\alpha \geq \beta$，故可以剪去节点 K 的余下分支。然后，将节点 K 的分数设为 2。

计算出节点 K 的分数后，节点 K 是节点 J 的一个子节点，故可以更新节点 J 的分数范围。节点 J 是 MAX 节点，更新 α，但是，由于节点 K 的分数小于 α，所以节点 J 的 α 值维持 3 保持不变。然后，将节点 J 的 α 和 β 值传递给节点 L。由于节点 L 是 MIN 节点，更新 $\beta = 3$，此时 $\alpha \geq \beta$，故可以剪去节点 L 的余下分支，由于节点 L 没有余下分支，所以此处并没有实际剪枝。然后，将节点 L 的分数设为 3。

实现

" 参考代码"

```
int alpha_beta(int u, int alph, int beta, bool is_max) {
  if (!son_num[u]) return val[u];
  if (is_max) {
    for (int i = 0; i < son_num[u]; ++i) {
      int d = son[u][i];
      alph = max(alph, alpha_beta(d, alph, beta, is_max ^ 1));
      if (alph >= beta) break;
    }
    return alph;
  } else {
    for (int i = 0; i < son_num[u]; ++i) {
      int d = son[u][i];
      beta = min(beta, alpha_beta(d, alph, beta, is_max ^ 1));
      if (alph >= beta) break;
```

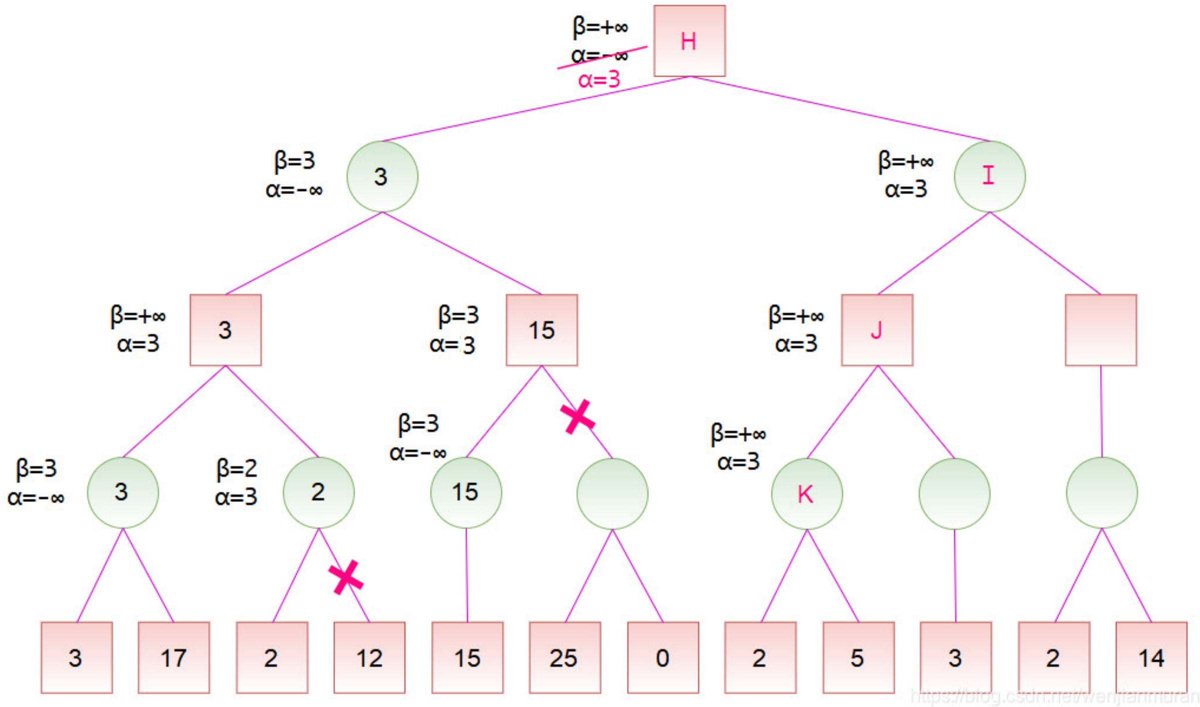

β=+∞
~~α=-∞~~
α=3
H
β=3
α=-∞
3
β=+∞
α=3
I
β=+∞
α=3
3
β=3
α= 3
15
β=+∞
α=3
J
β=3
α=-∞
3
β=2
α=3
2
β=3
α=-∞
15
β=+∞
α=3
K
3
17
2
12
15
25
0
2
5
3
2
14

图 6.16

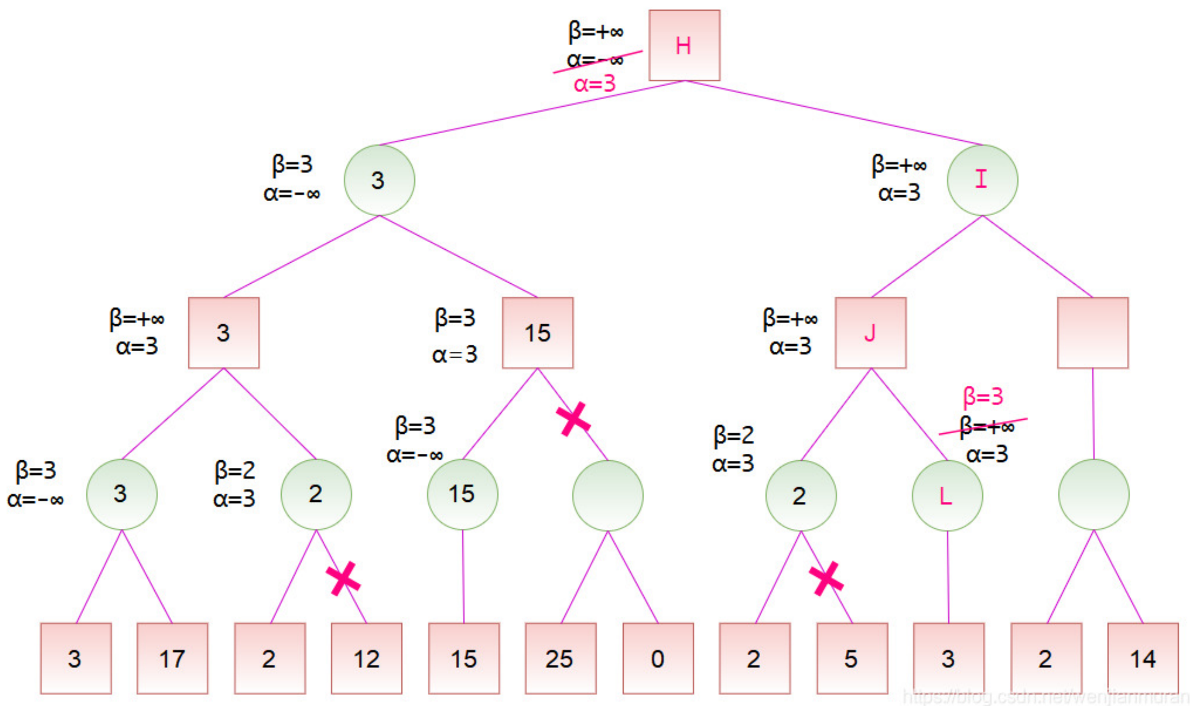

β=+∞
~~α=-∞~~
α=3
H
β=3
α=-∞
3
β=+∞
α=3
I
β=+∞
α=3
3
β=3
α=3
15
β=+∞
α=3
J
β=3
~~β=+∞~~
α=3
β=3
α=-∞
3
β=2
α=3
2
β=3
α=-∞
15
β=2
α=3
2
L
3
17
2
12
15
25
0
2
5
3
2
14

图 6.17

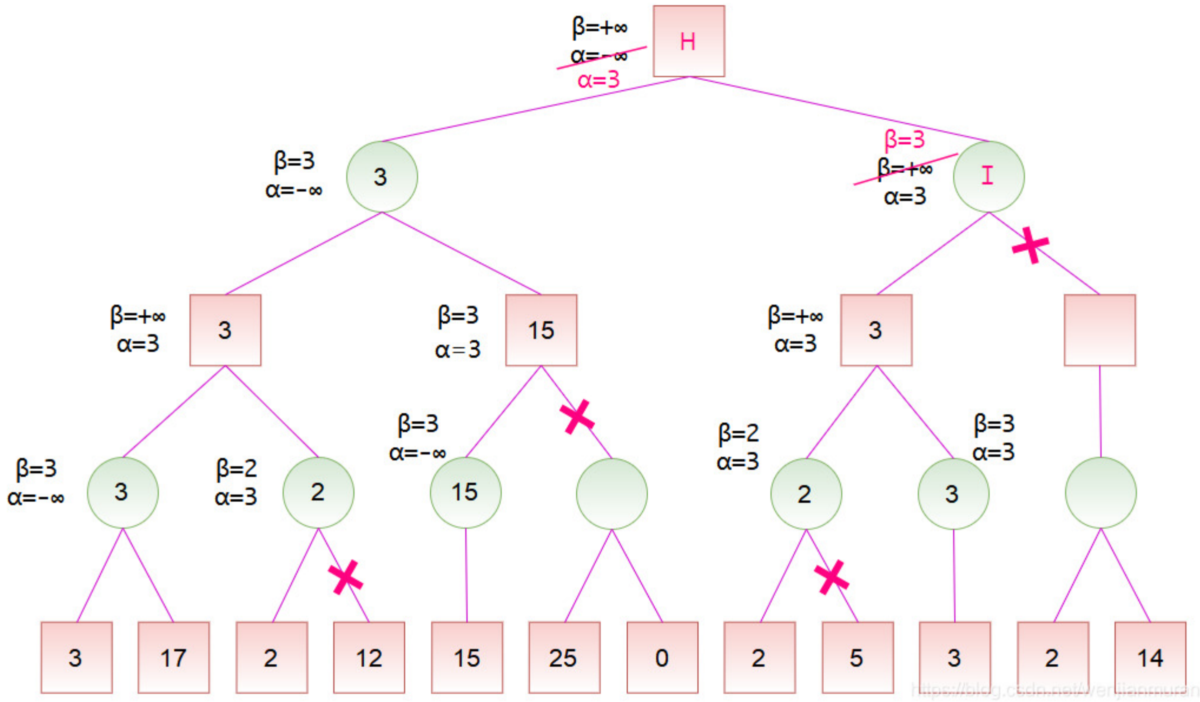


图 6.18

```
    }
    return beta;
  }
}
```

参考资料与注释

本文部分引用自博文 详解 Minimax 算法与 - 剪枝 _ 文剑木然[2]，遵循 CC 4.0 BY-SA 版权协议。

[1] 极小化极大算法 - 维基百科 ， 自由的百科全书

[2] 详解 Minimax 算法与 - 剪枝 _ 文剑木然

6.12 优化

Authors: CBW2007, ChungZH, Marcythm, abc1763613206, Ir1d

前言

DFS（深度优先搜索）是一种常见的算法，大部分的题目都可以用 DFS 解决，但是大部分情况下，这都是骗分算法，很少会有爆搜为正解的题目。因为 DFS 的时间复杂度特别高。（没学过 DFS 的请自行补上这一课）

既然不能成为正解，那就多骗一点分吧。那么这一篇文章将介绍一些实用的优化算法（俗称「剪枝」）。

先来一段深搜模板，之后的模板将在此基础上进行修改。

```
int ans = 最坏情况, now;  // now 为当前答案
```

```
void dfs(传入数值) {
  if (到达目的地) ans = 从当前解与已有解中选最优;
  for (遍历所有可能性)
    if (可行) {
      进行操作;
      dfs(缩小规模);
      撤回操作;
    }
}
```

其中的 ans 可以是解的记录，那么从当前解与已有解中选最优就变成了输出解。

剪枝方法

最常用的剪枝有三种，记忆化搜索、最优性剪枝、可行性剪枝。

记忆化搜索

因为在搜索中，相同的传入值往往会带来相同的解，那我们就可以用数组来记忆，详见 记忆化搜索。

模板：

```
int g[MAXN];  // 定义记忆化数组
int ans = 最坏情况, now;

void dfs f(传入数值) {
  if (g[规模] != 无效数值) return;  // 或记录解，视情况而定
  if (到达目的地) ans = 从当前解与已有解中选最优;  // 输出解，视情况而定
  for (遍历所有可能性)
    if (可行) {
      进行操作;
      dfs(缩小规模);
      撤回操作;
    }
}

int main() {
  // ...
  memset(g, 无效数值, sizeof(g));  // 初始化记忆化数组
  // ...
}
```

最优性剪枝

在搜索中导致运行慢的原因还有一种，就是在当前解已经比已有解差时仍然在搜索，那么我们只需要判断一下当前解是否已经差于已有解。

模板：

```
int ans = 最坏情况, now;

void dfs(传入数值) {
  if (now 比 ans 的答案还要差) return;
  if (到达目的地) ans = 从当前解与已有解中选最优;
  for (遍历所有可能性)
```

```
    if (可行) {
      进行操作;
      dfs(缩小规模);
      撤回操作;
    }
}
```

可行性剪枝

在搜索过程中当前解已经不可用了还继续搜索下去也是运行慢的原因。

模板：

```
int ans = 最坏情况, now;

void dfs(传入数值) {
  if (当前解已不可用) return;
  if (到达目的地) ans = 从当前解与已有解中选最优;
  for (遍历所有可能性)
    if (可行) {
      进行操作;
      dfs(缩小规模);
      撤回操作;
    }
}
```

剪枝思路

剪枝思路有很多种，大多需要对于具体问题来分析，在此简要介绍几种常见的剪枝思路。

- 极端法：考虑极端情况，如果最极端（最理想）的情况都无法满足，那么肯定实际情况搜出来的结果不会更优了。
- 调整法：通过对子树的比较剪掉重复子树和明显不是最有「前途」的子树。
- 数学方法：比如在图论中借助连通分量，数论中借助模方程的分析，借助不等式的放缩来估计下界等等。

例题

" 工作分配问题"

题目描述

有 n 份工作要分配给 n 个人来完成，每个人完成一份。第 i 个人完成第 k 份工作所用的时间为一个正整数 $t_{i,k}$，其中 $1 \leq i, k \leq n$。试确定一个分配方案，使得完成这 n 份工作的时间总和最小。

输入包含 $n+1$ 行。

第 1 行为一个正整数 n。

第 2 行到第 $n+1$ 行中每行都包含 n 个正整数，形成了一个 $n \times n$ 的矩阵。在该矩阵中，第 i 行第 k 列元素 $t_{i,k}$ 表示第 i 个人完成第 k 件工作所要用的时间。

输出包含一个正整数，表示所有分配方案中最小的时间总和。

数据范围

$1 \leq n \leq 15$

$1 \leq t_{i,k} \leq 10^4$

输入样例

```
5
9 2 9 1 9
1 9 8 9 6
9 9 9 9 1
8 8 1 8 4
9 1 7 8 9
```

输出样例

```
5
```

由于每个人都必须分配到工作，在这里可以建一个二维数组 `time[i][j]`，用以表示 i 个人完成 j 号工作所花费的时间。给定一个循环，从第 1 个人开始循环分配工作，直到所有人都分配到。为第 i 个人分配工作时，再循环检查每个工作是否已被分配，没有则分配给 i 个人，否则检查下一个工作。可以用一个一维数组 `is_working[j]` 来表示第 j 号工作是否已被分配，未分配则 `is_working[j]=0`，否则 `is_working[j]=1`。利用回溯思想，在工人循环结束后回到上一工人，取消此次分配的工作，而去分配下一工作直到可以分配为止。这样，一直回溯到第 1 个工人后，就能得到所有的可行解。

检查工作分配，其实就是判断取得可行解时的二维数组的第一维下标各不相同并且第二维下标各不相同。而我们是要得到完成这 n 份工作的最小时间总和，即可行解中时间总和最小的一个，故需要再定义一个全局变量 `cost_time_total_min` 表示目前找到的解中最小的时间总和，初始 `cost_time_total_min` 为 `time[i][i]` 之和，即对角线工作时间相加之和。在所有人分配完工作时，比较 `count` 与 `cost_time_total_min` 的大小，如果 `count` 小于 `cost_time_total_min`，说明找到了一个最优解，此时就把 `count` 赋给 `cost_time_total_min`。

但考虑到算法的效率，这里还有一个剪枝优化的工作可以做。就是在每次计算局部费用变量 `count` 的值时，如果判断 `count` 已经大于 `cost_time_total_min`，就没必要再往下分配了，因为这时得到的解必然不是最优解。

" 参考代码"

```cpp
#include <cstdio>
#define N 16
int is_working[N] = {0};  // 某项工作是否被分配
int time[N][N];           // 完成某项工作所需的时间
int cost_time_total_min;  // 完成 n 份工作的最小时间总和

// i 表示第几个人，count 表示工作费用总和
void work(int i, int count, int n) {
  // 如果 i 超出了所能分配的最大工作件数，表示分配完成，并且 count 比原来
  // cost_time_total_min 花费少，则更新 cost_time_total_min 的值
  if (i > n && count < cost_time_total_min) {
    cost_time_total_min = count;
    return;
  }
  // 回溯思想
  if (count < cost_time_total_min) {
    // j 表示第几件工作
    for (int j = 1; j <= n; j++) {
      // 如果工作未被分配 is_working = 0
      if (is_working[j] == 0) {
        // 分配工作 is_working = 1
        is_working[j] = 1;
        // 工作交给第 i + 1 个人
        work(i + 1, count + time[i][j], n);
        // 在一轮迭代完成之后，返回到上一个人，要对此次的工作进行重新分配
        // 将 is_working[j] 重设为 0
        is_working[j] = 0;
```

```
      }
    }
  }
}

int main() {
  int n;
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      scanf("%d", &time[i][j]);
    }
    cost_time_total_min += time[i][i];
  }
  work(1, 0, n);
  printf("%d\n", cost_time_total_min);
  return 0;
}
```

第 7 章

动态规划

7.1 动态规划部分简介

本章将介绍介绍动态规划（Dynamic Programming, DP）及其解决的问题、根据其设计的算法及优化。

动态规划是一种通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法。

由于动态规划并不是某种具体的算法，而是一种解决特定问题的方法，因此它会出现在各式各样的数据结构中，与之相关的题目种类也更为繁杂。

在 OI 中，计数等非最优化问题的递推解法也常被不规范地称作 DP，因此本章将它们一并列出。事实上，动态规划与其它类型的递推的确有很多相似之处，学习时可以注意它们之间的异同。

参考资料

动态规划 - 维基百科， 自由的百科全书[1]

参考资料与注释

[1] 动态规划 - 维基百科，自由的百科全书

7.2 动态规划基础

Authors: Ir1d, CBW2007, ChungZH, xhn16729, Xeonacid, tptpp, hsfzLZH1, ouuan, Marcythm, HeRaNO, greyqz, Chrogeek, partychicken, zhb2000, xyf007, Persdre, XiaoSuan250, hhc0001, ZhangZhanhaoxiang

本页面主要介绍了动态规划的基本思想，以及动态规划中状态及状态转移方程的设计思路，帮助各位初学者对动态规划有一个初步的了解。

本部分的其他页面，将介绍各种类型问题中动态规划模型的建立方法，以及一些动态规划的优化技巧。

引入

"[IOI1994] **数字三角形**[3]"

给定一个 r 行的数字三角形（$r \leq 1000$），需要找到一条从最高点到底部任意处结束的路径，使路径经过数字的和最大。每一步可以走到当前点左下方的点或右下方的点。

```
        7
      3   8
    8   1   0
  2   7   4   4
4   5   2   6   5
```

在上面这个例子中，最优路径是 $7 \to 3 \to 8 \to 7 \to 5$。

最简单粗暴的思路是尝试所有的路径。因为路径条数是 $O(2^r)$ 级别的，这样的做法无法接受。

注意到这样一个事实，一条最优的路径，它的每一步决策都是最优的。

以例题里提到的最优路径为例，只考虑前四步 $7 \to 3 \to 8 \to 7$，不存在一条从最顶端到 4 行第 2 个数的权值更大的路径。

而对于每一个点，它的下一步决策只有两种：往左下角或者往右下角（如果存在）。因此只需要记录当前点的最大权值，用这个最大权值执行下一步决策，来更新后续点的最大权值。

这样做还有一个好处：我们成功缩小了问题的规模，将一个问题分成了多个规模更小的问题。要想得到从顶端到第 r 行的最优方案，只需要知道从顶端到第 $r-1$ 行的最优方案的信息就可以了。

这时候还存在一个问题：子问题间重叠的部分会有很多，同一个子问题可能会被重复访问多次，效率还是不高。解决这个问题的方法是把每个子问题的解存储下来，通过记忆化的方式限制访问顺序，确保每个子问题只被访问一次。

上面就是动态规划的一些基本思路。下面将会更系统地介绍动态规划的思想。

动态规划原理

能用动态规划解决的问题，需要满足三个条件：最优子结构，无后效性和子问题重叠。

最优子结构

具有最优子结构也可能是适合用贪心的方法求解。

注意要确保我们考察了最优解中用到的所有子问题。

1. 证明问题最优解的第一个组成部分是做出一个选择；
2. 对于一个给定问题，在其可能的第一步选择中，假定你已经知道哪种选择才会得到最优解。你现在并不关心这种选择具体是如何得到的，只是假定已经知道了这种选择；
3. 给定可获得的最优解的选择后，确定这次选择会产生哪些子问题，以及如何最好地刻画子问题空间；
4. 证明作为构成原问题最优解的组成部分，每个子问题的解就是它本身的最优解。方法是反证法，考虑加入某个子问题的解不是其自身的最优解，那么就可以从原问题的解中用该子问题的最优解替换掉当前的非最优解，从而得到原问题的一个更优的解，从而与原问题最优解的假设矛盾。

要保持子问题空间尽量简单，只在必要时扩展。

最优子结构的不同体现在两个方面：

1. 原问题的最优解中涉及多少个子问题；
2. 确定最优解使用哪些子问题时，需要考察多少种选择。

子问题图中每个定点对应一个子问题，而需要考察的选择对应关联至子问题顶点的边。

无后效性

已经求解的子问题，不会再受到后续决策的影响。

子问题重叠

如果有大量的重叠子问题，我们可以用空间将这些子问题的解存储下来，避免重复求解相同的子问题，从而提升效率。

基本思路

对于一个能用动态规划解决的问题，一般采用如下思路解决：

1. 将原问题划分为若干**阶段**，每个阶段对应若干个子问题，提取这些子问题的特征（称之为**状态**）；
2. 寻找每一个状态的可能**决策**，或者说是各状态间的相互转移方式（用数学的语言描述就是**状态转移方程**）。
3. 按顺序求解每一个阶段的问题。

如果用图论的思想理解，我们建立一个 有向无环图，每个状态对应图上一个节点，决策对应节点间的连边。这样问题就转变为了一个在 DAG 上寻找最长（短）路的问题（参见：DAG 上的 DP）。

最长公共子序列

" 最长公共子序列问题"

给定一个长度为 n 的序列 A 和一个长度为 m 的序列 B（$n, m \le 5000$），求出一个最长的序列，使得该序列既是 A 的子序列，也是 B 的子序列。

子序列的定义可以参考 子序列。一个简要的例子：字符串 `abcde` 与字符串 `acde` 的公共子序列有 `a`、`c`、`d`、`e`、`ac`、`ad`、`ae`、`cd`、`ce`、`de`、`ade`、`ace`、`cde`、`acde`，最长公共子序列的长度是 4。

设 $f(i,j)$ 表示只考虑 A 的前 i 个元素，B 的前 j 个元素时的最长公共子序列的长度，求这时的最长公共子序列的长度就是**子问题**。$f(i,j)$ 就是我们所说的**状态**，则 $f(n,m)$ 是最终要达到的状态，即为所求结果。

对于每个 $f(i,j)$，存在三种决策：如果 $A_i = B_j$，则可以将它接到公共子序列的末尾；另外两种决策分别是跳过 A_i 或者 B_j。状态转移方程如下：

$$f(i,j) = \begin{cases} f(i-1,j-1)+1 & A_i = B_j \\ \max(f(i-1,j), f(i,j-1)) & A_i \ne B_j \end{cases}$$

可参考 SourceForge 的 LCS 交互网页[4] 来更好地理解 LCS 的实现过程。

该做法的时间复杂度为 $O(nm)$。

另外，本题存在 $O\left(\frac{nm}{w}\right)$ 的算法[1]。有兴趣的同学可以自行探索。

```
int a[MAXN], b[MAXM], f[MAXN][MAXM];

int dp() {
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++)
      if (a[i] == b[j])
        f[i][j] = f[i - 1][j - 1] + 1;
      else
        f[i][j] = std::max(f[i - 1][j], f[i][j - 1]);
  return f[n][m];
}
```

最长不下降子序列

"最长不下降子序列问题"

给定一个长度为 n 的序列 A（$n \le 5000$），求出一个最长的 A 的子序列，满足该子序列的后一个元素不小于前一个元素。

算法一

设 $f(i)$ 表示以 A_i 为结尾的最长不下降子序列的长度，则所求为 $\max_{1\le i\le n} f(i)$。

计算 $f(i)$ 时，尝试将 A_i 接到其他的最长不下降子序列后面，以更新答案。于是可以写出这样的状态转移方程：$f(i) = \max_{1\le j<i, A_j\le A_i}(f(j)+1)$。

容易发现该算法的时间复杂度为 $O(n^2)$。

```cpp
int a[MAXN], d[MAXN];

int dp() {
  d[1] = 1;
  int ans = 1;
  for (int i = 2; i <= n; i++) {
    d[i] = 1;
    for (int j = 1; j < i; j++)
      if (a[j] <= a[i]) {
        d[i] = max(d[i], d[j] + 1);
        ans = max(ans, d[i]);
      }
  }
  return ans;
}
```

```python
a = [0] * MAXN
d = [0] * MAXN
def dp():
    d[1] = 1
    ans = 1
    for i in range(2, n + 1):
        for j in range(1, i):
            if a[j] <= a[i]:
                d[i] = max(d[i], d[j] + 1)
                ans = max(ans, d[i])
    return ans
```

算法二[2]

当 n 的范围扩大到 $n \le 10^5$ 时，第一种做法就不够快了，下面给出了一个 $O(n\log n)$ 的做法。

回顾一下之前的状态：(i,l)。

但这次，我们不是要按照相同的 i 处理状态，而是直接判断合法的 (i,l)。

再看一下之前的转移：$(j,l-1) \to (i,l)$，就可以判断某个 (i,l) 是否合法。

初始时 $(1,1)$ 肯定合法。

那么，只需要找到一个 l 最大的合法的 (i,l)，就可以得到最终最长不下降子序列的长度了。

那么，根据上面的方法，我们就需要维护一个可能的转移列表，并逐个处理转移。

所以可以定义 $a_1 \dots a_n$ 为原始序列，d_i 为所有的长度为 i 的不下降子序列的末尾元素的最小值，len 为子序列的长度。

初始化：$d_1 = a_1, len = 1$。

现在我们已知最长的不下降子序列长度为 1，那么我们让 i 从 2 到 n 循环，依次求出前 i 个元素的最长不下降子序列的长度，循环的时候我们只需要维护好 d 这个数组还有 len 就可以了。**关键在于如何维护**。

考虑进来一个元素 a_i：

1. 元素大于等于 d_{len}，直接将该元素插入到 d 序列的末尾。
2. 元素小于 d_{len}，找到**第一个**大于它的元素，用 a_i 替换它。

为什么：

- 对于步骤 1：

  由于我们是从前往后扫，所以说当元素大于等于 d_{len} 时一定会有一个不下降子序列使得这个不下降子序列的末项后面可以再接这个元素。如果 d 不接这个元素，可以发现既不符合定义，又不是最优解。
- 对于步骤 2：

  同步骤 1，如果插在 d 的末尾，那么由于前面的元素大于要插入的元素，所以不符合 d 的定义，因此必须先找到**第一个**大于它的元素，再用 a_i 替换。

步骤 2 如果采用暴力查找，则时间复杂度仍然是 $O(n^2)$ 的。但是根据 d 数组的定义，又由于本题要求不下降子序列，所以 d 一定是**单调不减**的，因此可以用二分查找将时间复杂度降至 $O(n \log n)$.

参考代码如下：

```cpp
for (int i = 0; i < n; ++i) scanf("%d", a + i);
memset(dp, 0x1f, sizeof dp);
mx = dp[0];
for (int i = 0; i < n; ++i) {
  *std::upper_bound(dp, dp + n, a[i]) = a[i];
}
ans = 0;
while (dp[ans] != mx) ++ans;
```

```python
dp = [0x1f1f1f1f] * MAXN
mx = dp[0]
for i in range(0, n):
    bisect.insort_left(dp, a[i], 0, len(dp))
ans = 0
while dp[ans] != mx:
    ans += 1
```

参考资料与注释

[1] 位运算求最长公共子序列 - -Wallace- - 博客园

[2] 最长不下降子序列 nlogn 算法详解 - lvmememe - 博客园

[3] [IOI1994] 数字三角形

[4] SourceForge 的 LCS 交互网页

7.3 记忆化搜索

定义

记忆化搜索是一种通过记录已经遍历过的状态的信息，从而避免对同一状态重复遍历的搜索实现方式。

因为记忆化搜索确保了每个状态只访问一次，它也是一种常见的动态规划实现方式。

引入

"[NOIP2005] 采药[1]"

山洞里有 M 株不同的草药，采每一株都需要一些时间 t_i，每一株也有它自身的价值 v_i。给你一段时间 T，在这段时间里，你可以采到一些草药。让采到的草药的总价值最大。

$1 \le T \le 10^3$，$1 \le t_i, v_i, M \le 100$

朴素的 DFS 做法

很容易实现这样一个朴素的搜索做法：在搜索时记录下当前准备选第几个物品、剩余的时间是多少、已经获得的价值是多少这三个参数，然后枚举当前物品是否被选，转移到相应的状态。

" 实现"

```cpp
int n, t;
int tcost[103], mget[103];
int ans = 0;

void dfs(int pos, int tleft, int tans) {
  if (tleft < 0) return;
  if (pos == n + 1) {
    ans = max(ans, tans);
    return;
  }
  dfs(pos + 1, tleft, tans);
  dfs(pos + 1, tleft - tcost[pos], tans + mget[pos]);
}

int main() {
  cin >> t >> n;
  for (int i = 1; i <= n; i++) cin >> tcost[i] >> mget[i];
  dfs(1, t, 0);
  cout << ans << endl;
  return 0;
}
```

```python
tcost = [0] * 103
mget = [0] * 103
ans = 0
def dfs(pos, tleft, tans):
    global ans
    if tleft < 0:
        return
```

```
    if pos == n + 1:
        ans = max(ans, tans)
        return
    dfs(pos + 1, tleft, tans)
    dfs(pos + 1, tleft - tcost[pos], tans + mget[pos])
t, n = map(lambda x:int(x), input().split())
for i in range(1, n + 1):
    tcost[i], mget[i] = map(lambda x:int(x), input().split())
dfs(1, t, 0)
print(ans)
```

这种做法的时间复杂度是指数级别的，并不能通过本题。

优化

上面的做法为什么效率低下呢？因为同一个状态会被访问多次。

如果我们每查询完一个状态后将该状态的信息存储下来，再次需要访问这个状态就可以直接使用之前计算得到的信息，从而避免重复计算。这充分利用了动态规划中很多问题具有大量重叠子问题的特点，属于用空间换时间的「记忆化」思想。

具体到本题上，我们在朴素的 DFS 的基础上，增加一个数组 `mem` 来记录每个 `dfs(pos,tleft)` 的返回值。刚开始把 `mem` 中每个值都设成 `-1`（代表没求解过）。每次需要访问一个状态时，如果相应状态的值在 `mem` 中为 `-1`，则递归访问该状态。否则我们直接使用 `mem` 中已经存储过的值即可。

通过这样的处理，我们确保了每个状态只会被访问一次，因此该算法的的时间复杂度为 $O(TM)$。

" 实现"

```
int n, t;
int tcost[103], mget[103];
int mem[103][1003];

int dfs(int pos, int tleft) {
  if (mem[pos][tleft] != -1)
    return mem[pos][tleft];  // 已经访问过的状态，直接返回之前记录的值
  if (pos == n + 1) return mem[pos][tleft] = 0;
  int dfs1, dfs2 = -INF;
  dfs1 = dfs(pos + 1, tleft);
  if (tleft >= tcost[pos])
    dfs2 = dfs(pos + 1, tleft - tcost[pos]) + mget[pos];  // 状态转移
  return mem[pos][tleft] = max(dfs1, dfs2);  // 最后将当前状态的值存下来
}

int main() {
  memset(mem, -1, sizeof(mem));
  cin >> t >> n;
  for (int i = 1; i <= n; i++) cin >> tcost[i] >> mget[i];
  cout << dfs(1, t) << endl;
  return 0;
}
```

```
tcost = [0] * 103
mget = [0] * 103
mem = [[-1 for i in range(1003)] for j in range(103)]
def dfs(pos, tleft):
```

```
    if mem[pos][tleft] != -1:
        return mem[pos][tleft]
    if pos == n + 1:
        mem[pos][tleft] = 0
        return mem[pos][tleft]
    dfs1 = dfs2 = -INF
    dfs1 = dfs(pos + 1, tleft)
    if tleft >= tcost[pos]:
        dfs2 = dfs(pos + 1, tleft - tcost[pos]) + mget[pos]
    mem[pos][tleft] = max(dfs1, dfs2)
    return mem[pos][tleft]
t, n = map(lambda x:int(x), input().split())
for i in range(1, n + 1):
    tcost[i], mget[i] = map(lambda x:int(x), input().split())
print(dfs(1, t))
```

与递推的联系与区别

在求解动态规划的问题时，记忆化搜索与递推的代码，在形式上是高度类似的。这是由于它们使用了相同的状态表示方式和类似的状态转移。也正因为如此，一般来说两种实现的时间复杂度是一样的。

下面给出的是递推实现的代码（为了方便对比，没有添加滚动数组优化），通过对比可以发现二者在形式上的类似性。

```
const int maxn = 1010;
int n, t, w[105], v[105], f[105][1005];

int main() {
  cin >> n >> t;
  for (int i = 1; i <= n; i++) cin >> w[i] >> v[i];
  for (int i = 1; i <= n; i++)
    for (int j = 0; j <= t; j++) {
      f[i][j] = f[i - 1][j];
      if (j >= w[i])
        f[i][j] = max(f[i][j], f[i - 1][j - w[i]] + v[i]);  // 状态转移方程
    }
  cout << f[n][t];
  return 0;
}
```

在求解动态规划的问题时，记忆化搜索和递推，都确保了同一状态至多只被求解一次。而它们实现这一点的方式则略有不同：递推通过设置明确的访问顺序来避免重复访问，记忆化搜索虽然没有明确规定访问顺序，但通过给已经访问过的状态打标记的方式，也达到了同样的目的。

与递推相比，记忆化搜索因为不用明确规定访问顺序，在实现难度上有时低于递推，且能比较方便地处理边界情况，这是记忆化搜索的一大优势。但与此同时，记忆化搜索难以使用滚动数组等优化，且由于存在递归，运行效率会低于递推。因此应该视题目选择更适合的实现方式。

如何写记忆化搜索

方法一

1. 把这道题的 dp 状态和方程写出来
2. 根据它们写出 dfs 函数
3. 添加记忆化数组

举例：

$dp_i = \max\{dp_j + 1\} \quad (1 \le j < i \land a_j < a_i)$（最长上升子序列）

转为

```cpp
int dfs(int i) {
  if (mem[i] != -1) return mem[i];
  int ret = 1;
  for (int j = 1; j < i; j++)
    if (a[j] < a[i]) ret = max(ret, dfs(j) + 1);
  return mem[i] = ret;
}

int main() {
  memset(mem, -1, sizeof(mem));
  // 读入部分略去
  int ret = 0;
  for (int j = 1; j <= n; j++) {
    ret = max(ret, dfs(j));
  }
  cout << ret << endl;
}
```

```python
def dfs(i):
    if mem[i] != -1:
        return mem[i]
    ret = 1
    for j in range(1, i):
        if a[j] < a[i]:
            ret = max(ret, dfs(j) + 1)
    mem[i] = ret
    return mem[i]
```

方法二

1. 写出这道题的暴搜程序（最好是 dfs）
2. 将这个 dfs 改成「无需外部变量」的 dfs
3. 添加记忆化数组

举例：本文中「采药」的例子

参考资料与注释

[1] [NOIP2005] 采药

7.4 背包 DP

Authors: hydingsy, Link-cute, Ir1d, greyqz, LuoshuiTianyi, Odeinjul, xyf007, GoodCoder666, paigeman, shenshuaijie

前置知识：动态规划部分简介。

引入

在具体讲何为「背包 dp」前，先来看如下的例题：

"「USACO07 DEC」Charm Bracelet[1]"

题意概要：有 n 个物品和一个容量为 W 的背包，每个物品有重量 w_i 和价值 v_i 两种属性，要求选若干物品放入背包使背包中物品的总价值最大且背包中物品的总重量不超过背包的容量。

在上述例题中，由于每个物体只有两种可能的状态（取与不取），对应二进制中的 0 和 1，这类问题便被称为「0-1 背包问题」。

0-1 背包

解释

例题中已知条件有第 i 个物品的重量 w_i，价值 v_i，以及背包的总容量 W。

设 DP 状态 $f_{i,j}$ 为在只能放前 i 个物品的情况下，容量为 j 的背包所能达到的最大总价值。

考虑转移。假设当前已经处理好了前 $i-1$ 个物品的所有状态，那么对于第 i 个物品，当其不放入背包时，背包的剩余容量不变，背包中物品的总价值也不变，故这种情况的最大价值为 $f_{i-1,j}$；当其放入背包时，背包的剩余容量会减小 w_i，背包中物品的总价值会增大 v_i，故这种情况的最大价值为 $f_{i-1,j-w_i}+v_i$。

由此可以得出状态转移方程：

$$f_{i,j}=\max(f_{i-1,j},f_{i-1,j-w_i}+v_i)$$

这里如果直接采用二维数组对状态进行记录，会出现 MLE。可以考虑改用滚动数组的形式来优化。

由于对 f_i 有影响的只有 f_{i-1}，可以去掉第一维，直接用 f_i 来表示处理到当前物品时背包容量为 i 的最大价值，得出以下方程：

$$f_j=\max\left(f_j,f_{j-w_i}+v_i\right)$$

务必牢记并理解这个转移方程，因为大部分背包问题的转移方程都是在此基础上推导出来的。

实现

还有一点需要注意的是，很容易写出这样的**错误核心代码**：

```cpp
for (int i = 1; i <= n; i++)
  for (int l = 0; l <= W - w[i]; l++)
    f[l + w[i]] = max(f[l] + v[i], f[l + w[i]]);
// 由 f[i][l + w[i]] = max(max(f[i - 1][l + w[i]], f[i - 1][l] + w[i]),
// f[i][l + w[i]]); 简化而来
```

```python
for i in range(1, n + 1):
    for l in range(0, W - w[i] + 1):
        f[l + w[i]] = max(f[l] + v[i], f[l + w[i]])
# 由 f[i][l + w[i]] = max(max(f[i - 1][l + w[i]], f[i - 1][l] + w[i]),
# f[i][l + w[i]]) 简化而来
```

这段代码哪里错了呢？枚举顺序错了。

仔细观察代码可以发现：对于当前处理的物品 i 和当前状态 $f_{i,j}$，在 $j\geqslant w_i$ 时，$f_{i,j}$ 是会被 $f_{i,j-w_i}$ 所影响的。这就相当于物品 i 可以多次被放入背包，与题意不符。（事实上，这正是完全背包问题的解法）

为了避免这种情况发生，我们可以改变枚举的顺序，从 W 枚举到 w_i，这样就不会出现上述的错误，因为 $f_{i,j}$ 总是在 $f_{i,j-w_i}$ 前被更新。

因此实际核心代码为

```cpp
for (int i = 1; i <= n; i++)
  for (int l = W; l >= w[i]; l--) f[l] = max(f[l], f[l - w[i]] + v[i]);
```

```python
for i in range(1, n + 1):
    for l in range(W, w[i] - 1, -1):
        f[l] = max(f[l], f[l - w[i]] + v[i])
```

例题代码

```cpp
#include <iostream>
using namespace std;
const int maxn = 13010;
int n, W, w[maxn], v[maxn], f[maxn];

int main() {
  cin >> n >> W;
  for (int i = 1; i <= n; i++) cin >> w[i] >> v[i];  // 读入数据
  for (int i = 1; i <= n; i++)
    for (int l = W; l >= w[i]; l--)
      if (f[l - w[i]] + v[i] > f[l]) f[l] = f[l - w[i]] + v[i];  // 状态方程
  cout << f[W];
  return 0;
}
```

完全背包

解释

完全背包模型与 0-1 背包类似，与 0-1 背包的区别仅在于一个物品可以选取无限次，而非仅能选取一次。

我们可以借鉴 0-1 背包的思路，进行状态定义：设 $f_{i,j}$ 为只能选前 i 个物品时，容量为 j 的背包可以达到的最大价值。

需要注意的是，虽然定义与 0-1 背包类似，但是其状态转移方程与 0-1 背包并不相同。

过程

可以考虑一个朴素的做法：对于第 i 件物品，枚举其选了多少个来转移。这样做的时间复杂度是 $O(n^3)$ 的。

状态转移方程如下：

$$f_{i,j} = \max_{k=0}^{+\infty}(f_{i-1,j-k\times w_i} + v_i \times k)$$

考虑做一个简单的优化。可以发现，对于 $f_{i,j}$，只要通过 $f_{i,j-w_i}$ 转移就可以了。因此状态转移方程为：

$$f_{i,j} = \max(f_{i-1,j}, f_{i,j-w_i} + v_i)$$

理由是当我们这样转移时，$f_{i,j-w_i}$ 已经由 $f_{i,j-2\times w_i}$ 更新过，那么 $f_{i,j-w_i}$ 就是充分考虑了第 i 件物品所选次数后得到的最优结果。换言之，我们通过局部最优子结构的性质重复使用了之前的枚举过程，优化了枚举的复杂度。

与 0-1 背包相同，我们可以将第一维去掉来优化空间复杂度。如果理解了 0-1 背包的优化方式，就不难明白压缩后的循环是正向的（也就是上文中提到的错误优化）。

"「Luogu P1616」 疯狂的采药[2]"

题意概要：有 n 种物品和一个容量为 W 的背包，每种物品有重量 w_i 和价值 v_i 两种属性，要求选若干个物品放入背包使背包中物品的总价值最大且背包中物品的总重量不超过背包的容量。

例题代码

```
#include <iostream>
using namespace std;
const int maxn = 1e4 + 5;
const int maxW = 1e7 + 5;
int n, W, w[maxn], v[maxn];
long long f[maxW];

int main() {
  cin >> W >> n;
  for (int i = 1; i <= n; i++) cin >> w[i] >> v[i];
  for (int i = 1; i <= n; i++)
    for (int l = w[i]; l <= W; l++)
      if (f[l - w[i]] + v[i] > f[l]) f[l] = f[l - w[i]] + v[i];  // 核心状态方程
  cout << f[W];
  return 0;
}
```

多重背包

多重背包也是 0-1 背包的一个变式。与 0-1 背包的区别在于每种物品有 k_i 个，而非一个。

一个很朴素的想法就是：把「每种物品选 k_i 次」等价转换为「有 k_i 个相同的物品，每个物品选一次」。这样就转换成了一个 0-1 背包模型，套用上文所述的方法就可已解决。状态转移方程如下：

$$f_{i,j} = \max_{k=0}^{k_i}(f_{i-1,j-k\times w_i} + v_i \times k)$$

时间复杂度 $O(W\sum_{i=1}^{n} k_i)$。

二进制分组优化

考虑优化。我们仍考虑把多重背包转化成 0-1 背包模型来求解。

解释

显然，复杂度中的 $O(nW)$ 部分无法再优化了，我们只能从 $O(\sum k_i)$ 处入手。为了表述方便，我们用 $A_{i,j}$ 代表第 i 种物品拆分出的第 j 个物品。

在朴素的做法中，$\forall j \le k_i$，$A_{i,j}$ 均表示相同物品。那么我们效率低的原因主要在于我们进行了大量重复性的工作。举例来说，我们考虑了「同时选 $A_{i,1}, A_{i,2}$」与「同时选 $A_{i,2}, A_{i,3}$」这两个完全等效的情况。这样的重复性工作我们进行了许多次。那么优化拆分方式就成为了解决问题的突破口。

过程

我们可以通过「二进制分组」的方式使拆分方式更加优美。

具体地说就是令 $A_{i,j}\left(j\in[0,\lfloor\log_2(k_i+1)\rfloor-1]\right)$ 分别表示由 2^j 个单个物品「捆绑」而成的大物品。特殊地，若 k_i+1 不是 2 的整数次幂，则需要在最后添加一个由 $k_i-2^{\lfloor\log_2(k_i+1)\rfloor-1}$ 个单个物品「捆绑」而成的大物品用于补足。

举几个例子：

- $6 = 1 + 2 + 3$
- $8 = 1 + 2 + 4 + 1$
- $18 = 1 + 2 + 4 + 8 + 3$

- $31 = 1 + 2 + 4 + 8 + 16$

显然，通过上述拆分方式，可以表示任意 $\le k_i$ 个物品的等效选择方式。将每种物品按照上述方式拆分后，使用 0-1 背包的方法解决即可。

时间复杂度 $O(W\sum_{i=1}^{n}\log_2 k_i)$

实现

二进制分组代码

```
index = 0;
for (int i = 1; i <= m; i++) {
  int c = 1, p, h, k;
  cin >> p >> h >> k;
  while (k > c) {
    k -= c;
    list[++index].w = c * p;
    list[index].v = c * h;
    c *= 2;
  }
  list[++index].w = p * k;
  list[index].v = h * k;
}

index = 0
for i in range(1, m + 1):
    c = 1
    p, h, k = map(int, input().split())
    while k > c:
        k -= c
        index += 1
        list[index].w = c * p
        list[index].v = c * h
        c *= 2
    index += 1
    list[index].w = p * k
    list[index].v = h * k
```

单调队列优化

见 单调队列 / 单调栈优化。

习题：「Luogu P1776」 宝物筛选 _NOI 导刊 2010 提高 （02）[3]

混合背包

混合背包就是将前面三种的背包问题混合起来，有的只能取一次，有的能取无限次，有的只能取 k 次。

这种题目看起来很吓人，可是只要领悟了前面几种背包的中心思想，并将其合并在一起就可以了。下面给出伪代码：

过程

```
for (循环物品种类) {
  if (是 0 - 1 背包)
    套用 0 - 1 背包代码;
  else if (是完全背包)
    套用完全背包代码;
  else if (是多重背包)
    套用多重背包代码;
}
```

"「Luogu P1833」 **樱花**[4]"

题意概要：有 n 种樱花树和长度为 T 的时间，有的樱花树只能看一遍，有的樱花树最多看 A_i 遍，有的樱花树可以看无数遍。每棵樱花树都有一个美学值 C_i，求在 T 的时间内看哪些樱花树能使美学值最高。

二维费用背包

"「Luogu P1855」 **榨取 kkksc03**[5]"

有 n 个任务需要完成，完成第 i 个任务需要花费 t_i 分钟，产生 c_i 元的开支。

现在有 T 分钟时间，W 元钱来处理这些任务，求最多能完成多少任务。

这道题是很明显的 0-1 背包问题，可是不同的是选一个物品会消耗两种价值（经费、时间），只需在状态中增加一维存放第二种价值即可。

这时候就要注意，再开一维存放物品编号就不合适了，因为容易 MLE。

实现

```
for (int k = 1; k <= n; k++)
  for (int i = m; i >= mi; i--)    // 对经费进行一层枚举
    for (int j = t; j >= ti; j--)  // 对时间进行一层枚举
      dp[i][j] = max(dp[i][j], dp[i - mi][j - ti] + 1);
```

```
for k in range(1, n + 1):
    for i in range(m, mi - 1, -1): # 对经费进行一层枚举
        for j in range(t, ti - 1, -1): # 对时间进行一层枚举
            dp[i][j] = max(dp[i][j], dp[i - mi][j - ti] + 1)
```

分组背包

"「Luogu P1757」 **通天之分组背包**[6]"

有 n 件物品和一个大小为 m 的背包，第 i 个物品的价值为 w_i，体积为 v_i。同时，每个物品属于一个组，同组内最多只能选择一个物品。求背包能装载物品的最大总价值。

这种题怎么想呢？其实是从「在所有物品中选择一件」变成了「从当前组中选择一件」，于是就对每一组进行一次 0-1 背包就可以了。

再说一说如何进行存储。我们可以将 $t_{k,i}$ 表示第 k 组的第 i 件物品的编号是多少，再用 cnt_k 表示第 k 组物品有多少个。

实现

```
for (int k = 1; k <= ts; k++)          // 循环每一组
  for (int i = m; i >= 0; i--)         // 循环背包容量
    for (int j = 1; j <= cnt[k]; j++)  // 循环该组的每一个物品
      if (i >= w[t[k][j]])             // 背包容量充足
        dp[i] = max(dp[i],
                    dp[i - w[t[k][j]]] + c[t[k][j]]);  // 像 0-1 背包一样状态转移
```

```
for k in range(1, ts + 1): # 循环每一组
    for i in range(m, -1, -1): # 循环背包容量
        for j in range(1, cnt[k] + 1):     # 循环该组的每一个物品
            if i >= w[t[k][j]]: # 背包容量充足
                dp[i] = max(dp[i], dp[i - w[t[k][j]]] + c[t[k][j]]) # 像 0-1 背包一
样状态转移
```

这里要注意：**一定不能搞错循环顺序**，这样才能保证正确性。

有依赖的背包

"「Luogu P1064」 **金明的预算方案**[7]"

金明有 n 元钱，想要买 m 个物品，第 i 件物品的价格为 v_i，重要度为 p_i。有些物品是从属于某个主件物品的附件，要买这个物品，必须购买它的主件。

目标是让所有购买的物品的 $v_i \times p_i$ 之和最大。

考虑分类讨论。对于一个主件和它的若干附件，有以下几种可能：只买主件，买主件 + 某些附件。因为这几种可能性只能选一种，所以可以将这看成分组背包。

如果是多叉树的集合，则要先算子节点的集合，最后算父节点的集合。

泛化物品的背包

这种背包，没有固定的费用和价值，它的价值是随着分配给它的费用而定。在背包容量为 V 的背包问题中，当分配给它的费用为 v_i 时，能得到的价值就是 $h(v_i)$。这时，将固定的价值换成函数的引用即可。

杂项

小优化

根据贪心原理，当费用相同时，只需保留价值最高的；当价值一定时，只需保留费用最低的；当有两件物品 i, j 且 i 的价值大于 j 的价值并且 i 的费用小于 j 的费用时，只需保留 i。

背包问题变种

实现

输出方案其实就是记录下来背包中的某一个状态是怎么推出来的。我们可以用 $g_{i,v}$ 表示第 i 件物品占用空间为 v 的时候是否选择了此物品。然后在转移时记录是选用了哪一种策略（选或不选）。输出时的伪代码：

```
int v = V;  // 记录当前的存储空间
```

```
// 因为最后一件物品存储的是最终状态，所以从最后一件物品进行循环
for (从最后一件循环至第一件) {
  if (g[i][v]) {
    选了第 i 项物品;
    v -= 第 i 项物品的重量;
  } else {
    未选第 i 项物品;
  }
}
```

求方案数

对于给定的一个背包容量、物品费用、其他关系等的问题，求装到一定容量的方案总数。

这种问题就是把求最大值换成求和即可。

例如 0-1 背包问题的转移方程就变成了：

$$dp_i = \sum(dp_i, dp_{i-c_i})$$

初始条件：$dp_0 = 1$

因为当容量为 0 时也有一个方案，即什么都不装。

求最优方案总数

要求最优方案总数，我们要对 0-1 背包里的 dp 数组的定义稍作修改，DP 状态 $f_{i,j}$ 为在只能放前 i 个物品的情况下，容量为 j 的背包「正好装满」所能达到的最大总价值。

这样修改之后，每一种 DP 状态都可以用一个 $g_{i,j}$ 来表示方案数。

$f_{i,j}$ 表示只考虑前 i 个物品时背包体积「正好」是 j 时的最大价值。

$g_{i,j}$ 表示只考虑前 i 个物品时背包体积「正好」是 j 时的方案数。

转移方程：

如果 $f_{i,j} = f_{i-1,j}$ 且 $f_{i,j} \neq f_{i-1,j-v} + w$ 说明我们此时不选择把物品放入背包更优，方案数由 $g_{i-1,j}$ 转移过来，

如果 $f_{i,j} \neq f_{i-1,j}$ 且 $f_{i,j} = f_{i-1,j-v} + w$ 说明我们此时选择把物品放入背包更优，方案数由 $g_{i-1,j-v}$ 转移过来，

如果 $f_{i,j} = f_{i-1,j}$ 且 $f_{i,j} = f_{i-1,j-v} + w$ 说明放入或不放入都能取得最优解，方案数由 $g_{i-1,j}$ 和 $g_{i-1,j-v}$ 转移过来。

初始条件：

```
memset(f, 0x3f3f, sizeof(f));  // 避免没有装满而进行了转移
f[0] = 0;
g[0] = 1;  // 什么都不装是一种方案
```

因为背包体积最大值有可能装不满，所以最优解不一定是 f_m。

最后我们通过找到最优解的价值，把 g_j 数组里取到最优解的所有方案数相加即可。

" 实现"

```
for (int i = 0; i < N; i++) {
  for (int j = V; j >= v[i]; j--) {
    int tmp = std::max(dp[j], dp[j - v[i]] + w[i]);
    int c = 0;
    if (tmp == dp[j]) c += cnt[j];                    // 如果从 dp[j] 转移
    if (tmp == dp[j - v[i]] + w[i]) c += cnt[j - v[i]];  // 如果从 dp[j-v[i]] 转移
    dp[j] = tmp;
    cnt[j] = c;
  }
```

```cpp
}
int max = 0;  // 寻找最优解
for (int i = 0; i <= V; i++) {
  max = std::max(max, dp[i]);
}
int res = 0;
for (int i = 0; i <= V; i++) {
  if (dp[i] == max) {
    res += cnt[i];  // 求和最优解方案数
  }
}
```

背包的第 k 优解

普通的 0-1 背包是要求最优解，在普通的背包 DP 方法上稍作改动，增加一维用于记录当前状态下的前 k 优解，即可得到求 0-1 背包第 k 优解的算法。具体来讲：$dp_{i,j,k}$ 记录了前 i 个物品中，选择的物品总体积为 j 时，能够得到的第 k 大的价值和。这个状态可以理解为将普通 0-1 背包只用记录一个数据的 $dp_{i,j}$ 扩展为记录一个有序的优解序列。转移时，普通背包最优解的求法是 $dp_{i,j}=\max(dp_{i-1,j},dp_{i-1,j-v_i}+w_i)$，现在我们则是要合并 $dp_{i-1,j}$，$dp_{i-1,j-v_i}+w_i$ 这两个大小为 k 的递减序列，并保留合并后前 k 大的价值记在 $dp_{i,j}$ 里，这一步利用双指针法，复杂度是 $O(k)$ 的，整体时间复杂度为 $O(nmk)$。空间上，此方法与普通背包一样可以压缩掉第一维，复杂度是 $O(mk)$ 的。

"例题 hdu 2639 Bone Collector II[8]"

求 0-1 背包的严格第 k 优解。$n\le 100, v\le 1000, k\le 30$

" 实现"

```cpp
memset(dp, 0, sizeof(dp));
int i, j, p, x, y, z;
scanf("%d%d%d", &n, &m, &K);
for (i = 0; i < n; i++) scanf("%d", &w[i]);
for (i = 0; i < n; i++) scanf("%d", &c[i]);
for (i = 0; i < n; i++) {
  for (j = m; j >= c[i]; j--) {
    for (p = 1; p <= K; p++) {
      a[p] = dp[j - c[i]][p] + w[i];
      b[p] = dp[j][p];
    }
    a[p] = b[p] = -1;
    x = y = z = 1;
    while (z <= K && (a[x] != -1 || b[y] != -1)) {
      if (a[x] > b[y])
        dp[j][z] = a[x++];
      else
        dp[j][z] = b[y++];
      if (dp[j][z] != dp[j][z - 1]) z++;
    }
  }
}
printf("%d\n", dp[m][K]);
```

参考资料与注释

- 背包问题九讲 - 崔添翼[9]。

参考资料与注释

[1] 「USACO07 DEC」Charm Bracelet

[2] 「Luogu P1616」疯狂的采药

[3] 「Luogu P1776」宝物筛选 _NOI 导刊 2010 提高（02）

[4] 「Luogu P1833」樱花

[5] 「Luogu P1855」榨取 kkksc03

[6] 「Luogu P1757」通天之分组背包

[7] 「Luogu P1064」金明的预算方案

[8] 例题 hdu 2639 Bone Collector II

[9] 背包问题九讲 - 崔添翼

7.5 区间 DP

定义

区间类动态规划是线性动态规划的扩展，它在分阶段地划分问题时，与阶段中元素出现的顺序和由前一阶段的哪些元素合并而来有很大的关系。

令状态 $f(i,j)$ 表示将下标位置 i 到 j 的所有元素合并能获得的价值的最大值，那么 $f(i,j)=\max\{f(i,k)+f(k+1,j)+cost\}$，$cost$ 为将这两组元素合并起来的价值。

性质

区间 DP 有以下特点：

合并：即将两个或多个部分进行整合，当然也可以反过来；

特征：能将问题分解为能两两合并的形式；

求解：对整个问题设最优值，枚举合并点，将问题分解为左右两个部分，最后合并两个部分的最优值得到原问题的最优值。

解释

例题

"「NOI1995」 **石子合并**[1]"

题目大意：在一个环上有 n 个数 $a_1,a_2,\dots,a_n$，进行 $n-1$ 次合并操作，每次操作将相邻的两堆合并成一堆，能获得新的一堆中的石子数量的和的得分。你需要最大化你的得分。

需要考虑不在环上，而在一条链上的情况。

令 $f(i,j)$ 表示将区间 $[i,j]$ 内的所有石子合并到一起的最大得分。

写出**状态转移方程**：$f(i,j)=\max\{f(i,k)+f(k+1,j)+\sum_{t=i}^{j}a_t\}\ (i\le k<j)$

令 sum_i 表示 a 数组的前缀和，状态转移方程变形为 $f(i,j)=\max\{f(i,k)+f(k+1,j)+sum_j-sum_{i-1}\}$。

怎样进行状态转移

由于计算 $f(i,j)$ 的值时需要知道所有 $f(i,k)$ 和 $f(k+1,j)$ 的值，而这两个中包含的元素的数量都小于 $f(i,j)$，所以我们以 $len=j-i+1$ 作为 DP 的阶段。首先从小到大枚举 len，然后枚举 i 的值，根据 len 和 i 用公式计算出 j 的值，然后枚举 k，时间复杂度为 $O(n^3)$

怎样处理环

题目中石子围成一个环，而不是一条链，怎么办呢?

方法一：由于石子围成一个环，我们可以枚举分开的位置，将这个环转化成一个链，由于要枚举 n 次，最终的时间复杂度为 $O(n^4)$。

方法二：我们将这条链延长两倍，变成 $2\times n$ 堆，其中第 i 堆与第 $n+i$ 堆相同，用动态规划求解后，取 $f(1,n),f(2,n+1),\dots,f(n-1,2n-2)$ 中的最优值，即为最后的答案。时间复杂度 $O(n^3)$。

实现

```cpp
for (len = 2; len <= n; len++)
  for (i = 1; i <= 2 * n - 1 - len; i++) {
    int j = len + i - 1;
    for (k = i; k < j; k++)
      f[i][j] = max(f[i][j], f[i][k] + f[k + 1][j] + sum[j] - sum[i - 1]);
  }
```

```python
for len in range(2, n + 1):
    for i in range(1, 2 * n - len):
        j = len + i - 1
        for k in range(i, j):
            f[i][j] = max(f[i][j], f[i][k] + f[k + 1][j] + sum[j] - sum[i - 1])
```

几道练习题

NOIP 2006 能量项链[2]

NOIP 2007 矩阵取数游戏[3]

「IOI2000」 邮局[4]

参考资料与注释

[1] 「NOI1995」石子合并

[2] NOIP 2006 能量项链

[3] NOIP 2007 矩阵取数游戏

[4] 「IOI2000」邮局

7.6 DAG 上的 DP

定义

DAG 即 有向无环图，一些实际问题中的二元关系都可使用 DAG 来建模，从而将这些问题转化为 DAG 上的最长（短）路问题。

解释

以这道题为例子，来分析一下 DAG 建模的过程。

" **例题** UVa 437 **巴比伦塔** The Tower of Babylon[1]"

有 $n(n \leqslant 30)$ 种砖块，已知三条边长，每种都有无穷多个。要求选一些立方体摞成一根尽量高的柱子（每个砖块可以自行选择一条边作为高），使得每个砖块的底面长宽分别严格小于它下方砖块的底面长宽，求塔的最大高度。

过程

建立 DAG

由于每个砖块的底面长宽分别严格小于它下方砖块的底面长宽，因此不难将这样一种关系作为建图的依据，而本题也就转化为最长路问题。

也就是说如果砖块 j 能放在砖块 i 上，那么 i 和 j 之间存在一条边 (i, j)，且边权就是砖块 j 所选取的高。

本题的另一个问题在于每个砖块的高有三种选法，怎样建图更合适呢？

不妨将每个砖块拆解为三种堆叠方式，即将一个砖块分解为三个砖块，每一个拆解得到的砖块都选取不同的高。

初始的起点是大地，大地的底面是无穷大的，则大地可达任意砖块，当然我们写程序时不必特意写上无穷大。

假设有两个砖块，三条边分别为 $31, 41, 59$ 和 $33, 83, 27$，那么整张 DAG 应该如下图所示。

图中蓝色实线框所表示的是一个砖块拆解得到的一组砖块，之所以用 $\{\}$ 表示底面边长，是因为砖块一旦选取了高，底面边长就是无序的。

图中黄色虚线框表示的是重复计算部分，可以采用 记忆化搜索 的方法来避免重复计算。

转移

题目要求的是塔的最大高度，已经转化为最长路问题，其起点上文已指出是大地，那么终点呢？显然终点已经自然确定，那就是某砖块上不能再搭别的砖块的时候。

下面我们开始考虑转移方程。

设 $d(i, r)$ 表示第 i 块砖块在最上面，且采取第 r 种堆叠方式时的最大高度。那么有如下转移方程：

$$d(i, r) = \max\{d(j, r') + h\}$$

其中 j 是所有那些在砖块 i 以 r 方式堆叠时可放上的砖块，r' 对应 j 此时的摆放方式，h 对应砖块 i 采用第 r 种堆叠方式时的高度。

" **实现**"

```cpp
#include <cmath>
#include <cstring>
#include <iostream>
```

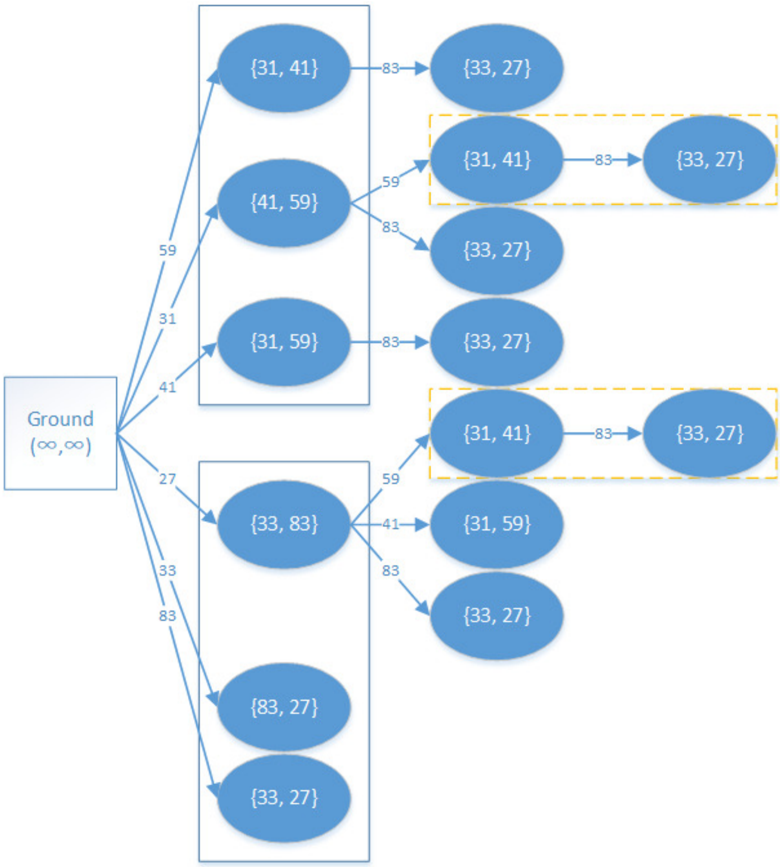


图 7.1

```
using namespace std;
#define MAXN (30 + 5)
#define MAXV (500 + 5)
int d[MAXN][3];
int x[MAXN], y[MAXN], z[MAXN];

int babylon_sub(int c, int rot, int n) {
  if (d[c][rot] != -1) {
    return d[c][rot];
  }
  d[c][rot] = 0;
  int base1, base2;
  if (rot == 0) {  // 处理三个方向
    base1 = x[c];
    base2 = y[c];
  }
  if (rot == 1) {
    base1 = y[c];
    base2 = z[c];
  }
  if (rot == 2) {
    base1 = x[c];
    base2 = z[c];
  }
  for (int i = 0; i < n; i++) {  // 根据不同条件，分别调用不同的递归
    if ((x[i] < base1 && y[i] < base2) || (y[i] < base1 && x[i] < base2))
```

```
      d[c][rot] = max(d[c][rot], babylon_sub(i, 0, n) + z[i]);
    if ((y[i] < base1 && z[i] < base2) || (z[i] < base1 && y[i] < base2))
      d[c][rot] = max(d[c][rot], babylon_sub(i, 1, n) + x[i]);
    if ((x[i] < base1 && z[i] < base2) || (z[i] < base1 && x[i] < base2))
      d[c][rot] = max(d[c][rot], babylon_sub(i, 2, n) + y[i]);
  }
  return d[c][rot];
}

int babylon(int n) {
  for (int i = 0; i < n; i++) {
    d[i][0] = -1;
    d[i][1] = -1;
    d[i][2] = -1;
  }
  int r = 0;
  for (int i = 0; i < n; i++) {  // 三种建法
    r = max(r, babylon_sub(i, 0, n) + z[i]);
    r = max(r, babylon_sub(i, 1, n) + x[i]);
    r = max(r, babylon_sub(i, 2, n) + y[i]);
  }
  return r;
}

int main() {
  int t = 0;
  while (true) {  // 死循环求答案
    int n;
    cin >> n;
    if (n == 0) break;  // 没有砖头了就停止
    t++;
    for (int i = 0; i < n; i++) {
      cin >> x[i] >> y[i] >> z[i];
    }
    cout << "Case " << t << ":"
         << " maximum height = " << babylon(n);  // 递归
    cout << endl;
  }
  return 0;
}
```

参考资料与注释

[1] UVa 437 巴比伦塔 The Tower of Babylon

7.7 树形 DP

树形 DP，即在树上进行的 DP。由于树固有的递归性质，树形 DP 一般都是递归进行的。

基础

以下面这道题为例，介绍一下树形 DP 的一般过程。

" 例题 洛谷 P1352 没有上司的舞会[2]"

某大学有 n 个职员，编号为 $1 \sim N$。他们之间有从属关系，也就是说他们的关系就像一棵以校长为根的树，父结点就是子结点的直接上司。现在有个周年庆宴会，宴会每邀请来一个职员都会增加一定的快乐指数 a_i，但是呢，如果某个职员的直接上司来参加舞会了，那么这个职员就无论如何也不肯来参加舞会了。所以，请你编程计算，邀请哪些职员可以使快乐指数最大，求最大的快乐指数。

我们设 $f(i,0/1)$ 代表以 i 为根的子树的最优解（第二维的值为 0 代表 i 不参加舞会的情况，1 代表 i 参加舞会的情况）。

对于每个状态，都存在两种决策（其中下面的 x 都是 i 的儿子）：

- 上司不参加舞会时，下属可以参加，也可以不参加，此时有 $f(i,0) = \sum \max\{f(x,1), f(x,0)\}$；
- 上司参加舞会时，下属都不会参加，此时有 $f(i,1) = \sum f(x,0) + a_i$。

我们可以通过 DFS，在返回上一层时更新当前结点的最优解。

```
#include <algorithm>
#include <cstdio>
using namespace std;

struct edge {
  int v, next;
} e[6005];

int head[6005], n, cnt, f[6005][2], ans, is_h[6005], vis[6005];

void addedge(int u, int v) {  // 建图
  e[++cnt].v = v;
  e[cnt].next = head[u];
  head[u] = cnt;
}

void calc(int k) {
  vis[k] = 1;
  for (int i = head[k]; i; i = e[i].next) {  // 枚举该结点的每个子结点
    if (vis[e[i].v]) continue;
    calc(e[i].v);
    f[k][1] += f[e[i].v][0];
    f[k][0] += max(f[e[i].v][0], f[e[i].v][1]);  // 转移方程
  }
  return;
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) scanf("%d", &f[i][1]);
  for (int i = 1; i < n; i++) {
    int l, k;
    scanf("%d%d", &l, &k);
    is_h[l] = 1;
    addedge(k, l);
  }
  for (int i = 1; i <= n; i++)
    if (!is_h[i]) {  // 从根结点开始 DFS
```

```
    calc(i);
    printf("%d", max(f[i][1], f[i][0]));
    return 0;
  }
}
```

习题

- HDU 2196 Computer[3]
- POJ 1463 Strategic game[4]
- [POI2014]FAR-FarmCraft[5]

树上背包

树上的背包问题，简单来说就是背包问题与树形 DP 的结合。

" 例题 洛谷 P2014 CTSC1997 选课[6]"

现在有 n 门课程，第 i 门课程的学分为 a_i，每门课程有零门或一门先修课，有先修课的课程需要先学完其先修课，才能学习该课程。

一位学生要学习 m 门课程，求其能获得的最多学分数。

$n, m \leq 300$

每门课最多只有一门先修课的特点，与有根树中一个点最多只有一个父亲结点的特点类似。

因此可以想到根据这一性质建树，从而所有课程组成了一个森林的结构。为了方便起见，我们可以新增一门 0 学分的课程（设这个课程的编号为 0），作为所有无先修课课程的先修课，这样我们就将森林变成了一棵以 0 号课程为根的树。

我们设 $f(u,i,j)$ 表示以 u 号点为根的子树中，已经遍历了 u 号点的前 i 棵子树，选了 j 门课程的最大学分。

转移的过程结合了树形 DP 和 背包 DP 的特点，我们枚举 u 点的每个子结点 v，同时枚举以 v 为根的子树选了几门课程，将子树的结果合并到 u 上。

记点 x 的儿子个数为 s_x，以 x 为根的子树大小为 siz_x，可以写出下面的状态转移方程：

$$f(u,i,j)=\max_{v,k\leq j,k\leq siz\_v} f(u,i-1,j-k)+f(v,s_v,k)$$

注意上面状态转移方程中的几个限制条件，这些限制条件确保了一些无意义的状态不会被访问到。

f 的第二维可以很轻松地用滚动数组的方式省略掉，注意这时需要倒序枚举 j 的值。

可以证明，该做法的时间复杂度为 $O(nm)$[1]。

" 参考代码"

```
#include <algorithm>
#include <cstdio>
#include <vector>
using namespace std;
int f[305][305], s[305], n, m;
vector<int> e[305];

int dfs(int u) {
  int p = 1;
  f[u][1] = s[u];
  for (auto v : e[u]) {
```

```
    int siz = dfs(v);
    // 注意下面两重循环的上界和下界
    // 只考虑已经合并过的子树，以及选的课程数超过 m+1 的状态没有意义
    for (int i = min(p, m + 1); i; i--)
      for (int j = 1; j <= siz && i + j <= m + 1; j++)
        f[u][i + j] = max(f[u][i + j], f[u][i] + f[v][j]);  // 转移方程
    p += siz;
  }
  return p;
}

int main() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= n; i++) {
    int k;
    scanf("%d%d", &k, &s[i]);
    e[k].push_back(i);
  }
  dfs(0);
  printf("%d", f[0][m + 1]);
  return 0;
}
```

习题

- 「CTSC1997」 选课[6]
- 「JSOI2018」 潜入行动[7]
- 「SDOI2017」 苹果树[8]
- 「Codeforces Round 875 Div. 1」Problem D. Mex Tree[9]

换根 DP

树形 DP 中的换根 DP 问题又被称为二次扫描，通常不会指定根结点，并且根结点的变化会对一些值，例如子结点深度和、点权和等产生影响。

通常需要两次 DFS，第一次 DFS 预处理诸如深度，点权和之类的信息，在第二次 DFS 开始运行换根动态规划。

接下来以一些例题来带大家熟悉这个内容。

" 例题 [POI2008]STA-Station[10] "

给定一个 n 个点的树，请求出一个结点，使得以这个结点为根时，所有结点的深度之和最大。

不妨令 u 为当前结点，v 为当前结点的子结点。首先需要用 s_i 来表示以 i 为根的子树中的结点个数，并且有 $s_u = 1 + \sum s_v$。显然需要一次 DFS 来计算所有的 s_i，这次的 DFS 就是预处理，我们得到了以某个结点为根时其子树中的结点总数。

考虑状态转移，这里就是体现 " 换根 " 的地方了。令 f_u 为以 u 为根时，所有结点的深度之和。

$f_v \leftarrow f_u$ 可以体现换根，即以 u 为根转移到以 v 为根。显然在换根的转移过程中，以 v 为根或以 u 为根会导致其子树中的结点的深度产生改变。具体表现为：

- 所有在 v 的子树上的结点深度都减少了一，那么总深度和就减少了 s_v；
- 所有不在 v 的子树上的结点深度都增加了一，那么总深度和就增加了 $n - s_v$；

根据这两个条件就可以推出状态转移方程 $f_v = f_u - s_v + n - s_v = f_u + n - 2 \times s_v$。

于是在第二次 DFS 遍历整棵树并状态转移 $f_v = f_u + n - 2 \times s_v$，那么就能求出以每个结点为根时的深度和了。

最后只需要遍历一次所有根结点深度和就可以求出答案。

" 参考代码"

```cpp
#include <bits/stdc++.h>
using namespace std;

int head[1000010 << 1], tot;
long long n, sz[1000010], dep[1000010];
long long f[1000010];

struct node {
  int to, next;
} e[1000010 << 1];

void add(int u, int v) {  // 建图
  e[++tot] = {v, head[u]};
  head[u] = tot;
}

void dfs(int u, int fa) {  // 预处理 dfs
  sz[u] = 1;
  dep[u] = dep[fa] + 1;
  for (int i = head[u]; i; i = e[i].next) {
    int v = e[i].to;
    if (v != fa) {
      dfs(v, u);
      sz[u] += sz[v];
    }
  }
}

void get_ans(int u, int fa) {  // 第二次 dfs 换根 dp
  for (int i = head[u]; i; i = e[i].next) {
    int v = e[i].to;
    if (v != fa) {
      f[v] = f[u] - sz[v] * 2 + n;
      get_ans(v, u);
    }
  }
}

int main() {
  scanf("%lld", &n);
  int u, v;
  for (int i = 1; i <= n - 1; i++) {
    scanf("%d%d", &u, &v);
    add(u, v);
    add(v, u);
  }
  dfs(1, 1);
  for (int i = 1; i <= n; i++) f[1] += dep[i];
  get_ans(1, 1);
  long long int ans = -1;
```

```
  int id;
  for (int i = 1; i <= n; i++) {  // 统计答案
    if (f[i] > ans) {
      ans = f[i];
      id = i;
    }
  }
  printf("%d\n", id);
  return 0;
}
```

习题

- Atcoder Educational DP Contest, Problem V, Subtree[11]
- Educational Codeforces Round 67, Problem E, Tree Painting[12]
- POJ 3585 Accumulation Degree[13]
- [USACO10MAR]Great Cow Gathering G[14]
- CodeForce 708C Centroids[15]

参考资料与注释

[1] 子树合并背包类型的 dp 的复杂度证明 - LYD729 的 CSDN 博客

[2] 洛谷 P1352 没有上司的舞会

[3] HDU 2196 Computer

[4] POJ 1463 Strategic game

[5] [POI2014]FAR-FarmCraft

[6] 洛谷 P2014 CTSC1997 选课 [6-1] [6-2]

[7] 「JSOI2018」潜入行动

[8] 「SDOI2017」苹果树

[9] 「Codeforces Round 875 Div. 1」Problem D. Mex Tree

[10] [POI2008]STA-Station

[11] Atcoder Educational DP Contest, Problem V, Subtree

[12] Educational Codeforces Round 67, Problem E, Tree Painting

[13] POJ 3585 Accumulation Degree

[14] [USACO10MAR]Great Cow Gathering G

[15] CodeForce 708C Centroids

7.8 状压 DP

定义

状压 DP 是动态规划的一种，通过将状态压缩为整数来达到优化转移的目的。

例题 1

"「SCOI2005」 互不侵犯[1]"

在 $N \times N$ 的棋盘里面放 K 个国王（$1 \leq N \leq 9, 1 \leq K \leq N \times N$），使他们互不攻击，共有多少种摆放方案。国王能攻击到它上下左右，以及左上左下右上右下八个方向上附近的各一个格子，共 8 个格子。

解释

设 $f(i,j,l)$ 表示前 i 行，第 i 行的状态为 j，且棋盘上已经放置 l 个国王时的合法方案数。

对于编号为 j 的状态，我们用二进制整数 $sit(j)$ 表示国王的放置情况，$sit(j)$ 的某个二进制位为 0 表示对应位置不放国王，为 1 表示在对应位置上放置国王；用 $sta(j)$ 表示该状态的国王个数，即二进制数 $sit(j)$ 中 1 的个数。例如，如下图所示的状态可用二进制数 100101 来表示（棋盘左边对应二进制低位），则有 $sit(j) = 100101_{(2)} = 37, sta(j) = 3$。

图 7.2

设当前行的状态为 j，上一行的状态为 x，可以得到下面的状态转移方程：$f(i,j,l) = \sum f(i-1,x,l-sta(j))$。

设上一行的状态编号为 x，在保证当前行和上一行不冲突的前提下，枚举所有可能的 x 进行转移，转移方程：

$$f(i,j,l) = \sum f(i-1,x,l-sta(j))$$

实现

"参考代码"

```cpp
#include <algorithm>
#include <iostream>
using namespace std;
long long sta[2005], sit[2005], f[15][2005][105];
int n, k, cnt;

void dfs(int x, int num, int cur) {
  if (cur >= n) {  // 有新的合法状态
    sit[++cnt] = x;
    sta[cnt] = num;
```

```
    return;
  }
  dfs(x, num, cur + 1);  // cur 位置不放国王
  dfs(x + (1 << cur), num + 1,
      cur + 2);  // cur 位置放国王，与它相邻的位置不能再放国王
}

bool compatible(int j, int x) {
  if (sit[j] & sit[x]) return false;
  if ((sit[j] << 1) & sit[x]) return false;
  if (sit[j] & (sit[x] << 1)) return false;
  return true;
}

int main() {
  cin >> n >> k;
  dfs(0, 0, 0);  // 先预处理一行的所有合法状态
  for (int j = 1; j <= cnt; j++) f[1][j][sta[j]] = 1;
  for (int i = 2; i <= n; i++)
    for (int j = 1; j <= cnt; j++)
      for (int x = 1; x <= cnt; x++) {
        if (!compatible(j, x)) continue;  // 排除不合法转移
        for (int l = sta[j]; l <= k; l++) f[i][j][l] += f[i - 1][x][l - sta[j]];
      }
  long long ans = 0;
  for (int i = 1; i <= cnt; i++) ans += f[n][i][k];  // 累加答案
  cout << ans << endl;
  return 0;
}
```

例题 2

"[POI2004] PRZ[2]"

有 n 个人需要过桥，第 i 的人的重量为 w_i，过桥用时为 t_i. 这些人过桥时会分成若干组，只有在某一组的所有人全部过桥后，其余的组才能过桥。桥最大承重为 W，问这些人全部过桥的最短时间。

$100 \le W \le 400$，$1 \le n \le 16$，$1 \le t_i \le 50$，$10 \le w_i \le 100$.

解释

我们用 S 表示所有人构成集合的一个子集，设 $t(S)$ 表示 S 中人的最长过桥时间，$w(S)$ 表示 S 中所有人的总重量，$f(S)$ 表示 S 中所有人全部过桥的最短时间，则：

$$\begin{cases} f(\emptyset) = 0, \\ f(S) = \min\limits_{T \subseteq S;\ w(T) \le W} \left\{ t(T) + f(S\ \ T) \right\}. \end{cases}$$

需要注意的是这里不能直接枚举集合再判断是否为子集，而应使用 子集枚举，从而使时间复杂度为 $O(3^n)$.

实现

" 参考代码"

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);
  int W, n;
  cin >> W >> n;
  const int S = (1 << n) - 1;
  vector<int> ts(S + 1), ws(S + 1);
  for (int j = 0, t, w; j < n; ++j) {
    cin >> t >> w;
    for (int i = 0; i <= S; ++i)
      if (i & (1 << j)) {
        ts[i] = max(ts[i], t);
        ws[i] += w;
      }
  }
  vector<int> dp(S + 1, numeric_limits<int>::max() / 2);
  for (int i = 0; i <= S; ++i) {
    if (ws[i] <= W) dp[i] = ts[i];
    for (int j = i; j; j = i & (j - 1))
      if (ws[i ^ j] <= W) dp[i] = min(dp[i], dp[j] + ts[i ^ j]);
  }
  cout << dp[S] << '\n';
  return 0;
}
```

习题

- 「NOI2001」 炮兵阵地[3]
- 「USACO06NOV」 玉米田 Corn Fields[4]
- 「 九省联考 2018」 一双木棋[5]

参考资料与注释

[1] 「SCOI2005」互不侵犯

[2] [POI2004] PRZ

[3] 「NOI2001」炮兵阵地

[4] 「USACO06NOV」玉米田 Corn Fields

[5] 「九省联考 2018」一双木棋

7.9 数位 DP

本页面将简要介绍数位 DP。

引入

数位是指把一个数字按照个、十、百、千等等一位一位地拆开，关注它每一位上的数字。如果拆的是十进制数，那么每一位数字都是 0~9，其他进制可类比十进制。

数位 DP：用来解决一类特定问题，这种问题比较好辨认，一般具有这几个特征：

1. 要求统计满足一定条件的数的数量（即，最终目的为计数）；
2. 这些条件经过转化后可以使用「数位」的思想去理解和判断；
3. 输入会提供一个数字区间（有时也只提供上界）来作为统计的限制；
4. 上界很大（比如 10^{18}），暴力枚举验证会超时。

数位 DP 的基本原理：

考虑人类计数的方式，最朴素的计数就是从小到大开始依次加一。但我们发现对于位数比较多的数，这样的过程中有许多重复的部分。例如，从 7000 数到 7999、从 8000 数到 8999、和从 9000 数到 9999 的过程非常相似，它们都是后三位从 000 变到 999，不一样的地方只有千位这一位，所以我们可以把这些过程归并起来，将这些过程中产生的计数答案也都存在一个通用的数组里。此数组根据题目具体要求设置状态，用递推或 DP 的方式进行状态转移。

数位 DP 中通常会利用常规计数问题技巧，比如把一个区间内的答案拆成两部分相减（即 $ans_{[l,r]} = ans_{[0,r]} - ans_{[0,l-1]}$）

那么有了通用答案数组，接下来就是统计答案。统计答案可以选择记忆化搜索，也可以选择循环迭代递推。为了不重不漏地统计所有不超过上限的答案，要从高到低枚举每一位，再考虑每一位都可以填哪些数字，最后利用通用答案数组统计答案。

接下来我们具体看几道题目。

例题一

" **例 1 Luogu P2602 数字计数**[1]"

题目大意：给定两个正整数 a, b，求在 $[a, b]$ 中的所有整数中，每个数码（digit）各出现了多少次。

方法一

解释

发现对于满 i 位的数，所有数字出现的次数都是相同的，故设数组 dp_i 为满 i 位的数中每个数字出现的次数，此时暂时不处理前导零。则有 $dp_i = 10 \times dp_{i-1} + 10^{i-1}$，这两部分前一个是来自前 $i-1$ 位数字的贡献，后一个是来自第 i 位的数字的贡献。

有了 dp 数组，我们来考虑如何统计答案。将上界按位分开，从高到低枚举，不贴着上界时，后面可以随便取值。贴着上界时，后面就只能取 0 到上界，分两部分分别计算贡献。最后考虑下前导零，第 i 位为前导 0 时，此时 1 到 $i-1$ 位也都是 0，也就是多算了将 $i-1$ 位填满的答案，需要额外减去。

实现

" **参考代码**"

```
#include <bits/stdc++.h>
```

```cpp
using namespace std;
const int N = 15;
typedef long long ll;
ll l, r, dp[N], mi[N];
ll ans1[N], ans2[N];
int a[N];

void solve(ll n, ll *ans) {
  ll tmp = n;
  int len = 0;
  while (n) a[++len] = n % 10, n /= 10;
  for (int i = len; i >= 1; --i) {
    for (int j = 0; j < 10; j++) ans[j] += dp[i - 1] * a[i];
    for (int j = 0; j < a[i]; j++) ans[j] += mi[i - 1];
    tmp -= mi[i - 1] * a[i], ans[a[i]] += tmp + 1;
    ans[0] -= mi[i - 1];
  }
}

int main() {
  scanf("%lld%lld", &l, &r);
  mi[0] = 1ll;
  for (int i = 1; i <= 13; ++i) {
    dp[i] = dp[i - 1] * 10 + mi[i - 1];
    mi[i] = 10ll * mi[i - 1];
  }
  solve(r, ans1), solve(l - 1, ans2);
  for (int i = 0; i < 10; ++i) printf("%lld ", ans1[i] - ans2[i]);
  return 0;
}
```

方法二

解释

此题也可以使用记忆化搜索。dp_i 表示不贴上限、无前导零时，位数为 i 的答案。

详见代码注释

过程

" 参考代码"

```cpp
#include <cstdio>  //code by Alphnia
#include <cstring>
#include <iostream>
using namespace std;
#define N 50005
#define ll long long
ll a, b;
ll f[15], ksm[15], p[15], now[15];

ll dfs(int u, int x, bool f0,
       bool lim) {  // u 表示位数，f0 是否有前导零，lim 是否都贴在上限上
  if (!u) {
```

```
    if (f0) f0 = 0;
    return 0;
  }
  if (!lim && !f0 && (~f[u])) return f[u];
  ll cnt = 0;
  int lst = lim ? p[u] : 9;
  for (int i = 0; i <= lst; i++) {  // 枚举这位要填的数字
    if (f0 && i == 0)
      cnt += dfs(u - 1, x, 1, lim && i == lst);  // 处理前导零
    else if (i == x && lim && i == lst)
      cnt += now[u - 1] + 1 +
             dfs(u - 1, x, 0,
                 lim && i == lst);  // 此时枚举的前几位都贴在给定的上限上。
    else if (i == x)
      cnt += ksm[u - 1] + dfs(u - 1, x, 0, lim && i == lst);
    else
      cnt += dfs(u - 1, x, 0, lim && i == lst);
  }
  if ((!lim) && (!f0)) f[u] = cnt;  // 只有不贴着上限和没有前导零才能记忆
  return cnt;
}

ll gans(ll d, int dig) {
  int len = 0;
  memset(f, -1, sizeof(f));
  while (d) {
    p[++len] = d % 10;
    d /= 10;
    now[len] = now[len - 1] + p[len] * ksm[len - 1];
  }
  return dfs(len, dig, 1, 1);
}

int main() {
  scanf("%lld%lld", &a, &b);
  ksm[0] = 1;
  for (int i = 1; i <= 12; i++) ksm[i] = ksm[i - 1] * 10ll;
  for (int i = 0; i < 9; i++) printf("%lld ", gans(b, i) - gans(a - 1, i));
  printf("%lld\n", gans(b, 9) - gans(a - 1, 9));
  return 0;
}
```

例题二

" 例 2 hdu 2089 不要 62[2]"

题面大意：统计一个区间内数位上不能有 4 也不能有连续的 62 的数有多少。

解释

没有 4 的话在枚举的时候判断一下，不枚举 4 就可以保证状态合法了，所以这个约束没有记忆化的必要，而对于 62 的话，涉及到两位，当前一位是 6 或者不是 6 这两种不同情况计数是不相同的，所以要用状态来记录不同的方案数。$dp_{pos,sta}$ 表示当前第 pos 位，前一位是否是 6 的状态，这里 sta 只需要取 0 和 1 两种状态就可以了，不是 6 的情况可视为同种，不会影响计数。

实现

" 参考代码"

```
#include <cstdio>  //code by Alphnia
#include <cstring>
#include <iostream>
using namespace std;
int x, y, dp[15][3], p[50];

int pre() {
  memset(dp, 0, sizeof(dp));
  dp[0][0] = 1;
  for (int i = 1; i <= 10; i++) {
    dp[i][0] = dp[i - 1][0] * 9 - dp[i - 1][1];
    dp[i][1] = dp[i - 1][0];
    dp[i][2] = dp[i - 1][2] * 10 + dp[i - 1][1] + dp[i - 1][0];
  }
}

int cal(int x) {
  int cnt = 0, ans = 0, tmp = x;
  while (x) {
    p[++cnt] = x % 10;
    x /= 10;
  }
  bool flag = 0;
  p[cnt + 1] = 0;
  for (int i = cnt; i; i--) {  // 从高到低枚举数位
    ans += p[i] * dp[i - 1][2];
    if (flag)
      ans += p[i] * dp[i - 1][0];
    else {
      if (p[i] > 4) ans += dp[i - 1][0];
      if (p[i] > 6) ans += dp[i - 1][1];
      if (p[i] > 2 && p[i + 1] == 6) ans += dp[i][1];
      if (p[i] == 4 || (p[i] == 2 && p[i + 1] == 6)) flag = 1;
    }
  }
  return tmp - ans;
}

int main() {
  pre();
  while (~scanf("%d%d", &x, &y)) {
    if (!x && !y) break;
    x = min(x, y), y = max(x, y);
    printf("%d\n", cal(y + 1) - cal(x));
  }
  return 0;
}
```

例题三

" 例 3 SCOI2009 windy 数[3]"

题目大意：给定一个区间 $[l,r]$，求其中满足条件**不含前导** 0 **且相邻两个数字相差至少为** 2 的数字个数。

解释

首先我们将问题转化成更加简单的形式。设 ans_i 表示在区间 $[1,i]$ 中满足条件的数的数量，那么所求的答案就是 $ans_r - ans_{l-1}$。

对于一个小于 n 的数，它从高到低肯定出现某一位，使得这一位上的数值小于 n 这一位上对应的数值。而之前的所有位都和 n 上的位相等。

有了这个性质，我们可以定义 $f(i,st,op)$ 表示当前将要考虑的是从高到低的第 i 位，当前该前缀的状态为 st 且前缀和当前求解的数字的大小关系是 op（$op=1$ 表示等于，$op=0$ 表示小于）时的数字个数。在本题中，这个前缀的状态就是上一位的值，因为当前将要确定的位不能取哪些数只和上一位有关。在其他题目中，这个值可以是：前缀的数字和，前缀所有数字的 gcd，该前缀取模某个数的余数，也有两种或多种合用的情况。

写出**状态转移方程**：$f(i,st,op)=\sum_{k=1}^{maxx} f(i+1,k,op=1 \text{ and } k=maxx) \quad (|st-k|\geq 2)$

这里的 k 就是当前枚举的下一位的值，而 $maxx$ 就是当前能取到的最高位。因为如果 $op=1$，那么你在这一位上取的值一定不能大于求解的数字上该位的值，否则则没有限制。

我们发现，尽管前缀所选择的状态不同，而 f 的三个参数相同，答案就是一样的。为了防止这个答案被计算多次，可以使用 记忆化搜索 的方式实现。

实现

" 参考代码"

```
int dfs(int x, int st, int op)  // op=1 =;op=0 <
{
  if (!x) return 1;
  if (!op && ~f[x][st]) return f[x][st];
  int maxx = op ? dim[x] : 9, ret = 0;
  for (int i = 0; i <= maxx; i++) {
    if (abs(st - i) < 2) continue;
    if (st == 11 && i == 0)
      ret += dfs(x - 1, 11, op & (i == maxx));
    else
      ret += dfs(x - 1, i, op & (i == maxx));
  }
  if (!op) f[x][st] = ret;
  return ret;
}

int solve(int x) {
  memset(f, -1, sizeof f);
  dim.clear();
  dim.push_back(-1);
  int t = x;
  while (x) {
    dim.push_back(x % 10);
    x /= 10;
```

```
  }
  return dfs(dim.size() - 1, 11, 1);
}
```

例题四

" 例 4.SPOJMYQ10[4]"

题面大意：假如手写下 $[n,m]$ 之间所有整数，会有多少数看起来和在镜子里看起来一模一样？（$n,m<10^{44}, T<10^5$）

解释

注：由于这里考虑到的镜像，只有 $0,1,8$ 的镜像是自己本身。所以，这里的「一模一样」并不是传统意义上的回文串，而是只含有 $0,1,8$ 的回文串。

首先，在数位 DP 过程中，显然只有 $0,1,8$ 能被选中。

其次，由于数值超过 long long 范围，所以 $[n,m]=[1,m]-[1,n-1]$ 不再适用（高精度比较繁琐），而是需要对 n 是否合法进行判断，得出：$[n,m]=[1,m]-[1,n]+\text{check}(n)$。

镜像解决了，如何判断回文？

我们需要用一个小数组记录一下之前的值。在未超过一半的长度时，只要不超上限就行；在超过一半的长度时，还需要判断是否和与之「镜面对称」的位相等。

需要额外注意的是，这道题的记忆化部分，不能用 `memset`，否则会导致超时。

实现

" 参考代码"

```
int check(char cc[]) {  // n 的特判
  int strc = strlen(cc);
  for (int i = 0; i < strc; ++i) {
    if (!(cc[i] == cc[strc - i - 1] &&
          (cc[i] == '1' || cc[i] == '8' || cc[i] == '0')))
      return 0ll;
  }
  return 1ll;
}

// now: 当前位, eff: 有效位, fulc: 是否全顶格, ful0: 是否全 0
int dfs(int now, int eff, bool ful0, bool fulc) {
  if (now == 0) return 1ll;
  if (!fulc && f[now][eff][ful0] != -1)  // 记忆化
    return f[now][eff][ful0];

  int res = 0, maxk = fulc ? dig[now] : 9;
  for (int i = 0; i <= maxk; ++i) {
    if (i != 0 && i != 1 && i != 8) continue;
    b[now] = i;
    if (ful0 && i == 0)  // 全前导 0
      res += dfs(now - 1, eff - 1, 1, 0);
    else if (now > eff / 2)                                // 未过半程
```

```
      res += dfs(now - 1, eff, 0, fulc && (dig[now] == i));  // 已过半程
    else if (b[now] == b[eff - now + 1])
      res += dfs(now - 1, eff, 0, fulc && (dig[now] == i));
  }
  if (!fulc) f[now][eff][ful0] = res;
  return res;
}

char cc1[100], cc2[100];
int strc, ansm, ansn;

int get(char cc[]) {  // 处理封装
  strc = strlen(cc);
  for (int i = 0; i < strc; ++i) dig[strc - i] = cc[i] - '0';
  return dfs(strc, strc, 1, 1);
}

scanf("%s%s", cc1, cc2);
printf("%lld\n", get(cc2) - get(cc1) + check(cc1));
```

例题五

" 例 5.P3311 数数[5]"

题面：我们称一个正整数 x 是幸运数，当且仅当它的十进制表示中不包含数字串集合 S 中任意一个元素作为其子串。例如当 $S=\{22,333,0233\}$ 时，233233 是幸运数，23332333、2023320233、32233223 不是幸运数。给定 n 和 S，计算不大于 n 的幸运数个数。答案对 10^9+7 取模。

$1 \le n < 10^{1201}$ $1 \le m \le 100$ $1 \le \sum_{i=1}^{m} |s_i| \le 1500$ $\min_{i=1}^{m} |s_i| \ge 1$，其中 $|s_i|$ 表示字符串 s_i 的长度。n 没有前导 0，但是 s_i 可能有前导 0。

解释

阅读题面发现，如果将数字看成字符串，那么这就是需要完成一个多模匹配，自然而然就想到 AC 自动机。普通数位 DP 中，先从高到低枚举数位，再枚举每一位都填什么，在这道题中，我们也就自然地转化为枚举已经填好的位数，再枚举此时停在 AC 自动机上的哪个节点，然后从当前节点转移到它在 AC 自动机上的子节点。

设 $f(i,j,0/1)$ 表示当前从高到低已经填了 i 位（即在 AC 自动机上走过了 i 条边），此时停在标号为 j 的节点上，当前是否正好贴着上界。

至于题目中的「不包含」条件，只需在 AC 自动机上给每个模式串的结尾节点都打上标记，DP 过程中一旦遇上这些结尾节点就跳过即可。

转移很好想，详见代码主函数部分。

实现

" 参考代码"

```
#include <bits/stdc++.h>  //code by Alphnia
using namespace std;
#define N 1505
#define ll long long
#define mod 1000000007
int n, m;
```

```
char s[N], c[N];
int ch[N][10], fail[N], ed[N], tot, len;

void insert() {
  int now = 0;
  int L = strlen(s);
  for (int i = 0; i < L; ++i) {
    if (!ch[now][s[i] - '0']) ch[now][s[i] - '0'] = ++tot;
    now = ch[now][s[i] - '0'];
  }
  ed[now] = 1;
}

queue<int> q;

void build() {
  for (int i = 0; i < 10; ++i)
    if (ch[0][i]) q.push(ch[0][i]);
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    for (int i = 0; i < 10; ++i) {
      if (ch[u][i]) {
        fail[ch[u][i]] = ch[fail[u]][i], q.push(ch[u][i]),
        ed[ch[u][i]] |= ed[fail[ch[u][i]]];
      } else
        ch[u][i] = ch[fail[u]][i];
    }
  }
  ch[0][0] = 0;
}

ll f[N][N][2], ans;

void add(ll &x, ll y) { x = (x + y) % mod; }

int main() {
  scanf("%s", c);
  n = strlen(c);
  scanf("%d", &m);
  for (int i = 1; i <= m; ++i) scanf("%s", s), insert();
  build();
  f[0][0][1] = 1;
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j <= tot; ++j) {
      if (ed[j]) continue;
      for (int k = 0; k < 10; ++k) {
        if (ed[ch[j][k]]) continue;
        add(f[i + 1][ch[j][k]][0], f[i][j][0]);
        if (k < c[i] - '0') add(f[i + 1][ch[j][k]][0], f[i][j][1]);
        if (k == c[i] - '0') add(f[i + 1][ch[j][k]][1], f[i][j][1]);
      }
    }
  }
```

```
  for (int j = 0; j <= tot; ++j) {
    if (ed[j]) continue;
    add(ans, f[n][j][0]);
    add(ans, f[n][j][1]);
  }
  printf("%lld\n", ans - 1);
  return 0;
}
```

此题可以很好地帮助理解数位 DP 的原理。

习题

Ahoi2009 self　同类分布[6]

洛谷　P3413 SAC#1 -　萌数[7]

HDU 6148 Valley Number[8]

CF55D Beautiful numbers[9]

CF628D Magic Numbers[10]

参考资料与注释

[1] Luogu P2602 数字计数

[2] hdu 2089 不要 62

[3] SCOI2009 windy 数

[4] SPOJMYQ10

[5] P3311 数数

[6] Ahoi2009 self 同类分布

[7] 洛谷 P3413 SAC#1 - 萌数

[8] HDU 6148 Valley Number

[9] CF55D Beautiful numbers

[10] CF628D Magic Numbers

7.10　插头 DP

定义

有些 状压　DP 问题要求我们记录状态的连通性信息，这类问题一般被形象的称为插头 DP 或连通性状态压缩 DP。例如格点图的哈密顿路径计数，求棋盘的黑白染色方案满足相同颜色之间形成一个连通块的方案数，以及特定

图的生成树计数等等。这些问题通常需要我们对状态的连通性进行编码，讨论状态转移过程中连通性的变化。

引入

骨牌覆盖与轮廓线 DP

温故而知新，在开始学习插头 DP 之前，不妨先让我们回顾一个经典问题。

" **例题** 「HDU 1400」Mondriaan’s Dream[1]"

题目大意：在 $N \times M$ 的棋盘内铺满 1×2 或 2×1 的多米诺骨牌，求方案数。

当 n 或 m 规模不大的时候，这类问题可以使用 状压 DP 解决。逐行划分阶段，设 $dp(i,s)$ 表示当前已考虑过前 i 行，且第 i 行的状态为 s 的方案数。这里的状态 s 的每一位可以表示这个这个位置是否已被上一行覆盖。

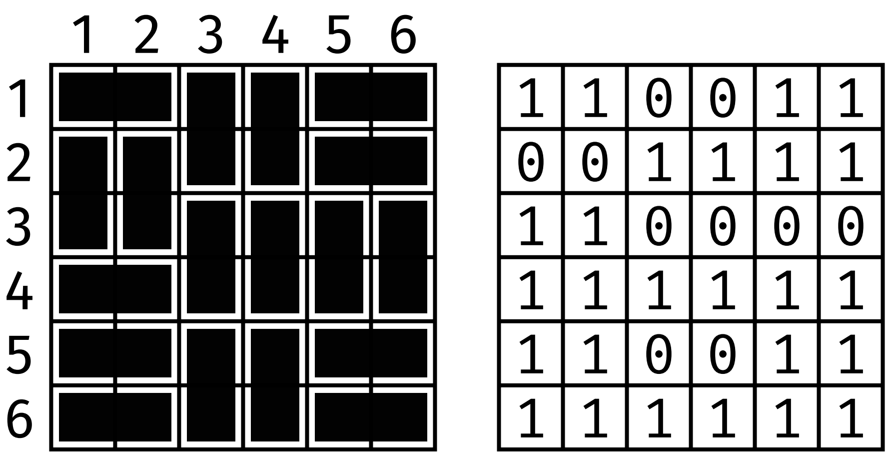


图 7.3 domino

另一种划分阶段的方法是逐格 DP，或者称之为轮廓线 DP。$dp(i,j,s)$ 表示已经考虑到第 i 行第 j 列，且当前轮廓线上的状态为 s 的方案数。

虽然逐格 DP 中我们的状态增加了一个维度，但是转移的时间复杂度减少为 $O(1)$，所以时间复杂度未变。我们用 f_0 表示当前阶段的状态，用 f_1 表示下一阶段的状态，$u = f_0(s)$ 表示当前枚举的函数值，那么有如下的状态转移方程：

```
if (s >> j & 1) {       // 如果已被覆盖
  f1[s ^ 1 << j] += u;  // 不放
} else {                // 如果未被覆盖
  if (j != m - 1 && (!(s >> j + 1 & 1))) f1[s ^ 1 << j + 1] += u;  // 横放
  f1[s ^ 1 << j] += u;                                             // 竖放
}
```

观察到这里不放和竖放的方程可以合并。

" **实现**"

```
#include <bits/stdc++.h>
using namespace std;
const int N = 11;
long long f[2][1 << N], *f0, *f1;
int n, m;
```

```
int main() {
  while (cin >> n >> m && n) {
    f0 = f[0];
    f1 = f[1];
    fill(f1, f1 + (1 << m), 0);
    f1[0] = 1;
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < m; ++j) {
        swap(f0, f1);
        fill(f1, f1 + (1 << m), 0);
#define u f0[s]
        for (int s = 0; s < 1 << m; ++s)
          if (u) {
            if (j != m - 1 && (!(s >> j & 3))) f1[s ^ 1 << j + 1] += u;  // 横放
            f1[s ^ 1 << j] += u;  // 竖放或不放
          }
      }
    }
    cout << f1[0] << endl;
  }
}
```

" **习题** 「SRM 671. Div 1 900」BearDestroys[2]"

题目大意：给定 $n \times m$ 的矩阵，每个格子有 `E` 或 `S`。对于一个矩阵，有一个计分方案。按照行优先的规则扫描每个格子，如果这个格子之前被骨牌占据，则 skip。否则尝试放多米诺骨牌。如果放骨牌的方向在矩阵外或被其他骨牌占据，则放置失败，切换另一种方案或 skip。如果是 `E` 则优先放一个 1×2 的骨牌，如果是 `S` 则优先放一个 2×1 的骨牌。一个矩阵的得分为最后放的骨牌数。问所有 2^{nm} 种矩阵的得分的和。

术语

阶段：动态规划执行的顺序，后续阶段的结果只与前序阶段的结果有关（无后效性）。很多 DP 问题可以有多种划分阶段的方式。例如在背包问题中，我们通常既可以按照物品划分阶段，也可以按照背包容量划分阶段（外层循环先枚举什么）。而在多米诺骨牌问题中，我们可以按照行、列、格子以及对角线等特征划分阶段。

轮廓线：已决策状态和未决策状态的分界线。

插头：一个格子某个方向的插头存在，表示这个格子在这个方向与相邻格子相连。

路径模型

多条回路

例题

" **例题** 「HDU 1693」Eat the Trees[3]"

题目大意：求用若干条回路覆盖 $N \times M$ 棋盘的方案数，有些位置有障碍。

严格来说，多条回路问题并不属于插头 DP，因为我们只需要和上面的骨牌覆盖问题一样，记录插头是否存在，然后成对的合并和生成插头就可以了。

注意对于一个宽度为 m 的棋盘，轮廓线的宽度为 $m+1$，因为包含 m 个上插头，和 1 个左插头。注意，当一行迭代完成之后，最右边的左插头通常是不合法的状态，同时我们需要补上下一行第一个左插头，这需要我们调整当前轮廓线的状态，通常是所有状态进行左移，我们把这个操作称为滚动 `roll()`。

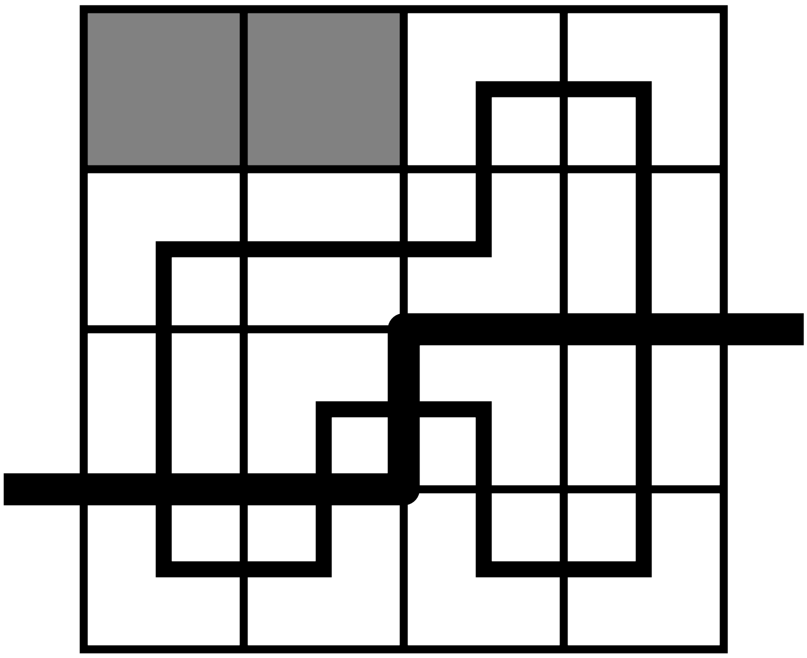


图 7.4 contour line

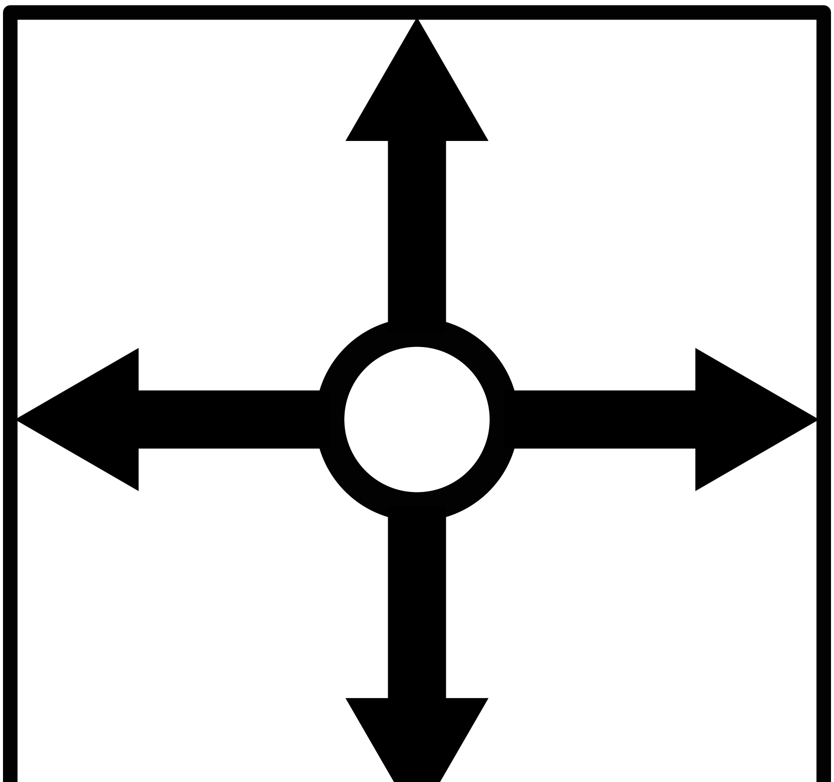


图 7.5 plug

例题代码

```
#include <bits/stdc++.h>
using namespace std;
const int N = 11;
long long f[2][1 << (N + 1)], *f0, *f1;
int n, m;

int main() {
  int T;
  cin >> T;
  for (int Case = 1; Case <= T; ++Case) {
    cin >> n >> m;
    f0 = f[0];
    f1 = f[1];
    fill(f1, f1 + (1 << m + 1), 0);
    f1[0] = 1;  // 初始化
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < m; ++j) {
        bool bad;
        cin >> bad;
        bad ^= 1;
        swap(f0, f1);
        fill(f1, f1 + (1 << m + 1), 0);
        for (int s = 0; s < 1 << m + 1; ++s)  // 具体的 dp 转移，上面都是初始化
          if (f0[s]) {
            bool lt = s >> j & 1, up = s >> j + 1 & 1;
            if (bad) {
              if (!lt && !up) f1[s] += f0[s];
            } else {
              f1[s ^ 3 << j] += f0[s];
              if (lt != up) f1[s] += f0[s];
            }
          }
      }
      swap(f0, f1);
      fill(f1, f1 + (1 << m + 1), 0);
      for (int s = 0; s < 1 << m; ++s) f1[s << 1] = f0[s];
    }
    printf("Case %d: There are %lld ways to eat the trees.\n", Case, f1[0]);
  }
}
```

习题

" 习题 「ZJU 4231」The Hive II[4]"

题目大意：同上题，但格子变成了六边形。

一条回路

例题

" **例题** 「Andrew Stankevich Contest 16 - Problem F」Pipe Layout[5]"

题目大意：求用一条回路覆盖 $N \times M$ 棋盘的方案数。

在上面的状态表示中我们每合并一组连通的插头，就会生成一条独立的回路，因而在本题中，我们还需要区分插头之间的连通性（出现了!）。这需要我们对状态进行额外的编码。

状态编码

通常的编码方案有括号表示和最小表示，这里着重介绍泛用性更好的最小表示。我们用长度 $m+1$ 的整形数组，记录轮廓线上每个插头的状态，0 表示没有插头，并约定连通的插头用相同的数字进行标记。

那么下面两组编码方式表示的是相同的状态：

- `0 3 1 0 1 3`
- `0 1 2 0 2 1`

我们将相同的状态都映射成字典序最小表示，例如在上例中的 `0 1 2 0 2 1` 就是一组最小表示。

我们用 `b[]` 数组表示轮廓线上插头的状态。`bb[]` 表示在最小表示的编码的过程中，每个数字被映射到的最小数字。注意 0 表示插头不存在，不能被映射成其他值。

" **代码实现**"

```
int b[M + 1], bb[M + 1];

int encode() {
  int s = 0;
  memset(bb, -1, sizeof(bb));
  int bn = 1;
  bb[0] = 0;
  for (int i = m; i >= 0; --i) {
#define bi bb[b[i]]
    if (!~bi) bi = bn++;
    s <<= offset;
    s |= bi;
  }
  return s;
}

void decode(int s) {
  REP(i, m + 1) {
    b[i] = s & mask;
    s >>= offset;
  }
}
```

我们注意到插头总是成对出现，成对消失的。因而 `0 1 2 0 1 2` 这样的状态是不合法的。合法的状态构成一组括号序列，实际中合法状态可能是非常稀疏的。

手写哈希

在一些 状压 DP 的问题中，合法的状态可能是稀疏的（例如本题），为了优化时空复杂度，我们可以使用哈希表存储合法的 DP 状态。对于 C++ 选手，我们可以使用 std::unordered_map[6]，当然也可以直接手写，这样可以灵活的将状态转移函数也封装于其中。

" 代码实现"

```
const int MaxSZ = 16796, Prime = 9973;

struct hashTable {
  int head[Prime], next[MaxSZ], sz;
  int state[MaxSZ];
  long long key[MaxSZ];

  void clear() {
    sz = 0;
    memset(head, -1, sizeof(head));
  }

  void push(int s) {
    int x = s % Prime;
    for (int i = head[x]; ~i; i = next[i]) {
      if (state[i] == s) {
        key[i] += d;
        return;
      }
    }
    state[sz] = s, key[sz] = d;
    next[sz] = head[x];
    head[x] = sz++;
  }

  void roll() { REP(i, sz) state[i] <<= offset; }
} H[2], *H0, *H1;
```

上面的代码中：

- MaxSZ 表示合法状态的上界，可以估计，也可以预处理出较为精确的值。
- Prime 一个小于 MaxSZ 的大素数。
- head[] 表头节点的指针。
- next[] 后续状态的指针。
- state[] 节点的状态。
- key[] 节点的关键字，在本题中是方案数。
- clear() 初始化函数，和手写邻接表类似，我们只需要初始化表头节点的指针。
- push() 状态转移函数，其中 d 是一个全局变量（偷懒），表示每次状态转移所带来的增量。如果找到的话就 +=，否则就创建一个状态为 s，关键字为 d 的新节点。
- roll() 迭代完一整行之后，滚动轮廓线。

关于哈希表的复杂度分析，以及开哈希和闭哈希的不同，可以参见《算法导论》中关于散列表的相关章节。

状态转移

" 代码实现"

```
REP(ii, H0->sz) {
  decode(H0->state[ii]);                    // 取出状态，并解码
```

```
    d = H0->key[ii];                        // 得到增量 delta
    int lt = b[j], up = b[j + 1];           // 左插头，上插头
    bool dn = i != n - 1, rt = j != m - 1;  // 下插头，右插头
    if (lt && up) {                         // 如果左、上均有插头
      if (lt == up) {                       // 来自同一个连通块
        if (i == n - 1 &&
            j == m - 1) {  // 只有在最后一个格子时，才能合并，封闭回路。
          push(j, 0, 0);
        }
      } else {  // 否则，必须合并这两个连通块，因为本题中需要回路覆盖
        REP(i, m + 1) if (b[i] == lt) b[i] = up;
        push(j, 0, 0);
      }
    } else if (lt || up) {  // 如果左、上之中有一个插头
      int t = lt | up;      // 得到这个插头
      if (dn) {             // 如果可以向下延伸
        push(j, t, 0);
      }
      if (rt) {  // 如果可以向右延伸
        push(j, 0, t);
      }
    } else {           // 如果左、上均没有插头
      if (dn && rt) {  // 生成一对新插头
        push(j, m, m);
      }
    }
  }
```

例题代码

```
#include <bits/stdc++.h>
using namespace std;
const int M = 10;
const int offset = 3, mask = (1 << offset) - 1;
int n, m;
long long ans, d;
const int MaxSZ = 16796, Prime = 9973;

struct hashTable {
  int head[Prime], next[MaxSZ], sz;
  int state[MaxSZ];
  long long key[MaxSZ];

  void clear() {
    sz = 0;
    memset(head, -1, sizeof(head));
  }

  void push(int s) {
    int x = s % Prime;
    for (int i = head[x]; ~i; i = next[i]) {
      if (state[i] == s) {
        key[i] += d;
        return;
```

```
      }
    }
    state[sz] = s, key[sz] = d;
    next[sz] = head[x];
    head[x] = sz++;
  }

  void roll() {
    for (int i = 0; i < sz; i++) state[i] <<= offset;
  }
} H[2], *H0, *H1;

int b[M + 1], bb[M + 1];

int encode() {
  int s = 0;
  memset(bb, -1, sizeof(bb));
  int bn = 1;
  bb[0] = 0;
  for (int i = m; i >= 0; --i) {
    if (!~bb[b[i]]) bb[b[i]] = bn++;
    s <<= offset;
    s |= bb[b[i]];
  }
  return s;
}

void decode(int s) {
  for (int i = 0; i < m + 1; i++) {
    b[i] = s & mask;
    s >>= offset;
  }
}

void push(int j, int dn, int rt) {
  b[j] = dn;
  b[j + 1] = rt;
  H1->push(encode());
}

int main() {
  cin >> n >> m;
  if (m > n) swap(n, m);
  H0 = H, H1 = H + 1;
  H1->clear();
  d = 1;
  H1->push(0);
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      swap(H0, H1);
      H1->clear();
      for (int ii = 0; ii < (H0->sz); ii++) {
        decode(H0->state[ii]);
        d = H0->key[ii];
```

```
        int lt = b[j], up = b[j + 1];
        bool dn = i != n - 1, rt = j != m - 1;
        if (lt && up) {
          if (lt == up) {
            if (i == n - 1 && j == m - 1) {
              push(j, 0, 0);
            }
          } else {
            for (int i = 0; i < m + 1; i++)
              if (b[i] == lt) b[i] = up;
            push(j, 0, 0);
          }
        } else if (lt || up) {
          int t = lt | up;
          if (dn) {
            push(j, t, 0);
          }
          if (rt) {
            push(j, 0, t);
          }
        } else {
          if (dn && rt) {
            push(j, m, m);
          }
        }
      }
    }
    H1->roll();
  }
  assert(H1->sz <= 1);
  cout << (H1->sz == 1 ? H1->key[0] : 0) << endl;
}
```

习题

" 习题 「Ural 1519」Formula 1[7] "

题目大意：求用一条回路覆盖 $N \times M$ 棋盘的方案数，有些位置有障碍。

" 习题 「USACO 5.4.4」Betsy's Tours[8] "

题目大意：一个 $N \times N$ 的方阵（$N \leq 7$），求从左上角出发到左下角结束经过每个格子的路径总数。虽然是一条路径，但因为起点和终点固定，可以转化为一条回路问题。

" 习题 「POJ 1739」Tony's Tour[9] "

题目大意：一个 $N \times M$ 的棋盘，求从左下角出发到右下角结束经过每个格子的路径总数，有些位置有障碍。

" 习题 「USACO 6.1.1」Postal Vans[10] "

题目大意：求用一条有向回路覆盖 $4 \times N$ 的棋盘的方案数，需要高精度。

" 习题 「HNOI 2007」 神奇游乐园[11] "

题目大意：给定一个 $n \times m$ 的网格图，每格内有一个权值，求一个任意一个回路，最大化经过的权值和。

" **习题** 「ProjectEuler 393」Migrating ants[12]"

题目大意：用多条回路覆盖 $n \times n$ 的方阵，每个有 m 条回路的方案对答案的贡献是 2^m，求所有方案的贡献和。

一条路径

例题

" **例题** 「ZOJ 3213」Beautiful Meadow[13]"

题目大意：一个 $N \times M$ 的方阵（$N, M \leq 8$），每个格点有一个权值，求一段路径，最大化路径覆盖的格点的权值和。

本题是标准的一条路径问题，在一条路径问题中，编码的状态中还会存在不能配对的独立插头。需要在状态转移函数中，额外讨论独立插头的生成、合并与消失的情况。独立插头的生成和消失对应着路径的一端，因而这类事件不会发生超过两次（一次生成一次消失，或者两次生成一次合并），否则最终结果一定会出现多个连通块。

我们需要在状态中额外记录这类事件发生的总次数，可以将这个信息编码进状态里（注意，类似这样的额外信息在调整轮廓线的时候，不需要跟着滚动），当然也可以在 `hashTable` 数组的外面加维。下面的范例程序中我们选择后者。

状态转移

" **代码实现**"

```
REP(i, n) {
  REP(j, m) {
    checkMax(ans, A[i][j]);  // 需要单独处理一个格子的情况
    if (!A[i][j]) continue;  // 如果有障碍，则跳过，注意这时状态数组不需要滚动
    swap(H0, H1);
    REP(c, 3)
    H1[c].clear();  // c 表示生成和消失事件发生的总次数，最多不超过 2 次
    REP(c, 3) REP(ii, H0[c].sz) {
      decode(H0[c].state[ii]);
      d = H0[c].key[ii] + A[i][j];
      int lt = b[j], up = b[j + 1];
      bool dn = A[i + 1][j], rt = A[i][j + 1];
      if (lt && up) {
        if (lt == up) {  // 在一条路径问题中，我们不能合并相同的插头。
          // Cannot deploy here...
        } else {  // 有可能参与合并的两者中有独立插头，但是也可以用同样的代码片段
处理
          REP(i, m + 1) if (b[i] == lt) b[i] = up;
          push(c, j, 0, 0);
        }
      } else if (lt || up) {
        int t = lt | up;
        if (dn) {
          push(c, j, t, 0);
        }
        if (rt) {
          push(c, j, 0, t);
        }
```

```
        // 一个插头消失的情况，如果是独立插头则意味着消失，如果是成对出现的插头则相
当于生成了一个独立插头，
        // 无论哪一类事件都需要将 c + 1。
        if (c < 2) {
          push(c + 1, j, 0, 0);
        }
      } else {
        d -= A[i][j];
        H1[c].push(H0[c].state[ii]);
        d += A[i][j];    // 跳过插头生成，本题中不要求全部覆盖
        if (dn && rt) {  // 生成一对插头
          push(c, j, m, m);
        }
        if (c < 2) {  // 生成一个独立插头
          if (dn) {
            push(c + 1, j, m, 0);
          }
          if (rt) {
            push(c + 1, j, 0, m);
          }
        }
      }
    }
  }
  REP(c, 3) H1[c].roll();  // 一行结束，调整轮廓线
}
```

例题代码

```
#include <bits/stdc++.h>
using namespace std;

template <class T>
bool checkMax(T &a, const T b) {
  return a < b ? a = b, 1 : 0;
}

const int N = 8, M = 8;
const int offset = 3, mask = (1 << offset) - 1;
int A[N + 1][M + 1];
int n, m;
int ans, d;
const int MaxSZ = 16796, Prime = 9973;

struct hashTable {
  int head[Prime], next[MaxSZ], sz;
  int state[MaxSZ];
  int key[MaxSZ];

  void clear() {
    sz = 0;
    memset(head, -1, sizeof(head));
  }
```

```
  void push(int s) {
    int x = s % Prime;
    for (int i = head[x]; ~i; i = next[i]) {
      if (state[i] == s) {
        checkMax(key[i], d);
        return;
      }
    }
    state[sz] = s, key[sz] = d;
    next[sz] = head[x];
    head[x] = sz++;
  }

  void roll() {
    for (int i = 0; i < sz; i++) state[i] <<= offset;
  }
} H[2][3], *H0, *H1;

int b[M + 1], bb[M + 1];

int encode() {
  int s = 0;
  memset(bb, -1, sizeof(bb));
  int bn = 1;
  bb[0] = 0;
  for (int i = m; i >= 0; --i) {
    if (!~bb[b[i]]) bb[b[i]] = bn++;
    s <<= offset;
    s |= bb[b[i]];
  }
  return s;
}

void decode(int s) {
  for (int i = 0; i < m + 1; i++) {
    b[i] = s & mask;
    s >>= offset;
  }
}

void push(int c, int j, int dn, int rt) {
  b[j] = dn;
  b[j + 1] = rt;
  H1[c].push(encode());
}

void init() {
  cin >> n >> m;
  H0 = H[0], H1 = H[1];
  for (int c = 0; c < 3; c++) H1[c].clear();
  d = 0;
  H1[0].push(0);
  memset(A, 0, sizeof(A));
  for (int i = 0; i < n; i++)
```

```
    for (int j = 0; j < m; j++) cin >> A[i][j];
}

void solve() {
  ans = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      checkMax(ans, A[i][j]);  // 需要单独处理一个格子的情况
      if (!A[i][j]) continue;  // 如果有障碍，则跳过，注意这时状态数组不需要滚动
      swap(H0, H1);
      for (int c = 0; c < 3; c++)
        H1[c].clear();  // c 表示生成和消失事件发生的总次数，最多不超过 2 次
      for (int c = 0; c < 3; c++)
        for (int ii = 0; ii < H0[c].sz; ii++) {
          decode(H0[c].state[ii]);
          d = H0[c].key[ii] + A[i][j];
          int lt = b[j], up = b[j + 1];
          bool dn = A[i + 1][j], rt = A[i][j + 1];
          if (lt && up) {
            if (lt == up) {  // 在一条路径问题中，我们不能合并相同的插头。
              // Cannot deploy here...
            } else {  // 有可能参与合并的两者中有独立插头，但是也可以用同样的代码
片段处理
              for (int i = 0; i < m + 1; i++)
                if (b[i] == lt) b[i] = up;
              push(c, j, 0, 0);
            }
          } else if (lt || up) {
            int t = lt | up;
            if (dn) {
              push(c, j, t, 0);
            }
            if (rt) {
              push(c, j, 0, t);
            }
            // 一个插头消失的情况，如果是独立插头则意味着消失，如果是成对出现的插头
则相当于生成了一个独立插头，
            // 无论哪一类事件都需要将 c + 1。
            if (c < 2) {
              push(c + 1, j, 0, 0);
            }
          } else {
            d -= A[i][j];
            H1[c].push(H0[c].state[ii]);
            d += A[i][j];  // 跳过插头生成，本题中不要求全部覆盖
            if (dn && rt) {  // 生成一对插头
              push(c, j, m, m);
            }
            if (c < 2) {  // 生成一个独立插头
              if (dn) {
                push(c + 1, j, m, 0);
              }
              if (rt) {
                push(c + 1, j, 0, m);
```

```
                }
              }
            }
          }
      }
      for (int c = 0; c < 3; c++) H1[c].roll();  // 一行结束，调整轮廓线
    }
    for (int ii = 0; ii < H1[2].sz; ii++) checkMax(ans, H1[2].key[ii]);
    cout << ans << endl;
}

int main() {
    int T;
    cin >> T;
    while (T--) {
      init();
      solve();
    }
}
```

习题

" 习题 「BZOJ 2310」ParkII[14] "

题目大意：$m \times n$ 的棋盘，每个格点有一个权值，求一条路径覆盖，最大化路径经过的点的权值和。

" 习题 「NOI 2010 Day2」 旅行路线[15] "

题目大意：$n \times m$ 的棋盘，棋盘的每个格子有一个 01 权值 T[x][y]，要求寻找一个路径覆盖，满足：

- 第 i 个参观的格点 (x, y)，满足 T[x][y]= L[i]
- 路径的一端在棋盘的边界上

求可行的方案数。

染色模型

除了路径模型之外，还有一类常见的模型，需要我们对棋盘进行染色，相邻的相同颜色节点被视为连通。在路径类问题中，状态转移的时候我们枚举当前路径的方向，而在染色类问题中，我们枚举当前节点染何种颜色。在染色模型中，状态中处在相同连通性的节点可能不止两个。但总体来说依然大同小异。我们不妨来看一个经典的例题。

例题「UVa 10572」Black & White

" 例题 「UVa 10572」Black & White[16] "

题目大意：在 $N \times M$ 的棋盘内对未染色的格点进行黑白染色，要求所有黑色区域和白色区域连通，且任意一个 2×2 的子矩形内的颜色不能完全相同（例如下图中的情况非法），求合法的方案数，并构造一组合法的方案。

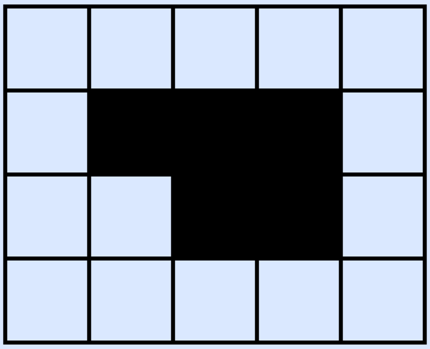


图 7.6 black_and_white1

状态编码

我们先考虑状态编码。不考虑连通性，那么就是 SGU 197. Nice Patterns Strike Back[17]，不难用 状压 DP 直接解决。现在我们需要在状态中同时体现颜色和连通性的信息，考察轮廓线上每个位置的状态，二进制的每 `Offset` 位描述轮廓线上的一个位置，因为只有黑白两种颜色，我们用最低位的奇偶性表示颜色，其余部分示连通性。

考虑第一行上面的节点，和第一列左侧节点，如果要避免特判的话，可以考虑引入第三种颜色区分它们，这里我们观察到这些边界状态的连通性信息一定为 0，所以不需要对第三种颜色再进行额外编码。

在路径问题中我们的轮廓线是由 m 个上插头与 1 个左插头组成的。本题中，由于我们还需要判断当前格点为右下角的 2×2 子矩形是否合法，所以需要记录左上角格子的颜色，因此轮廓线的长度依然是 $m+1$。

这样的编码方案中依然保留了很多冗余信息，（连通的区域颜色一定相同，且左上角的格子只需要颜色信息不需要连通性），但是因为已经用了哈希表和最小表示，对时间复杂度的影响不大，为了降低编程压力，就不再细化了。

在最多情况下（例如第一行黑白相间），每个插头的连通性信息都不一样，因此我们需要 4 位二进制位记录连通性，再加上颜色信息，本题的 `Offset` 为 5 位。

" 代码实现"

```
const int Offset = 5, Mask = (1 << Offset) - 1;
int c[N + 2];
int b[N + 2], bb[N + 3];

T_state encode() {
  T_state s = 0;
  memset(bb, -1, sizeof(bb));
  int bn = 1;
  bb[0] = 0;
  for (int i = m; i >= 0; --i) {
#define bi bb[b[i]]
    if (!~bi) bi = bn++;
    s <<= Offset;
    s |= (bi << 1) | c[i];
  }
  return s;
}

void decode(T_state s) {
  REP(i, m + 1) {
    b[i] = s & Mask;
    c[i] = b[i] & 1;
```

```
      b[i] >>= 1;
      s >>= Offset;
    }
  }
```

手写哈希

因为需要构造任意一组方案，这里的哈希表我们需要添加一组域 `pre[]` 来记录每个状态在上一阶段的任意一个前驱。

" 代码实现"

```
const int Prime = 9979, MaxSZ = 1 << 20;

template <class T_state, class T_key>
struct hashTable {
  int head[Prime];
  int next[MaxSZ], sz;
  T_state state[MaxSZ];
  T_key key[MaxSZ];
  int pre[MaxSZ];

  void clear() {
    sz = 0;
    memset(head, -1, sizeof(head));
  }

  void push(T_state s, T_key d, T_state u) {
    int x = s % Prime;
    for (int i = head[x]; ~i; i = next[i]) {
      if (state[i] == s) {
        key[i] += d;
        return;
      }
    }
    state[sz] = s, key[sz] = d, pre[sz] = u;
    next[sz] = head[x], head[x] = sz++;
  }

  void roll() { REP(ii, sz) state[ii] <<= Offset; }
};

hashTable<T_state, T_key> _H, H[N][N], *H0, *H1;
```

方案构造

有了上面的信息，我们就可以容易的构造方案了。首先遍历当前哈希表中的状态，如果连通块数目不超过 2，那么统计进方案数。如果方案数不为 0，我们倒序用 `pre` 数组构造出方案，注意每一行的末尾因为我们执行了 `Roll()` 操作，颜色需要取 `c[j+1]`。

" 代码实现"

```
void print() {
  T_key z = 0;
  int u;
  REP(i, H1->sz) {
    decode(H1->state[i]);
    if (*max_element(b + 1, b + m + 1) <= 2) {
      z += H1->key[i];
      u = i;
    }
  }
  cout << z << endl;
  if (z) {
    DWN(i, n, 0) {
      B[i][m] = 0;
      DWN(j, m, 0) {
        decode(H[i][j].state[u]);
        int cc = j == m - 1 ? c[j + 1] : c[j];
        B[i][j] = cc ? 'o' : '#';
        u = H[i][j].pre[u];
      }
    }
    REP(i, n) puts(B[i]);
  }
  puts("");
}
```

状态转移

我们记：

- cc 当前正在染色的格子的颜色
- lf 左边格子的颜色
- up 上边格子的颜色
- lu 左上格子的颜色

我们用 -1 表示颜色不存在。接下来讨论状态转移，一共有三种情况，合并，继承与生成：

" 状态转移 - 代码"

```
void trans(int i, int j, int u, int cc) {
  decode(H0->state[u]);
  int lf = j ? c[j - 1] : -1, lu = b[j] ? c[j] : -1,
      up = b[j + 1] ? c[j + 1] : -1;  // 没有颜色也是颜色的一种!
  if (lf == cc && up == cc) {         // 合并
    if (lu == cc) return;             // 2x2 子矩形相同的情况
    int lf_b = b[j - 1], up_b = b[j + 1];
    REP(i, m + 1) if (b[i] == up_b) { b[i] = lf_b; }
    b[j] = lf_b;
  } else if (lf == cc || up == cc) {  // 继承
    if (lf == cc)
      b[j] = b[j - 1];
    else
      b[j] = b[j + 1];
```

```
  } else {                                        // 生成
    if (i == n - 1 && j == m - 1 && lu == cc) return;  // 特判
    b[j] = m + 2;
  }
  c[j] = cc;
  if (!ok(i, j, cc)) return;  // 判断是否会因生成封闭的连通块导致不合法
  H1->push(encode(), H0->key[u], u);
}
```

对于最后一种情况需要注意的是，如果已经生成了一个封闭的连通区域，那么我们不能再使用她的颜色染色，否则这种颜色会出现两个连通块。我们似乎需要额度记录这种事件，可以参考「ZOJ 3213」Beautiful Meadow 中的做法，再开一维记录这个事件。不过利用本题的特殊性，我们也可以特判掉。

" 特判 - 代码"

```
bool ok(int i, int j, int cc) {
  if (cc == c[j + 1]) return true;
  int up = b[j + 1];
  if (!up) return true;
  int c1 = 0, c2 = 0;
  REP(i, m + 1) if (i != j + 1) {
    if (b[i] == b[j + 1]) {  // 连通性相同，颜色一定相同
      assert(c[i] == c[j + 1]);
    }
    if (c[i] == c[j + 1] && b[i] == b[j + 1]) ++c1;
    if (c[i] == c[j + 1]) ++c2;
  }
  if (!c1) {                 // 如果会生成新的封闭连通块
    if (c2) return false;  // 如果轮廓线上还有相同的颜色
    if (i < n - 1 || j < m - 2) return false;
  }
  return true;
}
```

进一步讨论连通块消失的情况。每当我们对一个格子进行染色后，如果没有其他格子与其上侧的格子连通，那么会形成一个封闭的连通块。这个事件仅在最后一行的最后两列时可以发生，否则后续为了不出现 2×2 的同色连通块，这个颜色一定会再次出现，除了下面的情况：

```
2 2
o#
#o
```

我们特判掉这种情况，这样在本题中，就可以偷懒不用记录之前是否已经生成了封闭的连通块了。

例题代码

```
#include <bits/stdc++.h>
using namespace std;
typedef long long T_state;
typedef int T_key;
const int N = 8;
int n, m;
char A[N + 1][N + 1], B[N + 1][N + 1];
const int Offset = 5, Mask = (1 << Offset) - 1;
int c[N + 2];
```

```
int b[N + 2], bb[N + 3];

T_state encode() {
  T_state s = 0;
  memset(bb, -1, sizeof(bb));
  int bn = 1;
  bb[0] = 0;
  for (int i = m; i >= 0; --i) {
    if (!~bb[b[i]]) bb[b[i]] = bn++;
    s <<= Offset;
    s |= (bb[b[i]] << 1) | c[i];
  }
  return s;
}

void decode(T_state s) {
  for (int i = 0; i < m + 1; i++) {
    b[i] = s & Mask;
    c[i] = b[i] & 1;
    b[i] >>= 1;
    s >>= Offset;
  }
}

const int Prime = 9979, MaxSZ = 1 << 20;

template <class T_state, class T_key>
struct hashTable {
  int head[Prime];
  int next[MaxSZ], sz;
  T_state state[MaxSZ];
  T_key key[MaxSZ];
  int pre[MaxSZ];

  void clear() {
    sz = 0;
    memset(head, -1, sizeof(head));
  }

  void push(T_state s, T_key d, T_state u) {
    int x = s % Prime;
    for (int i = head[x]; ~i; i = next[i]) {
      if (state[i] == s) {
        key[i] += d;
        return;
      }
    }
    state[sz] = s, key[sz] = d, pre[sz] = u;
    next[sz] = head[x], head[x] = sz++;
  }

  void roll() {
    for (int ii = 0; ii < sz; ii++) state[ii] <<= Offset;
  }
```

```
};

hashTable<T_state, T_key> _H, H[N][N], *H0, *H1;

bool ok(int i, int j, int cc) {
  if (cc == c[j + 1]) return true;
  int up = b[j + 1];
  if (!up) return true;
  int c1 = 0, c2 = 0;
  for (int i = 0; i < m + 1; i++)
    if (i != j + 1) {
      if (b[i] == b[j + 1]) {  // 连通性相同，颜色一定相同
        assert(c[i] == c[j + 1]);
      }
      if (c[i] == c[j + 1] && b[i] == b[j + 1]) ++c1;
      if (c[i] == c[j + 1]) ++c2;
    }
  if (!c1) {                // 如果会生成新的封闭连通块
    if (c2) return false;  // 如果轮廓线上还有相同的颜色
    if (i < n - 1 || j < m - 2) return false;
  }
  return true;
}

void trans(int i, int j, int u, int cc) {
  decode(H0->state[u]);
  int lf = j ? c[j - 1] : -1, lu = b[j] ? c[j] : -1,
      up = b[j + 1] ? c[j + 1] : -1;  // 没有颜色也是颜色的一种!
  if (lf == cc && up == cc) {         // 合并
    if (lu == cc) return;             // 2x2 子矩形相同的情况
    int lf_b = b[j - 1], up_b = b[j + 1];
    for (int i = 0; i < m + 1; i++)
      if (b[i] == up_b) {
        b[i] = lf_b;
      }
    b[j] = lf_b;
  } else if (lf == cc || up == cc) {  // 继承
    if (lf == cc)
      b[j] = b[j - 1];
    else
      b[j] = b[j + 1];
  } else {                                             // 生成
    if (i == n - 1 && j == m - 1 && lu == cc) return;  // 特判
    b[j] = m + 2;
  }
  c[j] = cc;
  if (!ok(i, j, cc)) return;  // 判断是否会因生成封闭的连通块导致不合法
  H1->push(encode(), H0->key[u], u);
}

void init() {
  cin >> n >> m;
  for (int i = 0; i < n; i++) scanf("%s", A[i]);
}
```

```
void solve() {
  H1 = &_H, H1->clear(), H1->push(0, 1, 0);
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      H0 = H1, H1 = &H[i][j], H1->clear();
      for (int u = 0; u < H0->sz; u++) {
        if (A[i][j] == '.' || A[i][j] == '#') trans(i, j, u, 0);
        if (A[i][j] == '.' || A[i][j] == 'o') trans(i, j, u, 1);
      }
    }
    H1->roll();
  }
}

void print() {
  T_key z = 0;
  int u;
  for (int i = 0; i < H1->sz; i++) {
    decode(H1->state[i]);
    if (*max_element(b + 1, b + m + 1) <= 2) {
      z += H1->key[i];
      u = i;
    }
  }
  cout << z << endl;
  if (z) {
    for (int i = n - 1; i >= 0; i--) {
      B[i][m] = 0;
      for (int j = m - 1; j >= 0; j--) {
        decode(H[i][j].state[u]);
        int cc = j == m - 1 ? c[j + 1] : c[j];
        B[i][j] = cc ? 'o' : '#';
        u = H[i][j].pre[u];
      }
    }
    for (int i = 0; i < n; i++) puts(B[i]);
  }
  puts("");
}

int main() {
  int T;
  cin >> T;
  while (T--) {
    init();
    solve();
    print();
  }
}
```

习题

" **习题** 「Topcoder SRM 312. Div1 Hard」CheapestIsland[18]"

题目大意：给一个棋盘图，每个格子有权值，求权值之和最小的连通块。

" **习题** 「JLOI 2009」 **神秘的生物**[19]"

题目大意：给一个棋盘图，每个格子有权值，求权值之和最大的连通块。

" **习题** 「AtCoder Beginner Contest 211. Problem E」Red Polyomino[20]"

题目大意：给一个 $N \times N$ 大小的棋盘图，每个格子初始为黑色或白色。你可以从白色格子中挑选恰好 K 个并将之染成红色，问有多少种染色方案满足红色格子形成一个连通块。

图论模型

" **例题** 「NOI 2007 Day2」 **生成树计数**[21]"

题目大意：某类特殊图的生成树计数，每个节点恰好与其前 k 个节点之间有边相连。

" **例题** 「2015 ACM-ICPC Asia Shenyang Regional Contest - Problem E」Efficient Tree[22]"

题目大意：给出一个 $N \times M$ 的网格图，以及相邻四连通格子之间的边权。对于一颗生成树，每个节点的得分为 1+[有一条连向上的边]+[有一条连向左的边]。生成树的得分为所有节点的得分之积。

你需要求出：最小生成树的边权和，以及所有最小生成树的得分之和。($n \leq 800, m \leq 7$)

实战篇

例题

" **例题** 「HDU 4113」Construct the Great Wall[23]"

题目大意：在 $N \times M$ 的棋盘内构造一组回路，分割所有的 x 和 o。

有一类插头 DP 问题要求我们在棋盘上构造一组墙，以分割棋盘上的某些元素。不妨称之为修墙问题，这类问题既可视作染色模型，也可视作路径模型。

在本题中，如果视作染色模型的话，不仅需要额外讨论染色区域的周长，还要判断在角上触碰而导致不合法的情况（图 2）。另外与「UVa 10572」Black & White[16] 不同的是，本题中要求围墙为简单多边形，因而对于下面的回字形的情况，在本题中是不合法的。

```
3 3
ooo
oxo
ooo
```

因而我们使用路径模型，转化为一条回路来处理。

我们沿着棋盘的交叉点进行 DP（因而长宽需要增加 1），每次转移时，需要保证所有的 x 在回路之外，o 在回路之内。因此我们还需要维护当前位置是否在回路内部。对于这个信息我们可以加维，也可以直接统计轮廓线上到这个位置之前出现下插头次数的奇偶性（射线法）。

例题代码

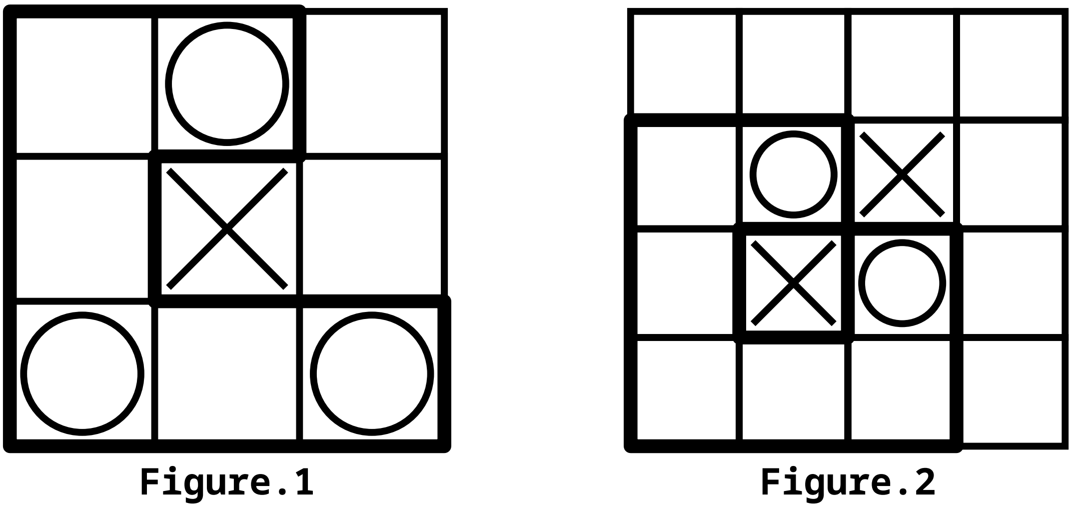


图 7.7 greatwall

```
#include <bits/stdc++.h>
using namespace std;
#define REP(i, n) for (int i = 0; i < n; ++i)

template <class T>
bool checkMin(T &a, const T b) {
  return b < a ? a = b, 1 : 0;
}

const int N = 10, M = N;
const int offset = 3, mask = (1 << offset) - 1;
int n, m;
int d;
const int INF = 0x3f3f3f3f;
int b[M + 1], bb[M + 1];

int encode() {
  int s = 0;
  memset(bb, -1, sizeof(bb));
  int bn = 1;
  bb[0] = 0;
  for (int i = m; i >= 0; --i) {
#define bi bb[b[i]]
    if (!~bi) bi = bn++;
    s <<= offset;
    s |= bi;
  }
  return s;
}

void decode(int s) {
  REP(i, m + 1) {
    b[i] = s & mask;
```

```
    s >>= offset;
  }
}

const int MaxSZ = 16796, Prime = 9973;

struct hashTable {
  int head[Prime], next[MaxSZ], sz;
  int state[MaxSZ];
  int key[MaxSZ];

  void clear() {
    sz = 0;
    memset(head, -1, sizeof(head));
  }

  void push(int s) {
    int x = s % Prime;
    for (int i = head[x]; ~i; i = next[i]) {
      if (state[i] == s) {
        checkMin(key[i], d);
        return;
      }
    }
    state[sz] = s, key[sz] = d;
    next[sz] = head[x];
    head[x] = sz++;
  }

  void roll() { REP(i, sz) state[i] <<= offset; }
} H[2], *H0, *H1;

char A[N + 1][M + 1];

void push(int i, int j, int dn, int rt) {
  b[j] = dn;
  b[j + 1] = rt;
  if (A[i][j] != '.') {
    bool bad = A[i][j] == 'o';
    REP(jj, j + 1) if (b[jj]) bad ^= 1;
    if (bad) return;
  }
  H1->push(encode());
}

int solve() {
  cin >> n >> m;
  int ti, tj;
  REP(i, n) {
    scanf("%s", A[i]);
    REP(j, m) if (A[i][j] == 'o') ti = i, tj = j;
    A[i][m] = '.';
  }
  REP(j, m + 1) A[n][j] = '.';
```

```
  ++n, ++m, ++ti, ++tj;
  H0 = H, H1 = H + 1;
  H1->clear();
  d = 0;
  H1->push(0);
  int z = INF;
  REP(i, n) {
    REP(j, m) {
      swap(H0, H1);
      H1->clear();
      REP(ii, H0->sz) {
        decode(H0->state[ii]);
        d = H0->key[ii] + 1;
        int lt = b[j], up = b[j + 1];
        bool dn = i != n - 1, rt = j != m - 1;
        if (lt && up) {
          if (lt == up) {
            int cnt = 0;
            REP(i, m + 1) if (b[i])++ cnt;
            if (cnt == 2 && i == ti && j == tj) {
              checkMin(z, d);
            }
          } else {
            REP(i, m + 1) if (b[i] == lt) b[i] = up;
            push(i, j, 0, 0);
          }
        } else if (lt || up) {
          int t = lt | up;
          if (dn) {
            push(i, j, t, 0);
          }
          if (rt) {
            push(i, j, 0, t);
          }
        } else {
          --d;
          push(i, j, 0, 0);
          ++d;
          if (dn && rt) {
            push(i, j, m, m);
          }
        }
      }
    }
    H1->roll();
  }
  if (z == INF) z = -1;
  return z;
}

int main() {
#ifndef ONLINE_JUDGE
  freopen("in.txt", "r", stdin);
#endif
```

```
  int T;
  cin >> T;
  for (int Case = 1; Case <= T; ++Case) {
    printf("Case #%d: %d\n", Case, solve());
  }
}
```

习题

” **习题** 「SCOI 2011」 **地板**[24]”

题目大意：$r \times c$ 的棋盘上有一些位置设置障碍，问使用 L 型的瓷砖铺满所有没有障碍的格子，有多少种方案。

” **习题** 「HDU 4796」Winter's Coming[25]”

题目大意：在 $N \times M$ 的棋盘内对未染色的格点进行黑白灰染色，要求所有黑色区域和白色区域连通，且黑色区域与白色区域分别与棋盘的上下边界连通，且其中黑色区域与白色区域不能相邻。每个格子有对应的代价，求一组染色方案，最小化灰色区域的代价。

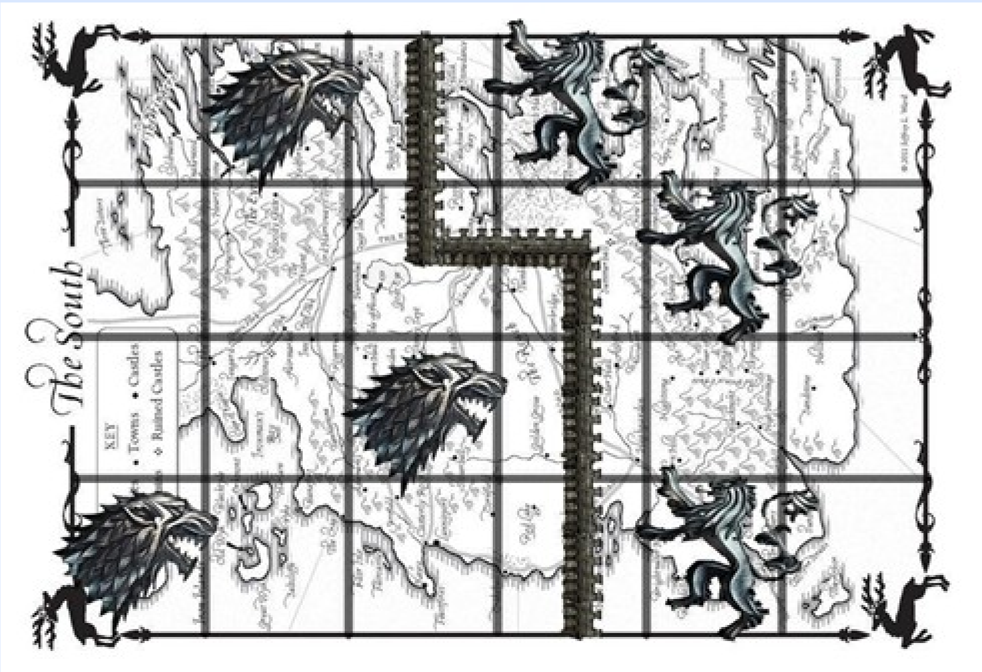


图 7.8 4796

” **习题** 「ZOJ 2125」Rocket Mania[26]”

题目大意：9×6 的地图上每个格子里是一种管道（-,T,L,+ 型或没有），可以把管道旋转 0°,90°,180°,270°，问地图最多能有几行的右边界与第 X 行的左边界通过管道相连。

” **习题** 「ZOJ 2126」Rocket Mania Plus[27]”

题目大意：9×6 的地图上每个格子里是一种管道（-,T,L,+ 型或没有），可以把管道旋转 0°,90°,180°,270°，问地图最多能有几行的右边界与左边界通过管道相连。

” **习题** 「World Finals 2009/2010 Harbin」Channel[28]”

题目大意：一张方格地图上用 . 表示空地、# 表示石头，找到最长的一条路径满足：

1. 起点在左上角，终点在右下角。
2. 不能经过石头。
3. 路径自身不能在八连通的意义下成环。（即包括拐角处也不能接触）

" **习题** 「HDU 3958」Tower Defence[29]"

题目大意：可以转化为求解一条从 S 到 T 的不能接触的最长路径，拐角处可以接触。

" **习题** 「UVa 10531」Maze Statistics[30]"

题目大意：有一个 $N \times M$ 的图，每个格子有独立概率 p 变成障碍物。你要从迷宫左上角走到迷宫右下角。求每个格子成为一个**有解迷宫（即起点终点四联通）**中的障碍物的概率。（$N \le 5$，$M \le 6$）

" **习题** 「AIZU 2452」Pipeline Plans[31]"

题目大意：现有一共 12 种图案的瓷砖，每种瓷砖数量给定。要求铺到一块可视为 $R \times C$ 网格图的矩形地板上，一个格子铺一块瓷砖，且左上角格子的中心与右下角格子的中心通过瓷砖图案上的线联通。($2 \le R \times C \le 15$)

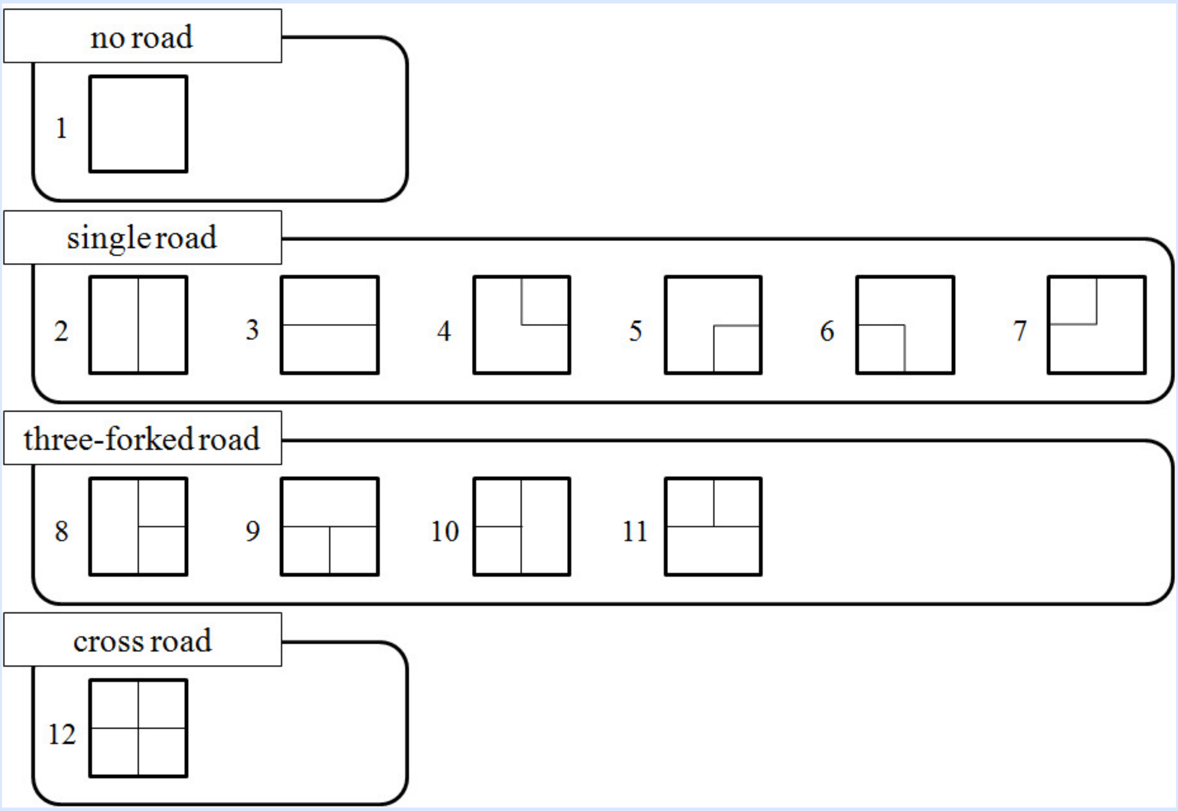


图 7.9 plug2

" **习题** 「SDOI 2014」 **电路板**[32]"

题目大意：一块 $N \times M$ 的电路板，上面有些位置是电线不能走的障碍，给定 K 个格子对，要求每对格子都有电线相连，且电线之间互不相交（允许一条电路线从上边界进入当前格子，从左边界离开这个格子，另外一条电路线可以从下边界进入格子，从右边界出去）。视电线为无向边，求满足要求的最短电线长度和方案数。

" **习题** 「SPOJ CAKE3」Delicious Cake[33]"

题目大意：一块可视为 $N \times M$ 网格的蛋糕，现沿着格线将蛋糕切成数块，问有多少种不同的切割方法。切法相同当且仅当切成的每块蛋糕都形状相同且在同一位置上。($\min(N, M) \le 5, \max(N, M) \le 130$)

本章注记

插头 DP 问题通常编码难度较大，讨论复杂，因而属于 OI/ACM 中相对较为 偏门的领域[34]。这方面最为经典的资料，当属 2008 年 陈丹琦[35] 的集训队论文——基于连通性状态压缩的动态规划问题[36]。其次，HDU 的 notonlysuccess 2011 年曾经在博客中连续写过两篇由浅入深的专题，也是不可多得的好资料，不过现在需要在 Web Archive 里考古。

- notonlysuccess，【专辑】 插头 DP[37]
- notonlysuccess，【完全版】 插头 DP[38]

多米诺骨牌覆盖

「HDU 1400」Mondriaan's Dream[1] 也出现在 《算法竞赛入门经典训练指南》 中，并作为《轮廓线上的动态规划》一节的例题。多米诺骨牌覆盖 （Domino tiling）[39] 是一组非常经典的数学问题，稍微修改其数据范围就可以得到不同难度，需要应用不同的算法解决的子问题。

当限定 $m = 2$ 时，多米诺骨牌覆盖等价于斐波那契数列。《具体数学》[40] 中使用了该问题以引出斐波那契数列，并使用了多种方法得到其解析解。

当 $m \le 10, n \le 10^9$ 时，可以将转移方程预处理成矩阵形式，并使用 矩阵乘法进行加速[41]。

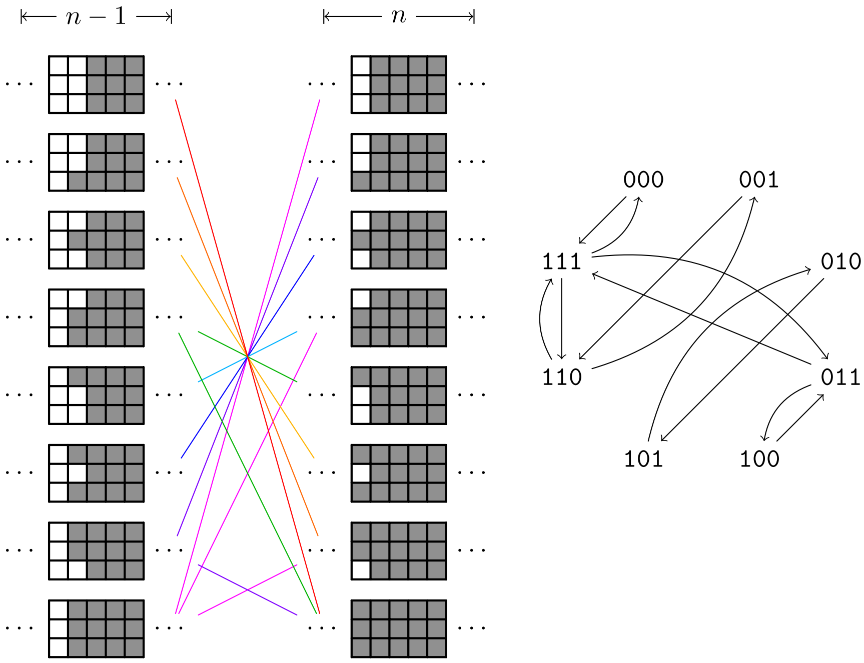


图 7.10 domino_v2_transform_matrix

当 $n, m \le 100$，可以用 FKT Algorithm[42] 计算其所对应平面图的完美匹配数。

- 「51nod 1031」 骨牌覆盖[43]
- 「51nod 1033」 骨牌覆盖 V2[44]|「Vijos 1194」Domino[45]
- 「51nod 1034」 骨牌覆盖 V3[46]|「Ural 1594」Aztec Treasure[47]
- Wolfram MathWorld, Chebyshev Polynomial of the Second Kind[48]

一条路径

「一条路径」是 哈密顿路径 (Hamiltonian Path)[49] 问题在 格点图 (Grid Graph)[50] 中的一种特殊情况。哈密顿路径的判定性问题是 NP-complete[51] 家族中的重要成员。

参考资料与注释

[1] 「HDU 1400」Mondriaan' s Dream [1-1] [1-2]

[2] 「SRM 671. Div 1 900」BearDestroys

[3] 「HDU 1693」Eat the Trees

[4] 「ZJU 4231」The Hive II

[5] 「Andrew Stankevich Contest 16 - Problem F」Pipe Layout

[6] std::unordered_map

[7] 「Ural 1519」Formula 1

[8] 「USACO 5.4.4」Betsy's Tours

[9] 「POJ 1739」Tony's Tour

[10] 「USACO 6.1.1」Postal Vans

[11] 「HNOI 2007」神奇游乐园

[12] 「ProjectEuler 393」Migrating ants

[13] 「ZOJ 3213」Beautiful Meadow

[14] 「BZOJ 2310」ParkII

[15] 「NOI 2010 Day2」旅行路线

[16] 「UVa 10572」Black & White [16-1] [16-2]

[17] SGU 197. Nice Patterns Strike Back

[18] 「Topcoder SRM 312. Div1 Hard」CheapestIsland

[19] 「JLOI 2009」神秘的生物

[20] 「AtCoder Beginner Contest 211. Problem E」Red Polyomino

[21] 「NOI 2007 Day2」生成树计数

[22] 「2015 ACM-ICPC Asia Shenyang Regional Contest - Problem E」Efficient Tree

[23] 「HDU 4113」Construct the Great Wall

[24] 「SCOI 2011」地板

[25] 「HDU 4796」Winter's Coming

[26] 「ZOJ 2125」Rocket Mania

[27] 「ZOJ 2126」Rocket Mania Plus

[28] 「World Finals 2009/2010 Harbin」Channel

[29] 「HDU 3958」Tower Defence

[30] 「UVa 10531」Maze Statistics

[31] 「AIZU 2452」Pipeline Plans

[32] 「SDOI 2014」电路板

[33] 「SPOJ CAKE3」Delicious Cake

[34] 偏门的领域

[35] 陈丹琦

[36] 基于连通性状态压缩的动态规划问题

[37] notonlysuccess，【专辑】插头 DP

[38] notonlysuccess，【完全版】插头 DP

[39] 多米诺骨牌覆盖（Domino tiling）

[40] 《具体数学》

[41] 矩阵乘法进行加速

[42] FKT Algorithm

[43] 「51nod 1031」骨牌覆盖

[44] 「51nod 1033」骨牌覆盖 V2

[45] 「Vijos 1194」Domino

[46] 「51nod 1034」骨牌覆盖 V3

[47] 「Ural 1594」Aztec Treasure

[48] Wolfram MathWorld, Chebyshev Polynomial of the Second Kind

[49] 哈密顿路径（Hamiltonian Path）

[50] 格点图（Grid Graph）

[51] NP-complete

7.11 计数 DP

计数 DP 是一种利用类似 DP 的记忆化搜索方法（与在狭义上的 DP，即最优化问题有一定区别），用于解决计数（以及求和）问题。

基础

基本思想

计数问题一般指求一个集合 S 的大小，在 OI 中，S 的大小有时会达到 $\Theta(n^n)$ 甚至 $\Theta(2^{n!})$ 的级别（当然，一般会对某一个固定的数取模），其中 n 是问题规模，所以我们不能逐一求出 S 的元素。

如果我们能够将 S 分成若干无交的子集，那么 S 的元素个数就等于这些部分的元素个数的和。如果这些子集的计数恰好与原问题类似，那么我们就可以通过类似动态规划的方法来解决。

例题

" 例题"

给定一个正整数 n，求有多少个把 n 划分成 k 个正整数的和的方案，位置调换视为不同的划分方案。

需集合 $S_{n,k}$ 为形如 $(a_1,\dots,a_k)$ 的正整数组组成的集合，其中 $a_1+\dots+a_k=n$。如果 a_k 固定，则有如下推导：因为 $a_1+a_2+\dots+a_{k-1}+a_k=n$，所以 $a_1+a_2+\dots+a_{k-1}=n-a_k$。根据 $S_{n,k}$ 的定义，$(a_1,a_2,\dots,a_{k-1})\in S_{n-a_k,k-1}$。

由于 $a_1,a_2,\dots,a_k$ 是正整数，所以 a_k 的取值范围是 $[1,n-k+1]\cap\mathbb{Z}$。因此，$S_{n,k}$ 可以按照 a_k 被划分，分成 $n-k+1$ 个子集，其中当 $a_k=i$ 时，这个子集为：

$$\{(L,i)\mid L\in S_{n-i,k-1}\}.$$

这个子集的元素个数显然等于 $S_{n-i,k-1}$，由于 i 的不同，这些子集两两无交。所以：

$$|S_{n,k}|=\sum_{i=1}^{n-k+1}|S_{n-i,k-1}|.$$

这样我们就可以使用类似 DP 的方法处理它：设 $f_{n,k}$ 为 $|S_{n,k}|$，则有状态转移方程：

$$f_{n,k} = \sum_{i=1}^{n-k+1} f_{n-i,k-1}.$$

这样就可以使用 DP 的方法求解了。

与最优化 DP 的异同

可以发现，计数 DP 和最优化 DP 都是在一个范围 Ω 内求一个值（大小值、最优值），这个值通过将 Ω 中的所有元素做一次处理，再对处理值做一次整合得到。

例如，对于 0-1 背包问题，Ω 中的元素为背包内的所有物品组成的集合，对于 Ω 中的一个方案 S，我们对 S 做一次处理，处理得到的结果 $w(S)$ 为 S 中物品的总价值，对所有得到的处理值，我们取最大值，得到问题的答案。

对于计数问题，Ω 中的元素为要计算元素个数的集合 S，它的处理是把所有的 S 中元素变为 1，然后将这些 1 通过加法的方式汇总起来，因为每一个 S 中元素都对应一个 1，所以这样得到的值就是 S 中元素个数。

当汇总操作为最大/最小值时，我们可以将 Ω 分成任意若干个部分，只需这些部分的并为 Ω 即可，无需无交的条件。而计数问题由于不满足这个条件，所以我们需要将 Ω 分成若干个部分，这些部分两两无交，这就是与最优化 DP 的区别。

例题

" 例题"

给定一个正整数 n，求有多少个把 n 划分成任意多个正整数的和的方案，位置调换视为**相同**的划分方案。

解法 1

需要计算的集合的元素为满足其和为 n 的正整数多重集。但是这样显然不好推。

若一个多重集 T 只包含 $\le M$ 的正整数，且 T 中所有元素的和为 n，则称 $T \in S_{n,M}$。考虑 M 出现的个数。可能为 $k \in \left[0, \left\lfloor \frac{n}{M} \right\rfloor\right] \cap \mathbb{Z}$。于是它可以被转移到 $S_{n-kM,M-1}$。求和一下即可。复杂度是 $\Theta(n^2 \log n)$（log 来自于 k 的范围导致的调和级数）。

但是这样还不够优秀。考虑下面所示的一个例子：

$$\begin{aligned}
f_{8,3} &= f_{8,2} + f_{5,2} + f_{2,2} \\
f_{9,3} &= f_{9,2} + f_{6,2} + f_{3,2} + f_{0,2} \\
f_{10,3} &= f_{10,2} + f_{7,2} + f_{4,2} + f_{1,2} \\
f_{11,3} &= f_{11,2} + f_{8,2} + f_{5,2} + f_{2,2} \\
f_{12,3} &= f_{12,2} + f_{9,2} + f_{6,2} + f_{3,2} + f_{0,2} \\
f_{13,3} &= f_{13,2} + f_{10,2} + f_{7,2} + f_{4,2} + f_{1,2}
\end{aligned}$$

等量代换得 $f_{11,3} = f_{11,2} + f_{8,3}$，$f_{12,3} = f_{12,2} + f_{9,3}$，$f_{13,3} = f_{13,2} + f_{10,3}$。同理我们可以得到一个通用的状态转移方程：

$$f_{n,M} = f_{n,M-1} + \begin{cases} f_{n-M,M} & n \ge M, \\ 0 & \text{otherwise.} \end{cases}$$

此时，时间复杂度为 $\Theta(n^2)$。

解法 2

考虑到某一个正整数组成的多重集 T 必然可以通过「将 T 中每一个元素自增」、「在 T 中加一个值为 1 的元素」两个操作得到，并且不同的操作序列得到的结果是不同的。

这样对 T 的转移可以变为对操作序列的转移。考虑将 n 划分成 m 个数的操作序列（所有的这些操作序列记作 $B_{n,m}$）中的最后一次操作，如果是 1 操作，那么不会增加数，但是 $\sum T$ 增加了 m。为了使最终的 $\sum T = n$，原来的 T（记作 T'）的和需要为 $n-m$。所以 $B_{n,m} \to B_{n-m,m}$；如果是 2 操作，那么会增加一个数，$\sum T$ 增加了 1。所以 $B_{n,m} \to B_{n-1,m-1}$。

这样做的时间复杂度依旧是 $\Theta(n^2)$。

解法 3

考虑将 T 分为大于 $\sqrt{n}$ 的部分 T_1 和小于等于 $\sqrt{n}$ 的部分 T_2。T_2 可以使用解法 1 求出，而 T_1 的数量可以通过略微修改解法 2 求出：考虑将两个操作变为「将 T_1 中每一个元素自增」、「在 T_1 中加一个值为 $\lfloor\sqrt{n}\rfloor+1$ 的元素」。容易列出状态转移方程。

将 n 拆为 A 和 B 两部分。枚举其中一个即可得出另一个。将满足 $\sum T_1 = A$ 的 T_1 个数和 $\sum T_2 = B$ 的 T_2 个数求出，乘起来，对所有的 A 求和便是最终结果。

由于在计算 T_1 个数的过程中，$M \le \sqrt{n}$，所以我们利用解法 1 计算 T_1 的时间复杂度为 $\Theta(n^{3/2})$。同样地，由于在计算 T_2 个数的过程中，$|T_2| \le \dfrac{\sum T_2}{\sqrt{n}} \le \dfrac{n}{\sqrt{n}} = \sqrt{n}$，所以我们利用解法 2 计算 T_2 的时间复杂度也是 $\Theta(n^{3/2})$。所以总时间复杂度为 $\Theta(n^{3/2})$。

7.12 动态 DP

前置知识：矩阵，树链剖分。

动态 DP 问题是猫锟在 WC2018 讲的黑科技，一般用来解决树上的带有点权（边权）修改操作的 DP 问题。

例子

以这道模板题为例子讲解一下动态 DP 的过程。

" 例题 洛谷 P4719【模板】动态 DP[1]"

给定一棵 n 个点的树，点带点权。有 m 次操作，每次操作给定 x,y 表示修改点 x 的权值为 y。你需要在每次操作之后求出这棵树的最大权独立集的权值大小。

广义矩阵乘法

定义广义矩阵乘法 $A \times B = C$ 为：

$$C_{i,j} = \max_{k=1}^{n}(A_{i,k} + B_{k,j})$$

相当于将普通的矩阵乘法中的乘变为加，加变为 max 操作。

同时广义矩阵乘法满足结合律，所以可以使用矩阵快速幂。

不带修改操作

令 $f_{i,0}$ 表示不选择 i 的最大答案，$f_{i,1}$ 表示选择 i 的最大答案。

则有 DP 方程：

$$\begin{cases} f_{i,0} = \sum_{son} \max(f_{son,0}, f_{son,1}) \\ f_{i,1} = w_i + \sum_{son} f_{son,0} \end{cases}$$

答案就是 $\max(f_{root,0}, f_{root,1})$.

带修改操作

首先将这棵树进行树链剖分，假设有这样一条重链：

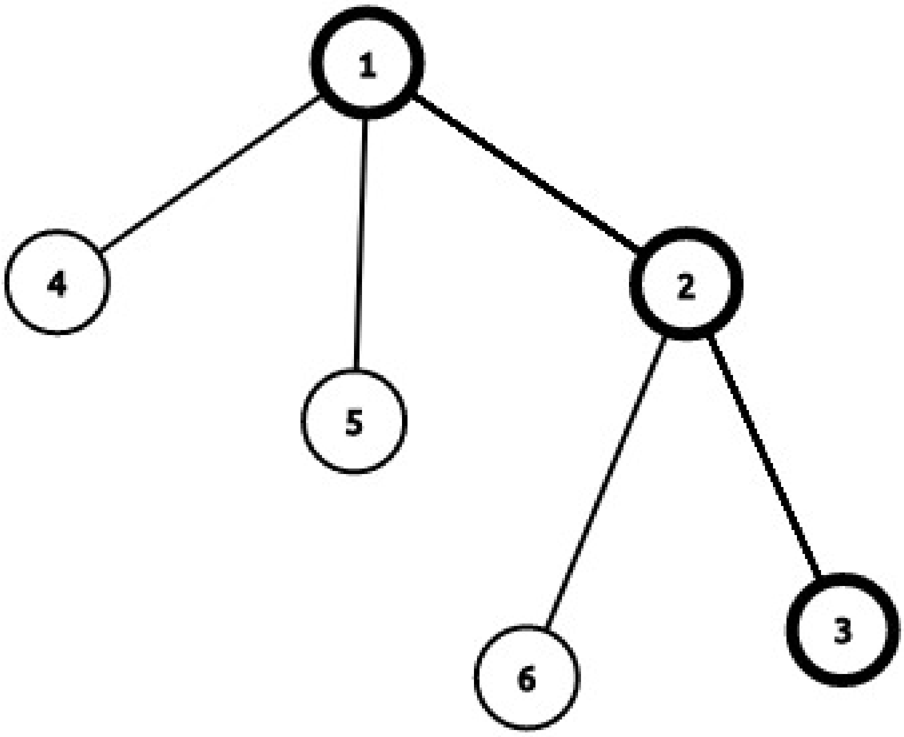


图 7.11

设 $g_{i,0}$ 表示不选择 i 且只允许选择 i 的轻儿子所在子树的最大答案，$g_{i,1}$ 表示不考虑 son_i 的情况下选择 i 的最大答案，son_i 表示 i 的重儿子。

假设我们已知 $g_{i,0/1}$ 那么有 DP 方程：

$$\begin{cases} f_{i,0} = g_{i,0} + \max(f_{son_i,0}, f_{son_i,1}) \\ f_{i,1} = g_{i,1} + f_{son_i,0} \end{cases}$$

答案是 $\max(f_{root,0}, f_{root,1})$。

可以构造出矩阵：

$$\begin{bmatrix} g_{i,0} & g_{i,0} \\ g_{i,1} & -\infty \end{bmatrix} \times \begin{bmatrix} f_{son_i,0} \\ f_{son_i,1} \end{bmatrix} = \begin{bmatrix} f_{i,0} \\ f_{i,1} \end{bmatrix}$$

注意，我们这里使用的是广义乘法规则。

可以发现，修改操作时只需要修改 $g_{i,1}$ 和每条往上的重链即可。

具体思路

1. DFS 预处理求出 $f_{i,0/1}$ 和 $g_{i,0/1}$。
2. 对这棵树进行树链剖分（注意，因为我们对一个点进行询问需要计算从该点到该点所在的重链末尾的区间矩阵乘，所以对于每一个点记录 End_i 表示 i 所在的重链末尾节点编号），每一条重链建立线段树，线段树维护 g 矩阵和 g 矩阵区间乘积。
3. 修改时首先修改 $g_{i,1}$ 和线段树中 i 节点的矩阵，计算 top_i 矩阵的变化量，修改到 fa_{top_i} 矩阵。
4. 查询时就是 1 到其所在的重链末尾的区间乘，最后取一个 max 即可。

" 代码实现"

```cpp
#include <bits/stdc++.h>
using namespace std;

typedef long long LL;

const int maxn = 500010;
const int INF = 0x3f3f3f3f;

int Begin[maxn], Next[maxn], To[maxn], e, n, m;
int sz[maxn], son[maxn], top[maxn], fa[maxn], dis[maxn], p[maxn], id[maxn],
    End[maxn];
// p[i] 表示 i 树剖后的编号，id[p[i]] = i
int cnt, tot, a[maxn], f[maxn][2];

struct matrix {
  int g[2][2];

  matrix() { memset(g, 0, sizeof(g)); }

  matrix operator*(const matrix &b) const  // 重载矩阵乘
  {
    matrix c;
    for (int i = 0; i <= 1; i++)
      for (int j = 0; j <= 1; j++)
        for (int k = 0; k <= 1; k++)
          c.g[i][j] = max(c.g[i][j], g[i][k] + b.g[k][j]);
    return c;
  }
} Tree[maxn], g[maxn];  // Tree[] 是建出来的线段树，g[] 是维护的每个点的矩阵

void PushUp(int root) { Tree[root] = Tree[root << 1] * Tree[root << 1 | 1]; }

void Build(int root, int l, int r) {
  if (l == r) {
    Tree[root] = g[id[l]];
    return;
  }
  int Mid = l + r >> 1;
  Build(root << 1, l, Mid);
  Build(root << 1 | 1, Mid + 1, r);
  PushUp(root);
}

matrix Query(int root, int l, int r, int L, int R) {
  if (L <= l && r <= R) return Tree[root];
  int Mid = l + r >> 1;
  if (R <= Mid) return Query(root << 1, l, Mid, L, R);
  if (Mid < L) return Query(root << 1 | 1, Mid + 1, r, L, R);
  return Query(root << 1, l, Mid, L, R) *
         Query(root << 1 | 1, Mid + 1, r, L, R);
  // 注意查询操作的书写
}
```

```
void Modify(int root, int l, int r, int pos) {
  if (l == r) {
    Tree[root] = g[id[l]];
    return;
  }
  int Mid = l + r >> 1;
  if (pos <= Mid)
    Modify(root << 1, l, Mid, pos);
  else
    Modify(root << 1 | 1, Mid + 1, r, pos);
  PushUp(root);
}

void Update(int x, int val) {
  g[x].g[1][0] += val - a[x];
  a[x] = val;
  // 首先修改 x 的 g 矩阵
  while (x) {
    matrix last = Query(1, 1, n, p[top[x]], End[top[x]]);
    // 查询 top[x] 的原本 g 矩阵
    Modify(1, 1, n,
           p[x]);  // 进行修改 (x 点的 g 矩阵已经进行修改但线段树上的未进行修改)
    matrix now = Query(1, 1, n, p[top[x]], End[top[x]]);
    // 查询 top[x] 的新 g 矩阵
    x = fa[top[x]];
    g[x].g[0][0] +=
        max(now.g[0][0], now.g[1][0]) - max(last.g[0][0], last.g[1][0]);
    g[x].g[0][1] = g[x].g[0][0];
    g[x].g[1][0] += now.g[0][0] - last.g[0][0];
    // 根据变化量修改 fa[top[x]] 的 g 矩阵
  }
}

void add(int u, int v) {
  To[++e] = v;
  Next[e] = Begin[u];
  Begin[u] = e;
}

void DFS1(int u) {
  sz[u] = 1;
  int Max = 0;
  f[u][1] = a[u];
  for (int i = Begin[u]; i; i = Next[i]) {
    int v = To[i];
    if (v == fa[u]) continue;
    dis[v] = dis[u] + 1;
    fa[v] = u;
    DFS1(v);
    sz[u] += sz[v];
    if (sz[v] > Max) {
      Max = sz[v];
      son[u] = v;
```

```
    }
    f[u][1] += f[v][0];
    f[u][0] += max(f[v][0], f[v][1]);
    // DFS1 过程中同时求出 f[i][0/1]
  }
}

void DFS2(int u, int t) {
  top[u] = t;
  p[u] = ++cnt;
  id[cnt] = u;
  End[t] = cnt;
  g[u].g[1][0] = a[u];
  g[u].g[1][1] = -INF;
  if (!son[u]) return;
  DFS2(son[u], t);
  for (int i = Begin[u]; i; i = Next[i]) {
    int v = To[i];
    if (v == fa[u] || v == son[u]) continue;
    DFS2(v, v);
    g[u].g[0][0] += max(f[v][0], f[v][1]);
    g[u].g[1][0] += f[v][0];
    // g 矩阵根据 f[i][0/1] 求出
  }
  g[u].g[0][1] = g[u].g[0][0];
}

int main() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= n; i++) scanf("%d", &a[i]);
  for (int i = 1; i <= n - 1; i++) {
    int u, v;
    scanf("%d%d", &u, &v);
    add(u, v);
    add(v, u);
  }
  dis[1] = 1;
  DFS1(1);
  DFS2(1, 1);
  Build(1, 1, n);
  for (int i = 1; i <= m; i++) {
    int x, val;
    scanf("%d%d", &x, &val);
    Update(x, val);
    matrix ans = Query(1, 1, n, 1, End[1]);  // 查询 1 所在重链的矩阵乘
    printf("%d\n", max(ans.g[0][0], ans.g[1][0]));
  }
  return 0;
}
```

习题

- SPOJ GSS3 - Can you answer these queries III[2]
- 「NOIP2018」 保卫王国[3]

- 「SDOI2017」 切树游戏[4]

参考资料与注释

[1] 洛谷 P4719【模板】动态 DP

[2] SPOJ GSS3 - Can you answer these queries III

[3] 「NOIP2018」保卫王国

[4] 「SDOI2017」切树游戏

7.13　概率 DP

引入

概率 DP 用于解决概率问题与期望问题，建议先对 概率 & 期望 的内容有一定了解。一般情况下，解决概率问题需要顺序循环，而解决期望问题使用逆序循环，如果定义的状态转移方程存在后效性问题，还需要用到 高斯消元 来优化。概率 DP 也会结合其他知识进行考察，例如 状态压缩，树上进行 DP 转移等。

DP 求概率

这类题目采用顺推，也就是从初始状态推向结果。同一般的 DP 类似的，难点依然是对状态转移方程的刻画，只是这类题目经过了概率论知识的包装。

" **例题** Codeforces 148 D Bag of mice[1]"

题目大意：袋子里有 w 只白鼠和 b 只黑鼠，公主和龙轮流从袋子里抓老鼠。谁先抓到白色老鼠谁就赢，如果袋子里没有老鼠了并且没有谁抓到白色老鼠，那么算龙赢。公主每次抓一只老鼠，龙每次抓完一只老鼠之后会有一只老鼠跑出来。每次抓的老鼠和跑出来的老鼠都是随机的。公主先抓。问公主赢的概率。

过程

设 $f_{i,j}$ 为轮到公主时袋子里有 i 只白鼠，j 只黑鼠，公主赢的概率。初始化边界，$f_{0,j}=0$ 因为没有白鼠了算龙赢，$f_{i,0}=1$ 因为抓一只就是白鼠，公主赢。考虑 $f_{i,j}$ 的转移：

- 公主抓到一只白鼠，公主赢了。概率为 $\frac{i}{i+j}$；
- 公主抓到一只黑鼠，龙抓到一只白鼠，龙赢了。概率为 $\frac{j}{i+j}\cdot\frac{i}{i+j-1}$；
- 公主抓到一只黑鼠，龙抓到一只黑鼠，跑出来一只黑鼠，转移到 $f_{i,j-3}$。概率为 $\frac{j}{i+j}\cdot\frac{j-1}{i+j-1}\cdot\frac{j-2}{i+j-2}$；
- 公主抓到一只黑鼠，龙抓到一只黑鼠，跑出来一只白鼠，转移到 $f_{i-1,j-2}$。概率为 $\frac{j}{i+j}\cdot\frac{j-1}{i+j-1}\cdot\frac{i}{i+j-2}$；

考虑公主赢的概率，第二种情况不参与计算。并且要保证后两种情况合法，所以还要判断 i,j 的大小，满足第三种情况至少要有 3 只黑鼠，满足第四种情况要有 1 只白鼠和 2 只黑鼠。

实现

" **参考实现**"

```
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;
int w, b;
double dp[1010][1010];

int main() {
  scanf("%d %d", &w, &b);
  memset(dp, 0, sizeof(dp));
  for (int i = 1; i <= w; i++) dp[i][0] = 1;  // 初始化
  for (int i = 1; i <= b; i++) dp[0][i] = 0;
  for (int i = 1; i <= w; i++) {
    for (int j = 1; j <= b; j++) {  // 以下为题面概率转移
      dp[i][j] += (double)i / (i + j);
      if (j >= 3) {
        dp[i][j] += (double)j / (i + j) * (j - 1) / (i + j - 1) * (j - 2) /
                    (i + j - 2) * dp[i][j - 3];
      }
      if (i >= 1 && j >= 2) {
        dp[i][j] += (double)j / (i + j) * (j - 1) / (i + j - 1) * i /
                    (i + j - 2) * dp[i - 1][j - 2];
      }
    }
  }
  printf("%.9lf\n", dp[w][b]);
  return 0;
}
```

习题

- CodeForces 148 D Bag of mice[1]
- POJ3071 Football[2]
- CodeForces 768 D Jon and Orbs[3]

DP 求期望

例一

" **例题** POJ2096 Collecting Bugs[4]"

题目大意：一个软件有 s 个子系统，会产生 n 种 bug。某人一天发现一个 bug，这个 bug 属于某种 bug 分类，也属于某个子系统。每个 bug 属于某个子系统的概率是 $\frac{1}{s}$，属于某种 bug 分类的概率是 $\frac{1}{n}$。求发现 n 种 bug，且 s 个子系统都找到 bug 的期望天数。

过程

令 $f_{i,j}$ 为已经找到 i 种 bug 分类，j 个子系统的 bug，达到目标状态的期望天数。这里的目标状态是找到 n 种 bug 分类，s 个子系统的 bug。那么就有 $f_{n,s} = 0$，因为已经达到了目标状态，不需要用更多的天数去发现 bug 了，于是就以目标状态为起点开始递推，答案是 $f_{0,0}$。

考虑 $f_{i,j}$ 的状态转移：

- $f_{i,j}$，发现一个 bug 属于已经发现的 i 种 bug 分类，j 个子系统，概率为 $p_1 = \frac{i}{n} \cdot \frac{j}{s}$
- $f_{i,j+1}$，发现一个 bug 属于已经发现的 i 种 bug 分类，不属于已经发现的子系统，概率为 $p_2 = \frac{i}{n} \cdot (1 - \frac{j}{s})$
- $f_{i+1,j}$，发现一个 bug 不属于已经发现 bug 分类，属于 j 个子系统，概率为 $p_3 = (1 - \frac{i}{n}) \cdot \frac{j}{s}$
- $f_{i+1,j+1}$，发现一个 bug 不属于已经发现 bug 分类，不属于已经发现的子系统，概率为 $p_4 = (1 - \frac{i}{n}) \cdot (1 - \frac{j}{s})$

再根据期望的线性性质，就可以得到状态转移方程：

$$\begin{aligned} f_{i,j} &= p_1 \cdot f_{i,j} + p_2 \cdot f_{i,j+1} + p_3 \cdot f_{i+1,j} + p_4 \cdot f_{i+1,j+1} + 1 \\ &= \frac{p_2 \cdot f_{i,j+1} + p_3 \cdot f_{i+1,j} + p_4 \cdot f_{i+1,j+1} + 1}{1 - p_1} \end{aligned}$$

实现

" 参考实现"

```
#include <cstdio>
using namespace std;
int n, s;
double dp[1010][1010];

int main() {
  scanf("%d %d", &n, &s);
  dp[n][s] = 0;
  for (int i = n; i >= 0; i--) {
    for (int j = s; j >= 0; j--) {
      if (i == n && s == j) continue;
      dp[i][j] = (dp[i][j + 1] * i * (s - j) + dp[i + 1][j] * (n - i) * j +
                  dp[i + 1][j + 1] * (n - i) * (s - j) + n * s) /
                 (n * s - i * j);  // 概率转移
    }
  }
  printf("%.4lf\n", dp[0][0]);
  return 0;
}
```

例二

" 例题 「NOIP2016」 换教室[5]"

题目大意：牛牛要上 n 个时间段的课，第 i 个时间段在 c_i 号教室，可以申请换到 d_i 号教室，申请成功的概率为 p_i，至多可以申请 m 节课进行交换。第 i 个时间段的课上完后要走到第 $i+1$ 个时间段的教室，给出一张图 v 个教室 e 条路，移动会消耗体力，申请哪几门课程可以使他因在教室间移动耗费的体力值的总和的期望值最小，也就是求出最小的期望路程和。

过程

对于这个无向连通图，先用 Floyd 求出最短路，为后续的状态转移带来便利。以移动一步为一个阶段（从第 i 个时间段到达第 $i+1$ 个时间段就是移动了一步），那么每一步就有 p_i 的概率到 d_i，不过在所有的 d_i 中只能选 m 个，有 $1 - p_i$ 的概率到 c_i，求出在 n 个阶段走完后的最小期望路程和。定义 $f_{i,j,0/1}$ 为在第 i 个时间段，连同这一个时间段已经用了 j 次换教室的机会，在这个时间段换（1）或者不换（0）教室的最小期望路程和，那么答案就是 $\min\{f_{n,i,0}, f_{n,i,1}\}, i \in [0, m]$。注意边界 $f_{1,0,0} = f_{1,1,1} = 0$。

考虑 $f_{i,j,0/1}$ 的状态转移：

- 如果这一阶段不换，即 $f_{i,j,0}$。可能是由上一次不换的状态转移来的，那么就是 $f_{i-1,j,0}+w_{c_{i-1},c_i}$，也有可能是由上一次交换的状态转移来的，这里结合条件概率和全概率的知识分析可以得到 $f_{i-1,j,1}+w_{d_{i-1},c_i}\cdot p_{i-1}+w_{c_{i-1},c_i}\cdot(1-p_{i-1})$，状态转移方程就有

$$f_{i,j,0}=min(f_{i-1,j,0}+w_{c_{i-1},c_i},f_{i-1,j,1}+w_{d_{i-1},c_i}\cdot p_{i-1}+w_{c_{i-1},c_i}\cdot(1-p_{i-1}))$$

- 如果这一阶段交换，即 $f_{i,j,1}$。类似地，可能由上一次不换的状态转移来，也可能由上一次交换的状态转移来。那么遇到不换的就乘上 $(1-p_i)$，遇到交换的就乘上 p_i，将所有会出现的情况都枚举一遍出进行计算就好了。这里不再赘述各种转移情况，相信通过上一种阶段例子，这里的状态转移应该能够很容易写出来。

实现

"参考实现"

```cpp
#include <bits/stdc++.h>

using namespace std;

const int maxn = 2010;
int n, m, v, e;
int f[maxn][maxn], c[maxn], d[maxn];
double dp[maxn][maxn][2], p[maxn];

int main() {
  scanf("%d %d %d %d", &n, &m, &v, &e);
  for (int i = 1; i <= n; i++) scanf("%d", &c[i]);
  for (int i = 1; i <= n; i++) scanf("%d", &d[i]);
  for (int i = 1; i <= n; i++) scanf("%lf", &p[i]);
  for (int i = 1; i <= v; i++)
    for (int j = 1; j < i; j++) f[i][j] = f[j][i] = 1e9;

  int u, V, w;
  for (int i = 1; i <= e; i++) {
    scanf("%d %d %d", &u, &V, &w);
    f[u][V] = f[V][u] = min(w, f[u][V]);
  }

  for (int k = 1; k <= v; k++)
    for (int i = 1; i <= v; i++)  // 前面的，按照前面的题解进行一个状态转移
      for (int j = 1; j < i; j++)
        if (f[i][k] + f[k][j] < f[i][j]) f[i][j] = f[j][i] = f[i][k] + f[k][j];

  for (int i = 1; i <= n; i++)
    for (int j = 0; j <= m; j++) dp[i][j][0] = dp[i][j][1] = 1e9;

  dp[1][0][0] = dp[1][1][1] = 0;
  for (int i = 2; i <= n; i++)  // 有后效性方程
    for (int j = 0; j <= min(i, m); j++) {
      dp[i][j][0] = min(dp[i - 1][j][0] + f[c[i - 1]][c[i]],
                        dp[i - 1][j][1] + f[c[i - 1]][c[i]] * (1 - p[i - 1]) +
                            f[d[i - 1]][c[i]] * p[i - 1]);
      if (j != 0) {
        dp[i][j][1] = min(dp[i - 1][j - 1][0] + f[c[i - 1]][d[i]] * p[i] +
                              f[c[i - 1]][c[i]] * (1 - p[i]),
```

```
                            dp[i - 1][j - 1][1] +
                                f[c[i - 1]][c[i]] * (1 - p[i - 1]) * (1 - p[i]) +
                                f[c[i - 1]][d[i]] * (1 - p[i - 1]) * p[i] +
                                f[d[i - 1]][c[i]] * (1 - p[i]) * p[i - 1] +
                                f[d[i - 1]][d[i]] * p[i - 1] * p[i]);
        }
      }

    double ans = 1e9;
    for (int i = 0; i <= m; i++) ans = min(dp[n][i][0], min(dp[n][i][1], ans));
    printf("%.2lf", ans);

    return 0;
}
```

比较这两个问题可以发现，DP 求期望题目在对具体是求一个值或是最优化问题上会对方程得到转移方式有一些影响，但无论是 DP 求概率还是 DP 求期望，总是离不开概率知识和列出、化简计算公式的步骤，在写状态转移方程时需要思考的细节也类似。

习题

- POJ2096 Collecting Bugs[4]
- HDU3853 LOOPS[6]
- HDU4035 Maze[7]
- 「NOIP2016」 换教室[5]
- 「SCOI2008」 奖励关[8]

有后效性 DP

"CodeForces 24 D Broken robot[9]"

题目大意：给出一个 $n \times m$ 的矩阵区域，一个机器人初始在第 x 行第 y 列，每一步机器人会等概率地选择停在原地，左移一步，右移一步，下移一步，如果机器人在边界则不会往区域外移动，问机器人到达最后一行的期望步数。

过程

在 $m = 1$ 时每次有 $\frac{1}{2}$ 的概率不动，有 $\frac{1}{2}$ 的概率向下移动一格，答案为 $2 \cdot (n - x)$。设 $f_{i,j}$ 为机器人机器人从第 i 行第 j 列出发到达第 n 行的期望步数，最终状态为 $f_{n,j} = 0$。由于机器人会等概率地选择停在原地，左移一步，右移一步，下移一步，考虑 $f_{i,j}$ 的状态转移：

- $f_{i,1} = \frac{1}{3} \cdot (f_{i+1,1} + f_{i,2} + f_{i,1}) + 1$
- $f_{i,j} = \frac{1}{4} \cdot (f_{i,j} + f_{i,j-1} + f_{i,j+1} + f_{i+1,j}) + 1$
- $f_{i,m} = \frac{1}{3} \cdot (f_{i,m} + f_{i,m-1} + f_{i+1,m}) + 1$

在行之间由于只能向下移动，是满足无后效性的。在列之间可以左右移动，在移动过程中可能产生环，不满足无后效性。将方程变换后可以得到：

- $2f_{i,1} - f_{i,2} = 3 + f_{i+1,1}$
- $3f_{i,j} - f_{i,j-1} - f_{i,j+1} = 4 + f_{i+1,j}$
- $2f_{i,m} - f_{i,m-1} = 3 + f_{i+1,m}$

由于是逆序的递推，所以每一个 $f_{i+1,j}$ 是已知的。由于有 m 列，所以右边相当于是一个 m 行的列向量，那么左边就是 m 行 m 列的矩阵。使用增广矩阵，就变成了 m 行 m+1 列的矩阵，然后进行 高斯消元 即可解出答案。

实现

" 参考实现"

```
#include <bits/stdc++.h>
using namespace std;

const int maxn = 1e3 + 10;

double a[maxn][maxn], f[maxn];
int n, m;

void solve(int x) {
  memset(a, 0, sizeof a);
  for (int i = 1; i <= m; i++) {
    if (i == 1) {
      a[i][i] = 2;
      a[i][i + 1] = -1;
      a[i][m + 1] = 3 + f[i];
      continue;
    } else if (i == m) {
      a[i][i] = 2;
      a[i][i - 1] = -1;
      a[i][m + 1] = 3 + f[i];
      continue;
    }
    a[i][i] = 3;
    a[i][i + 1] = -1;
    a[i][i - 1] = -1;
    a[i][m + 1] = 4 + f[i];
  }

  for (int i = 1; i < m; i++) {
    double p = a[i + 1][i] / a[i][i];
    a[i + 1][i] = 0;
    a[i + 1][i + 1] -= a[i][i + 1] * p;
    a[i + 1][m + 1] -= a[i][m + 1] * p;
  }

  f[m] = a[m][m + 1] / a[m][m];
  for (int i = m - 1; i >= 1; i--)
    f[i] = (a[i][m + 1] - f[i + 1] * a[i][i + 1]) / a[i][i];
}

int main() {
  scanf("%d %d", &n, &m);
  int st, ed;
  scanf("%d %d", &st, &ed);
  if (m == 1) {
    printf("%.10f\n", 2.0 * (n - st));
```

```
        return 0;
    }
    for (int i = n - 1; i >= st; i--) {
        solve(i);
    }
    printf("%.10f\n", f[ed]);
    return 0;
}
```

习题

- CodeForce 24 D Broken robot[9]
- HDU Time Travel[10]
- 「HNOI2013」 游走[11]

参考文献

kuangbin 概率 DP 总结[12]

参考资料与注释

[1] Codeforces 148 D Bag of mice [1-1] [1-2]

[2] POJ3071 Football

[3] CodeForces 768 D Jon and Orbs

[4] POJ2096 Collecting Bugs [4-1] [4-2]

[5] 「NOIP2016」换教室 [5-1] [5-2]

[6] HDU3853 LOOPS

[7] HDU4035 Maze

[8] 「SCOI2008」奖励关

[9] CodeForces 24 D Broken robot [9-1] [9-2]

[10] HDU Time Travel

[11] 「HNOI2013」游走

[12] kuangbin 概率 DP 总结

7.14 DP 优化

7.14.1 单调队列/单调栈优化

Authors: Marcythm, hsfzLZH1, Ir1d, greyqz, Anguei, billchenchina, Chrogeek, ChungZH

介绍

前置知识：单调队列、单调栈。

" **例题** CF372C Watching Fireworks is Fun[1]"

题目大意：城镇中有 n 个位置，有 m 个烟花要放。第 i 个烟花放出的时间记为 t_i，放出的位置记为 a_i。如果烟花放出的时候，你处在位置 x，那么你将收获 $b_i - |a_i - x|$ 点快乐值。

初始你可在任意位置，你每个单位时间可以移动不大于 d 个单位距离。现在你需要最大化你能获得的快乐值。

设 $f_{i,j}$ 表示在放第 i 个烟花时，你的位置在 j 所能获得的最大快乐值。

写出**状态转移方程**：$f_{i,j} = \max\{f_{i-1,k} + b_i - |a_i - j|\}$

这里的 k 是有范围的，$j - (t_i - t_{i-1}) \times d \le k \le j + (t_i - t_{i-1}) \times d$。

我们尝试将状态转移方程进行变形：

由于 max 里出现了一个确定的常量 b_i，我们可以将它提到外面去。

$f_{i,j} = \max\{f_{i-1,k} + b_i - |a_i - j|\} = \max\{f_{i-1,k} - |a_i - j|\} + b_i$

如果确定了 i 和 j 的值，那么 $|a_i - j|$ 的值也是确定的，也可以将这一部分提到外面去。

最后，式子变成了这个样子：$f_{i,j} = \max\{f_{i-1,k} - |a_i - j|\} + b_i = \max\{f_{i-1,k}\} - |a_i - j| + b_i$

看到这一熟悉的形式，我们想到了什么？**单调队列优化**。由于最终式子中的 max 只和上一状态中连续的一段的最大值有关，所以我们在计算一个新的 i 的状态值时候只需将原来的 f_{i-1} 构造成一个单调队列，并维护单调队列，使得其能在均摊 $O(1)$ 的时间复杂度内计算出 $\max\{f_{i-1,k}\}$ 的值，从而根据公式计算出 $f_{i,j}$ 的值。

总的时间复杂度为 $O(nm)$。

" **参考代码**"

```
#include <algorithm>
#include <cstring>
#include <iostream>
using namespace std;
typedef long long ll;

const int maxn = 150000 + 10;
const int maxm = 300 + 10;

ll f[2][maxn];
ll a[maxm], b[maxm], t[maxm];
int n, m, d;

int que[maxn];

int fl = 1;
```

```cpp
void init() {
  memset(f, 207, sizeof(f));
  memset(que, 0, sizeof(que));
  for (int i = 1; i <= n; i++) f[0][i] = 0;
  fl = 1;
}

void dp() {
  init();
  for (int i = 1; i <= m; i++) {
    int l = 1, r = 0, k = 1;
    for (int j = 1; j <= n;
         j++) {  // 在这里使用了单调队列的优化，推式子详见上面
      for (; k <= min(1ll * n, j + d * (t[i] - t[i - 1])); k++) {
        while (l <= r && f[fl ^ 1][que[r]] <= f[fl ^ 1][k]) r--;
        que[++r] = k;
      }

      while (l <= r && que[l] < max(1ll, j - d * (t[i] - t[i - 1]))) l++;
      f[fl][j] = f[fl ^ 1][que[l]] - abs(a[i] - j) + b[i];
    }

    fl ^= 1;
  }
}

int main() {
  cin >> n >> m >> d;
  for (int i = 1; i <= m; i++) cin >> a[i] >> b[i] >> t[i];

  // then dp
  dp();
  ll ans = -1e18;
  for (int i = 1; i <= n; i++) ans = max(ans, f[fl ^ 1][i]);
  cout << ans << endl;
  return 0;
}
```

讲完了，让我们归纳一下单调队列优化动态规划问题的基本形态：当前状态的所有值可以从上一个状态的某个连续的段的值得到，要对这个连续的段进行 RMQ 操作，相邻状态的段的左右区间满足非降的关系。

单调队列优化多重背包

" 问题描述"

你有 n 个物品，每个物品重量为 w_i，价值为 v_i，数量为 k_i。你有一个承重上限为 W 的背包，现在要求你在不超过重量上限的情况下选取价值和尽可能大的物品放入背包。求最大价值。

不了解背包 DP 的请先阅读 背包 DP。设 $f_{i,j}$ 表示前 i 个物品装入承重为 j 的背包的最大价值，朴素的转移方程为

$$f_{i,j} = \max_{k=0}^{k_i}(f_{i-1,j-k\times w_i} + v_i \times k)$$

时间复杂度 $O(W\sum k_i)$。

考虑优化 f_i 的转移。为方便表述，设 $g_{x,y}=f_{i,x\times w_i+y}, g'_{x,y}=f_{i-1,x\times w_i+y}$，则转移方程可以表示为：

$$g_{x,y}=\max_{k=0}^{k_i}(g'_{x-k,y}+v_i\times k)$$

设 $G_{x,y}=g'_{x,y}-v_i\times x$。则方程可以表示为：

$$g_{x,y}=\max_{k=0}^{k_i}(G_{x-k,y})+v_i\times x$$

这样就转化为一个经典的单调队列优化形式了。$G_{x,y}$ 可以 $O(1)$ 计算，因此对于固定的 y，我们可以在 $O\left(\left\lfloor\dfrac{W}{w_i}\right\rfloor\right)$ 的时间内计算出 $g_{x,y}$。因此求出所有 $g_{x,y}$ 的复杂度为 $O\left(\left\lfloor\dfrac{W}{w_i}\right\rfloor\right)\times O(w_i)=O(W)$。这样转移的总复杂度就降为 $O(nW)$。

习题

- 「Luogu P1886」 滑动窗口[2]
- 「NOI2005」 瑰丽华尔兹[3]
- 「SCOI2010」 股票交易[4]

参考资料与注释

[1] CF372C Watching Fireworks is Fun

[2] 「Luogu P1886」滑动窗口

[3] 「NOI2005」瑰丽华尔兹

[4] 「SCOI2010」股票交易

7.14.2 斜率优化

> **Authors:** Marcythm, hsfzLZH1, abc1763613206, greyqz, Ir1d, billchenchina, Chrogeek, Enter-tainer, StudyingFather, Mr-FoodinChina, luoguyuntianming, sshwy, wood3

例题引入

> "「HNOI2008」 **玩具装箱**[1]"
>
> 有 n 个玩具，第 i 个玩具价值为 c_i。要求将这 n 个玩具排成一排，分成若干段。对于一段 $[l,r]$，它的代价为 $(r-l+\sum_{i=l}^{r}c_i-L)^2$。其中 L 是一个常量，求分段的最小代价。
>
> $1\le n\le 5\times 10^4, 1\le L, c_i\le 10^7$。

朴素的 DP 做法

令 f_i 表示前 i 个物品，分若干段的最小代价。

状态转移方程：$f_i=\min_{j<i}\{f_j+(i-(j+1)+pre_i-pre_j-L)^2\}=\min_{j<i}\{f_j+(pre_i-pre_j+i-j-1-L)^2\}$。

其中 pre_i 表示前 i 个数的和，即 $\sum_{j=1}^{i}c_j$。

该做法的时间复杂度为 $O(n^2)$，无法解决本题。

优化

考虑简化上面的状态转移方程式：令 $s_i = pre_i + i, L' = L + 1$，则 $f_i = \min_{j<i}\{f_j + (s_i - s_j - L')^2\}$。

将与 j 无关的移到外面，我们得到

$$f_i - (s_i - L')^2 = \min_{j<i}\{f_j + s_j^2 + 2s_j(L' - s_i)\}$$

考虑一次函数的斜截式 $y = kx + b$，将其移项得到 $b = y - kx$。我们将与 j 有关的信息表示为 y 的形式，把同时与 i, j 有关的信息表示为 kx，把要最小化的信息（与 i 有关的信息）表示为 b，也就是截距。具体地，设

$$\begin{aligned}
x_j &= s_j \\
y_j &= f_j + s_j^2 \\
k_i &= -2(L' - s_i) \\
b_i &= f_i - (s_i - L')^2
\end{aligned}$$

则转移方程就写作 $b_i = \min_{j<i}\{y_j - k_i x_j\}$。我们把 (x_j, y_j) 看作二维平面上的点，则 k_i 表示直线斜率，b_i 表示一条过 (x_j, y_j) 的斜率为 k_i 的直线的截距。问题转化为了，选择合适的 j（$1 \le j < i$），最小化直线的截距。

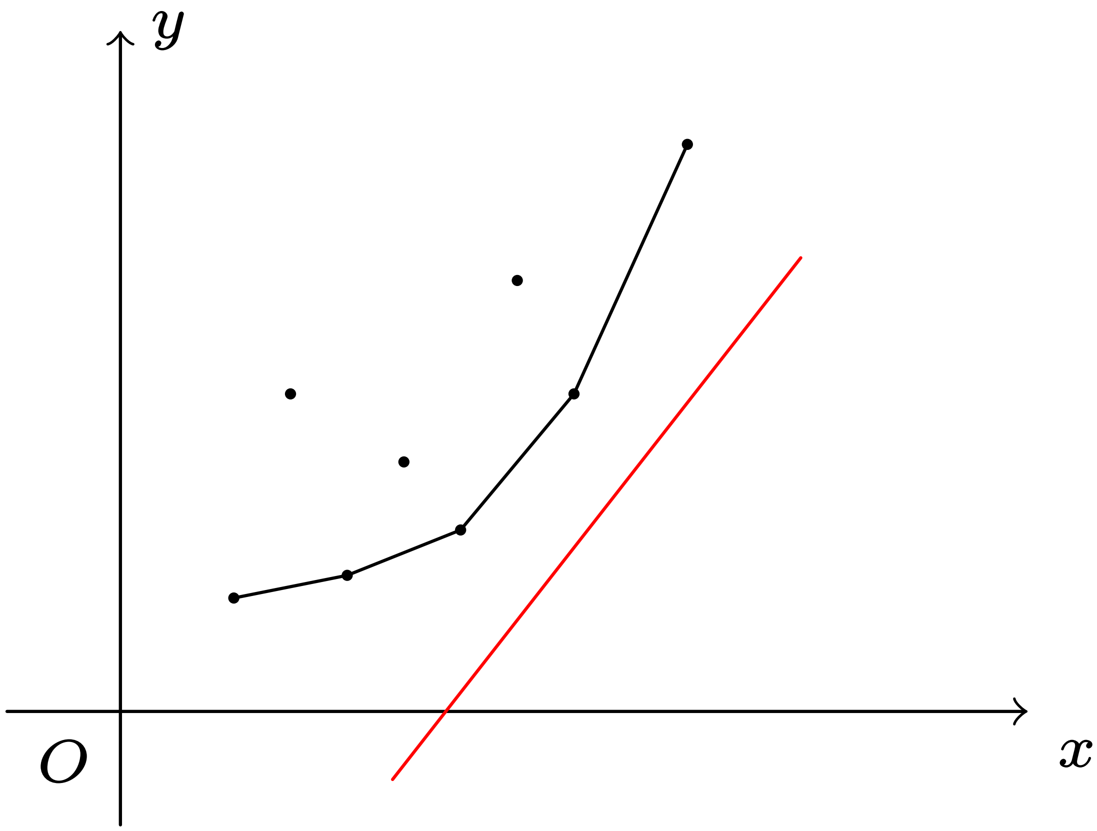


图 7.12 slope_optimization

如图，我们将这个斜率为 k_i 的直线从下往上平移，直到有一个点 (x_p, y_p) 在这条直线上，则有 $b_i = y_p - k_i x_p$，这时 b_i 取到最小值。算完 f_i，我们就把 (x_i, y_i) 这个点加入点集中，以做为新的 DP 决策。那么，我们该如何维护点集？

容易发现，可能让 b_i 取到最小值的点一定在下凸壳上。因此在寻找 p 的时候我们不需要枚举所有 $i-1$ 个点，只需要考虑凸包上的点。而在本题中 k_i 随 i 的增加而递增，因此我们可以单调队列维护凸包。

具体地，设 $K(a, b)$ 表示过 (x_a, y_a) 和 (x_b, y_b) 的直线的斜率。考虑队列 $q_l, q_{l+1}, \ldots, q_r$，维护的是下凸壳上的点。也就是说，对于 $l < i < r$，始终有 $K(q_{i-1}, q_i) < K(q_i, q_{i+1})$ 成立。

我们维护一个指针 e 来计算 b_i 最小值。我们需要找到一个 $K(q_{e-1}, q_e) \le k_i < K(q_e, q_{e+1})$ 的 e（特别地，当 $e = l$ 或者 $e = r$ 时要特别判断），这时就有 $p = q_e$，即 q_e 是 i 的最优决策点。由于 k_i 是单调递减的，因此 e 的移动次数是均摊 $O(1)$ 的。

在插入一个点 (x_i, y_i) 时，我们要判断是否 $K(q_{r-1}, q_r) < K(q_r, i)$，如果不等式不成立就将 q_r 弹出，直到等式满足。然后将 i 插入到 q 队尾。

这样我们就将 DP 的复杂度优化到了 $O(n)$。

概括一下上述斜率优化模板题的算法：

1. 将初始状态入队。
2. 每次使用一条和 i 相关的直线 $f(i)$ 去切维护的凸包，找到最优决策，更新 dp_i。
3. 加入状态 dp_i。如果一个状态（即凸包上的一个点）在 dp_i 加入后不再是凸包上的点，需要在 dp_i 加入前将其剔除。

接下来我们介绍斜率优化的进阶应用，将斜率优化与二分/分治/数据结构等结合，来维护性质不那么好（缺少一些单调性性质）的 DP 方程。

二分/CDQ/平衡树优化 DP

当我们在 i 这个点寻找最优决策时，会使用一个和 i 相关的直线 $f(i)$ 去切我们维护的凸包。切到的点即为最优决策。

在上述例题中，直线的斜率随 i 单调变化，但是对于有些问题，斜率并不是单调的。这时我们需要维护凸包上的每一个节点，然后每次用当前的直线去切这个凸包。这个过程可以使用二分解决，因为凸包上相邻两个点的斜率是有单调性的。

" 玩具装箱改"

有 n 个玩具，第 i 个玩具价值为 c_i。要求将这 n 个玩具排成一排，分成若干段。对于一段 $[l, r]$，它的代价为 $(r - l + \sum_{i=l}^{r} c_i - L)^2$。其中 L 是一个常量，求分段的最小代价。

$1 \le n \le 5 \times 10^4, 1 \le L \le 10^7, -10^7 \le c_i \le 10^7$。

本题与「玩具装箱」问题唯一的区别是，玩具的价值可以为负。延续之前的思路，令 f_i 表示前 i 个物品，分若干段的最小代价。

状态转移方程：$f_i = \min_{j<i}\{f_j + (pre_i - pre_j + i - j - 1 - L)^2\}$。

其中 $pre_i = \sum_{j=1}^{i} c_j$。

将方程做相同的变换

$$f_i - (s_i - L')^2 = \min_{j<i}\{f_j + s_j^2 + 2s_j(L' - s_i)\}$$

然而这时有两个条件不成立了：

1. 直线的斜率不再单调；
2. 每次加入的决策点的横坐标不再单调。

仍然考虑凸壳的维护。

在寻找最优决策点，也就是用直线切凸壳的时候，我们将单调队列找队首改为：凸壳上二分。我们二分出斜率最接近直线斜率的那条凸壳边，就可以找到最优决策。

在加入决策点，也就是凸壳上加一个点的时候，我们有两种方法维护。

第一种方法是直接用平衡树维护凸壳。那么寻找决策点的二分操作就转化为在平衡树上二分，插入决策点就转化为在平衡树上插入一个结点，并删除若干个被踢出凸壳的点。此方法思路简洁但实现繁琐。

下面介绍一种基于 CDQ 分治 的做法。

设 $\text{CDQ}(l, r)$ 代表计算 $f_i, i \in [l, r]$。考虑 $\text{CDQ}(1, n)$：

- 我们先调用 $\text{CDQ}(1, mid)$ 算出 $f_i, i \in [1, mid]$。然后我们对 $[1, mid]$ 这个区间内的决策点建凸壳，然后使用这个凸壳去更新 $f_i, i \in [mid + 1, n]$。这时我们决策点集是固定的，不像之前那样边计算 DP 值边加入决策点，那么我们就可以把 $i \in [mid + 1, n]$ 的 f_i 先按照直线的斜率 k_i 排序，然后就可以使用单调队列来计算 DP 值了。当

然，也可以在静态凸壳上二分计算 DP 值。

- 对于 $[mid+1,n]$ 中的每个点，如果它的最优决策的位置是在 $[1,mid]$ 这个区间，在这一步操作中他就会被更新成最优答案。当执行完这一步操作时，我们发现 $[1,mid]$ 中的所有点已经发挥了全部的作用，凸壳中他们存不存在已经不影响之后的答案更新。因此我们可以直接舍弃这个区间的决策点，并使用 CDQ$(mid+1,n)$ 解决右区间剩下的问题。

时间复杂度 $O(n\log^2 n)$。

对比「玩具装箱」和「玩具装箱改」，可以总结出以下两点：

- 二分/CDQ/平衡树等能够优化 DP 方程的计算，于一定程度上降低复杂度，但不能改变这个方程本身。
- DP 方程的性质会取决于数据的特征，但 DP 方程本身取决于题目中的数学模型。

小结

斜率优化 DP 需要灵活运用，其宗旨是将最优化问题转化为二维平面上与凸包有关的截距最值问题。遇到性质不太好的方程，有时需要辅以数据结构来加以解决，届时还请就题而论。

习题

- 「SDOI2016」 征途[2]
- 「ZJOI2007」 仓库建设[3]
- 「APIO2010」 特别行动队[4]
- 「JSOI2011」 柠檬[5]
- 「Codeforces 311B」Cats Transport[6]
- 「NOI2007」 货币兑换[7]
- 「NOI2019」 回家路线[8]
- 「NOI2016」 国王饮水记[9]
- 「NOI2014」 购票[10]

参考资料与注释

[1] 「HNOI2008」玩具装箱

[2] 「SDOI2016」征途

[3] 「ZJOI2007」仓库建设

[4] 「APIO2010」特别行动队

[5] 「JSOI2011」柠檬

[6] 「Codeforces 311B」Cats Transport

[7] 「NOI2007」货币兑换

[8] 「NOI2019」回家路线

[9] 「NOI2016」国王饮水记

[10] 「NOI2014」购票

7.14.3 四边形不等式优化

Authors: Marcythm, zyf0726, hsfzLZH1, MingqiHuang, Ir1d, greyqz, billchenchina, Chrogeek, StudyingFather, NFLSCode, c-forrest

四边形不等式优化利用的是状态转移方程中的决策单调性。

基础知识

考虑最简单的情形，我们要解决如下一系列最优化问题。

$$f(i) = \min_{1 \le j \le i} w(j,i) \qquad (1 \le i \le n) \tag{1}$$

这里假定成本函数 $w(j,i)$ 可以在 $O(1)$ 时间内计算。

" 约定"

动态规划的状态转移方程经常可以写作一系列最优化问题的形式。以（1）式为例，这些问题含有参数 i，问题的目标函数和可行域都可以依赖于 i。每一个问题都是在给定参数 i 时，选取某个可行解 j 来最小化目标函数的取值。为表述方便，下文将参数为 i 的最优化问题简称为「问题 i」，该最优化问题的可行解 j 称为「决策 j」，目标函数在最优解处取得的值则称为「状态 $f(i)$」。同时，记问题 i 对应的最小最优决策点为 $\mathrm{opt}(i)$。

在一般的情形下，这些问题总时间复杂度为 $O(n^2)$。这是由于对于问题 i，我们需要考虑所有可能的决策 j。而在满足决策单调性时，可以有效缩小决策空间，优化总复杂度。

- **决策单调性**：对于任意 $i_1 < i_2$，必然成立 $\mathrm{opt}(i_1) \le \mathrm{opt}(i_2)$。

" 附注"

对于问题 i，最优决策集合未必是一个区间。决策单调性实际可以定义在最优决策集合上。对于集合 A 和 B，可以定义 $A \le B$ 当且仅当对于任意 $a \in A$ 和 $b \in B$，成立 $\min\{a,b\} \in A$ 和 $\max\{a,b\} \in B$。这蕴含最小（最大）最优决策点的单调性，即此处采取的定义。本文关于最小最优决策点叙述的结论，同样适用于最大最优决策点。但是，存在情形，某更大问题的最小最优决策严格小于另一更小问题的最大最优决策，亦即可能对某些 $i_1 < i_2$ 成立 $\mathrm{optmax}(i_1) > \mathrm{optmin}(i_2)$，所以在书写代码时，应保证总是求得最小或最大的最优决策点。

另一方面，拥有相同最小最优决策的问题构成一个区间。这一区间，作为最小最优决策的函数，应严格递增。亦即，给定 $j_1 = \mathrm{opt}(i_1)$，$j_2 = \mathrm{opt}(i_2)$，如果 $j_1 < j_2$，那么必然有 $i_1 < i_2$。换言之，如果决策 $j_1 < j_2$ 能够成为最小最优决策的问题区间分别是 $[l_{j_1}, r_{j_1}]$ 和 $[l_{j_2}, r_{j_2}]$，那么必然有 $r_{j_1} < l_{j_2}$。

最常见的判断决策单调性的方法是通过四边形不等式（quadrangle inequality）。

- **四边形不等式**：如果对于任意 $a \le b \le c \le d$ 均成立

$$w(a,c) + w(b,d) \le w(a,d) + w(b,c),$$

则称函数 w 满足四边形不等式（简记为「交叉小于包含」）。若等号永远成立，则称函数 w 满足**四边形恒等式**。

如果没有特别说明，以下都会保证 $a \le b \le c \le d$。四边形不等式给出了一个决策单调性的充分不必要条件。

" 定理 1"

若 w 满足四边形不等式，则问题 (1) 满足决策单调性。

" 证明"

要证明这一点，可采用反证法。假设对某些 $c < d$，成立 $a = \mathrm{opt}(d) < \mathrm{opt}(c) = b$。此时有 $a < b \le c < d$。根据最优化条件，$w(a,d) \le w(b,d)$ 且 $w(b,c) < w(a,c)$，于是，$w(a,d) - w(b,d) \le 0 < w(a,c) - w(b,c)$，这与四边形不等式矛盾。

四边形不等式可以理解在合理的定义域内，w 的二阶混合差分 $\Delta_i\Delta_j w(j,i)$ 非正。

利用决策单调性，有两种常见算法可以将算法复杂度优化到 $O(n\log n)$。

分治

要求解所有状态，只需要求解所有最优决策点。为了对所有 $1 \le i \le n$ 求解 $\mathrm{opt}(i)$，首先计算 $\mathrm{opt}(n/2)$，而后分别计算 $1 \le i < n/2$ 和 $n/2 < i \le n$ 上的 $\mathrm{opt}(i)$，注意此时已知前半段的 $\mathrm{opt}(i)$ 必然位于 1 和 $\mathrm{opt}(n/2)$ 之间（含端点），而后半段的 $\mathrm{opt}(i)$ 必然位于 $\mathrm{opt}(n/2)$ 和 $\mathrm{opt}(n)$ 之间（含端点）。对于两个子区间，也类似处理，直至计算出每个问题的最优决策。在分治的过程中记录搜索的上下边界，就可以保证算法复杂度控制在 $O(n\log n)$。递归树层数为 $O(\log n)$，而每层中，单个决策点至多计算两次，所以总的计算次数是 $O(n\log n)$。

" 核心代码"

```cpp
int w(int j, int i);

void DP(int l, int r, int k_l, int k_r) {
  int mid = (l + r) / 2, k = k_l;
  // 求状态 f[mid] 的最优决策点
  for (int j = k_l; j <= min(k_r, mid - 1); ++j)
    if (w(j, mid) < w(k, mid)) k = j;
  f[mid] = w(k, mid);
  // 根据决策单调性得出左右两部分的决策区间，递归处理
  if (l < mid) DP(l, mid - 1, k_l, k);
  if (r > mid) DP(mid + 1, r, k, k_r);
}
```

```python
def DP(l, r, k_l, k_r):
    mid = int((l + r) / 2)
    k = k_l          # 求状态 f[mid] 的最优决策点
    for i in range(k_l, min(k_r, mid - 1)):
        if w(i, mid) < w(k, mid):
            k = i
    f[mid] = w(k, mid)          # 根据决策单调性得出左右两部分的决策区间，递归处理
    if l < mid:
        DP(l, mid - 1, k_l, k)
    if r > mid:
        DP(mid + 1, r, k, k_r)
```

二分队列

注意到对于每个决策点 j，能使其成为最小最优决策点的问题 i 必然构成一个区间。可以通过单调队列记录到目前为止每个决策点可以解决的问题的区间，这样，问题的最优解自然可以通过队列中记录的决策点计算得到。算法大致如下。

" 核心代码"

```cpp
int val(int j, int i);
int lt[N], rt[N], f[N];
```

```cpp
deque<int> dq;
// 初始化队列
dq.emplace_back(1);
lt[1] = 1;
rt[n] = n;
// 顺次考虑所有问题和决策
for (int j = 1; j <= n; ++j) {
  // 出队
  while (!dq.empty() && rt[dq.front()] < j) {
    dq.pop_front();
  }
  // 计算
  f[j] = val(dq.front(), j);
  // 入队
  while (!dq.empty() && val(j, lt[dq.back()]) < val(dq.back(), lt[dq.back()])) {
    dq.pop_back();
  }
  if (dq.empty()) {
    dq.emplace_back(j);
    lt[j] = j + 1;
    rt[j] = n;
  } else if (val(j, rt[dq.back()]) < val(dq.back(), rt[dq.back()])) {
    if (rt[dq.back()] < n) {
      dq.emplace_back(j);
      lt[j] = rt[dq.back()] + 1;
      rt[j] = n;
    }
  } else {
    int ll = lt[dq.back()];
    int rr = rt[dq.back()];
    int i;
    // 二分
    while (ll <= rr) {
      int mm = (ll + rr) / 2;
      if (val(j, mm) < val(dq.back(), mm)) {
        i = mm;
        rr = mm - 1;
      } else {
        ll = mm + 1;
      }
    }
    rt[dq.back()] = i - 1;
    dq.emplace_back(j);
    lt[j] = i;
    rt[j] = n;
  }
}
```

掌握这一算法，需要理解如下要点：

- 队列需要记录到目前为止每个可行的决策点 j 和能够解决的问题区间左右端点 l_j 和 r_j 构成的三元组。对于给定区间 $[l_j, r_j]$ 内的问题，j 应该是到目前为止考虑过的决策点中最小最优的（以下简称最优决策）。每时每刻，队列中存储的决策未必是连续的，但是尚未解决的问题应该是队列中存储的问题区间的不交并。
- **初始化**：将首个决策放于队列中，并记录它对于所有问题都是最优的。
- 类似于单调队列，每次考虑下一个决策 j 的时候，都需要进行出队和入队操作。

- **出队**：当所有决策 $j \le i$ 都考虑结束后，问题 i 的解就是队列中首个满足 $l_j \le i \le r_j$ 的决策点 j。此时可以弹出所有满足 $r_j < i$ 的队首。由于决策单调性，弹出的决策也不会是后续问题的最优决策。
- **入队**：要对决策 j 进行入队时，首先比较它和队尾的决策 j'。
  - 如果对于问题 $l_{j'}$，将入队的决策 j 比已有的决策 j' 更优，即 $w(j, l_{j'}) < w(j', l_{j'})$ 时，则弹出队尾的决策 j'。此操作持续到队尾的决策 j' 比起 j 对于问题 $l_{j'}$ 更优时为止。
  - 如果队列已空，入队 $(j, j+1, n)$，即认为决策 j 是尚未解决的所有问题的最优解。
  - 如果队尾决策 j' 对于问题 $r_{j'}$ 同样优于将入队的决策 j，那么当 $r_{j'} < n$ 时，入队 $(j, r_{j'}+1, n)$，表示 j 是对于问题 $[r_{j'}+1, n]$ 的最优解，否则，不需要入队 j，因为它并不比已有的决策更优。
  - 最后的情形是，队尾决策 j' 比起要入队的决策 j 对于问题 $l_{j'}$ 更优，而对于问题 $r_{j'}$ 更劣，那么，需要通过二分找到最小的 $i \in [l_{j'}, r_{j'}]$ 使得 $w(j,i) < w(j',i)$，将队尾的区间右端点修改为 $i-1$，并入队 (j, i, n)。

类似于单调队列，每个决策点至多入队一次，出队一次。这里，出队是 $O(1)$ 的，而入队是 $O(\log n)$ 的（可能需要二分），所以总的时间复杂度是 $O(n \log n)$。

" **例题** 1：「POI2011」Lightning Conductor[1]"

给定一个长度为 n 的序列 $a_1, a_2, \cdots, a_n$，要求对于每一个 $1 \le i \le n$，找到最小的非负整数 f_i 满足

$$\forall j \in [1, n] : a_j \le a_i + f_i - \sqrt{|i-j|}.$$

" **参考思路**"

显然，经过不等式变形，我们可以得到待求整数 $f_i = \max_j\{a_j + \sqrt{|i-j|} - a_i\}$。不妨先考虑 $j \le i$ 的情况（另外一种情况类似），此时我们可以得到状态转移方程：

$$f_i = \min_{j \le i}\{-a_j - \sqrt{i-j} + a_i\}.$$

根据 $-\sqrt{x}$ 的凸性，我们很容易得出（后文将详细描述）函数 $w(l, r) = -a_l - \sqrt{r-l} + a_r$ 满足四边形不等式，因此套用上述的算法便可在 $O(n \log n)$ 的时间内解决此题了。

区间分拆问题

考虑将某个区间拆分成若干个子区间的问题。形式化地说，将给定区间 $[1, n]$ 拆分成 $[a_1, b_1], \cdots, [a_k, b_k]$，其中，$b_1 = 1$，$a_k = n$，以及 $b_i + 1 = a_{i+1}$ 对任意 $i < k$ 都成立。对于给定拆分，成本为 $\sum_{i=1}^{k} w(a_i, b_i)$。问题要求最小化这一成本。可以列出如下的 1D1D 状态转移方程。

$$f(i) = \min_{1 \le j \le i} f(j-1) + w(j, i) \qquad (1 \le i \le n)$$

这里，$f(0) = 0$。注意到，只要 $w(j, i)$ 满足四边形不等式，$f(j-1) + w(j, i)$ 必然满足四边形不等式，因为第一项并不包括 j 和 i 的交叉项，在混合差分时会消去。但是由于成本函数依赖于前面的子问题，这一转移只能够顺序计算，所以通常只适合应用二分队列算法。算法复杂度为 $O(n \log n)$。

" **例题** 2：「HNOI2008」 **玩具装箱** toy[2]"

有 n 个玩具需要装箱，要求每个箱子中的玩具编号必须是连续的。每个玩具有一个长度 C_i，如果一个箱子中有多个玩具，那么每两个玩具之间要加入一个单位长度的分隔物。形式化地说，如果将编号在 $[j, i]$ 间的玩具装在一个箱子里，那么这个箱子的长度为 $i - j + \sum_{k=j}^{i} C_k$。现在需要制定一个装箱方案，使得所有容器的长度与 K 差值的平方之和最小。

" **参考思路**"

设 f_i 表示将前 i 个玩具装箱的最小代价，则枚举第 i 个玩具与哪些玩具放在一个箱子中，可以得到状态转

移方程为

$$f_i = \min_{j=1}^{i} f_{j-1} + \left(i - j + \sum_{k=j}^{i} C_k \right)^2.$$

记 $s(i) = i + \sum_{k=1}^{i} C_k$，则有 $w(j,i) = (s(i) - s(j-1) - 1 - K)^2$。显然 $s(i)$ 单调增加，因此根据下文的性质 1 和性质 2 可知 $s(i) - s(j-1) - 1 - K$ 满足区间包含单调性和四边形不等式。再根据 x^2 的单调性和凸性以及性质 3 可知，$w(j,i)$ 也满足四边形不等式，此时使用二分队列优化即可。

限制区间个数的情形

上述问题可以加强为限制区间个数的情形，即问题指定将区间拆分成 m 个子区间。此时需要将拆分后的区间个数作为转移状态的一维。相应地，有 2D1D 状态转移方程如下。

$$f(k,i) = \min_{1 \le j \le i} f(k-1, j-1) + w(j,i) \qquad (1 \le k \le m,\ 1 \le i \le n) \tag{2}$$

这里，$f(0,0) = 0$，$f(0,i) = f(k,0) = \infty$ 对任意 $1 \le k \le m$ 和 $1 \le i \le n$ 都成立。和上文同样的道理，这里的 $f(k-1, j-1) + w(j,i)$ 必然满足四边形不等式。此时对于第 i 层的计算，并不再依赖于该层的结果，所以对于每一层，都可以通过分治或者二分队列的方法进行计算，此时算法复杂度为 $O(mn \log n)$。

对于这一问题，利用决策单调性，实际上还存在其他的优化算法。第二种优化思路依赖于如下结果。这种优化算法和下文详细描述的 Knuth 优化算法十分相似。

" 定理 2"

若 w 满足四边形不等式，则对于问题 (2) 成立 $\operatorname{opt}(k-1,i) \le \operatorname{opt}(k,i) \le \operatorname{opt}(k,i+1)$。

" 证明"

第二个不等式只是第 k 层的决策单调性。关键在于第一个不等式。

下证 $\operatorname{opt}(k,i) \le \operatorname{opt}(k+1,i)$。假设有如下两个区间 $[1,i]$ 的分划（逆序标号）：$[a_k,d_k],\cdots,[a_1,d_1]$ 和 $[b_{k+1},c_{k+1}],\cdots,[b_1,c_1]$。这里，每个区间的左端点都是其右端点处对应问题的最小最优决策；同样地，从右向左考虑可能的分划，应该有右端点也是左端点对应问题的最小最优决策。例如，d_j 和 c_j 分别是将 $[a_j,i]$ 和 $[b_j,i]$ 分成 j 段左起第一个区间右端点的最小最优决策。根据决策单调性，如果 $a_{j-1} > b_{j-1}$，亦即 $d_j > c_j$，那么必然有 $a_j > c_j$。由此，如果所证不成立，则有 $a_1 > b_1$。进而可以归纳地证明 $a_k > b_k$。这显然与所设矛盾。由此得证。

第一个不等式可以另证如下。同样考虑上面证明中的两个分划。如果所证命题不成立，则有 $a_1 > b_1$，但是由于有 $a_k < b_k$，我们可以找到最小的 $j > 1$ 使得 $a_j \le b_j$。进而，此时有 $a_{j-1} > b_{j-1}$，故 $d_j > c_j$。我们找到了一组区间满足 $a_j \le b_j \le c_j < d_j$。考虑将这两个分拆重新组合的结果。考虑分拆 $[b_{k+1},c_{k+1}],\cdots,[b_{j+1},c_{j+1}],[b_j,d_j],[a_{j-1},d_{j-1}],\cdots,[a_1,d_1]$，共 $(k+1)$ 段，于是由前设的最优性可推知，

$$\begin{aligned} &w(b_{k+1},c_{k+1}) + \cdots + w(b_{j+1},c_{j+1}) + w(b_j,c_j) + w(b_{j-1},c_{j-1}) + \cdots + w(b_1,c_1) \\ &\quad \le w(b_{k+1},c_{k+1}) + \cdots + w(b_{j+1},c_{j+1}) + w(b_j,d_j) + w(a_{j-1},d_{j-1}) + \cdots + w(a_1,d_1). \end{aligned}$$

同样地，考虑分拆 $[a_k,d_k],\cdots,[a_{j+1},d_{j+1}],[a_j,c_j],[b_{j-1},c_{j-1}],\cdots,[b_1,c_1]$，共 k 段，则有

$$\begin{aligned} &w(a_k,d_k) + \cdots + w(a_{j+1},d_{j+1}) + w(a_j,d_j) + w(a_{j-1},d_{j-1}) + \cdots + w(a_1,d_1) \\ &\quad < w(a_k,d_k) + \cdots + w(a_{j+1},d_{j+1}) + w(a_j,c_j) + w(b_{j-1},c_{j-1}) + \cdots + w(b_1,c_1). \end{aligned}$$

此时，不等号是严格的，因为 $a_1 > b_1$，但是按假设，a_1 是所有 k 段分拆最末一段的左端点中最小最优的。两个不等式条件相加，得到 $w(b_j,c_j) + w(a_j,d_j) < w(b_j,d_j) + w(a_j,c_j)$，这有悖于四边形不等式。故而原结论得证。

利用这一结果，我们可以限制决策 j 的搜索范围。算法实现时，对 k 正向遍历，对 i 逆向遍历，在之前已确定的上下界范围内暴力搜索 j 就可以保证 $O(n(n+m))$ 的算法复杂度。

" 注意"

这里算法复杂度不是 $O(nm)$ 的。正确的复杂度计算需要考虑 $n \times m$ 维状态矩阵。因为对于问题 (i,k) 只需要考虑 $\mathrm{opt}(k-1,i) \le j \le \mathrm{opt}(k,i+1)$ 中的决策，所以每条次对角线上（即 $i-k$ 为一定值）的问题所需遍历的决策总数为 $O(n)$ 的。这样的对角线共计 $(n+m)$ 条，故而总的时间复杂度为 $O(n(n+m))$。

最后一种优化方法来源于如下的观察。

" 定理 3"

若 w 满足四边形不等式，则问题 (2) 的最优解 $g(k) := f(n,k)$ 是关于 k 的凸函数。

" 证明"

下证 $g(k-1)+g(k+1) \ge 2g(k)$。为此，考虑长度为 $(k-1)$ 段和 $(k+1)$ 段的最优分划，分别是 $[a_1,d_1],\cdots,[a_{k-1},d_{k-1}]$ 和 $[b_1,c_1],\cdots,[b_{k+1},c_{k+1}]$。取最小的 $1 \le j \le k-1$ 使得 $c_{j+1} \le d_j$，其存在性可由 $c_k < n = d_{k-1}$ 推知。根据其最小性得知，$b_{j+1} > a_j$。所以，$a_j < b_{j+1} \le c_{j+1} \le d_j$。与上文类似，交换两个现有分拆的后半段，可以得到如下两个区间分拆：

$$
\begin{aligned}
&[a_1,d_1],\cdots,[a_{j-1},d_{j-1}],[a_j,c_{j+1}],[b_{j+2},c_{j+2}],\cdots,[b_{k+1},c_{k+1}],\\
&[b_1,c_1],\cdots,[b_j,c_j],[b_{j+1},d_j],[a_{j+1},d_{j+1}],\cdots,[a_{k-1},d_{k-1}].
\end{aligned}
$$

两个所得区间都是 k 段的，所以由最优性条件可知

$$
\begin{aligned}
2g(k) &\le w(a_1,d_1)+\cdots+w(a_{j-1},d_{j-1})+w(a_j,c_{j+1})+w(b_{j+2},c_{j+2})+\cdots+w(b_{k+1},c_{k+1})\\
&\quad +w(b_1,c_1)+\cdots+w(b_j,c_j)+w(b_{j+1},d_j)+w(a_{j+1},d_{j+1})+\cdots+w(a_{k-1},d_{k-1})\\
&\le w(a_1,d_1)+\cdots+w(a_{j-1},d_{j-1})+w(a_j,d_j)+w(a_{j+1},d_{j+1})+\cdots+w(a_{k-1},d_{k-1})\\
&\quad +w(b_1,c_1)+\cdots+w(b_j,c_j)+w(b_{j+1},c_{j+1})+w(b_{j+2},c_{j+2})+\cdots+w(b_{k+1},c_{k+1})\\
&= g(k-1)+g(k+1).
\end{aligned}
$$

这里第二个不等式正是四边形不等式。所求凸性由此得证。

这一结论保证了可以通过 wqs 二分（国外称 Alien's trick）的方法解决此问题。具体来说，考虑带参的成本函数 $w_c(j,i) := w(j,i)+c$，解决不限制区间个数的问题，求得其最优解为 $f_c(n)$。随着实数 c 递增，相应的最优区间的数目单调递减，故而可以通过二分的方法找到恰使得最优区间个数等于 m 的参数 c，则原题最优解为 $f(n,m) = f_c(n) - cm$。这里的实数 c 可以看作区间个数限制的 Lagrange 乘子。该算法的实现有很多细节，可以参考 专门介绍 wqs 二分的文章[3]。这一算法的时间复杂度为 $O(n \log n \log C)$，这里 C 为某一常数。

对于限制区间个数的区间分拆问题的三种算法，在不同的数据范围时表现各有优劣，需要结合具体的题目选择合适的算法。

区间合并问题

另一类可以通过四边形不等式优化的动态规划问题是区间合并问题，即要将 n 个长度为一的区间 $[i,i]$ 两两合并起来，直到得到区间 $[1,n]$。每次合并 $[j,k]$ 和 $[k+1,i]$ 时都需要支付成本 $w(j,i)$。问题要求找到成本最低的合并方式。对于此类问题，有如下 2D1D 状态转移方程。

$$
f(j,i) = \min_{j \le k < i} f(j,k) + f(k+1,i) + w(j,i) \qquad (1 \le j < i \le n) \tag{3}
$$

这里给定任意初始成本 $f(i,i) = w(i,i)$。暴力算法的总复杂度为 $O(n^3)$，而当存在决策单调性时，可以优化至 $O(n^2)$ 的算法复杂度。这一算法最早由 Knuth 在解决最优二叉搜索树问题时提出，并由姚储枫进一步研究总结，在国外称为 Knuth's optimization 或 Knuth-Yao speedup。

除了四边形不等式以外，区间合并问题的决策单调性还要求成本函数满足区间包含单调性。

- **区间包含单调性**：如果对于任意 $a \le b \le c \le d$ 均成立

$$w(b,c) \le w(a,d),$$

则称函数 w 对于区间包含关系具有单调性。

这实质是成本函数的一阶条件，即 $w(j,i)$ 关于 j 递减，关于 i 递增。

" 引理 1"

若 w 满足区间包含单调性和四边形不等式，则状态 $f(j,i)$ 满足四边形不等式。

" 证明"

不妨设 $a \le b \le c \le d$。下证 $f(a,d)+f(b,c) \ge f(a,c)+f(b,d)$。考虑依 $d-a$ 归纳。当 $a=b$ 或 $c=d$ 时，所求即一等式。对于一般的情形，根据 $d'=\operatorname{opt}(a,d)$ 的位置分类讨论。

第一种情况，$c \le d'$ 或 $d' < b$，即 $[b,c]$ 包含于 $[a,d']$ 或 $[d'+1,d]$ 之中。

不妨假设 $c \le d'$，另一种情形同理。此时有

$$\begin{aligned}
f(a,d)+f(b,c) &= f(a,d')+f(d'+1,d)+w(a,d)+f(b,c)\\
&\ge f(a,c)+f(b,d')+f(d'+1,d)+w(a,d)\\
&\ge f(a,c)+f(b,d')+f(d'+1,d)+w(b,d)\\
&\ge f(a,c)+f(b,d).
\end{aligned}$$

这里，第一个不等式来自于归纳假设 $f(a,c)+f(b,d') \le f(a,d')+f(b,c)$，第二个不等式来自于区间包含单调性 $w(b,d) \le w(a,d)$，第三个不等式来自于最优性条件 $f(b,d) \le f(b,d')+f(d'+1,d)+w(b,d)$。

第二种情况，$b \le d' < c$，即 d' 位于 $[b,c]$ 之中。此时，考虑 $c'=\operatorname{opt}(b,c)$ 的位置。

不妨假设 $c' \le d'$，即 $[b,c']$ 包含于 $[a,d']$ 之中，另一种情形同理。此时有

$$\begin{aligned}
f(a,d)+f(b,c) &= f(a,d')+f(d'+1,d)+w(a,d)+f(b,c')+f(c'+1,c)+w(b,c)\\
&\ge f(a,c')+f(c'+1,c)+w(b,c)+f(b,d')+f(d'+1,d)+w(a,d)\\
&\ge f(a,c')+f(c'+1,c)+w(a,c)+f(b,d')+f(d'+1,d)+w(b,d)\\
&\ge f(a,c)+f(b,d).
\end{aligned}$$

这里，第一个不等式来自于归纳假设 $f(a,c')+f(b,d') \le f(a,d')+f(b,c')$，第二个不等式来自于四边形不等式 $w(a,c)+w(b,d) \le w(a,d)+w(b,c)$，第三个不等式来自于 $f(a,c)$ 和 $f(b,d)$ 的最优性条件。

" 定理 4"

若 w 满足区间包含单调性和四边形不等式，则问题 (3) 中最小最优决策 $\operatorname{opt}(j,i)$ 满足

$$\operatorname{opt}(j,i-1) \le \operatorname{opt}(j,i) \le \operatorname{opt}(j+1,i). \qquad (j+1<i)$$

" 证明"

引理 1 已经证得 $f(j,i)$ 满足四边形不等式，所以目标函数 $f(j,k)+f(k+1,i)+w(j,i)$ 对于给定 j 作为 (k,i) 的函数满足四边形不等式，所以由定理 1 有，$\operatorname{opt}(j,i-1) \le \operatorname{opt}(j,i)$。注意，不同时含有 (k,i) 的项并不影响四边形不等式成立。类似地，它对于给定 i 作为 (k,j) 的函数也满足四边形不等式，所以 $\operatorname{opt}(j,i) \le \operatorname{opt}(j+1,i)$。即得所证。

利用这一结论，同样可以限制决策点 k 的搜索范围。在这里，正序遍历区间长度 $i-j+1$，再遍历具有同样长度的所有区间 $[j,i]$，暴力搜索 $\operatorname{opt}(j,i-1)$ 和 $\operatorname{opt}(j+1,i)$ 之间的所有 k 求得最优解 $f(j,i)$ 并记录最小最优决策 $\operatorname{opt}(j,i)$。对于同样长度的所有区间，此算法中决策空间总长度是 $O(n)$ 的，而可能的区间长度的数目同样是 $O(n)$ 的，故而总的算法复杂度为 $O(n^2)$ 的。

" 核心代码"

```cpp
for (int len = 2; len <= n; ++len)  // 枚举区间长度
  for (int j = 1, i = len; i <= n; ++j, ++i) {
    // 枚举长度为 len 的所有区间
    f[j][i] = INF;
    for (int k = opt[j][i - 1]; k <= opt[j + 1][i]; ++k)
      if (f[j][i] > f[j][k] + f[k + 1][i] + w(j, i)) {
        f[j][i] = f[j][k] + f[k + 1][i] + w(j, i);  // 更新状态值
        opt[j][i] = k;  // 更新（最小）最优决策点
      }
  }
```

```python
for len in range(2, n + 1): # 枚举区间长度
    for i in range(len, n + 1):
        # 枚举长度为 len 的所有区间
        j = i - len + 1
        f[j][i] = INF
        for k in range(opt[j][i - 1], opt[j + 1][i] + 1):
            if f[j][i] > f[j][k] + f[k + 1][i] + w(j, i):
                f[j][i] = f[j][k] + f[k + 1][i] + w(j, i) # 更新状态值
                opt[j][i] = k # 更新（最小）最优决策点
```

满足四边形不等式的函数类

为了更方便地证明一个函数满足四边形不等式，我们有以下几条性质：

性质 1：若函数 $w_1(j,i)$ 和 $w_2(j,i)$ 均满足四边形不等式（或区间包含单调性），则对于任意 $c_1, c_2 \ge 0$，函数 $c_1 w_1 + c_2 w_2$ 也满足四边形不等式（或区间包含单调性）。

性质 2：若存在函数 $f(x)$ 和 $g(x)$ 使得 $w(j,i) = f(j) - g(i)$，则函数 w 满足四边形恒等式。当函数 f 和 g 单调增加时，函数 w 还满足区间包含单调性。

性质 3：设 $h(x)$ 是一个单调增加的凸函数，若函数 $w(j,i)$ 满足四边形不等式并且对区间包含关系具有单调性，则复合函数 $h(w(j,i))$ 也满足四边形不等式和区间包含单调性。

性质 4：设 $h(x)$ 是一个凸函数，若函数 $w(j,i)$ 满足四边形恒等式并且对区间包含关系具有单调性，则复合函数 $h(w(j,i))$ 也满足四边形不等式。

首先需要澄清一点，凸函数（Convex Function）的定义在国内教材中有分歧，此处的凸函数指的是下凸函数，即（可微时）一阶导数单调增加的函数。

" 证明"

前两条性质根据定义很容易证明，下面证明第三条性质，性质四的证明过程类似。由于 $h(x)$ 单调，$h(w(j,i))$ 自然保持对区间包含的单调性。关键在于四边形不等式的证明。

为此，下面考虑 $a \le j \le b \le c \le i \le d$ 上的二阶混合差分。

$$\begin{aligned}\Delta_i \Delta_j h(w(j,i)) = {} & h(w(b,d)) - h\left(w(a,c) + \Delta_j w(j,c) + \Delta_i w(a,i)\right) \\ & + h\left(w(a,c) + \Delta_j w(j,c) + \Delta_i w(a,i)\right) - h\left(w(a,c) + \Delta_j w(j,c)\right) \\ & - h\left(w(a,c) + \Delta_i w(a,i)\right) + h(w(a,c)).\end{aligned}$$

这里，根据区间单调性，$\Delta_i w(a,i) := w(a,d) - w(a,c) \ge 0$ 和 $\Delta_j w(j,c) := w(b,c) - w(a,c) \le 0$。由于 $h(x)$ 具有凸性，对于 $t_1, t_2 \ge 0$ 成立 $h(x + t_1 - t_2) - h(x + t_1) \le h(x - t_2) - h(x)$，所以后两行必然非正。同时，由于四边形不等式，$w(b,d) \le w(a,c) + \Delta_j w(j,c) + \Delta_i w(a,i) = w(b,c) + w(a,d) - w(a,c)$，故而，第一行的差在 $h(x)$ 单调增加的情况下必然也非正。所以，总的二阶混合差分非正。此即四边形不等式。

这一证明实际是如下导数证明的离散版本。

$$\frac{\partial^2}{\partial x \partial y} h(w(x,y)) = h''(w(x,y)) \frac{\partial}{\partial x} w(x,y) \frac{\partial}{\partial y} w(x,y) + h'(w(x,y)) \frac{\partial^2}{\partial x \partial y} w(x,y) \le 0.$$

这在 $h' \ge 0$，$h'' \ge 0$，$w_x \le 0$，$w_y \ge 0$ 以及 $w_{xy} \le 0$ 的条件下显然成立。其中，区间包含单调性给出了 w 的一阶条件，而四边形不等式给出了其二阶条件。

习题

- 「IOI2000」 邮局[4]
- Codeforces - Ciel and Gondolas[5] (Be careful with I/O!)
- SPOJ - LARMY[6]
- Codechef - CHEFAOR[7]
- Hackerrank - Guardians of the Lunatics[8]
- ACM ICPC World Finals 2017 - Money[9]

参考资料

- noiau 的 CSDN 博客[10]
- Quora Answer by Michael Levin[11]
- Video Tutorial by "Sothe" the Algorithm Wolf[12]
- Divide and Conquer DP[13]
- Knuth's Optimization[14]
- Quadrangle Inequality Properties[15]
- 王钦石 《浅析一类二分方法》[16]

参考资料与注释

[1] 「POI2011」Lightning Conductor

[2] 「HNOI2008」玩具装箱 toy

[3] 专门介绍 wqs 二分的文章

[4] 「IOI2000」邮局

[5] Codeforces - Ciel and Gondolas

[6] SPOJ - LARMY

[7] Codechef - CHEFAOR

[8] Hackerrank - Guardians of the Lunatics

[9] ACM ICPC World Finals 2017 - Money

[10] noiau 的 CSDN 博客

[11] Quora Answer by Michael Levin

[12] Video Tutorial by "Sothe" the Algorithm Wolf

[13] Divide and Conquer DP

[14] Knuth's Optimization

[15] Quadrangle Inequality Properties

[16] 王钦石《浅析一类二分方法》

7.14.4 状态设计优化

Authors: Marcythm, partychicken, Xeonacid

概述

优化 dp 时，不止可以从转移过程入手，加速转移。有时，也可以从状态定义入手，通过改变设计状态的方式实现复杂度上的优化。

令人比较头疼的是，这类优化大多不具有通用性，即不能很套路地应用于多个题目中。因此，下文将从具体例题出发，力求提供思路上的启发，希望可以对读者有一定帮助。

例 1

"题面"

给定两个长度分别为 n,m 且仅由小写字母构成的字符串 A,B，求 A,B 的最长公共子序列。($n \le 10^6, m \le 10^3$)

朴素的解法

您一眼秒了它，这不是板子吗？

定义状态 $f_{i,j}$ 为 A 的前 i 位与 B 的前 j 位最长公共子序列，则有

$$f_{i,j} = \begin{cases} \max(f_{i-1,j}, f_{i,j-1}) & ,A_i \ne B_j \\ f_{i-1,j-1}+1 & ,A_i = B_j \end{cases}$$

上述做法的时间复杂度 $O(nm)$，无法通过本题。

更优的解法

我们仔细一想，发现了一个性质：最终答案不会超过 m。

我们又仔细一想，发现 LCS 满足贪心的性质。

更改状态定义 $f_{i,j}$ 为与 B 前 i 位的最长公共子序列长度为 j 的 A 的最短前缀长度（即将朴素做法的答案与第一维状态对调）

可以通过预处理 A 的每一位的下一个 $a, b, \cdots, z$ 的出现位置进行 $O(1)$ 的顺推转移。

复杂度 $O(m^2 + 26n)$，可以通过本题。

例 2

" 题面"

给定一个 n 个点的无权有向图，判断该图是否存在哈密顿回路。$(2 \le n \le 20)$

朴素的解法

看到数据范围，我们考虑状压。

设 $f_{s,i}$ 表示从点 1 出发，仅经过点集 s 中的点能否到达点 i。记 g 为原图的邻接矩阵。则有

$$f_{s,i} = \bigvee_{j \in s, j \neq i} f_{s\ \{i\},j} \wedge g_{j,i}\,(i \in s)$$

时间复杂度 $O(n^2 \times 2^n)$，写得好看或许能过，但是并不优美。

更优的解法

上面的状态设计中，每个 dp 值只代表一个 `bool` 值，这让我们觉得有些浪费。

我们可以考虑对于每个状态 s 将 $f_{s,1}, f_{s,2}, \cdots, f_{s,n}$ 压成一个 `int`，发现我们可以将邻接矩阵同样压缩后进行 $O(1)$ 转移。

时间复杂度 $O(n^2/w \times 2^n)$，可以通过这道题，其中 w 为 `int` 的位数。

7.15 其它 DP 方法

第 8 章

字符串

8.1 字符串部分简介

字符串，就是由字符连接而成的序列。

常见的字符串问题包括字符串匹配问题、子串相关问题、前缀/后缀相关问题、回文串相关问题、子序列相关问题等。

8.2 字符串基础

Authors: Ir1d, ouuan, qinggniq, i-Yirannn, minghu6

定义

字符集

一个**字符集** Σ 是一个建立了 全序 关系的集合，也就是说，Σ 中的任意两个不同的元素 α 和 β 都可以比较大小，要么 $\alpha<\beta$，要么 $\beta<\alpha$。字符集 Σ 中的元素称为字符。

字符串

一个**字符串** S 是将 n 个字符顺次排列形成的序列，n 称为 S 的长度，表示为 $|S|$。

如果字符串下标从 1 开始计算，S 的第 i 个字符表示为 $S[i]$；

如果字符串下标从 0 开始计算，S 的第 i 个字符表示为 $S[i-1]$。

子串

字符串 S 的**子串** $S[i..j]$ $i \le j$，表示 S 串中从 i 到 j 这一段，也就是顺次排列 $S[i], S[i+1], \ldots, S[j]$ 形成的字符串。

有时也会用 $S[i..j]$，$i>j$ 来表示空串。

子序列

字符串 S 的**子序列**是从 S 中将若干元素提取出来并不改变相对位置形成的序列，即 $S[p_1], S[p_2], \ldots, S[p_k]$，$1 \le p_1 < p_2 < \cdots < p_k \le |S|$。

后缀

后缀是指从某个位置 i 开始到整个串末尾结束的一个特殊子串。字符串 S 的从 i 开头的后缀表示为 $\textit{Suffix(S,i)}$，也就是 $\textit{Suffix(S,i)} = S[i..|S|-1]$。

真后缀指除了 S 本身的 S 的后缀。

举例来说，字符串 `abcabcd` 的所有后缀为 `{d，cd，bcd，abcd，cabcd，bcabcd，abcabcd}`，而它的真后缀为 `{d，cd，bcd，abcd，cabcd，bcabcd}`。

前缀

前缀是指从串首开始到某个位置 i 结束的一个特殊子串。字符串 S 的以 i 结尾的前缀表示为 $\textit{Prefix(S,i)}$，也就是 $\textit{Prefix(S,i)} = S[0..i]$。

真前缀指除了 S 本身的 S 的前缀。

举例来说，字符串 `abcabcd` 的所有前缀为 `{a，ab，abc，abca，abcab，abcabc，abcabcd}`，而它的真前缀为 `{a，ab，abc，abca，abcab，abcabc}`。

字典序

以第 i 个字符作为第 i 关键字进行大小比较，空字符小于字符集内任何字符（即：$a < aa$）。

回文串

回文串是正着写和倒着写相同的字符串，即满足 $\forall 1 \le i \le |s|, s[i] = s[|s|+1-i]$ 的 s。

字符串的存储

- 使用 `char` 数组存储，用空字符 `\0` 表示字符串的结尾（C 风格字符串）。
- 使用 C++ 标准库提供的 `string` 类。
- 字符串常量可以用字符串字面量（用双引号括起来的字符串）表示。

参考资料与注释

- 后缀数组 by. 徐智磊[1]

参考资料与注释

[1] 后缀数组 by. 徐智磊

8.3 标准库

Authors: Frankaiyou, henrytbtrue, zymooll

C 标准库

C 标准库操作字符数组 `char[]/const char*`。

参见：fprintf[1]、fscanf[2]、空终止字节字符串[3]

- `printf("%s", s)`：用 `%s` 来输出一个字符串（字符数组）。

- `scanf("%s", &s)`：用 `%s` 来读入一个字符串（字符数组）。
- `sscanf(const char *__source, const char *__format, ...)`：从字符串 `__source` 里读取变量，比如 `sscanf(str,"%d",&a)`。
- `sprintf(char *__stream, const char *__format, ...)`：将 `__format` 字符串里的内容输出到 `__stream` 中，比如 `sprintf(str,"%d",i)`。
- `strlen(const char *str)`：返回从 `str[0]` 开始直到 `'\0'` 的字符数。注意，未开启 O2 优化时，该操作写在循环条件中复杂度是 $\Theta(N)$ 的。
- `strcmp(const char *str1, const char *str2)`：按照字典序比较 `str1 str2` 若 `str1` 字典序小返回负值，两者一样返回 `0`，`str1` 字典序更大则返回正值。请注意，不要简单的认为返回值只有 `0`、`1`、`-1` 三种，在不同平台下的返回值都遵循正负，但并非都是 `0`、`1`、`-1`。
- `strcpy(char *str, const char *src)`: 把 `src` 中的字符复制到 `str` 中，`str src` 均为字符数组头指针，返回值为 `str` 包含空终止符号 `'\0'`。
- `strncpy(char *str, const char *src, int cnt)`：复制至多 `cnt` 个字符到 `str` 中，若 `src` 终止而数量未达 `cnt` 则写入空字符到 `str` 直至写入总共 `cnt` 个字符。
- `strcat(char *str1, const char *str2)`: 将 `str2` 接到 `str1` 的结尾，用 `*str2` 替换 `str1` 末尾的 `'\0'` 返回 `str1`。
- `strstr(char *str1, const char *str2)`：若 `str2` 是 `str1` 的子串，则返回 `str2` 在 `str1` 的首次出现的地址；如果 `str2` 不是 `str1` 的子串，则返回 `NULL`。
- `strchr(const char *str, int c)`：找到在字符串 `str` 中第一次出现字符 `c` 的位置，并返回这个位置的地址。如果未找到该字符则返回 `NULL`。
- `strrchr(const char *str, char c)`：找到在字符串 `str` 中最后一次出现字符 `c` 的位置，并返回这个位置的地址。如果未找到该字符则返回 `NULL`。

C++ 标准库

C++ 标准库操作字符串对象 `std::string`，同时也提供对字符数组的兼容。

参见：std::basic_string[4]、std::basic_string_view[5]

- 重载了赋值运算符 `+`，当 `+` 两边是 `string/char/char[]/const char*` 类型时，可以将这两个变量连接，返回连接后的字符串（`string`）。
- 赋值运算符 `=` 右侧可以是 `const string/string/const char*/char*`。
- 访问运算符 `[cur]` 返回 `cur` 位置的引用。
- 访问函数 `data()/c_str()` 返回一个 `const char*` 指针，内容与该 `string` 相同。
- 容量函数 `size()` 返回字符串字符个数。
- `find(ch, start = 0)` 查找并返回从 `start` 开始的字符 `ch` 的位置；`rfind(ch)` 从末尾开始，查找并返回第一个找到的字符 `ch` 的位置（皆从 `0` 开始）（如果查找不到，返回 `-1`）。
- `substr(start, len)` 可以从字符串的 `start`（从 `0` 开始）截取一个长度为 `len` 的字符串（缺省 `len` 时代码截取到字符串末尾）。
- `append(s)` 将 `s` 添加到字符串末尾。
- `append(s, pos, n)` 将字符串 `s` 中，从 `pos` 开始的 `n` 个字符连接到当前字符串结尾。
- `replace(pos, n, s)` 删除从 `pos` 开始的 `n` 个字符，然后在 `pos` 处插入串 `s`。
- `erase(pos, n)` 删除从 `pos` 开始的 `n` 个字符。
- `insert(pos, s)` 在 `pos` 位置插入字符串 `s`。
- `std::string` 重载了比较逻辑运算符，复杂度是 $\Theta(N)$ 的。

参考资料与注释

[1] fprintf

[2] fscanf

[3] 空终止字节字符串

[4] std::basic_string

[5] std::basic_string_view

8.4 字符串匹配

本页面将简述字符串匹配问题以及它的解法。

字符串匹配问题

定义

又称模式匹配（pattern matching）。该问题可以概括为「给定字符串 S 和 T，在主串 S 中寻找子串 T」。字符 T 称为模式串 (pattern)。

类型

- 单串匹配：给定一个模式串和一个待匹配串，找出前者在后者中的所有位置。
- 多串匹配：给定多个模式串和一个待匹配串，找出这些模式串在后者中的所有位置。
  - 出现多个待匹配串时，将它们直接连起来便可作为一个待匹配串处理。
  - 可以直接当做单串匹配，但是效率不够高。
- 其他类型：例如匹配一个串的任意后缀，匹配多个串的任意后缀……

暴力做法

简称 BF (Brute Force) 算法。该算法的基本思想是从主串 S 的第一个字符开始和模式串 T 的第一个字符进行比较，若相等，则继续比较二者的后续字符；否则，模式串 T 回退到第一个字符，重新和主串 S 的第二个字符进行比较。如此往复，直到 S 或 T 中所有字符比较完毕。

实现

```
/*
 * s: 待匹配的主串
 * t: 模式串
 * n: 主串的长度
 * m: 模式串的长度
 */
std::vector<int> match(char *s, char *t, int n, int m) {
  std::vector<int> ans;
```

```
  int i, j;
  for (i = 0; i < n - m + 1; i++) {
    for (j = 0; j < m; j++) {
      if (s[i + j] != t[j]) break;
    }
    if (j == m) ans.push_back(i);
  }
  return ans;
}
```

```python
def match(s, t, n, m):
    if m < 1:
        return []

    ans = []
    for i in range(0, n - m + 1):
        for j in range(0, m):
            if s[i + j] != t[j]:
                break
        else:
            ans.append(i)
    return ans
```

时间复杂度

设 n 为主串的长度，m 为模式串的长度。默认 $m \ll n$。

BF 算法匹配成功时，在最好情况下，只有一趟匹配成功，此趟比较次数为 m，而其余每趟不成功的匹配都发生在模式串的第一个字符，还需要 $n-m$ 次比较，总比较次数为 n，故时间复杂度为 $O(n)$；在最坏情况下，匹配成功的趟数为 $n-m+1$，每趟比较次数为 m，总比较次数为 $m(n-m+1)$，故时间复杂度为 $O(mn)$。

BF 算法匹配失败时，在最好情况下，每趟不成功的匹配都发生在模式串的第一个字符，BF 算法要执行 $n-m+1$ 次比较，时间复杂度为 $O(n)$；在最坏情况下，每趟不成功的匹配都发生在模式串的最后一个字符，BF 算法要执行 $m(n-m+1)$ 次比较，时间复杂度为 $O(mn)$。

如果模式串有至少两个不同的字符，则 BF 算法的平均时间复杂度为 $O(n)$。但是在 OI 题目中，给出的字符串一般都不是纯随机的。

Hash 的方法

参见：字符串哈希

KMP 算法

参见：前缀函数与 KMP 算法

8.5　字符串哈希

定义

我们定义一个把字符串映射到整数的函数 f，这个 f 称为是 Hash 函数。

我们希望这个函数 f 可以方便地帮我们判断两个字符串是否相等。

Hash 的思想

Hash 的核心思想在于，将输入映射到一个值域较小、可以方便比较的范围。

warning

这里的「值域较小」在不同情况下意义不同。

在 哈希表 中，值域需要小到能够接受线性的空间与时间复杂度。

在字符串哈希中，值域需要小到能够快速比较（10^9、10^{18} 都是可以快速比较的）。

同时，为了降低哈希冲突率，值域也不能太小。

性质

具体来说，哈希函数最重要的性质可以概括为下面两条：

1. 在 Hash 函数值不一样的时候，两个字符串一定不一样；
2. 在 Hash 函数值一样的时候，两个字符串不一定一样（但有大概率一样，且我们当然希望它们总是一样的）。

   我们将 Hash 函数值一样但原字符串不一样的现象称为哈希碰撞。

解释

我们需要关注的是什么？

时间复杂度和 Hash 的准确率。

通常我们采用的是多项式 Hash 的方法，对于一个长度为 l 的字符串 s 来说，我们可以这样定义多项式 Hash 函数：$f(s)=\sum_{i=1}^{l} s[i]\times b^{l-i} \pmod M$。例如，对于字符串 xyz，其哈希函数值为 xb^2+yb+z。

特别要说明的是，也有很多人使用的是另一种 Hash 函数的定义，即 $f(s)=\sum_{i=1}^{l} s[i]\times b^{i-1} \pmod M$，这种定义下，同样的字符串 xyz 的哈希值就变为了 $x+yb+zb^2$ 了。

显然，上面这两种哈希函数的定义函数都是可行的，但二者在之后会讲到的计算子串哈希值时所用的计算式是不同的，因此千万注意**不要弄混了这两种不同的 Hash 方式**。

由于前者的 Hash 定义计算更简便、使用人数更多、且可以类比为一个 b 进制数来帮助理解，所以本文下面所将要讨论的都是使用 $f(s)=\sum_{i=1}^{l} s[i]\times b^{l-i} \pmod M$ 来定义的 Hash 函数。

下面讲一下如何选择 M 和计算哈希碰撞的概率。

这里 M 需要选择一个素数（至少要比最大的字符要大），b 可以任意选择。

如果我们用未知数 x 替代 b，那么 $f(s)$ 实际上是多项式环 $\mathbb{Z}_M[x]$ 上的一个多项式。考虑两个不同的字符串 s,t，有 $f(s)=f(t)$。我们记 $h(x)=f(s)-f(t)=\sum_{i=1}^{l}(s[i]-t[i])x^{l-i} \pmod M$，其中 $l=\max(|s|,|t|)$。可以发现 $h(x)$ 是一个 $l-1$ 阶的非零多项式。

如果 s 与 t 在 $x=b$ 的情况下哈希碰撞，则 b 是 $h(x)$ 的一个根。由于 $h(x)$ 在 $\mathbb{Z}_M$ 是一个域（等价于 M 是一个素数，这也是为什么 M 要选择素数的原因）的时候，最多有 $l-1$ 个根，如果我们保证 b 是从 $[0,M)$ 之间均匀随机选取的，那么 $f(s)$ 与 $f(t)$ 碰撞的概率可以估计为 $\frac{l-1}{M}$。简单验算一下，可以发现如果两个字符串长度都是 1 的时候，哈希碰撞的概率为 $\frac{1-1}{M}=0$，此时不可能发生碰撞。

实现

参考代码：（效率低下的版本，实际使用时一般不会这么写）

```
using std::string;

const int M = 1e9 + 7;
```

```cpp
const int B = 233;

typedef long long ll;

int get_hash(const string& s) {
  int res = 0;
  for (int i = 0; i < s.size(); ++i) {
    res = ((ll)res * B + s[i]) % M;
  }
  return res;
}

bool cmp(const string& s, const string& t) {
  return get_hash(s) == get_hash(t);
}
```

```python
M = int(1e9 + 7)
B = 233

def get_hash(s):
    res = 0
    for char in s:
        res = (res * B + ord(char)) % M
    return res

def cmp(s, t):
    return get_hash(s) == get_hash(t)
```

Hash 的分析与改进

错误率

假定哈希函数将字符串随机地映射到大小为 M 的值域中，总共有 n 个不同的字符串，那么未出现碰撞的概率是 $\prod_{i=0}^{n-1} \frac{M-i}{M}$（第 i 次进行哈希时，有 $\frac{M-i}{M}$ 的概率不会发生碰撞）。在随机数据下，若 $M = 10^9 + 7$，$n = 10^6$，未出现碰撞的概率是极低的。

所以，进行字符串哈希时，经常会对两个大质数分别取模，这样的话哈希函数的值域就能扩大到两者之积，错误率就非常小了。

多次询问子串哈希

单次计算一个字符串的哈希值复杂度是 $O(n)$，其中 n 为串长，与暴力匹配没有区别，如果需要多次询问一个字符串的子串的哈希值，每次重新计算效率非常低下。

一般采取的方法是对整个字符串先预处理出每个前缀的哈希值，将哈希值看成一个 b 进制的数对 M 取模的结果，这样的话每次就能快速求出子串的哈希了：

令 $f_i(s)$ 表示 $f(s[1..i])$，即原串长度为 i 的前缀的哈希值，那么按照定义有 $f_i(s) = s[1] \cdot b^{i-1} + s[2] \cdot b^{i-2} + \cdots + s[i-1] \cdot b + s[i]$

现在，我们想要用类似前缀和的方式快速求出 $f(s[l..r])$，按照定义有字符串 $s[l..r]$ 的哈希值为 $f(s[l..r]) = s[l] \cdot b^{r-l} + s[l+1] \cdot b^{r-l-1} + \cdots + s[r-1] \cdot b + s[r]$

对比观察上述两个式子，我们发现 $f(s[l..r]) = f_r(s) - f_{l-1}(s) \times b^{r-l+1}$ 成立（可以手动代入验证一下），因此我们用这个式子就可以快速得到子串的哈希值。其中 b^{r-l+1} 可以 $O(n)$ 的预处理出来然后 $O(1)$ 的回答每次询问（当然也可以快速幂 $O(\log n)$ 的回答每次询问）。

Hash 的应用

字符串匹配

求出模式串的哈希值后，求出文本串每个长度为模式串长度的子串的哈希值，分别与模式串的哈希值比较即可。

允许 k 次失配的字符串匹配

问题：给定长为 n 的源串 s，以及长度为 m 的模式串 p，要求查找源串中有多少子串与模式串匹配。s' 与 s 匹配，当且仅当 s' 与 s 长度相同，且最多有 k 个位置字符不同。其中 $1 \le n, m \le 10^6$，$0 \le k \le 5$。

这道题无法使用 KMP 解决，但是可以通过哈希 + 二分来解决。

枚举所有可能匹配的子串，假设现在枚举的子串为 s'，通过哈希 + 二分可以快速找到 s' 与 p 第一个不同的位置。之后将 s' 与 p 在这个失配位置及之前的部分删除掉，继续查找下一个失配位置。这样的过程最多发生 k 次。

总的时间复杂度为 $O(m + kn\log_2 m)$。

最长回文子串

二分答案，判断是否可行时枚举回文中心（对称轴），哈希判断两侧是否相等。需要分别预处理正着和倒着的哈希值。时间复杂度 $O(n\log n)$。

这个问题可以使用 manacher 算法 在 $O(n)$ 的时间内解决。

通过哈希同样可以 $O(n)$ 解决这个问题，具体方法就是记 R_i 表示以 i 作为结尾的最长回文的长度，那么答案就是 $\max_{i=1}^n R_i$。考虑到 $R_i \le R_{i-1} + 2$，因此我们只需要暴力从 $R_{i-1} + 2$ 开始递减，直到找到第一个回文即可。记变量 z 表示当前枚举的 R_i，初始时为 0，则 z 在每次 i 增大的时候都会增大 2，之后每次暴力循环都会减少 1，故暴力循环最多发生 $2n$ 次，总的时间复杂度为 $O(n)$。

最长公共子字符串

问题：给定 m 个总长不超过 n 的非空字符串，查找所有字符串的最长公共子字符串，如果有多个，任意输出其中一个。其中 $1 \le m, n \le 10^6$。

很显然如果存在长度为 k 的最长公共子字符串，那么 $k-1$ 的公共子字符串也必定存在。因此我们可以二分最长公共子字符串的长度。假设现在的长度为 k，`check(k)` 的逻辑为我们将所有所有字符串的长度为 k 的子串分别进行哈希，将哈希值放入 n 个哈希表中存储。之后求交集即可。

时间复杂度为 $O(m + n\log n)$。

确定字符串中不同子字符串的数量

问题：给定长为 n 的字符串，仅由小写英文字母组成，查找该字符串中不同子串的数量。

为了解决这个问题，我们遍历了所有长度为 $l = 1, \cdots, n$ 的子串。对于每个长度为 l，我们将其 Hash 值乘以相同的 b 的幂次方，并存入一个数组中。数组中不同元素的数量等于字符串中长度不同的子串的数量，并此数字将添加到最终答案中。

为了方便起见，我们将使用 $h[i]$ 作为 Hash 的前缀字符，并定义 $h[0] = 0$。

" 参考代码"

```
int count_unique_substrings(string const& s) {
  int n = s.size();

  const int b = 31;
  const int m = 1e9 + 9;
```

```
  vector<long long> b_pow(n);
  b_pow[0] = 1;
  for (int i = 1; i < n; i++) b_pow[i] = (b_pow[i - 1] * b) % m;

  vector<long long> h(n + 1, 0);
  for (int i = 0; i < n; i++)
    h[i + 1] = (h[i] + (s[i] - 'a' + 1) * b_pow[i]) % m;

  int cnt = 0;
  for (int l = 1; l <= n; l++) {
    set<long long> hs;
    for (int i = 0; i <= n - l; i++) {
      long long cur_h = (h[i + l] + m - h[i]) % m;
      cur_h = (cur_h * b_pow[n - i - 1]) % m;
      hs.insert(cur_h);
    }
    cnt += hs.size();
  }
  return cnt;
}
```

例题

"CF1200E Compress Words[1]"

给你若干个字符串，答案串初始为空。第 i 步将第 i 个字符串加到答案串的后面，但是尽量地去掉重复部分（即去掉一个最长的、是原答案串的后缀、也是第 i 个串的前缀的字符串），求最后得到的字符串。

字符串个数不超过 10^5，总长不超过 10^6。

" 题解"

每次需要求最长的、是原答案串的后缀、也是第 i 个串的前缀的字符串。枚举这个串的长度，哈希比较即可。

当然，这道题也可以使用 KMP 算法 解决。

" 参考代码"

```
#include <bits/stdc++.h>
using namespace std;

const int L = 1e6 + 5;
const int HASH_CNT = 2;

int hashBase[HASH_CNT] = {29, 31};
int hashMod[HASH_CNT] = {int(1e9 + 9), 998244353};

struct StringWithHash {
  char s[L];
  int ls;
  int hsh[HASH_CNT][L];
  int pwMod[HASH_CNT][L];

  void init() {  // 初始化
    ls = 0;
```

```cpp
    for (int i = 0; i < HASH_CNT; ++i) {
      hsh[i][0] = 0;
      pwMod[i][0] = 1;
    }
  }

  StringWithHash() { init(); }

  void extend(char c) {
    s[++ls] = c;                          // 记录字符数和每一个字符
    for (int i = 0; i < HASH_CNT; ++i) {  // 双哈希的预处理
      pwMod[i][ls] =
          1ll * pwMod[i][ls - 1] * hashBase[i] % hashMod[i];  // 得到 b^ls
      hsh[i][ls] = (1ll * hsh[i][ls - 1] * hashBase[i] + c) % hashMod[i];
    }
  }

  vector<int> getHash(int l, int r) {  // 得到哈希值
    vector<int> res(HASH_CNT, 0);
    for (int i = 0; i < HASH_CNT; ++i) {
      int t =
          (hsh[i][r] - 1ll * hsh[i][l - 1] * pwMod[i][r - l + 1]) % hashMod[i];
      t = (t + hashMod[i]) % hashMod[i];
      res[i] = t;
    }
    return res;
  }
};

bool equal(const vector<int> &h1, const vector<int> &h2) {
  assert(h1.size() == h2.size());
  for (unsigned i = 0; i < h1.size(); i++)
    if (h1[i] != h2[i]) return false;
  return true;
}

int n;
StringWithHash s, t;
char str[L];

void work() {
  int len = strlen(str);  // 取字符串长度
  t.init();
  for (int j = 0; j < len; ++j) t.extend(str[j]);
  int d = 0;
  for (int j = min(len, s.ls); j >= 1; --j) {
    if (equal(t.getHash(1, j), s.getHash(s.ls - j + 1, s.ls))) {  // 比较哈希值
      d = j;
      break;
    }
  }
  for (int j = d; j < len; ++j) s.extend(str[j]);  // 更新答案数组
}
```

```
int main() {
  scanf("%d", &n);
  for (int i = 1; i <= n; ++i) {
    scanf("%s", str);
    work();
  }
  printf("%s\n", s.s + 1);
  return 0;
}
```

本页面部分内容译自博文 [2] 与其英文翻译版 String Hashing[3]。其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0。

参考资料与注释

[1] CF1200E Compress Words

[2]

[3] String Hashing

8.6 字典树 (Trie)

定义

字典树，英文名 trie。顾名思义，就是一个像字典一样的树。

引入

先放一张图：

可以发现，这棵字典树用边来代表字母，而从根结点到树上某一结点的路径就代表了一个字符串。举个例子，$1 \to 4 \to 8 \to 12$ 表示的就是字符串 `caa`。

trie 的结构非常好懂，我们用 $\delta(u,c)$ 表示结点 u 的 c 字符指向的下一个结点，或着说是结点 u 代表的字符串后面添加一个字符 c 形成的字符串的结点。(c 的取值范围和字符集大小有关，不一定是 $0 \sim 26$。)

有时需要标记插入进 trie 的是哪些字符串，每次插入完成时在这个字符串所代表的节点处打上标记即可。

实现

放一个结构体封装的模板：

```
struct trie {
  int nex[100000][26], cnt;
  bool exist[100000];  // 该结点结尾的字符串是否存在

  void insert(char *s, int l) {  // 插入字符串
    int p = 0;
    for (int i = 0; i < l; i++) {
      int c = s[i] - 'a';
```

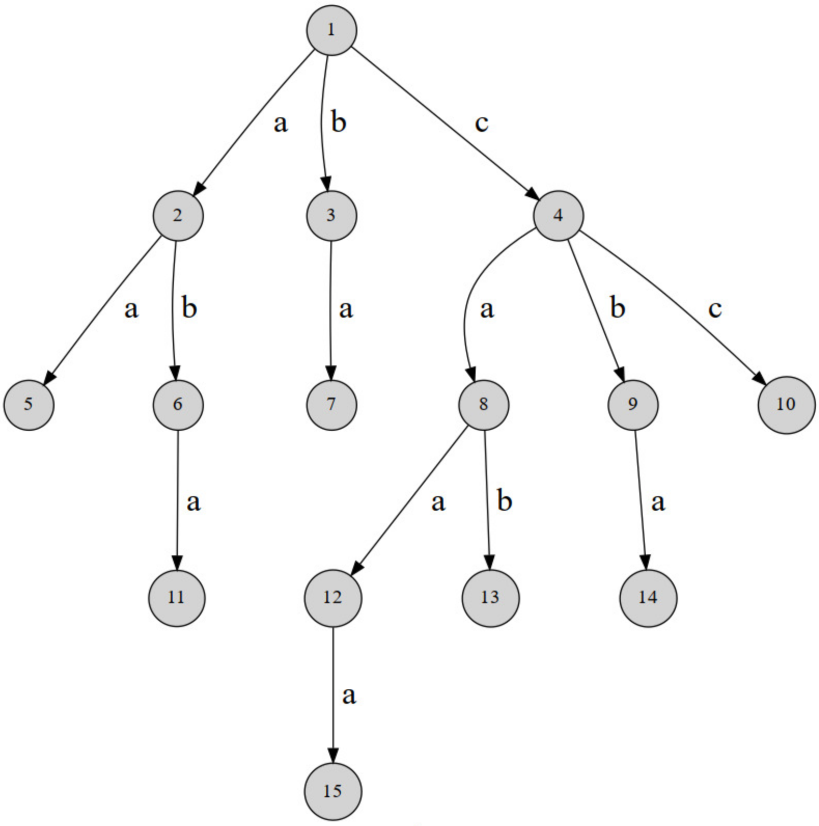


图 8.1 trie1

```
      if (!nex[p][c]) nex[p][c] = ++cnt;  // 如果没有，就添加结点
      p = nex[p][c];
    }
    exist[p] = 1;
  }

  bool find(char *s, int l) {  // 查找字符串
    int p = 0;
    for (int i = 0; i < l; i++) {
      int c = s[i] - 'a';
      if (!nex[p][c]) return 0;
      p = nex[p][c];
    }
    return exist[p];
  }
};
```

```python
class trie:
    def __init__(self):
        self.nex = [[0 for i in range(26)] for j in range(100000)]
        self.cnt = 0
        self.exist = [False] * 100000  # 该结点结尾的字符串是否存在

    def insert(self, s):  # 插入字符串
        p = 0
        for i in s:
```

```
            c = ord(i) - ord('a')
            if not self.nex[p][c]:
                self.cnt += 1
                self.nex[p][c] = self.cnt  # 如果没有，就添加结点
            p = self.nex[p][c]
        self.exist[p] = True

    def find(self, s):  # 查找字符串
        p = 0
        for i in s:
            c = ord(i) - ord('a')
            if not self.nex[p][c]:
                return False
            p = self.nex[p][c]
        return self.exist[p]
```

应用

检索字符串

字典树最基础的应用——查找一个字符串是否在「字典」中出现过。

"于是他错误的点名开始了[1]"

给你 n 个名字串，然后进行 m 次点名，每次你需要回答「名字不存在」、「第一次点到这个名字」、「已经点过这个名字」之一。

$1 \le n \le 10^4$，$1 \le m \le 10^5$，所有字符串长度不超过 50。

" 题解"

对所有名字建 trie，再在 trie 中查询字符串是否存在、是否已经点过名，第一次点名时标记为点过名。

" 参考代码"

```
#include <cstdio>

const int N = 500010;

char s[60];
int n, m, ch[N][26], tag[N], tot = 1;

int main() {
  scanf("%d", &n);

  for (int i = 1; i <= n; ++i) {
    scanf("%s", s + 1);
    int u = 1;
    for (int j = 1; s[j]; ++j) {
      int c = s[j] - 'a';
      if (!ch[u][c])
        ch[u][c] =
            ++tot;  // 如果这个节点的子节点中没有这个字符，添加上并将该字符的节点
号记录为++tot
      u = ch[u][c];  // 往更深一层搜索
```

```
    }
    tag[u] = 1;  // 最后一个字符为节点 u 的名字未被访问到记录为 1
  }

  scanf("%d", &m);

  while (m--) {
    scanf("%s", s + 1);
    int u = 1;
    for (int j = 1; s[j]; ++j) {
      int c = s[j] - 'a';
      u = ch[u][c];
      if (!u) break;  // 不存在对应字符的出边说明名字不存在
    }
    if (tag[u] == 1) {
      tag[u] = 2;  // 最后一个字符为节点 u 的名字已经被访问
      puts("OK");
    } else if (tag[u] == 2)  // 已经被访问，重复访问
      puts("REPEAT");
    else
      puts("WRONG");
  }

  return 0;
}
```

AC 自动机

trie 是 AC 自动机 的一部分。

维护异或极值

将数的二进制表示看做一个字符串，就可以建出字符集为 $\{0,1\}$ 的 trie 树。

"BZOJ1954 最长异或路径[2]"

给你一棵带边权的树，求 (u,v) 使得 u 到 v 的路径上的边权异或和最大，输出这个最大值。这里的异或和指的是所有边权的异或。

点数不超过 10^5，边权在 $[0,2^{31})$ 内。

" 题解"

随便指定一个根 $root$，用 $T(u,v)$ 表示 u 和 v 之间的路径的边权异或和，那么 $T(u,v)=T(root,u)\oplus T(root,v)$，因为 LCA 以上的部分异或两次抵消了。

那么，如果将所有 $T(root,u)$ 插入到一棵 trie 中，就可以对每个 $T(root,u)$ 快速求出和它异或和最大的 $T(root,v)$：

从 trie 的根开始，如果能向和 $T(root,u)$ 的当前位不同的子树走，就向那边走，否则没有选择。

贪心的正确性：如果这么走，这一位为 1；如果不这么走，这一位就会为 0。而高位是需要优先尽量大的。

" 参考代码"

```
#include <algorithm>
#include <cstdio>
```

```
using namespace std;

const int N = 100010;

int head[N], nxt[N << 1], to[N << 1], weight[N << 1], cnt;
int n, dis[N], ch[N << 5][2], tot = 1, ans;

void insert(int x) {
  for (int i = 30, u = 1; i >= 0; --i) {
    int c = ((x >> i) & 1);  // 二进制一位一位向下取
    if (!ch[u][c]) ch[u][c] = ++tot;
    u = ch[u][c];
  }
}

void get(int x) {
  int res = 0;
  for (int i = 30, u = 1; i >= 0; --i) {
    int c = ((x >> i) & 1);
    if (ch[u][c ^ 1]) {  // 如果能向和当前位不同的子树走，就向那边走
      u = ch[u][c ^ 1];
      res |= (1 << i);
    } else
      u = ch[u][c];
  }
  ans = max(ans, res);  // 更新答案
}

void add(int u, int v, int w) {  // 建边
  nxt[++cnt] = head[u];
  head[u] = cnt;
  to[cnt] = v;
  weight[cnt] = w;
}

void dfs(int u, int fa) {
  insert(dis[u]);
  get(dis[u]);
  for (int i = head[u]; i; i = nxt[i]) {  // 遍历子节点
    int v = to[i];
    if (v == fa) continue;
    dis[v] = dis[u] ^ weight[i];
    dfs(v, u);
  }
}

int main() {
  scanf("%d", &n);

  for (int i = 1; i < n; ++i) {
    int u, v, w;
    scanf("%d%d%d", &u, &v, &w);
    add(u, v, w);  // 双向边
    add(v, u, w);
```

```
  }

  dfs(1, 0);

  printf("%d", ans);

  return 0;
}
```

维护异或和

01-trie 是指字符集为 $\{0,1\}$ 的 trie。01-trie 可以用来维护一些数字的异或和，支持修改（删除 + 重新插入），和全局加一（即：让其所维护所有数值递增 `1`，本质上是一种特殊的修改操作）。

如果要维护异或和，需要按值从低位到高位建立 trie。

一个约定：文中说当前节点**往上**指当前节点到根这条路径，当前节点**往下**指当前结点的子树。

插入 & 删除

如果要维护异或和，我们**只需要**知道某一位上 `0` 和 `1` 个数的**奇偶性**即可，也就是对于数字 `1` 来说，当且仅当这一位上数字 `1` 的个数为奇数时，这一位上的数字才是 `1`，请时刻记住这段文字：如果只是维护异或和，我们只需要知道某一位上 `1` 的数量即可，而不需要知道 trie 到底维护了哪些数字。

对于每一个节点，我们需要记录以下三个量：

- `ch[o][0/1]` 指节点 `o` 的两个儿子，`ch[o][0]` 指下一位是 `0`，同理 `ch[o][1]` 指下一位是 `1`。
- `w[o]` 指节点 `o` 到其父亲节点这条边上数值的数量（权值）。每插入一个数字 `x`，`x` 二进制拆分后在 trie 上路径的权值都会 `+1`。
- `xorv[o]` 指以 `o` 为根的子树维护的异或和。

具体维护结点的代码如下所示。

```
void maintain(int o) {
  w[o] = xorv[o] = 0;
  if (ch[o][0]) {
    w[o] += w[ch[o][0]];
    xorv[o] ^= xorv[ch[o][0]] << 1;
  }
  if (ch[o][1]) {
    w[o] += w[ch[o][1]];
    xorv[o] ^= (xorv[ch[o][1]] << 1) | (w[ch[o][1]] & 1);
  }
  // w[o] = w[o] & 1;
  // 只需知道奇偶性即可，不需要具体的值。当然这句话删掉也可以，因为上文就只利用了他
的奇偶性。
}
```

插入和删除的代码非常相似。

需要注意的地方就是：

- 这里的 `MAXH` 指 trie 的深度，也就是强制让每一个叶子节点到根的距离为 `MAXH`。对于一些比较小的值，可能有时候不需要建立这么深（例如：如果插入数字 `4`，分解成二进制后为 `100`，从根开始插入 `001` 这三位即可），但是我们强制插入 `MAXH` 位。这样做的目的是为了便于全局 `+1` 时处理进位。例如：如果原数字是 `3`（`11`），递增之后变成 `4`（`100`），如果当初插入 `3` 时只插入了 `2` 位，那这里的进位就没了。
- 插入和删除，只需要修改叶子节点的 `w[]` 即可，在回溯的过程中一路维护即可。

" 实现"

```
namespace trie {
const int MAXH = 21;
int ch[_ * (MAXH + 1)][2], w[_ * (MAXH + 1)], xorv[_ * (MAXH + 1)];
int tot = 0;

int mknode() {
  ++tot;
  ch[tot][1] = ch[tot][0] = w[tot] = xorv[tot] = 0;
  return tot;
}

void maintain(int o) {
  w[o] = xorv[o] = 0;
  if (ch[o][0]) {
    w[o] += w[ch[o][0]];
    xorv[o] ^= xorv[ch[o][0]] << 1;
  }
  if (ch[o][1]) {
    w[o] += w[ch[o][1]];
    xorv[o] ^= (xorv[ch[o][1]] << 1) | (w[ch[o][1]] & 1);
  }
  w[o] = w[o] & 1;
}

void insert(int &o, int x, int dp) {
  if (!o) o = mknode();
  if (dp > MAXH) return (void)(w[o]++);
  insert(ch[o][x & 1], x >> 1, dp + 1);
  maintain(o);
}

void erase(int o, int x, int dp) {
  if (dp > 20) return (void)(w[o]--);
  erase(ch[o][x & 1], x >> 1, dp + 1);
  maintain(o);
}
}  // namespace trie
```

全局加一

所谓全局加一就是指，让这棵 trie 中所有的数值 `+1`。

形式化的讲，设 trie 中维护的数值有 $V_1, V_2, V_3 \dots V_n$，全局加一后其中维护的值应该变成 $V_1{+}1, V_2{+}1, V_3{+}1 \dots V_n{+}1$

```
void addall(int o) {
  swap(ch[o][0], ch[o][1]);
  if (ch[o][0]) addall(ch[o][0]);
  maintain(o);
}
```

过程

我们思考一下二进制意义下 `+1` 是如何操作的。

我们只需要从低位到高位开始找第一个出现的 `0`，把它变成 `1`，然后这个位置后面的 `1` 都变成 `0` 即可。

下面给出几个例子感受一下：（括号内的数字表示其对应的十进制数字）

```
1000(8)  + 1 = 1001(9)  ;
10011(19) + 1 = 10100(20) ;
11111(31) + 1 = 100000(32);
10101(21) + 1 = 10110(22) ;
100000000111111(16447) + 1 = 100000001000000(16448);
```

对应 trie 的操作，其实就是交换其左右儿子，顺着**交换后**的 `0` 边往下递归操作即可。

回顾一下 `w[o]` 的定义：`w[o]` 指节点 `o` 到其父亲节点这条边上数值的数量（权值）。

有没有感觉这个定义有点怪呢？如果在父亲结点存储到两个儿子的这条边的边权也许会更接近于习惯。但是在这里，在交换左右儿子的时候，在儿子结点存储到父亲这条边的距离，显然更加方便。

01-trie 合并

指的是将上述的两个 01-trie 进行合并，同时合并维护的信息。

可能关于合并 trie 的文章比较少，其实合并 trie 和合并线段树的思路非常相似，可以搜索「合并线段树」来学习如何合并 trie。

其实合并 trie 非常简单，就是考虑一下我们有一个 `int merge(int a, int b)` 函数，这个函数传入两个 trie 树位于同一相对位置的结点编号，然后合并完成后返回合并完成的结点编号。

过程

考虑怎么实现？

分三种情况：

- 如果 `a` 没有这个位置上的结点，新合并的结点就是 `b`
- 如果 `b` 没有这个位置上的结点，新合并的结点就是 `a`
- 如果 `a`,`b` 都存在，那就把 `b` 的信息合并到 `a` 上，新合并的结点就是 `a`，然后递归操作处理 a 的左右儿子。

  提示：如果需要的合并是将 a，b 合并到一棵新树上，这里可以新建结点，然后合并到这个新结点上，这里的代码实现仅仅是将 b 的信息合并到 a 上。

实现

```cpp
int merge(int a, int b) {
  if (!a) return b;  // 如果 a 没有这个位置上的结点，返回 b
  if (!b) return a;  // 如果 b 没有这个位置上的结点，返回 a
  /*
    如果 `a`, `b` 都存在，
    那就把 `b` 的信息合并到 `a` 上。
  */
  w[a] = w[a] + w[b];
  xorv[a] ^= xorv[b];
  /* 不要使用 maintain()，
    maintain() 是合并 a 的两个儿子的信息
    而这里需要 a b 两个节点进行信息合并
   */
  ch[a][0] = merge(ch[a][0], ch[b][0]);
  ch[a][1] = merge(ch[a][1], ch[b][1]);
  return a;
}
```

其实 trie 都可以合并，换句话说，trie 合并不仅仅限于 01-trie。

"【luogu-P6018】【Ynoi2010】Fusion tree[3]"

给你一棵 n 个结点的树，每个结点有权值。m 次操作。需要支持以下操作。

- 将树上与一个节点 x 距离为 1 的节点上的权值 $+1$。这里树上两点间的距离定义为从一点出发到另外一点的最短路径上边的条数。
- 在一个节点 x 上的权值 $-v$。
- 询问树上与一个节点 x 距离为 1 的所有节点上的权值的异或和。对于 100% 的数据，满足 $1 \le n \le 5 \times 10^5$，$1 \le m \le 5 \times 10^5$，$0 \le a_i \le 10^5$，$1 \le x \le n$，$opt \in \{1, 2, 3\}$。保证任意时刻每个节点的权值非负。

" 题解"

每个结点建立一棵 trie 维护其儿子的权值，trie 应该支持全局加一。可以使用在每一个结点上设置懒标记来标记儿子的权值的增加量。

" 参考代码"

```
#include <bits/stdc++.h>
using namespace std;
const int _ = 5e5 + 10;

namespace trie {
const int _n = _ * 25;
int rt[_];
int ch[_n][2];
int w[_n];  //`w[o]` 指节点 `o` 到其父亲节点这条边上数值的数量（权值）。
int xorv[_n];
int tot = 0;

void maintain(int o) {  // 维护 w 数组和 xorv（权值的异或）数组
  w[o] = xorv[o] = 0;
  if (ch[o][0]) {
    w[o] += w[ch[o][0]];
    xorv[o] ^= xorv[ch[o][0]] << 1;
  }
  if (ch[o][1]) {
    w[o] += w[ch[o][1]];
    xorv[o] ^= (xorv[ch[o][1]] << 1) | (w[ch[o][1]] & 1);
  }
}

int mknode() {  // 创造一个新的节点
  ++tot;
  ch[tot][0] = ch[tot][1] = 0;
  w[tot] = 0;
  return tot;
}

void insert(int &o, int x, int dp) {  // x 是权重，dp 是深度
  if (!o) o = mknode();
  if (dp > 20) return (void)(w[o]++);
  insert(ch[o][x & 1], x >> 1, dp + 1);
```

```
  maintain(o);
}

void erase(int o, int x, int dp) {
  if (dp > 20) return (void)(w[o]--);
  erase(ch[o][x & 1], x >> 1, dp + 1);
  maintain(o);
}

void addall(int o) {  // 对所有节点 +1 即将所有节点的 ch[o][1] 和 ch[o][0] 交换
  swap(ch[o][1], ch[o][0]);
  if (ch[o][0]) addall(ch[o][0]);
  maintain(o);
}
}  // namespace trie

int head[_];

struct edges {
  int node;
  int nxt;
} edge[_ << 1];

int tot = 0;

void add(int u, int v) {
  edge[++tot].nxt = head[u];
  head[u] = tot;
  edge[tot].node = v;
}

int n, m;
int rt;
int lztar[_];
int fa[_];

void dfs0(int o, int f) {  // 得到 fa 数组
  fa[o] = f;
  for (int i = head[o]; i; i = edge[i].nxt) {  // 遍历子节点
    int node = edge[i].node;
    if (node == f) continue;
    dfs0(node, o);
  }
}

int V[_];

int get(int x) { return (fa[x] == -1 ? 0 : lztar[fa[x]]) + V[x]; }  // 权值函数

int main() {
  cin >> n >> m;
  for (int i = 1; i < n; i++) {
    int u, v;
    cin >> u >> v;
```

```
    add(u, v);  // 双向建边
    add(rt = v, u);
  }
  dfs0(rt, -1);  // rt 是随机的一个点
  for (int i = 1; i <= n; i++) {
    cin >> V[i];
    if (fa[i] != -1) trie::insert(trie::rt[fa[i]], V[i], 0);
  }
  while (m--) {
    int opt, x;
    cin >> opt >> x;
    if (opt == 1) {
      lztar[x]++;
      if (x != rt) {
        if (fa[fa[x]] != -1) trie::erase(trie::rt[fa[fa[x]]], get(fa[x]), 0);
        V[fa[x]]++;
        if (fa[fa[x]] != -1)
          trie::insert(trie::rt[fa[fa[x]]], get(fa[x]), 0);  // 重新插入
      }
      trie::addall(trie::rt[x]);  // 对所有节点 +1
    } else if (opt == 2) {
      int v;
      cin >> v;
      if (x != rt) trie::erase(trie::rt[fa[x]], get(x), 0);
      V[x] -= v;
      if (x != rt) trie::insert(trie::rt[fa[x]], get(x), 0);  // 重新插入
    } else {
      int res = 0;
      res = trie::xorv[trie::rt[x]];
      res ^= get(fa[x]);
      printf("%d\n", res);
    }
  }
  return 0;
}
```

"【luogu-P6623】【省选联考 2020 A 卷】树[4]"

给定一棵 n 个结点的有根树 T，结点从 1 开始编号，根结点为 1 号结点，每个结点有一个正整数权值 v_i。设 x 号结点的子树内（包含 x 自身）的所有结点编号为 $c_1, c_2, \ldots, c_k$，定义 x 的价值为：

$val(x) = (v_{c_1} + d(c_1, x)) \oplus (v_{c_2} + d(c_2, x)) \oplus \cdots \oplus (v_{c_k} + d(c_k, x))$ 其中 $d(x, y)$。

表示树上 x 号结点与 y 号结点间唯一简单路径所包含的边数，$d(x, x) = 0$。$\oplus$ 表示异或运算。请你求出 $\sum\limits_{i=1}^{n} val(i)$ 的结果。

"题解"

考虑每个结点对其所有祖先的贡献。每个结点建立 trie，初始先只存这个结点的权值，然后从底向上合并每个儿子结点上的 trie，然后再全局加一，完成后统计答案。

" 参考代码"

```
const int _ = 526010;
int n;
int V[_];
int debug = 0;

namespace trie {
const int MAXH = 21;
int ch[_ * (MAXH + 1)][2], w[_ * (MAXH + 1)], xorv[_ * (MAXH + 1)];
int tot = 0;

int mknode() {
  ++tot;
  ch[tot][1] = ch[tot][0] = w[tot] = xorv[tot] = 0;
  return tot;
}

void maintain(int o) {
  w[o] = xorv[o] = 0;
  if (ch[o][0]) {
    w[o] += w[ch[o][0]];
    xorv[o] ^= xorv[ch[o][0]] << 1;
  }
  if (ch[o][1]) {
    w[o] += w[ch[o][1]];
    xorv[o] ^= (xorv[ch[o][1]] << 1) | (w[ch[o][1]] & 1);
  }
  w[o] = w[o] & 1;
}

void insert(int &o, int x, int dp) {
  if (!o) o = mknode();
  if (dp > MAXH) return (void)(w[o]++);
  insert(ch[o][x & 1], x >> 1, dp + 1);
  maintain(o);
}

int merge(int a, int b) {
  if (!a) return b;
  if (!b) return a;
  w[a] = w[a] + w[b];
  xorv[a] ^= xorv[b];
  ch[a][0] = merge(ch[a][0], ch[b][0]);
  ch[a][1] = merge(ch[a][1], ch[b][1]);
  return a;
}

void addall(int o) {
  swap(ch[o][0], ch[o][1]);
  if (ch[o][0]) addall(ch[o][0]);
  maintain(o);
}
}  // namespace trie
```

```
int rt[_];
long long Ans = 0;
vector<int> E[_];

void dfs0(int o) {
  for (int i = 0; i < E[o].size(); i++) {
    int node = E[o][i];
    dfs0(node);
    rt[o] = trie::merge(rt[o], rt[node]);
  }
  trie::addall(rt[o]);
  trie::insert(rt[o], V[o], 0);
  Ans += trie::xorv[rt[o]];
}

int main() {
  n = read();
  for (int i = 1; i <= n; i++) V[i] = read();
  for (int i = 2; i <= n; i++) E[read()].push_back(i);
  dfs0(1);
  printf("%lld", Ans);
  return 0;
}
```

可持久化字典树

参见 可持久化字典树。

参考资料与注释

[1] 于是他错误的点名开始了

[2] BZOJ1954 最长异或路径

[3] 【luogu-P6018】【Ynoi2010】Fusion tree

[4] 【luogu-P6623】【省选联考 2020 A 卷】树

8.7 前缀函数与 KMP 算法

Authors: Ir1d, LeoJacob, Xeonacid, greyqz, StudyingFather, Marcythm, minghu6, Backl1ght

字符串前缀和后缀定义

关于字符串前缀、真前缀，后缀、真后缀的定义详见 字符串基础

前缀函数

定义

给定一个长度为 n 的字符串 s，其**前缀函数**被定义为一个长度为 n 的数组 π。其中 $\pi[i]$ 的定义是：

1. 如果子串 $s[0\dots i]$ 有一对相等的真前缀与真后缀：$s[0\dots k-1]$ 和 $s[i-(k-1)\dots i]$，那么 $\pi[i]$ 就是这个相等的真前缀（或者真后缀，因为它们相等））的长度，也就是 $\pi[i]=k$；
2. 如果不止有一对相等的，那么 $\pi[i]$ 就是其中最长的那一对的长度；
3. 如果没有相等的，那么 $\pi[i]=0$。

简单来说 $\pi[i]$ 就是，子串 $s[0\dots i]$ 最长的相等的真前缀与真后缀的长度。

用数学语言描述如下：

$$\pi[i]=\max_{k=0\dots i}\{k:s[0\dots k-1]=s[i-(k-1)\dots i]\}$$

特别地，规定 $\pi[0]=0$。

过程

举例来说，对于字符串 `abcabcd`，

$\pi[0]=0$，因为 `a` 没有真前缀和真后缀，根据规定为 0

$\pi[1]=0$，因为 `ab` 无相等的真前缀和真后缀

$\pi[2]=0$，因为 `abc` 无相等的真前缀和真后缀

$\pi[3]=1$，因为 `abca` 只有一对相等的真前缀和真后缀：`a`，长度为 1

$\pi[4]=2$，因为 `abcab` 相等的真前缀和真后缀只有 `ab`，长度为 2

$\pi[5]=3$，因为 `abcabc` 相等的真前缀和真后缀只有 `abc`，长度为 3

$\pi[6]=0$，因为 `abcabcd` 无相等的真前缀和真后缀

同理可以计算字符串 `aabaaab` 的前缀函数为 $[0,1,0,1,2,2,3]$。

计算前缀函数的朴素算法

过程

一个直接按照定义计算前缀函数的算法流程：

- 在一个循环中以 $i=1\to n-1$ 的顺序计算前缀函数 $\pi[i]$ 的值（$\pi[0]$ 被赋值为 0）。
- 为了计算当前的前缀函数值 $\pi[i]$，我们令变量 j 从最大的真前缀长度 i 开始尝试。
- 如果当前长度下真前缀和真后缀相等，则此时长度为 $\pi[i]$，否则令 j 自减 1，继续匹配，直到 $j=0$。
- 如果 $j=0$ 并且仍没有任何一次匹配，则置 $\pi[i]=0$ 并移至下一个下标 $i+1$。

" 实现"

具体实现如下：

```
vector<int> prefix_function(string s) {
  int n = (int)s.length();
  vector<int> pi(n);
  for (int i = 1; i < n; i++)
    for (int j = i; j >= 0; j--)
      if (s.substr(0, j) == s.substr(i - j + 1, j)) {
```

```
        pi[i] = j;
        break;
      }
  return pi;
}

def prefix_function(s):
    n = len(s)
    pi = [0] * n
    for i in range(1, n):
        for j in range(i, -1, -1):
            if s[0 : j] == s[i - j + 1 : i + 1]:
                pi[i] = j
                break
    return pi
```

注:

- `string substr (size_t pos = 0, size_t len = npos) const;`

显见该算法的时间复杂度为 $O(n^3)$，具有很大的改进空间。

计算前缀函数的高效算法

第一个优化

第一个重要的观察是**相邻的前缀函数值至多增加** 1。

参照下图所示，只需如此考虑：当取一个尽可能大的 $\pi[i+1]$ 时，必然要求新增的 $s[i+1]$ 也与之对应的字符匹配，即 $s[i+1]=s[\pi[i]]$, 此时 $\pi[i+1]=\pi[i]+1$。

$$\underbrace{\overbrace{s_0\ s_1\ s_2}^{\pi[i]=3}\ s_3}_{\pi[i+1]=4}\ \cdots\ \underbrace{\overbrace{s_{i-2}\ s_{i-1}\ s_i}^{\pi[i]=3}\ s_{i+1}}_{\pi[i+1]=4}$$

所以当移动到下一个位置时，前缀函数的值要么增加一，要么维持不变，要么减少。

" 实现"

此时的改进的算法为：

```
vector<int> prefix_function(string s) {
  int n = (int)s.length();
  vector<int> pi(n);
  for (int i = 1; i < n; i++)
    for (int j = pi[i - 1] + 1; j >= 0; j--)  // improved: j=i => j=pi[i-1]+1
      if (s.substr(0, j) == s.substr(i - j + 1, j)) {
        pi[i] = j;
        break;
      }
  return pi;
}

def prefix_function(s):
    n = len(s)
    pi = [0] * n
    for i in range(1, n):
```

```
        for j in range(pi[i - 1] + 1, -1, -1):
            if s[0 : j] == s[i - j + 1 : i + 1]:
                pi[i] = j
                break
    return pi
```

在这个初步改进的算法中，在计算每个 $\pi[i]$ 时，最好的情况是第一次字符串比较就完成了匹配，也就是说基础的字符串比较次数是 `n-1` 次。

而由于存在 `j = pi[i-1]+1`（`pi[0]=0`）对于最大字符串比较次数的限制，可以看出每次只有在最好情况才会为字符串比较次数的上限积累 1，而每次超过一次的字符串比较消耗的是之后次数的增长空间。

由此我们可以得出字符串比较次数最多的一种情况：至少 `1` 次字符串比较次数的消耗和最多 `n-2` 次比较次数的积累，此时字符串比较次数为 `n-1 + n-2 = 2n-3`。

可见经过此次优化，计算前缀函数只需要进行 $O(n)$ 次字符串比较，总复杂度降为了 $O(n^2)$。

第二个优化

在第一个优化中，我们讨论了计算 $\pi[i+1]$ 时的最好情况：$s[i+1]=s[\pi[i]]$，此时 $\pi[i+1]=\pi[i]+1$。现在让我们沿着这个思路走得更远一点：讨论当 $s[i+1]\neq s[\pi[i]]$ 时如何跳转。

失配时，我们希望找到对于子串 $s[0\dots i]$，仅次于 $\pi[i]$ 的第二长度 j，使得在位置 i 的前缀性质仍得以保持，也即 $s[0\dots j-1]=s[i-j+1\dots i]$：

$$\overbrace{\underbrace{s_0\ s_1}_{j}\ s_2\ s_3}^{\pi[i]}\ \dots\ \overbrace{s_{i-3}\ s_{i-2}\ \underbrace{s_{i-1}\ s_i}_{j}}^{\pi[i]}\ s_{i+1}$$

如果我们找到了这样的长度 j，那么仅需要再次比较 $s[i+1]$ 和 $s[j]$。如果它们相等，那么就有 $\pi[i+1]=j+1$。否则，我们需要找到子串 $s[0\dots i]$ 仅次于 j 的第二长度 $j^{(2)}$，使得前缀性质得以保持，如此反复，直到 $j=0$。如果 $s[i+1]\neq s[0]$，则 $\pi[i+1]=0$。

观察上图可以发现，因为 $s[0\dots\pi[i]-1]=s[i-\pi[i]+1\dots i]$，所以对于 $s[0\dots i]$ 的第二长度 j，有这样的性质：

$$s[0\dots j-1]=s[i-j+1\dots i]=s[\pi[i]-j\dots\pi[i]-1]$$

也就是说 j 等价于子串 $s[\pi[i]-1]$ 的前缀函数值，即 $j=\pi[\pi[i]-1]$。同理，次于 j 的第二长度等价于 $s[j-1]$ 的前缀函数值，$j^{(2)}=\pi[j-1]$

显然我们可以得到一个关于 j 的状态转移方程：$j^{(n)}=\pi[j^{(n-1)}-1],\ \ (j^{(n-1)}>0)$

最终算法

所以最终我们可以构建一个不需要进行任何字符串比较，并且只进行 $O(n)$ 次操作的算法。

而且该算法的实现出人意料的短且直观：

" 实现"

```
vector<int> prefix_function(string s) {
  int n = (int)s.length();
  vector<int> pi(n);
  for (int i = 1; i < n; i++) {
    int j = pi[i - 1];
    while (j > 0 && s[i] != s[j]) j = pi[j - 1];
    if (s[i] == s[j]) j++;
    pi[i] = j;
  }
  return pi;
}
```

```python
def prefix_function(s):
    n = len(s)
    pi = [0] * n
    for i in range(1, n):
        j = pi[i - 1]
        while j > 0 and s[i] != s[j]:
            j = pi[j - 1]
        if s[i] == s[j]:
            j += 1
        pi[i] = j
    return pi
```

这是一个**在线**算法，即其当数据到达时处理它——举例来说，你可以一个字符一个字符的读取字符串，立即处理它们以计算出每个字符的前缀函数值。该算法仍然需要存储字符串本身以及先前计算过的前缀函数值，但如果我们已经预先知道该字符串前缀函数的最大可能取值 M，那么我们仅需要存储该字符串的前 $M+1$ 个字符以及对应的前缀函数值。

应用

在字符串中查找子串：Knuth–Morris–Pratt 算法

该算法由 Knuth、Pratt 和 Morris 在 1977 年共同发布[1][3]。

该任务是前缀函数的一个典型应用。

过程

给定一个文本 t 和一个字符串 s，我们尝试找到并展示 s 在 t 中的所有出现（occurrence）。

为了简便起见，我们用 n 表示字符串 s 的长度，用 m 表示文本 t 的长度。

我们构造一个字符串 $s+\#+t$，其中 $\#$ 为一个既不出现在 s 中也不出现在 t 中的分隔符。接下来计算该字符串的前缀函数。现在考虑该前缀函数除去最开始 $n+1$ 个值（即属于字符串 s 和分隔符的函数值）后其余函数值的意义。根据定义，$\pi[i]$ 为右端点在 i 且同时为一个前缀的最长真子串的长度，具体到我们的这种情况下，其值为与 s 的前缀相同且右端点位于 i 的最长子串的长度。由于分隔符的存在，该长度不可能超过 n。而如果等式 $\pi[i]=n$ 成立，则意味着 s 完整出现在该位置（即其右端点位于位置 i）。注意该位置的下标是对字符串 $s+\#+t$ 而言的。

因此如果在某一位置 i 有 $\pi[i]=n$ 成立，则字符串 s 在字符串 t 的 $i-(n-1)-(n+1)=i-2n$ 处出现。

正如在前缀函数的计算中已经提到的那样，如果我们知道前缀函数的值永远不超过一特定值，那么我们不需要存储整个字符串以及整个前缀函数，而只需要二者开头的一部分。在我们这种情况下这意味着只需要存储字符串 $s+\#$ 以及相应的前缀函数值即可。我们可以一次读入字符串 t 的一个字符并计算当前位置的前缀函数值。

因此 Knuth–Morris–Pratt 算法（简称 KMP 算法）用 $O(n+m)$ 的时间以及 $O(n)$ 的内存解决了该问题。

" 实现"

```cpp
vector<int> find_occurrences(string text, string pattern) {
  string cur = pattern + '#' + text;
  int sz1 = text.size(), sz2 = pattern.size();
  vector<int> v;
  vector<int> lps = prefix_function(cur);
  for (int i = sz2 + 1; i <= sz1 + sz2; i++) {
    if (lps[i] == sz2) v.push_back(i - 2 * sz2);
  }
  return v;
}
```

```python
def find_occurrences(t, s):
    cur = s + '#' + t
    sz1, sz2 = len(t), len(s)
    ret = []
    lps = prefix_function(cur)
    for i in range(sz2 + 1, sz1 + sz2 + 1):
        if lps[i] == sz2:
            ret.append(i - 2 * sz2)
    return ret
```

字符串的周期

对字符串 s 和 $0 < p \le |s|$，若 $s[i] = s[i+p]$ 对所有 $i \in [0, |s| - p - 1]$ 成立，则称 p 是 s 的周期。

对字符串 s 和 $0 \le r < |s|$，若 s 长度为 r 的前缀和长度为 r 的后缀相等，就称 s 长度为 r 的前缀是 s 的 border。

由 s 有长度为 r 的 border 可以推导出 $|s| - r$ 是 s 的周期。

根据前缀函数的定义，可以得到 s 所有的 border 长度，即 $\pi[n-1], \pi[\pi[n-1]-1], \ldots$。[2]

所以根据前缀函数可以在 $O(n)$ 的时间内计算出 s 所有的周期。其中，由于 $\pi[n-1]$ 是 s 最长 border 的长度，所以 $n - \pi[n-1]$ 是 s 的最小周期。

统计每个前缀的出现次数

在该节我们将同时讨论两个问题。给定一个长度为 n 的字符串 s，在问题的第一个变种中我们希望统计每个前缀 $s[0 \ldots i]$ 在同一个字符串的出现次数，在问题的第二个变种中我们希望统计每个前缀 $s[0 \ldots i]$ 在另一个给定字符串 t 中的出现次数。

首先让我们来解决第一个问题。考虑位置 i 的前缀函数值 $\pi[i]$。根据定义，其意味着字符串 s 一个长度为 $\pi[i]$ 的前缀在位置 i 出现并以 i 为右端点，同时不存在一个更长的前缀满足前述定义。与此同时，更短的前缀可能以该位置为右端点。容易看出，我们遇到了在计算前缀函数时已经回答过的问题：给定一个长度为 j 的前缀，同时其也是一个右端点位于 i 的后缀，下一个更小的前缀长度 $k < j$ 是多少？该长度的前缀需同时也是一个右端点为 i 的后缀。因此以位置 i 为右端点，有长度为 $\pi[i]$ 的前缀，有长度为 $\pi[\pi[i]-1]$ 的前缀，有长度为 $\pi[\pi[\pi[i]-1]-1]$ 的前缀，等等，直到长度变为 0。故而我们可以通过下述方式计算答案。

" 实现"

```cpp
vector<int> ans(n + 1);
for (int i = 0; i < n; i++) ans[pi[i]]++;
for (int i = n - 1; i > 0; i--) ans[pi[i - 1]] += ans[i];
for (int i = 0; i <= n; i++) ans[i]++;
```

```python
ans = [0] * (n + 1)
for i in range(0, n):
    ans[pi[i]] += 1
for i in range(n - 1, 0, -1):
    ans[pi[i - 1]] += ans[i]
for i in range(0, n + 1):
    ans[i] += 1
```

解释

在上述代码中我们首先统计每个前缀函数值在数组 π 中出现了多少次，然后再计算最后答案：如果我们知道长度为 i 的前缀出现了恰好 $\text{ans}[i]$ 次，那么该值必须被叠加至其最长的既是后缀也是前缀的子串的出现次数中。在最后，为了统计原始的前缀，我们对每个结果加 1。

现在考虑第二个问题。我们应用来自 Knuth–Morris–Pratt 的技巧：构造一个字符串 $s + \# + t$ 并计算其前缀函

数。与第一个问题唯一的不同之处在于，我们只关心与字符串 t 相关的前缀函数值，即 $i \geq n+1$ 的 $\pi[i]$。有了这些值之后，我们可以同样应用在第一个问题中的算法来解决该问题。

一个字符串中本质不同子串的数目

给定一个长度为 n 的字符串 s，我们希望计算其本质不同子串的数目。

我们将迭代的解决该问题。换句话说，在知道了当前的本质不同子串的数目的情况下，我们要找出一种在 s 末尾添加一个字符后重新计算该数目的方法。

令 k 为当前 s 的本质不同子串数量。我们添加一个新的字符 c 至 s。显然，会有一些新的子串以字符 c 结尾。我们希望对这些以该字符结尾且我们之前未曾遇到的子串计数。

构造字符串 $t = s + c$ 并将其反转得到字符串 $t^{\sim}$。现在我们的任务变为计算有多少 $t^{\sim}$ 的前缀未在 $t^{\sim}$ 的其余任何地方出现。如果我们计算了 $t^{\sim}$ 的前缀函数最大值 $\pi_{\max}$，那么最长的出现在 s 中的前缀其长度为 $\pi_{\max}$。自然的，所有更短的前缀也出现了。

因此，当添加了一个新字符后新出现的子串数目为 $|s| + 1 - \pi_{\max}$。

所以对于每个添加的字符，我们可以在 $O(n)$ 的时间内计算新子串的数目，故最终复杂度为 $O(n^2)$。

值得注意的是，我们也可以重新计算在头部添加一个字符，或者从尾或者头移除一个字符时的本质不同子串数目。

字符串压缩

给定一个长度为 n 的字符串 s，我们希望找到其最短的「压缩」表示，也即我们希望寻找一个最短的字符串 t，使得 s 可以被 t 的一份或多份拷贝的拼接表示。

显然，我们只需要找到 t 的长度即可。知道了该长度，该问题的答案即为长度为该值的 s 的前缀。

让我们计算 s 的前缀函数。通过使用该函数的最后一个值 $\pi[n-1]$，我们定义值 $k = n - \pi[n-1]$。我们将证明，如果 k 整除 n，那么 k 就是答案，否则不存在一个有效的压缩，故答案为 n。

假定 n 可被 k 整除。那么字符串可被划分为长度为 k 的若干块。根据前缀函数的定义，该字符串长度为 $n-k$ 的前缀等于其后缀。但是这意味着最后一个块同倒数第二个块相等，并且倒数第二个块同倒数第三个块相等，等等。作为其结果，所有块都是相等的，因此我们可以将字符串 s 压缩至长度 k。

" 证明"

诚然，我们仍需证明该值为最优解。实际上，如果有一个比 k 更小的压缩表示，那么前缀函数的最后一个值 $\pi[n-1]$ 必定比 $n-k$ 要大。因此 k 就是答案。

现在假设 n 不可以被 k 整除，我们将通过反证法证明这意味着答案为 n[1]。假设其最小压缩表示 r 的长度为 p（p 整除 n），字符串 s 被划分为 $n/p \geq 2$ 块。那么前缀函数的最后一个值 $\pi[n-1]$ 必定大于 $n-p$（如果等于则 n 可被 k 整除），也即其所表示的后缀将部分的覆盖第一个块。现在考虑字符串的第二个块。该块有两种解释：第一种为 $r_0 r_1 \dots r_{p-1}$，另一种为 $r_{p-k} r_{p-k+1} \dots r_{p-1} r_0 r_1 \dots r_{p-k-1}$。由于两种解释对应同一个字符串，因此可得到 p 个方程组成的方程组，该方程组可简写为 $r_{(i+k) \bmod p} = r_{i \bmod p}$，其中 $\cdot \bmod p$ 表示模 p 意义下的最小非负剩余。

$$\overbrace{r_0\ r_1\ r_2\ r_3\ r_4\ r_5}^{p}\ \overbrace{r_0\ r_1\ r_2\ r_3\ r_4 r_5}^{p}$$

$$r_0\ r_1\ r_2\ r_3\ \underbrace{\overbrace{r_0\ r_1\ r_2\ r_3\ r_4\ r_5}^{p}\ r_0\ r_1}_{\pi[11]=8}$$

根据扩展欧几里得算法我们可以得到一组 x 和 y 使得 $xk + yp = \gcd(k,p)$。通过与等式 $pk - kp = 0$ 适当叠加我们可以得到一组 $x' > 0$ 和 $y' < 0$ 使得 $x'k + y'p = \gcd(k,p)$。这意味着通过不断应用前述方程组中的方程我们可以得到新的方程组 $r_{(i+\gcd(k,p)) \bmod p} = r_{i \bmod p}$。

由于 $\gcd(k,p)$ 整除 p，这意味着 $\gcd(k,p)$ 是 r 的一个周期。又因为 $\pi[n-1] > n-p$，故有 $n - \pi[n-1] = k < p$，所以 $\gcd(k,p)$ 是一个比 p 更小的 r 的周期。因此字符串 s 有一个长度为 $\gcd(k,p) < p$ 的压缩表示，同 p 的最小性矛盾。

综上所述，不存在一个长度小于 k 的压缩表示，因此答案为 k。

根据前缀函数构建一个自动机

让我们重新回到通过一个分隔符将两个字符串拼接的新字符串。对于字符串 s 和 t 我们计算 $s+\#+t$ 的前缀函数。显然，因为 $\#$ 是一个分隔符，前缀函数值永远不会超过 $|s|$。因此我们只需要存储字符串 $s+\#$ 和其对应的前缀函数值，之后就可以动态计算对于之后所有字符的前缀函数值：

$$\underbrace{s_0\ s_1\ \dots\ s_{n-1}\ \#}_{\text{need to store}}\ \underbrace{t_0\ t_1\ \dots\ t_{m-1}}_{\text{do not need to store}}$$

实际上在这种情况下，知道 t 的下一个字符 c 以及之前位置的前缀函数值便足以计算下一个位置的前缀函数值，而不需要用到任何其它 t 的字符和对应的前缀函数值。

换句话说，我们可以构造一个**自动机**（一个有限状态机）：其状态为当前的前缀函数值，而从一个状态到另一个状态的转移则由下一个字符确定。

因此，即使没有字符串 t，我们同样可以应用构造转移表的算法构造一个转移表 (old $\pi, c) \to$ new _π：

" 实现"

```
void compute_automaton(string s, vector<vector<int>>& aut) {
  s += '#';
  int n = s.size();
  vector<int> pi = prefix_function(s);
  aut.assign(n, vector<int>(26));
  for (int i = 0; i < n; i++) {
    for (int c = 0; c < 26; c++) {
      int j = i;
      while (j > 0 && 'a' + c != s[j]) j = pi[j - 1];
      if ('a' + c == s[j]) j++;
      aut[i][c] = j;
    }
  }
}
```

然而在这种形式下，对于小写字母表，算法的时间复杂度为 $O(|\Sigma|n^2)$。注意到我们可以应用动态规划来利用表中已计算过的部分。只要我们从值 j 变化到 $\pi[j-1]$，那么我们实际上在说转移 (j,c) 所到达的状态同转移 $(\pi[j-1],c)$ 一样，但该答案我们之前已经精确计算过了。

" 实现"

```
void compute_automaton(string s, vector<vector<int>>& aut) {
  s += '#';
  int n = s.size();
  vector<int> pi = prefix_function(s);
  aut.assign(n, vector<int>(26));
  for (int i = 0; i < n; i++) {
    for (int c = 0; c < 26; c++) {
      if (i > 0 && 'a' + c != s[i])
        aut[i][c] = aut[pi[i - 1]][c];
      else
        aut[i][c] = i + ('a' + c == s[i]);
    }
  }
}
```

最终我们可在 $O(|\Sigma|n)$ 的时间复杂度内构造该自动机。

该自动机在什么时候有用呢？首先，记得大部分时候我们为了一个目的使用字符串 $s + \# + t$ 的前缀函数：寻找字符串 s 在字符串 t 中的所有出现。

因此使用该自动机的最直接的好处是**加速计算字符串 $s + \# + t$ 的前缀函数**。

通过构建 $s+\#$ 的自动机，我们不再需要存储字符串 s 以及其对应的前缀函数值。所有转移已经在表中计算过了。

但除此以外，还有第二个不那么直接的应用。我们可以在字符串 t 是**某些通过一些规则构造的巨型字符串**时，使用该自动机加速计算。Gray 字符串，或者一个由一些短的输入串的递归组合所构造的字符串都是这种例子。

出于完整性考虑，我们来解决这样一个问题：给定一个数 $k \le 10^5$，以及一个长度 $\le 10^5$ 的字符串 s，我们需要计算 s 在第 k 个 Gray 字符串中的出现次数。回想起 Gray 字符串以下述方式定义：

$$g_1 = \mathsf{a}$$
$$g_2 = \mathsf{aba}$$
$$g_3 = \mathsf{abacaba}$$
$$g_4 = \mathsf{abacabadabacaba}$$

由于其天文数字般的长度，在这种情况下即使构造字符串 t 都是不可能的：第 k 个 Gray 字符串有 $2^k - 1$ 个字符。然而我们可以在仅仅知道开头若干前缀函数值的情况下，有效计算该字符串末尾的前缀函数值。

除了自动机之外，我们同时需要计算值 $G[i][j]$：在从状态 j 开始处理 g_i 后的自动机的状态，以及值 $K[i][j]$：当从状态 j 开始处理 g_i 后，s 在 g_i 中的出现次数。实际上 $K[i][j]$ 为在执行操作时前缀函数取值为 $|s|$ 的次数。易得问题的答案为 $K[k][0]$。

我们该如何计算这些值呢？首先根据定义，初始条件为 $G[0][j] = j$ 以及 $K[0][j] = 0$。之后所有值可以通过先前的值以及使用自动机计算得到。为了对某个 i 计算相应值，回想起字符串 g_i 由 g_{i-1}，字母表中第 i 个字符，以及 g_{i-1} 三者拼接而成。因此自动机会途径下列状态：

$$\mathrm{mid} = \mathrm{aut}[G[i-1][j]][i]$$
$$G[i][j] = G[i-1][\mathrm{mid}]$$

$K[i][j]$ 的值同样可被简单计算。

$$K[i][j] = K[i-1][j] + [\mathrm{mid} == |s|] + K[i-1][\mathrm{mid}]$$

其中 $[\cdot]$ 当其中表达式取值为真时值为 1，否则为 0。综上，我们已经可以解决关于 Gray 字符串的问题，以及一大类与之类似的问题。举例来说，应用同样的方法可以解决下列问题：给定一个字符串 s 以及一些模式 t_i，其中每个模式以下列方式给出：该模式由普通字符组成，当中可能以 t_k^{cnt} 的形式递归插入先前的字符串，也即在该位置我们必须插入字符串 t_k cnt 次。以下是这些模式的一个例子：

$$t_1 = \mathsf{abdeca}$$
$$t_2 = \mathsf{abc} + t_1^{30} + \mathsf{abd}$$
$$t_3 = t_2^{50} + t_1^{100}$$
$$t_4 = t_2^{10} + t_3^{100}$$

递归代入会使字符串长度爆炸式增长，他们的长度甚至可以达到 100^{100} 的数量级。而我们必须找到字符串 s 在每个字符串中的出现次数。

该问题同样可通过构造前缀函数的自动机解决。同之前一样，我们利用先前计算过的结果对每个模式计算其转移然后相应统计答案即可。

练习题目

- UVa 455 "Periodic Strings"[4]
- UVa 11022 "String Factoring"[5]

- UVa 11452 "Dancing the Cheeky-Cheeky"[6]
- UVa 12604 - Caesar Cipher[7]
- UVa 12467 - Secret Word[8]
- UVa 11019 - Matrix Matcher[9]
- SPOJ - Pattern Find[10]
- Codeforces - Anthem of Berland[11]
- Codeforces - MUH and Cube Walls[12]

参考资料与注释

本页面主要译自博文 - . - -[13] 与其英文翻译版 Prefix function. Knuth–Morris–Pratt algorithm[14]。其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0。

[1] 在俄文版及英文版中该部分证明均疑似有误。本文章中的该部分证明由作者自行添加。

[2] 金策 - 字符串算法选讲

[3] [1]

[4] UVa 455 "Periodic Strings"

[5] UVa 11022 "String Factoring"

[6] UVa 11452 "Dancing the Cheeky-Cheeky"

[7] UVa 12604 - Caesar Cipher

[8] UVa 12467 - Secret Word

[9] UVa 11019 - Matrix Matcher

[10] SPOJ - Pattern Find

[11] Codeforces - Anthem of Berland

[12] Codeforces - MUH and Cube Walls

[13] - . - -

[14] Prefix function. Knuth–Morris–Pratt algorithm

8.8 Boyer–Moore 算法

Authors: minghu6

本章节内容需要以《前缀函数与 KMP 算法》作为前置章节。

之前的 KMP 算法将前缀匹配的信息用到了极致，

而 BM 算法背后的基本思想是通过后缀匹配获得比前缀匹配更多的信息来实现更快的字符跳转。

引入

想象一下，如果我们的的模式字符串 pat，被放在文本字符串 $string$ 的左手起头部，使它们的第一个字符对齐。

$$
\begin{array}{ll}
pat: & \texttt{EXAMPLE} \\
string: & \texttt{HERE IS A SIMPLE EXAMPLE} \ldots \\
 & \qquad\qquad \Uparrow
\end{array}
$$

在这里做定义，往后不赘述：

pat 的长度为 $patlen$，特别地对于从 0 开始的串来说，规定 $patlastpos = patlen - 1$ 为 pat 串最后一个字符的位置；

$string$ 的长度 $stringlen$，$stringlastpos = stringlen - 1$。

假如我们知道了 $string$ 的第 $patlen$ 个字符 $char$（与 pat 的最后一个字符对齐）考虑我们能得到什么信息：

观察 1

如果我们知道 $char$ 这个字符不在 pat 中，我们就不用考虑 pat 从 $string$ 的第 1 个、第 2 个……第 $patlen$ 个字符起出现的情况，，而可以直接将 pat 向下滑动 $patlen$ 个字符。

观察 2

更一般地，**如果出现在 pat 最末尾（也就是最右边）的那一个 $char$ 字符的位置是离末尾端差了 $delta_1$ 个字符**，

那么就可以不用匹配，直接将 pat 向后滑动 $delta_1$ 个字符：如果滑动距离少于 $delta_1$，那么仅就 $char$ 这个字符就无法被匹配，当然模式字符串 pat 也就不会被匹配。

因此除非 $char$ 字符可以和 pat 末尾的那个字符匹配，否则 $string$ 要跳过 $delta_1$ 个字符（相当于 pat 向后滑动了 $delta_1$ 个字符）。并且我们可以得到一个计算 $delta_1$ 的函数 $delta_1(char)$：

```
int delta1(char char)
    if char 不在 pat 中 || char 是 pat 上最后一个字符
        return patlen
    else
        return patlastpos − i    // i 为出现在 pat 最末尾的那一个 char 出现的位置，即 pat[i]=char
```

注意：显然这个表只需计算到 $patlastpos - 1$ 的位置

现在假设 $char$ 和 pat 最后一个字符匹配到了，那我们就看看 $char$ 前一个字符和 pat 的倒数第二个字符是否匹配：

如果是，就继续回退直到整个模式串 pat 完成匹配（这时我们就在 $string$ 上成功得到了一个 pat 的匹配）；

或者，我们也可能会在匹配完 pat 的倒数第 m 个字符后，在倒数第 $m+1$ 个字符上失配，这时我们就希望把 pat 向后滑动到下一个可能会实现匹配的位置，当然我们希望滑动得越远越好。

观察 3(a)

在**观察 2** 中提到，当匹配完 pat 的倒数 m 个字符后，如果在倒数第 $m+1$ 个字符失配，为了使得 $string$ 中的失配字符与 pat 上对应字符对齐，

需要把 pat 向后滑动 k 个字符，也就是说我们应该把注意力看向之后的 $k+m$ 个字符（也就是看向 pat 滑动 k 之后，末段与 $string$ 对齐的那个字符）。

而 $k = delta_1 - m$，

所以我们的注意力应该沿着 $string$ 向后跳 $delta_1 - m + m = delta_1$ 个字符。

然而，我们有机会跳过更多的字符，请继续看下去。

观察 3(b)

如果我们知道 $string$ 接下来的 m 个字符和 pat 的最后 m 个字符匹配，假设这个子串为 $subpat$，

我们还知道在 $string$ 失配字符 $char$ 后面是与 $subpat$ 相匹配的子串，而假如 pat 对应失配字符前面存在 $subpat$，我们可以将 pat 向下滑动一段距离，

使得失配字符 $char$ 在 pat 上对应的字符前面出现的 $subpat$（合理重现，plausible reoccurrence，以下也简称 pr）与 $string$ 的 $subpat$ 对齐。如果 pat 上有多个 $subpat$，按照从右到左的后缀匹配顺序，取第一个（rightmost plausible reoccurrence，以下也简称 rpr）。

假设此时 pat 向下滑动的 k 个字符（也即 pat 末尾端的 $subpat$ 与其最右边的合理重现的距离），这样我们的注意力应该沿着 $string$ 向后滑动 $k + m$ 个字符，这段距离我们称之为 $delta_2(j)$：

假定 $rpr(j)$ 为 $subpat = pat[j + 1 \ldots patlastpos]$ 在 $pat[j]$ 上失配时的最右边合理重现的位置，$rpr(j) < j$（这里只给出简单定义，在下文的算法设计章节里会有更精确的讨论），那么显然 $k = j - rpr(j)$，$m = patlastpos - j$。

所以有：

$$\textbf{int } delta2(\textbf{int } j) \quad \text{// j 为失配字符在 pat 上对应字符的位置}$$
$$\textbf{return } patlastpos - rpr(j)$$

于是我们在失配时，可以把把 $string$ 上的注意力往后跳过 $\max(delta_1, delta_2)$ 个字符

过程

箭头指向失配字符 $char$：

```
pat:              AT-THAT
string:    ... WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
                        ⇑
```

`F` 没有出现 pat 中，根据**观察 1**，pat 直接向下移动 $patlen$ 个字符，也就是 7 个字符：

```
pat:                     AT-THAT
string:    ... WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
                               ⇑
```

根据**观察 2**，我们需要将 pat 向下移动 4 个字符使得短横线字符对齐：

```
pat:                         AT-THAT
string:    ... WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
                                   ⇑
```

现在 $char$:`T` 匹配了，把 $string$ 上的指针左移一步继续匹配：

```
pat:                         AT-THAT
string:    ... WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
                                  ⇑
```

根据**观察 3(a)**，`L` 失配，因为 `L` 不在 pat 中，所以 pat 向下移动 $k = delta_1 - m = 7 - 1 = 6$ 个字符，而 $string$ 上指针向下移动 $delta_1 = 7$ 个字符：

```
pat:                               AT-THAT
string:    ... WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
                                         ⇑
```

这时 $char$ 又一次匹配到了 pat 的最后一个字符 `T`，$string$ 上的指针向左匹配，匹配到了 `A`，继续向左匹配，发现在字符 `-` 失配：

```
pat:                        AT-THAT
string:  ... WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
                              ⇑
```

显然直观上看，此时根据**观察 3(b)**，将 pat 向下移动 $k=5$ 个字符，使得后缀 `AT` 对齐，这种滑动可以获得 $string$ 指针最大的滑动距离，此时 $delta_2 = k + patlastpos - j = 5 + 6 - 4 = 7$，即 $string$ 上指针向下滑动 7 个字符。

而从形式化逻辑看，此时，$delta_1 = 7 - 1 - 2 = 4$，$delta_2 = 7, \max(delta_1, delta_2) = 7$，这样从形式逻辑上支持了进行**观察 3(b)** 的跳转：

```
pat:                             AT-THAT
string:  ... WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
                                      ⇑
```

现在我们发现了 pat 上每一个字符都和 $string$ 上对应的字符相等，我们在 $string$ 上找到了一个 pat 的匹配。而我们只花费了 14 次对 $string$ 的引用，其中 7 次是完成一个成功的匹配所必需的比较次数（$patlen = 7$），另外 7 次让我们跳过了 22 个字符，Amazing（浮夸口气）!

算法设计

最初的匹配算法

解释

现在看这样一个利用 $delta_1$ 和 $delta_2$ 进行字符串匹配的算法：

$i \leftarrow patlastpos.$
$j \leftarrow patlastpos.$
loop
  if $j < 0$
    return $i + 1$

  if $string[i] = pat[j]$
    $j \leftarrow j - 1$
    $i \leftarrow i - 1$
    continue

  $i \leftarrow i + max(delta_1(string[i]), delta_2(j))$

  if $i > stringlastpos$
    return $false$
  $j \leftarrow patlastpos$

如果上面的算法 **return** $false$，表明 pat 不在 $string$ 中；如果返回一个数字，表示 pat 在 $string$ 左起第一次出现的位置。

然后让我们更精细地描述下计算 $delta_2$，所依靠的 $rpr(j)$ 函数。

根据前文定义，$rpr(j)$ 表示在 $pat(j)$ 失配时，子串 $subpat = pat[j+1 \dots patlastpos]$ 在 $pat[j]$ 最右边合理重现的位置。

也就是说需要找到一个最好的 k, 使得 $pat[k \dots k + patlastpos - j - 1] = pat[j+1 \dots patlastpos]$，另外要考虑两种特殊情况：

1. 当 $k<0$ 时，相当于在 pat 前面补充了一段虚拟的前缀，实际上也符合 $delta_2$ 跳转的原理。
2. 当 $k>0$ 时，如果 $pat[k-1]=pat[j]$，则这个 $pat[k \dots k+patlastpos-j-1]$ 不能作为 $subpat$ 的合理重现。原因是 $pat[j]$ 本身是失配字符，所以 pat 向下滑动 k 个字符后，在后缀匹配过程中仍然会在 $pat[k-1]$ 处失配。

还要注意两个限制条件：

1. $k<j$。因为当 $k=j$ 时，有 $pat[k]=pat[j]$，在 $pat[j]$ 上失配的字符也会在 $pat[k]$ 上失配。
2. 考虑到 $delta_2(patlastpos)=0$，所以规定 $rpr(patlastpos)=patlastpos$。

过程

由于理解 $rpr(j)$ 是实现 BoyerMoore 算法的核心，所以我们使用如下两个例子进行详细说明：

j：	`0 1 2 3 4 5 6 7 8`
pat：	`A B C X X X A B C`
$rpr(j)$：	`5 4 3 2 1 0 2 1 8`
sgn：	`- - - - - - - - +`

对于 $rpr(0)$，$subpat$ 为 `BCXXXABC`，在 $pat[0]$ 之前的最右边合理重现只能是 `[(BCXXX)ABC]XXXABC`，也就是最右边合理重现位置为 -5，即 $rpr(j)=-5$；

对于 $rpr(1)$，$subpat$ 为 `CXXXABC`，在 $pat[1]$ 之前的最右边的合理重现是 `[(CXXX)ABC]XXXABC`，所以 $rpr(j)=-4$；

对于 $rpr(2)$，$subpat$ 为 `XXXABC`，在 $pat[2]$ 之前的最右边的合理重现是 `[(XXX)ABC]XXXABC`，所以 $rpr(j)=-3$；

对于 $rpr(3)$，$subpat$ 为 `XXABC`，在 $pat[3]$ 之前的最右边的合理重现是 `[(XX)ABC]XXXABC`，所以 $rpr(j)=-2$；

对于 $rpr(4)$，$subpat$ 为 `XABC`，在 $pat[4]$ 之前的最右边的合理重现是 `[(X)ABC]XXXABC`，所以 $rpr(j)=-1$；

对于 $rpr(5)$，$subpat$ 为 `ABC`，在 $pat[5]$ 之前的最右边的合理重现是 `[ABC]XXXABC`，所以 $rpr(j)=0$；

对于 $rpr(6)$，$subpat$ 为 `BC`，又因为 $string[0]=string[6]$，即 $string[0]$ 等于失配字符 $string[6]$，所以 $string[0 \dots 2]$ 并不是符合条件的 $subpat$ 的合理重现，所以在最右边的合理重现是 `[(BC)]ABCXXXABC`，所以 $rpr(j)=-2$；

对于 $rpr(7)$，$subpat$ 为 `C`，同理又因为 $string[7]=string[1]$，所以 $string[1 \dots 2]$ 并不是符合条件的 $subpat$ 的合理重现，在最右边的合理重现是 `[(C)]ABCXXXABC`，所以 $rpr(j)=-1$；

对于 $rpr(8)$，根据 $delta_2$ 定义，$rpr(patlastpos)=patlastpos$，得到 $rpr(8)=8$。

现在再看一下另一个例子：

j：	`0 1 2 3 4 5 6 7 8`
pat：	`A B Y X C D E Y X`
$rpr(j)$：	`8 7 6 5 4 3 2 1 8`
sgn：	`- - - - - - + - +`

对于 $rpr(0)$，$subpat$ 为 `BYXCDEYX`，在 $pat[0]$ 之前的最右边合理重现只能是 `[(BYXCDEYX)]ABYXCDEYX`，也就是最右边合理重现位置为 -8，即 $rpr(j)=-8$；

对于 $rpr(1)$，$subpat$ 为 `YXCDEYX`，在 $pat[1]$ 之前的最右边合理重现只能是 `[(YXCDEYX)]ABYXCDEYX`，$rpr(j)=-7$；

对于 $rpr(2)$，$subpat$ 为 `XCDEYX`，在 $pat[2]$ 之前的最右边合理重现只能是 `[(XCDEYX)]ABYXCDEYX`，$rpr(j)=-6$；

对于 $rpr(3)$，$subpat$ 为 `CDEYX`，在 $pat[3]$ 之前的最右边合理重现只能是 `[(CDEYX)]ABYXCDEYX`，$rpr(j)=-5$；

对于 $rpr(4)$，$subpat$ 为 `DEYX`，在 $pat[4]$ 之前的最右边合理重现只能是 `[(DEYX)]ABYXCDEYX`，$rpr(j)=-4$；

对于 $rpr(5)$，$subpat$ 为 `EYX`，在 $pat[5]$ 之前的最右边合理重现只能是 `[(EYX)]ABYXCDEYX`，$rpr(j)=-3$；

对于 $rpr(6)$，$subpat$ 为 `YX`，因为 $string[2 \dots 3]=string[7 \dots 8]$ 并且有 $string[6] \neq string[1]$，所以在 $pat[6]$ 之前的最右边的合理重现是 `AB[YX]CDEYX`，$rpr(j)=2$；

对于 $rpr(7)$，$subpat$ 为 `X`，虽然 $string[3]=string[8]$ 但是因为 $string[2]=string[7]$，所以在 $pat[7]$ 之前的最右边的合理重现是 `[X]ABYXCDEYX`，$rpr(j)=-1$；

对于 $rpr(8)$，根据 $delta_2$ 定义，$rpr(patlastpos)=patlastpos$，得到 $rpr(8)=8$。

对匹配算法的一个改进

最后，实践过程中考虑到搜索过程中估计有 80% 的时间用在了基于**观察 1** 的跳转上，也就是 $string[i]$ 和 $pat[patlastpos]$ 不匹配，然后跳越整个 $patlen$ 进行下一次匹配的过程。

于是，可以为此进行特别的优化：

我们定义一个 $delta0$：

```
int delta0(char char)
    if char = pat[patlastpos]
        return large  // large 为一个整数，需要满足 large>stringlastpos+patlen
    return delta1(char)
```

用 $delta0$ 代替 $delta_1$，得到改进后的匹配算法：

```
i ← patlastpos
loop
    if i > stringlastpos
        return false

    while i < stringlen
        i ← i + delta0(string(i))  // 除非 string[i] 和 pat 末尾字符匹配，否则至多向下滑动 patlen
    if i ⩽ large                   //此时表示 string 上没有一个字符和 pat 末尾字符匹配
        return false

    i ← i − large
    j ← patlastpos.
    while j ⩾ 0 and string[i] = pat[j]
        j ← j − 1
        i ← i − 1

    if j < 0
        return i + 1
    i ← i + max(delta_1(string[i]), delta_2(j))
```

其中 $large$ 起到多重作用，一是类似后面介绍的 Horspool 算法进行快速的坏字符跳转，二是辅助检测字符串搜索是否完成。

经过改进，比起原算法，在做**观察 1** 跳转时不必每次进行 $delta_2$ 的多余计算，使得在通常字符集下搜索字符串的性能有了明显的提升。

$delta_2$ 构建细节

引入

说起 $delta_2$ 的实现，发表在 1977 年 10 月的 *Communications of the ACM* 上的在 Boyer、Moor 的论文[1] 里只描述了这个静态表，并没有说明如何产生它。

而构造 $delta_2$ 的具体实现的讨论出现在 1977 年 6 月 Knuth、Morris、Pratt 在 *SIAM Journal on Computing* 上正式联合发表的 KMP 算法的论文[2-1] 里（这篇论文是个宝藏，除了 KMP，其中还提及了若干字符串搜索的算法构想和介绍，其中就包括了本文介绍的 BM 算法），听起来有点儿魔幻，嗯哼？这就不得不稍微介绍一点历史细节了：

1. 1969 年夏天 Morris 为某个大型机编写文本编辑器时利用有限自动机的理论发明了等价于 KMP 算法的字符串匹配算法，而他的算法由于过于复杂，被不理解他算法的同事当做 bug 修改得一团糟，哈哈。

2. 1970 年 KMP 中的「带头人」Knuth 在研究 Cook 的关于两路确定下推自动机（two-way deterministic pushdown automaton）的理论时受到启发，也独立发明了 KMP 算法的雏形，并把它展示给他的同事 Pratt，Pratt 改进了算法的数据结构。
3. 1974 年 Boyer、Moor 发现通过更快地跳过不可能匹配的文本能实现比 KMP 更快的字符串匹配，（Gosper 也独立地发现了这一点），而一个只有原始 $delta_1$ 定义的匹配算法是 BM 算法的最原始版本。
4. 1975 年 Boyer、Moor 提出了原始的 $delta_2$ 表，而这个版本的 $delta_2$ 表不仅不会对性能有所改善，还会在处理小字符表时拖累性能表现，而同年 MIT 人工智能实验室的 Kuipers 和我们熟悉的 Knuth 向他们提出了类似的关于 $delta_2$ 的改进建议，于是 Boyer、Moor 在论文的下一次修改中提到了这个建议，并提出一个用二维表代替 $delta_1$ 和 $delta_2$ 的想法。
5. 1976 年 1 月 Knuth 证明了关于 $delta_2$ 的改进会得到更好的性能，于是 Boyer、Moor 两人又一次修改了论文，得到了现在版本的 $delta_2$ 定义。同年 4 月，斯坦福的 Floyd 又发现了 Boyer、Moor 两人第一版本的公式中的严重的统计错误，并给出了现在版本的公式。
6. Standish 又为 Boyer、Moor 提供了现在的匹配算法的改进。
7. 1977 年 6 月 Knuth、Morris、Pratt 正式联合发表了 KMP 算法的论文，其中在提及比 KMP 表现更好的算法中提出了 $delta_2$ 的构建方式。（其中也感谢了 Boyer、Moor 对于证明线性定理（linearity theorem）提供的帮助）

这个 BM 算法的发展的故事，切实地向我们展示了团结、友谊、协作，以及谦虚好学不折不挠「在平凡中实现伟大」!

时间复杂度为 $O(n^3)$ 的构建 $delta_2$ 的朴素算法

在介绍 Knuth 的 $delta_2$ 构建算法之前，根据定义，我们会有一个原始、简单但有时可能已经够用的朴素算法（除非你需要构建长度成百上千的 pat）:

1. 对于 `[0, patlen)` 区间的每一个位置 `i`，根据 `subpat` 的长度确定其重现位置的区间，也就是 `[-subpatlen, i]`；
2. 可能的重现位置按照从右到左进行逐字符比较，寻找符合 $delta_2$ 要求的最右边 $subpat$ 的重现位置；
3. 最后别忘了令 $delta_2(lastpos) = 0$。

" 实现"

```rust
use std::cmp::PartialEq;

pub fn build_delta_2_table_naive(p: &[impl PartialEq]) -> Vec<usize> {
    let patlen = p.len();
    let lastpos = patlen - 1;
    let mut delta_2 = vec![];

    for i in 0..patlen {
        let subpatlen = (lastpos - i) as isize;

        if subpatlen == 0 {
            delta_2.push(0);
            break;
        }

        for j in (-subpatlen..(i + 1) as isize).rev() {
            // subpat 匹配
            if (j..j + subpatlen)
            .zip(i + 1..patlen)
            .all(|(rpr_index, subpat_index)| {
```

```
                if rpr_index < 0 {
                    return true;
                }

                if p[rpr_index as usize] == p[subpat_index] {
                    return true;
                }

                false
            })
            && (j <= 0 || p[(j - 1) as usize] != p[i])
            {
                delta_2.push((lastpos as isize - j) as usize);
                break;
            }
        }
    }

    delta_2
}
```

特别地，对 Rust 语言特性进行必要地解释，下不赘述：

- `usize` 和 `isize` 是和内存指针同字节数的无符号整数和有符号整数，在 32 位机上相当于 `u32` 和 `i32`，64 位机上相当于 `u64` 和 `i64`。
- 索引数组、向量、分片时使用 `usize` 类型的数字（因为在做内存上的随机访问并且下标不能为负值），所以如果需要处理负值要用 `isize`，而进行索引时又要用 `usize`，这就看到使用 `as` 关键字进行二者之间的显式转换。
- `impl PartialEq` 只是用作泛型，可以同时支持 `Unicode` 编码的 `char` 和二进制的 `u8`。

显见这是个时间复杂度为 $O(n^3)$ 的暴力算法。

时间复杂度为 $O(n)$ 的构建 $delta_2$ 的高效算法

下面我们要介绍的是时间复杂度为 $O(n)$，但是需要额外 $O(n)$ 空间复杂度的高效算法。

需要指出的是，虽然 1977 年 Knuth 提出了这个构建方法，然而他的原始版本的构建算法存在一个缺陷，实际上对于某些 pat 产生不出符合定义的 $delta_2$。

Rytter 在 1980 年 *SIAM Journal on Computing* 上发表的文章[3] 对此提出了修正，但是 Rytter 的这篇文章在细节上有些令人疑惑的地方，包括不限于：

- 示例中奇怪的 $delta_2$ 数值（笔者注：不清楚他依据的 $delta_2$ 是否和最终版 $delta_2$ 定义有微妙的差别，但我实在不想因为这事儿继续考古了 ）
- 明显的在复述 Knuth 算法时的笔误、算法上错误的缩进（可能是文章录入时的问题?）
- 奇妙的变量命名（考虑到那个时代的标签：`goto` 语句、汇编语言、大型机，随性的变量命名也很合理）

总之就是你绝对不想看他的那个修正算法的具体实现，不过好在他在用文字描述的时候比用伪代码清晰多了呢，现在我们用更清晰的思路和代码结构整理这么一个

$delta_2$ 的构建算法：

首先考虑到 $delta_2$ 的定义比较复杂，我们按照 $subpat$ 的重现位置进行分类，每一类进行单独处理，这是高效实现的关键思路。

按照重现位置由远到近，也就是偏移量由大到小，分成如下几类：

1. 整个 $subpat$ 重现位置完全在 pat 左边的，比如 `[(EYX)]ABYXCDEYX`，此时 $delta_2(j) = patlastpos \times 2 - j$；
2. $subpat$ 的重现有一部分在 pat 左边，有一部分是 pat 头部，比如 `[(XX)ABC]XXXABC`，此时 $patlastpos <$

$delta_2(j) < patlastpos \times 2 - j$；我们把 $subpat$ 完全在 pat 头部的的边际情况也归类在这里（当然根据实现也可以归类在下边），比如 [ABC]XXXABC，此时 $patlastpos = delta_2(j)$；

3. $subpat$ 的重现完全在 pat 中，比如 AB[YX]CDEYX，此时 $delta_2(j) < patlastpos$。

现在来讨论如何高效地计算这三种情况：

第一种情况

这是最简单的情况，只需一次遍历并且可以顺便将 $delta_2$ 初始化。

第二种情况

我们观察什么时候会出现 $subpat$ 的重现一部分在 pat 左边，一部分是 pat 的头部的情况呢？应该是 $subpat$ 的某个后缀和 pat 的某个前缀相等，

比如之前的例子：

j:	0 1 2 3 4 5 6 7 8
pat:	A B C X X X A B C

$delta_2(3)$ 的重现 [(XX)ABC]XXXABC，$subpat$ XXABC 的后缀与 pat 前缀中，有相等的，是 ABC。

说到这个拗口的前缀后缀相等，此时看过之前《前缀函数与 KMP 算法》的小伙伴们可能已经有所悟了，

没错，实际上对第二种和第三种情况的计算的关键都离不开前缀函数的计算和和应用

那么只要 j 取值使得 $subpat$ 包含这个相等的后缀，那么就可以得到第二种情况的 $subpat$ 的重现，对于例子，我们只需要使得 $j \leqslant 5$，

而当 $j = 5$ 时，就是 $subpat$ 完全在 pat 头部的边际情况。

可以计算此时的 $delta_2(j)$：

设此时这对相等的前后缀长度为 $prefixlen$，可知 $subpatlen = patlastpos - j$，那么在 pat 左边的部分长度是 $subpatlen - prefixlen$，

而 $rpr(j) = -(subpatlen - prefixlen)$，所以得到 $delta_2(j) = patlastpos - rpr(j) = patlastpos \times 2 - j - prefixlen$。

那么问题到这儿是不是结束了呢，并不是，因为可能会有多对相等的前缀和后缀，比如：

j:	0 1 2 3 4 5 6 7 8 9
pat:	A B A A B A A B A A

在 $j \leq 2$ 处有 ABAABAA，$2 < j \leq 5$ 处有 ABAA，在 $5 < j \leq 8$ 处有 A

之前提到的 Knuth 算法的缺陷就是只考虑了最长的那一对的情况。

所以实际上我们要考虑所有 $subpat$ 后缀与 pat 前缀相等的情况，其实也就是计算 pat 所有真后缀和真前缀相等的情况，然后按照长度从大到小，j 分区间计算

不同的 $delta_2(j)$。而如何得到 pat 所有相等的真前缀和真后缀长度呢？答案正是利用前缀函数和逆向运用计算前缀函数的状态转移方程：$j^{(n)} = \pi[j^{(n-1)} - 1]$。

从 $\pi[patlastpos]$ 开始作为最长一对的长度，然后通过逆向运行状态转移方程，得到下一个次长相等真前缀和真后缀的长度，直到这里我们就完成了第二种情况的 $delta_2$ 的计算。

第三种情况

$subpat$ 的重现不在别的地方，恰好就在 pat 中（不包括 pat 的头部）。

也就是按照从右到左的顺序，在 $pat[0 \ldots patlastpos - 1]$ 中寻找 $subpat$。

开启脑洞：既然是个字符串搜索的问题，那么当然可以用著名的 BM 算法本身解决，于是我们就得到了一个 BM 的递归实现的第三种情况，结束条件是 $patlen \leqslant 2$

而且根据 $delta_2$ 的定义，找到的 $subpat$ 的重现的下一个（也就是左边一个）字符和作为 pat 后缀的 $subpat$ 的下一个字符不能一样。

这就很好地启发了我们（起码很好地启发了 Knuth）使用类似于计算前缀函数的过程计算第三种情况，只不过是左右反过来的前缀函数：

- 两个指针分别指向子串的左端点和子串最长公共前后缀的「前缀」位置，从右向左移动，在发现指向的两个字符相等时继续移动，此时相当于「前缀」变大；
- 当两个字符不相等时，之前相等的部分就满足了 $delta_2$ 对重现的要求，并且回退指向「前缀」位置的指针直到构成新的字符相等或者出界。

同前缀函数一样，需要一个辅助数组，用于回退，可以使用之前计算第二种情况所生成的前缀数组的空间。

上述实现

```
use std::cmp::PartialEq;
use std::cmp::min;

pub fn build_delta_2_table_improved_minghu6(p: &[impl PartialEq]) -> Vec<usize>
{
    let patlen = p.len();
    let lastpos = patlen - 1;
    let mut delta_2 = Vec::with_capacity(patlen);

    // 第一种情况
    // delta_2[j] = lastpos * 2 - j
    for i in 0..patlen {
        delta_2.push(lastpos * 2 - i);
    }

    // 第二种情况
    // lastpos <= delata2[j] = lastpos * 2 - j
    let pi = compute_pi(p);  // 计算前缀函数
    let mut i = lastpos;
    let mut last_i = lastpos; // 只是为了初始化
    while pi[i] > 0 {
        let start;
        let end;

        if i == lastpos {
            start = 0;
        } else {
            start = patlen - pi[last_i];
        }

        end = patlen - pi[i];

        for j in start..end {
            delta_2[j] = lastpos * 2 - j - pi[i];
        }

        last_i = i;
        i = pi[i] - 1;
    }

    // 第三种情况
    // delata2[j] < lastpos
```

```
    let mut j = lastpos;
    let mut t = patlen;
    let mut f = pi;
    loop {
        f[j] = t;
        while t < patlen && p[j] != p[t] {
            delta_2[t] = min(delta_2[t], lastpos - 1 - j);  // 使用 min 函数保证后
面可能的回退不会覆盖前面的数据
            t = f[t];
        }

        t -= 1;
        if j == 0 {
            break;
        }
        j -= 1;
    }

    // 没有实际意义，只是为了完整定义
    delta_2[lastpos] = 0;

    delta_2
}
```

Galil 规则对多次匹配时最坏情况的改善

关于后缀匹配算法的多次匹配问题

之前的搜索算法只涉及到在 *string* 中寻找第一次 *pat* 匹配的情况，而对与在 *string* 中寻找全部 *pat* 的匹配的情况有很多不同的算法思路，这个问题的核心关注点是：

如何利用之前匹配成功的字符的信息，将最坏情况下的时间复杂度降为线性。

在原始的成功匹配后，简单的 *string* 的指针向后滑动 *patlen* 距离后重新开始后缀匹配，这会导致最坏情况下回到 $O(mn)$ 的时间复杂度（按照惯例，m 为 *patlen*，n 为 *stringlen*，下同）。

比如一个极端的例子：*pat*：`AAA`，*string*：`AAAAA`…。

对此 Knuth 提出来的一个方法是用一个「数量有限」的状态的集合来记录 *patlen* 长度的字符，这种算法保证 *string* 上每一个字符最多比较一次，但代价是这个「数量有限」的状态可能数目并不怎么「有限」，比如立刻就能想到它的上限是 2^m 个，但并不清楚它到底能变得多大，对于一个字符彼此不相等的 *pat*，需要 $\frac{1}{2}m^2+m$ 个状态。这个算法思路同在 1977 年 6 月的发表 KMP 论文[2-2] 里被介绍，也许在未来某个节点匹配代价很高但状态存储代价很低的新场景能重新得到应用，但对于现在简单的字符串匹配，这个设计并不特别合适。

而 Knuth 提出的另一个方法，嗯这里就不介绍了，同在上面的 Knuth 那篇「宝藏」论文里被介绍，缺点是除了过于复杂以外主要是构建辅助的数据结构需要的预处理时间太大：$O(qm)$

q 为全字符集的大小，而且 qm 前面的系数很大。

于是在这个背景下就有了下面介绍的思路简单，不需要额外预处理开销的 Galil 算法[4]。

Galil 规则

原理很简单，假定一个 *pat*，它是某个子串 U 重复 n 次构成的字符串 $UUUU\ldots$ 的前缀，那么我们称 U 为 *pat* 的一个周期。

比如，*pat*：`ABCABCAB`，是 `ABC` 的重复 `ABCABCABC` 的前缀，所以 `ABC` 的长度 3 就是这个 *pat* 的周期长度，也即 *pat* 满足 $pat[i]=pat[i+3]$。

当然其实 `ABCABC`… 也是 *pat* 的周期，但我们只关注最短的那个。

事实上，广义地讲，pat 至少拥有一个长度为它自身的周期。

我们规定这个最短的周期为 k，$k \leq patlen$。

在搜索过程中，假如我们的 pat 成功地完成了一次匹配，那么依照周期的特点，实际上只需将 $string$ 向后滑动 k 个字符，比较这 k 个字符是否对应相等就可以直接判断是否存在 pat 的又一个匹配。

而如何计算这个最短周期的长度呢，假如我们知道 pat 的相等的一对儿前缀 - 后缀，设它们的长度为 $prefixlen$，那么有 $pat[i] = pat[i + (patlen - prefixlen)]$。

而从数学的角度看这个公式，显然我们已经有了长度为 $patlen - prefixlen$ 的周期，而当我们知道 pat 最长的那一对相等的前缀 - 后缀，我们就得到了 pat 最短的周期。

而这个最长相等的前后缀长度，$\pi[patlastpos]$，在我们在计算 $delta_2$ 的时候已经计算过了，所以实际不需要额外的预处理时间和空间，就能改善后缀匹配算法最坏情况的时间复杂度为线性。

结合上述优化的 BM 的搜索算法最终实现

```rust
#[cfg(target_pointer_width = "64")]
const LARGE: usize = 10_000_000_000_000_000_000;

#[cfg(not(target_pointer_width = "64"))]
const LARGE: usize = 2_000_000_000;

pub struct BMPattern<'a> {
    pat_bytes: &'a [u8],
    delta_1: [usize; 256],
    delta_2: Vec<usize>,
    k: usize  // pat 的最短周期长度
}

impl<'a> BMPattern<'a> {
    // ...

    pub fn find_all(&self, string: &str) -> Vec<usize> {
        let mut result = vec![];
        let string_bytes = string.as_bytes();
        let stringlen = string_bytes.len();
        let patlen = self.pat_bytes.len();
        let pat_last_pos = patlen - 1;
        let mut string_index = pat_last_pos;
        let mut pat_index;
        let l0 =  patlen - self.k;
        let mut l = 0;

        while string_index < stringlen {
            let old_string_index = string_index;

            while string_index < stringlen {
                string_index += self.delta0(string_bytes[string_index]);
            }
            if string_index < LARGE {
                break;
            }

            string_index -= LARGE;
```

```
            // 如果 string_index 发生移动，意味着自从上次成功匹配后发生了至少一次的
失败匹配。
            // 此时需要将 Galil 规则的二次匹配的偏移量归零。
            if old_string_index < string_index {
                l = 0;
            }

            pat_index = pat_last_pos;

            while pat_index > l && string_bytes[string_index] == self.pat_bytes[
pat_index] {
                string_index -= 1;
                pat_index -= 1;
            }

            if pat_index == l && string_bytes[string_index] == self.pat_bytes[pa
t_index] {
                result.push(string_index - l);

                string_index += pat_last_pos - l + self.k;
                l = l0;
            } else {
                l = 0;
                string_index += max(
                    self.delta_1[string_bytes[string_index] as usize],
                    self.delta_2[pat_index],
                );
            }
        }

        result
    }
}
```

最坏情况在实践中性能影响

从实践的角度上说，理论上的最坏情况并不容易影响性能表现，哪怕是很小的只有 4 的字符集的随机文本测试下这种最坏情况的影响也小到难以观察。

也因此如果没有很好地设计，使用 Galil 法则会拖累一点平均的性能表现，但对于一些极端特殊的 *pat* 和 *string* 比如例子中的：*pat*：`AAA`，*string*：`AAAAA...`，Galil 规则的应用确实会使得性能表现提高数倍。

实践及后续

这个部分要讨论实践中的具体问题。

尽管前面给出了一些算法的实现代码，但并没有真正讨论过完整实现可能面临的一些「小问题」。

字符类型的考虑

在英语环境下，特别是上世纪 70 年代那个时候，人们考虑字符，默认的前提是它是 ASCII 码，通用字符表是容易通过一个固定大小的数组来确定的。$delta_1$ 的初始化只需要基于这个固定大小的数组。

而在尝试用 Rust 实现上述算法的时候，第一个遇到的问题是字符的问题，用一门很新的 2010 + 发展起来的语言来实现 1970 + 时代的算法，是一件很有意思的事情。

会观察到一些因时代发展而产生的一些变化，现代的编程语言，内生的 `char` 类型就是 Unicode，首先不可能用一个全字符集大小的数组来计算 $delta_1$，（其实也可以，只是完成一个 UTF-8 编码的字符串搜索可能需要额外 1GB 内存）但是可以使用哈希表来代替，同样是 $O(1)$ 的随机访问成本，毕竟哈希表是现代编程语言最基础的标准件之一了（哪怕是 Go 都有呢）。

但更严重的问题是 Unicode 使用的都是变长的字节编码方案，所以没办法直接按照字符个数计算跳转的字节数，当然，如果限定文本是简单的 ASCII 字符集，我们仍然可以按照 1 字符宽 1 字节来进行快速跳转，但这样的实现根本就没啥卵用！

在思考的过程中，首先的一个想法是直接将字符串转为按字符索引的向量数组，但这意味着啥都不用做就先有了一个遍历字符串的时间开销，和额外的大于等于字符串字节数的额外空间开销（因为 `char` 类型是 Unicode 字面值，采用固定 4 字节大小保存）。

于是我改进了思路，对于变长编码字符串，至少要完全遍历一遍，才能完成字符串匹配，那么在遍历过程中，我使用一个基于可增长的环结构实现的双头队列作为滑动窗口，保存过去 *patlen* 个字符，如果当前 *string* 的索引小于算法计算的跳转，就让循环空转直到等于算法要求的索引。实践证明，这个巧妙的设计使得在一般字符上搜索的 BM 算法的实现比暴力匹配算法还要慢一些。

于是挫折使我困惑，困惑使我思考，终于一束阳光照进了石头缝里：

1. 字符串匹配算法高效的关键在于字符索引的快速跳转
2. 字符索引一定要建立在等宽字符的基础上，

基于这两条原则思考，我就发现二进制字节本身：1 字节等宽、字符全集大小是 256，就是符合条件的完美字符！在这个基础上完成了一系列后缀匹配算法的高效实现。

Simplified Boyer–Moore 算法

BM 算法最复杂的地方就在于 $delta_2$ 表（通俗的名字是好后缀表）的构建，而实践中发现，在一般的字符集上的匹配性能主要依靠 $delta_1$ 表（通俗的名字是坏字符表），于是出现了仅仅使用 $delta_1$ 表的简化版 BM 算法，通常表现和完整版差距很小。

Boyer–Moore–Horspol 算法

Horspol 算法同样是基于坏字符的规则，不过是在与 *pat* 尾部对齐的字符上应用 $delta_1$，这个效果类似于前文对匹配算法的改进，所以它的通常表现优于原始 BM 和匹配算法改进后的 BM 差不多。

" 实现"

```rust
pub struct HorspoolPattern<'a> {
    pat_bytes: &'a [u8],
    bm_bc: [usize; 256],
}

impl<'a> HorspoolPattern<'a> {
    // ...
    pub fn find_all(&self, string: &str) -> Vec<usize> {
        let mut result = vec![];
        let string_bytes = string.as_bytes();
        let stringlen = string_bytes.len();
        let pat_last_pos = self.pat_bytes.len() - 1;
        let mut string_index = pat_last_pos;

        while string_index < stringlen {
            if &string_bytes[string_index-pat_last_pos..string_index+1] == self.
pat_bytes {
```

```
                result.push(string_index-pat_last_pos);
            }

            string_index += self.bm_bc[string_bytes[string_index] as usize];
        }

        result
    }
}
```

Boyer–Moore–Sunday 算法

Sunday 算法同样是利用坏字符规则，只不过相比 Horspool 它更进一步，直接关注 *pat* 尾部对齐的那个字符的下一个字符。

实现它只需要稍微修改一下 $delta_1$ 表，使得它相当于在 $patlen+1$ 长度的 *pat* 上进行构建。

Sunday 算法通常用作一般情况下实现最简单而且平均表现最好之一的实用算法，通常表现比 Horspool、BM 都要快一点。

" 实现"

```
pub struct SundayPattern<'a> {
    pat_bytes: &'a [u8],
    sunday_bc: [usize; 256],
}

impl<'a> SundayPattern<'a> {
    // ...
    fn build_sunday_bc(p: &'a [u8]) -> [usize; 256] {
        let mut sunday_bc_table = [p.len() + 1; 256];

        for i in 0..p.len() {
            sunday_bc_table[p[i] as usize] = p.len() - i;
        }

        sunday_bc_table
    }

    pub fn find_all(&self, string: &str) -> Vec<usize> {
        let mut result = vec![];
        let string_bytes = string.as_bytes();
        let pat_last_pos = self.pat_bytes.len() - 1;
        let stringlen = string_bytes.len();
        let mut string_index = pat_last_pos;

        while string_index < stringlen {
            if &string_bytes[string_index - pat_last_pos..string_index+1] == sel
f.pat_bytes {
                result.push(string_index - pat_last_pos);
            }

            if string_index + 1 == stringlen {
                break;
            }
```

```
            string_index += self.sunday_bc[string_bytes[string_index + 1] as usi
ze];
        }

        result
    }
}
```

BMHBNFS 算法

该算法结合了 Horspool 和 Sunday，是 CPython 实现 `stringlib` 模块时用到的 `find` 的算法[5]，似乎国内更有名气，不清楚为何叫这个名字，怎么就「AKA」了？

以下简称 B5S。

B5S 基本想法是：

1. 按照后缀匹配的思路，首先比较 $patlastpos$ 位置对应的字符是否相等，如果相等就比较 $0 \ldots patlastpos - 1$ 对应位置的字符是否相等，如果仍然相等，那么就发现一个匹配；
2. 如果任何一个阶段发生不匹配，就进入跳转阶段；
3. 在跳转阶段，首先观察 $patlastpos$ 位置的下一个字符是否在 pat 中，如果不在，直接向右滑动 $patlen + 1$，这是 Sunday 算法的最大利用；

   如果这个字符在 pat 中，对 $patlastpos$ 处的字符利用 $delta_1$ 进行 Horspool 跳转。

而这个算法根据时间节省还是空间节省为第一目标，会有差别巨大的不同实现。

时间节省版本

" 实现"

```
pub struct B5STimePattern<'a> {
    pat_bytes: &'a [u8],
    alphabet: [bool;256],
    bm_bc: [usize;256],
    k: usize
}

impl<'a> B5STimePattern<'a> {
    pub fn new(pat: &'a str) -> Self {
        assert_ne!(pat.len(), 0);

        let pat_bytes = pat.as_bytes();
        let (alphabet, bm_bc, k) = B5STimePattern::build(pat_bytes);

        B5STimePattern { pat_bytes, alphabet, bm_bc, k }
    }

    fn build(p: &'a [u8]) -> ([bool;256], [usize;256], usize)  {
        let mut alphabet = [false;256];
        let mut bm_bc = [p.len(); 256];
        let lastpos = p.len() - 1;

        for i in 0..lastpos {
            alphabet[p[i] as usize] = true;
            bm_bc[p[i] as usize] = lastpos - i;
```

```
        }

        alphabet[p[lastpos] as usize] = true;

        (alphabet, bm_bc, compute_k(p))
    }

    pub fn find_all(&self, string: &str) -> Vec<usize> {
        let mut result = vec![];
        let string_bytes = string.as_bytes();
        let pat_last_pos = self.pat_bytes.len() - 1;
        let patlen = self.pat_bytes.len();
        let stringlen = string_bytes.len();
        let mut string_index = pat_last_pos;
        let mut offset = pat_last_pos;
        let offset0 = self.k - 1;

        while string_index < stringlen {
            if string_bytes[string_index] == self.pat_bytes[pat_last_pos] {
                if &string_bytes[string_index-offset..string_index] == &self.pat
_bytes[pat_last_pos-offset..pat_last_pos] {
                    result.push(string_index-pat_last_pos);

                    offset = offset0;

                    // Galil rule
                    string_index += self.k;
                    continue;
                }
            }

            if string_index + 1 == stringlen {
                break;
            }

            offset = pat_last_pos;

            if !self.alphabet[string_bytes[string_index+1] as usize] {
                string_index += patlen + 1;  // sunday
            } else {
                string_index += self.bm_bc[string_bytes[string_index] as usize];
  // horspool
            }
        }

        result
    }
}
```

这个版本的 B5S 性能表现非常理想，是通常情况下，目前介绍的后缀匹配系列算法中最快的。

空间节省版本

这也是 CPython `stringlib` 中实现的版本，使用了两个整数近似取代了字符表和 $delta_1$ 的作用，极大地节省了空间：

用一个简单的 Bloom 过滤器取代字符表（alphabet）

" 实现"

```
pub struct BytesBloomFilter {
    mask: u64,
}
```

impl BytesBloomFilter {pub fn new() -> Self { SimpleBloomFilter { mask: 0, } }

```
fn insert(&mut self, byte: &u8) {
    (self.mask) |= 1u64 << (byte & 63);
}

fn contains(&self, char: &u8) -> bool {
    (self.mask & (1u64 << (byte & 63))) != 0
}
```

}

Bloom 过滤器设设计通过牺牲准确率（实际还有运行时间）来极大地节省存储空间的 `Set` 类型的数据结构，它的特点是会将集合中不存在的项误判为存在（False Positives，简称 FP），但不会把集合中存在的项判断为不存在（False Negatives，简称 FN），因此使用它可能会因为 FP 而没有得到最大的字符跳转，但不会因为 FN 而跳过本应匹配的字符。

理论上分析，上述「Bloom 过滤器」的实现在 *pat* 长度在 50 个 Bytes 时，FP 概率约为 0.5，而 *pat* 长度在 10 个 Bytes 时，FP 概率约为 0.15。

当然这不是一个标准的 Bloom 过滤器，首先它实际上没有使用一个真正的哈希函数，实际上它只是一个字符映射，就是将 0-255 的字节映射为它的前六位构成的数，考虑到我们在做内存上的字符搜索，这样的简化就非常重要，因为即使使用目前已知最快的非加密哈希算法 xxHash[6]，计算所需要的时间都要比它高一个数量级。

另外，按照计算，当 pat 在 30 字节以下时，为了达到最佳的 FP 概率，需要超过一个哈希函数，但这么做意义不大，因为用装有两个 `u128` 数字的数组就已经可以构建字符表的全字符集。

使用 $delta_1(pat[patlastpos])$ 代替整个 $delta_1$

观察 $delta_1$ 最常使用的地方就是后缀匹配时第一个字符就不匹配是最常见的不匹配的情况，于是令 `skip = delta1(pat[patlastpos])`，

在第一阶段不匹配时，直接向下滑动 `skip` 个字符；但当第二阶段不配时，因为缺乏整个 $delta_1$ 的信息，只能向下滑动一个字符。

" 实现"

```
pub struct B5SSpacePattern<'a> {
    pat_bytes: &'a [u8],
    alphabet: BytesBloomFilter,
    skip: usize,
}

impl<'a> B5SSpacePattern<'a> {
    pub fn new(pat: &'a str) -> Self {
        assert_ne!(pat.len(), 0);

        let pat_bytes = pat.as_bytes();
```

```
        let (alphabet, skip) = B5SSpacePattern::build(pat_bytes);

        B5SSpacePattern { pat_bytes, alphabet, skip}
    }

    fn build(p: &'a [u8]) -> (BytesBloomFilter, usize)  {
        let mut alphabet = BytesBloomFilter::new();
        let lastpos = p.len() - 1;
        let mut skip = p.len();

        for i in 0..p.len()-1 {
            alphabet.insert(&p[i]);

            if p[i] == p[lastpos] {
                skip = lastpos - i;
            }
        }

        alphabet.insert(&p[lastpos]);

        (alphabet, skip)
    }

    pub fn find_all(&self, string: &'a str) -> Vec<usize> {
        let mut result = vec![];
        let string_bytes = string.as_bytes();
        let pat_last_pos = self.pat_bytes.len() - 1;
        let patlen = self.pat_bytes.len();
        let stringlen = string_bytes.len();
        let mut string_index = pat_last_pos;

        while string_index < stringlen {
            if string_bytes[string_index] == self.pat_bytes[pat_last_pos] {
                if &string_bytes[string_index-pat_last_pos..string_index] == &se
lf.pat_bytes[..patlen-1] {
                    result.push(string_index-pat_last_pos);
                }

                if string_index + 1 == stringlen {
                    break;
                }

                if !self.alphabet.contains(&string_bytes[string_index+1]) {
                    string_index += patlen + 1;  // sunday
                } else {
                    string_index += self.skip;  // horspool
                }
            } else {
                if string_index + 1 == stringlen {
                    break;
                }

                if !self.alphabet.contains(&string_bytes[string_index+1]) {
                    string_index += patlen + 1;  // sunday
```

```
                } else {
                    string_index += 1;
                }
            }

        }

        result
    }
}
```

这个版本的算法相对于前面的后缀匹配算法不够快，但差距并不大，仍然比 KMP 这种快得多，特别是考虑到它极为优秀的空间复杂度：至多两个 `u64` 的整数，这确实是极为实用的适合作为标准库实现的一种算法！

理论分析

现在我们通过一个简单的概率模型来做一些绝不枯燥的理论上的分析，借此可以发现一些有趣而更深入的事实。

建立模型

想象一下，我们滑动字符串 pat 到某个新的位置，这个位置还没有完成匹配，我们可以用发现失配所需要的代价与发现失配后 pat 能够向下滑动的字符数的比值来衡量算法的平均性能表现。

假如这个代价是用对 $string$ 的引用来衡量，那么我们就可以知道平均每个字符需要多少次 $string$ 的引用，这是在理论上衡量算法表现的关键指标；

而如果这个代价是用机器指令衡量，那我们可以知道平均每个字符需要多少条机器指令；

当然也可以有其他的衡量方式，这并不影响什么，这里我们采用对 $string$ 的引用进行理论分析。

同时为我们的概率模型提出一个假设：pat，$string$ 中的每个字符是独立随机变量，它们出现的概率相等，为 p，p 取决于全字母表的大小。

显然，假如全字母表的大小为 q，则 $p = \frac{1}{q}$，例如假设我们之前基于字节的实现，在日常一般搜索时，可以近似为 $q = \frac{1}{256}$。

现在可以更准确地刻画这个比率，$rate(patlen, p)$：

$$\frac{\sum_{m=0}^{patlen-1} cost(m) \times prob(m)}{\sum_{m=0}^{patlen-1} prob(m) \times \sum_{k=1}^{patlen} skip(m,k) \times k}$$

其中，$cost(m)$ 为前面讨论到的在匹配成功了 m 个字符后失配时的代价：

$$cost(m) = m + 1$$

$prob(m)$ 为匹配成功 m 个字符后失配的概率（其中 $1 - p^{patlen}$ 排除掉 pat 全部匹配的情况）：

$$prob(m) = \frac{p^m(1-p)}{1 - p^{patlen}}$$

$skip(m,k)$ 为发生失配时 pat 向下滑动 k 个字符的概率，（这里的 k 如同前文讨论的 k 一样，为 pat 实际滑动距离，不包括指针从失配位置回退到 $patlastpos$ 位置的距离）。实际上所有字符串匹配算法的核就在于 $skip(m,k)$，下面我们会通过分析 $delta_1$ 和 $delta_2$ 来计算 BoyerMoore 算法的 $skip(m,k)$。

计算 BoyerMoore 算法的 $skip(m,k)$

$delta_1$

首先考虑 $delta_1$ 不起作用的情况，也就是发现失配字符在 pat 上重现的位置在已经匹配完的 m 个字符中，这种情况的概率 $probdelta\_1\_worthless$ 为：

$$probdelta1\_worthless(m) = 1-(1-p)^m$$

而对于 $delta_1$ 起作用的情况，可以根据 k 的范围分为四种情况进行讨论：

1. 当 $k=1$ 时：
   (a) 失配字符对应位置的下一个字符恰好等于失配字符；
   (b) 失配字符已经是 pat 右手起最后一个字符。
2. 当 $1<k<patlen-m$ 时，pat 在失配字符对应位置的左边还有与失配字符相等的字符，并且不满足情况 1；
3. 当 $k=patlen-m$ 时，pat 在失配字符对应位置左边找不到另一个与失配字符相等的字符，并且不满足情况 1，这时 pat 有最大可能的向下滑动距离；
4. 当 $k>patlen-m$ 时，显然对于 $delta_1$，这是不可能存在的情况。

于是有计算 $delta_1$ 的概率函数：

$$probdelta1(m,k) = \begin{cases} lcl(1-p)^m \times \begin{cases} lcl1 & \text{for } m+1 = patlen \\ p & \text{for } m+1 \neq patlen \end{cases} & \text{for } k=1 \\ (1-p)^{m+k-1} \times p & \text{for } 1<k<patlen-m \\ (1-p)^{patlen-1} & \text{for } k = patlen-m \\ 0 & \text{for } patlenpatlen-m<k\leqslant patlen \end{cases}$$

$delta_2$

对于 $delta_2$ 概率的计算，根据定义，首先计算某个 $subpat$ 的重现的概率，只要考虑该重现左边还有没有字符来提供额外的判断与失配字符是否相等的检查：

$$probpr(m,k) = \begin{cases} lcl(1-p)\times p^m & \text{for } 1\leqslant k<patlen-m \\ p^{patlen-k} & \text{for } patlen-m\leqslant k\leqslant patlen \end{cases}$$

于是 $delta_2(m,k)$ 就可以通过保证 $pr(m,k)$ 存在并且 k 更小的 $delta_2$ 不存在，来递归计算：

$$probdelta2(m,k) = probpr(m,k)(1-\sum_{n=1}^{k-1} probdelta2(m,n))$$

汇总

前面已经独立讨论了 $delta_1$，$delta_2$ 的概率函数，不过还需要额外考虑一下这两个概率函数之间相互影响的情况，虽然只是一个很少数的情况：

当 $delta_2$ 计算的 k 为 1 的时候，根据 $delta_2$ 定义我们就知道 $pat[-(m+1)] = pat[-m] = pat[-(m-1)] \ldots pat[-1]$，（$pat[-n]$ 表示 pat 的倒数第 n 个字符，下同）。而这种情况已经排除了 $delta_1$ 不起作用的情况，因为当如前文讨论的，$delta_1$ 不起作用要求与失配字符 $pat[-(m+1)]$ 相等的字符出现在 $pat[-m]\ldots pat[-1]$ 中，这就产生了不可能在倒数 $m+1$ 个字符上失配的矛盾。

因此针对 $delta_1$ 不起作用的情况需要一个稍微修改过的 $delta_2$ 概率函数：

$$probdelta2'(m,k) = \begin{cases} 0 & \text{for } k=1 \\ probpr(m,k)\left(1-\sum_{n=2}^{k-1} probdelta2'(m,n)\right) & \text{for } 1\leqslant k\leqslant patlen \end{cases}$$

于是通过组合 $delta_1$ 和 $delta_2$ 起作用的情况，我们就得到了 BoyerMoore 算法的 $skip$ 概率函数：

$$skip(m,k) = \begin{cases} probdelta1(m,1) \times probdelta2(m,1) & \text{for } k=1 \\ prodelta1\backslash\_worthless(m) \times probdelta2'(m,1) & \\ + probdelta1(m,k) \times \sum_{n=1}^{k-1} probdelta2(m,n) & \\ + probdelta2(m,k) \times \sum_{n=1}^{k-1} probdelta1(m,n) & \\ + probdelta1(m,k) \times probdelta2(m,k) & \text{for } 1 < k \leqslant patlen \end{cases}$$

分析比较

为了结构清晰、书写简单、演示方便，我们使用 Python 平台的 Lisp 方言 Hy 来进行实际计算：

myprob.hy

```
(require [hy.contrib.sequences [defseq seq]])

(import [hy.contrib.sequences [Sequence end-sequence]])
(import [hy.models [HySymbol]])


(defmacro simplify-probfn [patlen p probfn-list]
    "(prob-xxx patlen p m k) -> (prob-xxx-s m k)"
    (lfor probfn probfn-list
        [(setv simplified-probfn-symbol (HySymbol (.format "{}-s" (name probfn))
))
        `(defn ~simplified-probfn-symbol [&rest args] (~probfn patlen p #*args))
]))


(defn map-sum [range-args func]
    (setv [start end] range-args)
    (-> func (map (range start (inc end))) sum))


(defn cost [m] (+ m 1))


(defn prob-m [patlen p m]
    (/
        (* (** p m) (- 1 p))
        (- 1 (** p patlen))))


(defn prob-delta1 [patlen p m k]
    (cond [(= 1 k) (* (** (- 1 p) m)
                      (if (= (inc m) patlen) 1 p))]
          [(< k (- patlen m)) (* p (** (- 1 p) (dec (+ k m))))]
          [(= k (- patlen m)) (** (- 1 p) (dec patlen))]
          [(> k (- patlen m)) 0]))


(defn prob-delta1-worthless [p m] (- 1 (** (- 1 p) m)))
```

```
(defn prob-pr [patlen p m k] (if (< patlen (+ m k))
                                (* (- 1 p) (** p m))
                                (** p (- patlen k))))


(defn prob-delta2 [patlen p m]
    "prob-delta2(_, k) = prob-delta2-seq[k]"
    (defseq prob-delta2-seq [n]
        (cond [(< n 1) 0]
              [(= n 1) (prob-pr patlen p m 1)]
              [(> n 1) (* (prob-pr patlen p m n) (- 1 (sum (take n prob-delta2-s
eq))))]))
    prob-delta2-seq)


(defn prob-delta2-1 [patlen p m]
    (defseq prob-delta2-1-seq [n]
        (cond [(< n 2) 0]
              [(>= n 2) (* (prob-pr patlen p m n) (- 1 (sum (take n prob-delta2-
1-seq))))]))
    prob-delta2-1-seq)


(defn skip [patlen p m k prob-delta2-seq prob-delta2-1-seq]
    (simplify-probfn patlen p [prob-delta1])
    (if (= k 1) (* (prob-delta1-s m 1) (get prob-delta2-seq 1))
        (sum [(* (prob-delta1-worthless p m) (get prob-delta2-1-seq k))
              (* (prob-delta1-s m k) (sum (take k prob-delta2-seq)))
              (* (get prob-delta2-seq k) (map-sum [1 (- k 1)]
                                                  (fn [n] (prob-delta1-s m n))))
              (* (prob-delta1-s m k) (get prob-delta2-seq k))])))


(defn bm-rate [patlen p]
    (simplify-probfn patlen p [prob-m prob-delta2 prob-delta2-1 skip])
    (/
        (map-sum [0 (dec patlen)]
            (fn [m] (* (cost m) (prob-m-s m))))

        (map-sum [0 (dec patlen)]
            (fn [m]
                (setv prob-delta2-seq (prob-delta2-s m)
                      prob-delta2-1-seq (prob-delta2-1-s m))
                (* (prob-m-s m) (map-sum [1 patlen]
                                    (fn [k] (* k (skip-s m k prob-delta2-seq pro
b-delta2-1-seq)))))))))
```

并且为了进行比较，还额外计算了简化 BM 算法：

myprob.hy

```
(defn simplified-bm-skip [patlen p m k]
    (simplify-probfn patlen p [prob-delta1])
    (if (= k 1) (+ (prob-delta1-s m 1) (prob-delta1-worthless p m))
        (prob-delta1-s m k)))
```

```
(defn sbm-rate [patlen p]
    (simplify-probfn patlen p [prob-m simplified-bm-skip])
    (/
        (map-sum [0 (dec patlen)]
            (fn [m] (* (cost m) (prob-m-s m))))

        (map-sum [0 (dec patlen)]
            (fn [m] (* (prob-m-s m) (map-sum [1 patlen]
                                        (fn [k] (* k (simplified-bm-skip-s m k))
)))))))
```

和 KMP 算法:

myprob.hy

```
(defn getone [&rest body &kwonly [default None]]
  (try
    (get #*body)
    (except [_ [IndexError KeyError TypeError]]
      default)))


(defn prob-pi [patlen p]
    "prob-pi-s(m, l) = prob-pi-seq[m][l]"
    (defseq prob-pi-seq [n]
        (cond [(= n 0) []]
              [(= n 1) [1]]
              [(= n 2) [(- 1 p) p]]
              [(> n 2) (lfor i (range n)
                            (+
                                (* (getone prob-pi-seq (dec n) i :default 0) (-
1 p))
                                (* (get prob-pi-seq (dec n) (dec i)))))]))
    prob-pi-seq)


(defn skip [m k prob-pi-seq]
    (if (= m 0) (if (= k 1) 1 0)
        (get prob-pi-seq m (- m k))))


(defn at-least-1 [n]
    (if (< n 1) 1 n))


(defn kmp-rate [patlen p]
    (simplify-probfn patlen p [prob-m prob-pi])
    (setv prob-pi-seq (prob-pi-s))
    (/
        (map-sum [0 (dec patlen)]
            (fn [m] (* (cost m) (prob-m-s m))))

        (map-sum [0 (dec patlen)]
            (fn [m] (* (prob-m-s m) (map-sum [1 (at-least-1 (dec m))]
                                        (fn [k] (* k (skip m k prob-pi-seq))))))
```

```
)))
```

然后我们就可以通过 Python 上的 `plotnine` 图形包看一下计算的数据（并用高斯过程回归拟合曲线）：

```python
import pandas as pd
from plotnine import *
import hy

from my_prob import bm_rate, sbm_rate, kmp_rate


theme_update(text=element_text(family="SimHei"))

def plot(p, title, N=30):
    model_range = range(1, N+1)
    data = {'rate':[], 'alg':[], 'patlen':[]}
    categories_list = [(bm_rate, 'BoyerMoore'),
                       (sbm_rate, 'S BoyerMoore'),
                       (kmp_rate, 'KMP'),
                       (lambda patlen, p: 1/patlen, '$\\frac{1}{patlen}$')]

    for alg_fun, label in categories_list:
        data['rate'].extend([alg_fun(patlen, p) for patlen in model_range])
        data['alg'].extend([label for _ in model_range])
        data['patlen'].extend(model_range)

    df = pd.DataFrame(data)

    return (ggplot(df, aes(x='patlen', y='rate', color='alg'))
            + geom_point()
            + geom_smooth(method='gpr')
            + labs(color='Algs', title=title, x='$patlen$', y='$\\frac{cost}{ski
p}$'))
```

`plot(1/256, '$p= \\frac{1}{256}$')`：

观察这个图像，令人印象深刻的首先就是抬头的一条大兰线，几乎笔直地画出了算法性能的下限，不愧是 KMP 算法，$O(n)$ 的时间复杂度，一看就很真实。

接着会发现 BoyerMoore 算法与简化版 BoyerMoore 算法高度重叠的这条红绿紫曲线，同时也是 $\frac{1}{patlen}$，这就是在一般字符集下随机文本搜索能达到的 $O(\frac{n}{m})$ 的强力算法吗？

另外此时可以绝大多数的字符跳转依靠 $delta_1$（比 $delta_2$ 高几个数量），这也是基于 $delta_1$ 表的 BM 变种算法最佳的应用场景！

接着我们可以看一下在经典的小字符集，比如在 DNA {A, C, T, G} 碱基对序列中算法的性能表现（`plot(1/4, '$p= \\frac{1}{4}$')`）：

曲线出现了明显的分化，当然 KMP 还是一如既往地稳定，如果此时在测试中监控一下一下 $delta_1$ 表和 $delta_2$ 表作用情况会发现：$delta_2$ 起作用的次数超过了 $delta_1$，而且 $delta_2$ 贡献的跳过字符数更是远超 $delta_1$，思考下，这件事其实也很好理解。

总结一下，通过概率模型的计算，一方面看到了在较大的字符集，比如日常搜索的过程中 BoyerMoore 系列算法的优越表现，其中主要依赖 $delta_1$ 表实现字符跳转；另一方面，在较小的字符集里，$delta_1$ 的作用下降，而 $delta_2$ 的作用得到了体现。如果有一定富裕空间的情况下，使用完整的空间复杂度为 $O(m)$ 的 BoyerMoore 算法应该是一种适用各种情况、综合表现都很优异的算法选择。

引用

[1] 1977 年 Boyer–Moore 算法论文

[2] 1977 年 KMP 算法论文 [2-1] [2-2]

[3] 1980 年 Rytter 纠正 Knuth 的论文

[4] 1979 年介绍 Galil 算法的论文

[5] B5S 算法的介绍

[6] xxHash

8.9 Z 函数（扩展 KMP）

Authors: LeoJacob, Marcythm, minghu6

约定：字符串下标以 0 为起点。

定义

对于一个长度为 n 的字符串 s，定义函数 $z[i]$ 表示 s 和 $s[i,n-1]$（即以 $s[i]$ 开头的后缀）的最长公共前缀（LCP）的长度，则 z 被称为 s 的 **Z 函数**。特别地，$z[0]=0$。

国外一般将计算该数组的算法称为 **Z Algorithm**，而国内则称其为**扩展 KMP**。

这篇文章介绍在 $O(n)$ 时间复杂度内计算 Z 函数的算法以及其各种应用。

解释

下面若干样例展示了对于不同字符串的 Z 函数：

- $z(\texttt{aaaaa})=[0,4,3,2,1]$
- $z(\texttt{aaabaab})=[0,2,1,0,2,1,0]$
- $z(\texttt{abacaba})=[0,0,1,0,3,0,1]$

朴素算法

Z 函数的朴素算法复杂度为 $O(n^2)$：

" 实现"

```
vector<int> z_function_trivial(string s) {
  int n = (int)s.length();
  vector<int> z(n);
  for (int i = 1; i < n; ++i)
    while (i + z[i] < n && s[z[i]] == s[i + z[i]]) ++z[i];
  return z;
```

```
}

def z_function_trivial(s):
    n = len(s)
    z = [0] * n
    for i in range(1, n):
        while i + z[i] < n and s[z[i]] == s[i + z[i]]:
            z[i] += 1
    return z
```

线性算法

如同大多数字符串主题所介绍的算法，其关键在于，运用自动机的思想寻找限制条件下的状态转移函数，使得可以借助之前的状态来加速计算新的状态。

在该算法中，我们从 1 到 $n-1$ 顺次计算 $z[i]$ 的值（$z[0]=0$）。在计算 $z[i]$ 的过程中，我们会利用已经计算好的 $z[0],\dots,z[i-1]$。

对于 i，我们称区间 $[i,i+z[i]-1]$ 是 i 的**匹配段**，也可以叫 Z-box。

算法的过程中我们维护右端点最靠右的匹配段。为了方便，记作 $[l,r]$。根据定义，$s[l,r]$ 是 s 的前缀。在计算 $z[i]$ 时我们保证 $l\le i$。初始时 $l=r=0$。

在计算 $z[i]$ 的过程中：

- 如果 $i\le r$，那么根据 $[l,r]$ 的定义有 $s[i,r]=s[i-l,r-l]$，因此 $z[i]\ge\min(z[i-l],r-i+1)$。这时：
  - 若 $z[i-l]<r-i+1$，则 $z[i]=z[i-l]$。
  - 否则 $z[i-l]\ge r-i+1$，这时我们令 $z[i]=r-i+1$，然后暴力枚举下一个字符扩展 $z[i]$ 直到不能扩展为止。
- 如果 $i>r$，那么我们直接按照朴素算法，从 $s[i]$ 开始比较，暴力求出 $z[i]$。
- 在求出 $z[i]$ 后，如果 $i+z[i]-1>r$，我们就需要更新 $[l,r]$，即令 $l=i,r=i+z[i]-1$。

可以访问 这个网站[1] 来看 Z 函数的模拟过程。

实现

```cpp
vector<int> z_function(string s) {
  int n = (int)s.length();
  vector<int> z(n);
  for (int i = 1, l = 0, r = 0; i < n; ++i) {
    if (i <= r && z[i - l] < r - i + 1) {
      z[i] = z[i - l];
    } else {
      z[i] = max(0, r - i + 1);
      while (i + z[i] < n && s[z[i]] == s[i + z[i]]) ++z[i];
    }
    if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
  }
  return z;
}
```

```python
def z_function(s):
    n = len(s)
    z = [0] * n
    l, r = 0, 0
    for i in range(1, n):
```

```
        if i <= r and z[i - l] < r - i + 1:
            z[i] = z[i - l]
        else:
            z[i] = max(0, r - i + 1)
            while i + z[i] < n and s[z[i]] == s[i + z[i]]:
                z[i] += 1
        if i + z[i] - 1 > r:
            l = i
            r = i + z[i] - 1
    return z
```

复杂度分析

对于内层 while 循环，每次执行都会使得 r 向后移至少 1 位，而 $r < n - 1$，所以总共只会执行 n 次。

对于外层循环，只有一遍线性遍历。

总复杂度为 $O(n)$。

应用

我们现在来考虑在若干具体情况下 Z 函数的应用。

这些应用在很大程度上同 前缀函数 的应用类似。

匹配所有子串

为了避免混淆，我们将 t 称作**文本**，将 p 称作**模式**。所给出的问题是：寻找在文本 t 中模式 p 的所有出现（occurrence）。

为了解决该问题，我们构造一个新的字符串 $s = p + \diamond + t$，也即我们将 p 和 t 连接在一起，但是在中间放置了一个分割字符 $\diamond$（我们将如此选取 $\diamond$ 使得其必定不出现在 p 和 t 中）。

首先计算 s 的 Z 函数。接下来，对于在区间 $[0, |t| - 1]$ 中的任意 i，我们考虑以 $t[i]$ 为开头的后缀在 s 中的 Z 函数值 $k = z[i + |p| + 1]$。如果 $k = |p|$，那么我们知道有一个 p 的出现位于 t 的第 i 个位置，否则没有 p 的出现位于 t 的第 i 个位置。

其时间复杂度（同时也是其空间复杂度）为 $O(|t| + |p|)$。

本质不同子串数

给定一个长度为 n 的字符串 s，计算 s 的本质不同子串的数目。

考虑计算增量，即在知道当前 s 的本质不同子串数的情况下，计算出在 s 末尾添加一个字符后的本质不同子串数。

令 k 为当前 s 的本质不同子串数。我们添加一个新的字符 c 至 s 的末尾。显然，会出现一些以 c 结尾的新的子串（以 c 结尾且之前未出现过的子串）。

设串 t 是 $s + c$ 的反串（反串指将原字符串的字符倒序排列形成的字符串）。我们的任务是计算有多少 t 的前缀未在 t 的其他地方出现。考虑计算 t 的 Z 函数并找到其最大值 $z_{\max}$。则 t 的长度小于等于于 $z_{\max}$ 的前缀的反串在 s 中是已经出现过的以 c 结尾的子串。

所以，将字符 c 添加至 s 后新出现的子串数目为 $|t| - z_{\max}$。

算法时间复杂度为 $O(n^2)$。

值得注意的是，我们可以用同样的方法在 $O(n)$ 时间内，重新计算在端点处添加一个字符或者删除一个字符（从尾或者头）后的本质不同子串数目。

字符串整周期

给定一个长度为 n 的字符串 s，找到其最短的整周期，即寻找一个最短的字符串 t，使得 s 可以被若干个 t 拼接而成的字符串表示。

考虑计算 s 的 Z 函数，则其整周期的长度为最小的 n 的因数 i，满足 $i+z[i]=n$。

该事实的证明同应用 前缀函数 的证明一样。

练习题目

- CF126B Password[2]
- UVa # 455 Periodic Strings[3]
- UVa # 11022 String Factoring[4]
- UVa 11475 - Extend to Palindrome[5]
- LA 6439 - Pasti Pas![6]
- Codechef - Chef and Strings[7]
- Codeforces - Prefixes and Suffixes[8]
- Leetcode 2223 - Sum of Scores of Built Strings[9]

参考资料与注释

[1] 这个网站

[2] CF126B Password

[3] UVa # 455 Periodic Strings

[4] UVa # 11022 String Factoring

[5] UVa 11475 - Extend to Palindrome

[6] LA 6439 - Pasti Pas!

[7] Codechef - Chef and Strings

[8] Codeforces - Prefixes and Suffixes

[9] Leetcode 2223 - Sum of Scores of Built Strings

[10] Z-

[11] Z-function and its calculation

8.10 自动机

OI 中所说的「自动机」一般都指「确定有限状态自动机」。

自动机是 OI、计算机科学中被广泛使用的一个数学模型，其思想在许多字符串算法中都有涉及，因此推荐在学习一些字符串算法（KMP、AC 自动机、SAM）前先完成自动机的学习。学习自动机有助于理解上述算法。

前置知识

- 基础图论。

自动机入门

首先理解一下自动机是用来干什么的：自动机是一个对信号序列进行判定的数学模型。

这句话涉及到的名词比较多，逐一解释一下。「信号序列」是指一连串有顺序的信号，例如字符串从前到后的每一个字符、数组从 1 到 n 的每一个数、数从高到低的每一位等。「判定」是指针对某一个命题给出或真或假的回答。有时我们需要对一个信号序列进行判定。一个简单的例子就是判定一个二进制数是奇数还是偶数，较复杂的例子例如判定一个字符串是否回文，判定一个字符串是不是某个特定字符串的子序列等等。

自动机的工作原理和地图很类似。假设你在你家，然后你从你家到学校，按顺序经过了很多路口。每个路口都有岔路，而你在所有这些路口的选择就构成了一个序列。

例如，你的选择序列是「家门 -> 右拐 -> 萍水西街 -> 尚园街 -> 古墩路 -> 地铁站 -> 下宁桥」，那你按顺序经过的路口可能是「家 -> 家门口 -> 萍水西街竞舟北路口 -> 萍水西街尚圆街路口 -> 尚园街古墩路口 -> 古墩路中 -> 三坝地铁站 -> 下宁桥地铁站」。可以发现，上学的选择序列不止这一个。同样要去地铁站，你还可以从竞舟北路绕道，或者横穿文鼎苑抄近路。

而我们如果找到一个选择序列，就可以在地图上比划出这个选择序列能不能去学校。比如，如果一个选择序列是「家门 -> 右拐 -> 萍水西街 -> 尚园街 -> 古墩路 -> 地铁站 -> 钱江路 -> 四号线站台 -> 联庄」，那么它就不会带你去同一个学校，但是仍旧可能是一个可被接受的序列，因为目标地点可能不止一个。

也就是说，我们通过这个地图和一组目的地，将信号序列分成了三类，一类是无法识别的信号序列（例如「家门 -> ???」），一类是能去学校的信号序列，另一类是不能的信号序列。我们将所有合法的信号序列分成了两类，完成了一个判定问题。

既然自动机是一个数学模型，那么显然不可能是一张地图。对地图进行抽象之后，可以简化为一个有向图。因此，自动机的结构就是一张有向图。

而自动机的工作方式和流程图类似，不同的是：自动机的每一个结点都是一个判定结点；自动机的结点只是一个单纯的状态而非任务；自动机的边可以接受多种字符（不局限于 T 或 F）。

例如，完成「判断一个二进制数是不是偶数」的自动机如下：

从起始结点开始，从高到低接受这个数的二进制序列，然后看最终停在哪里。如果最终停在红圈结点，则是偶数，否则不是。

如果需要判定一个有限的信号序列和另外一个信号序列的关系（例如另一个信号序列是不是某个信号序列的子序列），那么常用的方法是针对那个有限的信号序列构建一个自动机。这个在学习 KMP 的时候会讲到。

需要注意的是，自动机只是一个**数学模型**，而**不是算法**，也**不是数据结构**。实现同一个自动机的方法有很多种，可能会有不一样的时空复杂度。

接下来你可以选择继续看自动机的形式化定义，也可以去学习 KMP、AC 自动机 或 SAM。

形式化定义

一个**确定有限状态自动机（DFA）**由以下五部分构成：

1. **字符集**（Σ），该自动机只能输入这些字符。

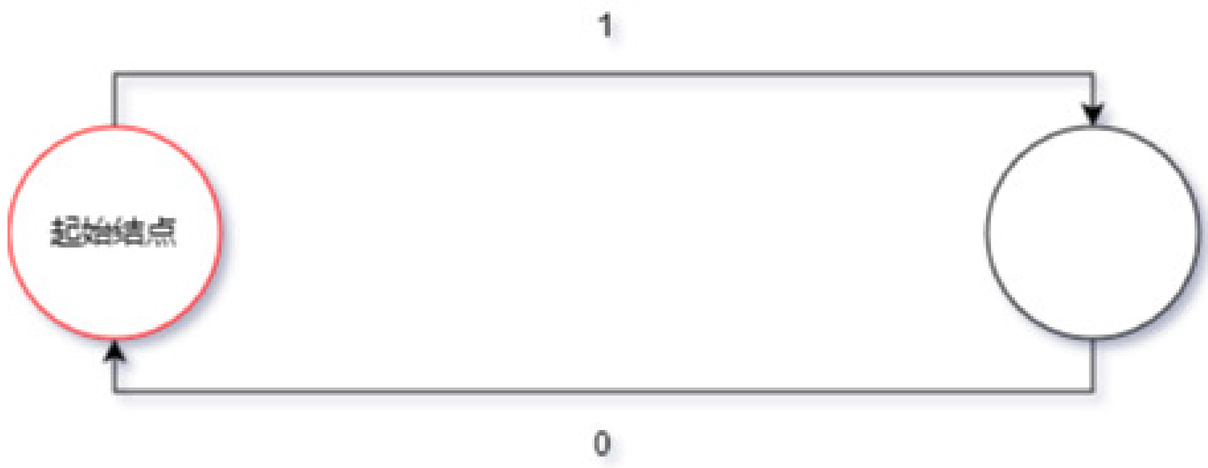


图 8.2 example automaton

2. **状态集合**（Q）。如果把一个 DFA 看成一张有向图，那么 DFA 中的状态就相当于图上的顶点。
3. **起始状态**（$start$），$start \in Q$，是一个特殊的状态。起始状态一般用 s 表示，为了避免混淆，本文中使用 $start$。
4. **接受状态集合**（F），$F \subseteq Q$，是一组特殊的状态。
5. **转移函数**（δ），δ 是一个接受两个参数返回一个值的函数，其中第一个参数和返回值都是一个状态，第二个参数是字符集中的一个字符。如果把一个 DFA 看成一张有向图，那么 DFA 中的转移函数就相当于顶点间的边，而每条边上都有一个字符。

DFA 的作用就是识别字符串，一个自动机 A，若它能识别（接受）字符串 S，那么 $A(S) = \mathrm{True}$，否则 $A(S) = \mathrm{False}$。

当一个 DFA 读入一个字符串时，从初始状态起按照转移函数一个一个字符地转移。如果读入完一个字符串的所有字符后位于一个接受状态，那么我们称这个 DFA **接受**这个字符串，反之我们称这个 DFA **不接受**这个字符串。

如果一个状态 v 没有字符 c 的转移，那么我们令 $\delta(v,c) = \mathrm{null}$，而 null 只能转移到 null，且 null 不属于接受状态集合。无法转移到任何一个接受状态的状态都可以视作 null，或者说，null 代指所有无法转移到任何一个接受状态的状态。

我们扩展定义转移函数 δ，令其第二个参数可以接收一个字符串：$\delta(v,s) = \delta(\delta(v,s[1]),s[2..|s|])$，扩展后的转移函数就可以表示从一个状态起接收一个字符串后转移到的状态。那么，$A(s) = [\delta(start,s) \in F]$。

如，一个接受且仅接受字符串"a", "ab", "aac" 的 DFA：

OI 中常用的自动机

字典树

字典树 是大部分 OIer 接触到的第一个自动机，接受且仅接受指定的字符串集合中的元素。

转移函数就是 Trie 上的边，接受状态是将每个字符串插入到 Trie 时到达的那个状态。

KMP 自动机

KMP 算法 可以视作自动机，基于字符串 s 的 KMP 自动机接受且仅接受以 s 为后缀的字符串，其接受状态为 $|s|$。

转移函数：

$$\delta(i,c) = \begin{cases} i+1 & s[i+1] = c \\ 0 & s[1] \neq c \land i = 0 \\ \delta(\pi(i),c) & s[i+1] \neq c \land i > 0 \end{cases}$$

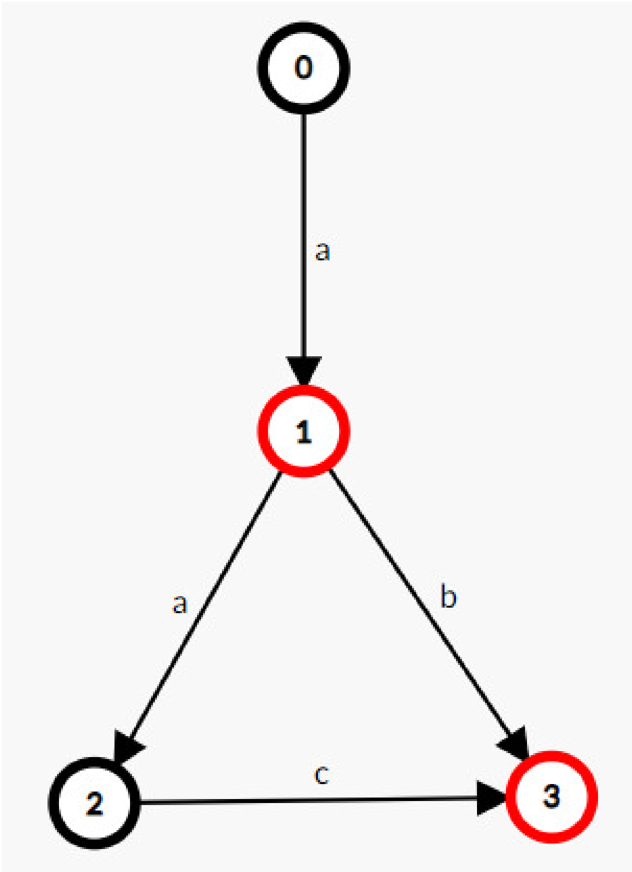


图 8.3

AC 自动机

AC 自动机 接受且仅接受以指定的字符串集合中的某个元素为后缀的字符串。也就是 Trie + KMP。

后缀自动机

后缀自动机 接受且仅接受指定字符串的后缀。

广义后缀自动机

广义后缀自动机 接受且仅接受指定的字符串集合中的某个元素的后缀。也就是 Trie + SAM。

广义 SAM 与 SAM 的关系就是 AC 自动机与 KMP 自动机的关系。

回文自动机

回文自动机 比较特殊，它不能非常方便地定义为自动机。

如果需要定义的话，它接受且仅接受某个字符串的所有回文子串的**中心及右半部分**。

「中心及右边部分」在奇回文串中就是字面意思，在偶回文串中定义为一个特殊字符加上右边部分。这个定义看起来很奇怪，但它能让 PAM 真正成为一个自动机，而不仅是两棵树。

序列自动机

序列自动机 接受且仅接受指定字符串的子序列。

后缀链接

由于自动机和匹配有着密不可分的关系，而匹配的一个基本思想是「这个串不行，就试试它的后缀可不可以」，所以在很多自动机（KMP、AC 自动机、SAM、PAM）中，都有后缀链接的概念。

一个状态会对应若干字符串，而这个状态的后缀链接，是在自动机上的、是这些字符串的公共真后缀的字符串中，最长的那一个。

一般来讲，后缀链接会形成一棵树，并且不同自动机的后缀链接树有着一些相同的性质，学习时可以加以注意。

扩展阅读

在计算复杂性领域中，自动机是一个经典的模型。并且，自动机与正则语言有着密不可分的关系。

如果对相关内容感兴趣的话，推荐阅读博客 计算复杂性 （1）Warming Up: 自动机模型[1]。

参考资料与注释

[1] 计算复杂性（1）Warming Up: 自动机模型

8.11 AC 自动机

Authors: iamtwz, Marcythm, 383494, abc1763613206, aofall, Chrogeek, CoelacanthusHex, Dafenghh, DanJoshua, Entertainer, GavinZhengOI, Gesrua, Henry-ZHR, Ir1d, kenlig, ksyx, lyccrius, Menci, opsiff, orzAtalod, ouuan, partychicken, Persdre, qq2964, Ruakker, shuzhouliu, sshwy, StudyingFather, szdytom, Tiphereth-A, Xeonacid, ZXyaang, rickyxrc

AC 自动机是**以 Trie 的结构为基础**，结合 **KMP 的思想**建立的自动机，用于解决多模式匹配等任务。

" 引入"

很多人在第一次看到这个东西的时侯是非常兴奋的。不过这个自动机叫作 `Automaton`，不是 `Automation`，这里的 AC 也不是 Accepted，而是 Aho–Corasick（Alfred V. Aho, Margaret J. Corasick. 1975），让萌新失望啦。切入正题。似乎在初学自动机相关的内容时，许多人难以建立对自动机的初步印象，尤其是在自学的时侯。而这篇文章就是为你们打造的。笔者在自学 AC 自动机后花费两天时间制作若干的 gif，呈现出一个相对直观的自动机形态。尽管这个图似乎不太可读，但这绝对是在作者自学的时侯，画得最认真的 gif 了。另外有些小伙伴问这个 gif 拿什么画的。笔者用 Windows 画图软件制作。

解释

简单来说，建立一个 AC 自动机有两个步骤：

1. 基础的 Trie 结构：将所有的模式串构成一棵 Trie。
2. KMP 的思想：对 Trie 树上所有的结点构造失配指针。

然后就可以利用它进行多模式匹配了。

字典树构建

AC 自动机在初始时会将若干个模式串丢到一个 Trie 里，然后在 Trie 上建立 AC 自动机。这个 Trie 就是普通的 Trie，该怎么建怎么建。

这里需要仔细解释一下 Trie 的结点的含义，尽管这很小儿科，但在之后的理解中极其重要。Trie 中的结点表示的是某个模式串的前缀。我们在后文也将其称作状态。一个结点表示一个状态，Trie 的边就是状态的转移。

形式化地说，对于若干个模式串 $s_1, s_2 \dots s_n$，将它们构建一棵字典树后的所有状态的集合记作 Q。

失配指针

AC 自动机利用一个 fail 指针来辅助多模式串的匹配。

状态 u 的 fail 指针指向另一个状态 v，其中 $v \in Q$，且 v 是 u 的最长后缀（即在若干个后缀状态中取最长的一个作为 fail 指针）。这里简单对比一下这里的 fail 指针与 KMP 中的 next 指针：

1. 共同点：两者同样是在失配的时候用于跳转的指针。
2. 不同点：next 指针求的是最长 Border（即最长的相同前后缀），而 fail 指针指向所有模式串的前缀中匹配当前状态的最长后缀。

因为 KMP 只对一个模式串做匹配，而 AC 自动机要对多个模式串做匹配。有可能 fail 指针指向的结点对应着另一个模式串，两者前缀不同。

没看懂上面的对比不要急，你只需要知道，AC 自动机的失配指针指向当前状态的最长后缀状态即可。

AC 自动机在做匹配时，同一位上可匹配多个模式串。

构建指针

下面介绍构建 fail 指针的**基础思想**：（强调！基础思想！基础！）

构建 fail 指针，可以参考 KMP 中构造 Next 指针的思想。

考虑字典树中当前的结点 u，u 的父结点是 p，p 通过字符 `c` 的边指向 u，即 $trie[p, \texttt{c}] = u$。假设深度小于 u 的所有结点的 fail 指针都已求得。

1. 如果 $\text{trie}[\text{fail}[p], \texttt{c}]$ 存在：则让 u 的 fail 指针指向 $\text{trie}[\text{fail}[p], \texttt{c}]$。相当于在 p 和 $\text{fail}[p]$ 后面加一个字符 `c`，分别对应 u 和 $fail[u]$。
2. 如果 $\text{trie}[\text{fail}[p], \texttt{c}]$ 不存在：那么我们继续找到 $\text{trie}[\text{fail}[\text{fail}[p]], \texttt{c}]$。重复 1 的判断过程，一直跳 fail 指针直到根结点。
3. 如果真的没有，就让 fail 指针指向根结点。

如此即完成了 $\text{fail}[u]$ 的构建。

例子

下面放一张 GIF 帮助大家理解。对字符串 `i he his she hers` 组成的字典树构建 fail 指针：

1. 黄色结点：当前的结点 u。
2. 绿色结点：表示已经 BFS 遍历完毕的结点，
3. 橙色的边：fail 指针。
4. 红色的边：当前求出的 fail 指针。

我们重点分析结点 6 的 fail 指针构建：

找到 6 的父结点 5，$\text{fail}[5] = 10$。然而 10 结点没有字母 `s` 连出的边；继续跳到 10 的 fail 指针，$\text{fail}[10] = 0$。发现 0 结点有字母 `s` 连出的边，指向 7 结点；所以 $\text{fail}[6] = 7$。最后放一张建出来的图：

字典树与字典图

我们直接上代码吧。字典树插入的代码就不分析了（后面完整代码里有），先来看构建函数 `build()`，该函数的目标有两个，一个是构建 fail 指针，一个是构建自动机。参数如下：

1. `tr[u,c]`：有两种理解方式。我们可以简单理解为字典树上的一条边，即 $\text{trie}[u, c]$；也可以理解为从状态（结点）

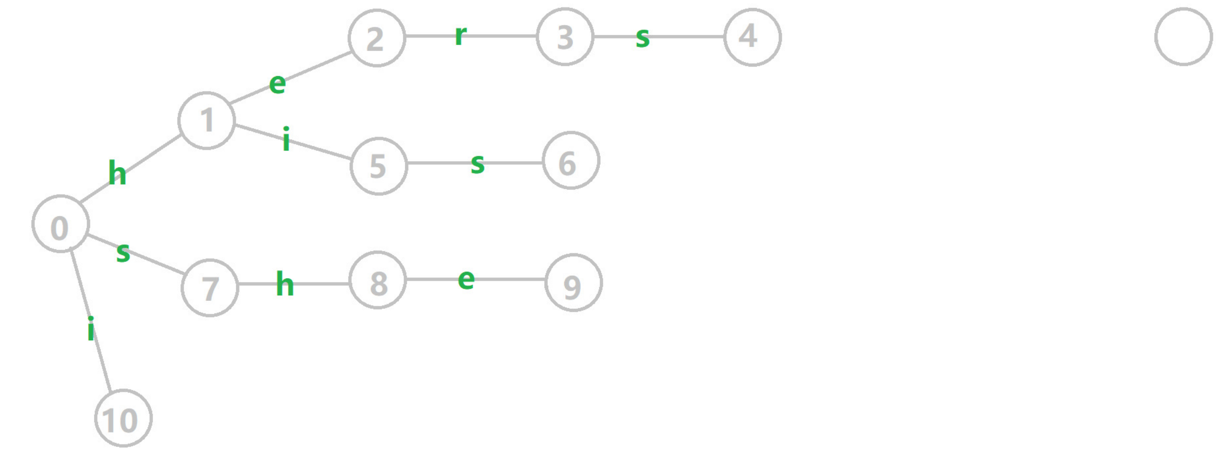


图 8.4 AC_automation_gif_b_3.gif

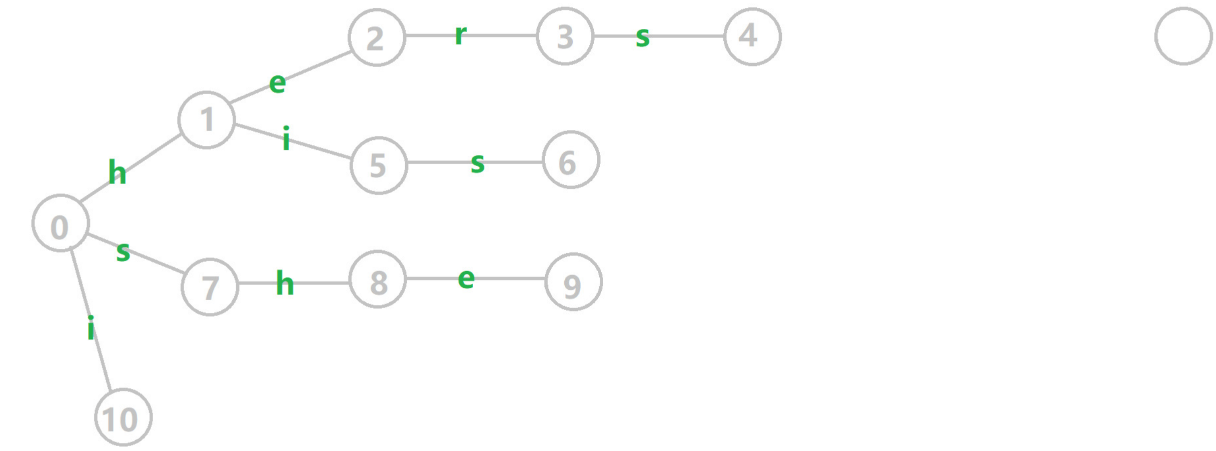


图 8.5 AC_automation_6_9.png

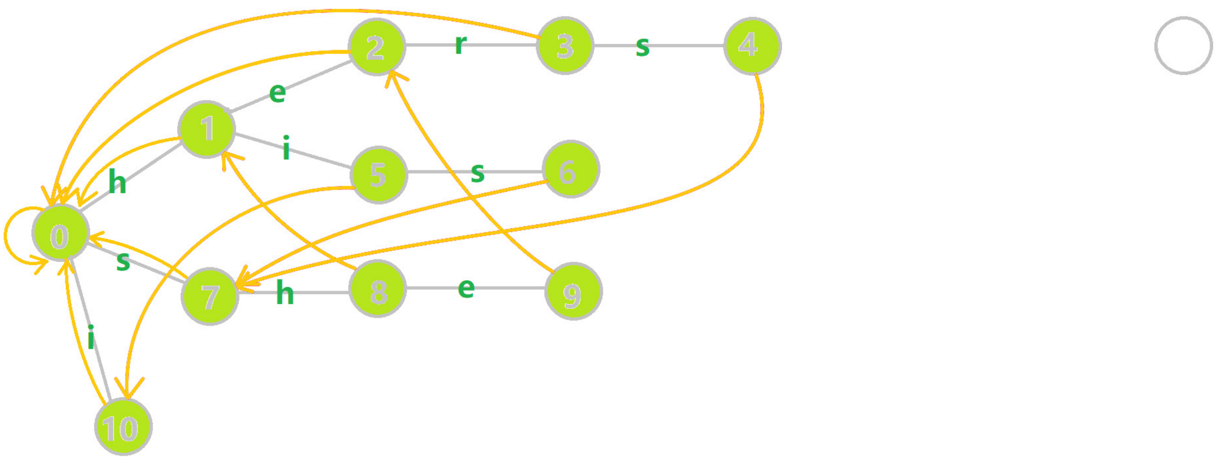


图 8.6 finish

u 后加一个字符 `c` 到达的状态（结点），即一个状态转移函数 $\text{trans}(u, c)$。下文中我们将用第二种理解方式继续讲解。

2. 队列 `q`：用于 BFS 遍历字典树。
3. `fail[u]`：结点 u 的 fail 指针。

" 实现"

```cpp
void build() {
  for (int i = 0; i < 26; i++)
    if (tr[0][i]) q.push(tr[0][i]);
  while (q.size()) {
    int u = q.front();
    q.pop();
    for (int i = 0; i < 26; i++) {
      if (tr[u][i])
        fail[tr[u][i]] = tr[fail[u]][i], q.push(tr[u][i]);
      else
        tr[u][i] = tr[fail[u]][i];
    }
  }
}
```

```python
def build():
    for i in range(0, 26):
        if tr[0][i] == 1:
            q.append(tr[0][i])
    while len(q) > 0:
        u = q[0]
        q.pop()
        for i in range(0, 26):
            if tr[u][i] == 1:
                fail[tr[u][i]] = tr[fail[u]][i]
                q.append(tr[u][i])
            else:
                tr[u][i] = tr[fail[u]][i]
```

解释

解释一下上面的代码：build 函数将结点按 BFS 顺序入队，依次求 fail 指针。这里的字典树根结点为 0，我们将根结点的子结点一一入队。若将根结点入队，则在第一次 BFS 的时候，会将根结点儿子的 fail 指针标记为本身。因此我们将根结点的儿子一一入队，而不是将根结点入队。

然后开始 BFS：每次取出队首的结点 u（$\text{fail}[u]$ 在之前的 BFS 过程中已求得），然后遍历字符集（这里是 0-25，对应 a-z，即 u 的各个子节点）：

1. 如果 $\text{trans}[u][\texttt{i}]$ 存在，我们就将 $\text{trans}[u][\texttt{i}]$ 的 fail 指针赋值为 $\text{trans}[\text{fail}[u]][\texttt{i}]$。这里似乎有一个问题。根据之前的讲解，我们应该用 while 循环，不停的跳 fail 指针，判断是否存在字符 `i` 对应的结点，然后赋值，但是这里通过特殊处理简化了这些代码。
2. 否则，令 $\text{trans}[u][\texttt{i}]$ 指向 $\text{trans}[\text{fail}[u]][\texttt{i}]$ 的状态。

这里的处理是，通过 `else` 语句的代码修改字典树的结构。没错，它将不存在的字典树的状态链接到了失配指针的对应状态。在原字典树中，每一个结点代表一个字符串 S，是某个模式串的前缀。而在修改字典树结构后，尽管增加了许多转移关系，但结点（状态）所代表的字符串是不变的。

而 trans[S][c] 相当于是在 S 后添加一个字符 c 变成另一个状态 S'。如果 S' 存在，说明存在一个模式串的前缀是 S'，否则我们让 trans[S][c] 指向 trans[fail[S]][c]。由于 fail[S] 对应的字符串是 S 的后缀，因此 trans[fail[S]][c] 对应的字符串也是 S' 的后缀。

换言之在 Trie 上跳转的时侯，我们只会从 S 跳转到 S'，相当于匹配了一个 S'；但在 AC 自动机上跳转的时侯，我们会从 S 跳转到 S' 的后缀，也就是说我们匹配一个字符 c，然后舍弃 S 的部分前缀。舍弃前缀显然是能匹配的。那么 fail 指针呢？它也是在舍弃前缀啊！试想一下，如果文本串能匹配 S，显然它也能匹配 S 的后缀。所谓的 fail 指针其实就是 S 的一个后缀集合。

tr 数组还有另一种比较简单的理解方式：如果在位置 u 失配，我们会跳转到 fail[u] 的位置。所以我们可能沿着 fail 数组跳转多次才能来到下一个能匹配的位置。所以我们可以用 tr 数组直接记录记录下一个能匹配的位置，这样就能节省下很多时间。

这样修改字典树的结构，使得匹配转移更加完善。同时它将 fail 指针跳转的路径做了压缩（就像并查集的路径压缩），使得本来需要跳很多次 fail 指针变成跳一次。

过程

我们将之前的 GIF 图改一下：

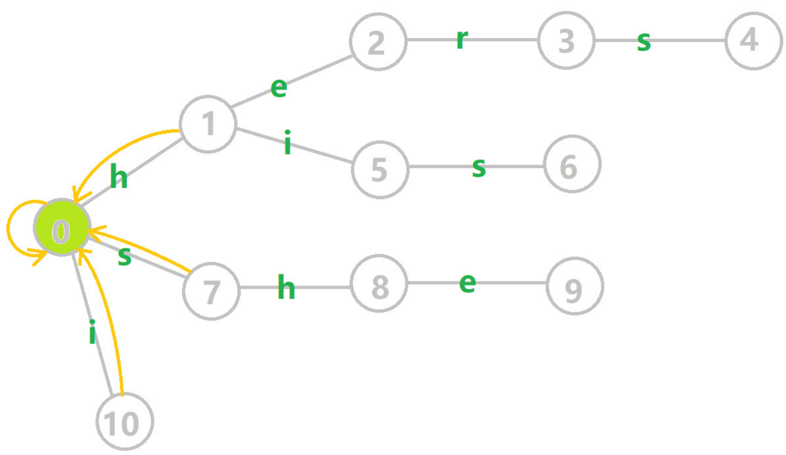


图 8.7 AC_automation_gif_b_pro3.gif

1. 蓝色结点：BFS 遍历到的结点 u
2. 蓝色的边：当前结点下，AC 自动机修改字典树结构连出的边。
3. 黑色的边：AC 自动机修改字典树结构连出的边。
4. 红色的边：当前结点求出的 fail 指针
5. 黄色的边：fail 指针
6. 灰色的边：字典树的边

可以发现，众多交错的黑色边将字典树变成了**字典图**。图中省略了连向根结点的黑边（否则会更乱）。我们重点分析一下结点 5 遍历时的情况。我们求 trans[5][s] $= 6$ 的 fail 指针：

本来的策略是找 fail 指针，于是我们跳到 fail[5] $= 10$ 发现没有 s 连出的字典树的边，于是跳到 fail[10] $= 0$，发现有 trie[0][s] $= 7$，于是 fail[6] $= 7$；但是有了黑边、蓝边，我们跳到 fail[5] $= 10$ 之后直接走 trans[10][s] $= 7$ 就走到 7 号结点了。

这就是 build 完成的两件事：构建 fail 指针和建立字典图。这个字典图也会在查询的时候起到关键作用。

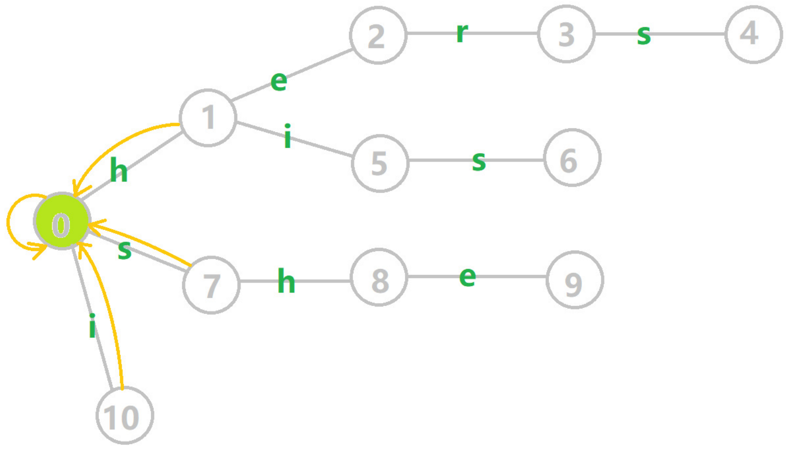


图 8.8 AC_automation_b_7.png

多模式匹配

接下来分析匹配函数 `query()`：

实现

```cpp
int query(char *t) {
  int u = 0, res = 0;
  for (int i = 1; t[i]; i++) {
    u = tr[u][t[i] - 'a'];  // 转移
    for (int j = u; j && e[j] != -1; j = fail[j]) {
      res += e[j], e[j] = -1;
    }
  }
  return res;
}
```

```python
def query(t):
    u, res = 0, 0
    i = 1
    while t[i] == False:
        u = tr[u][t[i] - ord('a')]
        j = u
        while j == True and e[j] != -1:
            res += e[j]
            e[j] = -1
            j = fail[j]
        i += 1
    return res
```

解释

这里 u 作为字典树上当前匹配到的结点，`res` 即返回的答案。循环遍历匹配串，u 在字典树上跟踪当前字符。利用 fail 指针找出所有匹配的模式串，累加到答案中。然后清零。在上文中我们分析过，字典树的结构其实就是一个 trans 函数，而构建好这个函数后，在匹配字符串的过程中，我们会舍弃部分前缀达到最低限度的匹配。fail 指针则指向了更多的匹配状态。最后上一份图。对于刚才的自动机：

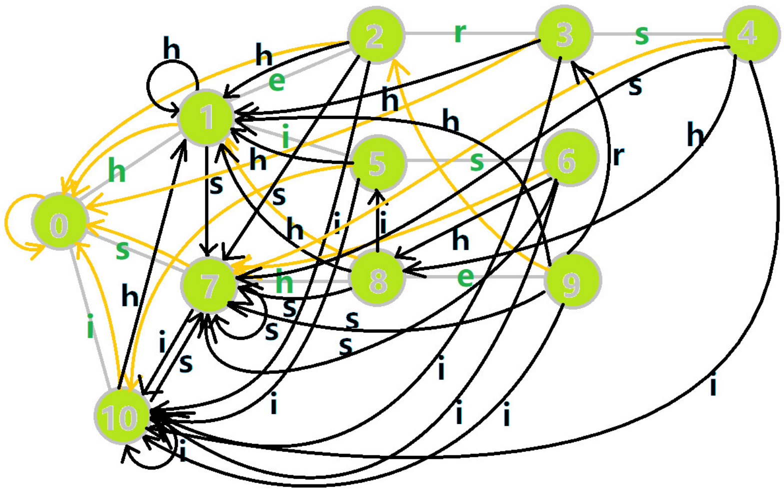


图 8.9 AC_automation_b_13.png

我们从根结点开始尝试匹配 `ushersheishis`，那么 p 的变化将是：

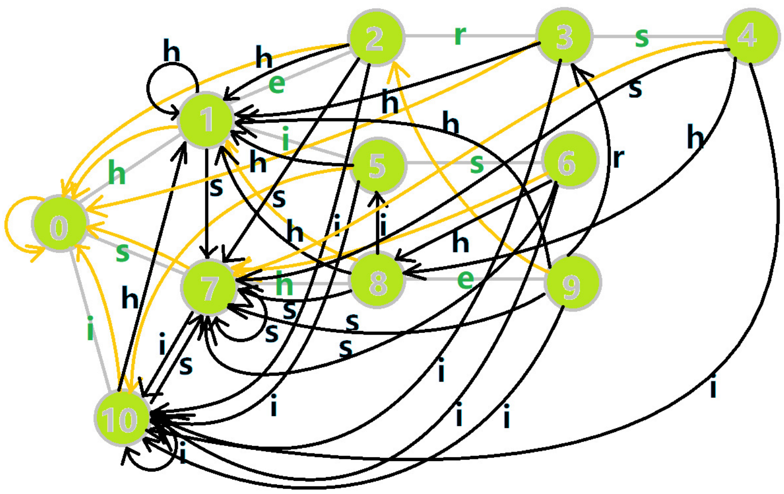


图 8.10 AC_automation_gif_c.gif

1. 红色结点：p 结点
2. 粉色箭头：p 在自动机上的跳转，
3. 蓝色的边：成功匹配的模式串
4. 蓝色结点：示跳 fail 指针时的结点（状态）。

效率优化

题目请参考洛谷 P5357【模板】AC 自动机（二次加强版）[1]

因为我们的 AC 自动机中，每次匹配，会一直向 fail 边跳来找到所有的匹配，但是这样的效率较低，在某些题目中会被卡 T。

那么我们如何优化呢？首先我们需要了解 fail 指针的一个性质：一个 AC 自动机中，如果只保留 fail 边，那么剩余的图一定是一棵树。

这是显然的，因为 fail 不会成环，且深度一定比现在低，所以得证。

而我们 AC 自动机的匹配就可以转化为在 fail 树上的链求和问题。

所以我们只需要优化一下这部分就可以了。

我们这里提供两种思路。

拓扑排序优化建图

我们浪费的时间在哪里呢？在每次都要跳 fail。如果我们可以预先记录，最后一并求和，那么效率就会优化。

于是我们按照 fail 树建图（不用真的建，只需要记录入度）：

" 建图"

```
void getfail()  // 实际上也可以叫 build
{
  for (int i = 0; i < 26; i++) trie[0].son[i] = 1;
  q.push(1);
  trie[1].fail = 0;
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    int Fail = trie[u].fail;
    for (int i = 0; i < 26; i++) {
      int v = trie[u].son[i];
      if (!v) {
        trie[u].son[i] = trie[Fail].son[i];
        continue;
      }
      trie[v].fail = trie[Fail].son[i];
      indeg[trie[Fail].son[i]]++;  // 修改点在这里，增加了入度记录
      q.push(v);
    }
  }
}
```

然后我们在查询的时候就可以只为找到节点的 ans 打上标记，在最后再用拓扑排序求出答案。

" 查询"

```
void query(char *s) {
  int u = 1, len = strlen(s);
  for (int i = 0; i < len; i++) u = trie[u].son[s[i] - 'a'], trie[u].ans++;
}

void topu() {
  for (int i = 1; i <= cnt; i++)
    if (!indeg[i]) q.push(i);
  while (!q.empty()) {
    int fr = q.front();
    q.pop();
    vis[trie[fr].flag] = trie[fr].ans;
    int u = trie[fr].fail;
    trie[u].ans += trie[fr].ans;
    if (!(--indeg[u])) q.push(u);
  }
}
```

主函数里这么写:

```
int main() {
  // do_something();
  scanf("%s", s);
  query(s);
  topu();
  for (int i = 1; i <= n; i++) cout << vis[rev[i]] << std::endl;
  // do_another_thing();
}
```

" 完整代码"

Luogu P5357【模板】AC 自动机（二次加强版）[1]

```
// Code by rickyxrc | https://www.luogu.com.cn/record/115706921
#include <bits/stdc++.h>
#define maxn 8000001
using namespace std;
char s[maxn];
int n, cnt, vis[maxn], rev[maxn], indeg[maxn], ans;

struct trie_node {
  int son[27];
  int fail;
  int flag;
  int ans;

  void init() {
    memset(son, 0, sizeof(son));
    fail = flag = 0;
  }
} trie[maxn];

queue<int> q;

void init() {
  for (int i = 0; i <= cnt; i++) trie[i].init();
  for (int i = 1; i <= n; i++) vis[i] = 0;
  cnt = 1;
  ans = 0;
}

void insert(char *s, int num) {
  int u = 1, len = strlen(s);
  for (int i = 0; i < len; i++) {
    int v = s[i] - 'a';
    if (!trie[u].son[v]) trie[u].son[v] = ++cnt;
    u = trie[u].son[v];
  }
  if (!trie[u].flag) trie[u].flag = num;
  rev[num] = trie[u].flag;
  return;
}
```

```cpp
void getfail(void) {
  for (int i = 0; i < 26; i++) trie[0].son[i] = 1;
  q.push(1);
  trie[1].fail = 0;
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    int Fail = trie[u].fail;
    for (int i = 0; i < 26; i++) {
      int v = trie[u].son[i];
      if (!v) {
        trie[u].son[i] = trie[Fail].son[i];
        continue;
      }
      trie[v].fail = trie[Fail].son[i];
      indeg[trie[Fail].son[i]]++;
      q.push(v);
    }
  }
}

void topu() {
  for (int i = 1; i <= cnt; i++)
    if (!indeg[i]) q.push(i);
  while (!q.empty()) {
    int fr = q.front();
    q.pop();
    vis[trie[fr].flag] = trie[fr].ans;
    int u = trie[fr].fail;
    trie[u].ans += trie[fr].ans;
    if (!(--indeg[u])) q.push(u);
  }
}

void query(char *s) {
  int u = 1, len = strlen(s);
  for (int i = 0; i < len; i++) u = trie[u].son[s[i] - 'a'], trie[u].ans++;
}

int main() {
  scanf("%d", &n);
  init();
  for (int i = 1; i <= n; i++) scanf("%s", s), insert(s, i);
  getfail();
  scanf("%s", s);
  query(s);
  topu();
  for (int i = 1; i <= n; i++) cout << vis[rev[i]] << std::endl;
  return 0;
}
```

子树求和

和拓扑排序的思路接近，我们预先将子树求和，询问时直接累加和值即可。

完整代码请见总结模板 3。

AC 自动机上 DP

这部分将以 P2292 [HNOI2004] L 语言[2] 为例题讲解。

一看题，不难想到一个 naive 的思路：建立 AC 自动机，在 AC 自动机上对于所有 fail 指针的子串转移，最后取最大值得到答案。

主要代码如下（若不熟悉代码中的类型定义可以跳到末尾的完整代码）：

" 查询部分主要代码"

```
void query(char *s) {
  int u = 1, len = strlen(s), l = 0;
  for (int i = 0; i < len; i++) {
    int v = s[i] - 'a';
    int k = trie[u].son[v];
    while (k > 1) {
      if (trie[k].flag && (dp[i - trie[k].len] || i - trie[k].len == -1))
        dp[i] = dp[i - trie[k].len] + trie[k].len;
      k = trie[k].fail;
    }
    u = trie[u].son[v];
  }
}
```

主函数里取 max 即可。

```
for (int i = 0, e = strlen(T); i < e; i++) mx = std::max(mx, dp[i]);
```

但是这样的思路复杂度不是线性（因为要跳每个节点的 fail），会被 subtask#2 卡到 T，所以我们需要一个优化的思路。

我们再看看题目的特殊性质，我们发现所有单词的长度只有 20，所以可以想到状态压缩优化。

具体怎么优化呢？我们发现，目前的时间瓶颈主要在跳 fail 这一步，如果我们可以将这一步优化到 $O(1)$，就可以保证整个问题在严格线性的时间内被解出。

那我们就将前 20 位字母中，可能的子串长度存下来，并压缩到状态中，存在每个子节点中。

那么我们在 buildfail 的时候就可以这么写：

" 构建 fail 指针"

```
void getfail(void) {
  for (int i = 0; i < 26; i++) trie[0].son[i] = 1;
  q.push(1);
  trie[1].fail = 0;
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    int Fail = trie[u].fail;
    // 对状态的更新在这里
    trie[u].stat = trie[Fail].stat;
    if (trie[u].flag) trie[u].stat |= 1 << trie[u].depth;
    for (int i = 0; i < 26; i++) {
      int v = trie[u].son[i];
      if (!v)
        trie[u].son[i] = trie[Fail].son[i];
      else {
```

```
        trie[v].depth = trie[u].depth + 1;
        trie[v].fail = trie[Fail].son[i];
        q.push(v);
      }
    }
  }
}
```

然后查询时就可以去掉跳 fail 的循环，将代码简化如下：

"查询"

```
int query(char *s) {
  int u = 1, len = strlen(s), mx = 0;
  unsigned st = 1;
  for (int i = 0; i < len; i++) {
    int v = s[i] - 'a';
    u = trie[u].son[v];
    // 因为往下跳了一位每一位的长度都 +1
    st <<= 1;
    // 这里的 & 值是状压 dp 的使用，代表两个长度集的交非空
    if (trie[u].stat & st) st |= 1, mx = i + 1;
  }
  return mx;
}
```

我们的 `trie[u].stat` 维护的是从 u 节点开始，整条 fail 链上的长度集（因为长度集小于 32 所以不影响），而 `st` 则维护的是查询字符串走到现在，前 32 位（因为状态压缩自然溢出）的长度集。

& 值不为 0，则代表两个长度集的交集非空，我们此时就找到了一个匹配。

"完整代码"

P2292 [HNOI2004] L 语言[2]

```
// Code by rickyxrc | https://www.luogu.com.cn/record/115806238
#include <stdio.h>
#include <string.h>

#include <queue>
#define maxn 3000001
char T[maxn];
int n, cnt, vis[maxn], ans, m, dp[maxn];

struct trie_node {
  int son[26];
  int fail, flag, depth;
  unsigned stat;

  void init() {
    memset(son, 0, sizeof(son));
    fail = flag = depth = 0;
  }
} trie[maxn];

std::queue<int> q;
```

```
void init() {
  for (int i = 0; i <= cnt; i++) trie[i].init();
  for (int i = 1; i <= n; i++) vis[i] = 0;
  cnt = 1;
  ans = 0;
}

void insert(char *s, int num) {
  int u = 1, len = strlen(s);
  for (int i = 0; i < len; i++) {
    // trie[u].depth = i + 1;
    int v = s[i] - 'a';
    if (!trie[u].son[v]) trie[u].son[v] = ++cnt;
    u = trie[u].son[v];
  }
  trie[u].flag = num;
  // trie[u].stat = 1;
  // printf("set %d stat %d\n", u-1, 1);
  return;
}

void getfail(void) {
  for (int i = 0; i < 26; i++) trie[0].son[i] = 1;
  q.push(1);
  trie[1].fail = 0;
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    int Fail = trie[u].fail;
    trie[u].stat = trie[Fail].stat;
    if (trie[u].flag) trie[u].stat |= 1 << trie[u].depth;
    for (int i = 0; i < 26; i++) {
      int v = trie[u].son[i];
      if (!v)
        trie[u].son[i] = trie[Fail].son[i];
      else {
        trie[v].depth = trie[u].depth + 1;
        trie[v].fail = trie[Fail].son[i];
        q.push(v);
      }
    }
  }
}

int query(char *s) {
  int u = 1, len = strlen(s), mx = 0;
  unsigned st = 1;
  for (int i = 0; i < len; i++) {
    int v = s[i] - 'a';
    u = trie[u].son[v];
    st <<= 1;
    if (trie[u].stat & st) st |= 1, mx = i + 1;
  }
```

```
  return mx;
}

int main() {
  scanf("%d%d", &n, &m);
  init();
  for (int i = 1; i <= n; i++) {
    scanf("%s", T);
    insert(T, i);
  }
  getfail();
  for (int i = 1; i <= m; i++) {
    scanf("%s", T);
    printf("%d\n", query(T));
  }
}
```

总结

希望大家看懂了文章。

时间复杂度：定义 $|s_i|$ 是模板串的长度，$|S|$ 是文本串的长度，$|\Sigma|$ 是字符集的大小（常数，一般为 26）。如果连了 trie 图，时间复杂度就是 $O(\sum|s_i|+n|\Sigma|+|S|)$，其中 n 是 AC 自动机中结点的数目，并且最大可以达到 $O(\sum|s_i|)$。如果不连 trie 图，并且在构建 fail 指针的时候避免遍历到空儿子，时间复杂度就是 $O(\sum|s_i|+|S|)$。

" 模板 1"

Luogu P3808【模板】AC 自动机（简单版）[3]

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e6 + 6;
int n;

namespace AC {
int tr[N][26], tot;
int e[N], fail[N];

void insert(char *s) {
  int u = 0;
  for (int i = 1; s[i]; i++) {
    if (!tr[u][s[i] - 'a']) tr[u][s[i] - 'a'] = ++tot;  // 如果没有则插入新节点
    u = tr[u][s[i] - 'a'];                              // 搜索下一个节点
  }
  e[u]++;  // 尾为节点 u 的串的个数
}

queue<int> q;

void build() {
  for (int i = 0; i < 26; i++)
    if (tr[0][i]) q.push(tr[0][i]);
  while (q.size()) {
    int u = q.front();
    q.pop();
```

```
    for (int i = 0; i < 26; i++) {
      if (tr[u][i]) {
        fail[tr[u][i]] =
            tr[fail[u]][i];  // fail 数组：同一字符可以匹配的其他位置
        q.push(tr[u][i]);
      } else
        tr[u][i] = tr[fail[u]][i];
    }
  }
}

int query(char *t) {
  int u = 0, res = 0;
  for (int i = 1; t[i]; i++) {
    u = tr[u][t[i] - 'a'];  // 转移
    for (int j = u; j && e[j] != -1; j = fail[j]) {
      res += e[j], e[j] = -1;
    }
  }
  return res;
}
}  // namespace AC

char s[N];

int main() {
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) scanf("%s", s + 1), AC::insert(s);
  scanf("%s", s + 1);
  AC::build();
  printf("%d", AC::query(s));
  return 0;
}
```

" 模板 2"

Luogu P3796【模板】AC 自动机（加强版）[4]

```
#include <bits/stdc++.h>
using namespace std;
const int N = 156, L = 1e6 + 6;

namespace AC {
const int SZ = N * 80;
int tot, tr[SZ][26];
int fail[SZ], idx[SZ], val[SZ];
int cnt[N];  // 记录第 i 个字符串的出现次数

void init() {
  memset(fail, 0, sizeof(fail));
  memset(tr, 0, sizeof(tr));
  memset(val, 0, sizeof(val));
  memset(cnt, 0, sizeof(cnt));
```

```
  memset(idx, 0, sizeof(idx));
  tot = 0;
}

void insert(char *s, int id) {  // id 表示原始字符串的编号
  int u = 0;
  for (int i = 1; s[i]; i++) {
    if (!tr[u][s[i] - 'a']) tr[u][s[i] - 'a'] = ++tot;
    u = tr[u][s[i] - 'a'];  // 转移
  }
  idx[u] = id;  // 以 u 为结尾的字符串编号为 idx[u]
}

queue<int> q;

void build() {
  for (int i = 0; i < 26; i++)
    if (tr[0][i]) q.push(tr[0][i]);
  while (q.size()) {
    int u = q.front();
    q.pop();
    for (int i = 0; i < 26; i++) {
      if (tr[u][i]) {
        fail[tr[u][i]] =
            tr[fail[u]][i];  // fail 数组：同一字符可以匹配的其他位置
        q.push(tr[u][i]);
      } else
        tr[u][i] = tr[fail[u]][i];
    }
  }
}

int query(char *t) {  // 返回最大的出现次数
  int u = 0, res = 0;
  for (int i = 1; t[i]; i++) {
    u = tr[u][t[i] - 'a'];
    for (int j = u; j; j = fail[j]) val[j]++;
  }
  for (int i = 0; i <= tot; i++)
    if (idx[i]) res = max(res, val[i]), cnt[idx[i]] = val[i];
  return res;
}
}  // namespace AC

int n;
char s[N][100], t[L];

int main() {
  while (~scanf("%d", &n)) {
    if (n == 0) break;
    AC::init();  // 数组清零
    for (int i = 1; i <= n; i++)
      scanf("%s", s[i] + 1), AC::insert(s[i], i);  // 需要记录该字符串的序号
    AC::build();
```

```
    scanf("%s", t + 1);
    int x = AC::query(t);
    printf("%d\n", x);
    for (int i = 1; i <= n; i++)
      if (AC::cnt[i] == x) printf("%s\n", s[i] + 1);
  }
  return 0;
}
```

" 模版 3"

Luogu P5357【模板】AC 自动机（二次加强版）[1]

```
#include <deque>
#include <iostream>

void promote() {
  std::ios::sync_with_stdio(0);
  std::cin.tie(0);
  std::cout.tie(0);
  return;
}

typedef char chr;
typedef std::deque<int> dic;

const int maxN = 2e5;
const int maxS = 2e5;
const int maxT = 2e6;

int n;
chr s[maxS + 10];
chr t[maxT + 10];
int cnt[maxN + 10];

struct AhoCorasickAutomaton {
  struct Node {
    int son[30];
    int val;
    int fail;
    int head;
    dic index;
  } node[maxS + 10];

  struct Edge {
    int head;
    int next;
  } edge[maxS + 10];

  int root;
  int ncnt;
  int ecnt;
```

```
void Insert(chr *str, int i) {
  int u = root;
  for (int i = 1; str[i]; i++) {
    if (node[u].son[str[i] - 'a' + 1] == 0)
      node[u].son[str[i] - 'a' + 1] = ++ncnt;
    u = node[u].son[str[i] - 'a' + 1];
  }
  node[u].index.push_back(i);
  return;
}

void Build() {
  dic q;
  for (int i = 1; i <= 26; i++)
    if (node[root].son[i]) q.push_back(node[root].son[i]);
  while (!q.empty()) {
    int u = q.front();
    q.pop_front();
    for (int i = 1; i <= 26; i++) {
      if (node[u].son[i]) {
        node[node[u].son[i]].fail = node[node[u].fail].son[i];
        q.push_back(node[u].son[i]);
      } else {
        node[u].son[i] = node[node[u].fail].son[i];
      }
    }
  }
  return;
}

void Query(chr *str) {
  int u = root;
  for (int i = 1; str[i]; i++) {
    u = node[u].son[str[i] - 'a' + 1];
    node[u].val++;
  }
  return;
}

void addEdge(int tail, int head) {
  ecnt++;
  edge[ecnt].head = head;
  edge[ecnt].next = node[tail].head;
  node[tail].head = ecnt;
  return;
}

void DFS(int u) {
  for (int e = node[u].head; e; e = edge[e].next) {
    int v = edge[e].head;
    DFS(v);
    node[u].val += node[v].val;
  }
  for (auto i : node[u].index) cnt[i] += node[u].val;
```

```
    return;
  }

  void FailTree() {
    for (int u = 1; u <= ncnt; u++) addEdge(node[u].fail, u);
    DFS(root);
    return;
  }
} ACM;

int main() {
  std::cin >> n;
  for (int i = 1; i <= n; i++) {
    std::cin >> (s + 1);
    ACM.Insert(s, i);
  }
  ACM.Build();
  std::cin >> (t + 1);
  ACM.Query(t);
  ACM.FailTree();
  for (int i = 1; i <= n; i++) std::cout << cnt[i] << '\n';
  return 0;
}
```

拓展

确定有限状态自动机

如果大家理解了上面的讲解，那么作为拓展延伸，文末我们简单介绍一下 自动机 与 KMP 自动机。（现在你再去看自动机的定义就会好懂很多啦）

有限状态自动机（Deterministic Finite Automaton，DFA）是由

1. 状态集合 Q；
2. 字符集 Σ；
3. 状态转移函数 $\delta : Q \times \Sigma \to Q$，即 $\delta(q, \sigma) = q'$，$q, q' \in Q, \sigma \in \Sigma$；
4. 一个开始状态 $s \in Q$；
5. 一个接收的状态集合 $F \subseteq Q$。

组成的五元组 $(Q, \Sigma, \delta, s, F)$。

那这东西你用 AC 自动机理解，状态集合就是字典树（图）的结点；字符集就是 `a` 到 `z`（或者更多）；状态转移函数就是 $\text{trans}(u, c)$ 的函数（即 $\text{trans}[u][c]$）；开始状态就是字典树的根结点；接收状态就是你在字典树中标记的字符串结尾结点组成的集合。

KMP 自动机

KMP 自动机就是一个不断读入待匹配串，每次匹配时走到接受状态的 DFA。如果共有 m 个状态，第 i 个状态表示已经匹配了前 i 个字符。那么我们定义 $\text{trans}_{i,c}$ 表示状态 i 读入字符 c 后到达的状态，next_i 表示 prefix function，则有：

$$\text{trans}_{i,c} = \begin{cases} i + 1, & \text{if } b_i = c \\ \text{trans}_{\text{next}_i, c}, & \text{otherwise} \end{cases}$$

（约定 $\text{next}_0 = 0$）

我们发现 trans_i 只依赖于之前的值，所以可以跟 KMP 一起求出来。（一些细节：走到接受状态之后立即转移到该状态的 next）

时间和空间复杂度：$O(m|\Sigma|)$。

对比之下，AC 自动机其实就是 Trie 上的自动机。（虽然一开始丢给你这句话可能不知所措）

参考资料与注释

[1] P5357【模板】AC 自动机（二次加强版） [1-1] [1-2] [1-3]

[2] P2292 [HNOI2004] L 语言 [2-1] [2-2]

[3] Luogu P3808【模板】AC 自动机（简单版）

[4] Luogu P3796【模板】AC 自动机（加强版）

8.12 后缀数组 (SA)

8.12.1 后缀数组简介

一些约定

字符串相关的定义请参考 字符串基础。

字符串下标从 1 开始。

字符串 s 的长度为 n。

" 后缀 i" 代指以第 i 个字符开头的后缀，存储时用 i 代表字符串 s 的后缀 $s[i \dots n]$。

后缀数组是什么？

后缀数组（Suffix Array）主要关系到两个数组：sa 和 rk。

其中，$sa[i]$ 表示将所有后缀排序后第 i 小的后缀的编号，也是所说的后缀数组，后文也称编号数组 sa；

$rk[i]$ 表示后缀 i 的排名，是重要的辅助数组，后文也称排名数组 rk。

这两个数组满足性质：$sa[rk[i]] = rk[sa[i]] = i$。

解释

后缀数组示例：

后缀数组怎么求？

O(n^2logn) 做法

相信这个做法大家还是能自己想到的：将盛有全部后缀字符串的数组进行 `sort` 排序，由于排序进行 $O(n \log n)$ 次字符串比较，每次字符串比较要 $O(n)$ 次字符比较，所以这个排序是 $O(n^2 \log n)$ 的时间复杂度。

O(nlog^2n) 做法

这个做法要用到倍增的思想。

首先对字符串 s 的所有长度为 1 的子串，即每个字符进行排序，得到排序后的编号数组 sa_1 和排名数组 rk_1。

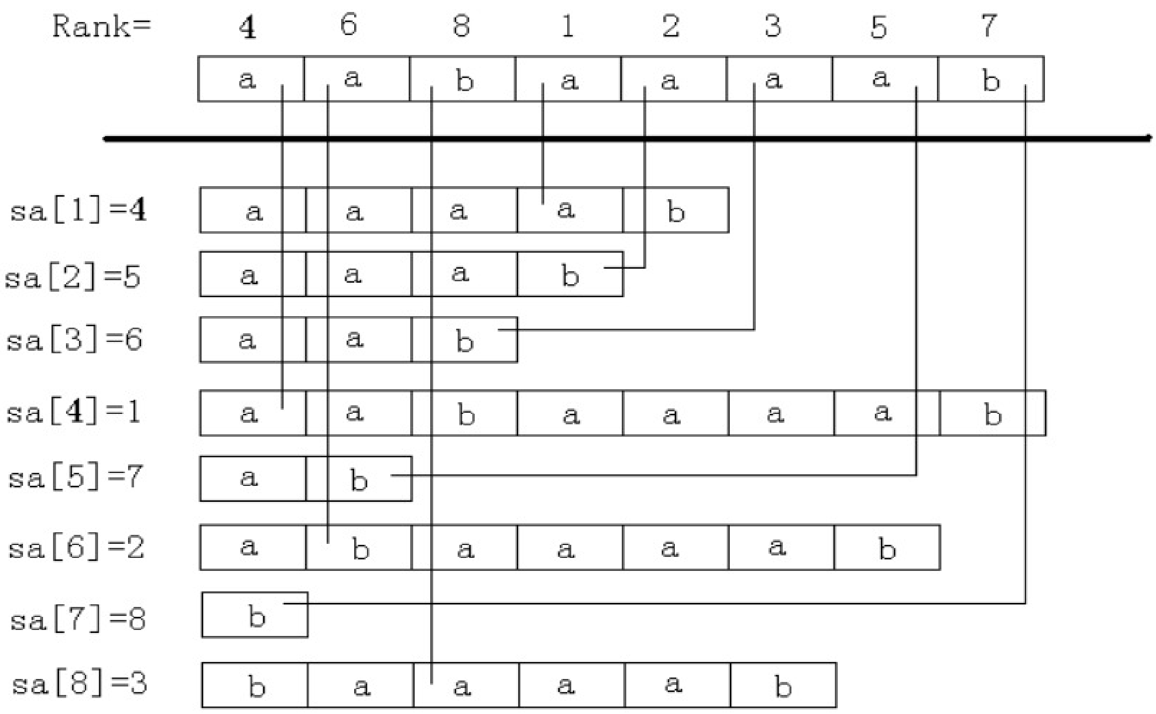


图 8.11

倍增过程：

1. 用两个长度为 1 的子串的排名，即 $rk_1[i]$ 和 $rk_1[i+1]$，作为排序的第一第二关键字，就可以对字符串 s 的每个长度为 2 的子串：$\{s[i\ldots\min(i+1,n)] \mid i\in[1,\ n]\}$ 进行排序，得到 sa_2 和 rk_2；
2. 之后用两个长度为 2 的子串的排名，即 $rk_2[i]$ 和 $rk_2[i+2]$，作为排序的第一第二关键字，就可以对字符串 s 的每个长度为 4 的子串：$\{s[i\ldots\min(i+3,n)] \mid i\in[1,\ n]\}$ 进行排序，得到 sa_4 和 rk_4；
3. 以此倍增，用长度为 $w/2$ 的子串的排名，即 $rk_{w/2}[i]$ 和 $rk_{w/2}[i+w/2]$，作为排序的第一第二关键字，就可以对字符串 s 的每个长度为 w 的子串 $s[i\ldots\min(i+w-1,\ n)]$ 进行排序，得到 sa_w 和 rk_w。其中，类似字母序排序规则，当 $i+w>n$ 时，$rk_w[i+w]$ 视为无穷小；
4. $rk_w[i]$ 即是子串 $s[i\ldots i+w-1]$ 的排名，这样当 $w\geqslant n$ 时，得到的编号数组 sa_w，也就是我们需要的后缀数组。

过程

倍增排序示意图：

显然倍增的过程是 $O(\log n)$，而每次倍增用 `sort` 对子串进行排序是 $O(n\log n)$，而每次子串的比较花费 2 次字符比较；

除此之外，每次倍增在 `sort` 排序完后，还有额外的 $O(n)$ 时间复杂度的，更新 rk 的操作，但是相对于 $O(n\log n)$ 被忽略不计；

所以这个算法的时间复杂度就是 $O(n\log^2 n)$。

"实现"

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <iostream>
```

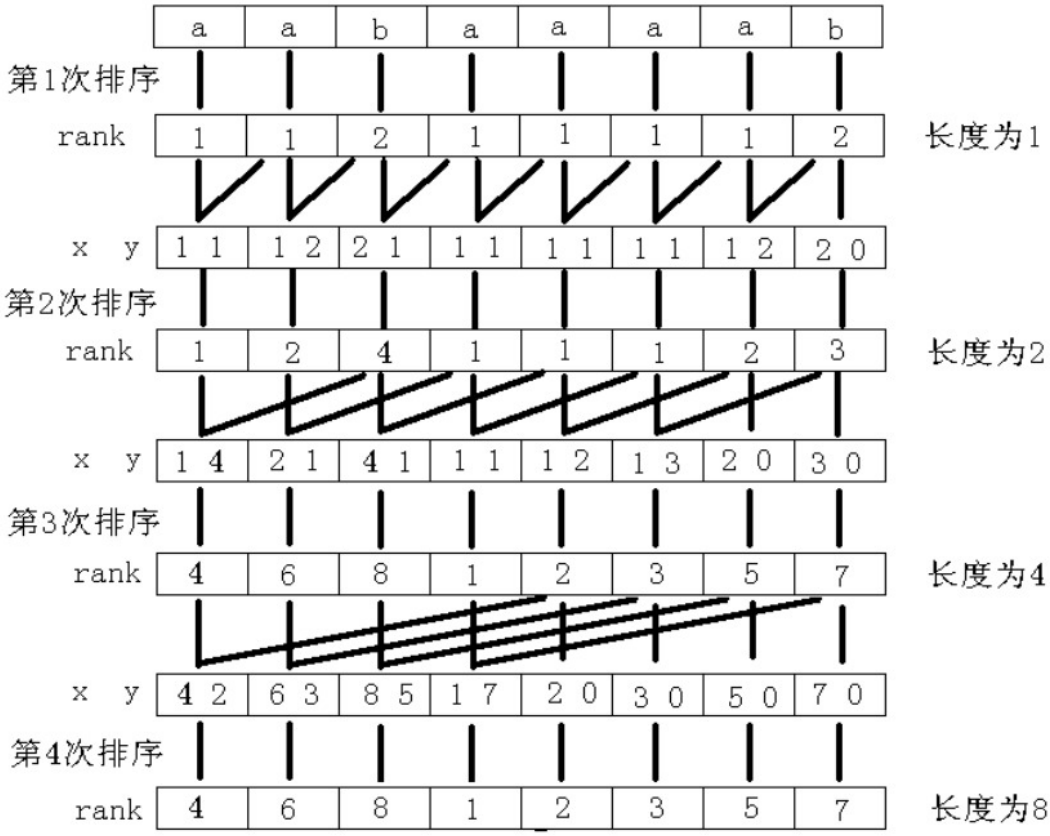


图 8.12

```cpp
using namespace std;

const int N = 1000010;

char s[N];
int n, w, sa[N], rk[N << 1], oldrk[N << 1];

// 为了防止访问 rk[i+w] 导致数组越界，开两倍数组。
// 当然也可以在访问前判断是否越界，但直接开两倍数组方便一些。

int main() {
  int i, p;

  scanf("%s", s + 1);
  n = strlen(s + 1);
  for (i = 1; i <= n; ++i) sa[i] = i, rk[i] = s[i];

  for (w = 1; w < n; w <<= 1) {
    sort(sa + 1, sa + n + 1, [](int x, int y) {
      return rk[x] == rk[y] ? rk[x + w] < rk[y + w] : rk[x] < rk[y];
    });  // 这里用到了 lambda
    memcpy(oldrk, rk, sizeof(rk));
    // 由于计算 rk 的时候原来的 rk 会被覆盖，要先复制一份
    for (p = 0, i = 1; i <= n; ++i) {
      if (oldrk[sa[i]] == oldrk[sa[i - 1]] &&
          oldrk[sa[i] + w] == oldrk[sa[i - 1] + w]) {
```

```
        rk[sa[i]] = p;
      } else {
        rk[sa[i]] = ++p;
      }  // 若两个子串相同，它们对应的 rk 也需要相同，所以要去重
    }
  }

  for (i = 1; i <= n; ++i) printf("%d ", sa[i]);

  return 0;
}
```

O(nlogn) 做法

在刚刚的 $O(n\log^2 n)$ 做法中，单次排序是 $O(n\log n)$ 的，如果能 $O(n)$ 排序，就能 $O(n\log n)$ 计算后缀数组了。

前置知识：计数排序，基数排序。

由于计算后缀数组的过程中排序的关键字是排名，值域为 $O(n)$，并且是一个双关键字的排序，可以使用基数排序优化至 $O(n)$。

" 实现"

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <iostream>

using namespace std;

const int N = 1000010;

char s[N];
int n, sa[N], rk[N << 1], oldrk[N << 1], id[N], cnt[N];

int main() {
  int i, m, p, w;

  scanf("%s", s + 1);
  n = strlen(s + 1);
  m = 127;
  for (i = 1; i <= n; ++i) ++cnt[rk[i] = s[i]];
  for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
  for (i = n; i >= 1; --i) sa[cnt[rk[i]]--] = i;
  memcpy(oldrk + 1, rk + 1, n * sizeof(int));
  for (p = 0, i = 1; i <= n; ++i) {
    if (oldrk[sa[i]] == oldrk[sa[i - 1]]) {
      rk[sa[i]] = p;
    } else {
      rk[sa[i]] = ++p;
    }
  }

  for (w = 1; w < n; w <<= 1, m = n) {
    // 对第二关键字：id[i] + w 进行计数排序
    memset(cnt, 0, sizeof(cnt));
```

```
    memcpy(id + 1, sa + 1,
           n * sizeof(int));  // id 保存一份儿 sa 的拷贝，实质上就相当于 oldsa
    for (i = 1; i <= n; ++i) ++cnt[rk[id[i] + w]];
    for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
    for (i = n; i >= 1; --i) sa[cnt[rk[id[i] + w]]--] = id[i];

    // 对第一关键字：id[i] 进行计数排序
    memset(cnt, 0, sizeof(cnt));
    memcpy(id + 1, sa + 1, n * sizeof(int));
    for (i = 1; i <= n; ++i) ++cnt[rk[id[i]]];
    for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
    for (i = n; i >= 1; --i) sa[cnt[rk[id[i]]]--] = id[i];

    memcpy(oldrk + 1, rk + 1, n * sizeof(int));
    for (p = 0, i = 1; i <= n; ++i) {
      if (oldrk[sa[i]] == oldrk[sa[i - 1]] &&
          oldrk[sa[i] + w] == oldrk[sa[i - 1] + w]) {
        rk[sa[i]] = p;
      } else {
        rk[sa[i]] = ++p;
      }
    }
  }

  for (i = 1; i <= n; ++i) printf("%d ", sa[i]);

  return 0;
}
```

一些常数优化

如果你把上面那份代码交到 LOJ #111：后缀排序[1] 上：

测试点 #1	Accepted	得分：100	用时：20 ms	内存：11900 KiB
测试点 #2	Accepted	得分：100	用时：27 ms	内存：12004 KiB
测试点 #3	Accepted	得分：100	用时：27 ms	内存：11900 KiB
测试点 #4	Accepted	得分：100	用时：37 ms	内存：12112 KiB
测试点 #5	Accepted	得分：100	用时：36 ms	内存：12112 KiB
测试点 #6	Accepted	得分：100	用时：103 ms	内存：13200 KiB
测试点 #7	Accepted	得分：100	用时：111 ms	内存：13248 KiB
测试点 #8	Time Limit Exceeded	得分：0	用时：4018 ms	内存：24576 KiB
测试点 #9	Time Limit Exceeded	得分：0	用时：4014 ms	内存：24576 KiB
测试点 #10	Time Limit Exceeded	得分：0	用时：4018 ms	内存：24576 KiB

图 8.13

这是因为，上面那份代码的常数的确很大。

第二关键字无需计数排序

思考一下第二关键字排序的实质，其实就是把超出字符串范围（即 $sa[i]+w>n$）的 $sa[i]$ 放到 sa 数组头部，然后把剩下的依原顺序放入：

```
for (p = 0, i = n; i > n - w; --i) id[++p] = i;

for (i = 1; i <= n; ++i) {
  if (sa[i] > w) id[++p] = sa[i] - w;
}
```

优化计数排序的值域

每次对 rk 进行更新之后，我们都计算了一个 p，这个 p 即是 rk 的值域，将值域改成它即可。

将 rk[id[i]] 存下来，减少不连续内存访问

这个优化在数据范围较大时效果非常明显。

用函数 cmp 来计算是否重复

同样是减少不连续内存访问，在数据范围较大时效果比较明显。

把 `oldrk[sa[i]] == oldrk[sa[i - 1]] && oldrk[sa[i] + w] == oldrk[sa[i - 1] + w]`

替换成 `cmp(sa[i], sa[i - 1], w)`，

`bool cmp(int x, int y, int w) { return oldrk[x] == oldrk[y] && oldrk[x + w] == oldrk[y + w]; }`。

若排名都不相同可直接生成后缀数组

考虑新的 rk 数组，若其值域为 $[1, n]$ 那么每个排名都不同，此时无需再排序。

" 实现"

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <iostream>

using namespace std;

const int N = 1000010;

char s[N];
// key1[i] = rk[id[i]]（作为基数排序的第一关键字数组）
int n, sa[N], rk[N], oldrk[N << 1], id[N], key1[N], cnt[N];

bool cmp(int x, int y, int w) {
  return oldrk[x] == oldrk[y] && oldrk[x + w] == oldrk[y + w];
}

int main() {
  int i, m = 127, p, w;

  scanf("%s", s + 1);
  n = strlen(s + 1);
  for (i = 1; i <= n; ++i) ++cnt[rk[i] = s[i]];
  for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
  for (i = n; i >= 1; --i) sa[cnt[rk[i]]--] = i;

  for (w = 1;; w <<= 1, m = p) {  // m=p 就是优化计数排序值域
    for (p = 0, i = n; i > n - w; --i) id[++p] = i;
```

```
    for (i = 1; i <= n; ++i)
      if (sa[i] > w) id[++p] = sa[i] - w;

    memset(cnt, 0, sizeof(cnt));
    for (i = 1; i <= n; ++i) ++cnt[key1[i] = rk[id[i]]];
    // 注意这里 px[i] != i，因为 rk 没有更新，是上一轮的排名数组

    for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
    for (i = n; i >= 1; --i) sa[cnt[key1[i]]--] = id[i];
    memcpy(oldrk + 1, rk + 1, n * sizeof(int));
    for (p = 0, i = 1; i <= n; ++i)
      rk[sa[i]] = cmp(sa[i], sa[i - 1], w) ? p : ++p;
    if (p == n) {
      break;
    }
  }

  for (i = 1; i <= n; ++i) printf("%d ", sa[i]);

  return 0;
}
```

O(n) 做法

在一般的题目中，常数较小的倍增求后缀数组是完全够用的，求后缀数组以外的部分也经常有 $O(n\log n)$ 的复杂度，倍增求解后缀数组不会成为瓶颈。

但如果遇到特殊题目、时限较紧的题目，或者是你想追求更短的用时，就需要学习 $O(n)$ 求后缀数组的方法。

SA-IS

可以参考 诱导排序与 SA-IS 算法[2]，另外它的 评论页面[3] 也有参考价值。

DC3

可以参考[2009] 后缀数组 —— 处理字符串的有力工具 by. 罗穗骞。

后缀数组的应用

寻找最小的循环移动位置

将字符串 S 复制一份变成 SS 就转化成了后缀排序问题。

例题：「JSOI2007」 字符加密[4]。

在字符串中找子串

任务是在线地在主串 T 中寻找模式串 S。在线的意思是，我们已经预先知道知道主串 T，但是当且仅当询问时才知道模式串 S。我们可以先构造出 T 的后缀数组，然后查找子串 S。若子串 S 在 T 中出现，它必定是 T 的一些后缀的前缀。因为我们已经将所有后缀排序了，我们可以通过在 p 数组中二分 S 来实现。比较子串 S 和当前后缀的时间复杂度为 $O(|S|)$，因此找子串的时间复杂度为 $O(|S|\log|T|)$。注意，如果该子串在 T 中出现了多次，每次出现都是在 p 数组中相邻的。因此出现次数可以通过再次二分找到，输出每次出现的位置也很轻松。

从字符串首尾取字符最小化字典序

例题：「USACO07DEC」Best Cow Line[5]。

题意：给你一个字符串，每次从首或尾取一个字符组成字符串，问所有能够组成的字符串中字典序最小的一个。

" 题解"

暴力做法就是每次最坏 $O(n)$ 地判断当前应该取首还是尾（即比较取首得到的字符串与取尾得到的反串的大小），只需优化这一判断过程即可。

由于需要在原串后缀与反串后缀构成的集合内比较大小，可以将反串拼接在原串后，并在中间加上一个没出现过的字符（如 #，代码中可以直接使用空字符），求后缀数组，即可 $O(1)$ 完成这一判断。

" 参考代码"

```
#include <cctype>
#include <cstdio>
#include <cstring>
#include <iostream>

using namespace std;

const int N = 1000010;

char s[N];
int n, sa[N], id[N], oldrk[N * 2], rk[N * 2], px[N], cnt[N];

bool cmp(int x, int y, int w) {
  return oldrk[x] == oldrk[y] && oldrk[x + w] == oldrk[y + w];
}

int main() {
  int i, w, m = 200, p, l = 1, r, tot = 0;

  cin >> n;
  r = n;

  for (i = 1; i <= n; ++i)
    while (!isalpha(s[i] = getchar()))
      ;
  for (i = 1; i <= n; ++i)
    rk[i] = rk[2 * n + 2 - i] = s[i];  // 拼接正反两个字符串，中间空出一个字符

  n = 2 * n + 1;
  // 求后缀数组
  for (i = 1; i <= n; ++i) ++cnt[rk[i]];
  for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
  for (i = n; i >= 1; --i) sa[cnt[rk[i]]--] = i;

  for (w = 1; w < n; w *= 2, m = p) {  // m=p 就是优化计数排序值域
    for (p = 0, i = n; i > n - w; --i) id[++p] = i;
    for (i = 1; i <= n; ++i)
      if (sa[i] > w) id[++p] = sa[i] - w;
    memset(cnt, 0, sizeof(cnt));
    for (i = 1; i <= n; ++i) ++cnt[px[i] = rk[id[i]]];
    for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
    for (i = n; i >= 1; --i) sa[cnt[px[i]]--] = id[i];
    memcpy(oldrk, rk, sizeof(rk));
    for (p = 0, i = 1; i <= n; ++i)
```

```
      rk[sa[i]] = cmp(sa[i], sa[i - 1], w) ? p : ++p;
  }
  // 利用后缀数组 O(1) 进行判断
  while (l <= r) {
    printf("%c", rk[l] < rk[n + 1 - r] ? s[l++] : s[r--]);
    if ((++tot) % 80 == 0) puts("");  // 回车
  }

  return 0;
}
```

height 数组

LCP（最长公共前缀）

两个字符串 S 和 T 的 LCP 就是最大的 $x(x \le \min(|S|, |T|))$ 使得 $S_i = T_i\ (\forall\ 1 \le i \le x)$。

下文中以 $lcp(i, j)$ 表示后缀 i 和后缀 j 的最长公共前缀（的长度）。

height 数组的定义

$height[i] = lcp(sa[i], sa[i-1])$，即第 i 名的后缀与它前一名的后缀的最长公共前缀。

$height[1]$ 可以视作 0。

O(n) 求 height 数组需要的一个引理

$height[rk[i]] \ge height[rk[i-1]] - 1$

" 证明"

当 $height[rk[i-1]] \le 1$ 时，上式显然成立（右边小于等于 0）。

当 $height[rk[i-1]] > 1$ 时：

根据 $height$ 定义，有 $lcp(sa[rk[i-1]], sa[rk[i-1]-1]) = height[rk[i-1]] > 1$。

既然后缀 $i-1$ 和后缀 $sa[rk[i-1]-1]$ 有长度为 $height[rk[i-1]]$ 的最长公共前缀，

那么不妨用 aA 来表示这个最长公共前缀。（其中 a 是一个字符，A 是长度为 $height[rk[i-1]]-1$ 的字符串，非空）

那么后缀 $i-1$ 可以表示为 aAD，后缀 $sa[rk[i-1]-1]$ 可以表示为 aAB。（$B < D$，B 可能为空串，D 非空）

进一步地，后缀 i 可以表示为 AD，存在后缀（$sa[rk[i-1]-1]+1$）AB。

因为后缀 $sa[rk[i]-1]$ 在大小关系的排名上仅比后缀 $sa[rk[i]]$ 也就是后缀 i，小一位，而 $AB < AD$。

所以 $AB \leqslant$ 后缀 $sa[rk[i]-1] < AD$，显然后缀 i 和后缀 $sa[rk[i]-1]$ 有公共前缀 A。

于是就可以得出 $lcp(i, sa[rk[i]-1])$ 至少是 $height[rk[i-1]]-1$，也即 $height[rk[i]] \ge height[rk[i-1]]-1$。

O(n) 求 height 数组的代码实现

利用上面这个引理暴力求即可：

```
for (i = 1, k = 0; i <= n; ++i) {
  if (rk[i] == 0) continue;
  if (k) --k;
  while (s[i + k] == s[sa[rk[i] - 1] + k]) ++k;
  height[rk[i]] = k;
}
```

k 不会超过 n，最多减 n 次，所以最多加 $2n$ 次，总复杂度就是 $O(n)$。

height 数组的应用

两子串最长公共前缀

$lcp(sa[i], sa[j]) = \min\{height[i+1..j]\}$

感性理解：如果 $height$ 一直大于某个数，前这么多位就一直没变过；反之，由于后缀已经排好序了，不可能变了之后变回来。

严格证明可以参考[2004] 后缀数组 by. 徐智磊。

有了这个定理，求两子串最长公共前缀就转化为了 RMQ 问题。

比较一个字符串的两个子串的大小关系

假设需要比较的是 $A = S[a..b]$ 和 $B = S[c..d]$ 的大小关系。

若 $lcp(a, c) \ge \min(|A|, |B|)$，$A < B \iff |A| < |B|$。

否则，$A < B \iff rk[a] < rk[c]$。

不同子串的数目

子串就是后缀的前缀，所以可以枚举每个后缀，计算前缀总数，再减掉重复。

「前缀总数」其实就是子串个数，为 $n(n+1)/2$。

如果按后缀排序的顺序枚举后缀，每次新增的子串就是除了与上一个后缀的 LCP 剩下的前缀。这些前缀一定是新增的，否则会破坏 $lcp(sa[i], sa[j]) = \min\{height[i+1..j]\}$ 的性质。只有这些前缀是新增的，因为 LCP 部分在枚举上一个前缀时计算过了。

所以答案为：

$\frac{n(n+1)}{2} - \sum\limits_{i=2}^{n} height[i]$

出现至少 k 次的子串的最大长度

例题：「USACO06DEC」Milk Patterns[6]。

" 题解"

出现至少 k 次意味着后缀排序后有至少连续 k 个后缀以这个子串作为公共前缀。

所以，求出每相邻 $k-1$ 个 $height$ 的最小值，再求这些最小值的最大值就是答案。

可以使用单调队列 $O(n)$ 解决，但使用其它方式也足以 AC。

" 参考代码"

```
#include <cstdio>
#include <cstring>
#include <iostream>
#include <set>

using namespace std;

const int N = 40010;

int n, k, a[N], sa[N], rk[N], oldrk[N], id[N], px[N], cnt[1000010], ht[N], ans;
multiset<int> t;  // multiset 是最好写的实现方式

bool cmp(int x, int y, int w) {
  return oldrk[x] == oldrk[y] && oldrk[x + w] == oldrk[y + w];
```

```
}

int main() {
  int i, j, w, p, m = 1000000;

  scanf("%d%d", &n, &k);
  --k;

  for (i = 1; i <= n; ++i) scanf("%d", a + i);  // 求后缀数组
  for (i = 1; i <= n; ++i) ++cnt[rk[i] = a[i]];
  for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
  for (i = n; i >= 1; --i) sa[cnt[rk[i]]--] = i;

  for (w = 1; w < n; w <<= 1, m = p) {
    for (p = 0, i = n; i > n - w; --i) id[++p] = i;
    for (i = 1; i <= n; ++i)
      if (sa[i] > w) id[++p] = sa[i] - w;
    memset(cnt, 0, sizeof(cnt));
    for (i = 1; i <= n; ++i) ++cnt[px[i] = rk[id[i]]];
    for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
    for (i = n; i >= 1; --i) sa[cnt[px[i]]--] = id[i];
    memcpy(oldrk, rk, sizeof(rk));
    for (p = 0, i = 1; i <= n; ++i)
      rk[sa[i]] = cmp(sa[i], sa[i - 1], w) ? p : ++p;
  }

  for (i = 1, j = 0; i <= n; ++i) {  // 求 height
    if (j) --j;
    while (a[i + j] == a[sa[rk[i] - 1] + j]) ++j;
    ht[rk[i]] = j;
  }

  for (i = 1; i <= n; ++i) {  // 求所有最小值的最大值
    t.insert(ht[i]);
    if (i > k) t.erase(t.find(ht[i - k]));
    ans = max(ans, *t.begin());
  }

  cout << ans;

  return 0;
}
```

是否有某字符串在文本串中至少不重叠地出现了两次

可以二分目标串的长度 $|s|$，将 h 数组划分成若干个连续 LCP 大于等于 $|s|$ 的段，利用 RMQ 对每个段求其中出现的数中最大和最小的下标，若这两个下标的距离满足条件，则一定有长度为 $|s|$ 的字符串不重叠地出现了两次。

连续的若干个相同子串

我们可以枚举连续串的长度 $|s|$，按照 $|s|$ 对整个串进行分块，对相邻两块的块首进行 LCP 与 LCS 查询，具体可见[2009] 后缀数组 —— 处理字符串的有力工具。

例题：「NOI2016」 优秀的拆分[7]。

结合并查集

某些题目求解时要求你将后缀数组划分成若干个连续 LCP 长度大于等于某一值的段，亦即将 h 数组划分成若干个连续最小值大于等于某一值的段并统计每一段的答案。如果有多次询问，我们可以将询问离线。观察到当给定值单调递减的时候，满足条件的区间个数总是越来越少，而新区间都是两个或多个原区间相连所得，且新区间中不包含在原区间内的部分的 h 值都为减少到的这个值。我们只需要维护一个并查集，每次合并相邻的两个区间，并维护统计信息即可。

经典题目：「NOI2015」 品酒大会[8]

结合线段树

某些题目让你求满足条件的前若干个数，而这些数又在后缀排序中的一个区间内。这时我们可以用归并排序的性质来合并两个结点的信息，利用线段树维护和查询区间答案。

结合单调栈

例题：「AHOI2013」 差异[9]

" 题解"

被加数的前两项很好处理，为 $n(n-1)(n+1)/2$（每个后缀都出现了 $n-1$ 次，后缀总长是 $n(n+1)/2$），关键是最后一项，即后缀的两两 LCP。

我们知道 $lcp(i,j)=k$ 等价于 $\min\{height[i+1..j]\}=k$。所以，可以把 $lcp(i,j)$ 记作 $\min\{x|i+1\le x\le j, height[x]=lcp(i,j)\}$ 对答案的贡献。

考虑每个位置对答案的贡献是哪些后缀的 LCP，其实就是从它开始向左若干个连续的 $height$ 大于它的后缀中选一个，再从向右若干个连续的 $height$ 不小于它的后缀中选一个。这个东西可以用 单调栈 计算。

单调栈部分类似于 Luogu P2659 美丽的序列[10] 以及 悬线法。

" 参考代码"

```cpp
#include <cstdio>
#include <cstring>
#include <iostream>

using namespace std;

const int N = 500010;

char s[N];
int n, sa[N], rk[N << 1], oldrk[N << 1], id[N], px[N], cnt[N], ht[N], sta[N],
    top, l[N];
long long ans;

bool cmp(int x, int y, int w) {
  return oldrk[x] == oldrk[y] && oldrk[x + w] == oldrk[y + w];
}

int main() {
  int i, k, w, p, m = 300;

  scanf("%s", s + 1);
  n = strlen(s + 1);
  ans = 1ll * n * (n - 1) * (n + 1) / 2;
  // 求后缀数组
```

```
  for (i = 1; i <= n; ++i) ++cnt[rk[i] = s[i]];
  for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
  for (i = n; i >= 1; --i) sa[cnt[rk[i]]--] = i;

  for (w = 1; w < n; w <<= 1, m = p) {
    for (p = 0, i = n; i > n - w; --i) id[++p] = i;
    for (i = 1; i <= n; ++i)
      if (sa[i] > w) id[++p] = sa[i] - w;
    memset(cnt, 0, sizeof(cnt));
    for (i = 1; i <= n; ++i) ++cnt[px[i] = rk[id[i]]];
    for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
    for (i = n; i >= 1; --i) sa[cnt[px[i]]--] = id[i];
    memcpy(oldrk, rk, sizeof(rk));
    for (p = 0, i = 1; i <= n; ++i)
      rk[sa[i]] = cmp(sa[i], sa[i - 1], w) ? p : ++p;
  }
  // 求 height
  for (i = 1, k = 0; i <= n; ++i) {
    if (k) --k;
    while (s[i + k] == s[sa[rk[i] - 1] + k]) ++k;
    ht[rk[i]] = k;
  }
  // 维护单调栈
  for (i = 1; i <= n; ++i) {
    while (ht[sta[top]] > ht[i]) --top;  // top 类似于一个指针
    l[i] = i - sta[top];
    sta[++top] = i;
  }
  // 最后利用单调栈算 ans
  sta[++top] = n + 1;
  ht[n + 1] = -1;
  for (i = n; i >= 1; --i) {
    while (ht[sta[top]] >= ht[i]) --top;
    ans -= 2ll * ht[i] * l[i] * (sta[top] - i);
    sta[++top] = i;
  }

  cout << ans;

  return 0;
}
```

类似的题目：「HAOI2016」找相同字符[11]。

习题

- Uva 760 - DNA Sequencing[12]
- Uva 1223 - Editor[13]
- Codechef - Tandem[14]
- Codechef - Substrings and Repetitions[15]
- Codechef - Entangled Strings[16]
- Codeforces - Martian Strings[17]

- Codeforces - Little Elephant and Strings[18]
- SPOJ - Ada and Terramorphing[19]
- SPOJ - Ada and Substring[20]
- UVa - 1227 - The longest constant gene[21]
- SPOJ - Longest Common Substring[22]
- UVa 11512 - GATTACA[23]
- LA 7502 - Suffixes and Palindromes[24]
- GYM - Por Costel and the Censorship Committee[25]
- UVa 1254 - Top 10[26]
- UVa 12191 - File Recover[27]
- UVa 12206 - Stammering Aliens[28]
- Codechef - Jarvis and LCP[29]
- LA 3943 - Liking's Letter[30]
- UVa 11107 - Life Forms[31]
- UVa 12974 - Exquisite Strings[32]
- UVa 10526 - Intellectual Property[33]
- UVa 12338 - Anti-Rhyme Pairs[34]
- DevSkills Reconstructing Blue Print of Life[35]
- UVa 12191 - File Recover[27]
- SPOJ - Suffix Array[36]
- LA 4513 - Stammering Aliens[37]
- SPOJ - LCS2[38]
- Codeforces - Fake News (hard)[39]
- SPOJ - Longest Commong Substring[40]
- SPOJ - Lexicographical Substring Search[41]
- Codeforces - Forbidden Indices[42]
- Codeforces - Tricky and Clever Password[43]
- LA 6856 - Circle of digits[44]

参考资料

本页面中（4070a9b[45] 引入的部分）主要译自博文 [46] 与其英文翻译版 Suffix Array[47]。其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0。

论文：

1. [2004] 后缀数组 by. 许智磊
2. [2009] 后缀数组 —— 处理字符串的有力工具 by. 罗穗骞

参考资料与注释

[1] LOJ #111: 后缀排序

[2] 诱导排序与 SA-IS 算法

[3] 评论页面

[4] 「JSOI2007」字符加密

[5] 「USACO07DEC」Best Cow Line

[6] 「USACO06DEC」Milk Patterns

[7] 「NOI2016」优秀的拆分

[8] 「NOI2015」品酒大会

[9] 「AHOI2013」差异

[10] Luogu P2659 美丽的序列

[11] 「HAOI2016」找相同字符

[12] Uva 760 - DNA Sequencing

[13] Uva 1223 - Editor

[14] Codechef - Tandem

[15] Codechef - Substrings and Repetitions

[16] Codechef - Entangled Strings

[17] Codeforces - Martian Strings

[18] Codeforces - Little Elephant and Strings

[19] SPOJ - Ada and Terramorphing

[20] SPOJ - Ada and Substring

[21] UVa - 1227 - The longest constant gene

[22] SPOJ - Longest Common Substring

[23] UVa 11512 - GATTACA

[24] LA 7502 - Suffixes and Palindromes

[25] GYM - Por Costel and the Censorship Committee

[26] UVa 1254 - Top 10

[27] UVa 12191 - File Recover [27-1] [27-2]

[28] UVa 12206 - Stammering Aliens

[29] Codechef - Jarvis and LCP

[30] LA 3943 - Liking's Letter

[31] UVa 11107 - Life Forms

[32] UVa 12974 - Exquisite Strings

[33] UVa 10526 - Intellectual Property

[34] UVa 12338 - Anti-Rhyme Pairs

[35] DevSkills Reconstructing Blue Print of Life

[36] SPOJ - Suffix Array

[37] LA 4513 - Stammering Aliens

[38] SPOJ - LCS2

[39] Codeforces - Fake News (hard)

[40] SPOJ - Longest Commong Substring

[41] SPOJ - Lexicographical Substring Search

[42] Codeforces - Forbidden Indices

[43] Codeforces - Tricky and Clever Password

[44] LA 6856 - Circle of digits

[45] 4070a9b

[46]

[47] Suffix Array

8.12.2 最优原地后缀排序算法

本章介绍线性时间复杂度的后缀排序的就地算法[1-1]（Optimal In-Place Suffix Sorting）。

warning

本章**只建议在非常非常熟悉** SA-IS[2][3] 的前提下阅读。

全局设定

目标字符串 `Pat`，后缀数组 `SA`，串的序号从 0 开始，结尾字符是警戒哨，不妨设为 0。

在整形字母表上的后缀排序

事实上这一部分可以看成是原地版本的 SA-IS 算法。

因为是原文中细节相对最清楚，实现也较为简单的算法，也是了解后续算法的基础，是本文介绍的重点。

原地化的原理是用重命名的 `Pat` 代替 S、L 桶，用额外 $O(n)$ 的操作代替类型桶。

重命名目标串 Pat

简单来说，我们会在不改变后缀大小的相对顺序的前提下，重命名 `Pat`，用重命名后的 `Pat` 来取代原来 S、L 桶，来指明桶头或者桶尾。

重命名的方法是将 `Pat` 中的 S 型字符替换为所在桶的桶尾索引，L 型字符替换为所在桶的桶头索引。

如下图所示：

```
 Index :    0  1  2  3  4  5  6  7  8  9  10 11 12
   Pat :    2  1  1  3  3  1  1  3  3  1  2  1  0
  Type :    L  S  S  L  L  S  S  L  L  S  L  L  S
Bucket :   (0) (1    1    1    1    1    1) (2    2) (3    3    3    3)
```

重命名后的 `Pat'`（之后直接将重命名后的 `Pat'` 称做 `Pat`）：

```
Index :    0  1  2  3  4  5  6  7  8  9  10 11 12
 Pat' :    7  6  6  9  9  6  6  9  9  6  7  1  0
```

由于桶内的字符，L 型字符后缀小，作为桶头；而 S 型字符后缀大，作为桶尾，因此保持了后缀大小的相对顺序。

描述一下重命名的具体步骤：

1. 和 SA-IS 一样，对 `Pat` 中每个字符计数，计算其前缀和（计数排序），来构建 S/L 桶，只不过这里用 `SA` 盛放这个前缀和；
2. 从尾到头，扫描 `Pat` 的每个字符，这样只需记录上一个字符的类型，就可以动态地判断每个字符的类型，然后依据前缀和将其重命名。

对 LMS 字符排序

这里重点是使用了一个内部计数器的技巧。

初始化

初始的时候将 `SA` 每一项设为 E（EMPTY）。

从尾到头扫描 `Pat`，如果发现是 LMS 字符，`Pat[i]`，那么就设置 `SA[Pat[i]]` 的标记：

如果 SA[Pat[i]] 是 E，就将其设为 U（UNIQUE）；

如果 SA[Pat[i]] 是 U，就将其设为 M（MULTIPLE）；

其他情况，不做处理。

结果如下图所示：

```
Index :   0 1 2 3 4 5 6 7 8 9 10 11 12
  Pat :   7 6 6 9 9 6 6 9 9 6 7 1 0
  LMS :                 *           *         *
   SA : (U) (E) (E   E   E   E   M) (E   E) (E   E   E   E)
```

把 LMS 字符的索引放入 SA

从尾到头扫描 Pat，对于 LMS 字符 Pat[i]，根据 SA[Pat[i]] 的符号进行分类讨论：

U：直接让 SA[Pat[i]] = i

M：意味着桶中有至少两个 LMS 字符。

1. 如果桶中有至少三个 LMS 字符：就把桶中倒数第二个位置作为临时计数器，标志桶中已填充的 LMS 字符数（桶中倒数第一位就是标志 M）将新的 LMS 字符从倒数第三个位置开始插入，让临时计数器自增 1。如果发现桶已经满了，就把桶中从桶头到倒数第三个的所有元素向右平移 2 个位置，然后把新元素插入到桶中第二个位置（桶中第一个位置填为 E）
2. 如果桶中有且只有 2 个 LMS 字符，显然不需要计数器，直接从右到左顺序插入即可。

正常的值：

```
根据我们之前的讨论，此时不管桶中有两个还是两个以上的 LMS 字符，这都意味着 $\texttt
{i}$ 是桶中最后一个待插入的 LMS 字符的位置，

只需要从桶头开始向左扫描，找到第一个标记为 E 的位置，将其设为 $\texttt{i}$。
```

最后要从尾到头扫描一遍 SA，清除可能残余的特殊符号 M（桶中未被填满，所以 M 和计数器未被覆盖）。

方法是将桶中 LMS 字符如上述步骤一样向右平移 2 位，将左边空出来的位置填为 E。

如下图所示：

```
Index :    0 1 2 3 4 5 6 7 8 9 10 11 12
  Pat :    7 6 6 9 9 6 6 9 9 6 7 1 0
   SA : (12) (E) (E   E   E   E   M) (E   E) (E   E   E   E)
   SA : (12) (E) (E   E   9   1   M) (E   E) (E   E   E   E)
   SA : (12) (E) (E   5   9   2   M) (E   E) (E   E   E   E)
   SA : (12) (E) (1   5   9   3   M) (E   E) (E   E   E   E)
   SA : (12) (E) (E   E   1   5   9) (E   E) (E   E   E   E)
```

这个阶段，由于每个桶只需要被移动和扫描一次，所以时间复杂度是 $O(n)$。

诱导排序 LMS 子串

诱导排序 LMS 前缀

将 LMS 前缀进行诱导排序，同 SA-IS 一样，这部分同后面对后缀的诱导排序完全一样（使用同一个函数），因此这里直接跳过。

这里直接给出排序结果：

```
Index :    0 1 2 3 4 5 6 7 8 9 10 11 12
   SA : (12)(11) (1   5   9   2   6) (10   0) (4   8   3   7)
```

将已排序的 LMS 子串放到 SA 尾部

```
Index :   0  1  2  3  4  5  6  7  8  9  10 11 12
   SA :   E  E  E  E  E  E  E  E  E  12 1  5  9
```

构建规模缩减的子目标串 Pat1

从左到右扫描 SA 尾部的 LMS 子串，确定其大小关系「重命名」，将 SA[i] 重命名的值存储在 SA $\left[\left\lfloor \frac{SA[i]}{2} \right\rfloor\right]$。

因为 LMS 字符并不相邻，所以不会有冲突，这样做是将重命名后的值按照所代表的子串在 Pat 中的原顺序放置：

```
Index :   0  1  2  3  4  5  6  7  8  9  10 11 12
   SA :   1  E  1  E  2  E  0  E  E  12 1  5  9
```

然后扫描 SA，收集这些重命名的值到 SA 头部：

```
Index :   0  1  2  3  4  5  6  7  8  9  10 11 12
   SA :   1  1  2  0  E  E  E  E  E  12 1  5  9
```

通过递归解决 Pat1，完成对 LMS 后缀的排序

同 SA-IS 一样，递归解决 SA 头部的规模缩减的 Pat1 的后缀排序，结果存到 SA 尾部：

```
Index :   0  1  2  3  4  5  6  7  8  9  10 11 12
   SA :   1  1  2  0  E  E  E  E  E  3  0  1  2
```

将 SA 尾部的 SA1 挪到 SA 头部，重新从尾到头扫描 Pat，将其中 LMS 字符按照在 Pat 中的顺序放到 SA 尾部：

```
Index :   0  1  2  3  4  5  6  7  8  9  10 11 12
   SA :   3  0  1  2  E  E  E  E  E  1  5  9  12
```

依照 SA 尾部的「对照表」，将 SA1 头部的 SA 还原为 Pat 中对应的 LMS 后缀的索引位置：

```
Index :   0  1  2  3  4  5  6  7  8  9  10 11 12
   SA :   12 1  5  9  E  E  E  E  E  1  5  9  12
```

将 SA 头部的排好序的 LMS 后缀按顺序放入到对应的桶中（从尾部开始放）：

```
Index :    0    1    2  3  4  5  6    7  8    9  10 11 12
   SA :   (12) (E) (E  E  1  5  9) (E  E) (E  E  E  E)
```

对 Pat1 中所有的后缀进行诱导排序

这一部分就是利用前面用过的内部计数器技巧，进行原地版的诱导排序。

假如我们已经有排好序的 LMS 后缀（在桶尾），来诱导 L 型后缀[4]：

```
Index :    0    1    2  3  4  5  6    7  8    9  10 11 12
  Pat :    7    6    6  9  9  6  6    9  9    6  7  1  0
   SA :   (12) (E) (E  E  1  5  9) (E  E) (E  E  E  E)
```

如同排序 LMS 字符一样，先对 L 型字符用特殊符号计数：

```
Index :    0    1    2  3  4  5  6    7  8    9  10 11 12
  Pat :    7    6    6  9  9  6  6    9  9    6  7  1  0
   SA :   (12) (U) (E  E  1  5  9) (M  E) (M  E  E  E)
```

从左到右扫描 SA，同对 LMS 字符排序一样，复杂一点的是判断 `suf[SA[i] - 1]` 的类型，需要分类讨论（详情参考代码）：

```
Index :    0 1 2 3 4 5 6 7 8 9 10 11 12
   SA :  (12)(11) (E E 1 5 9) (M E) (M E E E)
   SA :  (12)(11) (E E 1 5 9)(10   E) (M E E E)
   SA :  (12)(11) (E E   1   5 9)(10   0) (M E E E)
   SA :  (12)(11) (E E 1   5   9)(10 0) (M 1   4   E)
   SA :  (12)(11) (E E 1 5   9)(10 0) (M 2   4   8)
   SA :  (12)(11) (E E 1 5 9)(10 0) (4   8   3   E)
   SA :  (12)(11) (E E 1 5 9)(10 0) (4   8   3   7)
```

区别于 SA-IS 的是，对一个类型字符诱导排序后，需要清理 LMS 字符以免对后面的原地诱导排序：

```
Index :    0 1 2 3 4 5 6 7 8 9 10 11 12
   SA :  (12)(11) (E E   E   E   E)(10 0) (4 8 3 7)
```

至于从 L 后缀诱导 S 后缀与从 LMS 后缀诱导 L 后缀完全对称，这里就不做多余介绍。

到这儿为止，诱导排序就完成了。

实现

时间性能上和 SA-IS 没有显著差别，空间占用变为不到原来的 $\frac{1}{3}$（代码量多 1 倍），算是不愧为原文 Optimal In-Place Suffix Sorting[1-2] 的标题。

" 参考代码"

```rust
use std::cmp::max;
use std::cmp::Ordering;
use std::slice::from_raw_parts_mut;


const LTYPE: bool = false;
const STYPE: bool = true;
const MAX_SA_VALUE: usize = usize::MAX / 2;
const EMPTY: usize = MAX_SA_VALUE + 1;
const UNIQUE: usize = MAX_SA_VALUE + 2;
const MULTI: usize = MAX_SA_VALUE + 3;  // >= 258


fn lms_str_cmp<E: Ord>(l1: &[E], l2: &[E]) -> Ordering {
    for (x, y) in l1.iter().zip(l2.iter()) {
        let cmp_res = x.cmp(&y);

        if cmp_res != Ordering::Equal { return cmp_res; }
    }

    Ordering::Equal
}

#[inline]
fn pat_char_type(cur: usize, prev: usize, last_scanned_type: bool) -> bool {
    if cur < prev || cur == prev && last_scanned_type == STYPE { STYPE }
    else { LTYPE }
```

```
}


fn rename_pat(pat: &mut [usize], sa: &mut [usize]) {
    let patlastpos = pat.len() - 1;
    // 全部刷成 bucket head
    //sa.fill(0);
    for i in 0..sa.len() { sa[i] = 0 }

    for i in 0..pat.len() { sa[pat[i]] += 1 }
    for i in 1..sa.len() { sa[i] += sa[i - 1] }

    for i in 0..pat.len() - 1 {
        pat[i] = sa[pat[i]] - 1;
    };
    // 将 L-suffix 刷成 bucket head
    //sa.fill(0);
    for i in 0..sa.len() { sa[i] = 0 }

    for i in 0..pat.len() { sa[pat[i]] += 1 }
    let mut last_scanned_type = STYPE;
    pat[patlastpos] = 0;
    for i in (0..pat.len() - 1).rev() {
        if pat_char_type(pat[i], pat[i + 1], last_scanned_type) == STYPE {
            last_scanned_type = STYPE;
        } else {
            pat[i] -= sa[pat[i]] - 1;
            last_scanned_type = LTYPE;
        }
    }

}


fn sort_lms_char(pat: &mut [usize], sa: &mut [usize]) -> usize {
    //sa.fill(EMPTY);
    for i in 0..sa.len() { sa[i] = EMPTY }

    let mut last_scanned_type = STYPE;
    for i in (0..pat.len() - 1).rev() {
        if pat_char_type(pat[i], pat[i + 1], last_scanned_type) == STYPE {
            last_scanned_type = STYPE;
        } else {
            if last_scanned_type == STYPE {  // pat[i + 1] is LMS type
                sa[pat[i + 1]] += 1;
            }

            last_scanned_type = LTYPE;
        }
    }

    let mut lms_cnt = 0;
    last_scanned_type = STYPE;
    for i in (0..pat.len() - 1).rev() {
```

```
        if pat_char_type(pat[i], pat[i + 1], last_scanned_type) == STYPE {
            last_scanned_type = STYPE;
        } else {
            let e_i = i + 1;
            let e = pat[e_i];

            if last_scanned_type == STYPE {  // pat[i + 1] is LMS type
                lms_cnt += 1;
                if sa[e] == UNIQUE {
                    sa[e] = e_i;
                } else if sa[e] >= MULTI && sa[e - 1] == EMPTY {
                    if sa[e - 2] == EMPTY {
                        sa[e - 2] = e_i;
                        sa[e - 1] = 1;  // set counter
                    } else {  // MUL = 2
                        sa[e] = e_i;
                        sa[e - 1] = EMPTY;
                    }
                } else if sa[e] >= MULTI && sa[e - 1] != EMPTY {
                    let c = sa[e - 1];  // get counter

                    if sa[e - 2 - c] == EMPTY {
                        sa[e - 2 - c] = e_i;
                        sa[e - 1] += 1;  // update counter
                    } else {
                        for j in (1..c + 1).rev() {
                            sa[e - c + j] = sa[e - 2 - c + j]
                        }
                        sa[e - c] = e_i;
                        sa[e - c - 1] = EMPTY;
                    }
                } else if sa[e] < EMPTY {
                    for j in (0..e).rev() {
                        if sa[j] == EMPTY {
                            sa[j] = e_i;
                            break;
                        }
                    }
                }
            }

            last_scanned_type = LTYPE;
        }
    }

    for i in (0..pat.len()).rev() {
        if sa[i] >= MULTI {
            let c = sa[i - 1];
            for j in (1..c + 1).rev() {  // 逆序防止前面的覆盖后面的
                sa[i - c + j] = sa[i - 2 - c + j];
            }
            sa[i - c - 1] = EMPTY;
            sa[i - c] = EMPTY;
        }
```

```
    }

    lms_cnt
}


fn sort_lms_substr(pat: &mut [usize], sa: &mut [usize]) {
    // step 1
    induced_sort(pat, sa);

    // step 2
    let pat_last_pos = pat.len() - 1;
    let mut lms_cnt = 0;
    let mut i = pat_last_pos;
    let mut bucket_tail_ptr = pat_last_pos + 1;  // for renamed bucket ver
    let mut bucket = EMPTY;  // 可以省略，但是为了书写代码方便
    let mut num = 0;  // S type number of bucket
    while i > 0 {
        if pat[sa[i]] != bucket {  // reach new bucket
            num = 0;

            let mut l = 0;
            while pat[sa[i - l]] == pat[sa[i]] {  // 扫描桶来计算桶中 S 字符数量，根
据定义 当l=i 时循环必然终止
                let pat_i = sa[i - l];             // l < i, 即 i - l > 0, 0 <=
pat_i < patlen - 1
                if pat[pat_i] < pat[pat_i + 1] {
                    let mut k = pat_i;
                    while k > 0 && pat[k - 1] == pat[pat_i] { k -= 1 }
                    num += pat_i - k + 1;
                } else {
                    break;   // bucket 不含 S 字符，结束扫描
                }

                l += 1;
            }

            bucket_tail_ptr = i;
            bucket = pat[sa[bucket_tail_ptr]];
        }

        if num > 0
        && i > bucket_tail_ptr - num
        && sa[i] > 0
        && pat[sa[i]] < pat[sa[i] - 1]  {
            sa[pat_last_pos - lms_cnt] = sa[i];
            lms_cnt += 1;
        }

        i -= 1;
    }

    sa[pat_last_pos - lms_cnt ] = sa[i];  // i = 0
    lms_cnt += 1;
```

```
    //sa[0..pat_last_pos - lms_cnt + 1].fill(EMPTY);
    for i in 0..pat_last_pos - lms_cnt + 1 { sa[i] = EMPTY }
}


fn construct_pat1(pat: &mut [usize], sa: &mut [usize], lms_cnt: usize) -> bool {
    let patlen = pat.len();

    let mut prev_lms_str_len = 1;
    let mut rank = 0;
    sa[(patlen - 1) / 2] = rank;
    let mut has_duplicated_char = false;
    for i in patlen - lms_cnt + 1..patlen {  // 从警戒哨字符的下一个字符开始
        let mut j = sa[i];
        while pat[j] <= pat[j + 1] { j += 1 } // 寻找 suf(sa[i]) 右边第一个 L 字符, 因
为排除了警戒哨这个LMS 后缀, 所以必然不会越界
        let mut k = j;
        while k + 1 < patlen && pat[k] >= pat[k + 1] { k += 1 }  // 找到 suf(sa[i
])右边第一个LMS 字符
        let cur_lms_str_len = k + 1 - sa[i];
        let cmp_res = lms_str_cmp(&pat[sa[i]..sa[i] + cur_lms_str_len], &pat[sa[
i - 1]..sa[i - 1] + prev_lms_str_len]);

        if  cmp_res != Ordering::Equal {
            rank += 1
        }

        if rank == sa[sa[i - 1] / 2] {
            has_duplicated_char = true;
        }
        let rank_index = sa[i] / 2;
        sa[rank_index] = rank;  // 整除

        prev_lms_str_len = cur_lms_str_len;
    }

    // move to head of sa
    let mut j = 0;
    for i in 0..patlen - lms_cnt {
        if sa[i] != EMPTY {
            sa[j] = sa[i];
            if i > j {
                sa[i] = EMPTY;
            }
            j += 1;
        }
    }
    //sa[lms_cnt..patlen].fill(EMPTY);
    for i in lms_cnt..patlen { sa[i] = EMPTY }

    has_duplicated_char
}

fn sort_lms_suf(pat: &mut [usize], sa: &mut [usize], lms_cnt: usize, has_duplica
```

```
ted_char: bool) {
    // solve T1 recursively
    let patlen = pat.len();
    let salen = sa.len();
    unsafe {
        let sa_ptr = sa.as_mut_ptr();
        let mut pat1 = from_raw_parts_mut(sa_ptr, lms_cnt);
        let mut sa1 = from_raw_parts_mut(sa_ptr.offset((patlen - lms_cnt) as isi
ze), salen - (patlen - lms_cnt));

        if has_duplicated_char {
            _compute_suffix_array_16_1(&mut pat1, &mut sa1);
        } else {
            for i in 0..lms_cnt { sa1[pat1[i]] = i }
        }
    }

    // move SA1 to SA[0...n1-1]
    for i in 0..lms_cnt {
        sa[i] = sa[patlen- lms_cnt + i];
    }

    // put all LMS-suffixes in SA tail
    let mut last_scanned_type = STYPE;
    let mut j = 0;
    for i in (0..pat.len() - 1).rev() {
        if pat[i] < pat[i + 1] || pat[i] == pat[i + 1] && last_scanned_type == S
TYPE {
            last_scanned_type = STYPE;
        } else {
            if last_scanned_type == STYPE {
                sa[patlen - 1 - j] = i + 1;
                j += 1;
            }

            last_scanned_type = LTYPE;
        }
    }

    // backward map the LMS-suffixes rank
    for i in 0..lms_cnt {
        let relative_rank = sa[i];
        sa[i] = sa[patlen - lms_cnt + relative_rank];
        sa[patlen - lms_cnt + relative_rank] = EMPTY;
    }

    let mut tail = EMPTY;
    let mut rfp = EMPTY;
    for i in (1..lms_cnt).rev() { // sa[0] 保持原位
        if pat[sa[i]] != tail {
            tail = pat[sa[i]];
            rfp = tail;
        }
```

```
        sa[rfp] = sa[i];
        if rfp != i { sa[i] = EMPTY }
        rfp -= 1;
    }
}

// PASS!
fn induced_sort(pat: &mut [usize], sa: &mut [usize]) {
    let patlen = pat.len();

    // place L-suff in SA
    // init
    let mut last_scanned_type = STYPE;
    for i in (0..patlen - 1).rev() {
        if pat_char_type(pat[i], pat[i + 1], last_scanned_type) == LTYPE {
            sa[pat[i]] += 1;  // >= EMPTY
            last_scanned_type = LTYPE;
        } else {
            last_scanned_type = STYPE;
        }
    }
    //place
    let mut i = 0;
    while i < patlen {
        if sa[i] < EMPTY && sa[i] > 0 {
            let j = sa[i] - 1;
            let mut is_ltype = false;
            if pat[j] > pat[j + 1] {
                is_ltype = true;
            } else if pat[j] == pat[j + 1] {  // 判断 sa[i] 是否是 L 后缀的编号
                let next_i = sa[pat[sa[i]]];
                if next_i >= MULTI {
                    is_ltype = true;
                } else if next_i < EMPTY && pat[sa[i]] + 1 < patlen {
                    if sa[pat[sa[i]] + 1] == EMPTY {
                        is_ltype = true;
                    } else if sa[pat[sa[i]] + 1] < EMPTY {
                        if pat[sa[pat[sa[i]] + 1]] == pat[sa[i]] {
                            is_ltype = true;
                        }
                    }
                }
            }

            if is_ltype {
                if sa[pat[j]] == UNIQUE {
                    sa[pat[j]] = j;
                } else if sa[pat[j]] >= MULTI && sa[pat[j] + 1] == EMPTY {
                    if sa[pat[j]] - EMPTY > 2 {
                        sa[pat[j] + 2] = j;
                        sa[pat[j] + 1] = 1;  // set counter
                    } else {
                        sa[pat[j]] = j;
                    }
```

```
            } else if sa[pat[j]] >= MULTI && sa[pat[j] + 1] != EMPTY {
                let e = pat[j];
                let c = sa[e + 1];
                let lfp = e + c + 2;
                if  c + 2 < sa[pat[j]] - EMPTY {  // 没到 bucket 尾部
                    sa[lfp] = j;
                    sa[e + 1] += 1;  // update counter
                } else {
                    for k in 1..c + 1 {
                        sa[e + k - 1] = sa[e + k + 1];
                    }
                    sa[e + c] = j;
                    sa[e + c + 1] = EMPTY;
                    if i >= e + 2 && i <= e + c + 1 {
                        i -= 2;
                    }
                }
            } else if sa[pat[j]] < EMPTY {
                for k in pat[j]..patlen {
                    if sa[k] == EMPTY {
                        sa[k] = j;
                        break;
                    }
                }
            }
        }
    } else if sa[i] >= MULTI {
        i += 1;
    }

    i += 1;
}

// remove LMS-suff form SA, 一个桶里可能有多个 LMS 后缀
last_scanned_type = STYPE;
for i in (0..pat.len() - 1).rev() {
    if pat_char_type(pat[i], pat[i + 1], last_scanned_type) == STYPE {
        last_scanned_type = STYPE;
    } else {
        if last_scanned_type == STYPE {  // pat[i + 1] is LMS type
            if sa[pat[i + 1]] <= EMPTY {
                sa[pat[i + 1]] = UNIQUE;
            } else {
                sa[pat[i + 1]] += 1;
            }
        }

        last_scanned_type = LTYPE;
    }
}
i = patlen - 1;
while i > 0 {
    if sa[i] > EMPTY {
        let c = sa[i] - EMPTY;
```

```rust
            for k in 0..c {
                sa[i - k] = EMPTY;
            }
            i -= c - 1;
        }

        i -= 1;
    }
    sa[0] = pat.len() - 1;

    // place S-suff in SA
    // init
    let mut last_scanned_type = STYPE;
    for i in (0..patlen - 1).rev() {
        if pat_char_type(pat[i], pat[i + 1], last_scanned_type) == STYPE {
            if sa[pat[i]] >= EMPTY {
                sa[pat[i]] += 1;
            } else {
                sa[pat[i]] = UNIQUE;
            }
            last_scanned_type = STYPE;
        } else {
            last_scanned_type = LTYPE;
        }
    }
    i = patlen - 1;
    while i > 0 {
        if sa[i] < EMPTY && sa[i] > 0 {
            let j = sa[i] - 1;
            let mut is_stype = false;
            if pat[j] < pat[j + 1] {
                is_stype = true;
            } else if pat[j] == pat[j + 1] {  // 判断 sa[i] 是否是 S 后缀的编号
                let next_i = sa[pat[sa[i]]];
                if next_i >= MULTI {
                    is_stype = true;
                } else if next_i < EMPTY && pat[sa[i]] - 1 > 0 {
                    if sa[pat[sa[i]] - 1] == EMPTY {
                        is_stype = true;
                    } else if sa[pat[sa[i]] - 1] < EMPTY {
                        if pat[sa[pat[sa[i]] - 1]] == pat[sa[i]] {
                            is_stype = true;
                        }
                    }
                }
            }

            if is_stype {
                if sa[pat[j]] == UNIQUE {
                    sa[pat[j]] = j;
                } else if sa[pat[j]] >= MULTI && sa[pat[j] - 1] == EMPTY {
                    if sa[pat[j]] - EMPTY > 2 {
                        sa[pat[j] - 2] = j;
                        sa[pat[j] - 1] = 1;  // set counter
```

```
                    } else {
                        sa[pat[j]] = j;
                    }
                } else if sa[pat[j]] >= MULTI && sa[pat[j] - 1] != EMPTY {
                    let e = pat[j];
                    let c = sa[e - 1];
                    let num = sa[pat[j]] - EMPTY;
                    if c + 2 < num {  // 没到 bucket 头部
                        let rfp = e - c - 2;
                        sa[rfp] = j;
                        sa[e - 1] += 1;
                    } else {
                        for k in 1..c + 1 {
                            sa[e - k + 1] = sa[e - k - 1];
                        }
                        sa[e - c] = j;
                        sa[e - c - 1] = EMPTY;
                        if i >= e - num + 1 && i <= e - 2 {
                            i += 2;
                        }
                    }
                } else if sa[pat[j]] < EMPTY {
                    for k in (0..pat[j]).rev() {
                        if sa[k] == EMPTY {
                            sa[k] = j;
                            break;
                        }
                    }
                }
            }
        } else if sa[i] >= MULTI {
            i -= 1;
        }
        i -= 1;
    }
}

fn _compute_suffix_array_16_1(pat: &mut [usize], sa: &mut [usize]) {
    rename_pat(pat, sa);
    let lms_cnt = sort_lms_char(pat, sa);
    sort_lms_substr(pat, sa);
    let has_duplicated_char = construct_pat1(pat, sa, lms_cnt);
    sort_lms_suf(pat, sa, lms_cnt, has_duplicated_char);
    induced_sort(pat, sa);
}

pub fn suffix_array_16(pat: &[u8]) -> Vec<usize> {
    let mut pat = pat.into_iter().map(|x| *x as usize).collect::<Vec<usize>>();
    pat.push(0);
    let mut sa = vec![0; max(pat.len(), 256) * 1];
    _compute_suffix_array_16_1(&mut pat[..], &mut sa[..]);

    sa
}
```

```rust
fn input() -> String {
    use std::io;

    let mut input = String::new();
    io::stdin().read_line(&mut input).unwrap();
    String::from(input.trim())
}


fn main() {
    let pat = input();

    let sa_16 = suffix_array_16(pat.as_bytes());

    for i in 1..pat.len() + 1 { print!("{} ", sa_16[i] + 1) }
}
```

在只读的整形字母表上的后缀排序

使用复杂方法解决复杂问题，通过分治，解决空间紧张的问题。

算法实现的难点在于在 SA 上构建 BitMaps[5]，来替代本来由重命名后的 T 所指示的指示桶尾/桶头的位置。

这里的 BitMaps 指得是使用比特向量（bit vector）表示的有序字典（multiset），是一种紧凑型结构（compact data structure）。

有兴趣了解的暂时只能阅读原文以及本文引用的 BitMaps 的有关论文自行了解。

在只读的一般字母表上的后缀排序

前置知识是归并排序和堆排序。

由于笔者对于其中确定字符类型的方法的时间复杂度有疑问，这里也不再介绍，建议阅读原文自行了解。

注解

[1] Li, Zhize; Li, Jian; Huo, Hongwei (2016). *Optimal In-Place Suffix Sorting*. Proceedings of the 25th International Symposium on String Processing and Information Retrieval (SPIRE). Lecture Notes in Computer Science. 11147. Springer. pp. 268–284. arXiv:1610.08305. doi:10.1007/978-3-030-00479-8_22. ISBN:978-3-030-00478-1. [1-1] [1-2]

[2] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In Data Compression Conference (DCC), pages 193–202. IEEE, 2009.

[3] 推荐阅读 博文 和它的 issue 列表

[4] 如果是 LML 后缀，就先诱导 S 型后缀，唯一区别是计算 LML 后缀时需要将警戒哨也算进去。

[5] Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In Proc. 11th International Symposium on Experimental Algorithms (SEA), pages 295–306, 2012.

8.13 后缀自动机 (SAM)

Authors: abc1763613206, ksyx

一些前置约定/定义

记 Σ 为字符集，$|\Sigma|$ 为字符集大小。对于一个字符串 s，记 $|s|$ 为其长度。

后缀自动机概述

后缀自动机 (suffix automaton, SAM) 是一个能解决许多字符串相关问题的有力的数据结构。

举个例子，以下的字符串问题都可以在线性时间内通过 SAM 解决。

- 在另一个字符串中搜索一个字符串的所有出现位置。
- 计算给定的字符串中有多少个不同的子串。

直观上，字符串的 SAM 可以理解为给定字符串的**所有子串**的压缩形式。值得注意的事实是，SAM 将所有的这些信息以高度压缩的形式储存。对于一个长度为 n 的字符串，它的空间复杂度仅为 $O(n)$。此外，构造 SAM 的时间复杂度仅为 $O(n)$。准确地说，一个 SAM 最多有 $2n-1$ 个节点和 $3n-4$ 条转移边。

定义

字符串 s 的 SAM 是一个接受 s 的所有后缀的最小 **DFA**（确定性有限自动机或确定性有限状态自动机）。

换句话说：

- SAM 是一张有向无环图。结点被称作**状态**，边被称作状态间的**转移**。
- 图存在一个源点 t_0，称作**初始状态**，其它各结点均可从 t_0 出发到达。
- 每个**转移**都标有一些字母。从一个结点出发的所有转移均**不同**。
- 存在一个或多个**终止状态**。如果我们从初始状态 t_0 出发，最终转移到了一个终止状态，则路径上的所有转移连接起来一定是字符串 s 的一个后缀。s 的每个后缀均可用一条从 t_0 到某个终止状态的路径构成。
- 在所有满足上述条件的自动机中，SAM 的结点数是最少的。

子串的性质

SAM 最简单、也最重要的性质是，它包含关于字符串 s 的所有子串的信息。任意从初始状态 t_0 开始的路径，如果我们将转移路径上的标号写下来，都会形成 s 的一个**子串**。反之每个 s 的子串对应从 t_0 开始的某条路径。

为了简化表达，我们称子串**对应**一条路径（从 t_0 开始、由一些标号构成这个子串）。反过来，我们说任意一条路径**对应**它的标号构成的字符串。

到达某个状态的路径可能不止一条，因此我们说一个状态对应一些字符串的集合，这个集合的每个元素对应这些路径。

构建过程

我们将会在这里展示一些简单的字符串的后缀自动机。

我们用蓝色表示初始状态，用绿色表示终止状态。

对于字符串 $s=\emptyset$：

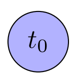


图 8.14

对于字符串 $s=\mathsf{a}$：

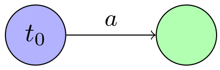


图 8.15

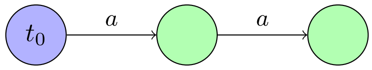


图 8.16

对于字符串 $s=$ aa：

对于字符串 $s=$ ab：

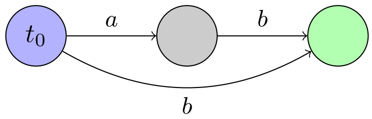


图 8.17

对于字符串 $s=$ abb：

对于字符串 $s=$ abbb：

在线性时间内构造

在我们描述线性时间内构造 SAM 的算法之前，我们需要引入几个对理解构造过程非常重要的概念并对其进行简单证明。

结束位置 endpos

考虑字符串 s 的任意非空子串 t，我们记 $\text{endpos}(t)$ 为在字符串 s 中 t 的所有结束位置（假设对字符串中字符的编号从零开始）。例如，对于字符串 abcbc，我们有 $\text{endpos}(\texttt{bc})=2,4$。

两个子串 t_1 与 t_2 的 endpos 集合可能相等：$\text{endpos}(t_1)=\text{endpos}(t_2)$。这样所有字符串 s 的非空子串都可以根据它们的 endpos 集合被分为若干**等价类**。

显然，SAM 中的每个状态对应一个或多个 endpos 相同的子串。换句话说，SAM 中的状态数等于所有子串的等价类的个数，再加上初始状态。SAM 的状态个数等价于 endpos 相同的一个或多个子串所组成的集合的个数 +1。

我们稍后将会用这个假设来介绍构造 SAM 的算法。我们将发现，SAM 需要满足的所有性质，除了最小性以外都满足了。由 Nerode 定理我们可以得出最小性（不会在这篇文章中证明）。

由 endpos 的值我们可以得到一些重要结论：

> **引理 1：** 字符串 s 的两个非空子串 u 和 w（假设 $|u|\le|w|$）的 endpos 相同，当且仅当字符串 u 在 s 中的每次出现，都是以 w 后缀的形式存在的。

引理显然成立。如果 u 和 w 的 endpos 相同，则 u 是 w 的一个后缀，且只以 s 中的一个 w 的后缀的形式出现。且根据定义，如果 u 为 w 的一个后缀，且只以后缀的形式在 s 中出现时，两个子串的 endpos 相同。

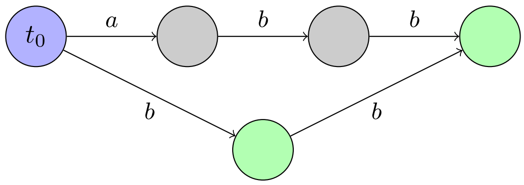


图 8.18

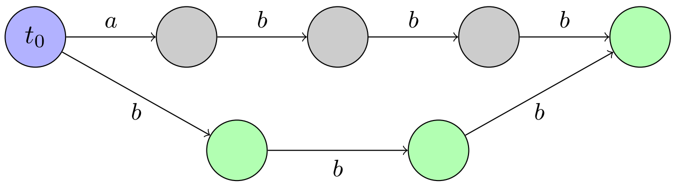


图 8.19

引理 2： 考虑两个非空子串 u 和 w（假设 $|u| \leq |w|$）。那么要么 $\mathrm{endpos}(u) \cap \mathrm{endpos}(w) = \emptyset$，要么 $\mathrm{endpos}(w) \subseteq \mathrm{endpos}(u)$，取决于 u 是否为 w 的一个后缀：

$$\begin{cases} \mathrm{endpos}(w) \subseteq \mathrm{endpos}(u) & \text{if } u \text{ is a suffix of } w \\ \mathrm{endpos}(w) \cap \mathrm{endpos}(u) = \emptyset & \text{otherwise} \end{cases}$$

证明：如果集合 $\mathrm{endpos}(u)$ 与 $\mathrm{endpos}(w)$ 有至少一个公共元素，那么由于字符串 u 与 w 在相同位置结束，u 是 w 的一个后缀。所以在每次 w 出现的位置，子串 u 也会出现。所以 $\mathrm{endpos}(w) \subseteq \mathrm{endpos}(u)$。

引理 3： 考虑一个 endpos 等价类，将类中的所有子串按长度非递增的顺序排序。每个子串都不会比它前一个子串长，与此同时每个子串也是它前一个子串的后缀。换句话说，对于同一等价类的任一两子串，较短者为较长者的后缀，且该等价类中的子串长度恰好覆盖整个区间 $[x, y]$。

" 证明"

如果 endpos 等价类中只包含一个子串，引理显然成立。现在我们来讨论子串元素个数大于 1 的等价类。

由引理 1，两个不同的 endpos 等价的字符串中，较短者总是较长者的真后缀。因此，等价类中没有等长的字符串。

记 w 为等价类中最长的字符串、u 为等价类中最短的字符串。由引理 1，字符串 u 是字符串 w 的真后缀。现在考虑长度在区间 $[|u|, |w|]$ 中的 w 的任意后缀。容易看出，这个后缀也在同一等价类中，因为这个后缀只能在字符串 s 中以 w 的一个后缀的形式存在（也因为较短的后缀 u 在 s 中只以 w 的后缀的形式存在）。因此，由引理 1，这个后缀和字符串 w 的 endpos 相同。

后缀链接 `link`

考虑 SAM 中某个不是 t_0 的状态 v。我们已经知道，状态 v 对应于具有相同 endpos 的等价类。我们如果定义 w 为这些字符串中最长的一个，则所有其它的字符串都是 w 的后缀。

我们还知道字符串 w 的前几个后缀（按长度降序考虑）全部包含于这个等价类，且所有其它后缀（至少有一个——空后缀）在其它的等价类中。我们记 t 为最长的这样的后缀，然后将 v 的后缀链接连到 t 上。

换句话说，一个**后缀链接** $\mathrm{link}(v)$ 连接到对应于 w 的最长后缀的另一个 endpos 等价类的状态。

以下我们假设初始状态 t_0 对应于它自己这个等价类（只包含一个空字符串）。为了方便，我们规定 $\mathrm{endpos}(t_0) =$

$\{-1, 0, \dots, |S|-1\}$。

> **引理 4：** 所有后缀链接构成一棵根节点为 t_0 的树。

证明：考虑任意不是 t_0 的状态 v，后缀链接 $\text{link}(v)$ 连接到的状态对应于严格更短的字符串（后缀链接的定义、引理 3）。因此，沿后缀链接移动，我们总是能到达对应空串的初始状态 t_0。

> **引理 5：** 通过 endpos 集合构造的树（每个子节点的 *subset* 都包含在父节点的 *subset* 中）与通过后缀链接 link 构造的树相同。

证明：由引理 2，任意一个 SAM 的 endpos 集合形成了一棵树（因为两个集合要么完全没有交集要么其中一个是另一个的子集）。

我们现在考虑任意不是 t_0 的状态 v 及后缀链接 $\text{link}(v)$，由后缀链接和引理 2，我们可以得到

$$\text{endpos}(v) \subsetneq \text{endpos}(\text{link}(v)),$$

注意这里应该是 $\subsetneq$ 而不是 $\subseteq$，因为若 $\text{endpos}(v) = \text{endpos}(\text{link}(v))$，那么 v 和 $\text{link}(v)$ 应该被合并为一个节点

结合前面的引理有：后缀链接构成的树本质上是 endpos 集合构成的一棵树。

以下是对字符串 `abcbc` 构造 SAM 时产生的后缀链接树的一个**例子**，节点被标记为对应等价类中最长的子串。

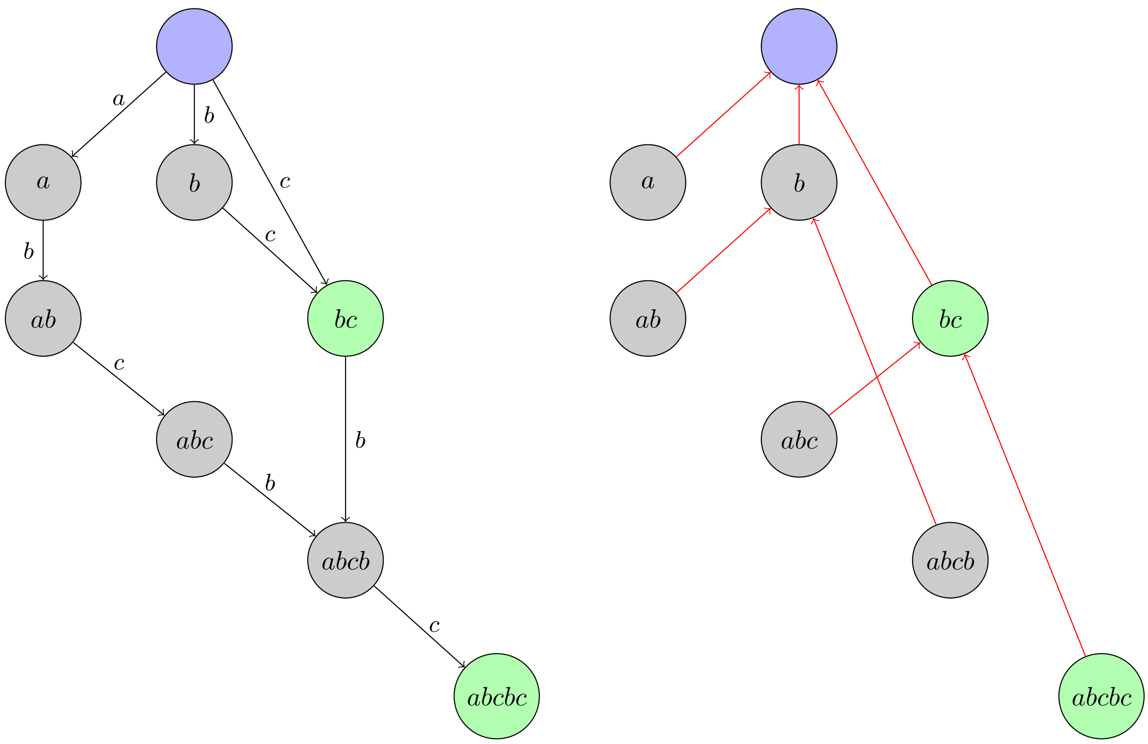


图 8.20

小结

在学习算法本身前，我们总结一下之前学过的知识，并引入一些辅助记号。

- s 的子串可以根据它们结束的位置 endpos 被划分为多个等价类；
- SAM 由初始状态 t_0 和与每一个 endpos 等价类对应的每个状态组成；
- 对于每一个状态 v，一个或多个子串与之匹配。我们记 $\text{longest}(v)$ 为其中最长的一个字符串，记 $\text{len}(v)$ 为它的长度。类似地，记 $\text{shortest}(v)$ 为最短的子串，它的长度为 $\text{minlen}(v)$。那么对应这个状态的所有字符串都是字符

串 $\mathrm{longest}(v)$ 的不同的后缀，且所有字符串的长度恰好覆盖区间 $[\mathrm{minlen}(v), \mathrm{len}(v)]$ 中的每一个整数。

- 对于任意不是 t_0 的状态 v，定义后缀链接为连接到对应字符串 $\mathrm{longest}(v)$ 的长度为 $\mathrm{minlen}(v) - 1$ 的后缀的一条边。从根节点 t_0 出发的后缀链接可以形成一棵树。这棵树也表示 endpos 集合间的包含关系。
- 对于 t_0 以外的状态 v，可用后缀链接 $\mathrm{link}(v)$ 表达 $\mathrm{minlen}(v)$：

$$\mathrm{minlen}(v) = \mathrm{len}(\mathrm{link}(v)) + 1.$$

- 如果我们从任意状态 v_0 开始顺着后缀链接遍历，总会到达初始状态 t_0。这种情况下我们可以得到一个互不相交的区间 $[\mathrm{minlen}(v_i), \mathrm{len}(v_i)]$ 的序列，且它们的并集形成了连续的区间 $[0, \mathrm{len}(v_0)]$。

算法

现在我们可以学习构造 SAM 的算法了。这个算法是**在线**算法，我们可以逐个加入字符串中的每个字符，并且在每一步中对应地维护 SAM。

为了保证线性的空间复杂度，我们将只保存 len 和 link 的值和每个状态的转移列表，我们不会标记终止状态（但是我们稍后会展示在构造 SAM 后如何分配这些标记）。

一开始 SAM 只包含一个状态 t_0，编号为 0（其它状态的编号为 $1, 2, ...$）。为了方便，对于状态 t_0 我们指定 $\mathrm{len} = 0$、$\mathrm{link} = -1$（-1 表示虚拟状态）。

过程

现在，任务转化为实现给当前字符串添加一个字符 c 的过程。算法流程如下：

- 令 $last$ 为添加字符 c 之前，整个字符串对应的状态（一开始我们设 $last = 0$，算法的最后一步更新 $last$）。
- 创建一个新的状态 cur，并将 $\mathrm{len}(cur)$ 赋值为 $\mathrm{len}(last) + 1$，在这时 $\mathrm{link}(cur)$ 的值还未知。
- 现在我们按以下流程进行（从状态 $last$ 开始）。如果还没有到字符 c 的转移，我们就添加一个到状态 cur 的转移，遍历后缀链接。如果在某个点已经存在到字符 c 的转移，我们就停下来，并将这个状态标记为 p。
- 如果没有找到这样的状态 p，我们就到达了虚拟状态 -1，我们将 $\mathrm{link}(cur)$ 赋值为 0 并退出。
- 假设现在我们找到了一个状态 p，其可以通过字符 c 转移。我们将转移到的状态标记为 q。
- 现在我们分类讨论两种状态，要么 $\mathrm{len}(p) + 1 = \mathrm{len}(q)$，要么不是。
- 如果 $\mathrm{len}(p) + 1 = \mathrm{len}(q)$，我们只要将 $\mathrm{link}(cur)$ 赋值为 q 并退出。
- 否则就会有些复杂。需要**复制**状态 q：我们创建一个新的状态 $clone$，复制 q 的除了 len 的值以外的所有信息（后缀链接和转移）。我们将 $\mathrm{len}(clone)$ 赋值为 $\mathrm{len}(p) + 1$。

  复制之后，我们将后缀链接从 cur 指向 $clone$，也从 q 指向 $clone$。

  最终我们需要使用后缀链接从状态 p 往回走，只要存在一条通过 p 到状态 q 的转移，就将该转移重定向到状态 $clone$。
- 以上三种情况，在完成这个过程之后，我们将 $last$ 的值更新为状态 cur。

如果我们还想知道哪些状态是**终止状态**而哪些不是，我们可以在为字符串 s 构造完完整的 SAM 后找到所有的终止状态。为此，我们从对应整个字符串的状态（存储在变量 $last$ 中），遍历它的后缀链接，直到到达初始状态。我们将所有遍历到的节点都标记为终止节点。容易理解这样做我们会准确地标记字符串 s 的所有后缀，这些状态都是终止状态。

在下一部分，我们将详细叙述算法每一步的细节，并证明它的**正确性**。因为我们只为 s 的每个字符创建一个或两个新状态，所以 SAM 只包含**线性个**状态。

而线性规模的转移个数，以及算法总体的线性运行时间还不那么清楚。

正确性证明

- 若一个转移 (p, q) 满足 $\mathrm{len}(p) + 1 = \mathrm{len}(q)$，则我们称这个转移是**连续的**。否则，即当 $\mathrm{len}(p) + 1 < \mathrm{len}(q)$ 时，这个转移被称为**不连续的**。从算法描述中可以看出，连续的、不连续的转移是算法的不同情况。连续的转移是固

定的，我们不会再改变了。与此相反，当向字符串中插入一个新的字符时，不连续的转移可能会改变（转移边的端点可能会改变）。

- 为了避免引起歧义，我们记向 SAM 中插入当前字符 c 之前的字符串为 s。
- 算法从创建一个新状态 cur 开始，对应于整个字符串 $s+c$。我们创建一个新的节点的原因很清楚。与此同时我们也创建了一个新的字符和一个新的等价类。
- 在创建一个新的状态之后，我们会从对应整个字符串 s 的状态通过后缀链接进行遍历。对于每一个状态，我们尝试添加一个通过字符 c 到新状态 cur 的转移。然而我们只能添加与原有转移不冲突的转移。因此我们只要找到已存在的 c 的转移，我们就必须停止。
- 最简单的情况是我们到达了虚拟状态 -1，这意味着我们为所有 s 的后缀添加了 c 的转移。这也意味着，字符 c 从未在字符串 s 中出现过。因此 cur 的后缀链接为状态 0。
- 第二种情况下，我们找到了现有的转移 (p,q)。这意味着我们尝试向自动机内添加一个**已经存在的**字符串 $x+c$（其中 x 为 s 的一个后缀，且字符串 $x+c$ 已经作为 s 的一个子串出现过了）。因为我们假设字符串 s 的自动机的构造是正确的，我们不应该在这里添加一个新的转移。然而，难点在于，从状态 cur 出发的后缀链接应该连接到哪个状态呢？我们要把后缀链接连到一个状态上，且其中最长的一个字符串恰好是 $x+c$，即这个状态的 len 应该是 $\operatorname{len}(p)+1$。然而还不存在这样的状态，即 $\operatorname{len}(q)>\operatorname{len}(p)+1$。这种情况下，我们必须通过拆开状态 q 来创建一个这样的状态。
- 如果转移 $(p,\,q)$ 是连续的，那么 $\operatorname{len}(q)=\operatorname{len}(p)+1$。在这种情况下一切都很简单。我们只需要将 cur 的后缀链接指向状态 q。
- 否则转移是不连续的，即 $\operatorname{len}(q)>\operatorname{len}(p)+1$，这意味着状态 q 不只对应于长度为 $\operatorname{len}(p)+1$ 的后缀 $s+c$，还对应于 s 的更长的子串。除了将状态 q 拆成两个子状态以外我们别无他法，所以第一个子状态的长度就是 $\operatorname{len}(p)+1$ 了。

  我们如何拆开一个状态呢？我们**复制**状态 q，产生一个状态 $clone$，我们将 $\operatorname{len}(clone)$ 赋值为 $\operatorname{len}(p)+1$。由于我们不想改变遍历到 q 的路径，我们将 q 的所有转移复制到 $clone$。我们也将从 $clone$ 出发的后缀链接设置为 q 的后缀链接的目标，并设置 q 的后缀链接为 $clone$。

  在拆开状态后，我们将从 cur 出发的后缀链接设置为 $clone$。

  最后一步我们将一些到 q 转移重定向到 $clone$。我们需要修改哪些转移呢？只重定向相当于所有字符串 $w+c$（其中 w 是 p 的最长字符串）的后缀就够了。即，我们需要继续沿着后缀链接遍历，从结点 p 直到虚拟状态 -1 或者是转移到不是状态 q 的一个转移。

对操作次数为线性的证明

首先我们假设字符集大小为**常数**。如果字符集大小不是常数，SAM 的时间复杂度就不是线性的。从一个结点出发的转移存储在支持快速查询和插入的平衡树中。因此如果我们记 Σ 为字符集，$|\Sigma|$ 为字符集大小，则算法的渐进时间复杂度为 $O(n\log|\Sigma|)$，空间复杂度为 $O(n)$。然而如果字符集足够小，可以不写平衡树，以空间换时间将每个结点的转移存储为长度为 $|\Sigma|$ 的数组（用于快速查询）和链表（用于快速遍历所有可用关键字）。这样算法的时间复杂度为 $O(n)$，空间复杂度为 $O(n\,|\Sigma|)$。

所以我们将认为字符集的大小为常数，即每次对一个字符搜索转移、添加转移、查找下一个转移。这些操作的时间复杂度都为 $O(1)$。

如果我们考虑算法的各个部分，算法中有三处时间复杂度不明显是线性的：

- 第一处是遍历所有状态 $last$ 的后缀链接，添加字符 c 的转移。
- 第二处是当状态 q 被复制到一个新的状态 $clone$ 时复制转移的过程。
- 第三处是修改指向 q 的转移，将它们重定向到 $clone$ 的过程。

我们使用 SAM 的大小（状态数和转移数）为**线性的**的事实（对状态数是线性的的证明就是算法本身，对转移数为线性的的证明将在稍后实现算法后给出）。

因此上述**第一处和第二处**的总复杂度显然为线性的，因为单次操作均摊只为自动机添加了一个新转移。

还需为**第三处**估计总复杂度，我们将最初指向 q 的转移重定向到 $clone$。我们记 $v=\operatorname{longest}(p)$，这是一个字符串

s 的后缀，每次迭代长度都递减——因为字符串 s 的位置每次迭代都单调上升。这种情况下，如果在循环的第一次迭代之前，相对应的字符串 v 在距离 $last$ 的深度为 k $(k \geq 2)$ 的位置上（深度记为后缀链接的数量），那么在最后一次迭代后，字符串 $v + c$ 将会成为路径上第二个从 cur 出发的后缀链接（它将会成为新的 $last$ 的值）。

因此，循环中的每次迭代都会使作为当前字符串的后缀的字符串 $\operatorname{longest}(\operatorname{link}(\operatorname{link}(last))$ 的位置单调递增。因此这个循环最多不会执行超过 n 次迭代，这正是我们需要证明的。

实现

首先，我们实现一种存储一个转移的全部信息的数据结构。如果需要的话，你可以在这里加入一个终止标记，也可以是一些其它信息。我们将用一个 `map` 存储转移的列表，允许我们在总计 $O(n)$ 的空间复杂度和 $O(n \log |\Sigma|)$ 的时间复杂度内处理整个字符串。（注：在字符集大小为较小的常数，比如 26 时，将 `next` 定义为 `int[26]` 更方便）

```
struct state {
  int len, link;
  std::map<char, int> next;
};
```

SAM 本身将会存储在一个 `state` 结构体数组中。我们记录当前自动机的大小 `sz` 和变量 `last`，当前整个字符串对应的状态。

```
const int MAXLEN = 100000;
state st[MAXLEN * 2];
int sz, last;
```

我们定义一个函数来初始化 SAM（创建一个只有初始状态的 SAM）。

```
void sam_init() {
  st[0].len = 0;
  st[0].link = -1;
  sz++;
  last = 0;
}
```

最终我们给出主函数的实现：给当前行末增加一个字符，对应地在之前的基础上建造自动机。

" 实现"

```
void sam_extend(char c) {
  int cur = sz++;
  st[cur].len = st[last].len + 1;
  int p = last;
  while (p != -1 && !st[p].next.count(c)) {
    st[p].next[c] = cur;
    p = st[p].link;
  }
  if (p == -1) {
    st[cur].link = 0;
  } else {
    int q = st[p].next[c];
    if (st[p].len + 1 == st[q].len) {
      st[cur].link = q;
    } else {
      int clone = sz++;
      st[clone].len = st[p].len + 1;
      st[clone].next = st[q].next;
```

```
      st[clone].link = st[q].link;
      while (p != -1 && st[p].next[c] == q) {
        st[p].next[c] = clone;
        p = st[p].link;
      }
      st[q].link = st[cur].link = clone;
    }
  }
  last = cur;
}
```

正如之前提到的一样，如果你用内存换时间（空间复杂度为 $O(n|\Sigma|)$，其中 $|\Sigma|$ 为字符集大小），你可以在 $O(n)$ 的时间内构造字符集大小任意的 SAM。但是这样你需要为每一个状态储存一个大小为 $|\Sigma|$ 的数组（用于快速跳转到转移的字符），和另外一个所有转移的链表（用于快速在转移中迭代）。

更多性质

状态数

对于一个长度为 n 的字符串 s，它的 SAM 中的状态数**不会超过** $2n-1$（假设 $n \geq 2$）。

算法本身即可证明该结论。一开始，自动机含有一个状态，第一次和第二次迭代中只会创建一个节点，剩余的 $n-2$ 步中每步会创建至多 2 个状态。

然而我们也能在**不借助这个算法**的情况下**证明**这个估计值。我们回忆一下状态数等于不同的 endpos 集合个数。这些 endpos 集合形成了一棵树（祖先节点的集合包含了它所有孩子节点的集合）。考虑将这棵树稍微变形一下：只要它有一个只有一个孩子的内部结点（这意味着该子节点的集合至少遗漏了它的父集合中的一个位置），我们创建一个含有这个遗漏位置的集合。最后我们可以获得一棵每一个内部结点的度数大于 1 的树，且叶子节点的个数不超过 n。因此这样的树里有不超过 $2n-1$ 个节点。

字符串 `abbb…bbb` 的状态数达到了该上界：从第三次迭代后的每次迭代，算法都会拆开一个状态，最终产生恰好 $2n-1$ 个状态。

转移数

对于一个长度为 n 的字符串 s，它的 SAM 中的转移数**不会超过** $3n-4$（假设 $n \geq 3$）。

证明如下：

我们首先估计连续的转移的数量。考虑自动机中从状态 t_0 开始的所有最长路径的生成树。生成树只包含连续的边，因此数量少于状态数，即边数不会超过 $2n-2$。

现在我们来估计不连续的转移的数量。令当前不连续转移为 (p,q)，其字符为 c。我们取它的对应字符串 $u+c+w$，其中字符串 u 对应于初始状态到 p 的最长路径，w 对应于从 q 到任意终止状态的最长路径。一方面，每个不完整的字符串所对应的形如 $u+c+w$ 的字符串是不同的（因为字符串 u 和 w 仅由完整的转移组成）。另一方面，由终止状态的定义，每个形如 $u+c+w$ 的字符串都是整个字符串 s 的后缀。因为 s 只有 n 个非空后缀，且形如 $u+c+w$ 的字符串都不包含 s（因为整个字符串只包含完整的转移），所以非完整的转移的总数不会超过 $n-1$。

将以上两个估计值相加，我们可以得到上界 $3n-3$。然而，最大的状态数只能在类似于 `abbb…bbb` 的情况中产生，而此时转移数量显然少于 $3n-3$。

因此我们可以获得更为紧确的 SAM 的转移数的上界：$3n-4$。字符串 `abbb…bbbc` 就达到了这个上界。

额外信息

观察实现中的结构体的每个变量。实际上，尽管 SAM 本身由 `next` 组成，但 SAM 构造算法中作为辅助变量的 `link` 和 `len` 在应用中常常比 `next` 重要，甚至可以抛开 `next` 单独使用。

设字符串的长度为 n，考虑 `extend` 操作中 `cur` 变量的值，这个节点对应的状态是执行 `extend` 操作时的当前

字符串，即字符串的一个前缀，每个前缀有一个终点。这样得到的 n 个节点，对应了 n 个不同的**终点**。设第 i 个节点为 v_i，对应的是 $S_{1\dots i}$，终点是 i。姑且把这些节点称之为「终点节点」。

考虑给 SAM 赋予树形结构，树的根为 0，且其余节点 v 的父亲为 $\text{link}(v)$。则这棵树与原 SAM 的关系是：

- 每个节点的终点集合等于其**子树**内所有终点节点对应的终点的集合。

在此基础上可以给每个节点赋予一个最长字符串，是其终点集合中**任意**一个终点开始**往前**取 `len` 个字符得到的字符串。每个这样的字符串都一样，且 `len` 恰好是满足这个条件的最大值。

这些字符串满足的性质是：

- 如果节点 A 是 B 的祖先，则节点 A 对应的字符串是节点 B 对应的字符串的**后缀**。

这条性质把字符串所有前缀组成了一棵树，且有许多符合直觉的树的性质。例如，$S_{1\dots p}$ 和 $S_{1\dots q}$ 的最长公共后缀对应的字符串就是 v_p 和 v_q 对应的 LCA 的字符串。实际上，这棵树与将字符串 S 翻转后得到字符串的压缩后缀树结构相同。

每个状态 i 对应的子串数量是 $\text{len}(i)-\text{len}(\text{link}(i))$（节点 0 例外）。注意到 $\text{link}(i)$ 对应的字符串是 i 对应的字符串的一个后缀，这些子串就是 i 对应字符串的所有后缀，去掉被父亲「抢掉」的那部分，即 $\text{link}(i)$ 对应字符串的所有后缀。

应用

下面我们来看一些可以用 SAM 解决的问题。简单起见，假设字符集的大小 k 为常数。这允许我们认为增加一个字符和遍历的复杂度为常数。

检查字符串是否出现

> 给一个文本串 T 和多个模式串 P，我们要检查字符串 P 是否作为 T 的一个子串出现。

我们在 $O(|T|)$ 的时间内对文本串 T 构造后缀自动机。为了检查模式串 P 是否在 T 中出现，我们沿转移（边）从 t_0 开始根据 P 的字符进行转移。如果在某个点无法转移下去，则模式串 P 不是 T 的一个子串。如果我们能够这样处理完整个字符串 P，那么模式串在 T 中出现过。

对于每个字符串 P，算法的时间复杂度为 $O(|P|)$。此外，这个算法还找到了模式串 P 在文本串中出现的最大前缀长度。

不同子串个数

> 给一个字符串 S，计算不同子串的个数。

对字符串 S 构造后缀自动机。

每个 S 的子串都相当于自动机中的一些路径。因此不同子串的个数等于自动机中以 t_0 为起点的不同路径的条数。

考虑到 SAM 为有向无环图，不同路径的条数可以通过动态规划计算。即令 d_v 为从状态 v 开始的路径数量（包括长度为零的路径），则我们有如下递推方程：

$$d_v = 1 + \sum_{w:(v,w,c)\in DAWG} d_w$$

即，d_v 可以表示为所有 v 的转移的末端的和。

所以不同子串的个数为 $d_{t_0}-1$（因为要去掉空子串）。

总时间复杂度为：$O(|S|)$。

另一种方法是利用上述后缀自动机的树形结构。每个节点对应的子串数量是 $\text{len}(i)-\text{len}(\text{link}(i))$，对自动机所有节点求和即可。

例题：【模板】后缀自动机[1]，SDOI2016 生成魔咒[2]

所有不同子串的总长度

> 给定一个字符串 S，计算所有不同子串的总长度。

本题做法与上一题类似，只是现在我们需要考虑分两部分进行动态规划：不同子串的数量 d_v 和它们的总长度 ans_v。

我们已经在上一题中介绍了如何计算 d_v。ans_v 的值可以通过以下递推式计算：

$$ans_v = \sum_{w:(v,w,c)\in DAWG} d_w + ans_w$$

我们取每个邻接结点 w 的答案，并加上 d_w（因为从状态 v 出发的子串都增加了一个字符）。

算法的时间复杂度仍然是 $O(|S|)$。

同样可以利用上述后缀自动机的树形结构。每个节点对应的所有后缀长度是 $\frac{\operatorname{len}(i)\times(\operatorname{len}(i)+1)}{2}$，减去其 link 节点的对应值就是该节点的净贡献，对自动机所有节点求和即可。

字典序第 k 大子串

> 给定一个字符串 S。多组询问，每组询问给定一个数 K_i，查询 S 的所有子串中字典序第 K_i 大的子串。

解决这个问题的思路可以从解决前两个问题的思路发展而来。字典序第 k 大的子串对应于 SAM 中字典序第 k 大的路径，因此在计算每个状态的路径数后，我们可以很容易地从 SAM 的根开始找到第 k 大的路径。

预处理的时间复杂度为 $O(|S|)$，单次查询的复杂度为 $O(|ans|\cdot|\Sigma|)$（其中 ans 是查询的答案，$|\Sigma|$ 为字符集的大小）。

> 虽然该题是后缀自动机的经典题，但实际上这题由于涉及字典序，用后缀数组做最方便。

例题：SPOJ - SUBLEX[3]，TJOI2015 弦论[4]

最小循环移位

> 给定一个字符串 S。找出字典序最小的循环移位。

容易发现字符串 $S+S$ 包含字符串 S 的所有循环移位作为子串。

所以问题简化为在 $S+S$ 对应的后缀自动机上寻找最小的长度为 $|S|$ 的路径，这可以通过平凡的方法做到：我们从初始状态开始，贪心地访问最小的字符即可。

总的时间复杂度为 $O(|S|)$。

出现次数

> 对于一个给定的文本串 T，有多组询问，每组询问给一个模式串 P，回答模式串 P 在字符串 T 中作为子串出现了多少次。

利用后缀自动机的树形结构，进行 dfs 即可预处理每个节点的终点集合大小。在自动机上查找模式串 P 对应的节点，如果存在，则答案就是该节点的终点集合大小；如果不存在，则答案为 0.

以下为原方法：

> 对文本串 T 构造后缀自动机。
>
> 接下来做预处理：对于自动机中的每个状态 v，预处理 cnt_v，使之等于 $\operatorname{endpos}(v)$ 集合的大小。事实上，对应同一状态 v 的所有子串在文本串 T 中的出现次数相同，这相当于集合 endpos 中的位置数。
>
> 然而我们不能明确的构造集合 endpos，因此我们只考虑它们的大小 cnt。
>
> 为了计算这些值，我们进行以下操作。对于每个状态，如果它不是通过复制创建的（且它不是初始状态 t_0），我们将它的 cnt 初始化为 1。然后我们按它们的长度 len 降序遍历所有状态，并将当前的 cnt_v 的值加到后缀链接指向的状态上，即：
>
> $$cnt_{\operatorname{link}(v)} += cnt_v$$
>
> 这样做每个状态的答案都是正确的。
>
> 为什么这是正确的？不是通过复制获得的状态，恰好有 $|T|$ 个，并且它们中的前 i 个在我们插入前 i 个字符时产生。因此对于每个这样的状态，我们在它被处理时计算它们所对应的位置的数量。因此我们初始将这些状态的 cnt 的值赋为 1，其它状态的 cnt 值赋为 0。
>
> 接下来我们对每一个 v 执行以下操作：$cnt_{\operatorname{link}(v)} += cnt_v$。其背后的含义是，如果有一个字符串 v 出现了 cnt_v 次，那么它的所有后缀也在完全相同的地方结束，即也出现了 cnt_v 次。
>
> 为什么我们在这个过程中不会重复计数（即把某些位置数了两次）呢？因为我们只将一个状态的位置添加到**一个**其它的状态上，所以一个状态不可能以两种不同的方式将其位置重复地指向另一个状态。
>
> 因此，我们可以在 $O(|T|)$ 的时间内计算出所有状态的 cnt 的值。
>
> 最后回答询问只需要查找值 cnt_t，其中 t 为模式串对应的状态，如果该模式串不存在答案就为 0。单次查询的时间复杂度为 $O(|P|)$。

第一次出现的位置

> 给定一个文本串 T，多组查询。每次查询字符串 P 在字符串 T 中第一次出现的位置（P 的开头位置）。

我们构造一个后缀自动机。我们对 SAM 中的所有状态预处理位置 firstpos。即，对每个状态 v 我们想要找到第一次出现这个状态的末端的位置 $\operatorname{firstpos}[v]$。换句话说，我们希望先找到每个集合 endpos 中的最小的元素（显然我们不能显式地维护所有 endpos 集合）。

为了维护 firstpos 这些位置，我们将原函数扩展为 `sam_extend()`。当我们创建新状态 cur 时，我们令：

$$\operatorname{firstpos}(cur) = \operatorname{len}(cur) - 1$$

；当我们将结点 q 复制到 $clone$ 时，我们令：

$$\operatorname{firstpos}(clone) = \operatorname{firstpos}(q)$$

（因为值的唯一的其它选项 $\operatorname{firstpos}(cur)$ 显然太大了）。

那么查询的答案就是 $\operatorname{firstpos}(t) - |P| + 1$，其中 t 为对应字符串 P 的状态。单次查询只需要 $O(|P|)$ 的时间。

所有出现的位置

> 问题同上，这一次需要查询文本串 T 中模式串出现的所有位置。

利用后缀自动机的树形结构，遍历子树，一旦发现终点节点就输出。

以下为原解法：

我们还是对文本串 T 构造后缀自动机。与上一个问题相似，我们为所有状态计算位置 firstpos。

如果 t 为对应于模式串 T 的状态，显然 $\operatorname{firstpos}(t)$ 为答案的一部分。需要查找的其它位置怎么办？我们使用了含有字符串 P 的自动机，我们还需要将哪些状态纳入自动机呢？所有对应于以 P 为后缀的字符串的状态。换句话说我们要找到所有可以通过后缀链接到达状态 t 的状态。

因此为了解决这个问题，我们需要为每一个状态保存一个指向它的后缀引用列表。查询的答案就包含了对于每个我们能从状态 t 只使用后缀引用进行 DFS 或 BFS 的所有状态的 firstpos 值。

这种变通方案的时间复杂度为 $O(answer(P))$，因为我们不会重复访问一个状态（因为对于仅有一个后缀链接指向一个状态，所以不存在两条不同的路径指向同一状态）。

我们只需要考虑两个可能有相同 endpos 值的不同状态。如果一个状态是由另一个复制而来的，则这种情况会发生。然而，这并不会对复杂度分析造成影响，因为每个状态至多被复制一次。

此外，如果我们不从被复制的节点输出位置，我们也可以去除重复的位置。事实上对于一个状态，如果经过被复制状态可以到达，则经过原状态也可以到达。因此，如果我们给每个状态记录标记 `is_clone` 来代表这个状态是不是被复制出来的，我们就可以简单地忽略掉被复制的状态，只输出其它所有状态的 $firstpos$ 的值。

以下是大致的实现：

```cpp
struct state {
  bool is_clone;
  int first_pos;
  std::vector<int> inv_link;
  // some other variables
};

// 在构造 SAM 后
for (int v = 1; v < sz; v++) st[st[v].link].inv_link.push_back(v);

// 输出所有出现位置
void output_all_occurrences(int v, int P_length) {
  if (!st[v].is_clone) cout << st[v].first_pos - P_length + 1 << endl;
  for (int u : st[v].inv_link) output_all_occurrences(u, P_length);
}
```

最短的没有出现的字符串

> 给定一个字符串 S 和一个特定的字符集，我们要找一个长度最短的没有在 S 中出现过的字符串。

我们在字符串 S 的后缀自动机上做动态规划。

令 d_v 为节点 v 的答案，即，我们已经处理完了子串的一部分，当前在状态 v，想找到不连续的转移需要添加的最小字符数量。计算 d_v 非常简单。如果不存在使用字符集中至少一个字符的转移，则 $d_v = 1$。否则添加一个字符是不够的，我们需要求出所有转移中的最小值：

$$d_v = 1 + \min_{w:(v,w,c)\in SAM} d_w$$

问题的答案就是 d_{t_0}，字符串可以通过计算过的数组 d 逆推回去。

两个字符串的最长公共子串

> 给定两个字符串 S 和 T，求出最长公共子串，公共子串定义为在 S 和 T 中都作为子串出现过的字符串 X。

我们对字符串 S 构造后缀自动机。

我们现在处理字符串 T，对于每一个前缀，都在 S 中寻找这个前缀的最长后缀。换句话说，对于每个字符串 T 中的位置，我们想要找到这个位置结束的 S 和 T 的最长公共子串的长度。显然问题的答案就是所有 l 的最大值。

为了达到这一目的，我们使用两个变量，**当前状态** v 和**当前长度** l。这两个变量描述当前匹配的部分：它的长度和它们对应的状态。

一开始 $v = t_0$ 且 $l = 0$，即，匹配为空串。

现在我们来描述如何添加一个字符 T_i 并为其重新计算答案：

- 如果存在一个从 v 到字符 T_i 的转移，我们只需要转移并让 l 自增一。
- 如果不存在这样的转移，我们需要缩短当前匹配的部分，这意味着我们需要按照后缀链接进行转移：

$$v = \operatorname{link}(v)$$

与此同时，需要缩短当前长度。显然我们需要将 l 赋值为 $\operatorname{len}(v)$，因为经过这个后缀链接后我们到达的状态所对应的最长字符串是一个子串。

- 如果仍然没有使用这一字符的转移，我们继续重复经过后缀链接并减小 l，直到我们找到一个转移或到达虚拟状态 -1（这意味着字符 T_i 根本没有在 S 中出现过，所以我们设置 $v = l = 0$）。

这一部分的时间复杂度为 $O(|T|)$，因为每次移动我们要么可以使 l 增加一，要么可以在后缀链接间移动几次，每次都减小 l 的值。

代码实现：

```
string lcs(const string &S, const string &T) {
  sam_init();
  for (int i = 0; i < S.size(); i++) sam_extend(S[i]);

  int v = 0, l = 0, best = 0, bestpos = 0;
  for (int i = 0; i < T.size(); i++) {
    while (v && !st[v].next.count(T[i])) {
      v = st[v].link;
      l = st[v].length;
    }
    if (st[v].next.count(T[i])) {
      v = st[v].next[T[i]];
      l++;
    }
    if (l > best) {
      best = l;
      bestpos = i;
    }
  }
  return T.substr(bestpos - best + 1, best);
}
```

例题：SPOJ Longest Common Substring[5]

多个字符串间的最长公共子串

> 给定 k 个字符串 S_i。我们需要找到它们的最长公共子串，即作为子串出现在每个字符串中的字符串 X。

我们将所有的子串连接成一个较长的字符串 T，以特殊字符 D_i 分开每个字符串（一个字符对应一个字符串）：

$$T = S_1 + D_1 + S_2 + D_2 + \cdots + S_k + D_k.$$

然后对字符串 T 构造后缀自动机。

现在我们需要在自动机中找到存在于所有字符串 S_i 中的一个字符串，这可以通过使用添加的特殊字符完成。注意如果 S_j 包含了一个子串，则 SAM 中存在一条从包含字符 D_j 的子串而不包含以其它字符 $D_1, \dots, D_{j-1}, D_{j+1}, \dots, D_k$ 开始的路径。

因此我们需要计算可达性，即对于自动机中的每个状态和每个字符 D_i，是否存在这样的一条路径。这可以容易地通过 DFS 或 BFS 及动态规划计算。之后，问题的答案就是状态 v 的字符串 $\text{longest}(v)$ 中存在所有特殊字符的路径。

例题：SPOJ Longest Common Substring II[6]

例题

- HihoCoder #1441 : 后缀自动机一 · 基本概念[7]
- 【模板】后缀自动机[1]
- SDOI2016 生成魔咒[2]
- SPOJ - SUBLEX[3]
- TJOI2015 弦论[4]
- SPOJ Longest Common Substring[5]
- SPOJ Longest Common Substring II[6]
- Codeforces 1037H Security[8]
- Codeforces 666E Forensic Examination[9]
- HDU4416 Good Article Good sentence[10]
- HDU4436 str2int[11]
- HDU6583 Typewriter[12]
- Codeforces 235C Cyclical Quest[13]
- CTSC2012 熟悉的文章[14]
- NOI2018 你的名字[15]

相关资料

我们先给出与 SAM 有关的最初的一些文献：

- A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, R. McConnell.**Linear Size Finite Automata for the Set of All Subwords of a Word. An Outline of Results**[1983]
- A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler.**The Smallest Automaton Recognizing the Subwords of a Text**[1984]
- Maxime Crochemore.**Optimal Factor Transducers**[1985]
- Maxime Crochemore.**Transducers and Repetitions**[1986]
- A. Nerode.**Linear automaton transformations**[1958]

另外，在更新的一些资源以及很多关于字符串算法的书中，都能找到这个主题：

- Maxime Crochemore, Rytter Wowjcieh.**Jewels of Stringology**[2002]
- Bill Smyth.**Computing Patterns in Strings**[2003]
- Bill Smith.**Methods and algorithms of calculations on lines**[2006]

另外，还有一些资料：

- 《后缀自动机》，陈立杰。
- 《后缀自动机在字典树上的拓展》，刘研绎。
- 《后缀自动机及其应用》，张天扬。
- https://www.cnblogs.com/zinthos/p/3899679.html[16]
- https://codeforces.com/blog/entry/20861[17]
- https://zhuanlan.zhihu.com/p/25948077[18]

参考资料与注释

[1] 【模板】后缀自动机 [1-1] [1-2]

[2] SDOI2016 生成魔咒 [2-1] [2-2]

[3] SPOJ - SUBLEX [3-1] [3-2]

[4] TJOI2015 弦论 [4-1] [4-2]

[5] SPOJ Longest Common Substring [5-1] [5-2]

[6] SPOJ Longest Common Substring II [6-1] [6-2]

[7] HihoCoder #1441 : 后缀自动机一 · 基本概念

[8] Codeforces 1037H Security

[9] Codeforces 666E Forensic Examination

[10] HDU4416 Good Article Good sentence

[11] HDU4436 str2int

[12] HDU6583 Typewriter

[13] Codeforces 235C Cyclical Quest

[14] CTSC2012 熟悉的文章

[15] NOI2018 你的名字

[16] https://www.cnblogs.com/zinthos/p/3899679.html

[17] https://codeforces.com/blog/entry/20861

[18] https://zhuanlan.zhihu.com/p/25948077

[19]

[20] Suffix Automaton

8.14 后缀平衡树

定义

后缀之间的大小由字典序定义，后缀平衡树就是一个维护这些后缀顺序的平衡树，即字符串 T 的后缀平衡树是 T 所有后缀的有序集合。后缀平衡树上的一个节点相当于原字符串的一个后缀。

特别地，后缀平衡树的中序遍历即为后缀数组。

构造过程

对长度为 n 的字符串 T 建立其后缀平衡树，考虑逆序将其后缀加入后缀平衡树。

记后缀平衡树维护的集合为 X，当前添加的后缀为 S，则添加下一个后缀就是向 X 中加入 $\mathtt{c}S$（亦可理解为后缀平衡树维护的字符串为 S，下一步往 S 前加入一个字符 $\mathtt{c}$）。这一操作其实就是向平衡树中插入节点。

这里使用期望树高为 $O(\log n)$ 的平衡树，例如替罪羊树或 Treap 等。

做法 1

插入时，暴力比较两个后缀之间的大小关系，从而判断之后是往哪一个子树添加。这样子，单次插入至多比较 $O(\log n)$ 次，单次比较的时间复杂度至多为 $O(n)$，一共 $O(n\log n)$。

一共会插入 n 次，所以该做法的时间复杂度存在上界 $O(n^2\log n)$。

做法 2

注意到 $\mathtt{c}S$ 与 S 的区别仅在于 $\mathtt{c}$，且 S 已经属于 X 了，可以利用这一点来优化插入操作。

假设当前要比较 $\mathtt{c}S$ 与 A 两个字符串的大小，且 $A, S \in X$。每次比较时，首先比较两串的首字符。若首字符不等，则两串的大小关系就已经确定了；若首字符相等，那么就只需要判断去除首字符后两字符串的大小关系。而两串去除首字符后都已经属于 X 了，这时候可以借助平衡树 $O(\log n)$ 求排名的操作来完成后续的比较。这样，单次插入的操作至多 $O(\log^2 n)$。

一共会插入 n 次，所以该做法的时间复杂度存在上界 $O(n\log^2 n)$。

做法 3

根据做法 2，如果能够 $O(1)$ 判断平衡树中两个节点之间的大小关系，那么就可以在 $O(n\log n)$ 的时间内完成后缀平衡树的构造。

记 val_i 表示节点 i 的值。如果在建平衡树时，每个节点多维护一个标记 tag_i，使得若 $tag_i > tag_j \iff val_i > val_j$，那么就可以根据 tag_i 的大小 $O(1)$ 判断平衡树中两个节点的大小。

不妨令平衡树中每个节点对应一个实数区间，令根节点对应 $(0,1)$。对于节点 i，记其对应的实数区间为 (l,r)，则 $tag_i = \frac{l+r}{2}$，其左子树对应实数区间 (l, tag_i)，其右子树对应实数区间 (tag_i, r)。易证 tag_i 满足上述要求。

由于使用了期望树高为 $O(\log n)$ 的平衡树，所以精度是有一定保证的。实际实现时也可以用一个较大的区间来做，例如让根对应 $(0, 10^{18})$。

做法 4

其实可以先构建出后缀数组，然后再根据后缀数组构建后缀平衡树。这样做的复杂度瓶颈在于后缀数组的构建复杂度或者所用平衡树一次性插入 n 个元素的复杂度。

删除操作

假设当前添加的后缀为 $\mathtt{c}S$，上一个添加的后缀为 S。后缀平衡树还支持删除后缀 $\mathtt{c}S$ 的操作（亦可理解为后缀平衡树维护的字符串为 $\mathtt{c}S$，将开头的 c 删除）。

类似于插入操作，借助平衡树的删除节点操作可以完成删除 $\mathtt{c}S$ 的操作。

后缀平衡树的优点

- 后缀平衡树的思路比较清晰，相比后缀自动机等后缀结构更好理解，会写平衡树就能写。
- 后缀平衡树的复杂度不依赖于字符集的大小
- 后缀平衡树支持在字符串开头删除一个字符
- 如果使用支持可持久化的平衡树，那么后缀平衡树也能可持久化

例题

P3809【模板】后缀排序[1]

后缀数组的模板题，建出后缀平衡树之后，通过中序遍历得到后缀数组。

"SGT 版本的参考代码"

```cpp
#include <bits/stdc++.h>
using namespace std;

const int N = 1e6 + 5;
const double INF = 1e18;

int n, m, sa[N];
char t[N];

// SuffixBST(SGT Ver)

// 顺序加入，查询时将询问串翻转
// 以 i 开始的后缀，对应节点的编号为 i
const double alpha = 0.75;
int root;
int sz[N], L[N], R[N];
double tag[N];
int buffer_size, buffer[N];

bool cmp(int x, int y) {
  if (t[x] != t[y]) return t[x] < t[y];
  return tag[x + 1] < tag[y + 1];
```

```cpp
}

void init() { root = 0; }

void new_node(int& rt, int p, double lv, double rv) {
  rt = p;
  sz[rt] = 1;
  tag[rt] = (lv + rv) / 2;
  L[rt] = R[rt] = 0;
}

void push_up(int x) {
  if (!x) return;
  sz[x] = sz[L[x]] + 1 + sz[R[x]];
}

bool balance(int rt) { return alpha * sz[rt] > max(sz[L[rt]], sz[R[rt]]); }

void flatten(int rt) {
  if (!rt) return;
  flatten(L[rt]);
  buffer[++buffer_size] = rt;
  flatten(R[rt]);
}

void build(int& rt, int l, int r, double lv, double rv) {
  if (l > r) {
    rt = 0;
    return;
  }
  int mid = (l + r) >> 1;
  double mv = (lv + rv) / 2;

  rt = buffer[mid];
  tag[rt] = mv;
  build(L[rt], l, mid - 1, lv, mv);
  build(R[rt], mid + 1, r, mv, rv);
  push_up(rt);
}

void rebuild(int& rt, double lv, double rv) {
  buffer_size = 0;
  flatten(rt);
  build(rt, 1, buffer_size, lv, rv);
}

void insert(int& rt, int p, double lv, double rv) {
  if (!rt) {
    new_node(rt, p, lv, rv);
    return;
  }

  if (cmp(p, rt))
    insert(L[rt], p, lv, tag[rt]);
```

```
  else
    insert(R[rt], p, tag[rt], rv);

  push_up(rt);
  if (!balance(rt)) rebuild(rt, lv, rv);
}

void inorder(int rt) {
  if (!rt) return;
  inorder(L[rt]);
  sa[++m] = rt;
  inorder(R[rt]);
}

void solve(int Case) {
  scanf("%s", t + 1);
  n = strlen(t + 1);

  init();
  for (int i = n; i >= 1; --i) {
    insert(root, i, 0, INF);
  }

  // 后缀平衡树的中序遍历即为后缀数组
  m = 0;
  inorder(root);

  for (int i = 1; i <= n; ++i) printf("%d ", sa[i]);
  printf("\n");
}

int main() {
  solve(1);
  return 0;
}
```

P6164【模板】后缀平衡树[2]

" 题意"

给定初始字符串 s 和 q 个操作：

1. 在当前字符串的后面插入若干个字符。
2. 在当前字符串的后面删除若干个字符。
3. 询问字符串 t 作为连续子串在当前字符串中出现了几次？

题目**强制在线**，字符串变化长度以及初始长度 $\leq 8\times 10^5$，$q\leq 10^5$，询问的总长度 $\leq 3\times 10^6$。

对于操作 1 和操作 2，由于后缀平衡树维护头插和头删操作比较方便，所以想到把尾插和尾删操作搞成头插和头删。这里如果维护 s 的反串的后缀平衡树，而非 s 的后缀平衡树，就可以完成上述转换。平衡树的添加和删除都是 $O(\log n)$ 的，所以添加或者删除一个字符的时间复杂度为 $O(\log n)$。记添加和删除的总字符数为 N，那么这一部分总的时间复杂度为 $O(N\log n)$。

对于操作 3，t 的出现次数等于以 t 为前缀的后缀数量，而以 t 为前缀的后缀数量等于其后继的排名减去其前驱

的排名。在 t 后面加入一个极大的字符，就可以构造出 t 的一个后继。将 t 的最后一个字符减小 1，就可以构造出 t 的一个前驱。

现在要查询某一个串 t 在后缀平衡树中排名，由于不能保证 t 在后缀平衡树中出现过，所以每次只能暴力比较字符串大小。单次比较的时间复杂度为 $O(|t|)$，每次查询至多比较 $O(\log n)$ 次，所以单次查询的复杂度为 $O(|t|\log n)$。记所有询问串的长度和为 L，那么这一部分总的时间复杂度为 $O(L\log n)$。

"SGT 版本的参考代码"

```cpp
#include <bits/stdc++.h>
using namespace std;

const int N = 8e5 + 5;
const double INF = 1e18;

void decode(char* s, int len, int mask) {
  for (int i = 0; i < len; ++i) {
    mask = (mask * 131 + i) % len;
    swap(s[i], s[mask]);
  }
}

int q, n, na;
char a[N], t[N];

// SuffixBST(SGT Ver)

// 顺序加入，查询时将询问串翻转
// 以 i 结束的前缀，对应节点的编号为 i
// 注意：不能写懒惰删除，否则可能会破坏树的结构
const double alpha = 0.75;
int root;
int sz[N], L[N], R[N];
double tag[N];
int buffer_size, buffer[N];

bool cmp(int x, int y) {
  if (t[x] != t[y]) return t[x] < t[y];
  return tag[x - 1] < tag[y - 1];
}

void init() { root = 0; }

void new_node(int& rt, int p, double lv, double rv) {
  rt = p;
  sz[rt] = 1;
  tag[rt] = (lv + rv) / 2;
  L[rt] = R[rt] = 0;
}

void push_up(int x) {
  if (!x) return;
  sz[x] = sz[L[x]] + 1 + sz[R[x]];
}
```

```
bool balance(int rt) { return alpha * sz[rt] > max(sz[L[rt]], sz[R[rt]]); }

void flatten(int rt) {
  if (!rt) return;
  flatten(L[rt]);
  buffer[++buffer_size] = rt;
  flatten(R[rt]);
}

void build(int& rt, int l, int r, double lv, double rv) {
  if (l > r) {
    rt = 0;
    return;
  }
  int mid = (l + r) >> 1;
  double mv = (lv + rv) / 2;

  rt = buffer[mid];
  tag[rt] = mv;
  build(L[rt], l, mid - 1, lv, mv);
  build(R[rt], mid + 1, r, mv, rv);
  push_up(rt);
}

void rebuild(int& rt, double lv, double rv) {
  buffer_size = 0;
  flatten(rt);
  build(rt, 1, buffer_size, lv, rv);
}

void insert(int& rt, int p, double lv, double rv) {
  if (!rt) {
    new_node(rt, p, lv, rv);
    return;
  }

  if (cmp(p, rt))
    insert(L[rt], p, lv, tag[rt]);
  else
    insert(R[rt], p, tag[rt], rv);

  push_up(rt);
  if (!balance(rt)) rebuild(rt, lv, rv);
}

void remove(int& rt, int p, double lv, double rv) {
  if (!rt) return;

  if (rt == p) {
    if (!L[rt] || !R[rt]) {
      rt = (L[rt] | R[rt]);
      rebuild(rt, lv, rv);
    } else {
      // 找到 rt 的前驱来替换 rt
```

```
      int nrt = L[rt];
      while (R[nrt]) {
        nrt = R[nrt];
      }
      remove(L[rt], nrt, lv, tag[rt]);
      L[nrt] = L[rt];
      R[nrt] = R[rt];
      rt = nrt;
      tag[rt] = (lv + rv) / 2;
    }
  } else {
    double mv = (lv + rv) / 2;
    if (cmp(p, rt))
      remove(L[rt], p, lv, mv);
    else
      remove(R[rt], p, mv, rv);
  }

  push_up(rt);
  if (!balance(rt)) rebuild(rt, lv, rv);
}

bool cmp1(char* s, int len, int p) {
  for (int i = 1; i <= len; ++i, --p) {
    if (s[i] < t[p]) return true;
    if (s[i] > t[p]) return false;
  }
  return false;
}

int query(int rt, char* s, int len) {
  if (!rt) return 0;
  if (cmp1(s, len, rt))
    return query(L[rt], s, len);
  else
    return sz[L[rt]] + 1 + query(R[rt], s, len);
}

void solve() {
  n = 0;
  scanf("%d", &q);
  init();

  scanf("%s", a + 1);
  na = strlen(a + 1);
  for (int i = 1; i <= na; ++i) {
    t[++n] = a[i];
    insert(root, n, 0, INF);
  }

  int mask = 0;
  char op[10];
  for (int i = 1; i <= q; ++i) {
    scanf("%s", op);
```

```
    // 三种情况分别处理

    if (op[0] == 'A') {  // ADD
      scanf("%s", a + 1);
      na = strlen(a + 1);
      decode(a + 1, na, mask);

      for (int i = 1; i <= na; ++i) {
        t[++n] = a[i];
        insert(root, n, 0, INF);
      }
    } else if (op[0] == 'D') {  // DEL
      int x;
      scanf("%d", &x);
      while (x) {
        remove(root, n, 0, INF);
        --n;
        --x;
      }
    } else if (op[0] == 'Q') {  // QUERY
      scanf("%s", a + 1);
      na = strlen(a + 1);
      decode(a + 1, na, mask);

      reverse(a + 1, a + 1 + na);

      a[na + 1] = 'Z' + 1;
      a[na + 2] = 0;
      int ans = query(root, a, na + 1);

      --a[na];
      ans -= query(root, a, na + 1);

      printf("%d\n", ans);
      mask ^= ans;
    }
  }
}

int main() {
  solve();
  return 0;
}
```

参考资料

- 陈立杰 -《重量平衡树和后缀平衡树在信息学奥赛中的应用》

参考资料与注释

[1] P3809【模板】后缀排序

[2] P6164【模板】后缀平衡树

8.15 广义后缀自动机

前置知识

广义后缀自动机基于下面的知识点

- 字典树（Trie 树）
- 后缀自动机

请务必对上述两个知识点非常熟悉之后，再来阅读本文，特别是对于**后缀自动机**中的**后缀链接**能够有一定的理解

引入

起源

广义后缀自动机是由刘研绎在其 2015 国家队论文《后缀自动机在字典树上的拓展》上提出的一种结构，即将后缀自动机直接建立在字典树上。

> 大部分可以用后缀自动机处理的字符串的问题均可扩展到 Trie 树上。——刘研绎

约定

参考 字符串约定

字符串个数为 k 个，即 $S_1, S_2, S_3 \dots S_k$

约定字典树和广义后缀自动机的根节点为 0 号节点

概述

后缀自动机 (suffix automaton, SAM) 是用于处理单个字符串的子串问题的强力工具。

而广义后缀自动机 (General Suffix Automaton) 则是将后缀自动机整合到字典树中来解决对于多个字符串的子串问题

常见的伪广义后缀自动机

1. 通过用特殊符号将多个串直接连接后，再建立 SAM
2. 对每个串，重复在同一个 SAM 上进行建立，每次建立前，将 `last` 指针置零

方法 1 和方法 2 的实现方式简单，而且在面对题目时通常可以达到和广义后缀自动机一样的正确性。所以在网络上很多人会选择此类写法，例如在后缀自动机一文中最后一个应用，便使用了方法 1（原文链接）

但是无论方法 1 还是方法 2，其时间复杂度较为危险

构造广义后缀自动机

根据原论文的描述，应当在多个字符串上先建立字典树，然后在字典树的基础上建立广义后缀自动机。

字典树的使用

首先应对多个串创建一棵字典树，这不是什么难事，如果你已经掌握了前置知识的前提下，可以很快的建立完毕。这里为了统一上下文的代码，给出一个可能的字典树代码。

" 实现"

```cpp
#define MAXN 2000000
#define CHAR_NUM 30

struct Trie {
  int next[MAXN][CHAR_NUM];  // 转移
  int tot;                   // 节点总数：[0, tot)

  void init() { tot = 1; }

  int insertTrie(int cur, int c) {
    if (next[cur][c]) return next[cur][c];
    return next[cur][c] = tot++;
  }

  void insert(const string &s) {
    int root = 0;
    for (auto ch : s) root = insertTrie(root, ch - 'a');
  }
};
```

这里我们得到了一棵依赖于 `next` 数组建立的一棵字典树。

后缀自动机的建立

如果我们把这样一棵树直接认为是一个后缀自动机，则我们可以得到如下结论

- 对于节点 `i`，其 `len[i]` 和它在字典树中的深度相同
- 如果我们对字典树进行拓扑排序，我们可以得到一串根据 `len` 不递减的序列。BFS 的结果相同

而后缀自动机在建立的过程中，可以视为不断的插入 `len` 严格递增的值，且差值为 1。所以我们可以将对字典树进行拓扑排序后的结果做为一个队列，然后按照这个队列的顺序不断地插入到后缀自动机中。

由于在普通后缀自动机上，其前一个节点的 `len` 值为固定值，即为 `last` 节点的 `len`。但是在广义后缀自动机中，插入的队列是一个不严格递增的数列。所以对于每一个值，对于它的 `last` 应该是已知而且固定的，在字典树上，即为其父亲节点。

由于在字典树中，已经建立了一个近似的后缀自动机，所以只需要对整个字典树的结构进行一定的处理即可转化为广义后缀自动机。我们可以按照前面提出的队列顺序来对整个字典树上的每一个节点进行更新操作。最终我们可以得到广义后缀自动机。

对于每个点的更新操作，我们可以稍微修改一下 SAM 中的插入操作来得到。

对于整个插入的过程，需要注意的是，由于插入是按照 `len` 不递减的顺序插入，在进行 `clone` 后的数据复制过程中，不可以复制其 `len` 小于当前 `len` 的数据。

过程

根据上述的逻辑，可以将整个构建过程描述为如下操作

1. 将所有字符串插入到字典树中

2. 从字典树的根节点开始进行 BFS，记录下顺序以及每个节点的父亲节点
3. 将得到的 BFS 序列按照顺序，对每个节点在原字典树上进行构建，注意不能将 `len` 小于当前 `len` 的数据进行操作

对操作次数为线性的证明

由于仅处理 BFS 得到的序列，可以保证字典树上所有节点仅经过一次。

对于最坏情况，考虑字典树本身节点个数最多的情况，即任意两个字符串没有相同的前缀，则节点个数为 $\sum_{i=1}^{k}|S_i|$，即所有的字符串长度之和。

而在后缀自动机的更新操作的复杂度已经在 后缀自动机 中证明

所以可以证明其最坏复杂度为线性

而通常伪广义后缀自动机的平均复杂度等同于广义后缀自动机的最差复杂度，面对对于大量的字符串时，伪广义后缀自动机的效率远不如标准的广义后缀自动机

实现

对插入函数进行少量必要的修改即可得到所需要的函数

" 参考代码"

```
struct GSA {
  int len[MAXN];             // 节点长度
  int link[MAXN];            // 后缀链接，link
  int next[MAXN][CHAR_NUM];  // 转移
  int tot;                   // 节点总数：[0, tot)

  int insertSAM(int last, int c) {
    int cur = next[last][c];
    len[cur] = len[last] + 1;
    int p = link[last];
    while (p != -1) {
      if (!next[p][c])
        next[p][c] = cur;
      else
        break;
      p = link[p];
    }
    if (p == -1) {
      link[cur] = 0;
      return cur;
    }
    int q = next[p][c];
    if (len[p] + 1 == len[q]) {
      link[cur] = q;
      return cur;
    }
    int clone = tot++;
    for (int i = 0; i < CHAR_NUM; ++i)
      next[clone][i] = len[next[q][i]] != 0 ? next[q][i] : 0;
    len[clone] = len[p] + 1;
    while (p != -1 && next[p][c] == q) {
      next[p][c] = clone;
```

```
      p = link[p];
    }
    link[clone] = link[q];
    link[cur] = clone;
    link[q] = clone;
    return cur;
  }

  void build() {
    queue<pair<int, int>> q;
    for (int i = 0; i < 26; ++i)
      if (next[0][i]) q.push({i, 0});
    while (!q.empty()) {
      auto item = q.front();
      q.pop();
      auto last = insertSAM(item.second, item.first);
      for (int i = 0; i < 26; ++i)
        if (next[last][i]) q.push({i, last});
    }
  }
}
```

- 由于整个 BFS 的过程得到的顺序，其父节点始终在变化，所以并不需要保存 `last` 指针。
- 插入操作中，`int cur = next[last][c];` 与正常后缀自动机的 `int cur = tot++;` 有差异，因为我们插入的节点已经在树型结构中完成了，所以只需要直接获取即可
- 在 `clone` 后的数据拷贝中，有这样的判断 `next[clone][i] = len[next[q][i]] != 0 ? next[q][i] : 0;` 这与正常的后缀自动机的直接赋值 `next[clone][i] = next[q][i];` 有一定差异，此次是为了避免更新了 `len` 大于当前节点的值。由于数组中 `len` 当且仅当这个值被 BFS 遍历并插入到后缀自动机后才会被赋值

性质

1. 广义后缀自动机与后缀自动机的结构一致，在后缀自动机上的性质绝大部分均可在广义后缀自动机上生效（后缀自动机的性质）
2. 当广义后缀自动机建立后，通常字典树结构将会被破坏，即通常不可以用广义后缀自动机来解决字典树问题。当然也可以选择准备双倍的空间，将后缀自动机建立在另外一个空间上。

应用

所有字符中不同子串个数

可以根据后缀自动机的性质得到，以点 i 为结束节点的子串个数等于 $len[i] - len[link[i]]$

所以可以遍历所有的节点求和得到

例题：【模板】广义后缀自动机（广义 SAM）[1]

" 参考代码"

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 2000000;  // 双倍字符串长度
const int CHAR_NUM = 30;   // 字符集个数，注意修改下方的 (-'a')
```

```cpp
struct exSAM {
  int len[MAXN];             // 节点长度
  int link[MAXN];            // 后缀链接，link
  int next[MAXN][CHAR_NUM];  // 转移
  int tot;                   // 节点总数：[0, tot)

  void init() {  // 初始化函数
    tot = 1;
    link[0] = -1;
  }

  int insertSAM(int last, int c) {  // last 为父 c 为子
    int cur = next[last][c];
    if (len[cur]) return cur;
    len[cur] = len[last] + 1;
    int p = link[last];
    while (p != -1) {
      if (!next[p][c])
        next[p][c] = cur;
      else
        break;
      p = link[p];
    }
    if (p == -1) {
      link[cur] = 0;
      return cur;
    }
    int q = next[p][c];
    if (len[p] + 1 == len[q]) {
      link[cur] = q;
      return cur;
    }
    int clone = tot++;
    for (int i = 0; i < CHAR_NUM; ++i)
      next[clone][i] = len[next[q][i]] != 0 ? next[q][i] : 0;
    len[clone] = len[p] + 1;
    while (p != -1 && next[p][c] == q) {
      next[p][c] = clone;
      p = link[p];
    }
    link[clone] = link[q];
    link[cur] = clone;
    link[q] = clone;
    return cur;
  }

  int insertTrie(int cur, int c) {
    if (next[cur][c]) return next[cur][c];  // 已有该节点直接返回
    return next[cur][c] = tot++;            // 无该节点建立节点
  }

  void insert(const string &s) {
    int root = 0;
    for (auto ch : s) root = insertTrie(root, ch - 'a');
```

```
  }

  void insert(const char *s, int n) {
    int root = 0;
    for (int i = 0; i < n; ++i)
      root =
          insertTrie(root, s[i] - 'a');  // 一边插入一边更改所插入新节点的父节点
  }

  void build() {
    queue<pair<int, int>> q;
    for (int i = 0; i < 26; ++i)
      if (next[0][i]) q.push({i, 0});
    while (!q.empty()) {  // 广搜遍历
      auto item = q.front();
      q.pop();
      auto last = insertSAM(item.second, item.first);
      for (int i = 0; i < 26; ++i)
        if (next[last][i]) q.push({i, last});
    }
  }
} exSam;

char s[1000100];

int main() {
  int n;
  cin >> n;
  exSam.init();
  for (int i = 0; i < n; ++i) {
    cin >> s;
    int len = strlen(s);
    exSam.insert(s, len);
  }
  exSam.build();
  long long ans = 0;
  for (int i = 1; i < exSam.tot; ++i) {
    ans += exSam.len[i] - exSam.len[exSam.link[i]];
  }
  cout << ans << endl;
}
```

多个字符串间的最长公共子串

我们需要对每个节点建立一个长度为 k 的数组 `flag`（对于本题而言，可以仅为标记数组，若需要求出此子串的个数，则需要改成计数数组）

在字典树插入字符串时，对所有节点进行计数，保存在当前字符串所在的数组

然后按照 `len` 递减的顺序遍历，通过后缀链接将当前节点的 `flag` 与其他节点的合并

遍历所有的节点，找到一个 `len` 最大且满足对于所有的 `k`，其 `flag` 的值均为非 0 的节点，此节点的 len 即为解

例题：SPOJ Longest Common Substring II[2]

" 参考代码"

```cpp
#include <bits/stdc++.h>
using namespace std;

const int MAXN = 2000000;  // 双倍字符串长度
const int CHAR_NUM = 30;   // 字符集个数，注意修改下方的 (-'a')
const int NUM = 15;        // 字符串个数

struct exSAM {
  int len[MAXN];             // 节点长度
  int link[MAXN];            // 后缀链接，link
  int next[MAXN][CHAR_NUM];  // 转移
  int tot;                   // 节点总数：[0, tot)
  int lenSorted[MAXN];   // 按照 len 排序后的数组，仅排序 [1, tot)
                         // 部分，最终下标范围 [0, tot - 1)
  int sizeC[MAXN][NUM];  // 表示某个字符串的子串个数
  int curString;         // 字符串实际个数
  /**
   * 计数排序使用的辅助空间数组
   */
  int lc[MAXN];  // 统计个数

  void init() {
    tot = 1;
    link[0] = -1;
  }

  int insertSAM(int last, int c) {
    int cur = next[last][c];
    len[cur] = len[last] + 1;
    int p = link[last];
    while (p != -1) {
      if (!next[p][c])
        next[p][c] = cur;
      else
        break;
      p = link[p];
    }
    if (p == -1) {
      link[cur] = 0;
      return cur;
    }
    int q = next[p][c];
    if (len[p] + 1 == len[q]) {
      link[cur] = q;
      return cur;
    }
    int clone = tot++;
    for (int i = 0; i < CHAR_NUM; ++i)
      next[clone][i] = len[next[q][i]] != 0 ? next[q][i] : 0;
    len[clone] = len[p] + 1;
    while (p != -1 && next[p][c] == q) {
      next[p][c] = clone;
      p = link[p];
```

```
    }
    link[clone] = link[q];
    link[cur] = clone;
    link[q] = clone;
    return cur;
  }

  int insertTrie(int cur, int c) {
    if (!next[cur][c]) next[cur][c] = tot++;
    sizeC[next[cur][c]][curString]++;
    return next[cur][c];
  }

  void insert(const string &s) {
    int root = 0;
    for (auto ch : s) root = insertTrie(root, ch - 'a');
    curString++;
  }

  void insert(const char *s, int n) {
    int root = 0;
    for (int i = 0; i < n; ++i) root = insertTrie(root, s[i] - 'a');
    curString++;
  }

  void build() {
    queue<pair<int, int>> q;
    for (int i = 0; i < 26; ++i)
      if (next[0][i]) q.push({i, 0});
    while (!q.empty()) {  // 广搜遍历
      auto item = q.front();
      q.pop();
      auto last = insertSAM(item.second, item.first);
      for (int i = 0; i < 26; ++i)
        if (next[last][i]) q.push({i, last});
    }
  }

  void sortLen() {
    for (int i = 1; i < tot; ++i) lc[i] = 0;
    for (int i = 1; i < tot; ++i) lc[len[i]]++;
    for (int i = 2; i < tot; ++i) lc[i] += lc[i - 1];
    for (int i = 1; i < tot; ++i) lenSorted[--lc[len[i]]] = i;
  }

  void getSizeLen() {
    for (int i = tot - 2; i >= 0; --i)
      for (int j = 0; j < curString; ++j)
        sizeC[link[lenSorted[i]]][j] += sizeC[lenSorted[i]][j];
  }
} exSam;

int main() {
  exSam.init();  // 初始化
```

```
  string s;
  while (cin >> s) exSam.insert(s);
  exSam.build();
  exSam.sortLen();
  exSam.getSizeLen();
  int ans = 0;
  for (int i = 0; i < exSam.tot; ++i) {
    bool flag = true;
    for (int j = 0; j < exSam.curString; ++j) {
      if (!exSam.sizeC[i][j]) {
        flag = false;
        break;
      }
    }
    if (flag) ans = max(ans, exSam.len[i]);
  }
  cout << ans << endl;
}
```

参考资料与注释

[1] 【模板】广义后缀自动机（广义 SAM）

[2] SPOJ Longest Common Substring II

8.16 后缀树

后缀树是一种维护一个字符串所有后缀的数据结构。

一些记号

记构建后缀树的母串为 S，长度为 n，字符集为 Σ。

令 $S[i]$ 表示 S 中的第 i 个字符，其中 $1 \le i \le n$。

令 $S[l,r]$ 表示 S 中第 l 个字符至第 r 个字符组成的字符串，称为 S 的一个子串。

记 $S[i,n]$ 为 S 的以 i 开头的后缀，$S[1,i]$ 为 S 的以 i 结尾的前缀。

定义

定义字符串 S 的**后缀 trie** 为将 S 的所有后缀插入至 trie 树中得到的字典树。在后缀 trie 中，节点 x 对应的字符串为从根节点走到 x 的路径上经过的字符拼接而成的字符串。记后缀 trie 中所有对应 S 的某个后缀的节点为后缀节点。

容易看出后缀 trie 的优越性质：它的非根节点恰好能接受 S 的所有本质不同非空子串。但构建后缀 trie 的时空复杂度均为 $O(n^2)$，在很多情况下不能接受，所以我们引入后缀树的概念。

如果令后缀 trie 中所有拥有多于一个儿子的节点和后缀节点为关键点，定义只保留关键点，将非关键点形成的链压缩成一条边形成的压缩 trie 树为**后缀树 (Suffix Tree)**。如果仅令后缀 trie 中所有拥有多于一个儿子的节点和叶结点为关键点，定义只保留关键点形成的压缩 trie 树为**隐式后缀树 (Implicit Suffix Tree)**。容易看出隐式后缀树为后缀树进一步压缩后得到的结果。

在后缀树和隐式后缀树中，每条边对应一个字符串；每个非根节点 x 对应了一个字符串集合，为从根节点走到 x

的父亲节点 fa_x 经过的字符串，拼接上 fa_x 至 x 的树边对应的字符串的任意一个非空前缀，称为 str_x。同时，在隐式后缀树中，称一个没有对应任何节点的后缀为**隐式后缀**。

下图从左至右分别为以字符串 `cabab` 为母串构建的后缀 trie、后缀树和隐式后缀树。

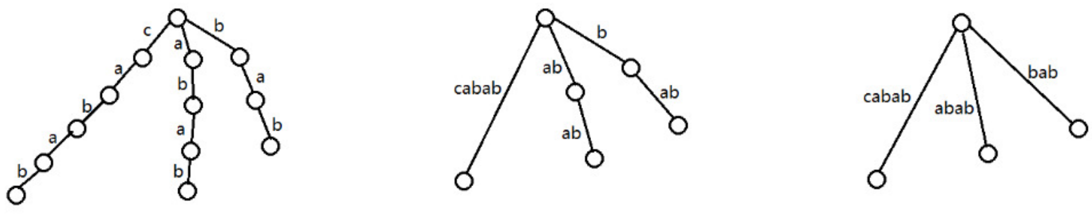


图 8.21 suffix-tree_cabab1.png

考虑将 S 的后缀逐个插入至后缀 trie 中。从第二次插入开始，每次最多新增一个拥有多于一个儿子的节点和一个后缀节点，所以后缀树中节点个数最多为 $2n$ 个，十分优秀。

后缀树的建立

支持前端动态添加字符的算法

反串建 SAM 建出的 parent 树就是这个串的后缀树，所以我们将反串的字符逐个加入 SAM 即可。

" 参考实现"

```
struct SuffixAutomaton {
  int tot, lst;
  int siz[N << 1];
  int buc[N], id[N << 1];

  struct Node {
    int len, link;
    int ch[26];
  } st[N << 1];

  SuffixAutomaton() : tot(1), lst(1) {}

  void extend(int ch) {
    int cur = ++tot, p = lst;
    lst = cur;
    siz[cur] = 1, st[cur].len = st[p].len + 1;
    for (; p && !st[p].ch[ch]; p = st[p].link) st[p].ch[ch] = cur;
    if (!p)
      st[cur].link = 1;
    else {
      int q = st[p].ch[ch];
      if (st[q].len == st[p].len + 1)
        st[cur].link = q;
      else {
        int pp = ++tot;
        st[pp] = st[q];
        st[pp].len = st[p].len + 1;
```

```
        st[cur].link = st[q].link = pp;
        for (; p && st[p].ch[ch] == q; p = st[p].link) st[p].ch[ch] = pp;
      }
    }
  }
} SAM;
```

支持后端动态添加字符的算法

Ukkonen 算法是一种增量构造算法。我们依次向树中插入串 S 的每一个字符，并在每一次插入之后正确地维护当前的后缀树。

朴素算法

首先介绍一下一种较为暴力的构建方式，我们用字符串 `abbbc` 来演示一下构建的过程。

初始建立一个根节点，称为 0 号节点。同时每条边我们维护一个区间 $[l, r]$ 表示这条边上的字符串为 $S[l, r]$。另外，维护已经插入的字符个数 m，初始为 0。

首先插入字符 `a`，直接从 0 号节点伸出一条边，标为 $[1, \infty]$，指向一个新建的节点。这里的 ∞ 是一个极大值，可理解为串的结尾，这样在插入新字符时，这条边会自动的包含新的字符。

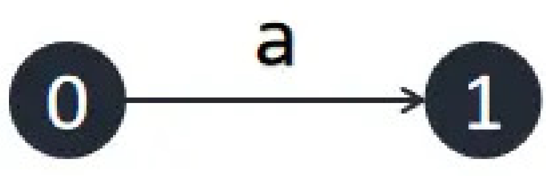


图 8.22 suffix-tree_a.webp

接下来我们插入字符 `b`，同样从 0 伸出一条边，标为 $[2, \infty]$。注意到之前延伸出的边 $[1, \infty]$ 的意义自动地发生了变化，随着串结尾的改变，其表示的串从 `a` 变为了 `ab`。这样是正确的，因为之前所有后缀都已经以一个叶节点的形式出现在树中，只需要向所有叶节点的末端插入一个当前字符即可。

接下来，我们要再次插入一个字符 `b`，但是 `b` 是之前已经插入的字符串的一个子串，因此原树已经包含 `b`，此时，我们什么都不做，记录一个 k 表示 $S[k, m]$ 是当前最长的隐式后缀。

接下来我们插入另一个 `b`。因为前一个 `b` 没有插入成功，此时 $k = 3$，代表要插入的后缀为 `bb`。我们从根开始向下寻找 `bb`，发现也在原树之中。同样，我们还是什么都不做。

注意到我们没有管 k 之后的后缀。因为如果 $S[k, m]$ 是一个隐式后缀，那么对于 $l > k$，$S[l, m]$ 都是隐式后缀。因为由 $S[k, m]$ 为隐式后缀可知，存在字符 c 使得 $S[k, m] + c$ 为 S 的子串，所以 $S[l, m] + c$ 也为 S 的子串，由隐式后缀树的定义可知 $S[l, m]$ 也不作为叶结点出现。

接下来我们插入 `c`，此时 $k = 3$，因此我们需要沿着根向下寻找 `bbc`，发现不在原树中。我们需要在 `bb` 处代表的节点延伸出一条为 $[5, \infty]$ 的出边。但发现这个节点其实不存在，而是包含在一条边中，因此我们需要分裂这条边，创建一个新节点，再在创建的节点处伸展出我们要创建的出边。此时成功插入，令 $k \to k + 1$，因为 $S[k, m]$ 不再是隐式后缀。

接下来，因为 k 变化了，我们重复这个过程，直到再次出现隐式后缀，或 $k > m$（在这个例子中，是后者）。

构建过程结束。

该算法每次暴力从根向下寻找并插入的复杂度最坏为 $O(n)$，所以总的复杂度为 $O(n^2)$。

后缀链接

朴素算法慢主要是因为每次 extend 都要从根找到最长隐式后缀的插入位置。所以考虑把这个位置记下来。首先，我们采用一个二元组 (now, rem) 来描述当前这个最长的被隐式包含的后缀 $S[k, m]$。沿着节点 now 的开头为

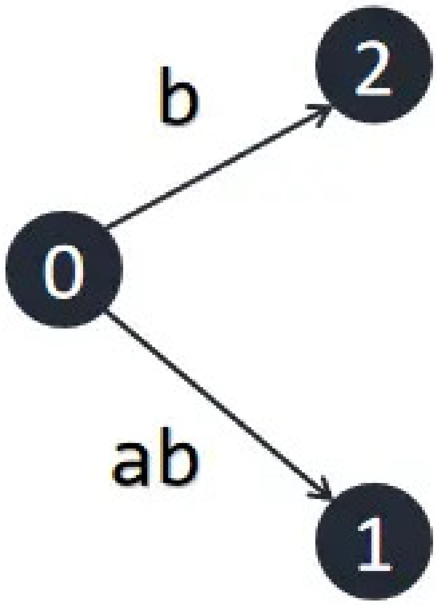


图 8.23 suffix-tree_ab.webp

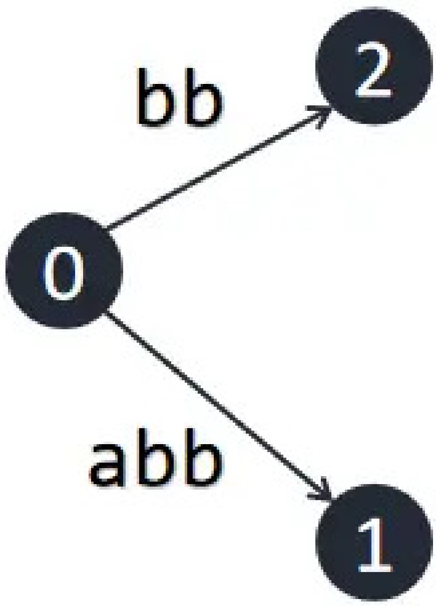


图 8.24 suffix-tree_abb.webp

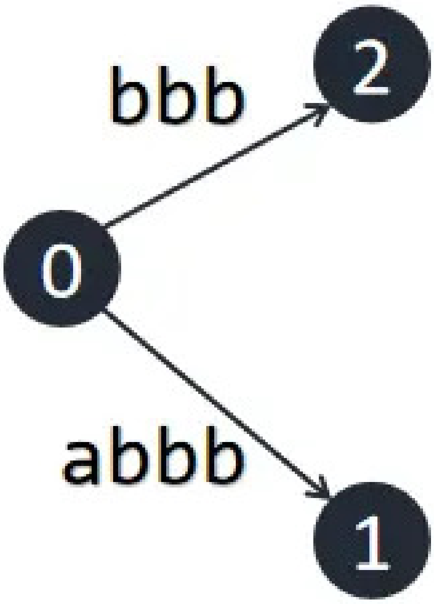


图 8.25 suffix-tree_abbb.webp

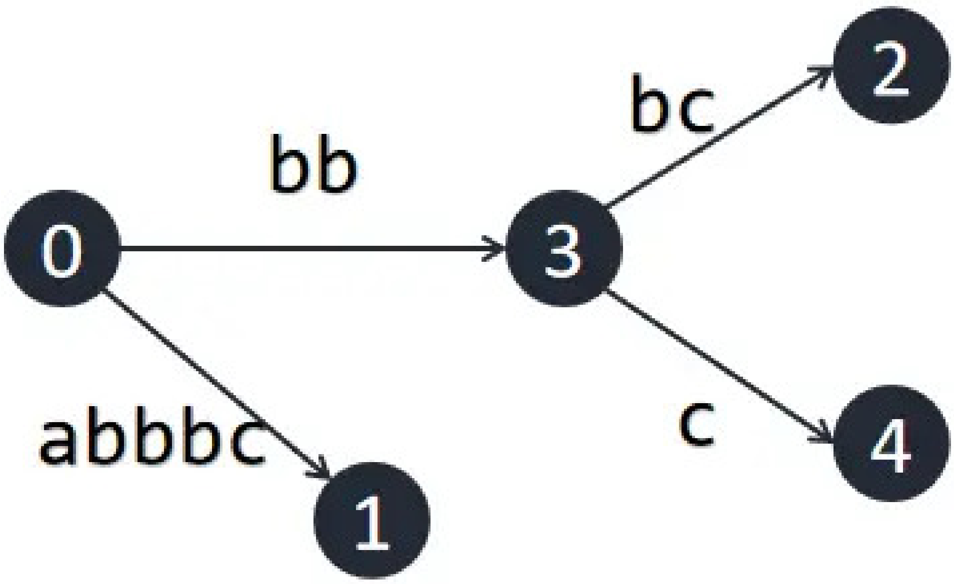


图 8.26 suffix-tree_abbbc1.webp

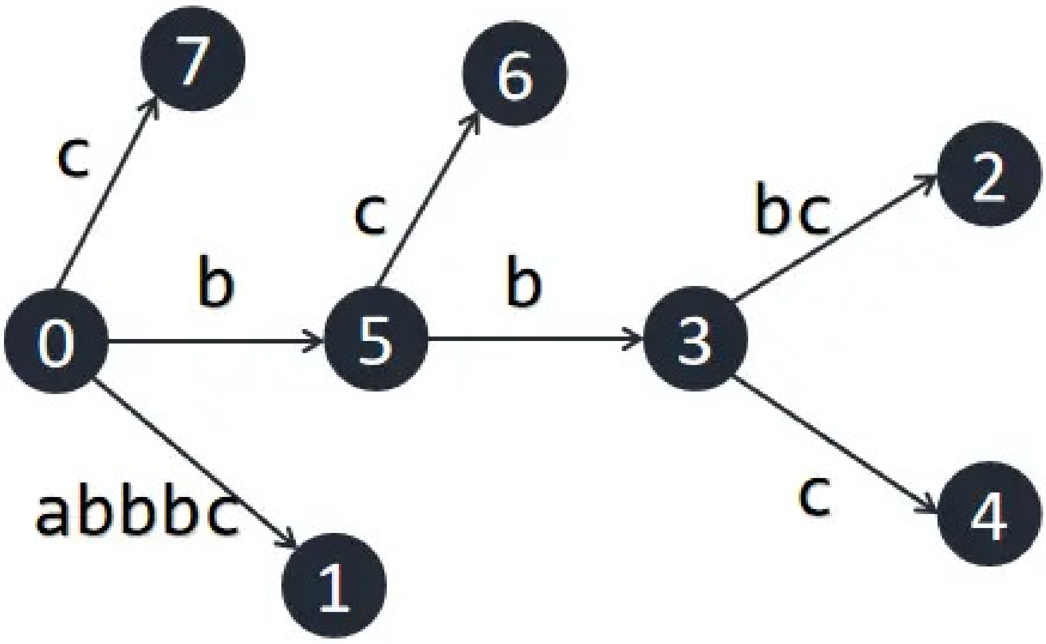


图 8.27 suffix-tree_abbbc2.webp

$S[m-rem+1]$ 的出边走长度 rem 到达的位置应该唯一表示一个字符串，每次插入新的字符时，我们只需要从 now 和 rem 描述的位置查找即可。

现在，我们只需要在 $k \to k+1$ 时更新 (now, rem)。此时如果 $now = 0$，只需要让 $rem \to rem-1$，因为下一个要插入的后缀是刚才插入的长度 -1。否则，设 str_{now} 对应的子串为 $S[l, r]$，我们需要找到一个节点 now' 对应 $S[l+1, r]$，令 $now \to now'$ 即可。

首先有引理：对隐式后缀树中任意非叶非根节点 x，在树中存在另一非叶节点 y，使得 str_y 是 str_x 对应的子串删去开头的字符。

证明。令 s 表示 str_x 删去开头字符形成的字符串。由隐式后缀树的定义可知，存在两个不同的字符 c_1, c_2，满足 $str_x + c1$ 与 $str_x + c_2$ 均为 S 的子串。所以，$s + c_1$ 与 $s + c_2$ 也为 S 的子串，所以 s 在后缀 trie 中也对应了一个有分叉的关键点，即在隐式后缀 trie 中存在 y 使得 $str_y = s$。证毕。

由该引理，我们定义 $\mathrm{Link}(x) = y$，称为 x 的**后缀链接 (Suffix Link)**。于是 $now' = \mathrm{Link}(now)$ 一定存在。现在我们只要能求出隐式后缀树中所有非根非叶节点的 Link 即可。

Ukkonen 算法

Ukkonen 算法的整体流程如下：

为了构建隐式后缀树，我们从前往后加入 S 中的字符。假设根节点为 0，且当前已经建出 $S[1, m]$ 的隐式后缀树且维护好了后缀链接。$S[1, m]$ 的最长隐式后缀为 $S[k, m]$，在树中的位置为 (now, rem)。设 $S[m+1] = x$, 现在我们需要加入字符 x。此时，$S[1, m]$ 的每一个后缀都需要在末尾添加字符 x。由于所有显式后缀都对应树中某个叶结点，它们父边右端点为 ∞，无需维护。所以，现在我们只用考虑隐式后缀末尾添加 x 对树的形态产生的影响。首先考虑 $S[k, m]$，有两种情况：

1. (now, rem) 位置已经存在 x 的转移。此时后缀树形态不会发生变化。由于 $S[k, m+1]$ 已经在后缀树中出现，所以对于 $l > k$，$S[l, m+1]$ 也会在后缀树中出现，此时只需将 $rem \to rem+1$，不需做任何修改。
2. (now, rem) 不存在 x 的转移。如果 (now, rem) 恰好为树中的节点，则此节点新增一条出边 x；否则需要对节点进行分裂，在此位置新增一个节点，并在新增节处添加出边 x。此时对于 $l > k$，我们并不知道 $S[l, m]$ 会对后缀树形态造成什么影响，所以我们还需继续考虑 $S[k+1, m]$。考虑怎么求出 $S[k+1, m]$ 在后缀树中的位置：如果 now 不为 0，可以利用后缀链接，令 $now = \mathrm{Link}(now)$；否则，令 $rem \to rem-1$。最后令 $k \to k+1$，再次重复这个过程。

每一步都只消耗常数时间，而算法在插入全部的字符后停止，所以时间复杂度为 $O(n)$。

由于 Ukkonen 算法只能处理出 S 的隐式后缀树，而隐式后缀树在一些问题中的功能可能不如后缀树强大，所以在需要时，可以在 S 的末端添加一个从未出现过的字符，这时 S 的所有后缀可以和树的所有叶子一一对应。

" 参考实现"

```cpp
struct SuffixTree {
  int ch[M + 5][RNG + 1], st[M + 5], len[M + 5], link[M + 5];
  int s[N + 5];
  int now{1}, rem{0}, n{0}, tot{1};

  SuffixTree() { len[0] = inf; }

  int new_node(int s, int le) {
    ++tot;
    st[tot] = s;
    len[tot] = le;
    return tot;
  }

  void extend(int x) {
    s[++n] = x;
    ++rem;
    for (int lst{1}; rem;) {
      while (rem > len[ch[now][s[n - rem + 1]]])
        rem -= len[now = ch[now][s[n - rem + 1]]];
      int &v{ch[now][s[n - rem + 1]]}, c{s[st[v] + rem - 1]};
      if (!v || x == c) {
        lst = link[lst] = now;
        if (!v)
          v = new_node(n, inf);
        else
          break;
      } else {
        int u{new_node(st[v], rem - 1)};
        ch[u][c] = v;
        ch[u][x] = new_node(n, inf);
        st[v] += rem - 1;
        len[v] -= rem - 1;
        lst = link[lst] = v = u;
      }
      if (now == 1)
        --rem;
      else
        now = link[now];
    }
  }
} Tree;
```

作用

后缀树上每一个节点到根的路径都是 S 的一个非空子串，这在处理很多字符串问题时都很有用。

后缀树的 DFS 序就是后缀数组。后缀树的一个子树也就对应到后缀数组上的一个区间。后缀树上两个后缀的最

长公共前缀是它们对应的叶节点的 LCA，因此，后缀数组的 height 的结论可以理解为树上若干个节点的 LCA 等于 DFS 序最小的和最大的节点的 LCA。

例题

洛谷 P3804【模板】后缀自动机（SAM）[1]

题意：

给定一个只包含小写字母的字符串 S。

请你求出 S 的所有出现次数不为 1 的子串的出现次数乘上该子串长度的最大值。

" 解法"

建出插入一个终止符的隐式后缀树。树上每条从根出发的路径都构成子串。一个显示后缀的出现次数即为对应节点子树内的叶子节点个数，隐式后缀不用考虑，因为一个隐式后缀的出现次数等于向下走到的第一个节点对应显示后缀的出现次数，而且一定没有该显示后缀长。所以遍历整棵树，求出每个节点子树内叶子个数和每个节点到根的路径长度。如果叶子个数 >1 则更新答案。复杂度 $O(|S||\Sigma|)$。

" 参考代码"

```cpp
#include <bits/stdc++.h>
using namespace std;
constexpr int N(1e6), M(2 * N), inf(1e7), RNG{26};

struct SuffixTree {
  int ch[M + 5][RNG + 1], st[M + 5], len[M + 5], link[M + 5];
  int s[N + 5];
  int now{1}, rem{0}, n{0}, tot{1};

  SuffixTree() { len[0] = inf; }

  int new_node(int s, int le) {
    ++tot;
    st[tot] = s;
    len[tot] = le;
    return tot;
  }

  void extend(int x) {
    s[++n] = x;
    ++rem;
    for (int lst{1}; rem;) {
      while (rem > len[ch[now][s[n - rem + 1]]])
        rem -= len[now = ch[now][s[n - rem + 1]]];
      int &v{ch[now][s[n - rem + 1]]}, c{s[st[v] + rem - 1]};
      if (!v || x == c) {
        lst = link[lst] = now;
        if (!v)
          v = new_node(n, inf);
        else
          break;
      } else {
        int u{new_node(st[v], rem - 1)};
```

```
      ch[u][c] = v;
      ch[u][x] = new_node(n, inf);
      st[v] += rem - 1;
      len[v] -= rem - 1;
      lst = link[lst] = v = u;
    }
    if (now == 1)
      --rem;
    else
      now = link[now];
  }
}

pair<long long, int> search(int u, int dep = 0) {
  if (st[u] + len[u] >= n) return {0, 1};
  dep += len[u];
  long long ans{0};
  int ys{0};
  for (int i{0}; i <= RNG; ++i)
    if (ch[u][i]) {
      auto [__, _]{search(ch[u][i], dep)};
      ans = max(ans, __);
      ys += _;
    }
  if (ys > 1) ans = max(ans, 1LL * dep * ys);
  return {ans, ys};
}
} T;

char s[N + 5];

int main() {
  scanf("%s", s + 1);
  for (int i{1}; s[i]; ++i) T.extend(s[i] - 'a' + 1);
  T.extend(0);
  cout << T.search(1).first << endl;
  return 0;
}
```

CF235C Cyclical Quest[2]

题意：给定一个小写字母主串 S 和 n 个询问串，求每个询问串 x_i 的所有循环同构在主串中出现的次数总和。

" 解法"

建立插入终止符的隐式后缀树。

枚举当前在那个循环节，记录在树上能查找到多长的前缀。

重复类似 Ukkonen 算法的过程，记录当前能匹配到的位置 (now, rem)。每次尝试插入下一个字符，如果成功则继续插入，否则跳出循环。

如果某一个次成功匹配了当前的循环节，且该循环节之前没出现过，则更新答案。

然后切换到下个循环节的时候，我们要删去当前匹配的子串开头的字符：这正好就相当于令 $now \to \mathrm{Link}(now)$。当然，如果 $now = 1$ 则直接让 $rem \to rem - 1$ 就行了。

复杂度 $O(|S||\Sigma| + \sum |x_i|)$

" 参考代码"

```cpp
#include <bits/stdc++.h>
using namespace std;
constexpr int N(1e6);

struct SuffixTree {
  static constexpr int N{2 * ::N}, RNG{26}, inf = 1e7;
  int ch[N + 5][RNG + 1];
  int st[N + 5], len[N + 5], link[N + 5], s[::N + 5];
  int now{1}, rem{0}, tot{1}, n{0};
  int cnt[N + 5], vis[N + 5];

  SuffixTree() { len[0] = inf; }

  void clear() {
    memset(ch, 0, sizeof ch);
    now = tot = 1;
    rem = n = 0;
  }

  int new_node(int s, int le) {
    ++tot;
    st[tot] = s;
    len[tot] = le;
    return tot;
  }

  void extend(int x) {
    s[++n] = x;
    ++rem;
    for (int lst{1}; rem;) {
      while (rem > len[ch[now][s[n - rem + 1]]])
        rem -= len[now = ch[now][s[n - rem + 1]]];
      int &v{ch[now][s[n - rem + 1]]}, c{s[st[v] + rem - 1]};
      if (!v || x == c) {
        lst = link[lst] = now;
        if (!v)
          v = new_node(n, inf);
        else
          break;
      } else {
        int u{new_node(st[v], rem - 1)};
        ch[u][c] = v;
        ch[u][x] = new_node(n, inf);
        st[v] += rem - 1;
        len[v] -= rem - 1;
        lst = link[lst] = v = u;
      }
      if (now == 1)
        --rem;
      else
        now = link[now];
    }
```

```cpp
  }

  void init(int u) {
    if (len[u] > 1e6) return cnt[u] = 1, void();
    for (int i{0}; i <= RNG; ++i)
      if (ch[u][i]) init(ch[u][i]), cnt[u] += cnt[ch[u][i]];
  }

  long long test(const char *t, int m) {
    static int time{0};
    ++time;
    int now{1}, rem{0}, o{0};
    long long ans{0};
    for (int i{1}; i <= m; ++i) {
      while (o < i + m - 1) {
        while (rem >= len[ch[now][t[o - rem + 1]]])
          rem -= len[now = ch[now][t[o - rem + 1]]];
        int v{ch[now][t[o - rem + 1]]}, c{s[st[v] + rem]};
        if (v && c == t[o + 1]) {
          ++o;
          ++rem;
        } else {
          break;
        }
      }
      if (o == i + m - 1 && vis[ch[now][t[o - rem + 1]]] != time)
        ans += cnt[ch[now][t[o - rem + 1]]],
            vis[ch[now][t[o - rem + 1]]] = time;
      if (now == 1)
        --rem;
      else
        now = link[now];
    }
    return ans;
  }
} T;

char s[N * 2 + 5];

int main() {
  scanf("%s", s + 1);
  for (int i{1}; s[i]; ++i) T.extend(s[i] - 'a' + 1);
  T.extend(0);
  T.init(1);
  int pw;
  cin >> pw;
  while (pw--) {
    scanf("%s", s + 1);
    int n{strlen(s + 1)};
    for (int i{1}; i <= n; ++i) s[i] += 1 - 'a';
    copy(s + 1, s + 1 + n, s + 1 + n);
    cout << T.test(s, n) << "\n";
  }
  return 0;
}
```

参考文献

1. 2021 国家集训队论文《后缀树的构建》代晨昕
2. 炫酷后缀树魔术 - EternalAlexander 的博客[3]

参考资料与注释

[1] 洛谷 P3804【模板】后缀自动机（SAM）

[2] CF235C Cyclical Quest

[3] 炫酷后缀树魔术 - EternalAlexander 的博客

8.17 Manacher

描述

给定一个长度为 n 的字符串 s，请找到所有对 (i,j) 使得子串 $s[i\dots j]$ 为一个回文串。当 $t=t_{\text{rev}}$ 时，字符串 t 是一个回文串（t_{rev} 是 t 的反转字符串）。

解释

显然在最坏情况下可能有 $O(n^2)$ 个回文串，因此似乎一眼看过去该问题并没有线性算法。

但是关于回文串的信息可用**一种更紧凑的方式**表达：对于每个位置 $i=0\dots n-1$，我们找出值 $d_1[i]$ 和 $d_2[i]$。二者分别表示以位置 i 为中心的长度为奇数和长度为偶数的回文串个数。换个角度，二者也表示了以位置 i 为中心的最长回文串的半径长度（半径长度 $d_1[i]$，$d_2[i]$ 均为从位置 i 到回文串最右端位置包含的字符个数）。

举例来说，字符串 $s=$ `abababc` 以 $s[3]=b$ 为中心有三个奇数长度的回文串，最长回文串半径为 3，也即 $d_1[3]=3$：

$$a\ \overbrace{b\ a\ \underset{s_3}{b}\ a\ b}^{d_1[3]=3}\ c$$

字符串 $s=$ `cbaabd` 以 $s[3]=a$ 为中心有两个偶数长度的回文串，最长回文串半径为 2，也即 $d_2[3]=2$：

$$c\ \overbrace{b\ a\ \underset{s_3}{a}\ b}^{d_2[3]=2}\ d$$

因此关键思路是，如果以某个位置 i 为中心，我们有一个长度为 l 的回文串，那么我们有以 i 为中心的长度为 $l-2$，$l-4$，等等的回文串。所以 $d_1[i]$ 和 $d_2[i]$ 两个数组已经足够表示字符串中所有子回文串的信息。

一个令人惊讶的事实是，存在一个复杂度为线性并且足够简单的算法计算上述两个「回文性质数组」$d_1[]$ 和 $d_2[]$。在这篇文章中我们将详细的描述该算法。

解法

总的来说，该问题具有多种解法：应用字符串哈希，该问题可在 $O(n\log n)$ 时间内解决，而使用后缀数组和快速 LCA 该问题可在 $O(n)$ 时间内解决。

但是这里描述的算法**压倒性**的简单，并且在时间和空间复杂度上具有更小的常数。该算法由 **Glenn K. Manacher** 在 1975 年提出。

朴素算法

为了避免在之后的叙述中出现歧义，这里我们指出什么是「朴素算法」。

该算法通过下述方式工作：对每个中心位置 i，在比较一对对应字符后，只要可能，该算法便尝试将答案加 1。

该算法是比较慢的：它只能在 $O(n^2)$ 的时间内计算答案。

该朴素算法的实现如下：

" 实现"

```
vector<int> d1(n), d2(n);
for (int i = 0; i < n; i++) {
  d1[i] = 1;
  while (0 <= i - d1[i] && i + d1[i] < n && s[i - d1[i]] == s[i + d1[i]]) {
    d1[i]++;
  }

  d2[i] = 0;
  while (0 <= i - d2[i] - 1 && i + d2[i] < n &&
         s[i - d2[i] - 1] == s[i + d2[i]]) {
    d2[i]++;
  }
}
```

```
d1 = [0] * n
d2 = [0] * n
for i in range(0, n):
    d1[i] = 1
    while 0 <= i - d1[i] and i + d1[i] < n and s[i - d1[i]] == s[i + d1[i]]:
        d1[i] += 1

    d2[i] = 0
    while 0 <= i - d2[i] - 1 and i + d2[i] < n and s[i - d2[i] - 1] == s[i + d2[
i]]:
        d2[i] += 1
```

Manacher 算法

这里我们将只描述算法中寻找所有奇数长度子回文串的情况，即只计算 $d_1[]$；寻找所有偶数长度子回文串的算法（即计算数组 $d_2[]$）将只需对奇数情况下的算法进行一些小修改。

为了快速计算，我们维护已找到的最靠右的子回文串的**边界** (l, r)（即具有最大 r 值的回文串，其中 l 和 r 分别为该回文串左右边界的位置）。初始时，我们置 $l = 0$ 和 $r = -1$（*-1* 需区别于倒序索引位置，这里可为任意负数，仅为了循环初始时方便）。

过程

现在假设我们要对下一个 i 计算 $d_1[i]$，而之前所有 $d_1[]$ 中的值已计算完毕。我们将通过下列方式计算：

- 如果 i 位于当前子回文串之外，即 $i > r$，那么我们调用朴素算法。

  因此我们将连续地增加 $d_1[i]$，同时在每一步中检查当前的子串 $[i - d_1[i] \dots i + d_1[i]]$（$d_1[i]$ 表示半径长度，下同）是否为一个回文串。如果我们找到了第一处对应字符不同，又或者碰到了 s 的边界，则算法停止。在两种情况下我们均已计算完 $d_1[i]$。此后，仍需记得更新 (l, r)。
- 现在考虑 $i \le r$ 的情况。我们将尝试从已计算过的 $d_1[]$ 的值中获取一些信息。首先在子回文串 (l, r) 中反转位置

i，即我们得到 $j = l + (r - i)$。现在来考察值 $d_1[j]$。因为位置 j 同位置 i 对称，我们**几乎总是**可以置 $d_1[i] = d_1[j]$。该想法的图示如下（可认为以 j 为中心的回文串被「拷贝」至以 i 为中心的位置上）：

$$\ldots\ \overbrace{s_l\ \ldots\ \underbrace{s_{j-d_1[j]+1}\ \ldots\ s_j\ \ldots\ s_{j+d_1[j]-1}}_{\text{palindrome}}\ \ldots\ \underbrace{s_{i-d_1[j]+1}\ \ldots\ s_i\ \ldots\ s_{i+d_1[j]-1}}_{\text{palindrome}}\ \ldots\ s_r}^{\text{palindrome}}\ \ldots$$

然而有一个**棘手的情况**需要被正确处理：当「内部」的回文串到达「外部」回文串的边界时，即 $j - d_1[j] + 1 \le l$（或者等价的说，$i + d_1[j] - 1 \ge r$）。因为在「外部」回文串范围以外的对称性没有保证，因此直接置 $d_1[i] = d_1[j]$ 将是不正确的：我们没有足够的信息来断言在位置 i 的回文串具有同样的长度。

实际上，为了正确处理这种情况，我们应该「截断」回文串的长度，即置 $d_1[i] = r - i$。之后我们将运行朴素算法以尝试尽可能增加 $d_1[i]$ 的值。

该种情况的图示如下（以 j 为中心的回文串已经被截断以落在「外部」回文串内）：

$$\ldots\ \overbrace{\underbrace{s_l\ \ldots\ s_j\ \ldots\ s_{j+(j-l)}}_{\text{palindrome}}\ \ldots\ \underbrace{s_{i-(r-i)}\ \ldots\ s_i\ \ldots\ s_r}_{\text{palindrome}}}^{\text{palindrome}}\ \underbrace{\ldots\ldots\ldots\ldots\ldots}_{\text{try moving here}}$$

该图示显示出，尽管以 j 为中心的回文串可能更长，以致于超出「外部」回文串，但在位置 i，我们只能利用其完全落在「外部」回文串内的部分。然而位置 i 的答案可能比这个值更大，因此接下来我们将运行朴素算法来尝试将其扩展至「外部」回文串之外，也即标识为"try moving here" 的区域。

最后，仍有必要提醒的是，我们应当记得在计算完每个 $d_1[i]$ 后更新值 (l, r)。

同时，再让我们重复一遍：计算偶数长度回文串数组 $d_2[]$ 的算法同上述计算奇数长度回文串数组 $d_1[]$ 的算法十分类似。

Manacher 算法的复杂度

因为在计算一个特定位置的答案时我们总会运行朴素算法，所以一眼看去该算法的时间复杂度为线性的事实并不显然。

然而更仔细的分析显示出该算法具有线性复杂度。此处我们需要指出，计算 Z 函数的算法 和该算法较为类似，并同样具有线性时间复杂度。

实际上，注意到朴素算法的每次迭代均会使 r 增加 1，以及 r 在算法运行过程中从不减小。这两个观察告诉我们朴素算法总共会进行 $O(n)$ 次迭代。

Manacher 算法的另一部分显然也是线性的，因此总复杂度为 $O(n)$。

Manacher 算法的实现

分类讨论

为了计算 $d_1[]$，我们有以下代码：

```
vector<int> d1(n);
for (int i = 0, l = 0, r = -1; i < n; i++) {
  int k = (i > r) ? 1 : min(d1[l + r - i], r - i + 1);
  while (0 <= i - k && i + k < n && s[i - k] == s[i + k]) {
    k++;
  }
  d1[i] = k--;
  if (i + k > r) {
    l = i - k;
    r = i + k;
  }
```

```
}
```

```python
d1 = [0] * n
l, r = 0, -1
for i in range(0, n):
    k = 1 if i > r else min(d1[l + r - i], r - i + 1)
    while 0 <= i - k and i + k < n and s[i - k] == s[i + k]:
        k += 1
    d1[i] = k
    k -= 1
    if i + k > r:
        l = i - k
        r = i + k
```

计算 $d_2[]$ 的代码十分类似，但是在算术表达式上有些许不同：

```cpp
vector<int> d2(n);
for (int i = 0, l = 0, r = -1; i < n; i++) {
  int k = (i > r) ? 0 : min(d2[l + r - i + 1], r - i + 1);
  while (0 <= i - k - 1 && i + k < n && s[i - k - 1] == s[i + k]) {
    k++;
  }
  d2[i] = k--;
  if (i + k > r) {
    l = i - k - 1;
    r = i + k;
  }
}
```

```python
d2 = [0] * n
l, r = 0, -1
for i in range(0, n):
    k = 0 if i > r else min(d2[l + r - i + 1], r - i + 1)
    while 0 <= i - k - 1 and i + k < n and s[i - k - 1] == s[i + k]:
        k += 1
    d2[i] = k
    k -= 1
    if i + k > r:
        l = i - k - 1
        r = i + k
```

统一处理

虽然在讲解过程及上述实现中我们将 $d_1[]$ 和 $d_2[]$ 的计算分开考虑，但实际上可以通过一个技巧将二者的计算统一为 $d_1[]$ 的计算。

给定一个长度为 n 的字符串 s，我们在其 $n+1$ 个空中插入分隔符 $\#$，从而构造一个长度为 $2n+1$ 的字符串 s'。举例来说，对于字符串 $s=$ abababc，其对应的 $s'=$ #a#b#a#b#a#b#c#。

对于字母间的 $\#$，其实际意义为 s 中对应的「空」。而两端的 $\#$ 则是为了实现的方便。

注意到，在对 s' 计算 $d_1[]$ 后，对于一个位置 i，$d_1[i]$ 所描述的最长的子回文串必定以 $\#$ 结尾（若以字母结尾，由于字母两侧必定各有一个 $\#$，因此可向外扩展一个得到一个更长的）。因此，对于 s 中一个以字母为中心的极大子回文串，设其长度为 $m+1$，则其在 s' 中对应一个以相应字母为中心，长度为 $2m+3$ 的极大子回文串；而对于 s 中一个以空为中心的极大子回文串，设其长度为 m，则其在 s' 中对应一个以相应表示空的 $\#$ 为中心，长度为 $2m+1$ 的极大子回文串（上述两种情况下的 m 均为偶数，但该性质成立与否并不影响结论）。综合以上观察及少许计算后易得，在 s' 中，$d_1[i]$ 表示在 s 中以对应位置为中心的极大子回文串的**总长度加一**。

上述结论建立了 s' 的 $d_1[]$ 同 s 的 $d_1[]$ 和 $d_2[]$ 间的关系。

由于该统一处理本质上即求 s' 的 $d_1[]$，因此在得到 s' 后，代码同上节计算 $d_1[]$ 的一样。

练习题目

- UVa #11475 "Extend to Palindrome"[1]
- 「国家集训队」最长双回文串[2]

本页面主要译自博文 [3] 与其英文翻译版 Finding all sub-palindromes in $O(N)$[4]。其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0。

参考资料与注释

[1] UVa #11475 "Extend to Palindrome"

[2] 「国家集训队」最长双回文串

[3]

[4] Finding all sub-palindromes in $O(N)$

8.18 回文树

定义

回文树（EER Tree，Palindromic Tree，也被称为回文自动机）是一种可以存储一个串中所有回文子串的高效数据结构。最初由 Mikhail Rubinchik 和 Arseny M. Shur 在 2015 年发表。它的灵感来源于后缀树等字符串后缀数据结构，使用回文树可以简单高效地解决一系列涉及回文串的问题。

结构

回文树大概长这样

和其它自动机类似的，回文树也是由转移边和后缀链接（fail 指针）组成，每个节点都可以代表一个回文子串。

因为回文串长度分为奇数和偶数，我们可以像 manacher 那样加入一个不在字符集中的字符（如'#'）作为分隔符来将所有回文串的长度都变为奇数，但是这样过于麻烦了。有没有更好的办法呢?

答案自然是有。更好的办法就是建两棵树，一棵树中的节点对应的回文子串长度均为奇数，另一棵树中的节点对应的回文子串长度均为偶数。

和其它的自动机一样，一个节点的 fail 指针指向的是这个节点所代表的回文串的最长回文后缀所对应的节点，但是转移边并非代表在原节点代表的回文串后加一个字符，而是表示在原节点代表的回文串前后各加一个相同的字符（不难理解，因为要保证存的是回文串）。

我们还需要在每个节点上维护此节点对应回文子串的长度 len，这个信息保证了我们可以轻松地构造出回文树。

建造

回文树有两个初始状态，分别代表长度为 $-1, 0$ 的回文串。我们可以称它们为奇根，偶根。它们不表示任何实际的字符串，仅作为初始状态存在，这与其他自动机的根节点是异曲同工的。

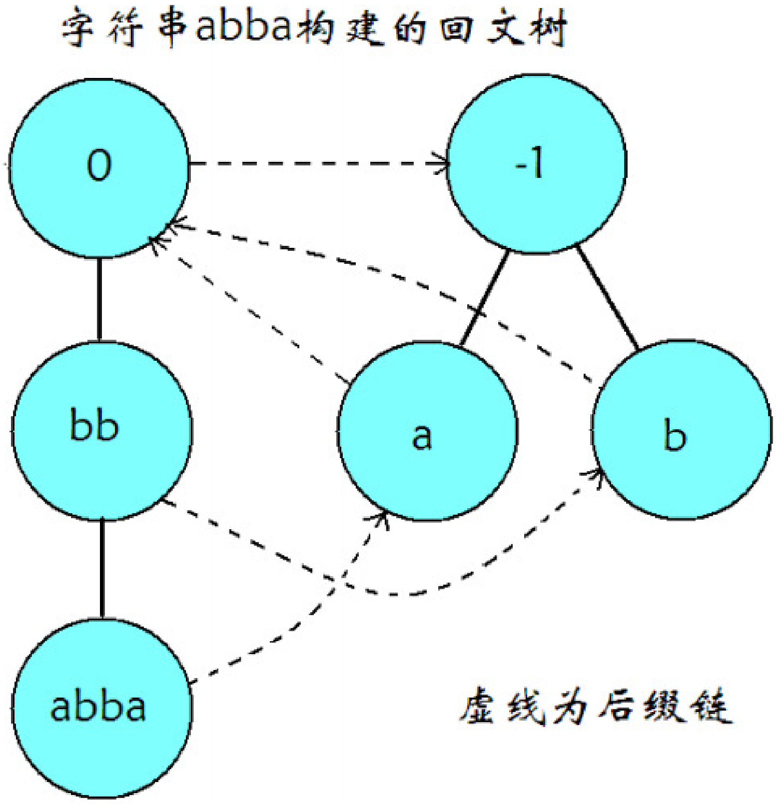


图 8.28

偶根的 fail 指针指向奇根，而我们并不关心奇根的 fail 指针，因为奇根不可能失配（奇根转移出的下一个状态长度为 1，即单个字符。一定是回文子串）

类似后缀自动机，我们增量构造回文树。

考虑构造完前 $p-1$ 个字符的回文树后，向自动机中添加在原串里位置为 p 的字符。

我们从以上一个字符结尾的最长回文子串对应的节点开始，不断沿着 fail 指针走，直到找到一个节点满足 $s_p = s_{p-len-1}$，即满足此节点所对应回文子串的上一个字符与待添加字符相同。

这里贴出论文中的那张图

我们通过跳 fail 指针找到 A 所对应的节点，然后两边添加 `X` 就到了现在的回文串了（即 `XAX`），很显然，这个节点就是以 p 结尾的最长回文子串对应的树上节点。（同时，这个时候长度 -1 节点优势出来了，如果没有 `X` 能匹配条件就是同一个位置的 $s_p = s_p$，就自然得到了代表字符 `X` 的节点。）此时要判断一下：没有这个节点，就需要新建。

然后我们还需要求出新建的节点的 fail 指针。具体方法与上面的过程类似，不断跳转 fail 指针，从 `A` 出发，即可找到 `XAX` 的最长回文后缀 `XBX`，将对应节点设为 fail 指针所指的对象即可。

显然，这个节点是不需新建的，`A` 的前 len_B 位和后 len_B 位相同，都是 `B`，前 len_B 位的两端根据回文串对应关系，都是 `X`，后面被钦定了是 `X`，于是这个节点 `XBX` 肯定已经被包含了。

如果 fail 没匹配到，那么将它连向长度为 0 的那个节点，显然这是可行的（因为这是所有节点的后缀）。

线性状态数证明

定理

对于一个字符串 s，它的本质不同回文子串个数最多只有 $|s|$ 个。

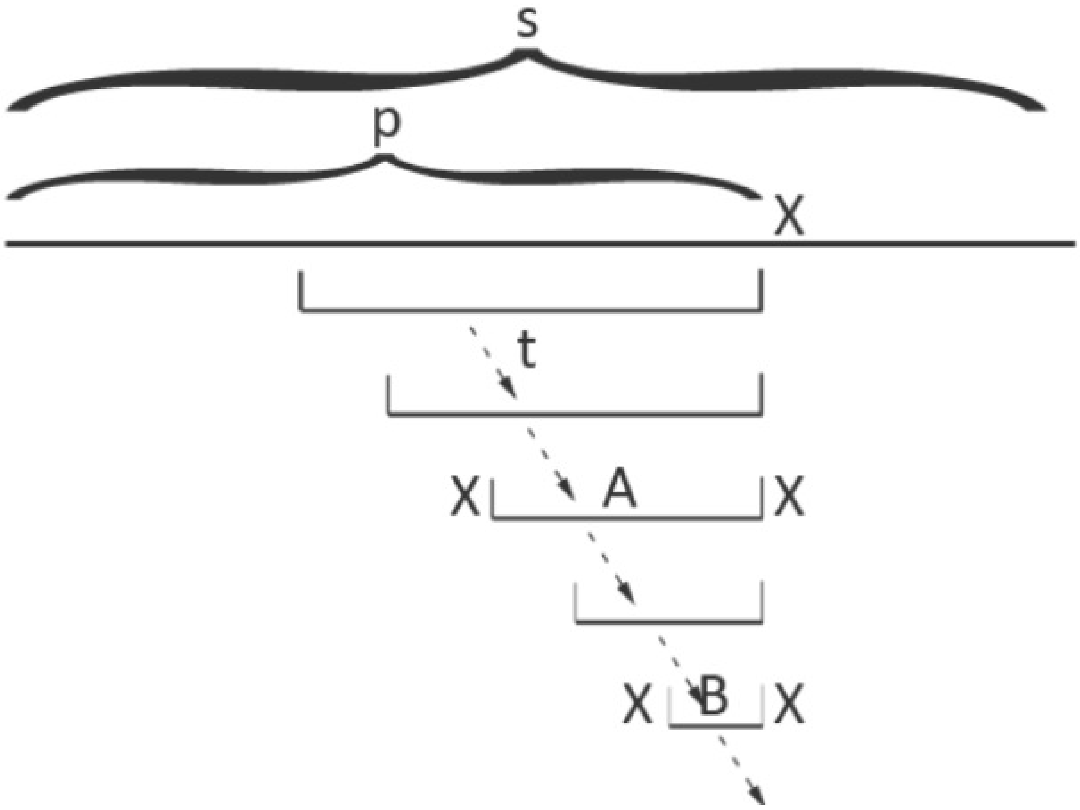


图 8.29

证明

考虑使用数学归纳法。

- 当 $|s|=1$ 时，s 只有一个字符，同时也只有一个子串，并且这个子串是回文的，因此结论成立。
- 当 $|s|>1$ 时，设 $t=sc$，其中 t 表示 s 最后增加一个字符 c 后形成的字符串，假设结论对 s 串成立。考虑以最后一个字符 c 结尾的回文子串，假设它们的左端点由小到大排序为 $l_1, l_2, \dots, l_k$。由于 $t[l_1..|t|]$ 是回文串，因此对于所有位置 $l_1 \le p \le |t|$，有 $t[p..|t|]=t[l_1..l_1+|t|-p]$。所以，对于 $1<i\le k$，$t[l_i..|t|]$ 已经在 $t[1..|t|-1]$ 中出现过。因此，每次增加一个字符，本质不同的回文子串个数最多增加 1 个。

由数学归纳法，可知该定理成立。

因此回文树状态数是 $O(|s|)$ 的。对于每一个状态，它实际只代表一个本质不同的回文子串，即转移到该节点的状态唯一，因此总转移数也是 $O(|s|)$ 的。

正确性证明

以上图为例，增加当前字符 X，由线性状态数的证明，我们只需要找到包含最后一个字符 X 的最长回文后缀，也就是 XAX。继续寻找 XAX 的最长回文后缀 XBX，建立后缀链接。XBX 对应状态已经在回文树中出现，包含最后一个字符的回文后缀就是 XAX，XBX 本身及其对应状态在 fail 树上的所有祖先。

对于 s 回文树的构造，令 $n=|s|$，显然除了跳 fail 指针的其他操作都是 $O(n)$ 的。

加入字符时，在上一次的基础上，每次跳 fail 后对应节点在 fail 树的深度 -1，而连接 fail 后，仅为深度 $+1$（但 fail 为 0 时（即到 -1 才符合），深度相当于在 -1 的基础上 $+2$）。

因为只加入 n 个字符，所以只会加 n 次深度，最多也只会跳 $2n$ 次 fail。

因此，构造 s 的回文树的时间复杂度是 $O(|s|)$。

应用

本质不同回文子串个数

由线性状态数的证明，容易知道一个串的本质不同回文子串个数等于回文树的状态数（排除奇根和偶根两个状态）。

回文子串出现次数

建出回文树，使用类似后缀自动机统计出现次数的方法。

由于回文树的构造过程中，节点本身就是按照拓扑序插入，因此只需要逆序枚举所有状态，将当前状态的出现次数加到其 fail 指针对应状态的出现次数上即可。

例题：「APIO2014」 回文串[1]

定义 s 的一个子串的存在值为这个子串在 s 中出现的次数乘以这个子串的长度。对于给定的字符串 s，求所有回文子串中的最大存在值。

" 参考代码"

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 300000 + 5;

namespace pam {
int sz, tot, last;
int cnt[maxn], ch[maxn][26], len[maxn], fail[maxn];
char s[maxn];

int node(int l) {  // 建立一个新节点，长度为 l
  sz++;
  memset(ch[sz], 0, sizeof(ch[sz]));
  len[sz] = l;
  fail[sz] = cnt[sz] = 0;
  return sz;
}

void clear() {  // 初始化
  sz = -1;
  last = 0;
  s[tot = 0] = '$';
  node(0);
  node(-1);
  fail[0] = 1;
}

int getfail(int x) {  // 找后缀回文
  while (s[tot - len[x] - 1] != s[tot]) x = fail[x];
  return x;
}

void insert(char c) {  // 建树
  s[++tot] = c;
  int now = getfail(last);
  if (!ch[now][c - 'a']) {
```

```
    int x = node(len[now] + 2);
    fail[x] = ch[getfail(fail[now])][c - 'a'];
    ch[now][c - 'a'] = x;
  }
  last = ch[now][c - 'a'];
  cnt[last]++;
}

long long solve() {
  long long ans = 0;
  for (int i = sz; i >= 0; i--) {
    cnt[fail[i]] += cnt[i];
  }
  for (int i = 1; i <= sz; i++) {  // 更新答案
    ans = max(ans, 1ll * len[i] * cnt[i]);
  }
  return ans;
}
}  // namespace pam

char s[maxn];

int main() {
  pam::clear();
  scanf("%s", s + 1);
  for (int i = 1; s[i]; i++) {
    pam::insert(s[i]);
  }
  printf("%lld\n", pam::solve());
  return 0;
}
```

最小回文划分

> 给定一个字符串 $s(1 \le |s| \le 10^5)$，求最小的 k，使得存在 $s_1, s_2, \dots, s_k$，满足 $s_i(1 \le i \le k)$ 均为回文串，且 $s_1, s_2, \dots, s_k$ 依次连接后得到的字符串等于 s。

考虑动态规划，记 $dp[i]$ 表示 s 长度为 i 的前缀的最小划分数，转移只需要枚举以第 i 个字符结尾的所有回文串

$$dp[i] = 1 + \min_{s[j+1..i] \text{ 为回文串}} dp[j]$$

由于一个字符串最多会有 $O(n^2)$ 个回文子串，因此上述算法的时间复杂度为 $O(n^2)$，无法接受，为了优化转移过程，下面给出一些引理。

记字符串 s 长度为 i 的前缀为 $pre(s,i)$，长度为 i 的后缀为 $suf(s,i)$。

周期：若 $0 < p \le |s|$，$\forall 1 \le i \le |s| - p, s[i] = s[i+p]$，就称 p 是 s 的周期。

border：若 $0 \le r < |s|$，$pre(s,r) = suf(s,r)$，就称 $pre(s,r)$ 是 s 的 border。

周期和 border 的关系：t 是 s 的 border，当且仅当 $|s| - |t|$ 是 s 的周期。

" 证明"

若 t 是 s 的 border，那么 $pre(s,|t|) = suf(s,|t|)$，因此 $\forall 1 \le i \le |t|, s[i] = s[|s| - |t| + i]$，所以 $|s| - |t|$ 就是 s 的周期。

若 $|s|-|t|$ 为 s 周期，则 $\forall 1 \le i \le |s|-(|s|-|t|)=|t|, s[i]=s[|s|-|t|+i]$，因此 $pre(s,|t|)=suf(s,|t|)$，所以 t 是 s 的 border。

引理一

t 是回文串 s 的后缀，t 是 s 的 border 当且仅当 t 是回文串。

" 证明"

对于 $1 \le i \le |t|$，由 s 和 t 为回文串，因此有 $s[i]=s[|s|-i+1]=s[|s|-|t|+i]$，所以 t 是 s 的 border。

对于 $1 \le i \le |t|$，由 t 是 s 的 border，有 $s[i]=s[|s|-|t|+i]$，由 s 是回文串，有 $s[i]=s[|s|-i+1]$，因此 $s[|s|-i+1]=s[|s|-|t|+i]$，所以 t 是回文串。

下图中，相同颜色的位置表示字符对应相同。

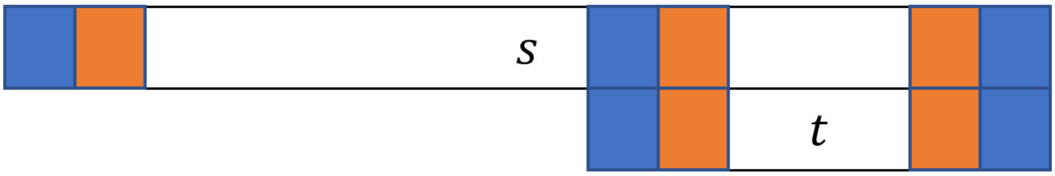


图 8.30

引理二

t 是串 s 的 border ($|s| \le 2|t|$)，s 是回文串当且仅当 t 是回文串。

" 证明"

若 s 是回文串，由引理 1，t 也是回文串。

若 t 是回文串，由 t 是 s 的 border，因此 $\forall 1 \le i \le |t|, s[i]=s[|s|-|t|+i]=s[|s|-i+1]$，因为 $|s| \le 2|t|$，所以 s 也是回文串。

引理三

t 是回文串 s 的 border，则 $|s|-|t|$ 是 s 的周期，$|s|-|t|$ 为 s 的最小周期，当且仅当 t 是 s 的最长回文真后缀。

引理四

x 是一个回文串，y 是 x 的最长回文真后缀，z 是 y 的最长回文真后缀。令 u,v 分别为满足 $x=uy, y=vz$ 的字符串，则有下面三条性质

1. $|u| \ge |v|$；
2. 如果 $|u|>|v|$，那么 $|u|>|z|$；
3. 如果 $|u|=|v|$，那么 $u=v$。

" 证明"

1. 由引理 3 的推论，$|u|=|x|-|y|$ 是 x 的最小周期，$|v|=|y|-|z|$ 是 y 的最小周期。考虑反证法，假设 $|u|<|v|$，因为 y 是 x 的后缀，所以 u 既是 x 的周期，也是 y 的周期，而 $|v|$ 是 y 的最小周期，矛盾。所以 $|u| \ge |v|$。
2. 因为 y 是 x 的 border，所以 v 是 x 的前缀，设字符串 w，满足 $x=vw$（如下图所示），其中 z 是 w 的

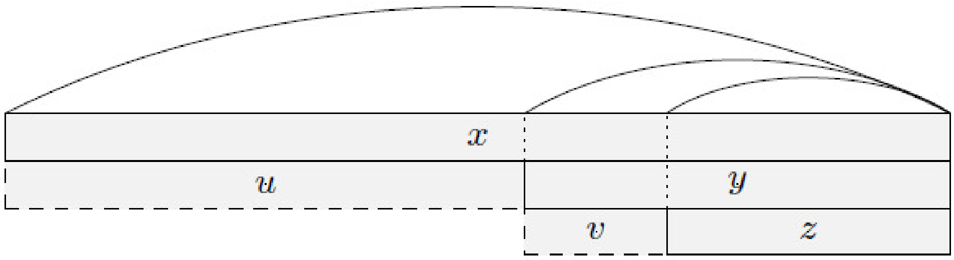


图 8.31

border。考虑反证法，假设 $|u| \le |z|$，那么 $|zu| \le 2|z|$，所以由引理 2，w 是回文串，由引理 1，w 是 x 的 border，又因为 $|u| > |v|$，所以 $|w| > |y|$，矛盾。所以 $|u| > |z|$。

3. u, v 都是 x 的前缀，$|u| = |v|$，所以 $u = v$。

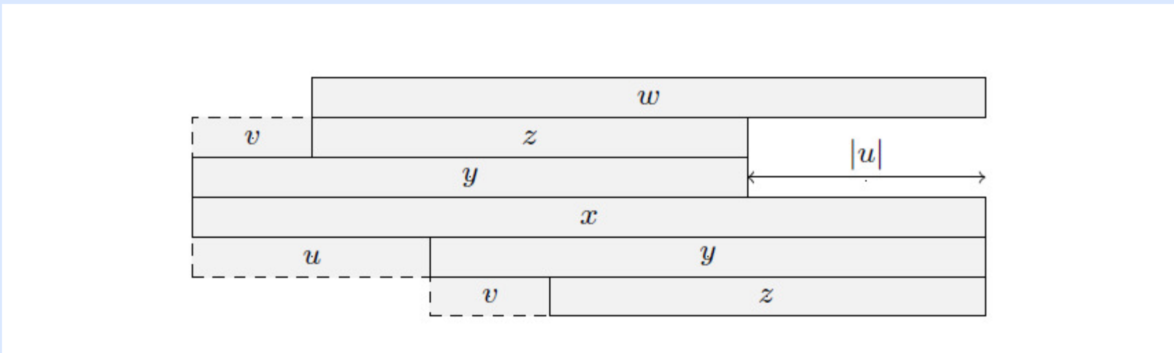


图 8.32

推论

s 的所有回文后缀按照长度排序后，可以划分成 $\log |s|$ 段等差数列。

" 证明"

设 s 的所有回文后缀长度从小到大排序为 $l_1, l_2, \dots, l_k$。对于任意 $2 \le i \le k-1$，若 $l_i - l_{i-1} = l_{i+1} - l_i$，则 l_{i-1}, l_i, l_{i+1} 构成一个等差数列。否则 $l_i - l_{i-1} \ne l_{i+1} - l_i$，由引理 4，有 $l_{i+1} - l_i > l_i - l_{i-1}$，且 $l_{i+1} - l_i > l_{i-1}$，$l_{i+1} > 2l_{i-1}$。因此，若相邻两对回文后缀的长度之差发生变化，那么这个最大长度一定会相对于最小长度翻一倍。显然，长度翻倍最多只会发生 $O(\log |s|)$ 次，也就是 s 的回文后缀长度可以划分成 $\log |s|$ 段等差数列。

该推论也可以通过使用弱周期引理，对 s 的最长回文后缀的所有 border 按照长度 x 分类，$x \in [2^0, 2^1), [2^1, 2^2), \dots, [2^k, n)$，考虑这 $\log |s|$ 组内每组的最长 border 进行证明。详细证明可以参考金策的《字符串算法选讲》和陈孙立的 2019 年 IOI 国家候选队论文《子串周期查询问题的相关算法及其应用》。

有了这个结论后，我们现在可以考虑如何优化 dp 的转移。

优化

回文树上的每个节点 u 需要多维护两个信息，$diff[u]$ 和 $slink[u]$。$diff[u]$ 表示节点 u 和 $fail[u]$ 所代表的回文串的长度差，即 $len[u] - len[fail[u]]$。$slink[u]$ 表示 u 一直沿着 fail 向上跳到第一个节点 v，使得 $diff[v] \ne diff[u]$，也就是 u 所在等差数列中长度最小的那个节点。

根据上面证明的结论，如果使用 $slink$ 指针向上跳的话，每向后填加一个字符，只需要向上跳 $O(\log |s|)$ 次。因此，可以考虑将一个等差数列表示的所有回文串的 dp 值之和（在原问题中指 min），记录到最长的那一个回文串对应节点上。

$g[v]$ 表示 v 所在等差数列的 dp 值之和，且 v 是这个等差数列中长度最长的节点，则 $g[v] = \sum_{slink[x]=slink[v]} dp[i-$

$len[x]]$，这里 i 是当前枚举到的下标。

下面我们考虑如何更新 g 数组和 dp 数组。以下图为例，假设当前枚举到第 i 个字符，回文树上对应节点为 x。$g[x]$ 为橙色三个位置的 dp 值之和（最短的回文串 $slink[x]$ 算在下一个等差数列中）。$fail[x]$ 上一次出现位置是 $i - diff[x]$（在 $i - diff[x]$ 处结束），$g[fail[x]]$ 包含的 dp 值是蓝色位置。因此，$g[x]$ 实际上等于 $g[fail[x]]$ 和多出来一个位置的 dp 值之和，多出来的位置是 $i - (len[slink[x]] + diff[x])$。最后再用 $g[x]$ 去更新 $dp[i]$，这部分等差数列的贡献就计算完毕了，不断跳 $slink[x]$，重复这个过程即可。具体实现方式可参考例题代码。

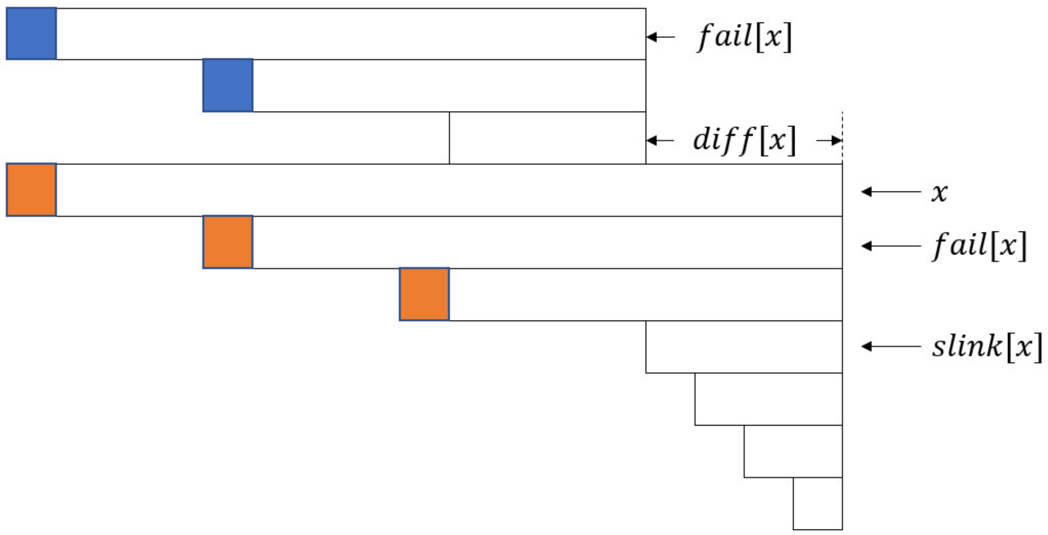


图 8.33

最后，上述做法的正确性依赖于：如果 x 和 $fail[x]$ 属于同一个等差数列，那么 $fail[x]$ 上一次出现位置是 $i - diff[x]$。

" 证明"

根据引理 1，$fail[x]$ 是 x 的 border，因此其在 $i - diff[x]$ 处出现。

假设 $fail[x]$ 在 $(i - diff[x], i)$ 中的 j 位置出现。由于 x 和 $fail[x]$ 属于同一个等差数列，因此 $2|fail[x]| \geq x$。多余的 $fail[x]$ 和 $i - diff[x]$ 处的 $fail[x]$ 有交集，记交集为 w，设串 u 满足 $uw = fail[x]$。用类似引理 1 的方式可以证明，w 是回文串，而 x 的前缀 $s[i - len[x] + 1..j] = uwu$ 也是回文串，这与 $fail[x]$ 是 x 的最长回文前缀（后缀）矛盾。

例题：Codeforces 932G Palindrome Partition[2]

给定一个字符串 s，要求将 s 划分为 $t_1, t_2, \dots, t_k$，其中 k 是偶数，且 $t_i = t_{k-i+1}$，求这样的划分方案数。

" 题解"

构造字符串 $t = s[0]s[n-1]s[1]s[n-2]s[2]s[n-3]\dots s[n/2-1]s[n/2]$，问题等价于求 t 的偶回文划分方案数，把上面的转移方程改成求和形式并且只在偶数位置更新 dp 数组即可。时间复杂度 $O(n\log n)$，空间复杂度 $O(n)$。

" 参考代码"

```
#include <bits/stdc++.h>
using namespace std;
```

```
typedef long long ll;
const int mod = 1e9 + 7;
const int maxn = 1000000 + 5;

int add(int x, int y) {
  x += y;
  return x >= mod ? x -= mod : x;
}

namespace pam {
int sz, tot, last;
int ch[maxn][26], len[maxn], fail[maxn];
int cnt[maxn], dep[maxn], dif[maxn], slink[maxn];
char s[maxn];

int node(int l) {  // 建立一个长度为 l 的新节点
  sz++;
  memset(ch[sz], 0, sizeof(ch[sz]));
  len[sz] = l;
  fail[sz] = 0;
  cnt[sz] = 0;
  dep[sz] = 0;
  return sz;
}

void clear() {  // 初始化
  sz = -1;
  last = 0;
  s[tot = 0] = '$';
  node(0);
  node(-1);
  fail[0] = 1;
}

int getfail(int x) {  // 找到后缀回文
  while (s[tot - len[x] - 1] != s[tot]) x = fail[x];
  return x;
}

void insert(char c) {  // 建树
  s[++tot] = c;
  int now = getfail(last);
  if (!ch[now][c - 'a']) {
    int x = node(len[now] + 2);
    fail[x] = ch[getfail(fail[now])][c - 'a'];
    dep[x] = dep[fail[x]] + 1;
    ch[now][c - 'a'] = x;
    dif[x] = len[x] - len[fail[x]];
    if (dif[x] == dif[fail[x]])
      slink[x] = slink[fail[x]];
    else
      slink[x] = fail[x];
  }
  last = ch[now][c - 'a'];
```

```cpp
    cnt[last]++;
  }
}  // namespace pam

using pam::dif;
using pam::fail;
using pam::len;
using pam::slink;
int n, dp[maxn], g[maxn];
char s[maxn], t[maxn];

int main() {
  pam::clear();
  scanf("%s", s + 1);
  n = strlen(s + 1);
  for (int i = 1, j = 0; i <= n; i++) t[++j] = s[i], t[++j] = s[n - i + 1];
  dp[0] = 1;
  for (int i = 1; i <= n; i++) {
    pam::insert(t[i]);
    for (int x = pam::last; x > 1; x = slink[x]) {
      g[x] = dp[i - len[slink[x]] - dif[x]];
      if (dif[x] == dif[fail[x]]) g[x] = add(g[x], g[fail[x]]);
      if (i % 2 == 0) dp[i] = add(dp[i], g[x]);  // 在偶数位置更新 dp 数组
    }
  }
  printf("%d", dp[n]);
  return 0;
}
```

例题

- 最长双回文串[3]
- 拉拉队排练[4]
- 「SHOI2011」 双倍回文[5]
- HDU 5421 Victor and String[6]
- CodeChef Palindromeness[7]

相关资料

- EERTREE: An Efficient Data Structure for Processing Palindromes in Strings[8]
- Palindromic tree[9]
- 2017 年 IOI 国家候选队论文集回文树及其应用翁文涛
- 2019 年 IOI 国家候选队论文集子串周期查询问题的相关算法及其应用陈孙立
- 字符串算法选讲金策
- A bit more about palindromes[10]
- A Subquadratic Algorithm for Minimum Palindromic Factorization[11]

参考资料与注释

[1] 「APIO2014」回文串

[2] Codeforces 932G Palindrome Partition

[3] 最长双回文串

[4] 拉拉队排练

[5] 「SHOI2011」双倍回文

[6] HDU 5421 Victor and String

[7] CodeChef Palindromeness

[8] EERTREE: An Efficient Data Structure for Processing Palindromes in Strings

[9] Palindromic tree

[10] A bit more about palindromes

[11] A Subquadratic Algorithm for Minimum Palindromic Factorization

8.19　序列自动机

在阅读本文之前，请先阅读 自动机。

定义

序列自动机是接受且仅接受一个字符串的子序列的自动机。

本文中用 s 代指这个字符串。

状态

若 s 包含 n 个字符，那么序列自动机包含 $n+1$ 个状态。

令 t 是 s 的一个子序列，那么 $\delta(start, t)$ 是 t 在 s 中第一次出现时末端的位置。

也就是说，一个状态 i 表示前缀 $s[1..i]$ 的子序列与前缀 $s[1..i-1]$ 的子序列的差集。

序列自动机上的所有状态都是接受状态。

转移

由状态定义可以得到，$\delta(u,c) = \min\{i|i > u, s[i] = c\}$，也就是字符 c 下一次出现的位置。

为什么是「下一次」出现的位置呢？因为若 $i > j$，后缀 $s[i..|s|]$ 的子序列是后缀 $s[j..|s|]$ 的子序列的子集，一定是选尽量靠前的最优。

实现

从后向前扫描，过程中维护每个字符最前的出现位置：

Input. A string S
Output. The state transition of the sequence automaton of S
Method.
for $c \in \Sigma$
 $next[c] \leftarrow null$
for $i \leftarrow |S|$ **downto** 1
 $next[S[i]] \leftarrow i$
 for $c \in \Sigma$
  $\delta(i-1, c) \leftarrow next[c]$
return δ

这样构建的复杂度是 $O(n|\Sigma|)$。

例题

"「HEOI2015」 最短不公共子串[1]"

给你两个由小写英文字母组成的串 A 和 B，求：

1. A 的一个最短的子串，它不是 B 的子串；
2. A 的一个最短的子串，它不是 B 的子序列；
3. A 的一个最短的子序列，它不是 B 的子串；
4. A 的一个最短的子序列，它不是 B 的子序列。

$1 \le |A|, |B| \le 2000$。

" 题解"

这题的 (1) 和 (3) 两问需要后缀自动机，而且做法类似，在这里只讲解 (2) 和 (4) 两问。

(2) 比较简单，枚举 A 的子串输入进 B 的序列自动机，若不接受则计入答案。

(4) 需要 DP。令 $f(i,j)$ 表示在 A 的序列自动机中处于状态 i，在 B 的序列自动机中处于状态 j，需要再添加多少个字符能够不是公共子序列。

$f(i, null) = 0$

$f(i,j) = \min_{\delta_A(i,c) \neq null} f(\delta_A(i,c), \delta_B(j,c)) + 1$

" 参考代码"

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <iostream>

using namespace std;

const int N = 2005;

char s[N], t[N];
int na[N][26], nb[N][26], nxt[26];
```

```cpp
int n, m, a[N], b[N], tot = 1, p = 1, f[N][N << 1];

struct SAM {
  int par, ch[26], len;
} sam[N << 1];

void insert(int x) {
  int np = ++tot;  // 新节点
  sam[np].len = sam[p].len + 1;
  while (p && !sam[p].ch[x]) {
    sam[p].ch[x] = np;
    p = sam[p].par;
  }
  if (p == 0)
    sam[np].par = 1;
  else {
    int q = sam[p].ch[x];
    if (sam[q].len == sam[p].len + 1)
      sam[np].par = q;
    else {
      int nq = ++tot;
      sam[nq].len = sam[p].len + 1;
      memcpy(sam[nq].ch, sam[q].ch, sizeof(sam[q].ch));
      sam[nq].par = sam[q].par;
      sam[q].par = sam[np].par = nq;
      while (p && sam[p].ch[x] == q) {
        sam[p].ch[x] = nq;
        p = sam[p].par;
      }
    }
  }
  p = np;
}

int main() {
  scanf("%s%s", s + 1, t + 1);

  n = strlen(s + 1);
  m = strlen(t + 1);

  for (int i = 1; i <= n; ++i) a[i] = s[i] - 'a';
  for (int i = 1; i <= m; ++i) b[i] = t[i] - 'a';

  for (int i = 1; i <= m; ++i) insert(b[i]);

  // nxt[S[i]]<-i
  for (int i = 0; i < 26; ++i) nxt[i] = n + 1;
  for (int i = n; i >= 0; --i) {
    memcpy(na[i], nxt, sizeof(nxt));
    nxt[a[i]] = i;
  }

  for (int i = 0; i < 26; ++i) nxt[i] = m + 1;
  for (int i = m; i >= 0; --i) {
```

```
    memcpy(nb[i], nxt, sizeof(nxt));
    nxt[b[i]] = i;
  }

  // 四种情况计算答案
  //  1
  int ans = N;
  for (int l = 1; l <= n; ++l) {
    for (int r = l, u = 1; r <= n; ++r) {
      u = sam[u].ch[a[r]];
      if (!u) {
        ans = min(ans, r - l + 1);
        break;
      }
    }
  }

  printf("%d\n", ans == N ? -1 : ans);

  // 2
  ans = N;

  for (int l = 1; l <= n; ++l) {
    for (int r = l, u = 0; r <= n; ++r) {
      u = nb[u][a[r]];
      if (u == m + 1) {
        ans = min(ans, r - l + 1);
        break;
      }
    }
  }

  printf("%d\n", ans == N ? -1 : ans);

  // 3
  for (int i = n; i >= 0; --i) {
    for (int j = 1; j <= tot; ++j) {
      f[i][j] = N;
      for (int c = 0; c < 26; ++c) {
        int u = na[i][c];
        int v = sam[j].ch[c];
        if (u <= n) f[i][j] = min(f[i][j], f[u][v] + 1);
      }
    }
  }

  printf("%d\n", f[0][1] == N ? -1 : f[0][1]);

  // 4
  memset(f, 0, sizeof(f));

  for (int i = n; i >= 0; --i) {
    for (int j = 0; j <= m; ++j) {
      f[i][j] = N;
```

```
      for (int c = 0; c < 26; ++c) {
        int u = na[i][c];
        int v = nb[j][c];
        if (u <= n) f[i][j] = min(f[i][j], f[u][v] + 1);
      }
    }
  }

  printf("%d\n", f[0][0] == N ? -1 : f[0][0]);

  return 0;
}
```

参考资料与注释

[1] 「HEOI2015」最短不公共子串

8.20 最小表示法

定义

最小表示法是用于解决字符串最小表示问题的方法。

字符串的最小表示

循环同构

当字符串 S 中可以选定一个位置 i 满足

$$S[i \cdots n] + S[1 \cdots i-1] = T$$

则称 S 与 T 循环同构

最小表示

字符串 S 的最小表示为与 S 循环同构的所有字符串中字典序最小的字符串

simple 的暴力

我们每次比较 i 和 j 开始的循环同构，把当前比较到的位置记作 k，每次遇到不一样的字符时便把大的跳过，最后剩下的就是最优解。

实现

```
int k = 0, i = 0, j = 1;
while (k < n && i < n && j < n) {
  if (sec[(i + k) % n] == sec[(j + k) % n]) {
    ++k;
  } else {
    if (sec[(i + k) % n] > sec[(j + k) % n])
```

```
      ++i;
    else
      ++j;
    k = 0;
    if (i == j) i++;
  }
}
i = min(i, j);
```

```
k, i, j = 0, 0, 1
while k < n and i < n and j < n:
    if sec[(i + k) % n] == sec[(j + k) % n]:
        k += 1
    else:
        if sec[(i + k) % n] > sec[(j + k) % n]:
            i += 1
        else:
            j += 1
        k = 0
        if i == j:
            i += 1
i = min(i, j)
```

解释

该实现方法随机数据下表现良好，但是可以构造特殊数据卡掉。

例如：对于 aaa⋯aab, 不难发现这个算法的复杂度退化为 $O(n^2)$。

我们发现，当字符串中出现多个连续重复子串时，此算法效率降低，我们考虑优化这个过程。

最小表示法

算法核心

考虑对于一对字符串 A, B, 它们在原字符串 S 中的起始位置分别为 i, j, 且它们的前 k 个字符均相同，即

$$S[i \cdots i+k-1] = S[j \cdots j+k-1]$$

不妨先考虑 $S[i+k] > S[j+k]$ 的情况，我们发现起始位置下标 l 满足 $i \le l \le i+k$ 的字符串均不能成为答案。因为对于任意一个字符串 S_{i+p}（表示以 $i+p$ 为起始位置的字符串，$p \in [0, k]$）一定存在字符串 S_{j+p} 比它更优。

所以我们比较时可以跳过下标 $l \in [i, i+k]$, 直接比较 S_{i+k+1}

这样，我们就完成了对于上文暴力的优化。

时间复杂度

$O(n)$

过程

1. 初始化指针 i 为 0，j 为 1；初始化匹配长度 k 为 0
2. 比较第 k 位的大小，根据比较结果跳转相应指针。若跳转后两个指针相同，则随意选一个加一以保证比较的两个字符串不同

3. 重复上述过程，直到比较结束
4. 答案为 i, j 中较小的一个

实现

```cpp
int k = 0, i = 0, j = 1;
while (k < n && i < n && j < n) {
  if (sec[(i + k) % n] == sec[(j + k) % n]) {
    k++;
  } else {
    sec[(i + k) % n] > sec[(j + k) % n] ? i = i + k + 1 : j = j + k + 1;
    if (i == j) i++;
    k = 0;
  }
}
i = min(i, j);
```

```python
k, i, j = 0, 0, 1
while k < n and i < n and j < n:
    if sec[(i + k) % n] == sec[(j + k) % n]:
        k += 1
    else:
        if sec[(i + k) % n] > sec[(j + k) % n]:
            i = i + k + 1
        else:
            j = j + k + 1
        if i == j:
            i += 1
        k = 0
i = min(i, j)
```

8.21 Lyndon 分解

Authors: sshwy, StudyingFather, orzAtalod

定义

首先我们介绍 Lyndon 分解的概念。

Lyndon 串：对于字符串 s，如果 s 的字典序严格小于 s 的所有后缀的字典序，我们称 s 是简单串，或者 **Lyndon 串**。举一些例子，a,b,ab,aab,abb,ababb,abcd 都是 Lyndon 串。当且仅当 s 的字典序严格小于它的所有非平凡的（非平凡：非空且不同于自身）循环同构串时，s 才是 Lyndon 串。

Lyndon 分解：串 s 的 Lyndon 分解记为 $s = w_1 w_2 \cdots w_k$，其中所有 w_i 为简单串，并且他们的字典序按照非严格单减排序，即 $w_1 \ge w_2 \ge \cdots \ge w_k$。可以发现，这样的分解存在且唯一。

Duval 算法

解释

Duval 可以在 $O(n)$ 的时间内求出一个串的 Lyndon 分解。

首先我们介绍另外一个概念：如果一个字符串 t 能够分解为 $t = ww \cdots \overline{w}$ 的形式，其中 w 是一个 Lyndon 串，而 $\overline{w}$ 是 w 的前缀（$\overline{w}$ 可能是空串），那么称 t 是近似简单串（pre-simple），或者近似 Lyndon 串。一个 Lyndon 串也是近似 Lyndon 串。

Duval 算法运用了贪心的思想。算法过程中我们把串 s 分成三个部分 $s = s_1 s_2 s_3$，其中 s_1 是一个 Lyndon 串，它的 Lyndon 分解已经记录；s_2 是一个近似 Lyndon 串；s_3 是未处理的部分。

过程

整体描述一下，该算法每一次尝试将 s_3 的首字符添加到 s_2 的末尾。如果 s_2 不再是近似 Lyndon 串，那么我们就可以将 s_2 截出一部分前缀（即 Lyndon 分解）接在 s_1 末尾。

我们来更详细地解释一下算法的过程。定义一个指针 i 指向 s_2 的首字符，则 i 从 1 遍历到 n（字符串长度）。在循环的过程中我们定义另一个指针 j 指向 s_3 的首字符，指针 k 指向 s_2 中我们当前考虑的字符（意义是 j 在 s_2 的上一个循环节中对应的字符）。我们的目标是将 $s[j]$ 添加到 s_2 的末尾，这就需要将 $s[j]$ 与 $s[k]$ 做比较：

1. 如果 $s[j] = s[k]$，则将 $s[j]$ 添加到 s_2 末尾不会影响它的近似简单性。于是我们只需要让指针 j, k 自增（移向下一位）即可。
2. 如果 $s[j] > s[k]$，那么 $s_2 s[j]$ 就变成了一个 Lyndon 串，于是我们将指针 j 自增，而让 k 指向 s_2 的首字符，这样 s_2 就变成了一个循环次数为 1 的新 Lyndon 串了。
3. 如果 $s[j] < s[k]$，则 $s_2 s[j]$ 就不是一个近似简单串了，那么我们就要把 s_2 分解出它的一个 Lyndon 子串，这个 Lyndon 子串的长度将是 $j - k$，即它的一个循环节。然后把 s_2 变成分解完以后剩下的部分，继续循环下去（注意，这个情况下我们没有改变指针 j, k），直到循环节被截完。对于剩余部分，我们只需要将进度「回退」到剩余部分的开头即可。

实现

下面的代码返回串 s 的 Lyndon 分解方案。

```cpp
// duval_algorithm
vector<string> duval(string const& s) {
  int n = s.size(), i = 0;
  vector<string> factorization;
  while (i < n) {
    int j = i + 1, k = i;
    while (j < n && s[k] <= s[j]) {
      if (s[k] < s[j])
        k = i;
      else
        k++;
      j++;
    }
    while (i <= k) {
      factorization.push_back(s.substr(i, j - k));
      i += j - k;
    }
  }
  return factorization;
}
```

```python
# duval_algorithm
def duval(s):
    n, i = len(s), 0
    factorization = []
```

```
    while i < n:
        j, k = i + 1, i
        while j < n and s[k] <= s[j]:
            if s[k] < s[j]:
                k = i
            else:
                k += 1
            j += 1
        while i <= k:
            factorization.append(s[i : i + j - k])
            i += j - k
    return factorization
```

复杂度分析

接下来我们证明一下这个算法的复杂度。

外层的循环次数不超过 n，因为每一次 i 都会增加。第二个内层循环也是 $O(n)$ 的，因为它只记录 Lyndon 分解的方案。接下来我们分析一下内层循环。很容易发现，每一次在外层循环中找到的 Lyndon 串是比我们所比较过的剩余的串要长的，因此剩余的串的长度和要小于 n，于是我们最多在内层循环 $O(n)$ 次。事实上循环的总次数不超过 $4n-3$，时间复杂度为 $O(n)$。

最小表示法（Finding the smallest cyclic shift）

对于长度为 n 的串 s，我们可以通过上述算法寻找该串的最小表示法。

我们构建串 ss 的 Lyndon 分解，然后寻找这个分解中的一个 Lyndon 串 t，使得它的起点小于 n 且终点大于等于 n。可以很容易地使用 Lyndon 分解的性质证明，子串 t 的首字符就是 s 的最小表示法的首字符，即我们沿着 t 的开头往后 n 个字符组成的串就是 s 的最小表示法。

于是我们在分解的过程中记录每一次的近似 Lyndon 串的开头即可。

```
// smallest_cyclic_string
string min_cyclic_string(string s) {
  s += s;
  int n = s.size();
  int i = 0, ans = 0;
  while (i < n / 2) {
    ans = i;
    int j = i + 1, k = i;
    while (j < n && s[k] <= s[j]) {
      if (s[k] < s[j])
        k = i;
      else
        k++;
      j++;
    }
    while (i <= k) i += j - k;
  }
  return s.substr(ans, n / 2);
}
```

```
# smallest_cyclic_string
def min_cyclic_string(s):
    s += s
    n = len(s)
```

```
    i, ans = 0, 0
    while i < n / 2:
        ans = i
        j, k = i + 1, i
        while j < n and s[k] <= s[j]:
            if s[k] < s[j]:
                k = i
            else:
                k += 1
            j += 1
        while i <= k:
            i += j - k
    return s[ans : ans + n / 2]
```

习题

- UVa #719 - Glass Beads[1]

  本页面主要译自博文 . . [2] 与其英文翻译版 Lyndon factorization[3]。其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0。

参考资料与注释

[1] UVa #719 - Glass Beads

[2] . .

[3] Lyndon factorization

8.22 Main–Lorentz 算法

重串

定义

给定一个长度为 n 的字符串 s。

我们将一个字符串连续写两遍所产生的新字符串称为**重串 (tandem repetition)**。下文中，为了表述精准，我们将被重复的这个字符串称为原串。换言之，一个重串等价于一对下标 (i,j)，其使得 $s[i \dots j]$ 是两个相同字符串拼接而成的。

你的目标是找出给定的字符串 s 中所有的重串。或者，解决一个较为简单的问题：找到字符串 s 中任意重串或者最长的一个重串。

下文的算法由 Michael Main 和 Richard J. Lorentz 在 1982 年提出。

声明：

下文所有的字符串下标从 0 开始。

下文中，记 $\overline{s}$ 为 s 的反串。如 $\overline{\texttt{abc}} = \texttt{cba}$。

解释

考虑字符串 `acababaee`，这个字符串包括三个重串，分别是：

- $s[2 \dots 5] =$ abab
- $s[3 \dots 6] =$ baba
- $s[7 \dots 8] =$ ee

下面是另一个例子，考虑字符串 abaaba，这个字符串只有两个重串：

- $s[0 \dots 5] =$ abaaba
- $s[2 \dots 3] =$ aa

重串的个数

一个长度为 n 的字符串可能有高达 $O(n^2)$ 个重串，一个显然的例子就是 n 个字母全部相同的字符串，这种情况下，只要其子串长度为偶数，这个子串就是重串。多数情况下，一个周期比较小的周期字符串会有很多重串。

但这并不影响我们在 $O(n \log n)$ 的时间内计算出重串数量。因为这个算法通过某种压缩形式来表达一个重串，使得我们可以将多个重串压缩为一个。

这里有一些关于重串数量的有趣结论：

- 如果一个重串的原串不是重串，则我们称这个重串为**本原重串 (primitive repetition)**。可以证明，本原重串最多有 $O(n \log n)$ 个。
- 如果我们把一个重串用 Crochemore 三元组 (i, p, r) 进行压缩，其中 i 是重串的起始位置，p 是该重串某个循环节的长度（注意不是原串长度！），r 为这个循环节重复的次数。则某字符串的所有重串可以被 $O(n \log n)$ 个 Crochemore 三元组表示。
- Fibonacci 字符串定义如下：

$$t_0 = a, \tag{8.1}$$
$$t_1 = b, \tag{8.2}$$
$$t_i = t_{i-1} + t_{i-2}, \tag{8.3}$$

可以发现 Fibonacci 字符串具有高度的周期性。对于长度为 f_i 的 Fibonacci 字符串 t_i，即使用 Crochemore 三元组压缩，也有 $O(f_i \log f_i)$ 个三元组。其本原重串的数量也有 $O(f_i \log f_i)$ 个。

Main–Lorentz 算法

解释

Main–Lorentz 算法的核心思想是**分治**。

这个算法将字符串分为左、右两部分，首先计算完全处于字符串左部（或右部）的重串数量，然后计算起始位置在左部，终止在右部的重串数量。（下文中，我们将这种重串称为**交叉重串**）

计算交叉重串的数量是 Main–Lorentz 算法的关键点，我们将在下文详细探讨。

过程

寻找交叉重串

我们记某字符串的左部为 u，右部为 v。则 $s = u + v$，且 u, v 的长度大约等于 s 长度的一半。

对于任意一个重串，我们考虑其中间字符。此处我们将一个重串右半边的第一个字符称为其中间字符，换言之，若 $s[i \dots j]$ 为重串，则其中间字符为 $s[(i+j+1)/2]$。如果一个重串的中间字符在 u 中，则称这个重串**左偏 (left)**，反之则称其**右偏 (right)**。

接下来，我们将会探讨如何找到所有的左偏重串。

我们记一个左边重串的长度为 $2l$。考虑该重串第一个落入 v 的字符（即 $s[|u|]$），这个字符一定与 u 中的某个字符 $u[cntr]$ 一致。

我们考虑固定 $cntr$，并找到所有符合条件的重串。举个例子：对于字符串 $\texttt{c}\ \underset{cntr}{\texttt{a}}\ \texttt{c}\ |\ \texttt{a}\ \texttt{d}\ \texttt{a}$（这个 | 是用于分辨左右两部分的），固定 $cntr = 1$，则我们可以发现重串 `caca` 符合要求。

显然，我们一旦固定了 $cntr$，那我们同时也固定了 l 的取值。我们一旦知道如何找到所有重串，我们就可以从 0 到 $|u| - 1$ 枚举 $cntr$ 的取值，然后找到所有符合条件的重串。

左偏重串的判定

即使固定 $cntr$ 后，仍然可能会有多个符合条件的重串，我们怎么找到所有符合条件的重串呢？

我们再来举一个例子，对于字符串 `abcabcac` 中的重串 $\overset{l_1}{\texttt{a}}\overbrace{\underset{cntr}{\texttt{b}}\ \texttt{c}}^{l_2}\overset{l_1}{\texttt{a}}\ |\ \overbrace{\texttt{b}\,\texttt{c}}^{l_2}$，我们记 l_1 为该重串的首字符到 $s[cntr-1]$ 所组成的子串的长度，记 l_2 为 $s[cntr]$ 到该重串左边原串的末字符所组成的子串的长度。

于是，我们可以给出某个长度为 $2l = 2(l_1 + l_2) = 2(|u| - cntr)$ 的子串是重串的**充分必要条件**：

记 k_1 为满足 $u[cntr - k_1 \ldots cntr - 1] = u[|u| - k_1 \ldots |u| - 1]$ 的最大整数，记 k_2 为满足 $u[cntr \ldots cntr + k_2 - 1] = v[0 \ldots k_2 - 1]$ 的最大整数。则对于任意满足 $l_1 \le k_1$，$l_2 \le k_2$ 的二元组 (l_1, l_2)，我们都能恰好找到一个与之对应的重串。

总结一下，即有：

- 固定一个 $cntr$。
- 那么我们此时要找的重串长度均为 $2l = 2(|u| - cntr)$。此时可能仍有多个符合条件的重串，取决于 l_1 与 l_2 的取值。
- 计算上文提到的 k_1，k_2。
- 则所有符合条件的重串符合条件：

$$l_1 + l_2 = l = |u| - cntr \tag{8.4}$$
$$l_1 \le k_1, \tag{8.5}$$
$$l_2 \le k_2. \tag{8.6}$$
$$\tag{8.7}$$

接下来，只需要考虑如何快速算出 k_1 与 k_2 了。借助 Z 函数，我们可以 $O(1)$ 计算它们：

- 计算 k_1：只需计算 $\overline{u}$ 的 Z 函数即可。
- 计算 k_2：只需计算 $v + \# + u$ 的 Z 函数即可，其中 $\#$ 是一个 u，v 中均没有的字符。

右偏重串

计算右偏重串的方法与计算左偏重串的方法几乎一致。考虑该重串第一个落入 u 的字符（即 $s[|u| - 1]$），则其一定与 v 中的某个字符一致，记这个字符在 v 中的位置为 $cntr$。

令 k_1 为满足 $v[cntr - k_1 + 1 \ldots cntr] = u[|u| - k_1 \ldots |u| - 1]$ 的最大整数，k_2 为满足 $v[cntr + 1 \ldots cntr + k_2] = v[0 \ldots k_2 - 1]$ 的最大整数。则我们可以分别通过计算 $\overline{u} + \# + \overline{v}$ 和 v 的 Z 函数来得出 k_1 与 k_2。

枚举 $cntr$，用相仿的方法寻找右偏重串即可。

实现

Main–Lorentz 算法以四元组 $(cntr, l, k_1, k_2)$ 的形式给出所有重串。如果你只需要计算重串的数量，或者只需要找到最长的一个重串，这个四元组给的信息是足够的。由 主定理 可得，Main–Lorentz 算法的时间复杂度为 $O(n \log n)$。

请注意，如果你想通过这些四元组来找到所有重串的起始位置与终止位置，则最坏时间复杂度会达到 $O(n^2)$。我们在下面的程序中实现了这一点，将所有重串的起始位置与终止位置存于 `repetitions` 中。

```cpp
vector<int> z_function(string const& s) {
  int n = s.size();
  vector<int> z(n);
  for (int i = 1, l = 0, r = 0; i < n; i++) {
    if (i <= r) z[i] = min(r - i + 1, z[i - l]);
    while (i + z[i] < n && s[z[i]] == s[i + z[i]]) z[i]++;
    if (i + z[i] - 1 > r) {
      l = i;
      r = i + z[i] - 1;
    }
  }
  return z;
}

int get_z(vector<int> const& z, int i) {
  if (0 <= i && i < (int)z.size())
    return z[i];
  else
    return 0;
}

vector<pair<int, int>> repetitions;

void convert_to_repetitions(int shift, bool left, int cntr, int l, int k1,
                            int k2) {
  for (int l1 = max(1, l - k2); l1 <= min(l, k1); l1++) {
    if (left && l1 == l) break;
    int l2 = l - l1;
    int pos = shift + (left ? cntr - l1 : cntr - l - l1 + 1);
    repetitions.emplace_back(pos, pos + 2 * l - 1);
  }
}

void find_repetitions(string s, int shift = 0) {
  int n = s.size();
  if (n == 1) return;

  int nu = n / 2;
  int nv = n - nu;
  string u = s.substr(0, nu);
  string v = s.substr(nu);
  string ru(u.rbegin(), u.rend());
  string rv(v.rbegin(), v.rend());

  find_repetitions(u, shift);
  find_repetitions(v, shift + nu);

  vector<int> z1 = z_function(ru);
  vector<int> z2 = z_function(v + '#' + u);
  vector<int> z3 = z_function(ru + '#' + rv);
  vector<int> z4 = z_function(v);

  for (int cntr = 0; cntr < n; cntr++) {
    int l, k1, k2;
```

```
      if (cntr < nu) {
        l = nu - cntr;
        k1 = get_z(z1, nu - cntr);
        k2 = get_z(z2, nv + 1 + cntr);
      } else {
        l = cntr - nu + 1;
        k1 = get_z(z3, nu + 1 + nv - 1 - (cntr - nu));
        k2 = get_z(z4, (cntr - nu) + 1);
      }
      if (k1 + k2 >= l) convert_to_repetitions(shift, cntr < nu, cntr, l, k1, k2);
    }
  }
```

本页面主要译自博文 . - [1] 与其英文翻译版 Finding repetitions[2]。其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0。

参考资料与注释

[1] . -

[2] Finding repetitions

第 9 章

数学

9.1 数学部分简介

本章介绍 OI 中可能会用到的数学知识。计算机科学与数学紧密相关，而在算法竞赛中尤其强调以数论、排列组合、概率期望、多项式为代表离散、具体的数学：其注重程序实现和现实问题，可以出现在几乎任何类别的题目中。

实际上，算法竞赛中涉及到的算法和数据结构以及自动机等也可以被认为属于数学范畴，但是这些内容被细分到诸如字符串等的具体章节加以应用背景以更好理解。本章主要介绍数学中一些基础概念、代数、数论、博弈论及概率论等知识。运用这些知识可以帮助优化其他算法和数据结构，例如：

- 用多项式优化卷积形式的背包，来做一些字符串题。
- 很多递推类型的题背景都是排列组合或概率期望，可以更进一步用生成函数推导和解决，以及使用基于 FFT 的分治优化算法效率。
- 利用同余和环分析图上非简单路径在模意义下可能的权值和，并用带权并查集维护。

此外，高中数学是信息学竞赛数学的基础，学好课本上的基本概念和性质能更好地帮助学习本章内容。

9.2 符号

Authors: sshwy, hsfzLZH1, Enter-tainer

在学习数学的过程中大家会见到许多复杂的公式符号。因此在学习具体内容之前，建议大家首先理解下列常见符号的含义。一些特殊的符号会在对应的章节中讲到，而这里则有一些极为常见的符号需要大家提前掌握。

渐进符号

请参见 复杂度。

整除/同余理论常见符号

1. 整除符号：$x \mid y$，表示 x 整除 y，即 x 是 y 的因数。
2. 取模符号：$x \bmod y$，表示 x 除以 y 得到的余数。
3. 互质符号：$x \perp y$，表示 x，y 互质。
4. 最大公约数：$\gcd(x, y)$，在无混淆意义的时候可以写作 (x, y)。
5. 最小公倍数：$\operatorname{lcm}(x, y)$，在无混淆意义的时候可以写作 $[x, y]$。

数论函数常见符号

求和符号：$\sum$ 符号，表示满足特定条件的数的和。举几个例子：

- $\sum_{i=1}^{n} i$ 表示 $1+2+\cdots+n$ 的和。其中 i 是一个变量，在求和符号的意义下 i 通常是**正整数或者非负整数**（除非特殊说明）。这个式子的含义可以理解为，i 从 1 循环到 n，所有 i 的和。这个式子用代码的形式很容易表达。当然，学过简单的组合数学的同学都知道 $\sum_{i=1}^{n} i = \dfrac{n(n+1)}{2}$。
- $\sum_{S \subseteq T} |S|$ 表示所有被 T 包含的集合的大小的和。
- $\sum_{p \le n, p \perp n} 1$ 表示的是 n 以内有多少个与 n 互质的数，即 $\varphi(n)$，φ 是欧拉函数。

求积符号：$\prod$ 符号，表示满足特定条件的数的积。举几个例子：

- $\prod_{i=1}^{n} i$ 表示 n 的阶乘，即 $n!$。在组合数学常见符号中会讲到。
- $\prod_{i=1}^{n} a_i$ 表示 $a_1 \times a_2 \times a_3 \times \cdots \times a_n$。
- $\prod_{x|d} x$ 表示 d 的所有因数的乘积。

在行间公式中，求和符号与求积符号的上下条件会放到符号的上面和下面，这一点要注意。

其他常见符号

1. 阶乘符号 $!$，$n!$ 表示 $1 \times 2 \times 3 \times \cdots \times n$。特别地，$0! = 1$。
2. 向下取整符号：$\lfloor x \rfloor$，表示小于等于 x 的最大的整数。常用于分数，比如分数的向下取整 $\left\lfloor \dfrac{x}{y} \right\rfloor$。
3. 向上取整符号：$\lceil x \rceil$，与向下取整符号相对，表示大于等于 x 的最小的整数。
4. 组合数：$\binom{x}{y}$
5. 第一类斯特林数：$\begin{bmatrix} x \\ y \end{bmatrix}$
6. 第二类斯特林数：$\begin{Bmatrix} x \\ y \end{Bmatrix}$

9.3 进位制

在计算机中，除了二进制，比较常用的还有八进制和十六进制。

二进制

二进制是计算机内部运算中采用的进制，在这样的进制系统下，只有 $0,1$ 两个数字，计算机内部的所有运算（包括位运算）都是在二进制的基础上进行的。

但用二进制表示数字会让数字过长，因此为了方便表示的需要，通常会把二进制数转换为八进制或十六进制表示。

八进制

在八进制下，有 $0,1,2,3,4,5,6,7$ 八个数字。

一般情况下，八进制数以 `oxx`（其中 `o` 为八进制的前缀，`xx` 代表八进制数）的形式来表示。

十六进制

在十六进制下，有 $0,1,2,3,4,5,6,7,8,9,A(10),B(11),C(12),D(13),E(14),F(15)$ 十六个数字。

十六进制与二进制相比，最大的优点就是表示的数字长度较短，一位十六进制数可以表示 4 位二进制数。

一般情况下，十六进制数以 `0xdbf`（其中 `0x` 为十六进制数的前缀）的形式来表示。

进制间的相互转化

十进制转二进制/八进制/十六进制

这里以二进制为例来演示，其他进制的原理与其类似。

整数部分，把十进制数不断执行除 2 操作，直至商数为 0。读余数从下读到上，即是二进制的整数部分数字。小数部分，则用其乘 2，取其整数部分的结果，再用计算后的小数部分依此重复计算，算到小数部分全为 0 为止，之后从上到下，读所有计算后整数部分的数字，即为二进制的小数部分数字。

```
将 33.25 转化为二进制数
整数部分:
33/2=16    ......1
16/2=8        ......0
8/2=4        ......0
4/2=2        ......0
2/2=1        ......0
1/2=0        ......1
小数部分:
0.25*2=0.5       0
0.5*2=1               1
```

即 $33.25 = (100001.01)_2$

二进制/八进制/十六进制转十进制

还是以二进制为例。

二进制数转换为十进制数，只需将每个位的值，乘以 2^i 次即可，其中 i 为当前位的位数，个位的位数为 0。

```
将 11010.01(2) 转换为十进制数
11010.01(2)=1*2^4+1*2^3+0*2^2+1*2^1+0*2^0+0*2^(-1)+1*2(-2)
        =26.25
```

即 $(11010.01)_2 = (26.25)_{10}$

二进制/八进制/十六进制间的相互转换

一个八进制位可以用 3 个二进制位来表示（因为 $2^3 = 8$），一个十六进制位可以用 4 个二进制位来表示（$2^4 = 16$），反之同理。

9.4 位运算

位运算就是基于整数的二进制表示进行的运算。由于计算机内部就是以二进制来存储数据，位运算是相当快的。

基本的位运算共 6 种，分别为按位与、按位或、按位异或、按位取反、左移和右移。

为了方便叙述，下文中省略「按位」。

与、或、异或

这三者都是两数间的运算，因此在这里一起讲解。

它们都是将两个整数作为二进制数，对二进制表示中的每一位逐一运算。

运算	运算符	数学符号表示	解释
与	`&`	$\&$、and	只有两个对应位都为 1 时才为 1
或	‘	‘	|、or
异或	`^`	$\oplus$、xor	只有两个对应位不同时才为 1

注意区分逻辑与（对应的数学符号为 $\wedge$）和按位与、逻辑或（$\vee$）和按位或的区别。网络中的资料中使用的符号多有不规范之处，以上下文为准。

异或运算的逆运算是它本身，也就是说两次异或同一个数最后结果不变，即 $a \oplus b \oplus b = a$ 。

举例：

$$\begin{aligned} 5 &= (101)_2 \\ 6 &= (110)_2 \\ 5 \,\&\, 6 &= (100)_2 = \ 4 \\ 5 \,|\, 6 &= (111)_2 = \ 7 \\ 5 \oplus 6 &= (011)_2 = \ 3 \end{aligned}$$

取反

取反是对一个数 num 进行的位运算，即单目运算。

取反暂无默认的数学符号表示，其对应的运算符为 `~`。它的作用是把 num 的二进制补码中的 0 和 1 全部取反（0 变为 1，1 变为 0）。有符号整数的符号位在 `~` 运算中同样会取反。

补码：在二进制表示下，正数和 0 的补码为其本身，负数的补码是将其对应正数按位取反后加一。

举例（有符号整数）：

$$\begin{aligned} 5 &= (00000101)_2 \\ \sim 5 &= (11111010)_2 = -6 \\ -5 \text{ 的补码} &= (11111011)_2 \\ \sim(-5) &= (00000100)_2 = 4 \end{aligned}$$

左移和右移

`num << i` 表示将 num 的二进制表示向左移动 i 位所得的值。

`num >> i` 表示将 num 的二进制表示向右移动 i 位所得的值。

举例：

$$\begin{aligned} 11 &= (00001011)_2 \\ 11 << 3 &= (01011000)_2 = 88 \\ 11 >> 2 &= (00000010)_2 = 2 \end{aligned}$$

移位运算中如果出现如下情况，则其行为未定义：

1. 右操作数（即移位数）为负值；
2. 右操作数大于等于左操作数的位数；

例如，对于 `int` 类型的变量 `a` ，`a<<-1` 和 `a<<32` 都是未定义的。

对于左移操作，需要确保移位后的结果能被原数的类型容纳，否则行为也是未定义的。[1] 对一个负数执行左移操作也未定义。[2]

对于右移操作，右侧多余的位将会被舍弃，而左侧较为复杂：对于无符号数，会在左侧补 0；而对于有符号数，则会用最高位的数（其实就是符号位，非负数为 0，负数为 1）补齐。[3]

复合赋值位运算符

和 `+=` , `-=` 等运算符类似，位运算也有复合赋值运算符：`&=` , `|=` , `^=` , `<<=` , `>>=` 。（取反是单目运算，所以没有。）

关于优先级

位运算的优先级低于算术运算符（除了取反），而按位与、按位或及异或低于比较运算符（详见 C++ 运算符优先级总表），所以使用时需多加注意，在必要时添加括号。

位运算的应用

位运算一般有三种作用：

1. 高效地进行某些运算，代替其它低效的方式。
2. 表示集合（常用于 状压 DP）。
3. 题目本来就要求进行位运算。

需要注意的是，用位运算代替其它运算方式（即第一种应用）在很多时候并不能带来太大的优化，反而会使代码变得复杂，使用时需要斟酌。（但像「乘 2 的非负整数次幂」和「除以 2 的非负整数次幂」就最好使用位运算，因为此时使用位运算可以优化复杂度。）

有关 2 的幂的应用

由于位运算针对的是变量的二进制位，因此可以推广出许多与 2 的整数次幂有关的应用。

将一个数乘（除）2 的非负整数次幂：

```
int mulPowerOfTwo(int n, int m) {  // 计算 n*(2^m)
  return n << m;
}
int divPowerOfTwo(int n, int m) {  // 计算 n/(2^m)
  return n >> m;
}
```

```
def mulPowerOfTwo(n, m): # 计算 n*(2^m)
    return n << m
def divPowerOfTwo(n, m): # 计算 n/(2^m)
    return n >> m
```

warning

我们平常写的除法是向 0 取整，而这里的右移是向下取整（注意这里的区别），即当数大于等于 0 时两种方法等价，当数小于 0 时会有区别，如：`-1 / 2` 的值为 0 ，而 `-1 >> 1` 的值为 -1 。

取绝对值

在某些机器上，效率比 `n > 0 ? n : -n` 高。

```
int Abs(int n) {
  return (n ^ (n >> 31)) - (n >> 31);
  /* n>>31 取得 n 的符号，若 n 为正数，n>>31 等于 0，若 n 为负数，n>>31 等于 -1
    若 n 为正数 n^0=n, 数不变，若 n 为负数有 n^(-1)
    需要计算 n 和 -1 的补码，然后进行异或运算，
    结果 n 变号并且为 n 的绝对值减 1，再减去 -1 就是绝对值 */
```

```
}
```

```python
def Abs(n):
    return (n ^ (n >> 31)) - (n >> 31)
    """
    n>>31 取得 n 的符号，若 n 为正数，n>>31 等于 0，若 n 为负数，n>>31 等于 -1
    若 n 为正数 n^0=n, 数不变，若 n 为负数有 n^(-1)
    需要计算 n 和 -1 的补码，然后进行异或运算，
    结果 n 变号并且为 n 的绝对值减 1，再减去 -1 就是绝对值
    """
```

取两个数的最大/最小值

在某些机器上，效率比 `a > b ? a : b` 高。

```cpp
// 如果 a >= b, (a - b) >> 31 为 0，否则为 -1
int max(int a, int b) { return (b & ((a - b) >> 31)) | (a & (~(a - b) >> 31)); }
int min(int a, int b) { return (a & ((a - b) >> 31)) | (b & (~(a - b) >> 31)); }
```

```python
# 如果 a >= b, (a - b) >> 31 为 0，否则为 -1
def max(a, b):
    return b & ((a - b) >> 31) | a & (~(a - b) >> 31)
def min(a, b):
    return a & ((a - b) >> 31) | b & (~(a - b) >> 31)
```

判断两非零数符号是否相同

```cpp
bool isSameSign(int x, int y) {  // 有 0 的情况例外
  return (x ^ y) >= 0;
}
```

```python
# 有 0 的情况例外
def isSameSign(x, y):
    return (x ^ y) >= 0
```

交换两个数

" 该方法具有局限性"

这种方式只能用来交换两个整数，使用范围有限。

对于一般情况下的交换操作，推荐直接调用 `algorithm` 库中的 `std::swap` 函数。

```cpp
void swap(int &a, int &b) { a ^= b ^= a ^= b; }
```

操作一个数的二进制位

获取一个数二进制的某一位：

```
// 获取 a 的第 b 位，最低位编号为 0
int getBit(int a, int b) { return (a >> b) & 1; }
```

```
# 获取 a 的第 b 位，最低位编号为 0
def getBit(a, b):
    return (a >> b) & 1
```

将一个数二进制的某一位设置为 0：

```
// 将 a 的第 b 位设置为 0 ，最低位编号为 0
int unsetBit(int a, int b) { return a & ~(1 << b); }
```

```
# 将 a 的第 b 位设置为 0 ，最低位编号为 0
def unsetBit(a, b):
    return a & ~(1 << b)
```

将一个数二进制的某一位设置为 1：

```
// 将 a 的第 b 位设置为 1 ，最低位编号为 0
int setBit(int a, int b) { return a | (1 << b); }
```

```
# 将 a 的第 b 位设置为 1 ，最低位编号为 0
def setBit(a, b):
    return a | (1 << b)
```

将一个数二进制的某一位取反：

```
// 将 a 的第 b 位取反 ，最低位编号为 0
int flapBit(int a, int b) { return a ^ (1 << b); }
```

```
# 将 a 的第 b 位取反 ，最低位编号为 0
def flapBit(a, b):
    return a ^ (1 << b)
```

这些操作相当于将一个 32 位整型变量当作一个长度为 32 的布尔数组。

汉明权重

汉明权重是一串符号中不同于（定义在其所使用的字符集上的）零符号（zero-symbol）的个数。对于一个二进制数，它的汉明权重就等于它 1 的个数（即 `popcount`）。

求一个数的汉明权重可以循环求解：我们不断地去掉这个数在二进制下的最后一位（即右移 1 位），维护一个答案变量，在除的过程中根据最低位是否为 1 更新答案。

代码如下：

```
// 求 x 的汉明权重
int popcount(int x) {
    int cnt = 0;
    while (x) {
        cnt += x & 1;
        x >>= 1;
    }
    return cnt;
}
```

求一个数的汉明权重还可以使用 `lowbit` 操作：我们将这个数不断地减去它的 `lowbit`[4]，直到这个数变为 0。

代码如下：

```
// 求 x 的汉明权重
int popcount(int x) {
    int cnt = 0;
    while (x) {
        cnt++;
        x -= x & -x;
    }
    return cnt;
}
```

构造汉明权重递增的排列

在 状压 DP 中，按照 popcount 递增的顺序枚举有时可以避免重复枚举状态。这是构造汉明权重递增的排列的一大作用。

下面我们来具体探究如何在 $O(n)$ 时间内构造汉明权重递增的排列。

我们知道，一个汉明权重为 n 的最小的整数为 2^n-1。只要可以在常数时间构造出一个整数汉明权重相等的后继，我们就可以通过枚举汉明权重，从 2^n-1 开始不断寻找下一个数的方式，在 $O(n)$ 时间内构造出 $0\sim n$ 的符合要求的排列。

而找出一个数 x 汉明权重相等的后继有这样的思路，以 $(10110)_2$ 为例：

- 把 $(10110)_2$ 最右边的 1 向左移动，如果不能移动，移动它左边的 1，以此类推，得到 $(11010)_2$。
- 把得到的 $(11010)_2$ 最后移动的 1 原先的位置一直到最低位的所有 1 都移到最右边。这里最后移动的 1 原来在第三位，所以最后三位 010 要变成 001，得到 $(11001)_2$。

这个过程可以用位运算优化：

```
int t = x + (x & -x);
x = t | ((((t&-t)/(x&-x))>>1)-1);
```

- 第一个步骤中，我们把数 x 加上它的 `lowbit`，在二进制表示下，就相当于把 x 最右边的连续一段 1 换成它左边的一个 1。如刚才提到的二进制数 $(10110)_2$，它在加上它的 `lowbit` 后是 $(11000)_2$。这其实得到了我们答案的前半部分。
- 我们接下来要把答案后面的 1 补齐，t 的 `lowbit` 是 x 最右边连续一段 1 最左边的 1 移动后的位置，而 x 的 `lowbit` 则是 x 最右边连续一段 1 最左边的位置。还是以 $(10110)_2$ 为例，$t=(11000)_2$，$\text{lowbit}(t)=(01000)_2$，$\text{lowbit}(x)=(00010)_2$。
- 接下来的除法操作是这种位运算中最难理解的部分，但也是最关键的部分。我们设**原数**最右边连续一段 1 最高位的 1 在第 r 位上（位数从 0 开始），最低位的 1 在第 l 位，t 的 `lowbit` 等于 `1 << (r+1)` ，x 的 `lowbit` 等于 `1 << l`，`(((t&-t)/(x&-x))>>1)` 得到的，就是 `(1<<(r+1))/(1<<l)/2 = (1<<r)/(1<<l) = 1<<(r-l)` ，在二进制表示下就是 1 后面跟上 $r-l$ 个零，零的个数正好等于连续 1 的个数减去 1 。举我们刚才的数为例，$\frac{\text{lowbit(t)}/2}{\text{lowbit(x)}}=\frac{(00100)_2}{(00010)_2}=(00010)_2$ 。把这个数减去 1 得到的就是我们要补全的低位，或上原来的数就可以得到答案。

所以枚举 $0\sim n$ 按汉明权重递增的排列的完整代码为：

```
for (int i = 0; (1<<i)-1 <= n; i++) {
    for (int x = (1<<i)-1, t; x <= n; t = x+(x&-x), x = x ? (t|((((t&-t)/(x&-x))
>>1)-1)) : (n+1)) {
        //  写下需要完成的操作
    }
```

```
}
```

其中要注意 0 的特判，因为 0 没有相同汉明权重的后继。

内建函数

GCC 中还有一些用于位运算的内建函数：

1. `int __builtin_ffs(int x)`：返回 x 的二进制末尾最后一个 1 的位置，位置的编号从 1 开始（最低位编号为 1）。当 x 为 0 时返回 0。
2. `int __builtin_clz(unsigned int x)`：返回 x 的二进制的前导 0 的个数。当 x 为 0 时，结果未定义。
3. `int __builtin_ctz(unsigned int x)`：返回 x 的二进制末尾连续 0 的个数。当 x 为 0 时，结果未定义。
4. `int __builtin_clrsb(int x)`：当 x 的符号位为 0 时返回 x 的二进制的前导 0 的个数减一，否则返回 x 的二进制的前导 1 的个数减一。
5. `int __builtin_popcount(unsigned int x)`：返回 x 的二进制中 1 的个数。
6. `int __builtin_parity(unsigned int x)`：判断 x 的二进制中 1 的个数的奇偶性。

这些函数都可以在函数名末尾添加 `l` 或 `ll`（如 `__builtin_popcountll`）来使参数类型变为（`unsigned`）`long` 或（`unsigned`）`long long`（返回值仍然是 `int` 类型）。例如，我们有时候希望求出一个数以二为底的对数，如果不考虑 `0` 的特殊情况，就相当于这个数二进制的位数 `-1`，而一个数 `n` 的二进制表示的位数可以使用 `32-__builtin_clz(n)` 表示，因此 `31-__builtin_clz(n)` 就可以求出 `n` 以二为底的对数。

由于这些函数是内建函数，经过了编译器的高度优化，运行速度十分快（有些甚至只需要一条指令）。

更多位数

如果需要操作的集合非常大，可以使用 bitset。

题目推荐

Luogu P1225 黑白棋游戏[5]

参考资料与注释

1. 位运算技巧：https://graphics.stanford.edu/~seander/bithacks.html[6]
2. Other Builtins of GCC：https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html[7]

[1] 适用于 C++14 以前的标准。在 C++14 和 C++17 标准中，若原值为带符号类型，且移位后的结果能被原类型的无符号版本容纳，则将该结果 转换 为相应的带符号值，否则行为未定义。在 C++20 标准中，规定了无论是带符号数还是无符号数，左移均直接舍弃移出结果类型的位。

[2] 适用于 C++20 以前的标准。

[3] 这种右移方式称为算术右移。在 C++20 以前的标准中，并没有规定带符号数右移运算的实现方式，大多数平台均采用算术右移。在 C++20 标准中，规定了带符号数右移运算是算术右移。

[4] 一个数二进制表示从低往高的第一个 1 连同后面的零，如 $(1010)_2$ 的 `lowbit` 是 $(0010)_2$，详见 树状数组。

[5] Luogu P1225 黑白棋游戏

[6] https://graphics.stanford.edu/~seander/bithacks.html

[7] https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html

9.5 二进制集合操作

二进制集合操作

一个数的二进制表示可以看作是一个集合（0 表示不在集合中，1 表示在集合中）。比如集合 $\{1,3,4,8\}$，可以表示成 $(100011010)_2$。而对应的位运算也就可以看作是对集合进行的操作。

操作	集合表示	位运算语句
交集	$a \cap b$	`a & b`
并集	$a \cup b$	a | b
补集	$\bar{a}$	`~a` （全集为二进制都是 1）
差集	$a \quad b$	`a & (~b)`
对称差	$a \triangle b$	`a ^ b`

在进一步介绍集合的子集遍历操作之前，先看位运算的有关应用例子。

模 2 的幂

一个数对 2 的非负整数次幂取模，等价于取二进制下一个数的后若干位，等价于和 $mod - 1$ 进行与操作。

```cpp
int modPowerOfTwo(int x, int mod) { return x & (mod - 1); }
```

```python
def modPowerOfTwo(x, mod):
    return x & (mod - 1)
```

于是可以知道，2 的非负整数次幂对它本身取模，结果为 0，即如果 n 是 2 的非负整数次幂，n 和 $n-1$ 的与操作结果为 0。

事实上，对于一个正整数 n，$n-1$ 会将 n 的最低 1 位置零，并将后续位数全部置 1。因此，n 和 $n-1$ 的与操作等价于删掉 n 的最低 1 位。

借此可以判断一个数是不是 2 的非负整数次幂。当且仅当 n 的二进制表示只有一个 1 时，n 为 2 的非负整数次幂。

```cpp
bool isPowerOfTwo(int n) { return n > 0 && (n & (n - 1)) == 0; }
```

```python
def isPowerOfTwo(n):
    return n > 0 and (n & (n - 1)) == 0
```

子集遍历

遍历一个二进制数表示的集合的全部子集，等价于枚举二进制数对应掩码的所有子掩码。

掩码是一串二进制码，用于和源码进行与运算，得到屏蔽源码的若干输入位后的新操作数。

掩码对于源码可以起到遮罩的作用，掩码中的 1 位意味着源码的相应位得到保留，掩码中的 0 位意味着源码的相应位进行置 0 操作。将掩码的若干 1 位改为 0 位可以得到掩码的子掩码，掩码本身也是自己的子掩码。

给定一个掩码 m，希望有效迭代 m 的所有子掩码 s，可以考虑基于位运算技巧的实现。

```
// 降序遍历 m 的非空子集
int s = m;
while (s > 0) {
  // s 是 m 的一个非空子集
  s = (s - 1) & m;
}
```

或者使用更紧凑的 for 语句：

```
// 降序遍历 m 的非空子集
for (int s = m; s; s = (s - 1) & m)
// s 是 m 的一个非空子集
```

这两段代码都不会处理等于 0 的子掩码，要想处理等于 0 的子掩码可以使用其他办法，例如：

```
// 降序遍历 m 的子集
for (int s = m;; s = (s - 1) & m) {
  // s 是 m 的一个子集
  if (s == 0) break;
}
```

接下来证明，上面的代码访问了所有 m 的子掩码，没有重复，并且按降序排列。

假设有一个当前位掩码 s，并且想继续访问下一个位掩码。在掩码 s 中减去 1，等价于删除掩码 s 中最右边的设置位，并将其右边的所有位变为 1。

为了使 $s-1$ 变为新的子掩码，需要删除掩码 m 中未包含的所有额外的 1 位，可以使用位运算 $(s-1)\&m$ 来进行此移除。

这两步操作等价于切割掩码 $s-1$，以确定算术上可以取到的最大值，即按降序排列的 s 之后的下一个子掩码。

因此，该算法按降序生成该掩码的所有子掩码，每次迭代仅执行两个操作。

特殊情况是 $s=0$。在执行 $s-1$ 之后得到 -1，其中所有位都为 1。在 $(s-1)\&m$ 操作之后将得到新的 s 等于 m。因此，如果循环不以 $s=0$ 结束，算法的循环将无法终止。

使用 $\mathrm{popcount}(m)$ 表示 m 二进制中 1 的个数，用这种方法可以在 $O(2^{\mathrm{popcount}(m)})$ 的时间复杂度内遍历集合 m 的子集。

遍历所有掩码的子掩码

在使用掩码动态编程的问题中，有时会希望对于每个掩码，遍历掩码的所有子掩码：

```
for (int m = 0; m < (1 << n); ++m)
  // 降序遍历 m 的非空子集
  for (int s = m; s; s = (s - 1) & m)
// s 是 m 的一个非空子集
```

这样做可以遍历大小为 n 的集合的每个子集的子集。

接下来证明，该操作的时间复杂度为 $O(3^n)$，n 为掩码总共的位数，即集合中元素的总数。

考虑第 i 位，即集合中第 i 个元素，有三种情况：

- 在掩码 m 中为 0，因此在子掩码 s 中为 0，即元素不在大小子集中。
- 在 m 中为 1，但在 s 中为 0，即元素只在大子集中，不在小子集中。
- 在 m 和 s 中均为 1，即元素同时在大小子集中。

总共有 n 位，因此有 3^n 个不同的组合。

还有一种证明方法是：

如果掩码 m 具有 k 个 1，那么它有 2^k 个子掩码。对于给定的 k，对应有 $\binom{n}{k}$ 个掩码 m，那么所有掩码的总数为：

$$\sum_{k=0}^{n}\binom{n}{k}2^n$$

上面的和等于使用二项式定理对 $(1+2)^n$ 的展开，因此有 3^n 个不同的组合。

参考资料

本页面主要译自博文 [1] 与其英文翻译版 Submask Enumeration[2]。其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0。

习题

- Atcoder - Close Group[3]
- Codeforces - Nuclear Fusion[4]
- Codeforces - Sandy and Nuts[5]
- Uva 1439 - Exclusive Access 2[6]
- UVa 11825 - Hackers' Crackdown[7]

参考资料与注释

[1]

[2] Submask Enumeration

[3] Atcoder - Close Group

[4] Codeforces - Nuclear Fusion

[5] Codeforces - Sandy and Nuts

[6] Uva 1439 - Exclusive Access 2

[7] UVa 11825 - Hackers' Crackdown

9.6 平衡三进制

定义

平衡三进制，也称为对称三进制。这是一个不太标准的**计数体系**。

正规的三进制的数字都是由 `0,1,2` 构成的，而平衡三进制的数字是由 `-1,0,1` 构成的。它的基数也是 `3`（因为有三个可能的值）。由于将 `-1` 写成数字不方便，我们将使用字母 `Z` 来代替 `-1`。

解释

这里有几个例子：

十进制	平衡三进制	十进制	平衡三进制
0	0	5	1ZZ
1	1	6	1Z0
2	1Z	7	1Z1
3	10	8	10Z
4	11	9	100

该**计数体系**的负数表示起来很容易：只需要将正数的数字倒转即可（Z 变成 1,1 变成 Z）。

十进制	平衡三进制
-1	Z
-2	Z1
-3	Z0
-4	ZZ
-5	Z11

很容易就可以看到，负数最高位是 Z，正数最高位是 1。

过程

在平衡三进制的转转换法中，需要先写出一个给定的数 x 在标准三进制中的表示。当 x 是用标准三进制表示时，其数字的每一位都是 0、1 或 2。从最低的数字开始迭代，我们可以先跳过任何的 0 和 1，但是如果遇到 2 就应该先将其变成 Z，下一位数字再加上 1。而遇到数字 3 则应该转换为 0 下一位数字再加上 1。

应用一

把 64 转换成平衡三进制。

首先，我们用标准三进制数来重写这个数：

$$64_{10} = 02101_3$$

让我们从对整个数影响最小的数字（最低位）进行处理：

- 101 被跳过（因为在平衡三进制中允许 0 和 1）；
- 2 变成了 Z，它左边的数字加 1，得到 1Z101；
- 1 被跳过，得到 1Z101。

最终的结果是 1Z101。

我们再把它转换回十进制：

$$\texttt{1Z101} = 81 \times 1 + 27 \times (-1) + 9 \times 1 + 3 \times 0 + 1 \times 1 = 64_{10}$$

应用二

把 237 转换成平衡三进制。

首先，我们用标准三进制数来重写这个数：

$$237_{10} = 22210_3$$

- `0` 和 `1` 被跳过（因为在平衡三进制中允许 `0` 和 `1`）；
- `2` 变成 `Z`，左边的数字加 `1`，得到 `23Z10`；
- `3` 变成 `0`，左边的数字加 `1`，得到 `30Z10`；
- `3` 变成 `0`，左边的数字（默认是 `0`）加 `1`，得到 `100Z10`；
- `1` 被跳过，得到 `100Z10`。

最终的结果是 `100Z10`。

我们再把它转换回十进制：

$$\texttt{100Z10} = 243 \cdot 1 + 81 \cdot 0 + 27 \cdot 0 + 9 \cdot (-1) + 3 \cdot 1 + 1 \cdot 0 = 237_{10}$$

性质

对于一个平衡三进制数 X_3 来说，其可以按照每一位 x_i 乘上对应的权值 3^i 来唯一得到一个十进制数 Y_{10}。

那对于一个十进制数 Y_{10}，是否**唯一对应一个平衡三进制数**呢？

答案是肯定的，这种性质被叫做平衡三进制的唯一性。

”证明”

我们利用**反证法**来求证：

假设一个十进制数 Y_{10}，存在两个**不同的平衡三进制数** A_3, B_3 转化成十进制时等于 Y_{10}，即证 $A_3 = B_3$。分情况讨论：

1. 当 $Y_{10} = 0$，显然 $A_3 = B_3 = 0_3$，与假设矛盾。
2. 当 $Y_{10} > 0$：
   - 将 A_3，B_3 的数位按低位到高位编号，记 a_i 为 A_3 的第 i 位，b_i 为 B 的第 i 位。在 A_3, B_3 中，必存在 i 使得 $a_i \neq b_i$。可以发现第 $i-1, i-2, \ldots, 0$ 位均与证明无关。因此，将 A_3, B_3 按位右移 i 位，得到 A_3', B_3'，原问题等价于证明 $A_3' = B_3'$。
   - 对于 A_3', B_3' 第 0 位，$a_0 \neq b_0$。假设 $b_0 > a_0$（$a_0 > b_0$ 时结果相同），易知 $b_0 - a_0 \in \{1, 2\}$。A_3' 的位 $i = 1, 2, 3, \ldots$ 对于 A_3' 的值的贡献为 $S_1 = a_1 \times 3^1 + a_2 \times 3^2 + \ldots$，$B_3'$ 的位 $i = 1, 2, 3, \ldots$ 对于 B_3' 的值的贡献为 $S_2 = b_1 \times 3^1 + b_2 \times 3^2 + \ldots$。由于 $A_3' = B_3'$，得 $S_1 - S_2 = b_0 - a_0$。S_1, S_2 有公因子 3，而 $b_0 - a_0$ 不能被 3 整除，与假设矛盾，因此 $A_3' \neq B_3'$
3. 当 $Y_{10} < 0$，证法与 $Y_{10} > 0$ 相同。

故对于任意十进制 Y_{10}，均有唯一对应的平衡三进制 X_3。

练习题

Topcoder SRM 604, Div1-250[1]

参考资料与注释

[1] Topcoder SRM 604, Div1-250

[2]

[3] Balanced Ternary

9.7 高精度计算

> 太长不看版：结尾自取模板……

定义

高精度计算（Arbitrary-Precision Arithmetic），也被称作大整数（bignum）计算，运用了一些算法结构来支持更大整数间的运算（数字大小超过语言内建整型）。

引入

高精度问题包含很多小的细节，实现上也有很多讲究。

所以今天就来一起实现一个简单的计算器吧。

" 任务"

输入：一个形如 a <op> b 的表达式。

- a、b 分别是长度不超过 1000 的十进制非负整数；
- <op> 是一个字符（+、-、* 或 /），表示运算。
- 整数与运算符之间由一个空格分隔。

输出：运算结果。

- 对于 +、-、* 运算，输出一行表示结果；
- 对于 / 运算，输出两行分别表示商和余数。
- 保证结果均为非负整数。

存储

在平常的实现中，高精度数字利用字符串表示，每一个字符表示数字的一个十进制位。因此可以说，高精度数值计算实际上是一种特别的字符串处理。

读入字符串时，数字最高位在字符串首（下标小的位置）。但是习惯上，下标最小的位置存放的是数字的**最低位**，即存储反转的字符串。这么做的原因在于，数字的长度可能发生变化，但我们希望同样权值位始终保持对齐（例如，希望所有的个位都在下标 [0]，所有的十位都在下标 [1]……）；同时，加、减、乘的运算一般都从个位开始进行（回想小学的竖式运算），这都给了「反转存储」以充分的理由。

此后我们将一直沿用这一约定。定义一个常数 LEN = 1004 表示程序所容纳的最大长度。

由此不难写出读入高精度数字的代码：

```
void clear(int a[]) {
  for (int i = 0; i < LEN; ++i) a[i] = 0;
}
```

```
void read(int a[]) {
  static char s[LEN + 1];
  scanf("%s", s);

  clear(a);

  int len = strlen(s);
  // 如上所述，反转
  for (int i = 0; i < len; ++i) a[len - i - 1] = s[i] - '0';
  // s[i] - '0' 就是 s[i] 所表示的数码
  // 有些同学可能更习惯用 ord(s[i]) - ord('0') 的方式理解
}
```

输出也按照存储的逆序输出。由于不希望输出前导零，故这里从最高位开始向下寻找第一个非零位，从此处开始输出；终止条件 `i >= 1` 而不是 `i >= 0` 是因为当整个数字等于 0 时仍希望输出一个字符 `0`。

```
void print(int a[]) {
  int i;
  for (i = LEN - 1; i >= 1; --i)
    if (a[i] != 0) break;
  for (; i >= 0; --i) putchar(a[i] + '0');
  putchar('\n');
}
```

拼起来就是一个完整的复读机程序咯。

"copycat.cpp"

```
#include <cstdio>
#include <cstring>

static const int LEN = 1004;

int a[LEN], b[LEN];

void clear(int a[]) {
  for (int i = 0; i < LEN; ++i) a[i] = 0;
}

void read(int a[]) {
  static char s[LEN + 1];
  scanf("%s", s);

  clear(a);

  int len = strlen(s);
  for (int i = 0; i < len; ++i) a[len - i - 1] = s[i] - '0';
}

void print(int a[]) {
  int i;
  for (i = LEN - 1; i >= 1; --i)
    if (a[i] != 0) break;
  for (; i >= 0; --i) putchar(a[i] + '0');
  putchar('\n');
```

```
}

int main() {
  read(a);
  print(a);

  return 0;
}
```

四则运算

四则运算中难度也各不相同。最简单的是高精度加减法，其次是高精度—单精度（普通的 `int`）乘法和高精度—高精度乘法，最后是高精度—高精度除法。

我们将按这个顺序分别实现所有要求的功能。

加法

高精度加法，其实就是竖式加法啦。

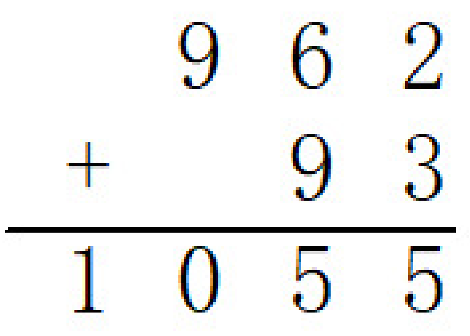


图 9.1

也就是从最低位开始，将两个加数对应位置上的数码相加，并判断是否达到或超过 10。如果达到，那么处理进位：将更高一位的结果上增加 1，当前位的结果减少 10。

```
void add(int a[], int b[], int c[]) {
  clear(c);

  // 高精度实现中，一般令数组的最大长度 LEN 比可能的输入大一些
  // 然后略去末尾的几次循环，这样一来可以省去不少边界情况的处理
  // 因为实际输入不会超过 1000 位，故在此循环到 LEN - 1 = 1003 已经足够
  for (int i = 0; i < LEN - 1; ++i) {
    // 将相应位上的数码相加
    c[i] += a[i] + b[i];
    if (c[i] >= 10) {
      // 进位
      c[i + 1] += 1;
      c[i] -= 10;
    }
  }
```

```
}
```

试着和上一部分结合，可以得到一个加法计算器。

"adder.cpp"

```cpp
#include <cstdio>
#include <cstring>

static const int LEN = 1004;

int a[LEN], b[LEN], c[LEN];

void clear(int a[]) {
  for (int i = 0; i < LEN; ++i) a[i] = 0;
}

void read(int a[]) {
  static char s[LEN + 1];
  scanf("%s", s);

  clear(a);

  int len = strlen(s);
  for (int i = 0; i < len; ++i) a[len - i - 1] = s[i] - '0';
}

void print(int a[]) {
  int i;
  for (i = LEN - 1; i >= 1; --i)
    if (a[i] != 0) break;
  for (; i >= 0; --i) putchar(a[i] + '0');
  putchar('\n');
}

void add(int a[], int b[], int c[]) {
  clear(c);

  for (int i = 0; i < LEN - 1; ++i) {
    c[i] += a[i] + b[i];
    if (c[i] >= 10) {
      c[i + 1] += 1;
      c[i] -= 10;
    }
  }
}

int main() {
  read(a);
  read(b);

  add(a, b, c);
  print(c);

  return 0;
```

```
}
```

减法

高精度减法，也就是竖式减法啦。

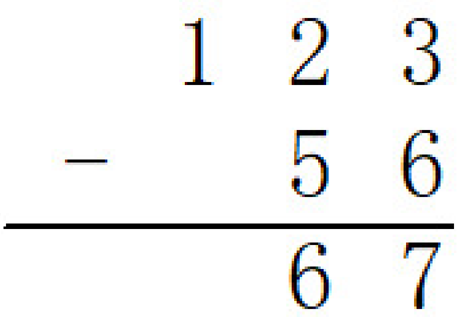


图 9.2

从个位起逐位相减，遇到负的情况则向上一位借 1。整体思路与加法完全一致。

```
void sub(int a[], int b[], int c[]) {
  clear(c);

  for (int i = 0; i < LEN - 1; ++i) {
    // 逐位相减
    c[i] += a[i] - b[i];
    if (c[i] < 0) {
      // 借位
      c[i + 1] -= 1;
      c[i] += 10;
    }
  }
}
```

将上一个程序中的 add() 替换成 sub()，就有了一个减法计算器。

"subtractor.cpp"

```
#include <cstdio>
#include <cstring>

static const int LEN = 1004;
```

```
int a[LEN], b[LEN], c[LEN];

void clear(int a[]) {
  for (int i = 0; i < LEN; ++i) a[i] = 0;
}

void read(int a[]) {
  static char s[LEN + 1];
  scanf("%s", s);

  clear(a);

  int len = strlen(s);
  for (int i = 0; i < len; ++i) a[len - i - 1] = s[i] - '0';
}

void print(int a[]) {
  int i;
  for (i = LEN - 1; i >= 1; --i)
    if (a[i] != 0) break;
  for (; i >= 0; --i) putchar(a[i] + '0');
  putchar('\n');
}

void sub(int a[], int b[], int c[]) {
  clear(c);

  for (int i = 0; i < LEN - 1; ++i) {
    c[i] += a[i] - b[i];
    if (c[i] < 0) {
      c[i + 1] -= 1;
      c[i] += 10;
    }
  }
}

int main() {
  read(a);
  read(b);

  sub(a, b, c);
  print(c);

  return 0;
}
```

试一试，输入 `1 2`——输出 `/9999999`，诶这个 **OI Wiki** 怎么给了我一份假的代码啊……

事实上，上面的代码只能处理减数 a 大于等于被减数 b 的情况。处理被减数比减数小，即 $a<b$ 时的情况很简单。

$a-b=-(b-a)$

要计算 $b-a$ 的值，因为有 $b>a$，可以调用以上代码中的 `sub` 函数，写法为 `sub(b,a,c)`。要得到 $a-b$ 的值，在得数前加上负号即可。

乘法

高精度—单精度

高精度乘法，也就是竖⋯⋯等会儿等会儿！

先考虑一个简单的情况：乘数中的一个是普通的 `int` 类型。有没有简单的处理方法呢？

一个直观的思路是直接将 a 每一位上的数字乘以 b。从数值上来说，这个方法是正确的，但它并不符合十进制表示法，因此需要将它重新整理成正常的样子。

重整的方式，也是从个位开始逐位向上处理进位。但是这里的进位可能非常大，甚至远大于 9，因为每一位被乘上之后都可能达到 $9b$ 的数量级。所以这里的进位不能再简单地进行 -10 运算，而是要通过除以 10 的商以及余数计算。详见代码注释，也可以参考下图展示的一个计算高精度数 1337 乘以单精度数 42 的过程。

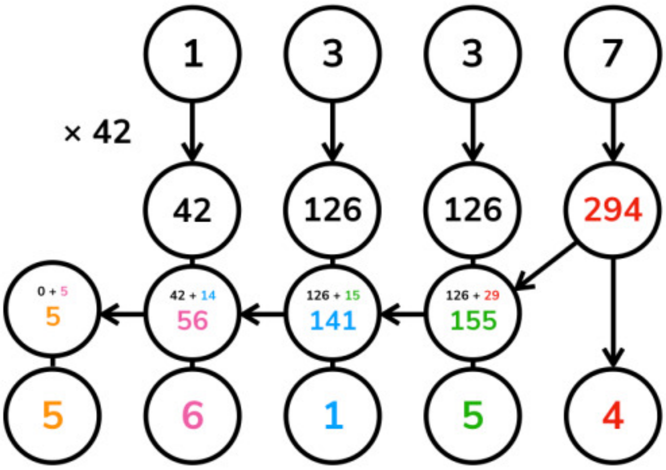


图 9.3

当然，也是出于这个原因，这个方法需要特别关注乘数 b 的范围。若它和 10^9（或相应整型的取值上界）属于同一数量级，那么需要慎用高精度—单精度乘法。

```cpp
void mul_short(int a[], int b, int c[]) {
  clear(c);

  for (int i = 0; i < LEN - 1; ++i) {
    // 直接把 a 的第 i 位数码乘以乘数，加入结果
    c[i] += a[i] * b;

    if (c[i] >= 10) {
      // 处理进位
      // c[i] / 10 即除法的商数成为进位的增量值
      c[i + 1] += c[i] / 10;
      // 而 c[i] % 10 即除法的余数成为在当前位留下的值
      c[i] %= 10;
    }
  }
}
```

高精度—高精度

如果两个乘数都是高精度，那么竖式乘法又可以大显身手了。

回想竖式乘法的每一步，实际上是计算了若干 $a \times b_i \times 10^i$ 的和。例如计算 1337×42，计算的就是 $1337 \times 2 \times 10^0 + 1337 \times 4 \times 10^1$。

于是可以将 b 分解为它的所有数码，其中每个数码都是单精度数，将它们分别与 a 相乘，再向左移动到各自的位置上相加即得答案。当然，最后也需要用与上例相同的方式处理进位。

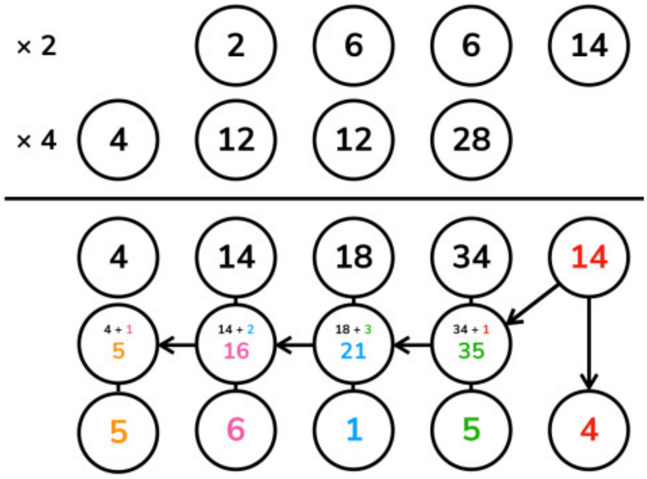


图 9.4

注意这个过程与竖式乘法不尽相同，我们的算法在每一步乘的过程中并不进位，而是将所有的结果保留在对应的位置上，到最后再统一处理进位，但这不会影响结果。

```
void mul(int a[], int b[], int c[]) {
  clear(c);

  for (int i = 0; i < LEN - 1; ++i) {
    // 这里直接计算结果中的从低到高第 i 位，且一并处理了进位
    // 第 i 次循环为 c[i] 加上了所有满足 p + q = i 的 a[p] 与 b[q] 的乘积之和
    // 这样做的效果和直接进行上图的运算最后求和是一样的，只是更加简短的一种实现方
式
    for (int j = 0; j <= i; ++j) c[i] += a[j] * b[i - j];

    if (c[i] >= 10) {
      c[i + 1] += c[i] / 10;
      c[i] %= 10;
    }
  }
}
```

除法

高精度除法的一种实现方式就是竖式长除法。

竖式长除法实际上可以看作一个逐次减法的过程。例如上图中商数十位的计算可以这样理解：将 45 减去三次 12 后变得小于 12，不能再减，故此位为 3。

为了减少冗余运算，我们提前得到被除数的长度 l_a 与除数的长度 l_b，从下标 $l_a - l_b$ 开始，从高位到低位来计算商。这和手工计算时将第一次乘法的最高位与被除数最高位对齐的做法是一样的。

参考程序实现了一个函数 `greater_eq()` 用于判断被除数以下标 `last_dg` 为最低位，是否可以再减去除数而保持非负。此后对于商的每一位，不断调用 `greater_eq()`，并在成立的时候用高精度减法从余数中减去除数，也即模拟了竖式除法的过程。

```
// 被除数 a 以下标 last_dg 为最低位，是否可以再减去除数 b 而保持非负
// len 是除数 b 的长度，避免反复计算
```

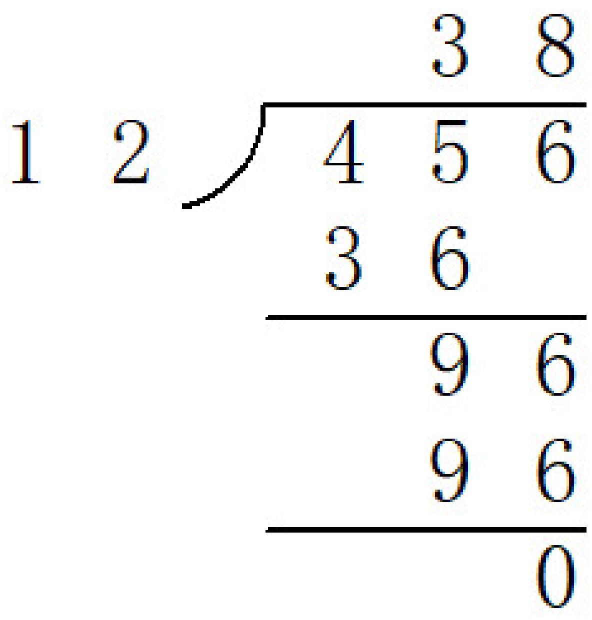


图 9.5

```
bool greater_eq(int a[], int b[], int last_dg, int len) {
  // 有可能被除数剩余的部分比除数长，这个情况下最多多出 1 位，故如此判断即可
  if (a[last_dg + len] != 0) return true;
  // 从高位到低位，逐位比较
  for (int i = len - 1; i >= 0; --i) {
    if (a[last_dg + i] > b[i]) return true;
    if (a[last_dg + i] < b[i]) return false;
  }
  // 相等的情形下也是可行的
  return true;
}

void div(int a[], int b[], int c[], int d[]) {
  clear(c);
  clear(d);

  int la, lb;
  for (la = LEN - 1; la > 0; --la)
    if (a[la - 1] != 0) break;
  for (lb = LEN - 1; lb > 0; --lb)
    if (b[lb - 1] != 0) break;
  if (lb == 0) {
    puts("> <");
    return;
  }  // 除数不能为零

  // c 是商
  // d 是被除数的剩余部分，算法结束后自然成为余数
  for (int i = 0; i < la; ++i) d[i] = a[i];
```

```
  for (int i = la - lb; i >= 0; --i) {
    // 计算商的第 i 位
    while (greater_eq(d, b, i, lb)) {
      // 若可以减，则减
      // 这一段是一个高精度减法
      for (int j = 0; j < lb; ++j) {
        d[i + j] -= b[j];
        if (d[i + j] < 0) {
          d[i + j + 1] -= 1;
          d[i + j] += 10;
        }
      }
      // 使商的这一位增加 1
      c[i] += 1;
      // 返回循环开头，重新检查
    }
  }
}
```

入门篇完成！

将上面介绍的四则运算的实现结合，即可完成开头提到的计算器程序。

"calculator.cpp"

```
#include <cstdio>
#include <cstring>

static const int LEN = 1004;

int a[LEN], b[LEN], c[LEN], d[LEN];

void clear(int a[]) {
  for (int i = 0; i < LEN; ++i) a[i] = 0;
}

void read(int a[]) {
  static char s[LEN + 1];
  scanf("%s", s);

  clear(a);

  int len = strlen(s);
  for (int i = 0; i < len; ++i) a[len - i - 1] = s[i] - '0';
}

void print(int a[]) {
  int i;
  for (i = LEN - 1; i >= 1; --i)
    if (a[i] != 0) break;
  for (; i >= 0; --i) putchar(a[i] + '0');
  putchar('\n');
}
```

```
void add(int a[], int b[], int c[]) {
  clear(c);

  for (int i = 0; i < LEN - 1; ++i) {
    c[i] += a[i] + b[i];
    if (c[i] >= 10) {
      c[i + 1] += 1;
      c[i] -= 10;
    }
  }
}

void sub(int a[], int b[], int c[]) {
  clear(c);

  for (int i = 0; i < LEN - 1; ++i) {
    c[i] += a[i] - b[i];
    if (c[i] < 0) {
      c[i + 1] -= 1;
      c[i] += 10;
    }
  }
}

void mul(int a[], int b[], int c[]) {
  clear(c);

  for (int i = 0; i < LEN - 1; ++i) {
    for (int j = 0; j <= i; ++j) c[i] += a[j] * b[i - j];

    if (c[i] >= 10) {
      c[i + 1] += c[i] / 10;
      c[i] %= 10;
    }
  }
}

bool greater_eq(int a[], int b[], int last_dg, int len) {
  if (a[last_dg + len] != 0) return true;
  for (int i = len - 1; i >= 0; --i) {
    if (a[last_dg + i] > b[i]) return true;
    if (a[last_dg + i] < b[i]) return false;
  }
  return true;
}

void div(int a[], int b[], int c[], int d[]) {
  clear(c);
  clear(d);

  int la, lb;
  for (la = LEN - 1; la > 0; --la)
    if (a[la - 1] != 0) break;
  for (lb = LEN - 1; lb > 0; --lb)
```

```
    if (b[lb - 1] != 0) break;
  if (lb == 0) {
    puts("> <");
    return;
  }

  for (int i = 0; i < la; ++i) d[i] = a[i];
  for (int i = la - lb; i >= 0; --i) {
    while (greater_eq(d, b, i, lb)) {
      for (int j = 0; j < lb; ++j) {
        d[i + j] -= b[j];
        if (d[i + j] < 0) {
          d[i + j + 1] -= 1;
          d[i + j] += 10;
        }
      }
      c[i] += 1;
    }
  }
}

int main() {
  read(a);

  char op[4];
  scanf("%s", op);

  read(b);

  switch (op[0]) {
    case '+':
      add(a, b, c);
      print(c);
      break;
    case '-':
      sub(a, b, c);
      print(c);
      break;
    case '*':
      mul(a, b, c);
      print(c);
      break;
    case '/':
      div(a, b, c, d);
      print(c);
      print(d);
      break;
    default:
      puts("> <");
  }

  return 0;
}
```

压位高精度

引入

在一般的高精度加法，减法，乘法运算中，我们都是将参与运算的数拆分成一个个单独的数码进行运算。

例如计算 8192×42 时，如果按照高精度乘高精度的计算方式，我们实际上算的是 $(8000+100+90+2) \times (40+2)$。

在位数较多的时候，拆分出的数也很多，高精度运算的效率就会下降。

有没有办法作出一些优化呢?

注意到拆分数字的方式并不影响最终的结果，因此我们可以将若干个数码进行合并。

过程

还是以上面这个例子为例，如果我们每两位拆分一个数，我们可以拆分成 $(8100 + 92) \times 42$。

这样的拆分不影响最终结果，但是因为拆分出的数字变少了，计算效率也就提升了。

从 进位制 的角度理解这一过程，我们通过在较大的进位制（上面每两位拆分一个数，可以认为是在 100 进制下进行运算）下进行运算，从而达到减少参与运算的数字的位数，提升运算效率的目的。

这就是**压位高精度**的思想。

下面我们给出压位高精度的加法代码，用于进一步阐述其实现方法：

" 压位高精度加法参考实现"

```
// 这里的 a,b,c 数组均为 p 进制下的数
// 最终输出答案时需要将数字转为十进制
void add(int a[], int b[], int c[]) {
  clear(c);

  for (int i = 0; i < LEN - 1; ++i) {
    c[i] += a[i] + b[i];
    if (c[i] >= p) {  // 在普通高精度运算下，p=10
      c[i + 1] += 1;
      c[i] -= p;
    }
  }
}
```

压位高精下的高效竖式除法

在使用压位高精时，如果试商时仍然使用上文介绍的方法，由于试商次数会很多，计算常数会非常大。例如在万进制下，平均每个位需要试商 5000 次，这个巨大的常数是不可接受的。因此我们需要一个更高效的试商办法。

我们可以把 double 作为媒介。假设被除数有 4 位，是 a_4, a_3, a_2, a_1，除数有 3 位，是 b_3, b_2, b_1，那么我们只要试一位的商：使用 $base$ 进制，用式子 $\dfrac{a_4 base + a_3}{b_3 + b_2 base^{-1} + (b_1 + 1) base^{-2}}$ 来估商。而对于多个位的情况，就是一位的写法加个循环。由于除数使用 3 位的精度来参与估商，能保证估的商 q' 与实际商 q 的关系满足 $q - 1 \le q' \le q$，这样每个位在最坏的情况下也只需要两次试商。但与此同时要求 $base^3$ 在 double 的有效精度内，即 $base^3 < 2^{53}$，所以在运用这个方法时建议不要超过 32768 进制，否则很容易因精度不足产生误差从而导致错误。

另外，由于估的商总是小于等于实际商，所以还有再进一步优化的空间。绝大多数情况下每个位只估商一次，这样在下一个位估商时，虽然得到的商有可能因为前一位的误差造成试商结果大于等于 base，但这没有关系，只要在最后再最后做统一进位便可。举个例子，假设 base 是 10，求 395081/9876，试商计算步骤如下：

1. 首先试商计算得到 $3950/988 = 3$，于是 $395081 - (9876 \times 3 \times 10^1) = 98801$，这一步出现了误差，但不用管，继续下一步计算。

2. 对余数 98801 继续试商计算得到 $9880/988 = 10$，于是 $98801 - (9876 \times 10 \times 10^0) = 41$，这就是最终余数。
3. 把试商过程的结果加起来并处理进位，即 $3 \times 10^1 + 10 \times 10^0 = 40$ 便是准确的商。

方法虽然看着简单，但具体实现上很容易进坑，所以以下提供一个经过多番验证确认没有问题的实现供大家参考，要注意的细节也写在注释当中。

" 压位高精度高效竖式除法参考实现"

```
// 完整模板和实现 https://baobaobear.github.io/post/20210228-bigint1/
// 对 b 乘以 mul 再左移 offset 的结果相减，为除法服务
BigIntSimple &sub_mul(const BigIntSimple &b, int mul, int offset) {
  if (mul == 0) return *this;
  int borrow = 0;
  // 与减法不同的是，borrow 可能很大，不能使用减法的写法
  for (size_t i = 0; i < b.v.size(); ++i) {
    borrow += v[i + offset] - b.v[i] * mul - BIGINT_BASE + 1;
    v[i + offset] = borrow % BIGINT_BASE + BIGINT_BASE - 1;
    borrow /= BIGINT_BASE;
  }
  // 如果还有借位就继续处理
  for (size_t i = b.v.size(); borrow; ++i) {
    borrow += v[i + offset] - BIGINT_BASE + 1;
    v[i + offset] = borrow % BIGINT_BASE + BIGINT_BASE - 1;
    borrow /= BIGINT_BASE;
  }
  return *this;
}

BigIntSimple div_mod(const BigIntSimple &b, BigIntSimple &r) const {
  BigIntSimple d;
  r = *this;
  if (absless(b)) return d;
  d.v.resize(v.size() - b.v.size() + 1);
  // 提前算好除数的最高三位 +1 的倒数，若最高三位是 a3,a2,a1
  // 那么 db 是 a3+a2/base+(a1+1)/base^2 的倒数，最后用乘法估商的每一位
  // 此法在 BIGINT_BASE<=32768 时可在 int32 范围内用
  // 但即使使用 int64，那么也只有 BIGINT_BASE<=131072 时可用（受 double 的精度限制）
  // 能保证估计结果 q'与实际结果 q 的关系满足 q'<=q<=q'+1
  // 所以每一位的试商平均只需要一次，只要后面再统一处理进位即可
  // 如果要使用更大的 base，那么需要更换其它试商方案
  double t = (b.get((unsigned)b.v.size() - 2) +
              (b.get((unsigned)b.v.size() - 3) + 1.0) / BIGINT_BASE);
  double db = 1.0 / (b.v.back() + t / BIGINT_BASE);
  for (size_t i = v.size() - 1, j = d.v.size() - 1; j <= v.size();) {
    int rm = r.get(i + 1) * BIGINT_BASE + r.get(i);
    int m = std::max((int)(db * rm), r.get(i + 1));
    r.sub_mul(b, m, j);
    d.v[j] += m;
    if (!r.get(i + 1))  // 检查最高位是否已为 0，避免极端情况
      --i, --j;
  }
  r.trim();
  // 修正结果的个位
  int carry = 0;
```

```
  while (!r.absless(b)) {
    r.subtract(b);
    ++carry;
  }
  // 修正每一位的进位
  for (size_t i = 0; i < d.v.size(); ++i) {
    carry += d.v[i];
    d.v[i] = carry % BIGINT_BASE;
    carry /= BIGINT_BASE;
  }
  d.trim();
  d.sign = sign * b.sign;
  return d;
}

BigIntSimple operator/(const BigIntSimple &b) const {
  BigIntSimple r;
  return div_mod(b, r);
}

BigIntSimple operator%(const BigIntSimple &b) const {
  BigIntSimple r;
  div_mod(b, r);
  return r;
}
```

Karatsuba 乘法

记高精度数字的位数为 n，那么高精度—高精度竖式乘法需要花费 $O(n^2)$ 的时间。本节介绍一个时间复杂度更为优秀的算法，由前苏联（俄罗斯）数学家 Anatoly Karatsuba 提出，是一种分治算法。

考虑两个十进制大整数 x 和 y，均包含 n 个数码（可以有前导零）。任取 $0 < m < n$，记

$$\begin{aligned}x &= x_1 \cdot 10^m + x_0,\\ y &= y_1 \cdot 10^m + y_0,\\ x \cdot y &= z_2 \cdot 10^{2m} + z_1 \cdot 10^m + z_0,\end{aligned}$$

其中 $x_0, y_0, z_0, z_1 < 10^m$。可得

$$\begin{aligned}z_2 &= x_1 \cdot y_1,\\ z_1 &= x_1 \cdot y_0 + x_0 \cdot y_1,\\ z_0 &= x_0 \cdot y_0.\end{aligned}$$

观察知

$$z_1 = (x_1 + x_0) \cdot (y_1 + y_0) - z_2 - z_0,$$

于是要计算 z_1，只需计算 $(x_1 + x_0) \cdot (y_1 + y_0)$，再与 z_0、z_2 相减即可。

上式实际上是 Karatsuba 算法的核心，它将长度为 n 的乘法问题转化为了 3 个长度更小的子问题。若令 $m = \left\lceil \frac{n}{2} \right\rceil$，记 Karatsuba 算法计算两个 n 位整数乘法的耗时为 $T(n)$，则有 $T(n) = 3 \cdot T\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n)$，由主定理可得 $T(n) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$。

整个过程可以递归实现。为清晰起见，下面的代码通过 Karatsuba 算法实现了多项式乘法，最后再处理所有的进位问题。

"karatsuba_mulc.cpp"

```cpp
int *karatsuba_polymul(int n, int *a, int *b) {
  if (n <= 32) {
    // 规模较小时直接计算，避免继续递归带来的效率损失
    int *r = new int[n * 2 + 1]();
    for (int i = 0; i <= n; ++i)
      for (int j = 0; j <= n; ++j) r[i + j] += a[i] * b[j];
    return r;
  }

  int m = n / 2 + 1;
  int *r = new int[m * 4 + 1]();
  int *z0, *z1, *z2;

  z0 = karatsuba_polymul(m - 1, a, b);
  z2 = karatsuba_polymul(n - m, a + m, b + m);

  // 计算 z1
  // 临时更改，计算完毕后恢复
  for (int i = 0; i + m <= n; ++i) a[i] += a[i + m];
  for (int i = 0; i + m <= n; ++i) b[i] += b[i + m];
  z1 = karatsuba_polymul(m - 1, a, b);
  for (int i = 0; i + m <= n; ++i) a[i] -= a[i + m];
  for (int i = 0; i + m <= n; ++i) b[i] -= b[i + m];
  for (int i = 0; i <= (m - 1) * 2; ++i) z1[i] -= z0[i];
  for (int i = 0; i <= (n - m) * 2; ++i) z1[i] -= z2[i];

  // 由 z0、z1、z2 组合获得结果
  for (int i = 0; i <= (m - 1) * 2; ++i) r[i] += z0[i];
  for (int i = 0; i <= (m - 1) * 2; ++i) r[i + m] += z1[i];
  for (int i = 0; i <= (n - m) * 2; ++i) r[i + m * 2] += z2[i];

  delete[] z0;
  delete[] z1;
  delete[] z2;
  return r;
}

void karatsuba_mul(int a[], int b[], int c[]) {
  int *r = karatsuba_polymul(LEN - 1, a, b);
  memcpy(c, r, sizeof(int) * LEN);
  for (int i = 0; i < LEN - 1; ++i)
    if (c[i] >= 10) {
      c[i + 1] += c[i] / 10;
      c[i] %= 10;
    }
  delete[] r;
}
```

" 关于 new 和 delete"

见 内存池。

但是这样的实现存在一个问题：在 b 进制下，多项式的每一个系数都有可能达到 $n \cdot b^2$ 量级，在压位高精度实现中可能造成整数溢出；而若在多项式乘法的过程中处理进位问题，则 $x_1 + x_0$ 与 $y_1 + y_0$ 的结果可能达到 $2 \cdot b^m$，增加

一个位（如果采用 $x_1 - x_0$ 的计算方式，则不得不特殊处理负数的情况）。因此，需要依照实际的应用场景来决定采用何种实现方式。

基于多项式的高效大整数乘法

如果数据规模达到了 10^{10^5} 或更大，普通的高精度乘法可能会超时。本节将介绍用多项式优化此类乘法的方法。

对于一个 n 位的十进制整数 a，可以将它看作一个每位系数均为整数且不超过 10 的多项式 $A = a_0 10^0 + a_1 10^1 + \cdots + a_{n-1} 10^{n-1}$。这样，我们就将两个整数乘法转化为了两个多项式乘法。

普通的多项式乘法时间复杂度仍是 $O(n^2)$，但可以用多项式一节中的 快速傅里叶变换、快速数论变换 等算法优化，优化后的时间复杂度是 $O(n \log n)$。

封装类

这里[1] 有一个封装好的高精度整数类，以及 这里[2] 支持动态长度及四则运算的超迷你实现类。

这里是另一个模板

```
#define MAXN 9999
// MAXN 是一位中最大的数字
#define MAXSIZE 10024
// MAXSIZE 是位数
#define DLEN 4

// DLEN 记录压几位
struct Big {
  int a[MAXSIZE], len;
  bool flag;  // 标记符号'-'

  Big() {
    len = 1;
    memset(a, 0, sizeof a);
    flag = 0;
  }

  Big(const int);
  Big(const char*);
  Big(const Big&);
  Big& operator=(const Big&);
  Big operator+(const Big&) const;
  Big operator-(const Big&) const;
  Big operator*(const Big&) const;
  Big operator/(const int&) const;
  // TODO: Big / Big;
  Big operator^(const int&) const;
  // TODO: Big ^ Big;

  // TODO: Big 位运算;

  int operator%(const int&) const;
  // TODO: Big ^ Big;
  bool operator<(const Big&) const;
  bool operator<(const int& t) const;
```

```
  void print() const;
};

Big::Big(const int b) {
  int c, d = b;
  len = 0;
  // memset(a,0,sizeof a);
  CLR(a);
  while (d > MAXN) {
    c = d - (d / (MAXN + 1) * (MAXN + 1));
    d = d / (MAXN + 1);
    a[len++] = c;
  }
  a[len++] = d;
}

Big::Big(const char* s) {
  int t, k, index, l;
  CLR(a);
  l = strlen(s);
  len = l / DLEN;
  if (l % DLEN) ++len;
  index = 0;
  for (int i = l - 1; i >= 0; i -= DLEN) {
    t = 0;
    k = i - DLEN + 1;
    if (k < 0) k = 0;
    g(j, k, i) t = t * 10 + s[j] - '0';
    a[index++] = t;
  }
}

Big::Big(const Big& T) : len(T.len) {
  CLR(a);
  f(i, 0, len) a[i] = T.a[i];
  // TODO: 重载此处？
}

Big& Big::operator=(const Big& T) {
  CLR(a);
  len = T.len;
  f(i, 0, len) a[i] = T.a[i];
  return *this;
}

Big Big::operator+(const Big& T) const {
  Big t(*this);
  int big = len;
  if (T.len > len) big = T.len;
  f(i, 0, big) {
    t.a[i] += T.a[i];
    if (t.a[i] > MAXN) {
      ++t.a[i + 1];
      t.a[i] -= MAXN + 1;
```

```
    }
  }
  if (t.a[big])
    t.len = big + 1;
  else
    t.len = big;
  return t;
}

Big Big::operator-(const Big& T) const {
  int big;
  bool ctf;
  Big t1, t2;
  if (*this < T) {
    t1 = T;
    t2 = *this;
    ctf = 1;
  } else {
    t1 = *this;
    t2 = T;
    ctf = 0;
  }
  big = t1.len;
  int j = 0;
  f(i, 0, big) {
    if (t1.a[i] < t2.a[i]) {
      j = i + 1;
      while (t1.a[j] == 0) ++j;
      --t1.a[j--];
      // WTF?
      while (j > i) t1.a[j--] += MAXN;
      t1.a[i] += MAXN + 1 - t2.a[i];
    } else
      t1.a[i] -= t2.a[i];
  }
  t1.len = big;
  while (t1.len > 1 && t1.a[t1.len - 1] == 0) {
    --t1.len;
    --big;
  }
  if (ctf) t1.a[big - 1] = -t1.a[big - 1];
  return t1;
}

Big Big::operator*(const Big& T) const {
  Big res;
  int up;
  int te, tee;
  f(i, 0, len) {
    up = 0;
    f(j, 0, T.len) {
      te = a[i] * T.a[j] + res.a[i + j] + up;
      if (te > MAXN) {
        tee = te - te / (MAXN + 1) * (MAXN + 1);
```

```
        up = te / (MAXN + 1);
        res.a[i + j] = tee;
      } else {
        up = 0;
        res.a[i + j] = te;
      }
    }
    if (up) res.a[i + T.len] = up;
  }
  res.len = len + T.len;
  while (res.len > 1 && res.a[res.len - 1] == 0) --res.len;
  return res;
}

Big Big::operator/(const int& b) const {
  Big res;
  int down = 0;
  gd(i, len - 1, 0) {
    res.a[i] = (a[i] + down * (MAXN + 1)) / b;
    down = a[i] + down * (MAXN + 1) - res.a[i] * b;
  }
  res.len = len;
  while (res.len > 1 && res.a[res.len - 1] == 0) --res.len;
  return res;
}

int Big::operator%(const int& b) const {
  int d = 0;
  gd(i, len - 1, 0) d = (d * (MAXN + 1) % b + a[i]) % b;
  return d;
}

Big Big::operator^(const int& n) const {
  Big t(n), res(1);
  int y = n;
  while (y) {
    if (y & 1) res = res * t;
    t = t * t;
    y >>= 1;
  }
  return res;
}

bool Big::operator<(const Big& T) const {
  int ln;
  if (len < T.len) return 233;
  if (len == T.len) {
    ln = len - 1;
    while (ln >= 0 && a[ln] == T.a[ln]) --ln;
    if (ln >= 0 && a[ln] < T.a[ln]) return 233;
    return 0;
  }
  return 0;
}
```

```
bool Big::operator<(const int& t) const {
  Big tee(t);
  return *this < tee;
}

void Big::print() const {
  printf("%d", a[len - 1]);
  gd(i, len - 2, 0) { printf("%04d", a[i]); }
}

void print(const Big& s) {
  int len = s.len;
  printf("%d", s.a[len - 1]);
  gd(i, len - 2, 0) { printf("%04d", s.a[i]); }
}

char s[100024];
```

习题

- NOIP 2012 国王游戏[3]
- SPOJ - Fast Multiplication[4]
- SPOJ - GCD2[5]
- UVa - Division[6]
- UVa - Fibonacci Freeze[7]
- Codeforces - Notepad[8]

参考资料与链接

1. Karatsuba algorithm - Wikipedia[9]

参考资料与注释

[1] 这里

[2] 这里

[3] NOIP 2012 国王游戏

[4] SPOJ - Fast Multiplication

[5] SPOJ - GCD2

[6] UVa - Division

[7] UVa - Fibonacci Freeze

[8] Codeforces - Notepad

[9] Karatsuba algorithm - Wikipedia

9.8 快速幂

autor: iamtwz, billchenchina, CBW2007, CCXXXI, chinggg, Enter-tainer, eyedeng, FFjet, gaojude, Great-designer, H-J-Granger, Henry-ZHR, hsfzLZH1, Ir1d, kenlig, Konano, ksyx, luoguyuntianming, Marcythm, Menci, NachtgeistW, ouuan, Peanut-Tang, qwqAutomaton, sshwy, StudyingFather, Tiphereth-A, TrisolarisHD, TRSWNCA, Xeonacid, Yuuko10032, Zhangjiacheng2006, Zhoier, Hszzzx, shenshuaijie, kfy666

定义

快速幂，二进制取幂（Binary Exponentiation，也称平方法），是一个在 $\Theta(\log n)$ 的时间内计算 a^n 的小技巧，而暴力的计算需要 $\Theta(n)$ 的时间。

这个技巧也常常用在非计算的场景，因为它可以应用在任何具有结合律的运算中。其中显然的是它可以应用于模意义下取幂、矩阵幂等运算，我们接下来会讨论。

解释

计算 a 的 n 次方表示将 n 个 a 乘在一起：$a^n = \underbrace{a \times a \cdots \times a}_{n \text{ 个 a}}$。然而当 a, n 太大的时侯，这种方法就不太适用了。不过我们知道：$a^{b+c} = a^b \cdot a^c$，$a^{2b} = a^b \cdot a^b = (a^b)^2$。二进制取幂的想法是，我们将取幂的任务按照指数的**二进制表示**来分割成更小的任务。

过程

迭代版本

首先我们将 n 表示为 2 进制，举一个例子：

$$3^{13} = 3^{(1101)_2} = 3^8 \cdot 3^4 \cdot 3^1$$

因为 n 有 $\lfloor \log_2 n \rfloor + 1$ 个二进制位，因此当我们知道了 $a^1, a^2, a^4, a^8, \dots, a^{2^{\lfloor \log_2 n \rfloor}}$ 后，我们只用计算 $\Theta(\log n)$ 次乘法就可以计算出 a^n。

于是我们只需要知道一个快速的方法来计算上述 3 的 2^k 次幂的序列。这个问题很简单，因为序列中（除第一个）任意一个元素就是其前一个元素的平方。举一个例子：

$$3^1 = 3 \tag{9.1}$$
$$3^2 = \left(3^1\right)^2 = 3^2 = 9 \tag{9.2}$$
$$3^4 = \left(3^2\right)^2 = 9^2 = 81 \tag{9.3}$$
$$3^8 = \left(3^4\right)^2 = 81^2 = 6561 \tag{9.4}$$

因此为了计算 3^{13}，我们只需要将对应二进制位为 1 的整系数幂乘起来就行了：

$$3^{13} = 6561 \cdot 81 \cdot 3 = 1594323$$

将上述过程说得形式化一些，如果把 n 写作二进制为 $(n_t n_{t-1} \cdots n_1 n_0)_2$，那么有：

$$n = n_t 2^t + n_{t-1} 2^{t-1} + n_{t-2} 2^{t-2} + \cdots + n_1 2^1 + n_0 2^0$$

其中 $n_i \in \{0,1\}$。那么就有

$$
\begin{aligned}
a^n &= (a^{n_t 2^t + \cdots + n_0 2^0}) \\
&= a^{n_0 2^0} \times a^{n_1 2^1} \times \cdots \times a^{n_t 2^t}
\end{aligned}
$$

根据上式我们发现，原问题被我们转化成了形式相同的子问题的乘积，并且我们可以在常数时间内从 2^i 项推出 2^{i+1} 项。

这个算法的复杂度是 $\Theta(\log n)$ 的，我们计算了 $\Theta(\log n)$ 个 2^k 次幂的数，然后花费 $\Theta(\log n)$ 的时间选择二进制为 1 对应的幂来相乘。

递归版本

上述迭代版本中，由于 2^{i+1} 项依赖于 2^i，使得其转换为递归版本比较困难（一方面需要返回一个额外的 a^{2^i}，对函数来说无法实现一个只返回计算结果的接口；另一方面则是必须从低位往高位计算，即从高位往低位调用，这也造成了递归实现的困扰），下面则提供递归版本的思路。

给定形式 $n_{t\dots i} = (n_t n_{t-1} \cdots n_i)_2$，即 $n_{t\dots i}$ 表示将 n 的前 $t-i+1$ 位二进制位当作一个二进制数，则有如下变换：

$$
\begin{aligned}
n &= n_{t\dots 0} \\
&= 2 \times n_{t\dots 1} + n_0 \\
&= 2 \times (2 \times n_{t\dots 2} + n_1) + n_0 \\
&\cdots
\end{aligned}
$$

那么有：

$$
\begin{aligned}
a^n &= a^{n_{t\dots 0}} \\
&= a^{2 \times n_{t\dots 1} + n_0} = \left(a^{n_{t\dots 1}}\right)^2 a^{n_0} \\
&= \left(a^{2 \times n_{t\dots 2} + n_1}\right)^2 a^{n_0} = \left(\left(a^{n_{t\dots 2}}\right)^2 a^{n_1}\right)^2 a^{n_0} \\
&\cdots
\end{aligned}
$$

如上所述，在递归时，对于不同的递归深度是相同的处理：$a^{n_{t\dots i}} = (a^{n_{t\dots (i+1)}})^2 a^{n_i}$，即将当前递归的二进制数拆成两部分：最低位在递归出来时乘上去，其余部分则变成新的二进制数递归进入更深一层作相同的处理。

可以观察到，每递归深入一层则二进制位减少一位，所以该算法的时间复杂度也为 $\Theta(\log n)$。

实现

首先我们可以直接按照上述递归方法实现：

```cpp
long long binpow(long long a, long long b) {
  if (b == 0) return 1;
  long long res = binpow(a, b / 2);
  if (b % 2)
    return res * res * a;
  else
    return res * res;
}
```

```python
def binpow(a, b):
    if b == 0:
        return 1
    res = binpow(a, b // 2)
    if (b % 2) == 1:
        return res * res * a
```

```
    else:
        return res * res
```

第二种实现方法是非递归式的。它在循环的过程中将二进制位为 1 时对应的幂累乘到答案中。尽管两者的理论复杂度是相同的，但第二种在实践过程中的速度是比第一种更快的，因为递归会花费一定的开销。

```
long long binpow(long long a, long long b) {
  long long res = 1;
  while (b > 0) {
    if (b & 1) res = res * a;
    a = a * a;
    b >>= 1;
  }
  return res;
}
```

```
def binpow(a, b):
    res = 1
    while b > 0:
        if (b & 1):
            res = res * a
        a = a * a
        b >>= 1
    return res
```

模板：Luogu P1226[2]

应用

模意义下取幂

" 问题描述"

计算 $x^n \bmod m$。

这是一个非常常见的应用，例如它可以用于计算模意义下的乘法逆元。

既然我们知道取模的运算不会干涉乘法运算，因此我们只需要在计算的过程中取模即可。

```
long long binpow(long long a, long long b, long long m) {
  a %= m;
  long long res = 1;
  while (b > 0) {
    if (b & 1) res = res * a % m;
    a = a * a % m;
    b >>= 1;
  }
  return res;
}
```

```
def binpow(a, b, m):
    a = a % m
    res = 1
    while b > 0:
        if (b & 1):
            res = res * a % m
```

```
        a = a * a % m
        b >>= 1
    return res
```

注意：根据费马小定理，如果 m 是一个质数，我们可以计算 $x^{n \bmod (m-1)}$ 来加速算法过程。

计算斐波那契数

" 问题描述"

计算斐波那契数列第 n 项 F_n。

根据斐波那契数列的递推式 $F_n = F_{n-1} + F_{n-2}$，我们可以构建一个 2×2 的矩阵来表示从 F_i, F_{i+1} 到 F_{i+1}, F_{i+2} 的变换。于是在计算这个矩阵的 n 次幂的时侯，我们使用快速幂的思想，可以在 $\Theta(\log n)$ 的时间内计算出结果。对于更多的细节参见 斐波那契数列，矩阵快速幂的实现参见 矩阵加速递推 中的实现。

多次置换

" 问题描述"

给你一个长度为 n 的序列和一个置换，把这个序列置换 k 次。

简单地把这个置换取 k 次幂，然后把它应用到序列 n 上即可。时间复杂度是 $O(n \log k)$ 的。

注意：给这个置换建图，然后在每一个环上分别做 k 次幂（事实上做一下 k 对环长取模的运算即可）可以取得更高效的算法，达到 $O(n)$ 的复杂度。

加速几何中对点集的操作

引入

> 三维空间中，n 个点 p_i，要求将 m 个操作都应用于这些点。包含 3 种操作：
>
> 1. 沿某个向量移动点的位置（Shift）。
> 2. 按比例缩放这个点的坐标（Scale）。
> 3. 绕某个坐标轴旋转（Rotate）。
>
> 还有一个特殊的操作，就是将一个操作序列重复 k 次（Loop），这个序列中也可能有 Loop 操作（Loop 操作可以嵌套）。现在要求你在低于 $O(n \cdot length)$ 的时间内将这些变换应用到这个 n 个点，其中 $length$ 表示把所有的 Loop 操作展开后的操作序列的长度。

解释

让我们来观察一下这三种操作对坐标的影响：

1. Shift 操作：将每一维的坐标分别加上一个常量；
2. Scale 操作：把每一维坐标分别乘上一个常量；
3. Rotate 操作：这个有点复杂，我们不打算深入探究，不过我们仍然可以使用一个线性组合来表示新的坐标。

可以看到，每一个变换可以被表示为对坐标的线性运算，因此，一个变换可以用一个 4×4 的矩阵来表示：

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

使用这个矩阵就可以将一个坐标（向量）进行变换，得到新的坐标（向量）：

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}$$

你可能会问，为什么一个三维坐标会多一个 1 出来？原因在于，如果没有这个多出来的 1，我们没法使用矩阵的线性变换来描述 Shift 操作。

过程

接下来举一些简单的例子来说明我们的思路：

1. Shift 操作：让 x 坐标方向的位移为 5，y 坐标的位移为 7，z 坐标的位移为 9：

$$\begin{bmatrix} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 7 \\ 0 & 0 & 1 & 9 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2. Scale 操作：把 x 坐标拉伸 10 倍，y, z 坐标拉伸 5 倍：

$$\begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3. Rotate 操作：绕 x 轴旋转 θ 弧度，遵循右手定则（逆时针方向）

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

现在，每一种操作都被表示为了一个矩阵，变换序列可以用矩阵的乘积来表示，而一个 Loop 操作相当于取一个矩阵的 k 次幂。这样可以用 $O(m \log k)$ 计算出整个变换序列最终形成的矩阵。最后将它应用到 n 个点上，总复杂度 $O(n + m \log k)$。

定长路径计数

" 问题描述"

给一个有向图（边权为 1），求任意两点 u, v 间从 u 到 v，长度为 k 的路径的条数。

我们把该图的邻接矩阵 M 取 k 次幂，那么 $M_{i,j}$ 就表示从 i 到 j 长度为 k 的路径的数目。该算法的复杂度是 $O(n^3 \log k)$。有关该算法的细节请参见 矩阵 页面。

模意义下大整数乘法

> 计算 $a \times b \bmod m$，$a, b \le m \le 10^{18}$。

与二进制取幂的思想一样，这次我们将其中的一个乘数表示为若干个 2 的整数次幂的和的形式。因为在对一个数做乘 2 并取模的运算的时候，我们可以转化为加减操作防止溢出。这样仍可以在 $O(\log_2 m)$ 的时内解决问题。递归方

法如下：

$$a \cdot b = \begin{cases} 0 & \text{if } a = 0 \\ 2 \cdot \frac{a}{2} \cdot b & \text{if } a > 0 \text{ and } a \text{ even} \\ 2 \cdot \frac{a-1}{2} \cdot b + b & \text{if } a > 0 \text{ and } a \text{ odd} \end{cases}$$

快速乘

但是 $O(\log_2 m)$ 的「龟速乘」还是太慢了，这在很多对常数要求比较高的算法比如 Miller_Rabin 和 Pollard-Rho 中，就显得不够用了。所以我们要介绍一种可以处理模数在 `long long` 范围内、不需要使用黑科技 `__int128` 的、复杂度为 $O(1)$ 的「快速乘」。

我们发现：

$$a \times b \bmod m = a \times b - \left\lfloor \frac{ab}{m} \right\rfloor \times m$$

我们巧妙运用 `unsigned long long` 的自然溢出：

$$a \times b \bmod m = a \times b - \left\lfloor \frac{ab}{m} \right\rfloor \times m = \left(a \times b - \left\lfloor \frac{ab}{m} \right\rfloor \times m \right) \bmod 2^{64}$$

于是在算出 $\left\lfloor \frac{ab}{m} \right\rfloor$ 后，两边的乘法和中间的减法部分都可以使用 `unsigned long long` 直接计算，现在我们只需要解决如何计算 $\left\lfloor \frac{ab}{m} \right\rfloor$。

我们考虑先使用 `long double` 算出 $\frac{a}{m}$ 再乘上 b。

既然使用了 `long double`，就无疑会有精度误差。极端情况就是第一个有效数字（二进制下）在小数点后一位。在 `x86-64` 机器下，`long double` 将被解释成 80 位拓展小数（即符号为 1 位，指数为 15 位，尾数为 64 位），所以 `long double` 最多能精确表示的有效位数为 64[1]。所以 $\frac{a}{m}$ 最差从第 65 位开始出错，误差范围为 $(-2^{-64}, 2^{64})$。乘上 b 这个 64 位整数，误差范围为 $(-0.5, 0.5)$，再加上 0.5 误差范围为 $(0, 1)$，取整后误差范围位 $\{0, 1\}$。于是乘上 $-m$ 后，误差范围变成 $\{0, -m\}$，我们需要判断这两种情况。

因为 m 在 `long long` 范围内，所以如果计算结果 r 在 $[0, m)$ 时，直接返回 r，否则返回 $r + m$，当然你也可以直接返回 $(r + m) \bmod m$。

代码实现如下：

```cpp
long long binmul(long long a, long long b, long long m) {
  unsigned long long c =
      (unsigned long long)a * b -
      (unsigned long long)((long double)a / m * b + 0.5L) * m;
  if (c < m) return c;
  return c + m;
}
```

高精度快速幂

" 前置技能"

请先学习 高精度

" 例题【NOIP2003 普及组改编 · 麦森数】（原题在此[3]）"

题目大意：从文件中输入 P（$1000 < P < 3100000$），计算 $2^P - 1$ 的最后 100 位数字（用十进制高精度数表示），不足 100 位时高位补 0。

代码实现如下：

```cpp
#include <bits/stdc++.h>
using namespace std;
int a[505], b[505], t[505], i, j;

void mult(int x[], int y[])  // 高精度乘法
{
  memset(t, 0, sizeof(t));
  for (i = 1; i <= x[0]; i++) {
    for (j = 1; j <= y[0]; j++) {
      if (i + j - 1 > 100) continue;
      t[i + j - 1] += x[i] * y[j];
      t[i + j] += t[i + j - 1] / 10;
      t[i + j - 1] %= 10;
      t[0] = i + j;
    }
  }
  memcpy(b, t, sizeof(b));
}

void ksm(int p)  // 快速幂
{
  if (p == 1) {
    memcpy(b, a, sizeof(b));
    return;
  }
  ksm(p / 2);  //(2^(p/2))^2=2^p
  mult(b, b);  // 对 b 平方
  if (p % 2 == 1) mult(b, a);
}

int main() {
  int p;
  scanf("%d", &p);
  a[0] = 1;  // 记录 a 数组的位数
  a[1] = 2;  // 对 2 进行平方
  b[0] = 1;  // 记录 b 数组的位数
  b[1] = 1;  // 答案数组
  ksm(p);
  for (i = 100; i >= 1; i--) {
    if (i == 1) {
      printf("%d\n", b[i] - 1);  // 最后一位减 1
    } else
      printf("%d", b[i]);
  }
}
```

同一底数与同一模数的预处理快速幂

在同一底数与同一模数的条件下，可以利用分块思想，用一定的时间（一般是 $O(\sqrt{n})$）预处理后用 $O(1)$ 的时间回答一次幂询问。

过程

1. 选定一个数 s，预处理出 a^0 到 a^s 与 $a^{0\cdot s}$ 到 $a^{\lceil \frac{p}{s} \rceil \cdot s}$ 的值并存在一个（或两个）数组里；
2. 对于每一次询问 $a^b \bmod p$，将 b 拆分成 $\left\lfloor \dfrac{b}{s} \right\rfloor \cdot s + b \bmod s$，则 $a^b = a^{\lfloor \frac{b}{s} \rfloor \cdot s} \times a^{b \bmod s}$，可以 $O(1)$ 求出答案。

关于这个数 s 的选择，我们一般选择 $\sqrt{p}$ 或者一个大小适当的 2 的次幂（选择 $\sqrt{p}$ 可以使预处理较优，选择 2 的次幂可以使用位运算优化/简化计算）。

" 参考代码"

```
int pow1[65536], pow2[65536];

void preproc(int a, int mod) {
  pow1[0] = pow2[0] = 1;
  for (int i = 1; i < 65536; i++) pow1[i] = 1LL * pow1[i - 1] * a % mod;
  int pow65536 = 1LL * pow1[65535] * a % mod;
  for (int i = 1; i < 65536; i++) pow2[i] = 1LL * pow2[i - 1] * pow65536 % mod;
}

int query(int pows) { return 1LL * pow1[pows & 65535] * pow2[pows >> 16]; }
```

习题

- UVa 1230 - MODEX[4]
- UVa 374 - Big Mod[5]
- UVa 11029 - Leading and Trailing[6]
- Codeforces - Parking Lot[7]
- SPOJ - The last digit[8]
- SPOJ - Locker[9]
- LA - 3722 Jewel-eating Monsters[10]
- SPOJ - Just add it[11]

参考资料与注释

[1] 参见 C 语言小数表示法 - 维基百科

[2] Luogu P1226

[3] 原题在此

[4] UVa 1230 - MODEX

[5] UVa 374 - Big Mod

[6] UVa 11029 - Leading and Trailing

[7] Codeforces - Parking Lot

[8] SPOJ - The last digit

[9] SPOJ - Locker

[10] LA - 3722 Jewel-eating Monsters

[11] SPOJ - Just add it

[12]

[13] Binary Exponentiation

9.9 置换和排列

置换

一个有限集合 S 到自身的双射（即一一对应）称为 S 的一个置换。集合 $S=\{a_1,a_2,\dots,a_n\}$ 上的置换可以表示为

$$f=\begin{pmatrix} a_1,a_2,\dots,a_n \\ a_{p_1},a_{p_2},\dots,a_{p_n} \end{pmatrix}$$

意为将 a_i 映射为 a_{p_i}，其中 $p_1,p_2,\dots,p_n$ 是 $1,2,\dots,n$ 的一个排列。显然 S 上所有置换的数量为 $n!$。

乘法

对于两个置换 $f=\begin{pmatrix} a_1,a_2,\dots,a_n \\ a_{p_1},a_{p_2},\dots,a_{p_n} \end{pmatrix}$ 和 $g=\begin{pmatrix} a_{p_1},a_{p_2},\dots,a_{p_n} \\ a_{q_1},a_{q_2},\dots,a_{q_n} \end{pmatrix}$，$f$ 和 g 的乘积记为 $g\circ f$，其值为

$$g\circ f=\begin{pmatrix} a_1,a_2,\dots,a_n \\ a_{q_1},a_{q_2},\dots,a_{q_n} \end{pmatrix}$$

简单来说就是先经过 f 的映射，再经过 g 的映射。

排列

设 $I=\{1,2,\cdots,n\}$ 是前 n 个正整数构成的集合，σ 是 I 的一个置换。记：

$$\sigma(1)=i_1 \quad \sigma(2)=i_2 \quad \cdots \quad \sigma(n)=i_n$$

于是 $i_1,i_2,\cdots,i_n$ 仍然为 $1,2,\cdots,n$ 这 n 个数，只是顺序有所不同。

由 $1,2,\cdots,n$ 组成的一个有序组，称为 $1,2,\cdots,n$ 的一个排列。例如把前文 $i_1,i_2,\cdots,i_n$ 叫做 $1,2,\cdots,n$ 的一个排列。

前 n 个正整数 $1,2,\cdots,n$ 的不同排列共有 $n!$ 个。

逆序和逆序数

在一个排列中，如果某一个较大的数排在某一个较小的数前面，就说这两个数构成一个反序或逆序。

在一个排列里出现的反序的总个数，叫做这个排列的反序数或逆序数。

一个排列的反序数可能是偶数也可能是奇数。有偶数个反序的排列叫做一个偶排列，有奇数个反序的排列叫做一个奇排列。

对换

如果把 $1, 2, \cdots, n$ 的排列中，任意两个数 i 和 j 交换，其余数保持不动，就得到一个新排列。对于排列施加这样一个变换叫做一个对换，用 (i, j) 表示。

定理：设 $i_1 i_2 \cdots i_n$ 和 $j_1 j_2 \cdots j_n$ 是 n 个数码的任意两个排列，那么总可以通过一系列对换，由 $i_1 i_2 \cdots i_n$ 得出 $j_1 j_2 \cdots j_n$。

定理：每一个对换都改变排列的奇偶性。

定理：当 n 至少为 2 时，n 个数码的奇排列与偶排列个数相等，各为 $\frac{n!}{2}$ 个。

逆序数的计算方法

逆序数的编程计算方法，可以使用归并排序，时间复杂度为 $O(n \log n)$。

9.10 弧度制与坐标系

Authors: Ir1d, HeRaNO, Chrogeek, abc1763613206

角的定义

在小学或初中已经学习过角的**静态定义**：具有公共端点的两条射线组成的图形叫做角。

但是该定义将角度限制在了 $[0, 360^\circ]$，这给深入研究带来了一定的困难，还有其他的问题无法解释清，比如：旋转 720° 是什么意思?

在高中数学，讲了角的**动态定义**：平面内一条射线绕其端点从一个位置旋转到另一个位置形成的图形叫做角。

开始的位置称为**始边**，结束的位置称为**终边**。并规定：

- 按**逆时针**方向旋转形成的角叫做**正角**，其角度为正；
- 按**顺时针**方向旋转形成的角叫做**负角**，其角度为负；
- 终边相对于始边没有做任何旋转的角叫做**零角**，其角度为 0°。

这样就把角的概念推向了**任意角**。

" 注意"

零角始边和终边重合，但始边和终边重合的角并不都是零角，如以 360° 为倍数的角。

弧度制

实际应用中经常有角度到各种参数的转换，而使用弧度制描述角可以减少系数的使用。所以接下来，介绍**弧度制**：

把长度等于半径长的弧所对的圆心角称为 1 弧度的角，用符号 rad 表示，读作：弧度。

根据前面的规定，正角的弧度为正，负角的弧度为负，零角的弧度为 0，如果半径为 r 的圆的圆心角 α 所对弧长为 l，则：

$$|\alpha| = \frac{l}{r}$$

利用这个公式还可以写出弧长和扇形面积公式，在此略过。

于是，360° 角的弧度为 2π，这样有了对应关系之后就可以进行角度值和弧度制的转化了：

$$k\,\mathrm{rad} = \frac{\pi}{180^\circ} n^\circ$$

考虑一个角，将其终边再旋转一周，甚至多周，始边位置不动，那么终边位置永远是相同的，称这些角为终边位置相同的角。

与角 α 终边位置相同的角的集合很容易得出，为 $\{\varphi \mid \varphi = \alpha + 2k\pi, k \in \mathbf{Z}\}$。

可以理解为：给这个角的边不停加转一圈，终边位置不变。

"π 和 τ 两个数学常数"

目前西方数学界有一些观点认为，「真正的圆周率」应为 2π，将这个值记为希腊字母 τ。新圆周率的支持者们选择在 6 月 28 日庆祝「真正的」圆周率日。

比如，在弧度制下，一个周角是 2π，直接对 2π 进行等分可以得到周角的等分。又例如，在复变函数中频繁出现 2π 的组合，等等。

为了迎合中国各地区约定俗成的习惯，在 **OI Wiki**，采用参数 π 表示圆周率。

" 编程中圆周率的习惯写法"

在 C/C++ 语言中，一般取 π 为 `acos(-1)`，只有这个值是最接近 π 的浮点数。使用 `acos(-1)` 或者 `4 * atan(1)` 写出来的 π 是 3.14159265358979310000。

采用其他值，例如 `acos(-1.0/2.0)`，`acos(1.0/2.0)`，`asin(1.0/2.0)` 等等，写出来的 π 是 3.14159265358979360000，这就不是最接近 π 的浮点数了。

如果你背得下来，也可以直接写 3.1415926535897932。

平面直角坐标系

在同一个平面上互相垂直且有公共原点的两条数轴构成平面直角坐标系（Rectangular Coordinates）。

通常，两条数轴分别置于水平位置与垂直位置，取向右与向上的方向分别为两条数轴的正方向。水平的数轴叫做 x 轴（x-axis）或横轴，垂直的数轴叫做 y 轴（y-axis）或纵轴，x 轴 y 轴统称为坐标轴，它们的公共原点 O 称为平面直角坐标系的原点（origin），以点 O 为原点的平面直角坐标系记作平面直角坐标系 xOy。

x 轴 y 轴将坐标平面分成了四个象限（quadrant），右上方的部分叫做第一象限，其他三个部分按逆时针方向依次叫做第二象限、第三象限和第四象限。象限以数轴为界，横轴、纵轴上的点及原点不在任何一个象限内。一般情况下，x 轴 y 轴取相同的单位长度，但在特殊的情况下，也可以取不同的单位长度。

平面直角坐标系下位置的描述

在平面直角坐标系中，对于平面上的任意一点，都有唯一的一个有序数对（即点的坐标（coordinates））与它对应；反过来，对于任意一个有序数对，都有平面上唯一的一点与它对应。

对于平面内任意一点 C，过点 C 分别向 x 轴、y 轴作垂线，垂足在 x 轴、y 轴上的对应点 a,b 分别叫做点 C 的横坐标、纵坐标，有序数对（ordered pair）(a,b) 叫做点 C 的直角坐标。一个点在不同的象限或坐标轴上，其坐标都不一样。

平面极坐标系

考虑实际情况，比如航海，说「点 B 在点 A 的北偏东 30° 方向上，距离为 100 米」，而不是「以 A 为原点建立平面直角坐标系，$B(50, 50\sqrt{3})$」。

这样：

1. 在平面上选一定点 O，称为**极点**；

2. 自极点引出一条射线 Ox，称为**极轴**；
3. 选择一个单位长度（在数学问题中通常为 1），一个角度单位（通常为弧度）及其正方向（通常为逆时针方向）；

就建立了**极坐标系**。

极坐标系下位置的描述

设 A 为平面上一点。

- 极点 O 与 A 之间的距离 $|OA|$ 称为**极径**，记为 ρ；
- 以极轴为始边，OA 为终边的角 $\angle xOA$ 称为**极角**，记为 φ；

那么有序数对 (ρ,φ) 即为 A 的**极坐标**。

由终边相同的角的定义可知，(ρ,φ) 与 $(\rho,\varphi+2k\pi)$ $(k\in\mathbf{Z})$ 其实表示的是一样的点。特别地，极点的极坐标为 $(0,\varphi)$ $(\varphi\in\mathbf{R})$，于是平面内的点的极坐标表示有无数多种。

如果规定 $\rho\geq 0, 0\leq\varphi<2\pi$，那么除极点外，其他平面内的点可以用唯一有序数对 (ρ,φ) 表示，而极坐标 (ρ,φ) 表示的点是唯一确定的。

平面直角坐标系与极坐标系的相互转换

当然，有时候研究极坐标系下的图形有些不方便。要想转到直角坐标系下研究，有互化公式。点 $A(\rho,\varphi)$ 的直角坐标 (x,y) 可以如下表示：

$$x=\rho\cos\varphi$$
$$y=\rho\sin\varphi$$

进而可知：

$$\rho^2=x^2+y^2$$
$$\tan\varphi=\frac{y}{x}\qquad(x\neq 0)$$

于是有 $\rho=\sqrt{x^2+y^2}$。

但具有相同 $\dfrac{y}{x}$ 的 $\tan\varphi$ 有两个可能的 φ 的值，此时还需要根据 x,y 的值来确定方向。具体地，定义函数：

$$\operatorname{atan2}(y,x)=\begin{cases}\arctan(\frac{y}{x}) & \text{if } x>0\\ \arctan(\frac{y}{x})+\pi & \text{if } y\geq 0, x<0\\ \arctan(\frac{y}{x})-\pi & \text{if } y<0, x<0\\ \pi/2 & \text{if } y>0, x=0\\ -\pi/2 & \text{if } y<0, x=0\\ \text{any} & \text{if } y=0, x=0\end{cases}$$

则 $\varphi=\operatorname{atan2}(y,x)$。注意上述函数的值域为 $(-\pi,\pi]$。

在 C/C++ 语言的 `<math.h>` 或 `<cmath>` 库里定义了 该函数[1]，调用 `atan2(y, x)` 即可。

空间直角坐标系

使用如下方法建立空间直角坐标系：

1. 在空间内选定一点 O；
2. 过点 O 作三条互相垂直的数轴 $\overrightarrow{Ox},\overrightarrow{Oy},\overrightarrow{Oz}$，分别称作 x 轴（横轴），y 轴（纵轴），z 轴（竖轴），统称为坐标轴；它们的正方向符合右手规则，即以右手握住 z 轴，当右手的四个手指 x 轴的正向以角度转向 y 轴正向时，大拇指的指向就是 z 轴的正向；

3. 设定各轴上的长度单位，通常都设为 1。

这样就构成了一个空间直角坐标系，称为空间直角坐标系 $O-xyz$。定点 O 称为该坐标系的原点。

任意两条坐标轴确定一个平面，这样可确定三个互相垂直的平面，统称为坐标面。其中 x 轴与 y 轴所确定的坐标面称为 xOy 面，类似地有 yOz 面和 zOx 面。三个坐标面把空间分成八个部分，每一部分称为一个卦限。

空间直角坐标系下位置的描述

取定空间直角坐标系 $O-xyz$ 后，就可以建立空间的点与三元组之间的一一对应关系。

设点 M 为空间的一点，过点 M 分别作垂直于 x 轴、y 轴和 z 轴的平面。设三个平面与 x 轴、y 轴和 z 轴的交点依次为 P,Q,R，点 P,Q,R 分别称为点 M 在 x 轴、y 轴和 z 轴上的投影。又设点 P,Q,R 在 x 轴、y 轴和 z 轴上的坐标依次为 x,y,z，于是点 M 确定了一个三元组 (x,y,z)。

反之，如果给定一个三元组 (x,y,z)，可以在 x 轴上取坐标为 x 的点 P，在 y 轴上取坐标为 y 的点 Q，在 z 轴上取坐标为 z 的点 R，然后点 P,Q,R 分别作垂直于 x 轴、y 轴和 z 轴的三个平面，它们相交于空间的一点 M，点 M 就是由三元组 (x,y,z) 所确定的点。

这样一来，空间的点 M 与三元组 (x,y,z) 之间就建立了一一对应的关系。把三元组 (x,y,z) 称为点 M 的坐标，记作 $M(x,y,z)$，其中 x 称为横坐标、y 称为纵坐标、z 称为竖坐标。

空间柱坐标系

空间柱坐标系，将极坐标扩展为三维的方式：从应用于平面工作中的极坐标系开始，然后过极点 O 添加垂直于该平面的 z 轴，方向朝上。

为了找到由柱坐标 (ρ,φ,z) 所描述的点，可以首先在极坐标系下处理 ρ 和 φ，然后根据 z 坐标沿着 z 轴「向上」或「向下」移动。

柱坐标系与空间直角坐标系的相互转换

两坐标系下 z 的值是相同的。

(x,y) 与 (ρ,φ) 的相互转换参见上文平面直角坐标系与极坐标系的相互转换。

空间球坐标系

球坐标可以通过以下方法确定：

1. 站在原点，面向水平极轴的方向；垂直轴的指向是从脚指向头部；
2. 手臂向上，指向垂直极轴方向；
3. 逆时针旋转角度 φ；
4. 将手臂向下旋转角度 ϑ，手臂指向 φ 和 ϑ 指定的方向；
5. 沿着该方向从原点移位距离 r。

这样即可到达球坐标 (r,ϑ,φ) 所描述的点。其中 ϑ 称为**天顶角**，φ 称为**方位角**。

warning

由于诸多原因，有的地方使用 ϕ 表示天顶角，用 θ 表示方位角。阅读文章遇到球坐标系时请务必注意这一点。同时，在写文章时，如果用到了球坐标系，建议提前声明清楚使用什么符号表示天顶角和方位角。

柱坐标系与球坐标系的相互转换

两坐标系下 φ 的值是相同的。

从柱坐标系到球坐标系：

$$
r = \sqrt{\rho^2 + z^2}
$$

$$
\vartheta = \begin{cases} \arctan\left(\frac{\rho}{z}\right) & \text{if } z > 0 \\ \pi/2 & \text{if } z = 0, \rho \neq 0 \\ \arctan\left(\frac{\rho}{z}\right) + \pi & \text{if } z < 0 \end{cases}
$$

注意对于柱坐标系下的点 $(0, 0, 0)$，其球坐标的 ϑ 不明确。

从球坐标系到柱坐标系：

$$
\rho = r \sin \vartheta
$$

$$
z = r \cos \vartheta
$$

空间直角坐标系与球坐标系的相互转换

可以结合上文平面直角坐标系与极坐标系的相互转换和上文柱坐标系与球坐标系的相互转换一起使用，或直接使用下面的公式：

从空间直角坐标系到球坐标系：

$$
r = \sqrt{x^2 + y^2 + z^2}
$$

$$
\vartheta = \arccos\left(\frac{z}{\sqrt{x^2 + y^2 + z^2}}\right)
$$

$$
\varphi = \operatorname{atan2}(y, x)
$$

其中 atan2 的定义见平面直角坐标系与极坐标系的相互转换。

注意对于平面直角坐标系下的点 $(0, 0, 0)$，其球坐标的 ϑ 和 φ 取值不明确。

从球坐标系到空间直角坐标系：

$$
x = r \sin \vartheta \cos \varphi
$$

$$
y = r \sin \vartheta \sin \varphi
$$

$$
z = r \cos \vartheta
$$

参考资料与注释

[1] 该函数

9.11 复数

如果您已经学习过复数相关知识，请跳过本页面。

学习复数知识需要一部分向量基础，如果并未学习过向量知识请移步 向量页面。

复数

引入

" 注"

下面的引入方法来自人教版高中数学 A 版必修二。

从方程的角度看，负实数能不能开平方，就是方程 $x^2 + a = 0 (a > 0)$ 有没有解，进而可以归结为方程 $x^2 + 1 = 0$ 有没有解。

回顾已有的数集扩充过程，可以看到，每次扩充都与实际需求密切相关。例如，为了解决正方形对角线的度量，以及 $x^2 - 2 = 0$ 这样的方程在有理数集中无解的问题，人们把有理数集扩充到了实数集。数集扩充后，在实数集中规

定的加法运算、乘法运算，与原来在有理数集中规定的加法运算、乘法运算协调一致，并且加法和乘法都满足交换律和结合律，乘法对加法满足分配律。

依照这种思想，为了解决 $x^2+1=0$ 这样的方程在实数系中无解的问题，我们设想引入一个新数 i，使得 $x=\mathrm{i}$ 是方程 $x^2+1=0$ 的解，即使得 $\mathrm{i}^2=-1$。

思考：把新引进的数 i 添加到实数集中，我们希望数 i 和实数之间仍然能像实数那样进行加法和乘法运算，并希望加法和乘法都满足交换律、结合律，以及乘法对加法满足分配律。那么，实数系经过扩充后，得到的新数系由哪些数组成呢?

依照以上设想，把实数 b 与 i 相乘，结果记作 $b\mathrm{i}$；把实数 a 与 $b\mathrm{i}$ 相加，结果记作 $a+b\mathrm{i}$。注意到所有实数以及 i 都可以写成 $a+b\mathrm{i}(a,b\in\mathbf{R})$ 的形式，从而这些数都在扩充后的新数集中。

定义

我们定义形如 $a+b\mathrm{i}$，其中 $a,b\in\mathbf{R}$ 的数叫做**复数**，其中 i 被称为**虚数单位**，全体复数的集合叫做**复数集**，记作 $\mathbf{C}$。

复数通常用 z 表示，即 $z=a+b\mathrm{i}$。这种形式被称为**复数的代数形式**。其中 a 称为复数 z 的**实部**，记作 $\mathrm{Re}(z)$，b 称为复数 z 的**虚部**，记作 $\mathrm{Im}(z)$。如无特殊说明，都有 $a,b\in\mathbf{R}$。

对于一个复数 z，当且仅当 $b=0$ 时，它是实数，当 $b\neq0$ 时，它是虚数，当 $a=0$ 且 $b\neq0$ 时，它是纯虚数。

纯虚数，虚数，实数，复数的关系如下图所示。

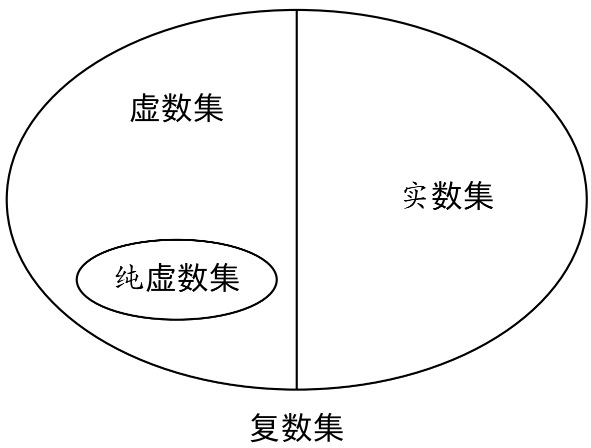


图 9.6

性质与运算

几何意义

我们知道了 $a+bi$ 这样类似的形式的数被称为复数，并且给出了定义和分类，我们还可以挖掘一下更深层的性质。

我们把所有实数都放在了数轴上，并且发现数轴上的点与实数一一对应。我们考虑对复数也这样处理。

首先我们定义**复数相等**：两个复数 $z_1=a+b\mathrm{i},z_2=c+d\mathrm{i}$ 是相等的，当且仅当 $a=c$ 且 $b=d$。

这么定义是十分自然的，在此不做过多解释。

也就是说，我们可以用唯一的有序实数对 (a,b) 表示一个复数 $z=a+b\mathrm{i}$。这样，联想到平面直角坐标系，我们可以发现**复数集与平面直角坐标系中的点集一一对应**。好了，我们找到了复数的一种几何意义。

那么这个平面直角坐标系就不再一般，因为平面直角坐标系中的点具有了特殊意义——表示一个复数，所以我们把这样的平面直角坐标系称为**复平面**，x 轴称为**实轴**，y 轴称为**虚轴**。我们进一步地说：**复数集与复平面内所有的点所构成的集合是一一对应的**。

我们考虑到学过的平面向量的知识，发现向量的坐标表示也是一个有序实数对 (a,b)，显然，复数 $z=a+b\mathrm{i}$ 对

应复平面内的点 $Z(a,b)$，那么它还对应平面向量 $\overrightarrow{OZ}=(a,b)$，于是我们又找到了复数的另一种几何意义：**复数集与复平面内的向量所构成的集合是一一对应的（实数 0 与零向量对应）**。

于是，我们由向量的知识迁移到复数上来，定义**复数的模**就是复数所对应的向量的模。复数 $z=a+b\mathrm{i}$ 的模 $|z|=\sqrt{a^2+b^2}$。

于是为了方便，我们常把复数 $z=a+b\mathrm{i}$ 称为点 Z 或向量 $\overrightarrow{OZ}$，并规定相等的向量表示同一个复数。

并且由向量的知识我们发现，虚数不可以比较大小（但是实数是可以的）。

加法与减法

对复数 $z_1=a+b\mathrm{i},z_2=c+d\mathrm{i}$，定义加法规则如下：

$$z_1+z_2=(a+c)+(b+d)\mathrm{i}$$

很明显，两个复数的和仍为复数。

考虑到向量的加法运算，我们发现复数的加法运算符合向量的加法运算法则，这同样证明了复数的几何意义的正确性。

同样可以验证，复数的加法满足**交换律**和**结合律**。即：

$$\begin{gathered}z_1+z_2=z_2+z_1\\(z_1+z_2)+z_3=z_1+(z_2+z_3)\end{gathered}$$

减法作为加法的逆运算，我们可以通过加法法则与复数相等的定义来推导出减法法则：

$$z_1-z_2=(a-c)+(b-d)\mathrm{i}$$

这同样符合向量的减法运算。

乘法、除法与共轭

对复数 $z_1=a+b\mathrm{i},z_2=c+d\mathrm{i}$，定义乘法规则如下：

$$\begin{aligned}z_1z_2&=(a+b\mathrm{i})(c+d\mathrm{i})\\&=ac+bc\mathrm{i}+ad\mathrm{i}+bd\mathrm{i}^2\\&=(ac-bd)+(bc+ad)\mathrm{i}\end{aligned}$$

可以看出，两个复数相乘类似于两个多项式相乘，只需要把 i^2 换成 -1，并将实部与虚部分别合并即可。

复数的乘法与向量的向量积形式类似。

易得复数乘法满足**交换律**，**结合律**和**对加法的分配律**，即：

- $z_1z_2=z_2z_1$
- $(z_1z_2)z_3=z_1(z_2z_3)$
- $z_1(z_2+z_3)=z_1z_2+z_1z_3$

由于满足运算律，我们可以发现实数域中的**乘法公式在复数域中同样适用**。

除法运算是乘法运算的逆运算，我们可以推导一下：

$$\begin{aligned}\frac{a+b\mathrm{i}}{c+d\mathrm{i}}&=\frac{(a+b\mathrm{i})(c-d\mathrm{i})}{(c+d\mathrm{i})(c-d\mathrm{i})}\\&=\frac{ac+bd}{c^2+d^2}+\frac{bc-ad}{c^2+d^2}\mathrm{i}\quad(c+d\mathrm{i}\neq 0)\end{aligned}$$

由于向量没有除法，这里不讨论与向量的关系。

为了分母实数化，我们乘了一个 $c-d\mathrm{i}$，这个式子很有意义。

对复数 $z=a+bi$，称 $a-bi$ 为 z 的**共轭复数**，通常记为 $\bar{z}$。我们可以发现，若两个复数互为共轭复数，那么它们**关于实轴对称**。

对复数 z,w，复数共轭有如下性质

- $z\cdot\bar{z}=|z|^2$
- $\bar{\bar{z}}=z$
- $\mathrm{Re}(z)=\dfrac{z+\bar{z}}{2}$，$\mathrm{Im}(z)=\dfrac{z-\bar{z}}{2}$
- $\overline{z\pm w}=\bar{z}\pm\bar{w}$
- $\overline{zw}=\bar{z}\bar{w}$
- $\overline{z/w}=\bar{z}/\bar{w}$

辐角和辐角主值

如果设定实数单位 1 作为水平正方向，虚数单位 i 作为竖直正方向，得到的就是直角坐标视角下的复平面。

表示复数 z 的位置，也可以借助于极坐标 (r,θ) 确定。前文已经提到了 r 为复数 z 的模。

从实轴正向到**非零**复数 $z=x+\mathrm{i}y$ 对应向量的夹角 θ 满足关系：

$$\tan\theta=\frac{y}{x}$$

称为复数 z 的**辐角**，记为：

$$\theta=\operatorname{Arg}z$$

任一个**非零**复数 z 有无穷多个辐角，故 $\operatorname{Arg}z$ 事实上是一个集合。借助开头小写的 $\arg z$ 表示**其中一个特定值**，满足条件：

$$-\pi<\arg z\le\pi$$

称 $\arg z$ 为**辐角主值**或**主辐角**。辐角就是辐角主值基础上加若干整数个（可以为零或负整数）$2k\pi$，即 $\operatorname{Arg}z=\{\arg z+2k\pi\mid k\in\mathbf{Z}\}$。

需要注意的是两个辐角主值相加后不一定还是辐角主值，而两个辐角相加一定还是合法的辐角。

称模小于 1 的复数，在复平面上构成的图形为**单位圆**。称模等于 1 的复数为**单位复数**，全体单位复数在复平面上构成的图形为**单位圆周**。在不引起混淆的情况下，有时单位圆周也简称单位圆。

在极坐标的视角下，复数的乘除法变得很简单。复数乘法，模相乘，辐角相加。复数除法，模相除，辐角相减。

欧拉公式

" **欧拉公式**（Euler's formula）[1]"

对任意实数 x，有

$$\mathrm{e}^{\mathrm{i}x}=\cos x+\mathrm{i}\sin x$$

在补充复指数函数与复三角函数的定义后，该公式可推广至全体复数。

指数函数与三角函数

对于复数 $z=x+\mathrm{i}y$，函数 $f(z)=\mathrm{e}^x(\cos y+\mathrm{i}\sin y)$ 满足 $f(z_1+z_2)=f(z_1)f(z_2)$。由此给出**复指数函数**的定义：

$$\exp z=\mathrm{e}^x(\cos y+\mathrm{i}\sin y)$$

复指数函数在实数集上与实指数函数的定义完全一致。在复平面上拥有性质：

- 模恒正：$|\exp z|=\exp x>0$。

- 辐角主值：$\arg\exp z = y$。
- 加法定理：$\exp(z_1 + z_2) = \exp(z_1)\exp(z_2)$。
- 周期性：$\exp z$ 是以 $2\pi\mathrm{i}$ 为基本周期的周期函数。如果一个函数 $f(z)$ 的周期是某一周期的整倍数，称该周期为**基本周期**。

复三角函数（也简称三**角函数**）的定义如下：

$$\cos z = \frac{\exp(\mathrm{i}z) + \exp(-\mathrm{i}z)}{2}$$

$$\sin z = \frac{\exp(\mathrm{i}z) - \exp(-\mathrm{i}z)}{2\mathrm{i}}$$

若取 $z \in \mathbf{R}$，则由欧拉公式有：

$$\cos z = \operatorname{Re}\left(\mathrm{e}^{\mathrm{i}z}\right)$$

$$\sin z = \operatorname{Im}\left(\mathrm{e}^{\mathrm{i}z}\right)$$

复三角函数在实数集上与实三角函数的定义完全一致。在复平面上拥有性质：

- 奇偶性：正弦函数是奇函数，余弦函数是偶函数。
- 三角恒等式：通常的三角恒等式都成立，例如平方和为 1，或者角的和差公式等。
- 周期性：正弦与余弦函数以 2π 为基本周期。
- 零点：实正弦与实余弦函数的全体零点，构成了复正弦与复余弦函数的全体零点。这个推广没有引进新的零点。
- 模的无界性：复正弦与复余弦函数，模长可以大于任意给定的正数，不再像实正弦与实余弦函数一样被限制在 1 的范围内。

复数的三种形式

借助直角坐标系的视角以及极坐标系的视角，可以写出复数的三种形式。

复数的**代数形式**用于表示任意复数。

$$z = x + y\mathrm{i}$$

代数形式用于计算复数的加减乘除四个运算比较方便。

复数的**三角形式**和**指数形式**，用于表示非零复数。

$$z = r(\cos\theta + \mathrm{i}\sin\theta) = r\exp(\mathrm{i}\theta)$$

这两种形式用于计算复数的乘除两个运算以及后面的运算较为方便。如果只用高中见过的函数，可以使用三角形式。如果引入了复指数函数，写成等价的指数形式会更加方便。

单位根

称 $x^n = 1$ 在复数意义下的解是 n 次复根。显然，这样的解有 n 个，称这 n 个解都是 n 次**单位根**或**单位复根**（the n-th root of unity）。根据复平面的知识，n 次单位根把单位圆 n 等分。

设 $\omega_n = \exp\frac{2\pi\mathrm{i}}{n}$（即幅角为 $2\frac{\pi}{n}$ 的单位复数），则 $x^n = 1$ 的解集表示为 $\{\omega_n^k \mid k = 0, 1\cdots, n-1\}$。

如果不加说明，一般叙述的 n 次单位根，是指从 1 开始逆时针方向的第一个解，即上述 ω_n，其它解均可以用 ω_n 的幂表示。

性质

单位根有三个重要的性质。对于任意正整数 n 和整数 k：

$$\omega_n^n = 1$$
$$\omega_n^k = \omega_{2n}^{2k}$$
$$\omega_{2n}^{k+n} = -\omega_{2n}^k$$

推导留给读者自证。这三个性质在快速傅里叶变换中会得到应用。

本原单位根

为什么说，上述 n 个解都是 n 次单位根，而平时说的 n 次单位根一般特指第一个？

特指第一个，是为了在应用时方便。

在解方程的视角看来，满足 ω_n 性质的不止 ω_n 一个，对于 ω_n 的若干次幂也会满足性质。

称集合：

$$\{\omega_n^k \mid 0 \le k < n, \gcd(n,k) = 1\}$$

中的元素为**本原单位根**。任意一个本原单位根 ω，与上述 ω_n 具有相同的性质：对于任意的 $0 < k < n$，ω 的 k 次幂不为 1。因此，借助任意一个本原单位根，都可以生成全体单位根。

全体 n 次本原单位根共有 $\varphi(n)$ 个。

编程语言中的复数

C 中的复数

在 C99 标准中，有 `<complex.h>` 头文件。

在 `<complex.h>` 头文件中，提供了 `double complex`、`float complex` 和 `long double complex` 三种类型。

算术运算符'+'、'-'、'*' 和'/'，可以用于浮点数和复数的任意混合。当表达式两端有一个为复数时，计算结果为复数。

头文件 `<complex.h>` 提供了虚数单位 `I`，引入此头文件时，大写字母 `I` 不可以作为变量名使用。

对于单个复数，`<complex.h>` 提供了若干操作：`creal` 函数用于提取实部，`cimag` 函数用于提取虚部，`cabs` 函数用于计算模，`carg` 函数用于计算辐角主值。

所有的函数根据类型不同，都有三个。例如 `creal` 函数有 `creal`、`crealf`、`creall` 三个，用于处理对应的 `double`、`float` 和 `long double` 三种类型。末尾什么都不带的默认处理 `double` 类型。以下所有函数均遵从此规律，不再特别说明。

这些函数返回值都是一般的浮点数。可以将普通浮点数直接赋值给复数，但是不可以将复数直接赋值给浮点数，而是需要使用上述提取操作。

函数 `conj` 用于计算共轭复数，返回值是复数。

函数 `cexp` 计算复指数，`clog` 计算对数主值，`csin` 计算正弦，`ccos` 计算余弦，`ctan` 计算正切。

函数 `cpow` 计算幂函数，`csqrt` 计算平方根，`casin` 计算反正弦，`cacos` 计算反余弦，`ctan` 计算反正切。这部分函数计算的全部都是多值函数的主值。

C++ 中的复数

在 C 里面的 `<ctype.h>`，到 C++ 会变成 `<cctype>`，几乎所有的头文件遵从这个命名规律。

但是，`<complex.h>` 不遵守，C++ 没有 `<ccomplex>` 头文件。C++ 的复数直接是 `<complex>`，并且装的东西和 C 完全不一样。

很有趣。这是因为，在 C++ 的第一个版本 C++98，即已经有了 `<complex>`，而 C 语言在 C99 才添加。

在 C++ 中，复数类型定义使用 `complex<float>`、`complex<double>` 和 `complex<long double>`。由于面向对象的多态性，下面函数的名字都是唯一的，无需 f 或 l 的后缀。

一个复数对象拥有成员函数 `real` 和 `imag`，可以访问实部和虚部。

一个复数对象拥有非成员函数 `real`、`imag`、`abs`、`arg`，返回实部、虚部、模和辐角。

一个复数对象还拥有非成员函数：`norm` 为模的平方，`conj` 为共轭复数。

一个复数对象还拥有非成员函数 `exp`、`log`（底为 e 的对数主值）、`log10`（底为 10 的对数主值，C 中没有）、`pow`、`sqrt`、`sin`、`cos`、`tan`，含义与 C 中的含义相同。

在 C++14 及以后的版本中，定义了字面量运算符 `std::literals::complex_literals::""if, ""i, ""il`[2]。例如输入 `100if`、`100i` 和 `100il`，三者将分别返回 `std::complex<float>{0.0f, 100.0f}`、`std::complex<double>{0.0, 100.0}` 以及 `std::complex<long double>{0.0l, 100.0l}`。这使得我们可以方便地书写形如 `auto z = 4 + 3i` 的复数声明。

参考资料与链接

- Complex number - Wikipedia[3]
- Euler's formula - Wikipedia[4]
- Complex number arithmetic - cppreference.com[5]
- std::complex - cppreference.com[6]

[1] 有关欧拉公式的更多介绍，可以参考两个视频：欧拉公式与初等群论、微分方程概论 - 第五章： 在 3.14 分钟内理解 $e^{i\pi}$。

[2] 字面量运算符 `std::literals::complex_literals::""if, ""i, ""il`

[3] Complex number - Wikipedia

[4] Euler's formula - Wikipedia

[5] Complex number arithmetic - cppreference.com

[6] std::complex - cppreference.com

9.12 数论

9.12.1 数论基础

本文对于数论的开头部分做一个简介。

整除

整除的定义：设 $a, b \in \mathbf{Z}$，$a \neq 0$。如果 $\exists q \in \mathbf{Z}$，使得 $b = aq$，那么就说 b 可被 a 整除，记作 $a \mid b$；b 不被 a 整除记作 $a \nmid b$。

整除的性质：

- $a \mid b \iff -a \mid b \iff a \mid -b \iff |a| \mid |b|$
- $a \mid b \land b \mid c \implies a \mid c$

- $a \mid b \wedge a \mid c \iff \forall x, y \in \mathbf{Z}, a \mid (xb + yc)$
- $a \mid b \wedge b \mid a \implies b = \pm a$
- 设 $m \neq 0$，那么 $a \mid b \iff ma \mid mb$。
- 设 $b \neq 0$，那么 $a \mid b \implies |a| \leq |b|$。
- 设 $a \neq 0, b = qa + c$，那么 $a \mid b \iff a \mid c$。

约数（因数）：若 $a \mid b$，则称 b 是 a 的倍数，a 是 b 的约数。

0 是所有非 0 整数的倍数。对于整数 $b \neq 0$，b 的约数只有有限个。

平凡约数（平凡因数）：对于整数 $b \neq 0$，± 1、$\pm b$ 是 b 的平凡约数。当 $b = \pm 1$ 时，b 只有两个平凡约数。

对于整数 $b \neq 0$，b 的其他约数称为真约数（真因数、非平凡约数、非平凡因数）。

约数的性质：

- 设整数 $b \neq 0$。当 d 遍历 b 的全体约数的时候，$\dfrac{b}{d}$ 也遍历 b 的全体约数。
- 设整数 $b > 0$，则当 d 遍历 b 的全体正约数的时候，$\dfrac{b}{d}$ 也遍历 b 的全体正约数。

在具体问题中，**如果没有特别说明，约数总是指正约数**。

带余数除法

余数的定义：设 a,b 为两个给定的整数，$a \neq 0$。设 d 是一个给定的整数。那么，一定存在唯一的一对整数 q 和 r，满足 $b = qa + r, d \leq r < |a| + d$。

无论整数 d 取何值，r 统称为余数。$a \mid b$ 等价于 $a \mid r$。

一般情况下，d 取 0，此时等式 $b = qa + r, 0 \leq r < |a|$ 称为带余数除法（带余除法）。这里的余数 r 称为最小非负余数。

余数往往还有两种常见取法：

绝对最小余数：d 取 a 的绝对值的一半的相反数。即 $b = qa + r, -\dfrac{|a|}{2} \leq r < |a| - \dfrac{|a|}{2}$。

最小正余数：d 取 1。即 $b = qa + r, 1 \leq r < |a| + 1$。

带余数除法的余数只有最小非负余数。**如果没有特别说明，余数总是指最小非负余数**。

余数的性质：

- 任一整数被正整数 a 除后，余数一定是且仅是 0 到 $(a - 1)$ 这 a 个数中的一个。
- 相邻的 a 个整数被正整数 a 除后，恰好取到上述 a 个余数。特别地，一定有且仅有一个数被 a 整除。

最大公约数与最小公倍数

关于公约数、公倍数、最大公约数与最小公倍数，四个名词的定义，见 最大公约数。

互素

两个整数互素（既约）的定义：若 $\gcd(a_1, a_2) = 1$，则称 a_1 和 a_2 互素（既约）。

多个整数互素（既约）的定义：若 $\gcd(a_1, \ldots, a_k) = 1$，则称 $a_1, \ldots, a_k$ 互素（既约）。

多个整数互素，不一定两两互素。例如 6、10 和 15 互素，但是任意两个都不互素。

互素的性质与最大公约数理论：裴蜀定理（Bézout's identity）。见 裴蜀定理。

辗转相除法

辗转相除法是一种算法，也称 Euclid 算法。见 最大公约数。

素数与合数

关于素数的算法见 素数。

设整数 $p \neq 0, \pm 1$。如果 p 除了平凡约数外没有其他约数，那么称 p 为素数（不可约数）。

若整数 $a \neq 0, \pm 1$ 且 a 不是素数，则称 a 为合数。

p 和 $-p$ 总是同为素数或者同为合数。**如果没有特别说明，素数总是指正的素数**。

整数的因数是素数，则该素数称为该整数的素因数（素约数）。

素数与合数的简单性质：

- 大于 1 的整数 a 是合数，等价于 a 可以表示为整数 d 和 e（$1 < d, e < a$）的乘积。
- 如果素数 p 有大于 1 的约数 d，那么 $d = p$。
- 大于 1 的整数 a 一定可以表示为素数的乘积。
- 对于合数 a，一定存在素数 $p \leq \sqrt{a}$ 使得 $p \mid a$。
- 素数有无穷多个。
- 所有大于 3 的素数都可以表示为 $6n \pm 1$ 的形式[1]。

算术基本定理

算术基本引理：

设 p 是素数，$p \mid a_1 a_2$，那么 $p \mid a_1$ 和 $p \mid a_2$ 至少有一个成立。

算术基本引理是素数的本质属性，也是素数的真正定义。

算术基本定理（唯一分解定理）：

设正整数 a，那么必有表示：

$$a = p_1 p_2 \cdots p_s$$

其中 $p_j (1 \leq j \leq s)$ 是素数。并且在不计次序的意义下，该表示唯一。

标准素因数分解式：

将上述表示中，相同的素数合并，可得：

$$a = {p_1}^{\alpha_1} {p_2}^{\alpha_2} \cdots {p_s}^{\alpha_s}, p_1 < p_2 < \cdots < p_s$$

称为正整数 a 的标准素因数分解式。

算术基本定理和算术基本引理，两个定理是等价的。

同余

同余的定义：设整数 $m \neq 0$。若 $m \mid (a - b)$，称 m 为模数（模），a 同余于 b 模 m，b 是 a 对模 m 的剩余。记作 $a \equiv b \pmod m$。

否则，a 不同余于 b 模 m，b 不是 a 对模 m 的剩余。记作 $a \not\equiv b \pmod m$。

这样的等式，称为模 m 的同余式，简称同余式。

根据整除的性质，上述同余式也等价于 $a \equiv b \pmod{(-m)}$。

如果没有特别说明，模数总是正整数。

式中的 b 是 a 对模 m 的剩余，这个概念与余数完全一致。通过限定 b 的范围，相应的有 a 对模 m 的最小非负剩余、绝对最小剩余、最小正剩余。

同余的性质：

- 自反性：$a \equiv a \pmod m$。
- 对称性：若 $a \equiv b \pmod m$，则 $b \equiv a \pmod m$。

- 传递性：若 $a \equiv b \pmod{m}, b \equiv c \pmod{m}$，则 $a \equiv c \pmod{m}$。
- 线性运算：若 $a, b, c, d \in \mathbf{Z}, m \in \mathbf{N}^*, a \equiv b \pmod{m}, c \equiv d \pmod{m}$ 则有：
  - $a \pm c \equiv b \pm d \pmod{m}$。
  - $a \times c \equiv b \times d \pmod{m}$。
- 若 $a, b \in \mathbf{Z}, k, m \in \mathbf{N}^*, a \equiv b \pmod{m}$, 则 $ak \equiv bk \pmod{mk}$。
- 若 $a, b \in \mathbf{Z}, d, m \in \mathbf{N}^*, d \mid a, d \mid b, d \mid m$，则当 $a \equiv b \pmod{m}$ 成立时，有 $\frac{a}{d} \equiv \frac{b}{d} \left(\bmod \frac{m}{d}\right)$。
- 若 $a, b \in \mathbf{Z}, d, m \in \mathbf{N}^*, d \mid m$，则当 $a \equiv b \pmod{m}$ 成立时，有 $a \equiv b \pmod{d}$。
- 若 $a, b \in \mathbf{Z}, d, m \in \mathbf{N}^*$，则当 $a \equiv b \pmod{m}$ 成立时，有 $\gcd(a, m) = \gcd(b, m)$。若 d 能整除 m 及 a, b 中的一个，则 d 必定能整除 a, b 中的另一个。

还有性质是乘法逆元。见 乘法逆元。

C/C++ 的整数除法和取模运算

在 C/C++ 中，整数除法和取模运算，与数学上习惯的取模和除法不一致。

对于所有标准版本的 C/C++，规定在整数除法中：

1. 当除数为 0 时，行为未定义；
2. 否则 `(a / b) * b + a % b` 的运算结果与 `a` 相等。

也就是说，取模运算的符号取决于除法如何取整；而除法如何取整，这是实现定义的（由编译器决定）。

从 C99[2] **和 C++11**[3] **标准版本起**，规定**商向零取整**（舍弃小数部分）；取模的符号即与被除数相同。从此以下运算结果保证为真：

```
5 % 3 == 2;
5 % -3 == 2;
-5 % 3 == -2;
-5 % -3 == -2;
```

数论函数

数论函数指定义域为正整数的函数。数论函数也可以视作一个数列。

积性函数

定义

若函数 $f(n)$ 满足 $f(1) = 1$ 且 $\forall x, y \in \mathbf{N}^*, \gcd(x, y) = 1$ 都有 $f(xy) = f(x)f(y)$，则 $f(n)$ 为积性函数。

若函数 $f(n)$ 满足 $f(1) = 1$ 且 $\forall x, y \in \mathbf{N}^*$ 都有 $f(xy) = f(x)f(y)$，则 $f(n)$ 为完全积性函数。

性质

若 $f(x)$ 和 $g(x)$ 均为积性函数，则以下函数也为积性函数：

$$
\begin{aligned}
h(x) &= f(x^p) \\
h(x) &= f^p(x) \\
h(x) &= f(x)g(x) \\
h(x) &= \sum_{d \mid x} f(d) g\left(\frac{x}{d}\right)
\end{aligned}
$$

设 $x = \prod p_i^{k_i}$

若 $F(x)$ 为积性函数，则有 $F(x)=\prod F(p_i^{k_i})$。

若 $F(x)$ 为完全积性函数，则有 $F(x)=\prod F(p_i)^{k_i}$。

例子

- 单位函数：$\varepsilon(n)=[n=1]$。（完全积性）
- 恒等函数：$\mathrm{id}_k(n)=n^k$，$\mathrm{id}_1(n)$ 通常简记作 $\mathrm{id}(n)$。（完全积性）
- 常数函数：$1(n)=1$。（完全积性）
- 除数函数：$\sigma_k(n)=\sum_{d\mid n}d^k$。$\sigma_0(n)$ 通常简记作 $d(n)$ 或 $\tau(n)$，$\sigma_1(n)$ 通常简记作 $\sigma(n)$。
- 欧拉函数：$\varphi(n)=\sum_{i=1}^n[\gcd(i,n)=1]$
- 莫比乌斯函数：$\mu(n)=\begin{cases}1 & n=1\\ 0 & \exists d>1,d^2\mid n\\ (-1)^{\omega(n)} & \text{otherwise}\end{cases}$，其中 $\omega(n)$ 表示 n 的本质不同质因子个数，它是一个加性函数。

" 加性函数"

此处加性函数指数论上的加性函数 (Additive function)。对于加性函数 f，当整数 a,b 互质时，均有 $f(ab)=f(a)+f(b)$。应与代数中的加性函数 (Additive map) 区分。

参考资料与注释

[1] Are all primes (past 2 and 3) of the forms 6n+1 and 6n-1?

[2] Arithmetic operators (C) - cppreference.com

[3] Arithmetic operators (C++) - cppreference.com

9.12.2 素数

素数与合数的定义，见 数论基础。

素数计数函数：小于或等于 x 的素数的个数，用 $\pi(x)$ 表示。随着 x 的增大，有这样的近似结果：$\pi(x)\sim\dfrac{x}{\ln(x)}$。

素数判定

我们自然地会想到，如何用计算机来判断一个数是不是素数呢?

实现

暴力做法自然可以枚举从小到大的每个数看是否能整除

```
bool isPrime(a) {
  if (a < 2) return 0;
  for (int i = 2; i < a; ++i)
    if (a % i == 0) return 0;
  return 1;
}
```

```
def isPrime(a):
    if a < 2:
        return False
    for i in range(2, a):
        if a % i == 0:
            return False
    return True
```

这样做是十分稳妥了，但是真的有必要每个数都去判断吗？

很容易发现这样一个事实：如果 x 是 a 的约数，那么 $\frac{a}{x}$ 也是 a 的约数。

这个结论告诉我们，对于每一对 $(x, \frac{a}{x})$，只需要检验其中的一个就好了。为了方便起见，我们之考察每一对里面小的那个数。不难发现，所有这些较小数就是 $[1, \sqrt{a}]$ 这个区间里的数。

由于 1 肯定是约数，所以不检验它。

```
bool isPrime(a) {
  if (a < 2) return 0;
  for (int i = 2; i * i <= a; ++i)
    if (a % i == 0) return 0;
  return 1;
}
```

```
def isPrime(a):
    if a < 2:
        return False
    for i in range(2, int(sqrt(a)) + 1):
        if a % i == 0:
            return False
    return True
```

素性测试

定义

素性测试（Primality test）是一类在**不对给定数字进行素数分解**（prime factorization）的情况下，测试其是否为素数的算法。

素性测试有两种：

1. 确定性测试：绝对确定一个数是否为素数。常见示例包括 Lucas–Lehmer 测试和椭圆曲线素性证明。
2. 概率性测试：通常比确定性测试快很多，但有可能（尽管概率很小）错误地将 合数 识别为质数（尽管反之则不会）。因此，通过概率素性测试的数字被称为**可能素数**，直到它们的素数可以被确定性地证明。而通过测试但实际上是合数的数字则被称为**伪素数**。有许多特定类型的伪素数，最常见的是费马伪素数，它们是满足费马小定理的合数。概率性测试的常见示例包括 Miller–Rabin 测试。

接下来我们将着重介绍几个概率性素性测试：

Fermat 素性测试

Fermat 素性检验是最简单的概率性素性检验。

我们可以根据 费马小定理 得出一种检验素数的思路：

基本思想是不断地选取在 $[2, n-1]$ 中的基 a，并检验是否每次都有 $a^{n-1} \equiv 1 \pmod n$。

实现

```cpp
bool millerRabin(int n) {
  if (n < 3) return n == 2;
  // test_time 为测试次数，建议设为不小于 8
  // 的整数以保证正确率，但也不宜过大，否则会影响效率
  for (int i = 1; i <= test_time; ++i) {
    int a = rand() % (n - 2) + 2;
    if (quickPow(a, n - 1, n) != 1) return 0;
  }
  return 1;
}
```

```python
def millerRabin(n):
    if n < 3:
        return n == 2
    # test_time 为测试次数，建议设为不小于 8
    # 的整数以保证正确率，但也不宜过大，否则会影响效率
    for i in range(1, test_time + 1):
        a = random.randint(0, 32767) % (n - 2) + 2
        if quickPow(a, n - 1, n) != 1:
            return False
    return True
```

如果 $a^{n-1} \equiv 1 \pmod n$ 但 n 不是素数，则 n 被称为以 a 为底的**伪素数**。我们在实践中观察到，如果 $a^{n-1} \equiv 1 \pmod n$，那么 n 通常是素数。但这里也有个反例：如果 $n = 341$ 且 $a = 2$，即使 $341 = 11 \cdot 31$ 是合数，有 $2^{340} \equiv 1 \pmod{341}$。事实上，341 是最小的伪素数基数。

很遗憾，费马小定理的逆定理并不成立，换言之，满足了 $a^{n-1} \equiv 1 \pmod n$，n 也不一定是素数。甚至有些合数 n 满足对任意满足 $n \nmid a$ 的整数 a 均有 $a^{n-1} \equiv 1 \pmod n$，这样的数称为 Carmichael 数。

Carmichael 函数

对正整数 n，定义 Carmichael 函数（卡迈克尔函数）为对任意满足 $(a, n) = 1$ 的整数 a，使

$$a^m \equiv 1 \pmod n$$

恒成立的最小正整数 m.

即：

$$\lambda(n) = \max\{\delta_n(a) : (a, n) = 1\}$$

Carmichael 函数有如下性质：

1. （**Carmichael 定理**）对任意素数 p 和任意正整数 r，

$$\lambda\left(p^r\right) = \begin{cases} \frac{1}{2}\varphi\left(p^r\right), & p = 2 \wedge r \geq 3, \\ \varphi\left(p^r\right), & \text{otherwise.} \end{cases}$$

"证明"

该定理等价于：

若模 $n = p^r$ 有 原根，则 $\lambda(n) = \varphi(n)$，否则 $\lambda(n) = \dfrac{1}{2}\varphi(n)$.

当模 p^r 有原根时，由 原根存在定理 可知命题成立。否则 $p = 2$ 且 $r \geq 3$，我们有：

$$\lambda\left(2^r\right) \mid 2^{r-2}$$

又由 $5^{2^{r-3}} \equiv 1+2^{r-1} \pmod{2^{r-2}}$ 知 $\lambda\left(2^r\right)>2^{r-3}$，因此

$$\lambda\left(p^r\right)=2^{r-2}=\frac{1}{2}\varphi\left(p^r\right)$$

进而有：

(a) 对任意正整数 n，有 $\lambda(n) \mid \varphi(n)$

(b) 对任意正整数 a，b，有 $a \mid b \implies \lambda(a) \mid \lambda(b)$

2. 令 n 的唯一分解式为 $n=\prod_{i=1}^{k} p_i^{r_i}$，则

$$\lambda(n)=\left[\lambda\left(p_1^{r_1}\right), \lambda\left(p_2^{r_2}\right), \ldots, \lambda\left(p_k^{r_k}\right)\right]$$

由 中国剩余定理 和 Carmichael 定理易证。

进而有：

(a) 对任意正整数 a，b，有 $\lambda([a, b])=[\lambda(a), \lambda(b)]$

Carmichael 数

对于合数 n，如果对于所有正整数 a，a 和 n 互素，都有同余式 $a^{n-1} \equiv 1 \pmod{n}$ 成立，则合数 n 为 **Carmichael 数**（卡迈克尔数，OEIS:A002997[5]）。

比如 $561=3 \times 11 \times 17$ 就是一个 Carmichael 数，同时也是最小的 Carmichael 数。

我们可以用如下方法判断合数 n 是否为 Carmichael 数：

"Korselt 判别法[1]"

合数 n 是 Carmichael 数当且仅当 n 无平方因子且对 n 的任意质因子 p 均有 $p-1 \mid n-1$.

上述判别法可简化为：

note

合数 n 是 Carmichael 数当且仅当 $\lambda(n) \mid n-1$，其中 $\lambda(n)$ 为 Carmichael 函数。

Carmichael 数有如下性质：

1. Carmichael 数无平方因子且至少有 3 个不同的质因子。
2. 设 $C(n)$ 为小于 n 的 Carmichael 数个数，则：

   (a)（Alford, Granville, Pomerance. 1994[2]）$C(n)>n^{2/7}$

   由此可知 Carmichael 数有无限多个。

   (b)（Erdős. 1956[3]）$C(n)<n \exp \left(-c \dfrac{\ln n \ln \ln \ln n}{\ln \ln n}\right)$，其中 c 为常数。

   由此可知 Carmichael 数的分布十分稀疏。实际上 $C(10^9)=646$，$C(10^{18})=1\,401\,644$[4].

" 注意"

「若 n 为 Carmichael 数，则 2^n-1 也为 Carmichael 数」是错误的。

如 $561=3 \cdot 11 \cdot 17$ 为 Carmichael 数，考虑 $2^{561}-1$。

注意到 $23 \cdot 89=2^{11}-1 \mid 2^{561}-1$，由 Korselt 判别法知，若 $2^{561}-1$ 是 Carmichael 数，则 22 和 88 均为 $2^{561}-2$ 的因子。

而 $v_2\left(2^{561}-2\right)=1<v_2(88)=3$，故 $88 \nmid 2^{561}-2$，因此 $2^{561}-1$ 不是 Carmichael 数。

Miller–Rabin 素性测试

Miller–Rabin 素性测试（Miller–Rabin primality test）是进阶的素数判定方法。它是由 Miller 和 Rabin 二人根据费马小定理的逆定理（费马测试）优化得到的。因为和许多类似算法一样，它是使用伪素数的概率性测试，我们必

须使用慢得多的确定性算法来保证素性。然而，实际上没有已知的数字通过了高级概率性测试（例如 Miller–Rabin）但实际上却是复合的。因此我们可以放心使用。

在不考虑乘法的复杂度时，对数 n 进行 k 轮测试的时间复杂度是 $O(k\log n)$。Miller-Rabbin 素性测试常用于对高精度数进行测试，此时时间复杂度是 $O(k\log^3 n)$，利用 FFT 等技术可以优化到 $O(k\log^2 n\log\log n\log\log\log n)$[6]。

二次探测定理

如果 p 是奇素数，则 $x^2\equiv 1 \pmod p$ 的解为 $x\equiv 1 \pmod p$ 或者 $x\equiv p-1 \pmod p$。

要证明该定理，只需将上面的方程移项，再使用平方差公式，得到 $(x+1)(x-1)\equiv 0 \bmod p$，即可得出上面的结论。

实现

根据卡迈克尔数的性质，可知其一定不是 p^e。

不妨将费马小定理和二次探测定理结合起来使用：

将 $a^{n-1}\equiv 1 \pmod n$ 中的指数 $n-1$ 分解为 $n-1=u\times 2^t$，在每轮测试中对随机出来的 a 先求出 $v=a^u \bmod n$，之后对这个值执行最多 t 次平方操作，若发现非平凡平方根时即可判断出其不是素数，否则再使用 Fermat 素性测试判断。

还有一些实现上的小细节：

- 对于一轮测试，如果某一时刻 $a^{u\times 2^s}\equiv n-1 \pmod n$，则之后的平方操作全都会得到 1，则可以直接通过本轮测试。
- 如果找出了一个非平凡平方根 $a^{u\times 2^s}\not\equiv n-1 \pmod n$，则之后的平方操作全都会得到 1。可以选择直接返回 `false`，也可以放到 t 次平方操作后再返回 `false`。

这样得到了较正确的 Miller Rabin：（来自 fjzzq2002）

```cpp
bool millerRabin(int n) {
  if (n < 3 || n % 2 == 0) return n == 2;
  int u = n - 1, t = 0;
  while (u % 2 == 0) u /= 2, ++t;
  // test_time 为测试次数，建议设为不小于 8
  // 的整数以保证正确率，但也不宜过大，否则会影响效率
  for (int i = 0; i < test_time; ++i) {
    int a = rand() % (n - 2) + 2, v = quickPow(a, u, n);
    if (v == 1) continue;
    int s;
    for (s = 0; s < t; ++s) {
      if (v == n - 1) break;  // 得到平凡平方根 n-1，通过此轮测试
      v = (long long)v * v % n;
    }
    // 如果找到了非平凡平方根，则会由于无法提前 break; 而运行到 s == t
    // 如果 Fermat 素性测试无法通过，则一直运行到 s == t 前 v 都不会等于 -1
    if (s == t) return 0;
  }
  return 1;
}
```

```python
def millerRabin(n):
    if n < 3 or n % 2 == 0:
        return n == 2
    u, t = n - 1, 0
    while u % 2 == 0:
        u = u // 2
```

```
        t = t + 1
    # test_time 为测试次数，建议设为不小于 8
    # 的整数以保证正确率，但也不宜过大，否则会影响效率
    for i in range(test_time):
        a = random.randint(2, n - 1)
        v = pow(a, u, n)
        if v == 1:
            continue
        s = 0
        while s < t:
            if v == n - 1:
                break
            v = v * v % n
            s = s + 1
        # 如果找到了非平凡平方根，则会由于无法提前 break; 而运行到 s == t
        # 如果 Fermat 素性测试无法通过，则一直运行到 s == t 前 v 都不会等于 -1
        if s == t:
            return False
    return True
```

另外，假设 广义 Riemann 猜想[7]（generalized Riemann hypothesis, GRH）成立，则对数 n 最多只需要测试 $[2, \min\{n-2, \lfloor 2\ln^2 n \rfloor\}]$ 中的全部整数即可**确定**数 n 的素性，证明参见注释 7。

而在 OI 范围内，通常都是对 $[1, 2^{64})$ 范围内的数进行素性检验。对于 $[1, 2^{32})$ 范围内的数，选取 $\{2, 7, 61\}$ 三个数作为基底进行 Miller–Rabin 素性检验就可以确定素性；对于 $[1, 2^{64})$ 范围内的数，选取 $\{2, 325, 9375, 28178, 450775, 9780504, 1795265$ 七个数作为基底进行 Miller–Rabin 素性检验就可以确定素性。参见注释 8。

也可以选取 $\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37\}$（即前 12 个素数）检验 $[1, 2^{64})$ 范围内的素数。

注意如果要使用上面的数列中的数 a 作为基底判断 n 的素性：

- 所有的数都要取一遍，不能只选小于 n 的；
- 把 a 换成 $a \bmod n$；
- 如果 $a \equiv 0 \pmod n$，则直接通过该轮测试。

反素数

引入

顾名思义，素数就是因子只有两个的数，那么反素数，就是因子最多的数（并且因子个数相同的时候值最小），所以反素数是相对于一个集合来说的。

一种符合直觉的反素数定义是：在一个正整数集合中，因子最多并且值最小的数，就是反素数。

定义

如果某个正整数 n 满足如下条件，则称为是**反素数**：任何小于 n 的正数的约数个数都小于 n 的约数个数。

" 注意"

注意区分 emirp[8]，它表示的是逐位反转后是不同素数的素数（如 149 和 941 均为 emirp，101 不是 emirp）。

过程

那么，如何来求解反素数呢？

首先，既然要求因子数，首先要做的就是素因子分解。把 n 分解成 $n = p_1^{k_1} p_2^{k_2} \cdots p_n^{k_n}$ 的形式，其中 p 是素数，k 为他的指数。这样的话总因子个数就是 $(k_1+1) \times (k_2+1) \times (k_3+1) \cdots \times (k_n+1)$。

但是显然质因子分解的复杂度是很高的，并且前一个数的结果不能被后面利用。所以要换个方法。

我们来观察一下反素数的特点。

1. 反素数肯定是从 2 开始的连续素数的幂次形式的乘积。
2. 数值小的素数的幂次大于等于数值大的素数，即 $n = p_1^{k_1} p_2^{k_2} \cdots p_n^{k_n}$ 中，有 $k_1 \geq k_2 \geq k_3 \geq \cdots \geq k_n$

解释：

1. 如果不是从 2 开始的连续素数，那么如果幂次不变，把素数变成数值更小的素数，那么此时因子个数不变，但是 n 的数值变小了。交换到从 2 开始的连续素数的时候 n 值最小。
2. 如果数值小的素数的幂次小于数值大的素数的幂，那么如果把这两个素数交换位置（幂次不变），那么所得的 n 因子数量不变，但是 n 的值变小。

另外还有两个问题，

1. 对于给定的 n，要枚举到哪一个素数呢？

   最极端的情况大不了就是 $n = p_1 p_2 \cdots p_n$，所以只要连续素数连乘到刚好小于等于 n 就可以的呢。再大了，连全都一次幂，都用不了，当然就是用不到的啦！
2. 我们要枚举到多少次幂呢？

   我们考虑一个极端情况，当我们最小的素数的某个幂次已经比所给的 n（的最大值）大的话，那么展开成其他的形式，最大幂次一定小于这个幂次。`unsigned long long` 的最大值是 $2^{64}-1$，所以可以枚举到 $2^{64}-1$。

细节有了，那么我们具体如何具体实现呢？

我们可以把当前走到每一个素数前面的时候列举成一棵树的根节点，然后一层层的去找。找到什么时候停止呢？

1. 当前走到的数字已经大于我们想要的数字了
2. 当前枚举的因子已经用不到了（和 1 重复了嘻嘻嘻）
3. 当前因子大于我们想要的因子了
4. 当前因子正好是我们想要的因子（此时判断是否需要更新最小 ans）

然后 dfs 里面不断一层一层枚举次数继续往下迭代可以。

常见题型

1. 求因子数一定的最小数

" **例题** Codeforces 27E. A number with a given number of divisors[9] "

求具有给定除数的最小自然数。请确保答案不超过 10^{18}。

" **解题思路** "

对于这种题，我们只要以因子数为 dfs 的返回条件基准，不断更新找到的最小值就可以了

" **参考代码** "

```
#include <stdio.h>
unsigned long long p[16] = {
    2,  3,  5,  7,  11, 13, 17, 19,
    23, 29, 31, 37, 41, 43, 47, 53};  // 根据数据范围可以确定使用的素数最大为 53

unsigned long long ans;
unsigned long long n;

// depth: 当前在枚举第几个素数
```

```cpp
// temp: 当前因子数量为 num 的时候的数值
// num: 当前因子数
// up: 上一个素数的幂，这次应该小于等于这个幂次嘛
void dfs(unsigned long long depth, unsigned long long temp,
         unsigned long long num, unsigned long long up) {
  if (num > n || depth >= 16) return;  // 边界条件
  if (num == n && ans > temp) {        // 取最小的 ans
    ans = temp;
    return;
  }
  for (int i = 1; i <= up; i++) {
    if (temp * p[depth] > ans)
      break;  // 剪枝：如果加一个这个乘数的结果比 ans 要大，则必不是最佳方案
    dfs(depth + 1, temp = temp * p[depth], num * (i + 1),
        i);  // 取一个该乘数，进行对下一个乘数的搜索
  }
}

int main() {
  scanf("%llu", &n);
  ans = ~(unsigned long long)0;
  dfs(0, 1, 1, 64);
  printf("%llu\n", ans);
  return 0;
}
```

2. 求 n 以内因子数最多的数

" **例题 ZOJ - More Divisors**[10]"

大家都知道我们使用十进制记数法，即记数的基数是 10。历史学家说这是因为人有十个手指，也许他们是对的。然而，这通常不是很方便，十只有四个除数——1、2、5 和 10。因此，像 $\frac{1}{3}$、$\frac{1}{4}$ 或 $\frac{1}{6}$ 这样的分数不便于用十进制表示。从这个意义上说，以 12、24 甚至 60 为底会方便得多。主要原因是这些数字的除数要大得多——分别是 6、8 和 12。请回答：除数最多的不超过 n 的数是多少？

" **解题思路**"

思路同上，只不过要改改 dfs 的返回条件。注意这样的题目的数据范围，32 位整数可能溢出。

" **参考代码**"

```cpp
#include <cstdio>
#include <iostream>

int p[16] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53};
unsigned long long n;
unsigned long long ans,
    ans_num;  // ans 为 n 以内的最大反素数（会持续更新），ans_sum 为
              // ans 的因子数。

// depth: 当前在枚举第几个素数
// temp: 当前因子数量为 num 的时候的数值
// num: 当前因子数
```

```cpp
// up: 上一个素数的幂，这次应该小于等于这个幂次嘛
void dfs(int depth, unsigned long long temp, unsigned long long num, int up) {
  if (depth >= 16 || temp > n) return;
  if (num > ans_num) {  // 更新答案
    ans = temp;
    ans_num = num;
  }
  if (num == ans_num && ans > temp) ans = temp;  // 更新答案
  for (int i = 1; i <= up; i++) {
    if (temp * p[depth] > n)
      break;  // 剪枝：如果加一个这个乘数的结果比 ans 要大，则必不是最佳方案
    dfs(depth + 1, temp *= p[depth], num * (i + 1),
        i);  // 取一个该乘数，进行对下一个乘数的搜索
  }
  return;
}

int main() {
  while (scanf("%llu", &n) != EOF) {
    ans_num = 0;
    dfs(0, 1, 1, 60);
    printf("%llu\n", ans);
  }
  return 0;
}
```

参考资料与注释

1. Rui-Juan Jing, Marc Moreno-Maza, Delaram Talaashrafi, "Complexity Estimates for Fourier-Motzkin Elimination[11]", Journal of Functional Programming 16:2 (2006) pp 197-217.
2. 数论部分第一节： 素数与素性测试[12]
3. Miller-Rabin 与 Pollard-Rho 学习笔记 - Bill Yang's Blog[13]
4. Primality test - Wikipedia[14]
5. 桃子的算法笔记 —— 反素数详解 （acm/OI） [15]
6. The Rabin-Miller Primality Test[16]
7. Bach, Eric , "Explicit bounds for primality testing and related problems[17]", Mathematics of Computation, 55:191 (1990) pp 355–380.
8. Deterministic variant of the Miller-Rabin primality test[18]
9. Fermat pseudoprime - Wikipedia[19]
10. Carmichael number - Wikipedia[20]
11. Carmichael function - Wikipedia[21]
12. Carmichael Number -- from Wolfram MathWorld[22]
13. Carmichael's Lambda Function | Brilliant Math & Science Wiki[23]

[1] Korselt, A. R. (1899). "Problème chinois". *L'Intermédiaire des Mathématiciens*. **6**: 142–143.

[2] W. R. Alford; Andrew Granville; Carl Pomerance (1994). "There are Infinitely Many Carmichael Numbers". *Annals of Mathematics*. 140 (3): 703–722.

[3] Erdős, P. (1956). "On pseudoprimes and Carmichael numbers". *Publ. Math. Debrecen*. 4 (3–4): 201–206.

[4] PINCH, Richard GE. The Carmichael numbers up to 10^{20}.

[5] OEIS:A002997

[6] $O(k\log^2 n\log\log n\log\log\log n)$

[7] 广义 Riemann 猜想

[8] emirp

[9] Codeforces 27E. A number with a given number of divisors

[10] ZOJ - More Divisors

[11] Complexity Estimates for Fourier-Motzkin Elimination

[12] 数论部分第一节：素数与素性测试

[13] Miller-Rabin 与 Pollard-Rho 学习笔记 - Bill Yang's Blog

[14] Primality test - Wikipedia

[15] 桃子的算法笔记——反素数详解（acm/OI）

[16] The Rabin-Miller Primality Test

[17] Explicit bounds for primality testing and related problems

[18] Deterministic variant of the Miller-Rabin primality test

[19] Fermat pseudoprime - Wikipedia

[20] Carmichael number - Wikipedia

[21] Carmichael function - Wikipedia

[22] Carmichael Number – from Wolfram MathWorld

[23] Carmichael's Lambda Function | Brilliant Math & Science Wiki

9.12.3 最大公约数

定义

最大公约数即为 Greatest Common Divisor，常缩写为 gcd。

一组整数的公约数，是指同时是这组数中每一个数的约数的数。± 1 是任意一组整数的公约数。

一组整数的最大公约数，是指所有公约数里面最大的一个。

那么如何求最大公约数呢？我们先考虑两个数的情况。

欧几里得算法

过程

如果我们已知两个数 a 和 b，如何求出二者的最大公约数呢？

不妨设 $a > b$

我们发现如果 b 是 a 的约数，那么 b 就是二者的最大公约数。下面讨论不能整除的情况，即 $a = b \times q + r$，其中 $r < b$。

我们通过证明可以得到 $\gcd(a,b) = \gcd(b, a \bmod b)$，过程如下：

" 证明"

设 $a = bk + c$，显然有 $c = a \bmod b$。设 $d \mid a$，$d \mid b$，则 $c = a - bk, \frac{c}{d} = \frac{a}{d} - \frac{b}{d}k$。

由右边的式子可知 $\frac{c}{d}$ 为整数，即 $d \mid c$，所以对于 a, b 的公约数，它也会是 $b, a \bmod b$ 的公约数。

反过来也需要证明：

设 $d \mid b$，$d \mid (a \bmod b)$，我们还是可以像之前一样得到以下式子 $\frac{a \bmod b}{d} = \frac{a}{d} - \frac{b}{d}k, \frac{a \bmod b}{d} + \frac{b}{d}k = \frac{a}{d}$。

因为左边式子显然为整数，所以 $\frac{a}{d}$ 也为整数，即 $d \mid a$，所以 $b, a \bmod b$ 的公约数也是 a, b 的公约数。

既然两式公约数都是相同的，那么最大公约数也会相同。

所以得到式子 $\gcd(a,b) = \gcd(b, a \bmod b)$

既然得到了 $\gcd(a,b) = \gcd(b,r)$，这里两个数的大小是不会增大的，那么我们也就得到了关于两个数的最大公约数的一个递归求法。

实现

```cpp
// Version 1
int gcd(int a, int b) {
  if (b == 0) return a;
  return gcd(b, a % b);
}

// Version 2
int gcd(int a, int b) { return b == 0 ? a : gcd(b, a % b); }
```

```java
// Version 1
public int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a % b);
}

// Version 2
public int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}
```

```python
def gcd(a, b):
    if b == 0:
        return a
    return gcd(b, a % b)
```

递归至 `b == 0`（即上一步的 `a % b == 0`）的情况再返回值即可。

根据上述递归求法，我们也可以写出一个迭代求法：

```cpp
int gcd(int a, int b) {
  while (b != 0) {
    int tmp = a;
    a = b;
    b = tmp % b;
  }
  return a;
}
```

```java
public int gcd(int a, int b) {
    while(b != 0) {
        int tmp = a;
        a = b;
        b = tmp % b;
    }
    return a;
}
```

```python
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

上述算法都可被称作欧几里得算法（Euclidean algorithm）。

另外，对于 C++ 17，我们可以使用 `<numeric>`[2] 头中的 `std::gcd`[3] 与 `std::lcm`[4] 来求最大公约数和最小公倍数。

" 注意"

在部分编译器中，C++14 中可以用 `std::__gcd(a,b)` 函数来求最大公约数，但是其仅作为 `std::rotate` 的私有辅助函数。[1-1] 使用该函数可能会导致预期之外的问题，故一般情况下不推荐使用。

如果两个数 a 和 b 满足 $\gcd(a,b)=1$，我们称 a 和 b 互质。

性质

欧几里得算法的时间效率如何呢？下面我们证明，在输入为两个长为 n 的二进制整数时，欧几里得算法的时间复杂度为 $O(n)$。（换句话说，在默认 a,b 同阶的情况下，时间复杂度为 $O(\log\max(a,b))$。）

" 证明"

当我们求 $\gcd(a,b)$ 的时候，会遇到两种情况：

- $a<b$，这时候 $\gcd(a,b)=\gcd(b,a)$；
- $a\geq b$，这时候 $\gcd(a,b)=\gcd(b,a \bmod b)$，而对 a 取模会让 a 至少折半。这意味着这一过程最多发生 $O(\log a)=O(n)$ 次。

第一种情况发生后一定会发生第二种情况，因此第一种情况的发生次数一定**不多于**第二种情况的发生次数。从而我们最多递归 $O(n)$ 次就可以得出结果。

事实上，假如我们试着用欧几里得算法去求 斐波那契数列 相邻两项的最大公约数，会让该算法达到最坏复杂度。

更相减损术

大整数取模的时间复杂度较高，而加减法时间复杂度较低。针对大整数，我们可以用加减代替乘除求出最大公约数。

过程

已知两数 a 和 b，求 $\gcd(a,b)$。

不妨设 $a \geq b$，若 $a=b$，则 $\gcd(a,b)=a=b$。否则，$\forall d \mid a, d \mid b$，可以证明 $d \mid a-b$。

因此，a 和 b 的**所有**公因数都是 $a-b$ 和 b 的公因数，$\gcd(a,b)=\gcd(a-b,b)$。

Stein 算法的优化

如果 $a \gg b$，更相减损术的 $O(n)$ 复杂度将会达到最坏情况。

考虑一个优化，若 $2 \mid a, 2 \mid b$，$\gcd(a,b)=2\gcd\left(\frac{a}{2},\frac{b}{2}\right)$。

否则，若 $2 \mid a$（$2 \mid b$ 同理），因为 $2 \mid b$ 的情况已经讨论过了，所以 $2 \nmid b$。因此 $\gcd(a,b)=\gcd\left(\frac{a}{2},b\right)$。

优化后的算法（即 Stein 算法）时间复杂度是 $O(\log n)$。

" 证明"

若 $2 \mid a$ 或 $2 \mid b$，每次递归至少会将 a,b 之一减半。

否则，$2 \mid a-b$，回到了上一种情况。

算法最多递归 $O(\log n)$ 次。

实现

高精度模板见 高精度计算。

高精度运算需实现：减法、大小比较、左移、右移（可用低精乘除代替）、判断奇偶。

"C++"

```cpp
Big gcd(Big a, Big b) {
  // 记录 a 和 b 的公因数 2 出现次数
  int atimes = 0, btimes = 0;
  while (a % 2 == 0) {
    a >>= 1;
    atimes++;
  }
  while (b % 2 == 0) {
    b >>= 1;
    btimes++;
  }
  for (;;) {
    // a 和 b 公因数中的 2 已经计算过了，后面不可能出现 a,b 均为偶数的情况
    while (a % 2 == 0) {
      a >>= 1;
    }
    while (b % 2 == 0) {
      b >>= 1;
    }
    if (a == b) break;
    // 确保 a>=b
    if (a < b) swap(a, b);
    a -= b;
  }
  return a << min(atimes, btimes);
}
```

多个数的最大公约数

那怎么求多个数的最大公约数呢？显然答案一定是每个数的约数，那么也一定是每相邻两个数的约数。我们采用归纳法，可以证明，每次取出两个数求出答案后再放回去，不会对所需要的答案造成影响。

最小公倍数

接下来我们介绍如何求解最小公倍数（Least Common Multiple, LCM）。

定义

一组整数的公倍数，是指同时是这组数中每一个数的倍数的数。0 是任意一组整数的公倍数。

一组整数的最小公倍数，是指所有正的公倍数里面，最小的一个数。

两个数

设 $a = p_1^{k_{a_1}} p_2^{k_{a_2}} \cdots p_s^{k_{a_s}}$，$b = p_1^{k_{b_1}} p_2^{k_{b_2}} \cdots p_s^{k_{b_s}}$

我们发现，对于 a 和 b 的情况，二者的最大公约数等于

$p_1^{\min(k_{a_1},k_{b_1})} p_2^{\min(k_{a_2},k_{b_2})} \cdots p_s^{\min(k_{a_s},k_{b_s})}$

最小公倍数等于

$p_1^{\max(k_{a_1},k_{b_1})} p_2^{\max(k_{a_2},k_{b_2})} \cdots p_s^{\max(k_{a_s},k_{b_s})}$

由于 $k_a + k_b = \max(k_a, k_b) + \min(k_a, k_b)$

所以得到结论是 $\gcd(a, b) \times \operatorname{lcm}(a, b) = a \times b$

要求两个数的最小公倍数，先求出最大公约数即可。

多个数

可以发现，当我们求出两个数的 gcd 时，求最小公倍数是 $O(1)$ 的复杂度。那么对于多个数，我们其实没有必要求一个共同的最大公约数再去处理，最直接的方法就是，当我们算出两个数的 gcd，或许在求多个数的 gcd 时候，我们将它放入序列对后面的数继续求解，那么，我们转换一下，直接将最小公倍数放入序列即可。

扩展欧几里得算法

扩展欧几里得算法（Extended Euclidean algorithm, EXGCD），常用于求 $ax + by = \gcd(a, b)$ 的一组可行解。

过程

设

$ax_1 + by_1 = \gcd(a, b)$

$bx_2 + (a \bmod b)y_2 = \gcd(b, a \bmod b)$

由欧几里得定理可知：$\gcd(a, b) = \gcd(b, a \bmod b)$

所以 $ax_1 + by_1 = bx_2 + (a \bmod b)y_2$

又因为 $a \bmod b = a - (\lfloor \frac{a}{b} \rfloor \times b)$

所以 $ax_1 + by_1 = bx_2 + (a - (\lfloor \frac{a}{b} \rfloor \times b))y_2$

$ax_1 + by_1 = ay_2 + bx_2 - \lfloor \frac{a}{b} \rfloor \times by_2 = ay_2 + b(x_2 - \lfloor \frac{a}{b} \rfloor y_2)$

因为 $a = a, b = b$，所以 $x_1 = y_2, y_1 = x_2 - \lfloor \frac{a}{b} \rfloor y_2$

将 x_2, y_2 不断代入递归求解直至 gcd（最大公约数，下同）为 `0` 递归 `x=1,y=0` 回去求解。

实现

```cpp
int Exgcd(int a, int b, int &x, int &y) {
  if (!b) {
    x = 1;
    y = 0;
    return a;
  }
  int d = Exgcd(b, a % b, x, y);
  int t = x;
  x = y;
  y = t - (a / b) * y;
  return d;
}
```

```python
def Exgcd(a, b):
    if b == 0:
        return a, 1, 0
    d, x, y = Exgcd(b, a % b)
    return d, y, x - (a // b) * y
```

函数返回的值为 gcd，在这个过程中计算 x,y 即可。

值域分析

$ax+by=\gcd(a,b)$ 的解有无数个，显然其中有的解会爆 long long。

万幸的是，若 $b\neq 0$，扩展欧几里得算法求出的可行解必有 $|x|\le b,|y|\le a$。

下面给出这一性质的证明。

" 证明"

- $\gcd(a,b)=b$ 时，$a \bmod b=0$，必在下一层终止递归。

  得到 $x_1=0,y_1=1$，显然 $a,b\ge 1\ge |x_1|,|y_1|$。

- $\gcd(a,b)\neq b$ 时，设 $|x_2|\le (a \bmod b),|y_2|\le b$。

  因为 $x_1=y_2,y_1=x_2-\left\lfloor\frac{a}{b}\right\rfloor y_2$

  所以 $|x_1|=|y_2|\le b,|y_1|\le |x_2|+|\left\lfloor\frac{a}{b}\right\rfloor y_2|\le (a \bmod b)+\left\lfloor\frac{a}{b}\right\rfloor|y_2|$

  $\le a-\left\lfloor\frac{a}{b}\right\rfloor b+\left\lfloor\frac{a}{b}\right\rfloor|y_2|\le a-\left\lfloor\frac{a}{b}\right\rfloor(b-|y_2|)$

  $a \bmod b=a-\left\lfloor\frac{a}{b}\right\rfloor b\le a-\left\lfloor\frac{a}{b}\right\rfloor(b-|y_2|)\le a$

  因此 $|x_1|\le b,|y_1|\le a$ 成立。

迭代法编写扩展欧几里得算法

首先，当 $x=1$，$y=0$，$x_1=0$，$y_1=1$ 时，显然有：

$$\begin{cases} ax+by & =a \\ ax_1+by_1 & =b \end{cases}$$

成立。

已知 $a \bmod b=a-(\lfloor\frac{a}{b}\rfloor\times b)$，下面令 $q=\lfloor\frac{a}{b}\rfloor$。参考迭代法求 gcd，每一轮的迭代过程可以表示为：

$$(a,b)\to(b,a-qb)$$

将迭代过程中的 a 替换为 $ax+by=a$，b 替换为 $ax_1+by_1=b$，可以得到：

$$
\begin{aligned}
&\begin{cases} ax+by &= a \\ ax_1+by_1 &= b \end{cases} \\
\to &\begin{cases} ax_1+by_1 &= b \\ a(x-qx_1)+b(y-qy_1) &= a-qb \end{cases}
\end{aligned}
$$

据此就可以得到迭代法求 exgcd。

因为迭代的方法避免了递归，所以代码运行速度将比递归代码快一点。

```
int gcd(int a, int b, int& x, int& y) {
  x = 1, y = 0;
  int x1 = 0, y1 = 1, a1 = a, b1 = b;
  while (b1) {
    int q = a1 / b1;
    tie(x, x1) = make_tuple(x1, x - q * x1);
    tie(y, y1) = make_tuple(y1, y - q * y1);
    tie(a1, b1) = make_tuple(b1, a1 - q * b1);
  }
  return a1;
}
```

如果你仔细观察 a_1 和 b_1，你会发现，他们在迭代版本的欧几里德算法中取值完全相同，并且以下公式无论何时（在 while 循环之前和每次迭代结束时）都是成立的：$x\cdot a+y\cdot b=a_1$ 和 $x_1\cdot a+y_1\cdot b=b_1$。因此，该算法肯定能正确计算出 gcd。

最后我们知道 a_1 就是要求的 gcd，有 $x\cdot a+y\cdot b=g$。

矩阵的解释

对于正整数 a 和 b 的一次辗转相除即 $\gcd(a,b)=\gcd(b,a \bmod b)$ 使用矩阵表示如

$$
\begin{bmatrix} b \\ a \bmod b \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -\lfloor a/b \rfloor \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}
$$

其中向下取整符号 $\lfloor c \rfloor$ 表示不大于 c 的最大整数。我们定义变换 $\begin{bmatrix} a \\ b \end{bmatrix} \mapsto \begin{bmatrix} 0 & 1 \\ 1 & -\lfloor a/b \rfloor \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$。

易发现欧几里得算法即不停应用该变换，有

$$
\begin{bmatrix} \gcd(a,b) \\ 0 \end{bmatrix} = \left(\cdots \begin{bmatrix} 0 & 1 \\ 1 & -\lfloor a/b \rfloor \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) \begin{bmatrix} a \\ b \end{bmatrix}
$$

令

$$
\begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix} = \cdots \begin{bmatrix} 0 & 1 \\ 1 & -\lfloor a/b \rfloor \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}
$$

那么

$$
\begin{bmatrix} \gcd(a,b) \\ 0 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}
$$

满足 $a\cdot x_1+b\cdot x_2=\gcd(a,b)$ 即扩展欧几里得算法，注意在最后乘了一个单位矩阵不会影响结果，提示我们可以在开始时维护一个 2×2 的单位矩阵编写更简洁的迭代方法如

```cpp
int exgcd(int a, int b, int &x, int &y) {
  int x1 = 1, x2 = 0, x3 = 0, x4 = 1;
  while (b != 0) {
    int c = a / b;
    std::tie(x1, x2, x3, x4, a, b) =
        std::make_tuple(x3, x4, x1 - x3 * c, x2 - x4 * c, b, a - b * c);
  }
  x = x1, y = x2;
  return a;
}
```

这种表述相较于递归更简单。

应用

- 10104 - Euclid Problem[5]
- GYM - (J) once upon a time[6]
- UVa - 12775 - Gift Dilemma[7]

参考资料与链接

[1] libstdc++: std Namespace Reference [1-1] [1-2]

[2] `<numeric>`

[3] `std::gcd`

[4] `std::lcm`

[5] 10104 - Euclid Problem

[6] GYM - (J) once upon a time

[7] UVa - 12775 - Gift Dilemma

9.12.4 数论分块

数论分块可以快速计算一些含有除法向下取整的和式（即形如 $\sum_{i=1}^{n} f(i)g(\left\lfloor\frac{n}{i}\right\rfloor)$ 的和式）。当可以在 $O(1)$ 内计算 $f(r)-f(l)$ 或已经预处理出 f 的前缀和时，数论分块就可以在 $O(\sqrt{n})$ 的时间内计算上述和式的值。

它主要利用了富比尼定理（Fubini's theorem），将 $\left\lfloor\frac{n}{i}\right\rfloor$ 相同的数打包同时计算。

" 富比尼定理"

又称「算两次」，以意大利数学家圭多・富比尼（Guido Fubini）命名。富比尼定理的积分形式：只要二重积分 $\int\int |f(x,y)|dxdy$ 有界，则可以逐次计算二重积分，并且可以交换逐次积分的顺序。积分号也是特殊的求和号，因此在一般求和中，富比尼定理往往呈现为更换计数顺序，即交换两个求和号。组合数学中的富比尼定理表现为，用两种不同的方法计算同一个量，从而建立相等关系。

例如这里的双曲线下整点的图片：

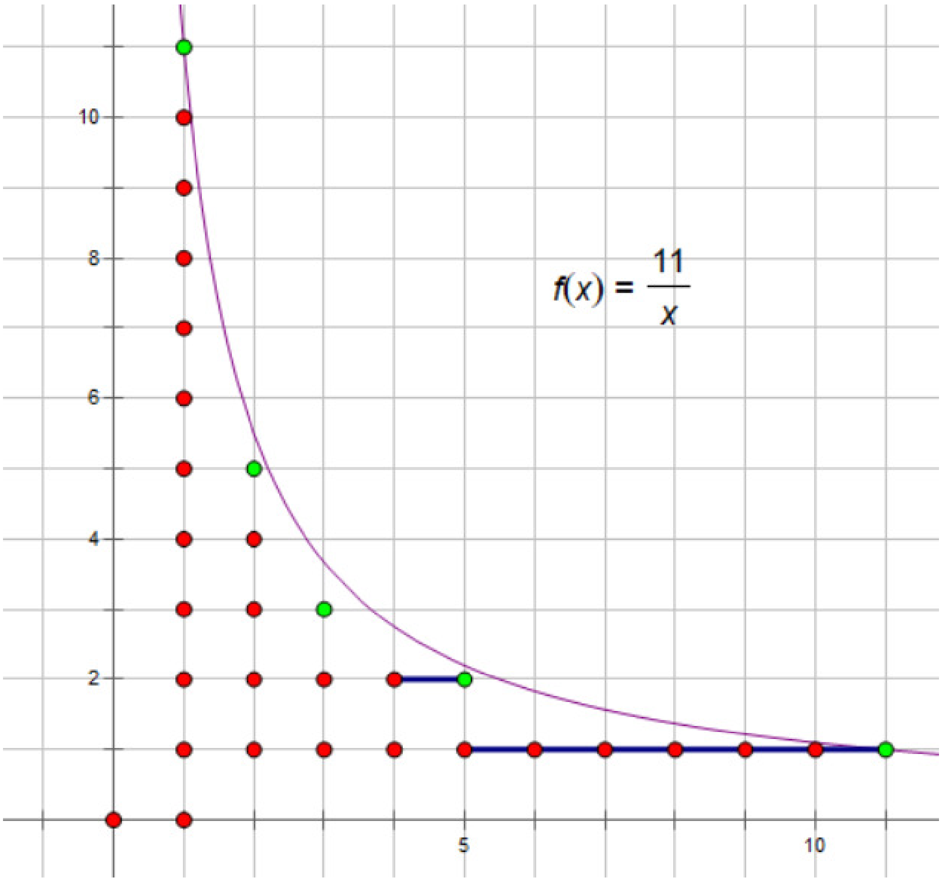


图 9.7 双曲线下整点

图中共分为了 5 块，这 5 块整点的最大纵坐标都相同。如果统计整点的个数，可以从纵向计数改为横向计数，直接计算 5 个矩形即可。

引理 1

$$\forall a,b,c \in \mathbb{Z}, \left\lfloor \frac{a}{bc} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{a}{b} \right\rfloor}{c} \right\rfloor$$

略证：

$$\begin{aligned} & \frac{a}{b} = \left\lfloor \frac{a}{b} \right\rfloor + r (0 \le r < 1) \\ \Longrightarrow & \left\lfloor \frac{a}{bc} \right\rfloor = \left\lfloor \frac{a}{b} \cdot \frac{1}{c} \right\rfloor = \left\lfloor \frac{1}{c} \left(\left\lfloor \frac{a}{b} \right\rfloor + r \right) \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{a}{b} \right\rfloor}{c} + \frac{r}{c} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{a}{b} \right\rfloor}{c} \right\rfloor \end{aligned}$$

□

" 关于证明最后的小方块"

QED 是拉丁词组「Quod Erat Demonstrandum」(这就是所要证明的) 的缩写，代表证明完毕。现在的 QED 符号通常是 ■ 或者 □。(维基百科[1])

引理 2

$$\forall n \in \mathbb{N}_+, \left| \left\{ \lfloor \frac{n}{d} \rfloor \mid d \in \mathbb{N}_+, d \le n \right\} \right| \le \lfloor 2\sqrt{n} \rfloor$$

$|V|$ 表示集合 V 的元素个数

略证：

对于 $d \le \lfloor\sqrt{n}\rfloor$，$\lfloor\frac{n}{d}\rfloor$ 有 $\lfloor\sqrt{n}\rfloor$ 种取值

对于 $d > \lfloor\sqrt{n}\rfloor$，有 $\lfloor\frac{n}{d}\rfloor \le \lfloor\sqrt{n}\rfloor$，也只有 $\lfloor\sqrt{n}\rfloor$ 种取值

综上，得证

数论分块结论

对于常数 n，使得式子

$$\left\lfloor\frac{n}{i}\right\rfloor = \left\lfloor\frac{n}{j}\right\rfloor$$

成立的最大的满足 $i \le j \le n$ 的 j 的值为 $\left\lfloor\frac{n}{\lfloor\frac{n}{i}\rfloor}\right\rfloor$。即值 $\left\lfloor\frac{n}{i}\right\rfloor$ 所在的块的右端点为 $\left\lfloor\frac{n}{\lfloor\frac{n}{i}\rfloor}\right\rfloor$。

" **证明**"

令 $k = \left\lfloor\frac{n}{i}\right\rfloor$，可以知道 $k \le \frac{n}{i}$。

$$\therefore \left\lfloor\frac{n}{k}\right\rfloor \ge \left\lfloor\frac{n}{\frac{n}{i}}\right\rfloor = \lfloor i\rfloor = i$$

$$\therefore j = \max \text{满足条件的所有} i = i_{\max} = \left\lfloor\frac{n}{k}\right\rfloor = \left\lfloor\frac{n}{\left\lfloor\frac{n}{i}\right\rfloor}\right\rfloor \square$$

过程

数论分块的过程大概如下：考虑和式

$\sum_{i=1}^{n} f(i)\left\lfloor\frac{n}{i}\right\rfloor$

那么由于我们可以知道 $\left\lfloor\frac{n}{i}\right\rfloor$ 的值成一个块状分布（就是同样的值都聚集在连续的块中），那么就可以用数论分块加速计算，降低时间复杂度。

利用上述结论，我们先求出 $f(i)$ 的**前缀和**（记作 $s(i) = \sum_{j=1}^{i} f(j)$），然后每次以 $[l, r] = [l, \left\lfloor\frac{n}{\lfloor\frac{n}{i}\rfloor}\right\rfloor]$ 为一块，分块求出贡献累加到结果中即可。

伪代码如下：

$$\begin{array}{l}
\text{获取 } f(i) \text{ 函数的前缀和，记为 } s(i). \\
l \leftarrow 1 \\
r \leftarrow 0 \\
\mathit{result} \leftarrow 0 \\
\textbf{while } l \le n \textbf{ do}: \\
\quad r \leftarrow \left\lfloor\dfrac{n}{\lfloor n/l\rfloor}\right\rfloor \\
\quad \mathit{result} \leftarrow \mathit{result} + [s(r) - s(l-1)] \times \left\lfloor\dfrac{n}{l}\right\rfloor \\
\quad l \leftarrow r + 1 \\
\textbf{end while}
\end{array}$$

最终得到的 $result$ 即为所求的和式。

" **例题**：UVa11526 H(n)[2]"

题意：T 组数据，每组一个整数 n。对于每组数据，输出 $\sum_{i=1}^{n}\left\lfloor\frac{n}{i}\right\rfloor$。

思路：如上推导，对于每一块相同的 $\left\lfloor\frac{n}{i}\right\rfloor$ 一起计算。时间复杂度为 $O(T\sqrt{n})$。

" 参考实现"

```
long long H(int n) {
  long long res = 0;  // 储存结果
  int l = 1, r;       // 块左端点与右端点
  while (l <= n) {
    r = n / (n / l);  // 计算当前块的右端点
    res += (r - l + 1) * 1LL *
           (n / l);  // 累加这一块的贡献到结果中。乘上 1LL 防止溢出
    l = r + 1;  // 左端点移到下一块
  }
  return res;
}
```

"N 维数论分块"

求含有 $\left\lfloor\frac{a_1}{i}\right\rfloor$、$\left\lfloor\frac{a_2}{i}\right\rfloor\cdots\left\lfloor\frac{a_n}{i}\right\rfloor$ 的和式时，数论分块右端点的表达式从一维的 $\left\lfloor\frac{n}{i}\right\rfloor$ 变为 $\min\limits_{j=1}^{n}\{\left\lfloor\frac{a_j}{i}\right\rfloor\}$，即对于每一个块的右端点取最小（最接近左端点）的那个作为整体的右端点。可以借助下图理解：

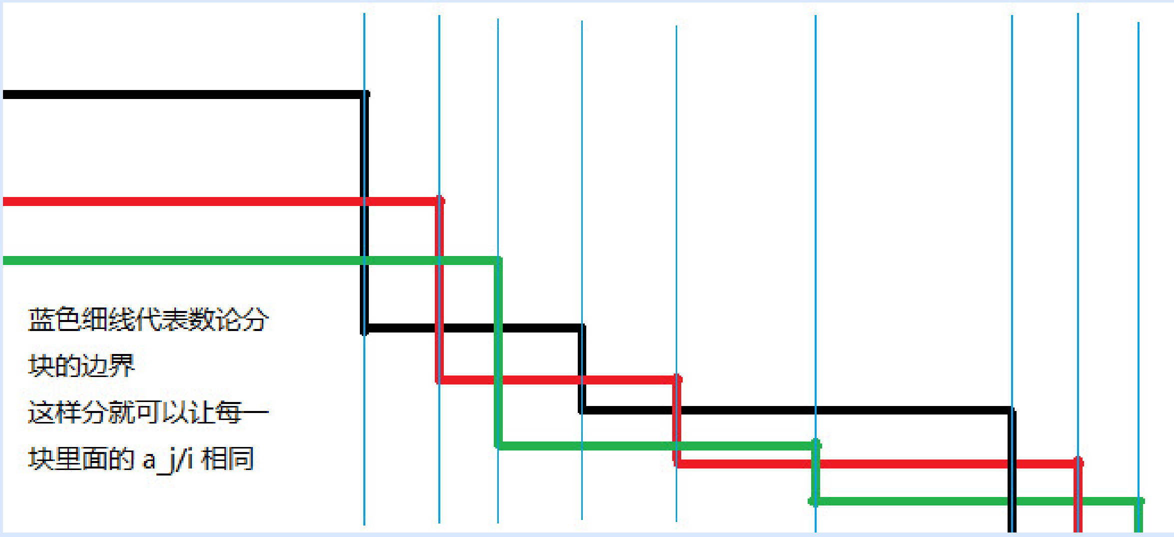


图 9.8 多维数论分块图解

一般我们用的较多的是二维形式，此时可将代码中 `r = n / (n / i)` 替换成 `r = min(n / (n / i), m / (m / i))`。

习题

1. CQOI2007 余数求和[3]（需要一点转化和特判）
2. UVa11526 H(n)[2]（几乎可以当做模板题）
3. POI2007 ZAP-Queries[4]（数论分块一般配合 莫比乌斯反演 用以进一步降低复杂度；本题需要用到 $[n=1]=\sum_{d|n}\mu(n)$ 这一条莫反结论）

参考资料与注释

[1] 维基百科

[2] UVa11526 H(n) [2-1] [2-2]

[3] CQOI2007 余数求和

[4] POI2007 ZAP-Queries

9.12.5　欧拉函数

Authors: iamtwz, Chrogeek, Enter-tainer, StudyingFather, aofall, CCXXXI, CoelacanthusHex, frank-xjh, Great-designer, greyqz, guodong2005, henrytbtrue, Ir1d, kZime, lihaoyu1234, Marcythm, MegaOwIer, Menci, nalemy, orzAtalod, ouuan, Persdre, segment-tree, ShaoChenHeng, shuzhouliu, sshwy, Struggler-q, Tiphereth-A, TrisolarisHD, Xeonacid, yuhuoji

定义

欧拉函数（Euler's totient function），即 $\varphi(n)$，表示的是小于等于 n 和 n 互质的数的个数。

比如说 $\varphi(1)=1$。

当 n 是质数的时候，显然有 $\varphi(n)=n-1$。

性质

- 欧拉函数是积性函数。
  积性是什么意思呢？如果有 $\gcd(a,b)=1$，那么 $\varphi(a\times b)=\varphi(a)\times\varphi(b)$。
  特别地，当 n 是奇数时 $\varphi(2n)=\varphi(n)$。
- $n=\sum_{d\mid n}\varphi(d)$。

" 证明"

利用 莫比乌斯反演 相关知识可以得出。

也可以这样考虑：如果 $\gcd(k,n)=d$，那么 $\gcd(\dfrac{k}{d},\dfrac{n}{d})=1,(k<n)$。

如果我们设 $f(x)$ 表示 $\gcd(k,n)=x$ 的数的个数，那么 $n=\sum_{i=1}^{n}f(i)$。

根据上面的证明，我们发现，$f(x)=\varphi(\dfrac{n}{x})$，从而 $n=\sum_{d\mid n}\varphi(\dfrac{n}{d})$。注意到约数 d 和 $\dfrac{n}{d}$ 具有对称性，所以上式化为 $n=\sum_{d\mid n}\varphi(d)$。

- 若 $n=p^k$，其中 p 是质数，那么 $\varphi(n)=p^k-p^{k-1}$。（根据定义可知）
- 由唯一分解定理，设 $n=\prod_{i=1}^{s}p_i^{k_i}$，其中 p_i 是质数，有 $\varphi(n)=n\times\prod_{i=1}^{s}\dfrac{p_i-1}{p_i}$。

" 证明"

– 引理：设 p 为任意质数，那么 $\varphi(p^k)=p^{k-1}\times(p-1)$。

证明：显然对于从 1 到 p^k 的所有数中，除了 p^{k-1} 个 p 的倍数以外其它数都与 p^k 互素，故 $\varphi(p^k)=p^k-p^{k-1}=p^{k-1}\times(p-1)$，证毕。

接下来我们证明 $\varphi(n) = n \times \prod_{i=1}^{s} \dfrac{p_i - 1}{p_i}$。由唯一分解定理与 $\varphi(x)$ 函数的积性

$$\begin{aligned}\varphi(n) &= \prod_{i=1}^{s} \varphi(p_i^{k_i}) \\ &= \prod_{i=1}^{s} (p_i - 1) \times {p_i}^{k_i - 1} \\ &= \prod_{i=1}^{s} {p_i}^{k_i} \times (1 - \frac{1}{p_i}) \\ &= n \prod_{i=1}^{s} (1 - \frac{1}{p_i}) \qquad \square\end{aligned}$$

实现

如果只要求一个数的欧拉函数值，那么直接根据定义质因数分解的同时求就好了。这个过程可以用 Pollard Rho 算法优化。

```cpp
#include <cmath>

int euler_phi(int n) {
  int m = int(sqrt(n + 0.5));
  int ans = n;
  for (int i = 2; i <= m; i++)
    if (n % i == 0) {
      ans = ans / i * (i - 1);
      while (n % i == 0) n /= i;
    }
  if (n > 1) ans = ans / n * (n - 1);
  return ans;
}
```

```python
import math
def euler_phi(n):
    m = math.isqrt(n + 0.5)
    ans = n
    for i in range(2, m + 1):
        if n % i == 0:
            ans = ans // i * (i - 1)
            while n % i == 0:
                n = n // i
    if n > 1:
        ans = ans // n * (n - 1)
    return ans
```

注：如果将上面的程序改成如下形式，会提升一点效率：

```cpp
#include <cmath>

int euler_phi(int n) {
  int ans = n;
  for (int i = 2; i * i <= n; i++)
    if (n % i == 0) {
      ans = ans / i * (i - 1);
      while (n % i == 0) n /= i;
```

```
    }
  if (n > 1) ans = ans / n * (n - 1);
  return ans;
}
```

```python
import math
def euler_phi(n):
    ans = n
    for i in range(2, math.isqrt(n) + 1):
        if n % i == 0:
            ans = ans // i * (i - 1)
            while n % i == 0:
                n = n // i
    if n > 1:
        ans = ans // n * (n - 1)
    return ans
```

如果是多个数的欧拉函数值，可以利用后面会提到的线性筛法来求得。详见：筛法求欧拉函数

欧拉定理

与欧拉函数紧密相关的一个定理就是欧拉定理。其描述如下：

若 $\gcd(a, m) = 1$，则 $a^{\varphi(m)} \equiv 1 \pmod{m}$。

扩展欧拉定理

当然也有扩展欧拉定理

$$a^b \equiv \begin{cases} a^{b \bmod \varphi(p)}, & \gcd(a,\,p) = 1 \\ a^b, & \gcd(a,\,p) \ne 1,\, b < \varphi(p) \\ a^{b \bmod \varphi(p) + \varphi(p)}, & \gcd(a,\,p) \ne 1,\, b \ge \varphi(p) \end{cases} \pmod{p}$$

证明和**习题**详见 欧拉定理

9.12.6 筛法

Authors: inkydragon, TravorLZH, YOYO-UIAT, wood3, shuzhouliu, Mr-Python-in-China, HeRaNO

素数筛法

引入

如果我们想要知道小于等于 n 有多少个素数呢?

一个自然的想法是对于小于等于 n 的每个数进行一次质数检验。这种暴力的做法显然不能达到最优复杂度。

埃拉托斯特尼筛法

过程

考虑这样一件事情：对于任意一个大于 1 的正整数 n，那么它的 x 倍就是合数（$x > 1$）。利用这个结论，我们可以避免很多次不必要的检测。

如果我们从小到大考虑每个数，然后同时把当前这个数的所有（比自己大的）倍数记为合数，那么运行结束的时候没有被标记的数就是素数了。

实现

```cpp
vector<int> prime;
bool is_prime[N];

void Eratosthenes(int n) {
  is_prime[0] = is_prime[1] = false;
  for (int i = 2; i <= n; ++i) is_prime[i] = true;
  for (int i = 2; i <= n; ++i) {
    if (is_prime[i]) {
      prime.push_back(i);
      if ((long long)i * i > n) continue;
      for (int j = i * i; j <= n; j += i)
        // 因为从 2 到 i - 1 的倍数我们之前筛过了，这里直接从 i
        // 的倍数开始，提高了运行速度
        is_prime[j] = false;  // 是 i 的倍数的均不是素数
    }
  }
}
```

```python
prime = []
is_prime = [False] * N

def Eratosthenes(n):
    is_prime[0] = is_prime[1] = False
    for i in range(2, n + 1):
        is_prime[i] = True
    for i in range(2, n + 1):
        if is_prime[i]:
            prime.append(i)
            if i * i > n:
                continue
            for j in range(i * i, n + 1, i):
                is_prime[j] = False
```

以上为 **Eratosthenes 筛法**（埃拉托斯特尼筛法，简称埃氏筛法），时间复杂度是 $O(n\log\log n)$。

" 证明"

现在我们就来看看推导过程：

如果每一次对数组的操作花费 1 个单位时间，则时间复杂度为：

$$O\left(\sum_{k=1}^{\pi(n)}\frac{n}{p_k}\right)=O\left(n\sum_{k=1}^{\pi(n)}\frac{1}{p_k}\right)$$

其中 p_k 表示第 k 小的素数，$\pi(n)$ 表示 $\le n$ 的素数个数。$\sum_{k=1}^{\pi(n)}$ 表示第一层 for 循环，其中累加上界 $\pi(n)$ 为 `if (prime[i])` 进入 true 分支的次数；$\frac{n}{p_k}$ 表示第二层 for 循环的执行次数。

根据 Mertens 第二定理，存在常数 B_1 使得：

$$\sum_{k=1}^{\pi(n)}\frac{1}{p_k}=\log\log n+B_1+O\left(\frac{1}{\log n}\right)$$

所以 **Eratosthenes 筛法**的时间复杂度为 $O(n\log\log n)$。接下来我们证明 Mertens 第二定理的弱化版本 $\sum_{k\le\pi(n)}1/p_k=O(\log\log n)$：

根据 $\pi(n)=\Theta(n/\log n)$，可知第 n 个素数的大小为 $\Theta(n\log n)$。于是就有

$$
\begin{aligned}
\sum_{k=1}^{\pi(n)}\frac{1}{p_k} &= O\left(\sum_{k=2}^{\pi(n)}\frac{1}{k\log k}\right)\\
&= O\left(\int_2^{\pi(n)}\frac{\mathrm{d}x}{x\log x}\right)\\
&= O(\log\log\pi(n)) = O(\log\log n)
\end{aligned}
$$

当然，上面的做法效率仍然不够高效，应用下面几种方法可以稍微提高算法的执行效率。

筛至平方根

显然，要找到直到 n 为止的所有素数，仅对不超过 $\sqrt{n}$ 的素数进行筛选就足够了。

```cpp
vector<int> prime;
bool is_prime[N];

void Eratosthenes(int n) {
  is_prime[0] = is_prime[1] = false;
  for (int i = 2; i <= n; ++i) is_prime[i] = true;
  // i * i <= n 说明 i <= sqrt(n)
  for (int i = 2; i * i <= n; ++i) {
    if (is_prime[i])
      for (int j = i * i; j <= n; j += i) is_prime[j] = false;
  }
  for (int i = 2; i <= n; ++i)
    if (is_prime[i]) prime.push_back(i);
}
```

```python
prime = []
is_prime = [False] * N

def Eratosthenes(n):
    is_prime[0] = is_prime[1] = False
    for i in range(2, n + 1):
        is_prime[i] = True
    # 让 i 循环到 <= sqrt(n)
    for i in range(2, isqrt(n) + 1): # `isqrt` 是 Python 3.8 新增的函数
        if is_prime[i]:
            for j in range(i * i, n + 1, i):
                is_prime[j] = False
    for i in range(2, n + 1):
        if is_prime[i]:
            prime.append(i)
```

这种优化不会影响渐进时间复杂度，实际上重复以上证明，我们将得到 $n\ln\ln\sqrt{n}+o(n)$，根据对数的性质，它们的渐进相同，但操作次数会明显减少。

只筛奇数

因为除 2 以外的偶数都是合数，所以我们可以直接跳过它们，只用关心奇数就好。

首先，这样做能让我们内存需求减半；其次，所需的操作大约也减半。

减少内存的占用

我们注意到筛选时只需要 `bool` 类型的数组。`bool` 数组的一个元素一般占用 1 字节（即 8 比特），但是存储一个布尔值只需要 1 个比特就足够了。

我们可以使用 位运算 的相关知识，将每个布尔值压到一个比特位中，这样我们仅需使用 n 比特（即 $\frac{n}{8}$ 字节）而非 n 字节，可以显著减少内存占用。

但是，这种称为**位级压缩**的方法会使这些位的操作复杂化。任何位上的读写操作都需要多次算术运算，最终会使算法变慢。因此，这种方法只有在 n 特别大，以至于我们不能再分配内存时才合理。在这种情况下，我们将牺牲效率，通过显著降低算法速度以节省内存（减小到原来的 $\frac{n}{8}$）。

值得一提的是，存在自动执行位级压缩的数据结构，如 C++ 中的 `vector<bool>` 和 `bitset<>`（参见 bitset: 与埃氏筛结合）。

分块筛选

由优化「筛至平方根」可知，不需要一直保留整个 `is_prime[1...n]` 数组。为了进行筛选，只保留到 $\sqrt{n}$ 的素数就足够了，即 `prime[1...sqrt(n)]`。并将整个范围分成块，每个块分别进行筛选。这样，我们就不必同时在内存中保留多个块，而且 CPU 可以更好地处理缓存。

设 s 是一个常数，它决定了块的大小，那么我们就有了 $\lceil \frac{n}{s} \rceil$ 个块，而块 $k(k = 0 \ldots \lfloor \frac{n}{s} \rfloor)$ 包含了区间 $[ks, ks+s-1]$ 中的数字。我们可以依次处理块，也就是说，对于每个块 k，我们将遍历所有质数（从 1 到 $\sqrt{n}$）并使用它们进行筛选。

值得注意的是，我们在处理第一个数字时需要稍微修改一下策略：首先，应保留 $[1, \sqrt{n}]$ 中的所有的质数；第二，数字 0 和 1 应该标记为非素数。在处理最后一个块时，不应该忘记最后一个数字 n 并不一定位于块的末尾。

以下实现使用块筛选来计算小于等于 n 的质数数量。

" 实现"

```cpp
int count_primes(int n) {
  const int S = 10000;
  vector<int> primes;
  int nsqrt = sqrt(n);
  vector<char> is_prime(nsqrt + 1, true);
  for (int i = 2; i <= nsqrt; i++) {
    if (is_prime[i]) {
      primes.push_back(i);
      for (int j = i * i; j <= nsqrt; j += i) is_prime[j] = false;
    }
  }
  int result = 0;
  vector<char> block(S);
  for (int k = 0; k * S <= n; k++) {
    fill(block.begin(), block.end(), true);
    int start = k * S;
    for (int p : primes) {
      int start_idx = (start + p - 1) / p;
      int j = max(start_idx, p) * p - start;
      for (; j < S; j += p) block[j] = false;
    }
    if (k == 0) block[0] = block[1] = false;
    for (int i = 0; i < S && start + i <= n; i++) {
      if (block[i]) result++;
    }
  }
  return result;
}
```

分块筛法的渐进时间复杂度与埃氏筛法是一样的（除非块非常小），但是所需的内存将缩小为 $O(\sqrt{n}+S)$，并且有更好的缓存结果。另一方面，对于每一对块和区间 $[1,\sqrt{n}]$ 中的素数都要进行除法，而对于较小的块来说，这种情况要糟糕得多。因此，在选择常数 S 时要保持平衡。

块大小 S 取 10^4 到 10^5 之间，可以获得最佳的速度。

线性筛法

埃氏筛法仍有优化空间，它会将一个合数重复多次标记。有没有什么办法省掉无意义的步骤呢？答案是肯定的。

如果能让每个合数都只被标记一次，那么时间复杂度就可以降到 $O(n)$ 了。

" 实现"

```cpp
vector<int> pri;
bool not_prime[N];

void pre(int n) {
  for (int i = 2; i <= n; ++i) {
    if (!not_prime[i]) {
      pri.push_back(i);
    }
    for (int pri_j : pri) {
      if (i * pri_j > n) break;
      not_prime[i * pri_j] = true;
      if (i % pri_j == 0) {
        // i % pri_j == 0
        // 换言之，i 之前被 pri_j 筛过了
        // 由于 pri 里面质数是从小到大的，所以 i 乘上其他的质数的结果一定会被
        // pri_j 的倍数筛掉，就不需要在这里先筛一次，所以这里直接 break
        // 掉就好了
        break;
      }
    }
  }
}
```

```python
pri = []
not_prime = [False] * N

def pre(n):
    for i in range(2, n + 1):
        if not not_prime[i]:
            pri.append(i)
        for pri_j in pri:
            if i * pri_j > n:
                break
            not_prime[i * pri_j] = True
            if i % pri_j == 0:
                """
                i % pri_j == 0
                换言之，i 之前被 pri_j 筛过了
                由于 pri 里面质数是从小到大的，所以 i 乘上其他的质数的结果一定会被
                pri_j 的倍数筛掉，就不需要在这里先筛一次，所以这里直接 break
                掉就好了
                """
```

```
                break
```

上面的这种**线性筛法**也称为 **Euler 筛法**（欧拉筛法）。

note

注意到筛法求素数的同时也得到了每个数的最小质因子。

筛法求欧拉函数

注意到在线性筛中，每一个合数都是被最小的质因子筛掉。比如设 p_1 是 n 的最小质因子，$n' = \frac{n}{p_1}$，那么线性筛的过程中 n 通过 $n' \times p_1$ 筛掉。

观察线性筛的过程，我们还需要处理两个部分，下面对 $n' \bmod p_1$ 分情况讨论。

如果 $n' \bmod p_1 = 0$，那么 n' 包含了 n 的所有质因子。

$$\varphi(n) = n \times \prod_{i=1}^{s} \frac{p_i - 1}{p_i}$$

$$= p_1 \times n' \times \prod_{i=1}^{s} \frac{p_i - 1}{p_i}$$

$$= p_1 \times \varphi(n')$$

那如果 $n' \bmod p_1 \neq 0$ 呢，这时 n' 和 p_1 是互质的，根据欧拉函数性质，我们有：

$$\varphi(n) = \varphi(p_1) \times \varphi(n')$$

$$= (p_1 - 1) \times \varphi(n')$$

实现

```cpp
vector<int> pri;
bool not_prime[N];
int phi[N];

void pre(int n) {
  phi[1] = 1;
  for (int i = 2; i <= n; i++) {
    if (!not_prime[i]) {
      pri.push_back(i);
      phi[i] = i - 1;
    }
    for (int pri_j : pri) {
      if (i * pri_j > n) break;
      not_prime[i * pri_j] = true;
      if (i % pri_j == 0) {
        phi[i * pri_j] = phi[i] * pri_j;
        break;
      }
      phi[i * pri_j] = phi[i] * phi[pri_j];
    }
  }
}
```

```python
pri = []
not_prime = [False] * N
phi = [0] * N

def pre(n):
    phi[1] = 1
    for i in range(2, n + 1):
        if not not_prime[i]:
            pri.append(i)
            phi[i] = i - 1
        for pri_j in pri:
            if i * pri_j > n:
                break
            not_prime[i * pri_j] = True
            if i % pri_j == 0:
                phi[i * pri_j] = phi[i] * pri_j
                break
            phi[i * pri_j] = phi[i] * phi[pri_j]
```

筛法求莫比乌斯函数

定义

根据莫比乌斯函数的定义，设 n 是一个合数，p_1 是 n 的最小质因子，$n'=\frac{n}{p_1}$，有：

$$\mu(n)=\begin{cases}0 & n' \bmod p_1=0\\ \\ -\mu(n') & \text{otherwise}\end{cases}$$

若 n 是质数，有 $\mu(n)=-1$。

实现

```cpp
vector<int> pri;
bool not_prime[N];
int mu[N];

void pre(int n) {
  mu[1] = 1;
  for (int i = 2; i <= n; ++i) {
    if (!not_prime[i]) {
      mu[i] = -1;
      pri.push_back(i);
    }
    for (int pri_j : pri) {
      if (i * pri_j > n) break;
      not_prime[i * pri_j] = true;
      if (i % pri_j == 0) {
        mu[i * pri_j] = 0;
        break;
      }
      mu[i * pri_j] = -mu[i];
    }
  }
```

```
}
```

```
pri = []
not_prime = [False] * N
mu = [0] * N

def pre(n):
    mu[1] = 1
    for i in range(2, n + 1):
        if not not_prime[i]:
            pri.append(i)
            mu[i] = -1
        for pri_j in pri:
            if i * pri_j > n:
                break
            not_prime[i * pri_j] = True
            if i % pri_j == 0:
                mu[i * pri_j] = 0
                break
            mu[i * pri_j] = -mu[i]
```

筛法求约数个数

用 d_i 表示 i 的约数个数，num_i 表示 i 的最小质因子出现次数。

约数个数定理

定理：若 $n=\prod_{i=1}^{m} p_i^{c_i}$ 则 $d_i=\prod_{i=1}^{m}(c_i+1)$。

证明：我们知道 $p_i^{c_i}$ 的约数有 $p_i^0, p_i^1, \dots, p_i^{c_i}$ 共 c_i+1 个，根据乘法原理，n 的约数个数就是 $\prod_{i=1}^{m}(c_i+1)$。

实现

因为 d_i 是积性函数，所以可以使用线性筛。

在这里简单介绍一下线性筛实现原理。

1. 当 i 为质数时，$num_i \leftarrow 1, d_i \leftarrow 2$，同时设 $q=\left\lfloor \dfrac{i}{p} \right\rfloor$，其中 p 为 i 的最小质因子。
2. 当 p 为 q 的质因子时，$num_i \leftarrow num_q+1, d_i \leftarrow \dfrac{d_q}{num_i} \times (num_i+1)$。
3. 当 p, q 互质时，$num_i \leftarrow 1, d_i \leftarrow d_q \times (num_i+1)$。

```
vector<int> pri;
bool not_prime[N];
int d[N], num[N];

void pre(int n) {
  d[1] = 1;
  for (int i = 2; i <= n; ++i) {
    if (!not_prime[i]) {
      pri.push_back(i);
      d[i] = 2;
      num[i] = 1;
    }
    for (int pri_j : pri) {
```

```
      if (i * pri_j > n) break;
      not_prime[i * pri_j] = true;
      if (i % pri_j == 0) {
        num[i * pri_j] = num[i] + 1;
        d[i * pri_j] = d[i] / num[i * pri_j] * (num[i * pri_j] + 1);
        break;
      }
      num[i * pri_j] = 1;
      d[i * pri_j] = d[i] * 2;
    }
  }
}
```

```
pri = []
not_prime = [False] * N
d = [0] * N
num = [0] * N

def pre(n):
    d[1] = 1
    for i in range(2, n + 1):
        if not not_prime[i]:
            pri.append(i)
            d[i] = 2
            num[i] = 1
        for pri_j in pri:
            if i * pri_j > n:
                break
            not_prime[i * pri_j] = True
            if i % pri_j == 0:
                num[i * pri_j] = num[i] + 1
                d[i * pri_j] = d[i] // num[i * pri_j] * (num[i * pri_j] + 1)
                break
            num[i * pri_j] = 1
            d[i * pri_j] = d[i] * 2
```

筛法求约数和

f_i 表示 i 的约数和，g_i 表示 i 的最小质因子的 $p^0 + p^1 + p^2 + \ldots p^k$.

实现

```
vector<int> pri;
bool not_prime[N];
int g[N], f[N];

void pre(int n) {
  g[1] = f[1] = 1;
  for (int i = 2; i <= n; ++i) {
    if (!not_prime[i]) {
      pri.push_back(i);
      g[i] = i + 1;
      f[i] = i + 1;
```

```
    }
    for (int pri_j : pri) {
      if (i * pri_j > n) break;
      not_prime[i * pri_j] = true;
      if (i % pri_j == 0) {
        g[i * pri_j] = g[i] * pri_j + 1;
        f[i * pri_j] = f[i] / g[i] * g[i * pri_j];
        break;
      }
      f[i * pri_j] = f[i] * f[pri_j];
      g[i * pri_j] = 1 + pri_j;
    }
  }
}
```

```
pri = []
not_prime = [False] * N
f = [0] * N
g = [0] * N

def pre(n):
    g[1] = f[1] = 1
    for i in range(2, n + 1):
        if not not_prime[i]:
            pri.append(i)
            g[i] = i + 1
            f[i] = i + 1
        for pri_j in pri:
            if i * pri_j > n:
                break
            not_prime[i * pri_j] = True
            if i % pri_j == 0:
                g[i * pri_j] = g[i] * pri_j + 1
                f[i * pri_j] = f[i] // g[i] * g[i * pri_j]
                break
            f[i * pri_j] = f[i] * f[pri_j]
            g[i * pri_j] = 1 + pri_j
```

一般的积性函数

假如一个 积性函数 f 满足：对于任意质数 p 和正整数 k，可以在 $O(1)$ 时间内计算 $f(p^k)$，那么可以在 $O(n)$ 时间内筛出 $f(1), f(2), \dots, f(n)$ 的值。

设合数 n 的质因子分解是 $\prod_{i=1}^{k} p_i^{\alpha_i}$，其中 $p_1 < p_2 < \dots < p_k$ 为质数，我们在线性筛中记录 $g_n = p_1^{\alpha_1}$，假如 n 被 $x \cdot p$ 筛掉（p 是质数），那么 g 满足如下递推式：

$$g_n = \begin{cases} g_x \cdot p & x \bmod p = 0 \\ \\ p & \text{otherwise} \end{cases}$$

假如 $n = g_n$，说明 n 就是某个质数的次幂，可以 $O(1)$ 计算 $f(n)$；否则，$f(n) = f(\frac{n}{g_n}) \cdot f(g_n)$。

参考资料与注释

[1]

[2] Sieve of Eratosthenes

9.12.7 Meissel–Lehmer 算法

Authors: Peanut-Tang, Early0v0, Vxlimo, GHLinZhengyu, 1196131597

「Meissel–Lehmer 算法」是一种能在亚线性时间复杂度内求出 $1 \sim n$ 内质数个数的一种算法。

记号规定

$[x]$ 表示对 x 下取整得到的结果。

p_k 表示第 k 个质数，$p_1 = 2$。

$\pi(x)$ 表示 $1 \sim x$ 范围内素数的个数。

$\mu(x)$ 表示莫比乌斯函数。

对于集合 S，$\#S$ 表示集合 S 的大小。

$\delta(x)$ 表示 x 最小的质因子。

$P^{+}(n)$ 表示 x 最大的质因子。

Meissel–Lehmer 算法求 (x)

定义 $\phi(x, a)$ 为所有小于 x 的正整数中满足其所有质因子都大于 p_a 的数的个数，即：

$$\phi(x,a) = \#\{n \le x \mid n \bmod p = 0 \implies p > p_a\} \tag{1}$$

再定义 $P_k(x, a)$ 表示为所有小于 x 的正整数中满足可重质因子恰好有 k 个且所有质因子都大于 p_a 的数的个数，即：

$$P_k(x,a) = \#\{n \le x \mid n = q_1 q_2 \cdots q_k \implies \forall i, q_i > p_a\} \tag{2}$$

特殊的，我们定义：$P_0(x, a) = 1$，如此便有：

$$\phi(x,a) = P_0(x,a) + P_1(x,a) + \cdots + P_k(x,a) + \cdots$$

这个无限和式实际上是可以表示为有限和式的，因为在 $p_a^k > x$ 时，有 $P_k(x, a) = 0$。

设 y 为满足 $x^{1/3} \le y \le x^{1/2}$ 的整数，再记 $a = \pi(y)$。

在 $k \ge 3$ 时，有 $P_1(x, a) = \pi(x) - a$ 与 $P_k(x, a) = 0$，由此我们可以推出：

$$\pi(x) = \phi(x,a) + a - 1 - P_2(x,a) \tag{3}$$

这样，计算 $\pi(x)$ 便可以转化为计算 $\phi(x, a)$ 与 $P_2(x, a)$。

计算 P (x,a)

由等式 (2) 我们可以得出 $P_2(x, a)$ 等于满足 $y < p \le q$ 且 $pq \le x$ 的质数对 (p, q) 的个数。

首先我们注意到 $p \in [y + 1, \sqrt{x}]$。此外，对于每个 p，我们都有 $q \in [p, x/p]$。因此：

$$P_2(x,a) = \sum_{y < p \le \sqrt{x}} \left(\pi\left(\frac{x}{p}\right) - \pi(p) + 1\right) \tag{4}$$

当 $p\in[y+1,\sqrt{x}]$ 时，我们有 $\dfrac{x}{p}\in\left[1,\dfrac{x}{y}\right]$。因此，我们可以筛区间 $\left[1,\dfrac{x}{y}\right]$，然后对于所有的的质数 $p\in[y+1,\sqrt{x}]$ 计算 $\pi\left(\dfrac{x}{p}\right)-\pi(p)+1$。为了减少上述算法的空间复杂度，我们可以考虑分块，块长为 L。若块长 $L=y$，则我们可以在 $O\left(\dfrac{x}{y}\log\log x\right)$ 的时间复杂度，$O(y)$ 的空间复杂度内计算 $P_2(x,a)$。

计算 (x,a)

对于 $b\le a$，考虑所有不超过 x 的正整数，满足它的所有质因子都大于 p_{b-1}。这些数可以被分为两类：

1. 可以被 p_b 整除的；
2. 不可以被 p_b 整除的。

属于第 1 类的数有 $\phi\left(\dfrac{x}{p_b},b-1\right)$ 个，属于第二类的数有 $\phi(x,b)$ 个。

因此我们得出结论：

> **定理** 5.1：函数 ϕ 满足下列性质
>
> $$\phi(u,0)=[u] \tag{5}$$
>
> $$\phi(x,b)=\phi(x,b-1)-\phi\left(\frac{x}{p_b},b-1\right) \tag{6}$$

计算 $\phi(x,a)$ 的简单方法可以从这个定理推导出来：我们重复使用等式 (7)，知道最后得到 $\phi(u,0)$。这个过程可以看作从根节点 $\phi(x,a)$ 开始创建有根二叉树，图 1 画出了这一过程。通过这种方法，我们得到如下公式：

$$\phi(x,a)=\sum_{\substack{1\le n\le x\\ P^{+}(n)\le y}}\mu(n)\,[x/n]$$

$$\begin{array}{cccc}
 & \phi(x,a) & & \\
 \swarrow & & \searrow & \\
 \phi(x,a-1) & & -\phi\left(\frac{x}{p_a},a-1\right) & \\
 \swarrow \quad \downarrow & & \downarrow \quad \searrow & \\
 \phi(x,a-2) \quad \phi\left(\frac{x}{p_{a-1}},a-2\right) & & -\phi\left(\frac{x}{p_a},a-2\right) \quad \phi\left(\frac{x}{p_a p_{a-1}},a-2\right) & \\
 & \vdots & &
\end{array}$$

上图表示计算 $\phi(x,a)$ 过程的二叉树：叶子节点权值之和就是 $\phi(x,a)$。

但是，这样需要计算太多东西。因为 $y\ge x^{1/3}$，仅仅计算为 3 个不超过 y 质数的乘积的数，如果按照这个方法计算，会有至少 $\dfrac{x}{\log^3 x}$ 个项，没有办法我们对复杂度的需求。

为了限制这个二叉树的「生长」，我们要改变原来的终止条件。这是原来的终止条件。

> **终止条件** 1：如果 $b=0$，则不要再对节点 $\mu(n)\phi\left(\dfrac{x}{n},b\right)$ 调用等式 (6)。

我们把它改成更强的终止条件：

> **终止条件** 2：如果满足下面 2 个条件中的一个，不要再对节点 $\mu(n)\phi\left(\dfrac{x}{n},b\right)$ 调用等式 (6)：
>
> 1. $b=0$ 且 $n\le y$；
> 2. $n>y$。

我们根据**终止条件** 2 将原二叉树上的叶子分成两种：

1. 如果叶子节点 $\mu(n)\phi\left(\dfrac{x}{n},b\right)$ 满足 $n\le y$，则称这种叶子节点为**普通叶子**；

2. 如果叶子节点 $\mu(n)\phi\left(\frac{x}{n},b\right)$ 满足 $n>y$ 且 $n=mp_b\,(m\le y)$，则称这种节点为**特殊叶子**。

由此我们得出：

定理 5.2**：** 我们有：

$$\phi(x,a)=S_0+S \tag{7}$$

其中 S_0 表示**普通叶子**的贡献：

$$S_0=\sum_{n\le y}\mu(n)\left[\frac{x}{n}\right] \tag{8}$$

S 表示**特殊叶子**的贡献：

$$S=\sum_{n/\delta(n)\le y\le n}\mu(n)\phi\left(\frac{x}{n},\pi(\delta(n))-1\right) \tag{9}$$

计算 S_0 显然是可以在 $O(y\log\log x)$ 的时间复杂度内解决的，现在我们要考虑如何计算 S。

计算 S

我们有：

$$S=-\sum_{p\le y}\sum_{\substack{\delta(m)>p\\ m\le y<mp}}\mu(m)\phi\left(\frac{x}{mp},\pi(p)-1\right) \tag{10}$$

我们将这个等式改写为：

$$S=S_1+S_2+S_3$$

其中：

$$S_1=-\sum_{x^{1/3}<p\le y}\sum_{\substack{\delta(m)>p\\ m\le y<mp}}\mu(m)\phi\left(\frac{x}{mp},\pi(p)-1\right)$$

$$S_2=-\sum_{x^{1/4}<p\le x^{1/3}}\sum_{\substack{\delta(m)>p\\ m\le y<mp}}\mu(m)\phi\left(\frac{x}{mp},\pi(p)-1\right)$$

$$S_3=-\sum_{p\le x^{1/4}}\sum_{\substack{\delta(m)>p\\ m\le y<mp}}\mu(m)\phi\left(\frac{x}{mp},\pi(p)-1\right)$$

注意到计算 S_1,S_2 的和式中涉及到的 m 都是质数，证明如下：

> 如果不是这样，因为有 $\delta(m)>p>x^{1/4}$，所以有 $m>p^2>\sqrt{x}$，这与 $m\le y$ 矛盾，所以原命题成立。

更多的，当 $mp>x^{1/2}\ge y$ 时，有 $y\le mp$。因此我们有：

$$S_1=\sum_{x^{1/3}<p\le y}\sum_{p<q\le y}\phi\left(\frac{x}{pq},\pi(p)-1\right)$$

$$S_2=\sum_{x^{1/4}<p\le x^{1/3}}\sum_{p<q\le y}\phi\left(\frac{x}{pq},\pi(p)-1\right)$$

计算 S

因为：

$$\frac{x}{pq}<x^{1/3}<p$$

所以：

$$\phi\left(\frac{x}{pq},\pi(p)-1\right)=1$$

所以计算 S_1 的和式中的项都是 1。所以我们实际上要计算质数对 (p,q) 的个数，满足：$x^{1/3} < p < q \le y$。
因此：

$$S_1 = \frac{\left(\pi\left(y\right) - \pi\left(x^{1/3}\right)\right)\left(\pi\left(y\right) - \pi\left(x^{1/3}\right) - 1\right)}{2}$$

有了这个等式我们便可以在 $O\left(1\right)$ 的时间内计算 S_1。

计算 S

我们有：

$$S_2 = \sum_{x^{1/4}<p\le x^{1/3}} \sum_{p<q\le y} \phi\left(\frac{x}{pq}, \pi\left(p\right) - 1\right)$$

我们将 S_2 分成 $q > \dfrac{x}{p^2}$ 与 $q \le \dfrac{x}{p^2}$ 两部分：

$$S_2 = U + V$$

其中：

$$U = \sum_{x^{1/4}<p\le x^{1/3}} \sum_{\substack{p<q<y \\ q>x/p^2}} \phi\left(\frac{x}{pq}, \pi\left(p\right) - 1\right)$$

$$V = \sum_{x^{1/4}<p\le x^{1/3}} \sum_{\substack{p<q<y \\ q\le x/p^2}} \phi\left(\frac{x}{pq}, \pi\left(p\right) - 1\right)$$

计算 U

由 $q > \dfrac{x}{p^2}$ 可得 $p^2 > \dfrac{x}{q} \le \dfrac{x}{y}, p > \sqrt{\dfrac{x}{y}}$，因此：

$$U = \sum_{\sqrt{x/y}<p\le x^{1/3}} \sum_{\substack{p<q\le y \\ q>x/p^2}} \phi\left(\frac{x}{pq}, \pi\left(p\right) - 1\right)$$

因此：

$$U = \sum_{\sqrt{x/y}<p\le x^{1/3}} \#\left\{q \mid \frac{x}{p^2} < q \le y\right\}$$

因此：

$$U = \sum_{\sqrt{x/y}<p\le x^{1/3}} \left(\pi\left(y\right) - \pi\left(\frac{x}{p^2}\right)\right)$$

因为有 $\dfrac{x}{p^2} < y$，所以我们可以预处理出所有的 $\pi\left(t\right)\left(t \le y\right)$，这样我们就可以在 $O\left(y\right)$ 的时间复杂度内计算出 U。

计算 V

对于计算 V 的和式中的每一项，我们都有 $p \le \dfrac{x}{pq} < x^{1/2} < p^2$。因此：

$$\phi\left(\frac{x}{pq}, \pi\left(p\right) - 1\right) = 1 + \pi\left(\frac{x}{pq}\right) - \left(\pi\left(p\right) - 1\right) = 2 - \pi\left(p\right) + \pi\left(\frac{x}{pq}\right)$$

所以 V 可以被表示为：

$$V = V_1 + V_2$$

其中：

$$V_1 = \sum_{x^{1/4}<p\le x^{1/3}} \sum_{p<q\le \min(x/p^2, y)} \left(2 - \pi\left(p\right)\right)$$

$$V_2 = \sum_{x^{1/4}<p\le x^{1/3}} \sum_{p<q\le \min(x/p^2, y)} \pi\left(\frac{x}{pq}\right)$$

预处理出 $\pi(t)\,(t\le y)$ 我们就可以在 $O\left(x^{1/3}\right)$ 的时间复杂度内计算出 V_1。

考虑我们如何加速计算 V_2 的过程。我们可以把 q 的贡献拆分成若干个 $\pi\left(\frac{x}{pq}\right)$ 为定值的区间上，这样就只需要计算出每一个区间的长度和从一个区间到下一个区间的 $\pi\left(\frac{x}{pq}\right)$ 的改变量。

更准确的说，我们首先将 V_2 分成两个部分，将 $q\le\min\left(\frac{x}{p^2},y\right)$ 这个复杂的条件简化：

$$V_2=\sum_{x^{1/4}<p\le\sqrt{x/y}}\ \sum_{p<q\le y}\pi\left(\frac{x}{pq}\right)+\sum_{\sqrt{x/y}<p\le x^{1/3}}\ \sum_{p<q\le x/p^2}\pi\left(\frac{x}{pq}\right)$$

接着我们把这个式子改写为：

$$V_2=W_1+W_2+W_3+W_4+W_5$$

其中：

$$W_1=\sum_{x^{1/4}<p\le x/y^2}\ \sum_{p<q\le y}\pi\left(\frac{x}{pq}\right)$$

$$W_2=\sum_{x/y^2<p\le\sqrt{x/y}}\ \sum_{p<q\le\sqrt{x/p}}\pi\left(\frac{x}{pq}\right)$$

$$W_3=\sum_{x/y^2<p\le\sqrt{x/y}}\ \sum_{\sqrt{x/p}<q\le y}\pi\left(\frac{x}{pq}\right)$$

$$W_4=\sum_{\sqrt{x/y}<p\le x^{1/3}}\ \sum_{p<q\le\sqrt{x/p}}\pi\left(\frac{x}{pq}\right)$$

$$W_5=\sum_{\sqrt{x/y}<p\le x^{1/3}}\ \sum_{\sqrt{x/p}<q\le x/p^2}\pi\left(\frac{x}{pq}\right)$$

计算 W 与 W

计算这两个值需要计算满足 $y<\frac{x}{pq}<x^{1/2}$ 的 $\pi\left(\frac{x}{pq}\right)$ 的值。可以在区间 $[1,\sqrt{x}]$ 分块筛出。在每个块中我们对于所有满足条件的 (p,q) 都累加 $\pi\left(\frac{x}{pq}\right)$。

计算 W

对于每个 p，我们把 q 分成若干个区间，每个区间都满足它们的 $\pi\left(\frac{x}{pq}\right)$ 是定值，每个区间我们都可以 $O(1)$ 计算它的贡献。当我们获得一个新的 q 时，我们用 $\pi(t)$（$t\le y$）的值表计算 $\pi\left(\frac{x}{pq}\right)$。$y$ 以内的质数表可以给出使得 $\pi(t)<\pi(t+1)=\pi\left(\frac{x}{pq}\right)$ 成立的 t。以此类推使得 $\pi\left(\frac{x}{pq}\right)$ 变化的下一个 q 的值。

计算 W

相比于 W_3，W_4 中 q 更小，所以 $\pi\left(\frac{x}{pq}\right)$ 改变得更快。这时候再按照计算 W_3 的方法计算 W_4 就显得没有任何优势。于是我们直接暴力枚举数对 (p,q) 来计算 W_4。

计算 W

我们像计算 W_3 那样来计算 W_5。

计算 S

我们使用所有小于 $x^{1/4}$ 的素数一次筛出区间 $\left[1,\frac{x}{y}\right]$。当我们的筛法进行到 p_k 的时候，我们算出了所有 m 满足没有平方因子并且 $\delta(m)>p_k$ 的 $-\mu(m)\phi\left(\frac{x}{mp_k},k-1\right)$ 值。这个筛法是分块进行的，我们在筛选间隔中维护一个二叉树，以实时维护所有素数筛选到给定素数后的中间结果。这样我们就可以只用 $O(\log x)$ 的时间复杂度求出在筛法进行到某一个值的时候没有被筛到的数的数量。

算法的时空复杂度

时空复杂度被如下 3 个过程影响：

1. 计算 $P_2\left(x,a\right)$；
2. 计算 W_1,W_2,W_3,W_4,W_5；
3. 计算 S_3。

计算 P (x,y) 的复杂度

我们已经知道了这个过程的时间复杂度为 $O\left(\frac{x}{y}\log\log x\right)$，空间复杂度为 $O\left(y\right)$。

计算 W ,W ,W ,W ,W 的复杂度

计算 W_1,W_2 所进行的块长度为 y 的筛的时间按复杂度为 $O\left(\sqrt{x}\log\log x\right)$，空间复杂度为 $O\left(y\right)$。

计算 W_1 所需的时间复杂度为：

$$\pi\left(\frac{x}{y^2}\right)\pi\left(y\right)=O\left(\frac{x}{y\log^2 x}\right)$$

计算 W_2 的时间复杂度为：

$$O\left(\sum_{x/y^2<p\le\sqrt{x/y}}\pi\left(\sqrt{\frac{x}{p}}\right)\right)=O\left(\frac{x^{3/4}}{y^{1/4}\log^2 x}\right)$$

因此，计算 W_3 的时间复杂度为：

$$O\left(\sum_{x/y^2<p\le\sqrt{x/y}}\pi\left(\sqrt{\frac{x}{p}}\right)\right)=O\left(\frac{x^{3/4}}{y^{1/4}\log^2 x}\right)$$

计算 W_4 的时间复杂度为：

$$O\left(\sum_{\sqrt{x/y}<p\le x^{1/3}}\pi\left(\sqrt{\frac{x}{p}}\right)\right)=O\left(\frac{x^{2/3}}{\log^2 x}\right)$$

计算 W_5 的时间复杂度为：

$$O\left(\sum_{\sqrt{x/y}<p\le x^{1/3}}\pi\left(\sqrt{\frac{x}{p}}\right)\right)=O\left(\frac{x^{2/3}}{\log^2 x}\right)$$

计算 S 的复杂度

对于预处理：由于要快速查询 $\phi(u,b)$ 的值，我们没办法用普通的筛法 $O(1)$ 求出，而是要维护一个数据结构使得每次查询的时间复杂度是 $O(\log x)$，因此时间复杂度为 $O\left(\frac{x}{y}\log x\log\log x\right)$。

对于求和：对于计算 S_3 和式中的每一项，我们查询上述数据结构，一共 $O\left(\log x\right)$ 次查询。我们还需要计算和式的项数，即二叉树中叶子的个数。所有叶子的形式均为 $\pm\phi\left(\frac{x}{mp_b},b-1\right)$，其中 $m\le y,b<\pi(x^{1/4})$。因此，叶子的数目是 $O\left(y\pi\left(x^{1/4}\right)\right)$ 级别的。所以计算 S_3 的总时间复杂度为：

$$O\left(\frac{x}{y}\log x\log\log x+yx^{1/4}\right)$$

总复杂度

这个算法的空间复杂度为 $O\left(y\right)$，时间复杂度为：

$$O\left(\frac{x}{y}\log\log x+\frac{x}{y}\log x\log\log x+x^{1/4}y+\frac{x^{2/3}}{\log^2 x}\right)$$

我们取 $y=x^{1/3}\log^3 x\log\log x$，就有最优时间复杂度为 $O\left(\frac{x^{2/3}}{\log^2 x}\right)$，空间复杂度为 $O\left(x^{1/3}\log^3 x\log\log x\right)$。

一些改进

我们在这里给出改进方法，以减少算法的常数，提高它的实际效率。

- 在**终止条件** 2 中，我们可以用一个 z 来代替 y，其中 z 满足 $z > y$。我们可以证明这样子计算 S_3 的时间复杂度可以优化到：
$$O\left(\frac{x}{z}\log x\log\log x+\frac{yx^{1/4}}{\log x}+z^{3/2}\right)$$
这也为通过改变 z 的值来检查计算提供了一个很好的方法。
- 为了清楚起见，我们在阐述算法的时候选择在 $x^{1/4}$ 处拆分来计算总和 S，但实际上我们只需要有 $p \le \frac{x}{pq} < p^2$ 就可以计算。我们可以利用这一点，渐近复杂性保持不变。
- 用前几个素数 $2,3,5$ 预处理计算可以节省更多的时间。

参考资料与拓展阅读

本文翻译自：Computing $\pi(x)$: the Meissel, Lehmer, Lagarias, Miller, Odlyzko method[1]

参考资料与注释

[1] Computing $\pi(x)$: the Meissel, Lehmer, Lagarias, Miller, Odlyzko method

9.12.8 分解质因数

引入

给定一个正整数 $N \in \mathbf{N}_+$，试快速找到它的一个 非平凡因数。

考虑朴素算法，因数是成对分布的，N 的所有因数可以被分成两块，即 $[2,\sqrt{N}]$ 和 $[\sqrt{N}+1,N)$。只需要把 $[2,\sqrt{N}]$ 里的数遍历一遍，再根据除法就可以找出至少两个因数了。这个方法的时间复杂度为 $O(\sqrt{N})$。

当 $N \ge 10^{18}$ 时，这个算法的运行时间我们是无法接受的，希望有更优秀的算法。一种想法是通过随机的方法，猜测一个数是不是 N 的因数，如果运气好可以在 $O(1)$ 的时间复杂度下求解答案，但是对于 $N \ge 10^{18}$ 的数据，成功猜测的概率是 $\frac{1}{10^{18}}$，期望猜测的次数是 10^{18}。如果是在 $[2,\sqrt{N}]$ 里进行猜测，成功率会大一些。我们希望有方法来优化猜测。

朴素算法

最简单的算法即为从 $[2,\sqrt{N}]$ 进行遍历。

```
vector<int> breakdown(int N) {
  vector<int> result;
  for (int i = 2; i * i <= N; i++) {
    if (N % i == 0) {  // 如果 i 能够整除 N，说明 i 为 N 的一个质因子。
      while (N % i == 0) N /= i;
      result.push_back(i);
    }
  }
  if (N != 1) {  // 说明再经过操作之后 N 留下了一个素数
    result.push_back(N);
  }
  return result;
}
```

```python
def breakdown(N):
    result = []
    for i in range(2, int(sqrt(N)) + 1):
        if N % i == 0: # 如果 i 能够整除 N，说明 i 为 N 的一个质因子。
            while N % i == 0:
                N //= i
            result.append(i)
    if N != 1: # 说明再经过操作之后 N 留下了一个素数
        result.append(N)
    return result
```

我们能够证明 `result` 中的所有元素均为 `N` 的素因数。

" 证明 `result` 中均为 N 的素因数"

首先证明元素均为 N 的素因数：因为当且仅当 `N % i == 0` 满足时，`result` 发生变化：储存 i，说明此时 i 能整除 $\frac{N}{A}$，说明了存在一个数 p 使得 $pi = \frac{N}{A}$，即 $piA = N$（其中，A 为 N 自身发生变化后遇到 i 时所除的数。我们注意到 `result` 若在 push i 之前就已经有数了，为 $R_1, R_2, \ldots, R_n$，那么有 `N` $= \frac{N}{R_1^{q_1} \cdot R_2^{q_2} \cdot \cdots \cdot R_n^{q_n}}$，被除的乘积即为 A）。所以 i 为 N 的因子。

其次证明 `result` 中均为素数。我们假设存在一个在 `result` 中的合数 K，并根据整数基本定理，分解为一个素数序列 $K = K_1^{e_1} \cdot K_2^{e_2} \cdot \cdots \cdot K_3^{e_3}$，而因为 $K_1 < K$，所以它一定会在 K 之前被遍历到，并令 `while(N % k1 == 0) N /= k1`，即让 `N` 没有了素因子 K_1，故遍历到 K 时，`N` 和 K 已经没有了整除关系了。

值得指出的是，如果开始已经打了一个素数表的话，时间复杂度将从 $O(\sqrt{N})$ 下降到 $O(\sqrt{\frac{N}{\ln N}})$。去 筛法 处查阅更多打表的信息。

例题：CF 1445C[2]

Pollard Rho 算法

引入

而下面复杂度复杂度更低的 Pollard-Rho 算法是一种用于快速分解非平凡因数的算法（**注意**！非平凡因子不是素因子）。而在此之前需要先引入生日悖论。

生日悖论

不考虑出生年份（假设每年都是 365 天），问：一个房间中至少多少人，才能使其中两个人生日相同的概率达到 50%?

解：假设一年有 n 天，房间中有 k 人，用整数 $1, 2, \ldots, k$ 对这些人进行编号。假定每个人的生日均匀分布于 n 天之中，且两个人的生日相互独立。

设 k 个人生日互不相同为事件 A, 则事件 A 的概率为

$$P(A) = \prod_{i=0}^{k-1} \frac{n-i}{n}$$

至少有两个人生日相同的概率为 $P(\overline{A}) = 1 - P(A)$。根据题意可知 $P(\overline{A}) \geq \frac{1}{2}$, 那么就有

$$P(A) = \prod_{i=0}^{k-1} \frac{n-i}{n} \leq \frac{1}{2}$$

由不等式 $1 + x \leq \mathrm{e}^x$ 可得

$$P(A) \leq \prod_{i=1}^{k-1} \exp\left(-\frac{i}{n}\right) = \exp\left(-\frac{k(k-1)}{2n}\right)$$

因此

$$\exp\left(-\frac{k(k-1)}{2n}\right) \leq \frac{1}{2} \implies P(A) \leq \frac{1}{2}$$

将 $n = 365$ 代入，解得 $k \geq 23$。所以一个房间中至少 23 人，使其中两个人生日相同的概率达到 50%，但这个数学事实十分反直觉，故称之为一个悖论。

当 $k > 56$，$n = 365$ 时，出现两个人同一天生日的概率将大于 99%[1]。那么在一年有 n 天的情况下，当房间中有 $\frac{1}{2}(\sqrt{8n\ln 2+1}+1) \approx \sqrt{2n\ln 2}$ 个人时，至少有两个人的生日相同的概率约为 50%。

考虑一个问题，设置一个数据 n，在 $[1,1000]$ 里随机选取 i 个数（$i = 1$ 时就是它自己），使它们之间有两个数的差值为 k。当 $i = 1$ 时成功的概率是 $\frac{1}{1000}$，当 $i = 2$ 时成功的概率是 $\frac{1}{500}$（考虑绝对值，k_2 可以取 $k_1 - k$ 或 $k_1 + k$），随着 i 的增大，这个概率也会增大最后趋向于 1。

利用最大公约数求出一个约数

n 和某个数的最大公约数一定是 n 的约数，即 $\forall k \in \mathbf{N}_+, \gcd(k,n) \mid n$，只要选适当的 k 使得 $1 < \gcd(k,n) < n$，就可以求得 n 的一个约数 $\gcd(k,n)$。满足这样条件的 k 不少，n 有若干个质因子，每个质因子及其大部分倍数都是可行的。

我们通过 $f(x) = (x^2+c) \bmod n$ 来生成一个序列 $\{x_i\}$：随机取一个 x_1，令 $x_2 = f(x_1), x_3 = f(x_2), \ldots, x_i = f(x_{i-1})$。其中 c 是一个随机选取的常数。

举个例子，设 $N = 50, c = 2, x_1 = 1$，$f(x)$ 生成的数据为

$$1, 3, 11, 23, 31, 11, 23, 31, \ldots$$

可以发现数据在 3 以后都在 $11, 23, 31$ 之间循环。如果将这些数如下图一样排列起来，会发现这个图像酷似一个 ρ，算法也因此得名 rho。

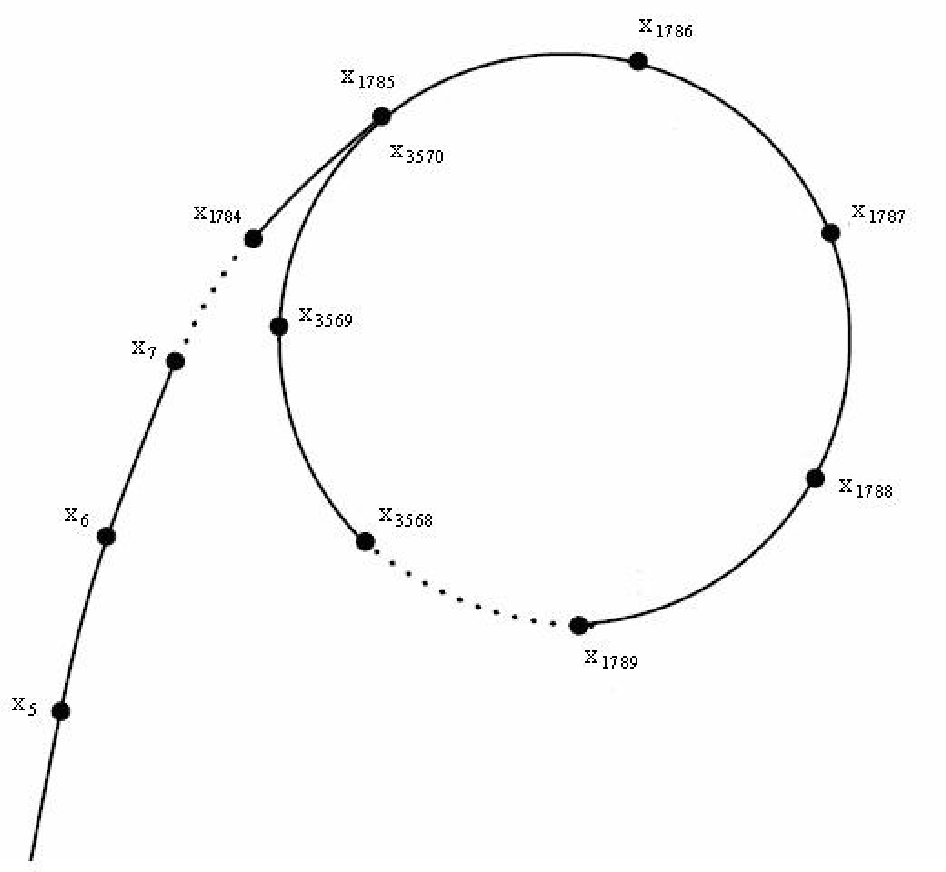


图 9.9 Pollard-rho1

选择 $f(x) = (x^2 + c) \bmod n$ 这个函数生成序列，是因为它有一个性质：$\forall x \equiv y \pmod p, f(x) \equiv f(y) \pmod p$，其中 $p \mid n$。

" 证明"

若 $x \equiv y \pmod p$，则可以将它们表示为 $x = k_1 p + a$，$y = k_2 p + a$，满足 $k_1, k_2, a \in \mathbb{Z}, a \in [0, p)$。

$f(x) = (x^2 + c) \bmod n$，因此 $f(x) = x^2 + c - kn$，其中 $k \in \mathbb{Z}$。

$$\begin{aligned} f(x) &= x^2 + c - kn \\ &= (k_1 p + a)^2 + c - kn \\ &= k_1^2 p^2 + 2k_1 pa + a^2 + c - kn \\ &\equiv a^2 + c \pmod p \end{aligned}$$

同理，$f(y) \equiv a^2 + c \pmod p$，因此 $f(x) \equiv f(y) \pmod p$。

根据生日悖论，生成的序列中不同值的数量约为 $O(\sqrt{n})$ 个。设 m 为 n 的最小非平凡因子，显然有 $m \leq \sqrt{n}$。

将 $\{x_i\}$ 中每一项对 m 取模，我们得到了一个新序列 $\{y_i\}$（当然也可以写作 $\{x_i \bmod m\}$），并且根据生日悖论可以得知新序列中不同值的个数约为 $O(\sqrt{m}) \leq O(n^{\frac{1}{4}})$。

因此，我们可以期望在 $O(n^{\frac{1}{4}})$ 的时间内找到两个位置 i, j，使得 $x_i \neq x_j \wedge y_i = y_j$，这意味着 $n \nmid |x_i - x_j| \wedge m \mid |x_i - x_j|$，我们可以通过 $\gcd(n, |x_i - x_j|)$ 获得 n 的一个非平凡因子。

同理，任何满足 $\forall x \equiv y \pmod p, f(x) \equiv f(y) \pmod p$ 的函数 $f(x)$（例如多项式函数）都可以用在此处，且我们希望它对大部分初始值尽可能快地进入循环、循环周期尽可能短（但对于一个合数与其任一质因子，循环周期长度不能相同，否则 Pollard Rho 算法将无法分离出这个因子）。

一般选择 $x^2 + c$ 生成序列，原因是它满足上述条件，且运行效率较高。同时，每次调用函数都将 c 设置为随机常数可以保证分解失败后重新分解的成功率。

性质

我们期望枚举 $O(\sqrt{m})$ 个 i 来分解出 n 的一个非平凡因子 $\gcd(|x_i - x_j|, n)$，因此。Pollard-rho 算法能够在 $O(\sqrt{m})$ 的期望时间复杂度内分解出 n 的一个非平凡因子，通过上面的分析可知 $O(\sqrt{m}) \leq O(n^{\frac{1}{4}})$，那么 Pollard-rho 算法的总时间复杂度为 $O(n^{\frac{1}{4}})$。

下面介绍两种实现算法，两种算法都可以在 $O(\sqrt{m})$ 的时间复杂度内完成。

实现

Floyd 判环

假设两个人在赛跑，A 的速度快，B 的速度慢，经过一定时间后，A 一定会和 B 相遇，且相遇时 A 跑过的总距离减去 B 跑过的总距离一定是圈长的 n 倍。

设 $a = f(1), b = f(f(1))$，每一次更新 $a = f(a), b = f(f(b))$，只要检查在更新过程中 a、b 是否相等，如果相等了，那么就出现了环。

我们每次令 $d = \gcd(|x_i - x_j|, n)$，判断 d 是否满足 $1 < d < n$，若满足则可直接返回 d。由于 x_i 是一个伪随机数列，必定会形成环，在形成环时就不能再继续操作了，直接返回 n 本身，并且在后续操作里调整随机常数 c，重新分解。

" 基于 Floyd 判环的 Pollard-Rho 算法"

```
ll Pollard_Rho(ll N) {
  ll c = rand() % (N - 1) + 1;
  ll t = f(0, c, N);
  ll r = f(f(0, c, N), c, N);
  while (t != r) {
    ll d = gcd(abs(t - r), N);
    if (d > 1) return d;
    t = f(t, c, N);
```

```
    r = f(f(r, c, N), c, N);
  }
  return N;
}

import random
def Pollard_Rho(N):
    c = random.randint(1, N - 1)
    t = f(0, c, N)
    r = f(f(0, c, N), c, N)
    while t != r:
        d = gcd(abs(t - r), N)
        if d > 1:
            return d
        t = f(t, c, N)
        r = f(f(r, c, N), c, N)
    return N
```

倍增优化

使用 gcd 求解的时间复杂度为 $O(\log N)$，频繁地调用会使算法运行地很慢，可以通过乘法累积来减少求 gcd 的次数。如果 $1 < \gcd(a, b)$，则有 $1 < \gcd(ac, b)$，$c \in \mathbf{N}_+$，并且由 欧几里得算法 有 $1 < \gcd(ac \bmod b, b) = \gcd(ac, b)$。

我们每过一段时间将这些差值进行 gcd 运算，设 $s = \prod |x_0 - x_j| \bmod n$，如果某一时刻得到 $s = 0$ 那么表示分解失败，退出并返回 n 本身。每隔 $2^k - 1$ 个数，计算是否满足 $1 < \gcd(s, n) < n$。此处取 $k = 7$，可以根据实际情况进行调节。

注意到在环上更容易分解出因数，我们可以先迭代一定的次数。

" 实现"

```
ll Pollard_Rho(ll x) {
  ll t = 0;
  ll c = rand() % (x - 1) + 1;
  // 加速算法，这一步可以省略
  for (int i = 1; i < 1145; ++i) t = f(t, c, x);
  ll s = t;
  int step = 0, goal = 1;
  ll val = 1;
  for (goal = 1;; goal <<= 1, s = t, val = 1) {
    for (step = 1; step <= goal; ++step) {
      t = f(t, c, x);
      val = val * abs(t - s) % x;
      // 如果 val 为 0，退出重新分解
      if (!val) return x;
      if (step % 127 == 0) {
        ll d = gcd(val, x);
        if (d > 1) return d;
      }
    }
    ll d = gcd(val, x);
    if (d > 1) return d;
  }
}

from random import randint
```

```python
from math import gcd
def Pollard_Rho(x):
    c = randint(1, x-1)
    s = t = f(0, c, x)
    goal = val = 1
    while True:
        for step in range(1, goal+1):
            t = f(t, c, x)
            val = val * abs(t - s) % x
            if val == 0:
                return x # 如果 val 为 0，退出重新分解
            if step % 127 == 0:
                d = gcd(val, x)
                if d > 1:
                    return d
        d = gcd(val, x)
        if d > 1:
            return d
        s = t
        goal <<= 1
        val = 1
```

例题：P4718【模板】Pollard-Rho 算法[3]

对于一个数 n，用 Miller Rabin 算法 判断是否为素数，如果是就可以直接返回了，否则用 Pollard-Rho 算法找一个因子 p，将 n 除去因子 p。再递归分解 n 和 p，用 Miller Rabin 判断是否出现质因子，并用 max_factor 更新就可以求出最大质因子了。由于这个题目的数据过于庞大，用 Floyd 判环的方法是不够的，这里采用倍增优化的方法。

" 实现"

```cpp
#include <bits/stdc++.h>

using namespace std;

typedef long long ll;

int t;
long long max_factor, n;

long long gcd(long long a, long long b) {
  if (b == 0) return a;
  return gcd(b, a % b);
}

long long quick_pow(long long x, long long p, long long mod) {  // 快速幂
  long long ans = 1;
  while (p) {
    if (p & 1) ans = (__int128)ans * x % mod;
    x = (__int128)x * x % mod;
    p >>= 1;
  }
  return ans;
}

bool Miller_Rabin(long long p) {  // 判断素数
```

```cpp
  if (p < 2) return 0;
  if (p == 2) return 1;
  if (p == 3) return 1;
  long long d = p - 1, r = 0;
  while (!(d & 1)) ++r, d >>= 1;  // 将 d 处理为奇数
  for (long long k = 0; k < 10; ++k) {
    long long a = rand() % (p - 2) + 2;
    long long x = quick_pow(a, d, p);
    if (x == 1 || x == p - 1) continue;
    for (int i = 0; i < r - 1; ++i) {
      x = (__int128)x * x % p;
      if (x == p - 1) break;
    }
    if (x != p - 1) return 0;
  }
  return 1;
}

long long Pollard_Rho(long long x) {
  long long s = 0, t = 0;
  long long c = (long long)rand() % (x - 1) + 1;
  int step = 0, goal = 1;
  long long val = 1;
  for (goal = 1;; goal *= 2, s = t, val = 1) {  // 倍增优化
    for (step = 1; step <= goal; ++step) {
      t = ((__int128)t * t + c) % x;
      val = (__int128)val * abs(t - s) % x;
      if ((step % 127) == 0) {
        long long d = gcd(val, x);
        if (d > 1) return d;
      }
    }
    long long d = gcd(val, x);
    if (d > 1) return d;
  }
}

void fac(long long x) {
  if (x <= max_factor || x < 2) return;
  if (Miller_Rabin(x)) {              // 如果 x 为质数
    max_factor = max(max_factor, x);  // 更新答案
    return;
  }
  long long p = x;
  while (p >= x) p = Pollard_Rho(x);  // 使用该算法
  while ((x % p) == 0) x /= p;
  fac(x), fac(p);  // 继续向下分解 x 和 p
}

int main() {
  scanf("%d", &t);
  while (t--) {
    srand((unsigned)time(NULL));
    max_factor = 0;
```

```
    scanf("%lld", &n);
    fac(n);
    if (max_factor == n)  // 最大的质因数即自己
      printf("Prime\n");
    else
      printf("%lld\n", max_factor);
  }
  return 0;
}
```

参考资料与链接

[1] https://en.wikipedia.org/wiki/Birthday_problem#Reverse_problem

[2] CF 1445C

[3] P4718【模板】Pollard-Rho 算法

9.12.9 裴蜀定理

定义

裴蜀定理，又称贝祖定理（Bézout's lemma）、贝祖等式（Bézout's identity）。是一个关于最大公约数的定理。

其内容是：

设 a, b 是不全为零的整数，对任意整数 x, y，满足 $\gcd(a,b) \mid ax+by$，且存在整数 x, y, 使得 $ax+by=\gcd(a,b)$.

证明

对于第一点

由于 $\gcd(a,b) \mid a, \gcd(a,b) \mid b$

所以 $\gcd(a,b) \mid ax, \gcd(a,b) \mid by$，其中 x, y 均为整数

因此 $\gcd(a,b) \mid ax+by$

对于第二点

1. 若任何一个等于 0, 则 $\gcd(a,b)=a$. 这时定理显然成立。
2. 若 a, b 不等于 0.

   由于 $\gcd(a,b)=\gcd(a,-b)$,

   不妨设 a, b 都大于 0，$a \geq b, \gcd(a,b)=d$.

   对 $ax+by=d$, 两边同时除以 d, 可得 $a_1x+b_1y=1$, 其中 $(a_1,b_1)=1$.

   转证 $a_1x+b_1y=1$.

   我们先回顾一下辗转相除法是怎么做的，由 $\gcd(a,b) \to \gcd(b, a \mod b) \to \ldots$ 我们把模出来的数据叫做 r 于是，有

$$\gcd(a_1,b_1)=\gcd(b_1,r_1)=\gcd(r_1,r_2)=\cdots=(r_{n-1},r_n)=1$$

把辗转相除法中的运算展开，做成带余数的除法，得

$$\begin{aligned}
a_1 &= q_1 b_1 + r_1 && (0 \le r_1 < b_1) \\
b_1 &= q_2 r_1 + r_2 && (0 \le r_2 < r_1) \\
r_1 &= q_3 r_2 + r_3 && (0 \le r_3 < r_2) \\
&\cdots \\
r_{n-3} &= q_{n-1} r_{n-2} + r_{n-1} \\
r_{n-2} &= q_n r_{n-1} + r_n \\
r_{n-1} &= q_{n+1} r_n
\end{aligned}$$

不妨令辗转相除法在除到互质的时候退出则 $r_n = 1$ 所以有（q 被换成了 x，为了符合最终形式）

$$r_{n-2} = x_n r_{n-1} + 1$$

即

$$1 = r_{n-2} - x_n r_{n-1}$$

由倒数第三个式子 $r_{n-1} = r_{n-3} - x_{n-1} r_{n-2}$ 代入上式，得

$$1 = (1 + x_n x_{n-1}) r_{n-2} - x_n r_{n-3}$$

然后用同样的办法用它上面的等式逐个地消去 $r_{n-2}, \cdots, r_1$，
可证得 $1 = a_1 x + b_1 y$. 这样等于是一般式中 $d = 1$ 的情况。

推广

逆定理

设 a, b 是不全为零的整数，若 $d > 0$ 是 a, b 的公因数，且存在整数 x, y, 使得 $ax + by = d$，则 $d = \gcd(a, b)$。

特殊地，设 a, b 是不全为零的整数，若存在整数 x, y, 使得 $ax + by = 1$，则 a, b 互质。

多个整数

裴蜀定理可以推广到 n 个整数的情形：设 $a_1, a_2, \dots, a_n$ 是不全为零的整数，则存在整数 $x_1, x_2, \dots, x_n$，使得 $a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = \gcd(a_1, a_2, \dots, a_n)$。其逆定理也成立：设 $a_1, a_2, \dots, a_n$ 是不全为零的整数，$d > 0$ 是 $a_1, a_2, \dots, a_n$ 的公因数，若存在整数 $x_1, x_2, \dots, x_n$，使得 $a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = d$，则 $d = \gcd(a_1, a_2, \dots, a_n)$。

应用

"Codeforces Round #290 (Div. 2) D. Fox And Jumping"

给出 n 张卡片，分别有 l_i 和 c_i。在一条无限长的纸带上，你可以选择花 c_i 的钱来购买卡片 i，从此以后可以向左或向右跳 l_i 个单位。问你至少花多少元钱才能够跳到纸带上全部位置。若不行，输出 -1。

分析该问题，发现想要跳到每一个格子上，必须使得所选数 $l_{i_1}, \dots, l_{i_k}$ 通过数次相加或相减得出的绝对值为 1，也即存在整数 $x_1, \dots, x_n$ 使得 $l_{i_1} x_1 + \cdots + l_{i_k} x_k = 1$。由多个整数的裴蜀定理逆定理，这相当于从数组 $l_1, \dots, l_n$ 选择若干个数，满足它们的最大公因数为 1，同时要求代价和最小。

解法 1：我们可以转移思想，因为这些数互质，即为 0 号节点开始，每走一步求 gcd（节点号，下一个节点），同时记录代价（求边权），就成为了从 0 通过不断 gcd 最后变为 1 的最小代价。

由于：互质即为最大公因数为 1，$\gcd(0, x) = x$ 这两个定理，可以证明该算法的正确。选择优先队列优化 Dijkstra 求解。

不过还有个问题，即为需要记录是否已经买过一个卡片，开数组标记由于数据范围达到 10^9 会超出内存限制，可以想到使用 `unordered_map`（比普通的 `map` 更快地访问各个元素，迭代效率较低，详见 STL-map）

解法 2：从数组 $l_1, \dots, l_n$ 选择若干个数，满足它们的最大公因数为 1，且代价和最小，由此可以想到 0-1 背包问题。

设 $f_{i,j}$ 表示考虑前 i 个数且最大公因数为 j 的最小代价，则有转移方程：

$$f_{i,j} = \min_{\gcd(k,l_i)=j} f_{i-1,k} + c_i.$$

DP 后最终的总代价即为 $f_{n,1}$。

如同一般的 0-1 背包问题，可以用滚动数组优化，去掉第一维。而这里 300 个数可达的最大公因数 j 是很稀疏的，因此还可以使用 `unordered_map` 代替数组储存下标 j，优化内存并进一步减少枚举量。

实际上，这里解法 1 建出的图便是解法 2 中动态规划的状态转移图，解法 2 相当于用动态规划求有向无环图的最短路，因此解法 1 和解法 2 是等价的。但解法 2 无需储存全图，同时 DP 的时间复杂度为 $O(n+m)$，相比 Dijkstra 算法更低，因此解法 2 在时间和空间上更优。

进一步结论

对自然数 a、b 和整数 n，a 与 b 互素，考察不定方程：

$$ax + by = n$$

其中 x 和 y 为自然数。如果方程有解，称 n 可以被 a、b 表示。

记 $C = ab - a - b$。由 a 与 b 互素，C 必然为奇数。则有结论：

对任意的整数 n，n 与 $C - n$ 中有且仅有一个可以被表示。

即：可表示的数与不可表示的数在区间 $[0, C]$ 对称（关于 C 的一半对称）。0 可被表示，C 不可被表示；负数不可被表示，大于 C 的数可被表示。

证明

由于 a、b 互素，因此原方程有整数解。设解为：

$$\begin{cases} x = x_0 + bt \\ y = y_0 - at \end{cases}$$

其中 t 为整数。取适当的 t，使得 y 位于 0 到 $a-1$ 之间。这只需在 y_0 上加上或减去若干个 a，即可得到这样的 t。

第一步：证明大于 C 的数都可以被表示。当 n 大于 C 时：

$$ax = n - by > ab - a - b - by \geqslant ab - a - b - b(a-1) = -a$$

于是 x 也是非负整数。

第二步：证明 C 不可被表示，进而 n 与 $C - n$ 不可能都被表示。

反证法。若 $ax + by = ab - a - b$ 有非负整数解 x、y，则：

$$ab = a(x+1) + b(y+1)$$

由于 a 与 b 互素，所以 a 整除 $y+1$，b 整除 $x+1$，a 不超过 $y+1$，b 不超过 $x+1$。于是有：

$$ab = a(x+1) + b(y+1) \geqslant ab + ab = 2ab$$

矛盾！第二步证完。

第三步：证明如果 n 不可被表示，则 $C - n$ 可被表示。

由上可知，若 n 不可被表示，由于上述方程中已规定 y 在 0 到 $a-1$ 之间，则 x 为负。所以：

$$ab - a - b - ax - by = a(-x-1) + b(a-1-y)$$

显然 $-x-1$ 和 $a-1-y$ 均非负，于是 $C - n$ 可被表示。

几何意义

重新观察方程 $ax+by=n$，将它看成一条直线。直线与两坐标轴在第一象限围成三角形。

当 $n<ab$ 的时候，这个直线在第一象限，至多只能通过一个整点。

根据上述讨论：当 n 可以被表示的时候，直线恰好经过一个整点；当 n 不可以被表示的时候，直线不经过整点（在第一象限）。

这结论也可以理解为：作三角形 $(0,0)(b,0)(0,a)$。随着 n 从 0 不断增加，直线向右上方平移，整点会一个一个地通过直线，直到最后才撞上两个整点。

因此，小于等于 n 的能被表示的非负整数的数量，恰好就是直线 $ax+by=n$（含）与两坐标轴（含）在第一象限围成三角形覆盖的整点个数。

另一种解释

考虑模 b 意义下每个剩余系中最小能被表示的值是多少——大于他们的可以通过增加若干个 b 得到。

观察原方程，a 的若干倍数 $0,a,\cdots,(b-1)a$ 在 $\pmod b$ 意义下互不相同。这些数恰好是这些最小值。那么当 $n<ab$ 时，小于等于 n 的能被表示的非负整数的数量是：

$$\sum_{i=0}^{[\frac{n}{a}]}\left[\frac{n-ia}{b}\right]$$

这是一个非常经典的直线下整点问题，恰好是这条直线：

$$y=-\frac{a}{b}x+\frac{n}{b}$$

即 $ax+by=n$。

使用类欧几里得算法可以在 $O(\log\max(a,b))$ 的时间内求解。因此我们得到了计算小于等于 n 的能被表示的非负整数的数量的工具。

题目

P3951 NOIP2017 提高组 小凯的疑惑 / 蓝桥杯 2013 省 买不到的数目[1]

参考资料与注释

[1] P3951 NOIP2017 提高组小凯的疑惑/蓝桥杯 2013 省买不到的数目

9.12.10 类欧几里德算法

Authors: sshwy, FFjet

定义

类欧几里德算法是洪华敦在 2016 年冬令营营员交流中提出的内容。

其本质可以理解为，使用一个类似辗转相除法的方法来进行函数求和。

引入

设

$$f(a,b,c,n)=\sum_{i=0}^{n}\left\lfloor\frac{ai+b}{c}\right\rfloor$$

其中 a,b,c,n 是常数。需要一个 $O(\log n)$ 的算法。

这个式子和我们以前见过的式子都长得不太一样。带向下取整的式子容易让人想到数论分块，然而数论分块似乎不适用于这个求和。但是我们是可以做一些预处理的。

如果说 $a \geq c$ 或者 $b \geq c$，意味着可以将 a,b 对 c 取模以简化问题：

$$\begin{aligned}
f(a,b,c,n) &= \sum_{i=0}^{n}\left\lfloor\frac{ai+b}{c}\right\rfloor \\
&= \sum_{i=0}^{n}\left\lfloor\frac{\left(\left\lfloor\frac{a}{c}\right\rfloor c+a \bmod c\right)i+\left(\left\lfloor\frac{b}{c}\right\rfloor c+b \bmod c\right)}{c}\right\rfloor \\
&= \frac{n(n+1)}{2}\left\lfloor\frac{a}{c}\right\rfloor+(n+1)\left\lfloor\frac{b}{c}\right\rfloor+\sum_{i=0}^{n}\left\lfloor\frac{(a \bmod c)i+(b \bmod c)}{c}\right\rfloor \\
&= \frac{n(n+1)}{2}\left\lfloor\frac{a}{c}\right\rfloor+(n+1)\left\lfloor\frac{b}{c}\right\rfloor+f(a \bmod c, b \bmod c, c, n)
\end{aligned}$$

那么问题转化为了 $a<c,b<c$ 的情况。观察式子，你发现只有 i 这一个变量。因此要推就只能从 i 下手。在推求和式子中有一个常见的技巧，就是条件与贡献的放缩与转化。具体地说，在原式 $f(a,b,c,n)=\sum_{i=0}^{n}\left\lfloor\frac{ai+b}{c}\right\rfloor$ 中，$0 \leq i \leq n$ 是条件，而 $\left\lfloor\frac{ai+b}{c}\right\rfloor$ 是对总和的贡献。

要加快一个和式的计算过程，所有的方法都可以归约为**贡献合并计算**。但你发现这个式子的贡献难以合并，怎么办？**将贡献与条件做转化**得到另一个形式的和式。具体地，我们直接把原式的贡献变成条件：

$$\sum_{i=0}^{n}\left\lfloor\frac{ai+b}{c}\right\rfloor=\sum_{i=0}^{n}\sum_{j=0}^{\left\lfloor\frac{ai+b}{c}\right\rfloor-1}1$$

现在多了一个变量 j，既然算 i 的贡献不方便，我们就想办法算 j 的贡献。因此想办法搞一个和 j 有关的贡献式。这里有另一个家喻户晓的变换方法，笔者概括为限制转移。具体来说，在上面的和式中 n 限制 i 的上界，而 i 限制 j 的上界。为了搞 j，就先把 j 放到贡献的式子里，于是我们交换一下 i,j 的求和算子，强制用 n 限制 j 的上界。

$$=\sum_{j=0}^{\left\lfloor\frac{an+b}{c}\right\rfloor-1}\sum_{i=0}^{n}\left[j<\left\lfloor\frac{ai+b}{c}\right\rfloor\right]$$

这样做的目的是让 j 摆脱 i 的限制，现在 i,j 都被 n 限制，而贡献式看上去是一个条件，但是我们仍把它叫作贡献式，再对贡献式做变换后就可以改变 i,j 的限制关系。于是我们做一些放缩的处理。首先把向下取整的符号拿掉

$$j<\left\lfloor\frac{ai+b}{c}\right\rfloor \iff j+1 \leq \left\lfloor\frac{ai+b}{c}\right\rfloor \iff j+1 \leq \frac{ai+b}{c}$$

然后可以做一些变换

$$j+1 \leq \frac{ai+b}{c} \iff jc+c \leq ai+b \iff jc+c-b-1<ai$$

最后一步，向下取整得到：

$$jc+c-b-1<ai \iff \left\lfloor\frac{jc+c-b-1}{a}\right\rfloor<i$$

这一步的重要意义在于，我们可以把变量 i 消掉了！具体地，令 $m=\left\lfloor\frac{an+b}{c}\right\rfloor$，那么原式化为

$$\begin{aligned}
f(a,b,c,n) &= \sum_{j=0}^{m-1}\sum_{i=0}^{n}\left[i>\left\lfloor\frac{jc+c-b-1}{a}\right\rfloor\right] \\
&= \sum_{j=0}^{m-1}\left(n-\left\lfloor\frac{jc+c-b-1}{a}\right\rfloor\right) \\
&= nm-f\left(c,c-b-1,a,m-1\right)
\end{aligned}$$

这是一个递归的式子。并且你发现 a,c 分子分母换了位置，又可以重复上述过程。先取模，再递归。这就是一个辗转相除的过程，这也是类欧几里德算法的得名。

容易发现时间复杂度为 $O(\log n)$。

扩展

理解了最基础的类欧几里德算法，我们再来思考以下两个变种求和式：

$$g(a,b,c,n)=\sum_{i=0}^{n} i\left\lfloor\frac{ai+b}{c}\right\rfloor$$

$$h(a,b,c,n)=\sum_{i=0}^{n}\left\lfloor\frac{ai+b}{c}\right\rfloor^2$$

推导 g

我们先考虑 g，类似地，首先取模：

$$g(a,b,c,n)=g(a \bmod c,b \bmod c,c,n)+\left\lfloor\frac{a}{c}\right\rfloor\frac{n(n+1)(2n+1)}{6}+\left\lfloor\frac{b}{c}\right\rfloor\frac{n(n+1)}{2}$$

接下来考虑 $a<c,b<c$ 的情况，令 $m=\left\lfloor\frac{an+b}{c}\right\rfloor$。之后的过程比较简略，因为方法和上文略同：

$$\begin{aligned}g(a,b,c,n)&=\sum_{i=0}^{n} i\left\lfloor\frac{ai+b}{c}\right\rfloor\\&=\sum_{j=0}^{m-1}\sum_{i=0}^{n}\left[j<\left\lfloor\frac{ai+b}{c}\right\rfloor\right]\cdot i\end{aligned}$$

这时我们设 $t=\left\lfloor\frac{jc+c-b-1}{a}\right\rfloor$，可以得到

$$\begin{aligned}g(a,b,c,n)&=\sum_{j=0}^{m-1}\sum_{i=0}^{n}[i>t]\cdot i\\&=\sum_{j=0}^{m-1}\frac{1}{2}(t+n+1)(n-t)\\&=\frac{1}{2}\left[mn(n+1)-\sum_{j=0}^{m-1}t^2-\sum_{j=0}^{m-1}t\right]\\&=\frac{1}{2}[mn(n+1)-h(c,c-b-1,a,m-1)-f(c,c-b-1,a,m-1)]\end{aligned}$$

推导 h

同样的，首先取模：

$$\begin{aligned}h(a,b,c,n)=&\,h(a \bmod c,b \bmod c,c,n)\\&+2\left\lfloor\frac{b}{c}\right\rfloor f(a \bmod c,b \bmod c,c,n)+2\left\lfloor\frac{a}{c}\right\rfloor g(a \bmod c,b \bmod c,c,n)\\&+\left\lfloor\frac{a}{c}\right\rfloor^2\frac{n(n+1)(2n+1)}{6}+\left\lfloor\frac{b}{c}\right\rfloor^2(n+1)+\left\lfloor\frac{a}{c}\right\rfloor\left\lfloor\frac{b}{c}\right\rfloor n(n+1)\end{aligned}$$

考虑 $a<c,b<c$ 的情况，$m=\left\lfloor\dfrac{an+b}{c}\right\rfloor,t=\left\lfloor\dfrac{jc+c-b-1}{a}\right\rfloor$.

我们发现这个平方不太好处理，于是可以这样把它拆成两部分：

$$n^2=2\frac{n(n+1)}{2}-n=\left(2\sum_{i=0}^{n} i\right)-n$$

这样做的意义在于，添加变量 j 的时侯就只会变成一个求和算子，不会出现 $\sum\times\sum$ 的形式：

$$\begin{aligned}h(a,b,c,n)&=\sum_{i=0}^{n}\left\lfloor\frac{ai+b}{c}\right\rfloor^2\\&=\sum_{i=0}^{n}\left[\left(2\sum_{j=1}^{\left\lfloor\frac{ai+b}{c}\right\rfloor}j\right)-\left\lfloor\frac{ai+b}{c}\right\rfloor\right]\\&=\left(2\sum_{i=0}^{n}\sum_{j=1}^{\left\lfloor\frac{ai+b}{c}\right\rfloor}j\right)-f(a,b,c,n)\end{aligned}$$

接下来考虑化简前一部分：

$$\begin{aligned}\sum_{i=0}^{n}\sum_{j=1}^{\left\lfloor\frac{ai+b}{c}\right\rfloor}j&=\sum_{i=0}^{n}\sum_{j=0}^{\left\lfloor\frac{ai+b}{c}\right\rfloor-1}(j+1)\\&=\sum_{j=0}^{m-1}(j+1)\sum_{i=0}^{n}\left[j<\left\lfloor\frac{ai+b}{c}\right\rfloor\right]\\&=\sum_{j=0}^{m-1}(j+1)\sum_{i=0}^{n}[i>t]\\&=\sum_{j=0}^{m-1}(j+1)(n-t)\\&=\frac{1}{2}nm(m+1)-\sum_{j=0}^{m-1}(j+1)\left\lfloor\frac{jc+c-b-1}{a}\right\rfloor\\&=\frac{1}{2}nm(m+1)-g(c,c-b-1,a,m-1)-f(c,c-b-1,a,m-1)\end{aligned}$$

因此

$$h(a,b,c,n)=nm(m+1)-2g(c,c-b-1,a,m-1)-2f(c,c-b-1,a,m-1)-f(a,b,c,n)$$

在代码实现的时侯，因为 3 个函数各有交错递归，因此可以考虑三个一起整体递归，同步计算，否则有很多项会被多次计算。这样实现的复杂度是 $O(\log n)$ 的。

实现

"模板题代码实现[1]"

```
#include <bits/stdc++.h>
#define int long long
using namespace std;
const int P = 998244353;
int i2 = 499122177, i6 = 166374059;

struct data {
  data() { f = g = h = 0; }

  int f, g, h;
};  // 三个函数打包

data calc(int n, int a, int b, int c) {
  int ac = a / c, bc = b / c, m = (a * n + b) / c, n1 = n + 1, n21 = n * 2 + 1;
  data d;
```

```cpp
  if (a == 0) {  // 迭代到最底层
    d.f = bc * n1 % P;
    d.g = bc * n % P * n1 % P * i2 % P;
    d.h = bc * bc % P * n1 % P;
    return d;
  }
  if (a >= c || b >= c) {  // 取模
    d.f = n * n1 % P * i2 % P * ac % P + bc * n1 % P;
    d.g = ac * n % P * n1 % P * n21 % P * i6 % P + bc * n % P * n1 % P * i2 % P;
    d.h = ac * ac % P * n % P * n1 % P * n21 % P * i6 % P +
          bc * bc % P * n1 % P + ac * bc % P * n % P * n1 % P;
    d.f %= P, d.g %= P, d.h %= P;

    data e = calc(n, a % c, b % c, c);  // 迭代

    d.h += e.h + 2 * bc % P * e.f % P + 2 * ac % P * e.g % P;
    d.g += e.g, d.f += e.f;
    d.f %= P, d.g %= P, d.h %= P;
    return d;
  }
  data e = calc(m - 1, c, c - b - 1, a);
  d.f = n * m % P - e.f, d.f = (d.f % P + P) % P;
  d.g = m * n % P * n1 % P - e.h - e.f, d.g = (d.g * i2 % P + P) % P;
  d.h = n * m % P * (m + 1) % P - 2 * e.g - 2 * e.f - d.f;
  d.h = (d.h % P + P) % P;
  return d;
}

int T, n, a, b, c;

signed main() {
  scanf("%lld", &T);
  while (T--) {
    scanf("%lld%lld%lld%lld", &n, &a, &b, &c);
    data ans = calc(n, a, b, c);
    printf("%lld %lld %lld\n", ans.f, ans.h, ans.g);
  }
  return 0;
}
```

参考资料与注释

[1] 模板题代码实现

9.12.11 欧拉定理 & 费马小定理

Authors: PeterlitsZo, Tiphereth-A

费马小定理

定义

若 p 为素数，$\gcd(a,p)=1$，则 $a^{p-1}\equiv 1 \pmod p$。

另一个形式：对于任意整数 a，有 $a^p \equiv a \pmod p$。

证明

设一个质数为 p，我们取一个不为 p 倍数的数 a。

构造一个序列：$A = \{1, 2, 3 \ldots, p-1\}$，这个序列有着这样一个性质：

$$\prod_{i=1}^{p-1} A_i \equiv \prod_{i=1}^{p-1}(A_i \times a) \pmod p$$

证明：

$$\because (A_i, p) = 1, (A_i \times a, p) = 1$$

又因为每一个 $A_i \times a \pmod p$ 都是独一无二的，且 $A_i \times a \pmod p < p$

得证（每一个 $A_i \times a$ 都对应了一个 A_i）

设 $f = (p-1)!$, 则 $f \equiv a \times A_1 \times a \times A_2 \times a \times A_3 \cdots \times A_{p-1} \pmod p$

$$\begin{aligned} a^{p-1} \times f &\equiv f \pmod p \\ a^{p-1} &\equiv 1 \pmod p \end{aligned}$$

证毕。

也可用归纳法证明：

显然 $1^p \equiv 1 \pmod p$，假设 $a^p \equiv a \pmod p$ 成立，那么通过二项式定理有

$$(a+1)^p = a^p + \binom{p}{1} a^{p-1} + \binom{p}{2} a^{p-2} + \cdots + \binom{p}{p-1} a + 1$$

因为 $\binom{p}{k} = \frac{p(p-1)\cdots(p-k+1)}{k!}$ 对于 $1 \le k \le p-1$ 成立，在模 p 意义下 $\binom{p}{1} \equiv \binom{p}{2} \equiv \cdots \equiv \binom{p}{p-1} \equiv 0 \pmod p$，那么 $(a+1)^p \equiv a^p + 1 \pmod p$，将 $a^p \equiv a \pmod p$ 带入得 $(a+1)^p \equiv a+1 \pmod p$ 得证。

欧拉定理

在了解欧拉定理（Euler's theorem）之前，请先了解 欧拉函数。定理内容如下：

定义

若 $\gcd(a, m) = 1$，则 $a^{\varphi(m)} \equiv 1 \pmod m$。

证明

实际上这个证明过程跟上文费马小定理的证明过程是非常相似的：**构造一个与 m 互质的数列**，再进行操作。

设 $r_1, r_2, \cdots, r_{\varphi(m)}$ 为模 m 意义下的一个简化剩余系，则 $ar_1, ar_2, \cdots, ar_{\varphi(m)}$ 也为模 m 意义下的一个简化剩余系。所以 $r_1 r_2 \cdots r_{\varphi(m)} \equiv ar_1 \cdot ar_2 \cdots ar_{\varphi(m)} \equiv a^{\varphi(m)} r_1 r_2 \cdots r_{\varphi(m)} \pmod m$，可约去 $r_1 r_2 \cdots r_{\varphi(m)}$，即得 $a^{\varphi(m)} \equiv 1 \pmod m$。

当 m 为素数时，由于 $\varphi(m) = m-1$，代入欧拉定理可立即得到费马小定理。

扩展欧拉定理

定义

$$a^b \equiv \begin{cases} a^{b \bmod \varphi(m)}, & \gcd(a,m) = 1, \\ a^b, & \gcd(a,m) \ne 1, b < \varphi(m), \\ a^{(b \bmod \varphi(m)) + \varphi(m)}, & \gcd(a,m) \ne 1, b \ge \varphi(m). \end{cases} \pmod m$$

解释

读者可能对第二行产生疑问，这一行表达的意思是：如果 $b < \varphi(m)$ 的话，就不能降幂了。

主要是因为题目中 m 不会太大，而如果 $b < \varphi(m)$，自然复杂度是可以接受的。而如果 $b \geq \varphi(m)$ 的话，复杂度可能就超出预期了，这个时候我们才需要降幂来降低复杂度。

证明

直观理解

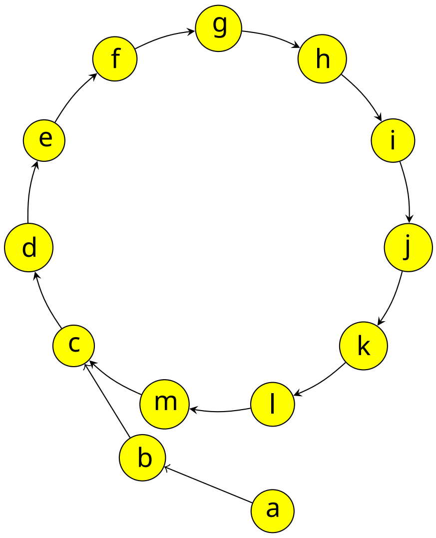


图 9.10 fermat1

需要知道的是，在 $\pmod m$ 的条件下，$a^b \bmod m$ 的取值范围一定在 $[0, m)$，而 $a^i \bmod m = (a^{i-1} \bmod m) \times a \bmod m$，那么对于任意一个数 a，那么很容易就能知道它的**后继**，在有限的空间内这一定会形成一个循环。

在扩展欧拉定理中，循环分为纯循环和混循环。其中纯循环中不存在节点有两个前驱，而混循环则反之。而 $a^i \bmod n$ 形成的序列可以是一个混循环，那么只需要知道循环节的长度，和前面那一小段未进入循环节的长度，就可以根据这个性质来进行降幂了。

值得注意的是，无论是费马小定理，还是（扩展）欧拉定理，一个很重要的应用就是降幂，从而将不可能的表达式化为可能。

形式证明

证明转载自 synapse7[1]，并进行了一些整理。

1. **命题**：a 的从 0 次，1 次到 b 次幂模 m 构成的序列中，存在 r 和 s，使得前 r 个数（即从 $a^0 \bmod m$ 到 $a^{r-1} \bmod m$）互不相同，从第 r 个数开始，每 s 个数就循环一次。

   证明：

   - 由鸽巢定理易证。

     我们把 r 称为 a 幂次模 m 的循环起始点，s 称为循环长度。（注意：r 可以为 0）

     用公式表述为：$\forall i \geq r, a^i \equiv a^{i+s} \pmod m$

2. **命题**：a 为素数的情况，该式成立。

   证明：

   - **若模 m 不能被 a 整除**，而因为 a 是一个素数，那么 $\gcd(a, m) = 1$ 成立，根据欧拉定理，容易证明该式成立。
   - **若模 m 能被 a 整除**，那么存在 r 和 m' 使得 $m = a^r m'$，且 $\gcd(a, m') = 1$ 成立。所以根据欧拉定理有 $a^{\varphi(m')} \equiv 1 \pmod{m'}$。

又由于 $\gcd(a^r, m') = 1$，所以根据欧拉函数的求值规则，容易得到：$\varphi(m) = \varphi(m') \times (a-1)a^{r-1}$，即我们有：$\varphi(m') \mid \varphi(m)$。

所以 $a^{\varphi(m')} \equiv 1 \pmod{m'}, \varphi(m') \mid \varphi(m) \implies a^{\varphi(m)} \equiv 1 \pmod{m'}$，即 $a^{\varphi(m)} = km' + 1$，两边同时乘以 a^r，得 $a^{r+\varphi(m)} = km + a^r$（因为 $m = a^r m'$）

所以对于 m 中素因子 a 的次数 r 满足：$a^r \equiv a^{r+\varphi(m)} \pmod{m}$。我们可以简单变换形式，得到**推论**：

$$b > r \implies a^b \equiv a^{r+((b-r) \bmod \varphi(m))} \pmod{m}$$

又由于 $m = a^r m'$，所以 $\varphi(m) = \varphi(a^r)\varphi(m') \geq \varphi(a^r) = a^{r-1}(a-1) \geq r$（tips：$a$ 是素数，最小是 2，而 $r \geq 1$）。

所以因为 $\varphi(m) \geq r$，故有：

$$a^r \equiv a^{r+\varphi(m)} \equiv a^{r \bmod \varphi(m) + \varphi(m)} \pmod{m}$$

所以

$$\begin{aligned} a^b &\equiv a^{r+(b-r) \bmod \varphi(m)} \\ &\equiv a^{r \bmod \varphi(m) + \varphi(m) + (b-r) \bmod \varphi(m)} \pmod{m} \\ &\equiv a^{\varphi(m) + b \bmod \varphi(m)} \end{aligned}$$

即 $a^b \equiv a^{b \bmod \varphi(m) + \varphi(m)} \pmod{m}$。

3. **命题**：a 为素数的幂的情况，该式成立。

   证明：

   - 不妨令 $a = p^k$，是否依然有 $\forall r, a^r \equiv a^{r+\varphi(m)} \pmod{m}$?

     答案是肯定的，由命题 1 可知存在 s 使得 $a^s \equiv 1 \pmod{m}$，所以 $p^{\operatorname{lcm}(s,k)} \equiv 1 \pmod{m}$，所以令 $s' = \frac{s}{\gcd(s,k)}$ 时，我们能有 $p^{s'k} \equiv 1 \pmod{m}$。

     此时有关系：$s' \mid s$ 且 $s \mid \varphi(m)$，且 $r' = \lceil \frac{r}{k} \rceil \leq r \leq \varphi(m)$，由 r', s' 与 $\varphi(m)$ 的关系，依然可以得到 $a^b \equiv a^{b \bmod \varphi(m) + \varphi(m)} \pmod{m}$。

4. **命题**：a 为合数的情况，该式成立。

   证明：

   - 只证 a 拆成两个素数的幂的情况，大于两个的用数学归纳法可证。

     设 $a = a_1 a_2$，其中 $a_i = p_i^{k_i}$，而 a_i 的循环长度为 s_i；

     则 $s \mid \operatorname{lcm}(s_1, s_2)$，由于 $s_1 \mid \varphi(m), s_2 \mid \varphi(m)$，那么 $\operatorname{lcm}(s_1, s_2) \mid \varphi(m)$，所以 $s \mid \varphi(m)$，$r = \max(\lceil \frac{r_i}{k_i} \rceil) \leq \max(r_i) \leq \varphi(m)$；

     由 r, s 与 $\varphi(m)$ 的关系，依然可以得到 $a^b \equiv a^{b \bmod \varphi(m) + \varphi(m)} \pmod{m}$。

     证毕。

习题

1. SPOJ #4141 "Euler Totient Function"[Difficulty: CakeWalk][2]
2. UVa #10179 "Irreducible Basic Fractions"[Difficulty: Easy][3]
3. UVa #10299 "Relatives"[Difficulty: Easy][4]
4. UVa #11327 "Enumerating Rational Numbers"[Difficulty: Medium][5]
5. TIMUS #1673 "Admission to Exam"[Difficulty: High][6]

参考资料与注释

[1] synapse7

[2] SPOJ #4141 ”Euler Totient Function”[Difficulty: CakeWalk]

[3] UVa #10179 ”Irreducible Basic Fractions”[Difficulty: Easy]

[4] UVa #10299 ”Relatives”[Difficulty: Easy]

[5] UVa #11327 ”Enumerating Rational Numbers”[Difficulty: Medium]

[6] TIMUS #1673 ”Admission to Exam”[Difficulty: High]

9.12.12 乘法逆元

本文介绍模意义下乘法运算的逆元（Modular Multiplicative Inverse），并介绍如何使用扩展欧几里德算法（Extended Euclidean algorithm）求解乘法逆元。

定义

如果一个线性同余方程 $ax \equiv 1 \pmod b$，则 x 称为 $a \bmod b$ 的逆元，记作 a^{-1}。

如何求逆元

扩展欧几里得法

" 实现"

```cpp
void exgcd(int a, int b, int& x, int& y) {
  if (b == 0) {
    x = 1, y = 0;
    return;
  }
  exgcd(b, a % b, y, x);
  y -= a / b * x;
}
```

```python
def exgcd(a, b):
    if b == 0:
        x = 1
        y = 0
        return x, y
    x1, y1 = exgcd(b, a % b)
    x = y1
    y = x1 - (a // b) * y1
    return x, y
```

扩展欧几里得法和求解 线性同余方程 是一个原理，在这里不展开解释。

快速幂法

证明

因为 $ax \equiv 1 \pmod b$；

所以 $ax \equiv a^{b-1} \pmod b$（根据 费马小定理）；

所以 $x \equiv a^{b-2} \pmod b$。

然后我们就可以用快速幂来求了。

" 实现"

```
int qpow(long long a, int b) {
  int ans = 1;
  a = (a % p + p) % p;
  for (; b; b >>= 1) {
    if (b & 1) ans = (a * ans) % p;
    a = (a * a) % p;
  }
  return ans;
}
```

```
def qpow(a, b):
  ans = 1
  a = (a % p + p) % p
  while b:
      if b & 1:
          ans = (a * ans) % p
      a = (a * a) % p
      b >>= 1
  return ans
```

注意：快速幂法使用了 费马小定理，要求 b 是一个素数；而扩展欧几里得法只要求 $\gcd(a,b)=1$。

线性求逆元

求出 $1,2,\dots,n$ 中每个数关于 p 的逆元。

如果对于每个数进行单次求解，以上两种方法就显得慢了，很有可能超时，所以下面来讲一下如何线性（$O(n)$）求逆元。

首先，很显然的 $1^{-1} \equiv 1 \pmod p$；

" 证明"

对于 $\forall p \in \mathbf{Z}$，有 $1\times 1 \equiv 1 \pmod p$ 恒成立，故在 p 下 1 的逆元是 1，而这是推算出其他情况的基础。

其次对于递归情况 i^{-1}，我们令 $k=\lfloor \frac{p}{i} \rfloor$，$j = p \bmod i$，有 $p = ki+j$。再放到 $\mod p$ 意义下就会得到：$ki+j \equiv 0 \pmod p$；

两边同时乘 $i^{-1}\times j^{-1}$：

$kj^{-1}+i^{-1} \equiv 0 \pmod p$

$i^{-1} \equiv -kj^{-1} \pmod p$

再带入 $j = p \bmod i$，有 $p = ki+j$，有：

$i^{-1} \equiv -\lfloor \frac{p}{i} \rfloor (p \bmod i)^{-1} \pmod p$

我们注意到 $p \bmod i < i$，而在迭代中我们完全可以假设我们已经知道了所有的模 p 下的逆元 $j^{-1}, j<i$。

故我们就可以推出逆元，利用递归的形式，而使用迭代实现：

$$i^{-1} \equiv \begin{cases} 1, & \text{if } i=1, \\ -\lfloor \frac{p}{i} \rfloor (p \bmod i)^{-1}, & \text{otherwise.} \end{cases} \pmod p$$

" 实现"

```cpp
inv[1] = 1;
for (int i = 2; i <= n; ++i) {
  inv[i] = (long long)(p - p / i) * inv[p % i] % p;
}
```

```python
inv[1] = 1
for i in range(2, n + 1):
    inv[i] = (p - p // i) * inv[p % i] % p
```

使用 $p-\lfloor\frac{p}{i}\rfloor$ 来防止出现负数。

另外我们注意到我们没有对 `inv[0]` 进行定义却可能会使用它：当 $i|p$ 成立时，我们在代码中会访问 `inv[p % i]`，也就是 `inv[0]`，这是因为当 $i|p$ 时不存在 i 的逆元 i^{-1}。线性同余方程 中指出，如果 i 与 p 不互素时不存在相应的逆元（当一般而言我们会使用一个大素数，比如 10^9+7 来确保它有着有效的逆元）。因此需要指出的是：如果没有相应的逆元的时候，`inv[i]` 的值是未定义的。

另外，根据线性求逆元方法的式子：$i^{-1}\equiv -kj^{-1} \pmod p$

递归求解 j^{-1}，直到 $j=1$ 返回 1。

中间优化可以加入一个记忆化来避免多次递归导致的重复，这样求 $1,2,\ldots,n$ 中所有数的逆元的时间复杂度仍是 $O(n)$。

注意：如果用以上给出的式子递归进行单个数的逆元求解，目前已知的时间复杂度的上界为 $O(n^{\frac{1}{3}})$，具体请看 知乎讨论[1]。算法竞赛中更好地求单个数的逆元的方法有扩展欧几里得法和快速幂法。

线性求任意 n 个数的逆元

上面的方法只能求 1 到 n 的逆元，如果需要求任意给定 n 个数（$1\le a_i<p$）的逆元，就需要下面的方法：

首先计算 n 个数的前缀积，记为 s_i，然后使用快速幂或扩展欧几里得法计算 s_n 的逆元，记为 sv_n。

因为 sv_n 是 n 个数的积的逆元，所以当我们把它乘上 a_n 时，就会和 a_n 的逆元抵消，于是就得到了 a_1 到 a_{n-1} 的积逆元，记为 sv_{n-1}。

同理我们可以依次计算出所有的 sv_i，于是 a_i^{-1} 就可以用 $s_{i-1}\times sv_i$ 求得。

所以我们就在 $O(n+\log p)$ 的时间内计算出了 n 个数的逆元。

" 实现"

```cpp
s[0] = 1;
for (int i = 1; i <= n; ++i) s[i] = s[i - 1] * a[i] % p;
sv[n] = qpow(s[n], p - 2);
// 当然这里也可以用 exgcd 来求逆元，视个人喜好而定.
for (int i = n; i >= 1; --i) sv[i - 1] = sv[i] * a[i] % p;
for (int i = 1; i <= n; ++i) inv[i] = sv[i] * s[i - 1] % p;
```

```python
s[0] = 1
for i in range(1, n + 1):
    s[i] = s[i - 1] * a[i] % p
sv[n] = qpow(s[n], p - 2)
# 当然这里也可以用 exgcd 来求逆元，视个人喜好而定.
for i in range(n, 0, -1):
    sv[i - 1] = sv[i] * a[i] % p
for i in range(1, n + 1):
    inv[i] = sv[i] * s[i - 1] % p
```

逆元练习题

乘法逆元[2]

乘法逆元 2[3]

「NOIP2012」 同余方程[4]

「AHOI2005」 洗牌[5]

「SDOI2016」 排列计数[6]

参考资料与注释

[1] 知乎讨论

[2] 乘法逆元

[3] 乘法逆元 2

[4] 「NOIP2012」同余方程

[5] 「AHOI2005」洗牌

[6] 「SDOI2016」排列计数

9.12.13 线性同余方程

定义

形如

$$ax \equiv b \pmod n$$

的方程称为**线性同余方程**（Linear Congruence Equation）。其中，a、b 和 n 为给定整数，x 为未知数。需要从区间 $[0, n-1]$ 中求解 x，当解不唯一时需要求出全体解。

用逆元求解

首先考虑简单的情况，当 a 和 n 互素（coprime 或 relatively prime）时，即 $\gcd(a, n) = 1$。

此时可以计算 a 的逆元，并将方程的两边乘以 a 的逆元，可以得到唯一解。

证明

$$x \equiv ba^{-1} \pmod n$$

接下来考虑 a 和 n 不互素（not coprime），即 $\gcd(a, n) \neq 1$ 的情况。此时不一定有解。例如，$2x \equiv 1 \pmod 4$ 没有解。

设 $g = \gcd(a, n)$，即 a 和 n 的最大公约数，其中 a 和 n 在本例中大于 1。

当 b 不能被 g 整除时无解。此时，对于任意的 x，方程 $ax \equiv b \pmod n$ 的左侧始终可被 g 整除，而右侧不可被 g 整除，因此无解。

如果 g 整除 b，则通过将方程两边 a、b 和 n 除以 g，得到一个新的方程：

$$a^{'}x \equiv b^{'} \pmod {n^{'}}$$

其中 $a^{'}$ 和 $n^{'}$ 已经互素，这种情形已经解决，于是得到 $x^{'}$ 作为 x 的解。

很明显，$x^{'}$ 也将是原始方程的解。这不是唯一的解。可以看出，原始方程有如下 g 个解：

$$x_i \equiv (x^{'} + i \cdot n^{'}) \pmod n \quad \text{for } i = 0 \dots g-1$$

总之，线性同余方程的**解的数量**等于 $g = \gcd(a, n)$ 或等于 0。

用扩展欧几里得算法求解

根据以下两个定理，可以求出线性同余方程 $ax \equiv b \pmod n$ 的解。

定理 1：线性同余方程 $ax \equiv b \pmod n$ 可以改写为如下线性不定方程：

$$ax + nk = b$$

其中 x 和 k 是未知数。这两个方程是等价的，有整数解的充要条件为 $\gcd(a, n) \mid b$。

应用扩展欧几里德算法可以求解该线性不定方程。根据定理 1，对于线性不定方程 $ax + nk = b$，可以先用扩展欧几里得算法求出一组 x_0, k_0，也就是 $ax_0 + nk_0 = \gcd(a, n)$，然后两边同时除以 $\gcd(a, n)$，再乘 b。就得到了方程

$$a\frac{b}{\gcd(a, n)}x_0 + n\frac{b}{\gcd(a, n)}k_0 = b$$

于是找到方程的一个解。

定理 2：若 $\gcd(a, n) = 1$，且 x_0、k_0 为方程 $ax + nk = b$ 的一组解，则该方程的任意解可表示为：

$$x = x_0 + nt$$

$$k = k_0 - at$$

并且对任意整数 t 都成立。

根据定理 2，可以从已求出的一个解，求出方程的所有解。实际问题中，往往要求出一个最小整数解，也就是一个特解

$$x = (x \bmod t + t) \bmod t$$

其中有

$$t = \frac{n}{\gcd(a, n)}$$

如果仔细考虑，用扩展欧几里得算法求解与用逆元求解，两种方法是等价的。

实现

" 代码实现"

```
int ex_gcd(int a, int b, int& x, int& y) {
  if (b == 0) {
    x = 1;
    y = 0;
    return a;
  }
  int d = ex_gcd(b, a % b, x, y);
  int temp = x;
  x = y;
  y = temp - a / b * y;
  return d;
}
```

```
bool liEu(int a, int b, int c, int& x, int& y) {
  int d = ex_gcd(a, b, x, y);
  if (c % d != 0) return 0;
  int k = c / d;
  x *= k;
  y *= k;
  return 1;
}

def ex_gcd(a, b ,x, y):
  if b == 0:
      x = 1; y = 0
      return a
  d = ex_gcd(b, a % b, x, y)
  temp = x
  x = y
  y = temp - a // b * y
  return d

def liEu(a, b, c, x, y):
  d = ex_gcd(a, b, x, y)
  if c % d != 0:
      return 0
  k = c // d
  x = x * k
  y = y * k
  return 1
```


习题

「NOIP2012」 同余方程[3]

参考资料与注释

[1]

[2] Linear Congruence Equation

[3] 「NOIP2012」同余方程

9.12.14 中国剩余定理

引入

> 「物不知数」问题：有物不知其数，三三数之剩二，五五数之剩三，七七数之剩二。问物几何？

即求满足以下条件的整数：除以 3 余 2，除以 5 余 3，除以 7 余 2。

该问题最早见于《孙子算经》中，并有该问题的具体解法。宋朝数学家秦九韶于 1247 年《数书九章》卷一、二《大衍类》对「物不知数」问题做出了完整系统的解答。上面具体问题的解答口诀由明朝数学家程大位在《算法统宗》

中给出：

> 三人同行七十希，五树梅花廿一支，七子团圆正半月，除百零五便得知。

$2\times 70+3\times 21+2\times 15=233=2\times 105+23$，故答案为 23。

定义

中国剩余定理 (Chinese Remainder Theorem, CRT) 可求解如下形式的一元线性同余方程组（其中 $n_1,n_2,\cdots,n_k$ 两两互质）：

$$\begin{cases}x &\equiv a_1 \pmod {n_1}\\ x &\equiv a_2 \pmod {n_2}\\ &\vdots \\ x &\equiv a_k \pmod {n_k}\end{cases}$$

上面的「物不知数」问题就是一元线性同余方程组的一个实例。

过程

1. 计算所有模数的积 n；
2. 对于第 i 个方程：
   (a) 计算 $m_i=\frac{n}{n_i}$；
   (b) 计算 m_i 在模 n_i 意义下的 逆元 m_i^{-1}；
   (c) 计算 $c_i=m_im_i^{-1}$（**不要对** n_i **取模**）。
3. 方程组在模 n 意义下的唯一解为：$x=\sum_{i=1}^k a_ic_i \pmod n$。

实现

```cpp
LL CRT(int k, LL* a, LL* r) {
  LL n = 1, ans = 0;
  for (int i = 1; i <= k; i++) n = n * r[i];
  for (int i = 1; i <= k; i++) {
    LL m = n / r[i], b, y;
    exgcd(m, r[i], b, y);  // b * m mod r[i] = 1
    ans = (ans + a[i] * m * b % n) % n;
  }
  return (ans % n + n) % n;
}
```

```python
def CRT(k, a, r):
    n = 1; ans = 0
    for i in range(1, k + 1):
        n = n * r[i]
    for i in range(1, k + 1):
        m = n // r[i]; b = y = 0
        exgcd(m, r[i], b, y) # b * m mod r[i] = 1
        ans = (ans + a[i] * m * b % n) % n
    return (ans % n + n) % n
```

证明

我们需要证明上面算法计算所得的 x 对于任意 $i=1,2,\cdots,k$ 满足 $x\equiv a_i \pmod{n_i}$。

当 $i\neq j$ 时，有 $m_j\equiv 0 \pmod{n_i}$，故 $c_j\equiv m_j\equiv 0 \pmod{n_i}$。又有 $c_i\equiv m_i\cdot(m_i^{-1} \bmod n_i)\equiv 1 \pmod{n_i}$，所以我们有：

$$
\begin{aligned}
x &\equiv \sum_{j=1}^{k} a_jc_j &&\pmod{n_i}\\
&\equiv a_ic_i &&\pmod{n_i}\\
&\equiv a_i\cdot m_i\cdot(m_i^{-1} \bmod n_i) &&\pmod{n_i}\\
&\equiv a_i &&\pmod{n_i}
\end{aligned}
$$

即对于任意 $i=1,2,\cdots,k$，上面算法得到的 x 总是满足 $x\equiv a_i \pmod{n_i}$，即证明了解同余方程组的算法的正确性。

因为我们没有对输入的 a_i 作特殊限制，所以任何一组输入 $\{a_i\}$ 都对应一个解 x。

另外，若 $x\neq y$，则总存在 i 使得 x 和 y 在模 n_i 下不同余。

故系数列表 $\{a_i\}$ 与解 x 之间是一一映射关系，方程组总是有唯一解。

解释

下面演示 CRT 如何解「物不知数」问题。

1. $n=3\times5\times7=105$；
2. 三人同行**七十**希：$n_1=3, m_1=n/n_1=35, m_1^{-1}\equiv 2 \pmod 3$，故 $c_1=35\times2=70$；
3. 五树梅花**廿**一支：$n_2=5, m_2=n/n_2=21, m_2^{-1}\equiv 1 \pmod 5$，故 $c_2=21\times1=21$；
4. 七子团圆正**半月**：$n_3=7, m_3=n/n_3=15, m_3^{-1}\equiv 1 \pmod 7$，故 $c_3=15\times1=15$；
5. 所以方程组的唯一解为 $x\equiv 2\times70+3\times21+2\times15\equiv 233\equiv 23 \pmod{105}$。（除**百零五**便得知）

Garner 算法

CRT 的另一个用途是用一组比较小的质数表示一个大的整数。

例如，若 a 满足如下线性方程组，且 $a<\prod_{i=1}^{k} p_i$（其中 p_i 为质数）：

$$
\begin{cases}
a &\equiv a_1 \pmod{p_1}\\
a &\equiv a_2 \pmod{p_2}\\
&\vdots\\
a &\equiv a_k \pmod{p_k}
\end{cases}
$$

我们可以用以下形式的式子（称作 a 的混合基数表示）表示 a：

$$a=x_1+x_2p_1+x_3p_1p_2+\ldots+x_kp_1\ldots p_{k-1}$$

Garner 算法将用来计算系数 $x_1,\ldots,x_k$。

令 r_{ij} 为 p_i 在模 p_j 意义下的 逆：

$$p_i\cdot r_{i,j}\equiv 1 \pmod{p_j}$$

把 a 代入我们得到的第一个方程：

$$a_1\equiv x_1 \pmod{p_1}$$

代入第二个方程得出：

$$a_2\equiv x_1+x_2p_1 \pmod{p_2}$$

方程两边减 x_1，除 p_1 后得

$$
\begin{aligned}
a_2 - x_1 &\equiv x_2 p_1 && \pmod{p_2} \\
(a_2 - x_1) r_{1,2} &\equiv x_2 && \pmod{p_2} \\
x_2 &\equiv (a_2 - x_1) r_{1,2} && \pmod{p_2}
\end{aligned}
$$

类似地，我们可以得到：

$$x_k = (\dots((a_k - x_1) r_{1,k} - x_2) r_{2,k}) - \dots) r_{k-1,k} \bmod p_k$$

" 实现"

```cpp
for (int i = 0; i < k; ++i) {
  x[i] = a[i];
  for (int j = 0; j < i; ++j) {
    x[i] = r[j][i] * (x[i] - x[j]);
    x[i] = x[i] % p[i];
    if (x[i] < 0) x[i] += p[i];
  }
}
```

```python
for i in range(0, k):
    x[i] = a[i]
    for j in range(0, i):
        x[i] = r[j][i] * (x[i] - x[j])
        x[i] = x[i] % p[i]
        if (x[i] < 0):
            x[i] = x[i] + p[i]
```

该算法的时间复杂度为 $O(k^2)$。实际上 Garner 算法并不要求模数为质数，只要求模数两两互质，我们有如下伪代码：

Chinese Remainder Algorithm $\operatorname{cra}(\mathbf{v}, \mathbf{m})$:

Input: $\mathbf{m} = (m_0, m_1, \dots, m_{n-1})$, $m_i \in \mathbb{Z}^+ \wedge \gcd(m_i, m_j) = 1$ for all $i \neq j$,
$\mathbf{v} = (v_0, \dots, v_{n-1})$ where $v_i = x \bmod m_i$.

Output: $x \bmod \prod_{i=0}^{n-1} m_i$.

1 **for** i from 1 to $(n-1)$ **do**
2 $\quad C_i \leftarrow \left(\prod_{j=0}^{i-1} m_j\right)^{-1} \bmod m_i$
3 $x \leftarrow v_0$
4 **for** i from 1 to $(n-1)$ **do**
5 $\quad u \leftarrow (v_i - x) \cdot C_i \bmod m_i$
6 $\quad x \leftarrow x + u \prod_{j=0}^{i-1} m_j$
7 **return** (x)

可以发现在第六行中的计算过程对应上述混合基数的表示。

应用

某些计数问题或数论问题出于加长代码、增加难度、或者是一些其他原因，给出的模数：**不是质数**！

但是对其质因数分解会发现它没有平方因子，也就是该模数是由一些不重复的质数相乘得到。

那么我们可以分别对这些模数进行计算，最后用 CRT 合并答案。

下面这道题就是一个不错的例子。

"洛谷 P2480 [SDOI2010] 古代猪文[1]"

给出 G,n（$1 \le G,n \le 10^9$），求：

$$G^{\sum_{k\mid n}\binom{n}{k}} \bmod 999\ 911\ 659$$

首先，当 $G = 999\ 911\ 659$ 时，所求显然为 0。

否则，根据 欧拉定理，可知所求为：

$$G^{\sum_{k\mid n}\binom{n}{k} \bmod 999\ 911\ 658} \bmod 999\ 911\ 659$$

现在考虑如何计算：

$$\sum_{k\mid n}\binom{n}{k} \bmod 999\ 911\ 658$$

因为 999 911 658 不是质数，无法保证 $\forall x \in [1, 999\ 911\ 657]$，$x$ 都有逆元存在，上面这个式子我们无法直接计算。

注意到 $999\ 911\ 658 = 2 \times 3 \times 4679 \times 35617$，其中每个质因子的最高次数均为一，我们可以考虑分别求出 $\sum_{k\mid n}\binom{n}{k}$ 在模 2，3，4679，35617 这几个质数下的结果，最后用中国剩余定理来合并答案。

也就是说，我们实际上要求下面一个线性方程组的解：

$$\begin{cases} x \equiv a_1 \pmod 2 \\ x \equiv a_2 \pmod 3 \\ x \equiv a_3 \pmod{4679} \\ x \equiv a_4 \pmod{35617} \end{cases}$$

而计算一个组合数对较小的质数取模后的结果，可以利用 卢卡斯定理。

扩展：模数不互质的情况

两个方程

设两个方程分别是 $x \equiv a_1 \pmod{m_1}$、$x \equiv a_2 \pmod{m_2}$；

将它们转化为不定方程：$x = m_1 p + a_1 = m_2 q + a_2$，其中 p, q 是整数，则有 $m_1 p - m_2 q = a_2 - a_1$。

由 裴蜀定理，当 $a_2 - a_1$ 不能被 $\gcd(m_1, m_2)$ 整除时，无解；

其他情况下，可以通过 扩展欧几里得算法 解出来一组可行解 (p, q)；

则原来的两方程组成的模方程组的解为 $x \equiv b \pmod M$，其中 $b = m_1 p + a_1$，$M = \operatorname{lcm}(m_1, m_2)$。

多个方程

用上面的方法两两合并即可。

习题

- 【模板】中国剩余定理（CRT）/ 曹冲养猪[2]
- 【模板】扩展中国剩余定理[3]
- 「NOI2018」屠龙勇士[4]
- 「TJOI2009」猜数字[5]

本页面部分内容译自博文 [6] 与其英文翻译版 Chinese Remainder Theorem[7]。其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0。

参考资料与注释

[1] 洛谷 P2480 [SDOI2010] 古代猪文

[2] 【模板】中国剩余定理（CRT）/曹冲养猪

[3] 【模板】扩展中国剩余定理

[4] 「NOI2018」屠龙勇士

[5] 「TJOI2009」猜数字

[6]

[7] Chinese Remainder Theorem

9.12.15 升幂引理

内容

升幂（Lift the Exponent，LTE）引理是初等数论中比较常用的一个定理。

定义 $v_p(n)$ 为整数 n 的标准分解中素因子 p 的幂次，即 $v_p(n)$ 满足 $p^{v_p(n)} \mid n$ 且 $p^{v_p(n)+1} \nmid n$.

由于升幂引理内容较长，我们将其分为三部分介绍：

以下内容设 p 为素数，x,y 为满足 $p \nmid x$ 且 $p \nmid y$ 的整数，n 为正整数。

第一部分

对所有的素数 p 和满足 $(n,p)=1$ 的整数 n，

1. 若 $p \mid x-y$，则：
$$v_p\left(x^n-y^n\right)=v_p(x-y)$$
2. 若 $p \mid x+y$，则对奇数 n 有：
$$v_p\left(x^n+y^n\right)=v_p(x+y)$$

”证明”

若 $p \mid x-y$，则不难发现 $p \mid x-y \iff x \equiv y \pmod p$，则显然有：

$$\sum_{i=0}^{n-1} x^i y^{n-1-i} \equiv n x^{n-1} \not\equiv 0 \pmod p$$

进而由 $x^n-y^n=(x-y)\sum_{i=0}^{n-1} x^i y^{n-1-i}$ 可知命题得证。

对 $p \mid x+y$ 的情况证明方法类似。

第二部分

若 p 是奇素数，

1. 若 $p \mid x-y$，则：
$$v_p\left(x^n-y^n\right)=v_p(x-y)+v_p(n)$$
2. 若 $p \mid x+y$，则对奇数 n 有：
$$v_p\left(x^n+y^n\right)=v_p(x+y)+v_p(n)$$

" 证明"

若 $p \mid x-y$，令 $y=x+kp$，我们只需证明 $p \mid n$ 的情况。

- 若 $n=p$，则由二项式定理：

$$\begin{aligned}\sum_{i=0}^{p-1} x^{p-1-i} y^i &= \sum_{i=0}^{p-1} x^{p-1-i} \sum_{j=0}^{i} \binom{i}{j} x^j (kp)^{i-j} \\ &\equiv p x^{p-1} \pmod{p^2}\end{aligned}$$

  从而

$$v_p\left(x^n-y^n\right)=v_p(x-y)+1$$

- 若 $n=p^a$，则由数学归纳法可得

$$v_p\left(x^n-y^n\right)=v_p(x-y)+a$$

因此命题得证。

对 $p \mid x+y$ 的情况证明方法类似。

第三部分

若 $p=2$ 且 $p \mid x-y$，

1. 对奇数 n 有（与第一部分的 1 相同）：

$$v_p\left(x^n-y^n\right)=v_p(x-y)$$

2. 对偶数 n 有：

$$v_p\left(x^n-y^n\right)=v_p(x-y)+v_p(x+y)+v_p(n)-1$$

另外对上述的 x, y, n，我们有：

若 $4 \mid x-y$，则：

- $v_2(x+y)=1$
- $v_2\left(x^n-y^n\right)=v_2(x-y)+v_2(n)$

" 证明"

我们只需证明 n 为偶数的情况。由于此时 $p \nmid \binom{p}{2}$，故我们不能用第二部分的方法证明。

令 $n=2^a b$，其中 $a=v_p(n)$，$2 \nmid b$，从而

$$\begin{aligned}v_p\left(x^n-y^n\right) &= v_p\left(x^{2^a}-y^{2^a}\right) \\ &= v_p\left((x-y)(x+y) \prod_{i=1}^{a-1}\left(x^{2^i}+y^{2^i}\right)\right)\end{aligned}$$

注意到 $2 \mid x-y \implies 4 \mid x^2-y^2$，从而 $(\forall i \geq 1)$，$x^{2^i}+y^{2^i} \equiv 2 \pmod 4$，进而上式可变为：

$$v_p\left(x^n-y^n\right)=v_p(x-y)+v_p(x+y)+v_p(n)-1$$

因此命题得证。

参考资料

1. Lifting-the-exponent lemma - Wikipedia[1]

参考资料与注释

[1] Lifting-the-exponent lemma - Wikipedia

9.12.16 威尔逊定理

内容

"Wilson 定理"

对于素数 p 有 $(p-1)! \equiv -1 \pmod p$.

" 证明"

我们可以利用 同余方程 或 原根 得到两种简洁的证明，此处略去不表。

我们选择介绍前置知识较少的一种证明方法：

当 $p=2$ 时，命题显然成立。

以下设 $p \geq 3$，此时我们要证 $\mathbf{Z}_p$ 中所有非零元素的积为 $\overline{-1}$.

我们知道 $\mathbf{Z}_p$ 中所有非零元素 a 都有逆元 a^{-1}，于是 $\mathbf{Z}_p$ 中彼此互逆的元素乘积为 $\overline{1}$.

但是要注意 a 和 a^{-1} 可能相等。若 $a=a^{-1}$，则 $a^2 \equiv 1 \pmod p$，即

$$0 \equiv a^2-1 \equiv (a+1)(a-1) \pmod p$$

从而 $a \equiv 1 \pmod p$ 或 $a \equiv -1 \pmod p$.

这说明 $\mathbf{Z}_p$ $\{\overline{0},\overline{1},\overline{-1}\}$ 中所有元素的乘积为 $\overline{1}$, 进而 $\mathbf{Z}_p$ 中所有非零元素的积为 $\overline{-1}$.

对于整数 n，令 $(n!)_p$ 表示所有小于等于 n 但不能被 p 整除的正整数的乘积，即 $(n!)_p = n!/(\lfloor n/p \rfloor ! p^{\lfloor n/p \rfloor})$.

Wilson 定理指出 $(p!)_p = (p-1)! \equiv -1 \pmod p$ 且可被推广至模素数 p 的幂次。

应用

阶乘与素数

在某些情况下，有必要考虑以某个素数 p 为模的公式，包含分子和分母中的阶乘，就像在二项式系数公式中遇到的那样。

只有当阶乘同时出现在分数的分子和分母中时，这个问题才有意义。否则，后续项 $p!$ 将减少为零。但在分数中，因子 p 可以抵消，结果将是非零。

因此，要计算 $n! \bmod p$，而不考虑阶乘中出现因数 p。写下素因子分解，去掉所有因子 p，并计算乘积模。

用 $(n!)_p$ 表示这个修改的因子。例如：

$$(7!)_p \equiv 1 \cdot 2 \cdot \underbrace{1}_{3} \cdot 4 \cdot 5 \underbrace{2}_{6} \cdot 7 \equiv 2 \bmod 3$$

这种修正的阶乘，可用于快速计算各种带取模和组合数的公式的值。

计算余数的算法

计算上述去掉因子 p 的阶乘模 p 的余数。

$$(n!)_p = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-2) \cdot (p-1) \cdot \underbrace{1}_{p} \cdot (p+1) \cdot (p+2) \cdot \ldots \cdot (2p-1) \cdot \underbrace{2}_{2p}$$
$$\cdot (2p+1) \cdot \ldots \cdot (p^2-1) \cdot \underbrace{1}_{p^2} \cdot (p^2+1) \cdot \ldots \cdot n \pmod p$$

$$= 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-2) \cdot (p-1) \cdot \underbrace{1}_{p} \cdot 1 \cdot 2 \cdot \ldots \cdot (p-1) \cdot \underbrace{2}_{2p} \cdot 1 \cdot 2$$
$$\cdot \ldots \cdot (p-1) \cdot \underbrace{1}_{p^2} \cdot 1 \cdot 2 \cdot \ldots \cdot (n \bmod p) \pmod p$$

可以清楚地看到，除了最后一个块外，阶乘被划分为几个长度相同的块。

$$(n!)_p = \underbrace{1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-2) \cdot (p-1) \cdot 1}_{1st} \cdot \underbrace{1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-2) \cdot (p-1) \cdot 2}_{2nd} \cdot \ldots$$

$$\cdot \underbrace{1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-2) \cdot (p-1) \cdot 1}_{p\text{th}} \cdot \ldots \cdot \quad \underbrace{1 \cdot 2 \cdot \cdot \ldots \cdot (n \bmod p)}_{\text{tail}} \pmod p.$$

块的主要部分 $(p-1)! \bmod p$ 很容易计算，可以应用 Wilson 定理：

$$(p-1)! \equiv -1 \pmod p$$

总共有 $\lfloor \frac{n}{p} \rfloor$ 个块，因此需要将 $\lfloor \frac{n}{p} \rfloor$ 写到 -1 的指数上。可以注意到结果在 -1 和 1 之间切换，因此只需要查看指数的奇偶性，如果是奇数，则乘以 -1。除了使用乘法，也可以从结果中减去 p.

最后一个部分块的值可以在 $O(p)$ 的时间复杂度单独计算。

这只留下每个块的最后一个元素。如果隐藏已处理的元素，可以看到以下模式：

$$(n!)_p = \underbrace{\ldots \cdot 1} \cdot \underbrace{\ldots \cdot 2} \cdot \ldots \cdot \underbrace{\ldots \cdot (p-1)} \cdot \underbrace{\ldots \cdot 1} \cdot \underbrace{\ldots \cdot 1} \cdot \underbrace{\ldots \cdot 2} \cdots$$

这也是一个修正的阶乘，只是维数小得多。它是：

$$\left(\left\lfloor \frac{n}{p} \right\rfloor !\right)_p$$

因此，在计算修改的阶乘 $(n!)_p$ 中，执行了 $O(p)$ 个操作，剩下的是计算 $(\lfloor \frac{n}{p} \rfloor !)_p$，于是有了递归，递归深度为 $O(\log_p n)$，因此算法的总时间复杂度为 $O(p \log_p n)$.

如果预先计算阶乘 $0!, 1!, 2!, \ldots, (p-1)!$ 模 p，那么时间复杂度为 $O(\log_p n)$.

计算余数算法的实现

具体实现不需要递归，因为这是尾部递归的情况，因此可以使用迭代轻松实现。在下面的实现中预计算了阶乘 $0!, 1!, \ldots, (p-1)!$.

因此时间复杂度为 $O(p + \log_p n)$. 如果需要多次调用函数，则可以在函数外部进行预计算，于是计算 $(n!)_p$ 的时间复杂度为 $O(\log_p n)$.

" 实现"

```cpp
int factmod(int n, int p) {
  vector<int> f(p);
  f[0] = 1;
  for (int i = 1; i < p; i++) f[i] = f[i - 1] * i % p;
  int res = 1;
  while (n > 1) {
    if ((n / p) % 2) res = p - res;
```

```
    res = res * f[n % p] % p;
    n /= p;
  }
  return res;
}
```

如果空间有限，无法存储所有阶乘，也可以只存储需要的阶乘，对它们进行排序，然后计算阶乘 $0!$, $1!$, $2!$, $\dots$, $(p-1)!$ 而不显式存储它们。

Legendre 公式

如果想计算二项式系数模 p，那么还需要考虑在 n 的阶乘的素因子分解中 p 出现的次数，或在计算修改因子时删除因子 p 的个数。

"Legendre 公式"

$n!$ 中含有的素数 p 的幂次 $v_p(n!)$ 为：

$$v_p(n!) = \sum_{i=1}^{\infty} \left\lfloor \frac{n}{p^i} \right\rfloor = \frac{n - S_p(n)}{p-1}$$

其中 $S_p(n)$ 为 p 进制下 n 的各个数位的和。

特别地，阶乘中 2 的幂次是 $v_2(n!) = n - S_2(n)$

" 证明"

将 $n!$ 记为 $1\times 2\times\cdots\times p\times\cdots\times 2p\times\cdots\times\lfloor n/p\rfloor p\times\cdots\times n$ 那么其中 p 的倍数有 $p\times 2p\times\cdots\times\lfloor n/p\rfloor p = p^{\lfloor n/p\rfloor}\lfloor n/p\rfloor!$ 然后在 $\lfloor n/p\rfloor!$ 中继续寻找 p 的倍数即可，这是一个递归的过程。

第二个等号与等比数列求和公式很相似。由于涉及各位数字和，利用数学归纳法可以轻松证明。

" 实现"

```
int multiplicity_factorial(int n, int p) {
  int count = 0;
  do {
    n /= p;
    count += n;
  } while (n);
  return count;
}
```

时间复杂度 $O(\log n)$

以下记 $\nu(n!) = \sum_{j\ge 1}\lfloor n/p^j\rfloor$.

Kummer 定理

组合数对一个数取模的结果，往往构成分形结构，例如谢尔宾斯基三角形就可以通过组合数模 2 得到。

如果仔细分析，p 是否整除组合数其实和上下标在 p 进制下减法是否需要借位有关。这就有了 **Kummer 定理**。

"Kummer 定理"

p 在组合数 $\binom{m}{n}$ 中的幂次，恰好是 p 进制下 m 减掉 n 需要借位的次数。

即

$$v_p\left(\binom{m}{n}\right)=\frac{S_p(n)+S_p(m-n)-S_p(m)}{p-1}$$

特别地，组合数中 2 的幂次是 $v_2\left(\binom{m}{n}\right)=S_2(n)+S_2(m-n)-S_2(m)$.

Wilson 定理的推广

"Wilson 定理的推广"

对于素数 p 和正整数 q 有 $(p^q!)_p\equiv\pm 1 \pmod{p^q}$.

" 证明"

依然考虑配对一个数与其逆元，也就是考虑关于 m 的同余方程 $m^2\equiv 1 \pmod{p^q}$ 的根的乘积，当 $p^q=2$ 时方程仅有一根，当 $p=2$ 且 $q\geq 3$ 时有四根为 $\pm 1, 2^{q-1}\pm 1$ 其他时候则有两根为 ± 1.

至此我们对 Wilson 定理的推广中的 ± 1 有了详细的定义，即

$$(p^q!)_p\equiv\begin{cases}1, & (p=2)\wedge(q\geq 3),\\ -1, & \text{otherwise}.\end{cases}$$

下文两个推论中的 ± 1，均特指这样的定义：当模数 p^q 取 8 及以上的 2 的幂时取 1，其余取 -1.

" 推论 1"

对于素数 p、正整数 q、非负整数 n 和 $N_0=n \bmod p^q$ 有

$$(n!)_p\equiv(\pm 1)^{\lfloor n/p^q\rfloor}(N_0!)_p \pmod{p^q}$$

" 证明"

令 $\prod'$ 表示不能被 p 整除的数的乘积，有

$$\begin{aligned}(n!)_p&=\prod_{1\leq r\leq n}{}' r\\&=\left(\prod_{i=0}^{\lfloor n/p^q\rfloor-1}\prod_{1\leq j\leq p^q}{}'(ip^q+j)\right)\left(\prod_{1\leq j\leq N_0}{}'(\lfloor n/p^q\rfloor p^q+j)\right)\\&\equiv((p^q!)_p)^{\lfloor n/p^q\rfloor}(N_0!)_p\\&\equiv(\pm 1)^{\lfloor n/p^q\rfloor}(N_0!)_p \pmod{p^q}\end{aligned}$$

将 $1\times 2\times 3\times\cdots\times n$ 记为 $(0\times p^q+1)\times(0\times p^q+2)\times\cdots\times(\lfloor n/p^q\rfloor p^q+N_0)$ 就得到了上述第二行。

至此得到了：

" 推论 2"

对于素数 p 和正整数 q 和非负整数 n 有

$$\frac{n!}{p^{\sum_{j\geq 1}\lfloor\frac{n}{p^j}\rfloor}}\equiv(\pm 1)^{\sum_{j\geq q}\lfloor\frac{n}{p^j}\rfloor}\prod_{j\geq 0}(N_j!)_p \pmod{p^q}$$

其中 $N_j=\lfloor n/p^j\rfloor \bmod p^q$ 而 ± 1 与上述相同。

记 $r=n-m$ 且 $n>m$ 有

$$\frac{(\pm 1)^{\sum_{j\geq q}(\lfloor n/p^j\rfloor-\lfloor m/p^j\rfloor-\lfloor r/p^j\rfloor)}}{p^{\nu(n!)-\nu(m!)-\nu(r!)}}\binom{n}{m}\equiv\frac{n!/p^{\nu(n!)}}{(m!/p^{\nu(m!)})(r!/p^{\nu(r!)})}\pmod{p^q}$$

右边的分母中括号内的项均在模 p^q 意义下均存在逆元，可直接计算，而 ± 1 的与上述相同。

例题

" 例题 HDU 2973 - YAPTCHA[1]"

给定 n, 计算

$$\sum_{k=1}^{n}\left\lfloor\frac{(3k+6)!+1}{3k+7}-\left\lfloor\frac{(3k+6)!}{3k+7}\right\rfloor\right\rfloor$$

" 解题思路"

若 $3k+7$ 是质数，则

$$(3k+6)!\equiv -1\pmod{3k+7}$$

设 $(3k+6)!+1=k(3k+7)$

则

$$\left\lfloor\frac{(3k+6)!+1}{3k+7}-\left\lfloor\frac{(3k+6)!}{3k+7}\right\rfloor\right\rfloor=\left\lfloor k-\left\lfloor k-\frac{1}{3k+7}\right\rfloor\right\rfloor=1$$

若 $3k+7$ 不是质数，则有 $(3k+7)\mid(3k+6)!$，即

$$(3k+6)!\equiv 0\pmod{3k+7}$$

设 $(3k+6)!=k(3k+7)$，则

$$\left\lfloor\frac{(3k+6)!+1}{3k+7}-\left\lfloor\frac{(3k+6)!}{3k+7}\right\rfloor\right\rfloor=\left\lfloor k+\frac{1}{3k+7}-k\right\rfloor=0$$

因此

$$\sum_{k=1}^{n}\left\lfloor\frac{(3k+6)!+1}{3k+7}-\left\lfloor\frac{(3k+6)!}{3k+7}\right\rfloor\right\rfloor=\sum_{k=1}^{n}[3k+7\text{ is prime}]$$

" 参考代码"

```cpp
#include <iostream>

const int M = 1e6 + 5, N = 3 * M + 7;

bool not_prime[N];
int sum[M];

int main() {
  for (int i = 2; i < N; ++i)
    if (!not_prime[i])
      for (int j = 2; j * i < N; ++j) not_prime[j * i] = 1;
  for (int i = 1; i < M; ++i) sum[i] = sum[i - 1] + !not_prime[3 * i + 7];

  int t;
  std::cin >> t;
```

```
  while (t--) {
    int n;
    std::cin >> n;
    std::cout << sum[n] << std::endl;
  }
}
```


参考资料

1. 冯克勤。初等数论及其应用。

参考资料与注释

[1] HDU 2973 - YAPTCHA

[2]

[3] Factorial modulo p

9.12.17 卢卡斯定理

Lucas 定理

引入

Lucas 定理用于求解大组合数取模的问题，其中模数必须为素数。正常的组合数运算可以通过递推公式求解（详见 排列组合），但当问题规模很大，而模数是一个不大的质数的时候，就不能简单地通过递推求解来得到答案，需要用到 Lucas 定理。

定义

Lucas 定理内容如下：对于质数 p，有

$$\binom{n}{m} \bmod p = \binom{\lfloor n/p \rfloor}{\lfloor m/p \rfloor} \cdot \binom{n \bmod p}{m \bmod p} \bmod p$$

观察上述表达式，可知 $n \bmod p$ 和 $m \bmod p$ 一定是小于 p 的数，可以直接求解，$\binom{\lfloor n/p \rfloor}{\lfloor m/p \rfloor}$ 可以继续用 Lucas 定理求解。这也就要求 p 的范围不能够太大，一般在 10^5 左右。边界条件：当 $m = 0$ 的时候，返回 1。

时间复杂度为 $O(f(p) + g(n)\log n)$，其中 $f(n)$ 为预处理组合数的复杂度，$g(n)$ 为单次求组合数的复杂度。

" 实现"

```
long long Lucas(long long n, long long m, long long p) {
  if (m == 0) return 1;
  return (C(n % p, m % p, p) * Lucas(n / p, m / p, p)) % p;
}

def Lucas(n, m, p):
  if m == 0:
```

```
        return 1
    return (C(n % p, m % p, p) * Lucas(n // p, m // p, p)) % p
```

证明

考虑 $\binom{p}{n} \bmod p$ 的取值，注意到 $\binom{p}{n} = \frac{p!}{n!(p-n)!}$，分子的质因子分解中 p 的次数恰好为 1，因此只有当 $n = 0$ 或 $n = p$ 的时候 $n!(p-n)!$ 的质因子分解中含有 p，因此 $\binom{p}{n} \bmod p = [n = 0 \vee n = p]$。进而我们可以得出

$$
\begin{aligned}
(a+b)^p &= \sum_{n=0}^{p} \binom{p}{n} a^n b^{p-n} && (9.5)\\
&\equiv \sum_{n=0}^{p} [n = 0 \vee n = p] a^n b^{p-n} && (9.6)\\
&\equiv a^p + b^p \pmod p && (9.7)
\end{aligned}
$$

注意过程中没有用到费马小定理，因此这一推导不仅适用于整数，亦适用于多项式。因此我们可以考虑二项式 $f^p(x) = (ax^n + bx^m)^p \bmod p$ 的结果

$$
\begin{aligned}
(ax^n + bx^m)^p &\equiv a^p x^{pn} + b^p x^{pm} && (9.8)\\
&\equiv ax^{pn} + bx^{pm} && (9.9)\\
&\equiv f(x^p) && (9.10)
\end{aligned}
$$

考虑二项式 $(1+x)^n \bmod p$，那么 $\binom{n}{m}$ 就是求其在 x^m 次项的取值。使用上述引理，我们可以得到

$$
\begin{aligned}
(1+x)^n &\equiv (1+x)^{p\lfloor n/p \rfloor} (1+x)^{n \bmod p} && (9.11)\\
&\equiv (1+x^p)^{\lfloor n/p \rfloor} (1+x)^{n \bmod p} && (9.12)
\end{aligned}
$$

注意前者只有在 p 的倍数位置才有取值，而后者最高次项为 $n \bmod p \le p-1$，因此这两部分的卷积在任何一个位置只有最多一种方式贡献取值，即在前者部分取 p 的倍数次项，后者部分取剩余项，即 $\binom{n}{m} \bmod p = \binom{\lfloor n/p \rfloor}{\lfloor m/p \rfloor} \cdot \binom{n \bmod p}{m \bmod p} \bmod p$。

exLucas 定理

Lucas 定理中对于模数 p 要求必须为素数，那么对于 p 不是素数的情况，就需要用到 exLucas 定理。

过程

第一部分：中国剩余定理

要求计算二项式系数 $\binom{n}{m} \bmod M$，其中 M 可能为合数。

考虑利用 中国剩余定理 合并答案，这种情况下我们只需求出 $\binom{n}{m} \bmod p^\alpha$ 的值即可（其中 p 为素数且 α 为正整数）。

根据**唯一分解定理**，将 M 质因数分解：

$$
M = {p_1}^{\alpha_1} \cdot {p_2}^{\alpha_2} \cdots {p_r}^{\alpha_r} = \prod_{i=1}^{r} {p_i}^{\alpha_i}
$$

对于任意 i,j，有 ${p_i}^{\alpha_i}$ 与 ${p_j}^{\alpha_j}$ 互质，所以可以构造如下 r 个同余方程：

$$\begin{cases} a_1 \equiv \binom{n}{m} \pmod{{p_1}^{\alpha_1}} \\ a_2 \equiv \binom{n}{m} \pmod{{p_2}^{\alpha_2}} \\ \cdots \\ a_r \equiv \binom{n}{m} \pmod{{p_r}^{\alpha_r}} \end{cases}$$

我们发现，在求出 a_i 后，就可以用中国剩余定理求解出 $\binom{n}{m}$。

第二部分：移除分子分母中的素数

根据同余的定义，$a_i = \binom{n}{m} \bmod {p_i}^{\alpha_i}$，问题转化成，求 $\binom{n}{m} \bmod p^\alpha$（$p$ 为质数）的值。

根据组合数定义 $\binom{n}{m} = \frac{n!}{m!(n-m)!}$，$\binom{n}{m} \bmod p^\alpha = \frac{n!}{m!(n-m)!} \bmod p^\alpha$。

由于式子是在模 p^α 意义下，所以分母要算乘法逆元。

同余方程 $ax \equiv 1 \pmod p$（即乘法逆元）**有解**的充要条件为 $\gcd(a,p)=1$（裴蜀定理），

然而**无法保证有解**，发现无法直接求 $\mathrm{inv}_{m!}$ 和 $\mathrm{inv}_{(n-m)!}$，

所以将原式转化为：

$$\frac{\frac{n!}{p^x}}{\frac{m!}{p^y}\frac{(n-m)!}{p^z}} p^{x-y-z} \bmod p^\alpha$$

x 表示 $n!$ 中包含多少个 p 因子，y,z 同理。

第三部分：Wilson 定理的推论

问题转化成，求形如：

$$\frac{n!}{q^x} \bmod q^k$$

的值。这时可以利用 Wilson 定理的推论。如果难以理解，可以看看下面的解释。

解释

一个示例：$22! \bmod 9$

先考虑 $n! \bmod q^k$，

比如 $n=22, q=3, k=2$ 时：

$22! = 1\times2\times3\times4\times5\times6\times7\times8\times9\times10\times11\times12$

$\times13\times14\times15\times16\times17\times18\times19\times20\times21\times22$

将其中所有 q 的倍数提取，得到：

$22! = 3^7 \times (1\times2\times3\times4\times5\times6\times7)\times(1\times2\times4\times5\times7\times8\times10\times11\times13\times14\times16\times17\times19\times20\times22)$

可以看到，式子分为三个整式的乘积：

1. 是 3 的幂，次数是 $\lfloor\frac{n}{q}\rfloor$；
2. 是 $7!$，即 $\lfloor\frac{n}{q}\rfloor!$，由于阶乘中仍然可能有 q 的倍数，考虑递归求解；
3. 是 $n!$ 中与 q 互质的部分的乘积，具有如下性质：

   $1\times2\times4\times5\times7\times8 \equiv 10\times11\times13\times14\times16\times17 \pmod{3^2}$，

   即：$\prod_{i,(i,q)=1}^{q^k} i \equiv \prod_{i,(i,q)=1}^{q^k} (i+tq^k) \pmod{q^k}$（$t$ 是任意正整数）。

$\prod_{i,(i,q)=1}^{q^k} i$ 一共循环了 $\lfloor \frac{n}{q^k} \rfloor$ 次，暴力求出 $\prod_{i,(i,q)=1}^{q^k} i$，然后用快速幂求 $\lfloor \frac{n}{q^k} \rfloor$ 次幂。

最后要乘上 $\prod_{i,(i,q)=1}^{n \bmod q^k} i$，即 $19 \times 20 \times 22$，显然长度小于 q^k，暴力乘上去。

上述三部分乘积为 $n!$。最终要求的是 $\frac{n!}{q^x} \bmod q^k$。

所以有：

$$n! = q^{\lfloor \frac{n}{q} \rfloor} \cdot \left(\left\lfloor \frac{n}{q} \right\rfloor\right)! \cdot \left(\prod_{i,(i,q)=1}^{q^k} i\right)^{\lfloor \frac{n}{q^k} \rfloor} \cdot \left(\prod_{i,(i,q)=1}^{n \bmod q^k} i\right)$$

于是：

$$\frac{n!}{q^{\lfloor \frac{n}{q} \rfloor}} = \left(\left\lfloor \frac{n}{q} \right\rfloor\right)! \cdot \left(\prod_{i,(i,q)=1}^{q^k} i\right)^{\lfloor \frac{n}{q^k} \rfloor} \cdot \left(\prod_{i,(i,q)=1}^{n \bmod q^k} i\right)$$

$\left(\left\lfloor \frac{n}{q} \right\rfloor\right)!$ **同样是一个数的阶乘，所以也可以分为上述三个部分，于是可以递归求解。**

等式的右边两项不含素数 q。事实上，如果直接把 n 的阶乘中所有 q 的幂都拿出来，等式右边的阶乘也照做，这个等式可以直接写成：

$$\frac{n!}{q^x} = \frac{\left(\left\lfloor \frac{n}{q} \right\rfloor\right)!}{q^{x'}} \cdot \left(\prod_{i,(i,q)=1}^{q^k} i\right)^{\lfloor \frac{n}{q^k} \rfloor} \cdot \left(\prod_{i,(i,q)=1}^{n \bmod q^k} i\right)$$

式中的 x 和 x' 都表示把分子中所有的素数 q 都拿出来。改写成这样，每一项就完全不含 q 了。

递归的结果，三个部分中，左边部分随着递归结束而自然消失，中间部分可以利用 Wilson 定理的推论 0，右边部分就是推论 2 中的 $\prod_{j\geq 0}(N_j!)_p$。

下面这种写法，拥有单次询问 $O(p\log p)$ 的时间复杂度。其中 `int inverse(int x)` 函数返回 x 在模 p 意义下的逆元。

" 实现"

```
LL calc(LL n, LL x, LL P) {
  if (!n) return 1;
  LL s = 1;
  for (LL i = 1; i <= P; i++)
    if (i % x) s = s * i % P;
  s = Pow(s, n / P, P);
  for (LL i = n / P * P + 1; i <= n; i++)
    if (i % x) s = i % P * s % P;
  return s * calc(n / x, x, P) % P;
}

LL multilucas(LL m, LL n, LL x, LL P) {
  int cnt = 0;
  for (LL i = m; i; i /= x) cnt += i / x;
  for (LL i = n; i; i /= x) cnt -= i / x;
  for (LL i = m - n; i; i /= x) cnt -= i / x;
  return Pow(x, cnt, P) % P * calc(m, x, P) % P * inverse(calc(n, x, P), P) %
         P * inverse(calc(m - n, x, P), P) % P;
}

LL exlucas(LL m, LL n, LL P) {
  int cnt = 0;
```

```
  LL p[20], a[20];
  for (LL i = 2; i * i <= P; i++) {
    if (P % i == 0) {
      p[++cnt] = 1;
      while (P % i == 0) p[cnt] = p[cnt] * i, P /= i;
      a[cnt] = multilucas(m, n, i, p[cnt]);
    }
  }
  if (P > 1) p[++cnt] = P, a[cnt] = multilucas(m, n, P, P);
  return CRT(cnt, a, p);
}
```

若不考虑 excrt 的复杂度，通过预处理 $\frac{n!}{n_p}$ mod p，可以使时间复杂度优化至单次 $O(p+\log p)$。而如果 p 是固定的，我们在一开始就可以对 p 进行分解，并进行预处理，可以达到总复杂度 $O(p+T\log p)$。

习题

- Luogu3807【 模板 】 卢卡斯定理[1]
- SDOI2010 古代猪文　 卢卡斯定理[2]
- Luogu4720【 模板 】 扩展卢卡斯[3]
- Ceizenpok' s formula[4]

参考资料与注释

[1] Luogu3807【模板】卢卡斯定理

[2] SDOI2010 古代猪文卢卡斯定理

[3] Luogu4720【模板】扩展卢卡斯

[4] Ceizenpok' s formula

9.12.18 同余方程

Authors: iamtwz, aofall, CCXXXI, CoelacanthusHex, Great-designer, Marcythm, Persdre, shuzhouliu, Tiphereth-A, Xeonacid

定义

" 同余方程"

对正整数 m 和一元整系数多项式 $f(x)=\sum_{i=0}^n a_ix^i$，其中未知数 $x\in\mathbf{Z}_m$，称形如

$$f(x)\equiv 0 \pmod m \tag{1}$$

的方程为关于未知数 x 的模 m 的一元**同余方程**（Congruence Equation）。

若 $a_n\not\equiv 0 \pmod m$，则称上式为 n 次同余方程。

类似可定义同余方程组。

关于一次同余方程与方程组的相关内容请参见 线性同余方程 与 中国剩余定理。

本文首先研究同余方程的可解性和解集结构，之后会简要介绍高次同余方程的解法。

由 中国剩余定理 可知，求解关于模合数 m 的同余方程可转化为求解模素数幂次的情况。故以下只介绍素数幂模同余方程和素数模同余方程的相关理论。

素数幂模同余方程

以下假设模数 $m = p^a$ $(p \in \mathbf{P}, a \in \mathbf{Z}_{>1})$.

注意到若 x_0 是方程

$$f(x) \equiv 0 \pmod{p^a}$$

的解，则 x_0 是方程

$$f(x) \equiv 0 \pmod{p^{a-1}}$$

的解，这启发我们尝试通过较低的模幂次的解去构造较高的模幂次的解。我们有如下定理：

定理 1

对素数 p 和整数 $a > 1$，取整系数多项式 $f(x) = \sum_{i=0}^{n} a_i x^i$ $(p^a \nmid a_n)$，令 $f'(x) = \sum_{i=1}^{n} i a_i x^{i-1}$ 为其导数。令 x_0 为方程

$$f(x) \equiv 0 \pmod{p^{a-1}} \tag{2}$$

的解，则：

1. 若 $f'(x_0) \not\equiv 0 \pmod{p}$, 则存在整数 t 使得
$$x = x_0 + p^{a-1}t \tag{3}$$
是方程
$$f(x) \equiv 0 \pmod{p^a} \tag{4}$$
的解。
2. 若 $f'(x_0) \equiv 0 \pmod{p}$ 且 $f(x_0) \equiv 0 \pmod{p^a}$, 则对 $t = 0, 1, \dots, p-1$，由式 (3) 确定的 x 均为方程 (4) 的解。
3. 若 $f'(x_0) \equiv 0 \pmod{p}$ 且 $f(x_0) \not\equiv 0 \pmod{p^a}$, 则不能由式 (3) 构造方程 (4) 的解。

" 证明"

我们假设式 (3) 是方程 (4) 的解，即

$$f(x_0 + p^{a-1}t) \equiv 0 \pmod{p^a}$$

整理后可得

$$f(x_0) + p^{a-1}tf'(x_0) \equiv 0 \pmod{p^a}$$

于是

$$tf'(x_0) \equiv -\frac{f(x_0)}{p^{a-1}} \pmod{p} \tag{5}$$

1. 若 $f'(x_0) \not\equiv 0 \pmod{p}$，则关于 t 的方程 (5) 有唯一解 t_0，代入式 (3) 可验证其为方程 (4) 的解。
2. 若 $f'(x_0) \equiv 0 \pmod{p}$ 且 $f(x_0) \equiv 0 \pmod{p^a}$，则任意 t 均能使方程 (5) 成立，代入式 (3) 可验证其均为方程 (4) 的解。
3. 若 $f'(x_0) \equiv 0 \pmod{p}$ 且 $f(x_0) \not\equiv 0 \pmod{p^a}$，则方程 (5) 无解，从而不能由式 (3) 构造方程 (4) 的解。

进而我们有推论：

推论 1

对定理 1 的 p，a，$f(x)$，x_0，

1. 若 s 是方程 $f(x)\equiv 0 \pmod p$ 的解，且 $f'(a)\not\equiv 0 \pmod p$，则存在 $x_s\in\mathbf{Z}_{p^a}$，$x_s\equiv s \pmod p$ 使得 x_s 是方程 (4) 的解。
2. 若方程 $f(x)\equiv 0 \pmod p$ 与 $f'(a)\equiv 0 \pmod p$ 无公共解，则方程 (4) 和方程 $f(x)\equiv 0 \pmod p$ 的解数相同。

从而我们可以将素数幂模同余方程化归到素数模同余方程的情况。

素数模同余方程

以下令 $p\in\mathbf{P}$，整系数多项式 $f(x)=\sum_{i=0}^n a_ix^i$，其中 $p\nmid a_n$，$x\in\mathbf{Z}_p$.

定理 2

若方程

$$f(x)\equiv 0 \pmod p \tag{6}$$

有 k 个不同的解 $x_1,x_2,\dots,x_k$ $(k\le n)$，则：

$$f(x)\equiv g(x)\prod_{i=1}^k(x-x_i) \pmod p$$

其中 $\deg g=n-k$ 且 $[x^{n-k}]g(x)=a_n$.

" 证明"

对 k 应用数学归纳法。

- 当 $k=1$ 时，做多项式带余除法，有 $f(x)=(x-x_1)g(x)+r$，其中 $r\in\mathbf{Z}$.
  由 $f(x_1)\equiv 0 \pmod p$ 可知 $r\equiv 0 \pmod p$，从而 $f(x)\equiv(x-x_1)g(x) \pmod p$.
- 假设命题对 $k-1(k>1)$ 时的情况成立，现在设 $f(x)$ 有 k 个不同的解 $x_1,x_2,\dots,x_k$, 则 $f(x)\equiv(x-x_1)h(x)$ $\pmod p$, 进而有
  $$(\forall i=2,3,\dots,k),\quad 0\equiv f(x_i)\equiv(x_i-x_1)h(x_i) \pmod p$$
  从而 $h(x)$ 有 $k-1$ 个不同的解 $x_2,x_3,\dots,x_k$, 由归纳假设有
  $$h(x)\equiv g(x)\prod_{i=2}^k(x-x_i) \pmod p$$
  其中 $\deg g=n-k$ 且 $[x^{n-k}]g(x)=a_n$.
  因此命题得证。

推论 2

对素数 p，

- $(\forall x\in\mathbf{Z})$，$x^{p-1}-1\equiv\prod_{i=1}^{p-1}(x-i) \pmod p$
- （Wilson 定理）$(p-1)!\equiv -1 \pmod p$

定理 3（Lagrange 定理）

方程 (6) 至多有 n 个不同解。

" 证明"

假设 $f(x)$ 有 $n+1$ 个不同解 $x_1, x_2, \dots, x_{n+1}$，则由定理 2，对 $x_1, x_2, \dots, x_n$ 有

$$f(x) \equiv a_n \prod_{i=1}^{n} (x - x_i) \pmod p$$

令 $x = x_{n+1}$，则

$$0 \equiv f(x_{n+1}) \equiv a_n \prod_{i=1}^{n} (x_{n+1} - x_i) \pmod p$$

而右侧显然不是 p 的倍数，因此假设矛盾。

推论 3

若同余方程 $\sum_{i=0}^{n} b_i x^i \equiv 0 \pmod p$ 的解数大于 n，则

$$(\forall i = 0, 1, \dots, n), \quad p \mid b_i$$

定理 4

方程 (6) 若解的个数不为 p，则必存在满足 $\deg r < p$ 的整系数多项式 $r(x)$ 使得 $f(x) \equiv 0 \pmod p$ 和 $r(x) \equiv 0 \pmod p$ 的解集相同。

"证明"

不妨设 $n \geq p$，对 $f(x)$ 做多项式带余除法

$$f(x) = g(x)\left(x^p - x\right) + r(x)$$

其中 $\deg r < p$.

由 Fermat 小定理 知对任意整数 x 有 $x^p \equiv x \pmod p$，从而

- 若 $r(x) \equiv 0 \pmod p$，则由推论 2 可知 $f(x)$ 有 p 个不同的解。
- 若 $r(x) \not\equiv 0 \pmod p$，则由 $f(x) \equiv r(x) \pmod p$ 可知 $f(x)$ 和 $r(x)$ 的解集相同。

我们可以通过这个定理对同余方程降次。

定理 5

设 $n \leq p$，则方程

$$x^n + \sum_{i=0}^{n-1} a_i x^i \equiv 0 \pmod p \tag{7}$$

有 n 个解当且仅当存在整系数多项式 $q(x)$，$r(x)$ $(\deg r < n)$ 使得

$$x^p - x = f(x)q(x) + pr(x) \tag{8}$$

"证明"

- 必要性：由多项式除法知存在整系数多项式 $q(x)$，$r_1(x)$ $(\deg r_1 < n)$ 使得

$$x^p - x = f(x)q(x) + r_1(x)$$

若方程 (7) 有 n 个解，则 $r_1 \equiv 0 \pmod p$ 也有 n 个相同的解，进而由推论 3 可知存在整系数多项式 $r(x)$ 满足 $r_1(x) = pr(x)$，从而命题得证。

- 充分性：若式 (8) 成立，则由 Fermat 小定理 可知，对任意整数 x，

$$0 \equiv x^p - x \equiv f(x)q(x) \pmod p$$

即方程 $f(x)q(x) \equiv 0 \pmod p$ 有 p 个解。

设方程 (7) 的解数为 s，则由 Lagrange 定理可知 $s \le n$.

又由于 $\deg q = p-n$，则由 Lagrange 定理可知 $q(x) \equiv 0 \pmod p$ 的解数不超过 $p-n$，而方程 $f(x)q(x) \equiv 0 \pmod p$ 的解集是 $f(x) \equiv 0 \pmod p$ 解集和 $q(x) \equiv 0 \pmod p$ 解集的并集，故 $s+(p-n) \ge p$，有 $s \ge n$.

因此 $s = n$.

对于非首 1 多项式，由于 $\mathbf{Z}_p$ 是域，故可以将其化为首 1 多项式，从而适用该定理。

定理 6

设 $n \nmid p-1$，$p \nmid a$, 则方程

$$x^n \equiv a \pmod p \tag{9}$$

有解当且仅当

$$a^{\frac{p-1}{n}} \equiv 1 \pmod p$$

且若 (9) 有解，则解数为 n.

note

方程 (9) 解集的具体结构可参见 k 次剩余。

" 证明"

- 必要性：若方程 (9) 有解 x_0，则

$$a^{\frac{p-1}{n}} \equiv \left(x_0^n\right)^{\frac{p-1}{n}} \equiv 1 \pmod p$$

- 充分性：若 $a^{\frac{p-1}{n}} \equiv 1 \pmod p$，则

$$\begin{aligned} x^p - x &= x\left(x^{p-1}-1\right) \\ &= x\left(\left(x^n\right)^{\frac{p-1}{n}} - a^{\frac{p-1}{n}} + a^{\frac{p-1}{n}} - 1\right) \\ &= \left(x^n - a\right)P(x) + x\left(a^{\frac{p-1}{n}} - 1\right) \end{aligned}$$

其中 $P(x)$ 是某个整系数多项式，因此由定理 5 可知方程 (9) 有 n 个解。

高次同余方程（组）的求解方法

首先我们可以借助 中国剩余定理 将求解**同余方程组**转为求解**同余方程**，以及将求解模**合数** m 的同余方程转化为求解模**素数幂次**的同余方程。之后我们借助定理 1 将求解模**素数幂次**的同余方程转化为求解模**素数**的同余方程。

结合模素数同余方程的若干定理，我们只需考虑方程

$$x^n + \sum_{i=0}^{n-1} a_i x^i \equiv 0 \pmod p$$

的求法，其中 p 是素数，$n < p$.

我们可以通过将 x 代换为 $x - \dfrac{a_{n-1}}{n}$ 来消去 x^{n-1} 项，从而我们只需考虑方程

$$x^n + \sum_{i=0}^{n-2} a_i x^i \equiv 0 \pmod p \tag{10}$$

的求法，其中 p 是素数，$n < p$.

- 若 $n = 1$，则求法参见 线性同余方程。

- 若 $n=2$，则求法参见 二次剩余。
- 若方程 (10) 可化为
$$x^n \equiv a \pmod p$$
  则求法参见 k 次剩余。

参考资料

1. Congruence Equation -- from Wolfram MathWorld[1]
2. Lagrange's theorem (number theory) - Wikipedia[2]
3. 潘承洞，潘承彪。初等数论。
4. 冯克勤。初等数论及其应用。
5. 闵嗣鹤，严士健。初等数论。

参考资料与注释

[1] Congruence Equation – from Wolfram MathWorld

[2] Lagrange's theorem (number theory) - Wikipedia

9.12.19 二次剩余

定义

令整数 a，p 满足 $(a,p)=1$，若存在整数 x 使得
$$x^2 \equiv a \pmod p$$
则称 a 为模 p 的二次剩余，否则称 a 为模 p 的二次非剩余。

通俗一些，可以认为是求模意义下的**开平方**运算。对于更高次方的开方可参见 k 次剩余。

这里只讨论 p 为**奇素数**的求解方法。后文可能在模 p 显然的情况下简写成二次（非）剩余。

Euler 判别法

对奇素数 p 和满足 $(a,p)=1$ 的整数 a，
$$a^{\frac{p-1}{2}} \equiv \begin{cases} 1 \pmod p, & (\exists x \in \mathbf{Z}),\quad a \equiv x^2 \pmod p, \\ -1 \pmod p, & \text{otherwise.} \end{cases}$$

即对上述的 p 和 a，

1. a 是 p 的二次剩余当且仅当 $a^{\frac{p-1}{2}} \equiv 1 \pmod p$.
2. a 是 p 的二次非剩余当且仅当 $a^{\frac{p-1}{2}} \equiv -1 \pmod p$.

" 证明"

首先由 Fermat 小定理，有 $a^{p-1} \equiv 1 \pmod p$，故
$$\left(a^{\frac{p-1}{2}}+1\right)\left(a^{\frac{p-1}{2}}-1\right) \equiv 0 \pmod p$$
从而对任意满足 $(a,p)=1$ 的 a 均有 $a^{(p-1)/2} \equiv \pm 1 \pmod p$

另外由 p 是奇素数，我们有：

$$x^{p-1}-a^{\frac{p-1}{2}}=\left(x^{2}\right)^{\frac{p-1}{2}}-a^{\frac{p-1}{2}}=\left(x^{2}-a\right)P(x)$$

其中 $P(x)$ 是某个整系数多项式，进而：

$$\begin{aligned} x^{p}-x&=x\left(x^{p-1}-a^{\frac{p-1}{2}}\right)+x\left(a^{\frac{p-1}{2}}-1\right)\\ &=\left(x^{2}-a\right)xP(x)+\left(a^{\frac{p-1}{2}}-1\right)x \end{aligned}$$

由 同余方程的定理 5 可知，a 是模 p 的二次剩余当且仅当 $a^{(p-1)/2}\equiv 1 \pmod p$. 进而 a 是模 p 的非二次剩余当且仅当 $a^{(p-1)/2}\equiv -1 \pmod p$.

Legendre 符号

" 定义"

对奇素数 p 和整数 a，定义 Legendre 符号如下：

$$\left(\frac{a}{p}\right)=\begin{cases}0, & p\mid a,\\ 1, & (p\nmid a)\wedge((\exists x\in\mathbf{Z}),\quad a\equiv x^{2}\pmod p),\\ -1, & \text{otherwise}.\end{cases}$$

Legendre 符号可进一步推广为 Jacobi 符号[6]，Jacobi 符号可进一步推广为 Kronecker 符号[7]。

下表为部分 Legendre 符号的值（From Wikipedia[8]）

p \ a	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
3	1	-1	0	1	-1	0	1	-1	0	1	-1	0	1	-1	0	1	-1	0	1	-1	0	1	-1	0	1	-1	0	1	-1	0
5	1	-1	-1	1	0	1	-1	-1	1	0	1	-1	-1	1	0	1	-1	-1	1	0	1	-1	-1	1	0	1	-1	-1	1	0
7	1	1	-1	1	-1	-1	0	1	1	-1	1	-1	-1	0	1	1	-1	1	-1	-1	0	1	1	-1	1	-1	-1	0	1	1
11	1	-1	1	1	1	-1	-1	-1	1	-1	0	1	-1	1	1	1	-1	-1	-1	1	-1	0	1	-1	1	1	1	-1	-1	-1
13	1	-1	1	1	-1	-1	-1	-1	1	1	-1	1	0	1	-1	1	1	-1	-1	-1	-1	1	1	-1	1	0	1	-1	1	1
17	1	1	-1	1	-1	-1	-1	1	1	-1	-1	-1	1	-1	1	1	0	1	1	-1	1	-1	-1	-1	1	1	-1	-1	-1	1
19	1	-1	-1	1	1	1	1	-1	1	-1	1	-1	-1	-1	-1	1	1	-1	0	1	-1	-1	1	1	1	1	-1	1	-1	1
23	1	1	1	1	-1	1	-1	1	1	-1	-1	1	1	-1	-1	1	-1	1	-1	-1	-1	-1	0	1	1	1	1	-1	1	-1
29	1	-1	-1	1	1	1	1	-1	1	-1	-1	-1	1	-1	-1	1	-1	-1	-1	1	-1	1	1	1	1	-1	-1	1	0	1
31	1	1	-1	1	1	-1	1	1	1	1	-1	-1	-1	1	-1	1	-1	1	1	1	-1	-1	-1	-1	1	-1	-1	1	-1	-1
37	1	-1	1	1	-1	-1	1	-1	1	1	1	1	-1	-1	-1	1	-1	-1	-1	-1	1	-1	-1	-1	1	1	1	1	-1	1
41	1	1	-1	1	1	-1	-1	1	1	1	-1	-1	-1	-1	-1	1	-1	1	-1	1	1	-1	1	-1	1	-1	-1	-1	-1	-1
43	1	-1	-1	1	-1	1	-1	-1	1	1	1	-1	1	1	1	1	1	-1	-1	-1	1	-1	1	1	1	-1	-1	-1	-1	-1
47	1	1	1	1	-1	1	1	1	1	-1	-1	1	-1	1	-1	1	1	1	-1	-1	1	-1	-1	1	1	-1	1	1	-1	-1
53	1	-1	-1	1	-1	1	1	-1	1	1	1	-1	1	-1	1	1	1	-1	-1	-1	-1	-1	-1	1	1	-1	-1	1	1	-1
59	1	-1	1	1	1	-1	1	-1	1	-1	-1	1	-1	-1	1	1	1	-1	1	1	1	1	-1	-1	1	1	1	1	1	-1
61	1	-1	1	1	1	-1	-1	-1	1	-1	-1	1	1	1	1	1	-1	-1	1	1	-1	1	-1	-1	1	-1	1	-1	-1	-1
67	1	-1	-1	1	-1	1	-1	-1	1	1	-1	-1	-1	1	1	1	1	-1	1	-1	1	1	1	1	1	1	-1	-1	1	-1
71	1	1	1	1	1	1	-1	1	1	1	-1	1	-1	-1	1	1	-1	1	1	1	-1	-1	-1	1	1	-1	1	-1	1	1
73	1	1	1	1	-1	1	-1	1	1	-1	-1	1	-1	-1	-1	1	-1	1	1	-1	-1	-1	1	1	1	-1	1	-1	-1	-1
79	1	1	-1	1	1	-1	-1	1	1	1	1	-1	1	-1	-1	1	-1	1	1	1	1	1	1	-1	1	1	-1	-1	-1	-1
83	1	-1	1	1	-1	-1	1	-1	1	1	1	1	-1	-1	-1	1	1	-1	-1	-1	1	-1	1	-1	1	1	1	1	1	1
89	1	1	-1	1	1	-1	-1	1	1	1	1	-1	-1	-1	-1	1	1	1	-1	1	1	1	-1	-1	1	-1	-1	-1	-1	-1
97	1	1	1	1	-1	1	-1	1	1	-1	1	1	-1	-1	-1	1	-1	1	-1	-1	-1	1	-1	1	1	-1	1	-1	-1	-1
101	1	-1	-1	1	1	1	-1	-1	1	-1	-1	-1	1	1	-1	1	1	-1	1	1	1	1	1	1	1	-1	-1	-1	-1	1
103	1	1	-1	1	-1	-1	1	1	1	-1	-1	-1	1	1	1	1	1	1	1	-1	-1	-1	1	-1	1	1	-1	1	1	1
107	1	-1	1	1	-1	-1	-1	-1	1	1	1	1	1	1	-1	1	-1	-1	1	-1	-1	-1	1	-1	1	-1	1	-1	1	1
109	1	-1	1	1	1	-1	1	-1	1	-1	-1	1	-1	-1	1	1	-1	-1	-1	1	1	1	-1	-1	1	1	1	1	1	-1
113	1	1	-1	1	-1	-1	1	1	1	-1	1	-1	1	1	1	1	-1	1	-1	-1	-1	1	-1	-1	1	1	-1	1	-1	1
127	1	1	-1	1	-1	-1	-1	1	1	-1	1	-1	1	-1	1	1	1	1	1	-1	1	1	-1	-1	1	1	-1	-1	-1	1

图 9.11

性质

1. 对任意整数 a，

$$a^{\frac{p-1}{2}} \equiv \left(\frac{a}{p}\right) \pmod p$$

进一步，我们有推论：

- $$\left(\frac{1}{p}\right) = 1$$

- $$\begin{aligned}\left(\frac{-1}{p}\right) &= (-1)^{\frac{p-1}{2}} \\ &= \begin{cases} 1, & p \equiv 1 \pmod 4, \\ -1, & p \equiv 3 \pmod 4. \end{cases}\end{aligned}$$

2. $a_1 \equiv a_2 \pmod p \iff \left(\frac{a_1}{p}\right) = \left(\frac{a_2}{p}\right)$

3. （完全积性）对任意整数 a_1, a_2，

$$\left(\frac{a_1 a_2}{p}\right) = \left(\frac{a_1}{p}\right)\left(\frac{a_2}{p}\right)$$

我们有推论：对整数 a, b，$p \nmid b$ 有

$$\left(\frac{ab^2}{p}\right) = \left(\frac{a}{p}\right)$$

4. $$\begin{aligned}\left(\frac{2}{p}\right) &= (-1)^{\frac{p^2-1}{8}} \\ &= \begin{cases} 1, & p \equiv \pm 1 \pmod 8 \\ -1, & p \equiv \pm 3 \pmod 8 \end{cases}\end{aligned}$$

" 证明"

1. 由 Legendre 符号的定义和 Euler 判别法易得。

2. 注意到

$$a_1 \equiv a_2 \pmod p \iff \left(\frac{a_1}{p}\right) \equiv \left(\frac{a_2}{p}\right) \pmod p$$

而 $\left|\left(\frac{a_1}{p}\right) - \left(\frac{a_2}{p}\right)\right| \leq 2$ 且 $p > 2$, 故：

$$a_1 \equiv a_2 \pmod p \iff \left(\frac{a_1}{p}\right) = \left(\frac{a_2}{p}\right)$$

3. 由 1 得

$$\left(\frac{a_1 a_2}{p}\right) \equiv a_1^{\frac{p-1}{2}} a_2^{\frac{p-1}{2}} \equiv \left(\frac{a_1}{p}\right)\left(\frac{a_2}{p}\right) \pmod p$$

而 $\left|\left(\frac{a_1 a_2}{p}\right) - \left(\frac{a_1}{p}\right)\left(\frac{a_2}{p}\right)\right| \leq 2$ 且 $p > 2$, 故：

$$\left(\frac{a_1 a_2}{p}\right) = \left(\frac{a_1}{p}\right)\left(\frac{a_2}{p}\right)$$

4. 参见二次互反律

二次互反律

" 二次互反律"

设 p，q 是两个不同的奇素数，则

$$\left(\frac{p}{q}\right)\left(\frac{q}{p}\right)=(-1)^{\frac{p-1}{2}\frac{q-1}{2}}$$

证明方式很多，读者感兴趣的话可参考[5]。一种证明方式是基于如下引理（Gauss 引理）：

"Gauss 引理"

设 $(n,p)=1$, 对整数 k $\left(1\le k\le \frac{p-1}{2}\right)$，令 r_k 为 nk 模 p 的最小非负剩余，设 m 为所有 r_k 中大于 $\frac{p}{2}$ 的个数，则

$$\left(\frac{n}{p}\right)=(-1)^m$$

这个引理可以证明如下有用的结论：

" 结论"

对奇素数 p，

$$\begin{aligned}\left(\frac{2}{p}\right)&=(-1)^{\frac{p^2-1}{8}}\\&=\begin{cases}1, & p\equiv \pm 1 \pmod 8\\ -1, & p\equiv \pm 3 \pmod 8\end{cases}\end{aligned}$$

二次互反律不仅能用于判断数 n 是否是模 p 的二次剩余，还能用于确定使数 n 为二次剩余的模数的结构。

二次剩余的数量

对于奇素数 p，模 p 意义下二次剩余和二次非剩余均有 $\frac{p-1}{2}$ 个。

" 证明"

根据 Euler 判别法，考虑 $a^{\frac{p-1}{2}}\equiv 1 \pmod p$.

注意到 $\frac{p-1}{2}\mid(p-1)$，由 同余方程的定理 6 可知 $a^{\frac{p-1}{2}}\equiv 1 \pmod p$ 有 $\frac{p-1}{2}$ 个解。所以模 p 意义下二次剩余和二次非剩余均有 $\frac{p-1}{2}$ 个。

相关算法

特殊情况时的算法

对于同余方程 $x^2\equiv a \pmod p$，其中 p 为奇素数且 a 为二次剩余在 $p \bmod 4=3$ 时有更简单的解法，考虑

$$\begin{aligned}\left(a^{(p+1)/4}\right)^2&\equiv a^{(p+1)/2} && \pmod p\\&\equiv x^{p+1} && \pmod p\\&\equiv \left(x^2\right)\left(x^{p-1}\right) && \pmod p\\&\equiv x^2 && \pmod p \quad (\because\text{Fermat's little theorem})\end{aligned}$$

那么 $a^{(p+1)/4} \bmod p$ 为一个解。

Atkin 算法

仍然考虑上述同余方程，此时 $p \bmod 8=5$，那么令 $b\equiv(2a)^{(p-5)/8} \pmod p$ 和 $\mathrm{i}\equiv 2ab^2 \pmod p$ 那么此时 $\mathrm{i}^2\equiv -1 \pmod p$ 且 $ab(\mathrm{i}-1) \bmod p$ 为一个解。

" 证明"

$$\begin{aligned}
\mathrm{i}^2 &\equiv \left(2ab^2\right)^2 && \pmod p \\
&\equiv \left(2a\cdot(2a)^{(p-5)/4}\right)^2 && \pmod p \\
&\equiv \left((2a)^{(p-1)/4}\right)^2 && \pmod p \\
&\equiv (2a)^{\frac{p-1}{2}} && \pmod p \\
&\equiv -1 && \pmod p
\end{aligned}$$

那么

$$\begin{aligned}
(ab(\mathrm{i}-1))^2 &\equiv a^2\cdot(2a)^{(p-5)/4}\cdot(-2\mathrm{i}) && \pmod p \\
&\equiv a\cdot(-\mathrm{i})\cdot(2a)^{(p-1)/4} && \pmod p \\
&\equiv a && \pmod p
\end{aligned}$$

Cipolla 算法

Cipolla 算法用于求解同余方程 $x^2\equiv a \pmod p$，其中 p 为奇素数且 a 为二次剩余。

算法可描述为找到 r 满足 r^2-a 为二次非剩余，$(r-x)^{(p+1)/2} \bmod (x^2-(r^2-a))$ 为一个解。

在复数域 $\mathbf{C}$ 中，考虑令 $x^2+1\in\mathbf{R}[x]$ 和实系数多项式的集合 $\mathbf{R}[x]$ 对 x^2+1 取模后的集合记作 $\mathbf{R}[x]/(x^2+1)$，那么集合中的元素都可以表示为 a_0+a_1x 的形式，其中 $a_0,a_1\in\mathbf{R}$，又因为 $x^2\equiv -1 \pmod{(x^2+1)}$，考虑多项式的运算可以发现 $\mathbf{R}[x]/(x^2+1)$ 中元素的运算与 $\mathbf{C}$ 中一致。

后文考虑对于系数属于有限域 $\mathbb{F}_p$ 的多项式 $\mathbb{F}_p[x]$ 和对 $x^2-(r^2-a)\in\mathbb{F}_p[x]$ 取模后的集合 $\mathbb{F}_p[x]/(x^2-(r^2-a))$ 中的运算。

选择 r：

若 $a\equiv 0 \pmod p$ 那么 r^2-a 为二次剩余，此时解显然为 $x\equiv 0 \pmod p$。所以假设 $a\not\equiv 0 \pmod p$。使得 r^2-a 为非零二次剩余的选择有 $\dfrac{p-3}{2}$ 个，而使得 $r^2\equiv a \pmod p$ 的选择恰有两个，那么有 $\dfrac{p-1}{2}$ 种选择可以使得 r^2-a 为二次非剩余，使用随机方法平均约两次可得 r.

" 证明"

令 $f(x)=x^2-(r^2-a)\in\mathbb{F}_p[x]$ 和 $a_0+a_1x=(r-x)^{(p+1)/2} \bmod (x^2-(r^2-a))$ 那么有 $a_0^2\equiv a \pmod p$ 且 $a_1\equiv 0 \pmod p$.

$$\begin{aligned}
x^p &\equiv x(x^2)^{\frac{p-1}{2}} && \pmod{f(x)} \\
&\equiv x(r^2-a)^{\frac{p-1}{2}} && \pmod{f(x)} \quad (\because x^2\equiv r^2-a \pmod{f(x)}) \\
&\equiv -x && \pmod{f(x)} \quad (\because r^2-a \text{ is quadratic non-residue})
\end{aligned}$$

又因为二项式定理

$$\begin{aligned}
(a+b)^p &= \sum_{i=0}^{p}\binom{p}{i}a^ib^{p-i} \\
&= \sum_{i=0}^{p}\frac{p!}{i!(p-i)!}a^ib^{p-i} \\
&\equiv a^p+b^p \pmod p
\end{aligned}$$

可以发现只有当 $i=0$ 和 $i=p$ 时由于没有因子 p 不会因为模 p 被消去，其他的项都因为有 p 因子被消去了。所以

$$\begin{aligned}
(r-x)^p &\equiv r^p-x^p && \pmod{f(x)} \\
&\equiv r+x && \pmod{f(x)}
\end{aligned}$$

所以

$$\begin{aligned}(a_0+a_1x)^2 &= a_0^2+2a_0a_1x+a_1^2x^2\\ &\equiv (r-x)^{p+1} && \pmod{f(x)}\\ &\equiv (r-x)^p(r-x) && \pmod{f(x)}\\ &\equiv (r+x)(r-x) && \pmod{f(x)}\\ &\equiv r^2-x^2 && \pmod{f(x)}\\ &\equiv a && \pmod{f(x)}\end{aligned}$$

若 $a_1\not\equiv 0 \pmod p$ 且

$$\begin{aligned}(a_0+a_1x)^2 &= a_0^2+2a_0a_1x+a_1^2x^2\\ &\equiv a_0^2+2a_0a_1x+a_1^2(r^2-a) \pmod{f(x)}\end{aligned}$$

所以 x 的系数必须为零即 $a_0\equiv 0 \pmod p$ 此时考虑 Legendre 符号为完全积性函数可知 $r^2-a\equiv a/a_1^2 \pmod p$ 显然为二次剩余，不符合定义。因此 $a_1\equiv 0 \pmod p$ 且 $a_0^2\equiv a \pmod p$.

Bostan–Mori 算法

该算法基于 Cipolla 算法，我们将问题转换为 常系数齐次线性递推 再应用 Bostan–Mori 算法。考虑另一种常见的 Cipolla 算法的描述为 $b=x^{(p+1)/2} \bmod (x^2-tx+a)$ 为满足 $b^2\equiv a \pmod p$ 的一个解[3]，其中 $x^2-tx+a\in\mathbb{F}_p[x]$ 为不可约多项式。选取 t 同样使用随机。证明过程略。参考文献[4] 中的算法我们可以发现问题可转化为求解形式幂级数的乘法逆元的某一项系数：

$$b=\left[x^{(p+1)/2}\right]\frac{1}{1-tx+ax^2}$$

且

$$[x^n]\frac{k_0+k_1x}{1+k_2x+k_3x^2}=\begin{cases}\left[x^{(n-1)/2}\right]\dfrac{k_1-k_0k_2+k_1k_3x}{1+(2k_3-k_2^2)x+k_3^2x^2}, & \text{if } n \bmod 2=1\\ \left[x^{n/2}\right]\dfrac{k_0+(k_0k_3-k_1k_2)x}{1+(2k_3-k_2^2)x+k_3^2x^2}, & \text{else if } n\neq 0\end{cases}$$

而 $n=0$ 时显然有 $[x^0]\dfrac{k_0+k_1x}{1+k_2x+k_3x^2}=k_0$，该算法乘法次数相较于 Cipolla 算法更少，其他相关乘法次数较少的算法可见[2]。

Legendre 算法

对于同余方程 $x^2\equiv a \pmod p$，其中 p 为奇素数且 a 为二次剩余。Legendre 算法可描述为找到 r 满足 r^2-a 为二次非剩余，令 $a_0+a_1x=(r-x)^{\frac{p-1}{2}} \bmod (x^2-a)$，那么 $a_0\equiv 0 \pmod p$ 且 $a_1^{-2}\equiv a \pmod p$.

" 证明"

考虑选择一个 b 满足 $b^2\equiv a \pmod p$，那么 $(r-b)(r+b)=r^2-a$ 为二次非剩余，所以

$$(r-b)^{\frac{p-1}{2}}(r+b)^{\frac{p-1}{2}}\equiv -1 \pmod p$$

存在环态射

$$\begin{aligned}\phi:\mathbb{F}_p[x]/(x^2-a) &\to \mathbb{F}_p\times\mathbb{F}_p\\ x &\mapsto (b,-b)\end{aligned}$$

那么

$$\begin{aligned}(a_0+a_1b,a_0-a_1b) &= \phi(a_0+a_1x)\\ &= \phi(r-x)^{\frac{p-1}{2}}\\ &= \left((r-b)^{\frac{p-1}{2}},(r+b)^{\frac{p-1}{2}}\right)\\ &= (\pm 1,\mp 1)\end{aligned}$$

所以 $2a_0 = (\pm 1) + (\mp 1) = 0$ 而 $2a_1 b = (\pm 1) - (\mp 1) = \pm 2$.

Tonelli–Shanks 算法

Tonelli–Shanks 算法是基于离散对数求解同余方程 $x^2 \equiv a \pmod p$ 的算法[1]，其中 p 为奇素数且 a 为模 p 的二次剩余。

令 $p - 1 = 2^n \cdot m$ 其中 m 为奇数。仍然使用随机方法寻找 $r \in \mathbb{F}_p$ 满足 r 为二次非剩余。令 $g \equiv r^m \pmod p$ 且 $b \equiv a^{(m-1)/2} \pmod p$，那么存在整数 $e \in \{0, 1, 2, \dots, 2^n - 1\}$ 满足 $ab^2 \equiv g^e \pmod p$。若 a 为二次剩余，那么 e 为偶数且 $\left(abg^{-e/2}\right)^2 \equiv a \pmod p$.

" 证明"

$$
\begin{aligned}
g^{2^n} &\equiv r^{2^n \cdot m} && \pmod p \\
&\equiv r^{p-1} && \pmod p \\
&\equiv 1 && \pmod p
\end{aligned}
$$

而

$$
\begin{aligned}
g^{2^{n-1}} &\equiv r^{2^{n-1} \cdot m} && \pmod p \\
&\equiv r^{\frac{p-1}{2}} && \pmod p \\
&\equiv -1 && \pmod p
\end{aligned}
$$

所以 g 的阶为 2^n，又因为 $ab^2 \equiv a^m \pmod p$ 是 $x^{2^n} \equiv 1 \pmod p$ 的解，所以 a^m 是 g 的幂次，记 $a^m \equiv g^e \pmod p$.

若 a 是二次剩余，那么

$$
\begin{aligned}
g^{2^{n-1} \cdot e} &\equiv (-1)^e && \pmod p \\
&\equiv a^{2^{n-1} \cdot m} && \pmod p \\
&\equiv a^{\frac{p-1}{2}} && \pmod p \\
&\equiv 1 && \pmod p
\end{aligned}
$$

所以 e 为偶数，而

$$
\begin{aligned}
\left(abg^{-e/2}\right)^2 &\equiv a^2 b^2 g^{-e} && \pmod p \\
&\equiv a^{m+1} g^{-e} && \pmod p \\
&\equiv a && \pmod p
\end{aligned}
$$

剩下的问题是如何计算 e，Tonelli 和 Shanks 提出一次确定 e 的一个比特。令 e 在二进制下表示为 $e = e_0 + 2e_1 + 4e_2 + \cdots$ 其中 $e_k \in \{0, 1\}$.

因为 a 是二次剩余，所以开始时 $e_0 = 0$，然后计算 e_1 然后 e_2 等等，由以下公式给出

$$
\left(g^e g^{-(e \bmod 2^k)}\right)^{2^{n-1-k}} \equiv g^{2^{n-1} \cdot e_k} \equiv \begin{cases} 1 \pmod p, & \text{if } e_k = 0 \\ -1 \pmod p, & \text{else if } e_k = 1 \end{cases}
$$

正确性显然。

习题

- 洛谷 P5491【模板】二次剩余[9]
- 「Timus 1132」Square Root[10]

参考资料与注释

1. Quadratic residue - Wikipedia[11]

2. Euler's criterion - Wikipedia[12]

[1] Daniel. J. Bernstein. Faster Square Roots in Annoying Finite Fields.

[2] S. Müller, On the computation of square roots in finite fields, Design, Codes and Cryptography, Vol.31, pp. 301-312, 2004

[3] A. Menezes, P. van Oorschot and S. Vanstone. Handbook of Applied Cryptography, 1996.

[4] Alin Bostan, Ryuhei Mori.A Simple and Fast Algorithm for Computing the N-th Term of a Linearly Recurrent Sequence.

[5] Quadratic reciprocity - Wikipedia

[6] Jacobi 符号

[7] Kronecker 符号

[8] Wikipedia

[9] 洛谷 P5491【模板】二次剩余

[10] 「Timus 1132」Square Root

[11] Quadratic residue - Wikipedia

[12] Euler's criterion - Wikipedia

9.12.20 原根

前置知识

前置：费马小定理，欧拉定理，拉格朗日定理。

这部分知识与抽象代数相关。如果想要进一步了解文中的「阶」、「原根」名字来源，可以参考群论部分。

阶

" 定义"

由欧拉定理可知，对 $a \in \mathbf{Z}$，$m \in \mathbf{N}^*$，若 $(a,m)=1$，则 $a^{\varphi(m)} \equiv 1 \pmod m$.

因此满足同余式 $a^n \equiv 1 \pmod m$ 的最小正整数 n 存在，这个 n 称作 a 模 m 的阶，记作 $\delta_m(a)$ 或 $\operatorname{ord}_m(a)$.

" 注"

在抽象代数中，这里的「阶」就是模 m 缩剩余系关于乘法形成的群中，元素 a 的阶。记号 δ 表示阶也只用于这个特殊的群。

下面的诸多性质可以直接扩展到抽象代数中阶的性质。

另外还有「半阶」的概念，在数论中会出现 δ^- 记号，表示同余式 $a^n \equiv -1 \pmod m$ 的最小正整数。半阶不是群论中的概念。阶一定存在，半阶不一定存在。

性质

性质 1

$a, a^2, \cdots, a^{\delta_m(a)}$ 模 m 两两不同余。

" 证明"

考虑反证，假设存在两个数 $i \neq j$，且 $a^i \equiv a^j \pmod{m}$，则有 $a^{|i-j|} \equiv 1 \pmod{p}$.

但是显然的有：$0 < |i-j| < \delta_m(a)$，这与阶的最小性矛盾，故原命题成立。

性质 2

若 $a^n \equiv 1 \pmod{m}$，则 $\delta_m(a) \mid n$.

" 证明"

对 n 除以 $\delta_m(a)$ 作带余除法，设 $n = \delta_m(a)q + r, 0 \leq r < \delta_m(a)$.

若 $r > 0$，则

$$a^r \equiv a^r(a^{\delta_m(a)})^q \equiv a^n \equiv 1 \pmod{m}$$

这与 $\delta_m(a)$ 的最小性矛盾。故 $r = 0$，即 $\delta_m(a) \mid n$.

据此还可推出：

若 $a^p \equiv a^q \pmod{m}$，则有 $p \equiv q \pmod{\delta_m(a)}$.

还有两个与四则运算有关的重要性质。

性质 3

设 $m \in \mathbf{N}^*$，$a, b \in \mathbf{Z}$，$(a, m) = (b, m) = 1$，则

$$\delta_m(ab) = \delta_m(a)\delta_m(b)$$

的充分必要条件是

$$(\delta_m(a), \delta_m(b)) = 1$$

" 证明"

- 必要性：由 $a^{\delta_m(a)} \equiv 1 \pmod{m}$ 及 $b^{\delta_m(b)} \equiv 1 \pmod{m}$，可知

$$(ab)^{[\delta_m(a), \delta_m(b)]} \equiv 1 \pmod{m}$$

由前面所述阶的性质，有

$$\delta_m(ab) \mid [\delta_m(a), \delta_m(b)]$$

又由于 $\delta_m(ab) = \delta_m(a)\delta_m(b)$，故

$$\delta_m(a)\delta_m(b) \mid [\delta_m(a), \delta_m(b)]$$

即 $(\delta_m(a), \delta_m(b)) = 1$.

- 充分性：由 $(ab)^{\delta_m(ab)} \equiv 1 \pmod{m}$ 可知

$$1 \equiv (ab)^{\delta_m(ab)\delta_m(b)} \equiv a^{\delta_m(ab)\delta_m(b)} \pmod{m}$$

故 $\delta_m(a) \mid \delta_m(ab)\delta_m(b)$. 结合 $(\delta_m(a), \delta_m(b)) = 1$ 即得

$$\delta_m(a) \mid \delta_m(ab)$$

对称地，同理可得

$$\delta_m(b) \mid \delta_m(ab)$$

所以

$$\delta_m(a)\delta_m(b) \mid \delta_m(ab)$$

另一方面，有

$$(ab)^{\delta_m(a)\delta_m(b)} \equiv (a^{\delta_m(a)})^{\delta_m(b)} \times (b^{\delta_m(b)})^{\delta_m(a)} \equiv 1 \pmod m$$

故

$$\delta_m(ab) \mid \delta_m(a)\delta_m(b)$$

综合以上两点即得

$$\delta_m(ab) = \delta_m(a)\delta_m(b)$$

性质 4

设 $k \in \mathbf{N}$，$m \in \mathbf{N}^*$，$a \in \mathbf{Z}$，$(a,m)=1$，则

$$\delta_m(a^k) = \frac{\delta_m(a)}{(\delta_m(a),k)}$$

" 证明"

注意到：

$$\begin{aligned} a^{k\delta_m(a^k)} &= (a^k)^{\delta_m(a^k)} \equiv 1 \pmod m \\ &\implies \delta_m(a) \mid k\delta_m(a^k) \\ &\implies \frac{\delta_m(a)}{(\delta_m(a),k)} \mid \delta_m(a^k) \end{aligned}$$

另一方面，由 $a^{\delta_m(a)} \equiv 1 \pmod m$，可知：

$$(a^k)^{\frac{\delta_m(a)}{(\delta_m(a),k)}} = (a^{\delta_m(a)})^{\frac{k}{(\delta_m(a),k)}} \equiv 1 \pmod m$$

故：

$$\delta_m(a^k) \mid \frac{\delta_m(a)}{(\delta_m(a),k)}$$

综合以上两点，得：

$$\delta_m(a^k) = \frac{\delta_m(a)}{(\delta_m(a),k)}$$

原根

" 定义"

设 $m \in \mathbf{N}^*$，$g \in \mathbf{Z}$. 若 $(g,m)=1$，且 $\delta_m(g)=\varphi(m)$，则称 g 为模 m 的原根。

即 g 满足 $\delta_m(g) = |\mathbf{Z}_m^*| = \varphi(m)$. 当 m 是质数时，我们有 $g^i \bmod m, 0 < i < m$ 的结果互不相同。

" 注"

在抽象代数中，原根就是循环群的生成元。这个概念只在模 m 缩剩余系关于乘法形成的群中有「原根」这个名字，在一般的循环群中都称作「生成元」。

并非每个模 m 缩剩余系关于乘法形成的群都是循环群，存在原根就表明它同构于循环群，如果不存在原根就表明不同构。

原根判定定理

设 $m \geqslant 3, (g,m)=1$，则 g 是模 m 的原根的充要条件是，对于 $\varphi(m)$ 的每个素因数 p，都有 $g^{\frac{\varphi(m)}{p}} \not\equiv 1 \pmod m$.

" 证明"

必要性显然，下面用反证法证明充分性。

当对于 $\varphi(m)$ 的每个素因数 p，都有 $g^{\frac{\varphi(m)}{p}} \not\equiv 1 \pmod m$ 成立时，我们假设存在一个 g，其不是模 m 的原根。

因为 g 不是 m 的原根，则存在一个 $t < \varphi(m)$ 使得 $g^t \equiv 1 \pmod m$.

由 裴蜀定理 得，一定存在一组 k, x 满足 $kt = x\varphi(m) + (t, \varphi(m))$.

又由 欧拉定理 得 $g^{\varphi(m)} \equiv 1 \pmod m$，故有：

$$1 \equiv g^{kt} \equiv g^{x\varphi(m)+(t,\varphi(m))} \equiv g^{(t,\varphi(m))} \pmod m$$

由于 $(t,\varphi(m)) \mid \varphi(m)$ 且 $(t,\varphi(m)) \leqslant t < \varphi(m)$.

故存在 $\varphi(m)$ 的素因数 p 使得 $(t,\varphi(m)) \mid \frac{\varphi(m)}{p}$.

则 $g^{\frac{\varphi(m)}{p}} \equiv g^{(t,\varphi(m))} \equiv 1 \pmod m$，与条件矛盾。

故假设不成立，原命题成立。

原根个数

若一个数 m 有原根，则它原根的个数为 $\varphi(\varphi(m))$.

" 证明"

若 m 有原根 g，则：

$$\delta_m(g^k) = \frac{\delta_m(g)}{(\delta_m(g), k)} = \frac{\varphi(m)}{(\varphi(m), k)}$$

所以若 $(k, \varphi(m)) = 1$，则有：$\delta_m(g^k) = \varphi(m)$，即 g^k 也是模 m 的原根。

而满足 $(\varphi(m), k) = 1$ 且 $1 \le k \le \varphi(m)$ 的 k 有 $\varphi(\varphi(m))$ 个。所以原根就有 $\varphi(\varphi(m))$ 个。

原根存在定理

" 原根存在定理"

一个数 m 存在原根当且仅当 $m = 2, 4, p^\alpha, 2p^\alpha$，其中 p 为奇素数，$\alpha \in \mathbf{N}^*$.

我们来证明它，分成 $m = 2, 4$、$m = p^\alpha$、$m = 2p^\alpha$ 与 $m \neq 2, 4, p, p^\alpha$，四个部分。

1. $m = 2, 4$，原根显然存在。
2. $m = p^\alpha$，其中 p 为奇素数，$\alpha \in \mathbf{N}^*$.

" 定理 1"

对于奇素数 p，p 有原根。

" 证明"

先证一个引理：

设 a 与 b 是与 p 互素的两个整数，则存在 $c \in \mathbf{Z}$ 使得 $\delta_p(c) = [\delta_p(a), \delta_p(b)]$.

" **证明**"

我们先将 $\delta_m(a),\delta_m(b)$ 表示成质因数分解的形式：

$$\left(\delta_m(a)=\prod_{i=1}^{k}p_i^{\alpha_i},\delta_m(b)=\prod_{i=1}^{k}p_i^{\beta_i}\right)$$

接着将它们表示成如下形式：

$$\delta_m(a)=XY,\delta_m(b)=ZW$$

其中：

- $Y=\prod_{i=1}^{k}p_i^{[\alpha_i>\beta_i]\alpha_i}$
- $X=\dfrac{\delta_m(a)}{Y}$
- $W=\prod_{i=1}^{k}p_i^{[\alpha_i\le\beta_i]\beta_i}$
- $Z=\dfrac{\delta_m(b)}{W}$

则由阶的性质 4，可得：

$$\begin{aligned}\delta_m\left(a^X\right)&=\frac{\delta_m(a)}{(\delta_m(a),X)}\\&=\frac{XY}{X}\\&=Y\end{aligned}$$

同理：

$$\delta_m\left(b^Z\right)=W$$

又因为显然有 $(Y,W)=1$，$YW=\left[\delta_p(a),\delta_p(b)\right]$，则再由阶的性质 3，可得：

$$\begin{aligned}\delta_m\left(a^Xb^Z\right)&=\delta_m\left(a^X\right)\delta_m\left(b^Z\right)\\&=YW\\&=\left[\delta_p(a),\delta_p(b)\right]\end{aligned}$$

于是令 $c=a^Xb^Z$ 则原命题得证。

回到原命题，对 $1\sim(p-1)$ 依次两两使用引理，可知存在 $g\in\mathbf{Z}$ 使得

$$\delta_p(g)=\left[\delta_p(1),\delta_p(2),\cdots,\delta_p(p-1)\right]$$

这表明 $\delta_p(j)\mid\delta_p(g)(j=1,2,\cdots,p-1)$，所以 $j=1,2,\cdots,p-1$ 都是同余方程

$$x^{\delta_p(g)}\equiv 1\pmod p$$

的根。由拉格朗日定理，可知方程的次数 $\delta_p(g)\ge p-1$.

又由费马小定理，易知 $\delta_p(g)\le p-1$，故 $\delta_p(g)=p-1=\varphi(p)$.

综上可知 g 为模 p 的原根。

" **定理** 2"

对于奇素数 p，$\alpha\in\mathbf{N}^*$，p^α 有原根。

" **证明**"

一个基本的想法是将模 p 的原根平移。

先证明一个引理:

存在模 p 的原根 g，使得 $g^{p-1} \not\equiv 1 \pmod{p^2}$.

" **证明**"

事实上，任取模 p 的原根 g，若 g 不满足条件，我们认定 $g+p$ 满足条件。

易知 $g+p$ 也是模 p 的原根。

我们有

$$\begin{aligned}(g+p)^{p-1} &\equiv \binom{p-1}{0}g^{p-1}+\binom{p-1}{1}pg^{p-2} \pmod{p^2}\\ &\equiv g^{p-1}+p(p-1)g^{p-2} \pmod{p^2}\\ &\equiv 1-pg^{p-2} \pmod{p^2}\\ &\not\equiv 1 \pmod{p^2}\end{aligned}$$

回到原题，我们证明若 g 是一个满足引理条件的原根，则对任意 $\alpha \in \mathbf{N}^*$，g 是模 p^α 的原根。

首先，证明下面的结论：对任意 $\beta \in \mathbf{N}^*$，都可设

$$g^{\varphi(p^\beta)} = 1 + p^\beta k_\beta$$

这里 $p \nmid k_\beta$。事实上，$\beta = 1$ 时，由 g 的选取可知结论成立。现设上式对 β 时成立，则

$$\begin{aligned}g^{\varphi(p^{\beta+1})} &= \left(g^{\varphi(p^\beta)}\right)^p\\ &= \left(1+p^\beta k_\beta\right)^p\\ &\equiv 1+p^{\beta+1}k_\beta \pmod{p^{\beta+2}}\end{aligned}$$

结合 $p \nmid k_\beta$ 可知命题对 $\beta+1$ 成立。

所以命题对任意 $\beta \in \mathbf{N}^*$ 都成立。

其次，记 $\delta = \delta_{p^\alpha}(g)$，则由欧拉定理，可知 $\delta \mid p^{\alpha-1}(p-1)$.

而由 g 为模 p 的原根，及 $g^\delta \equiv 1 \pmod{p^\alpha}$.

所以可设 $\delta = p^{\beta-1}(p-1)$，这里 $1 \le \beta \le \alpha$.

现在利用之前的结论，可知:

$$g^{\varphi(p^\beta)} \not\equiv 1 \pmod{p^{\beta+1}} \implies g^\delta \not\equiv 1 \pmod{p^{\beta+1}}$$

结合 $g^\delta \equiv 1 \pmod{p^\alpha}$ 可知 $\beta \ge \alpha$.

综上可知，$\beta = \alpha$，即:

$$\delta_{p^\alpha}(g) = p^{\alpha-1}(p-1) = \varphi(p^\alpha)$$

从而，g 是模 p^α 的原根。

3. $m = 2p^\alpha$，其中 p 为奇素数，$\alpha \in \mathbf{N}^*$.

" **定理** 3"

对于奇素数 p，$\alpha \in \mathbf{N}^*$，$2p^\alpha$ 的原根存在。

" **证明**"

设 g 是模 p^α 的原根，则 $g+p^\alpha$ 也是模 p^α 的原根。

在 g 和 $g+p^\alpha$ 中有一个是奇数，设这个奇数是 G，则 $(G, 2p^\alpha) = 1$.

由欧拉定理，$\delta_{2p^\alpha}(G) \mid \varphi(2p^\alpha)$.

而 $G^{\delta_{2p^\alpha}(G)} \equiv 1 \pmod{2p^\alpha}$，故：

$$G^{\delta_{2p^\alpha}(G)} \equiv 1 \pmod{p^\alpha}$$

利用 G 为模 p^α 的原根可知 $\varphi(p^\alpha) \mid \delta_{2p^\alpha}(G)$.

结合 $\varphi(p^\alpha) = \varphi(2p^\alpha)$ 可知 G 为模 $2p^\alpha$ 的原根。

4. $m \neq 2, 4, p^\alpha, p^\alpha$，其中 p 为奇素数，$\alpha \in \mathbf{N}^*$.

" 定理 4"

对于 $m \neq 2, 4$，且不存在奇素数 p 及 $\alpha \in \mathbf{N}^*$ 使得 $m = p^\alpha, 2p^\alpha$，模 m 的原根不存在。

" 证明"

对于 $m = 2^\alpha$，$\alpha \in \mathbf{N}^*, \alpha \geq 3$，则对任意奇数 $a = 2k+1$ 均有：

$$\begin{aligned}
a^{2^{\alpha-2}} &= (2k+1)^{2^{\alpha-2}} \\
&\equiv 1 + \binom{2^{\alpha-2}}{1}(2k) + \binom{2^{\alpha-2}}{2}(2k)^2 \pmod{2^\alpha} \\
&\equiv 1 + 2^{\alpha-1}k + 2^{\alpha-1}(2^{\alpha-2}-1)k^2 \pmod{2^\alpha} \\
&\equiv 1 + 2^{\alpha-1}(k + (2^{\alpha-2}-1)k^2) \pmod{2^\alpha} \\
&\equiv 1 \pmod{2^\alpha}
\end{aligned}$$

其中最后一步用到 k 与 $(2^{\alpha-2}-1)k^2$ 同奇偶，故其和为偶数。

若 m 不是 2 的幂，且 m 为符合题目条件的数，则可设 $m = rt$，这里 $2 < r < t$ 且 $(r, t) = 1$.

此时，若 $(a, m) = 1$，由欧拉定理可知：

$$a^{\varphi(r)} \equiv 1 \pmod{r}, \quad a^{\varphi(t)} \equiv 1 \pmod{t}$$

注意到 $n > 2$ 时，$\varphi(n)$ 为偶数，所以：

$$a^{\frac{1}{2}\varphi(r)\varphi(t)} \equiv 1 \pmod{rt}$$

进而：

$$\delta_m(a) \leq \frac{1}{2}\varphi(r)\varphi(t) = \frac{1}{2}\varphi(rt) = \frac{1}{2}\varphi(m) < \varphi(m)$$

由原根定义可得：模 m 的原根不存在。

综合以上 4 个定理，我们便给出了一个数存在原根的充要条件。

最小原根的范围估计

王元[2] 和 Burgess[1] 证明了素数 p 的最小原根 $g_p = O\left(p^{0.25+\epsilon}\right)$，其中 $\epsilon > 0$.

Fridlander[3] 和 Salié[4] 证明了素数 p 的最小原根 $g_p = \Omega(\log p)$.

这保证了我们暴力找一个数的最小原根，复杂度是可以接受的。

参考资料与注释

1. Primitive root modulo n - Wikipedia[5]

[1] BURGESS, David A. On character sums and primitive roots.*Proceedings of the London Mathematical Society*, 1962, 3.1: 179-192.

[2] Wang Y. On the least primitive root of a prime (in Chinese).*Acta Math Sinica*, 1959, 4: 432–441; English transl. in*Sci. Sinica*, 1961, 10: 1–14

[3] FRIDLENDER, V. R. On the least n-th power non-residue. In:*Dokl. Akad. Nauk SSSR*. 1949. p. 351-352.

[4] SALIÉ, Hans. Über den kleinsten positiven quadratischen Nichtrest nach einer Primzahl.*Mathematische Nachrichten*, 1949, 3.1: 7-8.

[5] Primitive root modulo n - Wikipedia

9.12.21 离散对数

定义

前置知识：阶与原根。

离散对数的定义方式和对数类似。取有原根的正整数模数 m，设其一个原根为 g. 对满足 $(a,m)=1$ 的整数 a，我们知道必存在唯一的整数 $0\leq k<\varphi(m)$ 使得

$$g^k\equiv a\pmod m$$

我们称这个 k 为以 g 为底，模 m 的离散对数，记作 $k=\operatorname{ind}_g a$，在不引起混淆的情况下可记作 $\operatorname{ind} a$.

显然 $\operatorname{ind}_g 1=0$，$\operatorname{ind}_g g=1$.

性质

离散对数的性质也和对数有诸多类似之处。

" 性质"

设 g 是模 m 的原根，$(a,m)=(b,m)=1$，则：

1. $\operatorname{ind}_g(ab)\equiv\operatorname{ind}_g a+\operatorname{ind}_g b\pmod{\varphi(m)}$
   进而 $(\forall n\in\mathbf{N})$，$\operatorname{ind}_g a^n\equiv n\operatorname{ind}_g a\pmod{\varphi(m)}$
2. 若 g_1 也是模 m 的原根，则 $\operatorname{ind}_g a\equiv\operatorname{ind}_{g_1} a\cdot\operatorname{ind}_g g_1\pmod{\varphi(m)}$
3. $a\equiv b\pmod m\iff\operatorname{ind}_g a=\operatorname{ind}_g b$

" 证明"

1. $g^{\operatorname{ind}_g(ab)}\equiv ab\equiv g^{\operatorname{ind}_g a}g^{\operatorname{ind}_g b}\equiv g^{\operatorname{ind}_g a+\operatorname{ind}_g b}\pmod m$
2. 令 $x=\operatorname{ind}_{g_1} a$，则 $a\equiv g_1^x\pmod m$. 又令 $y=\operatorname{ind}_g g_1$，则 $g_1\equiv g^y\pmod m$.
   故 $a\equiv g^{xy}\pmod m$，即 $\operatorname{ind}_g a\equiv xy\equiv\operatorname{ind}_{g_1} a\cdot\operatorname{ind}_g g_1\pmod{\varphi(m)}$
3. 注意到

$$\begin{aligned}\operatorname{ind}_g a=\operatorname{ind}_g b&\iff\operatorname{ind}_g a\equiv\operatorname{ind}_g b\pmod{\varphi(m)}\\&\iff g^{\operatorname{ind}_g a}\equiv g^{\operatorname{ind}_g b}\pmod m\\&\iff a\equiv b\pmod m\end{aligned}$$

大步小步算法

目前离散对数问题仍不存在多项式时间经典算法（离散对数问题的输入规模是输入数据的位数）。在密码学中，基于这一点人们设计了许多非对称加密算法，如 Ed25519[1]。

在算法竞赛中，BSGS（baby-step giant-step，大步小步算法）常用于求解离散对数问题。形式化地说，对 $a,b,m\in\mathbf{Z}^+$，该算法可以在 $O(\sqrt{m})$ 的时间内求解

$$a^x\equiv b\pmod m$$

其中 $a\perp m$。方程的解 x 满足 $0\leq x<m$.（注意 m 不一定是素数）

算法描述

令 $x = A\lceil\sqrt{m}\rceil - B$，其中 $0 \le A, B \le \lceil\sqrt{m}\rceil$，则有 $a^{A\lceil\sqrt{m}\rceil - B} \equiv b \pmod m$，稍加变换，则有 $a^{A\lceil\sqrt{m}\rceil} \equiv ba^B \pmod m$.

我们已知的是 a, b，所以我们可以先算出等式右边的 ba^B 的所有取值，枚举 B，用 `hash/map` 存下来，然后逐一计算 $a^{A\lceil\sqrt{m}\rceil}$，枚举 A，寻找是否有与之相等的 ba^B，从而我们可以得到所有的 x，$x = A\lceil\sqrt{m}\rceil - B$.

注意到 A, B 均小于 $\lceil\sqrt{m}\rceil$，所以时间复杂度为 $\Theta(\sqrt{m})$，用 `map` 则多一个 log.

" 为什么要求 a 与 m 互质"

注意到我们求出的是 A, B，我们需要保证从 $a^{A\lceil\sqrt{m}\rceil} \equiv ba^B \pmod m$ 可以推回 $a^{A\lceil\sqrt{m}\rceil - B} \equiv b \pmod m$，后式是前式左右两边除以 a^B 得到，所以必须有 $a^B \perp m$ 即 $a \perp m$.

进阶篇

对 $a, b \in \mathbf{Z}^+$，$p \in \mathbf{P}$，求解

$$x^a \equiv b \pmod p$$

该问题可以转化为 BSGS 求解的问题。

由于式子中的模数 p 是一个质数，那么 p 一定存在一个原根 g. 因此对于模 p 意义下的任意的数 x $(1 \le x < p)$ 有且仅有一个数 i $(0 \le i < p-1)$ 满足 $x = g^i$.

方法一

我们令 $x = g^c$，g 是 p 的原根（我们一定可以找到这个 g 和 c），问题转化为求解 $(g^c)^a \equiv b \pmod p$. 稍加变换，得到

$$(g^a)^c \equiv b \pmod p$$

于是就转换成了 BSGS 的基本模型了，可以在 $O(\sqrt{p})$ 解出 c，这样可以得到原方程的一个特解 $x_0 \equiv g^c \pmod p$.

方法二

我们仍令 $x = g^c$，并且设 $b = g^t$，于是我们得到

$$g^{ac} \equiv g^t \pmod p$$

方程两边同时取离散对数得到

$$ac \equiv t \pmod{\varphi(p)}$$

我们可以通过 BSGS 求解 $g^t \equiv b \pmod p$ 得到 t，于是这就转化成了一个线性同余方程的问题。这样也可以解出 c，求出 x 的一个特解 $x_0 \equiv g^c \pmod p$.

找到所有解

在知道 $x_0 \equiv g^c \pmod p$ 的情况下，我们想得到原问题的所有解。首先我们知道 $g^{\varphi(p)} \equiv 1 \pmod p$，于是可以得到

$$\forall\ t \in \mathbf{Z},\ x^a \equiv g^{c \cdot a + t \cdot \varphi(p)} \equiv b \pmod p$$

于是得到所有解为

$$\forall\ t \in \mathbf{Z}, a \mid t \cdot \varphi(p),\ x \equiv g^{c + \frac{t \cdot \varphi(p)}{a}} \pmod p$$

对于上面这个式子，显然有 $\frac{a}{(a,\varphi(p))} \mid t$. 因此我们设 $t = \frac{a}{(a,\varphi(p))} \cdot i$，得到

$$\forall\ i \in \mathbf{Z}, x \equiv g^{c + \frac{\varphi(p)}{(a,\varphi(p))} \cdot i} \pmod p$$

这就是原问题的所有解。

实现

下面的代码实现的找原根、离散对数解和原问题所有解的过程。

" 参考代码"

```
int gcd(int a, int b) { return a ? gcd(b % a, a) : b; }

int powmod(int a, int b, int p) {
  int res = 1;
  while (b > 0) {
    if (b & 1) res = res * a % p;
    a = a * a % p, b >>= 1;
  }
  return res;
}

// Finds the primitive root modulo p
int generator(int p) {
  vector<int> fact;
  int phi = p - 1, n = phi;
  for (int i = 2; i * i <= n; ++i) {
    if (n % i == 0) {
      fact.push_back(i);
      while (n % i == 0) n /= i;
    }
  }
  if (n > 1) fact.push_back(n);
  for (int res = 2; res <= p; ++res) {
    bool ok = true;
    for (int factor : fact) {
      if (powmod(res, phi / factor, p) == 1) {
        ok = false;
        break;
      }
    }
    if (ok) return res;
  }
  return -1;
}

// This program finds all numbers x such that x^k=a (mod n)
int main() {
  int n, k, a;
  scanf("%d %d %d", &n, &k, &a);
  if (a == 0) return puts("1\n0"), 0;
  int g = generator(n);
  // Baby-step giant-step discrete logarithm algorithm
  int sq = (int)sqrt(n + .0) + 1;
  vector<pair<int, int>> dec(sq);
  for (int i = 1; i <= sq; ++i)
    dec[i - 1] = {powmod(g, i * sq * k % (n - 1), n), i};
  sort(dec.begin(), dec.end());
  int any_ans = -1;
  for (int i = 0; i < sq; ++i) {
```

```cpp
    int my = powmod(g, i * k % (n - 1), n) * a % n;
    auto it = lower_bound(dec.begin(), dec.end(), make_pair(my, 0));
    if (it != dec.end() && it->first == my) {
      any_ans = it->second * sq - i;
      break;
    }
  }
  if (any_ans == -1) return puts("0"), 0;
  // Print all possible answers
  int delta = (n - 1) / gcd(k, n - 1);
  vector<int> ans;
  for (int cur = any_ans % delta; cur < n - 1; cur += delta)
    ans.push_back(powmod(g, cur, n));
  sort(ans.begin(), ans.end());
  printf("%d\n", ans.size());
  for (int answer : ans) printf("%d ", answer);
}
```

扩展篇（扩展 BSGS）

对 $a,b,m \in \mathbf{Z}^+$，求解

$$a^x \equiv b \pmod m$$

其中 a,m 不一定互质。

当 $(a,m)=1$ 时，在模 m 意义下 a 存在逆元，因此可以使用 BSGS 算法求解。于是我们想办法让他们变得互质。

具体地，设 $d_1=(a,m)$. 如果 $d_1 \nmid b$，则原方程无解。否则我们把方程同时除以 d_1，得到

$$\frac{a}{d_1}\cdot a^{x-1} \equiv \frac{b}{d_1} \pmod{\frac{m}{d_1}}$$

如果 a 和 $\frac{m}{d_1}$ 仍不互质就再除，设 $d_2=\left(a,\frac{m}{d_1}\right)$. 如果 $d_2 \nmid \frac{b}{d_1}$，则方程无解；否则同时除以 d_2 得到

$$\frac{a^2}{d_1 d_2}\cdot a^{x-2} \equiv \frac{b}{d_1 d_2} \pmod{\frac{m}{d_1 d_2}}$$

同理，这样不停的判断下去，直到 $a \perp \dfrac{m}{d_1 d_2 \cdots d_k}$.

记 $D=\prod_{i=1}^k d_i$，于是方程就变成了这样：

$$\frac{a^k}{D}\cdot a^{x-k} \equiv \frac{b}{D} \pmod{\frac{m}{D}}$$

由于 $a \perp \dfrac{m}{D}$，于是推出 $\dfrac{a^k}{D} \perp \dfrac{m}{D}$. 这样 $\dfrac{a^k}{D}$ 就有逆元了，于是把它丢到方程右边，这就是一个普通的 BSGS 问题了，于是求解 $x-k$ 后再加上 k 就是原方程的解啦。

注意，不排除解小于等于 k 的情况，所以在消因子之前做一下 $\Theta(k)$ 枚举，直接验证 $a^i \equiv b \pmod m$，这样就能避免这种情况。

习题

- SPOJ MOD[2] 模板
- SDOI2013 随机数生成器[3]
- SGU261 Discrete Roots[4] 模板
- SDOI2011 计算器[5] 模板
- Luogu4195【模板】exBSGS/Spoj3105 Mod[6] 模板

- Codeforces - Lunar New Year and a Recursive Sequence[7]
- LOJ6542 离散对数[8] index calculus 方法，非模板

本页面部分内容以及代码译自博文 [9] 与其英文翻译版 Discrete Root[10]。其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0。

参考资料

1. Discrete logarithm - Wikipedia[11]
2. 潘承洞，潘承彪。初等数论。
3. 冯克勤。初等数论及其应用。

参考资料与注释

[1] Ed25519

[2] SPOJ MOD

[3] SDOI2013 随机数生成器

[4] SGU261 Discrete Roots

[5] SDOI2011 计算器

[6] Luogu4195【模板】exBSGS/Spoj3105 Mod

[7] Codeforces - Lunar New Year and a Recursive Sequence

[8] LOJ6542 离散对数

[9]

[10] Discrete Root

[11] Discrete logarithm - Wikipedia

9.12.22 剩余

剩余

前置知识：离散对数

模运算下的剩余问题，是将开方运算引入模运算的尝试。

定义

令整数 $k \geq 2$，整数 a，m 满足 $(a,m)=1$，若存在整数 x 使得

$$x^k \equiv a \pmod{m} \tag{1}$$

则称 a 为模 m 的 k 次剩余，否则称 a 为模 m 的 k 次非剩余。

二次剩余 即是 k 次剩余的特例。

性质

当整数 $k \geq 2$，整数 a，m 满足 $(a, m) = 1$，模 m **有原根** g 时，令 $d = (k, \varphi(m))$，则：

1. a 为模 m 的 k 次剩余当且仅当 $d \mid \operatorname{ind}_g a$，即：

$$a^{\frac{\varphi(m)}{d}} \equiv 1 \pmod m$$

2. 方程 (1) 若有解，则模 m 下恰有 d 个解
3. 模 m 的 k 次剩余类的个数为 $\dfrac{\varphi(m)}{d}$，其有形式

$$a \equiv g^{di} \pmod m, \qquad \left(0 \leq i < \frac{\varphi(m)}{d}\right)$$

" 证明"

1. 令 $x = g^y$，则方程 (1) 等价于

$$g^{ky} \equiv g^{\operatorname{ind}_g a} \pmod m$$

其等价于

$$ky \equiv \operatorname{ind}_g a \pmod{\varphi(m)} \tag{2}$$

由同余的性质，我们知道 y 有整数解当且仅当 $d = (k, \varphi(m)) \mid \operatorname{ind}_g a$，进而

$$\begin{aligned}
a^{\frac{\varphi(m)}{d}} \equiv 1 \pmod m &\iff g^{\frac{\varphi(m)}{d}\operatorname{ind}_g a} \equiv 1 \pmod m \\
&\iff \varphi(m) \mid \frac{\varphi(m)}{d} \operatorname{ind}_g a \\
&\iff \varphi(m) d \mid \varphi(m) \operatorname{ind}_g a \\
&\iff d \mid \operatorname{ind}_g a
\end{aligned}$$

2. 当 $d \mid \operatorname{ind}_g a$ 时，由同余的性质可知方程 (2) 模 $\varphi(m)$ 下恰有 d 个解，进而方程 (1) 模 m 下恰有 d 个解。
3. 由 1 知 a 为模 m 的 k 次剩余当且仅当 $d \mid \operatorname{ind}_g a$，故

$$\operatorname{ind}_g a \equiv di \pmod{\varphi(m)}, \qquad \left(0 \leq i < \frac{\varphi(m)}{d}\right)$$

故模 m 的 k 次剩余共有 $\dfrac{\varphi(m)}{d}$ 个同余类：

$$a \equiv g^{di} \pmod m, \qquad \left(0 \leq i < \frac{\varphi(m)}{d}\right)$$

参考资料

1. 冯克勤。初等数论及其应用。

9.12.23 莫比乌斯反演

Authors: hydingsy, hyp1231, ranwen, 383494

引入

莫比乌斯反演是数论中的重要内容。对于一些函数 $f(n)$，如果很难直接求出它的值，而容易求出其倍数和或约数和 $g(n)$，那么可以通过莫比乌斯反演简化运算，求得 $f(n)$ 的值。

开始学习莫比乌斯反演前，需要先学习一些前置知识：数论分块、狄利克雷卷积。

莫比乌斯函数

定义

μ 为莫比乌斯函数，定义为

$$\mu(n)=\begin{cases}1 & n=1\\ 0 & n\text{ 含有平方因子}\\ (-1)^k & k\text{ 为}n\text{ 的本质不同质因子个数}\end{cases}$$

详细解释一下：

令 $n=\prod_{i=1}^k p_i^{c_i}$，其中 p_i 为质因子，$c_i\ge 1$。上述定义表示：

1. $n=1$ 时，$\mu(n)=1$；
2. 对于 $n\ne 1$ 时：
   (a) 当存在 $i\in[1,k]$，使得 $c_i>1$ 时，$\mu(n)=0$，也就是说只要某个质因子出现的次数超过一次，$\mu(n)$ 就等于 0；
   (b) 当任意 $i\in[1,k]$，都有 $c_i=1$ 时，$\mu(n)=(-1)^k$，也就是说每个质因子都仅仅只出现过一次时，即 $n=\prod_{i=1}^k p_i$，$\{p_i\}_{i=1}^k$ 中个元素唯一时，$\mu(n)$ 等于 -1 的 k 次幂，此处 k 指的便是仅仅只出现过一次的质因子的总个数。

性质

莫比乌斯函数不仅是积性函数，还有如下性质：

$$\sum_{d\mid n}\mu(d)=\begin{cases}1 & n=1\\ 0 & n\ne 1\end{cases}$$

即 $\sum_{d\mid n}\mu(d)=\varepsilon(n)$，$\mu*1=\varepsilon$

证明

设 $n=\prod\limits_{i=1}^k {p_i}^{c_i}, n'=\prod\limits_{i=1}^k p_i$

那么 $\sum\limits_{d\mid n}\mu(d)=\sum\limits_{d\mid n'}\mu(d)=\sum\limits_{i=0}^k\binom{k}{i}\cdot(-1)^i=(1+(-1))^k$

根据二项式定理，易知该式子的值在 $k=0$ 即 $n=1$ 时值为 1 否则为 0，这也同时证明了 $\sum\limits_{d\mid n}\mu(d)=[n=1]=\varepsilon(n)$ 以及 $\mu*1=\varepsilon$

" 注"

这个性质意味着，莫比乌斯函数在狄利克雷生成函数中，等价于黎曼函数 ζ 的倒数。所以在狄利克雷卷积中，莫比乌斯函数是常数函数 1 的逆元。

补充结论

反演结论：$[\gcd(i,j)=1]=\sum\limits_{d\mid\gcd(i,j)}\mu(d)$

直接推导：如果看懂了上一个结论，这个结论稍加思考便可以推出：如果 $\gcd(i,j)=1$ 的话，那么代表着我们按上个结论中枚举的那个 n 是 1，也就是式子的值是 1，反之，有一个与 $[\gcd(i,j)=1]$ 相同的值：0

利用 ε 函数：根据上一结论，$[\gcd(i,j)=1]=\varepsilon(\gcd(i,j))$，将 ε 展开即可。

线性筛

由于 μ 函数为积性函数，因此可以线性筛莫比乌斯函数（线性筛基本可以求所有的积性函数，尽管方法不尽相同）。

" 线性筛实现"

```
void getMu() {
  mu[1] = 1;
  for (int i = 2; i <= n; ++i) {
    if (!flg[i]) p[++tot] = i, mu[i] = -1;
    for (int j = 1; j <= tot && i * p[j] <= n; ++j) {
      flg[i * p[j]] = 1;
      if (i % p[j] == 0) {
        mu[i * p[j]] = 0;
        break;
      }
      mu[i * p[j]] = -mu[i];
    }
  }
}
```

```
def getMu():
mu[1] = 1
for i in range(2, n + 1):
    if flg[i] != 0:
        p[tot] = i; tot = tot + 1; mu[i] = -1
    j = 1
    while j <= tot and i * p[j] <= n:
        flg[i * p[j]] = 1
        if i % p[j] == 0:
            mu[i * p[j]] = 0
            break
        mu[i * p[j]] = mu[i * p[j]] - mu[i]
        j = j + 1
```

拓展

证明

$$\varphi * 1 = \mathrm{id}$$

将 n 分解质因数：$n=\prod_{i=1}^{k} {p_i}^{c_i}$

首先，因为 φ 是积性函数，故只要证明当 $n'=p^c$ 时 $\varphi * 1=\sum_{d|n'}\varphi(\frac{n'}{d})=\mathrm{id}$ 成立即可。

因为 p 是质数，于是 $d=p^0,p^1,p^2,\cdots,p^c$

易知如下过程：

$$
\begin{aligned}
\varphi * 1 &= \sum_{d|n} \varphi(\frac{n}{d}) \\
&= \sum_{i=0}^{c} \varphi(p^i) \\
&= 1 + p^0 \cdot (p-1) + p^1 \cdot (p-1) + \cdots + p^{c-1} \cdot (p-1) \\
&= p^c \\
&= \mathrm{id}
\end{aligned}
$$

该式子两侧同时卷 μ 可得 $\varphi(n) = \sum\limits_{d|n} d \cdot \mu(\frac{n}{d})$

莫比乌斯变换

设 $f(n), g(n)$ 为两个数论函数。

形式一：如果有 $f(n) = \sum_{d|n} g(d)$，那么有 $g(n) = \sum_{d|n} \mu(d) f(\frac{n}{d})$。

这种形式下，数论函数 $f(n)$ 称为数论函数 $g(n)$ 的莫比乌斯变换，数论函数 $g(n)$ 称为数论函数 $f(n)$ 的莫比乌斯逆变换（反演）。

容易看出，数论函数 $g(n)$ 的莫比乌斯变换，就是将数论函数 $g(n)$ 与常数函数 1 进行狄利克雷卷积。

" 注"

根据狄利克雷卷积与狄利克雷生成函数的对应关系，数论函数 $g(n)$ 的莫比乌斯变换对应的狄利克雷生成函数，就是数论函数 $g(n)$ 的狄利克雷生成函数与黎曼函数 ζ 的乘积。

形式二：如果有 $f(n) = \sum_{n|d} g(d)$，那么有 $g(n) = \sum_{n|d} \mu(\frac{d}{n}) f(d)$。

证明

方法一：对原式做数论变换。

$$
\sum_{d|n} \mu(d) f(\frac{n}{d}) = \sum_{d|n} \mu(d) \sum_{k|\frac{n}{d}} g(k) = \sum_{k|n} g(k) \sum_{d|\frac{n}{k}} \mu(d) = g(n)
$$

用 $\sum\limits_{d|n} g(d)$ 来替换 $f(\frac{n}{d})$，再变换求和顺序。最后一步变换的依据：$\sum\limits_{d|n} \mu(d) = [n=1]$，因此在 $\frac{n}{k} = 1$ 时第二个和式的值才为 1。此时 $n = k$，故原式等价于 $\sum\limits_{k|n} [n=k] \cdot g(k) = g(n)$

方法二：运用卷积。

原问题为：已知 $f = g * 1$，证明 $g = f * \mu$

易知如下转化：$f * \mu = g * 1 * \mu \implies f * \mu = g$（其中 $1 * \mu = \varepsilon$）。

对于第二种形式：

类似上面的方法一，我们考虑逆推这个式子。

$$
\begin{aligned}
&\sum_{n|d} \mu(\frac{d}{n}) f(d) \\
=&\sum_{k=1}^{+\infty} \mu(k) f(kn) = \sum_{k=1}^{+\infty} \mu(k) \sum_{kn|d} g(d) \\
=&\sum_{n|d} g(d) \sum_{k|\frac{d}{n}} \mu(k) = \sum_{n|d} g(d) \epsilon(\frac{d}{n}) \\
=&g(n)
\end{aligned}
$$

我们把 d 表示为 kn 的形式，然后把 f 的原定义代入式子。

发现枚举 k 再枚举 kn 的倍数可以转换为直接枚举 n 的倍数再求出 k，发现后面那一块其实就是 ϵ，整个式子只有在 $d=n$ 的时候才能取到值。

问题形式

「HAOI 2011」Problem b[1]

求值（多组数据）

$$\sum_{i=x}^{n}\sum_{j=y}^{m}[\gcd(i,j)=k] \qquad (1\leqslant T,x,y,n,m,k\leqslant 5\times 10^4)$$

根据容斥原理，原式可以分成 4 块来处理，每一块的式子都为

$$\sum_{i=1}^{n}\sum_{j=1}^{m}[\gcd(i,j)=k]$$

考虑化简该式子

$$\sum_{i=1}^{\lfloor\frac{n}{k}\rfloor}\sum_{j=1}^{\lfloor\frac{m}{k}\rfloor}[\gcd(i,j)=1]$$

因为 $\gcd(i,j)=1$ 时对答案才用贡献，于是我们可以将其替换为 $\varepsilon(\gcd(i,j))$（$\varepsilon(n)$ 当且仅当 $n=1$ 时值为 1 否则为 0），故原式化为

$$\sum_{i=1}^{\lfloor\frac{n}{k}\rfloor}\sum_{j=1}^{\lfloor\frac{m}{k}\rfloor}\varepsilon(\gcd(i,j))$$

将 ε 函数展开得到

$$\sum_{i=1}^{\lfloor\frac{n}{k}\rfloor}\sum_{j=1}^{\lfloor\frac{m}{k}\rfloor}\sum_{d\mid\gcd(i,j)}\mu(d)$$

变换求和顺序，先枚举 $d\mid\gcd(i,j)$ 可得

$$\sum_{d=1}\mu(d)\sum_{i=1}^{\lfloor\frac{n}{k}\rfloor}[d\mid i]\sum_{j=1}^{\lfloor\frac{m}{k}\rfloor}[d\mid j]$$

易知 $1\sim\lfloor\dfrac{n}{k}\rfloor$ 中 d 的倍数有 $\lfloor\dfrac{n}{kd}\rfloor$ 个，故原式化为

$$\sum_{d=1}^{\min(\lfloor\frac{n}{k}\rfloor,\lfloor\frac{m}{k}\rfloor)}\mu(d)\lfloor\frac{n}{kd}\rfloor\lfloor\frac{m}{kd}\rfloor$$

很显然，式子可以数论分块求解。

时间复杂度 $\Theta(N+T\sqrt{n})$

" 代码实现"

```cpp
#include <algorithm>
#include <cstdio>
using namespace std;
const int N = 50000;
int mu[N + 5], p[N + 5];
bool flg[N + 5];

void init() {
  int tot = 0;
  mu[1] = 1;
  for (int i = 2; i <= N; ++i) {
```

```
    if (!flg[i]) {
      p[++tot] = i;
      mu[i] = -1;
    }
    for (int j = 1; j <= tot && i * p[j] <= N; ++j) {
      flg[i * p[j]] = 1;
      if (i % p[j] == 0) {
        mu[i * p[j]] = 0;
        break;
      }
      mu[i * p[j]] = -mu[i];
    }
  }
  for (int i = 1; i <= N; ++i) mu[i] += mu[i - 1];
}

int solve(int n, int m) {
  int res = 0;
  for (int i = 1, j; i <= min(n, m); i = j + 1) {
    j = min(n / (n / i), m / (m / i));
    res += (mu[j] - mu[i - 1]) * (n / i) * (m / i);  // 代推出来的式子
  }
  return res;
}

int main() {
  int T, a, b, c, d, k;
  init();  // 预处理 mu 数组
  scanf("%d", &T);
  for (int i = 1; i <= T; i++) {
    scanf("%d%d%d%d%d", &a, &b, &c, &d, &k);
    // 根据容斥原理，1<=x<=b&&1<=y<=d 范围中的答案数减去 1<=x<=b&&1<=y<=c-1 范围中的
答案数和
    //   1<=x<=a-1&&1<=y<=d 范围中的答案数再加上 1<=x<=a-1&&1<=y<=c-1 范围中的答案数
    //   即可得到 a<=x<=b&&c<=y<=d 范围中的答案数
    // 这一步如果不懂可以画坐标图进行理解
    printf("%d\n", solve(b / k, d / k) - solve(b / k, (c - 1) / k) -
                       solve((a - 1) / k, d / k) +
                       solve((a - 1) / k, (c - 1) / k));
  }
  return 0;
}
```

「SPOJ 5971」LCMSUM[2]

求值（多组数据）

$$\sum_{i=1}^{n}\operatorname{lcm}(i,n) \quad \text{s.t. } 1\leqslant T\leqslant 3\times 10^5, 1\leqslant n\leqslant 10^6$$

易得原式即

$$\sum_{i=1}^{n}\frac{i\cdot n}{\gcd(i,n)}$$

将原式复制一份并且颠倒顺序，然后将 n 一项单独提出，可得

$$\frac{1}{2}\cdot\left(\sum_{i=1}^{n-1}\frac{i\cdot n}{\gcd(i,n)}+\sum_{i=n-1}^{1}\frac{i\cdot n}{\gcd(i,n)}\right)+n$$

根据 $\gcd(i,n)=\gcd(n-i,n)$，可将原式化为

$$\frac{1}{2}\cdot\left(\sum_{i=1}^{n-1}\frac{i\cdot n}{\gcd(i,n)}+\sum_{i=n-1}^{1}\frac{i\cdot n}{\gcd(n-i,n)}\right)+n$$

两个求和式中分母相同的项可以合并。

$$\frac{1}{2}\cdot\sum_{i=1}^{n-1}\frac{n^2}{\gcd(i,n)}+n$$

即

$$\frac{1}{2}\cdot\sum_{i=1}^{n}\frac{n^2}{\gcd(i,n)}+\frac{n}{2}$$

可以将相同的 $\gcd(i,n)$ 合并在一起计算，故只需要统计 $\gcd(i,n)=d$ 的个数。当 $\gcd(i,n)=d$ 时，$\gcd(\frac{i}{d},\frac{n}{d})=1$，所以 $\gcd(i,n)=d$ 的个数有 $\varphi(\frac{n}{d})$ 个。

故答案为

$$\frac{1}{2}\cdot\sum_{d|n}\frac{n^2\cdot\varphi(\frac{n}{d})}{d}+\frac{n}{2}$$

变换求和顺序，设 $d'=\frac{n}{d}$，合并公因式，式子化为

$$\frac{1}{2}n\cdot\left(\sum_{d'|n}d'\cdot\varphi(d')+1\right)$$

设 $\mathrm{g}(n)=\sum_{d|n}d\cdot\varphi(d)$，已知 g 为积性函数，于是可以 $\Theta(n)$ 筛出。每次询问 $\Theta(1)$ 计算即可。

下面给出这个函数筛法的推导过程：

首先考虑 $\mathrm{g}(p_j^k)$ 的值，显然它的约数只有 $p_j^0,p_j^1,\cdots,p_j^k$，因此

$$\mathrm{g}(p_j^k)=\sum_{w=0}^{k}p_j^w\cdot\varphi(p_j^w)$$

又有 $\varphi(p_j^w)=p_j^{w-1}\cdot(p_j-1)$，则原式可化为

$$\sum_{w=0}^{k}p_j^{2w-1}\cdot(p_j-1)$$

于是有

$$\mathrm{g}(p_j^{k+1})=\mathrm{g}(p_j^k)+p_j^{2k+1}\cdot(p_j-1)$$

那么，对于线性筛中的 $\mathrm{g}(i\cdot p_j)(p_j|i)$，令 $i=a\cdot p_j^w(\gcd(a,p_j)=1)$，可得

$$\mathrm{g}(i\cdot p_j)=\mathrm{g}(a)\cdot\mathrm{g}(p_j^{w+1})$$

$$\mathrm{g}(i)=\mathrm{g}(a)\cdot\mathrm{g}(p_j^w)$$

即

$$\mathrm{g}(i\cdot p_j)-\mathrm{g}(i)=\mathrm{g}(a)\cdot p_j^{2w+1}\cdot(p_j-1)$$

同理有

$$g(i) - g(\frac{i}{p_j}) = g(a) \cdot p_j^{2w-1} \cdot (p_j - 1)$$

因此

$$g(i \cdot p_j) = g(i) + \left(g(i) - g(\frac{i}{p_j})\right) \cdot p_j^2$$

时间复杂度：$\Theta(n+T)$

" 代码实现"

```cpp
#include <cstdio>
const int N = 1000000;
int tot, p[N + 5];
long long g[N + 5];
bool flg[N + 5];  // 标记数组

void solve() {
  g[1] = 1;
  for (int i = 2; i <= N; ++i) {
    if (!flg[i]) {
      p[++tot] = i;
      g[i] = (long long)1 * i * (i - 1) + 1;
    }
    for (int j = 1; j <= tot && i * p[j] <= N; ++j) {
      flg[i * p[j]] = 1;
      if (i % p[j] == 0) {
        g[i * p[j]] =
            g[i] + (g[i] - g[i / p[j]]) * p[j] * p[j];  // 代入推出来的式子
        break;
      }
      g[i * p[j]] = g[i] * g[p[j]];
    }
  }
}

int main() {
  int T, n;
  solve();  // 预处理 g 数组
  scanf("%d", &T);
  for (int i = 1; i <= T; ++i) {
    scanf("%d", &n);
    printf("%lld\n", (g[n] + 1) * n / 2);
  }
  return 0;
}
```

「BZOJ 2154」Crash 的数字表格[3]

求值（对 20101009 取模）

$$\sum_{i=1}^{n}\sum_{j=1}^{m} \operatorname{lcm}(i,j) \qquad (n,m \leqslant 10^7)$$

易知原式等价于

$$\sum_{i=1}^{n}\sum_{j=1}^{m}\frac{i\cdot j}{\gcd(i,j)}$$

枚举最大公因数 d，显然两个数除以 d 得到的数互质

$$\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{d\mid i,d\mid j,\gcd(\frac{i}{d},\frac{j}{d})=1}\frac{i\cdot j}{d}$$

非常经典的 gcd 式子的化法

$$\sum_{d=1}^{n}d\cdot\sum_{i=1}^{\lfloor\frac{n}{d}\rfloor}\sum_{j=1}^{\lfloor\frac{m}{d}\rfloor}[\gcd(i,j)=1]\ i\cdot j$$

后半段式子中，出现了互质数对之积的和，为了让式子更简洁就把它拿出来单独计算。于是我们记

$$\text{sum}(n,m)=\sum_{i=1}^{n}\sum_{j=1}^{m}[\gcd(i,j)=1]\ i\cdot j$$

接下来对 $\text{sum}(n,m)$ 进行化简。首先枚举约数，并将 $[\gcd(i,j)=1]$ 表示为 $\varepsilon(\gcd(i,j))$

$$\sum_{d=1}^{n}\sum_{d\mid i}^{n}\sum_{d\mid j}^{m}\mu(d)\cdot i\cdot j$$

设 $i=i'\cdot d$，$j=j'\cdot d$，显然式子可以变为

$$\sum_{d=1}^{n}\mu(d)\cdot d^2\cdot\sum_{i=1}^{\lfloor\frac{n}{d}\rfloor}\sum_{j=1}^{\lfloor\frac{m}{d}\rfloor}i\cdot j$$

观察上式，前半段可以预处理前缀和；后半段又是一个范围内数对之和，记

$$g(n,m)=\sum_{i=1}^{n}\sum_{j=1}^{m}i\cdot j=\frac{n\cdot(n+1)}{2}\times\frac{m\cdot(m+1)}{2}$$

可以 $\Theta(1)$ 求解

至此

$$\text{sum}(n,m)=\sum_{d=1}^{n}\mu(d)\cdot d^2\cdot g(\lfloor\frac{n}{d}\rfloor,\lfloor\frac{m}{d}\rfloor)$$

我们可以 $\lfloor\frac{n}{\lfloor\frac{n}{d}\rfloor}\rfloor$ 数论分块求解 $\text{sum}(n,m)$ 函数。

在求出 $\text{sum}(n,m)$ 后，回到定义 sum 的地方，可得原式为

$$\sum_{d=1}^{n}d\cdot\text{sum}(\lfloor\frac{n}{d}\rfloor,\lfloor\frac{m}{d}\rfloor)$$

可见这又是一个可以数论分块求解的式子！

本题除了推式子比较复杂、代码细节较多之外，是一道很好的莫比乌斯反演练习题！（上述过程中，默认 $n\leqslant m$）

时间复杂度：$\Theta(n+m)$（瓶颈为线性筛）

" 代码实现"

```
#include <algorithm>
#include <cstdio>
using namespace std;

const int N = 1e7;
const int mod = 20101009;
int n, m, mu[N + 5], p[N / 10 + 5], sum[N + 5];
bool flg[N + 5];
```

```cpp
int Sum(int x, int y) {
  return ((long long)1 * x * (x + 1) / 2 % mod) *
         ((long long)1 * y * (y + 1) / 2 % mod) % mod;
}

int func(int x, int y) {
  int res = 0;
  int j;
  for (int i = 1; i <= min(x, y); i = j + 1) {
    j = min(x / (x / i), y / (y / i));
    res = (res + (long long)1 * (sum[j] - sum[i - 1] + mod) *
                     Sum(x / i, y / i) % mod) %
          mod;  //+mod 防负数
  }
  return res;
}

int solve(int x, int y) {
  int res = 0;
  int j;
  for (int i = 1; i <= min(x, y); i = j + 1) {  // 整除分块处理
    j = min(x / (x / i), y / (y / i));
    res = (res + (long long)1 * (j - i + 1) * (i + j) / 2 % mod *
                     func(x / i, y / i) % mod) %
          mod;  // ! 每步取模防爆
  }
  return res;
}

void init() {  // 线性筛
  mu[1] = 1;
  int tot = 0, k = min(n, m);
  for (int i = 2; i <= k; ++i) {
    if (!flg[i]) {
      p[++tot] = i;
      mu[i] = -1;
    }
    for (int j = 1; j <= tot && i * p[j] <= k; ++j) {
      flg[i * p[j]] = 1;
      if (i % p[j] == 0) {
        mu[i * p[j]] = 0;
        break;
      }
      mu[i * p[j]] = -mu[i];
    }
  }
  for (int i = 1; i <= k; ++i)
    sum[i] = (sum[i - 1] + (long long)1 * i * i % mod * (mu[i] + mod)) % mod;
}

int main() {
  scanf("%d%d", &n, &m);
  init();
```

```
  printf("%d\n", solve(n, m));
}
```

「SDOI2015」 约数个数和[4]

多组数据，求

$$\sum_{i=1}^{n}\sum_{j=1}^{m} d(i\cdot j) \qquad (n,m,T\leq 5\times 10^4)$$

其中 $d(n)=\sum_{i|n}1$，$d(n)$ 表示 n 的约数个数

要推这道题首先要了解 d 函数的一个特殊性质

$$d(i\cdot j)=\sum_{x|i}\sum_{y|j}[\gcd(x,y)=1]$$

再化一下这个式子

$$\begin{aligned}
d(i\cdot j) &= \sum_{x|i}\sum_{y|j}[\gcd(x,y)=1]\\
&= \sum_{x|i}\sum_{y|j}\sum_{p|\gcd(x,y)}\mu(p)\\
&= \sum_{p=1}^{min(i,j)}\sum_{x|i}\sum_{y|j}[p\mid \gcd(x,y)]\cdot\mu(p)\\
&= \sum_{p|i,p|j}\mu(p)\sum_{x|i}\sum_{y|j}[p\mid \gcd(x,y)]\\
&= \sum_{p|i,p|j}\mu(p)\sum_{x|\frac{i}{p}}\sum_{y|\frac{j}{p}}1\\
&= \sum_{p|i,p|j}\mu(p)d\left(\frac{i}{p}\right)d\left(\frac{j}{p}\right)
\end{aligned}$$

将上述式子代回原式

$$\begin{aligned}
&\sum_{i=1}^{n}\sum_{j=1}^{m}d(i\cdot j)\\
=&\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{p|i,p|j}\mu(p)d\left(\frac{i}{p}\right)d\left(\frac{j}{p}\right)\\
=&\sum_{p=1}^{min(n,m)}\sum_{i=1}^{n}\sum_{j=1}^{m}[p\mid i,p\mid j]\cdot\mu(p)d\left(\frac{i}{p}\right)d\left(\frac{j}{p}\right)\\
=&\sum_{p=1}^{min(n,m)}\sum_{i=1}^{\left\lfloor\frac{n}{p}\right\rfloor}\sum_{j=1}^{\left\lfloor\frac{m}{p}\right\rfloor}\mu(p)d(i)d(j)\\
=&\sum_{p=1}^{min(n,m)}\mu(p)\sum_{i=1}^{\left\lfloor\frac{n}{p}\right\rfloor}d(i)\sum_{j=1}^{\left\lfloor\frac{m}{p}\right\rfloor}d(j)\\
=&\sum_{p=1}^{min(n,m)}\mu(p)S\left(\left\lfloor\frac{n}{p}\right\rfloor\right)S\left(\left\lfloor\frac{m}{p}\right\rfloor\right)\quad\left(S(n)=\sum_{i=1}^{n}d(i)\right)
\end{aligned}$$

那么 $O(n)$ 预处理 μ,d 的前缀和，$O(\sqrt{n})$ 分块处理询问，总复杂度 $O(n+T\sqrt{n})$.

" 代码实现"

```
#include <algorithm>
#include <cstdio>
using namespace std;
```

```cpp
const long long N = 5e4 + 5;
long long n, m, T, pr[N], mu[N], d[N], t[N],
    cnt;  // t 表示 i 的最小质因子出现的次数
bool bp[N];

void prime_work(long long k) {
  bp[0] = bp[1] = 1, mu[1] = 1, d[1] = 1;
  for (long long i = 2; i <= k; i++) {  // 线性筛
    if (!bp[i]) pr[++cnt] = i, mu[i] = -1, d[i] = 2, t[i] = 1;
    for (long long j = 1; j <= cnt && i * pr[j] <= k; j++) {
      bp[i * pr[j]] = 1;
      if (i % pr[j] == 0) {
        mu[i * pr[j]] = 0;
        d[i * pr[j]] = d[i] / (t[i] + 1) * (t[i] + 2);
        t[i * pr[j]] = t[i] + 1;
        break;
      } else {
        mu[i * pr[j]] = -mu[i];
        d[i * pr[j]] = d[i] << 1;
        t[i * pr[j]] = 1;
      }
    }
  }
  for (long long i = 2; i <= k; i++)
    mu[i] += mu[i - 1], d[i] += d[i - 1];  // 求前缀和
}

long long solve() {
  long long res = 0, mxi = min(n, m);
  for (long long i = 1, j; i <= mxi; i = j + 1) {  // 整除分块
    j = min(n / (n / i), m / (m / i));
    res += d[n / i] * d[m / i] * (mu[j] - mu[i - 1]);
  }
  return res;
}

int main() {
  scanf("%lld", &T);
  prime_work(50000);  // 预处理
  while (T--) {
    scanf("%lld%lld", &n, &m);
    printf("%lld\n", solve());
  }
  return 0;
}
```

莫比乌斯反演扩展

结尾补充一个莫比乌斯反演的非卷积形式。

对于数论函数 f,g 和完全积性函数 t 且 $t(1)=1$：

$$f(n)=\sum_{i=1}^{n}t(i)g\left(\left\lfloor\frac{n}{i}\right\rfloor\right)$$

$$\Longleftrightarrow$$

$$g(n)=\sum_{i=1}^{n}\mu(i)t(i)f\left(\left\lfloor\frac{n}{i}\right\rfloor\right)$$

我们证明一下

$$\begin{aligned}
g(n)&=\sum_{i=1}^{n}\mu(i)t(i)f\left(\left\lfloor\frac{n}{i}\right\rfloor\right)\\
&=\sum_{i=1}^{n}\mu(i)t(i)\sum_{j=1}^{\lfloor\frac{n}{i}\rfloor}t(j)g\left(\left\lfloor\frac{\lfloor\frac{n}{i}\rfloor}{j}\right\rfloor\right)\\
&=\sum_{i=1}^{n}\mu(i)t(i)\sum_{j=1}^{\lfloor\frac{n}{i}\rfloor}t(j)g\left(\left\lfloor\frac{n}{ij}\right\rfloor\right)\\
&=\sum_{T=1}^{n}\sum_{i=1}^{n}\mu(i)t(i)\sum_{j=1}^{\lfloor\frac{n}{i}\rfloor}[ij=T]t(j)g\left(\left\lfloor\frac{n}{T}\right\rfloor\right) && \text{【先枚举 ij 乘积】}\\
&=\sum_{T=1}^{n}\sum_{i|T}\mu(i)t(i)t\left(\frac{T}{i}\right)g\left(\left\lfloor\frac{n}{T}\right\rfloor\right) && \text{【}\sum_{j=1}^{\lfloor\frac{n}{i}\rfloor}[ij=T]\text{对答案的贡献为 1，于是省略】}\\
&=\sum_{T=1}^{n}g\left(\left\lfloor\frac{n}{T}\right\rfloor\right)\sum_{i|T}\mu(i)t(i)t\left(\frac{T}{i}\right)\\
&=\sum_{T=1}^{n}g\left(\left\lfloor\frac{n}{T}\right\rfloor\right)\sum_{i|T}\mu(i)t(T) && \text{【t 是完全积性函数】}\\
&=\sum_{T=1}^{n}g\left(\left\lfloor\frac{n}{T}\right\rfloor\right)t(T)\sum_{i|T}\mu(i)\\
&=\sum_{T=1}^{n}g\left(\left\lfloor\frac{n}{T}\right\rfloor\right)t(T)\varepsilon(T) && \text{【}\mu*1=\varepsilon\text{】}\\
&=g(n)t(1) && \text{【当且仅当 T=1,}\varepsilon(T)=1\text{时】}\\
&=g(n) && \square
\end{aligned}$$

参考文献

algocode 算法博客[5]

参考资料与注释

[1] 「HAOI 2011」Problem b

[2] 「SPOJ 5971」LCMSUM

[3] 「BZOJ 2154」Crash 的数字表格

[4] 「SDOI2015」约数个数和

[5] algocode 算法博客

9.12.24 杜教筛

Authors: hsfzLZH1, sshwy, StudyingFather, Marcythm

杜教筛被用于处理一类数论函数的前缀和问题。对于数论函数 f，杜教筛可以在低于线性时间的复杂度内计算 $S(n)=\sum_{i=1}^{n} f(i)$。

算法思想

我们想办法构造一个 $S(n)$ 关于 $S\left(\left\lfloor\frac{n}{i}\right\rfloor\right)$ 的递推式。

对于任意一个数论函数 g，必满足：

$$\begin{aligned}\sum_{i=1}^{n}(f*g)(i) &= \sum_{i=1}^{n}\sum_{d\mid i} g(d)f\left(\frac{i}{d}\right)\\ &= \sum_{i=1}^{n} g(i)S\left(\left\lfloor\frac{n}{i}\right\rfloor\right)\end{aligned}$$

其中 $f*g$ 为数论函数 f 和 g 的 狄利克雷卷积。

" 略证"

$g(d)f\left(\frac{i}{d}\right)$ 就是对所有 $i\le n$ 的做贡献，因此变换枚举顺序，枚举 $d,\frac{i}{d}$（分别对应新的 i,j）

$$\begin{aligned}\sum_{i=1}^{n}\sum_{d\mid i} g(d)f\left(\frac{i}{d}\right) &= \sum_{i=1}^{n}\sum_{j=1}^{\lfloor n/i\rfloor} g(i)f(j)\\ &= \sum_{i=1}^{n} g(i)\sum_{j=1}^{\lfloor n/i\rfloor} f(j)\\ &= \sum_{i=1}^{n} g(i)S\left(\left\lfloor\frac{n}{i}\right\rfloor\right)\end{aligned}$$

那么可以得到递推式：

$$\begin{aligned}g(1)S(n) &= \sum_{i=1}^{n} g(i)S\left(\left\lfloor\frac{n}{i}\right\rfloor\right)-\sum_{i=2}^{n} g(i)S\left(\left\lfloor\frac{n}{i}\right\rfloor\right)\\ &= \sum_{i=1}^{n}(f*g)(i)-\sum_{i=2}^{n} g(i)S\left(\left\lfloor\frac{n}{i}\right\rfloor\right)\end{aligned}$$

假如我们可以构造恰当的数论函数 g 使得：

1. 可以快速计算 $\sum_{i=1}^{n}(f*g)(i)$；
2. 可以快速计算 g 的前缀和，以用数论分块求解 $\sum_{i=2}^{n} g(i)S\left(\left\lfloor\frac{n}{i}\right\rfloor\right)$。

则我们可以在较短时间内求得 $g(1)S(n)$。

" 注意"

无论数论函数 f 是否为积性函数，只要可以构造出恰当的数论函数 g，便都可以考虑用杜教筛求 f 的前缀和。

如考虑 $f(n)=\mathrm{i}\varphi(n)$，显然 f 不是积性函数，但可取 $g(n)=1$，从而：

$$\sum_{k=1}^{n}(f*g)(k)=\mathrm{i}\frac{n(n+1)}{2}$$

计算 $\sum_{k\le m}(f*g)(k)$ 和 $\sum_{k\le m} g(k)$ 的时间复杂度均为 $O(1)$，故可以考虑使用杜教筛。

时间复杂度

令 $R(n)=\left\{\left\lfloor\frac{n}{k}\right\rfloor : k=2,3,\dots,n\right\}$, 我们有如下引理:

"引理"

对任意的 $m\in R(n)$，我们有 $R(m)\subseteq R(n)$.

"证明"

令 $m=\left\lfloor\frac{n}{x}\right\rfloor$, 任取 $\left\lfloor\frac{m}{y}\right\rfloor\in R(m)$, 由整除分块/数论分块的 引理 1 可知:

$$\left\lfloor\frac{m}{y}\right\rfloor=\left\lfloor\frac{n}{xy}\right\rfloor\in R(n).$$

设计算 $\sum_{i=1}^{n}(f*g)(i)$ 和 $\sum_{i=1}^{n}g(i)$ 的时间复杂度均为 $O(1)$. 由引理可知: 使用记忆化之后, 每个 $S(k)$ $(k\in R(n))$ 均只会计算一次。

由整除分块/数论分块的 引理 2 可知 $|R(n)|=2\sqrt{n}+\Theta(1)$. 设计算 $S(n)$ 的时间复杂度为 $T(n)$, 则:

$$\begin{aligned}T(n)&=\sum_{k\in R(n)}T(k)\\&=\Theta(\sqrt{n})+\sum_{k=1}^{\lfloor\sqrt{n}\rfloor}O(\sqrt{k})+\sum_{k=2}^{\lfloor\sqrt{n}\rfloor}O\left(\sqrt{\frac{n}{k}}\right)\\&=O\left(\int_0^{\sqrt{n}}\left(\sqrt{x}+\sqrt{\frac{n}{x}}\right)\mathrm{d}x\right)\\&=O\left(n^{3/4}\right).\end{aligned}$$

若我们可以预处理出一部分 $S(k)$, 其中 $k=1,2,\dots,m$， $m\ge\lfloor\sqrt{n}\rfloor$. 设预处理的时间复杂度为 $T_0(m)$, 则此时的 $T(n)$ 为:

$$\begin{aligned}T(n)&=T_0(m)+\sum_{k\in R(n);k>m}T(k)\\&=T_0(m)+\sum_{k=1}^{\lfloor n/m\rfloor}O\left(\sqrt{\frac{n}{k}}\right)\\&=O\left(T_0(m)+\int_0^{n/m}\sqrt{\frac{n}{x}}\mathrm{d}x\right)\\&=O\left(T_0(m)+\frac{n}{\sqrt{m}}\right).\end{aligned}$$

若 $T_0(m)=O(m)$（如线性筛），由均值不等式可知：当 $m=\Theta\left(n^{2/3}\right)$ 时，$T(n)$ 取得最小值 $O\left(n^{2/3}\right)$.

"伪证一例"

设计算 $S(n)$ 的复杂度为 $T(n)$, 则有:

$$T(n)=\Theta\left(\sqrt{n}\right)+O\left(\sum_{i=2}^{\lfloor\sqrt{n}\rfloor}T\left(\left\lfloor\frac{n}{i}\right\rfloor\right)\right)$$

$$\begin{aligned}T\left(\left\lfloor\frac{n}{i}\right\rfloor\right)&=\Theta\left(\sqrt{\frac{n}{i}}\right)+O\left(\sum_{j=2}^{\lfloor\sqrt{n/i}\rfloor}T\left(\left\lfloor\frac{n}{ij}\right\rfloor\right)\right)\\&=O\left(\sqrt{\frac{n}{i}}\right)\end{aligned}$$

note

$O\left(\sum_{j=2}^{\lfloor\sqrt{n/i}\rfloor} T\left(\left\lfloor\frac{n}{ij}\right\rfloor\right)\right)$ 视作高阶无穷小，从而可以舍去。

故：

$$\begin{aligned}
T(n) &= \Theta\left(\sqrt{n}\right) + O\left(\sum_{i=2}^{\lfloor\sqrt{n}\rfloor}\sqrt{\frac{n}{i}}\right) \\
&= O\left(\sum_{i=1}^{\lfloor\sqrt{n}\rfloor}\sqrt{\frac{n}{i}}\right) \\
&= O\left(\int_0^{\sqrt{n}}\sqrt{\frac{n}{x}}\mathrm{d}x\right) \\
&= O\left(n^{3/4}\right)
\end{aligned}$$

bug

问题在于「视作高阶无穷小，从而可以舍去」这一处。我们将 $T\left(\left\lfloor\frac{n}{i}\right\rfloor\right)$ 代入 $T(n)$ 的式子里，有：

$$\begin{aligned}
T(n) &= \Theta\left(\sqrt{n}\right) + O\left(\sum_{i=2}^{\lfloor\sqrt{n}\rfloor}\sqrt{\frac{n}{i}}\right) + O\left(\sum_{i=2}^{\lfloor\sqrt{n}\rfloor}\sum_{j=2}^{\lfloor\sqrt{n/i}\rfloor} T\left(\left\lfloor\frac{n}{ij}\right\rfloor\right)\right) \\
&= O\left(\sqrt{n} + \int_0^{\sqrt{n}}\sqrt{\frac{n}{x}}\mathrm{d}x\right) + O\left(\sum_{i=2}^{\lfloor\sqrt{n}\rfloor}\sum_{j=2}^{\lfloor\sqrt{n/i}\rfloor} T\left(\left\lfloor\frac{n}{ij}\right\rfloor\right)\right) \\
&= O\left(n^{3/4}\right) + O\left(\sum_{i=2}^{\lfloor\sqrt{n}\rfloor}\sum_{j=2}^{\lfloor\sqrt{n/i}\rfloor} T\left(\left\lfloor\frac{n}{ij}\right\rfloor\right)\right)
\end{aligned}$$

我们考虑 $\displaystyle\sum_{i=2}^{\lfloor\sqrt{n}\rfloor}\sum_{j=2}^{\lfloor\sqrt{n/i}\rfloor} T\left(\left\lfloor\frac{n}{ij}\right\rfloor\right)$ 这部分，不难发现：

$$\begin{aligned}
\sum_{i=2}^{\lfloor\sqrt{n}\rfloor}\sum_{j=2}^{\lfloor\sqrt{n/i}\rfloor} T\left(\left\lfloor\frac{n}{ij}\right\rfloor\right) &= \Omega\left(\sum_{i=2}^{\lfloor\sqrt{n}\rfloor} T\left(\left\lfloor\frac{n}{i}\cdot\left\lfloor\sqrt{\frac{n}{i}}\right\rfloor^{-1}\right\rfloor\right)\right) \\
&= \Omega\left(\sum_{i=2}^{\lfloor\sqrt{n}\rfloor} T\left(\left\lfloor\sqrt{\frac{n}{i}}\right\rfloor\right)\right)
\end{aligned}$$

由于没有引入记忆化，因此上式中的 $T\left(\left\lfloor\sqrt{\frac{n}{i}}\right\rfloor\right)$ 仍然是 $\Omega\left(\left(\frac{n}{i}\right)^{1/4}\right)$ 的，进而所谓的「高阶无穷小」部分是不可以舍去的。

实际上杜教筛的亚线性时间复杂度是由记忆化保证的。只有使用了记忆化之后才能保证不会出现那个多重求和的项。

例题

问题一

"P4213【模板】杜教筛（Sum）[1]"

求 $S_1(n) = \sum_{i=1}^{n}\mu(i)$ 和 $S_2(n) = \sum_{i=1}^{n}\varphi(i)$ 的值，$1 \leq n < 2^{31}$.

我们知道：

$$\epsilon = [n=1] = \mu * 1 = \sum_{d\mid n}\mu(d)$$

$$\begin{aligned} S_1(n) &= \sum_{i=1}^{n}\epsilon(i) - \sum_{i=2}^{n} S_1\left(\left\lfloor \frac{n}{i} \right\rfloor\right) \\ &= 1 - \sum_{i=2}^{n} S_1\left(\left\lfloor \frac{n}{i} \right\rfloor\right) \end{aligned}$$

时间复杂度的推导见时间复杂度一节。

对于较大的值，需要用 `map/unordered_map` 存下其对应的值，方便以后使用时直接使用之前计算的结果。

当然也可以用杜教筛求出 $\varphi(x)$ 的前缀和，但是更好的方法是应用莫比乌斯反演。

$$\begin{aligned} \sum_{i=1}^{n}\sum_{j=1}^{n}[\gcd(i,j)=1] &= \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{d\mid i,d\mid j}\mu(d) \\ &= \sum_{d=1}^{n}\mu(d)\left\lfloor \frac{n}{d} \right\rfloor^2 \end{aligned}$$

由于题目所求的是 $\sum_{i=1}^{n}\sum_{j=1}^{i}[\gcd(i,j)=1]$, 所以我们排除掉 $i=1,j=1$ 的情况，并将结果除以 2 即可。

观察到，只需求出莫比乌斯函数的前缀和，就可以快速计算出欧拉函数的前缀和了。时间复杂度 $O\left(n^{\frac{2}{3}}\right)$.

求 $S(n)=\sum_{i=1}^{n}\varphi(i)$.

同样的，$\varphi * 1 = \mathrm{id}$, 从而：

$$\begin{aligned} S(n) &= \sum_{i=1}^{n} i - \sum_{i=2}^{n} S\left(\left\lfloor \frac{n}{i} \right\rfloor\right) \\ &= \frac{1}{2}n(n+1) - \sum_{i=2}^{n} S\left(\left\lfloor \frac{n}{i} \right\rfloor\right) \end{aligned}$$

" 代码实现"

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <map>
using namespace std;
const int maxn = 2000010;
long long T, n, pri[maxn], cur, mu[maxn], sum_mu[maxn];
bool vis[maxn];
map<long long, long long> mp_mu;

long long S_mu(long long x) {  // 求 mu 的前缀和
  if (x < maxn) return sum_mu[x];
  if (mp_mu[x]) return mp_mu[x];  // 如果 map 中已有该大小的 mu 值，则可直接返回
  long long ret = (long long)1;
  for (long long i = 2, j; i <= x; i = j + 1) {
    j = x / (x / i);
    ret -= S_mu(x / i) * (j - i + 1);
  }
  return mp_mu[x] = ret;  // 路径压缩，方便下次计算
}

long long S_phi(long long x) {  // 求 phi 的前缀和
  long long ret = (long long)0;
  long long j;
  for (long long i = 1; i <= x; i = j + 1) {
    j = x / (x / i);
    ret += (S_mu(j) - S_mu(i - 1)) * (x / i) * (x / i);
```

```
  }
  return (ret - 1) / 2 + 1;
}

int main() {
  scanf("%lld", &T);
  mu[1] = 1;
  for (int i = 2; i < maxn; i++) {  // 线性筛预处理 mu 数组
    if (!vis[i]) {
      pri[++cur] = i;
      mu[i] = -1;
    }
    for (int j = 1; j <= cur && i * pri[j] < maxn; j++) {
      vis[i * pri[j]] = true;
      if (i % pri[j])
        mu[i * pri[j]] = -mu[i];
      else {
        mu[i * pri[j]] = 0;
        break;
      }
    }
  }
  for (int i = 1; i < maxn; i++)
    sum_mu[i] = sum_mu[i - 1] + mu[i];  // 求 mu 数组前缀和
  while (T--) {
    scanf("%lld", &n);
    printf("%lld %lld\n", S_phi(n), S_mu(n));
  }
  return 0;
}
```

问题二

"「LuoguP3768」 简单的数学题[2]"

大意：求

$$\sum_{i=1}^{n}\sum_{j=1}^{n} i\cdot j\cdot \gcd(i,j) \pmod p$$

其中 $n\le 10^{10}, 5\times 10^8\le p\le 1.1\times 10^9$,$p$ 是质数。

利用 $\varphi * 1 = \mathrm{id}$ 做莫比乌斯反演化为：

$$\sum_{d=1}^{n} F^2\left(\left\lfloor \frac{n}{d}\right\rfloor\right)\cdot d^2\varphi(d)$$

其中 $F(n)=\dfrac{1}{2}n(n+1)$

对 $\sum_{d=1}^{n} F\left(\left\lfloor \frac{n}{d}\right\rfloor\right)^2$ 做数论分块，$d^2\varphi(d)$ 的前缀和用杜教筛处理：

$$f(n)=n^2\varphi(n)=(\mathrm{id}^2\,\varphi)(n)$$

$$S(n)=\sum_{i=1}^{n} f(i)=\sum_{i=1}^{n}(\mathrm{id}^2\,\varphi)(i)$$

需要构造积性函数 g，使得 $f\times g$ 和 g 能快速求和。

单纯的 φ 的前缀和可以用 $\varphi * 1$ 的杜教筛处理，但是这里的 f 多了一个 id^2，那么我们就卷一个 id^2 上去，让它变成常数：

$$S(n) = \sum_{i=1}^{n} \left(\left(\mathrm{id}^2 \varphi \right) * \mathrm{id}^2 \right)(i) - \sum_{i=2}^{n} \mathrm{id}^2(i) S\left(\left\lfloor \frac{n}{i} \right\rfloor \right)$$

化一下卷积：

$$\begin{aligned}
((\mathrm{id}^2 \varphi) * \mathrm{id}^2)(i) &= \sum_{d|i} \left(\mathrm{id}^2 \varphi \right)(d)\, \mathrm{id}^2 \left(\frac{i}{d} \right) \\
&= \sum_{d|i} d^2 \varphi(d) \left(\frac{i}{d} \right)^2 \\
&= \sum_{d|i} i^2 \varphi(d) = i^2 \sum_{d|i} \varphi(d) \\
&= i^2 (\varphi * 1)(i) = i^3
\end{aligned}$$

再化一下 $S(n)$：

$$\begin{aligned}
S(n) &= \sum_{i=1}^{n} \left((\mathrm{id}^2 \varphi) * \mathrm{id}^2 \right)(i) - \sum_{i=2}^{n} \mathrm{id}^2(i) S\left(\left\lfloor \frac{n}{i} \right\rfloor \right) \\
&= \sum_{i=1}^{n} i^3 - \sum_{i=2}^{n} i^2 S\left(\left\lfloor \frac{n}{i} \right\rfloor \right) \\
&= \left(\frac{1}{2} n(n+1) \right)^2 - \sum_{i=2}^{n} i^2 S\left(\left\lfloor \frac{n}{i} \right\rfloor \right)
\end{aligned}$$

分块求解即可。

" 代码实现"

```
#include <cmath>
#include <cstdio>
#include <map>
using namespace std;
const int N = 5e6, NP = 5e6, SZ = N;
long long n, P, inv2, inv6, s[N];
int phi[N], p[NP], cnt, pn;
bool bp[N];
map<long long, long long> s_map;

long long ksm(long long a, long long m) {  // 求逆元用
  long long res = 1;
  while (m) {
    if (m & 1) res = res * a % P;
    a = a * a % P, m >>= 1;
  }
  return res;
}

void prime_work(int k) {  // 线性筛 phi, s
  bp[0] = bp[1] = 1, phi[1] = 1;
  for (int i = 2; i <= k; i++) {
    if (!bp[i]) p[++cnt] = i, phi[i] = i - 1;
    for (int j = 1; j <= cnt && i * p[j] <= k; j++) {
      bp[i * p[j]] = 1;
      if (i % p[j] == 0) {
        phi[i * p[j]] = phi[i] * p[j];
        break;
```

```
      } else
        phi[i * p[j]] = phi[i] * phi[p[j]];
    }
  }
  for (int i = 1; i <= k; i++)
    s[i] = (1ll * i * i % P * phi[i] % P + s[i - 1]) % P;
}

long long s3(long long k) {  // 立方和
  return k %= P, (k * (k + 1) / 2) % P * ((k * (k + 1) / 2) % P) % P;
}

long long s2(long long k) {  // 平方和
  return k %= P, k * (k + 1) % P * (k * 2 + 1) % P * inv6 % P;
}

long long calc(long long k) {  // 计算 S(k)
  if (k <= pn) return s[k];
  if (s_map[k]) return s_map[k];  // 对于超过 pn 的用 map 离散存储
  long long res = s3(k), pre = 1, cur;
  for (long long i = 2, j; i <= k; i = j + 1)
    j = k / (k / i), cur = s2(j),
    res = (res - calc(k / i) * (cur - pre) % P) % P, pre = cur;
  return s_map[k] = (res + P) % P;
}

long long solve() {
  long long res = 0, pre = 0, cur;
  for (long long i = 1, j; i <= n; i = j + 1) {
    j = n / (n / i);
    cur = calc(j);
    res = (res + (s3(n / i) * (cur - pre)) % P) % P;
    pre = cur;
  }
  return (res + P) % P;
}

int main() {
  scanf("%lld%lld", &P, &n);
  inv2 = ksm(2, P - 2), inv6 = ksm(6, P - 2);
  pn = (long long)pow(n, 0.666667);  // n^(2/3)
  prime_work(pn);
  printf("%lld", solve());
  return 0;
}  // 不要为了省什么内存把数组开小，会卡 80
```

参考资料

1. 任之洲，2016，《积性函数求和的几种方法》，2016 年信息学奥林匹克中国国家队候选队员论文
2. 杜教筛的时空复杂度分析 - riteme.site[3]

参考资料与注释

[1] P4213【模板】杜教筛（Sum）

[2] 「LuoguP3768」简单的数学题

[3] 杜教筛的时空复杂度分析 - riteme.site

9.12.25 Powerful Number 筛

定义

Powerful Number（以下简称 PN）筛类似于杜教筛，或者说是杜教筛的一个扩展，可以拿来求一些积性函数的前缀和。

要求：

- 存在一个函数 g 满足：
  - g 是积性函数。
  - g 易求前缀和。
  - 对于质数 p，$g(p)=f(p)$。

假设现在要求积性函数 f 的前缀和 $F(n)=\sum_{i=1}^{n} f(i)$。

Powerful Number

定义：对于正整数 n，记 n 的质因数分解为 $n=\prod_{i=1}^{m} p_i^{e_i}$。$n$ 是 PN 当且仅当 $\forall 1\le i\le m, e_i>1$。

性质 1：所有 PN 都可以表示成 a^2b^3 的形式。

证明：若 e_i 是偶数，则将 $p_i^{e_i}$ 合并进 a^2 里；若 e_i 为奇数，则先将 p_i^3 合并进 b^3 里，再将 $p_i^{e_i-3}$ 合并进 a^2 里。

性质 2：n 以内的 PN 至多有 $O(\sqrt{n})$ 个。

证明：考虑枚举 a，再考虑满足条件的 b 的个数，有 PN 的个数约等于

$$\int_1^{\sqrt{n}} \sqrt[3]{\frac{n}{x^2}}\mathrm{d}x=O(\sqrt{n})$$

那么如何求出 n 以内所有的 PN 呢？线性筛找出 $\sqrt{n}$ 内的所有素数，再 DFS 搜索各素数的指数即可。由于 n 以内的 PN 至多有 $O(\sqrt{n})$ 个，所以至多搜索 $O(\sqrt{n})$ 次。

PN 筛

首先，构造出一个易求前缀和的积性函数 g，且满足对于素数 p，$g(p)=f(p)$。记 $G(n)=\sum_{i=1}^{n} g(i)$。

然后，构造函数 $h=f/g$，这里的 $/$ 表示狄利克雷卷积除法。根据狄利克雷卷积的性质可以得知 h 也为积性函数，因此 $h(1)=1$。$f=g*h$，这里 $*$ 表示狄利克雷卷积。

对于素数 p，$f(p)=g(1)h(p)+g(p)h(1)=h(p)+g(p)\implies h(p)=0$。根据 $h(p)=0$ 和 h 是积性函数可以推出对于非 PN 的数 n 有 $h(n)=0$，即 h 仅在 PN 处取有效值。

现在，根据 $f=g*h$ 有

$$
\begin{aligned}
F(n) &= \sum_{i=1}^{n} f(i) \\
&= \sum_{i=1}^{n} \sum_{d \mid i} h(d) g\left(\frac{i}{d}\right) \\
&= \sum_{d=1}^{n} \sum_{i=1}^{\lfloor \frac{n}{d} \rfloor} h(d) g(i) \\
&= \sum_{d=1}^{n} h(d) \sum_{i=1}^{\lfloor \frac{n}{d} \rfloor} g(i) \\
&= \sum_{d=1}^{n} h(d) G\left(\left\lfloor \frac{n}{d} \right\rfloor\right) \\
&= \sum_{\substack{d=1 \\ d \text{ is PN}}}^{n} h(d) G\left(\left\lfloor \frac{n}{d} \right\rfloor\right)
\end{aligned}
$$

$O(\sqrt{n})$ 找出所有 PN，计算出所有 h 的有效值。对于 h 有效值的计算，只需要计算出所有 $h(p^c)$ 处的值，就可以根据 h 为积性函数推出 h 的所有有效值。现在对于每一个有效值 d，计算 $h(d)G\left(\left\lfloor \frac{n}{d} \right\rfloor\right)$ 并累加即可得到 $F(n)$。

下面考虑计算 $h(p^c)$，一共有两种方法：一种是直接推出 $h(p^c)$ 仅与 p,c 有关的计算公式，再根据公式计算 $h(p^c)$；另一种是根据 $f=g*h$ 有 $f(p^c)=\sum_{i=0}^{c} g(p^i)h(p^{c-i})$，移项可得 $h(p^c)=f(p^c)-\sum_{i=1}^{c} g(p^i)h(p^{c-i})$，现在就可以枚举素数 p 再枚举指数 c 求解出所有 $h(p^c)$。

过程

1. 构造 g
2. 构造快速计算 G 的方法
3. 计算 $h(p^c)$
4. 搜索 PN，过程中累加答案
5. 得到结果

对于第 3 步，可以直接根据公式计算，可以使用枚举法预处理打表，也可以搜索到了再临时推。

性质

以使用第二种方法计算 $h(p^c)$ 为例进行分析。可以分为计算 $h(p^c)$ 和搜索两部分进行分析。

对于第一部分，根据 $O(\sqrt{n})$ 内的素数个数为 $O\left(\frac{\sqrt{n}}{\log n}\right)$，每个素数 p 的指数 c 至多为 $\log n$，计算 $h(p^c)$ 需要循环 $(c-1)$ 次，由此有第一部分的时间复杂度为 $O\left(\frac{\sqrt{n}}{\log n} \cdot \log n \cdot \log n\right)=O(\sqrt{n}\log n)$，且这是一个宽松的上界。根据题目的不同还可以添加不同的优化，从而降低第一部分的时间复杂度。

对于搜索部分，由于 n 以内的 PN 至多有 $O(\sqrt{n})$ 个，所以至多搜索 $O(\sqrt{n})$ 次。对于每一个 PN，根据计算 G 的方法不同，时间复杂度也不同。例如，假设计算 $G\left(\left\lfloor \frac{n}{d} \right\rfloor\right)$ 的时间复杂度为 $O(1)$，则第二部分的复杂度为 $O(\sqrt{n})$。

特别地，若借助杜教筛计算 $G\left(\left\lfloor \frac{n}{d} \right\rfloor\right)$，则第二部分的时间复杂度为杜教筛的时间复杂度，即 $O(n^{\frac{2}{3}})$。因为若事先计算一次 $G(n)$，并且预先使用线性筛优化和用支持快速随机访问的数据结构（如 C++ 中的 `std::map` 和 `std::unordered_map`）记录较大的值，则杜教筛过程中用到的 $G\left(\left\lfloor \frac{n}{d} \right\rfloor\right)$ 都是线性筛中记录的或者 `std::map` 中记录的，这一点可以直接用程序验证。

对于空间复杂度，其瓶颈在于存储 $h(p^c)$。若使用二维数组 a 记录，$a_{i,j}$ 表示 $h(p_i^j)$ 的值，则空间复杂度为 $O\left(\frac{\sqrt{n}}{\log n} \cdot \log n\right)=O(\sqrt{n})$。

例题

Luogu P5325【模板】Min_25 筛[1]

题意：给定积性函数 $f(p^k)=p^k(p^k-1)$，求 $\sum_{i=1}^{n} f(i)$。

易得 $f(p)=p(p-1)=\mathrm{id}(p)\varphi(p)$，构造 $g(n)=\mathrm{id}(n)\varphi(n)$。

考虑使用杜教筛求 $G(n)$，根据 $(\mathrm{id}\cdot\varphi)*\mathrm{id}=\mathrm{id}_2$ 可得 $G(n)=\sum_{i=1}^{n} i^2-\sum_{i=2}^{n} d\cdot G\left(\left\lfloor\frac{n}{d}\right\rfloor\right)$。

之后 $h(p^k)$ 的取值可以枚举计算，这种方法不再赘述。

此外，此题还可以直接求出 $h(p^k)$ 仅与 p,k 有关的公式，过程如下：

$$
\begin{aligned}
& f(p^k)=\sum_{i=0}^{k} g(p^{k-i})h(p^i) \\
\iff & p^k(p^k-1)=\sum_{i=0}^{k} p^{k-i}\varphi(p^{k-i})h(p^i) \\
\iff & p^k(p^k-1)=\sum_{i=0}^{k} p^{2k-2i-1}(p-1)h(p^i) \\
\iff & p^k(p^k-1)=h(p^k)+\sum_{i=0}^{k-1} p^{2k-2i-1}(p-1)h(p^i) \\
\iff & h(p^k)=p^k(p^k-1)-\sum_{i=0}^{k-1} p^{2k-2i-1}(p-1)h(p^i) \\
\iff & h(p^k)-p^2h(p^{k-1})=p^k(p^k-1)-p^{k+1}(p^{k-1}-1)-p(p-1)h(p^{k-1}) \\
\iff & h(p^k)-ph(p^{k-1})=p^{k+1}-p^k \\
\iff & \frac{h(p^k)}{p^k}-\frac{h(p^{k-1})}{p^{k-1}}=p-1
\end{aligned}
$$

再根据 $h(p)=0$，通过累加法即可推出 $h(p^k)=(k-1)(p-1)p^k$。

" 参考代码"

```cpp
#include <bits/stdc++.h>
using namespace std;

const int MOD = 1e9 + 7;

template <typename T>
int mint(T x) {
  x %= MOD;
  if (x < 0) x += MOD;
  return x;
}

int add(int x, int y) { return x + y >= MOD ? x + y - MOD : x + y; }

int mul(int x, int y) { return (long long)1 * x * y % MOD; }

int sub(int x, int y) { return x < y ? x - y + MOD : x - y; }  // 防止负数

int qp(int x, int y) {
  int r = 1;
  for (; y; y >>= 1) {
    if (y & 1) r = mul(r, x);
```

```
    x = mul(x, x);
  }
  return r;
}

int inv(int x) { return qp(x, MOD - 2); }

namespace PNS {
const int N = 2e6 + 5;
const int M = 35;

long long global_n;

int g[N], sg[N];

int h[N][M];
bool vis_h[N][M];

int ans;

int pcnt, prime[N], phi[N];
bool isp[N];

void sieve(int n) {
  pcnt = 0;
  for (int i = 2; i <= n; ++i) isp[i] = true;  // 判断质数数组
  phi[1] = 1;
  for (int i = 2; i <= n; ++i) {
    if (isp[i]) {
      ++pcnt;
      prime[pcnt] = i;
      phi[i] = i - 1;
    }
    for (int j = 1; j <= pcnt; ++j) {  // 筛去非质数
      long long nxt = (long long)1 * i * prime[j];
      if (nxt > n) break;
      isp[nxt] = false;
      if (i % prime[j] == 0) {  // i 是非质数的情况
        phi[nxt] = phi[i] * prime[j];
        break;
      }
      phi[nxt] = phi[i] * phi[prime[j]];
    }
  }

  for (int i = 1; i <= n; ++i) g[i] = mul(i, phi[i]);

  sg[0] = 0;
  for (int i = 1; i <= n; ++i) sg[i] = add(sg[i - 1], g[i]);  // g 函数的前缀和
}

int inv2, inv6;

void init() {
```

```
  sieve(N - 1);
  for (int i = 1; i <= pcnt; ++i) h[i][0] = 1, h[i][1] = 0;
  for (int i = 1; i <= pcnt; ++i) vis_h[i][0] = vis_h[i][1] = true;
  inv2 = inv(2);
  inv6 = inv(6);
}

int S1(long long n) { return mul(mul(mint(n), mint(n + 1)), inv2); }

int S2(long long n) {
  return mul(mul(mint(n), mul(mint(n + 1), mint(n * 2 + 1))), inv6);
}

map<long long, int> mp_g;

int G(long long n) {
  if (n < N) return sg[n];
  if (mp_g.count(n)) return mp_g[n];

  int ret = S2(n);
  for (long long i = 2, j; i <= n; i = j + 1) {
    j = n / (n / i);
    ret = sub(ret, mul(sub(S1(j), S1(i - 1)), G(n / i)));
  }
  mp_g[n] = ret;
  return ret;
}

void dfs(long long d, int hd, int pid) {
  ans = add(ans, mul(hd, G(global_n / d)));

  for (int i = pid, p; i <= pcnt; ++i) {
    if (i > 1 && d > global_n / prime[i] / prime[i]) break;  // 剪枝

    int c = 2;
    for (long long x = d * prime[i] * prime[i]; x <= global_n;
         x *= prime[i], ++c) {  // 计算 f.g 函数
      if (!vis_h[i][c]) {
        int f = qp(prime[i], c);
        f = mul(f, sub(f, 1));
        int g = mul(prime[i], prime[i] - 1);
        int t = mul(prime[i], prime[i]);

        for (int j = 1; j <= c; ++j) {
          f = sub(f, mul(g, h[i][c - j]));
          g = mul(g, t);
        }
        h[i][c] = f;
        vis_h[i][c] = true;
      }

      if (h[i][c]) dfs(x, mul(hd, h[i][c]), i + 1);
    }
  }
```

```
}

int solve(long long n) {
  global_n = n;
  ans = 0;
  dfs(1, 1, 1);
  return ans;
}
}  // namespace PNS

int main() {
  PNS::init();
  long long n;
  scanf("%lld", &n);
  printf("%d\n", PNS::solve(n));
  return 0;
}
```

「LOJ #6053」 简单的函数[2]

给定 $f(n)$:

$$f(n)=\begin{cases}1 & n=1\\ p\oplus c & n=p^c\\ f(a)f(b) & n=ab \text{ and } a\perp b\end{cases}$$

易得:

$$f(p)=\begin{cases}p+1 & p=2\\ p-1 & \text{otherwise}\end{cases}$$

构造 g 为

$$g(n)=\begin{cases}3\varphi(n) & 2\mid n\\ \varphi(n) & \text{otherwise}\end{cases}$$

易证 $g(p)=f(p)$ 且 g 为积性函数。

下面考虑求 $G(n)$。

$$\begin{aligned}G(n)&=\sum_{i=1}^{n}[i \bmod 2=1]\varphi(i)+3\sum_{i=1}^{n}[i \bmod 2=0]\varphi(i)\\&=\sum_{i=1}^{n}\varphi(i)+2\sum_{i=1}^{n}[i \bmod 2=0]\varphi(i)\\&=\sum_{i=1}^{n}\varphi(i)+2\sum_{i=1}^{\lfloor\frac{n}{2}\rfloor}\varphi(2i)\end{aligned}$$

记 $S_1(n)=\sum_{i=1}^{n}\varphi(i)$, $S_2(n)=\sum_{i=1}^{n}\varphi(2i)$, 则 $G(n)=S_1(n)+2S_2\left(\left\lfloor\frac{n}{2}\right\rfloor\right)$。

当 $2 \mid n$ 时，有

$$\begin{aligned}
S_2(n) &= \sum_{i=1}^{n} \varphi(2i) \\
&= \sum_{i=1}^{\frac{n}{2}} (\varphi(2(2i-1)) + \varphi(2(2i))) \\
&= \sum_{i=1}^{\frac{n}{2}} (\varphi(2i-1) + 2\varphi(2i)) \\
&= \sum_{i=1}^{\frac{n}{2}} (\varphi(2i-1) + \varphi(2i)) + \sum_{i=1}^{\frac{n}{2}} \varphi(2i) \\
&= \sum_{i=1}^{n} \varphi(i) + S_2\left(\frac{n}{2}\right) \\
&= S_1(n) + S_2\left(\left\lfloor\frac{n}{2}\right\rfloor\right)
\end{aligned}$$

当 $2 \nmid n$ 时，有

$$\begin{aligned}
S_2(n) &= S_2(n-1) + \varphi(2n) \\
&= S_2(n-1) + \varphi(n) \\
&= \sum_{i=1}^{n-1} \varphi(i) + S_2\left(\frac{n-1}{2}\right) + \varphi(n) \\
&= S_1(n) + S_2\left(\left\lfloor\frac{n}{2}\right\rfloor\right)
\end{aligned}$$

综上，有 $S_2(n) = S_1(n) + S_2\left(\left\lfloor\frac{n}{2}\right\rfloor\right)$。

S_1 可以用杜教筛求，S_2 直接按照公式推，这样 G 也可以求出来了。

" 参考代码"

```cpp
#include <bits/stdc++.h>
using namespace std;

const int MOD = 1e9 + 7;
const int inv2 = (MOD + 1) / 2;

template <typename T>
int mint(T x) {
  x %= MOD;
  if (x < 0) x += MOD;
  return x;
}

int add(int x, int y) {
  return x + y >= MOD ? x + y - MOD : x + y;
}  // 防止大于模数

int mul(int x, int y) { return (long long)1 * x * y % MOD; }

int sub(int x, int y) { return x < y ? x - y + MOD : x - y; }  // 防负数

namespace PNS {
const int N = 2e6 + 5;
const int M = 35;
```

```
long long global_n;

int s1[N], s2[N];

int h[N][M];
bool vis_h[N][M];

int ans;

int pcnt, prime[N], phi[N];
bool isp[N];

void sieve(int n) {
  pcnt = 0;
  for (int i = 2; i <= n; ++i) isp[i] = true;  // 判断质数数组
  phi[1] = 1;
  for (int i = 2; i <= n; ++i) {
    if (isp[i]) {
      ++pcnt;
      prime[pcnt] = i;
      phi[i] = i - 1;
    }
    for (int j = 1; j <= pcnt; ++j) {  // 筛去非质数
      long long nxt = (long long)1 * i * prime[j];
      if (nxt > n) break;
      isp[nxt] = false;
      if (i % prime[j] == 0) {  // i 是非质数的情况
        phi[nxt] = phi[i] * prime[j];
        break;
      }
      phi[nxt] = phi[i] * phi[prime[j]];
    }
  }

  s1[0] = 0;
  for (int i = 1; i <= n; ++i) s1[i] = add(s1[i - 1], phi[i]);

  s2[0] = 0;
  for (int i = 1; i <= n / 2; ++i) {
    s2[i] = add(s2[i - 1], phi[2 * i]);
  }
}

void init() {
  sieve(N - 1);
  for (int i = 1; i <= pcnt; ++i) h[i][0] = 1;
  for (int i = 1; i <= pcnt; ++i) vis_h[i][0] = true;
}

map<long long, int> mp_s1;

int S1(long long n) {
  if (n < N) return s1[n];
  if (mp_s1.count(n)) return mp_s1[n];
```

```
  int ret = mul(mul(mint(n), mint(n + 1)), inv2);
  for (long long i = 2, j; i <= n; i = j + 1) {
    j = n / (n / i);
    ret = sub(ret, mul(mint(j - i + 1), S1(n / i)));
  }
  mp_s1[n] = ret;
  return ret;
}

map<long long, int> mp_s2;

int S2(long long n) {
  if (n < N / 2) return s2[n];
  if (mp_s2.count(n)) return mp_s2[n];
  int ret = add(S1(n), S2(n / 2));
  mp_s2[n] = ret;
  return ret;
}

int G(long long n) { return add(S1(n), mul(2, S2(n / 2))); }

void dfs(long long d, int hd, int pid) {
  ans = add(ans, mul(hd, G(global_n / d)));

  for (int i = pid, p; i <= pcnt; ++i) {
    if (i > 1 && d > global_n / prime[i] / prime[i]) break;  // 剪枝

    int c = 2;
    for (long long x = d * prime[i] * prime[i]; x <= global_n;
         x *= prime[i], ++c) {
      if (!vis_h[i][c]) {
        int f = prime[i] ^ c, g = prime[i] - 1;

        // p = 2 时特判一下
        if (i == 1) g = mul(g, 3);

        for (int j = 1; j <= c; ++j) {
          f = sub(f, mul(g, h[i][c - j]));
          g = mul(g, prime[i]);
        }
        h[i][c] = f;
        vis_h[i][c] = true;
      }

      if (h[i][c]) dfs(x, mul(hd, h[i][c]), i + 1);
    }
  }
}

int solve(long long n) {
  global_n = n;
  ans = 0;
  dfs(1, 1, 1);
```

```
  return ans;
}
}  // namespace PNS

int main() {
  PNS::init();  // 预处理函数
  long long n;
  scanf("%lld", &n);
  printf("%d\n", PNS::solve(n));
  return 0;
}
```

习题

- PE708 Twos are all you need[3]
- PE639 Summing a multiplicative function[4]
- PE484 Arithmetic Derivative[5]

参考资料

- 破壁人五号 - Powerful number 筛略解[6]
- command_block - 杜教筛（+ 贝尔级数 + powerful number）[7]

参考资料与注释

[1] Luogu P5325【模板】Min_25 筛

[2] 「LOJ #6053」简单的函数

[3] PE708 Twos are all you need

[4] PE639 Summing a multiplicative function

[5] PE484 Arithmetic Derivative

[6] 破壁人五号 - Powerful number 筛略解

[7] command_block - 杜教筛（+ 贝尔级数 + powerful number）

9.12.26 Min_25 筛

Authors: Marcythm, Xeonacid

定义

从此种筛法的思想方法来说，其又被称为「Extended Eratosthenes Sieve」。

由于其由 Min_25[1] 发明并最早开始使用，故称「Min_25 筛」。

性质

其可以在 $O\left(\frac{n^{\frac{3}{4}}}{\log n}\right)$ 或 $\Theta\left(n^{1-\epsilon}\right)$ 的时间复杂度下解决一类**积性函数**的前缀和问题。

要求：$f(p)$ 是一个关于 p 的项数较少的多项式或可以快速求值；$f(p^c)$ 可以快速求值。

记号

- **如无特别说明，本节中所有记为 p 的变量的取值集合均为全体质数**。
- $x/y := \left\lfloor \frac{x}{y} \right\rfloor$
- $\operatorname{isprime}(n) := [|\{d : d \mid n\}| = 2]$，即 n 为质数时其值为 1，否则为 0。
- p_k：全体质数中第 k 小的质数（如：$p_1 = 2, p_2 = 3$）。特别地，令 $p_0 = 1$。
- $\operatorname{lpf}(n) := [1 < n] \min\{p : p \mid n\} + [1 = n]$，即 n 的最小质因数。特别地，$n = 1$ 时，其值为 1。
- $F_{\text{prime}}(n) := \sum_{2 \le p \le n} f(p)$
- $F_k(n) := \sum_{i=2}^{n} [p_k \le \operatorname{lpf}(i)] f(i)$

解释

观察 $F_k(n)$ 的定义，可以发现答案即为 $F_1(n) + f(1) = F_1(n) + 1$。

考虑如何求出 $F_k(n)$。通过枚举每个 i 的最小质因子及其次数可以得到递推式：

$$
\begin{aligned}
F_k(n) &= \sum_{i=2}^{n} [p_k \le \operatorname{lpf}(i)] f(i) \\
&= \sum_{\substack{k \le i \\ p_i^2 \le n}} \sum_{\substack{c \ge 1 \\ p_i^c \le n}} f\left(p_i^c\right)\left([c > 1] + F_{i+1}\left(n/p_i^c\right)\right) + \sum_{\substack{k \le i \\ p_i \le n}} f(p_i) \\
&= \sum_{\substack{k \le i \\ p_i^2 \le n}} \sum_{\substack{c \ge 1 \\ p_i^c \le n}} f\left(p_i^c\right)\left([c > 1] + F_{i+1}\left(n/p_i^c\right)\right) + F_{\text{prime}}(n) - F_{\text{prime}}(p_{k-1}) \\
&= \sum_{\substack{k \le i \\ p_i^2 \le n}} \sum_{\substack{c \ge 1 \\ p_i^{c+1} \le n}} \left(f\left(p_i^c\right) F_{i+1}\left(n/p_i^c\right) + f\left(p_i^{c+1}\right)\right) + F_{\text{prime}}(n) - F_{\text{prime}}(p_{k-1})
\end{aligned}
$$

最后一步推导基于这样一个事实：对于满足 $p_i^c \le n < p_i^{c+1}$ 的 c，有 $p_i^{c+1} > n \iff n/p_i^c < p_i < p_{i+1}$，故 $F_{i+1}\left(n/p_i^c\right) = 0$。

其边界值即为 $F_k(n) = 0 (p_k > n)$。

假设现在已经求出了所有的 $F_{\text{prime}}(n)$，那么有两种方式可以求出所有的 $F_k(n)$：

1. 直接按照递推式计算。
2. 从大到小枚举 p 转移，仅当 $p^2 < n$ 时转移增加值不为零，故按照递推式后缀和优化即可。

现在考虑如何计算 $F_{\text{prime}}(n)$。

观察求 $F_k(n)$ 的过程，容易发现 F_{prime} 有且仅有 $1, 2, \dots, \lfloor\sqrt{n}\rfloor, n/\sqrt{n}, \dots, n/2, n$ 这 $O(\sqrt{n})$ 处的点值是有用的。

一般情况下，$f(p)$ 是一个关于 p 的低次多项式，可以表示为 $f(p) = \sum a_i p^{c_i}$。

那么对于每个 p^{c_i}，其对 $F_{\text{prime}}(n)$ 的贡献即为 $a_i \sum_{2 \le p \le n} p^{c_i}$。

分开考虑每个 p^{c_i} 的贡献，问题就转变为了：给定 $n, s, g(p) = p^s$，对所有的 $m = n/i$，求 $\sum_{p \le m} g(p)$。

> Notice：$g(p) = p^s$ 是完全积性函数！

于是设 $G_k(n) := \sum_{i=1}^{n} \left[p_k < \operatorname{lpf}(i) \vee \operatorname{isprime}(i)\right] g(i)$，即埃筛第 k 轮筛完后剩下的数的 g 值之和。

对于一个合数 x，必定有 $\operatorname{lpf}(x) \le \sqrt{x}$，则 $F_{\text{prime}} = G_{\lfloor\sqrt{n}\rfloor}$，故只需筛到 $G_{\lfloor\sqrt{n}\rfloor}$ 即可。

考虑 G 的边界值，显然为 $G_0(n)=\sum_{i=2}^{n} g(i)$。（还记得吗？特别约定了 $p_0=1$）

对于转移，考虑埃筛的过程，分开讨论每部分的贡献，有：

1. 对于 $n<p_k^2$ 的部分，G 值不变，即 $G_k(n)=G_{k-1}(n)$。
2. 对于 $p_k^2\le n$ 的部分，被筛掉的数必有质因子 p_k，即 $-g(p_k)G_{k-1}(n/p_k)$。
3. 对于第二部分，由于 $p_k^2\le n \iff p_k\le n/p_k$，故会有 $\operatorname{lpf}(i)<p_k$ 的 i 被减去。这部分应当加回来，即 $g(p_k)G_{k-1}(p_{k-1})$。

则有：

$$G_k(n)=G_{k-1}(n)-\left[p_k^2\le n\right]g(p_k)(G_{k-1}(n/p_k)-G_{k-1}(p_{k-1}))$$

复杂度分析

对于 $F_k(n)$ 的计算，其第一种方法的时间复杂度被证明为 $O\left(n^{1-\epsilon}\right)$（见 zzt 集训队论文 2.3）；

对于第二种方法，其本质即为洲阁筛的第二部分，在洲阁论文中也有提及（6.5.4），其时间复杂度被证明为 $O\left(\frac{n^{\frac{3}{4}}}{\log n}\right)$。

对于 $F_{\text{prime}}(n)$ 的计算，事实上，其实现与洲阁筛第一部分是相同的。

考虑对于每个 $m=n/i$，只有在枚举满足 $p_k^2\le m$ 的 p_k 转移时会对时间复杂度产生贡献，则时间复杂度可估计为：

$$\begin{aligned}
T(n)&=\sum_{i^2\le n}O\left(\pi\left(\sqrt{i}\right)\right)+\sum_{i^2\le n}O\left(\pi\left(\sqrt{\frac{n}{i}}\right)\right)\\
&=\sum_{i^2\le n}O\left(\frac{\sqrt{i}}{\ln\sqrt{i}}\right)+\sum_{i^2\le n}O\left(\frac{\sqrt{\frac{n}{i}}}{\ln\sqrt{\frac{n}{i}}}\right)\\
&=O\left(\int_1^{\sqrt{n}}\frac{\sqrt{\frac{n}{x}}}{\log\sqrt{\frac{n}{x}}}\mathrm{d}x\right)\\
&=O\left(\frac{n^{\frac{3}{4}}}{\log n}\right)
\end{aligned}$$

对于空间复杂度，可以发现不论是 F_k 还是 F_{prime}，其均只在 n/i 处取有效点值，共 $O(\sqrt{n})$ 个，仅记录有效值即可将空间复杂度优化至 $O(\sqrt{n})$。

首先，通过一次数论分块可以得到所有的有效值，用一个大小为 $O(\sqrt{n})$ 的数组 lis 记录。对于有效值 v，记 $\operatorname{id}(v)$ 为 v 在 lis 中的下标，易得：对于所有有效值 v，$\operatorname{id}(v)\le\sqrt{n}$。

然后分开考虑小于等于 $\sqrt{n}$ 的有效值和大于 $\sqrt{n}$ 的有效值：对于小于等于 $\sqrt{n}$ 的有效值 v，用一个数组 le 记录其 $\operatorname{id}(v)$，即 $\mathrm{le}_v=\operatorname{id}(v)$；对于大于 $\sqrt{n}$ 的有效值 v，用一个数组 ge 记录 $\operatorname{id}(v)$，由于 v 过大所以借助 $v'=n/v<\sqrt{n}$ 记录 $\operatorname{id}(v)$，即 $\mathrm{ge}_{v'}=\operatorname{id}(v)$。

这样，就可以使用两个大小为 $O(\sqrt{n})$ 的数组记录所有有效值的 id 并 $O(1)$ 查询。在计算 F_k 或 F_{prime} 时，使用有效值的 id 代替有效值作为下标，即可将空间复杂度优化至 $O(\sqrt{n})$。

过程

对于 $F_k(n)$ 的计算，我们实现时一般选择实现难度较低的第一种方法，其在数据规模较小时往往比第二种方法的表现要好；

对于 $F_{\text{prime}}(n)$ 的计算，直接按递推式实现即可。

对于 $p_k^2\le n$，可以用线性筛预处理出 $s_k:=F_{\text{prime}}(p_k)$ 来替代 F_k 递推式中的 $F_{\text{prime}}(p_{k-1})$。

相应地，G 递推式中的 $G_{k-1}(p_{k-1})=\sum_{i=1}^{k-1}g(p_i)$ 也可以用此方法预处理。

用 Extended Eratosthenes Sieve 求**积性函数** f 的前缀和时，应当明确以下几点：

- 如何快速（一般是线性时间复杂度）筛出前 $\sqrt{n}$ 个 f 值；

- $f(p)$ 的多项式表示；
- 如何快速求出 $f(p^c)$。

明确上述几点之后按顺序实现以下几部分即可：

1. 筛出 $[1,\sqrt{n}]$ 内的质数与前 $\sqrt{n}$ 个 f 值；
2. 对 $f(p)$ 多项式表示中的每一项筛出对应的 G，合并得到 F_{prime} 的所有 $O(\sqrt{n})$ 个有用点值；
3. 按照 F_k 的递推式实现递归，求出 $F_1(n)$。

例题

求莫比乌斯函数的前缀和

求 $\sum\limits_{i=1}^{n}\mu(i)$。

易知 $f(p)=-1$。则 $g(p)=-1, G_0(n)=\sum_{i=2}^{n}g(i)=-n+1$。

直接筛即可得到 F_{prime} 的所有 $O(\sqrt{n})$ 个所需点值。

求欧拉函数的前缀和

求 $\sum\limits_{i=1}^{n}\varphi(i)$。

首先易知 $f(p)=p-1$。

对于 $f(p)$ 的一次项 (p)，有 $g(p)=p, G_0(n)=\sum_{i=2}^{n}g(i)=\frac{(n+2)(n-1)}{2}$；

对于 $f(p)$ 的常数项 (-1)，有 $g(p)=-1, G_0(n)=\sum_{i=2}^{n}g(i)=-n+1$。

筛两次加起来即可得到 F_{prime} 的所有 $O(\sqrt{n})$ 个所需点值。

「LOJ #6053」 简单的函数[2]

给定 $f(n)$：

$$f(n)=\begin{cases}1 & n=1\\ p\operatorname{xor}c & n=p^c\\ f(a)f(b) & n=ab\land a\perp b\end{cases}$$

易知 $f(p)=p-1+2[p=2]$。则按照筛 φ 的方法筛，对 2 讨论一下即可。

此处给出一种 C++ 实现：

参考代码

```cpp
/* 「LOJ #6053」简单的函数 */
#include <algorithm>
#include <cmath>
#include <cstdio>

const int maxs = 200000;  // 2sqrt(n)
const int mod = 1000000007;

template <typename x_t, typename y_t>
void inc(x_t &x, const y_t &y) {
  x += y;
  (mod <= x) && (x -= mod);
}
```

```cpp
template <typename x_t, typename y_t>
void dec(x_t &x, const y_t &y) {
  x -= y;
  (x < 0) && (x += mod);
}

template <typename x_t, typename y_t>
int sum(const x_t &x, const y_t &y) {
  return x + y < mod ? x + y : (x + y - mod);
}

template <typename x_t, typename y_t>
int sub(const x_t &x, const y_t &y) {
  return x < y ? x - y + mod : (x - y);
}

template <typename _Tp>
int div2(const _Tp &x) {
  return ((x & 1) ? x + mod : x) >> 1;
}

// 以上目的均为防负数和取模
template <typename _Tp>
long long sqrll(const _Tp &x) {  // 平方函数
  return (long long)x * x;
}

int pri[maxs / 7], lpf[maxs + 1], spri[maxs + 1], pcnt;

void sieve(const int &n) {
  for (int i = 2; i <= n; ++i) {
    if (lpf[i] == 0) {  // 记录质数
      lpf[i] = ++pcnt;
      pri[lpf[i]] = i;
      spri[pcnt] = sum(spri[pcnt - 1], i);  // 前缀和
    }
    for (int j = 1, v; j <= lpf[i] && (v = i * pri[j]) <= n; ++j) lpf[v] = j;
  }
}

long long global_n;
int lim;
int le[maxs + 1],  // x <= \sqrt{n}
    ge[maxs + 1];  // x > \sqrt{n}
#define idx(v) (v <= lim ? le[v] : ge[global_n / v])

int G[maxs + 1][2], Fprime[maxs + 1];
long long lis[maxs + 1];
int cnt;

void init(const long long &n) {
  for (long long i = 1, j, v; i <= n; i = n / j + 1) {
    j = n / i;
    v = j % mod;
```

```
    lis[++cnt] = j;
    (j <= lim ? le[j] : ge[global_n / j]) = cnt;
    G[cnt][0] = sub(v, 1ll);
    G[cnt][1] = div2((long long)(v + 2ll) * (v - 1ll) % mod);
  }
}

void calcFprime() {
  for (int k = 1; k <= pcnt; ++k) {
    const int p = pri[k];
    const long long sqrp = sqrll(p);
    for (int i = 1; lis[i] >= sqrp; ++i) {
      const long long v = lis[i] / p;
      const int id = idx(v);
      dec(G[i][0], sub(G[id][0], k - 1));
      dec(G[i][1], (long long)p * sub(G[id][1], spri[k - 1]) % mod);
    }
  }
  /* F_prime = G_1 - G_0 */
  for (int i = 1; i <= cnt; ++i) Fprime[i] = sub(G[i][1], G[i][0]);
}

int f_p(const int &p, const int &c) {
  /* f(p^{c}) = p xor c */
  return p xor c;
}

int F(const int &k, const long long &n) {
  if (n < pri[k] || n <= 1) return 0;
  const int id = idx(n);
  long long ans = Fprime[id] - (spri[k - 1] - (k - 1));
  if (k == 1) ans += 2;
  for (int i = k; i <= pcnt && sqrll(pri[i]) <= n; ++i) {
    long long pw = pri[i], pw2 = sqrll(pw);
    for (int c = 1; pw2 <= n; ++c, pw = pw2, pw2 *= pri[i])
      ans +=
          ((long long)f_p(pri[i], c) * F(i + 1, n / pw) + f_p(pri[i], c + 1)) %
          mod;
  }
  return ans % mod;
}

int main() {
  scanf("%lld", &global_n);
  lim = sqrt(global_n);  // 上限

  sieve(lim + 1000);  // 预处理
  init(global_n);
  calcFprime();
  printf("%lld\n", (F(1, global_n) + 1ll + mod) % mod);

  return 0;
}
```

参考资料与注释

[1] Min_25

[2] 「LOJ #6053」简单的函数

9.12.27 洲阁筛

Authors: Early0v0

前置知识

- 积性函数

定义

洲阁筛是一种能在亚线性时间复杂度内求出大多数积性函数前缀和的筛法。

下面将以求解 $\sum\limits_{i=1}^{n} f(i)$ 为例，具体阐述洲阁筛的原理。

约定

- $\mathbb{P}$ 表示质数集，p_i 表示第 i 个质数。
- m 表示 $\sqrt{n}$ 内的质数个数。

要求

当 $p \in \mathbb{P}, c \in \mathbb{N}$ 时，$f(p^c)$ 为一个关于 p 的低阶多项式。

思想

- 对于任意 $[1,n]$ 内的整数，其至多只有一个 $>\sqrt{n}$ 的质因子。
- 利用 $\left\lfloor\dfrac{n}{i}\right\rfloor (i \in [1,n] \cap \mathbb{N})$ 只有 $\sqrt{n}$ 级别个取值的性质来降低时间复杂度。

过程

将 $[1,n]$ 内的所有整数按是否有 $>\sqrt{n}$ 的质因子分为两类：

$$\sum_{i=1}^{n} f(i) = \sum_{i=1}^{n} [\exists d \in (\sqrt{n}, n] \cap \mathbb{P}, d \mid i]\, f(i) + \sum_{i=1}^{n} [\forall d \in (\sqrt{n}, n] \cap \mathbb{P}, d \nmid i]\, f(i)$$

对于前半部分，枚举最大因子，根据积性函数的性质可以转换：

$$\sum_{i=1}^{n} f(i) = \sum_{i=1}^{\sqrt{n}} f(i) \cdot \left(\sum_{d=\lfloor\sqrt{n}\rfloor+1}^{\lfloor\frac{n}{i}\rfloor} [d \in \mathbb{P}] f(d) \right) + \sum_{i=1}^{n} [\forall d \in (\sqrt{n}, n] \cap \mathbb{P}, d \nmid i]\, f(i)$$

前后部分可以分别计算。

Part 1

> 计算 $\sum_{i=1}^{\sqrt{n}} f(i) \cdot \left(\sum_{d=\lfloor\sqrt{n}\rfloor+1}^{\lfloor\frac{n}{i}\rfloor} [d \in \mathbb{P}] f(d) \right)$。

考虑枚举 i，然后 $\mathcal{O}(1)$ 计算括号内部分。

记 $g(t,l) = \sum_{i=1}^{l} [\forall j \in [1,t], \gcd(i,p_j)=1] f(i)$，即 $[1,l]$ 中与 $p_1, p_2, \dots, p_t$ 均互质的数的 f 值之和。

这样 Part 1 的计算就变成了 $\sum_{i=1}^{\sqrt{n}} f(i) \cdot g\left(m, \left\lfloor \frac{n}{i} \right\rfloor\right)$。

边界 $g(0,l) = \sum_{i=1}^{l} f(i)$，转移 $g(t,l) = g(t-1,l) - f(p_t) \cdot g\left(t-1, \left\lfloor \frac{l}{p_t} \right\rfloor\right)$。

l 共有 $\sqrt{n}$ 级别种取值，对于每种取值则需要枚举其质因子，所以复杂度为 $\mathcal{O}\left(\frac{\sqrt{n}}{\ln\sqrt{n}} \cdot \sqrt{n}\right) = \mathcal{O}\left(\frac{n}{\log n}\right)$，需要优化。

注意到 $p_{t+1}^2 > l$ 时符合条件的数只有 1，所以此时 $g(t,l) = f(1) = 1$。

代入递推式可得：当 $p_t^2 > l$ 时，$g(t,l) = g(t-1,l) - f(p_t)$。

所以一旦发现 $p_t^2 > l$ 就停止转移，记此时的 t 为 t_l，则 $\forall t > t_l, g(t,l) = g(t_l,l) - \sum_{i=t_l}^{t-1} f(p_i)$。

预处理质数的 f 值前缀和即可快速求出 g，时间复杂度被优化至 $\mathcal{O}\left(\frac{n^{\frac{3}{4}}}{\log n}\right)$。

Part 2

> 计算 $\sum_{i=1}^{n} [\forall d \in (\sqrt{n}, n] \cap \mathbb{P}, d \nmid i] f(i)$。

记 $h(t,l) = \sum_{i=1}^{l} \left[i = \prod_{j=t}^{m} p_j^{c_j}, c_j \in \mathbb{N} \right] f(i)$，即 $[1,l]$ 中所有只含 $p_t, p_{t+1}, \dots, p_m$ 质因子的数的 f 值之和。

Part 2 即为求 $h(0,n)$。

边界 $h(m+1,l) = 1$，转移 $h(t,l) = h(t+1,l) + \sum_{c \in \mathbb{N}^*} f(p_t^c) \cdot h\left(t+1, \left\lfloor \frac{l}{p_t^c} \right\rfloor\right)$。

l 共有 $\sqrt{n}$ 级别种取值，所以直接转移复杂度为 $\mathcal{O}\left(\sqrt{n} \cdot \frac{\sqrt{n}}{\ln\sqrt{n}}\right) = \mathcal{O}\left(\frac{n}{\log n}\right)$，需要优化。

与 g 的优化方式类似，注意到 $p_t > l$ 时，能用 $p_t, p_{t+1}, \dots, p_m$ 组成的数只有 1，此时的 $h(t,l) = f(1) = 1$。

类似的，推出 $\forall p_t^2 > l, h(t,l) = h(t-1,l) + f(p_t)$。

所以一旦发现 $p_t^2 > l$ 就停止转移，记此时的 t 为 t_l，之后用到 h 时，把此时的 h 值加上 $\sum_{i=p_{t_l}}^{\min(l,\sqrt{n})} [i \in \mathbb{P}] f(i)$ 即可。

时间复杂度被优化至 $\mathcal{O}\left(\frac{n^{\frac{3}{4}}}{\log n}\right)$。

求和

算出了 Part 1 和 Part 2 的答案，将其相加即为 $\sum_{i=1}^{n} f(i)$。

参考

积性函数线性筛 / 杜教筛 / 洲阁筛学习笔记 | Bill Yang's Blog[1]

参考资料与注释

[1] 积性函数线性筛/杜教筛/洲阁筛学习笔记 | Bill Yang's Blog

9.12.28 连分数

连分数

连分数是实数作为有理数的特定收敛序列的表示。它们在算法竞赛（competitive programming）中很有用，因为它们易于计算，并且可以有效地用于在分母不超过给定值的所有数字中，找到基础实数（underlying real number）的最佳可能有理近似（best possible rational approximation）。

除此之外，连分数与欧几里得算法密切相关，这使得它们在一系列数论问题中非常有用。

定义

连分数是一种记号。例如，长为 4 的连分数：

$$[a_0,a_1,a_2,a_3]=a_0+\frac{1}{a_1+\frac{1}{a_2+\frac{1}{a_3}}}$$

只是为了形式上简洁，才记成等号左边的样子。这里的四个变元可以任意取值。

连分数各变元的下标从 0 开始。

简单连分数

可以证明，任何有理数都可以精确地以两种方式表示为连分数：

$$r=[a_0;a_1,\dots,a_k,1]=[a_0;a_1,\dots,a_k+1]$$

此外，对于 $r=\frac{p}{q}$，这种连分数的长度 k 估计为 $k=O(\log\min(p,q))$。

一旦深入研究了连分数构造的细节，这背后的原因就会很清楚。

定义

设 $a_0,a_1,\dots,a_k\in\mathbb{Z}$ 和 $a_1,a_2,\dots,a_k\geq 1$。然后表达式

$$r=a_0+\frac{1}{a_1+\frac{1}{\dots+\frac{1}{a_k}}}$$

称为有理数 r 的**连分数表示**，并简短地表示为 $r=[a_0;a_1,a_2,\dots,a_k]$。

示例

设 $r=\frac{5}{3}$。有两种方法可以将其表示为连分数：

$$r=[1;1,1,1]=1+\frac{1}{1+\frac{1}{1+\frac{1}{1}}},\tag{9.13}$$

$$r=[1;1,2]=1+\frac{1}{1+\frac{1}{2}}.\tag{9.14}$$

对于有限连分数，全体结尾为 1 的有限连分数和全体结尾不为 1 的有限连分数一一对应，即同一个连分数有两种表示：

$[a_0,a_1,a_2,a_3]=[a_0,a_1,a_2,a_3-1,1]$

简单连分数：连分数从第 1 项开始全都是正整数。如果有限，要求最后一项不为 1。（第 0 项可以任意）

简单连分数的值，一定大于偶数的渐进分数，一定小于奇数的渐进分数。无限简单连分数一定收敛。

仿照一般分数的概念，第 0 项是 0 的连分数称为「真分数」。显然如果这之后的所有变元都大于等于 1，那么得到的真分数一定落在 0 到 1 之间。

无限连分数

如果分式无限地写下去，有无限个变元，就得到无限连分数。无限连分数收敛等价于渐进分数收敛。

有定理：

无限连分数，如果各变元均大于等于 1，那么一定收敛。

因为只要各变元为正，无限连分数的偶渐进分数单调递增（都比它小），奇渐进分数单调递减（都比它大）。而在均大于等于 1 时，相邻（奇偶间）两个渐进分数之间距离可以给出估计式，趋于 0，因此收敛。

显然可以看到，连分数关于下标为偶数的变元单调递增，关于下标为奇数的变元单调递减。这无论它有限或无限都成立。

定义

设 $a_0, a_1, a_2, \dots$ 为整数序列，使得 $a_1, a_2, \dots \geq 1$。设 $r_k = [a_0; a_1, \dots, a_k]$。然后表达式

$$r = a_0 + \cfrac{1}{a_1 + \frac{1}{a_2 + \dots}} = \lim_{k \to \infty} r_k$$

称为无理数 r 的**连分数表示**，并简短地表示为 $r = [a_0; a_1, a_2, \dots]$。

注意，对于 $r = [a_0; a_1, \dots]$ 和整数 k，有 $r + k = [a_0 + k; a_1, \dots]$。

另一个重要的观察结果是，当 $a_0 > 0$ 时，$\frac{1}{r} = [0; a_0, a_1, \dots]$；当 a_1 0 时，$\frac{1}{r} = [a_1; a_2, \dots]$。

渐进分数

定义

在上面的定义中，有理数 $r_0, r_1, r_2, \dots$ 称为 r 的**渐进分数**（convergents，意为「收敛」）。

相应地，单个 $r_k = [a_0; a_1, \dots, a_k] = \frac{p_k}{q_k}$ 称为 r 的第 k 个渐进分数。

示例

考虑 $r = [1; 1, 1, 1, \dots]$。可以通过归纳法证明 $r_k = \frac{F_{k+2}}{F_{k+1}}$，其中 F_k 是斐波那契序列，定义为 $F_0 = 0$、$F_1 = 1$ 和 $F_k = F_{k-1} + F_{k-2}$。从 Binet 公式可知

$$r_k = \frac{\phi^{k+2} - \psi^{k+2}}{\phi^{k+1} - \psi^{k+1}}$$

其中 $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ 是黄金比率，$\psi = \frac{1-\sqrt{5}}{2} = -\frac{1}{\phi} \approx -0.618$。因此

$$r = 1 + \cfrac{1}{1 + \frac{1}{1 + \dots}} = \lim_{k \to \infty} r_k = \phi = \frac{1 + \sqrt{5}}{2}.$$

请注意，在这种特定情况下，找到 r 的另一种方法是求解方程

$$r = 1 + \frac{1}{r} \implies r^2 = r + 1$$

定义

设 $r_k = [a_0; a_1, \dots, a_{k-1}, a_k]$。对于 $1 \leq t \leq a_k$，$[a_0; a_1, \dots, a_{k-1}, t]$ 称为**中间分数**（semiconvergents，「semi」意为「半」）。

通常将大于 r 的分数称为**上**（upper）渐进分数或中间分数，将小于 r 者称为**下**（lower）渐进分数或中间分数。

余项和部分商

定义

作为渐进分数的补充，定义**余项**（完全商，complete quotients）为 $s_k = [a_k; a_{k+1}, a_{k+2}, \dots]$。

相应地，将单个 s_k 称为 r 的第 k 个完全商。

相应地，取整得到的 a_k 称为部分商。

对于各项均为正数的连分数，所有的余项也都是正数。

根据以上定义，可以得出 $s_k \geq 1$ 代表 $k \geq 1$ 的结论。

将 $[a_0; a_1, \dots, a_k]$ 视为一个形式代数表达式，并允许任意实数代替 a_i，得到

$$r = [a_0; a_1, \dots, a_{k-1}, s_k]$$

特别地，$r = [s_0] = s_0$。另一方面，可以将 s_k 表示为

$$s_k = [a_k; s_{k+1}] = a_k + \frac{1}{s_{k+1}}$$

这意味着可以从 s_k 计算 $a_k = \lfloor s_k \rfloor$ 和 $s_{k+1} = (s_k - a_k)^{-1}$。

序列 $a_0, a_1, \dots$ 定义良好，除非 $s_k = a_k$，这仅在 r 为有理数时发生。

因此，对于任何无理数 r，连分数表示都是唯一的。

求解简单连分数表示

在代码片段中主要假设有限的连分数。

如果要求 $(0, 1)$ 区间内某个数的简单连分数表示（第 0 项为 0），只需：

- 取倒数，得到的余项大于 1。
- 取整得到整数部分为部分商，小数部分在 0 到 1 之间。
- 对小数部分重复上述操作。

这样就得到了相应的表示。

从 s_k 到 s_{k+1} 的转换如下

$$s_k = \lfloor s_k \rfloor + \frac{1}{s_{k+1}}$$

从这个表达式中，下一个完全商 s_{k+1} 如下

$$s_{k+1} = \left(s_k - \lfloor s_k \rfloor\right)^{-1}$$

对于 $s_k = \frac{p}{q}$，这意味着

$$s_{k+1} = \left(\frac{p}{q} - \left\lfloor \frac{p}{q} \right\rfloor\right)^{-1} = \frac{q}{p - q \cdot \lfloor \frac{p}{q} \rfloor} = \frac{q}{p \bmod q}.$$

因此，$r = \frac{p}{q}$ 的连分数表示的计算遵循 p 和 q 的欧几里得算法的步骤。

由此，$\frac{p_k}{q_k} = [a_0; a_1, \dots, a_k]$ 的 $\gcd(p_k, q_k) = 1$。因此，渐进分数总是不可约的。

```cpp
auto fraction(int p, int q) {
  vector<int> a;
  while (q) {
    a.push_back(p / q);
    tie(p, q) = make_pair(q, p % q);
  }
  return a;
}
```

```
def fraction(p, q):
    a = []
    while q:
        a.append(p // q)
        p, q = q, p % q
    return a
```

如果规定第 0 项是该数的取整，那么全体实数都有「唯一的简单连分数表示」。其中：

如果两个无限简单连分数的值相等，必然逐项相等。

如果两个有限简单连分数的值相等，不仅要逐项相等，而且必然项数也相同。

无限简单连分数不能与有限简单连分数值相等。有理数与有限简单连分数具有一一对应关系，因此无限简单连分数全都是无理数。

性质

为了给连分数的进一步研究提供一些动力，现在给出一些性质。

递推

" 递推"

对于渐进分数 $r_k = \frac{p_k}{q_k}$，以下递推公式适用于它们的快速计算：

$$\frac{p_k}{q_k} = \frac{a_k p_{k-1} + p_{k-2}}{a_k q_{k-1} + q_{k-2}}$$

其中 $\frac{p_{-1}}{q_{-1}} = \frac{1}{0}$ 并且 $\frac{p_{-2}}{q_{-2}} = \frac{0}{1}$。

渐进分数分子和分母具有完全相同的递推关系：

$$p_k = a_k p_{k-1} + p_{k-2}$$

$$q_k = a_k q_{k-1} + q_{k-2}$$

这里和 Farey 数列的递推关系很像。

形式上记初项：

$$p_{-1} = 1 \quad p_0 = a_0$$

$$q_{-1} = 0 \quad q_0 = 1$$

只是形式上成立。第 -1 项渐进分数是 $1/0$，没有实际意义。

" 证明"

可以注意到，p_k 与 q_k 对于 a_k 和 b_k 都是线性函数。这是因为，a_k 和 b_k 都只出现了一次，无论如何通分也不会有另一个 a_k 或 b_k 乘上去。于是通过待定系数，即可解得这个递推关系。

反序定理

$$\frac{q_k}{q_{k-1}} = [a_k, a_{k-1}, \dots, a_1]$$

如果 $a_0 \neq 0$：

$$\frac{p_k}{p_{k-1}} = [a_k, a_{k-1}, \dots, a_0]$$

如果 $a_0=0$：

$$\frac{p_k}{p_{k-1}}=[a_k,a_{k-1},\dots,a_2]$$

"证明"

对递推关系稍加改造，有：

$$\frac{p_k}{p_{k-1}}=a_k+\frac{1}{\frac{p_{k-1}}{p_{k-2}}}$$

$$\frac{q_k}{q_{k-1}}=a_k+\frac{1}{\frac{q_{k-1}}{q_{k-2}}}$$

又利用初值，即可证明反序定理。

渐进分数的差分

计算相邻两项渐进分数的差，需要通分。通分后的分子代入递推关系：

$$p_{k+1}q_k-q_{k+1}p_k=\left(a_{k+1}p_k+p_{k-1}\right)q_k-\left(a_{k+1}q_k+q_{k-1}\right)p_k=-\left(p_kq_{k-1}-q_kp_{k-1}\right)$$

代入初值就有渐进分数的差分：

$$p_{k+1}q_k-q_{k+1}p_k=(-1)^k$$

$$\frac{p_{k+1}}{q_{k+1}}-\frac{p_k}{q_k}=\frac{(-1)^k}{q_{k+1}q_k}$$

"注"

可以观察到，式 $p_{k+1}q_k-q_{k+1}p_k$ 特别像一个行列式，完全可以按「行列式」理解。

渐进分数的递推关系很像行列式的列变换。行列式一列加到另一列上不改变它的值，两列交换则反号。

根据递推式，如果连分数各项均为整数，则渐进分数分子分母总是互素。

对于有理数的简单连分数展开，常用渐进分数差分的等式，求解一次线性不定方程（参见 扩展欧几里得算法）：

$$p_{k+1}q_k-q_{k+1}p_k=(-1)^k$$

$$ax-by=1$$

因为 a 与 b 互素，$\frac{a}{b}$ 就是最简的有理数，也就是它本身的最后一个渐进分数。那么，它的前一个渐进分数就是所求的解。

倒数定理

由于实数与简单连分数一一对应，称实数的简单连分数的渐进分数，就是实数的渐进分数。于是就有倒数定理：

对于大于 1 的实数 x，x 的渐进分数的倒数恰好是 $\frac{1}{x}$ 的渐进分数。显然，该定理也应该对于 0 到 1 之间的实数 x 成立。

"证明"

$$x=[a_0,a_1,a_2,\ldots]$$

$$\frac{1}{x}=\frac{1}{[a_0,a_1,a_2,\ldots]}=[0,a_0,a_1,a_2,\ldots]$$

于是根据新的初值与递推就能发现倒数关系成立。

最佳逼近

" 最佳逼近"

让 $\frac{p}{q}$ 是最小化 $\left|r-\frac{p}{q}\right|$ 的分数，对于某些 x，该分数服从 $q \leq x$。
那么 $\frac{p}{q}$ 是 r 的中间分数。

因此允许通过检查 r 是否中间分数来找到其最佳有理逼近。
下面会对这些性质建立一些直觉并做出进一步解释。

连行列式

继续看前面定义的渐近分数。对于 $r=[a_0, a_1, a_2, \ldots]$，其渐近分数为

$$
\begin{aligned}
r_0 &= [a_0] \\
r_1 &= [a_0, a_1] \\
&\ldots \\
r_k &= [a_0, a_1, \ldots, a_k]
\end{aligned}
$$

渐近分数是连分数的核心概念，因此研究它们的性质很重要。
对于数字 r，其第 k 个渐近分数 $r_k=\frac{p_k}{q_k}$ 可以计算为

$$r_k=\frac{P_k(a_0, a_1, \ldots, a_k)}{P_{k-1}(a_1, \ldots, a_k)}=\frac{a_k p_{k-1}+p_{k-2}}{a_k q_{k-1}+q_{k-2}}$$

其中 $P_k(a_0, \ldots, a_k)$ 是连行列式（continuant），定义为

$$P_k(x_0, x_1, \ldots, x_k)=\det \begin{bmatrix} x_k & 1 & 0 & \ldots & 0 \\ -1 & x_{k-1} & 1 & \ldots & 0 \\ 0 & -1 & x_2 & . & \vdots \\ \vdots & \vdots & . & \ddots & 1 \\ 0 & 0 & \ldots & -1 & x_0 \end{bmatrix}.$$

因此，r_k 是 r_{k-1} 和 r_{k-2} 的加权中间值（mediant）。
为了一致性，定义了两个额外的渐近分数 $r_{-1}=\frac{1}{0}$ 和 $r_{-2}=\frac{0}{1}$。

" 详细说明"

渐进分数 r_k 的分子和分母可以看作 $a_0, a_1, \ldots, a_k$ 的多元多项式：

$$r_k=\frac{P_k(a_0, a_1, \ldots, a_k)}{Q_k(a_0, a_1, \ldots, a_k)}$$

根据渐进分数的定义，

$$r_k=a_0+\frac{1}{[a_1; a_2, \ldots, a_k]}=a_0+\frac{Q_{k-1}(a_1, \ldots, a_k)}{P_{k-1}(a_1, \ldots, a_k)}=\frac{a_0 P_{k-1}(a_1, \ldots, a_k)+Q_{k-1}(a_1, \ldots, a_k)}{P_{k-1}(a_1, \ldots, a_k)}$$

由此得出 $Q_k(a_0, \ldots, a_k)=P_{k-1}(a_1, \ldots, a_k)$。这产生了关系

$$P_k(a_0, \ldots, a_k)=a_0 P_{k-1}(a_1, \ldots, a_k)+P_{k-2}(a_2, \ldots, a_k)$$

最初，$r_0=\frac{a_0}{1}$ 和 $r_1=\frac{a_0 a_1+1}{a_1}$，因此

$$P_0(a_0)=a_0, \tag{9.15}$$
$$P_1(a_0, a_1)=a_0 a_1+1. \tag{9.16}$$

为了保持一致性，可以方便地定义 $P_{-1}=1$ 和 $P_{-2}=0$，并正式表示 $r_{-1}=\frac{1}{0}$ 和 $r_{-2}=\frac{0}{1}$。

从数值分析可知，任意三对角矩阵的行列式

$$T_k=\det\begin{bmatrix} a_0 & b_0 & 0 & \dots & 0 \\ c_0 & a_1 & b_1 & \dots & 0 \\ 0 & c_1 & a_2 & . & \vdots \\ \vdots & \vdots & . & \ddots & c_{k-1} \\ 0 & 0 & \dots & b_{k-1} & a_k \end{bmatrix}$$

可以递归地计算为 $T_k=a_kT_{k-1}-b_{k-1}c_{k-1}T_{k-2}$。将其与 P_k 进行比较，得到一个直接表达式

$$P_k=\det\begin{bmatrix} x_k & 1 & 0 & \dots & 0 \\ -1 & x_{k-1} & 1 & \dots & 0 \\ 0 & -1 & x_2 & . & \vdots \\ \vdots & \vdots & . & \ddots & 1 \\ 0 & 0 & \dots & -1 & x_0 \end{bmatrix}.$$

这个多项式也被称为连行列式（continuant），由于其与连分数的密切关系。如果主对角线上的顺序颠倒，则连行列式不会改变。这产生了一个计算公式：

$$P_k(a_0,\dots,a_k)=a_kP_{k-1}(a_0,\dots,a_{k-1})+P_{k-2}(a_0,\dots,a_{k-2})$$

实现

把渐进分数计算为一对序列 $p_{-2},p_{-1},p_0,p_1,\dots,p_k$ 和 $q_{-2},q_{-1},q_0,q_1,\dots,q_k$：

```cpp
auto convergents(vector<int> a) {
  vector<int> p = {0, 1};
  vector<int> q = {1, 0};
  for (auto it : a) {
    p.push_back(p[p.size() - 1] * it + p[p.size() - 2]);
    q.push_back(q[q.size() - 1] * it + q[q.size() - 2]);
  }
  return make_pair(p, q);
}
```

```python
def convergents(a):
    p = [0, 1]
    q = [1, 0]
    for it in a:
        p.append(p[-1]*it + p[-2])
        q.append(q[-1]*it + q[-2])
    return p, q
```

误差和余项的估计

" 误差"

渐进分数 $r_k=\frac{p_k}{q_k}$ 与 r 的误差（deviation）通常可以估计为

$$\left|\frac{p_k}{q_k}-r\right|\le\frac{1}{q_kq_{k+1}}\le\frac{1}{q_k^2}$$

将两边乘以 q_k，得到另一个估计：

$$|p_k - q_k r| \le \frac{1}{q_{k+1}}$$

从上面的循环可以看出，q_k 的增长速度至少与斐波那契数一样快。

在下图中可以看到收敛 r_k 渐进分数 $r = \frac{1+\sqrt{5}}{2}$ 的可视化：

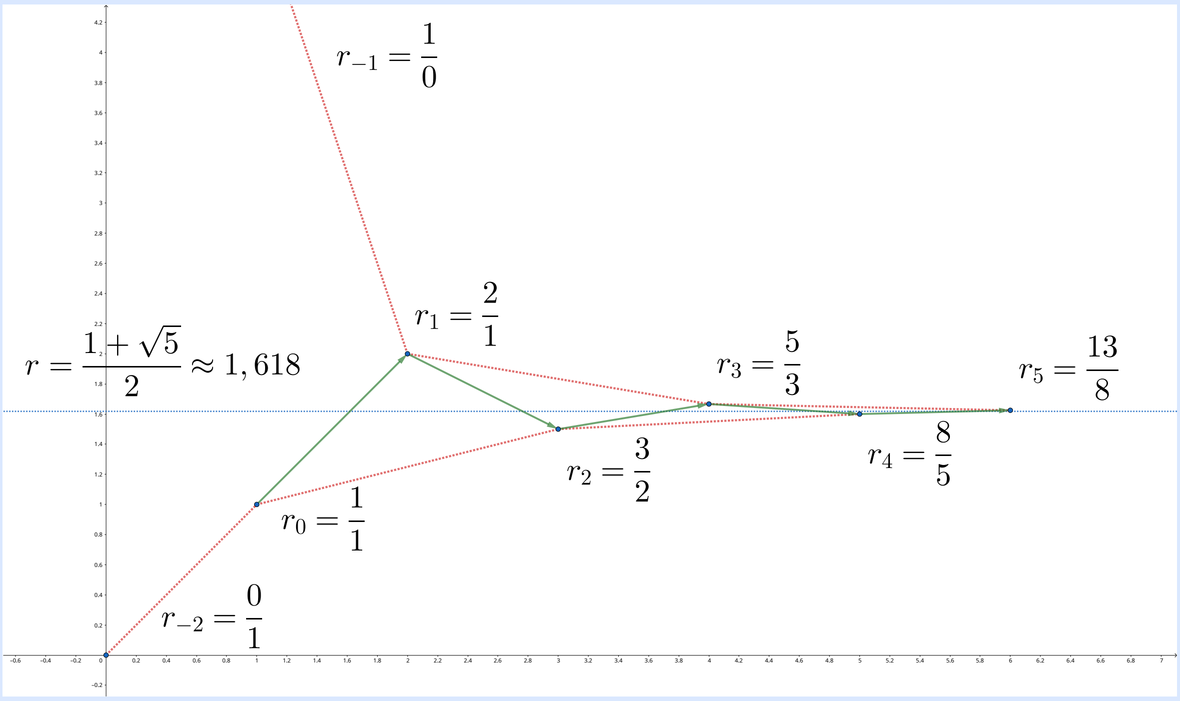


图 9.12

无理数 $r = \frac{1+\sqrt{5}}{2}$ 由蓝色虚线表示。奇数渐进分数从上面接近它，偶数渐进分数从下面接近它。

实数 x 也可以写成：

$$x = [a_0, a_1, \dots, a_k, r_{k+1}]$$

最后一项渐近分数就是 x 本身。于是根据渐进分数的递推式，就有：

$$x = \frac{r_{k+1} p_k + p_{k-1}}{r_{k+1} q_k + q_{k-1}}$$

于是可以估计渐进分数的误差：

$$x - \frac{p_k}{q_k} = \frac{(-1)^k}{q_k \left(r_{k+1} q_k + q_{k-1}\right)}$$

分别对 k 取奇数偶数就得到，x 总小于其奇数阶渐近分数，大于其偶数阶渐近分数。

对于数字 r 及其第 k 个渐进分数 $r_k = \frac{p_k}{q_k}$，以下公式成立：

$$r_k = a_0 + \sum_{i=1}^{k} \frac{(-1)^{i-1}}{q_i q_{i-1}}$$

特别地，这意味着

$$r_k - r_{k-1} = \frac{(-1)^{k-1}}{q_k q_{k-1}}$$

并且

$$p_k q_{k-1} - p_{k-1} q_k = (-1)^{k-1}$$

由此可以得出结论

$$\left|r-\frac{p_k}{q_k}\right| \le \frac{1}{q_{k+1}q_k} \le \frac{1}{q_k^2}$$

后一种不等式是由于 r_k 和 r_{k+1} 通常位于 r 的不同侧面，因此

$$|r-r_k| = |r_k - r_{k+1}| - |r - r_{k+1}| \le |r_k - r_{k+1}|$$

" 详细说明"

为了估计 $|r-r_k|$，首先估计相邻渐进分数之间的差异。根据定义，

$$\frac{p_k}{q_k} - \frac{p_{k-1}}{q_{k-1}} = \frac{p_k q_{k-1} - p_{k-1}q_k}{q_k q_{k-1}}$$

将分子中的 p_k 和 q_k 替换为它们的循环，得到

$$\begin{aligned} p_k q_{k-1} - p_{k-1} q_k &= (a_k p_{k-1} + p_{k-2}) q_{k-1} - p_{k-1}(a_k q_{k-1} + q_{k-2}) && (9.17) \\ &= p_{k-2} q_{k-1} - p_{k-1} q_{k-2}, && (9.18) \end{aligned}$$

因此，$r_k - r_{k-1}$ 的分子总是与 $r_{k-1} - r_{k-2}$ 相反。反过来，它等于

$$r_1 - r_0 = \left(a_0 + \frac{1}{a_1}\right) - a_0 = \frac{1}{a_1}$$

因此

$$r_k - r_{k-1} = \frac{(-1)^{k-1}}{q_k q_{k-1}}$$

这产生了 r_k 作为无限级数的部分和的替代表示：

$$r_k = (r_k - r_{k-1}) + \cdots + (r_1 - r_0) + r_0 = a_0 + \sum_{i=1}^{k} \frac{(-1)^{i-1}}{q_i q_{i-1}}$$

根据递归关系，q_k 单调增加的速度至少与斐波那契数一样快，因此

$$r = \lim_{k\to\infty} r_k = a_0 + \sum_{i=1}^{\infty} \frac{(-1)^{i-1}}{q_i q_{i-1}}$$

总是定义明确的，因为基础系列总是收敛的。值得注意的是，剩余系列

$$r - r_k = \sum_{i=k+1}^{\infty} \frac{(-1)^{i-1}}{q_i q_{i-1}}$$

由于 $q_i q_{i-1}$ 下降的速度，与 $(-1)^k$ 具有相同的符号。因此，偶数索引的 r_k 从下面接近 r，而奇数索引的 r_k 从上面接近：

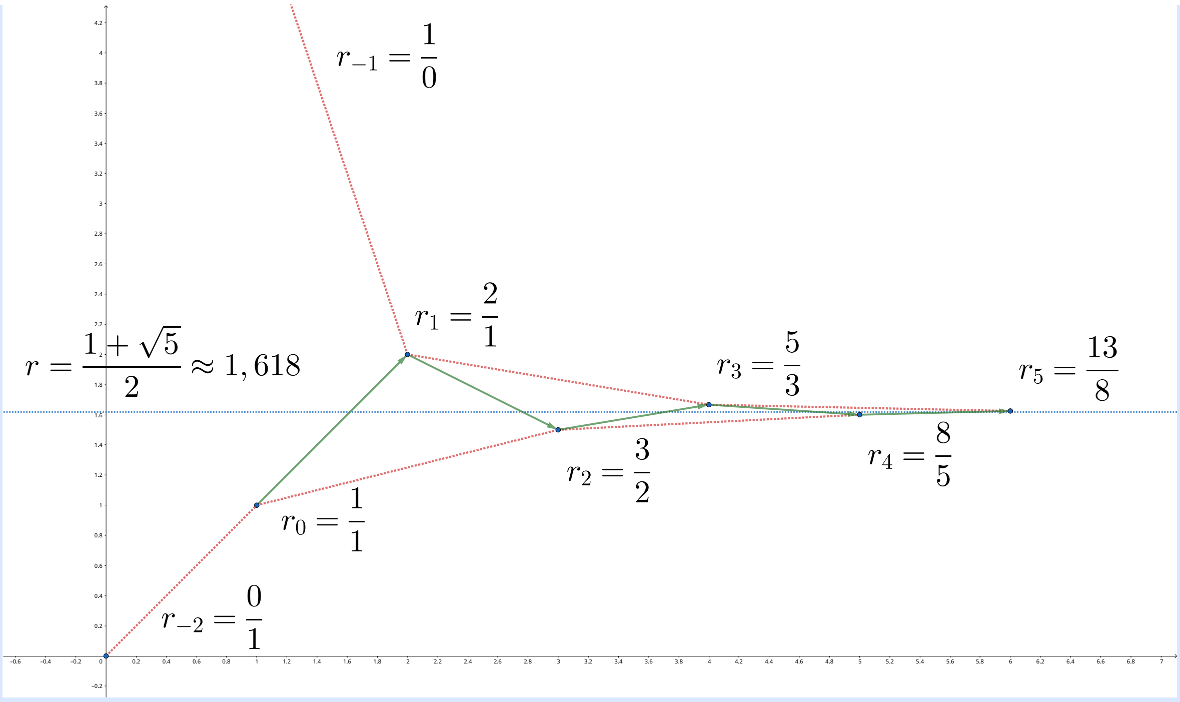


图 9.13

从这张图中可以看到

$$|r - r_k| = |r_k - r_{k+1}| - |r - r_{k+1}| \leq |r_k - r_{k+1}|$$

因此，r 和 r_k 之间的距离永远不会大于 r_k 和 r_{k+1} 之间的间距：

$$\left| r - \frac{p_k}{q_k} \right| \leq \frac{1}{q_k q_{k+1}} \leq \frac{1}{q_k^2}$$

例题扩展欧几里得

您将获得 $A, B, C \in \mathbb{Z}$。查找 $x, y \in \mathbb{Z}$，使 $Ax + By = C$.

" 解答"

虽然这个问题通常是用扩展欧几里得算法解决的，但有一个简单而直接的连分数的解决方案。

设 $\frac{A}{B} = [a_0; a_1, \dots, a_k]$。上面证明了 $p_k q_{k-1} - p_{k-1} q_k = (-1)^{k-1}$。将 p_k 和 q_k 替换为 A 和 B，得到

$$A q_{k-1} - B p_{k-1} = (-1)^{k-1} g$$

其中 $g = \gcd(A, B)$。如果 C 可被 g 整除，则解为 $x = (-1)^{k-1} \frac{C}{g} q_{k-1}$ 和 $y = (-1)^k \frac{C}{g} p_{k-1}$。

```python
# return (x, y) such that Ax+By=C
# assumes that such (x, y) exists
def dio(A, B, C):
    p, q = convergents(fraction(A, B))
    C //= A // p[-1] # divide by gcd(A, B)
    t = (-1) if len(p) % 2 else 1
    return t*C*q[-2], -t*C*p[-2]
```

几何解释

" 格点"

考虑线 $y=rx$ 上方和下方的点的凸包。

奇数渐进分数 $(q_k;p_k)$ 是上壳的顶点，而偶数渐进分数 $(q_k;p_k)$ 则是下壳的顶点。

外壳上的所有整数顶点都作为 $(q;p)$ 获得，这样

$$\frac{p}{q}=\frac{tp_{k-1}+p_{k-2}}{tq_{k-1}+q_{k-2}}$$

对于整数 $0\le t\le a_k$。换句话说，外壳上的格点集对应于中间分数。

在下图中可以看到 $r=\frac{9}{7}$ 的渐进分数和中间分数（灰点）。

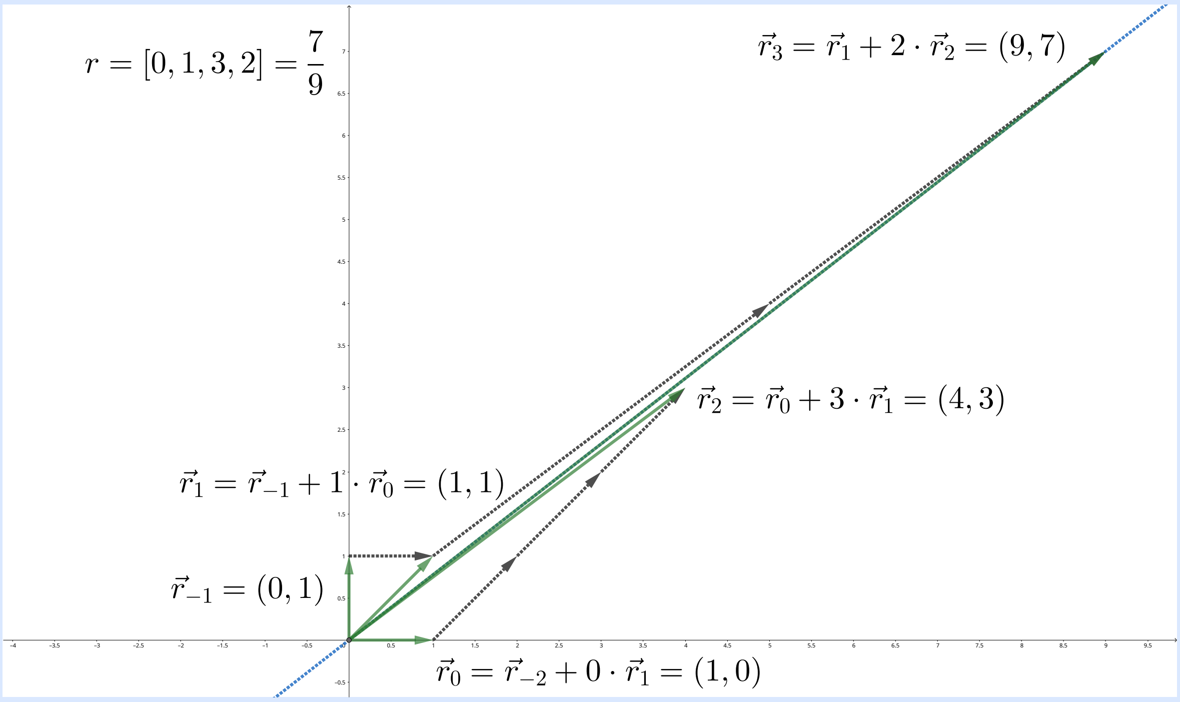


图 9.14

对于渐进分数 $r_k=\frac{p_k}{q_k}$，设 $\vec{r}_k=(q_k;p_k)$。然后，以下重复出现：

$$\vec{r}_k=a_k\vec{r}_{k-1}+\vec{r}_{k-2}$$

设 $\vec{r}=(1;r)$。然后，每个向量 $(x;y)$ 对应于等于其斜率系数 $\frac{y}{x}$ 的数字。

利用外积 $(x_1;y_1)\times(x_2;y_2)=x_1y_2-x_2y_1$ 的概念，可以看出（参见下面的解释）

$$s_k=-\frac{\vec{r}_{k-2}\times\vec{r}}{\vec{r}_{k-1}\times\vec{r}}=\left|\frac{\vec{r}_{k-2}\times\vec{r}}{\vec{r}_{k-1}\times\vec{r}}\right|$$

最后一个等式是由于 r_{k-1} 和 r_{k-2} 位于 r 的不同侧，因此 $\vec{r}_{k-1}$ 和 $\vec{r}_{k-2}$ 与 $\vec{r}$ 的外积具有不同的符号。考虑到 $a_k=\lfloor s_k\rfloor$，$\vec{r}_k$ 的公式如下

$$\vec{r}_k=\vec{r}_{k-2}+\left\lfloor\left|\frac{\vec{r}\times\vec{r}_{k-2}}{\vec{r}\times\vec{r}_{k-1}}\right|\right\rfloor\vec{r}_{k-1}$$

注意到 $\vec{r}_k\times r=(q;p)\times(1;r)=qr-p$，因此

$$a_k=\left\lfloor\left|\frac{q_{k-1}r-p_{k-1}}{q_{k-2}r-p_{k-2}}\right|\right\rfloor$$

" 解释"

正如已经注意到的，$a_k = \lfloor s_k \rfloor$，其中 $s_k = [a_k; a_{k+1}, a_{k+2}, \dots]$。另一方面，从渐进分数的递推，可以得出

$$r = [a_0; a_1, \dots, a_{k-1}, s_k] = \frac{s_k p_{k-1} + p_{k-2}}{s_k q_{k-1} + q_{k-2}}$$

在向量形式中，它重写为

$$\vec{r} \parallel s_k \vec{r}_{k-1} + \vec{r}_{k-2}$$

这意味着 $\vec{r}$ 和 $s_k \vec{r}_{k-1} + \vec{r}_{k-2}$ 共线（即具有相同的斜率系数）。用 $\vec{r}$ 计算两个部分的外积，可以得到

$$0 = s_k (\vec{r}_{k-1} \times \vec{r}) + (\vec{r}_{k-2} \times \vec{r})$$

得出最终公式

$$s_k = -\frac{\vec{r}_{k-2} \times \vec{r}}{\vec{r}_{k-1} \times \vec{r}}$$

例题鼻子拉伸算法

每次将 $\vec{r}_{k-1}$ 添加到向量 $\vec{p}$ 时，$\vec{p} \times \vec{r}$ 的值都会增加 $\vec{r}_{k-1} \times \vec{r}$。

因此，$a_k = \lfloor s_k \rfloor$ 是 $\vec{r}_{k-1}$ 向量的最大整数，可以将其添加到 $\vec{r}_{k-2}$，而无需更改与 $\vec{r}$ 的外积的符号。

换句话说，a_k 是您可以将 $\vec{r}_{k-1}$ 添加到 $\vec{r}_{k-2}$ 的最大整数次数，而无需跨越 $\vec{r}$ 定义的线：

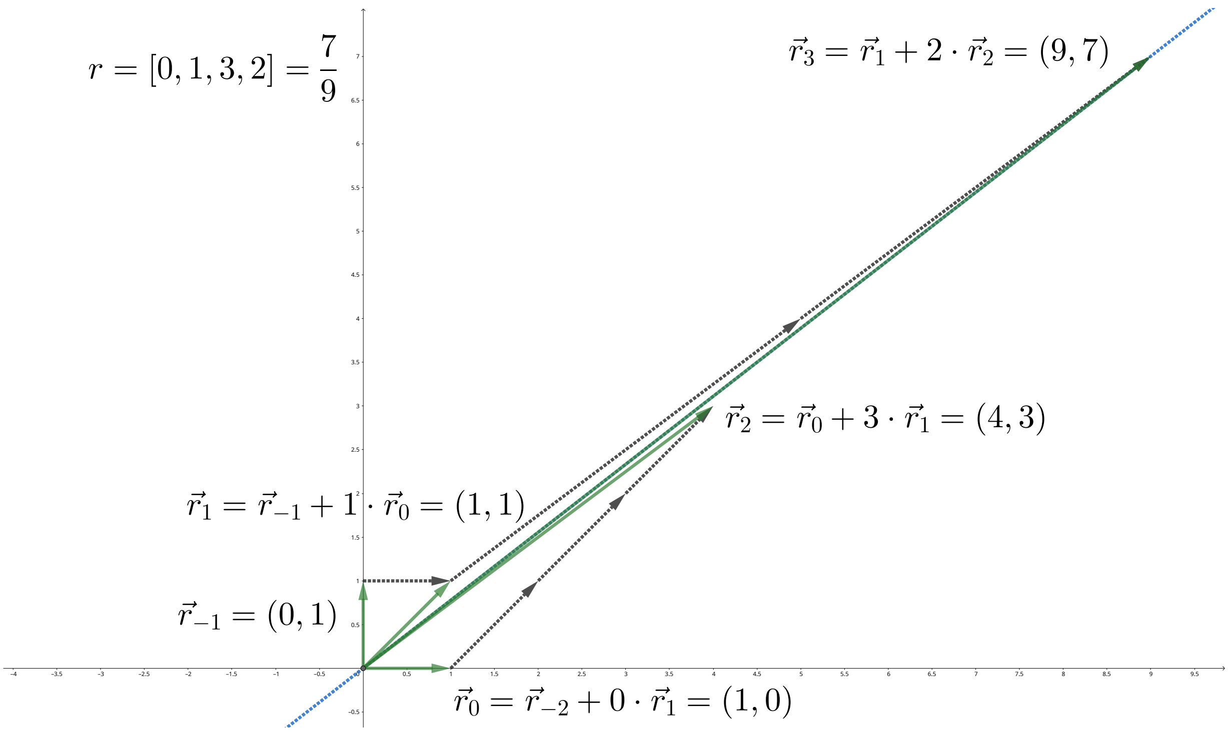


图 9.15

在上面的图片中，$\vec{r}_2 = (4; 3)$ 是通过将 $\vec{r}_1 = (1; 1)$ 重复添加到 $\vec{r}_0 = (1; 0)$ 而获得的。

当不可能在不跨越 $y = rx$ 线的情况下将 $\vec{r}_1$ 进一步添加到 $\vec{r}_0$ 时，转到另一侧，重复将 $\vec{r}_2$ 添加到 $\vec{r}_1$ 以获得 $\vec{r}_3 = (9; 7)$。

此过程生成接近直线的指数较长的向量。

对于这一特性，Boris Delaunay 将生成结果收敛向量的过程称为**鼻子拉伸算法**（Nose stretching algorithm）。

如果观察在点 $\vec{r}_{k-2}$、$\vec{r}_k$ 和 $\vec{0}$ 上绘制的三角形，会注意到它的加倍面积是

$$|\vec{r}_{k-2} \times \vec{r}_k| = |\vec{r}_{k-2} \times (\vec{r}_{k-2} + a_k \vec{r}_{k-1})| = a_k |\vec{r}_{k-2} \times \vec{r}_{k-1}| = a_k$$

结合 Pick 定理，这意味着三角形内部没有严格的格点，其边界上的唯一格点是 $\vec{0}$ 和 $\vec{r}_{k-2}+t\cdot\vec{r}_{k-1}$，对于所有整数 t，使得 $0\le t\le a_k$。当连接所有可能的 k 时，这意味着在由偶数索引和奇数索引收敛向量形成的多边形之间的空间中没有整数点。

这反过来意味着，具有奇数系数的 $\vec{r}_k$ 形成了线 $y=rx$ 上方 $x\ge 0$ 的格点凸包，而具有偶数系数的 $\vec{r}_k$ 形成线 $y=rx$ 下方 $x>0$ 的格点凸包。

定义

这些多边形也被称为**克莱因多边形**（Klein polygons），以费利克斯 · 克莱因（Felix Klein）的名字命名，他首次提出了对连续分数的几何解释。

例题

既然已经介绍了最重要的事实和概念，那么是时候深入研究具体的例题了。

线下凸包

找到格点 $(x;y)$ 的凸包，使得 $r=[a_0;a_1,\dots,a_k]=\frac{p_k}{q_k}$ 的 $0\le x\le N$ 和 $0\le y\le rx$。

" 解答"

如果我们考虑无界集合 $0\le x$，则上凸包将由线 $y=rx$ 本身给出。

然而，在附加约束 $x\le N$ 的情况下，最终需要偏离直线以保持适当的凸包。

设 $t=\lfloor\frac{N}{q_k}\rfloor$，则对于整数 $1\le\alpha\le t$，在 $(0;0)$ 之后的外壳上的第一个 t 格点是 $\alpha\cdot(q_k;p_k)$。

然而，$(t+1)(q_k;p_k)$ 不能是下一个格点，因为 $(t+1)q_k$ 大于 N。

为了到达外壳中的下一个格点，应该到达点 $(x;y)$，该点与 $y=rx$ 相差最小，同时保持 $x\le N$。

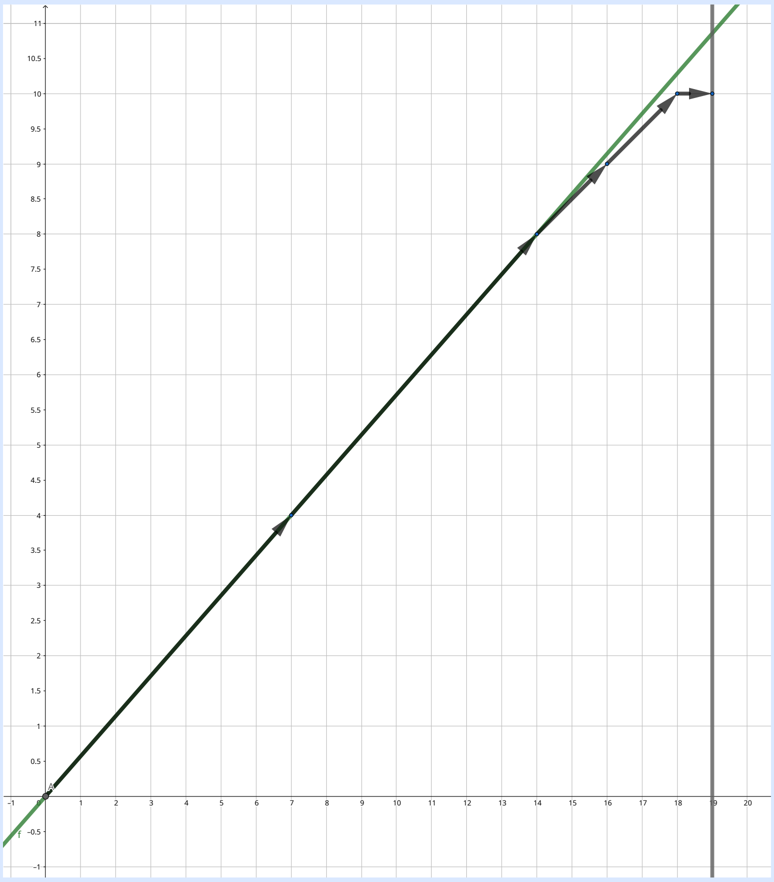


图 9.16

设 $(x;y)$ 为凸包中的最后一个当前点。然后，下一点 $(x';y')$ 是这样的：$x'\leq N$ 和 $(x';y')-(x;y)=(\Delta x;\Delta y)$ 尽可能接近线 $y=rx$。换句话说，$(\Delta x;\Delta y)$ 根据 $\Delta x\leq N-x$ 和 $\Delta y\leq r\Delta x$ 最大化 $r\Delta x-\Delta y$。

这样的点位于 $y=rx$ 以下的格点的凸包上。换句话说，$(\Delta x;\Delta y)$ 必须是 r 的下中间分数。

也就是说，对于某些奇数 i 和 $0\leq t<a_i$，$(\Delta x;\Delta y)$ 的形式为 $(q_{i-1};p_{i-1})+t\cdot(q_i;p_i)$。

要找到这样的 i，可以遍历所有可能的 i，从最大的一个开始，并对 i 使用 $t=\lfloor\frac{N-x-q_{i-1}}{q_i}\rfloor$，这样 $N-x-q_{i-1}\geq 0$。当 $(\Delta x;\Delta y)=(q_{i-1};p_{i-1})+t\cdot(q_i;p_i)$ 时，条件 $\Delta y\leq r\Delta x$ 由中间分数的性质保持。

并且 $t<a_i$ 成立，因为已经耗尽了从 $i+2$ 获得的半收敛，因此 $x+q_{i-1}+a_iq_i=x+q_{i+1}$ 大于 N。

现在，可以将 $(\Delta x;\Delta y)$ 添加到 $(x;y)$ 中 $k=\lfloor\frac{N-x}{\Delta x}\rfloor$ 次，然后再超过 N，之后将尝试下一个中间分数。

```cpp
// returns [ah, ph, qh] such that points r[i]=(ph[i], qh[i]) constitute upper
// convex hull of lattice points on 0 <= x <= N and 0 <= y <= r * x, where r =
// [a0; a1, a2, ...] and there are ah[i]-1 integer points on the segment between
// r[i] and r[i+1]
auto hull(auto a, int N) {
  auto [p, q] = convergents(a);
  int t = N / q.back();
  vector ah = {t};
  vector ph = {0, t * p.back()};
  vector qh = {0, t * q.back()};

  for (int i = q.size() - 1; i >= 0; i--) {
    if (i % 2) {
      while (qh.back() + q[i - 1] <= N) {
        t = (N - qh.back() - q[i - 1]) / q[i];
        int dp = p[i - 1] + t * p[i];
        int dq = q[i - 1] + t * q[i];
        int k = (N - qh.back()) / dq;
        ah.push_back(k);
        ph.push_back(ph.back() + k * dp);
        qh.push_back(qh.back() + k * dq);
      }
    }
  }
  return make_tuple(ah, ph, qh);
}
```

```python
# returns [ah, ph, qh] such that points r[i]=(ph[i], qh[i]) constitute upper con
vex hull
# of lattice points on 0 <= x <= N and 0 <= y <= r * x, where r = [a0; a1, a2, .
..]
# and there are ah[i]-1 integer points on the segment between r[i] and r[i+1]
def hull(a, N):
    p, q = convergents(a)
    t = N // q[-1]
    ah = [t]
    ph = [0, t*p[-1]]
    qh = [0, t*q[-1]]
    for i in reversed(range(len(q))):
        if i % 2 == 1:
            while qh[-1] + q[i-1] <= N:
                t = (N - qh[-1] - q[i-1]) // q[i]
                dp = p[i-1] + t*p[i]
```

```python
                dq = q[i-1] + t*q[i]
                k = (N - qh[-1]) // dq
                ah.append(k)
                ph.append(ph[-1] + k * dp)
                qh.append(qh[-1] + k * dq)
    return ah, ph, qh
```

Timus - Crime and Punishment[1]

您将得到整数 A、B 和 N。查找 $x \geq 0$ 和 $y \geq 0$，使 $Ax + By \leq N$ 和 $Ax + By$ 达到最大值。

" 解答"

在这个问题中有 $1 \leq A, B, N \leq 2 \cdot 10^9$，因此可以用 $O(\sqrt{N})$ 来解决。但是，有一个 $O(\log N)$ 解决方案包含连分数。

为了方便起见，通过替换 $x \mapsto \lfloor \frac{N}{A} \rfloor - x$ 来反转 x 的方向，因此需要找到点 $(x; y)$，使得 $0 \leq x \leq \lfloor \frac{N}{A} \rfloor$、$By - Ax \leq N \mod A$ 和 $By - Ax$ 是可能的最大值。每个 x 的最佳 y 值为 $\lfloor \frac{Ax + (N \bmod A)}{B} \rfloor$。

为了更一般地对待它，编写一个函数，该函数在 $0 \leq x \leq N$ 和 $y = \lfloor \frac{Ax+B}{C} \rfloor$ 上找到最佳点。

这个问题的核心解决方案思想基本上重复了前面的问题，但不是使用下中间分数来偏离直线，而是使用上中间分数来接近直线，而不跨越直线，也不违反 $x \leq N$。不幸的是，与前一个问题不同，您需要确保在靠近 $y = \frac{Ax+B}{C}$ 线时不会越过该线，因此在计算中间分数的系数 t 时应牢记这一点。

```python
# (x, y) such that y = (A*x+B) // C,
# Cy - Ax is max and 0 <= x <= N.
def closest(A, B, C, N):
    # y <= (A*x + B)/C <=> diff(x, y) <= B
    def diff(x, y):
        return C*y-A*x
    a = fraction(A, C)
    p, q = convergents(a)
    ph = [B // C]
    qh = [0]
    for i in range(2, len(q) - 1):
        if i % 2 == 0:
            while diff(qh[-1] + q[i+1], ph[-1] + p[i+1]) <= B:
                t = 1 + (diff(qh[-1] + q[i-1], ph[-1] + p[i-1]) - B - 1) // abs(
diff(q[i], p[i]))
                dp = p[i-1] + t*p[i]
                dq = q[i-1] + t*q[i]
                k = (N - qh[-1]) // dq
                if k == 0:
                    return qh[-1], ph[-1]
                if diff(dq, dp) != 0:
                    k = min(k, (B - diff(qh[-1], ph[-1])) // diff(dq, dp))
                qh.append(qh[-1] + k*dq)
                ph.append(ph[-1] + k*dp)
    return qh[-1], ph[-1]

def solve(A, B, N):
    x, y = closest(A, N % A, B, N // A)
    return N // A - x, y
```

June Challenge 2017 - Euler Sum[2]

计算 $\sum\limits_{x=1}^{N}\lfloor ex\rfloor$，其中 $e=[2;1,2,1,1,4,1,1,6,1,\dots,1,2n,1,\dots]$ 是自然对数的底，$N\le 10^{4000}$。

" 解答"

此和等于格点 $(x;y)$ 的数量，使得 $1\le x\le N$ 和 $1\le y\le ex$。

在构造了 $y=ex$ 以下的点的凸包之后，可以使用 Pick 定理计算这个数：

```cpp
// sum floor(k * x) for k in [1, N] and x = [a0; a1, a2, ...]
int sum_floor(auto a, int N) {
  N++;
  auto [ah, ph, qh] = hull(a, N);

  // The number of lattice points within a vertical right trapezoid
  // on points (0; 0) - (0; y1) - (dx; y2) - (dx; 0) that has
  // a+1 integer points on the segment (0; y1) - (dx; y2).
  auto picks = [](int y1, int y2, int dx, int a) {
    int b = y1 + y2 + a + dx;
    int A = (y1 + y2) * dx;
    return (A - b + 2) / 2 + b - (y2 + 1);
  };

  int ans = 0;
  for (size_t i = 1; i < qh.size(); i++) {
    ans += picks(ph[i - 1], ph[i], qh[i] - qh[i - 1], ah[i - 1]);
  }
  return ans - N;
}
```

```python
# sum floor(k * x) for k in [1, N] and x = [a0; a1, a2, ...]
def sum_floor(a, N):
    N += 1
    ah, ph, qh = hull(a, N)

    # The number of lattice points within a vertical right trapezoid
    # on points (0; 0) - (0; y1) - (dx; y2) - (dx; 0) that has
    # a+1 integer points on the segment (0; y1) - (dx; y2).
    def picks(y1, y2, dx, a):
        b = y1 + y2 + a + dx
        A = (y1 + y2) * dx
        return (A - b + 2) // 2 + b - (y2 + 1)

    ans = 0
    for i in range(1, len(qh)):
        ans += picks(ph[i-1], ph[i], qh[i]-qh[i-1], ah[i-1])
    return ans - N
```

NAIPC 2019 - It's a Mod, Mod, Mod, Mod World[3]

给定 p、q 和 n，计算 $\sum\limits_{i=1}^{n}[p\cdot i \bmod q]$。

" 解答"

如果您注意到 $a \bmod b = a - \lfloor \frac{a}{b} \rfloor b$，则此问题会减少到上一个问题。有了这个事实，总数减少到

$$\sum_{i=1}^{n} \left(p \cdot i - \left\lfloor \frac{p \cdot i}{q} \right\rfloor q\right) = \frac{pn(n+1)}{2} - q \sum_{i=1}^{n} \left\lfloor \frac{p \cdot i}{q} \right\rfloor$$

然而，将 x 从 1 到 N 的 $\lfloor rx \rfloor$ 相加，是我们能够从上一个问题中得出的结果。

```cpp
void solve(int p, int q, int N) {
  cout << p * N * (N + 1) / 2 - q * sum_floor(fraction(p, q), N) << "\n";
}
```

```python
def solve(p, q, N):
    return p * N * (N + 1) // 2 - q * sum_floor(fraction(p, q), N)
```

Library Checker - Sum of Floor of Linear[4]

给定 N、M、A 和 B，计算 $\sum\limits_{i=0}^{N-1} \lfloor \frac{A \cdot i + B}{M} \rfloor$。

"解答"

这是迄今为止技术上最麻烦的问题。

可以使用相同的方法来构造线 $y = \frac{Ax+B}{M}$ 以下的点的全凸包。

已经知道如何解决 $B = 0$ 的问题。此外，已经知道如何构造这个凸包，直到 $[0, N-1]$ 段上的这条线的最近格点（这在上面的「罪与罚」问题中完成）。

现在应该注意到，一旦到达了离直线最近的点，就可以假设直线实际上通过了最近的点。因为在实际直线和稍微向下移动以通过最近点的直线之间，$[0, N-1]$ 上没有其他格点。

也就是说，要在 $[0, N-1]$ 上的线 $y = \frac{Ax+B}{M}$ 下方构造全凸包，可以将其构造到与 $[0, N-1]$ 的线最近的点，然后继续，就像该线通过该点一样，重用用于构造 $B = 0$ 的凸包的算法：

```python
# hull of lattice (x, y) such that C*y <= A*x+B
def hull(A, B, C, N):
    def diff(x, y):
        return C*y-A*x
    a = fraction(A, C)
    p, q = convergents(a)
    ah = []
    ph = [B // C]
    qh = [0]

    def insert(dq, dp):
        k = (N - qh[-1]) // dq
        if diff(dq, dp) > 0:
            k = min(k, (B - diff(qh[-1], ph[-1])) // diff(dq, dp))
        ah.append(k)
        qh.append(qh[-1] + k*dq)
        ph.append(ph[-1] + k*dp)

    for i in range(1, len(q) - 1):
        if i % 2 == 0:
            while diff(qh[-1] + q[i+1], ph[-1] + p[i+1]) <= B:
                t = (B - diff(qh[-1] + q[i+1], ph[-1] + p[i+1])) // abs(diff(q[i
], p[i]))
```

```
                dp = p[i+1] - t*p[i]
                dq = q[i+1] - t*q[i]
                if dq < 0 or qh[-1] + dq > N:
                    break
                insert(dq, dp)

    insert(q[-1], p[-1])

    for i in reversed(range(len(q))):
        if i % 2 == 1:
            while qh[-1] + q[i-1] <= N:
                t = (N - qh[-1] - q[i-1]) // q[i]
                dp = p[i-1] + t*p[i]
                dq = q[i-1] + t*q[i]
                insert(dq, dp)
    return ah, ph, qh
```

OKC 2 - From Modular to Rational[5]

有一个有理数 $\frac{p}{q}$，即 $1 \le p, q \le 10^9$。您可以询问几个素数 m 的 pq^{-1} 模 $m \sim 10^9$ 的值。恢复 $\frac{p}{q}$。

这个问题等价于：查找 $1 \le x \le N$ 中，使 $Ax \bmod M$ 最小的 x。

" 解答"

根据中国剩余定理，要求结果模化几个素数与要求其模化其乘积是相同的。因此，在不丧失一般性的情况下，假设知道余数模足够大的数 m。

对于给定的余数 r，可能有几种可能的解决方案 (p, q) 到 $p \equiv qr \pmod m$。然而，如果 (p_1, q_1) 和 (p_2, q_2) 都是解，那么它也认为 $p_1 q_2 \equiv p_2 q_1 \pmod m$。假设 $\frac{p_1}{q_1} \ne \frac{p_2}{q_2}$，则意味着 $|p_1 q_2 - p_2 q_1|$ 至少为 m。

题面有 $1 \le p, q \le 10^9$，因此，如果 p_1, q_1 和 p_2, q_2 最多都是 10^9 的话，那么差额最多为 10^{18}。对于 $m > 10^{18}$，这意味着具有 $\frac{p}{q}$ 的解 $1 \le p, q \le 10^9$ 作为有理数是唯一的。

因此，问题归结为，给定 r 模 m，找到任何 q，使得 $1 \le q \le 10^9$ 和 $qr \mod m \le 10^9$。

这实际上与找到 $1 \le q \le 10^9$ 的 q 是相同的，该 q 提供了可能的最小 $qr \bmod m$。

对于 $qr = km + b$，这意味着需要找到一对 (q, m)，使得 $1 \le q \le 10^9$ 和 $qr - km \ge 0$ 是可能的最小值。

由于 m 是常量，可以除以它，并进一步将其重新表述为求解 q，这样 $1 \le q \le 10^9$ 和 $\frac{r}{m}q - k \ge 0$ 是可能的最小值。

就连分数而言，这意味着 $\frac{k}{q}$ 是 $\frac{r}{m}$ 的最佳丢番图近似值，并且仅检查 $\frac{r}{m}$ 的下中间分数就足够了。

```
# find Q that minimizes Q*r mod m for 1 <= k <= n < m
def mod_min(r, n, m):
    a = fraction(r, m)
    p, q = convergents(a)
    for i in range(2, len(q)):
        if i % 2 == 1 and (i + 1 == len(q) or q[i+1] > n):
            t = (n - q[i-1]) // q[i]
            return q[i-1] + t*q[i]
```

习题

- UVa OJ - Continued Fractions[6]
- ProjectEuler+ #64: Odd period square roots[7]
- Codeforces Round #184 (Div. 2) - Continued Fractions[8]
- Codeforces Round #201 (Div. 1) - Doodle Jump[9]

- Codeforces Round #325 (Div. 1) - Alice, Bob, Oranges and Apples[10]
- POJ Founder Monthly Contest 2008.03.16 - A Modular Arithmetic Challenge[11]
- 2019 Multi-University Training Contest 5 - fraction[12]
- SnackDown 2019 Elimination Round - Election Bait[13]

参考资料与注释

[1] Timus - Crime and Punishment

[2] June Challenge 2017 - Euler Sum

[3] NAIPC 2019 - It's a Mod, Mod, Mod, Mod World

[4] Library Checker - Sum of Floor of Linear

[5] OKC 2 - From Modular to Rational

[6] UVa OJ - Continued Fractions

[7] ProjectEuler+ #64: Odd period square roots

[8] Codeforces Round #184 (Div. 2) - Continued Fractions

[9] Codeforces Round #201 (Div. 1) - Doodle Jump

[10] Codeforces Round #325 (Div. 1) - Alice, Bob, Oranges and Apples

[11] POJ Founder Monthly Contest 2008.03.16 - A Modular Arithmetic Challenge

[12] 2019 Multi-University Training Contest 5 - fraction

[13] SnackDown 2019 Elimination Round - Election Bait

[14] Continued fractions

9.12.29 Stern–Brocot 树与 Farey 序列

连分数的树

有两种主要方法，可以将所有可能的连分数，合并为有用的树结构。

Stern–Brocot 树

定义

Stern–Brocot 树是一种维护分数的优雅的结构，包含所有不同的正有理数。这个结构由 Moritz Stern 在 1858 年和 Achille Brocot 在 1861 年发现。

解释

Stern–Borcot 树从两个简单的分数开始：

$$\frac{0}{1}, \frac{1}{0}$$

这个 $\frac{1}{0}$ 可能看得你有点懵逼。不过我们不讨论这方面的严谨性，你只需要把它当作 ∞ 就行了。

每次在相邻的两个分数 $\frac{a}{b}, \frac{c}{d}$ 中间插入一个分数 $\frac{a+c}{b+d}$，这样就完成了一次迭代，得到下一个序列。于是它就会变成这样

$$\frac{0}{1}, \quad \frac{1}{1}, \quad \frac{1}{0}$$

$$\frac{0}{1}, \quad \frac{1}{2}, \quad \frac{1}{1}, \quad \frac{2}{1}, \quad \frac{1}{0}$$

$$\frac{0}{1}, \quad \frac{1}{3}, \quad \frac{1}{2}, \quad \frac{2}{3}, \quad \frac{1}{1}, \quad \frac{3}{2}, \quad \frac{2}{1}, \quad \frac{3}{1}, \quad \frac{1}{0}$$

既然称它为 Stern–Brocot 树，那么它总得有一个树的样子。来一张图：

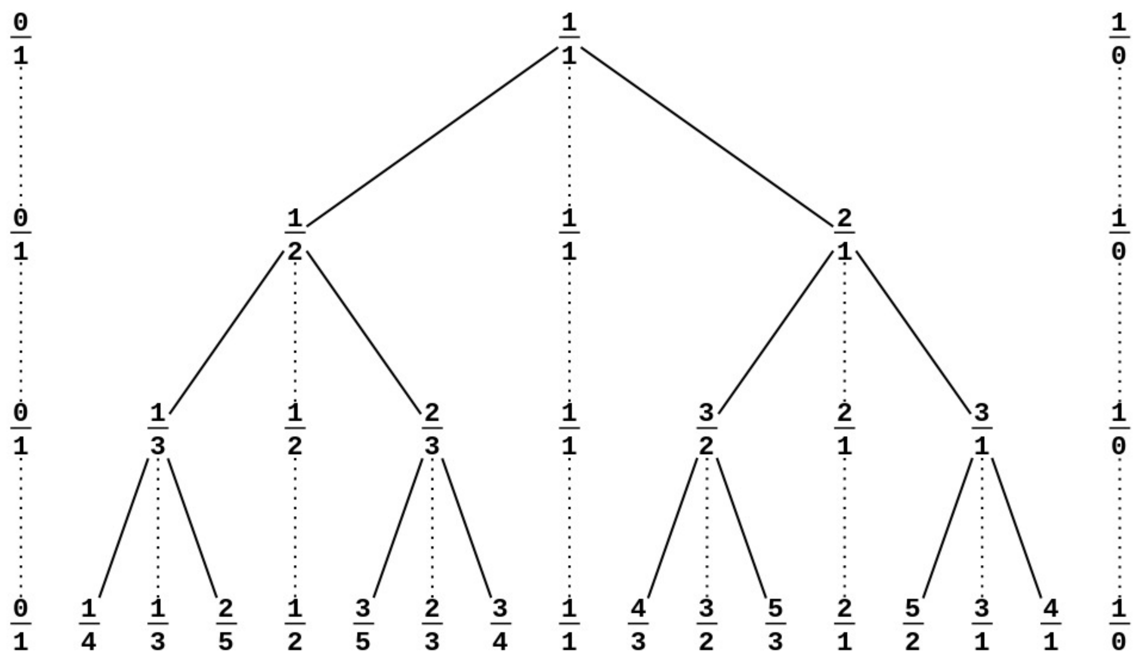


图 9.17 pic

可以把第 i 层的序列当作是深度为 $i-1$ 的 Stern–Brocot 树的中序遍历。

性质

接下来讨论 Stern–Brocot 树的性质。

单调性

在每一层的序列中，真分数是单调递增的。

略证：只需要在 $\frac{a}{b} \le \frac{c}{d}$ 的情况下证明

$$\frac{a}{b} \le \frac{a+c}{b+d} \le \frac{c}{d}$$

就行了。这个很容易，直接做一下代数变换即可

$$\begin{aligned} & \frac{a}{b} \le \frac{c}{d} \\ \implies & ad \le bc \\ \implies & ad + ab \le bc + ab \\ \implies & \frac{a}{b} \le \frac{a+c}{b+d} \end{aligned}$$

另一边同理可证。

最简性

序列中的分数（除了 $\frac{0}{1}, \frac{1}{0}$）都是最简分数。

略证：为证明最简性，我们首先证明对于序列中连续的两个分数 $\frac{a}{b}, \frac{c}{d}$：

$$bc - ad = 1$$

显然，只需要在 $bc - ad = 1$ 的条件下证明 $\frac{a}{b}, \frac{a+c}{b+d}, \frac{c}{d}$ 的情况成立即可。

$$a(b+d) - b(a+c) = ad - bc = 1$$

后半部分同理。证明了这个，利用扩展欧几里德定理，如果上述方程有解，显然 $\gcd(a,b) = \gcd(c,d) = 1$。这样就证完了。

有了上面的证明，可以证明 $\frac{a}{b} < \frac{c}{d}$。

有了这两个性质，就可以把它当成一棵平衡树来做了。建立和查询就向平衡树一样做就行了。

连分数表示与父子节点

递推 $\frac{p_k}{q_k} = \frac{a_k p_{k-1} + p_{k-2}}{a_k q_{k-1} + q_{k-2}}$，意味着连分数表示对树中 $\frac{p_k}{q_k}$ 的路径进行编码。要查找 $[a_0; a_1, \dots, a_k, 1]$，必须使 a_0 向右移动，a_1 向左移动，a_2 向右移动等等，直到 a_k。

连分数节点 $[a_0; a_1, \dots, a_k, 1]$ 的父节点是沿最后使用的方向后退一步获得的分数。

换句话说，当 $a_k > 1$ 时，它是 $[a_0; a_1, \dots, a_k - 1, 1]$，而当 $a_k = 1$ 时，则是 $[a_0; a_1, \dots, a_{k-1}, 1]$。

因此，$[a_0; a_1, \dots, a_k, 1]$ 的子节点是 $[a_0; a_1, \dots, a_k + 1, 1]$ 和 $[a_0; a_1, \dots, a_k, 1, 1]$。

索引

为 Stern Brocot 树编制索引。根顶点被分配了一个索引 1。然后，对于顶点 v，通过将 v 的前导位从 1 更改为 10 来分配其左子节点的索引，对于右子节点，通过将前导位从 1 更改为 11 来分配索引：

在这种索引中，连分数表示规定了有理数的**游程长度编码**（run-length encoding）。

对于 $\frac{5}{2} = [2; 2] = [2; 1, 1]$，其索引为 1011_2，其游程长度编码（考虑按升序排列的位）为 $[2; 1, 1]$。

另一个例子是 $\frac{2}{5} = [0; 2, 2] = [0; 2, 1, 1]$，其索引为 1100_2，其游程编码实际上为 $[0; 2, 2]$。

值得注意的是，Stern–Brocot 树实际上是一个堆。也就是说，它是 $\frac{p}{q}$ 的二叉树，它也是 p 和 q 的堆。

实现

构建实现

```
void build(int a = 0, int b = 1, int c = 1, int d = 0, int level = 1) {
  int x = a + c, y = b + d;
  // ... output the current fraction x/y
  // at the current level in the tree
  build(a, b, x, y, level + 1);
  build(x, y, c, d, level + 1);
```

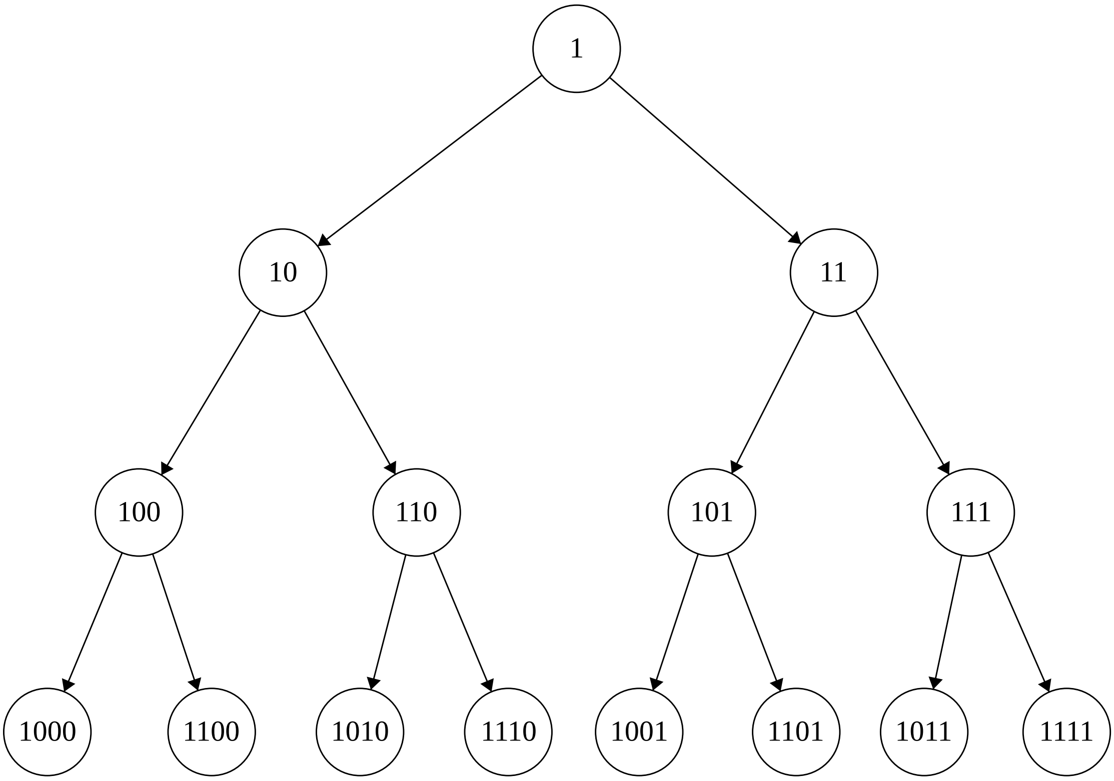


图 9.18 pic

```
}
```

```python
def build(a = 1, b = 1, c = 1, d = 0, level = 1):
    x = a + c; y = b + d
    # ... output the current fraction x/y
    # at the current level in the tree
    build(a, b, x, y, level + 1)
    build(x, y, c, d, level + 1)
```

查询实现

```cpp
string find(int x, int y, int a = 0, int b = 1, int c = 1, int d = 0) {
  int m = a + c, n = b + d;
  if (x == m && y == n) return "";
  if (x * n < y * m)
    return 'L' + find(x, y, a, b, m, n);
  else
    return 'R' + find(x, y, m, n, c, d);
}
```

```python
def find(x, y, a = 0, b = 1, c = 1, d = 0):
    m = a + c; n = b + d
    if x == m and y == n:
        return ""
    if x * n < y * m:
        return 'L' + find(x, y, a, b, m, n)
    else:
```

```
        return 'R' + find(x, y, m, n, c, d)
```

例题

例题" 比较连分数的大小"

对于 $A = [a_0; a_1, \dots, a_n]$ 和 $B = [b_0; b_1, \dots, b_m]$，哪个分数更小?

" 解答"

假设 A 和 B 是无理数，它们的连分数表示是 Stern–Brocot 树中的无限下降。

正如已经提到的，在这个表示中，a_0 表示下降中右转的次数，a_1 表示随后左转的次数，依此类推。因此，当比较 a_k 和 b_k 时，如果 $a_k = b_k$，应该继续比较 a_{k+1} 和 b_{k+1}。否则，如果在右下降，应该检查是否 $a_k < b_k$；如果在左下降，应检查 $a_k > b_k$，以判断 $a < b$。

换言之，对于无理数 A 和 B，当且仅当字典序（lexicographical comparison）有 $(a_0, -a_1, a_2, -a_3, \dots) < (b_0, -b_1, b_2, -b_3, \dots)$ 时，将是 $A < B$。

现在，正式使用 ∞ 作为连分数表示的元素，可以模拟无理数 $A - \varepsilon$ 和 $A + \varepsilon$，即比 A 小（大）但比任何其他实数大（小）的元素。具体而言，对于 $A = [a_0; a_1, \dots, a_n]$，这两个元素中的一个可以模拟 $[a_0; a_1, \dots, a_n, \infty]$，另一个可以模拟 $[a_0; a_1, \dots, a_n - 1, 1, \infty]$。

哪一个对应于 $A - \varepsilon$，哪一个对应于 $A + \varepsilon$，可以通过 n 的奇偶性或通过将它们作为无理数进行比较来确定。

```
# check if a < b assuming that a[-1] = b[-1] = infty and a != b
def less(a, b):
    a = [(-1)**i*a[i] for i in range(len(a))]
    b = [(-1)**i*b[i] for i in range(len(b))]
    return a < b

# [a0; a1, ..., ak] -> [a0, a1, ..., ak-1, 1]
def expand(a):
    if a: # empty a = inf
        a[-1] -= 1
        a.append(1)
    return a

# return a-eps, a+eps
def pm_eps(a):
    b = expand(a.copy())
    a.append(float('inf'))
    b.append(float('inf'))
    return (a, b) if less(a, b) else (b, a)
```

例题" 最佳内点"

对于 $\frac{0}{1} \le \frac{p_0}{q_0} < \frac{p_1}{q_1} \le \frac{1}{0}$，找到有理数 $\frac{p}{q}$ 使得 $(q; p)$ 在字典序最小，并且 $\frac{p_0}{q_0} < \frac{p}{q} < \frac{p_1}{q_1}$。

" 解答"

就 Stern–Brocot 树而言，这意味着需要找到 $\frac{p_0}{q_0}$ 和 $\frac{p_1}{q_1}$ 的 LCA。由于 Stern–Brocot 树和连分数之间的联系，该 LCA 将大致对应于 $\frac{p_0}{q_0}$ 和 $\frac{p_1}{q_1}$ 的连分数表示的最大公共前缀。

因此，如果 $\frac{p_0}{q_0} = [a_0; a_1, \dots, a_{k-1}, a_k, \dots]$ 和 $\frac{p_1}{q_1} = [a_0; a_1, \dots, a_{k-1}, b_k, \dots]$ 是无理数，则 LCA 为 $[a_0; a_1, \dots, \min(a_k, b_k) + 1]$。

对于有理数 r_0 和 r_1，其中之一可能是 LCA 本身，这需要对其进行讨论。为了简化有理数 r_0 和 r_1 的解决

方案，可以使用前面问题中导出的 $r_0+\varepsilon$ 和 $r_1-\varepsilon$ 的连分数表示。

```
# finds lexicographically smallest (q, p)
# such that p0/q0 < p/q < p1/q1
def middle(p0, q0, p1, q1):
    a0 = pm_eps(fraction(p0, q0))[1]
    a1 = pm_eps(fraction(p1, q1))[0]
    a = []
    for i in range(min(len(a0), len(a1))):
        a.append(min(a0[i], a1[i]))
        if a0[i] != a1[i]:
            break
    a[-1] += 1
    p, q = convergents(a)
    return p[-1], q[-1]
```

例题"GCJ 2019, Round 2 - New Elements: Part 2[1]"

您得到 N 个正整数对 (C_i,J_i)。您需要找到一个正整数对 (x,y)，这样 C_ix+J_iy 就是一个严格递增的序列。在这类配对中，找到词典中最小的一对。

" 解答"

重新表述语句，A_ix+B_iy 对于所有 i 都必须为正，其中 $A_i=C_i-C_{i-1}$，$B_i=J_i-J_{i-1}$。

在这些方程中，对于 $A_ix+B_iy>0$，有四个情况：

1. $A_i,B_i>0$ 可以忽略，因为正在查找 $x,y>0$。
2. $A_i,B_i\le 0$ 将提供「IMPOSSIBLE」作为答案。
3. $A_i>0,B_i\le 0$。这样的约束相当于 $\frac{y}{x}<\frac{A_i}{-B_i}$。
4. $A_i\le 0,B_i>0$。这样的约束相当于 $\frac{y}{x}>\frac{-A_i}{B_i}$。

让 $\frac{p_0}{q_0}$ 是第四组中最大的 $\frac{-A_i}{B_i}$，而 $\frac{p_1}{q_1}$ 则是第三组中最小的 $\frac{A_i}{-B_i}$。

现在的问题是，给定 $\frac{p_0}{q_0}<\frac{p_1}{q_1}$，找到一个分数 $\frac{p}{q}$ 使得 $(q;p)$ 在字典上最小，并且 $\frac{p_0}{q_0}<\frac{p}{q}<\frac{p_1}{q_1}$。

```
def solve():
n = int(input())
C = [0] * n
J = [0] * n
# p0/q0 < y/x < p1/q1
p0, q0 = 0, 1
p1, q1 = 1, 0
fail = False
for i in range(n):
    C[i], J[i] = map(int, input().split())
    if i > 0:
        A = C[i] - C[i-1]
        B = J[i] - J[i-1]
        if A <= 0 and B <= 0:
            fail = True
        elif B > 0 and A < 0: # y/x > (-A)/B if B > 0
            if (-A)*q0 > p0*B:
                p0, q0 = -A, B
        elif B < 0 and A > 0: # y/x < A/(-B) if B < 0
```

```
                if A*q1 < p1*(-B):
                    p1, q1 = A, -B
    if p0*q1 >= p1*q0 or fail:
        return 'IMPOSSIBLE'

    p, q = middle(p0, q0, p1, q1)
    return str(q) + ' ' + str(p)
```

Calkin–Wilf 树

定义

在二叉树中组织连分数的一种更简单的方法是 Calkin–Wilf 树。

通常如下所示：

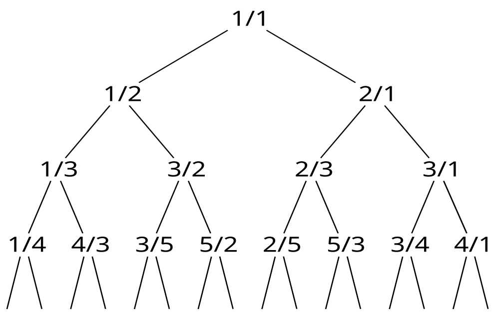


图 9.19 pic

树的根节点为 $\frac{1}{1}$。然后，对于数字为 $\frac{p}{q}$ 的顶点，其子节点为 $\frac{p}{p+q}$ 和 $\frac{p+q}{q}$。

性质

与 Stern–Brocot 树不同，Calkin–Wilf 树不是二叉搜索树，因此不能用于执行有理二叉搜索。

在 Calkin–Wilf 树中，当 $p > q$ 时，分数 $\frac{p}{q}$ 的直接父节点为 $\frac{p-q}{q}$；当 $p < q$ 时，为 $\frac{p}{q-p}$。

在 Stern–Brocot 树中使用了收敛的递归。为了得出连分数和 Calkin–Wilf 树之间的联系，应该使用完整商（complete quotients）的递归。如果 $s_k = \frac{p}{q}$，则 $s_{k+1} = \frac{q}{p \mod q} = \frac{q}{p-\lfloor p/q \rfloor \cdot q}$。

另一方面，当 $p > q$ 时，在 Calkin–Wilf 树中重复从 $s_k = \frac{p}{q}$ 到它的父节点，那么将以 $\frac{p \mod q}{q} = \frac{1}{s_{k+1}}$ 结尾。如果继续这样做，将以 s_{k+2}，然后 $\frac{1}{s_{k+3}}$ 等结尾。由此可以推断：

1. 当 $a_0 > 0$ 时，Calkin–Wilf 树中 $[a_0; a_1, \dots, a_k]$ 的直接父节点为 $\frac{p-q}{q} = [a_0 - 1; a_1, \dots, a_k]$。
2. 当 $a_0 = 0$ 且 $a_1 > 1$ 时，其直接父节点为 $\frac{p}{q-p} = [0; a_1 - 1, a_2, \dots, a_k]$。
3. 当 $a_0 = 0$ 且 $a_1 = 1$ 时，其直接父节点为 $\frac{p}{q-p} = [a_2; a_3, \dots, a_k]$。

相应地，$\frac{p}{q} = [a_0; a_1, \dots, a_k]$ 的子节点为

1. $\frac{p+q}{q} = 1 + \frac{p}{q}$，即 $[a_0 + 1; a_1, \dots, a_k]$。
2. $\frac{p}{p+q} = \frac{1}{1+\frac{q}{p}}$，对于 $a_0 > 0$，它是 $[0, 1, a_0, a_1, \dots, a_k]$；对于 $a_0 = 0$，则是 $[0, a_1 + 1, a_2, \dots, a_k]$。

值得注意的是，如果以广度优先搜索顺序枚举 Calkin–Wilf 树的顶点（即，根有一个数字 1，而顶点 v 的子节点有相应的索引 $2v$ 和 $2v + 1$），Calkin–Welf 树中有理数的索引将与 Stern–Brocot 树中的索引相同。

因此，Stern–Brocot 树和 Calkin–Wilf 树的相同级别上的数字是相同的，但它们的排序通过**位反转排列**（bit-reversal permutation）而不同。

Farey 序列

定义

Stern–Brocot 树与 Farey 序列有着极其相似的特征。第 i 个 Farey 序列记作 F_i，表示把分母小于等于 i 的所有最简真分数按大小顺序排列形成的序列。

$$F_1 = \left\{ \frac{0}{1}, \frac{1}{1} \right\}$$

$$F_2 = \left\{ \frac{0}{1}, \frac{1}{2}, \frac{1}{1} \right\}$$

$$F_3 = \left\{ \frac{0}{1}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{1}{1} \right\}$$

$$F_4 = \left\{ \frac{0}{1}, \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \frac{1}{1} \right\}$$

$$F_5 = \left\{ \frac{0}{1}, \frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \frac{1}{1} \right\}$$

显然，上述构建 Stern–Brocot 树的算法同样适用于构建 Farey 序列。因为 Stern–Brocot 树中的数是最简分数，因此在边界条件（分母）稍微修改一下就可以形成构造 Farey 序列的代码。可以认为 Farey 序列 F_i 是 Stern–Brocot 第 $i-1$ 次迭代后得到的序列的子序列。

Farey 序列同样满足最简性和单调性，并且满足一个与 Stern–Brocot 树相似的性质：对于序列中连续的三个数 $\frac{a}{b}, \frac{x}{y}, \frac{c}{d}$，有 $x = a + c, y = b + d$。这个可以轻松证明，不再赘述。

由 Farey 序列的定义，可以得到 F_i 的长度 L_i 公式为：

$$L_i = L_{i-1} + \varphi(i)$$
$$L_i = 1 + \sum_{k=1}^{i} \varphi(k)$$

中间分数

对于 $0 \le n \le a_{k+1}$，记中间分数为

$$F_{k,n} = \frac{np_k + p_{k-1}}{nq_k + q_{k-1}}$$

中间分数还包含这样的分数：

$$F_{0,n} = \frac{np_0 + p_{-1}}{nq_0 + q_{-1}} = \frac{na_0 + 1}{n} = a_0 + \frac{1}{n}$$

这些中间分数介于 $\frac{p_{-1}}{q_{-1}}$ 与 $\frac{p_0}{q_0} = a_0$ 之间。

对于给定的 k，第 0 个中间分数 $F_{k,0} = \frac{p_{k-1}}{q_{k-1}}$ 与最后一个中间分数 $F_{k,0} = \frac{p_{k-1}}{q_{k-1}}$ 是渐进分数。

对比上述表达式与

$$x = \frac{r_{k+1}p_k + p_{k-1}}{r_{k+1}q_k + q_{k-1}}$$

可以得到 $F_{k,n} = [a_0, \ldots, a_k, n]$。因此，中间分数也是某个连分数展开的渐进分数，中间分数表达式的分子分母也一定互素。

两个相邻中间分数的差为

$$F_{k,n+1} - F_{k,n} = \frac{(p_{k-1}q_k - p_k q_{k-1})n + (p_k q_{k-1} - p_{k-1}q_k)(n+1)}{((n+1)q_k + q_{k-1})(nq_k + q_{k-1})} = \frac{(-1)^{k-1}}{((n+1)q_k + q_{k-1})(nq_k + q_{k-1})}$$

对于确定的 k，中间分数随着 n 的增加递增或递减。

定理：对实数 x 与渐进分数 $\frac{p_k}{q_k}$，有

$$\frac{1}{q_k(q_k+q_{k+1})} < \left|x-\frac{p_k}{q_k}\right| < \frac{1}{q_kq_{k+1}}$$

" 证明"

一种证法可以定量计算。根据连分数中的定理有

$$\left|x-\frac{p_k}{q_k}\right| = \frac{1}{q_k(r_{k+1}q_k+q_{k-1})}$$

由于

$$a_{k+1} < \lfloor r_{k+1} \rfloor < a_{k+1}+1$$

因此

$$q_{k+1}=a_{k+1}q_k+q_{k-1} < r_{k+1}q_k+q_{k-1} < a_{k+1}q_k+q_k+q_{k-1}=q_k+q_{k+1}$$

取倒数即证毕。

另一种证法使用中间分数。首先，两个相邻渐进分数之间的距离有

$$\left|\frac{p_{k+1}}{q_{k+1}}-\frac{p_k}{q_k}\right| = \frac{1}{q_{k+1}q_k}$$

由于 x 本身介于两个相邻渐进分数之间，到其中一个渐进分数的距离 $\left|x-\frac{p_k}{q_k}\right|$ 一定小于 $\frac{1}{q_{k+1}q_k}$。不等式右边即证毕。

对于不等式左边，由于对于固定的 k，两头的中间分数就是渐进分数，因此有位置关系

$$\frac{p_{k-1}}{q_{k-1}} = F_{k,0}\ Fk,1\ \frac{p_{k+1}}{q_{k+1}}\ x\ \frac{p_k}{q_k}$$

因此有距离关系

$$\left|x-\frac{p_{k-1}}{q_{k-1}}\right| > |F_{k,0}-F_{k,1}| = \frac{1}{q_{k-1}(q_k+q_{k+1})}$$

更换下标，不等式左边即证毕。

在使用中这个定理经常放缩一下。由于 $q_k \le q_{k+1}$，有

$$\frac{1}{2{q_{k+1}}^2} < \frac{1}{q_k(q_k+q_{k+1})} < \left|x-\frac{p_k}{q_k}\right| < \frac{1}{q_kq_{k+1}} < \frac{1}{{q_k}^2}$$

右半部分 $\left|x-\frac{p_k}{q_k}\right| < \frac{1}{{q_k}^2}$ 无需计算更高阶的渐进分数，很常用，即下文 Dirichlet 逼近定理的连分数证法。

Dirichlet 逼近定理

Dirichlet 逼近定理是说，对于任意的一个无理数 θ，均能找到无穷个有理数逼近它，满足不等式：$\left|\frac{p}{q}-\theta\right| \leqslant \frac{1}{q^2}$

由于任一实数 qx 到最近的整数距离不超过 $\frac{1}{2}$，显然存在整数 p 和 q 使得

$$|qx-p| \le \frac{1}{2}$$

$$\left|x-\frac{p}{q}\right| \le \frac{1}{2q}$$

Dirichlet 逼近定理将右侧优化到了 $\frac{1}{q^2}$。等价的写法，$|qx-p| < \frac{1}{q}$ 总有解。

当 x 是无理数的时候，可以让渐进分数的分母任意大，$|qx-p|$ 任意小。

另外还有瓦伦定理：实数 x 连续两个渐进分数至少有一个满足 $\left|x-\frac{p}{q}\right|<\frac{1}{2q^2}$，由瓦伦（Vahlen）在 1895 年证明。

另外还有伯雷尔定理：实数 x 连续两个渐进分数至少有一个满足 $\left|x-\frac{p}{q}\right|<\frac{1}{\sqrt{5}q^2}$，由伯雷尔（Borel）在 1903 年证明。

这个 $\sqrt{5}$ 是最优的，在无理数为例如黄金分割 $\frac{1+\sqrt{5}}{2}=[1,1,1,...]$ 等连分数展开自某位起均为 1 的时候取等。这些后续的定理也同样表明，在 Dirichlet 逼近定理中的小于等于号事实上也可以是小于号。

另外还有定理：实数 x 连分数展开无穷项不小于 n，则存在无穷多渐进分数满足 $\left|x-\frac{p}{q}\right|<\frac{1}{\sqrt{n^2+4}q^2}$。这种条件的极端情形例如 $\frac{n+\sqrt{n^2+4}}{2}=[n,n,n,...]$。

其他的，还有 Kronecker 逼近定理：

如果 θ 为无理数，α 为实数，则对任意正数 ε, 存在整数 n 和整数 m，使得：

$|n\theta-m-\alpha|<\varepsilon$

Dirichlet 逼近定理和 Kronecker 逼近定理，都可以用抽屉原理来解决。事实上，正是 Dirichlet 为了解决 Pell 方程，研究有理数逼近条件，抽屉原理才在历史上被第一次正式提出。

当然，采用抽屉原理的证明可以发现，下文中提到的最佳逼近有理数列，每项满足定理中右边改成分母为一次式的不等式。

进一步有结论，渐进有理数列中，每一项均满足 Dirichlet 逼近定理的不等式。

最佳有理逼近

讨论如何用有理数「最佳地」逼近无理数，不妨假设无理数落入 $(0,1)$ 区间。

「最佳」一词的概念：选定的有理数必须保证，比它与无理数的距离更近的有理数，分母都比它大。不存在分母比它小的有理数，到给定无理数的距离更近。

最佳有理数：在 Farey 数列的某一行中，与给定无理数距离最近的那个有理数。

这个有理数可能在下面几行依旧与无理数「距离最近」，但一定有某一行，会找到一个新的有理数，与无理数距离更近。因此去重复后可以得到**最佳逼近有理数列**，分母严格递增，距离递减。

更加严谨的叙述为：

对实数 x 和有理数 $\frac{p}{q}$，如果对于任意的整数 a 和整数 $0<b\leq q$，$\frac{a}{b}\neq\frac{p}{q}$，均有

$$\left|x-\frac{a}{b}\right|>\left|x-\frac{p}{q}\right|$$

则称 $\frac{p}{q}$ 是 x 的一个最佳有理逼近。

比无理数大的称为上逼近，否则为下逼近。由于无理数和有理数之间一定有有理数，最佳逼近有理数列必然为若干个上逼近，之后若干个下逼近，交替进行的形式。

有结论：

渐进分数必然为最佳有理逼近。

偶项渐进分数全都是下逼近，奇项渐进分数全都是上逼近。渐进分数列是下上交错的逼近。

在最佳逼近有理数列中，渐进分数是下上关系改变之前的倒数第一个数。如果将最佳逼近有理数列都写成有限简单连分数，那么在渐进分数之后（下上关系改变之后），连分数长度加一。

例如，$[0,2,4,2]$ 是 $\sqrt{6}$ 减去 2 的一个渐进分数。

那么它的渐进分数列是：

$$[0,2]=\frac{1}{2},[0,2,4]=\frac{4}{9},[0,2,4,2]=\frac{9}{20}\cdots$$

它的最佳逼近有理数列是：

$$[0,1]=1,[0,2]=\frac{1}{2}$$

$$[0,2,1]=\frac{1}{3},[0,2,2]=\frac{2}{5},[0,2,3]=\frac{3}{7},[0,2,4]=\frac{4}{9}$$

$$[0,2,4,1]=\frac{5}{11},[0,2,4,2]=\frac{9}{20},...$$

从每个渐进分数（不包含）开始，到下一个渐进分数（包含）为止，同为上逼近，或同为下逼近。

在最佳逼近列中，每一个最佳分数是上一个最佳分数与再往前一个渐进分数的分子分母对应求和。

性质

定理：最佳有理逼近一定是中间分数。

它的逆定理并不成立，中间分数不一定是最佳有理逼近。

" 证明"

首先，a_0 与 a_0+1 中必有一个是 x 的分母为 1 的最佳有理逼近。它们分别是 $F_{1,0}$ 和 $F_{0,1}$。其他分母更大的第一类最优逼近肯定介于这两个数中间。

当然，除非 $x=n+\frac{1}{2}$ 为半奇数。这时两边距离一样，x 没有分母为 1 的最佳有理逼近，不过这属于连分数太短的情况。

当 n 增加时，渐近分数从 $F_{1,0}$ 和 $F_{0,1}$ 两边向中间排布。由于 $F_{k,a_{k+1}}=F_{k+2,0}$，有分布

$$F_{1,0}<F_{1,1}\leq F_{1,a_2}=F_{3,0}<...<x<...<F_{2,0}=F_{0,a_1}\leq F_{0,1}$$

因为中间分数可以任意接近 x，从而任何一个不是中间分数的有理数 $\frac{a}{b}$，一定介于两个同阶的渐近分数 F*{k,r} 与 F*{k,r+1} 之间。其位置分布情况为：

$$F_{k,r}\ \frac{a}{b}\ F_{k,r+1}\ x$$

于是有

$$\left|F_{k,r}-\frac{a}{b}\right|<|F_{k,r+1}-F_{k,r}|=\frac{1}{((r+1)q_k+q_{k-1})(rq_k+q_{k-1})}$$

另一方面，$\frac{a}{b}$ 与中间分数 $F_{k,r}$，有

$$\left|F_{k,r}-\frac{a}{b}\right|=\frac{|a(rq_k+q_{k-1})-b(rp_k+p_{k-1})|}{b(rq_k+q_{k-1})}\geq\frac{1}{b(rq_k+q_{k-1})}$$

因此有 $b>(r+1)q_k+q_{k-1}$。介于 $F_{k,r}$ 和 $F_{k,r+1}$ 之间的有理数，分母一定大于 $F_{k,r+1}$ 的分母。由位置分布可以看出

$$\left|x-\frac{a}{b}\right|>|x-F_{k,r+1}|$$

分母更小的分数 $F_{k,r+1}$ 到 x 的距离更近。因此，非中间分数的 $\frac{a}{b}$ 不可能是最佳有理逼近。证毕。

渐进分数的等价性质

实数 x 的渐进分数 $\frac{p}{q}$ 等价于这样的一类分数：

对于任意的整数 a 和整数 $0<b\leq q$，$\frac{a}{b}\neq\frac{p}{q}$，均有

$$|bx-a|>|qx-p|$$

也就是

$$\left|x-\frac{a}{b}\right|>\frac{q}{b}\left|x-\frac{p}{q}\right|$$

根据等价性，这可以直接作为渐进分数的另一个定义方法。这个性质比最佳逼近更加严格，因此根据这个性质，渐进分数必然为最佳逼近。

证明：

首先有 $|q_kx-p_k|<\dfrac{1}{q_{k+1}}$。对于一个非渐近分数的中间分数 $F_{k,n}=\dfrac{a}{b}$，有位置分布

$$\frac{p_{k-1}}{q_{k-1}}\ \frac{a}{b}\ \frac{p_{k+1}}{q_{k+1}}\ x\ \frac{p_k}{q_k}$$

于是有

$$\left|x-\frac{a}{b}\right|>\left|\frac{p_{k+1}}{q_{k+1}}-\frac{a}{b}\right|$$

即 $|bx-a|\geq\dfrac{1}{q_{k+1}}$。由于 $b=nq_k+q_{k-1}>q_k$，因此 $F_{k,n}=\dfrac{a}{b}$ 劣于比它分母更小的 $\dfrac{p_k}{q_k}$。

接下来证明，满足这个条件的都是渐进分数。

首先，越高阶的渐近分数越靠近 x，具有严格单调性。如果有 $s<k$，就有 $|q_sx-p_s|>|q_kx-p_k|$。

这是因为 $q_{k+1}\geq q_s+q_{s+1}$，有

$$|q_sx-p_s|>\frac{1}{q_s+q_{s+1}}\leq\frac{1}{q_{k+1}}>|q_kx-p_k|$$

整理即为上式。

接下来找出分母为 q_k 的满足上述性质的数，并指出它是渐进分数。

在给定 b 的时候，$\min|bx-a|\leq\dfrac{1}{2}$，并且取到最小的 a 最多只有两个取值。

当 b 在范围 $0<b\leq q_k$ 遍历的时候，将上述的最小值继续比较其中的最小值，并记取到最小值中的最小值的多个 b 中，最小的 b 为 b_0，对应的 a 为 a_0。

这里的 a_0 至多有两个，而事实上只有一个，可以证明其唯一性。唯一性的证明补在最后。

从上述选取方式可以看出 $\dfrac{a_0}{b_0}$ 也满足定理的条件，因此它是一个渐进分数 $\dfrac{p_s}{q_s}$。

又根据严格单调性，在 $s<k$ 时无法取到最小，只能有 $s=k$。

至此距离证毕只剩下 a_0 的唯一性。

如果给定 b_0，取到 $|b_0x-a|$ 的 a 不唯一，而是相邻的两个数，只能有 $|b_0x-a|=\dfrac{1}{2}$。此时记较小的一个为 a_0。于是有

$$x=\frac{2a_0+1}{2b_0}$$

如果它们不互素，记 $d=\gcd(2a_0+1,2b_0)\geq3$，则有

$$b_1=\frac{2b_0}{d}<b_0$$

$$a_1=\frac{2a_0+1}{d}$$

使得 $|b_1x-a_1|=|b_0x-a_0|=\dfrac{1}{2}$。这与上述选取方式中 b_0 的最小性矛盾。

此时将有理数 x 做连分数展开，则最后一个渐进分数的分子分母就是

$$p_N=2a_0+1$$

$$q_N=2b_0=a_Nq_{N-1}+q_{N-2}$$

由于 $a_N\geq2$，$q_{N-2}\geq1$，得到 $q_{N-1}<b_0$。于是

$$|q_{N-1}x-p_{N-1}|=\frac{1}{q_N}=\frac{1}{2b_0}\leq\frac{1}{2}$$

此时有更小的 q_{N-1} 可以取到最小值，仍旧与上述 b_0 的选取方式矛盾。证毕。

勒让德判别法

对实数 x 与分数 $\frac{p}{q}$，如果有

$$\left|x-\frac{p}{q}\right|<\frac{1}{2q^2}$$

则 $\frac{p}{q}$ 一定是 x 的渐近分数。

勒让德判别法的原始形式更加复杂，由勒让德（Legendre）于 1797 年发表在他的专著中。

勒让德判别法的原始表述是一个等价关系，这里给出的形式相对简化与宽松，是一个渐进分数的充分条件而非必要条件。

" 证明"

假设 $\frac{p}{q}$ 不是渐进分数，则存在 $0<b\le q$，且 $\frac{a}{b}\ne\frac{p}{q}$ 使得

$$|bx-a|\le|qx-p|<\frac{1}{2q}$$

于是有

$$\left|x-\frac{a}{b}\right|<\frac{1}{2qb}$$

此时首先两个分数之间的距离有

$$\left|\frac{p}{q}-\frac{a}{b}\right|=\frac{|pb-aq|}{qb}\ge\frac{1}{qb}$$

又有绝对值不等式

$$\left|\frac{p}{q}-\frac{a}{b}\right|=\left|\frac{p}{q}-x+x-\frac{a}{b}\right|\le\left|\frac{p}{q}-x\right|+\left|x-\frac{a}{b}\right|<\frac{b+q}{2bq^2}$$

连起来有

$$\frac{1}{qb}\le\left|\frac{p}{q}-\frac{a}{b}\right|<\frac{b+q}{2bq^2}$$

整理有 $b>q$。这与假设矛盾。证毕。

本页面主要译自博文 - . [2] 与其英文翻译版 The Stern-Brocot Tree and Farey Sequences[3]。其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0。

参考资料与注释

[1] GCJ 2019, Round 2 - New Elements: Part 2

[2] - .

[3] The Stern-Brocot Tree and Farey Sequences

9.12.30 二次域

二次有理数

定义

二次有理数是可以表示为整系数一元二次方程的解的数。

"用词说明"

在初等数论书上，「二次有理数」写为「二次无理数」。这是因为，**二次有理数不是有理数，而是无理数**。在近世代数书上，写为「二次有理数」或者「二次代数」，表明它与有理数拥有相似的性质。

同样地，还有「二次整数」。二次整数不是整数。「二次有理数」一词与「二次整数」相对应，与有理数和整数的关系完全一致，有理数是整数的比值。

所有二次有理数均可以表示成以下的形式：

$$a + b\sqrt{d}$$

其中，a 与 b 为有理数，d 为整数。任意这种形式的数都是二次有理数，两者为一一对应。

若 d 为正，则集合中所有数均为实数，称为实二次整环或者实二次域。若 d 为负，则集合中除了一般的有理数以外全部不是实数，称为虚二次整环或虚二次域。

范数

同一个整系数二次方程有两个根。如果它们不是一般的有理数，那么它们在形式上只在二次根号前相差一个正负号。

如果两个二次有理数只在二次根号之前相差正负号，称它们互为**共轭**关系。因为一般的有理数在二次根号前面的系数是 0，因此一般的有理数与它自身为共轭关系。

显然，在虚二次域中，某数的共轭的概念，与复数共轭的概念一致。但是在实二次域中这两个概念不一致。

在二次域中，由加减乘除（非 0）四则运算产生的等式，无法区分共轭关系。也就是说，在等式中将每一个数换成它的共轭数，即将每一个二次根号的符号改变，等式仍然成立。

二次有理数与它的共轭的和称为**迹**。某数的迹就是它的有理数部分的 2 倍，形式简单，因此很少研究迹。

二次有理数与它的共轭的积称为**范数**：

$$N(a + b\sqrt{d}) = a^2 - db^2$$

显然，在虚二次域中，范数的概念，与复数的模的平方的概念一致。但是在实二次域中这两个概念不一致。由于 d 不含平方因子，不可能是平方数，因此只有 0 的范数是 0。

范数具有保持乘法和除法（非 0）的良好性质。

$$N(a_1 + b_1\sqrt{d})N(a_2 + b_2\sqrt{d}) = N((a_1 + b_1\sqrt{d})(a_2 + b_2\sqrt{d}))$$

$$\frac{N(a_1 + b_1\sqrt{d})}{N(a_2 + b_2\sqrt{d})} = N\left(\frac{a_1 + b_1\sqrt{d}}{a_2 + b_2\sqrt{d}}\right)$$

一个二次有理数与它的共轭相乘为这个数的范数，因此它的倒数就是它的共轭与范数之比。

$$a + b\sqrt{d} = \frac{a - b\sqrt{d}}{N(a + b\sqrt{d})}$$

二次整数

首项系数为 1 的整系数二次多项式 $x^2 + px + q = 0$ 的零点是：

$$\frac{-p \pm \sqrt{p^2 - 4q}}{2}$$

称为「含有根号 d 的二次整数」，全体记作二次整环 $Z(\sqrt{d})$，对于加减乘封闭。不同的 d 对应于不同的整环。普通的整数环是每一个二次整环的子环。

第一种情况：对于所有的 d，$a + b\sqrt{d}$ 一定是二次整数。

第二种情况：当 d 模 4 余 1，a 与 b 是奇数的时候，$\frac{a+b\sqrt{d}}{2}$ 也是二次整数。因为这种情况也是首系数为 1 的整系数多项式的零点：

$$x^2 - ax + \frac{a^2 - db^2}{4} = 0$$

奇数的一半称半整数。两个半整数配上除以 4 余 1 的 d 开二次根号，也是二次整数。

以上的 d 全部可正可负。当 d 为正时就是普通意义的二次根号，当 d 为负的时候可以理解成对绝对值开根号，并乘以虚数单位 i。

二次整数有两个线性无关的分量，因此二次整数是二维的。

同类二次整数的比是二次有理数。

单位数

如果一个二次整数的倒数还是二次整数，称这个二次整数为**单位数**。二次整数是单位数的充要条件是它的范数为 1 或 -1。

单位数对于乘法封闭，构成单位群。有一个核心位置的定理（证明极难）：

狄利克雷单位定理：数域的单位群是有限生成阿贝尔群。

狄利克雷单位定理表明：单位群维数有限，存在一组基。所有的单位数可以由基的乘积表示。这组基（不含 1 和 -1）称为**基本单位数**。

三种整环

有三种整环的概念：

Euclid 整环：满足**辗转相除法**的整环。

主理想整环：每一个理想都是主理想的整环。一个重要的性质是，它满足**Bezout 定理**。

唯一分解整环：每个元素的非相伴分解都唯一的整环，满足**唯一分解定理**。

三个概念是层层嵌套包含的关系，唯一分解整环在最外面，欧几里得整环在最里面。欧几里得整环一定是主理想整环，主理想整环一定是唯一分解整环，而反之则不然。因此三个定理也有层层递推的关系。

虽然唯一分解整环不一定是主理想整环，例如在取模多项式整环中可以找到反例，但是在二次域中，这两个概念是重合的，即二次域的主理想整环与唯一分解整环范畴重合。因此，二次域只分为辗转相除和唯一分解两种特殊情形。

在虚二次域中，只有 -1、-2、-3、-7 和 -11 对应的虚二次整环是 Euclid 整环，其余均不满足辗转相除法。

在实二次域中，只有 2、3、5、6、7、11、13、17、19、21、29、33、37、41、57 和 73，共 16 个整环是 Euclid 整环。

对于二次域，有很重要的概念叫类数。理想的全体除以理想构成的群，得到商群的大小就称为类数。类数为 1，说明相应的整环是主理想整环。

Gauss 猜想有无穷个类数为 1 的实二次域，这个问题至今没有得到解决——关于实二次域的大多数此类研究进展都很慢。

已经得到解决的是，虚二次域中，加上上面的 5 个，只有 -19、-43、-67 和 -163 也是主理想整环。

好在之前的嵌套关系成立。我们只需知道高斯整环（-1）和艾森斯坦整环（-3）都是 Euclid 整环，满足辗转相除法和唯一分解定理就够了。

参见 OEIS：

Squarefree values of n for which the quadratic field Q(sqrt(n)) is norm-Euclidean[1]

Q(sqrt(n)) is a unique factorization domain (or simple quadratic field)[2]

相伴与唯一分解

如果一个二次整数乘一个单位数得到另一个二次整数，那么这两个二次整数是**相伴**关系。

唯一分解定理一定要考虑相伴关系才有可能成立。例如，若不考虑相伴关系，由于 -1 是单位数，整数不满足唯一分解：

$$10 = 2 * 5 = (-2) * (-5)$$

我们必须在相伴这个等价关系构成的诸多等价类中，为每个类指定一个数作为这个类的代表，即定义**本原数**，才可能有唯一分解。

例如在上面的例子中，如果指定 2 和 5 为本原数，那么 -2 和 -5 就不是本原数，此时 10 的分解才变得唯一了。

本原数的规定是人为的，即如果定义 -2 和 5、2 和 -5 或者 -2 和 -5 为本原数，在唯一分解的角度不会引起矛盾。一般会根据实际问题的研究方便定义本原数。例如，如果我们习惯于在正整数范畴研究问题，那么将正整数定义为本原数即可。

我们看到，事实上只需为所有的素数（在唯一分解前提下与不可约数等价）定义本原数就够了，其他的非素数的本原数定义必然由素数的本原数定义合成。

狄利克雷特征

讲述虚二次域的相关内容，需要先讲讲有关特征的概念。

定义：对于正整数 k，$\chi(n)$ 是定义在全体整数集合上不恒为 0 的数论函数。如果满足条件：

不互素时取值为 0：$\chi(n)=0$，当 $\gcd(n,k)>1$。

周期为 k：$\chi(n+k)=\chi(n)$。

完全积性：对于任意整数 m 和 n，有 $\chi(mn)=\chi(m)\chi(n)$。

那么，$\chi(n)$ 称为模 k 的狄利克雷特征，简称模 k 的特征，可以记作 $\chi(n,k)$。

根据上述定义，可以直接推出：

在 1 处取值为 1：$\chi(1)=1$。

在 -1 处取值为 ±1：$\chi(-1)=\pm1$。

互素时取值：当 $\gcd(n,k)=1$ 时，$(\chi(n))^{\varphi(k)}=\chi(n^{\varphi(k)})=\chi(1)=1$。

即，当自变量 n 与模数 k 互素时，模 k 的特征只能取 1 的 $\varphi(k)$ 次单位根，值域有限，因此模 k 的特征的个数也有限。

显然，当 $\gcd(n,k)=1$ 时，$\chi(n)$ 恒取值为 1 的数论函数一定是模 k 的特征，称为模 k 的主特征，记作 $\chi^0(n,k)$。模 1 和模 2 只有主特征。

一个特征，如果只取实数值（即取值为 ±1），称为实特征。模 3 和模 4 的特征都是实特征。

能取到非实数值得特征称为复特征。两个模 k 的特征，如果取值在复数域上共轭，称为共轭特征。实特征的共轭特征为本身。共轭特征的乘积为主特征。模 k 的全体特征的共轭仍旧为模 k 的全体特征。

关于特征，有如下一些定理：

定理：设 $\gcd(k_1,k_2)=1$，那么一定存在唯一的模 k_1 的特征 $\chi(n,k_1)$，使得当 $n\equiv 1 \pmod k)_2$ 时，

$$\chi(n,k_1k_2)=\chi(n,k_1)$$

定理：设 $\gcd(k_1,k_2)=1$，那么一定存在唯一的模 k_1 的特征 $\chi(n,k_1)$ 以及模 k_2 的特征 $\chi(n,k_2)$，使得对于任意整数 n，有：

$$\chi(n,k_1k_2)=\chi(n,k_1)\chi(n,k_2)$$

根据这个定理，特征可以随着模数的分解而分解，因此只需研究模为素数幂的特征即可。

对于奇素数的幂 p^a，存在原根 g，特征完全由它在原根 g 上的取值唯一确定。特征在原根 g 上可能的取值有：

$$\chi(g,p^a)=\mathrm{e}^{\frac{2\pi il}{\varphi(p^a)}}$$

因此在模 p^a 情形下至多有 $\varphi(p^a)$ 个不同的特征。根据原根对数的性质，上述特征在 l 不同时不同。因此，模 p^a 的特征恰好有 $\phi(p^a)$ 个。

标记顺序以作为区分：当 $\gcd(n,p)=1$ 时，取定模 p^a 的原根 g，则有

$$\chi(g,p^a,l)=\mathrm{e}^{\frac{2\pi il}{\varphi(p^a)}}$$

该式唯一确定一个模 p^a 的特征。并且，当且仅当 $l=0$ 时为主特征，当且仅当 $l=0$ 或 $l=\frac{\varphi(p^a)}{2}$ 时为实特征。

同样，根据模 2^a 的性质可以证明，模 2^a 的特征恰好有 $\varphi(2^a)$ 个。综上就有模 k 的特征恰好有 $\varphi(k)$ 个。

可以证明，模 k 的特征的乘法群，与模 k 的缩剩余系的乘法群同构。

定理：设 k 是不为 1 的正整数，$\gcd(a,k)=1$，a 与 1 模 k 不同余，那么一定存在模 k 的一个非主特征 χ，使得 $\chi(a)$ 不为 1。

类似于本原单位根，也有原特征的概念。模 k 的原特征的取值的最小正周期为 k，否则为非原特征。非原特征的最小正周期整除 k。

二次剩余符号 $\left(\frac{p}{q}\right)$ 是模 q 的实特征。

模 3 的非主特征只有 $\left(\frac{n}{3}\right)$，模 4 的非主特征只有 $\left(\frac{-1}{n}\right)=\left(\frac{-4}{n}\right)$。

二次域

具有同样 $\sqrt{d}$ 的二次有理数的全体，构成一个集合，记作 $Q(\sqrt{d})$。

容易证明，集合 $Q(\sqrt{d})$ 对于加、减、乘、除封闭，即任意取出两个元素，都可以进行四种运算（保证除数非 0），并且结果也在集合中。因此，它是一个域，称为 $\sqrt{d}$ 的二次域。

1 和 $\sqrt{d}$ 在有理数域上线性无关，所以在同一个二次域中，二次有理数的表示法具有唯一性。如果有：

$$a+b\sqrt{d}=e+f\sqrt{d}$$

那么有：

$$a=e \quad b=f$$

共轭定理：在同一个二次域中，如果一个等式仅经过有限次合法四则运算构成，那么对等式两边所有数同时取共轭，等式仍然成立。

证明：取共轭后的新的等式左右两边，结果一定仍然在该二次域中。只需证明它们对应的有理系数和无理系数相等。

无论如何，系数都与根号 d 的整体无关，取共轭只是将根号 d 换成了负根号 d，从头到尾只用到「平方等于 d」一个性质，因此，对应系数相等。

" 拓展"

二次域 $Q(\sqrt{d})$ 中的四则运算与一类特殊形式的二阶方阵同构：

$$\begin{pmatrix} a & b \\ db & a \end{pmatrix}$$

比如，乘法的行为模式完全一致：

$$\begin{pmatrix} a_1 & b_1 \\ db_1 & a_1 \end{pmatrix}\begin{pmatrix} a_2 & b_2 \\ db_2 & a_2 \end{pmatrix}=\begin{pmatrix} a_1a_2+db_1b_2 & a_1b_2+b_1a_2 \\ d(a_1b_2+b_1a_2) & a_1a_2+db_1b_2 \end{pmatrix}$$

因此二次有理数的一些性质可以由二阶方阵来解释。比如，范数恰好就是它的行列式：

$$N(a+b\sqrt{d})=\begin{vmatrix} a & b \\ db & a \end{vmatrix}$$

求倒数也就与伴随方阵求逆法一致。伴随方阵恰好就是它的共轭：

$$\begin{pmatrix} a & b \\ db & a \end{pmatrix}^*=\begin{pmatrix} a & -b \\ -db & a \end{pmatrix}$$

这种二阶方阵的记法参考了二维坐标系的旋转矩阵：

$$\begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$

二维坐标系的旋转矩阵的行为模式就像 d 为 -1 的特殊数域一样。

关于实二次域的相关研究，可以参见连分数和佩尔方程的部分。

虚二次域

在虚二次域中，仅当 d 为 -1 和 -3 的时候，存在除了 1 和 -1 以外的单位数。当 d 为负数且不为 -1 或 -3 的时候，单位数只有 1 和 -1。

当 d 为 -1 的时候，单位数有 4 个：1，-1，i，$-$i。当 d 为 -3 的时候，单位数有 6 个：1，-1，$\frac{1+\sqrt{3}\mathrm{i}}{2}$，$\frac{1-\sqrt{3}\mathrm{i}}{2}$，$\frac{-1+\sqrt{3}\mathrm{i}}{2}$，$\frac{-1-\sqrt{3}\mathrm{i}}{2}$。

在虚二次域中，仅当 d 为 -1 和 -3 时，存在基本单位数 i 和 $\frac{1+\sqrt{3}\mathrm{i}}{2}$。其他情况不存在 1 和 -1 以外的其他单位数，也就不存在基本单位数。

因此，两个整环 $Z(\mathrm{i})$ 和 $Z(\sqrt{3}\mathrm{i})$ 是特殊的整环，称为高斯整环和艾森斯坦整环。它们直观上分别构成复平面上正方形点阵和正六边形点阵（正三角形格点），研究虚二次域的时候最经常用到这两个整环。

虚二次域中对范数的研究可以转化为**椭圆上整点问题**，有名的「圆上整点问题」可以转化为对 d 为 -1 的虚二次域的研究。

Gauss 整数

一般将 $Q(\mathrm{i})$ 称为高斯域，相应的 $Z(\mathrm{i})$ 为高斯整环，高斯整环中的每个元素为高斯整数，即复平面上正方形格点。

高斯域恰好是四次分圆域，因此常用来解决**四次互反律**问题。

高斯整数中，一个数有四个相伴数（含本身）。

高斯整数中的全体素数分为三类：

分歧数：$1+\mathrm{i}$，为原来的 2 的因子。分歧数的共轭是它的相伴数，因此可以指定任一分歧数为本原数代表。

惯性数：所有正整数中 $4k+3$ 形式的素数，在高斯整数中仍旧为素数。在整环扩张中保持了素数的特性，因此称为「惯性」。

分裂数：所有正整数中 $4k+1$ 形式的素数，在高斯整数中可以拆成一对共轭的两个素数，这两个素数不相伴。这样的新素数是分裂的。

当然，这两个共轭的素数是不同的，即共轭的两个分裂数是互素的。

对于素数中的分裂数和惯性数，本原数的指定往往有着严格的规定，这是为了解决四次剩余问题的方便。

规定：高斯整数中的本原素数 π 有：

$\pi \equiv 1 \mod 2(1+\mathrm{i})$

在 $2(1+\mathrm{i})$ 的缩系中有 4 个剩余类，除了 $1+\mathrm{i}$ 的每个素数的每个相伴数恰好落入其中一类。

对于 $1+\mathrm{i}$ 与它的相伴数，一般指定 $1+\mathrm{i}$ 是本原素数。

勾股方程

高斯整数最简单的应用是解决勾股方程的解。勾股方程是满足下面形式的方程：

$$x^2+y^2=z^2$$

左边恰好构成高斯整数的范数，即：

$$N(x+y\mathrm{i})=z^2$$

通过模 4 的分析，我们知道右边模 4 必然余 1，即如果含模 4 余 3 的惯性数因子，必然含偶数个。

由于分歧数和分裂数的范数都是一般整数中的素数，将左边唯一分解后必然也只能成对出现（在共轭与相伴的意义下）。即：

$$(u+v\mathrm{i})^2=x+y\mathrm{i}$$

$$N(u+v\mathrm{i})=z$$

用一般的整数写出来就是：

$$u^2 - v^2 = x$$

$$2uv = y$$

$$u^2 + v^2 = z$$

勾股方程的几何意义是单位圆上的圆周角定理，或者半正切的外能代换公式。如下图：

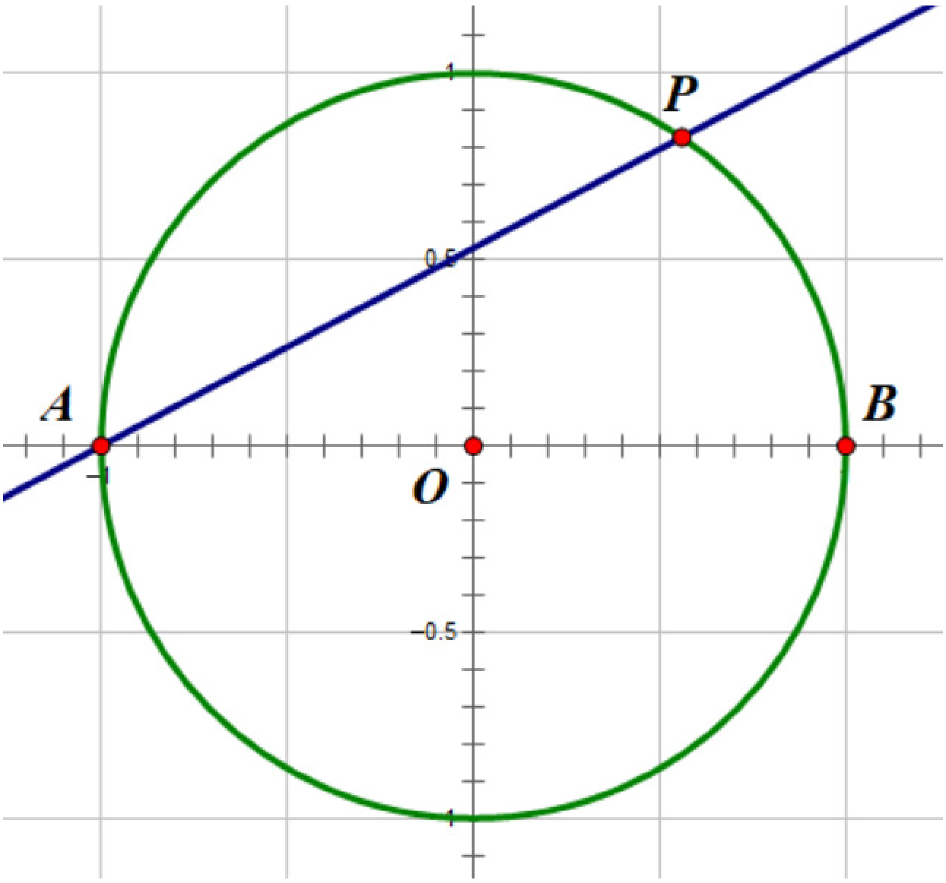


图 9.20 pic

单位圆周上的点 P 是有理点，等价于直线 AP 的斜率是有理数。

还证明相应的四次形式无解。即：

$$x^4 + y^4 = z^4$$

事实上，可以用无穷递降法证明，

$$x^4 + y^4 = z^2$$

没有整数解。

圆上整点问题

利用高斯整数的唯一分解，可以解决圆上整点问题。即给定范数为 n 的条件下，有多少个高斯整数满足这个范数 n：

$$N(x + y\mathrm{i}) = n$$

仍旧将左边和右边唯一分解。左边在高斯整数意义下唯一分解，右边在正整数范畴唯一分解。

$$N((1+\mathrm{i})^a)N(u_1 + v_1\mathrm{i})N(u_2 + v_2\mathrm{i}) = 2^a pq$$

对于分歧和分裂的素数，范数是原整数中的素数，而 $4k+3$ 形式惯性的素数，范数是原素数的平方。因此 n 中 $4k+3$ 形式的素数必须成对出现，否则无解。

然后利用简单的计数法就知道，在 n 中 $4k+3$ 形式的素数成对出现前提下，整点个数与含多少个 2（或 $1+\mathrm{i}$）无关，只与 $4k+1$ 形式的素数个数有关，每一个 $4k+1$ 形式的素数提供 2 中选择方法，在计数中扩大 2 倍。最后由对称性，整点个数乘 4 即可。

有解数的公式：

$$f(n)=4\sum_{d|n}\chi(n,4,1)$$

式中 χ 为上文提到的狄利克雷特征，$\chi(n,4,1)=\left(\frac{-1}{n}\right)=\left(\frac{-4}{n}\right)$。

Eisenstein 整数

注：Eisenstein（艾森斯坦）是 Gauss 的得意门生。

一般将 $Q(\sqrt{3}\mathrm{i})$ 称为艾森斯坦域，相应的 $Z(\sqrt{3}\mathrm{i})$ 为艾森斯坦整环，艾森斯坦整环中的每个元素为艾森斯坦整数，即复平面上正六边形格点。

艾森斯坦域恰好是三次分圆域，也是六次分圆域，因此常用来解决三**次互反律**问题。结合已经解决的二次互反律，就能给出六次剩余的手动计算。同样，如果结合高斯域中的四次互反律，就能解决十二次剩余的手动计算。

艾森斯坦整数中，一个数有六个相伴数（含本身）。

同样，艾森斯坦整数中的全体素数分为三类：

分歧数：$\frac{3+\sqrt{3}\mathrm{i}}{2}$，为原来的 3 的因子。

惯性数：所有正整数中 $3k+2$ 形式（2 和 $6k+5$ 形式）的素数，在艾森斯坦整数中仍旧为素数。

分裂数：所有正整数中 $3k+1$ 形式（$6k+1$ 形式）的素数，在高斯整数中可以拆成一对共轭的两个素数，这两个素数不相伴。这样的新素数是分裂的。同样，这两个共轭的素数是不同的，即共轭的两个分裂数是互素的。

对于素数中的分裂数和惯性数，本原数的指定也有着严格的规定，这是为了解决三次剩余问题的方便。

规定：艾森斯坦整数中的本原素数 π 有：

$\pi\equiv 1\mod 3$

在 3 的缩系中有 6 个剩余类，除了 $\frac{3+\sqrt{3}\mathrm{i}}{2}$ 的每个素数的每个相伴数恰好落入其中一类。注意，这与通常的 3 的剩余类不同。艾森斯坦整数中 3 的全部剩余类有 9 个，而缩系中有 6 个。

对于 $\frac{3+\sqrt{3}\mathrm{i}}{2}$ 与它的相伴数，可以指定 $\frac{3+\sqrt{3}\mathrm{i}}{2}$ 是本原素数。

艾森斯坦整数可以解决下面形式的方程的解：

$x^2+3y^2=z^2$

或者：

$x^2-xy+y^2=z^2$

或者：

$x^2+xy+y^2=z^2$

后两个在整数范畴是等价的。这里的求解完全仿照勾股方程即可，不再赘述。

类勾股方程

定理：设 z 为奇数，则当 $\gcd(x,y)=1$ 时

$$x^2+3y^2=z^3$$

成立，等价于存在 u 和 v，$\gcd(u,3v)=1$，使得

$$u^2+3v^2=z$$

$$x=u^3-9uv^2$$

$$y = 3u^2v - 3v^3$$

利用艾森斯坦整环的唯一分解性，该定理是显然的。

利用上述结论与无穷递降法，同样能证明三次的某种形式无解，即：

$x^3 + y^3 = z^3$

椭圆上整点问题

利用艾森斯坦整数的唯一分解，可以解决一种椭圆上整点问题。即给定范数为 n 的条件下，有多少个艾森斯坦整数满足这个范数 n。三种形式为：

$$x^2 + 3y^2 = n$$

或者：

$$x^2 - xy + y^2 = n$$

或者：

$$x^2 + xy + y^2 = n$$

方法仍旧完全一样，不再赘述。它们的结论是：

方程

$$x^2 - xy + y^2 = n$$

解的个数为

$$f(n) = 6\sum_{d|n}\chi(n,3,1)$$

式中 χ 为上文提到的狄利克雷特征，$\chi(n,3,1) = \left(\frac{n}{3}\right)$。

记 $n = 2^l m$，m 为正奇数。对于方程

$$x^2 + 3y^2 = n$$

的结论，当 l 为奇数时无解，当 $l = 0$ 时，解数为

$$12\sum_{d|n}\chi(n,3,1)$$

当 l 为正偶数时，解数为

$$36\sum_{d|n}\chi(n,3,1)$$

参考资料与注释

[1] Squarefree values of n for which the quadratic field Q(sqrt(n)) is norm-Euclidean

[2] Q(sqrt(n)) is a unique factorization domain (or simple quadratic field)

9.12.31 循环连分数

线性分式变换

连分数的另一个重要概念是所谓的线性分式变换（Linear fractional transformations）。

定义

线性分式变换是一个函数 $f:\mathbb{R}\to\mathbb{R}$，使得 $f(x)=\frac{ax+b}{cx+d}$ 对于一些 $a,b,c,d\in\mathbb{R}$。

线性分式变换 $L_0(x)=\frac{a_0x+b_0}{c_0x+d_0}$ 和 $L_1(x)=\frac{a_1x+b_1}{c_1x+d_1}$ 的组合 $(L_0\circ L_1)(x)=L_0(L_1(x))$ 也是线性分式变换：

$$\frac{a_0\frac{a_1x+b_1}{c_1x+d_1}+b_0}{c_0\frac{a_1x+b_1}{c_1x+d_1}+d_0}=\frac{a_0(a_1x+b_1)+b_0(c_1x+d_1)}{c_0(a_1x+b_1)+d_0(c_1x+d_1)}=\frac{(a_0a_1+b_0c_1)x+(a_0b_1+b_0d_1)}{(c_0a_1+d_0c_1)x+(c_0b_1+d_0d_1)}$$

线性分式变换的逆，也是线性分式变换：

$$y=\frac{ax+b}{cx+d}\iff y(cx+d)=ax+b\iff x=-\frac{dy-b}{cy-a}$$

DMOPC '19 Contest 7 P4 - Bob and Continued Fractions[1]

给您一个正整数数组 $a_1,\dots,a_n$。您需要回答 m 查询。每个查询都要计算 $[a_l;a_{l+1},\dots,a_r]$。

" 解答"

如果能够连接连分数，则可以用线段树来解决这个问题。

通常情况下，$[a_0;a_1,\dots,a_k,b_0,b_1,\dots,b_k]=[a_0;a_1,\dots,a_k,[b_1;b_2,\dots,b_k]]$ 是正确的。

表示 $L_k(x)=[a_k;x]=a_k+\frac{1}{x}=\frac{a_k\cdot x+1}{1\cdot x+0}$。注意 $L_k(\infty)=a_k$。在这个概念中，它认为

$$[a_0;a_1,\dots,a_k,x]=[a_0;[a_1;[\dots;[a_k;x]]]]=(L_0\circ L_1\circ\dots\circ L_k)(x)=\frac{p_kx+p_{k-1}}{q_kx+q_{k-1}}$$

因此，问题归结为计算

$$(L_l\circ L_{l+1}\circ\dots\circ L_r)(\infty)$$

变换的组合是关联的，因此可以在线段树的每个节点中计算其子树中变换的组合。

例题连分数的线性分式变换

设 $L(x)=\frac{ax+b}{cx+d}$。对于 $A=[a_0;a_1,\dots,a_n]$，计算 $L(A)$ 的连分数表示 $[b_0;b_1,\dots,b_m]$。

从而，对任意的 $\frac{p}{q}$，可以计算 $A+\frac{p}{q}=\frac{qA+p}{q}$ 和 $A\cdot\frac{p}{q}=\frac{pA}{q}$。

" 解答"

如上所述，$[a_0;a_1,\dots,a_k]=(L_{a_0}\circ L_{a_1}\circ\dots\circ L_{a_k})(\infty)$，因此 $L([a_0;a_1,\dots,a_k])=(L\circ L_{a_0}\circ L_{a_1}\circ\dots L_{a_k})(\infty)$。

因此，通过依次添加 L_{a_0},L_{a_1} 等，可以计算

$$(L\circ L_{a_0}\circ\dots\circ L_{a_k})(x)=L\left(\frac{p_kx+p_{k-1}}{q_kx+q_{k-1}}\right)=\frac{a_kx+b_k}{c_kx+d_k}$$

由于 $L(x)$ 是可逆的，因此在 x 中也是单调的。因此，对于任何 $x\geq 0$，都有 $L(\frac{p_kx+p_{k-1}}{q_kx+q_{k-1}})$ 介于 $L(\frac{p_k}{q_k})=\frac{a_k}{c_k}$ 和 $L(\frac{p_{k-1}}{q_{k-1}})=\frac{b_k}{d_k}$ 之间。

此外，对于 $x=[a_{k+1};\dots,a_n]$，它等于 $L(A)$。因此，$b_0=\lfloor L(A)\rfloor$ 介于 $\lfloor L(\frac{p_k}{q_k})\rfloor$ 和 $\lfloor L(\frac{p_{k-1}}{q_{k-1}})\rfloor$ 之间。当它们相等时，它们也等于 b_0。

请注意，$L(A)=(L_{b_0}\circ L_{b_1}\circ\dots\circ L_{b_m})(\infty)$。知道 b_0 后，可以用当前变换合成 $L_{b_0}^{-1}$，并继续添加 $L_{a_{k+1}}$、$L_{a_{k+2}}$ 等，寻找新的下界（floor）以达成一致，从中可以推断 b_1 等，直到恢复 $[b_0;b_1,\dots,b_m]$ 的所有值。

例题连分数算法

令 $A=[a_0;a_1,\dots,a_n]$，$B=[b_0;b_1,\dots,b_m]$。计算 $A+B$ 和 $A\cdot B$ 的连分数表示。

" 解答"

这里的想法与前面的问题类似，但不应使用 $L(x)=\frac{ax+b}{cx+d}$，而应考虑双线性分数变换 $L(x,y)=\frac{axy+bx+cy+d}{exy+fx+gy+h}$。您可以将当前变换更改为 $L(x,y)\mapsto L(L_{a_k}(x),y)$ 或 $L(x,y)\mapsto L(x,L_{b_k}(y))$，而不是 $L(x)\mapsto L(L_{a_k}(x))$。然后，检查 $\lfloor\frac{a}{e}\rfloor=\lfloor\frac{b}{f}\rfloor=\lfloor\frac{c}{g}\rfloor=\lfloor\frac{d}{h}\rfloor$，如果它们都一致，则在生成的分数中使用该值作为 c_k，并将转换更改为

$$L(x,y)\mapsto\frac{1}{L(x,y)-c_k}$$

循环连分数

仿照循环小数的概念，如果在连分数后面形成了循环，则形成**循环连分数**。

循环连分数的最小正周期一般用字母 l 来表示，循环的部分称为循环节。容易发现，进入循环的充要条件是余项 r_k 的重复出现。

纯循环连分数

定义

如果存在 k 使得 $x=[a_0;a_1,\dots,a_k,x]$，则连分数 $x=[a_0;a_1,\dots]$ 被称为**纯循环**（periodic）。

如果 $x=[a_0;a_1,\dots,a_k,y]$，其中 y 是纯循环，则连分数 $x=[a_0;a_1,\dots]$ 被称为**混循环**（eventually periodic）。

例如纯循环连分数：

$[a_0,a_1,a_2,a_3,a_0,a_1,a_2,a_3,...]=[\overline{a_0,a_1,a_2,a_3}]$

混循环连分数：

$[a_0,a_1,a_2,a_3,a_1,a_2,a_3,...]=[a_0,\overline{a_1,a_2,a_3}]$

混循环连分数后面循环的部分，最早循环的部分称为它的「纯循环部分」。

纯循环连分数的整数部分 a_0 直接进入到了循环里面，因此要求 a_0 必须至少是 1。因此，纯循环连分数大于 1。

对于 $x=[1;1,1,\dots]$，有 $x=1+\frac{1}{x}$，因此 $x^2=x+1$。在循环连分数和二次方程之间存在着一般的联系。考虑以下等式：

$$x=[a_0;a_1,\dots,a_k,x]$$

一方面，这个方程意味着 x 的连分数表示是周期为 $k+1$。

另一方面，使用渐进分数的公式，这个方程意味着

$$x=\frac{p_kx+p_{k-1}}{q_kx+q_{k-1}}$$

也就是说，x 是其自身的线性分数变换。从等式中可以看出，x 是二次方程式的根：

$$q_kx^2+(q_{k-1}-p_k)x-p_{k-1}=0$$

类似的推理代表了混循环连分数，即 $x=[a_0;a_1,\dots,a_k,y]$ 代表 $y=[b_0;b_1,\dots,b_k,y]$。事实上，从第一个方程导出 $x=L_0(y)$，从第二个方程导出 $y=L_1(y)$，其中 L_0 和 L_1 是线性分数变换。因此

$$x=(L_0\circ L_1)(y)=(L_0\circ L_1\circ L_0^{-1})(x)$$

可以进一步证明（由拉格朗日首先完成），对于具有整数系数的任意二次方程 $ax^2+bx+c=0$，其解 x 是一个混循环连分数。

拉格朗日连分数定理

关于循环连分数有一个特别重要的基础定理：

拉格朗日连分数定理：实二次有理数与循环连分数一一对应。

该定理最早由拉格朗日（Lagrange）于 1769 年证明。

根据余项的重复出现，证明循环连分数一定是二次有理数非常容易。重点在于证明二次有理数一定是循环连分数。

" 证明"

对二次有理数执行「无限简单连分数」计算，即取倒数、取整交替，得到的余数还是二次有理数。

接下来要强行刻画余项，将余项束缚在有限的范围，有限范围里的无限余项必然会发生重复。

设 ξ 是如下形式的二次有理数：

$\frac{1}{q}\left(c+\sqrt{d}\right) \quad q|N(c-\sqrt{d})$

如果分母不整除分子的范数，那么分子分母同时乘以分母的绝对值，并强行压入根号，在新二次有理数中，分母整除新的分子的范数。因此，任何二次有理数都能写成这形式。

根据上文的简单连分数算法：对余数取整可以得到 a，进而得到新的 c。

$a_k = \frac{c_{k+1}+c_k}{q_k}$

取整后得到的新的 c 为负数，且绝对值一定比 $\sqrt{d}$ 小，因此范数为负。取倒数，得到新的分母 q，q 总是正的。

$q_k q_{k+1} = -N(c_{k+1}-\sqrt{d})$

由于范数为负，取倒数之后 $\sqrt{d}$ 前面的符号不变，而 c 的符号由负变正（负数前面加负号变为正数）。

余数 r 里面，每个 c 都为负数，且绝对值一定比 $\sqrt{d}$ 小，因此分子 c 的个数有限。

每个 q 都整除对应 c 构成二次整数的范数，因此分母 q 的个数有限。余数有限必重复，至此证完。

例题二次有理数

找到 $\alpha = \frac{x+y\sqrt{n}}{z}$ 的连分数，其中 $x, y, z, n \in \mathbb{Z}$ 和 $n > 0$ 不是完全平方。

" 解答"

对于数字的第 k 个完全商 s_k，通常认为

$$\alpha = [a_0; a_1, \dots, a_{k-1}, s_k] = \frac{s_k p_{k-1} + p_{k-2}}{s_k q_{k-1} + q_{k-2}}$$

从而，

$$s_k = -\frac{\alpha q_{k-1} - p_{k-1}}{\alpha q_k - p_k} = -\frac{q_{k-1} y\sqrt{n} + (x q_{k-1} - z p_{k-1})}{q_k y\sqrt{n} + (x q_k - z p_k)}$$

将分子和分母乘以 $(x q_k - z p_k) - q_k y\sqrt{n}$，将去掉分母中的 $\sqrt{n}$，因此完全商为

$$s_k = \frac{x_k + y_k\sqrt{n}}{z_k}$$

接下来求解 s_{k+1}，假设 s_k 是已知的。

首先，$a_k = \lfloor s_k \rfloor = \left\lfloor \frac{x_k + y_k \lfloor\sqrt{n}\rfloor}{z_k} \right\rfloor$。然后

$$s_{k+1} = \frac{1}{s_k - a_k} = \frac{z_k}{(x_k - z_k a_k) + y_k\sqrt{n}} = \frac{z_k(x_k - y_k a_k) - y_k z_k\sqrt{n}}{(x_k - y_k a_k)^2 - y_k^2 n}$$

因此，如果表示 $t_k = x_k - y_k a_k$，将有

$$x * k + 1 = z_k t_k \tag{9.19}$$

$$y * k + 1 = -y * k z_k \tag{9.20}$$

$$z * k + 1 = t_k^2 - y_k^2 n \tag{9.21}$$

这种表示法的优点在于，如果将 $x_{k+1}, y_{k+1}, z_{k+1}$ 减去它们的最大公约数，结果将是唯一的。因此，可以使用它来检查当前状态是否已经重复，以及检查具有此状态的上一个索引的位置。

下面是计算 $\alpha = \sqrt{n}$ 的连分数表示的代码：

```
# compute the continued fraction of sqrt(n)
def sqrt(n):
    n0 = math.floor(math.sqrt(n))
    x, y, z = 0, 1, 1
    a = []
    def step(x, y, z):
        a.append((x * n0 + y) // z)
        t = y - a[-1]*z
        x, y, z = z*t, -z*y, t**2 - n*x**2
        g = math.gcd(x, math.gcd(y, z))
        return x // g, y // g, z // g

    used = dict()
    for i in range(n):
        used[x, y, z] = i
        x, y, z = step(x, y, z)
        if (x, y, z) in used:
            return a
```

使用相同的「step」函数，但不同的初始 x,y 和 z，可以计算任意 $\frac{x+y\sqrt{n}}{z}$。

伽罗瓦连分数定理

纯循环连分数有类似于反序定理的定理。记：

$$x = [\overline{a_0, a_1, \dots, a_{l-1}}]$$

$$x' = [\overline{a_{l-1}, a_{l-2}, \dots, a_0}]$$

则两个 x 互为「倒数负共轭」。即，若记：

$$y = -\frac{1}{x'}$$

则 x 与 y 共轭。

该定理最早由伽罗瓦（Galois）在他 17 岁那年（1828 年）发现并证明。

" 证明"

不妨改取比较长（例如至少有 5 项）的循环节。将循环节部分反序，根据反序定理，渐进分数有：

$$\frac{p_{l-1}}{p_{l-2}} = [a_{l-1}, a_{l-2}, \dots, a_0] = \frac{p'_{l-1}}{q'_{l-1}}$$

$$\frac{q_{l-1}}{q_{l-2}} = [a_{l-1}, a_{l-2}, \dots, a_1] = \frac{p'_{l-2}}{q'_{l-2}}$$

由于渐进分数的分子分母总是互素，只能分子分母对应相等。

根据纯循环，循环节的余项与它本身相等，有：

$$x = \frac{xp_{l-1} + p_{l-2}}{xq_{l-1} + q_{l-2}}$$

$$q_{l-1}x^2 + (q_{l-2} - p_{l-1})x - p_{l-2} = 0$$

之后只需将上述反序定理代入验证即证完。

根据伽罗瓦连分数定理，纯循环连分数的共轭一定落在 -1 到 0 之间。考虑它的逆问题，就有下面的定理：

如果二次有理数 x 大于 1，它的共轭落在 -1 到 0 之间，则它是纯循环连分数。

" 证明"

对二次无理数进行连分数算法，它的余项 r_{k+1} 容易得到。

根据共轭落在 -1 和 0 之间，利用归纳法可以知道，余项的共轭总是落在 -1 到 0 之间，因此部分商 a_k 是 r_{k+1} 的「倒数负共轭」的取整。这给了一种「倒推」的关系。

由拉格朗日连分数定理，x 一定是循环连分数，存在余项 r 相同，于是它们的前一个部分商 a 相同，前一个余项是小数部分的倒数，也相同。最后推得第 0 项在循环节中，该二次有理数纯循环。

根号 d 的连分数

对于非平方整数 d，有：

$$\sqrt{d} = [\lfloor\sqrt{d}\rfloor, \overline{a_1, a_2, \dots, a_{l-1}, 2\lfloor\sqrt{d}\rfloor}]$$

$$a_k = a_{l-k}$$

这是根号 d 的连分数形式。不仅最后一位是整数部分的两倍，而且中间部分还是对称的。

" 证明"

对于非平方整数 d，二次有理数

$$\lfloor\sqrt{d}\rfloor + \sqrt{d}$$

是纯循环连分数。于是就有：

$$\sqrt{d} = [\lfloor\sqrt{d}\rfloor, \overline{a_1, a_2, \dots, a_{l-1}, 2\lfloor\sqrt{d}\rfloor}]$$

而上述纯循环连分数的倒数负共轭既可以用伽罗瓦连分数定理表达，也可以由它本身直接表达：

$$\frac{1}{-\lfloor\sqrt{d}\rfloor + \sqrt{d}} = [\overline{a_{l-1}, a_{l-2}, \dots, a_1, 2\lfloor\sqrt{d}\rfloor}] = [\overline{a_1, a_2, \dots, a_{l-1}, 2\lfloor\sqrt{d}\rfloor}]$$

根据简单连分数展开的唯一性就证明了该结论。

同样也可以证明，整数部分为半整数的相同结构：

$$\frac{\sqrt{d}}{2} = [\lfloor\frac{\sqrt{d}-1}{2}\rfloor + \frac{1}{2}, \overline{a_1, a_2, \dots, a_{l-1}, 2\lfloor\frac{\sqrt{d}-1}{2}\rfloor + 1}]$$

$$a_k = a_{l-k}$$

一个实例

以 $\sqrt{74}$ 为例，实际运算一下连分数算法：

$$\begin{aligned}\sqrt{74} &= 8+(-8)+\sqrt{74} = \left[8, \frac{8+\sqrt{74}}{10}\right] \\ &= \left[8, 1+\frac{-2+\sqrt{74}}{10}\right] = \left[8, 1, \frac{2+\sqrt{74}}{7}\right] \\ &= \left[8, 1, 1+\frac{-5+\sqrt{74}}{7}\right] = \left[8, 1, 1, \frac{5+\sqrt{74}}{7}\right] \\ &= \left[8, 1, 1, 1+\frac{-2+\sqrt{74}}{7}\right] = \left[8, 1, 1, 1, \frac{2+\sqrt{74}}{10}\right] \\ &= \left[8, 1, 1, 1, 1+\frac{-8+\sqrt{74}}{10}\right] = \left[8, 1, 1, 1, 1, 8+\sqrt{74}\right] \\ &= \left[8, 1, 1, 1, 1, 16+(-8)+\sqrt{74}\right] = \left[8, \overline{1, 1, 1, 1, 16}\right]\end{aligned}$$

各个余项分别是：

$$\begin{aligned} r_1 &= \frac{8+\sqrt{74}}{10} = \left[\overline{1, 1, 1, 1, 16}\right] \\ r_2 &= \frac{2+\sqrt{74}}{7} = \left[\overline{1, 1, 1, 16, 1}\right] \\ r_3 &= \frac{5+\sqrt{74}}{7} = \left[\overline{1, 1, 16, 1, 1}\right] \\ r_4 &= \frac{2+\sqrt{74}}{10} = \left[\overline{1, 16, 1, 1, 1}\right] \\ r_5 &= 8+\sqrt{74} = \left[\overline{16, 1, 1, 1, 1}\right] \end{aligned}$$

根据伽罗瓦连分数定理，对称的余项 r_k 和 r_{l+1-k} 循环部分恰好相反，因此互为倒数负共轭。

如果循环节 l 是奇数，则对称部分最中间的余项与自己互为倒数负共轭。但如果循环节 l 是偶数，就不会出现这种现象。

在后面的 Pell 方程一节中将指出，在根号 d 的连分数中，循环节 l 的奇偶性，将直接决定 Pell 方程中的 -1 形式是否有解。

Tavrida NU Akai Contest - Continued Fraction[2]

你得到了 x 和 k,x 不是一个完全平方数。让 $\sqrt{x} = [a_0; a_1, \dots]$，找到 $\frac{p_k}{q_k} = [a_0; a_1, \dots, a_k]$ 的 $0 \le k \le 10^9$。

" 解答"

在计算完 $\sqrt{x}$ 的周期后，可以使用连分数表示引起的线性分数变换上的二进制幂来计算 a_k。要查找结果转换，请将大小为 T 的周期压缩为单个转换，并将其重复 $\lfloor \frac{k-1}{T} \rfloor$ 次，然后手动将其与其余转换组合。

```
x, k = map(int, input().split())

mod = 10**9+7

# compose (A[0]*x + A[1]) / (A[2]*x + A[3]) and (B[0]*x + B[1]) / (B[2]*x + B[3]
)
def combine(A, B):
    return [t % mod for t in [A[0]*B[0]+A[1]*B[2], A[0]*B[1]+A[1]*B[3], A[2]*B[0
]+A[3]*B[2], A[2]*B[1]+A[3]*B[3]]]

A = [1, 0, 0, 1] # (x + 0) / (0*x + 1) = x
```

```python
a = sqrt(x)

T = len(a) - 1 # period of a

# apply ak + 1/x = (ak*x+1)/(1x+0) to (Ax + B) / (Cx + D)
for i in reversed(range(1, len(a))):
    A = combine([a[i], 1, 1, 0], A)

def bpow(A, n):
    return [1, 0, 0, 1] if not n else combine(A, bpow(A, n-1)) if n % 2 else bpow(combine(A, A), n // 2)

C = (0, 1, 0, 0) # = 1 / 0
while k % T:
    i = k % T
    C = combine([a[i], 1, 1, 0], C)
    k -= 1

C = combine(bpow(A, k // T), C)
C = combine([a[0], 1, 1, 0], C)
print(str(C[1]) + '/' + str(C[3]))
```

参考资料与注释

[1] DMOPC '19 Contest 7 P4 - Bob and Continued Fractions

[2] Tavrida NU Akai Contest - Continued Fraction

9.12.32 Pell 方程

二次整数

对于二次有理数 $a+b\sqrt{D}$，此处要求 D 是不含平方因子的整数。当以下情形成立时：

- a 与 b 是整数，$D\equiv 2 \pmod 4$ 或 $D\equiv 3 \pmod 4$。
- a 与 b 是整数，或者 a 与 b 同时是半整数，$D\equiv 1 \pmod 4$。

此时称该二次有理数 $a+b\sqrt{D}$ 是二次整数。二次整数与首一整系数二次方程的解构成对应关系。

如果二次整数 $a+b\sqrt{D}$ 的范数 a^2-Db^2 为 1 或 -1，则它的倒数也是二次整数，恰好是它的共轭或者共轭的相反数。此时称它为整环 $Z(\sqrt{D})$ 的**单位数**，简称单位数。

可以证明，存在**基本单位数**，使得全体单位数都可以表示成为基本单位数的幂（或幂的相反数）。它也就是对应 Pell 方程的**基本解**，通解可以表示为基本解的幂（或幂的相反数）。

我们用 Dirichlet 逼近定理来逼近二次根式 $\sqrt{D}$。即有无穷个有理数（显然为正有理数）满足：

$$\left|\frac{x}{y}-\sqrt{D}\right| \leqslant \frac{1}{y^2}$$

于是，下面的范数就有：

$$\left|N(x+y\sqrt{D})\right| = |x-y\sqrt{D}||x+y\sqrt{D}| \leqslant \frac{1}{y}(\frac{1}{y}+2y\sqrt{D}) \leqslant 2\sqrt{D}+1$$

这是对范数拆出的两项进行估值。这也直观地说明只要有理数与 $\sqrt{D}$ 越接近，范数越小。

因此，范数较小的二次整数有无限个，进而采用一些手段，就可以推出范数为 ± 1 的单位数存在，也存在无限个。

进而可以发现，对于所有 $\sqrt{D}$ 的渐进分数，配上系数之后得到的二次整数的范数都落在非常小的区间。由于 $\sqrt{D}$ 的渐进分数是余数循环的，只要其中出现使得范数为 ± 1 的渐进分数，经过一个循环之后新的渐进分数凑成的二次整数也应当满足范数为 ± 1，即这个新渐进分数也是单位数。由于第 -1 个渐进分数规定为 $\frac{1}{0}$，对应的二次整数范数为 1，那么只要计算每个循环节处前一个渐进分数即可。

根据上逼近与下逼近的结论，第奇数个渐进分数得到的范数为负，偶数个为正。即是否存在范数为 -1 的二次整数取决于循环连分数的循环节长度是否为奇数。

最后还有一个结论，每经过一个循环，相当于旧的二次整数乘上了一个单位数，得到新的二次整数。因此上面得到的单位数是基本单位数。这样，就提供了一种 Pell 方程通解的直接计算方法。

连分数过渡到 Pell 方程的一些定理

定理：记 $x^2 - Dy^2 = s$。如果有 $|s| < \sqrt{D}$，则 $\frac{x}{y}$ 一定是 $\sqrt{D}$ 的渐进分数。

证明：分情况讨论。

当 $s > 0$ 时，根据 $x^2 - Dy^2 > 0$，有 $x > y\sqrt{D}$。并且有

$$\left|\frac{x}{y} - \sqrt{D}\right| = \frac{s}{y(x + y\sqrt{D})} < \frac{s}{2y^2\sqrt{D}} < \frac{1}{2y^2}$$

此时根据勒让德判别法，$\frac{x}{y}$ 是 $\sqrt{D}$ 的渐进分数。

当 $s < 0$ 时，原始方程 $x^2 - Dy^2 = s$ 可以变形为 $y^2 - \frac{1}{D}x^2 = -\frac{s}{D}$。变回上一种情况。于是 $\frac{y}{x}$ 是 $\frac{1}{\sqrt{D}}$ 的渐进分数。由倒数定理，$\frac{x}{y}$ 是 $\sqrt{D}$ 的渐进分数。

定理：

$${p_k}^2 - D{q_k}^2 = (-1)^{k+1}Q_{k+1}$$

式中的 Q_{k+1} 也是拉格朗日连分数定理中的 r_k 分母。

证明：根据

$$r_{k+1} = \frac{P_{k+1} + \sqrt{D}}{Q_{k+1}}$$

$$\sqrt{D} = \frac{r_{k+1}p_k + p_{k-1}}{r_{k+1}q_k + q_{k-1}}$$

消去 r 可以得到

$$p_kP_{k+1} + p_{k-1}Q_{k+1} + p_k\sqrt{D} = q_kD + (q_kP_{k+1} + q_{k-1}Q_{k+1})\sqrt{D}$$

根据有理项和无理项对应相等，有

$$p_k = q_kP_{k+1} + q_{k-1}Q_{k+1}$$

$$q_kD = p_kP_{k+1} + p_{k-1}Q_{k+1}$$

分别乘以 p_k、q_k，再相减，得到

$${p_k}^2 - D{q_k}^2 = (p_kq_{k-1} - p_{k-1}q_k)Q_{k+1} = (-1)^{k-1}Q_{k+1} = (-1)^{k+1}Q_{k+1}$$

证毕。

定理：当且仅当 $k = nL$，其中 n 是正整数，L 是最短循环周期时，有 $Q_k = 1$。

证明：

已经知道

$$\Delta + \sqrt{D}$$

是纯循环连分数。并且有

$$r_1(\sqrt{D}) = r_1(\Delta + \sqrt{D})$$

$$Q_0(\sqrt{D}) = Q_0(\Delta + \sqrt{D}) = 1$$

因此 $\sqrt{D}$ 和 $\Delta + \sqrt{D}$ 从第 1 项起余项相同，第 0 项起分母相同。后续的 Q_k 完全一致。

因为 $Q_0 = 1$，并且有纯循环的周期性，所以 $Q_{nL} = 1$。

纯循环连分数的余项也纯循环。当 $Q_k = 1$ 时，有

$$r_k = P_k + \sqrt{D} > 1$$

$$-1 < P_k - \sqrt{D} < 0$$

$$P_k < \sqrt{D} < P_k + 1$$

$$P_k = \lfloor \sqrt{D} \rfloor = \Delta$$

$$r_k = \Delta + \sqrt{D} = r_0$$

根据 L 为最短周期，有 $k = nL$。证毕。

定理：如果 (x_1, y_1) 和 (x_2, y_2) 都是 $x^2 - Dy^2 = 1$ 的一组整数解，那么

$$X = x_1x_2 + y_1y_2D$$

$$Y = x_1y_2 + x_2y_1$$

也是方程的一组整数解。这是因为

$$X + Y\sqrt{D} = (x_1 + y_1\sqrt{D})(x_2 + y_2\sqrt{D})$$

Pell 方程

我们给出两个不定方程：$x^2 - Dy^2 = 1$ 和 $x^2 - Dy^2 = -1$，若 D 为完全平方数，则第一个方程只有解 $(\pm 1, 0)$，第二个方程无解。

若 D 不为完全平方数，称形如此类的方程为 Pell 方程。根据相应的二次整环不同，一般研究的 Pell 方程分为 1、-1、4、-4 四类。

设 $\xi_0 = \sqrt{D}$，它的循环连分数周期为 l，渐近分数为 $\frac{p_n}{q_n}$，则：

- 当 l 为偶数时，第一个方程的全体正解为 $x = p_{jl-1}, y = q_{jl-1}, j = 1, 2, 3, \cdots$，第二个方程无解。
- 当 l 为奇数时，第一个方程的全体正解为 $x = p_{jl-1}, y = q_{jl-1}, j = 2, 4, 6, \cdots$，第二个方程的全体正解为 $x = p_{jl-1}, y = q_{jl-1}, j = 1, 3, 5, \cdots$。

还有另一种更加简单的表示方法：

- 当 l 为偶数时，第一个方程的全体解为 $x + y\sqrt{D} = \pm(p_{l-1} \pm q_{l-1}\sqrt{D})^j, j = 0, 1, 2, \cdots$，第二个方程无解。
- 当 l 为奇数时，第一个方程的全体正解为 $x + y\sqrt{D} = \pm(p_{l-1} \pm q_{l-1}\sqrt{D})^j, j = 0, 2, 4, \cdots$，第二个方程的全体正解为 $x + y\sqrt{D} = \pm(p_{l-1} \pm q_{l-1}\sqrt{D})^j, j = 1, 3, 5, \cdots$。

这是循环连分数渐进分数与二次有理数乘法的对应关系。该结论由下面的定理给出。

定理：记 Pell 方程 $x^2 - Dy^2 = 1$ 使得 $x + y\sqrt{D}$ 最小的一组正整数解为基本解 (x_1, y_1)，则方程的全部正整数解为

$$x_n + y_n\sqrt{D} = (x_1 + y_1\sqrt{D})^n$$

证明：

假如有一组正整数解 (X, Y) 不出现在上述序列中。因为 $x_1 + y_1\sqrt{D} > 1$，所以必然有

$$(x_1 + y_1\sqrt{D})^n < X + Y\sqrt{D} < (x_1 + y_1\sqrt{D})^{n+1}$$

两边同时乘 $x_n - y_n\sqrt{D}$，也就是除以 $(x_1 + y_1\sqrt{D})^n$，有

$$1 < S + T\sqrt{D} < x_1 + y_1\sqrt{D}$$

并且 (S, T) 也是一组整数解。有

$$0 < S - T\sqrt{D} = \frac{1}{S + T\sqrt{D}} < 1$$

$$2S = S + T\sqrt{D} + S - T\sqrt{D} > 0$$

$$2T\sqrt{D} = S + T\sqrt{D} - (S - T\sqrt{D}) > 0$$

因此 (S, T) 是一组正整数解。这与 (x_1, y_1) 的选取矛盾。证毕。

定理：对于具有奇数位循环节的 $\sqrt{D}$，记最小的一组满足 $x^2 - Dy^2 = -1$ 的正整数解为 (x_1, y_1)，则满足 $x^2 - Dy^2 = \pm 1$ 的所有解由

$$x_n + y_n\sqrt{D} = (x_1 + y_1\sqrt{D})^n$$

给出。并且 (x_{2n}, y_{2n}) 满足 $x^2 - Dy^2 = 1$，(x_{2n-1}, y_{2n-1}) 满足 $x^2 - Dy^2 = -1$。

证明完全同上。

方程 $x^2 - Dy^2 = 1$ 必然有解，而方程 $x^2 - Dy^2 = -1$ 不一定有解，有解等价于连分数循环节长度为奇数。

定理：如果 $\sqrt{D}$ 连分数循环节长度为奇数，并且 D 存在唯一的平方和表示 $D = r^2 + s^2$，则两个方程 $x^2 - Dy^2 = r$、$x^2 - Dy^2 = s$ 至少有一个有解。

如果了解高斯整数的知识，只有当一个数所有的 $4k + 3$ 型素因子均成对，这个数才能进行平方和表示，此时平方和表示的个数就是这个数含有的 $4k + 1$ 型素因子数的个数。

证明：

根据伽罗瓦连分数定理，对称的余项 r_k 和 r_{l+1-k} 循环部分恰好相反，因此互为倒数负共轭。

因为循环节是奇数，连分数展开中对称部分最中间的余项与自己互为倒数负共轭。记对称部分最中间位置下标为 v。于是有

$$\left(\frac{P_v - \sqrt{D}}{Q_v}\right) = -\frac{1}{\left(\frac{P_v + \sqrt{D}}{Q_v}\right)}$$

$$D = {P_v}^2 + {Q_v}^2$$

因为 D 的平方和表示是唯一的，所以下标 v 必然有 $P_v = r$、$Q_v = s$，或者 $P_v = s$、$Q_v = r$。

由于得到余项的前面的操作为取倒数，即负共轭，再前面为取整，下标 $v - 1$ 的余项 $\left\lfloor \frac{P_{v-1} + \sqrt{D}}{Q_{v-1}} \right\rfloor$ 分母应当与下标 v 这项相同，即 $Q_v = Q_{v-1}$。

由于连分数的结论有 $x^2 - Dy^2 = (-1)^v Q_v$ 的解为相应渐进分数，因此上述两个方程必然存在一个解，为下标 v 或 $v - 1$ 的渐进分数。证毕。

如果直接对方程 $x^2 - Dy^2 = -1$ 两端取模 8，能够知道 $D \equiv 1 or 2 \pmod 8$。然而，这只是一个充分条件，而非必要条件。通过取模的方式确定什么时候方程有解，基本做不到。例如，可以给出一个无解的充分条件：

定理：如果 D 存在唯一的平方和表示 $D = r^2 + s^2$，并且 $r, s \equiv \pm 3 \pmod 8$，则方程 $x^2 - Dy^2 = -1$ 无解。

例如对于 34 有 $34 = 3^2 + 5^2$，于是 $x^2 - 34y^2 = -1$ 无解。

证明：

如果方程 $x^2 - Dy^2 = -1$ 有解，则两个方程 $x^2 - Dy^2 = r$、$x^2 - Dy^2 = s$ 至少有一个有解。模 8 就得到了矛盾。证毕。

对于 D 为 $4k+1$ 形式的时候，有可能相应基本单位数的系数是半整数。此时有结论：如果 D 为 $4k+1$ 形式时，相应基本单位数的系数是半整数，则基本单位数的三次方系数为整数。

此时，上述方法求出的基本解不是基本单位数，而是基本单位数的三次方。

如果想直接求解 D 为 $4k+1$ 形式时的基本单位数，改令 $\xi_0 = \frac{\sqrt{D}}{2}$，并规定这里的连分数第零项为半整数，重复上述操作，并将结果乘 2（提出二分之一）。

例如当 D 为 5 的时候，$\frac{\sqrt{5}}{2}$ 的半整数连分数表示为：

$$\frac{\sqrt{5}}{2} = [\frac{1}{2}, \overline{1}]$$

$$\frac{1}{2} = \frac{1}{2}\frac{1}{1}$$

于是解得基本单位数 $\frac{1}{2} + \frac{\sqrt{5}}{2}$。

但是 D 为 17 的时候，$\frac{\sqrt{17}}{2}$ 的半整数连分数表示为：

$$\frac{\sqrt{17}}{2} = [\frac{3}{2}, \overline{1, 1, 3}]$$

$$\frac{3}{2} + \frac{1}{1+\frac{1}{1}} = \frac{3}{2} + \frac{1}{2} = \frac{1}{2}\frac{4}{1}$$

于是解得基本单位数 $4+\sqrt{17}$。它不属于半整数形式。

在 1 到 100 中，5，13，29，53，61 和 85 的基本单位数属于这种分母中含 2 的半整数形式，而 17，37，41，65，73，89 和 97 的基本单位数属于非半整数形式。

如果快速求解第 n 个解（或第 n 个单位数），只需要求出基本解（或基本单位数），然后借助快速幂的想法去乘就可以了。注意乘一个二次有理数的时候，a 与 b 的变化是一个递推关系。

如果要求从头开始连续若干个解（或连续若干个单位数），a 与 b 的变化就是一个固定的递推关系，相邻三项一定满足特征方程，即基本解（或基本单位数）对应的二次三项式。即：

如果基本解（或基本单位数）$a+b\sqrt{D}$ 是对应的二次方程 $x^2+px+q=0$ 的解，则有递推：

$$a_n + pa_{n-1} + qa_{n-2} = 0$$
$$b_n + pb_{n-1} + qb_{n-2} = 0$$

事实上，斐波那契数列（的一半）与卢卡斯数列（的一半）恰好组合成了基本单位数 $\frac{1}{2}+\frac{1}{2}\sqrt{5}$ 的全体幂，即使引入负下标也成立。这是它们的很多性质的来源。

9.13 多项式与生成函数

9.13.1 多项式与生成函数简介

多项式与生成函数

操纵有限项/无限项的多项式是 OI 数学中，尤其是生成函数中的重要内容。

以 快速傅里叶变换 为基石的多项式算法赋予了算法竞赛选手直接操纵生成函数的能力。

基本概念

对于求和式 $\sum a_n x^n$，如果是有限项相加，称为多项式，记作 $f(x) = \sum_{n=0}^{m} a_n x^n$。

可列项相加的求和式称为级数。在和式 $\sum_{n=0}^{\infty} a_n x^n$ 中，每项均为非负整数次幂函数乘常数系数，这种形式的级数称为幂级数。

研究多项式算术时先考虑较简单的多项式，幂级数概念仅用于方便理解。到了数学分析中会进一步研究幂级数的敛散性。

环、域及其衍生结构的一般定义详见 群论简介。

对于一般环 R，定义 R 上的**多项式环**（polynomial ring）$R[x]$。

每个元素 f 称为 R 上的**多项式**（polynomial），可表示为

$$f = \langle f_0, f_1, f_2, \cdots, f_n \rangle \quad (f_0, f_1, f_2, \cdots, f_n \in R)$$

换言之，我们将多项式直接定义为系数序列。也可以表示为

$$f(x) = f_0 + f_1 x + f_2 x^2 + \cdots + f_n x^n$$

此处我们认为 x 只是一个**形式符号**，一个对系数位置的标识符。

如果我们还允许无穷项的存在，即

$$f(x) = f_0 + f_1 x + f_2 x^2 + \cdots$$

则可得到**形式幂级数环**（formal power series ring）$R[[x]]$，其中的每个元素 f 称为**形式幂级数**（formal power series），以下简称幂级数。

多项式的度

对于一个多项式 $f(x)$，称其最高次项的次数为该多项式的**度（degree）**，也称次数，记作 $\deg f$。

多项式的乘法

最核心的操作是两个多项式的乘法，即给定多项式 $f(x)$ 和 $g(x)$：

$$f(x) = a_0 + a_1 x + \cdots + a_n x^n \qquad (1)$$

$$g(x) = b_0 + b_1 x + \cdots + b_m x^m \qquad (2)$$

要计算多项式 $Q(x) = f(x) \cdot g(x)$：

$$\boxed{Q(x) = \sum_{i=0}^{n} \sum_{j=0}^{m} a_i b_j x^{i+j}} = c_0 + c_1 x + \cdots + c_{n+m} x^{n+m}$$

多项式或幂级数的乘法，满足结合律，关于加法满足分配律。若 R 为交换环或幺环，乘法相应的有交换律和单位元。

若 R 上存在 2^n 次单位根，快速傅里叶变换 允许我们在 $O(n2^n)$ 而不是 $O(2^{2n})$ 的时间内计算两个 2^n 次多项式的乘积。

复合

定义 $R[[x]]$ 中元素 f 的乘方为

$$f^1 = f, f^k = f^{k-1} \times f$$

在此基础上，定义 $R[[x]]$ 中元素 f, g 的复合为

$$(f \circ g)(x) = f(g(x)) = f_0 + \sum_{k=1}^{+\infty} f_k g^k(x)$$

我们规定 $f \circ g$ 存在当且仅当 f 为有限项或 $g_0 = 0$，这样就不涉及 R 上的极限了。

$\circ$ 满足结合律（$(f \circ g) \circ h$ 和 $f \circ (g \circ h)$ 均存在时），不满足交换律。R 为幺环时 $\circ$ 存在单位元 $1 \times x$。

FFT 可行时，有限项多项式的复合有 $O((n \log n)^{1.5})$ 的算法，但此算法常数较大，实战中往往 $O(n^2)$ 的一种「分块 FFT」算法具有更好的表现。

导数

尽管一般环甚至未必存在极限，

我们依然可以定义形式幂级数的**形式导数**（formal derivative）为

$$\left(\sum_{k=0}^{+\infty} f_k x^k\right)' = \sum_{k=1}^{+\infty} k f_k x^{k-1}$$

其中

$$k f_k = \underbrace{f_k + f_k + \cdots + f_k}_{k\text{个}f_k}$$

基本求导法则——加法法则、乘法法则、链式法则（复合允许的情况下）依然是正确的。

如果 R 上允许作除法，同样可以类似定义形式幂级数的**形式不定积分**（formal indefinite integral）。

乘法逆元

根据例子

$$\frac{1}{1-x} = 1 + x + x^2 + \cdots$$

可以知道，多项式的倒数是能展开为无穷级数的。存在倒数，当且仅当常数项不为 0，并且倒数也满足常数项不为 0。

因此定义：对于形式幂级数 f，若 $f_0 \neq 0$，其**乘法逆元**（multiplicative inversion）f^{-1} 为另一形式幂级数，满足

$$f \times f^{-1} = f^{-1} \times f = 1$$

用形式幂级数乘法定义展开该式，可得 f^{-1} 系数的递推式

$$f_0^{-1} = \frac{1}{f_0}, f_n^{-1} = \frac{-1}{f_0} \sum_{k=0}^{n-1} f_k^{-1} f_{n-k}$$

直接用递推式计算前 n 项是 $O(n^2)$ 的，运用 FFT 可得到 $O(n \log n)$ 的算法。

" 注"

容易发现，$f(x)$ 的倒数就是 $\frac{1}{f(x)}$ 的无穷项麦克劳林展开（在 $x = 0$ 处的无穷项泰勒展开）。

常见的幂级数展开式

在数学分析中，在某点处或某区间上若干阶可导的一元函数可以在相应范围进行多项式展开，一般统称为泰勒展开，如果在 0 处展开，也称为麦克劳林展开。

如果无穷阶可导，则可以进行幂级数展开。最常见的还是在 0 处展开。

在复变函数中，一些函数在奇点上虽然不能进行泰勒展开，但是可以进行洛朗展开。

下列等式只在幂级数收敛时成立，在不收敛时不成立。这里只写出展开式，不讨论收敛域。

基本的展开式有以下两个，指数函数和幂函数：

$$\mathrm{e}^x = 1 + x + \frac{1}{2!}x^2 + \ldots + \frac{1}{n!}x^n + \ldots$$

$$(1+x)^a = 1 + ax + \frac{a(a-1)}{2!}x^2 + \ldots + \frac{a(a-1)\ldots(a-n+1)}{n!}x^n + \ldots$$

更多的展开式经常由上述两个变形得到。比如正余弦函数由指数函数代入复数得到：

$$\cos x = 1 - \frac{1}{2!}x^2 + \frac{1}{4!}x^4 + \ldots + \frac{(-1)^n}{(2n)!}x^{2n} + \ldots$$

$$\sin x = x - \frac{1}{3!}x^3 + \frac{1}{5!}x^5 + ... + \frac{(-1)^n}{(2n+1)!}x^{2n+1} + ...$$

对数和反正切、反正弦函数由积分得到：

$$\frac{1}{1+x} = 1 - x + x^2 + ... + (-1)^n x^n + ...$$

$$\ln(1+x) = x - \frac{1}{2}x^2 + \frac{1}{3}x^3 + ... + \frac{(-1)^{n-1}}{n}x^n + ...$$

$$\frac{1}{1+x^2} = 1 - x^2 + x^4 + ... + (-1)^n x^{2n} + ...$$

$$\arctan x = x - \frac{1}{3}x^3 + \frac{1}{5}x^5 + ... + \frac{(-1)^n}{2n+1}x^{2n+1} + ...$$

$$\frac{1}{\sqrt{1+x}} = 1 - \frac{1}{2}x + \frac{3}{8}x^2 + ... + (-1)^n \frac{(2n)!}{(n!)^2 4^n}x^n + ...$$

$$\frac{1}{\sqrt{1-x^2}} = 1 + \frac{1}{2}x^2 + \frac{3}{8}x^4 + ... + \frac{(2n)!}{(n!)^2 4^n}x^{2n} + ...$$

$$\arcsin x = x + \frac{1}{6}x^3 + \frac{3}{40}x^5 + ... + \frac{(2n)!}{(n!)^2 (2n+1) 4^n}x^{2n+1} + ...$$

复合逆

复合逆（compound inversion）即反函数概念在形式幂级数环上的推广。

对于满足 $f_0 = 0$ 且 $f_1 \neq 0$ 的形式幂级数 f，其复合逆为满足 $g(f(x)) = f(g(x)) = x$ 的形式幂级数 g。由拉格朗日反演可得对于任意整数 n, k 有

$$n[x^n]f^k = k[x^{n-k}]\left(\frac{x}{g}\right)^n$$

式中记号 $[x^k]f(x)$ 表示 $f(x)$ 在 x^k 处的系数。

多项式整除

对于多项式 $f(x)$ 和多项式 $g(x)$，如果存在一个多项式 $h(x)$，使得：

$$f(x) = g(x)h(x)$$

则多项式 $g(x)$ 整除多项式 $f(x)$。

显而易见，多项式 $g(x)$ 整除多项式 $f(x)$，当且仅当 $g(x)$ 的根全部为 $f(x)$ 的根，并且在 $g(x)$ 中的重数不超过在 $f(x)$ 中的相应重数。

多项式的余数和商

对于多项式 $f(x), g(x)$，存在**唯一**的 $Q(x), R(x)$ 满足：

$$f(x) = Q(x)g(x) + R(x)$$
$$\deg R < \deg g$$

当 $\deg f \geq \deg g$ 时有 $\deg Q = \deg f - \deg g$，否则有 $Q(x) = 0$。我们称 $Q(x)$ 为 $g(x)$ 除 $f(x)$ 的**商（quotient）**，$R(x)$ 为 $g(x)$ 除 $f(x)$ 的**余数（remainder）**。

模多项式

模多项式是多项式环的子环，由多项式环除以同余的等价关系得到。

在上文提到的带余除法中，多项式 $f(x)$ 与它的余式 $R(x)$ 在模多项式 $g(x)$ 的意义下同余。

$$f(x) \equiv R(x) \pmod{g(x)}$$

这个同余式也意味着，对于多项式 $g(x)$ 的任意一个根 x_0，代入 $f(x)$ 和 $R(x)$ 中，得到的点值相同。即：

$$f(x_0) = R(x_0)$$

并且，如果根 x_0 在多项式 $g(x)$ 中的重数是 k，即 $(x-x_0)^k$ 整除 $g(x)$，则对任意大于等于 0 小于 k 的整数 t，有：

$$f^t(x_0) = R^t(x_0)$$

这里的记号表示 t 阶导数。

模多项式同余可以应用于幂级数。一个无限项的幂级数，可以在模具体的多项式情形下，和一个有限项的多项式同余。例如：

$$1 + x + x^2 + x^3 + ... \equiv 1 + x + ... + x^{n-1} \pmod{x^n}$$

显然剩余的所有项都被 x^n 整除，因此模 x^n 的操作等价于「截断」，将无穷项的幂级数截断到前 n 项，直接将更高位的信息丢失。

在一些特定的情况下，也可以模其他的多项式，下文将解释相应情况。

多项式的多点求值和插值

多项式的多点求值（multi-point evaluation）即给出一个多项式 $f(x)$ 和 n 个点 $x_1, x_2, \ldots, x_n$，求

$$f(x_1), f(x_2), \ldots, f(x_n)$$

多项式的插值（interpolation）即给出 $n+1$ 个点

$$(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$$

求一个 n 次多项式 $f(x)$ 使得这 $n+1$ 个点都在 $f(x)$ 上。

这两种操作的实质就是将多项式在**系数表示**和**点值表示**间转化。多点求值将多项式的系数表示转为点值表示，插值将多项式的点值表示转为系数表示。

" 注"

按照幂级数的观点看，多点求值相当于将无穷项的信息「压缩」到有限个点值表示，因此丢失了一些信息，而插值相当于还原到相应次数的系数表示。

编程常见的求值与插值，例如离散傅里叶变换（及其逆变换）等等，选择的 $n+1$ 个点重数均为 1，即两两不同，免去求导的麻烦。

这种「压缩」只保证了在 $n+1$ 个点上的一致。根据上文对模多项式同余的解释，如果幂级数 $f(x)$ 经过在 x_0 到 x_n 点处求值再插值，得到多项式 $R(x)$，则作多项式：

$$g(x) = (x - x_0) \ldots (x - x_n)$$

就有：

$$f(x) \equiv R(x) \pmod{g(x)}$$

由于 $R(x)$ 的次数严格小于 $g(x)$，所以利用求值与插值求出的 $R(x)$ 就是余式。因此在这种情况下，如果幂级数在根处可以求值，就可以模多项式。一个反例例如：

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \ldots$$

在 $x=1$ 处不可求值，因此级数 $1+x+x^2+x^3+...$ 不能模多项式 $x-1$。

由于幂级数的任意阶导数，在 0 处的值总是存在，因此模 x^n 始终可以计算，与上文「截断」的意义一致。离散傅里叶变换（及其逆变换）相当于模多项式 x^n-1。

因式分解和欧几里得

初等数论中的许多结论可以推广到多项式上。

复数域上，由代数基本定理可得，对于 n 次多项式 f，方程

$$f(x)=0$$

有且仅有 n 个解（重根按重数计）。

于是 $f(x)$ 在复数域内可唯一因式分解为如下形式

$$a(x-x_1)^{c_1}(x-x_2)^{c_2}\cdots(x-x_m)^{c_m}$$

$$c_1+c_2+\cdots+c_m=n, x_1,x_2,\cdots,x_m \text{ 互不相同}$$

此时类比正整数的最大公因数，可得多项式的 **最大公因式**（greatest common divisor, gcd）。其可用欧几里得算法求解

$$\gcd(f,0)=f, \gcd(f,g)=\gcd(g,f \bmod g)$$

该性质可以推广到较为一般的情况：

> 对于任意域 P 上的多项式环 $P[x]$，
>
> 多项式均可唯一因式分解，且可用欧几里得算法计算最大公因式。需要注意的是，对于一般环上的多项式，该结论未必成立。

欧几里得算法成立时，可用扩展欧几里得给出不定方程

$$f(x)P(x)+g(x)Q(x)=\gcd(f(x),g(x))$$

的一组特解 $(P(x),Q(x))$，并用 裴蜀定理 判断不定方程

$$f(x)P(x)+g(x)Q(x)=h(x)$$

的可解性。

HALF-GCD[1] 允许我们在 $O(n\log^2 n)$ 时间内计算多项式欧几里得。

模多项式的乘法逆元

在模多项式 $h(x)$ 意义下，幂级数 $f(x)$ 有时存在逆元。逆元就是幂级数 $f(x)$ 的倒数模多项式 $h(x)$ 得到的余式。

这个定义也等价于，对于多项式 $f(x)$，若存在 $g(x)$ 满足：

$$f(x)g(x)\equiv 1 \pmod{h(x)}$$

则称 $g(x)$ 为 $f(x)$ 在模 $h(x)$ 意义下的**逆元（inverse element）**。当多项式欧几里得允许时，逆元存在当且仅当 $\gcd(f,g)=1$。

模多项式 $h(x)$ 意义下逆元总是唯一的。如果多项式 $f(x)$ 的次数也小于 $h(x)$，则得到的 $g(x)$ 与 $f(x)$ 互为逆元。

考虑「截断」的概念，一般把模 x^n 意义下的逆元记作 $f^{-1}(x)$，也是后文默认使用的逆元概念。如果不加说明，逆元的模就是指 x^n。

" 注"

一个问题是，可不可以用各种插值变换，直接求解「逆元」，比如计算：

$$IDFT\left(\frac{DFT(1)}{DFT(f(x))}\right)$$

答案是否定的，不可以这样做。根据上文的解释，这里利用离散傅里叶变换（及其逆变换）直接求出的逆元是模多项式 x^n-1 的逆元，不是通常的模多项式 x^n 的逆元。并且，由于原多项式在各点值处可能为 0，这个求解未必可以进行。

生成函数

生成函数（generating function），又称母函数，是一种形式幂级数，其每一项的系数可以提供关于这个序列的信息。

生成函数有许多不同的种类，但大多可以表示为单一的形式：

$$F(x)=\sum_n a_n k_n(x)$$

其中 $k_n(x)$ 被称为核函数。不同的核函数会导出不同的生成函数，拥有不同的性质。举个例子：

1. 普通生成函数：$k_n(x)=x^n$。
2. 指数生成函数：$k_n(x)=\frac{x^n}{n!}$。
3. 狄利克雷生成函数：$k_n(x)=\frac{1}{n^x}$。

参考资料与拓展阅读

- **Picks's Blog**[2]
- **Miskcoo's Space**[3]
- **Polynomial ring - Wikipedia**[4]
- **Formal power series - Wikipedia**[5]
- 《信息学竞赛中的生成函数计算理论框架》

参考资料与注释

[1] HALF-GCD

[2] **Picks's Blog**

[3] **Miskcoo's Space**

[4] **Polynomial ring - Wikipedia**

[5] **Formal power series - Wikipedia**

9.13.2 代数基本定理

定义

任何复系数一元 n 次多项式（n 至少为 1）方程在复数域上至少有一根。

由此推出，n 次复系数多项式方程在复数域内有且只有 n 个根，重根按重数计算。

有时这个定理也表述为：

任何一个非零的一元 n 次复系数多项式，都正好有 n 个复数根。

代数基本定理的证明，一般会用到复变函数或者近世代数，因此往往作为一个熟知结论直接应用。

根据代数基本定理，一个复系数多项式 $f(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_0$ 一定可以唯一地分解为：

$$f(x) = a_n (x - x_1)^{k_1} (x - x_2)^{k_2} ... (x - x_t)^{k_t}$$

其中各个根均为复数，$k_1 + k_2 + ... + k_t = n$。

虚根成对定理

代数基本定理的研究对象是复系数多项式。当对实系数多项式进行研究时，虽然也能分解出复数根，却需要将研究范围扩大，不太方便。

虚根：非实数根。

定理：实系数多项式的根的共轭复数也是该多项式的根。

证明：直接在代数基本定理的等式两端取共轭即证毕。

如果根本身是实数，则取共轭仍为它本身，不受影响。

如果根是虚根，则虚根的共轭复数也是原多项式的根。那么，两个虚根就可以配对。

定理：实数系数方程的共轭虚根一定成对出现，并且共轭虚根的重数相等。

证明：假设一个根为 $a + b\mathrm{i}$，则另一个根为 $a - b\mathrm{i}$。这意味着在分解式中存在两项：

$$(x - a - b\mathrm{i})(x - a + b\mathrm{i}) = x^2 - 2ax + a^2 + b^2$$

可以看到两项乘在一起，各项系数会全部变为实数。这个等式右端的二次实系数多项式整除原始的多项式。

于是，在代数基本定理的等式中，两遍同时除以这个二次三项式，得到的仍旧是实系数多项式的等式。对新等式重复操作，随着次数的下降，若干次后即不存在虚根。

因此，每对共轭虚根的重数相等。证毕。

以下是虚根成对定理的推论：

- 实系数奇次多项式至少有一个实根，并且总共有奇数个实根。
- 实系数偶次多项式可能没有实根，总共有偶数个实根。

称上述二次三项式 $x^2 - 2ax + a^2 + b^2 = x^2 + px + q$ 为二次实系数不可约因式。不可约是指它在实数范围内不可约。

定理：实系数多项式一定是一次或者二次实系数不可约因式的积。

证明：

只要实系数多项式有一个实根 c，就有一个实系数因式 $x - c$ 和它对应；有一对虚根 $a \pm b\mathrm{i}$，就有一个实系数因式 $x^2 - 2ax + a^2 + b^2$ 和它对应。

因此，只要在原始的代数基本定理分解式中，利用虚根成对定理进行配对，即证毕。

根据虚根成对定理，一个实系数多项式 $f(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_0$ 一定可以唯一地分解为：

$$f(x) = a_n (x - x_1)^{k_1} (x - x_2)^{k_2} ... (x - x_t)^{k_t} (x^2 + p_1 x + q_1)^{l_1} (x^2 + p_2 x + q_2)^{l_2} ... (x^2 + p_s x + q_s)^{l_s}$$

其中各项系数均为实数，$k_1 + k_2 + ... + k_t + 2(l_1 + l_2 + ... + l_s) = n$。

林士谔算法

简介

怎样对实系数多项式进行代数基本定理的分解？如果将数域扩充至复数会很复杂。

如果只在实数范围内进行分解，只能保证，当次数大于 2 的时候，一定存在实系数二次三项式因式。

这是因为，如果该多项式有虚根，直接凑出一对共轭虚根即可。如果该多项式只有实根，任取两个实根对应的一次因式乘在一起，也能得到实系数二次三项式因式。

找到二次三项式因式之后，再从二次式中解实根或复根就极为容易。于是便有逐次**找出一个二次因子**来求得方程的复根的计算方法，这种方法避免了复数运算。

在 1940 年 8 月、1943 年 8 月和 1947 年 7 月，林士谔先后在 MIT 出版的《数学物理》杂志上接连正式发表了 3 篇关于解算高阶方程式复根方法的论文[1]，每次均有改进。

这个方法今天还在现代计算机中进行快速运算，计算机程序包（如 MATLAB）中的多项式求根程序依据的原理也是这个算法。

过程

要想找到一个二次三项式因子，就要将多项式分解为：

$$f(x) = (x^2 + p_1x + q_1)g(x)$$

由于无法一下子找到二次三项式因子，按照迭代求解的思路，对于初始值有：

$$f(x) = (x^2 + px + q)g(x) + rx + s$$

会产生一个一次式作为余项。只要余项足够小，即可近似地找到待求因子。

我们希望最终解是初始值加一个偏移修正：

$$p_1 = p + dp$$

$$q_1 = q + dq$$

余式中的两个数 (r, s) 由除式的给定系数 (p, q) 决定。有偏导数关系：

$$dr = \frac{\partial r}{\partial p}dp + \frac{\partial r}{\partial q}dq$$

$$ds = \frac{\partial s}{\partial p}dp + \frac{\partial s}{\partial q}dq$$

在初始的等式中，被除式 $f(x)$ 是给定的，商式 $g(x)$ 和余式 $rx + s$ 随着除式 $x^2 + px + q$ 的变化而变化。因此有偏导数关系

$$0 = xg(x) + \frac{\partial g(x)}{\partial p}(x^2 + px + q) + \frac{\partial r}{\partial p}x + \frac{\partial s}{\partial p}$$

$$0 = g(x) + \frac{\partial g(x)}{\partial q}(x^2 + px + q) + \frac{\partial r}{\partial q}x + \frac{\partial s}{\partial q}$$

注意到，偏导数只是一个数值，与变元 x 无关。因此有整除关系

$$xg(x) = -\frac{\partial g(x)}{\partial p}(x^2 + px + q) - \frac{\partial r}{\partial p}x - \frac{\partial s}{\partial p}$$

$$g(x) = -\frac{\partial g(x)}{\partial q}(x^2 + px + q) - \frac{\partial r}{\partial q}x - \frac{\partial s}{\partial q}$$

这里的结论是，待求的偏导数，恰好是对商式继续做除法的余式。多项式对给定二次三项式的除法，直接计算即可。这里就求得了四个偏导数。

我们希望 s 和 r 加上偏移 ds 与 dr 得到 0，即 ds 与 dr 是 s 和 r 的相反数。因此要解方程：

$$-\frac{\partial r}{\partial p}dp - \frac{\partial r}{\partial q}dq = r$$

$$-\frac{\partial s}{\partial p}dp - \frac{\partial s}{\partial q}dq = s$$

从上述方程组中解得 p 和 q 相应的偏移 dp 和 dq，直接用二阶行列式求解即可。

实现

```
// a 是原始的多项式，n 是多项式次数，p 是待求的一次项，q 是待求的常数项
void Shie(double a[], int n, double *p, double *q) {
  // 数组 b 是多项式 a 除以当前迭代二次三项式的商
  memset(b, 0, sizeof(b));
  // 数组 c 是多项式 b 乘以 x 平方再除以当前迭代二次三项式的商
  memset(c, 0, sizeof(c));
  *p = 0;
  *q = 0;
  double dp = 1;
  double dq = 1;
  while (dp > eps || dp < -eps || dq > eps || dq < -eps)  // eps 自行设定
  {
    double p0 = p;
    double q0 = q;
    b[n - 2] = a[n];
    c[n - 2] = b[n - 2];
    b[n - 3] = a[n - 1] - p0 * b[n - 2];
    c[n - 3] = b[n - 3] - p0 * b[n - 2];
    int j;
    for (j = n - 4; j >= 0; j--) {
      b[j] = a[j + 2] - p0 * b[j + 1] - q0 * b[j + 2];
      c[j] = b[j] - p0 * c[j + 1] - q0 * c[j + 2];
    }
    double r = a[1] - p0 * b[0] - q0 * b[1];
    double s = a[0] - q0 * b[0];
    double rp = c[1];
    double sp = b[0] - q0 * c[2];
    double rq = c[0];
    double sq = -q0 * c[1];
    dp = (rp * s - r * sp) / (rp * sq - rq * sp);
    dq = (r * sq - rq * s) / (rp * sq - rq * sp);
    *p += dp;
    *q += dq;
  }
}
```

参考资料与注释

[1] 林士谔 。 论劈因法解高阶特征方程根值的应用问题 。 数学进展 ，1963(03):207-217.

9.13.3　快速傅里叶变换

前置知识：复数。

本文将介绍一种算法，它支持在 $O(n\log n)$ 的时间内计算两个 n 度的多项式的乘法，比朴素的 $O(n^2)$ 算法更高效。由于两个整数的乘法也可以被当作多项式乘法，因此这个算法也可以用来加速大整数的乘法计算。

引入

我们现在引入两个多项式 A 和 B：

$$A = 5x^2 + 3x + 7$$
$$B = 7x^2 + 2x + 1$$

两个多项式相乘的积 $C = A \times B$，我们可以在 $O(n^2)$ 的时间复杂度中解得（这里 n 为 A 或者 B 多项式的度）：

$$\begin{aligned} C &= A \times B \\ &= 35x^4 + 31x^3 + 60x^2 + 17x + 7 \end{aligned}$$

很明显，多项式 C 的系数 c_i 满足 $c_i = \sum_{j=0}^{i} a_j b_{i-j}$。而对于这种朴素算法而言，计算每一项的时间复杂度都为 $O(n)$，一共有 $O(n)$ 项，那么时间复杂度为 $O(n^2)$。

能否加速使得它的时间复杂度降低呢？如果使用快速傅里叶变换的话，那么我们可以使得其复杂度降低到 $O(n\log n)$。

傅里叶变换

傅里叶变换（Fourier Transform）是一种分析信号的方法，它可分析信号的成分，也可用这些成分合成信号。许多波形可作为信号的成分，傅里叶变换用正弦波作为信号的成分。

设 $f(t)$ 是关于时间 t 的函数，则傅里叶变换可以检测频率 ω 的周期在 $f(t)$ 出现的程度：

$$F(\omega) = \mathbb{F}[f(t)] = \int_{-\infty}^{\infty} f(t)\mathrm{e}^{-\mathrm{i}\omega t}dt$$

它的逆变换是

$$f(t) = \mathbb{F}^{-1}[F(\omega)] = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)\mathrm{e}^{\mathrm{i}\omega t}d\omega$$

逆变换的形式与正变换非常类似，分母 2π 恰好是指数函数的周期。

傅里叶变换相当于将时域的函数与周期为 2π 的复指数函数进行连续的内积。逆变换仍旧为一个内积。

傅里叶变换有相应的卷积定理，可以将时域的卷积转化为频域的乘积，也可以将频域的卷积转化为时域的乘积。

离散傅里叶变换

离散傅里叶变换（Discrete Fourier transform，DFT）是傅里叶变换在时域和频域上都呈离散的形式，将信号的时域采样变换为其 DTFT（discrete-time Fourier transform）的频域采样。

傅里叶变换是积分形式的连续的函数内积，离散傅里叶变换是求和形式的内积。

设 $\{x_n\}_{n=0}^{N-1}$ 是某一满足有限性条件的序列，它的离散傅里叶变换（DFT）为：

$$X_k = \sum_{n=0}^{N-1} x_n \mathrm{e}^{-\mathrm{i}\frac{2\pi}{N}kn}$$

其中 e 是自然对数的底数，i 是虚数单位。通常以符号 $\mathcal{F}$ 表示这一变换，即

$$\hat{x} = \mathcal{F}x$$

类似于积分形式，它的**逆离散傅里叶变换**（IDFT）为：

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k \mathrm{e}^{\mathrm{i}\frac{2\pi}{N}kn}$$

可以记为：

$$x = \mathcal{F}^{-1}\hat{x}$$

实际上，DFT 和 IDFT 变换式中和式前面的归一化系数并不重要。在上面的定义中，DFT 和 IDFT 前的系数分别为 1 和 $\frac{1}{N}$。有时我们会将这两个系数都改 $\frac{1}{\sqrt{N}}$。

离散傅里叶变换仍旧是时域到频域的变换。由于求和形式的特殊性，可以有其他的解释方法。

如果把序列 x_n 看作多项式 $f(x)$ 的 x^n 项系数，则计算得到的 X_k 恰好是多项式 $f(x)$ 代入单位根 $\mathrm{e}^{\frac{-2\pi \mathrm{i}k}{N}}$ 的点值 $f(\mathrm{e}^{\frac{-2\pi \mathrm{i}k}{N}})$。

这便构成了卷积定理的另一种解释办法，即对多项式进行特殊的插值操作。离散傅里叶变换恰好是多项式在单位根处进行插值。

例如计算：

$$\binom{n}{3} + \binom{n}{7} + \binom{n}{11} + \binom{n}{15} + \dots$$

定义函数 $f(x)$ 为：

$$f(x) = (1+x)^n = \binom{n}{0}x^0 + \binom{n}{1}x^1 + \binom{n}{2}x^2 + \binom{n}{3}x^3 + \dots$$

然后可以发现，代入四次单位根 $f(\mathrm{i})$ 得到这样的序列：

$$f(\mathrm{i}) = (1+\mathrm{i})^n = \binom{n}{0} + \binom{n}{1}\mathrm{i} - \binom{n}{2} - \binom{n}{3}\mathrm{i} + \dots$$

于是下面的求和恰好可以把其余各项消掉：

$$f(1) + \mathrm{i}f(\mathrm{i}) - f(-1) - \mathrm{i}f(-\mathrm{i}) = 4\binom{n}{3} + 4\binom{n}{7} + 4\binom{n}{11} + 4\binom{n}{15} + \dots$$

因此这道数学题的答案为：

$$\binom{n}{3} + \binom{n}{7} + \binom{n}{11} + \binom{n}{15} + \dots = \frac{2^n + \mathrm{i}(1+\mathrm{i})^n - \mathrm{i}(1-\mathrm{i})^n}{4}$$

这道数学题在单位根处插值，恰好构成离散傅里叶变换。

矩阵公式

由于离散傅立叶变换是一个**线性**算子，所以它可以用矩阵乘法来描述。在矩阵表示法中，离散傅立叶变换表示如下：

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ \vdots \\ X_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \alpha & \alpha^2 & \cdots & \alpha^{n-1} \\ 1 & \alpha^2 & \alpha^4 & \cdots & \alpha^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha^{n-1} & \alpha^{2(n-1)} & \cdots & \alpha^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{bmatrix}$$

其中 $\alpha = \mathrm{e}^{-\mathrm{i}\frac{2\pi}{N}n}$。

快速傅里叶变换

FFT 是一种高效实现 DFT 的算法，称为快速傅立叶变换（Fast Fourier Transform，FFT）。它对傅里叶变换的理论并没有新的发现，但是对于在计算机系统或者说数字系统中应用离散傅立叶变换，可以说是进了一大步。快速数论变换（NTT）是快速傅里叶变换（FFT）在数论基础上的实现。

在 1965 年，Cooley 和 Tukey 发表了快速傅里叶变换算法。事实上 FFT 早在这之前就被发现过了，但是在当时现代计算机并未问世，人们没有意识到 FFT 的重要性。一些调查者认为 FFT 是由 Runge 和 König 在 1924 年发现的。但事实上高斯早在 1805 年就发明了这个算法，但一直没有发表。

分治法实现

FFT 算法的基本思想是分治。就 DFT 来说，它分治地来求当 $x = \omega_n^k$ 的时候 $f(x)$ 的值。基 - 2 FFT 的分治思想体现在将多项式分为奇次项和偶次项处理。

举个例子，对于一共 8 项的多项式：

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7$$

按照次数的奇偶来分成两组，然后右边提出来一个 x：

$$\begin{aligned} f(x) &= (a_0 + a_2x^2 + a_4x^4 + a_6x^6) + (a_1x + a_3x^3 + a_5x^5 + a_7x^7) \\ &= (a_0 + a_2x^2 + a_4x^4 + a_6x^6) + x(a_1 + a_3x^2 + a_5x^4 + a_7x^6) \end{aligned}$$

分别用奇偶次次项数建立新的函数：

$$\begin{aligned} G(x) &= a_0 + a_2x + a_4x^2 + a_6x^3 \\ H(x) &= a_1 + a_3x + a_5x^2 + a_7x^3 \end{aligned}$$

那么原来的 $f(x)$ 用新函数表示为：

$$f(x) = G\left(x^2\right) + x \times H\left(x^2\right)$$

利用偶数次单位根的性质 $\omega_n^i = -\omega_n^{i+n/2}$，和 $G\left(x^2\right)$ 和 $H\left(x^2\right)$ 是偶函数，我们知道在复平面上 ω_n^i 和 $\omega_n^{i+n/2}$ 的 $G(x^2)$ 的 $H(x^2)$ 对应的值相同。得到：

$$\begin{aligned} f(\omega_n^k) &= G((\omega_n^k)^2) + \omega_n^k \times H((\omega_n^k)^2) \\ &= G(\omega_n^{2k}) + \omega_n^k \times H(\omega_n^{2k}) \\ &= G(\omega_{n/2}^k) + \omega_n^k \times H(\omega_{n/2}^k) \end{aligned}$$

和：

$$\begin{aligned} f(\omega_n^{k+n/2}) &= G(\omega_n^{2k+n}) + \omega_n^{k+n/2} \times H(\omega_n^{2k+n}) \\ &= G(\omega_n^{2k}) - \omega_n^k \times H(\omega_n^{2k}) \\ &= G(\omega_{n/2}^k) - \omega_n^k \times H(\omega_{n/2}^k) \end{aligned}$$

因此我们求出了 $G(\omega_{n/2}^k)$ 和 $H(\omega_{n/2}^k)$ 后，就可以同时求出 $f(\omega_n^k)$ 和 $f(\omega_n^{k+n/2})$。于是对 G 和 H 分别递归 DFT 即可。

考虑到分治 DFT 能处理的多项式长度只能是 $2^m(m \in \mathbf{N}^*)$，否则在分治的时候左右不一样长，右边就取不到系数了。所以要在第一次 DFT 之前就把序列向上补成长度为 $2^m(m \in \mathbf{N}^*)$（高次系数补 0）、最高项次数为 $2^m - 1$ 的多项式。

在代入值的时候，因为要代入 n 个不同值，所以我们代入 $\omega_n^0, \omega_n^1, \omega_n^2, \cdots, \omega_n^{n-1}(n = 2^m(m \in \mathbf{N}^*))$ 一共 2^m 个不同值。

代码实现方面，STL 提供了复数的模板，当然也可以手动实现。两者区别在于，使用 STL 的 `complex` 可以调用 `exp` 函数求出 ω_n。但事实上使用欧拉公式得到的虚数来求 ω_n 也是等价的。

以上就是 FFT 算法中 DFT 的介绍，它将一个多项式从系数表示法变成了点值表示法。

值的注意的是，因为是单位复根，所以说我们需要令 n 项式的高位补为零，使得 $n = 2^k, k \in \mathbf{N}^*$。

" 递归版 FFT"

```
#include <cmath>
#include <complex>

typedef std::complex<double> Comp;  // STL complex

const Comp I(0, 1);  // i
const int MAX_N = 1 << 20;
```

```cpp
Comp tmp[MAX_N];

// rev=1,DFT; rev=-1,IDFT
void DFT(Comp* f, int n, int rev) {
  if (n == 1) return;
  for (int i = 0; i < n; ++i) tmp[i] = f[i];
  // 偶数放左边，奇数放右边
  for (int i = 0; i < n; ++i) {
    if (i & 1)
      f[n / 2 + i / 2] = tmp[i];
    else
      f[i / 2] = tmp[i];
  }
  Comp *g = f, *h = f + n / 2;
  // 递归 DFT
  DFT(g, n / 2, rev), DFT(h, n / 2, rev);
  // cur 是当前单位复根，对于 k = 0 而言，它对应的单位复根 omega^0_n = 1。
  // step 是两个单位复根的差，即满足 omega^k_n = step*omega^{k-1}*n,
  // 定义等价于 exp(I*(2*M_PI/n*rev))
  Comp cur(1, 0), step(cos(2 * M_PI / n), sin(2 * M_PI * rev / n));
  for (int k = 0; k < n / 2;
       ++k) {  // F(omega^k_n) = G(omega^k*{n/2}) + omega^k*n\*H(omega^k*{n/2})
    tmp[k] = g[k] + cur * h[k];
    // F(omega^{k+n/2}*n) = G(omega^k*{n/2}) - omega^k_n*H(omega^k\_{n/2})
    tmp[k + n / 2] = g[k] - cur * h[k];
    cur *= step;
  }
  for (int i = 0; i < n; ++i) f[i] = tmp[i];
}
```

时间复杂度 $O(n \log n)$。

倍增法实现

这个算法还可以从「分治」的角度继续优化。对于基 - 2 FFT，我们每一次都会把整个多项式的奇数次项和偶数次项系数分开，一直分到只剩下一个系数。但是，这个递归的过程需要更多的内存。因此，我们可以先「模仿递归」把这些系数在原数组中「拆分」，然后再「倍增」地去合并这些算出来的值。

对于「拆分」，可以使用位逆序置换实现。

对于「合并」，使用蝶形运算优化可以做到只用 $O(1)$ 的额外空间来完成。

位逆序置换

以 8 项多项式为例，模拟拆分的过程：

- 初始序列为 $\{x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$
- 一次二分之后 $\{x_0, x_2, x_4, x_6\}, \{x_1, x_3, x_5, x_7\}$
- 两次二分之后 $\{x_0, x_4\}\{x_2, x_6\}, \{x_1, x_5\}, \{x_3, x_7\}$
- 三次二分之后 $\{x_0\}\{x_4\}\{x_2\}\{x_6\}\{x_1\}\{x_5\}\{x_3\}\{x_7\}$

规律：其实就是原来的那个序列，每个数用二进制表示，然后把二进制翻转对称一下，就是最终那个位置的下标。比如 x_1 是 001，翻转是 100，也就是 4，而且最后那个位置确实是 4。我们称这个变换为位逆序置换（bit-reversal permutation），证明留给读者自证。

根据它的定义，我们可以在 $O(n \log n)$ 的时间内求出每个数变换后的结果：

" 位逆序置换实现（$O(n\log n)$）"

```
/*
 * 进行 FFT 和 IFFT 前的反置变换
 * 位置 i 和 i 的二进制反转后的位置互换
 * len 必须为 2 的幂
 */
void change(Complex y[], int len) {
  // 一开始 i 是 0...01，而 j 是 10...0，在二进制下相反对称。
  // 之后 i 逐渐加一，而 j 依然维持着和 i 相反对称，一直到 i = 1...11。
  for (int i = 1, j = len / 2, k; i < len - 1; i++) {
    // 交换互为小标反转的元素，i < j 保证交换一次
    if (i < j) swap(y[i], y[j]);
    // i 做正常的 + 1，j 做反转类型的 + 1，始终保持 i 和 j 是反转的。
    // 这里 k 代表了 0 出现的最高位。j 先减去高位的全为 1 的数字，直到遇到了
    // 0，之后再加上即可。
    k = len / 2;
    while (j >= k) {
      j = j - k;
      k = k / 2;
    }
    if (j < k) j += k;
  }
}
```

实际上，位逆序置换可以 $O(n)$ 从小到大递推实现，设 $len=2^k$，其中 k 表示二进制数的长度，设 $R(x)$ 表示长度为 k 的二进制数 x 翻转后的数（高位补 0）。我们要求的是 $R(0),R(1),\cdots,R(n-1)$。

首先 $R(0)=0$。

我们从小到大求 $R(x)$。因此在求 $R(x)$ 时，$R\left(\left\lfloor\frac{x}{2}\right\rfloor\right)$ 的值是已知的。因此我们把 x 右移一位（除以 2），然后翻转，再右移一位，就得到了 x **除了（二进制）个位**之外其它位的翻转结果。

考虑个位的翻转结果：如果个位是 0，翻转之后最高位就是 0。如果个位是 1，则翻转后最高位是 1，因此还要加上 $\dfrac{len}{2}=2^{k-1}$。综上

$$R(x)=\left\lfloor\frac{R\left(\left\lfloor\frac{x}{2}\right\rfloor\right)}{2}\right\rfloor+(x \bmod 2)\times\frac{len}{2}$$

举个例子：设 $k=5$，$len=(100000)_2$。为了翻转 $(11001)_2$：

1. 考虑 $(1100)_2$，我们知道 $R((1100)_2)=R((01100)_2)=(00110)_2$，再右移一位就得到了 $(00011)_2$。
2. 考虑个位，如果是 1，它就要翻转到数的最高位，即翻转数加上 $(10000)_2=2^{k-1}$，如果是 0 则不用更改。

" 位逆序置换实现（$O(n)$）"

```
// 同样需要保证 len 是 2 的幂
// 记 rev[i] 为 i 翻转后的值
void change(Complex y[], int len) {
  for (int i = 0; i < len; ++i) {
    rev[i] = rev[i >> 1] >> 1;
    if (i & 1) {  // 如果最后一位是 1，则翻转成 len/2
      rev[i] |= len >> 1;
    }
  }
  for (int i = 0; i < len; ++i) {
```

```
    if (i < rev[i]) {  // 保证每对数只翻转一次
      swap(y[i], y[rev[i]]);
    }
  }
  return;
}
```

蝶形运算优化

已知 $G(\omega_{n/2}^k)$ 和 $H(\omega_{n/2}^k)$ 后，需要使用下面两个式子求出 $f(\omega_n^k)$ 和 $f(\omega_n^{k+n/2})$：

$$\begin{aligned} f(\omega_n^k) &= G(\omega_{n/2}^k) + \omega_n^k \times H(\omega_{n/2}^k) \\ f(\omega_n^{k+n/2}) &= G(\omega_{n/2}^k) - \omega_n^k \times H(\omega_{n/2}^k) \end{aligned}$$

使用位逆序置换后，对于给定的 n, k：

- $G(\omega_{n/2}^k)$ 的值存储在数组下标为 k 的位置，$H(\omega_{n/2}^k)$ 的值存储在数组下标为 $k + \dfrac{n}{2}$ 的位置。
- $f(\omega_n^k)$ 的值将存储在数组下标为 k 的位置，$f(\omega_n^{k+n/2})$ 的值将存储在数组下标为 $k + \dfrac{n}{2}$ 的位置。

因此可以直接在数组下标为 k 和 $k + \frac{n}{2}$ 的位置进行覆写，而不用开额外的数组保存值。此方法即称为**蝶形运算**，或更准确的，基 - 2 蝶形运算。

再详细说明一下如何借助蝶形运算完成所有段长度为 $\frac{n}{2}$ 的合并操作：

1. 令段长度为 $s = \frac{n}{2}$；
2. 同时枚举序列 $\{G(\omega_{n/2}^k)\}$ 的左端点 $l_g = 0, 2s, 4s, \cdots, N - 2s$ 和序列 $\{H(\omega_{n/2}^k)\}$ 的左端点 $l_h = s, 3s, 5s, \cdots, N - s$；
3. 合并两个段时，枚举 $k = 0, 1, 2, \cdots, s - 1$，此时 $G(\omega_{n/2}^k)$ 存储在数组下标为 $l_g + k$ 的位置，$H(\omega_{n/2}^k)$ 存储在数组下标为 $l_h + k$ 的位置；
4. 使用蝶形运算求出 $f(\omega_n^k)$ 和 $f(\omega_n^{k+n/2})$，然后直接在原位置覆写。

快速傅里叶逆变换

傅里叶逆变换可以用傅里叶变换表示。对此我们有两种理解方式。

线性代数角度

IDFT（傅里叶反变换）的作用，是把目标多项式的点值形式转换成系数形式。而 DFT 本身是个线性变换，可以理解为将目标多项式当作向量，左乘一个矩阵得到变换后的向量，以模拟把单位复根代入多项式的过程：

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n^1 & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

现在我们已经得到最左边的结果了，中间的 x 值在目标多项式的点值表示中也是一一对应的，所以，根据矩阵的基础知识，我们只要在式子两边左乘中间那个大矩阵的逆矩阵就行了。

由于这个矩阵的元素非常特殊，它的逆矩阵也有特殊的性质，就是每一项**取倒数**，再**除以变换的长度** n，就能得到它的逆矩阵。

注意：傅里叶变换的长度，并不是多项式的长度，变换的长度应比乘积多项式的长度长。待相乘的多项式不够长，需要在高次项处补 0。

为了使计算的结果为原来的倒数，根据欧拉公式，可以得到

$$\frac{1}{\omega_k} = \omega_k^{-1} = \mathrm{e}^{-\frac{2\pi \mathrm{i}}{k}} = \cos\left(\frac{2\pi}{k}\right) + \mathrm{i}\sin\left(-\frac{2\pi}{k}\right)$$

因此我们可以尝试着把单位根 ω_k 取成 $\mathrm{e}^{-\frac{2\pi \mathrm{i}}{k}}$，这样我们的计算结果就会变成原来的倒数，之后唯一多的操作就只有再**除以它的长度** n，而其它的操作过程与 DFT 是完全相同的。我们可以定义一个函数，在里面加一个参数 1 或者是 -1，然后把它乘到 π 上。传入 1 就是 DFT，传入 -1 就是 IDFT。

单位复根周期性

利用单位复根的周期性同样可以理解 IDFT 与 DFT 之间的关系。

考虑原本的多项式是 $f(x)=a_0+a_1x+a_2x^2+\cdots+a_{n-1}x^{n-1}=\sum_{i=0}^{n-1}a_ix^i$。而 IDFT 就是把你的点值表示还原为系数表示。

考虑**构造法**。我们已知 $y_i=f\left(\omega_n^i\right), i\in\{0,1,\cdots,n-1\}$，求 $\{a_0,a_1,\cdots,a_{n-1}\}$。构造多项式如下

$$A(x)=\sum_{i=0}^{n-1}y_ix^i$$

相当于把 $\{y_0,y_1,y_2,\cdots,y_{n-1}\}$ 当做多项式 A 的系数表示法。

这时我们有两种推导方式，这对应了两种实现方法。

方法一

设 $b_i=\omega_n^{-i}$，则多项式 A 在 $x=b_0,b_1,\cdots,b_{n-1}$ 处的点值表示法为 $\{A(b_0),A(b_1),\cdots,A(b_{n-1})\}$。

对 $A(x)$ 的定义式做一下变换，可以将 $A(b_k)$ 表示为

$$\begin{aligned}A(b_k)&=\sum_{i=0}^{n-1}f(\omega_n^i)\omega_n^{-ik}=\sum_{i=0}^{n-1}\omega_n^{-ik}\sum_{j=0}^{n-1}a_j(\omega_n^i)^j\\&=\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}a_j\omega_n^{i(j-k)}=\sum_{j=0}^{n-1}a_j\sum_{i=0}^{n-1}\left(\omega_n^{j-k}\right)^i\end{aligned}$$

记 $S\left(\omega_n^a\right)=\sum_{i=0}^{n-1}\left(\omega_n^a\right)^i$。

当 $a=0 \pmod n$ 时，$S\left(\omega_n^a\right)=n$。

当 $a\neq 0 \pmod n$ 时，我们错位相减

$$\begin{aligned}S\left(\omega_n^a\right)&=\sum_{i=0}^{n-1}\left(\omega_n^a\right)^i\\\omega_n^aS\left(\omega_n^a\right)&=\sum_{i=1}^{n}\left(\omega_n^a\right)^i\\S\left(\omega_n^a\right)&=\frac{\left(\omega_n^a\right)^n-\left(\omega_n^a\right)^0}{\omega_n^a-1}=0\end{aligned}$$

也就是说

$$S\left(\omega_n^a\right)=\begin{cases}n, & a=0\\0, & a\neq 0\end{cases}$$

那么代回原式

$$A(b_k)=\sum_{j=0}^{n-1}a_jS\left(\omega_n^{j-k}\right)=a_k\cdot n$$

也就是说给定点 $b_i=\omega_n^{-i}$，则 A 的点值表示法为

$$\begin{aligned}&\{(b_0,A(b_0)),(b_1,A(b_1)),\cdots,(b_{n-1},A(b_{n-1}))\}\\=&\{(b_0,a_0\cdot n),(b_1,a_1\cdot n),\cdots,(b_{n-1},a_{n-1}\cdot n)\}\end{aligned}$$

综上所述，我们取单位根为其倒数，对 $\{y_0,y_1,y_2,\cdots,y_{n-1}\}$ 跑一遍 FFT，然后除以 n 即可得到 $f(x)$ 的系数表示。

方法二

我们直接将 ω_n^i 代入 $A(x)$。

推导的过程与方法一大同小异，最终我们得到 $A(\omega_n^k)=\sum_{j=0}^{n-1} a_j S\left(\omega_n^{j+k}\right)$。

当且仅当 $j+k=0 \pmod n$ 时有 $S\left(\omega_n^{j+k}\right)=n$，否则为 0。因此 $A(\omega_n^k)=a_{n-k}\cdot n$。

这意味着我们将 $\{y_0,y_1,y_2,\cdots,y_{n-1}\}$ 做 DFT 变换后除以 n，再反转后 $n-1$ 个元素，同样可以还原 $f(x)$ 的系数表示。

代码实现

所以我们 FFT 函数可以集 DFT 和 IDFT 于一身。代码实现如下：

" 非递归版 FFT（对应方法一）"

```
/*
 * 做 FFT
 * len 必须是 2^k 形式
 * on == 1 时是 DFT，on == -1 时是 IDFT
 */
void fft(Complex y[], int len, int on) {
  // 位逆序置换
  change(y, len);
  // 模拟合并过程，一开始，从长度为一合并到长度为二，一直合并到长度为 len。
  for (int h = 2; h <= len; h <<= 1) {
    // wn：当前单位复根的间隔：w^1_h
    Complex wn(cos(2 * PI / h), sin(on * 2 * PI / h));
    // 合并，共 len / h 次。
    for (int j = 0; j < len; j += h) {
      // 计算当前单位复根，一开始是 1 = w^0_n，之后是以 wn 为间隔递增： w^1_n
      // ...
      Complex w(1, 0);
      for (int k = j; k < j + h / 2; k++) {
        // 左侧部分和右侧是子问题的解
        Complex u = y[k];
        Complex t = w * y[k + h / 2];
        // 这就是把两部分分治的结果加起来
        y[k] = u + t;
        y[k + h / 2] = u - t;
        // 后半个 「step」中的 ω 一定和 「前半个」中的成相反数
        // 「红圈」上的点转一整圈「转回来」，转半圈正好转成相反数
        // 一个数相反数的平方与这个数自身的平方相等
        w = w * wn;
      }
    }
  }
  // 如果是 IDFT，它的逆矩阵的每一个元素不只是原元素取倒数，还要除以长度 len。
  if (on == -1) {
    for (int i = 0; i < len; i++) {
      y[i].x /= len;
    }
  }
}
```

" 非递归版 FFT（对应方法二）"

```
/*
 * 做 FFT
 * len 必须是 2^k 形式
 * on == 1 时是 DFT，on == -1 时是 IDFT
 */
void fft(Complex y[], int len, int on) {
  change(y, len);
  for (int h = 2; h <= len; h <<= 1) {             // 模拟合并过程
    Complex wn(cos(2 * PI / h), sin(2 * PI / h));  // 计算当前单位复根
    for (int j = 0; j < len; j += h) {
      Complex w(1, 0);  // 计算当前单位复根
      for (int k = j; k < j + h / 2; k++) {
        Complex u = y[k];
        Complex t = w * y[k + h / 2];
        y[k] = u + t;  // 这就是把两部分分治的结果加起来
        y[k + h / 2] = u - t;
        // 后半个 「step」中的 ω 一定和 「前半个」中的成相反数
        // 「红圈」上的点转一整圈「转回来」，转半圈正好转成相反数
        // 一个数相反数的平方与这个数自身的平方相等
        w = w * wn;
      }
    }
  }
  if (on == -1) {
    reverse(y + 1, y + len);
    for (int i = 0; i < len; i++) {
      y[i].x /= len;
    }
  }
}
```

"FFT 模板（HDU 1402 - A * B Problem Plus[1]）"

```
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>

const double PI = acos(-1.0);

struct Complex {
  double x, y;

  Complex(double _x = 0.0, double _y = 0.0) {
    x = _x;
    y = _y;
  }

  Complex operator-(const Complex &b) const {
    return Complex(x - b.x, y - b.y);
  }
```

```
  Complex operator+(const Complex &b) const {
    return Complex(x + b.x, y + b.y);
  }

  Complex operator*(const Complex &b) const {
    return Complex(x * b.x - y * b.y, x * b.y + y * b.x);
  }
};

/*
 * 进行 FFT 和 IFFT 前的反置变换
 * 位置 i 和 i 的二进制反转后的位置互换
 *len 必须为 2 的幂
 */
void change(Complex y[], int len) {
  int i, j, k;

  for (int i = 1, j = len / 2; i < len - 1; i++) {
    if (i < j) std::swap(y[i], y[j]);

    // 交换互为小标反转的元素，i<j 保证交换一次
    // i 做正常的 + 1，j 做反转类型的 + 1，始终保持 i 和 j 是反转的
    k = len / 2;

    while (j >= k) {
      j = j - k;
      k = k / 2;
    }

    if (j < k) j += k;
  }
}

/*
 * 做 FFT
 *len 必须是 2^k 形式
 *on == 1 时是 DFT，on == -1 时是 IDFT
 */
void fft(Complex y[], int len, int on) {
  change(y, len);

  for (int h = 2; h <= len; h <<= 1) {
    Complex wn(cos(2 * PI / h), sin(on * 2 * PI / h));

    for (int j = 0; j < len; j += h) {
      Complex w(1, 0);

      for (int k = j; k < j + h / 2; k++) {
        Complex u = y[k];
        Complex t = w * y[k + h / 2];
        y[k] = u + t;
        y[k + h / 2] = u - t;
        w = w * wn;
```

```
      }
    }
  }

  if (on == -1) {
    for (int i = 0; i < len; i++) {
      y[i].x /= len;
    }
  }
}

const int MAXN = 200020;
Complex x1[MAXN], x2[MAXN];
char str1[MAXN / 2], str2[MAXN / 2];
int sum[MAXN];

int main() {
  while (scanf("%s%s", str1, str2) == 2) {
    int len1 = strlen(str1);
    int len2 = strlen(str2);
    int len = 1;

    while (len < len1 * 2 || len < len2 * 2) len <<= 1;

    for (int i = 0; i < len1; i++) x1[i] = Complex(str1[len1 - 1 - i] - '0', 0);


    for (int i = len1; i < len; i++) x1[i] = Complex(0, 0);

    for (int i = 0; i < len2; i++) x2[i] = Complex(str2[len2 - 1 - i] - '0', 0);


    for (int i = len2; i < len; i++) x2[i] = Complex(0, 0);

    fft(x1, len, 1);
    fft(x2, len, 1);

    for (int i = 0; i < len; i++) x1[i] = x1[i] * x2[i];

    fft(x1, len, -1);

    for (int i = 0; i < len; i++) sum[i] = int(x1[i].x + 0.5);

    for (int i = 0; i < len; i++) {
      sum[i + 1] += sum[i] / 10;
      sum[i] %= 10;
    }

    len = len1 + len2 - 1;

    while (sum[len] == 0 && len > 0) len--;

    for (int i = len; i >= 0; i--) printf("%c", sum[i] + '0');
```

```
    printf("\n");
  }

  return 0;
}
```

参考文献

1. 桃酱的算法笔记[2].

参考资料与注释

[1] HDU 1402 - A * B Problem Plus

[2] 桃酱的算法笔记

9.13.4 快速数论变换

Authors: ChungZH, Yukimaikoriya, tigerruanyifan, isdanni, Saisyc, 383494, Tiphereth-A

简介

数论变换（number-theoretic transform, NTT）是离散傅里叶变换（DFT）在数论基础上的实现；**快速数论变换**（fast number-theoretic transform, FNTT）是 快速傅里叶变换（FFT）在数论基础上的实现。

数论变换是一种计算卷积（convolution）的快速算法。最常用算法就包括了前文提到的快速傅里叶变换。然而快速傅立叶变换具有一些实现上的缺点，举例来说，资料向量必须乘上复数系数的矩阵加以处理，而且每个复数系数的实部和虚部是一个正弦及余弦函数，因此大部分的系数都是浮点数，也就是说，必须做复数而且是浮点数的运算，因此计算量会比较大，而且浮点数运算产生的误差会比较大。

NTT 解决的是多项式乘法带模数的情况，可以说有些受模数的限制，数也比较大。目前最常见的模数是 998244353。

前置知识

学习数论变换需要前置知识：离散傅里叶变换、生成子群、原根、离散对数。相关知识可以在对应页面中学习，此处不再赘述。

定义

数论变换

在数学中，NTT 是关于任意 环 上的离散傅立叶变换（DFT）。在有限域的情况下，通常称为数论变换（NTT）。

数论变换（NTT）是通过将离散傅立叶变换化为 $F=\mathbb{Z}/p$，整数模质数 p。这是一个**有限域**，只要 n 可除 $p-1$，就存在本原 n 次方根，所以我们有 $p=\xi n+1$ 对于正整数 ξ。具体来说，对于质数 $p=qn+1,(n=2^m)$，原根 g 满足 $g^{qn}\equiv 1 \pmod p$，将 $g_n=g^q \pmod p$ 看做 ω_n 的等价，则其满足相似的性质，比如 $g_n^n\equiv 1 \pmod p, g_n^{n/2}\equiv -1 \pmod p$。

因为这里涉及到数论变化，所以 N（为了区分 FFT 中的 n，我们把这里的 n 称为 N）可以比 FFT 中的 n 大，但是只要把 $\frac{qN}{n}$ 看做这里的 q 就行了，能够避免大小问题。

常见的有：

$$p=167772161=5\times 2^{25}+1, g=3$$

$$p = 469762049 = 7 \times 2^{26} + 1, g = 3$$

$$p = 754974721 = 3^2 \times 5 \times 2^{24} + 1, g = 11$$

$$p = 998244353 = 7 \times 17 \times 2^{23} + 1, g = 3$$

$$p = 1004535809 = 479 \times 2^{21} + 1, g = 3$$

就是 g^{qn} 的等价 $\mathrm{e}^{2\pi n}$。

迭代到长度 l 时 $g_l = g^{\frac{p-1}{l}}$，或者 $\omega_n = g_l = g_N^{\frac{N}{l}} = g_N^{\frac{p-1}{l}}$。

快速数论变换

快速数论变换（FNTT）是数论变换（NTT）增加分治操作之后的快速算法。

快速数论变换使用的分治办法，与快速傅里叶变换使用的分治办法完全一致。这意味着，只需在快速傅里叶变换的代码基础上进行简单修改，即可得到快速数论变换的代码。

在算法竞赛中常提到的 NTT 一词，往往实际指的是快速数论变换，一般默认「数论变换」是指「快速数论变换」。

这样简写的逻辑与快速傅里叶变换相似。事实上，「快速傅里叶变换」（FFT）一词指的是「快速离散傅里叶变换」（FDFT），但由于「快速」只能作用于离散，甚至是本原单位根阶数为 2 的幂的特殊情形，不能作用于连续，因此「离散」一词被省略掉，FDFT 变为 FFT，即 FFT 永远指的是特殊的离散情形。

数论变换或快速数论变换是在取模意义下进行的操作，不存在连续的情形，永远是离散的，自然也无需提到离散一词。

在算法领域，不进行提速的操作是无意义的。在快速傅里叶变换中介绍 DFT 一词，是因为 DFT 在信号处理、图像处理领域也有其他的具体应用，同时 DFT 也是 FFT 的原理或前置知识。

在不引起混淆的情形下，常用 NTT 来代指 FNTT。为了不引起下文进一步介绍的混淆，下文的 NTT 与 FNTT 两个词进行了分离。

DFT、FFT、NTT、FNTT 的具体关系是：

- 在 DFT 与 NTT 的基础上，增加分治操作，得到 FFT 与 FNTT。分治操作的办法与原理，可以参见快速傅里叶变换一文。
- 在 DFT 与 FFT 的基础上，将复数加法与复数乘法替换为模 p 意义下的加法和乘法，一般大小限制在 0 到 $p-1$ 之间；将本原单位根改为模 p 意义下的相同阶数的本原单位根，阶数为 2 的幂，即可得到 NTT 与 FNTT。

由于替换的运算只涉及加法和乘法，因此 DFT、FFT、NTT、FNTT 拥有相同的原理，均在满足加法与乘法的环上进行，无需域上满足除法运算的更加严格的条件。

事实上，只要拥有原根，即群论中的生成元，该模数下的 NTT 或 FNTT 即可进行。考虑到模数为 1、2 和 4 的情形太小，不具有实际意义，对于奇素数 p 和正整数 α，只要给出模数为 p^α 和 $2p^\alpha$ 的原根 g，采用同样的办法，则 NTT 或 FNTT 仍然可以进行。

模板

下面是一个大数相乘的模板，参考来源[1]。

" 参考代码"

```
#include <algorithm>
#include <bitset>
#include <cmath>
```

```cpp
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <ctime>
#include <iomanip>
#include <iostream>
#include <map>
#include <queue>
#include <set>
#include <string>
#include <vector>
using namespace std;

int read() {
  int x = 0, f = 1;
  char ch = getchar();
  while (ch < '0' || ch > '9') {
    if (ch == '-') f = -1;
    ch = getchar();
  }
  while (ch <= '9' && ch >= '0') {
    x = 10 * x + ch - '0';
    ch = getchar();
  }
  return x * f;
}

void print(int x) {
  if (x < 0) putchar('-'), x = -x;
  if (x >= 10) print(x / 10);
  putchar(x % 10 + '0');
}

const int N = 300100, P = 998244353;

int qpow(int x, int y) {
  int res(1);
  while (y) {
    if (y & 1) res = 1ll * res * x % P;
    x = 1ll * x * x % P;
    y >>= 1;
  }
  return res;
}

int r[N];

void ntt(int *x, int lim, int opt) {
  int i, j, k, m, gn, g, tmp;
  for (i = 0; i < lim; ++i)
    if (r[i] < i) swap(x[i], x[r[i]]);
  for (m = 2; m <= lim; m <<= 1) {
    k = m >> 1;
    gn = qpow(3, (P - 1) / m);
```

```
    for (i = 0; i < lim; i += m) {
      g = 1;
      for (j = 0; j < k; ++j, g = 1ll * g * gn % P) {
        tmp = 1ll * x[i + j + k] * g % P;
        x[i + j + k] = (x[i + j] - tmp + P) % P;
        x[i + j] = (x[i + j] + tmp) % P;
      }
    }
  }
  if (opt == -1) {
    reverse(x + 1, x + lim);
    int inv = qpow(lim, P - 2);
    for (i = 0; i < lim; ++i) x[i] = 1ll * x[i] * inv % P;
  }
}

int A[N], B[N], C[N];

char a[N], b[N];

int main() {
  int i, lim(1), n;
  scanf("%s", &a);
  n = strlen(a);
  for (i = 0; i < n; ++i) A[i] = a[n - i - 1] - '0';
  while (lim < (n << 1)) lim <<= 1;
  scanf("%s", &b);
  n = strlen(b);
  for (i = 0; i < n; ++i) B[i] = b[n - i - 1] - '0';
  while (lim < (n << 1)) lim <<= 1;
  for (i = 0; i < lim; ++i) r[i] = (i & 1) * (lim >> 1) + (r[i >> 1] >> 1);
  ntt(A, lim, 1);
  ntt(B, lim, 1);
  for (i = 0; i < lim; ++i) C[i] = 1ll * A[i] * B[i] % P;
  ntt(C, lim, -1);
  int len(0);
  for (i = 0; i < lim; ++i) {
    if (C[i] >= 10) len = i + 1, C[i + 1] += C[i] / 10, C[i] %= 10;
    if (C[i]) len = max(len, i);
  }
  while (C[len] >= 10) C[len + 1] += C[len] / 10, C[len] %= 10, len++;
  for (i = len; ~i; --i) putchar(C[i] + '0');
  puts("");
  return 0;
}
```

参考资料与拓展阅读

1. FWT（快速沃尔什变换）零基础详解 qaq（ACM/OI）[2]
2. FFT（快速傅里叶变换）0 基础详解！附 NTT（ACM/OI）[3]
3. Number-theoretic transform(NTT) - Wikipedia[4]
4. Tutorial on FFT/NTT—The tough made simple. (Part 1)[5]

参考资料与注释

[1] 参考来源

[2] FWT（快速沃尔什变换）零基础详解 qaq（ACM/OI）

[3] FFT（快速傅里叶变换）0 基础详解！附 NTT（ACM/OI）

[4] Number-theoretic transform(NTT) - Wikipedia

[5] Tutorial on FFT/NTT—The tough made simple. (Part 1)

9.13.5 快速沃尔什变换

Authors: Xeonacid, nocriz

（本文转载自 桃酱的算法笔记[1]，原文戳 链接[2]，已获得作者授权）

简介

> 沃尔什转换（Walsh Transform）是在频谱分析上作为离散傅立叶变换的替代方案的一种方法。——维基百科[3]

其实这个变换在信号处理中应用很广泛，fft 是 double 类型的，但是 walsh 把信号在不同震荡频率方波下拆解，因此所有的系数都是绝对值大小相同的整数，这使得不需要作浮点数的乘法运算，提高了运算速度。

所以，FWT 和 FFT 的核心思想应该是相同的，都是对数组的变换。我们记对数组 A 进行快速沃尔什变换后得到的结果为 $FWT[A]$。

那么 FWT 核心思想就是：

我们需要一个新序列 C，由序列 A 和序列 B 经过某运算规则得到，即 $C = A \cdot B$；

我们先正向得到 $FWT[A], FWT[B]$，再根据 $FWT[C] = FWT[A] \cdot FWT[B]$ 在 $O(n)$ 的时间复杂度内求出 $FWT[C]$；

然后逆向运算得到原序列 C。时间复杂度为 $O(n \log n)$。

在算法竞赛中，FWT 是用于解决对下标进行位运算卷积问题的方法。

公式：$C_i = \sum_{i=j \bigoplus k} A_j B_k$

（其中 $\bigoplus$ 是二元位运算中的某一种，$*$ 是普通乘法）

FWT 的运算

FWT 之与（&）运算和或（|）运算

与运算和或运算的本质是差不多的，所以这里讲一下或运算，与运算也是可以自己根据公式 yy 出来的。

或运算

如果有 $k = i|j$，那么 i 的二进制位为 1 的位置和 j 的二进制位为 1 的位置肯定是 k 的二进制位为 1 的位置的子集。

现在要得到 $FWT[C] = FWT[A] * FWT[B]$，我们就要构造这个 fwt 的规则。

我们按照定义，显然可以构造 $FWT[A] = A' = \sum_{i=i|j} A_j$，来表示 j 满足二进制中 1 为 i 的子集。

那么显然会有 $C_i = \sum_{i=j|k} A_j * B_k \implies FWT[C] = FWT[A] * FWT[B]$

那么我们接下来看 $FWT[A]$ 怎么求。

首先肯定不能枚举了，复杂度为 $O(n^2)$。既然不能整体枚举，我们就考虑分治。

我们把整个区间二分，其实二分区间之后，下标写成二进制形式是有规律可循的。

我们令 A_0 表示 A 的前一半，A_1 表示区间的后一半，那么 A_0 就是 A 下标最大值的最高位为 0，他的子集就是他本身的子集（因为最高位为 0 了），但是 A_1 的最高位是 1，他满足条件的子集不仅仅是他本身，还包最高位为 0 的子集，即

$$FWT[A] = merge(FWT[A_0], FWT[A_0] + FWT[A_1])$$

其中 merge 表示像字符串拼接一样把它们拼起来，+ 就是普通加法，表示对应二进制位相加。

这样我们就通过二分能在 $O(\log n)$ 的时间复杂度内完成拼接，每次拼接的时候要完成一次运算，也就是说在 $O(n \log n)$ 的时间复杂度得到了 $FWT[A]$。

接下来就是反演了，其实反演是很简单的，既然知道了 A_0 的本身的子集是他自己 ($A_0 = FWT[A_0]$)，A_1 的子集是 $FWT[A_0] + FWT[A_1]$ $A_1 = A'_0 + A'_1$)，那就很简单的得出反演的递推式了：

$$UFWT[A'] = merge(UFWT[A'_0], UFWT[A'_1] - UFWT[A'_0])$$

与运算

与运算类比或运算可以得到类似结论

$$FWT[A] = merge(FWT[A_0] + FWT[A_1], FWT[A_1])$$

$$UFWT[A'] = merge(UFWT[A'_0] - UFWT[A'_1], UFWT[A'_1])$$

异或运算

最常考的异或运算。

异或的卷积是基于如下原理：

若我们令 $i \ \& \ j$ 中 1 数量的奇偶性为 i 与 j 的奇偶性，那么 i 与 k 的奇偶性异或 j 与 k 的奇偶性等于 $i \operatorname{xor} j$ 与 k 的奇偶性。

对于 $FWT[A]$ 的运算其实也很好得到。

公式如下：

$A_i = \sum_{C_1} A_j - \sum_{C_2} A_j$($C_1$ 表示 $i \ \& \ j$ 奇偶性为 0，C_2 表示 $i \ \& \ j$ 的奇偶性为 1)

结论：

$$FWT[A] = merge(FWT[A_0] + FWT[A_1], FWT[A_0] - FWT[A_1])$$

$$UFWT[A'] = merge(\frac{UFWT[A'_0] + UFWT[A'_1]}{2}, \frac{UFWT[A'_0] - UFWT[A'_1]}{2})$$

同或运算

类比异或运算给出公式：

$A_i = \sum_{C_1} A_j - \sum_{C_2} A_j$($C_1$ 表示 $i|j$ 奇偶性为 0，C_2 表示 $i|j$ 的奇偶性为 1)

$$FWT[A] = merge(FWT[A_1] - FWT[A_0], FWT[A_1] + FWT[A_0])$$

$$UFWT[A'] = merge(\frac{UFWT[A'_1] - UFWT[A'_0]}{2}, \frac{UFWT[A'_1] + UFWT[A'_0]}{2})$$

例题

"【CF103329F】【XXII Opencup, Grand Prix of XiAn】The Struggle[4]"

给出一个椭圆 E，其中所有整点的坐标均在 $[1, 4\cdot 10^6]$ 之间。求 $\sum_{(x,y)\in E}(x\oplus y)^{33}x^{-2}y^{-1} \mod 10^9+7$ 的值。

" 题解"

这是一道比较不裸的题，出题人提供了详细的英文题解，具体请见 此链接[5]。

参考资料与注释

[1] 桃酱的算法笔记

[2] 链接

[3] 维基百科

[4] 【CF103329F】【XXII Opencup, Grand Prix of XiAn】The Struggle

[5] 此链接

9.13.6 Chirp Z 变换

Chirp Z 变换也被称为 Bluestein 算法。与离散傅里叶变换类似，Chirp Z 变换是给出多项式 $A(x)=\sum_{i=0}^{n}a_ix^i\in\mathbb{C}[x]$ 和 $c\in\mathbb{C}\ \{0\}$ 求出 $A(1),A(c),A(c^2),\ldots$ 的一种算法，不要求 c 为单位根。也可用于数论变换。

方法一

令幂级数 $A_0(x)=\sum_{i\geq 0}a_ic^{i^2}x^i$ 且对于 $\forall j>n$ 令 $a_j=0$、$B_0(x)=\sum_{i\geq 0}c^{-(i-n)^2}x^i$，对于 $t\geq 0$ 有

$$\begin{aligned}
[x^{n+t}](A_0(x)B_0(x)) &= \sum_{i=0}^{n+t}\left([x^i]A_0(x)\right)\left([x^{n+t-i}]B_0(x)\right)\\
&= \sum_{i=0}^{n+t}a_ic^{i^2-(i-t)^2}\\
&= c^{-t^2}\sum_{i=0}^{n+t}a_ic^{2it}\\
&= c^{-t^2}A(c^{2t})
\end{aligned}$$

通过计算 $c^{t^2}[x^{n+t}](A_0(x)B_0(x))$ 可得到 $A(1),A(c^2),\ldots$。而对于 $A(c),A(c^3),\ldots$ 可构造 $A(cx)$ 后同理，该算法需两次卷积。因为我们从 x^n 开始提取系数，所以可以利用循环卷积。

方法二

对于非负整数 k 和 i 考虑

$$ki=\binom{i+k}{2}-\binom{i}{2}-\binom{k}{2}$$

其中 $\binom{a}{b}=\frac{a(a-1)\cdots(a-b+1)}{b!}$ 为二项式系数，那么

$$A(c^k)=c^{-\binom{k}{2}}\sum_{i=0}^{n}a_ic^{\binom{i+k}{2}-\binom{i}{2}}$$

令 $A_0(x)=\sum_i a_{n-i}c^{-\binom{n-i}{2}}x^i$ 且对于 $\forall j>n$ 和 $\forall j<0$ 令 $a_j=0$、$B_0(x)=\sum_{i\geq 0}c^{\binom{i}{2}}x^i$ 那么对于 $t\geq 0$ 有

$$\begin{aligned}[x^{n+t}](A_0(x)B_0(x)) &= \sum_{i=0}^{n+t}\left([x^{n+t-i}]A_0(x)\right)\left([x^i]B_0(x)\right)\\ &= \sum_{i=0}^{n+t} a_{i-t}c^{\binom{i}{2}-\binom{i-t}{2}}\\ &= \sum_{i=-t}^{n} a_i c^{\binom{i+t}{2}-\binom{i}{2}}\\ &= c^{\binom{t}{2}}\cdot A(c^t)\end{aligned}$$

通过计算 $c^{-\binom{t}{2}}[x^{n+t}](A_0(x)B_0(x))$ 可得到 $A(1),A(c),\dots$，该算法需一次卷积。且 $\forall i\geq 0$ 有 $c^{\binom{i+1}{2}}=c^{\binom{i}{2}}\cdot c^i$，可递推计算。

9.13.7 多项式牛顿迭代

描述

给定多项式 $g(x)$，已知有 $f(x)$ 满足：

$$g(f(x))\equiv 0 \pmod{x^n}$$

求出模 x^n 意义下的 $f(x)$。

Newton's Method

考虑倍增。

首先当 $n=1$ 时，$[x^0]\,g(f(x))=0$ 的解需要单独求出。

假设现在已经得到了模 $x^{\lceil\frac{n}{2}\rceil}$ 意义下的解 $f_0(x)$，要求模 x^n 意义下的解 $f(x)$。

将 $g(f(x))$ 在 $f_0(x)$ 处进行泰勒展开，有：

$$\sum_{i=0}^{+\infty}\frac{g^{(i)}(f_0(x))}{i!}(f(x)-f_0(x))^i\equiv 0 \pmod{x^n}$$

因为 $f(x)-f_0(x)$ 的最低非零项次数最低为 $\lceil\frac{n}{2}\rceil$，故有：

$$\forall 2\leqslant i: (f(x)-f_0(x))^i\equiv 0 \pmod{x^n}$$

则：

$$\sum_{i=0}^{+\infty}\frac{g^{(i)}(f_0(x))}{i!}(f(x)-f_0(x))^i\equiv g(f_0(x))+g'(f_0(x))(f(x)-f_0(x))\equiv 0 \pmod{x^n}$$

$$f(x)\equiv f_0(x)-\frac{g(f_0(x))}{g'(f_0(x))} \pmod{x^n}$$

例题

多项式求逆

设给定函数为 $h(x)$，有方程：

$$g(f(x))=\frac{1}{f(x)}-h(x)\equiv 0 \pmod{x^n}$$

应用 Newton's Method 可得：

$$\begin{aligned} f(x) &\equiv f_0(x) - \frac{\frac{1}{f_0(x)} - h(x)}{-\frac{1}{f_0^2(x)}} && \pmod{x^n} \\ &\equiv 2f_0(x) - f_0^2(x)\,h(x) && \pmod{x^n} \end{aligned}$$

时间复杂度

$$T(n) = T\left(\frac{n}{2}\right) + O(n\log n) = O(n\log n)$$

多项式开方

设给定函数为 $h(x)$，有方程：

$$g(f(x)) = f^2(x) - h(x) \equiv 0 \pmod{x^n}$$

应用 Newton's Method 可得：

$$\begin{aligned} f(x) &\equiv f_0(x) - \frac{f_0^2(x) - h(x)}{2f_0(x)} && \pmod{x^n} \\ &\equiv \frac{f_0^2(x) + h(x)}{2f_0(x)} && \pmod{x^n} \end{aligned}$$

时间复杂度

$$T(n) = T\left(\frac{n}{2}\right) + O(n\log n) = O(n\log n)$$

多项式 exp

设给定函数为 $h(x)$，有方程：

$$g(f(x)) = \ln f(x) - h(x) \pmod{x^n}$$

应用 Newton's Method 可得：

$$\begin{aligned} f(x) &\equiv f_0(x) - \frac{\ln f_0(x) - h(x)}{\frac{1}{f_0(x)}} && \pmod{x^n} \\ &\equiv f_0(x)\left(1 - \ln f_0(x) + h(x)\right) && \pmod{x^n} \end{aligned}$$

时间复杂度

$$T(n) = T\left(\frac{n}{2}\right) + O(n\log n) = O(n\log n)$$

9.13.8 多项式多点求值 | 快速插值

多项式的多点求值

描述

给出一个多项式 $f(x)$ 和 n 个点 $x_1, x_2, \ldots, x_n$，求

$$f(x_1), f(x_2), \ldots, f(x_n)$$

解法

考虑使用分治来将问题规模减半。

将给定的点分为两部分：

$$\begin{aligned} X_0 &= \left\{x_1, x_2, \ldots, x_{\lfloor \frac{n}{2} \rfloor}\right\} \\ X_1 &= \left\{x_{\lfloor \frac{n}{2} \rfloor + 1}, x_{\lfloor \frac{n}{2} \rfloor + 2}, \ldots, x_n\right\} \end{aligned}$$

构造多项式

$$g_0(x) = \prod_{x_i \in X_0} (x - x_i)$$

则有 $\forall x \in X_0 : g_0(x) = 0$。

考虑将 $f(x)$ 表示为 $g_0(x)Q(x) + f_0(x)$ 的形式，即：

$$f_0(x) \equiv f(x) \pmod{g_0(x)}$$

则有 $\forall x \in X_0 : f(x) = g_0(x)Q(x) + f_0(x) = f_0(x)$，$X_1$ 同理。

至此，问题的规模被减半，可以使用分治 + 多项式取模解决。

时间复杂度

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n\log n) = O\left(n\log^2 n\right)$$

多项式的快速插值

描述

给出一个 $n+1$ 个点的集合

$$X = \{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\}$$

求一个 n 次多项式 $f(x)$ 使得其满足 $\forall (x, y) \in X : f(x) = y$。

解法

考虑拉格朗日插值公式

$$f(x) = \sum_{i=1}^{n} \prod_{j \neq i} \frac{x - x_j}{x_i - x_j} y_i$$

记多项式 $M(x) = \prod_{i=1}^{n}(x - x_i)$，由洛必达法则可知

$$\prod_{j \neq i}(x_i - x_j) = \lim_{x \to x_i} \frac{\prod_{j=1}^{n}(x - x_j)}{x - x_i} = M'(x_i)$$

因此多项式被表示为

$$f(x) = \sum_{i=1}^{n} \frac{y_i}{M'(x_i)} \prod_{j \neq i}(x - x_j)$$

我们首先通过分治计算出 $M(x)$ 的系数表示，接着可以通过多点求值在 $O(n\log^2 n)$ 时间内计算出所有的 $M'(x_i)$。我们令 $v_i = \frac{y_i}{M'(x_i)}$，接下来考虑计算出 $f(x)$。对于 $n = 1$ 的情况，有 $f(x) = v_1, M(x) = x - x_1$。否则令

$$\begin{aligned}
f_0(x) &= \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} v_i \prod_{j \neq i \wedge j \leq \lfloor \frac{n}{2} \rfloor} (x - x_j) \\
M_0(x) &= \prod_{i=1}^{\lfloor \frac{n}{2} \rfloor}(x - x_i) \\
f_1(x) &= \sum_{i=\lfloor \frac{n}{2} \rfloor + 1}^{n} v_i \prod_{j \neq i \wedge \lfloor \frac{n}{2} \rfloor < j \leq n} (x - x_j) \\
M_1(x) &= \prod_{i=\lfloor \frac{n}{2} \rfloor + 1}^{n} (x - x_i)
\end{aligned}$$

可得 $f(x) = f_0(x)M_1(x) + f_1(x)M_0(x), M(x) = M_0(x)M_1(x)$，因此可以分治计算，这一部分的复杂度同样是 $O(n\log^2 n)$。

9.13.9 多项式初等函数

Authors: 97littleleaf11, abc1763613206, CCXXXI, EndlessCheng, Enter-tainer, fps5283, Great-designer, H-J-Granger, hly1204, hsfzLZH1, huayucaiji, Ir1d, kenlig, Marcythm, ouuan, SamZhangQingChuan, shuzhouliu, sshwy, StudyingFather, test12345-pupil, Tiphereth-A, TrisolarisHD, untitledunrevised

本页面包含多项式常见的初等函数操作。具体而言，本页面包含如下内容：

1. 多项式求逆
2. 多项式开方
3. 多项式除法
4. 多项式取模
5. 多项式指数函数
6. 多项式对数函数
7. 多项式三角函数
8. 多项式反三角函数

" 初等函数与非初等函数"

初等函数的定义如下[1]：

若域 F 中存在映射 $u \to \partial u$ 满足：

1. $\partial(u+v) = \partial u + \partial v$
2. $\partial(uv) = u\partial v + v\partial u$

则称这个域为**微分域**。

若微分域 F 上的函数 u 满足以下的任意一条条件，则称该函数 u 为初等函数：

1. u 是 F 上的代数函数。
2. u 是 F 上的指数性函数，即存在 $a \in F$ 使得 $\partial u = u\partial a$.
3. u 是 F 上的对数性函数，即存在 $a \in F$ 使得 $\partial u = \frac{\partial a}{a}$.

以下是常见的初等函数：

1. 代数函数：存在有限次多项式 P 使得 $P(f(x)) = 0$ 的函数 $f(x)$，如 $2x+1$,$\sqrt{x}$,$(1+x^2)^{-1}$,$|x|$.
2. 指数函数
3. 对数函数
4. 三角函数
5. 反三角函数
6. 双曲函数
7. 反双曲函数
8. 以上函数的复合，如：

$$\frac{\mathrm{e}^{\tan x}}{1+x^2}\sin\left(\sqrt{1+\ln^2 x}\right)$$

$$-\mathrm{i}\ln\left(x+\mathrm{i}\sqrt{1-x^2}\right)$$

以下是常见的非初等函数：

1. 误差函数：

$$\operatorname{erf}(x) := \frac{2}{\sqrt{\pi}}\int_0^x \exp\left(-t^2\right)\mathrm{d}t$$

多项式求逆

给定多项式 $f(x)$，求 $f^{-1}(x)$。

解法

倍增法

首先，易知

$$[x^0]f^{-1}(x) = \left([x^0]f(x)\right)^{-1}$$

假设现在已经求出了 $f(x)$ 在模 $x^{\lceil \frac{n}{2} \rceil}$ 意义下的逆元 $f_0^{-1}(x)$。有：

$$\begin{aligned} f(x)f_0^{-1}(x) &\equiv 1 \pmod{x^{\lceil \frac{n}{2} \rceil}} \\ f(x)f^{-1}(x) &\equiv 1 \pmod{x^{\lceil \frac{n}{2} \rceil}} \\ f^{-1}(x) - f_0^{-1}(x) &\equiv 0 \pmod{x^{\lceil \frac{n}{2} \rceil}} \end{aligned}$$

两边平方可得：

$$f^{-2}(x) - 2f^{-1}(x)f_0^{-1}(x) + f_0^{-2}(x) \equiv 0 \pmod{x^n}$$

两边同乘 $f(x)$ 并移项可得：

$$f^{-1}(x) \equiv f_0^{-1}(x)\left(2 - f(x)f_0^{-1}(x)\right) \pmod{x^n}$$

递归计算即可。

时间复杂度

$$T(n) = T\left(\frac{n}{2}\right) + O(n\log n) = O(n\log n)$$

Newton's Method

参见 Newton's Method.

Graeffe 法

欲求 $f^{-1}(x) \bmod x^{2n}$ 考虑

$$\begin{aligned} f^{-1}(x) \bmod x^{2n} &= f(-x)(f(x)f(-x))^{-1} \bmod x^{2n} \\ &= f(-x)g^{-1}(x^2) \bmod x^{2n} \end{aligned}$$

只需求出 $g^{-1}(x) \bmod x^n$ 即可还原出 $g^{-1}(x^2) \bmod x^{2n}$ 因为 $f(x)f(-x)$ 是偶函数，时间复杂度同上。

代码

" 多项式求逆"

```cpp
constexpr int maxn = 262144;
constexpr int mod = 998244353;

using i64 = long long;
using poly_t = int[maxn];
using poly = int *const;

void polyinv(const poly &h, const int n, poly &f) {
  /* f = 1 / h = f_0 (2 - f_0 h) */
  static poly_t inv_t;
  std::fill(f, f + n + n, 0);
  f[0] = fpow(h[0], mod - 2);
  for (int t = 2; t <= n; t <<= 1) {
    const int t2 = t << 1;
```

```
    std::copy(h, h + t, inv_t);
    std::fill(inv_t + t, inv_t + t2, 0);

    DFT(f, t2);
    DFT(inv_t, t2);
    for (int i = 0; i != t2; ++i)
      f[i] = (i64)f[i] * sub(2, (i64)f[i] * inv_t[i] % mod) % mod;
    IDFT(f, t2);

    std::fill(f + t, f + t2, 0);
  }
}
```

例题

1. 有标号简单无向连通图计数：「POJ 1737」Connected Graph[2]

多项式开方

给定多项式 $g(x)$，求 $f(x)$，满足：

$$f^2(x) \equiv g(x) \pmod{x^n}$$

解法

倍增法

首先讨论 $[x^0]\, g(x)$ 不为 0 的情况。

易知：

$$[x^0]\, f(x) = \sqrt{[x^0]\, g(x)}$$

若 $[x^0]\, g(x)$ 没有平方根，则多项式 $g(x)$ 没有平方根。

> $[x^0]\, g(x)$ 可能有多个平方根，选取不同的根会求出不同的 $f(x)$。

假设现在已经求出了 $g(x)$ 在模 $x^{\lceil \frac{n}{2} \rceil}$ 意义下的平方根 $f_0(x)$，则有：

$$\begin{aligned}
f_0^2(x) &\equiv g(x) &\pmod{x^{\lceil \frac{n}{2} \rceil}} \\
f_0^2(x) - g(x) &\equiv 0 &\pmod{x^{\lceil \frac{n}{2} \rceil}} \\
\left(f_0^2(x) - g(x)\right)^2 &\equiv 0 &\pmod{x^n} \\
\left(f_0^2(x) + g(x)\right)^2 &\equiv 4f_0^2(x)\, g(x) &\pmod{x^n} \\
\left(\frac{f_0^2(x) + g(x)}{2f_0(x)}\right)^2 &\equiv g(x) &\pmod{x^n} \\
\frac{f_0^2(x) + g(x)}{2f_0(x)} &\equiv f(x) &\pmod{x^n} \\
2^{-1} f_0(x) + 2^{-1} f_0^{-1}(x)\, g(x) &\equiv f(x) &\pmod{x^n}
\end{aligned}$$

倍增计算即可。

时间复杂度

$$T(n) = T\left(\frac{n}{2}\right) + O(n \log n) = O(n \log n)$$

还有一种常数较小的写法就是在倍增维护 $f(x)$ 的时候同时维护 $f^{-1}(x)$ 而不是每次都求逆。

> 当 $[x^0]\, g(x) \neq 1$ 时，可能需要使用二次剩余来计算 $[x^0]\, f(x)$。

上述方法需要知道 $f_0(x)$ 的逆，所以常数项不能为 0。

若 $[x^0]\,g(x)=0$，则将 $g(x)$ 分解成 $x^k h(x)$，其中 $[x^0]\,h(x)\neq 0$。

- 若 k 是奇数，则 $g(x)$ 没有平方根。
- 若 k 是偶数，则求出 $h(x)$ 的平方根 $\sqrt{h(x)}$，然后得到 $f(x)\equiv x^{k/2}\sqrt{h(x)} \pmod{x^n}$。

" 洛谷模板题 P5205【模板】多项式开根[3] 参考代码"

```
#include <bits/stdc++.h>
using namespace std;

const int maxn = 1 << 20, mod = 998244353;

int a[maxn], b[maxn], g[maxn], gg[maxn];

int qpow(int x, int y) {  // 快速幂
  int ans = 1;

  while (y) {
    if (y & 1) {
      ans = (long long)1 * ans * x % mod;
    }
    x = (long long)1 * x * x % mod;
    y >>= 1;
  }
  return ans;
}

int inv2 = qpow(2, mod - 2);  // 逆元

void change(int *f, int len) {
  for (int i = 1, j = len / 2; i < len - 1; i++) {
    if (i < j) {
      swap(f[i], f[j]);
    }

    int k = len / 2;
    while (j >= k) {
      j -= k;
      k /= 2;
    }
    if (j < k) {
      j += k;
    }
  }
}

void NTT(int *f, int len, int type) {  // NTT
  change(f, len);

  for (int q = 2; q <= len; q <<= 1) {
    int nxt = qpow(3, (mod - 1) / q);
    for (int i = 0; i < len; i += q) {
```

```
      int w = 1;

      for (int k = i; k < i + (q >> 1); k++) {
        int x = f[k];
        int y = (long long)1 * w * f[k + (q / 2)] % mod;

        f[k] = (x + y) % mod;
        f[k + (q / 2)] = (x - y + mod) % mod;
        w = (long long)1 * w * nxt % mod;
      }
    }
  }

  if (type == -1) {
    reverse(f + 1, f + len);
    int iv = qpow(len, mod - 2);

    for (int i = 0; i < len; i++) {
      f[i] = (long long)1 * f[i] * iv % mod;
    }
  }
}

void inv(int deg, int *f, int *h) {  // 求逆元
  if (deg == 1) {
    h[0] = qpow(f[0], mod - 2);
    return;
  }

  inv(deg + 1 >> 1, f, h);

  int len = 1;
  while (len < deg * 2) {  // 倍增
    len *= 2;
  }

  copy(f, f + deg, gg);
  fill(gg + deg, gg + len, 0);

  NTT(gg, len, 1);
  NTT(h, len, 1);
  for (int i = 0; i < len; i++) {
    h[i] = (long long)1 * (2 - (long long)1 * gg[i] * h[i] % mod + mod) % mod *
           h[i] % mod;
  }

  NTT(h, len, -1);
  fill(h + deg, h + len, 0);
}

int n, t[maxn];

// deg: 次数
// f: 被开根数组
```

```
// h: 答案数组
void sqrt(int deg, int *f, int *h) {
  if (deg == 1) {
    h[0] = 1;
    return;
  }

  sqrt(deg + 1 >> 1, f, h);

  int len = 1;
  while (len < deg * 2) {  // 倍增
    len *= 2;
  }
  fill(g, g + len, 0);
  inv(deg, h, g);
  copy(f, f + deg, t);
  fill(t + deg, t + len, 0);
  NTT(t, len, 1);
  NTT(g, len, 1);
  NTT(h, len, 1);

  for (int i = 0; i < len; i++) {
    h[i] = (long long)1 * inv2 *
           ((long long)1 * h[i] % mod + (long long)1 * g[i] * t[i] % mod) % mod;
  }
  NTT(h, len, -1);
  fill(h + deg, h + len, 0);
}

int main() {
  cin >> n;

  for (int i = 0; i < n; i++) {
    scanf("%d", &a[i]);
  }
  sqrt(n, a, b);

  for (int i = 0; i < n; i++) {
    printf("%d ", b[i]);
  }

  return 0;
}
```

Newton's Method

参见 Newton's Method.

例题

1. 「Codeforces Round #250」E. The Child and Binary Tree[4]

多项式除法 & 取模

给定多项式 $f(x), g(x)$，求 $g(x)$ 除 $f(x)$ 的商 $Q(x)$ 和余数 $R(x)$。

解法

发现若能消除 $R(x)$ 的影响则可直接多项式求逆解决。

考虑构造变换

$$f^R(x) = x^{\deg f} f\left(\frac{1}{x}\right)$$

观察可知其实质为反转 $f(x)$ 的系数。

设 $n = \deg f, m = \deg g$。

将 $f(x) = Q(x)g(x) + R(x)$ 中的 x 替换成 $\frac{1}{x}$ 并将其两边都乘上 x^n，得到：

$$x^n f\left(\frac{1}{x}\right) = x^{n-m}Q\left(\frac{1}{x}\right)x^m g\left(\frac{1}{x}\right) + x^{n-m+1}x^{m-1}R\left(\frac{1}{x}\right)$$
$$f^R(x) = Q^R(x)g^R(x) + x^{n-m+1}R^R(x)$$

注意到上式中 $R^R(x)$ 的系数为 x^{n-m+1}，则将其放到模 x^{n-m+1} 意义下即可消除 $R^R(x)$ 带来的影响。

又因 $Q^R(x)$ 的次数为 $(n-m) < (n-m+1)$，故 $Q^R(x)$ 不会受到影响。

则：

$$f^R(x) \equiv Q^R(x)g^R(x) \pmod{x^{n-m+1}}$$

使用多项式求逆即可求出 $Q(x)$，将其反代即可得到 $R(x)$。

时间复杂度 $O(n\log n)$。

多项式对数函数 & 指数函数

给定多项式 $f(x)$，求模 x^n 意义下的 $\ln f(x)$ 与 $\exp f(x)$。

解法

普通方法

首先，对于多项式 $f(x)$，若 $\ln f(x)$ 存在，则由其 定义，其必须满足：

$$[x^0]f(x) = 1$$

对 $\ln f(x)$ 求导再积分，可得：

$$\frac{\mathrm{d}\ln f(x)}{\mathrm{d}x} \equiv \frac{f'(x)}{f(x)} \pmod{x^n}$$
$$\ln f(x) \equiv \int \mathrm{d}\ln x \equiv \int \frac{f'(x)}{f(x)}\mathrm{d}x \pmod{x^n}$$

多项式的求导，积分时间复杂度为 $O(n)$，求逆时间复杂度为 $O(n\log n)$，故多项式求 $\ln$ 时间复杂度 $O(n\log n)$。

首先，对于多项式 $f(x)$，若 $\exp f(x)$ 存在，则其必须满足：

$$[x^0]f(x) = 0$$

否则 $\exp f(x)$ 的常数项不收敛。

对 $\exp f(x)$ 求导，可得：

$$\frac{\mathrm{d}\exp f(x)}{\mathrm{d}x} \equiv \exp f(x)f'(x) \pmod{x^n}$$

比较两边系数可得：

$$[x^{n-1}]\frac{\mathrm{d}\exp f(x)}{\mathrm{d}x} = \sum_{i=0}^{n-1}\left([x^i]\exp f(x)\right)\left([x^{n-i-1}]f'(x)\right)$$
$$n[x^n]\exp f(x) = \sum_{i=0}^{n-1}\left([x^i]\exp f(x)\right)\left((n-i)[x^{n-i}]f(x)\right)$$

使用分治 FFT 即可解决。

时间复杂度 $O(n\log^2 n)$。

Newton's Method

使用 Newton's Method 即可在 $O(n\log n)$ 的时间复杂度内解决多项式 exp。

代码

" 多项式 ln/exp"

```
constexpr int maxn = 262144;
constexpr int mod = 998244353;

using i64 = long long;
using poly_t = int[maxn];
using poly = int *const;

void derivative(const poly &h, const int n, poly &f) {
  for (int i = 1; i != n; ++i) f[i - 1] = (i64)h[i] * i % mod;
  f[n - 1] = 0;
}

void integrate(const poly &h, const int n, poly &f) {
  for (int i = n - 1; i; --i) f[i] = (i64)h[i - 1] * inv[i] % mod;
  f[0] = 0; /* C */
}

void polyln(const poly &h, const int n, poly &f) {
  /* f = ln h = ∫ h' / h dx */
  assert(h[0] == 1);
  static poly_t ln_t;
  const int t = n << 1;

  derivative(h, n, ln_t);
  std::fill(ln_t + n, ln_t + t, 0);
  polyinv(h, n, f);

  DFT(ln_t, t);
  DFT(f, t);
  for (int i = 0; i != t; ++i) ln_t[i] = (i64)ln_t[i] * f[i] % mod;
  IDFT(ln_t, t);

  integrate(ln_t, n, f);
}

void polyexp(const poly &h, const int n, poly &f) {
  /* f = exp(h) = f_0 (1 - ln f_0 + h) */
  assert(h[0] == 0);
  static poly_t exp_t;
  std::fill(f, f + n + n, 0);
  f[0] = 1;
  for (int t = 2; t <= n; t <<= 1) {
    const int t2 = t << 1;

    polyln(f, t, exp_t);
    exp_t[0] = sub(pls(h[0], 1), exp_t[0]);
    for (int i = 1; i != t; ++i) exp_t[i] = sub(h[i], exp_t[i]);
```

```
    std::fill(exp_t + t, exp_t + t2, 0);

    DFT(f, t2);
    DFT(exp_t, t2);
    for (int i = 0; i != t2; ++i) f[i] = (i64)f[i] * exp_t[i] % mod;
    IDFT(f, t2);

    std::fill(f + t, f + t2, 0);
  }
}
```

例题

1. 计算 $f^k(x)$

   普通做法为多项式快速幂，时间复杂度 $O(n\log n\log k)$。

   当 $[x^0]f(x)=1$ 时，有：

$$f^k(x)=\exp\left(k\ln f(x)\right)$$

   当 $[x^0]f(x)\neq 1$ 时，设 $f(x)$ 的最低次项为 f_ix^i，则：

$$f^k(x)=f_i^kx^{ik}\exp\left(k\ln\frac{f(x)}{f_ix^i}\right)$$

   时间复杂度 $O(n\log n)$。

多项式三角函数

给定多项式 $f\left(x\right)$，求模 x^{n} 意义下的 $\sin{f\left(x\right)}, \cos{f\left(x\right)}$ 与 $\tan{f\left(x\right)}$。

解法

首先由 Euler's formula $\left(\mathrm{e}^{\mathrm{i}x} = \cos{x} + \mathrm{i}\sin{x}\right)$ 可以得到 三角函数的另一个表达式[5]：

$$\begin{aligned}\sin{x} &= \frac{\mathrm{e}^{\mathrm{i}x} - \mathrm{e}^{-\mathrm{i}x}}{2\mathrm{i}}\\ \cos{x} &= \frac{\mathrm{e}^{\mathrm{i}x} + \mathrm{e}^{-\mathrm{i}x}}{2}\end{aligned}$$

那么代入 $f\left(x\right)$ 就有：

$$\begin{aligned}\sin{f\left(x\right)} &= \frac{\exp{\left(\mathrm{i}f\left(x\right)\right)} - \exp{\left(-\mathrm{i}f\left(x\right)\right)}}{2\mathrm{i}}\\ \cos{f\left(x\right)} &= \frac{\exp{\left(\mathrm{i}f\left(x\right)\right)} + \exp{\left(-\mathrm{i}f\left(x\right)\right)}}{2}\end{aligned}$$

直接按上述表达式编写程序即可得到模 x^{n} 意义下的 $\sin{f\left(x\right)}$ 与 $\cos{f\left(x\right)}$。再由 $\tan{f\left(x\right)} = \frac{\sin{f\left(x\right)}}{\cos{f\left(x\right)}}$ 可求得 $\tan{f\left(x\right)}$。

代码

" 多项式三角函数"

注意到我们是在 $\mathbb{Z}_{998244353}$ 上做 NTT，那么相应地，虚数单位 i 应该被换成 86583718 或 911660635：

$$\begin{aligned} & \mathrm{i}=\sqrt{-1}\equiv\sqrt{998244352}\pmod{998244353}\\ \implies\quad & \mathrm{i}\equiv 86583718\pmod{998244353}\\ \text{or}\quad & \mathrm{i}\equiv 911660635\pmod{998244353}\end{aligned}$$

```
constexpr int maxn = 262144;
constexpr int mod = 998244353;
constexpr int imgunit = 86583718; /* sqrt(-1) = sqrt(998233452) */

using i64 = long long;
using poly_t = int[maxn];
using poly = int *const;

void polytri(const poly &h, const int n, poly &sin_t, poly &cos_t) {
  /* sin(f) = (exp(i * f) - exp(- i * f)) / 2i */
  /* cos(f) = (exp(i * f) + exp(- i * f)) / 2 */
  /* tan(f) = sin(f) / cos(f) */
  assert(h[0] == 0);
  static poly_t tri1_t, tri2_t;

  for (int i = 0; i != n; ++i) tri2_t[i] = (i64)h[i] * imgunit % mod;
  polyexp(tri2_t, n, tri1_t);
  polyinv(tri1_t, n, tri2_t);

  if (sin_t != nullptr) {
    const int invi = fpow(pls(imgunit, imgunit), mod - 2);
    for (int i = 0; i != n; ++i)
      sin_t[i] = (i64)(tri1_t[i] - tri2_t[i] + mod) * invi % mod;
  }
  if (cos_t != nullptr) {
    for (int i = 0; i != n; ++i) cos_t[i] = div2(pls(tri1_t[i], tri2_t[i]));
  }
}
```

多项式反三角函数

给定多项式 $f(x)$，求模 x^n 意义下的 $\arcsin f(x), \arccos f(x)$ 与 $\arctan f(x)$。

解法

仿照求多项式 ln 的方法，对反三角函数求导再积分可得：

$$
\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}x}\arcsin x &= \frac{1}{\sqrt{1-x^2}} \\
\arcsin x &= \int \frac{1}{\sqrt{1-x^2}}\mathrm{d}x \\
\frac{\mathrm{d}}{\mathrm{d}x}\arccos x &= -\frac{1}{\sqrt{1-x^2}} \\
\arccos x &= -\int \frac{1}{\sqrt{1-x^2}}\mathrm{d}x \\
\frac{\mathrm{d}}{\mathrm{d}x}\arctan x &= \frac{1}{1+x^2} \\
\arctan x &= \int \frac{1}{1+x^2}\mathrm{d}x
\end{aligned}
$$

那么代入 $f(x)$ 就有：

$$
\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}x}\arcsin f(x) &= \frac{f'(x)}{\sqrt{1-f^2(x)}} \\
\arcsin f(x) &= \int \frac{f'(x)}{\sqrt{1-f^2(x)}}\mathrm{d}x \\
\frac{\mathrm{d}}{\mathrm{d}x}\arccos f(x) &= -\frac{f'(x)}{\sqrt{1-f^2(x)}} \\
\arccos f(x) &= -\int \frac{f'(x)}{\sqrt{1-f^2(x)}}\mathrm{d}x \\
\frac{\mathrm{d}}{\mathrm{d}x}\arctan f(x) &= \frac{f'(x)}{1+f^2(x)} \\
\arctan f(x) &= \int \frac{f'(x)}{1+f^2(x)}\mathrm{d}x
\end{aligned}
$$

直接按式子求就可以了。

代码

" 多项式反三角函数"

```
constexpr int maxn = 262144;
constexpr int mod = 998244353;

using i64 = long long;
using poly_t = int[maxn];
using poly = int *const;

void derivative(const poly &h, const int n, poly &f) {
  for (int i = 1; i != n; ++i) f[i - 1] = (i64)h[i] * i % mod;
  f[n - 1] = 0;
}

void integrate(const poly &h, const int n, poly &f) {
  for (int i = n - 1; i; --i) f[i] = (i64)h[i - 1] * inv[i] % mod;
  f[0] = 0; /* C */
}

void polyarcsin(const poly &h, const int n, poly &f) {
  /* arcsin(f) = ∫ f' / sqrt(1 - f^2) dx  */
  static poly_t arcsin_t;
  const int t = n << 1;
  std::copy(h, h + n, arcsin_t);
  std::fill(arcsin_t + n, arcsin_t + t, 0);

  DFT(arcsin_t, t);
  for (int i = 0; i != t; ++i) arcsin_t[i] = sqr(arcsin_t[i]);
  IDFT(arcsin_t, t);

  arcsin_t[0] = sub(1, arcsin_t[0]);
  for (int i = 1; i != n; ++i)
    arcsin_t[i] = arcsin_t[i] ? mod - arcsin_t[i] : 0;

  polysqrt(arcsin_t, n, f);
  polyinv(f, n, arcsin_t);
```

```
  derivative(h, n, f);

  DFT(f, t);
  DFT(arcsin_t, t);
  for (int i = 0; i != t; ++i) arcsin_t[i] = (i64)f[i] * arcsin_t[i] % mod;
  IDFT(arcsin_t, t);

  integrate(arcsin_t, n, f);
}

void polyarccos(const poly &h, const int n, poly &f) {
  /* arccos(f) = - ∫ f' / sqrt(1 - f^2) dx  */
  polyarcsin(h, n, f);
  for (int i = 0; i != n; ++i) f[i] = f[i] ? mod - f[i] : 0;
}

void polyarctan(const poly &h, const int n, poly &f) {
  /* arctan(f) = ∫ f' / (1 + f^2) dx  */
  static poly_t arctan_t;
  const int t = n << 1;
  std::copy(h, h + n, arctan_t);
  std::fill(arctan_t + n, arctan_t + t, 0);

  DFT(arctan_t, t);
  for (int i = 0; i != t; ++i) arctan_t[i] = sqr(arctan_t[i]);
  IDFT(arctan_t, t);

  inc(arctan_t[0], 1);
  std::fill(arctan_t + n, arctan_t + t, 0);

  polyinv(arctan_t, n, f);
  derivative(h, n, arctan_t);

  DFT(f, t);
  DFT(arctan_t, t);
  for (int i = 0; i != t; ++i) arctan_t[i] = (i64)f[i] * arctan_t[i] % mod;
  IDFT(arctan_t, t);

  integrate(arctan_t, n, f);
}
```

参考资料与链接

[1] Elementary function——Wikipedia

[2] 「POJ 1737」Connected Graph

[3] P5205【模板】多项式开根

[4] 「Codeforces Round #250」E. The Child and Binary Tree

[5] 三角函数的另一个表达式

9.13.10 常系数齐次线性递推

问题

给定一个线性递推数列 $\{f_i\}$ 的前 k 项 $f_0 \dots f_{k-1}$，和其递推式 $f_n = \sum_{i=1}^{k} f_{n-i}a_i$ 的各项系数 a_i，求 f_n。

前置知识

多项式取模。

做法

定义 $F(\sum c_i x^i) = \sum c_i f_i$，那么答案就是 $F(x^n)$。

由于 $f_n = \sum_{i=1}^{k} f_{n-i}a_i$，所以 $F(x^n) = F(\sum_{i=1}^{k} a_i x^{n-i})$，所以 $F(x^n - \sum_{i=1}^{k} a_i x^{n-i}) = F(x^{n-k}(x^k - \sum_{i=0}^{k-1} a_{k-i} x^i)) = 0$。

设 $G(x) = x^k - \sum_{i=0}^{k-1} a_{k-i} x^i$。

那么 $F(A(x) + x^n G(x)) = F(A(x)) + F(x^n G(x)) = F(A(x))$。

那么就可以通过多次对 $A(x)$ 加上 $x^n G(x)$ 的倍数来降低 $A(x)$ 的次数。

也就是求 $F(A(x) \bmod G(x))$。$A(x) \bmod G(x)$ 的次数不超过 $k-1$，而 $f_{0..k-1}$ 已经给出了，就可以算了。

问题转化成了快速地求 $x^n \bmod G(x)$，只要将 普通快速幂 中的乘法与取模换成 多项式乘法 与 多项式取模 就可以在 $O(k \log k \log n)$ 的时间复杂度内解决这个问题了。

矩阵的解释

该算法由 Fiduccia 在 1985 年提出，对于 $t \geq 0$ 我们定义列向量 v_t 为

$$v_t = \begin{bmatrix} f_t \\ f_{t+1} \\ \vdots \\ f_{t+k-1} \end{bmatrix}$$

那么不难发现

$$\underbrace{\begin{bmatrix} f_{t+1} \\ f_{t+2} \\ \vdots \\ f_{t+k} \end{bmatrix}}_{v_{t+1}} = \underbrace{\begin{bmatrix} & 1 & & \\ & & \ddots & \\ & & & 1 \\ a_k & a_{k-1} & \cdots & a_1 \end{bmatrix}}_{M} \times \underbrace{\begin{bmatrix} f_t \\ f_{t+1} \\ \vdots \\ f_{t+k-1} \end{bmatrix}}_{v_t}$$

而因为 v_{t+k} 中每一行都满足这个递推关系，我们将 v_{t+k} 描述为一个线性组合如

$$v_{t+k} = \sum_{i=1}^{k} a_i v_{t+k-i}$$

有

$$M^k v_t = \sum_{i=1}^{k} a_i M^{k-i} v_t$$

发现若能将两边的 v_t 消去后得到多项式 $\Gamma(x) = x^k - \sum_{i=1}^{k} a_i x^{k-i}$ 满足 $\Gamma(M) = O$ 其中 O 为一个 $k \times k$ 的零矩阵。

假设我们要求 M^n 可以构造多项式 $f(x) = x^n$ 那么 $f(M) = M^n$，而现在我们可将 $f(x)$ 写成 $f(x) = Q(x)\Gamma(x) + R(x)$ 而其中零矩阵是没有贡献的，那么 $f(M) = R(M)$。

但是注意矩阵乘法不满足消去律，此处我们定义矩阵 M 的特征多项式为 $\Gamma(x) = \det(xI - M)$，其中 I 为一个 $k \times k$ 的单位矩阵。Cayley–Hamilton 定理指出 $\Gamma(M) = O$，我们观察 M 的形式较为特殊，为下 Hessenberg 矩阵，其特征多项式比起一般矩阵更容易计算。

我们从右下角的 2×2 矩阵开始计算特征多项式

$$\Gamma_2(x)=\det\left(\begin{bmatrix} x & -1 \\ -a_2 & x-a_1 \end{bmatrix}\right)=x^2-a_1x-a_2$$

右下角 3×3 矩阵按照第一行的余子式展开有

$$\begin{aligned}\Gamma_3(x)&=\det\left(\begin{bmatrix} x & -1 & 0 \\ 0 & x & -1 \\ -a_3 & -a_2 & x-a_1 \end{bmatrix}\right)\\ &=x\cdot(-1)^{1+1}\cdot\Gamma_2(x)+(-1)\cdot(-1)^{1+2}\cdot\det\left(\begin{bmatrix} 0 & -1 \\ -a_3 & x-a_1 \end{bmatrix}\right)\\ &=x^3-a_1x^2-a_2x-a_3\end{aligned}$$

观察并归纳有

$$\Gamma(x)=\Gamma_k(x)=x^k-\sum_{i=1}^{k}a_ix^{k-i}$$

至此我们可以使用上面的结论。令 $g(x)=f(x) \bmod \Gamma(x)$ 有 $g(M)=M^n$。而 $\deg(g(x))<k$ 显然，令 $g(x)=g_0+g_1x+\cdots+g_{k-1}x^{k-1}$ 那么

$$M^nv_0=\sum_{i=0}^{k-1}g_iM^iv_0$$

即

$$v_n=\sum_{i=0}^{k-1}g_iv_i$$

我们关注 $v_0,v_1,\dots,v_{k-1}$ 的第一行就是 $f_0,f_1,\dots,f_{k-1}$ 已知，那么 f_n 可在 $O(k)$ 时间简单得到。求出 $g(x)$ 则可用快速幂和多项式取模与上述解释是一样的。该算法常数较大，使用生成函数可以得到一个常数更小的算法，见 一种新的线性递推计算方法[1]。

参考资料与注释

[1] 一种新的线性递推计算方法

9.13.11 多项式平移 | 连续点值平移

多项式平移

多项式平移是简单情况的多项式复合变换，给出 $f(x)=\sum_{i=0}^{n}f_ix^i$ 的系数和一个常数 c，求 $f(x+c)$ 的系数，即 $f(x)\mapsto f(x+c)$。

分治法

令

$$f(x)=f_0(x)+x^{\lfloor n/2\rfloor}f_1(x)$$

那么

$$f(x+c)=f_0(x+c)+(x+c)^{\lfloor n/2\rfloor}f_1(x+c)$$

$(x+c)^{\lfloor n/2\rfloor}$ 的系数为二项式系数，那么

$$T(n)=2T(n/2)+O(n\log n)=O(n\log^2 n)$$

其中 $O(n\log n)$ 为多项式乘法的时间。

Taylor 公式法

对 $f(x)$ 在 c 处应用 Taylor 公式，有

$$f(x)=f(c)+\frac{f'(c)}{1!}(x-c)+\frac{f''(c)}{2!}(x-c)^2+\cdots+\frac{f^{(n)}(c)}{n!}(x-c)^n$$

那么

$$f(x+c)=f(c)+\frac{f'(c)}{1!}x+\frac{f''(c)}{2!}x^2+\cdots+\frac{f^{(n)}(c)}{n!}x^n$$

观察到对于 $t\geq 0$ 有

$$\begin{aligned}t![x^t]f(x+c)&=f^{(t)}(c)\\&=\sum_{i=t}^{n}f_i i!\frac{c^{i-t}}{(i-t)!}\\&=\sum_{i=0}^{n-t}f_{i+t}(i+t)!\frac{c^i}{i!}\end{aligned}$$

令

$$\begin{aligned}A_0(x)&=\sum_{i=0}^{n}f_{n-i}(n-i)!x^i\\B_0(x)&=\sum_{i=0}^{n}\frac{c^i}{i!}x^i\end{aligned}$$

那么

$$\begin{aligned}[x^{n-t}](A_0(x)B_0(x))&=\sum_{i=0}^{n-t}([x^{n-t-i}]A_0(x))([x^i]B_0(x))\\&=\sum_{i=0}^{n-t}f_{i+t}(i+t)!\frac{c^i}{i!}\\&=t![x^t]f(x+c)\end{aligned}$$

二项式定理法

考虑二项式定理 $(a+b)^n=\sum_{i=0}^{n}\binom{n}{i}a^ib^{n-i}$ 那么

$$\begin{aligned}f(x+c)&=\sum_{i=0}^{n}f_i(x+c)^i\\&=\sum_{i=0}^{n}f_i\left(\sum_{j=0}^{i}\binom{i}{j}x^jc^{i-j}\right)\\&=\sum_{i=0}^{n}f_i i!\left(\sum_{j=0}^{i}\frac{x^j}{j!}\frac{c^{i-j}}{(i-j)!}\right)\\&=\sum_{i=0}^{n}\frac{x^i}{i!}\left(\sum_{j=i}^{n}f_j j!\frac{c^{j-i}}{(j-i)!}\right)\end{aligned}$$

得到的结果与上述方法相同。

连续点值平移

” 例题 LOJ 166 拉格朗日插值 2[1]”

给出度数小于等于 n 的多项式 f 的连续点值 $f(0),f(1),\dots,f(n)$，在模 998244353 意义下计算 $f(c),f(c+1),\dots,f(c+n)$，其中 $1\leq n\leq 10^5,n<m\leq 10^8$。

Lagrange 插值公式法

考虑 Lagrange 插值公式

$$
\begin{aligned}
f(x) &= \sum_{0\leq i\leq n} f(i) \prod_{0\leq j\leq n\land j\neq i} \frac{x-j}{i-j} \\
&= \sum_{0\leq i\leq n} f(i)\frac{x!}{(x-n-1)!(x-i)}\frac{(-1)^{n-i}}{i!(n-i)!} \\
&= \frac{x!}{(x-n-1)!}\sum_{0\leq i\leq n}\frac{f(i)}{(x-i)}\frac{(-1)^{n-i}}{i!(n-i)!}
\end{aligned}
$$

上式虽然是卷积形式但不能保证分母上 $x-i\neq 0$，所以下面仅考虑 $c>n$ 的情况，其他情况（如系数在模素数意义下时须避免 $B_0(x)$ 系数的分母出现零）可以分类讨论解决，令

$$
\begin{aligned}
A_0(x) &= \sum_{0\leq i\leq n}\frac{f(i)(-1)^{n-i}}{i!(n-i)!}x^i \\
B_0(x) &= \sum_{i\geq 0}\frac{1}{c-n+i}x^i
\end{aligned}
$$

那么对于 $t\geq 0$ 有

$$
\begin{aligned}
[x^{n+t}](A_0(x)B_0(x)) &= \sum_{i=0}^{n+t}([x^i]A_0(x))([x^{n+t-i}]B_0(x)) \\
&= \sum_{i=0}^{n}\frac{f(i)(-1)^{n-i}}{i!(n-i)!}\frac{1}{c+t-i} \\
&= \frac{(c+t-n-1)!}{(c+t)!}f(c+t)
\end{aligned}
$$

实现中取 $B_0(x)$ 需要的部分截断可求出更多点值，且可利用循环卷积。

对问题稍加修改，假设对于某个 d 给出的点值为 $f(d),f(d+k),\dots,f(d+nk)$，我们可以计算 $f(c+d),f(c+d+k),\dots,f(c+d+nk)$，视作平移 $g(x)=f(d+kx)$ 的点值 $g(0),g(1),\dots,g(n)$ 为 $g(c/k),g(c/k+1),\dots,g(c/k+n)$。

Lagrange 插值公式也给出了通过维护一些前后缀积的线性计算单个点值的方法。

应用

同一行第一类无符号 Stirling 数

" 例题 P5408 第一类斯特林数 · 行[2]"

在模素数 167772161 意义下求 ${n \brack 0},{n \brack 1},\dots,{n \brack n}$，其中 $1\leq n<262144$。

考虑

$$
x^{\overline{n}} = \sum_{i=0}^{n}{n \brack i}x^i,\quad n\geq 0
$$

其中 $x^{\overline{n}}=x\cdot(x+1)\cdots(x+n-1)$ 为上升阶乘幂，令 $f_n(x)=x^{\overline{n}}$ 那么

$$
f_{2n}(x)=x^{\overline{n}}\cdot(x+n)^{\overline{n}}=f_n(x)f_n(x+n)
$$

通过多项式平移可在 $O(n\log n)$ 求出 $f_n(x+n)$，问题被缩小为原先的一半即求出 $f_n(x)$ 的系数，那么

$$
T(n)=T(n/2)+O(n\log n)=O(n\log n)
$$

模素数意义下阶乘

" 例题 P5282【模板】快速阶乘算法[3]"

求 $n! \bmod p$，其中 p 为素数且 $1 \le n < p \le 2^{31}-1$。

令 $v = \lfloor \sqrt{n} \rfloor$ 和 $g(x) = \prod_{i=1}^{v}(x+i)$ 那么

$$n! \equiv \left(\prod_{i=0}^{v-1} g(iv)\right) \cdot \prod_{i=v^2+1}^{n} i \pmod p$$

其中 $\prod_{i=v^2+1}^{n} i$ 可在 $O(\sqrt{n})$ 时间计算，我们希望可以快速计算上式的前半部分。

多项式多点求值

$g(x)$ 系数的计算可用上述多项式平移算法在 $O(n\log n)$ 时间得到，但多点求值计算 $g(0), g(v), g(2v), \dots, g(v^2-v)$ 需要 $O(\sqrt{n}\log^2 n)$ 时间。

连续点值平移

令 $g_d(x) = \prod_{i=1}^{d}(x+i)$，我们可以用 $d+1$ 个点值 $g_d(0), g_d(v), \dots, g_d(dv)$ 唯一确定这个度数为 d 的多项式，又

$$g_{2d}(x) = g_d(x)g_d(x+d)$$

所以只需 $2d+1$ 个点值可以唯一确定 $g_{2d}(x)$，那么使用连续点值平移计算 $g_d((d+1)v), g_d((d+2)v), \dots, g_d(2dv)$（即平移 $h(x) = g_d(vx)$ 的点值 $h(0), h(1), \dots, h(d)$ 为 $h(d+1), h(d+2), \dots, h(2d)$）和 $g_d(d), g_d(v+d), \dots, g_d(2dv+d)$（即平移 $h(x) = g_d(vx)$ 的点值 $h(0), h(1), \dots, h(d)$ 为 $h(d/v), h(d/v+1), h(d/v+2), \dots, h(d/v+2d)$）后将这两者的对应点值相乘即得 $g_{2d}(0), g_{2d}(v), \dots, g_{2d}(2dv)$。

由 $g_d(0), g_d(v), \dots, g_d(dv)$ 计算 $g_{d+1}(0), g_{d+1}(v), \dots, g_{d+1}(dv), g_{d+1}((d+1)v)$ 考虑

$$g_{d+1}(x) = (x+d+1)\cdot g_d(x)$$

额外增加的一个点值使用线性时间的算法即可。那么在开始时维护 $g_1(0)=1, g_1(v)=v+1$ 后使用连续点值平移来倍增地维护这些点值，有

$$T(n) = T(n/2) + O(n\log n) = O(n\log n)$$

而我们只需要约 $\sqrt{n}$ 个点值，所以时间复杂度为 $O(\sqrt{n}\log n)$。

模素数意义下二项式系数前缀和

" 例题 LOJ 6386 组合数前缀和[4]"

求 $\sum_{i=0}^{m}\binom{n}{i} \bmod 998244353$，其中 $0 \le m \le n \le 9\times 10^8$。

考虑使用矩阵描述 $n! = n\cdot(n-1)!$ 这一步递推，我们有

$$\begin{bmatrix} n! \end{bmatrix} = \left(\prod_{i=0}^{n-1}\begin{bmatrix} i+1 \end{bmatrix}\right)\begin{bmatrix} 1 \end{bmatrix}$$

类似的可以将二项式系数前缀和的递推描述为

$$\begin{bmatrix} \binom{n}{m+1} \\ \sum_{i=0}^{m}\binom{n}{i} \end{bmatrix} = \begin{bmatrix} (n-m)/(m+1) & 0 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} \binom{n}{m} \\ \sum_{i=0}^{m-1}\binom{n}{i} \end{bmatrix}$$

注意矩阵乘法的顺序，那么

$$\begin{aligned}\begin{bmatrix} \binom{n}{m+1} \\ \sum_{i=0}^{m}\binom{n}{i} \end{bmatrix} &= \left(\prod_{i=0}^{m}\begin{bmatrix} (n-i)/(i+1) & 0 \\ 1 & 1 \end{bmatrix}\right)\begin{bmatrix} 1 \\ 0 \end{bmatrix} \\ &= \frac{1}{(m+1)!}\left(\prod_{i=0}^{m}\begin{bmatrix} n-i & 0 \\ i+1 & i+1 \end{bmatrix}\right)\begin{bmatrix} 1 \\ 0 \end{bmatrix}\end{aligned}$$

令 $v=\lfloor\sqrt{m}\rfloor$，考虑维护矩阵

$$\begin{aligned}M_d(x)&=\prod_{i=1}^{d}\begin{bmatrix}-x+n+1-i & 0\\ x+i & x+i\end{bmatrix}\\&=\begin{bmatrix}f_d(x) & 0\\ g_d(x) & h_d(x)\end{bmatrix}\end{aligned}$$

的点值 $M_d(0),M_d(v),\dots,M_d(dv)$ 即 $f_d(0),f_d(v),\dots,f_d(dv)$、$h_d(0),\dots,h_d(dv)$ 和 $g_d(0),\dots,g_d(dv)$，又

$$\begin{aligned}M_{2d}(x)&=\prod_{i=1}^{2d}\begin{bmatrix}-x+n+1-i & 0\\ x+i & x+i\end{bmatrix}\\&=\left(\prod_{i=1}^{d}\begin{bmatrix}-x-d+n+1-i & 0\\ x+d+i & x+d+i\end{bmatrix}\right)\left(\prod_{i=1}^{d}\begin{bmatrix}-x+n+1-i & 0\\ x+i & x+i\end{bmatrix}\right)\\&=\begin{bmatrix}f_d(x+d) & 0\\ g_d(x+d) & h_d(x+d)\end{bmatrix}\begin{bmatrix}f_d(x) & 0\\ g_d(x) & h_d(x)\end{bmatrix}\\&=\begin{bmatrix}f_d(x+d)f_d(x) & 0\\ g_d(x+d)f_d(x)+h_d(x+d)g_d(x) & h_d(x+d)h_d(x)\end{bmatrix}\end{aligned}$$

且矩阵右下角元素恰为我们在阶乘算法中所维护的，那么

$$\prod_{i=0}^{m}\begin{bmatrix}n-i & 0\\ i+1 & i+1\end{bmatrix}=\left(\prod_{i=(k+1)v}^{m}\begin{bmatrix}n-i & 0\\ i+1 & i+1\end{bmatrix}\right)\begin{bmatrix}f_v(kv) & 0\\ g_v(kv) & h_v(kv)\end{bmatrix}\cdots\begin{bmatrix}f_v(0) & 0\\ g_v(0) & h_v(0)\end{bmatrix}$$

可在 $O(\sqrt{m}\log m)$ 时间完成计算。

模素数意义下调和数

" **例题 P5702 调和级数求和**[5] "

求 $\sum_{i=1}^{n} i^{-1} \bmod p$，其中 p 为素数且 $1\le n<p<2^{30}$。

记 $H_n=\sum_{k=1}^{n}k^{-1}$，一步递推为

$$\begin{bmatrix}(n+1)!\\ (n+1)!H_{n+1}\end{bmatrix}=\begin{bmatrix}n+1 & 0\\ 1 & n+1\end{bmatrix}\begin{bmatrix}n!\\ n!H_n\end{bmatrix}$$

那么

$$\begin{bmatrix}\begin{bmatrix}n+1\\1\end{bmatrix}\\ \begin{bmatrix}n+1\\2\end{bmatrix}\end{bmatrix}=\begin{bmatrix}n!\\ n!H_n\end{bmatrix}=\left(\prod_{i=0}^{n-1}\begin{bmatrix}i+1 & 0\\ 1 & i+1\end{bmatrix}\right)\begin{bmatrix}1\\0\end{bmatrix}$$

在这里 $\begin{bmatrix}n+1\\1\end{bmatrix}$ 和 $\begin{bmatrix}n+1\\2\end{bmatrix}$ 为第一类无符号 Stirling 数。维护点值矩阵的方法同上。

整式递推

对于更一般的情况，类似于上述快速阶乘算法的案例，我们期望得到一个怎么样的算法？

" **例题 P6115【模板】整式递推**[6] "

现有数列 a 满足 $\forall n\ge m,\sum_{k=0}^{m}a_{n-k}P_k(n)=0$，其中 P_k 为不超过 d 次的多项式。

给定所有 P_k 的系数，和 $a_0,a_1,\dots,a_{m-1}$，求 a_n。对 998244353 取模。$n\le 6\times 10^8$，$1\le m,d\le 7$，时限 $7s$。

为了更系统地描述上述几道例题中构造矩阵的过程，我们引入 λ 矩阵 的概念。

为了实现整式递推，我们应当注意到快速阶乘算法过程中，我们维护的点值其实并不是 $n!$，而是 $\prod_{i=0}^{T-1}(aT+i)$，即**一对点值之间的倍数关系**。

由于整式递推阶数 m 不止是 1 了，我们就**不能直接维护一对数之间的倍数关系了**；而是维护出一**对 m 维向量之间的线性变换**，即 $m \times m$ 的一个矩阵，**矩阵的每一项对应于某个多项式的一个点值**。

容易发现，对于一般的整式递推远处系数求值问题，我们可以构造

$$-\frac{1}{P_0(n)}\begin{bmatrix} P_1(n) & P_2(n) & P_3(n) & \cdots & P_{m-1}(n) & P_m(n) \\ -P_0(n) & & & & & \\ & -P_0(n) & & & & \\ & & -P_0(n) & & & \\ & & & \ddots & & \\ & & & & -P_0(n) & \end{bmatrix}\begin{bmatrix} a_{n-1} \\ a_{n-2} \\ a_{n-3} \\ \vdots \\ a_{n-m+1} \\ a_{n-m} \end{bmatrix} = \begin{bmatrix} a_n \\ a_{n-1} \\ a_{n-2} \\ \vdots \\ a_{n-m+2} \\ a_{n-m+1} \end{bmatrix}$$

设

$$B(\lambda) = \begin{bmatrix} P_1(\lambda) & P_2(\lambda) & P_3(\lambda) & \cdots & P_{m-1}(\lambda) & P_m(\lambda) \\ -P_0(\lambda) & & & & & \\ & -P_0(\lambda) & & & & \\ & & -P_0(\lambda) & & & \\ & & & \ddots & & \\ & & & & -P_0(\lambda) & \end{bmatrix}$$

我们先撇开前面的 $-\frac{1}{P_0(n)}$ 因子不论，我们现在要维护 $\prod_{i=0}^{T-1} B(aT+m+i)$ 这种形式的量，其中乘法自右往左。

容易发现 $B_T(\lambda) = \prod_{i=0}^{T-1} B(\lambda+i)$ 是一个各项次数不高于 dT 的 λ 矩阵，只用 $dT+1$ 个值即可维护。

于是我们维护出 $B_T(m)$，$B_T(m+T)$，$B_T(m+2T)$，...，$B_T(m+(dT-1)T)$，$B_T(m+dT^2)$ 这几个 λ 矩阵的**点值**，然后用类似于快速阶乘算法的方式暴力进行多项式点值平移和倍增就好了。

具体地，为了让 $t = \log_2 T$ 抬高 1，我们这么干：

1. 在 $O(m^2 dT \log(dT))$ 时间内获取 $B_T(p+dT^2)$，$B_T(p+(dT+1)T)$，$B_T(p+(dT+2)T)$，⋯，$B_T(p+(2dT-1)T)$，$B_T(p+(2dT)dT)$。
2. 在 $O(m^2 dT \log(dT))$ 时间内获取 $B_T(p+2dT^2)$，$B_T(p+(2dT+1)T)$，$B_T(p+(2dT+2)T)$，⋯，$B_T(p+(3dT-1)T)$，$B_T(p+(3dT)dT)$。
3. 在 $O(m^2 dT \log(dT))$ 时间内获取 $B_T(p+3dT^2)$，$B_T(p+(3dT+1)T)$，$B_T(p+(3dT+2)T)$，⋯，$B_T(p+(4dT-1)T)$，$B_T(p+(4dT)dT)$。
4. 计算 $B_{2T}(v) = B_T(v+T)B_T(v)$。

我们每轮花费 $O(m^2 dT \log(dT))$ 的复杂度进行平移；同时，我们每轮只用做 $\Theta(dT)$ 次矩阵乘法，复杂度可以认为是 $O(m^3 dT)$。

最后，我们只用做到 $T \geq \sqrt{n/d}$ 即可。

之前的 $-\frac{1}{P_0(n)}$ 因子可以用类似的方法解决。

这样，我们预处理的复杂度即为 $\Theta(\sqrt{nd}(m^3 + m^2\log(nd)))$。

考虑查询，我们只用 $\Theta(n/T)$ 次向量与矩阵的乘法，以及 $O(T)$ 次暴力转移。

容易发现这部分计算并不是复杂度瓶颈。

因此，该算法的总复杂度为 $\Theta(\sqrt{nd}(m^3 + m^2\log(nd)))$。

编码时，我们可以使用循环卷积的技巧来减小 NTT 的常数。

在实际应用时，我们往往是对一个已知的微分有限的 GF 提取其远处系数，从而 m, d 均为常数，也即做到了 $\Theta(\sqrt{n}\log n)$ 的远处系数求值。

参考文献

- Alin Bostan, Pierrick Gaudry, and Eric Schost. Linear recurrences with polynomial coefficients and application to integer factorization and Cartier–Manin operator.
- Min_25 的博客

- ZZQ 的博客 - 阶乘模大质数[7]

参考资料与注释

[1] LOJ 166 拉格朗日插值 2

[2] P5408 第一类斯特林数 · 行

[3] P5282【模板】快速阶乘算法

[4] LOJ 6386 组合数前缀和

[5] P5702 调和级数求和

[6] P6115【模板】整式递推

[7] ZZQ 的博客 - 阶乘模大质数

9.13.12 符号化方法

符号化方法（symbolic method）是将组合对象快速转换成生成函数的一种方法，我们将考虑对于集合上定义的特定运算，然后导出其对应的生成函数的运算。

我们称一个组合类（或简称为类）为 $(\mathcal{A},|\cdot|)$，其中 $\mathcal{A}$ 为组合对象的集合，函数 $|\cdot|$ 将每一个组合对象映射为一个非负整数，一般称为大小函数。需要注意的是这个非负整数不能是无限大的。例如对于字符集为 $\{0,1\}$ 的字符串，可以将字符串的长度设置为其大小函数；对于树或图可将节点的数量设置为其大小函数，注意这并非绝对，也可能将某些特定节点的大小函数设置为 0 等。

本文是基于 Analytic Combinatorics 一书第一章的简化。

无标号体系

在无标号体系中将使用普通生成函数（OGF）。对于集合 $\mathcal{A}$ 其对应 OGF 记为

$$A(z)=\sum_{\alpha\in\mathcal{A}}z^{|\alpha|}=\sum_{n\geq 0}a_nz^n$$

我们约定使用同一组的字母表示同一个类对应的生成函数等，例如用 a_n 表示 $[z^n]A(z)$ 即 $A(z)$ 中 z^n 的系数，用 $\mathcal{A}_n$ 表示 $\mathcal{A}$ 中大小函数为 n 的对象的集合（所以 $a_n=\operatorname{card}(\mathcal{A}_n)$ 其中 card 为基数（cardinality））。

本文将不讨论可容许性（admissibility），读者可参考文献中的内容。

下面将引入两种特殊的组合类和组合对象：

- 记 ϵ 为中性对象（neutral object）和 $\mathcal{E}=\{\epsilon\}$ 为中性类（neutral class），中性对象的大小为 0，中性类的 OGF 为 $E(z)=1$。
- 记 $\circ$ 或 $\bullet$ 为原子对象（atom object）和 $\mathcal{Z}_\circ=\{\circ\}$ 或 $\mathcal{Z}_\bullet=\{\bullet\}$ 或简写为 $\mathcal{Z}$ 为原子类（atom class），原子对象的大小为 1，原子类的 OGF 为 $Z(z)=z$。

对于两个组合类 $\mathcal{A}$ 和 $\mathcal{B}$ 在组合意义上同构记为 $\mathcal{A}=\mathcal{B}$ 或 $\mathcal{A}\cong\mathcal{B}$，但仅当该同构不平凡时才使用后者的记号。我们有

$$\mathcal{A}\cong\mathcal{E}\times\mathcal{A}\cong\mathcal{A}\times\mathcal{E}$$

其中 $\times$ 为二元运算，表示集合的笛卡尔积。

集合的（不相交）并构造

对于类 $\mathcal{A}$ 和 $\mathcal{B}$ 的并记为

$$\mathcal{A}+\mathcal{B}=(\mathcal{E}_1\times\mathcal{A})+(\mathcal{E}_2\times\mathcal{B})$$

如此定义可以不违背集合论中集合不相交的要求，我们可以想象成将 $\mathcal{A}$ 中的对象染色成红色，将 $\mathcal{B}$ 中的对象染色成蓝色。

对应 OGF 为

$$A(z)+B(z)$$

考虑

$$A(z)+B(z)=\sum_{\alpha\in\mathcal{A}}z^{|\alpha|}+\sum_{\beta\in\mathcal{B}}z^{|\beta|}=\sum_{n\ge0}(a_n+b_n)z^n$$

对应形式幂级数的加法。

集合的笛卡尔积构造

对于类 $\mathcal{A}$ 和 $\mathcal{B}$ 的笛卡尔积记为

$$\mathcal{A}\times\mathcal{B}=\{(\alpha,\beta)\mid\alpha\in\mathcal{A},\beta\in\mathcal{B}\}$$

对应 OGF 为

$$A(z)\cdot B(z)$$

我们定义 (α,β) 的大小为其组成部分的大小之和，那么显然也有

$$\gamma=(\alpha_1,\alpha_2,\dots,\alpha_n)\implies|\gamma|=|\alpha_1|+|\alpha_2|+\cdots+|\alpha_n|$$

所以

$$A(z)\cdot B(z)=\left(\sum_{\alpha\in\mathcal{A}}z^{|\alpha|}\right)\left(\sum_{\beta\in\mathcal{B}}z^{|\beta|}\right)=\sum_{(\alpha,\beta)\in(\mathcal{A}\times\mathcal{B})}z^{|\alpha|+|\beta|}=\sum_{n\ge0}\sum_{i+j=n}a_ib_jz^n$$

对应形式幂级数的乘法。

集合的 Sequence 构造

Sequence 构造生成了所有可能的组合。

" 例"

$$\begin{aligned}\mathrm{SEQ}(\{a\})&=\{\epsilon\}+\{a\}+\{(a,a)\}+\{(a,a,a)\}+\cdots\\ \mathrm{SEQ}(\{a,b\})&=\{\epsilon\}+\{a,b\}+\{(a,b)\}+\{(b,a)\}+\{(a,a)\}+\{(b,b)\}\\&\quad+\{(a,b,a)\}+\{(a,b,b)\}+\{(a,a,b)\}\\&\quad+\{(b,b,a)\}+\{(b,a,b)\}+\{(b,b,b)\}+\{(a,a,a)\}+\{(b,a,a)\}\\&\quad+\cdots\end{aligned}$$

可以看到 $\{(a,b)\},\{(b,a)\}$ 这样组成部分的顺序不同的元素被生成了，可以认为 Sequence 构造生成了有序的组合。

我们定义

$$\mathrm{SEQ}(\mathcal{A})=\mathcal{E}+\mathcal{A}+(\mathcal{A}\times\mathcal{A})+(\mathcal{A}\times\mathcal{A}\times\mathcal{A})+\cdots$$

且要求 $\mathcal{A}_0=\emptyset$，也就是 $\mathcal{A}$ 中没有大小为 0 的对象。

对应 OGF 为

$$Q(A(z))=1+A(z)+A(z)^2+A(z)^3+\cdots=\frac{1}{1-A(z)}$$

其中 Q 为 Pólya 准逆（quasi-inversion）。

" 例：有序有根树（ordered rooted tree）"

我们可以使用 Sequence 构造来定义有序有根树，即孩子之间的顺序有意义的有根树，设该组合类为 $\mathcal{T}$ 那么一棵树为一个根节点和树的 Sequence，即

$$\mathcal{T} = \{\bullet\} \times \mathrm{SEQ}(\mathcal{T})$$

对应 OGF 为

$$T(z) = \frac{z}{1 - T(z)}$$

前几项系数为 `0 1 1 2 5 14 42 132 429 1430 4862 16796`，忽略常数项即 OEIS A000108[1]。

集合的 Multiset 构造

Multiset 构造生成了所有可能的组合，但不区分组成部分的元素之间的顺序。

" 例"

$$\begin{aligned}
\mathrm{MSET}(\{a\}) &= \{\epsilon\} + \{a\} + \{(a,a)\} + \{(a,a,a)\} + \cdots \\
\mathrm{MSET}(\{a,b\}) &= \{\epsilon\} + \{a\} + \{(a,a)\} + \{(a,a,a)\} + \cdots \\
&\quad + \{b\} + \{(a,b)\} + \{(a,a,b)\} + \cdots \\
&\quad + \{(b,b)\} + \{(a,b,b)\} + \{(a,a,b,b)\} + \cdots \\
&\quad + \cdots
\end{aligned}$$

注意到 $\{(b,a)\}, \{(a,b,a)\}$ 在 $\mathrm{SEQ}(\{a,b\})$ 中出现，但在 $\mathrm{MSET}(\{a,b\})$ 没有出现，可以认为 Multiset 生成了无序的组合。

我们定义其递推式为

$$\mathrm{MSET}(\{\alpha_0, \alpha_1, \dots, \alpha_n\}) = \mathrm{MSET}(\{\alpha_0, \alpha_1, \dots, \alpha_{n-1}\}) \times \mathrm{SEQ}(\{\alpha_n\})$$

即

$$\mathrm{MSET}(\mathcal{A}) = \prod_{\alpha \in \mathcal{A}} \mathrm{SEQ}(\{\alpha\})$$

且要求 $\mathcal{A}_0 = \emptyset$。或者也可以给出等价的

$$\mathrm{MSET}(\mathcal{A}) = \mathrm{SEQ}(\mathcal{A})/\mathbf{R}$$

其中 $\mathbf{R}$ 为等价关系，我们说 $(\alpha_1, \dots, \alpha_n)\mathbf{R}(\beta_1, \dots, \beta_n)$ 当且仅当存在任一置换 σ 对于所有 j 满足 $\beta_j = \alpha_{\sigma(j)}$。对应 OGF 为

$$\mathrm{Exp}(A(z)) = \prod_{\alpha \in \mathcal{A}} \left(1 - z^{|\alpha|}\right)^{-1} = \prod_{n \ge 1} (1 - z^n)^{-a_n}$$

注意到

$$\ln(1+z) = \frac{z}{1} - \frac{z^2}{2} + \frac{z^3}{3} - \cdots = \sum_{n \ge 1} \frac{(-1)^{n-1} z^n}{n}$$

且 $A(z) = \exp(\ln(A(z)))$ 所以

$$\begin{aligned}
\mathrm{Exp}(A(z)) &= \exp\left(\sum_{n \ge 1} -a_n \cdot \ln(1 - z^n)\right) \\
&= \exp\left(\sum_{n \ge 1} -a_n \cdot \sum_{m \ge 1} \frac{-z^{nm}}{m}\right) \\
&= \exp\left(\frac{A(z)}{1} + \frac{A(z^2)}{2} + \frac{A(z^3)}{3} + \cdots\right)
\end{aligned}$$

其中 Exp 为 Pólya 指数，也被称为 Euler 变换。

" 例题 LOJ 6268. 分拆数[2]"

题意：令 $f(n)$ 表示将 n 进行分拆的方案数，求 $f(1), f(2), \dots, f(10^5)$ 对 998244353 取模的值。

解：设全体正整数类为 $\mathcal{I}$，那么 $\mathcal{I} = \mathrm{SEQ}_{\geq 1}(\mathcal{Z}) = \mathcal{Z} \times \mathrm{SEQ}(\mathcal{Z})$（下标 ≥ 1 为有限制的构造，见后文）。所求即

$$\mathrm{MSET}(\mathcal{I})$$

对应 OGF 前几项系数为 `1 2 3 5 7 11 15 22 30 42`（忽略常数项）即 OEIS A000041[3]。

" 例题 洛谷 P4389 付公主的背包[4]"

题意：给出 n 种体积分别为 $v_1, \dots, v_n$ 的商品和正整数 m，求体积为 $1, 2, \dots, m$ 的背包装满的方案数（商品数量不限，有同体积的不同种商品）对 998244353 取模的值。约定 $1 \leq n, m \leq 10^5$ 且 $1 \leq v_i \leq m$。

解：设商品的组合类为 $\mathcal{A}$，所求即 $\mathrm{MSET}(\mathcal{A})$ 对应 OGF 的系数。

" 例题 洛谷 P5900 无标号无根树计数[5]"

题意：求出 n 个节点的无标号无根树的个数对 998244353 取模的值。约定 $1 \leq n \leq 2 \times 10^5$。

解：设无标号有根树的组合类为 $\mathcal{T}$，那么

$$\mathcal{T} = \{\bullet\} \times \mathrm{MSET}(\mathcal{T})$$

根据 Richard Otter 的论文 The Number of Trees[6] 中的描述，对应无根树的 OGF 为

$$T(z) - \frac{1}{2}T^2(z) + \frac{1}{2}T(z^2)$$

前几项系数为 `1 1 1 2 3 6 11 23 47 106`（忽略常数项）即 OEIS A000055[7]。

集合的 Powerset 构造

Powerset 构造生成了所有子集。

" 例"

$$\begin{aligned}
\mathrm{PSET}(\{a\}) &= \{\epsilon\} + \{a\} \\
\mathrm{PSET}(\{a, b\}) &= \{\epsilon\} + \{a\} + \{b\} + \{(a, b)\} \\
\mathrm{PSET}(\{a, b, c\}) &= \{\epsilon\} + \{a\} + \{b\} + \{(a, b)\} + \{c\} + \{(a, c)\} + \{(b, c)\} + \{(a, b, c)\}
\end{aligned}$$

我们定义其递推式为

$$\mathrm{PSET}(\{\alpha_0, \alpha_1, \dots, \alpha_n\}) = \mathrm{PSET}(\{\alpha_0, \alpha_1, \dots, \alpha_{n-1}\}) \times (\{\epsilon\} + \{\alpha_n\})$$

即

$$\mathrm{PSET}(\mathcal{A}) \cong \prod_{\alpha \in \mathcal{A}} (\{\epsilon\} + \{\alpha\})$$

且要求 $\mathcal{A}_0 = \emptyset$。

对应 OGF 为

$$
\begin{aligned}
\overline{\mathrm{Exp}}(A(z)) &= \prod_{\alpha\in\mathcal{A}} \left(1+z^{|\alpha|}\right) = \prod_{n\geq 1} (1+z^n)^{a_n} \\
&= \exp\left(\sum_{n\geq 1} a_n \cdot \ln\left(1+z^n\right)\right) \\
&= \exp\left(\sum_{n\geq 1} a_n \cdot \sum_{m\geq 1} \frac{(-1)^{m-1} z^{nm}}{m}\right) \\
&= \exp\left(\frac{A(z)}{1} - \frac{A(z^2)}{2} + \frac{A(z^3)}{3} - \cdots\right)
\end{aligned}
$$

其中 $\overline{\mathrm{Exp}}$ 为 Pólya 指数 · 改。

容易发现 $\mathrm{PSET}(\mathcal{A}) \subset \mathrm{MSET}(\mathcal{A})$。

集合的 Cycle 构造

Cycle 构造生成了所有可能的组合，但不区分仅轮换不同的组合。

我们定义为

$$
\mathrm{CYC}(\mathcal{A}) = \left(\mathrm{SEQ}(\mathcal{A})\quad \{\epsilon\}\right)/\mathbf{S}
$$

其中 $\mathbf{S}$ 为等价关系，我们说 $(\alpha_1,\dots,\alpha_n)\mathbf{S}(\beta_1,\dots,\beta_n)$ 当且仅当存在任一循环移位 τ 对于所有 j 都满足 $\beta_j=\alpha_{\tau(j)}$。

” 例”

为了简便我们令 a, b 均为大小为 1 的字符，这里仅列举大小为 3 和 4 的字符串：

$$
\mathrm{CYC}(\{\mathsf{a},\mathsf{b}\})_3 = \{\mathsf{aaa}\} + \{\mathsf{aab}\} + \{\mathsf{abb}\} + \{\mathsf{bbb}\}
$$

其中 aab**S**baa**S**aba 只保留其一，同样的 abb**S**bab**S**bba 只保留其一。

$$
\mathrm{CYC}(\{\mathsf{a},\mathsf{b}\})_4 = \{\mathsf{aaaa}\} + \{\mathsf{aaab}\} + \{\mathsf{aabb}\} + \{\mathsf{abbb}\} + \{\mathsf{bbbb}\} + \{\mathsf{abab}\}
$$

其中 aaab**S**baaa**S**abaa**S**aaba，aabb**S**baab**S**bbaa**S**abba，abbb**S**babb**S**bbab**S**bbba 和 abab**S**baba。

对应 OGF 为

$$
\mathrm{Log}(A(z)) = \sum_{n\geq 1} \frac{\varphi(n)}{n} \ln \frac{1}{1-A(z^n)}
$$

其中 φ 为 Euler 函数，Log 为 Pólya 对数。

由于证明较复杂，读者可参考 Flajolet 的论文 The Cycle Construction[8] 或 Analytic Combinatorics 的附录。

有限制的构造

对于上述所有构造，我们都没有限制其「组成部分」的个数，若在 SEQ 的下标给一个作用于整数的谓词用于约束其组成部分，如

$$
\mathrm{SEQ}_{=k}(\mathcal{B}),\quad \mathrm{SEQ}_{\geq k}(\mathcal{B}),\quad \mathrm{SEQ}_{1..k}(\mathcal{B})
$$

其中 $\mathrm{SEQ}_{=k}(\mathcal{B})$ 也常简写为 $\mathrm{SEQ}_k(\mathcal{B})$，$\mathrm{SEQ}_{1..k}(\mathcal{B})$ 表示在区间 $[1..k]$ 上。

令 $\mathfrak{K}$ 为任意上述 SEQ, PSET, MSET, CYC 之一，以及

$$
\mathcal{A} = \mathfrak{K}_k(\mathcal{B})
$$

即我们需要对于 $\alpha\in\mathcal{A}$ 有

$$
\alpha = \{(\beta_1,\beta_2,\dots,\beta_k) \mid \beta\in\mathcal{B}\}
$$

设 χ 函数作用于组合对象上为其组成部分的个数，也就是要令 $\chi(\alpha)=k$，不妨增加一元来「跟踪」组成部分的个数。

令

$$A_{n,k} = \mathrm{card}\left\{\alpha \in \mathcal{A} \mid |\alpha| = n, \chi(\alpha) = k\right\}$$

那么

$$A(z,u) = \sum_{n,k} A_{n,k} u^k z^n = \sum_{\alpha \in \mathcal{A}} z^{|\alpha|} u^{\chi(\alpha)}$$

然后我们只要提取出 u^k 的系数即可获得对应表达式，例如 $\mathcal{A} = \mathrm{SEQ}_k(\mathcal{B})$ 可直接导出

$$\begin{aligned} A(z,u) &= \sum_{k \geq 0} u^k B(z)^k = \frac{1}{1 - uB(z)} \\ \Longrightarrow A(z) &= B(z)^k \end{aligned}$$

显然也有

$$\mathcal{A} = \mathrm{SEQ}_{\geq k}(\mathcal{B}) \implies A(z) = \frac{B(z)^k}{1 - B(z)}$$

而对于 $\mathrm{MSET}_k(\mathcal{B})$ 和 $\mathrm{PSET}_k(\mathcal{B})$ 已经有

$$\begin{aligned} A(z,u) &= \prod_n (1 - uz^n)^{-b_n} \\ \Longrightarrow A(z) &= [u^k] \exp\left(\frac{u}{1}B(z) + \frac{u^2}{2}B(z^2) + \frac{u^3}{3}B(z^3) + \cdots\right) \end{aligned}$$

和

$$\begin{aligned} A(z,u) &= \prod_n (1 + uz^n)^{b_n} \\ \Longrightarrow A(z) &= [u^k] \exp\left(\frac{u}{1}B(z) - \frac{u^2}{2}B(z^2) + \frac{u^3}{3}B(z^3) - \cdots\right) \end{aligned}$$

对于 $\mathrm{CYC}_k(\mathcal{B})$ 同理。

" 使用上式计算 $\mathrm{MSET}_3(\mathcal{B})$ 和 $\mathrm{MSET}_4(\mathcal{B})$ 对应 OGF"

尝试计算 $\mathcal{A} = \mathrm{MSET}_3(\mathcal{B})$ 为

$$\begin{aligned} [u^3]A(z,u) = &\frac{1}{0!}\left([u^3]1\right) + \frac{1}{1!}\left([u^3]\left(\frac{u}{1}B(z) + \frac{u^2}{2}B(z^2) + \frac{u^3}{3}B(z^3) + \cdots\right)\right) \\ &+ \frac{1}{2!}\left([u^3]\left(\frac{u}{1}B(z) + \frac{u^2}{2}B(z^2) + \cdots\right)^2\right) \\ &+ \frac{1}{3!}\left([u^3]\left(\frac{u}{1}B(z) + \cdots\right)^3\right) \\ &= \frac{B(z)^3}{6} + \frac{B(z)B(z^2)}{2} + \frac{B(z)^3}{3} \end{aligned}$$

尝试计算 $\mathcal{A} = \mathrm{MSET}_4(\mathcal{B})$ 为

$$
\begin{aligned}
[u^4]A(z,u) = & \frac{1}{0!}\left([u^4]1\right) + \frac{1}{1!}\left([u^4]\left(\frac{u}{1}B(z) + \frac{u^2}{2}B(z^2) + \frac{u^3}{3}B(z^3) + \frac{u^4}{4}B(z^4) + \cdots\right)\right) \\
& + \frac{1}{2!}\left([u^4]\left(\frac{u}{1}B(z) + \frac{u^2}{2}B(z^2) + \frac{u^3}{3}B(z^3) + \cdots\right)^2\right) \\
& + \frac{1}{3!}\left([u^4]\left(\frac{u}{1}B(z) + \frac{u^2}{2}B(z^2) + \cdots\right)^3\right) \\
& + \frac{1}{4!}\left([u^4]\left(\frac{u}{1}B(z) + \cdots\right)^4\right) \\
= & \frac{B(z^4)}{4} + \frac{1}{2!}\left(\frac{B(z^2)^2}{4} + \frac{2B(z)B(z^3)}{3}\right) + \frac{1}{3!}\left(\frac{3B(z)^2B(z^2)}{2}\right) + \frac{B(z)^4}{4!} \\
= & \frac{B(z)^4}{24} + \frac{B(z)^2B(z^2)}{4} + \frac{B(z)B(z^3)}{3} + \frac{B(z^2)^2}{8} + \frac{B(z^4)}{4}
\end{aligned}
$$

我们发现 $\mathcal{A} = \mathfrak{K}_k(\mathcal{B})$ 中 $A(z)$ 是关于 $B(z), B(z^2), \ldots, B(z^k)$ 的一个表达式。

需要注意的是对于有限制的构造 $\mathfrak{K}_k(\mathcal{B})$ 并没有要求 $\mathcal{B}_0 = \emptyset$。

" 常用有限制的构造"

$$
\begin{aligned}
\mathrm{PSET}_2(\mathcal{A}): \quad & \frac{A(z)^2}{2} - \frac{A(z^2)}{2} \\
\mathrm{MSET}_2(\mathcal{A}): \quad & \frac{A(z)^2}{2} + \frac{A(z^2)}{2} \\
\mathrm{CYC}_2(\mathcal{A}): \quad & \frac{A(z)^2}{2} + \frac{A(z^2)}{2} \\
\mathrm{PSET}_3(\mathcal{A}): \quad & \frac{A(z)^3}{6} - \frac{A(z)A(z^2)}{2} + \frac{A(z^3)}{3} \\
\mathrm{MSET}_3(\mathcal{A}): \quad & \frac{A(z)^3}{6} + \frac{A(z)A(z^2)}{2} + \frac{A(z^3)}{3} \\
\mathrm{CYC}_3(\mathcal{A}): \quad & \frac{A(z)^3}{3} + \frac{2A(z^3)}{3} \\
\mathrm{PSET}_4(\mathcal{A}): \quad & \frac{A(z)^4}{24} - \frac{A(z)^2A(z^2)}{4} + \frac{A(z)A(z^3)}{3} + \frac{A(z^2)^2}{8} - \frac{A(z^4)}{4} \\
\mathrm{MSET}_4(\mathcal{A}): \quad & \frac{A(z)^4}{24} + \frac{A(z)^2A(z^2)}{4} + \frac{A(z)A(z^3)}{3} + \frac{A(z^2)^2}{8} + \frac{A(z^4)}{4} \\
\mathrm{CYC}_4(\mathcal{A}): \quad & \frac{A(z)^4}{4} + \frac{A(z^2)^2}{4} + \frac{A(z^4)}{2}
\end{aligned}
$$

上面的计算方法虽然有效但比较麻烦，读者可阅读 WolframMathWorld 网站的 Pólya Enumeration Theorem[9] 和 Cycle Index[10] 等相关资料，后者 Cycle Index 在 OEIS 的生成函数表达式中也经常出现。

" 例题 LOJ 6538. 烷基计数　加强版　加强版[11]"

题意：求出 n 个节点的有根且根节点度数不超过 3，其余节点度数不超过 4 的无序树的个数对 998244353 取模的值。约定 $1 \le n \le 10^5$。

解：设组合类为 $\mathcal{T}$ 那么

$$\mathcal{T} = \{\bullet\} \times \mathrm{MSET}_{0,1,2,3}(\mathcal{T})$$

或令组合类 $\hat{\mathcal{T}} = \mathcal{T} + \{\epsilon\}$ 那么

$$\hat{\mathcal{T}} = \{\epsilon\} + \{\bullet\} \times \mathrm{MSET}_3(\hat{\mathcal{T}})$$

可得到相同的结果。

参考文献

- Philippe Flajolet and Robert Sedgewick.Analytic Combinatorics[12].

参考资料与注释

[1] A000108

[2] LOJ 6268. 分拆数

[3] A000041

[4] 洛谷 P4389 付公主的背包

[5] 洛谷 P5900 无标号无根树计数

[6] The Number of Trees

[7] A000055

[8] The Cycle Construction

[9] Pólya Enumeration Theorem

[10] Cycle Index

[11] LOJ 6538. 烷基计数加强版加强版

[12] Analytic Combinatorics

9.13.13 普通生成函数

Authors: sshwy

序列 a 的普通生成函数（ordinary generating function，OGF）定义为形式幂级数：

$$F(x)=\sum_n a_n x^n$$

a 既可以是有穷序列，也可以是无穷序列。常见的例子（假设 a 以 0 为起点）：

1. 序列 $a=\langle 1,2,3\rangle$ 的普通生成函数是 $1+2x+3x^2$。
2. 序列 $a=\langle 1,1,1,\cdots\rangle$ 的普通生成函数是 $\sum_{n\ge 0}x^n$。
3. 序列 $a=\langle 1,2,4,8,16,\cdots\rangle$ 的生成函数是 $\sum_{n\ge 0}2^nx^n$。
4. 序列 $a=\langle 1,3,5,7,9,\cdots\rangle$ 的生成函数是 $\sum_{n\ge 0}(2n+1)x^n$。

换句话说，如果序列 a 有通项公式，那么它的普通生成函数的系数就是通项公式。

基本运算

考虑两个序列 a, b 的普通生成函数，分别为 $F(x), G(x)$。那么有

$$F(x) \pm G(x) = \sum_n (a_n \pm b_n) x^n$$

因此 $F(x) \pm G(x)$ 是序列 $\langle a_n \pm b_n \rangle$ 的普通生成函数。

考虑乘法运算，也就是卷积：

$$F(x)G(x) = \sum_n x^n \sum_{i=0}^{n} a_i b_{n-i}$$

因此 $F(x)G(x)$ 是序列 $\langle \sum_{i=0}^{n} a_i b_{n-i} \rangle$ 的普通生成函数。

封闭形式

在运用生成函数的过程中，我们不会一直使用形式幂级数的形式，而会适时地转化为封闭形式以更好地化简。

例如 $\langle 1, 1, 1, \cdots \rangle$ 的普通生成函数 $F(x) = \sum_{n \ge 0} x^n$，我们可以发现

$$F(x)x + 1 = F(x)$$

那么解这个方程得到

$$F(x) = \frac{1}{1-x}$$

这就是 $\sum_{n \ge 0} x^n$ 的封闭形式。

考虑等比数列 $\langle 1, p, p^2, p^3, p^4, \cdots \rangle$ 的生成函数 $F(x) = \sum_{n \ge 0} p^n x^n$，有

$$\begin{aligned} F(x)px + 1 &= F(x) \\ F(x) &= \frac{1}{1-px} \end{aligned}$$

等比数列的封闭形式与展开形式是常用的变换手段。

" 小练习"

请求出下列数列的普通生成函数（形式幂级数形式和封闭形式）。难度是循序渐进的。

1. $a = \langle 0, 1, 1, 1, 1, \cdots \rangle$。
2. $a = \langle 1, 0, 1, 0, 1, \cdots \rangle$。
3. $a = \langle 1, 2, 3, 4, \cdots \rangle$。
4. $a_n = \binom{m}{n}$（m 是常数，$n \ge 0$）。
5. $a_n = \binom{m+n}{n}$（m 是常数，$n \ge 0$）。

" 答案"

第一个：

$$F(x) = \sum_{n \ge 1} x^n = \frac{x}{1-x}$$

第二个：

$$\begin{aligned} F(x) &= \sum_{n \ge 0} x^{2n} \\ &= \sum_{n \ge 0} (x^2)^n \\ &= \frac{1}{1-x^2} \end{aligned}$$

第三个（求导）：

$$\begin{aligned} F(x) &= \sum_{n\geq 0}(n+1)x^n \\ &= \sum_{n\geq 1} nx^{n-1} \\ &= \sum_{n\geq 0}(x^n)' \\ &= \left(\frac{1}{1-x}\right)' \\ &= \frac{1}{(1-x)^2} \end{aligned}$$

第四个（二项式定理）：

$$F(x) = \sum_{n\geq 0}\binom{m}{n}x^n = (1+x)^m$$

第五个：

$$F(x) = \sum_{n\geq 0}\binom{m+n}{n}x^n = \frac{1}{(1-x)^{m+1}}$$

可以使用归纳法证明。

首先当 $m=0$ 时，有 $F(x)=\dfrac{1}{1-x}$。

而当 $m>0$ 时，有

$$\begin{aligned} \frac{1}{(1-x)^{m+1}} &= \frac{1}{(1-x)^m}\frac{1}{1-x} \\ &= \left(\sum_{n\geq 0}\binom{m+n-1}{n}x^n\right)\left(\sum_{n\geq 0}x^n\right) \\ &= \sum_{n\geq 0}x^n\sum_{i=0}^{n}\binom{m+i-1}{i} \\ &= \sum_{n\geq 0}\binom{m+n}{n}x^n \end{aligned}$$

斐波那契数列的生成函数

接下来我们来推导斐波那契数列的生成函数。

斐波那契数列定义为 $a_0=0, a_1=1, a_n=a_{n-1}+a_{n-2}\ (n>1)$。设它的普通生成函数是 $F(x)$，那么根据它的递推式，我们可以类似地列出关于 $F(x)$ 的方程：

$$F(x) = xF(x) + x^2F(x) - a_0x + a_1x + a_0$$

那么解得

$$F(x) = \frac{x}{1-x-x^2}$$

那么接下来的问题是，如何求出它的展开形式？

展开方式一

不妨将 $x+x^2$ 当作一个整体，那么可以得到

$$\begin{aligned}F(x) &= \frac{x}{1-(x+x^2)}\\ &= \sum_{n\geq 0}(x+x^2)^n\\ &= \sum_{n\geq 0}\sum_{i=0}^{n}\binom{n}{i}x^{2i}x^{n-i}\\ &= \sum_{n\geq 0}\sum_{i=0}^{n}\binom{n}{i}x^{n+i}\\ &= \sum_{n\geq 0}x^n\sum_{i=0}^{n}\binom{n-i}{i}\end{aligned}$$

我们得到了 a_n 的通项公式，但那并不是我们熟知的有关黄金分割比的形式。

展开方式二

考虑求解一个待定系数的方程：

$$\frac{A}{1-ax}+\frac{B}{1-bx}=\frac{x}{1-x-x^2}$$

通分得到

$$\frac{A-Abx+B-aBx}{(1-ax)(1-bx)}=\frac{x}{1-x-x^2}$$

待定项系数相等，我们得到

$$\begin{cases}A+B=0\\ -Ab-aB=1\\ a+b=1\\ ab=-1\end{cases}$$

解得

$$\begin{cases}A=\frac{1}{\sqrt{5}}\\ B=-\frac{1}{\sqrt{5}}\\ a=\frac{1+\sqrt{5}}{2}\\ b=\frac{1-\sqrt{5}}{2}\end{cases}$$

那么我们根据等比数列的展开式，就可以得到斐波那契数列的通项公式：

$$\frac{x}{1-x-x^2}=\sum_{n\geq 0}x^n\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n-\left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

这也被称为斐波那契数列的另一个封闭形式（$\frac{x}{1-x-x^2}$ 是一个封闭形式）。

对于任意多项式 $P(x),Q(x)$，生成函数 $\dfrac{P(x)}{Q(x)}$ 的展开式都可以使用上述方法求出。在实际运用的过程中，我们往往先求出 $Q(x)$ 的根，把分母表示为 $\prod(1-p_ix)^{d_i}$ 的形式，然后再求分子。

当对分母进行因式分解但有重根时，每有一个重根就要多一个分式，如考虑生成函数

$$G(x)=\frac{1}{(1-x)(1-2x)^2}$$

的系数的通项公式，那么有

$$G(x)=\frac{c_0}{1-x}+\frac{c_1}{1-2x}+\frac{c_2}{(1-2x)^2}$$

解得

$$\begin{cases} c_0 &= 1 \\ c_1 &= -2 \\ c_2 &= 2 \end{cases}$$

那么

$$[x^n]G(x) = 1 - 2^{n+1} + (n+1) \cdot 2^{n+1}$$

牛顿二项式定理

我们重新定义组合数的运算：

$$\binom{r}{k} = \frac{r^{\underline{k}}}{k!} \quad (r \in \mathbf{C}, k \in \mathbf{N})$$

注意 r 的范围是复数域。在这种情况下。对于 $\alpha \in \mathbf{C}$，有

$$(1+x)^\alpha = \sum_{n \geq 0} \binom{\alpha}{n} x^n$$

二项式定理其实是牛顿二项式定理的一个特殊情况。

卡特兰数的生成函数

参考 Catalan 数的封闭形式

应用

接下来给出一些例题，来介绍生成函数在 OI 中的具体应用。

食物

"食物[1]"

在许多不同种类的食物中选出 n 个，每种食物的限制如下：

1. 承德汉堡：偶数个
2. 可乐：0 个或 1 个
3. 鸡腿：0 个，1 个或 2 个
4. 蜜桃多：奇数个
5. 鸡块：4 的倍数个
6. 包子：0 个，1 个，2 个或 3 个
7. 土豆片炒肉：不超过一个。
8. 面包：3 的倍数个

每种食物都是以「个」为单位，只要总数加起来是 n 就算一种方案。对于给出的 n 你需要计算出方案数，对 10007 取模。

这是一道经典的生成函数题。对于一种食物，我们可以设 a_n 表示这种食物选 n 个的方案数，并求出它的生成函数。而两种食物一共选 n 个的方案数的生成函数，就是它们生成函数的卷积。多种食物选 n 个的方案数的生成函数也是它们生成函数的卷积。

在理解了方案数可以用卷积表示以后，我们就可以构造生成函数（标号对应题目中食物的标号）：

1. $\sum_{n \geq 0} x^{2n} = \frac{1}{1-x^2}$。

2. $1+x$。
3. $1+x+x^2=\dfrac{1-x^3}{1-x}$。
4. $\dfrac{x}{1-x^2}$。
5. $\sum_{n\ge 0}x^{4n}=\dfrac{1}{1-x^4}$。
6. $1+x+x^2+x^3=\dfrac{1-x^4}{1-x}$。
7. $1+x$。
8. $\dfrac{1}{1-x^3}$。

那么全部乘起来，得到答案的生成函数：

$$F(x)=\frac{(1+x)(1-x^3)x(1-x^4)(1+x)}{(1-x^2)(1-x)(1-x^2)(1-x^4)(1-x)(1-x^3)}=\frac{x}{(1-x)^4}$$

然后将它转化为展开形式（使用封闭形式练习中第五个练习）：

$$F(x)=\sum_{n\ge 1}\binom{n+2}{n-1}x^n$$

因此答案就是 $\binom{n+2}{n-1}=\binom{n+2}{3}$。

Sweet

"「CEOI2004」Sweet[2]"

有 n 堆糖果。不同的堆里糖果的种类不同（即同一个堆里的糖果种类是相同的，不同的堆里的糖果的种类是不同的）。第 i 个堆里有 m_i 个糖果。现在要吃掉至少 a 个糖果，但不超过 b 个。求有多少种方案。

两种方案不同当且仅当吃的个数不同，或者吃的糖果中，某一种糖果的个数在两个方案中不同。

$n\le 10, 0\le a\le b\le 10^7, m_i\le 10^6$。

在第 i 堆吃 j 个糖果的方案数（显然为 1）的生成函数为

$$F_i(x)=\sum_{j=0}^{m_i}x^j=\frac{1-x^{m_i+1}}{1-x}$$

因此总共吃 i 个糖果的方案数的生成函数就是

$$G(x)=\prod_{i=1}^{n}F_i(x)=(1-x)^{-n}\prod_{i=1}^{n}(1-x^{m_i+1})$$

现在我们要求的是 $\sum_{i=a}^{b}[x^i]G(x)$。

由于 $n\le 10$，因此我们可以暴力展开 $\prod_{i=1}^{n}(1-x^{m_i+1})$（最多只有 2^n 项）。

然后对 $(1-x)^{-n}$ 使用牛顿二项式定理：

$$\begin{aligned}(1-x)^{-n}&=\sum_{i\ge 0}\binom{-n}{i}(-x)^i\\&=\sum_{i\ge 0}\binom{n-1+i}{i}x^i\end{aligned}$$

我们枚举 $\prod_{i=1}^{n}(1-x^{m_i+1})$ 中 x^k 项的系数，假设为 c_k。那么它和 $(1-x)^{-n}$ 相乘后，对答案的贡献就是

$$c_k\sum_{i=a-k}^{b-k}\binom{n-1+i}{i}=c_k\left(\binom{n+b-k}{b-k}-\binom{n+a-k-1}{a-k-1}\right)$$

这样就可以 $O(b)$ 地求出答案了。

时间复杂度 $O(2^n+b)$。

参考资料与注释

[1] 食物

[2] 「CEOI2004」Sweet

9.13.14 指数生成函数

Authors: sshwy, ComeIntoCalm

序列 a 的指数生成函数（exponential generating function，EGF）定义为形式幂级数：

$$\hat{F}(x)=\sum_n a_n\frac{x^n}{n!}$$

基本运算

指数生成函数的加减法与普通生成函数是相同的，也就是对应项系数相加。

考虑指数生成函数的乘法运算。对于两个序列 a,b，设它们的指数生成函数分别为 $\hat{F}(x),\hat{G}(x)$，那么

$$\begin{aligned}\hat{F}(x)\hat{G}(x)&=\sum_{i\ge 0}a_i\frac{x^i}{i!}\sum_{j\ge 0}b_j\frac{x^j}{j!}\\&=\sum_{n\ge 0}x^n\sum_{i=0}^{n}a_ib_{n-i}\frac{1}{i!(n-i)!}\\&=\sum_{n\ge 0}\frac{x^n}{n!}\sum_{i=0}^{n}\binom{n}{i}a_ib_{n-i}\end{aligned}$$

因此 $\hat{F}(x)\hat{G}(x)$ 是序列

$$\left\langle\sum_{i=0}^{n}\binom{n}{i}a_ib_{n-i}\right\rangle$$

的指数生成函数。

封闭形式

我们同样考虑指数生成函数的封闭形式。

序列 $\langle 1,1,1,\cdots\rangle$ 的指数生成函数是：

$$\hat{F}(x)=\sum_{n\ge 0}\frac{x^n}{n!}=\mathrm{e}^x$$

因为你将 e^x 在 $x=0$ 处泰勒展开就得到了它的无穷级数形式。

类似地，等比数列 $\langle 1,p,p^2,\cdots\rangle$ 的指数生成函数是：

$$\hat{F}(x)=\sum_{n\ge 0}\frac{p^nx^n}{n!}=\mathrm{e}^{px}$$

指数生成函数与普通生成函数

如何理解指数生成函数？我们定义序列 a 的指数生成函数是：

$$F(x)=\sum_{n\ge 0}a_n\frac{x^n}{n!}$$

但 $F(x)$ 实际上也是序列 $\left\langle\frac{a_n}{n!}\right\rangle$ 的普通生成函数。

这两种理解没有任何问题。也就是说，不同的生成函数只是对问题理解方式的转变。

EGF 中多项式 exp 的组合意义

如果您还没有学习多项式 exp 请先跳过这里，这是由 exp 理解引出的意义，从某种意义上来说可以加深对 EGF 的理解。

EGF 中 $f^n(x)$ 的 f 默认是一个 EGF，那么我们首先考虑任意两个 EGF 的乘积

$$\hat{H}(x)=\hat{F}(x)\hat{G}(x)=\sum_{n\ge 0}\left[\sum_{i=0}^{n}\binom{n}{i}f_ig_{n-i}\right]\frac{x^n}{n!}$$

对于两个 EGF 相乘得到的 $[x^k]\hat{H}(x)$，实际上是一个卷积。而如果考虑多个 EGF 相乘得到的 $[x^k]\hat{H}(x)$，实际上就是对每个 EGF 选择一项 x^{a_i} 使得 $\sum_i a_i=k$ 时每种情况系数的和。从集合的角度来理解就是把 n 个有标号元素划分为 $k>0$ 个有标号集合的方案数。

> 如果 $k=0$ 则系数显然为原 EGF 各项常数的积，但是多项式 exp 中某些要求导致 exp 的 $f(x)$ 常数项必须为 0，具体的原因在下文中会做出一些说明。

多项式系数定义（具体请参考排列组合一栏中的多项式系数组合意义）里是默认集合有序的，但是 $\exp(f(x))$ 中 $f^k(x)$，k 个 $f(x)$ 相乘得到的 EGF 相同，而集合划分显然是无序的，因此其系数应该乘上 $\frac{1}{k!}$。

设 $F_k(n)$ 为 n 个有标号元素划分成 k 个非空无序集合（因为是 exp 所以有非空要求）的情况，f_i 为 i 个元素组成一个集合时，i 个元素的有限集上特定组合结构的数量（是原有的 EGF，对这一个集合元素计数的方案，仅仅与该集合大小有关），那么 $F_k(n)$

$$F_k(n)=\frac{n!}{k!}\sum_{\sum_i^k a_i=n}\prod_{j=1}^{k}\frac{f_{a_j}}{a_j!}$$

设 f_n 的 EGF 为 $\hat{F}(x)$，即：

$$\hat{F}(x)=\sum_{n\ge 0}f_n\frac{x^n}{n!}$$

设 $F_k(n)$ 的 EGF 为 $G_k(x)$，则：

$$\begin{aligned}
G_k(x)&=\sum_{n\ge 0}F_k(n)\frac{x^n}{n!}\\
&=\sum_{n\ge 0}x^n\frac{1}{k!}\sum_{\sum_i^k a_i=n}\prod_{j=1}^{k}\frac{f_{a_j}}{a_j!}\\
&=\frac{1}{k!}\sum_{n\ge 0}\sum_{\sum_i^k a_i=n}\prod_{j=1}^{k}\frac{f_{a_j}x^{a_j}}{a_j!}\\
&=\frac{1}{k!}\hat{F}^k(x)
\end{aligned}$$

对于所有的 $k\ge 0$：

$$\sum_{k\ge 0}G_k(x)=\sum_{k\ge 0}\frac{\hat{F}^k(x)}{k!}=\exp\hat{F}(x)$$

上面是从组合角度直接列式理解，我们也可以从递推方面来证明 $\exp(f(x))$ 和 $f(x)$ 两者间的关系。

同样设 $F_k(n)$ 为 n 个有标号元素划分成 k 个非空集合（无标号）的情况，g_i 为 i 个元素组成一个集合内部的方案数（意义同上文中的 f_i），并令 $G(x)$ 为 $\{g_i\}$ 的 EGF，$H_k(x)$ 为 $\{F_k(n)\}$ 的 EGF。

n 个元素中取出 i 个元素作为一个单独划分出去的集合共有 g_i 种方案，剩下的 $n-i$ 个元素构成 $k-1$ 个集合共 $F_{k-1}(n-i)$ 种方案，但最后的划分方案中，一个方案里的每个集合都会被枚举为单独划分出去的集合，所以重复计算了 k 次，故还需要除以 k。

$$
\begin{aligned}
H_k(x) &= \sum_{n\geq 0}\frac{x^n}{n!}F_k(n) \\
&= \sum_{n\geq 0}\frac{x^n}{n!}\sum_{i=1}^{n-k+1}\binom{n}{i}F_{k-1}(n-i)\times g_i\times\frac{1}{k} \\
&= \frac{1}{k}\sum_{n\geq 0}\frac{x^n}{n!}\sum_{i=0}^{n}\binom{n}{i}F_{k-1}(n-i)\times g_i \\
&= \frac{1}{k}\cdot H_{k-1}(x)G(x)
\end{aligned}
$$

上界是由非空集合划分推出的 $n-(k-1)\geq i$（前 $k-1$ 个集合每个集合最少有一个元素），但是如果超过枚举上界涉及的 $F_{k-1}(n-i)$ 设为 0，那么就没有影响。

得到递推式之后可递归展开，边界为 $k=1$ 时 $H_1(x)=G(x)$。

$$
\begin{aligned}
H_k(x) &= \frac{1}{k}\cdot H_{k-1}(x)G(x) \\
&= \frac{1}{k}\cdot\frac{1}{k-1}\cdot H_{k-2}(x)G^2(x) \\
&= \cdots \\
&= \frac{1}{k}\cdot\frac{1}{k-1}\cdots\frac{1}{2}\cdot H_1(x)G^{k-1}(x) \\
&= \frac{1}{k!}G^k(x)
\end{aligned}
$$

同样的有：

$$\sum_{k\geq 0}H_k(x)=\sum_{k\geq 0}\frac{G^k(x)}{k!}=\exp G(x)$$

显然**定义成划分为非空集合**（$g_0=0$）是符合本身的意义的，如果**包含空集**（$g_0=1$），那么对应 $[x^n]G^k$ 中就会有 $[x^n]G^y, y>k$ 的贡献（在至少一个 G 中选择常数项），有计重，得不到所求量。

从递推式的角度讲，多个 EGF 的乘积也可以看作一个类似于背包的组合（合并两组计数对象的过程）。

总结多项式 exp 的意义就是：**有标号元素构成的集合的生成集族有多少种情况**，或划分为任意个非空子集的总方案数。

排列与圆排列

长度为 n 的排列数的指数生成函数是

$$\hat{P}(x)=\sum_{n\geq 0}\frac{n!x^n}{n!}=\sum_{n\geq 0}x^n=\frac{1}{1-x}$$

圆排列的定义是把 $1,2,\cdots,n$ 排成一个环的方案数。也就是说旋转后的方案的等价的（但翻转是不等价的）。

n 个数的圆排列数显然是 $(n-1)!$。因此 n 个数的圆排列数的指数生成函数是

$$\hat{Q}(x)=\sum_{n\geq 1}\frac{(n-1)!x^n}{n!}=\sum_{n\geq 1}\frac{x^n}{n}=-\ln(1-x)=\ln\left(\frac{1}{1-x}\right)$$

也就是说 $\exp\hat{Q}(x)=\hat{P}(x)$。但这只是数学层面的推导。我们该怎样直观理解：圆排列数的 EGF 的 exp 是排列数的 EGF？

一个排列，是由若干个置换环构成的。例如 $p=[4,3,2,5,1]$ 有两个置换环：

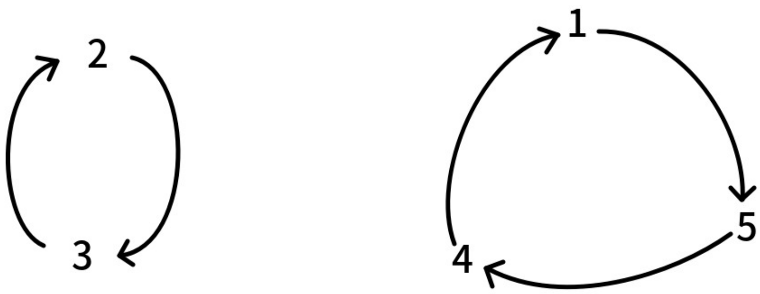


图 9.21

（也就是说我们从 p_i 向 i 连有向边）

而不同的置换环，会导出不同的排列。例如将第二个置换环改成

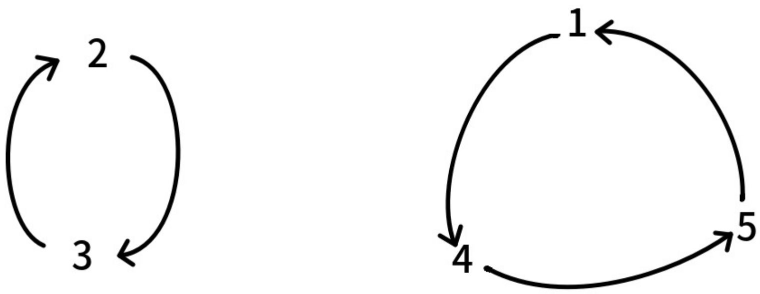


图 9.22

那么它对应的排列就是 $[5,3,2,1,4]$。

也就是说，长度为 n 的排列的方案数是

1. 把 $1,2,\cdots,n$ 分成若干个集合
2. 每个集合形成一个置换环

的方案数。而一个集合的数形成置换环的方案数显然就是这个集合大小的圆排列方案数。因此长度为 n 的排列的方案数就是：把 $1,2,\cdots,n$ 分成若干个集合，每个集合的圆排列方案数之积。

这就是多项式 exp 的直观理解。

推广之

- 如果 n 个点**带标号**生成树的 EGF 是 $\hat{F}(x)$，那么 n 个点**带标号**生成森林的 EGF 就是 $\exp\hat{F}(x)$——直观理解为，将 n 个点分成若干个集合，每个集合构成一个生成树的方案数之积。
- 如果 n 个点带标号无向连通图的 EGF 是 $\hat{F}(x)$，那么 n 个点带标号无向图的 EGF 就是 $\exp\hat{F}(x)$，后者可以很容易计算得到

$$\exp\hat{F}(x)=\sum_{n\geq 0}2^{\binom{n}{2}}\frac{x^n}{n!}$$

因此要计算前者，只需要一次多项式 ln 即可。

接下来我们来看一些指数生成函数的应用。

应用

错排数

" 错排数"

定义长度为 n 的一个错排是满足 $p_i \neq i$ 的排列。

求错排数的指数生成函数。

从置换环的角度考虑，错排就是指置换环中不存在自环的排列。也就是说不存在长度为 1 的置换环。后者的指数生成函数是

$$\sum_{n\geq 2}\frac{x^n}{n} = -\ln(1-x) - x$$

因此错排数的指数生成函数就是 $\exp(-\ln(1-x)-x)$。

不动点

"不动点[1]"

题意：求有多少个映射 $f:\{1,2,\cdots,n\}\to\{1,2,\cdots,n\}$，使得

$$\underbrace{f\circ f\circ\cdots\circ f}_{k} = \underbrace{f\circ f\circ\cdots\circ f}_{k-1}$$

$nk\leq 2\times 10^6, 1\leq k\leq 3$。

考虑 i 向 $f(i)$ 连边。相当于我们从任意一个 i 走 k 步和走 $k-1$ 步到达的是同一个点。也就是说基环树的环是自环且深度不超过 k（根结点深度为 1）。把这个基环树当成有根树是一样的。因此我们的问题转化为：n 个点带标号，深度不超过 k 的有根树森林的计数。

考虑 n 个点带标号深度不超过 k 的有根树，假设它的生成函数是：

$$\hat{F}_k(x) = \sum_{n\geq 0} f_{n,k}\frac{x^n}{n!}$$

考虑递推求 $\hat{F}_k(x)$。深度不超过 k 的有根树，实际上就是深度不超过 $k-1$ 的若干棵有根树，把它们的根结点全部连到一个结点上去。因此

$$\hat{F}_k(x) = x\exp\hat{F}_{k-1}(x)$$

那么答案的指数生成函数就是 $\exp\hat{F}_k(x)$。求它的第 n 项即可。

Lust

"Lust[2]"

给你一个 n 个数的序列 $a_1,a_2,\cdots,a_n$，和一个初值为 0 的变量 s，要求你重复以下操作 k 次：

- 在 $1,2,\cdots,n$ 中等概率随机选择一个 x。
- 令 s 加上 $\prod_{i\neq x} a_i$。
- 令 a_x 减一。

求 k 次操作后 s 的期望。

$1\leq n\leq 5000, 1\leq k\leq 10^9, 0\leq a_i\leq 10^9$。

假设 k 次操作后 a_i 减少了 b_i，那么实际上

$$s = \prod_{i=1}^{n} a_i - \prod_{i=1}^{n}(a_i - b_i)$$

因此实际上我们的问题转化为，求 k 次操作后 $\prod_{i=1}^{n}(a_i-b_i)$ 的期望。

不妨考虑计算每种方案的的 $\prod_{i=1}^{n}(a_i-b_i)$ 的和，最后除以 n^k。

而 k 次操作序列中，要使得 i 出现了 b_i 次的方案数是

$$\frac{k!}{b_1!b_2!\cdots b_n!}$$

这与指数生成函数乘法的系数类似。

设 a_j 的指数生成函数是

$$F_j(x)=\sum_{i\geq 0}(a_j-i)\frac{x^i}{i!}$$

那么答案就是

$$[x^k]\prod_{j=1}^{n}F_j(x)$$

为了快速计算答案，我们需要将 $F_j(x)$ 转化为封闭形式：

$$\begin{aligned}F_j(x)&=\sum_{i\geq 0}a_j\frac{x^i}{i!}-\sum_{i\geq 1}\frac{x^i}{(i-1)!}\\&=a_j\mathrm{e}^x-x\mathrm{e}^x\\&=(a_j-x)\mathrm{e}^x\end{aligned}$$

因此我们得到

$$\prod_{j=1}^{n}F_j(x)=\mathrm{e}^{nx}\prod_{j=1}^{n}(a_j-x)$$

其中 $\prod_{j=1}^{n}(a_j-x)$ 是一个 n 次多项式，可以暴力计算出来。假设它的展开式是 $\sum_{i=0}^{n}c_ix^i$，那么

$$\begin{aligned}\prod_{j=1}^{n}F_j(x)&=\left(\sum_{i\geq 0}\frac{n^ix^i}{i!}\right)\left(\sum_{i=0}^{n}c_ix^i\right)\\&=\sum_{i\geq 0}\sum_{j=0}^{i}c_jx^j\frac{n^{i-j}x^{i-j}}{(i-j)!}\\&=\sum_{i\geq 0}\frac{x^i}{i!}\sum_{j=0}^{i}n^{i-j}i^{\underline{j}}c_j\end{aligned}$$

计算这个多项式的 x^k 项系数即可。

参考资料与注释

[1] 不动点

[2] Lust

9.13.15 狄利克雷生成函数

Authors: sshwy

记 $\mathcal{P}$ 表示素数集合。

狄利克雷生成函数

对于无穷序列 $f_1, f_2, \ldots$，定义其狄利克雷生成函数（Dirichlet series generating function，DGF）[1-1] 为：

$$\tilde{F}(x) = \sum_{i \geq 1} \frac{f_i}{i^x}$$

如果序列 f 满足积性（是 积性函数）：$\forall i \perp j,\ f_{ij} = f_i f_j$，那么其 DGF 可以由质数幂处的取值表示：

$$\tilde{F}(x) = \prod_{p \in \mathcal{P}} \left(1 + \frac{f_p}{p^x} + \frac{f_{p^2}}{p^{2x}} + \frac{f_{p^3}}{p^{3x}} + \cdots\right)$$

对于两个序列 f, g，其 DGF 之积对应的是两者的狄利克雷卷积序列的 DGF：

$$\tilde{F}(x)\tilde{G}(x) = \sum_i \sum_j \frac{f_i g_j}{(ij)^x} = \sum_i \frac{1}{i^x} \sum_{d|i} f_d g_{\frac{i}{d}}$$

常见积性函数的 DGF

DGF 最适合用于研究与积性函数的狄利克雷卷积相关的问题。因此首先我们要了解常见积性函数的 DGF。

黎曼函数

序列 $[1, 1, 1, \ldots]$ 的 DGF 是 $\sum_{i \geq 1} \frac{1}{i^x} = \zeta(x)$。$\zeta$ 是黎曼函数。

由于其满足积性，因此我们可以得到 $[1, 1, 1, \ldots]$ 的 DGF 的另一种形式：

$$\zeta(x) = \prod_{p \in \mathcal{P}} \left(1 + \frac{1}{p^x} + \frac{1}{p^{2x}} + \ldots\right) = \prod_{p \in \mathcal{P}} \frac{1}{1 - p^{-x}}$$

莫比乌斯函数

对于莫比乌斯函数 μ，它的 DGF 定义为

$$\tilde{M}(x) = \prod_{p \in \mathcal{P}} \left(1 - \frac{1}{p^x}\right) = \prod_{p \in \mathcal{P}} (1 - p^{-x})$$

容易发现 $\zeta(x)\tilde{M}(x) = 1$，也就是说 $\tilde{M}(x) = \frac{1}{\zeta(x)}$。

欧拉函数

对于欧拉函数 φ，它的 DGF 定义为

$$\tilde{\Phi}(x) = \prod_{p \in \mathcal{P}} \left(1 + \frac{p-1}{p^x} + \frac{p(p-1)}{p^{2x}} + \frac{p^2(p-1)}{p^{3x}} + \ldots\right) = \prod_{p \in \mathcal{P}} \frac{1 - p^{-x}}{1 - p^{1-x}}$$

因此有 $\tilde{\Phi}(x) = \frac{\zeta(x-1)}{\zeta(x)}$。

幂函数

对于函数 $I_k(n) = n^k$，它的 DGF 定义为

$$\tilde{I}_k(x) = \prod_{p \in \mathcal{P}} \left(1 + \frac{p^k}{p^x} + \frac{p^{2k}}{p^{2x}} + \ldots\right) = \prod_{p \in \mathcal{P}} \frac{1}{1 - p^{k-x}} = \zeta(x-k)$$

根据这些定义，容易推导出 $\varphi * 1 = I$，$*$ 表示狄利克雷卷积。因为 $\tilde{\Phi}(x)\zeta(x) = \zeta(x-1)$。

其他函数

对于约数幂函数 $\sigma_k(n) = \sum_{d|n} d^k$，它的 DGF 可以表示为狄利克雷卷积的形式：$\tilde{S}(x) = \zeta(x-k)\zeta(x)$。

对于 $u(n) = |\mu(n)|$（无平方因子数），它的 DGF 为 $\tilde{U}(x) = \prod_{p \in \mathcal{P}} (1 + p^{-x}) = \frac{\zeta(x)}{\zeta(2x)}$。

Dirichlet 卷积

定义

对于两个数论函数 $f(x)$ 和 $g(x)$，则它们的狄利克雷卷积得到的结果 $h(x)$ 定义为：

$$h(x) = \sum_{d|x} f(d)g\left(\frac{x}{d}\right) = \sum_{ab=x} f(a)g(b)$$

上式可以简记为：

$$h = f * g$$

狄利克雷卷积是数论函数的重要运算，数论函数的许多性质都是通过这个运算挖掘出来的。

狄利克雷卷积与狄利克雷生成函数（DGF）密切相关。对于两个序列 f, g，其狄利克雷生成函数之积，对应的是两者的狄利克雷卷积序列的狄利克雷生成函数：

$$\tilde{F}(x)\tilde{G}(x) = \sum_i \sum_j \frac{f_i g_j}{(ij)^x} = \sum_i \frac{1}{i^x} \sum_{d|i} f_d g_{\frac{i}{d}}$$

性质

交换律： $f * g = g * f$。

结合律： $(f * g) * h = f * (g * h)$。

分配律： $(f + g) * h = f * h + g * h$。

等式的性质： $f = g$ 的充要条件是 $f * h = g * h$，其中数论函数 $h(x)$ 要满足 $h(1) \neq 0$。

证明： 充分性是显然的。

证明必要性，我们先假设存在 x，使得 $f(x) \neq g(y)$。那么我们找到最小的 $y \in \mathbb{N}$，满足 $f(y) \neq g(y)$，并设 $r = f * h - g * h = (f - g) * h$。

则有：

$$\begin{aligned} r(y) &= \sum_{d|y} (f(d) - g(d))h\left(\frac{y}{d}\right) \\ &= (f(y) - g(y))h(1) \\ &\neq 0 \end{aligned}$$

则 $f * h$ 和 $g * h$ 在 y 处的取值不一样，即有 $f * h \neq g * h$。矛盾，所以必要性成立。

证毕

" 注"

以上性质在狄利克雷生成函数的观点下是显然的，这种特殊的卷积等价于相应生成函数的乘法。

单位元： 单位函数 ε 是 Dirichlet 卷积运算中的单位元，即对于任何数论函数 f，都有 $f * \varepsilon = f$。

" 注"

狄利克雷卷积运算中的单位元不是常函数，但是在狄利克雷生成函数中等价于常数 1。

狄利克雷卷积运算中的数论函数常函数 1，在狄利克雷生成函数中等价于黎曼函数 ζ。

逆元： 对于任何一个满足 $f(x) \neq 0$ 的数论函数，如果有另一个数论函数 $g(x)$ 满足 $f * g = \varepsilon$，则称 $g(x)$ 是 $f(x)$ 的逆元。由**等式的性质**可知，逆元是唯一的。

" 注"

狄利克雷卷积运算中的逆元，在狄利克雷生成函数中相当于倒数运算。

容易构造出 $g(x)$ 的表达式为：

$$g(x)=\frac{\varepsilon(x)-\sum_{d|x,d\neq 1}f(d)g\left(\frac{x}{d}\right)}{f(1)}$$

重要结论

两个积性函数的 Dirichlet 卷积也是积性函数

证明： 设两个积性函数为 $f(x)$ 和 $g(x)$，再记 $h=f*g$。

设 $\gcd(a,b)=1$，则：

$$h(a)=\sum_{d_1|a}f(d_1)g\left(\frac{a}{d_1}\right),h(b)=\sum_{d_2|b}f(d_2)g\left(\frac{b}{d_2}\right),$$

所以：

$$\begin{aligned}h(a)h(b)&=\sum_{d_1|a}f(d_1)g\left(\frac{a}{d_1}\right)\sum_{d_2|b}f(d_2)g\left(\frac{b}{d_2}\right)\\&=\sum_{d|ab}f(d)g\left(\frac{ab}{d}\right)\\&=h(ab)\end{aligned}$$

所以结论成立。

证毕

积性函数的逆元也是积性函数

证明：我们设 $f*g=\varepsilon$，并且不妨设 $f(1)=1$。考虑归纳法：

- 若 $nm=1$，则 $g(nm)=g(1)=1$，结论显然成立；
- 若 $nm>1(\gcd(n,m)=1)$，假设现在对于所有的 $xy<nm(\gcd(x,y)=1)$，都有 $g(xy)=g(x)g(y)$，所以有：

$$g(nm)=-\sum_{d|nm,d\neq 1}f(d)g\left(\frac{nm}{d}\right)=-\sum_{a|n,b|m,ab\neq 1}f(ab)g\left(\frac{nm}{ab}\right)$$

又因为 $\frac{nm}{ab}<nm$，所以有：

$$\begin{aligned}g(nm)&=-\sum_{a|n,b|m,ab\neq 1}f(ab)g\left(\frac{nm}{ab}\right)\\&=-\sum_{a|n,b|m,ab\neq 1}f(a)f(b)g\left(\frac{n}{a}\right)g\left(\frac{m}{b}\right)\\&=f(1)f(1)g(n)g(m)-\sum_{a|n,b|m}f(a)f(b)g\left(\frac{n}{a}\right)g\left(\frac{m}{b}\right)\\&=g(n)g(m)-\sum_{a|n}f(a)g\left(\frac{n}{a}\right)\sum_{b|m}f(b)g\left(\frac{m}{b}\right)\\&=g(n)g(m)-\varepsilon(n)-\varepsilon(m)\\&=g(n)g(m)\end{aligned}$$

综合以上两点，结论成立。

证毕

" 注"

这也说明，数论函数的积性，在狄利克雷生成函数中的对应具有封闭性。

例子

$$\varepsilon = \mu * 1 \iff \varepsilon(n) = \sum_{d|n} \mu(d)$$
$$d = 1 * 1 \iff d(n) = \sum_{d|n} 1$$
$$\sigma = \mathrm{id} * 1 \iff \sigma(n) = \sum_{d|n} d$$
$$\varphi = \mu * \mathrm{id} \iff \varphi(n) = \sum_{d|n} d \cdot \mu(\frac{n}{d})$$

相关应用

DGF 的应用主要体现在构造积性序列的狄利克雷卷积序列。研究方向通常是质数处的取值。

例如在杜教筛的过程中，要计算积性序列（积性函数在正整数处的取值构成的序列）f 的前缀和，我们需要找到一个积性序列 g 使得 $f * g$ 和 g 都可以快速求前缀和。那么我们可以利用 DGF 推导这一过程。

以 洛谷 P3768 简单的数学题 为例，我们要对 $f_i = i^2\varphi(i)$ 构造一个满足上述条件的积性序列 g。由于 f 是积性的，考虑其 DGF

$$\tilde{F}(x) = \prod_{p\in\mathcal{P}} \left(1 + \sum_{k\ge 1} \frac{p^{3k-1}(p-1)}{p^{kx}}\right) = \prod_{p\in\mathcal{P}} \frac{1-p^{2-x}}{1-p^{3-x}} = \frac{\zeta(x-3)}{\zeta(x-2)}$$

因此 $\tilde{F}(x)\zeta(x-2) = \zeta(x-3)$。而 $\zeta(x-2)$ 对应的积性函数为 I_2，所以令 $g = I_2$ 即可。这样有 $f * g = I_3$，两者都是可以快速计算前缀和的。

[1] https://en.wikipedia.org/wiki/Generating_function#Dirichlet_series_generating_functions_(DGFs) [1-1] [1-2]

9.14 组合数学

9.14.1 排列组合

引入

排列组合是组合数学中的基础。排列就是指从给定个数的元素中取出指定个数的元素进行排序；组合则是指从给定个数的元素中仅仅取出指定个数的元素，不考虑排序。排列组合的中心问题是研究给定要求的排列和组合可能出现的情况总数。排列组合与古典概率论关系密切。

在高中初等数学中，排列组合多是利用列表、枚举等方法解题。

加法 & 乘法原理

加法原理

完成一个工程可以有 n 类办法，$a_i(1 \le i \le n)$ 代表第 i 类方法的数目。那么完成这件事共有 $S = a_1 + a_2 + \cdots + a_n$ 种不同的方法。

乘法原理

完成一个工程需要分 n 个步骤，$a_i(1 \le i \le n)$ 代表第 i 个步骤的不同方法数目。那么完成这件事共有 $S = a_1 \times a_2 \times \cdots \times a_n$ 种不同的方法。

排列与组合基础

排列数

从 n 个不同元素中，任取 m（$m \le n$，m 与 n 均为自然数，下同）个元素按照一定的顺序排成一列，叫做从 n 个不同元素中取出 m 个元素的一个排列；从 n 个不同元素中取出 $m(m \le n)$ 个元素的所有排列的个数，叫做从 n 个不同元素中取出 m 个元素的排列数，用符号 A_n^m（或者是 P_n^m）表示。

排列的计算公式如下：

$$\mathrm{A}_n^m = n(n-1)(n-2)\cdots(n-m+1) = \frac{n!}{(n-m)!}$$

$n!$ 代表 n 的阶乘，即 $6! = 1 \times 2 \times 3 \times 4 \times 5 \times 6$。

公式可以这样理解：n 个人选 m 个来排队 $(m \le n)$。第一个位置可以选 n 个，第二位置可以选 $n-1$ 个，以此类推，第 m 个（最后一个）可以选 $n-m+1$ 个，得：

$$\mathrm{A}_n^m = n(n-1)(n-2)\cdots(n-m+1) = \frac{n!}{(n-m)!}$$

全排列：n 个人全部来排队，队长为 n。第一个位置可以选 n 个，第二位置可以选 $n-1$ 个，以此类推得：

$$\mathrm{A}_n^n = n(n-1)(n-2)\cdots 3 \times 2 \times 1 = n!$$

全排列是排列数的一个特殊情况。

组合数

从 n 个不同元素中，任取 $m \le n$ 个元素组成一个集合，叫做从 n 个不同元素中取出 m 个元素的一个组合；从 n 个不同元素中取出 $m \le n$ 个元素的所有组合的个数，叫做从 n 个不同元素中取出 m 个元素的组合数，用符号 $\dbinom{n}{m}$ 来表示，读作「n 选 m」。

组合数计算公式

$$\binom{n}{m} = \frac{\mathrm{A}_n^m}{m!} = \frac{n!}{m!(n-m)!}$$

如何理解上述公式？我们考虑 n 个人选 m 个出来（$m \le n$），不排队，不在乎顺序。如果在乎顺序那么就是 A_n^m，如果不在乎那么就要除掉重复，那么重复了多少？同样选出来的 m 个人，他们还要「全排」得 $m!$，所以得：

$$\begin{aligned} \binom{n}{m} \times m! &= \mathrm{A}_n^m \\ \binom{n}{m} &= \frac{\mathrm{A}_n^m}{m!} = \frac{n!}{m!(n-m)!} \end{aligned}$$

组合数也常用 C_n^m 表示，即 $\mathrm{C}_n^m = \dbinom{n}{m}$。现在数学界普遍采用 $\dbinom{n}{m}$ 的记号而非 C_n^m。

组合数也被称为「二项式系数」，下文二项式定理将会阐述其中的联系。

特别地，规定当 $m > n$ 时，$\mathrm{A}_n^m = \dbinom{n}{m} = 0$。

插板法

插板法（Stars and bars）是用于求一类给相同元素分组的方案数的一种技巧，也可以用于求一类线性不定方程的解的组数。

正整数和的数目

问题一：现有 n 个**完全相同**的元素，要求将其分为 k 组，保证每组至少有一个元素，一共有多少种分法？

考虑拿 $k-1$ 块板子插入到 n 个元素两两形成的 $n-1$ 个空里面。

因为元素是完全相同的，所以答案就是 $\binom{n-1}{k-1}$。

本质是求 $x_1+x_2+\cdots+x_k=n$ 的正整数解的组数。

非负整数和的数目

问题二：如果问题变化一下，每组允许为空呢？

显然此时没法直接插板了，因为有可能出现很多块板子插到一个空里面的情况，非常不好计算。

我们考虑创造条件转化成有限制的问题一，先借 k 个元素过来，在这 $n+k$ 个元素形成的 $n+k-1$ 个空里面插板，答案为

$$\binom{n+k-1}{k-1}=\binom{n+k-1}{n}$$

虽然不是直接求的原问题，但这个式子就是原问题的答案，可以这么理解：

开头我们借来了 k 个元素，用于保证每组至少有一个元素，插完板之后再把这 k 个借来的元素从 k 组里面拿走。因为元素是相同的，所以转化过的情况和转化前的情况可以一一对应，答案也就是相等的。

由此可以推导出插板法的公式：$\binom{n+k-1}{n}$。

本质是求 $x_1+x_2+\cdots+x_k=n$ 的非负整数解的组数（即要求 $x_i\geq 0$）。

不同下界整数和的数目

问题三：如果再扩展一步，要求对于第 i 组，至少要分到 $a_i, \sum a_i\leq n$ 个元素呢？

本质是求 $x_1+x_2+\cdots+x_k=n$ 的解的数目，其中 $x_i\geq a_i$。

类比无限制的情况，我们借 $\sum a_i$ 个元素过来，保证第 i 组至少能分到 a_i 个。也就是令

$$x_i'=x_i-a_i$$

得到新方程：

$$\begin{aligned}(x_1'+a_1)+(x_2'+a_2)+\cdots+(x_k'+a_k)&=n\\ x_1'+x_2'+\cdots+x_k'&=n-a_1-a_2-\cdots-a_k\\ x_1'+x_2'+\cdots+x_k'&=n-\sum a_i\end{aligned}$$

其中

$$x_i'\geq 0$$

然后问题三就转化成了问题二，直接用插板法公式得到答案为

$$\binom{n-\sum a_i+k-1}{n-\sum a_i}$$

不相邻的排列

$1\sim n$ 这 n 个自然数中选 k 个，这 k 个数中任何两个数都不相邻的组合有 $\binom{n-k+1}{k}$ 种。

二项式定理

在进入排列组合进阶篇之前，我们先介绍一个与组合数密切相关的定理——二项式定理。

二项式定理阐明了一个展开式的系数：

$$(a+b)^n=\sum_{i=0}^{n}\binom{n}{i}a^{n-i}b^i$$

证明可以采用数学归纳法，利用 $\binom{n}{k}+\binom{n}{k-1}=\binom{n+1}{k}$ 做归纳。

二项式定理也可以很容易扩展为多项式的形式：

设 n 为正整数，x_i 为实数，

$$(x_1+x_2+\cdots+x_t)^n=\sum_{n_1+\cdots+n_t=n}\binom{n}{n_1,n_2,\cdots,n_t}x_1^{n_1}x_2^{n_2}\cdots x_t^{n_t}$$

其中的 $\binom{n}{n_1,n_2,\cdots,n_t}$ 是多项式系数，它的性质也很相似：

$$\sum\binom{n}{n_1,n_2,\cdots,n_t}=t^n$$

排列与组合进阶篇

接下来我们介绍一些排列组合的变种。

多重集的排列数 | 多重组合数

请大家一定要区分**多重组合数**与**多重集的组合数**！两者是完全不同的概念！

多重集是指包含重复元素的广义集合。设 $S=\{n_1\cdot a_1,n_2\cdot a_2,\cdots,n_k\cdot a_k\}$ 表示由 n_1 个 a_1，n_2 个 a_2，$\cdots$，n_k 个 a_k 组成的多重集，S 的全排列个数为

$$\frac{n!}{\prod_{i=1}^k n_i!}=\frac{n!}{n_1!n_2!\cdots n_k!}$$

相当于把相同元素的排列数除掉了。具体地，你可以认为你有 k 种不一样的球，每种球的个数分别是 $n_1,n_2,\cdots,n_k$，且 $n=n_1+n_2+\ldots+n_k$。这 n 个球的全排列数就是**多重集的排列数**。多重集的排列数常被称作**多重组合数**。我们可以用多重组合数的符号表示上式：

$$\binom{n}{n_1,n_2,\cdots,n_k}=\frac{n!}{\prod_{i=1}^k n_i!}$$

可以看出，$\binom{n}{m}$ 等价于 $\binom{n}{m,n-m}$，只不过后者较为繁琐，因而不采用。

多重集的组合数 1

设 $S=\{n_1\cdot a_1,n_2\cdot a_2,\cdots,n_k\cdot a_k\}$ 表示由 n_1 个 a_1，n_2 个 a_2，$\cdots$，n_k 个 a_k 组成的多重集。那么对于整数 $r(r<n_i,\forall i\in[1,k])$，从 S 中选择 r 个元素组成一个多重集的方案数就是**多重集的组合数**。这个问题等价于 $x_1+x_2+\cdots+x_k=r$ 的非负整数解的数目，可以用插板法解决，答案为

$$\binom{r+k-1}{k-1}$$

多重集的组合数 2

考虑这个问题：设 $S=\{n_1\cdot a_1,n_2\cdot a_2,\cdots,n_k\cdot a_k,\}$ 表示由 n_1 个 a_1，n_2 个 a_2，$\cdots$，n_k 个 a_k 组成的多重集。那么对于正整数 r，从 S 中选择 r 个元素组成一个多重集的方案数。

这样就限制了每种元素的取的个数。同样的，我们可以把这个问题转化为带限制的线性方程求解：

$$\forall i\in[1,k],\ x_i\le n_i,\ \sum_{i=1}^k x_i=r$$

于是很自然地想到了容斥原理。容斥的模型如下：

1. 全集：$\sum_{i=1}^k x_i=r$ 的非负整数解。
2. 属性：$x_i\le n_i$。

于是设满足属性 i 的集合是 S_i，$\overline{S_i}$ 表示不满足属性 i 的集合，即满足 $x_i \geq n_i + 1$ 的集合（转化为上面插板法的问题三）。那么答案即为

$$\left|\bigcap_{i=1}^{k} S_i\right| = |U| - \left|\bigcup_{i=1}^{k} \overline{S_i}\right|$$

根据容斥原理，有：

$$\begin{aligned}\left|\bigcup_{i=1}^{k} \overline{S_i}\right| =& \sum_i |\overline{S_i}| - \sum_{i,j} |\overline{S_i} \cap \overline{S_j}| + \sum_{i,j,k} |\overline{S_i} \cap \overline{S_j} \cap \overline{S_k}| - \cdots \\ &+ (-1)^{k-1} \left|\bigcap_{i=1}^{k} \overline{S_i}\right| \\ =& \sum_i \binom{k+r-n_i-2}{k-1} - \sum_{i,j} \binom{k+r-n_i-n_j-3}{k-1} + \sum_{i,j,k} \binom{k+r-n_i-n_j-n_k-4}{k-1} - \cdots \\ &+ (-1)^{k-1} \binom{k+r-\sum_{i=1}^{k} n_i - k - 1}{k-1}\end{aligned}$$

拿全集 $|U| = \binom{k+r-1}{k-1}$ 减去上式，得到多重集的组合数

$$Ans = \sum_{p=0}^{k} (-1)^p \sum_A \binom{k+r-1-\sum_A n_{A_i} - p}{k-1}$$

其中 A 是充当枚举子集的作用，满足 $|A| = p$，$A_i < A_{i+1}$。

圆排列

n 个人全部来围成一圈，所有的排列数记为 Q_n^n。考虑其中已经排好的一圈，从不同位置断开，又变成不同的队列。所以有

$$\mathrm{Q}_n^n \times n = \mathrm{A}_n^n \Longrightarrow \mathrm{Q}_n = \frac{\mathrm{A}_n^n}{n} = (n-1)!$$

由此可知部分圆排列的公式：

$$\mathrm{Q}_n^r = \frac{\mathrm{A}_n^r}{r} = \frac{n!}{r \times (n-r)!}$$

组合数性质｜二项式推论

由于组合数在 OI 中十分重要，因此在此介绍一些组合数的性质。

$$\binom{n}{m} = \binom{n}{n-m} \tag{1}$$

相当于将选出的集合对全集取补集，故数值不变。（对称性）

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1} \tag{2}$$

由定义导出的递推式。

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1} \tag{3}$$

组合数的递推式（杨辉三角的公式表达）。我们可以利用这个式子，在 $O(n^2)$ 的复杂度下推导组合数。

$$\binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{n} = \sum_{i=0}^{n} \binom{n}{i} = 2^n \tag{4}$$

这是二项式定理的特殊情况。取 $a = b = 1$ 就得到上式。

$$\sum_{i=0}^{n} (-1)^i \binom{n}{i} = [n = 0] \tag{5}$$

二项式定理的另一种特殊情况，可取 $a=1,b=-1$。式子的特殊情况是取 $n=0$ 时答案为 1。

$$\sum_{i=0}^{m}\binom{n}{i}\binom{m}{m-i}=\binom{m+n}{m}\quad(n\geq m)\tag{6}$$

拆组合数的式子，在处理某些数据结构题时会用到。

$$\sum_{i=0}^{n}\binom{n}{i}^2=\binom{2n}{n}\tag{7}$$

这是 (6) 的特殊情况，取 $n=m$ 即可。

$$\sum_{i=0}^{n}i\binom{n}{i}=n2^{n-1}\tag{8}$$

带权和的一个式子，通过对 (3) 对应的多项式函数求导可以得证。

$$\sum_{i=0}^{n}i^2\binom{n}{i}=n(n+1)2^{n-2}\tag{9}$$

与上式类似，可以通过对多项式函数求导证明。

$$\sum_{l=0}^{n}\binom{l}{k}=\binom{n+1}{k+1}\tag{10}$$

通过组合分析——考虑 $S=\{a_1,a_2,\cdots,a_{n+1}\}$ 的 $k+1$ 子集数可以得证，在恒等式证明中比较常用。

$$\binom{n}{r}\binom{r}{k}=\binom{n}{k}\binom{n-k}{r-k}\tag{11}$$

通过定义可以证明。

$$\sum_{i=0}^{n}\binom{n-i}{i}=F_{n+1}\tag{12}$$

其中 F 是斐波那契数列。

二项式反演

记 f_n 表示恰好使用 n 个不同元素形成特定结构的方案数，g_n 表示从 n 个不同元素中选出 $i\geq 0$ 个元素形成特定结构的总方案数。

若已知 f_n 求 g_n，那么显然有：

$$g_n=\sum_{i=0}^{n}\binom{n}{i}f_i$$

若已知 g_n 求 f_n，那么：

$$f_n=\sum_{i=0}^{n}\binom{n}{i}(-1)^{n-i}g_i$$

上述已知 g_n 求 f_n 的过程，就称为**二项式反演**。

证明

将反演公式的 g_i 展开得到：

$$\begin{aligned}f_n&=\sum_{i=0}^{n}\binom{n}{i}(-1)^{n-i}\left[\sum_{j=0}^{i}\binom{i}{j}f_j\right]\\&=\sum_{i=0}^{n}\sum_{j=0}^{i}\binom{n}{i}\binom{i}{j}(-1)^{n-i}f_j\end{aligned}$$

先枚举 j，再枚举 i，得到：

$$\begin{aligned} f_n &= \sum_{j=0}^{n}\sum_{i=j}^{n}\binom{n}{i}\binom{i}{j}(-1)^{n-i}f_j \\ &= \sum_{j=0}^{n} f_j \sum_{i=j}^{n}\binom{n}{i}\binom{i}{j}(-1)^{n-i} \end{aligned}$$

使用 「 组合数性质 ｜ 二项式推论 」 的公式 (11) 得到：

$$\begin{aligned} f_n &= \sum_{j=0}^{n} f_j \sum_{i=j}^{n}\binom{n}{j}\binom{n-j}{i-j}(-1)^{n-i} \\ &= \sum_{j=0}^{n}\binom{n}{j} f_j \sum_{i=j}^{n}\binom{n-j}{i-j}(-1)^{n-i} \end{aligned}$$

令 $k=i-j$。则 $i=k+j$，上式转换为：

$$f_n = \sum_{j=0}^{n}\binom{n}{j} f_j \sum_{k=0}^{n-j}\binom{n-j}{k}(-1)^{n-j-k}1^k$$

使用 「 组合数性质 ｜ 二项式推论 」 的公式 (5) 得到：

$$f_n = \sum_{j=0}^{n}\binom{n}{j} f_j[n=j] = f_n$$

证毕。

9.14.2 抽屉原理

定义

抽屉原理，亦称鸽巢原理（the pigeonhole principle）。

它常被用于证明存在性证明和求最坏情况下的解。

简单情况

将 $n+1$ 个物体，划分为 n 组，那么有至少一组有两个（或以上）的物体。

这个定理看起来比较显然，证明方法考虑反证法：假如每个分组有至多 1 个物体，那么最多有 $1\times n$ 个物体，而实际上有 $n+1$ 个物体，矛盾。

推广

将 n 个物体，划分为 k 组，那么至少存在一个分组，含有大于或等于 $\left\lceil \frac{n}{k} \right\rceil$ 个物品。

推广的形式也可以使用反证法证明：若每个分组含有小于 $\left\lceil \frac{n}{k} \right\rceil$ 个物体，则其总和 $S \le (\left\lceil \frac{n}{k} \right\rceil - 1)\times k = k\left\lceil \frac{n}{k} \right\rceil - k < k(\frac{n}{k}+1) - k = n$ 矛盾。

此外，划分还可以弱化为覆盖结论不变。

给定集合 S，一个 S 的非空子集构成的簇 $\{A_1, A_2 \dots A_k\}$

- 若满足 $\bigcup_{i=1}^{k} A_i$ 则称为 S 的一个覆盖（cover）
- 若一个覆盖还满足 $i\neq j \to A_i\cap A_j=\emptyset$ 则称为 S 的一个划分。

鸽巢原理可以有如下叙述：对于 S 的一个覆盖 $\{A_1, A_2 \dots A_k\}$ 有至少一个集合 A_i 满足 $|A_i| \ge \left\lceil \frac{|S|}{k} \right\rceil$。

参考文献

- Wikipedia: Pigeonhole principle[1]
- *Discrete Mathematics and Its Applications*: Chapter 6, Section 1

参考资料与注释

[1] Wikipedia: Pigeonhole principle

9.14.3 容斥原理

引入

" 入门例题"

假设班里有 10 个学生喜欢数学，15 个学生喜欢语文，21 个学生喜欢编程，班里至少喜欢一门学科的有多少个学生呢?

是 $10+15+21=46$ 个吗? 不是的，因为有些学生可能同时喜欢数学和语文，或者语文和编程，甚至还有可能三者都喜欢。

为了叙述方便，我们把喜欢语文、数学、编程的学生集合分别用 A,B,C 表示，则学生总数等于 $|A\cup B\cup C|$。刚才已经讲过，如果把这三个集合的元素个数 $|A|,|B|,|C|$ 直接加起来，会有一些元素重复统计了，因此需要扣掉 $|A\cap B|,|B\cap C|,|C\cap A|$，但这样一来，又有一小部分多扣了，需要加回来，即 $|A\cap B\cap C|$。即

$$|A\cup B\cup C| = |A|+|B|+|C|-|A\cap B|-|B\cap C|-|C\cap A|+|A\cap B\cap C|$$

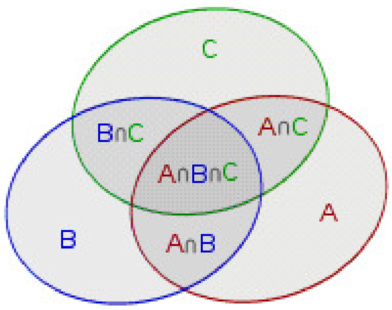


图 9.23 容斥原理 - venn 图示例

把上述问题推广到一般情况，就是我们熟知的容斥原理。

定义

设 U 中元素有 n 种不同的属性，而第 i 种属性称为 P_i，拥有属性 P_i 的元素构成集合 S_i，那么

$$\begin{aligned}\left|\bigcup_{i=1}^{n} S_i\right| = &\sum_i |S_i| - \sum_{i<j} |S_i\cap S_j| + \sum_{i<j<k} |S_i\cap S_j\cap S_k| - \cdots \\ &+ (-1)^{m-1}\sum_{a_i<a_{i+1}}\left|\bigcap_{i=1}^{m} S_{a_i}\right| + \cdots + (-1)^{n-1}|S_1\cap\cdots\cap S_n|\end{aligned}$$

即

$$\left|\bigcup_{i=1}^{n} S_i\right| = \sum_{m=1}^{n}(-1)^{m-1}\sum_{a_i<a_{i+1}}\left|\bigcap_{i=1}^{m} S_{a_i}\right|$$

证明

对于每个元素使用二项式定理计算其出现的次数。对于元素 x，假设它出现在 $T_1, T_2, \cdots, T_m$ 的集合中，那么它的出现次数为

$$\begin{aligned}
Cnt =& |\{T_i\}| - |\{T_i \cap T_j | i < j\}| + \cdots + (-1)^{k-1}\left|\left\{\bigcap_{i=1}^{k} T_{a_i} | a_i < a_{i+1}\right\}\right| \\
&+ \cdots + (-1)^{m-1}|\{T_1 \cap \cdots \cap T_m\}| \\
=& \binom{m}{1} - \binom{m}{2} + \cdots + (-1)^{m-1}\binom{m}{m} \\
=& \binom{m}{0} - \sum_{i=0}^{m}(-1)^i\binom{m}{i} \\
=& 1 - (1-1)^m = 1
\end{aligned}$$

于是每个元素出现的次数为 1，那么合并起来就是并集。证毕。

补集

对于全集 U 下的**集合的并**可以使用容斥原理计算，而集合的交则用全集减去**补集的并集**求得：

$$\left|\bigcap_{i=1}^{n} S_i\right| = |U| - \left|\bigcup_{i=1}^{n} \overline{S_i}\right|$$

右边使用容斥即可。

可能接触过容斥的读者都清楚上述内容，而更关心的是容斥的应用

那么接下来我们给出 3 个层次不同的例题来为大家展示容斥原理的应用。

不定方程非负整数解计数

" 不定方程非负整数解计数"

给出不定方程 $\sum_{i=1}^{n} x_i = m$ 和 n 个限制条件 $x_i \le b_i$，其中 $m, b_i \in \mathbb{N}$. 求方程的非负整数解的个数。

没有限制时

如果没有 $x_i < b_i$ 的限制，那么不定方程 $\sum_{i=1}^{n} x_i = m$ 的非负整数解的数目为 $\binom{m+n-1}{n-1}$.

略证：插板法。

相当于你有 m 个球要分给 n 个盒子，允许某个盒子是空的。这个问题不能直接用组合数解决。

于是我们再加入 $n-1$ 个球，于是问题就变成了在一个长度为 $m+n-1$ 的球序列中选择 $n-1$ 个球，然后这个 $n-1$ 个球把这个序列隔成了 n 份，恰好可以一一对应放到 n 个盒子中。那么在 $m+n-1$ 个球中选择 $n-1$ 个球的方案数就是 $\binom{m+n-1}{n-1}$。

容斥模型

接着我们尝试抽象出容斥原理的模型：

1. 全集 U：不定方程 $\sum_{i=1}^{n} x_i = m$ 的非负整数解
2. 元素：变量 x_i.
3. 属性：x_i 的属性即 x_i 满足的条件，即 $x_i \le b_i$ 的条件

目标：所有变量满足对应属性时集合的大小，即 $|\bigcap_{i=1}^{n} S_i|$.

这个东西可以用 $|\bigcap_{i=1}^{n} S_i| = |U| - |\bigcup_{i=1}^{n} \overline{S_i}|$ 求解。$|U|$ 可以用组合数计算，后半部分自然使用容斥原理展开。

那么问题变成，对于一些 $\overline{S_{a_i}}$ 的交集求大小。考虑 $\overline{S_{a_i}}$ 的含义，表示 $x_{a_i} \ge b_{a_i}+1$ 的解的数目。而交集表示同时满足这些条件。因此这个交集对应的不定方程中，有些变量有**下界限制**，而有些则没有限制。

能否消除这些下界限制呢？既然要求的是非负整数解，而有些变量的下界又大于 0，那么我们直接**把这个下界减掉**，就可以使得这些变量的下界变成 0，即没有下界啦。因此对于

$$\left|\bigcap_{a_i<a_{i+1}}^{1\le i\le k} S_{a_i}\right|$$

的不定方程形式为

$$\sum_{i=1}^{n} x_i = m - \sum_{i=1}^{k}(b_{a_i}+1)$$

于是这个也可以组合数计算啦。这个长度为 k 的 a 数组相当于在枚举子集。

HAOI2008 硬币购物

"HAOI2008 硬币购物"

4 种面值的硬币，第 i 种的面值是 C_i。n 次询问，每次询问给出每种硬币的数量 D_i 和一个价格 S，问付款方式。

$n \le 10^3, S \le 10^5$.

如果用背包做的话复杂度是 $O(4nS)$，无法承受。这道题最明显的特点就是硬币一共只有四种。抽象模型，其实就是让我们求方程 $\sum_{i=1}^{4} C_i x_i = S, x_i \le D_i$ 的非负整数解的个数。

采用同样的容斥方式，x_i 的属性为 $x_i \le D_i$. 套用容斥原理的公式，最后我们要求解

$$\sum_{i=1}^{4} C_i x_i = S - \sum_{i=1}^{k} C_{a_i}(D_{a_i}+1)$$

也就是无限背包问题。这个问题可以预处理，算上询问，总复杂度 $O(4S + 2^4 n)$。

" 代码实现"

```
#include <bits/stdc++.h>
using namespace std;
const long long S = 1e5 + 5;
long long c[5], d[5], n, s;
long long f[S];

int main() {
  scanf("%lld%lld%lld%lld%lld", &c[1], &c[2], &c[3], &c[4], &n);
  f[0] = 1;
  for (long long j = 1; j <= 4; j++)
    for (long long i = 1; i < S; i++)
      if (i >= c[j]) f[i] += f[i - c[j]];  // f[i]: 价格为 i 时的硬币组成方法数
  for (long long k = 1; k <= n; k++) {
    scanf("%lld%lld%lld%lld%lld", &d[1], &d[2], &d[3], &d[4], &s);
    long long ans = 0;
    for (long long i = 1; i < 16;
         i++) {  // 容斥，因为物品一共有 4 种，所以从 1 到 2^4-1=15 循环
      long long m = s, bit = 0;
      for (long long j = 1; j <= 4; j++) {
        if ((i >> (j - 1)) % 2 == 1) {
          m -= (d[j] + 1) * c[j];
```

```
            bit++;
        }
      }
      if (m >= 0) ans += (bit % 2 * 2 - 1) * f[m];
    }
    printf("%lld\n", f[s] - ans);
  }
  return 0;
}
```

完全图子图染色问题

前面的三道题都是容斥原理的正向运用，这道题则需要用到容斥原理逆向分析。

" 完全图子图染色问题"

A 和 B 喜欢对图（不一定连通）进行染色，而他们的规则是，相邻的结点必须染同一种颜色。今天 A 和 B 玩游戏，对于 n 阶**完全图** $G=(V,E)$。他们定义一个估价函数 $F(S)$，其中 S 是边集，$S \subseteq E.F(S)$ 的值是对图 $G'=(V,S)$ 用 m 种颜色染色的总方案数。他们的另一个规则是，如果 $|S|$ 是奇数，那么 A 的得分增加 $F(S)$，否则 B 的得分增加 $F(S)$. 问 A 和 B 的得分差值。

数学形式

一看这道题的算法趋向并不明显，因此对于棘手的题目首先抽象出数学形式。得分差即为奇偶对称差，可以用 -1 的幂次来作为系数。我们求的是

$$Ans=\sum_{S\subseteq E}(-1)^{|S|-1}F(S)$$

容斥模型

相邻结点染同一种颜色，我们把它当作属性。在这里我们先不遵守染色的规则，假定我们用 m 种颜色直接对图染色。对于图 $G'=(V,S)$，我们把它当作**元素**。**属性** $x_i=x_j$ 的含义是结点 i,j 染同色（注意，并未要求 i,j 之间有连边）。

而属性 $x_i=x_j$ 对应的**集合**定义为 $Q_{i,j}$，其含义是所有满足该属性的图 G' 的染色方案，集合的大小就是满足该属性的染色方案数，集合内的元素相当于所有满足该属性的图 G' 的染色图。

回到题目，「相邻的结点必须染同一种颜色」，可以理解为若干个 Q 集合的交集。因此可以写出

$$F(S)=\left|\bigcap_{(i,j)\in S}Q_{i,j}\right|$$

上述式子右边的含义就是说对于 S 内的每一条边 (i,j) 都满足 $x_i=x_j$ 的染色方案数，也就是 $F(S)$.

是不是很有容斥的味道了？由于容斥原理本身没有二元组的形式，因此我们把**所有**的边 (i,j) 映射到 $T=\frac{n(n+1)}{2}$ 个整数上，假设将 (i,j) 映射为 $k, 1\le k\le T$，同时 $Q_{i,j}$ 映射为 Q_k. 那么属性 $x_i=x_j$ 则定义为 P_k.

同时 S 可以表示为若干个 k 组成的集合，即 $S\iff K=\{k_1,k_2,\cdots,k_m\}$.（也就是说我们在边集与数集间建立了等价关系）。

而 E 对应集合 $M=\left\{1,2,\cdots,\frac{n(n+1)}{2}\right\}$. 于是乎

$$F(S)\iff F(\{k_i\})=\left|\bigcap_{k_i}Q_{k_i}\right|$$

逆向分析

那么要求的式子展开

$$\begin{aligned}Ans &= \sum_{K\subseteq M}(-1)^{|K|-1}\left|\bigcap_{k_i\in K}Q_{k_i}\right| \\ &= \sum_i |Q_i| - \sum_{i<j}|Q_i\cap Q_j| + \sum_{i<j<k}|Q_i\cap Q_j\cap Q_k| - \cdots + (-1)^{T-1}\left|\bigcap_{i=1}^{T}Q_i\right|\end{aligned}$$

于是就出现了容斥原理的展开形式，因此对这个式子逆向推导

$$Ans = \left|\bigcup_{i=1}^{T}Q_i\right|$$

再考虑等式右边的含义，只要满足 $1\sim T$ 任一条件即可，也就是存在两个点同色（不一定相邻）的染色方案数！而我们知道染色方案的全集是 U，显然 $|U|=m^n$. 而转化为补集，就是求两两异色的染色方案数，即 $A_m^n=\frac{m!}{n!}$. 因此

$$Ans = m^n - A_m^n$$

解决这道题，我们首先抽象出题目数学形式，然后从题目中信息量最大的条件，$F(S)$ 函数的定义入手，将其转化为集合的交并补。然后将式子转化为容斥原理的形式，并**逆向推导**出最终的结果。这道题体现的正是容斥原理的逆用。

数论中的容斥

使用容斥原理能够巧妙地求解一些数论问题。

容斥原理求最大公约数为 k 的数对个数

考虑下面的问题：

" 求最大公约数为 k 的数对个数"

设 $1\le x,y\le N$，$f(k)$ 表示最大公约数为 k 的有序数对 (x,y) 的个数，求 $f(1)$ 到 $f(N)$ 的值。

这道题固然可以用欧拉函数或莫比乌斯反演的方法来做，但是都不如用容斥原理来的简单。

由容斥原理可以得知，先找到所有以 k 为**公约数**的数对，再从中剔除所有以 k 的倍数为**公约数**的数对，余下的数对就是以 k 为**最大公约数**的数对。即 $f(k)=$ 以 k 为**公约数**的数对个数 $-$ 以 k 的倍数为**公约数**的数对个数。

进一步可发现，以 k 的倍数为**公约数**的数对个数等于所有以 k 的倍数为**最大公约数**的数对个数之和。于是，可以写出如下表达式：

$$f(k) = \lfloor (N/k)\rfloor^2 - \sum_{i=2}^{i*k\le N} f(i*k)$$

由于当 $k>N/2$ 时，我们可以直接算出 $f(k)=\lfloor (N/k)\rfloor^2$，因此我们可以倒过来，从 $f(N)$ 算到 $f(1)$ 就可以了。于是，我们使用容斥原理完成了本题。

```
for (long long k = N; k >= 1; k--) {
  f[k] = (N / k) * (N / k);
  for (long long i = k + k; i <= N; i += k) f[k] -= f[i];
}
```

上述方法的时间复杂度为 $O(\sum_{i=1}^{N}N/i)=O(N\sum_{i=1}^{N}1/i)=O(N\log N)$。

附赠三倍经验供大家练手。

- Luogu P2398 GCD SUM[1]
- Luogu P2158[SDOI2008] 仪仗队[2]
- Luogu P1447[NOI2010] 能量采集[3]

容斥原理推导欧拉函数

考虑下面的问题:

" 欧拉函数公式"

求欧拉函数 $\varphi(n)$。其中 $\varphi(n) = |\{1 \le x \le n | \gcd(x, n) = 1\}|$。

直接计算是 $O(n \log n)$ 的，用线性筛是 $O(n)$ 的，杜教筛是 $O(n^{\frac{2}{3}})$ 的（话说一道数论入门题用容斥做为什么还要扯到杜教筛上），接下来考虑用容斥推出欧拉函数的公式

判断两个数是否互质，首先分解质因数

$$n = \prod_{i=1}^{k} {p_i}^{c_i}$$

那么就要求对于任意 p_i，x 都不是 p_i 的倍数，即 $p_i \nmid x$. 把它当作属性，对应的集合为 S_i，因此有

$$\varphi(n) = \left|\bigcap_{i=1}^{k} S_i\right| = |U| - \left|\bigcup_{i=1}^{k} \overline{S_i}\right|$$

全集大小 $|U| = n$，而 $\overline{S_i}$ 表示的是 $p_i \mid x$ 构成的集合，显然 $|\overline{S_i}| = \frac{n}{p_i}$，并由此推出

$$\left|\bigcap_{a_i < a_{i+1}} S_{a_i}\right| = \frac{n}{\prod p_{a_i}}$$

因此可得

$$\begin{aligned}
\varphi(n) &= n - \sum_i \frac{n}{p_i} + \sum_{i<j} \frac{n}{p_i p_j} - \cdots + (-1)^k \frac{n}{p_1 p_2 \cdots p_n} \\
&= n\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\cdots\left(1 - \frac{1}{p_k}\right) \\
&= n \prod_{i=1}^{k}\left(1 - \frac{1}{p_i}\right)
\end{aligned}$$

这就是欧拉函数的数学表示啦

容斥原理一般化

容斥原理常用于集合的计数问题，而对于两个集合的函数 $f(S), g(S)$，若

$$f(S) = \sum_{T \subseteq S} g(T)$$

那么就有

$$g(S) = \sum_{T \subseteq S} (-1)^{|S|-|T|} f(T)$$

证明

接下来我们简单证明一下。我们从等式的右边开始推:

$$\begin{aligned}
&\sum_{T \subseteq S} (-1)^{|S|-|T|} f(T) \\
=&\sum_{T \subseteq S} (-1)^{|S|-|T|} \sum_{Q \subseteq T} g(Q) \\
=&\sum_{Q} g(Q) \sum_{Q \subseteq T \subseteq S} (-1)^{|S|-|T|}
\end{aligned}$$

我们发现后半部分的求和与 Q 无关，因此把后半部分的 Q 剔除:

$$= \sum_{Q} g(Q) \sum_{T \subseteq (S\ Q)} (-1)^{|S\ Q|-|T|}$$

记关于集合 P 的函数 $F(P)=\sum_{T\subseteq P}(-1)^{|P|-|T|}$，并化简这个函数：

$$
\begin{aligned}
F(P) &= \sum_{T\subseteq P}(-1)^{|P|-|T|} \\
&= \sum_{i=0}^{|P|}\binom{|P|}{i}(-1)^{|P|-i} = \sum_{i=0}^{|P|}\binom{|P|}{i}1^{i}(-1)^{|P|-i} \\
&= (1-1)^{|P|} = 0^{|P|}
\end{aligned}
$$

因此原来的式子的值是

$$
\sum_{Q} g(Q) \sum_{T\subseteq(S\ \ Q)}(-1)^{|S\ \ Q|-|T|} = \sum_{Q} g(Q)F(S\ \ Q) = \sum_{Q} g(Q)\cdot 0^{|S\ \ Q|}
$$

分析发现，仅当 $|S\ \ Q|=0$ 时有 $0^0=1$，这时 $Q=S$，对答案的贡献就是 $g(S)$，其他时侯 $0^{|S\ \ Q|}=0$，则对答案无贡献。于是得到

$$
\sum_{Q} g(Q)\cdot 0^{|S\ \ Q|} = g(S)
$$

综上所述，得证。

推论

该形式还有这样一个推论。在全集 U 下，对于函数 $f(S),g(S)$，如果

$$
f(S)=\sum_{S\subseteq T} g(T)
$$

那么

$$
g(S)=\sum_{S\subseteq T}(-1)^{|T|-|S|}f(T)
$$

这个推论其实就是补集形式，证法类似。

DAG 计数

"DAG 计数"

对 n 个点带标号的有向无环图进行计数，对 10^9+7 取模。$n\le 5\times 10^3$。

直接 DP

考虑 DP，定义 $f[i,j]$ 表示 i 个点的 DAG，有 j 点个入度为 0 的图的个数。假设去掉这 j 个点后，有 k 个点入度为 0，那么在去掉前这 k 个点至少与这 j 个点中的某几个有连边，即 2^j-1 种情况；而这 j 个点除了与 k 个点连边，还可以与剩下的点任意连边，有 2^{i-j-k} 种情况。因此方程如下：

$$
f[i,j]=\binom{i}{j}\sum_{k=1}^{i-j}(2^j-1)^k 2^{(i-j-k)j} f[i-j,k]
$$

计算上式的复杂度是 $O(n^3)$ 的。

放宽限制

上述 DP 的定义是恰好 j 个点入度为 0, 太过于严格，可以放宽为至少 j 个点入度为 0。直接定义 $f[i]$ 表示 i 个点的 DAG 个数。可以直接容斥。考虑选出的 j 个点，这 j 个点可以和剩下的 $i-j$ 个点有任意的连边，即 $\left(2^{i-j}\right)^j=2^{(i-j)j}$ 种情况：

$$
f[i]=\sum_{j=1}^{i}(-1)^{j-1}\binom{i}{j}2^{(i-j)j}f[i-j]
$$

计算上式的复杂度是 $O(n^2)$ 的。

Min-max 容斥

对于满足 全序 关系并且其中元素满足可加减性的序列 $\{x_i\}$，设其长度为 n，并设 $S=\{1,2,3,\cdots,n\}$，则有：

$$\max_{i\in S} x_i=\sum_{T\subseteq S}(-1)^{|T|-1}\min_{j\in T} x_j$$

$$\min_{i\in S} x_i=\sum_{T\subseteq S}(-1)^{|T|-1}\max_{j\in T} x_j$$

证明： 考虑做一个到一般容斥原理的映射。对于 $x\in S$，假设 x 是第 k 大的元素。那么我们定义一个映射 $f:x\mapsto\{1,2,\cdots,k\}$。显然这是一个双射。

那么容易发现，对于 $x,y\in S$，$f(\min(x,y))=f(x)\cap f(y)$，$f(\max(x,y))=f(x)\cup f(y)$。因此我们得到：

$$\begin{aligned}\left|f\left(\max_{i\in S} x_i\right)\right| &= \left|\bigcup_{i\in S} f(x_i)\right|\\ &= \sum_{T\subseteq S}(-1)^{|T|-1}\left|\bigcap_{j\in T} f(x_j)\right|\\ &= \sum_{T\subseteq S}(-1)^{|T|-1}\left|f\left(\min_{j\in T} x_j\right)\right|\end{aligned}$$

然后再把 $|f(\max_{i\in S} x_i)|$ 映射回 $\max_{i\in S} x_i$，而 min 是类似的。

证毕

但是你可能觉得这个式子非常蠢，最大值明明可以直接求。之所以 min-max 容斥这么重要，是因为它在期望上也是成立的，即：

$$E\left(\max_{i\in S} x_i\right)=\sum_{T\subseteq S}(-1)^{|T|-1}E\left(\min_{j\in T} x_j\right)$$

$$E\left(\min_{i\in S} x_i\right)=\sum_{T\subseteq S}(-1)^{|T|-1}E\left(\max_{j\in T} x_j\right)$$

证明： 我们考虑计算期望的一种方法：

$$E\left(\max_{i\in S} x_i\right)=\sum_{y}P(y=x)\max_{j\in S} y_j$$

其中 y 是一个长度为 n 的序列。

我们对后面的 max 使用之前的式子：

$$\begin{aligned}E\left(\max_{i\in S} x_i\right) &= \sum_{y}P(y=x)\max_{j\in S} y_j\\ &= \sum_{y}P(y=x)\sum_{T\subseteq S}(-1)^{|T|-1}\min_{j\in T} y_j\end{aligned}$$

调换求和顺序：

$$\begin{aligned}E\left(\max_{i\in S} x_i\right) &= \sum_{y}P(y=x)\sum_{T\subseteq S}(-1)^{|T|-1}\min_{j\in T} y_j\\ &= \sum_{T\subseteq S}(-1)^{|T|-1}\sum_{y}P(y=x)\min_{j\in T} y_j\\ &= \sum_{T\subseteq S}(-1)^{|T|-1}E\left(\min_{j\in T} y_j\right)\end{aligned}$$

min 是类似的。

证毕

还有更强的：

$$\operatorname*{kthmax}_{i\in S} x_i = \sum_{T\subseteq S} (-1)^{|T|-k} \binom{|T|-1}{k-1} \min_{j\in T} x_j$$

$$\operatorname*{kthmin}_{i\in S} x_i = \sum_{T\subseteq S} (-1)^{|T|-k} \binom{|T|-1}{k-1} \max_{j\in T} x_j$$

$$E\left(\operatorname*{kthmax}_{i\in S} x_i\right) = \sum_{T\subseteq S} (-1)^{|T|-k} \binom{|T|-1}{k-1} E\left(\min_{j\in T} x_j\right)$$

$$E\left(\operatorname*{kthmin}_{i\in S} x_i\right) = \sum_{T\subseteq S} (-1)^{|T|-k} \binom{|T|-1}{k-1} E\left(\max_{j\in T} x_j\right)$$

规定若 $n < m$，则 $\binom{n}{m} = 0$。

证明：不妨设 $\forall 1 \le i < n, x_i \le x_{i+1}$。则有：

$$\begin{aligned}\sum_{T\subseteq S} (-1)^{|T|-k} \binom{|T|-1}{k-1} \min_{j\in T} x_j &= \sum_{i\in S} x_i \sum_{T\subseteq S} (-1)^{|T|-k} \binom{|T|-1}{k-1} \left[x_i = \min_{j\in T} x_j\right] \\ &= \sum_{i\in S} x_i \sum_{j=k}^{n} \binom{n-i}{j-1} \binom{j-1}{k-1} (-1)^{j-k}\end{aligned}$$

又因为有组合恒等式：$\binom{a}{b}\binom{b}{c} = \binom{a}{c}\binom{a-c}{b-c}$，所以有：

$$\begin{aligned}\sum_{T\subseteq S} (-1)^{|T|-k} \binom{|T|-1}{k-1} \min_{j\in T} x_j &= \sum_{i\in S} x_i \sum_{j=k}^{n} \binom{n-i}{j-1} \binom{j-1}{k-1} (-1)^{j-k} \\ &= \sum_{i\in S} x_i \sum_{j=k}^{n} \binom{n-i}{k-1} \binom{n-i-k+1}{j-k} (-1)^{j-k} \\ &= \sum_{i\in S} \binom{n-i}{k-1} x_i \sum_{j=k}^{n} \binom{n-i-k+1}{j-k} (-1)^{j-k} \\ &= \sum_{i\in S} \binom{n-i}{k-1} x_i \sum_{j=0}^{n-i-k+1} \binom{n-i-k+1}{j} (-1)^{j}\end{aligned}$$

当 $i = n-k+1$ 时：

$$\binom{n-i}{k-1} \sum_{j=0}^{n-i-k+1} \binom{n-i-k+1}{j} (-1)^{j} = 1$$

否则：

$$\binom{n-i}{k-1} \sum_{j=0}^{n-i-k+1} \binom{n-i-k+1}{j} (-1)^{j} = 0$$

所以：

$$\sum_{i\in S} \binom{n-i}{k-1} x_i \sum_{j=0}^{n-i-k+1} \binom{n-i-k+1}{j} (-1)^{j} = \operatorname*{kthmax}_{i\in S} x_i$$

剩下三个是类似的。

证毕

根据 min-max 容斥，我们还可以得到下面的式子：

$$\operatorname*{lcm}_{i\in S} x_i = \prod_{T\subseteq S} \left(\gcd_{j\in T} x_j\right)^{(-1)^{|T|-1}}$$

因为 $\operatorname{lcm}, \gcd, a^1, a^{-1}$ 分别相当于 $\max, \min, +, -$，就是说相当于对于指数做了一个 min-max 容斥，自然就是对的了

PKUWC2018 随机游走

"PKUWC2018 随机游走[4]"

给定一棵 n 个点的树，你从 x 出发，每次等概率随机选择一条与所在点相邻的边走过去。

有 Q 次询问。每次询问给出一个集合 S，求如果从 x 出发一直随机游走，直到点集 S 中的点都至少经过一次的话，期望游走几步。

特别地，点 x（即起点）视为一开始就被经过了一次。

对 998244353 取模。

$1 \le n \le 18, 1 \le Q \le 5000, 1 \le |S| \le n$。

期望游走的步数也就是游走的时间。那么设随机变量 x_i 表示第一次走到结点 i 的时间。那么我们要求的就是

$$E\left(\max_{i \in S} x_i\right)$$

使用 min-max 容斥可以得到

$$E\left(\max_{i \in S} x_i\right) = E\left(\sum_{T \subseteq S} (-1)^{|T|-1} \min_{i \in T} x_i\right) = \sum_{T \subseteq S} (-1)^{|T|-1} E\left(\min_{i \in T} x_i\right)$$

对于一个集合 $T \in [n]$，考虑求出 $F(T) = E(\min_{i \in T} x_i)$。

考虑 $E(\min_{i \in T} x_i)$ 的含义，是第一次走到 T 中某一个点的期望时间。不妨设 $f(i)$ 表示从结点 i 出发，第一次走到 T 中某个结点的期望时间。

- 对于 $i \in T$，有 $f(i) = 0$。
- 对于 $i \notin T$，有 $f(i) = 1 + \frac{1}{\deg(i)} \sum_{(i,j) \in E} f(j)$。

如果直接高斯消元，复杂度 $O(n^3)$。那么我们对每个 T 都计算 $F(T)$ 的总复杂度就是 $O(2^n n^3)$，不能接受。我们使用树上消元的技巧。

不妨设根结点是 1，结点 u 的父亲是 p_u。对于叶子结点 i，$f(i)$ 只会和 i 的父亲有关（也可能 $f(i) = 0$，那样更好）。因此我们可以把 $f(i)$ 表示成 $f(i) = A_i + B_i f(p_i)$ 的形式，其中 A_i, B_i 可以快速计算。

对于非叶结点 i，考虑它的儿子序列 $j_1, \cdots, j_k$。由于 $f(j_e) = A_{j_e} + B_{j_e} f(i)$。因此可以得到

$$f(i) = 1 + \frac{1}{\deg(i)} \sum_{e=1}^{k} \left(A_{j_e} + B_{j_e} f(i)\right) + \frac{f(p_i)}{\deg(i)}$$

那么变换一下可以得到

$$f(i) = \frac{\deg(i) + \sum_{e=1}^{k} A_{j_e}}{\deg(i) - \sum_{e=1}^{k} B_{j_e}} + \frac{f(p_i)}{\deg(i) - \sum_{e=1}^{k} B_{j_e}}$$

于是我们把 $f(i)$ 也写成了 $A_i + B_i f(p_i)$ 的形式。这样可以一直倒推到根结点。而根结点没有父亲。也就是说

$$f(1) = \frac{\deg(1) + \sum_{e=1}^{k} A_{j_e}}{\deg(1) - \sum_{e=1}^{k} B_{j_e}}$$

解一下这个方程我们就得到了 $f(1)$，再从上往下推一次就得到了每个点的 $f(i)$。那么 $F(T) = f(x)$。时间复杂度 $O(n)$。

这样，我们可以对于每一个 T 计算出 $F(T)$，时间复杂度 $O(2^n n)$。

回到容斥的部分，我们知道 $E(\max_{i \in S} x_i) = \sum_{T \subseteq S} (-1)^{|T|-1} F(T)$。

不妨设 $F'(T) = (-1)^{|T|-1} F(T)$，那么进一步得到 $E(\max_{i \in S} x_i) = \sum_{T \subseteq S} F'(T)$。因此可以使用 FMT（也叫子集前缀和，或者 FWT 或变换）在 $O(2^n n)$ 的时间内对每个 S 计算出 $E(\max_{i \in S} x_i)$，这样就可以 $O(1)$ 回答询问了。

习题

- ABC331- G - Collect Them All[5]
- 洛谷 P4707 重返现世[6]

参考文献

王迪《容斥原理》，2013 年信息学奥林匹克中国国家队候选队员论文集

Cyhlnj《 有标号的 DAG 计数系列问题 》[7]

Wikipedia - 全序关系[8]

参考资料与注释

[1] Luogu P2398 GCD SUM

[2] Luogu P2158[SDOI2008] 仪仗队

[3] Luogu P1447[NOI2010] 能量采集

[4] PKUWC2018 随机游走

[5] ABC331- G - Collect Them All

[6] 洛谷 P4707 重返现世

[7] Cyhlnj《有标号的 DAG 计数系列问题》

[8] Wikipedia - 全序关系

9.14.4 康托展开

康托展开可以用来求一个 $1 \sim n$ 的任意排列的排名。

什么是排列的排名？

把 $1 \sim n$ 的所有排列按字典序排序，这个排列的位次就是它的排名。

时间复杂度？

康托展开可以在 $O(n^2)$ 的复杂度内求出一个排列的排名，在用到树状数组优化时可以做到 $O(n \log n)$。

怎么实现？

因为排列是按字典序排名的，因此越靠前的数字优先级越高。也就是说如果两个排列的某一位之前的数字都相同，那么如果这一位如果不相同，就按这一位排序。

比如 4 的排列，$[2,3,1,4] < [2,3,4,1]$，因为在第 3 位出现不同，则 $[2,3,1,4]$ 的排名在 $[2,3,4,1]$ 前面。

举个栗子

我们知道长为 5 的排列 $[2,5,3,4,1]$ 大于以 1 为第一位的任何排列，以 1 为第一位的 5 的排列有 $4!$ 种。这是非常好理解的。但是我们对第二位的 5 而言，它大于**第一位与这个排列相同的，而这一位比** 5 **小的**所有排列。不过我们要注意的是，这一位不仅要比 5 小，还要满足没有在当前排列的前面出现过，不然统计就重复了。因此这一位为 $1,3$ 或 4，第一位为 2 的所有排列都比它要小，数量为 $3\times 3!$。

按照这样统计下去，答案就是 $1+4!+3\times 3!+2!+1=46$。注意我们统计的是排名，因此最前面要 $+1$。

注意到我们每次要用到**当前有多少个小于它的数还没有出现**，这里用树状数组统计比它小的数出现过的次数就可以了。

逆康托展开

因为排列的排名和排列是一一对应的，所以康托展开满足双射关系，是可逆的。可以通过类似上面的过程倒推回来。

如果我们知道一个排列的排名，就可以推出这个排列。因为 $4!$ 是严格大于 $3\times 3!+2\times 2!+1\times 1!$ 的，所以可以认为对于长度为 5 的排列，排名 x 除以 $4!$ 向下取整就是有多少个数小于这个排列的第一位。

引用上面展开的例子

首先让 $46-1=45$，45 代表着有多少个排列比这个排列小。$\lfloor\frac{45}{4!}\rfloor=1$，有一个数小于它，所以第一位是 2。

此时让排名减去 $1\times 4!$ 得到 21，$\lfloor\frac{21}{3!}\rfloor=3$，有 3 个数小于它，去掉已经存在的 2，这一位是 5。

$21-3\times 3!=3$，$\lfloor\frac{3}{2!}\rfloor=1$，有一个数小于它，那么这一位就是 3。

让 $3-1\times 2!=1$，有一个数小于它，这一位是剩下来的第二位，4，剩下一位就是 1。即 $[2,5,3,4,1]$。

实际上我们得到了形如**有两个数小于它**这一结论，就知道它是当前第 3 个没有被选上的数，这里也可以用线段树维护，时间复杂度为 $O(n\log n)$。

9.14.5 斐波那契数列

斐波那契数列（The Fibonacci sequence，OEIS A000045[1]）的定义如下：

$$F_0=0,F_1=1,F_n=F_{n-1}+F_{n-2}$$

该数列的前几项如下：

$$0,1,1,2,3,5,8,13,21,34,55,89,\ldots$$

卢卡斯数列

卢卡斯数列（The Lucas sequence，OEIS A000032[2]）的定义如下：

$$L_0=2,L_1=1,L_n=L_{n-1}+L_{n-2}$$

该数列的前几项如下：

$$2,1,3,4,7,11,18,29,47,76,123,199,\ldots$$

研究斐波那契数列，很多时候需要借助卢卡斯数列为工具。

斐波那契数列通项公式

第 n 个斐波那契数可以在 $\Theta(n)$ 的时间内使用递推公式计算。但我们仍有更快速的方法计算。

解析解

解析解即公式解。我们有斐波那契数列的通项公式（Binet's Formula）：

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

这个公式可以很容易地用归纳法证明，当然也可以通过生成函数的概念推导，或者解一个方程得到。

当然你可能发现，这个公式分子的第二项总是小于 1，并且它以指数级的速度减小。因此我们可以把这个公式写成

$$F_n = \left[\frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}}\right]$$

这里的中括号表示取离它最近的整数。

这两个公式在计算的时候要求极高的精确度，因此在实践中很少用到。但是请不要忽视！结合模意义下二次剩余和逆元的概念，在 OI 中使用这个公式仍是有用的。

卢卡斯数列通项公式

我们有卢卡斯数列的通项公式：

$$L_n = \left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n$$

与斐波那契数列非常相似。事实上有：

$$\frac{L_n + F_n\sqrt{5}}{2} = \left(\frac{1+\sqrt{5}}{2}\right)^n$$

也就是说，L_n 和 F_n 恰好构成 $\left(\frac{1+\sqrt{5}}{2}\right)^n$ 二项式展开再合并同类项后的分子系数。也就是说，Pell 方程

$$x^2 - 5y^2 = -4$$

的全体解，恰好是

$$\frac{x_n + y_n\sqrt{5}}{2} = \frac{L_n + F_n\sqrt{5}}{2}$$

恰好是卢卡斯数列和斐波那契数列。因此有

$${L_n}^2 - 5{F_n}^2 = -4$$

矩阵形式

斐波那契数列的递推可以用矩阵乘法的形式表达：

$$\begin{bmatrix} F_{n-1} & F_n \end{bmatrix} = \begin{bmatrix} F_{n-2} & F_{n-1} \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

设 $P = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$，我们得到

$$\begin{bmatrix} F_n & F_{n+1} \end{bmatrix} = \begin{bmatrix} F_0 & F_1 \end{bmatrix} P^n$$

于是我们可以用矩阵乘法在 $\Theta(\log n)$ 的时间内计算斐波那契数列。此外，前一节讲述的公式也可通过矩阵对角化的技巧来得到。

快速倍增法

使用上面的方法我们可以得到以下等式：

$$F_{2k} = F_k(2F_{k+1} - F_k)$$
$$F_{2k+1} = F_{k+1}^2 + F_k^2$$

于是可以通过这样的方法快速计算两个相邻的斐波那契数（常数比矩乘小）。代码如下，返回值是一个二元组 (F_n, F_{n+1})。

```
pair<int, int> fib(int n) {
  if (n == 0) return {0, 1};
  auto p = fib(n >> 1);
  int c = p.first * (2 * p.second - p.first);
  int d = p.first * p.first + p.second * p.second;
  if (n & 1)
    return {d, c + d};
  else
    return {c, d};
}
```

性质

斐波那契数列拥有许多有趣的性质，这里列举出一部分简单的性质：

1. 卡西尼性质（Cassini's identity）：$F_{n-1}F_{n+1} - F_n^2 = (-1)^n$。
2. 附加性质：$F_{n+k} = F_k F_{n+1} + F_{k-1} F_n$。
3. 取上一条性质中 $k = n$，我们得到 $F_{2n} = F_n(F_{n+1} + F_{n-1})$。
4. 由上一条性质可以归纳证明，$\forall k \in \mathbb{N}, F_n | F_{nk}$。
5. 上述性质可逆，即 $\forall F_a | F_b, a | b$。
6. GCD 性质：$(F_m, F_n) = F_{(m,n)}$。
7. 以斐波那契数列相邻两项作为输入会使欧几里德算法达到最坏复杂度（具体参见 维基 - 拉梅[3]）。

斐波那契数列与卢卡斯数列的关系

不难发现，关于卢卡斯数列与斐波那契数列的等式，与三角函数公式具有很高的相似性。比如：

$$\frac{L_n + F_n\sqrt{5}}{2} = \left(\frac{1+\sqrt{5}}{2}\right)^n$$

与

$$\cos nx + i \sin nx = (\cos x + i \sin x)^n$$

很像。以及

$$L_n{}^2 - 5F_n{}^2 = -4$$

与

$$\cos^2 x + \sin^2 x = 1$$

很像。因此，卢卡斯数列与余弦函数很像，而斐波那契数列与正弦函数很像。比如，根据

$$\left(\frac{1+\sqrt{5}}{2}\right)^m \left(\frac{1+\sqrt{5}}{2}\right)^n = \left(\frac{1+\sqrt{5}}{2}\right)^{m+n}$$

可以得到两下标之和的等式：

$$2L_{m+n} = 5F_mF_n + L_mL_n$$

$$2F_{m+n} = F_mL_n + L_mF_n$$

于是推论就有二倍下标的等式：

$$L_{2n} = {L_n}^2 - 2(-1)^n$$

$$F_{2n} = F_nL_n$$

这也是一种快速倍增下标的办法。同样地，也可以仿照三角函数的公式，比如奇偶性、和差化积、积化和差、半角、万能代换等等，推理出更多有关卢卡斯数列与斐波那契数列的相应等式。

斐波那契编码

我们可以利用斐波那契数列为正整数编码。根据 齐肯多夫定理[4]，任何自然数 n 可以被唯一地表示成一些斐波那契数的和：

$$N = F_{k_1} + F_{k_2} + ... + F_{k_r}$$

并且 $k_1 \geq k_2 + 2$，$k_2 \geq k_3 + 2$，…，$k_r \geq 2$（即不能使用两个相邻的斐波那契数）

于是我们可以用 $d_0d_1d_2 \dots d_s1$ 的编码表示一个正整数，其中 $d_i = 1$ 则表示 F_{i+2} 被使用。编码末位我们强制给它加一个 1（这样会出现两个相邻的 1），表示这一串编码结束。举几个例子：

$$\begin{aligned}
1 = && 1 = && F_2 = && (11)_F \\
2 = && 2 = && F_3 = && (011)_F \\
6 = && 5+1 = && F_5 + F_2 = && (10011)_F \\
8 = && 8 = && F_6 = && (000011)_F \\
9 = && 8+1 = && F_6 + F_2 = && (100011)_F \\
19 = && 13+5+1 = && F_7 + F_5 + F_2 = && (1001011)_F
\end{aligned}$$

给 n 编码的过程可以使用贪心算法解决：

1. 从大到小枚举斐波那契数 F_i，直到 $F_i \leq n$。
2. 把 n 减掉 F_i，在编码的 $i-2$ 的位置上放一个 1（编码从左到右以 0 为起点）。
3. 如果 n 为正，回到步骤 1。
4. 最后在编码末位添加一个 1，表示编码的结束位置。

解码过程同理，先删掉末位的 1，对于编码为 1 的位置 i（编码从左到右以 0 为起点），累加一个 F_{i+2} 到答案。最后的答案就是原数字。

模意义下周期性

考虑模 p 意义下的斐波那契数列，可以容易地使用抽屉原理证明，该数列是有周期性的。考虑模意义下前 p^2+1 个斐波那契数对（两个相邻数配对）：

$$(F_1,\ F_2),\ (F_2,\ F_3),\ \dots,\ (F_{p^2+1},\ F_{p^2+2})$$

p 的剩余系大小为 p，意味着在前 p^2+1 个数对中必有两个相同的数对，于是这两个数对可以往后生成相同的斐波那契数列，那么他们就是周期性的。

皮萨诺周期

模 m 意义下斐波那契数列的最小正周期被称为 皮萨诺周期[5]（Pisano periods,OEIS A001175[6]）。

皮萨诺周期总是不超过 $6m$，且只有在满足 $m = 2 \times 5^k$ 的形式时才取到等号。

当需要计算第 n 项斐波那契数模 m 的值的时候，如果 n 非常大，就需要计算斐波那契数模 m 的周期。当然，只需要计算周期，不一定是最小正周期。

容易验证，斐波那契数模 2 的最小正周期是 3，模 5 的最小正周期是 20。

显然，如果 a 与 b 互素，ab 的皮萨诺周期就是 a 的皮萨诺周期与 b 的皮萨诺周期的最小公倍数。

计算周期还需要以下结论：

结论 1：对于奇素数 $p \equiv 1, 4 \pmod 5$，$p-1$ 是斐波那契数模 p 的周期。即，奇素数 p 的皮萨诺周期整除 $p-1$。

证明：

此时 $5^{\frac{p-1}{2}} \equiv 1 \pmod p$。

由二项式展开：

$$F_p = \frac{2}{2^p\sqrt{5}}\left(\binom{p}{1}\sqrt{5} + \binom{p}{3}\sqrt{5}^3 + \ldots + \binom{p}{p}\sqrt{5}^p\right) \equiv \sqrt{5}^{p-1} \equiv 1 \pmod p$$

$$F_{p+1} = \frac{2}{2^{p+1}\sqrt{5}}\left(\binom{p+1}{1}\sqrt{5} + \binom{p+1}{3}\sqrt{5}^3 + \ldots + \binom{p+1}{p}\sqrt{5}^p\right) \equiv \frac{1}{2}\left(1 + \sqrt{5}^{p-1}\right) \equiv 1 \pmod p$$

因为 F_p 和 F_{p+1} 两项都同余于 1，与 F_1 和 F_2 一致，所以 $p-1$ 是周期。

结论 2：对于奇素数 $p \equiv 2, 3 \pmod 5$，$2p+2$ 是斐波那契数模 p 的周期。即，奇素数 p 的皮萨诺周期整除 $2p+2$。

证明：

此时 $5^{\frac{p-1}{2}} \equiv -1 \pmod p$。

由二项式展开：

$$F_{2p} = \frac{2}{2^{2p}\sqrt{5}}\left(\binom{2p}{1}\sqrt{5} + \binom{2p}{3}\sqrt{5}^3 + \ldots + \binom{2p}{2p-1}\sqrt{5}^{2p-1}\right)$$

$$F_{2p+1} = \frac{2}{2^{2p+1}\sqrt{5}}\left(\binom{2p+1}{1}\sqrt{5} + \binom{2p+1}{3}\sqrt{5}^3 + \ldots + \binom{2p+1}{2p+1}\sqrt{5}^{2p+1}\right)$$

模 p 之后，在 F_{2p} 式中，只有 $\binom{2p}{p} \equiv 2 \pmod p$ 项留了下来；在 F_{2p+1} 式中，有 $\binom{2p+1}{1} \equiv 1 \pmod p$、$\binom{2p+1}{p} \equiv 2 \pmod p$、$\binom{2p+1}{2p+1} \equiv 1 \pmod p$，三项留了下来。

$$F_{2p} \equiv \frac{1}{2}\binom{2p}{p}\sqrt{5}^{p-1} \equiv -1 \pmod p$$

$$F_{2p+1} \equiv \frac{1}{4}\left(\binom{2p+1}{1} + \binom{2p+1}{p}\sqrt{5}^{p-1} + \binom{2p+1}{2p+1}\sqrt{5}^{2p}\right) \equiv \frac{1}{4}(1 - 2 + 5) \equiv 1 \pmod p$$

于是 F_{2p} 和 F_{2p+1} 两项与 F_{-2} 和 F_{-1} 一致，所以 $2p+2$ 是周期。

结论 3：对于素数 p，M 是斐波那契数模 p^{k-1} 的周期，等价于 Mp 是斐波那契数模 p^k 的周期。特别地，M 是模 p^{k-1} 的皮萨诺周期，等价于 Mp 是模 p^k 的皮萨诺周期。

证明：

这里的证明需要把 $\frac{1+\sqrt{5}}{2}$ 看作一个整体。

由于：

$$F_M = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^M - \left(\frac{1-\sqrt{5}}{2}\right)^M\right) \equiv 0 \pmod{p^{k-1}}$$

$$F_{M+1}=\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{M+1}-\left(\frac{1-\sqrt{5}}{2}\right)^{M+1}\right)\equiv 1 \pmod{p^{k-1}}$$

因此：

$$\left(\frac{1+\sqrt{5}}{2}\right)^{M}\equiv\left(\frac{1-\sqrt{5}}{2}\right)^{M} \pmod{p^{k-1}}$$

$$1\equiv\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{M}\left(\left(\frac{1+\sqrt{5}}{2}\right)-\left(\frac{1-\sqrt{5}}{2}\right)\right)=\left(\frac{1+\sqrt{5}}{2}\right)^{M} \pmod{p^{k-1}}$$

因为反方向也可以推导，所以 M 是斐波那契数模 p^{k-1} 的周期，等价于：

$$\left(\frac{1+\sqrt{5}}{2}\right)^{M}\equiv\left(\frac{1-\sqrt{5}}{2}\right)^{M}\equiv 1 \pmod{p^{k-1}}$$

当 p 是奇素数时，由 升幂引理，有：

$$v_p\left(a^t-1\right)=v_p\left(a-1\right)+v_p(t)$$

当 $p=2$ 时，由 升幂引理，有：

$$v_2\left(a^t-1\right)=v_2\left(a-1\right)+v_2\left(a+1\right)+v_2(t)-1$$

代入 a 为 $\left(\frac{1+\sqrt{5}}{2}\right)$ 和 $\left(\frac{1-\sqrt{5}}{2}\right)$，$t$ 为 M 和 Mp，上述条件也就等价于：

$$\left(\frac{1+\sqrt{5}}{2}\right)^{Mp}\equiv\left(\frac{1-\sqrt{5}}{2}\right)^{Mp}\equiv 1 \pmod{p^{k}}$$

因此也等价于 Mp 是斐波那契数模 p^k 的周期。

因为周期等价，所以最小正周期也等价。

三个结论证完。据此可以写出代码：

```
struct prime {
  unsigned long long p;
  int times;
};

struct prime pp[2048];
int pptop;

unsigned long long get_cycle_from_mod(
    unsigned long long mod)  // 这里求解的只是周期，不一定是最小正周期
{
  pptop = 0;
  srand(time(0));
  while (n != 1) {
    __int128_t factor = (__int128_t)10000000000 * 10000000000;
    min_factor(mod, &factor);  // 计算最小素因数
    struct prime temp;
    temp.p = factor;
    for (temp.times = 0; mod % factor == 0; temp.times++) {
      mod /= factor;
    }
    pp[pptop] = temp;
    pptop++;
```

```
  }
  unsigned long long m = 1;
  for (int i = 0; i < pptop; ++i) {
    int g;
    if (pp[i].p == 2) {
      g = 3;
    } else if (pp[i].p == 5) {
      g = 20;
    } else if (pp[i].p % 5 == 1 || pp[i].p % 5 == 4) {
      g = pp[i].p - 1;
    } else {
      g = (pp[i].p + 1) << 1;
    }
    m = lcm(m, g * qpow(pp[i].p, pp[i].times - 1));
  }
  return m;
}
```

习题

- SPOJ - Euclid Algorithm Revisited[7]
- SPOJ - Fibonacci Sum[8]
- HackerRank - Is Fibo[9]
- Project Euler - Even Fibonacci numbers[10]
- 洛谷 P4000 斐波那契数列[11]

本页面主要译自博文 [12] 与其英文翻译版 Fibonacci Numbers[13]。其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0。

参考资料与注释

[1] OEIS A000045

[2] OEIS A000032

[3] 维基 - 拉梅

[4] 齐肯多夫定理

[5] 皮萨诺周期

[6] OEIS A001175

[7] SPOJ - Euclid Algorithm Revisited

[8] SPOJ - Fibonacci Sum

[9] HackerRank - Is Fibo

[10] Project Euler - Even Fibonacci numbers

[11] 洛谷 P4000 斐波那契数列

[12]

[13] Fibonacci Numbers

9.14.6 错位排列

错位排列

定义

错位排列（derangement）是没有任何元素出现在其有序位置的排列。即，对于 $1\sim n$ 的排列 P，如果满足 $P_i\neq i$，则称 P 是 n 的错位排列。

例如，三元错位排列有 $\{2,3,1\}$ 和 $\{3,1,2\}$。四元错位排列有 $\{2,1,4,3\}$、$\{2,3,4,1\}$、$\{2,4,1,3\}$、$\{3,1,4,2\}$、$\{3,4,1,2\}$、$\{3,4,2,1\}$、$\{4,1,2,3\}$、$\{4,3,1,2\}$ 和 $\{4,3,2,1\}$。错位排列是没有不动点的排列，即没有长度为 1 的循环。

容斥原理的计算

全集 U 即为 $1\sim n$ 的排列，$|U|=n!$；属性就是 $P_i\neq i$. 套用补集的公式，问题变成求 $\left|\bigcup_{i=1}^{n}\overline{S_i}\right|$.

可以知道，$\overline{S_i}$ 的含义是满足 $P_i=i$ 的排列的数量。用容斥原理把问题式子展开，需要对若干个特定的集合的交集求大小，即：

$$\left|\bigcap_{i=1}^{k}S_{a_i}\right|$$

其中省略了 $a_i<a_{i+1}$ 的条件以方便表示。上述 k 个集合的交集表示有 k 个变量满足 $P_{a_i}=a_i$ 的排列数，而剩下 $n-k$ 个数的位置任意，因此排列数：

$$\left|\bigcap_{i=1}^{k}S_{a_i}\right|=(n-k)!$$

那么选择 k 个元素的方案数为 $\binom{n}{k}$，因此有：

$$\begin{aligned}\left|\bigcup_{i=1}^{n}\overline{S_i}\right|&=\sum_{k=1}^{n}(-1)^{k-1}\sum_{a_{1,\cdots,k}}\left|\bigcap_{i=1}^{k}S_{a_i}\right|\\&=\sum_{k=1}^{n}(-1)^{k-1}\binom{n}{k}(n-k)!\\&=\sum_{k=1}^{n}(-1)^{k-1}\frac{n!}{k!}\\&=n!\sum_{k=1}^{n}\frac{(-1)^{k-1}}{k!}\end{aligned}$$

因此 n 的错位排列数为：

$$D_n=n!-n!\sum_{k=1}^{n}\frac{(-1)^{k-1}}{k!}=n!\sum_{k=0}^{n}\frac{(-1)^k}{k!}$$

错位排列数列的前几项为 $0,1,2,9,44,265$（OEIS A000166[1]）。

递推的计算

把错位排列问题具体化，考虑这样一个问题：

n 封不同的信，编号分别是 $1,2,3,4,5$，现在要把这五封信放在编号 $1,2,3,4,5$ 的信封中，要求信封的编号与信的编号不一样。问有多少种不同的放置方法？

假设考虑到第 n 个信封，初始时暂时把第 n 封信放在第 n 个信封中，然后考虑两种情况的递推：

- 前面 $n-1$ 个信封全部装错；
- 前面 $n-1$ 个信封有一个没有装错其余全部装错。

对于第一种情况，前面 $n-1$ 个信封全部装错：因为前面 $n-1$ 个已经全部装错了，所以第 n 封只需要与前面任一一个位置交换即可，总共有 $D_{n-1}\times(n-1)$ 种情况。

对于第二种情况，前面 $n-1$ 个信封有一个没有装错其余全部装错：考虑这种情况的目的在于，若 $n-1$ 个信封中如果有一个没装错，那么把那个没装错的与 n 交换，即可得到一个全错位排列情况。

其他情况，不可能通过一次操作来把它变成一个长度为 n 的错排。

于是可得，错位排列数满足递推关系：

$$D_n=(n-1)(D_{n-1}+D_{n-2})$$

这里也给出另一个递推关系：

$$D_n=nD_{n-1}+(-1)^n$$

其他关系

错位排列数有一个向下取整的简单表达式，增长速度与阶乘仅相差常数：

$$D_n=\left\lfloor\frac{n!}{\mathrm{e}}\right\rfloor$$

随着元素数量的增加，形成错位排列的概率 P 接近：

$$P=\lim_{n\to\infty}\frac{D_n}{n!}=\frac{1}{\mathrm{e}}$$

参考资料与注释

[1] OEIS A000166

9.14.7 卡特兰数

Catalan 数列

Catalan 数列 H_n 可以应用于以下问题：

1. 有 $2n$ 个人排成一行进入剧场。入场费 5 元。其中只有 n 个人有一张 5 元钞票，另外 n 人只有 10 元钞票，剧院无其它钞票，问有多少种方法使得只要有 10 元的人买票，售票处就有 5 元的钞票找零？
2. 有一个大小为 $n\times n$ 的方格图左下角为 $(0,0)$ 右上角为 (n,n)，从左下角开始每次都只能向右或者向上走一单位，不走到对角线 $y=x$ 上方（但可以触碰）的情况下到达右上角有多少可能的路径？
3. 在圆上选择 $2n$ 个点，将这些点成对连接起来使得所得到的 n 条线段不相交的方法数？
4. 对角线不相交的情况下，将一个凸多边形区域分成三角形区域的方法数？
5. 一个栈（无穷大）的进栈序列为 $1,2,3,\cdots,n$ 有多少个不同的出栈序列？
6. n 个结点可构造多少个不同的二叉树？
7. 由 n 个 $+1$ 和 n 个 -1 组成的 $2n$ 个数 $a_1,a_2,\cdots,a_{2n}$，其部分和满足 $a_1+a_2+\cdots+a_k\geq 0$ $(k=1,2,3,\cdots,2n)$，有多少个满足条件的数列？

其对应的序列为：

H_0	H_1	H_2	H_3	H_4	H_5	H_6	$\cdots$
1	1	2	5	14	42	132	$\cdots$

递推式

该递推关系的解为：

$$H_n = \frac{\binom{2n}{n}}{n+1}(n \geq 2, n \in \mathbf{N}_+)$$

关于 Catalan 数的常见公式：

$$H_n = \begin{cases} \sum_{i=1}^{n} H_{i-1}H_{n-i} & n \geq 2, n \in \mathbf{N}_+ \\ 1 & n = 0, 1 \end{cases}$$

$$H_n = \frac{H_{n-1}(4n-2)}{n+1}$$

$$H_n = \binom{2n}{n} - \binom{2n}{n-1}$$

" **例题 洛谷 P1044 栈**[1]"

题目大意：入栈顺序为 $1, 2, \ldots, n$，求所有可能的出栈顺序的总数。

```cpp
#include <iostream>
using namespace std;
int n;
long long f[25];

int main() {
  f[0] = 1;
  cin >> n;
  for (int i = 1; i <= n; i++) f[i] = f[i - 1] * (4 * i - 2) / (i + 1);
  // 这里用的是常见公式 2
  cout << f[n] << endl;
  return 0;
}
```

```python
f = [0] * 25
f[0] = 1
n = int(input())
for i in range(1, n + 1):
    f[i] = int(f[i - 1] * (4 * i - 2) // (i + 1))
    # 这里用的是常见公式 2
print(f[n])
```

封闭形式

卡特兰数的递推式为

$$H_n = \sum_{i=0}^{n-1} H_i H_{n-i-1} \quad (n \geq 2)$$

其中 $H_0=1,H_1=1$。设它的普通生成函数为 $H(x)$。

我们发现卡特兰数的递推式与卷积的形式很相似，因此我们用卷积来构造关于 $H(x)$ 的方程：

$$
\begin{aligned}
H(x) &= \sum_{n\geq 0} H_n x^n \\
&= 1+\sum_{n\geq 1}\sum_{i=0}^{n-1} H_i x^i H_{n-i-1} x^{n-i-1} x \\
&= 1+x\sum_{i\geq 0} H_i x^i \sum_{n\geq 0} H_n x^n \\
&= 1+xH^2(x)
\end{aligned}
$$

解得

$$H(x)=\frac{1\pm\sqrt{1-4x}}{2x}$$

那么这就产生了一个问题：我们应该取哪一个根呢？我们将其分子有理化：

$$H(x)=\frac{2}{1\mp\sqrt{1-4x}}$$

代入 $x=0$，我们得到的是 $H(x)$ 的常数项，也就是 H_0。当 $H(x)=\dfrac{2}{1+\sqrt{1-4x}}$ 的时候有 $H(0)=1$，满足要求。而另一个解会出现分母为 0 的情况（不收敛），舍弃。

因此我们得到了卡特兰数生成函数的封闭形式：

$$H(x)=\frac{1-\sqrt{1-4x}}{2x}$$

接下来我们要将其展开。但注意到它的分母不是斐波那契数列那样的多项式形式，因此不方便套用等比数列的展开形式。在这里我们需要使用牛顿二项式定理。我们来先展开 $\sqrt{1-4x}$：

$$
\begin{aligned}
(1-4x)^{\frac{1}{2}} &= \sum_{n\geq 0}\binom{\frac{1}{2}}{n}(-4x)^n \\
&= 1+\sum_{n\geq 1}\frac{\left(\frac{1}{2}\right)^{\underline{n}}}{n!}(-4x)^n
\end{aligned}
\tag{1}
$$

注意到

$$
\begin{aligned}
\left(\frac{1}{2}\right)^{\underline{n}} &= \frac{1}{2}\frac{-1}{2}\frac{-3}{2}\cdots\frac{-(2n-3)}{2} \\
&= \frac{(-1)^{n-1}(2n-3)!!}{2^n} \\
&= \frac{(-1)^{n-1}(2n-2)!}{2^n(2n-2)!!} \\
&= \frac{(-1)^{n-1}(2n-2)!}{2^{2n-1}(n-1)!}
\end{aligned}
$$

这里使用了双阶乘的化简技巧。那么带回 (1) 得到

$$
\begin{aligned}
(1-4x)^{\frac{1}{2}} &= 1+\sum_{n\geq 1}\frac{(-1)^{n-1}(2n-2)!}{2^{2n-1}(n-1)!n!}(-4x)^n \\
&= 1-\sum_{n\geq 1}\frac{(2n-2)!}{(n-1)!n!}2x^n \\
&= 1-\sum_{n\geq 1}\binom{2n-1}{n}\frac{1}{(2n-1)}2x^n
\end{aligned}
$$

带回原式得到

$$
\begin{aligned}
H(x) &= \frac{1-\sqrt{1-4x}}{2x} \\
&= \frac{1}{2x}\sum_{n\geq 1}\binom{2n-1}{n}\frac{1}{(2n-1)}2x^n \\
&= \sum_{n\geq 1}\binom{2n-1}{n}\frac{1}{(2n-1)}x^{n-1} \\
&= \sum_{n\geq 0}\binom{2n+1}{n+1}\frac{1}{(2n+1)}x^n \\
&= \sum_{n\geq 0}\binom{2n}{n}\frac{1}{n+1}x^n
\end{aligned}
$$

这样我们就得到了卡特兰数的通项公式。

路径计数问题

非降路径是指只能向上或向右走的路径。

1. 从 $(0,0)$ 到 (m,n) 的非降路径数等于 m 个 x 和 n 个 y 的排列数，即 $\dbinom{n+m}{m}$。
2. 从 $(0,0)$ 到 (n,n) 的除端点外不接触直线 $y=x$ 的非降路径数：
   先考虑 $y=x$ 下方的路径，都是从 $(0,0)$ 出发，经过 $(1,0)$ 及 $(n,n-1)$ 到 (n,n)，可以看做是 $(1,0)$ 到 $(n,n-1)$ 不接触 $y=x$ 的非降路径数。
   所有的的非降路径有 $\dbinom{2n-2}{n-1}$ 条。对于这里面任意一条接触了 $y=x$ 的路径，可以把它最后离开这条线的点到 $(1,0)$ 之间的部分关于 $y=x$ 对称变换，就得到从 $(0,1)$ 到 $(n,n-1)$ 的一条非降路径。反之也成立。从而 $y=x$ 下方的非降路径数是 $\dbinom{2n-2}{n-1}-\dbinom{2n-2}{n}$。根据对称性可知所求答案为 $2\dbinom{2n-2}{n-1}-2\dbinom{2n-2}{n}$。
3. 从 $(0,0)$ 到 (n,n) 的除端点外不穿过直线 $y=x$ 的非降路径数：
   用类似的方法可以得到：$\dfrac{2}{n+1}\dbinom{2n}{n}$

参考资料与注释

[1] 洛谷 P1044 栈

9.14.8 斯特林数

第二类斯特林数（Stirling Number）

" 为什么先介绍第二类斯特林数"

虽然被称作「第二类」，第二类斯特林数却在斯特林的相关著作和具体数学中被首先描述，同时也比第一类斯特林数常用得多。

第二类斯特林数（斯特林子集数）$\begin{Bmatrix}n\\k\end{Bmatrix}$，也可记做 $S(n,k)$，表示将 n 个两两不同的元素，划分为 k 个互不区分的非空子集的方案数。

递推式

$$
\begin{Bmatrix}n\\k\end{Bmatrix}=\begin{Bmatrix}n-1\\k-1\end{Bmatrix}+k\begin{Bmatrix}n-1\\k\end{Bmatrix}
$$

边界是 $\begin{Bmatrix}n\\0\end{Bmatrix}=[n=0]$。

考虑用组合意义来证明。

我们插入一个新元素时，有两种方案：

- 将新元素单独放入一个子集，有 $\begin{Bmatrix}n-1\\k-1\end{Bmatrix}$ 种方案；
- 将新元素放入一个现有的非空子集，有 $k\begin{Bmatrix}n-1\\k\end{Bmatrix}$ 种方案。

根据加法原理，将两式相加即可得到递推式。

通项公式

$$\begin{Bmatrix}n\\m\end{Bmatrix}=\sum_{i=0}^{m}\frac{(-1)^{m-i}i^n}{i!(m-i)!}$$

使用容斥原理证明该公式。设将 n 个两两不同的元素，划分到 i 个两两不同的集合（允许空集）的方案数为 G_i，将 n 个两两不同的元素，划分到 i 个两两不同的非空集合（不允许空集）的方案数为 F_i。

显然

$$G_i=i^n$$
$$G_i=\sum_{j=0}^{i}\binom{i}{j}F_j$$

根据二项式反演

$$\begin{aligned}F_i&=\sum_{j=0}^{i}(-1)^{i-j}\binom{i}{j}G_j\\&=\sum_{j=0}^{i}(-1)^{i-j}\binom{i}{j}j^n\\&=\sum_{j=0}^{i}\frac{i!(-1)^{i-j}j^n}{j!(i-j)!}\end{aligned}$$

考虑 F_i 与 $\begin{Bmatrix}n\\i\end{Bmatrix}$ 的关系。第二类斯特林数要求集合之间互不区分，因此 F_i 正好就是 $\begin{Bmatrix}n\\i\end{Bmatrix}$ 的 $i!$ 倍。于是

$$\begin{Bmatrix}n\\m\end{Bmatrix}=\frac{F_m}{m!}=\sum_{i=0}^{m}\frac{(-1)^{m-i}i^n}{i!(m-i)!}$$

同一行第二类斯特林数的计算

「同一行」的第二类斯特林数指的是，有着不同的 i，相同的 n 的一系列 $\begin{Bmatrix}n\\i\end{Bmatrix}$。求出同一行的所有第二类斯特林数，就是对 $i=0..n$ 求出了将 n 个不同元素划分为 i 个非空集的方案数。

根据上面给出的通项公式，卷积计算即可。该做法的时间复杂度为 $O(n\log n)$。

下面的代码使用了名为 `poly` 的多项式类，仅供参考。

" 实现"

```
#ifndef _FEISTDLIB_POLY_
#define _FEISTDLIB_POLY_

/*
```

```cpp
 * This file is part of the fstdlib project.
 * Version: Build v0.0.2
 * You can check for details at https://github.com/FNatsuka/fstdlib
 */

#include <algorithm>
#include <cmath>
#include <cstdio>
#include <vector>

namespace fstdlib {

typedef long long ll;
int mod = 998244353, grt = 3;

class poly {
 private:
  std::vector<int> data;

  void out(void) {
    for (int i = 0; i < (int)data.size(); ++i) printf("%d ", data[i]);
    puts("");
  }

 public:
  poly(std::size_t len = std::size_t(0)) { data = std::vector<int>(len); }

  poly(const std::vector<int> &b) { data = b; }

  poly(const poly &b) { data = b.data; }

  void resize(std::size_t len, int val = 0) { data.resize(len, val); }

  std::size_t size(void) const { return data.size(); }

  void clear(void) { data.clear(); }
#if __cplusplus >= 201103L
  void shrink_to_fit(void) { data.shrink_to_fit(); }
#endif
  int &operator[](std::size_t b) { return data[b]; }

  const int &operator[](std::size_t b) const { return data[b]; }

  poly operator*(const poly &h) const;
  poly operator*=(const poly &h);
  poly operator*(const int &h) const;
  poly operator*=(const int &h);
  poly operator+(const poly &h) const;
  poly operator+=(const poly &h);
  poly operator-(const poly &h) const;
  poly operator-=(const poly &h);
  poly operator<<(const std::size_t &b) const;
  poly operator<<=(const std::size_t &b);
  poly operator>>(const std::size_t &b) const;
```

```
  poly operator>>=(const std::size_t &b);
  poly operator/(const int &h) const;
  poly operator/=(const int &h);
  poly operator==(const poly &h) const;
  poly operator!=(const poly &h) const;
  poly operator+(const int &h) const;
  poly operator+=(const int &h);
  poly inv(void) const;
  poly inv(const int &h) const;
  friend poly sqrt(const poly &h);
  friend poly log(const poly &h);
  friend poly exp(const poly &h);
};

int qpow(int a, int b, int p = mod) {
  int res = 1;
  while (b) {
    if (b & 1) res = (ll)res * a % p;
    a = (ll)a * a % p, b >>= 1;
  }
  return res;
}

std::vector<int> rev;

void dft_for_module(std::vector<int> &f, int n, int b) {
  static std::vector<int> w;
  w.resize(n);
  for (int i = 0; i < n; ++i)
    if (i < rev[i]) std::swap(f[i], f[rev[i]]);
  for (int i = 2; i <= n; i <<= 1) {
    w[0] = 1, w[1] = qpow(grt, (mod - 1) / i);
    if (b == -1) w[1] = qpow(w[1], mod - 2);
    for (int j = 2; j < i / 2; ++j) w[j] = (ll)w[j - 1] * w[1] % mod;
    for (int j = 0; j < n; j += i)
      for (int k = 0; k < i / 2; ++k) {
        int p = f[j + k], q = (ll)f[j + k + i / 2] * w[k] % mod;
        f[j + k] = (p + q) % mod, f[j + k + i / 2] = (p - q + mod) % mod;
      }
  }
}

poly poly::operator*(const poly &h) const {
  int N = 1;
  while (N < (int)(size() + h.size() - 1)) N <<= 1;
  std::vector<int> f(this->data), g(h.data);
  f.resize(N), g.resize(N);
  rev.resize(N);
  for (int i = 0; i < N; ++i)
    rev[i] = (rev[i >> 1] >> 1) | (i & 1 ? N >> 1 : 0);
  dft_for_module(f, N, 1), dft_for_module(g, N, 1);
  for (int i = 0; i < N; ++i) f[i] = (ll)f[i] * g[i] % mod;
  dft_for_module(f, N, -1), f.resize(size() + h.size() - 1);
  for (int i = 0, inv = qpow(N, mod - 2); i < (int)f.size(); ++i)
```

```
    f[i] = (ll)f[i] * inv % mod;
  return f;
}

poly poly::operator*=(const poly &h) { return *this = *this * h; }

poly poly::operator*(const int &h) const {
  std::vector<int> f(this->data);
  for (int i = 0; i < (int)f.size(); ++i) f[i] = (ll)f[i] * h % mod;
  return f;
}

poly poly::operator*=(const int &h) {
  for (int i = 0; i < (int)size(); ++i) data[i] = (ll)data[i] * h % mod;
  return *this;
}

poly poly::operator+(const poly &h) const {
  std::vector<int> f(this->data);
  if (f.size() < h.size()) f.resize(h.size());
  for (int i = 0; i < (int)h.size(); ++i) f[i] = (f[i] + h[i]) % mod;
  return f;
}

poly poly::operator+=(const poly &h) {
  std::vector<int> &f = this->data;
  if (f.size() < h.size()) f.resize(h.size());
  for (int i = 0; i < (int)h.size(); ++i) f[i] = (f[i] + h[i]) % mod;
  return f;
}

poly poly::operator-(const poly &h) const {
  std::vector<int> f(this->data);
  if (f.size() < h.size()) f.resize(h.size());
  for (int i = 0; i < (int)h.size(); ++i) f[i] = (f[i] - h[i] + mod) % mod;
  return f;
}

poly poly::operator-=(const poly &h) {
  std::vector<int> &f = this->data;
  if (f.size() < h.size()) f.resize(h.size());
  for (int i = 0; i < (int)h.size(); ++i) f[i] = (f[i] - h[i] + mod) % mod;
  return f;
}

poly poly::operator<<(const std::size_t &b) const {
  std::vector<int> f(size() + b);
  for (int i = 0; i < (int)size(); ++i) f[i + b] = data[i];
  return f;
}

poly poly::operator<<=(const std::size_t &b) { return *this = (*this) << b; }

poly poly::operator>>(const std::size_t &b) const {
```

```
  std::vector<int> f(size() - b);
  for (int i = 0; i < (int)f.size(); ++i) f[i] = data[i + b];
  return f;
}

poly poly::operator>>=(const std::size_t &b) { return *this = (*this) >> b; }

poly poly::operator/(const int &h) const {
  std::vector<int> f(this->data);
  int inv = qpow(h, mod - 2);
  for (int i = 0; i < (int)f.size(); ++i) f[i] = (ll)f[i] * inv % mod;
  return f;
}

poly poly::operator/=(const int &h) {
  int inv = qpow(h, mod - 2);
  for (int i = 0; i < (int)data.size(); ++i) data[i] = (ll)data[i] * inv % mod;
  return *this;
}

poly poly::inv(void) const {
  int N = 1;
  while (N < (int)(size() + size() - 1)) N <<= 1;
  std::vector<int> f(N), g(N), d(this->data);
  d.resize(N), f[0] = qpow(d[0], mod - 2);
  for (int w = 2; w < N; w <<= 1) {
    for (int i = 0; i < w; ++i) g[i] = d[i];
    rev.resize(w << 1);
    for (int i = 0; i < w * 2; ++i)
      rev[i] = (rev[i >> 1] >> 1) | (i & 1 ? w : 0);
    dft_for_module(f, w << 1, 1), dft_for_module(g, w << 1, 1);
    for (int i = 0; i < w * 2; ++i)
      f[i] = (ll)f[i] * (2 + mod - (ll)f[i] * g[i] % mod) % mod;
    dft_for_module(f, w << 1, -1);
    for (int i = 0, inv = qpow(w << 1, mod - 2); i < w; ++i)
      f[i] = (ll)f[i] * inv % mod;
    for (int i = w; i < w * 2; ++i) f[i] = 0;
  }
  f.resize(size());
  return f;
}

poly poly::operator==(const poly &h) const {
  if (size() != h.size()) return 0;
  for (int i = 0; i < (int)size(); ++i)
    if (data[i] != h[i]) return 0;
  return 1;
}

poly poly::operator!=(const poly &h) const {
  if (size() != h.size()) return 1;
  for (int i = 0; i < (int)size(); ++i)
    if (data[i] != h[i]) return 1;
  return 0;
```

```
}

poly poly::operator+(const int &h) const {
  poly f(this->data);
  f[0] = (f[0] + h) % mod;
  return f;
}

poly poly::operator+=(const int &h) { return *this = (*this) + h; }

poly poly::inv(const int &h) const {
  poly f(*this);
  f.resize(h);
  return f.inv();
}

int modsqrt(int h, int p = mod) { return 1; }

poly sqrt(const poly &h) {
  int N = 1;
  while (N < (int)(h.size() + h.size() - 1)) N <<= 1;
  poly f(N), g(N), d(h);
  d.resize(N), f[0] = modsqrt(d[0]);
  for (int w = 2; w < N; w <<= 1) {
    g.resize(w);
    for (int i = 0; i < w; ++i) g[i] = d[i];
    f = (f + f.inv(w) * g) / 2;
    f.resize(w);
  }
  f.resize(h.size());
  return f;
}

poly log(const poly &h) {
  poly f(h);
  for (int i = 1; i < (int)f.size(); ++i) f[i - 1] = (ll)f[i] * i % mod;
  f[f.size() - 1] = 0, f = f * h.inv(), f.resize(h.size());
  for (int i = (int)f.size() - 1; i > 0; --i)
    f[i] = (ll)f[i - 1] * qpow(i, mod - 2) % mod;
  f[0] = 0;
  return f;
}

poly exp(const poly &h) {
  int N = 1;
  while (N < (int)(h.size() + h.size() - 1)) N <<= 1;
  poly f(N), g(N), d(h);
  f[0] = 1, d.resize(N);
  for (int w = 2; w < N; w <<= 1) {
    f.resize(w), g.resize(w);
    for (int i = 0; i < w; ++i) g[i] = d[i];
    f = f * (g + 1 - log(f));
    f.resize(w);
  }
```

```
  f.resize(h.size());
  return f;
}

struct comp {
  long double x, y;

  comp(long double _x = 0, long double _y = 0) : x(_x), y(_y) {}

  comp operator*(const comp &b) const {
    return comp(x * b.x - y * b.y, x * b.y + y * b.x);
  }

  comp operator+(const comp &b) const { return comp(x + b.x, y + b.y); }

  comp operator-(const comp &b) const { return comp(x - b.x, y - b.y); }

  comp conj(void) { return comp(x, -y); }
};

const int EPS = 1e-9;

template <typename FLOAT_T>
FLOAT_T fabs(const FLOAT_T &x) {
  return x > 0 ? x : -x;
}

template <typename FLOAT_T>
FLOAT_T sin(const FLOAT_T &x, const long double &EPS = fstdlib::EPS) {
  FLOAT_T res = 0, delt = x;
  int d = 0;
  while (fabs(delt) > EPS) {
    res += delt, ++d;
    delt *= -x * x / ((2 * d) * (2 * d + 1));
  }
  return res;
}

template <typename FLOAT_T>
FLOAT_T cos(const FLOAT_T &x, const long double &EPS = fstdlib::EPS) {
  FLOAT_T res = 0, delt = 1;
  int d = 0;
  while (fabs(delt) > EPS) {
    res += delt, ++d;
    delt *= -x * x / ((2 * d) * (2 * d - 1));
  }
  return res;
}

const long double PI = std::acos((long double)(-1));

void dft_for_complex(std::vector<comp> &f, int n, int b) {
  static std::vector<comp> w;
  w.resize(n);
```

```
  for (int i = 0; i < n; ++i)
    if (i < rev[i]) std::swap(f[i], f[rev[i]]);
  for (int i = 2; i <= n; i <<= 1) {
    w[0] = comp(1, 0), w[1] = comp(cos(2 * PI / i), b * sin(2 * PI / i));
    for (int j = 2; j < i / 2; ++j) w[j] = w[j - 1] * w[1];
    for (int j = 0; j < n; j += i)
      for (int k = 0; k < i / 2; ++k) {
        comp p = f[j + k], q = f[j + k + i / 2] * w[k];
        f[j + k] = p + q, f[j + k + i / 2] = p - q;
      }
  }
}

class arbitrary_module_poly {
 private:
  std::vector<int> data;

  int construct_element(int D, ll x, ll y, ll z) const {
    x %= mod, y %= mod, z %= mod;
    return ((ll)D * D * x % mod + (ll)D * y % mod + z) % mod;
  }

 public:
  int mod;

  arbitrary_module_poly(std::size_t len = std::size_t(0),
                        int module_value = 1e9 + 7) {
    mod = module_value;
    data = std::vector<int>(len);
  }

  arbitrary_module_poly(const std::vector<int> &b, int module_value = 1e9 + 7) {
    mod = module_value;
    data = b;
  }

  arbitrary_module_poly(const arbitrary_module_poly &b) {
    mod = b.mod;
    data = b.data;
  }

  void resize(std::size_t len, const int &val = 0) { data.resize(len, val); }

  std::size_t size(void) const { return data.size(); }

  void clear(void) { data.clear(); }
#if __cplusplus >= 201103L
  void shrink_to_fit(void) { data.shrink_to_fit(); }
#endif
  int &operator[](std::size_t b) { return data[b]; }

  const int &operator[](std::size_t b) const { return data[b]; }

  arbitrary_module_poly operator*(const arbitrary_module_poly &h) const;
```

```
  arbitrary_module_poly operator*=(const arbitrary_module_poly &h);
  arbitrary_module_poly operator*(const int &h) const;
  arbitrary_module_poly operator*=(const int &h);
  arbitrary_module_poly operator+(const arbitrary_module_poly &h) const;
  arbitrary_module_poly operator+=(const arbitrary_module_poly &h);
  arbitrary_module_poly operator-(const arbitrary_module_poly &h) const;
  arbitrary_module_poly operator-=(const arbitrary_module_poly &h);
  arbitrary_module_poly operator<<(const std::size_t &b) const;
  arbitrary_module_poly operator<<=(const std::size_t &b);
  arbitrary_module_poly operator>>(const std::size_t &b) const;
  arbitrary_module_poly operator>>=(const std::size_t &b);
  arbitrary_module_poly operator/(const int &h) const;
  arbitrary_module_poly operator/=(const int &h);
  arbitrary_module_poly operator==(const arbitrary_module_poly &h) const;
  arbitrary_module_poly operator!=(const arbitrary_module_poly &h) const;
  arbitrary_module_poly inv(void) const;
  arbitrary_module_poly inv(const int &h) const;
  friend arbitrary_module_poly sqrt(const arbitrary_module_poly &h);
  friend arbitrary_module_poly log(const arbitrary_module_poly &h);
};

arbitrary_module_poly arbitrary_module_poly::operator*(
    const arbitrary_module_poly &h) const {
  int N = 1;
  while (N < (int)(size() + h.size() - 1)) N <<= 1;
  std::vector<comp> f(N), g(N), p(N), q(N);
  const int D = std::sqrt(mod);
  for (int i = 0; i < (int)size(); ++i)
    f[i].x = data[i] / D, f[i].y = data[i] % D;
  for (int i = 0; i < (int)h.size(); ++i) g[i].x = h[i] / D, g[i].y = h[i] % D;
  rev.resize(N);
  for (int i = 0; i < N; ++i)
    rev[i] = (rev[i >> 1] >> 1) | (i & 1 ? N >> 1 : 0);
  dft_for_complex(f, N, 1), dft_for_complex(g, N, 1);
  for (int i = 0; i < N; ++i) {
    p[i] = (f[i] + f[(N - i) % N].conj()) * comp(0.50, 0) * g[i];
    q[i] = (f[i] - f[(N - i) % N].conj()) * comp(0, -0.5) * g[i];
  }
  dft_for_complex(p, N, -1), dft_for_complex(q, N, -1);
  std::vector<int> r(size() + h.size() - 1);
  for (int i = 0; i < (int)r.size(); ++i)
    r[i] = construct_element(D, p[i].x / N + 0.5, (p[i].y + q[i].x) / N + 0.5,
                             q[i].y / N + 0.5);
  return arbitrary_module_poly(r, mod);
}

arbitrary_module_poly arbitrary_module_poly::operator*=(
    const arbitrary_module_poly &h) {
  return *this = *this * h;
}

arbitrary_module_poly arbitrary_module_poly::operator*(const int &h) const {
  std::vector<int> f(this->data);
  for (int i = 0; i < (int)f.size(); ++i) f[i] = (ll)f[i] * h % mod;
```

```
  return arbitrary_module_poly(f, mod);
}

arbitrary_module_poly arbitrary_module_poly::operator*=(const int &h) {
  for (int i = 0; i < (int)size(); ++i) data[i] = (ll)data[i] * h % mod;
  return *this;
}

arbitrary_module_poly arbitrary_module_poly::operator+(
    const arbitrary_module_poly &h) const {
  std::vector<int> f(this->data);
  if (f.size() < h.size()) f.resize(h.size());
  for (int i = 0; i < (int)h.size(); ++i) f[i] = (f[i] + h[i]) % mod;
  return arbitrary_module_poly(f, mod);
}

arbitrary_module_poly arbitrary_module_poly::operator+=(
    const arbitrary_module_poly &h) {
  if (size() < h.size()) resize(h.size());
  for (int i = 0; i < (int)h.size(); ++i) data[i] = (data[i] + h[i]) % mod;
  return *this;
}

arbitrary_module_poly arbitrary_module_poly::operator-(
    const arbitrary_module_poly &h) const {
  std::vector<int> f(this->data);
  if (f.size() < h.size()) f.resize(h.size());
  for (int i = 0; i < (int)h.size(); ++i) f[i] = (f[i] + mod - h[i]) % mod;
  return arbitrary_module_poly(f, mod);
}

arbitrary_module_poly arbitrary_module_poly::operator-=(
    const arbitrary_module_poly &h) {
  if (size() < h.size()) resize(h.size());
  for (int i = 0; i < (int)h.size(); ++i)
    data[i] = (data[i] + mod - h[i]) % mod;
  return *this;
}

arbitrary_module_poly arbitrary_module_poly::operator<<(
    const std::size_t &b) const {
  std::vector<int> f(size() + b);
  for (int i = 0; i < (int)size(); ++i) f[i + b] = data[i];
  return arbitrary_module_poly(f, mod);
}

arbitrary_module_poly arbitrary_module_poly::operator<<=(const std::size_t &b) {
  return *this = (*this) << b;
}

arbitrary_module_poly arbitrary_module_poly::operator>>(
    const std::size_t &b) const {
  std::vector<int> f(size() - b);
  for (int i = 0; i < (int)f.size(); ++i) f[i] = data[i + b];
```

```
  return arbitrary_module_poly(f, mod);
}

arbitrary_module_poly arbitrary_module_poly::operator>>=(const std::size_t &b) {
  return *this = (*this) >> b;
}

arbitrary_module_poly arbitrary_module_poly::inv(void) const {
  int N = 1;
  while (N < (int)(size() + size() - 1)) N <<= 1;
  arbitrary_module_poly f(1, mod), g(N, mod), h(*this), f2(1, mod);
  f[0] = qpow(data[0], mod - 2, mod), h.resize(N), f2[0] = 2;
  for (int w = 2; w < N; w <<= 1) {
    g.resize(w);
    for (int i = 0; i < w; ++i) g[i] = h[i];
    f = f * (f * g - f2) * (mod - 1);
    f.resize(w);
  }
  f.resize(size());
  return f;
}

arbitrary_module_poly arbitrary_module_poly::inv(const int &h) const {
  arbitrary_module_poly f(*this);
  f.resize(h);
  return f.inv();
}

arbitrary_module_poly arbitrary_module_poly::operator/(const int &h) const {
  int inv = qpow(h, mod - 2, mod);
  std::vector<int> f(this->data);
  for (int i = 0; i < (int)f.size(); ++i) f[i] = (ll)f[i] * inv % mod;
  return arbitrary_module_poly(f, mod);
}

arbitrary_module_poly arbitrary_module_poly::operator/=(const int &h) {
  int inv = qpow(h, mod - 2, mod);
  for (int i = 0; i < (int)size(); ++i) data[i] = (ll)data[i] * inv % mod;
  return *this;
}

arbitrary_module_poly arbitrary_module_poly::operator==(
    const arbitrary_module_poly &h) const {
  if (size() != h.size() || mod != h.mod) return 0;
  for (int i = 0; i < (int)size(); ++i)
    if (data[i] != h[i]) return 0;
  return 1;
}

arbitrary_module_poly arbitrary_module_poly::operator!=(
    const arbitrary_module_poly &h) const {
  if (size() != h.size() || mod != h.mod) return 1;
  for (int i = 0; i < (int)size(); ++i)
    if (data[i] != h[i]) return 1;
```

```
  return 0;
}

arbitrary_module_poly sqrt(const arbitrary_module_poly &h) {
  int N = 1;
  while (N < (int)(h.size() + h.size() - 1)) N <<= 1;
  arbitrary_module_poly f(1, mod), g(N, mod), d(h);
  f[0] = modsqrt(h[0], mod), d.resize(N);
  for (int w = 2; w < N; w <<= 1) {
    g.resize(w);
    for (int i = 0; i < w; ++i) g[i] = d[i];
    f = (f + f.inv(w) * g) / 2;
    f.resize(w);
  }
  f.resize(h.size());
  return f;
}

arbitrary_module_poly log(const arbitrary_module_poly &h) {
  arbitrary_module_poly f(h);
  for (int i = 1; i < (int)f.size(); ++i) f[i - 1] = (ll)f[i] * i % f.mod;
  f[f.size() - 1] = 0, f = f * h.inv(), f.resize(h.size());
  for (int i = (int)f.size() - 1; i > 0; --i)
    f[i] = (ll)f[i - 1] * qpow(i, f.mod - 2, f.mod) % f.mod;
  f[0] = 0;
  return f;
}

typedef arbitrary_module_poly m_poly;
}  // namespace fstdlib

#endif
```

" 实现"

```
int main() {
  scanf("%d", &n);
  fact[0] = 1;
  for (int i = 1; i <= n; ++i) fact[i] = (ll)fact[i - 1] * i % mod;
  exgcd(fact[n], mod, ifact[n], ifact[0]),
      ifact[n] = (ifact[n] % mod + mod) % mod;
  for (int i = n - 1; i >= 0; --i) ifact[i] = (ll)ifact[i + 1] * (i + 1) % mod;
  poly f(n + 1), g(n + 1);
  for (int i = 0; i <= n; ++i)
    g[i] = (i & 1 ? mod - 1ll : 1ll) * ifact[i] % mod,
    f[i] = (ll)qpow(i, n) * ifact[i] % mod;
  f *= g, f.resize(n + 1);
  for (int i = 0; i <= n; ++i) printf("%d ", f[i]);
  return 0;
}
```

同一列第二类斯特林数的计算

「同一列」的第二类斯特林数指的是，有着不同的 i，相同的 k 的一系列 $\begin{Bmatrix}i\\k\end{Bmatrix}$。求出同一列的所有第二类斯特林数，就是对 $i=0..n$ 求出了将 i 个不同元素划分为 k 个非空集的方案数。

利用指数型生成函数计算。

一个盒子装 i 个物品且盒子非空的方案数是 $[i>0]$。我们可以写出它的指数型生成函数为 $F(x)=\sum\limits_{i=1}^{+\infty}\frac{x^i}{i!}=\mathrm{e}^x-1$。经过之前的学习，我们明白 $F^k(x)$ 就是 i 个有标号物品放到 k 个有标号盒子里的指数型生成函数，那么除掉 $k!$ 就是 i 个有标号物品放到 k 个无标号盒子里的指数型生成函数。

$\begin{Bmatrix}i\\k\end{Bmatrix}=\frac{\left[\frac{x^i}{i!}\right]F^k(x)}{k!}$，$O(n\log n)$ 计算多项式幂即可。

另外，$\exp F(x)=\sum\limits_{i=0}^{+\infty}\frac{F^i(x)}{i!}$ 就是 i 个有标号物品放到任意多个无标号盒子里的指数型生成函数（EXP 通过每项除以一个 $i!$ 去掉了盒子的标号）。这其实就是贝尔数的生成函数。

这里涉及到很多「有标号」「无标号」的内容，注意辨析。

" 实现"

```
int main() {
  scanf("%d%d", &n, &k);
  poly f(n + 1);
  fact[0] = 1;
  for (int i = 1; i <= n; ++i) fact[i] = (ll)fact[i - 1] * i % mod;
  for (int i = 1; i <= n; ++i) f[i] = qpow(fact[i], mod - 2);
  f = exp(log(f >> 1) * k) << k, f.resize(n + 1);
  int inv = qpow(fact[k], mod - 2);
  for (int i = 0; i <= n; ++i)
    printf("%lld ", (ll)f[i] * fact[i] % mod * inv % mod);
  return 0;
}
```

第一类斯特林数（Stirling Number）

第一类斯特林数（斯特林轮换数）$\begin{bmatrix}n\\k\end{bmatrix}$，也可记做 $s(n,k)$，表示将 n 个两两不同的元素，划分为 k 个互不区分的非空轮换的方案数。

一个轮换就是一个首尾相接的环形排列。我们可以写出一个轮换 $[A,B,C,D]$，并且我们认为 $[A,B,C,D]=[B,C,D,A]=[C,D,A,B]=[D,A,B,C]$，即，两个可以通过旋转而互相得到的轮换是等价的。注意，我们不认为两个可以通过翻转而相互得到的轮换等价，即 $[A,B,C,D]\neq[D,C,B,A]$。

递推式

$$\begin{bmatrix}n\\k\end{bmatrix}=\begin{bmatrix}n-1\\k-1\end{bmatrix}+(n-1)\begin{bmatrix}n-1\\k\end{bmatrix}$$

边界是 $\begin{bmatrix}n\\0\end{bmatrix}=[n=0]$。

该递推式的证明可以考虑其组合意义。

我们插入一个新元素时，有两种方案：

- 将该新元素置于一个单独的轮换中，共有 $\begin{bmatrix}n-1\\k-1\end{bmatrix}$ 种方案；
- 将该元素插入到任何一个现有的轮换中，共有 $(n-1)\begin{bmatrix}n-1\\k\end{bmatrix}$ 种方案。

根据加法原理，将两式相加即可得到递推式。

通项公式

第一类斯特林数没有实用的通项公式。

同一行第一类斯特林数的计算

类似第二类斯特林数，我们构造同行第一类斯特林数的生成函数，即

$$F_n(x)=\sum_{i=0}^{n}\begin{bmatrix}n\\i\end{bmatrix}x^i$$

根据递推公式，不难写出

$$F_n(x)=(n-1)F_{n-1}(x)+xF_{n-1}(x)$$

于是

$$F_n(x)=\prod_{i=0}^{n-1}(x+i)=\frac{(x+n-1)!}{(x-1)!}$$

这其实是 x 的 n 次上升阶乘幂，记做 $x^{\overline{n}}$。这个东西自然是可以暴力分治乘 $O(n\log^2 n)$ 求出的，但用上升幂相关做法可以 $O(n\log n)$ 求出。

同一列第一类斯特林数的计算

仿照第二类斯特林数的计算，我们可以用指数型生成函数解决该问题。注意，由于递推公式和行有关，我们不能利用递推公式计算同列的第一类斯特林数。

显然，单个轮换的指数型生成函数为

$$F(x)=\sum_{i=1}^{n}\frac{(i-1)!x^i}{i!}=\sum_{i=1}^{n}\frac{x^i}{i}$$

它的 k 次幂就是 $\begin{bmatrix}i\\k\end{bmatrix}$ 的指数型生成函数，$O(n\log n)$ 计算即可。

" 实现"

```cpp
int main() {
  scanf("%d%d", &n, &k);
  fact[0] = 1;
  for (int i = 1; i <= n; ++i) fact[i] = (ll)fact[i - 1] * i % mod;
  ifact[n] = qpow(fact[n], mod - 2);
  for (int i = n - 1; i >= 0; --i) ifact[i] = (ll)ifact[i + 1] * (i + 1) % mod;
  poly f(n + 1);
  for (int i = 1; i <= n; ++i) f[i] = (ll)fact[i - 1] * ifact[i] % mod;
  f = exp(log(f >> 1) * k) << k, f.resize(n + 1);
  for (int i = 0; i <= n; ++i)
    printf("%lld ", (ll)f[i] * fact[i] % mod * ifact[k] % mod);
  return 0;
}
```

应用

上升幂与普通幂的相互转化

我们记上升阶乘幂 $x^{\overline{n}} = \prod_{k=0}^{n-1}(x+k)$。

则可以利用下面的恒等式将上升幂转化为普通幂：

$$x^{\overline{n}} = \sum_k \begin{bmatrix} n \\ k \end{bmatrix} x^k$$

如果将普通幂转化为上升幂，则有下面的恒等式：

$$x^n = \sum_k \begin{Bmatrix} n \\ k \end{Bmatrix} (-1)^{n-k} x^{\overline{k}}$$

下降幂与普通幂的相互转化

我们记下降阶乘幂 $x^{\underline{n}} = \dfrac{x!}{(x-n)!} = \prod_{k=0}^{n-1}(x-k)$。

则可以利用下面的恒等式将普通幂转化为下降幂：

$$x^n = \sum_k \begin{Bmatrix} n \\ k \end{Bmatrix} x^{\underline{k}}$$

如果将下降幂转化为普通幂，则有下面的恒等式：

$$x^{\underline{n}} = \sum_k \begin{bmatrix} n \\ k \end{bmatrix} (-1)^{n-k} x^k$$

多项式下降阶乘幂表示与多项式点值表示的关系

在这里，多项式的下降阶乘幂表示就是用

$$f(x) = \sum_{i=0}^{n} b_i x^{\underline{i}}$$

的形式表示一个多项式，而点值表示就是用 $n+1$ 个点

$$(i, a_i), i = 0..n$$

来表示一个多项式。

显然，下降阶乘幂 b 和点值 a 间满足这样的关系：

$$a_k = \sum_{i=0}^{n} b_i k^{\underline{i}}$$

即

$$a_k = \sum_{i=0}^{n} \frac{b_i k!}{(k-i)!}$$

$$\frac{a_k}{k!} = \sum_{i=0}^{k} b_i \frac{1}{(k-i)!}$$

这是一个卷积形式的式子，我们可以在 $O(n \log n)$ 的时间复杂度内完成点值和下降阶乘幂的互相转化。

习题

- HDU3625 Examining the Rooms[1]
- UOJ540 联合省选 2020 组合数问题[2]
- UOJ269 清华集训 2016 如何优雅地求和[3]

参考资料与注释

1. Stirling Number of the First Kind - Wolfram MathWorld[4]
2. Stirling Number of the Second Kind - Wolfram MathWorld[5]

参考资料与注释

[1] HDU3625 Examining the Rooms

[2] UOJ540 联合省选 2020 组合数问题

[3] UOJ269 清华集训 2016 如何优雅地求和

[4] Stirling Number of the First Kind - Wolfram MathWorld

[5] Stirling Number of the Second Kind - Wolfram MathWorld

9.14.9 贝尔数

贝尔数 B_n 以埃里克 · 坦普尔 · 贝尔命名，是组合数学中的一组整数数列，开首是（OEIS A000110[1]）：

$$B_0 = 1, B_1 = 1, B_2 = 2, B_3 = 5, B_4 = 15, B_5 = 52, B_6 = 203, \ldots$$

B_n 是基数为 n 的集合的划分方法的数目。集合 S 的一个划分是定义为 S 的两两不相交的非空子集的族，它们的并是 S。例如 $B_3 = 5$ 因为 3 个元素的集合 a, b, c 有 5 种不同的划分方法：

$$\begin{aligned}
&\{\{a\}, \{b\}, \{c\}\} \\
&\{\{a\}, \{b, c\}\} \\
&\{\{b\}, \{a, c\}\} \\
&\{\{c\}, \{a, b\}\} \\
&\{\{a, b, c\}\}
\end{aligned}$$

B_0 是 1 因为空集正好有 1 种划分方法。

递推公式

贝尔数适合递推公式：

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$$

证明：

B_{n+1} 是含有 $n+1$ 个元素集合的划分个数，设 D_n 的集合为 $\{b_1, b_2, b_3, \ldots, b_n\}$，$D_{n+1}$ 的集合为 $\{b_1, b_2, b_3, \ldots, b_n, b_{n+1}\}$，那么可以认为 D_{n+1} 是有 D_n 增添了一个 b_{n+1} 而产生的，考虑元素 b_{n+1}。

- 假如它被单独分到一类，那么还剩下 n 个元素，这种情况下划分数为 $\binom{n}{n} B_n$；
- 假如它和某 1 个元素分到一类，那么还剩下 $n-1$ 个元素，这种情况下划分数为 $\binom{n}{n-1} B_{n-1}$；
- 假如它和某 2 个元素分到一类，那么还剩下 $n-2$ 个元素，这种情况下划分数为 $\binom{n}{n-2} B_{n-2}$；
- ……

以此类推就得到了上面的公式。

每个贝尔数都是相应的 第二类斯特林数 的和。因为第二类斯特林数是把基数为 n 的集合划分为正好 k 个非空集的方法数目。

$$B_n = \sum_{k=0}^{n} \left\{ {n \atop k} \right\}$$

贝尔三角形

用以下方法构造一个三角矩阵（形式类似杨辉三角形）：

- $a_{0,0} = 1$；
- 对于 $n \geq 1$，第 n 行首项等于上一行的末项，即 $a_{n,0} = a_{n-1,n-1}$；
- 对于 $m, n \geq 1$，第 n 行第 m 项等于它左边和左上角两个数之和，即 $a_{n,m} = a_{n,m-1} + a_{n-1,m-1}$。

部分结果如下：

1						
1	2					
2	3	5				
5	7	10	15			
15	20	27	37	52		
52	67	87	114	151	203	
203	255	322	409	523	674	877

每行的首项是贝尔数。可以利用这个三角形来递推求出贝尔数。

" 参考实现"

```cpp
const int maxn = 2000 + 5;
int bell[maxn][maxn];

void f(int n) {
  bell[0][0] = 1;
  for (int i = 1; i <= n; i++) {
    bell[i][0] = bell[i - 1][i - 1];
    for (int j = 1; j <= i; j++)
      bell[i][j] = bell[i - 1][j - 1] + bell[i][j - 1];
  }
}
```

```python
maxn = 2000 + 5
bell = [[0 for i in range(maxn + 1)] for j in range(maxn + 1)]
def f(n):
    bell[0][0] = 1
    for i in range(1, n + 1):
        bell[i][0] = bell[i - 1][i - 1]
        for j in range(1, i + 1):
            bell[i][j] = bell[i - 1][j - 1] + bell[i][j - 1]
```

指数生成函数

考虑贝尔数的指数生成函数及其导函数：

$$
\begin{aligned}
\hat{B}(x) &= \sum_{n=0}^{+\infty} \frac{B_n}{n!} x^n \\
&= 1 + \sum_{n=0}^{+\infty} \frac{B_{n+1}}{(n+1)!} x^{n+1} \\
\hat{B}'(x) &= \sum_{n=0}^{+\infty} \frac{B_{n+1}}{n!} x^n
\end{aligned}
$$

根据贝尔数的递推公式可以得到：

$$
\frac{B_{n+1}}{n!} = \sum_{k=0}^{n} \frac{1}{(n-k)!} \frac{B_k}{k!}
$$

这是一个卷积的式子，因此有：

$$
\hat{B}'(x) = \mathrm{e}^x \hat{B}(x)
$$

这是一个微分方程，解得：

$$
\hat{B}(x) = \exp\left(\mathrm{e}^x + C\right)
$$

最后当 $x=0$ 时 $\hat{B}(x)=1$，带入后解得 $C=-1$，得到贝尔数指数生成函数的封闭形式：

$$
\hat{B}(x) = \exp\left(\mathrm{e}^x - 1\right)
$$

预处理出 $\mathrm{e}^x - 1$ 的前 n 项后做一次 多项式 exp 即可得出贝尔数前 n 项，时间复杂度瓶颈在多项式 exp，可做到 $O(n \log n)$ 的时间复杂度。

参考文献

https://en.wikipedia.org/wiki/Bell_number[2]

参考资料与注释

[1] OEIS A000110

[2] https://en.wikipedia.org/wiki/Bell_number

9.14.10　伯努利数

伯努利数 B_n 是一个与数论有密切关联的有理数序列。前几项被发现的伯努利数分别为：
$B_0 = 1, B_1 = -\frac{1}{2}, B_2 = \frac{1}{6}, B_3 = 0, B_4 = -\frac{1}{30}, \ldots$

等幂求和

伯努利数是由雅各布 · 伯努利的名字命名的，他在研究 m 次幂和的公式时发现了奇妙的关系。我们记

$$
S_m(n) = \sum_{k=0}^{n-1} k^m = 0^m + 1^m + \cdots + (n-1)^m
$$

伯努利观察了如下一列公式，勾画出一种模式：

$$
\begin{aligned}
S_0(n) &= n \\
S_1(n) &= \frac{1}{2}n^2 - \frac{1}{2}n \\
S_2(n) &= \frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n \\
S_3(n) &= \frac{1}{4}n^4 - \frac{1}{2}n^3 + \frac{1}{4}n^2 \\
S_4(n) &= \frac{1}{5}n^5 - \frac{1}{2}n^4 + \frac{1}{3}n^3 - \frac{1}{30}n
\end{aligned}
$$

可以发现，在 $S_m(n)$ 中 n^{m+1} 的系数总是 $\frac{1}{m+1}$，n^m 的系数总是 $-\frac{1}{2}$，n^{m-1} 的系数总是 $\frac{m}{12}$，n^{m-3} 的系数是 $-\frac{m(m-1)(m-2)}{720}$，n^{m-4} 的系数总是零等。

而 n^{m-k} 的系数总是某个常数乘以 $m^{\underline{k}}$，$m^{\underline{k}}$ 表示下降阶乘幂，即 $\frac{m!}{(m-k)!}$。

递推公式

$$
\begin{aligned}
S_m(n) &= \frac{1}{m+1}\left(B_0 n^{m+1} + \binom{m+1}{1}B_1 n^m + \cdots + \binom{m+1}{m}B_m n\right) \\
&= \frac{1}{m+1}\sum_{k=0}^{m}\binom{m+1}{k}B_k n^{m+1-k}
\end{aligned}
$$

伯努利数由隐含的递推关系定义：

$$
\sum_{j=0}^{m}\binom{m+1}{j}B_j = 0, (m > 0)
$$

$$
B_0 = 1
$$

例如，$\binom{2}{0}B_0 + \binom{2}{1}B_1 = 0$，前几个值显然是

n	0	1	2	3	4	5	6	7	8	$\cdots$
B_n	1	$-\frac{1}{2}$	$\frac{1}{6}$	0	$-\frac{1}{30}$	0	$\frac{1}{42}$	0	$-\frac{1}{30}$	$\cdots$

证明

利用归纳法证明

这个证明方法来自 Concrete Mathematics 6.5 BERNOULLI NUMBER。

运用二项式系数的恒等变换和归纳法进行证明：

$$
\begin{aligned}
S_{m+1}(n) + n^{m+1} &= \sum_{k=0}^{n-1}(k+1)^{m+1} \\
&= \sum_{k=0}^{n-1}\sum_{j=0}^{m+1}\binom{m+1}{j}k^j \\
&= \sum_{j=0}^{m+1}\binom{m+1}{j}S_j(n)
\end{aligned}
$$

令 $\hat{S}_m(n) = \frac{1}{m+1}\sum_{k=0}^{m}\binom{m+1}{k}B_k n^{m+1-k}$，我们希望证明 $S_m(n) = \hat{S}_m(n)$，假设对 $j \in [0, m)$，有 $S_j(n) = \hat{S}_j(n)$。

将原式中两边都减去 $S_{m+1}(n)$ 后可以得到:

$$
\begin{aligned}
S_{m+1}(n)+n^{m+1} &= \sum_{j=0}^{m+1}\binom{m+1}{j}S_j(n) \\
n^{m+1} &= \sum_{j=0}^{m}\binom{m+1}{j}S_j(n) \\
&= \sum_{j=0}^{m-1}\binom{m+1}{j}\hat{S}_j(n)+\binom{m+1}{m}S_m(n)
\end{aligned}
$$

尝试在式子的右边加上 $\binom{m+1}{m}\hat{S}_m(n)-\binom{m+1}{m}\hat{S}_m(n)$ 再进行化简，可以得到:

$$
n^{m+1}=\sum_{j=0}^{m}\binom{m+1}{j}\hat{S}_j(n)+(m+1)(S_m(n)-\hat{S}_m(n))
$$

不妨设 $\Delta=S_m(n)-\hat{S}_m(n)$，并且将 $\hat{S}_j(n)$ 展开，那么有

$$
\begin{aligned}
n^{m+1} &= \sum_{j=0}^{m}\binom{m+1}{j}\hat{S}_j(n)+(m+1)\Delta \\
&= \sum_{j=0}^{m}\binom{m+1}{j}\frac{1}{j+1}\sum_{k=0}^{j}\binom{j+1}{k}B_k n^{j+1-k}+(m+1)\Delta
\end{aligned}
$$

将第二个 $\sum$ 中的求和顺序改为逆向，再将组合数的写法恒等变换可以得到:

$$
\begin{aligned}
n^{m+1} &= \sum_{j=0}^{m}\binom{m+1}{j}\frac{1}{j+1}\sum_{k=0}^{j}\binom{j+1}{j-k}B_{j-k}n^{k+1}+(m+1)\Delta \\
&= \sum_{j=0}^{m}\binom{m+1}{j}\frac{1}{j+1}\sum_{k=0}^{j}\binom{j+1}{k+1}B_{j-k}n^{k+1}+(m+1)\Delta \\
&= \sum_{j=0}^{m}\binom{m+1}{j}\frac{1}{j+1}\sum_{k=0}^{j}\frac{j+1}{k+1}\binom{j}{k}B_{j-k}n^{k+1}+(m+1)\Delta \\
&= \sum_{j=0}^{m}\binom{m+1}{j}\sum_{k=0}^{j}\binom{j}{k}\frac{B_{j-k}}{k+1}n^{k+1}+(m+1)\Delta
\end{aligned}
$$

对两个求和符号进行交换，可以得到:

$$
n^{m+1}=\sum_{k=0}^{m}\frac{n^{k+1}}{k+1}\sum_{j=k}^{m}\binom{m+1}{j}\binom{j}{k}B_{j-k}+(m+1)\Delta
$$

对 $\binom{m+1}{j}\binom{j}{k}$ 进行恒等变换:

$$
\binom{m+1}{j}\binom{j}{k}\ \binom{m+1}{k}\binom{m-k+1}{j-k}
$$

那么式子就变成了:

$$
\begin{aligned}
n^{m+1} &= \sum_{k=0}^{m}\frac{n^{k+1}}{k+1}\sum_{j=k}^{m}\binom{m+1}{k}\binom{m-k+1}{j-k}B_{j-k}+(m+1)\Delta \\
&= \sum_{k=0}^{m}\frac{n^{k+1}}{k+1}\binom{m+1}{k}\sum_{j=k}^{m}\binom{m-k+1}{j-k}B_{j-k}+(m+1)\Delta
\end{aligned}
$$

将所有的 $j-k$ 用 j 代替，那么就可以得到:

$$
n^{m+1}=\sum_{k=0}^{m}\frac{n^{k+1}}{k+1}\binom{m+1}{k}\sum_{j=0}^{m-k}\binom{m-k+1}{j}B_j+(m+1)\Delta
$$

考虑我们前面提到过的递归关系

$$\sum_{j=0}^{m}\binom{m+1}{j}B_j = 0, (m > 0)$$

$$B_0 = 1$$

$$\sum_{j=0}^{m}\binom{m+1}{j}B_j = [m = 0]$$

代入后可以得到：

$$\begin{aligned}
n^{m+1} &= \sum_{k=0}^{m}\frac{n^{k+1}}{k+1}\binom{m+1}{k}[m-k=0] + (m+1)\Delta \\
&= \sum_{k=0}^{m}\frac{n^{k+1}}{k+1}\binom{m+1}{k} + (m+1)\Delta \\
&= \frac{n^{m+1}}{m+1}\binom{m+1}{m} + (m+1)\Delta \\
&= n^{m+1} + (m+1)\Delta
\end{aligned}$$

于是 $\Delta = 0$，且有 $S_m(n) = \hat{S}_m(n)$。

利用指数生成函数证明

对递推式 $\sum_{j=0}^{m}\binom{m+1}{j}B_j = [m = 0]$

两边都加上 B_{m+1}，即得到：

$$\sum_{j=0}^{m+1}\binom{m+1}{j}B_j = [m = 0] + B_{m+1}$$

$$\sum_{j=0}^{m}\binom{m}{j}B_j = [m = 1] + B_m$$

$$\sum_{j=0}^{m}\frac{B_j}{j!}\cdot\frac{1}{(m-j)!} = [m = 1] + \frac{B_m}{m!}$$

设 $B(z) = \sum\limits_{i\geq 0}\dfrac{B_i}{i!}z^i$，注意到左边为卷积形式，故：

$$B(z)\mathrm{e}^z = z + B(z)$$

$$B(z) = \frac{z}{\mathrm{e}^z - 1}$$

设 $F_n(z) = \sum_{m\geq 0}\dfrac{S_m(n)}{m!}z^m$，则：

$$\begin{aligned}
F_n(z) &= \sum_{m\geq 0}\frac{S_m(n)}{m!}z^m \\
&= \sum_{m\geq 0}\sum_{i=0}^{n-1}\frac{i^m z^m}{m!}
\end{aligned}$$

调换求和顺序：

$$\begin{aligned}
F_n(z) &= \sum_{i=0}^{n-1}\sum_{m\geq 0}\frac{i^m z^m}{m!} \\
&= \sum_{i=0}^{n-1}\mathrm{e}^{iz} \\
&= \frac{\mathrm{e}^{nz}-1}{\mathrm{e}^z - 1} \\
&= \frac{z}{\mathrm{e}^z - 1}\cdot\frac{\mathrm{e}^{nz}-1}{z}
\end{aligned}$$

代入 $B(z)=\dfrac{z}{\mathrm{e}^z-1}$：

$$\begin{aligned}F_n(z)&=B(z)\cdot\frac{\mathrm{e}^{nz}-1}{z}\\&=\left(\sum_{i\geq 0}\frac{B_i}{i!}\right)\left(\sum_{i\geq 1}\frac{n^i z^{i-1}}{i!}\right)\\&=\left(\sum_{i\geq 0}\frac{B_i}{i!}\right)\left(\sum_{i\geq 0}\frac{n^{i+1} z^{i}}{(i+1)!}\right)\end{aligned}$$

由于 $F_n(z)=\sum_{m\geq 0}\dfrac{S_m(n)}{m!}z^m$，即 $S_m(n)=m![z^m]F_n(z)$：

$$\begin{aligned}S\times m(n)&=m![z^m]F_n(z)\\&=m!\sum_{i=0}^{m}\frac{B\times i}{i!}\cdot\frac{n^{m-i+1}}{(m-i+1)!}\\&=\frac{1}{m+1}\sum_{i=0}^{m}\binom{m+1}{i}B_i n^{m-i+1}\end{aligned}$$

故得证。

"参考实现"

```
typedef long long ll;
const int maxn = 10000;
const int mod = 1e9 + 7;
ll B[maxn];         // 伯努利数
ll C[maxn][maxn];   // 组合数
ll inv[maxn];       // 逆元（计算伯努利数）

void init() {
  // 预处理组合数
  for (int i = 0; i < maxn; i++) {
    C[i][0] = C[i][i] = 1;
    for (int k = 1; k < i; k++) {
      C[i][k] = (C[i - 1][k] % mod + C[i - 1][k - 1] % mod) % mod;
    }
  }
  // 预处理逆元
  inv[1] = 1;
  for (int i = 2; i < maxn; i++) {
    inv[i] = (mod - mod / i) * inv[mod % i] % mod;
  }
  // 预处理伯努利数
  B[0] = 1;
  for (int i = 1; i < maxn; i++) {
    ll ans = 0;
    if (i == maxn - 1) break;
    for (int k = 0; k < i; k++) {
      ans += C[i + 1][k] * B[k];
      ans %= mod;
    }
    ans = (ans * (-inv[i + 1]) % mod + mod) % mod;
    B[i] = ans;
```

```
    }
}
```

9.14.11 Entringer Number

恩特林格数

恩特林格数（Entringer number，OEIS A008281[1]）$E(n,k)$ 是满足下述条件的 0 到 n 共 $n+1$ 个数的置换数目：

- 首元素是 k；
- 首元素的下一个元素比首元素小，再下一个元素比前一个元素大，再下一个元素比前一个元素小⋯⋯后面相邻元素的大小关系均满足这样的规则。

恩特林格数的初值有：

$$E(0,0)=1$$

$$E(n,0)=0$$

有递推关系：

$$E(n,k)=E(n,k-1)+E(n-1,n-k)$$

Seidel–Entringer–Arnold 三角

恩特林格数的一个适当排列的数字三角，称为 Seidel–Entringer–Arnold 三角（Seidel–Entringer–Arnold triangle，OEIS A008280[2]）。该三角是按照「牛耕」顺序（ox-plowing order）排列的恩特林格数 $E_(n,k)$：

$$\begin{aligned}
&E(0,0)\\
&E(1,0)\rightarrow E(1,1)\\
&E(2,2)\leftarrow E(2,1)\leftarrow E(2,0)\\
&E(3,0)\rightarrow E(3,1)\rightarrow E(3,2)\rightarrow E(3,3)\\
&E(4,4)\leftarrow E(4,3)\leftarrow E(4,2)\leftarrow E(4,1)\leftarrow E(4,0)
\end{aligned}$$

即：

$$\begin{aligned}
&1\\
&0\rightarrow 1\\
&1\leftarrow 1\leftarrow 0\\
&0\rightarrow 1\rightarrow 2\rightarrow 2\\
&5\leftarrow 5\leftarrow 4\leftarrow 2\leftarrow 0
\end{aligned}$$

按照这种方式排列的恩特林格数的优势是，与它的递推关系 $E(n,k)=E(n,k-1)+E(n-1,n-k)$ 一致，可以方便记忆和理解。

恩特林格数有一个指数型生成函数：

$$\sum_{m=0}^{\infty}\sum_{n=0}^{\infty}E\left(m+n,\frac{1}{2}\left(m+n+(-1)^{m+n}(n-m)\right)\right)\frac{x^m}{m!}\frac{x^n}{n!}=\frac{\cos x+\sin x}{\cos(x+y)}$$

这个生成函数的系数分布事实上是上面的 Seidel–Entringer–Arnold 三角的简单拉伸变形：

$$\begin{array}{lllll}
E(0,0) & E(1,1) & E(2,0) & E(3,3) & E(4,0)\\
E(1,0) & E(2,1) & E(3,2) & E(4,1) & \\
E(2,2) & E(3,1) & E(4,2) & & \\
E(3,0) & E(4,3) & & & \\
E(4,4) & & & &
\end{array}$$

即：

$$\begin{array}{ccccc} 1 & 1 & 0 & 2 & 0 \\ 0 & 1 & 2 & 2 & \\ 1 & 1 & 4 & & \\ 0 & 5 & & & \\ 5 & & & & \end{array}$$

zigzag 置换

一个 zigzag 置换（zigzag permutation）是一个 1 到 n 的排列 c_1 到 c_i，使得任意一个元素 c_i 的大小都不介于 c_{i-1} 和 c_{i+1} 之间。

对于 zigzag 置换的个数 Z_n（OEIS A001250[3]），从 $n = 0$ 开始有：

$$1, 1, 2, 4, 10, 32, 122, 544, \cdots$$

例如，前几个 n 的交替置换有：

$$\begin{aligned} n = 1 :& \{1\} \\ n = 2 :& \{1,2\}, \{2,1\} \\ n = 3 :& \{1,3,2\}, \{2,1,3\}, \{2,3,1\}, \{3,1,2\} \\ n = 4 :& \{1,3,2,4\}, \{1,4,2,3\}, \{2,1,4,3\}, \{2,3,1,4\}, \{2,4,1,3\}, \\ & \{3,1,4,2\}, \{3,2,4,1\}, \{3,4,1,2\}, \{4,1,3,2\}, \{4,2,3,1\} \end{aligned}$$

交替置换与 zigzag 数

（注意和「错位排列」进行概念上的区分。）

对于大于 1 的 n，每个 zigzag 置换翻转过来仍旧为 zigzag 置换，可以两两配对，所以必然为偶数。

这里再给出一种配对的方法：将 zigzag 置换分为交替置换（alternating permutation）和反交替置换（reverse alternating permutation）。

交替置换的首元素大于第二个元素，大小关系为：

$$c_1 > c_2 < c_3 > \cdots$$

反交替置换的首元素小于第二个元素，大小关系为：

$$c_1 < c_2 > c_3 < \cdots$$

如果将 1 和 n 位置互换，2 和 $n-1$ 位置互换，以此类推，即可将交替置换与反交替置换两个集合互换。因此，交替置换与反交替置换的个数相等，恰好为 zigzag 置换的一半。

对于大于 1 的 n，记：

$$A_n = \frac{Z_n}{2}$$

定义初值：

$$A_0 = A_1 = 1$$

这里的 A_n 称为 zigzag 数（Euler zigzag number，OEIS A000111[4]），从 $n = 0$ 开始有：

$$1, 1, 1, 2, 5, 16, 61, 272, \cdots$$

接下来试着求解 A_n。

从 1 到 n 之中，选取 k 个数构成子集，有 $\binom{n}{k}$ 种选法。

在这个 k 元子集中，选反交替置换 u，有 A_k 种选法；用全集减掉这个 k 元子集，剩余的 $n-k$ 元子集中，选反交替置换 v，有 A_{n-k} 种选法。

考虑 $n+1$ 元排列 w，将 u 倒置作为开头，接上 $n+1$，再接上 v。那么，w 一定是 zigzag 置换，并且任意一个 $n+1$ 元 zigzag 置换，都可以在 $n+1$ 处截断得到对应的反交替置换 u 和 v，并且不同的 $n+1$ 元 zigzag 置换对应的 u 和 v 不同。

因此有递推关系：

$$2A_{n+1}=\sum_{k=0}^{n}\binom{n}{k}A_kA_{n-k}$$

$$2(n+1)\frac{A_{n+1}}{(n+1)!}=\sum_{k=0}^{n}\frac{A_k}{k!}\frac{A_{n-k}}{(n-k)!}$$

当 n 为 0 时并不满足这个递推式，初值 A_0 和 A_1 都是 1。

可见，这是一个指数型生成函数的卷积。假设 A_n 的指数型生成函数为 y，就有微分方程：

$$2\frac{\mathrm{d}y}{\mathrm{d}x}=y^2+1$$

等式右面加 1 是为了处理 n 为 0 时的特殊情况。该方程的通解为：

$$y=\tan\left(\frac{1}{2}x+C\right)$$

代入第 0 项为 1 之后，可以得到特解：

$$y=\tan x+\sec x$$

正切函数是奇函数，正割函数是偶函数，两者之和构成 zigzag 数的生成函数。

恩特林格数与 zigzag 数的关系

根据恩特林格数的定义，恩特林格数 $E(n,k)$ 是首元素为 k 的 0 到 n 的交替置换个数。因此恩特林格数与 zigzag 数事实上有关系：

$$A_n=E(n,n)$$

将 A_n 称为「zigzag 数」也有原因：记 E_n 是欧拉数（Euler number），B_n 是伯努利数。

当 n 为偶数时，偶数项下标的 zigzag 数也称「正割数」S_n 或者「zig 数」。有关系：

$$A_n=(-1)^{n/2}E_n$$

前几项为（OEIS A000364[5]）：

$$1,1,5,61,1385,\cdots$$

当 n 为奇数时，奇数项下标的 zigzag 数也称「正切数」T_n 或者「zag 数」。有关系：

$$A_n=\frac{(-1)^{(n-1)/2}2^{n+1}(2^{n+1}-1)B_{n+1}}{n+1}$$

前几项为（OEIS A000182[6]）：

$$1,2,16,272,7936,\cdots$$

于是对于在 $x=0$ 处的泰勒展开，可以给出正割数和正切数：

$$\sec x=A_0+A_2\frac{x^2}{2!}+A_4\frac{x^4}{4!}+\cdots$$

$$\tan x=A_1x+A_3\frac{x^3}{3!}+A_5\frac{x^5}{5!}+\cdots$$

或者写到一起：

$$\sec x+\tan x=A_0+A_1x+A_2\frac{x^2}{2!}+A_3\frac{x^3}{3!}+A_4\frac{x^4}{4!}+A_5\frac{x^5}{5!}+\cdots$$

构成 zigzag 数的生成函数。

参考资料与链接

1. Alternating permutation - Wikipedia[7]

参考资料与注释

[1] OEIS A008281

[2] OEIS A008280

[3] OEIS A001250

[4] OEIS A000111

[5] OEIS A000364

[6] OEIS A000182

[7] Alternating permutation - Wikipedia

9.14.12 Eulerian Number

" 注意"

下文中的欧拉数特指 Eulerian number。注意与 Euler number，以及 Euler's number（指与欧拉相关的数学常数例如 γ 或 e）作区分。

在计算组合中，**欧拉数**（Eulerian Number）是从 1 到 n 中正好满足 m 个元素大于前一个元素（具有 m 个「上升」的排列）条件的排列**个数**。定义为：

$$A(n,m) = \left\langle {n \atop m-1} \right\rangle$$

例如，从数字 1 到 3 一共有 4 种排列使得恰好有一个元素比前面的数字大：

排列	满足要求的排列	个数
1 2 3	1, 2 & 2, 3	2
1 3 2	1, 3	1
2 1 3	1, 3	1
2 3 1	2, 3	1
3 1 2	1, 2	1
3 2 1		0

所以按照 $A(n,m)$ 定义：如果 n 等于 3,m 等于 1，欧拉数值为 4，表示共有 4 个有 1 个元素大于前一个元素的排列。

对于 n 和 m 值比较小的欧拉数来说，我们可以直接得到结果：

$A(n,m)$	满足要求的排列	个数
$A(1,0)$	(1)	1
$A(2,0)$	$(2,1)$	1
$A(2,1)$	$(1,2)$	1
$A(3,0)$	$(3,2,1)$	1
$A(3,1)$	$(1,3,2),(2,1,3),(2,3,1),(3,1,2)$	4
$A(3,2)$	$(1,2,3)$	1

公式

可以通过递推或者递归的方法计算欧拉数。

首先，当 $m \geq n$ 或 $n = 0$ 时，没有满足条件的排列，即此时欧拉数为 0。

其次，当 $m = 0$ 时，只有降序的排列满足条件，即此时欧拉数为 1。

最后，考虑在 $n-1$ 的排列的基础上插入 n 从而得到 n 的排列，由于插入 n 至多使欧拉数增加 1，所以 $A(n,m)$ 可以仅从 $A(n-1,m-1)$ 处和 $A(n-1,m)$ 处转移得到。

考虑 n 插入的位置：当 $p_{i-1} < p_i$ 时，若将 n 插到 p_i 之前，即将 n 插入到" 上升" 中，排列的欧拉数不变；此外，将 n 插在排列之前，排列的欧拉数也不变；否则，若将 n 插到其余位置，排列的欧拉数增加 1。

考虑从 $A(n-1,m-1)$ 转移到 $A(n,m)$，此时需要使欧拉数增加 1，此时不能将 n 插在" 上升" 中或者排列开头，共有 $n-(m-1)-1 = n-m$ 种方案。

考虑从 $A(n-1,m)$ 转移到 $A(n,m)$，此时需要欧拉数保持不变，只能将 n 插在" 上升" 中或者排列开头，共 $m+1$ 种方案。

综上所述，有

$$A(n,m) = \begin{cases} 0, & m > n \text{ or } n = 0, \\ 1, & m = 0, \\ (n-m)\cdot A(n-1,m-1) + (m+1)\cdot A(n-1,m), & \text{otherwise.} \end{cases}$$

实现

```cpp
int eulerianNumber(int n, int m) {
  if (m >= n || n == 0) return 0;
  if (m == 0) return 1;
  return (((n - m) * eulerianNumber(n - 1, m - 1)) +
          ((m + 1) * eulerianNumber(n - 1, m)));
}
```

```python
def eulerianNumber(n, m):
    if m >= n or n == 0:
        return 0
    if m == 0:
        return 1
    return (((n - m) * eulerianNumber(n - 1, m - 1)) + \
            ((m + 1) * eulerianNumber(n - 1, m)))
```

习题

- CF1349F1 Slime and Sequences (Easy Version)[1]
- CF1349F2 Slime and Sequences (Hard Version)[2]
- UOJ 593.　新年的军队[3]
- P7511　三到六[4]

参考资料与注释

[1] CF1349F1 Slime and Sequences (Easy Version)

[2] CF1349F2 Slime and Sequences (Hard Version)

[3] UOJ 593. 新年的军队

[4] P7511 三到六

9.14.13　分拆数

分拆：将自然数 n 写成递降正整数和的表示。

$$n = r_1 + r_2 + \ldots + r_k \quad r_1 \geq r_2 \geq \ldots \geq r_k \geq 1$$

和式中每个正整数称为一个部分。

分拆数：p_n。自然数 n 的分拆方法数。

自 0 开始的分拆数：

n	0	1	2	3	4	5	6	7	8
p_n	1	1	2	3	5	7	11	15	22

k 部分拆数

将 n 分成恰有 k 个部分的分拆，称为 k 部分拆数，记作 $p(n,k)$。

显然，k 部分拆数 $p(n,k)$ 同时也是下面方程的解数：

$$n-k = y_1 + y_2 + \ldots + y_k \quad y_1 \geq y_2 \geq \ldots \geq y_k \geq 0$$

如果这个方程里面恰有 j 个部分非 0，则恰有 $p(n-k,j)$ 个解。因此有和式：

$$p(n,k) = \sum_{j=0}^{k} p(n-k,j)$$

相邻两个和式作差，得：

$$p(n,k) = p(n-1,k-1) + p(n-k,k)$$

如果列出表格，每个格里的数，等于左上方的数，加上该格向上方数，所在列数个格子中的数。

k	0	1	2	3	4	5	6	7	8
$p(0,k)$	1	0	0	0	0	0	0	0	0
$p(1,k)$	0	1	0	0	0	0	0	0	0
$p(2,k)$	0	1	1	0	0	0	0	0	0
$p(3,k)$	0	1	1	1	0	0	0	0	0
$p(4,k)$	0	1	2	1	1	0	0	0	0
$p(5,k)$	0	1	2	2	1	1	0	0	0
$p(6,k)$	0	1	3	3	2	1	1	0	0
$p(7,k)$	0	1	3	4	3	2	1	1	0
$p(8,k)$	0	1	4	5	5	3	2	1	1

例题

"计算 k 部分拆数"

计算 k 部分拆数 $p(n,k)$。多组输入，其中 n 上界为 10000，k 上界为 1000，对 1000007 取模。

观察表格与递推式，按列更新对于存储更有利。不难写出程序：

```
#include <stdio.h>
#include <string.h>

int p[10005][1005]; /* 将自然数 n 分拆为 k 个部分的方法数 */

int main() {
  int n, k;
  while (~scanf("%d%d", &n, &k)) {
    memset(p, 0, sizeof(p));
    p[0][0] = 1;
    int i;
    for (i = 1; i <= n; ++i) {
      int j;
      for (j = 1; j <= k; ++j) {
        if (i - j >= 0) /*p[i-j][j] 所有部分大于 1*/
        {
          p[i][j] = (p[i - j][j] + p[i - 1][j - 1]) %
                    1000007; /*p[i-1][j-1] 至少有一个部分为 1。*/
        }
      }
    }
    printf("%d\n", p[n][k]);
  }
}
```

生成函数

由等比数列求和公式，有：

$$\frac{1}{1-x^k} = 1 + x^k + x^{2k} + x^{3k} + \ldots$$

$$1+p_1x+p_2x^2+p_3x^3+...=\frac{1}{1-x}\frac{1}{1-x^2}\frac{1}{1-x^3}\cdots$$

对于 k 部分拆数，生成函数稍微复杂。具体写出如下：

$$\sum_{n,k=0}^{\infty}p(n,k)x^ny^k=\frac{1}{1-xy}\frac{1}{1-x^2y}\frac{1}{1-x^3y}\cdots$$

Ferrers 图

Ferrers 图：将分拆的每个部分用点组成的行表示。每行点的个数为这个部分的大小。

根据分拆的定义，Ferrers 图中不同的行按照递减的次序排放。最长行在最上面。

例如：分拆 $12=5+4+2+1$ 的 Ferrers 图。

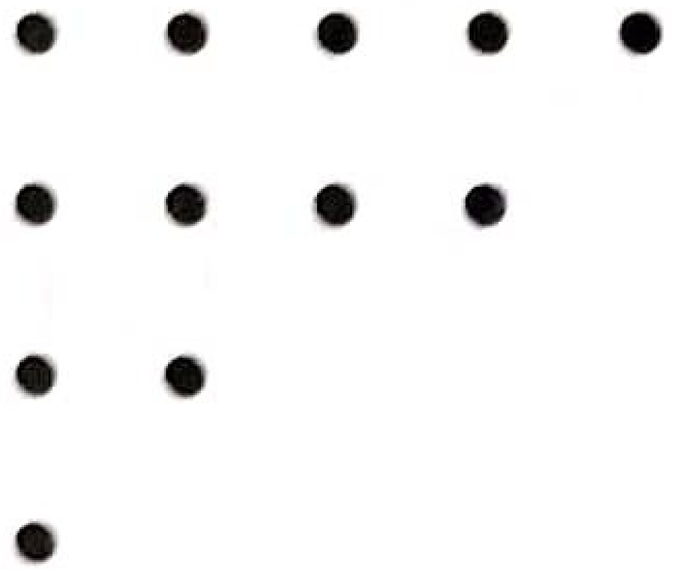


图 9.24

将一个 Ferrers 图沿着对角线翻转，得到的新 Ferrers 图称为原图的共轭，新分拆称为原分拆的共轭。显然，共轭是对称的关系。

例如上述分拆 $12=5+4+2+1$ 的共轭是分拆 $12=4+3+2+2+1$。

最大 k 分拆数：自然数 n 的最大部分为 k 的分拆个数。

根据共轭的定义，有显然结论：

最大 k 分拆数与 k 部分拆数相同，均为 $p(n,k)$。

互异分拆数

互异分拆数：pd_n。自然数 n 的各部分互不相同的分拆方法数。（Different）

n	0	1	2	3	4	5	6	7	8
pd_n	1	1	1	2	2	3	4	5	6

同样地，定义互异 k 部分拆数 $pd(n,k)$，表示最大拆出 k 个部分的互异分拆，是这个方程的解数：

$$n = r_1 + r_2 + \ldots + r_k \quad r_1 > r_2 > \ldots > r_k \geq 1$$

完全同上，也是这个方程的解数：

$$n - k = y_1 + y_2 + \ldots + y_k \quad y_1 > y_2 > \ldots > y_k \geq 0$$

这里与上面不同的是，由于互异，新方程中至多只有一个部分为零。有不变的结论：恰有 j 个部分非 0，则恰有 $pd(n-k,j)$ 个解，这里 j 只取 k 或 $k-1$。因此直接得到递推：

$$pd(n,k) = pd(n-k,k-1) + pd(n-k,k)$$

同样像组合数一样列出表格，每个格里的数，等于该格前一列上数，所在列数个格子中的数，加上该格向上方数，所在列数个格子中的数。

k	0	1	2	3	4	5	6	7	8
$pd(0,k)$	1	0	0	0	0	0	0	0	0
$pd(1,k)$	0	1	0	0	0	0	0	0	0
$pd(2,k)$	0	1	0	0	0	0	0	0	0
$pd(3,k)$	0	1	1	0	0	0	0	0	0
$pd(4,k)$	0	1	1	0	0	0	0	0	0
$pd(5,k)$	0	1	2	0	0	0	0	0	0
$pd(6,k)$	0	1	2	1	0	0	0	0	0
$pd(7,k)$	0	1	3	1	0	0	0	0	0
$pd(8,k)$	0	1	3	2	0	0	0	0	0

例题

" 计算互异分拆数"

计算互异分拆数 pd_n。多组输入，其中 n 上界为 50000，对 1000007 取模。

观察表格与递推式，按列更新对于存储更有利。代码中将后一位缩减了空间，仅保留相邻两项。

```c
#include <stdio.h>
#include <string.h>

int pd[50005][2]; /* 将自然数 n 分拆为 k 个部分的互异方法数 */

int main() {
  int n;
  while (~scanf("%d", &n)) {
    memset(pd, 0, sizeof(pd));
    pd[0][0] = 1;
    int ans = 0;
    int j;
    for (j = 1; j < 350; ++j) {
      int i;
```

```
      for (i = 0; i < 350; ++i) {
        pd[i][j & 1] = 0; /*pd[i][j] 只与 pd[][j] 和 pd[][j-1] 有关 */
      }
      for (i = 0; i <= n; ++i) {
        if (i - j >= 0) /*pd[i-j][j] 所有部分大于 1*/
        {
          pd[i][j & 1] = (pd[i - j][j & 1] + pd[i - j][(j - 1) & 1]) %
                         1000007; /*pd[i-j][j-1] 至少有一个部分为 1。*/
        }
      }
      ans = (ans + pd[n][j & 1]) % 1000007;
    }
    printf("%d\n", ans);
  }
}
```

奇分拆数

奇分拆数：po_n。自然数 n 的各部分都是奇数的分拆方法数。(Odd)

有一个显然的等式：

$$\prod_{i=1}^{\infty}(1+x^i)=\frac{\prod_{i=1}^{\infty}(1-x^{2i})}{\prod_{i=1}^{\infty}(1-x^i)}=\prod_{i=1}^{\infty}\frac{1}{1-x^{2i-1}}$$

最左边是互异分拆数的生成函数，最右边是奇分拆数的生成函数。两者对应系数相同，因此，奇分拆数和互异分拆数相同：

$$po_n=pd_n$$

但显然 k 部奇分拆数和互异 k 部分拆数不是一个概念，这里就不列出了。

再引入两个概念：

互异偶分拆数：pde_n。自然数 n 的部分数为偶数的互异分拆方法数。(Even)

互异奇分拆数：pdo_n。自然数 n 的部分数为奇数的互异分拆方法数。(Odd)

因此有：

$$pd_n=pde_n+pdo_n$$

同样也有相应的 k 部概念。由于过于复杂，不再列出。

五边形数定理

单独观察分拆数的生成函数的分母部分：

$$\prod_{i=1}^{\infty}(1-x^i)$$

将这部分展开，可以想到互异分拆，与互异分拆拆出的部分数奇偶性有关。

具体地，互异偶部分拆在展开式中被正向计数，互异奇部分拆在展开式中被负向计数。因此展开式中各项系数为两方法数之差。即：

$$\sum_{i=0}^{\infty}(pde_n-pdo_n)x^n=\prod_{i=1}^{\infty}(1-x^i)$$

接下来说明，多数情况下，上述两方法数相等，在展开式中系数为 0；仅在少数位置，两方法数相差 1 或 -1。

这里可以借助构造对应的办法。

画出每个互异分拆的 Ferrers 图。最后一行称为这个图的底，底上点的个数记为 b (Bottom)；连接最上面一行的最后一个点与图中某点的最长 45 度角线段，称为这个图的坡，坡上点的个数记为 s (Slide)。

要想在互异偶部分拆与互异奇部分拆之间构造对应，就要定义变换，在保证互异条件不变的前提下，使得行数改变 1：

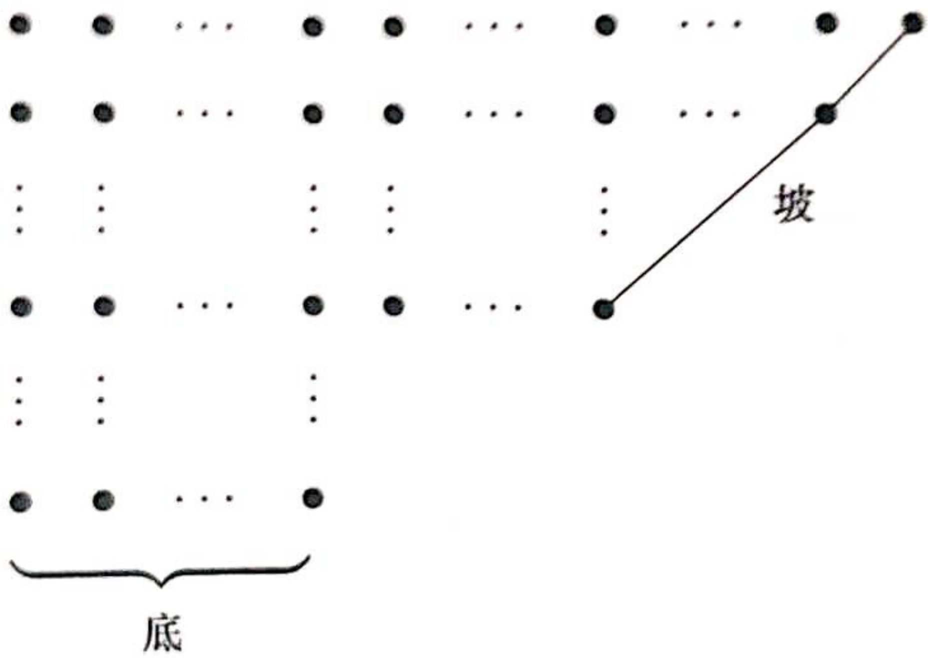


图 9.25

变换 A：当 b 小于等于 s 的时候，就将底移到右边，成为一个新坡。

变换 B：当 b 大于 s 的时候，就将坡移到下边，成为一个新底。

这两个变换，对于多数时候的 n，恰有一个变换可以进行，就在互异偶部分拆与互异奇部分拆之间构造了一个一一对应。已经构造了一一对应的两部分分拆个数相等，因此这时展开式中第 n 项系数为 0。

变换 A 不能进行的条件：底与坡有一个公共点，且 $b=s$。这种情形只发生于：

$$n = b + (b+1) + ... + (b+b-1) = \frac{b(3b-1)}{2}$$

这时，展开式中第 n 项为：

$$\prod_{i=0}^{b-1}(-x)^{b+i} = (-1)^b \prod_{i=0}^{b-1} x^{b+i} = (-1)^b x^n$$

变换 B 不能进行的条件：底与坡有一个公共点，且 $b=s+1$。这种情形只发生于：

$$n = (s+1) + (s+2) + ... + (s+s) = \frac{s(3s-1)}{2}$$

这时，展开式中第 n 项为：

$$\prod_{i=1}^{s}(-x)^{s+i} = (-1)^s \prod_{i=1}^{s} x^{s+i} = (-1)^s x^n$$

至此，我们就证明了：

$$(1-x)(1-x^2)(1-x^3)... = ... + x^{26} - x^{15} + x^7 - x^2 + 1 - x + x^5 - x^{12} + x^{22} - ... = \sum_{k=-\infty}^{+\infty} (-1)^k x^{\frac{k(3k-1)}{2}}$$

将这个式子整理，对比两边各项系数，就得到递推式。

$$(1 + p_1 x + p_2 x^2 + p_3 x^3 + ...)(1 - x - x^2 + x^5 + x^7 - x^{12} - x^{15} + x^{22} + x^{26} - ...) = 1$$

$$p_n = p_{n-1} + p_{n-2} - p_{n-5} - p_{n-7} + ...$$

这个递推式有无限项，但是如果规定负数的分拆数是 0（0 的分拆数已经定义为 1），那么就简化为了有限项。

例题

" 计算分拆数"

计算分拆数 p_n。多组输入，其中 n 上界为 50000，对 1000007 取模。

采用五边形数定理的方法。有代码：

```
#include <stdio.h>

long long a[100010];
long long p[50005];

int main() {
  p[0] = 1;
  p[1] = 1;
  p[2] = 2;
  int i;
  for (i = 1; i < 50005;
       i++) /* 递推式系数 1,2,5,7,12,15,22,26...i*(3*i-1)/2,i*(3*i+1)/2*/
  {
    a[2 * i] = i * (i * 3 - 1) / 2; /* 五边形数为 1,5,12,22...i*(3*i-1)/2*/
    a[2 * i + 1] = i * (i * 3 + 1) / 2;
  }
  for (
      i = 3; i < 50005;
      i++) /*p[n]=p[n-1]+p[n-2]-p[n-5]-p[n-7]+p[12]+p[15]-...+p[n-i*[3i-1]/2]+p[
n-i*[3i+1]/2]*/
  {
    p[i] = 0;
    int j;
    for (j = 2; a[j] <= i; j++) /* 有可能为负数，式中加 1000007*/
    {
      if (j & 2) {
        p[i] = (p[i] + p[i - a[j]] + 1000007) % 1000007;
      } else {
        p[i] = (p[i] - p[i - a[j]] + 1000007) % 1000007;
      }
    }
  }
  int n;
  while (~scanf("%d", &n)) {
    printf("%lld\n", p[n]);
  }
}
```

9.14.14 范德蒙德卷积

引入

范德蒙德卷积是一种合并组合数的式子，主要应用于组合数学的公式推导。

范德蒙德卷积公式

$$\sum_{i=0}^{k}\binom{n}{i}\binom{m}{k-i}=\binom{n+m}{k}$$

证明

考虑用二项式定理证明：

$$\begin{aligned}\sum_{k=0}^{n+m}\binom{n+m}{k}x^k&=(x+1)^{n+m}\\&=(x+1)^n(x+1)^m\\&=\sum_{r=0}^{n}\binom{n}{r}x^r\sum_{s=0}^{m}\binom{m}{s}x^s\\&=\sum_{k=0}^{n+m}\sum_{r=0}^{k}\binom{n}{r}\binom{m}{k-r}x^k\end{aligned}$$

即有：

$$\binom{n+m}{k}=\sum_{r=0}^{k}\binom{n}{r}\binom{m}{k-r}$$

若考虑其组合意义证明：

在一个大小为 $n+m$ 的集合中取出 k 个数，可以等于把大小为 $n+m$ 的集合拆成两个集合，大小分别为 n 与 m，然后从 n 中取出 i 个数，从 m 中取出 $k-i$ 个数的方案数。由于我们有了对于 i 的枚举，于是只需要考虑一种拆法，因为不同的拆法之间是等价的。

推论

推论 1 及证明

$$\sum_{i=-r}^{s}\binom{n}{r+i}\binom{m}{s-i}=\binom{n+m}{r+s}$$

证明与原公式证明相似。

推论 2 及证明

$$\sum_{i=1}^{n}\binom{n}{i}\binom{n}{i-1}=\binom{2n}{n-1}$$

根据基础的组合数学知识推导，有：

$$\sum_{i=1}^{n}\binom{n}{i}\binom{n}{i-1}=\sum_{i=0}^{n-1}\binom{n}{i+1}\binom{n}{i}=\sum_{i=0}^{n-1}\binom{n}{n-1-i}\binom{n}{i}=\binom{2n}{n-1}$$

推论 3 及证明

$$\sum_{i=0}^{n}\binom{n}{i}^2=\binom{2n}{n}$$

根据基础的组合数学知识推导，有：

$$\sum_{i=0}^{n}\binom{n}{i}^2=\sum_{i=0}^{n}\binom{n}{i}\binom{n}{n-i}=\binom{2n}{n}$$

推论 4 及证明

$$\sum_{i=0}^{m}\binom{n}{i}\binom{m}{i}=\binom{n+m}{m}$$

根据基础的组合数学知识推导，有：

$$\sum_{i=0}^{m}\binom{n}{i}\binom{m}{i}=\sum_{i=0}^{m}\binom{n}{i}\binom{m}{m-i}=\binom{n+m}{m}$$

其中 $\binom{n+m}{m}$ 是我们较为熟悉的网格图路径计数的方案数。所以我们可以考虑其组合意义的证明。

在一张网格图中，从 $(0,0)$ 走到 (n,m) 共走 $n+m$ 步。规定 $(0,0)$ 位于网格图左上角，其中向下走了 n 步，向右走了 m 步，方案数为 $\binom{n+m}{m}$。

换个视角，我们将 $n+m$ 步拆成两部分走，先走 n 步，再走 m 步，那么 n 步中若有 i 步向右，则 m 步中就有 $m-i$ 步向右，故得证。

习题

- CF785D Anton and School - 2[1]
- 洛谷 P2791 幼儿园篮球题[2]

参考资料与注释

1. Vandermonde's Convolution Formula[3]

参考资料与注释

[1] CF785D Anton and School - 2

[2] 洛谷 P2791 幼儿园篮球题

[3] Vandermonde's Convolution Formula

9.14.15 图论计数

在组合数学中，图论计数（Graph Enumeration）是研究满足特定性质的图的计数问题的分支。生成函数、波利亚计数定理 与 符号化方法 和 OEIS[3] 是解决这类问题时最重要的数学工具。图论计数可分为有标号和无标号两大类问题，大多数情况下[1] 有标号版本的问题都比其对应的无标号问题更加简单，因此我们将先考察有标号问题的计数。

有标号树

即 Cayley 公式，参见 Prüfer 序列 一文，我们也可以使用 Kirchhoff 矩阵树定理 或 生成函数 和 拉格朗日定理[4] 得到这一结果。

习题

- Hihocoder 1047. Random Tree[5]

有标号连通图

例题「POJ 1737」Connected Graph

" 例题 「POJ 1737」Connected Graph[6]"

题目大意：求有 n 个结点的有标号连通图的方案数（$n \le 50$）。

这类问题最早出现于楼教主的男人八题系列中，我们设 g_n 为 n 个点有标号图的方案数，c_n 为待求序列。n 个点的图至多有 $\binom{n}{2}$ 条边，每条边根据其出现与否有两种状态，每种状态之间独立，因而有 $g_n = 2^{\binom{n}{2}}$。我们固定其中一个节点，枚举其所在连通块的大小，那么还需要从剩下的 $n-1$ 个节点中选择 $i-1$ 个节点组成一个连通块。连通块之外的节点可以任意连边，因而有如下递推关系：

$$\sum_{i=1}^{n} \binom{n-1}{i-1} c_i g_{n-i} = g_n \tag{9.22}$$

$$c_n = g_n - \sum_{i=1}^{n-1} \binom{n-1}{i-1} c_i g_{n-i} \tag{9.23}$$

移项得到 c_n 序列的 $O(n^2)$ 递推公式，可以通过此题。

例题「集训队作业 2013」城市规划

" 例题 「 集训队作业 2013」 城市规划[7]"

题目大意：求有 n 个结点的有标号连通图的方案数（$n \le 130000$）。

对于数据范围更大的序列问题，往往我们需要构造这些序列的生成函数，以使用高效的多项式算法。

方法一：分治 FFT

上述的递推式可以看作一种自卷积形式，因而可以使用分治 FFT 进行计算，复杂度 $O(n \log^2 n)$。

方法二：多项式求逆

我们将上述递推式中的组合数展开，并进行变形：

$$\sum_{i=1}^{n} \binom{n-1}{i-1} c_i g_{n-i} = g_n \tag{9.24}$$

$$\sum_{i=1}^{n} \frac{c_i}{(i-1)!} \frac{g_{n-i}}{(n-i)!} = \frac{g_n}{(n-1)!} \tag{9.25}$$

构造多项式：

$$C(x) = \sum_{n=1} \frac{c_n}{(n-1)!} x^n \tag{9.26}$$

$$G(x) = \sum_{n=0} \frac{g_n}{n!} x^n \tag{9.27}$$

$$H(x) = \sum_{n=1} \frac{g_n}{(n-1)!} x^n \tag{9.28}$$

代换进上式得到 $CG = H$，使用 多项式求逆 后再卷积解出 $C(x)$ 即可。

方法三：多项式 exp

另一种做法是使用 EGF 中多项式 exp 的组合意义，我们设有标号连通图和简单图序列的 EGF 分别为 $C(x)$ 和 $G(x)$，那么它们将有下列关系：

$$\exp(C(x)) = G(x) \tag{9.29}$$

$$C(x) = \ln(G(x)) \tag{9.30}$$

使用 多项式 ln 解出 $C(x)$ 即可。

有标号欧拉图、二分图

例题「SPOJ KPGRAPHS」Counting Graphs

" **例题** 「SPOJ KPGRAPHS」Counting Graphs[8]"

题目大意：求有 n 个结点的分别满足下列性质的有标号图的方案数（$n \leq 1000$）。

- 连通图 A001187[9]。
- 欧拉图 A033678[10]。
- 二分图 A047864[11]。

本题限制代码长度，因而无法直接使用多项式模板，但生成函数依然可以帮助我们进行分析。

连通图问题在之前的例题中已被解决，考虑欧拉图。注意到上述对连通图计数的几种方法，均可以在满足任意性质的有标号连通图进行推广。例如我们可以将连通图递推公式中的 g_n，从任意图替换成满足顶点度数均为偶数的图，此时得到的 c_n 即为欧拉图。

我们将 POJ 1737 的递推过程封装成连通化函数，

```
void ln(Int C[], Int G[]) {
  for (int i = 1; i <= n; ++i) {
    C[i] = G[i];
    for (int j = 1; j <= i - 1; ++j)
      C[i] -= binom[i - 1][j - 1] * C[j] * G[i - j];
  }
}
```

前两问即可轻松解决：

```
for (int i = 1; i <= n; ++i) G[i] = pow(2, binom[i][2]);
ln(C, G);
for (int i = 1; i <= n; ++i) G[i] = pow(2, binom[i - 1][2]);
ln(E, G);
```

注意到这里的连通化递推过程其实等价于对其 EGF 求多项式 ln，同理我们也可以写出逆连通化函数，它等价于对其 EGF 求多项式 exp。

```
void exp(Int G[], Int C[]) {
  for (int i = 1; i <= n; ++i) {
    G[i] = C[i];
    for (int j = 1; j <= i - 1; ++j)
      G[i] += binom[i - 1][j - 1] * C[j] * G[i - j];
  }
}
```

下面讨论有标号二分图计数，

我们设 b_n 表示 n 个结点的二分图方案数，g_n 表示 n 个结点对结点进行 2 染色，满足相同颜色的结点之间不存在边的图的方案数。枚举其中一种颜色节点的数量，有[2-1]：

$$g_n = \sum_{i=0}^{n} \binom{n}{i} 2^{i(n-i)}$$

接下来我们用两种不同的方法建立 g_n 与 b_n 之间的关系。

方法一：算两次

我们设 $c_{n,k}$ 表示有 k 个连通分量的二分图方案数，那么不难得到如下关系：

$$b_n = \sum_{i=1}^{n} c_{n,i} \tag{9.31}$$

$$g_n = \sum_{i=1}^{n} c_{n,i} 2^i \tag{9.32}$$

比较两种 g_n 的表达式，展开得：

$$\sum_{i=0}^{n} \binom{n}{i} 2^{i(n-i)} = \sum_{i=1}^{n} c_{n,i} 2^i \tag{9.33}$$

$$c_{n,i} = \sum_{i=0} n-1 \binom{n-1}{i-1} c_{n,1} c_{n-i,k-1} \tag{9.34}$$

不难得到 b_n 的递推关系，复杂度 $O(n^3)$，进一步使用容斥原理，可以优化到 $O(n^2)$ 通过本题。

方法二：连通化递推

方法二和方法三均使用连通二分图 $b1_n$ A001832[12] 来建立 g_n 与 b_n 之间的桥梁。

注意到对于每个连通二分图，我们恰好有两种不同的染色方法，对应到两组不同的连通 2 染色图，因而对 g_n 进行连通化，得到的序列恰好是 $b1_n$ 的两倍，而 b_n 则由 $b1_n$ 进行逆连通化得到。

因此：

```
for (int i = 1; i <= n; ++i) {
  G[i] = 0;
  for (int j = 0; j < i + 1; ++j) G[i] += binom[i][j] * pow(2, j * (i - j));
}
ln(B1, G);
for (int i = 1; i <= n; ++i) B1[i] /= 2;
exp(B, B1);
```

两种递推的过程复杂度均为 $O(n^2)$，可以通过本题。

方法三：多项式 exp

我们注意到也可以使用 EGF 理解上面的递推过程。

设 $G(x)$ 为 g_n 的 EGF，$B1(x)$ 为 $b1_n$ 的 EGF，$B(x)$ 为 b_n 的 EGF，应用做法二的方法，我们有：

$$G(x) = \exp(2B1(x)) \tag{9.35}$$

$$B(x) = \exp(B1(x)) \tag{9.36}$$

$$= \exp(\frac{\ln G(x)}{2}) \tag{9.37}$$

$$= \sqrt{G} \tag{9.38}$$

我们可以对等式两边分别进行求导并比较两边系数，以得到易于编码的递推公式，通过此题。注意到做法二与做法三本质相同，且一般情况下做法三可以得到更优的时间复杂度。

$$B_n^2 = G \tag{9.39}$$

$$2B_n B_n' = G' \tag{9.40}$$

参考代码

```
#include <bits/stdc++.h>
using namespace std;
```

```
#define Ts *this
#define rTs return Ts
typedef long long LL;
const int MOD = int(1e9) + 7;

// <<= '2. Number Theory .,//{
namespace NT {
void INC(int& a, int b) {
  a += b;
  if (a >= MOD) a -= MOD;
}

int sum(int a, int b) {
  a += b;
  if (a >= MOD) a -= MOD;
  return a;
}

void DEC(int& a, int b) {
  a -= b;
  if (a < 0) a += MOD;
}

int dff(int a, int b) {
  a -= b;
  if (a < 0) a += MOD;
  return a;
}

void MUL(int& a, int b) { a = (LL)a * b % MOD; }

int pdt(int a, int b) { return (LL)a * b % MOD; }

int _I(int b) {
  int a = MOD, x1 = 0, x2 = 1, q;
  while (1) {
    q = a / b, a %= b;
    if (!a) return x2;
    DEC(x1, pdt(q, x2));

    q = b / a, b %= a;
    if (!b) return x1;
    DEC(x2, pdt(q, x1));
  }
}

void DIV(int& a, int b) { MUL(a, _I(b)); }

int qtt(int a, int b) { return pdt(a, _I(b)); }

inline int pow(int a, LL b) {
  int c(1);
  while (b) {
    if (b & 1) MUL(c, a);
```

```
    MUL(a, a), b >>= 1;
  }
  return c;
}

template <class T>
inline T pow(T a, LL b) {
  T c(1);
  while (b) {
    if (b & 1) c *= a;
    a *= a, b >>= 1;
  }
  return c;
}

template <class T>
inline T pow(T a, int b) {
  return pow(a, (LL)b);
}

struct Int {
  int val;

  operator int() const { return val; }

  Int(int _val = 0) : val(_val) {
    val %= MOD;
    if (val < 0) val += MOD;
  }

  Int(LL _val) : val(_val) {
    _val %= MOD;
    if (_val < 0) _val += MOD;
    val = _val;
  }

  Int& operator+=(const int& rhs) {
    INC(val, rhs);
    rTs;
  }

  Int operator+(const int& rhs) const { return sum(val, rhs); }

  Int& operator-=(const int& rhs) {
    DEC(val, rhs);
    rTs;
  }

  Int operator-(const int& rhs) const { return dff(val, rhs); }

  Int& operator*=(const int& rhs) {
    MUL(val, rhs);
    rTs;
  }
```

```
  Int operator*(const int& rhs) const { return pdt(val, rhs); }

  Int& operator/=(const int& rhs) {
    DIV(val, rhs);
    rTs;
  }

  Int operator/(const int& rhs) const { return qtt(val, rhs); }

  Int operator-() const { return MOD - *this; }
};

}  // namespace NT

using namespace NT;

const int N = int(1e3) + 9;
Int binom[N][N], C[N], E[N], B[N], B1[N], G[N];
int n;

void ln(Int C[], Int G[]) {
  for (int i = 1; i <= n; ++i) {
    C[i] = G[i];
    for (int j = 1; j <= i - 1; ++j)
      C[i] -= binom[i - 1][j - 1] * C[j] * G[i - j];
  }
}

void exp(Int G[], Int C[]) {
  for (int i = 1; i <= n; ++i) {
    G[i] = C[i];
    for (int j = 1; j <= i - 1; ++j)
      G[i] += binom[i - 1][j - 1] * C[j] * G[i - j];
  }
}

int main() {
#ifndef ONLINE_JUDGE
  // freopen("in.txt", "r", stdin);
#endif

  n = 1000;
  for (int i = 0; i < n + 1; ++i) {
    binom[i][0] = 1;
    for (int j = 0; j < i; ++j)
      binom[i][j + 1] = binom[i - 1][j] + binom[i - 1][j + 1];
  }

  for (int i = 1; i <= n; ++i) G[i] = pow(2, binom[i][2]);
  ln(C, G);
  for (int i = 1; i <= n; ++i) G[i] = pow(2, binom[i - 1][2]);
  ln(E, G);
  for (int i = 1; i <= n; ++i) {
```

```
    G[i] = 0;
    for (int j = 0; j < i + 1; ++j) G[i] += binom[i][j] * pow(2, j * (i - j));
  }
  ln(B1, G);
  for (int i = 1; i <= n; ++i) B1[i] /= 2;
  exp(B, B1);

  int T;
  cin >> T;
  while (T--) {
    scanf("%d", &n);
    printf("Connected: %d\n", C[n]);
    printf("Eulerian: %d\n", E[n]);
    printf("Bipartite: %d\n", B[n]);
    puts("");
  }
}
```

习题

- UOJ Goodbye Jihai D. 新年的追逐战[13]
- BZOJ 3864. 大朋友和多叉树[14]
- BZOJ 2863. 愤怒的元首[15]
- Luogu P6295. 有标号 DAG 计数[16]
- LOJ 6569. 仙人掌计数[17]
- LOJ 6570. 毛毛虫计数[18]
- Luogu P5434. 有标号荒漠计数[19]
- Luogu P3343. [ZJOI2015] 地震后的幻想乡[20]
- HDU 5279. YJC plays Minecraft[21]
- Luogu P7364. 有标号二分图计数[22]
- Luogu P5827. 点双连通图计数[23]
- Luogu P5827. 边双连通图计数[24]
- Luogu P6596. How Many of Them[25]
- Luogu U152448. 有标号强连通图计数[26]
- Project Euler 434. Rigid graphs[27]

Riddell's Formula

上述关于 EGF 的 exp 的用法，有时又被称作 Riddell's formula for labeled graphs，生成函数的 欧拉变换，有时也被称为 Riddell's formula for unlabeled graphs，后者最早出现在欧拉对分拆数的研究中，除了解决图论计数问题之外，也在完全背包问题中出现。

对于给定序列 a_i，和对应的 OGF $A(x)$，定义 $A(x)$ 的欧拉变换为：

$$\mathcal{E}(A(x)) = \prod_i (1 - x^i)^{-a_i} \tag{9.41}$$

$$= \exp\left(\sum_i \frac{A(x^i)}{i}\right) \tag{9.42}$$

设 $\mathcal{E}(A(x))$ 的各项系数为 b_i，定义辅助数组 $c_i=\sum_{d|n} da_d$，则有递推公式

$$nb_n=c_n+\sum_{i=1}^{n-1}c_ib_{n-i}$$

无标号树

例题「SPOJ PT07D」Let us count 1 2 3

" **例题 「SPOJ PT07D」Let us count 1 2 3**[28]"

题目大意：求有 n 个结点的分别满足下列性质的树的方案数。

- 有标号有根树 A000169[29]。
- 有标号无根树 A000272[30]。
- 无标号有根树 A000081[31]。
- 无标号无根树 A000055[32]。

有根树

有标号情况以在前文中解决，下面考察无标号有根树，设其 OGF 为 $F(x)$，应用欧拉变换，可得：

$$F(x)=x\mathcal{E}(F(x))$$

取出系数即可。

无根树

考虑容斥，我们用有根树的方案中减去根不是重心的方案，并对 n 的奇偶性进行讨论。

当 n 是奇数时：

必然存在一棵子树大小 $\ge\lceil\frac{n}{2}\rceil$，枚举这棵子树的大小有。

$$g_n=f_n-\sum_{i=\lceil\frac{n}{2}\rceil}^{n-1}f_if_{n-i}$$

当 n 是偶数时：

注意到当有两个重心的情况时，上面的过程只会减去一次，因此还需要减去

$$g_n=f_n-\sum_{i=\lceil\frac{n}{2}\rceil}^{n-1}f_if_{n-i}-\binom{f_{\frac{n}{2}}}{2}$$

例题「Luogu P5900」无标号无根树计数

" **例题 「Luogu P5900」 无标号无根树计数**[33]"

题目大意：求有 n 个结点的无标号无根树的方案数（$n\le 200000$）。

对于数据范围更大的情况，做法同理，欧拉变换后使用多项式模板即可。

无标号简单图

例题「SGU 282. Isomorphism」Isomorphism

" **例题** 「SGU 282. Isomorphism」Isomorphism[34]"

题目大意：求有 n 个结点的无标号完全图的边进行 m 染色的方案数。

注意到当 $m = 2$ 时，所求对象就是无标号简单图 A000088[35]，考察波利亚计数定理，

$$\frac{1}{|G|}\sum_{g\in G} m^{c(g)}$$

本题中置换群 G 为顶点的 n 阶对称群生成的边集置换群，但暴力做法的枚举量为 $O(n!)$，无法通过此题。

考虑根据按照置换的循环结构进行分类，每种循环结构对应一种数的分拆，我们用 dfs() 生成分拆，那么问题即转化为求每一种分拆 p 所对应的置换数目 $w(p)$ 和每一类置换中的循环个数 $c(p)$，答案为

$$\frac{1}{|G|}\sum_{p\in P} w(p)m^{c(p)}$$

考虑 $w(p)$，每一个分拆对应一个循环排列，同时同一种大小的分拆之间的顺序无关，因而我们有：

$$w(p) = \frac{n!}{\prod_i(p_i)\prod_i(q_i!)}$$

这里 q_i 表示大小为 i 的分拆在 p 中出现的次数。

考虑 $c(p)$，p 所影响的点集的循环即为 $|p|$，但题目考察的是边染色，所以还需要考察点置换所生成的边置换，

如果一条边关联的顶点处在同一个循环内，设该循环大小为 p_i，那么边所生成的循环数恰好为 $\lfloor \frac{p_i}{2} \rfloor$。

如果一条边关联的顶点处在两个不同的循环中，设分别为 p_i,p_j，每个循环节的长度均为 $\operatorname{lcm}(p_i, p_j)$，因而边所生成的循环数恰好为 $\frac{p_ip_j}{\operatorname{lcm}(p_i,p_j)} = \gcd(p_i, p_j)$。

参考代码

```cpp
#include <bits/stdc++.h>
using namespace std;

#define Ts *this
#define rTs return Ts
typedef long long LL;
int MOD = int(1e9) + 7;

namespace NT {
void INC(int& a, int b) {
  a += b;
  if (a >= MOD) a -= MOD;
}

int sum(int a, int b) {
  a += b;
  if (a >= MOD) a -= MOD;
  return a;
}

void DEC(int& a, int b) {
  a -= b;
  if (a < 0) a += MOD;
}

int dff(int a, int b) {
```

```
  a -= b;
  if (a < 0) a += MOD;
  return a;
}

void MUL(int& a, int b) { a = (LL)a * b % MOD; }

int pdt(int a, int b) { return (LL)a * b % MOD; }

int _I(int b) {
  int a = MOD, x1 = 0, x2 = 1, q;
  while (1) {
    q = a / b, a %= b;
    if (!a) return x2;
    DEC(x1, pdt(q, x2));

    q = b / a, b %= a;
    if (!b) return x1;
    DEC(x2, pdt(q, x1));
  }
}

void DIV(int& a, int b) { MUL(a, _I(b)); }

int qtt(int a, int b) { return pdt(a, _I(b)); }

inline int pow(int a, LL b) {
  int c(1);
  while (b) {
    if (b & 1) MUL(c, a);
    MUL(a, a), b >>= 1;
  }
  return c;
}

template <class T>
inline T pow(T a, LL b) {
  T c(1);
  while (b) {
    if (b & 1) c *= a;
    a *= a, b >>= 1;
  }
  return c;
}

template <class T>
inline T pow(T a, int b) {
  return pow(a, (LL)b);
}

struct Int {
  int val;

  operator int() const { return val; }
```

```
  Int(int _val = 0) : val(_val) {
    val %= MOD;
    if (val < 0) val += MOD;
  }

  Int(LL _val) : val(_val) {
    _val %= MOD;
    if (_val < 0) _val += MOD;
    val = _val;
  }

  Int& operator+=(const int& rhs) {
    INC(val, rhs);
    rTs;
  }

  Int operator+(const int& rhs) const { return sum(val, rhs); }

  Int& operator-=(const int& rhs) {
    DEC(val, rhs);
    rTs;
  }

  Int operator-(const int& rhs) const { return dff(val, rhs); }

  Int& operator*=(const int& rhs) {
    MUL(val, rhs);
    rTs;
  }

  Int operator*(const int& rhs) const { return pdt(val, rhs); }

  Int& operator/=(const int& rhs) {
    DIV(val, rhs);
    rTs;
  }

  Int operator/(const int& rhs) const { return qtt(val, rhs); }

  Int operator-() const { return MOD - *this; }
};

}  // namespace NT

using namespace NT;

const int N = int(5e1) + 9;
Int Fact[N];
vector<vector<int>> Partition;
vector<int> cur;
int n, m;

void gen(int n = ::n, int s = 1) {
```

```
  if (!n) {
    Partition.push_back(cur);
  } else if (n >= s) {
    cur.push_back(s);
    gen(n - s, s);
    cur.pop_back();
    gen(n, s + 1);
  }
}

Int w(const vector<int> P) {
  Int z = Fact[n];
  int c = 0, l = P.front();

  for (auto p : P) {
    z /= p;
    if (p != l) {
      z /= Fact[c];
      l = p;
      c = 1;
    } else {
      ++c;
    }
  }

  z /= Fact[c];
  return z;
}

int c(const vector<int> P) {
  int z = 0;
  for (int i = 0; i < P.size(); ++i) {
    z += P[i] / 2;
    for (int j = 0; j < i; ++j) z += gcd(P[i], P[j]);
  }
  return z;
}

int main() {
  cin >> n >> m >> MOD;
  Fact[0] = 1;
  for (int i = 1; i <= n; ++i) Fact[i] = Fact[i - 1] * i;

  gen();

  Int res = 0;
  for (auto P : Partition) {
    res += w(P) * pow(m, c(P));
  }
  res /= Fact[n];
  cout << res << endl;
}
```

习题

- CodeForces 438 E. The Child and Binary Tree[36]
- Luogu P5448. [THUPC2018] 好图计数[37]
- Luogu P5818. [JSOI2011] 同分异构体计数[38]
- Luogu P6597. 烯烃计数[39]
- Luogu P6598. 烷烃计数[40]
- Luogu P4128. [SHOI2006] 有色图[41]
- Luogu P4727. [HNOI2009] 图的同构计数[42]
- AtCoder Beginner Contest 222 H. Binary Tree[43]
- AtCoder Beginner Contest 284 Ex. Count Unlabeled Graphs[44]
- Luogu P4708. 画画[45]
- Luogu P7592. 数树（2021 CoE-II E）[46]
- Luogu P5206. [WC2019] 数树[47]

参考资料与注释

1. WC2015，顾昱洲营员交流资料 Graphical Enumeration[48]
2. WC2019，生成函数，多项式算法与图的计数[49]
3. Counting labeled graphs - Algorithms for Competitive Programming[50]
4. Graphical Enumeration Paperback, Frank Harary, Edgar M. Palmer[51]
5. The encyclopedia of integer sequences, N. J. A. Sloane, Simon Plouffe[52]
6. Combinatorial Problems and Exercises, László Lovász[53]
7. Graph Theory and Additive Combinatorics[54]

[1] 也许无标号二叉树是一个反例，在结构简单的情况下，对应的置换群是恒等群（Identity Group），此时有标号版本可以直接通过乘以 $n!$ 得到。

[2] 粉兔的 blog 告诉我们，这个序列也可以使用 Chirp Z-Transform 优化。[2-1] [2-2]

[3] OEIS

[4] 拉格朗日定理

[5] Hihocoder 1047. Random Tree

[6] 「POJ 1737」Connected Graph

[7] 「集训队作业 2013」城市规划

[8] 「SPOJ KPGRAPHS」Counting Graphs

[9] A001187

[10] A033678

[11] A047864

[12] A001832

[13] UOJ Goodbye Jihai D. 新年的追逐战

[14] BZOJ 3864. 大朋友和多叉树

[15] BZOJ 2863. 愤怒的元首

[16] Luogu P6295. 有标号 DAG 计数

[17] LOJ 6569. 仙人掌计数

[18] LOJ 6570. 毛毛虫计数

[19] Luogu P5434. 有标号荒漠计数

[20] Luogu P3343. [ZJOI2015] 地震后的幻想乡

[21] HDU 5279. YJC plays Minecraft

[22] Luogu P7364. 有标号二分图计数

[23] Luogu P5827. 点双连通图计数

[24] Luogu P5827. 边双连通图计数

[25] Luogu P6596. How Many of Them

[26] Luogu U152448. 有标号强连通图计数

[27] Project Euler 434. Rigid graphs

[28] 「SPOJ PT07D」Let us count 1 2 3

[29] A000169

[30] A000272

[31] A000081

[32] A000055

[33] 「Luogu P5900」无标号无根树计数

[34] 「SGU 282. Isomorphism」Isomorphism

[35] A000088

[36] CodeForces 438 E. The Child and Binary Tree

[37] Luogu P5448. [THUPC2018] 好图计数

[38] Luogu P5818. [JSOI2011] 同分异构体计数

[39] Luogu P6597. 烯烃计数

[40] Luogu P6598. 烷烃计数

[41] Luogu P4128. [SHOI2006] 有色图

[42] Luogu P4727. [HNOI2009] 图的同构计数

[43] AtCoder Beginner Contest 222 H. Binary Tree

[44] AtCoder Beginner Contest 284 Ex. Count Unlabeled Graphs

[45] Luogu P4708. 画画

[46] Luogu P7592. 数树（2021 CoE-II E）

[47] Luogu P5206. [WC2019] 数树

[48] WC2015, 顾昱洲营员交流资料 Graphical Enumeration

[49] WC2019, 生成函数，多项式算法与图的计数

[50] Counting labeled graphs - Algorithms for Competitive Programming

[51] Graphical Enumeration Paperback, Frank Harary, Edgar M. Palmer

[52] The encyclopedia of integer sequences, N. J. A. Sloane, Simon Plouffe

[53] Combinatorial Problems and Exercises, László Lovász

[54] Graph Theory and Additive Combinatorics

9.15 线性代数

9.15.1 线性代数简介

Authors: codewasp942

" 提示"

本篇与「线性代数」分类下的其他篇目关联不大。但笔者认为，讲讲线性代数的本质，追溯概念的根源与联系，让读者对于线性代数有一个初步但是成体系的认识，确实有其必要性。

早在几千年前，就有古人应用线性方程组解决问题，而如今，线性代数仍然应用广泛。

线性代数源于人们的观察。人们发现，很多对象都拥有相似的性质，比如：

- 力可以被分解、合成。
- 对于任意的 k, x_0，$k\sin(x-x_0)$ 可以分解成 $k_1\sin x + k_2\cos x$。

这些性质与所描述对象的**缩放**、**分解**、**叠加**等有关。线性代数把这些性质从具体对象中抽象出来，作为一个独立的学科来研究。在 OI 中，线性代数的知识可以直接用来解决问题，也可以用于优化算法、数据结构等。例如：

- 用树剖维护线性基求链上最大异或和
- 利用矩阵树定理把图的生成树计数问题转化为求矩阵的行列式
- 用矩阵快速幂优化递推

9.15.2 向量

在本文之前，特别说明一下翻译的相关问题。由于历史原因，数学学科和物理学科关于「vector」一词的翻译不同。

在物理学科，一般翻译成「矢量」，并且与「标量」一词相对。在数学学科，一般翻译成「向量」。这种翻译的差别还有「本征」与「特征」、「幺正」与「酉」，等等。

在 **OI Wiki**，主要面向计算机等工程类相关学科，与数学学科关系更近一些，因此采用「向量」这个词汇。

定义及相关概念

向量：既有大小又有方向的量称为向量。数学上研究的向量为**自由向量**，即只要不改变它的大小和方向，起点和终点可以任意平行移动的向量。记作 $\vec{a}$ 或 $\boldsymbol{a}$。

有向线段：带有方向的线段称为有向线段。有向线段有三要素：**起点，方向，长度**，知道了三要素，终点就唯一确定。一般使用有向线段表示向量。

向量的模：有向线段 $\overrightarrow{AB}$ 的长度称为向量的模，即为这个向量的大小。记为：$|\overrightarrow{AB}|$ 或 $|\boldsymbol{a}|$。

零向量：模为 0 的向量。零向量的方向任意。记为：$\vec{0}$ 或 $\mathbf{0}$。

单位向量：模为 1 的向量称为该方向上的单位向量。一般记为 $\vec{e}$ 或 $\boldsymbol{e}$。

平行向量：方向相同或相反的两个**非零**向量。记作：$\boldsymbol{a} \parallel \boldsymbol{b}$。对于多个互相平行的向量，可以任作一条直线与这些向量平行，那么任一组平行向量都可以平移到同一直线上，所以平行向量又叫**共线向量**。

相等向量：模相等且方向相同的向量。

相反向量：模相等且方向相反的向量。

向量的夹角：已知两个非零向量 $\boldsymbol{a}, \boldsymbol{b}$，作 $\overrightarrow{OA} = \boldsymbol{a}, \overrightarrow{OB} = \boldsymbol{b}$，那么 $\theta = \angle AOB$ 就是向量 $\boldsymbol{a}$ 与向量 $\boldsymbol{b}$ 的夹角。记作：$\langle \boldsymbol{a}, \boldsymbol{b} \rangle$。显然当 $\theta = 0$ 时两向量同向，$\theta = \pi$ 时两向量反向，$\theta = \frac{\pi}{2}$ 时两向量垂直，记作 $\boldsymbol{a} \perp \boldsymbol{b}$，并且规定 $\theta \in [0, \pi]$。

注意到平面向量具有方向性，两个向量不能比较大小（但可以比较两向量的模长）。但是两个向量可以相等。

向量的线性运算

向量的加减法

在定义了一种量之后，就希望让它具有运算。向量的运算可以类比数的运算，从物理学的角度出发也可以研究向量的运算。

类比物理学中的位移概念，假如一个人从 A 经 B 走到 C，那么他经过的位移为 $\overrightarrow{AB}+\overrightarrow{BC}$，这其实等价于这个人直接从 A 走到 C，即 $\overrightarrow{AB}+\overrightarrow{BC}=\overrightarrow{AC}$。

注意到力的合成法则——平行四边形法则，同样也可以看做一些向量相加。

整理一下向量的加法法则：

1. **向量加法的三角形法则**：若要求和的向量首尾顺次相连，那么这些向量的和为第一个向量的起点指向最后一个向量的终点；
2. **向量加法的平行四边形法则**：若要求和的两个向量**共起点**，那么它们的和向量为以这两个向量为邻边的平行四边形的对角线，起点为两个向量共有的起点，方向沿平行四边形对角线方向。

这样，向量的加法就具有了几何意义。并且可以验证，向量的加法满足**交换律与结合律**。

因为实数的减法可以写成加上相反数的形式，考虑在向量做减法时也这么写。即：$a-b=a+(-b)$。

这样，考虑共起点的向量，按照平行四边形法则做出它们的差，经过平移后可以发现「**共起点向量的差向量」是由「减向量」指向「被减向量」的有向线段**。这也是向量减法的几何意义。

有时候有两点 A,B，想知道 $\overrightarrow{AB}$，可以利用减法运算 $\overrightarrow{AB}=\overrightarrow{OB}-\overrightarrow{OA}$ 获得。

向量的数乘

规定「实数 λ 与向量 a 的积」为一个向量，这种运算就是向量的**数乘运算**，记作 λa，它的长度与方向规定如下：

1. $|\lambda a|=|\lambda||a|$；
2. 当 $\lambda>0$ 时，λa 与 a 同向，当 $\lambda=0$ 时，$\lambda a=0$，当 $\lambda<0$ 时，λa 与 a 方向相反。

根据数乘的定义，可以验证有如下运算律：

$$\lambda(\mu a)=(\lambda\mu)a$$
$$(\lambda+\mu)a=\lambda a+\mu a$$
$$\lambda(a+b)=\lambda a+\lambda b$$

特别地：

$$(-\lambda)a=-(\lambda a)=-\lambda(a)$$
$$\lambda(a-b)=\lambda a-\lambda b$$

判定两向量共线

两个**非零**向量 a 与 b 共线 $\iff$ 有唯一实数 λ，使得 $b=\lambda a$。

证明：由数乘的定义可知，对于**非零**向量 a，如果存在实数 λ，使得 $b=\lambda a$，那么 $a\parallel b$。

反过来，如果 $a\parallel b$，$a\neq 0$，且 $|b|=\mu|a|$，那么当 a 与 b 同向时，$b=\mu a$，反向时 $b=-\mu a$。

最后，向量的加，减，数乘统称为向量的线性运算。

平面向量的基本定理及坐标表示

平面向量基本定理

定理内容：如果两个向量 e_1,e_2 不共线，那么存在唯一实数对 (x,y)，使得与 e_1,e_2 共面的任意向量 p 满足 $\mathbf{p}=xe_1+ye_2$。

平面向量那么多，怎样用尽可能少的量表示出所有平面向量?

只用一个向量表示出所有向量显然是不可能的，最多只能表示出某条直线上的向量。

再加入一个向量，用两个**不共线**向量表示（两个共线向量在此可以看成同一个向量），这样可以把任意一个平面向量分解到这两个向量的方向上了。

在同一平面内的两个不共线的向量称为**基底**。如果基底相互垂直，那么在分解的时候就是对向量**正交分解**。

平面向量的坐标表示

如果取与横轴与纵轴方向相同的单位向量 i, j 作为一组基底，根据平面向量基本定理，平面上的所有向量与有序实数对 (x, y) 一一对应。

而有序实数对 (x, y) 与平面直角坐标系上的点一一对应，于是作 $\overrightarrow{OP} = p$，那么终点 $P(x, y)$ 也是唯一确定的。由于研究的对象是自由向量，可以自由平移起点，这样，在平面直角坐标系里，每一个向量都可以用有序实数对唯一表示。

平面向量的坐标运算

平面向量线性运算

由平面向量的线性运算可以推导其坐标运算，主要方法是将坐标全部化为用基底表示，然后利用运算律进行合并，之后表示出运算结果的坐标形式。

若两向量 $a = (m, n)$，$b = (p, q)$，则：

$$a + b = (m + p, n + q)$$
$$a - b = (m - p, n - q)$$
$$ka = (km, kn)$$

求一个向量的坐标表示

已知两点 $A(a, b), B(c, d)$，易证 $\overrightarrow{AB} = (c - a, d - b)$。

平移一点

有时需要将一个点 P 沿一定方向平移某单位长度，这样把要平移的方向和距离组合成一个向量，利用向量加法的三角形法则，将 $\overrightarrow{OP}$ 加上这个向量，得到的向量终点即为平移后的点。

三点共线的判定

若 A, B, C 三点共线，则 $\overrightarrow{OB} = \lambda\overrightarrow{OA} + (1 - \lambda)\overrightarrow{OC}$。

三点共线判定的拓展

在三角形 ABC 中，若 D 为 BC 的 n 等分点（$n\ BD = k\ DC$），则有：$\overrightarrow{AD} = \frac{n}{k+n}\overrightarrow{AB} + \frac{k}{k+n}\overrightarrow{AC}$

在三维空间中的拓展（立体几何/空间向量）

在空间中，以上部分所述的所有内容均成立。更有：

空间向量基本定理

定理内容：如果两个向量 e_1, e_2, e_3 不共面，那么存在唯一实数对 (x, y, z)，使得空间中任意向量 p 满足 $\mathbf{p} = xe_1 + ye_2 + ze_3$。根据空间向量基本定理，我们同样可以使用三个相互垂直的基底 e_1, e_2, e_3 作为正交基底，建立**空间直角坐标系**并用一个三元组 (x, y, z) 作为坐标表示空间向量。

共面向量基本定理

如果存在两个不共线的向量 x, y, 则向量 p 与 x, y 共面的充要条件是存在唯一实数对 (a, b) 使得 $p = ax + by$。

法向量

对于一个面 $ABCD$，其法向量 n 与这个面垂直。

计算方法：任取两个面内直线 $\overrightarrow{AB},\overrightarrow{AD}$，使得 $\overrightarrow{AB}\cdot n=0$ 且 $\overrightarrow{AD}\cdot n=0$，利用坐标法即可计算。

扩展

向量与矩阵

矩阵运算的相关法则与向量运算相似，于是考虑将向量写成矩阵形式，这样就将向量问题化为矩阵问题了。详细内容请参考线性代数。

向量旋转

设 $a=(x,y)$，倾角为 θ，长度为 $l=\sqrt{x^2+y^2}$。则 $x=l\cos\theta, y=l\sin\theta$。令其逆时针旋转 α 度角，得到向量 $b=(l\cos(\theta+\alpha), l\sin(\theta+\alpha))$。

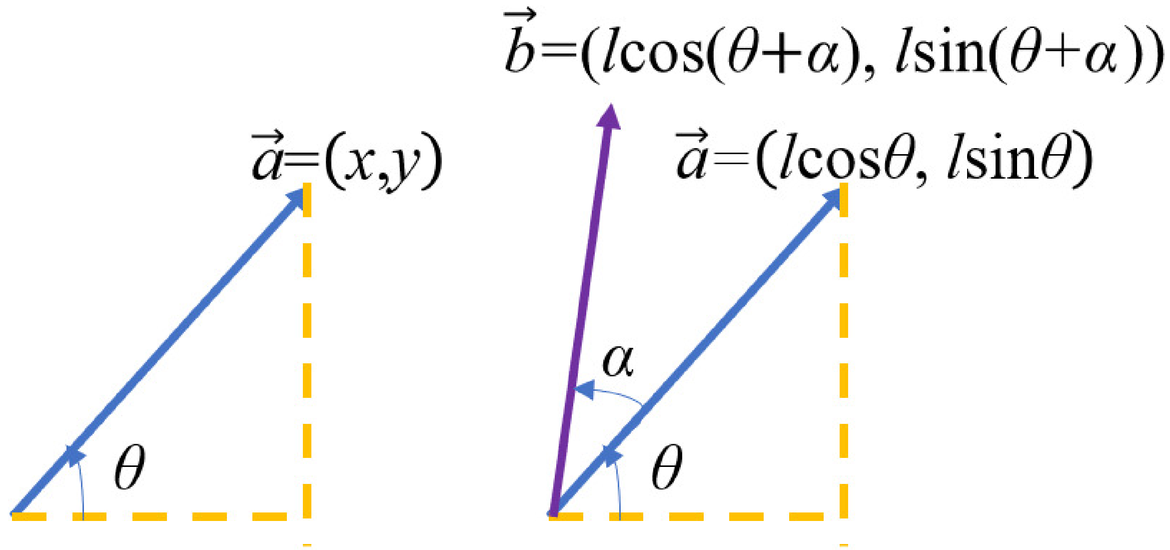


图 9.26

由三角恒等变换得，

$$b=(l(\cos\theta\cos\alpha-\sin\theta\sin\alpha), l(\sin\theta\cos\alpha+\cos\theta\sin\alpha))$$

化简，

$$b=(l\cos\theta\cos\alpha-l\sin\theta\sin\alpha, l\sin\theta\cos\alpha+l\cos\theta\sin\alpha)$$

把上面的 x,y 代回来得

$$b=(x\cos\alpha-y\sin\alpha, y\cos\alpha+x\sin\alpha)$$

即使不知道三角恒等变换，这个式子也很容易记下来。

向量的更严格定义

上文中，向量被定义为了空间中的有向线段。但是严格来说，向量不仅是有向线段。要作出向量的更严格定义，需要先定义 线性空间，具体内容参见 线性空间 页面的介绍。

9.15.3 内积和外积

本文介绍向量之间的简单运算。

在本文之前，特别说明一下翻译的相关问题。由于历史原因，数学学科和物理学科关于「inner product」和「outer product」两个词汇有着五花八门的翻译。

在物理学科，一般翻译成「标积」和「矢积」，表示运算的结果为标量和矢量。高中数学课本上「数量积」和「向量积」也采用了这种意译的办法。

在数学学科，通常也可以翻译成「内积」和「外积」，是两个名词的直译。「点乘」和「叉乘」是根据运算符号得来的俗称，这种俗称也很常见。

在「点乘」运算中，经常省略运算的点符号，在线性代数中更是会直接看作矩阵乘法，不写点符号。

内积

内积的概念**对于任意维数的向量都适用**。

已知两个向量 a, b，它们的夹角为 θ，那么：

$$a \cdot b = |a||b| \cos\theta$$

就是这两个向量的**内积**，也叫**点积**或**数量积**。其中称 $|a|\cos\theta$ 为 a 在 b 方向上的投影。内积的几何意义即为：内积 $a \cdot b$ 等于 a 的模与 b 在 a 方向上的投影的乘积。

可以发现，这种运算得到的结果是一个标量，并不属于向量的线性运算。

在不引起混淆的情况下，内积的点号可以省略不写。如果在向量的右上角有上角标 2，表示向量与自身内积的简写，即**向量模长的平方**，省略模长记号。该上角标 2 不可以理解为向量的平方，这是因为，向量内积的结果为标量，不存在除了 2 以外任何个数的向量的内积。同理，向量模长平方的平方，不可以简写为上角标 4，而是必须将上角标 2 的结果视为一个整体，以此类推。

内积满足交换律，即：

$$a \cdot b = b \cdot a$$

互相垂直的两个向量的内积，结果为 0。向量与零向量内积，结果为 0。

内积运算有以下应用：

判定两向量垂直

$a \perp b \iff a \cdot b = 0$

判定两向量共线

$a = \lambda b \iff |a \cdot b| = |a||b|$

数量积的坐标运算

若 $a = (m, n), b = (p, q)$, 则 $a \cdot b = mp + nq$

向量的模

$|a| = \sqrt{m^2 + n^2}$

两向量的夹角

$\cos\theta = \dfrac{a \cdot b}{|a||b|}$

二阶与三阶行列式

二阶与三阶行列式，可以作为行列式的较为简单的情形特殊定义。在微积分的最后一个部分场论部分，格林公式用到了二阶行列式，高斯公式用到了点乘，斯托克斯公式用到了三阶行列式。

二阶行列式可以视为四元函数，其定义为：

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

三阶行列式可以视为九元函数，其定义为：

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = aei + dhc + gbf - ahf - dbi - gec$$

一种特殊的记忆方法是采用「对角线法则」，对角线法则只适用于二阶与三阶行列式。

特别注意：四阶行列式展开后共有 24 项，并且副对角线一项的符号为正。如果强行应用三阶行列式的「对角线法则」，不仅项数不够，副对角线一项的符号也不正确，因此三阶行列式的「对角线法则」不适用于更高阶的行列式，更高阶的行列式也不适合使用直接展开法计算。

外积

外积是**三维向量特有的运算**。

在物理学中，三维向量为默认与空间位置相关的向量，一律采用粗体表示。然而，物理学中与相对论相关的四维向量不会采用粗体，而是使用特殊的记号与下标。

在线性代数中，所有的向量都会用粗体表示，并且由于麻烦，并且线性代数中大多为向量与矩阵的运算，很难造成歧义，在手写时可以省略向量记号不写。

定义向量 a, b 的外积为一个向量，记为 $a \times b$，其模与方向定义如下：

1. $|a \times b| = |a||b| \sin\langle a, b\rangle$；
2. $a \times b$ 与 a, b 都垂直，且 $a, b, a \times b$ 符合右手法则。

注意到外积的模，联想到三角形面积计算公式 $S = \frac{1}{2}ab\sin C$，可以发现外积的几何意义是：$|a \times b|$ **是以** a, b **为邻边的平行四边形的面积**。

两个向量 $a = (x_1, y_1, z_1)$，$b = (x_2, y_2, z_2)$ 外积的结果是一个向量 c。记作 $c = a \times b$。

向量的外积可以使用三阶行列式表示：

$$\begin{vmatrix} i & j & k \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{vmatrix}$$

其中 i, j, k 表示和坐标轴 x, y, z 平行的单位向量，并写在对应坐标处。展开得 $c = (y_1z_2 - y_2z_1, x_2z_1 - x_1z_2, x_1y_2 - x_2y_1)$。

对于二维向量，无法计算外积，但是仍然可以计算两向量张成的平行四边形面积：

记 $a = (m, n), b = (p, q)$，将平面直角坐标系扩充为空间直角坐标系，原平面位于新坐标系的 xOy 平面，原本的坐标 (m, n) 和 (p, q) 变为 $(m, n, 0)$ 和 $(p, q, 0)$，那么两个向量的外积为 $(0, 0, mq - np)$，因此平行四边形面积为 $mq - np$，可以视为二阶行列式运算的结果。此时，根据右手法则和竖坐标符号，可以推断出 b 相对于 a 的方向，若在逆时针方向竖坐标为正值，反之为负值，简记为**顺负逆正**。

外积满足**反交换律**，即：

$$a \times b = -b \times a$$

共线的两个三维向量的外积，结果为 0。三维向量与自身外积，结果为 0。三维向量与零向量外积，结果为 0。

根据上文的两个定义：

$$|a \times b| = |a||b| \sin\langle a, b\rangle$$

$$a \cdot b = |a||b| \cos\theta$$

可以写出恒等式：

$$(a \times b)^2 = a^2b^2 - (a \cdot b)^2$$

混合积

与外积一样，向量的混合积是**三维向量特有的运算**。

设 a, b, c 是空间中三个向量，则 $(a \times b)c$ 称为三个向量 a, b, c 的混合积，记作 $[abc]$ 或 (a, b, c) 或 (abc)。混合积的绝对值 $|(a \times b)c|$ 的几何意义表示以 a, b, c 为棱的平行六面体的体积。

向量的混合积可以使用三阶行列式表示：

$$(a, b, c) = (a \times b)c = \begin{vmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ a_z & b_z & c_z \end{vmatrix} = a_x b_y c_z + a_y b_z c_x + a_z b_x c_y - a_z b_y c_x - a_y b_x c_z - a_x b_z c_y$$

向量的混合积可以用来计算四面体的体积：

$$V = \frac{1}{6}|[ABACAD]|$$

混合积 (a, b, c) 的符号是正还是负，取决于 $a \times b$ 与 c 形成的夹角是锐角还是钝角，即指向 a 与 b 张成平面的同侧还是异侧，这相当于 a、b、c 三个向量依序构成右手系还是左手系。

有定理：三个三维向量 a、b、c 共面的充分必要条件是 $(a, b, c) = 0$。

混合积有性质：

$$(a, b, c) = (b, c, a) = (c, a, b) = -(b, a, c) = -(a, c, b) = -(c, b, a)$$

$$(a \times b)c = a(b \times c)$$

二重外积

三维向量的混合积是内积与外积的混搭，具有轮换对称性。三维向量和三维向量的外积还是三维向量，那么外积的外积是否存在相关结论？

先证明一个引理。

$$(a \times b) \times a = (a^2)b - (ab)a$$

证明：由右手定则，$a \times b$ 与 a 和 b 都垂直，待证等式左端与 $a \times b$ 垂直，因此待证等式左端与 a 和 b 共面。

因此可以假设：

$$(a \times b) \times a = \lambda a + \mu b$$

根据混合积的相关结论，上式两端同时对于 a 和 b 分别做内积，有：

$$\lambda a^2 + \mu(ab) = 0$$

$$\lambda(ab) + \mu b^2 = (b, a \times b, a) = (a \times b)^2$$

由前文推出的恒等式：

$$(a \times b)^2 = a^2 b^2 - (ab)^2$$

可以解得：

$$\lambda = -ab$$

$$\mu = a^2$$

证毕。

在上文的证明中提到，$a \times b$ 与任意向量叉乘，得到的向量与 a 和 b 共面。接下来证明**二重外积**的结论：

$$(a \times b) \times c = (ac)b - (bc)a$$

上述共面性有助于二重外积结论的记忆。可见，上文的引理为二重外积的特殊情况。

证明：这里只需考虑三个向量均为非零且不共线的情况，其他特例为显然的。

三维向量 a、b 和 $a \times b$ 不共面，因此可以假设：

$$c = \alpha a + \beta b + \gamma(a \times b)$$

所以有：

$$(a \times b) \times c = (a \times b) \times (\alpha a + \beta b + \gamma(a \times b)) = \alpha(a \times b) \times a + \beta(a \times b) \times b$$

根据上文的引理有：

$$(a \times b) \times a = (a^2)b - (ab)a$$

$$(a \times b) \times b = -(b \times a) \times b = -(b^2)a + (ab)b$$

因此有：

$$(a \times b) \times c = \alpha((a^2)b - (ab)a) + \beta((ab)b - (b^2)a) = (\alpha(-ab) + \beta(-b^2))a + (\alpha a^2 + \beta ab)b = (ac)b - (bc)a$$

证毕。

根据外积的反交换性，可以得到二重外积的两个公式：

$$(a \times b) \times c = (ac)b - (bc)a$$

$$a \times (b \times c) = (ac)b - (ab)c$$

可见，二重外积对于运算顺序有着严格的要求。

借助混合积与二重外积，还可以证明拉格朗日的恒等式。

$$(a \times b)(c \times d) = (ac)(bd) - (ad)(bc)$$

证明：

$$(a \times b)(c \times d) = (c, d, a \times b) = (a \times b, c, d) = ((a \times b) \times c)d = (b(ac) - a(bc))d = (ac)(bd) - (ad)(bc)$$

可见，前文的恒等式

$$(a \times b)^2 = a^2b^2 - (ab)^2$$

是拉格朗日的恒等式的特殊情形。

9.15.4 矩阵

本文介绍线性代数中一个非常重要的内容——矩阵（Matrix），主要讲解矩阵的性质、运算，以及矩阵乘法的一些应用。

向量与矩阵

在线性代数中，向量分为列向量和行向量。

warning

在中国台湾地区关于「列」与「行」的翻译，恰好与中国大陆地区相反。在 **OI Wiki** 按照中国大陆地区的习惯，采用列（column）与行（row）的翻译。

线性代数的主要研究对象是列向量，约定使用粗体小写字母表示列向量。在用到大量向量与矩阵的线性代数中，不引起混淆的情况下，在手写时，字母上方的向量记号可以省略不写。

向量也是特殊的矩阵。如果想要表示行向量，需要在粗体小写字母右上方写转置记号。行向量在线性代数中一般表示方程。

引入

矩阵的引入来自于线性方程组。与向量类似，矩阵体现了一种对数据「打包处理」的思想。

例如，将线性方程组：

$$\begin{cases} 7x_1 + 8x_2 + 9x_3 = 13 \\ 4x_1 + 5x_2 + 6x_3 = 12 \\ x_1 + 2x_2 + 3x_3 = 11 \end{cases} \tag{9.43}$$

一般用圆括号或方括号表示矩阵。将上述系数抽出来，写成矩阵乘法的形式：

$$\begin{pmatrix} 7 & 8 & 9 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 13 \\ 12 \\ 11 \end{pmatrix} \tag{9.44}$$

简记为：

$$Ax = b$$

即未知数列向量 x，左乘一个矩阵 A，得到列向量 b。这个式子可以认为是线性代数的基本形式。

线性代数主要研究的运算模型是内积。内积是先相乘再相加，是行向量左乘列向量，得到一个数的过程。

矩阵乘法是内积的拓展。矩阵乘法等价于左边矩阵抽出一行，与右边矩阵抽出一列进行内积，得到结果矩阵的对应元素，口诀「左行右列」。

当研究对象是右边的列向量时，矩阵乘法相当于对列向量进行左乘。在左乘的观点下，矩阵就是对列向量的变换，将矩阵乘法中右边矩阵的每一个列向量进行变换，对应地得到结果矩阵中每一个列向量。

矩阵可以对一个列向量进行变换，也可以对一组列向量进行「打包」变换，甚至可以对整个空间——即全体列向量进行变换。当矩阵被视为对整个空间变换的时候，也就脱离了空间，成为了纯粹变换的存在。

定义

对于矩阵 A，主对角线是指 $A_{i,i}$ 的元素。

一般用 I 来表示单位矩阵，就是主对角线上为 1，其余位置为 0。

同型矩阵

两个矩阵，行数与列数对应相同，称为同型矩阵。

方阵

行数等于列数的矩阵称为方阵。方阵是一种特殊的矩阵。对于「n 阶矩阵」的习惯表述，实际上讲的是 n 阶方阵。阶数相同的方阵为同型矩阵。

研究方程组、向量组、矩阵的秩的时候，使用一般的矩阵。研究特征值和特征向量、二次型的时候，使用方阵。

主对角线

方阵中行数等于列数的元素构成主对角线。

对称矩阵

如果方阵的元素关于主对角线对称，即对于任意的 i 和 j，i 行 j 列的元素与 j 行 i 列的元素相等，则将方阵称为对称矩阵。

对角矩阵

主对角线之外的元素均为 0 的方阵称为对角矩阵，一般记作：

$$\operatorname{diag}\{\lambda_1, \cdots, \lambda_n\}$$

式中的 $\lambda_1, \cdots, \lambda_n$ 是主对角线上的元素。

对角矩阵是对称矩阵。

如果对角矩阵的元素均为 1，称为单位矩阵，记为 I。只要乘法可以进行，无论形状，任何矩阵乘单位矩阵仍然保持不变。

三角矩阵

如果方阵主对角线左下方的元素均为 0，称为上三角矩阵。如果方阵主对角线右上方的元素均为 0，称为下三角矩阵。

两个上（下）三角矩阵的乘积仍然是上（下）三角矩阵。如果对角线元素均非 0，则上（下）三角矩阵可逆，逆也是上（下）三角矩阵。

单位三角矩阵

如果上三角矩阵 A 的对角线全为 1，则称 A 是单位上三角矩阵。如果下三角矩阵 A 的对角线全为 1，则称 A 是单位下三角矩阵。

两个单位上（下）三角矩阵的乘积仍然是单位上（下）三角矩阵，单位上（下）三角矩阵的逆也是单位上（下）三角矩阵。

运算

矩阵的线性运算

矩阵的线性运算分为加减法与数乘，它们均为逐个元素进行。只有同型矩阵之间可以对应相加减。

矩阵的转置

矩阵的转置，就是在矩阵的右上角写上转置「T」记号，表示将矩阵的行与列互换。

对称矩阵转置前后保持不变。

矩阵乘法

矩阵的乘法是向量内积的推广。

矩阵相乘只有在第一个矩阵的列数和第二个矩阵的行数相同时才有意义。

设 A 为 $P \times M$ 的矩阵，B 为 $M \times Q$ 的矩阵，设矩阵 C 为矩阵 A 与 B 的乘积，

其中矩阵 C 中的第 i 行第 j 列元素可以表示为：

$$C_{i,j} = \sum_{k=1}^{M} A_{i,k} B_{k,j}$$

在矩阵乘法中，结果 C 矩阵的第 i 行第 j 列的数，就是由矩阵 A 第 i 行 M 个数与矩阵 B 第 j 列 M 个数分别**相乘再相加**得到的。这里的**相乘再相加**，就是向量的内积。乘积矩阵中第 i 行第 j 列的数恰好是乘数矩阵 A 第 i 个行向量与乘数矩阵 B 第 j 个列向量的内积，口诀为**左行右列**。

线性代数研究的向量多为列向量，根据这样的对矩阵乘法的定义方法，经常研究对列向量左乘一个矩阵的左乘运算，同时也可以在这里看出「打包处理」的思想，同时处理很多个向量内积。

矩阵乘法满足结合律，不满足一般的交换律。

利用结合律，矩阵乘法可以利用 快速幂 的思想来优化。

在比赛中，由于线性递推式可以表示成矩阵乘法的形式，也通常用矩阵快速幂来求线性递推数列的某一项。

优化

首先对于比较小的矩阵，可以考虑直接手动展开循环以减小常数。

可以重新排列循环以提高空间局部性，这样的优化不会改变矩阵乘法的时间复杂度，但是会在得到常数级别的提升。

```cpp
// 以下文的参考代码为例
mat operator*(const mat& T) const {
  mat res;
  for (int i = 0; i < sz; ++i)
    for (int j = 0; j < sz; ++j)
      for (int k = 0; k < sz; ++k) {
        res.a[i][j] += mul(a[i][k], T.a[k][j]);
        res.a[i][j] %= MOD;
      }
  return res;
}

// 不如
mat operator*(const mat& T) const {
  mat res;
  int r;
  for (int i = 0; i < sz; ++i)
    for (int k = 0; k < sz; ++k) {
      r = a[i][k];
      for (int j = 0; j < sz; ++j)
        res.a[i][j] += T.a[k][j] * r, res.a[i][j] %= MOD;
    }
  return res;
}
```

方阵的逆

方阵 A 的逆矩阵 P 是使得 $A \times P = I$ 的矩阵。

逆矩阵不一定存在。如果存在，可以使用 高斯消元 进行求解。

方阵的行列式

行列式是方阵的一种运算。

参考代码

一般来说，可以用一个二维数组来模拟矩阵。

```cpp
struct mat {
  LL a[sz][sz];

  mat() { memset(a, 0, sizeof a); }

  mat operator-(const mat& T) const {
    mat res;
    for (int i = 0; i < sz; ++i)
      for (int j = 0; j < sz; ++j) {
        res.a[i][j] = (a[i][j] - T.a[i][j]) % MOD;
      }
    return res;
  }

  mat operator+(const mat& T) const {
    mat res;
```

```
    for (int i = 0; i < sz; ++i)
      for (int j = 0; j < sz; ++j) {
        res.a[i][j] = (a[i][j] + T.a[i][j]) % MOD;
      }
    return res;
  }

  mat operator*(const mat& T) const {
    mat res;
    int r;
    for (int i = 0; i < sz; ++i)
      for (int k = 0; k < sz; ++k) {
        r = a[i][k];
        for (int j = 0; j < sz; ++j)
          res.a[i][j] += T.a[k][j] * r, res.a[i][j] %= MOD;
      }
    return res;
  }

  mat operator^(LL x) const {
    mat res, bas;
    for (int i = 0; i < sz; ++i) res.a[i][i] = 1;
    for (int i = 0; i < sz; ++i)
      for (int j = 0; j < sz; ++j) bas.a[i][j] = a[i][j] % MOD;
    while (x) {
      if (x & 1) res = res * bas;
      bas = bas * bas;
      x >>= 1;
    }
    return res;
  }
};
```

看待线性方程组的两种视角

看待矩阵 A，或者变换 A，有两种视角。

第一种观点：按行看，观察 A 的每一行。这样一来把 A 看作方程组。于是就有了消元法解方程的过程。

第二种观点：按列看，观察 A 的每一列。A 本身也是由列向量构成的。此时相当于把变换 A 本身看成了列向量组，而 x 是未知数系数，思考 A 当中的这组列向量能不能配上未知数，凑出列向量 b。

例如，文章开头的例子变为：

$$\begin{pmatrix}7\\4\\1\end{pmatrix}x_1+\begin{pmatrix}8\\5\\2\end{pmatrix}x_2+\begin{pmatrix}9\\6\\3\end{pmatrix}x_3=\begin{pmatrix}13\\12\\11\end{pmatrix} \tag{9.45}$$

解方程变为研究，是否可以通过调整三个系数 x，使得给定的三个基向量能够凑出结果的向量。

按列看比按行看更新颖。在按列看的视角下，可以研究线性无关与线性相关。

矩阵乘法的应用

矩阵加速递推

以 斐波那契数列（Fibonacci Sequence） 为例。在斐波那契数列当中，$F_1=F_2=1$，$F_i=F_{i-1}+F_{i-2}(i\geq 3)$。

如果有一道题目让你求斐波那契数列第 n 项的值，最简单的方法莫过于直接递推了。但是如果 n 的范围达到了 10^{18} 级别，递推就不行了，此时我们可以考虑矩阵加速递推。

根据斐波那契数列 递推公式的矩阵形式:

$$\begin{bmatrix}F_{n-1} & F_{n-2}\end{bmatrix}\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}=\begin{bmatrix}F_n & F_{n-1}\end{bmatrix}$$

定义初始矩阵 $\text{ans}=\begin{bmatrix}F_2 & F_1\end{bmatrix}=\begin{bmatrix}1 & 1\end{bmatrix}, \text{base}=\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}$。那么，$F_n$ 就等于 ansbase^{n-2} 这个矩阵的第一行第一列元素，也就是 $\begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^{n-2}$ 的第一行第一列元素。

" 注意"

矩阵乘法不满足交换律，所以一定不能写成 $\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^{n-2}\begin{bmatrix}1 & 1\end{bmatrix}$ 的第一行第一列元素。另外，对于 $n\le 2$ 的情况，直接输出 1 即可，不需要执行矩阵快速幂。

为什么要乘上 base 矩阵的 $n-2$ 次方而不是 n 次方呢？因为 F_1, F_2 是不需要进行矩阵乘法就能求的。也就是说，如果只进行一次乘法，就已经求出 F_3 了。如果还不是很理解为什么幂是 $n-2$，建议手算一下。

下面是求斐波那契数列第 n 项对 10^9+7 取模的示例代码（核心部分）。

```cpp
const int mod = 1000000007;

struct Matrix {
  int a[3][3];

  Matrix() { memset(a, 0, sizeof a); }

  Matrix operator*(const Matrix &b) const {
    Matrix res;
    for (int i = 1; i <= 2; ++i)
      for (int j = 1; j <= 2; ++j)
        for (int k = 1; k <= 2; ++k)
          res.a[i][j] = (res.a[i][j] + a[i][k] * b.a[k][j]) % mod;
    return res;
  }
} ans, base;

void init() {
  base.a[1][1] = base.a[1][2] = base.a[2][1] = 1;
  ans.a[1][1] = ans.a[1][2] = 1;
}

void qpow(int b) {
  while (b) {
    if (b & 1) ans = ans * base;
    base = base * base;
    b >>= 1;
  }
}

int main() {
  int n = read();
  if (n <= 2) return puts("1"), 0;
  init();
```

```
  qpow(n - 2);
  println(ans.a[1][1] % mod);
}
```

这是一个稍微复杂一些的例子。

$$f_1 = f_2 = 0$$

$$f_n = 7f_{n-1} + 6f_{n-2} + 5n + 4 \times 3^n$$

我们发现，f_n 和 f_{n-1}, f_{n-2}, n 有关，于是考虑构造一个矩阵描述状态。

但是发现如果矩阵仅有这三个元素 $\begin{bmatrix} f_n & f_{n-1} & n \end{bmatrix}$ 是难以构造出转移方程的，因为乘方运算和 $+1$ 无法用矩阵描述。

于是考虑构造一个更大的矩阵。

$$\begin{bmatrix} f_n & f_{n-1} & n & 3^n & 1 \end{bmatrix}$$

我们希望构造一个递推矩阵可以转移到

$$\begin{bmatrix} f_{n+1} & f_n & n+1 & 3^{n+1} & 1 \end{bmatrix}$$

转移矩阵即为

$$\begin{bmatrix} 7 & 1 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 \\ 5 & 0 & 1 & 0 & 0 \\ 12 & 0 & 0 & 3 & 0 \\ 5 & 0 & 1 & 0 & 1 \end{bmatrix}$$

矩阵表达修改

"「THUSCH 2017」大魔法师"

大魔法师小 L 制作了 n 个魔力水晶球，每个水晶球有水、火、土三个属性的能量值。小 L 把这 n 个水晶球在地上从前向后排成一行，然后开始今天的魔法表演。

我们用 A_i，B_i，C_i 分别表示从前向后第 i 个水晶球（下标从 1 开始）的水、火、土的能量值。

小 L 计划施展 m 次魔法。每次，他会选择一个区间 $[l, r]$，然后施展以下 3 大类、7 种魔法之一：

1. 魔力激发：令区间里每个水晶球中**特定属性**的能量爆发，从而使另一个**特定属性**的能量增强。具体来说，有以下三种可能的表现形式：
   - 火元素激发水元素能量：令 $A_i = A_i + B_i$。
   - 土元素激发火元素能量：令 $B_i = B_i + C_i$。
   - 水元素激发土元素能量：令 $C_i = C_i + A_i$。

   需要注意的是，增强一种属性的能量并不会改变另一种属性的能量，例如 $A_i = A_i + B_i$ 并不会使 B_i 增加或减少。
2. 魔力增强：小 L 挥舞法杖，消耗自身 v 点法力值，来改变区间里每个水晶球的**特定属性**的能量。具体来说，有以下三种可能的表现形式：
   - 火元素能量定值增强：令 $A_i = A_i + v$。
   - 水元素能量翻倍增强：令 $B_i = B_i \cdot v$。
   - 土元素能量吸收融合：令 $C_i = v$。
3. 魔力释放：小 L 将区间里所有水晶球的能量聚集在一起，融合成一个新的水晶球，然后送给场外观众。生成的水晶球每种属性的能量值等于区间内所有水晶球对应能量值的代数和。**需要注意的是，魔力释放的过程不会真正改变区间内水晶球的能量。**

值得一提的是，小 L 制造和融合的水晶球的原材料都是定制版的 OI 工厂水晶，所以这些水晶球有一个能量阈值 998244353。当水晶球中某种属性的能量值大于等于这个阈值时，能量值会自动对阈值取模，从而避免水晶球爆炸。

小 W 为小 L（唯一的）观众，围观了整个表演，并且收到了小 L 在表演中融合的每个水晶球。小 W 想知道，这些水晶球蕴涵的三种属性的能量值分别是多少。

由于矩阵的结合律和分配律成立，单点修改可以自然地推广到区间，即推出矩阵后直接用线段树维护区间矩阵乘积即可。

下面将举几个例子。

$A_i = A_i + v$ 的转移

$$\begin{bmatrix} A & B & C & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ v & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} A+v & B & C & 1 \end{bmatrix}$$

$B_i = B_i \cdot v$ 的转移

$$\begin{bmatrix} A & B & C & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & v & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} A & B \cdot v & C & 1 \end{bmatrix}$$

"「LibreOJ 6208」树上询问"

有一棵 n 节点的树，根为 1 号节点。每个节点有两个权值 k_i, t_i，初始值均为 0。

给出三种操作：

1. $\text{Add}(x, d)$ 操作：将 x 到根的路径上所有点的 $k_i \leftarrow k_i + d$
2. $\text{Mul}(x, d)$ 操作：将 x 到根的路径上所有点的 $t_i \leftarrow t_i + d \times k_i$
3. $\text{Query}(x)$ 操作：询问点 x 的权值 t_x

   $n,\ m \leq 100000,\ \ -10 \leq d \leq 10$

若直接思考，下放操作和维护信息并不是很好想。但是矩阵可以轻松地表达。

$$\begin{bmatrix} k & t & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ d & 0 & 1 \end{bmatrix} = \begin{bmatrix} k+d & t & 1 \end{bmatrix}$$

$$\begin{bmatrix} k & t & 1 \end{bmatrix} \begin{bmatrix} 1 & d & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} k & t + d \times k & 1 \end{bmatrix}$$

定长路径统计

" 问题描述"

给一个 n 阶有向图，每条边的边权均为 1，然后给一个整数 k，你的任务是对于所有点对 (u, v) 求出从 u 到 v 长度为 k 的路径的数量（不一定是简单路径，即路径上的点或者边可能走多次）。

我们将这个图用邻接矩阵 G（对于图中的边 $(u \to v)$，令 $G[u, v] = 1$，其余为 0 的矩阵；如果有重边，则设 $G[u, v]$ 为重边的数量）表示这个有向图。下述算法同样适用于图有自环的情况。

显然，该邻接矩阵对应 $k = 1$ 时的答案。

假设我们知道长度为 k 的路径条数构成的矩阵，记为矩阵 C_k，我们想求 C_{k+1}。显然有 DP 转移方程

$$C_{k+1}[i,j]=\sum_{p=1}^{n} C_k[i,p]\cdot G[p,j]$$

我们可以把它看作矩阵乘法的运算，于是上述转移可以描述为

$$C_{k+1}=C_k\cdot G$$

那么把这个递推式展开可以得到

$$C_k=\underbrace{G\cdot G\cdots G}_{k\text{ 次}}=G^k$$

要计算这个矩阵幂，我们可以使用快速幂（二进制取幂）的思想，在 $O(n^3\log k)$ 的复杂度内计算结果。

定长最短路

" 问题描述"

给你一个 n 阶加权有向图和一个整数 k。对于每个点对 (u,v) 找到从 u 到 v 的恰好包含 k 条边的最短路的长度。(不一定是简单路径，即路径上的点或者边可能走多次)

我们仍构造这个图的邻接矩阵 G，$G[i,j]$ 表示从 i 到 j 的边权。如果 i,j 两点之间没有边，那么 $G[i,j]=\infty$。(有重边的情况取边权的最小值)

显然上述矩阵对应 $k=1$ 时问题的答案。我们仍假设我们知道 k 的答案，记为矩阵 L_k。现在我们想求 $k+1$ 的答案。显然有转移方程

$$L_{k+1}[i,j]=\min_{1\le p\le n}\{L_k[i,p]+G[p,j]\}$$

事实上我们可以类比矩阵乘法，你发现上述转移只是把矩阵乘法的乘积求和变成相加取最小值，于是我们定义这个运算为 $\odot$，即

$$A\odot B=C\iff C[i,j]=\min_{1\le p\le n}\{A[i,p]+B[p,j]\}$$

于是得到

$$L_{k+1}=L_k\odot G$$

展开递推式得到

$$L_k=\underbrace{G\odot\ldots\odot G}_{k\text{ 次}}=G^{\odot k}$$

我们仍然可以用矩阵快速幂的方法计算上式，因为它显然是具有结合律的。时间复杂度 $O(n^3\log k)$。

限长路径计数/最短路

上述算法只适用于边数固定的情况。然而我们可以改进算法以解决边数小于等于 k 的情况。具体地，考虑以下问题：

" 问题描述"

给一个 n 阶有向图，边权为 1，然后给一个整数 k，你的任务是对于每个点对 (u,v) 找到从 u 到 v 长度小于等于 k 的路径的数量（不一定是简单路径，即路径上的点或者边可能走多次)。

我们简单修改一下这个图，我们给每一个结点加一个权值为 1 的自环。这样走的时候就可以走自环，相当于原地走。这样就包含了小于等于 k 的情况。修改后再做矩阵快速幂即可。(即使这个图在修改之前就有自环，该算法仍是成立的)。

同样的方法可以用于求边数小于等于 k 的最短路，即加一个边权为 0 的自环。

习题

- 洛谷 P1962 斐波那契数列[1]，即上面的例题，同题 POJ3070
- 洛谷 P1349 广义斐波那契数列[2]，base 矩阵需要变化一下
- 洛谷 P1939【模板】矩阵加速（数列）[3]，base 矩阵变成了 3×3 的矩阵，推导过程与上面差不多。

参考资料与注释

[1] 洛谷 P1962 斐波那契数列

[2] 洛谷 P1349 广义斐波那契数列

[3] 洛谷 P1939【模板】矩阵加速（数列）

[4] ,

[5] Number of paths of fixed length/Shortest paths of fixed length

9.15.5 初等变换

初等矩阵

以下三类方阵称为初等矩阵。

倍乘矩阵

倍乘矩阵是一种特殊的对角矩阵。

$$D_i(k) = \operatorname{diag}\{1, \cdots, 1, k, 1, \cdots, 1\}$$

表示一个对角阵，主对角线上第 i 个元素为 k，并且规定 k 不能为 0，其余的元素全部为 1。

特别地，当 k 为 1 的时候，$D_i(1)$ 就是单位阵 I。

对换矩阵

对换矩阵是一种特殊的对称矩阵。

$$P_{ij} = \begin{pmatrix} I_{i-1} & & & & \\ & 0 & & 1 & \\ & & I_{j-i-1} & & \\ & 1 & & 0 & \\ & & & & I_{n-j} \end{pmatrix}$$

对换矩阵的元素全是 1 和 0，主对角线上其余元素均为 1，仅有第 i 个元素和第 j 个元素为 0，而在第 i 行第 j 列、第 j 行第 i 列上的两个元素为 1。

对换矩阵要求 i 与 j 不能相等。

倍加矩阵

倍加矩阵是在单位阵 I 的基础上，令第 i 行第 j 列为 k。

$$T_{ij}(k) = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & \cdots & k & & \\ & & & \ddots & \vdots & & \\ & & & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix}$$

倍加矩阵要求 i 与 j 不能相等。如果 k 为 0，则 $T_{ij}(0)$ 退化为单位阵 I。

倍加矩阵是一种上三角矩阵或者下三角矩阵。

初等矩阵的行列式

三种初等矩阵具有行列式：

$$|D_i(k)| = k$$

$$|P_{ij}| = -1$$

$$|T_{ij}(k)| = 1$$

由于方阵乘法的行列式等于行列式的乘法，借助下文初等变换与矩阵乘法的等价性，初等矩阵的这个性质可以用于行列式的计算。

初等变换

不仅限于方阵，对于一般的矩阵 A，可以进行初等行变换和初等列变换，统称为初等变换。

初等行变换与初等列变换一样，都有 3 种：倍乘（multiplication）、对换（switching）、倍加（addition）。这里先介绍初等行变换：

- 第 i 行乘非零数 k：$B \mapsto D_i(k)B$。
- 第 i，j 行互换：$B \mapsto P_{ij}B$。
- 第 j 行乘 k 加到第 i 行：$B \mapsto T_{ij}(k)B$。

将上述操作的行改为列，即得到初等列变换。

在初等变换中，对换可以通过倍乘和倍加实现。显然，倍加不能通过倍乘和对换实现。借助行列式的知识，以及下文的初等变换与矩阵乘法的等价性，也能说明倍乘不能通过倍加和对换实现。

因此，相较对换而言，倍乘和倍加是更为本质的操作。对换操作是为了在消元法中，保证消元的有序，而引入的辅助操作。

初等变换与矩阵乘法

可以发现，三类初等矩阵都是在单位阵 I 上进行一次相应的变换得到的结果。在后文的线性变换中指出，线性变换与矩阵之间有对应关系，与这里的关系类似。

无论矩阵 A 是否方阵，对矩阵 A 进行初等行变换，等价于对矩阵 A 左乘初等矩阵。对矩阵 A 进行初等列变换，等价于对矩阵 A 右乘初等矩阵。

倍乘操作

左乘一个倍乘矩阵 $D_i(k)$，等价于将第 i 行变为 k 倍。右乘一个倍乘矩阵 $D_i(k)$，等价于将第 i 列变为 k 倍。

对角阵乘对角阵还是对角阵，对于对角阵的乘法，将主对角线上对应的元素相乘。由于单位阵是特殊的倍乘阵，而倍乘阵要求 k 不为 0，可以看出，只要对角阵主对角线上的元素均非 0，就可以拆分为倍乘阵的乘积。

对于一般的对角阵，无论元素是否为 0，也有相应的结论。左乘对角阵，等价于将对应的行变为原来的若干倍，倍数恰为对角阵主对角线上的相应元素。右乘对角阵，是对相应的列进行同样操作。

由于倍乘矩阵 $D_i(k)$ 的行列式为 k，对于方阵的行或列进行倍乘操作之后，方阵对应的行列式变为原来的 k 倍。对角阵的行列式为主对角线元素的乘积。

倍乘矩阵的乘法可以交换，对角阵的乘法也可以交换，在乘法只有对角阵时，顺序可以任意排列。

单位阵对应的倍乘操作为保持矩阵 A 不变，在实际应用中不进行这样的操作。

对换操作

左乘一个对换矩阵 P_{ij}，等价于将第 i 行与第 j 行交换。右乘一个对换矩阵 P_{ij}，等价于将第 i 列与第 j 列交换。

与倍乘阵和对角阵的关系类似，这里引入置换矩阵的概念。置换矩阵是一个方阵，每行每列均恰有一个 1，其余位置均为 0。单位阵 I 也是特殊的置换矩阵。

置换阵和对于单位阵 I 的行进行置换操作一致，也和对于单位阵 I 的列进行置换操作一致。单位阵 I 本身对应于恒等变换。

左乘一个置换矩阵等价于对原矩阵的行进行置换，右乘一个置换矩阵等价于对原矩阵的列进行置换，相应置换的方法和对于单位阵 I 的行或列进行置换操作一致。

置换矩阵与置换完全对应，置换矩阵构成的乘法群与置换群同构。由于有定理，在恒等变换视为零个对换的乘积的情形下，任何置换都可以拆为对换的乘积，因此任何置换矩阵也可以拆分为对换矩阵的乘积。

由于对换矩阵的行列式为 -1，对于方阵的行或列进行对换操作之后，方阵对应的行列式变为原来的 -1 倍。

对换阵的乘法不可交换，置换阵的乘法也不可交换。

置换矩阵的行列式为 $(-1)^p$，其中 p 为置换矩阵对应置换的逆序数，即置换拆分为对换乘积的个数。

倍加操作

左乘倍加矩阵 $T_{ij}(k)$ 等价于把第 j 行的 k 倍加到第 i 行上。右乘倍加矩阵 $T_{ij}(k)$ 等价于把第 i 列的 k 倍加到第 j 列上。

如果难以记忆，可以观察倍加阵 $T_{ij}(k)$ 是对单位阵 I 进行了怎样的操作，两者是对应的，左乘是对行的操作，右乘是对列的操作，符合口诀左行右列。

由于倍加矩阵的行列式为 1，对于方阵进行倍加操作之后，方阵对应的行列式不变。

倍加矩阵的乘法不可交换。

单位阵对应的倍加操作为保持矩阵 A 不变，在实际应用中不进行这样的操作。

上三角矩阵

倍加矩阵是一种上三角矩阵或者下三角矩阵。由于两种矩阵关于主对角线对称，这里讨论上三角矩阵。事实上在这个例子中，只需要进行初等行变换，而不需要列变换。

如果一个上三角矩阵的主对角线均为 1，则可拆分为一连串倍加矩阵的乘积。拆分的顺序为，先对单位矩阵 I 的第一行进行倍加操作，再对单位矩阵 I 的第二行进行倍加操作，以此类推，直到每一行均被操作完毕为止。

由于倍加矩阵的乘法不可交换，上述操作不可调换顺序。

如果一个上三角矩阵的主对角线均非 0，则可拆分为一连串倍加矩阵和倍乘矩阵的乘积。可以在操作单位矩阵 I 的每一行时，先将该行进行倍乘操作，效果为主对角线元素变为指定非零值。

如果一个上三角矩阵的主对角线存在 0，则不可拆分为一连串初等矩阵的乘积。

无论上三角矩阵的主对角线上是否有 0，上三角矩阵的行列式等于主对角线元素乘积，与对角阵一致。

倍加操作将方阵转化为对角阵

只使用倍加操作可以使任意一个方阵变为对角阵，这个例子既需要初等行变换也需要初等列变换。

如果方阵的第一行和第一列存在非零元素，则可以通过倍加办法将左上角元素变为非零，进而借助初等行变换和初等列变换，将第一行和第一列除了左上角元素以外，均变为 0。

如果方阵的第一行和第一列已经均为 0，则直接看第二行和第二列即可。

借助这个办法，甚至可以规定对角阵的非零元素均在左上角。

如果方阵的第一行和第一列已经均为 0，则看剩余的行列是否有非零元素，只要有非零元素，则可以通过倍加操作将第一行和第一列中某个元素变为非 0，进而化归为一开始的情况，使得左上角元素非 0。

仅当剩余的行列也均没有非零元素时，左上角无法变为非零元素，此时剩余的方阵已经为零矩阵。

标准形矩阵

借助初等变换可以将任意的矩阵，无论形状，化归为标准形矩阵。

标准形矩阵拥有一个单位阵 I 作为子矩阵位于左上角，其余部分均为 0。化归的办法与将方阵转化为对角阵的操作类似，并需要借助倍乘操作使左上角非零元素变为 1。

矩阵转化为标准形矩阵后，含有元素 1 的个数恰好为矩阵的秩。

可逆矩阵

设 A 是一个 n 阶矩阵。如果存在一个 n 阶矩阵 B，使得 $AB = BA = I$，那么 A 叫做一个可逆矩阵或非奇异矩阵，B 叫做 A 的逆矩阵，并记为 A^{-1}。

如果矩阵 A 可逆，那么 A 的逆矩阵由 A 唯一确定。

可逆矩阵 A 的逆 A^{-1} 也可逆，并且 A^{-1} 的逆就是 A。

两个可逆矩阵 A 和 B 的乘积 AB 也可逆，并且逆为 $B^{-1}A^{-1}$。

可逆矩阵 A 的转置 A^T 也可逆，并且转置的逆等于逆的转置。

初等矩阵的逆

初等矩阵均可逆，并且逆为同类的初等矩阵：

$$D_i(k)^{-1} = D_i\left(\frac{1}{k}\right)$$

$$P_{ij}^{-1} = P_{ij}$$

$$T_{ij}(k)^{-1} = T_{ij}(-k)$$

显然单位阵 I 可逆，逆矩阵仍为 I。

初等变换保持矩阵的可逆性，变换前后矩阵要么同时可逆，要么同时不可逆。

矩阵 A 可逆，当且仅当矩阵 A 可以写成初等矩阵的乘积，即可以通过初等变换变为单位阵 I。

等到引入行列式之后可以知道：

矩阵 A 可逆，当且仅当矩阵 A 的秩为 n，当且仅当矩阵 A 的行列式非 0。

一种简单的记法为：记 E_{ij} 为第 i 行第 j 列的元素为 1、其余为零的 $n \times n$ 矩阵，那么

- $D_i(k) = I_n + (k-1)E_{ii}$
- $P_{ij} = I_n - E_{ii} - E_{jj} + E_{ij} + E_{ji}$
- $T_{ij}(k) = I_n + kE_{ij}$

这种记法也可以应用于它们的逆矩阵。

应用

线性方程组求解

对于一个线性方程组，未知数前的系数构成系数矩阵，如果在系数矩阵右端补上线性方程组的常数项则构成增广矩阵。

应用初等行变换，可以将线性方程组对应的增广矩阵先转化为行阶梯形矩阵，再转化为行最简形矩阵，进而完成线性方程组的求解。这个方法叫做消元法解线性方程组，后文的 Gauss–Jordan 消元，是按照一定的顺序进行的消元算法。

行列式计算

由于方阵乘积的行列式等于方阵行列式的乘积，初等矩阵的行列式便于计算，以及初等变换等价于初等矩阵的乘法，在行列式计算中也会使用初等变换。

由于按照一定的顺序进行初等变换更加便于程序书写，行列式计算也可以使用后文的 Gauss–Jordan 消元算法。

9.15.6 行列式

行列式，是方阵的一种运算。对于方阵 A，$\det A$ 表示方阵 A 的行列式。

本文介绍行列式的三种定义。可以证明，本文中的定义方法是等价的。

前置知识：置换、逆序数、初等变换。

全排列方法定义

手动计算较低阶的行列式可以采用这种方法，它的时间复杂度为阶乘量级。

使用记号 $\pi(j_1j_2\cdots j_n)$ 表示排列 $j_1j_2\cdots j_n$ 的逆序数。记号：

$$\det A=\begin{vmatrix}a_{11} & a_{12} & \cdots & a_{1n}\\ a_{21} & a_{22} & \cdots & a_{2n}\\ \vdots & \vdots & & \vdots\\ a_{n1} & a_{n2} & \cdots & a_{nn}\end{vmatrix}$$

表示的 n 阶行列式是指 $n!$ 项的代数和，这些项是一切可能的取自方阵 A 中不同的行与不同的列上的 n 个元素的乘积 $a_{1j_1}a_{2j_2}\cdots a_{nj_n}$。

项 $a_{1j_1}a_{2j_2}\cdots a_{nj_n}$ 前面的符号是 $(-1)^{\pi(j_1j_2\cdots j_n)}$，也就是说，当 $j_1j_2\cdots j_n$ 是偶排列时，符号为正，当 $j_1j_2\cdots j_n$ 是奇排列时，符号为负。

对于二三阶行列式的对角线法则，事实上就是采用了全排列定义。四阶以上行列式不再适用于对角线法则，也是同样的原因。特别地，一阶行列式就是元素本身。

定理：从 n 阶行列式的第 $i_1,i_2,\cdots,i_n$ 行和第 $j_1,j_2,\cdots,j_n$ 列取出元素做乘积

$$a_{i_1j_1}a_{i_2j_2}\cdots a_{i_nj_n}$$

这里 $i_1,i_2,\cdots,i_n$ 和 $j_1,j_2,\cdots,j_n$ 都是 $1,2,\cdots,n$ 这 n 个数的排列。那么这一项在行列式中的符号是 $(-1)^{s+t}$，其中

$$s=\pi(i_1i_2\cdots i_n)$$

$$t=\pi(j_1j_2\cdots j_n)$$

定理：行列式和它的转置行列式相等。

定理：设行列式 $\det A$ 的第 i 行的所有元素都可以表示成两项的和：

$$\begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & & \vdots \\ b_{i1}+c_{i1} & b_{i2}+c_{i2} & \cdots & b_{in}+c_{in} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix}$$

那么该行列式等于两个行列式 $\det A_1$ 和 $\det A_2$ 的和。其中 A_1 的第 i 行是 $b_{i1}, b_{i2}, \cdots, b_{in}$，$A_2$ 的第 i 行是 $c_{i1}, c_{i2}, \cdots, c_{in}$，$A_1$ 和 A_2 的其余各行都和 A 相同。同样的性质对于列来说也成立。

归纳方法定义

这种方法只是描述了行列式的一种代数性质，时间复杂度也为阶乘量级，不适合用于计算。

代数余子式

在 n 阶行列式 $\det A$ 中，任意取定 k 行和 k 列。位于这些行列相交处的元素构成的 k 阶行列式叫做该行列式的 k 阶子式。

对于 n 阶行列式 $\det A$，某一元素 a_{ij} 的余子式 M_{ij} 指的是该行列式中，划去 a_{ij} 所在的行和列后，余下的 $n-1$ 阶子式。

对于 n 阶行列式 $\det A$，元素 a_{ij} 的余子式 M_{ij} 附以符号 $(-1)^{i+j}$ 之后，叫做元素 a_{ij} 的代数余子式，用符号 A_{ij} 表示。

从上一节全排列方法的定义可以推出结论：

定理：若在一个 n 阶行列式 $\det A$ 中，第 i 行或第 j 列的元素除了 a_{ij} 都是 0，那么这个行列式等于 a_{ij} 和它的代数余子式 A_{ij} 的乘积。

行列式展开

由于方阵转置，行列式不变，只需介绍按行展开或按列展开之一即可。

行列式 $\det A$ 定义为它任意一行（或一列）的所有元素与它们的对应代数余子式乘积的和。

换句话说，行列式可以使用按行（或按列）的展开式递归定义：

$$\det A = a_{i1}A_{i1} + a_{i2}A_{i2} + \cdots + a_{in}A_{in}$$

$$\det A = a_{1j}A_{1j} + a_{2j}A_{2j} + \cdots + a_{nj}A_{nj}$$

于是有结论：

定理：行列式 $\det A$ 的某一行（或某一列）的元素与另外一行（或另外一列）对应元素的代数余子式的乘积之和等于 0。

换句话说，当 $i \neg j$ 时：

$$a_{i1}A_{j1} + a_{i2}A_{j2} + \cdots + a_{in}A_{jn} = 0$$

$$a_{1i}A_{1j} + a_{2i}A_{2j} + \cdots + a_{ni}A_{nj} = 0$$

代数方法定义

代数方法定义是说，满足了某些性质的运算只能是行列式。

利用行列式有关初等变换的性质，可以方便手动计算更高阶的行列式。后文的「高斯消元」方法计算行列式，也用到了这个性质，时间复杂度为 $O(n^3)$。

对于一个 n 阶矩阵 A 的运算，如果满足以下三个性质，称为行列式：

- 把一个行列式的某一行或某一列的所有元素同时乘以一个数 k，等于用 k 乘这个行列式。
- 交换一个行列式的两行或两列，行列式改变符号。
- 把行列式的某一行或某一列的元素乘以同一数后加到另一行或另一列的对应元素上，行列式不变。

上述性质也有若干推论：

- 一个行列式中某一行或某一列的公因子可以提到行列式符号的外边。
- 如果一个行列式的某一行或某一列的元素全部是 0，那么这个行列式等于 0。
- 如果一个行列式有两行或两列的对应元素成比例，那么这个行列式等于 0。
- 如果一个行列式有两行或两列完全相同，那么这个行列式等于 0。

这些推论在手算行列式的时候非常常用。

9.15.7 线性空间

Authors: codewasp942, Tiphereth-A

前置知识：阿贝尔群、域。

通俗地讲，一个集合关于某运算封闭，满足结合律、单位元与逆元则构成群。如果还满足交换律，则构成阿贝尔群。

如果一个集合关于四则运算封闭，则构成域。相关定义详见 群论简介。

定义

线性空间（向量空间）是线性代数的基本概念与重要研究对象。线性空间是由向量集合 V、域 $\mathbb{P}$、加法运算 $+$ 和标量乘法（数乘）组成的模类代数结构。

具体来说，设 $(V,+)$ 是一个阿贝尔群，$\mathbb{P}$ 是一个域。

定义 $\mathbb{P}$ 中的数与 V 中元素的一种代数运算，称为**数乘**：$\cdot : \mathbb{P} \times V \mapsto V$，记为 $p \cdot v$ 或 pv，其中 p 在域 $\mathbb{P}$ 中，v 在阿贝尔群 V 中。要求该数乘运算是封闭的，运算结果始终有意义，也在群 V 中。

且满足以下条件：

1. **数乘对向量加法分配律**：对于 $\mathbf{u},\mathbf{v} \in V, a \in \mathbb{P}$，$a(\mathbf{u}+\mathbf{v}) = a\mathbf{u} + a\mathbf{v}$
2. **数乘对标量加法分配律**：对于 $a,b \in \mathbb{P}, \mathbf{u} \in V$，$(a+b)\mathbf{u} = a\mathbf{u} + b\mathbf{u}$
3. **数乘结合律（一致于域乘法）**：对于 $a,b \in \mathbb{P}, \mathbf{u} \in V$，$a(b\mathbf{u}) = (ab)\mathbf{u}$
4. **标量乘法单位元**：令 $1 \in \mathbb{P}$ 是 $\mathbb{P}$ 的乘法单位元，则对于 $u \in V$，$1\mathbf{u} = \mathbf{u}$

则称代数系统 $(V,+,\cdot,\mathbb{P})$ 是 V 关于 $+,\cdot$ 构成 $\mathbb{P}$ 上的一个**线性空间**，$\mathbb{P}$ 为线性空间的**基域**，V 中元素称为**向量**，$\mathbb{P}$ 中元素称为**标量**。当域 $\mathbb{P}$ 为实数域时，称为实线性空间。当域 $\mathbb{P}$ 为复数域时，称为复线性空间。

不管是一列数还是箭头，或是别的什么东西，只要满足上述公理，都可以认为是向量，也就都可以利用线性代数的理论来研究。

称加法群中的零元为零向量，记作 $\mathbf{0}$ 或 $\boldsymbol{\theta}$。

原阿贝尔群中向量的加减法，与线性空间新定义的数乘，统称为**线性运算**。

note

为行文方便，下文中：

1. 对 V 中的元素不做加粗处理。
2. 将满足线性空间定义的代数系统 $(V,+,\cdot,\mathbb{P})$ 也称为线性空间。

请注意区分。

直观理解

不是很严谨地说，标量乘法对应着一种「**缩放**」，基域 $\mathbb{P}$ 中的元素就代表着缩放的「**比例**」，向量加法对应「**叠加**」。同时，$\mathbb{P}$ 中的元素还代表着向量的「**坐标**」的取值范围。

条件 1-4 描述的是「缩放」与「叠加」的关联。可以结合二维平面上的箭头来理解。

简单性质

note

以下性质可在群论等中找到。

对线性空间 $(V, +, \cdot, \mathbb{P})$,

1. θ 唯一
2. $\forall \alpha \in V$,$-\alpha$ 唯一
3. $\exists 0 \in \mathbb{P}$,$\forall \alpha \in V$, 有 $0\alpha = \theta$
4. $\forall k \in \mathbb{P}$, 有 $k\theta = \theta$
5. $(-1)\alpha = -\alpha$, $\forall \alpha \in V$
6. 无零因子：$\forall \alpha \in V, k \in \mathbb{P}$, 有 $k\alpha = \theta \implies k = 0 \vee \alpha = \theta$
7. 加法的消去律：$\forall \alpha, \beta, \gamma \in V$, 有 $\alpha + \beta = \alpha + \gamma \implies \beta = \gamma$

> 实际上，加法的消去律是阿贝尔群的性质。

例子

1. $\mathbb{P}^n$ 关于数域 $\mathbb{P}$ 上的加法和乘法构成 $\mathbb{P}$ 上的一个线性空间。例如 $\mathbb{P}$ 可以是 $\mathbb{R}$,$\mathbb{C}$,$\mathbb{N}_p$（p 为素数）等。
2. 数域 $\mathbb{P}$ 上的 $n \times m$ 阶矩阵 $\mathbb{P}^{n\times m}$ 关于矩阵的加法和数乘构成 $\mathbb{P}$ 上的一个线性空间。
3. 数域 $\mathbb{P}$ 上的一元多项式环 $\mathbb{P}[x]$ 关于多项式的加法和数乘构成 $\mathbb{P}$ 上的一个线性空间。
4. 区间 $[a, b]$ 上的全体连续函数（记作 $C[a, b]$）关于「函数加法」和「值与连续函数的数乘」构成值域上的一个线性空间。

相关概念

线性相关、线性无关

对线性空间 $(V, +, \cdot, \mathbb{P})$:

1. 称 $a_1, a_2, \dots, a_n \in V$ 为 V 的一个**向量组**。
2. 对于 $k_1, k_2, \dots, k_n \in \mathbb{P}$, 称 $\sum_{i=1}^n k_i a_i$ 为向量组 $a_1, a_2, \dots, a_n$ 的一个**线性组合**。
3. 若向量 $\beta \in V$ 可以表示为向量组 $a_1, a_2, \dots, a_n$ 的一个线性组合，则称 β 能被向量组 $a_1, a_2, \dots, a_n$ **线性表出**。
4. 对于 $k_1, k_2, \dots, k_n \in \mathbb{P}$, 若向量组 $a_1, a_2, \dots, a_n$ 满足 $\sum_{i=1}^n k_i a_i = \theta \iff k_i = 0, i = 1, 2, \dots, n$, 则称向量组 $a_1, a_2, \dots, a_n$ **线性无关**，否则称向量组 $a_1, a_2, \dots, a_n$ **线性相关**。

规定零向量与任意向量线性相关。

线性表示或线性相关的式子，可以写成矩阵乘法的形式：

$$\beta = k_1 a_1 + k_2 a_2 + \cdots + k_r a_r = (a_1, a_2, \cdots, a_r) \begin{pmatrix} k_1 \\ k_2 \\ \vdots \\ k_r \end{pmatrix}$$

根据习惯，把向量 a 按顺序并排写在左边；把标量 k 按顺序竖着写在右边，构成一个「列向量」。

注意：这里标量构成的「列向量」只是方便的形式记号，不在空间 V 中，与左边的向量有着本质的区别。左边的向量如果恰好是列向量，并排拼起来就可以形式上构成一个「矩阵」，上述乘积恰好是矩阵中常见的「矩阵左乘列向量」的形式。

下文指出，这里的线性表示也等价于，向量 β 落在矩阵 $(a_1, a_2 \cdots, a_r)$ 的像空间里。

根据下文中的定义，零向量一定会落在像空间里。如果用线性变换的观点看，线性相关等价于变换后多个向量变换到零向量，而线性无关等价于只有零向量本身变换到零向量。

性质

对线性空间 $(V, +, \cdot, \mathbb{P})$,

1. 若向量组的一部分线性相关，则向量组线性相关。若向量组线性无关，则其任意非空部分均线性无关。简记为：「**大无关、小无关**」；「**小相关、大相关**」。
2. 含 θ 的向量组线性相关。
3. 向量组线性相关当且仅当向量组的某个向量可以由其余向量线性表出。
4. 若向量 β 可被向量组 $a_1, a_2, \dots, a_n$ 线性表出，则表出方式唯一当且仅当向量组 $a_1, a_2, \dots, a_n$ 线性无关。
5. 若向量组 $a_1, a_2, \dots, a_n$ 线性无关，则向量 β 可被向量组 $a_1, a_2, \dots, a_n$ 线性表出当且仅当向量组 $a_1, a_2, \dots, a_n, \beta$ 线性相关。

极大线性无关组、秩

线性相关可以理解为「多余」，说明向量组内部有的向量可以被其他向量表出，可以删去。删完了之后，将剩下极大线性无关组。

对线性空间 $(V, +, \cdot, \mathbb{P})$：

1. 对于向量组 $b_1, b_2, \dots, b_m$, 令 $\{a_1, a_2, \dots, a_n\} \subseteq \{b_1, b_2, \dots, b_m\}$, 若有：

   - 向量组 $a_1, a_2, \dots, a_n$ 线性无关。
   - $\forall \beta \in \{b_1, b_2, \dots, b_m\} \quad \{a_1, a_2, \dots, a_n\}$, 向量组 $a_1, a_2, \dots, a_n, \beta$ 线性相关。

   则称向量组 $a_1, a_2, \dots, a_n$ 为向量组 $b_1, b_2, \dots, b_m$ 中的一个**极大线性无关组**。类似地，可定义线性空间 V 的极大线性无关组。

   规定向量组 $\theta, \theta, \dots, \theta$ 的极大线性无关组为空集，于是全 0 矩阵对应的向量组没有极大线性无关组。

   从向量组删向量的删法不唯一，因此极大线性无关组也不唯一。习惯上从左到右按顺序删。

   很巧的是，按顺序删，留下的向量，恰好就是「按行看」观点里面，高斯消元法剩下的行最简形矩阵中，元素 1 所在的列。

   称向量组 $b_1, b_2, \dots, b_m$ 的极大线性无关组的大小为向量组的**秩**, 记作 $\operatorname{rank}\{b_1, b_2, \dots, b_m\}$, 规定 $\operatorname{rank}\{\theta, \theta, \dots, \theta\} = 0$。

   于是，向量组的秩的定义与矩阵的秩的定义完全一致。

2. 若向量组 $a_1, a_2, \dots, a_n$ 能线性表出向量组 $b_1, b_2, \dots, b_m$ 中的所有向量，称向量组 $b_1, b_2, \dots, b_m$ 能被向量组 $a_1, a_2, \dots, a_n$ 线性表出。
3. 若向量组 $a_1, a_2, \dots, a_n$ 能被向量组 $b_1, b_2, \dots, b_m$ 线性表出，且向量组 $b_1, b_2, \dots, b_m$ 能被向量组 $a_1, a_2, \dots, a_n$ 线性表出，则称两向量组**等价**，记作 $\{a_1, a_2, \dots, a_n\} \cong \{b_1, b_2, \dots, b_m\}$。

   向量组的**等价**就是向量组张成的空间相同。张成空间相同的向量组相互等价，张成空间不同的向量组不等价。

   向量组等价比矩阵等价条件更强，不仅要求秩相同，还要求空间完全一样。因此，把两个矩阵**横向**拼在一起，秩不能发生变化。

   矩阵等价仅要求秩相同，因此矩阵等价表示前一个矩阵或空间，可以通过可逆变换，到达后一个矩阵或空间。

性质

对线性空间 $(V, +, \cdot, \mathbb{P})$,

1. 设向量组 $a_1, a_2, \dots, a_n$ 能被线性表出向量组 $b_1, b_2, \dots, b_m$ 线性表出。
   - 若 $n > m$, 则向量组 $a_1, a_2, \dots, a_n$ 线性相关。
   - 若向量组 $a_1, a_2, \dots, a_n$ 线性无关，则 $n \le m$。
2. 等价的线性无关向量组的大小相等。

   向量组的任意极大线性无关组的大小均相等。
3. 向量组线性无关当且仅当其秩等于其大小。
4. 若向量组 $a_1, a_2, \dots, a_n$ 能被线性表出向量组 $b_1, b_2, \dots, b_m$ 线性表出，则 $\mathrm{rank}\{a_1, a_2, \dots, a_n\} \le \mathrm{rank}\{b_1, b_2, \dots, b_m\}$。
5. 等价的向量组的秩相等。

线性包

对于线性空间 $(V, +, \cdot, \mathbb{P})$，$\left\{v = \sum_{i=1}^{n} k_i a_i : a_i \in V, k_i \in \mathbb{P}, i = 1, 2, \dots, n\right\}$ 也构成一个线性空间，称为由向量组 $a_1, a_2, \dots, a_n$ **张成**的线性空间（或**线性包**），记作 $\mathrm{span}\{a_1, a_2, \dots, a_n\}$。

这里的 n 个向量 a 不一定线性无关。

线性子空间

对线性空间 $(V, +, \cdot, \mathbb{P})$, 若代数系统 $(V_1, +, \cdot, \mathbb{P})$ 满足：

1. $\emptyset \ne V_1$
2. $V_1 \subseteq V$
3. V_1 关于 $+, \cdot$ 构成 $\mathbb{P}$ 上的线性空间

则称 V_1 为 V 的线性子空间，简称子空间，记作 $V_1 \le V$。

任何空间 V 都有两个**平凡子空间**：它本身 V 与零子空间。零子空间只含零向量，不含有线性无关的向量。

若第 2 条中的 $\subseteq$ 换为 $\subset$, 则称 V_1 为 V 的线性真子空间，记作 $V_1 < V$。

不难证明：线性空间 V 的非空子集 V_1 是其线性子空间当且仅当线性运算在 V_1 上封闭，即：

1. $\forall u, v \in V_1$,$u + v \in V_1$
2. $\forall v \in V_1$,$\forall k \in \mathbb{P}$,$kv \in V_1$

交、和与直和、直积

对线性空间 $(V_1, +, \cdot, \mathbb{P})$ 与 $(V_2, +, \cdot, \mathbb{P})$：

1. 不难验证：加法和数乘在 $V_1 \cap V_2$ 上封闭，故可称 $V_1 \cap V_2$ 为线性空间 V_1 和 V_2 的**交**。

   类似地，可定义多个线性空间的交 $\bigcap_{i=1}^{m} V_i$。
2. 若线性空间 V 满足 $V = \{u + v | u \in V_1, v \in V_2\}$, 则称 V 为线性空间 V_1 和 V_2 的**和**，记为 $V = V_1 + V_2$。

   可以验证：$V_1 + V_2$ 是包含 $V_1 \cup V_2$ 的最小子空间。

   类似地，可定义多个线性空间的和 $\sum_{i=1}^{m} V_i$。
3. 设 $V = V_1 + V_2$, 若线性空间 V 中的任意元素 v, 均只能找到唯一一组向量 v_1, v_2 满足 $v = v_1 + v_2$, 则称 V 为线性空间 V_1 和 V_2 的**直和** (direct sum)，记为 $V_1 \oplus V_2$。

   类似地，可定义多个线性空间的直和 $\bigoplus_{i=1}^{m} V_i$。
4. V_1 与 V_2 的**直积** $V_1 \times V_2$ 定义为二者的笛卡儿积关于如下的加法和数乘构成 $\mathbb{P}$ 上的线性空间：
   (a) $+ : (V_1 \times V_2) \times (V_1 \times V_2) \mapsto V_1 \times V_2; ((u_1, v_1), (u_2, v_2)) \to (u_1 + u_2, v_1 + v_2)$
   (b) $\cdot : \mathbb{P} \times (V_1 \times V_2) \mapsto V_1 \times V_2; (k, (u, v)) \to (ku, kv)$

   类似地，可定义多个线性空间的直积 $\prod_{i=1}^{m} V_i$。

例子

对于线性空间 $V = \mathbb{R}^3$，设线性空间：

- $V_1 := \{(x,0,0)|x \in \mathbb{R}\}$
- $V_2 := \{(x,y,0)|x,y \in \mathbb{R}\}$
- $V_3 := \{(0,y,z)|y,z \in \mathbb{R}\}$
- $V_4 := \{(x,0,z)|x,z \in \mathbb{R}\}$

则

1. $V_1 < V_2 < V$,$V_3 < V$
2. $V_2 = V_1 + V_2$
3. $V = V_1 \oplus V_3 = V_2 + V_3$
4. $V_2 \oplus V_3 = V_4$,$V_2 \oplus V_4 = V_3$,$V_3 \oplus V_4 = V_2$
5. $V_2 + V_3 \leq V$

性质

1. 令 V_1, V_2, V_3 是关于 $\mathbb{P}$ 的线性空间，和集合的交一样，线性空间的交适用如下法则：
   (a) 交换律：$V_1 \cap V_2 = V_2 \cap V_1$
   (b) 结合律：$V_1 \cap (V_2 \cap V_3) = (V_1 \cap V_2) \cap V_3$
2. 令 V_1, V_2, V_3 是关于 $\mathbb{P}$ 的线性空间，类似于集合的并，线性空间的和适用如下法则：
   (a) 交换律：$V_1 + V_2 = V_2 + V_1$
   (b) 结合律：$V_1 + (V_2 + V_3) = (V_1 + V_2) + V_3$
3. 令 V_1, V_2, V_3 是关于 $\mathbb{P}$ 的线性空间，线性空间的交与并有如下关系：
   (a) $V_1 \cap (V_2 + V_3) \supseteq (V_1 \cap V_2) + (V_1 \cap V_3)$
   (b) $V_1 + (V_2 \cap V_3) \subseteq (V_1 + V_2) \cap (V_1 + V_3)$
4. $\operatorname{span}\{a_1, a_2, \dots, a_n\} + \operatorname{span}\{b_1, b_2, \dots, b_m\} = \operatorname{span}\{a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_m\}$
5. 令 V_1, V_2 是关于 $\mathbb{P}$ 的线性空间，则下列诸款等价：
   (a) $V_1 + V_2 = V_1 \oplus V_2$
   (b) $\exists \beta \in V_1 + V_2$, 使得拆分为 V_1 和 V_2 中的向量和的方式唯一（任意 $\to$ 存在）
   (c) θ 拆分为 V_1 和 V_2 中向量的和的方式唯一
   (d) $V_1 \cap V_2 = \{\theta\}$

" 证明"

$1 \implies 2$：由定义立得。

$2 \implies 3$：

令 $\beta = \beta_1 + \beta_2$, 其中 $\beta_1 \in V_1, \beta_2 \in V_2$, 若 $\theta = \alpha_1 + \alpha_2, \theta \neq \alpha_1 \in V_1, \alpha_2 \in V_2$, 则 $\beta = \beta + \theta = (\beta_1 + \alpha_1) + (\beta_2 + \alpha_2)$。

而 $\beta_1 \neq \beta_1 + \alpha_1$, 与条件矛盾。

$3 \implies 4$：

在 V_1 和 V_2 中取一非零向量 α, 则 $\theta = \alpha + (-\alpha) = (-\alpha) + \alpha$, 这与条件矛盾。

$4 \implies 1$：

若 $V_1 + V_2$ 不是直和，则存在 $\beta \in V_1 + V_2$ 使得 $\beta = \beta_1 + \beta_2 = \gamma_1 + \gamma_2$, 其中 $\beta_1, \gamma_1 \in V_1, \beta_2, \gamma_2 \in V_2$ 且 $\beta_1, \beta_2, \gamma_1, \gamma_2$) 互不相同。

进而 $\theta \neq \beta_1 - \gamma_1 = \gamma_2 - \beta_2 \in V_1 \cap V_2$, 与条件矛盾。

同构

设 V, V' 均为域 $\mathbb{P}$ 上的线性空间，若存在双射 $\sigma : V \mapsto V'$ 且保持加法与数乘，即 $\forall u, v \in V, \forall k \in \mathbb{P}$ 满足：

1. $\sigma(u+v) = \sigma(u) + \sigma(v)$
2. $\sigma(ku) = k\sigma(u)$

则称 σ 是 V 到 V' 的**同构映射**，此时称 V 与 V' **同构**，记为 $V \cong V'$。

> note
>
> 若 σ 是单射，则可定义**单同态**；若 σ 是满射，则可定义**满同态**。

性质

1. 域 $\mathbb{P}$ 上的两线性空间同构当且仅当其维数相等。
2. （1 的推论）域 $\mathbb{P}$ 上的 n 维线性空间与线性空间 $\mathbb{P}^n$ 同构。

   > note
   >
   > 本性质说明我们基本上可以将坐标和向量等同看待。

应用

从本节开始主要讲述对于线性方程组「按列看」的观点。

矩阵 A 本身也是由列向量构成的。把 A 本身看成了列向量组，而 x 是未知数系数，思考 A 当中的这组列向量能不能配上未知数，凑出列向量 b。此时列向量 x 是完全未知的。

此时研究的等式 $Ax = b$ 整理为：

$$\alpha_1 x_1 + \alpha_2 x_2 + \cdots + \alpha_n x_n = b$$

这时，矩阵乘法中，位于左边的矩阵 A 可以看作向量组，即一组列向量。这组列向量作为一组基，张成一个空间，探讨列向量 b 是否落在这个空间里。

按列看待线性方程组的解

秩是极大线性无关组中向量的个数，代表了「约束」。那么其余的向量将赋予解的自由度，即允许在其他方向赋予冗余的向量。

如果记 n 是矩阵 A 的列数，即含有的列向量个数，记 $r(A)$ 为矩阵 A 的秩，则有自由度 S：

$$S = n - r(A)$$

方程组的全体解也构成向量组，自由度 S 就是 $Ax = 0$ 解向量组的秩，即下文核空间的维数。

方程组的同解

两个方程组的公共解定义为两组解的交集。

方程组的**同解**就是方程组的解的集合相等。解的集合相等的方程组同解，解的集合不相等的方程组不同解。

方程组同解也比矩阵等价条件强，不仅要求秩相等，还要求把两个矩阵**纵向**拼在一起之后，秩仍然不改变。

这里与向量组等价对比，向量组等价要求矩阵横向拼接，秩不改变。因此，有如下关系：

矩阵等价，不一定有对应的向量组等价或者方程组同解，但是若有向量组等价或者方程组同解，必然有对应的矩阵等价（秩相同）。

如果矩阵对应的向量组等价，那么将矩阵转置后，对应的方程组同解，反之亦然。

矩阵的核空间与像空间

这部分的核空间与像空间是站在线性空间的角度上叙述的。

对于矩阵 A，令 W 为方程 $Ax=0$ 的全体解 x 构成的集合，则 W 是一个线性空间，W 的标量域与 A 的元素所在的域相同。

称此时的 W 为矩阵 A 的**核空间**，记作 $N(A)$。

矩阵 A 的核空间 $N(A)$ 就是方程 $Ax=0$ 的**解空间**。根据后文基的定义，该方程的**基础解系**就是核空间的基。

如果矩阵 A 是可逆矩阵，则 A 的核空间 $N(A)$ 只含零向量。

对于矩阵 A，它的 n 个列为向量 α，称 n 个列向量 α 张成的空间为 A 的**像空间**，或者记作**列空间**，记作：

$$R(A)=\operatorname{span}\{\alpha_1,\alpha_2,\cdots,\alpha_n\}$$

根据后文维数的定义，像空间的维数等于矩阵 A 的秩。

由定义，对于像空间 $R(A)$ 中的每一个元素 y，均有相应的表示：

$$y=k_1\alpha_1+k_2\alpha_2+\cdots+k_n\alpha_n=(\alpha_1,\alpha_2,\cdots,\alpha_n)\begin{pmatrix}k_1\\k_2\\\vdots\\k_n\end{pmatrix}=A\begin{pmatrix}k_1\\k_2\\\vdots\\k_n\end{pmatrix}$$

因此像空间 $R(A)$ 就是对于任意向量 x，Ax 的**值域**。

同理可以定义 A 的**行空间**，即 A 的转置的值域 $R(A^T)$。

由于矩阵的行秩等于列秩，行空间的维数也为矩阵的秩，因此转置改变像空间，而不改变像空间的维数。

在这里可以与前文建立对应关系：

向量组等价，等价于对应矩阵的像空间 $R(A)$ 相同。

方程组同解，等价于对应矩阵的行空间 $R(A^T)$ 相同。

参考资料与注释

1. 丘维声，高等代数（下）。清华大学出版社。
2. Vector space[1]. *Wikipedia, The Free Encyclopedia.*

参考资料与注释

[1] Vector space

9.15.8 线性基

Authors: cesonic, Enter-tainer, Great-designer, Ir1d, ksyx, lychees, MegaOwIer, RUIN-RISE, wjy-yy, rsdbkhusky, ouuan, Menci, Tiphereth-A

前置知识：线性空间 的定义以及相关概念中的线性相关、线性无关、极大线性无关组、子空间等。

线性基是线性空间的一组基，是研究线性空间的重要工具。

由于 OI 中只涉及有限维线性空间，故本处仅介绍有限维线性空间的情况。

定义

称线性空间 V 的一个极大线性无关组为 V 的一组 **Hamel 基**或**线性基**，简称**基**。

规定线性空间 $\{\theta\}$ 的基为空集。

另外，将 V 的**维数**记作 $\dim V$，定义为基的元素个数。

性质

1. 对于有限维线性空间 V, 设其维数为 n, 则:
   (a) V 中的任意 $n+1$ 个向量线性相关。
   (b) V 中的任意 n 个线性无关的向量均为 V 的基。
   (c) 若 V 中的任意向量均可被向量组 $a_1, a_2, \ldots, a_n$ 线性表出，则其是 V 的一个基。

   "证明"

   任取 V 中的一组基 $b_1, b_2, \ldots, b_n$, 由已知条件，向量组 $b_1, b_2, \ldots, b_n$ 可被 $a_1, a_2, \ldots, a_n$ 线性表出，故

   $$n = \operatorname{rank}\{b_1, b_2, \ldots, b_n\} \leq \operatorname{rank}\{a_1, a_2, \ldots, a_n\} \leq n$$

   因此 $\operatorname{rank}\{a_1, a_2, \ldots, a_n\} = n$

   (d) V 中任意线性无关向量组 $a_1, a_2, \ldots, a_m$ 均可通过插入一些向量使得其变为 V 的一个基。
2. (子空间维数公式) 令 V_1, V_2 是关于 $\mathbb{P}$ 的有限维线性空间, 且 V_1+V_2 和 $V_1 \cap V_2$ 也是有限维的, 则 $\dim V_1 + \dim V_2 = \dim(V_1+V_2) + \dim(V_1 \cap V_2)$

   "证明"

   设 $\dim V_1 = n_1$,$\dim V_2 = n_2$,$\dim(V_1 \cap V_2) = m$.
   取 $V_1 \cap V_2$ 的一组基 $a_1, a_2, \ldots, a_m$, 将其分别扩充为 V_1 和 V_2 中的基: $a_1, a_2, \ldots, a_m, b_1, b_2, \ldots, b_{n_1-m}$ 和 $a_1, a_2, \ldots, a_m, c_1, c_2, \ldots, c_{n_2-m}$.
   接下来只需证明向量组 $a_1, a_2, \ldots, a_m, b_1, b_2, \ldots, b_{n_1-m}, c_1, c_2, \ldots, c_{n_2-m}$ 线性无关即可。
   设 $\sum_{i=1}^{m} r_i a_i + \sum_{i=1}^{n_1-m} s_i b_i + \sum_{i=1}^{n_2-m} t_i c_i = \theta$.
   则 $\sum_{i=1}^{n_2-m} t_i c_i = -\sum_{i=1}^{m} r_i a_i - \sum_{i=1}^{n_1-m} s_i b_i$.
   注意到上式左边在 V_2 中，右边在 V_1 中，故两边均在 $V_1 \cap V_2$ 中，因此 $\sum_{i=1}^{n_2-m} t_i c_i = \sum_{i=1}^{m} k_i a_i$
   故 $t_1 = t_2 = \cdots = t_{n_2-m} = k_1 = k_2 = \cdots = k_m = 0$, 进而 $r_1 = r_2 = \cdots = r_m = s_1 = s_2 = \cdots = s_{n_1-m} = t_1 = t_2 = \cdots = t_{n_2-m} = 0$
3. 令 V_1, V_2 是关于 $\mathbb{P}$ 的有限维线性空间，且 V_1+V_2 和 $V_1 \cap V_2$ 也是有限维的，则下列诸款等价:
   (a) $V_1 + V_2 = V_1 \oplus V_2$.
   (b) $\dim V_1 + \dim V_2 = \dim(V_1+V_2)$.
   (c) 若 $a_1, a_2, \ldots, a_n$ 是 V_1 的一组基，$b_1, b_2, \ldots, b_m$ 是 V_2 的一组基，则 $a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m$ 是 V_1+V_2 的一组基。

   note

   1,3 两条可推广到无限维线性空间中

例子

考虑 $\mathbb{R}^2$ 的基。

1. 如图
   u, v 是一组基。
2. 如图
   u, v 是一组基。
3. 如图
   u, v 不是一组基，因为 $u = -v$.

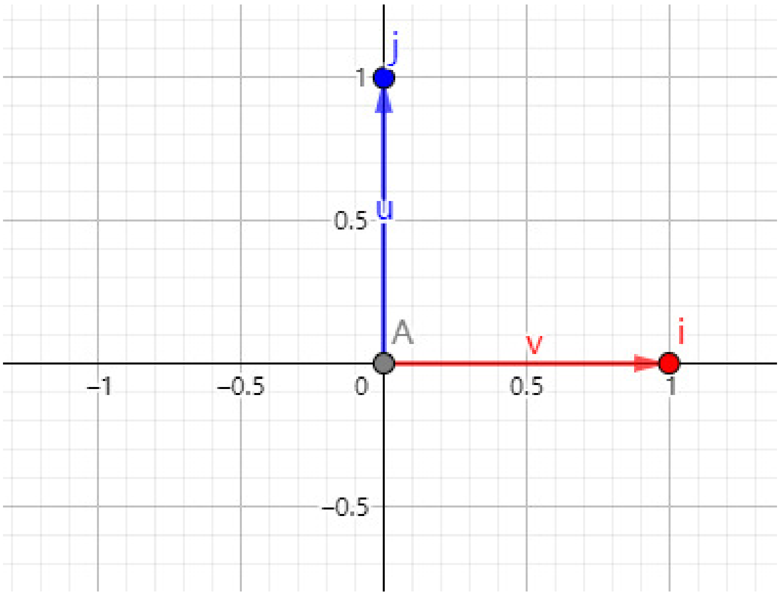


图 9.27

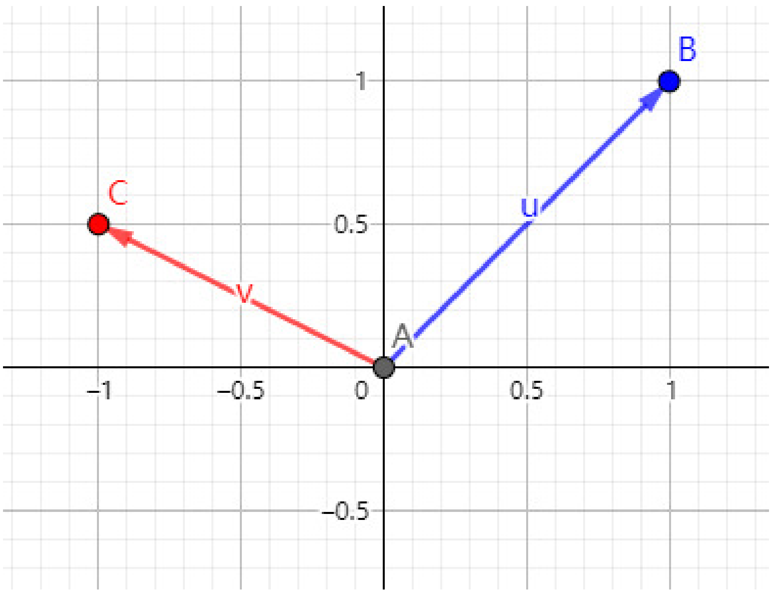


图 9.28

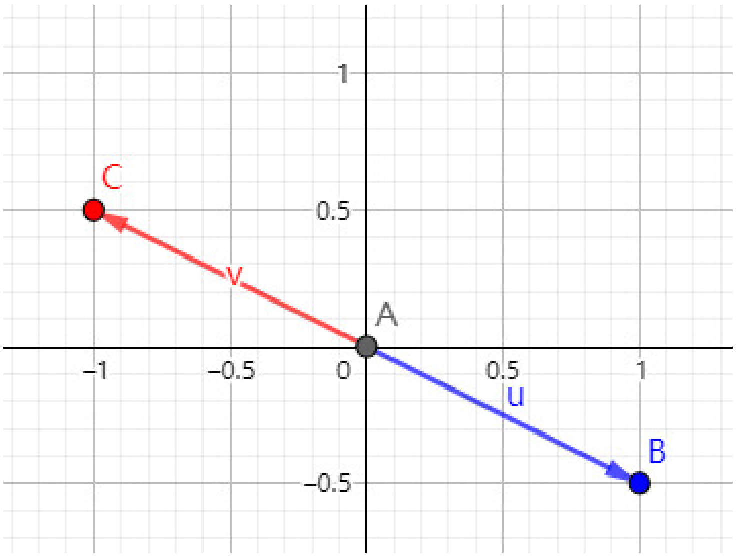


图 9.29

4. 如图

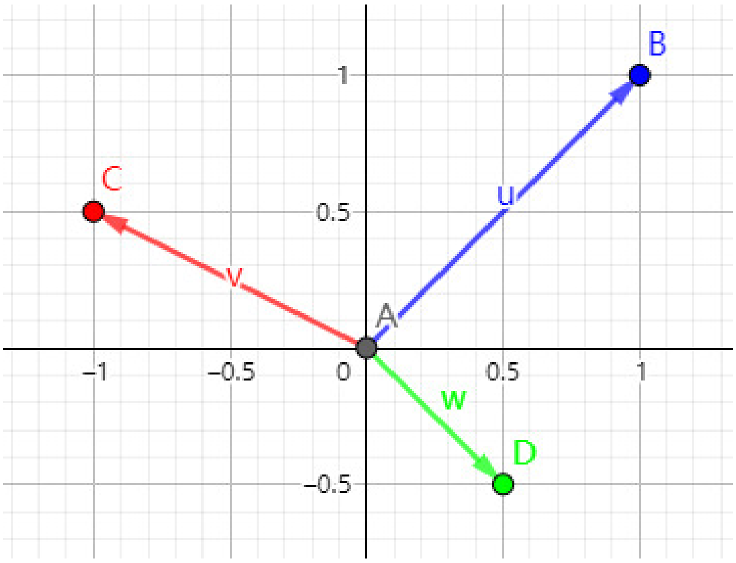


图 9.30

u, v, w 不是一组基，因为 $u + 4v + 6w = \theta$.

应用

线性基在 OI 中的应用一般包含以下方面：

1. 对给定的向量组，找到一组极大线性无关组（或其张成的线性空间的一组基）。
2. 向给定的向量组插入某些向量，在插入操作后的向量组中找到一组极大线性无关组（或其张成的线性空间的一组基）。
3. 对找到的一组极大线性无关组（或基），判断某向量能否被其线性表出
4. 求极大线性无关组（或基）的秩。
5. 对找到的一组极大线性无关组（或基），求其张成的线性空间中的最大元/最小元。

特别地：

- 若限定向量均在 $\mathbb{Z}_2^n$ 下，则称找到的基为**异或线性基**。
- 若限定向量均在 $\mathbb{R}^n$ 下，则称找到的基为**实数线性基**。

构造方法

因为异或线性基与实数线性基没有本质差别，所以接下来以异或线性基为例，实数线性基版本的代码只需做一点简单修改即可。

贪心法

对原集合的每个数 p 转为二进制，从高位向低位扫，对于第 x 位是 1 的，如果 a_x 不存在，那么令 $a_x \leftarrow p$ 并结束扫描，如果存在，令 $p \leftarrow p \operatorname{xor} a_x$。

查询原集合内任意几个元素 xor 的最大值，只需将线性基从高位向低位扫，若 xor 上当前扫到的 a_x 答案变大，就把答案异或上 a_x。

为什么能行呢？因为从高往低位扫，若当前扫到第 i 位，意味着可以保证答案的第 i 位为 1，且后面没有机会改变第 i 位。

查询原集合内任意几个元素 xor 的最小值，就是线性基集合所有元素中最小的那个。

查询某个数是否能被异或出来，类似于插入，如果最后插入的数 p 被异或成了 0，则能被异或出来。

" 代码（洛谷 P3812 【模板】线性基[1]）"

```cpp
#include <bits/stdc++.h>
using ull = unsigned long long;

ull p[64];

void insert(ull x) {
  for (int i = 63; ~i; --i) {
    if (!(x >> i))  // x 的第 i 位是 0
      continue;
    if (!p[i]) {
      p[i] = x;
      break;
    }
    x ^= p[i];
  }
}

int main() {
  int n;
  scanf("%d", &n);
  ull a;
  for (int i = 1; i <= n; ++i) {
    scanf("%llu", &a);
    insert(a);
  }
  ull ans = 0;
  for (int i = 63; ~i; --i) {
    ans = std::max(ans, ans ^ p[i]);
  }
  printf("%llu\n", ans);
  return 0;
}
```

高斯消元法

高斯消元法相当于从线性方程组的角度去构造线性基，正确性显然。

" 代码（洛谷 P3812 【 模板 】 线性基[1]）"

```
#include <bits/stdc++.h>
using ull = unsigned long long;
const int MAXN = 1e5 + 5;

ull deg(ull num, int deg) { return num & (1ull << deg); }

ull a[MAXN];

int main() {
  int n;
  scanf("%d", &n);
  for (int i = 1; i <= n; ++i) scanf("%llu", &a[i]);
  int row = 1;
  for (int col = 63; ~col && row <= n; --col) {
    for (int i = row; i <= n; ++i) {
      if (deg(a[i], col)) {
        std::swap(a[row], a[i]);
        break;
      }
    }
    if (!deg(a[row], col)) continue;
    for (int i = 1; i <= n; ++i) {
      if (i == row) continue;
      if (deg(a[i], col)) {
        a[i] ^= a[row];
      }
    }
    ++row;
  }
  ull ans = 0;
  for (int i = 1; i < row; ++i) {
    ans ^= a[i];
  }
  printf("%llu\n", ans);
  return 0;
}
```

性质

贪心法构造的线性基具有如下性质：

- 线性基没有异或和为 0 的子集。
- 线性基中各数二进制最高位不同。

高斯消元法构造出的线性基满足如下性质：

- 高斯消元后的矩阵是一个行简化阶梯形矩阵。

> 该性质包含了贪心法构造的线性基满足的两条性质

如果不理解这条性质的正确性，可以跳转 高斯消元。

提供一组样例：

```
5
633 211 169 841 1008
```

二进制表示：

```
1001111001
0011010011
0010101001
1101001001
1111110000
```

贪心法生成的线性基：

```
1001111001
0100110000
0011010011
0001111010
0000000000
0000010000
0000000000
0000000000
0000000000
0000000000
```

高斯消元法生成的线性基：

```
1000000011
0100100000
0010101001
0001101010
0000010000
0000000000
0000000000
0000000000
0000000000
0000000000
```

这是一条非常好的性质，能帮我们更方便的解决很多问题。比如：给定一些数，选其中一些异或起来，求异或最大值，如果用贪心法构造线性基，需要再做一遍贪心，如果 `ans` 的当前位是 `0`，那么异或一定会更优，否则当前位如果为 `1`，则一定不会更优；而使用高斯消元法构造线性基后直接将线性基中所有元素都异或起来输出即可。

对于其他比较经典的问题（查询一个数能否被异或得到，查询能被异或得到的第 k 大数等），高斯消元法得到的线性基也能更加方便地解决。

时间复杂度

设向量长度为 n，总数为 m，则时间复杂度为 $O(nm)$. 其中高斯消元法的常数略大。

若是实数线性基，则时间复杂度为 $O(n^2m)$.

练习题

- Luogu P3812【模板】 线性基[1]
- Acwing 3164. 线性基[2]
- SGU 275 to xor or not xor[3]

- HDU 3949 XOR[4]
- HDU 6579 Operation[5]
- Luogu P4151[WC2011] 最大 XOR 和路径[6]

参考资料与注释

1. 丘维声，高等代数（下）。清华大学出版社。
2. Basis (linear algebra) - Wikipedia[7]
3. Vector Basis -- from Wolfram MathWorld[8]

参考资料与注释

[1] 【模板】线性基 [1-1] [1-2] [1-3]

[2] Acwing 3164. 线性基

[3] SGU 275 to xor or not xor

[4] HDU 3949 XOR

[5] HDU 6579 Operation

[6] Luogu P4151[WC2011] 最大 XOR 和路径

[7] Basis (linear algebra) - Wikipedia

[8] Vector Basis – from Wolfram MathWorld

9.15.9 线性映射

研究线性映射是研究线性空间之间的映射。

线性映射可以表示为矩阵的形式，所以在线性映射中矩阵中的大量概念都可以找到对应关系。

线性映射与线性变换

设 V 和 W 是域 F 上的两个线性空间，T 是 V 到 W 的一个映射。

如果对于 W 中任意的向量 x 和 y，域 F 中任意的标量 k 和 l，有：

$$T(kx+ly)=kTx+lTy$$

称 T 是 V 到 W 的一个线性映射。如果 $W=V$，则称 T 是 V 上的一个线性变换。

例如，恒等变换 T_e 保持空间不变，零变换 T_0 将空间映射至零空间。

可以记 $L(V,W)$ 为所有 V 到 W 的线性映射构成的集合。对于全体线性变换 $L(V,V)$，也记为 $L(V)$。

性质

- 线性映射将零向量映射到零向量。
- 线性映射保持线性运算形式不变，即，线性运算的线性映射，等于线性映射的线性运算。

- 线性映射保持线性相关性，即，映射前线性相关，映射后也线性相关。

但是线性映射不保持线性无关性。映射前线性无关，映射后不一定线性无关。

线性映射的矩阵表示

设 V 的维数是 n，V 的一组基为 $\alpha_1,\cdots,\alpha_n$，W 的维数是 m，W 的一组基为 $\beta_1,\cdots,\beta_m$，T 是 V 到 W 的一个线性映射。

将每个 α 经由 T 映射后的向量用 β 表示：

$$T\alpha_j = a_{1j}\beta_1 + \cdots + a_{mj}\beta_n$$

采用矩阵记法：

$$T(\alpha_1,\cdots,\alpha_n) = (T\alpha_1,\cdots,T\alpha_n) = (\beta_1,\cdots,\beta_m)A$$

称矩阵 A 为线性映射 T 在这两组基下的矩阵表示。

线性映射的核空间与像空间

这里的核空间与像空间是站在线性映射的视角下叙述的。借助矩阵表示可以看出，线性映射的核空间与像空间与矩阵的核空间与像空间是一致的。

设 T 是由空间 V 到空间 W 的线性映射，令：

$$N(T) = \{x \in V | Tx = 0\}$$

$$R(T) = Im(T) = \{y \in W | y = Tx, Vx \in V\}$$

易验证 $N(T)$ 为 V 的子空间，$R(T)$ 为 W 的子空间，称 $N(T)$ 及 $R(T)$ 为 V 的核空间和像空间，并称 $N(T)$ 的维数为 T 的**零度**或**亏**，$R(T)$ 的维数为 T 的**秩**。

定理：设 T 是由空间 V 到空间 W 的线性映射，V 的维数有限，则 $N(T)$ 及 $R(T)$ 均为有限维，且有：

$$\dim N(T) + \dim R(T) = \dim V$$

即 T 的亏加秩等于其定义域 V 的维数。

线性变换的矩阵表示

设 V 的维数是 n，V 的一组基为 $\alpha_1,\cdots,\alpha_n$，T 是 V 上的一个线性变换，则有：

$$T\alpha_j = a_{1j}\alpha_1 + \cdots + a_{nj}\alpha_n$$

采用矩阵记法：

$$T(\alpha_1,\cdots,\alpha_n) = (T\alpha_1,\cdots,T\alpha_n) = (\alpha_1,\cdots,\alpha_n)A$$

称矩阵 A 为线性变换 T 在这组基下的矩阵表示。

由空间结构和 T 的线性性质，T 由 $T\alpha_1,\cdots,T\alpha_n$ 完全确定，故由 T 唯一确定一个矩阵 A。

定理：设 V 的维数是 n，$\alpha_1,\cdots,\alpha_n$ 为 V 的一组基，任取 n 阶方阵 A，有且仅有一个从 V 到 V 的线性变换 T，使得 T 的矩阵恰好为 A。

推论：在 $L(V,V)$ 和全体 n 阶方阵之间存在一一对应关系。

例如：零变换对应零矩阵，恒等变换对应单位矩阵。

线性变换构成的空间

定理：$L(V)$ 也可以构成线性空间，引入 $L(V)$ 中的运算：对于 $L(V)$ 中任意的 T_1 与 T_2，V 中任意的 x，域 F 中任意的 k，有：

$$(T_1+T_2)x = T_1x + T_2x$$

$$(kT_1)x = k(T_1x)$$

容易验证 $L(V)$ 是 F 上的一个线性空间，即线性变换空间。

对于 $L(V)$ 中的线性变换 T_1 与 T_2，定义 T_1 与 T_2 的乘积 T_1T_2 为：

$$(T_1T_2)x = T_2(T_1x)$$

可以验证 (T_1T_2) 也是 $L(V)$ 中的线性变换，并且线性变换的乘积满足结合律，而不满足交换律，与矩阵的乘积类似。

对于 $L(V)$ 中的线性变换 T_1，如果 $L(V)$ 中的线性变换 T_2，使得对于 V 中任意的向量 x，有：

$$(T_1T_2)x = T_1(T_2x) = x$$

则称 T_2 是 T_1 的逆变换，记作：

$$T_2 = T_1^{-1}$$

且有：

$$T_1T_2 = T_2T_1 = T_e$$

定理：设 V 的维数为 n，$\alpha_1, \cdots, \alpha_n$ 为 V 的一组基，在这组基下线性变换 T_1 的矩阵为 A，T_2 的矩阵为 B，则：

- 线性变换 T_1+T_2 的矩阵为 $A+B$
- 线性变换的数乘 kT_1 的矩阵为 kA
- 线性变换的乘积 T_1T_2 的矩阵为 AB
- 线性变换 T_1 的逆变换若存在，矩阵为 A^{-1}

坐标

设 n 个向量 x 是 n 维空间 V 的一个基，对于 V 中任意的向量 y，令 y 为：

$$y = a_1x_1 + a_2x_2 + \cdots + a_nx_n = (x_1, x_2, \cdots, x_n)\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$$

称列向量：

$$\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$$

为向量 y 在基 $x_1, x_2, \cdots, x_n$ 下的**坐标**。

可见，坐标是由域中的标量构成的列向量，与阿贝尔群中的向量应当进行区分。

坐标变换公式

设 V 的维数为 n，$L(V)$ 中有变换 T，T 在基 $\alpha_1, \cdots, \alpha_n$ 下的矩阵为 A。设：

$$\xi = (\alpha_1, \cdots, \alpha_n) \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

且有：

$$T\xi = T(\alpha_1, \cdots, \alpha_n) \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

则有：

$$T\xi = T(\alpha_1, \cdots, \alpha_n) \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = (\alpha_1, \cdots, \alpha_n) A \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

空间 V 中的列向量点本质上都是「基乘坐标」的形式。空间 V 中的列向量点 x，本身用了单位阵 I 作为基，即 $x = Ix$。

只有同一个基，基不动的时候，单纯的线性变换 T，就是坐标左乘普通矩阵。

把线性变换 T 看成对于空间 V 的一个观测滤镜。线性变换 T 的作用对象是空间 V，将空间 V 扭曲了。加了滤镜之后，点本身的位置没有变。

这个定理也说明，对于列向量基的线性变换 T，等价于对于基右乘一个过渡矩阵。

于是，在不同的基之间，坐标关系是左乘过渡矩阵的逆矩阵。

过渡矩阵

设 n 个向量 x 与 n 个向量 y 是空间 V 的两组基。对于 $1 \le i \le n$，令每个向量 y_i 在基 $x_1, x_2, \cdots, x_n$ 下的坐标为：

$$y_i = (x_1, x_2, \cdots, x_n) \begin{pmatrix} a_{1i} \\ a_{2i} \\ \vdots \\ a_{ni} \end{pmatrix}$$

于是 n 个向量 y 排成等式左边的矩阵，n 个坐标排成等式右边的矩阵 A：

$$(y_1, y_2, \cdots, y_n) = (x_1, x_2, \cdots, x_n) A$$

矩阵 A 称为由基 $x_1, x_2 \cdots, x_n$ 到基 $y_1, y_2 \cdots, y_n$ 的**过渡矩阵**，也称为变换矩阵。

显然过渡矩阵可逆。对于上式，由基 $y_1, y_2 \cdots, y_n$ 到基 $x_1, x_2 \cdots, x_n$ 的过渡矩阵为 A^{-1}。

可见，过渡矩阵是由域中的标量构成的矩阵，并非阿贝尔群中的向量排成的矩阵，应当予以区分。

设 n 个向量 x 与 n 个向量 y 是空间 V 的两组基。对于空间 V 中的同一个向量 z，有：

$$z = (x_1, x_2, \cdots, x_n) \begin{pmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_n \end{pmatrix} = (y_1, y_2 \cdots, y_n) \begin{pmatrix} \eta_1 \\ \eta_2 \\ \vdots \\ \eta_n \end{pmatrix}$$

代入上文的

$$(y_1, y_2 \cdots, y_n) = (x_1, x_2 \cdots, x_n) A$$

由唯一性，得到：

$$\begin{pmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_n \end{pmatrix} = A \begin{pmatrix} \eta_1 \\ \eta_2 \\ \vdots \\ \eta_n \end{pmatrix}$$

或者

$$\begin{pmatrix} \eta_1 \\ \eta_2 \\ \vdots \\ \eta_n \end{pmatrix} = A^{-1} \begin{pmatrix} \xi_1 \\ \xi_2 \\ \vdots \\ \xi_n \end{pmatrix}$$

这是纯粹坐标之间的变换，坐标变换公式均在标量域中。由于前文做了区分，线性空间与阿贝尔群中的向量是「抽象的向量」，而坐标与过渡矩阵的元素均在标量域中，视为「具体的向量」，两种向量应当视为「不同的东西」。

矩阵可以对整个空间，即全体坐标进行变换，列向量 x 作为坐标遍布整个空间。

单位矩阵 I 由单位向量构成。矩阵 A 会将单位矩阵 I 变换到矩阵 A 的每个列向量，即将单位向量变换到矩阵 A 的每个列向量。因此左乘矩阵 A，也可以视为将空间做了这样的变换。

向量左乘矩阵，也可以视为坐标左乘向量组。用坐标的观点看待就是：

$$Iy = Xa$$

同一个列向量 y，在「正常」的空间，单位矩阵 I 代表的空间下，坐标为 y，在变换后新的空间里，坐标将记为 a。这样一来，矩阵 X 不仅是正常空间下的一组基，也是从向量组 I 到向量组 X 的过渡矩阵。

线性变换 T 会将一个基映射为另一个基，于是坐标也被映射为另一个坐标。

如果将基 α 映射到 β 对应的线性变换 T 的过渡矩阵是 A，那么对应的基矩阵就有 $\beta = \alpha A$。

于是坐标的关系恰好反过来。假设线性变换 T 映射后的坐标是 b，即加滤镜后观察到坐标 b，于是点在 V 的表示就是 βb。还原的办法就是用过渡矩阵，把点在 V 的表示写成 $\alpha A b$。于是坐标变换为左乘过渡矩阵的逆矩阵的看法就明显了。

线性变换与矩阵相似

在空间 V 中的一个线性变换 T 对于空间 V 的基 α 的关系：

线性变换 T 作用于基 α，将基 α 映射到了 $T(\alpha)$，相当于在基 α 右乘一个 A，即 $T(\alpha) = \alpha A$。

矩阵相似考虑的问题是：同一个线性变换 T，在基 β 的空间 V 中描述为矩阵 B，在基 α 的空间 V 中描述为矩阵 A。

如果过渡矩阵为 C，即 $\beta = \alpha C$，那么两个描述 B 和 A 之间有怎样的联系。

由于是同一个变换 T，可以发现一个事实，变换前后的过渡矩阵关系始终成立，即：

$$T(\beta) = T(\alpha)C = \alpha AC$$

线性变换 T 在基 β 视角下仍旧为右乘，基 β 转化到基 α 再右乘一个 C，变换前后保持过渡矩阵 C 的关系：

$$T(\beta) = \beta B = \alpha CB$$

于是问题得到解决：

$$B = C^{-1}AC$$

定理：设 $L(V)$ 中有变换 T，则 T 在不同基下的矩阵**相似**。

对于方阵 A 和方阵 B，如果存在可逆矩阵 C 使得 $B = C^{-1}AC$，则 A 和 B 相似。

矩阵相似保持秩不变，因此矩阵相似可以推出矩阵等价。但是，等价的两个矩阵未必相似。

由于矩阵相似与形状密切相关，因此矩阵相似和向量组等价、方程组同解之间没有关系。

回过头来，矩阵相似的解释就是 4 个等式：$\beta = \alpha C$、$T(\alpha) = \alpha A$、$T(\beta) = \beta B$、$T(\beta) = T(\alpha)C$。

参考资料

- 【 官方双语 / 合集 】 线性代数的本质 - 系列合集 P13 09 - 基变换[1]

参考资料与注释

[1] 【官方双语/合集】线性代数的本质 - 系列合集 P13 09 - 基变换

9.15.10 特征多项式

特征的这部分只研究方阵，即矩阵 A 对应的线性变换将 n 个向量映射到 n 个向量。

由于在实际问题中，经常要考虑连续进行重复的变换，如果只用「矩阵 A 对应的线性变换将单位阵 I 变换为 A」的描述，就会很抽象。此时最好的办法是找「不动点」，即变换当中不动的部分。

然而事实上，矩阵 A 对应的线性变换很可能没有不动点，于是退而求其次，寻找共线或者类似于简单变形的部分。

特征值与特征向量

在矩阵 A 对应的线性变换作用下，一些向量的方向不改变，只是伸缩了。

设 V 是 F 上的线性空间，T 是 V 上的线性变换。若存在 F 中的 λ 与 V 中的**非零向量** ξ，使得：

$$T\xi = \lambda\xi$$

则称 λ 为 T 的一个**特征值**，而 ξ 为 T 的**属于特征值 λ 的一个特征向量**。

特征向量在同一直线上，在线性变换作用下保持方向不改变（压缩到零也认为是方向不改变）。特征向量不唯一，与特征向量共线的向量都是特征向量，但是规定零向量不是特征向量，拥有方向的向量自然是非零向量。特征向量的特征值就是它伸缩的倍数。

在实际应用中，一般对于拥有相同特征值的特征向量，会选取一组基作为它们全体的代表。

设 $\alpha_1, \alpha_2, \cdots, \alpha_n$ 是 V 的一组基，T 在这组基下的矩阵为 A，即：

$$T(\alpha_1, \alpha_2, \cdots, \alpha_n) = (\alpha_1, \alpha_2, \cdots, \alpha_n)A$$

设 λ_0 是 T 的一个特征值，ξ 为 T 的属于特征值 λ_0 的一个特征向量，且有非零向量 X 满足：

$$\xi = (\alpha_1, \alpha_2, \cdots, \alpha_n)X$$

于是有：

$$T\xi = \lambda_0\xi$$

$$T(\alpha_1, \alpha_2, \cdots, \alpha_n)X = \lambda_0(\alpha_1, \alpha_2, \cdots, \alpha_n)X$$

$$(\alpha_1, \alpha_2, \cdots, \alpha_n)AX = \lambda_0(\alpha_1, \alpha_2, \cdots, \alpha_n)X$$

$$AX = \lambda_0 X$$

$$(A - \lambda_0 I)X = 0$$

所以相应的行列式也为 0。

特征多项式

考虑一个 $n\times n$ 的矩阵 A，其中 $n\geq 0\wedge n\in\mathbb{Z}$。设 λ 为一个参量，矩阵 $\lambda I-A$ 称为 A 的**特征矩阵**。

特征矩阵的行列式称为 A 的**特征多项式**，展开为一个 n 次多项式，根为 A 的特征值，记为 $p_A(\lambda)$：

$$p_A(\lambda)=\det(\lambda I_n-A)=\begin{vmatrix}\lambda-a_{11} & -a_{12} & \cdots & -a_{1n}\\ -a_{21} & \lambda-a_{22} & \cdots & -a_{2n}\\ \vdots & \vdots & & \vdots\\ -a_{n1} & -a_{n2} & \cdots & \lambda-a_{nn}\end{vmatrix}$$

其中 I_n 为一个 $n\times n$ 的单位矩阵。一些地方会定义为 $p_A(\lambda)=\det(A-\lambda I_n)$ 与我们的定义仅相差了一个符号 $(-1)^n$，但采用这种定义得到的 $p_A(\lambda)$ 一定为首一多项式，而另外的定义则仅当 n 为偶数时才是首一多项式。需要注意的是 0×0 的矩阵行列式为 1 是良定义的。

相应于 $(\lambda_0 I-A)X=0$ 的非零解向量 X，称为 A 的属于 λ_0 的特征向量。

线性变换 T 有特征值 λ_0 等价于矩阵 A 有特征值 λ_0。

线性变换 T 有特征向量 ξ 等价于矩阵 A 有特征向量 X，其中有：

$$\xi=(\alpha_1,\cdots,\alpha_n)X$$

根据代数基本定理，特征多项式可以分解为：

$$f(\lambda)=|\lambda I-A|=(\lambda-\lambda_1)^{d_1}\cdots(\lambda-\lambda_m)^{d_m}$$

称 d_i 为特征值 λ_i 的**代数重数**。全体代数重数的和为空间维数 n。

求解矩阵的全部特征值及特征向量

分为以下步骤：

- 计算行列式 $|\lambda I-A|$。
- 求出多项式 $f(\lambda)=|\lambda I-A|$ 在域 F 中的全部根，即 A 的特征值。
- 对 A 的每个特征值 λ，解齐次线性方程组 $(\lambda I-A)X=0$，求出它的一组基础解系 $X_1,\cdots,X_t$，则 A 的属于 λ 的全部特征向量为：

$$k_1X_1+k_2X_2+\cdots+k_tX_t$$

该表达式中的 k_i 不全为零。

- 线性变换 T 的属于 λ 的特征向量为：

$$\xi_i=(\alpha_1,\cdots,\alpha_n)X_i$$

因此，属于 λ 的全部特征向量为：

$$k_1\xi_1+k_2\xi_2+\cdots+k_t\xi_t$$

该表达式中的 k_i 不全为零。

特征值与特征向量是否存在，依赖于 V 所在的域。

相似变换

引入

若 $n\times n$ 的矩阵 A 为上三角矩阵如

$$A=\begin{bmatrix}a_{1,1} & a_{1,2} & \cdots & a_{1,n}\\ & a_{2,2} & \cdots & a_{2,n}\\ & & \ddots & \vdots\\ & & & a_{n,n}\end{bmatrix}$$

那么

$$\begin{aligned}
p_A(x) &= \det(xI_n - A) \\
&= \begin{bmatrix} x-a_{1,1} & -a_{1,2} & \cdots & -a_{1,n} \\ & x-a_{2,2} & \cdots & -a_{2,n} \\ & & \ddots & \vdots \\ & & & x-a_{n,n} \end{bmatrix} \\
&= \prod_{i=1}^{n}(x-a_{i,i})
\end{aligned}$$

可轻松求得，下三角矩阵也是类似的。但如果 A 不属于这两种矩阵，则需要使用相似变换，使得矩阵变为容易求得特征多项式的形式。

定义

对于 $n \times n$ 的矩阵 A 和 B，当存在 $n \times n$ 的可逆矩阵 P 满足

$$B = P^{-1}AP$$

则矩阵 A 和 B 相似，记变换 $A \mapsto P^{-1}AP$ 为相似变换。且 A 和 $P^{-1}AP$ 有相同的特征多项式。

考虑

$$\begin{aligned}
\det(xI_n - P^{-1}AP) &= \det(xP^{-1}I_nP - P^{-1}AP) \\
&= \det(P^{-1}xI_nP - P^{-1}AP) \\
&= \det(P^{-1}) \cdot \det(P) \cdot \det(xI_n - A) \\
&= \det(xI_n - A) \\
&= p_A(x)
\end{aligned}$$

得证，对于 $A \mapsto PAP^{-1}$ 也是一样的。另外 $p_A(0) = (-1)^n \cdot \det(A)$，因为 $p_A(0) = \det(-1 \cdot I_nA) = \det(-1 \cdot I_n) \cdot \det(A)$ 故 $\det(A) = \det(P^{-1}AP)$。

定理：相似矩阵有相同的特征多项式及特征值，反之不然。

定理表明，线性变换的矩阵的特征多项式与基的选取无关，而直接由线性变换决定，故可称之为线性变换的特征多项式。

矩阵 A 的特征多项式 $f(\lambda) = |\lambda I - A|$ 是一个首一的多项式。根据韦达定理，它的 $n-1$ 次系数为：

$$-(\lambda_1 + \cdots + \lambda_n) = -(a_{11} + \cdots + a_{nn}) = -trA$$

其中 trA 称为 A 的迹，为 A 的主对角线元素之和。

根据韦达定理，特征多项式的常数项为：

$$(-1)^n|A| = (-1)^n(\lambda_1 \cdots \lambda_n)$$

定理：相似的矩阵有相同的迹。

换位公式

定理：无论矩阵 A 和矩阵 B 是否方阵，只要乘法能进行，则矩阵 AB 的迹等于矩阵 BA 的迹。

一种证法是直接展开，即证毕。另一种证法用到换位公式。

定理：设 A 为 m 行 n 列矩阵，设 B 为 n 行 m 列矩阵，则有：

$$\lambda^n|\lambda I_m - AB| = \lambda^m|\lambda I_n - BA|$$

该公式表明 AB 与 BA 有相同的非零特征值。

舒尔（Schur）引理

任意的 n 阶矩阵 A 都相似于一个上三角阵，即存在满秩阵 P，使得 $P^{-1}AP$ 为上三角阵，它的主对角线上元素为 A 的全部特征值。

推论：设 A 的 n 个特征值为 $\lambda_1, \cdots, \lambda_n$，$\phi(x)$ 为任一多项式，则矩阵多项式 $\phi(A)$ 的 n 个特征值为：

$$\phi(\lambda_1), \cdots, \phi(\lambda_n)$$

特别地，kA 的特征值为 $k\lambda_1, \cdots, k\lambda_n$，$A^m$ 的特征值为 ${\lambda_1}^m, \cdots, {\lambda_n}^m$。

使用高斯消元进行相似变换

对 $n \times n$ 的矩阵 B 可以进行高斯消元，其基本操作为初等行变换。

在对矩阵使用上述操作（左乘初等矩阵）后再右乘其逆矩阵即相似变换，左乘为行变换，易发现右乘即列变换。

若能将矩阵通过相似变换变为上三角或下三角的形式，那么可以轻松求出其特征多项式。但若对主对角线上的元素应用变换 $A \mapsto T_{ij}(k)AT_{ij}(-k)$ 后会导致原本通过 $A \mapsto T_{ij}(k)A$ 将第 i 行第 j 列的元素消为零后右乘 $T_{ij}(-k)$ 即将 A 的第 i 列的 $-k$ 倍加到第 j 列这一操作使得之前消为零的元素现在可能不为零，可能不能将其变为上三角或下三角形式。

后文将说明对次对角线上的元素应用变换后得到的矩阵依然可以轻松得到其特征多项式。

上 Hessenberg 矩阵

对于 $n > 2$ 的形如

$$H = \begin{bmatrix} \alpha_1 & h_{12} & \cdots & \cdots & h_{1n} \\ \beta_2 & \alpha_2 & h_{23} & \cdots & \vdots \\ & \ddots & \ddots & \ddots & \vdots \\ & & \ddots & \ddots & h_{(n-1)n} \\ & & & \beta_n & \alpha_n \end{bmatrix}$$

的矩阵我们称为上 Hessenberg 矩阵，其中 β 为次对角线。

我们使用相似变换将次对角线以下的元素消为零后即能得到上 Hessenberg 矩阵，而求出一个 $n \times n$ 上 Hessenberg 矩阵的特征多项式则可在 $O(n^3)$ 时间完成。

我们记 H_i 为只保留 H 的前 i 行和前 i 列的矩阵，记 $p_i(x) = \det(xI_i - H_i)$ 那么

$$H_0 \text{ 丰,[} \quad p_0(x) = 1$$

$$H_1 = \begin{bmatrix} \alpha_1 \end{bmatrix}, \quad p_1(x) = \det(xI_1 - H_1) = x - \alpha_1$$

$$H_2 = \begin{bmatrix} \alpha_1 & h_{12} \\ \beta_2 & \alpha_2 \end{bmatrix}, \quad p_2(x) = \det(xI_2 - H_2) = (x - \alpha_2)p_1(x) - \beta_2 h_{12} p_0(x)$$

在计算行列式时我们一般选择按零最多的行或列余子式展开，余子式即删除了当前选择的元素所在行和列之后的矩阵，在这里我们选择按最后一行进行展开，有

$$\begin{aligned} p_3(x) &= \det(xI_3 - H_3) \\ &= \begin{vmatrix} x - \alpha_1 & -h_{12} & -h_{13} \\ -\beta_2 & x - \alpha_2 & -h_{23} \\ & -\beta_3 & x - \alpha_3 \end{vmatrix} \\ &= (x - \alpha_3) \cdot (-1)^{3+3} p_2(x) - \beta_3 \cdot (-1)^{3+2} \begin{vmatrix} x - \alpha_1 & -h_{13} \\ -\beta_2 & -h_{23} \end{vmatrix} \\ &= (x - \alpha_3)p_2(x) - \beta_3(h_{23}p_1(x) + \beta_2 h_{13} p_0(x)) \end{aligned}$$

观察并归纳，对 $2 \le i \le n$ 有

$$p_i(x) = (x - \alpha_i)p_{i-1}(x) - \sum_{m=1}^{i-1} h_{i-m,i} \left(\prod_{j=i-m+1}^{i} \beta_j \right) p_{i-m-1}(x)$$

至此完成了整个算法，该算法一般被称为 Hessenberg 算法。

Cayley–Hamilton 定理

对于任意的 n 阶矩阵 A，特征多项式为 $f(\lambda) = |\lambda I - A|$，则必有 $f(A) = 0$。

对于线性变换 T 有平行的结果：如果 $f(\lambda)$ 为 T 的特征多项式，则 $f(T)$ 为零变换。

由本定理可知，对于任意的矩阵 A，必有可以使其零化的多项式。

最小多项式

设 V 是一个 n 维向量空间，由于线性变换对应的矩阵有 n^2 个元素，一切线性变换构成 n^2 维线性空间。

对于一个特定的线性变换 T，从作用 0 次到作用 n 次，总共 $n^2 + 1$ 个线性变换，它们对应的矩阵一定线性相关。于是存在非零多项式 f，使得 $f(T)$ 为零变换，称变换 T 满足多项式 f。在 T 满足的所有多项式 f 中，存在次数最低的。

可以将矩阵 A 零化的最小次数的首一多项式称为 A 的最小多项式，记为 $m_A(\lambda)$。

根据多项式的辗转相除法，最小多项式是唯一的，且可整除任一 A 的零化多项式。特别地，最小多项式整除特征多项式。

定理：在不计重数的情况下，矩阵 A 的特征多项式 $f(\lambda)$ 与最小多项式 $m_A(\lambda)$ 有相同的根。

定理：矩阵 A 的属于不同特征值的特征向量线性无关。

应用

在信息学中我们一般考虑 $(\mathbb{Z}/m\mathbb{Z})^{n\times n}$ 上的矩阵，通常 m 为素数，进行上述相似变换是简单的，当 m 为合数时，我们可以考虑类似辗转相除的方法来进行。

" 实现"

```cpp
#include <cassert>
#include <iostream>
#include <random>
#include <vector>

typedef std::vector<std::vector<int>> Matrix;
typedef long long i64;

Matrix to_upper_Hessenberg(const Matrix &M, int mod) {
  Matrix H(M);
  int n = H.size();
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      if ((H[i][j] %= mod) < 0) H[i][j] += mod;
    }
  }
  for (int i = 0; i < n - 1; ++i) {
    int pivot = i + 1;
    for (; pivot < n; ++pivot) {
      if (H[pivot][i] != 0) break;
```

```
    }
    if (pivot == n) continue;
    if (pivot != i + 1) {
      for (int j = i; j < n; ++j) std::swap(H[i + 1][j], H[pivot][j]);
      for (int j = 0; j < n; ++j) std::swap(H[j][i + 1], H[j][pivot]);
    }
    for (int j = i + 2; j < n; ++j) {
      for (;;) {
        if (H[j][i] == 0) break;
        if (H[i + 1][i] == 0) {
          for (int k = i; k < n; ++k) std::swap(H[i + 1][k], H[j][k]);
          for (int k = 0; k < n; ++k) std::swap(H[k][i + 1], H[k][j]);
          break;
        }
        if (H[j][i] >= H[i + 1][i]) {
          int q = H[j][i] / H[i + 1][i], mq = mod - q;
          for (int k = i; k < n; ++k)
            H[j][k] = (H[j][k] + i64(mq) * H[i + 1][k]) % mod;
          for (int k = 0; k < n; ++k)
            H[k][i + 1] = (H[k][i + 1] + i64(q) * H[k][j]) % mod;
        } else {
          int q = H[i + 1][i] / H[j][i], mq = mod - q;
          for (int k = i; k < n; ++k)
            H[i + 1][k] = (H[i + 1][k] + i64(mq) * H[j][k]) % mod;
          for (int k = 0; k < n; ++k)
            H[k][j] = (H[k][j] + i64(q) * H[k][i + 1]) % mod;
        }
      }
    }
  }
  return H;
}

std::vector<int> get_charpoly(const Matrix &M, int mod) {
  Matrix H(to_upper_Hessenberg(M, mod));
  int n = H.size();
  std::vector<std::vector<int>> p(n + 1);
  p[0] = {1 % mod};
  for (int i = 1; i <= n; ++i) {
    const std::vector<int> &pi_1 = p[i - 1];
    std::vector<int> &pi = p[i];
    pi.resize(i + 1, 0);
    int v = mod - H[i - 1][i - 1];
    if (v == mod) v -= mod;
    for (int j = 0; j < i; ++j) {
      pi[j] = (pi[j] + i64(v) * pi_1[j]) % mod;
      if ((pi[j + 1] += pi_1[j]) >= mod) pi[j + 1] -= mod;
    }
    int t = 1;
    for (int j = 1; j < i; ++j) {
      t = i64(t) * H[i - j][i - j - 1] % mod;
      int prod = i64(t) * H[i - j - 1][i - 1] % mod;
      if (prod == 0) continue;
      prod = mod - prod;
```

```
      for (int k = 0; k <= i - j - 1; ++k)
        pi[k] = (pi[k] + i64(prod) * p[i - j - 1][k]) % mod;
    }
  }
  return p[n];
}

bool verify(const Matrix &M, const std::vector<int> &charpoly, int mod) {
  if (mod == 1) return true;
  int n = M.size();
  std::vector<int> randvec(n), sum(n, 0);
  std::mt19937 gen(std::random_device{}());
  std::uniform_int_distribution<int> dis(1, mod - 1);
  for (int i = 0; i < n; ++i) randvec[i] = dis(gen);
  for (int i = 0; i <= n; ++i) {
    int v = charpoly[i];
    for (int j = 0; j < n; ++j) sum[j] = (sum[j] + i64(v) * randvec[j]) % mod;
    std::vector<int> prod(n, 0);
    for (int j = 0; j < n; ++j) {
      for (int k = 0; k < n; ++k) {
        prod[j] = (prod[j] + i64(M[j][k]) * randvec[k]) % mod;
      }
    }
    randvec.swap(prod);
  }
  for (int i = 0; i < n; ++i)
    if (sum[i] != 0) return false;
  return true;
}

int main() {
  std::ios::sync_with_stdio(false);
  std::cin.tie(0);
  int n, mod;
  std::cin >> n >> mod;
  Matrix M(n, std::vector<int>(n));
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j) std::cin >> M[i][j];
  std::vector<int> charpoly(get_charpoly(M, mod));
  for (int i = 0; i <= n; ++i) std::cout << charpoly[i] << ' ';
  assert(verify(M, charpoly, mod));
  return 0;
}
```

上述 Hessenberg 算法不具有数值的稳定性，所以 $\mathbb{R}^{n\times n}$ 上的矩阵在使用前需要其他算法进行调整或改用其他具有数值稳定性的算法。

我们可以将特征多项式与常系数齐次线性递推联系起来，也可结合 Cayley–Hamilton 定理、多项式取模加速一些域上求矩阵幂次的算法。

Cayley–Hamilton 定理指出

$$\begin{aligned}p_A(A) &= A^n + c_1A^{n-1} + \cdots + c_{n-1}A + c_nI\\ &= O\end{aligned}$$

其中 O 为 $n\times n$ 的零矩阵，$A\in\mathbb{C}^{n\times n}$ 且 $p_A(x)=x^n+\sum_{i=1}^{n}c_ix^{n-i}\in\mathbb{C}[x]$ 为 A 的特征多项式。

若我们要求 A^K 其中 K 较大，那么可以求出 $f(x)=x^K \bmod p_A(x)$ 后利用 $f(A)=A^K$。

而 $\deg(f(x)) < n$ 显然。我们令 $f(x) = \sum_{i=0}^{n-1} f_i x^i$ 且 $n = km$ 那么

$$\begin{aligned} f_{km-1}x^{km-1} + \cdots + f_1 x + f_0 = (\cdots(&f_{km-1}x^{k-1} + \cdots + f_{k(m-1)})x^k \\ &+ f_{k(m-1)-1}x^{k-1} + \cdots + f_{k(m-2)})x^k \\ &+ \cdots \\ &+ f_{k-1}x^{k-1} + \cdots + f_1 x + f_0 \end{aligned}$$

令 $k = \sqrt{n}$ 可以发现计算 $f(A)$ 大约需要 $O(\sqrt{n})$ 次矩阵与矩阵的乘法。

参考文献

- Rizwana Rehman, Ilse C.F. Ipsen.La Budde’s Method for Computing Characteristic Polynomials[1].
- Marshall Law.Computing Characteristic Polynomials of Matrices of Structured Polynomials[2].
- Mike Paterson.On the Number of Nonscalar Multiplications Necessary to Evaluate Polynomials[3].

参考资料与注释

[1] La Budde’s Method for Computing Characteristic Polynomials

[2] Computing Characteristic Polynomials of Matrices of Structured Polynomials

[3] On the Number of Nonscalar Multiplications Necessary to Evaluate Polynomials

9.15.11 对角化

特征子空间

矩阵 A 的属于 λ_0 的全部特征向量，再添上零向量，构成一个线性空间，称为矩阵 A 的一个特征子空间，记为 $E(\lambda_0)$。它是齐次线性方程组：

$$(\lambda_0 I - A)X = 0$$

的解空间。

对于特征子空间 $E(\lambda_i) = N(\lambda_i I - A)$，由亏加秩定理有：

$$r(\lambda_i I - A) + \dim N(\lambda_i I - A) = n$$

因此，特征子空间 $E(\lambda_i)$ 的维数为：

$$\dim E(\lambda_i) = n - r(\lambda_i I - A)$$

也称为 λ_i 的**几何重数**。

不变子空间

在研究线性变换 T 的时候，常常希望选取空间 V 的一个基，使得线性变换 T 对于这个基的矩阵具有尽可能简单的形状。

设 V 是数域 F 上的线性空间，W 是 V 的一个子空间，T 是 V 上的一个线性变换。如果对于 W 中任意的向量 x，都有 $T(x)$ 也在 W 中（也称为空间在变换下不变或稳定），称 W 是 T 的一个不变子空间。

空间在变换下不变，并不是说坐标在变换下真的「不变」，有可能是进行了一个拉伸等变形，只是变形后还落在空间里。

- 线性空间 V 的任意一个子空间都是数乘变换的不变子空间。

- 对于 V 中任意的线性变换 T，空间 V 和零子空间都是 T 的不变子空间，称为平凡不变子空间。
- 不变子空间的交与和也是不变子空间。

设 W 是线性变换 T 的一个不变子空间。只考虑 T 在不变子空间 W 上的作用，就得到子空间 W 本身的线性变换，称为 T 在子空间 W 上的限制，记作 $T|_W$。

对于 V 中任意的线性变换 T，像空间 $R(T)$ 与核空间 $N(T)$ 是 T 的不变子空间。这两种情况的含义是，空间 V 在变换前后，完成了自身的压缩（像空间），或者压缩到 0（核空间）。

对于 V 中任意的线性变换 T，T 的特征子空间是 T 的不变子空间。

准素分解

根据代数基本定理，最小多项式可以分解为：

$$m_A(\lambda) = (\lambda - \lambda_1)^{r_1} \cdots (\lambda - \lambda_S)^{r_S}$$

考虑最小多项式代入变元 λ 为矩阵 A 后，各个因式的核空间，构成矩阵 A 的一系列不变子空间：

$$W_i = N((\lambda_i I - A)^{r_i})$$

定理：该不变子空间 W_i 的维数，恰好为特征值 λ_i 的代数重数。

回顾一下，代数重数是指特征多项式各个因式的次数，几何重数是指特征子空间 $E(\lambda_i) = N(\lambda_i I - A)$ 的维数。这个不变子空间 W_i 与特征子空间 $E(\lambda_i)$，两者都是矩阵的核空间，并且两个矩阵构成最小多项式 r_i 次幂的关系。也就是说，特征子空间的维数是几何重数，「特征子空间」经过最小多项式 r_i 次幂后到达一个「不变子空间」，不变子空间的维数到达了特征多项式的代数重数。

该定理其实是下面准素分解定理的推论。

记矩阵 A 对应的线性变换 T，在每个子空间 W_i 上的限制 $T_i = T|_{W_i}$。于是 T_i 的最小多项式是 $(x - \lambda_i)^{r_i}$。

定理：设 V 是域 F 上的线性空间，T 是 V 上的一个线性变换。那么空间 V 可以关于线性变换 T 进行准素分解，拆成若干不变子空间 W_i 的直和。

$$V = W_1 \oplus W_2 \oplus \cdots \oplus W_S$$

这意味着，T 在某组基下的矩阵是准对角阵：

$$\operatorname{diag}\{A_1, A_2, \cdots, A_S\}$$

其中，A_i 是 T_i 在对应基下的矩阵。

该定理表明，可以使用不变子空间简化线性变换的矩阵。

可对角化矩阵

对于 n 阶方阵 A，如果相似于一个对角阵，则称 A 为可对角化矩阵，或称单纯矩阵。

- 对角阵的和、积、逆，如果存在，仍然是对角阵，其对角线上的元素就是它的特征值。
- 线性变换 T 的矩阵为可对角化矩阵，等价于 T 在某组基下的矩阵为对角阵。

定理：设矩阵 A 的全部互异特征根为 $\lambda_1, \cdots, \lambda_m$，则以下命题等价：

- 矩阵 A 可对角化。
- 矩阵 A 有 n 个线性无关的特征向量。
- 以下公式成立：

$$\dim E(\lambda_1) + \cdots + \dim E(\lambda_m) = n$$

前文已经指出，特征多项式的分解式中特征值的次数称为代数重数，特征子空间的维数称为几何重数。这个定理也表明，矩阵 A 可对角化，等价于 A 的每个特征值 λ 的代数重数都等于它的几何重数。

推论：如果 n 阶方阵 A 恰有 n 个互异特征值，则它必可对角化。反之则不一定。

定理：矩阵 A 可对角化当且仅当 A 的最小多项式没有重根。

矩阵的相似也会保持特征向量之间的线性相关关系不变。

特征向量完全可能不是实数，也完全可能找不到 n 个线性无关的特征向量。

对于重特征值而言，特征向量张成空间。为了描述这个空间，需要从其中选择代表。一般会选择线性无关的代表，代表的个数就是空间的维数。

选取代表时，常常将它们正交化与单位化。最终得到的就是一套单位正交的代表。

特征向量不一定正交，不同特征值的特征向量，可能无法正交。因此正交化只能对于重特征值的特征向量进行。但是单位化可以对任意特征向量进行。

幂零矩阵

设 T 是空间 V 的一个线性变换。如果存在一个正整数 r，使得 T^r 为零变换，称 T 是空间 V 的一个幂零变换。

对于某一个正整数 r，满足条件 $N^r = 0$ 的矩阵称为幂零矩阵。

一般可以进一步假定 r 是使 T^r 为零变换的最小正整数，于是 T 的最小多项式是 x^r。于是存在一个向量 ξ_0，使得：

- $$T^r(\xi_0) = 0$$
- $$T^{r-1}(\xi_0) \neq 0$$

循环子空间

定理：设 T 是空间 V 的一个线性变换，ξ 是空间 V 的一个向量。如果存在一个正整数 s，使得：

- $$T^s(\xi) = 0$$
- $$T^{s-1}(\xi) \neq 0$$

那么向量 $\xi, T(\xi), \cdots, T^{s-1}(\xi)$ 线性无关。

由这个定理可以给出一个定义：

设 T 是空间 V 的一个线性变换，W 是 V 的一个子空间。如果存在一个向量 ξ_0 和一个正整数 r，使得：

- 向量 $\xi_0, T(\xi_0), \cdots, T^{r-1}(\xi_0)$ 构成 W 的一个基。
- 如下等式成立：
  $$T^r(\xi_0) = 0$$

那么子空间 W 称为关于 T 的一个循环子空间，简称 T 循环子空间。此时 ξ_0 称为循环子空间 W 的一个生成向量，向量 $\xi_0, T(\xi_0), \cdots, T^{r-1}(\xi_0)$ 称为 W 的一个循环基。

显然，一个 T 循环子空间 W 在 T 作用下不变，并且对于循环子空间 W 中的任意向量 ξ，均有 $T^r(\xi) = 0$，这里 r 为循环子空间的维数。

幂零 Jordan 块

如果空间 W 是变换 T 的循环子空间，那么 T 在 W 上的限制 $T|_W$ 是 W 的一个幂零变换，并且 $T|_W$ 关于 W 的倒序排列的循环基 $T^{r-1}(\xi_0), T^{r-2}(\xi_0), \cdots, \xi_0$ 的矩阵是如下形状的 r 阶上三角矩阵：

$$N_r = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & 1 \\ 0 & 0 & 0 & \cdots & 0 & 0 \end{pmatrix}$$

矩阵 N_r 称为一个 r 阶幂零 Jordan 矩阵，或者 r 阶幂零 Jordan 块。

设 T 是 n 维空间 V 的一个幂零变换，把出现在 V 关于 T 的循环子空间的分解中，唯一确定的一组正整数 $r_1 \geq \cdots \geq r_S$ 叫做 T 的不变指数。

对于 n 阶幂零矩阵 A，A 与一个上述形状的矩阵 N 相似，也唯一确定一个正整数序列 $r_1 \geq \cdots \geq r_S$，称为矩阵 A 的不变指数。

幂零阵虽然不能和对角阵相似，但是可以相似于这样的标准形式。在 Jordan 标准型，将相似对角化与幂零阵的标准形式，二者结合起来，给出一般的矩阵通过相似变换可以达到的标准形式。

一些定理

1. 设 T 是空间 V 的一个幂零变换，而
$$h(x) = a_0 + a_1 x + \cdots + a_m x^m$$
是一个多项式，那么当且仅当 $a_0 \neq 0$ 时，线性变换 $h(T)$ 有逆变换。当 $h(T)$ 可逆时，$h(T)$ 的逆变换也是 T 的一个多项式。
2. 设 T 是空间 V 的一个幂零变换，W 是一个 r 维 T 循环子空间，ξ 是 W 中的向量。如果存在一个整数 k，使得
$$T^{r-k}(\xi) = 0$$
那么存在 W 中的向量 η，使得
$$\xi = T^k(\eta)$$
3. 设 T 是 n 维空间 V 的一个幂零变换，x^r 是 T 的最小多项式，令 W_1 是一个 r 维 T 循环子空间，那么存在 W_1 的一个余子空间 W_2，使得：
$$V = W_1 \oplus W_2$$
并且 W_2 也在 T 作用下不变。
4. 设 T 是 n 维空间 V 的一个幂零变换，那么 V 可以分解为 T 循环子空间的直和：
$$V = W_1 \oplus W_2 \oplus \cdots \oplus W_S$$
5. 每一个 n 阶幂零矩阵都与一个形如：
$$N = \begin{pmatrix} N_{r_1} & & & 0 \\ & N_{r_2} & & \\ & & \cdots & \\ 0 & & & N_{r_S} \end{pmatrix}$$
的矩阵相似，这里的每一个 N_{r_i} 是一个 r_i 阶幂零 Jordan 块。
6. 如果规定 T 循环子空间 W_i 按照维数 r_i 降序排列 $r_1 \geq \cdots \geq r_S$，那么将 V 分解为 T 循环子空间的方法是由 T 唯一确定的。

9.15.12 Jordan 标准型

Jordan 分解

设 T 是 n 维空间 V 上的一个线性变换。如果 T 的最小多项式为：

$$m_A(\lambda) = (\lambda - \lambda_1)^{r_1}(\lambda - \lambda_2)^{r_2}\cdots(\lambda - \lambda_k)^{r_k}$$

那么由准素分解可知，空间 V 可以分解为子空间的直和：

$$V = V_1 \oplus V_2 \oplus \cdots \oplus V_k$$

其中 $V_i = N\left((A-\lambda_i I)^{r_i}\right)$，式中 A 为 T 对应的矩阵，这些子空间都在 T 作用下不变。

令变换 T_i 为 V 在子空间 V_i 上的射影，即构造多项式 $u_i(T)$ 使得：

- $$T_i = u_i(T)\frac{m_A(T)}{(T-\lambda_i T_e)^{r_i}}$$
- $$T_1 + T_2 + \cdots + T_k = T_e$$

式中 T_e 表示空间 V 的恒等变换。于是有性质：

- 变换 T_i 在空间 V_i 上的限制 $T_i|_{V_i}$ 为空间 V_i 的恒等变换。
- 如果 i 与 j 不相等，变换 T_i 在空间 V_j 上的限制 $T_i|_{V_j}$ 为空间 V_j 的零变换。

于是变换 T_i 将空间 V 的每一个向量 ξ 映射为它在空间 V_i 中的分量 ξ_i。

构造变换：

$$T_D = \lambda_1 T_1 + \lambda_2 T_2 + \cdots + \lambda_k T_k$$

由于每一个变换 T_i 都是变换 T 的一个多项式，所以变换 T_D 也是变换 T 的一个多项式，于是每一个子空间 V_i 在变换 T_D 下不变。

由上述等式可知，变换 T_D 在子空间 V_i 上的限制 $T_D|_{V_i}$ 是子空间 V_i 的一个位似，位似系数为 λ_i。因此，变换 T_D 可以对角化。

构造：

$$T_N = T - T_D$$

于是变换 T_N 也是变换 T 的一个多项式，所以每一个子空间 V_i 在变换 T_N 下不变。对于子空间 V_i 中的任意向量 ξ_i，有：

$$T_N{}^{r_i}(\xi_i) = T - T_D{}^{r_i}(\xi_i) = T - \lambda_i T_i{}^{r_i}(\xi_i) = 0$$

令 r 为全体 r_i 的最大值，那么对于空间 V 中的任意向量 ξ，变换 T_N 的 r 次方将向量 ξ 映射至零向量。因此变换 T_N 是一个幂零变换。

这样，空间 V 的每一个变换 T 都可以写成：

$$T = T_D + T_N$$

其中 T_D 可以对角化，而 T_N 是一个幂零变换。因为 T_D 和 T_N 都是变换 T 的多项式，所以它们的乘积可交换：

$$T_D T_N = T_N T_D$$

定理：设 T_1 和 T_2 是空间 V 的两个可对角化变换，且 $T_1T_2 = T_2T_1$，那么存在一个基，使得 T_1 和 T_2 关于这同一个基的矩阵是对角形式。

定理：设 T 是 n 维空间 V 上的一个线性变换，那么存在一个可对角化变换 T_D 和一个幂零变换 T_N，使得：

- $$T = T_D + T_N$$

- $$T_D T_N = T_N T_D$$

它们都是变换 T 的多项式，并且它们由变换 T 唯一确定。

该定理给出关于变换 T 的分解，称为 T 的若尔当（Jordan）分解，T_D 叫做 T 的可对角化部分，T_N 叫做 T 的幂零部分。

同样地，有矩阵的 Jordan 分解：

定理：设 A 是一个 n 阶矩阵，那么存在一个可对角化矩阵 D 和一个幂零矩阵 N，使得：

- $$A = D + N$$

- $$DN = ND$$

它们都是矩阵 A 的多项式，并且它们由矩阵 A 唯一确定。

该定理给出关于矩阵 A 的分解，称为 A 的若尔当（Jordan）分解，D 叫做 A 的可对角化部分，N 叫做 A 的幂零部分。

lambda 矩阵

接下来引入的部分是含有变元参量 λ 的更广义的矩阵，不仅仅是一个数表。这部分讨论相较单纯由数构成的矩阵而言，更加广泛一些。

对于 λ 矩阵，对应空间相应的域，变为含有一个变元 λ 的有理式域。

以 λ 的多项式为元素的矩阵称为 λ 矩阵，记为 $A(\lambda)$。

由于多项式域包含数域，数字矩阵是特殊的 λ 矩阵，数字矩阵 A 的特征矩阵 $\lambda I - A$ 是一种 λ 矩阵。

lambda 矩阵的初等变换

对于 λ 矩阵，同样可以定义加减法、乘法、初等变换、秩。对于 λ 方阵，同样可以定义行列式、余子式、代数余子式。

对于 λ 矩阵，初等变换与数阵大多相同，仅将倍加变换改为（这里以行变换为例）：

- 用 λ 的多项式 $\varphi(\lambda)$ 乘某行并加到另一行上。

注意倍乘变换不进行修改。这是因为倍加变换不改变行列式，而倍乘变换改变行列式。为了保持多项式域的秩的性质，行列式只能在数域上进行改变。

相应的初等矩阵也一并进行修改。

易见三种初等阵的行列式均为非零常数，因此均为满秩。所以它们左乘或右乘，不改变 λ 矩阵的秩。

若 $A(\lambda)$ 经过有限次初等变换变为 $B(\lambda)$，则称 $A(\lambda)$ 和 $B(\lambda)$ 等价。

对于 λ 矩阵，如果等价，则秩相同。反之则不然，这与数字矩阵有区别。

Smith 标准型

定理：设 λ 矩阵的秩是 r，则 $A(\lambda)$ 一定等价于：

$$\begin{pmatrix} D(\lambda) & 0 \\ 0 & 0 \end{pmatrix}$$

其中：

$$D(\lambda) = \begin{pmatrix} d_1(\lambda) & & \\ & \ddots & \\ & & d_r(\lambda) \end{pmatrix}$$

每一个 $d_i(\lambda)$ 是一个首 1 多项式，并且相邻两个多项式有整除关系 $d_i(\lambda)|d_{i+1}(\lambda)$。

称此标准型为 Smith 标准型，称 $d_i(\lambda)$ 为不变因子。

具体求解 Smith 标准型的办法是，从左上角到右下角进行消元，每次左上角的元素是右下方剩余的全体多项式的最大公因式，并借助左上角的元素将该行该列全部消为 0。

定理：条件 $A(\lambda)$ 和 $B(\lambda)$ 等价，等价于条件 $A(\lambda)$ 和 $B(\lambda)$ 拥有完全一样的不变因子。

初等因子

由代数基本定理，设 $A(\lambda)$ 的不变因子 $d_1(\lambda), d_2(\lambda), \cdots, d_m(\lambda)$ 的分解为：

$$d_i(\lambda) = (\lambda - \lambda_1)^{e_{i1}} (\lambda - \lambda_2)^{e_{i2}} \cdots (\lambda - \lambda_S)^{e_{iS}}$$

其中 $\lambda_1, \cdots, \lambda_S$ 互不相同。由于：

$$d_i(\lambda)|d_{i+1}(\lambda)$$

因此指数 $e_{1j}, e_{2j}, \cdots, e_{mj}$ 递增，并且最后一项 $d_m(\lambda)$ 的各项指数均非零。

上式中指数大于零的全部因子，统称为 $A(\lambda)$ 的初等因子。

注意，初等因子计重数。如果对于某个 j，指数 e_{ij} 出现了若干次，则对应的初等因子 $(\lambda - \lambda_j)^{e_{ij}}$ 也应当出现相应次数。

之前的定理说明，$A(\lambda)$ 与 $B(\lambda)$ 等价，等价于他们两个拥有完全一致的不变因子。不变因子完全相同，自然初等因子也完全相同，但是反之则不然。事实上有结论：

定理：$A(\lambda)$ 与 $B(\lambda)$ 不变因子完全相同，等价于初等因子和秩均完全相同。

于是「初等因子和秩均完全相同」也成为判断 λ 矩阵等价性的条件。

在初等变换的时候，也可以先将 $A(\lambda)$ 变换为对角阵，再求出初等因子和秩，再求出不变因子得到标准型。有结论：

定理：设 $A(\lambda)$ 等价于对角阵：

$$\operatorname{diag}\{f_1(\lambda), f_2(\lambda), \cdots, f_r(\lambda), 0, \cdots, 0\}$$

那么有 $f_1(\lambda), f_2(\lambda), \cdots, f_r(\lambda)$ 的全体一次因子的幂 $(\lambda - \lambda_j)^{e_{ij}}$，构成 $A(\lambda)$ 的初等因子。

由初等因子和秩构造不变因子的具体方法为：先将初等因子按照因式分类，排成表格，把同类因式进行降幂排列放到同一行，各类因式的最高次幂放到一列，把列数用 1 补齐至秩 r，那么每一列的乘积构成一个不变因子。

在特征矩阵中的应用

如果 A 与 B 是数阵，那么它们的特征矩阵是 λ 矩阵。有结论：

定理：条件数阵 A 与 B 相似，等价于条件特征矩阵 $\lambda I - A$ 和 $\lambda I - B$ 等价。

由于特征矩阵 $\lambda I - A$ 只在主对角线含有 n 个 λ，所以秩为 n。由上述推理，同型的数阵的特征矩阵的秩始终相等，于是有等价性：

数阵 A 与 B 相似，等价于特征矩阵 $\lambda I - A$ 和 $\lambda I - B$ 有完全相同的初等因子。

对于特征矩阵 $\lambda I - A$，初等变换保持等价性，所以不改变秩。

观察三种初等变换，由于唯一被改写的倍加变换不改变行列式，事实上三种初等变换仅对行列式的结果多项式改变常数倍，因此不改变行列式的结果多项式的因式分解与次数。

因此特征矩阵 $\lambda I - A$ 的行列式为 n 次多项式，初等变换化为 Smith 标准型后，由于秩为 n，行列式就是主对角线全体不变因子的乘积，也等于全体初等因子的乘积。因此，特征矩阵 $\lambda I - A$ 的全体初等因子的次数之和等于 n。

Jordan 标准型

矩阵

$$
\begin{pmatrix}
\lambda & 1 & 0 & \cdots & 0 & 0 \\
0 & \lambda & 1 & \cdots & 0 & 0 \\
0 & 0 & \lambda & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & & \vdots & \vdots \\
0 & 0 & 0 & \cdots & \lambda & 1 \\
0 & 0 & 0 & \cdots & 0 & \lambda
\end{pmatrix}
$$

主对角线上的元素都是 λ，紧邻主对角线上方的元素都是 1，其余位置都是 0，叫做属于 λ 的一个 Jordan 矩阵，或称 Jordan 块。

显然，幂零 Jordan 矩阵是 Jordan 矩阵的特例，即 λ 为 0 的情形。

定理：设 T 是 n 维空间 V 的一个变换，$\lambda_1, \cdots, \lambda_k$ 是 T 的一切互不相同的特征值，那么存在一个基，使得 T 关于这个基的矩阵有形状：

$$
\begin{pmatrix}
B_1 & & & 0 \\
& B_2 & & \\
& & \ddots & \\
0 & & & B_k
\end{pmatrix}
$$

其中

$$
B_i = \begin{pmatrix}
J_{i1} & & & 0 \\
& J_{i2} & & \\
& & \ddots & \\
0 & & & J_{is_i}
\end{pmatrix}
$$

其中 $J_{i1}, \cdots, J_{is_i}$ 都是属于 λ_i 的 Jordan 块。

这是因为，首先根据最小多项式：

$$
m_A(\lambda) = (\lambda - \lambda_1)^{r_1} (\lambda - \lambda_2)^{r_2} \cdots (\lambda - \lambda_k)^{r_k}
$$

有准素分解：

$$
V = V_1 \oplus V_2 \oplus \cdots \oplus V_k
$$

其中：

$$
V_i = N\left((A - \lambda_i I)^{r_i}\right)
$$

式中 A 为 T 对应的矩阵。

令变换 S_i 为 T 在 V_i 上的限制 $T|_{V_i}$，接下来试图对每一个 S_i 进行 Jordan 分解。

记 T_e 为 V 上的恒等变换。与前文的 Jordan 分解不同，记 T_i 为 S_i 的 Jordan 分解中的幂零部分：

$$
S_i = \lambda_i T_e + T_i
$$

于是 T_i 为子空间 V_i 的一个幂零变换，事实上也是 $T - \lambda_i T_e$ 在 V_i 上的限制 $(T - \lambda_i T_e)|_{V_i}$。

子空间 V_i 可以分解为幂零变换 T_i 循环子空间的直和：

$$
V_i = W_{i1} \oplus W_{i2} \oplus \cdots \oplus W_{is_i}
$$

在每一个循环子空间 W_{ij} 里，取一个循环基并倒序排列，凑成 V_i 的一个基，于是 T_i 关于这个基的矩阵有形状：

$$
N_i = \begin{pmatrix}
N_{i1} & & & 0 \\
& N_{i2} & & \\
& & \ddots & \\
0 & & & N_{is_i}
\end{pmatrix}
$$

全体 N_{ij} 均为幂零 Jordan 块。于是对于 V_i 上述选取的基，S_i 对应的矩阵是：

$$B_i=\begin{pmatrix}\lambda_i & & & 0\\ & \lambda_i & & \\ & & \ddots & \\ 0 & & & \lambda_i\end{pmatrix}+\begin{pmatrix}N_{i1} & & & 0\\ & N_{i2} & & \\ & & \ddots & \\ 0 & & & N_{is_i}\end{pmatrix}=\begin{pmatrix}J_{i1} & & & 0\\ & J_{i2} & & \\ & & \ddots & \\ 0 & & & J_{is_i}\end{pmatrix}$$

这里 $J_{i1},J_{i2},\cdots,J_{is_i}$ 都是属于 λ_i 的 Jordan 块。

对于每一个子空间 V_i，按照以上方式选取一个基，凑起来成为 V 的基，那么 T 关于这个基的矩阵即构成定理规定的形式。

形如：

$$\begin{pmatrix}J_1 & & & 0\\ & J_2 & & \\ & & \ddots & \\ 0 & & & J_m\end{pmatrix}$$

的 n 阶矩阵，其中每一个 J_i 都是一个 Jordan 块，叫做一个 Jordan 标准型。

定理：每一个 n 阶矩阵 A 都与一个 Jordan 标准型相似。除了各个 Jordan 块排列的次序以外，与 A 相似的 Jordan 标准型是由 A 唯一确定的。

注意在上述构造的矩阵 B_i 中，第一项是一个单位阵的若干倍，自然可以和第二项交换。因此，第一项就是 B_i 的 Jordan 分解的可对角化部分，第二项就是 B_i 的 Jordan 分解的幂零部分。

在一个矩阵对应的 Jordan 标准型里面，主对角线上的元素构成的对角阵是这个矩阵对应的 Jordan 标准型的可对角化部分，把主对角线上的元素换成 0 就得到这个矩阵对应的 Jordan 标准型的幂零部分。

定理：对于矩阵 A 的 Jordan 标准型中，每一个 Jordan 块：

$$J_i=\begin{pmatrix}\lambda_i & 1 & & & \\ & \lambda_i & 1 & & \\ & & \ddots & \ddots & \\ & & & \ddots & 1\\ & & & & \lambda_i\end{pmatrix}$$

对应于特征矩阵 $\lambda I-A$ 的一个初等因子 $(\lambda-\lambda_i)^{n_i}$，特征矩阵 $\lambda I-A$ 的全体初等因子对应于矩阵 A 的 Jordan 标准型中的全体 Jordan 块。

这是因为，矩阵 A 相似于它的 Jordan 标准型，因此两者的特征矩阵也等价，将 Jordan 标准型的特征矩阵化为 Smith 标准型即可看出。

由这个定理，借助特征矩阵 $\lambda I-A$ 的初等因子，可以写出矩阵 A 的 Jordan 标准型。

一个推论是，矩阵 A 可对角化，等价于特征矩阵 $\lambda I-A$ 的初等因子均为一次的。

弗罗贝尼乌斯（Forbenious）定理

上文指出，n 阶特征矩阵的 Smith 标准形的秩为 n。

定理：设矩阵 A 的特征矩阵 $\lambda I-A$ 的 Smith 标准形为：

$$\mathrm{diag}\{d_1(\lambda),d_2(\lambda),\cdots,d_n(\lambda)\}$$

则最后一个不变因子 $d_n(\lambda)$ 恰好为矩阵 A 的最小多项式 $m_A(\lambda)$。

推论：矩阵 A 可对角化的等价条件为：

- 最小多项式 $m_A(\lambda)$ 无重根。
- 特征矩阵 $\lambda I-A$ 的不变因子无重根。
- 特征矩阵 $\lambda I-A$ 的初等因子均为一次的。

9.16 线性规划

9.16.1 线性规划简介

Authors: Ir1d, YZircon, huhaoo, QAQAutoMaton, Enter-tainer, Marcythm, sshwy, partychicken, Konano, H-J-Granger, baker221, isdanni, ksyx

定义

研究线性约束条件下线性目标函数极值问题的方法总称，是运筹学的一个分支，在多方面均有应用。线性规划的某些特殊情况，如网络流、多商品流量等问题都有可能在 OI 题目中出现

线性规划问题的描述

一个问题要能转化为线性规划问题，首先要有若干个线性约束条件，并且所求的目标函数也应该是线性的。那么，最容易也最常用的描述方法就是标准型。

我们以《算法导论》中线性规划一节提出的问题为例：

> 假如你是一位政治家，试图赢得一场选举，你的选区有三种：市区，郊区和乡村，这些选区分别有 100000、200000 和 50000 个选民，尽管并不是所有人都有足够的社会责任感去投票，你还是希望每个选区至少有半数选民投票以确保你可以当选
>
> 显而易见的，你是一个正直、可敬的人，然而你意识到，在某些选区，某些议题可以更有效的赢取选票。你的首要议题是修筑更多的道路、枪支管制、农场补贴以及调整汽油税。你的竞选班子可以为你估测每花费 $1000 做广告，在每个选区可以赢取或者失去的选票的数量（千人），如下表所示：

政策	市区	郊区	乡村
修路	-2	5	3
枪支管制	8	2	-5
农场补贴	0	0	10
汽油税	10	0	-2

你的目标是计算出要在市区，郊区和乡村分别获得至少 50000，100000 和 25000 张选票所花费的最少钱数。

我们可以使用数学语言来描述它：

x_1 为花费在修路广告上的钱（千美元）

设 x_2 为花费在枪支管制广告上的钱（千美元）

设 x_3 为花费在农场补贴广告上的钱（千美元）

设 x_4 为花费在汽油税广告上的钱（千美元）

那么我们可以将「在市区获得至少 50000 张市区选票」表述为

$-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50$

同样的，「在郊区获得至少 100000 张选票和在乡村获得至少 25000 张选票」可以表示为

$5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100$

$3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25$

显而易见的，广告服务提供商不会倒贴钱给你然后做反向广告，由此可得

$x_1, x_2, x_3, x_4 \geq 0$

又因为我们的目标是使总费用最小，综上所述，原问题可以表述为：

最小化 $x_1+x_2+x_3+x_4$，

满足

$-2x_1+8x_2+0x_3+10x_4\geq 50$

$5x_1+2x_2+0x_3+0x_4\geq 100$

$3x_1-5x_2+10x_3-2x_4\geq 25$

$x_1,x_2,x_3,x_4\geq 0$

这个线性规划的解就是这个问题的最优策略

用更具有普遍性的语言说：

已知一组实数 $[a_1..a_n]$ 和一组变量 $[x_1..x_n]$, 在定义上有函数 $f(x_1..x_n)=\sum_{i=1}^{n}a_ix_i$

这个函数是线性的。如果 b 是一个实数而满足 $f(x_1..x_n)=b$, 则这个等式被称为线性等式，同样的，满足 $f(x_1..x_n)\leq b$ 或者 $f(x_1..x_n)\geq b$ 则称之为线性不等式

在线性规划问题中，线性等式和线性不等式统称为线性约束。

一个线性规划问题是一个线性函数的极值问题，而这个线性函数应该服从于一个或者多个线性约束。

图解法

当线性规划问题的变量较少时，我们可以使用图解法来直观地解决问题。对于一般的情况，通常使用 单纯形法。

上一节问题的变量较多，不便于使用图解法，所以用下面的问题来介绍图解法：

最小化 $x+y$

满足

$3x+y\geq 9$

$x\leq 4$

$y\geq 3$

$x,y\in N$

知道这些约束条件以后，我们需要将它们在平面直角坐标系中画出来

$x\leq 4$（红色区域）

$y\geq 3$（黑色区域）

$3x+y\geq 9$（蓝色直线右侧区域）

黄色区域为三块区域的交集，这就是这个线性规划的所有可行解。因为题目中说明，需要最小化 $x+y$，观察图像可知，在点 $(2,3)$,x 和 y 同时取到最小值，此时 $x+y$ 当然最小。因此，$\min(x+y)=(2+3)=5$。

把求解线性规划的图解法总结起来，就是先在坐标系中作出所有的约束条件，然后作出需要求极值的线性函数（目标函数）的图像。目标函数中所有点的集合与满足约束条件的所有点的集合的交集就是这个线性规划的解集，而所需求的极值由观察可以轻易得出。

9.16.2 单纯形算法

定义

单纯形法是解决线性规划问题的一个有效的算法。

线性规划就是在一组线性约束条件下，求解目标函数最优解的问题。

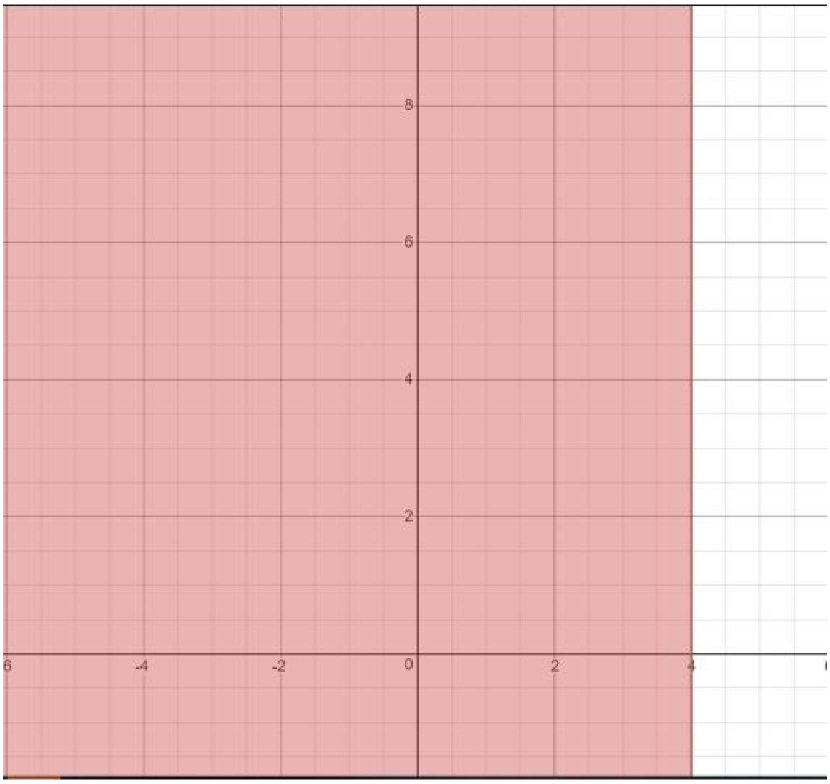


图 9.31 img

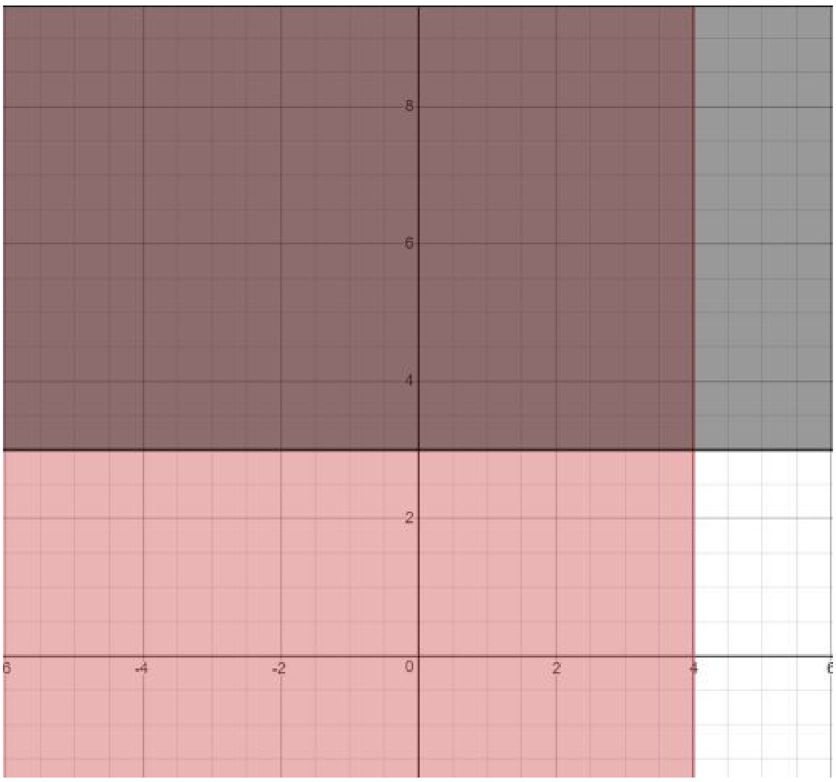


图 9.32 img

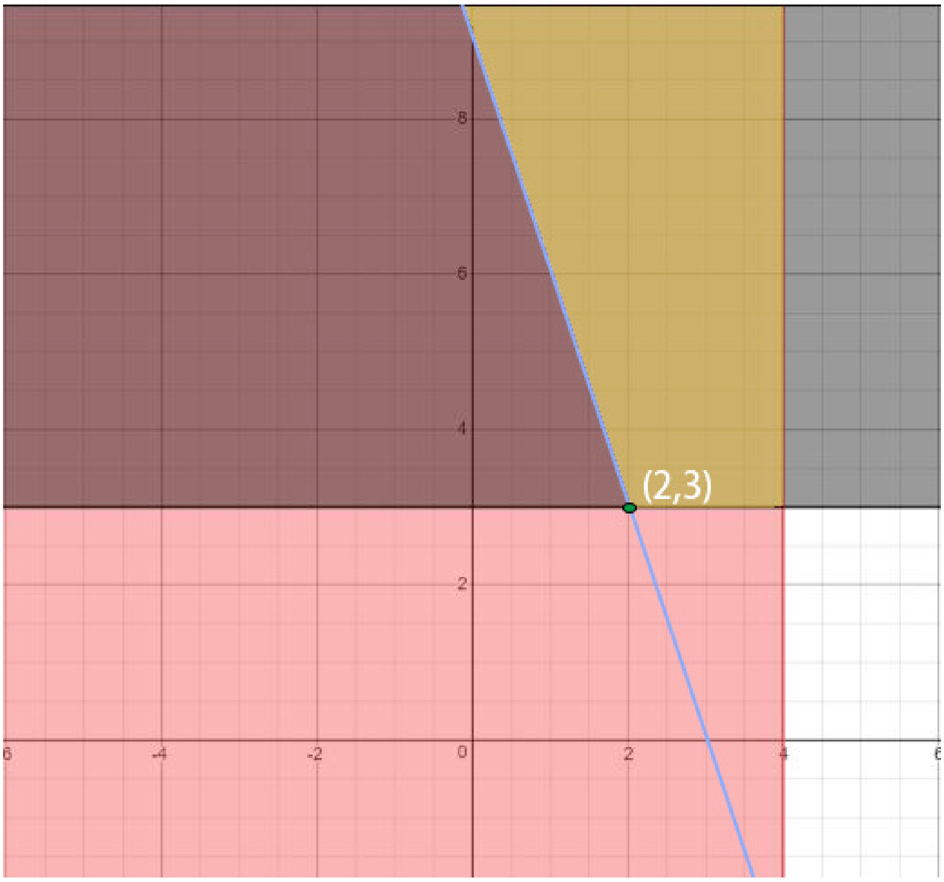


图 9.33 img

线性规划的一般形式

在约束条件下，寻找目标函数 z 的最大值：

$$\max\ z = x_1 + x_2$$

$$s.t\begin{cases}2x_1 + x_2 \le 12\\ x_1 + 2x_2 \le 9\\ x_1, x_2 \ge 0\end{cases}$$

线性规划的可行域

满足线性规划问题约束条件的所有点组成的集合就是线性规划的可行域。若可行域有界（以下主要考虑有界可行域），线性规划问题的目标函数最优解必然在可行域的顶点上达到最优。

单纯形法就是通过设置不同的基向量，经过矩阵的线性变换，求得基可行解（可行域顶点），并判断该解是否最优，否则继续设置另一组基向量，重复执行以上步骤，直到找到最优解。所以，单纯形法的求解过程是一个循环迭代的过程。

线性规划的标准形式

在说明单纯形法的原理之前，需要明白线性规划的标准形式。因为单纯形算法是通过线性规划的标准形来求解的。一般，规定线性规划的标准形式为：

$$\max\ z = \sum_{j=1}^{n} c_j x_j$$

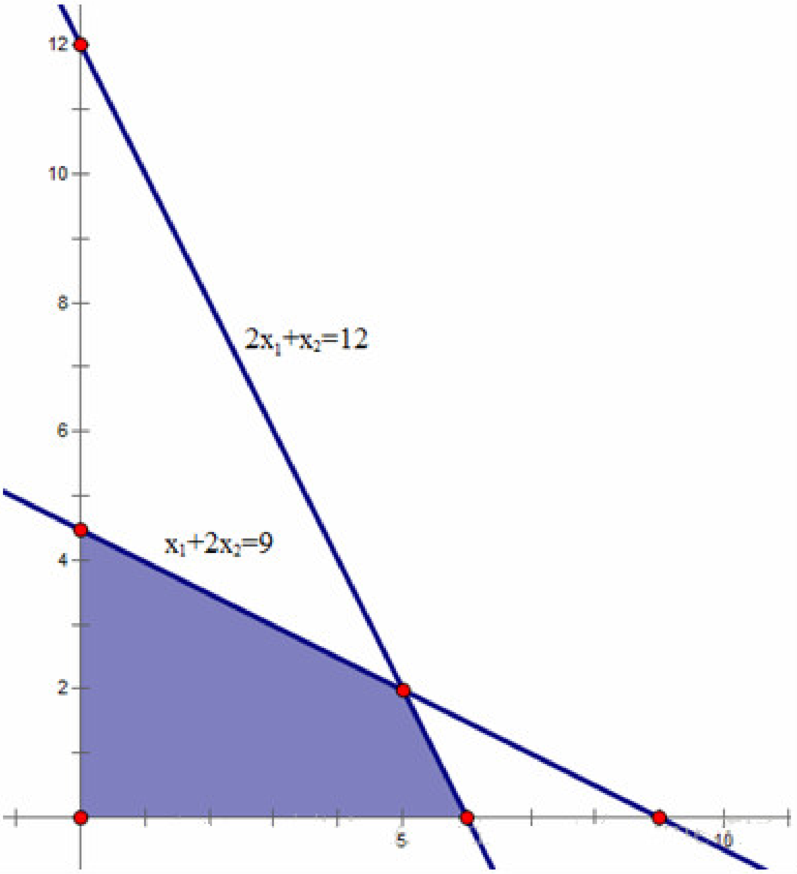


图 9.34 kexingyu

$$s.t\begin{cases}\sum_{j=1}^{n}a_{ij}x_j=b_i, i=1,2,\dots,m\\ x_j\geq 0, j=1,2,\dots,n\end{cases}$$

写成矩阵形式:

$$\max\ \ z=CX$$

$$AX=b$$

$$X\geq 0$$

$$A=\begin{bmatrix}a_{11} & a_{12} & \dots & a_{1n}\\ a_{21} & a_{22} & \dots & a_{2n}\\ \vdots & \vdots & \ddots & \vdots\\ a_{m1} & a_{m2} & \dots & a_{mn}\end{bmatrix}$$

标准形的形式为:

1. 目标函数要求 max
2. 约束条件均为等式
3. 决策变量为非负约束

普通线性规划化为标准形:

1. 若目标函数为最小化，可以通过取负，求最大化
2. 约束不等式为小于等于不等式，可以在左端加入非负变量，转变为等式，比如：

$$x_1 + 2x_2 \leq 9 \implies \begin{cases} x_1 + 2x_2 + x_3 = 9 \\ x_3 \geq 0 \end{cases}$$

   同理，约束不等式为大于等于不等式时，可以在左端减去一个非负松弛变量，变为等式。
3. 若存在取值无约束的变量，可转变为两个非负变量的差，比如：

$$-\infty \leq x_k \leq +\infty \implies \begin{cases} x_k = x_m - x_n \\ x_m, x_n \geq 0 \end{cases}$$

本文最开始的线性规划问题转化为标准形为：

$$\max\ z = x_1 + x_2$$

$$s.t \begin{cases} 2x_1 + x_2 + x_3 = 12 \\ x_1 + 2x_2 + x_4 = 9 \\ x_1, x_2, x_3, x_4 \geq 0 \end{cases}$$

单纯形法

几何意义

在标准形中，有 m 个约束条件（不包括非负约束），n 个决策变量，且 $n \geq m$。首先选取 m 个基变量 $x_j^{'}(j = 1, 2, ..., m)$，基变量对应约束系数矩阵的列向量线性无关。通过矩阵的线性变换，基变量可由非基变量表示：

$$x_i^{'} = C_i + \sum_{j=m+1}^{n} m_{ij} x_j^{'}(i = 1, 2, ..., m)$$

如果令非基变量等于 0，可求得基变量的值：

$$x_i^{'} = C_i$$

如果为可行解的话，C_i 大于 0。那么它的几何意义是什么呢？

还是通过上述具体的线性规划问题来说明：

$$\max\ z = x_1 + x_2$$

$$s.t \begin{cases} 2x_1 + x_2 + x_3 = 12 \\ x_1 + 2x_2 + x_4 = 9 \\ x_1, x_2, x_3, x_4 \geq 0 \end{cases}$$

如果选择 x_2、x_3 为基变量，那么令 x_1、x_4 等于 0，可以去求解基变量 x_2、x_3 的值。对系数矩阵做行变换，如下所示，$x_2 = \dfrac{9}{2}$，$x_3 = \dfrac{15}{2}$。

$$\begin{bmatrix} \mathrm{X} & x_1 & x_2 & x_3 & x_4 & b \\ & 2 & 1 & 1 & 0 & 12 \\ & 1 & 2 & 0 & 1 & 9 \\ \mathrm{C} & 1 & 1 & 0 & 0 & z \end{bmatrix} \rightarrow \begin{bmatrix} \mathrm{X} & x_1 & x_2 & x_3 & x_4 & b \\ & \frac{3}{2} & 0 & 1 & -\frac{1}{2} & \frac{15}{2} \\ & \frac{1}{2} & 1 & 0 & \frac{1}{2} & \frac{9}{2} \\ \mathrm{C} & \frac{1}{2} & 0 & 0 & -\frac{1}{2} & z - \frac{9}{2} \end{bmatrix}$$

$X_1 = 0$ 表示可行解在 y 轴上；$X_4 = 0$ 表示可行解在 $x_1 + 2x_2 = 9$ 的直线上。那么，求得的可行解即表示这两条直线的交点，也是可行域的顶点，如图所示：

所以，通过选择不同的基变量，可以获得不同的可行域的顶点。

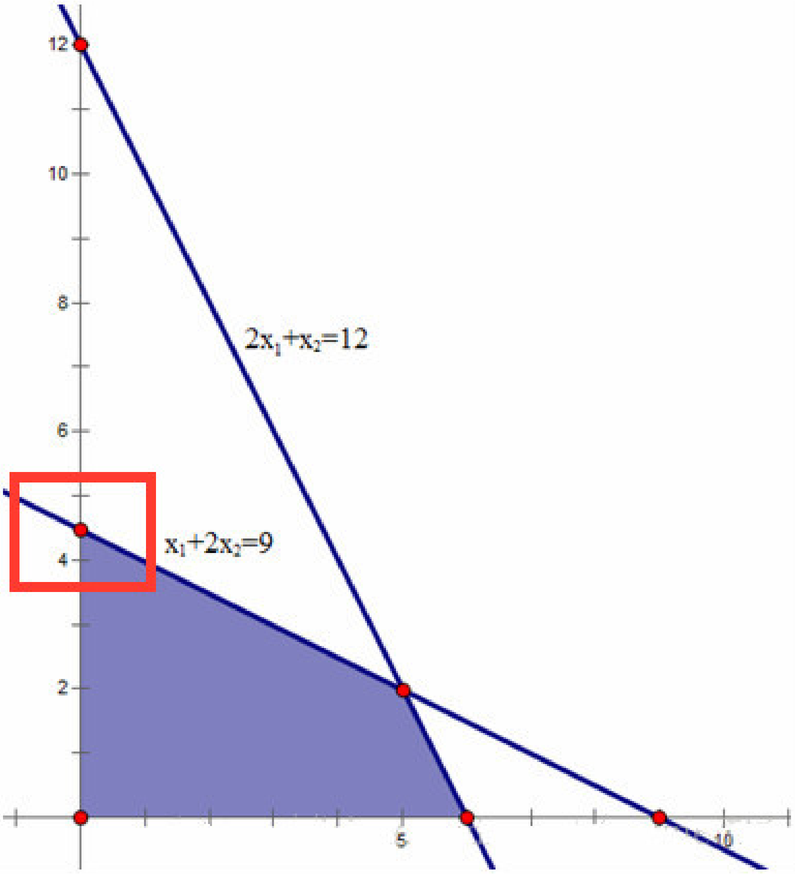


图 9.35 kexingyu_point

如何判断最优

如前所述，基变量可由非基变量表示：

$$x_i^{'} = C_i + \sum_{j=m+1}^{n} m_{ij} x_j^{'} (i = 1, 2, \ldots, m)$$

目标函数 z 也可以完全由非基变量表示：

$$z = z_0 + \sum_{j=m+1}^{n} \sigma_j x_j^{'}$$

当达到最优解时，所有的 σ_j 应小于等于 0，当存在 j，$\sigma_j > 0$ 时，当前解不是最优解，为什么?

当前的目标函数值为 z_0，其中所有的非基变量值均取 0。由之前分析可知，$x_j^{'} = 0$ 代表可行域的某个边界，是 $x_j^{'}$ 的最小值。如果可行解逐步离开这个边界，$x_j^{'}$ 会变大，因为 $\sigma_j > 0$，显然目标函数的取值也会变大，所以当前解不是最优解。我们需要寻找新的基变量。

如何选择新的基变量

如果存在多个 $\sigma_j > 0$，选择最大的 $\sigma_j > 0$ 对应的变量作为基变量，这表示目标函数随着 $x_j^{'}$ 的增加，增长的最快。

如何选择被替换的基变量

假如我们选择非基变量 $x_s^{'}$ 作为下一轮的基变量，那么被替换基变量 $x_j^{'}$ 在下一轮中作为非基变量，等于 0。选择 $x_j^{'}$ 的原则：替换后应该尽量使 $x_s^{'}$ 值最大（因为上面已分析过，目标函数会随着 $x_s^{'}$ 的增大而增大），但要保证替换基变量后的解仍是可行解，因此应该选择最紧的限制。

继续通过上面的例子来说明：

$$\begin{bmatrix} \mathrm{X} & x_1 & x_2 & x_3 & x_4 & b \\ & 2 & 1 & 1 & 0 & 12 \\ & 1 & 2 & 0 & 1 & 9 \\ \mathrm{C} & 1 & 1 & 0 & 0 & z \end{bmatrix} \rightarrow \begin{bmatrix} \mathrm{X} & x_1 & x_2 & x_3 & x_4 & b \\ & \frac{3}{2} & 0 & 1 & -\frac{1}{2} & \frac{15}{2} \\ & \frac{1}{2} & 1 & 0 & \frac{1}{2} & \frac{9}{2} \\ \mathrm{C} & \frac{1}{2} & 0 & 0 & -\frac{1}{2} & z-\frac{9}{2} \end{bmatrix}$$

从最后一行可以看到，x_1 的系数为 $\dfrac{1}{2} > 0$，所以选 x_2、x_3 为基变量并没有是目标函数达到最优。下一轮选取 x_1 作为基变量，替换 x_2、x_3 中的某个变量。

第一行是符号

第二行：若 x_1 替换 x_3 作为基变量，$x_3 = 0$ 时，$x_1 = \dfrac{15/2}{3/2} = 5$

第三行：若 x_1 替换 x_2 作为基变量，$x_2 = 0$ 时，$x_1 = \dfrac{9/2}{1/2} = 9$

尽管替换 x_2 后，x_1 的值更大，但将它代入 x_3 后会发现 x_3 的值为负，不满足约束。从几何的角度来看，选择 x_2 和 x_4 作为非基变量，得到的解是直线 $x_2 = 0$ 和 $x_1 + 2x_2 = 9$ 的交点，它在可行域外。因此应该选择 x_3 作为非基变量。

终止条件

当目标函数用非基变量的线性组合表示时，所有的系数均不大于 0，则表示目标函数达到最优。

如果，有一个非基变量的系数为 0，其他的均小于 0，表示目标函数的最优解有无穷多个。这是因为目标函数的梯度与某一边界正交，在这个边界上，目标函数的取值均相等，且为最优。

使用单纯形法来求解线性规划，输入单纯形法的松弛形式，是一个大矩阵，第一行为目标函数的系数，且最后一个数字为当前轴值下的 z 值。下面每一行代表一个约束，数字代表系数每行最后一个数字代表 b 值。

算法和使用单纯性表求解线性规划相同。

对于线性规划问题：

$$\max\ x_1 + 14x_2 + 6x_3$$

$$s.t \begin{cases} x_1 + x_2 + x_3 \le 4 \\ x_1 \le 2 \\ x_3 \le 3 \\ 3x_2 + x_3 \le 6 \\ x_1, x_2, x_3 \ge 0 \end{cases}$$

我们可以得到其松弛形式：

$$\max\ x_1 + 14x_2 + 6x_3$$

$$s.t \begin{cases} x_1 + x_2 + x_3 + x_4 = 4 \\ x_1 + x_5 = 2 \\ x_3 + x_6 = 3 \\ 3x_2 + x_3 + x_7 = 6 \\ x_1, x_2, x_3, x_4, x_5, x_6, x_7 \ge 0 \end{cases}$$

我们可以构造单纯形表，其中最后一行打星的列为轴值。

x_1	x_2	x_3	x_4	x_5	x_6	x_7	b
$c_1 = 1$	$c_2 = 14$	$c_3 = 6$	$c_4 = 0$	$c_5 = 0$	$c_6 = 0$	$c_7 = 0$	$-z = 0$
1	1	1	1	0	0	0	4
1	0	0	0	1	0	0	2
0	0	1	0	0	1	0	3
0	3	1	0	0	0	1	6
			*	*	*	*	

在单纯形表中，我们发现非轴值的 x 上的系数大于零，因此可以通过增加这些个 x 的值，来使目标函数增加。我们可以贪心的选择最大的 c，在上面的例子中我们选择 c_2 作为新的轴，加入轴集合中，那么谁该出轴呢?

其实我们由于每个 x 都大于零，对于 x_2 它的增加是有所限制的，如果 x_2 过大，由于其他的限制条件，就会使得其他的 x 小于零，于是我们应该让 x_2 一直增大，直到有一个其他的 x 刚好等于 0 为止，那么这个 x 就被换出轴。

我们可以发现，对于约束方程 1，即第一行约束，x_2 最大可以为 4 $(\frac{4}{1})$，对于约束方程 4，x_2 最大可以为 2 $(\frac{6}{3})$，因此 x_2 最大只能为他们之间最小的那个，这样才能保证每个 x 都大于零。因此使用第 4 行，来对各行进行高斯行变换，使得第二列第四行中的每个 x 都变成零，也包括 c_2。这样我们就完成了把 x_2 入轴，x_7 出轴的过程。变换后的单纯形表为:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	b
$c_1 = 1$	$c_2 = 0$	$c_3 = 1.33$	$c_4 = 0$	$c_5 = 0$	$c_6 = 0$	$c_7 = -4.67$	$-z = -28$
1	0	0.67	1	0	0	−0.33	2
1	0	0	0	1	0	0	2
0	0	1	0	0	1	0	3
0	1	0.33	0	0	0	0.33	2
	*		*	*	*		

继续计算，我们得到:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	b
$c_1 = -1$	$c_2 = 0$	$c_3 = 0$	$c_4 = -2$	$c_5 = 0$	$c_6 = 0$	$c_7 = -4$	$-z = -32$
1.5	0	1	1.5	0	0	−0.5	3
1	0	0	0	1	0	0	2
−1.5	0	0	−1.5	0	1	0.5	0
−0.5	1	0	−0.5	0	0	0.5	1
	*	*		*	*		

此时我们发现，所有非轴的 x 的系数全部小于零，即增大任何非轴的 x 值并不能使得目标函数最大，从而得到最优解 32。

整个过程代码如下所示:

```
#include <bits/stdc++.h>
using namespace std;
vector<vector<double> > Matrix;
double Z;
set<int> P;
size_t cn, bn;

bool Pivot(pair<size_t, size_t> &p) {  // 返回 0 表示所有的非轴元素都小于 0
  int x = 0, y = 0;
  double cmax = -INT_MAX;
  vector<double> C = Matrix[0];
  vector<double> B;

  for (size_t i = 0; i < bn; i++) {
    B.push_back(Matrix[i][cn - 1]);
  }

  for (size_t i = 0; i < C.size(); i++) {  // 在非轴元素中找最大的 c
    if (cmax < C[i] && P.find(i) == P.end()) {
      cmax = C[i];
      y = i;
    }
  }
  if (cmax < 0) {
    return 0;
  }

  double bmin = INT_MAX;
  for (size_t i = 1; i < bn; i++) {
    double tmp = B[i] / Matrix[i][y];
    if (Matrix[i][y] != 0 && bmin > tmp && tmp > 0) {
      bmin = tmp;
      x = i;
    }
  }

  p = make_pair(x, y);

  for (set<int>::iterator it = P.begin(); it != P.end(); it++) {
    if (Matrix[x][*it] != 0) {
      // cout<<"erase "<<*it<<endl;
      P.erase(*it);
      break;
    }
  }
  P.insert(y);
  // cout<<"add "<<y<<endl;
  return true;
}

void pnt() {
  for (size_t i = 0; i < Matrix.size(); i++) {
    for (size_t j = 0; j < Matrix[0].size(); j++) {
      cout << Matrix[i][j] << "\t";
```

```
    }
    cout << endl;
  }
  cout << "result z:" << -Matrix[0][cn - 1] << endl;
}

void Gaussian(pair<size_t, size_t> p) {  // 行变换
  size_t x = p.first;
  size_t y = p.second;
  double norm = Matrix[x][y];
  for (size_t i = 0; i < cn; i++) {  // 主行归一化
    Matrix[x][i] /= norm;
  }
  for (size_t i = 0; i < bn; i++) {
    if (i != x && Matrix[i][y] != 0) {
      double tmpnorm = Matrix[i][y];
      for (size_t j = 0; j < cn; j++) {
        Matrix[i][j] = Matrix[i][j] - tmpnorm * Matrix[x][j];
      }
    }
  }
}

void solve() {
  pair<size_t, size_t> t;
  while (1) {
    pnt();
    if (Pivot(t) == 0) {
      return;
    }
    cout << t.first << " " << t.second << endl;
    for (set<int>::iterator it = P.begin(); it != P.end(); it++) {
      cout << *it << " ";
    }
    cout << endl;
    Gaussian(t);
  }
}

int main(int argc, char *argv[]) {
  // ifstream fin;
  // fin.open("./test");
  cin >> cn >> bn;
  for (size_t i = 0; i < bn; i++) {
    vector<double> vectmp;
    for (size_t j = 0; j < cn; j++) {
      double tmp = 0;
      cin >> tmp;
      vectmp.push_back(tmp);
    }
    Matrix.push_back(vectmp);
  }

  for (size_t i = 0; i < bn - 1; i++) {
```

```
    P.insert(cn - i - 2);
  }
  solve();
}

////////////////////////////////////
// glpk input:
///* Variables */
// var x1 >= 0;
// var x2 >= 0;
// var x3 >= 0;
///* Object function */
// maximize z: x1 + 14*x2 + 6*x3;
///* Constrains */
// s.t. con1: x1 + x2 + x3 <= 4;
// s.t. con2: x1  <= 2;
// s.t. con3: x3  <= 3;
// s.t. con4: 3*x2 + x3  <= 6;
// end;
////////////////////////////////////
// myinput:
/*
8 5
1 14 6 0 0 0 0 0
1 1 1 1 0 0 0 4
1 0 0 0 1 0 0 2
0 0 1 0 0 1 0 3
0 3 1 0 0 0 1 6
*/
////////////////////////////////////
```

理论罗列

标准型

$m+n$ 个约束 n 个变量用 x 向量表示，A 是一个 $m\times n$ 的矩阵，c 是一个 n 的向量，b 是一个 m 的向量，最大化 cx 满足约束 $Ax\le b, x>0$。

最大化 $\sum_{j=1}^{n} c_j x_j$ 满足如下约束条件：

$$\sum_{j=1}^{n} a_{ij}x_j \le b_i, i=1,2,\dots,m$$

$$x_j \ge 0, j=1,2,\dots,n$$

n 个变量，$m+n$ 个约束，构造 $m\times n$ 的矩阵 A，m 维向量 b，n 维向量 c

最大化 $C^T x$ 满足如下约束条件：

$$Ax\le b$$

$$x\ge 0$$

转换为标准型

若目标函数要求取最小值，那么可以对其取相反数变成取最大值。对于限制条件 $f(x_1,x_2,\dots,x_n)=b$，可以用两个不等式 $f(x_1,x_2,\dots,x_n)\le b, -f(x_1,x_2,\dots,x_n)\le -b$ 描述，对于限制条件 $f(x_1,x_2,\dots,x_n)\ge b$，可以用不等式 $-f(x_1,x_2,\dots,x_n)\le -b$ 描述。对于无限制的变量 x，可以将其拆为两个非负变量 x_0, x_1，使得 $x=x_0-x_1$。

松弛型

基本变量 B，$|B| = m$，一个约束对应一个，表示松弛量，叫做松弛变量（基本变量）

非基变量 N，$|N| = n$，$x_n + i = b_i - \sum a_{ij}x_j \geq 0$

松弛变量 x_{n+i}

$$\sum_{j=1}^{n} a_{ij}x_j \leq b_i \to x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij}x_j, x_{n+i} \geq 0$$

等式左侧为基本变量，右侧为非基本变量。

变量

- 替入变量 x_e（非基变量）
- 替出变量 x_l（基本变量）

可行解

- 基本解：所有非基变量设为 0，基本变量为右侧的常数
- 基本可行解：所有 $b_i \geq 0$

> 注：单纯形法的过程中 B 和 N 不断交换，在 n 维空间中不断走，「相当于不等式上的高斯消元」。

转轴

选取一个非基本变量 x_e 为替入变量，基本变量 x_l 为替出变量，将其互换，为了防止循环，根据 **Bland 规则**，选择下标最小的变量。

> **Bland 规则**可以参看：最优化方法[1]

初始化

在所有 $b_i < 0$ 的约束中随机选一个作为 x_l，再随机选一个 $a_{le} < 0$ 作为 x_e，然后 $pivot(l, e)$ 后 b_i 就变正了。

算法实现

每个约束定义了 n 维空间中的一个半空间（超平面），交集形成的可行域是一个凸区域称为单纯形。目标函数是一个超平面，最优解在凸区域定点处取得。通过不断的转轴操作，在 n 维凸区域的顶点上不断移动（转轴），使得基本解的目标值不断变大，最终达到最优解。

> 以下问题可以转换为单纯形：
>
> - 最短路
> - 最大流
> - 最小费用最大流
> - 多商品流

基本思想就是改写 l 这个约束为 x_e 作为基本变量，然后把这个新 x_e 的值带到其他约束和目标函数中，就消去 x_e 了。改写和带入时要修改 b 和 a，目标函数则是 c 和 v。

转动时，l 和 e 并没有像算法导论上一样，a 矩阵用了两行分别是 $a_{l,\square}$ 和 $a_{e,\square}$（这样占用内存大），而是用了同一行，这样 a 矩阵的行数 $= |B|$，列数 $= |N|$。

也就是说，约束条件只用 m 个，尽管 B 和 N 不断交换，但同一时间还是只有 m 个约束（基本变量），n 个非基变量，注意改写成松弛型后 a 矩阵实际系数为负。（一个优化为 $a_{i,e}$ 的约束没必要带入了）。

simplex 是主过程，基本思想是找到一个 $c_e > 0$ 的，然后找对这个 e 限制最紧的 l，转动这组 l, e，注意精度控制 ϵ，$c_e > \epsilon$，还有找 l 的时候 $a_{i,e} > \epsilon$ 才行。

" **例题 「NOI2008」 志愿者招募**[2]"

题目大意：长度为 n 的序列，第 i 位至少 b_i，m 种区间使 $[l_i, r_i] + 1$ 代价为 a_i。

原始问题 m 个变量，n 个约束，当 $l_j \leq i \leq r_j$，$a_{ij} = 1$。

对偶问题 n 个变量，m 个约束

$$\max \sum_{i=1}^{n} b_i y_i$$

$$s.t \sum_{l_i \leq j \leq r_i} y_j \leq c_i, y_i \geq 0$$

把对应出的系数矩阵代入到单纯形算法就可以求出最优解了。

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
using namespace std;
const int M = 10005, N = 1005, INF = 1e9;

int read() {  // 快读
  char c = getchar();
  int x = 0, f = 1;
  while (c < '0' || c > '9') {
    if (c == '-') f = -1;
    c = getchar();
  }
  while (c >= '0' && c <= '9') {
    x = x * 10 + c - '0';
    c = getchar();
  }
  return x * f;
}

int n, m;
double a[M][N], b[M], c[N], v;

void pivot(int l, int e) {  // 转轴操作函数
  b[l] /= a[l][e];
  for (int j = 1; j <= n; j++)
    if (j != e) a[l][j] /= a[l][e];
  a[l][e] = 1 / a[l][e];

  for (int i = 1; i <= m; i++)
    if (i != l && fabs(a[i][e]) > 0) {
      b[i] -= a[i][e] * b[l];
      for (int j = 1; j <= n; j++)
        if (j != e) a[i][j] -= a[i][e] * a[l][j];
      a[i][e] = -a[i][e] * a[l][e];
    }
```

```
  v += c[e] * b[l];
  for (int j = 1; j <= n; j++)
    if (j != e) c[j] -= c[e] * a[l][j];
  c[e] = -c[e] * a[l][e];

  // swap(B[l],N[e])
}

double simplex() {
  while (true) {
    int e = 0, l = 0;
    for (e = 1; e <= n; e++)
      if (c[e] > (double)0) break;
    if (e == n + 1) return v;  // 此时 v 即为最优解
    double mn = INF;
    for (int i = 1; i <= m; i++) {
      if (a[i][e] > (double)0 && mn > b[i] / a[i][e]) {
        mn = b[i] / a[i][e];  // 找对这个 e 限制最紧的 l
        l = i;
      }
    }
    if (mn == INF) return INF;  // unbounded
    pivot(l, e);                // 转动 l,e
  }
}

int main() {
  n = read();
  m = read();
  for (int i = 1; i <= n; i++) c[i] = read();
  for (int i = 1; i <= m; i++) {
    int s = read(), t = read();
    for (int j = s; j <= t; j++) a[i][j] = 1;  // 表示第 i 种志愿者在 j 时间可以服务
    b[i] = read();
  }
  printf("%d", (int)(simplex() + 0.5));
}
```

对偶原理

最大化与最小化互换，常数与目标函数互换，改变不等号，变量与约束对应。

$$\max\ c^Tx : Ax \le b, x \ge 0$$

$$\min\ b^Ty : A^Ty \ge c, y \ge 0$$

d_{uv} 表示 u, v 是否匹配

$$\max \sum_{(u,v)\in E} c_{uv}d_{uv}$$

$$s.t \begin{cases} \sum_{(v)\in Y} d_{uv} \le 1, u \in X \\ \sum_{(u)\in X} d_{uv} \le 1, v \in Y \\ d_{u,v} \in \{0,1\} \end{cases}$$

令 p_u, p_v 为两类约束对偶之后的变量

$$\min \sum_{u \in x} p_u + \sum_{v \in Y} p_v$$

$$s.t \begin{cases} p_u + p_v \geq c_{uv} \\ u \in X, v \in Y \\ p_u, p_v \geq 0 \end{cases}$$

全幺模矩阵（Totally Unimodular Matrix）

充分条件：

- 仅有 $-1, 0, 1$ 构成
- 每列至多两个非零数
- 行可分为两个集合：
  - 一列包含两个同号非零数，两行不在同一个集合
  - 一列包含两个异号非零数，两行在同一个集合

线性规划中 A 为全幺模矩阵，则单纯形法过程中所有系数 $\in -1, 0, 1$，可以去除系数为 0 的项进行优化！

> 注：**任何最大流、最小费用最大流的线性规划都是全幺模矩阵**

更多详细的解释参看：https://www.cnblogs.com/ECJTUACM-873284962/p/7097864.html[3]

习题练习

- UOJ#179. 线性规划[4]

参考资料

- 线性规划之单纯形法 【超详解 + 图解】[3]
- 2016 国家集训队论文[5]
- 算法导论

参考资料与注释

[1] 最优化方法

[2] 「NOI2008」志愿者招募

[3] https://www.cnblogs.com/ECJTUACM-873284962/p/7097864.html [3-1] [3-2]

[4] UOJ#179. 线性规划

[5] 2016 国家集训队论文

9.17 群论

9.17.1 群论简介

引入

在数学和抽象代数中，**群论**（Group Theory）主要研究叫做「群」的代数结构。

定义

在数学中，**群**（group）是由一种集合以及一个二元运算所组成的，符合「群公理」的代数结构。

一个群是一个集合 G 加上对 G 的二元运算。二元运算用 $\cdot$ 表示，它结合了任意两个元素 a 和 b 形成了一个属于 G 的元素，记为 $a \cdot b$。

群公理包含下述四个性质（有时略去封闭性，只有三个性质）。若集合 $G \neq \emptyset$ 和 G 上的运算 $\cdot$ 构成的代数结构 $(G, \cdot)$ 满足以下性质：

1. **封闭性**：对于所有 G 中 a, b，运算 $a \cdot b$ 的结果也在 G 中。
2. **结合律**（associativity）：对于 G 中所有的 a, b, c，等式 $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ 成立。
3. **单位元**（identity element，也称幺元）：G 中存在一个元素 e，使得对于 G 中的每一个元素 a，都有一个 $e \cdot a = a \cdot e = a$ 成立。这样的元素是独一无二的。它被称为群的单位元。
4. **逆元**（inverse element）：对于每个 G 中的 a，总存在 G 中的一个元素 b 使 $a \cdot b = b \cdot a = e$，此处 e 为单位元，称 b 为 a 的逆元，记为 a^{-1}。

则称 $(G, \cdot)$ 为一个**群**。例如，整数集和整数间的加法 $(\mathbb{Z}, +)$ 构成一个群，单位元是 0，一个整数的逆元是它的相反数。

群的衍生结构

- 若代数结构 $(G, \cdot)$ 满足封闭性、结合律性质，则称 $(G, \cdot)$ 为一个**半群**（semigroup）。
- 若半群 $(G, \cdot)$ 还满足单位元性质，则称 $(G, \cdot)$ 为一个**幺半群**（monoid）。
- 若群 $(G, \cdot)$ 还满足**交换律**（commutativity）：对于 G 中所有的 a, b，等式 $a \cdot b = b \cdot a$ 成立。

  则称 $(G, \cdot)$ 为一个**阿贝尔群**（Abelian group），又称**交换群**（commutative group）。

环

形式上，**环**（ring）是一个集合 R 及对 R 的两个二元运算：加法 $+$ 和乘法 $\cdot$（注意这里不是我们一般所熟知的四则运算加法和乘法）所组成的，且满足如下性质的代数结构 $(R, +, \cdot)$：

1. $(R, +)$ 构成交换群，其单位元记为 0，R 中元素 a 的加法逆元记为 $-a$。
2. $(R, \cdot)$ 构成半群。
3. **分配律**（distributivity）：对于 R 中所有的 a, b, c，等式 $a \cdot (b + c) = a \cdot b + a \cdot c$ 和 $(a + b) \cdot c = a \cdot c + b \cdot c$ 成立。

> warning
>
> 在有的定义中，环必须存在乘法单位元；相对地，不存在乘法单位元的则被称为**伪环**（rng 或 pseudo-ring）。遇到的时候需根据上下文加以判断。
>
> 维基百科采用的就是这种定义：[1]

warning

In the terminology of this article, a ring is defined to have a multiplicative identity, while a structure with the same axiomatic definition but without the requirement for a multiplicative identity is instead called a rng (IPA:/r ŋ/). For example, the set of even integers with the usual + and is a rng, but not a ring. As explained in § History below, many authors apply the term "ring" without requiring a multiplicative identity.

在抽象代数中，研究环的分支为**环论**。

环的衍生结构

- 若环 R 上的乘法还满足交换律，则称 R 为**交换环**（commutative ring）。
- 若环 R 存在乘法单位元 1，则称 R 为**幺环**（ring with identity）。
- 若幺环 R 的所有非 0 元素 a 存在乘法逆元 a^{-1}，则称 R 为**除环**（division ring）。

域

域（field）是一个比环性质更强的代数结构，具体地，域是交换除环。

域的研究方法和环大不相同。在抽象代数中，研究域的分支为**域论**。

群的基本概念

定义 $X = x, y, z$ 为包含 x, y, z 元素的集合 X。$x \in X$ 表示 x 属于集合 X。$f : X \to Y$ 表示 f 是一个与 X 的每个元素和 Y 的元素相关联的函数。

在研究集合时，我们使用子集（subset）、函数（function）和等价关系商（quotient by an equivalence relation）等概念。在研究群时，我们通过等价关系用子群（subgroup）、同态（homomorphism）和商群（quotient group）来代替。

群同态

群同态是保持群结构的函数，可用于关联两个群。

从群 $(G, \cdot)$ 到群 $(H, *)$ 的同态是一个函数 $\varphi : G \to H$ 使得对于 G 中所有的元素 a 和 b：

$$\varphi(a \cdot b) = \varphi(a) * \varphi(b)$$

子群

子群：群 $(G, \cdot), (H, \cdot)$，满足 $H \subseteq G$，则 $(H, \cdot)$ 是 $(G, \cdot)$ 的子群。

子群是包含在更大的群 G 内的一个群 H。它具有 G 的元素的子集和相同操作。这意味着 G 的单位元素必须包含在 H 中，并且每当 h_1 和 h_2 都在 H 中，那么 $h_1 \cdot h_2$ 和 h_1^{-1} 也在 H 中。所以 H 中的元素，和在 G 上的限制为 H 的群操作，形成了一个群体。

即，若 $(G, \cdot)$ 是群，H 是 G 的非空子集，且 $(H, \cdot)$ 也是群，则称 $(H, \cdot)$ 是 $(G, \cdot)$ 的**子群**。

子群检验法（subgroup test）是群 G 的子集 H 是子群的充分必要条件：对于所有元素 $g, h \in H$，$g^{-1} \cdot h \in H$。

陪集

陪集（coset）是一个群的子集，它包含通过将群的一个固定元素乘以给定子群的每个元素在右边或左边相乘以得到的所有乘积。

在许多情况下，两个群元素可能是等价的。例如，在正方形的对称群中，一旦进行了反射，仅靠旋转就不能使正方形回到原来的位置，所以可以认为正方形的反射位置相互等价，而不等价于未反射的位置；旋转操作与是否进行了反射无关。陪集被用来正式表达这个观点：一个子群 H 决定了左右陪集，陪集可以说是 H 经过任何群元素 g 的变换

（即左乘或右乘）。用符号表示，包含元素 g 的 H 的左右陪集分别是：

$$gH = \{g \cdot h \mid h \in H\}$$
$$Hg = \{h \cdot g \mid h \in H\}$$

令 $[G : H]$ 表示 G 中 H 的左陪集数（等价于右陪集数）。

共轭

如果群中有一个元素 g 使得 $b = g^{-1}ag$，群的两个元素 a 和 b 是**共轭**（conjugate）的。这是一个等价关系，其等价类称为**共轭类**（conjugacy classes）。

正规子群

正规子群（normal subgroup）是在共轭变换下不变的子群；换句话说，如果对于所有 $g \in G, h \in H$，都有 $g^{-1}hg \in H$，则 H 为 G 的正规子群，记作 $H \lhd G$。

生成子群

$S \subset G$ 的**生成子群**（generated subgroup）$\langle S\rangle$ 是 G 的包含 S 的最小子群，也是 G 的包含 S 的所有子群的交，称 S 为**群的生成集** (generating set of a group)。

如果 $G = \langle S\rangle$，我们称 S 生成 G，S 中的元素叫做生成元或群生成元。

S 中只有一个元素 x 时，$\langle S\rangle$ 通常写为 $\langle x\rangle$。在这种情况下，$\langle x\rangle$ 是 x 的幂的循环子群（即 $\langle a\rangle = \{a^k, k \geq 1\}$），我们称这个循环群是用 x 生成的。

商群

商群（quotient group）或因子群（factor group）是通过使用保留一些群结构的等价关系聚合更大群的相似元素获得的群。

在某些情况下，子群的陪集集可以被赋予群律，给出商群或因子群。为了使其成立，子群必须是正规子群 (normal subgroup)。给定任何正规子群 N，商群定义为

$$G/N = \{gN \mid g \in G\}$$

阶

群 G 的阶是它元素的个数，记作 $\operatorname{ord}(G)$ 或 $|G|$，无限群有无限阶。

群 G 内的一个元素 a 的阶是使 $a^m = e$ 成立的最小正整数 m，记作 $\operatorname{ord}(a)$ 或 $|a|$，等于 $\operatorname{ord}(\langle a\rangle)$。若这个数不存在，则称 a 有无限阶。有限群的所有元素都有有限阶。

例如，群 $Z_n^\times = \{a \in \{0, 1, \cdots, n-1\} \mid \gcd(a, n) = 1\}$ 的阶为 $\varphi(n)$，其中元素 x 的阶为满足 $x^r \equiv 1 \pmod n$ 的最小正整数 r（这正是数论中 x 模 n 的阶）。

拉格朗日定理：如果 H 是 G 的子群，那么 $|G| = [G : H]|H|$。

证明的简要思路是：（左/右）陪集大小等于子群大小；而每个陪集要么不相交要么相等，且所有陪集的并是集合 G；那么陪集数就等于 G 与 H 的阶之比。

由拉格朗日定理可立即得到：群中任意一个元素的阶，一定整除群的阶。

如果群 G 中存在两个元素 a、b 的阶 m、n 互素，那么 $a^sb^t = e$ 当且仅当 $a^s = e$ 并且 $b^t = e$。

" 证明"

显然，在 $a^sb^t = e$ 成立的情况下，$a^s = e$ 和 $b^t = e$ 等价，所以不成立只能同时不成立。

反证法。如果 $a^sb^t = e$，但是两个部分 a^s、b^t 都不是单位元，那么 $e = a^sm = b^{-tm}$。因为 $\gcd(-m, n) = 1$，根据裴蜀定理或者乘法逆元，可以去掉 $-m$，得到 $e = b^t$，矛盾。

" 有关阶的常见误区"

1. 群 G 的阶一定等于其中所有元素阶的最大值（或 lcm）。

反例：二面体群 D_4（相当于群 $(\{0,1,2,3\},\oplus)$，其中 $\oplus$ 表示异或）的阶是 4，但是除了 e 的阶为 1，其他元素的阶都是 2。

2. 如果群 G 中存在两个元素 x_1、x_2 的阶是 d_1、d_2，那么 G 中一定存在阶为 $d = \mathrm{lcm}(d_1, d_2)$ 的元素。

反例：对称群 S_3（相当于 $X = \{1,2,3\}$ 的置换群）中存在阶为 2 和 3 的元素，却不存在阶为 6 的元素。

群的主要类别

置换群

置换群（Permutation group）是第一类被系统性研究的群。对给定的集合 X，X 到自身的一些置换集合 G 如果在复合运算和求逆运算下封闭，那么称 G 是一个作用于 X 上的群。详细内容请看 置换群 章节。

循环群

循环群（cyclic group，记作 C_n）是最简单的群。群 G 中任意一个元素 a 都可以表示为 $a = g^k$，其中 k 为整数。称 g 为群 G 的生成元。

生成元 g 的阶就是群 G 的阶。

" 证明"

记 G 的单位元为 e。由于 G 有限，对生成元 g 不断做幂运算，必然会在某时重复，即存在不同的整数 i 和 j 使得 $g^i = g^j$。两边同时去掉若干个 g 就有非 0 整数 n 使得 $g^n = e$。显然 $g^0 = e$。

设生成元 g 的阶是 d。G 中任意一个元素 a 都可以表示为 g 的幂，因此 d 不可能小于 m。否则 g 的幂当中出现 d 个元素之后就回到了单位元 e，剩余的元素就不能被 g 的幂表示，矛盾。

同样的，d 也不可能大于 m。否则在前 d 个 g 的幂中就会出现重复，存在不同的整数 i 和 j 使得 $g^i = g^j$，再得到的 $g^n = e$，n 介于 0 和 d 之间，就与 d 的最小性矛盾。因此，$d = m$。证完。

阶为 m 的有限循环群 G 同构于模 m 剩余类对于加法构成的群 Z_m。

" 证明"

构造映射 f：$Z_m \to G$，$f(n) = g^n$，可见 f 为双射，并且对于任意的 i 和 j，$f(i + j) = g^i g^j$。因此同构。证完。

矩阵群

矩阵群（Matrix group）或线性群（Linear group）是 G 是一个由给定 n 阶可逆矩阵组成的集合，该矩阵在域 K 上在乘积和逆矩阵下闭合。这样的群通过线性变换作用于 n 维向量空间 K^n。

矩阵群常见例子为**李群**（Lie group）。

变换群

置换群和矩阵群是**变换群**（Transformation group）的特例。

群作用于某个空间 X 并保留其固有结构。在置换群的情况下，X 是一个集合；对于矩阵群，X 是向量空间。变换群的概念与对称群的概念密切相关：变换群通常由所有保持某种结构的变换组成。

抽象群

抽象群（Abstract group）通常通过生成器和关系来表示：

$$G = \langle S | R \rangle$$

抽象群主要来源是通过正规子群 H 构造群 G 的商群 G/H。如果群 G 是集合 X 上的置换群，则商群 G/H 不再作用于 X；但是抽象群的概念允许人们不必担心这种差异。

参考资料与注释

- Group (mathematics) - Wikipedia[2]
- Group theory - Wikipedia[3]
- Group - Wolfram MathWorld[4]
- Visual Group Theory[5]

[1] Ring（mathematics）- Wikipedia

[2] Group (mathematics) - Wikipedia

[3] Group theory - Wikipedia

[4] Group - Wolfram MathWorld

[5] Visual Group Theory

9.17.2 置换群

Authors: Wajov

引入

置换群通常用来解决一些涉及「本质不同」的计数问题，例如用 3 种颜色给一个立方体染色，求本质不同的方案数（经过翻转后相同的两种方案视为同一种）。

有关群的定义、子群的定义，见 群论基础。

置换

置换的相关定义可见 置换和排列。

置换群

集合 S 上的所有置换关于置换的乘法满足封闭性、结合律、有单位元（恒等置换，即每个元素映射成它自己）、有逆元（交换置换表示中的上下两行），因此构成一个群。这个群的任意一个**子群**即称为**置换群**。

循环置换

循环置换是一类特殊的置换，可表示为

$$(a_1, a_2, \dots, a_m) = \begin{pmatrix} a_1, a_2, \dots, a_{m-1}, a_m \\ a_2, a_3, \dots, a_m, a_1 \end{pmatrix}$$

若两个循环置换不含有相同的元素，则称它们是**不相交**的。有如下定理：

任意一个置换都可以分解为若干不相交的循环置换的乘积，例如

$$\begin{pmatrix} a_1, a_2, a_3, a_4, a_5 \\ a_3, a_1, a_2, a_5, a_4 \end{pmatrix} = (a_1, a_3, a_2) \circ (a_4, a_5)$$

该定理的证明也非常简单。如果把元素视为图的节点，映射关系视为有向边，则每个节点的入度和出度都为 1，因此形成的图形必定是若干个环的集合，而一个环即可用一个循环置换表示。

Burnside 引理

前面的都算是铺垫，接下来我们进入正题。

定义

设 A 和 B 为有限集合，X 为一些从 A 到 B 的映射组成的集合。

G 是 A 上的置换群，且 X 中的映射在 G 中的置换作用下封闭。

X/G 表示 G 作用在 X 上产生的所有等价类的集合

（若 X 中的两个映射经过 G 中的置换作用后相等，则它们在同一等价类中），则

$$|X/G| = \frac{1}{|G|}\sum_{g\in G}|X^g|$$

其中 $|S|$ 表示集合 S 中元素的个数，且

$$X^g = \{x|x \in X, g(x) = x\}$$

是不是觉得很难懂？别急，请看下面的例子。

解释

我们还是以给立方体染色为例子，则上面式子中一些符号的解释如下：

- A：立方体 6 个面的集合
- B：3 种颜色的集合
- X：直接给每个面染色，不考虑本质不同的方案的集合，共有 3^6 种
- G：各种翻转操作构成的置换群
- X/G：本质不同的染色方案的集合
- X^g：对于某一种翻转操作 g，所有直接染色方案中，经过 g 这种翻转后保持不变的染色方案的集合

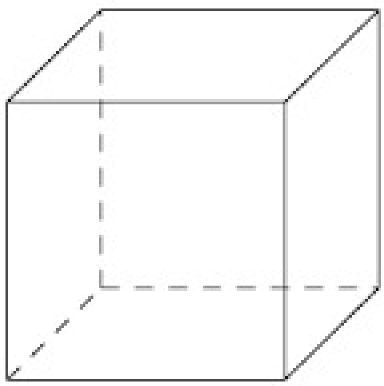


图 9.36

接下来我们需要对 G 中的所有置换进行分析，它们可以分为以下几类（方便起见，将立方体的 6 个面分别称为前、后、上、下、左、右）：

- 不动：即恒等变换，因为所有直接染色方案经过恒等变换都不变，因此它对应的 $|X^g| = 3^6$
- 以两个相对面的中心连线为轴的 90° 旋转：相对面有 3 种选择，旋转的方向有两种选择，因此这类共有 6 个置换。假设选择了前、后两个面中心的连线为轴，则必须要满足上、下、左、右 4 个面的颜色一样，才能使旋转后不变，因此它对应的 $|X^g| = 3^3$
- 以两个相对面的中心连线为轴的 180° 旋转：相对面有 3 种选择，旋转方向的选择对置换不再有影响，因此这类共有 3 个置换。假设选择了前、后两个面中心的连线为轴，则必须要满足上、下两个面的颜色一样，左、右两个面的颜色一样，才能使旋转后不变，因此它对应的 $|X^g| = 3^4$
- 以两条相对棱的中点连线为轴的 180° 旋转：相对棱有 6 种选择，旋转方向对置换依然没有影响，因此这类共有 6 个置换。假设选择了前、上两个面的边界和下、后两个面的边界作为相对棱，则必须要满足前、上两个面的颜色一样，下、后两个面的颜色一样，左、右两个面的颜色一样，才能使旋转后不变，因此它对应的 $|X^g| = 3^3$
- 以两个相对顶点的连线为轴的 120° 旋转：相对顶点有 4 种选择，旋转的方向有两种选择，因此这类共有 8 个置换。假设选择了前面的右上角和后面的左下角作为相对顶点，则必须满足前、上、右三个面的颜色一样，后、下、左三个面的颜色一样，才能使旋转后不变，因此它对应的 $|X^g| = 3^2$

因此，所有本质不同的方案数为

$$\frac{1}{1+6+3+6+8}(3^6 + 6 \times 3^3 + 3 \times 3^4 + 6 \times 3^3 + 8 \times 3^2) = 57$$

证明

看懂本部分需要群论的相关知识，如果你没有学习过群论或者对证明过程没有兴趣，建议直接跳过本部分。

为了证明 Burnside 引理，需要先引入轨道稳定子定理（Orbit-Stabilizer Theorem，也称轨道 - 稳定集定理）。

轨道稳定子定理 G 和 X 的定义同上，$\forall x \in X, G^x = \{g|g(x) = x, g \in G\}, G(x) = \{g(x)|g \in G\}$，其中 G^x 称为 x 的**稳定子**，$G(x)$ 称为 x 的**轨道**，则有

$$|G| = |G^x||G(x)|$$

轨道稳定子定理的证明首先可以证明 G^x 是 G 的子群，因为

- 封闭性：若 $f, g \in G$，则 $f \circ g(x) = f(g(x)) = f(x) = x$，所以 $f \circ g \in G^x$
- 结合律：显然置换的乘法满足结合律
- 单位元：因为 $I(x) = x$，所以 $I \in G^x$（I 为恒等置换）
- 逆元：若 $g \in G^x$，则 $g^{-1}(x) = g^{-1}(g(x)) = g^{-1} \circ g(x) = I(x) = x$，所以 $g^{-1} \in G^x$

则由群论中的拉格朗日定理，可得

$$|G| = |G^x|[G : G^x]$$

其中 $[G : G^x]$ 为 G^x 不同的左陪集个数。接下来只需证明 $|G(x)| = [G : G^x]$，我们将其转化为证明存在一个从 $G(x)$ 到 G^x 所有不同左陪集的双射。令 $\varphi(g(x)) = gG^x$，下证 φ 为双射

- 若 $g(x) = f(x)$，两边同时左乘 f^{-1}，可得 $f^{-1} \circ g(x) = I(x) = x$，所以 $f^{-1} \circ g \in G^x$，由陪集的性质可得 $(f^{-1} \circ g)G^x = G^x$，即 $gG^x = fG^x$
- 反过来可证，若 $gG^x = fG^x$，则有 $g(x) = f(x)$
- 以上两点说明对于一个 $g(x)$，只有一个左陪集与其对应，即 φ 是一个从 $G(x)$ 到左陪集的映射
- 又显然 φ 有逆映射，因此 φ 是一个双射

Burnside 引理的证明

$$\sum_{g \in G} |X^g| = |\{(g, x)|(g, x) \in G \times X, g(x) = x\}| \tag{9.46}$$

$$= \sum_{x \in X} |G^x| \tag{9.47}$$

$$
\begin{aligned}
&= \sum_{x\in X} \frac{|G|}{|G(x)|} && (9.48)\\
&= |G| \sum_{x\in X} \frac{1}{|G(x)|} && (9.49)\\
&= |G| \sum_{Y\in X/G} \sum_{x\in Y} \frac{1}{|G(x)|} && (9.50)\\
&= |G| \sum_{Y\in X/G} \sum_{x\in Y} \frac{1}{|Y|} && (9.51)\\
&= |G| \sum_{Y\in X/G} 1 && (9.52)\\
&= |G||X/G| && (9.53)
\end{aligned}
$$

所以有

$$|X/G| = \frac{1}{|G|} \sum_{g\in G} |X^g|$$

Pólya 定理

定义

在与 Burnside 引理相同的前置条件下，若 X 为**所有**从 A 到 B 的映射，内容修改为

$$|X/G| = \frac{1}{|G|} \sum_{g\in G} |B|^{c(g)}$$

其中 $c(g)$ 表示置换 g 能拆分成的不相交的循环置换的数量。

解释

依然考虑立方体染色问题。分析刚才提到的以相对棱的中点连线为轴的 180° 旋转，如果将前、后、上、下、左、右 6 个面依次编号为 1 到 6，则该置换可以表示为（翻转后原来编号为 1 的面的位置变为了编号为 3 的面，以此类推）

$$\begin{pmatrix} 1,3,2,4,5,6 \\ 3,1,4,2,6,5 \end{pmatrix} = (1,3)\circ(2,4)\circ(5,6)$$

因此 $c(g) = 3, |B|^{c(g)} = 3^3$，与刚才在 Burnside 引理中分析的结果相同。

证明

在 Burnside 引理中，显然 $g(x) = x$ 的充要条件是 x 将 g 中每个循环置换的元素都映射到了 B 中的同一元素，所以 $|X^g| = |B|^{c(g)}$，即可得 Pólya 定理。

9.18 概率论

9.18.1 基本概念

概述

在研究具体的随机现象时我们通常着重关注以下要素：

- 样本空间 Ω，指明随机现象所有可能出现的结果。
- 事件域 $\mathcal{F}$，表示我们所关心的所有事件。
- 概率 P，描述每一个事件发生的可能性大小。

样本空间、随机事件

定义

一个随机现象中可能发生的不能再细分的结果被称为**样本点**。所有样本点的集合称为**样本空间**，通常用 Ω 来表示。

一个**随机事件**是样本空间 Ω 的子集，它由若干样本点构成，用大写字母 $A, B, C, \cdots$ 表示。

对于一个随机现象的结果 ω 和一个随机事件 A，我们称事件 A **发生了**当且仅当 $\omega \in A$。

例如，掷一次骰子得到的点数是一个随机现象，其样本空间可以表示为 $\Omega = \{1,2,3,4,5,6\}$。设随机事件 A 为「获得的点数大于 4」，则 $A = \{5,6\}$。若某次掷骰子得到的点数 $\omega = 3$，由于 $\omega \notin A$，故事件 A 没有发生。

事件的运算

由于我们将随机事件定义为了样本空间 Ω 的子集，故我们可以将集合的运算（如交、并、补等）移植到随机事件上。记号与集合运算保持一致。

特别的，事件的并 $A \cup B$ 也可记作 $A + B$，事件的交 $A \cap B$ 也可记作 AB，此时也可分别称作**和事件**和**积事件**。

事件域

研究具体的随机现象时我们需要明确哪些事件是我们感兴趣的。根据随机事件的定义，显然有 $\mathcal{F} \subset 2^{\Omega}$（记号 2^{Ω} 表示由 Ω 的所有子集组成的集合族），但 $\mathcal{F} = 2^{\Omega}$ 却不是必须的。这在样本空间 Ω 有限时可能有些难以理解，毕竟 2^{Ω} 尽管更大了但仍然有限。而当 Ω 为无穷集时，2^{Ω} 的势变得更大，其中也难免会出现一些「性质不太好」且我们不关心的事件，这时为了兼顾这些事件而放弃一些性质就显得得不偿失了。

尽管 $\mathcal{F} = 2^{\Omega}$ 不是必须的，这并不代表 2^{Ω} 的任一子集都能成为事件域。我们通常会对一些事件进行运算得到的结果事件的概率感兴趣，因此我们希望事件域 $\mathcal{F}$ 满足下列条件：

- $\emptyset \in \mathcal{F}$；
- 若 $A \in \mathcal{F}$，则补事件 $\bar{A} \in \mathcal{F}$；
- 若有一列事件 $A_n \in \mathcal{F}, n = 1,2,3\dots$，则 $\bigcup A_n \in \mathcal{F}$。

简言之，就是事件域 $\mathcal{F}$ 对在补运算、和可数并下是封闭的，且包含元素 $\emptyset$。

可以证明满足上述三个条件的事件域 $\mathcal{F}$ 对可数交也是封闭的。

以掷骰子为例，当样本空间记为 $\Omega = \{1,2,3,4,5,6\}$ 时，以下两个集合能够成为事件域：

- $\mathcal{F}_1 = \{\emptyset, \Omega\}$
- $\mathcal{F}_2 = \{\emptyset, \{1,3,5\}, \{2,4,6\}, \Omega\}$

但以下两个集合则不能

- $\mathcal{F}_3 = \{\emptyset, \{1\}, \Omega\}$（对补不封闭）
- $\mathcal{F}_4 = \{\{1,3,5\}, \{2,4,6\}\}$（不含有 $\emptyset$ 且对并不封闭）

概率

定义

古典定义

在概率论早期实践中，由于涉及到的随机现象都比较简单，具体表现为样本空间 Ω 是有限集，且直观上所有样本点是等可能出现的，因此人们便总结出了下述定义：

如果一个随机现象满足：

- 只有有限个基本结果；
- 每个基本结果出现的可能性是一样的；

那么对于每个事件 A，定义它的概率为

$$P(A) = \frac{\#(A)}{\#(\Omega)}$$

其中 $\#(\cdot)$ 表示对随机事件（一个集合）大小的度量。

后来人们发现这一定义可以直接推广到 Ω 无限的一部分情景中，于是就有了所谓 几何概型[1]。

公理化定义

上述基于直观认识的定义在逻辑上有一个很大的漏洞：在定义「概率」这一概念时用到了「可能性」这一说法，产生了循环定义的问题。同时「等可能」在样本空间无限时会产生歧义，由此产生了包括 Bertrand 悖论[2] 在内的一系列问题。

经过不断探索，苏联数学家柯尔莫哥洛夫于 1933 年在他的《概率论基础》一书中第一次给出了概率的公理化定义：

概率函数 P 是一个从事件域 $\mathcal{F}$ 到闭区间 $[0,1]$ 的映射，且满足：

- **规范性**：事件 Ω 的概率值为 1，即 $P(\Omega)=1$。
- **可数可加性**：若一列事件 $A_1, A_2, \cdots$ 两两不交，则 $P\left(\bigcup_{i\geq 1} A_i\right) = \sum_{i\geq 1} P(A_i)$。

概率函数的性质

对于任意随机事件 $A, B \in \mathcal{F}$，有

- **单调性**：若 $A \subset B$，则有 $P(A) \leq P(B)$。
- **容斥原理**：$P(A+B) = P(A) + P(B) - P(AB)$。
- $P(A-B) = P(A) - P(AB)$，这里 $A-B$ 表示差集。

概率空间

我们在一开始提到，研究具体的随机现象时我们通常关注样本空间 Ω、事件域 $\mathcal{F}$ 以及概率函数 P。我们将三元组 $(\Omega, \mathcal{F}, P)$ 称为一个概率空间。

概率只有在确定的概率空间下讨论才有意义。我们前面提到的 Bertrand 悖论归根结底就是因对样本空间 Ω 的定义不明确而产生的。

参考资料与注释

- 概率论 （ 数学分支 ） _ 百度百科[3]
- Probability - Wikipedia[4]

参考资料与注释

[1] 几何概型

[2] Bertrand 悖论

[3] 概率论（数学分支）_ 百度百科

[4] Probability - Wikipedia

9.18.2 条件概率与独立性

概述

当某事件已经发生时，一些随机事件的概率会因为已知信息的增加发生变化。例如在手游抽卡时，我们可能会认为单次抽卡出六星与不出六星是等概率的，但随着我们连抽 50 发一个六星都没有，再固执地认为「出六星与不出六星等概率」就显得不是那么明智。

总之，研究在某些已知条件下事件发生的概率是必要的。

条件概率

定义

若已知事件 A 发生，在此条件下事件 B 发生的概率称为**条件概率**，记作 $P(B|A)$。

在概率空间 $(\Omega,\mathcal{F},P)$ 中，若事件 $A\in\mathcal{F}$ 满足 $P(A)>0$，则条件概率 $P(\cdot|A)$ 定义为

$$P(B|A)=\frac{P(AB)}{P(A)}\quad\forall B\in\mathcal{F}$$

可以验证根据上式定义出的 $P(\cdot|A)$ 是 $(\Omega,\mathcal{F})$ 上的概率函数。

根据条件概率的定义可以直接推出下面两个等式：

- **概率乘法公式**：在概率空间 $(\Omega,\mathcal{F},P)$ 中，若 $P(A)>0$，则对任意事件 B 都有

$$P(AB)=P(A)P(B|A)$$

- **全概率公式**：在概率空间 $(\Omega,\mathcal{F},P)$ 中，若一组事件 $A_1,\cdots,A_n$ 两两不交且和为 Ω，则对任意事件 B 都有

$$P(B)=\sum_{i=1}^{n}P(A_i)P(B|A_i)$$

Bayes 公式

一般来说，设可能导致事件 B 发生的原因为 $A_1,A_2,\cdots,A_n$，则在 $P(A_i)$ 和 $P(B|A_i)$ 已知时可以通过全概率公式计算事件 B 发生的概率。但在很多情况下，我们需要根据「事件 B 发生」这一结果反推其各个原因事件的发生概率。于是有

$$P(A_i|B)=\frac{P(A_iB)}{P(B)}=\frac{P(A_i)P(B|A_i)}{\sum_{j=1}^{n}P(A_j)P(B|A_j)}$$

上式即 Bayes 公式。

事件的独立性

在研究条件概率的过程中，可能会出现 $P(B|A)=P(B)$ 的情况。从直观上讲就是事件 B 是否发生并不会告诉我们关于事件 A 的任何信息，即事件 B 与事件 A「无关」。于是我们就有了下面的定义

定义

若同一概率空间中的事件 A,B 满足

$$P(AB)=P(A)P(B)$$

则称 A,B **独立**。对于多个事件 $A_1,A_2,\cdots,A_n$，我们称其独立，当且仅当对任意一组事件 $\{A_{i_k}:1\le i_1<i_2<\cdots<i_k\le n\}$ 都有

$$P(A_{i_1}A_{i_2}\cdots A_{i_r})=\prod_{k=1}^{r}P(A_{i_k})$$

多个事件的独立性

对于多个事件，一般不能从两两独立推出这些事件独立。考虑以下反例：

有一个正四面体骰子，其中三面被分别涂成红色、绿色、蓝色，另一面则三色皆有。现在扔一次该骰子，令事件 A,B,C 分别表示与桌面接触的一面包含红色、绿色、蓝色。

不难计算 $P(A)=P(B)=P(C)=\frac{1}{2}$，而 $P(AB)=P(BC)=P(CA)=P(ABC)=\frac{1}{4}$。

显然 A,B,C 两两独立，但由于 $P(ABC)\neq P(A)P(B)P(C)$，故 A,B,C 不独立。

9.18.3 随机变量

相关概念

随机变量

给定概率空间 $(\Omega,\mathcal{F},P)$，定义在样本空间 Ω 上的函数 $X:\Omega\to\mathbb{R}$ 若满足：对任意 $t\in\mathbb{R}$ 都有

$$\{\omega\in\Omega:X(\omega)\leq t\}\in\mathcal{F}$$

则称 X 为**随机变量**。

示性函数

对于样本空间 Ω 上的事件 A，定义随机变量

$$I_A(\omega)=\begin{cases}1, & \omega\in A\\ 0, & \omega\notin A\end{cases}$$

称 I_A 是事件 A 的**示性函数**。

分布函数

对于随机变量 X，称函数

$$F(x)=P(X\leq x)$$

为随机变量 X 的**分布函数**。记作 $X\sim F(x)$。

分布函数具有以下性质：

- **右连续性**：$F(x)=F(x+0)$
- **单调性**：在 $\mathbb{R}$ 上单调递增（非严格）
- $F(-\infty)=0$,$F(+\infty)=1$

同时我们可以证明，满足上述要求的函数都是某个随机变量的分布函数。因此，分布函数与随机变量之间一一对应。

随机变量的分类

随机变量按其值域（根据定义，随机变量是一个函数）是否可数分为**离散型**和**连续型**两种。

离散型随机变量

设 X 为离散型随机变量，其所有可能的取值为 $x_1,x_2,\cdots$，则我们可以用一系列形如 $P\{X=x_i\}=p_i$ 的等式来描述 X。这就是我们在高中课本中学过的**分布列**。

连续型随机变量

设 X 为连续型随机变量，考察 $P\{X=x\}$ 往往是无意义的（因为这一概率很可能是 0）。

" 为什么说概率「很可能」是 0"

考虑这样的随机变量 X：它以 $\frac{1}{2}$ 的概率取 0，以 $\frac{1}{2}$ 的概率服从开区间 $(0,1)$ 上的均匀分布。显然 X 满足连续型随机变量的定义。

对任何实数 $r\in(0,1)$，不难得到 $P\{X=r\}=0$，但同时有 $P\{X=0\}=\frac{1}{2}$。

另一方面，设 $X\sim F(x)$，则

$$P(l<x\leq l+\Delta x)=F(l+\Delta x)-F(l)$$

一个自然的想法是用极限 $\lim\limits_{\Delta x\to 0^+}\frac{F(l+\Delta x)-F(l)}{\Delta x}$ 来描述 X 取值为 l 的可能性。
这个式子就是我们熟知的导数，于是问题转化为寻找一个非负函数 $f(x)$ 使得

$$F(x)=\int_{-\infty}^{x}f(x)\mathrm{d}x$$

若这样的 $f(x)$ 存在，则称之为 X 的**密度函数**。

随机变量的独立性

前面讨论了随机事件的独立性。由于随机变量和随机事件紧密联系，我们还可以类似地给出随机变量独立性的定义。

定义

若随机变量 X,Y 满足对任意的 $x,y\in\mathbb{R}$ 都有

$$P(X\leq x,Y\leq y)=P(X\leq x)P(Y\leq y)$$

则称随机变量 X,Y **独立**。

Note

有些同学也许会注意到，中学课本中对随机变量独立性的定义是用形如 $P(X=\alpha)$ 的概率定义的，但由于连续性随机变量取特定值的概率通常是 0，故在更一般的情形下借助分布函数定义才是更加明智的选择。

性质

若随机变量 X,Y 相互独立，则对于任意函数 f,g，随机变量 $f(X)$ 与 $g(Y)$ 相互独立。

Warning 注意

有时候我们会研究相互独立的随机变量 X,Y 的某一函数 $f(X,Y)$（如 XY^2）的分布。

尽管 X 与 Y 是独立的，但不能想当然地认为对 Y 的某一取值 y，$f(X,y)$ 与 $f(X,Y)$ 服从同样的分布。

9.18.4 随机变量的数字特征

本文将介绍随机变量的期望、方差等数字特征。

期望

定义

离散型随机变量

设离散型随机变量 X 的概率分布为 $p_i = P\{X = x_i\}$，若和式

$$\sum x_i p_i$$

绝对收敛，则称其值为 X 的**期望**，记作 EX。

连续型随机变量

设连续型随机变量 X 的密度函数为 $f(x)$。若积分

$$\int_{\mathbb{R}} x f(x) \mathrm{d}x$$

绝对收敛，则称其值为 X 的**期望**，记作 EX。

统一定义

设随机变量 X 的分布函数为 $F(x)$，若 Stieltjes 积分[1]

$$\int_{\mathbb{R}} x \mathrm{d}F(x)$$

绝对收敛，则称其值为 X 的**期望**，记作 EX。

" 期望不存在的例子"

考虑有如下分布的离散型随机变量 X

$$P\left\{X = (-1)^k \frac{2^k}{k}\right\} = \frac{1}{2^k}, \quad k = 1, 2, \cdots$$

尽管和式 $\sum x_i p_i$ 收敛于 $-\ln 2$，但由于其不是绝对收敛的，故 X 的期望不存在。

再考虑有如下密度函数的连续型随机变量 Y

$$f(y) = \frac{1}{\pi} \cdot \frac{1}{1 + y^2}, \quad y \in (-\infty, +\infty)$$

容易验证 Y 的期望也不存在。

期望的性质

线性性

若随机变量 X, Y 的期望存在，则

- 对任意实数 a, b，有 $E(aX + b) = a \cdot EX + b$。
- $E(X + Y) = EX + EY$。

随机变量乘积的期望

若随机变量 X,Y 的期望存在且 X,Y 相互独立，则有

$$E(XY) = EX \cdot EY$$

注意：上述性质中的独立性**并非**必要条件。

" 反例"

考察随机变量 X 和 Y，其中 X 服从 $[-1,1]$ 上的均匀分布，$Y=X^2$。

期望与概率的转化

对于随机事件 A，考虑其示性函数 I_A：

$$I_A(\omega)=\begin{cases}1, & \omega\in A\\ 0, & \omega\notin A\end{cases}$$

根据定义可以求得其期望 $EI_A=P(A)$。这一转化在实际应用中非常常见。

例子

假设对于一个长为 n 的序列 $\{a_i\}$，其中 a_k 以 p_k 的概率取 k，以 $1-p_k$ 的概率取 0。考虑如何求 $S=\sum_{i=1}^n a_i$ 的期望。

如果使用定义直接求，需要求出 S 在每个可能取值处的概率，这个计算过程比较繁琐，这里不展开叙述。

另一方面，用 I_k 表示随机事件 $a_k=k$ 的示性函数，则有

$$S=\sum_{k=1}^n k\cdot I_k$$

进而不难求出

$$ES=E\left(\sum_{k=1}^n k\cdot I_k\right)=\sum_{k=1}^n k\cdot E[I_k]=\sum_{k=1}^n k\cdot p_k$$

条件分布与条件期望

我们之前研究过条件概率，类似的也可以提出所谓条件期望的概念。

定义

对于两个随机变量 X,Y，在已知 $Y=y$ 的条件下 X 的概率分布（密度函数）称之为**条件概率分布（条件概率密度）**，分别记作

$$P(X=x_i|Y=y)\qquad f_{X|Y}(x|y)$$

在此条件下，X 的期望称为**条件期望**，记作 $E[X|Y=y]$。

条件期望的性质

条件期望的诸多性质可由条件概率推知，在此不做赘述。

值得一提的是 $E[X|Y]$ 一般是随机变量 Y 的函数，且这个函数通常不是线性的。但实际上有

$$E[E[X|Y]]=EX$$

上式称作**全期望公式**。

应用

例题：[LA 7736]Pocky[2]

题意：有一根长为 L 的 Pocky，每次随机折成两段。若右边一段的长度不大于 d 则停止，否则对右边一段重复上述过程。求重复次数的期望。

" 题解"

记 $f(x)$ 表示长度为 x 的期望次数。$x \le d$ 的情形平凡。

当 $x > d$ 时，不妨设折断的位置距右端的长度为 k，则显然 $k \sim U[0,x]$，此时期望的重复次数为

$$g(k) = \begin{cases} 1, & k \le d \\ 1 + f(k), & k > d \end{cases}$$

由全期望公式可知

$$f(x) = Eg(k) = 1 + \frac{1}{x} \cdot \int_d^x f(t)\mathrm{d}t$$

解上述积分方程并代入初值条件得

$$f(x) = 1 + \ln \frac{x}{d}$$

方差

定义

设随机变量 X 的期望 EX 存在且期望

$$E(X - EX)^2$$

也存在，则称上式的值为随机变量 X 的**方差**，记作 DX 或 $Var(x)$。方差的算术平方根称为**标准差**，记作 $\sigma(X) = \sqrt{DX}$。

方差的性质

若随机变量 X 的方差存在，则

- 对任意常数 a, b 都有 $D(aX + b) = a^2 \cdot DX$
- $DX = E(X^2) - (EX)^2$

协方差与相关系数

一般来说，等式 $D(X + Y) = DX + DY$ 并不成立，我们自然会提出两个问题：

- $D(X + Y)$ 与 $DX + DY$ 之间相差的部分到底是什么。
- $D(X + Y)$ 与 $DX + DY$ 在什么情况下相等。

对于第一个问题，我们引入协方差作为解答。

协方差的定义

对于随机变量 X, Y，称

$$E((X - EX)(Y - EY))$$

为 X 与 Y 的**协方差**，记作 $\mathrm{Cov}(X, Y)$。

协方差的性质

对于随机变量 X, Y, Z，有

- $\mathrm{Cov}(X, Y) = \mathrm{Cov}(Y, X)$
- 对任意常数 a, b，有 $\mathrm{Cov}(aX + bY, Z) = a \cdot \mathrm{Cov}(X, Z) + b \cdot \mathrm{Cov}(Y, Z)$

同时协方差与方差也有如下联系：

- $DX = \mathrm{Cov}(X, X)$
- $D(X + Y) = DX + 2\,\mathrm{Cov}(X, Y) + DY$

" 关于协方差"

你可能会发现协方差的性质与向量内积的运算性质在形式上高度一致。

在泛函分析的视角下，对于给定的概率空间，其上的全体随机变量构成一个线性空间，而协方差是这个空间上的一个内积，标准差则是由该内积导出的范数。

对于刚才提出的第二个问题，不难看出 $D(X + Y) = DX + DY$ 当且仅当 $\mathrm{Cov}(X, Y) = 0$。一个直观的必要条件是 X 与 Y 独立，因为此时有

$$\mathrm{Cov}(X, Y) = E((X - EX)(Y - EY)) = E(X - EX)E(Y - EY) = 0$$

但这个条件并不是充分的。为了描述满足 $\mathrm{Cov}(X, Y) = 0$ 的随机变量 X,Y 之间的关系，我们引入相关系数

相关系数

对于随机变量 X, Y，称

$$\frac{\mathrm{Cov}(X, Y)}{\sigma(X)\sigma(Y)}$$

为 X 与 Y 的 **Pearson 相关系数**，记作 $\rho_{X,Y}$。

Pearson 相关系数描述了两个随机变量之间线性关联的紧密程度。$|\rho_{X,Y}|$ 越大，则 X 与 Y 之间的线性关联程度越强。不难证明 $|\rho_{X,Y}| \le 1$，且 $|\rho_{X,Y}| = 1$ 仅可能出现在以下两种情况

- 当存在实数 a 和正实数 b 使得 $P(X = a + bY) = 1$ 时，有 $\rho_{X,Y} = 1$；
- 当存在实数 a 和负实数 b 使得 $P(X = a + bY) = 1$ 时，有 $\rho_{X,Y} = -1$。

当 $\rho_{X,Y} = 0$ 时我们称随机变量 X 与 Y **不相关**，此时 X 和 Y 之间不存在线性关系。

"「不相关」与「独立」"

两随机变量不相关只是表明他们之间没有线性关联，并不代表没有其他形式的联系。

因此两随机变量 X, Y 不相关是他们相互独立的**必要而不充分**条件。

对于这一小节开头提到的第二个问题，我们给出结论：$\mathrm{Cov}(X, Y) = 0$ 的充要条件就是 X,Y 中的某一个以概率 1 取常值，或 X, Y 不相关。

参考资料与注释

[1] Stieltjes 积分

[2] [LA 7736]Pocky

9.18.5 概率不等式

算法竞赛中有时会用到 随机化算法，这些算法的正确性与时空复杂度通常依赖于「某些随机事件发生的概率很小」这一前提。例如，快速排序的复杂度依赖于「所选的 `pivot` 元素几乎是最小或最大元素」这一事件较少发生。

本文将简要介绍一些用于分析随机化算法的工具并给出几个简单应用的例子。

Union Bound

记 $A_1, \cdots, A_m$ 为随机事件，则

$$P\left\{\bigcup_{i=1}^{m} A_i\right\} \leq \sum_{i=1}^{m} P\{A_i\}$$

即：一组事件中至少一个发生的概率，不超过每一个的发生概率之和。

实际上，这一结论还可以稍作加强：

- 一组事件中至少一者发生的概率，**不小于**每一个的发生概率之和，减掉每两个同时发生的概率之和。
- 一组事件中至少一者发生的概率，**不超过**每一个的发生概率之和，减掉每两个同时发生的概率之和，加上每三个同时发生的概率之和。
- ……

随着层数越来越多，交替出现的上界和下界也越来越紧。这一系列结论形式上类似容斥原理，证明过程也和容斥类似，这里略去。

Markov 不等式

设 X 是一个取值非负的随机变量，则对任意正实数 a 有

$$P\{X \geq a\} \leq \frac{EX}{a}$$

事实上，由于 Markov 不等式本身并没有用到随机变量除期望外的与分布有关的任何信息，因此直接应用这个不等式得到的约束通常很松。

证明

记 I 为事件 $X \geq a$ 的示性函数，则有

$$I \leq \frac{X}{a}$$

进而

$$P\{X \geq a\} = EI \leq E\left[\frac{X}{a}\right] = \frac{EX}{a}$$

Chebyshev 不等式

设 X 是一随机变量，则对任意的 $a > 0$ 都有

$$P\{|X - EX| \geq a\} \leq \frac{DX}{a^2}$$

特别地，当 a 取 $k\sigma$ 时有

$$P\{|X - EX| \geq k\sigma\} \leq \frac{1}{k^2}$$

其中 σ 是 X 的标准差。

证明

由已知，有

$$P\{|X - EX| \geq a\} = P\{(X - EX)^2 \geq a^2\}$$

注意到 $(X - EX)^2$ 非负，故由 Markov 不等式可知

$$P\{(X - EX)^2 \geq a^2\} \leq \frac{E(X - EX)^2}{a^2} = \frac{DX}{a^2}$$

Chernoff 不等式

一般的 Chernoff 不等式可以从直接对随机变量 e^{tX} 应用 Markov 不等式得出：

设 X 是一随机变量，则对任意的 $t>0$ 都有

$$P\{X \geq a\} = P\{\mathrm{e}^{tX} > \mathrm{e}^{ta}\} \leq \frac{E\mathrm{e}^{tX}}{\mathrm{e}^{ta}}$$

类似地，当 $t<0$ 时有

$$P\{X \leq a\} = P\{\mathrm{e}^{tX} > \mathrm{e}^{ta}\} \leq \frac{E\mathrm{e}^{tX}}{\mathrm{e}^{ta}}$$

Poisson 试验之和的 Chernoff 不等式

算法竞赛中涉及的随机变量通常没有那么「一般」，我们可以用概率论中的 Poisson 试验对其进行描述。

所谓 Poisson 试验，是指在只有两种可能结果的随机试验。

一次的 Poisson 试验的结果可以用一个取值为 0 或 1 的随机变量 X 进行刻画，其概率分布为

$$P\{X=i\} = \begin{cases} p_i, & i=1 \\ 1-p_1, & i=0 \end{cases}$$

对于 Poisson 试验，我们有如下结论：

对于 n 个独立的 Poisson 试验 $X_1,X_2,\cdots,X_n$，记 $X=\sum_{i=1}^n X_i$ 以及 $\mu=EX$，则对任意 $0<\epsilon<1$ 有

$$P\left\{|X-\mu| \geq \epsilon\mu\right\} \leq 2\exp\left(-\frac{1}{3}\mu\epsilon^2\right)$$

Hoeffding 不等式

若 $X_1,\cdots,X_n$ 为互相独立的实随机变量且 $X_i\in[a_i,b_i]$，记随机变量 $X=\sum\limits_{i=1}^n X_i$，则

$$P\{|X-EX| \geq \epsilon\} \leq 2\exp\left(\frac{-2\epsilon^2}{\sum\limits_{i=1}^n (b_i-a_i)^2}\right)$$

Chernoff 不等式和 Hoeffding 不等式都限制了随机变量偏离其期望值的程度。这两个不等式的证明过程较为冗长，有兴趣的同学可以查阅 Probability and Computing 一书中的相关章节。

从经验上讲，如果 EX 不太接近 $a_1+\cdots+a_n$，则该不等式给出的界往往相对比较紧；如果非常接近的话（例如在 UOJ #72 全新做法[1] 中），给出的界则往往很松，此时更好的选择是使用 Chernoff 不等式。

应用举例

例：随机撒点估算圆周率

考虑下列估计圆周率 π 的精确值的算法：

在正方形区域 $[-1,1]^2$ 内随机生成 n 个点，记其中落入单位圆盘 $x^2+y^2\leq 1$ 的点数为 m，则可以取 $\dfrac{4m}{n}$ 为 π 的近似值。

问题：若要保证上述算法以至少 $(1-\delta)$ 的概率返回相对误差不超过 ϵ 的结果，n 应该如何取定?

" 解答"

记 X_i 表示事件「随机生成的第 i 个点在单位圆内」，则圆内总点数 $X=\sum_{i=1}^n X_i$。我们需要找到一个合适的 n 使得

$$P\left\{\left|\frac{4X}{n}-\pi\right| \geq \epsilon\pi\right\} \leq \delta$$

上式等价于

$$P\left\{\left|X-\frac{\pi}{4}n\right|\geq\epsilon\cdot\frac{\pi}{4}n\right\}\leq\delta$$

根据 Chernoff 不等式，我们只需令

$$2\exp\left(-\frac{1}{3}\epsilon^2\cdot\frac{\pi}{4}n\right)\leq\delta$$

即可，由此可解得

$$n\geq\frac{12}{\pi}\epsilon^{-2}\ln\frac{2}{\delta}$$

即当 $n=\Omega(\epsilon^{-2}\ln\frac{1}{\delta})$ 时可以达到需要的准确率。

例：抽奖问题

一个箱子里有 n 个球，其中恰有 k 个球对应着大奖。你要进行若干次独立、等概率的随机抽取，每次抽完之后会把球放回箱子。请问抽多少次能保证以至少 $(1-\epsilon)$ 的概率，满足**每一个**奖球都被抽到至少一次？

" 解答"

假如只有一个奖球，则抽取 $M=n\log\epsilon^{-1}$ 次即可保证，因为 M 次全不中的概率

$$\left(1-\frac{1}{n}\right)^{n\log\epsilon^{-1}}\leq e^{\log\epsilon}=\epsilon$$

现在有 $k>1$ 个奖球，那么根据 Union Bound，我们只需保证每个奖球被漏掉的概率都不超过 $\frac{\epsilon}{k}$ 即可。于是答案是 $n\log\frac{k}{\epsilon}$。

例：随机选取一半元素

给出一个算法，从 n 个元素中等概率随机选取一个大小为 $\frac{n}{2}$ 的子集，保证 n 是偶数。你能使用的唯一的随机源是一枚均匀硬币，同时请你尽量减少抛硬币的次数（不要求最少）。

" 解法"

首先可以想到这样的算法：

- 通过抛 n 次硬币，可以从所有子集中等概率随机选一个。
- 不断重复这一过程，直到选出的子集大小恰好为 $\frac{n}{2}$。
  - 注意到大小为 $\frac{n}{2}$ 的子集至少占所有子集的 $\frac{1}{n}$，因此重复次数的期望值 $\leq n$。

这一算法期望需要抛 n^2 次硬币。

另一个算法：

- 我们可以通过抛期望 $2\lceil\log_2 n\rceil$ 次硬币来实现随机 n 选 1。
  - 具体方法：随机生成 $\lceil\log_2 n\rceil$ 位的二进制数，如果大于等于 n 则重新随机，否则选择对应编号（编号从 0 开始）的元素并结束过程。
- 然后我们从所有元素中选一个，再从剩下的元素中再选一个，以此类推，直到选出 $\frac{n}{2}$ 个元素为止。

这一算法期望需要抛 $n\lceil\log_2 n\rceil$ 次硬币。

将两个算法缝合起来：

- 先用第一个算法随机得到一个子集。
- 如果该子集大小不到 $\frac{n}{2}$，则利用第二个算法不断添加元素，直到将大小补到 $\frac{n}{2}$。

- 如果该子集大小超过 $\frac{n}{2}$，则利用第二个算法不断删除元素，直到将大小削到 $\frac{n}{2}$。

尝试分析第二、第三步所需的操作次数（即添加/删除元素的次数）：

- 记 01 随机变量 X_i 表示 i 是否被选入初始的子集，令 $X := X_1 + \cdots + X_n$ 表示子集大小，则第二、第三步所需的操作次数等于 $\left|X - \mathrm{E}[X]\right|$。在 Hoeffding 不等式中取 $t = c \cdot \sqrt{n}$（其中 c 为任意常数），得到 $\Pr\left[\left|X - \mathrm{E}[X]\right| \geq t\right] \leq 2\mathrm{e}^{-c^2}$。也就是说，我们可以通过允许 $\Theta(\sqrt{n})$ 级别的偏移，来得到任意小的常数级别的失败概率。

至此我们已经说明：该算法可以以很大概率保证抛硬币次数在 $n + \Theta(\sqrt{n}\log n)$ 以内。

- 其中 n 来自获得初始子集的抛硬币次数；$\Theta(\sqrt{n}\log n)$ 是 $\Theta(\sqrt{n})$ 次添加/删除元素的总开销。

"计算期望复杂度"

我们再从另一个角度分析，尝试计算该算法的期望抛硬币次数。

用 Hoeffding 不等式求第二、第三步中操作次数期望值的上界：

$$E|X - EX| = \int_0^\infty P\{|X - E[X]| \geq t\}\mathrm{d}t \leq 2\int_0^\infty \exp\left(-\frac{t^2}{n}\right)\mathrm{d}t = \sqrt{\pi n}$$

从而第二、第三步所需抛硬币次数的期望值是 $\sqrt{\pi n} \cdot 2\lceil \log_2 n \rceil$。

综上，该算法期望需要抛 $n + 2\sqrt{\pi n}\lceil \log_2 n \rceil$ 次硬币。

练习：Balls and Bins

n 个球独立随机地扔到 n 个盒子里，试证明：球最多的盒子中的球数以 $1 - \frac{1}{n}$ 的概率不少于 $\Omega\left(\frac{\log n}{\log\log n}\right)$。

参考资料与注释

[1] UOJ #72 全新做法

9.19 博弈论

9.19.1 博弈论简介

Authors: cutekibry, woruo27, Backl1ght

博弈论，是经济学的一个分支，主要研究具有竞争或对抗性质的对象，在一定规则下产生的各种行为。博弈论考虑游戏中的个体的预测行为和实际行为，并研究它们的优化策略。

通俗地讲，博弈论主要研究的是：在一个游戏中，进行游戏的多位玩家的策略。

公平组合游戏

公平组合游戏（Impartial Game）的定义如下：

- 游戏有两个人参与，二者轮流做出决策，双方均知道游戏的完整信息；
- 任意一个游戏者在某一确定状态可以作出的决策集合只与当前的状态有关，而与游戏者无关；
- 游戏中的同一个状态不可能多次抵达，游戏以玩家无法行动为结束，且游戏一定会在有限步后以非平局结束。

非公平组合游戏

非公平组合游戏（Partizan Game）与公平组合游戏的区别在于在非公平组合游戏中，游戏者在某一确定状态可以做出的决策集合与游戏者有关。大部分的棋类游戏都**不是**公平组合游戏，如国际象棋、中国象棋、围棋、五子棋等（因为双方都不能使用对方的棋子）。

反常游戏

反常游戏（Misère Game）按照传统的游戏规则进行游戏，但是其胜者为第一个无法行动的玩家。以 Nim 游戏为例，Nim 游戏中取走最后一颗石子的为胜者，而反常 Nim 游戏中取走最后一刻石子的为败者。

参考资料

- Wikipedia - Misère[1]

参考资料与注释

[1] Wikipedia - Misère

9.19.2 公平组合游戏

Authors: cutekibry, woruo27, tinjyu

经典的公平组合游戏有很多，包括取数游戏，31 点，以及 Nim 游戏等。

Nim 游戏

n 堆物品，每堆有 a_i 个，两个玩家轮流取走任意一堆的任意个物品，但不能不取。

取走最后一个物品的人获胜。

例如，如果现在有 $n=3$ 堆物品，而每堆分别有 $2,5,4$ 个，那么可以取走第 1 堆中的 2 个物品，局面就变成了 $0,5,4$；或者也可以取走第 2 堆的 4 个物品，局面就变成了 $2,1,4$。

如果现在的局面为 $0,0,5$，甲取走了第 3 堆的 5 个物品，也就是取走了最后一个物品，此时甲获胜。

博弈图和状态

如果将每个状态视为一个节点，再从每个状态向它的后继状态连边，我们就可以得到一个博弈状态图。

例如，如果节点 (i,j,k) 表示局面为 i,j,k 时的状态，则我们可以画出下面的博弈图（由于篇幅有限，故仅显示部分状态节点和部分边）：

定义**必胜状态**为**先手必胜的状态**，**必败状态**为**先手必败的状态**。

通过推理，我们可以得出下面三条定理：

- 定理 1：没有后继状态的状态是必败状态。
- 定理 2：一个状态是必胜状态当且仅当存在至少一个必败状态为它的后继状态。
- 定理 3：一个状态是必败状态当且仅当它的所有后继状态均为必胜状态。

对于定理 1，如果游戏进行不下去了，那么这个玩家就输掉了游戏。

对于定理 2，如果该状态至少有一个后继状态为必败状态，那么玩家可以通过操作到该必败状态；此时对手的状态为必败状态——对手必定是失败的，而相反地，自己就获得了胜利。

对于定理 3，如果不存在一个后继状态为必败状态，那么无论如何，玩家只能操作到必胜状态；此时对手的状态为必胜状态——对手必定是胜利的，自己就输掉了游戏。

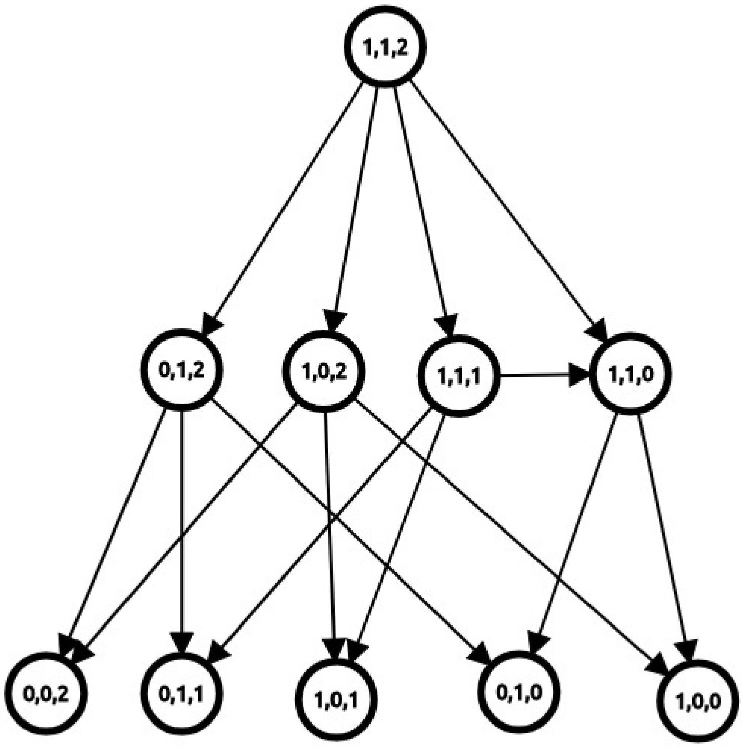


图 9.37 博弈图的例子

如果博弈图是一个有向无环图，则通过这三个定理，我们可以在绘出博弈图的情况下用 $O(N+M)$ 的时间（其中 N 为状态种数，M 为边数）得出每个状态是必胜状态还是必败状态。

Nim 和

让我们再次回顾 Nim 游戏。

通过绘画博弈图，我们可以在 $O(\prod_{i=1}^{n} a_i)$ 的时间里求出该局面是否先手必赢。

但是，这样的时间复杂度实在太高。有没有什么巧妙而快速的方法呢？

定义 Nim 和 $= a_1 \oplus a_2 \oplus ... \oplus a_n$。

当且仅当 Nim 和为 0 时，该状态为必败状态；否则该状态为必胜状态。

证明

为什么异或值会和状态的胜负有关？下面给出了这个定理的证明过程。

为了证明该定理，只需要证明下面三个定理：

- 定理 1：没有后继状态的状态是必败状态。
- 定理 2：对于 $a_1 \oplus a_2 \oplus ... \oplus a_n \neq 0$ 的局面，一定存在某种移动使得 $a_1 \oplus a_2 \oplus ... \oplus a_n = 0$。
- 定理 3：对于 $a_1 \oplus a_2 \oplus ... \oplus a_n = 0$ 的局面，一定不存在某种移动使得 $a_1 \oplus a_2 \oplus ... \oplus a_n = 0$。

对于定理 1，没有后继状态的状态只有一个，即全 0 局面。此时 $a_1 \oplus a_2 \oplus ... \oplus a_n = 0$。

对于定理 2，不妨假设 $a_1 \oplus a_2 \oplus ... a_n = k \neq 0$。如果我们要将 a_i 改为 a_i'，则 $a_i' = a_i \oplus k$。

假设 k 的二进制最高位 1 为 d，即 $2^d \leq k < 2^{d+1}$。根据异或定义，一定有奇数个 a_i 的二进制第 d 位为 1。满足这个条件的 a_i 一定也满足 $a_i > a_i \oplus k$，因而这也是个合法的移动。

对于定理 3，如果我们要将 a_i 改为 a_i'，则根据异或运算律可以得出 $a_i = a_i'$，因而这不是个合法的移动。

有向图游戏与 SG 函数

有向图游戏是一个经典的博弈游戏——实际上，大部分的公平组合游戏都可以转换为有向图游戏。

在一个有向无环图中，只有一个起点，上面有一个棋子，两个玩家轮流沿着有向边推动棋子，不能走的玩家判负。

定义 mex 函数的值为不属于集合 S 中的最小非负整数，即：

$$\operatorname{mex}(S) = \min\{x\} \quad (x \notin S, x \in N)$$

例如 $\operatorname{mex}(\{0,2,4\}) = 1$，$\operatorname{mex}(\{1,2\}) = 0$。

对于状态 x 和它的所有 k 个后继状态 $y_1, y_2, \ldots, y_k$，定义 SG 函数：

$$\operatorname{SG}(x) = \operatorname{mex}\{\operatorname{SG}(y_1), \operatorname{SG}(y_2), \ldots, \operatorname{SG}(y_k)\}$$

而对于由 n 个有向图游戏组成的组合游戏，设它们的起点分别为 $s_1, s_2, \ldots, s_n$，则有定理：**当且仅当** $\operatorname{SG}(s_1) \oplus \operatorname{SG}(s_2) \oplus \ldots \oplus \operatorname{SG}(s_n) \neq 0$ **时，这个游戏是先手必胜的。同时，这是这一个组合游戏的游戏状态** x **的 SG 值**。

这一定理被称作 **Sprague–Grundy 定理** (Sprague–Grundy Theorem), 简称 SG 定理。

SG 定理的证明

可以使用数学归纳法来证明。

我们假设对于游戏状态 x'，其当前节点 $s_1', s_2', \ldots, s_n'$（对于任意 i 有 $s_i' < s_i$），皆满足 SG 定理。

显然当 $\operatorname{SG}(s_1)' = \operatorname{SG}(s_2)' = \ldots \operatorname{SG}(s_n)' = 0$ 时，该状态能满足 SG 定理。

那么只需要证明对于游戏状态 x，其当前节点 $s_1', s_2', \ldots, s_n'$ 符合 SG 定理，SG 定理便成立。

事实上这一个状态可以看作一个 Nim 游戏，对于某个节点 s_i，它可以移动到任意一个 SG 值比它小或比它大的节点。

在有向图游戏中，当一方将某一节点 s_i 移动到 SG 值比它大的节点时，另一方可以移动回和 SG 值和 $\operatorname{SG}(s_i)$ 一样的节点，所以向 SG 值较大节点移动是无效操作。

当移动到 SG 值较小的节点时，情况则会和 Nim 游戏一样，能够到达任何一个游戏状态 x' 使得 $\operatorname{SG}(x') = \operatorname{SG}(s_1') \oplus \operatorname{SG}(s_2') \oplus \ldots \oplus \operatorname{SG}(s_n') < \operatorname{SG}(X)$（注意到前文已经假设 x' 满足 SG 定理），但到达不了 SG 值为 $\operatorname{SG}(s_1) \oplus \operatorname{SG}(s_2) \oplus \ldots \oplus \operatorname{SG}(s_n)$ 的节点。

所以状态 x 符合 SG 定理。

SG 定理的应用

SG 定理适用于**任何公平的两人游戏**, 它常被用于决定游戏的输赢结果。

计算给定状态的 Grundy 值的步骤一般包括：

- 获取从此状态所有可能的转换；
- 每个转换都可以导致**一系列独立的博弈**（退化情况下只有一个）。计算每个独立博弈的 Grundy 值并对它们进行**异或求和**。
- 在为每个转换计算了 Grundy 值之后，状态的值是这些数字的 mex。
- 如果该值为零，则当前状态为输，否则为赢。

将 Nim 游戏转换为有向图游戏

我们可以将一个有 x 个物品的堆视为节点 x，则当且仅当 $y < x$ 时，节点 x 可以到达 y。

那么，由 n 个堆组成的 Nim 游戏，就可以视为 n 个有向图游戏了。

根据上面的推论，可以得出 $\operatorname{SG}(x) = x$。再根据 SG 定理，就可以得出 Nim 和的结论了。

参考文献

（ 转载 ）Nim 游戏博弈 （ 收集完全版 ）- exponent - 博客园[1]

[组合游戏与博弈论]【 学习笔记 】- Candy? - 博客园[2]

参考资料与注释

[1] （转载）Nim 游戏博弈（收集完全版）- exponent - 博客园

[2] [组合游戏与博弈论]【学习笔记】- Candy? - 博客园

9.19.3 非公平组合游戏

9.19.4 反常游戏

Authors: 2008verser

反常游戏、反 Nim 游戏的 定义。

以反 Nim 游戏为例，这里给出反 Nim 游戏的结论以及证明：

规定：字母 N 和 P 分别代表先手必胜与必败。

一个局面为 N 态的充要条件是有至少一条出边连接至 P 态。

一个局面为 P 态的充要条件是每一条出边都连接到 N 态。

反 Nim 游戏

为方便书写，用字母 T 表示 $\oplus_{i=1}^{n} a_i$。

结论：

1. 当全部 $a_i = 1$，如果有奇数堆石子就为 P 态，有偶数堆则为 N 态。
2. 当至少一个 $a_i > 1$，$T \neq 0$ 时为 N 态，否则为 P 态。

证明 1：显然。

证明 2：

情况 A：若只有一个$a_i > 1$ **（此时 T 一定 $\neq 0$）**，则先手选择转移到全部 $a_i = 1$ 的局面，并且先手可以在这次决策中控制转移后堆数的奇偶。

故这种情况**是 N 态**。

情况 B：（不考虑 T 取值）有至少 2 个 $a_i > 1$。

小情况 B1：$T \neq 0$：通过 Nim 游戏可知一定能够转移到 $T = 0$ 的局面（小情况 B2）。

小情况 B2：$T = 0$：

一方面可以转移到至少 2 个 $a_i > 1, T \neq 0$ 的局面，即情况 B1。

另一方面随着游戏进行（B1, B2 循环），数量大于 1 的堆会逐渐减少，最终只剩一堆，这就变成了情况 A，为 N 态。

观察情况 B，小情况 B2 能给对面 N 态或至少 2 个 $a_i > 1, T \neq 0$ 的局面，而小情况 B1 仅能给对面 $T = 0$ 的局面。

所以在情况 B 下，小情况 B2 为 N 态，B1 为 P 态。

也就是说**当至少 2 个 $a_i > 1, T \neq 0$ 时为 N 态，否则为 P 态**。

综合情况 A 和情况 B 的结论，结论 2 得证。

综上，结论 1 和 2 皆得证。结论得证。

9.20 数值算法

9.20.1 插值

Authors: AtomAlpaca, billchenchina, Chrogeek, Early0v0, EndlessCheng, Enter-tainer, Henry-ZHR, hly1204, hsfzLZH1, Ir1d, Ghastlcon, kenlig, Marcythm, megakite, Peanut-Tang, qwqAutomaton, qz-cqy, StudyingFather, swift-zym, swiftqwq, Tiphereth-A, TrisolarisHD, x4Cx58x54, Xeonacid, xiaopangfeiyu, YanWQ-monad

引入

插值是一种通过已知的、离散的数据点推算一定范围内的新数据点的方法。插值法常用于函数拟合中。

例如对数据点：

x	0	1	2	3	4	5	6
$f(x)$	0	0.8415	0.9093	0.1411	-0.7568	-0.9589	-0.2794

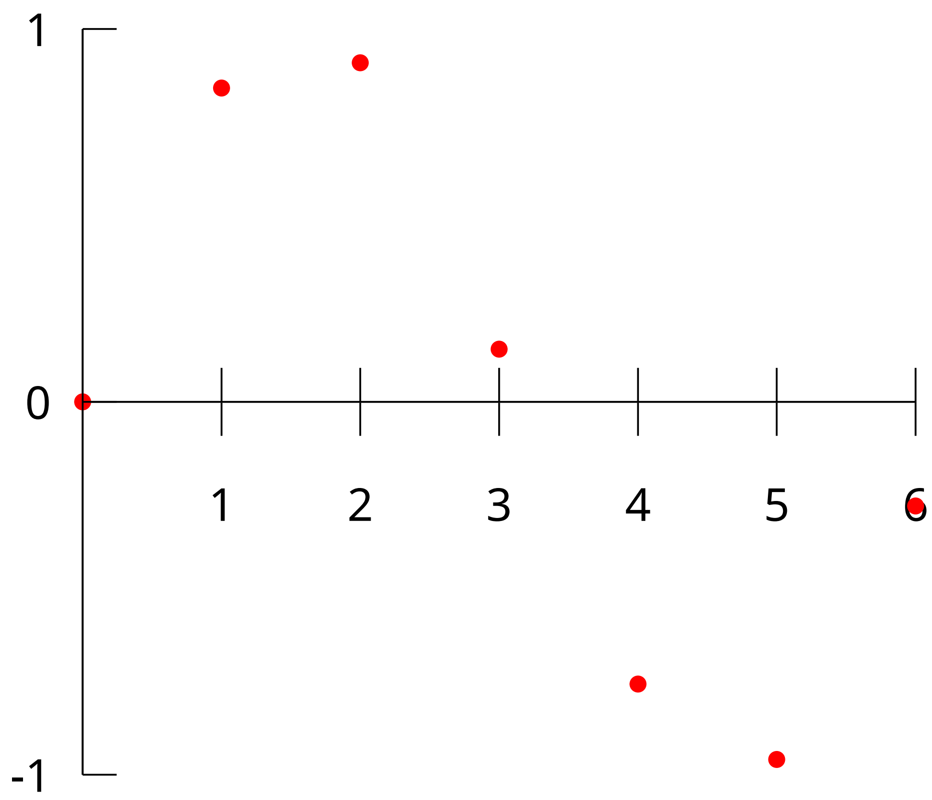


图 9.38

其中 $f(x)$ 未知，插值法可以通过按一定形式拟合 $f(x)$ 的方式估算未知的数据点。

例如，我们可以用分段线性函数拟合 $f(x)$：

这种插值方式叫做 线性插值[1]。

我们也可以用多项式拟合 $f(x)$：

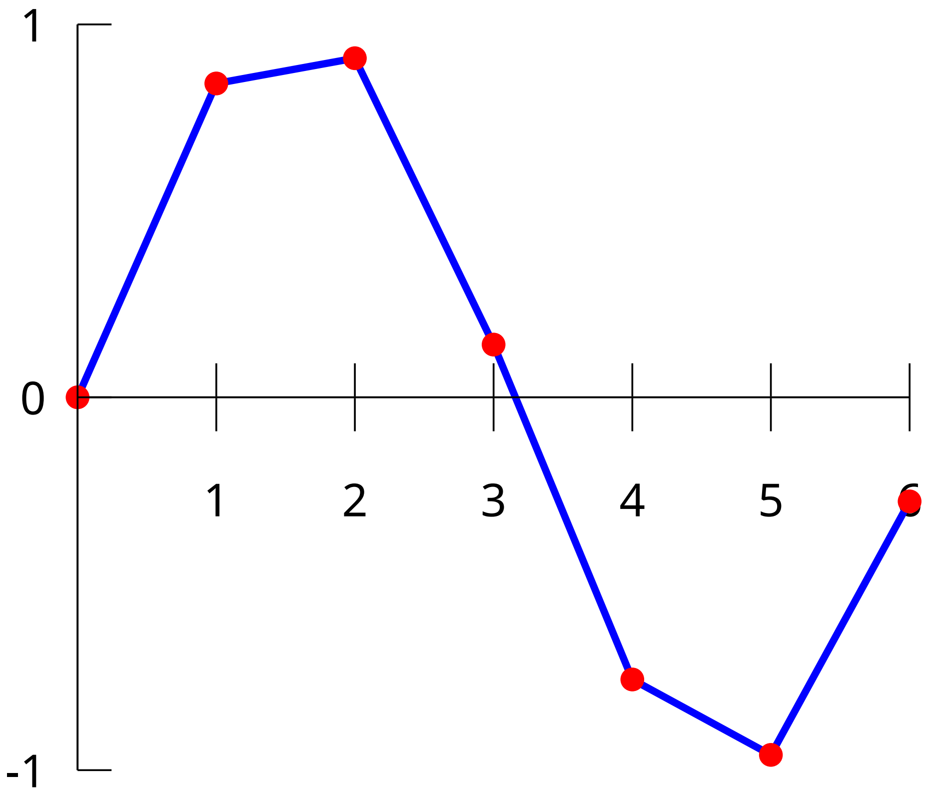


图 9.39

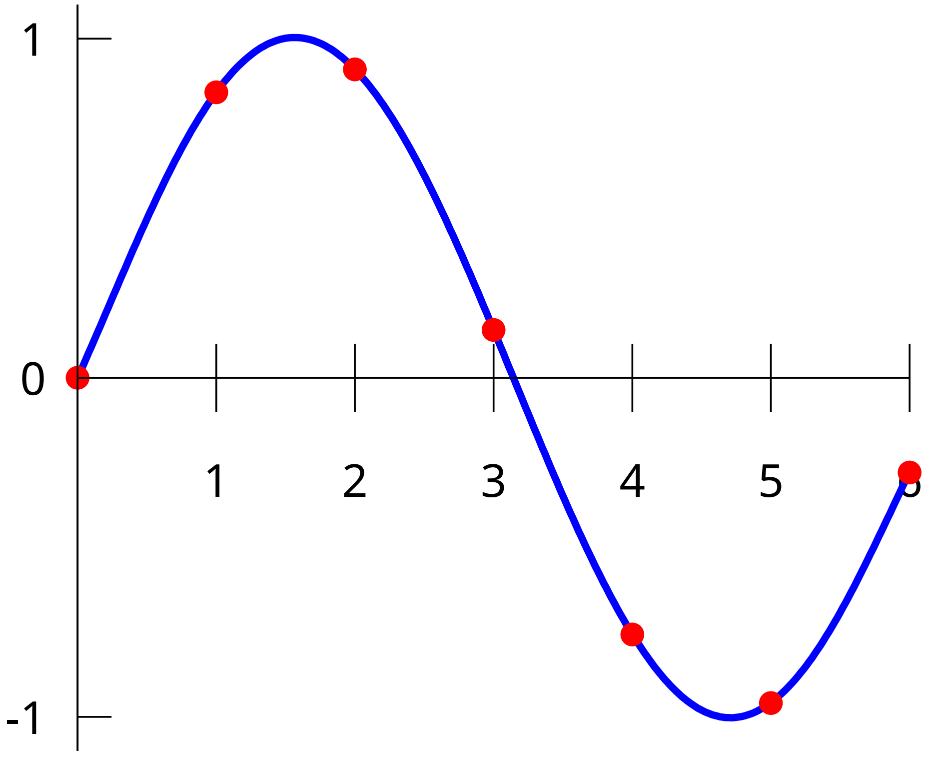


图 9.40

这种插值方式叫做 多项式插值[2]。

多项式插值的一般形式如下：

" 多项式插值"

对已知的 $n+1$ 的点 $(x_0,y_0),(x_1,y_1),\dots,(x_n,y_n)$，求 n 阶多项式 $f(x)$ 满足

$$f(x_i)=y_i,\qquad \forall i=0,1,\dots,n$$

下面介绍多项式插值中的两种方式：Lagrange 插值法与 Newton 插值法。不难证明这两种方法得到的结果是相等的。

Lagrange 插值法

由于要求构造一个函数 $f(x)$ 过点 $P_1(x_1,y_1),P_2(x_2,y_2),\cdots,P_n(x_n,y_n)$. 首先设第 i 个点在 x 轴上的投影为 $P_i'(x_i,0)$.

考虑构造 n 个函数 $f_1(x),f_2(x),\cdots,f_n(x)$，使得对于第 i 个函数 $f_i(x)$，其图像过 $\begin{cases}P_j'(x_j,0),(j\neq i)\\P_i(x_i,y_i)\end{cases}$，则可知题目所求的函数 $f(x)=\sum\limits_{i=1}^{n}f_i(x)$.

那么可以设 $f_i(x)=a\cdot\prod_{j\neq i}(x-x_j)$，将点 $P_i(x_i,y_i)$ 代入可以知道 $a=\dfrac{y_i}{\prod_{j\neq i}(x_i-x_j)}$，所以

$$f_i(x)=y_i\cdot\frac{\prod_{j\neq i}(x-x_j)}{\prod_{j\neq i}(x_i-x_j)}=y_i\cdot\prod_{j\neq i}\frac{x-x_j}{x_i-x_j}$$

那么我们就可以得出 Lagrange 插值的形式为：

$$f(x)=\sum_{i=1}^{n}y_i\cdot\prod_{j\neq i}\frac{x-x_j}{x_i-x_j}$$

朴素实现的时间复杂度为 $O(n^2)$，可以优化到 $O(n\log^2 n)$，参见 多项式快速插值。

"Luogu P4781【模板】拉格朗日插值[3]"

给出 n 个点对 (x_i,y_i) 和 k，且 $\forall i,j$ 有 $i\neq j\iff x_i\neq x_j$ 且 $f(x_i)\equiv y_i \pmod{998244353}$ 和 $\deg(f(x))<n$（定义 $\deg(0)=-\infty$），求 $f(k)\bmod 998244353$.

" 题解"

本题中只用求出 $f(k)$ 的值，所以在计算上式的过程中直接将 k 代入即可。

$$f(k)=\sum_{i=1}^{n}y_i\prod_{j\neq i}\frac{k-x_j}{x_i-x_j}$$

本题中，还需要求解逆元。如果先分别计算出分子和分母，再将分子乘进分母的逆元，累加进最后的答案，时间复杂度的瓶颈就不会在求逆元上，时间复杂度为 $O(n^2)$.

因为在固定模 998244353 意义下运算，计算乘法逆元的时间复杂度我们在这里暂且认为是常数时间。

" 代码实现"

```
#include <exception>
#include <iostream>
#include <optional>
#include <tuple>
```

```cpp
#include <utility>
#include <vector>

template <unsigned int Mod>
class Fp {
  static_assert(static_cast<int>(Mod) > 1);

 public:
  Fp() : v_() {}

  Fp(int v) : v_(safe_mod(v)) {}

  static unsigned int safe_mod(int v) {
    v %= static_cast<int>(Mod);
    return v < 0 ? v + static_cast<int>(Mod) : v;
  }

  unsigned int value() const { return v_; }

  Fp operator-() const { return Fp(Mod - v_); }

  Fp pow(int e) const {
    if (e < 0) return inv().pow(-e);
    for (Fp x(*this), res(1);; x *= x) {
      if (e & 1) res *= x;
      if ((e >>= 1) == 0) return res;
    }
  }

  Fp inv() const {
    int x1 = 1, x3 = 0, a = v_, b = Mod;
    while (b != 0) {
      int q = a / b, x1_old = x1, a_old = a;
      x1 = x3, x3 = x1_old - x3 * q, a = b, b = a_old - b * q;
    }
    return Fp(x1);
  }

  Fp &operator+=(const Fp &rhs) {
    if ((v_ += rhs.v_) >= Mod) v_ -= Mod;
    return *this;
  }

  Fp &operator-=(const Fp &rhs) {
    if ((v_ += Mod - rhs.v_) >= Mod) v_ -= Mod;
    return *this;
  }

  Fp &operator*=(const Fp &rhs) {
    v_ = static_cast<unsigned long long>(v_) * rhs.v_ % Mod;
    return *this;
  }

  Fp &operator/=(const Fp &rhs) { return operator*=(rhs.inv()); }
```

```
  void swap(Fp &rhs) {
    unsigned int v = v_;
    v_ = rhs.v_, rhs.v_ = v;
  }

  friend Fp operator+(const Fp &lhs, const Fp &rhs) { return Fp(lhs) += rhs; }

  friend Fp operator-(const Fp &lhs, const Fp &rhs) { return Fp(lhs) -= rhs; }

  friend Fp operator*(const Fp &lhs, const Fp &rhs) { return Fp(lhs) *= rhs; }

  friend Fp operator/(const Fp &lhs, const Fp &rhs) { return Fp(lhs) /= rhs; }

  friend bool operator==(const Fp &lhs, const Fp &rhs) {
    return lhs.v_ == rhs.v_;
  }

  friend bool operator!=(const Fp &lhs, const Fp &rhs) {
    return lhs.v_ != rhs.v_;
  }

  friend std::istream &operator>>(std::istream &lhs, Fp &rhs) {
    int v;
    lhs >> v;
    rhs = Fp(v);
    return lhs;
  }

  friend std::ostream &operator<<(std::ostream &lhs, const Fp &rhs) {
    return lhs << rhs.v_;
  }

 private:
  unsigned int v_;
};

template <typename T>
class Poly : public std::vector<T> {
 public:
  using std::vector<T>::vector;  // 使用继承的构造函数

  bool is_zero() const { return deg() == -1; }

  void shrink() { this->resize(std::max(deg() + 1, 1)); }

  int deg()
      const {  // 多项式的次数，当多项式为零时度数为 -1 而不是一般定义的负无穷
    int d = static_cast<int>(this->size()) - 1;
    const T z;
    while (d >= 0 && this->operator[](d) == z) --d;
    return d;
  }
```

```
  T leading_coeff() const {
    int d = deg();
    return d == -1 ? T() : this->operator[](d);
  }

  Poly operator-() const {
    Poly res;
    res.reserve(this->size());
    for (auto &&i : *this) res.emplace_back(-i);
    res.shrink();
    return res;
  }

  Poly &operator+=(const Poly &rhs) {
    if (this->size() < rhs.size()) this->resize(rhs.size());
    for (int i = 0, e = static_cast<int>(rhs.size()); i != e; ++i)
      this->operator[](i) += rhs[i];
    shrink();
    return *this;
  }

  Poly &operator-=(const Poly &rhs) {
    if (this->size() < rhs.size()) this->resize(rhs.size());
    for (int i = 0, e = static_cast<int>(rhs.size()); i != e; ++i)
      this->operator[](i) -= rhs[i];
    shrink();
    return *this;
  }

  Poly &operator*=(const Poly &rhs) {
    int n = deg(), m = rhs.deg();
    if (n == -1 || m == -1) return operator=(Poly{0});
    Poly res(n + m + 1);
    for (int i = 0; i <= n; ++i)
      for (int j = 0; j <= m; ++j) res[i + j] += this->operator[](i) * rhs[j];
    return operator=(res);
  }

  Poly &operator/=(const Poly &rhs) {
    int n = deg(), m = rhs.deg(), q = n - m;
    if (m == -1) throw std::runtime_error("Division by zero");
    if (q <= -1) return operator=(Poly{0});
    Poly res(q + 1);
    const T iv = 1 / rhs.leading_coeff();
    for (int i = q; i >= 0; --i)
      if ((res[i] = this->operator[](n--) * iv) != T())
        for (int j = 0; j != m; ++j) this->operator[](i + j) -= res[i] * rhs[j];
    return operator=(res);
  }

  Poly &operator%=(const Poly &rhs) {
    int n = deg(), m = rhs.deg(), q = n - m;
    if (m == -1) throw std::runtime_error("Division by zero");
    const T iv = 1 / rhs.leading_coeff();
```

```
    for (int i = q; i >= 0; --i)
      if (T res = this->operator[](n--) * iv; res != T())
        for (int j = 0; j <= m; ++j) this->operator[](i + j) -= res * rhs[j];
    shrink();
    return *this;
  }

  std::pair<Poly, Poly> div_mod(const Poly &rhs) const {
    int n = deg(), m = rhs.deg(), q = n - m;
    if (m == -1) throw std::runtime_error("Division by zero");
    if (q <= -1) return std::make_pair(Poly{0}, Poly(*this));
    const T iv = 1 / rhs.leading_coeff();
    Poly quo(q + 1), rem(*this);
    for (int i = q; i >= 0; --i)
      if ((quo[i] = rem[n--] * iv) != T())
        for (int j = 0; j <= m; ++j) rem[i + j] -= quo[i] * rhs[j];
    rem.shrink();
    return std::make_pair(quo, rem);  // (quotient, remainder)
  }

  T eval(const T &pt) const {
    T res;
    for (int i = deg(); i >= 0; --i) res = res * pt + this->operator[](i);
    return res;
  }

  friend Poly operator+(const Poly &lhs, const Poly &rhs) {
    return Poly(lhs) += rhs;
  }

  friend Poly operator-(const Poly &lhs, const Poly &rhs) {
    return Poly(lhs) -= rhs;
  }

  friend Poly operator*(const Poly &lhs, const Poly &rhs) {
    return Poly(lhs) *= rhs;
  }

  friend Poly operator/(const Poly &lhs, const Poly &rhs) {
    return Poly(lhs) /= rhs;
  }

  friend Poly operator%(const Poly &lhs, const Poly &rhs) {
    return Poly(lhs) %= rhs;
  }

  friend bool operator==(const Poly &lhs, const Poly &rhs) {
    int d = lhs.deg();
    if (d != rhs.deg()) return false;
    for (; d >= 0; --d)
      if (lhs[d] != rhs[d]) return false;
    return true;
  }
```

```cpp
  friend bool operator!=(const Poly &lhs, const Poly &rhs) {
    return !(lhs == rhs);
  }

  friend std::ostream &operator<<(std::ostream &lhs, const Poly &rhs) {
    int s = 0, e = static_cast<int>(rhs.size());
    lhs << '[';
    for (auto &&i : rhs) {
      lhs << i;
      if (s >= 1) lhs << 'x';
      if (s > 1) lhs << '^' << s;
      if (++s != e) lhs << " + ";
    }
    return lhs << ']';
  }
};

template <typename T>
Poly<T> lagrange_interpolation(const std::vector<T> &x,
                               const std::vector<T> &y) {
  if (x.size() != y.size()) throw std::runtime_error("x.size() != y.size()");
  const int n = static_cast<int>(x.size());
  Poly<T> M = {T(1)}, f;
  for (int i = 0; i != n; ++i) M *= Poly<T>{-x[i], T(1)};
  for (int i = 0; i != n; ++i) {
    auto m = M / Poly<T>{-x[i], T(1)};
    f += Poly<T>{y[i] / m.eval(x[i])} * m;
  }
  return f;
}

int main() {
  std::ios::sync_with_stdio(false);
  std::cin.tie(nullptr);
  using Z = Fp<998244353>;
  int n;
  Z k;
  std::cin >> n >> k;
  std::vector<Z> x(n), y(n);
  for (int i = 0; i != n; ++i) std::cin >> x[i] >> y[i];
  std::cout << lagrange_interpolation(x, y).eval(k) << std::endl;
  return 0;
}
```

横坐标是连续整数的 Lagrange 插值

如果已知点的横坐标是连续整数，我们可以做到 $O(n)$ 插值。

设要求 n 次多项式为 $f(x)$，我们已知 $f(1),\cdots,f(n+1)$（$1\leq i\leq n+1$），考虑代入上面的插值公式：

$$
\begin{aligned}
f(x)&=\sum_{i=1}^{n+1}y_i\prod_{j\neq i}\frac{x-x_j}{x_i-x_j}\\
&=\sum_{i=1}^{n+1}y_i\prod_{j\neq i}\frac{x-j}{i-j}
\end{aligned}
$$

后面的累乘可以分子分母分别考虑，不难得到分子为：

$$\frac{\prod_{j=1}^{n+1}(x-j)}{x-i}$$

分母的 $i-j$ 累乘可以拆成两段阶乘来算：

$$(-1)^{n+1-i}\cdot(i-1)!\cdot(n+1-i)!$$

于是横坐标为 $1,\cdots,n+1$ 的插值公式：

$$f(x)=\sum_{i=1}^{n+1}(-1)^{n+1-i}y_i\cdot\frac{\prod_{j=1}^{n+1}(x-j)}{(i-1)!(n+1-i)!(x-i)}$$

预处理 $(x-i)$ 前后缀积、阶乘阶乘逆，然后代入这个式子，复杂度为 $O(n)$.

" **例题** CF622F The Sum of the k-th Powers[4]"

给出 n,k，求 $\sum_{i=1}^{n} i^k$ 对 10^9+7 取模的值。

```
??? note " 题解"
    本题中，答案是一个 $k+1$ 次多项式，因此我们可以线性筛出 $1^i,\cdots,(k+2)^i$ 的
值然后进行 $O(n)$ 插值。

    也可以通过组合数学相关知识由差分法的公式推得下式：

    $$
    f(x)=\sum_{i=1}^{n+1}\binom{x-1}{i-1}\sum_{j=1}^{i}(-1)^{i+j}\binom{i-1}{j-1
}y_{j}=\sum\limits_{i=1}^{n+1}y_i\cdot\frac{\prod\limits_{j=1}^{n+1}(x-j)}{(x-i)
\cdot(-1)^{n+1-i}\cdot(i-1)!\cdot(n+1-i)!}
    $$

??? note " 代码实现"
    ```cpp
    // By: Luogu@rui_er(122461)
    #include <bits/stdc++.h>
    using namespace std;
    const int N = 1e6 + 5, mod = 1e9 + 7;

    int n, k, tab[N], p[N], pcnt, f[N], pre[N], suf[N], fac[N], inv[N], ans;

    int qpow(int x, int y) {
      int ans = 1;
      for (; y; y >>= 1, x = 1LL * x * x % mod)
        if (y & 1) ans = 1LL * ans * x % mod;
      return ans;
    }

    void sieve(int lim) {
      f[1] = 1;
      for (int i = 2; i <= lim; i++) {
        if (!tab[i]) {
          p[++pcnt] = i;
          f[i] = qpow(i, k);
```
```

```
    }
    for (int j = 1; j <= pcnt && 1LL * i * p[j] <= lim; j++) {
      tab[i * p[j]] = 1;
      f[i * p[j]] = 1LL * f[i] * f[p[j]] % mod;
      if (!(i % p[j])) break;
    }
  }
  for (int i = 2; i <= lim; i++) f[i] = (f[i - 1] + f[i]) % mod;
}

int main() {
  scanf("%d%d", &n, &k);
  sieve(k + 2);
  if (n <= k + 2) return printf("%d\n", f[n]) & 0;
  pre[0] = suf[k + 3] = 1;
  for (int i = 1; i <= k + 2; i++) pre[i] = 1LL * pre[i - 1] * (n - i) % mod
;
  for (int i = k + 2; i >= 1; i--) suf[i] = 1LL * suf[i + 1] * (n - i) % mod
;
  fac[0] = inv[0] = fac[1] = inv[1] = 1;
  for (int i = 2; i <= k + 2; i++) {
    fac[i] = 1LL * fac[i - 1] * i % mod;
    inv[i] = 1LL * (mod - mod / i) * inv[mod % i] % mod;
  }
  for (int i = 2; i <= k + 2; i++) inv[i] = 1LL * inv[i - 1] * inv[i] % mod;
  for (int i = 1; i <= k + 2; i++) {
    int P = 1LL * pre[i - 1] * suf[i + 1] % mod;
    int Q = 1LL * inv[i - 1] * inv[k + 2 - i] % mod;
    int mul = ((k + 2 - i) & 1) ? -1 : 1;
    ans = (ans + 1LL * (Q * mul + mod) % mod * P % mod * f[i] % mod) % mod;
  }
  printf("%d\n", ans);
  return 0;
}

```

Newton 插值法

Newton 插值法是基于高阶差分来插值的方法，优点是支持 $O(n)$ 插入新数据点。

为了实现 $O(n)$ 插入新数据点，我们令：

$$f(x)=\sum_{j=0}^{n}a_jn_j(x)$$

其中 $n_j(x):=\prod_{i=0}^{j-1}(x-x_i)$ 称为 **Newton 基**（Newton basis）。

若解出 a_j，则可得到 $f(x)$ 的插值多项式。我们按如下方式定义**前向差商**（forward divided differences）：

$$\begin{aligned}[y_k] &:= y_k, & k=0,\dots,n,\\ [y_k,\dots,y_{k+j}] &:= \frac{[y_{k+1},\dots,y_{k+j}]-[y_k,\dots,y_{k+j-1}]}{x_{k+j}-x_k}, & k=0,\dots,n-j,\ j=1,\dots,n.\end{aligned}$$

则：

$$\begin{aligned}f(x)&=[y_0]+[y_0,y_1](x-x_0)+\cdots+[y_0,\dots,y_n](x-x_0)\dots(x-x_{n-1})\\ &=\sum_{j=0}^{n}[y_0,\dots,y_j]n_j(x)\end{aligned}$$

此即 Newton 插值的形式。朴素实现的时间复杂度为 $O(n^2)$.

若样本点是等距的（即 $x_i = x_0 + ih$，$i = 1, \dots, n$），令 $x = x_0 + sh$，Newton 插值的公式可化为：

$$f(x) = \sum_{j=0}^{n} \binom{s}{j} j! h^j [y_0, \dots, y_j]$$

上式称为 **Newton 前向差分公式**（Newton forward divided difference formula）。

note

若样本点是等距的，我们还可以推出：

$$[y_k, \dots, y_{k+j}] = \frac{1}{j! h^j} \Delta^{(j)} y_k$$

其中 $\Delta^{(j)} y_k$ 为**前向差分**（forward differences），定义如下：

$$\begin{aligned} \Delta^{(0)} y_k &:= y_k, & k &= 0, \dots, n, \\ \Delta^{(j)} y_k &:= \Delta^{(j-1)} y_{k+1} - \Delta^{(j-1)} y_k, & k &= 0, \dots, n-j,\ j = 1, \dots, n. \end{aligned}$$

" 代码实现（Luogu P4781【模板】拉格朗日插值[3]）"

```
#include <bits/stdc++.h>
using namespace std;

constexpr uint32_t MOD = 998244353;

struct mint {
  uint32_t v_;

  mint() : v_(0) {}

  mint(int64_t v) {
    int64_t x = v % (int64_t)MOD;
    v_ = (uint32_t)(x + (x < 0 ? MOD : 0));
  }

  friend mint inv(mint const &x) {
    int64_t a = x.v_, b = MOD;
    if ((a %= b) == 0) return 0;
    int64_t s = b, m0 = 0;
    for (int64_t q = 0, _ = 0, m1 = 1; a;) {
      _ = s - a * (q = s / a);
      s = a;
      a = _;
      _ = m0 - m1 * q;
      m0 = m1;
      m1 = _;
    }
    return m0;
  }

  mint &operator+=(mint const &r) {
    if ((v_ += r.v_) >= MOD) v_ -= MOD;
    return *this;
```

```cpp
  }

  mint &operator-=(mint const &r) {
    if ((v_ -= r.v_) >= MOD) v_ += MOD;
    return *this;
  }

  mint &operator*=(mint const &r) {
    v_ = (uint32_t)((uint64_t)v_ * r.v_ % MOD);
    return *this;
  }

  mint &operator/=(mint const &r) { return *this = *this * inv(r); }

  friend mint operator+(mint l, mint const &r) { return l += r; }

  friend mint operator-(mint l, mint const &r) { return l -= r; }

  friend mint operator*(mint l, mint const &r) { return l *= r; }

  friend mint operator/(mint l, mint const &r) { return l /= r; }
};

template <class T>
struct NewtonInterp {
  // {(x_0,y_0),...,(x_{n-1},y_{n-1})}
  vector<pair<T, T>> p;
  // dy[r][l] = [y_l,...,y_r]
  vector<vector<T>> dy;
  // (x-x_0)...(x-x_{n-1})
  vector<T> base;
  // [y_0]+...+[y_0,y_1,...,y_n](x-x_0)...(x-x_{n-1})
  vector<T> poly;

  void insert(T const &x, T const &y) {
    p.emplace_back(x, y);
    size_t n = p.size();
    if (n == 1) {
      base.push_back(1);
    } else {
      size_t m = base.size();
      base.push_back(0);
      for (size_t i = m; i; --i) base[i] = base[i - 1];
      base[0] = 0;
      for (size_t i = 0; i < m; ++i)
        base[i] = base[i] - p[n - 2].first * base[i + 1];
    }
    dy.emplace_back(p.size());
    dy[n - 1][n - 1] = y;
    if (n > 1) {
      for (size_t i = n - 2; ~i; --i)
        dy[n - 1][i] =
            (dy[n - 2][i] - dy[n - 1][i + 1]) / (p[i].first - p[n - 1].first);
    }
```

```
    poly.push_back(0);
    for (size_t i = 0; i < n; ++i) poly[i] = poly[i] + dy[n - 1][0] * base[i];
  }

  T eval(T const &x) {
    T ans{};
    for (auto it = poly.rbegin(); it != poly.rend(); ++it) ans = ans * x + *it;
    return ans;
  }
};

int main() {
  NewtonInterp<mint> ip;
  int n, k;
  cin >> n >> k;
  for (int i = 1, x, y; i <= n; ++i) {
    cin >> x >> y;
    ip.insert(x, y);
  }
  cout << ip.eval(k).v_;
  return 0;
}
```

横坐标是连续整数的 Newton 插值

例如：求某三次多项式 $f(x)=\sum_{i=0}^{3}a_ix^i$ 的多项式系数，已知 $f(1)$ 至 $f(6)$ 的值分别为 $1,5,14,30,55,91$。

$$
\begin{array}{ccccccccccc}
1 & & 5 & & 14 & & 30 & & 55 & & 91 \\
& 4 & & 9 & & 16 & & 25 & & 36 & \\
& & 5 & & 7 & & 9 & & 11 & & \\
& & & 2 & & 2 & & 2 & & &
\end{array}
$$

第一行为 $f(x)$ 的连续的前 n 项；之后的每一行为之前一行中对应的相邻两项之差。观察到，如果这样操作的次数足够多（前提是 $f(x)$ 为多项式），最终总会返回一个定值。

计算出第 $i-1$ 阶差分的首项为 $\sum_{j=1}^{i}(-1)^{i+j}\binom{i-1}{j-1}f(j)$，第 $i-1$ 阶差分的首项对 $f(k)$ 的贡献为 $\binom{k-1}{i-1}$ 次。

$$
f(k)=\sum_{i=1}^{n}\binom{k-1}{i-1}\sum_{j=1}^{i}(-1)^{i+j}\binom{i-1}{j-1}f(j)
$$

时间复杂度为 $O(n^2)$.

C++ 中的实现

自 C++ 20 起，标准库添加了 `std::midpoint`[5] 和 `std::lerp`[6] 函数，分别用于求中点和线性插值。

参考资料

1. Interpolation - Wikipedia[7]
2. Newton polynomial - Wikipedia[8]

参考资料与注释

[1] 线性插值

[2] 多项式插值

[3] Luogu P4781【模板】拉格朗日插值 [3-1] [3-2]

[4] CF622F The Sum of the k-th Powers

[5] `std::midpoint`

[6] `std::lerp`

[7] Interpolation - Wikipedia

[8] Newton polynomial - Wikipedia

9.20.2 数值积分

Authors: H-J-Granger, Chrogeek, countercurrent-time, Enter-tainer, Great-designer, iamtwz, Ir1d, ksyx, mao1t, Menci, NachtgeistW, Nanarikom, ShaoChenHeng, StudyingFather, SukkaW, Tiphereth-A, zyj-111

定积分的定义

简单来说，函数 $f(x)$ 在区间 $[l,r]$ 上的定积分 $\int_l^r f(x)\mathrm{d}x$ 指的是 $f(x)$ 在区间 $[l,r]$ 中与 x 轴围成的区域的面积（其中 x 轴上方的部分为正值，x 轴下方的部分为负值）。

很多情况下，我们需要高效，准确地求出一个积分的近似值。下面介绍的**辛普森法**，就是这样一种求数值积分的方法。

辛普森法

这个方法的思想是将被积区间分为若干小段，每段套用二次函数的积分公式进行计算。

" 二次函数积分公式（辛普森公式）"

对于一个二次函数 $f(x)=ax^2+bx+c$，有：

$$\int_l^r f(x)\mathrm{d}x=\frac{(r-l)(f(l)+f(r)+4f(\frac{l+r}{2}))}{6}$$

推导过程：对于一个二次函数 $f(x)=ax^2+bx+c$；求积分可得 $F(x)=\int_0^x f(x)\mathrm{d}x=\frac{a}{3}x^3+\frac{b}{2}x^2+cx+D$

在这里 D 是一个常数，那么

$$
\begin{aligned}
\int_l^r f(x)\mathrm{d}x &= F(r)-F(l) \\
&= \frac{a}{3}(r^3-l^3)+\frac{b}{2}(r^2-l^2)+c(r-l) \\
&= (r-l)(\frac{a}{3}(l^2+r^2+lr)+\frac{b}{2}(l+r)+c) \\
&= \frac{r-l}{6}(2al^2+2ar^2+2alr+3bl+3br+6c) \\
&= \frac{r-l}{6}((al^2+bl+c)+(ar^2+br+c)+4(a(\frac{l+r}{2})^2+b(\frac{l+r}{2})+c)) \\
&= \frac{r-l}{6}(f(l)+f(r)+4f(\frac{l+r}{2}))
\end{aligned}
$$

根据这个辛普森公式，我们先介绍一种普通的辛普森积分法。

普通辛普森法

1743 年，这种方法发表于托马斯 · 辛普森的一篇论文中。

描述

给定一个自然数 n，将区间 $[l,r]$ 分成 $2n$ 个等长的区间 x。

$x_i = l+ih,\ \ i=0\ldots 2n,\ h=\frac{r-l}{2n}$.

我们就可以计算每个小区间 $[x_{2i-2},x_{2i}]$，$i=1\ldots n$ 的积分值，将所有区间的积分值相加即为总积分。

对于 $[x_{2i-2},x_{2i}]$，$i=1\ldots n$ 的一个区间，选其中的三个点 $(x_{2i-2},x_{2i-1},x_{2i})$ 就可以构成一条抛物线从而得到一个函数 $P(x)$，这个函数存在且唯一。计算原函数在该区间的积分值就变成了计算新的二次函数 $P(x)$ 在该段区间的积分值。这样我们就可以利用辛普森公式来近似计算它。

$\int_{x_{2i-2}}^{x_{2i}} f(x)\ dx \approx \int_{x_{2i-2}}^{x_{2i}} P(x)\ dx = (f(x_{2i-2})+4f(x_{2i-1})+(f(x_{2i}))\,\frac{h}{3}$

将其分段求和即可得到如下结论：

$\int_l^r f(x)dx \approx (f(x_0)+4f(x_1)+2f(x_2)+4f(x_3)+2f(x_4)+\ldots+4f(x_{2N-1})+f(x_{2N}))\,\frac{h}{3}$

误差

我们直接给出结论，普通辛普森法的误差为：

$$-\frac{1}{90}\left(\frac{r-l}{2}\right)^5 f^{(4)}(\xi)$$

其中 ξ 是位于区间 $[l,r]$ 的某个值。

实现

```
const int N = 1000 * 1000;

double simpson_integration(double a, double b) {
  double h = (b - a) / N;
  double s = f(a) + f(b);
  for (int i = 1; i <= N - 1; ++i) {
    double x = a + h * i;
    s += f(x) * ((i & 1) ? 4 : 2);
  }
  s *= h / 3;
  return s;
}
```

```
N = 1000 * 1000

def simpson_integration(a, b):
    h = (b - a) / N
    s = f(a) + f(b)
    for i in range(1, N):
        x = a + h * i
        if i & 1:
            s = s + f(x) * 4
        else:
            s = s + f(x) * 2
    s = s * (h / 3)
    return s
```

自适应辛普森法

普通的方法为保证精度在时间方面无疑会受到 n 的限制，我们应该找一种更加合适的方法。

现在唯一的问题就是如何进行分段。如果段数少了计算误差就大，段数多了时间效率又会低。我们需要找到一个准确度和效率的平衡点。

我们这样考虑：假如有一段图像已经很接近二次函数的话，直接带入公式求积分，得到的值精度就很高了，不需要再继续分割这一段了。

于是我们有了这样一种分割方法：每次判断当前段和二次函数的相似程度，如果足够相似的话就直接代入公式计算，否则将当前段分割成左右两段递归求解。

现在就剩下一个问题了：如果判断每一段和二次函数是否相似？

我们把当前段直接代入公式求积分，再将当前段从中点分割成两段，把这两段再直接代入公式求积分。如果当前段的积分和分割成两段后的积分之和相差很小的话，就可以认为当前段和二次函数很相似了，不用再递归分割了。

上面就是自适应辛普森法的思想。在分治判断的时候，除了判断精度是否正确，一般还要强制执行最少的迭代次数。

参考代码如下：

```
double simpson(double l, double r) {
  double mid = (l + r) / 2;
  return (r - l) * (f(l) + 4 * f(mid) + f(r)) / 6;  // 辛普森公式
}

double asr(double l, double r, double eps, double ans, int step) {
  double mid = (l + r) / 2;
  double fl = simpson(l, mid), fr = simpson(mid, r);
  if (abs(fl + fr - ans) <= 15 * eps && step < 0)
    return fl + fr + (fl + fr - ans) / 15;  // 足够相似的话就直接返回
  return asr(l, mid, eps / 2, fl, step - 1) +
         asr(mid, r, eps / 2, fr, step - 1);  // 否则分割成两段递归求解
}

double calc(double l, double r, double eps) {
  return asr(l, r, eps, simpson(l, r), 12);
}
```

```
def simpson(l, r):
    mid = (l + r) / 2
    return (r - l) * (f(l) + 4 * f(mid) + f(r)) / 6 # 辛普森公式
def asr(l, r, eps, ans, step):
```

```python
    mid = (l + r) / 2
    fl = simpson(l, mid); fr = simpson(mid, r)
    if abs(fl + fr - ans) <= 15 * eps and step < 0:
        return fl + fr + (fl + fr - ans) / 15 # 足够相似的话就直接返回
    return asr(l, mid, eps / 2, fl, step - 1) + \
           asr(mid, r, eps / 2, fr, step - 1) # 否则分割成两段递归求解
def calc(l, r, eps):
    return asr(l, r, eps, simpson(l, r), 12)
```

习题

- Luogu4525【模板】自适应辛普森法 1[1]
- HDU1724 Ellipse[2]
- NOI2005 月下柠檬树[3]

参考资料

https://doi.org/10.1145/321526.321537[4]：该文章讨论了自适应 Simpson 法的改进方案，其中详细论述了上文代码中的常数 `15` 的由来与优势。

参考资料与注释

[1] Luogu4525【模板】自适应辛普森法 1

[2] HDU1724 Ellipse

[3] NOI2005 月下柠檬树

[4] https://doi.org/10.1145/321526.321537

9.20.3 高斯消元

Authors: StudyingFather, CCXXXI, Chrogeek, ChungZH, countercurrent-time, Early0v0, Enter-tainer, GavinZhengOI, Great-designer, H-J-Granger, henrytbtrue, HeRaNO, huayucaiji, iamtwz, Ir1d, ksyx, MegaOwIer, NachtgeistW, P-Y-Y, qwqAutomaton, shuzhouliu, shuzhouliu-bot, Siger Young, sshwy, SukkaW, Tiphereth-A, tsentau, When-Melancholy, Xeonacid, Yukimaikoriya, Zhoier, zyj-111

引入

高斯消元法（Gauss–Jordan elimination）是求解线性方程组的经典算法，它在当代数学中有着重要的地位和价值，是线性代数课程教学的重要组成部分。

高斯消元法除了用于线性方程组求解外，还可以用于行列式计算、求矩阵的逆，以及其他计算机和工程方面。

消元法及高斯消元法思想

定义

消元法是将方程组中的一方程的未知数用含有另一未知数的代数式表示，并将其带入到另一方程中，这就消去了一未知数，得到一解；或将方程组中的一方程倍乘某个常数加到另外一方程中去，也可达到消去一未知数的目的。消元法主要用于二元一次方程组的求解。

解释

例一：利用消元法求解二元一次线性方程组：

$$\begin{cases} 4x + y &= 100 \\ x - y &= 100 \end{cases}$$

解：将方程组中两方程相加，消元 y 可得：

$5x = 200$

解得：

$x = 40$

将 $x = 40$ 代入方程组中第二个方程可得：

$y = -60$

消元法理论的核心

消元法理论的核心主要如下：

- 两方程互换，解不变；
- 一方程乘以非零数 k，解不变；
- 一方程乘以数 k 加上另一方程，解不变。

高斯消元法思想概念

德国数学家高斯对消元法进行了思考分析，得出了如下结论：

- 在消元法中，参与计算和发生改变的是方程中各变量的系数；
- 各变量并未参与计算，且没有发生改变；
- 可以利用系数的位置表示变量，从而省略变量；
- 在计算中将变量简化省略，方程的解不变。

> 高斯在这些结论的基础上，提出了高斯消元法，首先将方程的增广矩阵利用行初等变换化为行最简形，然后以线性无关为准则对自由未知量赋值，最后列出表达方程组通解。

高斯消元五步骤法

解释

高斯消元法在将增广矩阵化为最简形后对于自由未知量的赋值，需要掌握线性相关知识，且赋值存在人工经验的因素，使得在学习过程中有一定的困难，将高斯消元法划分为五步骤，从而提出五步骤法，内容如下：

1. 增广矩阵行初等行变换为行最简形；
2. 还原线性方程组；
3. 求解第一个变量；
4. 补充自由未知量；
5. 列表示方程组通解。

利用实例进一步说明该算法的运作情况。

过程

例二：利用高斯消元法五步骤法求解线性方程组：

$$\begin{cases} 2x_1 + 5x_3 + 6x_4 & = 9 \\ x_3 + x_4 & = -4 \\ 2x_3 + 2x_4 & = -8 \end{cases}$$

增广矩阵行（初等）变换为行最简形

所谓增广矩阵，即为方程组系数矩阵 A 与常数列 b 的并生成的新矩阵，即 $(A|b)$，增广矩阵行初等变换化为行最简形，即是利用了高斯消元法的思想理念，省略了变量而用变量的系数位置表示变量，增广矩阵中用竖线隔开了系数矩阵和常数列，代表了等于符号。

$$\left(\begin{array}{cccc|c} 2 & 0 & 5 & 6 & 9 \\ 0 & 0 & 1 & 1 & -4 \\ 0 & 0 & 2 & 2 & -8 \end{array}\right)$$

$$\xrightarrow{r_3 - 2r_2} \left(\begin{array}{cccc|c} 2 & 0 & 5 & 6 & 9 \\ 0 & 0 & 1 & 1 & -4 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

化为行阶梯形

$$\xrightarrow{\frac{r_1}{2}} \left(\begin{array}{cccc|c} 1 & 0 & 2.5 & 3 & 4.5 \\ 0 & 0 & 1 & 1 & -4 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

$$\xrightarrow{r_1 - r_2 \times 2.5} \left(\begin{array}{cccc|c} 1 & 0 & 0 & 0.5 & 14.5 \\ 0 & 0 & 1 & 1 & -4 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

化为最简形

还原线性方程组

$$\begin{cases} x_1 + 0.5x_4 & = 14.5 \\ x_3 + x_4 & = -4 \end{cases}$$

解释

> 所谓的还原线性方程组，即是在行最简形的基础上，将之重新书写为线性方程组的形式，即将行最简形中各位置的系数重新赋予变量，中间的竖线还原为等号。

求解第一个变量

$$\begin{cases} x_1 = -0.5x_4 + 14.5 \\ x_3 = -x_4 - 4 \end{cases}$$

解释

> 即是对于所还原的线性方程组而言，将方程组中每个方程的第一个变量，用其他量表达出来。如方程组两方程中的第一个变量 x_1 和 x_3

补充自由未知量

$$\begin{cases} x_1 = -0.5x_4 + 14.5 \\ x_2 = x_2 \\ x_3 = -x_4 - 4 \\ x_4 = x_4 \end{cases}$$

解释

> 第 3 步中，求解出变量 x_1 和 x_3，从而说明了方程剩余的变量 x_2 和 x_4 不受方程组的约束，是自由未知量，可以取任意值，所以需要在第 3 步骤解得基础上进行解得补充，补充的方法为 $x_2 = x_2, x_4 = x_4$，这种解得补充方式符合自由未知量定义，并易于理解，因为是自由未知量而不受约束，所以只能自己等于自己。

列表示方程组的通解

$$\begin{aligned} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} &= \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} x_2 + \begin{pmatrix} -0.5 \\ 0 \\ -1 \\ 1 \end{pmatrix} x_4 + \begin{pmatrix} 14.5 \\ 0 \\ -4 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} C_1 + \begin{pmatrix} -0.5 \\ 0 \\ -1 \\ 1 \end{pmatrix} C_2 + \begin{pmatrix} 14.5 \\ 0 \\ -4 \\ 0 \end{pmatrix} \end{aligned}$$

其中 C_1 和 C_2 为任意常数。

解释

> 即在第 4 步的基础上，将解表达为列向量组合的表示形式，同时由于 x_2 和 x_4 是自由未知量，可以取任意值，所以在解得右边，令二者分别为任意常数 C_1 和 C_2，即实现了对方程组的求解。

行列式计算

解释

$N \times N$ 方阵行列式（Determinant）可以理解为所有列向量所夹的几何体的有向体积

例如：

$$\begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix} = 1$$

$$\begin{vmatrix} 1 & 2 \\ 2 & 1 \end{vmatrix} = -3$$

行列式有公式

$$\det(A) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)}$$

其中 S_n 是指长度为 n 的全排列的集合，σ 就是一个全排列，如果 σ 的逆序对对数为偶数，则 $\operatorname{sgn}(\sigma) = 1$，否则 $\operatorname{sgn}(\sigma) = -1$。

通过体积概念理解行列式不变性是一个非常简单的办法：

- 矩阵转置，行列式不变；

- 矩阵行（列）交换，行列式取反；
- 矩阵行（列）相加或相减，行列式不变；
- 矩阵行（列）所有元素同时乘以数 k，行列式等比例变大。

> 由此，对矩阵应用高斯消元之后，我们可以得到一个对角线矩阵，此矩阵的行列式由对角线元素之积所决定。其符号可由交换行的数量来确定（如果为奇数，则行列式的符号应颠倒）。因此，我们可以在 $O(n^3)$ 的复杂度下使用高斯算法计算矩阵。
>
> 注意，如果在某个时候，我们在当前列中找不到非零单元，则算法应停止并返回 0。

实现

```
const double EPS = 1E-9;
int n;
vector<vector<double> > a(n, vector<double>(n));

double det = 1;
for (int i = 0; i < n; ++i) {
  int k = i;
  for (int j = i + 1; j < n; ++j)
    if (abs(a[j][i]) > abs(a[k][i])) k = j;
  if (abs(a[k][i]) < EPS) {
    det = 0;
    break;
  }
  swap(a[i], a[k]);
  if (i != k) det = -det;
  det *= a[i][i];
  for (int j = i + 1; j < n; ++j) a[i][j] /= a[i][i];
  for (int j = 0; j < n; ++j)
    if (j != i && abs(a[j][i]) > EPS)
      for (int k = i + 1; k < n; ++k) a[j][k] -= a[i][k] * a[j][i];
}

cout << det;
```

矩阵求逆

对于方阵 A，若存在方阵 A^{-1}，使得 $A \times A^{-1} = A^{-1} \times A = I$，则称矩阵 A 可逆，A^{-1} 被称为它的逆矩阵。

给出 n 阶方阵 A，求解其逆矩阵的方法如下：

1. 构造 $n \times 2n$ 的矩阵 (A, I_n)；
2. 用高斯消元法将其化简为最简形 (I_n, A^{-1})，即可得到 A 的逆矩阵 A^{-1}。如果最终最简形的左半部分不是单位矩阵 I_n，则矩阵 A 不可逆。

该方法的正确性证明需要用到较多线性代数的知识，限于篇幅这里不再给出。感兴趣的读者可以自行查阅相关资料。

高斯消元法解异或方程组

异或方程组是指形如

$$\begin{cases} a_{1,1}x_1 \oplus a_{1,2}x_2 \oplus \cdots \oplus a_{1,n}x_n & = b_1 \\ a_{2,1}x_1 \oplus a_{2,2}x_2 \oplus \cdots \oplus a_{2,n}x_n & = b_2 \\ \cdots & \cdots \\ a_{m,1}x_1 \oplus a_{m,2}x_2 \oplus \cdots \oplus a_{m,n}x_n & = b_1 \end{cases}$$

的方程组，其中 $\oplus$ 表示「按位异或」（即 `xor` 或 C++ 中的 `^`），且式中所有系数/常数（即 $a_{i,j}$ 与 b_i）均为 0 或 1。

由于「异或」符合交换律与结合律，故可以按照高斯消元法逐步消元求解。值得注意的是，我们在消元的时候应使用「异或消元」而非「加减消元」，且不需要进行乘除改变系数（因为系数均为 0 和 1）。

注意到异或方程组的增广矩阵是 01 矩阵（矩阵中仅含有 0 与 1），所以我们可以使用 C++ 中的 `std::bitset` 进行优化，将时间复杂度降为 $O(\dfrac{n^2m}{\omega})$，其中 n 为元的个数，m 为方程条数，ω 一般为 32（与机器有关）。

参考实现：

```cpp
std::bitset<1010> matrix[2010];  // matrix[1~n]：增广矩阵，0 位置为常数

std::vector<bool> GaussElimination(
    int n, int m)  // n 为未知数个数，m 为方程个数，返回方程组的解
                   // （多解 / 无解返回一个空的 vector）
{
  for (int i = 1; i <= n; i++) {
    int cur = i;
    while (cur <= m && !matrix[cur].test(i)) cur++;
    if (cur > m) return std::vector<bool>(0);
    if (cur != i) swap(matrix[cur], matrix[i]);
    for (int j = 1; j <= m; j++)
      if (i != j && matrix[j].test(i)) matrix[j] ^= matrix[i];
  }
  std::vector<bool> ans(n + 1, 0);
  for (int i = 1; i <= n; i++) ans[i] = matrix[i].test(0);
  return ans;
}
```

练习题

- Codeforces - 巫师和赌注[1]
- luogu - SDOI2010 外星千足虫[2]

参考资料与注释

[1] Codeforces - 巫师和赌注

[2] luogu - SDOI2010 外星千足虫

9.20.4 牛顿迭代法

Authors: Marcythm, iamtwz, nutshellfool, sshwy, allenanswerzq, countercurrent-time, Enter-tainer, H-J-Granger, hly1204, Ir1d, Menci, NachtgeistW, SukkaW, Tiphereth-A, Xeonacid

引入

本文介绍如何用牛顿迭代法（Newton's method for finding roots）求方程的近似解，该方法于 17 世纪由牛顿提出。

具体的任务是，对于在 $[a, b]$ 上连续且单调的函数 $f(x)$，求方程 $f(x)=0$ 的近似解。

解释

初始时我们从给定的 $f(x)$ 和一个近似解 x_0 开始（初值的问题与 Newton 分形有关，可参考 3Blue1Brown 的 牛顿分形[1]）。

假设我们目前的近似解是 x_i，我们画出与 $f(x)$ 切于点 $(x_i, f(x_i))$ 的直线 l，将 l 与 x 轴的交点横坐标记为 x_{i+1}，那么这就是一个更优的近似解。重复这个迭代的过程。根据导数的几何意义，可以得到如下关系：

$$f'(x_i) = \frac{f(x_i)}{x_i - x_{i+1}}$$

整理后得到如下递推式：

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

直观地说，如果 $f(x)$ 比较平滑，那么随着迭代次数的增加，x_i 会越来越逼近方程的解。

牛顿迭代法的收敛率是平方级别的，这意味着每次迭代后近似解的精确数位会翻倍。关于牛顿迭代法的收敛性证明可参考 citizendium - Newton method Convergence analysis[2]

当然牛顿迭代法也同样存在着缺陷，详情参考 Xiaolin Wu - Roots of Equations 第 18 - 20 页分析[3]

求解平方根

我们尝试用牛顿迭代法求解平方根。设 $f(x) = x^2 - n$，这个方程的近似解就是 $\sqrt{n}$ 的近似值。于是我们得到

$$x_{i+1} = x_i - \frac{x_i^2 - n}{2x_i} = \frac{x_i + \frac{n}{x_i}}{2}$$

在实现的时候注意设置合适的精度。代码如下

实现

```cpp
double sqrt_newton(double n) {
  const double eps = 1E-15;
  double x = 1;
  while (true) {
    double nx = (x + n / x) / 2;
    if (abs(x - nx) < eps) break;
    x = nx;
  }
  return x;
}
```

```python
def sqrt_newton(n):
    eps = 1e-15
    x = 1
    while True:
```

```
        nx = (x + n / x) / 2
        if abs(x - nx) < eps:
            break
        x = nx
    return x
```

求解整数平方根

尽管我们可以调用 `sqrt()` 函数来获取平方根的值，但这里还是讲一下牛顿迭代法的变种算法，用于求不等式 $x^2 \le n$ 的最大整数解。我们仍然考虑一个类似于牛顿迭代的过程，但需要在边界条件上稍作修改。如果 x 在迭代的过程中上一次迭代值得近似解变小，而这一次迭代使得近似解变大，那么我们就不进行这次迭代，退出循环。

实现

```cpp
int isqrt_newton(int n) {
  int x = 1;
  bool decreased = false;
  for (;;) {
    int nx = (x + n / x) >> 1;
    if (x == nx || (nx > x && decreased)) break;
    decreased = nx < x;
    x = nx;
  }
  return x;
}
```

```python
def isqrt_newton(n):
    x = 1
    decreased = False
    while True:
        nx = (x + n // x) // 2
        if x == nx or (nx > x and decreased):
            break
        decreased = nx < x
        x = nx
    return x
```

高精度平方根

最后考虑高精度的牛顿迭代法。迭代的方法是不变的，但这次我们需要关注初始时近似解的设置，即 x_0 的值。由于需要应用高精度的数一般都非常大，因此不同的初始值对于算法效率的影响也很大。一个自然的想法就是考虑 $x_0 = 2^{\lfloor \frac{1}{2} \log_2 n \rfloor}$，这样既可以快速计算出 x_0，又可以较为接近平方根的近似解。

实现

给出 Java 代码的实现：

```java
public static BigInteger isqrtNewton(BigInteger n) {
  BigInteger a = BigInteger.ONE.shiftLeft(n.bitLength() / 2);
  boolean p_dec = false;
  for (;;) {
    BigInteger b = n.divide(a).add(a).shiftRight(1);
    if (a.compareTo(b) == 0 || a.compareTo(b) < 0 && p_dec)
      break;
```

```
      p_dec = a.compareTo(b) > 0;
      a = b;
    }
    return a;
  }
```

实践效果：在 $n = 10^{1000}$ 的时候该算法的运行时间是 60 ms，如果我们不优化 x_0 的值，直接从 $x_0 = 1$ 开始迭代，那么运行时间将增加到 120 ms。

习题

- UVa 10428 - The Roots[4]
- LeetCode 69. x 的平方根[5]

本页面主要译自博文 ( )[6] 与其英文翻译版 Newton's method for finding roots[7]。其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0。

参考资料与注释

[1] 牛顿分形

[2] citizendium - Newton method Convergence analysis

[3] Xiaolin Wu - Roots of Equations 第 18 - 20 页分析

[4] UVa 10428 - The Roots

[5] LeetCode 69. x 的平方根

[6] ( )

[7] Newton's method for finding roots

9.21 傅里叶-莫茨金消元法

引入

傅里叶——莫茨金消元法（原名 Fourier–Motzkin Elimination，简称 **FME** 算法）是一种用于从线性不等式中消除变量的数学方法。

它的命名源自于在 1827 年和 1936 年独立发现该算法的 Joseph Fourier 和 Theodore Motzkin 的姓氏。

过程

从线性不等式中消除一组变量，是指通过将关系式中的若干个元素有限次地变换，消去其中的某些元素，从而解决问题的一种方法。

如果线性不等式中的所有变量都被消除，那么我们会得到一个常不等式。因为当且仅当原不等式有解时，消元后的不等式才为真，消除所有变量可用于检测不等式系统是否有解。

考虑一个含 n 个不等式的系统 S，有从 x_1 到 x_r 的 r 个变量，其中 x_r 为要消除的变量。根据 x_r 系数的符号（正、负或空），S 中的线性不等式可以分为三类：

1. 形式为 $x_r \geq b_i - \sum_{k=1}^{r-1} a_{ik}x_k$ 的不等式，对于范围从 1 到 n_A（n_A 为这种不等式的数量）的 j，用 $x_r \geq A_j(x_1, \dots, x_{r-1})$ 表示；
2. 形式为 $x_r \leq b_i - \sum_{k=1}^{r-1} a_{ik}x_k$ 的不等式，对于范围从 1 到 n_B（n_B 为这种不等式的数量）的 j，用 $x_r \leq B_j(x_1, \dots, x_{r-1})$ 表示；
3. 不包含 x_r 的不等式，设它们构成的不等式组为 ϕ。

因此原系统等价于

$$\max(A_1(x_1, \dots, x_{r-1}), \dots, A_{n_A}(x_1, \dots, x_{r-1})) \leq x_r \leq \min(B_1(x_1, \dots, x_{r-1}), \dots, B_{n_B}(x_1, \dots, x_{r-1})) \wedge \phi$$

消元包括产生一个等价于 $\exists x_r\ S$ 的系统。显然，这个公式等价于

$$\max(A_1(x_1, \dots, x_{r-1}), \dots, A_{n_A}(x_1, \dots, x_{r-1})) \leq \min(B_1(x_1, \dots, x_{r-1}), \dots, B_{n_B}(x_1, \dots, x_{r-1})) \wedge \phi$$

不等式

$$\max(A_1(x_1, \dots, x_{r-1}), \dots, A_{n_A}(x_1, \dots, x_{r-1})) \leq \min(B_1(x_1, \dots, x_{r-1}), \dots, B_{n_B}(x_1, \dots, x_{r-1}))$$

等价于对于 $1 \leq i \leq n_A$ 且 $1 \leq j \leq n_B$，所有 $n_A n_B$ 个不等式 $A_i(x_1, \dots, x_{r-1}) \leq B_j(x_1, \dots, x_{r-1})$ 构成的不等式组。

因此，我们将原系统 S 转换为另一个消掉 x_r 的系统，这个系统有 $(n - n_A - n_B) + n_A n_B$ 个不等式。特别地，如果 $n_A = n_B = n/2$，那么新系统不等式的个数为 $n^2/4$。

例题

考虑以下不等式系统：

$2x - 5y + 4z \leq 10$

$3x - 6y + 3z \leq 9$

$-x + 5y - 2z \leq -7$

$-3x + 2y + 6z \leq 12$

为了消除 x，我们可以根据 x 改写不等式：

$x \leq (10 + 5y - 4z)/2$

$x \leq (9 + 6y - 3z)/3$

$x \geq 7 + 5y - 2z$

$x \geq (-12 + 2y + 6z)/3$

这样我们得到两个 $\leq$ 不等式和两个 $\geq$ 不等式；如果每个 $\leq$ 不等式的右侧至少是每个 $\geq$ 不等式的右侧，则系统有一个解。我们有 2×2 这样的组合：

$7 + 5y - 2z \leq (10 + 5y - 4z)/2$

$7 + 5y - 2z \leq (9 + 6y - 3z)/3$

$(-12 + 2y + 6z)/3 \leq (10 + 5y - 4z)/2$

$(-12 + 2y + 6z)/3 \leq (9 + 6y - 3z)/3$

现在我们有了一个新的少了一个变量不等式系统。

时间复杂度

在 n 个不等式上消元可以最多得到 $n^2/4$ 个不等式，因此连续运行 d 步可以得到最多 $4(n/4)^{2^d}$ 的双指数复杂度。这是由于算法产生了许多不必要的约束（其他约束隐含的约束）。必要约束的数量以单一指数增长。

可以使用线性规划 (Linear Programming, LP) 检测不必要的约束。

应用

信息论的可实现性证明保证了存在性能良好的编码方案的条件。这些条件通常使用线性不等式系统描述。系统的变量包括传输速率和附加辅助速率。通常，人们旨在仅根据问题的参数（即传输速率）来描述通信的基本限制，因此述辅助率需要消除上。而我们正是通过傅立叶 - 莫茨金消元法来做到这一点的。

实现

在编程语言中，Racket[1]，一种基于 Lisp 的多范式编程语言在 fme - Fourier–Motzkin Elimination for Integer Systems)[2] 中对 FME 算法做了简单函数代数实现。

参考资料与拓展阅读

1. Rui-Juan Jing, Marc Moreno-Maza, Delaram Talaashrafi, "Complexity Estimates for Fourier-Motzkin Elimination[3]", Journal of Functional Programming 16:2 (2006) pp 197-217.
2. Fourier–Motzkin elimination - Wikipedia[4]
3. Fourier-Motzkin elimination and its dual[5]
4. GE Liepins,Fourier-Motzkin elimination for mixed systems[6], 1983

参考资料与注释

[1] Racket

[2] fme - Fourier–Motzkin Elimination for Integer Systems)

[3] Complexity Estimates for Fourier-Motzkin Elimination

[4] Fourier–Motzkin elimination - Wikipedia

[5] Fourier-Motzkin elimination and its dual

[6] Fourier-Motzkin elimination for mixed systems

9.22 序理论

引入

序理论是利用二元关系来将「次序」这一概念严格化的数学分支，下面将介绍这一分支的基本定义。

定义

二元关系

" 定义"

集合 X 和集合 Y 上的一个**二元关系**（binary relation）R 定义为元组 $(X, Y, G(R))$，其中 X 称为定义域

(domain), Y 称为陪域 (codomain), $G(R) \subseteq X \times Y = \{(x, y) : x \in X, y \in Y\}$ 称为二元关系 R 的图 (graph)。xRy 成立当且仅当 $(x, y) \in G(R)$。

若 $X = Y$，则称该二元关系为齐次二元关系 (homogeneous relation) 或内关系 (endorelation)。

若没有特别说明，下文中的二元关系均为齐次二元关系。

例如 $\mathbf{N}_+$ 上的整除 $|$ 和小于等于 $\leq$ 均为二元关系。

我们研究二元关系时，往往会关注其是否具有一些特别的性质。对集合 S 上的二元关系 R，我们定义如下特殊性质：

1. 自反性 (reflexive)：$(\forall\ a \in S)\ \ aRa$，
2. 反自反性 (irreflexive，anti-reflexive)：$(\forall\ a \in S)\ \ \neg(aRa)$，
3. 对称性 (symmetric)：$(\forall\ a, b \in S)\ \ aRb \iff bRa$，
4. 反对称性 (antisymmetric)：$(\forall\ a, b \in S)\ \ (aRb \wedge bRa) \implies a = b$，
5. 非对称性 (asymmetric)：$(\forall\ a, b \in S)\ \ aRb \implies \neg(bRa)$，
6. 传递性 (transitive)：$(\forall\ a, b, c \in S)\ \ (aRb \wedge bRc) \implies aRc$，
7. 连接性 (connected)：$(\forall\ a, b \in S)\ \ a \neq b \implies (aRb \vee bRa)$，
8. 良基性 (well-founded)：$(\exists\ m \in S \neq \emptyset)\ \ (\forall\ a \in S\ \ \{m\})\ \ \neg(aRm)$ (即非空集合 S 中有极小元 m)，
9. 不可比的传递性 (transitive of incomparability)：$(\forall\ a, b, c \in S)\ \ (\neg(aRb \vee bRa) \wedge \neg(bRc \vee cRb)) \implies \neg(aRc \vee cRa)$ (若 $\neg(aRb \vee bRa)$，则称 a 和 b 是不可比的)。

同时我们定义一些特殊的二元关系：

二元关系	自反性	反自反性	对称性	反对称性	非对称性	传递性	连接性	良基性	不可比的传递性
等价关系 (equivalence relation)	有		有			有			
预序 (preorder，quasiorder)	有					有			
偏序 (partial order)	有			有		有			
全序 (total order)	有			有		有	有		
良序 (well-order)	有			有		有	有	有	
严格预序 (strict preorder)		有				有			
严格偏序 (strict partial order)		有			有	有			
严格弱序 (strict weak order)		有			有	有			有
严格全序 (strict total order)		有			有	有	有		

关系间的运算

对集合 X 和集合 Y 上的二元关系 R 和 S，我们可以定义如下运算：

1. R 和 S 的并 $R \cup S$ 满足 $G(R \cup S) := \{(x, y) : xRy \vee xSy\}$ (如 $\leq$ 是 $<$ 和 $=$ 的并)，
2. R 和 S 的交 $R \cap S$ 满足 $G(R \cap S) := \{(x, y) : xRy \wedge xSy\}$，
3. R 的补 $\bar{R}$ 满足 $G(\bar{R}) := \{(x, y) : \neg(xRy)\}$，
4. R 的对偶 R^T 满足 $G(R^T) := \{(y, x) : xRy\}$.

对集合 X 和集合 Y 上的二元关系 R 以及集合 Y 和集合 Z 上的二元关系 S，我们可以定义其复合 $S \circ R$ 满足

$G(S \circ R) := \{(x,z) : (\exists\ y \in Y)\ \ xRy \wedge ySz\}$.

偏序集

" 定义"

若集合 S 上的一个二元关系 $\preceq$ 具有**自反性**、**反对称性**、**传递性**，则称 S 是**偏序集**（partially ordered set，poset），$\preceq$ 为其上一**偏序**（partial order）。

若偏序 $\preceq$ 还具有**连接性**，则称其为**全序**（total order），对应的集合称为**全序集**（totally ordered set）、**线性序集**（linearly ordered set，loset）、**简单序集**（simply ordered set）。

由传递性和反对称性可以推出自反性，由传递性和自反性也可以推出反对称性。

不难发现 $\mathbf{N}$，$\mathbf{Z}$，$\mathbf{Q}$、$\mathbf{R}$ 均关于 $\le$ 构成全序集。

偏序集的可视化表示：Hasse 图

对于有限偏序集，我们可以用 Hasse 图直观地表示其上的偏序关系。

" 定义"

对有限偏序集 S 和其上的偏序 $\preceq$，定义 $x \prec y \iff (x \preceq y \wedge x \neq y)$ 其对应的 **Hasse 图**为满足如下条件的图 $G = \langle V, E \rangle$：

- $V = S$,
- $E = \{(x,y) \in S \times S : x \prec y \wedge ((\nexists\ z \in S)\ \ x \prec z \prec y)\}$

如对于集合 $\{0,1,2\}$ 的幂集 S 和集合的包含关系 $\subseteq$，其对应的 Hasse 图为：

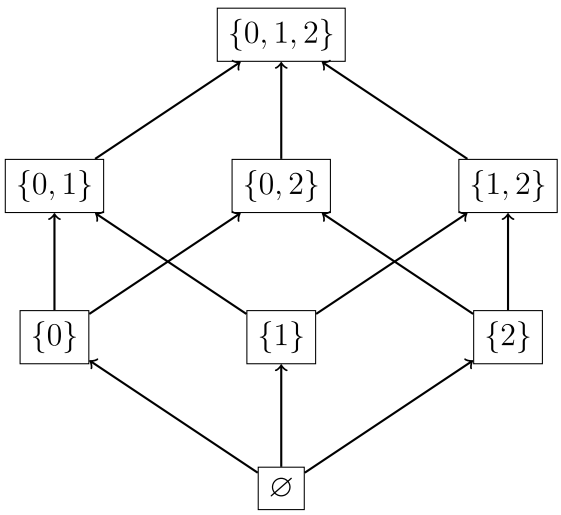


图 9.41

由于偏序具有反对称性，所以 Hasse 图一定是 有向无环图，进而我们可以根据 拓扑排序 对任意有限偏序集构造全序。

链与反链

" 定义"

对偏序集 S 和其上的偏序 $\preceq$，称 S 的全序子集为**链**（chain）。若 S 的子集 T 中任意两个不同元素均不可比

（即 $(\forall\ a,b\in T)\ \ a\neq b\implies (a\not\preceq b\wedge b\not\preceq a)$），则称 T 为**反链**（antichain）。

对偏序集 S 和其上的偏序 $\preceq$，我们将偏序集 S 的最长反链长度称为**宽度**（partial order width）。

如对于集合 $\{0,1,2\}$ 的幂集 S 和集合的包含关系 $\subseteq$，$\{\emptyset,\{1\},\{1,2\}\}$ 为一条链，$\{\{1\},\{0,2\}\}$ 为一条反链，S 的宽度为 3.

预序集中的特殊元素

在预序集中，我们可以定义极大（小）元、上（下）界、上（下）确界等概念，这些概念可以推广到其他序关系中。

" 定义"

对预序集 S 和其上的预序 $\preceq$，取 S 中的元素 m：

1. 若 $(\forall\ a\in S\ \ \{m\})\ \neg(m\preceq a)$，则称 m 为**极大元**（maximal element），
2. 若对 $T\subseteq S$ 满足 $(\forall\ t\in T)\ \ t\preceq m$，则称 m 为 T 的**上界**（upper bound），
3. 若对 $T\subseteq S$ 满足 m 是 T 的上界且对 T 的任意上界 n 均有 $m\preceq n$，则称 m 为 T 的**上确界**（supremum）。

类似可定义**极小元**（minimal element）、**下界**（lower bound）和**下确界**（infimum）。

如 1 是 $\mathbf{N}_+$ 的极小元和下界。

可以证明：

- 预序集中，极大（小）元、上（下）界、上（下）确界都是不一定存在的，即使存在也不一定唯一。
- 若偏序集 S 的子集 T 存在上（下）确界，则一定唯一。

我们可将 T 的上确界、下确界分别记为 $\sup T$，$\inf T$. 若偏序集 S 既有上界又有下界，则称 S 是有界的。

在无限偏序集中，极大元不一定存在。可用 **Zorn 引理**（Zorn's Lemma）来判断无限偏序集中是否存在极大元。

"Zorn 引理[1]"

Zorn 引理也被称为 **Kuratowski–Zorn 引理**，其内容为：若非空偏序集的每条链都有上界，则该偏序集存在极大元。

Zorn 引理与**选择公理**[2]、**良序定理**[3] 等价。

有向集与格

我们知道若偏序集的子集存在上（下）确界，则一定唯一。但是这一点并不适用于极大（小）元。例如：考虑偏序集 $S=\{\{0\},\{1\},\{2\},\{0,1\},\{0,2\},\{1,2\}\}$ 和其上的偏序 $\subseteq$，不难发现其有 3 个极大元和 3 个极小元。

我们希望通过向偏序集添加一定的条件来使得若极大（小）元存在则一定唯一，这样我们就可以定义最大（小）元的概念了。

" 有向集"

对预序集 S 和其上的预序 $\preceq$，若 $(\forall\ a,b\in S)\ \ (\exists\ c\in S)\ \ a\preceq c\wedge b\preceq c$，则称 $\preceq$ 为 S 的一个**方向**（direction），S 称为**有向集**（directed set）或**过滤集**（filtered set）。

有时也将满足上述定义的集合 S 称为**上有向集**（upward directed set），类似地可定义**下有向集**（downward directed set）。

有向集也可用如下方式定义：

" 有向集的等价定义"

对预序集 S 和其上的预序 $\preceq$，若 S 的任意有限子集 T 均有上界，则称 $\preceq$ 为 S 的一个方向，S 称为有向集。

不难发现：

- 若上有向集存在极大元，则一定唯一。我们将上有向集的极大元称为**最大元**（greatest element）。
- 若下有向集存在极小元，则一定唯一。我们将下有向集的极小元称为**最小元**（least element）。

有方向的偏序集中，对任意元素 a, b，$\{a, b\}$ 都有上界，若将上界修改为上确界，则得到了并半格的定义。

对偏序集 S 和其上的偏序 $\preceq$：

" 并半格"

若对 S 中的任意元素 a, b，$\{a, b\}$ 均有上确界 c，则称 S 为**并半格**（join-semilattice，upper semilattice），并且我们称 c 为 a 和 b 的**并**（join），记为 $a \vee b$.

" 交半格"

若对 S 中的任意元素 a, b，$\{a, b\}$ 均有下确界 c，则称 S 为**交半格**（meet-semilattice，lower semilattice），并且我们称 c 为 a 和 b 的**交**（meet），记为 $a \wedge b$.

" 格"

若 S 既是并半格也是交半格，则称 S 为**格**（lattice）。

例如 60 的正因子构成的集合 $S = \{1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60\}$ 关于整除构成偏序集，其上的任意正整数 a, b，$\operatorname{lcm}(a, b)$ 为 a 和 b 的并，$\gcd(a, b)$ 为 a 和 b 的交，从而 S 是格。

对偶

在序理论中，对偶是非常常见的概念，如上文提到的极大元与极小元对偶、上界与下界对偶、上确界与下确界对偶。

对偏序集 P 和其上的偏序 $\preceq$，定义其**对偶**（dual，opposite）偏序集 P^d 满足：$x \preceq y$ 在 P 中成立当且仅当 $y \preceq x$ 在 P^d 中成立。将 P 的 Hasse 图的边反转即可得到 P^d 的 Hasse 图。

Dilworth 定理与 Mirsky 定理

对有限偏序集 S 和其上的偏序 $\preceq$，我们有如下的一对对偶的定理：

"Dilworth 定理"

S 的宽度（最长反链长度）等于最小的链覆盖数。

" 证明"

考虑数学归纳法。当 $|S| \leq 3$ 时，命题显然成立。

假设命题对所有元素个数小于 $|S|$ 的偏序集都成立，令 S 的宽度为 d. 若 $|S|$ 中所有元素均不可比，则命题显然成立，否则在 S 中取一条长度大于 1 的链，令其中的最小元为 m，最大元为 M.

令 $T = S \ \ \{m, M\}$，若 T 中的宽度不超过 $d-1$，则由归纳假设知 T 可被至多 $d-1$ 条链覆盖，进而 S 可被这些链再加上链 $\{m, M\}$ 覆盖，命题成立，否则说明 T 中的宽度也为 d，令 T 中最长的一条反链为 A.

我们考虑如下两个集合：

$$S^+ := \{x \in S : (\exists\, a \in A) \quad a \preceq x\}$$

$$S^- := \{x \in S : (\exists\, a \in A) \quad x \preceq a\}$$

我们不难发现如下性质：

- $S^+ \cup S^- = S$,

- $S^+ \cap S^- = A$,
- $|S^+| < |S|$,$|S^-| < |S|$（因为 $m \notin S^+$ 且 $M \notin S^-$）。

对 S^+ 和 S^- 都应用归纳假设，则这两个集合的最小链覆盖数为 d，且这些链中恰好包含一个 A 中的元素 a，设这些链分别为 C_a^+，C_a^-，则 $\{C_a^- \cup \{a\} \cup C_a^+\}_{a\in A}$ 是 S 的一个最小链覆盖，命题得证。

"Mirsky **定理**"

S 的最长链长度等于最小的反链覆盖数。

" 证明"

设 S 的最长链长度为 d，则由定义，最小反链覆盖数至少为 d.

令 $f(s)$ 为以 s 为最小元的最长链长度，注意到若 $f(s)=f(t)$，则 s 与 t 不可比，进而 $(\forall\ n \in \mathbf{N})$ $f^{-1}(\{n\})$ 均为反链，其中 $f^{-1}(\{n\}) := \{a \in S : f(a)=n\}$ 称为 水平集 （level set）[4]。

因此不难得出 $\{f^{-1}(\{i\}) : 1 \le i \le d\}$ 是一个反链覆盖，从而最小反链覆盖数至多为 d.

Dilworth 定理与 Hall 婚配定理 等价。

我们可以用 Dilworth 定理证明如下定理：

"Erdős–Szekeres **定理**"

含至少 $rs+1$ 个元素的实数序列 $\{a_i\}$ 要么有一个长为 $r+1$ 的不下降子序列，要么有一个长为 $s+1$ 的不上升子序列。

" 证明"

设序列长度为 $n \ge rs+1$，定义偏序集 $\{(i,a_i)\}_{i=1}^n$，其上的偏序 $\preceq$ 定义为：

$$(i,a_i) \preceq (j,a_j) \iff (i \le j \wedge a_i \le a_j)$$

假设该偏序集的宽度不超过 $s+1$，则由 Dilworth 定理可知该偏序集可以被至多 s 条链覆盖，若这些链的长度都不超过 r，则序列所含元素数至多为 rs，与条件矛盾。

例题

"Luogu P1020 [NOIP1999 **提高组**] **导弹拦截**[5]"

某国为了防御敌国的导弹袭击，发展出一种导弹拦截系统。但是这种导弹拦截系统有一个缺陷：虽然它的第一发炮弹能够到达任意的高度，但是以后每一发炮弹都不能高于前一发的高度。某天，雷达捕捉到敌国的导弹来袭。由于该系统还在试用阶段，所以只有一套系统，因此有可能不能拦截所有的导弹。

输入导弹依次飞来的高度，计算这套系统最多能拦截多少导弹，如果要拦截所有导弹最少要配备多少套这种导弹拦截系统。

对于全部数据，满足导弹的高度为正整数，且不超过 5×10^4.

" 题解"

令一共有 n 个导弹，第 i 个导弹的高度为 h_i，则集合 $\{(i,h_i)\}_{i=1}^n$ 为偏序集，其上的偏序 $\preceq$ 定义为：

$$(i,h_i) \preceq (j,h_j) \iff (i \le j \wedge h_i \ge h_j)$$

进而根据 Dilworth 定理有：**序列的不上升子序列的最少覆盖数等于最长上升子序列长度**。从而可以通过 最长不下降子序列的 $O(n\log n)$ 做法 解决本题。

" 参考代码"

```
#include <bits/stdc++.h>
using namespace std;

int main() {
  vector<int> a;
  int x;
  while (cin >> x) a.push_back(x);
  vector<int> f, g;
  for (int i : a) {
    if (f.empty() || -i >= f.back())
      f.push_back(-i);
    else
      *upper_bound(f.begin(), f.end(), -i) = -i;
    if (g.empty() || i > g.back())
      g.push_back(i);
    else
      *lower_bound(g.begin(), g.end(), i) = i;
  }
  cout << f.size() << '\n' << g.size() << '\n';
  return 0;
}
```

"[TJOI2015] 组合数学[6]"

给一个 n 行 m 列的网格图，其中每个格子中均有若干块财宝。每次从左上角出发，只能往右或下走，每次经过一个格子至多只能捡走一块财宝。问至少要走几次才可能把财宝全捡完。

$1 \le n \le 1000$，$1 \le m \le 1000$，每个格子中的财宝不超过 10^6 块。

" 题解"

不考虑网格图的点权，不难发现按给定的规则下在网格图上行走等价于在 DAG 上行走，从而我们可以将其视作 Hasse 图来构造偏序集，进而根据 Dilworth 定理有：**DAG 的最小链覆盖数等于最大的点独立集大小**。

因此本题所求即为给定网格图最大点权独立集的点权和。

令 a_{ij} 为网格图在点 (i,j) 处的权值，$f(i,j)$ 为从 (i,j) 到 $(1,m)$ 这个子网格中的答案，注意到每个点都和其右上角的点不相邻，则状态转移方程为：

$$f(i,j) = \max\{f(i-1,j), f(i,j+1), f(i-1,j+1)+a_{ij}\}$$

答案即为 $f(n,1)$.

" 参考代码"

```
#include <bits/stdc++.h>
using namespace std;

int main() {
  int t = 0;
  cin >> t;
  while (t--) {
    int n, m;
    cin >> n >> m;
```

```
    vector<vector<int64_t>> a(n, vector<int64_t>(m));
    for (auto &i : a)
      for (auto &j : i) cin >> j;
    vector<vector<int64_t>> f(n, vector<int64_t>(m));
    for (int i = 0; i < n; ++i)
      for (int j = m - 1; j >= 0; --j)
        f[i][j] =
            max({(i == 0 ? 0 : f[i - 1][j]), (j == m - 1 ? 0 : f[i][j + 1]),
                 (i == 0 || j == m - 1 ? 0 : f[i - 1][j + 1]) + a[i][j]});
    cout << f[n - 1][0] << '\n';
  }
  return 0;
}
```

习题

- [CTSC2008] 祭祀[7]
- CodeForces 590E Birthday[8]

C++ 中的应用

另请参阅：排序相关 STL - 算法基础。

C++ STL 中 需要使用比较的算法和数据结构[9] 中有序理论的应用。我们经常需要在 C++ 中自定义比较器，STL 要求[10] 其必须为**严格弱序**。令 $<$ 为自定义比较器，则可以定义：

- $x > y$ 为 $y < x$；
- $x \le y$ 为 $y \nless x$；
- $x \ge y$ 为 $x \nless y$；
- $x = y$ 为 $x \nless y \land y \nless x$.

参考资料与拓展阅读

1. Order theory - From Academic Kids[11]
2. Binary Relation - Wikipedia[12]
3. Order Theory - Wikipedia[13]
4. Hasse diagram - Wikipedia[14]
5. Directed set - Wikipedia[15]
6. Order Theory, Lecture Notes by Mark Dean for Decision Theory[16]
7. 卢开澄，卢华明，《组合数学》（第 3 版）[17], 2006
8. List of Order Theory Topics - Wikipedia[18]
9. 浅谈邻项交换排序的应用以及需要注意的问题 by ouuan[19]
10. One thing you should know about comparators—Strict Weak Ordering[20]
11. Dilworth’s theorem - Wikipedia[21]
12. Dilworth’s Theorem | Brilliant Math & Science Wiki[22]
13. Hall’s marriage theorem - Wikipedia[23]
14. Hall’s Marriage Theorem | Brilliant Math & Science Wiki[24]
15. Dilworth 学习笔记 - Selfish[25]

参考资料与注释

[1] Zorn 引理

[2] 选择公理

[3] 良序定理

[4] 水平集（level set）

[5] Luogu P1020 [NOIP1999 提高组] 导弹拦截

[6] [TJOI2015] 组合数学

[7] [CTSC2008] 祭祀

[8] CodeForces 590E Birthday

[9] 需要使用比较的算法和数据结构

[10] 要求

[11] Order theory - From Academic Kids

[12] Binary Relation - Wikipedia

[13] Order Theory - Wikipedia

[14] Hasse diagram - Wikipedia

[15] Directed set - Wikipedia

[16] Order Theory, Lecture Notes by Mark Dean for Decision Theory

[17] 《组合数学》（第 3 版）

[18] List of Order Theory Topics - Wikipedia

[19] 浅谈邻项交换排序的应用以及需要注意的问题 by ouuan

[20] One thing you should know about comparators—Strict Weak Ordering

[21] Dilworth's theorem - Wikipedia

[22] Dilworth's Theorem | Brilliant Math & Science Wiki

[23] Hall's marriage theorem - Wikipedia

[24] Hall's Marriage Theorem | Brilliant Math & Science Wiki

[25] Dilworth 学习笔记 - Selfish

9.23 杨氏矩阵

引入

杨氏矩阵 (Young tableau)，又名杨表，是一种常用于表示论和舒伯特演算中的组合对象。

杨表是一种特殊的矩阵。它便于对称群和一般线性群的群表示和性质研究。杨表由剑桥大学数学家阿尔弗雷德 · 杨（Alfred Young）于 1900 年首次提出，于 1903 年被德国数学家弗罗贝尼乌斯（Ferdinand Georg Frobenius）应用于对称群的研究。

" 注释"

表示论（Representation theory）是数学的一个分支。它通过将元素表示为向量空间的线性变换来研究抽象代数结构。**舒伯特演算**（Schubert calculus）是代数几何的一个分支，于 19 世纪由赫尔曼 · 舒伯特为了解决射影几何的计数问题而引入。

定义

杨图

杨图（Young diagram，使用点表示时又称 Ferrers 图[1]，在 分拆数 一节中有相关介绍）是一个有限的框或单元格集合，左对齐排列，行长按非递增顺序排列。如果把杨图每行的方格数列出，我们得到了一个非负整数 n（总方格数）的**整数分拆** (integer partition)λ。因此，我们可以将杨图的形状看作 λ，因为它携带与其整数拆分相同的信息。

杨图之间的包含关系定义了整数分拆上的一个 偏序 关系，此关系拥有 格 的结构，被称为**杨格** (Young's lattice)。如果把杨图各列的方格数列出，则会得到整数分拆 λ 的「共轭分拆」，或「转置分拆」，它所对应到的杨图可由原本的杨图沿主对角线作镜射对称而得。

杨图每个方格的位置由分别代表**行数**与**列数**的两个座标点决定。列的顺序由左向右，行的顺序则按方格数的由多向少的方向。此处需要注意，根据习惯不同存在着两种不同的杨图画法：第一个将方格数较少的行排在方格数较多的行的下方，第二种画法将各行由大到小一层一层往上叠。由于前一种画法主要由英语国家使用，而后者通常被法语国家使用，习惯上我们分别称它们为英式画法和法式画法。

以下表格中分别为整数分拆 $(5,4,1)$ 对应的杨图不同画法：

- 英式画法：
- 法式画法：

杨表

定义

杨表（Young tableau）是通过用取自某个字母表的符号填充杨氏图的框来获得的，这通常需要是一个全序集和。填入的元素写作 x_1,x_2,x_3,…。但为了方便起见，都直接填入正整数。

杨表最初应用于对称群的表示理论时，允许在杨图的 n 的方格中任意填入 1 到 n 中相异的正整数。但现在的研究大多使用「标准」的杨表，即上述条件中各行与各列中方格的数字皆为严格递增的。由 n 个方格的相异杨表数个数形成 对和数[2]：

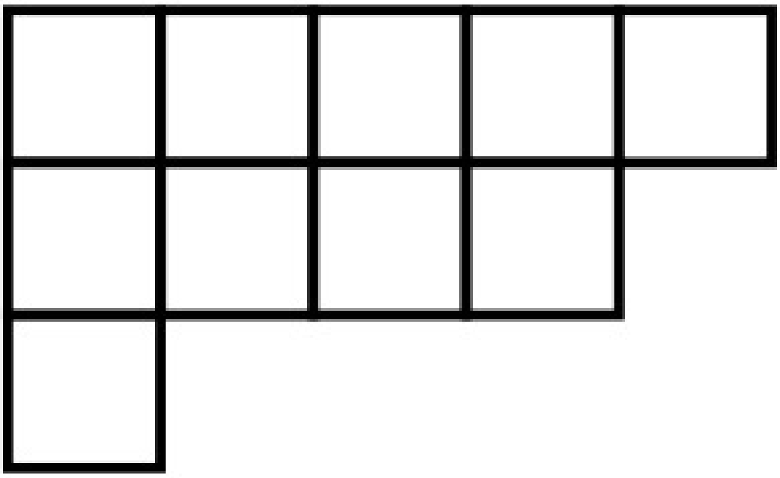


图 9.42

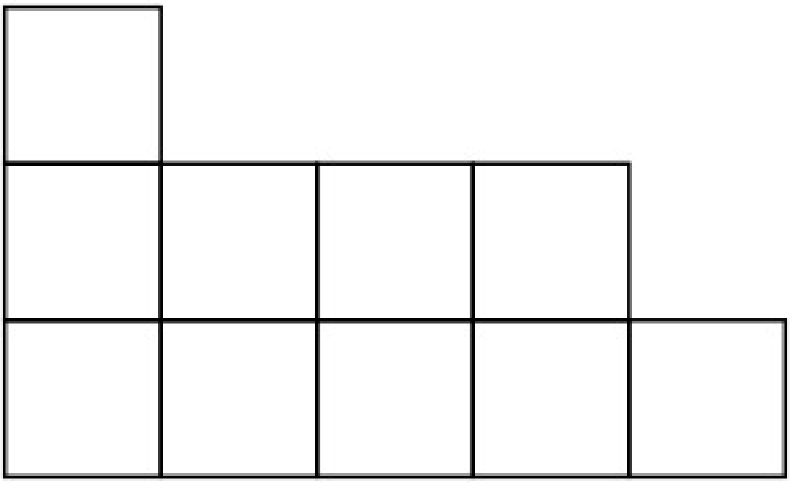


图 9.43

" 注释"

对和数（involution number/telephone number）是在数学中是一个整数序列，用来计算 n 条电话线中每条线路最多可以连接到另一条线路时可以相互连接的方法个数。它还可以用来描述完全图 n 个顶点上的匹配数，n 个对合元素的排列数，Hermite 多项式系数的绝对值之和，含有 n 个格子的标准杨表的个数，以及不可约对称群的度数之和。

$1, 1, 2, 4, 10, 26, 76, 232, 764, 2620, 9496, \ldots$（OEIS[3] 中的数列 A000085[4]）

在其他应用中，杨图也可以被填入相同的数字。若填法的同列数字严格递增，且同行数字单调递增，则该杨表被称为是**半标准的**（Semistandard Young Tableaux, 有时称为列严格）。杨表中个数字出现的次数记录下来得到的序列被视为杨表的**权重**。因此，标准杨表的权重必然是 $(1, 1, \ldots, 1)$。因为在标准杨表中，1 到 n 的每个正整数恰好各出现一次。

标准杨表的插入算法

排列的性质可以由杨表直观地表现出来。**RSK 插入算法**就提供了一个将杨表和排列联系起来的途径。它由 Robinson, Schensted 和 Knuth 提出。

令 S 是一个杨表，定义 $S \leftarrow x$ 表示将 x 从第一行插入杨表中，具体如下：

1. 在当前行中找到最小的比 x 大的数 y。
2. 如果找到了，用 x 去替换 y，移到下一行，令 $x \leftarrow y$ 重复操作 1。
3. 如果找不到，就把 x 放在该行末尾并退出。记 x 在第 s 行第 t 列，(s, t) 必定是一个边角。一个格子 (s, t) 是边角当且仅当 $(s+1, t)$ 和 $(s, t+1)$ 都不存在格子。

例如，将 3 插入杨表 $(2, 5, 9)(6, 7)(8)$ 的步骤为：

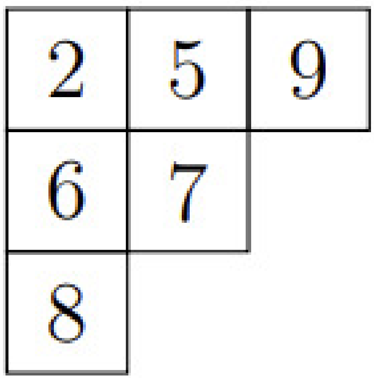


图 9.44

变体

非完全严格标准的杨表有许多变体（Variations）。例如行严格杨表要求同行数字严格递增，且同列数字单调递增，即列严格杨表的共轭。此外，在平面分拆（plane partitions）理论中，习惯上会将上述的定义中的递增改为递减。其他变体例如带状杨表，会先将一些方块打包成群，然后要求各群的方块必须填入相同数字。

斜杨表

给定两个杨图 $\lambda = (\lambda 1, \lambda 2 \ldots)$、$\mu = (\mu 1, \mu 2, \ldots)$，满足 λ 包含 μ，即 $\mu i \leq \mu i$ 对所有 i。定义「斜杨图」λ/μ 为 λ 中所有方格减去 μ 中的所有方格即 λ 差集 μ。在斜杨图的各方格中填入元素就形成了**斜杨表** (Skew tableaux)。

例如，下图为整数分拆 $(5, 4, 1)$ 对应的一个标准斜杨表：

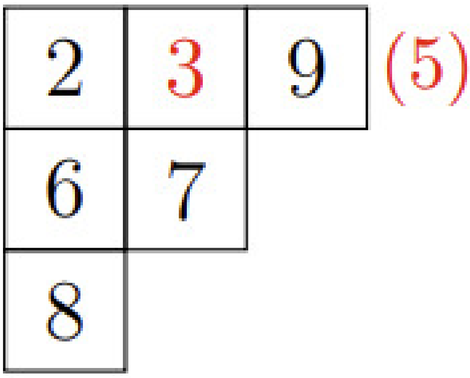


图 9.45

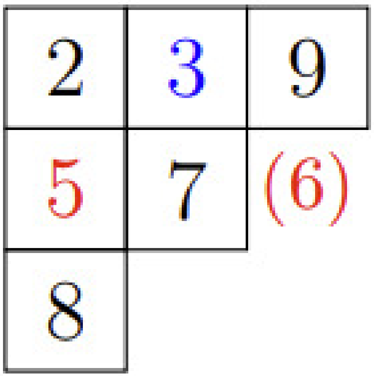


图 9.46

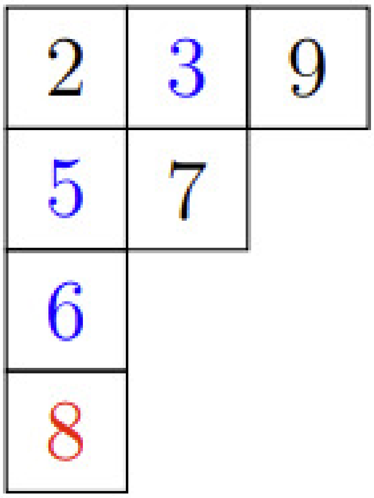


图 9.47

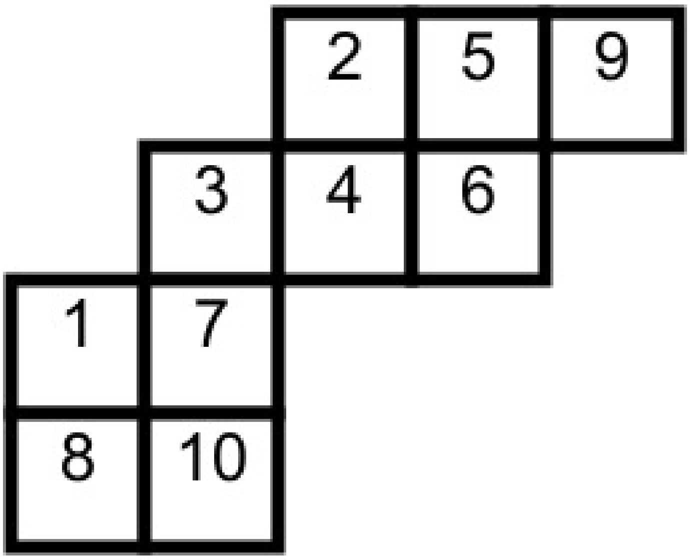


图 9.48

同理，若满足同一列中的数字严格递增，且同一行中的数字单调递增，则该斜杨表被称作**半标准斜杨表**；若半标准斜杨表满足各方格不重复的填入数字 1 到 n（方格总数），则该斜杨表被称作**标准斜杨表**。注意，由不同的 λ 和 μ 可得到相同的 λ/μ。虽然大部分斜杨表的性质都只依赖于取完差集的方格，但是仍然部分运算依赖于 λ 和 μ 的选取。因此，λ/μ 必须被视为包含两个元素信息：λ 和 μ。当 μ 是空分拆（0 的唯一一种分拆）时，斜杨表 λ/μ 就变成杨表 λ。

应用

杨表常用于在组合学、表示理论和代数几何中，用各种不同计算杨表个数的方法得到舒尔函数的定义及相关的恒等式。在信息学竞赛中，常有考察杨表钩长公式的题目。

勾长

给定一个共有 n 个方格的杨表 π_λ，把 1 到 n 的 n 个数字填入杨表中，使得每行从左到右，每列从下到上都是递增的。用 dim_{π_λ} 表示可以这样填的方法个数。

对于杨表中的一个方格 v，定义其**勾长** $\mathrm{hook}(v)$ 等于同行右边的方格数加上同列上面的方格数，再加 1（即方格本身）。

勾长公式

如果用 dim_λ 表示这样的方法个数，**勾长公式**就是方法个数等于 $n!$ 除以所有方格的勾长的乘积。

$$\dim \pi_\lambda = \frac{n!}{\prod_{x\in Y(\lambda)} \mathrm{hook}(x)}.$$

所以对于整数分拆 $10 = 5 + 4 + 1$ 的杨表，如上图所示。有

$$\dim \pi_\lambda = \frac{10!}{7\cdot 5\cdot 4\cdot 3\cdot 1\cdot 5\cdot 3\cdot 2\cdot 1\cdot 1} = 288.$$

种方法。

7	5	4	3	1
5	3	2	1	
1				

图 9.49

例题

子序列问题

对于杨表 P, 定义对于一个从 1 到 n 的排列 $X = x_1, \dots, x_n$。

1. P_X 中第一行的长度即为排列 X 的**最长上升子序列（LIS）**长度。注意，P 的第一行并不一定是 LIS 本身，所以不能直接利用杨表性质解决「LIS 划分」之类的问题。
2. 对于一个排列 X 和它产生的杨表 P_X，若 X^R 是 X 的翻转，那么 X^R 产生的杨表 P_{X^R} 即为 P_X 交换行列得到。例如，对于排列 $X = 1, 5, 7, 2, 8, 6, 3, 4$ 和 $X^R = 4, 3, 6, 8, 2, 7, 5, 1$, 我们可得到如下杨表 P_X:

$P_X =$

1	2	3	4
5	6	8	
7			

图 9.50

3. 杨表 P_X 中的第一列长度即为排列 X 的**最长下降子序列（LDS）**长度。

定义长度不超过 k 的 LIS/LDS 长度为 $k - LIS$ 和 $k - LDS$, 此类问题我们同样可以用杨表来解决。对于 $1 - LIS$，显而易见最长的 $1 - LIS$ 子序列就是该序列的 LDS，这也正是杨表的第一列；同样可得，杨表前 k 列的长度就是最长的 $k - LIS$ 子序列的长度。证明如下：

对于一个排列 X 和它的 m 行杨表 P，令排列 X^* 为 $(P_{m,1} \cdots, P_{m,\lambda_m}, P_{m-1,1} \cdots, P_{1,1} \cdots P_{1,\lambda_1})$（即将杨表从下往

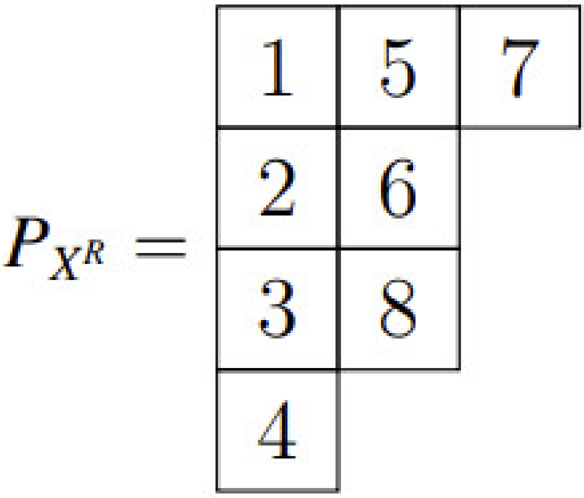


图 9.51

上每行依次写在后面)。那么 X 一定可以通过交换操作转化成 X^*。

所以，最长 $k-LIS$ 子序列长度可以表示成 $F(k)=\sum_{i=1}^{m}\min(k,\lambda_i)$，即前 k 列的长度和。

"CTSC2017 最长上升子序列[5]"

有一个长为 n 的数列 b。对于序列 $B_m=(b_1,b_2,\dots,b_m)$，设 C 是 B_m 的子序列，且 C 的最长上升子序列的长度不超过 k，询问 C 的长度最大值。

" 解题思路"

多个询问考虑使用扫描线的方法。这样我们就需要维护每个前缀的杨表。如果使用以上结论，可以发现问题变成了如何快速维护杨表前 k 列的长度之和。如果直接维护，复杂度是 $O(n^2\log n)$ 的不能接受。考虑维护前 $\sqrt{n}$ 列和前 $\sqrt{n}$ 行。

可以发现，杨表一定不会完全覆盖这个 $W\times H$ 的矩形。如果 $K\le W$，那么可以直接得答案；如果 $K>W$，那么大于 W 的部分一定在 H 行内。所以可以考虑如何同时维护前 $\sqrt{n}$ 列和前 $\sqrt{n}$ 行。将这个排列翻转一下就可以得到杨表的翻转，所以只需要再同时维护 $-A_i$ 即可，复杂度为 $O(n\sqrt{n}\log n)$。

"BJWC2018 最长上升子序列[6]"

现在有一个长度为 n 的随机排列，求它的最长上升子序列长度的期望。

"CF1268B 杨氏多米诺骨牌[7]"

给定一个具有 n 列长度 $a_1,a_2,\dots,a_n$ $(a_1\ge a_2\ge\dots\ge a_n\ge 1)$ 的直方图。$a=[3,2,2,2,1]$ 的杨图。找到可以在此直方图中绘制的最大数量的非重叠多米诺骨牌（1×2 或 2×1 矩形）。

参考资料与拓展阅读

1. Young Tableau - from Wolfram MathWorld[8]
2. Young tableau - Wikipedia[9]
3. Hook length formula - Wikipedia[10]
4. 袁方舟，《浅谈杨氏矩阵在信息学竞赛中的应用》IOI2019[11]，中国国家候选队论文集，202-229

参考资料与注释

[1] Ferrers 图

[2] 对和数

[3] OEIS

[4] A000085

[5] CTSC2017 最长上升子序列

[6] BJWC2018 最长上升子序列

[7] CF1268B 杨氏多米诺骨牌

[8] Young Tableau - from Wolfram MathWorld

[9] Young tableau - Wikipedia

[10] Hook length formula - Wikipedia

[11] 《浅谈杨氏矩阵在信息学竞赛中的应用》IOI2019

9.24 Schreier–Sims 算法

引入

Schreier–Sims 算法是计算群论的一种算法，以数学家 Otto Schreier 和 Charles Sims 的名字命名。它可以在多项式时间（polynomial time）内找到有限置换群的阶数、查看给定排列是否包含在群中以及其他许多任务。Schreier–Sims 算法最早由 Sims 在 1970 年基于 Schreier 的子群引理引入。1991 年，Donald Knuth 基于此改进了时序。后来，又有了该算法更快的随机版本。

" 注释"

群论（Group theory）是数学的一个分支。在数学和抽象代数中，群论研究称为群的代数结构。群是由一组元素和一个可以应用于该集合的两个元素的二元运算组成的系统。

背景

" 注释"

具体算法和时间复杂度分析请看之后章节，此节只是简单的背景介绍。

Schreier–Sims 算法是一种计算置换群的基强生成集（**BSGS**, base and strong generating set）的有效方法。特别是，SGS 确定了群的顺序并使查看给定排列是否包含在群中变得更加容易。由于 SGS 对于计算群论中的许多算法至关重要，因此计算机代数系统通常依赖 Schreier–Sims 算法进行群的有效计算。

Schreier–Sims 的运行时间因实现而异。令 $G \le S_n$ 由 t 个生成器给出。该算法可能的运行时间为：

- $O(n^2 \log^3 |G| + tn \log |G|)$ 需要 $O(n^2 \log |G| + tn)$ 的内存；
- $O(n^3 \log^3 |G| + tn^2 \log |G|)$ 需要 $O(n \log^2 |G| + tn)$ 的内存；

Schreier 向量的使用会对 Schreier–Sims 算法的实现性能产生重大影响。对于 Schreier–Sims 算法的 Monte Carlo 变体，预估复杂度如下：

- $O(n \log n \log^4 |G| + tn \log |G|)$ 需要 $O(n \log |G| + tn)$ 的内存；

现代计算机代数系统（例如 GAP 和 Magma）通常使用优化过的蒙特卡罗算法。

定义

定义任意群 G 是由一个子集 S 生成，$G = \langle S \rangle$, 对于所有 i,G 的每个元素都可以写成 $s_1, \cdots, s_r$ 与 $s_i \in S$ 或 $s_i^{-1} \in S$ 的乘积。

Schreier 树

对于 S 根为 α 的 **Schreier 树**是 α 轨道的如下表示：

- Schreier 树是一棵以 α 为根，以 α^G 的元素为顶点的树，
- 它的边描述了从 α 到每个顶点所需的 S 的元素，即树中的每条边 i, j，其中比 j 更靠近根的 i 由生成器 $s \in S$ 标记，将 i 移动到 j。
- Schreier 树可以通过**广度优先搜索**或**深度优先搜索**从 α 开始用所有生成器 $s \in S$ 尝试到达新的节点 α^s 来找到。因此，计算 Schreier 树所需的时间以 $O(rn)$ 为界。这意味着我们可以以一种有效的方式找到 $|\alpha^G|$。

Schreier 引理

Schreier 引理说明了如何使用 α 的 Schreier 树来查找稳定器 G_α 的生成器。

Schreier 引理：令 $G = \langle S \rangle$。那么由一组 Schreier 生成器生成的 α 稳定器 G_α：

$$G_\alpha = \langle t_i s t_{i^s}^{-1} | i \in \alpha^G, s \in S \rangle$$

其中 t_i 被定义为将 α 移到 i 的 G 中一个元素，即 i 的陪集（coset）代表。

强生成集

定义 G 的**基**（base）B 为一个序列 $B = (b_1, \cdots, b_k) \subseteq \Omega$, 因此逐点稳定器（pointwise stabilizer）$G_{b_1, \cdots, b_k}$ 是可以忽略不计的。

定义 G 相对于 B 的**强生成集**（SGS，strong generating set）是一个对于每个 i 有 $\langle S \cap G^{(i)} \rangle = G(i)$ 的集合 $S \subseteq G$，其中 $G^{(i)} := G_{b1, \cdots, bi}$,$G^{(0)} := G$。

过程

用来计算排列群（permutation group）G 阶数的 Schreier–Sims 算法一般有三种。它们同样也可以用来计算 G 的基和强生成集。

基础 Schreier–Sims 算法

1. 如果 G 为不平凡（non-trivial）群，选择一个尚未被选择的点 $b \in \Omega$。
2. 计算根为 b 的 Schreier 树，得到 $|b^G|$。
3. 使用 Schreier 引理找到 G_b 的生成器。

4. 对 G_b 递归使此算法，到找到 $|G_b|$ 为止。

当算法结束时，$(b_1, \cdots, b_m)$ 是一个基，找到的所有生成器的并集是一个强生成集。

基础 Schreier–Sims 算法的运行时间是**指数级**的，但可以被优化成更高效的算法。

增量 Schreier–Sims 算法

增量 Schreier–Sims 算法是常被用来构建强生成集的快速算法。

如果有一个群 G 的强生成集，因为已经得到了所有 $G^{(i)}$ 稳定器的生成器，那么很容易得到 G 的阶。**部分基**（partial base）$B = [b_1, \cdots, b_k]$ 和部分强生成集 S 是集合 $B \in \Omega$ 和集合 $S \subseteq G$，使得 S 的任何元素都不能固定 B 的每个元素。

增量 Schreier–Sims 算法可以将任意部分基和部分强生成集转化为基和强生成集。

定义 T_{i+1} 为通过 Schreier 树 $G^{(i)}$ 对 $G^{(i+1)}$ 的陪集的作用。

1. 如果 $S =$ ，返回 B, S;
2. 非空部分基 $B = (b_1, \cdots, b_k]$。部分强生成集 S。集 $C := [b_2, \cdots, b_k]$,$T := S \cap G_{b1}$，并递归地应用于输入 C, T，以将它们修改为 $H = \langle T \rangle$ 的基和强生成集。
3. 设 $B := B \cup C$,$S := S \cap T$。用筛选算法在 $H \leqslant G_{b_1}$ 中进行成员资格测试（Membership testing，检查集合（列表、集合、字典等）是否包含特定元素）。对 G_{b_1} 测试每个 Schreier 生成器 s 以查看 $s \in H$。如果都在 H 中，那么有 $H = G_{b_1}$，返回 B S。否则到步骤 4。
4. 否则有一个 Schreier 生成器 $s \in G_{b_1}$ 但 $s \notin H$。设 $S := S \cup s$。如果 s 固定了 B 的所有点，将一个由 s 移动的 Ω 点附加到 B。回到步骤 2。

当算法结束时，B 为基，S 是大小为 $O(n^2 \log n)$ 的强生成集。

增量 Schreier–Sims 算法的运行时间为 $O(n^8 \log^3 n)$，即 n 的多项式。t 个生成器构建 Schreier 树需要 $O(n^2 + nt)$，或对于 $t > n$ 为 $O(nt)$。因为已经用 $O(n^2 \log n)$ 限制了 Schreier 生成器 t 的数量，所以每个筛选过程都可以在 $nO(n(n^2 \log n)) = O(n^4 \log n)$ 中完成。

实现

以下为基础 Schreier–Sims 算法的参考代码。

" 参考代码"

```cpp
#include <iostream>
using namespace std;

const int maxn = 50;     // Maximum size of omega = {1, ,n}
const int maxr = 10000;  // Maximum number of generators

class Permutation {  // interface for permutations
 public:
  int p[maxn];  // the images of the points 0..   maxn-1

  Permutation() { n = maxn; };  // constructors

  Permutation(int m) { n = m; };

  Permutation(int m, char c) {
    n = m;
    switch (c) {
```

```
      case 'i':
        for (int i = 0; i < n; i++) p[i] = i;
        break;  // identity
      default:
        for (int i = 0; i < n; i++) p[i] = -1;
        break;  // undefined
    }
  }

  Permutation operator*(Permutation param) const {  // multiplication
    Permutation result(n);
    for (int i = 0; i < n; i++) result.p[i] = param.p[p[i]];
    return (result);
  }

  void operator*=(Permutation param) {  // direct multiplication
    for (int i = 0; i < n; i++) p[i] = param.p[p[i]];
  }

  Permutation inverse() const {  // inverse
    Permutation result(n);
    for (int i = 0; i < n; i++) result.p[p[i]] = i;
    return (result);
  }

  bool isdefined() const { return (p[0] > -1); }  // if it is     defined

  bool isidentity() const {  // if it is the     identity
    for (int i = 0; i < n; i++)
      if (p[i] != i) return false;
    return true;
  }

  bool operator==(Permutation param) const {  // comparison
    for (int i = 0; i < n; i++)
      if (param.p[i] != p[i]) return false;
    return true;
  }

  void input() {
    for (int i = 0; i < n; i++) {
      cin >> p[i];
      p[i]--;
    }
  }  // input

  void output() const {
    for (int i = 0; i < n; i++) cout << p[i] + 1 << " ";
    cout << endl;
  }  // output

  void setn(int m) { n = m; }

 private:
```

```
  int n;  // size of omega = {1, ,n}
};

int n;                                  // size of omega = {1, ,n}
int r;                                  // number of generators
Permutation* g = new Permutation[maxr];  // the generators
int nr;
Permutation* newg = new Permutation[maxr];
int cosreps;  // number of    (= size of orbit of alpha)
Permutation* cosrep =
    new Permutation[maxn];  // coset    representatives (to store the output of
                            // SchreierTree)
Permutation undefined(maxn, 'u');

/****** ScheierTree ******/
void ScheierTree(
    int alpha) {  // depth first search to determine the orbit of     alpha
  static Permutation gen(
      n, 'i');    // group element moving original alpha to actual    alpha
  static int ag;  // image of actual alpha under generator g[j]
  cosrep[alpha] = gen;
  cosreps++;
  for (int j = 0; j < r; j++) {
    ag = g[j].p[alpha];
    if (!cosrep[ag].isdefined()) {
      gen *= g[j];
      ScheierTree(ag);
      gen *= g[j].inverse();
    }
  }
}

void SchreierSims() {
  int alpha = 0;
  Permutation sg;
  cout << "THE ORDER OF THE GROUP:\n";
  do {
    cosreps = 0;
    for (int i = 0; i < n; i++) cosrep[i] = undefined;
    // get the coset representatives for G(alpha)
    ScheierTree(alpha);
    // schreier lemma loop to get the schreier generators
    nr = 0;
    for (int i = 0; i < n; i++) {
      if (cosrep[i].isdefined())
        for (int j = 0; j < r; j++) {
          // calculate the (schreier) generators
          sg = cosrep[i] * g[j] * cosrep[g[j].p[i]].inverse();
          bool alreadyhavethis = sg.isidentity();
          for (int k = 0; k < nr; k++)
            if (sg == newg[k]) alreadyhavethis = true;
          if (!alreadyhavethis) newg[nr++] = sg;
        }
    }
```

```
    cout << cosreps << flush;
    if (cosreps > 1) cout << "*";
    r = 0;
    for (int j = 0; j < nr; j++) {
      g[r++] = newg[j];
    }
    alpha++;
  } while (cosreps > 1);
  cout << endl;
}

int main() {
  cout << "n ( Size of Omega = {1..n} ) ? ";
  cin >> n;
  for (int j = 0; j < n; j++) {
    g[j].setn(n);
    newg[j].setn(n);
  }
  undefined.setn(n);

  cout << "How many group generators ? ";
  cin >> r;
  for (int j = 0; j < r; j++) g[j].input();

  SchreierSims();
  delete[] g;
  delete[] newg;
  delete[] cosrep;
  return 0;
}
```

例题

"Grand Prix of Yekaterinburg 2015 Problem H Heimdall[1]"

海姆达尔——阿斯加德最伟大的儿子之一，众神和世界之树的守护者。自古以来古他的主要职责就是守卫阿斯嘉德的入口——一座世界之间的桥梁。现存唯一古老的技术是将一定数量的桥梁结合起来，创造出一座穿越中间世界的桥梁。例如：如果第一座桥将物质从世界 A 传输到世界 B，第二座桥——从 B 到 C，那么它们的组合可以直接将物质从世界 A 传输到世界 C. 而且，这个古老的技术甚至可以让你自己结合一座桥。海姆达尔想知道——使用他所知道的桥梁以及它们的组合，可以创造出多少不同的桥梁。输入两个整数 R,N 分别是海姆达尔发现的桥梁总数和宇宙中的世界数（$1 \leqslant N \leqslant 15, 1 \leqslant R \leqslant 1000$）。接下来的 R 行包含这些桥的信息。每个桥由 N 个整数 $a_1, a_2, \cdots a_n$ 组成。其中 a_i 表示物质可以通过当前的桥梁转移到世界 i。如果当前的桥不影响那些世界，$a_i = i$。请输出一个可以通过古老技术建造的不同桥梁的总数。

参考资料与拓展阅读

1. Schreier–Sims algorithm - Wikipedia[2]
2. Knuth, Donald E，Efficient representation of perm groups[3], Combinatorica 11 (1991), no. 1, 33–43.
3. Ákos Seress,Permutation Group Algorithms[4], Cambridge University Press
4. Sims, Charles C,Computational methods in the study of permutation groups[5], Computational Problems in Abstract Algebra, pp. 169–183, Pergamon, Oxford, 1970.
5. Martin Jaggi,Implementations of 3 Types of the Schreier-Sims Algorithm[6], MAS334 - Mathematics Computing

Project, 2005

6. The Schreier-Sims algorithm for finite permutation groups[7]
7. Henrik B¨a¨arnhielm,The Schreier-Sims algorithm for matrix groups[8]

参考资料与注释

[1] Grand Prix of Yekaterinburg 2015 Problem H Heimdall

[2] Schreier–Sims algorithm - Wikipedia

[3] Efficient representation of perm groups

[4] Permutation Group Algorithms

[5] Computational methods in the study of permutation groups

[6] Implementations of 3 Types of the Schreier-Sims Algorithm

[7] The Schreier-Sims algorithm for finite permutation groups

[8] The Schreier-Sims algorithm for matrix groups

9.25 Berlekamp–Massey 算法

Authors: AntiLeaf

Berlekamp–Massey 算法是一种用于求数列的最短递推式的算法。给定一个长为 n 的数列，如果它的最短递推式的阶数为 m，则 Berlekamp–Massey 算法能够在 $O(nm)$ 时间内求出数列的每个前缀的最短递推式。最坏情况下 $m=O(n)$，因此算法的最坏复杂度为 $O(n^2)$。

定义

定义一个数列 $\{a_0 \dots a_{n-1}\}$ 的递推式为满足下式的序列 $\{r_0 \dots r_m\}$：

$\sum_{j=0}^{m} r_j a_{i-j} = 0, \forall i \ge m$

其中 $r_0 = 1$。m 称为该递推式的**阶数**。

数列 $\{a_i\}$ 的最短递推式即为阶数最小的递推式。

做法

与上面定义的稍有不同，这里定义一个新的递推系数 $\{f_0 \dots f_{m-1}\}$，满足：

$a_i = \sum_{j=0}^{m-1} f_j a_{i-j-1}, \forall i \ge m$

容易看出 $f_i = -r_{i+1}$，并且阶数 m 与之前的定义是相同的。

我们可以增量地求递推式，按顺序考虑 $\{a_i\}$ 的每一位，并在递推结果出现错误时对递推系数 $\{f_i\}$ 进行调整。方便起见，以下将前 i 位的最短递推式记为 $F_i = \{f_{i,j}\}$。

显然初始时有 $F_0 = \{\}$。假设递推系数 F_{i-1} 对数列 $\{a_i\}$ 的前 $i-1$ 项均成立，这时对第 i 项就有两种情况：

1. 递推系数对 a_i 也成立，这时不需要进行任何调整，直接令 $F_i = F_{i-1}$ 即可。

2. 递推系数对 a_i 不成立，这时需要对 F_{i-1} 进行调整，得到新的 F_i。

设 $\Delta_i = a_i - \sum_{j=0}^{m} f_{i-1,j} a_{i-j-1}$，即 a_i 与 F_{i-1} 的递推结果的差值。

如果这是第一次对递推系数进行修改，则说明 a_i 是序列中的第一个非零项。这时直接令 F_i 为 i 个 0 即可，显然这是一个合法的最短递推式。

否则设上一次对递推系数进行修改时，已考虑的 $\{a_i\}$ 的项数为 k。如果存在一个序列 $G = \{g_0 \cdots g_{m'-1}\}$，满足：$\sum_{j=0}^{m'-1} g_j a_{i'-j-1} = 0, \forall i' \in [m', i)$

并且 $\sum_{j=0}^{m'-1} g_j a_{i-j-1} = \Delta_i$，那么不难发现将 F_k 与 G 按位分别相加之后即可得到一个合法的递推系数 F_i。

考虑如何构造 G。一种可行的构造方案是令

$G = \{0, 0, \dots, 0, \frac{\Delta_i}{\Delta_k}, -\frac{\Delta_i}{\Delta_k} F_k\}$

其中前面一共有 $i-k-1$ 个 0，且最后的 $-\frac{\Delta_i}{\Delta_k} F_k$ 表示将 F_k 每项乘以 $-\frac{\Delta_i}{\Delta_k}$ 后接在序列后面。

不难验证此时 $\sum_{j=0}^{m'-1} g_j a_{i-j-1} = \Delta_k \frac{\Delta_i}{\Delta_k} = \Delta_i$，因此这样构造出的是一个合法的 G。将 F_i 赋值为 F_k 与 G 逐项相加后的结果即可。

如果要求的是符合最开始定义的递推式 $\{r_i\}$，则将 $\{f_j\}$ 全部取相反数后在最开始插入 $r_0 = 1$ 即可。

从上述算法流程中可以看出，如果数列的最短递推式的阶数为 m，则算法的复杂度为 $O(nm)$。最坏情况下 $m = O(n)$，因此算法的最坏复杂度为 $O(n^2)$。

在实现算法时，由于每次调整递推系数时都只需要用到上次调整时的递推系数 F_k，因此如果只需要求整个数列的最短递推式，可以只存储当前递推系数和上次调整时的递推系数，空间复杂度为 $O(n)$。

" 参考实现"

```cpp
vector<int> berlekamp_massey(const vector<int> &a) {
  vector<int> v, last;  // v is the answer, 0-based, p is the module
  int k = -1, delta = 0;

  for (int i = 0; i < (int)a.size(); i++) {
    int tmp = 0;
    for (int j = 0; j < (int)v.size(); j++)
      tmp = (tmp + (long long)a[i - j - 1] * v[j]) % p;

    if (a[i] == tmp) continue;

    if (k < 0) {
      k = i;
      delta = (a[i] - tmp + p) % p;
      v = vector<int>(i + 1);

      continue;
    }

    vector<int> u = v;
    int val = (long long)(a[i] - tmp + p) * power(delta, p - 2) % p;

    if (v.size() < last.size() + i - k) v.resize(last.size() + i - k);

    (v[i - k - 1] += val) %= p;

    for (int j = 0; j < (int)last.size(); j++) {
      v[i - k + j] = (v[i - k + j] - (long long)val * last[j]) % p;
      if (v[i - k + j] < 0) v[i - k + j] += p;
```

```
    }

    if ((int)u.size() - i < (int)last.size() - k) {
      last = u;
      k = i;
      delta = a[i] - tmp;
      if (delta < 0) delta += p;
    }
  }

  for (auto &x : v) x = (p - x) % p;
  v.insert(v.begin(), 1);

  return v;  // $\forall i, \sum_{j = 0} ^ m a_{i - j} v_j = 0$
}
```

朴素的 Berlekamp–Massey 算法求解的是有限项数列的最短递推式。如果待求递推式的序列有无限项，但已知最短递推式的阶数上界，则只需取出序列的前 $2m$ 项即可求出整个序列的最短递推式。（证明略）

应用

由于 Berlekamp–Massey 算法的数值稳定性比较差，在处理实数问题时一般很少使用。为了叙述方便，以下均假定在某个质数 p 的剩余系下进行运算。

求向量列或矩阵列的最短递推式

如果要求向量列 v_i 的最短递推式，设向量的维数为 n，我们可以随机一个 n 维行向量 $\mathbf{u}^T$，并计算标量序列 $\{u^T v_i\}$ 的最短递推式。由 Schwartz–Zippel 引理，二者的最短递推式有至少 $1 - \frac{n}{p}$ 的概率相同。

求矩阵列 $\{A_i\}$ 的最短递推式也是类似的，设矩阵的大小为 $n \times m$，则只需随机一个 $1 \times n$ 的行向量 $\mathbf{u}^T$ 和一个 $m \times 1$ 的列向量 v，并计算标量序列 $\{u^T A_i v\}$ 的最短递推式即可。由 Schwartz–Zippel 引理可以类似地得到二者相同的概率至少为 $1 - \frac{n+m}{p}$。

优化矩阵快速幂

设 f_i 是一个 n 维列向量，并且转移满足 $f_i = Af_{i-1}$，则可以发现 $\{f_i\}$ 是一个不超过 n 阶的线性递推向量列。（证明略）

我们可以直接暴力求出 $f_0 \dots f_{2n-1}$，然后用前面提到的做法求出 $\{f_i\}$ 的最短递推式，再调用 常系数齐次线性递推 即可。

如果要求的向量是 f_m，则算法的复杂度是 $O(n^3 + n \log n \log m)$。如果 A 是一个只有 k 个非零项的稀疏矩阵，则复杂度可以降为 $O(nk + n \log n \log m)$。但由于算法至少需要 $O(nk)$ 的时间预处理，因此在压力不大的情况下也可以使用 $O(n^2 \log m)$ 的线性递推算法，复杂度同样是可以接受的。

求矩阵的最小多项式

方阵 A 的最小多项式是次数最小的并且满足 $f(A) = 0$ 的多项式 f。

实际上最小多项式就是 $\{A^i\}$ 的最小递推式，所以直接调用 Berlekamp–Massey 算法就可以了。如果 A 是一个 n 阶方阵，则显然最小多项式的次数不超过 n。

瓶颈在于求出 A^i，因为如果直接每次做矩阵乘法的话复杂度会达到 $O(n^4)$。但考虑到求矩阵列的最短递推式时实际上求的是 $\{u^T A^i v\}$ 的最短递推式，因此我们只要求出 $A^i v$ 就行了。

假设 A 有 k 个非零项，则复杂度为 $O(kn + n^2)$。

求稀疏矩阵行列式

如果能求出方阵 A 的特征多项式，则常数项乘上 $(-1)^n$ 就是行列式。但是最小多项式不一定就是特征多项式。

实际上如果把 A 乘上一个随机对角阵 B，则 AB 的最小多项式有至少 $1-\frac{2n^2-n}{p}$ 的概率就是特征多项式。最后再除掉 $\det B$ 就行了。

设 A 为 n 阶方阵，且有 k 个非零项，则复杂度为 $O(kn+n^2)$。

求稀疏矩阵的秩

设 A 是一个 $n\times m$ 的矩阵，首先随机一个 $n\times n$ 的对角阵 P 和一个 $m\times m$ 的对角阵 Q, 然后计算 $QAPA^TQ$ 的最小多项式即可。

实际上不用调用矩阵乘法，因为求最小多项式时要用 $QAPA^TQ$ 乘一个向量，所以我们依次把这几个矩阵乘到向量里就行了。答案就是最小多项式除掉所有 x 因子后剩下的次数。

设 A 有 k 个非零项，且 $n\le m$，则复杂度为 $O(kn+n^2)$。

解稀疏方程组

问题：已知 $A\mathbf{x}=\mathbf{b}$, 其中 A 是一个 $n\times n$ 的**满秩**稀疏矩阵，$\mathbf{b}$ 和 $\mathbf{x}$ 是 $1\times n$ 的列向量。$A,\mathbf{b}$ 已知，需要在低于 n^ω 的复杂度内解出 x。

做法：显然 $\mathbf{x}=A^{-1}\mathbf{b}$。如果我们能求出 $\{A^i\mathbf{b}\}(i\ge 0)$ 的最小递推式 $\{r_0\dots r_{m-1}\}(m\le n)$, 那么就有结论

$A^{-1}\mathbf{b}=-\frac{1}{r_{m-1}}\sum_{i=0}^{m-2}A^i\mathbf{b}r_{m-2-i}$

（证明略）

因为 A 是稀疏矩阵，直接按定义递推出 $\mathbf{b}\dots A^{2n-1}\mathbf{b}$ 即可。

同样地，设 A 中有 k 个非零项，则复杂度为 $O(kn+n^2)$。

" 参考实现"

```cpp
vector<int> solve_sparse_equations(const vector<tuple<int, int, int> > &A,
                                   const vector<int> &b) {
  int n = (int)b.size();  // 0-based

  vector<vector<int> > f({b});

  for (int i = 1; i < 2 * n; i++) {
    vector<int> v(n);
    auto &u = f.back();

    for (auto [x, y, z] : A)  // [x, y, value]
      v[x] = (v[x] + (long long)u[y] * z) % p;

    f.push_back(v);
  }

  vector<int> w(n);
  mt19937 gen;
  for (auto &x : w) x = uniform_int_distribution<int>(1, p - 1)(gen);

  vector<int> a(2 * n);
  for (int i = 0; i < 2 * n; i++)
    for (int j = 0; j < n; j++) a[i] = (a[i] + (long long)f[i][j] * w[j]) % p;

  auto c = berlekamp_massey(a);
  int m = (int)c.size();

  vector<int> ans(n);
```

```
  for (int i = 0; i < m - 1; i++)
    for (int j = 0; j < n; j++)
      ans[j] = (ans[j] + (long long)c[m - 2 - i] * f[i][j]) % p;

  int inv = power(p - c[m - 1], p - 2);

  for (int i = 0; i < n; i++) ans[i] = (long long)ans[i] * inv % p;

  return ans;
}
```

例题

1. LibreOJ #163. 高斯消元 2[1]
2. ICPC2021 台北 Gym103443E. Composition with Large Red Plane, Yellow, Black, Gray, and Blue[2]

参考资料与注释

[1] LibreOJ #163. 高斯消元 2

[2] ICPC2021 台北 Gym103443E. Composition with Large Red Plane, Yellow, Black, Gray, and Blue

第 10 章

数据结构

10.1 数据结构部分简介

Authors: HeRaNO, Zhoier, hsfzLZH1

数据结构是在计算机中存储、组织数据的方式。小到变量、数组，大到线段树、平衡树，都是数据结构。

程序运行离不开数据结构，不同的数据结构又各有优劣，能够处理的问题各不相同，而根据具体问题选取合适的数据结构，可以大大提升程序的效率。所以，学习各种各样的数据结构是很有必要的。

10.2 栈

引入

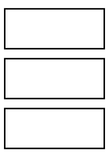


图 10.1

栈是 OI 中常用的一种线性数据结构。请注意，本文主要讲的是栈这种数据结构，而非程序运行时的系统栈/栈空间。

栈的修改与访问是按照后进先出的原则进行的，因此栈通常被称为是后进先出（last in first out）表，简称 LIFO 表。

warning

LIFO 表达的是**当前在容器**内最后进来的最先出去。

我们考虑这样一个栈

```
push(1)
```

```
pop(1)
push(2)
pop(2)
```

如果从整体考虑，1 最先入栈，最先出栈，2 最后入栈，最后出栈，这样就成了一个先进先出表，显然是错误的。

所以，在考虑数据结构是 LIFO 还是 FIFO 的时候，应当考虑在当前容器内的情况。

使用数组模拟栈

我们可以方便的使用数组来模拟一个栈，如下：

" 实现"

```cpp
int st[N];
// 这里使用 st[0] (即 *st) 代表栈中元素数量，同时也是栈顶下标

// 压栈 ：
st[++*st] = var1;
// 取栈顶 ：
int u = st[*st];
// 弹栈 ：注意越界问题, *st == 0 时不能继续弹出
if (*st) --*st;
// 清空栈
*st = 0;
```

```python
st = [0] * N
# 这里使用 st[0] 代表栈中元素数量，同时也是栈顶下标

# 压栈 ：
st[st[0] + 1] = var1
st[0] = st[0] + 1
# 取栈顶：
u = st[st[0]]
# 弹栈：注意越界问题, *st == 0 时不能继续弹出
if st[0]:
    st[0] = st[0] - 1
# 清空栈
st[0] = 0
```

C++ STL 中的栈

C++ 中的 STL 也提供了一个容器 `std::stack`，使用前需要引入 `stack` 头文件。

"STL 中对 stack 的定义"

```cpp
// clang-format off
template<
    class T,
    class Container = std::deque<T>
> class stack;
```

`T` 为 stack 中要存储的数据类型。

`Container` 为用于存储元素的底层容器类型。这个容器必须提供通常语义的下列函数：

- back()
- push_back()
- pop_back()

STL 容器 std::vector、std::deque 和 std::list 满足这些要求。如果不指定，则默认使用 std::deque 作为底层容器。

STL 中的 stack 容器提供了一众成员函数以供调用，其中较为常用的有：

- 元素访问
  - st.top() 返回栈顶
- 修改
  - st.push() 插入传入的参数到栈顶
  - st.pop() 弹出栈顶
- 容量
  - st.empty() 返回是否为空
  - st.size() 返回元素数量

此外，std::stack 还提供了一些运算符。较为常用的是使用赋值运算符 = 为 stack 赋值，示例：

```cpp
// 新建两个栈 st1 和 st2
std::stack<int> st1, st2;

// 为 st1 装入 1
st1.push(1);

// 将 st1 赋值给 st2
st2 = st1;

// 输出 st2 的栈顶元素
cout << st2.top() << endl;
// 输出：1
```

使用 Python 中的 list 模拟栈

在 Python 中，你可以使用列表来模拟一个栈：

" 实现"

```python
st = [5, 1, 4]

# 使用 append() 向栈顶添加元素
st.append(2)
st.append(3)
# >>> st
# [5, 1, 4, 2, 3]

# 使用 pop 取出栈顶元素
st.pop()
# >>> st
# [5, 1, 4, 2]
```

```
# 使用 clear 清空栈
st.clear()
```

参考资料

1. std::stack - zh.cppreference.com[1]

参考资料与注释

[1] std::stack - zh.cppreference.com

10.3 队列

本页面介绍和队列有关的数据结构及其应用。

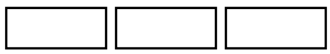


图 10.2

引入

队列（queue）是一种具有「先进入队列的元素一定先出队列」性质的表。由于该性质，队列通常也被称为先进先出（first in first out）表，简称 FIFO 表。

数组模拟队列

通常用一个数组模拟一个队列，用两个变量标记队列的首尾。

```
int q[SIZE], ql = 1, qr;
```

队列操作对应的代码如下：

- 插入元素：`q[++qr] = x;`
- 删除元素：`ql++;`
- 访问队首：`q[ql]`
- 访问队尾：`q[qr]`
- 清空队列：`ql = 1; qr = 0;`

双栈模拟队列

还有一种冷门的方法是使用两个 栈 来模拟一个队列。

这种方法使用两个栈 F, S 模拟一个队列，其中 F 是队尾的栈，S 代表队首的栈，支持 push（在队尾插入），pop（在队首弹出）操作：

- push：插入到栈 F 中。
- pop：如果 S 非空，让 S 弹栈；否则把 F 的元素倒过来压到 S 中（其实就是一个一个弹出插入，做完后是首尾颠倒的），然后再让 S 弹栈。

容易证明，每个元素只会进入/转移/弹出一次，均摊复杂度 $O(1)$。

C++ STL 中的队列

C++ 在 STL 中提供了一个容器 `std::queue`，使用前需要先引入 `<queue>` 头文件。

"STL 中对 queue 的定义"

```
// clang-format off
template<
    class T,
    class Container = std::deque<T>
> class queue;
```

T 为 queue 中要存储的数据类型。

Container 为用于存储元素的底层容器类型。这个容器必须提供通常语义的下列函数：

- back()
- front()
- push_back()
- pop_front()

STL 容器 `std::deque` 和 `std::list` 满足这些要求。如果不指定，则默认使用 `std::deque` 作为底层容器。

STL 中的 `queue` 容器提供了一众成员函数以供调用。其中较为常用的有：

- 元素访问
  - `q.front()` 返回队首元素
  - `q.back()` 返回队尾元素
- 修改
  - `q.push()` 在队尾插入元素
  - `q.pop()` 弹出队首元素
- 容量
  - `q.empty()` 队列是否为空
  - `q.size()` 返回队列中元素的数量

此外，`queue` 还提供了一些运算符。较为常用的是使用赋值运算符 `=` 为 `queue` 赋值，示例：

```
std::queue<int> q1, q2;

// 向 q1 的队尾插入 1
q1.push(1);

// 将 q1 赋值给 q2
q2 = q1;

// 输出 q2 的队首元素
std::cout << q2.front() << std::endl;
// 输出： 1
```

特殊队列

双端队列

双端队列是指一个可以在队首/队尾插入或删除元素的队列。相当于是栈与队列功能的结合。具体地，双端队列支持的操作有 4 个：

- 在队首插入一个元素
- 在队尾插入一个元素
- 在队首删除一个元素
- 在队尾删除一个元素

数组模拟双端队列的方式与普通队列相同。

C++ STL 中的双端队列

C++ 在 STL 中也提供了一个容器 `std::deque`，使用前需要先引入 `<deque>` 头文件。

"STL 中对 deque 的定义"

```cpp
// clang-format off
template<
    class T,
    class Allocator = std::allocator<T>
> class deque;
```

`T` 为 deque 中要存储的数据类型。

`Allocator` 为分配器，此处不做过多说明，一般保持默认即可。

STL 中的 `deque` 容器提供了一众成员函数以供调用。其中较为常用的有：

- 元素访问
  - `q.front()` 返回队首元素
  - `q.back()` 返回队尾元素
- 修改
  - `q.push_back()` 在队尾插入元素
  - `q.pop_back()` 弹出队尾元素
  - `q.push_front()` 在队首插入元素
  - `q.pop_front()` 弹出队首元素
  - `q.insert()` 在指定位置前插入元素（传入迭代器和元素）
  - `q.erase()` 删除指定位置的元素（传入迭代器）
- 容量
  - `q.empty()` 队列是否为空
  - `q.size()` 返回队列中元素的数量

此外，`deque` 还提供了一些运算符。其中较为常用的有：

- 使用赋值运算符 `=` 为 `deque` 赋值，类似 `queue`。
- 使用 `[]` 访问元素，类似 `vector`。

`<queue>` 头文件中还提供了优先队列 `std::priority_queue`，因其与 堆 更为相似，在此不作过多介绍。

Python 中的双端队列

在 Python 中，双端队列的容器由 `collections.deque` 提供。

示例如下：

" 实现"

```python
from collections import deque

# 新建一个 deque，并初始化内容为 [1, 2, 3]
queue = deque([1, 2, 3])

# 在队尾插入元素 4
queue.append(4)

# 在队首插入元素 0
queue.appendleft(0)

# 访问队列
# >>> queue
# deque([0, 1, 2, 3, 4])
```

循环队列

使用数组模拟队列会导致一个问题：随着时间的推移，整个队列会向数组的尾部移动，一旦到达数组的最末端，即使数组的前端还有空闲位置，再进行入队操作也会导致溢出（这种数组里实际有空闲位置而发生了上溢的现象被称为「假溢出」）。

解决假溢出的办法是采用循环的方式来组织存放队列元素的数组，即将数组下标为 0 的位置看做是最后一个位置的后继。（数组下标为 `x` 的元素，它的后继为 `(x + 1) % SIZE`）。这样就形成了循环队列。

例题

"LOJ6515「雅礼集训 2018 Day10」贪玩蓝月[1]"

一个双端队列（deque），m 个事件：

1. 在前端插入 (w,v)
2. 在后端插入 (w,v)
3. 删除前端的二元组
4. 删除后端的二元组
5. 给定 l,r，在当前 deque 中选择一个子集 S 使得 $\sum_{(w,v)\in S} w \bmod p \in [l,r]$，且最大化 $\sum_{(w,v)\in S} v$.
   $m \le 5\times 10^4, p \le 500$.

" 解题思路"

每个二元组是有一段存活时间的，因此对时间建立线段树，每个二元组做 log 个存活标记。因此我们要做的就是对每个询问，求其到根节点的路径上的标记的一个最优子集。显然这个可以 DP 做。$f[S,j]$ 表示选择集合 S 中的物品余数为 j 的最大价值。（其实实现的时侯是有序的，直接 f[i,j] 做）

一共有 $O(m\log m)$ 个标记，因此这么做的话复杂度是 $O(mp\log m)$ 的。

这是一个在线算法比离线算法快的神奇题目。而且还比离线的好写。

上述离线算法其实是略微小题大做的，因为如果把题目的 deque 改成直接维护一个集合的话（即随机删除集合内元素），那么离线算法同样适用。既然是 deque，不妨在数据结构上做点文章。

如果题目中维护的数据结构是一个栈呢？

直接 DP 即可。$f[i,j]$ 表示前 i 个二元组，余数为 j 时的最大价值。

$$f[i,j] = \max(f[i-1,j], f[i-1,(j-w_i) \bmod p] + v_i)$$

妥妥的背包啊。

删除的时侯直接指针前移即可。这样做的复杂度是 $O(mp)$ 的。

如果题目中维护的数据结构是队列？

有一种操作叫双栈模拟队列。这就是这个东西的用武之地。因为用栈是可以轻松维护 DP 过程的，而双栈模拟队列的复杂度是均摊 $O(1)$ 的，因此，复杂度仍是 $O(mp)$。

回到原题，那么 Deque 怎么做？

类比推理，我们尝试用栈模拟双端队列，于是似乎把维护队列的方法扩展一下就可以了。但如果每次是全部转移栈中的元素的话，单次操作复杂度很容易退化为 $O(m)$。

于是乎，神仙的想一想，我们可以丢一半过去啊。

这样的复杂度其实均摊下来仍是常数级别。具体地说，丢一半指的是把一个栈靠近栈底的一半倒过来丢到另一个栈中。也就是说要手写栈以支持这样的操作。

似乎可以用 势能分析法[2] 证明。其实本□□有一个很仙的想法。我们考虑这个双栈结构的整体复杂度。m 个事件，我们希望尽可能增加这个结构的复杂度。

首先，如果全是插入操作的话显然是严格 $\Theta(m)$ 的，因为插入的复杂度是 $O(1)$ 的。

「丢一半」操作是在什么时侯触发的？当某一个栈为空又要求删除元素的时侯。设另一个栈的元素个数是 $O(k)$，那么丢一半的复杂度就是 $O(k) \geq O(1)$ 的。因此我们要尽可能增加「丢一半」操作的次数。

为了增加丢一半的操作次数，必然需要不断删元素直到某一个栈为空。由于插入操作对增加复杂度是无意义的，因此我们不考虑插入操作。初始时有 m 个元素，假设全在一个栈中。则第一次丢一半的复杂度是 $O(m)$ 的。然后两个栈就各有 $\frac{m}{2}$ 个元素。这时就需要 $O(\frac{m}{2})$ 删除其中一个栈，然后就又可以触发一次复杂度为 $O(\frac{m}{2})$ 的丢一半操作……

考虑这样做的总复杂度。

$$T(m) = 2 \cdot O(m) + T\left(\frac{m}{2}\right)$$

解得 $T(m) = O(m)$。

于是，总复杂度仍是 $O(mp)$。

在询问的时侯，我们要处理的应该是「在两个栈中选若干个元素的最大价值」的问题。因此要对栈顶的 DP 值做查询，即两个 f, g 对于询问 [l,r] 的最大价值：

$$\max_{0 \leq i < p} \left\{ f[i] + \max_{l \leq i+j \leq r} g_j \right\}$$

这个问题暴力做是 $O(p^2)$ 的，不过一个妥妥的单调队列可以做到 $O(p)$。

" 参考代码"

```cpp
#include <algorithm>
#include <cctype>
#include <cmath>
#include <cstdio>
```

```cpp
#include <cstdlib>
#include <cstring>
#include <iostream>
#include <map>
#include <queue>
#include <set>
#include <vector>
using namespace std;
/*****************heading******************/
const int M = 5e4 + 5, P = 505;  // 定义常数
int I, m, p;

int _(int d) { return (d + p) % p; }  // 用于取模

namespace DQ {                // 双栈模拟双端队列
pair<int, int> fr[M], bc[M];  // 二元组，详见题目 3.4
int tf = 0, tb = 0;           // 一端的 top，因为是双端队列所以有俩
int ff[M][P], fb[M][P];

void update(pair<int, int> *s, int f[][P], int i) {  // 用 f[i-1] 更新 f[i]
  for (int j = 0; j <= (p - 1); j++) {
    f[i][j] = f[i - 1][j];
    if (~f[i - 1][_(j - s[i].first)])  // 按位取反
      f[i][j] = max(f[i][j], f[i - 1][_(j - s[i].first)] + s[i].second);
  }
}

// 以下两行代码表示 push 入队列，很好理解
void push_front(pair<int, int> x) { fr[++tf] = x, update(fr, ff, tf); }

void push_back(pair<int, int> x) { bc[++tb] = x, update(bc, fb, tb); }

// 以下两行代码表示从队列 pop 出元素
void pop_front() {
  if (tf) {
    --tf;
    return;
  }
  int mid = (tb + 1) / 2, top = tb;
  for (int i = mid; i >= 1; i--) push_front(bc[i]);
  tb = 0;
  for (int i = (mid + 1); i <= top; i++) push_back(bc[i]);
  --tf;
  // 上面的代码，逻辑和普通队列是一样的
}

void pop_back() {
  if (tb) {
    --tb;
    return;
  }
  int mid = (tf + 1) / 2, top = tf;
  for (int i = mid; i >= 1; i--) push_back(fr[i]);
  tf = 0;
```

```
  for (int i = (mid + 1); i <= top; i++) push_front(fr[i]);
  --tb;
  // 上面的代码，逻辑和普通队列是一样的
}

int q[M], ql, qr;  // 题目任务 5 要求的

int query(int l, int r) {
  const int *const f = ff[tf], *const g = fb[tb];
  int ans = -1;
  ql = 1, qr = 0;
  for (int i = (l - p + 1); i <= (r - p + 1); i++) {
    int x = g[_(i)];
    while (ql <= qr && g[q[qr]] <= x) --qr;
    q[++qr] = _(i);
  }
  for (int i = (p - 1); i >= 0; i--) {
    if (ql <= qr && ~f[i] && ~g[q[ql]]) ans = max(ans, f[i] + g[q[ql]]);
    // 删 l-i，加 r-i+1
    if (ql <= qr && _(l - i) == q[ql]) ++ql;
    int x = g[_(r - i + 1)];
    while (ql <= qr && g[q[qr]] <= x) --qr;
    q[++qr] = _(r - i + 1);
  }
  return ans;
}

void init() {
  for (int i = 1; i <= (P - 1); i++) ff[0][i] = fb[0][i] = -1;
}  // 初始化
}  // namespace DQ

int main() {
  DQ::init();
  scanf("%d%d%d", &I, &m, &p);
  for (int i = 1; i <= m; i++) {
    char op[5];
    int x, y;
    scanf("%s%d%d", op, &x, &y);
    if (op[0] == 'I' && op[1] == 'F')
      DQ::push_front(make_pair(_(x), y));
    else if (op[0] == 'I' && op[1] == 'G')
      DQ::push_back(make_pair(_(x), y));
    else if (op[0] == 'D' && op[1] == 'F')
      DQ::pop_front();
    else if (op[0] == 'D' && op[1] == 'G')
      DQ::pop_back();
    else
      printf("%d\n", DQ::query(x, y));
  }
  return 0;
}

/* example.in
```

```
0
11 10
QU 0 0
QU 1 9
IG 14 7
IF 3 5
QU 0 9
IG 1 8
DF
QU 0 4
IF 1 2
DG
QU 2 9
 */
/* example.out
0
-1
12
8
9
 */
/* LOJ:https://loj.ac/s/1149797*/
```

参考资料

1. std::queue - zh.cppreference.com[3]
2. std::deque - zh.cppreference.com[4]

参考资料与注释

[1] LOJ6515「雅礼集训 2018 Day10」贪玩蓝月

[2] 势能分析法

[3] std::queue - zh.cppreference.com

[4] std::deque - zh.cppreference.com

10.4 链表

本页面将简要介绍链表。

引入

链表是一种用于存储数据的数据结构，通过如链条一般的指针来连接元素。它的特点是插入与删除数据十分方便，但寻找与读取数据的表现欠佳。

与数组的区别

链表和数组都可用于存储数据。与链表不同，数组将所有元素按次序依次存储。不同的存储结构令它们有了不同的优势：

链表因其链状的结构，能方便地删除、插入数据，操作次数是 $O(1)$。但也因为这样，寻找、读取数据的效率不如数组高，在随机访问数据中的操作次数是 $O(n)$。

数组可以方便地寻找并读取数据，在随机访问中操作次数是 $O(1)$。但删除、插入的操作次数是 $O(n)$ 次。

构建链表

tip

构建链表时，使用指针的部分比较抽象，光靠文字描述和代码可能难以理解，建议配合作图来理解。

单向链表

单向链表中包含数据域和指针域，其中数据域用于存放数据，指针域用来连接当前结点和下一节点。

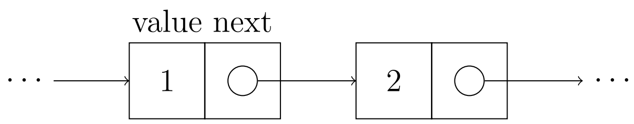


图 10.3

" 实现"

```
struct Node {
  int value;
  Node *next;
};

class Node:
    def __init__(self, value = None, next = None):
        self.value = value
        self.next = next
```

双向链表

双向链表中同样有数据域和指针域。不同之处在于，指针域有左右（或上一个、下一个）之分，用来连接上一个结点、当前结点、下一个结点。

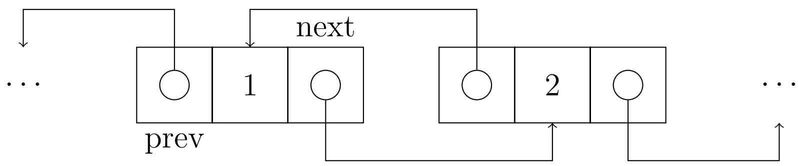


图 10.4

" 实现"

```
struct Node {
  int value;
  Node *left;
  Node *right;
};

class Node:
    def __init__(self, value = None, left = None, right = None):
        self.value = value
        self.left = left
        self.right = right
```

向链表中插入（写入）数据

单向链表

流程大致如下：

1. 初始化待插入的数据 `node`；
2. 将 `node` 的 `next` 指针指向 `p` 的下一个结点；
3. 将 `p` 的 `next` 指针指向 `node`。

具体过程可参考下图：

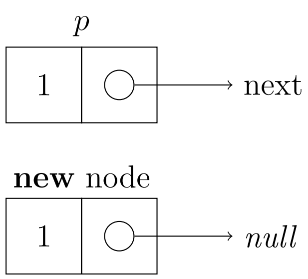


图 10.5

1.

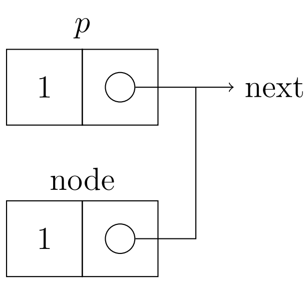


图 10.6

2.
3.

代码实现如下：

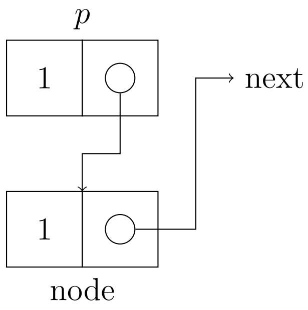


图 10.7

" 实现"

```
void insertNode(int i, Node *p) {
  Node *node = new Node;
  node->value = i;
  node->next = p->next;
  p->next = node;
}

def insertNode(i, p):
    node = Node()
    node.value = i
    node.next = p.next
    p.next = node
```

单向循环链表

将链表的头尾连接起来，链表就变成了循环链表。由于链表首尾相连，在插入数据时需要判断原链表是否为空：为空则自身循环，不为空则正常插入数据。

大致流程如下：

1. 初始化待插入的数据 `node`；
2. 判断给定链表 `p` 是否为空；
3. 若为空，则将 `node` 的 `next` 指针和 `p` 都指向自己；
4. 否则，将 `node` 的 `next` 指针指向 `p` 的下一个结点；
5. 将 `p` 的 `next` 指针指向 `node`。

具体过程可参考下图：

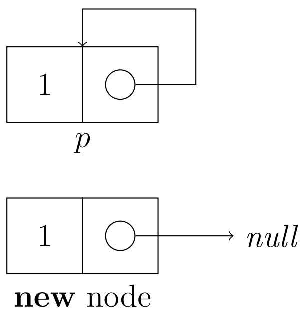


图 10.8

1.

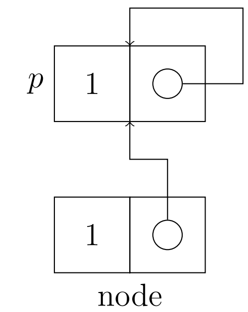


图 10.9

2.

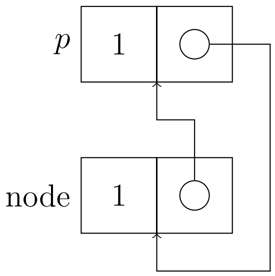


图 10.10

3.

代码实现如下：

" 实现"

```
void insertNode(int i, Node *p) {
  Node *node = new Node;
  node->value = i;
  node->next = NULL;
  if (p == NULL) {
    p = node;
    node->next = node;
  } else {
    node->next = p->next;
    p->next = node;
  }
}

def insertNode(i, p):
    node = Node()
    node.value = i
    node.next = None
    if p == None:
        p = node
        node.next = node
    else:
        node.next = p.next
        p.next = node
```

双向循环链表

在向双向循环链表插入数据时，除了要判断给定链表是否为空外，还要同时修改左、右两个指针。

大致流程如下：

1. 初始化待插入的数据 node；
2. 判断给定链表 p 是否为空；
3. 若为空，则将 node 的 left 和 right 指针，以及 p 都指向自己；
4. 否则，将 node 的 left 指针指向 p；
5. 将 node 的 right 指针指向 p 的右结点；
6. 将 p 右结点的 left 指针指向 node；
7. 将 p 的 right 指针指向 node。

代码实现如下：

" 实现"

```cpp
void insertNode(int i, Node *p) {
  Node *node = new Node;
  node->value = i;
  if (p == NULL) {
    p = node;
    node->left = node;
    node->right = node;
  } else {
    node->left = p;
    node->right = p->right;
    p->right->left = node;
    p->right = node;
  }
}
```

```python
def insertNode(i, p):
    node = Node()
    node.value = i
    if p == None:
        p = node
        node.left = node
        node.right = node
    else:
        node.left = p
        node.right = p.right
        p.right.left = node
        p.right = node
```

从链表中删除数据

单向（循环）链表

设待删除结点为 p，从链表中删除它时，将 p 的下一个结点 p->next 的值覆盖给 p 即可，与此同时更新 p 的下下个结点。

流程大致如下：

1. 将 p 下一个结点的值赋给 p，以抹掉 p->value；
2. 新建一个临时结点 t 存放 p->next 的地址；
3. 将 p 的 next 指针指向 p 的下下个结点，以抹掉 p->next；
4. 删除 t。此时虽然原结点 p 的地址还在使用，删除的是原结点 p->next 的地址，但 p 的数据被 p->next 覆盖，p 名存实亡。

具体过程可参考下图：

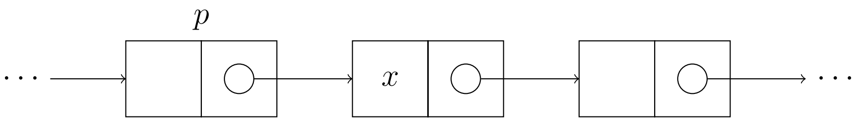


图 10.11

1.

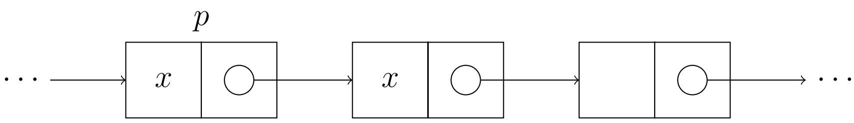


图 10.12

2.

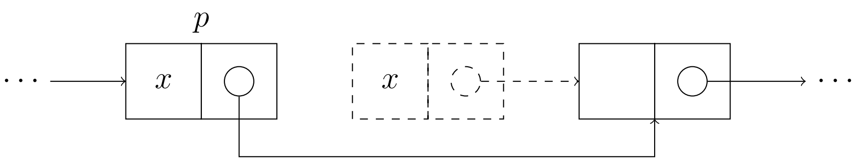


图 10.13

3.

代码实现如下：

"实现"

```
void deleteNode(Node *p) {
  p->value = p->next->value;
  Node *t = p->next;
  p->next = p->next->next;
  delete t;
}

def deleteNode(p):
    p.value = p.next.value
    p.next = p.next.next
```

双向循环链表

流程大致如下：

1. 将 p 左结点的右指针指向 p 的右节点；

2. 将 p 右结点的左指针指向 p 的左节点；
3. 新建一个临时结点 t 存放 p 的地址；
4. 将 p 的右节点地址赋给 p，以避免 p 变成悬垂指针；
5. 删除 t。

代码实现如下：

" 实现"

```
void deleteNode(Node *&p) {
  p->left->right = p->right;
  p->right->left = p->left;
  Node *t = p;
  p = p->right;
  delete t;
}

def deleteNode(p):
    p.left.right = p.right
    p.right.left = p.left
    p = p.right
```

技巧

异或链表

异或链表（XOR Linked List）本质上还是**双向链表**，但它利用按位异或的值，仅使用一个指针的内存大小便可以实现双向链表的功能。

我们在结构 `Node` 中定义 `lr = left ^ right`，即前后两个元素地址的**按位异或值**。正向遍历时用前一个元素的地址异或当前节点的 `lr` 可得到后一个元素的地址，反向遍历时用后一个元素的地址异或当前节点的 `lr` 又可得到前一个的元素地址。这样一来，便可以用一半的内存实现双向链表同样的功能。

10.5 哈希表

引入

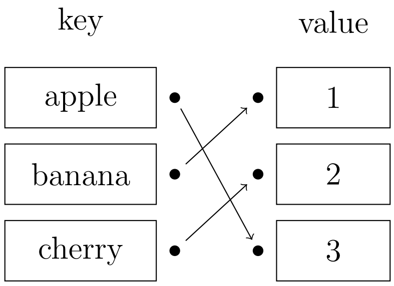


图 10.14

哈希表又称散列表，一种以「key-value」形式存储数据的数据结构。所谓以「key-value」形式存储数据，是指任意的键值 key 都唯一对应到内存中的某个位置。只需要输入查找的键值，就可以快速地找到其对应的 value。可以把哈希表理解为一种高级的数组，这种数组的下标可以是很大的整数，浮点数，字符串甚至结构体。

哈希函数

要让键值对应到内存中的位置，就要为键值计算索引，也就是计算这个数据应该放到哪里。这个根据键值计算索引的函数就叫做哈希函数，也称散列函数。举个例子，如果键值是一个人的身份证号码，哈希函数就可以是号码的后四位，当然也可以是号码的前四位。生活中常用的「手机尾号」也是一种哈希函数。在实际的应用中，键值可能是更复杂的东西，比如浮点数、字符串、结构体等，这时候就要根据具体情况设计合适的哈希函数。哈希函数应当易于计算，并且尽量使计算出来的索引均匀分布。

能为 key 计算索引之后，我们就可以知道每个键值对应的值 value 应该放在哪里了。假设我们用数组 a 存放数据，哈希函数是 f，那键值对 `(key, value)` 就应该放在 `a[f(key)]` 上。不论键值是什么类型，范围有多大，`f(key)` 都是在可接受范围内的整数，可以作为数组的下标。

在 OI 中，最常见的情况应该是键值为整数的情况。当键值的范围比较小的时候，可以直接把键值作为数组的下标，但当键值的范围比较大，比如以 10^9 范围内的整数作为键值的时候，就需要用到哈希表。一般把键值模一个较大的质数作为索引，也就是取 $f(x) = x \bmod M$ 作为哈希函数。

另一种比较常见的情况是 key 为字符串的情况，由于不支持以字符串作为数组下标，并且将字符串转化成数字存储也可以避免多次进行字符串比较。所以在 OI 中，一般不直接把字符串作为键值，而是先算出字符串的哈希值，再把其哈希值作为键值插入到哈希表里。关于字符串的哈希值，我们一般采用进制的思想，将字符串想象成一个 127 进制的数。那么，对于每一个长度为 n 的字符串 s，就有：

$$x = s_0 \cdot 127^0 + s_1 \cdot 127^1 + s_2 \cdot 127^2 + \dots + s_n \cdot 127^n$$

我们可以将得到的 x 对 2^{64}（即 `unsigned long long` 的最大值）取模。这样 `unsigned long long` 的自然溢出就等价于取模操作了。可以使操作更加方便。

这种方法虽然简单，但并不是完美的。可以构造数据使这种方法发生冲突（即两个字符串的 x 对 2^{64} 取模后的结果相同）。

我们可以使用双哈希的方法：选取两个大质数 a, b。当且仅当两个字符串的哈希值对 a 和对 b 取模都相等时，我们才认为这两个字符串相等。这样可以大大降低哈希冲突的概率。

冲突

如果对于任意的键值，哈希函数计算出来的索引都不相同，那只用根据索引把 `(key, value)` 放到对应的位置就行了。但实际上，常常会出现两个不同的键值，他们用哈希函数计算出来的索引是相同的。这时候就需要一些方法来处理冲突。在 OI 中，最常用的方法是拉链法。

拉链法

拉链法也称开散列法（open hashing）。

拉链法是在每个存放数据的地方开一个链表，如果有多个键值索引到同一个地方，只用把他们都放到那个位置的链表里就行了。查询的时候需要把对应位置的链表整个扫一遍，对其中的每个数据比较其键值与查询的键值是否一致。如果索引的范围是 $1 \ldots M$，哈希表的大小为 N，那么一次插入/查询需要进行期望 $O(\frac{N}{M})$ 次比较。

实现

```cpp
const int SIZE = 1000000;
const int M = 999997;

struct HashTable {
  struct Node {
    int next, value, key;
  } data[SIZE];

  int head[M], size;
```

```
  int f(int key) { return (key % M + M) % M; }

  int get(int key) {
    for (int p = head[f(key)]; p; p = data[p].next)
      if (data[p].key == key) return data[p].value;
    return -1;
  }

  int modify(int key, int value) {
    for (int p = head[f(key)]; p; p = data[p].next)
      if (data[p].key == key) return data[p].value = value;
  }

  int add(int key, int value) {
    if (get(key) != -1) return -1;
    data[++size] = (Node){head[f(key)], value, key};
    head[f(key)] = size;
    return value;
  }
};
```

```
M = 999997
SIZE = 1000000
class Node:
    def __init__(self, next = None, value = None, key = None):
        self.next = next
        self.value = value
        self.key = key
data = [Node() for _ in range(SIZE)]
head = [0] * M
size = 0
def f(key):
    return key % M
def get(key):
    p = head[f(key)]
    while p:
        if data[p].key == key:
            return data[p].value
        p = data[p].next
    return -1
def modify(key, value):
    p = head[f(key)]
    while p:
        if data[p].key == key:
            data[p].value = value
            return data[p].value
        p = data[p].next
def add(key, value):
    if get(key) != -1:
        return -1
    size = size + 1
    data[size] = Node(head[f(key)], value, key)
    head[f(key)] = size
    return value
```

这里再提供一个封装过的模板，可以像 map 一样用，并且较短

```
struct hash_map {  // 哈希表模板

  struct data {
    long long u;
    int v, nex;
  };  // 前向星结构

  data e[SZ << 1];  // SZ 是 const int 表示大小
  int h[SZ], cnt;

  int hash(long long u) { return (u % SZ + SZ) % SZ; }

  // 这里使用 (u % SZ + SZ) % SZ 而非 u % SZ 的原因是
  // C++ 中的 % 运算无法将负数转为正数

  int& operator[](long long u) {
    int hu = hash(u);  // 获取头指针
    for (int i = h[hu]; i; i = e[i].nex)
      if (e[i].u == u) return e[i].v;
    return e[++cnt] = (data){u, -1, h[hu]}, h[hu] = cnt, e[cnt].v;
  }

  hash_map() {
    cnt = 0;
    memset(h, 0, sizeof(h));
  }
};
```

在这里，hash 函数是针对键值的类型设计的，并且返回一个链表头指针用于查询。在这个模板中我们写了一个键值对类型为 `(long long, int)` 的 hash 表，并且在查询不存在的键值时返回 -1。函数 `hash_map()` 用于在定义时初始化。

闭散列法

闭散列方法把所有记录直接存储在散列表中，如果发生冲突则根据某种方式继续进行探查。

比如线性探查法：如果在 `d` 处发生冲突，就依次检查 `d + 1`，`d + 2`……

实现

```
const int N = 360007;  // N 是最大可以存储的元素数量

class Hash {
 private:
  int keys[N];
  int values[N];

 public:
  Hash() { memset(values, 0, sizeof(values)); }

  int& operator[](int n) {
    // 返回一个指向对应 Hash[Key] 的引用
    // 修改成不为 0 的值 0 时候视为空
```

```
    int idx = (n % N + N) % N, cnt = 1;
    while (keys[idx] != n && values[idx] != 0) {
      idx = (idx + cnt * cnt) % N;
      cnt += 1;
    }
    keys[idx] = n;
    return values[idx];
  }
};
```

例题

「JLOI2011」 不重复数字[1]

参考资料与注释

[1] 「JLOI2011」不重复数字

10.6 并查集

10.6.1 并查集

Authors: HeRaNO, JuicyMio, Xeonacid, sailordiary, ouuan

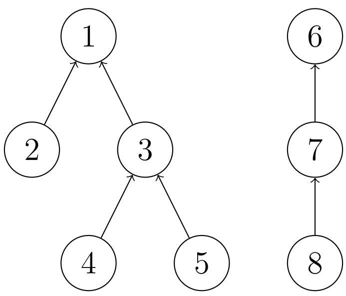


图 10.15

引入

并查集是一种用于管理元素所属集合的数据结构，实现为一个森林，其中每棵树表示一个集合，树中的节点表示对应集合中的元素。

顾名思义，并查集支持两种操作：

- 合并（Union）：合并两个元素所属集合（合并对应的树）
- 查询（Find）：查询某个元素所属集合（查询对应的树的根节点），这可以用于判断两个元素是否属于同一集合

并查集在经过修改后可以支持单个元素的删除、移动；使用动态开点线段树还可以实现可持久化并查集。

warning

并查集无法以较低复杂度实现集合的分离。

初始化

初始时，每个元素都位于一个单独的集合，表示为一棵只有根节点的树。方便起见，我们将根节点的父亲设为自己。

" 实现"

```cpp
struct dsu {
  vector<size_t> pa;

  explicit dsu(size_t size) : pa(size) { iota(pa.begin(), pa.end(), 0); }
};
```

```python
class Dsu:
    def __init__(self, size):
        self.pa = list(range(size))
```

查询

我们需要沿着树向上移动，直至找到根节点。

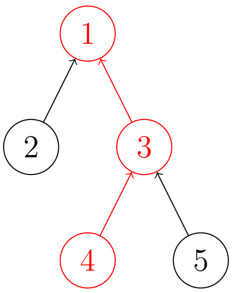


图 10.16

" 实现"

```cpp
size_t dsu::find(size_t x) { return pa[x] == x ? x : find(pa[x]); }
```

```python
def find(self, x):
    return x if self.pa[x] == x else self.find(self.pa[x])
```

路径压缩

查询过程中经过的每个元素都属于该集合，我们可以将其直接连到根节点以加快后续查询。

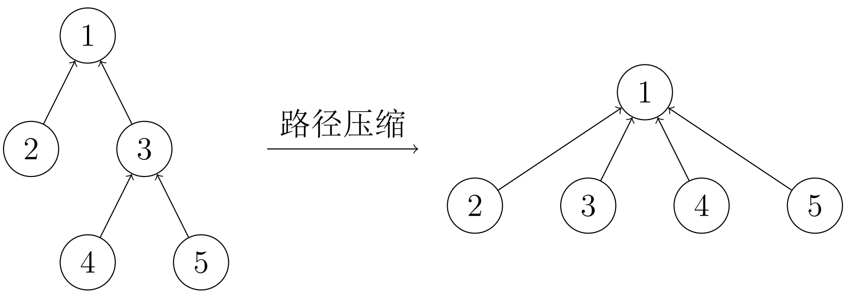


图 10.17

" 实现"

```
size_t dsu::find(size_t x) { return pa[x] == x ? x : pa[x] = find(pa[x]); }
```

```
def find(self, x):
    if self.pa[x] != x:
        self.pa[x] = self.find(self.pa[x])
    return self.pa[x]
```

合并

要合并两棵树，我们只需要将一棵树的根节点连到另一棵树的根节点。

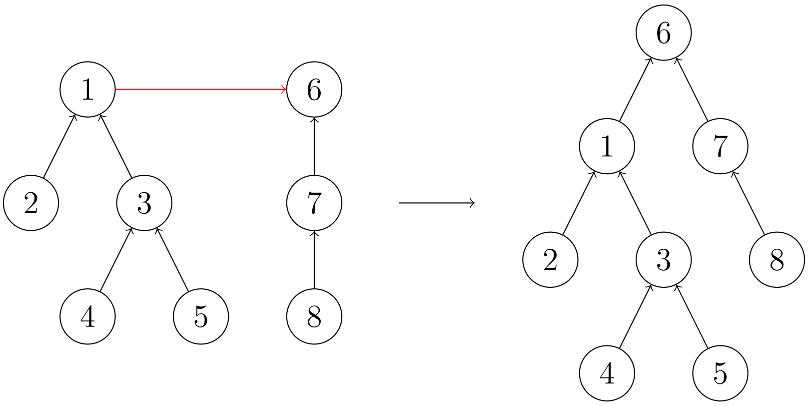


图 10.18

" 实现"

```
void dsu::unite(size_t x, size_t y) { pa[find(x)] = find(y); }
```

```
def union(self, x, y):
    self.pa[self.find(x)] = self.find(y)
```

启发式合并

合并时，选择哪棵树的根节点作为新树的根节点会影响未来操作的复杂度。我们可以将节点较少或深度较小的树连到另一棵，以免发生退化。

" 具体复杂度讨论"

由于需要我们支持的只有集合的合并、查询操作，当我们需要将两个集合合二为一时，无论将哪一个集合连接到另一个集合的下面，都能得到正确的结果。但不同的连接方法存在时间复杂度的差异。具体来说，如果我们将一棵点数与深度都较小的集合树连接到一棵更大的集合树下，显然相比于另一种连接方案，接下来执行查找操作的用时更小（也会带来更优的最坏时间复杂度）。

当然，我们不总能遇到恰好如上所述的集合——点数与深度都更小。鉴于点数与深度这两个特征都很容易维护，我们常常从中择一，作为估价函数。而无论选择哪一个，时间复杂度都为 $O(m\alpha(m,n))$，具体的证明可参见 References 中引用的论文。

在算法竞赛的实际代码中，即便不使用启发式合并，代码也往往能够在规定时间内完成任务。在 Tarjan 的论文[1] 中，证明了不使用启发式合并、只使用路径压缩的最坏时间复杂度是 $O(m\log n)$。在姚期智的论文[2] 中，证明了不使用启发式合并、只使用路径压缩，在平均情况下，时间复杂度依然是 $O(m\alpha(m,n))$。

如果只使用启发式合并，而不使用路径压缩，时间复杂度为 $O(m \log n)$。由于路径压缩单次合并可能造成大量修改，有时路径压缩并不适合使用。例如，在可持久化并查集、线段树分治 + 并查集中，一般使用只启发式合并的并查集。

按节点数合并的参考实现：

" 实现"

```
struct dsu {
  vector<size_t> pa, size;

  explicit dsu(size_t size_) : pa(size_), size(size_, 1) {
    iota(pa.begin(), pa.end(), 0);
  }

  void unite(size_t x, size_t y) {
    x = find(x), y = find(y);
    if (x == y) return;
    if (size[x] < size[y]) swap(x, y);
    pa[y] = x;
    size[x] += size[y];
  }
};
```

```
class Dsu:
    def __init__(self, size):
        self.pa = list(range(size))
        self.size = [1] * size

    def union(self, x, y):
        x, y = self.find(x), self.find(y)
        if x == y:
            return
        if self.size[x] < self.size[y]:
            x, y = y, x
        self.pa[y] = x
        self.size[x] += self.size[y]
```

删除

要删除一个叶子节点，我们可以将其父亲设为自己。为了保证要删除的元素都是叶子，我们可以预先为每个节点制作副本，并将其副本作为父亲。

" 实现"

```
struct dsu {
  vector<size_t> pa, size;

  explicit dsu(size_t size_) : pa(size_ * 2), size(size_ * 2, 1) {
    iota(pa.begin(), pa.begin() + size_, size_);
    iota(pa.begin() + size_, pa.end(), size_);
  }

  void erase(size_t x) {
```

```
    --size[find(x)];
    pa[x] = x;
  }
};

class Dsu:
    def __init__(self, size):
        self.pa = list(range(size, size * 2)) * 2
        self.size = [1] * size * 2

    def erase(self, x):
        self.size[self.find(x)] -= 1
        self.pa[x] = x
```

移动

与删除类似，通过以副本作为父亲，保证要移动的元素都是叶子。

" 实现"

```
void dsu::move(size_t x, size_t y) {
  auto fx = find(x), fy = find(y);
  if (fx == fy) return;
  pa[x] = fy;
  --size[fx], ++size[fy];
}

def move(self, x, y):
    fx, fy = self.find(x), self.find(y)
    if fx == fy:
        return
    self.pa[x] = fy
    self.size[fx] -= 1
    self.size[fy] += 1
```

复杂度

时间复杂度

同时使用路径压缩和启发式合并之后，并查集的每个操作平均时间仅为 $O(\alpha(n))$，其中 α 为阿克曼函数的反函数，其增长极其缓慢，也就是说其单次操作的平均运行时间可以认为是一个很小的常数。

Ackermann 函数[3] $A(m,n)$ 的定义是这样的：

$$A(m,n)=\begin{cases}n+1 & \text{if } m=0\\ A(m-1,1) & \text{if } m>0 \text{ and } n=0\\ A(m-1,A(m,n-1)) & \text{otherwise}\end{cases}$$

而反 Ackermann 函数 $\alpha(n)$ 的定义是阿克曼函数的反函数，即为最大的整数 m 使得 $A(m,m)\leqslant n$。

时间复杂度的证明 在这个页面中。

空间复杂度

显然为 $O(n)$。

带权并查集

我们还可以在并查集的边上定义某种权值、以及这种权值在路径压缩时产生的运算，从而解决更多的问题。比如对于经典的「NOI2001」食物链，我们可以在边权上维护模 3 意义下的加法群。

例题

"UVa11987 Almost Union-Find[4]"

实现类似并查集的数据结构，支持以下操作：

1. 合并两个元素所属集合
2. 移动单个元素
3. 查询某个元素所属集合的大小及元素和

" 参考代码"

```cpp
#include <bits/stdc++.h>

using namespace std;

struct dsu {
  vector<size_t> pa, size, sum;

  explicit dsu(size_t size_)
      : pa(size_ * 2), size(size_ * 2, 1), sum(size_ * 2) {
    // size 与 sum 的前半段其实没有使用，只是为了让下标计算更简单
    iota(pa.begin(), pa.begin() + size_, size_);
    iota(pa.begin() + size_, pa.end(), size_);
    iota(sum.begin() + size_, sum.end(), 0);
  }

  void unite(size_t x, size_t y) {
    x = find(x), y = find(y);
    if (x == y) return;
    if (size[x] < size[y]) swap(x, y);
    pa[y] = x;
    size[x] += size[y];
    sum[x] += sum[y];
  }

  void move(size_t x, size_t y) {
    auto fx = find(x), fy = find(y);
    if (fx == fy) return;
    pa[x] = fy;
    --size[fx], ++size[fy];
    sum[fx] -= x, sum[fy] += x;
  }

  size_t find(size_t x) { return pa[x] == x ? x : pa[x] = find(pa[x]); }
};

int main() {
```

```
  size_t n, m, op, x, y;
  while (cin >> n >> m) {
    dsu dsu(n + 1);  // 元素范围是 1..n
    while (m--) {
      cin >> op;
      switch (op) {
        case 1:
          cin >> x >> y;
          dsu.unite(x, y);
          break;
        case 2:
          cin >> x >> y;
          dsu.move(x, y);
          break;
        case 3:
          cin >> x;
          x = dsu.find(x);
          cout << dsu.size[x] << ' ' << dsu.sum[x] << '\n';
          break;
        default:
          assert(false);  // not reachable
      }
    }
  }
}
```

```
class Dsu:
    def __init__(self, size):
        # size 与 sum 的前半段其实没有使用，只是为了让下标计算更简单
        self.pa = list(range(size, size * 2)) * 2
        self.size = [1] * size * 2
        self.sum = list(range(size)) * 2

    def union(self, x, y):
        x, y = self.find(x), self.find(y)
        if x == y:
            return
        if self.size[x] < self.size[y]:
            x, y = y, x
        self.pa[y] = x
        self.size[x] += self.size[y]
        self.sum[x] += self.sum[y]

    def move(self, x, y):
        fx, fy = self.find(x), self.find(y)
        if fx == fy:
            return
        self.pa[x] = fy
        self.size[fx] -= 1
        self.size[fy] += 1
        self.sum[fx] -= x
        self.sum[fy] += x
```

```python
    def find(self, x):
        if self.pa[x] != x:
            self.pa[x] = self.find(self.pa[x])
        return self.pa[x]


if __name__ == "__main__":
    while True:
        try:
            n, m = map(int, input().split())
            dsu = Dsu(n + 1)  # 元素范围是 1..n
            for _ in range(m):
                op_x_y = list(map(int, input().split()))
                op = op_x_y[0]
                if op == 1:
                    dsu.union(op_x_y[1], op_x_y[2])
                elif op == 2:
                    dsu.move(op_x_y[1], op_x_y[2])
                elif op == 3:
                    x = dsu.find(op_x_y[1])
                    print(dsu.size[x], dsu.sum[x])
        except EOFError:
            break
```

习题

「NOI2015」 程序自动分析[5]

「JSOI2008」 星球大战[6]

「NOI2001」 食物链[7]

「NOI2002」 银河英雄传说[8]

其他应用

最小生成树算法 中的 Kruskal 和 最近公共祖先 中的 Tarjan 算法是基于并查集的算法。

相关专题见 并查集应用。

参考资料与拓展阅读

1. 知乎回答： 是否在并查集中真的有二分路径压缩优化？[9]
2. Gabow, H. N., & Tarjan, R. E. (1985). A Linear-Time Algorithm for a Special Case of Disjoint Set Union. JOURNAL OF COMPUTER AND SYSTEM SCIENCES, 30, 209-221.PDF[10]

[1] Tarjan, R. E., & Van Leeuwen, J. (1984). Worst-case analysis of set union algorithms. Journal of the ACM (JACM), 31(2), 245-281.ResearchGate PDF

[2] Yao, A. C. (1985). On the expected performance of path compression algorithms.SIAM Journal on Computing, 14(1), 129-133.

[3] Ackermann 函数

[4] UVa11987 Almost Union-Find

[5] 「NOI2015」程序自动分析

[6] 「JSOI2008」星球大战

[7] 「NOI2001」食物链

[8] 「NOI2002」银河英雄传说

[9] 知乎回答：是否在并查集中真的有二分路径压缩优化？

[10] PDF

10.6.2 并查集复杂度

Authors: orzAtalod

本部分内容转载并修改自 时间复杂度 - 势能分析浅谈[1]，已取得原作者授权同意。

定义

阿克曼函数

这里，先给出 $\alpha(n)$ 的定义。为了给出这个定义，先给出 $A_k(j)$ 的定义。

定义 $A_k(j)$ 为：

$$A_k(j) = \begin{cases} j+1 & k=0 \\ A_{k-1}^{(j+1)}(j) & k \geq 1 \end{cases}$$

即阿克曼函数。

这里，$f^i(x)$ 表示将 f 连续应用在 x 上 i 次，即 $f^0(x)=x$，$f^i(x)=f(f^{i-1}(x))$。

再定义 $\alpha(n)$ 为使得 $A_{\alpha(n)}(1) \geq n$ 的最小整数值。注意，我们之前将它描述为 $A_{\alpha(n)}(\alpha(n)) \geq n$，反正他们的增长速度都很慢，值都不超过 4。

基础定义

每个节点都有一个 rank。这里的 rank 不是节点个数，而是深度。节点的初始 rank 为 0，在合并的时候，如果两个节点的 rank 不同，则将 rank 小的节点合并到 rank 大的节点上，并且不更新大节点的 rank 值。否则，随机将某个节点合并到另外一个节点上，将根节点的 rank 值 +1。这里根节点的 rank 给出了该树的高度。记 x 的 rank 为 $rnk(x)$，类似的，记 x 的父节点为 $fa(x)$。我们总有 $rnk(x)+1 \leq rnk(fa(x))$。

为了定义势函数，需要预先定义一个辅助函数 $level(x)$。其中，$level(x)=\max(k : rnk(fa(x)) \geq A_k(rnk(x)))$。当 $rnk(x) \geq 1$ 的时候，再定义一个辅助函数 $iter(x)=\max(i : rnk(fa(x)) \geq A_{level(x)}^i(rnk(x))$。这些函数定义的 x 都满足 $rnk(x)>0$ 且 x 不是某个树的根。

上面那些定义可能让你有点头晕。再理一下，对于一个 x 和 $fa(x)$，如果 $rnk(x)>0$，总是可以找到一对 i,k 令 $rnk(fa(x)) \geq A_k^i(rnk(x))$，而 $level(x)=\max(k)$，在这个前提下，$iter(x)=\max(i)$。$level$ 描述了 A 的最大迭代级数，而 $iter$ 描述了在最大迭代级数时的最大迭代次数。

对于这两个函数，$level(x)$ 总是随着操作的进行而增加或不变，如果 $level(x)$ 不增加，$iter(x)$ 也只会增加或不变。并且，它们总是满足以下两个不等式：

$$0 \leq level(x) < \alpha(n)$$

$$1 \le iter(x) \le rnk(x)$$

考虑 $level(x)$、$iter(x)$ 和 A_k^j 的定义，这些很容易被证明出来，就留给读者用于熟悉定义了。

定义势能函数 $\Phi(S)=\sum\limits_{x\in S}\Phi(x)$，其中 S 表示一整个并查集，而 x 为并查集中的一个节点。定义 $\Phi(x)$ 为：

$$\Phi(x)=\begin{cases}\alpha(n)\times rnk(x) & rnk(x)=0 \text{ 或 } x \text{ 为某棵树的根节点}\\ (\alpha(n)-level(x))\times rnk(x)-iter(x) & \text{otherwise}\end{cases}$$

然后就是通过操作引起的势能变化来证明摊还时间复杂度为 $\Theta(\alpha(n))$ 啦。注意，这里我们讨论的 $union(x,y)$ 操作保证了 x 和 y 都是某个树的根，因此不需要额外执行 $find(x)$ 和 $find(y)$。

可以发现，势能总是个非负数。另，在开始的时候，并查集的势能为 0。

证明

union(x,y) 操作

其花费的时间为 $\Theta(1)$，因此我们考虑其引起的势能的变化。

这里，我们假设 $rnk(x)\le rnk(y)$，即 x 被接到 y 上。这样，势能增加的节点仅有 x（从树根变成非树根），y（秩可能增加）和操作前 y 的子节点（父节点的秩可能增加）。我们先证明操作前 y 的子节点 c 的势能不可能增加，并且如果减少了，至少减少 1。

设操作前 c 的势能为 $\Phi(c)$，操作后为 $\Phi(c')$，这里 c 可以是任意一个 $rnk(c)>0$ 的非根节点，操作可以是任意操作，包括下面的 find 操作。我们分三种情况讨论。

1. $iter(c)$ 和 $level(c)$ 并未增加。显然有 $\Phi(c)=\Phi(c')$。
2. $iter(c)$ 增加了，$level(c)$ 并未增加。这里 $iter(c)$ 至少增加一，即 $\Phi(c')\le\Phi(c)-1$，势能函数减少了，并且至少减少 1。
3. $level(c)$ 增加了，$iter(c)$ 可能减少。但是由于 $0<iter(c)\le rnk(c)$，$iter(c)$ 最多减少 $rnk(c)-1$，而 $level(c)$ 至少增加 1。由定义 $\Phi(c)=(\alpha(n)-level(c))\times rnk(c)-iter(c)$，可得 $\Phi(c')\le\Phi(c)-1$。
4. 其他情况。由于 $rnk(c)$ 不变，$rnk(fa(c))$ 不减，所以不存在。

所以，势能增加的节点仅可能是 x 或 y。而 x 从树根变成了非树根，如果 $rnk(x)=0$，则一直有 $\Phi(x)=\Phi(x')=0$。否则，一定有 $\alpha(x)\times rnk(x)\ge(\alpha(n)-level(x))\times rnk(x)-iter(x)$。即，$\Phi(x')\le\Phi(x)$。

因此，唯一势能可能增加的点就是 y。而 y 的势能最多增加 $\alpha(n)$。因此，可得 $union$ 操作均摊后的时间复杂度为 $\Theta(\alpha(n))$。

find(a) 操作

如果查找路径包含 $\Theta(s)$ 个节点，显然其查找的时间复杂度是 $\Theta(s)$。如果由于查找操作，没有节点的势能增加，且至少有 $s-\alpha(n)$ 个节点的势能至少减少 1，就可以证明 $find(a)$ 操作的时间复杂度为 $\Theta(\alpha(n))$。为了避免混淆，这里用 a 作为参数，而出现的 x 都是泛指某一个并查集内的结点。

首先证明没有节点的势能增加。很显然，我们在上面证明过所有非根节点的势能不增，而根节点的 rnk 没有改变，所以没有节点的势能增加。

接下来证明至少有 $s-\alpha(n)$ 个节点的势能至少减少 1。我们上面证明过了，如果 $level(x)$ 或者 $iter(x)$ 有改变的话，它们的势能至少减少 1。所以，只需要证明至少有 $s-\alpha(n)$ 个节点的 $level(x)$ 或者 $iter(x)$ 有改变即可。

回忆一下非根节点势能的定义，$\Phi(x)=(\alpha(n)-level(x))\times rnk(x)-iter(x)$，而 $level(x)$ 和 $iter(x)$ 是使 $rnk(fa(x))\ge A_{level(x)}^{iter(x)}(rnk(x))$ 的最大数。

所以，如果 $root_x$ 代表 x 所处的树的根节点，只需要证明 $rnk(root_x)\ge A_{level(x)}^{iter(x)+1}(rnk(x))$ 就好了。根据 A_k^i 的定义，$A_{level(x)}^{iter(x)+1}(rnk(x))=A_{level(x)}(A_{level(x)}^{iter(x)}(rnk(x)))$。

注意，我们可能会用 $k(x)$ 代表 $level(x)$，$i(x)$ 代表 $iter(x)$ 以避免式子过于冗长。这里，就是 $rnk(root_x)\ge A_{k(x)}(A_{k(x)}^{i(x)}(x))$。

当你看到这的时候，可能会有一种「这啥玩意」的感觉。这意味着你可能需要多看几遍，或者跳过一些内容以后再看。

这里，我们需要一个外接的 $A_{k(x)}$，意味着我们可能需要再找一个点 y。令 y 是搜索路径上在 x 之后的满足 $k(y)=k(x)$ 的点，这里「搜索路径之后」相当于「是 x 的祖先」。显然，不是每一个 x 都有这样一个 y。很容易证明，没有这样的 y 的 x 不超过 $\alpha(n)-2$ 个。因为只有每个 k 的最后一个 x 和 a 以及 $root_a$ 没有这样的 y。

我们再强调一遍 $fa(x)$ 指的是路径压缩**之前** x 的父节点，路径压缩**之后** x 的父节点一律用 $root_x$ 表示。对于每个存在 y 的 x，总是有 $rnk(y) \geq rnk(fa(x))$。同时，我们有 $rnk(fa(x)) \geq A_{k(x)}^{i(x)}(rnk(x))$。由于 $k(x)=k(y)$，我们用 k 来统称，即，$rnk(fa(x)) \geq A_k^{i(x)}(rnk(x))$。我们需要造一个 A_k 出来，所以我们可以不关注 $iter(y)$ 的值，直接使用弱化版的 $rnk(fa(y)) \geq A_k(rnk(y))$。

如果我们将不等式组合起来，神奇的事情就发生了。我们发现，$rnk(fa(y)) \geq A_k^{i(x)+1}(rnk(x))$。也就是说，为了从 $rnk(x)$ 迭代到 $rnk(fa(y))$，至少可以迭代 A_k 不少于 $i(x)+1$ 次而不超过 $rnk(fa(y))$。

显然，有 $rnk(root_y) \geq rnk(fa(y))$，且 $rnk(x)$ 在路径压缩时不变。因此，我们可以得到 $rnk(root_x) \geq A_k^{i(x)+1}(rnk(x))$，也就是说 $iter(x)$ 的值至少增加 1，如果 $rnk(x)$ 没有增加，一定是 $level(x)$ 增加了。

所以，$\Phi(x)$ 至少减少了 1。由于这样的 x 节点至少有 $s-\alpha(n)-2$ 个，所以最后 $\Phi(S)$ 至少减少了 $s-\alpha(n)-2$，均摊后的时间复杂度即为 $\Theta(\alpha(n)+2)=\Theta(\alpha(n))$。

为何并查集会被卡

这个问题也就是问，如果我们不按秩合并，会有哪些性质被破坏，导致并查集的时间复杂度不能保证为 $\Theta(m\alpha(n))$。

如果我们在合并的时候，rnk 较大的合并到了 rnk 较小的节点上面，我们就将那个 rnk 较小的节点的 rnk 值设为另一个节点的 rnk 值加一。这样，我们就能保证 $rnk(fa(x)) \geq rnk(x)+1$，从而不会出现类似于满地 compile error 一样的性质不符合。

显然，如果这样子的话，我们破坏的就是 $union(x,y)$ 函数「y 的势能最多增加 $\alpha(n)$」这一句。

存在一个能使路径压缩并查集时间复杂度降至 $\Omega(m\log_{1+\frac{m}{n}} n)$ 的结构，定义如下：

二项树（实际上和一般的二项树不太一样），其中 j 是常数，T_k 为一个 T_{k-1} 加上一个 T_{k-j} 作为根节点的儿子。

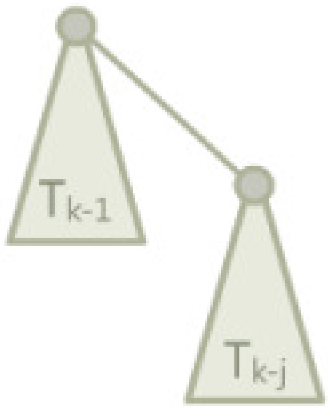


图 10.19 我们的二项树

边界条件，T_1 到 T_j 都是一个单独的点。

令 $rnk(T_k)=r_k$，这里我们有 $r_k=(k-1)/j$（证明略）。每轮操作，我们将它接到一个单节点上，然后查询底部的 j 个节点。也就是说，我们接到单节点上的时候，单节点的势能提高了 $(k-1)/j+1$。在 $j=\lfloor\frac{m}{n}\rfloor$，$i=\lfloor\log_{j+1}\frac{n}{2}\rfloor$，$k=ij$ 的时候，势能增加量为：

$$\alpha(n)\times((ij-1)/j+1)=\alpha(n)\times((\lfloor\log_{\lfloor\frac{m}{n}\rfloor+1}\frac{n}{2}\rfloor\times\lfloor\frac{m}{n}\rfloor-1)/\lfloor\frac{m}{n}\rfloor+1)$$

变换一下，去掉所有的取整符号，就可以得出，势能增加量 $\geq \alpha(n)\times(\log_{1+\frac{m}{n}} n-\frac{n}{m})$，m 次操作就是 $\Omega(m\log_{1+\frac{m}{n}} n-n)=\Omega(m\log_{1+\frac{m}{n}} n)$。

关于启发式合并

由于按秩合并比启发式合并难写，所以很多 dalao 会选择使用启发式合并来写并查集。具体来说，则是对每个根都维护一个 $size(x)$，每次将 $size$ 小的合并到大的上面。

所以，启发式合并会不会被卡？

首先，可以从秩参与证明的性质来说明。如果 $size$ 可以代替 rnk 的地位，则可以使用启发式合并。快速总结一下，秩参与证明的性质有以下三条：

1. 每次合并，最多有一个节点的秩上升，而且最多上升 1。
2. 总有 $rnk(fa(x)) \geq rnk(x) + 1$。
3. 节点的秩不减。

关于第二条和第三条，siz 显然满足，然而第一条不满足，如果将 x 合并到 y 上面，则 $siz(y)$ 会增大 $siz(x)$ 那么多。

所以，可以考虑使用 $\log_2 siz(x)$ 代替 $rnk(x)$。

关于第一条性质，由于节点的 siz 最多翻倍，所以 $\log_2 siz(x)$ 最多上升 1。关于第二三条性质，结论较为显然，这里略去证明。

所以说，如果不想写按秩合并，就写启发式合并好了，时间复杂度仍旧是 $\Theta(m\alpha(n))$。

参考资料与注释

[1] 时间复杂度 - 势能分析浅谈

10.7　堆

10.7.1　堆简介

Authors: ouuan, HeRaNO

堆是一棵树，其每个节点都有一个键值，且每个节点的键值都大于等于/小于等于其父亲的键值。

每个节点的键值都大于等于其父亲键值的堆叫做小根堆，否则叫做大根堆。STL 中的 `priority_queue` 其实就是一个大根堆。

（小根）堆主要支持的操作有：插入一个数、查询最小值、删除最小值、合并两个堆、减小一个元素的值。

一些功能强大的堆（可并堆）还能（高效地）支持 merge 等操作。

一些功能更强大的堆还支持可持久化，也就是对任意历史版本进行查询或者操作，产生新的版本。

堆的分类

操作 \ 数据结构[4]	配对堆	二叉堆	左偏树	二项堆	斐波那契堆
插入（insert）	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$[1]	$O(1)$
查询最小值（find-min）	$O(1)$	$O(1)$	$O(1)$	$O(1)$[2][3-1]	$O(1)$
删除最小值（delete-min）	$O(\log n)$[3-2]	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$[3-3]
合并 (merge)	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$
减小一个元素的值 (decrease-key)	$o(\log n)$（下界 $\Omega(\log\log n)$，上界 $O(2^{2\sqrt{\log\log n}})$）[3-4]	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$[3-5]

操作 \ 数据结构[4]	配对堆	二叉堆	左偏树	二项堆	斐波那契堆
是否支持可持久化	×	✓	✓	✓	×

习惯上，不加限定提到「堆」时往往都指二叉堆。

[1] 单次插入的复杂度为 $O(\log n)$，但有 k 次连续插入时，可创建一个只包含要插入元素的二项堆，再将此堆与原先的二项堆进行合并，均摊复杂度为 $O(1)$

[2] 可以保存一个指向最小元素的指针，在执行其他操作时修改该指针，即可在 $O(1)$ 的复杂度下进行查询了

[3] 复杂度为均摊复杂度 [3-1] [3-2] [3-3] [3-4] [3-5]

[4] 表格来自于 Wikipedia

10.7.2 二叉堆

Authors: HeRaNO, Xeonacid, AzurIce

结构

从二叉堆的结构说起，它是一棵二叉树，并且是完全二叉树，每个结点中存有一个元素（或者说，有个权值）。

堆性质：父亲的权值不小于儿子的权值（大根堆）。同样的，我们可以定义小根堆。本文以大根堆为例。

由堆性质，树根存的是最大值（getmax 操作就解决了）。

过程

插入操作

插入操作是指向二叉堆中插入一个元素，要保证插入后也是一棵完全二叉树。

最简单的方法就是，最下一层最右边的叶子之后插入。

如果最下一层已满，就新增一层。

插入之后可能会不满足堆性质？

向上调整：如果这个结点的权值大于它父亲的权值，就交换，重复此过程直到不满足或者到根。

可以证明，插入之后向上调整后，没有其他结点会不满足堆性质。

向上调整的时间复杂度是 $O(\log n)$ 的。

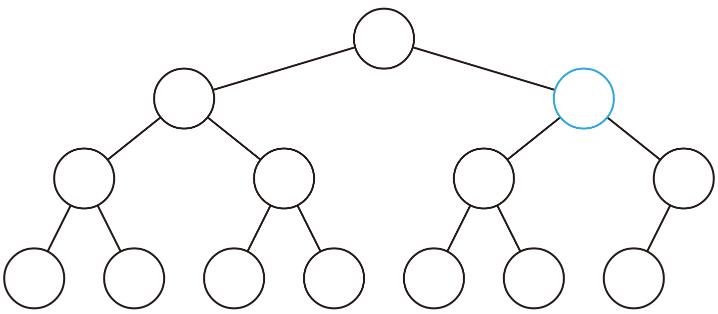


图 10.20 二叉堆的插入操作

删除操作

删除操作指删除堆中最大的元素，即删除根结点。

但是如果直接删除，则变成了两个堆，难以处理。

所以不妨考虑插入操作的逆过程，设法将根结点移到最后一个结点，然后直接删掉。

然而实际上不好做，我们通常采用的方法是，把根结点和最后一个结点直接交换。

于是直接删掉（在最后一个结点处的）根结点，但是新的根结点可能不满足堆性质⋯⋯

向下调整：在该结点的儿子中，找一个最大的，与该结点交换，重复此过程直到底层。

可以证明，删除并向下调整后，没有其他结点不满足堆性质。

时间复杂度 $O(\log n)$。

增加某个点的权值

很显然，直接修改后，向上调整一次即可，时间复杂度为 $O(\log n)$。

实现

我们发现，上面介绍的几种操作主要依赖于两个核心：向上调整和向下调整。

考虑使用一个序列 h 来表示堆。h_i 的两个儿子分别是 h_{2i} 和 h_{2i+1}，1 是根结点：

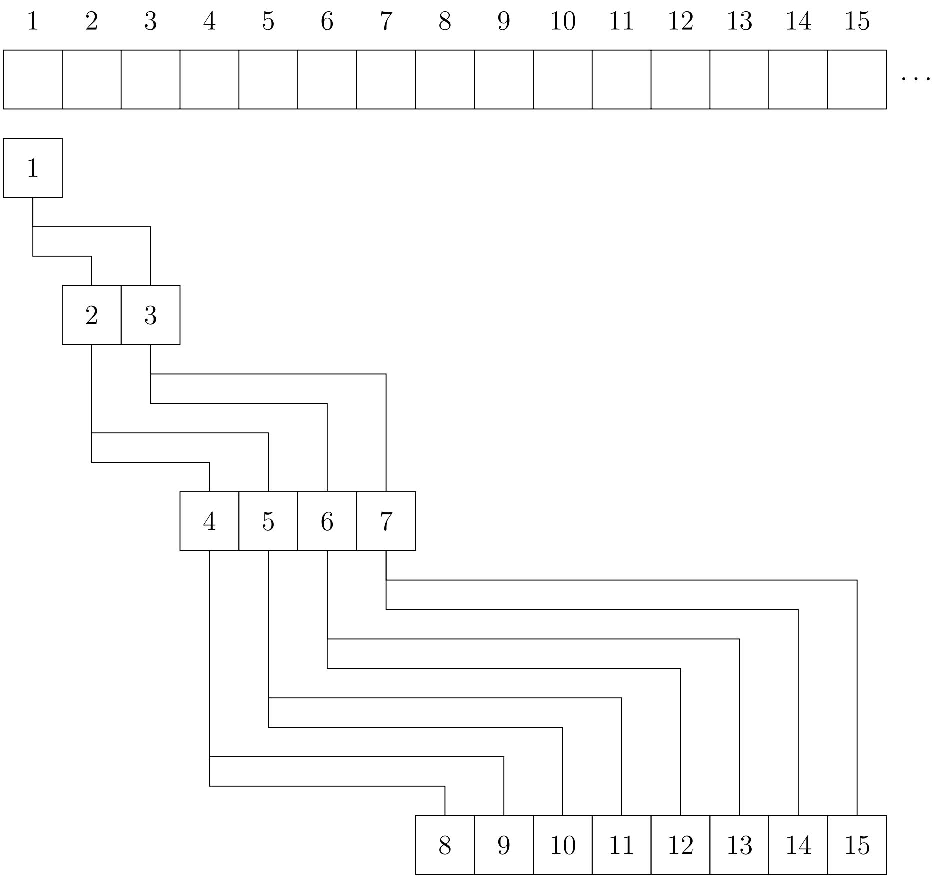


图 10.21 h 的堆结构

参考代码：

```
void up(int x) {
  while (x > 1 && h[x] > h[x / 2]) {
    swap(h[x], h[x / 2]);
```

```
      x /= 2;
    }
}

void down(int x) {
  while (x * 2 <= n) {
    t = x * 2;
    if (t + 1 <= n && h[t + 1] > h[t]) t++;
    if (h[t] <= h[x]) break;
    std::swap(h[x], h[t]);
    x = t;
  }
}
```

建堆

考虑这么一个问题，从一个空的堆开始，插入 n 个元素，不在乎顺序。

直接一个一个插入需要 $O(n\log n)$ 的时间，有没有更好的方法?

方法一：使用 decreasekey（即，向上调整）

从根开始，按 BFS 序进行。

```
void build_heap_1() {
  for (i = 1; i <= n; i++) up(i);
}
```

为啥这么做：对于第 k 层的结点，向上调整的复杂度为 $O(k)$ 而不是 $O(\log n)$。

总复杂度：$\log 1+\log 2+\cdots+\log n=\Theta(n\log n)$。

（在「基于比较的排序」中证明过）

方法二：使用向下调整

这时换一种思路，从叶子开始，逐个向下调整

```
void build_heap_2() {
  for (i = n; i >= 1; i--) down(i);
}
```

换一种理解方法，每次「合并」两个已经调整好的堆，这说明了正确性。

注意到向下调整的复杂度，为 $O(\log n-k)$，另外注意到叶节点无需调整，因此可从序列约 $n/2$ 的位置开始调整，可减少部分常数但不影响复杂度。

" 证明"

$$\begin{aligned}\text{总复杂度}&=n\log n-\log 1-\log 2-\cdots-\log n\\&\le n\log n-0\times 2^0-1\times 2^1-\cdots-(\log n-1)\times\frac{n}{2}\\&=n\log n-(n-1)-(n-2)-(n-4)-\cdots-(n-\frac{n}{2})\\&=n\log n-n\log n+1+2+4+\cdots+\frac{n}{2}\\&=n-1\\&=O(n)\end{aligned}$$

之所以能 $O(n)$ 建堆，是因为堆性质很弱，二叉堆并不是唯一的。

要是像排序那样的强条件就难说了。

应用

对顶堆

"SP16254 RMID2 - Running Median Again[1]"

维护一个序列，支持两种操作：

1. 向序列中插入一个元素
2. 输出并删除当前序列的中位数（若序列长度为偶数，则输出较小的中位数）

这个问题可以被进一步抽象成：动态维护一个序列上第 k 大的数，k 值可能会发生变化。

对于此类问题，我们可以使用**对顶堆**这一技巧予以解决（可以避免写权值线段树或 BST 带来的繁琐）。

对顶堆由一个大根堆与一个小根堆组成，小根堆维护大值即前 k 大的值（包含第 k 个），大根堆维护小值即比第 k 大数小的其他数。

这两个堆构成的数据结构支持以下操作：

- 维护：当小根堆的大小小于 k 时，不断将大根堆堆顶元素取出并插入小根堆，直到小根堆的大小等于 k；当小根堆的大小大于 k 时，不断将小根堆堆顶元素取出并插入大根堆，直到小根堆的大小等于 k；
- 插入元素：若插入的元素大于等于小根堆堆顶元素，则将其插入小根堆，否则将其插入大根堆，然后维护对顶堆；
- 查询第 k 大元素：小根堆堆顶元素即为所求；
- 删除第 k 大元素：删除小根堆堆顶元素，然后维护对顶堆；
- k 值 $+1/-1$：根据新的 k 值直接维护对顶堆。

显然，查询第 k 大元素的时间复杂度是 $O(1)$ 的。由于插入、删除或调整 k 值后，小根堆的大小与期望的 k 值最多相差 1，故每次维护最多只需对大根堆与小根堆中的元素进行一次调整，因此，这些操作的时间复杂度都是 $O(\log n)$ 的。

" 参考代码"

```cpp
#include <cstdio>
#include <iostream>
#include <queue>
using namespace std;

int main() {
  int t, x;
  scanf("%d", &t);
  while (t--) {
    // 大根堆，维护前一半元素（存小值）
    priority_queue<int, vector<int>, less<int> > a;
    // 小根堆，维护后一半元素（存大值）
    priority_queue<int, vector<int>, greater<int> > b;
    while (scanf("%d", &x) && x) {
      // 若为查询并删除操作，输出并删除大根堆堆顶元素
      // 因为这题要求输出中位数中较小者（偶数个数字会存在两个中位数候选）
      // 这个和上面的第 k 大讲解有稍许出入，但如果理解了上面的，这个稍微变通下便可理
清
      if (x == -1) {
        printf("%d\n", a.top());
        a.pop();
      }
```

```
      // 若为插入操作，根据大根堆堆顶的元素值，选择合适的堆进行插入
      else {
        if (a.empty() || x <= a.top())
          a.push(x);
        else
          b.push(x);
      }
      // 对对顶堆进行调整
      if (a.size() > (a.size() + b.size() + 1) / 2) {
        b.push(a.top());
        a.pop();
      } else if (a.size() < (a.size() + b.size() + 1) / 2) {
        a.push(b.top());
        b.pop();
      }
    }
  }
  return 0;
}
```

- 双倍经验：SP15376 RMID - Running Median[2]
- 典型习题：P1801 黑匣子[3]

参考资料与注释

[1] SP16254 RMID2 - Running Median Again

[2] SP15376 RMID - Running Median

[3] P1801 黑匣子

10.7.3 配对堆

引入

配对堆是一个支持插入，查询/删除最小值，合并，修改元素等操作的数据结构，是一种可并堆。有速度快和结构简单的优势，但由于其为基于势能分析的均摊复杂度，无法可持久化。

定义

配对堆是一棵满足堆性质的带权多叉树（如下图），即每个节点的权值都小于或等于他的所有儿子（以小根堆为例，下同）。

通常我们使用儿子 - 兄弟表示法储存一个配对堆（如下图），一个节点的所有儿子节点形成一个单向链表。每个节点储存第一个儿子的指针，即链表的头节点；和他的右兄弟的指针。

这种方式便于实现配对堆，也将方便复杂度分析。

```
struct Node {
  T v;  // T 为权值类型
  Node *child, *sibling;
  // child 指向该节点第一个儿子，sibling 指向该节点的下一个兄弟。
```

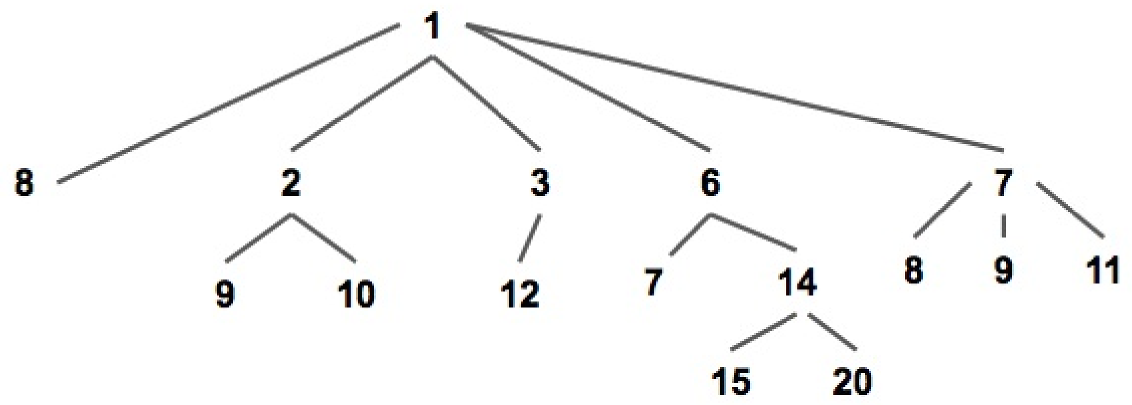


图 10.22

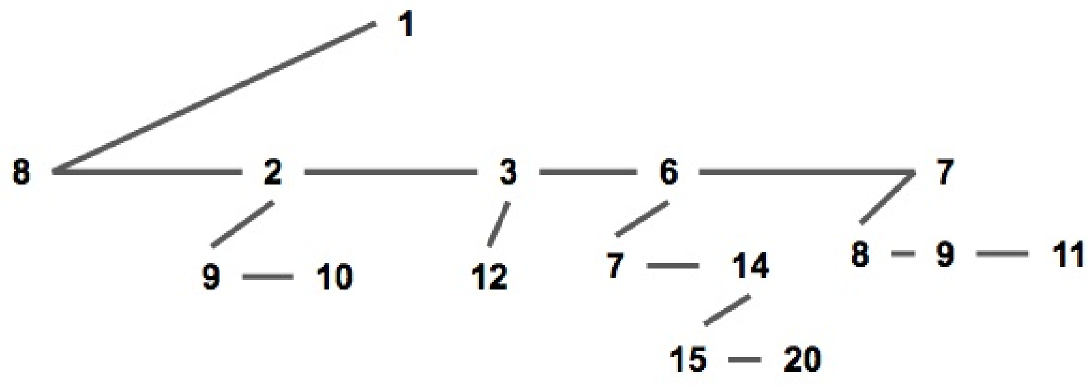


图 10.23

```
  // 若该节点没有儿子/下个兄弟则指针指向 nullptr。
};
```

从定义可以发现，和其他常见的堆结构相比，配对堆不维护任何额外的树大小，深度，排名等信息（二叉堆也不维护额外信息，但它是通过维持一个严格的完全二叉树结构来保证操作的复杂度），且任何一个满足堆性质的树都是一个合法的配对堆，这样简单又高度灵活的数据结构奠定了配对堆在实践中优秀效率的基础；作为对比，斐波那契堆糟糕的常数就是因为它需要维护很多额外的信息。

配对堆通过一套精心设计的操作顺序来保证它的总复杂度，原论文[1-1] 将其称为「一种自调整的堆（Self Adjusting Heap）」。在这方面和 Splay 树（在原论文中被称作「Self Adjusting Binary Tree」）颇有相似之处。

过程

查询最小值

从配对堆的定义可看出，配对堆的根节点的权值一定最小，直接返回根节点即可。

合并

合并两个配对堆的操作很简单，首先令两个根节点较小的一个为新的根节点，然后将较大的根节点作为它的儿子插入进去。（见下图）

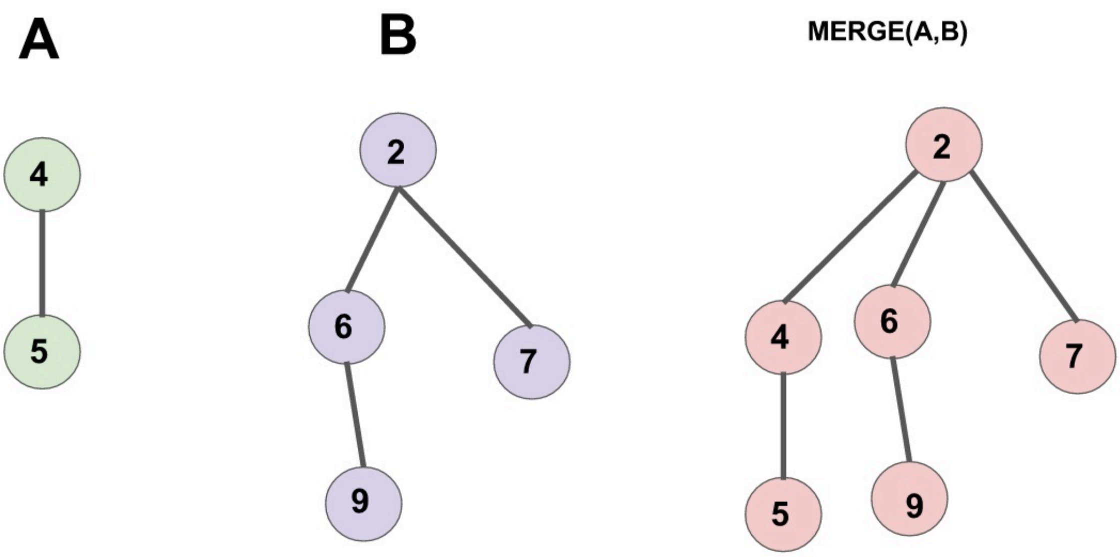


图 10.24

需要注意的是，一个节点的儿子链表是按插入时间排序的，即最右边的节点最早成为父节点的儿子，最左边的节点最近成为父节点的儿子。

" 实现"

```
Node* meld(Node* x, Node* y) {
  // 若有一个为空则直接返回另一个
  if (x == nullptr) return y;
  if (y == nullptr) return x;
  if (x->v > y->v) std::swap(x, y);  // swap 后 x 为权值小的堆，y 为权值大的堆
  // 将 y 设为 x 的儿子
  y->sibling = x->child;
  x->child = y;
  return x;  // 新的根节点为 x
}
```

插入

合并都有了，插入就直接把新元素视为一个新的配对堆和原堆合并就行了。

删除最小值

首先要提及的一点是，上文的几个操作都十分偷懒，完全没有对数据结构进行维护，所以我们需要小心设计删除最小值的操作，来保证总复杂度不出问题。

根节点即为最小值，所以要删除的是根节点。考虑拿掉根节点之后会发生什么：根节点原来的所有儿子构成了一片森林；而配对堆应当是一棵树，所以我们需要通过某种顺序把这些儿子全部合并起来。

一个很自然的想法是使用 `meld` 函数把儿子们从左到右挨个并在一起，这样做的话正确性是显然的，但是会导致单次操作复杂度退化到 $O(n)$。

为了保证总的均摊复杂度，需要使用一个「两步走」的合并方法：

1. 把儿子们两两配成一对，用 `meld` 操作把被配成同一对的两个儿子合并到一起（见下图 1），
2. 将新产生的堆**从右往左**（即老的儿子到新的儿子的方向）挨个合并在一起（见下图 2）。

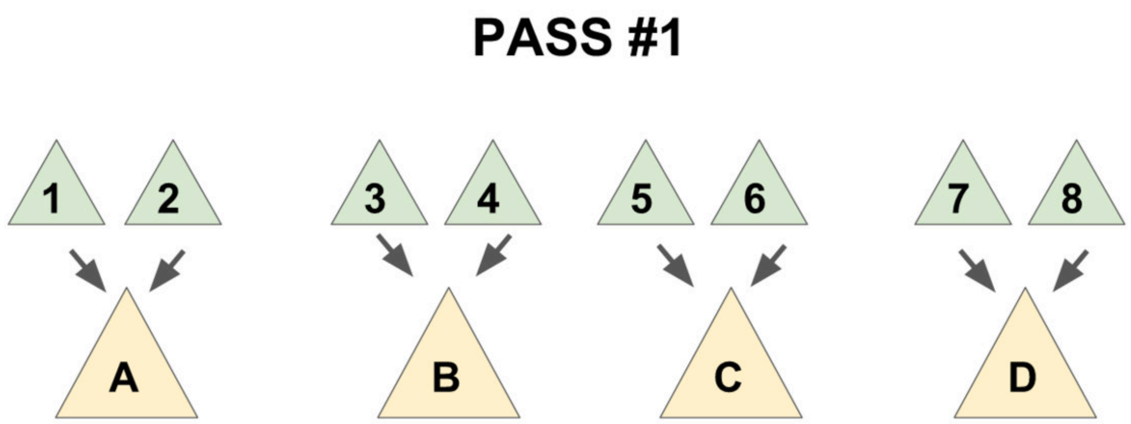


图 10.25

先实现一个辅助函数 `merges`，作用是合并一个节点的所有兄弟。

" 实现"

```
Node* merges(Node* x) {
  if (x == nullptr || x->sibling == nullptr)
    return x;  // 如果该树为空或他没有下一个兄弟，就不需要合并了，return。
  Node* y = x->sibling;                  // y 为 x 的下一个兄弟
  Node* c = y->sibling;                  // c 是再下一个兄弟
  x->sibling = y->sibling = nullptr;   // 拆散
  return meld(merges(c), meld(x, y));  // 核心部分
}
```

最后一句话是该函数的核心，这句话分三部分：

1. `meld(x,y)`「配对」了 x 和 y。
2. `merges(c)` 递归合并 c 和他的兄弟们。
3. 将上面 2 个操作产生的 2 个新树合并。

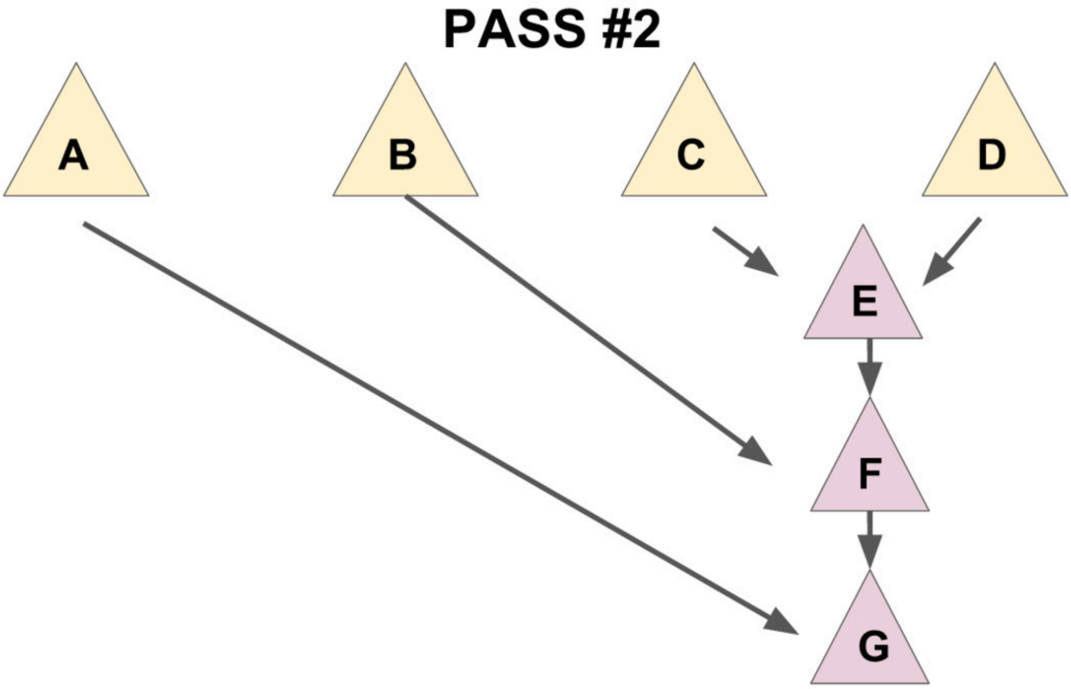


图 10.26

需要注意到的是，上文提到了第二步时的合并方向是有要求的（从右往左合并），该递归函数的实现已保证了这个顺序，如果读者需要自行实现迭代版本的话请务必注意保证该顺序，否则复杂度将失去保证。

有了 merges 函数，delete-min 操作就显然了。

" 实现"

```
Node* delete_min(Node* x) {
  Node* t = merges(x->child);
  delete x;  // 如果需要内存回收
  return t;
}
```

减小一个元素的值

要实现这个操作，需要给节点添加一个「父」指针，当节点有左兄弟时，其指向左兄弟而非实际的父节点；否则，指向其父节点。

首先节点的定义修改为：

" 实现"

```
struct Node {
  LL v;
  int id;
  Node *child, *sibling;
  Node *father;  // 新增：父指针，若该节点为根节点则指向空节点 nullptr
};
```

meld 操作修改为：

" 实现"

```
Node* meld(Node* x, Node* y) {
  if (x == nullptr) return y;
  if (y == nullptr) return x;
  if (x->v > y->v) std::swap(x, y);
  if (x->child != nullptr) {  // 新增：维护父指针
    x->child->father = y;
  }
  y->sibling = x->child;
  y->father = x;  // 新增：维护父指针
  x->child = y;
  return x;
}
```

merges 操作修改为：

" 实现"

```
Node *merges(Node *x) {
  if (x == nullptr) return nullptr;
  x->father = nullptr;  // 新增：维护父指针
  if (x->sibling == nullptr) return x;
  Node *y = x->sibling, *c = y->sibling;
  y->father = nullptr;  // 新增：维护父指针
  x->sibling = y->sibling = nullptr;
  return meld(merges(c), meld(x, y));
}
```

现在我们来考虑如何实现 decrease-key 操作。

首先我们发现，当我们减少节点 x 的权值之后，以 x 为根的子树仍然满足配对堆性质，但 x 的父亲和 x 之间可能不再满足堆性质。

因此我们把整棵以 x 为根的子树剖出来，现在两棵树都符合配对堆性质了，然后把他们合并起来，就完成了全部操作。

" 实现"

```
// root 为堆的根，x 为要操作的节点，v 为新的权值，调用时需保证 v <= x->v
// 返回值为新的根节点
Node *decrease_key(Node *root, Node *x, LL v) {
  x->v = v;                 // 更新权值
  if (x == root) return x;  // 如果 x 为根，则直接返回
  // 把 x 从 fa 的子节点中剖出去，这里要分 x 的位置讨论一下。
  if (x->father->child == x) {
    x->father->child = x->sibling;
  } else {
    x->father->sibling = x->sibling;
  }
  if (x->sibling != nullptr) {
    x->sibling->father = x->father;
  }
  x->sibling = nullptr;
  x->father = nullptr;
```

```
    return meld(root, x);  // 重新合并 x 和根节点
}
```

复杂度分析

配对堆结构与实现简单，但时间复杂度分析并不容易。

原论文[1-2] 仅将复杂度分析到 `meld` 和 `delete-min` 操作均为均摊 $O(\log n)$，但提出猜想认为其各操作都有和斐波那契堆相同的复杂度。

遗憾的是，后续发现，不维护额外信息的配对堆，在特定的操作序列下，`decrease-key` 操作的均摊复杂度下界至少为 $\Omega(\log\log n)$[2]。

目前对复杂度上界比较好的估计有，Iacono 的 $O(1)$ `meld`，$O(\log n)$ `decrease-key`[3]；Pettie 的 $O(2^{2\sqrt{\log\log n}})$ `meld` 和 `decrease-key`[4]。需要注意的是，前述复杂度均为均摊复杂度，因此不能对各结果分别取最小值。

参考文献

- https://en.wikipedia.org/wiki/Pairing_heap[5]
- https://brilliant.org/wiki/pairing-heap/[6]

[1] The pairing heap: a new form of self-adjusting heap [1-1] [1-2]

[2] On the efficiency of pairing heaps and related data structures

[3] Improved upper bounds for pairing heaps

[4] Towards a Final Analysis of Pairing Heaps

[5] https://en.wikipedia.org/wiki/Pairing_heap

[6] https://brilliant.org/wiki/pairing-heap/

10.7.4 左偏树

什么是左偏树？

左偏树与 **配对堆** 一样，是一种**可并堆**，具有堆的性质，并且可以快速合并。

dist 的定义和性质

对于一棵二叉树，我们定义**外节点**为左儿子或右儿子为空的节点，定义一个外节点的 dist 为 1，一个不是外节点的节点 dist 为其到子树中最近的外节点的距离加一。空节点的 dist 为 0。

> 注：很多其它教程中定义的 dist 都是本文中的 dist 减去 1，本文这样定义是因为代码写起来方便。

一棵有 n 个节点的二叉树，根的 dist 不超过 $\lceil\log(n+1)\rceil$，因为一棵根的 dist 为 x 的二叉树至少有 $x-1$ 层是满二叉树，那么就至少有 2^x-1 个节点。注意这个性质是所有二叉树都具有的，并不是左偏树所特有的。

左偏树的定义和性质

左偏树是一棵二叉树，它不仅具有堆的性质，并且是「左偏」的：每个节点左儿子的 dist 都大于等于右儿子的 dist。

因此，左偏树每个节点的 dist 都等于其右儿子的 dist 加一。

需要注意的是，dist 不是深度，**左偏树的深度没有保证**，一条向左的链也是左偏树。

核心操作：合并（merge）

合并两个堆时，由于要满足堆性质，先取值较小（为了方便，本文讨论小根堆）的那个根作为合并后堆的根节点，然后将这个根的左儿子作为合并后堆的左儿子，递归地合并其右儿子与另一个堆，作为合并后的堆的右儿子。为了满足左偏性质，合并后若左儿子的 dist 小于右儿子的 dist，就交换两个儿子。

参考代码：

" 实现"

```
int merge(int x, int y) {
  if (!x || !y) return x | y;  // 若一个堆为空则返回另一个堆
  if (t[x].val > t[y].val) swap(x, y);  // 取值较小的作为根
  t[x].rs = merge(t[x].rs, y);          // 递归合并右儿子与另一个堆
  if (t[t[x].rs].d > t[t[x].ls].d)
    swap(t[x].ls, t[x].rs);   // 若不满足左偏性质则交换左右儿子
  t[x].d = t[t[x].rs].d + 1;  // 更新 dist
  return x;
}
```

由于左偏性质，每递归一层，其中一个堆根节点的 dist 就会减小 1，而「一棵有 n 个节点的二叉树，根的 dist 不超过 $\lceil\log(n+1)\rceil$」，所以合并两个大小分别为 n 和 m 的堆复杂度是 $O(\log n+\log m)$。

左偏树还有一种无需交换左右儿子的写法：将 dist 较大的儿子视作左儿子，dist 较小的儿子视作右儿子：

" 实现"

```
int& rs(int x) { return t[x].ch[t[t[x].ch[1]].d < t[t[x].ch[0]].d]; }

int merge(int x, int y) {
  if (!x || !y) return x | y;
  if (t[x].val < t[y].val) swap(x, y);
  rs(x) = merge(rs(x), y);
  t[x].d = t[rs(x)].d + 1;
  return x;
}
```

左偏树的其它操作

插入节点

单个节点也可以视为一个堆，合并即可。

删除根

合并根的左右儿子即可。

删除任意节点

做法

先将左右儿子合并，然后自底向上更新 dist、不满足左偏性质时交换左右儿子，当 dist 无需更新时结束递归：

" 实现"

```cpp
int& rs(int x) { return t[x].ch[t[t[x].ch[1]].d < t[t[x].ch[0]].d]; }

// 有了 pushup，直接 merge 左右儿子就实现了删除节点并保持左偏性质
int merge(int x, int y) {
  if (!x || !y) return x | y;
  if (t[x].val < t[y].val) swap(x, y);
  t[rs(x) = merge(rs(x), y)].fa = x;
  pushup(x);
  return x;
}

void pushup(int x) {
  if (!x) return;
  if (t[x].d != t[rs(x)].d + 1) {
    t[x].d = t[rs(x)].d + 1;
    pushup(t[x].fa);
  }
}
```

复杂度证明

我们令当前 pushup 的这个节点为 x，其父亲为 y，一个节点的「初始 dist」为它在 pushup 前的 dist。我们先 pushup 一下删除的节点，然后从其父亲开始起讨论复杂度。

继续递归下去有两种情况：

1. x 是 y 的右儿子，此时 y 的初始 dist 为 x 的初始 dist 加一。
2. x 是 y 的左儿子，只有 y 的左右儿子初始 dist 相等时（此时左儿子 dist 减一会导致左右儿子互换）才会继续递归下去，因此 y 的初始 dist 仍然是 x 的初始 dist 加一。

所以，我们得到，除了第一次 pushup（因为被删除节点的父亲的初始 dist 不一定等于被删除节点左右儿子合并后的初始 dist 加一），每递归一层 x 的初始 dist 就会加一，因此最多递归 $O(\log n)$ 层。

整个堆加上/减去一个值、乘上一个正数

其实可以打标记且不改变相对大小的操作都可以。

在根打上标记，删除根/合并堆（访问儿子）时下传标记即可：

" 实现"

```cpp
int merge(int x, int y) {
  if (!x || !y) return x | y;
  if (t[x].val > t[y].val) swap(x, y);
  pushdown(x);
  t[x].rs = merge(t[x].rs, y);
  if (t[t[x].rs].d > t[t[x].ls].d) swap(t[x].ls, t[x].rs);
  t[x].d = t[t[x].rs].d + 1;
  return x;
}

int pop(int x) {
  pushdown(x);
  return merge(t[x].ls, t[x].rs);
```

```
}
```

随机合并

直接贴上代码

" 实现"

```
int merge(int x, int y) {
  if (!x || !y) return x | y;
  if (t[y].val < t[x].val) swap(x, y);
  if (rand() & 1)  // 随机选择是否交换左右子节点
    swap(t[x].ls, t[x].rs);
  t[x].ls = merge(t[x].ls, y);
  return x;
}
```

可以看到该实现方法唯一不同之处便是采用了随机数来实现合并，这样一来便可以省去 dist 的相关计算。且平均时间复杂度亦为 $O(\log n)$，详细证明可参考 Randomized Heap[1]。

例题

模板题

luogu P3377【模板】左偏树（可并堆）[2]

Monkey King[3]

罗马游戏[4]

需要注意的是：

1. 合并前要检查是否已经在同一堆中。
2. 左偏树的深度可能达到 $O(n)$，因此找一个点所在的堆顶要用并查集维护，不能直接暴力跳父亲。（虽然很多题数据水，暴力跳父亲可以过……）（用并查集维护根时要保证原根指向新根，新根指向自己。）

" 罗马游戏参考代码"

```
#include <algorithm>
#include <cctype>
#include <cstdio>
#include <iostream>

using namespace std;

int read() {  // 快读
  int out = 0;
  char c;
  while (!isdigit(c = getchar()))
    ;
  for (; isdigit(c); c = getchar()) out = out * 10 + c - '0';
  return out;
}

const int N = 1000010;
```

```
struct Node {
  int val, ch[2], d;
} t[N];

int& rs(int x);
int merge(int x, int y);

int find(int x);

int n, m, f[N];
bool kill[N];
char op[10];

int main() {
  int i, x, y;

  n = read();

  for (i = 1; i <= n; ++i) {
    t[i].val = read();
    f[i] = i;
  }

  m = read();

  while (m--) {
    scanf("%s", op);
    if (op[0] == 'M') {
      x = read();
      y = read();
      if (kill[x] || kill[y] || find(x) == find(y)) continue;  // 这是题意
      f[find(x)] = f[find(y)] = merge(find(x), find(y));
    } else {
      x = read();
      if (!kill[x]) {
        x = find(x);
        kill[x] = true;
        f[x] = f[t[x].ch[0]] = f[t[x].ch[1]] = merge(t[x].ch[0], t[x].ch[1]);
        // 由于堆中的点会 find 到 x，所以 f[x] 也要修改
        printf("%d\n", t[x].val);
      } else
        puts("0");
    }
  }

  return 0;
}

int& rs(int x) { return t[x].ch[t[t[x].ch[1]].d < t[t[x].ch[0]].d]; }

int merge(int x, int y) {  // 左偏树并堆
  if (!x || !y) return x | y;
  if (t[x].val > t[y].val) swap(x, y);
  rs(x) = merge(rs(x), y);
```

```
  t[x].d = t[rs(x)].d + 1;
  return x;
}

int find(int x) { return x == f[x] ? x : f[x] = find(f[x]); }  // 查找
```

树上问题

「APIO2012」派遣[5]

「JLOI2015」城池攻占[6]

这类题目往往是每个节点维护一个堆，与儿子合并，依题意弹出、修改、计算答案，有点像线段树合并的类似题目。

" 城池攻占参考代码"

```
#include <algorithm>
#include <cctype>
#include <cstdio>
#include <iostream>

using namespace std;

typedef long long ll;

ll read() {
  ll out = 0;
  int f = 1;
  char c;
  for (c = getchar(); !isdigit(c) && c != '-'; c = getchar())
    ;
  if (c == '-') f = -1, c = getchar();
  for (; isdigit(c); c = getchar()) out = out * 10 + c - '0';
  return out * f;
}

const int N = 300010;

struct Node {
  int ls, rs, d;
  ll val, add, mul;

  Node() {
    ls = rs = add = 0;
    d = mul = 1;
  }
} t[N];

int merge(int x, int y);
int pop(int x);
void madd(int u, ll x);
void mmul(int u, ll x);
void pushdown(int x);

void add(int u, int v);
```

```
void dfs(int u);

int head[N], nxt[N], to[N], cnt;
int n, m, p[N], f[N], a[N], dep[N], c[N], ans1[N],
    ans2[N];  // p 是树上每个点对应的堆顶
ll h[N], b[N];

int main() {
  int i;

  n = read();
  m = read();

  for (i = 1; i <= n; ++i) h[i] = read();

  for (i = 2; i <= n; ++i) {
    f[i] = read();
    add(f[i], i);
    a[i] = read();
    b[i] = read();
  }

  for (i = 1; i <= m; ++i) {
    t[i].val = read();
    c[i] = read();
    p[c[i]] = merge(i, p[c[i]]);
  }

  dfs(1);

  for (i = 1; i <= n; ++i) printf("%d\n", ans1[i]);
  for (i = 1; i <= m; ++i) printf("%d\n", ans2[i]);

  return 0;
}

void dfs(int u) {
  int i, v;  // 根据题意开始 dfs
  for (i = head[u]; i; i = nxt[i]) {
    v = to[i];
    dep[v] = dep[u] + 1;
    dfs(v);
  }
  while (p[u] && t[p[u]].val < h[u]) {
    ++ans1[u];
    ans2[p[u]] = dep[c[p[u]]] - dep[u];
    p[u] = pop(p[u]);
  }
  if (a[u])
    mmul(p[u], b[u]);
  else
    madd(p[u], b[u]);
  if (u > 1)
    p[f[u]] = merge(p[u], p[f[u]]);
```

```
    else
      while (p[u]) {
        ans2[p[u]] = dep[c[p[u]]] + 1;
        p[u] = pop(p[u]);
      }
  }

  void add(int u, int v) {
    nxt[++cnt] = head[u];
    head[u] = cnt;
    to[cnt] = v;
  }

  int merge(int x, int y) {  // 合并
    if (!x || !y) return x | y;
    if (t[x].val > t[y].val) swap(x, y);
    pushdown(x);
    t[x].rs = merge(t[x].rs, y);
    if (t[t[x].rs].d > t[t[x].ls].d) swap(t[x].ls, t[x].rs);
    t[x].d = t[t[x].rs].d + 1;
    return x;
  }

  int pop(int x) {
    pushdown(x);
    return merge(t[x].ls, t[x].rs);
  }

  void madd(int u, ll x) {
    t[u].val += x;
    t[u].add += x;
  }

  void mmul(int u, ll x) {
    t[u].val *= x;
    t[u].add *= x;
    t[u].mul *= x;
  }

  void pushdown(int x) {  // like 线段树，下传合并
    mmul(t[x].ls, t[x].mul);
    madd(t[x].ls, t[x].add);
    mmul(t[x].rs, t[x].mul);
    madd(t[x].rs, t[x].add);
    t[x].add = 0;
    t[x].mul = 1;
  }
```

「SCOI2011」 棘手的操作[7]

首先，找一个节点所在堆的堆顶要用并查集，而不能暴力向上跳。

再考虑单点查询，若用普通的方法打标记，就得查询点到根路径上的标记之和，最坏情况下可以达到 $O(n)$ 的复杂度。如果只有堆顶有标记，就可以快速地查询了，但如何做到呢？

可以用类似启发式合并的方式，每次合并的时候把较小的那个堆标记暴力下传到每个节点，然后把较大的堆的标记作为合并后的堆的标记。由于合并后有另一个堆的标记，所以较小的堆下传标记时要下传其标记减去另一个堆的标

记。由于每个节点每被合并一次所在堆的大小至少乘二，所以每个节点最多被下放 $O(\log n)$ 次标记，暴力下放标记的总复杂度就是 $O(n \log n)$。

再考虑单点加，先删除，再更新，最后插入即可。

然后是全局最大值，可以用一个平衡树/支持删除任意节点的堆（如左偏树）/multiset 来维护每个堆的堆顶。

所以，每个操作分别如下：

1. 暴力下传点数较小的堆的标记，合并两个堆，更新 size、tag，在 multiset 中删去合并后不在堆顶的那个原堆顶。
2. 删除节点，更新值，插入回来，更新 multiset。需要分删除节点是否为根来讨论一下。
3. 堆顶打标记，更新 multiset。
4. 打全局标记。
5. 查询值 + 堆顶标记 + 全局标记。
6. 查询根的值 + 堆顶标记 + 全局标记。
7. 查询 multiset 最大值 + 全局标记。

"棘手的操作参考代码"

```cpp
#include <algorithm>
#include <cctype>
#include <cstdio>
#include <iostream>
#include <set>

using namespace std;

int read() {
  int out = 0, f = 1;
  char c;
  for (c = getchar(); !isdigit(c) && c != '-'; c = getchar())
    ;
  if (c == '-') {
    f = -1;
    c = getchar();
  }
  for (; isdigit(c); c = getchar()) out = out * 10 + c - '0';
  return out * f;
}

const int N = 300010;

struct Node {
  int val, ch[2], d, fa;
} t[N];

int& rs(int x);
int merge(int x, int y);
void pushup(int x);
void pushdown(int x, int y);

int find(int x);

int n, m, f[N], tag[N], siz[N], delta;
```

```
char op[10];
multiset<int> s;

int main() {
  int i, x, y;

  n = read();

  for (i = 1; i <= n; ++i) {
    t[i].val = read();
    f[i] = i;
    siz[i] = 1;
    s.insert(t[i].val);
  }

  m = read();

  while (m--) {  // 根据题意处理具体看解法
    scanf("%s", op);
    if (op[0] == 'U') {
      x = find(read());
      y = find(read());
      if (x != y) {
        if (siz[x] > siz[y]) swap(x, y);
        pushdown(x, tag[x] - tag[y]);
        f[x] = f[y] = merge(x, y);
        if (f[x] == x) {
          s.erase(s.find(t[y].val + tag[y]));
          tag[x] = tag[y];
          siz[x] += siz[y];
          tag[y] = siz[y] = 0;
        } else {
          s.erase(s.find(t[x].val + tag[y]));
          siz[y] += siz[x];
          tag[x] = siz[x] = 0;
        }
      }
    } else if (op[0] == 'A') {
      if (op[1] == '1') {
        x = read();
        if (x == find(x)) {
          t[t[x].ch[0]].fa = t[t[x].ch[1]].fa = 0;
          y = merge(t[x].ch[0], t[x].ch[1]);
          s.erase(s.find(t[x].val + tag[x]));
          t[x].val += read();
          t[x].fa = t[x].ch[0] = t[x].ch[1] = 0;
          t[x].d = 1;
          f[x] = f[y] = merge(x, y);
          s.insert(t[f[x]].val + tag[x]);
          if (f[x] == y) {
            tag[y] = tag[x];
            siz[y] = siz[x];
            tag[x] = siz[x] = 0;
          }
```

```
        } else {
          t[t[x].ch[0]].fa = t[t[x].ch[1]].fa = t[x].fa;
          t[t[x].fa].ch[x == t[t[x].fa].ch[1]] = merge(t[x].ch[0], t[x].ch[1]);
          t[x].val += read();
          t[x].fa = t[x].ch[0] = t[x].ch[1] = 0;
          t[x].d = 1;
          y = find(x);
          f[x] = f[y] = merge(x, y);
          if (f[x] == x) {
            s.erase(s.find(t[y].val + tag[y]));
            s.insert(t[x].val + tag[y]);
            tag[x] = tag[y];
            siz[x] = siz[y];
            tag[y] = siz[y] = 0;
          }
        }
      } else if (op[1] == '2') {
        x = find(read());
        s.erase(s.find(t[x].val + tag[x]));
        tag[x] += read();
        s.insert(t[x].val + tag[x]);
      } else
        delta += read();
    } else {
      if (op[1] == '1') {
        x = read();
        printf("%d\n", t[x].val + tag[find(x)] + delta);
      } else if (op[1] == '2') {
        x = find(read());
        printf("%d\n", t[x].val + tag[x] + delta);
      } else
        printf("%d\n", *s.rbegin() + delta);
    }
  }

  return 0;
}

int& rs(int x) { return t[x].ch[t[t[x].ch[1]].d < t[t[x].ch[0]].d]; }

int merge(int x, int y) {  // 板子，合并
  if (!x || !y) return x | y;
  if (t[x].val < t[y].val) swap(x, y);
  t[rs(x) = merge(rs(x), y)].fa = x;
  pushup(x);
  return x;
}

// 以下俩是一个东西
void pushup(int x) {
  if (!x) return;
  if (t[x].d != t[rs(x)].d + 1) {
    t[x].d = t[rs(x)].d + 1;
    pushup(t[x].fa);
```

```
  }
}

void pushdown(int x, int y) {
  if (!x) return;
  t[x].val += y;
  pushdown(t[x].ch[0], y);
  pushdown(t[x].ch[1], y);
}

int find(int x) { return x == f[x] ? x : f[x] = find(f[x]); }
```

「BOI2004」Sequence 数字序列[8]

这是一道论文题，详见 《 黄源河 -- 左偏树的特点及其应用 》[9]。

参考资料与注释

[1] Randomized Heap

[2] luogu P3377【模板】左偏树（可并堆）

[3] Monkey King

[4] 罗马游戏

[5] 「APIO2012」派遣

[6] 「JLOI2015」城池攻占

[7] 「SCOI2011」棘手的操作

[8] 「BOI2004」Sequence 数字序列

[9] 《黄源河 – 左偏树的特点及其应用》

10.8 块状数据结构

10.8.1 分块思想

Authors: Ir1d, HeRaNO, Xeonacid

简介

其实，分块是一种思想，而不是一种数据结构。

从 NOIP 到 NOI 到 IOI，各种难度的分块思想都有出现。

分块的基本思想是，通过对原数据的适当划分，并在划分后的每一个块上预处理部分信息，从而较一般的暴力算法取得更优的时间复杂度。

分块的时间复杂度主要取决于分块的块长，一般可以通过均值不等式求出某个问题下的最优块长，以及相应的时间复杂度。

分块是一种很灵活的思想，相较于树状数组和线段树，分块的优点是通用性更好，可以维护很多树状数组和线段树无法维护的信息。

当然，分块的缺点是渐进意义的复杂度，相较于线段树和树状数组不够好。

不过在大多数问题上，分块仍然是解决这些问题的一个不错选择。

下面是几个例子。

区间和

" **例题** LibreOJ 6280 **数列分块入门** 4[1]"

给定一个长度为 n 的序列 $\{a_i\}$，需要执行 n 次操作。操作分为两种：

1. 给 $a_l \sim a_r$ 之间的所有数加上 x；
2. 求 $\sum_{i=l}^{r} a_i$。

$1 \le n \le 5 \times 10^4$

我们将序列按每 s 个元素一块进行分块，并记录每块的区间和 b_i。

$$\underbrace{a_1, a_2, \cdots, a_s}_{b_1}, \underbrace{a_{s+1}, \cdots, a_{2s}}_{b_2}, \cdots, \underbrace{a_{(s-1)\times s+1}, \cdots, a_n}_{b_{\frac{n}{s}}}$$

最后一个块可能是不完整的（因为 n 很可能不是 s 的倍数），但是这对于我们的讨论来说并没有太大影响。

首先看查询操作：

- 若 l 和 r 在同一个块内，直接暴力求和即可，因为块长为 s，因此最坏复杂度为 $O(s)$。
- 若 l 和 r 不在同一个块内，则答案由三部分组成：以 l 开头的不完整块，中间几个完整块，以 r 结尾的不完整块。对于不完整的块，仍然采用上面暴力计算的方法，对于完整块，则直接利用已经求出的 b_i 求和即可。这种情况下，最坏复杂度为 $O(\dfrac{n}{s} + s)$。

接下来是修改操作：

- 若 l 和 r 在同一个块内，直接暴力修改即可，因为块长为 s，因此最坏复杂度为 $O(s)$。
- 若 l 和 r 不在同一个块内，则需要修改三部分：以 l 开头的不完整块，中间几个完整块，以 r 结尾的不完整块。对于不完整的块，仍然是暴力修改每个元素的值（别忘了更新区间和 b_i），对于完整块，则直接修改 b_i 即可。这种情况下，最坏复杂度和仍然为 $O(\dfrac{n}{s} + s)$。

利用均值不等式可知，当 $\dfrac{n}{s} = s$，即 $s = \sqrt{n}$ 时，单次操作的时间复杂度最优，为 $O(\sqrt{n})$。

" **参考代码**"

```
#include <cmath>
#include <iostream>
using namespace std;
int id[50005], len;
// id 表示块的编号, len=sqrt(n) , 即上述题解中的 s, sqrt 的时候时间复杂度最优
long long a[50005], b[50005], s[50005];

// a 数组表示数据数组, b 数组记录每个块的整体赋值情况, 类似于 lazy_tag, s
// 表示块内元素总和
void add(int l, int r, long long x) {  // 区间加法
```

```cpp
  int sid = id[l], eid = id[r];
  if (sid == eid) {  // 在一个块中
    for (int i = l; i <= r; i++) a[i] += x, s[sid] += x;
    return;
  }
  for (int i = l; id[i] == sid; i++) a[i] += x, s[sid] += x;
  for (int i = sid + 1; i < eid; i++)
    b[i] += x, s[i] += len * x;  // 更新区间和数组（完整的块）
  for (int i = r; id[i] == eid; i--) a[i] += x, s[eid] += x;
  // 以上两行不完整的块直接简单求和，就 OK
}

long long query(int l, int r, long long p) {  // 区间查询
  int sid = id[l], eid = id[r];
  long long ans = 0;
  if (sid == eid) {  // 在一个块里直接暴力求和
    for (int i = l; i <= r; i++) ans = (ans + a[i] + b[sid]) % p;
    return ans;
  }
  for (int i = l; id[i] == sid; i++) ans = (ans + a[i] + b[sid]) % p;
  for (int i = sid + 1; i < eid; i++) ans = (ans + s[i]) % p;
  for (int i = r; id[i] == eid; i--) ans = (ans + a[i] + b[eid]) % p;
  // 和上面的区间修改是一个道理
  return ans;
}

int main() {
  int n;
  cin >> n;
  len = sqrt(n);  // 均值不等式可知复杂度最优为根号 n
  for (int i = 1; i <= n; i++) {  // 题面要求
    cin >> a[i];
    id[i] = (i - 1) / len + 1;
    s[id[i]] += a[i];
  }
  for (int i = 1; i <= n; i++) {
    int op, l, r, c;
    cin >> op >> l >> r >> c;
    if (op == 0)
      add(l, r, c);
    else
      cout << query(l, r, c + 1) << endl;
  }
  return 0;
}

/*
https://loj.ac/s/1151495
 */
```

区间和 2

上一个做法的复杂度是 $\Omega(1), O(\sqrt{n})$。

我们在这里介绍一种 $O(\sqrt{n}) - O(1)$ 的算法。

为了 $O(1)$ 询问，我们可以维护各种前缀和。

然而在有修改的情况下，不方便维护，只能维护单个块内的前缀和。

以及整块作为一个单位的前缀和。

每次修改 $O(T+\frac{n}{T})$。

询问：涉及三部分，每部分都可以直接通过前缀和得到，时间复杂度 $O(1)$。

对询问分块

同样的问题，现在序列长度为 n，有 m 个操作。

如果操作数量比较少，我们可以把操作记下来，在询问的时候加上这些操作的影响。

假设最多记录 T 个操作，则修改 $O(1)$，询问 $O(T)$。

T 个操作之后，重新计算前缀和，$O(n)$。

总复杂度：$O(mT+n\frac{m}{T})$。

$T=\sqrt{n}$ 时，总复杂度 $O(m\sqrt{n})$。

其他问题

分块思想也可以应用于其他整数相关问题：寻找零元素的数量、寻找第一个非零元素、计算满足某个性质的元素个数等等。

还有一些问题可以通过分块来解决，例如维护一组允许添加或删除数字的集合，检查一个数是否属于这个集合，以及查找第 k 大的数。要解决这个问题，必须将数字按递增顺序存储，并分割成多个块，每个块中包含 $\sqrt{n}$ 个数字。每次添加或删除一个数字时，必须通过在相邻块的边界移动数字来重新分块。

一种很有名的离线算法 莫队算法，也是基于分块思想实现的。

练习题

- UVa - 12003 - Array Transformer[2]
- UVa - 11990 Dynamic Inversion[3]
- SPOJ - Give Away[4]
- Codeforces - Till I Collapse[5]
- Codeforces - Destiny[6]
- Codeforces - Holes[7]
- Codeforces - XOR and Favorite Number[8]
- Codeforces - Powerful array[9]
- SPOJ - DQUERY[10]

参考资料与注释

[1] LibreOJ 6280 数列分块入门 4

[2] UVa - 12003 - Array Transformer

[3] UVa - 11990 Dynamic Inversion

[4] SPOJ - Give Away

[5] Codeforces - Till I Collapse

[6] Codeforces - Destiny

[7] Codeforces - Holes

[8] Codeforces - XOR and Favorite Number

[9] Codeforces - Powerful array

[10] SPOJ - DQUERY

[11] Sqrt-

[12] Sqrt Decomposition

10.8.2 块状数组

建立块状数组

块状数组，即把一个数组分为几个块，块内信息整体保存，若查询时遇到两边不完整的块直接暴力查询。一般情况下，块的长度为 $O(\sqrt{n})$。详细分析可以阅读 2017 年国家集训队论文中徐明宽的《非常规大小分块算法初探》。

下面直接给出一种建立块状数组的代码。

" 实现"

```
num = sqrt(n);
for (int i = 1; i <= num; i++)
  st[i] = n / num * (i - 1) + 1, ed[i] = n / num * i;
ed[num] = n;
for (int i = 1; i <= num; i++) {
  for (int j = st[i]; j <= ed[i]; j++) {
    belong[j] = i;
  }
  size[i] = ed[i] - st[i] + 1;
}
```

其中 `st[i]` 和 `ed[i]` 为块的起点和终点，`size[i]` 为块的大小。

保存与修改块内信息

例题 1：教主的魔法[1]

两种操作：

1. 区间 $[x, y]$ 每个数都加上 z；
2. 查询区间 $[x, y]$ 内大于等于 z 的数的个数。

我们要询问一个块内大于等于一个数的数的个数，所以需要一个 `t` 数组对块内排序，`a` 为原来的（未被排序的）数组。对于整块的修改，使用类似于标记永久化的方式，用 `delta` 数组记录现在块内整体加上的值。设 q 为查询和修改的操作次数总和，则时间复杂度 $O(q\sqrt{n}\log n)$。

用 `delta` 数组记录每个块的整体赋值情况。

"实现"

```cpp
void Sort(int k) {
  for (int i = st[k]; i <= ed[k]; i++) t[i] = a[i];
  sort(t + st[k], t + ed[k] + 1);
}

void Modify(int l, int r, int c) {
  int x = belong[l], y = belong[r];
  if (x == y)  // 区间在一个块内就直接修改
  {
    for (int i = l; i <= r; i++) a[i] += c;
    Sort(x);
    return;
  }
  for (int i = l; i <= ed[x]; i++) a[i] += c;     // 直接修改起始段
  for (int i = st[y]; i <= r; i++) a[i] += c;     // 直接修改结束段
  for (int i = x + 1; i < y; i++) delta[i] += c;  // 中间的块整体打上标记
  Sort(x);
  Sort(y);
}

int Answer(int l, int r, int c) {
  int ans = 0, x = belong[l], y = belong[r];
  if (x == y) {
    for (int i = l; i <= r; i++)
      if (a[i] + delta[x] >= c) ans++;
    return ans;
  }
  for (int i = l; i <= ed[x]; i++)
    if (a[i] + delta[x] >= c) ans++;
  for (int i = st[y]; i <= r; i++)
    if (a[i] + delta[y] >= c) ans++;
  for (int i = x + 1; i <= y - 1; i++)
    ans +=
        ed[i] - (lower_bound(t + st[i], t + ed[i] + 1, c - delta[i]) - t) + 1;
  // 用 lower_bound 找出中间每一个整块中第一个大于等于 c 的数的位置
  return ans;
}
```

例题 2：寒夜方舟

两种操作：

1. 区间 $[x,y]$ 每个数都变成 z；
2. 查询区间 $[x,y]$ 内小于等于 z 的数的个数。

用 `delta` 数组记录现在块内被整体赋值为何值。当该块未被整体赋值时，用一个特殊值（如 `0x3f3f3f3f3f3f3f3fll`）加以表示。对于边角块，查询前要 `pushdown`，把块内存的信息下放到每一个数上。赋值之后记得重新 `sort` 一遍。其他方面同上题。

"实现"

```
void Sort(int k) {
  for (int i = st[k]; i <= ed[k]; i++) t[i] = a[i];
  sort(t + st[k], t + ed[k] + 1);
}

void PushDown(int x) {
  if (delta[x] != 0x3f3f3f3f3f3f3f3fll)  // 用该值标记块内没有被整体赋值
    for (int i = st[x]; i <= ed[x]; i++) a[i] = t[i] = delta[x];
  delta[x] = 0x3f3f3f3f3f3f3f3fll;
}

void Modify(int l, int r, int c) {
  int x = belong[l], y = belong[r];
  PushDown(x);
  if (x == y) {
    for (int i = l; i <= r; i++) a[i] = c;
    Sort(x);
    return;
  }
  PushDown(y);
  for (int i = l; i <= ed[x]; i++) a[i] = c;
  for (int i = st[y]; i <= r; i++) a[i] = c;
  Sort(x);
  Sort(y);
  for (int i = x + 1; i < y; i++) delta[i] = c;
}

int Binary_Search(int l, int r, int c) {
  int ans = l - 1, mid;
  while (l <= r) {
    mid = (l + r) / 2;
    if (t[mid] <= c)
      ans = mid, l = mid + 1;
    else
      r = mid - 1;
  }
  return ans;
}

int Answer(int l, int r, int c) {
  int ans = 0, x = belong[l], y = belong[r];
  PushDown(x);
  if (x == y) {
    for (int i = l; i <= r; i++)
      if (a[i] <= c) ans++;
    return ans;
  }
  PushDown(y);
  for (int i = l; i <= ed[x]; i++)
    if (a[i] <= c) ans++;
  for (int i = st[y]; i <= r; i++)
    if (a[i] <= c) ans++;
```

```
  for (int i = x + 1; i <= y - 1; i++) {
    if (0x3f3f3f3f3f3f3f3fll == delta[i])
      ans += Binary_Search(st[i], ed[i], c) - st[i] + 1;
    else if (delta[i] <= c)
      ans += size[i];
  }
  return ans;
}
```

练习

1. 单点修改， 区间查询[2]
2. 区间修改， 区间查询[3]
3. 【模板】 线段树 2[4]
4. 「Ynoi2019 模拟赛」Yuno loves sqrt technology III[5]
5. 「Violet」 蒲公英[6]
6. 作诗[7]

参考资料与注释

[1] 教主的魔法

[2] 单点修改，区间查询

[3] 区间修改，区间查询

[4] 【模板】线段树 2

[5] 「Ynoi2019 模拟赛」Yuno loves sqrt technology III

[6] 「Violet」蒲公英

[7] 作诗

10.8.3 块状链表

Authors: HeRaNO, konnyakuxzy, littlefrog

块状链表大概就长这样……

不难发现块状链表就是一个链表，每个节点指向一个数组。我们把原来长度为 n 的数组分为 $\sqrt{n}$ 个节点，每个节点对应的数组大小为 $\sqrt{n}$。所以我们这么定义结构体，代码见下。其中 `sqn` 表示 `sqrt(n)` 即 $\sqrt{n}$，`pb` 表示 `push_back`，即在这个 `node` 中加入一个元素。

" 实现"

```
struct node {
  node* nxt;
  int size;
```

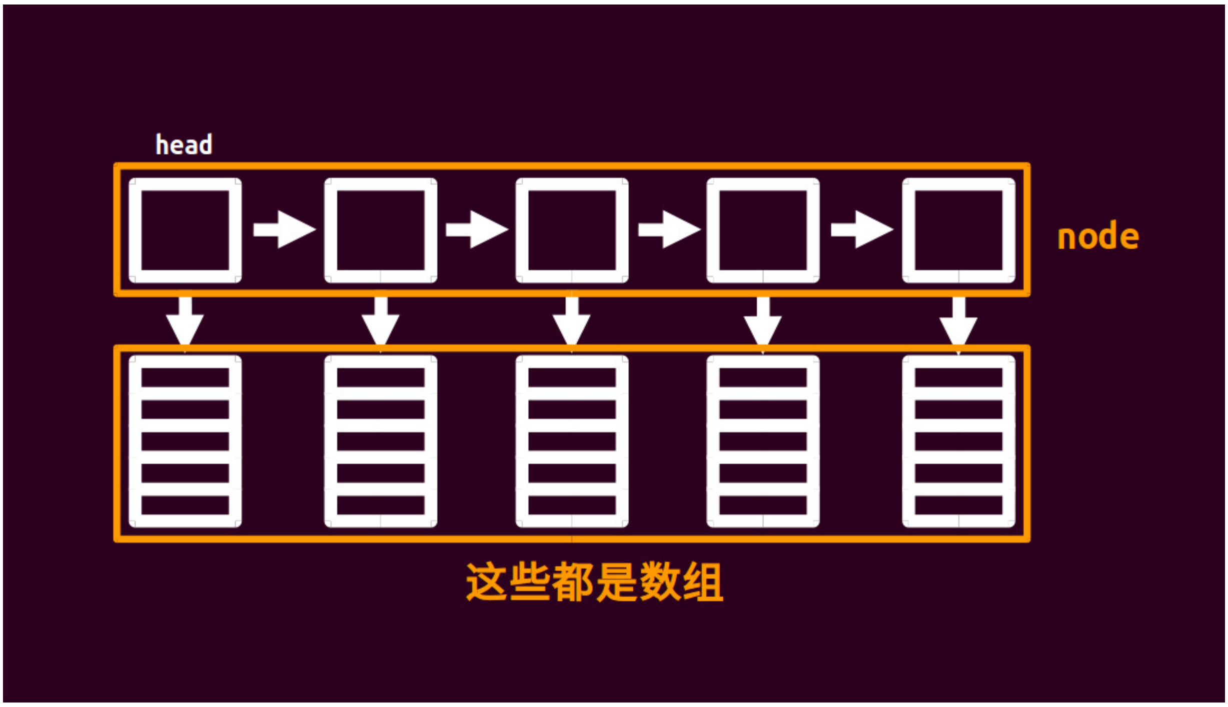


图 10.27 ./images/kuaizhuanglianbiao.png

```
  char d[(sqn << 1) + 5];

  node() { size = 0, nxt = NULL, memset(d, 0, sizeof(d)); }

  void pb(char c) { d[size++] = c; }
};
```

块状链表应该至少支持：分裂、插入、查找。什么是分裂？分裂就是分裂一个 `node`，变成两个小的 `node`，以保证每个 `node` 的大小都接近 $\sqrt{n}$（否则可能退化成普通数组）。当一个 `node` 的大小超过 $2 \times \sqrt{n}$ 时执行分裂操作。

分裂操作怎么做呢？先新建一个节点，再把被分裂的节点的后 $\sqrt{n}$ 个值 `copy` 到新节点，然后把被分裂的节点的后 $\sqrt{n}$ 个值删掉（`size--`），最后把新节点插入到被分裂节点的后面即可。

块状链表的所有操作的复杂度都是 $\sqrt{n}$ 的。

还有一个要说的。随着元素的插入（或删除），n 会变，$\sqrt{n}$ 也会变。这样块的大小就会变化，我们难道还要每次维护块的大小？

其实不然，把 $\sqrt{n}$ 设置为一个定值即可。比如题目给的范围是 10^6，那么 $\sqrt{n}$ 就设置为大小为 10^3 的常量，不用更改它。

```
list<vector<char> > orz_list;
```

STL 中的 rope

导入

STL 中的 `rope` 也起到块状链表的作用，它采用可持久化平衡树实现，可完成随机访问和插入、删除元素的操作。

由于 `rope` 并不是真正的用块状链表来实现，所以它的时间复杂度并不等同于块状链表，而是相当于可持久化平衡树的复杂度（即 $O(\log n)$）。

可以使用如下方法来引入：

```
#include <ext/rope>
using namespace __gnu_cxx;
```

" 关于双下划线开头的库函数"

OI 中，关于能否使用双下划线开头的库函数曾经一直不确定，2021 年 CCF 发布的 关于 NOI 系列活动中编程语言使用限制的补充说明[1] 中提到「允许使用以下划线开头的库函数或宏，但具有明确禁止操作的库函数和宏除外」。故 `rope` 目前可以在 OI 中正常使用。

基本操作

操作	作用
`rope <int > a`	初始化 `rope`（与 `vector` 等容器很相似）
`a.push_back(x)`	在 `a` 的末尾添加元素 `x`
`a.insert(pos, x)`	在 `a` 的 `pos` 个位置添加元素 `x`
`a.erase(pos, x)`	在 `a` 的 `pos` 个位置删除 `x` 个元素
`a.at(x)` 或 `a[x]`	访问 `a` 的第 `x` 个元素
`a.length()` 或 `a.size()`	获取 `a` 的大小

例题

POJ2887 Big String[2]

题解：很简单的模板题。代码如下：

```
#include <cctype>
#include <cstdio>
#include <cstring>
using namespace std;
static const int sqn = 1e3;

struct node {  // 定义块状链表
  node* nxt;
  int size;
  char d[(sqn << 1) + 5];

  node() { size = 0, nxt = NULL; }

  void pb(char c) { d[size++] = c; }
}* head = NULL;

char inits[(int)1e6 + 5];
int llen, q;

void readch(char& ch) {  // 读入字符
  do ch = getchar();
  while (!isalpha(ch));
}

void check(node* p) {  // 判断，记得要分裂
```

```
  if (p->size >= (sqn << 1)) {
    node* q = new node;
    for (int i = sqn; i < p->size; i++) q->pb(p->d[i]);
    p->size = sqn, q->nxt = p->nxt, p->nxt = q;
  }
}

void insert(char c, int pos) {  // 元素插入，借助链表来理解
  node* p = head;
  int tot, cnt;
  if (pos > llen++) {
    while (p->nxt != NULL) p = p->nxt;
    p->pb(c), check(p);
    return;
  }
  for (tot = head->size; p != NULL && tot < pos; p = p->nxt, tot += p->size)
    ;
  tot -= p->size, cnt = pos - tot - 1;
  for (int i = p->size - 1; i >= cnt; i--) p->d[i + 1] = p->d[i];
  p->d[cnt] = c, p->size++;
  check(p);
}

char query(int pos) {  // 查询
  node* p;
  int tot;
  for (p = head, tot = head->size; p != NULL && tot < pos;
       p = p->nxt, tot += p->size)
    ;
  tot -= p->size;
  return p->d[pos - tot - 1];
}

int main() {
  scanf("%s %d", inits, &q), llen = strlen(inits);
  node* p = new node;
  head = p;
  for (int i = 0; i < llen; i++) {
    if (i % sqn == 0 && i) p->nxt = new node, p = p->nxt;
    p->pb(inits[i]);
  }
  char a;
  int k;
  while (q--) {
    readch(a);
    if (a == 'Q')
      scanf("%d", &k), printf("%c\n", query(k));
    else
      readch(a), scanf("%d", &k), insert(a, k);
  }
  return 0;
}
```

参考资料与注释

[1] 关于 NOI 系列活动中编程语言使用限制的补充说明

[2] POJ2887 Big String

10.8.4 树分块

Authors: ouuan, Ir1d, Marcythm, Xeonacid

树分块的方式

可以参考 真 - 树上莫队。

也可以参考 ouuan 的博客 / 莫队 、 带修莫队 、 树上莫队详解 / 树上莫队[1]。

树上莫队同样可以参考以上两篇文章。

树分块的应用

树分块除了应用于莫队，还可以灵活地运用到某些树上问题中。但可以用树分块解决的题目往往都有更优秀的做法，所以相关的题目较少。

顺带提一句，「gty 的妹子树」的树分块做法可以被菊花图卡掉。

BZOJ4763 雪辉[2]

先进行树分块，然后对每个块的关键点，预处理出它到祖先中每个关键点的路径上颜色的 bitset，以及每个关键点的最近关键点祖先，复杂度是 $O(n\sqrt{n}+\frac{nc}{32})$，其中 $n\sqrt{n}$ 是暴力从每个关键点向上跳的复杂度，$\frac{nc}{32}$ 是把 $O(n)$ 个 `bitset` 存下来的复杂度。

回答询问的时候，先从路径的端点暴力跳到所在块的关键点，再从所在块的关键点一块一块地向上跳，直到 lca 所在块，然后再暴力跳到 lca。关键点之间的 `bitset` 已经预处理了，剩下的在暴力跳的过程中计算。单次询问复杂度是 $O(\sqrt{n}+\frac{c}{32})$，其中 $\sqrt{n}$ 是块内暴力跳以及块直接向上跳的复杂度，$O(\frac{c}{32})$ 是将预处理的结果与暴力跳的结果合并的复杂度。数颜色个数可以用 `bitset` 的 `count()`，求 mex 可以用 `bitset` 的 `_Find_first()`。

所以，总复杂度为 $O((n+m)(\sqrt{n}+\frac{c}{32}))$。

" 参考代码"

```
#include <algorithm>
#include <bitset>
#include <cctype>
#include <cstdio>
#include <iostream>

using namespace std;

int read() {
  int out = 0;
  char c;
  while (!isdigit(c = getchar()))
    ;
  for (; isdigit(c); c = getchar()) out = out * 10 + c - '0';
  return out;
```

```
}

const int N = 100010;
const int B = 666;
const int C = 30000;

void add(int u, int v);
void dfs(int u);

int head[N], nxt[N << 1], to[N << 1], cnt;
int n, m, type, c[N], fa[N], dep[N], sta[N], top, tot, bl[N], key[N / B + 5],
    p[N], keyid[N];
bool vis[N];
bitset<C> bs[N / B + 5][N / B + 5], temp;

int main() {
  int i, u, v, x, y, k, lastans = 0;

  n = read();
  m = read();
  type = read();

  for (i = 1; i <= n; ++i) c[i] = read();

  for (i = 1; i < n; ++i) {
    u = read();
    v = read();
    add(u, v);
    add(v, u);
  }

  dfs(1);

  if (!tot) ++tot;
  if (keyid[key[tot]] == tot) keyid[key[tot]] = 0;
  key[tot] = 1;
  keyid[1] = tot;
  while (top) bl[sta[top--]] = tot;

  for (i = 1; i <= tot; ++i) {  // 预处理
    if (vis[key[i]]) continue;
    vis[key[i]] = true;
    temp.reset();
    for (u = key[i]; u; u = fa[u]) {
      temp[c[u]] = 1;
      if (keyid[u]) {
        if (!p[key[i]] && u != key[i]) p[key[i]] = u;
        bs[keyid[key[i]]][keyid[u]] = temp;
      }
    }
  }

  while (m--) {
    k = read();
```

```
    temp.reset();
    while (k--) {
      u = x = read() ^ lastans;
      v = y = read() ^ lastans;

      while (key[bl[x]] != key[bl[y]]) {
        if (dep[key[bl[x]]] > dep[key[bl[y]]]) {
          if (x == u) {  // 若是第一次跳先暴力跳到关键点
            while (x != key[bl[u]]) {
              temp[c[x]] = 1;
              x = fa[x];
            }
          } else
            x = p[x];  // 否则跳一整块
        } else {
          if (y == v) {
            while (y != key[bl[v]]) {
              temp[c[y]] = 1;
              y = fa[y];
            }
          } else
            y = p[y];
        }
      }

      if (keyid[x]) temp |= bs[keyid[key[bl[u]]]][keyid[x]];
      if (keyid[y]) temp |= bs[keyid[key[bl[v]]]][keyid[y]];

      while (x != y) {
        if (dep[x] > dep[y]) {
          temp[c[x]] = 1;
          x = fa[x];
        } else {
          temp[c[y]] = 1;
          y = fa[y];
        }
      }
      temp[c[x]] = true;
    }
    int ans1 = temp.count(), ans2 = (~temp)._Find_first();
    printf("%d %d\n", ans1, ans2);
    lastans = (ans1 + ans2) * type;
  }

  return 0;
}

void dfs(int u) {  // 根据题意找点
  int i, v, t = top;
  for (i = head[u]; i; i = nxt[i]) {
    v = to[i];
    if (v == fa[u]) continue;
    fa[v] = u;
    dep[v] = dep[u] + 1;
```

```
      dfs(v);
      if (top - t >= B) {
        key[++tot] = u;
        if (!keyid[u]) keyid[u] = tot;
        while (top > t) bl[sta[top--]] = tot;
      }
    }
    sta[++top] = u;
  }

  void add(int u, int v) {
    nxt[++cnt] = head[u];
    head[u] = cnt;
    to[cnt] = v;
  }
```

BZOJ4812 由乃打扑克[3]

这题和上一题基本一样，唯一的区别是得到 `bitset` 后如何计算答案。

~~由于 BZOJ 是计算所有测试点总时限，不好卡，所以可以用 \hytt{_Find_next()} 水过去。~~

正解是每 16 位一起算，先预处理出 2^{16} 种可能的情况高位连续 1 的个数、低位连续 1 的个数以及中间的贡献。只不过这样要手写 `bitset`，因为标准库的 `bitset` 不能取某 16 位……

代码可以参考 这篇博客[4]。

参考资料与注释

[1] ouuan 的博客/莫队、带修莫队、树上莫队详解/树上莫队

[2] BZOJ4763 雪辉

[3] BZOJ4812 由乃打扑克

[4] 这篇博客

10.8.5 Sqrt Tree

引入

给你一个长度为 n 的序列 $\langle a_i\rangle_{i=1}^n$，再给你一个满足结合律的运算 $\circ$（比如 $\gcd, \min, \max, +, \text{and}, \text{or}, \text{xor}$ 均满足结合律），然后对于每一次区间询问 $[l, r]$，我们需要计算 $a_l \circ a_{l+1} \circ \cdots \circ a_r$。

Sqrt Tree 可以在 $O(n \log\log n)$ 的时间内预处理，并在 $O(1)$ 的时间内回答询问。

解释

序列分块

首先我们把整个序列分成 $O(\sqrt{n})$ 个块，每一块的大小为 $O(\sqrt{n})$。对于每个块，我们计算：

1. P_i 块内的前缀区间询问
2. S_i 块内的后缀区间询问
3. 维护一个额外的数组 $\langle B_{i,j}\rangle$ 表示第 i 个块到第 j 个块的区间答案。

举个例子，假设 $\circ$ 代表加法运算 $+$，序列为 $\{1,2,3,4,5,6,7,8,9\}$。

首先我们将序列分成三块，变成了 $\{1,2,3\},\{4,5,6\},\{7,8,9\}$。

那么每一块的前缀区间答案和后缀区间答案分别为

$$P_1=\{1,3,6\},S_1=\{6,5,3\}$$
$$P_2=\{4,9,15\},S_2=\{15,11,6\}$$
$$P_3=\{7,15,24\},S_3=\{24,17,9\}$$

B 数组为：

$$B=\begin{bmatrix}6 & 21 & 45\\ 0 & 15 & 39\\ 0 & 0 & 24\end{bmatrix}$$

（对于 $i>j$ 的不合法的情况我们假设答案为 0）

显然我们可以在 $O(n)$ 的时间内预处理这些值，空间复杂度同样是 $O(n)$ 的。处理好之后，我们可以利用它们在 $O(1)$ 的时间内回答一些跨块的询问。但对于那些整个区间都在一个块内的询问我们仍不能处理，因此我们还需要处理一些东西。

构建一棵树

容易想到我们在每个块内递归地构造上述结构以支持块内的查询。对于大小为 1 的块我们可以 $O(1)$ 地回答询问。这样我们就建出了一棵树，每一个结点代表序列的一个区间。叶子结点的区间长度为 1 或 2。一个大小为 k 的结点有 $O(\sqrt{k})$ 个子节点，于是整棵树的高度是 $O(\log\log n)$ 的，每一层的区间总长是 $O(n)$ 的，因此我们构建这棵树的复杂度是 $O(n\log\log n)$ 的。

现在我们可以在 $O(\log\log n)$ 的时间内回答询问。对于询问 $[l,r]$，我们只需要快速找到一个区间长度最小的结点 u 使得 u 能包含 $[l,r]$，这样 $[l,r]$ 在 u 的分块区间中一定是跨块的，就可以 $O(1)$ 地计算答案了。查询一次的总体复杂度是 $O(\log\log n)$，因为树高是 $O(\log\log n)$ 的。不过我们仍可以优化这个过程。

优化询问复杂度

容易想到二分高度，然后可以 $O(1)$ 判断是否合法。这样复杂度就变成了 $O(\log\log\log n)$。不过我们仍可以进一步加速这一过程。

我们假设

1. 每一块的大小都是 2 的整数幂次；
2. 每一层上的块大小是相同的。

为此我们需要在序列的末位补充一些 0 元素，使得它的长度变成 2 的整数次幂。尽管有些块可能会变成原来的两倍大小，但这样仍是 $O(\sqrt{k})$ 的，于是预处理分块的复杂度仍是 $O(n)$ 的。

现在我们可以轻松地确定一个询问区间是否被整个地包含在一个块中。对于区间 $[l,r]$（以 0 为起点），我们把端点写为二进制形式。举一个例子，对于 $k=4,l=39,r=46$，二进制表示为

$$l=39_{10}=100111_2,r=46_{10}=101110_2$$

我们知道每一层的区间长度是相同的，而分块的大小也是相同的（在上述示例中 $2^k=2^4=16$）。这些块完全覆盖了整个序列，因此第一块代表的元素为 $[0,15]$（二进制表示为 $[000000_2,001111_2]$），第二个块代表的元素区间为 $[16,31]$（二进制表示为 $[010000_2,011111_2]$），以此类推。我们发现这些在同一个块内的元素的位置在二进制上只有后 k 位不同（上述示例中 $k=4$）。而示例的 l,r 也只有后 k 位不同，因此他们在同一个块中。

因此我们需要检查区间两个端点是否只有后 k 位不同，即 $l\oplus r\le 2^k-1$。因此我们可以快速找到答案区间所在的层：

1. 对于每个 $i\in[1,n]$，我们找到 i 最高位上的 1；
2. 现在对于一个询问 $[l,r]$，我们计算 $l\oplus r$ 的最高位，这样就可以快速确定答案区间所在的层。

这样我们就可以在 $O(1)$ 的时间内回答询问啦。

更新元素的过程

我们可以在 Sqrt Tree 上更新元素，单点修改和区间修改都是支持的。

单点修改

考虑一次单点赋值操作 $a_x = val$，我们希望高效更新这个操作的信息。

朴素实现

首先我们来看看在做了一次单点修改后 Sqrt Tree 会变成什么样子。

考虑一个长度为 l 的结点以及对应的序列：$\langle P_i\rangle, \langle S_i\rangle, \langle B_{i,j}\rangle$。容易发现在 $\langle P_i\rangle$ 和 $\langle S_i\rangle$ 中都只有 $O(\sqrt{l})$ 个元素改变。而在 $\langle B_{i,j}\rangle$ 中则有 $O(l)$ 个元素被改变。因此有 $O(l)$ 个元素在树上被更新。因此在 Sqrt Tree 上单点修改的复杂度是 $O(n+\sqrt{n}+\sqrt{\sqrt{n}}+\cdots)=O(n)$。

使用 Sqrt Tree 替代 B 数组

注意到单点更新的瓶颈在于更新根结点的 $\langle B_{i,j}\rangle$。因此我们尝试用另一个 Sqrt Tree 代替根结点的 $\langle B_{i,j}\rangle$，称其为 $index$。它的作用和原来的二维数组一样，维护整段询问的答案。其他非根结点仍然使用 $\langle B_{i,j}\rangle$ 维护。注意，如果一个 Sqrt Tree 根结点有 $index$ 结构，称其 Sqrt Tree 是**含有索引**的；如果一个 Sqrt Tree 的根结点有 $\langle B_{i,j}\rangle$ 结构，称其是**没有索引**的。而 $index$ 这棵树本身是没有索引的。

因此我们可以这样更新 $index$ 树：

1. 在 $O(\sqrt{n})$ 的时间内更新 $\langle P_i\rangle$ 和 $\langle S_i\rangle$。
2. 更新 $index$，它的长度是 $O(n)$ 的，但我们只需要更新其中的一个元素（这个元素代表了被改变的块），这一步的时间复杂度是 $O(\sqrt{n})$ 的（使用朴素实现的算法）。
3. 进入产生变化的子节点并使用朴素实现的算法在 $O(\sqrt{n})$ 的时间内更新信息。

注意，查询的复杂度仍是 $O(1)$ 的，因为我们最多使用 $index$ 树一次。于是单点修改的复杂度就是 $O(\sqrt{n})$ 的。

更新一个区间

Sqrt Tree 也支持区间覆盖操作 $\mathrm{Update}(l,r,x)$，即把区间 $[l,r]$ 的数全部变成 x。对此我们有两种实现方式，其中一种会花费 $O(\sqrt{n}\log\log n)$ 的复杂度更新信息，$O(1)$ 的时间查询；另一种则是 $O(\sqrt{n})$ 更新信息，但查询的时间会增加到 $O(\log\log n)$。

我们可以像线段树一样在 Sqrt Tree 上打懒标记。但是在 Sqrt Tree 上有一点不同。因为下传一个结点的懒标记，复杂度可能达到 $O(\sqrt{n})$，因此我们不是在询问的时侯下传标记，而是看父节点是否有标记，如果有标记就把它下传。

第一种实现

在第一种实现中，我们只会给第 1 层的结点（结点区间长度为 $O(\sqrt{n})$）打懒标记，在下传标记的时侯直接更新整个子树，复杂度为 $O(\sqrt{n}\log\log n)$。操作过程如下：

1. 考虑第 1 层上的结点，对于那些被修改区间完全包含的结点，给他们打一个懒标记；
2. 有两个块只有部分区间被覆盖，我们直接在 $O(\sqrt{n}\log\log n)$ 的时间内**重建**这两个块。如果它本身带有之前修改的懒标记，就在重建的时侯顺便下传标记；
3. 更新根结点的 $\langle P_i\rangle$ 和 $\langle S_i\rangle$，时间复杂度 $O(\sqrt{n})$；
4. 重建 $index$ 树，时间复杂度 $O(\sqrt{n}\log\log n)$。

现在我们可以高效完成区间修改了。那么如何利用懒标记回答询问？操作如下：

1. 如果我们的询问被包含在一个有懒标记的块内，可以利用懒标记计算答案；
2. 如果我们的询问包含多个块，那么我们只需要关心最左边和最右边不完整块的答案。中间的块的答案可以在 $index$ 树中查询（因为 $index$ 树在每次修改完后会重建），复杂度是 $O(1)$。

因此询问的复杂度仍为 $O(1)$。

第二种实现

在这种实现中，每一个结点都可以被打上懒标记。因此在处理一个询问的时侯，我们需要考虑祖先中的懒标记，那么查询的复杂度将变成 $O(\log\log n)$。不过更新信息的复杂度就会变得更快。操作如下：

1. 被修改区间完全包含的块，我们把懒标记添加到这些块上，复杂度 $O(\sqrt{n})$；
2. 被修改区间部分覆盖的块，更新 $\langle P_i\rangle$ 和 $\langle S_i\rangle$，复杂度 $O(\sqrt{n})$（因为只有两个被修改的块）；
3. 更新 $index$ 树，复杂度 $O(\sqrt{n})$（使用同样的更新算法）；
4. 对于没有索引的子树更新他们的 $\langle B_{i,j}\rangle$；
5. 递归地更新两个没有被完全覆盖的区间。

时间复杂度是 $O(\sqrt{n}+\sqrt{\sqrt{n}}+\cdots)=O(\sqrt{n})$。

实现

下面的实现在 $O(n\log\log n)$ 的时间内建树，在 $O(1)$ 的时间内回答询问，在 $O(\sqrt{n})$ 的时间内单点修改。

```
SqrtTreeItem op(const SqrtTreeItem &a, const SqrtTreeItem &b);

int log2Up(int n) {
  int res = 0;
  while ((1 << res) < n) {
    res++;
  }
  return res;
}

class SqrtTree {
 private:
  int n, lg, indexSz;
  vector<SqrtTreeItem> v;
  vector<int> clz, layers, onLayer;
  vector<vector<SqrtTreeItem> > pref, suf, between;

  void buildBlock(int layer, int l, int r) {
    pref[layer][l] = v[l];
    for (int i = l + 1; i < r; i++) {
      pref[layer][i] = op(pref[layer][i - 1], v[i]);
    }
    suf[layer][r - 1] = v[r - 1];
    for (int i = r - 2; i >= l; i--) {
      suf[layer][i] = op(v[i], suf[layer][i + 1]);
    }
  }

  void buildBetween(int layer, int lBound, int rBound, int betweenOffs) {
    int bSzLog = (layers[layer] + 1) >> 1;
    int bCntLog = layers[layer] >> 1;
    int bSz = 1 << bSzLog;
    int bCnt = (rBound - lBound + bSz - 1) >> bSzLog;
    for (int i = 0; i < bCnt; i++) {
      SqrtTreeItem ans;
      for (int j = i; j < bCnt; j++) {
        SqrtTreeItem add = suf[layer][lBound + (j << bSzLog)];
```

```
        ans = (i == j) ? add : op(ans, add);
        between[layer - 1][betweenOffs + lBound + (i << bCntLog) + j] = ans;
      }
    }
  }

  void buildBetweenZero() {
    int bSzLog = (lg + 1) >> 1;
    for (int i = 0; i < indexSz; i++) {
      v[n + i] = suf[0][i << bSzLog];
    }
    build(1, n, n + indexSz, (1 << lg) - n);
  }

  void updateBetweenZero(int bid) {
    int bSzLog = (lg + 1) >> 1;
    v[n + bid] = suf[0][bid << bSzLog];
    update(1, n, n + indexSz, (1 << lg) - n, n + bid);
  }

  void build(int layer, int lBound, int rBound, int betweenOffs) {
    if (layer >= (int)layers.size()) {
      return;
    }
    int bSz = 1 << ((layers[layer] + 1) >> 1);
    for (int l = lBound; l < rBound; l += bSz) {
      int r = min(l + bSz, rBound);
      buildBlock(layer, l, r);
      build(layer + 1, l, r, betweenOffs);
    }
    if (layer == 0) {
      buildBetweenZero();
    } else {
      buildBetween(layer, lBound, rBound, betweenOffs);
    }
  }

  void update(int layer, int lBound, int rBound, int betweenOffs, int x) {
    if (layer >= (int)layers.size()) {
      return;
    }
    int bSzLog = (layers[layer] + 1) >> 1;
    int bSz = 1 << bSzLog;
    int blockIdx = (x - lBound) >> bSzLog;
    int l = lBound + (blockIdx << bSzLog);
    int r = min(l + bSz, rBound);
    buildBlock(layer, l, r);
    if (layer == 0) {
      updateBetweenZero(blockIdx);
    } else {
      buildBetween(layer, lBound, rBound, betweenOffs);
    }
    update(layer + 1, l, r, betweenOffs, x);
  }
```

```
  SqrtTreeItem query(int l, int r, int betweenOffs, int base) {
    if (l == r) {
      return v[l];
    }
    if (l + 1 == r) {
      return op(v[l], v[r]);
    }
    int layer = onLayer[clz[(l - base) ^ (r - base)]];
    int bSzLog = (layers[layer] + 1) >> 1;
    int bCntLog = layers[layer] >> 1;
    int lBound = (((l - base) >> layers[layer]) << layers[layer]) + base;
    int lBlock = ((l - lBound) >> bSzLog) + 1;
    int rBlock = ((r - lBound) >> bSzLog) - 1;
    SqrtTreeItem ans = suf[layer][l];
    if (lBlock <= rBlock) {
      SqrtTreeItem add =
          (layer == 0) ? (query(n + lBlock, n + rBlock, (1 << lg) - n, n))
                       : (between[layer - 1][betweenOffs + lBound +
                                             (lBlock << bCntLog) + rBlock]);
      ans = op(ans, add);
    }
    ans = op(ans, pref[layer][r]);
    return ans;
  }

 public:
  SqrtTreeItem query(int l, int r) { return query(l, r, 0, 0); }

  void update(int x, const SqrtTreeItem &item) {
    v[x] = item;
    update(0, 0, n, 0, x);
  }

  SqrtTree(const vector<SqrtTreeItem> &a)
      : n((int)a.size()), lg(log2Up(n)), v(a), clz(1 << lg), onLayer(lg + 1) {
    clz[0] = 0;
    for (int i = 1; i < (int)clz.size(); i++) {
      clz[i] = clz[i >> 1] + 1;
    }
    int tlg = lg;
    while (tlg > 1) {
      onLayer[tlg] = (int)layers.size();
      layers.push_back(tlg);
      tlg = (tlg + 1) >> 1;
    }
    for (int i = lg - 1; i >= 0; i--) {
      onLayer[i] = max(onLayer[i], onLayer[i + 1]);
    }
    int betweenLayers = max(0, (int)layers.size() - 1);
    int bSzLog = (lg + 1) >> 1;
    int bSz = 1 << bSzLog;
    indexSz = (n + bSz - 1) >> bSzLog;
    v.resize(n + indexSz);
```

```
    pref.assign(layers.size(), vector<SqrtTreeItem>(n + indexSz));
    suf.assign(layers.size(), vector<SqrtTreeItem>(n + indexSz));
    between.assign(betweenLayers, vector<SqrtTreeItem>((1 << lg) + bSz));
    build(0, 0, n, 0);
  }
};
```

习题

CodeChef - SEGPROD[1]

参考资料与注释

[1] CodeChef - SEGPROD

[2] Sqrt Tree - Algorithms for Competitive Programming

10.9 单调栈

引入

何为单调栈？顾名思义，单调栈即满足单调性的栈结构。与单调队列相比，其只在一端进行进出。

为了描述方便，以下举例及伪代码以维护一个整数的单调递增栈为例。

过程

插入

将一个元素插入单调栈时，为了维护栈的单调性，需要在保证将该元素插入到栈顶后整个栈满足单调性的前提下弹出最少的元素。

例如，栈中自顶向下的元素为 $\{0, 11, 45, 81\}$。

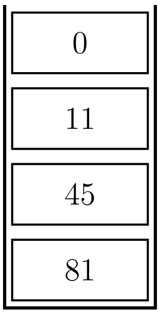


图 10.28

插入元素 14 时为了保证单调性需要依次弹出元素 $0, 11$，操作后栈变为 $\{14, 45, 81\}$。

用伪代码描述如下：

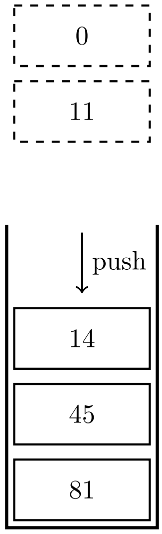


图 10.29

" 实现"

```
insert x
while !sta.empty() && sta.top()<x
    sta.pop()
sta.push(x)
```

使用

自然就是从栈顶读出来一个元素，该元素满足单调性的某一端。

例如举例中取出的即栈中的最小值。

应用

"POJ3250 Bad Hair Day[1]"

有 N 头牛从左到右排成一排，每头牛有一个高度 h_i，设左数第 i 头牛与「它右边第一头高度 $\geq h_i$」的牛之间有 c_i 头牛，试求 $\sum_{i=1}^{N} c_i$。

比较基础的应用有这一题，就是个单调栈的简单应用，记录每头牛被弹出的位置，如果没有被弹出过则为最远端，稍微处理一下即可计算出题目所需结果。

另外，单调栈也可以用于离线解决 RMQ 问题。

我们可以把所有询问按右端点排序，然后每次在序列上从左往右扫描到当前询问的右端点处，并把扫描到的元素插入到单调栈中。这样，每次回答询问时，单调栈中存储的值都是位置 $\leq r$ 的、可能成为答案的决策点，并且这些元素满足单调性质。此时，单调栈上第一个位置 $\geq l$ 的元素就是当前询问的答案，这个过程可以用二分查找实现。使用单调栈解决 RMQ 问题的时间复杂度为 $O(q\log q + q\log n)$，空间复杂度为 $O(n)$。

习题

- 洛谷 P5788【模板】单调栈[2]
- 洛谷 P1901 发射站[3]

参考资料与注释

[1] POJ3250 Bad Hair Day

[2] 洛谷 P5788【模板】单调栈

[3] 洛谷 P1901 发射站

10.10 单调队列

Authors: Link-cute, Xeonacid, ouuan, Alphnia, Lyccrius

引入

在学习单调队列前，让我们先来看一道例题。

" 例题"

Sliding Window[1]

本题大意是给出一个长度为 n 的数组，编程输出每 k 个连续的数中的最大值和最小值。

最暴力的想法很简单，对于每一段 $i \sim i+k-1$ 的序列，逐个比较来找出最大值（和最小值），时间复杂度约为 $O(n \times k)$。

很显然，这其中进行了大量重复工作，除了开头 $k-1$ 个和结尾 $k-1$ 个数之外，每个数都进行了 k 次比较，而题中 100% 的数据为 $n \leq 1000000$，当 k 稍大的情况下，显然会 TLE。

这时所用到的就是单调队列了。

定义

顾名思义，单调队列的重点分为「单调」和「队列」。

「单调」指的是元素的「规律」——递增（或递减）。

「队列」指的是元素只能从队头和队尾进行操作。

Ps. 单调队列中的" 队列" 与正常的队列有一定的区别，稍后会提到

例题分析

解释

有了上面「单调队列」的概念，很容易想到用单调队列进行优化。

要求的是每连续的 k 个数中的最大（最小）值，很明显，当一个数进入所要" 寻找" 最大值的范围中时，若这个数比其前面（先进队）的数要大，显然，前面的数会比这个数先出队且不再可能是最大值。

也就是说——当满足以上条件时，可将前面的数" 弹出"，再将该数真正 push 进队尾。

这就相当于维护了一个递减的队列，符合单调队列的定义，减少了重复的比较次数，不仅如此，由于维护出的队伍是查询范围内的且是递减的，队头必定是该查询区域内的最大值，因此输出时只需输出队头即可。

显而易见的是，在这样的算法中，每个数只要进队与出队各一次，因此时间复杂度被降到了 $O(N)$。

而由于查询区间长度是固定的，超出查询空间的值再大也不能输出，因此还需要 site 数组记录第 i 个队中的数在原数组中的位置，以弹出越界的队头。

过程

例如我们构造一个单调递增的队列会如下：

原序列为：

```
1 3 -1 -3 5 3 6 7
```

因为我们始终要维护队列保证其**递增**的特点，所以会有如下的事情发生：

操作	队列状态
1 入队	{1}
3 比 1 大，3 入队	{1 3}
-1 比队列中所有元素小，所以清空队列 -1 入队	{-1}
-3 比队列中所有元素小，所以清空队列 -3 入队	{-3}
5 比 -3 大，直接入队	{-3 5}
3 比 5 小，5 出队，3 入队	{-3 3}
-3 已经在窗体外，所以 -3 出队；6 比 3 大，6 入队	{3 6}
7 比 6 大，7 入队	{3 6 7}

" 例题参考代码"

```cpp
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <iostream>
#define maxn 1000100
using namespace std;
int q[maxn], a[maxn];
int n, k;

void getmin() {  // 得到这个队列里的最小值，直接找到最后的就行了
  int head = 0, tail = -1;
  for (int i = 1; i < k; i++) {
    while (head <= tail && a[q[tail]] >= a[i]) tail--;
    q[++tail] = i;
  }
  for (int i = k; i <= n; i++) {
    while (head <= tail && a[q[tail]] >= a[i]) tail--;
    q[++tail] = i;
    while (q[head] <= i - k) head++;
    printf("%d ", a[q[head]]);
  }
}

void getmax() {  // 和上面同理
  int head = 0, tail = -1;
  for (int i = 1; i < k; i++) {
    while (head <= tail && a[q[tail]] <= a[i]) tail--;
```

```
    q[++tail] = i;
  }
  for (int i = k; i <= n; i++) {
    while (head <= tail && a[q[tail]] <= a[i]) tail--;
    q[++tail] = i;
    while (q[head] <= i - k) head++;
    printf("%d ", a[q[head]]);
  }
}

int main() {
  scanf("%d%d", &n, &k);
  for (int i = 1; i <= n; i++) scanf("%d", &a[i]);
  getmin();
  printf("\n");
  getmax();
  printf("\n");
  return 0;
}
```

Ps. 此处的" 队列" 跟普通队列的一大不同就在于可以从队尾进行操作，STL 中有类似的数据结构 deque。

" 例题 2 Luogu P2698 Flowerpot S[2]"

给出 N 滴水的坐标，y 表示水滴的高度，x 表示它下落到 x 轴的位置。每滴水以每秒 1 个单位长度的速度下落。你需要把花盆放在 x 轴上的某个位置，使得从被花盆接着的第 1 滴水开始，到被花盆接着的最后 1 滴水结束，之间的时间差至少为 D。我们认为，只要水滴落到 x 轴上，与花盆的边沿对齐，就认为被接住。给出 N 滴水的坐标和 D 的大小，请算出最小的花盆的宽度 W。$1 \le N \le 100000, 1 \le D \le 1000000, 0 \le x, y \le 10^6$

将所有水滴按照 x 坐标排序之后，题意可以转化为求一个 x 坐标差最小的区间使得这个区间内 y 坐标的最大值和最小值之差至少为 D。我们发现这道题和上一道例题有相似之处，就是都与一个区间内的最大值最小值有关，但是这道题区间的大小不确定，而且区间大小本身还是我们要求的答案。

我们依然可以使用一个递增，一个递减两个单调队列在 R 不断后移时维护 $[L, R]$ 内的最大值和最小值，不过此时我们发现，如果 L 固定，那么 $[L, R]$ 内的最大值只会越来越大，最小值只会越来越小，所以设 $f(R) = \max[L, R] - \min[L, R]$，则 $f(R)$ 是个关于 R 的递增函数，故 $f(R) \ge D \implies f(r) \ge D, R < r \le N$。这说明对于每个固定的 L，向右第一个满足条件的 R 就是最优答案。所以我们整体求解的过程就是，先固定 L，从前往后移动 R，使用两个单调队列维护 $[L, R]$ 的最值。当找到了第一个满足条件的 R，就更新答案并将 L 也向后移动。随着 L 向后移动，两个单调队列都需及时弹出队头。这样，直到 R 移到最后，每个元素依然是各进出队列一次，保证了 $O(n)$ 的时间复杂度。

" 参考代码"

```
#include <bits/stdc++.h>
using namespace std;
const int N = 100005;
typedef long long ll;
int mxq[N], mnq[N];
int D, ans, n, hx, rx, hn, rn;

struct la {
  int x, y;

  bool operator<(const la &y) const { return x < y.x; }
```

```
} a[N];

int main() {
  scanf("%d%d", &n, &D);
  for (int i = 1; i <= n; ++i) {
    scanf("%d%d", &a[i].x, &a[i].y);
  }
  sort(a + 1, a + n + 1);
  hx = hn = 1;
  ans = 2e9;
  int L = 1;
  for (int i = 1; i <= n; ++i) {
    while (hx <= rx && a[mxq[rx]].y < a[i].y) rx--;
    mxq[++rx] = i;
    while (hn <= rn && a[mnq[rn]].y > a[i].y) rn--;
    mnq[++rn] = i;
    while (L <= i && a[mxq[hx]].y - a[mnq[hn]].y >= D) {
      ans = min(ans, a[i].x - a[L].x);
      L++;
      while (hx <= rx && mxq[hx] < L) hx++;
      while (hn <= rn && mnq[hn] < L) hn++;
    }
  }
  if (ans < 2e9)
    printf("%d\n", ans);
  else
    puts("-1");
  return 0;
}
```

参考资料与注释

[1] Sliding Window

[2] Luogu P2698 Flowerpot S

10.11　ST 表

定义

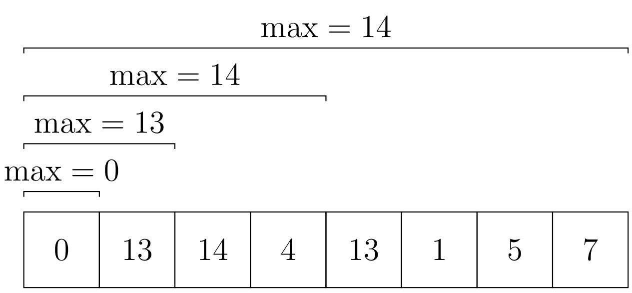


图 10.30 ST 表示意图

ST 表是用于解决**可重复贡献问题**的数据结构。

" 什么是可重复贡献问题？"

可重复贡献问题是指对于运算 opt，满足 $x \operatorname{opt} x = x$，则对应的区间询问就是一个可重复贡献问题。例如，最大值有 $\max(x, x) = x$，gcd 有 $\gcd(x, x) = x$，所以 RMQ 和区间 GCD 就是一个可重复贡献问题。像区间和就不具有这个性质，如果求区间和的时候采用的预处理区间重叠了，则会导致重叠部分被计算两次，这是我们所不愿意看到的。另外，opt 还必须满足结合律才能使用 ST 表求解。

" 什么是 RMQ？"

RMQ 是英文 Range Maximum/Minimum Query 的缩写，表示区间最大（最小）值。解决 RMQ 问题有很多种方法，可以参考 RMQ 专题。

引入

ST 表模板题[1]

题目大意：给定 n 个数，有 m 个询问，对于每个询问，你需要回答区间 $[l, r]$ 中的最大值。

考虑暴力做法。每次都对区间 $[l, r]$ 扫描一遍，求出最大值。

显然，这个算法会超时。

ST 表

ST 表基于 倍增 思想，可以做到 $\Theta(n \log n)$ 预处理，$\Theta(1)$ 回答每个询问。但是不支持修改操作。

基于倍增思想，我们考虑如何求出区间最大值。可以发现，如果按照一般的倍增流程，每次跳 2^i 步的话，询问时的复杂度仍旧是 $\Theta(\log n)$，并没有比线段树更优，反而预处理一步还比线段树慢。

我们发现 $\max(x, x) = x$，也就是说，区间最大值是一个具有「可重复贡献」性质的问题。即使用来求解的预处理区间有重叠部分，只要这些区间的并是所求的区间，最终计算出的答案就是正确的。

如果手动模拟一下，可以发现我们能使用至多两个预处理过的区间来覆盖询问区间，也就是说询问时的时间复杂度可以被降至 $\Theta(1)$，在处理有大量询问的题目时十分有效。

具体实现如下：

令 $f(i, j)$ 表示区间 $[i, i + 2^j - 1]$ 的最大值。

显然 $f(i, 0) = a_i$。

根据定义式，第二维就相当于倍增的时候「跳了 $2^j - 1$ 步」，依据倍增的思路，写出状态转移方程：$f(i, j) = \max(f(i, j-1), f(i + 2^{j-1}, j-1))$。

$$\max\left(\boxed{\max(\{a_i, \ldots, a_{i+2^{j-1}-1}\})} \boxed{\max(\{a_{i+2^{j-1}}, \ldots, a_{i+2^j-1}\})} \right)$$

$$\|$$

$$\boxed{\max(\{a_i, \ldots, a_{i+2^j-1}\})}$$

图 10.31

以上就是预处理部分。而对于查询，可以简单实现如下：

对于每个询问 $[l, r]$，我们把它分成两部分：$[l, l + 2^s - 1]$ 与 $[r - 2^s + 1, r]$，其中 $s = \lfloor \log_2(r - l + 1) \rfloor$。两部分的结果的最大值就是回答。

根据上面对于「可重复贡献问题」的论证，由于最大值是「可重复贡献问题」，重叠并不会对区间最大值产生影响。又因为这两个区间完全覆盖了 $[l, r]$，可以保证答案的正确性。

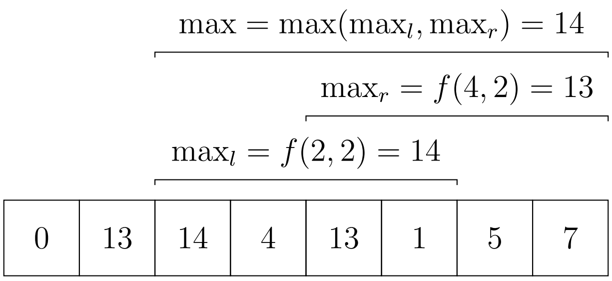


图 10.32 ST 表的查询过程

模板代码

ST 表模板题[1]

C 风格模板

```
#include <bits/stdc++.h>
using namespace std;
const int logn = 21;
const int maxn = 2000001;
int f[maxn][logn + 1], Logn[maxn + 1];

int read() {  // 快读
  char c = getchar();
  int x = 0, f = 1;
  while (c < '0' || c > '9') {
    if (c == '-') f = -1;
    c = getchar();
  }
  while (c >= '0' && c <= '9') {
    x = x * 10 + c - '0';
    c = getchar();
  }
  return x * f;
}

void pre() {  // 准备工作，初始化
  Logn[1] = 0;
  Logn[2] = 1;
  for (int i = 3; i < maxn; i++) {
    Logn[i] = Logn[i / 2] + 1;
  }
}

int main() {
  int n = read(), m = read();
  for (int i = 1; i <= n; i++) f[i][0] = read();
  pre();
  for (int j = 1; j <= logn; j++)
    for (int i = 1; i + (1 << j) - 1 <= n; i++)
      f[i][j] = max(f[i][j - 1], f[i + (1 << (j - 1))][j - 1]);  // ST 表具体实现
  for (int i = 1; i <= m; i++) {
```

```
    int x = read(), y = read();
    int s = Logn[y - x + 1];
    printf("%d\n", max(f[x][s], f[y - (1 << s) + 1][s]));
  }
  return 0;
}
```

C++ 风格模板

```
#include <bits/stdc++.h>
using namespace std;

template <typename T>
class SparseTable {
  using VT = vector<T>;
  using VVT = vector<VT>;
  using func_type = function<T(const T &, const T &)>;

  VVT ST;

  static T default_func(const T &t1, const T &t2) { return max(t1, t2); }

  func_type op;

 public:
  SparseTable(const vector<T> &v, func_type _func = default_func) {
    op = _func;
    int len = v.size(), l1 = ceil(log2(len)) + 1;
    ST.assign(len, VT(l1, 0));
    for (int i = 0; i < len; ++i) {
      ST[i][0] = v[i];
    }
    for (int j = 1; j < l1; ++j) {
      int pj = (1 << (j - 1));
      for (int i = 0; i + pj < len; ++i) {
        ST[i][j] = op(ST[i][j - 1], ST[i + (1 << (j - 1))][j - 1]);
      }
    }
  }

  T query(int l, int r) {
    int lt = r - l + 1;
    int q = floor(log2(lt));
    return op(ST[l][q], ST[r - (1 << q) + 1][q]);
  }
};
```

注意点

1. 输入输出数据一般很多，建议开启输入输出优化。
2. 每次用 std::log[2] 重新计算 log 函数值并不值得，建议进行如下的预处理：

$$\begin{cases} \text{Logn}[1] \leftarrow 0, \\ \text{Logn}\left[i\right] \leftarrow \text{Logn}\left[\frac{i}{2}\right] + 1. \end{cases}$$

ST 表维护其他信息

除 RMQ 以外，还有其它的「可重复贡献问题」。例如「区间按位和」、「区间按位或」、「区间 GCD」，ST 表都能高效地解决。

需要注意的是，对于「区间 GCD」，ST 表的查询复杂度并没有比线段树更优（令值域为 w，ST 表的查询复杂度为 $\Theta(\log w)$，而线段树为 $\Theta(\log n+\log w)$，且值域一般是大于 n 的），但是 ST 表的预处理复杂度也没有比线段树更劣，而编程复杂度方面 ST 表比线段树简单很多。

如果分析一下，「可重复贡献问题」一般都带有某种类似 RMQ 的成分。例如「区间按位与」就是每一位取最小值，而「区间 GCD」则是每一个质因数的指数取最小值。

总结

ST 表能较好的维护「可重复贡献」的区间信息（同时也应满足结合律），时间复杂度较低，代码量相对其他算法很小。但是，ST 表能维护的信息非常有限，不能较好地扩展，并且不支持修改操作。

练习

RMQ 模板题[1]

「SCOI2007」 降雨量[3]

[USACO07JAN] 平衡的阵容 Balanced Lineup[4]

附录：ST 表求区间 GCD 的时间复杂度分析

在算法运行的时候，可能要经过 $\Theta(\log n)$ 次迭代。每一次迭代都可能会使用 GCD 函数进行递归，令值域为 w，GCD 函数的时间复杂度最高是 $\Omega(\log w)$ 的，所以总时间复杂度看似有 $O(n\log n\log w)$。

但是，在 GCD 的过程中，每一次递归（除最后一次递归之外）都会使数列中的某个数至少减半，而数列中的数最多减半的次数为 $\log_2(w^n)=\Theta(n\log w)$，所以，GCD 的递归部分最多只会运行 $O(n\log w)$ 次。再加上循环部分（以及最后一层递归）的 $\Theta(n\log n)$，最终时间复杂度则是 $O(n(\log w+\log x))$，由于可以构造数据使得时间复杂度为 $\Omega(n(\log w+\log x))$，所以最终的时间复杂度即为 $\Theta(n(\log w+\log x))$。

而查询部分的时间复杂度很好分析，考虑最劣情况，即每次询问都询问最劣的一对数，时间复杂度为 $\Theta(\log w)$。因此，ST 表维护「区间 GCD」的时间复杂度为预处理 $\Theta(n(\log n+\log w))$，单次查询 $\Theta(\log w)$。

线段树的相应操作是预处理 $\Theta(n\log x)$，查询 $\Theta(n(\log n+\log x))$。

这并不是一个严谨的数学论证，更为严谨的附在下方：

" 更严谨的证明"

理解本段，可能需要具备 时间复杂度 的关于「势能分析法」的知识。

先分析预处理部分的时间复杂度：

设「待考虑数列」为在预处理 ST 表的时候当前层循环的数列。例如，第零层的数列就是原数列，第一层的数列就是第零层的数列经过一次迭代之后的数列，即 `st[1..n][1]`，我们将其记为 A。

而势能函数就定义为「待考虑数列」中所有数的累乘的以二为底的对数。即：$\Phi(A)=\log_2\left(\prod\limits_{i=1}^{n}A_i\right)$。

在一次迭代中，所花费的时间相当于迭代循环所花费的时间与 GCD 所花费的时间之和。其中，GCD 花费的时间有长有短。最短可能只有两次甚至一次递归，而最长可能有 $O(\log w)$ 次递归。但是，GCD 过程中，除最开头一层与最末一层以外，每次递归都会使「待考虑数列」中的某个结果至少减半。即，$\Phi(A)$ 会减少至少 1，该

层递归所用的时间可以被势能函数均摊。

同时，我们可以看到，$\Phi(A)$ 的初值最大为 $\log_2(w^n)=\Theta(n\log w)$，而 $\Phi(A)$ 不增。所以，ST 表预处理部分的时间复杂度为 $O(n(\log w+\log n))$。

参考资料与注释

[1] ST 表模板题 [1-1] [1-2] [1-3]

[2] std::log

[3] 「SCOI2007」降雨量

[4] [USACO07JAN] 平衡的阵容 Balanced Lineup

10.12 树状数组

Authors: HeRaNO, Zhoier, Ir1d, Xeonacid, wangdehu, ouuan, ranwen, ananbaobeichicun, Ycrpro, dbxxx-oi

引入

树状数组是一种支持**单点修改**和**区间查询**的，代码量小的数据结构。

" 什么是「单点修改」和「区间查询」？"

假设有这样一道题：

已知一个数列 a，你需要进行下面两种操作：

- 给定 x,y，将 $a[x]$ 自增 y。
- 给定 l,r，求解 $a[l\dots r]$ 的和。

其中第一种操作就是「单点修改」，第二种操作就是「区间查询」。

类似地，还有：「区间修改」、「单点查询」。它们分别的一个例子如下：

- 区间修改：给定 l,r,x，将 $a[l\dots r]$ 中的每个数都分别自增 x；
- 单点查询：给定 x，求解 $a[x]$ 的值。

注意到，区间问题一般严格强于单点问题，因为对单点的操作相当于对一个长度为 1 的区间操作。

普通树状数组维护的信息及运算要满足**结合律**且**可差分**，如加法（和）、乘法（积）、异或等。

- 结合律：$(x\circ y)\circ z=x\circ(y\circ z)$，其中 $\circ$ 是一个二元运算符。
- 可差分：具有逆运算的运算，即已知 $x\circ y$ 和 x 可以求出 y。

需要注意的是：

- 模意义下的乘法若要可差分，需保证每个数都存在逆元（模数为质数时一定存在）；
- 例如 gcd，max 这些信息不可差分，所以不能用普通树状数组处理，但是：
  - 使用两个树状数组可以用于处理区间最值，见 Efficient Range Minimum Queries using Binary Indexed Trees[1]。
  - 本页面也会介绍一种支持不可差分信息查询的，$\Theta(\log^2 n)$ 时间复杂度的拓展树状数组。

事实上，树状数组能解决的问题是线段树能解决的问题的子集：树状数组能做的，线段树一定能做；线段树能做的，树状数组不一定可以。然而，树状数组的代码要远比线段树短，时间效率常数也更小，因此仍有学习价值。

有时，在差分数组和辅助数组的帮助下，树状数组还可解决更强的**区间加单点值**和**区间加区间和**问题。

树状数组

初步感受

先来举个例子：我们想知道 $a[1\ldots 7]$ 的前缀和，怎么做？

一种做法是：$a_1+a_2+a_3+a_4+a_5+a_6+a_7$，需要求 7 个数的和。

但是如果已知三个数 A，B，C，$A=a[1\ldots 4]$ 的和，$B=a[5\ldots 6]$ 的总和，$C=a[7\ldots 7]$ 的总和（其实就是 $a[7]$ 自己）。你会怎么算？你一定会回答：$A+B+C$，只需要求 3 个数的和。

这就是树状数组能快速求解信息的原因：我们总能将一段前缀 $[1,n]$ 拆成**不多于 $\log n$ 段区间**，使得这 $\log n$ 段区间的信息是**已知的**。

于是，我们只需合并这 $\log n$ 段区间的信息，就可以得到答案。相比于原来直接合并 n 个信息，效率有了很大的提高。

不难发现信息必须满足结合律，否则就不能像上面这样合并了。

下面这张图展示了树状数组的工作原理：

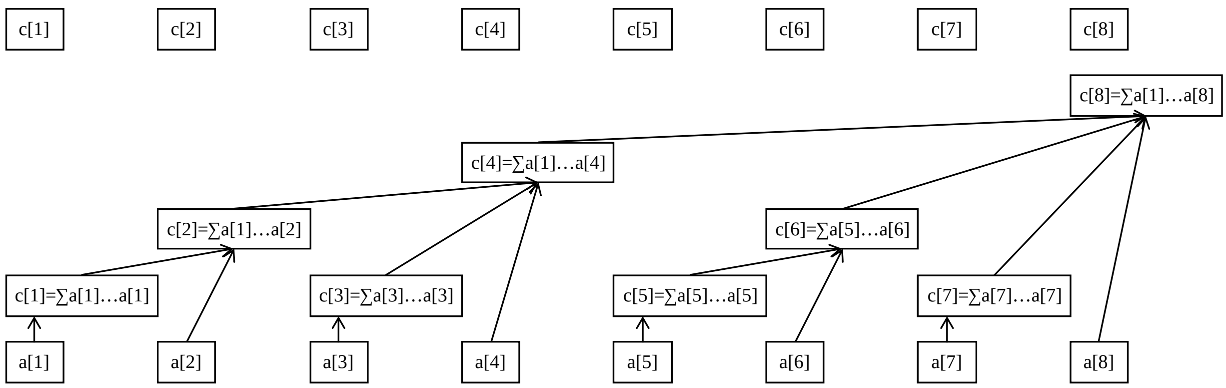


图 10.33

最下面的八个方块代表原始数据数组 a。上面参差不齐的方块（与最上面的八个方块是同一个数组）代表数组 a 的上级——c 数组。

c 数组就是用来储存原始数组 a 某段区间的和的，也就是说，这些区间的信息是已知的，我们的目标就是把查询前缀拆成这些小区间。

例如，从图中可以看出：

- c_2 管辖的是 $a[1\ldots 2]$；
- c_4 管辖的是 $a[1\ldots 4]$；
- c_6 管辖的是 $a[5\ldots 6]$；
- c_8 管辖的是 $a[1\ldots 8]$；
- 剩下的 $c[x]$ 管辖的都是 $a[x]$ 自己（可以看做 $a[x\ldots x]$ 的长度为 1 的小区间）。

不难发现，$c[x]$ 管辖的一定是一段右边界是 x 的区间总信息。我们先不关心左边界，先来感受一下树状数组是如何查询的。

举例：计算 $a[1\ldots 7]$ 的和。

过程：从 c_7 开始往前跳，发现 c_7 只管辖 a_7 这个元素；然后找 c_6，发现 c_6 管辖的是 $a[5\ldots 6]$，然后跳到 c_4，发现 c_4 管辖的是 $a[1\ldots 4]$ 这些元素，然后再试图跳到 c_0，但事实上 c_0 不存在，不跳了。

我们刚刚找到的 c 是 c_7, c_6, c_4，事实上这就是 $a[1\dots7]$ 拆分出的三个小区间，合并得到答案是 $c_7+c_6+c_4$。

举例：计算 $a[4\dots7]$ 的和。

我们还是从 c_7 开始跳，跳到 c_6 再跳到 c_4。此时我们发现它管理了 $a[1\dots4]$ 的和，但是我们不想要 $a[1\dots3]$ 这一部分，怎么办呢？很简单，减去 $a[1\dots3]$ 的和就行了。

那不妨考虑最开始，就将查询 $a[4\dots7]$ 的和转化为查询 $a[1\dots7]$ 的和，以及查询 $a[1\dots3]$ 的和，最终将两个结果作差。

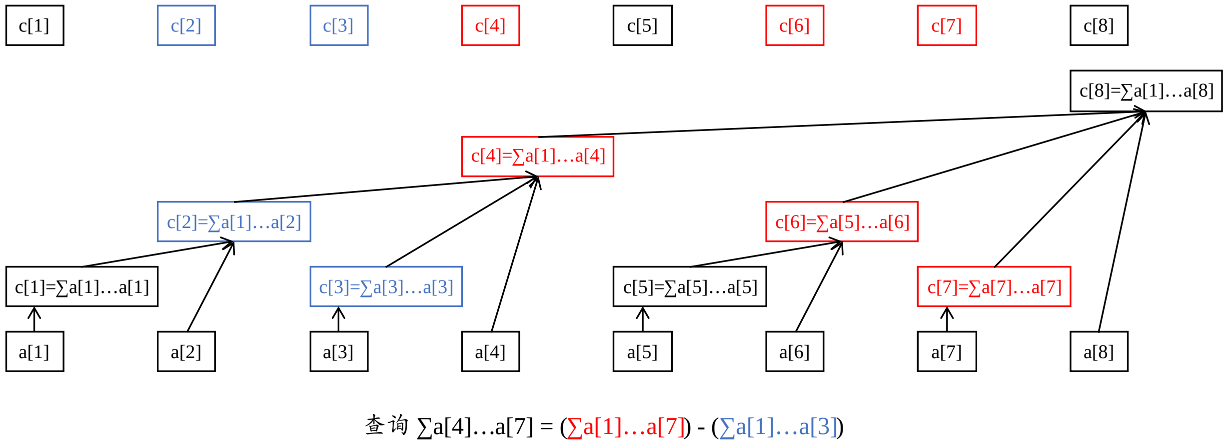


图 10.34

管辖区间

那么问题来了，$c[x](x \ge 1)$ 管辖的区间到底往左延伸多少？也就是说，区间长度是多少？

树状数组中，规定 $c[x]$ 管辖的区间长度为 2^k，其中：

- 设二进制最低位为第 0 位，则 k 恰好为 x 二进制表示中，最低位的 `1` 所在的二进制位数；
- 2^k（$c[x]$ 的管辖区间长度）恰好为 x 二进制表示中，最低位的 `1` 以及后面所有 `0` 组成的数。

举个例子，c_{88} 管辖的是哪个区间？

因为 $88_{(10)} = 01011000_{(2)}$，其二进制最低位的 `1` 以及后面的 `0` 组成的二进制是 `1000`，即 8，所以 c_{88} 管辖 8 个 a 数组中的元素。

因此，c_{88} 代表 $a[81\dots88]$ 的区间信息。

我们记 x 二进制最低位 `1` 以及后面的 `0` 组成的数为 $\operatorname{lowbit}(x)$，那么 $c[x]$ 管辖的区间就是 $[x-\operatorname{lowbit}(x)+1, x]$。

这里注意：lowbit 指的不是最低位 `1` 所在的位数 k，而是这个 `1` 和后面所有 `0` 组成的 2^k。

怎么计算 `lowbit`？根据位运算知识，可以得到 `lowbit(x) = x & -x`。

"lowbit 的原理"

将 `x` 的二进制所有位全部取反，再加 1，就可以得到 `-x` 的二进制编码。例如，6 的二进制编码是 `110`，全部取反后得到 `001`，加 `1` 得到 `010`。

设原先 `x` 的二进制编码是 `(...)10...00`，全部取反后得到 `[...]01...11`，加 `1` 后得到 `[...]10...00`，也就是 `-x` 的二进制编码了。这里 `x` 二进制表示中第一个 `1` 是 `x` 最低位的 `1`。

`(...)` 和 `[...]` 中省略号的每一位分别相反，所以 `x & -x = (...)10...00 & [...]10...00 = 10...00`，得到的结果就是 `lowbit`。

" 实现"

```
int lowbit(int x) {
  // x 的二进制中，最低位的 1 以及后面所有 0 组成的数。
  // lowbit(0b01011000) == 0b00001000
  //          ~~~~^~~~
  // lowbit(0b01110010) == 0b00000010
  //          ~~~~~~^~
  return x & -x;
}
```

```
def lowbit(x):
    """
    x 的二进制中，最低位的 1 以及后面所有 0 组成的数。
    lowbit(0b01011000) == 0b00001000
            ~~~~~^~~
    lowbit(0b01110010) == 0b00000010
            ~~~~~~~^~
    """
    return x & -x
```

区间查询

接下来我们来看树状数组具体的操作实现，先来看区间查询。

回顾查询 $a[4\ldots 7]$ 的过程，我们是将它转化为两个子过程：查询 $a[1\ldots 7]$ 和查询 $a[1\ldots 3]$ 的和，最终作差。

其实任何一个区间查询都可以这么做：查询 $a[l\ldots r]$ 的和，就是 $a[1\ldots r]$ 的和减去 $a[1\ldots l-1]$ 的和，从而把区间问题转化为前缀问题，更方便处理。

事实上，将有关 $l\ldots r$ 的区间询问转化为 $1\ldots r$ 和 $1\ldots l-1$ 的前缀询问再差分，在竞赛中是一个非常常用的技巧。

那前缀查询怎么做呢？回顾下查询 $a[1\ldots 7]$ 的过程：

> 从 c_7 往前跳，发现 c_7 只管辖 a_7 这个元素；然后找 c_6，发现 c_6 管辖的是 $a[5\ldots 6]$，然后跳到 c_4，发现 c_4 管辖的是 $a[1\ldots 4]$ 这些元素，然后再试图跳到 c_0，但事实上 c_0 不存在，不跳了。
>
> 我们刚刚找到的 c 是 c_7, c_6, c_4，事实上这就是 $a[1\ldots 7]$ 拆分出的三个小区间，合并一下，答案是 $c_7+c_6+c_4$。

观察上面的过程，每次往前跳，一定是跳到现区间的左端点的左一位，作为新区间的右端点，这样才能将前缀不重不漏地拆分。比如现在 c_6 管的是 $a[5\ldots 6]$，下一次就跳到 $5-1=4$，即访问 c_4。

我们可以写出查询 $a[1\ldots x]$ 的过程：

- 从 $c[x]$ 开始往前跳，有 $c[x]$ 管辖 $a[x-\operatorname{lowbit}(x)+1\ldots x]$；
- 令 $x\gets x-\operatorname{lowbit}(x)$，如果 $x=0$ 说明已经跳到尽头了，终止循环；否则回到第一步。
- 将跳到的 c 合并。

实现时，我们不一定要先把 c 都跳出来然后一起合并，可以边跳边合并。

比如我们要维护的信息是和，直接令初始 $\mathrm{ans}=0$，然后每跳到一个 $c[x]$ 就 $\mathrm{ans}\gets \mathrm{ans}+c[x]$，最终 ans 就是所有合并的结果。

" 实现"

```
int getsum(int x) {  // a[1]..a[x] 的和
  int ans = 0;
  while (x > 0) {
    ans = ans + c[x];
```

```
    x = x - lowbit(x);
  }
  return ans;
}

def getsum(x): # a[1]..a[x] 的和
    ans = 0
    while x > 0:
        ans = ans + c[x]
        x = x - lowbit(x)
    return ans
```

树状数组与其树形态的性质

在讲解单点修改之前，先讲解树状数组的一些基本性质，以及其树形态来源，这有助于更好理解树状数组的单点修改。

我们约定：

- $l(x)=x-\operatorname{lowbit}(x)+1$。即，$l(x)$ 是 $c[x]$ 管辖范围的左端点。
- 对于任意正整数 x，总能将 x 表示成 $s\times 2^{k+1}+2^k$ 的形式，其中 $\operatorname{lowbit}(x)=2^k$。
- 下面「$c[x]$ 和 $c[y]$ 不交」指 $c[x]$ 的管辖范围和 $c[y]$ 的管辖范围不相交，即 $[l(x),x]$ 和 $[l(y),y]$ 不相交。「$c[x]$ 包含于 $c[y]$」等表述同理。

性质 1：对于 $x\le y$，要么有 $c[x]$ 和 $c[y]$ 不交，要么有 $c[x]$ 包含于 $c[y]$。

" 证明"

证明：假设 $c[x]$ 和 $c[y]$ 相交，即 $[l(x),x]$ 和 $[l(y),y]$ 相交，则一定有 $l(y)\le x\le y$。

将 y 表示为 $s\times 2^{k+1}+2^k$，则 $l(y)=s\times 2^{k+1}+1$。所以，x 可以表示为 $s\times 2^{k+1}+b$，其中 $1\le b\le 2^k$。

不难发现 $\operatorname{lowbit}(x)=\operatorname{lowbit}(b)$。又因为 $b-\operatorname{lowbit}(b)\ge 0$，

所以 $l(x)=x-\operatorname{lowbit}(x)+1=s\times 2^{k+1}+b-\operatorname{lowbit}(b)+1\ge s\times 2^{k+1}+1=l(y)$，即 $l(y)\le l(x)\le x\le y$。

所以，如果 $c[x]$ 和 $c[y]$ 相交，那么 $c[x]$ 的管辖范围一定完全包含于 $c[y]$。

性质 2：在 $c[x]$ 真包含于 $c[x+\operatorname{lowbit}(x)]$。

" 证明"

证明：设 $y=x+\operatorname{lowbit}(x)$，$x=s\times 2^{k+1}+2^k$，则 $y=(s+1)\times 2^{k+1}$，$l(x)=s\times 2^{k+1}+1$。

不难发现 $\operatorname{lowbit}(y)\ge 2^{k+1}$，所以 $l(y)=(s+1)\times 2^{k+1}-\operatorname{lowbit}(y)+1\le s\times 2^{k+1}+1=l(x)$，即 $l(y)\le l(x)\le x<y$。

所以，$c[x]$ 真包含于 $c[x+\operatorname{lowbit}(x)]$。

性质 3：对于任意 $x<y<x+\operatorname{lowbit}(x)$，有 $c[x]$ 和 $c[y]$ 不交。

" 证明"

证明：设 $x=s\times 2^{k+1}+2^k$，则 $y=x+b=s\times 2^{k+1}+2^k+b$，其中 $1\le b<2^k$。

不难发现 $\operatorname{lowbit}(y)=\operatorname{lowbit}(b)$。又因为 $b-\operatorname{lowbit}(b)\ge 0$，

因此 $l(y)=y-\operatorname{lowbit}(y)+1=x+b-\operatorname{lowbit}(b)+1>x$，即 $l(x)\le x<l(y)\le y$。

所以，$c[x]$ 和 $c[y]$ 不交。

有了这三条性质的铺垫，我们接下来看树状数组的树形态（请忽略 a 向 c 的连边）。

事实上，树状数组的树形态是 x 向 $x+\operatorname{lowbit}(x)$ 连边得到的图，其中 $x+\operatorname{lowbit}(x)$ 是 x 的父亲。

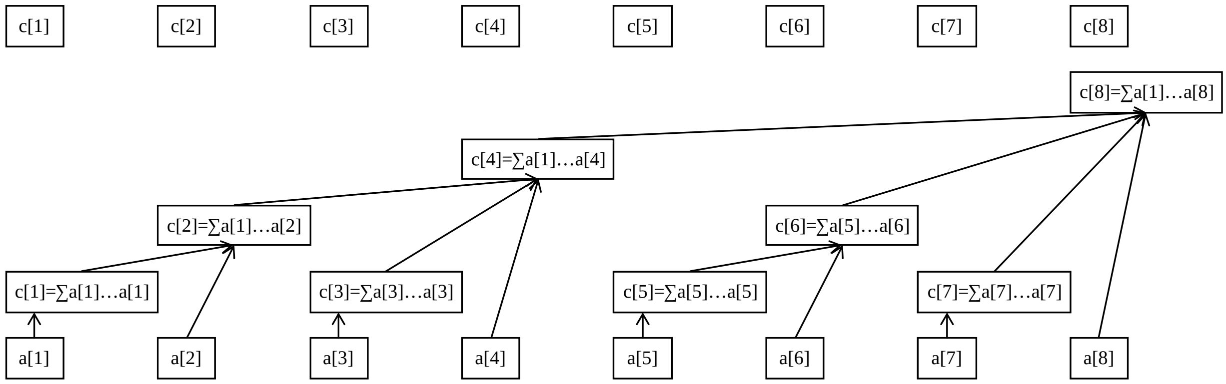


图 10.35

注意，在考虑树状数组的树形态时，我们不考虑树状数组大小的影响，即我们认为这是一棵无限大的树，方便分析。实际实现时，我们只需用到 $x \le n$ 的 $c[x]$，其中 n 是原数组长度。

这棵树天然满足了很多美好性质，下面列举若干（设 $fa[u]$ 表示 u 的直系父亲）：

- $u < fa[u]$。
- u 大于任何一个 u 的后代，小于任何一个 u 的祖先。
- 点 u 的 lowbit 严格小于 $fa[u]$ 的 lowbit。

" 证明"

设 $y = x + \text{lowbit}(x)$，$x = s \times 2^{k+1} + 2^k$，则 $y = (s+1) \times 2^{k+1}$，不难发现 $\text{lowbit}(y) \ge 2^{k+1} > \text{lowbit}(x)$，证毕。

- 点 x 的高度是 $\log_2 \text{lowbit}(x)$，即 x 二进制最低位 `1` 的位数。

" 高度的定义"

点 x 的高度 $h(x)$ 满足：如果 $x \bmod 2 = 1$，则 $h(x) = 0$，否则 $h(x) = \max(h(y)) + 1$，其中 y 代表 x 的所有儿子（此时 x 至少存在一个儿子 $x - 1$）。

也就是说，一个点的高度恰好比它最高的那个儿子再高 1。如果一个点没有儿子，它的高度是 0。

这里引出高度这一概念，是为后面解释复杂度更方便。

- $c[u]$ 真包含于 $c[fa[u]]$（性质 2）。
- $c[u]$ 真包含于 $c[v]$，其中 v 是 u 的任一祖先（在上一条性质上归纳）。
- $c[u]$ 真包含 $c[v]$，其中 v 是 u 的任一后代（上面那条性质 u，v 颠倒）。
- 对于任意 $v' > u$，若 v' 不是 u 的祖先，则 $c[u]$ 和 $c[v']$ 不交。

" 证明"

u 和 u 的祖先中，一定存在一个点 v 使得 $v < v' < fa[v]$，根据性质 3 得 $c[v']$ 不相交于 $c[v]$，而 $c[v]$ 包含 $c[u]$，因此 $c[v']$ 不交于 $c[u]$。

- 对于任意 $v < u$，如果 v 不在 u 的子树上，则 $c[u]$ 和 $c[v]$ 不交（上面那条性质 u，v' 颠倒）。
- 对于任意 $v > u$，当且仅当 v 是 u 的祖先，$c[u]$ 真包含于 $c[v]$（上面几条性质的总结）。这就是树状数组单点修改的核心原理。
- 设 $u = s \times 2^{k+1} + 2^k$，则其儿子数量为 $k = \log_2 \text{lowbit}(u)$，编号分别为 $u - 2^t (0 \le t < k)$。
  - 举例：假设 $k = 3$，u 的二进制编号为 `...1000`，则 u 有三个儿子，二进制编号分别为 `...0111`、`...0110`、`...0100`。

" 证明"

在一个数 x 的基础上减去 2^t，x 二进制第 t 位会反转，而更低的位保持不变。

考虑 u 的儿子 v，有 $v+\text{lowbit}(v)=u$，即 $v=u-2^t$ 且 $\text{lowbit}(v)=2^t$。设 $u=s\times 2^{k+1}+2^k$。

考虑 $0\le t<k$，u 的第 t 位及后方均为 0，所以 $v=u-2^t$ 的第 t 位变为 1，后面仍为 0，**满足** $\text{lowbit}(v)=2^t$。

考虑 $t=k$，则 $v=u-2^k$，v 的第 k 位变为 0，**不满足** $\text{lowbit}(v)=2^t$。

考虑 $t>k$，则 $v=u-2^t$，v 的第 k 位是 1，所以 $\text{lowbit}(v)=2^k$，**不满足** $\text{lowbit}(v)=2^t$。

- u 的所有儿子对应 c 的管辖区间恰好拼接成 $[l(u),u-1]$。
  - 举例：假设 $k=3$，u 的二进制编号为 `...1000`，则 u 有三个儿子，二进制编号分别为 `...0111`、`...0110`、`...0100`。
  - `c[...0100]` 表示 `a[...0001 ~ ...0100]`。
  - `c[...0110]` 表示 `a[...0101 ~ ...0110]`。
  - `c[...0111]` 表示 `a[...0111 ~ ...0111]`。
  - 不难发现上面是三个管辖区间的并集恰好是 `a[...0001 ~ ...0111]`，即 $[l(u),u-1]$。

" 证明"

u 的儿子总能表示成 $u-2^t(0\le t<k)$，不难发现，t 越小，$u-2^t$ 越大，代表的区间越靠右。我们设 $f(t)=u-2^t$，则 $f(k-1),f(k-2),\dots,f(0)$ 分别构成 u 从左到右的儿子。

不难发现 $\text{lowbit}(f(t))=2^t$，所以 $l(f(t))=u-2^t-2^t+1=u-2^{t+1}+1$。

考虑相邻的两个儿子 $f(t+1)$ 和 $f(t)$。前者管辖区间的右端点是 $f(t+1)=u-2^{t+1}$，后者管辖区间的左端点是 $l(f(t))=u-2^{t+1}+1$，恰好相接。

考虑最左面的儿子 $f(k-1)$，其管辖左边界 $l(f(k-1))=u-2^k+1$ 恰为 $l(u)$。

考虑最右面的儿子 $f(0)$，其管辖右边界就是 $u-1$。

因此，这些儿子的管辖区间可以恰好拼成 $[l(u),u-1]$。

单点修改

现在来考虑如何单点修改 $a[x]$。

我们的目标是快速正确地维护 c 数组。为保证效率，我们只需遍历并修改管辖了 $a[x]$ 的所有 $c[y]$，因为其他的 c 显然没有发生变化。

管辖 $a[x]$ 的 $c[y]$ 一定包含 $c[x]$（根据性质 1），所以 y 在树状数组树形态上是 x 的祖先。因此我们从 x 开始不断跳父亲，直到跳得超过了原数组长度为止。

设 n 表示 a 的大小，不难写出单点修改 $a[x]$ 的过程：

- 初始令 $x'=x$。
- 修改 $c[x']$。
- 令 $x'\leftarrow x'+\text{lowbit}(x')$，如果 $x'>n$ 说明已经跳到尽头了，终止循环；否则回到第二步。

区间信息和单点修改的种类，共同决定 $c[x']$ 的修改方式。下面给几个例子：

- 若 $c[x']$ 维护区间和，修改种类是将 $a[x]$ 加上 p，则修改方式则是将所有 $c[x']$ 也加上 p。
- 若 $c[x']$ 维护区间积，修改种类是将 $a[x]$ 乘上 p，则修改方式则是将所有 $c[x']$ 也乘上 p。

然而，单点修改的自由性使得修改的种类和维护的信息不一定是同种运算，比如，若 $c[x']$ 维护区间和，修改种类是将 $a[x]$ 赋值为 p，可以考虑转化为将 $a[x]$ 加上 $p-a[x]$。如果是将 $a[x]$ 乘上 p，就考虑转化为 $a[x]$ 加上 $a[x]\times p-a[x]$。

下面以维护区间和，单点加为例给出实现。

" 实现"

```cpp
void add(int x, int k) {
  while (x <= n) {  // 不能越界
    c[x] = c[x] + k;
    x = x + lowbit(x);
  }
}
```

```python
def add(x, k):
    while x <= n: # 不能越界
        c[x] = c[x] + k
        x = x + lowbit(x)
```

建树

也就是根据最开始给出的序列，将树状数组建出来（c 全部预处理好）。

一般可以直接转化为 n 次单点修改，时间复杂度 $\Theta(n\log n)$（复杂度分析在后面）。

比如给定序列 $a=(5,1,4)$ 要求建树，直接看作对 $a[1]$ 单点加 5，对 $a[2]$ 单点加 1，对 $a[3]$ 单点加 4 即可。

也有 $\Theta(n)$ 的建树方法，见本页面 $\Theta(n)$ 建树一节。

复杂度分析

空间复杂度显然 $\Theta(n)$。

时间复杂度：

- 对于区间查询操作：整个 $x \gets x-\operatorname{lowbit}(x)$ 的迭代过程，可看做将 x 二进制中的所有 1，从低位到高位逐渐改成 0 的过程，拆分出的区间数等于 x 二进制中 1 的数量（即 $\operatorname{popcount}(x)$）。因此，单次查询时间复杂度是 $\Theta(\log n)$；
- 对于单点修改操作：跳父亲时，访问到的高度一直严格增加，且始终有 $x \le n$。由于点 x 的高度是 $\log_2\operatorname{lowbit}(x)$，所以跳到的高度不会超过 $\log_2 n$，所以访问到的 c 的数量是 $\log n$ 级别。因此，单次单点修改复杂度是 $\Theta(\log n)$。

区间加区间和

前置知识：前缀和 & 差分。

该问题可以使用两个树状数组维护差分数组解决。

考虑序列 a 的差分数组 d，其中 $d[i]=a[i]-a[i-1]$。由于差分数组的前缀和就是原数组，所以 $a_i=\sum_{j=1}^i d_j$。

一样地，我们考虑将查询区间和通过差分转化为查询前缀和。那么考虑查询 $a[1\ldots r]$ 的和，即 $\sum_{i=1}^{r} a_i$，进行推导：

$$\begin{aligned}
&\sum_{i=1}^{r} a_i\\
=&\sum_{i=1}^{r}\sum_{j=1}^{i} d_j
\end{aligned}$$

观察这个式子，不难发现每个 d_j 总共被加了 $r-j+1$ 次。接着推导：

$$\begin{aligned}&\sum_{i=1}^{r}\sum_{j=1}^{i}d_j\\=&\sum_{i=1}^{r}d_i\times(r-i+1)\\=&\sum_{i=1}^{r}d_i\times(r+1)-\sum_{i=1}^{r}d_i\times i\end{aligned}$$

$\sum_{i=1}^{r}d_i$ 并不能推出 $\sum_{i=1}^{r}d_i\times i$ 的值，所以要用两个树状数组分别维护 d_i 和 $d_i\times i$ 的和信息。

那么怎么做区间加呢？考虑给原数组 $a[l\ldots r]$ 区间加 x 给 d 带来的影响。

因为差分是 $d[i]=a[i]-a[i-1]$，

- $a[l]$ 多了 v 而 $a[l-1]$ 不变，所以 $d[l]$ 的值多了 v。
- $a[r+1]$ 不变而 $a[r]$ 多了 v，所以 $d[r+1]$ 的值少了 v。
- 对于不等于 l 且不等于 $r+1$ 的任意 i，$a[i]$ 和 $a[i-1]$ 要么都没发生变化，要么都加了 v，$a[i]+v-(a[i-1]+v)$ 还是 $a[i]-a[i-1]$，所以其它的 $d[i]$ 均不变。

那就不难想到维护方式了：对于维护 d_i 的树状数组，对 l 单点加 v，$r+1$ 单点加 $-v$；对于维护 $d_i\times i$ 的树状数组，对 l 单点加 $v\times l$，$r+1$ 单点加 $-v\times(r+1)$。

而更弱的问题，「区间加求单点值」，只需用树状数组维护一个差分数组 d_i。询问 $a[x]$ 的单点值，直接求 $d[1\ldots x]$ 的和即可。

这里直接给出「区间加区间和」的代码：

" 实现"

```
int t1[MAXN], t2[MAXN], n;

int lowbit(int x) { return x & (-x); }

void add(int k, int v) {
  int v1 = k * v;
  while (k <= n) {
    t1[k] += v, t2[k] += v1;
    // 注意不能写成 t2[k] += k * v，因为 k 的值已经不是原数组的下标了
    k += lowbit(k);
  }
}

int getsum(int *t, int k) {
  int ret = 0;
  while (k) {
    ret += t[k];
    k -= lowbit(k);
  }
  return ret;
}

void add1(int l, int r, int v) {
  add(l, v), add(r + 1, -v);  // 将区间加差分为两个前缀加
}
```

```
long long getsum1(int l, int r) {
  return (r + 1ll) * getsum(t1, r) - 1ll * l * getsum(t1, l - 1) -
         (getsum(t2, r) - getsum(t2, l - 1));
}

t1 = [0] * MAXN, t2 = [0] * MAXN; n = 0

def lowbit(x):
    return x & (-x)

def add(k, v):
    v1 = k * v
    while k <= n:
        t1[k] = t1[k] + v; t2[k] = t2[k] + v1
        k = k + lowbit(k)

def getsum(t, k):
    ret = 0
    while k:
        ret = ret + t[k]
        k = k - lowbit(k)
    return ret

def add1(l, r, v):
    add(l, v)
    add(r + 1, -v)

def getsum1(l, r):
    return (r) * getsum(t1, r) - l * getsum(t1, l - 1) - \
          (getsum(t2, r) - getsum(t2, l - 1))
```

根据这个原理，应该可以实现「区间乘区间积」,「区间异或一个数，求区间异或值」等，只要满足维护的信息和区间操作是同种运算即可，感兴趣的读者可以自己尝试。

二维树状数组

单点修改，子矩阵查询

二维树状数组，也被称作树状数组套树状数组，用来维护二维数组上的单点修改和前缀信息问题。

与一维树状数组类似，我们用 $c(x,y)$ 表示 $a(x-\operatorname{lowbit}(x)+1, y-\operatorname{lowbit}(y)+1)\ldots a(x,y)$ 的矩阵总信息，即一个以 $a(x,y)$ 为右下角，高 $\operatorname{lowbit}(x)$，宽 $\operatorname{lowbit}(y)$ 的矩阵的总信息。

对于单点修改，设：

$$f(x,i)=\begin{cases}x & i=0\\ f(x,i-1)+\operatorname{lowbit}(f(x,i-1)) & i>0\end{cases}$$

即 $f(x,i)$ 为 x 在树状数组树形态上的第 i 级祖先（第 0 级祖先是自己）。

则只有 $c(f(x,i),f(y,j))$ 中的元素管辖 $a(x,y)$，修改 $a(x,y)$ 时只需修改所有 $c(f(x,i),f(y,j))$，其中 $f(x,i)\le n$，$f(y,j)\le m$。

" 正确性证明"

$c(p,q)$ 管辖 $a(x,y)$，求 p 和 q 的取值范围。

考虑一个大小为 n 的一维树状数组 c_1（对应原数组 a_1）和一个大小为 m 的一维树状数组 c_2（对应原数组

a_2）。

则命题等价为：$c_1(p)$ 管辖 $a_1[x]$ 且 $c_2(q)$ 管辖 $a_2[y]$ 的条件。

也就是说，在树状数组树形态上，p 是 x 及其祖先中的一个点，q 是 y 及其祖先中的一个点。

所以 $p = f(x, i)$，$q = f(y, j)$。

对于查询，我们设：

$$g(x,i) = \begin{cases} x & i = 0 \\ g(x, i-1) - \text{lowbit}(g(x, i-1)) & i, g(x, i-1) > 0 \\ 0 & \text{otherwise.} \end{cases}$$

则合并所有 $c(g(x,i), g(y,j))$，其中 $g(x,i), g(y,j) > 0$。

" **正确性证明**"

设 $\circ$ 表示合并两个信息的运算符（比如，如果信息是区间和，则 $\circ = +$）。

考虑一个一维树状数组 c_1，$c_1[g(x,0)] \circ c_1[g(x,1)] \circ c_1[g(x,2)] \circ \cdots$ 恰好表示原数组上 $[1 \ldots x]$ 这段区间信息。

类似地，设 $t(x) = c(x, g(y,0)) \circ c(x, g(y,1)) \circ c(x, g(y,2)) \circ \cdots$，则 $t(x)$ 恰好表示 $a(x - \text{lowbit}(x) + 1, 1) \ldots a(x, y)$ 这个矩阵信息。

又类似地，就有 $t(g(x,0)) \circ t(g(x,1)) \circ t(g(x,2)) \circ \cdots$ 表示 $a(1,1) \ldots a(x,y)$ 这个矩阵信息。

其实这里 $t(x)$ 这个函数如果看成一个树状数组，相当于一个树状数组套了一个树状数组，这也就是「树状数组套树状数组」这个名字的来源。

下面给出单点加、查询子矩阵和的代码。

" **实现**"

```cpp
void add(int x, int y, int v) {
  for (int i = x; i <= n; i += lowbit(i)) {
    for (int j = y; j <= m; j += lowbit(j)) {
      // 注意这里必须得建循环变量，不能像一维数组一样直接 while (x <= n) 了
      c[i][j] += v;
    }
  }
}

int sum(int x, int y) {
  int res = 0;
  for (int i = x; i > 0; i -= lowbit(i)) {
    for (int j = y; j > 0; j -= lowbit(j)) {
      res += c[i][j];
    }
  }
  return res;
}

int ask(int x1, int y1, int x2, int y2) {
  // 查询子矩阵和
  return sum(x2, y2) - sum(x2, y1 - 1) - sum(x1 - 1, y2) + sum(x1 - 1, y1 - 1);
}
```

子矩阵加，求子矩阵和

前置知识：前缀和 & 差分和本页面区间加区间和一节。

和一维树状数组的「区间加区间和」问题类似，考虑维护差分数组。

二维数组上的差分数组是这样的：

$$d(i,j)=a(i,j)-a(i-1,j)-a(i,j-1)+a(i-1,j-1)$$

" 为什么这么定义？"

这是因为，理想规定状态下，在差分矩阵上做二维前缀和应该得到原矩阵，因为这是一对逆运算。

二维前缀和的公式是这样的：

$s(i,j)=s(i-1,j)+s(i,j-1)-s(i-1,j-1)+a(i,j)$。

所以，设 a 是原数组，d 是差分数组，有：

$a(i,j)=a(i-1,j)+a(i,j-1)-a(i-1,j-1)+d(i,j)$

移项就得到二维差分的公式了。

$d(i,j)=a(i,j)-a(i-1,j)-a(i,j-1)+a(i-1,j-1)$。

这样以来，对左上角 (x_1,y_1)，右下角 (x_2,y_2) 的子矩阵区间加 v，相当于在差分数组上，对 $d(x_1,y_1)$ 和 $d(x_2+1,y_2+1)$ 分别单点加 v，对 $d(x_2+1,y_1)$ 和 $d(x_1,y_2+1)$ 分别单点加 $-v$。

至于原因，把这四个 d 分别用定义式表示出来，分析一下每项的变化即可。

举个例子吧，初始差分数组为 0，给 $a(2,2)\ldots a(3,4)$ 子矩阵加 v 后差分数组会变为：

$$\begin{pmatrix}0&0&0&0&0\\0&v&0&0&-v\\0&0&0&0&0\\0&-v&0&0&v\end{pmatrix}$$

（其中 $a(2,2)\ldots a(3,4)$ 这个子矩阵恰好是上面位于中心的 2×3 大小的矩阵。）

因此，子矩阵加的做法是：转化为差分数组上的四个单点加操作。

现在考虑查询子矩阵和：

对于点 (x,y)，它的二维前缀和可以表示为：

$$\sum_{i=1}^{x}\sum_{j=1}^{y}\sum_{h=1}^{i}\sum_{k=1}^{j}d(h,k)$$

原因就是差分的前缀和的前缀和就是原本的前缀和。

和一维树状数组的「区间加区间和」问题类似，统计 $d(h,k)$ 的出现次数，为 $(x-h+1)\times(y-k+1)$。

然后接着推导：

$$\begin{aligned}&\sum_{i=1}^{x}\sum_{j=1}^{y}\sum_{h=1}^{i}\sum_{k=1}^{j}d(h,k)\\=&\sum_{i=1}^{x}\sum_{j=1}^{y}d(i,j)\times(x-i+1)\times(y-j+1)\\=&\sum_{i=1}^{x}\sum_{j=1}^{y}d(i,j)\times(xy+x+y+1)-d(i,j)\times i\times(y+1)-d(i,j)\times j\times(x+1)+d(i,j)\times i\times j\end{aligned}$$

所以我们需维护四个树状数组，分别维护 $d(i,j)$，$d(i,j)\times i$，$d(i,j)\times j$，$d(i,j)\times i\times j$ 的和信息。

当然了，和一维同理，如果只需要子矩阵加求单点值，维护一个差分数组然后询问前缀和就足够了。

下面给出代码：

" 实现"

```
typedef long long ll;
ll t1[N][N], t2[N][N], t3[N][N], t4[N][N];

void add(ll x, ll y, ll z) {
  for (int X = x; X <= n; X += lowbit(X))
    for (int Y = y; Y <= m; Y += lowbit(Y)) {
      t1[X][Y] += z;
      t2[X][Y] += z * x;  // 注意是 z * x 而不是 z * X，后面同理
      t3[X][Y] += z * y;
      t4[X][Y] += z * x * y;
    }
}

void range_add(ll xa, ll ya, ll xb, ll yb,
               ll z) {  //(xa, ya) 到 (xb, yb) 子矩阵
  add(xa, ya, z);
  add(xa, yb + 1, -z);
  add(xb + 1, ya, -z);
  add(xb + 1, yb + 1, z);
}

ll ask(ll x, ll y) {
  ll res = 0;
  for (int i = x; i; i -= lowbit(i))
    for (int j = y; j; j -= lowbit(j))
      res += (x + 1) * (y + 1) * t1[i][j] - (y + 1) * t2[i][j] -
             (x + 1) * t3[i][j] + t4[i][j];
  return res;
}

ll range_ask(ll xa, ll ya, ll xb, ll yb) {
  return ask(xb, yb) - ask(xb, ya - 1) - ask(xa - 1, yb) + ask(xa - 1, ya - 1);
}
```

权值树状数组及应用

我们知道，普通树状数组直接在原序列的基础上构建，c_6 表示的就是 $a[5\ldots6]$ 的区间信息。

然而事实上，我们还可以在原序列的权值数组上构建树状数组，这就是权值树状数组。

" 什么是权值数组？"

一个序列 a 的权值数组 b，满足 $b[x]$ 的值为 x 在 a 中的出现次数。

例如：$a=(1,3,4,3,4)$ 的权值数组为 $b=(1,0,2,2)$。

很明显，b 的大小和 a 的值域有关。

若原数列值域过大，且重要的不是具体值而是值与值之间的相对大小关系，常离散化原数组后再建立权值数组。

另外，权值数组是原数组无序性的一种表示：它重点描述数组的元素内容，忽略了数组的顺序，若两数组只是顺序不同，所含内容一致，则它们的权值数组相同。

因此，对于给定数组的顺序不影响答案的问题，在权值数组的基础上思考一般更直观，比如 [NOIP2021] 数列[2]。

运用权值树状数组，我们可以解决一些经典问题。

单点修改，查询全局第 k 小

在此处只讨论第 k 小，第 k 大问题可以通过简单计算转化为第 k 小问题。

该问题可离散化，如果原序列 a 值域过大，离散化后再建立权值数组 b。注意，还要把单点修改中的涉及到的值也一起离散化，不能只离散化原数组 a 中的元素。

对于单点修改，只需将对原数列的单点修改转化为对权值数组的单点修改即可。具体来说，原数组 $a[x]$ 从 y 修改为 z，转化为对权值数组 b 的单点修改就是 $b[y]$ 单点减 1，$b[z]$ 单点加 1。

对于查询第 k 小，考虑二分 x，查询权值数组中 $[1,x]$ 的前缀和，找到 x_0 使得 $[1,x_0]$ 的前缀和 $<k$ 而 $[1,x_0+1]$ 的前缀和 $\geq k$，则第 k 大的数是 x_0+1（注：这里认为 $[1,0]$ 的前缀和是 0）。

这样做时间复杂度是 $\Theta(\log^2 n)$ 的。

考虑用倍增替代二分。

设 $x=0$，$\text{sum}=0$，枚举 i 从 $\log_2 n$ 降为 0：

- 查询权值数组中 $[x+1\ldots x+2^i]$ 的区间和 t。
- 如果 $\text{sum}+t<k$，扩展成功，$x\leftarrow x+2^i$，$\text{sum}\leftarrow\text{sum}+t$；否则扩展失败，不操作。

这样得到的 x 是满足 $[1\ldots x]$ 前缀和 $<k$ 的最大值，所以最终 $x+1$ 就是答案。

看起来这种方法时间效率没有任何改善，但事实上，查询 $[x+1\ldots x+2^i]$ 的区间和只需访问 $c[x+2^i]$ 的值即可。

原因很简单，考虑 $\text{lowbit}(x+2^i)$，它一定是 2^i，因为 x 之前只累加过 2^j 满足 $j>i$。因此 $c[x+2^i]$ 表示的区间就是 $[x+1\ldots x+2^i]$。

如此以来，时间复杂度降低为 $\Theta(\log n)$。

" 实现"

```cpp
// 权值树状数组查询第 k 小
int kth(int k) {
  int sum = 0, x = 0;
  for (int i = log2(n); ~i; --i) {
    x += 1 << i;                    // 尝试扩展
    if (x >= n || sum + t[x] >= k)  // 如果扩展失败
      x -= 1 << i;
    else
      sum += t[x];
  }
  return x + 1;
}
```

```python
# 权值树状数组查询第 k 小
def kth(k):
    sum = 0; x = 0
    i = log2(n)
    while ~i:
        x = x + (1 << i) # 尝试扩展
        if x >= n or sum + t[x] >= k: # 如果扩展失败
            x = x - (1 << i)
        else:
            sum = sum + t[x]
    return x + 1
```

全局逆序对（全局二维偏序）

全局逆序对也可以用权值树状数组巧妙解决。问题是这样的：给定长度为 n 的序列 a，求 a 中满足 $i<j$ 且 $a[i]>a[j]$ 的数对 (i,j) 的数量。

该问题可离散化，如果原序列 a 值域过大，离散化后再建立权值数组 b。

我们考虑从 n 到 1 倒序枚举 i，作为逆序对中第一个元素的索引，然后计算有多少个 $j>i$ 满足 $a[j]<a[i]$，最后累计答案即可。

事实上，我们只需要这样做（设当前 $a[i]=x$）：

- 查询 $b[1\dots x-1]$ 的前缀和，即为左端点为 $a[i]$ 的逆序对数量。
- $b[x]$ 自增 1；

原因十分自然：出现在 $b[1\dots x-1]$ 中的元素一定比当前的 $x=a[i]$ 小，而 i 的倒序枚举，自然使得这些已在权值数组中的元素，在原数组上的索引 j 大于当前遍历到的索引 i。

用例子说明，$a=(4,3,1,2,1)$。

i 按照 $5\to 1$ 扫：

- $a[5]=1$，查询 $b[1\dots 0]$ 前缀和，为 0，$b[1]$ 自增 1，$b=(1,0,0,0)$。
- $a[4]=2$，查询 $b[1\dots 1]$ 前缀和，为 1，$b[2]$ 自增 1，$b=(1,1,0,0)$。
- $a[3]=1$，查询 $b[1\dots 0]$ 前缀和，为 0，$b[1]$ 自增 1，$b=(2,1,0,0)$。
- $a[2]=3$，查询 $b[1\dots 2]$ 前缀和，为 3，$b[3]$ 自增 1，$b=(2,1,1,0)$。
- $a[1]=4$，查询 $b[1\dots 3]$ 前缀和，为 4，$b[4]$ 自增 1，$b=(2,1,1,1)$。

所以最终答案为 $0+1+0+3+4=8$。

注意到，遍历 i 后的查询 $b[1\dots x-1]$ 和自增 $b[x]$ 的两个步骤可以颠倒，变成先自增 $b[x]$ 再查询 $b[1\dots x-1]$，不影响答案。两个角度来解释：

- 对 $b[x]$ 的修改不影响对 $b[1\dots x-1]$ 的查询。
- 颠倒后，实质是在查询 $i\le j$ 且 $a[i]>a[j]$ 的数对数量，而 $i=j$ 时不存在 $a[i]>a[j]$，所以 $i\le j$ 相当于 $i<j$，所以这与原来的逆序对问题是等价的。

如果查询非严格逆序对（$i<j$ 且 $a[i]\ge a[j]$）的数量，那就要改为查询 $b[1\dots x]$ 的和，这时就不能颠倒两步了，还是两个角度来解释：

- 对 $b[x]$ 的修改**影响**对 $b[1\dots x]$ 的查询。
- 颠倒后，实质是在查询 $i\le j$ 且 $a[i]\ge a[j]$ 的数对数量，而 $i=j$ 时恒有 $a[i]\ge a[j]$，所以 $i\le j$ **不相当于** $i<j$，与原问题**不等价**。

如果查询 $i\le j$ 且 $a[i]\ge a[j]$ 的数对数量，那这两步就需要颠倒了。

另外，对于原逆序对问题，还有一种做法是正着枚举 j，查询有多少 $i<j$ 满足 $a[i]>a[j]$。做法如下（设 $x=a[j]$）：

- 查询 $b[x+1\dots V]$（V 是 b 的大小，即 a 的值域（或离散化后的值域））的区间和。
- 将 $b[x]$ 自增 1。

原因：出现在 $b[x+1\dots V]$ 中的元素一定比当前的 $x=a[j]$ 大，而 j 的正序枚举，自然使得这些已在权值数组中的元素，在原数组上的索引 i 小于当前遍历到的索引 j。

树状数组维护不可差分信息

比如维护区间最值等。

注意，这种方法虽然码量小，但单点修改和区间查询的时间复杂度均为 $\Theta(\log^2 n)$，比使用线段树的时间复杂度 $\Theta(\log n)$ 劣。

区间查询

我们还是基于之前的思路，从 r 沿着 lowbit 一直向前跳，但是我们不能跳到 l 的左边。

因此，如果我们跳到了 $c[x]$，先判断下一次要跳到的 $x-\text{lowbit}(x)$ 是否小于 l：

- 如果小于 l，我们直接把$a[x]$ **单点**合并到总信息里，然后跳到 $c[x-1]$。
- 如果大于等于 l，说明没越界，正常合并 $c[x]$，然后跳到 $c[x-\text{lowbit}(x)]$ 即可。

下面以查询区间最大值为例，给出代码：

" 实现"

```
int getmax(int l, int r) {
  int ans = 0;
  while (r >= l) {
    ans = max(ans, a[r]);
    --r;
    for (; r - lowbit(r) >= l; r -= lowbit(r)) {
      // 注意，循环条件不要写成 r - lowbit(r) + 1 >= l
      // 否则 l = 1 时，r 跳到 0 会死循环
      ans = max(ans, C[r]);
    }
  }
  return ans;
}
```

可以证明，上述算法的时间复杂度是 $\Theta(\log^2 n)$。

" 时间复杂度证明"

考虑 r 和 l 不同的最高位，一定有 r 在这一位上为 1，l 在这一位上为 0（因为 $r \ge l$）。

如果 r 在这一位的后面仍然有 1，一定有 $r-\text{lowbit}(r) \ge l$，所以下一步一定是把 r 的最低位 1 填为 0；

如果 r 的这一位 1 就是 r 的最低位 1，无论是 $r \leftarrow r-\text{lowbit}(r)$ 还是 $r \leftarrow r-1$，r 的这一位 1 一定会变为 0。

因此，r 经过至多 $\log n$ 次变换后，r 和 l 不同的最高位一定可以下降一位。所以，总时间复杂度是 $\Theta(\log^2 n)$。

单点更新

" 注"

请先理解树状数组树形态的以下两条性质，再学习本节。

- 设 $u = s \times 2^{k+1} + 2^k$，则其儿子数量为 $k = \log_2 \text{lowbit}(u)$，编号分别为 $u - 2^t (0 \le t < k)$。
- u 的所有儿子对应 c 的管辖区间恰好拼接成 $[l(u), u-1]$。

关于这两条性质的含义及证明，都可以在本页面的树状数组与其树形态的性质一节找到。

更新 $a[x]$ 后，我们只需要更新满足在树状数组树形态上，满足 y 是 x 的祖先的 $c[y]$。

对于最值（以最大值为例），一种常见的错误想法是，如果 $a[x]$ 修改成 p，则将所有 $c[y]$ 更新为 $\max(c[y], p)$。下面是一个反例：$(1,2,3,4,5)$ 中将 5 修改成 4，最大值是 4，但按照上面的修改这样会得到 5。将 $c[y]$ 直接修改为 p 也是错误的，一个反例是，将上面例子中的 3 修改为 4。

事实上，对于不可差分信息，不存在通过 p 直接修改 $c[y]$ 的方式。这是因为修改本身就相当于是把旧数从原区间「移除」，然后加入一个新数。「移除」时对区间信息的影响，相当于做「逆运算」，而不可差分信息不存在「逆运算」，所以无法直接修改 $c[y]$。

换句话说，对每个受影响的 $c[y]$，这个区间的信息我们必定要重构了。

考虑 $c[y]$ 的儿子们，它们的信息一定是正确的（因为我们先更新儿子再更新父亲），而这些儿子又恰好组成了 $[l(y), y-1]$ 这一段管辖区间，那再合并一个单点 $a[y]$ 就可以合并出 $[l(y), y]$，也就是 $c[y]$ 了。这样，我们能用至多 $\log n$ 个区间重构合并出每个需要修改的 c。

" 实现"

```cpp
void update(int x, int v) {
  a[x] = v;
  for (int i = x; i <= n; i += lowbit(i)) {
    // 枚举受影响的区间
    C[i] = a[i];
    for (int j = 1; j < lowbit(i); j *= 2) {
      C[i] = max(C[i], C[i - j]);
    }
  }
}
```

容易看出上述算法时间复杂度为 $\Theta(\log^2 n)$。

建树

可以考虑拆成 n 个单点修改，$\Theta(n\log^2 n)$ 建树。

也有 $\Theta(n)$ 的建树方法，见本页面 $\Theta(n)$ 建树一节的方法一。

Tricks

$\Theta(n)$ 建树

以维护区间和为例。

方法一：

每一个节点的值是由所有与自己直接相连的儿子的值求和得到的。因此可以倒着考虑贡献，即每次确定完儿子的值后，用自己的值更新自己的直接父亲。

" 实现"

```cpp
// Θ(n) 建树
void init() {
  for (int i = 1; i <= n; ++i) {
    t[i] += a[i];
    int j = i + lowbit(i);
    if (j <= n) t[j] += t[i];
  }
}
```

```python
# Θ(n) 建树
def init():
    for i in range(1, n + 1):
        t[i] = t[i] + a[i]
        j = i + lowbit(i)
        if j <= n:
            t[j] = t[j] + t[i]
```

方法二：

前面讲到 $c[i]$ 表示的区间是 $[i-\text{lowbit}(i)+1,i]$，那么我们可以先预处理一个 sum 前缀和数组，再计算 c 数组。

" 实现"

```
// Θ(n) 建树
void init() {
  for (int i = 1; i <= n; ++i) {
    t[i] = sum[i] - sum[i - lowbit(i)];
  }
}

# Θ(n) 建树
def init():
    for i in range(1, n + 1):
        t[i] = sum[i] - sum[i-lowbit(i)]
```

时间戳优化

对付多组数据很常见的技巧。若每次输入新数据都暴力清空树状数组，就可能会造成超时。因此使用 tag 标记，存储当前节点上次使用时间（即最近一次是被第几组数据使用）。每次操作时判断这个位置 tag 中的时间和当前时间是否相同，就可以判断这个位置应该是 0 还是数组内的值。

" 实现"

```
// 时间戳优化
int tag[MAXN], t[MAXN], Tag;

void reset() { ++Tag; }

void add(int k, int v) {
  while (k <= n) {
    if (tag[k] != Tag) t[k] = 0;
    t[k] += v, tag[k] = Tag;
    k += lowbit(k);
  }
}

int getsum(int k) {
  int ret = 0;
  while (k) {
    if (tag[k] == Tag) ret += t[k];
    k -= lowbit(k);
  }
  return ret;
}

# 时间戳优化
tag = [0] * MAXN; t = [0] * MAXN; Tag = 0
def reset():
    Tag = Tag + 1
def add(k, v):
    while k <= n:
```

```
        if tag[k] != Tag:
            t[k] = 0
        t[k] = t[k] + v
        tag[k] = Tag
        k = k + lowbit(k)
def getsum(k):
    ret = 0
    while k:
        if tag[k] == Tag:
            ret = ret + t[k]
        k = k - lowbit(k)
    return ret
```

例题

- 树状数组 1: 单点修改，区间查询[3]
- 树状数组 2: 区间修改，单点查询[4]
- 树状数组 3: 区间修改，区间查询[5]
- 二维树状数组 1: 单点修改，区间查询[6]
- 二维树状数组 2: 区间修改，单点查询[7]
- 二维树状数组 3: 区间修改，区间查询[8]

参考资料与注释

[1] Efficient Range Minimum Queries using Binary Indexed Trees

[2] [NOIP2021] 数列

[3] 树状数组 1：单点修改，区间查询

[4] 树状数组 2：区间修改，单点查询

[5] 树状数组 3：区间修改，区间查询

[6] 二维树状数组 1：单点修改，区间查询

[7] 二维树状数组 2：区间修改，单点查询

[8] 二维树状数组 3：区间修改，区间查询

10.13 线段树

Authors: Marcythm, Ir1d, Ycrpro, Xeonacid, konnyakuxzy, CJSoft, HeRaNO, ethan-enhe, ChungZH, Chrogeek, hsfzLZH1, billchenchina, orzAtalod, luoguojie, Early0v0, wy-luke

引入

线段树是算法竞赛中常用的用来维护**区间信息**的数据结构。

线段树可以在 $O(\log N)$ 的时间复杂度内实现单点修改、区间修改、区间查询（区间求和，求区间最大值，求区间最小值）等操作。

线段树

线段树的基本结构与建树

过程

线段树将每个长度不为 1 的区间划分成左右两个区间递归求解，把整个线段划分为一个树形结构，通过合并左右两区间信息来求得该区间的信息。这种数据结构可以方便的进行大部分的区间操作。

有个大小为 5 的数组 $a=\{10,11,12,13,14\}$，要将其转化为线段树，有以下做法：设线段树的根节点编号为 1，用数组 d 来保存我们的线段树，d_i 用来保存线段树上编号为 i 的节点的值（这里每个节点所维护的值就是这个节点所表示的区间总和）。

我们先给出这棵线段树的形态，如图所示：

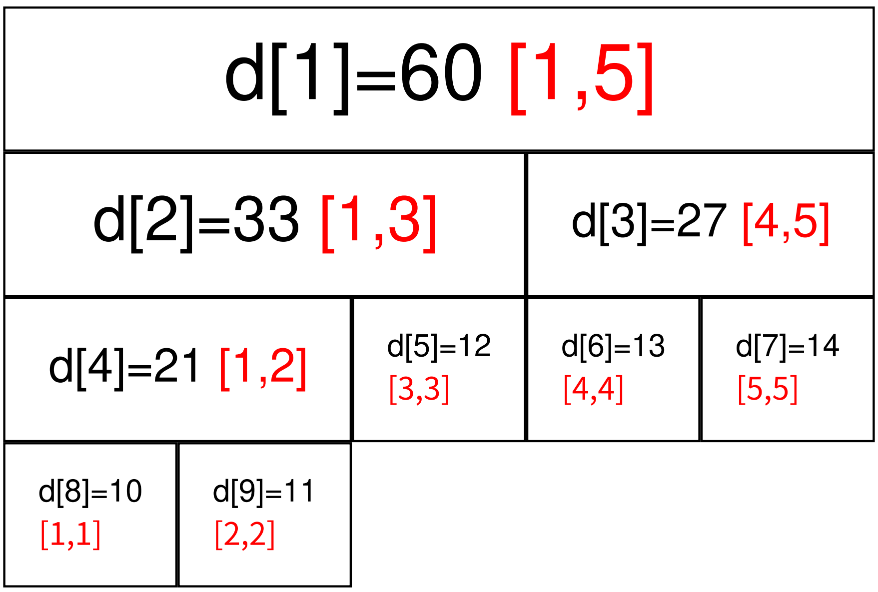


图 10.36

图中每个节点中用红色字体标明的区间，表示该节点管辖的 a 数组上的位置区间。如 d_1 所管辖的区间就是 $[1,5]$ $(a_1,a_2,\cdots,a_5)$，即 d_1 所保存的值是 $a_1+a_2+\cdots+a_5$，$d_1=60$ 表示的是 $a_1+a_2+\cdots+a_5=60$。

通过观察不难发现，d_i 的左儿子节点就是 $d_{2\times i}$，d_i 的右儿子节点就是 $d_{2\times i+1}$。如果 d_i 表示的是区间 $[s,t]$（即 $d_i=a_s+a_{s+1}+\cdots+a_t$）的话，那么 d_i 的左儿子节点表示的是区间 $[s,\frac{s+t}{2}]$，d_i 的右儿子表示的是区间 $[\frac{s+t}{2}+1,t]$。

在实现时，我们考虑递归建树。设当前的根节点为 p，如果根节点管辖的区间长度已经是 1，则可以直接根据 a 数组上相应位置的值初始化该节点。否则我们将该区间从中点处分割为两个子区间，分别进入左右子节点递归建树，

最后合并两个子节点的信息。

实现

此处给出代码实现，可参考注释理解：

```cpp
void build(int s, int t, int p) {
  // 对 [s,t] 区间建立线段树，当前根的编号为 p
  if (s == t) {
    d[p] = a[s];
    return;
  }
  int m = s + ((t - s) >> 1);
  // 移位运算符的优先级小于加减法，所以加上括号
  // 如果写成 (s + t) >> 1 可能会超出 int 范围
  build(s, m, p * 2), build(m + 1, t, p * 2 + 1);
  // 递归对左右区间建树
  d[p] = d[p * 2] + d[(p * 2) + 1];
}
```

```python
def build(s, t, p):
    # 对 [s,t] 区间建立线段树，当前根的编号为 p
    if s == t:
        d[p] = a[s]
        return
    m = s + ((t - s) >> 1)
    # 移位运算符的优先级小于加减法，所以加上括号
    # 如果写成 (s + t) >> 1 可能会超出 int 范围
    build(s, m, p * 2); build(m + 1, t, p * 2 + 1)
    # 递归对左右区间建树
    d[p] = d[p * 2] + d[(p * 2) + 1]
```

关于线段树的空间：如果采用堆式存储（$2p$ 是 p 的左儿子，$2p+1$ 是 p 的右儿子），若有 n 个叶子结点，则 d 数组的范围最大为 $2^{\lceil \log n \rceil + 1}$。

分析：容易知道线段树的深度是 $\lceil \log n \rceil$ 的，则在堆式储存情况下叶子节点（包括无用的叶子节点）数量为 $2^{\lceil \log n \rceil}$ 个，又由于其为一棵完全二叉树，则其总节点个数 $2^{\lceil \log n \rceil + 1} - 1$。当然如果你懒得计算的话可以直接把数组长度设为 $4n$，因为 $\frac{2^{\lceil \log n \rceil + 1} - 1}{n}$ 的最大值在 $n = 2^x + 1 (x \in N_+)$ 时取到，此时节点数为 $2^{\lceil \log n \rceil + 1} - 1 = 2^{x+2} - 1 = 4n - 5$。

线段树的区间查询

过程

区间查询，比如求区间 $[l, r]$ 的总和（即 $a_l + a_{l+1} + \cdots + a_r$）、求区间最大值/最小值等操作。

仍然以最开始的图为例，如果要查询区间 $[1, 5]$ 的和，那直接获取 d_1 的值（60）即可。

如果要查询的区间为 $[3, 5]$，此时就不能直接获取区间的值，但是 $[3, 5]$ 可以拆成 $[3, 3]$ 和 $[4, 5]$，可以通过合并这两个区间的答案来求得这个区间的答案。

一般地，如果要查询的区间是 $[l, r]$，则可以将其拆成最多为 $O(\log n)$ 个**极大**的区间，合并这些区间即可求出 $[l, r]$ 的答案。

实现

此处给出代码实现，可参考注释理解：

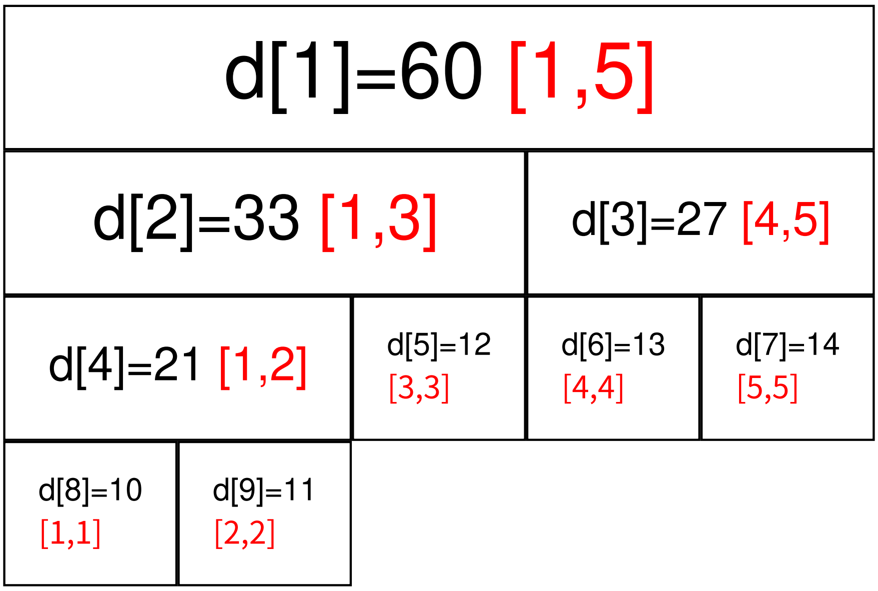


图 10.37

```cpp
int getsum(int l, int r, int s, int t, int p) {
  // [l, r] 为查询区间，[s, t] 为当前节点包含的区间，p 为当前节点的编号
  if (l <= s && t <= r)
    return d[p];  // 当前区间为询问区间的子集时直接返回当前区间的和
  int m = s + ((t - s) >> 1), sum = 0;
  if (l <= m) sum += getsum(l, r, s, m, p * 2);
  // 如果左儿子代表的区间 [s, m] 与询问区间有交集，则递归查询左儿子
  if (r > m) sum += getsum(l, r, m + 1, t, p * 2 + 1);
  // 如果右儿子代表的区间 [m + 1, t] 与询问区间有交集，则递归查询右儿子
  return sum;
}
```

```python
def getsum(l, r, s, t, p):
    # [l, r] 为查询区间，[s, t] 为当前节点包含的区间，p 为当前节点的编号
    if l <= s and t <= r:
        return d[p] # 当前区间为询问区间的子集时直接返回当前区间的和
    m = s + ((t - s) >> 1); sum = 0
    if l <= m:
        sum = sum + getsum(l, r, s, m, p * 2)
    # 如果左儿子代表的区间 [s, m] 与询问区间有交集，则递归查询左儿子
    if r > m:
        sum = sum + getsum(l, r, m + 1, t, p * 2 + 1)
    # 如果右儿子代表的区间 [m + 1, t] 与询问区间有交集，则递归查询右儿子
    return sum
```

线段树的区间修改与懒惰标记

过程

如果要求修改区间 $[l,r]$，把所有包含在区间 $[l,r]$ 中的节点都遍历一次、修改一次，时间复杂度无法承受。我们这里要引入一个叫做「**懒惰标记**」的东西。

懒惰标记，简单来说，就是通过延迟对节点信息的更改，从而减少可能不必要的操作次数。每次执行修改时，我们通过打标记的方法表明该节点对应的区间在某一次操作中被更改，但不更新该节点的子节点的信息。实质性的修改则在下一次访问带有标记的节点时才进行。

仍然以最开始的图为例，我们将执行若干次给区间内的数加上一个值的操作。我们现在给每个节点增加一个 t_i，表示该节点带的标记值。

最开始时的情况是这样的（为了节省空间，这里不再展示每个节点管辖的区间）：

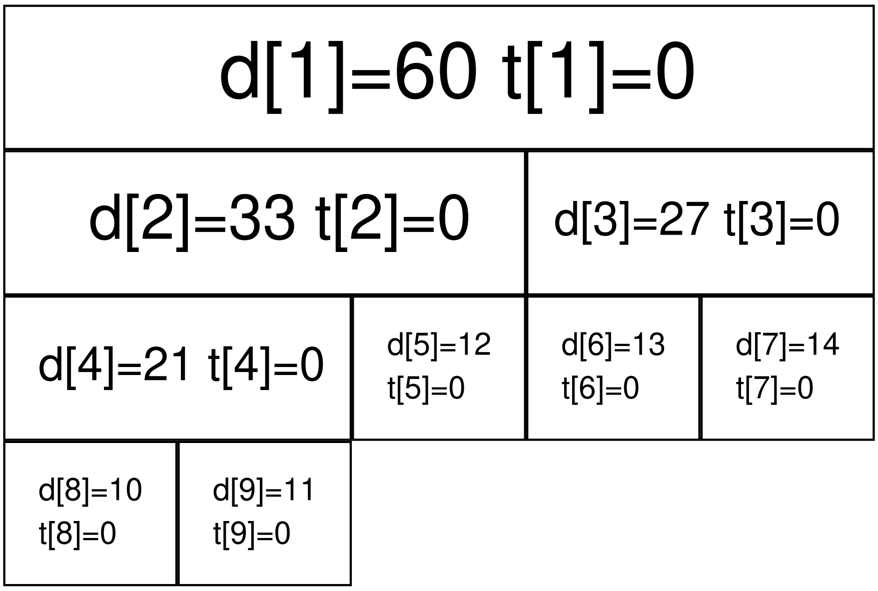


图 10.38

现在我们准备给 $[3,5]$ 上的每个数都加上 5。根据前面区间查询的经验，我们很快找到了两个极大区间 $[3,3]$ 和 $[4,5]$（分别对应线段树上的 3 号点和 5 号点）。

我们直接在这两个节点上进行修改，并给它们打上标记：

我们发现，3 号节点的信息虽然被修改了（因为该区间管辖两个数，所以 d_3 加上的数是 $5\times 2=10$），但它的两个子节点却还没更新，仍然保留着修改之前的信息。不过不用担心，虽然修改目前还没进行，但当我们要查询这两个子节点的信息时，我们会利用标记修改这两个子节点的信息，使查询的结果依旧准确。

接下来我们查询一下 $[4,4]$ 区间上各数字的和。

我们通过递归找到 $[4,5]$ 区间，发现该区间并非我们的目标区间，且该区间上还存在标记。这时候就到标记下放的时间了。我们将该区间的两个子区间的信息更新，并清除该区间上的标记。

现在 6、7 两个节点的值变成了最新的值，查询的结果也是准确的。

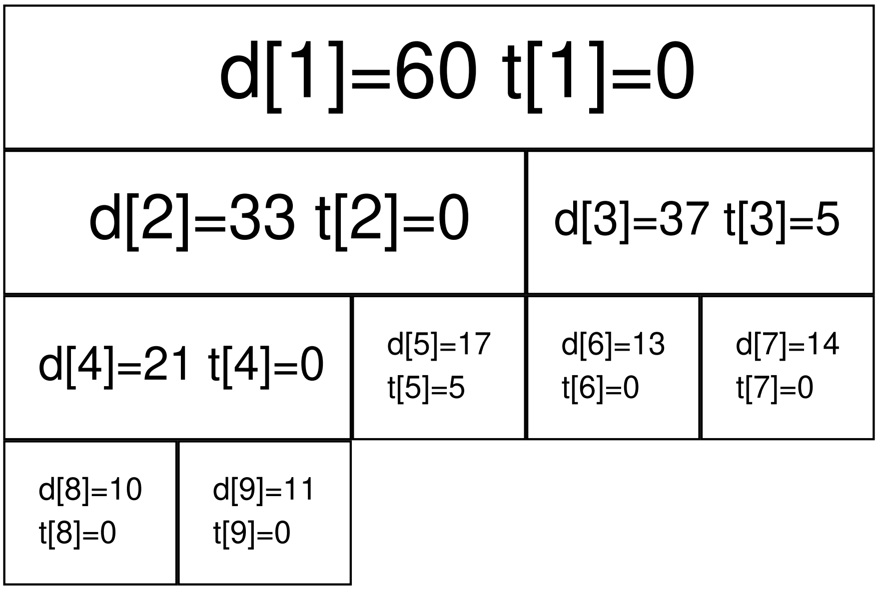

d[1]=60 t[1]=0
d[2]=33 t[2]=0
d[3]=37 t[3]=5
d[4]=21 t[4]=0
d[5]=17
t[5]=5
d[6]=13
t[6]=0
d[7]=14
t[7]=0
d[8]=10
t[8]=0
d[9]=11
t[9]=0

图 10.39

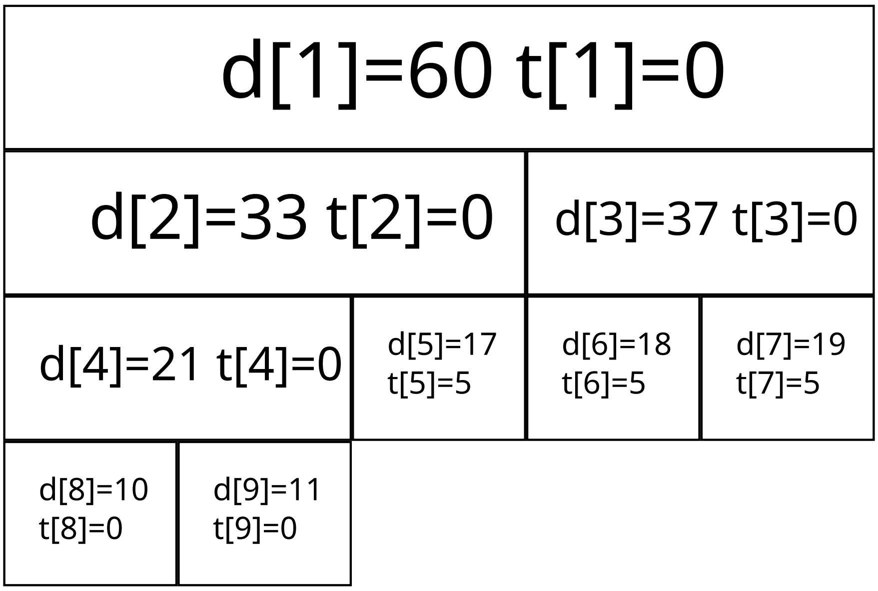

d[1]=60 t[1]=0
d[2]=33 t[2]=0
d[3]=37 t[3]=0
d[4]=21 t[4]=0
d[5]=17
t[5]=5
d[6]=18
t[6]=5
d[7]=19
t[7]=5
d[8]=10
t[8]=0
d[9]=11
t[9]=0

图 10.40

实现

接下来给出在存在标记的情况下，区间修改和查询操作的参考实现。

区间修改（区间加上某个值）：

```
void update(int l, int r, int c, int s, int t, int p) {
  // [l, r] 为修改区间，c 为被修改的元素的变化量，[s, t] 为当前节点包含的区间，p
  // 为当前节点的编号
  if (l <= s && t <= r) {
    d[p] += (t - s + 1) * c, b[p] += c;
    return;
  }  // 当前区间为修改区间的子集时直接修改当前节点的值，然后打标记，结束修改
  int m = s + ((t - s) >> 1);
  if (b[p] && s != t) {
    // 如果当前节点的懒标记非空，则更新当前节点两个子节点的值和懒标记值
    d[p * 2] += b[p] * (m - s + 1), d[p * 2 + 1] += b[p] * (t - m);
    b[p * 2] += b[p], b[p * 2 + 1] += b[p];  // 将标记下传给子节点
    b[p] = 0;                                // 清空当前节点的标记
  }
  if (l <= m) update(l, r, c, s, m, p * 2);
  if (r > m) update(l, r, c, m + 1, t, p * 2 + 1);
  d[p] = d[p * 2] + d[p * 2 + 1];
}
```

```
def update(l, r, c, s, t, p):
    # [l, r] 为修改区间，c 为被修改的元素的变化量，[s, t] 为当前节点包含的区间，
p
    # 为当前节点的编号
    if l <= s and t <= r:
        d[p] = d[p] + (t - s + 1) * c
        b[p] = b[p] + c
        return
    # 当前区间为修改区间的子集时直接修改当前节点的值，然后打标记，结束修改
    m = s + ((t - s) >> 1)
    if b[p] and s != t:
        # 如果当前节点的懒标记非空，则更新当前节点两个子节点的值和懒标记值
        d[p * 2] = d[p * 2] + b[p] * (m - s + 1)
        d[p * 2 + 1] = d[p * 2 + 1] + b[p] * (t - m)
        # 将标记下传给子节点
        b[p * 2] = b[p * 2] + b[p]
        b[p * 2 + 1] = b[p * 2 + 1] + b[p]
        # 清空当前节点的标记
        b[p] = 0
    if l <= m:
        update(l, r, c, s, m, p * 2)
    if r > m:
        update(l, r, c, m + 1, t, p * 2 + 1)
    d[p] = d[p * 2] + d[p * 2 + 1]
```

区间查询（区间求和）：

```
int getsum(int l, int r, int s, int t, int p) {
  // [l, r] 为查询区间，[s, t] 为当前节点包含的区间，p 为当前节点的编号
  if (l <= s && t <= r) return d[p];
  // 当前区间为询问区间的子集时直接返回当前区间的和
```

```
  int m = s + ((t - s) >> 1);
  if (b[p]) {
    // 如果当前节点的懒标记非空，则更新当前节点两个子节点的值和懒标记值
    d[p * 2] += b[p] * (m - s + 1), d[p * 2 + 1] += b[p] * (t - m);
    b[p * 2] += b[p], b[p * 2 + 1] += b[p];  // 将标记下传给子节点
    b[p] = 0;                                // 清空当前节点的标记
  }
  int sum = 0;
  if (l <= m) sum = getsum(l, r, s, m, p * 2);
  if (r > m) sum += getsum(l, r, m + 1, t, p * 2 + 1);
  return sum;
}
```

```
def getsum(l, r, s, t, p):
    # [l, r] 为查询区间，[s, t] 为当前节点包含的区间，p 为当前节点的编号
    if l <= s and t <= r:
        return d[p]
    # 当前区间为询问区间的子集时直接返回当前区间的和
    m = s + ((t - s) >> 1)
    if b[p]:
        # 如果当前节点的懒标记非空，则更新当前节点两个子节点的值和懒标记值
        d[p * 2] = d[p * 2] + b[p] * (m - s + 1)
        d[p * 2 + 1] = d[p * 2 + 1] + b[p] * (t - m)
        # 将标记下传给子节点
        b[p * 2] = b[p * 2] + b[p]
        b[p * 2 + 1] = b[p * 2 + 1] + b[p]
        # 清空当前节点的标记
        b[p] = 0
    sum = 0
    if l <= m:
        sum = getsum(l, r, s, m, p * 2)
    if r > m:
        sum = sum + getsum(l, r, m + 1, t, p * 2 + 1)
    return sum
```

如果你是要实现区间修改为某一个值而不是加上某一个值的话，代码如下：

```
void update(int l, int r, int c, int s, int t, int p) {
  if (l <= s && t <= r) {
    d[p] = (t - s + 1) * c, b[p] = c;
    return;
  }
  int m = s + ((t - s) >> 1);
  // 额外数组储存是否修改值
  if (v[p]) {
    d[p * 2] = b[p] * (m - s + 1), d[p * 2 + 1] = b[p] * (t - m);
    b[p * 2] = b[p * 2 + 1] = b[p];
    v[p * 2] = v[p * 2 + 1] = 1;
    v[p] = 0;
  }
  if (l <= m) update(l, r, c, s, m, p * 2);
  if (r > m) update(l, r, c, m + 1, t, p * 2 + 1);
  d[p] = d[p * 2] + d[p * 2 + 1];
}
```

```cpp
int getsum(int l, int r, int s, int t, int p) {
  if (l <= s && t <= r) return d[p];
  int m = s + ((t - s) >> 1);
  if (v[p]) {
    d[p * 2] = b[p] * (m - s + 1), d[p * 2 + 1] = b[p] * (t - m);
    b[p * 2] = b[p * 2 + 1] = b[p];
    v[p * 2] = v[p * 2 + 1] = 1;
    v[p] = 0;
  }
  int sum = 0;
  if (l <= m) sum = getsum(l, r, s, m, p * 2);
  if (r > m) sum += getsum(l, r, m + 1, t, p * 2 + 1);
  return sum;
}
```

```python
def update(l, r, c, s, t, p):
    if l <= s and t <= r:
        d[p] = (t - s + 1) * c
        b[p] = c
        return
    m = s + ((t - s) >> 1)
    if v[p]:
        d[p * 2] = b[p] * (m - s + 1)
        d[p * 2 + 1] = b[p] * (t - m)
        b[p * 2] = b[p * 2 + 1] = b[p]
        v[p * 2] = v[p * 2 + 1] = 1
        v[p] = 0
    if l <= m:
        update(l, r, c, s, m, p * 2)
    if r > m:
        update(l, r, c, m + 1, t, p * 2 + 1)
    d[p] = d[p * 2] + d[p * 2 + 1]

def getsum(l, r, s, t, p):
    if l <= s and t <= r:
        return d[p]
    m = s + ((t - s) >> 1)
    if v[p]:
        d[p * 2] = b[p] * (m - s + 1)
        d[p * 2 + 1] = b[p] * (t - m)
        b[p * 2] = b[p * 2 + 1] = b[p]
        v[p * 2] = v[p * 2 + 1] = 1
        v[p] = 0
    sum = 0
    if l <= m:
        sum = getsum(l, r, s, m, p * 2)
    if r > m:
        sum = sum + getsum(l, r, m + 1, t, p * 2 + 1)
    return sum
```

动态开点线段树

前面讲到堆式储存的情况下，需要给线段树开 $4n$ 大小的数组。为了节省空间，我们可以不一次性建好树，而是在最初只建立一个根结点代表整个区间。当我们需要访问某个子区间时，才建立代表这个区间的子结点。这样我们不

再使用 $2p$ 和 $2p+1$ 代表 p 结点的儿子，而是用 ls 和 rs 记录儿子的编号。总之，动态开点线段树的核心思想就是：**结点只有在有需要的时候才被创建**。

单次操作的时间复杂度是不变的，为 $O(\log n)$。由于每次操作都有可能创建并访问全新的一系列结点，因此 m 次单点操作后结点的数量规模是 $O(m\log n)$。最多也只需要 $2n-1$ 个结点，没有浪费。

单点修改：

```cpp
// root 表示整棵线段树的根结点；cnt 表示当前结点个数
int n, cnt, root;
int sum[n * 2], ls[n * 2], rs[n * 2];

// 用法：update(root, 1, n, x, f); 其中 x 为待修改节点的编号
void update(int& p, int s, int t, int x, int f) {  // 引用传参
  if (!p) p = ++cnt;  // 当结点为空时，创建一个新的结点
  if (s == t) {
    sum[p] += f;
    return;
  }
  int m = s + ((t - s) >> 1);
  if (x <= m)
    update(ls[p], s, m, x, f);
  else
    update(rs[p], m + 1, t, x, f);
  sum[p] = sum[ls[p]] + sum[rs[p]];  // pushup
}
```

区间询问：

```cpp
// 用法：query(root, 1, n, l, r);
int query(int p, int s, int t, int l, int r) {
  if (!p) return 0;  // 如果结点为空，返回 0
  if (s >= l && t <= r) return sum[p];
  int m = s + ((t - s) >> 1), ans = 0;
  if (l <= m) ans += query(ls[p], s, m, l, r);
  if (r > m) ans += query(rs[p], m + 1, t, l, r);
  return ans;
}
```

区间修改也是一样的，不过下放标记时要注意如果缺少孩子，就直接创建一个新的孩子。或者使用标记永久化技巧。

一些优化

这里总结几个线段树的优化：

- 在叶子节点处无需下放懒惰标记，所以懒惰标记可以不下传到叶子节点。
- 下放懒惰标记可以写一个专门的函数 `pushdown`，从儿子节点更新当前节点也可以写一个专门的函数 `maintain`（或者对称地用 `pushup`），降低代码编写难度。
- 标记永久化：如果确定懒惰标记不会在中途被加到溢出（即超过了该类型数据所能表示的最大范围），那么就可以将标记永久化。标记永久化可以避免下传懒惰标记，只需在进行询问时把标记的影响加到答案当中，从而降低程序常数。具体如何处理与题目特性相关，需结合题目来写。这也是树套树和可持久化数据结构中会用到的一种技巧。

C++ 模板

"SegTreeLazyRangeAdd 可以区间加/求和的线段树模板"

```
#include <bits/stdc++.h>
using namespace std;

template <typename T>
class SegTreeLazyRangeAdd {
  vector<T> tree, lazy;
  vector<T> *arr;
  int n, root, n4, end;

  void maintain(int cl, int cr, int p) {
    int cm = cl + (cr - cl) / 2;
    if (cl != cr && lazy[p]) {
      lazy[p * 2] += lazy[p];
      lazy[p * 2 + 1] += lazy[p];
      tree[p * 2] += lazy[p] * (cm - cl + 1);
      tree[p * 2 + 1] += lazy[p] * (cr - cm);
      lazy[p] = 0;
    }
  }

  T range_sum(int l, int r, int cl, int cr, int p) {
    if (l <= cl && cr <= r) return tree[p];
    int m = cl + (cr - cl) / 2;
    T sum = 0;
    maintain(cl, cr, p);
    if (l <= m) sum += range_sum(l, r, cl, m, p * 2);
    if (r > m) sum += range_sum(l, r, m + 1, cr, p * 2 + 1);
    return sum;
  }

  void range_add(int l, int r, T val, int cl, int cr, int p) {
    if (l <= cl && cr <= r) {
      lazy[p] += val;
      tree[p] += (cr - cl + 1) * val;
      return;
    }
    int m = cl + (cr - cl) / 2;
    maintain(cl, cr, p);
    if (l <= m) range_add(l, r, val, cl, m, p * 2);
    if (r > m) range_add(l, r, val, m + 1, cr, p * 2 + 1);
    tree[p] = tree[p * 2] + tree[p * 2 + 1];
  }

  void build(int s, int t, int p) {
    if (s == t) {
      tree[p] = (*arr)[s];
      return;
    }
    int m = s + (t - s) / 2;
    build(s, m, p * 2);
```

```
    build(m + 1, t, p * 2 + 1);
    tree[p] = tree[p * 2] + tree[p * 2 + 1];
  }

 public:
  explicit SegTreeLazyRangeAdd<T>(vector<T> v) {
    n = v.size();
    n4 = n * 4;
    tree = vector<T>(n4, 0);
    lazy = vector<T>(n4, 0);
    arr = &v;
    end = n - 1;
    root = 1;
    build(0, end, 1);
    arr = nullptr;
  }

  void show(int p, int depth = 0) {
    if (p > n4 || tree[p] == 0) return;
    show(p * 2, depth + 1);
    for (int i = 0; i < depth; ++i) putchar('\t');
    printf("%d:%d\n", tree[p], lazy[p]);
    show(p * 2 + 1, depth + 1);
  }

  T range_sum(int l, int r) { return range_sum(l, r, 0, end, root); }

  void range_add(int l, int r, int val) { range_add(l, r, val, 0, end, root); }
};
```

"SegTreeLazyRangeSet 可以区间修改/求和的线段树模板"

```
#include <bits/stdc++.h>
using namespace std;

template <typename T>
class SegTreeLazyRangeSet {
  vector<T> tree, lazy;
  vector<T> *arr;
  int n, root, n4, end;

  void maintain(int cl, int cr, int p) {
    int cm = cl + (cr - cl) / 2;
    if (cl != cr && lazy[p]) {
      lazy[p * 2] = lazy[p];
      lazy[p * 2 + 1] = lazy[p];
      tree[p * 2] = lazy[p] * (cm - cl + 1);
      tree[p * 2 + 1] = lazy[p] * (cr - cm);
      lazy[p] = 0;
    }
  }

  T range_sum(int l, int r, int cl, int cr, int p) {
```

```
    if (l <= cl && cr <= r) return tree[p];
    int m = cl + (cr - cl) / 2;
    T sum = 0;
    maintain(cl, cr, p);
    if (l <= m) sum += range_sum(l, r, cl, m, p * 2);
    if (r > m) sum += range_sum(l, r, m + 1, cr, p * 2 + 1);
    return sum;
  }

  void range_set(int l, int r, T val, int cl, int cr, int p) {
    if (l <= cl && cr <= r) {
      lazy[p] = val;
      tree[p] = (cr - cl + 1) * val;
      return;
    }
    int m = cl + (cr - cl) / 2;
    maintain(cl, cr, p);
    if (l <= m) range_set(l, r, val, cl, m, p * 2);
    if (r > m) range_set(l, r, val, m + 1, cr, p * 2 + 1);
    tree[p] = tree[p * 2] + tree[p * 2 + 1];
  }

  void build(int s, int t, int p) {
    if (s == t) {
      tree[p] = (*arr)[s];
      return;
    }
    int m = s + (t - s) / 2;
    build(s, m, p * 2);
    build(m + 1, t, p * 2 + 1);
    tree[p] = tree[p * 2] + tree[p * 2 + 1];
  }

 public:
  explicit SegTreeLazyRangeSet<T>(vector<T> v) {
    n = v.size();
    n4 = n * 4;
    tree = vector<T>(n4, 0);
    lazy = vector<T>(n4, 0);
    arr = &v;
    end = n - 1;
    root = 1;
    build(0, end, 1);
    arr = nullptr;
  }

  void show(int p, int depth = 0) {
    if (p > n4 || tree[p] == 0) return;
    show(p * 2, depth + 1);
    for (int i = 0; i < depth; ++i) putchar('\t');
    printf("%d:%d\n", tree[p], lazy[p]);
    show(p * 2 + 1, depth + 1);
  }
```

```
  T range_sum(int l, int r) { return range_sum(l, r, 0, end, root); }

  void range_set(int l, int r, int val) { range_set(l, r, val, 0, end, root); }
};
```

例题

"luogu P3372【模板】线段树 1[1]"

已知一个数列，你需要进行下面两种操作：

- 将某区间每一个数加上 k。
- 求出某区间每一个数的和。

" 参考代码"

```
#include <iostream>
typedef long long LL;
LL n, a[100005], d[270000], b[270000];

void build(LL l, LL r, LL p) {  // l: 区间左端点 r: 区间右端点 p: 节点标号
  if (l == r) {
    d[p] = a[l];  // 将节点赋值
    return;
  }
  LL m = l + ((r - l) >> 1);
  build(l, m, p << 1), build(m + 1, r, (p << 1) | 1);  // 分别建立子树
  d[p] = d[p << 1] + d[(p << 1) | 1];
}

void update(LL l, LL r, LL c, LL s, LL t, LL p) {
  if (l <= s && t <= r) {
    d[p] += (t - s + 1) * c, b[p] += c;  // 如果区间被包含了，直接得出答案
    return;
  }
  LL m = s + ((t - s) >> 1);
  if (b[p])
    d[p << 1] += b[p] * (m - s + 1), d[(p << 1) | 1] += b[p] * (t - m),
        b[p << 1] += b[p], b[(p << 1) | 1] += b[p];
  b[p] = 0;
  if (l <= m)
    update(l, r, c, s, m, p << 1);  // 本行和下面的一行用来更新 p*2 和 p*2+1 的节点
  if (r > m) update(l, r, c, m + 1, t, (p << 1) | 1);
  d[p] = d[p << 1] + d[(p << 1) | 1];  // 计算该节点区间和
}

LL getsum(LL l, LL r, LL s, LL t, LL p) {
  if (l <= s && t <= r) return d[p];
  LL m = s + ((t - s) >> 1);
  if (b[p])
    d[p << 1] += b[p] * (m - s + 1), d[(p << 1) | 1] += b[p] * (t - m),
        b[p << 1] += b[p], b[(p << 1) | 1] += b[p];
  b[p] = 0;
```

```
  LL sum = 0;
  if (l <= m)
    sum =
        getsum(l, r, s, m, p << 1);  // 本行和下面的一行用来更新 p*2 和 p*2+1 的答案
  if (r > m) sum += getsum(l, r, m + 1, t, (p << 1) | 1);
  return sum;
}

int main() {
  std::ios::sync_with_stdio(0);
  LL q, i1, i2, i3, i4;
  std::cin >> n >> q;
  for (LL i = 1; i <= n; i++) std::cin >> a[i];
  build(1, n, 1);
  while (q--) {
    std::cin >> i1 >> i2 >> i3;
    if (i1 == 2)
      std::cout << getsum(i2, i3, 1, n, 1) << std::endl;  // 直接调用操作函数
    else
      std::cin >> i4, update(i2, i3, i4, 1, n, 1);
  }
  return 0;
}
```

"luogu P3373【模板】线段树 2[2]"

已知一个数列，你需要进行下面三种操作：

- 将某区间每一个数乘上 x。
- 将某区间每一个数加上 x。
- 求出某区间每一个数的和。

" 参考代码"

```
#include <cstdio>
#define ll long long

ll read() {
  ll w = 1, q = 0;
  char ch = ' ';
  while (ch != '-' && (ch < '0' || ch > '9')) ch = getchar();
  if (ch == '-') w = -1, ch = getchar();
  while (ch >= '0' && ch <= '9') q = (ll)q * 10 + ch - '0', ch = getchar();
  return (ll)w * q;
}

int n, m;
ll mod;
ll a[100005], sum[400005], mul[400005], laz[400005];

void up(int i) { sum[i] = (sum[(i << 1)] + sum[(i << 1) | 1]) % mod; }
```

```
void pd(int i, int s, int t) {
  int l = (i << 1), r = (i << 1) | 1, mid = (s + t) >> 1;
  if (mul[i] != 1) {  // 懒标记传递，两个懒标记
    mul[l] *= mul[i];
    mul[l] %= mod;
    mul[r] *= mul[i];
    mul[r] %= mod;
    laz[l] *= mul[i];
    laz[l] %= mod;
    laz[r] *= mul[i];
    laz[r] %= mod;
    sum[l] *= mul[i];
    sum[l] %= mod;
    sum[r] *= mul[i];
    sum[r] %= mod;
    mul[i] = 1;
  }
  if (laz[i]) {  // 懒标记传递
    sum[l] += laz[i] * (mid - s + 1);
    sum[l] %= mod;
    sum[r] += laz[i] * (t - mid);
    sum[r] %= mod;
    laz[l] += laz[i];
    laz[l] %= mod;
    laz[r] += laz[i];
    laz[r] %= mod;
    laz[i] = 0;
  }
  return;
}

void build(int s, int t, int i) {
  mul[i] = 1;
  if (s == t) {
    sum[i] = a[s];
    return;
  }
  int mid = s + ((t - s) >> 1);
  build(s, mid, i << 1);  // 建树
  build(mid + 1, t, (i << 1) | 1);
  up(i);
}

void chen(int l, int r, int s, int t, int i, ll z) {
  int mid = s + ((t - s) >> 1);
  if (l <= s && t <= r) {
    mul[i] *= z;
    mul[i] %= mod;  // 这是取模的
    laz[i] *= z;
    laz[i] %= mod;  // 这是取模的
    sum[i] *= z;
    sum[i] %= mod;  // 这是取模的
    return;
  }
```

```
  pd(i, s, t);
  if (mid >= l) chen(l, r, s, mid, (i << 1), z);
  if (mid + 1 <= r) chen(l, r, mid + 1, t, (i << 1) | 1, z);
  up(i);
}

void add(int l, int r, int s, int t, int i, ll z) {
  int mid = s + ((t - s) >> 1);
  if (l <= s && t <= r) {
    sum[i] += z * (t - s + 1);
    sum[i] %= mod;  // 这是取模的
    laz[i] += z;
    laz[i] %= mod;  // 这是取模的
    return;
  }
  pd(i, s, t);
  if (mid >= l) add(l, r, s, mid, (i << 1), z);
  if (mid + 1 <= r) add(l, r, mid + 1, t, (i << 1) | 1, z);
  up(i);
}

ll getans(int l, int r, int s, int t,
          int i) {  // 得到答案，可以看下上面懒标记助于理解
  int mid = s + ((t - s) >> 1);
  ll tot = 0;
  if (l <= s && t <= r) return sum[i];
  pd(i, s, t);
  if (mid >= l) tot += getans(l, r, s, mid, (i << 1));
  tot %= mod;
  if (mid + 1 <= r) tot += getans(l, r, mid + 1, t, (i << 1) | 1);
  return tot % mod;
}

int main() {  // 读入
  int i, j, x, y, bh;
  ll z;
  n = read();
  m = read();
  mod = read();
  for (i = 1; i <= n; i++) a[i] = read();
  build(1, n, 1);  // 建树
  for (i = 1; i <= m; i++) {
    bh = read();
    if (bh == 1) {
      x = read();
      y = read();
      z = read();
      chen(x, y, 1, n, 1, z);
    } else if (bh == 2) {
      x = read();
      y = read();
      z = read();
      add(x, y, 1, n, 1, z);
    } else if (bh == 3) {
```

```
      x = read();
      y = read();
      printf("%lld\n", getans(x, y, 1, n, 1));
    }
  }
  return 0;
}
```

"HihoCoder 1078 线段树的区间修改[3]"

假设货架上从左到右摆放了 N 种商品，并且依次标号为 1 到 N，其中标号为 i 的商品的价格为 Pi。小 Hi 的每次操作分为两种可能，第一种是修改价格：小 Hi 给出一段区间 $[L, R]$ 和一个新的价格 $NewP$，所有标号在这段区间中的商品的价格都变成 $NewP$。第二种操作是询问：小 Hi 给出一段区间 $[L, R]$，而小 Ho 要做的便是计算出所有标号在这段区间中的商品的总价格，然后告诉小 Hi。

" 参考代码"

```
#include <iostream>

int n, a[100005], d[270000], b[270000];

void build(int l, int r, int p) {  // 建树
  if (l == r) {
    d[p] = a[l];
    return;
  }
  int m = l + ((r - l) >> 1);
  build(l, m, p << 1), build(m + 1, r, (p << 1) | 1);
  d[p] = d[p << 1] + d[(p << 1) | 1];
}

void update(int l, int r, int c, int s, int t,
            int p) {  // 更新，可以参考前面两个例题
  if (l <= s && t <= r) {
    d[p] = (t - s + 1) * c, b[p] = c;
    return;
  }
  int m = s + ((t - s) >> 1);
  if (b[p]) {
    d[p << 1] = b[p] * (m - s + 1), d[(p << 1) | 1] = b[p] * (t - m);
    b[p << 1] = b[(p << 1) | 1] = b[p];
    b[p] = 0;
  }
  if (l <= m) update(l, r, c, s, m, p << 1);
  if (r > m) update(l, r, c, m + 1, t, (p << 1) | 1);
  d[p] = d[p << 1] + d[(p << 1) | 1];
}

int getsum(int l, int r, int s, int t, int p) {  // 取得答案，和前面一样
  if (l <= s && t <= r) return d[p];
  int m = s + ((t - s) >> 1);
  if (b[p]) {
    d[p << 1] = b[p] * (m - s + 1), d[(p << 1) | 1] = b[p] * (t - m);
```

```
    b[p << 1] = b[(p << 1) | 1] = b[p];
    b[p] = 0;
  }
  int sum = 0;
  if (l <= m) sum = getsum(l, r, s, m, p << 1);
  if (r > m) sum += getsum(l, r, m + 1, t, (p << 1) | 1);
  return sum;
}

int main() {
  std::ios::sync_with_stdio(0);
  std::cin >> n;
  for (int i = 1; i <= n; i++) std::cin >> a[i];
  build(1, n, 1);
  int q, i1, i2, i3, i4;
  std::cin >> q;
  while (q--) {
    std::cin >> i1 >> i2 >> i3;
    if (i1 == 0)
      std::cout << getsum(i2, i3, 1, n, 1) << std::endl;
    else
      std::cin >> i4, update(i2, i3, i4, 1, n, 1);
  }
  return 0;
}
```

"2018 Multi-University Training Contest 5 Problem G. Glad You Came[4]"

" 解题思路"

维护一下每个区间的永久标记就可以了，最后在线段树上跑一边 DFS 统计结果即可。注意打标记的时候加个剪枝优化，否则会 TLE。

线段树合并

过程

顾名思义，线段树合并是指建立一棵新的线段树，这棵线段树的每个节点都是两棵原线段树对应节点合并后的结果。它常常被用于维护树上或是图上的信息。

显然，我们不可能真的每次建满一颗新的线段树，因此我们需要使用上文的动态开点线段树。

线段树合并的过程本质上相当暴力：

假设两颗线段树为 A 和 B，我们从 1 号节点开始递归合并。

递归到某个节点时，如果 A 树或者 B 树上的对应节点为空，直接返回另一个树上对应节点，这里运用了动态开点线段树的特性。

如果递归到叶子节点，我们合并两棵树上的对应节点。

最后，根据子节点更新当前节点并且返回。

" 线段树合并的复杂度"

显然，对于两颗满的线段树，合并操作的复杂度是 $O(n\log n)$ 的。但实际情况下使用的常常是权值线段树，总点数和 n 的规模相差并不大。并且合并时一般不会重复地合并某个线段树，所以我们最终增加的点数大致是 $n\log n$ 级别的。这样，总的复杂度就是 $O(n\log n)$ 级别的。当然，在一些情况下，可并堆可能是更好的选择。

实现

```
int merge(int a, int b, int l, int r) {
  if (!a) return b;
  if (!b) return a;
  if (l == r) {
    // do something...
    return a;
  }
  int mid = (l + r) >> 1;
  tr[a].l = merge(tr[a].l, tr[b].l, l, mid);
  tr[a].r = merge(tr[a].r, tr[b].r, mid + 1, r);
  pushup(a);
  return a;
}
```

例题

"luogu P4556 [Vani 有约会] 雨天的尾巴 / 【模板】线段树合并[5]"

"解题思路"

线段树合并模板题，用差分把树上修改转化为单点修改，然后向上 dfs 线段树合并统计答案即可。

"参考代码"

```
#include <bits/stdc++.h>
using namespace std;
int n, fa[100005][22], dep[100005], rt[100005];
int sum[5000005], cnt = 0, res[5000005], ls[5000005], rs[5000005];
int m, ans[100005];
vector<int> v[100005];

void update(int x) {
  if (sum[ls[x]] < sum[rs[x]]) {
    res[x] = res[rs[x]];
    sum[x] = sum[rs[x]];
  } else {
    res[x] = res[ls[x]];
    sum[x] = sum[ls[x]];
  }
}

int merge(int a, int b, int x, int y) {
  if (!a) return b;
  if (!b) return a;
  if (x == y) {
    sum[a] += sum[b];
    return a;
  }
  int mid = (x + y) >> 1;
  ls[a] = merge(ls[a], ls[b], x, mid);
  rs[a] = merge(rs[a], rs[b], mid + 1, y);
```

```
  update(a);
  return a;
}

int add(int id, int x, int y, int co, int val) {
  if (!id) id = ++cnt;
  if (x == y) {
    sum[id] += val;
    res[id] = co;
    return id;
  }
  int mid = (x + y) >> 1;
  if (co <= mid)
    ls[id] = add(ls[id], x, mid, co, val);
  else
    rs[id] = add(rs[id], mid + 1, y, co, val);
  update(id);
  return id;
}

void initlca(int x) {
  for (int i = 0; i <= 20; i++) fa[x][i + 1] = fa[fa[x][i]][i];
  for (int i : v[x]) {
    if (i == fa[x][0]) continue;
    dep[i] = dep[x] + 1;
    fa[i][0] = x;
    initlca(i);
  }
}

int lca(int x, int y) {
  if (dep[x] < dep[y]) swap(x, y);
  for (int d = dep[x] - dep[y], i = 0; d; d >>= 1, i++)
    if (d & 1) x = fa[x][i];
  if (x == y) return x;
  for (int i = 20; i >= 0; i--)
    if (fa[x][i] != fa[y][i]) x = fa[x][i], y = fa[y][i];
  return fa[x][0];
}

void cacl(int x) {
  for (int i : v[x]) {
    if (i == fa[x][0]) continue;
    cacl(i);
    rt[x] = merge(rt[x], rt[i], 1, 100000);
  }
  ans[x] = res[rt[x]];
  if (sum[rt[x]] == 0) ans[x] = 0;
}

int main() {
  ios::sync_with_stdio(0);
  cin >> n >> m;
  for (int i = 0; i < n - 1; i++) {
```

```
    int a, b;
    cin >> a >> b;
    v[a].push_back(b);
    v[b].push_back(a);
  }
  initlca(1);
  for (int i = 0; i < m; i++) {
    int a, b, c;
    cin >> a >> b >> c;
    rt[a] = add(rt[a], 1, 100000, c, 1);
    rt[b] = add(rt[b], 1, 100000, c, 1);
    int t = lca(a, b);
    rt[t] = add(rt[t], 1, 100000, c, -1);
    rt[fa[t][0]] = add(rt[fa[t][0]], 1, 100000, c, -1);
  }
  cacl(1);
  for (int i = 1; i <= n; i++) cout << ans[i] << endl;
  return 0;
}
```

线段树分裂

过程

线段树分裂实质上是线段树合并的逆过程。线段树分裂只适用于有序的序列，无序的序列是没有意义的，常用在动态开点的权值线段树。

注意当分裂和合并都存在时，我们在合并的时候必须回收节点，以避免分裂时会可能出现节点重复占用的问题。

从一颗区间为 $[1, N]$ 的线段树中分裂出 $[l, r]$，建一颗新的树：

从 1 号结点开始递归分裂，当节点不存在或者代表的区间 $[s, t]$ 与 $[l, r]$ 没有交集时直接回溯。

当 $[s, t]$ 与 $[l, r]$ 有交集时需要开一个新结点。

当 $[s, t]$ 包含于 $[l, r]$ 时，需要将当前结点直接接到新的树下面，并把旧边断开。

" 线段树分裂的复杂度"

可以发现被断开的边最多只会有 $\log n$ 条，所以最终每次分裂的时间复杂度就是 $O(\log\ n)$，相当于区间查询的复杂度。

实现

```
void split(int &p, int &q, int s, int t, int l, int r) {
  if (t < l || r < s) return;
  if (!p) return;
  if (l <= s && t <= r) {
    q = p;
    p = 0;
    return;
  }
  if (!q) q = New();
  int m = s + t >> 1;
  if (l <= m) split(ls[p], ls[q], s, m, l, r);
  if (m < r) split(rs[p], rs[q], m + 1, t, l, r);
  push_up(p);
```

```
  push_up(q);
}
```

例题

"P5494【模板】线段树分裂[6]"

"解题思路"

线段树分裂模板题，将 $[x, y]$ 分裂出来。

- 将 t 树合并入 p 树：单次合并即可。
- p 树中插入 x 个 q：单点修改。
- 查询 $[x, y]$ 中数的个数：区间求和。
- 查询第 k 小。

"参考代码"

```cpp
#include <iostream>
using namespace std;
const int N = 2e5 + 10;
int n, m;
int idx = 1;
long long sum[N << 5];
int ls[N << 5], rs[N << 5], root[N << 2], rub[N << 5], cnt, tot;

//内存分配与回收
int New() { return cnt ? rub[cnt--] : ++tot; }

void Del(int &p) {
  ls[p] = rs[p] = sum[p] = 0;
  rub[++cnt] = p;
  p = 0;
}

void push_up(int p) { sum[p] = sum[ls[p]] + sum[rs[p]]; }

void build(int s, int t, int &p) {
  if (!p) p = New();
  if (s == t) {
    cin >> sum[p];
    return;
  }
  int m = s + t >> 1;
  build(s, m, ls[p]);
  build(m + 1, t, rs[p]);
  push_up(p);
}

//单点修改
void update(int x, int c, int s, int t, int &p) {
  if (!p) p = New();
```

```
  if (s == t) {
    sum[p] += c;
    return;
  }
  int m = s + t >> 1;
  if (x <= m)
    update(x, c, s, m, ls[p]);
  else
    update(x, c, m + 1, t, rs[p]);
  push_up(p);
}

//合并
int merge(int p, int q, int s, int t) {
  if (!p || !q) return p + q;
  if (s == t) {
    sum[p] += sum[q];
    Del(q);
    return p;
  }
  int m = s + t >> 1;
  ls[p] = merge(ls[p], ls[q], s, m);
  rs[p] = merge(rs[p], rs[q], m + 1, t);
  push_up(p);
  Del(q);
  return p;
}

//分裂
void split(int &p, int &q, int s, int t, int l, int r) {
  if (t < l || r < s) return;
  if (!p) return;
  if (l <= s && t <= r) {
    q = p;
    p = 0;
    return;
  }
  if (!q) q = New();
  int m = s + t >> 1;
  if (l <= m) split(ls[p], ls[q], s, m, l, r);
  if (m < r) split(rs[p], rs[q], m + 1, t, l, r);
  push_up(p);
  push_up(q);
}

long long query(int l, int r, int s, int t, int p) {
  if (!p) return 0;
  if (l <= s && t <= r) return sum[p];
  int m = s + t >> 1;
  long long ans = 0;
  if (l <= m) ans += query(l, r, s, m, ls[p]);
  if (m < r) ans += query(l, r, m + 1, t, rs[p]);
  return ans;
}
```

```
int kth(int s, int t, int k, int p) {
  if (s == t) return s;
  int m = s + t >> 1;
  long long left = sum[ls[p]];
  if (k <= left)
    return kth(s, m, k, ls[p]);
  else
    return kth(m + 1, t, k - left, rs[p]);
}

int main() {
  cin >> n >> m;
  build(1, n, root[1]);
  while (m--) {
    int op;
    cin >> op;
    if (!op) {
      int p, x, y;
      cin >> p >> x >> y;
      split(root[p], root[++idx], 1, n, x, y);
    } else if (op == 1) {
      int p, t;
      cin >> p >> t;
      root[p] = merge(root[p], root[t], 1, n);
    } else if (op == 2) {
      int p, x, q;
      cin >> p >> x >> q;
      update(q, x, 1, n, root[p]);
    } else if (op == 3) {
      int p, x, y;
      cin >> p >> x >> y;
      cout << query(x, y, 1, n, root[p]) << endl;
    } else {
      int p, k;
      cin >> p >> k;
      if (sum[root[p]] < k)
        cout << -1 << endl;
      else
        cout << kth(1, n, k, root[p]) << endl;
    }
  }
}
```

线段树优化建图

在建图连边的过程中，我们有时会碰到这种题目，一个点向一段连续的区间中的点连边或者一个连续的区间向一个点连边，如果我们真的一条一条连过去，那一旦点的数量多了复杂度就爆炸了，这里就需要用线段树的区间性质来优化我们的建图了。

下面是一个线段树。

每个节点都代表了一个区间，假设我们要向区间 $[2,4]$ 连边。

在一些题目中，还会出现一个区间连向一个点的情况，则我们将上面第一张图的有向边全部反过来即可，上面的树叫做入树，下面这个叫做出树。

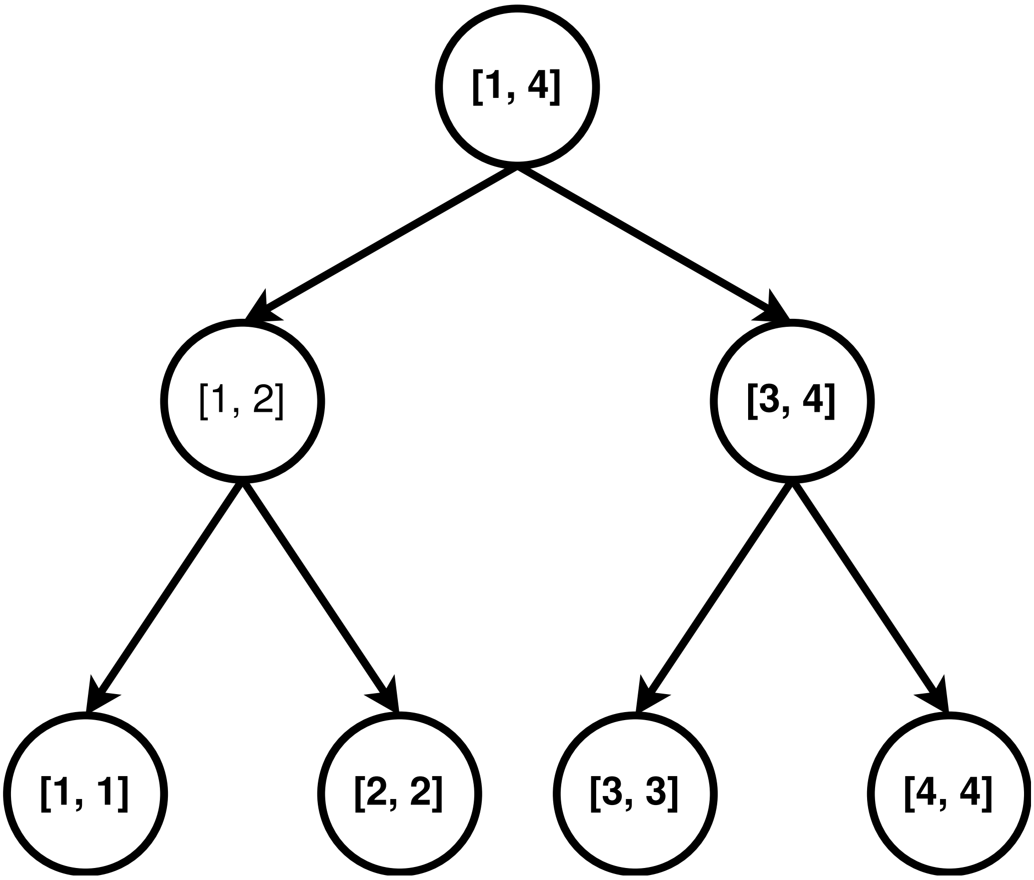


图 10.41

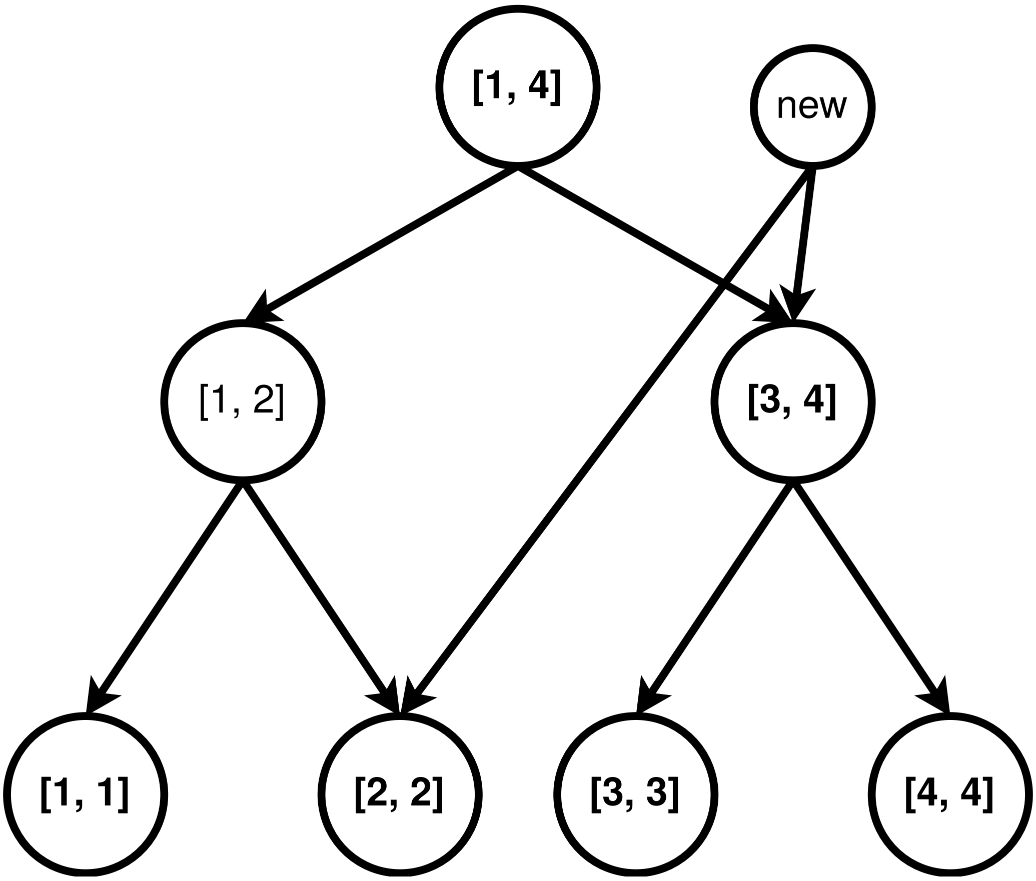


图 10.42

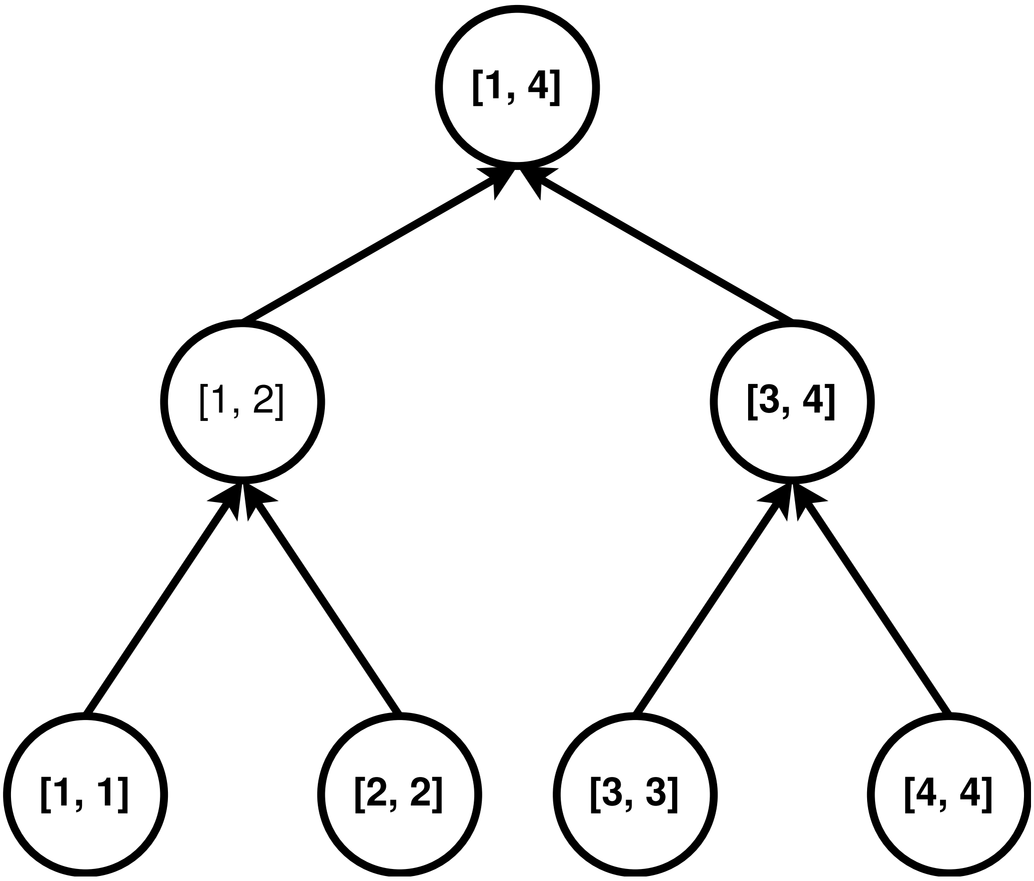


图 10.43

"Legacy[7]"

题目大意：有 n 个点、q 次操作。每一种操作为以下三种类型中的一种：

- 操作一：连一条 $u \to v$ 的有向边，权值为 w。
- 操作二：对于所有 $i \in [l, r]$ 连一条 $u \to i$ 的有向边，权值为 w。
- 操作三：对于所有 $i \in [l, r]$ 连一条 $i \to u$ 的有向边，权值为 w。

求从点 s 到其他点的最短路。

$1 \le n, q \le 10^5, 1 \le w \le 10^9$。

" 参考代码"

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;

const int N = 1e5 + 5;

using pil = pair<int, ll>;
using pli = pair<ll, int>;

int n, q, s, tot, rt1, rt2;
int pos[N];
ll dis[N << 3];
```

```
vector<pil> e[N << 3];
bitset<(N << 3)> vis;

struct seg {
  int l, r, lson, rson;
} t[N << 3];

inline int ls(int u) {  // 左儿子
  return t[u].lson;
}

inline int rs(int u) {  // 右儿子
  return t[u].rson;
}

void build(int &u, int l, int r) {  // 动态开点建造入树
  u = ++tot;
  t[u] = seg{l, r};
  if (l == r) {
    pos[l] = u;
    return;
  }
  int mid = (l + r) >> 1;
  build(t[u].lson, l, mid);
  build(t[u].rson, mid + 1, r);
  e[u].emplace_back(ls(u), 0);
  e[u].emplace_back(rs(u), 0);
}

void build2(int &u, int l, int r) {  // 动态开点建造出树
  if (l == r) {
    u = pos[l];
    return;
  }
  u = ++tot;
  t[u] = seg{l, r};
  int mid = (l + r) >> 1;
  build2(t[u].lson, l, mid);
  build2(t[u].rson, mid + 1, r);
  e[ls(u)].emplace_back(u, 0);
  e[rs(u)].emplace_back(u, 0);
}

void add1(int u, int lr, int rr, int v, ll w) {  // 点向区间连边
  if (lr <= t[u].l && t[u].r <= rr) {
    e[v].emplace_back(u, w);
    return;
  }
  int mid = (t[u].l + t[u].r) >> 1;
  if (lr <= mid) {
    add1(ls(u), lr, rr, v, w);
  }
  if (rr > mid) {
    add1(rs(u), lr, rr, v, w);
```

```
    }
}

void add2(int u, int lr, int rr, int v, ll w) {  // 区间向点连边
  if (lr <= t[u].l && t[u].r <= rr) {
    e[u].emplace_back(v, w);
    return;
  }
  int mid = (t[u].l + t[u].r) >> 1;
  if (lr <= mid) {
    add2(ls(u), lr, rr, v, w);
  }
  if (rr > mid) {
    add2(rs(u), lr, rr, v, w);
  }
}

void dij(int S) {
  priority_queue<pli, vector<pli>, greater<pli> > q;
  int tot = (n << 2);
  for (int i = 1; i <= tot; ++i) {
    dis[i] = 1e18;
  }
  dis[S] = 0;
  q.emplace(dis[S], S);
  while (!q.empty()) {
    pli fr = q.top();
    q.pop();
    int u = fr.second;
    if (vis[u]) continue;
    for (pil it : e[u]) {
      int v = it.first;
      ll w = it.second;
      if (dis[v] > dis[u] + w) {
        dis[v] = dis[u] + w;
        q.emplace(dis[v], v);
      }
    }
  }
}

int main() {
  scanf("%d%d%d", &n, &q, &s);
  build(rt1, 1, n);
  build2(rt2, 1, n);
  for (int i = 1, op, u; i <= q; ++i) {
    scanf("%d%d", &op, &u);
    if (op == 1) {
      int v;
      ll w;
      scanf("%d%lld", &v, &w);
      e[pos[u]].emplace_back(pos[v], w);
    } else if (op == 2) {
      int l, r;
```

```
      ll w;
      scanf("%d%d%lld", &l, &r, &w);
      add1(rt1, l, r, pos[u], w);
    } else {
      int l, r;
      ll w;
      scanf("%d%d%lld", &l, &r, &w);
      add2(rt2, l, r, pos[u], w);
    }
  }
  dij(pos[s]);
  for (int i = 1; i <= n; ++i) {
    if (dis[pos[i]] == 1e18) {
      printf("-1 ");
    } else {
      printf("%lld ", dis[pos[i]]);
    }
  }
  return 0;
}
```

拓展 - 猫树

众所周知线段树可以支持高速查询某一段区间的信息和，比如区间最大子段和，区间和，区间矩阵的连乘积等等。

但是有一个问题在于普通线段树的区间询问在某些毒瘤的眼里可能还是有些慢了。

简单来说就是线段树建树的时候需要做 $O(n)$ 次合并操作，而每一次区间询问需要做 $O(\log n)$ 次合并操作，询问区间和这种东西的时候还可以忍受，但是当我们需要询问区间线性基这种合并复杂度高达 $O(\log^2 w)$ 的信息的话，此时就算是做 $O(\log n)$ 次合并有些时候在时间上也是不可接受的。

而所谓「猫树」就是一种不支持修改，仅仅支持快速区间询问的一种静态线段树。

构造一棵这样的静态线段树需要 $O(n\log n)$ 次合并操作，但是此时的查询复杂度被加速至 $O(1)$ 次合并操作。

在处理线性基这样特殊的信息的时候甚至可以将复杂度降至 $O(n\log^2 w)$。

原理

在查询 $[l,r]$ 这段区间的信息和的时候，将线段树树上代表 $[l,l]$ 的节点和代表 $[r,r]$ 这段区间的节点在线段树上的 LCA 求出来，设这个节点 p 代表的区间为 $[L,R]$，我们会发现一些非常有趣的性质：

1. $[L,R]$ 这个区间一定包含 $[l,r]$。显然，因为它既是 l 的祖先又是 r 的祖先。
2. $[l,r]$ 这个区间一定跨越 $[L,R]$ 的中点。由于 p 是 l 和 r 的 LCA，这意味着 p 的左儿子是 l 的祖先而不是 r 的祖先，p 的右儿子是 r 的祖先而不是 l 的祖先。因此，l 一定在 $[L,mid]$ 这个区间内，r 一定在 $(mid,R]$ 这个区间内。

有了这两个性质，我们就可以将询问的复杂度降至 $O(1)$ 了。

实现

具体来讲我们建树的时候对于线段树树上的一个节点，设它代表的区间为 $(l,r]$。

不同于传统线段树在这个节点里只保留 $[l,r]$ 的和，我们在这个节点里面额外保存 $(l,mid]$ 的后缀和数组和 $(mid,r]$ 的前缀和数组。

这样的话建树的复杂度为 $T(n)=2T(n/2)+O(n)=O(n\log n)$ 同理空间复杂度也从原来的 $O(n)$ 变成了 $O(n\log n)$。

下面是最关键的询问了。

如果我们询问的区间是 $[l,r]$ 那么我们把代表 $[l,l]$ 的节点和代表 $[r,r]$ 的节点的 LCA 求出来，记为 p。

根据刚才的两个性质，l,r 在 p 所包含的区间之内并且一定跨越了 p 的中点。

这意味这一个非常关键的事实是我们可以使用 p 里面的前缀和数组和后缀和数组，将 $[l,r]$ 拆成 $[l,mid]+(mid,r]$ 从而拼出来 $[l,r]$ 这个区间。

而这个过程仅仅需要 $O(1)$ 次合并操作!

不过我们好像忽略了点什么?

似乎求 LCA 的复杂度似乎还不是 $O(1)$，暴力求是 $O(\log n)$ 的，倍增法则是 $O(\log\log n)$ 的，转 ST 表的代价又太大⋯⋯

堆式建树

具体来将我们将这个序列补成 2 的整次幂，然后建线段树。

此时我们发现线段树上两个节点的 LCA 编号，就是两个节点二进制编号的最长公共前缀 LCP。

稍作思考即可发现发现在 x 和 y 的二进制下 `lcp(x,y)=x>>log[x^y]`。

所以我们预处理一个 `log` 数组即可轻松完成求 LCA 的工作。

这样我们就构建了一个猫树。

由于建树的时候涉及到求前缀和和求后缀和，所以对于线性基这种虽然合并是 $O(\log^2 w)$ 但是求前缀和却是 $O(n\log n)$ 的信息，使用猫树可以将静态区间线性基从 $O(n\log^2 w+m\log^2 w\log n)$ 优化至 $O(n\log n\log w+m\log^2 w)$ 的复杂度。

参考

- immortalCO 大爷的博客[8]
- [Kle77][9] V. Klee, "Can the Measure of be Computed in Less than O (n log n) Steps?," Am. Math. Mon., vol. 84, no. 4, pp. 284–285, Apr. 1977.
- [BeW80][10] Bentley and Wood, "An Optimal Worst Case Algorithm for Reporting Intersections of Rectangles," IEEE Trans. Comput., vol. C–29, no. 7, pp. 571–577, Jul. 1980.

参考资料与注释

[1] luogu P3372【模板】线段树 1

[2] luogu P3373【模板】线段树 2

[3] HihoCoder 1078 线段树的区间修改

[4] 2018 Multi-University Training Contest 5 Problem G. Glad You Came

[5] luogu P4556 [Vani 有约会] 雨天的尾巴/【模板】线段树合并

[6] P5494【模板】线段树分裂

[7] Legacy

[8] immortalCO 大爷的博客

[9] [Kle77]

[10] [BeW80]

10.14 李超线段树

引入

"洛谷 4097 [HEOI2013]Segment[1]"

要求在平面直角坐标系下维护两个操作（强制在线）：

1. 在平面上加入一条线段。记第 i 条被插入的线段的标号为 i，该线段的两个端点分别为 (x_0, y_0)，(x_1, y_1)。
2. 给定一个数 k，询问与直线 $x = k$ 相交的线段中，交点纵坐标最大的线段的编号（若有多条线段与查询直线的交点纵坐标都是最大的，则输出编号最小的线段）。特别地，若不存在线段与给定直线相交，输出 0。

数据满足：操作总数 $1 \le n \le 10^5$，$1 \le k, x_0, x_1 \le 39989$，$1 \le y_0, y_1 \le 10^9$。

我们发现，传统的线段树无法很好地维护这样的信息。这种情况下，**李超线段树**便应运而生。

过程

我们可以把任务转化为维护如下操作：

- 加入一个一次函数，定义域为 $[l, r]$；
- 给定 k，求定义域包含 k 的所有一次函数中，在 $x = k$ 处取值最大的那个，如果有多个函数取值相同，选编号最小的。

" 注意"

当线段垂直于 x 轴时，会出现除以零的情况。假设线段两端点分别为 (x, y_0) 和 (x, y_1)，$y_0 < y_1$，则插入定义域为 $[x, x]$ 的一次函数 $f(x) = 0 \cdot x + y_1$。

看到区间修改，我们按照线段树解决区间问题的常见方法，给每个节点一个懒标记。每个节点 i 的懒标记都是一条线段，记为 l_i，表示要用 l_i 更新该节点所表示的整个区间。

现在我们需要插入一条线段 f，考虑某个被新线段 f 完整覆盖的线段树区间。若该区间无标记，直接打上用该线段更新的标记。

如果该区间已经有标记了，由于标记难以合并，只能把标记下传。但是子节点也有自己的标记，也可能产生冲突，所以我们要递归下传标记。

如图，按新线段 f 取值是否大于原标记 g，我们可以把当前区间分为两个子区间。其中**肯定有一个子区间被左区间或右区间完全包含**，也就是说，在两条线段中，肯定有一条线段，只可能成为左区间的答案，或者只可能成为右区间的答案。我们用这条线段递归更新对应子树，用另一条线段作为懒标记更新整个区间，这就保证了递归下传的复杂度。当一条线段只可能成为左或右区间的答案时，才会被下传，所以不用担心漏掉某些线段。

具体来说，设当前区间的中点为 m，我们拿新线段 f 在中点处的值与原最优线段 g 在中点处的值作比较。

如果新线段 f 更优，则将 f 和 g 交换。那么现在考虑在中点处 f 不如 g 优的情况：

1. 若在左端点处 f 更优，那么 f 和 g 必然在左半区间中产生了交点，f 只有在左区间才可能优于 g，递归到左儿子中进行下传；
2. 若在右端点处 f 更优，那么 f 和 g 必然在右半区间中产生了交点，f 只有在右区间才可能优于 g，递归到右儿子中进行下传；

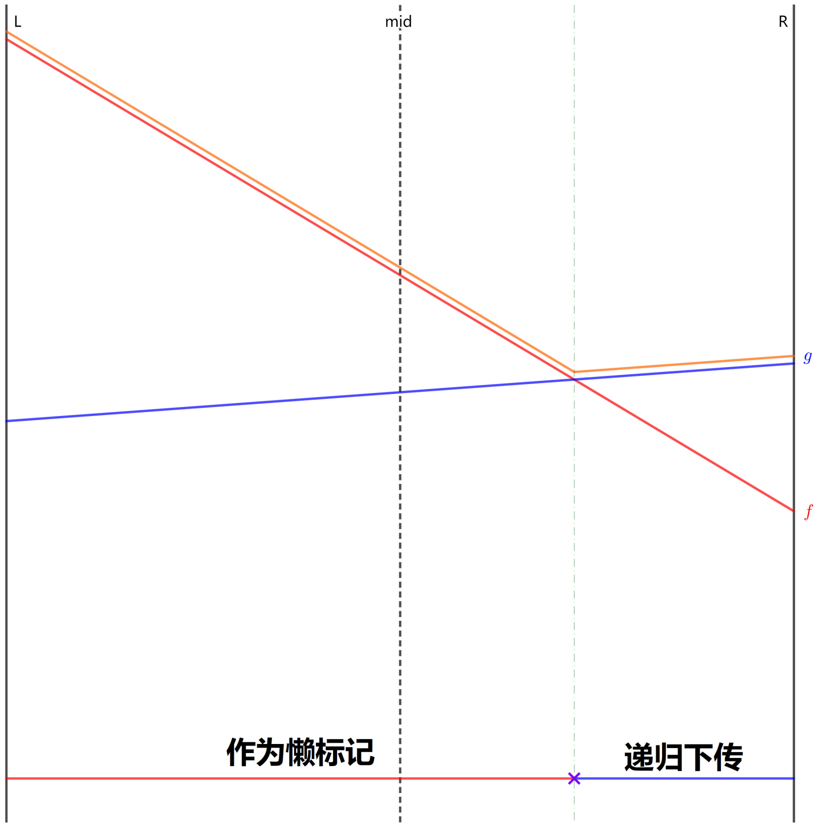


图 10.44

3. 若在左右端点处 g 都更优，那么 f 不可能成为答案，不需要继续下传。

除了这两种情况之外，还有一种情况是 f 和 g 刚好交于中点，在程序实现时可以归入中点处 f 不如 g 优的情况，结果会往 f 更优的一个端点进行递归下传。

最后将 g 作为当前区间的懒标记。

下传标记：

" 实现"

```
const double eps = 1e-9;

int cmp(double x, double y) {  // 因为用到了浮点数，所以会有精度误差
  if (x - y > eps) return 1;
  if (y - x > eps) return -1;
  return 0;
}

//...

void upd(int root, int cl, int cr, int u) {  // 对线段完全覆盖到的区间进行修改
  int &v = s[root], mid = (cl + cr) >> 1;
  int bmid = cmp(calc(u, mid), calc(v, mid));
  if (bmid == 1 || (!bmid && u < v))  // 在此题中记得判线段编号
    swap(u, v);
  int bl = cmp(calc(u, cl), calc(v, cl)), br = cmp(calc(u, cr), calc(v, cr));
  if (bl == 1 || (!bl && u < v)) upd(root << 1, cl, mid, u);
  if (br == 1 || (!br && u < v)) upd(root << 1 | 1, mid + 1, cr, u);
```

```
    // 上面两个 if 的条件最多只有一个成立，这保证了李超树的时间复杂度
}
```

拆分线段：

" 实现"

```
void update(int root, int cl, int cr, int l, int r,
            int u) {  // 定位插入线段完全覆盖到的区间
  if (l <= cl && cr <= r) {
    upd(root, cl, cr, u);  // 完全覆盖当前区间，更新当前区间的标记
    return;
  }
  int mid = (cl + cr) >> 1;
  if (l <= mid) update(root << 1, cl, mid, l, r, u);  // 递归拆分区间
  if (mid < r) update(root << 1 | 1, mid + 1, cr, l, r, u);
}
```

注意懒标记并不等价于在区间中点处取值最大的线段。

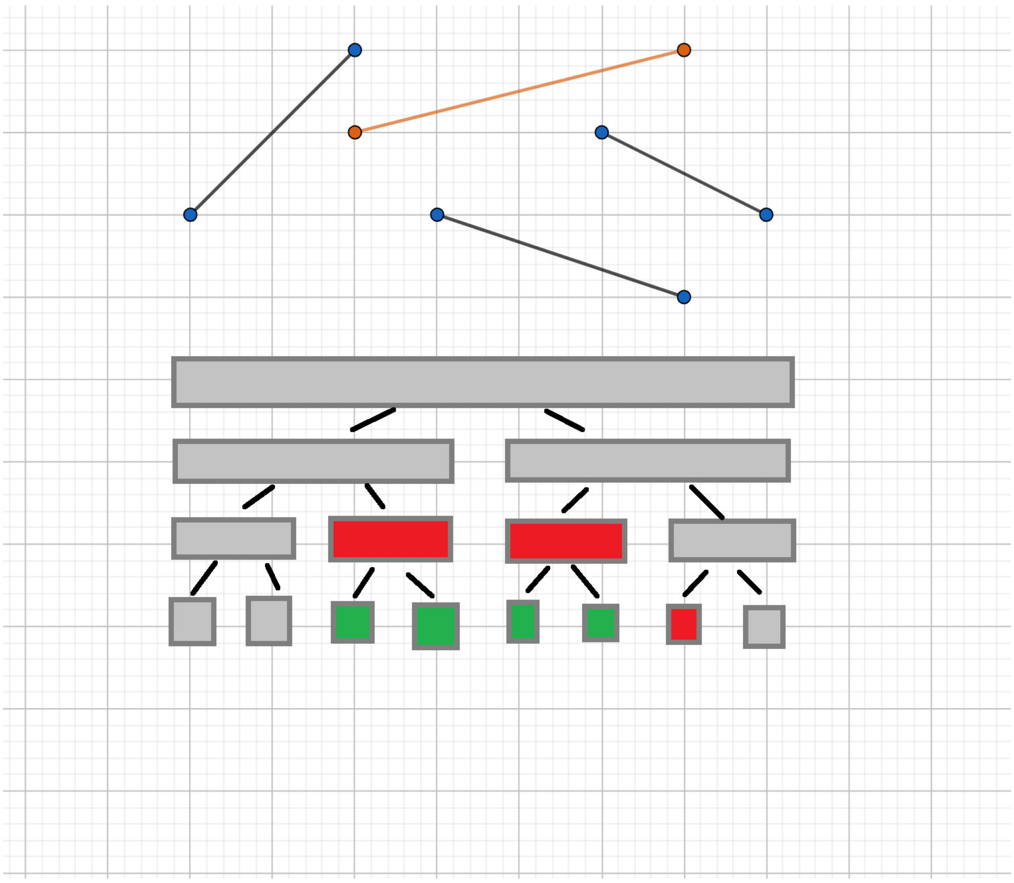


图 10.45

如图，加入黄色线段后，只有红色节点的标记被更新，而绿色节点的标记还未被改变。但在第二、三、四个绿色区间的中点处显然黄色线段取值最大。

查询时，我们可以利用标记永久化思想，在包含 x 的所有线段树区间（不超过 $O(\log n)$ 个）的标记线段中，比较得出最终答案。

查询：

"实现"

```
pdi query(int root, int l, int r, int d) {  // 查询
  if (r < d || d < l) return {0, 0};
  int mid = (l + r) >> 1;
  double res = calc(s[root], d);
  if (l == r) return {res, s[root]};
  return pmax({res, s[root]}, pmax(query(root << 1, l, mid, d),
                                    query(root << 1 | 1, mid + 1, r, d)));
}
```

根据上面的描述，查询过程的时间复杂度显然为 $O(\log n)$，而插入过程中，我们需要将原线段拆分到 $O(\log n)$ 个区间中，对于每个区间，我们又需要花费 $O(\log n)$ 的时间递归下传，从而插入过程的时间复杂度为 $O(\log^2 n)$。

"[HEOI2013]Segment[1] 参考代码"

```
#include <iostream>
#include <string>
#define MOD1 39989
#define MOD2 1000000000
#define MAXT 40000
using namespace std;
typedef pair<double, int> pdi;

const double eps = 1e-9;

int cmp(double x, double y) {
  if (x - y > eps) return 1;
  if (y - x > eps) return -1;
  return 0;
}

struct line {
  double k, b;
} p[100005];

int s[160005];
int cnt;

double calc(int id, int d) { return p[id].b + p[id].k * d; }

void add(int x0, int y0, int x1, int y1) {
  cnt++;
  if (x0 == x1)  // 特判直线斜率不存在的情况
    p[cnt].k = 0, p[cnt].b = max(y0, y1);
  else
    p[cnt].k = 1.0 * (y1 - y0) / (x1 - x0), p[cnt].b = y0 - p[cnt].k * x0;
}

void upd(int root, int cl, int cr, int u) {  // 对线段完全覆盖到的区间进行修改
  int &v = s[root], mid = (cl + cr) >> 1;
  int bmid = cmp(calc(u, mid), calc(v, mid));
  if (bmid == 1 || (!bmid && u < v)) swap(u, v);
```

```
  int bl = cmp(calc(u, cl), calc(v, cl)), br = cmp(calc(u, cr), calc(v, cr));
  if (bl == 1 || (!bl && u < v)) upd(root << 1, cl, mid, u);
  if (br == 1 || (!br && u < v)) upd(root << 1 | 1, mid + 1, cr, u);
}

void update(int root, int cl, int cr, int l, int r,
            int u) {  // 定位插入线段完全覆盖到的区间
  if (l <= cl && cr <= r) {
    upd(root, cl, cr, u);
    return;
  }
  int mid = (cl + cr) >> 1;
  if (l <= mid) update(root << 1, cl, mid, l, r, u);
  if (mid < r) update(root << 1 | 1, mid + 1, cr, l, r, u);
}

pdi pmax(pdi x, pdi y) {  // pair max 函数
  if (cmp(x.first, y.first) == -1)
    return y;
  else if (cmp(x.first, y.first) == 1)
    return x;
  else
    return x.second < y.second ? x : y;
}

pdi query(int root, int l, int r, int d) {  // 查询
  if (r < d || d < l) return {0, 0};
  int mid = (l + r) >> 1;
  double res = calc(s[root], d);
  if (l == r) return {res, s[root]};
  return pmax({res, s[root]}, pmax(query(root << 1, l, mid, d),
                                   query(root << 1 | 1, mid + 1, r, d)));
}

int main() {
  ios::sync_with_stdio(false);
  int n, lastans = 0;
  cin >> n;
  while (n--) {
    int op;
    cin >> op;
    if (op == 1) {
      int x0, y0, x1, y1;
      cin >> x0 >> y0 >> x1 >> y1;
      x0 = (x0 + lastans - 1 + MOD1) % MOD1 + 1,
      x1 = (x1 + lastans - 1 + MOD1) % MOD1 + 1;
      y0 = (y0 + lastans - 1 + MOD2) % MOD2 + 1,
      y1 = (y1 + lastans - 1 + MOD2) % MOD2 + 1;
      if (x0 > x1) swap(x0, x1), swap(y0, y1);
      add(x0, y0, x1, y1);
      update(1, 1, MOD1, x0, x1, cnt);
    } else {
      int x;
      cin >> x;
```

```
      x = (x + lastans - 1 + MOD1) % MOD1 + 1;
      cout << (lastans = query(1, 1, MOD1, x).second) << endl;
    }
  }
  return 0;
}
```

参考资料与注释

[1] 洛谷 4097 [HEOI2013]Segment [1-1] [1-2]

10.15 区间最值操作 & 区间历史最值

本文讲解吉老师在 2016 年国家集训队论文中提到的线段树处理历史区间最值的问题。

区间最值

笼统地说，区间最值操作指，将区间 $[l,r]$ 的数全部对 x 取 $\max$ 或 $\min$，即 $a_i=\max(a_i,x)$ 或者 $a_i=\min(a_i,x)$。给一道例题吧。

HDU5306 Gorgeous Sequence

> 维护一个序列 a：
>
> 1. `0 l r t` $\forall l\le i\le r,\ a_i=\min(a_i,t)$。
> 2. `1 l r` 输出区间 $[l,r]$ 中的最大值。
> 3. `2 l r` 输出区间和。
>
> 多组数据，$T\le 100, n\le 10^6, \sum m\le 10^6$。

区间取 $\min$，意味着只对那些大于 t 的数有更改。因此这个操作的对象不再是整个区间，而是「这个区间中大于 t 的数」。于是我们可以有这样的思路：每个结点维护该区间的最大值 Max、次大值 Se、区间和 Sum 以及最大值的个数 Cnt。接下来我们考虑区间对 t 取 $\min$ 的操作。

1. 如果 $Max\le t$，显然这个 t 是没有意义的，直接返回；
2. 如果 $Se<t\le Max$，那么这个 t 就能更新当前区间中的最大值。于是我们让区间和加上 $Cnt(t-Max)$，然后更新 Max 为 t，并打一个标记。
3. 如果 $t\le Se$，那么这时你发现你不知道有多少个数涉及到更新的问题。于是我们的策略就是，暴力递归向下操作。然后上传信息。

这个算法的复杂度如何？使用势能分析法可以得到复杂度是 $O(m\log n)$ 的。具体分析过程见论文。

```cpp
#include <algorithm>
#include <cctype>
#include <cstdio>
using namespace std;
const int N = 1e6 + 6;

char nc() {
  static char buf[1000000], *p = buf, *q = buf;
```

```
  return p == q && (q = (p = buf) + fread(buf, 1, 1000000, stdin), p == q)
             ? EOF
             : *p++;
}

int rd() {
  int res = 0;
  char c = nc();
  while (!isdigit(c)) c = nc();
  while (isdigit(c)) res = res * 10 + c - '0', c = nc();
  return res;
}

int t, n, m;
int a[N];
int mx[N << 2], se[N << 2], cn[N << 2], tag[N << 2];
long long sum[N << 2];

void pushup(int u) {  // 向上更新标记
  const int ls = u << 1, rs = u << 1 | 1;
  sum[u] = sum[ls] + sum[rs];
  if (mx[ls] == mx[rs]) {
    mx[u] = mx[rs];
    se[u] = max(se[ls], se[rs]);
    cn[u] = cn[ls] + cn[rs];
  } else if (mx[ls] > mx[rs]) {
    mx[u] = mx[ls];
    se[u] = max(se[ls], mx[rs]);
    cn[u] = cn[ls];
  } else {
    mx[u] = mx[rs];
    se[u] = max(mx[ls], se[rs]);
    cn[u] = cn[rs];
  }
}

void pushtag(int u, int tg) {  // 单纯地打标记，不暴搜
  if (mx[u] <= tg) return;
  sum[u] += (1ll * tg - mx[u]) * cn[u];
  mx[u] = tag[u] = tg;
}

void pushdown(int u) {  // 下传标记
  if (tag[u] == -1) return;
  pushtag(u << 1, tag[u]), pushtag(u << 1 | 1, tag[u]);
  tag[u] = -1;
}

void build(int u = 1, int l = 1, int r = n) {  // 建树
  tag[u] = -1;
  if (l == r) {
    sum[u] = mx[u] = a[l], se[u] = -1, cn[u] = 1;
    return;
  }
```

```cpp
  int mid = (l + r) >> 1;
  build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
  pushup(u);
}

void modify_min(int L, int R, int v, int u = 1, int l = 1, int r = n) {
  if (mx[u] <= v) return;
  if (L <= l && r <= R && se[u] < v) return pushtag(u, v);
  int mid = (l + r) >> 1;
  pushdown(u);
  if (L <= mid) modify_min(L, R, v, u << 1, l, mid);
  if (mid < R) modify_min(L, R, v, u << 1 | 1, mid + 1, r);
  pushup(u);
}

int query_max(int L, int R, int u = 1, int l = 1, int r = n) {  // 查询最值
  if (L <= l && r <= R) return mx[u];
  int mid = (l + r) >> 1, r1 = -1, r2 = -1;
  pushdown(u);
  if (L <= mid) r1 = query_max(L, R, u << 1, l, mid);
  if (mid < R) r2 = query_max(L, R, u << 1 | 1, mid + 1, r);
  return max(r1, r2);
}

long long query_sum(int L, int R, int u = 1, int l = 1, int r = n) {  // 数值
  if (L <= l && r <= R) return sum[u];
  int mid = (l + r) >> 1;
  long long res = 0;
  pushdown(u);
  if (L <= mid) res += query_sum(L, R, u << 1, l, mid);
  if (mid < R) res += query_sum(L, R, u << 1 | 1, mid + 1, r);
  return res;
}

void go() {  // 根据题意
  n = rd(), m = rd();
  for (int i = 1; i <= n; i++) a[i] = rd();
  build();
  for (int i = 1; i <= m; i++) {
    int op, x, y, z;
    op = rd(), x = rd(), y = rd();
    if (op == 0)
      z = rd(), modify_min(x, y, z);
    else if (op == 1)
      printf("%d\n", query_max(x, y));
    else
      printf("%lld\n", query_sum(x, y));
  }
}

signed main() {
  t = rd();
  while (t--) go();
  return 0;
}
```

BZOJ4695 最假女选手

> 长度为 n 的序列，支持区间加 x/区间对 x 取 max/区间对 x 取 min/求区间和/求区间最大值/求区间最小值。
>
> $N, M \le 5\times 10^5, |A_i| \le 10^8$。

同样的方法，我们维护最大、次大、最大个数、最小、次小、最小个数、区间和。除了这些信息，我们还需要维护区间 max、区间 min、区间加的标记。相比上一道题，这就涉及到标记下传的顺序问题了。我们采用这样的策略：

1. 我们认为区间加的标记是最优先的，其余两种标记地位平等。
2. 对一个结点加上一个 v 标记，除了用 v 更新卫星信息和当前结点的区间加标记外，我们用这个 v 更新区间 max 和区间 min 的标记。
3. 对一个结点取 v 的 min（这里忽略暴搜的过程，假定标记满足添加的条件），除了更新卫星信息，我们要与区间 max 的标记做比较。如果 v 小于区间 max 的标记，则所有的数最后都会变成 v，那么把区间 max 的标记也变成 v。否则不管。
4. 区间取 v 的 max 同理。

另外，BZOJ 这道题卡常⋯⋯多数组线段树的常数比结构体线段树的常数大⋯⋯在维护信息的时侯，当只有一两个数的时侯可能发生数集重合，比如一个数既是最大值又是次小值。这种要特判。

```
#include <cstdio>
#include <iostream>
using namespace std;

int rd() {
  char act = 0;
  int f = 1, x = 0;
  while (act = getchar(), act < '0' && act != '-')
    ;
  if (act == '-') f = -1, act = getchar();
  x = act - '0';
  while (act = getchar(), act >= '0') x = x * 10 + act - '0';
  return x * f;
}

const int N = 5e5 + 5, SZ = N << 2, INF = 0x7fffffff;

int n, m;
int a[N];

struct data {
  int mx, mx2, mn, mn2, cmx, cmn, tmx, tmn, tad;
  long long sum;
} t[SZ];

void pushup(int u) {
  const int lu = u << 1, ru = u << 1 | 1;
  t[u].sum = t[lu].sum + t[ru].sum;
  if (t[lu].mx == t[ru].mx) {
    t[u].mx = t[lu].mx, t[u].cmx = t[lu].cmx + t[ru].cmx;
    t[u].mx2 = max(t[lu].mx2, t[ru].mx2);
  } else if (t[lu].mx > t[ru].mx) {
    t[u].mx = t[lu].mx, t[u].cmx = t[lu].cmx;
```

```
    t[u].mx2 = max(t[lu].mx2, t[ru].mx);
  } else {
    t[u].mx = t[ru].mx, t[u].cmx = t[ru].cmx;
    t[u].mx2 = max(t[lu].mx, t[ru].mx2);
  }
  if (t[lu].mn == t[ru].mn) {
    t[u].mn = t[lu].mn, t[u].cmn = t[lu].cmn + t[ru].cmn;
    t[u].mn2 = min(t[lu].mn2, t[ru].mn2);
  } else if (t[lu].mn < t[ru].mn) {
    t[u].mn = t[lu].mn, t[u].cmn = t[lu].cmn;
    t[u].mn2 = min(t[lu].mn2, t[ru].mn);
  } else {
    t[u].mn = t[ru].mn, t[u].cmn = t[ru].cmn;
    t[u].mn2 = min(t[lu].mn, t[ru].mn2);
  }
}

void push_add(int u, int l, int r, int v) {
  // 更新加法标记的同时，更新 $\min$ 和 $\max$ 标记
  t[u].sum += (r - l + 1ll) * v;
  t[u].mx += v, t[u].mn += v;
  if (t[u].mx2 != -INF) t[u].mx2 += v;
  if (t[u].mn2 != INF) t[u].mn2 += v;
  if (t[u].tmx != -INF) t[u].tmx += v;
  if (t[u].tmn != INF) t[u].tmn += v;
  t[u].tad += v;
}

void push_min(int u, int tg) {
  // 注意比较 $\max$ 标记
  if (t[u].mx <= tg) return;
  t[u].sum += (tg * 1ll - t[u].mx) * t[u].cmx;
  if (t[u].mn2 == t[u].mx) t[u].mn2 = tg;  // !!!
  if (t[u].mn == t[u].mx) t[u].mn = tg;    // !!!!!
  if (t[u].tmx > tg) t[u].tmx = tg;        // 更新取 $\max$ 标记
  t[u].mx = tg, t[u].tmn = tg;
}

void push_max(int u, int tg) {
  if (t[u].mn > tg) return;
  t[u].sum += (tg * 1ll - t[u].mn) * t[u].cmn;
  if (t[u].mx2 == t[u].mn) t[u].mx2 = tg;
  if (t[u].mx == t[u].mn) t[u].mx = tg;
  if (t[u].tmn < tg) t[u].tmn = tg;
  t[u].mn = tg, t[u].tmx = tg;
}

void pushdown(int u, int l, int r) {
  const int lu = u << 1, ru = u << 1 | 1, mid = (l + r) >> 1;
  if (t[u].tad)
    push_add(lu, l, mid, t[u].tad), push_add(ru, mid + 1, r, t[u].tad);
  if (t[u].tmx != -INF) push_max(lu, t[u].tmx), push_max(ru, t[u].tmx);
  if (t[u].tmn != INF) push_min(lu, t[u].tmn), push_min(ru, t[u].tmn);
  t[u].tad = 0, t[u].tmx = -INF, t[u].tmn = INF;
```

```
}

void build(int u = 1, int l = 1, int r = n) {
  t[u].tmn = INF, t[u].tmx = -INF;  // 取极限
  if (l == r) {
    t[u].sum = t[u].mx = t[u].mn = a[l];
    t[u].mx2 = -INF, t[u].mn2 = INF;
    t[u].cmx = t[u].cmn = 1;
    return;
  }
  int mid = (l + r) >> 1;
  build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
  pushup(u);
}

void add(int L, int R, int v, int u = 1, int l = 1, int r = n) {
  if (R < l || r < L) return;
  if (L <= l && r <= R) return push_add(u, l, r, v);  // !!! 忘 return
  int mid = (l + r) >> 1;
  pushdown(u, l, r);
  add(L, R, v, u << 1, l, mid), add(L, R, v, u << 1 | 1, mid + 1, r);
  pushup(u);
}

void tomin(int L, int R, int v, int u = 1, int l = 1, int r = n) {
  if (R < l || r < L || t[u].mx <= v) return;
  if (L <= l && r <= R && t[u].mx2 < v) return push_min(u, v);  // BUG: 忘了返回
  int mid = (l + r) >> 1;
  pushdown(u, l, r);
  tomin(L, R, v, u << 1, l, mid), tomin(L, R, v, u << 1 | 1, mid + 1, r);
  pushup(u);
}

void tomax(int L, int R, int v, int u = 1, int l = 1, int r = n) {
  if (R < l || r < L || t[u].mn >= v) return;
  if (L <= l && r <= R && t[u].mn2 > v) return push_max(u, v);
  int mid = (l + r) >> 1;
  pushdown(u, l, r);
  tomax(L, R, v, u << 1, l, mid), tomax(L, R, v, u << 1 | 1, mid + 1, r);
  pushup(u);
}

long long qsum(int L, int R, int u = 1, int l = 1, int r = n) {
  if (R < l || r < L) return 0;
  if (L <= l && r <= R) return t[u].sum;
  int mid = (l + r) >> 1;
  pushdown(u, l, r);
  return qsum(L, R, u << 1, l, mid) + qsum(L, R, u << 1 | 1, mid + 1, r);
}

long long qmax(int L, int R, int u = 1, int l = 1, int r = n) {
  if (R < l || r < L) return -INF;
  if (L <= l && r <= R) return t[u].mx;
  int mid = (l + r) >> 1;
```

```
  pushdown(u, l, r);
  return max(qmax(L, R, u << 1, l, mid), qmax(L, R, u << 1 | 1, mid + 1, r));
}

long long qmin(int L, int R, int u = 1, int l = 1, int r = n) {
  if (R < l || r < L) return INF;
  if (L <= l && r <= R) return t[u].mn;
  int mid = (l + r) >> 1;
  pushdown(u, l, r);
  return min(qmin(L, R, u << 1, l, mid), qmin(L, R, u << 1 | 1, mid + 1, r));
}

int main() {
  n = rd();
  for (int i = 1; i <= n; i++) a[i] = rd();
  build();
  m = rd();
  for (int i = 1; i <= m; i++) {
    int op, l, r, x;
    op = rd(), l = rd(), r = rd();
    if (op <= 3) x = rd();  // scanf("%d",&x);
    if (op == 1)
      add(l, r, x);
    else if (op == 2)
      tomax(l, r, x);
    else if (op == 3)
      tomin(l, r, x);
    else if (op == 4)
      printf("%lld\n", qsum(l, r));
    else if (op == 5)
      printf("%lld\n", qmax(l, r));
    else
      printf("%lld\n", qmin(l, r));
  }
  return 0;
}
```

吉老师证出来这个算法的复杂度是 $O(m\log^2 n)$ 的。

Mzl loves segment tree

> 两个序列 A, B，一开始 B 中的数都是 0。维护的操作是：
>
> 1. 对 A 做区间取 min
> 2. 对 A 做区间取 max
> 3. 对 A 做区间加
> 4. 询问 B 的区间和
>
> 每次操作完后，如果 A_i 的值发生变化，就给 B_i 加 1。$n,m\le 3\times 10^5$。

先考虑最容易的区间加操作。只要 $x\neq 0$ 那么整个区间的数都变化，所以给 B 作一次区间加即可。

对于区间取最值的操作，你发现你打标记与下传标记是与 B 数组一一对应的。本质上你将序列的数分成三类：最大值、最小值、非最值。并分别维护（只不过你没有建出具体的最值集合而已，但这并不妨碍维护的操作）。因此在打标记的时候顺便给 B 更新信息即可（注意不是给 B 打标记！是更新信息！）。查询的时候，你在 A 上查询，下传标记

的时侯顺便给 B 更新信息。找到需要的结点后，返回 B 的信息即可。这种操作本质上就是把最值的信息拿给 B 去维护了。另外仍要处理数集的重复问题。

CTSN loves segment tree

> 两个序列 A, B：
>
> 1. 对 A 做区间取 min
> 2. 对 B 做区间取 min
> 3. 对 A 做区间加
> 4. 对 B 做区间加
> 5. 询问区间的 $A_i + B_i$ 的最大值
>
> $n, m \le 3 \times 10^5$。

我们把区间中的**位置**分成四类：在 A, B 中同是区间最大值的位置、在 A 中是区间最大值在 B 中不是的位置、在 B 中是区间最大值在 A 中不是的位置、在 A, B 中都不是区间最大值的位置。对这四类数分别维护**答案**和**标记**即可。举个例子，我们维护 $C_{1\sim4}, M_{1\sim4}, A_{\max}, B_{\max}$ 分别表示当前区间中四类数的个数，四类数的答案的最大值，A 序列的最大值、B 序列的最大值。然后合并信息该怎么合并就怎么合并了。

复杂度仍是 $O(m\log^2 n)$。

小结

在第本章节中我们给出了四道例题，分别讲解了基本区间最值操作的维护、多个标记的优先级处理、数集分类的思想以及多个分类的维护。本质上处理区间最值的基本思想就是数集信息的分类维护与高效合并。在下一章节中，我们将探讨历史区间最值的相关问题。

历史最值问题

历史最值不等于可持久化

注意，本章所讲到的历史最值问题不同于所谓的可持久化数据结构。这类特殊的问题我们将其称为历史最值问题。历史最值的问题可以分为三类。

历史最大值

简单地说，一个位置的历史最大值就是当前位置下曾经出现过的数的最大值。形式化地定义，我们定义一个辅助数组 B，一开始与 A 完全相同。在 A 的每次操作后，我们对整个数组取 max：

$$\forall i \in [1, n],\ B_i = \max(B_i, A_i)$$

这时，我们将 B_i 称作这个位置的历史最大值，

历史最小值

定义与历史最大值类似，在 A 的每次操作后，我们对整个数组取 min。这时，我们将 B_i 称作这个位置的历史最小值，

历史版本和

辅助数组 B 一开始全部是 0。在每一次操作后，我们把整个 A 数组累加到 B 数组上

$$\forall i \in [1, n],\ B_i = B_i + A_i$$

我们称 B_i 为 i 这个位置上的历史版本和。

接下来，我们将历史最值问题分成四类讨论。

可以用标记处理的问题

CPU 监控

> 序列 A, B 一开始相同：
>
> 1. 对 A 做区间覆盖 x
> 2. 对 A 做区间加 x
> 3. 询问 A 的区间 max
> 4. 询问 B 的区间 max
>
> 每次操作后，我们都进行一次更新，$\forall i \in [1,n]$，$B_i = \max(B_i, A_i)$。$n, m \le 10^5$。

我们先不考虑操作 1。那么只有区间加的操作，我们维护标记 Add 表示当前区间增加的值，这个标记可以解决区间 max 的问题。接下来考虑历史区间 max。我们定义标记 Pre，该标记的含义是：在该标记的生存周期内，Add 标记的历史最大值。

这个定义可能比较模糊。因此我们先解释一下标记的生存周期。一个标记会经历这样的过程：

1. 在结点 u 被建立。
2. 在结点 u 接受若干个新的标记的同时，与新的标记合并（指同类标记）
3. 结点 u 的标记下传给 u 的儿子，u 的标记清空

我们认为在这个过程中，从 1 开始到 3 之前，都是结点 u 的标记的生存周期。两个标记合并后，成为同一个标记，那么他们的生存周期也会合并（即取建立时间较早的那个做为生存周期的开始）。一个与之等价的说法是，从上次把这个结点的标记下传的时刻到当前时刻这一时间段。

为什么要定义生存周期？利用这个概念，我们可以证明：在一个结点标记的生存周期内，其子结点均不会发生任何变化，并保留在这个生存周期之前的状态。道理很简单，因为在这个期间你是没有下传标记的。

于是，你就可以保证，在当前标记生存周期内的历史 Add 的最大值是可以更新到子结点的标记和信息上的。因为子结点的标记和信息在这个时间段内都没有变过。于是我们把 u 的标记下传给它的儿子 s，不难发现

$$Pre_s = \max(Pre_s, Pre_u + Add_s), Add_s = Add_u + Add_s$$

那么信息的更新也是类似的，拿对应的标记更新即可。

接下来，我们考虑操作 1。

区间覆盖操作，会把所有的数变成一个数。在这之后，无论是区间加减还是覆盖，整个区间的数仍是同一个（除非你结束当前标记的生存周期，下传标记）。因此我们可以把第一次区间覆盖后的所有标记都看成区间覆盖标记。也就是说一个标记的生存周期被大致分成两个阶段：

1. 若干个加减操作标记的合并，没有接收过覆盖标记。
2. 覆盖操作的标记，没有所谓的加减标记（加减标记转化为覆盖标记）

于是我们把这个结点的 Pre 标记拆成 (P_1, P_2)。P_1 表示第一阶段的最大加减标记；P_2 表示第二阶段的最大覆盖标记。利用相似的方法，我们可以对这个做标记下传和信息更新。时间复杂度是 $O(m \log n)$ 的（这个问题并没有区间对 x 取最值的操作哦 ~）

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
```

```
char nc() {
  static char buf[1000000], *p = buf, *q = buf;
  return p == q && (q = (p = buf) + fread(buf, 1, 1000000, stdin), p == q)
             ? EOF
             : *p++;
}

ll rd() {  // LLONG_MIN LMAX=9,223,372,036,854,775,807
  ll s = 0, w = 1;
  char ch = nc();
  while (ch < '0' || ch > '9') {
    if (ch == '-') w = -1;
    ch = nc();
  }
  while (ch >= '0' && ch <= '9') s = s * 10 + ch - '0', ch = nc();
  return s * w;
}

const int N = 1e5 + 7;

struct Tree {
  int mx, _mx;  // 区间最大值区间历史最大值
  int ad, _ad;  // 区间加标记区间阶段历史最大加标记
  int st, _st;  // 区间修改值区间阶段历史最大修改标记
} g[N * 4];

int a[N];
int n, m;
#define ls u * 2
#define rs u * 2 + 1
#define mid (l + r) / 2

void pushup(int u) {
  g[u].mx = max(g[ls].mx, g[rs].mx);
  g[u]._mx = max(g[ls]._mx, g[rs]._mx);
}

void pushadd(int u, int v, int _v) {
  g[u]._mx = max(g[u]._mx, g[u].mx + _v), g[u].mx += v;
  if (g[u].st == INT_MIN)
    g[u]._ad = max(g[u]._ad, g[u].ad + _v), g[u].ad += v;
  else
    g[u]._st = max(g[u]._st, g[u].st + _v), g[u].st += v;
}

void pushset(int u, int v, int _v) {
  g[u]._mx = max(g[u]._mx, _v), g[u].mx = v;
  g[u]._st = max(g[u]._st, _v), g[u].st = v;
}

void pushdown(int u, int l, int r) {
  if (g[u].ad || g[u]._ad)
    pushadd(ls, g[u].ad, g[u]._ad), pushadd(rs, g[u].ad, g[u]._ad),
        g[u].ad = g[u]._ad = 0;
```

```
  if (g[u].st != INT_MIN || g[u]._st != INT_MIN)
    pushset(ls, g[u].st, g[u]._st), pushset(rs, g[u].st, g[u]._st),
        g[u].st = g[u]._st = INT_MIN;
}

void build(int u = 1, int l = 1, int r = n) {
  g[u]._st = g[u].st = INT_MIN;
  if (l == r) {
    g[u].mx = a[l];
    g[u]._mx = a[l];
    return;
  }
  build(ls, l, mid), build(rs, mid + 1, r);
  pushup(u);
}

int L, R;

void add(int v, int u = 1, int l = 1, int r = n) {
  if (R < l || r < L) return;
  if (L <= l && r <= R) return pushadd(u, v, max(v, 0));
  pushdown(u, l, r);
  add(v, ls, l, mid), add(v, rs, mid + 1, r);
  pushup(u);
}

void tset(int v, int u = 1, int l = 1, int r = n) {
  if (R < l || r < L) return;
  if (L <= l && r <= R) return pushset(u, v, v);
  pushdown(u, l, r);
  tset(v, ls, l, mid), tset(v, rs, mid + 1, r);
  pushup(u);
}

int qmax(int u = 1, int l = 1, int r = n) {
  if (R < l || r < L) return INT_MIN;
  if (L <= l && r <= R) return g[u].mx;
  pushdown(u, l, r);
  return max(qmax(ls, l, mid), qmax(rs, mid + 1, r));
}

int qmaxh(int u = 1, int l = 1, int r = n) {
  if (R < l || r < L) return INT_MIN;
  if (L <= l && r <= R) return g[u]._mx;
  pushdown(u, l, r);
  return max(qmaxh(ls, l, mid), qmaxh(rs, mid + 1, r));
}

int main() {
  n = rd();
  for (int i = 1; i <= n; ++i) a[i] = rd();
  build();
  int m = rd(), z;
  for (int i = 1; i <= m; ++i) {
```

```
    char op = nc();
    while (op == ' ' || op == '\r' || op == '\n') op = nc();
    L = rd(), R = rd();
    if (op == 'Q')
      printf("%d\n", qmax());
    else if (op == 'A')
      printf("%d\n", qmaxh());
    else if (op == 'P')
      add(rd());
    else
      tset(rd());
  }
  return 0;
}
```

10.16 划分树

Authors: Xarfa

引入

划分树是一种来解决区间第 K 大的一种数据结构，其常数、理解难度都要比主席树低很多。同时，划分树紧贴「第 K 大」，所以是一种基于排序的一种数据结构。

建议先学完主席树再看划分树哦

过程

建树

划分树的建树比较简单，但是相对于其他树来说比较复杂。

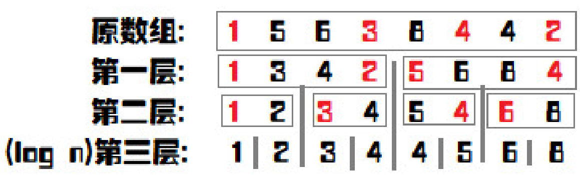


图 10.46

如图，每一层都有一个看似无序的数组。其实，每一个被红色标记的数字都是**要分配到左儿子的**。而分配的规则是什么？就是与**这一层的中位数**做比较，如果小于等于中位数，则分到左边，否则分到右边。但是这里要注意一下：并不是严格的**小于等于就分到左边，否则分到右边**。因为中位数可能有相同，而且与 N 的奇偶有一定关系。下面的代码展示会有一个巧妙的运用，大家可以参照代码。

我们不可能每一次都对每一层排序，这样子不说常数，就算是理论复杂度也过不去。我们想，找中位数，一次排序就够了。为什么？比如，我们求 l, r 的中位数，其实就是在排完序过后的 `num[mid]`。

两个关键数组：

tree[log(N),N]: 也就是树，要存下所有的值，空间复杂度 $O(n\log n)$。toleft[log(N),n]: 也就是每一层 1~i 进入左儿子的数量，这里需要理解一下，这是一个前缀和。

" 实现"

```
procedure Build(left,right,deep:longint); // left,right 表示区间左右端点,deep 是第
几层
var
  i,mid,same,ls,rs,flag:longint; // 其中 flag 是用来平衡左右两边的数量的
begin
  if left=right then exit; // 到底层了
  mid:=(left+right) >> 1;
  same:=mid-left+1;
  for i:=left to right do
    if tree[deep,i]<num[mid] then
      dec(same);

  ls:=left; // 分配到左儿子的第一个指针
  rs:=mid+1; // 分配到右儿子的第一个指针
  for i:=left to right do
  begin
    flag:=0;
    if (tree[deep,i]<num[mid])or((tree[deep,i]=num[mid])and(same>0)) then // 分配
到左边的条件
    begin
      flag:=1; tree[deep+1,ls]:=tree[deep,i]; inc(ls);
      if tree[deep,i]=num[mid] then // 平衡左右个数
        dec(same);
    end
    else
    begin
      tree[deep+1,rs]:=tree[deep,i]; inc(rs);
    end;
    toleft[deep,i]:=toleft[deep,i-1]+flag;
  end;
  Build(left,mid,deep+1); // 继续
  Build(mid+1,right,deep+1);
end;
```

查询

那我们先扯一下主席树的内容。在用主席树求区间第 K 小的时候，我们以 K 为基准，向左就向左，向右要减去向左的值，在划分树中也是这样子的。

查询难理解的，在于**区间缩小**这种东西。下图，查询的是 3 到 7, 那么下一层就只需要查询 2 到 3 了。当然，我们定义 $[\text{left}, \text{right}]$ 为缩小后的区间（目标区间），$[l, r]$ 还是所在节点的区间。那为什么要标出目标区间呢？因为那是**判定答案在左边还是右边的基准**。

" 实现"

```
function Query(left,right,k,l,r,deep:longint):longint;
var
  mid,x,y,cnt,rx,ry:longint;
begin
```

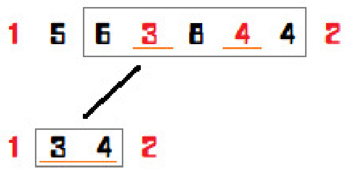


图 10.47

```
  if left=right then // 写成 l=r 也无妨，因为目标区间也一定有答案
    exit(tree[deep,left]);
  mid:=(l+r) >> 1;
  x:=toleft[deep,left-1]-toleft[deep,l-1]; // l 到 left 的去左儿子的个数
  y:=toleft[deep,right]-toleft[deep,l-1]; // l 到 right 的去左儿子的个数
  ry:=right-l-y; rx:=left-l-x; // ry 是 l 到 right 去右儿子的个数,rx 则是 l 到 l
eft 去右儿子的个数
  cnt:=y-x; // left 到 right 左儿子的个数
  if cnt>=k then // 主席树常识啦
    Query:=Query(l+x,l+y-1,k,l,mid,deep+1) // l+x 就是缩小左边界,l+y-1 就是缩小右
区间。对于上图来说,就是把节点 1 和 2 放弃了。
  else
    Query:=Query(mid+rx+1,mid+ry+1,k-cnt,mid+1,r,deep+1); // 同样是缩小区间, 只不
过变成了右边而已。注意要将 k 减去 cnt。
end;
```

性质

时间复杂度: 一次查询只需要 $O(\log n)$，m 次询问，就是 $O(m\log n)$。

空间复杂度: 只需要存储 $O(n\log n)$ 个数字。

亲测结果：主席树:1482ms、划分树:889ms。(非递归，常数比较小)

划分树的应用

例题：Luogu P3157[CQOI2011] 动态逆序对[1]

> 题意简述：给定一个 n 个元素的排列（$n\le 10^5$），有 m 次询问（$m\le 5\times 10^4$），每次删去排列中的一个数，求删去这个数之后排列的逆序对个数。

这题可以使用 CDQ 在 $\Theta(n\log^2 n)$ 的时间及 $\Theta(n)$ 的空间内解决，并且 CDQ 的常数也很优秀。

如果这道题改为强制在线，则一般使用树状数组 + 主席树的树套树解法解决，时间复杂度为 $\Theta(n\log^2 n)$，空间复杂度为 $\Theta(n\log^2 n)$，常数略大，同样可以过此题。

而使用划分树的话就可以在 $\Theta(n\log^2 n)$ 的时间及 $\Theta(n\log n)$ 的空间内在线解决本题，同时常数也比树套树解法少很多。(大致与 CDQ 相当。)

注意：为了编程实现方便，本文依照位置的中间值将大数组划分为两个小数组，即下文中的划分树相当于是归并排序的过程，而非快速排序的过程。最顶层的大数组为有序数组，最底层为原数组。

对于每一个划分树中的节点，我们称他为右节点当且仅当他在下一层会被划分到右孩子，即原数组中位置比较靠后的那些数，相似的可以定义左节点。如果在建树的过程中将最顶层排为有序的，类似于归并排序求逆序对，可以发现一个数组的逆序对个数就是在每个左节点之前的右节点的个树和。

再考虑删除操作。删除一个左节点会将整个数组的逆序对减少在他之前右结点的个数，而删除一个右节点会减少在他之后的左节点个数。那么可以考虑每次动态维护「每一个左节点之前的右结点个数」和「每一个右节点之后的左节点个数」。这可以使用树状数组简单维护。

需要注意的是，在使用树状数组维护时只能计算在划分树中同一块内的贡献，而不能跳出块。对于树状数组来说有一个较为巧妙的处理方式。

考虑划分树上每一块的下标范围肯定为 $[c \times 2^k + 1, (c+1) \times 2^k]$ 的形式，列举如下（由于代码中不会涉及到划分树最底层的处理，因此只枚举到倒数第二层）:

```
[0001 0010] [0011 0100] [0101 0110] [0111 1000] [1001 1010] [1011 1100] [1101 11
10] [1111 10000]  lev=1
[0001 0010 0011 0100]   [0101 0110 0111 1000]   [1001 1010 1011 1100]   [1101 11
10 1111 10000]    lev=2
[0001 0010 0011 0100 0101 0110 0111 1000]       [1001 1010 1011 1100 1101 1110 1
111 10000]        lev=3
[0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111 1000
0]                lev=4
```

回忆一下树状数组的原理，在向上跳的时候，我们每次 `x += lowbit(x)`。如果在向上跳的时候可以保证不跳出块，就可以保证只会影响到块内元素的值。向上查询也类似。

而如果要在向上跳的同时保证不跳出块，只需要保证在跳的时候满足 $lowbit(x) < 2^{lev}$ 即可。

而向下跳则是完全不同的处理方式。每一块的下标如果使用 0-index 表示的话，即为 $[c \times 2^k, (c+1) \times 2^k)$ 的形式。那么，只需将某一个下标的值右位移 k，即可得出它在哪一块中。在向下跳的时候时刻判断是否跳出块即可。

需要注意的是，按这一方法实现的树状数组会访问到的最大下标是距离 n 最近的 2 的整次幂，因此数组下标不能开 n。

由于需要在 $\log n$ 层修改，在第 k 层修改的时间复杂度为 $\Theta(k)$，最终时间复杂度即为 $\Theta(n \log n + m \log^2 n)$。

附代码：

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;
typedef long long lld;

int getint()  // 本题没有负数输入，因此快读不需判断负号。
{
  char ch;
  while ((ch = getchar()) < '!')
    ;
  int x = ch ^ '0';
  while ((ch = getchar()) > '!') x = (x * 10) + (ch ^ '0');
  return x;
}

void putll(lld x) {  // 输出 long long
  if (x / 10) putll(x / 10);
  putchar((x % 10) ^ '0');
}

int ti[2][17][131073];

int lowbit(int x) { return x & (-x); }

void fixup(int k, int x) {
  for (; lowbit(x) < 1 << k; x += lowbit(x)) {
    ++ti[0][k - 1][x];
  }
```

```
  ++ti[0][k - 1][x];
}

void fixdown(int k, int x) {
  for (; ((x ^ lowbit(x)) - 1) >> k == (x - 1) >> k; x ^= lowbit(x)) {
    ++ti[1][k - 1][x];
  }
  ++ti[1][k - 1][x];
}

int queryup(int k, int x) {
  int res = 0;
  for (; lowbit(x) < 1 << k; x += lowbit(x)) {
    res += ti[1][k - 1][x];
  }
  return res + ti[1][k - 1][x];
}

int querydown(int k, int x) {
  int res = 0;
  for (; ((x ^ lowbit(x)) - 1) >> k == (x - 1) >> k; x ^= lowbit(x)) {
    res += ti[0][k - 1][x];
  }
  return res + ti[0][k - 1][x];
}

int ai[100005];
int tx[100005];

lld mgx(int* npi, int* nai, int* rxi, int* lxi, int al, int ar, int bl,
        int br) {
  int rx = 1;
  int lx = ar - al;
  lld res = 0;
  for (; al != ar || bl != br; ++npi, ++nai, ++rxi, ++lxi) {
    if (al != ar && (bl == br || tx[al] < tx[bl])) {
      --lx;
      *rxi = rx;
      *nai = tx[al];
      *npi = al;
      ++al;
    } else {
      ++rx;
      *lxi = lx;
      res += ar - al;
      *nai = tx[bl];
      *npi = bl;
      ++bl;
    }
  }
  return res;
}

int npi[1700005];
```

```
int nri[1700005];
int nli[1700005];

int main() {
  const int n = getint();
  const int m = getint();
  for (int i = 1; i <= n; ++i) {
    ai[i] = getint();
  }

  if (n == 1) {
    for (int i = 1; i <= m; ++i) {
      putchar('0');
      putchar('\n');
    }
    return 0;
  }

  const int logn = 31 - __builtin_clz(n - 1);

  lld ans = 0;
  for (int i = logn; i >= 0; --i) {
    memcpy(tx, ai, (n + 1) * sizeof(int));
    for (int j = 1; j <= n; j += 1 << (logn - i + 1)) {
      ans += mgx(npi + n * i + j, ai + j, nri + n * i + j, nli + n * i + j, j,
                 min(n + 1, j + (1 << (logn - i))),
                 min(n + 1, j + (1 << (logn - i))),
                 min(n + 1, j + (1 << (logn - i + 1))));
    }
  }

  putll(ans);
  putchar('\n');

  for (int asdf = 1; asdf < m; ++asdf) {
    int x = getint();
    for (int i = 0, p = 0; i <= logn; ++i, p += n) {
      if (nri[p + x]) {
        ans -= nri[p + x] - querydown(logn - i + 1, x) - 1;
        fixdown(logn - i + 1, x);
      } else {
        ans -= nli[p + x] - queryup(logn - i + 1, x);
        fixup(logn - i + 1, x);
      }
      x = npi[p + x];
    }
    putll(ans);
    putchar('\n');
  }
}
```

后记

参考博文:传送门[2]。

参考资料与注释

[1] Luogu P3157[CQOI2011] 动态逆序对

[2] 传送门

10.17 二叉搜索树 & 平衡树

10.17.1 二叉搜索树 & 平衡树

Authors: 2323122, aofall, AtomAlpaca, Bocity, CoelacanthusHex, countercurrent-time, Early0v0, Enter-tainer, fearlessxjdx, Great-designer, H-J-Granger, hsfzLZH1, iamtwz, Ir1d, ksyx, Marcythm, NachtgeistW, ouuan, Persdre, shuzhouliu, StudyingFather, SukkaW, Tiphereth-A, wsyhb, Yesphet, yuhuoji, lingkerio

定义

二叉搜索树是一种二叉树的树形数据结构，其定义如下：

1. 空树是二叉搜索树。
2. 若二叉搜索树的左子树不为空，则其左子树上所有点的附加权值均小于其根节点的值。
3. 若二叉搜索树的右子树不为空，则其右子树上所有点的附加权值均大于其根节点的值。
4. 二叉搜索树的左右子树均为二叉搜索树。

二叉搜索树上的基本操作所花费的时间与这棵树的高度成正比。对于一个有 n 个结点的二叉搜索树中，这些操作的最优时间复杂度为 $O(\log n)$，最坏为 $O(n)$。随机构造这样一棵二叉搜索树的期望高度为 $O(\log n)$。

过程

二叉搜索树节点的定义

" 实现"

```
struct TreeNode {
  int key;
  TreeNode* left;
  TreeNode* right;
  // 维护其他信息，如高度，节点数量等
  int size;   // 当前节点为根的子树大小
  int count;  // 当前节点的重复数量

  TreeNode(int value)
      : key(value), size(1), count(1), left(nullptr), right(nullptr) {}
};
```

遍历二叉搜索树

由二叉搜索树的递归定义可得，二叉搜索树的中序遍历权值的序列为非降的序列。时间复杂度为 $O(n)$。

遍历一棵二叉搜索树的代码如下：

" 实现"

```
void inorderTraversal(TreeNode* root) {
  if (root == nullptr) {
    return;
  }
  inorderTraversal(root->left);
  std::cout << root->key << " ";
  inorderTraversal(root->right);
}
```

查找最小/最大值

由二叉搜索树的性质可得，二叉搜索树上的最小值为二叉搜索树左链的顶点，最大值为二叉搜索树右链的顶点。时间复杂度为 $O(h)$。

" 实现"

```
int findMin(TreeNode* root) {
  if (root == nullptr) {
    return -1;
  }
  while (root->left != nullptr) {
    root = root->left;
  }
  return root->key;
}

int findMax(TreeNode* root) {
  if (root == nullptr) {
    return -1;
  }
  while (root->right != nullptr) {
    root = root->right;
  }
  return root->key;
}
```

搜索元素

在以 `root` 为根节点的二叉搜索树中搜索一个值为 `value` 的节点。

分类讨论如下：

- 若 `root` 为空，返回 `false`。
- 若 `root` 的权值等于 `value`，返回 `true`。
- 若 `root` 的权值大于 `value`，在 `root` 的左子树中继续搜索。
- 若 `root` 的权值小于 `value`，在 `root` 的右子树中继续搜索。

时间复杂度为 $O(h)$。

" 实现"

```
bool search(TreeNode* root, int target) {
```

```
  if (root == nullptr) {
    return false;
  }
  if (root->key == target) {
    return true;
  } else if (target < root->key) {
    return search(root->left, target);
  } else {
    return search(root->right, target);
  }
}
```

插入，删除，修改都需要先在二叉搜索树中进行搜索。

插入一个元素

在以 root 为根节点的二叉搜索树中插入一个值为 value 的节点。

分类讨论如下：

- 若 root 为空，直接返回一个值为 value 的新节点。
- 若 root 的权值等于 value，该节点的附加域该值出现的次数自增 1。
- 若 root 的权值大于 value，在 root 的左子树中插入权值为 value 的节点。
- 若 root 的权值小于 value，在 root 的右子树中插入权值为 value 的节点。

时间复杂度为 $O(h)$。

" 实现"

```
TreeNode* insert(TreeNode* root, int value) {
  if (root == nullptr) {
    return new TreeNode(value);
  }
  if (value < root->key) {
    root->left = insert(root->left, value);
  } else if (value > root->key) {
    root->right = insert(root->right, value);
  } else {
    root->count++;  // 节点值相等，增加重复数量
  }
  root->size = root->count + (root->left ? root->left->size : 0) +
               (root->right ? root->right->size : 0);  // 更新节点的子树大小
  return root;
}
```

删除一个元素

在以 root 为根节点的二叉搜索树中删除一个值为 value 的节点。

先在二叉搜索树中搜索权值为 value 的节点，分类讨论如下：

- 若该节点的附加 count 大于 1，只需要减少 count。
- 若该节点的附加 count 为 1：
  - 若 root 为叶子节点，直接删除该节点即可。
  - 若 root 为链节点，即只有一个儿子的节点，返回这个儿子。
  - 若 count 有两个非空子节点，一般是用它左子树的最大值（左子树最右的节点）或右子树的最小值（右子

树最左的节点）代替它，然后将它删除。

时间复杂度 $O(h)$。

" 实现"

方法使用 `root = remove(root, 1)` 表示删除根节点为 `root` 树中值为 1 的节点，并返回新的根节点。

```cpp
// 此处返回值为删除 value 后的新 root
TreeNode* remove(TreeNode* root, int value) {
  if (root == nullptr) {
    return root;
  }
  if (value < root->key) {
    root->left = remove(root->left, value);
  } else if (value > root->key) {
    root->right = remove(root->right, value);
  } else {
    if (root->count > 1) {
      root->count--;  // 节点重复数量大于 1，减少重复数量
    } else {
      if (root->left == nullptr) {
        TreeNode* temp = root->right;
        delete root;
        return temp;
      } else if (root->right == nullptr) {
        TreeNode* temp = root->left;
        delete root;
        return temp;
      } else {
        TreeNode* successor = findMinNode(root->right);
        root->key = successor->key;
        root->count = successor->count;  // 更新重复数量
        root->right = remove(root->right, successor->key);
      }
    }
  }
  return root;
}

// 此处以右子树的最小值为例
TreeNode* findMinNode(TreeNode* root) {
  while (root->left != nullptr) {
    root = root->left;
  }
  return root;
}
```

求元素的排名

排名定义为将数组元素升序排序后第一个相同元素之前的数的个数加一。

查找一个元素的排名，首先从根节点跳到这个元素，若向右跳，答案加上左儿子节点个数加当前节点重复的数个数，最后答案加上终点的左儿子子树大小加一。

时间复杂度 $O(h)$。

" 实现"

```
int queryRank(TreeNode* root, int v) {
  if (root == nullptr) return 0;
  if (root->key == v) return (root->left ? root->left->size : 0) + 1;
  if (root->key > v) return queryRank(root->left, v);
  return queryRank(root->right, v) + (root->left ? root->left->size : 0) +
         root->count;
}
```

查找排名为 k 的元素

在一棵子树中，根节点的排名取决于其左子树的大小。

- 若其左子树的大小大于等于 k，则该元素在左子树中；
- 若其左子树的大小在区间 $[k-count, k-1]$（count 为当前结点的值的出现次数）中，则该元素为子树的根节点；
- 若其左子树的大小小于 $k-count$，则该元素在右子树中。

时间复杂度 $O(h)$。

" 实现"

```
int querykth(TreeNode* root, int k) {
  if (root == nullptr) return -1;  // 或者根据需求返回其他合适的值
  if (root->left) {
    if (root->left->size >= k) return querykth(root->left, k);
    if (root->left->size + root->count >= k) return root->key;
  } else {
    if (k == 1) return root->key;
  }
  return querykth(root->right,
                  k - (root->left ? root->left->size : 0) - root->count);
}
```

平衡树简介

使用搜索树的目的之一是缩短插入、删除、修改和查找（插入、删除、修改都包括查找操作）节点的时间。

关于查找效率，如果一棵树的高度为 h，在最坏的情况，查找一个关键字需要对比 h 次，查找时间复杂度（也为平均查找长度 ASL，Average Search Length）不超过 $O(h)$。一棵理想的二叉搜索树所有操作的时间可以缩短到 $O(\log n)$（n 是节点总数）。

然而 $O(h)$ 的时间复杂度仅为理想情况。在最坏情况下，搜索树有可能退化为链表。想象一棵每个结点只有右孩子的二叉搜索树，那么它的性质就和链表一样，所有操作（增删改查）的时间是 $O(n)$。

可以发现操作的复杂度与树的高度 h 有关。由此引出了平衡树，通过一定操作维持树的高度（平衡性）来降低操作的复杂度。

平衡性的定义

关于一棵搜索树是否「**平衡**」，不同的平衡树中对「**平衡**」有着不同的定义。比如以 T 为根节点的二叉搜索树，左子树和右子树的高度相差很大，或者左子树的节点个数远大于右子树的节点个数，这棵树显然不具有平衡性。

对于二叉搜索树来说，常见的平衡性的定义是指：以 T 为根节点的树，每一个结点的左子树和右子树高度差最多为 1。

- Splay 树 中，对于任意节点的访问操作（搜索、插入还是删除），都会将被访问的节点移动到树的根节点位置。

- AVL 树 每个节点 N 维护以 N 为根节点的树的高度信息。AVL 树对平衡性的定义：如果 T 是一棵 AVL 树，当且仅当左右子树也是 AVL 树，且 $|height(T->left) - height(T->right)| \leq 1$。
- Size Balanced Tree 每个节点 N 维护以 N 为根节点的树中节点个数 `size`。对平衡性的定义：任意节点的 `size` 不小于其兄弟节点（Sibling）的所有子节点（Nephew）的 `size`。
- B 树 对平衡性的定义：每个节点应该保持在一个预定义的范围内的关键字数量。

此外，对于拥有同样元素值集合的搜索树，平衡状态可能是不唯一的。也就是说，可能两棵不同的搜索树，含有的元素值集合相同，并且都是平衡的。

平衡的调整过程

对不满足平衡条件的搜索树进行调整操作，可以使不平衡的搜索树重新具有平衡性。

关于二叉平衡树，平衡的调整操作分为包括**左旋**（**Left Rotate 或者 zag**）和**右旋**（**Right Rotate 或者 zig**）两种。由于二叉平衡树在调整时需要保证中序遍历序列不变。这两种操作均不改变中序遍历序列。

在这里先介绍右旋，右旋也称为「右单旋转」或「LL 平衡旋转」。对于结点 A 的右旋操作是指：将 A 的左孩子 B 向右上旋转，代替 A 成为根节点，将 A 结点向右下旋转成为 B 的右子树的根结点，B 的原来的右子树变为 A 的左子树。

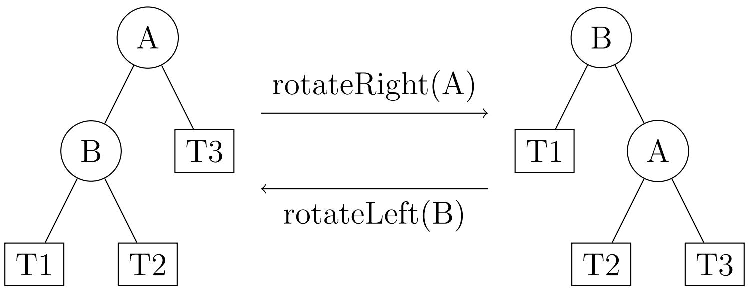


图 10.48 bst-rotate

右旋操作只改变了三组结点关联，相当于对三组边进行循环置换一下，因此需要暂存一个结点再进行轮换更新。

对于右旋操作一般的更新顺序是：暂存 B 结点（新的根节点），让 A 的左孩子指向 B 的右子树 $T2$，再让 B 的右孩子指针指向 A，最后让 A 的父结点指向暂存的 B。

完全同理，有对应的左旋操作，也称为「左单旋转」或「RR 平衡旋转」。左旋操作与右旋操作互为镜像。

下面给出左旋和右旋的代码。

" 实现"

```cpp
TreeNode* rotateLeft(TreeNode* root) {
  TreeNode* newRoot = root->right;
  root->right = newRoot->left;
  newRoot->left = root;
  // 更新相关节点的信息
  updateHeight(root);
  updateHeight(newRoot);
  return newRoot;  // 返回新的根节点
}

TreeNode* rotateRight(TreeNode* root) {
  TreeNode* newRoot = root->left;
  root->left = newRoot->right;
  newRoot->right = root;
  updateHeight(root);
  updateHeight(newRoot);
```

```
    return newRoot;
}
```

对于这段示例代码，在调用时需要保存 `root` 的父节点 `pre`。方法返回指向新的根节点的指针，只需要将 `pre` 指向新的根节点即可。

四种平衡性破坏的情况

虽然不同的二叉平衡树的定义有所区别，不同二叉平衡树区别只在于节点维护的信息不同，以及旋转调整后节点更新的信息不同。二叉平衡树平衡性被破坏的情况只有以下四种。进行平衡性调整的操作只包括左旋和右旋。以下先介绍四种情况，再对不同的二叉平衡树进行对比。

LL 型：T 的左孩子的左子树过长导致平衡性破坏。

调整方式：右旋节点 T。

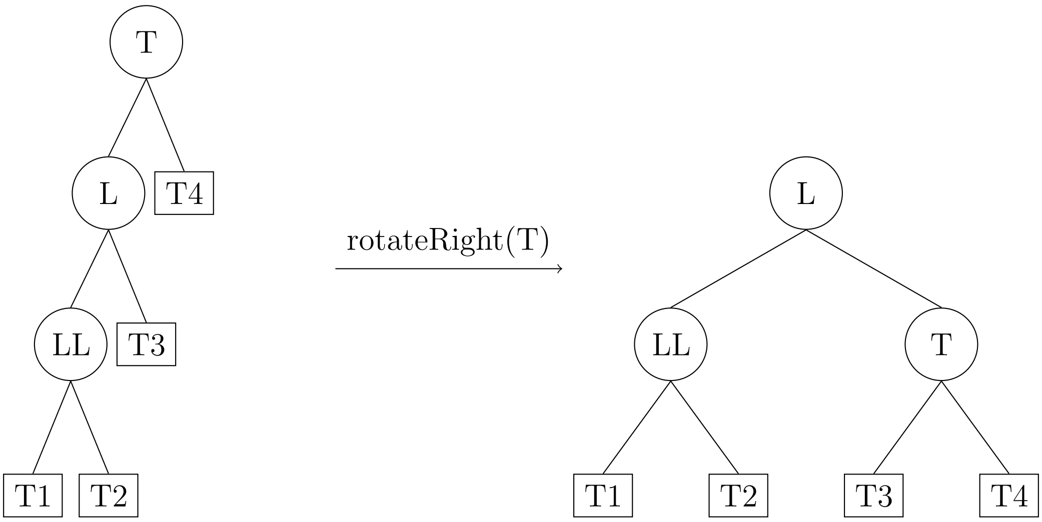


图 10.49 bst-LL

RR 型：与 LL 型类似，T 的右孩子的右子树过长导致平衡性破坏。

调整方式：左旋节点 T。

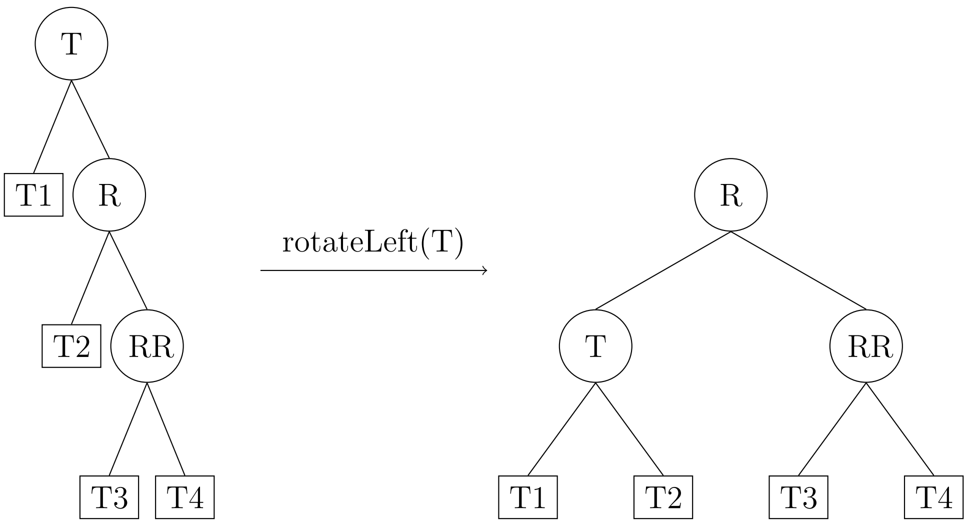


图 10.50 bst-RR

LR 型：T 的左孩子的右子树过长导致平衡性破坏。

调整方式：先左旋节点 L，成为 LL 型，再右旋节点 T。

RL 型：与 LR 型类似，T 的右孩子的左子树过长导致平衡性破坏。

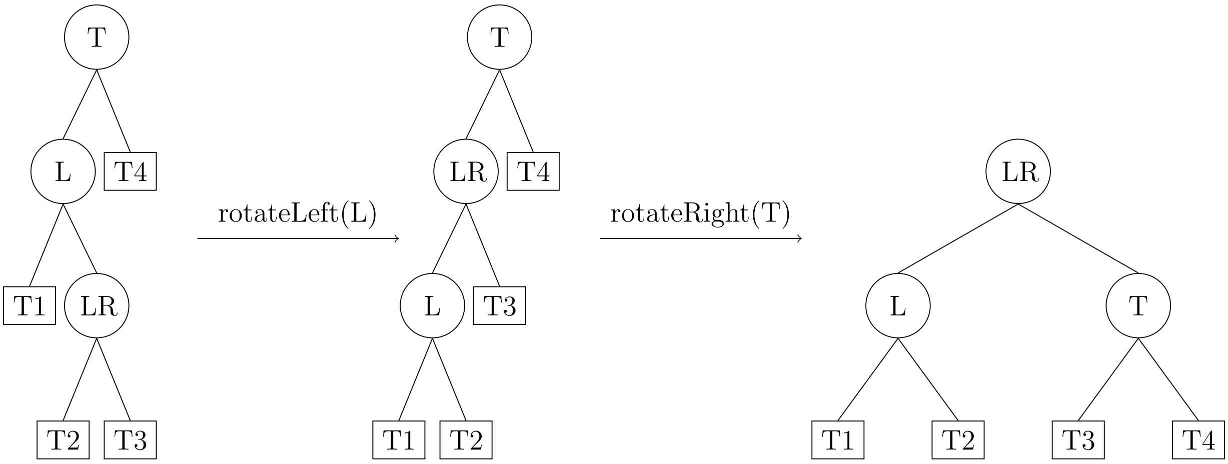


图 10.51 bst-LR

调整方式：先右旋节点 R，成为 RR 型，再左旋节点 T。

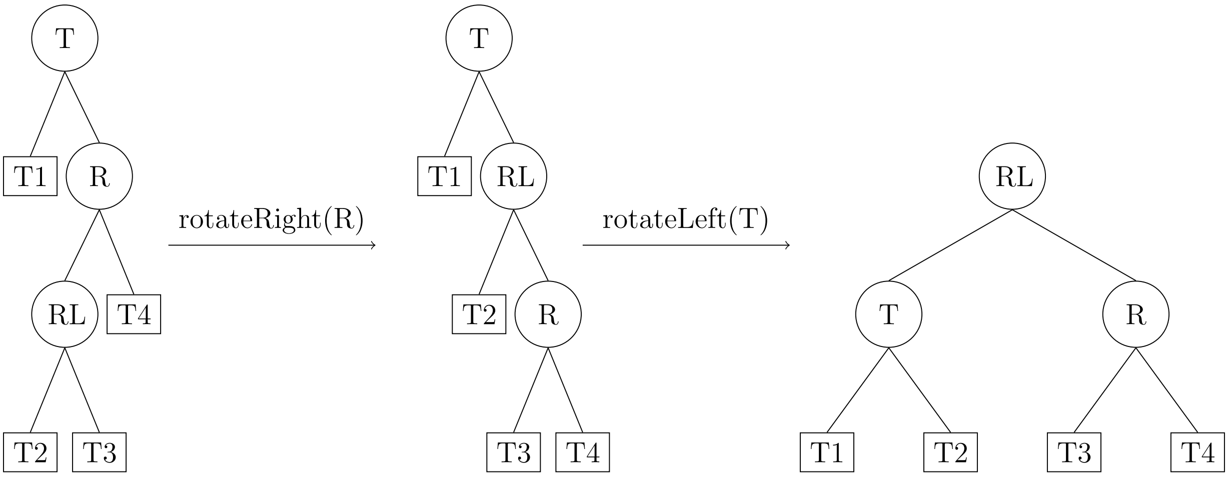


图 10.52 bst-RL

10.17.2 Treap

Authors: Dev-XYS, ttzytt, Sora233

前置知识：朴素二叉搜索树

简介

Treap（树堆）是一种**弱平衡**的**二叉搜索树**。它同时符合二叉搜索树和堆的性质，名字也因此为 tree（树）和 heap（堆）的组合。

其中，二叉搜索树的性质是：

- 左子节点的值（*val*）比父节点大
- 右子节点的值（*val*）比父节点小（当然这也是可以反过来的）

堆的性质是：

- 子节点值（*priority*）比父节点大或小（取决于是小根堆还是大根堆）

不难看出，如果用的是同一个值，那么这两种数据结构在组合后会变成一条链，所以我们再在搜索树的基础上，

引入一个给堆的值 *priority*。对于 *val* 值，我们维护搜索树的性质，对于 *priority* 值，我们维护堆的性质。其中 *priority* 这个值是随机给出的。

下图就是一个 Treap 的例子（这里使用的是小根堆，即根节点的值最小）。

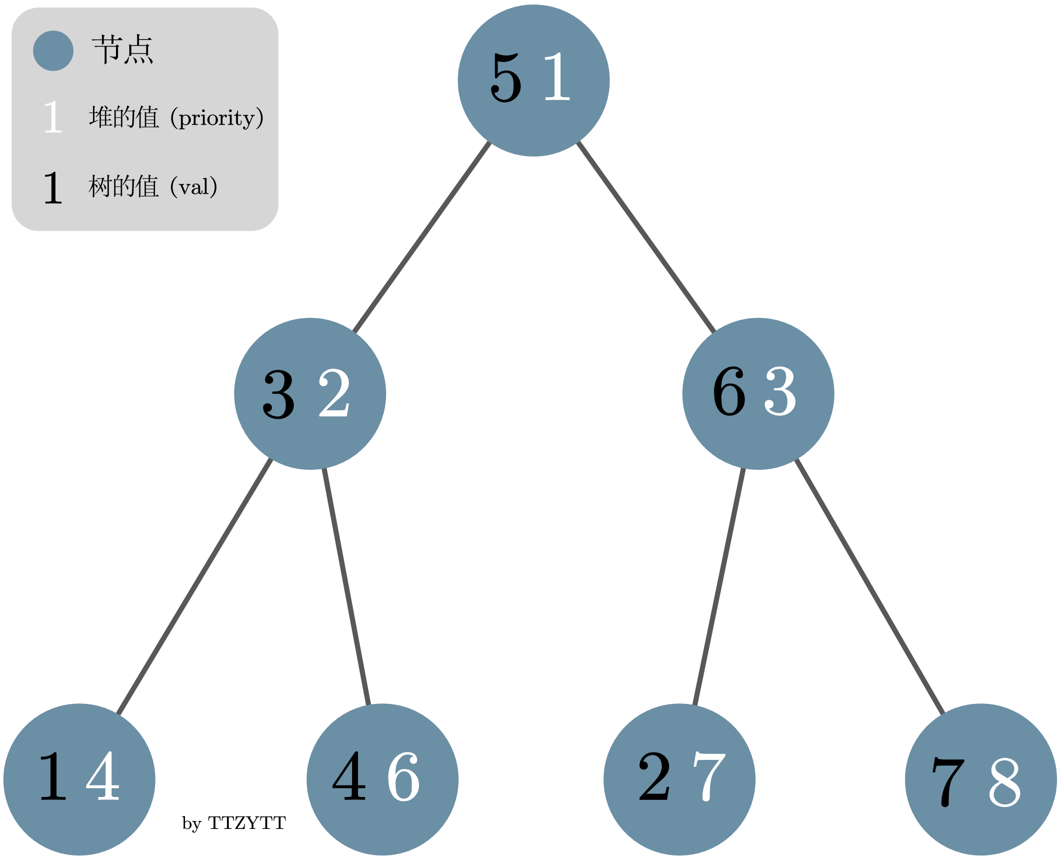


图 10.53

那我们为什么需要大费周章的去让这个数据结构符合树和堆的性质，并且随机给出堆的值呢？

要理解这个，首先需要理解朴素二叉搜索树的问题。在给朴素搜索树插入一个新节点时，我们需要从这个搜索树的根节点开始递归，如果新节点比当前节点小，那就向左递归，反之亦然。

最后当发现当前节点没有子节点时，就根据新节点的值的大小，让新节点成为当前节点的左或右子节点。

如果新插入的节点的值是随机的，那这个朴素搜索树的形状会非常的「胖」，上图的 Treap 就是一个例子。也就是说，每一层的节点比较多。

在这样的情况下，这个搜索树的层数是会比较接近 $\log_2 n$（n 为节点数）的，查询的复杂度也是 $\log_2 n$（因为只要递归这么多层就能查到）。

不过，这只是在随机情况下的复杂度，如果我们按照下面这个非常有序的顺序给一个朴素的搜索树插入节点。

```
1 2 3 4 5
```

那……

这个树就会变得非常「瘦长」（每次插入的节点都比前面的大，所以都被安排到右子节点了）：

不难看出，现在这个二叉搜索树已经退化成链了，查询的复杂度也从 $\log_2 n$ 变成了线性。

而 treap 要解决的正是这个问题。它通过随机化的 *priority* 属性，以及维护堆性质的过程，「打乱」了节点的插入顺序。从而让二叉搜索树达到了理想的复杂度，避免了退化成链的问题。

笔者并不清楚如何去严格的证明这样随机化的过程可以让搜索树的复杂度的**期望值**保持在 $\log_2 n$，但我们可以试着感性的去理解一下。

首先，我们需要认识到一个节点的 *priority* 属性是和它所在的层数有直接关联的。再回忆堆的性质：

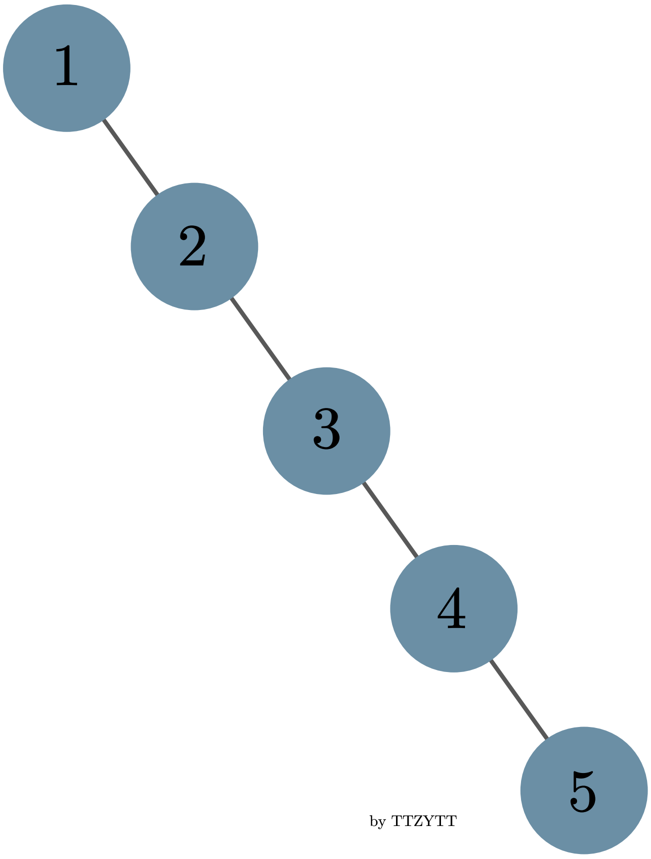


图 10.54

- 子节点值（*priority*）比父节点大或小（取决于是小根堆还是大根堆）

我们发现层数低的节点，比如整个树的根节点，它的 *priority* 属性也会更小（在小根堆中）。并且，在朴素的搜索树中，先被插入的节点，也更有可能会有比较小的层数。我们可以把这个 *priority* 属性和被插入的顺序关联起来理解，这样，也就理解了为什么 treap 可以把节点插入的顺序通过 *priority* 打乱。

在给 treap 插入新节点时，需要同时维护树和堆的性质，为了达到这个目的，有两种方法被发明了出来，分别是旋转和分裂、合并。使用这两种方法的 treap 被分别成为有旋式 treap 和无旋式 treap。

旋转 treap

旋转 treap 维护平衡的方式为旋转，和 AVL 树的旋转操作类似，分为**左旋**和**右旋**。即在满足二叉搜索树的条件下根据堆的优先级对 treap 进行平衡操作。

旋转 treap 在做普通平衡树题的时候，是所有平衡树中常数较小的。因为普通的二叉搜索树会被递增或递减的数据卡，用 treap 对每个节点定义一个由 `rand` 得到的权值，从而防止特殊数据卡。同时在每次删除/插入时通过这个权值决定要不要旋转即可，其他操作与二叉搜索树类似。

大部分的树形数据结构都有指针和数组模拟两种实现方法，下面将会详细的分部分讲解指针版的代码，如果想要学习数组实现，可以拉到最下面的完整代码部分。

> info
>
> 注意本代码中的 `rank` 代表前面讲的 *priority* 变量（堆的值）。并且，维护的堆的性质是小根堆（*priority* 小的在上面）。本代码来源。[1]

节点结构

```cpp
struct Node {
  Node *ch[2];  // 两个子节点的地址
  int val, rank;
  int rep_cnt;  // 当前这个值（val）重复出现的次数
  int siz;      // 以当前节点为根的子树大小

  Node(int val) : val(val), rep_cnt(1), siz(1) {
    ch[0] = ch[1] = nullptr;
    rank = rand();
    // 注意初始化的时候，rank 是随机给出的
  }

  void upd_siz() {
    // 用于旋转和删除过后，重新计算 siz 的值
    siz = rep_cnt;
    if (ch[0] != nullptr) siz += ch[0]->siz;
    if (ch[1] != nullptr) siz += ch[1]->siz;
  }
};
```

旋转

旋转操作是 treap 的一个非常重要的操作，主要用来在保持 treap 树性质的同时，调整不同节点的层数，以达到维护堆性质的作用。

旋转操作的左旋和右旋可能不是特别容易区分，以下是两个较为明显的特点：

旋转操作的含义：

- 在不影响搜索树性质的前提下，把和旋转方向相反的子树变成根节点（如左旋，就是把右子树变成根节点）
- 不影响性质，并且在旋转过后，跟旋转方向相同的子节点变成了原来的根节点（如左旋，旋转完之后的左子节点是旋转前的根节点）

左旋和右旋操作是相互的，如下图。

```cpp
enum rot_type { LF = 1, RT = 0 };

void _rotate(Node *&cur,
             rot_type dir) {  // dir 参数代表旋转的方向 0 为右旋，1 为左旋
  // 注意传进来的 cur 是指针的引用，也就是改了这个
  // cur，变量是跟着一起改的，如果这个 cur 是别的树的子节点，根据 ch
  // 找过来的时候，也是会找到这里的

  // 以下的代码解释的均是左旋时的情况
  Node *tmp = cur->ch[dir];  // 让 C 变成根节点，
                             // 这里的 tmp
                             // 是一个临时的节点指针，指向成为新的根节点的节点

  /* 左旋：也就是让右子节点变成根节点
   *         A                 C
   *        / \               / \
   *       B   C    ---->    A   E
   *          / \           / \
   *         D   E         B   D
   */
  cur->ch[dir] = tmp->ch[!dir];    // 让 A 的右子节点变成 D
```

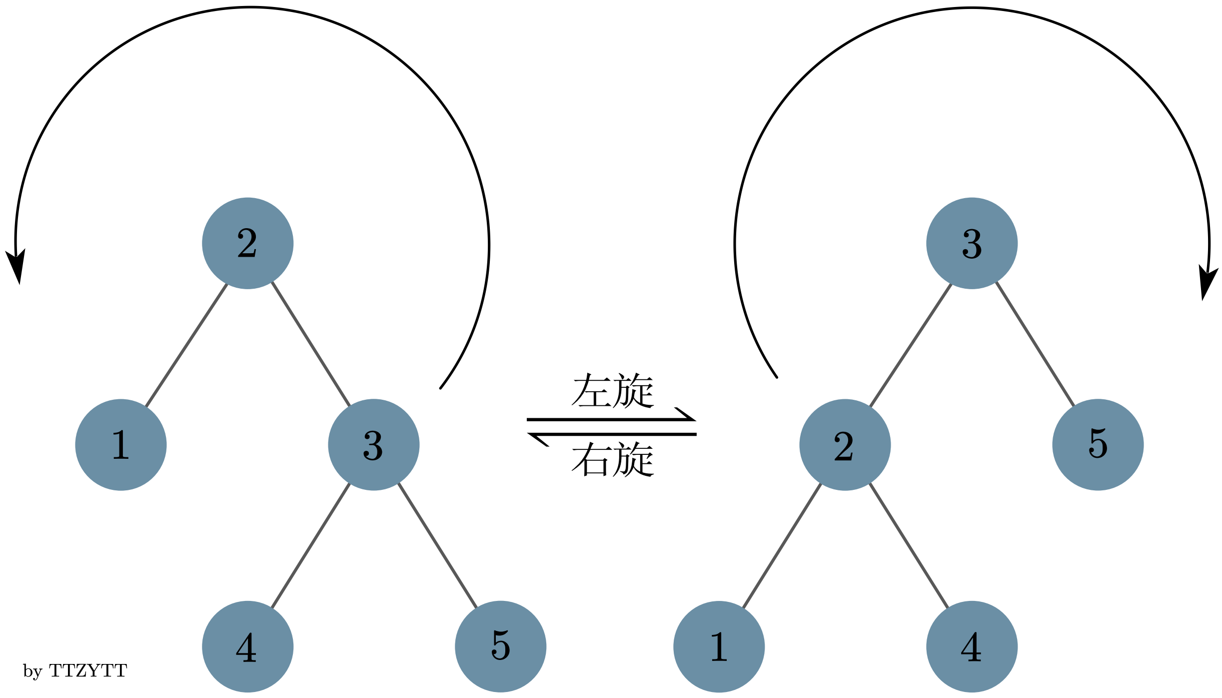


图 10.55

```
    tmp->ch[!dir] = cur;             // 让 C 的左子节点变成 A
    tmp->upd_siz(), cur->upd_siz();  // 更新大小信息
    cur = tmp;  // 最后把临时储存 C 树的变量赋值到当前根节点上 (注意 cur 是引用)
  }
```

插入

跟普通搜索树插入的过程没啥区别，但是需要在插的过程中通过旋转来维护树堆中堆的性质。

```
void _insert(Node *&cur, int val) {
  if (cur == nullptr) {
    // 没这个节点直接新建
    cur = new Node(val);
    return;
  } else if (val == cur->val) {
    // 如果有这个值相同的节点，就把重复数量加一
    cur->rep_cnt++;
    cur->siz++;
  } else if (val < cur->val) {
    // 维护搜索树性质，val 比当前节点小就插到左边，反之亦然
    _insert(cur->ch[0], val);
    if (cur->ch[0]->rank < cur->rank) {
      // 小根堆中，上面节点的优先级一定更小
      // 因为新插的左子节点比父节点小，现在需要让左子节点变成父节点
      _rotate(cur, RT);  // 注意前面的旋转性质，要把左子节点转上来，需要右旋
    }
    cur->upd_siz();  // 插入之后大小会变化，需要更新
  } else {
    _insert(cur->ch[1], val);
    if (cur->ch[1]->rank < cur->rank) {
      _rotate(cur, LF);
```

```
    }
    cur->upd_siz();
  }
}
```

删除

主要就是分类讨论，不同的情况有不同的处理方法，删完了树的大小会有变化，要注意更新。并且如果要删的节点有左子树和右子树，就要考虑删除之后让谁来当父节点（维护 rank 小的节点在上面）。

```
void _del(Node *&cur, int val) {
  if (val > cur->val) {
    _del(cur->ch[1], val);
    // 值更大就在右子树，反之亦然
    cur->upd_siz();
  } else if (val < cur->val) {
    _del(cur->ch[0], val);
    cur->upd_siz();
  } else {
    if (cur->rep_cnt > 1) {
      // 如果要删除的节点是重复的，可以直接把重复值减小
      cur->rep_cnt--, cur->siz--;
      return;
    }
    uint8_t state = 0;
    state |= (cur->ch[0] != nullptr);
    state |= ((cur->ch[1] != nullptr) << 1);
    // 00 都无，01 有左无右，10，无左有右，11 都有
    Node *tmp = cur;
    switch (state) {
      case 0:
        delete cur;
        cur = nullptr;
        // 没有任何子节点，就直接把这个节点删了
        break;
      case 1:  // 有左无右
        cur = tmp->ch[0];
        // 把根变成左儿子，然后把原来的根节删了，注意这里的 tmp 是从 cur
        // 复制的，而 cur 是引用
        delete tmp;
        break;
      case 2:  // 有右无左
        cur = tmp->ch[1];
        delete tmp;
        break;
      case 3:
        rot_type dir = cur->ch[0]->rank < cur->ch[1]->rank
                           ? RT
                           : LF;  // dir 是 rank 更小的那个儿子
        _rotate(cur, dir);  // 这里的旋转可以把优先级更小的儿子转上去，rt 是 0，
                            // 而 lf 是 1，刚好跟实际的子树下标反过来
        _del(
            cur->ch[!dir],
            val);  // 旋转完成后原来的根节点就在旋方向那边，所以需要
                   // 继续把这个原来的根节点删掉
```

```
                    // 如果说要删的这个节点是在整个树的「上层的」，那我们会一直通过这
                    // 这里的旋转操作，把它转到没有子树了（或者只有一个），再删掉它。
         cur->upd_siz();
         // 删除会造成大小改变
         break;
      }
    }
}
```

根据值查询排名

操作含义：查询以 cur 为根节点的子树中，val 这个值的大小的排名（该子树中小于 val 的节点的个数 + 1）

```
int _query_rank(Node *cur, int val) {
  int less_siz = cur->ch[0] == nullptr ? 0 : cur->ch[0]->siz;
  // 这个树中小于 val 的节点的数量
  if (val == cur->val)
    // 如果这个节点就是要查的节点
    return less_siz + 1;
  else if (val < cur->val) {
    if (cur->ch[0] != nullptr)
      return _query_rank(cur->ch[0], val);
    else
      return 1;  // 如果左子树是空的，说比最小的节点还要小，那这个数字就是最小的
  } else {
    if (cur->ch[1] != nullptr)
      // 如果要查的值比这个节点大，那这个节点的左子树以及这个节点自身肯定都比要查
的值小
      // 所以要加上这两个值，再加上往右边找的结果
      // （以右子树为根的子树中，val 这个值的大小的排名）
      return less_siz + cur->rep_cnt + _query_rank(cur->ch[1], val);
    else
      return cur->siz + 1;
    // 没有右子树的话直接整个树 + 1 相当于 less_siz + cur->rep_cnt + 1
  }
}
```

根据排名查询值

要根据排名查询值，我们首先要知道如何判断要查的节点在树的哪个部分：

以下是一个判断方法的表：

左子树	根节点/当前节点	右子树
排名 ≤ 左子树的大小	排名 > 左子树的大小，并且 ≤ 左子树的大小 + 根节点的重复次数	排名 > 左子树的大小 + 根节点的重复次数

注意如果在右子树，递归的时候需要对原来的 `rank` 进行处理。递归的时候就相当去查，在右子树中为这个排名的值，为了把排名转换成基于右子树的，需要把原来的 `rank` 减去左子树的大小和根节点的重复次数。

可以把所有节点想象成一个排好序的数组，或者数轴（如下），

```
1 -> | 左子树的节点 | 根节点 | 右子树的节点 | -> n
                          ^
                          要查的排名
```

```
                    转换成基于右子树的排名
1 -> | 右子树的节点 | -> n
      ^
      要查的排名
```

这里的转换方法就是直接把排名减去左子树的大小和根节点的重复数量。

```
int _query_val(Node *cur, int rank) {
  // 查询树中第 rank 大的节点的值
  int less_siz = cur->ch[0] == nullptr ? 0 : cur->ch[0]->siz;
  // less siz 是左子树的大小
  if (rank <= less_siz)
    return _query_val(cur->ch[0], rank);
  else if (rank <= less_siz + cur->rep_cnt)
    return cur->val;
  else
    return _query_val(cur->ch[1], rank - less_siz - cur->rep_cnt);  // 见前文
}
```

查询第一个比 val 小的节点

注意这里使用了一个类中的全局变量，q_prev_tmp。

这个值是只有在 val 比当前节点值大的时候才会被更改的，所以返回这个变量就是返回 val 最后一次比当前节点的值大，之后就是更小了。

```
int _query_prev(Node *cur, int val) {
  if (val <= cur->val) {
    // 还是比 val 大，所以往左子树找
    if (cur->ch[0] != nullptr) return _query_prev(cur->ch[0], val);
  } else {
    // 只有能进到这个 else 里，才会更新 q_prev_tmp 的值
    q_prev_tmp = cur->val;
    // 当前节点已经比 val，小了，但是不确定是否是最大的，所以要到右子树继续找
    if (cur->ch[1] != nullptr) _query_prev(cur->ch[1], val);
    // 接下来的递归可能不会更改 q_prev_tmp
    // 了，那就直接返回这个值，总之返回的就是最后一次进到这个 else 中的
    // cur->val
    return q_prev_tmp;
  }
  return -1145;
}
```

查询第一个比 val 大的节点

跟前一个很相似，只是大于小于号换了一下。

```
int _query_nex(Node *cur, int val) {
  if (val >= cur->val) {
    if (cur->ch[1] != nullptr) return _query_nex(cur->ch[1], val);
  } else {
    q_nex_tmp = cur->val;
    if (cur->ch[0] != nullptr) _query_nex(cur->ch[0], val);
    return q_nex_tmp;
  }
  return -1145;
```

```
}
```

无旋 treap

无旋 treap 的操作方式使得它天生支持维护序列、可持久化等特性。

无旋 treap 又称分裂合并 treap。它仅有两种核心操作，即为**分裂**与**合并**。通过这两种操作，在很多情况下可以比旋转 treap 更方便的实现别的操作。下面逐一介绍这两种操作。

> " 注释"
>
> 讲解无旋 treap 应当提到 **FHQ-Treap**（by 范浩强）。即可持久化，支持区间操作的无旋 Treap。更多内容请参照《范浩强谈数据结构》ppt。

分裂（split）

按值分裂

分裂过程接受两个参数：根指针 cur、关键值 key。结果为将根指针指向的 treap 分裂为两个 treap，第一个 treap 所有结点的值（val）小于等于 key，第二个 treap 所有结点的值大于 key。

该过程首先判断 key 是否小于 cur 的值，若小于，则说明 cur 及其右子树全部大于 key，属于第二个 treap。当然，也可能有一部分的左子树的值大于 key，所以还需要继续向左子树递归地分裂。对于大于 key 的那部分左子树，我们把它作为 cur 的左子树，这样，整个 cur 上的节点都是大于 key 的。

相应的，如果 key 大于等于 cur 的值，说明 cur 的整个左子树以及其自身都小于 key，属于分裂后的第一个 treap。并且，cur 的部分右子树也可能有部分小于 key，因此我们需要继续递归地分裂右子树。把小于 key 的那部分作为 cur 的右子树，这样，整个 cur 上的节点都小于 key。

下图展示了 cur 的值小于等于 key 时按值分裂的情况。[2]

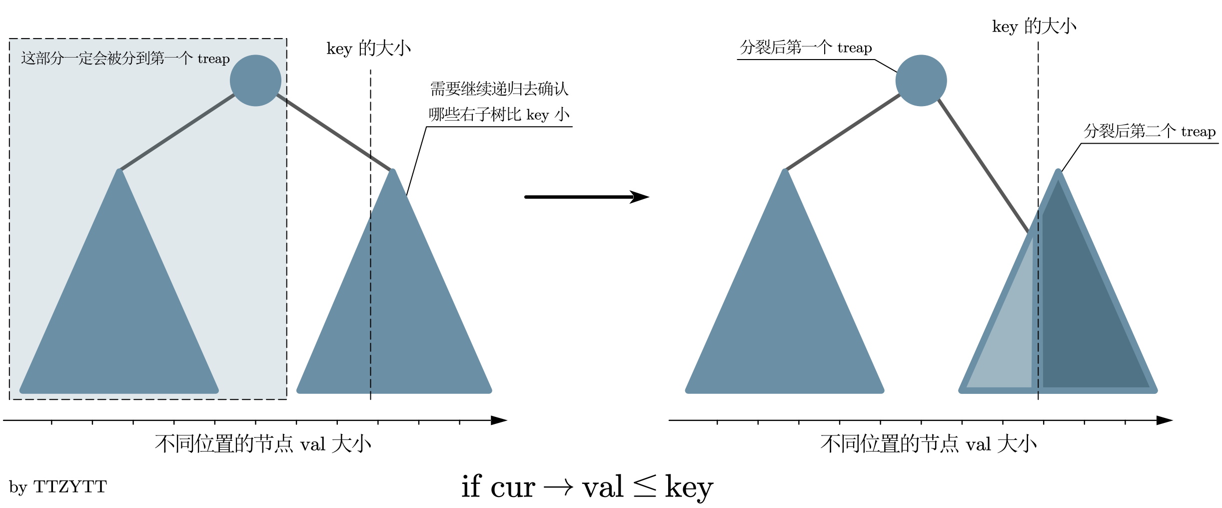


图 10.56

```
pair<Node *, Node *> split(Node *cur, int key) {
  if (cur == nullptr) return {nullptr, nullptr};
  if (cur->val <= key) {
    // cur 以及它的左子树一定属于分裂后的第一个树
    auto temp = split(cur->ch[1], key);
    // 但是它可能有部分右子树也比 key 小
    cur->ch[1] = temp.first;
    // 我们把小于 key 的那部分拿出来，作为 cur 的右子树，这样整个 cur 都是小于
    // key 的剩下的那部分右子树成为分裂后的第二个 treap
    cur->upd_siz();
```

```
    // 分裂过后树的大小会变化，需要更新
    return {cur, temp.second};
  } else {
    // 同上
    auto temp = split(cur->ch[0], key);
    cur->ch[0] = temp.second;
    cur->upd_siz();
    return {temp.first, cur};
  }
}
```

按排名分裂

比起按值分裂，这个操作更像是旋转 treap 中的根据排名（某个节点的排名是树中所有小于此节点值的节点的数量 +1）查询值：

此函数接受两个参数，节点指针 cur 和排名 rk，返回分裂后的三个 treap。

其中，第一个 treap 中每个节点的排名都小于 rk，第二个的排名等于 rk，并且第二个 treap 只有一个节点（不可能有多个等于的，如果有的话会增加 `Node` 结构体中的 `cnt`），第三个则是大于。

此操作的重点在于判断排名和 cur 相等的节点在树的哪个部分，这也是旋转 treap 根据排名查询值操作时的重要部分，在前文有非常详细的解释，这里不过多讲解。

并且，此操作的递归部分和按值分裂也非常相似，这里不赘述。

```
tuple<Node *, Node *, Node *> split_by_rk(Node *cur, int rk) {
  if (cur == nullptr) return {nullptr, nullptr, nullptr};
  int ls_siz = cur->ch[0] == nullptr ? 0 : cur->ch[0]->siz;
  if (rk <= ls_siz) {
    // 排名和 cur 相等的节点在左子树
    Node *l, *mid, *r;
    tie(l, mid, r) = split_by_rk(cur->ch[0], rk);
    cur->ch[0] = r;  // 返回的第三个 treap 中的排名都大于 rk
    // cur 的左子树被设成 r 后，整个 cur 中节点的排名都大于 rk
    cur->upd_siz();
    return {l, mid, cur};
  } else if (rk <= ls_siz + cur->cnt) {
    // 和 cur 相等的就是当前节点
    Node *lt = cur->ch[0];
    Node *rt = cur->ch[1];
    cur->ch[0] = cur->ch[1] = nullptr;
    // 分裂后第二个 treap 只有一个节点，所有要把它的子树设置为空
    return {lt, cur, rt};
  } else {
    // 排名和 cur 相等的节点在右子树
    // 递归过程同上
    Node *l, *mid, *r;
    tie(l, mid, r) = split_by_rk(cur->ch[1], rk - ls_siz - cur->cnt);
    cur->ch[1] = l;
    cur->upd_siz();
    return {cur, mid, r};
  }
}
```

合并（merge）

合并过程接受两个参数：左 treap 的根指针 u、右 treap 的根指针 v。必须满足 u 中所有结点的值小于等于 v 中所有结点的值。一般来说，我们合并的两个 treap 都是原来从一个 treap 中分裂出去的，所以不难满足 u 中所有节点

的值都小于 v

在旋转 treap 中，我们借助旋转操作来维护 $priority$ 符合堆的性质，同时旋转时还不能改变树的性质。在无旋 treap 中，我们用合并达到相同的效果。

因为两个 treap 已经有序，所以我们在合并的时候只需要考虑把哪个树「放在上面」，把哪个「放在下面」，也就是是需要判断将哪个一个树作为子树。显然，根据堆的性质，我们需要把 $priority$ 小的放在上面（这里采用小根堆）。

同时，我们还需要满足搜索树的性质，所以若 u 的根结点的 $priority$ 小于 v 的，那么 u 即为新根结点，并且 v 因为值比 u 更大，应与 u 的右子树合并；反之，则 v 作为新根结点，然后因为 u 的值比 v 小，与 v 的左子树合并。

```
Node *merge(Node *u, Node *v) {
  // 传进来的两个树的内部已经符合搜索树的性质了
  // 并且 u 内所有节点的值 < v 内所有节点的值
  // 所以在合并的时候需要维护堆的性质
  // 这里用的是小根堆
  if (u == nullptr && v == nullptr) return nullptr;
  if (u != nullptr && v == nullptr) return u;
  if (v != nullptr && u == nullptr) return v;

  if (u->prio < v->prio) {
    // u 的 prio 比较小, u 应该作为父节点
    u->ch[1] = merge(u->ch[1], v);
    // 因为 v 比 u 大, 所以把 v 作为 u 的右子树
    u->upd_siz();
    return u;
  } else {
    // v 比较小, v 应该作为父节点
    v->ch[0] = merge(u, v->ch[0]);
    // u 比 v 小, 所以递归时的参数是这样的
    v->upd_siz();
    return v;
  }
}
```

插入

在无旋 treap 中，插入，删除，根据值查询排名等基础操作既可以用普通二叉查找树的方法实现，也可以用分裂和合并来实现。通常来说，使用分裂和合并来实现更加简洁，但是速度会慢一点[3]。为了帮助更好的理解无旋 treap，下面的操作全部使用分裂和合并实现。

在实现插入操作时，我们利用了分裂操作的一些性质。也就是值小于等于 val 的节点会被分到第一个 treap。

所以，假设我们根据 val 分裂当前这个 treap。会有下面两棵树，并符合以下条件：

$$T_1 \le val$$
$$T_2 > val$$

其中 T_1 表示分裂后所有被分到第一个 treap 的节点的集合，T_2 则是第二个。

如果我们再按照 $val-1$ 继续分裂 T_1，那么会产生下面两棵树，并符合以下条件：

$$T_{1\ \text{left}} \le val - 1$$
$$T_{1\ \text{right}} > val - 1 \quad \& \quad T_{1\ \text{right}} \le val$$

其中 $T_{1\ \text{left}}$ 表示 T_1 分裂后所有被分到第一个 treap 的节点的集合，$T_{1\ \text{right}}$ 则是第二个。并且上面的式子中，后半部分的 $\&\ \ T_{1\ \text{right}} \le val$ 来自于 T_1 所符合的条件 $T_1 \le val$。

不难发现，只要 val 和节点的值是一个整数（大多数使用场景下会使用整数）那么符合 $T_{1\ \text{right}}$ 条件的节点只有一个，也就是值等于 val 的节点。

在插入时，如果我们发现符合 $T_{1\ \text{right}}$ 的节点存在，那就可以直接增加重复次数，否则，就新开一个节点。

注意把树分裂好了还需要用合并操作把它「粘」回去，这样下次还能继续使用。并且，还需要注意合并操作的参数顺序是有要求的，第一个树的所有节点的值都需要小于第二个。

```
void insert(int val) {
  auto temp = split(root, val);
  // 根据 val 的值把整个树分成两个
  // 注意 split 的实现，等于 val 的子树是在左子树的
  auto l_tr = split(temp.first, val - 1);
  // l_tr 的左子树 <= val - 1，如果有 = val 的节点，那一定在右子树
  Node *new_node;
  if (l_tr.second == nullptr) {
    // 没有这个节点就新开，否则直接增加重复次数。
    new_node = new Node(val);
  } else {
    l_tr.second->cnt++;
    l_tr.second->upd_siz();
  }
  Node *l_tr_combined =
      merge(l_tr.first, l_tr.second == nullptr ? new_node : l_tr.second);
  // 合并 T_1 left 和 T_1 right
  root = merge(l_tr_combined, temp.second);
  // 合并 T_1 和 T_2
}
```

删除

删除操作也使用和插入操作相似的方法，找到值和 val 相等的节点，并且删除它。

```
void del(int val) {
  auto temp = split(root, val);
  auto l_tr = split(temp.first, val - 1);
  if (l_tr.second->cnt > 1) {
    // 如果这个节点的重复次数大于 1，减小即可
    l_tr.second->cnt--;
    l_tr.second->upd_siz();
    l_tr.first = merge(l_tr.first, l_tr.second);
  } else {
    if (temp.first == l_tr.second) {
      // 有可能整个 T_1 只有这个节点，所以也需要把这个点设成 null 来标注已经删除
      temp.first = nullptr;
    }
    delete l_tr.second;
    l_tr.second = nullptr;
  }
  root = merge(l_tr.first, temp.second);
}
```

根据值查询排名

排名是比这个值小的节点的数量 $+1$，所以我们根据 $val-1$ 分裂当前树，那么分裂后的第一个树就符合：

$$T_1 \leq val - 1$$

如果树的值和 val 为整数，那么 T_1 就包含了所有值小于 val 的节点。

```cpp
int qrank_by_val(Node* cur, int val) {
  auto temp = split(cur, val - 1);
  int ret = (temp.first == nullptr ? 0 : temp.first->siz) + 1;  // 根据定义 + 1
  root = merge(temp.first, temp.second);  // 拆好了再粘回去
  return ret;
}
```

根据排名查询值

调用 `split_by_rk()` 函数后，会返回分裂好的三个 treap，其中第二个只包含一个节点，它的排名等于 rk，所以我们直接返回这个节点的 val。

```cpp
int qval_by_rank(Node *cur, int rk) {
  Node *l, *mid, *r;
  tie(l, mid, r) = split_by_rk(cur, rk);
  int ret = mid->val;
  root = merge(merge(l, mid), r);
  return ret;
}
```

查询第一个比 val 小的节点

可以把这个问题转化为，在比 val 小的所有节点中，找出排名最大的。我们根据 val 来分裂这个 treap，返回的第一个 treap 中的节点的值就全部小于 val，然后我们调用 `qval_by_rank()` 找出这个树中值最大的节点。

```cpp
int qprev(int val) {
  auto temp = split(root, val - 1);
  // temp.first 就是值小于 val 的子树
  int ret = qval_by_rank(temp.first, temp.first->siz);
  // 这里查询的是，所有小于 val 的节点里面，最大的那个的值
  root = merge(temp.first, temp.second);
  return ret;
}
```

查询第一个比 val 大的节点

和上个操作类似，可以把这个问题转化为，在比 val 大的所有节点中，找出排名最大的。那么根据 val 分裂后，返回的第二个 treap 中的所有节点的值就大于 val。

然后我们去查询这个树中排名为 1 的节点（也就是值最小的节点）的值，就可以成功查到第一个比 val 大的节点。

```cpp
int qnex(int val) {
  auto temp = split(root, val);
  int ret = qval_by_rank(temp.second, 1);
  // 查询所有大于 val 的子树里面，值最小的那个
  root = merge(temp.first, temp.second);
  return ret;
}
```

建树（build）

将一个有 n 个节点的序列 $\{a_n\}$ 转化为一棵 treap。

可以依次暴力插入这 n 个节点，每次插入一个权值为 v 的节点时，将整棵 treap 按照权值分裂成权值小于等于 v 的和权值大于 v 的两部分，然后新建一个权值为 v 的节点，将两部分和新节点按从小到大的顺序依次合并，单次插入时间复杂度 $O(\log n)$，总时间复杂度 $O(n\log n)$。

在某些题目内，可能会有多次插入一段有序序列的操作，这是就需要在 $O(n)$ 的时间复杂度内完成建树操作。

方法一：在递归建树的过程中，每次选取当前区间的中点作为该区间的树根，并对每个节点钦定合适的优先值，使得新树满足堆的性质。这样能保证树高为 $O(\log n)$。

方法二：在递归建树的过程中，每次选取当前区间的中点作为该区间的树根，然后给每个节点一个随机优先级。这样能保证树高为 $O(\log n)$，但不保证其满足堆的性质。这样也是正确的，因为无旋式 treap 的优先级是用来使 `merge` 操作更加随机一点，而不是用来保证树高的。

方法三：观察到 treap 是笛卡尔树，利用笛卡尔树的 $O(n)$ 建树方法即可，用单调栈维护右链即可。

无旋 treap 的区间操作

建树

无旋 treap 相比旋转 treap 的一大好处就是可以实现各种区间操作，下面我们以文艺平衡树的 模板题[6] 为例，介绍 treap 的区间操作。

> 您需要写一种数据结构（可参考题目标题），来维护一个有序数列。
>
> 其中需要提供以下操作：翻转一个区间，例如原有序序列是 5 4 3 2 1，翻转区间是 $[2,4]$ 的话，结果是 5 2 3 4 1。对于 100% 的数据，$1 \leq n$（初始区间长度）m（翻转次数）$\leq 1e5$

在这道题目中，我们需要实现的是区间翻转，那么我们首先需要考虑如何建树，建出来的树需要是初始的区间。

我们只需要把区间的下标依次插入 treap 中，这样在中序遍历（先遍历左子树，然后当前节点，最后右子树）时，就可以得到这个区间[4-1]。

我们知道在朴素的二叉查找树中按照递增的顺序插入节点，建出来的树是一个长链，按照中序遍历，自然可以得到这个区间。

如上图，按照 1 2 3 4 5 的顺序给朴素搜索树插入节点，中序遍历时，得到的也是 1 2 3 4 5。

但是在 treap 中，按增序插入节点后，在合并操作时还会根据 *priority* 调整树的结构，在这样的情况下，如何确保中序遍历一定能正确的输出呢?

可以参考 笛卡尔树的单调栈建树方法 来理解这个问题。

设新插入的节点为 u。

首先，因为是递增地插入节点，每一个新插入的节点肯定会被连接到 treap 的右链（即从根结点一直往右子树走，经过的结点形成的链）上。

从根节点开始，右链上的节点的 *priority* 是递增的（小根堆）。那我们可以找到右链上第一个 *priority* 大于 u 的节点，我们叫这个节点 v，并把这个节点换成 u。

因为 u 一定大于这个树上其他的全部节点，我们需要把 v 以及它的子树作为 u 的左子树。并且此时 u 没有右子树。

可以发现，中序遍历时 u 一定是最后一个被遍历到的（因为 u 是右链中的最后一个，而中序遍历中，右子树是最后被遍历到的）。

下图是一个 treap 根据递增顺序插入 $1 \sim 5$ 号节点时，插入 5 号节点时的变化，可以用这张图更好的理解按照增序插入的过程。

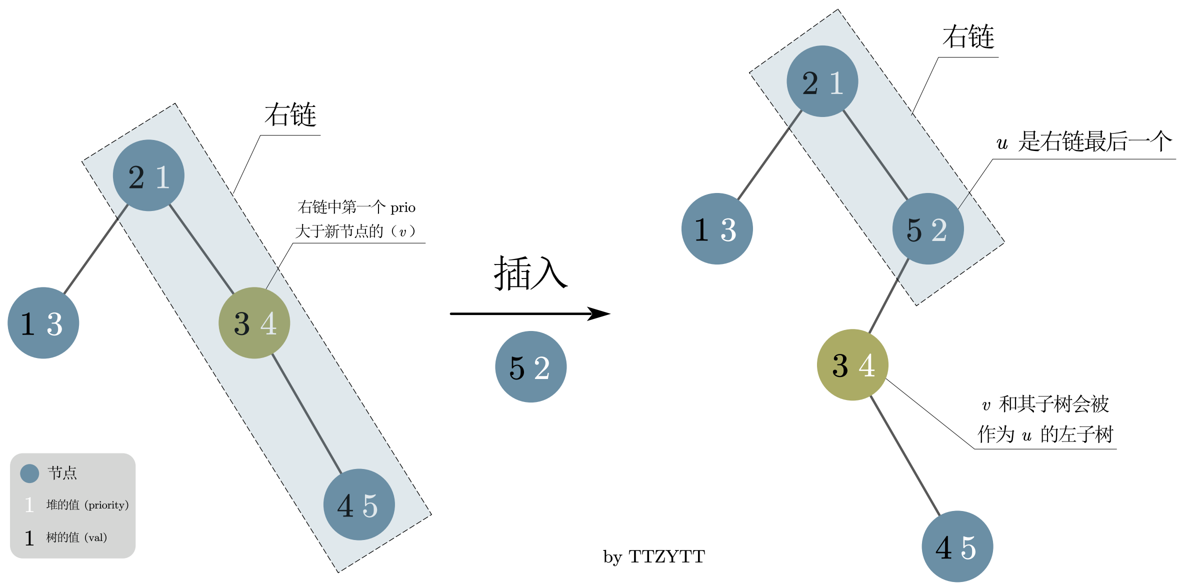
区间翻转

翻转 $[l,r]$ 这个区间时，基本思路是将树分裂成 $[1,l-1]$，$[l,r]$，$[r+1,n]$ 三个区间，再对中间的 $[l,r]$ 进行翻转[4-2]。

翻转的具体操作是把区间内的子树的每一个左，右子节点交换位置。如下图就展示了翻转上图中 treap 的 $[3,4]$ 和 $[3,5]$ 区间后的 treap。

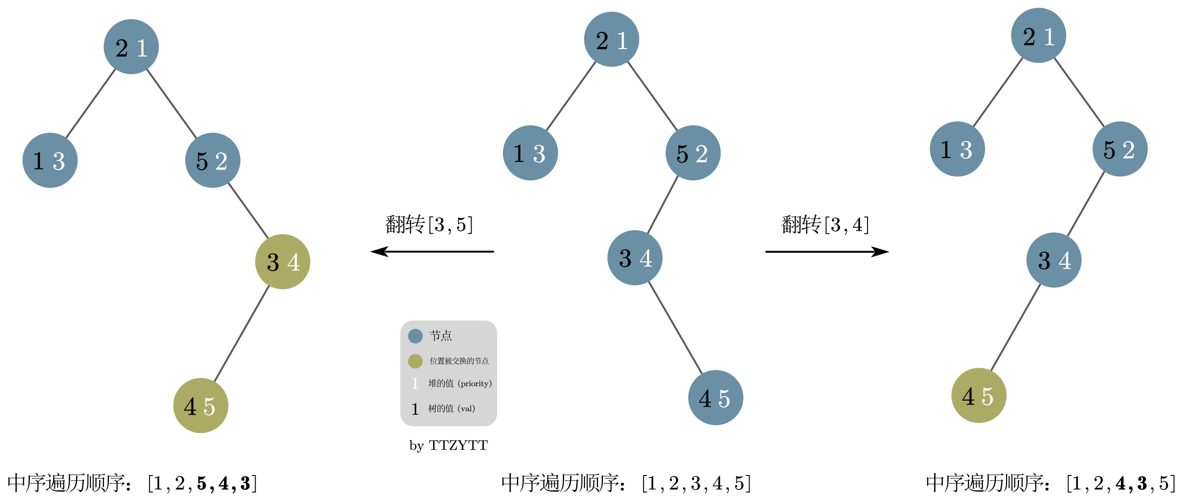
注意如果按照这个方法翻转，那么每次翻转 $[l,r]$ 区间时，就会有 $r-l$ 个节点会被交换位置，这样频繁的操作显然不能满足 10^5 的数据范围，其 $O(n \times \log_2 n)$ 的单次翻转复杂度甚至不如暴力（因为我们除了需要花线性时间交换节点外，还需要在树中花费 $O(\log_2 n)$ 的时间找到需要交换的节点）。

再观察题目要求，可以发现因为只需要最后输出操作完的区间，所以并不需要每次都真的去交换。如此一来，便可以使用线段树中常用的懒标记（lazy tag）来优化复杂度。交换时，只需要在父节点打上标记，代表这个子树下的每



右链
右链中第一个 prio
大于新节点的（v）
插入
右链
u 是右链最后一个
v 和其子树会被
作为 u 的左子树
节点
堆的值 (priority)
树的值 (val)
by TTZYTT

图 10.57



翻转[3,5]
翻转[3,4]
节点
位置被交换的节点
堆的值 (priority)
树的值 (val)
by TTZYTT
中序遍历顺序：[1,2,5,4,3]
中序遍历顺序：[1,2,3,4,5]
中序遍历顺序：[1,2,4,3,5]

图 10.58

个左右子节点都需要交换就行了。

在线段树中，我们一般在更新和查询时下传懒标记。这是因为，在更新和查询时，我们想要更新/查询的范围不一定和懒标记代表的范围重合，所以要先下传标记，确保查到和更新后的值是正确的。

在无旋 treap 中也是一样。具体操作时我们会把 treap 分裂成前文讲到的三个树，然后给中间的树打上懒标记后合并这三棵树。因为我们想要翻转的区间和懒标记代表的区间不一定重合，所以要在分裂时下传标记。并且，分裂和合并操作会造成每个节点及其懒标记所代表的节点发生变动，所以也需要在合并前下传懒标记。

换句话说，是当树的结构发生改变的时候，当我们进行分裂或合并操作时需要改变某一个点的左右儿子信息时之前，应该下放标记，而非之后，因为懒标记是需要下传给儿子节点的，但更改左右儿子信息之后若懒标记还未下放，则懒标记就丢失了下放的对象。[5]

以下为代码讲解，代码参考了[4-3]。

因为区间操作中大部分操作都和普通的无旋 treap 相同，所以这里只讲解和普通无旋 treap 不同的地方。

下传标记

需要注意这里的懒标记代表需要把这个树中的每一个子节点交换位置。所以如果当前节点的子节点也有懒标记，那两次翻转就抵消了。如果子节点不需要翻转，那么这个懒标记就需要继续被下传到子节点上。

```
// 这里这个 pushdown 是 Node 类的成员函数，其中 to_rev 是懒标记
void pushdown() {
  swap(ch[0], ch[1]);
  if (ch[0] != nullptr) ch[0]->to_rev ^= 1;
  if (ch[1] != nullptr) ch[1]->to_rev ^= 1;
  to_rev = false;
}

void check_tag() {
  if (to_rev) pushdown();
}
```

分裂

注意在这个题目中，因为翻转操作，treap 中的 val 会不符合二叉搜索树的性质（见区间翻转部分的图），所以我们不能根据 val 来判断应该往左子树还是右子树递归。

所以这里的分裂跟普通无旋 treap 中的按排名分裂更相似，是根据当前树的大小判断往左还是右子树递归的，换言之，我们是按照开始时这个节点在树中的位置来判断的。

返回的第一个 treap 中节点的排名全部小于等于 sz，而第二个 treap 中节点的排名则全部大于 sz。

```
#define siz(_) (_ == nullptr ? 0 : _->siz)

pair<Node*, Node*> split(Node* cur, int sz) {
  // 按照树的大小判断
  if (cur == nullptr) return {nullptr, nullptr};
  cur->check_tag();
  // 分裂前先下传
  if (sz <= siz(cur->ch[0])) {
    auto temp = split(cur->ch[0], sz);
    cur->ch[0] = temp.second;
    cur->upd_siz();
    return {temp.first, cur};
  } else {
    auto temp =
        split(cur->ch[1],
              sz - siz(cur->ch[0]) -
```

```
                    1);  // 这里的转换在有旋 treap 的 「根据排名查询值有讲」
      cur->ch[1] = temp.first;
      cur->upd_siz();
      return {cur, temp.second};
    }
  }
```

合并

唯一需要注意的是在合并前下传懒标记

```
Node *merge(Node *sm, Node *bg) {
  // small, big
  if (sm == nullptr && bg == nullptr) return nullptr;
  if (sm != nullptr && bg == nullptr) return sm;
  if (sm == nullptr && bg != nullptr) return bg;
  sm->check_tag(), bg->check_tag();
  if (sm->prio < bg->prio) {
    sm->ch[1] = merge(sm->ch[1], bg);
    sm->upd_siz();
    return sm;
  } else {
    bg->ch[0] = merge(sm, bg->ch[0]);
    bg->upd_siz();
    return bg;
  }
}
```

区间翻转

和前面介绍的一样，分裂出 $[1, l-1]$，$[l, r]$，$[r+1, n]$ 三个区间，然后对中间的区间打上标记后再合并。

```
void seg_rev(int l, int r) {
  // 这里的 less 和 more 是相对于 l 的
  auto less = split(root, l - 1);
  // 所有小于等于 l - 1 的会在 less 的左子树
  auto more = split(less.second, r - l + 1);
  // 从 l 开始的前 r - l + 1 个元素的区间
  more.first->to_rev = true;
  root = merge(less.first, merge(more.first, more.second));
}
```

中序遍历打印

要注意在打印时要下传标记。

```
void print(Node* cur) {
  if (cur == nullptr) return;
  cur->check_tag();
  // 中序遍历 -> 先左子树，再自己，最后右子树
  print(cur->ch[0]);
  cout << cur->val << " ";
  print(cur->ch[1]);
}
```

完整代码

旋转 treap

指针实现

" 完整代码"

以下是前文讲解的代码的完整版本，是普通平衡树的模板代码。

```
// author: (ttzytt)[ttzytt.com]
#include <bits/stdc++.h>
using namespace std;

struct Node {
  Node *ch[2];
  int val, rank;
  int rep_cnt;
  int siz;

  Node(int val) : val(val), rep_cnt(1), siz(1) {
    ch[0] = ch[1] = nullptr;
    rank = rand();
  }

  void upd_siz() {
    siz = rep_cnt;
    if (ch[0] != nullptr) siz += ch[0]->siz;
    if (ch[1] != nullptr) siz += ch[1]->siz;
  }
};

class Treap {
 private:
  Node *root;

  enum rot_type { LF = 1, RT = 0 };

  int q_prev_tmp = 0, q_nex_tmp = 0;

  void _rotate(Node *&cur, rot_type dir) {  // 0 为右旋，1 为左旋
    Node *tmp = cur->ch[dir];
    cur->ch[dir] = tmp->ch[!dir];
    tmp->ch[!dir] = cur;
    tmp->upd_siz(), cur->upd_siz();
    cur = tmp;
  }

  void _insert(Node *&cur, int val) {
    if (cur == nullptr) {
      cur = new Node(val);
      return;
    } else if (val == cur->val) {
      cur->rep_cnt++;
      cur->siz++;
    } else if (val < cur->val) {
```

```
      _insert(cur->ch[0], val);
      if (cur->ch[0]->rank < cur->rank) {
        _rotate(cur, RT);
      }
      cur->upd_siz();
    } else {
      _insert(cur->ch[1], val);
      if (cur->ch[1]->rank < cur->rank) {
        _rotate(cur, LF);
      }
      cur->upd_siz();
    }
  }

  void _del(Node *&cur, int val) {
    if (val > cur->val) {
      _del(cur->ch[1], val);
      cur->upd_siz();
    } else if (val < cur->val) {
      _del(cur->ch[0], val);
      cur->upd_siz();
    } else {
      if (cur->rep_cnt > 1) {
        cur->rep_cnt--, cur->siz--;
        return;
      }
      uint8_t state = 0;
      state |= (cur->ch[0] != nullptr);
      state |= ((cur->ch[1] != nullptr) << 1);
      // 00 都无，01 有左无右，10，无左有右，11 都有
      Node *tmp = cur;
      switch (state) {
        case 0:
          delete cur;
          cur = nullptr;
          break;
        case 1:  // 有左无右
          cur = tmp->ch[0];
          delete tmp;
          break;
        case 2:  // 有右无左
          cur = tmp->ch[1];
          delete tmp;
          break;
        case 3:
          rot_type dir = cur->ch[0]->rank < cur->ch[1]->rank ? RT : LF;
          _rotate(cur, dir);
          _del(cur->ch[!dir], val);
          cur->upd_siz();
          break;
      }
    }
  }
```

```
  int _query_rank(Node *cur, int val) {
    int less_siz = cur->ch[0] == nullptr ? 0 : cur->ch[0]->siz;
    if (val == cur->val)
      return less_siz + 1;
    else if (val < cur->val) {
      if (cur->ch[0] != nullptr)
        return _query_rank(cur->ch[0], val);
      else
        return 1;
    } else {
      if (cur->ch[1] != nullptr)
        return less_siz + cur->rep_cnt + _query_rank(cur->ch[1], val);
      else
        return cur->siz + 1;
    }
  }

  int _query_val(Node *cur, int rank) {
    int less_siz = cur->ch[0] == nullptr ? 0 : cur->ch[0]->siz;
    if (rank <= less_siz)
      return _query_val(cur->ch[0], rank);
    else if (rank <= less_siz + cur->rep_cnt)
      return cur->val;
    else
      return _query_val(cur->ch[1], rank - less_siz - cur->rep_cnt);
  }

  int _query_prev(Node *cur, int val) {
    if (val <= cur->val) {
      if (cur->ch[0] != nullptr) return _query_prev(cur->ch[0], val);
    } else {
      q_prev_tmp = cur->val;
      if (cur->ch[1] != nullptr) _query_prev(cur->ch[1], val);
      return q_prev_tmp;
    }
    return -1145;
  }

  int _query_nex(Node *cur, int val) {
    if (val >= cur->val) {
      if (cur->ch[1] != nullptr) return _query_nex(cur->ch[1], val);
    } else {
      q_nex_tmp = cur->val;
      if (cur->ch[0] != nullptr) _query_nex(cur->ch[0], val);
      return q_nex_tmp;
    }
    return -1145;
  }

 public:
  void insert(int val) { _insert(root, val); }

  void del(int val) { _del(root, val); }
```

```
  int query_rank(int val) { return _query_rank(root, val); }

  int query_val(int rank) { return _query_val(root, rank); }

  int query_prev(int val) { return _query_prev(root, val); }

  int query_nex(int val) { return _query_nex(root, val); }
};

Treap tr;

int main() {
  srand(0);
  int t;
  scanf("%d", &t);
  while (t--) {
    int mode;
    int num;
    scanf("%d%d", &mode, &num);
    switch (mode) {
      case 1:
        tr.insert(num);
        break;
      case 2:
        tr.del(num);
        break;
      case 3:
        printf("%d\n", tr.query_rank(num));
        break;
      case 4:
        printf("%d\n", tr.query_val(num));
        break;
      case 5:
        printf("%d\n", tr.query_prev(num));
        break;
      case 6:
        printf("%d\n", tr.query_nex(num));
        break;
    }
  }
}
```

数组实现

以下是 bzoj 普通平衡树模板代码，使用数组实现。

" 完整代码"

```
#include <algorithm>
#include <cstdio>
#include <iostream>

#define maxn 100005
#define INF (1 << 30)
```

```cpp
int n;

struct treap {  // 直接维护成数据结构，可以直接用
  int l[maxn], r[maxn], val[maxn], rnd[maxn], size_[maxn], w[maxn];
  int sz, ans, rt;

  void pushup(int x) { size_[x] = size_[l[x]] + size_[r[x]] + w[x]; }

  void lrotate(int &k) {
    int t = r[k];
    r[k] = l[t];
    l[t] = k;
    size_[t] = size_[k];
    pushup(k);
    k = t;
  }

  void rrotate(int &k) {
    int t = l[k];
    l[k] = r[t];
    r[t] = k;
    size_[t] = size_[k];
    pushup(k);
    k = t;
  }

  void insert(int &k, int x) {  // 插入
    if (!k) {
      sz++;
      k = sz;
      size_[k] = 1;
      w[k] = 1;
      val[k] = x;
      rnd[k] = rand();
      return;
    }
    size_[k]++;
    if (val[k] == x) {
      w[k]++;
    } else if (val[k] < x) {
      insert(r[k], x);
      if (rnd[r[k]] < rnd[k]) lrotate(k);
    } else {
      insert(l[k], x);
      if (rnd[l[k]] < rnd[k]) rrotate(k);
    }
  }

  bool del(int &k, int x) {  // 删除节点
    if (!k) return false;
    if (val[k] == x) {
      if (w[k] > 1) {
        w[k]--;
        size_[k]--;
```

```
      return true;
    }
    if (l[k] == 0 || r[k] == 0) {
      k = l[k] + r[k];
      return true;
    } else if (rnd[l[k]] < rnd[r[k]]) {
      rrotate(k);
      return del(k, x);
    } else {
      lrotate(k);
      return del(k, x);
    }
  } else if (val[k] < x) {
    bool succ = del(r[k], x);
    if (succ) size_[k]--;
    return succ;
  } else {
    bool succ = del(l[k], x);
    if (succ) size_[k]--;
    return succ;
  }
}

int queryrank(int k, int x) {
  if (!k) return 0;
  if (val[k] == x)
    return size_[l[k]] + 1;
  else if (x > val[k]) {
    return size_[l[k]] + w[k] + queryrank(r[k], x);
  } else
    return queryrank(l[k], x);
}

int querynum(int k, int x) {
  if (!k) return 0;
  if (x <= size_[l[k]])
    return querynum(l[k], x);
  else if (x > size_[l[k]] + w[k])
    return querynum(r[k], x - size_[l[k]] - w[k]);
  else
    return val[k];
}

void querypre(int k, int x) {
  if (!k) return;
  if (val[k] < x)
    ans = k, querypre(r[k], x);
  else
    querypre(l[k], x);
}

void querysub(int k, int x) {
  if (!k) return;
  if (val[k] > x)
```

```
      ans = k, querysub(l[k], x);
    else
      querysub(r[k], x);
  }
} T;

int main() {
  srand(123);
  scanf("%d", &n);
  int opt, x;
  for (int i = 1; i <= n; i++) {
    scanf("%d%d", &opt, &x);
    if (opt == 1)
      T.insert(T.rt, x);
    else if (opt == 2)
      T.del(T.rt, x);
    else if (opt == 3) {
      printf("%d\n", T.queryrank(T.rt, x));
    } else if (opt == 4) {
      printf("%d\n", T.querynum(T.rt, x));
    } else if (opt == 5) {
      T.ans = 0;
      T.querypre(T.rt, x);
      printf("%d\n", T.val[T.ans]);
    } else if (opt == 6) {
      T.ans = 0;
      T.querysub(T.rt, x);
      printf("%d\n", T.val[T.ans]);
    }
  }
  return 0;
}
```

无旋 treap

指针实现

" 完整代码"

以下是前文讲解的代码的完整版本，是普通平衡树的模板代码。

```
// author: (ttzytt)[ttzytt.com]
#include <bits/stdc++.h>
using namespace std;

struct Node {
  Node *ch[2];
  int val, prio;
  int cnt;
  int siz;

  Node(int _val) : val(_val), cnt(1), siz(1) {
    ch[0] = ch[1] = nullptr;
    prio = rand();
  }
```

```cpp
  Node(Node *_node) {
    val = _node->val, prio = _node->prio, cnt = _node->cnt, siz = _node->siz;
  }

  void upd_siz() {
    siz = cnt;
    if (ch[0] != nullptr) siz += ch[0]->siz;
    if (ch[1] != nullptr) siz += ch[1]->siz;
  }
};

struct none_rot_treap {
#define _3 second.second
#define _2 second.first
  Node *root;

  pair<Node *, Node *> split(Node *cur, int key) {
    if (cur == nullptr) return {nullptr, nullptr};
    if (cur->val <= key) {
      auto temp = split(cur->ch[1], key);
      cur->ch[1] = temp.first;
      cur->upd_siz();
      return {cur, temp.second};
    } else {
      auto temp = split(cur->ch[0], key);
      cur->ch[0] = temp.second;
      cur->upd_siz();
      return {temp.first, cur};
    }
  }

  tuple<Node *, Node *, Node *> split_by_rk(Node *cur, int rk) {
    if (cur == nullptr) return {nullptr, nullptr, nullptr};
    int ls_siz = cur->ch[0] == nullptr ? 0 : cur->ch[0]->siz;
    if (rk <= ls_siz) {
      Node *l, *mid, *r;
      tie(l, mid, r) = split_by_rk(cur->ch[0], rk);
      cur->ch[0] = r;
      cur->upd_siz();
      return {l, mid, cur};
    } else if (rk <= ls_siz + cur->cnt) {
      Node *lt = cur->ch[0];
      Node *rt = cur->ch[1];
      cur->ch[0] = cur->ch[1] = nullptr;
      return {lt, cur, rt};
    } else {
      Node *l, *mid, *r;
      tie(l, mid, r) = split_by_rk(cur->ch[1], rk - ls_siz - cur->cnt);
      cur->ch[1] = l;
      cur->upd_siz();
      return {cur, mid, r};
    }
  }
```

```
Node *merge(Node *u, Node *v) {
  if (u == nullptr && v == nullptr) return nullptr;
  if (u != nullptr && v == nullptr) return u;
  if (v != nullptr && u == nullptr) return v;
  if (u->prio < v->prio) {
    u->ch[1] = merge(u->ch[1], v);
    u->upd_siz();
    return u;
  } else {
    v->ch[0] = merge(u, v->ch[0]);
    v->upd_siz();
    return v;
  }
}

void insert(int val) {
  auto temp = split(root, val);
  auto l_tr = split(temp.first, val - 1);
  Node *new_node;
  if (l_tr.second == nullptr) {
    new_node = new Node(val);
  } else {
    l_tr.second->cnt++;
    l_tr.second->upd_siz();
  }
  Node *l_tr_combined =
      merge(l_tr.first, l_tr.second == nullptr ? new_node : l_tr.second);
  root = merge(l_tr_combined, temp.second);
}

void del(int val) {
  auto temp = split(root, val);
  auto l_tr = split(temp.first, val - 1);
  if (l_tr.second->cnt > 1) {
    l_tr.second->cnt--;
    l_tr.second->upd_siz();
    l_tr.first = merge(l_tr.first, l_tr.second);
  } else {
    if (temp.first == l_tr.second) {
      temp.first = nullptr;
    }
    delete l_tr.second;
    l_tr.second = nullptr;
  }
  root = merge(l_tr.first, temp.second);
}

int qrank_by_val(Node *cur, int val) {
  auto temp = split(cur, val - 1);
  int ret = (temp.first == nullptr ? 0 : temp.first->siz) + 1;
  root = merge(temp.first, temp.second);
  return ret;
}
```

```
  int qval_by_rank(Node *cur, int rk) {
    Node *l, *mid, *r;
    tie(l, mid, r) = split_by_rk(cur, rk);
    int ret = mid->val;
    root = merge(merge(l, mid), r);
    return ret;
  }

  int qprev(int val) {
    auto temp = split(root, val - 1);
    int ret = qval_by_rank(temp.first, temp.first->siz);
    root = merge(temp.first, temp.second);
    return ret;
  }

  int qnex(int val) {
    auto temp = split(root, val);
    int ret = qval_by_rank(temp.second, 1);
    root = merge(temp.first, temp.second);
    return ret;
  }
};

none_rot_treap tr;

int main() {
  srand(time(0));
  int t;
  scanf("%d", &t);
  while (t--) {
    int mode;
    int num;
    scanf("%d%d", &mode, &num);
    switch (mode) {
      case 1:
        tr.insert(num);
        break;
      case 2:
        tr.del(num);
        break;
      case 3:
        printf("%d\n", tr.qrank_by_val(tr.root, num));
        break;
      case 4:
        printf("%d\n", tr.qval_by_rank(tr.root, num));
        break;
      case 5:
        printf("%d\n", tr.qprev(num));
        break;
      case 6:
        printf("%d\n", tr.qnex(num));
        break;
    }
```

```
  }
}
```

无旋 treap 的区间操作

指针实现

" 完整代码"

以下是前文讲解的代码的完整版本，是文艺平衡树题目的模板代码。

```cpp
// author: (ttzytt)[ttzytt.com]
#include <bits/stdc++.h>
using namespace std;

// 参考：https://www.cnblogs.com/Equinox-Flower/p/10785292.html
struct Node {
  Node* ch[2];
  int val, prio;
  int cnt;
  int siz;
  bool to_rev = false;  // 需要把这个子树下的每一个节点都翻转过来

  Node(int _val) : val(_val), cnt(1), siz(1) {
    ch[0] = ch[1] = nullptr;
    prio = rand();
  }

  int upd_siz() {
    siz = cnt;
    if (ch[0] != nullptr) siz += ch[0]->siz;
    if (ch[1] != nullptr) siz += ch[1]->siz;
    return siz;
  }

  void pushdown() {
    swap(ch[0], ch[1]);
    if (ch[0] != nullptr) ch[0]->to_rev ^= 1;
    // 如果原来子节点也要翻转，那两次翻转就抵消了，如果子节点不翻转，那这个
    //  tag 就需要继续被 push 到子节点上
    if (ch[1] != nullptr) ch[1]->to_rev ^= 1;
    to_rev = false;
  }

  void check_tag() {
    if (to_rev) pushdown();
  }
};

struct Seg_treap {
  Node* root;
#define siz(_) (_ == nullptr ? 0 : _->siz)

  pair<Node*, Node*> split(Node* cur, int sz) {
    // 按照树的大小划分
```

```
    if (cur == nullptr) return {nullptr, nullptr};
    cur->check_tag();
    if (sz <= siz(cur->ch[0])) {
      // 左边的子树就够了
      auto temp = split(cur->ch[0], sz);
      // 左边的子树不一定全部需要，temp.second 是不需要的
      cur->ch[0] = temp.second;
      cur->upd_siz();
      return {temp.first, cur};
    } else {
      // 左边的加上右边的一部分（当然也包括这个节点本身）
      auto temp = split(cur->ch[1], sz - siz(cur->ch[0]) - 1);
      cur->ch[1] = temp.first;
      cur->upd_siz();
      return {cur, temp.second};
    }
  }

  Node* merge(Node* sm, Node* bg) {
    // small, big
    if (sm == nullptr && bg == nullptr) return nullptr;
    if (sm != nullptr && bg == nullptr) return sm;
    if (sm == nullptr && bg != nullptr) return bg;
    sm->check_tag(), bg->check_tag();
    if (sm->prio < bg->prio) {
      sm->ch[1] = merge(sm->ch[1], bg);
      sm->upd_siz();
      return sm;
    } else {
      bg->ch[0] = merge(sm, bg->ch[0]);
      bg->upd_siz();
      return bg;
    }
  }

  void insert(int val) {
    auto temp = split(root, val);
    auto l_tr = split(temp.first, val - 1);
    Node* new_node;
    if (l_tr.second == nullptr) new_node = new Node(val);
    Node* l_tr_combined =
        merge(l_tr.first, l_tr.second == nullptr ? new_node : l_tr.second);
    root = merge(l_tr_combined, temp.second);
  }

  void seg_rev(int l, int r) {
    // 这里的 less 和 more 是相对于 l 的
    auto less = split(root, l - 1);
    // 所有小于等于 l - 1 的会在 less 的左边
    auto more = split(less.second, r - l + 1);
    // 拿出从 l 开始的前 r - l + 1 个
    more.first->to_rev = true;
    root = merge(less.first, merge(more.first, more.second));
  }
```

```
  void print(Node* cur) {
    if (cur == nullptr) return;
    cur->check_tag();
    print(cur->ch[0]);
    cout << cur->val << " ";
    print(cur->ch[1]);
  }
};

Seg_treap tr;

int main() {
  srand(time(0));
  int n, m;
  cin >> n >> m;
  for (int i = 1; i <= n; i++) tr.insert(i);
  while (m--) {
    int l, r;
    cin >> l >> r;
    tr.seg_rev(l, r);
  }
  tr.print(tr.root);
}
```

例题

普通平衡树[7]

文艺平衡树（Splay）[6]

「ZJOI2006」书架[8]

「NOI2005」维护数列[9]

CF 702F T-Shirts[10]

参考资料与注释

[1] https://ttzytt.com/2022/06/treap_note/

[2] 本图的设计参考了 维基百科 treap 词条的配图

[3] https://charleswu.site/archives/1051

[4] https://www.cnblogs.com/Equinox-Flower/p/10785292.html [4-1] [4-2] [4-3]

[5] https://www.luogu.com.cn/blog/85514/fhq-treap-xue-xi-bi-ji

[6] 模板题 [6-1] [6-2]

[7] 普通平衡树

[8] 「ZJOI2006」书架

[9] 「NOI2005」维护数列

[10] CF 702F T-Shirts

10.17.3 Splay 树

本页面将简要介绍如何用 Splay 维护二叉查找树。

定义

Splay 树，或**伸展树**，是一种平衡二叉查找树，它通过 **Splay/伸展操作**不断将某个节点旋转到根节点，使得整棵树仍然满足二叉查找树的性质，能够在均摊 $O(\log N)$ 时间内完成插入，查找和删除操作，并且保持平衡而不至于退化为链。

Splay 树由 Daniel Sleator 和 Robert Tarjan 于 1985 年发明。

结构

二叉查找树的性质

Splay 树是一棵二叉搜索树，查找某个值时满足性质：左子树任意节点的值 $<$ 根节点的值 $<$ 右子树任意节点的值。

节点维护信息

rt	tot	fa[i]	ch[i][0/1]	val[i]	cnt[i]	sz[i]
根节点编号	节点个数	父亲	左右儿子编号	节点权值	权值出现次数	子树大小

操作

基本操作

- `maintain(x)`：在改变节点位置后，将节点 x 的 size 更新。
- `get(x)`：判断节点 x 是父亲节点的左儿子还是右儿子。
- `clear(x)`：销毁节点 x。

" 实现"

```cpp
void maintain(int x) { sz[x] = sz[ch[x][0]] + sz[ch[x][1]] + cnt[x]; }

bool get(int x) { return x == ch[fa[x]][1]; }

void clear(int x) { ch[x][0] = ch[x][1] = fa[x] = val[x] = sz[x] = cnt[x] = 0; }
```

旋转操作

为了使 Splay 保持平衡而进行旋转操作，旋转的本质是将某个节点上移一个位置。

旋转需要保证：

- 整棵 Splay 的中序遍历不变（不能破坏二叉查找树的性质）。
- 受影响的节点维护的信息依然正确有效。

- `root` 必须指向旋转后的根节点。

在 Splay 中旋转分为两种：左旋和右旋。

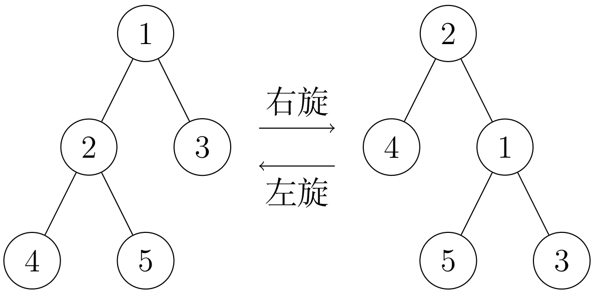


图 10.59

过程

具体分析旋转步骤（假设需要旋转的节点为 x，其父亲为 y，以右旋为例）

1. 将 y 的左儿子指向 x 的右儿子，且 x 的右儿子（如果 x 有右儿子的话）的父亲指向 y；`ch[y][0]=ch[x][1]; fa[ch[x][1]]=y;`
2. 将 x 的右儿子指向 y，且 y 的父亲指向 x；`ch[x][chk^1]=y; fa[y]=x;`
3. 如果原来的 y 还有父亲 z，那么把 z 的某个儿子（原来 y 所在的儿子位置）指向 x，且 x 的父亲指向 z。`fa[x]=z; if(z) ch[z][y==ch[z][1]]=x;`

实现

```
void rotate(int x) {
  int y = fa[x], z = fa[y], chk = get(x);
  ch[y][chk] = ch[x][chk ^ 1];
  if (ch[x][chk ^ 1]) fa[ch[x][chk ^ 1]] = y;
  ch[x][chk ^ 1] = y;
  fa[y] = x;
  fa[x] = z;
  if (z) ch[z][y == ch[z][1]] = x;
  maintain(y);
  maintain(x);
}
```

Splay 操作

Splay 操作规定：每访问一个节点 x 后都要强制将其旋转到根节点。

Splay 操作即对 x 做一系列的 **splay 步骤**。每次对 x 做一次 splay 步骤，x 到根节点的距离都会更近。定义 p 为 x 的父节点。Splay 步骤有三种，具体分为六种情况：

1. **zig**: 在 p 是根节点时操作。Splay 树会根据 x 和 p 间的边旋转。*zig* 存在是用于处理奇偶校验问题，仅当 x 在 splay 操作开始时具有奇数深度时作为 splay 操作的最后一步执行。

   即直接将 x 左旋或右旋（图 1, 2）
2. **zig-zig**: 在 p 不是根节点且 x 和 p 都是右侧子节点或都是左侧子节点时操作。下方例图显示了 x 和 p 都是左侧子节点时的情况。Splay 树首先按照连接 p 与其父节点 g 边旋转，然后按照连接 x 和 p 的边旋转。

   即首先将 g 左旋或右旋，然后将 x 右旋或左旋（图 3, 4）。
3. **zig-zag**: 在 p 不是根节点且 x 和 p 一个是右侧子节点一个是左侧子节点时操作。Splay 树首先按 p 和 x 之间的边旋转，然后按 x 和 g 新生成的结果边旋转。

   即将 x 先左旋再右旋、或先右旋再左旋（图 5, 6）。

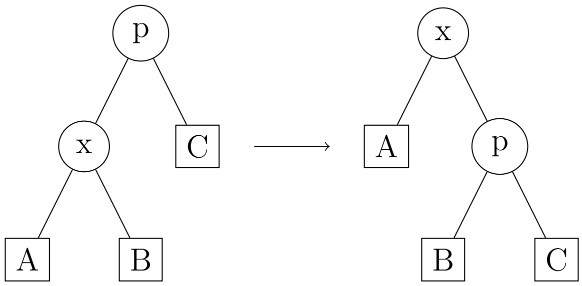
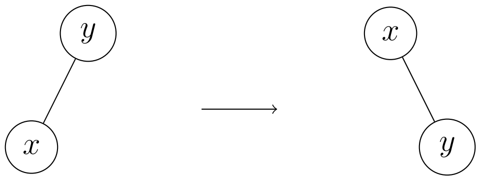
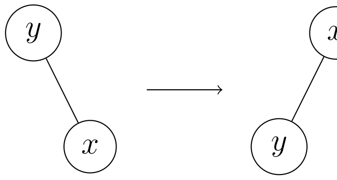


图 10.60 splay-zig



1

图 10.61 图 1



2

图 10.62 图 2

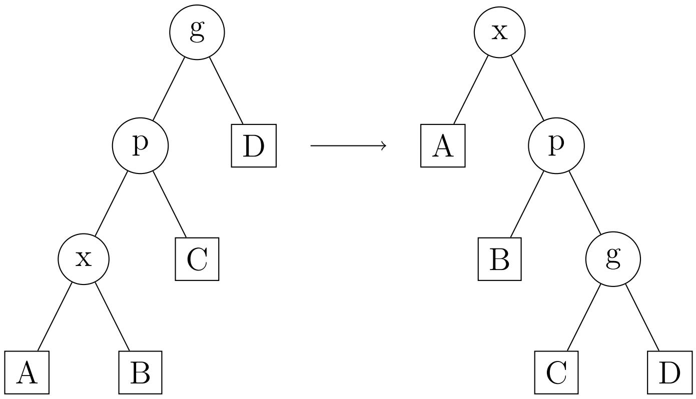
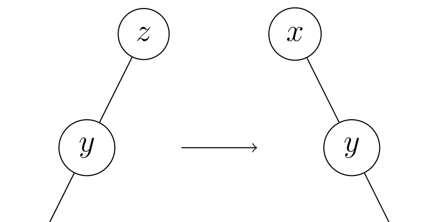
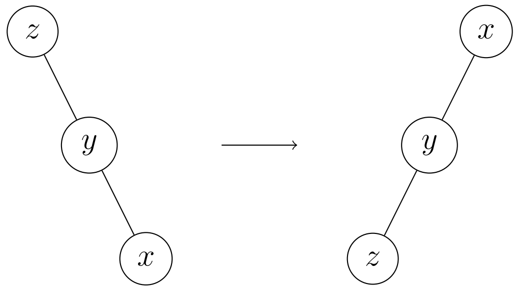


图 10.63 splay-zig-zig



3

图 10.64 图 3



4

图 10.65 图 4

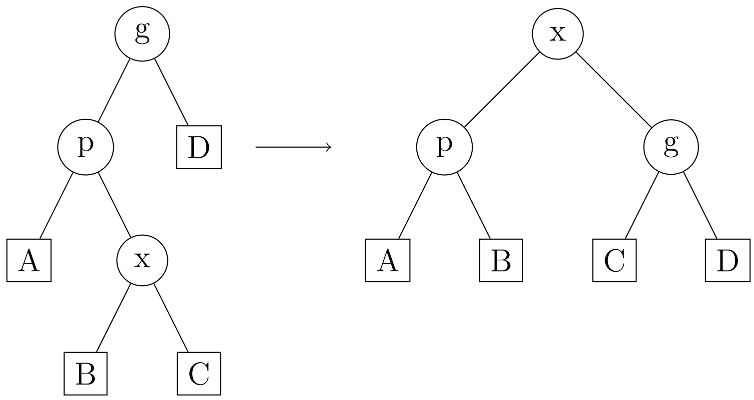
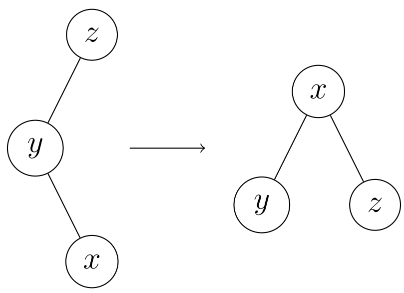
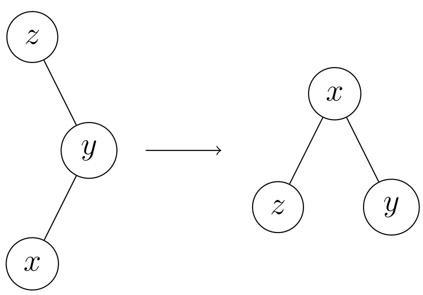


图 10.66 splay-zig-zag



5

图 10.67 图 5



6

图 10.68 图 6

> tip
>
> 请读者尝试自行模拟 6 种旋转情况，以理解 Splay 的基本思想。

实现

```
void splay(int x) {
  for (int f = fa[x]; f = fa[x], f; rotate(x))
    if (fa[f]) rotate(get(x) == get(f) ? f : x);
  rt = x;
}
```

Splay 操作的时间复杂度

因为 zig 和 zag 是**对称**操作，我们只需要对 zig，zig−zig，zig−zag 操作分析复杂度。采用 势能分析，定义一个 n 个节点的 splay 树进行了 m 次 splay 步骤。可记 $w(x)=[\log(\text{size}(x))]$，定义势能函数为 $\varphi=\sum w(x)$,$\varphi(0)\le n\log n$，在第 i 次操作后势能为 $\varphi(i)$，则我们只需要求出初始势能和每次的势能变化量的和即可。

1. **zig**: 势能的变化量为
$$\begin{aligned}1+w'(x)+w'(fa)-w(x)-w(fa) &\le 1+w'(fa)-w(x)\\ &\le 1+w'(x)-w(x)\end{aligned}$$
2. **zig-zig**: 势能变化量为
$$\begin{aligned}1+w'(x)+w'(fa)+w'(g)-w(x)-w(fa)-w(g) &\le 1+w'(fa)+w'(g)-w(x)-w(fa)\\ &\le 1+w'(x)+w'(g)-2w(x)\\ &\le 3(w'(x)-w(x))\end{aligned}$$
3. **zig-zag**: 势能变化量为
$$\begin{aligned}1+w'(x)+w'(fa)+w'(g)-w(x)-w(fa)-w(g) &\le 1+w'(fa)+w'(g)-w(x)-w(fa)\\ &\le 1+w'(g)+w'(fa)-2w(x)\\ &\le 2w'(x)-w'(g)-w'(fa)+w'(fa)+w'(g)-w(x)-w(fa)\\ &\le 2(w'(x)-w(x))\end{aligned}$$

由此可见，三种 splay 步骤的势能全部可以缩放为 $\le 3(w'(x)-w(x))$. 令 $w^{(n)}(x)=w'^{(n-1)}(x)$,$w^{(0)}(x)=w(x)$, 假设 splay 操作一次依次访问了 $x_1,x_2,\cdots,x_n$，最终 x_1 成为根节点，我们可以得到：

$$\begin{aligned}3\left(\sum_{i=0}^{n-2}\left(w^{(i+1)}(x_1)-w^{(i)}(x_1)\right)+w(n)-w^{(n-1)}(x_1)\right)+1 &= 3(w(n)-w(x_1))+1\\ &\le \log n\end{aligned}$$

继而可得：

$$\sum_{i=1}^{m}(\varphi(m-i+1)-\varphi(m-i))+\varphi(0)=n\log n+m\log n$$

因此，对于 n 个节点的 splay 树，做一次 splay 操作的均摊复杂度为 $O(\log n)$。因此基于 splay 的插入，查询，删除等操作的时间复杂度也为均摊 $O(\log n)$。

插入操作

过程

插入操作是一个比较复杂的过程，具体步骤如下（假设插入的值为 k）：

- 如果树空了，则直接插入根并退出。
- 如果当前节点的权值等于 k 则增加当前节点的大小并更新节点和父亲的信息，将当前节点进行 Splay 操作。
- 否则按照二叉查找树的性质向下找，找到空节点就插入即可（请不要忘记 Splay 操作）。

实现

```
void ins(int k) {
  if (!rt) {
    val[++tot] = k;
    cnt[tot]++;
    rt = tot;
    maintain(rt);
    return;
  }
  int cur = rt, f = 0;
  while (1) {
    if (val[cur] == k) {
      cnt[cur]++;
      maintain(cur);
      maintain(f);
      splay(cur);
      break;
    }
    f = cur;
    cur = ch[cur][val[cur] < k];
    if (!cur) {
      val[++tot] = k;
      cnt[tot]++;
      fa[tot] = f;
      ch[f][val[f] < k] = tot;
      maintain(tot);
      maintain(f);
      splay(tot);
      break;
    }
  }
}
```

查询 x 的排名

过程

根据二叉查找树的定义和性质，显然可以按照以下步骤查询 x 的排名：

- 如果 x 比当前节点的权值小，向其左子树查找。
- 如果 x 比当前节点的权值大，将答案加上左子树（$size$）和当前节点（cnt）的大小，向其右子树查找。
- 如果 x 与当前节点的权值相同，将答案加 1 并返回。

注意最后需要进行 Splay 操作。

实现

```
int rk(int k) {
  int res = 0, cur = rt;
  while (1) {
    if (k < val[cur]) {
      cur = ch[cur][0];
    } else {
      res += sz[ch[cur][0]];
      if (k == val[cur]) {
```

```
        splay(cur);
        return res + 1;
      }
      res += cnt[cur];
      cur = ch[cur][1];
    }
  }
}
```

查询排名 x 的数

过程

设 k 为剩余排名，具体步骤如下：

- 如果左子树非空且剩余排名 k 不大于左子树的大小 $size$，那么向左子树查找。
- 否则将 k 减去左子树的和根的大小。如果此时 k 的值小于等于 0，则返回根节点的权值，否则继续向右子树查找。

实现

```
int kth(int k) {
  int cur = rt;
  while (1) {
    if (ch[cur][0] && k <= sz[ch[cur][0]]) {
      cur = ch[cur][0];
    } else {
      k -= cnt[cur] + sz[ch[cur][0]];
      if (k <= 0) {
        splay(cur);
        return val[cur];
      }
      cur = ch[cur][1];
    }
  }
}
```

查询前驱

过程

前驱定义为小于 x 的最大的数，那么查询前驱可以转化为：将 x 插入（此时 x 已经在根的位置了），前驱即为 x 的左子树中最右边的节点，最后将 x 删除即可。

实现

```
int pre() {
  int cur = ch[rt][0];
  if (!cur) return cur;
  while (ch[cur][1]) cur = ch[cur][1];
  splay(cur);
  return cur;
}
```

查询后继

过程

后继定义为大于 x 的最小的数，查询方法和前驱类似：x 的右子树中最左边的节点。

实现

```
int nxt() {
  int cur = ch[rt][1];
  if (!cur) return cur;
  while (ch[cur][0]) cur = ch[cur][0];
  splay(cur);
  return cur;
}
```

合并两棵树

过程

合并两棵 Splay 树，设两棵树的根节点分别为 x 和 y，那么我们要求 x 树中的最大值小于 y 树中的最小值。合并操作如下：

- 如果 x 和 y 其中之一或两者都为空树，直接返回不为空的那一棵树的根节点或空树。
- 否则将 x 树中的最大值 Splay 到根，然后把它的右子树设置为 y 并更新节点的信息，然后返回这个节点。

删除操作

过程

删除操作也是一个比较复杂的操作，具体步骤如下：

首先将 x 旋转到根的位置。

- 如果 $cnt[x] > 1$（有不止一个 x），那么将 $cnt[x]$ 减 1 并退出。
- 否则，合并它的左右两棵子树即可。

实现

```
void del(int k) {
  rk(k);
  if (cnt[rt] > 1) {
    cnt[rt]--;
    maintain(rt);
    return;
  }
  if (!ch[rt][0] && !ch[rt][1]) {
    clear(rt);
    rt = 0;
    return;
  }
  if (!ch[rt][0]) {
    int cur = rt;
    rt = ch[rt][1];
    fa[rt] = 0;
    clear(cur);
    return;
  }
  if (!ch[rt][1]) {
    int cur = rt;
    rt = ch[rt][0];
    fa[rt] = 0;
    clear(cur);
    return;
```

```
  }
  int cur = rt, x = pre();
  fa[ch[cur][1]] = x;
  ch[x][1] = ch[cur][1];
  clear(cur);
  maintain(rt);
}
```

实现

```
#include <cstdio>
const int N = 100005;
int rt, tot, fa[N], ch[N][2], val[N], cnt[N], sz[N];

struct Splay {
  void maintain(int x) { sz[x] = sz[ch[x][0]] + sz[ch[x][1]] + cnt[x]; }

  bool get(int x) { return x == ch[fa[x]][1]; }

  void clear(int x) {
    ch[x][0] = ch[x][1] = fa[x] = val[x] = sz[x] = cnt[x] = 0;
  }

  void rotate(int x) {
    int y = fa[x], z = fa[y], chk = get(x);
    ch[y][chk] = ch[x][chk ^ 1];
    if (ch[x][chk ^ 1]) fa[ch[x][chk ^ 1]] = y;
    ch[x][chk ^ 1] = y;
    fa[y] = x;
    fa[x] = z;
    if (z) ch[z][y == ch[z][1]] = x;
    maintain(y);
    maintain(x);
  }

  void splay(int x) {
    for (int f = fa[x]; f = fa[x], f; rotate(x))
      if (fa[f]) rotate(get(x) == get(f) ? f : x);
    rt = x;
  }

  void ins(int k) {
    if (!rt) {
      val[++tot] = k;
      cnt[tot]++;
      rt = tot;
      maintain(rt);
      return;
    }
    int cur = rt, f = 0;
    while (1) {
      if (val[cur] == k) {
        cnt[cur]++;
        maintain(cur);
```

```
        maintain(f);
        splay(cur);
        break;
      }
      f = cur;
      cur = ch[cur][val[cur] < k];
      if (!cur) {
        val[++tot] = k;
        cnt[tot]++;
        fa[tot] = f;
        ch[f][val[f] < k] = tot;
        maintain(tot);
        maintain(f);
        splay(tot);
        break;
      }
    }
  }

  int rk(int k) {
    int res = 0, cur = rt;
    while (1) {
      if (k < val[cur]) {
        cur = ch[cur][0];
      } else {
        res += sz[ch[cur][0]];
        if (k == val[cur]) {
          splay(cur);
          return res + 1;
        }
        res += cnt[cur];
        cur = ch[cur][1];
      }
    }
  }

  int kth(int k) {
    int cur = rt;
    while (1) {
      if (ch[cur][0] && k <= sz[ch[cur][0]]) {
        cur = ch[cur][0];
      } else {
        k -= cnt[cur] + sz[ch[cur][0]];
        if (k <= 0) {
          splay(cur);
          return val[cur];
        }
        cur = ch[cur][1];
      }
    }
  }

  int pre() {
    int cur = ch[rt][0];
```

```
    if (!cur) return cur;
    while (ch[cur][1]) cur = ch[cur][1];
    splay(cur);
    return cur;
  }

  int nxt() {
    int cur = ch[rt][1];
    if (!cur) return cur;
    while (ch[cur][0]) cur = ch[cur][0];
    splay(cur);
    return cur;
  }

  void del(int k) {
    rk(k);
    if (cnt[rt] > 1) {
      cnt[rt]--;
      maintain(rt);
      return;
    }
    if (!ch[rt][0] && !ch[rt][1]) {
      clear(rt);
      rt = 0;
      return;
    }
    if (!ch[rt][0]) {
      int cur = rt;
      rt = ch[rt][1];
      fa[rt] = 0;
      clear(cur);
      return;
    }
    if (!ch[rt][1]) {
      int cur = rt;
      rt = ch[rt][0];
      fa[rt] = 0;
      clear(cur);
      return;
    }
    int cur = rt;
    int x = pre();
    fa[ch[cur][1]] = x;
    ch[x][1] = ch[cur][1];
    clear(cur);
    maintain(rt);
  }
} tree;

int main() {
  int n, opt, x;
  for (scanf("%d", &n); n; --n) {
    scanf("%d%d", &opt, &x);
    if (opt == 1)
```

```
      tree.ins(x);
    else if (opt == 2)
      tree.del(x);
    else if (opt == 3)
      printf("%d\n", tree.rk(x));
    else if (opt == 4)
      printf("%d\n", tree.kth(x));
    else if (opt == 5)
      tree.ins(x), printf("%d\n", val[tree.pre()]), tree.del(x);
    else
      tree.ins(x), printf("%d\n", val[tree.nxt()]), tree.del(x);
  }
  return 0;
}
```

序列操作

Splay 也可以运用在序列上，用于维护区间信息。与线段树对比，Splay 常数较大，但是支持更复杂的序列操作，如区间翻转等。

将序列建成的 Splay 有如下性质：

- Splay 的中序遍历相当于原序列从左到右的遍历。
- Splay 上的一个节点代表原序列的一个元素；Splay 上的一颗子树，代表原序列的一段区间。

因为有 splay 操作，可以快速提取出代表某个区间的 Splay 子树。

在操作之前，你需要先把这颗 Splay 建出来。根据 Splay 的特性，直接建出一颗只有右儿子的链即可，时间复杂度仍然是正确的。

一些进阶操作

Splay 的一颗子树代表原序列的一段区间。现在想找到序列区间 $[L,R]$ 代表的子树，只需要将代表 a_{L-1} 的节点 Splay 到根，再将代表 a_{R+1} 的节点 splay 到根的右儿子即可。根据「Splay 的中序遍历相当于原序列从左到右的遍历」，对应 a_{R+1} 的节点的左子树中序遍历为序列 $a[L,R]$，故其为区间 $[L,R]$ 代表的子树。

一般会建立左右两个哨兵节点 0 和 $n+1$，放在数列的最开头和最结尾，防止 $L-1$ 或 $R+1$ 超出数列范围。

所以要将 splay 函数进行一些修改，能够实现将节点旋转到目标点的儿子。如果目标点 `goal` 为 0 说明旋转到根节点。

实现

```
void splay(int x, int goal = 0) {
  if (goal == 0) rt = x;
  while (fa[x] != goal) {
    int f = fa[x], g = fa[fa[x]];
    if (g != goal) {
      if (get(f) == get(x))
        rotate(x);
      else
        rotate(f);
    }
    rotate(x);
  }
}
```

区间翻转

Splay 常见的应用之一，模板题目是 文艺平衡树[1]。

过程

先将询问区间的子树提取出来。因为是区间翻转，我们需要将这颗子树的中序遍历顺序翻转。

一个暴力做法是每次将根节点的左右儿子交换，然后递归左右子树做同样的操作，这样复杂度为 $O(n)$，不可承受。可以考虑使用懒标记，先给根打上「翻转标记」并交换其左右儿子。当递归到一个带懒标记的点时，将懒标记下传即可。

实现

```
void tagrev(int x) {
  swap(ch[x][0], ch[x][1]);
  lazy[x] ^= 1;
}

void pushdown(int x) {
  if (lazy[x]) {
    tagrev(ch[x][0]);
    tagrev(ch[x][1]);
    lazy[x] = 0;
  }
}

void reverse(int l, int r) {
  int L = kth(l - 1), R = kth(r + 1);
  splay(L), splay(R, L);
  int tmp = ch[ch[L][1]][0];
  tagrev(tmp);
}
```

实现

注意 kth 中要下传翻转标记。

```
#include <algorithm>
#include <cstdio>
const int N = 100005;

int n, m, l, r, a[N];

int rt, tot, fa[N], ch[N][2], val[N], sz[N], lazy[N];

struct Splay {
  void maintain(int x) { sz[x] = sz[ch[x][0]] + sz[ch[x][1]] + 1; }

  bool get(int x) { return x == ch[fa[x]][1]; }

  void clear(int x) {
    ch[x][0] = ch[x][1] = fa[x] = val[x] = sz[x] = lazy[x] = 0;
  }

  void rotate(int x) {
```

```
    int y = fa[x], z = fa[y], chk = get(x);
    ch[y][chk] = ch[x][chk ^ 1];
    if (ch[x][chk ^ 1]) fa[ch[x][chk ^ 1]] = y;
    ch[x][chk ^ 1] = y;
    fa[y] = x;
    fa[x] = z;
    if (z) ch[z][y == ch[z][1]] = x;
    maintain(y);
    maintain(x);
  }

  void splay(int x, int goal = 0) {
    if (goal == 0) rt = x;
    while (fa[x] != goal) {
      int f = fa[x], g = fa[fa[x]];
      if (g != goal) {
        if (get(f) == get(x))
          rotate(x);
        else
          rotate(f);
      }
      rotate(x);
    }
  }

  void tagrev(int x) {
    std::swap(ch[x][0], ch[x][1]);
    lazy[x] ^= 1;
  }

  void pushdown(int x) {
    if (lazy[x]) {
      tagrev(ch[x][0]);
      tagrev(ch[x][1]);
      lazy[x] = 0;
    }
  }

  int build(int l, int r, int f) {
    if (l > r) return 0;
    int mid = (l + r) / 2, cur = ++tot;
    val[cur] = a[mid], fa[cur] = f;
    ch[cur][0] = build(l, mid - 1, cur);
    ch[cur][1] = build(mid + 1, r, cur);
    maintain(cur);
    return cur;
  }

  int kth(int k) {
    int cur = rt;
    while (1) {
      pushdown(cur);
      if (ch[cur][0] && k <= sz[ch[cur][0]]) {
        cur = ch[cur][0];
```

```
      } else {
        k -= 1 + sz[ch[cur][0]];
        if (k <= 0) {
          splay(cur);
          return cur;
        }
        cur = ch[cur][1];
      }
    }
  }

  void reverse(int l, int r) {
    int L = kth(l), R = kth(r + 2);
    splay(L), splay(R, L);
    int tmp = ch[ch[L][1]][0];
    tagrev(tmp);
  }

  void print(int x) {
    pushdown(x);
    if (ch[x][0]) print(ch[x][0]);
    if (val[x] >= 1 && val[x] <= n) printf("%d ", val[x]);
    if (ch[x][1]) print(ch[x][1]);
  }
} tree;

int main() {
  scanf("%d%d", &n, &m);
  for (int i = 0; i <= n + 1; i++) a[i] = i;
  rt = tree.build(0, n + 1, 0);
  while (m--) {
    scanf("%d%d", &l, &r);
    tree.reverse(l, r);
  }
  tree.print(rt);
  return 0;
}
```

例题

以下题目都是裸的 Splay 维护二叉查找树。

- 【模板】普通平衡树[2]
- 【模板】文艺平衡树[1]
- 「HNOI2002」营业额统计[3]
- 「HNOI2004」宠物收养所[4]

习题

- 「Cerc2007」robotic sort 机械排序[5]
- 二逼平衡树（树套树）[6]
- bzoj 2827 千山鸟飞绝[7]
- 「Lydsy1706 月赛」K 小值查询[8]

- POJ3580 SuperMemo[9]

参考资料与注释

本文部分内容引用于 algocode 算法博客[10]，特别鸣谢！

参考资料与注释

[1] 文艺平衡树 [1-1] [1-2]

[2] 【模板】普通平衡树

[3] 「HNOI2002」营业额统计

[4] 「HNOI2004」宠物收养所

[5] 「Cerc2007」robotic sort 机械排序

[6] 二逼平衡树（树套树）

[7] bzoj 2827 千山鸟飞绝

[8] 「Lydsy1706 月赛」K 小值查询

[9] POJ3580 SuperMemo

[10] algocode 算法博客

10.17.4 WBLT

Authors: hsfzLZH1, cesonic, AtomAlpaca

前言

Weight Balanced Leafy Tree，下称 **WBLT**，是一种平衡树，比起其它平衡树主要有实现简单、常数小的优点。

Weight Balanced Leafy Tree 顾名思义是 Weight Balanced Tree 和 Leafy Tree 的结合。

Weight Balanced Tree 的每个结点储存这个结点下子树的大小，并且通过保持左右子树的大小关系在一定范围来保证树高。

Leafy Tree 维护的原始信息仅存储在树的**叶子节点**上，而非叶子节点仅用于维护子节点信息和维持数据结构的形态。我们熟知的线段树就是一种 Leafy Tree。

平衡树基础操作

代码约定

下文中，我们用 `ls[x]` 表示节点 x 的左儿子，`rs[x]` 表示节点 x 的右儿子，`vl[x]` 表示节点 x 的权值，`sz[x]` 表示节点 x 及其子树中叶子节点的个数。

建树

正如前言中所说的，WBLT 的原始信息仅存储在叶子节点上。而我们规定每个非叶子节点一定有两个子节点，这个节点要维护其子节点信息的合并。同时，每个节点还要维护自身及其子树中叶子节点的数量，用于实现维护平衡。

和大多数的平衡树一样，每个非叶子节点的右儿子的权值大于等于左儿子的权值，且在 WBLT 中非叶子节点节点的权值等于右儿子的权值。不难看出每个节点的权值就是其子树中的最大权值。

这样听起来就很像一棵维护区间最大值的动态开点线段树了，且所有叶子从左到右是递增的。事实上的建树操作也与线段树十分相似，只需要向下递归，直至区间长度为 1 时把要维护的信息放叶子节点上，回溯的时候合并区间信息即可。

代码实现如下：

```cpp
/* 添加一个权值为 v 的节点，返回这个节点的编号 */
int add(int v) {
  ++cnt;
  ls[cnt] = rs[cnt] = 0;
  sz[cnt] = 1;
  vl[cnt] = v;
  return cnt;
}

/* 更新节点编号为 x 的节点的信息 */
void pushup(int x) {
  vl[x] = vl[rs[x]];
  sz[x] = sz[ls[x]] + sz[rs[x]];
}

/* 递归建树 */
int build(int l, int r) {
  if (l == r) {
    return add(a[l]);
  }
  int x = add(0);
  int k = l + ((r - l) >> 1);
  ls[x] = build(l, k);
  rs[x] = build(k + 1, r);
  pushup(x);
}
```

插入和删除

由于 WBLT 的信息都存储在叶子节点上，插入和删除一个元素其实就是增加或减少了一个叶子节点。

对于插入操作，我们类似从根节点开始向下递归，直到找到权值大于等于插入元素的权值最小的叶子节点，再新建两个节点，其中一个用来存储新插入的值，另一个作为两个叶子的新父亲替代这个最小叶子节点的位置，再将这两个叶子连接到这个父亲上。

例如我们向以下树中加入一个值为 4 的元素。

我们首先找到了叶子节点 5，随后新建了一个非叶子节点 D，并将 4 和 5 连接到了 D 上。

对于删除，我们考虑上面过程的逆过程。即找到与要删除的值权值相等的一个叶子节点，将它和它的父亲节点删除，并用其父亲的另一个儿子代替父亲的位置。

上面提到的建树也可以通过不断往树里插入节点实现，不过如果这样做必须要加入一个权值为 ∞ 的节点作为根，否则会导致插入第一个元素的时候找不到大于自己的叶子节点。

代码实现：

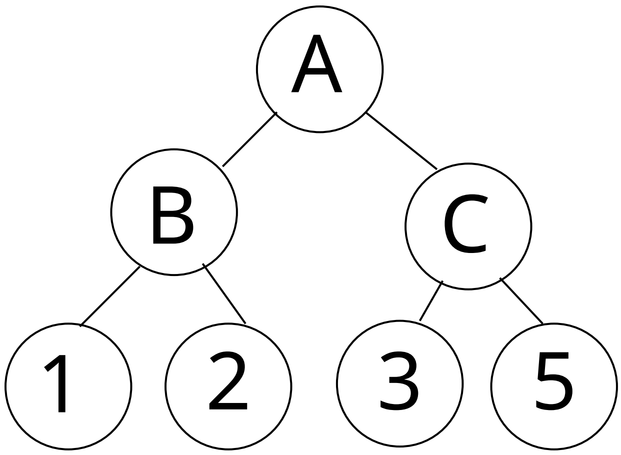


图 10.69 wblt-1

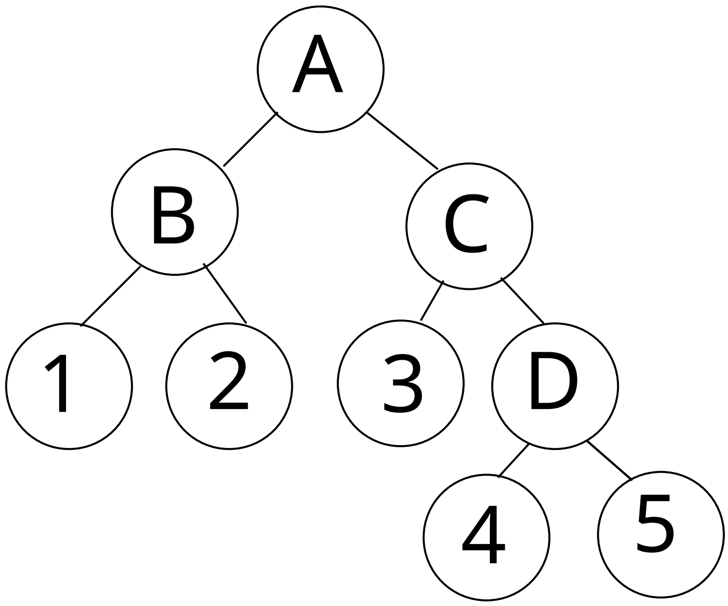


图 10.70 wblt-2

```cpp
/* 将某一节点的全部信息复制到另一节点上 */
void copynode(int x, int y) {
  ls[x] = ls[y];
  rs[x] = rs[y];
  sz[x] = sz[y];
  vl[x] = vl[y];
}

/* 判断某一节点是否为叶子节点 */
bool leaf(int x) { return !ls[x] || !rs[x]; }

void insert(int v) {
  if (leaf(x)) {
    ls[x] = add(std::min(v, vl[x]));
    rs[x] = add(std::max(v, vl[x]));
    pushup(x);
    maintain(x);
    return;
  }
  if (vl[ls[x]] >= v) {
    insert(ls[x], v);
  } else {
    insert(rs[x], v);
  }
  pushup(x);
  maintain(x);
}

void delete(int x, int v, int fa) {
  if (leaf(x)) {
    if (ls[fa] == x) {
      copynode(fa, rs[fa]);
    } else {
      copynode(fa, ls[fa]);
    }
    pushup(fa);
    return;
  }
  if (vl[ls[x]] >= v) {
    delete (ls[x], v, x);
  } else {
    delete (rs[x], v, x);
  }
  pushup(x);
  maintain(x);
}
```

维护平衡

类似替罪羊树地，我们引入重构参数 $\alpha \in (0, \dfrac{1}{2}]$，我们设一个节点的平衡度 ρ 为当前节点左子树大小和节点大小的比值。当一个节点满足 $\rho \in [\alpha, 1-\alpha]$ 时，我们称其为 α- 平衡的。如果一棵 WBLT 的每一个节点都是 α- 平衡的，那么这棵树的树高一定能保证是 $O(\log n)$ 量级的。证明是显然的，我们从一个叶子节点往父亲方向走，每次走到的节点维护的范围至少扩大到原来的 $\dfrac{1}{1-\alpha}$ 倍，那么树高就是 $O(\log_{\frac{1}{1-\alpha}} n) = O(\log n)$ 量级的。

当某个节点不满足 α- 平衡时，说明这个节点是失衡的，我们需要重新维护平衡。但是和替罪羊树不同的是，

WBLT 使用旋转操作维护平衡。旋转的大致过程为：将过重的儿子的两个儿子拆下来，一个和过轻的儿子合并，另一个成为一个新的儿子。

我们来举个例子：

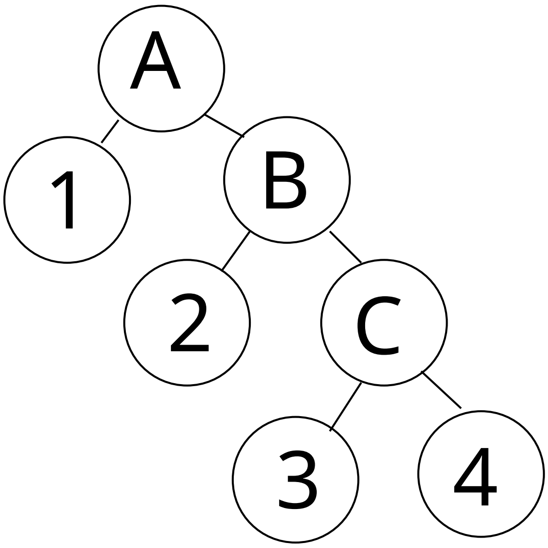


图 10.71 wblt-3

这是一棵十分不平衡的 WBLT，节点 A 的右儿子显著地重于左儿子。我们先把右儿子及其两个儿子和左儿子都拆下来：

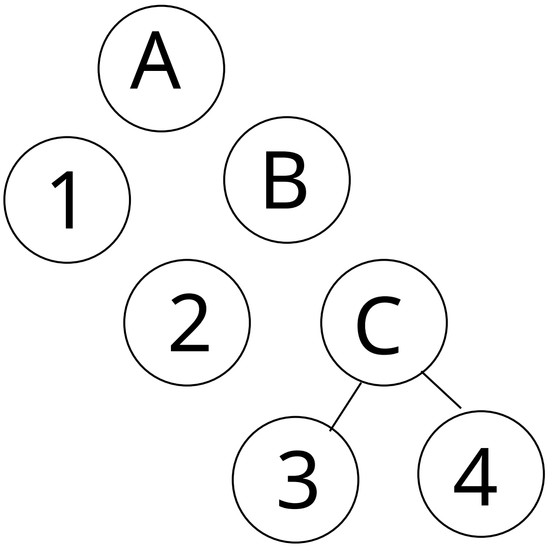


图 10.72 wblt-4

然后，我们将 1 和 2 两个节点合并作为 A 节点的左儿子，将 C 作为 A 的右儿子。由于 B 节点原本并不是叶子节点，因此其并不存储原始信息，直接删除就好。

旋转之后我们的树就变得十分平衡了。

但是上面的例子中，假设 A 节点的左子树过于大，我们把它合并到 A 的左子树上之后 A 的左子树又会很大，这时 A 依然可能不平衡。

不失一般性，我们接下来仅讨论一个方向上的旋转，另一方向的旋转是对称的。我们不妨设 `A` 的平衡度为 ρ_1，`B` 的平衡度为 ρ_2。那么我们可以得到旋转后 `A` 的平衡度 $\gamma_1 = \rho_1 + (1-\rho_1)\rho_2$，`B` 的平衡度 $\gamma_2 = \dfrac{\rho_1}{\rho_1 + (1-\rho_1)\rho_2}$，推导过程直接将各节点大小用 siz_A 表示后代入定义式化简即可，这里略去。

不难发现仅当 $\rho_2 \le \dfrac{1-2\alpha}{1-\alpha}$ 时 $\gamma_1, \gamma_2 \in [\alpha, 1-\alpha]$。

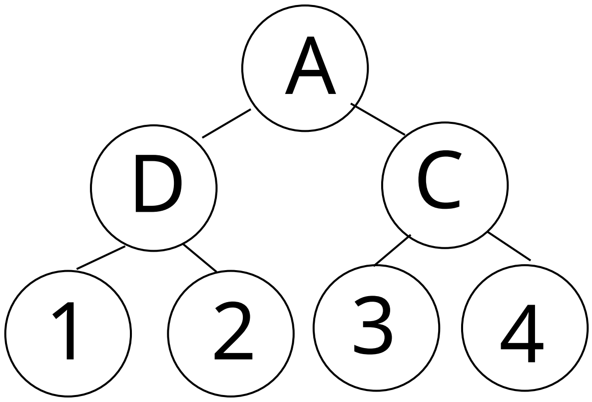


图 10.73 wblt-5

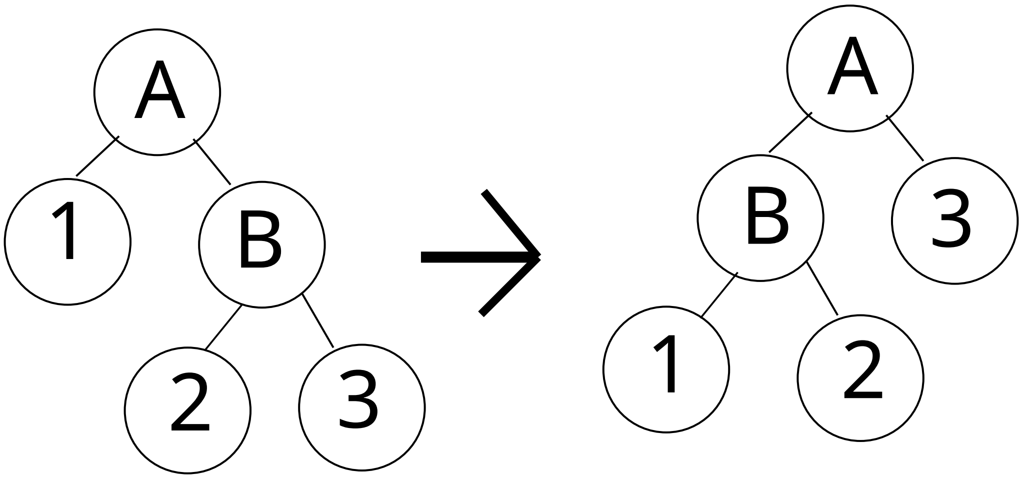


图 10.74 wblt-6

为了旋转后仍不平衡的情况出现，我们引入双旋操作。具体地，我们在较大子树上做一次相反方向的旋转操作，然后再维护当前节点的平衡。

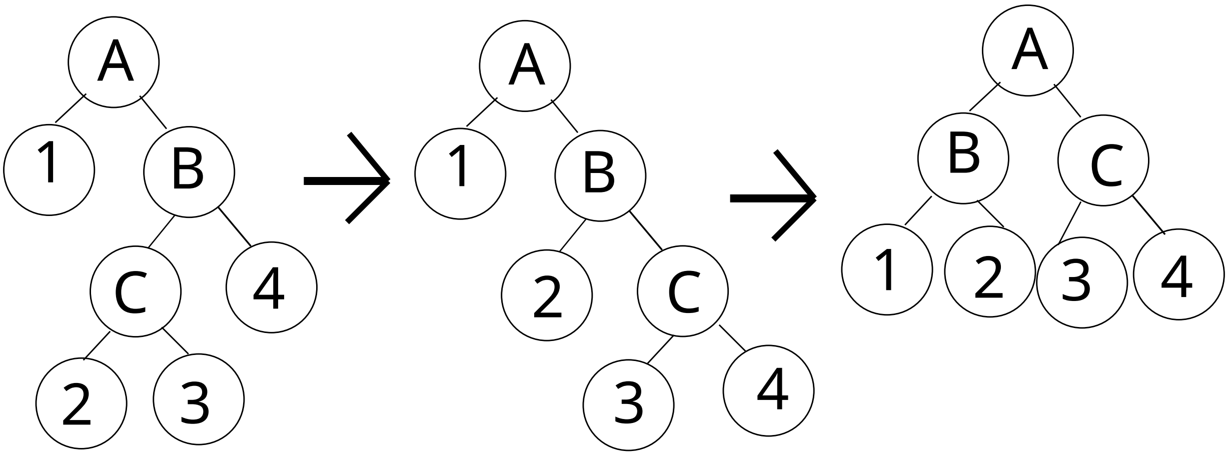


图 10.75 wblt-7

类似地定义 ρ_3, γ_3，则有 $\gamma_1 = \rho_1 + \rho_2\rho_3(1-\rho_1), \gamma_2 = \dfrac{\rho_1}{\rho 1 + (1-\rho_1)\rho 2\rho 3}, \gamma_3 = \dfrac{\rho_2(1-\rho_3)}{1-\rho_2\rho_3}$。可以证明当 $\alpha < 1 - \dfrac{\sqrt{2}}{2} \approx 0.292$ 时一定有 $\gamma_1, \gamma_2, \gamma_3 \in [\alpha, 1-\alpha]$。

实现上，我们在 $\rho_2 \leq \dfrac{1-2\alpha}{1-\alpha}$ 时进行单旋，否则进行双旋。

代码实现，这里取 $\alpha = 0.25$：

```
const double alpha = 0.25;

void rotate(int x, int flag) {
  if (!flag) {
    rs[x] = merge(rs[ls[x]], rs[x]);
    ls[x] = ls[ls[x]];
  } else {
    ls[x] = merge(ls[x], ls[rs[x]]);
    rs[x] = rs[rs[x]];
  }
}

void maintain(int x) {
  if (sz[ls[x]] > sz[rs[x]] * 3) {
    if (sz[rs[ls[x]]] > sz[ls[ls[x]]] * 2) {
      rotate(ls[x], 1);
    }
    rotate(x, 0);
  } else if (sz[rs[x]] > sz[ls[x]] * 3) {
    if (sz[ls[rs[x]]] > sz[rs[rs[x]]] * 2) {
      rotate(rs[x], 0);
    }
    rotate(x, 1);
  }
}
```

查询排名

我们发现 `WBLT` 的形态和线段树十分相似，因此查询排名可以使用类似线段树上二分的方式：如果左子树的最大值比大于等于待查值就往左儿子跳，否则就向右跳，同时答案加上左子树的 `size`。

```cpp
int rank(int x, int v) {
  if (leaf(x)) {
    return 1;
  }
  if (vl[ls[x]] >= v) {
    return rank(ls[x], v);
  } else {
    return rank(rs[x], v) + sz[ls[x]];
  }
}
```

查询第 k 大的数

依然是利用线段树上二分的思想，只不过这里比较的是节点的大小。

```cpp
int kth(int x, int v) {
  if (sz[x] == v) {
    return vl[x];
  }
  if (sz[ls[x]] >= v) {
    return kth(ls[x], v);
  } else {
    return kth(rs[x], v - sz[ls[x]]);
  }
}
```

总结

以上，我们利用 WBLT 完成了平衡树基本的几大操作。下面是用 WBLT 实现的 普通平衡树模板[1]。

" 完整代码"

```cpp
#include <bits/stdc++.h>

typedef long long ll;

const ll MAX = 2e6 + 5;
const ll INF = 0x7fffffff;

ll ans, lst, n, m, t, op, rt, cnt;
ll ls[MAX], rs[MAX], vl[MAX], sz[MAX];

void cp(ll x, ll y) {
  ls[x] = ls[y];
  rs[x] = rs[y];
  sz[x] = sz[y];
  vl[x] = vl[y];
}

ll add(ll v, ll s, ll l, ll r) {
  ++cnt;
  ls[cnt] = l;
  rs[cnt] = r;
  sz[cnt] = s;
```

```
  vl[cnt] = v;
  return cnt;
}

ll merge(ll x, ll y) { return add(vl[y], sz[x] + sz[y], x, y); }

void upd(ll x) {
  if (!ls[x]) {
    sz[x] = 1;
    return;
  }
  sz[x] = sz[ls[x]] + sz[rs[x]];
  vl[x] = vl[rs[x]];
}

void rot(int x, int flag) {
  if (!flag) {
    rs[x] = merge(rs[ls[x]], rs[x]);
    ls[x] = ls[ls[x]];
  } else {
    ls[x] = merge(ls[x], ls[rs[x]]);
    rs[x] = rs[rs[x]];
  }
}

void mat(int x) {
  if (sz[ls[x]] > sz[rs[x]] * 3) {
    if (sz[rs[ls[x]]] > sz[ls[ls[x]]] * 2) {
      rot(ls[x], 1);
    }
    rot(x, 0);
  } else if (sz[rs[x]] > sz[ls[x]] * 3) {
    if (sz[ls[rs[x]]] > sz[rs[rs[x]]] * 2) {
      rot(rs[x], 0);
    }
    rot(x, 1);
  }
}

void ins(ll x, ll v) {
  if (!ls[x]) {
    ls[x] = add(std::min(v, vl[x]), 1, 0, 0);
    rs[x] = add(std::max(v, vl[x]), 1, 0, 0);
    upd(x);
    mat(x);
    return;
  }
  if (vl[ls[x]] >= v) {
    ins(ls[x], v);
  } else {
    ins(rs[x], v);
  }
  upd(x);
  mat(x);
```

```
  return;
}

void del(ll x, ll v, ll fa) {
  if (!ls[x]) {
    if (vl[ls[fa]] == v) {
      cp(fa, rs[fa]);
    } else if (vl[rs[fa]] == v) {
      cp(fa, ls[fa]);
    }
    return;
  }
  if (vl[ls[x]] >= v) {
    del(ls[x], v, x);
  } else {
    del(rs[x], v, x);
  }
  upd(x);
  mat(x);
  return;
}

ll rnk(ll x, ll v) {
  if (sz[x] == 1) {
    return 1;
  }
  if (vl[ls[x]] >= v) {
    return rnk(ls[x], v);
  } else {
    return rnk(rs[x], v) + sz[ls[x]];
  }
}

ll kth(ll x, ll v) {
  if (sz[x] == v) {
    return vl[x];
  }
  if (sz[ls[x]] >= v) {
    return kth(ls[x], v);
  } else {
    return kth(rs[x], v - sz[ls[x]]);
  }
}

ll pre(ll x) { return kth(rt, rnk(rt, x) - 1); }

ll nxt(ll x) { return kth(rt, rnk(rt, x + 1)); }

int main() {
  scanf("%lld", &m);
  rt = add(INF, 1, 0, 0);
  while (m--) {
    scanf("%lld%lld", &op, &t);
    if (op == 1) {
```

```
      ins(rt, t);
    } else if (op == 2) {
      del(rt, t, -1);
    } else if (op == 3) {
      printf("%lld\n", rnk(rt, t));
    } else if (op == 4) {
      printf("%lld\n", kth(rt, t));
    } else if (op == 5) {
      printf("%lld\n", pre(t));
    } else {
      printf("%lld\n", nxt(t));
    }
  }
  return 0;
}
```

参考资料与注释

[1] 普通平衡树模板

10.17.5 Size Balanced Tree

Size Balanced Tree (SBT) 是由中国 OI 选手陈启峰在 2007 年提出的一种自平衡二叉搜索树 (Self-Balanced Binary Search Tree, SBBST), 通过检查子树的节点数量进行自身的平衡维护。相比于红黑树，AVL 等主流自平衡二叉搜索树而言，Size Balanced Tree 支持在 $O(\log n)$ 的时间复杂度内查询某个键值在树中的排名 (rank).

节点定义

相比与普通二叉搜索树，SBT 的每个节点 N 仅需要多维护一个整数字段 `size`, 用于储存以 N 为根的子树中节点的个数。节点类型 `Node` 的具体定义如下：

Identifier	Type	Description
`left`	`Node*`	左子节点引用
`right`	`Node*`	右子节点引用
`size`	`int`	以该节点为根的子树中节点的个数

性质

Size Balanced Tree 中任意节点 N 满足如下几条性质：

```
size(N.left) >= size(N.right.left)
size(N.left) >= size(N.right.right)
size(N.right) >= size(N.left.left)
size(N.right) >= size(N.left.right)
```

使用自然语言可描述为：任意节点的 `size` 不小于其兄弟节点（Sibling）的所有子节点（Nephew）的 `size`.

平衡维护

旋转

SBT 主要通过旋转操作改变自身高度从而进行平衡维护。其旋转操作与绝大部分自平衡二叉搜索树类似，唯一区别在于在完成旋转之后需要对旋转过程中左右子节点发生改变的节点更新 `size`. 示例代码如下：

```cpp
void updateSize() {
  USize leftSize = this->left != nullptr ? this->left->size : 0;
  USize rightSize = this->right != nullptr ? this->right->size : 0;
  this->size = leftSize + rightSize + 1;
}

static void rotateLeft(NodePtr& node) {
  assert(node != nullptr);
  // clang-format off
  //      |                       |
  //      N                       S
  //     / \     l-rotate(N)     / \
  //    L   S    ==========>    N   R
  //       / \                 / \
  //      M   R               L   M
  // clang-format on
  NodePtr successor = node->right;
  node->right = successor->left;
  successor->left = node;

  node->updateSize();
  successor->updateSize();

  node = successor;
}

static void rotateRight(NodePtr& node) {
  assert(node != nullptr);
  // clang-format off
  //        |                   |
  //        N                   S
  //       / \   r-rotate(N)   / \
  //      S   R  ==========>  L   N
  //     / \                     / \
  //    L   M                   M   R
  // clang-format on
  NodePtr successor = node->left;
  node->left = successor->right;
  successor->right = node;

  node->updateSize();
  successor->updateSize();

  node = successor;
}
```

维护

Case 1

`size(N.left) < size(N.right.left)`

```
if (size(node->right->left) > size(node->left)) {
  // clang-format off
  //     |                     |                     |
  //     N                     N                    [M]
  //    / \    r-rotate(R)    / \    l-rotate(N)    / \
  //  <L>  R   ==========>  <L> [M]  ==========>   N   R
  //      /                       \               /
  //    [M]                        R            <L>
  // clang-format on
  rotateRight(node->right);
  rotateLeft(node);
  fixBalance(node->left);
  fixBalance(node->right);
  fixBalance(node);
  return;
}
```

Case 2

`size(N.left) < size(N.right.right)`

```
if (size(node->right->right) > size(node->left)) {
  // clang-format off
  //     |                       |
  //     N                       R
  //    / \     l-rotate(N)     / \
  //  <L>  R    ==========>    N  [M]
  //        \                 /
  //        [M]             <L>
  // clang-format on
  rotateLeft(node);
  fixBalance(node->left);
  fixBalance(node);
  return;
}
```

Case 3

`size(N.right) < size(N.left.left)`

```
if (size(node->left->left) > size(node->right)) {
  // clang-format off
  //       |                       |
  //       N                       L
  //      / \     r-rotate(N)     / \
  //     L  <R>   ==========>   [M]  N
  //    /                             \
  //  [M]                             <R>
  // clang-format on
  rotateRight(node);
  fixBalance(node->right);
```

```
  fixBalance(node);
  return;
}
```

Case 4

```
size(N.right) < size(N.left.right)
```

```
if (size(node->left->right) > size(node->right)) {
  // clang-format off
  //      |                     |                       |
  //      N                     N                      [M]
  //     / \     l-rotate(L)   / \     r-rotate(N)    / \
  //    L   <R>  ==========> [M] <R>   ==========>   L   N
  //     \                   /                            \
  //     [M]                L                             <R>
  // clang-format on
  rotateLeft(node->left);
  rotateRight(node);
  fixBalance(node->left);
  fixBalance(node->right);
  fixBalance(node);
  return;
}
```

操作

插入

SBT 的插入操作需要在完成普通二叉搜索树的插入操作的基础上递归地进行节点 `size` 字段的更新及平衡维护。示例代码如下：

```
if (compare(key, node->key)) {
  /* key < node->key */
  if (node->left == nullptr) {
    node->left = Node::from(key, value);
    node->updateSize();
  } else {
    insert(node->left, key, value, replace);
    node->updateSize();
    fixBalance(node);
  }
} else {
  /* key > node->key */
  if (node->right == nullptr) {
    node->right = Node::from(key, value);
    node->updateSize();
  } else {
    insert(node->right, key, value, replace);
    node->updateSize();
    fixBalance(node);
  }
}
```

删除

根据 Size Balanced Tree 的提出者陈启峰在其论文中对于删除操作的描述：

> It can result in a destroyed SBT. But with the insertion above, a BST is still kept at the height of $O(\log n)$ where n is the total number of insertions, not the current size.

删除操作虽然有可能使得 SBT 的性质被打破，但并不会使树的高度增高，因此不会影响后续操作的效率。但在实际情况下，如果在一次批量插入操作后只进行大量的删除和查询操作，依然有可能由于树的失衡影响整体效率，因此本文在实现 SBT 的删除操作时依然选择加入平衡维护。参考代码如下：

```
bool remove(NodePtr& node, K key, NodeConsumer action) {
  assert(node != nullptr);

  if (key != node->key) {
    if (compare(key, node->key)) {
      /* key < node->key */
      NodePtr& left = node->left;
      if (left != nullptr && remove(left, key, action)) {
        node->updateSize();
        fixBalance(node);
        return true;
      } else {
        return false;
      }
    } else {
      /* key > node->key */
      NodePtr& right = node->right;
      if (right != nullptr && remove(right, key, action)) {
        node->updateSize();
        fixBalance(node);
        return true;
      } else {
        return false;
      }
    }
  }

  assert(key == node->key);
  action(node);

  if (node->isLeaf()) {
    // Case 1: no child
    node = nullptr;
  } else if (node->right == nullptr) {
    // Case 2: left child only
    // clang-format off
    //     P
    //     |  remove(N)  P
    //     N  ========>  |
    //    /              L
    //   L
    // clang-format on
    node = node->left;
  } else if (node->left == nullptr) {
```

```
    // Case 3: right child only
    // clang-format off
    //   P
    //   |    remove(N)  P
    //   N    ========>  |
    //    \              R
    //     R
    // clang-format on
    node = node->right;
  } else if (node->right->left == nullptr) {
    // Case 4: both left and right child, right child has no left child
    // clang-format off
    //    |                 |
    //    N    remove(N)    R
    //   / \   ========>   /
    //  L   R             L
    // clang-format on
    NodePtr right = node->right;
    swapNode(node, right);
    right->right = node->right;
    node = right;
    node->updateSize();
    fixBalance(node);
  } else {
    // Case 5: both left and right child, right child is not a leaf
    // clang-format off
    //   Step 1. find the node N with the smallest key
    //           and its parent P on the right subtree
    //   Step 2. swap S and N
    //   Step 3. remove node N like Case 1 or Case 3
    //   Step 4. update size for all nodes on the path
    //           from S to P
    //     |                  |
    //     N                  S                 |
    //    / \                / \                S
    //   L  ..  swap(N, S)  L  ..  remove(N)   / \
    //       |  =========>      |  ========>  L  ..
    //       P                  P                |
    //      / \                / \               P
    //     S  ..              N  ..             / \
    //      \                  \               R  ..
    //       R                  R
    //
    // clang-format on

    std::stack<NodePtr> path;

    // Step 1
    NodePtr successor = node->right;
    NodePtr parent = node;
    path.push(node);

    while (successor->left != nullptr) {
      path.push(successor);
```

```
      parent = successor;
      successor = parent->left;
    }

    // Step 2
    swapNode(node, successor);

    // Step 3
    parent->left = node->right;
    // Restore node
    node = successor;

    // Step 4
    while (!path.empty()) {
      path.top()->updateSize();
      path.pop();
    }
  }

  return true;
}
```

值得注意的是，在上述代码的 Case 5 中使用后继节点 S（也可以选择前驱节点）替换待删除节点 N 并删除替换后的 N 以后，需要更新替换前 S 节点的父节点 P 到替换后的 S 节点这条路径（如代码中注释所示）上的所有节点的 `size` 字段。本文的实现选择使用栈依次记录路径上的节点，最后再按遍历的相反顺序出栈进行更新。

查询排名

由于 SBT 节点中储存了子树节点个数的信息，因此可以在 $O(\log n)$ 的时间复杂度下查询某个 `key` 的排名（或者大于/小于某个 `key` 的节点个数）。示例代码如下：

```
USize countLess(ConstNodePtr node, K key, bool countEqual = false) const {
  if (node == nullptr) {
    return 0;
  } else if (key < node->key) {
    return countLess(node->left, key, countEqual);
  } else if (key > node->key) {
    return size(node->left) + 1 + countLess(node->right, key, countEqual);
  } else {
    return size(node->left) + (countEqual ? 1 : 0);
  }
}

USize countGreater(ConstNodePtr node, K key, bool countEqual = false) const {
  if (node == nullptr) {
    return 0;
  } else if (key < node->key) {
    return size(node->right) + 1 + countGreater(node->left, key, countEqual);
  } else if (key > node->key) {
    return countGreater(node->right, key, countEqual);
  } else {
    return size(node->right) + (countEqual ? 1 : 0);
  }
}
```

参考代码

下面的代码是用 SBT 实现的 `Map`，即有序不可重映射：

" 完整代码"

```cpp
/**
 * @file SizeBalancedTreeMap.hpp
 * @brief An SizeBalancedTree-based map implementation
 * @details The map is sorted according to the natural ordering of its
 *  keys or by a {@code Compare} function provided; This implementation
 *  provides guaranteed log(n) time cost for the contains, get, insert
 *  and remove operations.
 * @author [r.ivance](https://github.com/RIvance)
 */

#ifndef SIZE_BALANCED_TREE_MAP_HPP
#define SIZE_BALANCED_TREE_MAP_HPP

#include <cassert>
#include <cstddef>
#include <cstdint>
#include <functional>
#include <memory>
#include <stack>
#include <utility>
#include <vector>

/**
 * An SizeBalancedTree-based map implementation
 * http://wcipeg.com/wiki/Size_Balanced_Tree
 * @tparam Key the type of keys maintained by this map
 * @tparam Value the type of mapped values
 * @tparam Compare the compare function
 */
template <typename Key, typename Value, typename Compare = std::less<Key> >
class SizeBalancedTreeMap {
 private:
  using USize = size_t;

  Compare compare = Compare();

 public:
  struct Entry {
    Key key;
    Value value;

    bool operator==(const Entry &rhs) const noexcept {
      return this->key == rhs.key && this->value == rhs.value;
    }

    bool operator!=(const Entry &rhs) const noexcept {
      return this->key != rhs.key || this->value != rhs.value;
    }
  };
```

```
private:
 struct Node {
   using Ptr = std::shared_ptr<Node>;
   using Provider = const std::function<Ptr(void)> &;
   using Consumer = const std::function<void(const Ptr &)> &;

   Key key;
   Value value{};

   Ptr left = nullptr;
   Ptr right = nullptr;

   USize size = 1;

   explicit Node(Key k) : key(std::move(k)) {}

   explicit Node(Key k, Value v) : key(std::move(k)), value(std::move(v)) {}

   ~Node() = default;

   inline bool isLeaf() const noexcept {
     return this->left == nullptr && this->right == nullptr;
   }

   inline void updateSize() {
     USize leftSize = this->left != nullptr ? this->left->size : 0;
     USize rightSize = this->right != nullptr ? this->right->size : 0;
     this->size = leftSize + rightSize + 1;
   }

   inline Entry entry() const { return Entry{key, value}; }

   static Ptr from(const Key &k) { return std::make_shared<Node>(Node(k)); }

   static Ptr from(const Key &k, const Value &v) {
     return std::make_shared<Node>(Node(k, v));
   }
 };

 using NodePtr = typename Node::Ptr;
 using ConstNodePtr = const NodePtr &;
 using NodeProvider = typename Node::Provider;
 using NodeConsumer = typename Node::Consumer;

 NodePtr root = nullptr;

 using K = const Key &;
 using V = const Value &;

public:
 using EntryList = std::vector<Entry>;
 using KeyValueConsumer = const std::function<void(K, V)> &;
 using MutKeyValueConsumer = const std::function<void(K, Value &)> &;
```

```
using KeyValueFilter = const std::function<bool(K, V)> &;

class NoSuchMappingException : protected std::exception {
 private:
  const char *message;

 public:
  explicit NoSuchMappingException(const char *msg) : message(msg) {}

  const char *what() const noexcept override { return message; }
};

SizeBalancedTreeMap() noexcept = default;

/**
 * Returns the number of entries in this map.
 * @return size_t
 */
inline USize size() const noexcept {
  if (this->root != nullptr) {
    return this->root->size;
  } else {
    return 0;
  }
}

/**
 * Returns true if this collection contains no elements.
 * @return bool
 */
inline bool empty() const noexcept { return this->root == nullptr; }

/**
 * Removes all of the elements from this map.
 */
void clear() noexcept { this->root = nullptr; }

/**
 * Returns the value to which the specified key is mapped; If this map
 * contains no mapping for the key, a {@code NoSuchMappingException} will
 * be thrown.
 * @param key
 * @return SizeBalancedTreeMap<Key, Value>::Value
 * @throws NoSuchMappingException
 */
Value get(K key) const {
  if (this->root == nullptr) {
    throw NoSuchMappingException("Invalid key");
  } else {
    NodePtr node = this->getNode(this->root, key);
    if (node != nullptr) {
      return node->value;
    } else {
      throw NoSuchMappingException("Invalid key");
```

```
      }
    }
  }

  /**
   * Returns the value to which the specified key is mapped; If this map
   * contains no mapping for the key, a new mapping with a default value
   * will be inserted.
   * @param key
   * @return SizeBalancedTreeMap<Key, Value>::Value &
   */
  Value &getOrDefault(K key) {
    if (this->root == nullptr) {
      this->root = Node::from(key);
      return this->root->value;
    } else {
      return this
          ->getNodeOrProvide(this->root, key,
                             [&key]() { return Node::from(key); })
          ->value;
    }
  }

  /**
   * Returns true if this map contains a mapping for the specified key.
   * @param key
   * @return bool
   */
  bool contains(K key) const {
    return this->getNode(this->root, key) != nullptr;
  }

  /**
   * Associates the specified value with the specified key in this map.
   * @param key
   * @param value
   */
  void insert(K key, V value) {
    if (this->root == nullptr) {
      this->root = Node::from(key, value);
    } else {
      this->insert(this->root, key, value);
    }
  }

  /**
   * If the specified key is not already associated with a value, associates
   * it with the given value and returns true, else returns false.
   * @param key
   * @param value
   * @return bool
   */
  bool insertIfAbsent(K key, V value) {
    USize sizeBeforeInsertion = this->size();
```

```
    if (this->root == nullptr) {
      this->root = Node::from(key, value);
    } else {
      this->insert(this->root, key, value, false);
    }
    return this->size() > sizeBeforeInsertion;
  }

  /**
   * If the specified key is not already associated with a value, associates
   * it with the given value and returns the value, else returns the associated
   * value.
   * @param key
   * @param value
   * @return SizeBalancedTreeMap<Key, Value>::Value &
   */
  Value &getOrInsert(K key, V value) {
    if (this->root == nullptr) {
      this->root = Node::from(key, value);
      return root->value;
    } else {
      NodePtr node = getNodeOrProvide(this->root, key,
                                      [&]() { return Node::from(key, value); });
      return node->value;
    }
  }

  Value operator[](K key) const { return this->get(key); }

  Value &operator[](K key) { return this->getOrDefault(key); }

  /**
   * Removes the mapping for a key from this map if it is present;
   * Returns true if the mapping is present else returns false
   * @param key the key of the mapping
   * @return bool
   */
  bool remove(K key) {
    if (this->root == nullptr) {
      return false;
    } else {
      return this->remove(this->root, key, [](ConstNodePtr) {});
    }
  }

  /**
   * Removes the mapping for a key from this map if it is present and returns
   * the value which is mapped to the key; If this map contains no mapping for
   * the key, a {@code NoSuchMappingException} will be thrown.
   * @param key
   * @return SizeBalancedTreeMap<Key, Value>::Value
   * @throws NoSuchMappingException
   */
  Value getAndRemove(K key) {
```

```
    Value result;
    NodeConsumer action = [&](ConstNodePtr node) { result = node->value; };

    if (root == nullptr) {
      throw NoSuchMappingException("Invalid key");
    } else {
      if (remove(this->root, key, action)) {
        return result;
      } else {
        throw NoSuchMappingException("Invalid key");
      }
    }
  }

  /**
   * Gets the entry corresponding to the specified key; if no such entry
   * exists, returns the entry for the least key greater than the specified
   * key; if no such entry exists (i.e., the greatest key in the Tree is less
   * than the specified key), a {@code NoSuchMappingException} will be thrown.
   * @param key
   * @return SizeBalancedTreeMap<Key, Value>::Entry
   * @throws NoSuchMappingException
   */
  Entry getCeilingEntry(K key) const {
    if (this->root == nullptr) {
      throw NoSuchMappingException("No ceiling entry in this map");
    }

    NodePtr node = this->root;
    std::stack<NodePtr> ancestors;

    while (node != nullptr) {
      if (key == node->key) {
        return node->entry();
      }

      if (compare(key, node->key)) {
        /* key < node->key */
        if (node->left != nullptr) {
          ancestors.push(node);
          node = node->left;
        } else {
          return node->entry();
        }
      } else {
        /* key > node->key */
        if (node->right != nullptr) {
          ancestors.push(node);
          node = node->right;
        } else {
          if (ancestors.empty()) {
            throw NoSuchMappingException("No ceiling entry in this map");
          }
```

```
          NodePtr parent = ancestors.top();
          ancestors.pop();

          while (node == parent->right) {
            node = parent;
            if (!ancestors.empty()) {
              parent = ancestors.top();
              ancestors.pop();
            } else {
              throw NoSuchMappingException("No ceiling entry in this map");
            }
          }

          return parent->entry();
        }
      }
    }

    throw NoSuchMappingException("No ceiling entry in this map");
  }

  /**
   * Gets the entry corresponding to the specified key; if no such entry exists,
   * returns the entry for the greatest key less than the specified key;
   * if no such entry exists, a {@code NoSuchMappingException} will be thrown.
   * @param key
   * @return SizeBalancedTreeMap<Key, Value>::Entry
   * @throws NoSuchMappingException
   */
  Entry getFloorEntry(K key) const {
    if (this->root == nullptr) {
      throw NoSuchMappingException("No floor entry exists in this map");
    }

    NodePtr node = this->root;
    std::stack<NodePtr> ancestors;

    while (node != nullptr) {
      if (key == node->key) {
        return node->entry();
      }

      if (compare(key, node->key)) {
        /* key < node->key */
        if (node->left != nullptr) {
          ancestors.push(node);
          node = node->left;
        } else {
          if (ancestors.empty()) {
            throw NoSuchMappingException("No floor entry exists in this map");
          }

          NodePtr parent = ancestors.top();
          ancestors.pop();
```

```
        while (node == parent->left) {
          node = parent;
          if (!ancestors.empty()) {
            parent = ancestors.top();
            ancestors.pop();
          } else {
            throw NoSuchMappingException("No floor entry exists in this map");
          }
        }

        return parent->entry();
      }
    } else {
      /* key > node->key */
      if (node->right != nullptr) {
        ancestors.push(node);
        node = node->right;
      } else {
        return node->entry();
      }
    }
  }

  throw NoSuchMappingException("No floor entry exists in this map");
}

/**
 * Gets the entry for the least key greater than the specified
 * key; if no such entry exists, returns the entry for the least
 * key greater than the specified key; if no such entry exists,
 * a {@code NoSuchMappingException} will be thrown.
 * @param key
 * @return SizeBalancedTreeMap<Key, Value>::Entry
 * @throws NoSuchMappingException
 */
Entry getHigherEntry(K key) {
  if (this->root == nullptr) {
    throw NoSuchMappingException("No higher entry exists in this map");
  }

  NodePtr node = this->root;
  std::stack<NodePtr> ancestors;

  while (node != nullptr) {
    if (compare(key, node->key)) {
      /* key < node->key */
      if (node->left != nullptr) {
        ancestors.push(node);
        node = node->left;
      } else {
        return node->entry();
      }
    } else {
```

```
        /* key >= node->key */
        if (node->right != nullptr) {
          ancestors.push(node);
          node = node->right;
        } else {
          if (ancestors.empty()) {
            throw NoSuchMappingException("No higher entry exists in this map");
          }

          NodePtr parent = ancestors.top();
          ancestors.pop();

          while (node == parent->right) {
            node = parent;
            if (!ancestors.empty()) {
              parent = ancestors.top();
              ancestors.pop();
            } else {
              throw NoSuchMappingException(
                  "No higher entry exists in this map");
            }
          }

          return parent->entry();
        }
      }
    }

    throw NoSuchMappingException("No higher entry exists in this map");
  }

  /**
   * Returns the entry for the greatest key less than the specified key; if
   * no such entry exists (i.e., the least key in the Tree is greater than
   * the specified key), a {@code NoSuchMappingException} will be thrown.
   * @param key
   * @return SizeBalancedTreeMap<Key, Value>::Entry
   * @throws NoSuchMappingException
   */
  Entry getLowerEntry(K key) const {
    if (this->root == nullptr) {
      throw NoSuchMappingException("No lower entry exists in this map");
    }

    NodePtr node = this->root;
    std::stack<NodePtr> ancestors;

    while (node != nullptr) {
      if (compare(key, node->key) || key == node->key) {
        /* key <= node->key */
        if (node->left != nullptr) {
          ancestors.push(node);
          node = node->left;
        } else {
```

```
        if (ancestors.empty()) {
          throw NoSuchMappingException("No lower entry exists in this map");
        }

        NodePtr parent = ancestors.top();
        ancestors.pop();

        while (node == parent->left) {
          node = parent;
          if (!ancestors.empty()) {
            parent = ancestors.top();
            ancestors.pop();
          } else {
            throw NoSuchMappingException("No lower entry exists in this map");
          }
        }

        return parent->entry();
      }
    } else {
      /* key > node->key */
      if (node->right != nullptr) {
        ancestors.push(node);
        node = node->right;
      } else {
        return node->entry();
      }
    }
  }

  throw NoSuchMappingException("No lower entry exists in this map");
}

/**
 * Count the number of entries that are less than the given key
 * @param key
 * @return USize
 */
USize countLessThan(K key) { return this->countLess(this->root, key); }

/**
 * Count the number of entries that are less or equal to the given key
 * @param key
 * @return USize
 */
USize countLessOrEqualTo(K key) {
  return this->countLess(this->root, key, true);
}

/**
 * Count the number of entries that are greater than the given key
 * @param key
 * @return USize
 */
```

```
  USize countGreaterThan(K key) { return this->countGreater(this->root, key); }

  /**
   * Count the number of entries that are greater or equal to the given key
   * @param key
   * @return USize
   */
  USize countGreaterOrEqualTo(K key) {
    return this->countGreater(this->root, key, true);
  }

  /**
   * Remove all entries that satisfy the filter condition.
   * @param filter
   */
  void removeAll(KeyValueFilter filter) {
    std::vector<Key> keys;
    this->inorderTraversal([&](ConstNodePtr node) {
      if (filter(node->key, node->value)) {
        keys.push_back(node->key);
      }
    });
    for (const Key &key : keys) {
      this->remove(key);
    }
  }

  /**
   * Performs the given action for each key and value entry in this map.
   * The value is immutable for the action.
   * @param action
   */
  void forEach(KeyValueConsumer action) const {
    this->inorderTraversal(
        [&](ConstNodePtr node) { action(node->key, node->value); });
  }

  /**
   * Performs the given action for each key and value entry in this map.
   * The value is mutable for the action.
   * @param action
   */
  void forEachMut(MutKeyValueConsumer action) {
    this->inorderTraversal(
        [&](ConstNodePtr node) { action(node->key, node->value); });
  }

  /**
   * Returns a list containing all of the entries in this map.
   * @return SizeBalancedTreeMap<Key, Value>::EntryList
   */
  EntryList toEntryList() const {
    EntryList entryList;
    this->inorderTraversal(
```

```
        [&](ConstNodePtr node) { entryList.push_back(node->entry()); });
    return entryList;
  }

 private:
  static USize size(ConstNodePtr node) {
    return node != nullptr ? node->size : 0;
  }

  static void rotateLeft(NodePtr &node) {
    assert(node != nullptr);
    // clang-format off
    //     |                       |
    //     N                       S
    //    / \     l-rotate(N)     / \
    //   L   S    ==========>    N   R
    //      / \                 / \
    //     M   R               L   M
    // clang-format on
    NodePtr successor = node->right;
    node->right = successor->left;
    successor->left = node;

    node->updateSize();
    successor->updateSize();

    node = successor;
  }

  static void rotateRight(NodePtr &node) {
    assert(node != nullptr);
    // clang-format off
    //       |                   |
    //       N                   S
    //      / \   r-rotate(N)   / \
    //     S   R  ==========>  L   N
    //    / \                     / \
    //   L   M                   M   R
    // clang-format on
    NodePtr successor = node->left;
    node->left = successor->right;
    successor->right = node;

    node->updateSize();
    successor->updateSize();

    node = successor;
  }

  static void swapNode(NodePtr &lhs, NodePtr &rhs) {
    std::swap(lhs->key, rhs->key);
    std::swap(lhs->value, rhs->value);
    std::swap(lhs, rhs);
  }
```

```cpp
static void fixBalance(NodePtr &node) {
  if (node == nullptr) {
    return;
  }

  if (node->left != nullptr) {
    if (size(node->left->left) > size(node->right)) {
      // clang-format off
      //       |                       |
      //       N                       L
      //      / \    r-rotate(N)      / \
      //     L  <R>  ==========>   [M]  N
      //    /                             \
      //  [M]                            <R>
      // clang-format on
      rotateRight(node);
      fixBalance(node->right);
      fixBalance(node);
      return;
    } else if (size(node->left->right) > size(node->right)) {
      // clang-format off
      //     |                     |                     |
      //     N                     N                    [M]
      //    / \    l-rotate(L)    / \    r-rotate(N)    / \
      //   L  <R>  ==========> [M] <R>   ==========>   L   N
      //    \                   /                           \
      //    [M]                L                            <R>
      // clang-format on
      rotateLeft(node->left);
      rotateRight(node);
      fixBalance(node->left);
      fixBalance(node->right);
      fixBalance(node);
      return;
    }
  }

  if (node->right != nullptr) {
    if (size(node->right->right) > size(node->left)) {
      // clang-format off
      //     |                       |
      //     N                       R
      //    / \     l-rotate(N)     / \
      //  <L>  R    ==========>    N [M]
      //        \                 /
      //        [M]             <L>
      // clang-format on
      rotateLeft(node);
      fixBalance(node->left);
      fixBalance(node);
      return;
    } else if (size(node->right->left) > size(node->left)) {
      // clang-format off
```

```
      //     |                     |                       |
      //     N                     N                      [M]
      //    / \    r-rotate(R)    / \      l-rotate(N)    / \
      //  <L>  R   ==========>  <L> [M]    ==========>   N   R
      //      /                        \                /
      //    [M]                         R             <L>
      // clang-format on
      rotateRight(node->right);
      rotateLeft(node);
      fixBalance(node->left);
      fixBalance(node->right);
      fixBalance(node);
      return;
    }
  }
}

NodePtr getNodeOrProvide(NodePtr &node, K key, NodeProvider provide) {
  assert(node != nullptr);

  if (key == node->key) {
    return node;
  }

  assert(key != node->key);

  NodePtr result;

  if (compare(key, node->key)) {
    /* key < node->key */
    if (node->left == nullptr) {
      result = node->left = provide();
      node->updateSize();
    } else {
      result = getNodeOrProvide(node->left, key, provide);
      node->updateSize();
      fixBalance(node);
    }
  } else {
    /* key > node->key */
    if (node->right == nullptr) {
      result = node->right = provide();
      node->updateSize();
    } else {
      result = getNodeOrProvide(node->right, key, provide);
      node->updateSize();
      fixBalance(node);
    }
  }

  return result;
}

NodePtr getNode(ConstNodePtr node, K key) const {
```

```
    assert(node != nullptr);

    if (key == node->key) {
      return node;
    }

    if (compare(key, node->key)) {
      /* key < node->key */
      return node->left == nullptr ? nullptr : getNode(node->left, key);
    } else {
      /* key > node->key */
      return node->right == nullptr ? nullptr : getNode(node->right, key);
    }
  }

  void insert(NodePtr &node, K key, V value, bool replace = true) {
    assert(node != nullptr);

    if (key == node->key) {
      if (replace) {
        node->value = value;
      }
      return;
    }

    assert(key != node->key);

    if (compare(key, node->key)) {
      /* key < node->key */
      if (node->left == nullptr) {
        node->left = Node::from(key, value);
        node->updateSize();
      } else {
        insert(node->left, key, value, replace);
        node->updateSize();
        fixBalance(node);
      }
    } else {
      /* key > node->key */
      if (node->right == nullptr) {
        node->right = Node::from(key, value);
        node->updateSize();
      } else {
        insert(node->right, key, value, replace);
        node->updateSize();
        fixBalance(node);
      }
    }
  }

  bool remove(NodePtr &node, K key, NodeConsumer action) {
    assert(node != nullptr);

    if (key != node->key) {
```

```
    if (compare(key, node->key)) {
      /* key < node->key */
      NodePtr &left = node->left;
      if (left != nullptr && remove(left, key, action)) {
        node->updateSize();
        fixBalance(node);
        return true;
      } else {
        return false;
      }
    } else {
      /* key > node->key */
      NodePtr &right = node->right;
      if (right != nullptr && remove(right, key, action)) {
        node->updateSize();
        fixBalance(node);
        return true;
      } else {
        return false;
      }
    }
  }

  assert(key == node->key);
  action(node);

  if (node->isLeaf()) {
    // Case 1: no child
    node = nullptr;
  } else if (node->right == nullptr) {
    // clang-format off
    // Case 2: left child only
    //     P
    //     |  remove(N)  P
    //     N  ========>  |
    //    /              L
    //   L
    // clang-format on
    node = node->left;
  } else if (node->left == nullptr) {
    // clang-format off
    // Case 3: right child only
    //   P
    //   |    remove(N)  P
    //   N    ========>  |
    //    \              R
    //     R
    // clang-format on
    node = node->right;
  } else if (node->right->left == nullptr) {
    // clang-format off
    // Case 4: both left and right child, right child has no left child
    //    |                 |
    //    N    remove(N)    R
```

```
      //   / \   ========>   /
      //  L   R             L
      // clang-format on
      NodePtr right = node->right;
      swapNode(node, right);
      right->right = node->right;
      node = right;
      node->updateSize();
      fixBalance(node);
    } else {
      // clang-format off
      // Case 5: both left and right child, right child is not a leaf
      //   Step 1. find the node N with the smallest key
      //           and its parent P on the right subtree
      //   Step 2. swap S and N
      //   Step 3. remove node N like Case 1 or Case 3
      //   Step 4. update size for all nodes on the path
      //           from S to P
      //     |                  |
      //     N                  S                 |
      //    / \                / \                S
      //   L  ..  swap(N, S)  L  ..  remove(N)   / \
      //       |  =========>      |  ========>  L  ..
      //       P                  P                 |
      //      / \                / \                P
      //     S  ..              N  ..              / \
      //      \                  \                R  ..
      //       R                  R
      // clang-format on

      std::stack<NodePtr> path;

      // Step 1
      NodePtr successor = node->right;
      NodePtr parent = node;
      path.push(node);

      while (successor->left != nullptr) {
        path.push(successor);
        parent = successor;
        successor = parent->left;
      }

      // Step 2
      swapNode(node, successor);

      // Step 3
      parent->left = node->right;
      // Restore node
      node = successor;

      // Step 4
      while (!path.empty()) {
        path.top()->updateSize();
```

```
        path.pop();
      }
    }

    return true;
  }

  USize countLess(ConstNodePtr node, K key, bool countEqual = false) const {
    if (node == nullptr) {
      return 0;
    } else if (key < node->key) {
      return countLess(node->left, key, countEqual);
    } else if (key > node->key) {
      return size(node->left) + 1 + countLess(node->right, key, countEqual);
    } else {
      return size(node->left) + (countEqual ? 1 : 0);
    }
  }

  USize countGreater(ConstNodePtr node, K key, bool countEqual = false) const {
    if (node == nullptr) {
      return 0;
    } else if (key < node->key) {
      return size(node->right) + 1 + countGreater(node->left, key, countEqual);
    } else if (key > node->key) {
      return countGreater(node->right, key, countEqual);
    } else {
      return size(node->right) + (countEqual ? 1 : 0);
    }
  }

  void inorderTraversal(NodeConsumer action) const {
    if (this->root == nullptr) {
      return;
    }

    std::stack<NodePtr> stack;
    NodePtr node = this->root;

    while (node != nullptr || !stack.empty()) {
      while (node != nullptr) {
        stack.push(node);
        node = node->left;
      }
      if (!stack.empty()) {
        node = stack.top();
        stack.pop();
        action(node);
        node = node->right;
      }
    }
  }
};
```

```
#endif  // SIZE_BALANCED_TREE_MAP_HPP
```

10.17.6 AVL 树

AVL 树，是一种平衡的二叉搜索树。由于各种算法教材上对 AVL 的介绍十分冗长，造成了很多人对 AVL 树复杂、不实用的印象。但实际上，AVL 树的原理简单，实现也并不复杂。

性质

1. 空二叉树是一个 AVL 树
2. 如果 T 是一棵 AVL 树，那么其左右子树也是 AVL 树，并且 $|h(ls)-h(rs)|\leq 1$，h 是其左右子树的高度
3. 树高为 $O(\log n)$

平衡因子：右子树高度 - 左子树高度

" 树高的证明"

设 f_n 为高度为 n 的 AVL 树所包含的最少节点数，则有

$$f_n=\begin{cases}1 & (n=1)\\ 2 & (n=2)\\ f_{n-1}+f_{n-2}+1 & (n>2)\end{cases}$$

根据常系数非齐次线性差分方程的解法，$\{f_n+1\}$ 是一个斐波那契数列。这里 f_n 的通项为：

$$f_n=\frac{5+2\sqrt{5}}{5}\left(\frac{1+\sqrt{5}}{2}\right)^n+\frac{5-2\sqrt{5}}{5}\left(\frac{1-\sqrt{5}}{2}\right)^n-1$$

斐波那契数列以指数的速度增长，对于树高 n 有：

$$n<\log_{\frac{1+\sqrt{5}}{2}}(f_n+1)<\frac{3}{2}\log_2(f_n+1)$$

因此 AVL 树的高度为 $O(\log f_n)$，这里的 f_n 为结点数。

过程

插入结点

与 BST（二叉搜索树）中类似，先进行一次失败的查找来确定插入的位置，插入节点后根据平衡因子来决定是否需要调整。

删除结点

删除和 BST 类似，将结点与后继交换后再删除。

删除会导致树高以及平衡因子变化，这时需要沿着被删除结点到根的路径来调整这种变化。

平衡的维护

插入或删除节点后，可能会造成 AVL 树的性质 2 被破坏。因此，需要沿着从被插入/删除的节点到根的路径对树进行维护。如果对于某一个节点，性质 2 不再满足，由于我们只插入/删除了一个节点，对树高的影响不超过 1，因此该节点的平衡因子的绝对值至多为 2。由于对称性，我们在此只讨论左子树的高度比右子树大 2 的情况，即下图中 $h(B)-h(E)=2$。此时，还需要根据 $h(A)$ 和 $h(C)$ 的大小关系分两种情况讨论。需要注意的是，由于我们是自底向上维护平衡的，因此对节点 D 的所有后代来说，性质 2 仍然是被满足的。

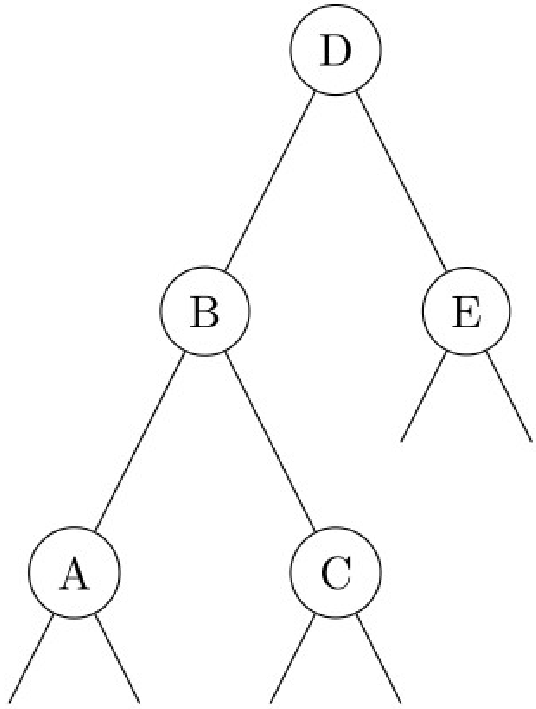


图 10.76

$h(A) \geq h(C)$

设 $h(E) = x$，则有

$$\begin{cases} h(B) = x + 2 \\ h(A) = x + 1 \\ x \leq h(C) \leq x + 1 \end{cases}$$

其中 $h(C) \geq x$ 是由于节点 B 满足性质 2，因此 $h(C)$ 和 $h(A)$ 的差不会超过 1。此时我们对节点 D 进行一次右旋操作（旋转操作与其它类型的平衡二叉搜索树相同），如下图所示。

显然节点 A、C、E 的高度不发生变化，并且有

$$\begin{cases} 0 \leq h(C) - h(E) \leq 1 \\ x + 1 \leq h'(D) = \max(h(C), h(E)) + 1 = h(C) + 1 \leq x + 2 \\ 0 \leq h'(D) - h(A) \leq 1 \end{cases}$$

因此旋转后的节点 B 和 D 也满足性质 2。

$h(A) < h(C)$

设 $h(E) = x$，则与刚才同理，有

$$\begin{cases} h(B) = x + 2 \\ h(C) = x + 1 \\ h(A) = x \end{cases}$$

此时我们先对节点 B 进行一次左旋操作，再对节点 D 进行一次右旋操作，如下图所示。

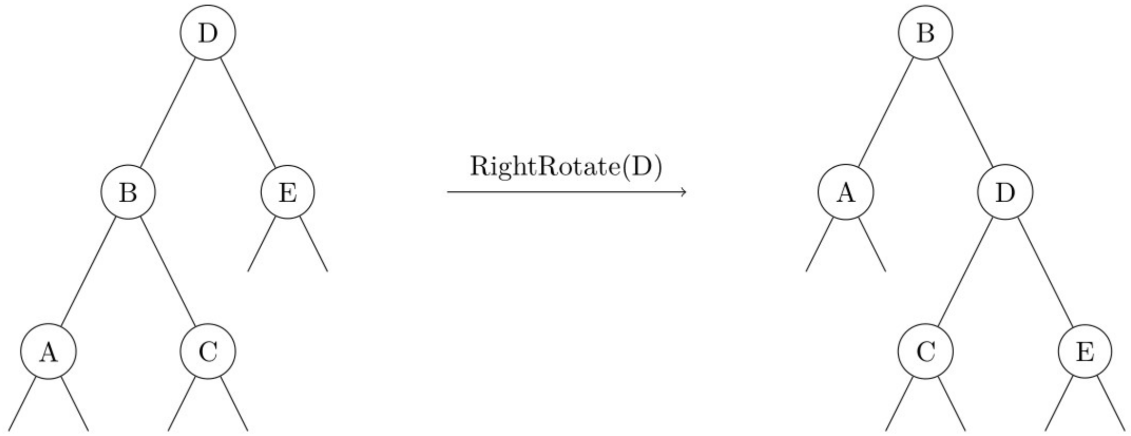

D
B
E
A
C
RightRotate(D)
B
A
D
C
E

图 10.77

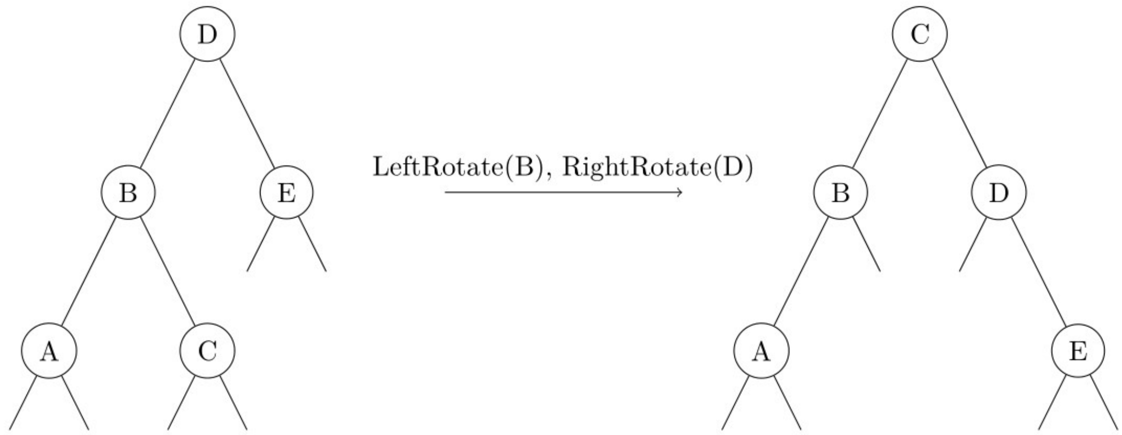

D
B
E
A
C
LeftRotate(B), RightRotate(D)
C
B
D
A
E

图 10.78

显然节点 A、E 的高度不发生变化，并且 B 的新右儿子和 D 的新左儿子分别为 C 原来的左右儿子，则有

$$\begin{cases} x-1 \le h'(rs_B), h'(ls_D) \le x \\ 0 \le h(A) - h'(rs_B) \le 1 \\ 0 \le h(E) - h'(ls_D) \le 1 \\ h'(B) = \max(h(A), h'(rs_B)) + 1 = x+1 \\ h'(D) = \max(h(E), h'(ls_D)) + 1 = x+1 \\ h'(B) - h'(D) = 0 \end{cases}$$

因此旋转后的节点 B、C、D 也满足性质 2。最后给出对于一个节点维护平衡操作的伪代码。

" 实现"

```
Maintain-Balanced(p)
    if h[ls[p]] - h[rs[p]] == 2
        if h[ls[ls[p]]] >= h[rs[ls[p]]]
            Right-Rotate(p)
        else
            Left-Rotate(ls[p])
            Right-Rotate(p)
    else if h[ls[p]] - h[rs[p]] == -2
        if h[ls[rs[p]]] <= h[rs[rs[p]]]
            Left-Rotate(p)
        else
            Right-Rotate(rs[p])
            Left-Rotate(p)
```

与其他平衡二叉搜索树相同，AVL 树中节点的高度、子树大小等信息需要在旋转时进行维护。

其他操作

AVL 树的其他操作（Predecessor、Successor、Select、Rank 等）与普通的二叉搜索树相同。

参考代码

下面的代码是用 AVL 树实现的 `Map`，即有序不可重映射：

" 参考代码"

```
/**
 * @brief An AVLTree-based map implementation
 * @details The map is sorted according to the natural ordering of its
 *  keys or by a {@code Compare} function provided; This implementation
 *  provides guaranteed log(n) time cost for the contains, get, insert
 *  and remove operations.
 */

#ifndef AVLTREE_MAP_HPP
#define AVLTREE_MAP_HPP

#include <cassert>
#include <cstddef>
#include <cstdint>
```

```cpp
#include <functional>
#include <memory>
#include <stack>
#include <utility>
#include <vector>

/**
 * An AVLTree-based map implementation
 * https://en.wikipedia.org/wiki/AVL_tree
 * @tparam Key the type of keys maintained by this map
 * @tparam Value the type of mapped values
 * @tparam Compare
 */
template <typename Key, typename Value, typename Compare = std::less<Key> >
class AvlTreeMap {
 private:
  using USize = size_t;
  using Factor = int64_t;

  Compare compare = Compare();

 public:
  struct Entry {
    Key key;
    Value value;

    bool operator==(const Entry &rhs) const noexcept {
      return this->key == rhs.key && this->value == rhs.value;
    }

    bool operator!=(const Entry &rhs) const noexcept {
      return this->key != rhs.key || this->value != rhs.value;
    }
  };

 private:
  struct Node {
    using Ptr = std::shared_ptr<Node>;
    using Provider = const std::function<Ptr(void)> &;
    using Consumer = const std::function<void(const Ptr &)> &;

    Key key;
    Value value{};

    Ptr left = nullptr;
    Ptr right = nullptr;

    USize height = 1;

    explicit Node(Key k) : key(std::move(k)) {}

    explicit Node(Key k, Value v) : key(std::move(k)), value(std::move(v)) {}

    ~Node() = default;
```

```
    inline bool isLeaf() const noexcept {
      return this->left == nullptr && this->right == nullptr;
    }

    inline void updateHeight() noexcept {
      if (this->isLeaf()) {
        this->height = 1;
      } else if (this->left == nullptr) {
        this->height = this->right->height + 1;
      } else if (this->right == nullptr) {
        this->height = this->left->height + 1;
      } else {
        this->height = std::max(left->height, right->height) + 1;
      }
    }

    inline Factor factor() const noexcept {
      if (this->isLeaf()) {
        return 0;
      } else if (this->left == nullptr) {
        return (Factor)this->right->height;
      } else if (this->right == nullptr) {
        return (Factor) - this->left->height;
      } else {
        return (Factor)(this->right->height - this->left->height);
      }
    }

    inline Entry entry() const { return Entry{key, value}; }

    static Ptr from(const Key &k) { return std::make_shared<Node>(Node(k)); }

    static Ptr from(const Key &k, const Value &v) {
      return std::make_shared<Node>(Node(k, v));
    }
  };

  using NodePtr = typename Node::Ptr;
  using ConstNodePtr = const NodePtr &;
  using NodeProvider = typename Node::Provider;
  using NodeConsumer = typename Node::Consumer;

  NodePtr root = nullptr;
  USize count = 0;

  using K = const Key &;
  using V = const Value &;

 public:
  using EntryList = std::vector<Entry>;
  using KeyValueConsumer = const std::function<void(K, V)> &;
  using MutKeyValueConsumer = const std::function<void(K, Value &)> &;
  using KeyValueFilter = const std::function<bool(K, V)> &;
```

```cpp
class NoSuchMappingException : protected std::exception {
 private:
  const char *message;

 public:
  explicit NoSuchMappingException(const char *msg) : message(msg) {}

  const char *what() const noexcept override { return message; }
};

AvlTreeMap() noexcept = default;

/**
 * Returns the number of entries in this map.
 * @return size_t
 */
inline USize size() const noexcept { return this->count; }

/**
 * Returns true if this collection contains no elements.
 * @return bool
 */
inline bool empty() const noexcept { return this->count == 0; }

/**
 * Removes all of the elements from this map.
 */
void clear() noexcept {
  this->root = nullptr;
  this->count = 0;
}

/**
 * Returns the value to which the specified key is mapped; If this map
 * contains no mapping for the key, a {@code NoSuchMappingException} will
 * be thrown.
 * @param key
 * @return AvlTreeMap<Key, Value>::Value
 * @throws NoSuchMappingException
 */
Value get(K key) const {
  if (this->root == nullptr) {
    throw NoSuchMappingException("Invalid key");
  } else {
    NodePtr node = this->getNode(this->root, key);
    if (node != nullptr) {
      return node->value;
    } else {
      throw NoSuchMappingException("Invalid key");
    }
  }
}
```

```
  /**
   * Returns the value to which the specified key is mapped; If this map
   * contains no mapping for the key, a new mapping with a default value
   * will be inserted.
   * @param key
   * @return AvlTreeMap<Key, Value>::Value &
   */
  Value &getOrDefault(K key) {
    if (this->root == nullptr) {
      this->root = Node::from(key);
      this->count += 1;
      return this->root->value;
    } else {
      return this
          ->getNodeOrProvide(this->root, key,
                             [&key]() { return Node::from(key); })
          ->value;
    }
  }

  /**
   * Returns true if this map contains a mapping for the specified key.
   * @param key
   * @return bool
   */
  bool contains(K key) const {
    return this->getNode(this->root, key) != nullptr;
  }

  /**
   * Associates the specified value with the specified key in this map.
   * @param key
   * @param value
   */
  void insert(K key, V value) {
    if (this->root == nullptr) {
      this->root = Node::from(key, value);
      this->count += 1;
    } else {
      this->insert(this->root, key, value);
    }
  }

  /**
   * If the specified key is not already associated with a value, associates
   * it with the given value and returns true, else returns false.
   * @param key
   * @param value
   * @return bool
   */
  bool insertIfAbsent(K key, V value) {
    USize sizeBeforeInsertion = this->size();
    if (this->root == nullptr) {
      this->root = Node::from(key, value);
```

```
            this->count += 1;
        } else {
            this->insert(this->root, key, value, false);
        }
        return this->size() > sizeBeforeInsertion;
    }

    /**
     * If the specified key is not already associated with a value, associates
     * it with the given value and returns the value, else returns the associated
     * value.
     * @param key
     * @param value
     * @return
     */
    Value &getOrInsert(K key, V value) {
        if (this->root == nullptr) {
            this->root = Node::from(key, value);
            this->count += 1;
            return root->value;
        } else {
            NodePtr node = getNodeOrProvide(this->root, key,
                                            [&]() { return Node::from(key, value); });
            return node->value;
        }
    }

    Value operator[](K key) const { return this->get(key); }

    Value &operator[](K key) { return this->getOrDefault(key); }

    /**
     * Removes the mapping for a key from this map if it is present;
     * Returns true if the mapping is present else returns false
     * @param key the key of the mapping
     * @return bool
     */
    bool remove(K key) {
        if (this->root == nullptr) {
            return false;
        } else {
            return this->remove(this->root, key, [](ConstNodePtr) {});
        }
    }

    /**
     * Removes the mapping for a key from this map if it is present and returns
     * the value which is mapped to the key; If this map contains no mapping for
     * the key, a {@code NoSuchMappingException} will be thrown.
     * @param key
     * @return AvlTreeMap<Key, Value>::Value
     * @throws NoSuchMappingException
     */
    Value getAndRemove(K key) {
```

```
    Value result;
    NodeConsumer action = [&](ConstNodePtr node) { result = node->value; };

    if (root == nullptr) {
      throw NoSuchMappingException("Invalid key");
    } else {
      if (remove(this->root, key, action)) {
        return result;
      } else {
        throw NoSuchMappingException("Invalid key");
      }
    }
  }

  /**
   * Gets the entry corresponding to the specified key; if no such entry
   * exists, returns the entry for the least key greater than the specified
   * key; if no such entry exists (i.e., the greatest key in the Tree is less
   * than the specified key), a {@code NoSuchMappingException} will be thrown.
   * @param key
   * @return AvlTreeMap<Key, Value>::Entry
   * @throws NoSuchMappingException
   */
  Entry getCeilingEntry(K key) const {
    if (this->root == nullptr) {
      throw NoSuchMappingException("No ceiling entry in this map");
    }

    NodePtr node = this->root;
    std::stack<NodePtr> ancestors;

    while (node != nullptr) {
      if (key == node->key) {
        return node->entry();
      }

      if (compare(key, node->key)) {
        /* key < node->key */
        if (node->left != nullptr) {
          ancestors.push(node);
          node = node->left;
        } else {
          return node->entry();
        }
      } else {
        /* key > node->key */
        if (node->right != nullptr) {
          ancestors.push(node);
          node = node->right;
        } else {
          if (ancestors.empty()) {
            throw NoSuchMappingException("No ceiling entry in this map");
          }
```

```
          NodePtr parent = ancestors.top();
          ancestors.pop();

          while (node == parent->right) {
            node = parent;
            if (!ancestors.empty()) {
              parent = ancestors.top();
              ancestors.pop();
            } else {
              throw NoSuchMappingException("No ceiling entry in this map");
            }
          }

          return parent->entry();
        }
      }
    }

    throw NoSuchMappingException("No ceiling entry in this map");
  }

  /**
   * Gets the entry corresponding to the specified key; if no such entry exists,
   * returns the entry for the greatest key less than the specified key;
   * if no such entry exists, a {@code NoSuchMappingException} will be thrown.
   * @param key
   * @return AvlTreeMap<Key, Value>::Entry
   * @throws NoSuchMappingException
   */
  Entry getFloorEntry(K key) const {
    if (this->root == nullptr) {
      throw NoSuchMappingException("No floor entry exists in this map");
    }

    NodePtr node = this->root;
    std::stack<NodePtr> ancestors;

    while (node != nullptr) {
      if (key == node->key) {
        return node->entry();
      }

      if (compare(key, node->key)) {
        /* key < node->key */
        if (node->left != nullptr) {
          ancestors.push(node);
          node = node->left;
        } else {
          if (ancestors.empty()) {
            throw NoSuchMappingException("No floor entry exists in this map");
          }

          NodePtr parent = ancestors.top();
          ancestors.pop();
```

```
        while (node == parent->left) {
          node = parent;
          if (!ancestors.empty()) {
            parent = ancestors.top();
            ancestors.pop();
          } else {
            throw NoSuchMappingException("No floor entry exists in this map");
          }
        }

        return parent->entry();
      }
    } else {
      /* key > node->key */
      if (node->right != nullptr) {
        ancestors.push(node);
        node = node->right;
      } else {
        return node->entry();
      }
    }
  }

  throw NoSuchMappingException("No floor entry exists in this map");
}

/**
 * Gets the entry for the least key greater than the specified
 * key; if no such entry exists, returns the entry for the least
 * key greater than the specified key; if no such entry exists,
 * a {@code NoSuchMappingException} will be thrown.
 * @param key
 * @return AvlTreeMap<Key, Value>::Entry
 * @throws NoSuchMappingException
 */
Entry getHigherEntry(K key) {
  if (this->root == nullptr) {
    throw NoSuchMappingException("No higher entry exists in this map");
  }

  NodePtr node = this->root;
  std::stack<NodePtr> ancestors;

  while (node != nullptr) {
    if (compare(key, node->key)) {
      /* key < node->key */
      if (node->left != nullptr) {
        ancestors.push(node);
        node = node->left;
      } else {
        return node->entry();
      }
    } else {
```

```
      /* key >= node->key */
      if (node->right != nullptr) {
        ancestors.push(node);
        node = node->right;
      } else {
        if (ancestors.empty()) {
          throw NoSuchMappingException("No higher entry exists in this map");
        }

        NodePtr parent = ancestors.top();
        ancestors.pop();

        while (node == parent->right) {
          node = parent;
          if (!ancestors.empty()) {
            parent = ancestors.top();
            ancestors.pop();
          } else {
            throw NoSuchMappingException(
                "No higher entry exists in this map");
          }
        }

        return parent->entry();
      }
    }
  }

  throw NoSuchMappingException("No higher entry exists in this map");
}

/**
 * Returns the entry for the greatest key less than the specified key; if
 * no such entry exists (i.e., the least key in the Tree is greater than
 * the specified key), a {@code NoSuchMappingException} will be thrown.
 * @param key
 * @return AvlTreeMap<Key, Value>::Entry
 * @throws NoSuchMappingException
 */
Entry getLowerEntry(K key) const {
  if (this->root == nullptr) {
    throw NoSuchMappingException("No lower entry exists in this map");
  }

  NodePtr node = this->root;
  std::stack<NodePtr> ancestors;

  while (node != nullptr) {
    if (compare(key, node->key) || key == node->key) {
      /* key <= node->key */
      if (node->left != nullptr) {
        ancestors.push(node);
        node = node->left;
      } else {
```

```
        if (ancestors.empty()) {
          throw NoSuchMappingException("No lower entry exists in this map");
        }

        NodePtr parent = ancestors.top();
        ancestors.pop();

        while (node == parent->left) {
          node = parent;
          if (!ancestors.empty()) {
            parent = ancestors.top();
            ancestors.pop();
          } else {
            throw NoSuchMappingException("No lower entry exists in this map");
          }
        }

        return parent->entry();
      }
    } else {
      /* key > node->key */
      if (node->right != nullptr) {
        ancestors.push(node);
        node = node->right;
      } else {
        return node->entry();
      }
    }
  }

  throw NoSuchMappingException("No lower entry exists in this map");
}

/**
 * Remove all entries that satisfy the filter condition.
 * @param filter
 */
void removeAll(KeyValueFilter filter) {
  std::vector<Key> keys;
  this->inorderTraversal([&](ConstNodePtr node) {
    if (filter(node->key, node->value)) {
      keys.push_back(node->key);
    }
  });
  for (const Key &key : keys) {
    this->remove(key);
  }
}

/**
 * Performs the given action for each key and value entry in this map.
 * The value is immutable for the action.
 * @param action
 */
```

```
  void forEach(KeyValueConsumer action) const {
    this->inorderTraversal(
        [&](ConstNodePtr node) { action(node->key, node->value); });
  }

  /**
   * Performs the given action for each key and value entry in this map.
   * The value is mutable for the action.
   * @param action
   */
  void forEachMut(MutKeyValueConsumer action) {
    this->inorderTraversal(
        [&](ConstNodePtr node) { action(node->key, node->value); });
  }

  /**
   * Returns a list containing all of the entries in this map.
   * @return AvlTreeMap<Key, Value>::EntryList
   */
  EntryList toEntryList() const {
    EntryList entryList;
    this->inorderTraversal(
        [&](ConstNodePtr node) { entryList.push_back(node->entry()); });
    return entryList;
  }

 private:
  static NodePtr rotateLeft(ConstNodePtr node) {
    // clang-format off
    //     |                       |
    //     N                       S
    //    / \     l-rotate(N)     / \
    //   L   S    ==========>    N   R
    //      / \                 / \
    //     M   R               L   M
    NodePtr successor = node->right;
    // clang-format on
    node->right = successor->left;
    successor->left = node;

    node->updateHeight();
    successor->updateHeight();

    return successor;
  }

  static NodePtr rotateRight(ConstNodePtr node) {
    // clang-format off
    //       |                   |
    //       N                   S
    //      / \   r-rotate(N)   / \
    //     S   R  ==========>  L   N
    //    / \                     / \
    //   L   M                   M   R
```

```
    NodePtr successor = node->left;
    // clang-format on
    node->left = successor->right;
    successor->right = node;

    node->updateHeight();
    successor->updateHeight();

    return successor;
  }

  static void swapNode(NodePtr &lhs, NodePtr &rhs) {
    std::swap(lhs->key, rhs->key);
    std::swap(lhs->value, rhs->value);
    std::swap(lhs, rhs);
  }

  static void fixBalance(NodePtr &node) {
    if (node->factor() < -1) {
      if (node->left->factor() < 0) {
        // clang-format off
        //  Left-Left Case
        //       |
        //       C                 |
        //      /   r-rotate(C)    B
        //     B    ==========>   / \
        //    /                  A   C
        //   A
        // clang-format on
        node = rotateRight(node);
      } else {
        // clang-format off
        //  Left-Right Case
        //     |                   |
        //     C                   C                 |
        //    /   l-rotate(A)     /   r-rotate(C)    B
        //   A    ==========>    B    ==========>   / \
        //    \                 /                  A   C
        //     B               A
        // clang-format on
        node->left = rotateLeft(node->left);
        node = rotateRight(node);
      }
    } else if (node->factor() > 1) {
      if (node->right->factor() > 0) {
        // clang-format off
        //  Right-Right Case
        //   |
        //   C                     |
        //    \     l-rotate(C)    B
        //     B    ==========>   / \
        //      \                A   C
        //       A
        // clang-format on
```

```
        node = rotateLeft(node);
      } else {
        // clang-format off
        //  Right-Left Case
        //   |                 |
        //   A                 A                     |
        //    \   r-rotate(C)   \     l-rotate(A)    B
        //     C  ==========>    B    ==========>   / \
        //    /                   \                A   C
        //   B                     C
        // clang-format on
        node->right = rotateRight(node->right);
        node = rotateLeft(node);
      }
    }
  }

  NodePtr getNodeOrProvide(NodePtr &node, K key, NodeProvider provide) {
    assert(node != nullptr);

    if (key == node->key) {
      return node;
    }

    assert(key != node->key);

    NodePtr result;

    if (compare(key, node->key)) {
      /* key < node->key */
      if (node->left == nullptr) {
        result = node->left = provide();
        this->count += 1;
        node->updateHeight();
      } else {
        result = getNodeOrProvide(node->left, key, provide);
        node->updateHeight();
        fixBalance(node);
      }
    } else {
      /* key > node->key */
      if (node->right == nullptr) {
        result = node->right = provide();
        this->count += 1;
        node->updateHeight();
      } else {
        result = getNodeOrProvide(node->right, key, provide);
        node->updateHeight();
        fixBalance(node);
      }
    }

    return result;
  }
```

```
NodePtr getNode(ConstNodePtr node, K key) const {
  assert(node != nullptr);

  if (key == node->key) {
    return node;
  }

  if (compare(key, node->key)) {
    /* key < node->key */
    return node->left == nullptr ? nullptr : getNode(node->left, key);
  } else {
    /* key > node->key */
    return node->right == nullptr ? nullptr : getNode(node->right, key);
  }
}

void insert(NodePtr &node, K key, V value, bool replace = true) {
  assert(node != nullptr);

  if (key == node->key) {
    if (replace) {
      node->value = value;
    }
    return;
  }

  assert(key != node->key);

  if (compare(key, node->key)) {
    /* key < node->key */
    if (node->left == nullptr) {
      node->left = Node::from(key, value);
      this->count += 1;
      node->updateHeight();
    } else {
      insert(node->left, key, value, replace);
      node->updateHeight();
      fixBalance(node);
    }
  } else {
    /* key > node->key */
    if (node->right == nullptr) {
      node->right = Node::from(key, value);
      this->count += 1;
      node->updateHeight();
    } else {
      insert(node->right, key, value, replace);
      node->updateHeight();
      fixBalance(node);
    }
  }
}
```

```
  bool remove(NodePtr &node, K key, NodeConsumer action) {
    assert(node != nullptr);

    if (key != node->key) {
      if (compare(key, node->key)) {
        /* key < node->key */
        NodePtr &left = node->left;
        if (left != nullptr && remove(left, key, action)) {
          node->updateHeight();
          fixBalance(node);
          return true;
        } else {
          return false;
        }
      } else {
        /* key > node->key */
        NodePtr &right = node->right;
        if (right != nullptr && remove(right, key, action)) {
          node->updateHeight();
          fixBalance(node);
          return true;
        } else {
          return false;
        }
      }
    }

    assert(key == node->key);
    action(node);

    if (node->isLeaf()) {
      // Case 1: no child
      node = nullptr;
    } else if (node->right == nullptr) {
      // clang-format off
      // Case 2: left child only
      //     P
      //     |  remove(N)  P
      //     N  ========>  |
      //    /              L
      //   L
      // clang-format on
      node = node->left;
      node->updateHeight();
    } else if (node->left == nullptr) {
      // clang-format off
      // Case 3: right child only
      //   P
      //   |    remove(N)  P
      //   N    ========>  |
      //    \              R
      //     R
      // clang-format on
      node = node->right;
```

```
      node->updateHeight();
    } else if (node->right->left == nullptr) {
      // clang-format off
      // Case 4: both left and right child, right child has no left child
      //    |                 |
      //    N    remove(N)    R
      //   / \   ========>   /
      //  L   R             L
      // clang-format on
      NodePtr right = node->right;
      swapNode(node, right);
      right->right = node->right;
      node = right;
      node->updateHeight();
      fixBalance(node);
    } else {
      // clang-format off
      // Case 5: both left and right child, right child is not a leaf
      //   Step 1. find the node N with the smallest key
      //           and its parent P on the right subtree
      //   Step 2. swap S and N
      //   Step 3. remove node N like Case 1 or Case 3
      //   Step 4. update height for P
      //     |                  |
      //     N                  S                 |
      //    / \                / \                S
      //   L   ..  swap(N, S)  L   ..  remove(N)   / \
      //       |  ==========      |  =========  L  ..
      //       P                  P                 |
      //      / \                / \                P
      //     S  ..              N  ..              / \
      //      \                  \                R  ..
      //       R                  R
      // clang-format on

      // Step 1
      NodePtr successor = node->right;
      NodePtr parent = node;
      while (successor->left != nullptr) {
        parent = successor;
        successor = parent->left;
      }
      // Step 2
      swapNode(node, successor);
      // Step 3
      parent->left = node->right;
      // Restore node
      node = successor;
      // Step 4
      parent->updateHeight();
    }

    this->count -= 1;
    return true;
```

```
  }

  void inorderTraversal(NodeConsumer action) const {
    if (this->root == nullptr) {
      return;
    }

    std::stack<NodePtr> stack;
    NodePtr node = this->root;

    while (node != nullptr || !stack.empty()) {
      while (node != nullptr) {
        stack.push(node);
        node = node->left;
      }
      if (!stack.empty()) {
        node = stack.top();
        stack.pop();
        action(node);
        node = node->right;
      }
    }
  }
};

#endif  // AVLTREE_MAP_HPP
```

其他资料

在 AVL Tree Visualization[1] 可以观察 AVL 树维护平衡的过程。

维基百科 -- AVL 树[2]

参考资料与注释

[1] AVL Tree Visualization

[2] 维基百科 – AVL 树

10.17.7 B 树

Authors: Persdre

引入

在计算机科学中，B 树（B-tree）是一种自平衡的树，能够保持数据有序。这种数据结构能够让查找数据、顺序访问、插入数据及删除的动作，都在对数时间内完成。

在 B 树中，有两种节点：

1. 内部节点（internal node）：存储了数据以及指向其子节点的指针。
2. 叶子节点（leaf node）：与内部节点不同的是，叶子节点只存储数据，并没有子节点。

树是一种数据结构。树用多个节点储存元素。某些节点存在一定的关系，用连线表示。二叉树是一种特殊的树，每个节点最多有两个子树。二叉树常用于实现二叉搜索树和二叉堆。而 AVL 树 是特殊的二叉树，是最早被发明的自平衡二叉查找树。B 树保留了自平衡的特点，但 B 树的每个节点可以拥有两个以上的子节点，因此 B 树是一种多路搜索树。

性质

首先介绍一下一棵 m 阶的 B 树的特性。m 表示这个树的每一个节点最多可以拥有的子节点个数。一棵 m 阶的 B 树满足的性质如下：

1. 每个节点最多有 m 个子节点。
2. 每一个非叶子节点（除根节点）最少有 $\lceil \frac{m}{2} \rceil$ 个子节点。
3. 如果根节点不是叶子节点，那么它至少有两个子节点。
4. 有 k 个子节点的非叶子节点拥有 $k-1$ 个键，且升序排列，满足 $k[i] < k[i+1]$。
5. 每个节点至多包含 $2k-1$ 个键。
6. 所有的叶子节点都在同一层。

一个简单的图例如下：

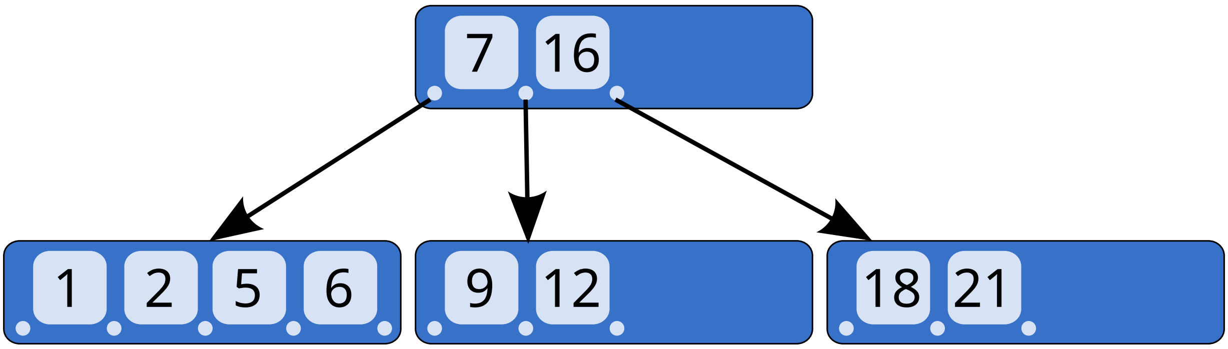


图 10.79

过程

与 二叉搜索树 类似，B 树的基本操作有查找，遍历，插入，删除。

查找

B 树中的节点包含有多个键。假设需要查找的是 k，那么从根节点开始，从上到下递归的遍历树。在每一层上，搜索的范围被减小到包含了搜索值的子树中。子树值的范围被它的父节点的键确定。因为是从根节点开始的二分法查找，所以查找一个键的代码如下：

" 实现"

```
BTreeNode *BTreeNode::search(int k) {
  // 找到第一个大于等于待查找键 k 的键
  int i = 0;
  while (i < n && k > keys[i]) i++;

  // 如果找到的第一个键等于 k ，返回节点指针
  if (keys[i] == k) return this;

  // 如果没有找到键 k 且当前节点为叶子节点则返回 NULL
```

```
    if (leaf == true) return NULL;

    // 递归
    return C[i]->search(k);
  }
```

遍历

B 树的中序遍历与二叉搜索树的中序遍历也很相似，从最左边的孩子节点开始，递归地打印最左边的孩子节点，然后对剩余的孩子和键重复相同的过程。最后，递归打印最右边的孩子节点。

遍历的代码如下：

" 实现"

```
void BTreeNode::traverse() {
  // 有 n 个键和 n+1 个孩子
  // 遍历 n 个键和前 n 个孩子
  int i;
  for (i = 0; i < n; i++) {
    // 如果当前节点不是叶子节点，在打印 key[i] 之前，
    // 先遍历以 C[i] 为根的子树.
    if (leaf == false) C[i]->traverse();
    cout << " " << keys[i];
  }

  // 打印以最后一个孩子为根的子树
  if (leaf == false) C[i]->traverse();
}
```

插入

为了方便表述，插入设定为在以 o 为根节点的 B 树中插入一个值为 v 的新节点。

一个新插入的 v 总是被插入到叶子节点。与二叉搜索树的插入操作类似，从根节点开始，向下遍历直到叶子节点，将值为 v 的新节点插入到相应的叶子节点。与二叉搜索树不同的是，通过最小度定义了一个节点可以包含键的个数的一个取值范围，所以在插入一个新节点时，就需要确认插入这个叶子节点之后，它的父节点是否超出该节点本身最大可容纳的节点个数。

针对一棵高度为 h 的 m 阶 B 树，插入一个元素时，首先要验证该元素在 B 树中是否存在，如果不存在，那么就要在叶子节点中插入该新的元素，此时分 3 种情况：

1. 如果叶子节点空间足够，即该节点的关键字数小于 $m-1$，则直接插入在叶子节点的左边或右边；
2. 如果空间满了以至于没有足够的空间去添加新的元素，即该节点的关键字数已经有了 m 个，则需要将该节点进行「分裂」，将一半数量的关键字元素分裂到新的其相邻右节点中，中间关键字元素上移到父节点中，而且当节点中关键元素向右移动了，相关的指针也需要向右移。
   (a) 从该节点的原有元素和新的元素中选择出中位数
   (b) 小于这一中位数的元素放入左边节点，大于这一中位数的元素放入右边节点，中位数作为分隔值。
   (c) 分隔值被插入到父节点中，这可能会造成父节点分裂，分裂父节点时可能又会使它的父节点分裂，以此类推。如果没有父节点（这一节点是根节点），就创建一个新的根节点（增加了树的高度）。

如果一直分裂到根节点，那么就需要创建一个新的根节点。它有一个分隔值和两个子节点。

这就是根节点并不像内部节点一样有最少子节点数量限制的原因。每个节点中元素的最大数量是 $U-1$。当一个节点分裂时，一个元素被移动到它的父节点，但是一个新的元素增加了进来。所以最大的元素数量 $U-1$ 必须能够被分成两个合法的节点。如果 $U-1$ 是奇数，那么 $U=2L$，总共有 $2L-1$ 个元素，一个新的节点有 $L-1$ 个元素，另外一个有 L 个元素，都是合法的节点。如果 $U-1$ 是偶数，那么 $U=2L-1$，总共有 $2L-2$ 个元素。一半是 $L-1$，

正好是节点允许的最小元素数量。

插入的代码如下：

" 实现"

```
void BTree::insert(int k) {
  // 如果树为空树
  if (root == NULL) {
    // 为根节点分配空间
    root = new BTreeNode(t, true);
    root->keys[0] = k;  // 插入节点 k
    root->n = 1;        // 更新根节点的关键字的个数为 1
  } else {
    // 当根节点已满，则对 B-树进行生长操作
    if (root->n == 2 * t - 1) {
      // 为新的根节点分配空间
      BTreeNode *s = new BTreeNode(t, false);

      // 将旧的根节点作为新的根节点的孩子
      s->C[0] = root;

      // 将旧的根节点分裂为两个，并将一个关键字上移到新的根节点
      s->splitChild(0, root);

      // 新的根节点有两个孩子节点
      // 确定哪一个孩子将拥有新插入的关键字
      int i = 0;
      if (s->keys[0] < k) i++;
      s->C[i]->insertNonFull(k);

      // 新的根节点更新为 s
      root = s;
    } else  // 根节点未满，调用 insertNonFull() 函数进行插入
      root->insertNonFull(k);
  }
}

// 将关键字 k 插入到一个未满的节点中
void BTreeNode::insertNonFull(int k) {
  // 初始化 i 为节点中的最后一个关键字的位置
  int i = n - 1;

  // 如果当前节点是叶子节点
  if (leaf == true) {
    // 下面的循环做两件事：
    // a) 找到新插入的关键字位置并插入
    // b) 移动所有大于关键字 k 的向后移动一个位置
    while (i >= 0 && keys[i] > k) {
      keys[i + 1] = keys[i];
      i--;
    }

    // 插入新的关键字，节点包含的关键字个数加 1
    keys[i + 1] = k;
```

```
    n = n + 1;
  } else {
    // 找到第一个大于关键字 k 的关键字 keys[i] 的孩子节点
    while (i >= 0 && keys[i] > k) i--;

    // 检查孩子节点是否已满
    if (C[i + 1]->n == 2 * t - 1) {
      // 如果已满，则进行分裂操作
      splitChild(i + 1, C[i + 1]);

      // 分裂后，C[i] 中间的关键字上移到父节点，
      // C[i] 分裂称为两个孩子节点
      // 找到新插入关键字应该插入的节点位置
      if (keys[i + 1] < k) i++;
    }
    C[i + 1]->insertNonFull(k);
  }
}

// 节点 y 已满，则分裂节点 y
void BTreeNode::splitChild(int i, BTreeNode *y) {
  // 创建一个新的节点存储 t - 1 个关键字
  BTreeNode *z = new BTreeNode(y->t, y->leaf);
  z->n = t - 1;

  // 将节点 y 的后 t -1 个关键字拷贝到 z 中
  for (int j = 0; j < t - 1; j++) z->keys[j] = y->keys[j + t];

  // 如果 y 不是叶子节点，拷贝 y 的后 t 个孩子节点到 z 中
  if (y->leaf == false) {
    for (int j = 0; j < t; j++) z->C[j] = y->C[j + t];
  }

  // 将 y 所包含的关键字的个数设置为 t -1
  // 因为已满则为 2t -1 ，节点 z 中包含 t - 1 个
  // 一个关键字需要上移
  // 所以 y 中包含的关键字变为 2t-1 - (t-1) -1
  y->n = t - 1;

  // 给当前节点的指针分配新的空间，
  // 因为有新的关键字加入，父节点将多一个孩子。
  for (int j = n; j >= i + 1; j--) C[j + 1] = C[j];

  // 当前节点的下一个孩子设置为 z
  C[i + 1] = z;

  // 将所有父节点中比上移的关键字大的关键字后移
  // 找到上移节点的关键字的位置
  for (int j = n - 1; j >= i; j--) keys[j + 1] = keys[j];

  // 拷贝 y 的中间关键字到其父节点中
  keys[i] = y->keys[t - 1];

  // 当前节点包含的关键字个数加 1
```

```
  n = n + 1;
}
```

删除

B 树的删除操作相比于插入操作更为复杂，因为删除之后经常需要重新排列节点。

与 B 树的插入操作类似，必须确保删除操作不违背 B 树的特性。正如插入操作中每一个节点所包含的键的个数不能超过 $2k-1$ 一样，删除操作要保证每一个节点包含的键的个数不少于 $k-1$ 个（除根节点允许包含比 $k-1$ 少的关键字的个数）。

有两种常用的删除策略：

1. 定位并删除元素，然后调整树使它满足约束条件。
2. 从上到下处理这棵树，在进入一个节点之前，调整树使得之后一旦遇到了要删除的键，它可以被直接删除而不需要再进行调整。

下面介绍使用第一种策略的删除。

首先，查找 B 树中需删除的元素，如果该元素在 B 树中存在，则将该元素在其节点中进行删除；删除该元素后，首先判断该元素是否有左右孩子节点，如果有，则上移孩子节点中的某相近元素（「左孩子最右边的节点」或「右孩子最左边的节点」）到父节点中，然后是移动之后的情况；如果没有，直接删除。

1. 某节点中元素数目小于 $m/2-1$，$m/2$ 向上取整，则需要看其某相邻兄弟节点是否丰满。
2. 如果丰满（节点中元素个数大于 $m/2-1$），则向父节点借一个元素来满足条件。
3. 如果其相邻兄弟都不丰满，即其节点数目等于 $m/2-1$，则该节点与其相邻的某一兄弟节点进行「合并」成一个节点。

接下来用一个 5 阶 B 树为例，详细讲解删除的操作。

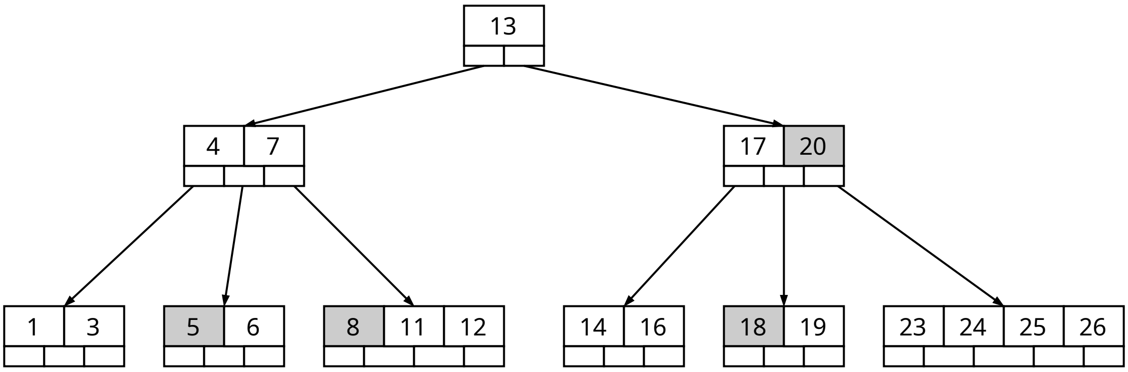


图 10.80

如图所示，接下来要依次删除 8，20，18，5。首先要删除元素 8。先查找到元素 8 在叶子节点中，删除 8 后叶子节点的元素个数为 2，符合 B 树的规则。然后需要把元素 11 和 12 都向前移动一位。完成后如图所示。

下一步，删除 20，因为 20 没有在叶子节点中，而是在中间节点中找到，可以发现 20 的继承者是 23（字母升序的下个元素），然后需要将 23 上移到 20 的位置，之后将孩子节点中的 23 进行删除。删除后检查一下，该孩子节点中元素个数大于 2，无需进行合并操作。

所以这一步之后，B 树如下图所示。

下一步删除 18，18 在叶子节点中，但是该节点中元素数目为 2，删除导致只有 1 个元素，已经小于最小元素数目 2。而由前面已经知道：如果其某个相邻兄弟节点中比较丰满（元素个数大于 $\lceil\frac{5}{2}\rceil$），则可以向父节点借一个元素，然后将最丰满的相邻兄弟节点中上移最后或最前一个元素到父节点中。在这个实例中，右相邻兄弟节点中比较丰满（3 个元素大于 2），所以先向父节点借一个元素 23 下移到该叶子节点中，代替原来 19 的位置。19 前移。然后 24 在相邻右兄弟节点中，需要上移到父节点中。最后在相邻右兄弟节点中删除 24，后面的元素前移。

这一步之后，B 树如下图所示。

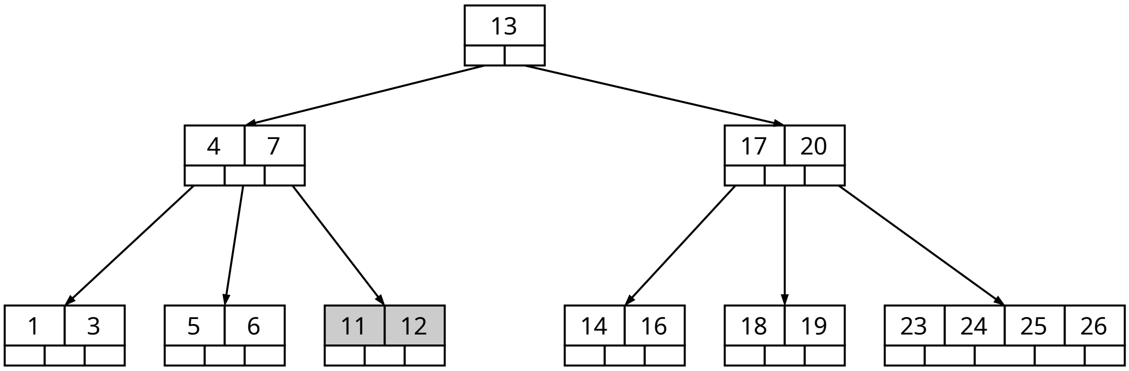


图 10.81

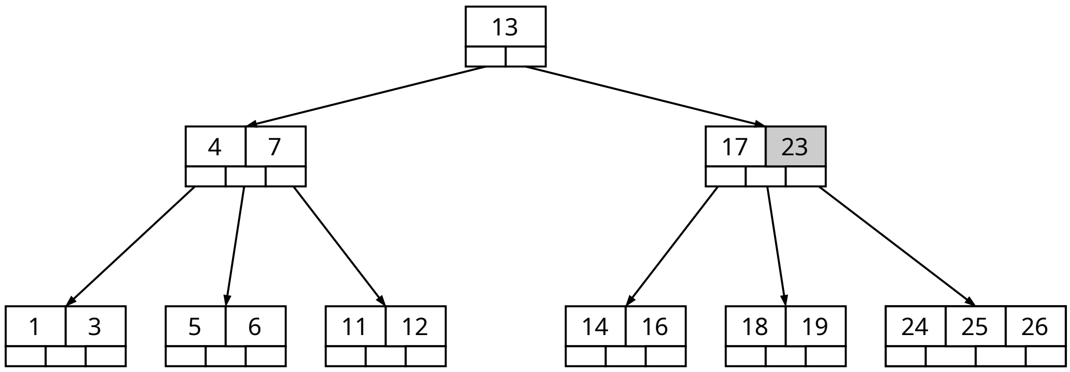


图 10.82

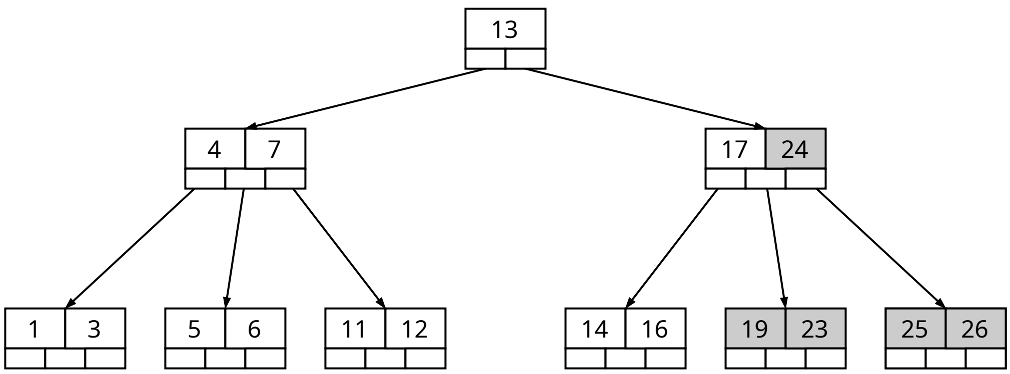


图 10.83

最后一步需要删除元素 5，但是删除后会导致很多问题。因为 5 所在的节点数目刚好达标也就是刚好满足最小元素个数 2。而相邻的兄弟节点也是同样的情况，删除一个元素都不能满足条件，所以需要该节点与某相邻兄弟节点进行合并操作；首先移动父节点中的元素（该元素在两个需要合并的两个节点元素之间）下移到其子节点中。然后将这两个节点进行合并成一个节点。所以在该实例中，首先将父节点中的元素 4 下移到已经删除 5 而只有 6 的节点中，然后将含有 4 和 6 的节点和含有 1，3 的相邻兄弟节点进行合并成一个节点。

这一步之后，B 树如下图所示。

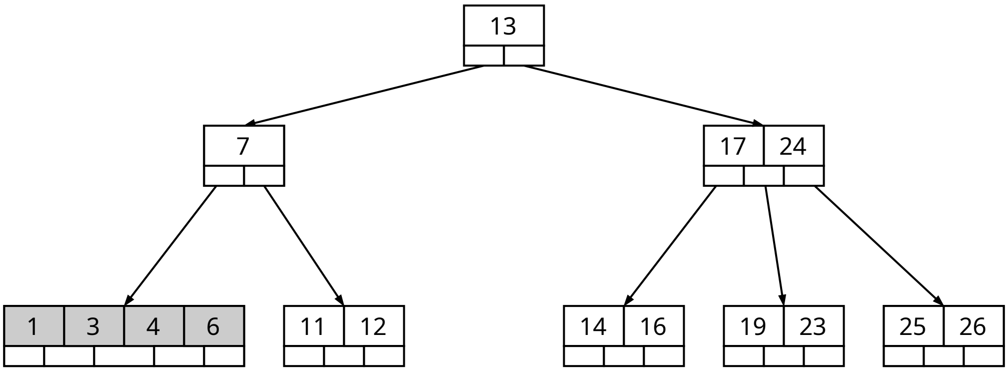


图 10.84

但是这里观察到父节点只包含了一个元素 7，这就没有达标（因为非根节点包括叶子节点的元素数量 K 必须满足于 $2 <= K <= 4$，而此处的 $K = 1$）。如果这个问题节点的相邻兄弟比较丰满，则可以向父节点借一个元素。而此时兄弟节点元素刚好为 2，刚刚满足，只能进行合并，而根节点中的唯一元素 13 下移到子节点。这样，树的高度减少一层。

所以最终的效果如下图。

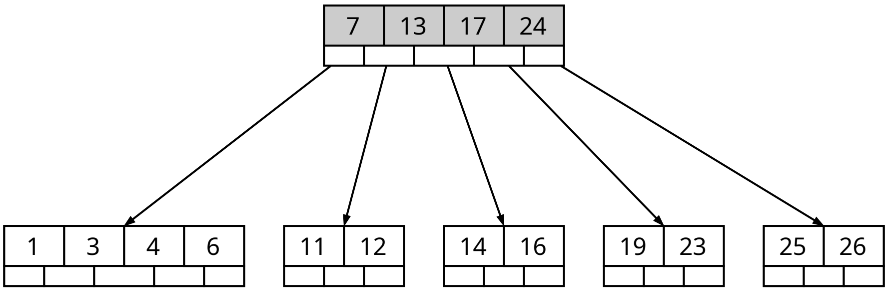


图 10.85

删除的伪代码如下：

" 实现"

```
B-Tree-Delete-Key(x, k)
if not leaf[x] then
    y ← Preceding-Child(x)
    z ← Successor-Child(x)
    if n[[y] > t - 1 then
        k' ← Find-Predecessor-Key(k, x)]()
        Move-Key(k', y, x)
        Move-Key(k, x, z)
        B-Tree-Delete-Key(k, z)
    else if n[z] > t - 1 then
        k' ← Find-Successor-Key(k, x)
        Move-Key(k', z, x)
```

```
        Move-Key(k, x, y)
        B-Tree-Delete-Key(k, y)
    else
        Move-Key(k, x, y)
        Merge-Nodes(y, z)
        B-Tree-Delete-Key(k, y)
    else (leaf node)
    y ← Preceding-Child(x)
    z ← Successor-Child(x)
    w ← root(x)
    v ← RootKey(x)
        if n[x] > t - 1 then Remove-Key(k, x)
        else if n[y] > t - 1 then
            k' ← Find-Predecessor-Key(w, v)
            Move-Key(k', y,w)
            k' ← Find-Successor-Key(w, v)
            Move-Key(k',w, x)
            B-Tree-Delete-Key(k, x)
        else if n[w] > t - 1 then
            k' ← Find-Successor-Key(w, v)
            Move-Key(k', z,w)
            k' ← Find-Predecessor-Key(w, v)
            Move-Key(k',w, x)
            B-Tree-Delete-Key(k, x)
        else
            s ← Find-Sibling(w)
            w' ← root(w)
                if n[w'] = t - 1 then
                    Merge-Nodes(w',w)
                    Merge-Nodes(w, s)
                    B-Tree-Delete-Key(k, x)
                else
                    Move-Key(v,w, x)
                    B-Tree-Delete-Key(k, x)
```

B 树优势

之前已经介绍过二叉查找树。但是这类型数据结构的问题在于，由于每个节点只能容纳一个数据，导致树的高度很高，逻辑上挨着的节点数据可能离得很远。

考虑在磁盘中存储数据的情况，与内存相比，读写磁盘有以下不同点：

1. 读写磁盘的速度相比内存读写慢很多。
2. 每次读写磁盘的单位要比读写内存的最小单位大很多。

由于读写磁盘的这个特点，因此对应的数据结构应该尽量的满足「局部性原理」：「当一个数据被用到时，其附近的数据也通常会马上被使用」，为了满足局部性原理，所以应该将逻辑上相邻的数据在物理上也尽量存储在一起。这样才能减少读写磁盘的数量。

所以，对比起一个节点只能存储一个数据的 BST 类数据结构来，要求这种数据结构在形状上更「胖」、更加「扁平」，即：每个节点能容纳更多的数据，这样就能降低树的高度，同时让逻辑上相邻的数据都能尽量存储在物理上也相邻的硬盘空间上，减少磁盘读写。

参考资料

- B 树 - 维基百科 ， 自由的百科全书[1]

- B 树详解[2]
- B 树、B+ 树索引算法原理（上）[3]
- B 树，B+ 树详解[4]

参考资料与注释

[1] B 树 - 维基百科，自由的百科全书

[2] B 树详解

[3] B 树、B + 树索引算法原理（上）

[4] B 树，B + 树详解

10.17.8 B+ 树

Authors: Persdre

引入

B+ 树是 B 树 的一个升级，它比 B 树更适合实际应用中操作系统的文件索引和数据库索引。目前现代关系型数据库最广泛的支持索引结构就是 B+ 树。

B+ 树是一种多叉排序树，即每个节点通常有多个孩子。一棵 B+ 树包含根节点、内部节点和叶子节点。根节点可能是一个叶子节点，也可能是一个包含两个或两个以上孩子节点的节点。

B+ 树的特点是能够保持数据稳定有序，其插入与修改拥有较稳定的对数时间复杂度。B+ 树元素自底向上插入，这与二叉树恰好相反。

首先介绍一棵 m 阶 B+ 树的特性。m 表示这个树的每一个节点最多可以拥有的子节点个数。一棵 m 阶的 B+ 树和 B 树的差异在于：

1. 有 n 棵子树的节点中含有 $n-1$ 个关键字（即将区间分为 n 个子区间，每个子区间对应一棵子树）。
2. 所有叶子节点中包含了全部关键字的信息，及指向含这些关键字记录的指针，且叶子节点本身依关键字的大小自小而大顺序链接。
3. 所有的非叶子节点可以看成是索引部分，节点中仅含有其子树（根节点）中的最大（或最小）关键字。
4. 除根节点外，其他所有节点中所含关键字的个数最少有 $\lceil\frac{m}{2}\rceil$（注意：B 树中除根以外的所有非叶子节点至少有 $\lceil\frac{m}{2}\rceil$ 棵子树）。

同时，B+ 树为了方便范围查询，叶子节点之间还用指针串联起来。

以下是一棵 B+ 树的典型结构：

B+ 树相比于 B 树的优势

由于索引节点上只有索引而没有数据，所以索引节点上能存储比 B 树更多的索引，这样树的高度就会更矮。树的高度越矮，磁盘寻道的次数就会越少。

因为数据都集中在叶子节点，而所有叶子节点的高度相同，那么可以在叶子节点中增加前后指针，指向同一个父节点的相邻兄弟节点，这样可以更好地支持查询一个值的前驱或后继，使连续访问更容易实现。

比如这样的 SQL 语句：`select * from tbl where t > 10`，如果使用 B+ 树存储数据的话，可以首先定位到数据为 10 的节点，再沿着它的 next 指针一路找到所有在该叶子节点右边的叶子节点，返回这些节点包含的数据。

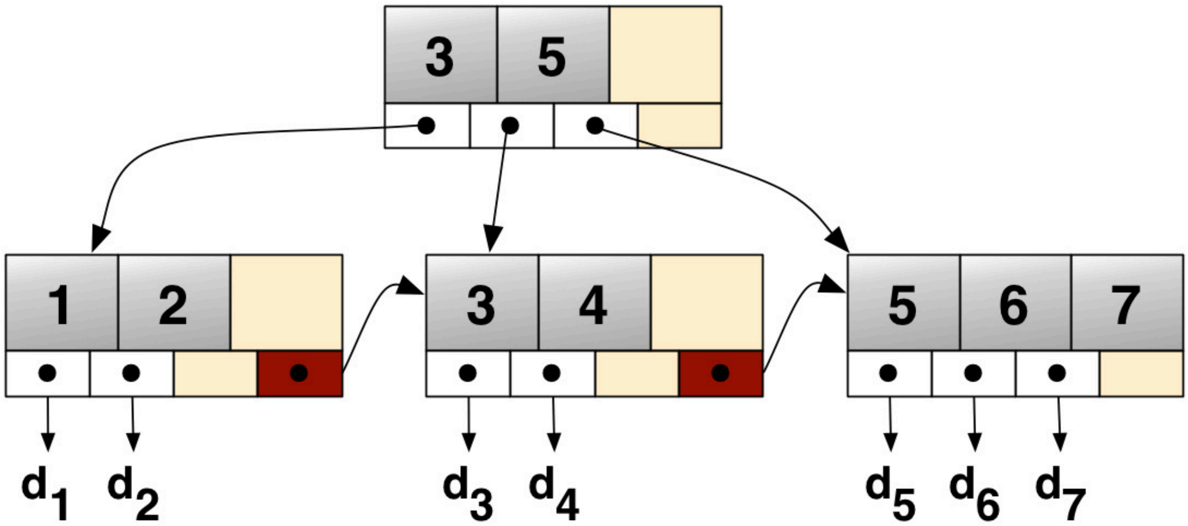


图 10.86

而如果使用 B 树结构，由于数据既可以存储在内部节点也可以存储在叶子节点，连续访问的实现会更加繁琐（需要在树的内部结构中进行移动）。

过程

与 B 树 类似，B+ 树的基本操作有查找，遍历，插入，删除。

查找

B+ 树的查找过程和 B 树类似。假设需要查找的键值是 k，那么从根节点开始，从上到下递归地遍历树。在每一层上，搜索的范围被减小到包含搜索值的子树中。

一个实例：在如下这棵 B+ 树上查找 45。

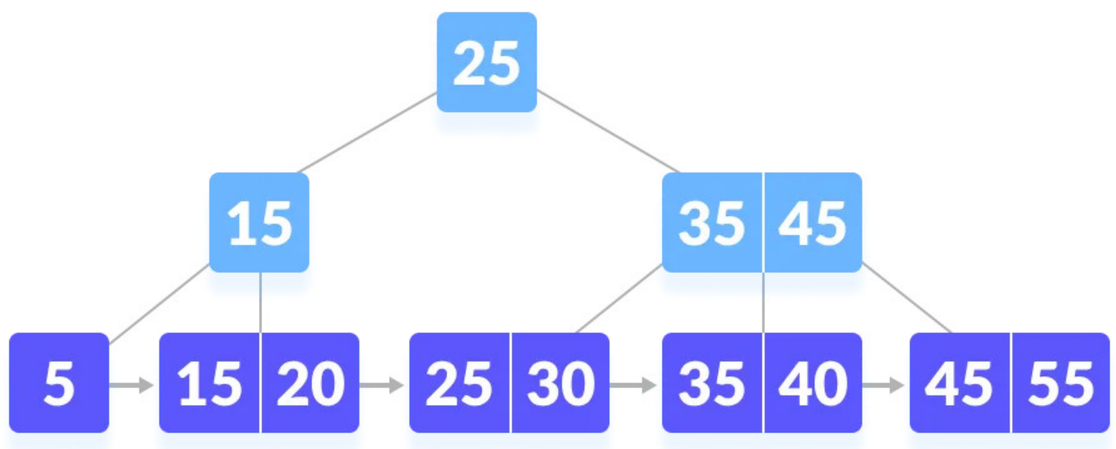


图 10.87

先和根节点比较

因为根节点的键值比 45 要小，所以去往根节点的右子树查找

因为 45 比 30 大，所以要与右边的索引相比

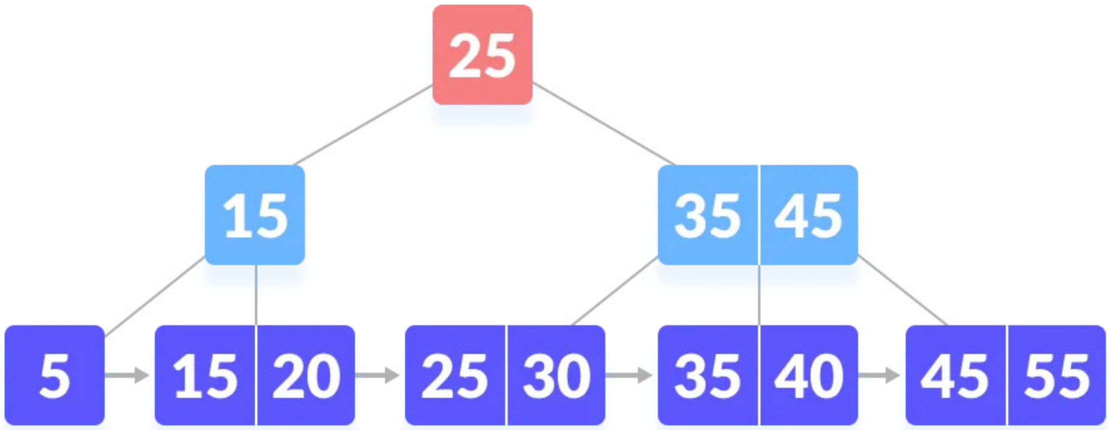


图 10.88

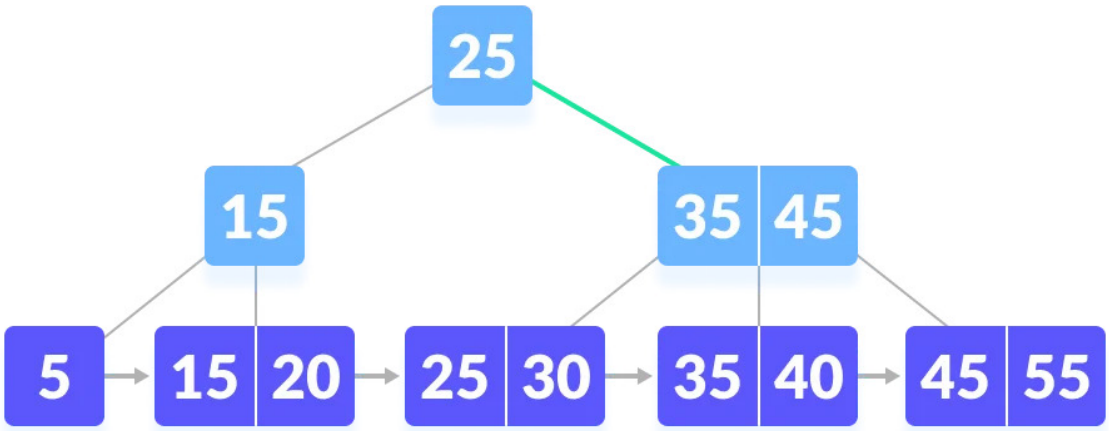


图 10.89

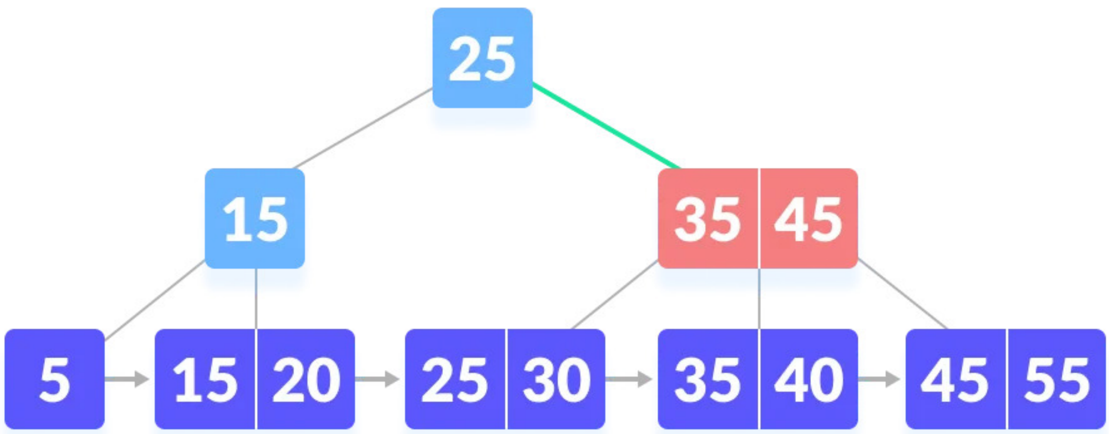


图 10.90

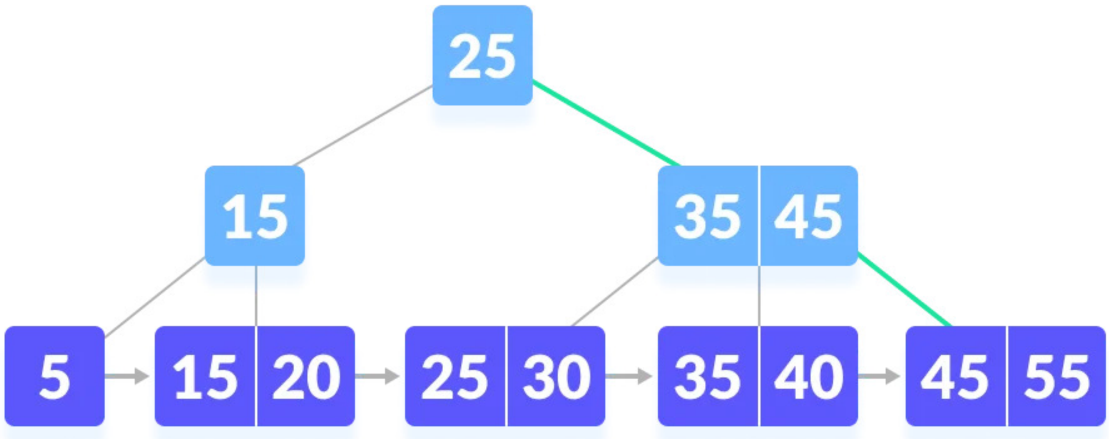


图 10.91

右侧的索引也为 45，所以要去往该节点的右子树继续查找

然后就可以找到 45

需要注意的是，在查找时，若非叶子节点上的关键字等于给定值，并不终止，而是继续向下直到叶子节点。因此，在 B+ 树中，不管查找成功与否，每次查找都是走了一条从根到叶子节点的路径。其余同 B 树的查找类似。

查找一个键的代码如下：

" 实现"

```
T find(V key) {
  int i = 0;
  while (i < this.number) {
    if (key.compareTo((V)this.keys[i]) <= 0) break;
    i++;
  }
  if (this.number == i) return null;
  return this.childs[i].find(key);
}
```

遍历

B+ 树只在叶子节点的层级上就可以实现整棵树的遍历。从根节点出发一路搜索到最左端的叶子节点之后即可根据指针遍历。

插入

B+ 树的插入算法与 B 树的相近：

1. 若为空树，创建一个叶子节点，然后将记录插入其中，此时这个叶子节点也是根节点，插入操作结束。
2. 针对叶子类型节点：根据关键字找到叶子节点，向这个叶子节点插入记录。插入后，若当前节点关键字的个数小于 m，则插入结束。否则将这个叶子节点分裂成左右两个叶子节点，左叶子节点包含前 $m/2$ 个记录，右节点包含剩下的记录，将第 $m/2+1$ 个记录的关键字进位到父节点中（父节点一定是索引类型节点），进位到父节点的关键字左孩子指针向左节点，右孩子指针向右节点。将当前节点的指针指向父节点，然后执行第 3 步。
3. 针对索引类型节点（内部节点）：若当前节点关键字的个数小于等于 $m-1$，则插入结束。否则，将这个索引类型节点分裂成两个索引节点，左索引节点包含前 $(m-1)/2$ 个 key，右节点包含 $m-(m-1)/2$ 个 key，将第 $m/2$ 个关键字进位到父节点中，进位到父节点的关键字左孩子指向左节点，进位到父节点的关键字右孩子指向

右节点。将当前节点的指针指向父节点，然后重复这一步。

比如在下图的 B+ 树中，插入新的数据 10：

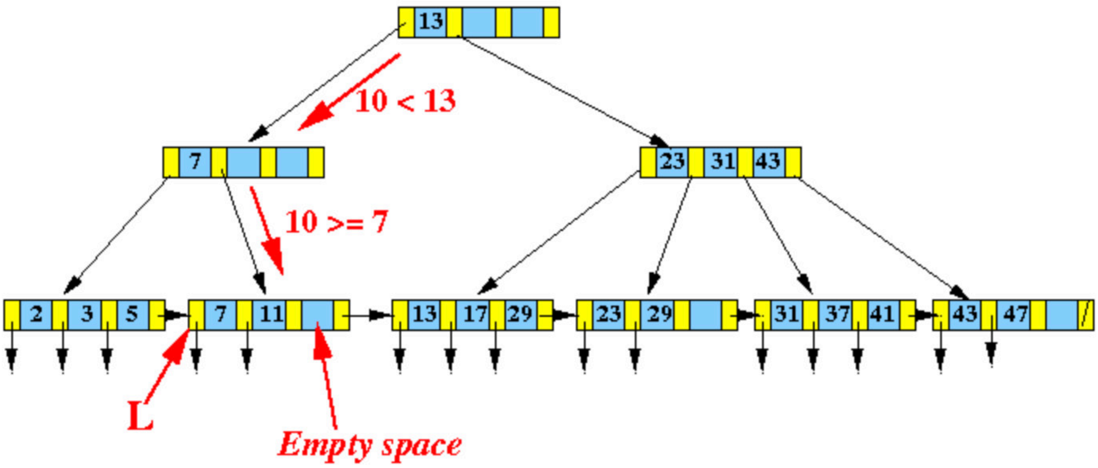


图 10.92

由于插入节点 $[7, 11]$ 在插入之后并没有溢出，所以可以直接变成 $[7, 10, 11]$。

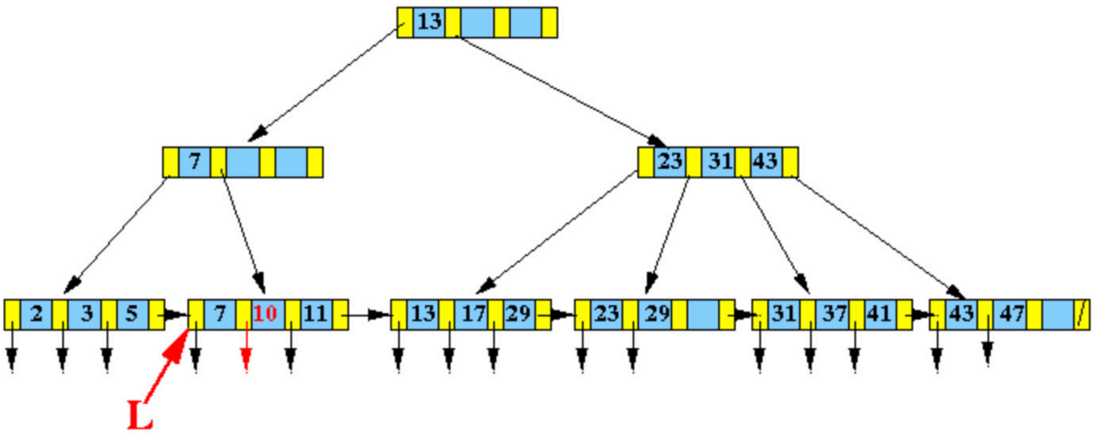


图 10.93

而如下图的 B+ 树中，插入数据 4：

由于所在节点 $[2, 3, 5]$ 在插入之后数据溢出，因此需要分裂为两个新的节点，同时调整父节点的索引数据：

$[2, 3, 4, 5]$ 分裂成了 $[2, 3]$ 和 $[4, 5]$，因此需要在这两个节点之间新增一个索引值，这个值应该满足：

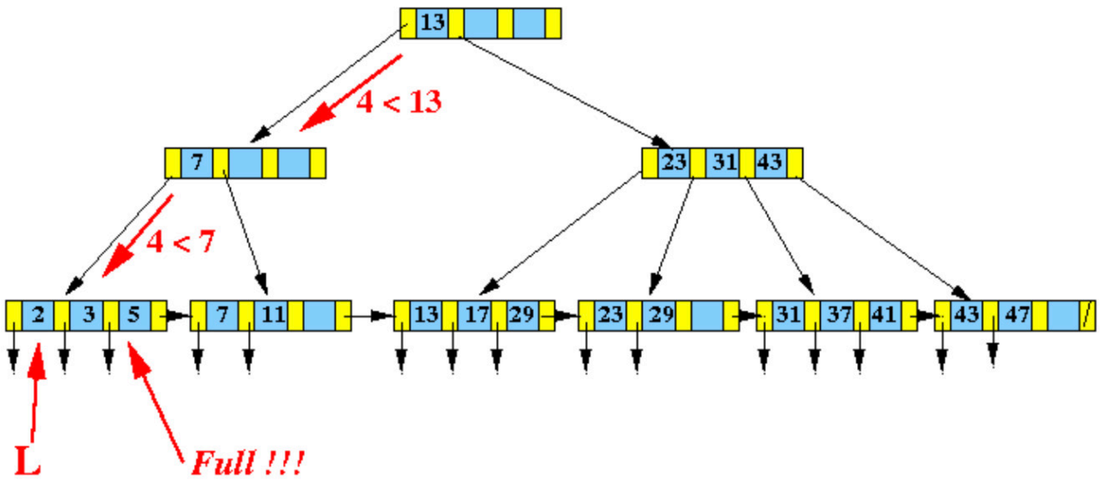
1. 大于左子树的最大值；
2. 小于等于右子树的最小值。

综上，需要在父节点中新增索引 4 和两个指向新节点的指针。

更多的例子可以参考演示网站 BPlustree[1]

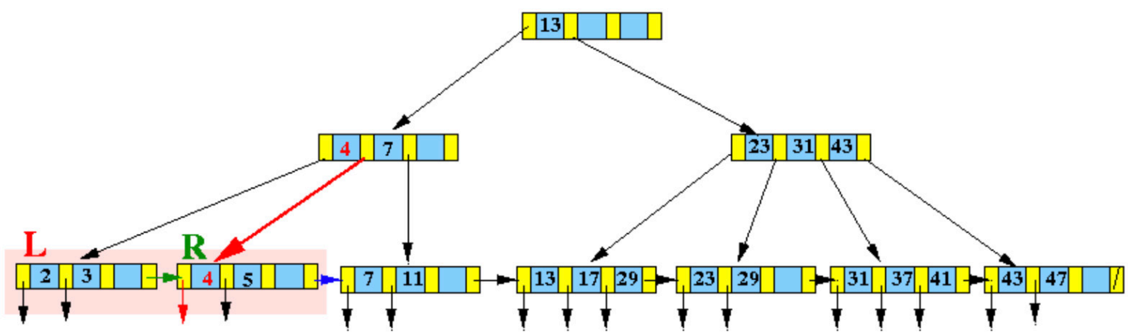
插入一个键的代码如下：

" 实现"



13
4 < 13
7
23 31 43
4 < 7
2 3 5
7 11
13 17 29
23 29
31 37 41
43 47
L
Full !!!

图 10.94



13
4 7
23 31 43
L
R
2 3
4 5
7 11
13 17 29
23 29
31 37 41
43 47

图 10.95

```
void BPTree::insert(int x) {
  if (root == NULL) {
    root = new Node;
    root->key[0] = x;
    root->IS_LEAF = true;
    root->size = 1;
    root->parent = NULL;
  } else {
    Node* cursor = root;
    Node* parent;

    while (cursor->IS_LEAF == false) {
      parent = cursor;
      for (int i = 0; i < cursor->size; i++) {
        if (x < cursor->key[i]) {
          cursor = cursor->ptr[i];
          break;
        }

        if (i == cursor->size - 1) {
          cursor = cursor->ptr[i + 1];
          break;
        }
      }
    }
    if (cursor->size < MAX) {
      insertVal(x, cursor);
      cursor->parent = parent;
      cursor->ptr[cursor->size] = cursor->ptr[cursor->size - 1];
      cursor->ptr[cursor->size - 1] = NULL;
    } else
      split(x, parent, cursor);
  }
}

void BPTree::split(int x, Node* parent, Node* cursor) {
  Node* LLeaf = new Node;
  Node* RLeaf = new Node;
  insertVal(x, cursor);
  LLeaf->IS_LEAF = RLeaf->IS_LEAF = true;
  LLeaf->size = (MAX + 1) / 2;
  RLeaf->size = (MAX + 1) - (MAX + 1) / 2;
  for (int i = 0; i < MAX + 1; i++) LLeaf->ptr[i] = cursor->ptr[i];
  LLeaf->ptr[LLeaf->size] = RLeaf;
  RLeaf->ptr[RLeaf->size] = LLeaf->ptr[MAX];
  LLeaf->ptr[MAX] = NULL;
  for (int i = 0; i < LLeaf->size; i++) {
    LLeaf->key[i] = cursor->key[i];
  }
  for (int i = 0, j = LLeaf->size; i < RLeaf->size; i++, j++) {
    RLeaf->key[i] = cursor->key[j];
  }
  if (cursor == root) {
```

```
    Node* newRoot = new Node;
    newRoot->key[0] = RLeaf->key[0];
    newRoot->ptr[0] = LLeaf;
    newRoot->ptr[1] = RLeaf;
    newRoot->IS_LEAF = false;
    newRoot->size = 1;
    root = newRoot;
    LLeaf->parent = RLeaf->parent = newRoot;
  } else {
    insertInternal(RLeaf->key[0], parent, LLeaf, RLeaf);
  }
}

void BPTree::insertInternal(int x, Node* cursor, Node* LLeaf, Node* RRLeaf) {
  if (cursor->size < MAX) {
    auto i = insertVal(x, cursor);
    for (int j = cursor->size; j > i + 1; j--) {
      cursor->ptr[j] = cursor->ptr[j - 1];
    }
    cursor->ptr[i] = LLeaf;
    cursor->ptr[i + 1] = RRLeaf;
  }

  else {
    Node* newLchild = new Node;
    Node* newRchild = new Node;
    Node* virtualPtr[MAX + 2];
    for (int i = 0; i < MAX + 1; i++) {
      virtualPtr[i] = cursor->ptr[i];
    }
    int i = insertVal(x, cursor);
    for (int j = MAX + 2; j > i + 1; j--) {
      virtualPtr[j] = virtualPtr[j - 1];
    }
    virtualPtr[i] = LLeaf;
    virtualPtr[i + 1] = RRLeaf;
    newLchild->IS_LEAF = newRchild->IS_LEAF = false;
    // 这里和叶子节点有区别
    newLchild->size = (MAX + 1) / 2;
    newRchild->size = MAX - (MAX + 1) / 2;
    for (int i = 0; i < newLchild->size; i++) {
      newLchild->key[i] = cursor->key[i];
    }
    for (int i = 0, j = newLchild->size + 1; i < newRchild->size; i++, j++) {
      newRchild->key[i] = cursor->key[j];
    }
    for (int i = 0; i < LLeaf->size + 1; i++) {
      newLchild->ptr[i] = virtualPtr[i];
    }
    for (int i = 0, j = LLeaf->size + 1; i < RRLeaf->size + 1; i++, j++) {
      newRchild->ptr[i] = virtualPtr[j];
    }
    if (cursor == root) {
      Node* newRoot = new Node;
```

```
      newRoot->key[0] = cursor->key[newLchild->size];
      newRoot->ptr[0] = newLchild;
      newRoot->ptr[1] = newRchild;
      newRoot->IS_LEAF = false;
      newRoot->size = 1;
      root = newRoot;
      newLchild->parent = newRchild->parent = newRoot;
    } else {
      insertInternal(cursor->key[newLchild->size], cursor->parent, newLchild,
                     newRchild);
    }
  }
}
```

删除

B+ 树的删除也仅在叶子节点中进行，当叶子节点中的最大关键字被删除时，其在非叶子节点中的值可以作为一个分界关键字存在。若因删除而使节点中关键字的个数少于 $\lceil\frac{m}{2}\rceil$ 时，其和兄弟节点的合并过程亦和 B 树类似。

具体步骤如下：

1. 首先查询到键值所在的叶子节点，删除该叶子节点的数据。
2. 如果删除叶子节点之后的数据数量，满足 B+ 树的平衡条件，则直接返回。
3. 否则，就需要做平衡操作：如果该叶子节点的左右兄弟节点的数据量可以借用，就借用过来满足平衡条件。否则，就与相邻的兄弟节点合并成一个新的子节点了。

在上面平衡操作中，如果是进行了合并操作，就需要向上修正父节点的指针：删除被合并节点的键值以及指针。

由于做了删除操作，可能父节点也会不平衡，那么就按照前面的步骤也对父节点进行重新平衡操作，这样一直到某个节点平衡为止。

可以参考 B 树 中的删除章节。

" 实现"

```
// Deletion operation on a B+ tree in C++
#include <climits>
#include <fstream>
#include <iostream>
#include <sstream>
using namespace std;
int MAX = 3;

class BPTree;

class Node {
  bool IS_LEAF;
  int *key, size;
  Node **ptr;
  friend class BPTree;

 public:
  Node();
};

class BPTree {
```

```
  Node *root;
  void insertInternal(int, Node *, Node *);
  void removeInternal(int, Node *, Node *);
  Node *findParent(Node *, Node *);

 public:
  BPTree();
  void search(int);
  void insert(int);
  void remove(int);
  void display(Node *);
  Node *getRoot();
};

Node::Node() {
  key = new int[MAX];
  ptr = new Node *[MAX + 1];
}

BPTree::BPTree() { root = NULL; }

void BPTree::insert(int x) {
  if (root == NULL) {
    root = new Node;
    root->key[0] = x;
    root->IS_LEAF = true;
    root->size = 1;
  } else {
    Node *cursor = root;
    Node *parent;
    while (cursor->IS_LEAF == false) {
      parent = cursor;
      for (int i = 0; i < cursor->size; i++) {
        if (x < cursor->key[i]) {
          cursor = cursor->ptr[i];
          break;
        }
        if (i == cursor->size - 1) {
          cursor = cursor->ptr[i + 1];
          break;
        }
      }
    }
    if (cursor->size < MAX) {
      int i = 0;
      while (x > cursor->key[i] && i < cursor->size) i++;
      for (int j = cursor->size; j > i; j--) {
        cursor->key[j] = cursor->key[j - 1];
      }
      cursor->key[i] = x;
      cursor->size++;
      cursor->ptr[cursor->size] = cursor->ptr[cursor->size - 1];
      cursor->ptr[cursor->size - 1] = NULL;
    } else {
```

```
      Node *newLeaf = new Node;
      int virtualNode[MAX + 1];
      for (int i = 0; i < MAX; i++) {
        virtualNode[i] = cursor->key[i];
      }
      int i = 0, j;
      while (x > virtualNode[i] && i < MAX) i++;
      for (int j = MAX + 1; j > i; j--) {
        virtualNode[j] = virtualNode[j - 1];
      }
      virtualNode[i] = x;
      newLeaf->IS_LEAF = true;
      cursor->size = (MAX + 1) / 2;
      newLeaf->size = MAX + 1 - (MAX + 1) / 2;
      cursor->ptr[cursor->size] = newLeaf;
      newLeaf->ptr[newLeaf->size] = cursor->ptr[MAX];
      cursor->ptr[MAX] = NULL;
      for (i = 0; i < cursor->size; i++) {
        cursor->key[i] = virtualNode[i];
      }
      for (i = 0, j = cursor->size; i < newLeaf->size; i++, j++) {
        newLeaf->key[i] = virtualNode[j];
      }
      if (cursor == root) {
        Node *newRoot = new Node;
        newRoot->key[0] = newLeaf->key[0];
        newRoot->ptr[0] = cursor;
        newRoot->ptr[1] = newLeaf;
        newRoot->IS_LEAF = false;
        newRoot->size = 1;
        root = newRoot;
      } else {
        insertInternal(newLeaf->key[0], parent, newLeaf);
      }
    }
  }
}

void BPTree::insertInternal(int x, Node *cursor, Node *child) {
  if (cursor->size < MAX) {
    int i = 0;
    while (x > cursor->key[i] && i < cursor->size) i++;
    for (int j = cursor->size; j > i; j--) {
      cursor->key[j] = cursor->key[j - 1];
    }
    for (int j = cursor->size + 1; j > i + 1; j--) {
      cursor->ptr[j] = cursor->ptr[j - 1];
    }
    cursor->key[i] = x;
    cursor->size++;
    cursor->ptr[i + 1] = child;
  } else {
    Node *newInternal = new Node;
    int virtualKey[MAX + 1];
```

```
    Node *virtualPtr[MAX + 2];
    for (int i = 0; i < MAX; i++) {
      virtualKey[i] = cursor->key[i];
    }
    for (int i = 0; i < MAX + 1; i++) {
      virtualPtr[i] = cursor->ptr[i];
    }
    int i = 0, j;
    while (x > virtualKey[i] && i < MAX) i++;
    for (int j = MAX + 1; j > i; j--) {
      virtualKey[j] = virtualKey[j - 1];
    }
    virtualKey[i] = x;
    for (int j = MAX + 2; j > i + 1; j--) {
      virtualPtr[j] = virtualPtr[j - 1];
    }
    virtualPtr[i + 1] = child;
    newInternal->IS_LEAF = false;
    cursor->size = (MAX + 1) / 2;
    newInternal->size = MAX - (MAX + 1) / 2;
    for (i = 0, j = cursor->size + 1; i < newInternal->size; i++, j++) {
      newInternal->key[i] = virtualKey[j];
    }
    for (i = 0, j = cursor->size + 1; i < newInternal->size + 1; i++, j++) {
      newInternal->ptr[i] = virtualPtr[j];
    }
    if (cursor == root) {
      Node *newRoot = new Node;
      newRoot->key[0] = cursor->key[cursor->size];
      newRoot->ptr[0] = cursor;
      newRoot->ptr[1] = newInternal;
      newRoot->IS_LEAF = false;
      newRoot->size = 1;
      root = newRoot;
    } else {
      insertInternal(cursor->key[cursor->size], findParent(root, cursor),
                     newInternal);
    }
  }
}

Node *BPTree::findParent(Node *cursor, Node *child) {
  Node *parent;
  if (cursor->IS_LEAF || (cursor->ptr[0])->IS_LEAF) {
    return NULL;
  }
  for (int i = 0; i < cursor->size + 1; i++) {
    if (cursor->ptr[i] == child) {
      parent = cursor;
      return parent;
    } else {
      parent = findParent(cursor->ptr[i], child);
      if (parent != NULL) return parent;
    }
```

```
  }
  return parent;
}

void BPTree::remove(int x) {
  if (root == NULL) {
    cout << "Tree empty\n";
  } else {
    Node *cursor = root;
    Node *parent;
    int leftSibling, rightSibling;
    while (cursor->IS_LEAF == false) {
      for (int i = 0; i < cursor->size; i++) {
        parent = cursor;
        leftSibling = i - 1;
        rightSibling = i + 1;
        if (x < cursor->key[i]) {
          cursor = cursor->ptr[i];
          break;
        }
        if (i == cursor->size - 1) {
          leftSibling = i;
          rightSibling = i + 2;
          cursor = cursor->ptr[i + 1];
          break;
        }
      }
    }
    bool found = false;
    int pos;
    for (pos = 0; pos < cursor->size; pos++) {
      if (cursor->key[pos] == x) {
        found = true;
        break;
      }
    }
    if (!found) {
      cout << "Not found\n";
      return;
    }
    for (int i = pos; i < cursor->size; i++) {
      cursor->key[i] = cursor->key[i + 1];
    }
    cursor->size--;
    if (cursor == root) {
      for (int i = 0; i < MAX + 1; i++) {
        cursor->ptr[i] = NULL;
      }
      if (cursor->size == 0) {
        cout << "Tree died\n";
        delete[] cursor->key;
        delete[] cursor->ptr;
        delete cursor;
        root = NULL;
```

```
      }
      return;
    }
    cursor->ptr[cursor->size] = cursor->ptr[cursor->size + 1];
    cursor->ptr[cursor->size + 1] = NULL;
    if (cursor->size >= (MAX + 1) / 2) {
      return;
    }
    if (leftSibling >= 0) {
      Node *leftNode = parent->ptr[leftSibling];
      if (leftNode->size >= (MAX + 1) / 2 + 1) {
        for (int i = cursor->size; i > 0; i--) {
          cursor->key[i] = cursor->key[i - 1];
        }
        cursor->size++;
        cursor->ptr[cursor->size] = cursor->ptr[cursor->size - 1];
        cursor->ptr[cursor->size - 1] = NULL;
        cursor->key[0] = leftNode->key[leftNode->size - 1];
        leftNode->size--;
        leftNode->ptr[leftNode->size] = cursor;
        leftNode->ptr[leftNode->size + 1] = NULL;
        parent->key[leftSibling] = cursor->key[0];
        return;
      }
    }
    if (rightSibling <= parent->size) {
      Node *rightNode = parent->ptr[rightSibling];
      if (rightNode->size >= (MAX + 1) / 2 + 1) {
        cursor->size++;
        cursor->ptr[cursor->size] = cursor->ptr[cursor->size - 1];
        cursor->ptr[cursor->size - 1] = NULL;
        cursor->key[cursor->size - 1] = rightNode->key[0];
        rightNode->size--;
        rightNode->ptr[rightNode->size] = rightNode->ptr[rightNode->size + 1];
        rightNode->ptr[rightNode->size + 1] = NULL;
        for (int i = 0; i < rightNode->size; i++) {
          rightNode->key[i] = rightNode->key[i + 1];
        }
        parent->key[rightSibling - 1] = rightNode->key[0];
        return;
      }
    }
    if (leftSibling >= 0) {
      Node *leftNode = parent->ptr[leftSibling];
      for (int i = leftNode->size, j = 0; j < cursor->size; i++, j++) {
        leftNode->key[i] = cursor->key[j];
      }
      leftNode->ptr[leftNode->size] = NULL;
      leftNode->size += cursor->size;
      leftNode->ptr[leftNode->size] = cursor->ptr[cursor->size];
      removeInternal(parent->key[leftSibling], parent, cursor);
      delete[] cursor->key;
      delete[] cursor->ptr;
      delete cursor;
```

```
    } else if (rightSibling <= parent->size) {
      Node *rightNode = parent->ptr[rightSibling];
      for (int i = cursor->size, j = 0; j < rightNode->size; i++, j++) {
        cursor->key[i] = rightNode->key[j];
      }
      cursor->ptr[cursor->size] = NULL;
      cursor->size += rightNode->size;
      cursor->ptr[cursor->size] = rightNode->ptr[rightNode->size];
      cout << "Merging two leaf nodes\n";
      removeInternal(parent->key[rightSibling - 1], parent, rightNode);
      delete[] rightNode->key;
      delete[] rightNode->ptr;
      delete rightNode;
    }
  }
}

void BPTree::removeInternal(int x, Node *cursor, Node *child) {
  if (cursor == root) {
    if (cursor->size == 1) {
      if (cursor->ptr[1] == child) {
        delete[] child->key;
        delete[] child->ptr;
        delete child;
        root = cursor->ptr[0];
        delete[] cursor->key;
        delete[] cursor->ptr;
        delete cursor;
        cout << "Changed root node\n";
        return;
      } else if (cursor->ptr[0] == child) {
        delete[] child->key;
        delete[] child->ptr;
        delete child;
        root = cursor->ptr[1];
        delete[] cursor->key;
        delete[] cursor->ptr;
        delete cursor;
        cout << "Changed root node\n";
        return;
      }
    }
  }
  int pos;
  for (pos = 0; pos < cursor->size; pos++) {
    if (cursor->key[pos] == x) {
      break;
    }
  }
  for (int i = pos; i < cursor->size; i++) {
    cursor->key[i] = cursor->key[i + 1];
  }
  for (pos = 0; pos < cursor->size + 1; pos++) {
    if (cursor->ptr[pos] == child) {
```

```
      break;
    }
  }
  for (int i = pos; i < cursor->size + 1; i++) {
    cursor->ptr[i] = cursor->ptr[i + 1];
  }
  cursor->size--;
  if (cursor->size >= (MAX + 1) / 2 - 1) {
    return;
  }
  if (cursor == root) return;
  Node *parent = findParent(root, cursor);
  int leftSibling, rightSibling;
  for (pos = 0; pos < parent->size + 1; pos++) {
    if (parent->ptr[pos] == cursor) {
      leftSibling = pos - 1;
      rightSibling = pos + 1;
      break;
    }
  }
  if (leftSibling >= 0) {
    Node *leftNode = parent->ptr[leftSibling];
    if (leftNode->size >= (MAX + 1) / 2) {
      for (int i = cursor->size; i > 0; i--) {
        cursor->key[i] = cursor->key[i - 1];
      }
      cursor->key[0] = parent->key[leftSibling];
      parent->key[leftSibling] = leftNode->key[leftNode->size - 1];
      for (int i = cursor->size + 1; i > 0; i--) {
        cursor->ptr[i] = cursor->ptr[i - 1];
      }
      cursor->ptr[0] = leftNode->ptr[leftNode->size];
      cursor->size++;
      leftNode->size--;
      return;
    }
  }
  if (rightSibling <= parent->size) {
    Node *rightNode = parent->ptr[rightSibling];
    if (rightNode->size >= (MAX + 1) / 2) {
      cursor->key[cursor->size] = parent->key[pos];
      parent->key[pos] = rightNode->key[0];
      for (int i = 0; i < rightNode->size - 1; i++) {
        rightNode->key[i] = rightNode->key[i + 1];
      }
      cursor->ptr[cursor->size + 1] = rightNode->ptr[0];
      for (int i = 0; i < rightNode->size; ++i) {
        rightNode->ptr[i] = rightNode->ptr[i + 1];
      }
      cursor->size++;
      rightNode->size--;
      return;
    }
  }
```

```
  if (leftSibling >= 0) {
    Node *leftNode = parent->ptr[leftSibling];
    leftNode->key[leftNode->size] = parent->key[leftSibling];
    for (int i = leftNode->size + 1, j = 0; j < cursor->size; j++) {
      leftNode->key[i] = cursor->key[j];
    }
    for (int i = leftNode->size + 1, j = 0; j < cursor->size + 1; j++) {
      leftNode->ptr[i] = cursor->ptr[j];
      cursor->ptr[j] = NULL;
    }
    leftNode->size += cursor->size + 1;
    cursor->size = 0;
    removeInternal(parent->key[leftSibling], parent, cursor);
  } else if (rightSibling <= parent->size) {
    Node *rightNode = parent->ptr[rightSibling];
    cursor->key[cursor->size] = parent->key[rightSibling - 1];
    for (int i = cursor->size + 1, j = 0; j < rightNode->size; j++) {
      cursor->key[i] = rightNode->key[j];
    }
    for (int i = cursor->size + 1, j = 0; j < rightNode->size + 1; j++) {
      cursor->ptr[i] = rightNode->ptr[j];
      rightNode->ptr[j] = NULL;
    }
    cursor->size += rightNode->size + 1;
    rightNode->size = 0;
    removeInternal(parent->key[rightSibling - 1], parent, rightNode);
  }
}

void BPTree::display(Node *cursor) {
  if (cursor != NULL) {
    for (int i = 0; i < cursor->size; i++) {
      cout << cursor->key[i] << " ";
    }
    cout << "\n";
    if (cursor->IS_LEAF != true) {
      for (int i = 0; i < cursor->size + 1; i++) {
        display(cursor->ptr[i]);
      }
    }
  }
}

Node *BPTree::getRoot() { return root; }

int main() {
  BPTree node;
  node.insert(5);
  node.insert(15);
  node.insert(25);
  node.insert(35);
  node.insert(45);

  node.display(node.getRoot());
```

```
  node.remove(15);

  node.display(node.getRoot());
}
```

参考资料

- B+ tree wikipedia[2]
- B 树、B+ 树索引算法原理（下）[3]
- B+ 树详解 + 代码实现（插入篇）[4]
- Deletion from a B+ Tree[5]

参考资料与注释

[1] BPlustree

[2] B+ tree wikipedia

[3] B 树、B+ 树索引算法原理（下）

[4] B+ 树详解 + 代码实现（插入篇）

[5] Deletion from a B+ Tree

10.17.9 替罪羊树

Authors: Ir1d, 0xis-cn

引入

替罪羊树是一种依靠重构操作维持平衡的重量平衡树。替罪羊树会在插入、删除操作时，检测途经的节点，若发现失衡，则将以该节点为根的子树重构。

我们在此实现一个可重的权值平衡树。

" 实现"

```
int cnt,               // 树中元素总数
    rt,                // 根节点，初值为 0 代表空树
    w[MAXN],           // 点中的数据 / 权值
    lc[MAXN], rc[MAXN], // 左右子树
    wn[MAXN],          // 本数据出现次数（为 0 代表已删除）
    s[MAXN],  // 以本节点为根的子树大小（每个节点记 1 次）
    sz[MAXN],  // 以本节点为根的子树大小（每个节点记 wn[k] 次）
    sd[MAXN];  // 已删除节点不计的子树大小（每个节点记 1 次）

void Calc(int k) {
  // 重新计算以 k 为根的子树大小
  s[k] = s[lc[k]] + s[rc[k]] + 1;
```

```
  sz[k] = sz[lc[k]] + sz[rc[k]] + wn[k];
  sd[k] = sd[lc[k]] + sd[rc[k]] + (wn[k] != 0);
}
```

重构

过程

首先，如前所述，我们需要判定一个节点是否应重构。为此我们引入一个比例常数 α（取值在 $(0.5, 1)$，一般采用 0.7 或 0.8），若某节点的子节点大小占它本身大小的比例超过 α，则重构。

另外由于我们采用惰性删除（删除只使用 `wn[k]--`），已删除节点过多也影响效率。因此若未被删除的子树大小占总大小的比例低于 α，则亦重构。

实现

```
bool CanRbu(int k) {
  // 判断节点 k 是否需要重构
  return wn[k] && (alpha * s[k] <= (double)std::max(s[lc[k]], s[rc[k]]) ||
                   (double)sd[k] <= alpha * s[k]);
}
```

重构分为两个步骤——先中序遍历展开存入数组，再二分重建成树。

```
void Rbu_Flatten(int& ldc, int k) {
  // 中序遍历展开以 k 节点为根子树
  if (!k) return;
  Rbu_Flatten(ldc, lc[k]);
  if (wn[k]) ldr[ldc++] = k;
  // 若当前节点已删除则不保留
  Rbu_Flatten(ldc, rc[k]);
}

int Rbu_Build(int l, int r) {
  // 将 ldr[] 数组内 [l, r) 区间重建成树，返回根节点
  int mid = l + r >> 1;  // 选取中间为根使其平衡
  if (l >= r) return 0;
  lc[ldr[mid]] = Rbu_Build(l, mid);
  rc[ldr[mid]] = Rbu_Build(mid + 1, r);  // 建左右子树
  Calc(ldr[mid]);
  return ldr[mid];
}

void Rbu(int& k) {
  // 重构节点 k 的全过程
  int ldc = 0;
  Rbu_Flatten(ldc, k);
  k = Rbu_Build(0, ldc);
}
```

实现

几种操作的处理方式较为类似，都规定了**到达空结点**与**找到对应结点**的行为，之后按**小于向左**、**大于向右**的方式向下递归。

插入

插入时，到达空结点则新建节点，找到对应结点则 `wn[k]++`。递归结束后，途经的节点可重构的要重构。

```
void Ins(int& k, int p) {
  // 在以 k 为根的子树内添加权值为 p 节点
  if (!k) {
    k = ++cnt;
    if (!rt) rt = 1;
    w[k] = p;
    lc[k] = rc[k] = 0;
    wn[k] = s[k] = sz[k] = sd[k] = 1;
  } else {
    if (w[k] == p)
      wn[k]++;
    else if (w[k] < p)
      Ins(rc[k], p);
    else
      Ins(lc[k], p);
    Calc(k);
    if (CanRbu(k)) Rbu(k);
  }
}
```

删除

惰性删除，到达空结点则忽略，找到对应结点则 `wn[k]--`。递归结束后，可重构节点要重构。

```
void Del(int& k, int p) {
  // 从以 k 为根子树移除权值为 p 节点
  if (!k)
    return;
  else {
    if (w[k] == p) {
      if (wn[k]) wn[k]--;
    } else {
      if (w[k] < p)
        Del(rc[k], p);
      else
        Del(lc[k], p);
    }
    Calc(k);
    if (CanRbu(k)) Rbu(k);
  }
}
```

upper_bound

返回权值严格大于某值的最小名次。

到达空结点则返回 1，因为只有该子树左边的数均小于查找数才会递归至此。找到对应结点，则返回该节点所占据的最后一个名次 + 1。

```
int MyUprBd(int k, int p) {
  // 在以 k 为根子树中，大于 p 的最小数的名次
  if (!k)
    return 1;
  else if (w[k] == p && wn[k])
    return sz[lc[k]] + 1 + wn[k];
```

```
    else if (p < w[k])
      return MyUprBd(lc[k], p);
    else
      return sz[lc[k]] + wn[k] + MyUprBd(rc[k], p);
}
```

以下是反义函数，相当于采用 `std::greater<>` 比较，即返回权值严格小于某值的最大名次。查询一个数的排名可以用 `MyUprGrt(rt, x) + 1`。

```
int MyUprGrt(int k, int p) {
  if (!k)
    return 0;
  else if (w[k] == p && wn[k])
    return sz[lc[k]];
  else if (w[k] < p)
    return sz[lc[k]] + wn[k] + MyUprGrt(rc[k], p);
  else
    return MyUprGrt(lc[k], p);
}
```

at

给定名次，返回该名次上的权值。到达空结点说明无此名次，找到对应结点则返回其权值。

```
int MyAt(int k, int p) {
  // 以 k 为根的子树中，名次为 p 的权值
  if (!k)
    return 0;
  else if (sz[lc[k]] < p && p <= sz[lc[k]] + wn[k])
    return w[k];
  else if (sz[lc[k]] + wn[k] < p)
    return MyAt(rc[k], p - sz[lc[k]] - wn[k]);
  else
    return MyAt(lc[k], p);
}
```

前驱后继

以上两种功能结合即可。

```
int MyPre(int k, int p) { return MyAt(k, MyUprGrt(k, p)); }

int MyPost(int k, int p) { return MyAt(k, MyUprBd(k, p)); }
```

10.17.10 Leafy Tree

Authors: Persdre

Leafy Tree 简介

Leafy Tree 是一种依靠旋转维持重量平衡的平衡树。通过判断一棵树的数据存储位置在每个节点上还是仅在叶子节点上，我们可以将树分为 Nodey 和 Leafy 的。Leafy Tree 被定义为一种维护的信息全部储存在叶子节点上的二叉树。这种结构中，每个叶子存储值，每个非叶节点值负责维护树的形态而不维护树的信息，但通常会维护孩子信息，从而加速查询。线段树的结构属于一种 Leafy Tree。所以 Leafy Tree 也被称为平衡线段树。

Leafy Tree 的特点

1. 所有的信息维护在叶子节点上。
2. 类似 Kruskal 重构树的结构，每个非叶子节点一定有两个孩子，且非叶子节点统计两个孩子的信息（类似线段树上传信息），所以维护 n 个信息的 Leafy Tree 有 $2n-1$ 个节点。
3. 可以完成区间操作，比如翻转，以及可持久化等。

注意到，一个 Leafy 结构的每个节点必定有两个孩子。对其进行插入删除时，在插入删除叶子时必定会额外修改一个非叶子节点。常见的平衡树均属于每个节点同时维护值和结构的 Nodey Tree。如果将一个 Nodey 结构的所有孩子的空指针指向一个维护值的节点，那么这棵树将变为一个 Leafy 结构。

Leafy Tree 是一个纯函数化的数据结构，因此其在实现函数化数据结构和可持久化效率上具有先天优势，时间效率极高。

一个简单的图例如下：

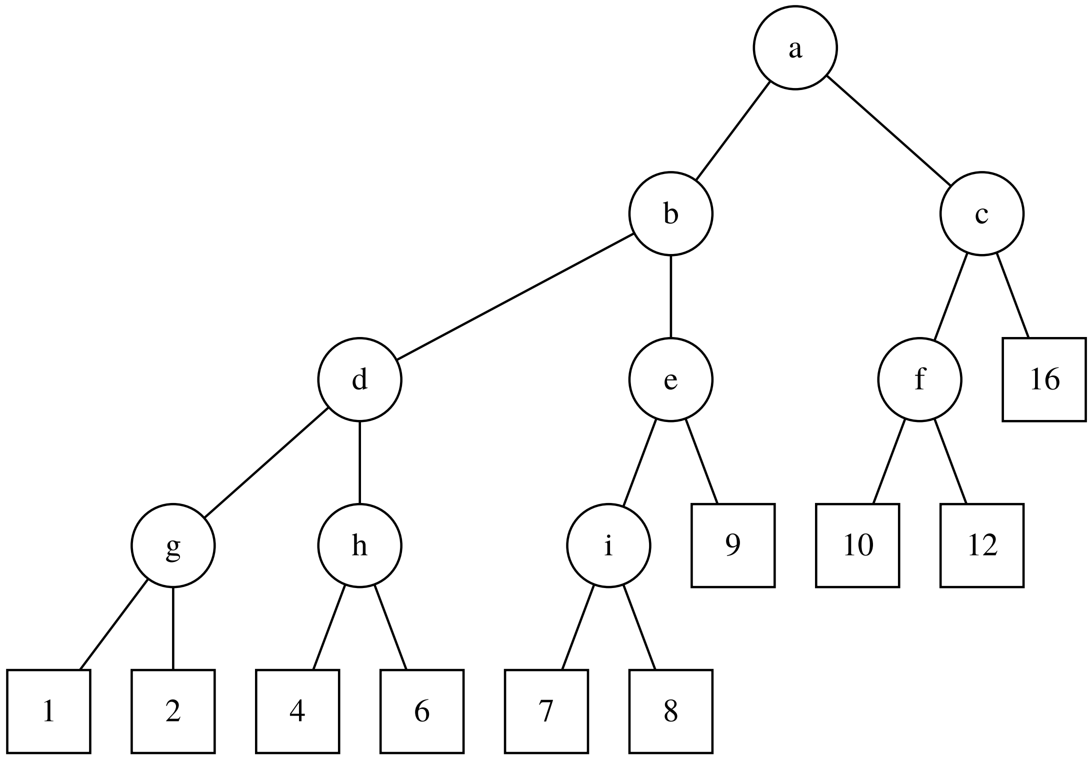


图 10.96

基本操作

Leafy Tree 的基本操作有：旋转，插入，删除，查找。

旋转

Leafy Tree 的旋转操作类似于替罪羊树，仅需一次旋转。

```
#define new_Node(a, b, c, d) (&(*st[cnt++] = Node(a, b, c, d)))
#define merge(a, b) new_Node(a->size + b->size, b->val, a, b)
#define ratio 4

struct Node {
  int size, val;
  Node *lf, *rf;
```

```
  Node(int a, int b, Node *l, Node *r) : size(a), val(b), lf(l), rf(r) {}

  Node() {}
};

Node *root, *null, *st[200010], t[200010];

void rotate(Node *u) {
  if (u->lf->size > u->rf->size * ratio)
    u->rf = merge(u->lf->rf, u->rf), st[--cnt] = u->lf, u->lf = u->lf->lf;
  if (u->rf->size > u->lf->size * ratio)
    u->lf = merge(u->lf, u->rf->lf), st[--cnt] = u->rf, u->rf = u->rf->rf;
}
```

插入

类似二叉树的插入过程。

```
void insert(Node *u, int x) {
  if (u->size == 1)
    u->lf = new_Node(1, Min(u->val, x), null, null),
    u->rf = new_Node(1, Max(u->val, x), null, null);
  else
    insert(x > u->lf->val ? u->rf : u->lf, x);
  update(u), rotate(u);
}
```

删除

根据二叉搜索树的性质，找到要删除的数所在的父亲节点，再暴力枚举在左孩子还是右孩子，然后将剩下的一个节点合并到当前节点。

删除的代码如下：

```
void BTreeNode::traverse() {
  if (u->lf->size == 1 && u->lf->val == x)
    st[--cnt] = u->lf, st[--cnt] = u->rf, *u = *u->rf;
  else if (u->rf->size == 1 && u->rf->val == x)
    st[--cnt] = u->lf, st[--cnt] = u->rf, *u = *u->lf;
  else
    erase(x > u->lf->val ? u->rf : u->lf, x);
  update(u), rotate(u);
```

查找

假设需要查找排名第 x 大的元素。

```
int find(Node *u, int x) {
  if (u->size == 1) return u->val;
  return u->lf->size < x ? find(u->rf, x - u->lf->size) : find(u->lf, x);
}
```

普通平衡树的模版代码

```cpp
#include <bits/stdc++.h>
#define update(u) \
  if (u->lf->size) u->size = u->lf->size + u->rf->size, u->val = u->rf->val
#define new_Node(a, b, c, d) (&(*st[cnt++] = Node(a, b, c, d)))
#define merge(a, b) new_Node(a->size + b->size, b->val, a, b)
#define ratio 4
using namespace std;

int Min(const int x, const int y) { return x < y ? x : y; }

int Max(const int x, const int y) { return x > y ? x : y; }

int n, cnt;

struct Node {
  int size, val;
  Node *lf, *rf;

  Node(int a, int b, Node *l, Node *r) : size(a), val(b), lf(l), rf(r) {}

  Node() {}
};

Node *root, *null, *st[200010], t[200010];

void rotate(Node *u) {
  if (u->lf->size > u->rf->size * ratio)
    u->rf = merge(u->lf->rf, u->rf), st[--cnt] = u->lf, u->lf = u->lf->lf;
  if (u->rf->size > u->lf->size * ratio)
    u->lf = merge(u->lf, u->rf->lf), st[--cnt] = u->rf, u->rf = u->rf->rf;
}

void insert(Node *u, int x) {
  if (u->size == 1)
    u->lf = new_Node(1, Min(u->val, x), null, null),
    u->rf = new_Node(1, Max(u->val, x), null, null);
  else
    insert(x > u->lf->val ? u->rf : u->lf, x);
  update(u), rotate(u);
}

void erase(Node *u, int x) {
  if (u->lf->size == 1 && u->lf->val == x)
    st[--cnt] = u->lf, st[--cnt] = u->rf, *u = *u->rf;
  else if (u->rf->size == 1 && u->rf->val == x)
    st[--cnt] = u->lf, st[--cnt] = u->rf, *u = *u->lf;
  else
    erase(x > u->lf->val ? u->rf : u->lf, x);
  update(u), rotate(u);
}

int find(Node *u, int x) {
  if (u->size == 1) return u->val;
  return u->lf->size < x ? find(u->rf, x - u->lf->size) : find(u->lf, x);
```

```
}

int Rank(Node *u, int x) {
  if (u->size == 1) return 1;
  return u->lf->val < x ? u->lf->size + Rank(u->rf, x) : Rank(u->lf, x);
}

int pre(int x) { return find(root, Rank(root, x) - 1); }

int nxt(int x) { return find(root, Rank(root, x + 1)); }

void debug(Node *u) {
  if (u->lf != null) debug(u->lf);
  if (u->size == 1) printf(" %d", u->val);
  if (u->rf != null) debug(u->rf);
}

int main() {
  scanf("%d", &n);
  cnt = 0;
  null = new Node(0, 0, 0, 0);
  root = new Node(1, INT_MAX, null, null);
  for (int i = 0; i <= 200000; i++) st[i] = &t[i];
  for (int i = 1; i <= n; i++) {
    int t, a;
    scanf("%d%d", &t, &a);
    if (t == 1)
      insert(root, a);
    else if (t == 2)
      erase(root, a);
    else if (t == 3)
      printf("%d\n", Rank(root, a));
    else if (t == 4)
      printf("%d\n", find(root, a));
    else if (t == 5)
      printf("%d\n", pre(a));
    else if (t == 6)
      printf("%d\n", nxt(a));
  }
  return 0;
}
```

例题

- Luogu P2286 宠物收养场[1]

参考资料

- WBLT 学习笔记[2]
- Leafy Tree[3]
- Luogu P2286[HNOI2004] 宠物收养场[4]

参考资料与注释

[1] Luogu P2286 宠物收养场

[2] WBLT 学习笔记

[3] Leafy Tree

[4] Luogu P2286[HNOI2004] 宠物收养场

10.17.11 笛卡尔树

Authors: sshwy, zhouyuyang2002, StudyingFather, Ir1d, ouuan, Enter-tainer

引入

本文介绍一种不太常用，但是与大家熟知的平衡树与堆密切相关的数据结构——笛卡尔树。

笛卡尔树是一种二叉树，每一个结点由一个键值二元组 (k,w) 构成。要求 k 满足二叉搜索树的性质，而 w 满足堆的性质。一个有趣的事实是，如果笛卡尔树的 k,w 键值确定，且 k 互不相同，w 互不相同，那么这个笛卡尔树的结构是唯一的。上图：

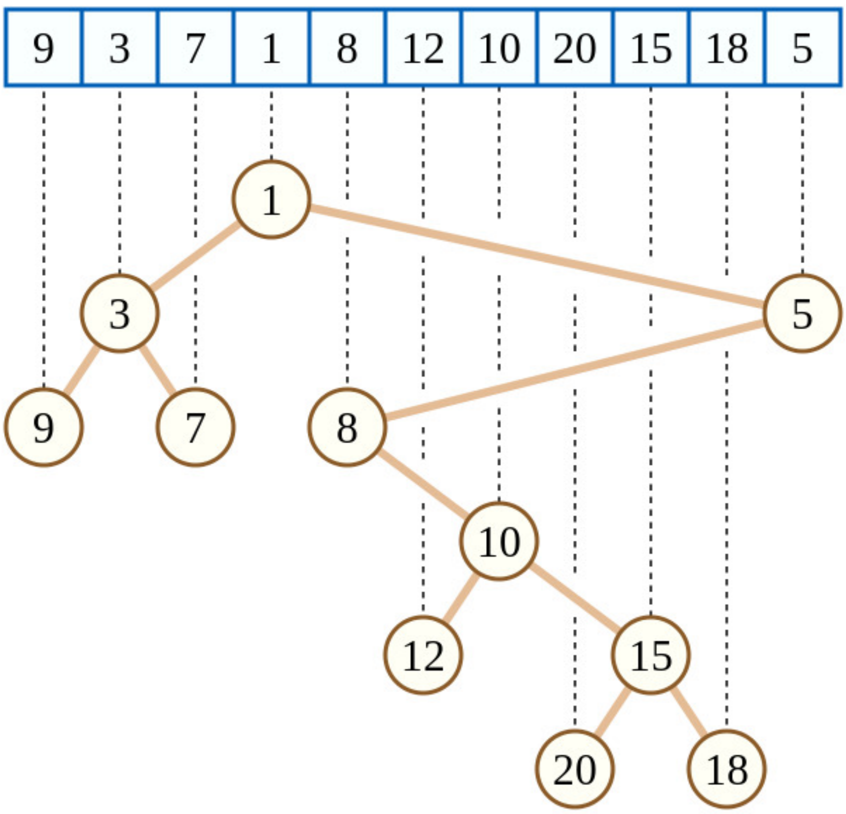


图 10.97 eg

（图源自维基百科）

上面这棵笛卡尔树相当于把数组元素值当作键值 w，而把数组下标当作键值 k。显然可以发现，这棵树的键值 k 满足二叉搜索树的性质，而键值 w 满足小根堆的性质。

其实图中的笛卡尔树是一种特殊的情况，因为二元组的键值 k 恰好对应数组下标，这种特殊的笛卡尔树有一个性质，就是一棵子树内的下标是连续的一个区间（这样才能满足二叉搜索树的性质）。更一般的情况则是任意二元组构建的笛卡尔树。

构建

栈构建

过程

我们考虑将元素按照键值 k 排序。然后一个一个插入到当前的笛卡尔树中。那么每次我们插入的元素必然在这个树的右链（右链：即从根结点一直往右子树走，经过的结点形成的链）的末端。于是我们执行这样一个过程，从下往上比较右链结点与当前结点 u 的 w，如果找到了一个右链上的结点 x 满足 $x_w < u_w$，就把 u 接到 x 的右儿子上，而 x 原本的右子树就变成 u 的左子树。

具体不解释，我们直接上图。图中红色框框部分就是我们始终维护的右链：

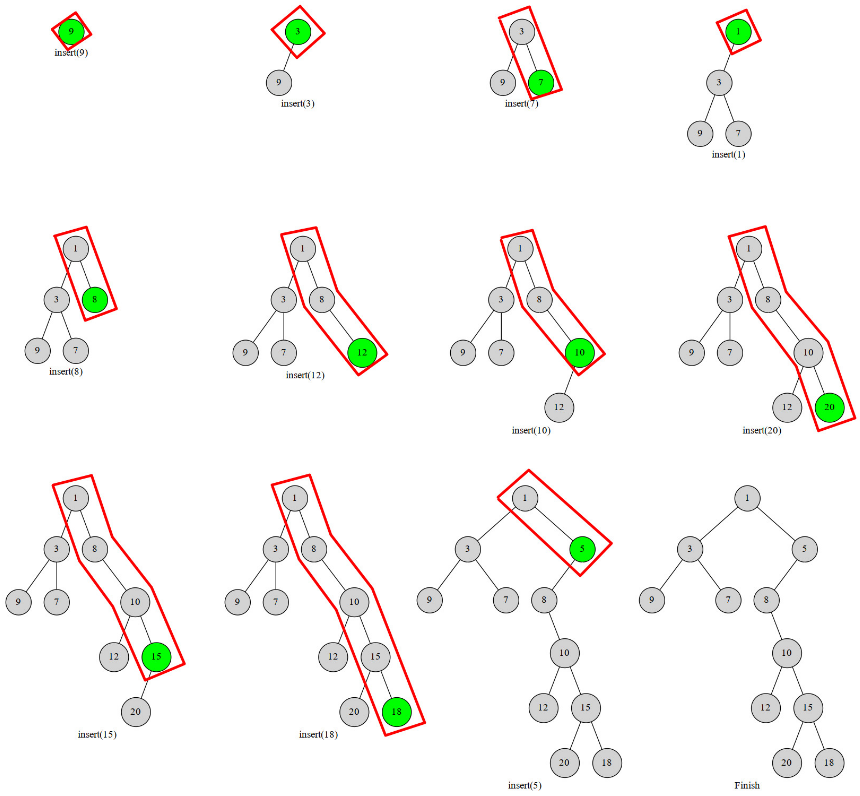


图 10.98 build

显然每个数最多进出右链一次（或者说每个点在右链中存在的是一段连续的时间）。这个过程我们可以用栈维护，栈中维护当前笛卡尔树的右链上的结点。一个点不在右链上了就把它弹掉。这样每个点最多进出一次，复杂度 $O(n)$。

实现

伪代码如下：

```
新建一个大小为 n 的空栈。用 top 来标操作前的栈顶，k 来标记当前栈顶。
For i := 1 to n
    k := top
    While 栈非空且栈顶元素 > 当前元素
        k--
```

```
    if 栈非空
        栈顶元素. 右儿子 := 当前元素
    if k < top
        当前元素. 左儿子 := 上一个被弹出的元素
    当前元素入栈
    top := k
```

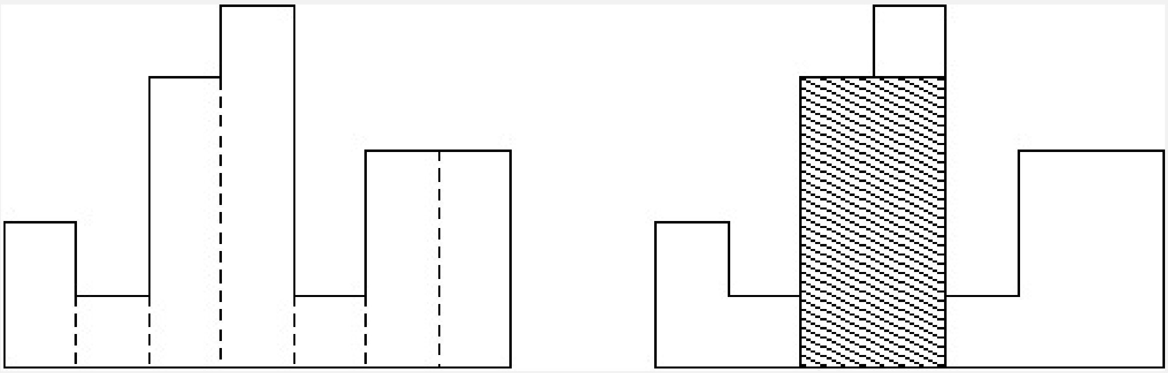
```cpp
for (int i = 1; i <= n; i++) {
  int k = top;
  while (k > 0 && h[stk[k]] > h[i]) k--;
  if (k) rs[stk[k]] = i;  // rs 代表笛卡尔树每个节点的右儿子
  if (k < top) ls[i] = stk[k + 1];  // ls 代表笛卡尔树每个节点的左儿子
  stk[++k] = i;
  top = k;
}
```

笛卡尔树与 Treap

谈到笛卡尔树，很容易让人想到一种家喻户晓的结构——Treap。没错，Treap 是笛卡尔树的一种，只不过 w 的值完全随机。Treap 也有线性的构建算法，如果提前将元素排好序，显然可以使用上述单调栈算法完成构建过程，只不过很少会这么用。

例题

HDU 1506 最大子矩形

> 题目大意：n 个位置，每个位置上的高度是 h_i，求最大子矩阵。举一个例子，如下图：
>
> 
>
> 图 10.99 eg
>
> 阴影部分就是图中的最大子矩阵。

这道题你可 DP，可单调栈，但你万万没想到的是它也可以笛卡尔树！具体地，我们把下标作为键值 k，h_i 作为键值 w 满足小根堆性质，构建一棵 (i, h_i) 的笛卡尔树。

这样我们枚举每个结点 u，把 u_w（即结点 u 的高度键值 h）作为最大子矩阵的高度。由于我们建立的笛卡尔树满足小根堆性质，因此 u 的子树内的结点的高度都大于等于 u。而我们又知道 u 子树内的下标是一段连续的区间。于是我们只需要知道子树的大小，然后就可以算这个区间的最大子矩阵的面积了。用每一个点计算出来的值更新答案即可。显然这个可以一次 DFS 完成，因此复杂度仍是 $O(n)$ 的。

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <iostream>
using namespace std;
typedef long long ll;
const int N = 100000 + 10, INF = 0x3f3f3f3f;

struct node {
  int idx, val, par, ch[2];

  friend bool operator<(node a, node b) { return a.idx < b.idx; }

  void init(int _idx, int _val, int _par) {
    idx = _idx, val = _val, par = _par, ch[0] = ch[1] = 0;
  }
} tree[N];

int root, top, stk[N];
ll ans;

int cartesian_build(int n) {  // 建树，满足小根堆性质
  for (int i = 1; i <= n; i++) {
    int k = i - 1;
    while (tree[k].val > tree[i].val) k = tree[k].par;
    tree[i].ch[0] = tree[k].ch[1];
    tree[k].ch[1] = i;
    tree[i].par = k;
    tree[tree[i].ch[0]].par = i;
  }
  return tree[0].ch[1];
}

int dfs(int x) {  // 一次 dfs 更新答案就可以了
  if (!x) return 0;
  int sz = dfs(tree[x].ch[0]);
  sz += dfs(tree[x].ch[1]);
  ans = max(ans, (ll)(sz + 1) * tree[x].val);
  return sz + 1;
}

int main() {
  int n, hi;
  while (scanf("%d", &n), n) {
    tree[0].init(0, 0, 0);
    for (int i = 1; i <= n; i++) {
      scanf("%d", &hi);
      tree[i].init(i, hi, 0);
    }
    root = cartesian_build(n);
    ans = 0;
    dfs(root);
    printf("%lld\n", ans);
  }
```

```
    return 0;
}
```

参考资料

维基百科 - 笛卡尔树[1]

参考资料与注释

[1] 维基百科 - 笛卡尔树

10.17.12 红黑树

Authors: LeverImmy

红黑树是一种自平衡的二叉搜索树。每个节点额外存储了一个 color 字段 (”RED” or ”BLACK”)，用于确保树在插入和删除时保持平衡。

性质

一棵合法的红黑树必须遵循以下四条性质：

1. 节点为红色或黑色
2. NIL 节点（空叶子节点）为黑色
3. 红色节点的子节点为黑色
4. 从根节点到 NIL 节点的每条路径上的黑色节点数量相同

下图为一棵合法的红黑树：

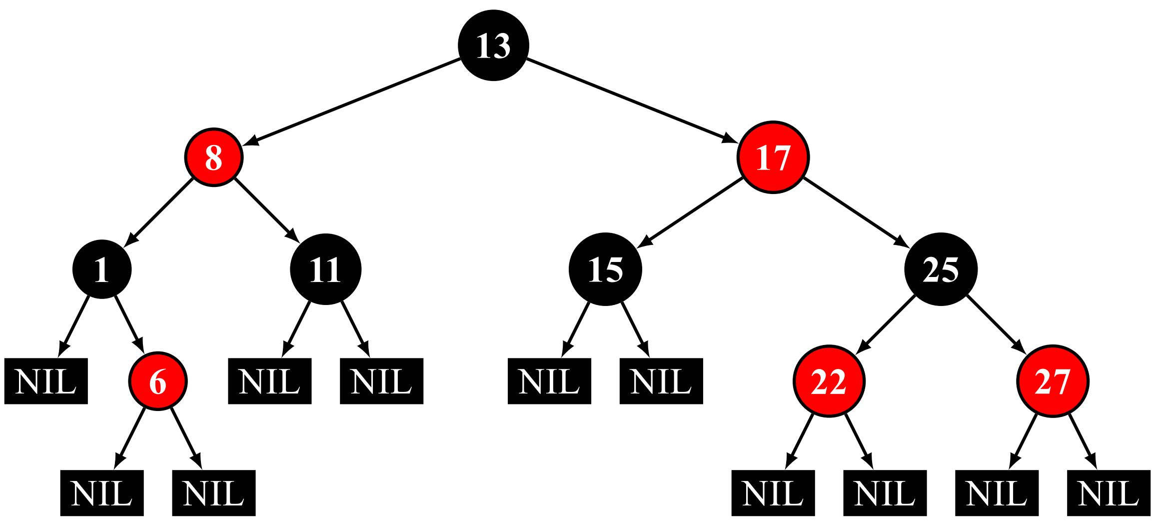


图 10.100 rbtree-example

注：部分资料中还加入了第五条性质，即根节点必须为黑色，这条性质要求完成插入操作后若根节点为红色则将其染黑，但由于将根节点染黑的操作也可以延迟至删除操作时进行，因此，该条性质并非必须满足。（在本文给出的代码实现中就没有选择满足该性质）。为严谨起见，这里同时引用 维基百科原文[4] 进行说明：

> Some authors, e.g. Cormen & al.,[1]claim "the root is black" as fifth requirement; but not Mehlhorn & Sanders[2]or Sedgewick & Wayne.[3]Since the root can always be changed from red to black, this rule has little effect on analysis. This article also omits it, because it slightly disturbs the recursive algorithms and proofs.

结构

红黑树类的定义

```cpp
template <typename Key, typename Value, typename Compare = std::less<Key>>
class RBTreeMap {
  // 排序函数
  Compare compare = Compare();

  // 节点结构体
  struct Node {
    ...
  };

  // 根节点指针
  Node* root = nullptr;
  // 记录红黑树中当前的节点个数
  size_t count = 0;
}
```

节点维护的信息

Identifier	Type	Description
`left`	`Node*`	左子节点指针
`right`	`Node*`	右子节点指针
`parent`	`Node*`	父节点指针
`color`	`enum { BLACK, RED }`	颜色枚举
`key`	`Key`	节点键值，具有唯一性和可排序性
`value`	`Value`	节点内储存的值

注：由于本文提供的代码示例中使用 `std::share_ptr` 进行内存管理，对此不熟悉的读者可以将下文中所有的 `NodePtr` 和 `ConstNodePtr` 理解为裸指针 `Node*`。但在实现删除操作时若使用 `Node*` 作为节点引用需注意应手动释放内存以避免内存泄漏，该操作在使用 `std::shared_ptr` 作为节点引用的示例代码中并未体现。

过程

注：由于红黑树是由 B 树衍生而来（发明时的最初的名字 symmetric binary B-tree 足以证明这点），并非直接由平衡二叉树外加限制条件推导而来，插入操作的后续维护和删除操作的后续维护中部分对操作的解释作用仅是帮助理解，并不能将其作为该操作的原理推导和证明。

旋转操作

旋转操作是多数平衡树能够维持平衡的关键，它能在不改变一棵合法 BST 中序遍历结果的情况下改变局部节点的深度。

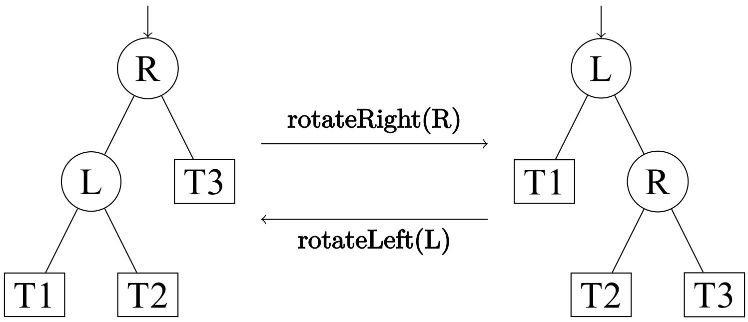


图 10.101 rbtree-rotations

如上图，从左图到右图的过程被称为右旋，右旋操作会使得 $T3$ 子树上结点的深度均减 1，使 $T1$ 子树上结点的深度均加 1，而 $T2$ 子树上节点的深度则不变。从右图到左图的过程被称为左旋，左旋是右旋的镜像操作。

这里给出红黑树中节点的左旋操作的示例代码：

" 实现"

```
void rotateLeft(ConstNodePtr node) {
  // clang-format off
  //      |                       |
  //      N                       S
  //     / \     l-rotate(N)     / \
  //    L   S    ==========>    N   R
  //       / \                 / \
  //      M   R               L   M
  assert(node != nullptr && node->right != nullptr);
  // clang-format on
  NodePtr parent = node->parent;
  Direction direction = node->direction();

  NodePtr successor = node->right;
  node->right = successor->left;
  successor->left = node;

  // 以下的操作用于维护各个节点的`parent`指针
  // `Direction`的定义以及`maintainRelationship`
  // 的实现请参照文章末尾的完整示例代码
  maintainRelationship(node);
  maintainRelationship(successor);

  switch (direction) {
    case Direction::ROOT:
      this->root = successor;
      break;
    case Direction::LEFT:
      parent->left = successor;
```

```
      break;
    case Direction::RIGHT:
      parent->right = successor;
      break;
  }

  successor->parent = parent;
}
```

注：代码中的 successor 并非平衡树中的后继节点，而是表示取代原本节点的新节点，由于在图示中 replacement 的简称 R 会与右子节点的简称 R 冲突，因此此处使用 successor 避免歧义。

插入操作

红黑树的插入操作与普通的 BST 类似，对于红黑树来说，新插入的节点初始为红色，完成插入后需根据插入节点及相关节点的状态进行修正以满足上文提到的四条性质。

插入后的平衡维护

Case 1

该树原先为空，插入第一个节点后不需要进行修正。

Case 2

当前的节点的父节点为黑色且为根节点，这时性质已经满足，不需要进行修正。

Case 3

当前节点 N 的父节点 P 是为根节点且为红色，将其染为黑色即可，此时性质也已满足，不需要进一步修正。

" 实现"

```
// clang-format off
// Case 3: Parent is root and is RED
//   Paint parent to BLACK.
//    <P>         [P]
//     |   ====>   |
//    <N>         <N>
//   p.s.
//    `<X>` is a RED node;
//    `[X]` is a BLACK node (or NIL);
//    `{X}` is either a RED node or a BLACK node;
// clang-format on
assert(node->parent->isRed());
node->parent->color = Node::BLACK;
return;
```

Case 4

当前节点 N 的父节点 P 和叔节点 U 均为红色，此时 P 包含了一个红色子节点，违反了红黑树的性质，需要进行重新染色。由于在当前节点 N 之前该树是一棵合法的红黑树，根据性质 3 可以确定 N 的祖父节点 G 一定是黑色，这时只要后续操作可以保证以 G 为根节点的子树在不违反性质 4 的情况下再递归维护祖父节点 G 以保证性质 3 即可。

因此，这种情况的维护需要：

1. 将 P，U 节点染黑，将 G 节点染红（可以保证每条路径上黑色节点个数不发生改变）。
2. 递归维护 G 节点（因为不确定 G 的父节点的状态，递归维护可以确保性质 3 成立）。

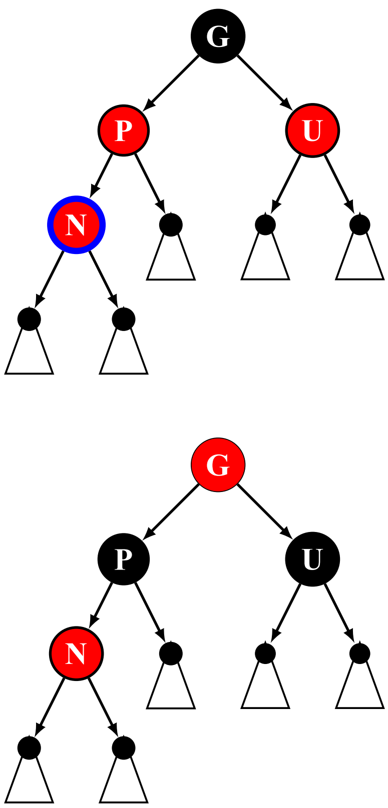


图 10.102 rbtree-insert-case4

" 实现"

```
// clang-format off
// Case 4: Both parent and uncle are RED
//   Paint parent and uncle to BLACK;
//   Paint grandparent to RED.
//        [G]             <G>
//        / \             / \
//      <P> <U>  ====>  [P] [U]
//      /               /
//    <N>             <N>
// clang-format on
assert(node->parent->isRed());
node->parent->color = Node::BLACK;
node->uncle()->color = Node::BLACK;
node->grandParent()->color = Node::RED;
maintainAfterInsert(node->grandParent());
return;
```

Case 5

当前节点 N 与父节点 P 的方向相反（即 N 节点为右子节点且父节点为左子节点，或 N 节点为左子节点且父节点为右子节点。类似 AVL 树中 LR 和 RL 的情况）。根据性质 4，若 N 为新插入节点，U 则为 NIL 黑色节点，否则为普通黑色节点。

该种情况无法直接进行维护，需要通过旋转操作将子树结构调整为 Case 6 的初始状态并进入 Case 6 进行后续维护。

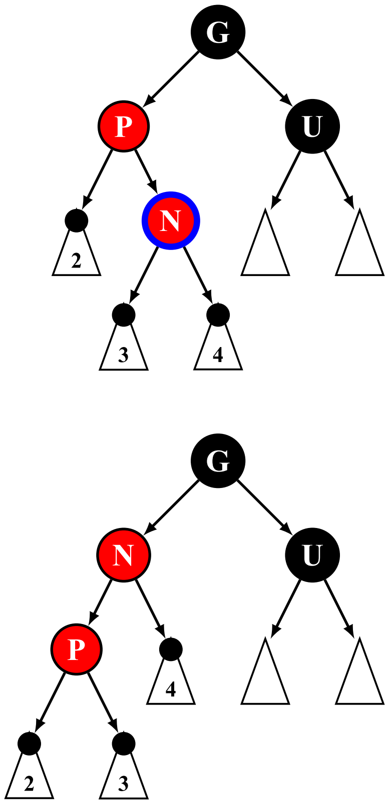


图 10.103 rbtree-insert-case5

"实现"

```
// clang-format off
// Case 5: Current node is the opposite direction as parent
//   Step 1. If node is a LEFT child, perform l-rotate to parent;
//           If node is a RIGHT child, perform r-rotate to parent.
//   Step 2. Goto Case 6.
//      [G]                 [G]
//      / \    rotate(P)    / \
//    <P> [U]  ========>  <N> [U]
//      \                  /
//      <N>              <P>
// clang-format on

// Step 1: Rotation
NodePtr parent = node->parent;
if (node->direction() == Direction::LEFT) {
  rotateRight(node->parent);
} else /* node->direction() == Direction::RIGHT */ {
  rotateLeft(node->parent);
}
node = parent;
// Step 2: vvv
```

Case 6

当前节点 N 与父节点 P 的方向相同（即 N 节点为右子节点且父节点为右子节点，或 N 节点为左子节点且父节点为右子节点。类似 AVL 树中 LL 和 RR 的情况）。根据性质 4，若 N 为新插入节点，U 则为 NIL 黑色节点，否则为普通黑色节点。

在这种情况下，若想在不改变结构的情况下使得子树满足性质 3，则需将 G 染成红色，将 P 染成黑色。但若这样维护的话则性质 4 被打破，且无法保证在 G 节点的父节点上性质 3 是否成立。而选择通过旋转改变子树结构后再进行重新染色即可同时满足性质 3 和 4。

因此，这种情况的维护需要：

1. 若 N 为左子节点则右旋祖父节点 G，否则左旋祖父节点 G.（该操作使得旋转过后 P - N 这条路径上的黑色节点个数比 P - G - U 这条路径上少 1，暂时打破性质 4）。
2. 重新染色，将 P 染黑，将 G 染红，同时满足了性质 3 和 4。

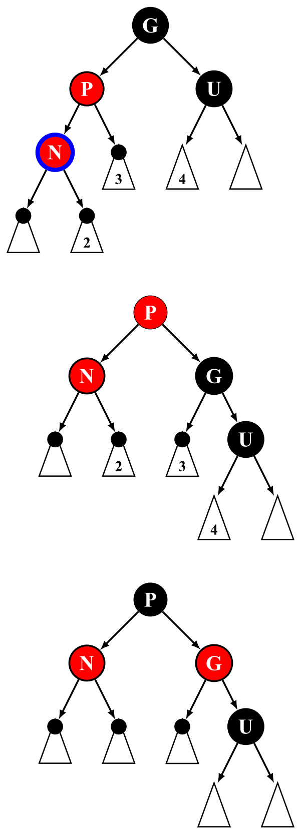


图 10.104 rbtree-insert-case6

"实现"

```
// clang-format off
// Case 6: Current node is the same direction as parent
//   Step 1. If node is a LEFT child, perform r-rotate to grandparent;
//           If node is a RIGHT child, perform l-rotate to grandparent.
//   Step 2. Paint parent (before rotate) to BLACK;
//           Paint grandparent (before rotate) to RED.
//        [G]                 <P>               [P]
//        / \    rotate(G)    / \    repaint    / \
//      <P> [U]  ========>  <N> [G]  ======>  <N> <G>
//      /                          \                \
```

```
//    <N>                        [U]                [U]
// clang-format on
assert(node->grandParent() != nullptr);

// Step 1
if (node->parent->direction() == Direction::LEFT) {
  rotateRight(node->grandParent());
} else {
  rotateLeft(node->grandParent());
}

// Step 2
node->parent->color = Node::BLACK;
node->sibling()->color = Node::RED;

return;
```

删除操作

红黑树的删除操作情况繁多，较为复杂。这部分内容主要通过代码示例来进行讲解。大多数红黑树的实现选择将节点的删除以及删除之后的维护写在同一个函数或逻辑块中（例如 Wikipedia[5] 给出的 代码示例[6]，linux 内核中的 rbtree[7] 以及 GNU libstdc++ 中的 `std::_Rb_tree`[8] 都使用了类似的写法）。笔者则认为这种实现方式并不利于对算法本身的理解，因此，本文给出的示例代码参考了 OpenJDK 中 `TreeMap`[9] 的实现，将删除操作本身与删除后的平衡维护操作解耦成两个独立的函数，并对这两部分的逻辑单独进行分析。

Case 0

若待删除节点为根节点的话，直接删除即可，这里不将其算作删除操作的 3 种基本情况中。

Case 1

若待删除节点 N 既有左子节点又有右子节点，则需找到它的前驱或后继节点进行替换（仅替换数据，不改变节点颜色和内部引用关系），则后续操作中只需要将后继节点删除即可。这部分操作与普通 BST 完全相同，在此不再过多赘述。

注：这里选择的前驱或后继节点保证不会是一个既有非 NIL 左子节点又有非 NIL 右子节点的节点。这里拿后继节点进行简单说明：若该节点包含非空左子节点，则该节点并非是 N 节点右子树上键值最小的节点，与后继节点的性质矛盾，因此后继节点的左子节点必须为 NIL。

" 实现"

```
// clang-format off
// Case 1: If the node is strictly internal
//   Step 1. Find the successor S with the smallest key
//           and its parent P on the right subtree.
//   Step 2. Swap the data (key and value) of S and N,
//           S is the node that will be deleted in place of N.
//   Step 3. N = S, goto Case 2, 3
//     |                    |
//     N                    S
//    / \                  / \
//   L  ..   swap(N, S)   L  ..
//       |   =========>       |
//       P                    P
//      / \                  / \
//     S  ..                N  ..
```

```
// clang-format on

// Step 1
NodePtr successor = node->right;
NodePtr parent = node;
while (successor->left != nullptr) {
  parent = successor;
  successor = parent->left;
}
// Step 2
swapNode(node, successor);
maintainRelationship(parent);
// Step 3: vvv
```

Case 2

待删除节点为叶子节点，若该节点为红色，直接删除即可，删除后仍能保证红黑树的 4 条性质。若为黑色，删除后性质 4 被打破，需要重新进行维护。

注：由于维护操作不会改变待删除节点的任何结构和数据，因此此处的代码示例中为了实现方便起见选择先进行维护，再解引用相关节点。

" 实现"

```
// clang-format off
// Case 2: Current node is a leaf
//   Step 1. Unlink and remove it.
//   Step 2. If N is BLACK, maintain N;
//           If N is RED, do nothing.
// clang-format on
// The maintain operation won't change the node itself,
//  so we can perform maintain operation before unlink the node.
if (node->isBlack()) {
  maintainAfterRemove(node);
}
if (node->direction() == Direction::LEFT) {
  node->parent->left = nullptr;
} else /* node->direction() == Direction::RIGHT */ {
  node->parent->right = nullptr;
}
```

Case 3

待删除节点 N 有且仅有一个非 NIL 子节点，则子节点 S 一定为红色。因为如果子节点 S 为黑色，则 S 的黑深度和待删除结点的黑深度不同，违反性质 4。由于子节点 S 为红色，则待删除节点 N 为黑色，直接使用子节点 S 替代 N 并将其染黑后即可满足性质 4。

" 实现"

```
// Case 3: Current node has a single left or right child
//   Step 1. Replace N with its child
//   Step 2. Paint N to BLACK
NodePtr parent = node->parent;
NodePtr replacement = (node->left != nullptr ? node->left : node->right);

switch (node->direction()) {
```

```
  case Direction::ROOT:
    this->root = replacement;
    break;
  case Direction::LEFT:
    parent->left = replacement;
    break;
  case Direction::RIGHT:
    parent->right = replacement;
    break;
}

if (!node->isRoot()) {
  replacement->parent = parent;
}

node->color = Node::BLACK;
```

删除后的平衡维护

Case 1

兄弟节点 (sibling node) S 为红色，则父节点 P 和侄节点 (nephew node) C 和 D 必为黑色（否则违反性质 3）。与插入后维护操作的 Case 5 类似，这种情况下无法通过直接的旋转或染色操作使其满足所有性质，因此通过前置操作优先保证部分结构满足性质，再进行后续维护即可。

这种情况的维护需要：

1. 若待删除节点 N 为左子节点，左旋 P; 若为右子节点，右旋 P。
2. 将 S 染黑，P 染红（保证 S 节点的父节点满足性质 4）。
3. 此时只需根据结构对以当前 P 节点为根的子树进行维护即可（无需再考虑旋转染色后的 S 和 D 节点）。

" 实现"

```
// clang-format off

// Case 1: Sibling is RED, parent and nephews must be BLACK
//   Step 1. If N is a left child, left rotate P;
//           If N is a right child, right rotate P.
//   Step 2. Paint S to BLACK, P to RED
//   Step 3. Goto Case 2, 3, 4, 5
//      [P]                  <S>               [S]
//      / \    l-rotate(P)   / \    repaint    / \
//    [N] <S>  ==========>  [P] [D]  ======>  <P> [D]
//        / \               / \               / \
//      [C] [D]           [N] [C]           [N] [C]
// clang-format on
ConstNodePtr parent = node->parent;
assert(parent != nullptr && parent->isBlack());
assert(sibling->left != nullptr && sibling->left->isBlack());
assert(sibling->right != nullptr && sibling->right->isBlack());
// Step 1
rotateSameDirection(node->parent, direction);
// Step 2
sibling->color = Node::BLACK;
parent->color = Node::RED;
```

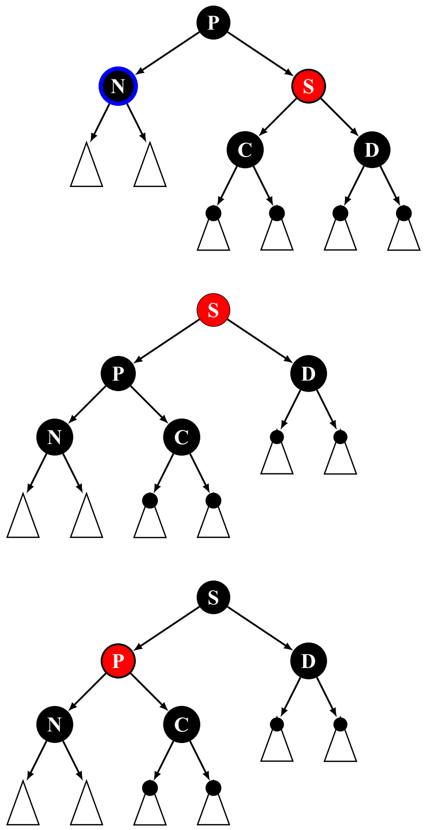


图 10.105 rbtree-remove-case1

```
// Update sibling after rotation
sibling = node->sibling();
// Step 3: vvv
```

Case 2

兄弟节点 S 和侄节点 C, D 均为黑色，父节点 P 为红色。此时只需将 S 染红，将 P 染黑即可满足性质 3 和 4。

" 实现"

```
// clang-format off
// Case 2: Sibling and nephews are BLACK, parent is RED
//   Swap the color of P and S
//      <P>             [P]
//      / \             / \
//    [N] [S]  ====>  [N] <S>
//        / \             / \
//      [C] [D]         [C] [D]
// clang-format on
sibling->color = Node::RED;
node->parent->color = Node::BLACK;
return;
```

Case 3

兄弟节点 S，父节点 P 以及侄节点 C, D 均为黑色。

此时也无法通过一步操作同时满足性质 3 和 4，因此选择将 S 染红，优先满足局部性质 4 的成立，再递归维护 P 节点根据上部结构进行后续维护。

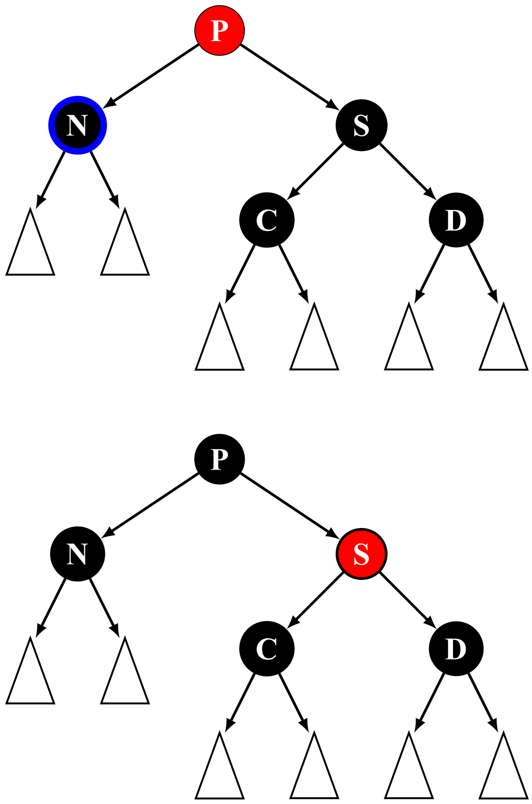


图 10.106 rbtree-remove-case2

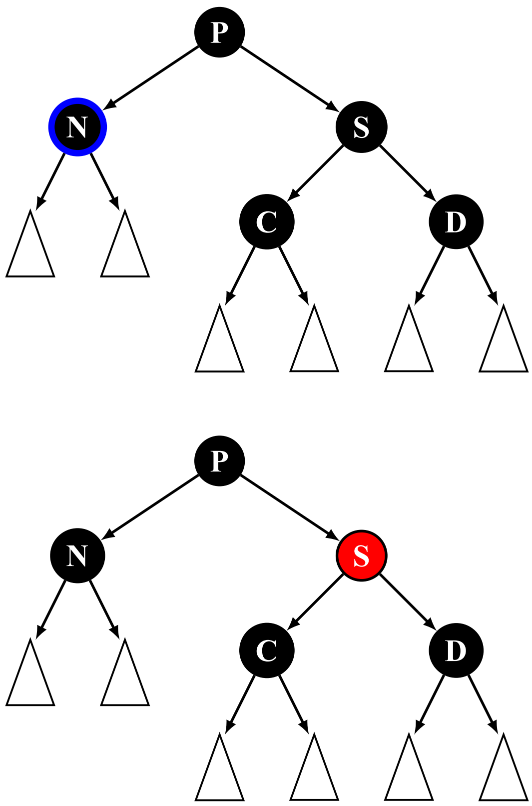


图 10.107 rbtree-remove-case3

" 实现"

```
// clang-format off
// Case 3: Sibling, parent and nephews are all black
//   Step 1. Paint S to RED
//   Step 2. Recursively maintain P
//      [P]             [P]
//      / \             / \
//    [N] [S]  ====>  [N] <S>
//        / \             / \
//      [C] [D]         [C] [D]
// clang-format on
sibling->color = Node::RED;
maintainAfterRemove(node->parent);
return;
```

Case 4

兄弟节点是黑色，且与 N 同向的侄节点 C（由于没有固定中文翻译，下文还是统一将其称作 close nephew）为红色，与 N 反向的侄节点 D（同理，下文称作 distant nephew）为黑色，父节点既可为红色又可为黑色。

此时同样无法通过一步操作使其满足性质，因此优先选择将其转变为 Case 5 的状态利用后续 Case 5 的维护过程进行修正。

该过程分为三步：

1. 若 N 为左子节点，右旋 P，否则左旋 P。
2. 将节点 S 染红，将节点 C 染黑。
3. 此时已满足 Case 5 的条件，进入 Case 5 完成后续维护。

" 实现"

```
// clang-format off
// Case 4: Sibling is BLACK, close nephew is RED,
//         distant nephew is BLACK
//   Step 1. If N is a left child, right rotate P;
//           If N is a right child, left rotate P.
//   Step 2. Swap the color of close nephew and sibling
//   Step 3. Goto case 5
//                            {P}                {P}
//      {P}                   / \                / \
//      / \    r-rotate(S)  [N] <C>   repaint  [N] [C]
//    [N] [S]  ==========>        \   ======>        \
//        / \                     [S]                <S>
//      <C> [D]                     \                  \
//                                  [D]                [D]
// clang-format on

// Step 1
rotateOppositeDirection(sibling, direction);
// Step 2
closeNephew->color = Node::BLACK;
sibling->color = Node::RED;
// Update sibling and nephews after rotation
sibling = node->sibling();
```

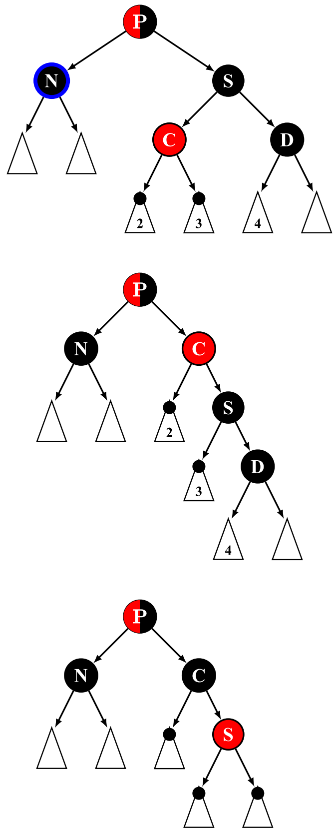


图 10.108 rbtree-remove-case4

```
closeNephew = direction == Direction::LEFT ? sibling->left : sibling->right;
distantNephew = direction == Direction::LEFT ? sibling->right : sibling->left;
// Step 3: vvv
```

Case 5

兄弟节点是黑色，且 close nephew 节点 C 为黑色，distant nephew 节点 D 为红色，父节点既可为红色又可为黑色。此时性质 4 无法满足，通过旋转操作使得黑色节点 S 变为该子树的根节点再进行染色即可满足性质 4。具体步骤如下：

1. 若 N 为左子节点，左旋 P，反之右旋 P。
2. 交换父节点 P 和兄弟节点 S 的颜色，此时性质 3 可能被打破。
3. 将 distant nephew 节点 D 染黑，同时保证了性质 3 和 4。

" 实现"

```
// clang-format off
// Case 5: Sibling is BLACK, close nephew is BLACK,
//         distant nephew is RED
//   Step 1. If N is a left child, left rotate P;
//           If N is a right child, right rotate P.
//   Step 2. Swap the color of parent and sibling.
//   Step 3. Paint distant nephew D to BLACK.
//      {P}                   [S]               {S}
//      / \     l-rotate(P)    / \     repaint    / \
//    [N] [S]   ==========>  {P} <D>   ======>  [P] [D]
```

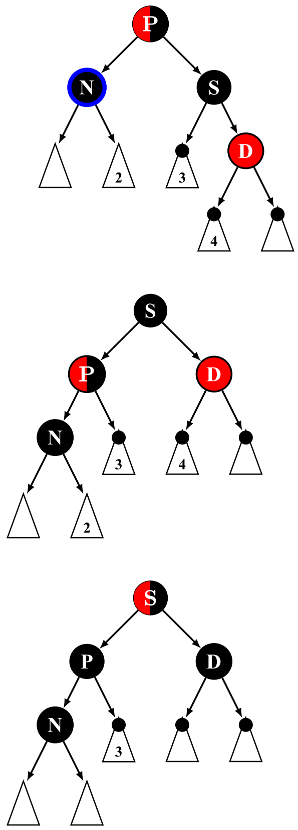


图 10.109 rbtree-remove-case5

```
//          / \                  / \                  / \
//        [C] <D>              [N] [C]              [N] [C]
// clang-format on
assert(closeNephew == nullptr || closeNephew->isBlack());
assert(distantNephew->isRed());
// Step 1
rotateSameDirection(node->parent, direction);
// Step 2
sibling->color = node->parent->color;
node->parent->color = Node::BLACK;
// Step 3
distantNephew->color = Node::BLACK;
return;
```

红黑树与 4 阶 B 树（2-3-4 树）的关系

红黑树是由德国计算机科学家 Rudolf Bayer[10] 在 1972 年从 B 树上改进过来的，红黑树在当时被称作"symmetric binary B-tree"，因此与 B 树有众多相似之处。比如红黑树与 4 阶 B 树每个簇（对于红黑树来说一个簇是一个非 NIL 黑色节点和它的两个子节点，对 B 树来说一个簇就是一个节点）的最大容量为 3 且最小填充量均为 $\frac{1}{3}$。因此我们甚至可以说红黑树与 4 阶 B 树（2-3-4 树）在结构上是等价的。

对这方面内容感兴趣的可以观看 从 2-3-4 树的角度学习理解红黑树 （ 视频 ）[11] 进行学习。

虽然二者在结构上是等价的，但这并不意味这二者可以互相取代或者在所有情况下都可以互换使用。最显然的例子就是数据库的索引，由于 B 树不存在旋转操作，因此其所有节点的存储位置都是可以被确定的，这种结构对于不区分堆栈的磁盘来说显然比红黑树动态分配节点存储空间要更加合适。另外一点就是由于 B 树/B+ 树内储存的数据都是连续的，对于有着大量连续查询需求的数据库来说更加友好。而对于小数据量随机插入/查询的需求，由于 B 树的每个节点都存储了若干条记录，因此发生 cache miss 时就需要将整个节点的所有数据读入缓存中，在这些情况下

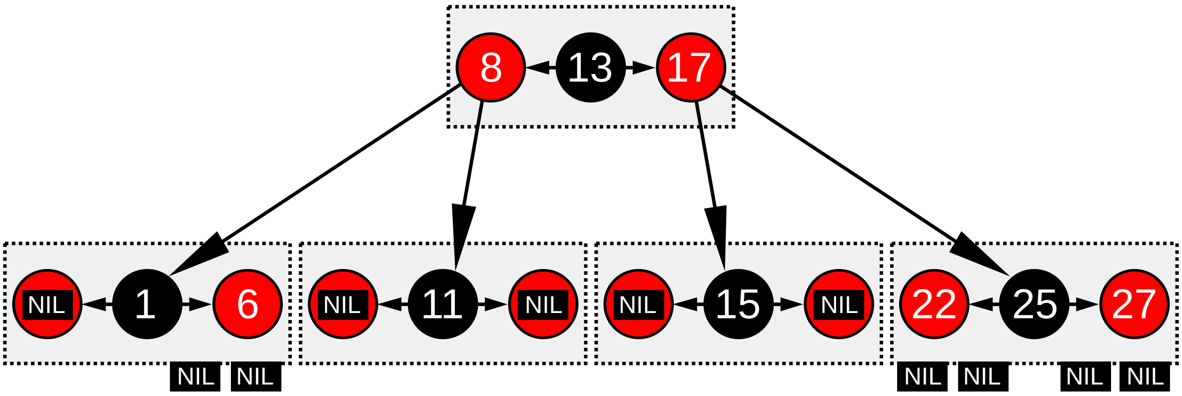


图 10.110 rbtree-btree-analogy

BST（红黑树，AVL，Splay 等）则反而会优与 B 树/B+ 树。对这方面内容感兴趣的读者可以去阅读一下 为什么 rust 中的 Map 使用的是 B 树而不是像其他主流语言一样使用红黑树[12]。

红黑树在实际工程项目中的使用

由于红黑树是目前主流工业界综合效率最高的内存型平衡树，其在实际的工程项目中有着广泛的使用，这里列举几个实际的使用案例并给出相应的源码链接，以便读者进行对比学习。

Linux

源码：

- `linux/lib/rbtree.c`[13]

Linux 中的红黑树所有操作均使用循环迭代进行实现，保证效率的同时又增加了大量的注释来保证代码可读性，十分建议读者阅读学习。Linux 内核中的红黑树使用非常广泛，这里仅列举几个经典案例。

CFS 非实时任务调度[14]

Linux 的稳定内核版本在 2.6.24 之后，使用了新的调度程序 CFS，所有非实时可运行进程都以虚拟运行时间为键值用一棵红黑树进行维护，以完成更公平高效地调度所有任务。CFS 弃用 active/expired 数组和动态计算优先级，不再跟踪任务的睡眠时间和区别是否交互任务，而是在调度中采用基于时间计算键值的红黑树来选取下一个任务，根据所有任务占用 CPU 时间的状态来确定调度任务优先级。

epoll[15]

epoll 全称 event poll，是 Linux 内核实现 IO 多路复用 (IO multiplexing) 的一个实现，是原先 poll/select 的改进版。Linux 中 epoll 的实现选择使用红黑树来储存文件描述符。

Nginx

源码：

- `nginx/src/core/ngx_rbtree.h`[16]
- `nginx/src/core/ngx_rbtree.c`[17]

nginx 中的用户态定时器是通过红黑树实现的。在 nginx 中，所有 timer 节点都由一棵红黑树进行维护，在 worker 进程的每一次循环中都会调用 `ngx_process_events_and_timers` 函数，在该函数中就会调用处理定时器的函数 `ngx_event_expire_timers`，每次该函数都不断的从红黑树中取出时间值最小的，查看他们是否已经超时，然后执行他们的函数，直到取出的节点的时间没有超时为止。

关于 nginx 中红黑树的源码分析公开资源很多，读者可以自行查找学习。

STL

源码：

- GNU libstdc++
  - `libstdc++-v3/include/bits/stl_tree.h`[8]
  - `libstdc++-v3/src/c++98/tree.cc`[18]
- LLVM libcxx
  - `libcxx/include/__tree`[19]
- Microsoft STL
  - `stl/inc/xtree`[20]

大多数 STL 中的 `std::map` 和 `std::set` 的内部数据结构就是一棵红黑树（例如上面提到的这些）。不过值得注意的是，这些红黑树（包括可能有读者用过的 `std::_Rb_tree`）都不是 C++ 标准，虽然部分竞赛（例如 NOIP）并未明令禁止这类数据结构，但还是应当注意这类标准库中的非标准实现不应该在工程项目中直接使用。

由于 STL 的特殊性，其中大多数实现的代码可读性都不高，因此并不建议读者使用 STL 学习红黑树。

OpenJDK

源码：

- `java.util.TreeMap<K, V>`[9]
- `java.util.TreeSet<K, V>`[21]
- `java.util.HashMap<K, V>`[22]

JDK 中的 `TreeMap` 和 `TreeSet` 都是使用红黑树作为底层数据结构的。同时在 JDK 1.8 之后 `HashMap` 内部哈希表中每个表项的链表长度超过 8 时也会自动转变为红黑树以提升查找效率。

笔者认为，JDK 中的红黑树实现是主流红黑树实现中可读性最高的，本文提供的参考代码很大程度上借鉴了 JDK 中 `TreeMap` 的实现，因此也建议读者阅读学习 JDK 中 `TreeMap` 的实现。

参考代码

下面的代码是用红黑树实现的 `Map`，即有序不可重映射：

" 完整代码"

```cpp
/**
 * @file RBTreeMap.hpp
 * @brief An RBTree-based map implementation
 * @details The map is sorted according to the natural ordering of its
 *  keys or by a {@code Compare} function provided; This implementation
 *  provides guaranteed log(n) time cost for the contains, get, insert
 *  and remove operations.
 * @author [r.ivance](https://github.com/RIvance)
 */

#ifndef RBTREE_MAP_HPP
#define RBTREE_MAP_HPP

#include <cassert>
#include <cstddef>
#include <cstdint>
#include <functional>
```

```cpp
#include <memory>
#include <stack>
#include <utility>
#include <vector>

/**
 * An RBTree-based map implementation
 * https://en.wikipedia.org/wiki/Red–black_tree
 *
 * A red–black tree (RBTree) is a kind of self-balancing binary search tree.
 * Each node stores an extra field representing "color" (RED or BLACK), used
 * to ensure that the tree remains balanced during insertions and deletions.
 *
 * In addition to the requirements imposed on a binary search tree the following
 * must be satisfied by a red–black tree:
 *
 *  1. Every node is either RED or BLACK.
 *  2. All NIL nodes (`nullptr` in this implementation) are considered BLACK.
 *  3. A RED node does not have a RED child.
 *  4. Every path from a given node to any of its descendant NIL nodes goes
 * through the same number of BLACK nodes.
 *
 * @tparam Key the type of keys maintained by this map
 * @tparam Value the type of mapped values
 * @tparam Compare the compare function
 */
template <typename Key, typename Value, typename Compare = std::less<Key> >
class RBTreeMap {
 private:
  using USize = size_t;

  Compare compare = Compare();

 public:
  struct Entry {
    Key key;
    Value value;

    bool operator==(const Entry &rhs) const noexcept {
      return this->key == rhs.key && this->value == rhs.value;
    }

    bool operator!=(const Entry &rhs) const noexcept {
      return this->key != rhs.key || this->value != rhs.value;
    }
  };

 private:
  struct Node {
    using Ptr = std::shared_ptr<Node>;
    using Provider = const std::function<Ptr(void)> &;
    using Consumer = const std::function<void(const Ptr &)> &;

    enum { RED, BLACK } color = RED;
```

```cpp
enum Direction { LEFT = -1, ROOT = 0, RIGHT = 1 };

Key key;
Value value{};

Ptr parent = nullptr;
Ptr left = nullptr;
Ptr right = nullptr;

explicit Node(Key k) : key(std::move(k)) {}

explicit Node(Key k, Value v) : key(std::move(k)), value(std::move(v)) {}

~Node() = default;

inline bool isLeaf() const noexcept {
  return this->left == nullptr && this->right == nullptr;
}

inline bool isRoot() const noexcept { return this->parent == nullptr; }

inline bool isRed() const noexcept { return this->color == RED; }

inline bool isBlack() const noexcept { return this->color == BLACK; }

inline Direction direction() const noexcept {
  if (this->parent != nullptr) {
    if (this == this->parent->left.get()) {
      return Direction::LEFT;
    } else {
      return Direction::RIGHT;
    }
  } else {
    return Direction::ROOT;
  }
}

inline Ptr &sibling() const noexcept {
  assert(!this->isRoot());
  if (this->direction() == LEFT) {
    return this->parent->right;
  } else {
    return this->parent->left;
  }
}

inline bool hasSibling() const noexcept {
  return !this->isRoot() && this->sibling() != nullptr;
}

inline Ptr &uncle() const noexcept {
  assert(this->parent != nullptr);
  return parent->sibling();
```

```cpp
    }

    inline bool hasUncle() const noexcept {
      return !this->isRoot() && this->parent->hasSibling();
    }

    inline Ptr &grandParent() const noexcept {
      assert(this->parent != nullptr);
      return this->parent->parent;
    }

    inline bool hasGrandParent() const noexcept {
      return !this->isRoot() && this->parent->parent != nullptr;
    }

    inline void release() noexcept {
      // avoid memory leak caused by circular reference
      this->parent = nullptr;
      if (this->left != nullptr) {
        this->left->release();
      }
      if (this->right != nullptr) {
        this->right->release();
      }
    }

    inline Entry entry() const { return Entry{key, value}; }

    static Ptr from(const Key &k) { return std::make_shared<Node>(Node(k)); }

    static Ptr from(const Key &k, const Value &v) {
      return std::make_shared<Node>(Node(k, v));
    }
  };

  using NodePtr = typename Node::Ptr;
  using ConstNodePtr = const NodePtr &;
  using Direction = typename Node::Direction;
  using NodeProvider = typename Node::Provider;
  using NodeConsumer = typename Node::Consumer;

  NodePtr root = nullptr;
  USize count = 0;

  using K = const Key &;
  using V = const Value &;

 public:
  using EntryList = std::vector<Entry>;
  using KeyValueConsumer = const std::function<void(K, V)> &;
  using MutKeyValueConsumer = const std::function<void(K, Value &)> &;
  using KeyValueFilter = const std::function<bool(K, V)> &;

  class NoSuchMappingException : protected std::exception {
```

```
  private:
   const char *message;

  public:
   explicit NoSuchMappingException(const char *msg) : message(msg) {}

   const char *what() const noexcept override { return message; }
 };

 RBTreeMap() noexcept = default;

 ~RBTreeMap() noexcept {
   // Unlinking circular references to avoid memory leak
   this->clear();
 }

 /**
  * Returns the number of entries in this map.
  * @return size_t
  */
 inline USize size() const noexcept { return this->count; }

 /**
  * Returns true if this collection contains no elements.
  * @return bool
  */
 inline bool empty() const noexcept { return this->count == 0; }

 /**
  * Removes all of the elements from this map.
  */
 void clear() noexcept {
   // Unlinking circular references to avoid memory leak
   if (this->root != nullptr) {
     this->root->release();
     this->root = nullptr;
   }
   this->count = 0;
 }

 /**
  * Returns the value to which the specified key is mapped; If this map
  * contains no mapping for the key, a {@code NoSuchMappingException} will
  * be thrown.
  * @param key
  * @return RBTreeMap<Key, Value>::Value
  * @throws NoSuchMappingException
  */
 Value get(K key) const {
   if (this->root == nullptr) {
     throw NoSuchMappingException("Invalid key");
   } else {
     NodePtr node = this->getNode(this->root, key);
     if (node != nullptr) {
```

```
      return node->value;
    } else {
      throw NoSuchMappingException("Invalid key");
    }
  }
}

/**
 * Returns the value to which the specified key is mapped; If this map
 * contains no mapping for the key, a new mapping with a default value
 * will be inserted.
 * @param key
 * @return RBTreeMap<Key, Value>::Value &
 */
Value &getOrDefault(K key) {
  if (this->root == nullptr) {
    this->root = Node::from(key);
    this->root->color = Node::BLACK;
    this->count += 1;
    return this->root->value;
  } else {
    return this
        ->getNodeOrProvide(this->root, key,
                           [&key]() { return Node::from(key); })
        ->value;
  }
}

/**
 * Returns true if this map contains a mapping for the specified key.
 * @param key
 * @return bool
 */
bool contains(K key) const {
  return this->getNode(this->root, key) != nullptr;
}

/**
 * Associates the specified value with the specified key in this map.
 * @param key
 * @param value
 */
void insert(K key, V value) {
  if (this->root == nullptr) {
    this->root = Node::from(key, value);
    this->root->color = Node::BLACK;
    this->count += 1;
  } else {
    this->insert(this->root, key, value);
  }
}

/**
 * If the specified key is not already associated with a value, associates
```

```
 * it with the given value and returns true, else returns false.
 * @param key
 * @param value
 * @return bool
 */
bool insertIfAbsent(K key, V value) {
  USize sizeBeforeInsertion = this->size();
  if (this->root == nullptr) {
    this->root = Node::from(key, value);
    this->root->color = Node::BLACK;
    this->count += 1;
  } else {
    this->insert(this->root, key, value, false);
  }
  return this->size() > sizeBeforeInsertion;
}

/**
 * If the specified key is not already associated with a value, associates
 * it with the given value and returns the value, else returns the associated
 * value.
 * @param key
 * @param value
 * @return RBTreeMap<Key, Value>::Value &
 */
Value &getOrInsert(K key, V value) {
  if (this->root == nullptr) {
    this->root = Node::from(key, value);
    this->root->color = Node::BLACK;
    this->count += 1;
    return root->value;
  } else {
    NodePtr node = getNodeOrProvide(this->root, key,
                                    [&]() { return Node::from(key, value); });
    return node->value;
  }
}

Value operator[](K key) const { return this->get(key); }

Value &operator[](K key) { return this->getOrDefault(key); }

/**
 * Removes the mapping for a key from this map if it is present;
 * Returns true if the mapping is present else returns false
 * @param key the key of the mapping
 * @return bool
 */
bool remove(K key) {
  if (this->root == nullptr) {
    return false;
  } else {
    return this->remove(this->root, key, [](ConstNodePtr) {});
  }
```

```
  }

  /**
   * Removes the mapping for a key from this map if it is present and returns
   * the value which is mapped to the key; If this map contains no mapping for
   * the key, a {@code NoSuchMappingException} will be thrown.
   * @param key
   * @return RBTreeMap<Key, Value>::Value
   * @throws NoSuchMappingException
   */
  Value getAndRemove(K key) {
    Value result;
    NodeConsumer action = [&](ConstNodePtr node) { result = node->value; };

    if (root == nullptr) {
      throw NoSuchMappingException("Invalid key");
    } else {
      if (remove(this->root, key, action)) {
        return result;
      } else {
        throw NoSuchMappingException("Invalid key");
      }
    }
  }

  /**
   * Gets the entry corresponding to the specified key; if no such entry
   * exists, returns the entry for the least key greater than the specified
   * key; if no such entry exists (i.e., the greatest key in the Tree is less
   * than the specified key), a {@code NoSuchMappingException} will be thrown.
   * @param key
   * @return RBTreeMap<Key, Value>::Entry
   * @throws NoSuchMappingException
   */
  Entry getCeilingEntry(K key) const {
    if (this->root == nullptr) {
      throw NoSuchMappingException("No ceiling entry in this map");
    }

    NodePtr node = this->root;

    while (node != nullptr) {
      if (key == node->key) {
        return node->entry();
      }

      if (compare(key, node->key)) {
        /* key < node->key */
        if (node->left != nullptr) {
          node = node->left;
        } else {
          return node->entry();
        }
      } else {
```

```
        /* key > node->key */
        if (node->right != nullptr) {
          node = node->right;
        } else {
          while (node->direction() == Direction::RIGHT) {
            if (node != nullptr) {
              node = node->parent;
            } else {
              throw NoSuchMappingException(
                  "No ceiling entry exists in this map");
            }
          }
          if (node->parent == nullptr) {
            throw NoSuchMappingException("No ceiling entry exists in this map");
          }
          return node->parent->entry();
        }
      }
    }

    throw NoSuchMappingException("No ceiling entry in this map");
  }

  /**
   * Gets the entry corresponding to the specified key; if no such entry exists,
   * returns the entry for the greatest key less than the specified key;
   * if no such entry exists, a {@code NoSuchMappingException} will be thrown.
   * @param key
   * @return RBTreeMap<Key, Value>::Entry
   * @throws NoSuchMappingException
   */
  Entry getFloorEntry(K key) const {
    if (this->root == nullptr) {
      throw NoSuchMappingException("No floor entry exists in this map");
    }

    NodePtr node = this->root;

    while (node != nullptr) {
      if (key == node->key) {
        return node->entry();
      }

      if (compare(key, node->key)) {
        /* key < node->key */
        if (node->left != nullptr) {
          node = node->left;
        } else {
          while (node->direction() == Direction::LEFT) {
            if (node != nullptr) {
              node = node->parent;
            } else {
              throw NoSuchMappingException("No floor entry exists in this map");
            }
```

```
        }
        if (node->parent == nullptr) {
          throw NoSuchMappingException("No floor entry exists in this map");
        }
        return node->parent->entry();
      }
    } else {
      /* key > node->key */
      if (node->right != nullptr) {
        node = node->right;
      } else {
        return node->entry();
      }
    }
  }

  throw NoSuchMappingException("No floor entry exists in this map");
}

/**
 * Gets the entry for the least key greater than the specified
 * key; if no such entry exists, returns the entry for the least
 * key greater than the specified key; if no such entry exists,
 * a {@code NoSuchMappingException} will be thrown.
 * @param key
 * @return RBTreeMap<Key, Value>::Entry
 * @throws NoSuchMappingException
 */
Entry getHigherEntry(K key) {
  if (this->root == nullptr) {
    throw NoSuchMappingException("No higher entry exists in this map");
  }

  NodePtr node = this->root;

  while (node != nullptr) {
    if (compare(key, node->key)) {
      /* key < node->key */
      if (node->left != nullptr) {
        node = node->left;
      } else {
        return node->entry();
      }
    } else {
      /* key >= node->key */
      if (node->right != nullptr) {
        node = node->right;
      } else {
        while (node->direction() == Direction::RIGHT) {
          if (node != nullptr) {
            node = node->parent;
          } else {
            throw NoSuchMappingException(
                "No higher entry exists in this map");
```

```
          }
        }
        if (node->parent == nullptr) {
          throw NoSuchMappingException("No higher entry exists in this map");
        }
        return node->parent->entry();
      }
    }
  }

  throw NoSuchMappingException("No higher entry exists in this map");
}

/**
 * Returns the entry for the greatest key less than the specified key; if
 * no such entry exists (i.e., the least key in the Tree is greater than
 * the specified key), a {@code NoSuchMappingException} will be thrown.
 * @param key
 * @return RBTreeMap<Key, Value>::Entry
 * @throws NoSuchMappingException
 */
Entry getLowerEntry(K key) const {
  if (this->root == nullptr) {
    throw NoSuchMappingException("No lower entry exists in this map");
  }

  NodePtr node = this->root;

  while (node != nullptr) {
    if (compare(key, node->key) || key == node->key) {
      /* key <= node->key */
      if (node->left != nullptr) {
        node = node->left;
      } else {
        while (node->direction() == Direction::LEFT) {
          if (node != nullptr) {
            node = node->parent;
          } else {
            throw NoSuchMappingException("No lower entry exists in this map");
          }
        }
        if (node->parent == nullptr) {
          throw NoSuchMappingException("No lower entry exists in this map");
        }
        return node->parent->entry();
      }
    } else {
      /* key > node->key */
      if (node->right != nullptr) {
        node = node->right;
      } else {
        return node->entry();
      }
    }
```

```
    }

    throw NoSuchMappingException("No lower entry exists in this map");
  }

  /**
   * Remove all entries that satisfy the filter condition.
   * @param filter
   */
  void removeAll(KeyValueFilter filter) {
    std::vector<Key> keys;
    this->inorderTraversal([&](ConstNodePtr node) {
      if (filter(node->key, node->value)) {
        keys.push_back(node->key);
      }
    });
    for (const Key &key : keys) {
      this->remove(key);
    }
  }

  /**
   * Performs the given action for each key and value entry in this map.
   * The value is immutable for the action.
   * @param action
   */
  void forEach(KeyValueConsumer action) const {
    this->inorderTraversal(
        [&](ConstNodePtr node) { action(node->key, node->value); });
  }

  /**
   * Performs the given action for each key and value entry in this map.
   * The value is mutable for the action.
   * @param action
   */
  void forEachMut(MutKeyValueConsumer action) {
    this->inorderTraversal(
        [&](ConstNodePtr node) { action(node->key, node->value); });
  }

  /**
   * Returns a list containing all of the entries in this map.
   * @return RBTreeMap<Key, Value>::EntryList
   */
  EntryList toEntryList() const {
    EntryList entryList;
    this->inorderTraversal(
        [&](ConstNodePtr node) { entryList.push_back(node->entry()); });
    return entryList;
  }

 private:
  static void maintainRelationship(ConstNodePtr node) {
```

```
    if (node->left != nullptr) {
      node->left->parent = node;
    }
    if (node->right != nullptr) {
      node->right->parent = node;
    }
  }

  static void swapNode(NodePtr &lhs, NodePtr &rhs) {
    std::swap(lhs->key, rhs->key);
    std::swap(lhs->value, rhs->value);
    std::swap(lhs, rhs);
  }

  void rotateLeft(ConstNodePtr node) {
    // clang-format off
    //      |                       |
    //      N                       S
    //     / \     l-rotate(N)     / \
    //    L   S    ==========>    N   R
    //       / \                 / \
    //      M   R               L   M
    assert(node != nullptr && node->right != nullptr);
    // clang-format on
    NodePtr parent = node->parent;
    Direction direction = node->direction();

    NodePtr successor = node->right;
    node->right = successor->left;
    successor->left = node;

    maintainRelationship(node);
    maintainRelationship(successor);

    switch (direction) {
      case Direction::ROOT:
        this->root = successor;
        break;
      case Direction::LEFT:
        parent->left = successor;
        break;
      case Direction::RIGHT:
        parent->right = successor;
        break;
    }

    successor->parent = parent;
  }

  void rotateRight(ConstNodePtr node) {
    // clang-format off
    //        |                   |
    //        N                   S
    //       / \   r-rotate(N)   / \
```

```
    //     S    R  ==========>  L    N
    //    / \                      / \
    //   L    M                   M    R
    assert(node != nullptr && node->left != nullptr);
    // clang-format on

    NodePtr parent = node->parent;
    Direction direction = node->direction();

    NodePtr successor = node->left;
    node->left = successor->right;
    successor->right = node;

    maintainRelationship(node);
    maintainRelationship(successor);

    switch (direction) {
      case Direction::ROOT:
        this->root = successor;
        break;
      case Direction::LEFT:
        parent->left = successor;
        break;
      case Direction::RIGHT:
        parent->right = successor;
        break;
    }

    successor->parent = parent;
  }

  inline void rotateSameDirection(ConstNodePtr node, Direction direction) {
    assert(direction != Direction::ROOT);
    if (direction == Direction::LEFT) {
      rotateLeft(node);
    } else {
      rotateRight(node);
    }
  }

  inline void rotateOppositeDirection(ConstNodePtr node, Direction direction) {
    assert(direction != Direction::ROOT);
    if (direction == Direction::LEFT) {
      rotateRight(node);
    } else {
      rotateLeft(node);
    }
  }

  void maintainAfterInsert(NodePtr node) {
    assert(node != nullptr);

    if (node->isRoot()) {
      // Case 1: Current node is root (RED)
```

```
      //  No need to fix.
      assert(node->isRed());
      return;
    }

    if (node->parent->isBlack()) {
      // Case 2: Parent is BLACK
      //  No need to fix.
      return;
    }

    if (node->parent->isRoot()) {
      // clang-format off
      // Case 3: Parent is root and is RED
      //   Paint parent to BLACK.
      //    <P>         [P]
      //     |   ====>   |
      //    <N>         <N>
      //   p.s.
      //    `<X>` is a RED node;
      //    `[X]` is a BLACK node (or NIL);
      //    `{X}` is either a RED node or a BLACK node;
      // clang-format on
      assert(node->parent->isRed());
      node->parent->color = Node::BLACK;
      return;
    }

    if (node->hasUncle() && node->uncle()->isRed()) {
      // clang-format off
      // Case 4: Both parent and uncle are RED
      //   Paint parent and uncle to BLACK;
      //   Paint grandparent to RED.
      //        [G]             <G>
      //        / \             / \
      //      <P> <U>  ====>  [P] [U]
      //      /               /
      //    <N>             <N>
      // clang-format on
      assert(node->parent->isRed());
      node->parent->color = Node::BLACK;
      node->uncle()->color = Node::BLACK;
      node->grandParent()->color = Node::RED;
      maintainAfterInsert(node->grandParent());
      return;
    }

    if (!node->hasUncle() || node->uncle()->isBlack()) {
      // Case 5 & 6: Parent is RED and Uncle is BLACK
      //   p.s. NIL nodes are also considered BLACK
      assert(!node->isRoot());

      if (node->direction() != node->parent->direction()) {
        // clang-format off
```

```
      // Case 5: Current node is the opposite direction as parent
      //   Step 1. If node is a LEFT child, perform l-rotate to parent;
      //           If node is a RIGHT child, perform r-rotate to parent.
      //   Step 2. Goto Case 6.
      //      [G]                 [G]
      //      / \    rotate(P)    / \
      //    <P> [U]  ========>  <N> [U]
      //      \                  /
      //      <N>              <P>
      // clang-format on

      // Step 1: Rotation
      NodePtr parent = node->parent;
      if (node->direction() == Direction::LEFT) {
        rotateRight(node->parent);
      } else /* node->direction() == Direction::RIGHT */ {
        rotateLeft(node->parent);
      }
      node = parent;
      // Step 2: vvv
    }

    // clang-format off
    // Case 6: Current node is the same direction as parent
    //   Step 1. If node is a LEFT child, perform r-rotate to grandparent;
    //           If node is a RIGHT child, perform l-rotate to grandparent.
    //   Step 2. Paint parent (before rotate) to BLACK;
    //           Paint grandparent (before rotate) to RED.
    //        [G]                 <P>               [P]
    //        / \    rotate(G)    / \    repaint    / \
    //      <P> [U]  ========>  <N> [G]  ======>  <N> <G>
    //      /                         \                 \
    //    <N>                         [U]               [U]
    // clang-format on

    assert(node->grandParent() != nullptr);

    // Step 1
    if (node->parent->direction() == Direction::LEFT) {
      rotateRight(node->grandParent());
    } else {
      rotateLeft(node->grandParent());
    }

    // Step 2
    node->parent->color = Node::BLACK;
    node->sibling()->color = Node::RED;

    return;
  }
}

NodePtr getNodeOrProvide(NodePtr &node, K key, NodeProvider provide) {
  assert(node != nullptr);
```

```
    if (key == node->key) {
      return node;
    }

    assert(key != node->key);

    NodePtr result;

    if (compare(key, node->key)) {
      /* key < node->key */
      if (node->left == nullptr) {
        result = node->left = provide();
        node->left->parent = node;
        maintainAfterInsert(node->left);
        this->count += 1;
      } else {
        result = getNodeOrProvide(node->left, key, provide);
      }
    } else {
      /* key > node->key */
      if (node->right == nullptr) {
        result = node->right = provide();
        node->right->parent = node;
        maintainAfterInsert(node->right);
        this->count += 1;
      } else {
        result = getNodeOrProvide(node->right, key, provide);
      }
    }

    return result;
  }

  NodePtr getNode(ConstNodePtr node, K key) const {
    assert(node != nullptr);

    if (key == node->key) {
      return node;
    }

    if (compare(key, node->key)) {
      /* key < node->key */
      return node->left == nullptr ? nullptr : getNode(node->left, key);
    } else {
      /* key > node->key */
      return node->right == nullptr ? nullptr : getNode(node->right, key);
    }
  }

  void insert(NodePtr &node, K key, V value, bool replace = true) {
    assert(node != nullptr);

    if (key == node->key) {
```

```
      if (replace) {
        node->value = value;
      }
      return;
    }

    assert(key != node->key);

    if (compare(key, node->key)) {
      /* key < node->key */
      if (node->left == nullptr) {
        node->left = Node::from(key, value);
        node->left->parent = node;
        maintainAfterInsert(node->left);
        this->count += 1;
      } else {
        insert(node->left, key, value, replace);
      }
    } else {
      /* key > node->key */
      if (node->right == nullptr) {
        node->right = Node::from(key, value);
        node->right->parent = node;
        maintainAfterInsert(node->right);
        this->count += 1;
      } else {
        insert(node->right, key, value, replace);
      }
    }
  }

  void maintainAfterRemove(ConstNodePtr node) {
    if (node->isRoot()) {
      return;
    }

    assert(node->isBlack() && node->hasSibling());

    Direction direction = node->direction();

    NodePtr sibling = node->sibling();
    if (sibling->isRed()) {
      // clang-format off
      // Case 1: Sibling is RED, parent and nephews must be BLACK
      //   Step 1. If N is a left child, left rotate P;
      //           If N is a right child, right rotate P.
      //   Step 2. Paint S to BLACK, P to RED
      //   Step 3. Goto Case 2, 3, 4, 5
      //      [P]                   <S>               [S]
      //      / \    l-rotate(P)    / \    repaint    / \
      //    [N] <S>  ==========>  [P] [D]  ======>  <P> [D]
      //        / \               / \               / \
      //      [C] [D]           [N] [C]           [N] [C]
      // clang-format on
```

```
      ConstNodePtr parent = node->parent;
      assert(parent != nullptr && parent->isBlack());
      assert(sibling->left != nullptr && sibling->left->isBlack());
      assert(sibling->right != nullptr && sibling->right->isBlack());
      // Step 1
      rotateSameDirection(node->parent, direction);
      // Step 2
      sibling->color = Node::BLACK;
      parent->color = Node::RED;
      // Update sibling after rotation
      sibling = node->sibling();
      // Step 3: vvv
    }

    NodePtr closeNephew =
        direction == Direction::LEFT ? sibling->left : sibling->right;
    NodePtr distantNephew =
        direction == Direction::LEFT ? sibling->right : sibling->left;

    bool closeNephewIsBlack = closeNephew == nullptr || closeNephew->isBlack();
    bool distantNephewIsBlack =
        distantNephew == nullptr || distantNephew->isBlack();

    assert(sibling->isBlack());

    if (closeNephewIsBlack && distantNephewIsBlack) {
      if (node->parent->isRed()) {
        // clang-format off
        // Case 2: Sibling and nephews are BLACK, parent is RED
        //   Swap the color of P and S
        //      <P>             [P]
        //      / \             / \
        //    [N] [S]  ====>  [N] <S>
        //        / \             / \
        //      [C] [D]         [C] [D]
        // clang-format on
        sibling->color = Node::RED;
        node->parent->color = Node::BLACK;
        return;
      } else {
        // clang-format off
        // Case 3: Sibling, parent and nephews are all black
        //   Step 1. Paint S to RED
        //   Step 2. Recursively maintain P
        //      [P]             [P]
        //      / \             / \
        //    [N] [S]  ====>  [N] <S>
        //        / \             / \
        //      [C] [D]         [C] [D]
        // clang-format on
        sibling->color = Node::RED;
        maintainAfterRemove(node->parent);
        return;
      }
```

```
    } else {
      if (closeNephew != nullptr && closeNephew->isRed()) {
        // clang-format off
        // Case 4: Sibling is BLACK, close nephew is RED,
        //         distant nephew is BLACK
        //   Step 1. If N is a left child, right rotate P;
        //           If N is a right child, left rotate P.
        //   Step 2. Swap the color of close nephew and sibling
        //   Step 3. Goto case 5
        //                              {P}                {P}
        //      {P}                     / \                / \
        //      / \    r-rotate(S)  [N] <C>   repaint  [N] [C]
        //    [N] [S]  ==========>        \   ======>        \
        //        / \                     [S]                <S>
        //      <C> [D]                     \                  \
        //                                  [D]                [D]
        // clang-format on

        // Step 1
        rotateOppositeDirection(sibling, direction);
        // Step 2
        closeNephew->color = Node::BLACK;
        sibling->color = Node::RED;
        // Update sibling and nephews after rotation
        sibling = node->sibling();
        closeNephew =
            direction == Direction::LEFT ? sibling->left : sibling->right;
        distantNephew =
            direction == Direction::LEFT ? sibling->right : sibling->left;
        // Step 3: vvv
      }

      // clang-format off
      // Case 5: Sibling is BLACK, close nephew is BLACK,
      //         distant nephew is RED
      //      {P}                   [S]
      //      / \    l-rotate(P)    / \
      //    [N] [S]  ==========>  {P} <D>
      //        / \               / \
      //      [C] <D>           [N] [C]
      // clang-format on
      assert(closeNephew == nullptr || closeNephew->isBlack());
      assert(distantNephew->isRed());
      // Step 1
      rotateSameDirection(node->parent, direction);
      // Step 2
      sibling->color = node->parent->color;
      node->parent->color = Node::BLACK;
      if (distantNephew != nullptr) {
        distantNephew->color = Node::BLACK;
      }
      return;
    }
  }
```

```
bool remove(NodePtr node, K key, NodeConsumer action) {
  assert(node != nullptr);

  if (key != node->key) {
    if (compare(key, node->key)) {
      /* key < node->key */
      NodePtr &left = node->left;
      if (left != nullptr && remove(left, key, action)) {
        maintainRelationship(node);
        return true;
      } else {
        return false;
      }
    } else {
      /* key > node->key */
      NodePtr &right = node->right;
      if (right != nullptr && remove(right, key, action)) {
        maintainRelationship(node);
        return true;
      } else {
        return false;
      }
    }
  }

  assert(key == node->key);
  action(node);

  if (this->size() == 1) {
    // Current node is the only node of the tree
    this->clear();
    return true;
  }

  if (node->left != nullptr && node->right != nullptr) {
    // clang-format off
    // Case 1: If the node is strictly internal
    //   Step 1. Find the successor S with the smallest key
    //           and its parent P on the right subtree.
    //   Step 2. Swap the data (key and value) of S and N,
    //           S is the node that will be deleted in place of N.
    //   Step 3. N = S, goto Case 2, 3
    //     |                    |
    //     N                    S
    //    / \                  / \
    //   L  ..   swap(N, S)   L  ..
    //       |   =========>       |
    //       P                    P
    //      / \                  / \
    //     S  ..                N  ..
    // clang-format on

    // Step 1
```

```
      NodePtr successor = node->right;
      NodePtr parent = node;
      while (successor->left != nullptr) {
        parent = successor;
        successor = parent->left;
      }
      // Step 2
      swapNode(node, successor);
      maintainRelationship(parent);
      // Step 3: vvv
    }

    if (node->isLeaf()) {
      // Current node must not be the root
      assert(node->parent != nullptr);

      // Case 2: Current node is a leaf
      //   Step 1. Unlink and remove it.
      //   Step 2. If N is BLACK, maintain N;
      //           If N is RED, do nothing.

      // The maintain operation won't change the node itself,
      //  so we can perform maintain operation before unlink the node.
      if (node->isBlack()) {
        maintainAfterRemove(node);
      }
      if (node->direction() == Direction::LEFT) {
        node->parent->left = nullptr;
      } else /* node->direction() == Direction::RIGHT */ {
        node->parent->right = nullptr;
      }
    } else /* !node->isLeaf() */ {
      assert(node->left == nullptr || node->right == nullptr);
      // Case 3: Current node has a single left or right child
      //   Step 1. Replace N with its child
      //   Step 2. If N is BLACK, maintain N
      NodePtr parent = node->parent;
      NodePtr replacement = (node->left != nullptr ? node->left : node->right);
      switch (node->direction()) {
        case Direction::ROOT:
          this->root = replacement;
          break;
        case Direction::LEFT:
          parent->left = replacement;
          break;
        case Direction::RIGHT:
          parent->right = replacement;
          break;
      }

      if (!node->isRoot()) {
        replacement->parent = parent;
      }
```

```
      if (node->isBlack()) {
        if (replacement->isRed()) {
          replacement->color = Node::BLACK;
        } else {
          maintainAfterRemove(replacement);
        }
      }
    }

    this->count -= 1;
    return true;
  }

  void inorderTraversal(NodeConsumer action) const {
    if (this->root == nullptr) {
      return;
    }

    std::stack<NodePtr> stack;
    NodePtr node = this->root;

    while (node != nullptr || !stack.empty()) {
      while (node != nullptr) {
        stack.push(node);
        node = node->left;
      }
      if (!stack.empty()) {
        node = stack.top();
        stack.pop();
        action(node);
        node = node->right;
      }
    }
  }
};

#endif  // RBTREE_MAP_HPP
```

其他资料

- Red-Black Tree - Wikipedia[5]
- Red-Black Tree Visualization[23]

[1] https://en.wikipedia.org/wiki/Red–black_tree#cite_note-Cormen2009-18

[2] https://en.wikipedia.org/wiki/Red–black_tree#cite_note-Mehlhorn2008-17

[3] https://en.wikipedia.org/wiki/Red–black_tree#cite_note-Algs4-16: 432–447

[4] 维基百科原文

[5] Wikipedia [5-1] [5-2]

[6] 代码示例

[7] linux 内核中的 rbtree

[8] std::_Rb_tree [8-1] [8-2]

[9] TreeMap [9-1] [9-2]

[10] Rudolf Bayer

[11] 从 2-3-4 树的角度学习理解红黑树（视频）

[12] 为什么 rust 中的 Map 使用的是 B 树而不是像其他主流语言一样使用红黑树

[13] linux/lib/rbtree.c

[14] CFS 非实时任务调度

[15] epoll

[16] nginx/src/core/ngx_rbtree.h

[17] nginx/src/core/ngx_rbtree.c

[18] libstdc++-v3/src/c++98/tree.cc

[19] libcxx/include/__tree

[20] stl/inc/xtree

[21] java.util.TreeSet<K, V>

[22] java.util.HashMap<K, V>

[23] Red-Black Tree Visualization

10.17.13 左偏红黑树

左偏红黑树是红黑树的一种变体，它的对红边（点）的位置做了一定限制，使得其插入与删除操作可以与 2-3 树构成一一对应。

我们假设读者已经至少掌握了一种基于旋转的平衡树，因此本文不会对旋转操作进行讲解。

红黑树

性质

一棵红黑树满足如下性质：

1. 节点是红色或黑色；
2. NIL 节点（空叶子节点）为黑色；
3. 红色的节点的所有儿子的颜色必须是黑色，即从每个叶子到根的所有路径上不能有两个连续的红色节点；
4. 从任一节点到其子树中的每个叶子的所有简单路径上都包含相同数目的黑色节点。（黑高平衡）

这保证了从根节点到任意叶子的最长路径（红黑交替）不会超过最短路径（全黑）的二倍。从而保证了树的平衡性。

维护这些性质是比较复杂的，如果我们要插入一个节点，首先，它一定会被染色成红色，否则会破坏性质 3。即使这样，我们还是有可能会破坏性质 2。因此需要进行调整。而删除节点就更加麻烦，与插入类似，我们不能删除黑色节点，否则会破坏黑高的平衡。如何方便地解决这些问题呢？

左偏红黑树（Left Leaning Red Black Tree）

解释

左偏红黑树是一种容易实现的红黑树变体。

在以下左偏红黑树示意图中，是边具有颜色而不是节点具有颜色。我们习惯用一个节点的颜色代指它的父亲边的颜色。

左偏红黑树对红黑树进行了进一步限制，一个黑色节点的左右儿子：

- 要么全是黑色；
- 要么左儿子是红色，右儿子是黑色。

符合条件的情况：

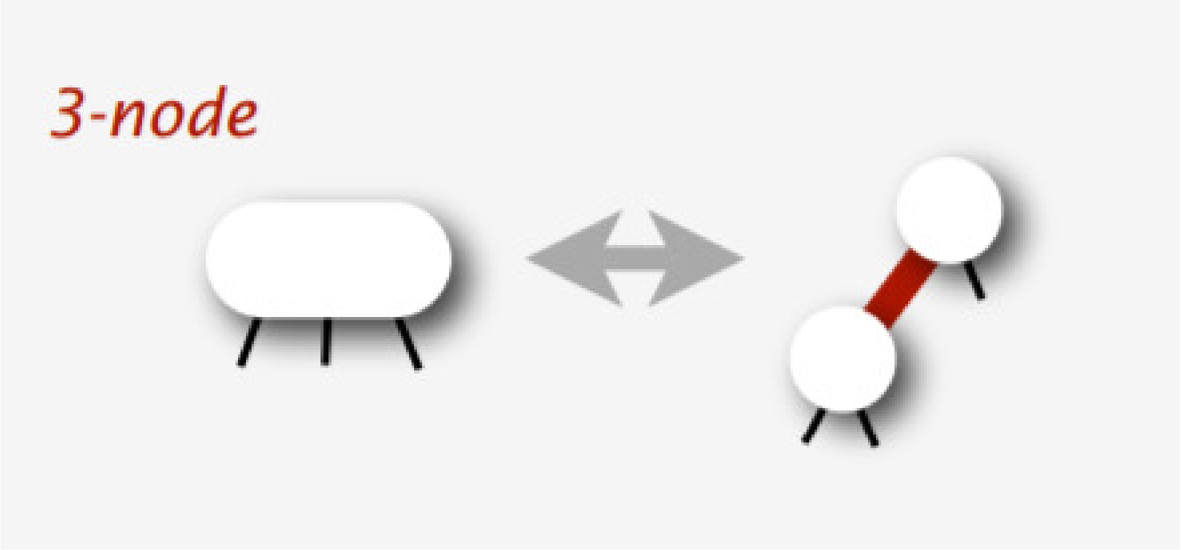


图 10.111 llrbt1

不符合条件的情况：

这是左偏树的「左偏」性质：红色边只能是左偏的。

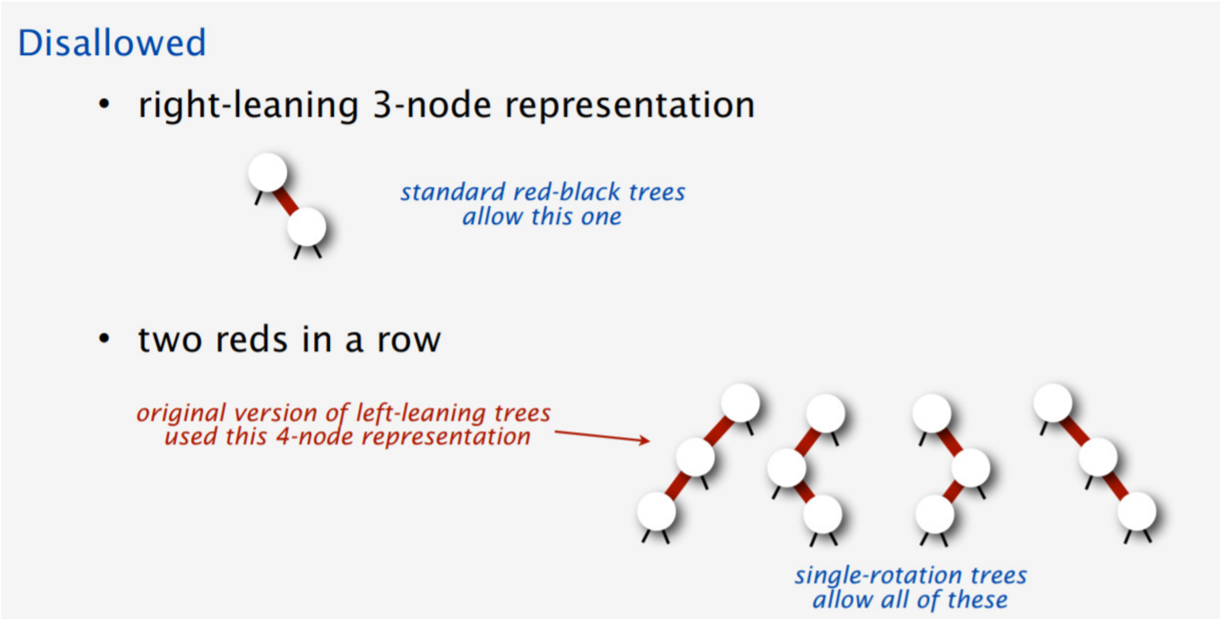


图 10.112 llrbt2

过程

插入

我们首先使用普通的 BST 插入方法，在树的底部插入一个红色的叶子节点，然后通过从下向上的调整，使得插入后的树仍然符合左偏红黑树的性质。下面描述调整的过程：

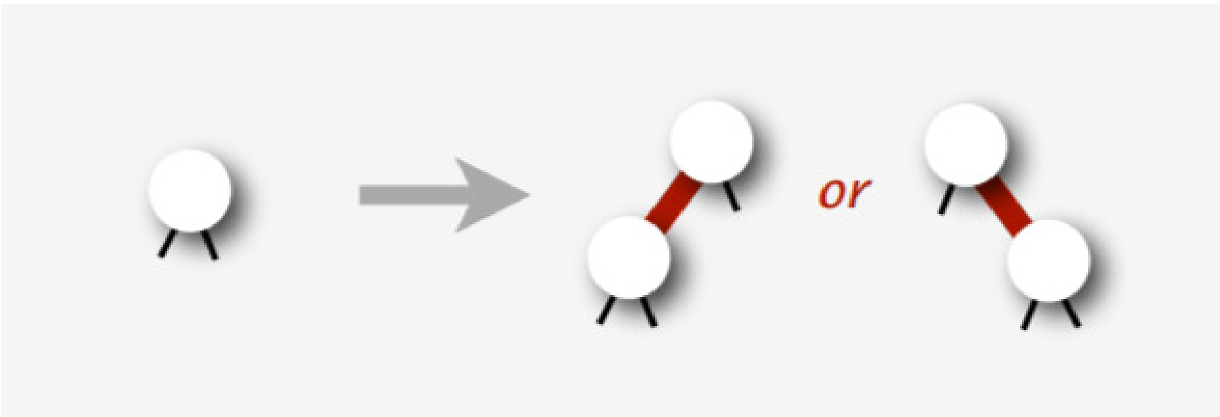


图 10.113 llrbt3

插入后，可能会产生一条右偏的红色边，因此需要对红边右偏的情况进行一次左旋：

考虑左旋后会产生两条连续的左偏红色边：

因此需要把它进行一次右旋。而对于右旋后的情况，我们应该对它进行 `color_flip`：即翻转该节点和它的两个儿子的颜色

从而消灭右偏的红边。

" 参考代码（部分）"

```
template <class Key, class Compare>
typename Set<Key, Compare>::Node *Set<Key, Compare>::fix_up(
    Set::Node *root) const {
  if (is_red(root->rc) && !is_red(root->lc))  // fix right leaned red link
```

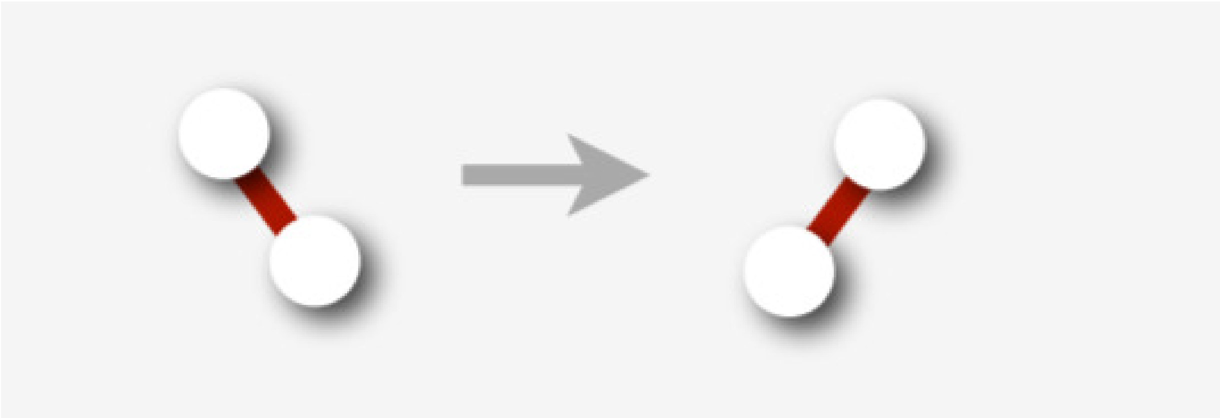


图 10.114 llrbt4

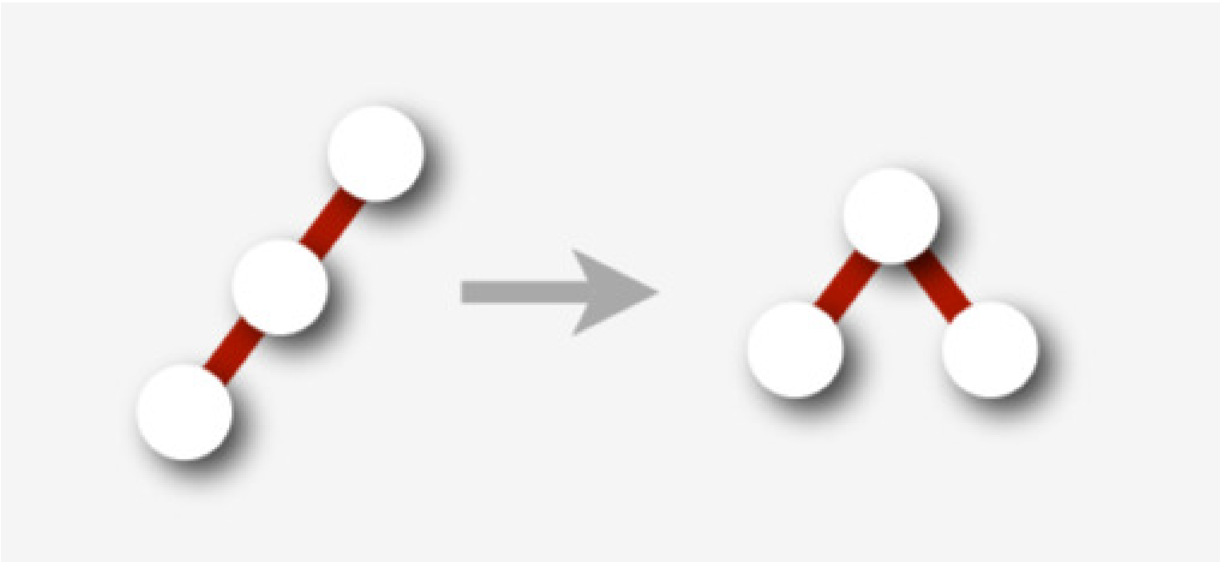


图 10.115 llrbt5

图 10.116 llrbt6

```
    root = rotate_left(root);
  if (is_red(root->lc) &&
      is_red(root->lc->lc))  // fix doubly linked left leaned red link
    // if (root->lc == nullptr), then the second expr won't be evaluated
    root = rotate_right(root);
  if (is_red(root->lc) && is_red(root->rc))
    // break up 4 node
    color_flip(root);
  root->size = size(root->lc) + size(root->rc) + 1;
  return root;
}

template <class Key, class Compare>
typename Set<Key, Compare>::Node_Set<Key, Compare>::insert(
    Set::Node_root, const Key &key) const {
  if (root == nullptr) return new Node(key, kRed, 1);
  if (root->key == key)
    ;
  else if (cmp\_(key, root->key))  // if (key < root->key)
    root->lc = insert(root->lc, key);
  else
    root->rc = insert(root->rc, key);
  return fix_up(root);
}
```

删除

删除操作基于这样的思想：我们不能删除黑色的节点，因为这样会破坏黑高。所以我们需要保证我们最后删除的节点是红色的。

删除最小值节点

首先来试一下删除整棵树里的最小值。

怎么才能保证最后删除的节点是红色的呢？我们需要在向下递归的过程中保证一个性质：如果当前节点是 `h`，那么需要保证 `h` 是红色，或者 `h->lc` 是红色。

考虑这样做的正确性，如果我们能够通过各种旋转和反转颜色操作成功维护这个性质，那么当我们到达最小的节点 `h_min` 的时候，有 `h_min` 是红色，或者 `h_min` 的左子树——但是 `h_min` 根本没有左子树！所以这就保证了最小值节点一定是红的，既然它是红色的，我们就可以大胆的删除它，然后用与插入操作相同的调整思路对树进行调整。

下面我们来考虑怎么满足这个性质，注意，我们会在向下递归的时候**临时地**破坏左偏红黑树的若干性质，但是当我们从递归中返回时还会将其恢复。

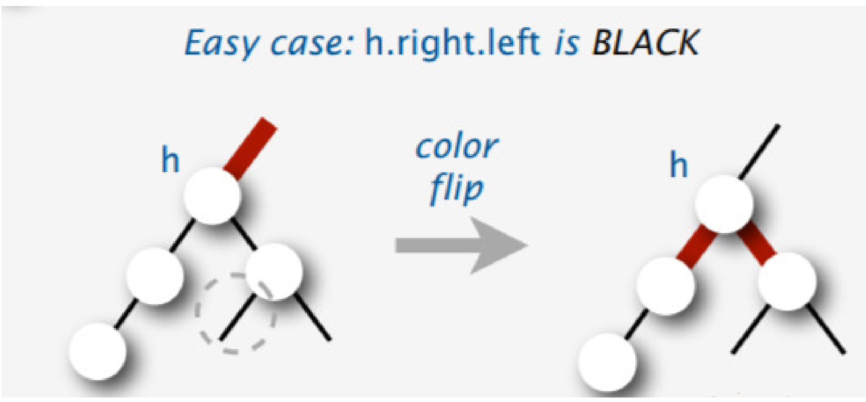


图 10.117 llrbt-7

下图描述一种简单的情况，我们只需要一次翻转颜色即可。

但如果 h->rc->lc 是红色，情况会比较复杂：

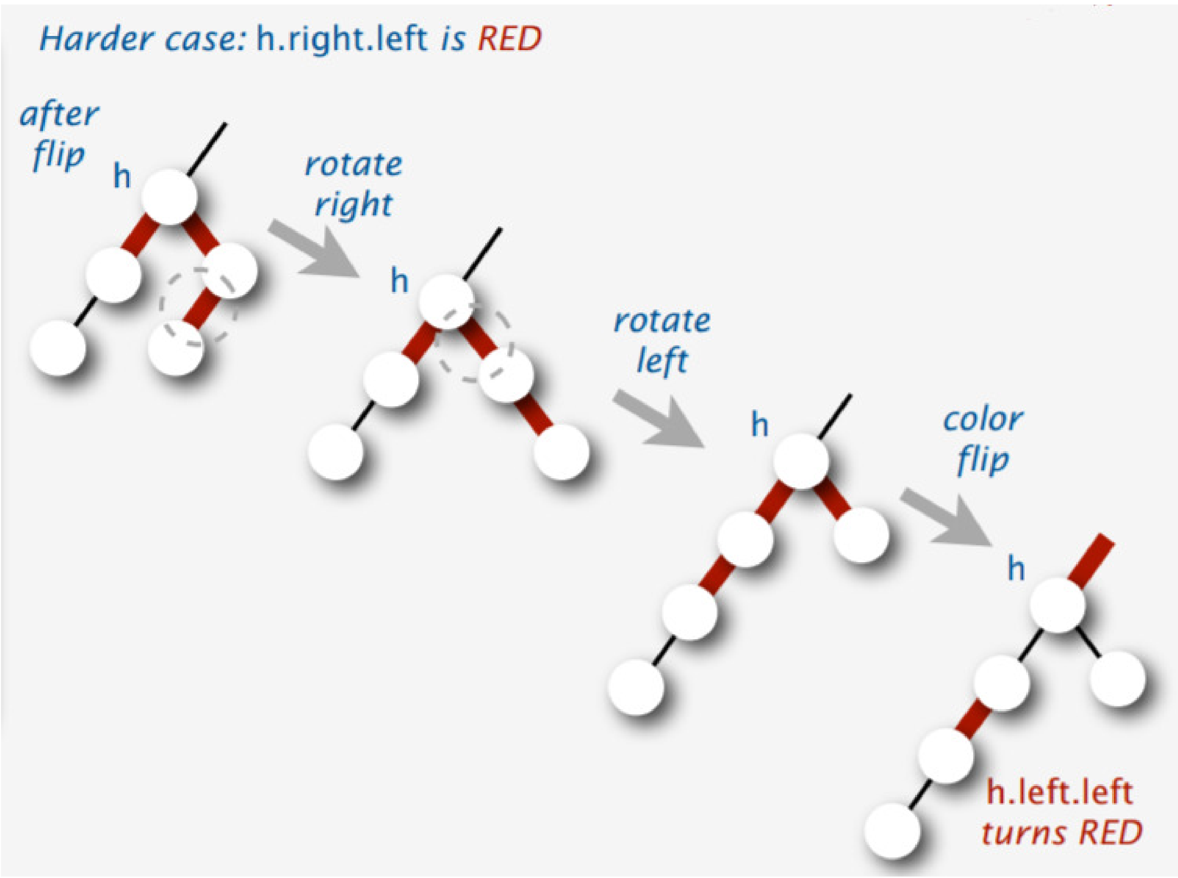


图 10.118 llrbt-8

如果只进行翻转颜色，会产生连续的红边，而考虑我们递归返回的时候，是无法修复这样的情况的，因此需要进行处理。

然后就可以进行删除了：

" 参考代码（部分）"

```cpp
template <class Key, class Compare>
typename Set<Key, Compare>::Node *Set<Key, Compare>::move_red_left(
    Set::Node *root) const {
  color_flip(root);
  if (is_red(root->rc->lc)) {
    // assume that root->rc != nullptr when calling this function
    root->rc = rotate_right(root->rc);
    root = rotate_left(root);
    color_flip(root);
  }
  return root;
}

template <class Key, class Compare>
typename Set<Key, Compare>::Node *Set<Key, Compare>::delete_min(
    Set::Node *root) const {
  if (root->lc == nullptr) {
    delete root;
```

```
      return nullptr;
    }
    if (!is_red(root->lc) && !is_red(root->lc->lc)) {
      // make sure either root->lc or root->lc->lc is red
      // thus make sure we will delete a red node in the end
      root = move_red_left(root);
    }
    root->lc = delete_min(root->lc);
    return fix_up(root);
  }
```

删除任意节点

我们首先考虑删除叶子：与删最小值类似，我们在删除任意值的过程中也要维护一个性质，不过这次比较特殊，因为我们不是只向左边走，而是可以向左右两个方向走，因此在删除过程中维护的性质是这样的：如果往左走，当前节点是 h，那么需要保证 h 是红色，或者 h->lc 是红色；如果往右走，当前节点是 h，那么需要保证 h 是红色，或者 h->rc 是红色。这样可以保证我们最后总会删掉一个红色节点。

下面考虑删除非叶子节点，我们只需要找到其右子树（如果有）里的最小节点，然后用右子树的最小节点的值代替该节点的值，最后删除右子树里的最小节点。

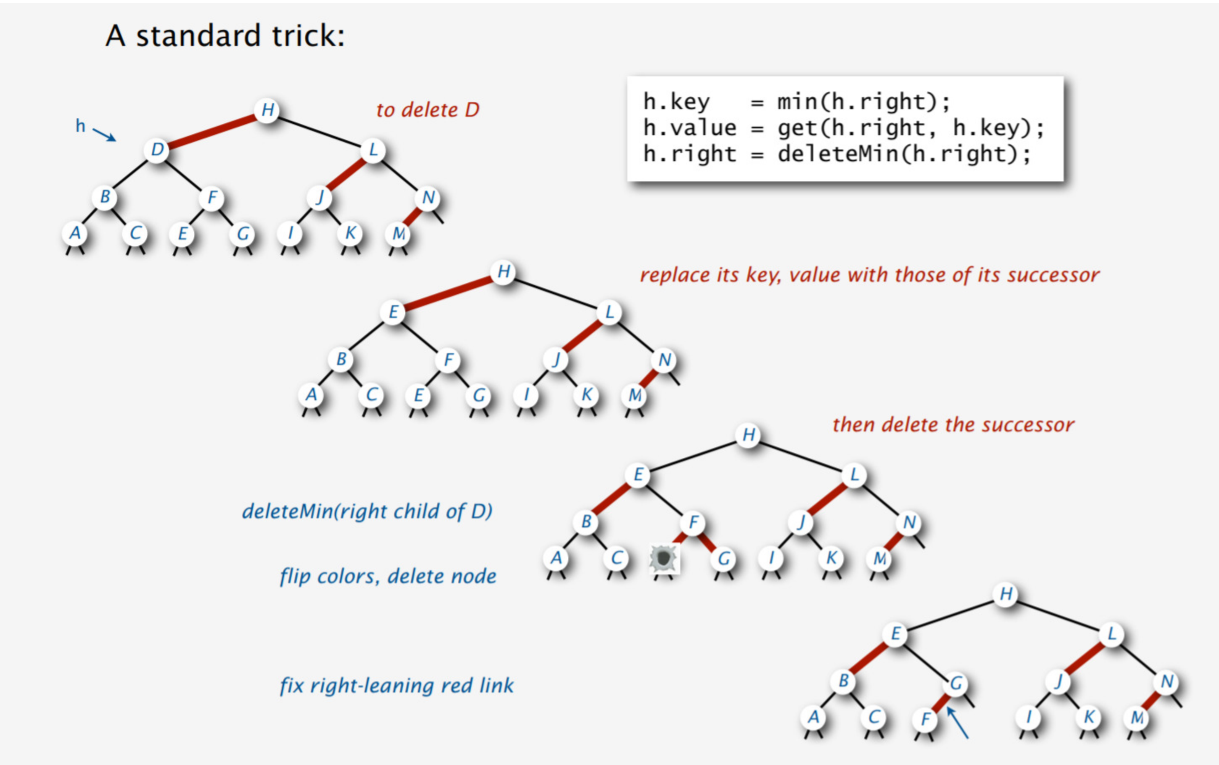


图 10.119 llrbt-9

那如果没有右子树怎么办？我们需要把左子树旋转过来，这样就不会出现这个问题了。

" 参考代码（部分）"

```
template <class Key, class Compare>
typename Set<Key, Compare>::Node *Set<Key, Compare>::delete_arbitrary(
    Set::Node *root, Key key) const {
  if (cmp_(key, root->key)) {
    // key < root->key
    if (!is_red(root->lc) && !(is_red(root->lc->lc)))
```

```
      root = move_red_left(root);
    // ensure the invariant: either root->lc or root->lc->lc (or root and
    // root->lc after dive into the function) is red, to ensure we will
    // eventually delete a red node. therefore we will not break the black
    // height balance
    root->lc = delete_arbitrary(root->lc, key);
  } else {
    // key >= root->key
    if (is_red(root->lc)) root = rotate_right(root);
    if (key == root->key && root->rc == nullptr) {
      delete root;
      return nullptr;
    }
    if (!is_red(root->rc) && !is_red(root->rc->lc)) root = move_red_right(root);
    if (key == root->key) {
      root->key = get_min(root->rc);
      root->rc = delete_min(root->rc);
    } else {
      root->rc = delete_arbitrary(root->rc, key);
    }
  }
  return fix_up(root);
}
```

实现

下面的代码是用左偏红黑树实现的 Set，即有序不可重集合：

" 参考代码"

```
#include <algorithm>
#include <memory>
#include <vector>

template <class Key, class Compare = std::less<Key>>
class Set {
 private:
  enum NodeColor { kBlack = 0, kRed = 1 };

  struct Node {
    Key key;
    Node *lc{nullptr}, *rc{nullptr};
    size_t size{0};
    NodeColor color;  // the color of the parent link

    Node(Key key, NodeColor color, size_t size)
        : key(key), color(color), size(size) {}

    Node() = default;
  };

  void destroyTree(Node *root) const {
    if (root != nullptr) {
      destroyTree(root->lc);
```

```
      destroyTree(root->rc);
      root->lc = root->rc = nullptr;
      delete root;
    }
  }

  bool is_red(const Node *nd) const {
    return nd == nullptr ? false : nd->color;  // kRed == 1, kBlack == 0
  }

  size_t size(const Node *nd) const { return nd == nullptr ? 0 : nd->size; }

  Node *rotate_left(Node *node) const {
    // left rotate a red link
    //          <1>                  <2>
    //        /    \\               //    \
    //       *      <2>    ==>     <1>     *
    //             /   \          /   \
    //            *     *        *     *
    Node *res = node->rc;
    node->rc = res->lc;
    res->lc = node;
    res->color = node->color;
    node->color = kRed;
    res->size = node->size;
    node->size = size(node->lc) + size(node->rc) + 1;
    return res;
  }

  Node *rotate_right(Node *node) const {
    // right rotate a red link
    //            <1>               <2>
    //          //    \           /    \\
    //         <2>     *   ==>   *      <1>
    //        /   \                    /   \
    //       *     *                  *     *
    Node *res = node->lc;
    node->lc = res->rc;
    res->rc = node;
    res->color = node->color;
    node->color = kRed;
    res->size = node->size;
    node->size = size(node->lc) + size(node->rc) + 1;
    return res;
  }

  NodeColor neg_color(NodeColor n) const { return n == kBlack ? kRed : kBlack; }

  void color_flip(Node *node) const {
    node->color = neg_color(node->color);
    node->lc->color = neg_color(node->lc->color);
    node->rc->color = neg_color(node->rc->color);
  }
```

```
  Node *insert(Node *root, const Key &key) const;
  Node *delete_arbitrary(Node *root, Key key) const;
  Node *delete_min(Node *root) const;
  Node *move_red_right(Node *root) const;
  Node *move_red_left(Node *root) const;
  Node *fix_up(Node *root) const;
  const Key &get_min(Node *root) const;
  void serialize(Node *root, std::vector<Key> *) const;
  void print_tree(Set::Node *root, int indent) const;
  Compare cmp_ = Compare();
  Node *root_{nullptr};

 public:
  typedef Key KeyType;
  typedef Key ValueType;
  typedef std::size_t SizeType;
  typedef std::ptrdiff_t DifferenceType;
  typedef Compare KeyCompare;
  typedef Compare ValueCompare;
  typedef Key &Reference;
  typedef const Key &ConstReference;

  Set() = default;

  Set(Set &) = default;

  Set(Set &&) noexcept = default;

  ~Set() { destroyTree(root_); }

  SizeType size() const;

  SizeType count(const KeyType &key) const;

  SizeType erase(const KeyType &key);

  void clear();

  void insert(const KeyType &key);

  bool empty() const;

  std::vector<Key> serialize() const;

  void print_tree() const;
};

template <class Key, class Compare>
typename Set<Key, Compare>::SizeType Set<Key, Compare>::count(
    ConstReference key) const {
  Node *x = root_;
  while (x != nullptr) {
    if (key == x->key) return 1;
    if (cmp_(key, x->key))  // if (key < x->key)
```

```
      x = x->lc;
    else
      x = x->rc;
  }
  return 0;
}

template <class Key, class Compare>
typename Set<Key, Compare>::SizeType Set<Key, Compare>::erase(
    const KeyType &key) {
  if (count(key) > 0) {
    if (!is_red(root_->lc) && !(is_red(root_->rc))) root_->color = kRed;
    root_ = delete_arbitrary(root_, key);
    if (root_ != nullptr) root_->color = kBlack;
    return 1;
  } else {
    return 0;
  }
}

template <class Key, class Compare>
void Set<Key, Compare>::clear() {
  destroyTree(root_);
  root_ = nullptr;
}

template <class Key, class Compare>
void Set<Key, Compare>::insert(const KeyType &key) {
  root_ = insert(root_, key);
  root_->color = kBlack;
}

template <class Key, class Compare>
bool Set<Key, Compare>::empty() const {
  return size(root_) == 0;
}

template <class Key, class Compare>
typename Set<Key, Compare>::Node *Set<Key, Compare>::insert(
    Set::Node *root, const Key &key) const {
  if (root == nullptr) return new Node(key, kRed, 1);
  if (root->key == key)
    ;
  else if (cmp_(key, root->key))  // if (key < root->key)
    root->lc = insert(root->lc, key);
  else
    root->rc = insert(root->rc, key);
  return fix_up(root);
}

template <class Key, class Compare>
typename Set<Key, Compare>::Node *Set<Key, Compare>::delete_min(
    Set::Node *root) const {
  if (root->lc == nullptr) {
```

```
    delete root;
    return nullptr;
  }
  if (!is_red(root->lc) && !is_red(root->lc->lc)) {
    // make sure either root->lc or root->lc->lc is red
    // thus make sure we will delete a red node in the end
    root = move_red_left(root);
  }
  root->lc = delete_min(root->lc);
  return fix_up(root);
}

template <class Key, class Compare>
typename Set<Key, Compare>::Node *Set<Key, Compare>::move_red_right(
    Set::Node *root) const {
  color_flip(root);
  if (is_red(root->lc->lc)) {  // assume that root->lc != nullptr when calling
                               // this function
    root = rotate_right(root);
    color_flip(root);
  }
  return root;
}

template <class Key, class Compare>
typename Set<Key, Compare>::Node *Set<Key, Compare>::move_red_left(
    Set::Node *root) const {
  color_flip(root);
  if (is_red(root->rc->lc)) {
    // assume that root->rc != nullptr when calling this function
    root->rc = rotate_right(root->rc);
    root = rotate_left(root);
    color_flip(root);
  }
  return root;
}

template <class Key, class Compare>
typename Set<Key, Compare>::Node *Set<Key, Compare>::fix_up(
    Set::Node *root) const {
  if (is_red(root->rc) && !is_red(root->lc))  // fix right leaned red link
    root = rotate_left(root);
  if (is_red(root->lc) &&
      is_red(root->lc->lc))  // fix doubly linked left leaned red link
    // if (root->lc == nullptr), then the second expr won't be evaluated
    root = rotate_right(root);
  if (is_red(root->lc) && is_red(root->rc))
    // break up 4 node
    color_flip(root);
  root->size = size(root->lc) + size(root->rc) + 1;
  return root;
}

template <class Key, class Compare>
```

```
const Key &Set<Key, Compare>::get_min(Set::Node *root) const {
  Node *x = root;
  // will crash as intended when root == nullptr
  for (; x->lc != nullptr; x = x->lc)
    ;
  return x->key;
}

template <class Key, class Compare>
typename Set<Key, Compare>::SizeType Set<Key, Compare>::size() const {
  return size(root_);
}

template <class Key, class Compare>
typename Set<Key, Compare>::Node *Set<Key, Compare>::delete_arbitrary(
    Set::Node *root, Key key) const {
  if (cmp_(key, root->key)) {
    // key < root->key
    if (!is_red(root->lc) && !(is_red(root->lc->lc)))
      root = move_red_left(root);
    // ensure the invariant: either root->lc or root->lc->lc (or root and
    // root->lc after dive into the function) is red, to ensure we will
    // eventually delete a red node. therefore we will not break the black
    // height balance
    root->lc = delete_arbitrary(root->lc, key);
  } else {
    // key >= root->key
    if (is_red(root->lc)) root = rotate_right(root);
    if (key == root->key && root->rc == nullptr) {
      delete root;
      return nullptr;
    }
    if (!is_red(root->rc) && !is_red(root->rc->lc)) root = move_red_right(root);
    if (key == root->key) {
      root->key = get_min(root->rc);
      root->rc = delete_min(root->rc);
    } else {
      root->rc = delete_arbitrary(root->rc, key);
    }
  }
  return fix_up(root);
}

template <class Key, class Compare>
std::vector<Key> Set<Key, Compare>::serialize() const {
  std::vector<int> v;
  serialize(root_, &v);
  return v;
}

template <class Key, class Compare>
void Set<Key, Compare>::serialize(Set::Node *root,
                                  std::vector<Key> *res) const {
  if (root == nullptr) return;
```

```
  serialize(root->lc, res);
  res->push_back(root->key);
  serialize(root->rc, res);
}

template <class Key, class Compare>
void Set<Key, Compare>::print_tree(Set::Node *root, int indent) const {
  if (root == nullptr) return;
  print_tree(root->lc, indent + 4);
  std::cout << std::string(indent, '-') << root->key << std::endl;
  print_tree(root->rc, indent + 4);
}

template <class Key, class Compare>
void Set<Key, Compare>::print_tree() const {
  print_tree(root_, 0);
}
```

参考资料与拓展阅读

- Left-Leaning Red-Black Trees[1]- Robert Sedgewick Princeton University
- Balanced Search Trees[2]-_Algorithms_Robert Sedgewick | Kevin Wayne

参考资料与注释

[1] Left-Leaning Red-Black Trees

[2] Balanced Search Trees

10.17.14 AA 树

AA 树是一种用于高效存储和检索有序数据的平衡树形结构，Arne Andersson 教授于 1993 年在他的论文"Balanced search trees made simple" 中介绍，设计的目的是减少红黑树考虑的不同情况。AA 树可以在 $O(\log N)$ 的时间内做查找，插入和删除。下面是一个 AA 树的例子。

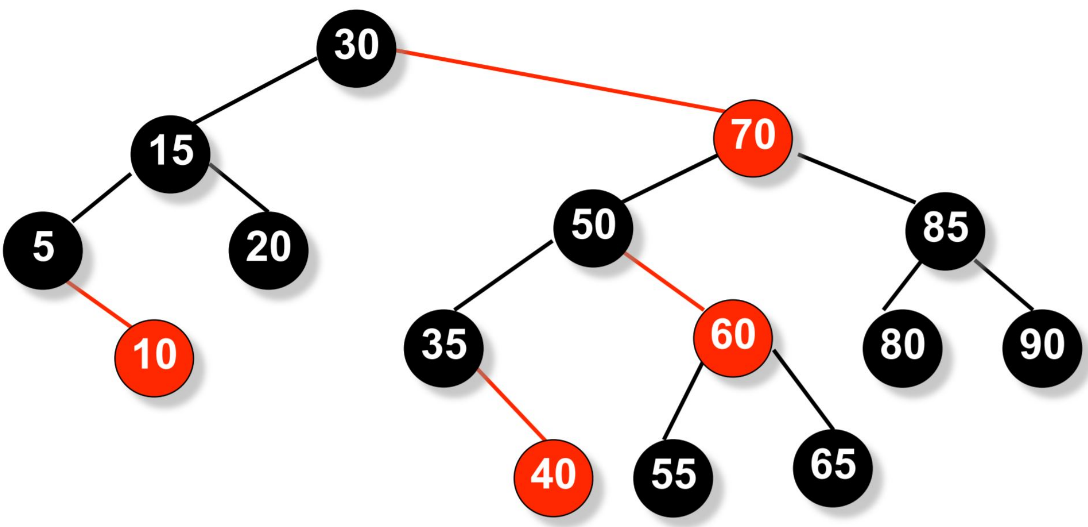


图 10.120 aa-tree-1

AA 树是红黑树的一种变体，与红黑树不同，AA 树上的红色节点只能作为右子节点。这导致 AA 树模拟了 2-3 树而不是 2-3-4 树，从而极大地简化了维护操作。红黑树的维护算法需要考虑七种不同的情况来正确平衡树。

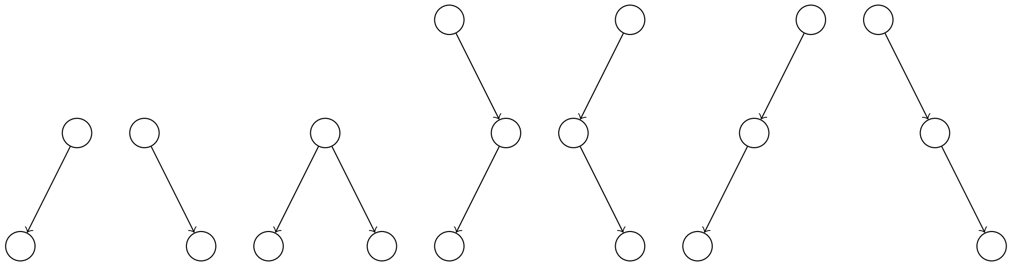


图 10.121 red-black tree

因为红色节点只能作为右子节点，AA 树只需要考虑两种情况。

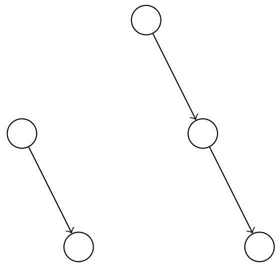


图 10.122 aa-tree

定义

AA 树遵循与红黑树相同的规则，但添加了一条新规则，**即红色节点不能作为左孩子出现**。

1. 每个节点都可以是红色或黑色。
2. 根节点总是黑色。
3. 叶节点（NULL）总是黑色。
4. 红色节点的两个子节点必须都是黑色，即没有两个相邻的红色节点。
5. 从根节点到 NULL 节点的每条路径都有相同数量的黑色节点。
6. 红色节点只能作为右子节点。

平衡维护

AA 树的每个节点维护一个 **level** 字段，类似红黑树的每个节点维护一个 color 字段 (”RED” or ”BLACK”)。level 的规定满足以下 5 个条件：

1、每个叶节点的 level 是 1。

2、每个左孩子的 level 是其父节点的 level 减 1。

3、每个右孩子的 level 等于其父节点的 level 或等于其父节点的 level 减 1。

4、每个右孙子的 level 严格小于其祖父节点的 level。

5、每个 level 大于 1 的节点有两个孩子。

水平链接（Horizontal Link）

子节点的 level 等于父节点的 level 的链接被称为**水平链接**，类似于红黑树中的红链接。允许单独的右水平链接，但不允许连续的右水平链接；不允许左水平链接。这些限制比红黑树的限制更加严格，因此 AA 树的平衡过程比红黑树的平衡过程在程序上要简单得多。

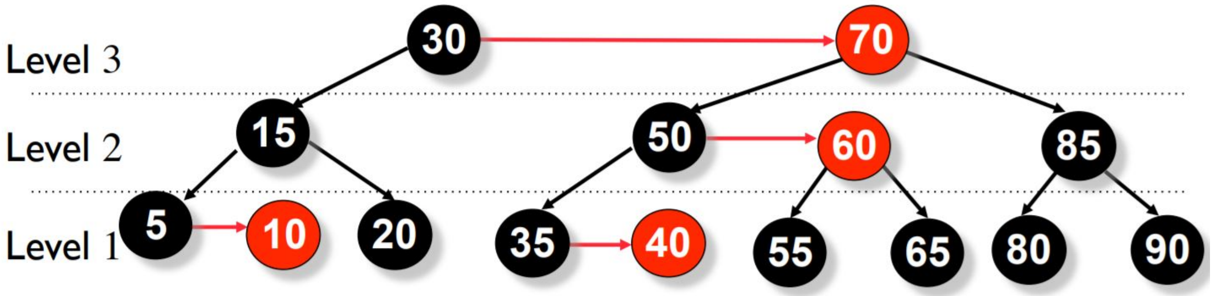


图 10.123 aa-tree-4

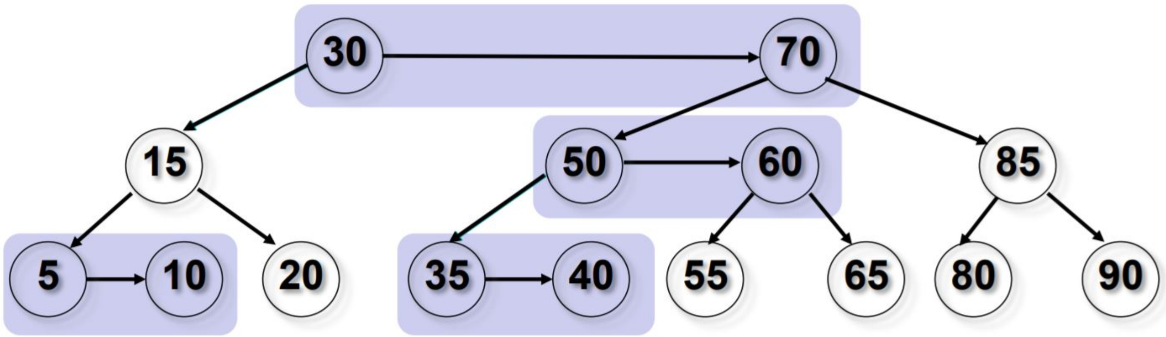


图 10.124 aa-tree-5

插入和删除操作可能会暂时导致 AA 树失去平衡（即违反 AA 树的不变性）。恢复平衡只需要两种不同的操作："**skew**"（斜化）和"**split**"（分裂）。"Skew" 是将一个包含左水平链接的子树进行右旋转，以替换为一个包含右水平链接的子树。"Split" 是进行左旋转并增加 level，以替换一个包含两个或更多连续的右水平链接的子树，使其变为一个包含两个较少连续的右水平链接的子树。保持平衡的插入和删除的实现通过依赖"skew" 和"split" 操作来仅在需要时修改树，而不是由调用者决定是否进行"skew" 或"split"，从而变得更加简化。

split（左旋）

出现连续向右的水平方向链（连续三个向右的孩子属于同一 level，节点 R 和节点 X 都是红色节点）。

此时向左旋转节点 T，把小于等于此 level 的节点看做一个子树。

1. 子树的根的右孩子变为新的子树根；
2. 原来的子树根变为新子树根的左孩子；
3. 新的子树根 level+1。

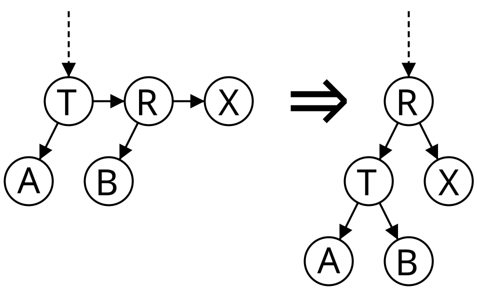


图 10.125 aa-tree-split

" 伪代码实现"

```
function split(root)
    if root → right → right → level == root → level
        rotate_left(root)
end function
```

skew（右旋）

出现向左的水平方向链（连续两个向左的孩子属于同一 level）

向右旋转节点 T，把小于等于此 level 的节点看做一个子树。

1. 子树的根的左孩子变为新的子树根；
2. 原来的子树根变为新子树根的右孩子。

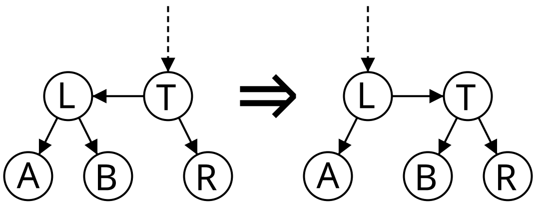


图 10.126 aa-tree-skew

" 伪代码实现"

```
function skew(root)
    if root → left → level == root → level
        rotate_right(root)
end function
```

AA 树的操作

AA 树本身是一棵二叉搜索树，所以搜索操作与其他二叉搜索树相同。插入和删除操作与 AVL 树相同，首先在树中将 key 插入或删除，然后沿着搜索路径回退到根，并在此过程中重构树。

插入

" 伪代码实现"

```
function insert(root, add)
    if root == NULL
        root ← add
    else if add → key < root → key      //      <=
        insert(root → left, add)
    else if add → key > root → key
        insert(root → right, add)
    end if
    //如果不允许重复，在每一 level 上进行 skew 和 split
    skew(root);
    split(root);
end function
```

删除

删除过程与其他二叉平衡树类似，首先将内部节点的删除转换为叶子节点的删除。具体方法是将内部节点与它最接近的前驱或后继节点替换。由于 AA 树的所有 level 大于 1 的节点都有两个子节点，前驱或后继节点将位于 level 1，删除 level 1 的节点较为简单。

" 伪代码实现"

```
//To rebalance the tree
if root->left->level < root->level − 1 or root->right->level < root->level − 1
{
    if root->right->level > –root->level
    {
        root->right->level ← root->level
    }
    skew(root)
    skew(root->right)
    skew(root->right->right)
    split(root)
    split(root->right)
}
```

性能

AA 树的性能与红黑树的性能相当。尽管 AA 树进行的旋转操作比红黑树多，但 AA 树的算法更简单，最终导致相近的性能。红黑树的性能在各种情况下更加一致，而 AA 树往往更扁平，这使 AA 树有稍快的搜索速度。

参考资料

1. AA tree - Wikipedia[1]
2. Introduction to AA trees[2]

参考资料与注释

[1] AA tree - Wikipedia

[2] Introduction to AA trees

10.17.15 2-3 树

本页面介绍 2-3 树。2-3 树是一种多路搜索树，是绝对平衡的树，其所有叶节点都处于同一层级上。2-3 树是 3 阶 B 树。

定义

2-3 树中的每一个节点都有两个孩子（称为 2 节点，2-node）或三个孩子（称为 3 节点，3-node）。

- 2 节点，有一个数据元素和两个孩子。只能有两个孩子或没有孩子，不能出现只有一个孩子的情况。如果 2 节点有孩子，左子树包含的元素小于 a，右子树包含的元素大于 a。

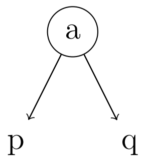


图 10.127 2-3-tree-2-node

- 3 节点，有两个数据元素和三个孩子。只能有三个孩子或没有孩子，不能出现有一个孩子或有两个孩子的情况。如果 3 节点有孩子，左子树包含小于较小元素的元素，右子树包含大于较大元素的元素，中间子树包含介于两元素之间的元素。

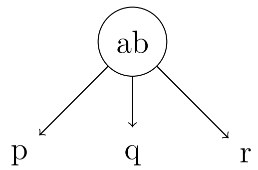


图 10.128 2-3-tree-3-node

- 4 节点，有三个数据元素。只会在操作树时暂时创建，而不会永久存储在树上。

2-3 树的叶节点不含有子节点，有一个或两个数据元素。

树 T 是一个 2-3 树，当且仅当以下表述之一成立：

1. T 是空树。
2. T 是一个 2 节点，并带有元素 a。如果 T 有左孩子 p 和右孩子 q，则：
    - p 和 q 是相同高度的 2-3 树。
    - a 大于 p 中的每个元素。
    - a 小于 q 中的每个数据元素。
3. T 是一个 3 节点，并带有数据元素 a 和 b，其中 a 小于 b。如果 T 有左孩子 p，中间孩子 q 和右孩子 r，则：
    - p、q 和 r 是相等高度的 2-3 树。
    - a 大于 p 中的每个数据元素且小于 q 中的每个数据元素。
    - b 大于 q 中的每个数据元素且小于 r 中的每个数据元素。

性质

- 所有内部节点都是 2 节点或 3 节点。
- 所有叶节点都在同一层级上。
- 树上的所有数据都是按顺序保存的（多路搜索树的性质）。

操作

查找

由于 2-3 树上的元素是按一定顺序存储的，所以 2-3 树的查找与二叉搜索树类似。下面在树 T 中查找元素 d。

1. 如果 2–3 树 T 为空，那么 d 肯定不在 T 中，查找结束。
2. 假设 t 是 T 的根节点。
3. 如果 t 是一个叶子节点：
    - 如果 d 不在节点 t 中，那么 d 肯定不在整个树 T 中，查找结束。
    - 如果 d 在节点 t 中，那么 d 在树 T 中，查找成功结束。
4. 假设 t 是一个"2 节点"，具有左子节点 p 和右子节点 q，a 是节点 t 中的数据元素。有以下三种情况：
    - 如果 d 等于 a，那么找到 d 在树 T 中，查找成功结束。
    - 如果 d 小于 a，将 T 设为子树 p，并回到步骤 2 继续查找。
    - 如果 d 大于 a，将 T 设为子树 q，并回到步骤 2 继续查找。
5. 假设 t 是一个"3 节点"，具有左子节点 p、中间子节点 q 和右子节点 r，a 和 b 是节点 t 中的两个数据元素，满足 $a < b$。有以下四种情况：
    - 如果 d 等于 a 或 b，那么找到 d 在树 T 中，查找成功结束。
    - 如果 d 小于 a，将 T 设为子树 p，回到步骤 2 继续查找。
    - 如果 d 在 a 和 b 之间，将 T 设为子树 q，回到步骤 2 继续查找。
    - 如果 d 大于 b，将 T 设为子树 r，回到步骤 2 继续查找。

插入

插入节点需要维持树的平衡。

对于空树，直接插入一个 2 节点即可。此外，为了保持完美平衡性，向 2-3 树添加元素不会直接添加到空节点，而是先进行搜索，将待插入元素添加到最后搜索到的叶子节点，与它融合。

1. 如果插入 2 节点，则融合形成 3 节点。如下图插入元素 4。

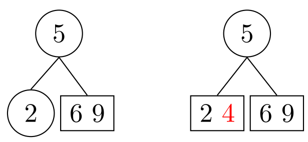


图 10.129 2-3-tree-insert

2. 如果插入 3 节点，则先融合形成 4 节点，再拆解形成 2 个 2 节点。
    - 如果父节点为 2 节点，则中间元素上移与父节点融合形成 3 节点。如下图插入元素 10。
    - 如果父节点为 3 节点，则重复步骤 2。如下图插入元素 1。

通过上述深度增加的例子，可以看出 2-3 树和标准二叉树不同，标准的二叉树的的深度是由上到下的增加的，而 2-3 树的深度是由下至上的增加的。

删除

删除节点需要维持树的平衡。2-3 树的删除可以分为三种情况。

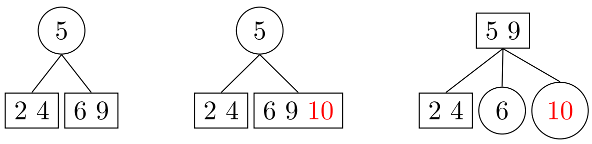


图 10.130 2-3-tree-insert

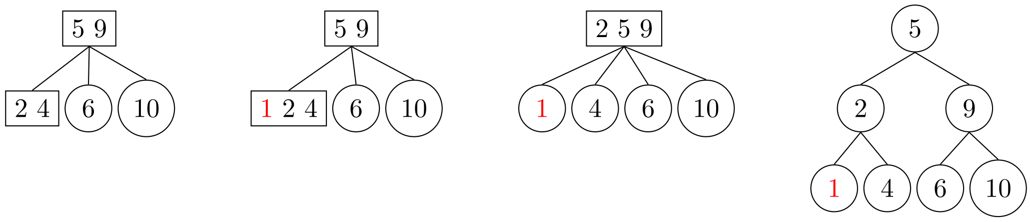


图 10.131 2-3-tree-insert

1. 待删除元素位于一个 3 节点的叶子节点上。只需要在该节点处删除该元素即可，不会影响到整棵树的其他节点结构。

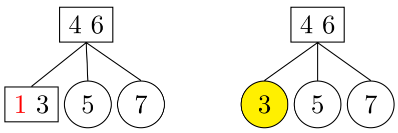


图 10.132 2-3-tree-delete

2. 待删除的元素位于非叶子的分支节点。通常先按中序遍历后得到此元素的前驱或后继元素，让他们来补位，再删除用于补位的前驱或后继元素。如果我们要删除的分支节点是 2 节点，如上下图所示，待删除的元素是 4，前驱是 1，后继是 6，显然 (6 7) 是 3 节点，只需要用 6 来补位即可。这里就不讲解删除的分支节点是 3 节点的情况了，与上述类似。

3. 所删除的元素位于一个 2 节点上。如果删除一个 2 节点，很有可能造成 2-3 树平衡破坏的情况，因为对于每一个 2 节点，要么有两个子树要么没有，对于每一个 3 节点要么有三个子树要么没有，贸然删除一个 2 节点，很可能出现平衡遭到破坏，所以我们需要分情况讨论。

1. 此节点的父亲节点是 2 节点，兄弟节点是 3 节点。将父亲节点移动到当前位置，再将兄弟节点中最接近当前位置的元素移动到父亲节点中。如下图所示，待删除元素为 1。
2. 此节点的父亲节点是 2 节点，兄弟节点也是 2 节点。先通过移动兄弟节点中序遍历的直接后驱到兄弟节点，使兄弟节点变为 3 节点，再进行上述 3.1 的操作。如下图所示，待删除元素为 4，如果直接执行上述 3.1 的操作会造成没有右孩子，因此需要对整棵树变形。通过移动兄弟节点 7 中序遍历的直接后驱 8 到兄弟节点，让节点 7 变成 3 节点，再让比 8 大的 9 补充到 8 的位置，最后再执行上述 3.1 的操作。
3. 此节点的父亲节点是 3 节点。拆分父亲节点使其成为 2 节点，再将父亲节点中最接近删除元素的元素与中孩子合并，将合并后的节点作为当前节点。如下图所示，待删除元素为 10。
4. 当前 2-3 树是一个满二叉树。将 2-3 树的层数减少，并将兄弟节点合并到父亲节点中，同时将父亲节点的所有兄弟节点合并到父亲节点的父亲节点中，如果生成了 4 节点，再分解 4 节点即可。

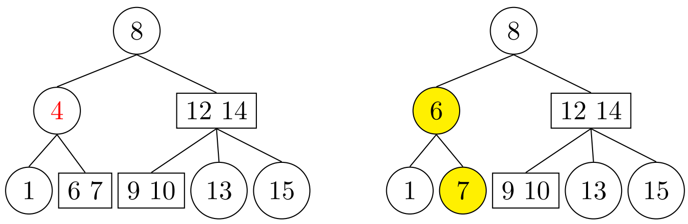


图 10.133 2-3-tree-delete

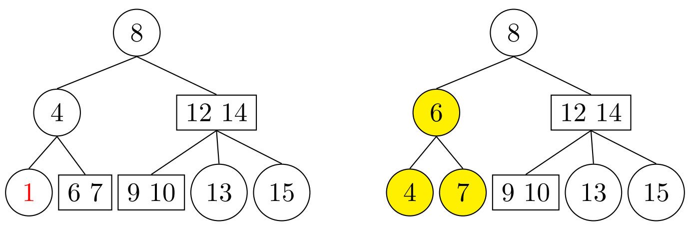


图 10.134 2-3-tree-delete

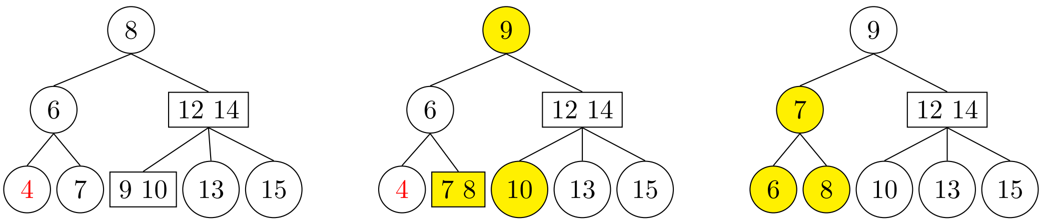


图 10.135 2-3-tree-delete

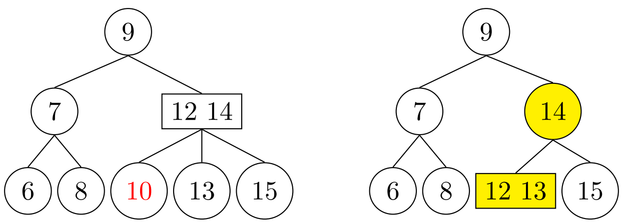


图 10.136 2-3-tree-delete

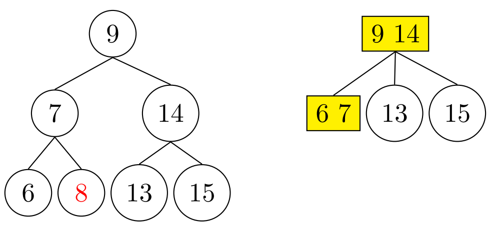


图 10.137 2-3-tree-delete

并行操作（Parallel operations）

由于 2-3 树在结构上与红黑树相似，因此红黑树的并行算法也可以应用于 2-3 树。

2-3 树和左偏红黑树

2-3 树和左偏红黑树实质是等价的。2-3 树中一个节点可以存储 1 个元素或 2 个元素，而红黑树的一个节点只能存储一个元素。如下图所示，2-3 树的 2 节点对应一个黑色节点，3 节点对应一个红色节点和一个黑色节点（可以将 bc 视作平行）。

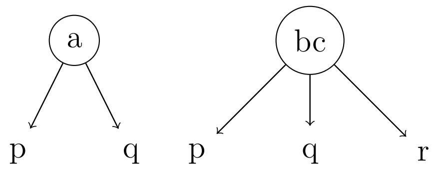


图 10.138 2-3-tree-rbt

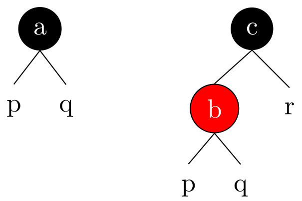


图 10.139 2-3-tree-rbt

下图是一棵 2-3 树对应的左偏红黑树。

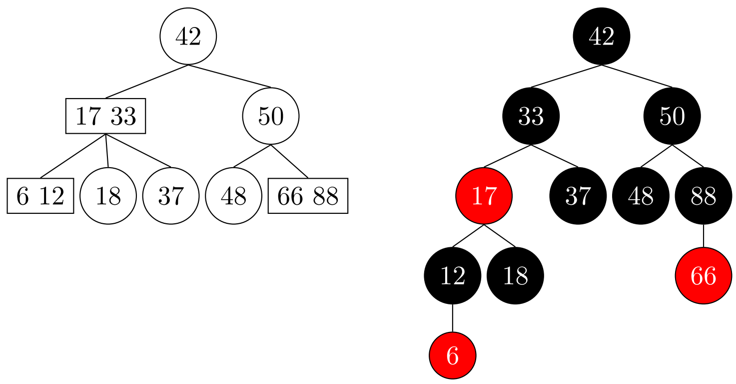


图 10.140 2-3-tree-rbt

参考资料

1. 2–3 tree[1]
2. 17、2 - 3 树[2]
3. 多路查找树 ---2-3 树和 2-3-4 树的深入理解[3]

参考资料与注释

[1] 2–3 tree

[2] 17、2 - 3 树

[3] 多路查找树 —2-3 树和 2-3-4 树的深入理解

10.17.16 2-3-4 树

本页介绍 2-3-4 树。在计算机科学中，2–3–4 树（也称为 2–4 树）是一种自平衡的树，所有的叶子节点都位于同一深度。2-3-4 树是 4 阶 B 树，与一般的 B 树一样，2-3-4 树可以实现在 $O(\log n)$ 时间内进行搜索、插入和删除操作。

2-3-4 树是对 2-3 树的概念扩展，包括了 4 节点的使用。

2–3–4 树与红黑树是等价的。对于每个 2–3–4 树，恰好存在一个相同顺序的红黑树。此外，对 2–3–4 树进行插入和删除操作，导致节点扩展、分裂和合并的过程与红黑树中的颜色翻转和旋转等效。2–3–4 树的操作中涉及大量特殊情况，在大多数编程语言中的实现可能会比较困难。而红黑树的实现较为简单，因此更常被使用。

定义

2-3-4 树的节点分为三种，2 节点，3 节点和 4 节点，分别包含一个、两个或三个数据元素。

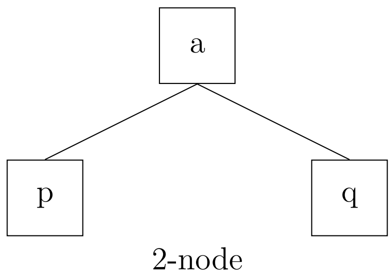


图 10.141 2-3-4-tree-2-node

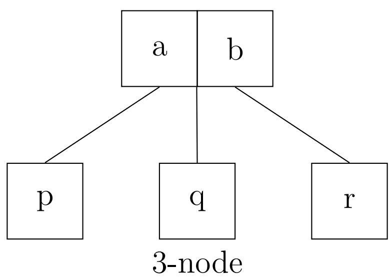


图 10.142 2-3-4-tree-3-node

性质

- 每个节点（叶节点或内部节点）可以是 2 节点、3 节点或 4 节点，分别包含一个、两个或三个数据元素。
- 所有的叶子节点都处于同一深度（最底层）。

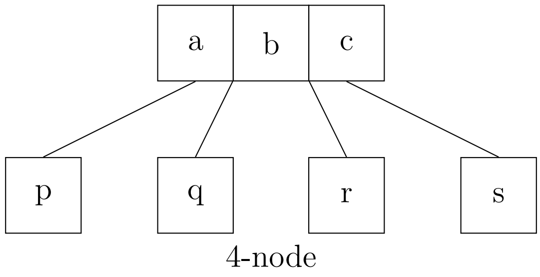


图 10.143 2-3-4-tree-4-node

- 所有数据都有序存储。
- 在 2-3-4 树的最坏情况下高度是 $\log N$，在最好的情况下高度是 $1/2 \log N$（所有节点都是 4 节点）。

操作

由于 2-3-4 树上的元素是按一定顺序存储的，所以查找与二叉搜索树类似。插入和删除都会保持平衡性，即叶节点的深度相同。

插入

2-3-4 树的插入只会发生在叶节点，而不会发生在内部节点。在叶节点上插入一个值，会让 2 节点变成 3 节点，3 节点变成 4 节点，而 4 节点就没有空间了。为了解决这个问题，这里有两种思路。

- 自底向上，从 4 叶节点开始分裂（**上溢**），如果分裂后其父节点也是 4 节点，继续向上分裂，直到到达根节点。如果根节点也是 4 节点，分裂后树的高度 + 1。
- 自顶向下，从根节点到插入所在的叶节点路径上，遇到 4 节点就将其分裂。

本文采用自底向上，将插入操作分为两部分，首先根据搜索树的性质在叶节点的位置插入，形成暂时的 **5 节点**，然后根据插入节点及相关节点的状态进行平衡的维护。

下面是 2-3-4 树插入的例子。

首先插入值 25。从根节点 $(10, 20)$ 开始查找，找到包括 25 的区间 $(20, \infty)$，进入右子节点 $(22, 24, 29)$。

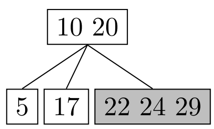


图 10.144 2-3-4-tree-insert-1

由于节点 $22, 24, 29$ 是一个 4 节点，先将 25 插入形成暂时的 **5 节点**。

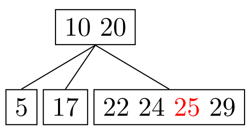


图 10.145 2-3-4-tree-insert-2

由于 $22, 24, 25, 29$ 超出了 4 节点，需要将第二个元素上溢。即将 24 与父节点合并形成 $(10, 20, 24)$。

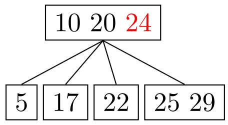


图 10.146 2-3-4-tree-insert-3

然后插入 30。

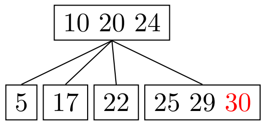


图 10.147 2-3-4-tree-insert-4

插入 31。形成 5 节点 $(25, 29, 30, 31)$。

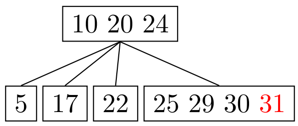


图 10.148 2-3-4-tree-insert-5

第二个元素 29 上溢，发现父节点也形成 5 节点 $(10, 20, 24, 29)$。

5 节点 $(10, 20, 24, 29)$ 中第二个元素 20 继续上溢，最终调整完成。

删除

从 2-3-4 树中删除一个元素需要根据待删除元素的位置分情况讨论。**删除操作只发生在叶节点上**。下面介绍从 2–3–4 树中删除一个元素的过程：

1. 查找需要删除某个元素的节点。
2. 如果节点是叶子节点（3 节点或 4 节点），则从该节点中删除所需的值，并将数据元素减少 1。
3. 如果节点不是叶子节点，则：
    - 查找该节点的后继节点。节点的后继是其中大于它的最小元素或小于它的最大元素。
    - 用后继节点交换当前节点，并在叶子中删除该节点。

但如果叶节点是一个 2 节点，删除该节点可能会导致下溢（underflow）。为了避免这种情况，我们在从上到下移动到要删除的节点的路径上遇到 2 节点时，会先执行以下调整操作，使找到的叶子节点不是 2 节点。这样，在之后的交换和删除后，不会出现一个空的叶节点。

Case 1

如果当前节点的兄弟节点之一是 3 节点或 4 节点，则将当前节点与那个兄弟节点一起进行以下旋转操作：

1. 选择一个与当前节点最接近的键值的兄弟节点。

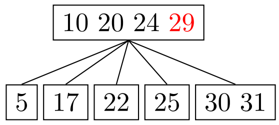


图 10.149 2-3-4-tree-insert-6

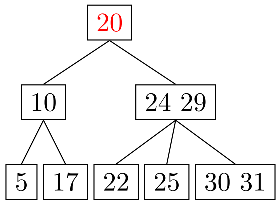


图 10.150 2-3-4-tree-insert-7

2. 将该兄弟节点的键值上移到当前节点的父节点中，使父节点变成一个 3 节点。
3. 原来兄弟节点的孩子现在成为当前节点的孩子。

下图在 2-3-4 树中删除元素 10。

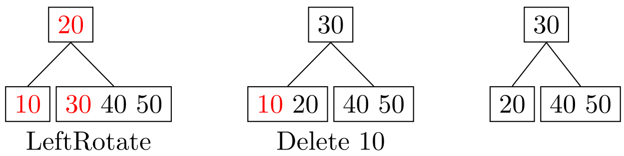


图 10.151 2-3-4-tree-delete-1

Case 2

如果父节点是 2 节点并且兄弟节点也是 2 节点。在这种情况下，父节点是根节点。所以将这三个 2 节点合并成一个新的 4 节点，并缩短树的高度。

下图在 2-3-4 树中删除元素 50。

Case 3

如果兄弟节点是 2 节点，但父节点是 3 节点或 4 节点：

1. 将相邻的兄弟节点和俯视两个兄弟节点的父键融合成一个 4 节点（下溢）。
2. 将兄弟节点的子节点移到该节点。

下图在 2-3-4 树中删除元素 50。

与红黑树的关系

2-3-4 树和红黑树是同构的，任意一棵红黑树都唯一对应一棵 2-3-4 树。在 2-3-4 树上的插入和删除操作导致节点的扩展、分裂和合并，相当于红黑树中的变色和旋转。下图是 2-3-4 树的 2 节点、3 节点和 4 节点对应的红黑树节点。注意到 2-3-4 树的 3 节点对应红黑树中红色节点左偏和右偏两种情况，所以一棵红黑树可能对应多棵 2-3-4 树。

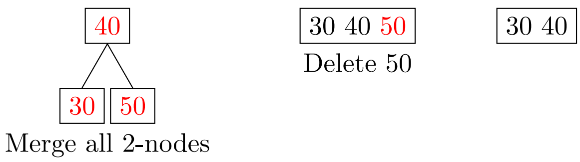


图 10.152 2-3-4-tree-delete-2

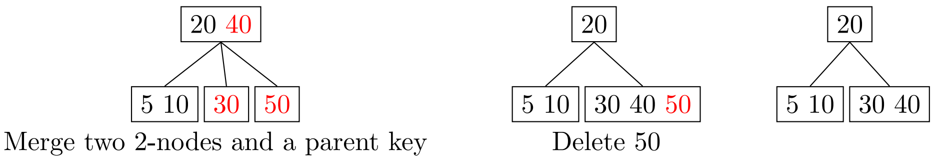


图 10.153 2-3-4-tree-delete-3

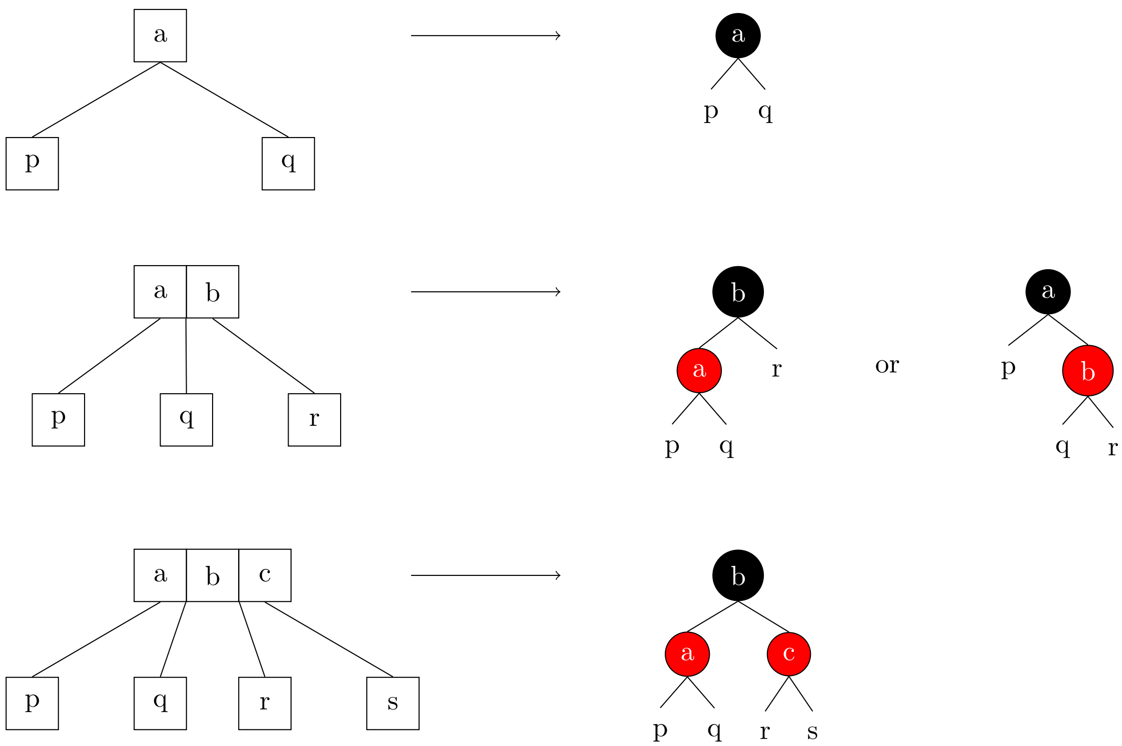


图 10.154 2-3-4-tree-rbt-1

下图是一棵红黑树和与之对应的 2-3-4 树。将红黑树中的红色节点上移到父节点的左右两侧，形成一个 B 树节点，就可以得到与之对应的 2-3-4 树。可以发现，红黑树的节点数等于 2-3-4 树的节点个数。

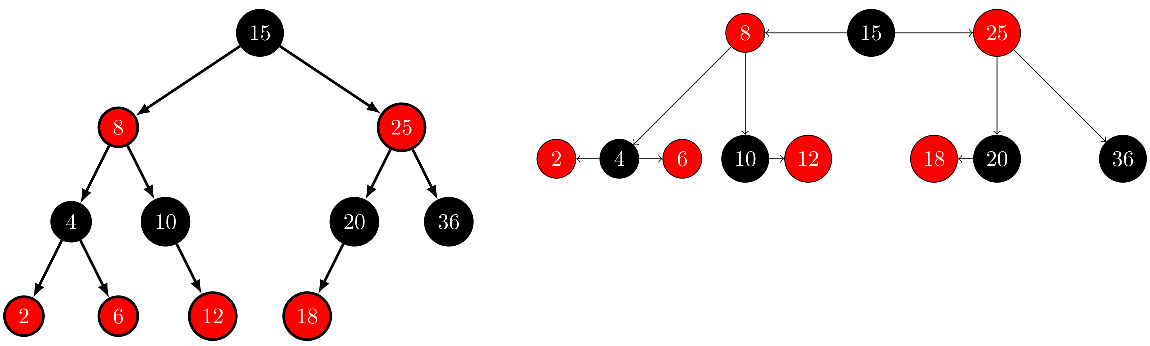


图 10.155 2-3-4-tree-rbt

下面用上图对比红黑树和 2-3-4 树的插入和删除操作。

红黑树的插入

下面根据插入节点，父节点以及祖父节点的情况将插入分为以下 3 类情况，每类情况均包含 4 种。

- 插入节点的父节点是黑色，可以直接插入，无需调整。
- 插入节点的叔父节点但不为红色（不存在或者为黑色），需要进行变色 + 旋转（单旋或双旋）。
- 插入节点的叔父节点是红色，需要进行变色并将祖父节点上溢，递归处理。

Case 1

插入节点的父节点是黑色，可以直接插入，不需要进行调整。这种情况插入新节点后形成 2-3-4 树的 3 节点或 4 节点。如下图插入 9 22 28 40 都属于这种情况。

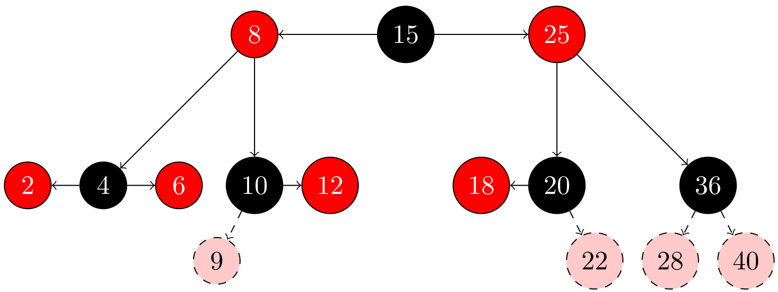


图 10.156 2-3-4-tree-rbt

Case 2

插入节点的父节点是红色，祖父节点是黑色，叔父节点不为红色（不存在或者为黑色）。这种情况插入后形成 2-3-4 树的 4 节点，但是违反红黑树的性质（出现两个连续红色节点）。按照父节点和祖父节点的位置可以分成 LL、RR、LR、RL 四种情况。其中 LL 和 RR 需要进行一次染色和单旋（左旋或右旋）。LR 和 RL 则需要进行染色和双旋。

如下图插入节点 11 13 16 19 都属于这种情况。其中插入节点 13 和 16 为 RR 型和 LL 型，11 和 19 为 RL 型和 LR 型。

以插入节点 13 举例，按照 2-3-4 树的插入方式中应将 13 插入 12 的右侧，但出现两个连续红色节点 12 和 13，因此需要先将父节点 12 染成黑色，将祖父节点 10 染成红色并左旋祖父节点 10，调整后满足红黑树的性质。调整后

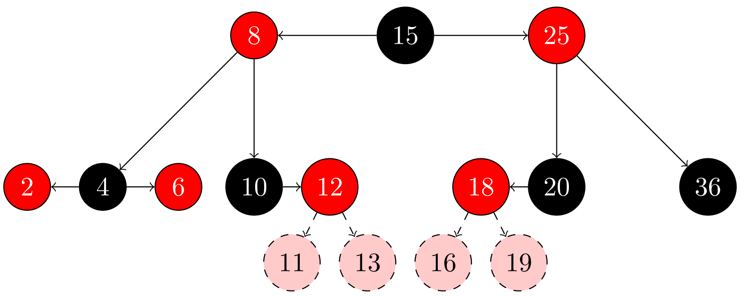


图 10.157 2-3-4-tree-rbt

的结果如下图所示。

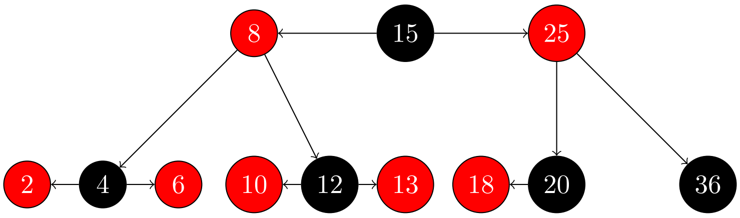


图 10.158 2-3-4-tree-rbt

Case 3

插入节点的父节点和叔父节点是红色，祖父节点是黑色。这种情况其实是形成了一个 2-3-4 树的 5 节点。需要将父节点和叔父节点染为黑色，将祖父节点染为红色并上溢到父节点，然后递归处理。如下图插入节点 1、3、5、7 都属于这种情况。

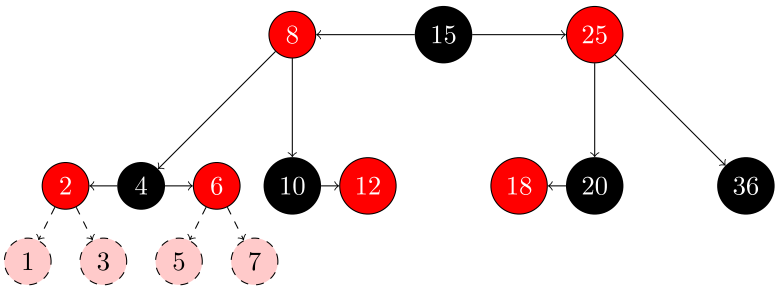


图 10.159 2-3-4-tree-rbt

下图以插入节点 1 为例，形成的 5 节点为 $(1, 2, 4, 6)$。将父节点 2 和叔父节点 6 染为黑色，将祖父节点 4 染为红色并上溢。可以发现节点 4 上溢后仍需要按照 Case 3 处理，此处省略具体步骤。其他三种情况的调整方式与之类似。

红黑树的删除

红黑树内部节点的删除，会转换为其前驱或后继节点的删除。所以红黑树节点的删除，最终都会转换为 2-3-4 树中最后一行节点的删除（对应红黑树最后两行节点的删除）。

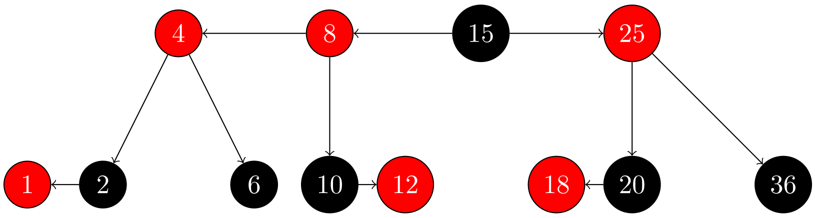


图 10.160 2-3-4-tree-rbt

Case 1

待删除节点为红色，可以直接删除，因为删除最后一层的红色节点不会违反红黑树的性质。

Case 2

待删除的节点为黑色，需要进行调整才能满足红黑树的黑平衡。其中黑色节点可以分为以下三种情况。

Case 2.1

待删除的黑色节点有两个红色子节点的黑色节点（2-3-4 树的 4 节点）。删除该节点会转换为其前驱或后继节点的删除，所以此处不做考虑。

Case 2.2

待删除的黑色节点有一个红色子节点的黑色节点（2-3-4 树的 3 节点）。需要用唯一的红色子节点替代被删除的节点，并将该红色子节点染为黑色。如下图删除节点 10，需要用唯一的红色子节点 12 替代，并将其染为黑色。

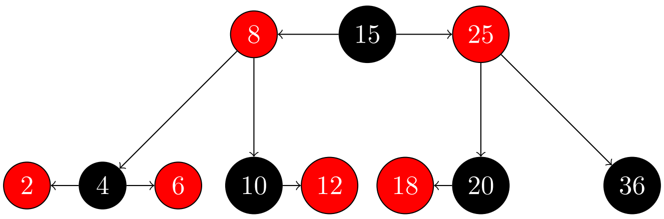


图 10.161 2-3-4-tree-rbt

调整完的结果如下图所示。

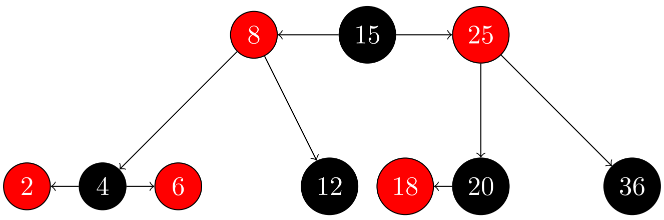


图 10.162 2-3-4-tree-rbt

Case 2.3

待删除的黑色节点为黑色叶节点（对应 2-3-4 树删除叶子 2 节点导致下溢的情况），需要进行调整，具体可以分为以下三种情况。

Case 2.3.1

待删除节点为根节点，当前只有一个节点，可以直接删除。

Case 2.3.2

待删除节点的兄弟节点为黑色。

- 兄弟节点有红色子节点（分为三种，有一个左红色子节点，或有一个右红色子节点，或两个红色子节点），需要借用兄弟节点的子节点进行调整。下图以删除节点 36，其有一个红色的左子节点 18 为例，这种情况需要进行旋转和染色。观察到待删除节点的父节点、兄弟节点、兄弟节点的子节点的关系为 LL 型，只需要右旋一次，并将中间节点 20 染为红色，两个子节点 18 和 25 染为黑色。

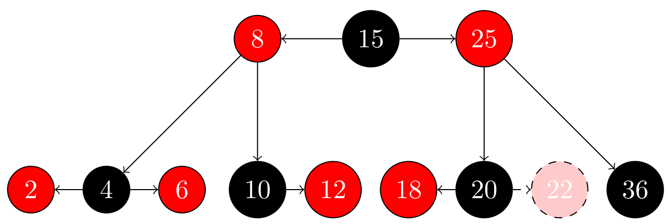


图 10.163 2-3-4-tree-rbt

下图是调整后的结果。其他两种情况，有一个红色的右子节点的情况需要进行双旋和染色，有两个红色的子节点的情况可以任选其中一个进行调整。

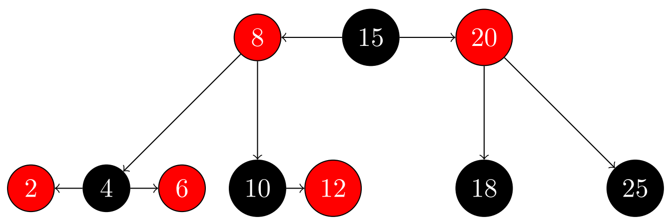


图 10.164 2-3-4-tree-rbt

- 兄弟节点没有红色子节点，需要将兄弟节点染红，父节点染黑。如果父节点原来就为黑色，需要把父节点当作已被删除的节点处理，然后向上递归调整。

  如下图所示，待删除节点 36，其兄弟节点 20 没有红色子节点，父节点 25 为红色，需要将父节点下溢（对应 2-3-4 树中删除叶子 2 节点将父节点下溢的情况，在红黑树中无需操作）并染黑，将 20 染红。

  下图是待删除节点的父节点原来就为黑色的情况。调整后由于父节点 25 发生了下溢，需要把父节点 25 当作一个删除的节点向上递归调整。

Case 2.3.3

待删除节点的兄弟节点为红色，需要转变为兄弟节点为黑色的情况进行处理。具体操作是将父节点旋转，将兄弟节点染黑，父节点染红，然后转换成待删除节点的兄弟节点为黑色的情况继续进行调整。如下图所示，待删除待删除节点 36，其兄弟节点 15 为红色，根据位置关系可知需要将父节点右旋，并将父节点染红，兄弟节点染黑，转换成待删除节点的兄弟节点为黑色的情况。

参考资料

1. 2–3–4 tree - Wikipedia[1]

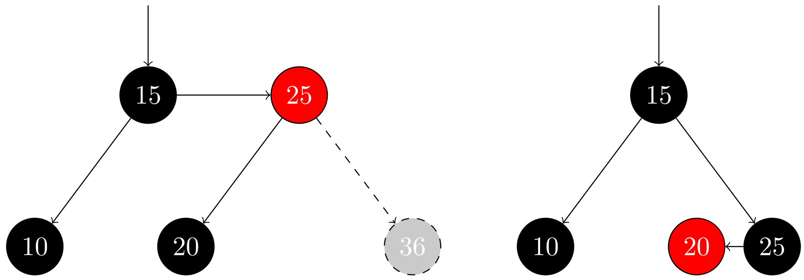


图 10.165 2-3-4-tree-rbt

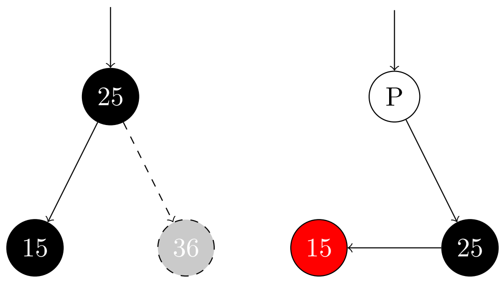


图 10.166 2-3-4-tree-rbt

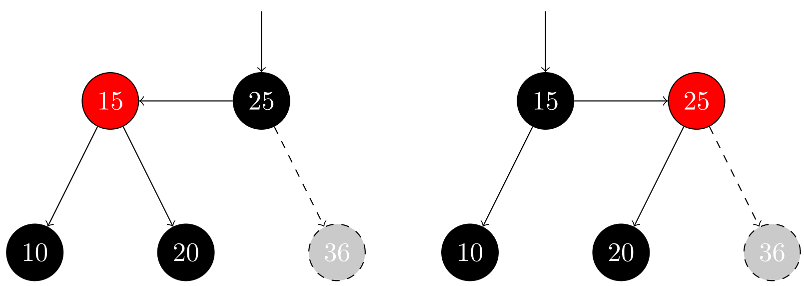


图 10.167 2-3-4-tree-rbt

2. 2-3-4 Trees | Algorithm Tutor[2]
3. 2-3-4 Tree - GeeksforGeeks[3]

参考资料与注释

[1] 2–3–4 tree - Wikipedia

[2] 2-3-4 Trees | Algorithm Tutor

[3] 2-3-4 Tree - GeeksforGeeks

10.18 跳表

跳表 (Skip List) 是由 William Pugh 发明的一种查找数据结构，支持对数据的快速查找，插入和删除。

跳表的期望空间复杂度为 $O(n)$，跳表的查询，插入和删除操作的期望时间复杂度都为 $O(\log n)$。

基本思想

顾名思义，跳表是一种类似于链表的数据结构。更加准确地说，跳表是对有序链表的改进。

为方便讨论，后续所有有序链表默认为**升序**排序。

一个有序链表的查找操作，就是从头部开始逐个比较，直到当前节点的值大于或者等于目标节点的值。很明显，这个操作的复杂度是 $O(n)$。

跳表在有序链表的基础上，引入了**分层**的概念。首先，跳表的每一层都是一个有序链表，特别地，最底层是初始的有序链表。每个位于第 i 层的节点有 p 的概率出现在第 $i+1$ 层，p 为常数。

记在 n 个节点的跳表中，期望包含 $\frac{1}{p}$ 个元素的层为第 $L(n)$ 层，易得 $L(n)=\log_{\frac{1}{p}} n$。

在跳表中查找，就是从第 $L(n)$ 层开始，水平地逐个比较直至当前节点的下一个节点大于等于目标节点，然后移动至下一层。重复这个过程直至到达第一层且无法继续进行操作。此时，若下一个节点是目标节点，则成功查找；反之，则元素不存在。这样一来，查找的过程中会跳过一些没有必要的比较，所以相比于有序链表的查询，跳表的查询更快。可以证明，跳表查询的平均复杂度为 $O(\log n)$。

复杂度证明

空间复杂度

对于一个节点而言，节点的最高层数为 i 的概率为 $p^{i-1}(1-p)$。所以，跳表的期望层数为 $\sum_{i>=1} ip^{i-1}(1-p)=\frac{1}{1-p}$，且因为 p 为常数，所以跳表的**期望空间复杂度**为 $O(n)$。

在最坏的情况下，每一层有序链表等于初始有序链表，即跳表的**最差空间复杂度**为 $O(n\log n)$。

时间复杂度

从后向前分析查找路径，这个过程可以分为从最底层爬到第 $L(n)$ 层和后续操作两个部分。在分析时，假设一个节点的具体信息在它被访问之前是未知的。

假设当前我们处于一个第 i 层的节点 x，我们并不知道 x 的最大层数和 x 左侧节点的最大层数，只知道 x 的最大层数至少为 i。如果 x 的最大层数大于 i，那么下一步应该是向上走，这种情况的概率为 p；如果 x 的最大层数等于 i，那么下一步应该是向左走，这种情况概率为 $1-p$。

令 $C(i)$ 为在一个无限长度的跳表中向上爬 i 层的期望代价，那么有：

$$\begin{aligned} C(0) &= 0 \\ C(i) &= (1-p)(1+C(i)) + p(1+C(i-1)) \end{aligned}$$

解得 $C(i) = \frac{i}{p}$。

由此可以得出：在长度为 n 的跳表中，从最底层爬到第 $L(n)$ 层的期望步数存在上界 $\frac{L(n)-1}{p}$。

现在只需要分析爬到第 $L(n)$ 层后还要再走多少步。易得，到了第 $L(n)$ 层后，向左走的步数不会超过第 $L(n)$ 层及更高层的节点数总和，而这个总和的期望为 $\frac{1}{p}$。所以到了第 $L(n)$ 层后向左走的期望步数存在上界 $\frac{1}{p}$。同理，到了第 $L(n)$ 层后向上走的期望步数存在上界 $\frac{1}{p}$。

所以，跳表查询的期望查找步数为 $\frac{L(n)-1}{p} + \frac{2}{p}$，又因为 $L(n) = \log_{\frac{1}{p}} n$，所以跳表查询的**期望时间复杂度**为 $O(\log n)$。

在最坏的情况下，每一层有序链表等于初始有序链表，查找过程相当于对最高层的有序链表进行查询，即跳表查询操作的**最差时间复杂度**为 $O(n)$。

插入操作和删除操作就是进行一遍查询的过程，途中记录需要修改的节点，最后完成修改。易得每一层至多只需要修改一个节点，又因为跳表期望层数为 $\log_{\frac{1}{p}} n$，所以插入和修改的**期望时间复杂度**也为 $O(\log n)$。

具体实现

获取节点的最大层数

模拟以 p 的概率往上加一层，最后和上限值取最小。

```
int randomLevel() {
  int lv = 1;
  // MAXL = 32, S = 0xFFFF, PS = S * P, P = 1 / 4
  while ((rand() & S) < PS) ++lv;
  return min(MAXL, lv);
}
```

查询

查询跳表中是否存在键值为 `key` 的节点。具体实现时，可以设置两个哨兵节点以减少边界条件的讨论。

```
V& find(const K& key) {
  SkipListNode<K, V>* p = head;

  // 找到该层最后一个键值小于 key 的节点，然后走向下一层
  for (int i = level; i >= 0; --i) {
    while (p->forward[i]->key < key) {
      p = p->forward[i];
    }
  }
  // 现在是小于，所以还需要再往后走一步
  p = p->forward[0];

  // 成功找到节点
  if (p->key == key) return p->value;

  // 节点不存在，返回 INVALID
```

```
  return tail->value;
}
```

插入

插入节点 (key, value)。插入节点的过程就是先执行一遍查询的过程，中途记录新节点是要插入哪一些节点的后面，最后再执行插入。每一层最后一个键值小于 key 的节点，就是需要进行修改的节点。

```
void insert(const K &key, const V &value) {
  // 用于记录需要修改的节点
  SkipListNode<K, V> *update[MAXL + 1];

  SkipListNode<K, V> *p = head;
  for (int i = level; i >= 0; --i) {
    while (p->forward[i]->key < key) {
      p = p->forward[i];
    }
    // 第 i 层需要修改的节点为 p
    update[i] = p;
  }
  p = p->forward[0];

  // 若已存在则修改
  if (p->key == key) {
    p->value = value;
    return;
  }

  // 获取新节点的最大层数
  int lv = randomLevel();
  if (lv > level) {
    lv = ++level;
    update[lv] = head;
  }

  // 新建节点
  SkipListNode<K, V> *newNode = new SkipListNode<K, V>(key, value, lv);
  // 在第 0~lv 层插入新节点
  for (int i = lv; i >= 0; --i) {
    p = update[i];
    newNode->forward[i] = p->forward[i];
    p->forward[i] = newNode;
  }

  ++length;
}
```

删除

删除键值为 key 的节点。删除节点的过程就是先执行一遍查询的过程，中途记录要删的节点是在哪一些节点的后面，最后再执行删除。每一层最后一个键值小于 key 的节点，就是需要进行修改的节点。

```
bool erase(const K &key) {
  // 用于记录需要修改的节点
```

```
  SkipListNode<K, V> *update[MAXL + 1];

  SkipListNode<K, V> *p = head;
  for (int i = level; i >= 0; --i) {
    while (p->forward[i]->key < key) {
      p = p->forward[i];
    }
    // 第 i 层需要修改的节点为 p
    update[i] = p;
  }
  p = p->forward[0];

  // 节点不存在
  if (p->key != key) return false;

  // 从最底层开始删除
  for (int i = 0; i <= level; ++i) {
    // 如果这层没有 p 删除就完成了
    if (update[i]->forward[i] != p) {
      break;
    }
    // 断开 p 的连接
    update[i]->forward[i] = p->forward[i];
  }

  // 回收空间
  delete p;

  // 删除节点可能导致最大层数减少
  while (level > 0 && head->forward[level] == tail) --level;

  // 跳表长度
  --length;
  return true;
}
```

完整代码

下列代码是用跳表实现的 map。未经正经测试，仅供参考。

" 参考代码"

```
#include <bits/stdc++.h>
using namespace std;

template <typename K, typename V>
struct SkipListNode {
  int level;
  K key;
  V value;
  SkipListNode **forward;

  SkipListNode() {}
```

```
  SkipListNode(K k, V v, int l, SkipListNode *nxt = NULL) {
    key = k;
    value = v;
    level = l;
    forward = new SkipListNode *[l + 1];
    for (int i = 0; i <= l; ++i) forward[i] = nxt;
  }

  ~SkipListNode() {
    if (forward != NULL) delete[] forward;
  }
};

template <typename K, typename V>
struct SkipList {
  static const int MAXL = 32;
  static const int P = 4;
  static const int S = 0xFFFF;
  static const int PS = S / P;
  static const int INVALID = INT_MAX;

  SkipListNode<K, V> *head, *tail;
  int length;
  int level;

  SkipList() {
    srand(time(0));

    level = length = 0;
    tail = new SkipListNode<K, V>(INVALID, 0, 0);
    head = new SkipListNode<K, V>(INVALID, 0, MAXL, tail);
  }

  ~SkipList() {
    delete head;
    delete tail;
  }

  int randomLevel() {
    int lv = 1;
    while ((rand() & S) < PS) ++lv;
    return MAXL > lv ? lv : MAXL;
  }

  void insert(const K &key, const V &value) {
    SkipListNode<K, V> *update[MAXL + 1];

    SkipListNode<K, V> *p = head;
    for (int i = level; i >= 0; --i) {
      while (p->forward[i]->key < key) {
        p = p->forward[i];
      }
      update[i] = p;
    }
```

```
    p = p->forward[0];

    if (p->key == key) {
      p->value = value;
      return;
    }

    int lv = randomLevel();
    if (lv > level) {
      lv = ++level;
      update[lv] = head;
    }

    SkipListNode<K, V> *newNode = new SkipListNode<K, V>(key, value, lv);
    for (int i = lv; i >= 0; --i) {
      p = update[i];
      newNode->forward[i] = p->forward[i];
      p->forward[i] = newNode;
    }

    ++length;
  }

  bool erase(const K &key) {
    SkipListNode<K, V> *update[MAXL + 1];
    SkipListNode<K, V> *p = head;

    for (int i = level; i >= 0; --i) {
      while (p->forward[i]->key < key) {
        p = p->forward[i];
      }
      update[i] = p;
    }
    p = p->forward[0];

    if (p->key != key) return false;

    for (int i = 0; i <= level; ++i) {
      if (update[i]->forward[i] != p) {
        break;
      }
      update[i]->forward[i] = p->forward[i];
    }

    delete p;

    while (level > 0 && head->forward[level] == tail) --level;
    --length;
    return true;
  }

  V &operator[](const K &key) {
    V v = find(key);
    if (v == tail->value) insert(key, 0);
```

```
    return find(key);
  }

  V &find(const K &key) {
    SkipListNode<K, V> *p = head;
    for (int i = level; i >= 0; --i) {
      while (p->forward[i]->key < key) {
        p = p->forward[i];
      }
    }
    p = p->forward[0];
    if (p->key == key) return p->value;
    return tail->value;
  }

  bool count(const K &key) { return find(key) != tail->value; }
};

int main() {
  SkipList<int, int> L;
  map<int, int> M;

  clock_t s = clock();

  for (int i = 0; i < 1e5; ++i) {
    int key = rand(), value = rand();
    L[key] = value;
    M[key] = value;
  }

  for (int i = 0; i < 1e5; ++i) {
    int key = rand();
    if (i & 1) {
      L.erase(key);
      M.erase(key);
    } else {
      int r1 = L.count(key) ? L[key] : 0;
      int r2 = M.count(key) ? M[key] : 0;
      assert(r1 == r2);
    }
  }

  clock_t e = clock();
  cout << "Time elapsed: " << (double)(e - s) / CLOCKS_PER_SEC << endl;
  // about 0.2s

  return 0;
}
```

跳表的随机访问优化

访问跳表中第 k 个节点，相当于访问初始有序链表中的第 k 个节点，很明显这个操作的时间复杂度是 $O(n)$ 的，并不足够优秀。

跳表的随机访问优化就是对每一个前向指针，再多维护这个前向指针的长度。假设 A 和 B 都是跳表中的节点，

其中 A 为跳表的第 a 个节点，B 为跳表的第 b 个节点 $(a < b)$，且在跳表的某一层中 A 的前向指针指向 B，那么这个前向指针的长度为 $b - a$。

现在访问跳表中的第 k 个节点，就可以从顶层开始，水平地遍历该层的链表，直到当前节点的位置加上当前节点在该层的前向指针长度大于等于 k，然后移动至下一层。重复这个过程直至到达第一层且无法继续行操作。此时，当前节点就是跳表中第 k 个节点。

这样，就可以快速地访问到跳表的第 k 个元素。可以证明，这个操作的时间复杂度为 $O(\log n)$。

参考资料

1. Skip Lists: A Probabilistic Alternative to Balanced Trees[1]
2. Skip List[2]
3. A Skip List Cookbook[3]

参考资料与注释

[1] Skip Lists: A Probabilistic Alternative to Balanced Trees

[2] Skip List

[3] A Skip List Cookbook

10.19 可持久化数据结构

10.19.1 可持久化数据结构简介

Authors: morris821028

简介

可持久化数据结构 (Persistent data structure) 总是可以保留每一个历史版本，并且支持操作的不可变特性 (immutable)。

可持久化分类

部分可持久化 (Partially Persistent)

所有版本都可以访问，但是只有最新版本可以修改。

完全可持久化 (Fully Persistent)

所有版本都既可以访问又可以修改。

若支持将两个历史版本合并，则又称为 Confluently Persistent

实际应用

几何计算

在几何计算中有许多离线算法，如扫描线算法一次扫过去回答所有询问，在时间复杂度分析上相当优异。但强迫在线的情况下，每一次都扫描一次，询问操作的时间复杂度就从对数时间降成线性。为了解决这一种情况，持久化技

术给了另一种思维，我们将扫描线的时间轴作为一个变动依据，持久化相关的结构，只要我们能将询问在对数时间内穿梭于这个时间轴，必能动态解决先前的问题。

字串处理

为了达到非常高效率的合并操作，防止大量重复性字串的生成伴随的效能退化，使得各方面的操作都能远低于线性操作。如 C++ rope 就是一个持久化的数据结构。不只是字串操作，若处理类型有大量重复的情况，持久化的概念便能派上用场。

版本回溯

实际上就是对应大部分的应用软体中的 redo/undo。如果资料库/操作变动为了高效率操作而会配上复杂的结构（并不像 hash, set 反转操作只需要常数或对数时间），那么为了快速回推变动结果，持久化结构就是要减少 redo/undo 的花费。

资料库本身可以常数回推，纪录变动的部分情况即可。而应用层的计算，大部分实作都是砍掉快取，并且重新计算出一份新的结构，有时候回推的变动大小为 m，为了重新计算结构而消耗了 n+m，如果 n 和 m 的差距非常大，那连续回推的体感就很糟糕。

函数式编程

函数式编程需要特别的数据结构以符合语言特性，其中不可变的性质更为重要，以利于并行环境与除错。如面向对象编程的 Java 8 后引入 stream 类，支援写出函数式的语法设计，可提供惰性求值、无限值域等的特殊功能。

参考

- https://en.wikipedia.org/wiki/Persistent_data_structure[1]
- MIT 课程 https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-854j-advanced-algorithms-fall-2005/lecture-notes/persistent.pdf[2]

参考资料与注释

[1] https://en.wikipedia.org/wiki/Persistent_data_structure

[2] https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-854j-advanced-algorithms-fall-2005/lecture-no

10.19.2 可持久化线段树

主席树

主席树全称是可持久化权值线段树，参见 知乎讨论[1]。

> **" 关于函数式线段树"**
>
> **函数式线段树**是指使用函数式编程思想的线段树。在函数式编程思想中，将计算机运算视为数学函数，并避免可改变的状态或变量。不难发现，函数式线段树是完全可持久化的。

引入

先引入一道题目：给定 n 个整数构成的序列 a，将对于指定的闭区间 $[l,r]$ 查询其区间内的第 k 小值。

你该如何解决？

一种可行的方案是：使用主席树。主席树的主要思想就是：保存每次插入操作时的历史版本，以便查询区间第 k 小。

怎么保存呢？简单暴力一点，每次开一棵线段树呗。

那空间还不爆掉？

解释

我们分析一下，发现每次修改操作修改的点的个数是一样的。

（例如下图，修改了 $[1,8]$ 中对应权值为 1 的结点，红色的点即为更改的点）

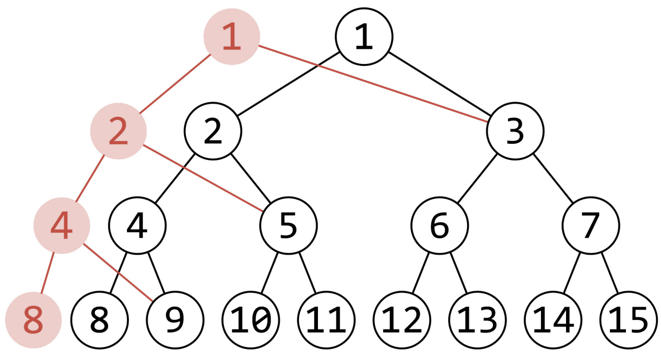


图 10.168

只更改了 $O(\log n)$ 个结点，形成一条链，也就是说每次更改的结点数 = 树的高度。

注意主席树不能使用堆式存储法，就是说不能用 $x \times 2$，$x \times 2 + 1$ 来表示左右儿子，而是应该动态开点，并保存每个节点的左右儿子编号。

所以我们只要在记录左右儿子的基础上，保存插入每个数的时候的根节点就可以实现持久化了。

我们把问题简化一下：每次求 $[1,r]$ 区间内的 k 小值。

怎么做呢？只需要找到插入 r 时的根节点版本，然后用普通权值线段树（有的叫键值线段树/值域线段树）做就行了。

这个相信大家都能理解，回到原问题——求 $[l,r]$ 区间 k 小值。

这里我们再联系另外一个知识：**前缀和**。

这个小东西巧妙运用了区间减法的性质，通过预处理从而达到 $O(1)$ 回答每个询问。

我们可以发现，主席树统计的信息也满足这个性质。

所以……如果需要得到 $[l,r]$ 的统计信息，只需要用 $[1,r]$ 的信息减去 $[1,l-1]$ 的信息就行了。

至此，该问题解决！

关于空间问题，我们分析一下：由于我们是动态开点的，所以一棵线段树只会出现 $2n-1$ 个结点。

然后，有 n 次修改，每次至多增加 $\lceil \log_2 n \rceil + 1$ 个结点。因此，最坏情况下 n 次修改后的结点总数会达到 $2n - 1 + n(\lceil \log_2 n \rceil + 1)$。此题的 $n \le 10^5$，单次修改至多增加 $\lceil \log_2 10^5 \rceil + 1 = 18$ 个结点，故 n 次修改后的结点总数为 $2 \times 10^5 - 1 + 18 \times 10^5$，忽略掉 -1，大概就是 20×10^5。

最后给一个忠告：千万不要吝啬空间（大多数题目中空间限制都较为宽松，因此一般不用担心空间超限的问题）！大胆一点，直接上个 $2^5 \times 10^5$，接近原空间的两倍（即 `n << 5`）。

实现

```
#include <algorithm>
#include <cstdio>
#include <cstring>
```

```cpp
using namespace std;
const int maxn = 1e5;  // 数据范围
int tot, n, m;
int sum[(maxn << 5) + 10], rt[maxn + 10], ls[(maxn << 5) + 10],
    rs[(maxn << 5) + 10];
int a[maxn + 10], ind[maxn + 10], len;

int getid(const int &val) {  // 离散化
  return lower_bound(ind + 1, ind + len + 1, val) - ind;
}

int build(int l, int r) {  // 建树
  int root = ++tot;
  if (l == r) return root;
  int mid = l + r >> 1;
  ls[root] = build(l, mid);
  rs[root] = build(mid + 1, r);
  return root;  // 返回该子树的根节点
}

int update(int k, int l, int r, int root) {  // 插入操作
  int dir = ++tot;
  ls[dir] = ls[root], rs[dir] = rs[root], sum[dir] = sum[root] + 1;
  if (l == r) return dir;
  int mid = l + r >> 1;
  if (k <= mid)
    ls[dir] = update(k, l, mid, ls[dir]);
  else
    rs[dir] = update(k, mid + 1, r, rs[dir]);
  return dir;
}

int query(int u, int v, int l, int r, int k) {  // 查询操作
  int mid = l + r >> 1,
      x = sum[ls[v]] - sum[ls[u]];  // 通过区间减法得到左儿子中所存储的数值个数
  if (l == r) return l;
  if (k <= x)  // 若 k 小于等于 x ，则说明第 k 小的数字存储在在左儿子中
    return query(ls[u], ls[v], l, mid, k);
  else  // 否则说明在右儿子中
    return query(rs[u], rs[v], mid + 1, r, k - x);
}

void init() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= n; ++i) scanf("%d", a + i);
  memcpy(ind, a, sizeof ind);
  sort(ind + 1, ind + n + 1);
  len = unique(ind + 1, ind + n + 1) - ind - 1;
  rt[0] = build(1, len);
  for (int i = 1; i <= n; ++i) rt[i] = update(getid(a[i]), 1, len, rt[i - 1]);
}

int l, r, k;
```

```
void work() {
  while (m--) {
    scanf("%d%d%d", &l, &r, &k);
    printf("%d\n", ind[query(rt[l - 1], rt[r], 1, len, k)]);  // 回答询问
  }
}

int main() {
  init();
  work();
  return 0;
}
```

拓展：基于主席树的可持久化并查集

主席树是实现可持久化并查集的便捷方式，在此也提供一个基于主席树的可持久化并查集实现示例。

```
#include <bits/stdc++.h>
using namespace std;

struct SegmentTree {
  int lc, rc, val, rnk;
};

const int MAXN = 100000 + 5;
const int MAXM = 200000 + 5;

SegmentTree
    t[MAXN * 2 +
      MAXM * 40];  //每次操作 1 会修改两次，一次修改父节点，一次修改父节点的秩
int rt[MAXM];
int n, m, tot;

int build(int l, int r) {
  int p = ++tot;
  if (l == r) {
    t[p].val = l;
    t[p].rnk = 1;
    return p;
  }
  int mid = (l + r) / 2;
  t[p].lc = build(l, mid);
  t[p].rc = build(mid + 1, r);
  return p;
}

int getRnk(int p, int l, int r, int pos) {  //查询秩
  if (l == r) {
    return t[p].rnk;
  }
  int mid = (l + r) / 2;
  if (pos <= mid) {
    return getRnk(t[p].lc, l, mid, pos);
  } else {
    return getRnk(t[p].rc, mid + 1, r, pos);
```

```
  }
}

int modifyRnk(int now, int l, int r, int pos, int val) {  //修改秩（高度）
  int p = ++tot;
  t[p] = t[now];
  if (l == r) {
    t[p].rnk = max(t[p].rnk, val);
    return p;
  }
  int mid = (l + r) / 2;
  if (pos <= mid) {
    t[p].lc = modifyRnk(t[now].lc, l, mid, pos, val);
  } else {
    t[p].rc = modifyRnk(t[now].rc, mid + 1, r, pos, val);
  }
  return p;
}

int query(int p, int l, int r, int pos) {  //查询父节点（序列中的值）
  if (l == r) {
    return t[p].val;
  }
  int mid = (l + r) / 2;
  if (pos <= mid) {
    return query(t[p].lc, l, mid, pos);
  } else {
    return query(t[p].rc, mid + 1, r, pos);
  }
}

int findRoot(int p, int pos) {  //查询根节点
  int f = query(p, 1, n, pos);
  if (pos == f) {
    return pos;
  }
  return findRoot(p, f);
}

int modify(int now, int l, int r, int pos, int fa) {  //修改父节点（合并）
  int p = ++tot;
  t[p] = t[now];
  if (l == r) {
    t[p].val = fa;
    return p;
  }
  int mid = (l + r) / 2;
  if (pos <= mid) {
    t[p].lc = modify(t[now].lc, l, mid, pos, fa);
  } else {
    t[p].rc = modify(t[now].rc, mid + 1, r, pos, fa);
  }
  return p;
}
```

```
int main() {
  scanf("%d%d", &n, &m);
  rt[0] = build(1, n);
  for (int i = 1; i <= m; i++) {
    int op, a, b;

    scanf("%d", &op);
    if (op == 1) {
      scanf("%d%d", &a, &b);
      int fa = findRoot(rt[i - 1], a), fb = findRoot(rt[i - 1], b);
      if (fa != fb) {
        if (getRnk(rt[i - 1], 1, n, fa) >
            getRnk(rt[i - 1], 1, n, fb)) {  //按秩合并
          swap(fa, fb);
        }
        int tmp = modify(rt[i - 1], 1, n, fa, fb);
        rt[i] = modifyRnk(tmp, 1, n, fb, getRnk(rt[i - 1], 1, n, fa) + 1);
      } else {
        rt[i] = rt[i - 1];
      }
    } else if (op == 2) {
      scanf("%d", &a);
      rt[i] = rt[a];
    } else {
      scanf("%d%d", &a, &b);
      rt[i] = rt[i - 1];
      if (findRoot(rt[i], a) == findRoot(rt[i], b)) {
        printf("1\n");
      } else {
        printf("0\n");
      }
    }
  }

  return 0;
}
```

参考

https://en.wikipedia.org/wiki/Persistent_data_structure[2]

https://www.cnblogs.com/zinthos/p/3899565.html[3]

参考资料与注释

[1] 知乎讨论

[2] https://en.wikipedia.org/wiki/Persistent_data_structure

[3] https://www.cnblogs.com/zinthos/p/3899565.html

10.19.3 可持久化块状数组

10.19.4 可持久化平衡树

前置知识

OI 常用的可持久化平衡树一般就是**可持久化无旋转 Treap** 所以推荐首先学习 无旋转 Treap。

思想/做法

对于非旋转 Treap，可通过 **Merge** 和 **Split** 操作过程中复制路径上经过的节点（一般在 **Split** 操作中复制，确保不影响以前的版本）就可完成可持久化。

对于旋转 Treap，在复制路径上经过的节点同时，还需复制受旋转影响的节点（若其已为这次操作中复制的节点，则无需再复制），对于一次旋转一般只影响两个节点，那么不会增加其时间复杂度。

上述方法一般被称为 path copying。

「一切可支持操作都可以通过 **Merge** **Split** **Newnode** **Build** 完成」，而 **Build** 操作只用于建造无需理会，**Newnode**（新建节点）就是用来可持久化的工具。

我们来观察一下 **Merge** 和 **Split**，我们会发现它们都是由上而下的操作！

因此我们完全可以**参考线段树的可持久化操作**对它进行可持久化。

可持久化操作

可持久化是对**数据结构**的一种操作，即保留历史信息，使得在后面可以调用之前的历史版本。

对于**可持久化线段树**来说，每一次新建历史版本就是把**沿途的修改路径**复制出来

那么对可持久化 Treap（目前国内 OI 常用的版本）来说：

在复制一个节点 X_a（X 节点的第 a 个版本）的新版本 X_{a+1}（X 节点的第 $a+1$ 个版本）以后：

- 如果某个儿子节点 Y 不用修改信息，那么就把 X_{a+1} 的指针直接指向 Y_a（Y 节点的第 a 个版本）即可。
- 反之，如果要修改 Y，那么就在**递归到下层**时**新建** Y_{a+1}（Y 节点的第 $a+1$ 个版本）这个新节点用于**存储新的信息**，同时把 X_{a+1} 的指针指向 Y_{a+1}（Y 节点的第 $a+1$ 个版本）。

可持久化

需要的东西：

- 一个 `struct` 数组存**每个节点**的信息（一般叫做 `tree` 数组）；（当然写**指针版**平衡树的大佬就可以考虑不用这个数组了）
- 一个**根节点数组**，存每个版本的树根，每次查询版本信息时就从**根数组存的节点**开始；
- `split()` 分裂**从树中分裂出两棵树**
- `merge()` 合并**把两棵树按照随机权值合并**
- `newNode()` 新建一个节点
- `build()` 建树

Split

对于**分裂操作**，每次分裂路径时**新建节点**指向分出来的路径，用 `std::pair` 存新分裂出来的两棵树的根。

`split(x,k)` 返回一个 `std::pair`；

表示把 $_x$ 为根的树的前 k 个元素放在**一棵树**中，剩下的节点构成在另一棵树中，返回这两棵树的根（first 是第一棵树的根，second 是第二棵树的）。

- 如果 x 的**左子树**的 $key \geq k$，那么**直接递归进左子树**，把左子树分出来的第二颗树和当前的 x **右子树**合并。
- 否则递归**右子树**。

```
static std::pair<int, int> _split(int _x, int k) {
  if (_x == 0)
    return std::make_pair(0, 0);
  else {
    int _vs = ++_cnt;  // 新建节点（可持久化的精髓）
    _trp[_vs] = _trp[_x];
    std::pair<int, int> _y;
    if (_trp[_vs].key <= k) {
      _y = _split(_trp[_vs].leaf[1], k);
      _trp[_vs].leaf[1] = _y.first;
      _y.first = _vs;
    } else {
      _y = _split(_trp[_vs].leaf[0], k);
      _trp[_vs].leaf[0] = _y.second;
      _y.second = _vs;
    }
    _trp[_vs]._update();
    return _y;
  }
}
```

Merge

`merge(x,y)` 返回 merge 出的树的根。

同样递归实现。如果 **x 的随机权值 >y 的随机权值**，则 `merge(x_{rc},y)`，否则 `merge(x,y_{lc})`。

```
static int _merge(int _x, int _y) {
  if (_x == 0 || _y == 0)
    return _x ^ _y;
  else {
    if (_trp[_x].fix < _trp[_y].fix) {
      _trp[_x].leaf[1] = _merge(_trp[_x].leaf[1], _y);
      _trp[_x]._update();
      return _x;
    } else {
      _trp[_y].leaf[0] = _merge(_x, _trp[_y].leaf[0]);
      _trp[_y]._update();
      return _y;
    }
  }
}
```

例题

"洛谷 P3835【模版】可持久化平衡树[1]"

你需要实现一个数据结构，要求提供如下操作（最开始时数据结构内无数据）：

1. 插入 x 数；
2. 删除 x 数（若有多个相同的数，应只删除一个，如果没有请忽略该操作）；

3. 查询 x 数的排名（排名定义为比当前数小的数的个数 + 1）；
4. 查询排名为 x 的数；
5. 求 x 的前驱（前驱定义为小于 x，且最大的数，如不存在输出 $-2\,147\,483\,647$）；
6. 求 x 的后继（后继定义为大于 x，且最小的数，如不存在输出 $2\,147\,483\,647$）。

以上操作均基于某一个历史版本，同时生成一个新的版本（操作 3, 4, 5, 6 即保持原版本无变化）。而每个版本的编号则为操作的序号。特别地，最初的版本编号为 0。

就是**普通平衡树**一题的可持久化版，操作和该题类似。

只是使用了可持久化的 merge 和 split 操作。

推荐的练手题

1. 「Luogu P3919」 可持久化数组 （ 模板题 ）[2]
2. 「Codeforces 702F」T-shirt[3]
3. 「Luogu P5055」 可持久化文艺平衡树[4]

参考资料与注释

[1] 洛谷 P3835【模版】可持久化平衡树

[2] 「Luogu P3919」可持久化数组（模板题）

[3] 「Codeforces 702F」T-shirt

[4] 「Luogu P5055」可持久化文艺平衡树

10.19.5 可持久化字典树

引入

可持久化 Trie 的方式和可持久化线段树的方式是相似的，即每次只修改被添加或值被修改的节点，而保留没有被改动的节点，在上一个版本的基础上连边，使最后每个版本的 Trie 树的根遍历所能分离出的 Trie 树都是完整且包含全部信息的。

大部分的可持久化 Trie 题中，Trie 都是以 01-Trie 的形式出现的。

" 例题 最大异或和[1] "

对一个长度为 n 的数组 a 维护以下操作：

1. 在数组的末尾添加一个数 x，数组的长度 n 自增 1。
2. 给出查询区间 $[l,r]$ 和一个值 k，求当 $l \le p \le r$ 时，$k \oplus \bigoplus_{i=p}^{n} a_i$ 的最大值。

过程

这个求的值可能有些麻烦，利用常用的处理连续异或的方法，记 $s_x = \bigoplus_{i=1}^{x} a_i$，则原式等价于 $s_{p-1} \oplus s_n \oplus k$，观察到 $s_n \oplus k$ 在查询的过程中是固定的，题目的查询变化为查询在区间 $[l-1,r-1]$ 中异或定值（$s_n \oplus k$）的最大值。

继续按类似于可持久化线段树的思路，考虑每次的查询都查询整个区间。我们只需把这个区间建一棵 Trie 树，将这个区间中的每个树都加入这棵 Trie 中，查询的时候，尽量往与当前位不相同的地方跳。

查询区间，只需要利用前缀和和差分的思想，用两棵前缀 Trie 树（也就是按顺序添加数的两个历史版本）相减即得到该区间的 Trie 树。再利用动态开点的思想，不添加没有计算过的点，以减少空间占用。

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;
const int maxn = 600010;
int n, q, a[maxn], s[maxn], l, r, x;
char op;

struct Trie {
  int cnt, rt[maxn], ch[maxn * 33][2], val[maxn * 33];

  void insert(int o, int lst, int v) {
    for (int i = 28; i >= 0; i--) {
      val[o] = val[lst] + 1;  // 在原版本的基础上更新
      if ((v & (1 << i)) == 0) {
        if (!ch[o][0]) ch[o][0] = ++cnt;
        ch[o][1] = ch[lst][1];
        o = ch[o][0];
        lst = ch[lst][0];
      } else {
        if (!ch[o][1]) ch[o][1] = ++cnt;
        ch[o][0] = ch[lst][0];
        o = ch[o][1];
        lst = ch[lst][1];
      }
    }
    val[o] = val[lst] + 1;
    // printf("%d\n",o);
  }

  int query(int o1, int o2, int v) {
    int ret = 0;
    for (int i = 28; i >= 0; i--) {
      // printf("%d %d %d\n",o1,o2,val[o1]-val[o2]);
      int t = ((v & (1 << i)) ? 1 : 0);
      if (val[ch[o1][!t]] - val[ch[o2][!t]])
        ret += (1 << i), o1 = ch[o1][!t],
                         o2 = ch[o2][!t];  // 尽量向不同的地方跳
      else
        o1 = ch[o1][t], o2 = ch[o2][t];
    }
    return ret;
  }
} st;

int main() {
  scanf("%d%d", &n, &q);
  for (int i = 1; i <= n; i++) scanf("%d", a + i), s[i] = s[i - 1] ^ a[i];
  for (int i = 1; i <= n; i++)
    st.rt[i] = ++st.cnt, st.insert(st.rt[i], st.rt[i - 1], s[i]);
  while (q--) {
```

```
    scanf(" %c", &op);
    if (op == 'A') {
      n++;
      scanf("%d", a + n);
      s[n] = s[n - 1] ^ a[n];
      st.rt[n] = ++st.cnt;
      st.insert(st.rt[n], st.rt[n - 1], s[n]);
    }
    if (op == 'Q') {
      scanf("%d%d%d", &l, &r, &x);
      l--;
      r--;
      if (l == 0)
        printf("%d\n", max(s[n] ^ x, st.query(st.rt[r], st.rt[0], s[n] ^ x)));
      else
        printf("%d\n", st.query(st.rt[r], st.rt[l - 1], s[n] ^ x));
    }
  }
  return 0;
}
```

参考资料与注释

[1] 最大异或和

10.19.6 可持久化可并堆

可持久化可并堆一般用于求解 k 短路问题。

如果一种可并堆的时间复杂度不是均摊的，那么它在可持久化后单次操作的时间复杂度就保证是 $O(\log n)$ 的，即不会因为特殊数据而使复杂度退化。

可持久化左偏树

在学习本内容前，请先了解 左偏树 的相关内容。

过程

回顾左偏树的合并过程，假设我们要合并分别以 x, y 为根节点的两棵左偏树，且维护的左偏树满足小根堆的性质：

1. 如果 x, y 中有结点为空，返回 $x + y$。
2. 选择 x, y 两结点中权值更小的结点，作为合并后左偏树的根。
3. 递归合并 x 的右子树与 y，将合并后的根节点作为 x 的右儿子。
4. 维护当前合并后左偏树的左偏性质，维护 `dist` 值，返回选择的根节点。

由于每次递归都会使 `dist[x]+dist[y]` 减少一，而 `dist[x]` 是 $O(\log n)$ 的，一次最多只会修改 $O(\log n)$ 个结点，所以这样做的时间复杂度是 $O(\log n)$ 的。

可持久化要求保留历史信息，使得之后能够访问之前的版本。要将左偏树可持久化，就要将其沿途修改的路径复制一遍。

所以可持久化左偏树的合并过程是这样的：

1. 如果 x, y 中有结点为空，返回 $x + y$。
2. 选择 x, y 两结点中权值更小的结点，新建该结点的一个复制 p，作为合并后左偏树的根。

3. 递归合并 p 的右子树与 y，将合并后的根节点作为 p 的右儿子。
4. 维护以 p 为根的左偏树的左偏性质，维护其 `dist` 值，返回 p。

由于左偏树一次最多只会修改并新建 $O(\log n)$ 个结点，设操作次数为 m，则可持久化左偏树的时间复杂度和空间复杂度均为 $O(m \log n)$。

参考实现

```
int merge(int x, int y) {
  if (!x || !y) return x + y;
  if (v[x] > v[y]) swap(x, y);
  int p = ++cnt;
  lc[p] = lc[x];
  v[p] = v[x];
  rc[p] = merge(rc[x], y);
  if (dist[lc[p]] < dist[rc[p]]) swap(lc[p], rc[p]);
  dist[p] = dist[rc[p]] + 1;
  return p;
}
```

10.20 树套树

10.20.1 线段树套线段树

Authors: Chrogeek, HeRaNO, Dev-XYS, Dev-jqe

常见用途

在算法竞赛中，我们有时需要维护多维度信息。在这种时候，我们经常需要树套树来记录信息。

实现原理

我们考虑用树套树如何实现在二维平面上进行单点修改，区域查询。我们考虑外层的线段树，最底层的 1 到 n 个节点的子树，分别代表第 1 到第 n 行的线段树。那么这些底层的节点对应的父节点，就代表其两个子节点的子树所在的一片区域。

性质

空间复杂度

通常情况下，我们不可能对于外层线段树的每一个结点都建立一颗子线段树，空间需求过大。树套树一般采取动态开点的策略。单次修改，我们会涉及到外层线段树的 $\log n$ 个节点，且对于每个节点的子树涉及 $\log n$ 个节点，所以单次修改产生的空间最多为 $\log^2 n$。

时间复杂度

对于询问操作，我们考虑我们在外层线段树上进行 $\log n$ 次操作，每次操作会在一个内层线段树上进行 $\log n$ 次操作，所以时间复杂度为 $\log^2 n$。修改操作，与询问操作复杂度相同，也为 $\log^2 n$。

经典例题

陌上花开[1] 将第一维排序处理，然后用树套树维护第二维和第三维。

示例代码

第二维查询

```
int tree_query(int k, int l, int r, int x) {
  if (k == 0) return 0;
  if (1 <= l && r <= sec[x].y) return vec_query(ou_root[k], 1, p, 1, sec[x].z);
  int mid = l + r >> 1, res = 0;
  if (1 <= mid) res += tree_query(ou_ch[k][0], l, mid, x);
  if (sec[x].y > mid) res += tree_query(ou_ch[k][1], mid + 1, r, x);
  return res;
}
```

第二维修改

```
void tree_insert(int &k, int l, int r, int x) {
  if (k == 0) k = ++ou_tot;
  vec_insert(ou_root[k], 1, p, sec[x].z);
  if (l == r) return;
  int mid = l + r >> 1;
  if (sec[x].y <= mid)
    tree_insert(ou_ch[k][0], l, mid, x);
  else
    tree_insert(ou_ch[k][1], mid + 1, r, x);
}
```

第三维查询

```
int vec_query(int k, int l, int r, int x, int y) {
  if (k == 0) return 0;
  if (x <= l && r <= y) return data[k];
  int mid = l + r >> 1, res = 0;
  if (x <= mid) res += vec_query(ch[k][0], l, mid, x, y);
  if (y > mid) res += vec_query(ch[k][1], mid + 1, r, x, y);
  return res;
}
```

第三维修改

```
void vec_insert(int &k, int l, int r, int loc) {
  if (k == 0) k = ++tot;
  data[k]++;
  if (l == r) return;
  int mid = l + r >> 1;
  if (loc <= mid) vec_insert(ch[k][0], l, mid, loc);
  if (loc > mid) vec_insert(ch[k][1], mid + 1, r, loc);
}
```

相关算法

面对多维度信息的题目时，如果题目没有要求强制在线，我们还可以考虑 **CDQ 分治**，或者**整体二分**等分治算法，来避免使用高级数据结构，减少代码实现难度。

参考资料与注释

[1] 陌上花开

10.20.2 平衡树套线段树

10.20.3 线段树套平衡树

Authors: Dev-jqe, HeRaNO, huaruoji

常见用途

在算法竞赛中，我们有时需要维护多维度信息。在这种时候，我们经常需要树套树来记录信息。当需要维护前驱，后继，第 k 大，某个数的排名，或者插入删除的时候，我们通常需要使用平衡树来满足我们的需求，即线段树套平衡树。

过程

我们以**二逼平衡树**为例，来解释实现原理。

关于树套树的构建，我们对于外层线段树正常建树，对于线段树上的某一个节点，建立一棵平衡树，包含该节点所覆盖的序列。具体操作时我们可以将序列元素一个个插入，每经过一个线段树节点，就将该元素加入到该节点的平衡树中。

操作一，求某区间中某值的排名：我们对于外层线段树正常操作，对于在某区间中的节点的平衡树，我们返回平衡树中比该值小的元素个数，合并区间时，我们将小的元素个数求和即可。最后将返回值 $+1$，即为某值在某区间中的排名。

操作二，求某区间中排名为 k 的值：我们可以采用二分策略。因为一个元素可能存在多个，其排名为一区间，且有些元素原序列不存在。所以我们采取和操作一类似的思路，我们用小于该值的元素个数作为参考进行二分，即可得解。

操作三，将某个数替换为另外一个数：我们只要在所有包含某数的平衡树中删除某数，然后再插入另外一个数即可。外层依旧正常线段树操作。

操作四，求某区间中某值的前驱：我们对于外层线段树正常操作，对于在某区间中的节点的平衡树，我们返回某值在该平衡树中的前驱，线段树的区间结果合并时，我们取最大值即可。

性质

空间复杂度

我们每个元素加入 $O(\log n)$ 个平衡树，所以空间复杂度为 $O((n+q)\log n)$。

时间复杂度

- 对于 1，3，4 操作，我们考虑我们在外层线段树上进行 $O(\log n)$ 次操作，每次操作会在一个内层平衡树树上进行 $O(\log n)$ 次操作，所以时间复杂度为 $O(\log^2 n)$。
- 对于 2 操作，多一个二分过程，为 $O(\log^3 n)$。

经典例题

二逼平衡树[1] 外层线段树，内层平衡树。

实现

平衡树部分代码请参考 Splay 等其他条目。

操作一：

```
int vec_rank(int k, int l, int r, int x, int y, int t) {
  if (x <= l && r <= y) {
    return spy[k].chk_rank(t);
  }
  int mid = l + r >> 1;
  int res = 0;
  if (x <= mid) res += vec_rank(k << 1, l, mid, x, y, t);
  if (y > mid) res += vec_rank(k << 1 | 1, mid + 1, r, x, y, t);
  if (x <= mid && y > mid) res--;
  return res;
}
```

操作二：

```
int el = 0, er = 100000001, emid;
while (el != er) {
  emid = el + er >> 1;
  if (vec_rank(1, 1, n, tl, tr, emid) - 1 < tk)
    el = emid + 1;
  else
    er = emid;
}
printf("%d\n", el - 1);
```

操作三：

```
void vec_chg(int k, int l, int r, int loc, int x) {
  int t = spy[k].find(dat[loc]);
  spy[k].dele(t);
  spy[k].insert(x);
  if (l == r) return;
  int mid = l + r >> 1;
  if (loc <= mid) vec_chg(k << 1, l, mid, loc, x);
  if (loc > mid) vec_chg(k << 1 | 1, mid + 1, r, loc, x);
}
```

操作四：

```
int vec_front(int k, int l, int r, int x, int y, int t) {
  if (x <= l && r <= y) return spy[k].chk_front(t);
  int mid = l + r >> 1;
  int res = 0;
  if (x <= mid) res = max(res, vec_front(k << 1, l, mid, x, y, t));
  if (y > mid) res = max(res, vec_front(k << 1 | 1, mid + 1, r, x, y, t));
  return res;
}
```

相关算法

面对多维度信息的题目时，如果题目没有要求强制在线，我们还可以考虑 CDQ 分治，或者 整体二分 等分治算法，来避免使用高级数据结构，减少代码实现难度。

参考资料与注释

[1] 二逼平衡树

10.20.4 树状数组套权值线段树

Authors: Ir1d, sshwy, Enter-tainer, H-J-Granger, ouuan, GavinZhengOI, hsfzLZH1, xyf007

静态区间 k 小值（POJ 2104 K-th Number）[1] 的问题可以用 权值线段树 在 $O(n\log n)$ 的时间复杂度内解决。

如果区间变成动态的呢？即，如果还要求支持一种操作：单点修改某一位上的值，又该怎么办呢？

" 例题 二逼平衡树 （ 树套树 ）[2]"

" 例题 ZOJ 2112 Dynamic Rankings[3]"

如果用 线段树套平衡树 中所论述的，用线段树套平衡树，即对于线段树的每一个节点，对于其所表示的区间维护一个平衡树，然后用二分来查找 k 小值。由于每次查询操作都要覆盖多个区间，即有多个节点，但是平衡树并不能多个值一起查找，所以时间复杂度是 $O(n\log^3 n)$，并不是最优的。

优化的思路是把二分答案的操作和查询小于一个值的数的数量两种操作结合起来，使用**线段树套动态开点权值线段树**，由于所有线段树的结构是相同的，可以在多棵树上同时进行线段树上二分。

在修改操作进行时，先在线段树上从上往下跳到被修改的点，删除所经过的点所指向的动态开点权值线段树上的原来的值，然后插入新的值，要经过 $O(\log n)$ 个线段树上的节点，在动态开点权值线段树上一次修改操作是 $O(\log n)$ 的，所以修改操作的时间复杂度为 $O(\log^2 n)$。

在查询答案时，先取出该区间覆盖在线段树上的所有点，然后用类似于静态区间 k 小值的方法，将这些点一起向左儿子或向右儿子跳。如果所有这些点左儿子存储的值大于等于 k，则往左跳，否则往右跳。由于最多只能覆盖 $O(\log n)$ 个节点，所以最多一次只有这么多个节点向下跳，时间复杂度为 $O(\log^2 n)$。

由于线段树的常数较大，在实现中往往使用常数更小且更方便处理前缀和的**树状数组**实现。另外空间复杂度是 $O(n\log^2 n)$ 的，使用时**注意空间限制**。

给出一种代码实现：

" 实现"

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <map>
#include <set>
#define LC o << 1
#define RC o << 1 | 1
using namespace std;
const int maxn = 1000010;
int n, m, a[maxn], u[maxn], x[maxn], l[maxn], r[maxn], k[maxn], cur, cur1, cur2,
    q1[maxn], q2[maxn], v[maxn];
char op[maxn];
set<int> ST;
map<int, int> mp;

struct segment_tree  // 封装的动态开点权值线段树
{
  int cur, rt[maxn * 4], sum[maxn * 60], lc[maxn * 60], rc[maxn * 60];

  void build(int& o) { o = ++cur; }

  void print(int o, int l, int r) {
```

```
    if (!o) return;
    if (l == r && sum[o]) printf("%d ", l);
    int mid = (l + r) >> 1;
    print(lc[o], l, mid);
    print(rc[o], mid + 1, r);
  }

  void update(int& o, int l, int r, int x, int v) {
    if (!o) o = ++cur;
    sum[o] += v;
    if (l == r) return;
    int mid = (l + r) >> 1;
    if (x <= mid)
      update(lc[o], l, mid, x, v);
    else
      update(rc[o], mid + 1, r, x, v);
  }
} st;

// 树状数组实现
int lowbit(int o) { return (o & (-o)); }

void upd(int o, int x, int v) {
  for (; o <= n; o += lowbit(o)) st.update(st.rt[o], 1, n, x, v);
}

void gtv(int o, int* A, int& p) {
  p = 0;
  for (; o; o -= lowbit(o)) A[++p] = st.rt[o];
}

int qry(int l, int r, int k) {
  if (l == r) return l;
  int mid = (l + r) >> 1, siz = 0;
  for (int i = 1; i <= cur1; i++) siz += st.sum[st.lc[q1[i]]];
  for (int i = 1; i <= cur2; i++) siz -= st.sum[st.lc[q2[i]]];
  // printf("j %d %d %d %d\n",cur1,cur2,siz,k);
  if (siz >= k) {
    for (int i = 1; i <= cur1; i++) q1[i] = st.lc[q1[i]];
    for (int i = 1; i <= cur2; i++) q2[i] = st.lc[q2[i]];
    return qry(l, mid, k);
  } else {
    for (int i = 1; i <= cur1; i++) q1[i] = st.rc[q1[i]];
    for (int i = 1; i <= cur2; i++) q2[i] = st.rc[q2[i]];
    return qry(mid + 1, r, k - siz);
  }
}

/* 线段树实现
void build(int o,int l,int r)
{
    st.build(st.rt[o]);
    if(l==r)return;
    int mid=(l+r)>>1;
```

```
    build(LC,l,mid);
    build(RC,mid+1,r);
}
void print(int o,int l,int r)
{
    printf("%d %d:",l,r);
    st.print(st.rt[o],1,n);
    printf("\n");
    if(l==r)return;
    int mid=(l+r)>>1;
    print(LC,l,mid);
    print(RC,mid+1,r);
}
void update(int o,int l,int r,int q,int x,int v)
{
    st.update(st.rt[o],1,n,x,v);
    if(l==r)return;
    int mid=(l+r)>>1;
    if(q<=mid)update(LC,l,mid,q,x,v);
    else update(RC,mid+1,r,q,x,v);
}
void getval(int o,int l,int r,int ql,int qr)
{
    if(l>qr||r<ql)return;
    if(ql<=l&&r<=qr){q[++cur]=st.rt[o];return;}
    int mid=(l+r)>>1;
    getval(LC,l,mid,ql,qr);
    getval(RC,mid+1,r,ql,qr);
}
int query(int l,int r,int k)
{
    if(l==r)return l;
    int mid=(l+r)>>1,siz=0;
    for(int i=1;i<=cur;i++)siz+=st.sum[st.lc[q[i]]];
    if(siz>=k)
    {
        for(int i=1;i<=cur;i++)q[i]=st.lc[q[i]];
        return query(l,mid,k);
    }
    else
    {
        for(int i=1;i<=cur;i++)q[i]=st.rc[q[i]];
        return query(mid+1,r,k-siz);
    }
}
*/

int main() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= n; i++) scanf("%d", a + i), ST.insert(a[i]);
  for (int i = 1; i <= m; i++) {
    scanf(" %c", op + i);
    if (op[i] == 'C')
      scanf("%d%d", u + i, x + i), ST.insert(x[i]);
```

```
    else
      scanf("%d%d%d", l + i, r + i, k + i);
  }
  for (set<int>::iterator it = ST.begin(); it != ST.end(); it++)
    mp[*it] = ++cur, v[cur] = *it;
  for (int i = 1; i <= n; i++) a[i] = mp[a[i]];
  for (int i = 1; i <= m; i++)
    if (op[i] == 'C') x[i] = mp[x[i]];
  n += m;
  // build(1,1,n);
  for (int i = 1; i <= n; i++) upd(i, a[i], 1);
  // print(1,1,n);
  for (int i = 1; i <= m; i++) {
    if (op[i] == 'C') {
      upd(u[i], a[u[i]], -1);
      upd(u[i], x[i], 1);
      a[u[i]] = x[i];
    } else {
      gtv(r[i], q1, cur1);
      gtv(l[i] - 1, q2, cur2);
      printf("%d\n", v[qry(1, n, k[i])]);
    }
  }
  return 0;
}
```

参考资料与注释

[1] 静态区间 k 小值（POJ 2104 K-th Number）

[2] 二逼平衡树（树套树）

[3] ZOJ 2112 Dynamic Rankings

10.20.5 分块套树状数组

简介

分块套树状数组在特定条件下可以用来做一些树套树可以做的事情，但是相比起树套树，分块套树状数组代码编写更加简短，更加容易实现。

简单的例子

一个简单的例子就是二维平面中矩阵区域内点数的查询。

" 矩形区域查询"

给出 n 个二维平面中的点 (x_i, y_i)，其中 $1 \le i \le n, 1 \le x_i, y_i \le n, 1 \le n \le 10^5$, 要求实现以下中操作：

1. 给出 a, b, c, d，询问以 (a, b) 为左上角，c, d 为右下角的矩形区域内点的个数。
2. 给出 x, y，将横坐标为 x 的点的纵坐标改为 y。

题目**强制在线**，保证 $x_i \ne x_j (1 \le i, j \le n, i \ne j)$。

对于操作 1，可以通过矩形容斥将其转化为 4 个二维偏序的查询去解决，然后因为强制在线，CDQ 分治之类的离线算法就解决不了，于是想到了树套树，比如树状数组套 Treap。这确实可以解决这个问题，但是代码太长了，也不是特别好实现。

注意到，题目还额外保证了 $x_i \neq x_j (1 \leq i, j \leq n, i \neq j)$，这个时候就可以用分块套树状数组解决。

初始化

首先，一个 x 只对应一个 y，所以可以用一个数组记录这个映射关系，比如令 Y_i 表示横坐标为 i 的点的纵坐标。

然后，以 $\sqrt{n}$ 为块大小对横坐标进行分块。为每个块建一棵权值树状数组。记 T_i 为第 i 个块对应的树状数组，$T_{i,j}$ 表示块 i 里纵坐标在 $(j - lowbit(j), j]$ 内的点的个数。

查询

对于操作 1，将其转化为 4 个二维偏序的查询。现在只需要解决给出 a, b，询问有多少个点满足 $1 \leq x_i \leq a, 1 \leq y_i \leq b$。

现在要查询横坐标的范围为 $[1, a]$。因为查询范围最右边可能有一段不是完整的块，所以暴力扫一遍这个段，看是否满足 $Y_i \leq b$，统计出这个段满足要求的点的个数。

现在就只需要处理完整的块。暴力扫一遍前面的块，查询每个块对应的树状数组中值小于 b 的个数，累加到答案上。

这就完事了？不，注意到处理完整的块的时候，其实相当于查询 T 的前缀和，如果修改时也使用树状数组的技巧处理 T，那么查询时复杂度会更低。

修改

普通的做法就先找到点 x 所在的块，然后一减一加两个权值树状数组单点修改，再将 Y_x 置为 y。

如果用了上面说的优化，那就是对 T 也走一个树状数组修改的流程，每次修改也是一减一加两个权值树状数组单点修改。

对上述步骤进行一定的改变，比如将一减一加改成只减，就是删点；改成只加，就是加点。但是必须要注意一个 x 只能对应一个 y。

空间复杂度

分块分了 $\sqrt{n}$ 个块，每个块一颗线段树 $O(n)$ 的空间，所以空间复杂度为 $O(n\sqrt{n})$。

时间复杂度

查询的话，遍历非完整块的段 $O(\sqrt{n})$。然后，对 T 走树状数组查询，每个经历到的 T_i 也走树状数组查询，这一步是 $O(\log(\sqrt{n}) \log n)$ 的复杂度。所以查询的时间复杂度为 $O(\sqrt{n} + \log(\sqrt{n}) \log n)$。

修改和查询一样，复杂度为 $O(\sqrt{n} + \log(\sqrt{n}) \log n)$。

例题 1

"Intersection of Permutations[1]"

给出两个排列 a 和 b，要求实现以下两种操作：

1. 给出 l_a, r_a, l_b, r_b，要求查询既出现在 $a[l_a ... r_a]$ 又出现在 $b[l_b ... r_b]$ 中的元素的个数。
2. 给出 x, y，$swap(b_x, b_y)$。

序列长度 n 满足 $2 \leq n \leq 2 \cdot 10^5$，操作个数 q 满足 $1 \leq q \leq 2 \cdot 10^5$。

对于每个值 i，记 x_i 是它在排列 b 中的下标，y_i 是它在排列 a 中的下标。这样，操作一就变成了一个矩形区域内点的个数的询问，操作 2 可以看成两个修改操作。而且因为是排列，所以满足一个 x 对应一个 y，所以这题可以用分块套树状数组来写。

" 参考代码（分块套树状数组 - 1s）"

```
#include <bits/stdc++.h>
using namespace std;
const int N = 2e5 + 5;
const int M = sqrt(N) + 5;

int n, m, pa[N], pb[N];

int nn, block_size, block_cnt, block_id[N], L[N], R[N], T[M][N];

void build(int n) {
  nn = n;
  block_size = sqrt(nn);
  block_cnt = nn / block_size;
  for (int i = 1; i <= block_cnt; ++i) {
    L[i] = R[i - 1] + 1;
    R[i] = i * block_size;
  }
  if (R[block_cnt] < nn) {
    ++block_cnt;
    L[block_cnt] = R[block_cnt - 1] + 1;
    R[block_cnt] = nn;
  }
  for (int j = 1; j <= block_cnt; ++j)
    for (int i = L[j]; i <= R[j]; ++i) block_id[i] = j;
}

int lb(int x) { return x & -x; }

void add(int p, int v, int d) {
  for (int i = block_id[p]; i <= block_cnt; i += lb(i))
    for (int j = v; j <= nn; j += lb(j)) T[i][j] += d;
}

int getsum(int p, int v) {
  if (!p) return 0;
  int res = 0;
  int id = block_id[p];
  for (int i = L[id]; i <= p; ++i)
    if (pb[i] <= v) ++res;
  for (int i = id - 1; i; i -= lb(i))
    for (int j = v; j; j -= lb(j)) res += T[i][j];
  return res;
}

void update(int x, int y) {
  add(x, pb[x], -1);
  add(y, pb[y], -1);
  swap(pb[x], pb[y]);
  add(x, pb[x], 1);
  add(y, pb[y], 1);
}
```

```cpp
int query(int la, int ra, int lb, int rb) {
  int res = getsum(rb, ra) - getsum(rb, la - 1) - getsum(lb - 1, ra) +
            getsum(lb - 1, la - 1);
  return res;
}

int main() {
  scanf("%d %d", &n, &m);
  int v;
  for (int i = 1; i <= n; ++i) scanf("%d", &v), pa[v] = i;
  for (int i = 1; i <= n; ++i) scanf("%d", &v), pb[i] = pa[v];

  build(n);
  for (int i = 1; i <= n; ++i) add(i, pb[i], 1);

  int op, la, lb, ra, rb, x, y;
  for (int i = 1; i <= m; ++i) {
    scanf("%d", &op);
    if (op == 1) {
      scanf("%d %d %d %d", &la, &ra, &lb, &rb);
      printf("%d\n", query(la, ra, lb, rb));
    } else if (op == 2) {
      scanf("%d %d", &x, &y);
      update(x, y);
    }
  }
  return 0;
}
```

node " 参考代码（树状数组套 Treap—TLE）"

```cpp
#include <bits/stdc++.h>
using namespace std;
const int N = 2e5 + 5;
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());

int n, m, pa[N], pb[N];

// Treap
struct Treap {
  struct node {
    node *l, *r;
    int sz, rnd, v;

    node(int _v) : l(NULL), r(NULL), sz(1), rnd(rng()), v(_v) {}
  };

  int get_size(node*& p) { return p ? p->sz : 0; }

  void push_up(node*& p) {
    if (!p) return;
    p->sz = get_size(p->l) + get_size(p->r) + 1;
  }
```

```cpp
node* root;

node* merge(node* a, node* b) {
  if (!a) return b;
  if (!b) return a;
  if (a->rnd < b->rnd) {
    a->r = merge(a->r, b);
    push_up(a);
    return a;
  } else {
    b->l = merge(a, b->l);
    push_up(b);
    return b;
  }
}

void split_val(node* p, const int& k, node*& a, node*& b) {
  if (!p)
    a = b = NULL;
  else {
    if (p->v <= k) {
      a = p;
      split_val(p->r, k, a->r, b);
      push_up(a);
    } else {
      b = p;
      split_val(p->l, k, a, b->l);
      push_up(b);
    }
  }
}

void split_size(node* p, int k, node*& a, node*& b) {
  if (!p)
    a = b = NULL;
  else {
    if (get_size(p->l) <= k) {
      a = p;
      split_size(p->r, k - get_size(p->l), a->r, b);
      push_up(a);
    } else {
      b = p;
      split_size(p->l, k, a, b->l);
      push_up(b);
    }
  }
}

void ins(int val) {
  node *a, *b;
  split_val(root, val, a, b);
  a = merge(a, new node(val));
  root = merge(a, b);
}
```

```
  void del(int val) {
    node *a, *b, *c, *d;
    split_val(root, val, a, b);
    split_val(a, val - 1, c, d);
    delete d;
    root = merge(c, b);
  }

  int qry(int val) {
    node *a, *b;
    split_val(root, val, a, b);
    int res = get_size(a);
    root = merge(a, b);
    return res;
  }

  int qry(int l, int r) { return qry(r) - qry(l - 1); }
};

// Fenwick Tree
Treap T[N];

int lb(int x) { return x & -x; }

void ins(int x, int v) {
  for (; x <= n; x += lb(x)) T[x].ins(v);
}

void del(int x, int v) {
  for (; x <= n; x += lb(x)) T[x].del(v);
}

int qry(int x, int mi, int ma) {
  int res = 0;
  for (; x; x -= lb(x)) res += T[x].qry(mi, ma);
  return res;
}

int main() {
  scanf("%d %d", &n, &m);
  int v;
  for (int i = 1; i <= n; ++i) scanf("%d", &v), pa[v] = i;
  for (int i = 1; i <= n; ++i) scanf("%d", &v), pb[i] = pa[v];
  for (int i = 1; i <= n; ++i) ins(i, pb[i]);

  int op, la, lb, ra, rb, x, y;
  for (int i = 1; i <= m; ++i) {
    scanf("%d", &op);
    if (op == 1) {
      scanf("%d %d %d %d", &la, &ra, &lb, &rb);
      printf("%d\n", qry(rb, la, ra) - qry(lb - 1, la, ra));
    } else if (op == 2) {
      scanf("%d %d", &x, &y);
```

```
      del(x, pb[x]);
      del(y, pb[y]);
      swap(pb[x], pb[y]);
      ins(x, pb[x]);
      ins(y, pb[y]);
    }
  }
  return 0;
}
```

例题 2

"Complicated Computations[2]"

给出一个序列 a，将 a 所有连续子序列的 MEX 构成的数组作为 b，问 b 的 MEX。一个序列的 MEX 是序列中最小的没出现过的**正整数**。

序列的长度 n 满足 $1 \le n \le 10^5$。

观察：一个序列的 MEX 为 mex，当且仅当这个序列包含 1 至 $mex-1$，但不包含 mex。

依次判断是否存在 MEX 为 1 至 $n+1$ 的连续子序列。如果没有 MEX 为 i 的连续子序列，那么答案即为 i。如果都存在，那么答案为 $n+2$。

在判断 i 时，将序列视为由零或多个 i 分隔的多个段。如果存在一个段，这个段中包含 1 至 $i-1$，但不包含 i，那么就说明存在值为 i 的连续子序列。

用一个数组 Y_j 记录上一个值为 a_j 的元素的位置，以 j 作为 x，Y_j 作为 y，a_j 作为 z。这样，计算段内是否包含 1 至 $i-1$ 就是一个三维偏序的问题。形式化的说，判断段 $[l,r]$ 的 MEX 值是否为 i，就是看满足 $l \le j \le r, Y_j \le l-1, a_j \le i-1$ 的点的个数是否为 $i-1$。

如果在判断完值为 i 的元素之后再将对应的点插入，这时因为 $[l,r]$ 内只存在 $a_j \le i-1$ 的元素，所以上述三维偏序问题就可以转换为二维偏序的问题。

" 参考代码（分块套树状数组 - 78ms）"

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e5 + 5;
const int M = sqrt(N) + 5;

// 分块
int nn, b[N], block_size, block_cnt, block_id[N], L[N], R[N], T[M][N];

void build(int n) {
  nn = n;
  block_size = sqrt(nn);
  block_cnt = nn / block_size;
  for (int i = 1; i <= block_cnt; ++i) {
    L[i] = R[i - 1] + 1;
    R[i] = i * block_size;
  }
  if (R[block_cnt] < nn) {
    ++block_cnt;
    L[block_cnt] = R[block_cnt - 1] + 1;
    R[block_cnt] = nn;
  }
```

```
  for (int j = 1; j <= block_cnt; ++j)
    for (int i = L[j]; i <= R[j]; ++i) block_id[i] = j;
}

int lb(int x) { return x & -x; }

// d = 1: 加点 (p, v)
// d = -1: 删点 (p, v)
void add(int p, int v, int d) {
  for (int i = block_id[p]; i <= block_cnt; i += lb(i))
    for (int j = v; j <= nn; j += lb(j)) T[i][j] += d;
}

// 询问 [1, r] 内，纵坐标小于等于 val 的点有多少个
int getsum(int p, int v) {
  if (!p) return 0;
  int res = 0;
  int id = block_id[p];
  for (int i = L[id]; i <= p; ++i)
    if (b[i] && b[i] <= v) ++res;
  for (int i = id - 1; i; i -= lb(i))
    for (int j = v; j; j -= lb(j)) res += T[i][j];
  return res;
}

// 询问 [1, r] 内，纵坐标小于等于 val 的点有多少个
int query(int l, int r, int val) {
  if (l > r) return -1;
  int res = getsum(r, val) - getsum(l - 1, val);
  return res;
}

// 加点 (p, v)
void update(int p, int v) {
  b[p] = v;
  add(p, v, 1);
}

int n, a[N];
vector<int> g[N];

int main() {
  scanf("%d", &n);

  // 为了减少讨论，加了哨兵节点
  // 因为树状数组添加的时候，为 0 可能会死循环，所以整体往右偏移一位
  // a_1 和 a_{n+2} 为哨兵节点
  for (int i = 2; i <= n + 1; ++i) scanf("%d", &a[i]);
  for (int i = 2; i <= n + 1; ++i) g[a[i]].push_back(i);

  // 分块
  build(n + 2);

  int ans = n + 2, lst, ok;
```

```
  for (int i = 1; i <= n + 1; ++i) {
    g[i].push_back(n + 2);

    lst = 1;
    ok = 0;
    for (int pos : g[i]) {
      if (query(lst + 1, pos - 1, lst) == i - 1) {
        ok = 1;
        break;
      }
      lst = pos;
    }

    if (!ok) {
      ans = i;
      break;
    }

    lst = 1;
    g[i].pop_back();
    for (int pos : g[i]) {
      update(pos, lst);
      lst = pos;
    }
  }
  printf("%d\n", ans);
  return 0;
}
```

" 参考代码（线段树套 Treap-468ms）"

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e5 + 5;

vector<int> g[N];
int n, a[N];

mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());

struct Treap {
  struct node {
    node *l, *r;
    unsigned rnd;
    int sz, v;

    node(int _v) : l(NULL), r(NULL), rnd(rng()), sz(1), v(_v) {}
  };

  int get_size(node*& p) { return p ? p->sz : 0; }

  void push_up(node*& p) {
    if (!p) return;
```

```
    p->sz = get_size(p->l) + get_size(p->r) + 1;
  }

  node* root;

  node* merge(node* a, node* b) {
    if (!a) return b;
    if (!b) return a;
    if (a->rnd < b->rnd) {
      a->r = merge(a->r, b);
      push_up(a);
      return a;
    } else {
      b->l = merge(a, b->l);
      push_up(b);
      return b;
    }
  }

  void split_val(node* p, const int& k, node*& a, node*& b) {
    if (!p)
      a = b = NULL;
    else {
      if (p->v <= k) {
        a = p;
        split_val(p->r, k, a->r, b);
        push_up(a);
      } else {
        b = p;
        split_val(p->l, k, a, b->l);
        push_up(b);
      }
    }
  }

  void split_size(node* p, int k, node*& a, node*& b) {
    if (!p)
      a = b = NULL;
    else {
      if (get_size(p->l) <= k) {
        a = p;
        split_size(p->r, k - get_size(p->l), a->r, b);
        push_up(a);
      } else {
        b = p;
        split_size(p->l, k, a, b->l);
        push_up(b);
      }
    }
  }

  void insert(int val) {
    node *a, *b;
    split_val(root, val, a, b);
```

```
    a = merge(a, new node(val));
    root = merge(a, b);
  }

  int query(int val) {
    node *a, *b;
    split_val(root, val, a, b);
    int res = get_size(a);
    root = merge(a, b);
    return res;
  }

  int qry(int l, int r) { return query(r) - query(l - 1); }
};

// Segment Tree
Treap T[N << 2];

void insert(int x, int l, int r, int p, int val) {
  T[x].insert(val);
  if (l == r) return;
  int mid = (l + r) >> 1;
  if (p <= mid)
    insert(x << 1, l, mid, p, val);
  else
    insert(x << 1 | 1, mid + 1, r, p, val);
}

int query(int x, int l, int r, int L, int R, int val) {
  if (l == L && r == R) return T[x].query(val);
  int mid = (l + r) >> 1;
  if (R <= mid) return query(x << 1, l, mid, L, R, val);
  if (L > mid) return query(x << 1 | 1, mid + 1, r, L, R, val);
  return query(x << 1, l, mid, L, mid, val) +
         query(x << 1 | 1, mid + 1, r, mid + 1, R, val);
}

int query(int l, int r, int val) {
  if (l > r) return -1;
  return query(1, 1, n, l, r, val);
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i <= n; ++i) scanf("%d", &a[i]);
  for (int i = 1; i <= n; ++i) g[a[i]].push_back(i);

  // a_0 和 a_{n+1} 为哨兵节点
  int ans = n + 2, lst, ok;
  for (int i = 1; i <= n + 1; ++i) {
    g[i].push_back(n + 1);

    lst = 0;
    ok = 0;
```

```
    for (int pos : g[i]) {
      if (query(lst + 1, pos - 1, lst) == i - 1) {
        ok = 1;
        break;
      }
      lst = pos;
    }

    if (!ok) {
      ans = i;
      break;
    }

    lst = 0;
    g[i].pop_back();
    for (int pos : g[i]) {
      insert(1, 1, n, pos, lst);
      lst = pos;
    }
  }
  printf("%d\n", ans);
  return 0;
}
```

参考资料与注释

[1] Intersection of Permutations

[2] Complicated Computations

10.21 K-D Tree

Authors: hsfzLZH1, Ir1d

k-D Tree(KDT , k-Dimension Tree) 是一种可以**高效处理** k **维空间信息**的数据结构。

在结点数 n 远大于 2^k 时，应用 k-D Tree 的时间效率很好。

在算法竞赛的题目中，一般有 $k=2$。在本页面分析时间复杂度时，将认为 k 是常数。

建树

k-D Tree 具有二叉搜索树的形态，二叉搜索树上的每个结点都对应 k 维空间内的一个点。其每个子树中的点都在一个 k 维的超长方体内，这个超长方体内的所有点也都在这个子树中。

假设我们已经知道了 k 维空间内的 n 个不同的点的坐标，要将其构建成一棵 k-D Tree，步骤如下：

1. 若当前超长方体中只有一个点，返回这个点。
2. 选择一个维度，将当前超长方体按照这个维度分成两个超长方体。
3. 选择切割点：在选择的维度上选择一个点，这一维度上的值小于这个点的归入一个超长方体（左子树），其余的归入另一个超长方体（右子树）。
4. 将选择的点作为这棵子树的根节点，递归对分出的两个超长方体构建左右子树，维护子树的信息。

为了方便理解，我们举一个 $k=2$ 时的例子。

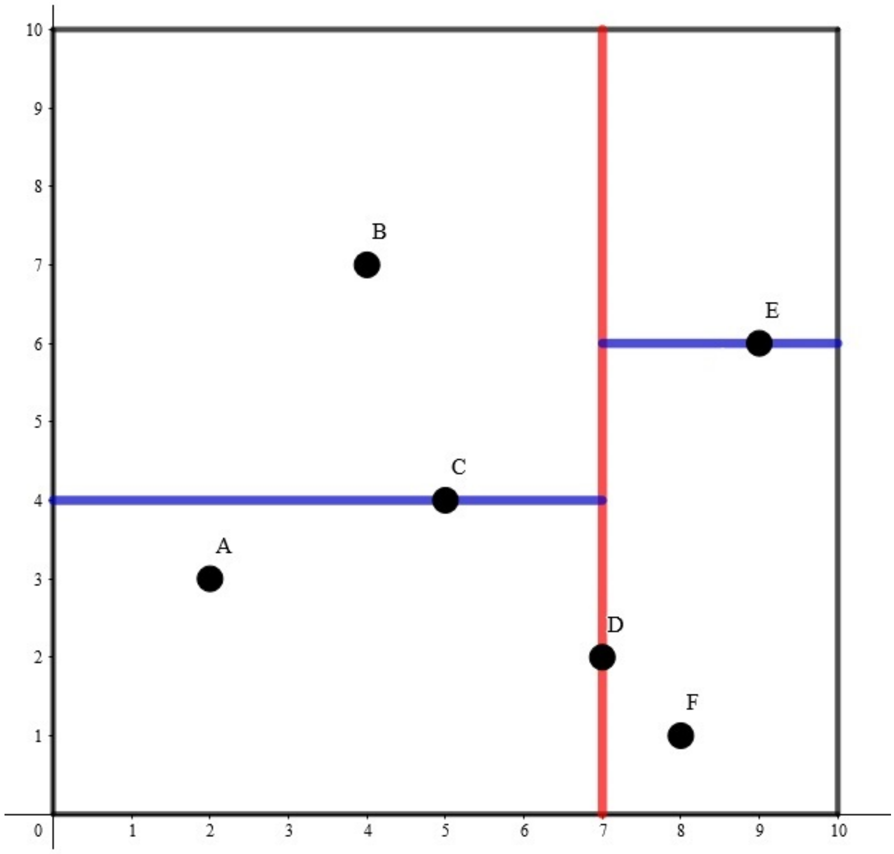


图 10.169

其构建出 k-D Tree 的形态可能是这样的：

其中树上每个结点上的坐标是选择的分割点的坐标，非叶子结点旁的 x 或 y 是选择的切割维度。

这样的复杂度无法保证。对于 2, 3 两步，我们提出两个优化：

1. 轮流选择 k 个维度，以保证在任意连续 k 层里每个维度都被切割到。
2. 每次在维度上选择切割点时选择该维度上的**中位数**，这样可以保证每次分成的左右子树大小尽量相等。

可以发现，使用优化 2 后，构建出的 k-D Tree 的树高最多为 $\log n + O(1)$。

现在，构建 k-D Tree 时间复杂度的瓶颈在于快速选出一个维度上的中位数，并将在该维度上的值小于该中位数的置于中位数的左边，其余置于右边。如果每次都使用 `sort` 函数对该维度进行排序，时间复杂度是 $O(n\log^2 n)$ 的。事实上，单次找出 n 个元素中的中位数并将中位数置于排序后正确的位置的复杂度可以达到 $O(n)$。

我们来回顾一下快速排序的思想。每次我们选出一个数，将小于该数的置于该数的左边，大于该数的置于该数的右边，保证该数在排好序后正确的位置上，然后递归排序左侧和右侧的值。这样的期望复杂度是 $O(n\log n)$ 的。但是由于 k-D Tree 只要求要中位数在排序后正确的位置上，所以我们只需要递归排序包含中位数的**一侧**。可以证明，这样的期望复杂度是 $O(n)$ 的。在 `algorithm` 库中，有一个实现相同功能的函数 `nth_element()`，要找到 `s[l]` 和 `s[r]` 之间的值按照排序规则 `cmp` 排序后在 `s[mid]` 位置上的值，并保证 `s[mid]` 左边的值小于 `s[mid]`，右边的值大于 `s[mid]`，只需写 `nth_element(s+l,s+mid,s+r+1,cmp)`。

借助这种思想，构建 k-D Tree 时间复杂度是 $O(n\log n)$ 的。

高维空间上的操作

在查询高维矩形区域内的所有点的一些信息时，记录每个结点子树内每一维度上的坐标的最大值和最小值。如果当前子树对应的矩形与所求矩形没有交点，则不继续搜索其子树；如果当前子树对应的矩形完全包含在所求矩形内，返回当前子树内所有点的权值和；否则，判断当前点是否在所求矩形内，更新答案并递归在左右子树中查找答案。

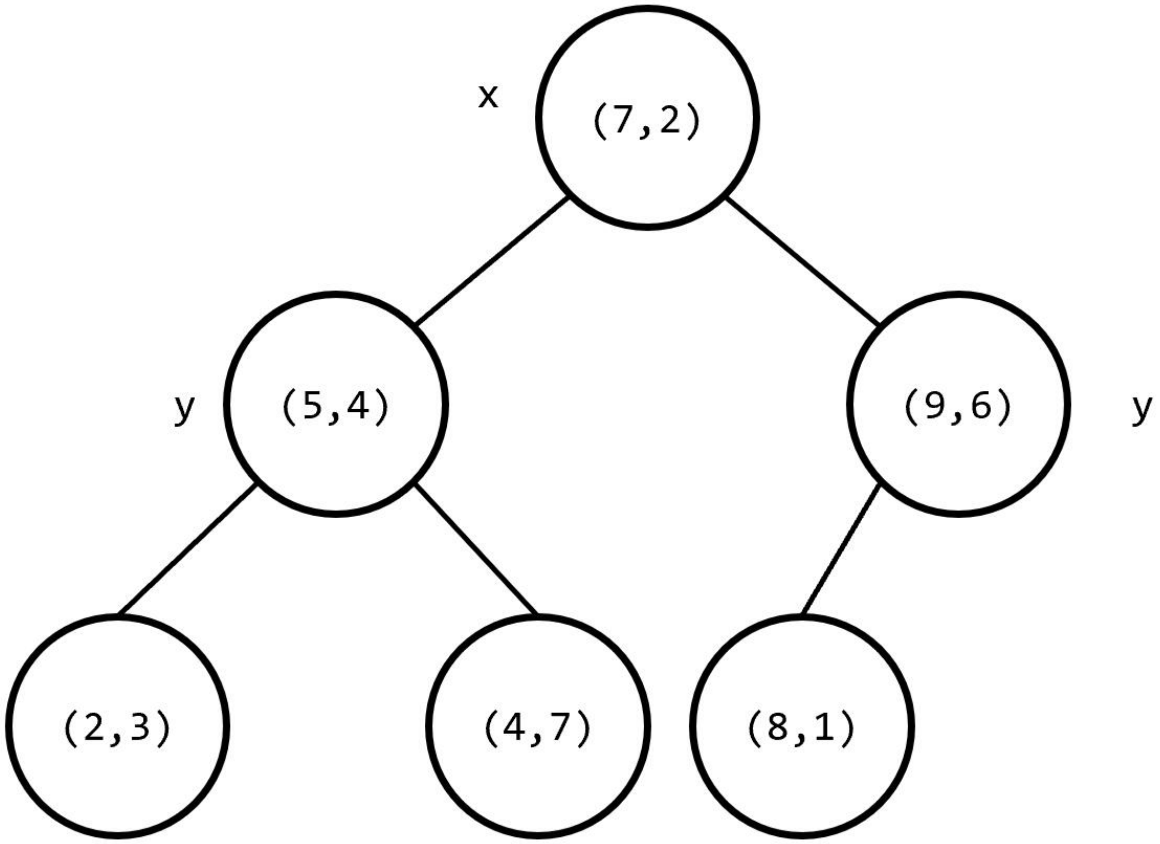


图 10.170

" 实现"

```
int query(int p) {
  if (!p) return 0;
  bool flag{false};
  for (int k : {0, 1}) flag |= (!(l.x[k] <= t[p].L[k] && t[p].R[k] <= h.x[k]));
  if (!flag) return t[p].sum;
  for (int k : {0, 1})
    if (t[p].R[k] < l.x[k] || h.x[k] < t[p].L[k]) return 0;
  int ans{0};
  flag = false;
  for (int k : {0, 1}) flag |= (!(l.x[k] <= t[p].x[k] && t[p].x[k] <= h.x[k]));
  if (!flag) ans = t[p].v;
  return ans += query(t[p].l) + query(t[p].r);
}
```

复杂度分析

先考虑二维的，在查询矩形 R 时，我们将 k-D Tree 上的结点分为三类：

1. 与 R 无交。
2. 完全被 R 包含。
3. 部分被 R 包含。

显然单次查询的复杂度是第 3 类点的个数。注意到第三类点的矩形要么完全包含 R，要么互不包含，而前者显然只有 $O(h)=O(\log n)$ 个，现在我们来分析后者的个数。

首先，我们不妨令矩形的所有边偏移 ϵ，使得查询矩形不穿过已经有的任何点。这样显然是不影响矩形的查询所涵盖的点集的。

注意到互不包含的第 3 类点所对应的矩形，一定有 R 的一条边穿过之。所以我们只需要计算 R 的每条边穿过的矩形个数，即任意一条线段最多经过多少个点对应的矩形。

考虑对于某一个结点 u，它有四个孙子，且它到每一个孙子都在两个维度上各进行了一次划分。经过观察可以发现，按照这种方法将一个矩形划分成四个子矩形，一条与坐标轴平行的线段最多经过两个区域，即从 u 出发的查询，最多向下进入两个孙子仍有第 3 类点（如果线段刚好与分割边界重合则不一定，但是我们偏移查询矩形边界的操作使得这种情况不存在）。

而因为建树的时候，每个点是其整个子树在当前划分维度上的中位数，所以子树大小必定减半。于是，设 u 的子树大小为 n，我们能写出如下递归式：

$$T(n) = 2T(n/4) + O(1)$$

由主定理得 $T(n) = O(\sqrt{n})$。

将递归式推广到 k 维，即 $T(n) = 2^{k-1}T(n/2^k) + O(1)$，于是 $T(n) = O(n^{1-\frac{1}{k}})$（将 k 视为常数）。

插入/删除

如果维护的这个 k 维点集是可变的，即可能会插入或删除一些点，此时 k-D Tree 的平衡性无法保证。由于 k-D Tree 的构造，不能支持旋转，类似与 FHQ Treap 的随机优先级也不能保证其复杂度。对此，有两种比较常见的维护方法。

> note
>
> 很多选手会使用替罪羊树结构来维护。但是注意到在刚才的复杂度分析中，要求儿子的子树大小严格减半，即树高必须为严格的 $\log n + O(1)$，而替罪羊树只满足树高 $O(\log n)$，故查询复杂度无法保证。

根号重构

插入的时候，先存下来要插入的点，每 B 次插入进行一次重构。

删除打个标记即可。如果要求较为严格，可以维护树内有多少个被删除了，达到 B 则重构。

修改复杂度均摊 $O(n\log n/B)$，查询 $O(B+n^{1-\frac{1}{k}})$，若二者数量同阶则 $B = O(\sqrt{n\log n})$ 最优（修改 $O(\sqrt{n\log n})$，查询 $O(\sqrt{n\log n} + n^{1-\frac{1}{k}})$）。

二进制分组

考虑维护若干棵 2 的自然数次幂的 k-D Tree，满足这些树的大小之和为 n。

插入的时候，新增一棵大小为 1 的 k-D Tree，然后不断将相同大小的树合并（直接拍扁重构）。实现的时候可以只重构一次。

容易发现需要合并的树的大小一定从 2^0 开始且指数连续。复杂度类似二进制加法，是均摊 $O(n\log^2 n)$ 的，因为重构本身带 log。

查询的时候，直接分别在每颗树上查询，复杂度为 $O\left(\sum_{i\geq 0}(\frac{n}{2^i})^{1-\frac{1}{k}}\right) = O(n^{1-\frac{1}{k}})$。

例题

> "洛谷 P4148 简单题[1]"
>
> 在一个初始值全为 0 的 $n\times n$ 的二维矩阵上，进行 q 次操作，每次操作为以下两种之一：
>
> 1. `1 x y A`：将坐标 (x,y) 上的数加上 A。
> 2. `2 x1 y1 x2 y2`：输出以 $(x1,y1)$ 为左下角，$(x2,y2)$ 为右上角的矩形内（包括矩形边界）的数字和。
>
> 强制在线。内存限制 `20M`。保证答案及所有过程量在 `int` 范围内。

$1 \leq n \leq 500000, 1 \leq q \leq 200000$

20M 的空间卡掉了所有树套树，强制在线卡掉了 CDQ 分治，只能使用 k-D Tree。

以下是二进制分组的参考代码。

" 参考代码"

```
#include <bits/stdc++.h>
using namespace std;
constexpr int N(2e5), LG{18};

struct pt {
  int x[2];
  int v, sum;
  int l, r;
  int L[2], R[2];
} t[N + 5], l, h;

int rt[LG];
int b[N + 5], cnt;

void upd(int p) {
  t[p].sum = t[t[p].l].sum + t[t[p].r].sum + t[p].v;
  for (int k : {0, 1}) {
    t[p].L[k] = t[p].R[k] = t[p].x[k];
    if (t[p].l) {
      t[p].L[k] = min(t[p].L[k], t[t[p].l].L[k]);
      t[p].R[k] = max(t[p].R[k], t[t[p].l].R[k]);
    }
    if (t[p].r) {
      t[p].L[k] = min(t[p].L[k], t[t[p].r].L[k]);
      t[p].R[k] = max(t[p].R[k], t[t[p].r].R[k]);
    }
  }
}

int build(int l, int r, int dep = 0) {
  int p{l + r >> 1};
  nth_element(b + l, b + p, b + r + 1,
              [dep](int x, int y) { return t[x].x[dep] < t[y].x[dep]; });
  int x{b[p]};
  if (l < p) t[x].l = build(l, p - 1, dep ^ 1);
  if (p < r) t[x].r = build(p + 1, r, dep ^ 1);
  upd(x);
  return x;
}

void append(int &p) {
  if (!p) return;
  b[++cnt] = p;
  append(t[p].l);
  append(t[p].r);
  p = 0;
}
```

```cpp
int query(int p) {
  if (!p) return 0;
  bool flag{false};
  for (int k : {0, 1}) flag |= (!(l.x[k] <= t[p].L[k] && t[p].R[k] <= h.x[k]));
  if (!flag) return t[p].sum;
  for (int k : {0, 1})
    if (t[p].R[k] < l.x[k] || h.x[k] < t[p].L[k]) return 0;
  int ans{0};
  flag = false;
  for (int k : {0, 1}) flag |= (!(l.x[k] <= t[p].x[k] && t[p].x[k] <= h.x[k]));
  if (!flag) ans = t[p].v;
  return ans += query(t[p].l) + query(t[p].r);
}

int main() {
  int n;
  cin >> n;
  int lst{0};
  n = 0;
  while (true) {
    int op;
    cin >> op;
    if (op == 1) {
      int x, y, A;
      cin >> x >> y >> A;
      x ^= lst;
      y ^= lst;
      A ^= lst;
      t[++n] = {{x, y}, A};
      b[cnt = 1] = n;
      for (int sz{0};; ++sz)
        if (!rt[sz]) {
          rt[sz] = build(1, cnt);
          break;
        } else
          append(rt[sz]);
    } else if (op == 2) {
      cin >> l.x[0] >> l.x[1] >> h.x[0] >> h.x[1];
      l.x[0] ^= lst;
      l.x[1] ^= lst;
      h.x[0] ^= lst;
      h.x[1] ^= lst;
      lst = 0;
      for (int i{0}; i < LG; ++i) lst += query(rt[i]);
      cout << lst << "\n";
    } else
      break;
  }
  return 0;
}
```

邻域查询

warning

使用 k-D Tree 单次查询最近点的时间复杂度最坏还是 $O(n)$ 的，但不失为一种优秀的骗分算法，使用时请注意。在这里对邻域查询的讲解仅限于加强对 k-D Tree 结构的认识。

" 例题 luogu P1429 平面最近点对 （ 加强版 ）[2]"

给定平面上的 n 个点 (x_i, y_i)，找出平面上最近两个点对之间的 欧几里得距离。

$2 \le n \le 200000, 0 \le x_i, y_i \le 10^9$

首先建出关于这 n 个点的 2-D Tree。

枚举每个结点，对于每个结点找到不等于该结点且距离最小的点，即可求出答案。每次暴力遍历 2-D Tree 上的每个结点的时间复杂度是 $O(n)$ 的，需要剪枝。我们可以维护一个子树中的所有结点在每一维上的坐标的最小值和最大值。假设当前已经找到的最近点对的距离是 ans，如果查询点到子树内所有点都包含在内的长方形的**最近**距离大于等于 ans，则在这个子树内一定没有答案，搜索时不进入这个子树。

此外，还可以使用一种启发式搜索的方法，即若一个结点的两个子树都有可能包含答案，先在与查询点距离最近的一个子树中搜索答案。可以认为，**查询点到子树对应的长方形的最近距离就是此题的估价函数**。

" 参考代码"

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <cstring>
using namespace std;
const int maxn = 200010;
int n, d[maxn], lc[maxn], rc[maxn];
double ans = 2e18;

struct node {
  double x, y;
} s[maxn];

double L[maxn], R[maxn], D[maxn], U[maxn];

double dist(int a, int b) {
  return (s[a].x - s[b].x) * (s[a].x - s[b].x) +
         (s[a].y - s[b].y) * (s[a].y - s[b].y);
}

bool cmp1(node a, node b) { return a.x < b.x; }

bool cmp2(node a, node b) { return a.y < b.y; }

void maintain(int x) {
  L[x] = R[x] = s[x].x;
  D[x] = U[x] = s[x].y;
  if (lc[x])
    L[x] = min(L[x], L[lc[x]]), R[x] = max(R[x], R[lc[x]]),
    D[x] = min(D[x], D[lc[x]]), U[x] = max(U[x], U[lc[x]]);
  if (rc[x])
    L[x] = min(L[x], L[rc[x]]), R[x] = max(R[x], R[rc[x]]),
```

```
    D[x] = min(D[x], D[rc[x]]), U[x] = max(U[x], U[rc[x]]);
}

int build(int l, int r) {
  if (l > r) return 0;
  if (l == r) {
    maintain(l);
    return l;
  }
  int mid = (l + r) >> 1;
  double avx = 0, avy = 0, vax = 0, vay = 0;  // average variance
  for (int i = l; i <= r; i++) avx += s[i].x, avy += s[i].y;
  avx /= (double)(r - l + 1);
  avy /= (double)(r - l + 1);
  for (int i = l; i <= r; i++)
    vax += (s[i].x - avx) * (s[i].x - avx),
        vay += (s[i].y - avy) * (s[i].y - avy);
  if (vax >= vay)
    d[mid] = 1, nth_element(s + l, s + mid, s + r + 1, cmp1);
  else
    d[mid] = 2, nth_element(s + l, s + mid, s + r + 1, cmp2);
  lc[mid] = build(l, mid - 1), rc[mid] = build(mid + 1, r);
  maintain(mid);
  return mid;
}

double f(int a, int b) {
  double ret = 0;
  if (L[b] > s[a].x) ret += (L[b] - s[a].x) * (L[b] - s[a].x);
  if (R[b] < s[a].x) ret += (s[a].x - R[b]) * (s[a].x - R[b]);
  if (D[b] > s[a].y) ret += (D[b] - s[a].y) * (D[b] - s[a].y);
  if (U[b] < s[a].y) ret += (s[a].y - U[b]) * (s[a].y - U[b]);
  return ret;
}

void query(int l, int r, int x) {
  if (l > r) return;
  int mid = (l + r) >> 1;
  if (mid != x) ans = min(ans, dist(x, mid));
  if (l == r) return;
  double distl = f(x, lc[mid]), distr = f(x, rc[mid]);
  if (distl < ans && distr < ans) {
    if (distl < distr) {
      query(l, mid - 1, x);
      if (distr < ans) query(mid + 1, r, x);
    } else {
      query(mid + 1, r, x);
      if (distl < ans) query(l, mid - 1, x);
    }
  } else {
    if (distl < ans) query(l, mid - 1, x);
    if (distr < ans) query(mid + 1, r, x);
  }
}
```

```
int main() {
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) scanf("%lf%lf", &s[i].x, &s[i].y);
  build(1, n);
  for (int i = 1; i <= n; i++) query(1, n, i);
  printf("%.4lf\n", sqrt(ans));
  return 0;
}
```

" 例题 「CQOI2016」K 远点对[3]"

给定平面上的 n 个点 (x_i, y_i)，求欧几里得距离下的第 k 远无序点对之间的距离。

$n \leq 100000, 1 \leq k \leq 100, 0 \leq x_i, y_i < 2^{31}$

和上一道例题类似，从最近点对变成了 k 远点对，估价函数改成了查询点到子树对应的长方形区域的最远距离。用一个小根堆来维护当前找到的前 k 远点对之间的距离，如果当前找到的点对距离大于堆顶，则弹出堆顶并插入这个距离，同样的，使用堆顶的距离来剪枝。

由于题目中强调的是无序点对，即交换前后两点的顺序后仍是相同的点对，则每个有序点对会被计算两次，那么读入的 k 要乘以 2。

" 参考代码"

```
#include <algorithm>
#include <cstring>
#include <iostream>
#include <queue>
using namespace std;
const int maxn = 100010;
long long n, k;
priority_queue<long long, vector<long long>, greater<long long> > q;

struct node {
  long long x, y;
} s[maxn];

bool cmp1(node a, node b) { return a.x < b.x; }

bool cmp2(node a, node b) { return a.y < b.y; }

long long lc[maxn], rc[maxn], L[maxn], R[maxn], D[maxn], U[maxn];

void maintain(int x) {
  L[x] = R[x] = s[x].x;
  D[x] = U[x] = s[x].y;
  if (lc[x])
    L[x] = min(L[x], L[lc[x]]), R[x] = max(R[x], R[lc[x]]),
    D[x] = min(D[x], D[lc[x]]), U[x] = max(U[x], U[lc[x]]);
  if (rc[x])
    L[x] = min(L[x], L[rc[x]]), R[x] = max(R[x], R[rc[x]]),
    D[x] = min(D[x], D[rc[x]]), U[x] = max(U[x], U[rc[x]]);
}
```

```
int build(int l, int r) {
  if (l > r) return 0;
  int mid = (l + r) >> 1;
  double av1 = 0, av2 = 0, va1 = 0, va2 = 0;  // average variance
  for (int i = l; i <= r; i++) av1 += s[i].x, av2 += s[i].y;
  av1 /= (r - l + 1);
  av2 /= (r - l + 1);
  for (int i = l; i <= r; i++)
    va1 += (av1 - s[i].x) * (av1 - s[i].x),
        va2 += (av2 - s[i].y) * (av2 - s[i].y);
  if (va1 > va2)
    nth_element(s + l, s + mid, s + r + 1, cmp1);
  else
    nth_element(s + l, s + mid, s + r + 1, cmp2);
  lc[mid] = build(l, mid - 1);
  rc[mid] = build(mid + 1, r);
  maintain(mid);
  return mid;
}

long long sq(long long x) { return x * x; }

long long dist(int a, int b) {
  return max(sq(s[a].x - L[b]), sq(s[a].x - R[b])) +
         max(sq(s[a].y - D[b]), sq(s[a].y - U[b]));
}

void query(int l, int r, int x) {
  if (l > r) return;
  int mid = (l + r) >> 1;
  long long t = sq(s[mid].x - s[x].x) + sq(s[mid].y - s[x].y);
  if (t > q.top()) q.pop(), q.push(t);
  long long distl = dist(x, lc[mid]), distr = dist(x, rc[mid]);
  if (distl > q.top() && distr > q.top()) {
    if (distl > distr) {
      query(l, mid - 1, x);
      if (distr > q.top()) query(mid + 1, r, x);
    } else {
      query(mid + 1, r, x);
      if (distl > q.top()) query(l, mid - 1, x);
    }
  } else {
    if (distl > q.top()) query(l, mid - 1, x);
    if (distr > q.top()) query(mid + 1, r, x);
  }
}

int main() {
  cin >> n >> k;
  k *= 2;
  for (int i = 1; i <= k; i++) q.push(0);
  for (int i = 1; i <= n; i++) cin >> s[i].x >> s[i].y;
  build(1, n);
```

```
    for (int i = 1; i <= n; i++) query(1, n, i);
    cout << q.top() << endl;
    return 0;
}
```

习题

「SDOI2010」 捉迷藏[4]

「Violet」 天使玩偶 /SJY 摆棋子[5]

「 国家集训队 」JZPFAR[6]

「BOI2007」Mokia 摩基亚[7]

luogu P4475 巧克力王国[8]

「CH 弱省胡策 R2」TATT[9]

参考资料与注释

[1] 洛谷 P4148 简单题

[2] luogu P1429 平面最近点对（加强版）

[3] 「CQOI2016」K 远点对

[4] 「SDOI2010」捉迷藏

[5] 「Violet」天使玩偶/SJY 摆棋子

[6] 「国家集训队」JZPFAR

[7] 「BOI2007」Mokia 摩基亚

[8] luogu P4475 巧克力王国

[9] 「CH 弱省胡策 R2」TATT

10.22 动态树

10.22.1 Link Cut Tree

简介

Link/Cut Tree 是一种数据结构，我们用它来解决**动态树问题**。

Link/Cut Tree 又称 Link-Cut Tree，简称 LCT，但它不叫动态树，动态树是指一类问题。

Splay Tree 是 LCT 的基础，但是 LCT 用的 Splay Tree 和普通的 Splay 在细节处不太一样（进行了一些扩展）。

问题引入

维护一棵树，支持如下操作：

- 修改两点间路径权值。
- 查询两点间路径权值和。
- 修改某点子树权值。
- 查询某点子树权值和。

这是一道树剖模版题。

但是再加一个操作：

- 断开并连接一些边，保证仍是一棵树。

要求在线求出上面的答案。

这就成了动态树问题，可以使用 LCT 求解。

动态树问题

维护一个**森林**，支持删除某条边，加入某条边，并保证加边，删边之后仍是森林。我们要维护这个森林的一些信息。

一般的操作有两点连通性，两点路径权值和，连接两点和切断某条边、修改信息等。

从 LCT 的角度回顾一下树链剖分

- 对整棵树按子树大小进行剖分，并重新标号。
- 我们发现重新标号之后，在树上形成了一些以链为单位的连续区间，并且可以用线段树进行区间操作。

转向动态树问题

我们发现我们刚刚讲的树剖是以子树大小作为划分条件。那我们能不能重定义一种剖分，使它更适应我们的动态树问题呢?

考虑动态树问题需要什么链。

由于动态维护一个森林，显然我们希望这个链是我们指定的链，以便利用来求解。

实链剖分

对于一个点连向它所有儿子的边，我们自己选择一条边进行剖分，我们称被选择的边为实边，其他边则为虚边。对于实边，我们称它所连接的儿子为实儿子。对于一条由实边组成的链，我们同样称之为实链。请记住我们选择实链剖分的最重要的原因：它是我们选择的，灵活且可变。正是它的这种灵活可变性，我们采用 Splay Tree 来维护这些实链。

LCT

我们可以简单的把 LCT 理解成用一些 Splay 来维护动态的树链剖分，以期实现动态树上的区间操作。对于每条实链，我们建一个 Splay 来维护整个链区间的信息。

辅助树

我们先来看一看辅助树的一些性质，再通过一张图实际了解一下辅助树的具体结构。

在本文里，你可以认为一些 Splay 构成了一个辅助树，每棵辅助树维护的是一棵树，一些辅助树构成了 LCT，其维护的是整个森林。

1. 辅助树由多棵 Splay 组成，每棵 Splay 维护原树中的一条路径，且中序遍历这棵 Splay 得到的点序列，从前到后对应原树「从上到下」的一条路径。
2. 原树每个节点与辅助树的 Splay 节点一一对应。
3. 辅助树的各棵 Splay 之间并不是独立的。每棵 Splay 的根节点的父亲节点本应是空，但在 LCT 中每棵 Splay 的根节点的父亲节点指向原树中**这条链**的父亲节点（即链最顶端的点的父亲节点）。这类父亲链接与通常 Splay 的父亲链接区别在于儿子认父亲，而父亲不认儿子，对应原树的一条**虚边**。因此，每个连通块恰好有一个点的父亲节点为空。
4. 由于辅助树的以上性质，我们维护任何操作都不需要维护原树，辅助树可以在任何情况下拿出一个唯一的原树，我们只需要维护辅助树即可。

现在我们有一棵原树，如图所示。（加粗边是实边，虚线边是虚边。）

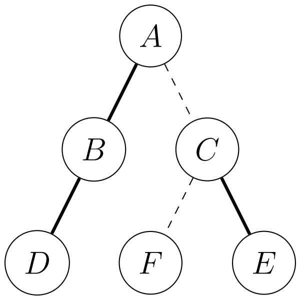


图 10.171 tree

由刚刚的定义，辅助树的结构如图所示。

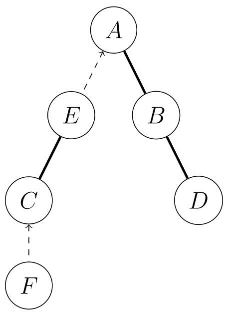


图 10.172 auxtree

考虑原树和辅助树的结构关系

- 原树中的实链: 在辅助树中节点都在一棵 Splay 中。
- 原树中的虚链: 在辅助树中，子节点所在 Splay 的 Father 指向父节点，但是父节点的两个儿子都不指向子节点。
- 注意：原树的根不等于辅助树的根。
- 原树的 Father 指向不等于辅助树的 Father 指向。
- 辅助树是可以在满足辅助树、Splay 的性质下任意换根的。
- 虚实链变换可以轻松在辅助树上完成，这也就是实现了动态维护树链剖分。

接下来要用到的变量声明

- `ch[N][2]` 左右儿子
- `f[N]` 父亲指向
- `sum[N]` 路径权值和
- `val[N]` 点权

- tag[N] 翻转标记
- laz[N] 权值标记
- siz[N] 辅助树上子树大小
- Other_Vars

函数声明

一般数据结构函数（字面意思）

1. PushUp(x)
2. PushDown(x)

Splay 树的函数

下面是 Splay 树中用到的函数，具体可以查阅 Splay 树。

1. Get(x) 获取 x 是父亲的哪个儿子。
2. Splay(x) 通过和 Rotate 操作联动实现把 x 旋转到**当前 Splay 的根**。
3. Rotate(x) 将 x 向上旋转一层的操作。

新操作

1. Access(x) 把从根到 x 的所有点放在一条实链里，使根到 x 成为一条实路径，并且在同一棵 Splay 里。**只有此操作是必须实现的，其他操作视题目而实现**。
2. IsRoot(x) 判断 x 是否是所在树的根。
3. Update(x) 在 Access 操作之后，递归地从上到下 PushDown 更新信息。
4. MakeRoot(x) 使 x 点成为其所在树的根。
5. Link(x, y) 在 x, y 两点间连一条边。
6. Cut(x, y) 把 x, y 两点间边删掉。
7. Find(x) 找到 x 所在树的根节点编号。
8. Fix(x, v) 修改 x 的点权为 v。
9. Split(x, y) 提取出 x, y 间的路径，方便做区间操作。

宏定义

- #define ls ch[p][0]
- #define rs ch[p][1]

函数讲解

PushUp()

```cpp
void PushUp(int p) {
  // maintain other variables
  siz[p] = siz[ls] + siz[rs] + 1;
}
```

PushDown()

```cpp
void PushDown(int p) {
  if (tag[p] != std_tag) {
    // pushdown the tag
```

```
    tag[p] = std_tag;
  }
}
```

Splay() && Rotate()

这里 `Splay()` 和 `Rotate()` 与 Splay 树的实现有些区别。

```
#define Get(x) (ch[f[x]][1] == x)

void Rotate(int x) {
  int y = f[x], z = f[y], k = Get(x);
  if (!isRoot(y)) ch[z][ch[z][1] == y] = x;
  // 上面这句一定要写在前面，普通的 Splay 是不用的，因为 isRoot  (后面会讲)
  ch[y][k] = ch[x][!k], f[ch[x][!k]] = y;
  ch[x][!k] = y, f[y] = x, f[x] = z;
  PushUp(y), PushUp(x);
}

void Splay(int x) {
  Update(
      x);  // 马上就能看到啦。在 Splay 之前要把旋转会经过的路径上的点都 PushDown
  for (int fa; fa = f[x], !isRoot(x); Rotate(x)) {
    if (!isRoot(fa)) Rotate(Get(fa) == Get(x) ? fa : x);
  }
}
```

以上函数可以查阅 Splay 树。

下面是 LCT 独有的函数。

isRoot()

```
// 在前面我们已经说过，LCT 具有如果一个儿子不是实儿子，他的父亲找不到它的性质
// 所以当一个点既不是它父亲的左儿子，又不是它父亲的右儿子，它就是当前 Splay 的根
#define isRoot(x) (ch[f[x]][0] != x && ch[f[x]][1] != x)
```

Access()

```
// Access 是 LCT
// 的核心操作，试想我们像求解一条路径，而这条路径恰好就是我们当前的一棵 Splay，
// 直接调用其信息即可。先来看一下代码，再结合图来看看过程
int Access(int x) {
  int p;
  for (p = 0; x; p = x, x = f[x]) {
    Splay(x), ch[x][1] = p, PushUp(x);
  }
  return p;
}
```

- 我们有这样一棵树，实线为实边，虚线为虚边。
- 它的辅助树可能长成这样（构图方式不同可能 LCT 的结构也不同）。
- 现在我们要 `Access(N)`，把 A 到 N 路径上的边都变为实边，拉成一棵 Splay。
- 实现的方法是从下到上逐步更新 Splay。

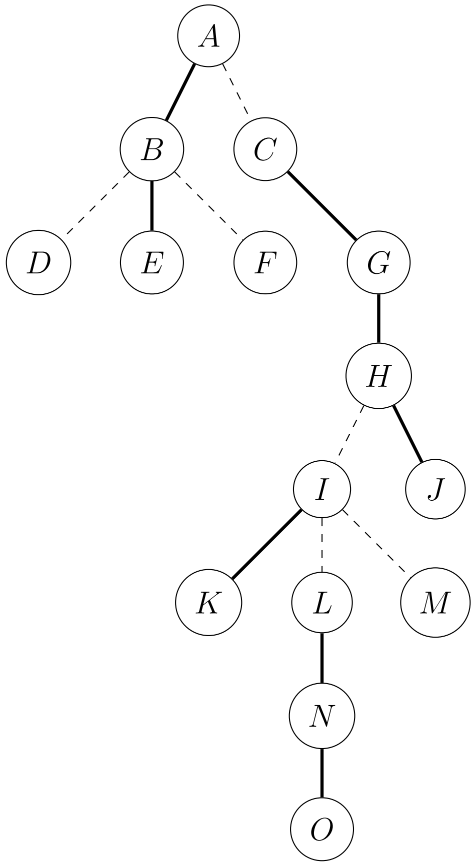


图 10.173 initial tree

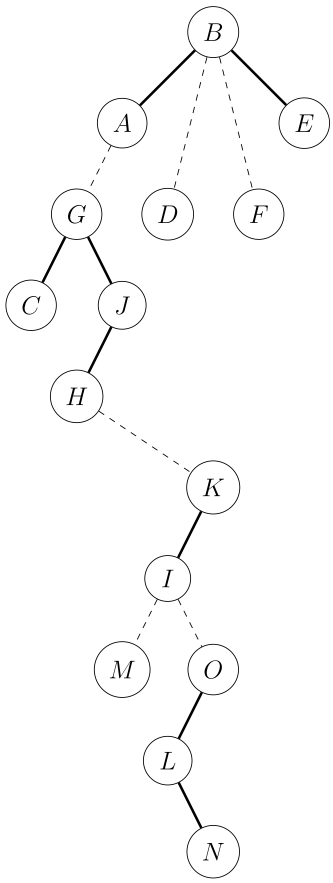


图 10.174 initial auxtree

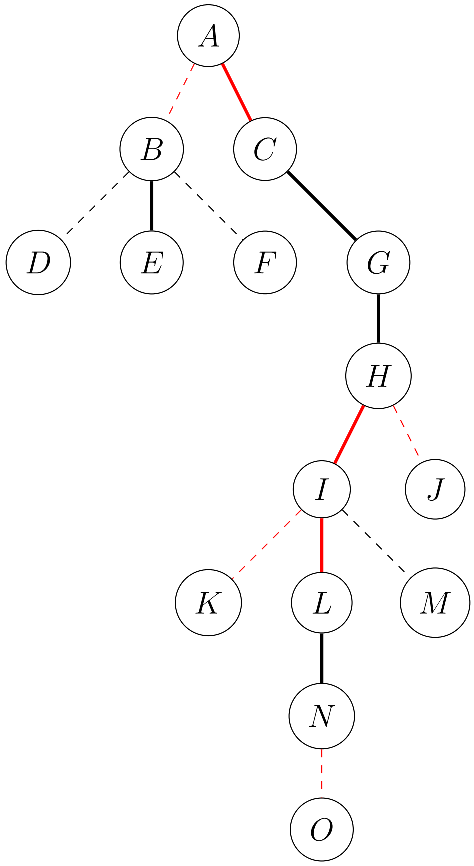


图 10.175 access tree

- 首先我们要把 N 旋至当前 Splay 的根。
- 为了保证 AuxTree（辅助树）的性质，原来 N 到 O 的实边要更改为虚边。
- 由于认父不认子的性质，我们可以单方面的把 N 的儿子改为 `NULL`。
- 于是原来的 AuxTree 就从下图变成了下下图。
- 下一步，我们把 N 指向的 Father I 也旋转到 I 的 Splay 树根。
- 原来的实边 I—K 要去掉，这时候我们把 I 的右儿子指向 N，就得到了 I—L 这样一棵 Splay。
- 接下来，按照刚刚的操作步骤，由于 I 的 Father 指向 H，我们把 H 旋转到他所在 Splay Tree 的根，然后把 H 的 rs 设为 I。
- 之后的树是这样的。
- 同理我们 `Splay(A)`，并把 A 的右儿子指向 H。
- 于是我们得到了这样一棵 AuxTree。并且发现 A—N 的整个路径已经在同一棵 Splay 中了。

```
// 回顾一下代码
int Access(int x) {
  int p;
  for (p = 0; x; p = x, x = f[x]) {
    Splay(x), ch[x][1] = p, PushUp(x);
  }
  return p;
}
```

我们发现 `Access()` 其实很容易，只有如下四步操作：

1. 把当前节点转到根。
2. 把儿子换成之前的节点。

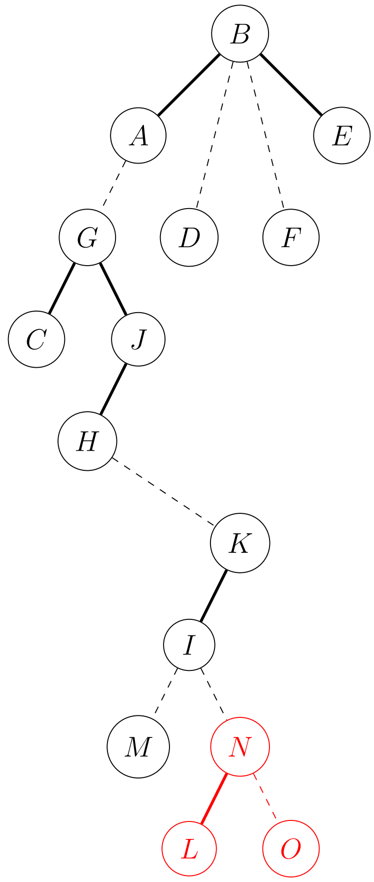


图 10.176 step 1 auxtree

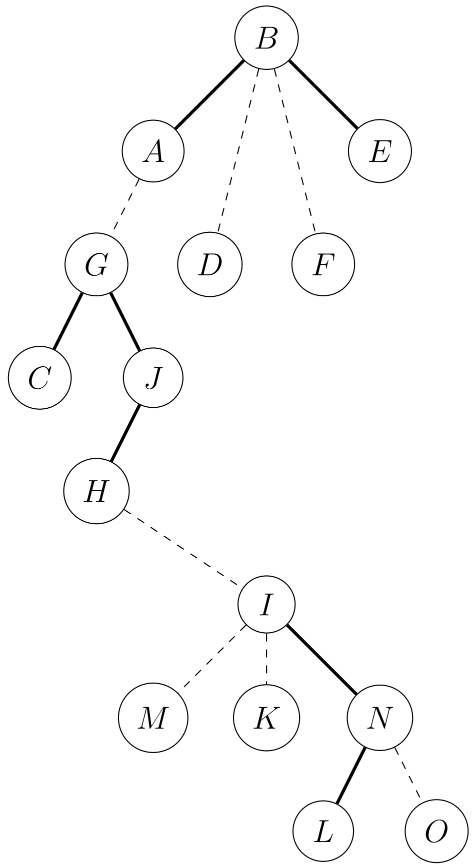


图 10.177 step 2 auxtree

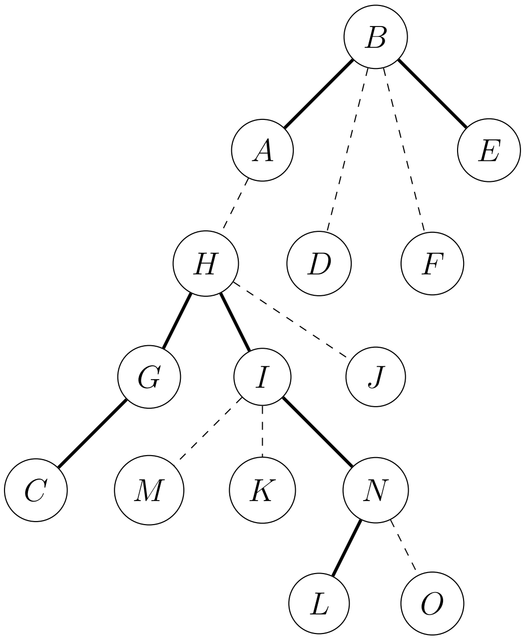


图 10.178 step 3 auxtree

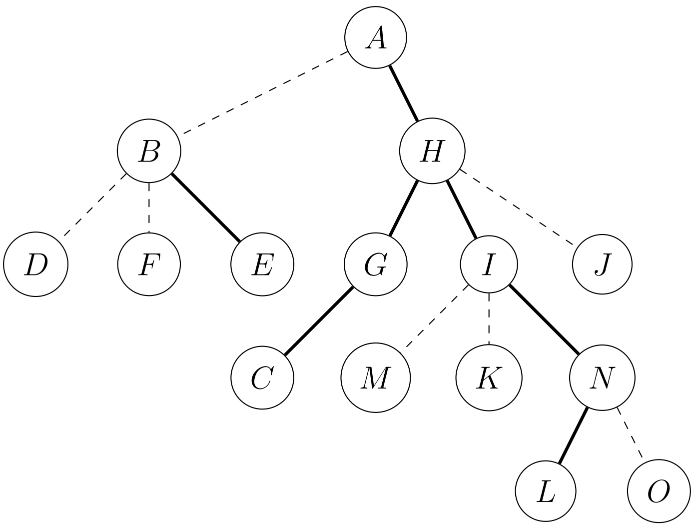


图 10.179 step final auxtree

3. 更新当前点的信息。
4. 把当前点换成当前点的父亲，继续操作。

这里提供的 Access 还有一个返回值。这个返回值相当于最后一次虚实链变换时虚边父亲节点的编号。该值有两个含义：

- 连续两次 Access 操作时，第二次 Access 操作的返回值等于这两个节点的 LCA.
- 表示 x 到根的链所在的 Splay 树的根。这个节点一定已经被旋转到了根节点，且父亲一定为空。

`Update()`

```cpp
// 从上到下一层一层 pushDown 即可
void Update(int p) {
  if (!isRoot(p)) Update(f[p]);
  pushDown(p);
}
```

`makeRoot()`

- `Make_Root()` 的重要性丝毫不亚于 `Access()`。我们在需要维护路径信息的时候，一定会出现路径深度无法严格递增的情况，根据 AuxTree 的性质，这种路径是不能出现在一棵 Splay 中的。
- 这时候我们需要用到 `Make_Root()`。
- `Make_Root()` 的作用是使指定的点成为原树的根，考虑如何实现这种操作。
- 设 `Access(x)` 的返回值为 y，则此时 x 到当前根的路径恰好构成一个 Splay，且该 Splay 的根为 y.
- 考虑将树用有向图表示出来，给每条边定一个方向，表示从儿子到父亲的方向。容易发现换根相当于将 x 到根的路径的所有边反向（请仔细思考）。
- 因此将 x 到当前根的路径翻转即可。
- 由于 y 是 x 到当前根的路径所代表的 Splay 的根，因此将以 y 为根的 Splay 树进行区间翻转即可。

```cpp
void makeRoot(int p) {
  p = Access(p);
  swap(ch[p][0], ch[p][1]);
  tag[p] ^= 1;
}
```

`Link()`

- Link 两个点其实很简单，先 `Make_Root(x)`，然后把 x 的父亲指向 y 即可。显然，这个操作肯定不能发生在同一棵树内，所以记得先判一下。

```cpp
void Link(int x, int p) {
  makeRoot(x);
  splay(x);
  f[x] = p;
}
```

`Split()`

- `Split` 操作意义很简单，就是拿出一棵 Splay，维护的是 x 到 y 的路径。
- 先 `MakeRoot(x)`，然后 `Access(y)`。如果要 y 做根，再 `Splay(y)`。
- 另外 Split 这三个操作可以直接把需要的路径拿出到 y 的子树上，可以进行其他操作。

Cut()

- Cut 有两种情况，保证合法和不一定保证合法。
- 如果保证合法，直接 Split(x, y)，这时候 y 是根，x 一定是它的儿子，双向断开即可。就像这样：

```
void Cut(int x, int p) { makeRoot(x), Access(p), Splay(p), ls = f[x] = 0; }
```

如果是不保证合法，我们需要判断一下是否有，这里选择使用 map 存一下，但是这里有一个利用性质的方法：

想要删边，必须要满足如下三个条件：

1. x, y 连通。
2. x, y 的路径上没有其他的链。
3. x 没有右儿子。

总结一下，上面三句话的意思就一个：x, y 之间有边。

具体实现就留作一个思考题给大家。判断连通需要用到后面的 Find，其他两点稍作思考分析一下结构就知道该怎么判断了。

Find()

- Find() 查找的是 x 所在的**原树**的根，请不要把原树根和辅助树根弄混。在 Access(p) 后，再 Splay(p)。这样根就是树里深度最小的那个，一直往左儿子走，沿途 PushDown 即可。
- 一直走到没有 ls，非常简单。
- 注意，每次查询之后需要把查询到的答案对应的结点 Splay 上去以保证复杂度。

```
int Find(int p) {
  Access(p);
  Splay(p);
  pushDown(p);
  while (ls) p = ls, pushDown(p);
  Splay(p);
  return p;
}
```

注意事项

- 操作前一定要想一想需不需要 PushUp 或者 PushDown，LCT 由于特别灵活的原因，少 Pushdown 或者 Pushup 一次就可能把修改改到不该改的点上！
- LCT 的 Rotate 和 Splay 的不太一样，if (z) 一定要放在前面。
- LCT 的 Splay 操作就是旋转到根，没有旋转到谁儿子的操作，因为不需要。

习题

- 「BZOJ 3282」Tree[1]
- 「HNOI2010」 弹飞绵羊[2]

维护树链信息

LCT 通过 Split(x,y) 操作，可以将树上从点 x 到点 y 的路径提取到以 y 为根的 Splay 内，树链信息的修改和统计转化为平衡树上的操作，这使得 LCT 在维护树链信息上具有优势。此外，借助 LCT 实现的在树链上二分比树链剖分少一个 $O(\log n)$ 的复杂度。

" 例题 「国家集训队」Tree II[3]"

给出一棵有 n 个结点的树，每个点的初始权值为 1。q 次操作，每次操作均为以下四种之一：

1. `- u1 v1 u2 v2`：将树上 u_1, v_1 两点之间的边删除，连接 u_2, v_2 两点，保证操作合法且连边后仍是一棵树。
2. `+ u v c`：将树上 u, v 两点之间的路径上的点权都增加 c。
3. `* u v c`：将树上 u, v 两点之间的路径上的点权都乘以 c。
4. `/ u v`：输出树上 u, v 两点之间的路径上的点权之和对 51061 取模后的值。

$1 \le n, q \le 10^5, 0 \le c \le 10^4$

`-` 操作可以直接 `Cut(u1,v1),Link(u2,v2)`。

对树上 u, v 两点之间的路径进行修改时，先 `Split(u,v)`。

此题要求进行在辅助树上的子树加，子树乘，子树求和操作，所以我们除了一般 LCT 需要维护的子树翻转标记，还要维护子树加法标记和子树乘法标记。处理标记的方法和在 Splay 上是一样的。

在打上和下传加法标记时，子树权值和的变化量和子树中的结点数有关，所以我们还要维护子树的大小 `siz`。

在下传标记时，需要注意顺序，先下传乘法标记再下传加法标记。子树翻转和子树加乘两种标记没有冲突。

" 参考代码"

```
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;
#define int long long
const int maxn = 100010;
const int mod = 51061;
int n, q, u, v, c;
char op;

struct Splay {
  int ch[maxn][2], fa[maxn], siz[maxn], val[maxn], sum[maxn], rev[maxn],
      add[maxn], mul[maxn];

  void clear(int x) {
    ch[x][0] = ch[x][1] = fa[x] = siz[x] = val[x] = sum[x] = rev[x] = add[x] =
        0;
    mul[x] = 1;
  }

  int getch(int x) { return (ch[fa[x]][1] == x); }

  int isroot(int x) {
    clear(0);
    return ch[fa[x]][0] != x && ch[fa[x]][1] != x;
  }

  void maintain(int x) {
    clear(0);
    siz[x] = (siz[ch[x][0]] + 1 + siz[ch[x][1]]) % mod;
    sum[x] = (sum[ch[x][0]] + val[x] + sum[ch[x][1]]) % mod;
  }
```

```cpp
void pushdown(int x) {
  clear(0);
  if (mul[x] != 1) {
    if (ch[x][0])
      mul[ch[x][0]] = (mul[x] * mul[ch[x][0]]) % mod,
      val[ch[x][0]] = (val[ch[x][0]] * mul[x]) % mod,
      sum[ch[x][0]] = (sum[ch[x][0]] * mul[x]) % mod,
      add[ch[x][0]] = (add[ch[x][0]] * mul[x]) % mod;
    if (ch[x][1])
      mul[ch[x][1]] = (mul[x] * mul[ch[x][1]]) % mod,
      val[ch[x][1]] = (val[ch[x][1]] * mul[x]) % mod,
      sum[ch[x][1]] = (sum[ch[x][1]] * mul[x]) % mod,
      add[ch[x][1]] = (add[ch[x][1]] * mul[x]) % mod;
    mul[x] = 1;
  }
  if (add[x]) {
    if (ch[x][0])
      add[ch[x][0]] = (add[ch[x][0]] + add[x]) % mod,
      val[ch[x][0]] = (val[ch[x][0]] + add[x]) % mod,
      sum[ch[x][0]] = (sum[ch[x][0]] + add[x] * siz[ch[x][0]] % mod) % mod;
    if (ch[x][1])
      add[ch[x][1]] = (add[ch[x][1]] + add[x]) % mod,
      val[ch[x][1]] = (val[ch[x][1]] + add[x]) % mod,
      sum[ch[x][1]] = (sum[ch[x][1]] + add[x] * siz[ch[x][1]] % mod) % mod;
    add[x] = 0;
  }
  if (rev[x]) {
    if (ch[x][0]) rev[ch[x][0]] ^= 1, swap(ch[ch[x][0]][0], ch[ch[x][0]][1]);
    if (ch[x][1]) rev[ch[x][1]] ^= 1, swap(ch[ch[x][1]][0], ch[ch[x][1]][1]);
    rev[x] = 0;
  }
}

void update(int x) {
  if (!isroot(x)) update(fa[x]);
  pushdown(x);
}

void print(int x) {
  if (!x) return;
  pushdown(x);
  print(ch[x][0]);
  printf("%lld ", x);
  print(ch[x][1]);
}

void rotate(int x) {
  int y = fa[x], z = fa[y], chx = getch(x), chy = getch(y);
  fa[x] = z;
  if (!isroot(y)) ch[z][chy] = x;
  ch[y][chx] = ch[x][chx ^ 1];
  fa[ch[x][chx ^ 1]] = y;
  ch[x][chx ^ 1] = y;
  fa[y] = x;
```

```
    maintain(y);
    maintain(x);
    maintain(z);
  }

  void splay(int x) {
    update(x);
    for (int f = fa[x]; f = fa[x], !isroot(x); rotate(x))
      if (!isroot(f)) rotate(getch(x) == getch(f) ? f : x);
  }

  void access(int x) {
    for (int f = 0; x; f = x, x = fa[x]) splay(x), ch[x][1] = f, maintain(x);
  }

  void makeroot(int x) {
    access(x);
    splay(x);
    swap(ch[x][0], ch[x][1]);
    rev[x] ^= 1;
  }

  int find(int x) {
    access(x);
    splay(x);
    while (ch[x][0]) x = ch[x][0];
    splay(x);
    return x;
  }
} st;

main() {
  scanf("%lld%lld", &n, &q);
  for (int i = 1; i <= n; i++) st.val[i] = 1, st.maintain(i);
  for (int i = 1; i < n; i++) {
    scanf("%lld%lld", &u, &v);
    if (st.find(u) != st.find(v)) st.makeroot(u), st.fa[u] = v;
  }
  while (q--) {
    scanf(" %c%lld%lld", &op, &u, &v);
    if (op == '+') {
      scanf("%lld", &c);
      st.makeroot(u), st.access(v), st.splay(v);
      st.val[v] = (st.val[v] + c) % mod;
      st.sum[v] = (st.sum[v] + st.siz[v] * c % mod) % mod;
      st.add[v] = (st.add[v] + c) % mod;
    }
    if (op == '-') {
      st.makeroot(u);
      st.access(v);
      st.splay(v);
      if (st.ch[v][0] == u && !st.ch[u][1]) st.ch[v][0] = st.fa[u] = 0;
      scanf("%lld%lld", &u, &v);
      if (st.find(u) != st.find(v)) st.makeroot(u), st.fa[u] = v;
```

```cpp
    }
    if (op == '*') {
      scanf("%lld", &c);
      st.makeroot(u), st.access(v), st.splay(v);
      st.val[v] = st.val[v] * c % mod;
      st.sum[v] = st.sum[v] * c % mod;
      st.mul[v] = st.mul[v] * c % mod;
    }
    if (op == '/')
      st.makeroot(u), st.access(v), st.splay(v), printf("%lld\n", st.sum[v]);
  }
  return 0;
}
```

习题

- luogu P3690【模板】Link Cut Tree（动态树）[4]
- 「SDOI2011」染色[5]
- 「SHOI2014」三叉神经树[6]

维护连通性质

判断是否连通

借助 LCT 的 `Find()` 函数，可以判断动态森林上的两点是否连通。如果有 `Find(x)==Find(y)`，则说明 x, y 两点在一棵树上，相互连通。

" 例题 「SDOI2008」洞穴勘测[7]"

一开始有 n 个独立的点，m 次操作。每次操作为以下之一：

1. `Connect u v`：在 u, v 两点之间连接一条边。
2. `Destroy u v`：删除在 u, v 两点之间的边，保证之前存在这样的一条边。
3. `Query u v`：询问 u, v 两点是否连通。

保证在任何时刻图的形态都是一个森林。

$n \le 10^4, m \le 2 \times 10^5$

" 参考代码"

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;
const int maxn = 10010;

struct Splay {
  int ch[maxn][2], fa[maxn], tag[maxn];

  void clear(int x) { ch[x][0] = ch[x][1] = fa[x] = tag[x] = 0; }

  int getch(int x) { return ch[fa[x]][1] == x; }

  int isroot(int x) { return ch[fa[x]][0] != x && ch[fa[x]][1] != x; }
```

```cpp
  void pushdown(int x) {
    if (tag[x]) {
      if (ch[x][0]) swap(ch[ch[x][0]][0], ch[ch[x][0]][1]), tag[ch[x][0]] ^= 1;
      if (ch[x][1]) swap(ch[ch[x][1]][0], ch[ch[x][1]][1]), tag[ch[x][1]] ^= 1;
      tag[x] = 0;
    }
  }

  void update(int x) {
    if (!isroot(x)) update(fa[x]);
    pushdown(x);
  }

  void rotate(int x) {
    int y = fa[x], z = fa[y], chx = getch(x), chy = getch(y);
    fa[x] = z;
    if (!isroot(y)) ch[z][chy] = x;
    ch[y][chx] = ch[x][chx ^ 1];
    fa[ch[x][chx ^ 1]] = y;
    ch[x][chx ^ 1] = y;
    fa[y] = x;
  }

  void splay(int x) {
    update(x);
    for (int f = fa[x]; f = fa[x], !isroot(x); rotate(x))
      if (!isroot(f)) rotate(getch(x) == getch(f) ? f : x);
  }

  void access(int x) {
    for (int f = 0; x; f = x, x = fa[x]) splay(x), ch[x][1] = f;
  }

  void makeroot(int x) {
    access(x);
    splay(x);
    swap(ch[x][0], ch[x][1]);
    tag[x] ^= 1;
  }

  int find(int x) {
    access(x);
    splay(x);
    while (ch[x][0]) x = ch[x][0];
    splay(x);
    return x;
  }
} st;

int n, q, x, y;
char op[maxn];

int main() {
```

```
  scanf("%d%d", &n, &q);
  while (q--) {
    scanf("%s%d%d", op, &x, &y);
    if (op[0] == 'Q') {
      if (st.find(x) == st.find(y))
        printf("Yes\n");
      else
        printf("No\n");
    }
    if (op[0] == 'C')
      if (st.find(x) != st.find(y)) st.makeroot(x), st.fa[x] = y;
    if (op[0] == 'D') {
      st.makeroot(x);
      st.access(y);
      st.splay(y);
      if (st.ch[y][0] == x && !st.ch[x][1]) st.ch[y][0] = st.fa[x] = 0;
    }
  }
  return 0;
}
```

维护边双连通分量

如果要求将边双连通分量缩成点，每次添加一条边，所连接的树上的两点如果相互连通，那么这条路径上的所有点都会被缩成一个点。

" 例题 「AHOI2005」 航线规划[8]"

给出 n 个点，初始时有 m 条无向边，q 次操作，每次操作为以下之一：

1. `0 u v`：删除 u, v 之间的连边，保证此时存在这样的一条边。
2. `1 u v`：查询此时 u, v 两点之间可能的所有路径必须经过的边的数量。

保证图在任意时刻都连通。

$1 < n < 3 \times 10^4, 1 < m < 10^5, 0 \leq q \leq 4 \times 10^4$

可以发现，u, v 两点之间的所有可能路径必须经过的边的数量为将所有边双连通分量缩成点之后 u 所在点和 v 所在点之间的路径上的结点数 -1。

由于题目中的删边操作不好进行，我们考虑离线逆向进行操作，改删边为加边。

加入一条边时，如果两点原来不连通，则在 LCT 上连接两点；否则提取出加这条边之前 LCT 上这两点之间的路径，遍历辅助树上的这个子树，相当于遍历了这条路径，将这些点合并，利用并查集维护合并的信息。

用合并后并查集的代表元素代替原来树上的路径。注意之后的每次操作都要找到操作点在并查集上的代表元素进行操作。

" 参考代码"

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <map>
using namespace std;
const int maxn = 200010;
int f[maxn];
```

```
int findp(int x) { return f[x] ? f[x] = findp(f[x]) : x; }

void merge(int x, int y) {
  x = findp(x);
  y = findp(y);
  if (x != y) f[x] = y;
}

struct Splay {
  int ch[maxn][2], fa[maxn], tag[maxn], siz[maxn];

  void clear(int x) { ch[x][0] = ch[x][1] = fa[x] = tag[x] = siz[x] = 0; }

  int getch(int x) { return ch[findp(fa[x])][1] == x; }

  int isroot(int x) {
    return ch[findp(fa[x])][0] != x && ch[findp(fa[x])][1] != x;
  }

  void maintain(int x) {
    clear(0);
    if (x) siz[x] = siz[ch[x][0]] + 1 + siz[ch[x][1]];
  }

  void pushdown(int x) {
    if (tag[x]) {
      if (ch[x][0]) tag[ch[x][0]] ^= 1, swap(ch[ch[x][0]][0], ch[ch[x][0]][1]);
      if (ch[x][1]) tag[ch[x][1]] ^= 1, swap(ch[ch[x][1]][0], ch[ch[x][1]][1]);
      tag[x] = 0;
    }
  }

  void print(int x) {
    if (!x) return;
    pushdown(x);
    print(ch[x][0]);
    printf("%d ", x);
    print(ch[x][1]);
  }

  void update(int x) {
    if (!isroot(x)) update(findp(fa[x]));
    pushdown(x);
  }

  void rotate(int x) {
    x = findp(x);
    int y = findp(fa[x]), z = findp(fa[y]), chx = getch(x), chy = getch(y);
    fa[x] = z;
    if (!isroot(y)) ch[z][chy] = x;
    ch[y][chx] = ch[x][chx ^ 1];
    fa[ch[x][chx ^ 1]] = y;
    ch[x][chx ^ 1] = y;
    fa[y] = x;
```

```
    maintain(y);
    maintain(x);
    if (z) maintain(z);
  }

  void splay(int x) {
    x = findp(x);
    update(x);
    for (int f = findp(fa[x]); f = findp(fa[x]), !isroot(x); rotate(x))
      if (!isroot(f)) rotate(getch(x) == getch(f) ? f : x);
  }

  void access(int x) {
    for (int f = 0; x; f = x, x = findp(fa[x]))
      splay(x), ch[x][1] = f, maintain(x);
  }

  void makeroot(int x) {
    x = findp(x);
    access(x);
    splay(x);
    tag[x] ^= 1;
    swap(ch[x][0], ch[x][1]);
  }

  int find(int x) {
    x = findp(x);
    access(x);
    splay(x);
    while (ch[x][0]) x = ch[x][0];
    splay(x);
    return x;
  }

  void dfs(int x) {
    pushdown(x);
    if (ch[x][0]) dfs(ch[x][0]), merge(ch[x][0], x);
    if (ch[x][1]) dfs(ch[x][1]), merge(ch[x][1], x);
  }
} st;

int n, m, q, x, y, cur, ans[maxn];

struct oper {
  int op, a, b;
} s[maxn];

map<pair<int, int>, int> mp;

int main() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= n; i++) st.maintain(i);
  for (int i = 1; i <= m; i++)
    scanf("%d%d", &x, &y), mp[{x, y}] = mp[{y, x}] = 1;
```

```
  while (scanf("%d", &s[++q].op)) {
    if (s[q].op == -1) {
      q--;
      break;
    }
    scanf("%d%d", &s[q].a, &s[q].b);
    if (!s[q].op) mp[{s[q].a, s[q].b}] = mp[{s[q].b, s[q].a}] = 0;
  }
  reverse(s + 1, s + q + 1);
  for (map<pair<int, int>, int>::iterator it = mp.begin(); it != mp.end(); it++)
    if (it->second) {
      mp[{it->first.second, it->first.first}] = 0;
      x = findp(it->first.first);
      y = findp(it->first.second);
      if (st.find(x) != st.find(y))
        st.makeroot(x), st.fa[x] = y;
      else {
        if (x == y) continue;
        st.makeroot(x);
        st.access(y);
        st.splay(y);
        st.dfs(y);
        int t = findp(y);
        st.fa[t] = findp(st.fa[y]);
        st.ch[t][0] = st.ch[t][1] = 0;
        st.maintain(t);
      }
    }
  for (int i = 1; i <= q; i++) {
    if (s[i].op == 0) {
      x = findp(s[i].a);
      y = findp(s[i].b);
      st.makeroot(x);
      st.access(y);
      st.splay(y);
      st.dfs(y);
      int t = findp(y);
      st.fa[t] = st.fa[y];
      st.ch[t][0] = st.ch[t][1] = 0;
      st.maintain(t);
    }
    if (s[i].op == 1) {
      x = findp(s[i].a);
      y = findp(s[i].b);
      st.makeroot(x);
      st.access(y);
      st.splay(y);
      ans[++cur] = st.siz[y] - 1;
    }
  }
  for (int i = cur; i >= 1; i--) printf("%d\n", ans[i]);
  return 0;
}
```

习题

- luogu P3950 部落冲突[9]
- bzoj 4998 星球联盟[10]
- bzoj 2959 长跑[11]

维护边权

LCT 并不能直接处理边权，此时需要对每条边建立一个对应点，方便查询链上的边信息。利用这一技巧可以动态维护生成树。

" 例题 luogu P4234 最小差值生成树[12]"

给定一个 n 个点，m 条边的带权无向图，求其边权最大值和边权最小值的差值最小的生成树，输出这个差值。

数据保证至少存在一棵生成树。

$1 \le n \le 5 \times 10^4, 1 \le m \le 2 \times 10^5, 1 \le w_i \le 10^4$

将边按照边权从小到大排序，枚举选择的最右边的一条边，要得到最优解，需要使边权最小边的边权最大。

每次按照顺序添加边，如果将要连接的这两个点已经连通，则删除这两点之间边权最小的一条边。如果整个图已经连通成了一棵树，则用当前边权减去最小边权更新答案。最小边权可用双指针法更新。

LCT 上没有固定的父子关系，所以不能将边权记录在点权中。

记录树链上的边的信息，可以使用**拆边**。对每条边建立一个对应的点，从这条边向其两个端点连接一条边，原先的连边与删边操作都变成两次操作。

" 参考代码"

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <set>
using namespace std;
const int maxn = 5000010;

struct Splay {
  int ch[maxn][2], fa[maxn], tag[maxn], val[maxn], minn[maxn];

  void clear(int x) {
    ch[x][0] = ch[x][1] = fa[x] = tag[x] = val[x] = minn[x] = 0;
  }

  int getch(int x) { return ch[fa[x]][1] == x; }

  int isroot(int x) { return ch[fa[x]][0] != x && ch[fa[x]][1] != x; }

  void maintain(int x) {
    if (!x) return;
    minn[x] = x;
    if (ch[x][0]) {
      if (val[minn[ch[x][0]]] < val[minn[x]]) minn[x] = minn[ch[x][0]];
    }
    if (ch[x][1]) {
      if (val[minn[ch[x][1]]] < val[minn[x]]) minn[x] = minn[ch[x][1]];
```

```
    }
  }

  void pushdown(int x) {
    if (tag[x]) {
      if (ch[x][0]) tag[ch[x][0]] ^= 1, swap(ch[ch[x][0]][0], ch[ch[x][0]][1]);
      if (ch[x][1]) tag[ch[x][1]] ^= 1, swap(ch[ch[x][1]][0], ch[ch[x][1]][1]);
      tag[x] = 0;
    }
  }

  void update(int x) {
    if (!isroot(x)) update(fa[x]);
    pushdown(x);
  }

  void print(int x) {
    if (!x) return;
    pushdown(x);
    print(ch[x][0]);
    printf("%d ", x);
    print(ch[x][1]);
  }

  void rotate(int x) {
    int y = fa[x], z = fa[y], chx = getch(x), chy = getch(y);
    fa[x] = z;
    if (!isroot(y)) ch[z][chy] = x;
    ch[y][chx] = ch[x][chx ^ 1];
    fa[ch[x][chx ^ 1]] = y;
    ch[x][chx ^ 1] = y;
    fa[y] = x;
    maintain(y);
    maintain(x);
    if (z) maintain(z);
  }

  void splay(int x) {
    update(x);
    for (int f = fa[x]; f = fa[x], !isroot(x); rotate(x))
      if (!isroot(f)) rotate(getch(x) == getch(f) ? f : x);
  }

  void access(int x) {
    for (int f = 0; x; f = x, x = fa[x]) splay(x), ch[x][1] = f, maintain(x);
  }

  void makeroot(int x) {
    access(x);
    splay(x);
    tag[x] ^= 1;
    swap(ch[x][0], ch[x][1]);
  }
```

```
  int find(int x) {
    access(x);
    splay(x);
    while (ch[x][0]) x = ch[x][0];
    splay(x);
    return x;
  }

  void link(int x, int y) {
    makeroot(x);
    fa[x] = y;
  }

  void cut(int x, int y) {
    makeroot(x);
    access(y);
    splay(y);
    ch[y][0] = fa[x] = 0;
    maintain(y);
  }
} st;

const int inf = 2e9 + 1;
int n, m, ans, nww, x, y;

struct Edge {
  int u, v, w;

  bool operator<(Edge x) const { return w < x.w; };
} s[maxn];

multiset<int> mp;

int main() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= n; i++) st.val[i] = inf, st.maintain(i);
  for (int i = 1; i <= m; i++) scanf("%d%d%d", &s[i].u, &s[i].v, &s[i].w);
  sort(s + 1, s + m + 1);
  for (int i = 1; i <= m; i++) st.val[n + i] = s[i].w, st.maintain(n + i);
  for (int i = 1; i <= m; i++) {
    x = s[i].u;
    y = s[i].v;
    if (x == y) continue;
    if (st.find(x) != st.find(y)) {
      nww++;
      st.link(x, n + i);
      st.link(n + i, y);
      mp.insert(s[i].w);
      if (nww == n - 1) ans = s[i].w - (*(mp.begin()++));
    } else {
      st.makeroot(x);
      st.access(y);
      st.splay(y);
      int t = st.minn[y] - n;
```

```
      st.cut(s[t].u, t + n);
      st.cut(t + n, s[t].v);
      mp.erase(mp.find(s[t].w));
      st.link(x, n + i);
      st.link(n + i, y);
      mp.insert(s[i].w);
      if (nww == n - 1) ans = min(ans, s[i].w - (*(mp.begin()++)));
    }
  }
  printf("%d\n", ans);
  return 0;
}
```

习题

- 「WC2006」 水管局长[13]
- 「BJWC2010」 严格次小生成树[14]
- 「NOI2014」 魔法森林[15]

维护子树信息

LCT 不擅长维护子树信息。统计一个结点所有虚子树的信息，就可以求得整棵树的信息。

" 例题 「BJOI2014」 大融合[16]"

给定 n 个结点和 q 次操作，每个操作为如下形式：

1. `A x y` 在结点 x 和 y 之间连接一条边。
2. `Q x y` 给定一条已经存在的边 (x,y)，求有多少条简单路径，其中包含边 (x,y)。

保证在任意时刻，图的形态都是一棵森林。

$1 \le n,q,x,y \le 10^5$

为询问 `Q` 考虑另一种表述，我们发现答案等于边 (x,y) 在 x 侧的结点数与 y 侧的结点数的乘积，即将边 (x,y) 断开后分别包含 x 和 y 的树的结点数。为了消除断边的影响，在询问后我们再次连接边 (x,y)。

题目中的操作既有连边，又有删边，还保证在任意时刻都是一棵森林，我们不由得想到用 LCT 来维护。但是这题中 LCT 维护的是子树的大小，不像我们印象中的维护一条链的信息，而 LCT 的构造**认父不认子**，不方便我们直接进行子树的统计。怎么办呢？

方法是统计一个结点 x 所有虚儿子（即父亲为 x，但 x 在 Splay 中的左右儿子并不包含它）所代表的子树的贡献。

定义 $siz2[x]$ 为结点 x 的所有虚儿子代表的子树的结点数，$siz[x]$ 为结点 x 子树中的结点数。

不同于以往我们维护 Splay 中子树结点个数的方法，我们在计算结点 x 子树中的结点数时，还要加上 $siz2[x]$，即

```
void maintain(int x) {
  clear(0);
  if (x) siz[x] = siz[ch[x][0]] + 1 + siz[ch[x][1]] + siz2[x];
}
```

而且在我们**改变 Splay 的形态**（即改变一个结点在 Splay 上的左右儿子指向时），需要及时修改 $siz2[x]$ 的值。

在 `Rotate(),Splay()` 操作中，我们都只是改变了 Splay 中结点的相对位置，没有改变任意一条边的虚实情况，所以不对 $siz2[x]$ 进行任何修改。

在 `access` 操作中，在每次 splay 完后，都会改变刚刚 splay 完的结点的右儿子，即该结点与其原右儿子的连边和该节点和新右儿子的连边的虚实情况发生了变化，我们需要加上新变成虚边所连的子树的贡献，减去刚刚变成实边所连的子树的贡献。代码如下：

```
void access(int x) {
  for (int f = 0; x; f = x, x = fa[x])
    splay(x), siz2[x] += siz[ch[x][1]] - siz[f], ch[x][1] = f, maintain(x);
}
```

在 `MakeRoot()`,`Find()` 操作中，我们都只是调用了之前的函数或者在 Splay 上条边，并不用做任何修改。

在连接两点时，我们修改了一个结点的父亲。我们需要在父亲结点的 $siz2$ 值中加上新子结点的子树大小贡献。

```
st.makeroot(x);
st.makeroot(y);
st.fa[x] = y;
st.siz2[y] += st.siz[x];
```

在断开一条边时，我们只是删除了 Splay 上的一条实边，`Maintain` 操作会维护这些信息，不需要做任何修改。

以上是代码修改的细节，最后总结一下 LCT 维护子树信息的要求与方法：

1. 维护的信息要有**可减性**，如子树结点数，子树权值和，但不能直接维护子树最大最小值，因为在将一条虚边变成实边时要排除原先虚边的贡献。
2. 新建一个附加值存储虚子树的贡献，在统计时将其加入本结点答案，在改变边的虚实时及时维护。
3. 其余部分同普通 LCT，在统计子树信息时一定将其作为根节点。
4. 如果维护的信息没有可减性，如维护区间最值，可以对每个结点开一个平衡树维护结点的虚子树中的最值。

" 参考代码"

```
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;
const int maxn = 100010;
typedef long long ll;

struct Splay {
  int ch[maxn][2], fa[maxn], siz[maxn], siz2[maxn], tag[maxn];

  void clear(int x) {
    ch[x][0] = ch[x][1] = fa[x] = siz[x] = siz2[x] = tag[x] = 0;
  }

  int getch(int x) { return ch[fa[x]][1] == x; }

  int isroot(int x) { return ch[fa[x]][0] != x && ch[fa[x]][1] != x; }

  void maintain(int x) {
    clear(0);
    if (x) siz[x] = siz[ch[x][0]] + 1 + siz[ch[x][1]] + siz2[x];
  }

  void pushdown(int x) {
    if (tag[x]) {
      if (ch[x][0]) swap(ch[ch[x][0]][0], ch[ch[x][0]][1]), tag[ch[x][0]] ^= 1;
      if (ch[x][1]) swap(ch[ch[x][1]][0], ch[ch[x][1]][1]), tag[ch[x][1]] ^= 1;
      tag[x] = 0;
    }
```

```
  }

  void update(int x) {
    if (!isroot(x)) update(fa[x]);
    pushdown(x);
  }

  void rotate(int x) {
    int y = fa[x], z = fa[y], chx = getch(x), chy = getch(y);
    fa[x] = z;
    if (!isroot(y)) ch[z][chy] = x;
    ch[y][chx] = ch[x][chx ^ 1];
    fa[ch[x][chx ^ 1]] = y;
    ch[x][chx ^ 1] = y;
    fa[y] = x;
    maintain(y);
    maintain(x);
    maintain(z);
  }

  void splay(int x) {
    update(x);
    for (int f = fa[x]; f = fa[x], !isroot(x); rotate(x))
      if (!isroot(f)) rotate(getch(x) == getch(f) ? f : x);
  }

  void access(int x) {
    for (int f = 0; x; f = x, x = fa[x])
      splay(x), siz2[x] += siz[ch[x][1]] - siz[f], ch[x][1] = f, maintain(x);
  }

  void makeroot(int x) {
    access(x);
    splay(x);
    swap(ch[x][0], ch[x][1]);
    tag[x] ^= 1;
  }

  int find(int x) {
    access(x);
    splay(x);
    while (ch[x][0]) x = ch[x][0];
    splay(x);
    return x;
  }
} st;

int n, q, x, y;
char op;

int main() {
  scanf("%d%d", &n, &q);
  while (q--) {
    scanf(" %c%d%d", &op, &x, &y);
```

```
    if (op == 'A') {
      st.makeroot(x);
      st.makeroot(y);
      st.fa[x] = y;
      st.siz2[y] += st.siz[x];
    }
    if (op == 'Q') {
      st.makeroot(x);
      st.access(y);
      st.splay(y);
      st.ch[y][0] = st.fa[x] = 0;
      st.maintain(x);
      st.makeroot(x);
      st.makeroot(y);
      printf("%lld\n", (ll)(st.siz[x] * st.siz[y]));
      st.makeroot(x);
      st.makeroot(y);
      st.fa[x] = y;
      st.siz2[y] += st.siz[x];
    }
  }
  return 0;
}
```

习题

- luogu P4299 首都[17]
- SPOJ QTREE5 - Query on a tree V[18]

参考资料与注释

[1] 「BZOJ 3282」Tree

[2] 「HNOI2010」弹飞绵羊

[3] 「国家集训队」Tree II

[4] luogu P3690【模板】Link Cut Tree（动态树）

[5] 「SDOI2011」染色

[6] 「SHOI2014」三叉神经树

[7] 「SDOI2008」洞穴勘测

[8] 「AHOI2005」航线规划

[9] luogu P3950 部落冲突

[10] bzoj 4998 星球联盟

[11] bzoj 2959 长跑

[12] luogu P4234 最小差值生成树

[13] 「WC2006」水管局长

[14] 「BJWC2010」严格次小生成树

[15] 「NOI2014」魔法森林

[16] 「BJOI2014」大融合

[17] luogu P4299 首都

[18] SPOJ QTREE5 - Query on a tree V

10.22.2 全局平衡二叉树

引入

前置知识：树链剖分

由于树链剖分的时间复杂度为 $O(n\log^2 n)$，而我们熟知的 LCT 虽然时间复杂度为 $O(n\log n)$，但常数较大，可能比树链剖分还慢。那么有什么既是 $O(n\log n)$ 的，常数又相对较小的方法呢？这个时候全局平衡二叉树就出现了。

全局平衡二叉树实际上是一颗二叉树森林，其中的每颗二叉树维护一条重链。但是这个森林里的二叉树又互有联系，其中每个二叉树的根连向这个重链链头的父亲，就像 LCT 中一样。但全局平衡二叉树是静态树，区别于 LCT，建成后树的形态不变。

全局平衡二叉树是一种可以处理树上链修改/查询的数据结构，可以做到：

- $O(\log n)$ 一条链整体修改。
- $O(\log n)$ 一条链整体查询。
- $O(\log n)$ 求最近公共祖先，子树修改，子树查询等，这些复杂度和重链剖分是一样的。

主要性质

1. 全局平衡二叉树由很多棵二叉树通过轻边连起来组成，每一棵二叉树维护了原树的一条重链，其中序遍历的顺序就是这条重链深度单调递增的顺序。每个节点都仅出现在一棵二叉树中。
2. 边分为重边和轻边，重边是包含在二叉树中的边，维护的时候就像正常维护二叉树一样，记录左右儿子和父节点。轻边从一颗二叉树的根节点指向它所对应的重链顶端节点的父节点。轻边维护的时候”认父不认子”，即只能从子节点访问到父节点，不能反过来。注意，全局平衡二叉树中的边和原树中的边没有对应关系。
3. 算上重边和轻边，全局平衡二叉树的高度是 $O(\log n)$ 级别的。这条是保证全局平衡二叉树时间复杂度的性质。

下面是一个全局平衡二叉树建树的例子。第一张图是原树，以节点 1 为根节点。实线是重边。

第二张图是建出来的全局平衡二叉树，其中虚线是轻边，实线是重边，每一棵二叉树用红圈表示。

建树

首先是像普通重链剖分一样，一次 DFS 求出每个节点的重儿子。然后从根开始，找到根节点所在的重链，对于这些点的轻儿子递归建树，并连上轻边。然后我们需要给重链上的点建一棵二叉树。我们先把重链上的点存到数组里，

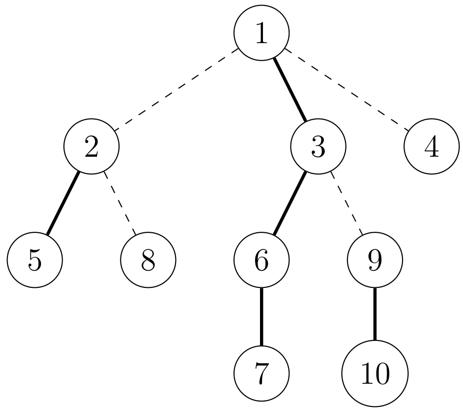


图 10.180 global-bst-1

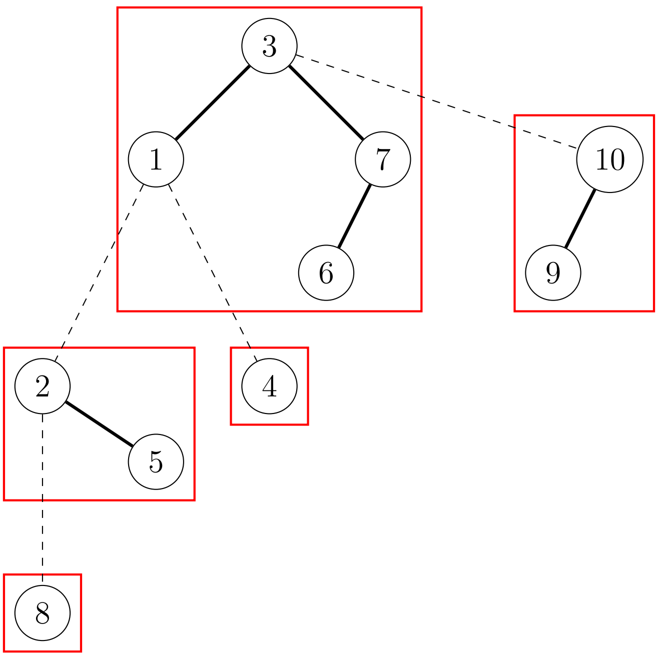


图 10.181 global-bst-2

求出每个点轻儿子的子树大小之和加一（即该点本身所贡献的 size）。然后我们按照这个求出这条重链的加权中点，把它作为二叉树的根，两边递归建树，并连上重边。

代码如下：

" 实现"

```
std::vector<int> G[N];
int n, fa[N], son[N], sz[N];

void dfsS(int u) {
  sz[u] = 1;
  for (int v : G[u]) {
    dfsS(v);
    sz[u] += sz[v];
    if (sz[v] > sz[son[u]]) son[u] = v;
  }
}

int b[N], bs[N], l[N], r[N], f[N], ss[N];

// 给 b 中 [bl,br) 内的点建二叉树，返回二叉树的根
int cbuild(int bl, int br) {
  int x = bl, y = br;
  while (y - x > 1) {
    int mid = (x + y) >> 1;
    if (2 * (bs[mid] - bs[bl]) <= bs[br] - bs[bl])
      x = mid;
    else
      y = mid;
  }
  // 二分求出按 bs 加权的中点
  y = b[x];
  ss[y] = br - bl;  // ss: 二叉树中重子树的大小
  if (bl < x) {
    l[y] = cbuild(bl, x);
    f[l[y]] = y;
  }
  if (x + 1 < br) {
    r[y] = cbuild(x + 1, br);
    f[r[y]] = y;
  }
  return y;
}

int build(int x) {
  int y = x;
  do
    for (int v : G[y])
      if (v != son[y])
        f[build(v)] =
            y;  // 递归建树并连轻边，注意要从二叉树的根连边，不是从儿子连边
  while (y = son[y]);
  y = 0;
  do {
```

```
    b[y++] = x;                           // 存放重链中的点
    bs[y] = bs[y - 1] + sz[x] - sz[son[x]];  // bs: 轻儿子 size 和 +1, 求前缀和
  } while (x = son[x]);
  return cbuild(0, y);
}
```

由代码可以看出建树的时间复杂度是 $O(n\log n)$。接下来我们可以证明树高是 $O(\log n)$ 的：考虑从任意一个点跳父节点到根。跳轻边就相当于在原树中跳到另一条重链，由重链剖分的性质可得跳轻边最多 $O(\log n)$ 条；因为建二叉树的时候根节点找的是算轻儿子的加权中点，那么跳一次重边算上轻儿子的 size 至少翻倍，所以跳重边最多也是 $O(\log n)$ 条。整体树高就是 $O(\log n)$ 的。

查询

以上就是关于全局平衡二叉树的部分。剩下关于链修改和链查询的操作方法相对简单，只需要从要操作的点出发，一直跳跃到根节点。要操作某个点所在的重链上比它深度小的所有点，本质上等同于在这条重链的二叉树中操作目标节点左侧的所有节点。这些操作可以分解成一系列子树操作，与普通二叉树的维护方法类似，其中涉及到维护子树和以及打子树标记。在这一过程中，使用的是标记永久化。也可以用 pushdown 来打标记，用 pushup 维护子树和，不过这种方式可能相对复杂，因为通常情况下，处理二叉树是自上而下进行操作，但在这里，需要首先确定跳跃路径，然后再从上到下进行 pushdown，可能导致常数较大。

代码如下：

"实现"

```
// a: 子树加标记
// s: 子树和 (不算加标记的)
int a[N], s[N];

void add(int x) {
  bool t = true;
  int z = 0;
  while (x) {
    s[x] += z;
    if (t) {
      a[x]++;
      if (r[x]) a[r[x]]--;
      z += 1 + ss[l[x]];
      s[x] -= ss[r[x]];
    }
    t = (x != l[f[x]]);
    if (t && x != r[f[x]]) z = 0;  // 跳过轻边要清空
    x = f[x];
  }
}

int query(int x) {
  int ret = 0;
  bool t = true;
  int z = 0;
  while (x) {
    if (t) {
      ret += s[x] - s[r[x]];
      ret -= 1ll * ss[r[x]] * a[r[x]];
      z += 1 + ss[l[x]];
```

```
    }
    ret += 1ll * z * a[x];
    t = (x != l[f[x]]);
    if (t && x != r[f[x]]) z = 0;  // 跳过轻边要清空
    x = f[x];
  }
  return ret;
}
```

此外，对于子树操作，就是要考虑轻儿子的，需要再维护一个包括轻儿子的子树和、子树标记，可以去做"P3384【模板】轻重链剖分[1]"。

例题

"P4751【模板】"动态 DP"&动态树分治（加强版）[2]"

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#define MAXN 1000000
#define MAXM 3000000
#define INF 0x3FFFFFFF
using namespace std;

struct edge {
  int to;
  edge *nxt;
} edges[MAXN * 2 + 5];

edge *ncnt = &edges[0], *Adj[MAXN + 5];
int n, m;

struct Matrix {
  int M[2][2];

  Matrix operator*(const Matrix &B) {
    static Matrix ret;
    for (int i = 0; i < 2; i++)
      for (int j = 0; j < 2; j++) {
        ret.M[i][j] = -INF;
        for (int k = 0; k < 2; k++)
          ret.M[i][j] = max(ret.M[i][j], M[i][k] + B.M[k][j]);
      }
    return ret;
  }
} matr1[MAXN + 5], matr2[MAXN + 5];  // 每个点维护两个矩阵

int root;
int w[MAXN + 5], dep[MAXN + 5], son[MAXN + 5], siz[MAXN + 5], lsiz[MAXN + 5];
int g[MAXN + 5][2], f[MAXN + 5][2], trfa[MAXN + 5], bstch[MAXN + 5][2];
int stk[MAXN + 5], tp;
bool vis[MAXN + 5];
```

```
void AddEdge(int u, int v) {
  edge *p = ++ncnt;
  p->to = v;
  p->nxt = Adj[u];
  Adj[u] = p;

  edge *q = ++ncnt;
  q->to = u;
  q->nxt = Adj[v];
  Adj[v] = q;
}

void DFS(int u, int fa) {
  siz[u] = 1;
  for (edge *p = Adj[u]; p != NULL; p = p->nxt) {
    int v = p->to;
    if (v == fa) continue;
    dep[v] = dep[u] + 1;
    DFS(v, u);
    siz[u] += siz[v];
    if (!son[u] || siz[son[u]] < siz[v]) son[u] = v;
  }
  lsiz[u] = siz[u] - siz[son[u]];  // 轻儿子的 siz 和 +1
}

void DFS2(int u, int fa) {
  f[u][1] = w[u], f[u][0] = 0;
  g[u][1] = w[u], g[u][0] = 0;
  if (son[u]) {
    DFS2(son[u], u);
    f[u][0] += max(f[son[u]][0], f[son[u]][1]);
    f[u][1] += f[son[u]][0];
  }
  for (edge *p = Adj[u]; p != NULL; p = p->nxt) {
    int v = p->to;
    if (v == fa || v == son[u]) continue;
    DFS2(v, u);
    f[u][0] += max(f[v][0], f[v][1]);  // f[][] 就是正常的 DP 数组
    f[u][1] += f[v][0];
    g[u][0] += max(f[v][0], f[v][1]);  // g[][] 数组只统计了自己和轻儿子的信息
    g[u][1] += f[v][0];
  }
}

void PushUp(int u) {
  matr2[u] = matr1[u];  // matr1 是单点加上轻儿子的信息，matr2 是区间信息
  if (bstch[u][0]) matr2[u] = matr2[bstch[u][0]] * matr2[u];
  // 注意转移的方向，但是如果我们的矩乘定义不同，可能方向也会不同
  if (bstch[u][1]) matr2[u] = matr2[u] * matr2[bstch[u][1]];
}

int getmx2(int u) { return max(matr2[u].M[0][0], matr2[u].M[0][1]); }

int getmx1(int u) { return max(getmx2(u), matr2[u].M[1][0]); }
```

```
int SBuild(int l, int r) {
  if (l > r) return 0;
  int tot = 0;
  for (int i = l; i <= r; i++) tot += lsiz[stk[i]];
  for (int i = l, sumn = lsiz[stk[l]]; i <= r; i++, sumn += lsiz[stk[i]])
    if (sumn * 2 >= tot)  // 是重心了
    {
      int lch = SBuild(l, i - 1), rch = SBuild(i + 1, r);
      bstch[stk[i]][0] = lch;
      bstch[stk[i]][1] = rch;
      trfa[lch] = trfa[rch] = stk[i];
      PushUp(stk[i]);  // 将区间的信息统计上来
      return stk[i];
    }
  return 0;
}

int Build(int u) {
  for (int pos = u; pos; pos = son[pos]) vis[pos] = true;
  for (int pos = u; pos; pos = son[pos])
    for (edge *p = Adj[pos]; p != NULL; p = p->nxt)
      if (!vis[p->to])  // 是轻儿子
      {
        int v = p->to, ret = Build(v);
        trfa[ret] = pos;  // 轻儿子的 treefa[] 接上来
      }
  tp = 0;
  for (int pos = u; pos; pos = son[pos]) stk[++tp] = pos;  // 把重链取出来
  int ret = SBuild(1, tp);  // 对重链进行单独的 SBuild(我猜是 Special Build?)
  return ret;               // 返回当前重链的二叉树的根
}

void Modify(int u, int val) {
  matr1[u].M[1][0] += val - w[u];
  w[u] = val;
  for (int pos = u; pos; pos = trfa[pos])
    if (trfa[pos] && bstch[trfa[pos]][0] != pos && bstch[trfa[pos]][1] != pos) {
      matr1[trfa[pos]].M[0][0] -= getmx1(pos);
      matr1[trfa[pos]].M[0][1] = matr1[trfa[pos]].M[0][0];
      matr1[trfa[pos]].M[1][0] -= getmx2(pos);
      PushUp(pos);
      matr1[trfa[pos]].M[0][0] += getmx1(pos);
      matr1[trfa[pos]].M[0][1] = matr1[trfa[pos]].M[0][0];
      matr1[trfa[pos]].M[1][0] += getmx2(pos);
    } else
      PushUp(pos);
}

int read() {
  int ret = 0, f = 1;
  char c = 0;
  while (c < '0' || c > '9') {
    c = getchar();
```

```
    if (c == '-') f = -f;
  }
  ret = 10 * ret + c - '0';
  while (true) {
    c = getchar();
    if (c < '0' || c > '9') break;
    ret = 10 * ret + c - '0';
  }
  return ret * f;
}

void print(int x) {
  if (x == 0) return;
  print(x / 10);
  putchar(x % 10 + '0');
}

int main() {
  scanf("%d %d", &n, &m);
  for (int i = 1; i <= n; i++) w[i] = read();
  int u, v;
  for (int i = 1; i < n; i++) {
    u = read(), v = read();
    AddEdge(u, v);
  }
  DFS(1, -1);
  // 求重儿子
  DFS2(1, -1);
  // 求初始的 DP 值，也可以在 Build() 里面求，但是这样写就和树剖的写法统一了
  for (int i = 1; i <= n; i++) {
    matr1[i].M[0][0] = matr1[i].M[0][1] = g[i][0];
    matr1[i].M[1][0] = g[i][1], matr1[i].M[1][1] = -INF;  // 初始化矩阵
  }
  root = Build(1);  // root 即为根节点所在重链的重心
  int lastans = 0;
  for (int i = 1; i <= m; i++) {
    u = read(), v = read();
    u ^= lastans;  // 强制在线
    Modify(u, v);
    lastans = getmx1(root);  // 直接取值
    if (lastans == 0)
      putchar('0');
    else
      print(lastans);
    putchar('\n');
  }
  return 0;
}
```

参考

P4211 [LNOI2014] LCA｜ 全局平衡二叉树[3]

参考资料与注释

[1] P3384【模板】轻重链剖分

[2] P4751【模板】" 动态 DP"& 动态树分治（加强版）

[3] P4211 [LNOI2014] LCA | 全局平衡二叉树

10.22.3 Euler Tour Tree

Authors: Backl1ght

Euler Tour Tree（欧拉游览树，欧拉回路树，后文简称 ETT）是一种可以解决**动态树**问题的数据结构。ETT 将动态树的操作转换成了其 DFS 序列上的区间操作，再用其他数据结构来维护序列的区间操作，从而维护动态树的操作。例如，ETT 将动态树的加边操作转换成了多个序列拆分操作和序列合并操作，如果能维护序列拆分操作和序列合并操作，就能维护动态树的加边操作。

LCT 也是一种可以解决动态树问题的数据结构，相比 ETT 而言 LCT 会更加常见。LCT 其实更适用于维护树链的信息，而 ETT 更加适用于维护**子树**的信息。例如，ETT 可以维护子树最小值而 LCT 不能。

ETT 可以使用任意数据结构维护，只需要该数据结构支持对应的序列区间操作，以及在复杂度上满足要求。一般情况下会使用例如 Splay，Treap 等平衡二叉搜索树来维护序列，而这些数据结构维护区间操作的复杂度均为 $O(\log n)$，由此也可以在 $O(\log n)$ 的时间内维护动态树的操作。如果使用多叉平衡搜索树例如 B 树来维护区间操作，也可以做到更优的复杂度。

其实 ETT 可以理解为一种思想，就是通过维护某种和原树一一对应的序列，从而达到维护原树的目的，本文介绍的只是这个思想的一些可行的实现和应用。

树的欧拉回路表示

如果把一条树边看成两条有向边的话，那么就可以把一棵树表示成一个有向图的欧拉回路，称为树的欧拉回路表示（Euler tour representation，ETR）。

后面要维护的序列其实是 ETR 的一个变种，把树中的点看成了自环也加到了 ETR 中，但是由于原始论文中作者没有给它起新的名字，就还是叫它 ETR 吧。

可以通过下述算法得到树 T 的欧拉回路表示：

Input. A rooted tree T
Output. The dfs sequence of rooted tree T
ET(u)
    visit vertex u
    for all child v of u
        visit directed edge $u \to v$
        ET(v)
        visit directed edge $v \to u$

树 T 的欧拉回路表示 ETR(T) 初始为空，DFS 的过程中每次访问一个节点或者一条有向边时就将其加到 ETR(T) 的尾部，如此便可得到 ETR(T)。

若 T 中包含 n 个节点，则中包含 $2n-2$ 条有向边，而 DFS 的过程中，每个点和每条有向边都会被访问一次，所以 ETR(T) 的长度为 $3n-2$。

把点 u 看成是一个自环，这样 ETR(T) 就可以看成有向图中的一个欧拉回路。可以在欧拉回路的某处断开，将其看成是一些边的首尾相连组成的链；也可以把这样的链在断开处重新粘起来变回欧拉回路；还可以通过新增一些边把两个这样的链拼成一个新的欧拉回路。

后文中，如未说明默认维护的序列是树的欧拉回路表示。

ETT 的基本操作

以下 3 个操作算是 ETT 的基本操作，均可以转换成常数次序列的操作，所以这 3 个操作的复杂度和序列操作同阶。

这里给出的只是一种可行的实现，只要能用常数次序列操作把修改后对应的序列拼出来即可。

MakeRoot(u)

即换根操作。ETT 中的换根操作被转换成了 1 个序列拆分操作和 1 个序列合并操作，也可以理解成 1 个区间平移操作。

记包含点 u 的树为 T，当前其树根为 r，现在要将树根换成 u。树 T 对应的序列为 L，将 L 在 (u,u) 处拆分成序列 L^1 和 L^2，前者包含 L 中 (u,u) 之前的元素以及 (u,u)，后者包含剩余元素。则依次将 L^2 和 L^1 合并得到的序列，即为换根之后树对应的序列。

这里可以理解成对一个欧拉回路进行旋转操作，欧拉回路是一个环，旋转并不会改变欧拉回路的结构，也即不会改变树的结构，只是把点 u 旋转到了根的位置而已。

Insert(u, v)

即加边操作。ETT 中加边操作被转换成了 2 个序列拆分操作和 5 个序列合并操作。

记包含点 u 的树为 T_1，包含点 v 的树为 T_2，加边之后两颗树合并成了一颗树 T。树 T_1 对应的序列为 L_1，树 T_2 对应的序列为 L_2。

将 L_1 在 (u,u) 处拆分成序列 L_1^1 和 L_1^2，前者包含 L_1 中 (u,u) 之前的元素以及 (u,u)，后者包含剩余元素。类似地将 L_2 在 (v,v) 处拆分成序列 L_2^1 和 L_2^2。则依次将 $L_1^2, L_1^1, [(u,v)], L_2^2, L_2^1, [(v,u)]$ 合并即可得到树 T 对应的序列 L。

这里可以理解成两次换根操作，然后把两个欧拉回路在当前根的位置处断开，再用新加的两条有向边把两个欧拉回路拼成一个新的欧拉回路。

Delete(u, v)

即删边操作。ETT 中删边操作被转换成了 4 个序列拆分操作以及 1 个序列合并操作。

记包含边 (u,v) 和边 (v,u) 的树为 T，其对应序列为 L。删边之后 T 分成了两颗树。

将 L 拆分成 $L_1, [(u,v)], L_2, [(v,u)], L_3$，删边形成的两颗树对应的序列分别为 L_2 以及 L_1, L_3。注意，在序列 L 中 $[(u,v)]$ 有可能出现在 $[(v,u)]$ 的后面，此时可以先交换 u 和 v 的值然后再操作。

这里可以理解成把一个欧拉回路从两条有向边处断开形成两条链，然后两条链自己首尾相连形成两个新的欧拉回路。

实现

以下以非旋 Treap 为例介绍 ETT 的实现，需要读者事先了解使用非旋 Treap 维护区间操作的相关内容。

`Split` 和 `Merge` 都是非旋 Treap 的基本操作了，这里不再赘述。

SplitUp2(u)

假设 u 所在的序列为 L，将 L 在 u 处拆分成序列 L^1 和 L^2，前者包含 L 中 u 之前的元素以及 u，后者包含剩余元素。

如果 Treap 的每个节点额外维护自己的父亲的话，就可以实现 $O(\log n)$ 的时间内计算一个 Treap 节点对应的元素在序列中的位置，再根据位置去 `Split` 就可以实现上述功能。

也可以自底向上地拆分从而实现上述功能，这样做相比上述方法会更高效。具体就是，在从 u 对应的节点往根跳的过程中，根据二叉搜索树的性质就可以确定每个节点在 L 中位于 u 之前还是之后，根据这点就可以计算 u 在序列中的位置，也可以确定每个节点属于拆分后的哪一棵树。

```
/*
 * Bottom up split treap p into 2 treaps a and b.
 *   - a: a treap containing nodes with position less than or equal to p.
 *   - b: a treap containing nodes with postion greater than p.
 *
 * In the other word, split sequence containning p into two sequences, the first
 * one contains elements before p and element p, the second one contains
 * elements after p.
 */
static std::pair<Node*, Node*> SplitUp2(Node* p) {
  Node *a = nullptr, *b = nullptr;
  b = p->right_;
  if (b) b->parent_ = nullptr;
  p->right_ = nullptr;

  bool is_p_left_child_of_parent = false;
  bool is_from_left_child = false;
  while (p) {
    Node* parent = p->parent_;

    if (parent) {
      is_p_left_child_of_parent = (parent->left_ == p);
      if (is_p_left_child_of_parent) {
        parent->left_ = nullptr;
      } else {
        parent->right_ = nullptr;
      }
      p->parent_ = nullptr;
    }

    if (!is_from_left_child) {
      a = Merge(p, a);
    } else {
      b = Merge(b, p);
    }

    is_from_left_child = is_p_left_child_of_parent;
    p->Maintain();
    p = parent;
  }

  return {a, b};
}
```

SplitUp3(u)

假设 u 所在的序列为 L，将 L 在 u 处拆分成序列 L^1,u 和 L^2，前者包含 L 中 u 之前的元素，后者包含剩余元素。在 `SplitUp2` 的基础上稍作修改即可。

MakeRoot(u)

基于 `SplitUp2` 以及 `Merge` 易得。

```
void MakeRoot(int u) {
  Node* vertex_u = vertices_[u];
  auto [L1, L2] = Treap::SplitUp2(vertex_u);
  Treap::Merge(L2, L1);
}
```

Insert(u, v)

基于 SplitUp2 以及 Merge 易得。

```
void Insert(int u, int v) {
  Node* vertex_u = vertices_[u];
  Node* vertex_v = vertices_[v];

  Node* edge_uv = AllocateNode(u, v);
  Node* edge_vu = AllocateNode(v, u);
  tree_edges_[u][v] = edge_uv;
  tree_edges_[v][u] = edge_vu;

  auto [L11, L12] = Treap::SplitUp2(vertex_u);
  auto [L21, L22] = Treap::SplitUp2(vertex_v);

  Node* L = L12;
  L = Treap::Merge(L, L11);
  L = Treap::Merge(L, edge_uv);
  L = Treap::Merge(L, L22);
  L = Treap::Merge(L, L21);
  L = Treap::Merge(L, edge_vu);
}
```

Delete(u, v)

基于 SplitUp3 以及 Merge 易得。

```
void Delete(int u, int v) {
  Node* edge_uv = tree_edges_[u][v];
  Node* edge_vu = tree_edges_[v][u];
  tree_edges_[u].erase(v);
  tree_edges_[v].erase(u);

  int position_uv = Treap::GetPosition(edge_uv);
  int position_vu = Treap::GetPosition(edge_vu);
  if (position_uv > position_vu) {
    std::swap(edge_uv, edge_vu);
    std::swap(position_uv, position_vu);
  }

  auto [L1, uv, _] = Treap::SplitUp3(edge_uv);
  auto [L2, vu, L3] = Treap::SplitUp3(edge_vu);
  Treap::Merge(L1, L3);

  FreeNode(edge_uv);
  FreeNode(edge_vu);
}
```

维护连通性

点 u 和点 v 连通，当且仅当两个点属于同一棵树 T，即 (u,u) 和 (v,v) 属于 $\mathrm{ETR}(T)$，这可以根据点 u 和点 v 对应的 Treap 节点所在的 Treap 的根是否相同判断。

例题 P2147[SDOI2008] 洞穴勘测[1]

维护连通性的模板题。

参考代码

```cpp
#include <bits/stdc++.h>

#define CPPIO \
  std::ios::sync_with_stdio(false), std::cin.tie(0), std::cout.tie(0);
#define freep(p) p ? delete p, p = nullptr, void(1) : void(0)

#if defined(BACKLIGHT) && !defined(NASSERT)
#define ASSERT(x)                                                   \
  ((x) || (fprintf(stderr, "assertion failed (" __FILE__ ":%d): \"%s\"\n", \
                   __LINE__, #x),                                   \
           assert(false), false))
#else
#define ASSERT(x) ;
#endif

#ifdef BACKLIGHT
#include "debug.h"
#else
#define logd(...) ;
#endif

using i64 = int64_t;
using u64 = uint64_t;

void solve_case(int Case);

int main(int argc, char* argv[]) {
  CPPIO;
  int T = 1;
  // std::cin >> T;
  for (int t = 1; t <= T; ++t) {
    solve_case(t);
  }
  return 0;
}

/**
 * Dynamic Forest Maintained With Euler Tour Tree.
 *
 * As said in reference, link and cut operation of dynamic trees can be
 * transformed into sequence split and sequence merge operation, which can be
 * easily maintained using balanced search trees like Treap.
 *
 * @reference: Dynamic trees as search trees via euler tours, applied to the
```

```
 * network simplex algorithm by Robert E. Tarjan.
 * https://link.springer.com/article/10.1007/BF02614369
 */
class DynamicForest {
 private:
  static std::mt19937 rng_;

  struct Node {
    int size_;
    int priority_;

    Node* left_;
    Node* right_;
    Node* parent_;

    int from_;
    int to_;

    Node(int from, int to)
        : size_(1),
          priority_(rng_()),
          left_(nullptr),
          right_(nullptr),
          parent_(nullptr),
          from_(from),
          to_(to) {}

    void Maintain() {
      size_ = 1;
      if (left_) {
        size_ += left_->size_;
        left_->parent_ = this;
      }
      if (right_) {
        size_ += right_->size_;
        right_->parent_ = this;
      }
    }
  };

  static int GetSize(Node* p) { return p == nullptr ? 0 : p->size_; }

  static Node* FindRoot(Node* p) {
    if (!p) return nullptr;

    while (p->parent_ != nullptr) p = p->parent_;
    return p;
  }

  static std::string to_string(Node* p) {
    std::stringstream ss;

    ss << "Node [\n";
```

```
    std::function<void(Node*)> dfs = [&](Node* p) {
      if (!p) return;
      dfs(p->left_);
      ss << "(" << p->from_ << "," << p->to_ << "),";
      dfs(p->right_);
    };
    dfs(p);

    ss << "]\n\n";

    return ss.str();
  }

  Node* AllocateNode(int u, int v) {
    Node* p = new Node(u, v);
    return p;
  }

  void FreeNode(Node*& p) {
    if (p) {
      delete p;
      p = nullptr;
    }
  }

  /*
   * Dynamic Sequence Maintained using Treap.
   */
  class Treap {
   public:
    /**
     * Merge two treap a and b into a single treap, with keys in a less than
     * keys in b.
     *
     * In the other word, concating sequence a and sequence b.
     */
    static Node* Merge(Node* a, Node* b) {
      if (a == nullptr) return b;
      if (b == nullptr) return a;

      if (a->priority_ < b->priority_) {
        a->right_ = Merge(a->right_, b);
        a->Maintain();
        return a;
      } else {
        b->left_ = Merge(a, b->left_);
        b->Maintain();
        return b;
      }
    }

    /**
     * Get the number of nodes with keys less than or equal to the key of p.
     *
```

```
 * In the other word, the the 1-based index of p inside the sequencec
 * containing p.
 */
static int GetPosition(Node* p) {
  ASSERT(p != nullptr);

  int position = GetSize(p->left_) + 1;
  while (p) {
    if (p->parent_ && p == p->parent_->right_)
      position += GetSize(p->parent_->left_) + 1;
    p = p->parent_;
  }
  return position;
}

/**
 * Split sequence containning p into two sequences, the first one contains
 * the first k elements, the second one contains the remaining elements.
 */
static std::pair<Node*, Node*> Split(Node* p, int k) {
  if (!p) return {nullptr, nullptr};

  std::pair<Node*, Node*> result;

  if (GetSize(p->left_) < k) {
    auto right_result = Split(p->right_, k - GetSize(p->left_) - 1);
    p->right_ = right_result.first;

    result.first = p;
    result.second = right_result.second;
  } else {
    auto left_result = Split(p->left_, k);
    p->left_ = left_result.second;

    result.first = left_result.first;
    result.second = p;
  }

  p->Maintain();

  if (result.first) result.first->parent_ = nullptr;

  if (result.second) result.second->parent_ = nullptr;

  return result;
}

/*
 * Bottom up split treap p into 2 treaps a and b.
 *   - a: a treap containing nodes with position less than or equal to p.
 *   - b: a treap containing nodes with postion greater than p.
 *
 * In the other word, split sequence containning p into two sequences, the
 * first one contains elements before p and element p, the second one
```

```
   * contains elements after p.
   */
  static std::pair<Node*, Node*> SplitUp2(Node* p) {
    ASSERT(p != nullptr);

    Node *a = nullptr, *b = nullptr;
    b = p->right_;
    if (b) b->parent_ = nullptr;
    p->right_ = nullptr;

    bool is_p_left_child_of_parent = false;
    bool is_from_left_child = false;
    while (p) {
      Node* parent = p->parent_;

      if (parent) {
        is_p_left_child_of_parent = (parent->left_ == p);
        if (is_p_left_child_of_parent) {
          parent->left_ = nullptr;
        } else {
          parent->right_ = nullptr;
        }
        p->parent_ = nullptr;
      }

      if (!is_from_left_child) {
        a = Merge(p, a);
      } else {
        b = Merge(b, p);
      }

      is_from_left_child = is_p_left_child_of_parent;
      p->Maintain();
      p = parent;
    }

    return {a, b};
  }

  /*
   * Bottom up split treap p into 3 treaps a, b and c.
   *   - a: a treap containing nodes with key less than p.
   *   - b: a treap containing nodes with key greater than p.
   *   - c: a treap containing nodes with key equal p.
   *
   * In the other word, split sequence containning p into three sequences, the
   * first one contains elements before p, the second one contains element p,
   * the third one contains elements after p.
   */
  static std::tuple<Node*, Node*, Node*> SplitUp3(Node* p) {
    ASSERT(p != nullptr);

    Node* a = p->left_;
    if (a) a->parent_ = nullptr;
```

```
      p->left_ = nullptr;

      Node* b = p->right_;
      if (b) b->parent_ = nullptr;
      p->right_ = nullptr;

      Node* c = p;

      bool is_p_left_child_of_parent = false;
      bool is_from_left_child = false;
      Node* parent = p->parent_;
      if (parent) {
        is_p_left_child_of_parent = (parent->left_ == p);
        if (is_p_left_child_of_parent) {
          parent->left_ = nullptr;
        } else {
          parent->right_ = nullptr;
        }
        p->parent_ = nullptr;
      }
      is_from_left_child = is_p_left_child_of_parent;
      p->Maintain();
      p = parent;

      while (p) {
        Node* parent = p->parent_;

        if (parent) {
          is_p_left_child_of_parent = (parent->left_ == p);
          if (is_p_left_child_of_parent) {
            parent->left_ = nullptr;
          } else {
            parent->right_ = nullptr;
          }
          p->parent_ = nullptr;
        }

        if (!is_from_left_child) {
          a = Merge(p, a);
        } else {
          b = Merge(b, p);
        }

        is_from_left_child = is_p_left_child_of_parent;
        p->Maintain();
        p = parent;
      }

      return {a, c, b};
    }
  };

 public:
  DynamicForest(int n) : n_(n), vertices_(n_), tree_edges_(n_) {
```

```
    ASSERT(n_ > 0);

    for (int i = 0; i < n_; ++i) vertices_[i] = AllocateNode(i, i);
  }

  ~DynamicForest() {
    for (int i = 0; i < n_; ++i) {
      for (auto [_, e] : tree_edges_[i]) {
        FreeNode(e);
      }
    }
    for (int i = 0; i < n_; ++i) {
      FreeNode(vertices_[i]);
    }
  }

  void Insert(int u, int v) {
    ASSERT(not tree_edges_[u].count(v));
    ASSERT(not tree_edges_[v].count(u));

    Node* vertex_u = vertices_[u];
    Node* vertex_v = vertices_[v];

    Node* edge_uv = AllocateNode(u, v);
    Node* edge_vu = AllocateNode(v, u);
    tree_edges_[u][v] = edge_uv;
    tree_edges_[v][u] = edge_vu;

    int position_u = Treap::GetPosition(vertex_u);
    int position_v = Treap::GetPosition(vertex_v);

    auto [L11, L12] = Treap::SplitUp2(vertex_u);
    auto [L21, L22] = Treap::SplitUp2(vertex_v);

    ASSERT(GetSize(L11) == position_u);
    ASSERT(GetSize(L21) == position_v);

    Node* result = nullptr;
    result = Treap::Merge(result, L12);
    result = Treap::Merge(result, L11);
    result = Treap::Merge(result, edge_uv);
    result = Treap::Merge(result, L22);
    result = Treap::Merge(result, L21);
    result = Treap::Merge(result, edge_vu);
  }

  void Delete(int u, int v) {
    ASSERT(tree_edges_[u].count(v));
    ASSERT(tree_edges_[v].count(u));

    Node* edge_uv = tree_edges_[u][v];
    Node* edge_vu = tree_edges_[v][u];
    tree_edges_[u].erase(v);
    tree_edges_[v].erase(u);
```

```
    int position_uv = Treap::GetPosition(edge_uv);
    int position_vu = Treap::GetPosition(edge_vu);
    if (position_uv > position_vu) {
      std::swap(edge_uv, edge_vu);
      std::swap(position_uv, position_vu);
    }

    auto [L1, uv, _] = Treap::SplitUp3(edge_uv);
    ASSERT(GetSize(L1) == position_uv - 1);
    ASSERT(GetSize(uv) == 1);

    auto [L2, vu, L3] = Treap::SplitUp3(edge_vu);
    ASSERT(GetSize(L2) == position_vu - position_uv - 1);
    ASSERT(GetSize(vu) == 1);

    L1 = Treap::Merge(L1, L3);

    FreeNode(edge_uv);
    FreeNode(edge_vu);
  }

  bool IsConnected(int u, int v) {
    Node* vertex_u = vertices_[u];
    Node* vertex_v = vertices_[v];
    return FindRoot(vertex_u) == FindRoot(vertex_v);
  }

  std::string to_string() const {
    std::stringstream ss;

    ss << "DynamicForest [\n";

    std::function<void(Node*)> dfs = [&](Node* p) {
      if (!p) return;
      dfs(p->left_);
      ss << "(" << p->from_ << "," << p->to_ << "),";
      dfs(p->right_);
    };
    for (int i = 0; i < n_; ++i) {
      if (vertices_[i]->parent_ == nullptr) {
        ss << "  Component [";
        dfs(vertices_[i]);
        ss << "]\n";
      }
    }
    for (int i = 0; i < n_; ++i) {
      for (auto [_, j] : tree_edges_[i]) {
        if (j->parent_ == nullptr) {
          ss << "  Component [";
          dfs(j);
          ss << "]\n";
        }
      }
```

```
    }

    ss << "]\n\n";

    return ss.str();
  }

 private:
  int n_;
  std::vector<Node*> vertices_;
  std::vector<std::map<int, Node*>> tree_edges_;
};

std::mt19937 DynamicForest::rng_(
    std::chrono::steady_clock::now().time_since_epoch().count());

void solve_case(int Case) {
  int n, m;
  std::cin >> n >> m;

  DynamicForest t(n + 1);
  std::string op;
  int u, v;
  for (int i = 1; i <= m; ++i) {
    std::cin >> op;
    std::cin >> u >> v;
    if (op[0] == 'Q') {
      std::cout << (t.IsConnected(u, v) ? "Yes" : "No") << "\n";
    } else if (op[0] == 'C') {
      t.Insert(u, v);
    } else if (op[0] == 'D') {
      t.Delete(u, v);
    }
  }
}
```

维护子树信息

下面以子树节点数量为例进行说明。

对于 ETR(T) 中每一个元素，如果这个元素对应的是树中的点，则令其权值为 1；如果这个元素对应的是树中的边，则令其权值为 0。现在树 T 的节点数量就可以看成 ETR(T) 中元素的权值和，只需要再维护序列权值和即可实现维护子树节点数量。而序列权值和的维护是非旋 Treap 的经典操作了。

类似地，可以将子树最小值等操作转化成序列最小值等平衡树经典操作然后维护。

例题 LOJ #2230.「BJOI2014」 大融合[2]

参考代码

```
#include <bits/stdc++.h>

#define CPPIO \
  std::ios::sync_with_stdio(false), std::cin.tie(0), std::cout.tie(0);
```

```cpp
#define freep(p) p ? delete p, p = nullptr, void(1) : void(0)

#if defined(BACKLIGHT) && !defined(NASSERT)
#define ASSERT(x)                                                          \
  ((x) || (fprintf(stderr, "assertion failed (" __FILE__ ":%d): \"%s\"\n", \
                   __LINE__, #x),                                          \
           assert(false), false))
#else
#define ASSERT(x) ;
#endif

#ifdef BACKLIGHT
#include "debug.h"
#else
#define logd(...) ;
#endif

using i64 = int64_t;
using u64 = uint64_t;

void solve_case(int Case);

int main(int argc, char* argv[]) {
  CPPIO;
  int T = 1;
  // std::cin >> T;
  for (int t = 1; t <= T; ++t) {
    solve_case(t);
  }
  return 0;
}

/**
 * Dynamic Forest Maintained With Euler Tour Tree.
 *
 * As said in reference, link and cut operation of dynamic trees can be
 * transformed into sequence split and sequence merge operation, which can be
 * easily maintained using balanced search trees like Treap.
 *
 * @reference: Dynamic trees as search trees via euler tours, applied to the
 * network simplex algorithm by Robert E. Tarjan.
 * https://link.springer.com/article/10.1007/BF02614369
 */
class DynamicForest {
 private:
  static std::mt19937 rng_;

  struct Node {
    int size_;
    int priority_;

    Node* left_;
    Node* right_;
    Node* parent_;
```

```
  int from_;
  int to_;

  int num_vertex_;
  int num_edge_;

  Node(int from, int to)
      : size_(1),
        priority_(rng_()),
        left_(nullptr),
        right_(nullptr),
        parent_(nullptr),
        from_(from),
        to_(to),
        num_vertex_(from_ == to_ ? 1 : 0),
        num_edge_(from_ == to_ ? 0 : 1) {}

  void Maintain() {
    size_ = 1;
    num_vertex_ = from_ == to_ ? 1 : 0;
    num_edge_ = from_ == to_ ? 0 : 1;
    if (left_) {
      size_ += left_->size_;
      left_->parent_ = this;

      num_vertex_ += left_->num_vertex_;
      num_edge_ += left_->num_edge_;
    }
    if (right_) {
      size_ += right_->size_;
      right_->parent_ = this;

      num_vertex_ += right_->num_vertex_;
      num_edge_ += right_->num_edge_;
    }
  }
};

static int GetSize(Node* p) { return p == nullptr ? 0 : p->size_; }

static Node* FindRoot(Node* p) {
  if (!p) return nullptr;

  while (p->parent_ != nullptr) p = p->parent_;
  return p;
}

static std::string to_string(Node* p) {
  std::stringstream ss;

  ss << "Node [\n";

  std::function<void(Node*)> dfs = [&](Node* p) {
```

```
    if (!p) return;
    dfs(p->left_);
    ss << "(" << p->from_ << "," << p->to_ << "),";
    dfs(p->right_);
  };
  dfs(p);

  ss << "]\n\n";

  return ss.str();
}

Node* AllocateNode(int u, int v) {
  Node* p = new Node(u, v);
  return p;
}

void FreeNode(Node*& p) {
  if (p) {
    delete p;
    p = nullptr;
  }
}

/*
 * Dynamic Sequence Maintained using Treap.
 */
class Treap {
 public:
  /**
   * Merge two treap a and b into a single treap, with keys in a less than
   * keys in b.
   *
   * In the other word, concating sequence a and sequence b.
   */
  static Node* Merge(Node* a, Node* b) {
    if (a == nullptr) return b;
    if (b == nullptr) return a;

    if (a->priority_ < b->priority_) {
      a->right_ = Merge(a->right_, b);
      a->Maintain();
      return a;
    } else {
      b->left_ = Merge(a, b->left_);
      b->Maintain();
      return b;
    }
  }

  /**
   * Get the number of nodes with keys less than or equal to the key of p.
   *
   * In the other word, the the 1-based index of p inside the sequencec
```

```
 * containing p.
 */
static int GetPosition(Node* p) {
  ASSERT(p != nullptr);

  int position = GetSize(p->left_) + 1;
  while (p) {
    if (p->parent_ && p == p->parent_->right_)
      position += GetSize(p->parent_->left_) + 1;
    p = p->parent_;
  }
  return position;
}

/**
 * Split sequence containning p into two sequences, the first one contains
 * the first k elements, the second one contains the remaining elements.
 */
static std::pair<Node*, Node*> Split(Node* p, int k) {
  if (!p) return {nullptr, nullptr};

  std::pair<Node*, Node*> result;

  if (GetSize(p->left_) < k) {
    auto right_result = Split(p->right_, k - GetSize(p->left_) - 1);
    p->right_ = right_result.first;

    result.first = p;
    result.second = right_result.second;
  } else {
    auto left_result = Split(p->left_, k);
    p->left_ = left_result.second;

    result.first = left_result.first;
    result.second = p;
  }

  p->Maintain();

  if (result.first) result.first->parent_ = nullptr;

  if (result.second) result.second->parent_ = nullptr;

  return result;
}

/*
 * Bottom up split treap p into 2 treaps a and b.
 *   - a: a treap containing nodes with position less than or equal to p.
 *   - b: a treap containing nodes with postion greater than p.
 *
 * In the other word, split sequence containning p into two sequences, the
 * first one contains elements before p and element p, the second one
 * contains elements after p.
```

```
 */
static std::pair<Node*, Node*> SplitUp2(Node* p) {
  ASSERT(p != nullptr);

  Node *a = nullptr, *b = nullptr;
  b = p->right_;
  if (b) b->parent_ = nullptr;
  p->right_ = nullptr;

  bool is_p_left_child_of_parent = false;
  bool is_from_left_child = false;
  while (p) {
    Node* parent = p->parent_;

    if (parent) {
      is_p_left_child_of_parent = (parent->left_ == p);
      if (is_p_left_child_of_parent) {
        parent->left_ = nullptr;
      } else {
        parent->right_ = nullptr;
      }
      p->parent_ = nullptr;
    }

    if (!is_from_left_child) {
      a = Merge(p, a);
    } else {
      b = Merge(b, p);
    }

    is_from_left_child = is_p_left_child_of_parent;
    p->Maintain();
    p = parent;
  }

  return {a, b};
}

/*
 * Bottom up split treap p into 3 treaps a, b and c.
 *   - a: a treap containing nodes with key less than p.
 *   - b: a treap containing nodes with key greater than p.
 *   - c: a treap containing nodes with key equal p.
 *
 * In the other word, split sequence containning p into three sequences, the
 * first one contains elements before p, the second one contains element p,
 * the third one contains elements after p.
 */
static std::tuple<Node*, Node*, Node*> SplitUp3(Node* p) {
  ASSERT(p != nullptr);

  Node* a = p->left_;
  if (a) a->parent_ = nullptr;
  p->left_ = nullptr;
```

```
      Node* b = p->right_;
      if (b) b->parent_ = nullptr;
      p->right_ = nullptr;

      Node* c = p;

      bool is_p_left_child_of_parent = false;
      bool is_from_left_child = false;
      Node* parent = p->parent_;
      if (parent) {
        is_p_left_child_of_parent = (parent->left_ == p);
        if (is_p_left_child_of_parent) {
          parent->left_ = nullptr;
        } else {
          parent->right_ = nullptr;
        }
        p->parent_ = nullptr;
      }
      is_from_left_child = is_p_left_child_of_parent;
      p->Maintain();
      p = parent;

      while (p) {
        Node* parent = p->parent_;

        if (parent) {
          is_p_left_child_of_parent = (parent->left_ == p);
          if (is_p_left_child_of_parent) {
            parent->left_ = nullptr;
          } else {
            parent->right_ = nullptr;
          }
          p->parent_ = nullptr;
        }

        if (!is_from_left_child) {
          a = Merge(p, a);
        } else {
          b = Merge(b, p);
        }

        is_from_left_child = is_p_left_child_of_parent;
        p->Maintain();
        p = parent;
      }

      return {a, c, b};
    }
  };

public:
  DynamicForest(int n) : n_(n), vertices_(n_), tree_edges_(n_) {
    ASSERT(n_ > 0);
```

```
    for (int i = 0; i < n_; ++i) vertices_[i] = AllocateNode(i, i);
  }

  ~DynamicForest() {
    for (int i = 0; i < n_; ++i) {
      for (auto [_, e] : tree_edges_[i]) {
        FreeNode(e);
      }
    }
    for (int i = 0; i < n_; ++i) {
      FreeNode(vertices_[i]);
    }
  }

  void MakeRoot(int u) {
    Node* vertex_u = vertices_[u];

    int position_u = Treap::GetPosition(vertex_u);

    auto [L1, L2] = Treap::SplitUp2(vertex_u);
    ASSERT(GetSize(L1) == position_u);

    Treap::Merge(L2, L1);
  }

  void Insert(int u, int v) {
    ASSERT(not tree_edges_[u].count(v));
    ASSERT(not tree_edges_[v].count(u));

    Node* vertex_u = vertices_[u];
    Node* vertex_v = vertices_[v];

    Node* edge_uv = AllocateNode(u, v);
    Node* edge_vu = AllocateNode(v, u);
    tree_edges_[u][v] = edge_uv;
    tree_edges_[v][u] = edge_vu;

    int position_u = Treap::GetPosition(vertex_u);
    int position_v = Treap::GetPosition(vertex_v);

    auto [L11, L12] = Treap::SplitUp2(vertex_u);
    auto [L21, L22] = Treap::SplitUp2(vertex_v);

    ASSERT(GetSize(L11) == position_u);
    ASSERT(GetSize(L21) == position_v);

    Node* result = nullptr;
    result = Treap::Merge(result, L12);
    result = Treap::Merge(result, L11);
    result = Treap::Merge(result, edge_uv);
    result = Treap::Merge(result, L22);
    result = Treap::Merge(result, L21);
    result = Treap::Merge(result, edge_vu);
```

```
  }

  void Delete(int u, int v) {
    ASSERT(tree_edges_[u].count(v));
    ASSERT(tree_edges_[v].count(u));

    Node* edge_uv = tree_edges_[u][v];
    Node* edge_vu = tree_edges_[v][u];
    tree_edges_[u].erase(v);
    tree_edges_[v].erase(u);

    int position_uv = Treap::GetPosition(edge_uv);
    int position_vu = Treap::GetPosition(edge_vu);
    if (position_uv > position_vu) {
      std::swap(edge_uv, edge_vu);
      std::swap(position_uv, position_vu);
    }

    auto [L1, uv, _] = Treap::SplitUp3(edge_uv);
    ASSERT(GetSize(L1) == position_uv - 1);
    ASSERT(GetSize(uv) == 1);

    auto [L2, vu, L3] = Treap::SplitUp3(edge_vu);
    ASSERT(GetSize(L2) == position_vu - position_uv - 1);
    ASSERT(GetSize(vu) == 1);

    L1 = Treap::Merge(L1, L3);

    FreeNode(edge_uv);
    FreeNode(edge_vu);
  }

  bool IsConnected(int u, int v) {
    Node* vertex_u = vertices_[u];
    Node* vertex_v = vertices_[v];
    return FindRoot(vertex_u) == FindRoot(vertex_v);
  }

  int GetComponentSize(int u) {
    Node* vertex_u = vertices_[u];
    Node* root_of_vertex_u = FindRoot(vertex_u);
    return GetSize(root_of_vertex_u);
  }

  int GetComponentNumberOfVertex(int u) {
    Node* vertex_u = vertices_[u];
    Node* root_of_vertex_u = FindRoot(vertex_u);
    return root_of_vertex_u ? root_of_vertex_u->num_vertex_ : 0;
  }

  std::string to_string() const {
    std::stringstream ss;

    ss << "DynamicForest [\n";
```

```
    std::function<void(Node*)> dfs = [&](Node* p) {
      if (!p) return;
      dfs(p->left_);
      ss << "(" << p->from_ << "," << p->to_ << "),";
      dfs(p->right_);
    };
    for (int i = 0; i < n_; ++i) {
      if (vertices_[i]->parent_ == nullptr) {
        ss << "  Component [";
        dfs(vertices_[i]);
        ss << "]\n";
      }
    }
    for (int i = 0; i < n_; ++i) {
      for (auto [_, j] : tree_edges_[i]) {
        if (j->parent_ == nullptr) {
          ss << "  Component [";
          dfs(j);
          ss << "]\n";
        }
      }
    }

    ss << "]\n\n";

    return ss.str();
  }

 private:
  int n_;
  std::vector<Node*> vertices_;
  std::vector<std::map<int, Node*>> tree_edges_;
};

std::mt19937 DynamicForest::rng_(
    std::chrono::steady_clock::now().time_since_epoch().count());

void solve_case(int Case) {
  int n, q;
  std::cin >> n >> q;

  DynamicForest t(n + 1);
  std::string op;
  int u, v;
  for (int i = 1; i <= q; ++i) {
    std::cin >> op >> u >> v;
    if (op[0] == 'A') {
      t.Insert(u, v);
    } else if (op[0] == 'Q') {
      t.Delete(u, v);
      int ans = i64(1) * t.GetComponentNumberOfVertex(u) *
                t.GetComponentNumberOfVertex(v);
      t.Insert(u, v);
```

```
      std::cout << ans << "\n";
    }
  }
}
```

维护树链信息

可以使用一个比较常见的技巧就是借助括号序的性质将树链信息转化成区间信息，然后就可以借助数据结构维护序列从而维护树链信息了。但是这个技巧要求维护的信息满足**可减性**。

前面介绍的动态树操作对应的序列操作可能会把括号序中的右括号移动到左括号前，所以维护树链点权和之类的信息时还需要额外注意，操作时不能改变对应左右括号的先后顺序，而这可能需要重新思考动态树操作对应的序列操作，甚至重新思考维护什么 DFS 序。

此外，ETT 很难维护树链修改。

例题 「 星际探索 」[3]

这题的动态树操作只有换父亲，可以看成删边再加边，但是这样可能会改变对应括号的先后顺序。

可以把点权转成边权，维护树的括号序，换父亲操作转化成把整个子树对应的括号序列平移至父亲左括号后面。

参考代码

```
/*
虽然上文提到过块状链表实现 ETT
在某些情况下可能较简单，但对于此题块状链表复杂度有可能无法通过而且实现较繁琐，所以
这份代码采用
FHQ Treap 实现。
*/
#include <bits/stdc++.h>
#define N 1000000
#define int long long
using namespace std;
/*FHQ TREAP*/
int rt, tot, f[N], rnd[N], ls[N], rs[N], siz[N], tag[N], val[N], sum[N], pd[N],
    pds[N];

void pushup(int x) {
  siz[x] = siz[ls[x]] + siz[rs[x]] + 1;
  sum[x] = sum[ls[x]] + sum[rs[x]] + val[x];
  pds[x] = pds[ls[x]] + pds[rs[x]] + pd[x];
}

void link(int x, int c, int y) {
  if (c)
    rs[x] = y;
  else
    ls[x] = y;
  if (y) f[y] = x;
  pushup(x);
}

int newNode(int x, int y) {
  siz[++tot] = 1;
  val[tot] = sum[tot] = x;
```

```
  pd[tot] = pds[tot] = y;
  rnd[tot] = rand();
  return tot;
}

void setTag(int x, int v) {
  tag[x] += v;
  sum[x] += v * pds[x];
  val[x] += v * pd[x];
}

void pushdown(int x) {
  if (ls[x]) setTag(ls[x], tag[x]);
  if (rs[x]) setTag(rs[x], tag[x]);
  tag[x] = 0;
}

void split(int now, int k, int &x, int &y) {
  f[now] = 0;
  if (!now) {
    x = y = 0;
    return;
  }
  pushdown(now);
  if (siz[ls[now]] + 1 <= k) {
    x = now;
    split(rs[now], k - siz[ls[now]] - 1, rs[x], y);
    link(x, 1, rs[x]);
  } else {
    y = now;
    split(ls[now], k, x, ls[y]);
    link(y, 0, ls[y]);
  }
}

int merge(int x, int y) {
  if (!x || !y) return x | y;
  if (rnd[x] < rnd[y]) {
    pushdown(x);
    link(x, 1, merge(rs[x], y));
    return x;
  } else {
    pushdown(y);
    link(y, 0, merge(x, ls[y]));
    return y;
  }
}

int rnk(int x) {
  int c = 1, ans = 0;
  while (x) {
    if (c) ans += siz[ls[x]] + 1;
    c = (rs[f[x]] == x);
    x = f[x];
```

```
  }
  return ans;
}

/*ETT*/
int s[N], e[N];

void add(int x, int v) {
  int a, b, c;
  split(rt, rnk(s[x]) - 1, a, b);
  split(b, rnk(e[x]) - rnk(s[x]) + 1, b,
        c);  // 这里 b 是我们要进行操作的子树的括号序列。
  setTag(b, v);
  rt = merge(merge(a, b), c);
}

int query(int x) {
  int a, b;
  split(rt, rnk(s[x]), a, b);
  int ans = sum[a];
  rt = merge(a, b);
  return ans;
}

void changeFa(int x, int y) {
  int a, b, c, d;
  split(rt, rnk(s[x]) - 1, a, b);
  split(b, rnk(e[x]) - rnk(s[x]) + 1, b, c);
  a = merge(
      a,
      c);  // 因为我们确定不了要设置为父亲的节点在括号序列中的哪边，所以先把两边合
并。
  split(a, rnk(s[y]), a, d);
  rt = merge(merge(a, b), d);  // 把要进行操作的子树放在父亲括号序列的最前面。
}

/*main function*/
int n, m, w[N];
vector<int> v[N];

void dfs(int x) {
  rt = merge(rt, s[x] = newNode(w[x], 1));
  for (auto to : v[x]) dfs(to);
  rt = merge(rt, e[x] = newNode(-w[x], -1));
}

signed main() {
  cin >> n;
  for (int i = 2; i <= n; i++) {
    int f;
    cin >> f;
    v[f].push_back(i);
  }
  for (int i = 1; i <= n; i++) cin >> w[i];
```

```
  dfs(1);
  cin >> m;
  for (int i = 1; i <= m; i++) {
    char c;
    cin >> c;
    if (c == 'Q') {
      int d;
      cin >> d;
      cout << query(d) << endl;
    } else if (c == 'C') {
      int x, y;
      cin >> x >> y;
      changeFa(x, y);
    } else {
      int p, q;
      cin >> p >> q;
      add(p, q);
    }
  }
  return 0;
}
```

参考资料

- Dynamic trees as search trees via euler tours, applied to the network simplex algorithm - Robert E. Tarjan
- Randomized fully dynamic graph algorithms with polylogarithmic time per operation - Henzinger et al.

参考资料与注释

[1] P2147[SDOI2008] 洞穴勘测

[2] LOJ #2230.「BJOI2014」大融合

[3] 「星际探索」

10.22.4 Top Tree

author:F7487

Self-Adjusting Top Tree

简介

Self-Adjusting Top Tree，是 2005 年 Tarjan 和 Werneck 在他们的论文《Self-Adjusting Top Trees》中提出的一种基于 Top Tree 理论的维护完全动态森林的数据结构，简称为 SATT。

Self-Adjusting Top Tree 可以实现森林中任一棵树的链修改/查询、子树修改/查询以及非局部搜索等操作。

Splay Tree 是 SATT 的基础，但是 SATT 用的 Splay Tree 和普通的 Splay 在细节处不太一样（进行了一些扩展）。

问题引入

维护一个森林，支持如下操作：

- 删除，添加一条边，保证操作前后仍是一个森林。

- 修改某棵树上某条简单路径的权值。
- 修改以某个点为根的子树权值。
- 查询某棵树上的某条简单路径权值和。
- 查询以某个点为根的子树权值和。

树收缩

对于任意一棵树，我们都可以运用**树收缩**理论来将它收缩为一条边。

具体地，树收缩有两个基本操作：**Compress** 和 **Rake**，Compress 操作指定一个度数为二的点 x，与点 x 相邻的那两个点记为 y、z，我们连一条新边 yz；将点 x、边 xz、边 xy 的信息放到 yz 中储存，并删去它们。如图所示。

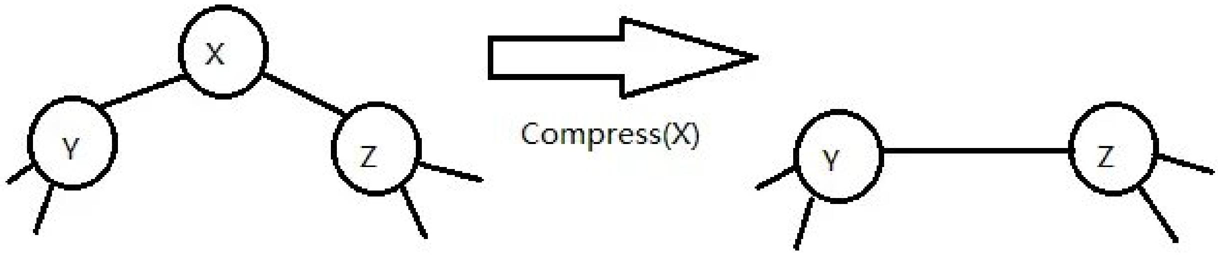


图 10.182

Rake 操作指定一个度为一的点 x，而且与点 x 相邻的点 y 的度数需大于一，设点 y 的另一个邻点为 z，我们将点 x、边 xy 的信息放入边 yz 中储存，并删去它们。如图所示。

不难证明，任何一棵树都可以只用 Compress 操作和 Rake 操作来将它收缩为一条边，如图所示。

簇

为了表达方便，我们记在进行任何操作之前的原树为 T。在对 T 进行某些树收缩操作（可以不做任何操作）之后的树记为 T_x。

我们研究某个 T_x 中某一条边所包含的信息情况。

这条边除了带有它本身的信息（当然，如果这条边在 T 中不存在，这条边就没有本身的信息）之外，还可能包含其它通过 Compress/Rake 操作合并到它上面的点、边的信息。我们不妨先从下图中的树收缩过程中选取一条边，看看它所包含的信息在 T 中代表哪些点、边。

如图，选取的边和对应的图已用红线圈出。

可以看出，这条边所包含的信息在 T 中代表的点、边是连通的。我们可以推及，对于任一 T_x 中的任一条边储存的信息在 T 中总体现为一个连通子图。我们将这样的连通子图称为**簇（Cluster）**。

然而，簇是**不完整的子图**，它包含的某些边的端点不被簇它自己包含。于是我们将这些端点称作簇的**端点（Endpoint）**，将它包含的那些连通子图的点称作**内点（Internal Node）**，连通子图的边称作**内边（Internal Edge）**。

对于任意一个簇，都有以下性质：

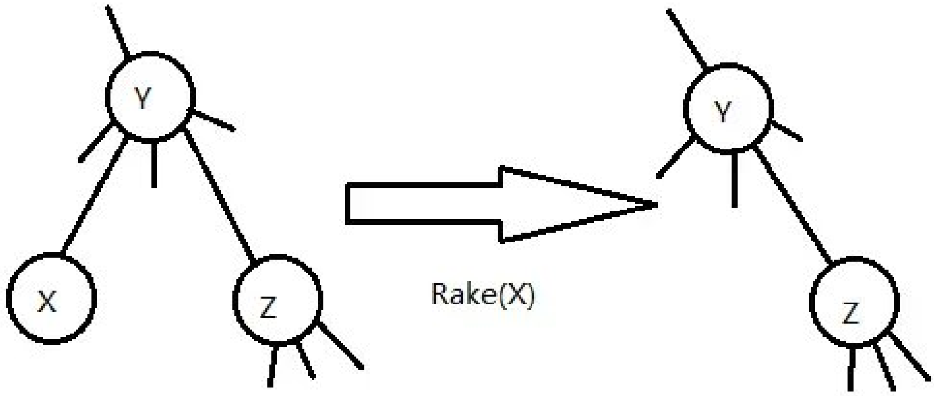


图 10.183

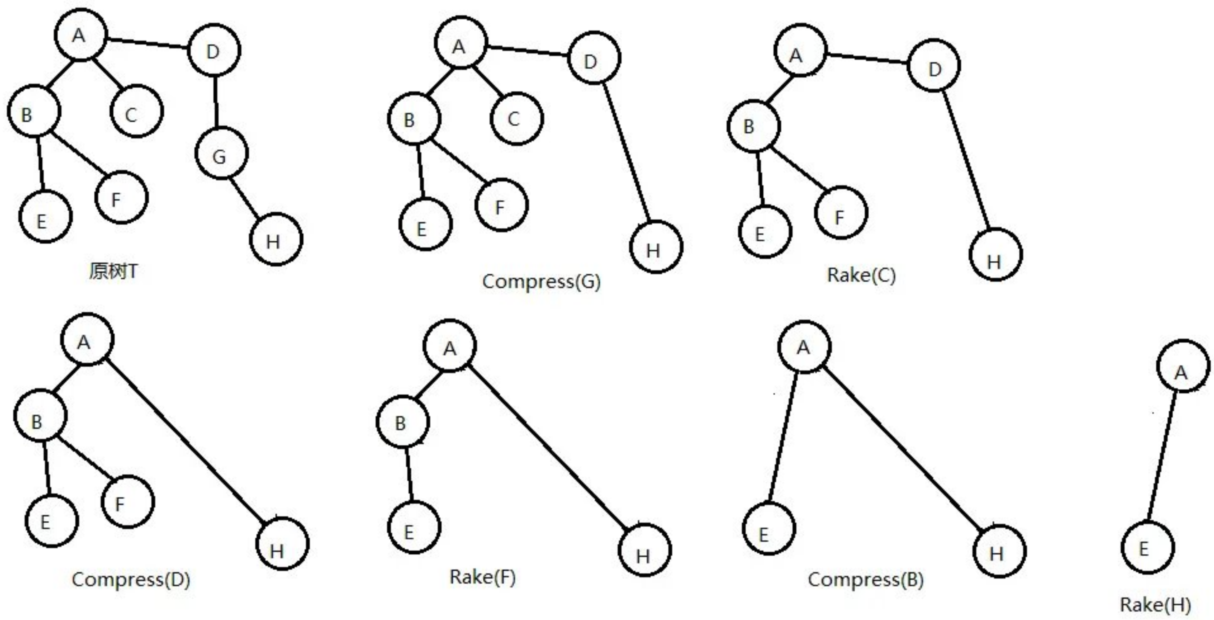


图 10.184

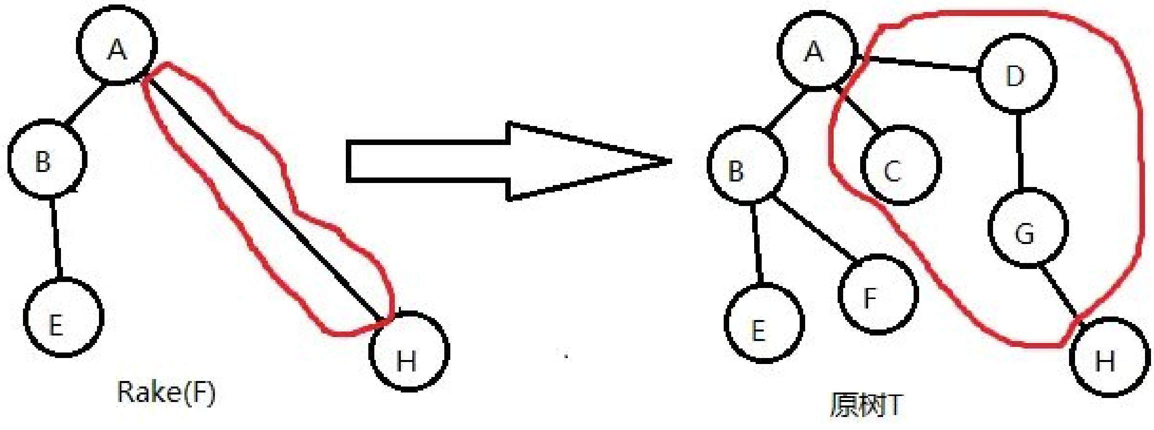


图 10.185

1. 簇只存储和维护内点和内边的信息。
2. 簇有两个端点。这两个端点即为 T_x 中代表那个簇的边相连的那两个点。两个端点之间的路径我们称之为**簇路径（Cluster Path）**；记一个簇的两个端点分别为 x、y，我们下面用 $C(x, y)$ 来表示这个簇。
3. 内点仅与端点或内点相连。

特别地，对于 T 中的每条边，都各自独立为一个簇（仅包含边自己的信息），这种簇我们称之为**基簇（Base Cluster）**。对于由 T 收缩到只有一条边的最终的 T_x，那条边代表的簇包含除了两个端点之外的整棵 T 的信息，这个簇我们称之为**根簇（Root Cluster）**。

如图，上文提到的基簇已用红线圈出。

从簇的视角来看 Compress/Rake 操作，我们发现这两个操作会将两个簇“合二为一”，剩下一个新簇，所以树收缩的过程也是所有的基簇合并为一个簇的过程。

所以我们也可以得到下图，是对一系列树收缩操作的另一表示。

Top Tree

我们现在想表示某一棵树进行树收缩的全过程。

我们可以用上文的两种方法来表示这一过程，但这样十分麻烦，如果树收缩进行了 n 步，我们就要用 n 棵树来表示整个树收缩。

考虑一个对某棵树进行某一树收缩的更简便表示，我们引入 **Top Tree**。

如图，是以上文的收缩方法和原树为基础的一棵 Top Tree。

Top Tree 有以下性质：

1. 一棵 Top Tree 对应一棵原树和一种对其进行树收缩的方法，Top Tree 的每个节点都表示在某个 T_x 中的某一条边，也就是树收缩过程中形成的某一个簇。图中的形如 N_x 的点表示 `compress(x)` 这一操作形成的簇。
2. Top Tree 中的一个节点有两个儿子（都分别代表一个簇），这个节点代表的簇是这两个簇通过 Compress 或 Rake 操作合并得到的新簇。
3. Top Tree 的叶子节点是基簇，其根节点是根簇。因此我们按一棵 Top Tree 的拓扑序分层，它的每一层就代表了一棵 T_x。

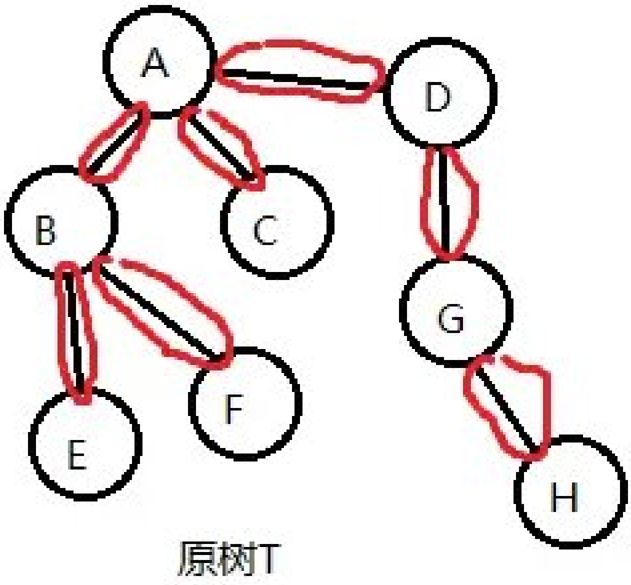


图 10.186

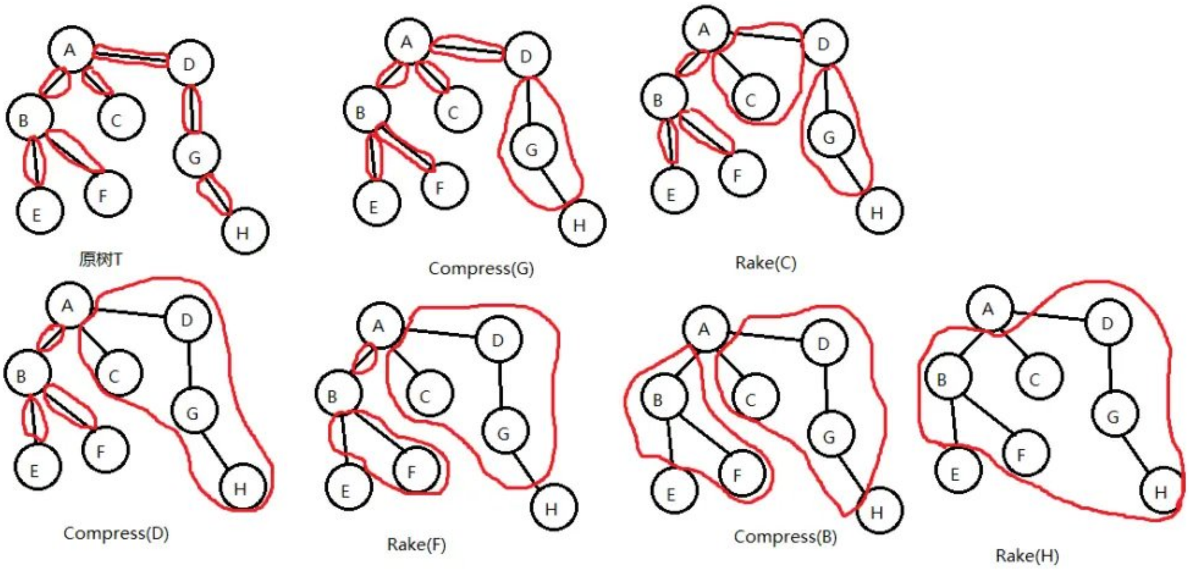


图 10.187

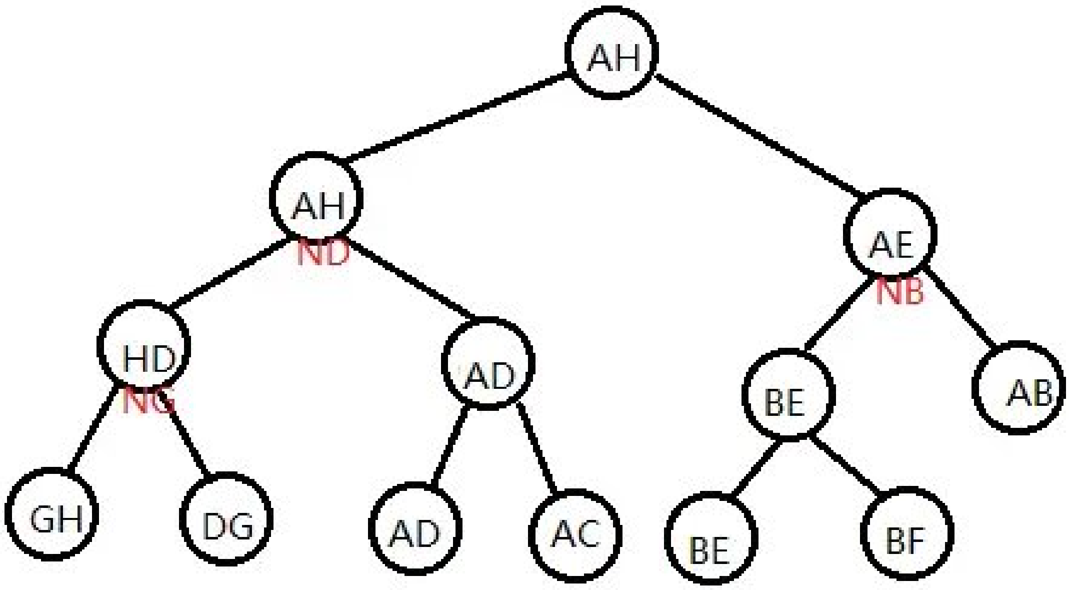


图 10.188

用三度化 Self-Adjusting Top Tree 实现信息维护

原理

Top Tree 对树收缩过程的极大简化使我们看到通过维护树收缩过程来维护树上信息的可能性，SATT 即是通过这一原理来维护树上信息的。

注意到树收缩的过程也是树上信息不断加入的过程，我们执行一次 `compress(x)`，x 点的信息从此刻起就开始在某个簇中出现，影响着我们的统计结果。

假如我们现在用 Top Tree 来维护某棵树 T，树上的每个点，边都有权值，我们要维护的是 T 的权值和。

现在我们在维护时要对 T 中某个点 x 的权值进行修改，很明显，我们就需要更改 Top Tree 中所有簇信息包含 x 的节点信息，这样做单次时间复杂度会是 $O(n)$ 级别的。

然而，如果我们选的点它在 Top Tree 中簇信息包含 x 的节点个数很少，也就是说使它的信息尽可能晚地加入簇中，我们单次操作的时间复杂度就会有一个很大的提升。如图。

SATT 就是通过修改**某个点/某条路径**在树收缩过程中信息被加入簇中的先后顺序（以降低其在被修改时的单次时间复杂度）来维护树上信息的。

实际结构

我们先将一棵原树 T 分层定根，然后我们考虑对某种树收缩顺序的 Top Tree 的根簇，它有两个端点，我们令这其中一个端点就是原树的根，另一个端点任选。

如图，给根簇选出一组端点，这里标注簇时将端点也圈进去了。

由树收缩的基本操作可知，簇路径上的点、边 (j, h, c, jh, hc) 的信息最后是通过 Compress 操作才加入 $C(k, g)$ 的，而的非簇路径点 $(a, b, i, f, g, e, ig, \cdots)$ 是通过 Rake 操作才加入 $C(k, g)$ 的。

我们将簇路径单独拿出来，这是一条形态特殊（为链）的树，我们为这棵树建出一棵 top tree（其代表的树收缩顺序任意）。

我们将这一结构称之为 **Compress Tree**，因为在这棵 Top Tree 中任一个点的两个儿子之间是通过 Compress 操作来合并成它们的父亲。

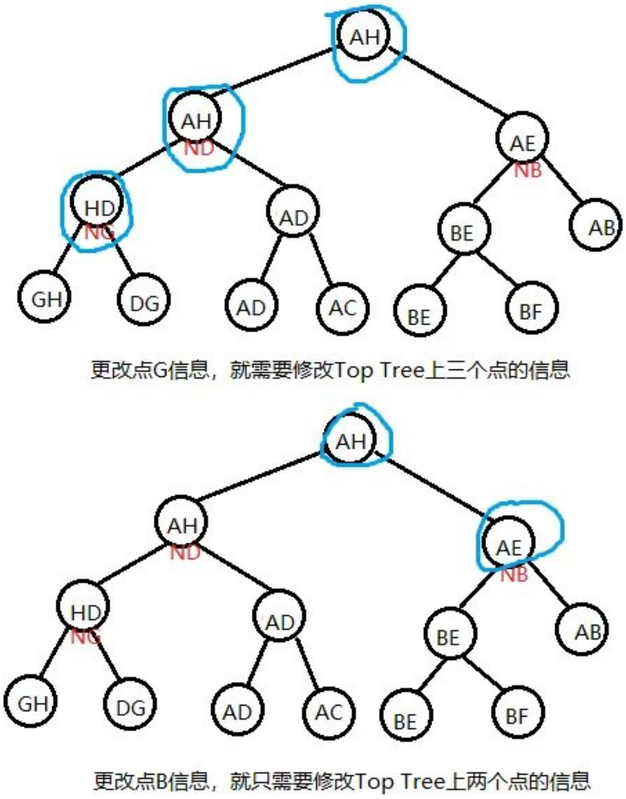


图 10.189

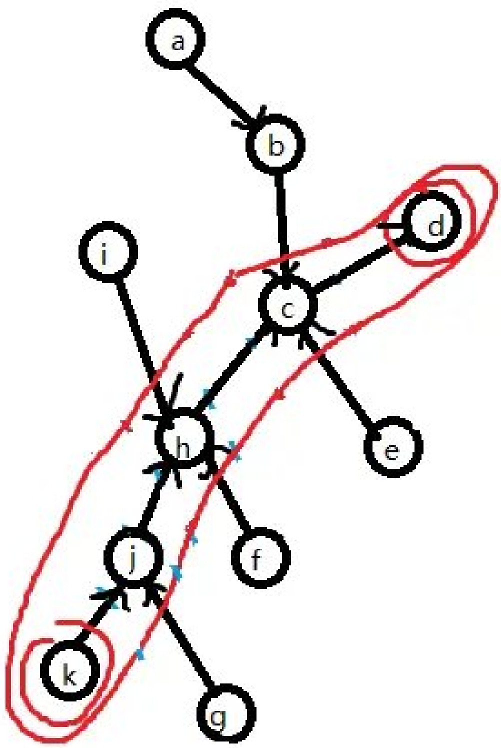


图 10.190

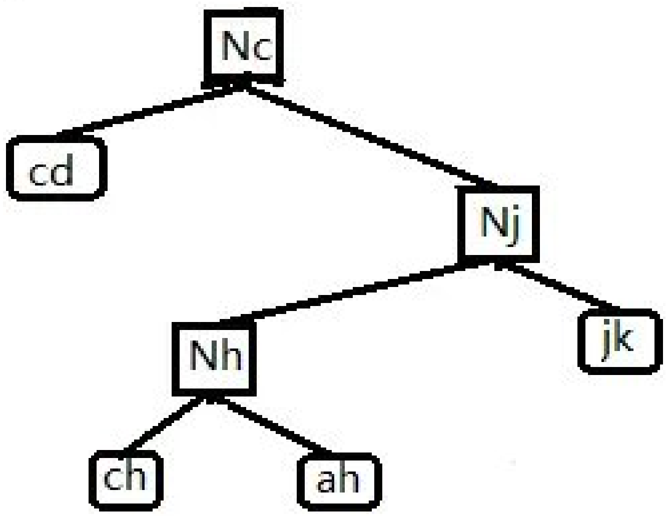


图 10.191

Compress Tree 里的节点称为 **Compress Node**。只考虑当前这条簇路径，一个非叶子的 Compress Node 就代表一次 compress 过程，表示将左儿子和右儿子信息合并起来，再将这个 $compress(x)$ 本身存储的点 x 信息加入。这棵 Compress Tree 就维护了 $C(k,g)$ 簇路径的信息。

另外，在 Compress Tree 中，我们实际上还对使用的 Top Tree 做了一些限制。注意到 compress tree 维护的是一个 T 中点的深度两两不同的链，我们规定在 Compress Tree 中基簇的中序遍历顺序与对应的 T 中边的深度是一致的，且中序遍历越小深度越浅。同样，对于每个点 x 对应的 $compress(x)$ 的关系也是如此。

现在来维护那些非簇路径的信息，我们假设这些非簇路径上的点、边已经形成了一个个极大簇，而这些极大簇是由这些用蓝线圈出的更小簇之间互相 Rake 形成的，对由一些更小簇合并形成一个极大簇的过程，我们用一个三叉树来表示，类似地，我们称这一结构为 **Rake Tree**，对应地 Rake Tree 里的点就是 **Rake Node**。每个 Rake Node 都代表一个簇，是由其左儿子和右儿子 Rake 到其中儿子代表的更小簇上形成的。具体可见下图，可知 Rake Tree 中的每个点都代表了 T 中具有相同端点的更小簇。

如图，蓝线圈出的是一个个极大簇，黄线圈出的是一个个更小簇。

对于那些更小簇，我们对它们进行相同处理，给它们选择簇路径、建出 Compress Tree ··· 如此递归下去，就建出了许多表示树收缩过程的 Compress Tree，Rake Tree。

如图 2-4，为原树的 Rake-Compress Tree（因为每个 Rake Node 都连着一棵 Compress Tree，所以表现为一棵 Rake Tree 连着许多 Compress Tree 的形态）和代表根簇路径的 Compress Tree。

考虑将这些树以某种方式拼接在一起，使它们形成一个有序的整体。记一个 Rake Tree 代表的最小簇的集合的公共端点是点 x。我们给这些 rake node 的中儿子（一个 compress tree 集合）都加入非 x 的另一端点，但仍保持其中序遍历和 top tree 的基本性质，如图。

这一步相当于是让 Rake 操作加入某个 T 中点的操作直接发生在 Compress Tree 中，这不仅使我们能正确维护 Rake Node 的信息（只需将三个儿子信息合并即可），还使我们 Compress Tree 的结构更完整。下一步，我们将 Compress Tree 改为三叉树，若某个 Rake Tree 的公共端点是点 x，我们就将 Rake Tree 挂在 $compress(x)$ 的中儿子处，如图。

此时经过三叉化的 $compress(x)$ 点，它的意义就变成先将其中儿子 Rake 到簇路径上，再统计左右儿子和点 x 的信息。

最后，我们再处理一下根簇路径的那棵 Compress Tree：与其它所有 Compress Tree 一致地，按中序遍历加入它

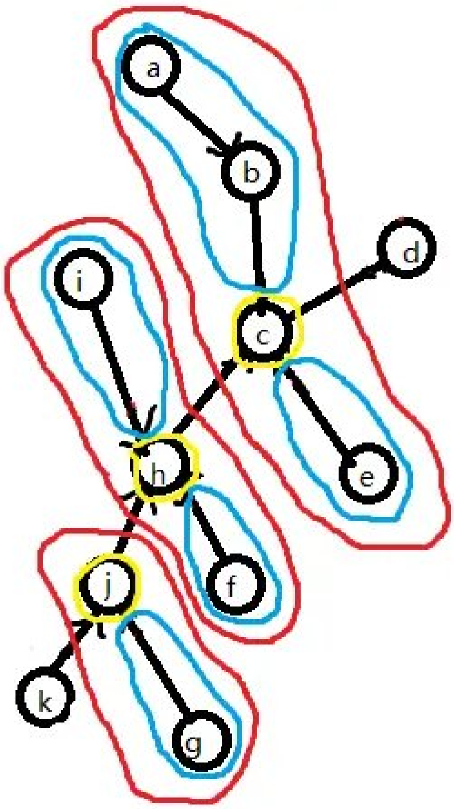


图 10.192

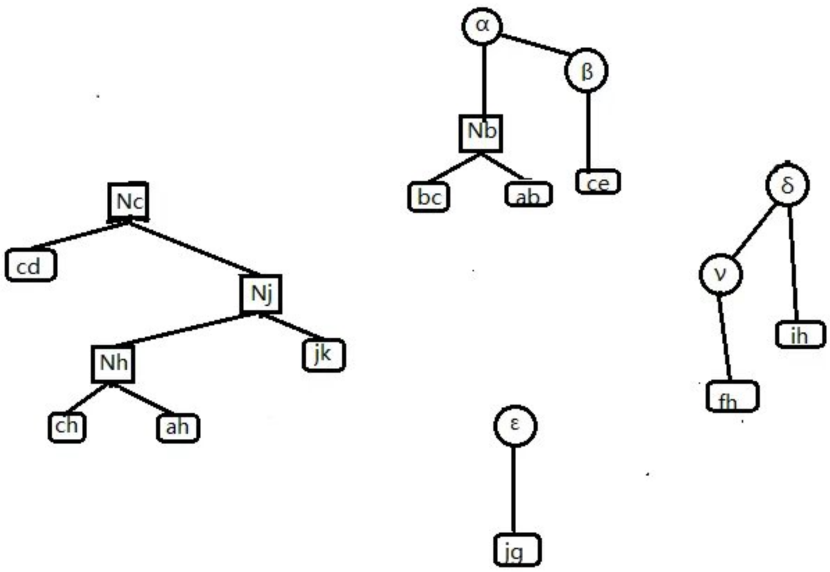


图 10.193

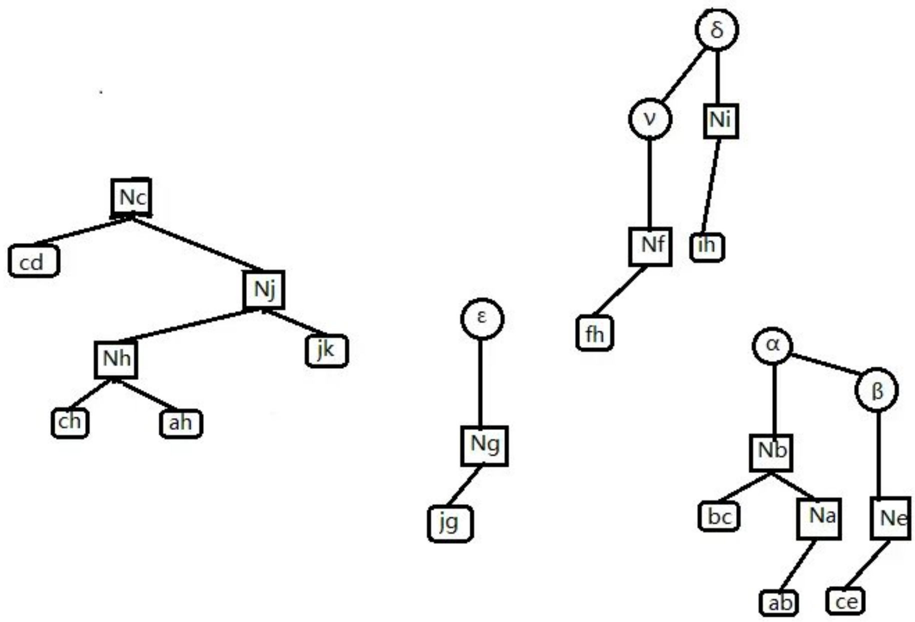


图 10.194

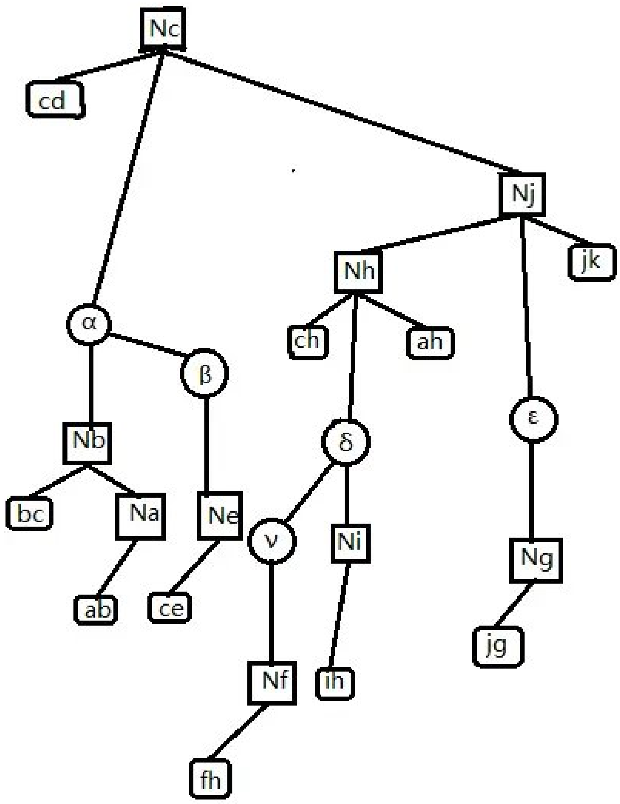


图 10.195

的两个端点，使得它的根储存整棵 T 的信息。

于是我们就实现了用三度化 Self-Adjusting Top Tree 实现一棵树的信息维护。

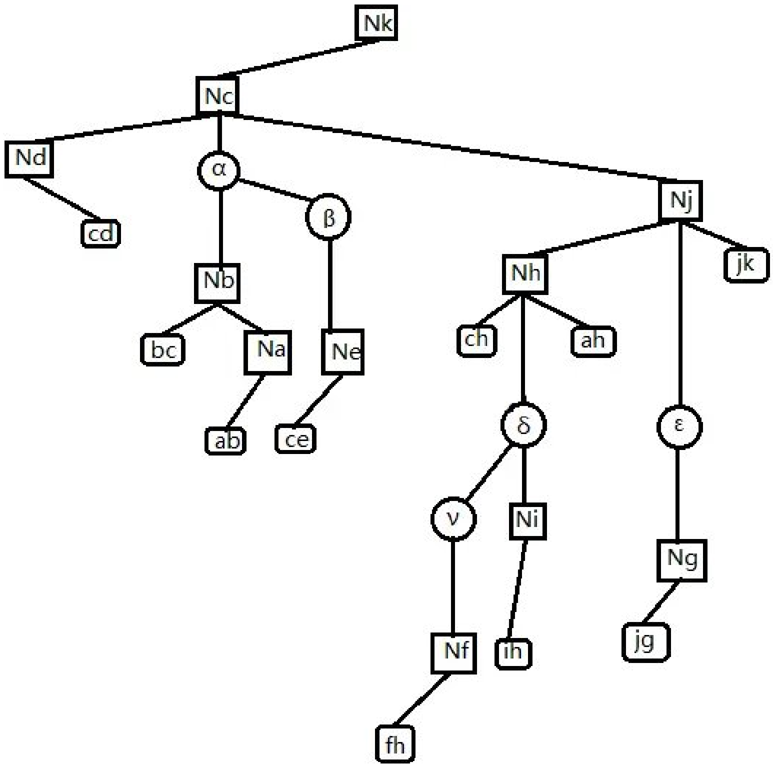


图 10.196

总结一下，SATT 有以下性质：

1. SATT 由 Compress Tree 和 Rake Tree 组成，Compress Tree 是一棵特殊的 Top Tree；Rake Tree 是一个三叉树，它们都对应一棵树进行树收缩的过程。
2. Compress Tree 里的点最多有三个儿子。Compress Tree 可以做类似于 Splay 树的旋转操作（只需保证其中序遍历不变即可，旋转一个点时保持其中儿子不动）。
3. Rake Tree 里的点一定有一个中儿子。Rake Tree 可以做类似于 Splay 树的旋转操作（只需保证其中序遍历不变即可，旋转一个点时保持其中儿子不动）。
4. SATT 的拓扑序反映了原树 T 的树收缩顺序。

我们在上文中提到的「修改某个点/某条路径在树收缩过程中信息被加入簇中的先后顺序」SATT 是否能实现呢，答案是肯定的。

在 SATT 中，有一个 `access(x)` 的操作，它的作用是使某点 x 成为根簇的非根端点，同时在 SATT 中使 $compress(x)$ 成为 SATT 的根。

我们可以通过 `access(x)` 操作以均摊 $O(\log n)$ 的复杂度使 SATT 中代表 $compress(x)$ 的点旋到整棵 SATT 的树根，根据 SATT 的第四个性质，我们改变了 $compress(x)$ 的操作顺序，使得它最晚执行，x 点的信息也就被最晚加入；这样当我们要修改 x 点的信息时，就只需要更新 $compress(x)$。

代码实现

Push 类函数

Pushup(x)

在考虑对 SATT 的某个节点维护信息时，首先分这个点在 Compress Tree 还是在 Rake Tree 进行讨论，原因可

见上文，不再赘述，下面以维护某个点的子树大小为例

```
// ls(x) x 的左儿子
// rs(x) x 的右儿子
// ms(x) x 的中儿子
// type==0 是 compress node
// type==1 是 rake node
void pushup(int x, int type) {
  if (type == 0)
    size[x] = size[rs(x)] + size[ms(x)] + 1;
  else
    size[x] = size[rs(x)] + size[ms(x)] + size[ls(x)];
  return;
}
```

查询点 x 的子树大小，就将其 Access 到 SATT 根，答案是其中儿子的 size $+1$；因为根据上文，在 Access 之后，其中儿子才是它的真子树。

`Pushdown(x)`

我们如果要对原树中的某个子树做整体修改，一个很自然的想法是：将这个节点直接 Access 到 SATT 根节点，给它的中儿子打上一个标记即可。同理，查询子树就直接 Access 后查询中儿子。

我们如果要对原树中的某条路径做整体修改，我们就 $expose$ 路径的两个端点，其中 $expose(x, y)$ 是使点 x 成为了 T 的根节点，使点 y 成为根簇的另一个端点。对应在 SATT 上，此时根簇的 compress tree 就是 x 到 y 的路径。于是直接给根簇的 Compress Tree 打上一个标记即可。同理查询链 $expose$ 后查询根节点即可。

于是我们就知道问题引入的问题怎么做了。

```
void pushdown(int x, int type) {
  if (type == 0) {
    // 处理链
    chain[ls(x)] += chain[x] chain[rs(x)] += chain[x];
    val[ls(x)] += chain[x];
    val[rs(x)] += chain[x];
    // 处理子树
    subtree[ls(x)] += subtree[x];
    subtree[rs(x)] += subtree[x];
    subtree[ms(x)] += subtree[x];
    val[ls(x)] += subtree[x];
    val[rs(x)] += subtree[x];
    val[ms(x)] += subtree[x];
    subtree[x] = 0;
  } else {
    subtree[ls(x)] += subtree[x];
    subtree[rs(x)] += subtree[x];
    subtree[ms(x)] += subtree[x];
    val[ls(x)] += subtree[x];
    val[rs(x)] += subtree[x];
    val[ms(x)] += subtree[x];
    subtree[x] = 0;
  }
  return;
}

// 下传标记
void pushall(int x, int type) {
```

```
  if (!isroot(x)) pushall(father[x], type);
  pushdown(x, type);
  return;
}
```

Splay 类函数

我们知道 SATT 中的 Rake Tree 和 Compress Tree 都是可以旋转的，也就是说它们可以用 Splay 来维护。因此我们可以写出以下代码：

```
// ls 一个 SATT 节点的左儿子
// rs 一个 SATT 节点的右儿子
// ms 一个 SATT 节点的中儿子
// type==1 在 rake tree 中
// type==0 在 compress tree 中
bool isroot(int x) {
  return rs(father[x]) != x && ls(father[x]) != x;
}  // 是一个节点的中儿子或无父亲

bool direction(int x) { return rs(father[x]) == x; }

void rotate(int x, int type) {
  int y = father[x], z = father[y], d = direction(x), w = son[x][d ^ 1];
  if (z) son[z][ms(z) == y ? 2 : direction(y)] = x;
  son[x][d ^ 1] = y;
  son[y][d] = w;
  if (w) father[w] = y;
  father[y] = x;
  father[x] = z;
  pushup(y, type);
  pushup(x, type);
  return;
}

void splay(int x, int type, int goal = 0) {
  pushall(x, ty); /* 下传标记 */
  for (int y; y = father[x], (!isroot(x)) && y != goal; rotate(x, ty)) {
    if (father[y] != goal && (!isroot(y))) {
      rotate(direction(x) ^ diretion(y) ? x : y, type);
    }
  }
  return;
}
```

值得注意的是，函数 `direction` 和 `isroot` 与普通 Splay 的不同。

因为无论你把这个点怎么转，这个点的中儿子是不会变的。

Access 类函数

`access(x)` 的意义是：将点 x 旋转到整个 SATT 的根处，使点 x 成为根簇的两个端点之一（另一端点即为 T 的根节点），同时不能改变原树的结构和原树的根。

为了实现 `access(x)`，我们先将其旋转到其所在 Compress Tree 的树根，再把点 x 的右儿子去掉，使点 x 成为其所在 compress tree 对应簇的端点。

```
if (rs(x)) {
  int y = new_node();
  setfather(ms(x), y, 0);
  setfather(rs(x), y, 2);
  rs(x) = 0;
  setfather(y, x, 2);
  pushup(y, 1);
  pushup(x, 0);
}
```

如果这时点 x 已经到了根部，则退出；若没有，则执行以下步骤，以让它跨过它上面的 Rake Tree：

1. 将其父亲节点（一定是一个 Rake Node），splay 到其 Rake Tree 的树根；
2. 将 x 的爷节点（一定是一个 Compress Node）splay 到其 Compress Tree 根部。
3. 若 x 的爷节点有一个右儿子，则将点 x 和爷节点的右儿子互换，更新信息，然后退出。
4. 若爷节点没有右儿子，则先让点 x 成为爷节点的右儿子，此时点 x 原来的父节点没有中儿子，根据上文 Rake Node 的性质，它不能存在。于是调用 `Delete` 函数，将其删除，然后退出。

1，2 两个步骤合称为 **Local Splay**。3，4 两个步骤合称为 **Splice**。但我们方便起见，将它们都写在 `Splice(x)` 函数里。

上文提到的 `Delete(x)` 函数是这样的：

1. 检视将要删除的点 x 有没有左儿子，若有，则将左儿子的子树后继续旋转到点 x 下方（成为新的左儿子），然后将右儿子（若有）变成左儿子的右儿子，此时点 x 的左儿子就代替了点 x。（这相当于 Splay 的合并操作）
2. 若没有左儿子，则直接让其右儿子代替点 x。

不难发现，Splice(x) 改变了原树的一些簇的端点选取。一次 splice 完了之后，我们将点 x 的父亲节点当作新的点 x，进行下一次 splice。

最终我们会发现我们最开始要操作的点 x 一定在根簇的 compress tree 最右端。我们只需最后做一次 **Global Splay**，将其旋至 SATT 根部即可。

```
// ls 一个 SATT 节点的左儿子
// rs 一个 SATT 节点的右儿子
// ms 一个 SATT 节点的中儿子
// son[x][0] ls
// son[x][1] rs
// son[x][2] ms
// type==1 在 rake tree 中
// type==0 在 compress tree 中
int new_node() {
  if (top) {
    top--;
    return Stack[top + 1];
  } else
    return ++tot;
}

void setfather(int x, int fa, int type) {
  if (x) father[x] = fa;
  son[fa][type] = x;
  return;
}
```

```
void Delete(int x) {
  setfather(ms(x), father[x], 1);
  if (ls(x)) {
    int p = ls(x);
    pushdown(p, 1);
    while (rs(p)) p = rs(p), pushdown(p, 1);
    splay(p, 1, x);
    setfather(rs(x), p, 1);
    setfather(p, father[x], 2);
    pushup(p, 1);
    pushup(father[x], 0);
  } else
    setfather(rs(x), father[x], 2);
  Clear(x);
}

void splice(int x) {
  /*  local splay */
  splay(x, 1);
  int y = father[x];
  splay(y, 0);
  pushdown(x, 1);
  /*  splice  */
  if (rs(y)) {
    swap(father[ms(x)], father[rs(y)]);
    swap(ms(x), rs(y));
  } else
    Delete(x);
  pushup(x, 1);
  pushup(y, 0);
  return;
}

void access(int x) {
  splay(x, 0);
  if (rs(x)) {
    int y = new_node();
    setfather(ms(x), y, 0);
    setfather(rs(x), y, 2);
    rs(x) = 0;
    setfather(y, x, 2);
    pushup(y, 1);
    pushup(x, 0);
  }
  while (father[x]) {
    splice(father[x]);
    x = father[x];
    pushup(x, 0);
  }
  splay(x, 0) /*global splay*/
      return;
}
```

关于 `makeroot(x)`:

若要让一个点成为原树的根，那么我们就将点 x Access 到 SATT 的根节点，可知此时点 x 已经是最终状态的簇

一个端点。由 Compress Tree 的中序遍历性质可知，将点 x 所在的 Compress Tree 左右颠倒（所有点的左右儿子互换），就使点 x 成为原树的根。在具体实现中，我们通过给点 x 打上一个翻转标记，之后下传来进行这一过程。

```
void makeroot(int x) {
  access(x);
  push_rev(x);
  return;
}
```

于是 expose(x,y) 就呼之欲出：

```
void expose(int x,int y){
    makeroot(x);
        access(y);
        return;
}
```

Link & Cut

现在我们要将原树中两个不连通的点之间连一条边，则我们先将其中的一个点 x $makeroot$，再将另一个点 y $access$ 到根，可知此时应该使点 y 成为点 x 的右儿子，并在点 y 的右儿子上挂上这一条边（在只需维护点的 SATT 中，这一步可省）。

```
void Link(int x, int y, int z) {
  /*z 代表连接 x,y 的边 */
  access(x);
  makeroot(y);
  setfather(y, x, 1);
  setfather(z, y, 0);
  pushup(x, 0);
  pushup(y, 0);
  return;
}
```

Cut 跟 link 原理差不多⋯⋯

```
void cut(int x, int y) {
  expose(x, y);
  clear(rs(x)); /* 删掉 xy 这一基簇。*/
  father[x] = ls(y) = rs(x);
  pushup(y, 0);
}
```

完整代码（Luogu P3690【模板】动态树）

" 示例代码"

```
#include <bits/stdc++.h>
#define ls(x) T[x][0]
#define rs(x) T[x][1]
#define ms(x) T[x][2]
#define maxn 1000005
using namespace std;

int read() {
```

```
  int s = 0;
  char a = getchar();
  while (!isdigit(a)) a = getchar();
  while (isdigit(a)) {
    s = (s << 1) + (s << 3);
    s += a ^ 48;
    a = getchar();
  }
  return s;
}

int T[maxn][3], s[maxn][2], tot, v[maxn], n, m, r[maxn], top, st[maxn], f[maxn];


int nnd() {
  if (top) {
    top--;
    return st[top + 1];
  } else
    return ++tot;
}

bool isr(int x) { return rs(f[x]) != x && ls(f[x]) != x; }

bool dir(int x) { return rs(f[x]) == x; }

void psu(int x, int ty) {
  if (ty) {
    s[x][1] = s[ls(x)][1] ^ s[rs(x)][1] ^ s[ms(x)][1];
    return;
  }
  s[x][0] = s[ls(x)][0] ^ v[x] ^ s[rs(x)][0];
  s[x][1] = s[ls(x)][1] ^ s[ms(x)][1] ^ s[rs(x)][1] ^ v[x];
}

void psr(int x) {
  if (!x) return;
  r[x] ^= 1;
  swap(ls(x), rs(x));
}

void psd(int x, int ty) {
  if (ty) return;
  if (r[x]) {
    psr(ls(x));
    psr(rs(x));
    r[x] = 0;
    return;
  }
}

void upd(int x, int ty) {
  if (!isr(x)) upd(f[x], ty);
  psd(x, ty);
```

```
}

void stf(int x, int fa, int ty) {
  if (x) f[x] = fa;
  T[fa][ty] = x;
  return;
}

void rtt(int x, int ty) {
  int y = f[x], z = f[y], d = dir(x), w = T[x][d ^ 1];
  if (z) T[z][ms(z) == y ? 2 : dir(y)] = x;
  T[x][d ^ 1] = y;
  T[y][d] = w;
  if (w) f[w] = y;
  f[y] = x;
  f[x] = z;
  psu(y, ty);
  psu(x, ty);
}

void spy(int x, int ty, int gl = 0) {
  upd(x, ty);
  for (int y; y = f[x], (!isr(x)) && y != gl; rtt(x, ty)) {
    if (f[y] != gl && (!isr(y))) rtt(dir(x) ^ dir(y) ? x : y, ty);
  }
}

void cle(int x) {
  ls(x) = ms(x) = rs(x) = s[x][0] = s[x][1] = r[x] = v[x] = 0;
  st[++top] = x;
}

void del(int x) {
  stf(ms(x), f[x], 1);
  if (ls(x)) {
    int p = ls(x);
    psd(p, 1);
    while (rs(p)) p = rs(p), psd(p, 1);
    spy(p, 1, x);
    stf(rs(x), p, 1);
    stf(p, f[x], 2);
    psu(p, 1);
    psu(f[x], 0);
  } else
    stf(rs(x), f[x], 2);
  cle(x);
}

void spl(int x) {
  spy(x, 1);
  int y = f[x];
  spy(y, 0);
  psd(x, 1);
  if (rs(y)) {
```

```
    swap(f[ms(x)], f[rs(y)]);
    swap(ms(x), rs(y));
    psu(x, 1);
  } else
    del(x);
  psu(rs(y), 0);
  psu(y, 0);
}

void acs(int x) {
  spy(x, 0);
  int ys = x;
  if (rs(x)) {
    int y = nnd();
    stf(ms(x), y, 0);
    stf(rs(x), y, 2);
    rs(x) = 0;
    stf(y, x, 2);
    psu(y, 1);
    psu(x, 0);
  }
  while (f[x]) {
    spl(f[x]);
    x = f[x];
  }
  spy(ys, 0);
}

int fdr(int x) {
  acs(x);
  psd(x, 0);
  while (ls(x)) x = ls(x), psd(x, 0);
  spy(x, 0);
  return x;
}

void mkr(int x) {
  acs(x);
  psr(x);
}

void epo(int x, int y) {
  mkr(x);
  acs(y);
}

void lnk(int x, int y) {
  if (fdr(x) == fdr(y)) return;
  acs(x);
  mkr(y);
  stf(y, x, 1);
  psu(x, 0);
  psu(y, 0);
}
```

```
void cu(int x, int y) {
  epo(x, y);
  if (ls(y) != x || rs(x)) return;
  f[x] = ls(y) = 0;
  psu(y, 0);
}

int main() {
  int i, j, op, U, V, n = read(), m = read();
  tot = n;
  for (i = 1; i <= n; i++) v[i] = read(), psu(i, 0);
  for (i = 1; i <= m; i++) {
    op = read();
    U = read();
    V = read();
    if (op == 0) {
      epo(U, V);
      cout << s[V][0] << '\n';
    }
    if (op == 1) lnk(U, V);
    if (op == 2) cu(U, V);
    if (op == 3) {
      acs(U);
      v[U] = V;
      psu(U, 0);
    }
  }
  return 0;
}
```

SATT 的时间复杂度证明

设在一棵 SATT（点数为 n）中，其当前状态 x 的势能函数为

$\varphi(x) = \sum_{i=1}^{n} r(i)$

其中 $r(i) = \lceil \log_2(\text{以}i\text{ 为根的子树大小}) \rceil$。

则 SATT 的 splay 的均摊复杂度显然仍是 $3n\log n + 1$，即使 SATT 是一个三叉树。

因此对于 SATT，我们只要证得 Access 函数复杂度正确，就能证得 SATT 的时间复杂度。

我们逐步分析 Accese 的均摊复杂度。

我们先要将点 x 旋至其所在 Compress Tree 的根，则这一步的均摊复杂度

$a \leq 3\log n + 1$

接着我们要使点 x 无右儿子，则这一步的均摊复杂度

$a = 1 + r'(\gamma) - 0 \leq \log n + 1$

如图，为去掉点 x 的右儿子过程。

然后是 Local Splay，Splice 交替进行的过程，经过若干次 Splice，点 x 被旋至 SATT 的根。我们对其中一组 Local Splay，Splice 进行分析：

如图，体现了对点 x 做一次 Splice 的过程，不包括最后左旋点 x 的部分。

为表达方便，设 $r_x(i)$ 为点 i 在状态 x 时的 r 值。

由图，易知由状态 1 到状态 2 的操作（将点 x 的父亲旋至其 Rake Tree 的根部的 Local Splay 操作）的均摊复杂度

$a \leq 3(r_2(\gamma) - r_1(\gamma)) + 1$

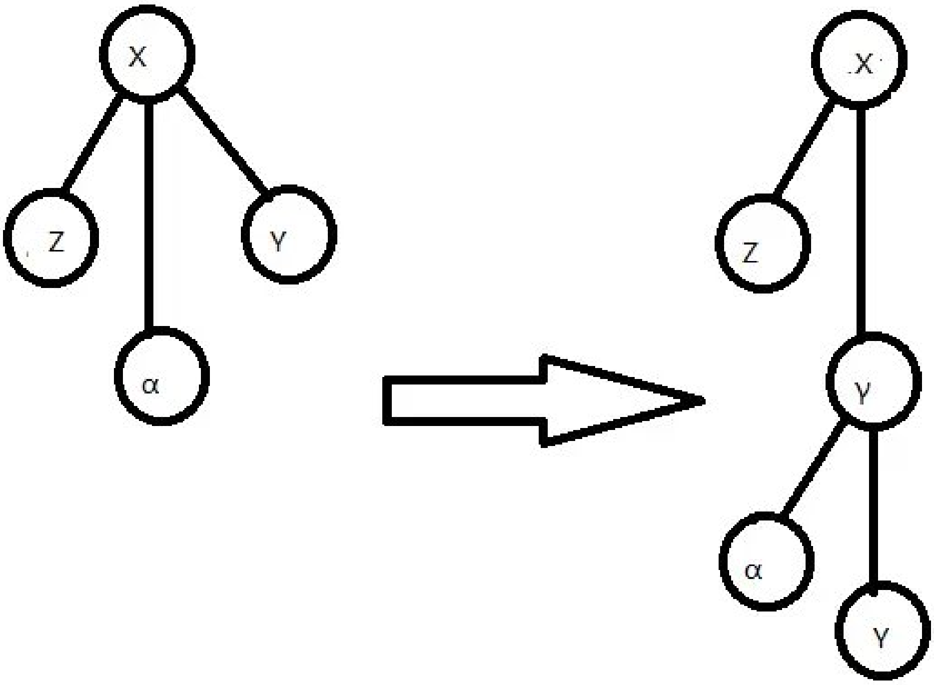


图 10.197

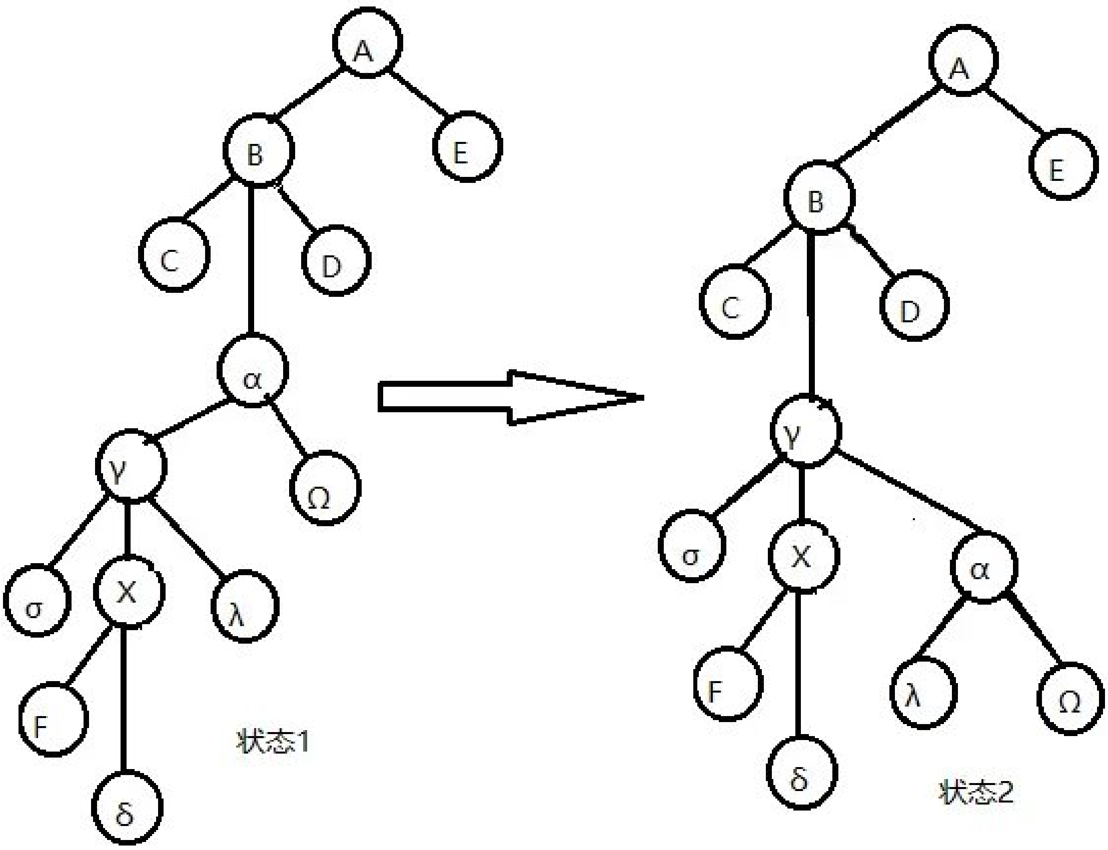


图 10.198

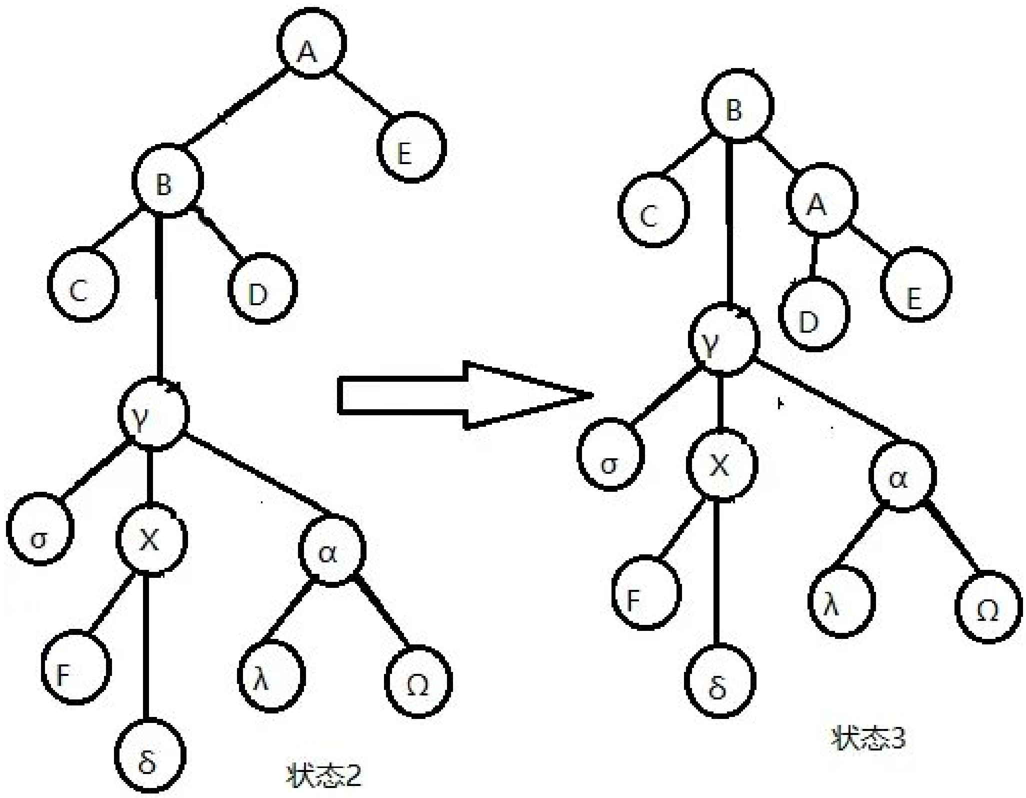


图 10.199

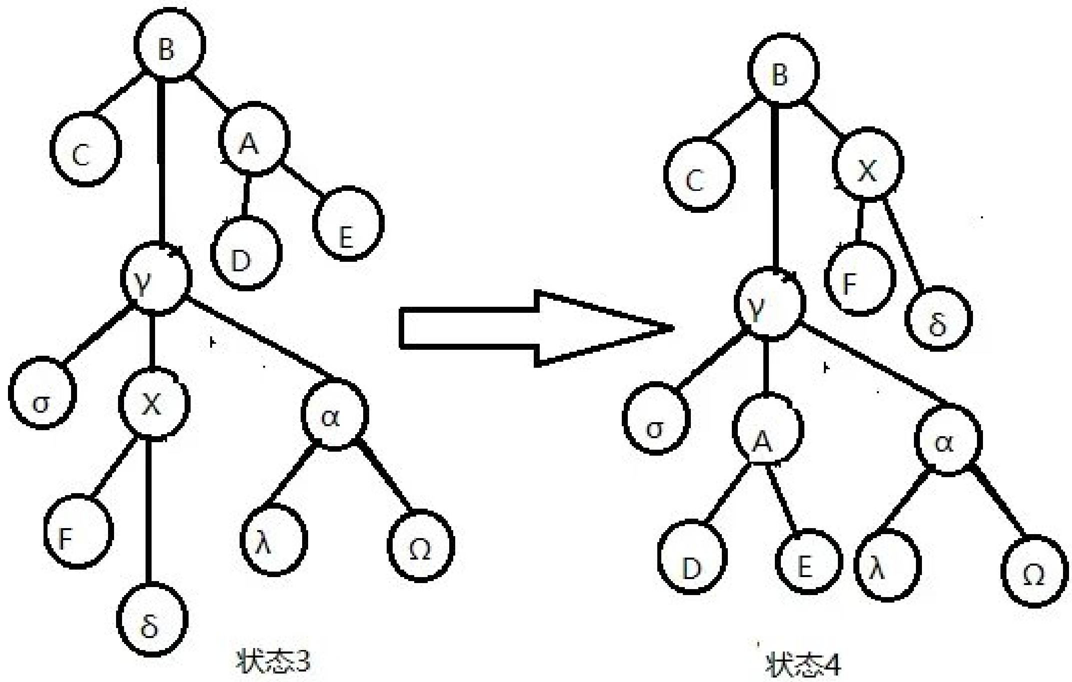


图 10.200

由图，易知由状态 2 到状态 3 的操作（将点 x 的爷节点旋至其 Compress Tree 的根部的 Local Splay 操作）的均摊复杂度

$a \le 3(r_3(B) - r_2(B)) + 1$

重点分析由状态 3 到状态 4 的操作（Splice ）

$a = r_4(\gamma) - r_3(\gamma) + 1$

不难发现 $r_4(\gamma) \le r_3(B)$

故这一次操作的均摊复杂度为

$$\le 3(r_3(B) - r_3(\gamma)) + 1$$

综合上述过程，一次 Splice 的复杂度为

$a \le 3r_3(B) + 3r_3(B) + 3r_2(\gamma) - 3r_3(\gamma) - 3r_2(B) - 3r_1(\gamma) + 3$

记下一次 Splice 的点 X（即状态 4 中的点 B）的 r 值为 $r'(X)$，并注意到

$r_3(\gamma), r_1(\gamma) \ge r_1(X)$

$r_3(B), r_2(\gamma) \le r'(X)$ 且$r_3(B) = r_2(B)$

所以

$a \le 9(r'(X) - r(X)) + 3$

除了上面这个复杂度以外，在 Splice 中可能还会有因 `delete(x)` 产生的额外均摊复杂度，记这一部分为 $a' \le 3\log n + 1$。

先不管 a' 部分，每次 Splice 的 $r'(X)$ 等于下一次的 $r(X)$，且第一次 Splice 的 $r(X)$ 等于我们一开始旋转点 x 到其 Compress Tree 树根时的 $r(X)$，则对于不计'delete(x)' 的一次'access(x)' 复杂度，我们有：

$a \le 9(r'(x) - r(x)) + 3k + 1$

其中 k 为 Splice 次数。

看样子 a 会带一个 $3k + 1$ 导致均摊复杂度无法分析，但我们有办法来对付它，注意到 zig-zig\zig-zag 的旋转可以这么均摊

$$\le 3(q - 1)(r'(X) - r(X))$$

如果我们能找到足够多的 zig-zig，zig-zag 操作，我们就可以将这 $3k + 1$ 平摊到这些操作上去，从而消掉这个 $3k + 1$。

我们发现 Globel Splay 里面就有这么多的 zig-zig，zag-zig 来给我们使用，因为 Globel Splay 里面点的个数一定大于 k，而从点 x 到 Globel Splay 根部路径的点数一定不少于 k，也就是说一次 `access(x)` 中一定会至少有 $\frac{k}{2}$ 个 zig-zag 操作，算上 Globel Splay 的均摊复杂度 $a \le 3\log n + 1$，一次 `access(x)` 不记 `delete(x)` 的均摊复杂度为

$a \le 9(r'(X) - r(X)) + 3k + 1 + 18(r''(X) - r'(X)) - S + 1 + 3\log n + 1, S \ge 3k$

$a \le 18(r''(X) - r(X)) + 2 + 3\log n + 1$

$a \le 21(r''(X) - r(X)) + 3$

现在算上 a'，列出进行 m 次 `access(x)` 操作的总式子。

$\sum_{i=1}^{m} a'_i + \sum_{i=1}^{m} a_i = \sum_{i=1}^{m} c_i + \varphi(x_n) - \varphi(x_0)$

我们要求的是实际复杂度

$\sum_{i=1}^{m} c_i = \sum_{i=1}^{m} a_i + \sum_{i=1}^{m} a'_i - \varphi(x_n) + \varphi(x_0)$

$\sum_{i=1}^{m} c_i \le \sum_{i=1}^{m} a'_i + 21m\log n + n\log n + 3m$

注意到 `delete(x)` 操作的本质是删掉一个 Rake Node，但我们在 m 次操作中最多只会添加 m 个 Rake Node，由 Rake Node 的定义，我们初始时最多有 n 个 Rake Node，也就是说我们总共只会做 $m + n$ 次 `delete(x)` 操作，由 $a' \le 3\log n + 1$ 可知

$\sum_{i=1}^{m} c_i \le 3(m + n)\log n + 21m\log n + n\log n + 4m + n$

所以我们就证明了 Access 的复杂度，而其他函数要么基于 Access 要么单次时间复杂度为常数，所以我们就证明了 SATT 的复杂度。

顺便一提，如果像 LCT 一样省略 Global Splay 的过程，改为在每次 Splice 时直接将要 Access 的点旋转一下，这样做时间复杂度也是对的（实测省略 Global Splay 的版本要快很多，能与 LCT 在 Luogu P3690 跑得不分上下）。

例题

CF1192B Dynamic Diameter

维护动态直径，建出 SATT 后，我们只需要在 `Pushup(x)` 里面维护每个点的答案，最后查询根节点的答案（即整棵树的直径）就可以了。

```
void pushup(int x, int op) {
  if (op == 0) {
    /* 是 compress node*/
    len[x] = len[ls(x)] + len[rs(x)];
    diam[x] = maxs[ls(x)][1] + maxs[rs(x)][0];
    diam[x] =
        max(diam[x], max(maxs[ls(x)][1], maxs[rs(x)][0]) + maxs[ms(x)][0]);
    diam[x] = max(diam[x], max(max(diam[ls(x)], diam[rs(x)]), diam[ms(x)]));
    maxs[x][0] =
        max(maxs[ls(x)][0], len[ls(x)] + max(maxs[ms(x)][0], maxs[rs(x)][0]));
    maxs[x][1] =
        max(maxs[rs(x)][1], len[rs(x)] + max(maxs[ms(x)][0], maxs[ls(x)][1]));
  } else {
    /* 是 rake node*/
    diam[x] = maxs[ls(x)][0] + maxs[rs(x)][0];
    diam[x] =
        max(diam[x], maxs[ms(x)][0] + max(maxs[ls(x)][0], maxs[rs(x)][0]));
    diam[x] = max(max(diam[x], diam[ms(x)]), max(diam[ls(x)], diam[rs(x)]));
    maxs[x][0] = max(maxs[ms(x)][0], max(maxs[ls(x)][0], maxs[rs(x)][0]));
  }
  return;
}
```

其中 $diam$ 是当前点的答案（这个点代表的簇的直径）。len 表示当前 Compress Node 所在簇路径的长度，$maxs_{0/1}$ 表示 compress node 到簇内点和端点的不选簇路径儿子/不选父亲的最大距离（如果是 Rake Node 则只存储选取当前簇的上端点到簇内点和端点的最大距离 $maxs_0$）。每次查询 SATT 根节点的 diam 即可，正确性显然。

注意对 `Pushrev(x)` 做一些改动。

```
void pushrev(int x) {
  if (!x) return;
  r[x] ^= 1;
  swap(ls(x), rs(x));
  swap(maxs[x][0], maxs[x][1]);
  return;
}
```

[CSP-S2019] **树的重心**

假如我们能动态 $O(\log n)$ 维护树的重心，我们就做出这个题了。

SATT 支持动态 $O(\log n)$ 维护树的重心，做到这需要**非局部搜索（Non-local Search）**。

对于一种树上的性质，如果一个点/一条边在整棵树中有这种性质，且在所有包含它的子树中都包含此种性质，我们就称这个性质是**局部的（Local）**，否则称它是**非局部的（Non-local）**。局部信息一般可以通过 `pushup(x)` 来维护

例如，权值最小值是局部的，因为一个点/一条边如果在整棵树中权值最小，那么在所有包含它的子树中它也是权值最小的，而权值第二小显然就是非局部的。

我们上文维护的 $diam$ 也是局部信息。

回到正题，重心显然是一个非局部信息，无法通过简单的 `pushup(x)` 来维护。我们考虑在 SATT 上搜索：

我们的搜索从 SATT 的根节点，即根簇开始。注意到重心有很好的性质：假如有一条边的一侧点的个数大于等于另一侧点的个数，那么边的这一侧一定至少有一个重心（重心可能有 2 个）。

记 sum 表示某一个簇的点个数，$maxs$ 为一棵 Rake Tree 的所有 Rake Node 中儿子的 sum 最大值。

```cpp
void pushup(int x, int op) {
  if (op == 0) {
    /* 是 compress node*/
    sum[x] = sum[ls(x)] + sum[rs(x)] + sum[ms(x)] + 1;
  } else {
    /* 是 rake node*/
    maxs[x] = max(maxs[ls(x)], max(maxs[rs(x)], sum[ms(x)]));
    sum[x] = sum[ls(x)] + sum[rs(x)] + sum[ms(x)];
  }
  return;
}
```

现在我们在根簇

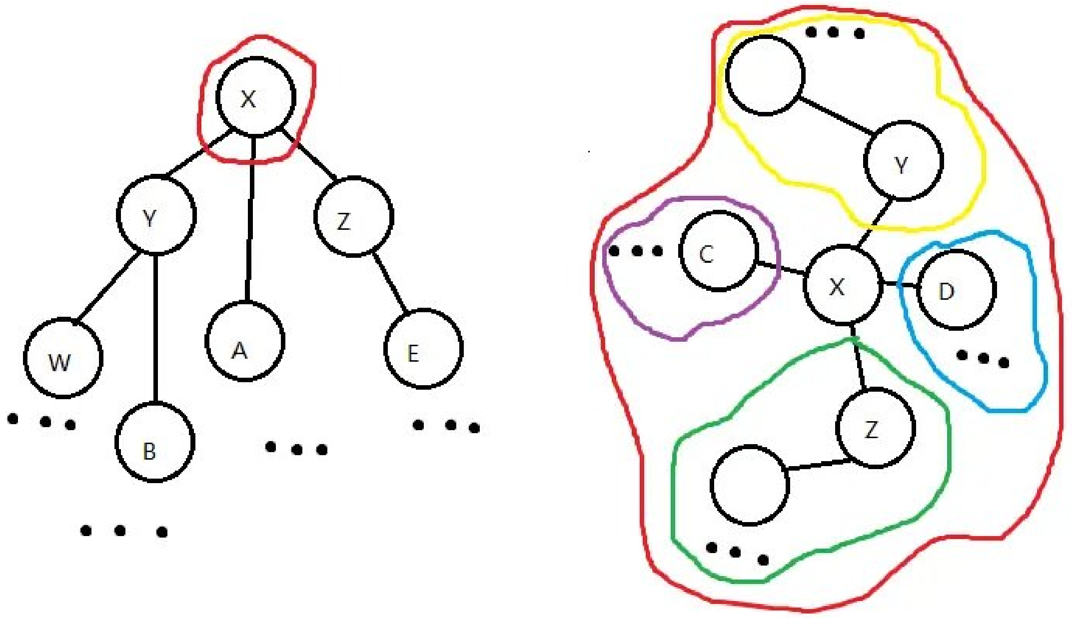


图 10.201

如图，为在进行 Non-local Search 时的 SATT 和对应的原树 T。

我们做如下比较：

1. 比较簇 $compress(Y)$ 的 sum 值与簇 $compress(Z)$、簇 A 和点 X 的并（我们暂称为簇 α）的 sum 值。若 $compress(Y)$ 的 sum 值大于等于后者，说明至少有一个重心在 $compress(Y)$ 的子树中，我们递归到 $compress(Y)$ 搜索。（如果此处取等，点 X 也是一个重心，需要记录）
2. 比较簇 $compress(Z)$ 的 sum 值与簇 $compress(Y)$、簇 A 和点 X 的并（我们暂称为簇 β）的 sum 值。若 $compress(Z)$ 的 sum 值大于等于后者，说明至少有一个重心在 $compress(Z)$ 的子树中，我们递归到 $compress(Z)$ 搜索。（如果此处取等，点 X 也是一个重心，需要记录）

3. 比较点 x 中儿子 Rake tree 之中 sum 最大的更小簇（见 3-2）的 sum 值与簇 $compress(Y)$、簇 A、点 X 及其它更小簇的并（我们暂称为簇 Y）的 sum 值，若那个更小簇的 sum 值大于等于后者，说明至少有一个重心在那个更小簇的子树中，我们递归到它搜索。(如果此处取等，点 X 也是一个重心，需要记录)
4. 若以上比较都不递归，则点 X 一定是一个重心，记录并退出。

第一步的搜索显然正确，之后应该怎么搜呢?

假如我们递归到 Y，则现在 Y 储存信息的并不完整，因为 $compress(Y)$ 里面只存储了它自己这个簇的信息，而我们要求的是整棵树的重心。解决方法是，将之前簇的信息记录下来，在点 Y 上比较计算时将上一个簇的信息与点 Y 自己的信息合并处理。具体实现如下：

```
void non_local_search(int x, int lv, int rv, int op) {
  /*lv 和 rv 都是搜索的上一个簇的信息 */

  if (!x) return;
  psd(x, 0);
  if (op == 0) {
    if (maxs[ms(x)] >=
        sum[ms(x)] - maxs[ms(x)] + sum[rs(x)] + sum[ls(x)] + lv + 1 + rv) {
      if (maxs[ms(x)] ==
          sum[ms(x)] - maxs[ms(x)] + sum[rs(x)] + sum[ls(x)] + lv + 1 + rv) {
        if (ans1)
          ans2 = x;
        else
          ans1 = x;
      }
      non_local_search(
          ms(x),
          sum[ms(x)] - maxs[ms(x)] + sum[rs(x)] + sum[ls(x)] + 1 + lv + rv, 0,
          1);
      return;
    }
    if (ss[rs(x)] + rv >= ss[ms(x)] + ss[ls(x)] + lv + 1) {
      if (ss[rs(x)] + rv == ss[ms(x)] + ss[ls(x)] + lv + 1) {
        if (ans1)
          ans2 = x;
        else
          ans1 = x;
      }
      non_local_search(rs(x), sum[ms(x)] + 1 + sum[ls(x)] + lv, rv, 0);
      return;
    }
    if (sum[ls(x)] + lv >= sum[ms(x)] + sum[rs(x)] + 1 + rv) {
      if (sum[ls(x)] + lv == sum[ms(x)] + sum[rs(x)] + 1 + rv) {
        if (ans1)
          ans2 = x;
        else
          ans1 = x;
      }
      non_local_search(ls(x), lv, rv + sum[ms(x)] + 1 + sum[rs(x)], 0);
      return;
    }
  } else {
    if (maxs[ls(x)] == maxs[x]) {
      non_local_search(ls(x), lv, rv, 1);
```

```
      return;
    }
    if (maxs[rs(x)] == maxs[x]) {
      non_local_search(rs(x), lv, rv, 1);
      return;
    }
    non_local_search(ms(x), lv, rv, 0);
    return;
  }
  if (ans1)
    ans2 = x;
  else
    ans1 = x;
  return;
}
```

完整代码如下：

" 示例代码"

```
#include <bits/stdc++.h>
#define ls(x) T[x][0]
#define rs(x) T[x][1]
#define ms(x) T[x][2]
#define maxn 1000005
using namespace std;

int read() {
  int s = 0;
  char a = getchar();
  while (!isdigit(a)) a = getchar();
  while (isdigit(a)) {
    s = (s << 1) + (s << 3);
    s += a ^ 48;
    a = getchar();
  }
  return s;
}

int T[maxn][3], tot, n, m, r[maxn], top, st[maxn], f[maxn], maxs[maxn],
    ss[maxn];

int nnd() {
  if (top) {
    top--;
    return st[top + 1];
  } else
    return ++tot;
}

bool isr(int x) { return rs(f[x]) != x && ls(f[x]) != x; }

bool dir(int x) { return rs(f[x]) == x; }

void psr(int x) {
```

```
  if (!x) return;
  r[x] ^= 1;
  swap(ls(x), rs(x));
}

void psd(int x, int ty) {
  if (ty) return;
  if (r[x]) {
    psr(ls(x));
    psr(rs(x));
    r[x] = 0;
    return;
  }
}

void psu(int x, int op) {
  psd(x, op); /* 不知道哪没 psd*/
  if (op == 0) {
    ss[x] = ss[ls(x)] + ss[rs(x)] + ss[ms(x)] + 1;
  } else {
    maxs[x] = max(maxs[ls(x)], max(maxs[rs(x)], ss[ms(x)]));
    ss[x] = ss[ls(x)] + ss[rs(x)] + ss[ms(x)];
  }
  return;
}

void upd(int x, int ty) {
  if (!isr(x)) upd(f[x], ty);
  psd(x, ty);
}

void stf(int x, int fa, int ty) {
  if (x) f[x] = fa;
  T[fa][ty] = x;
  return;
}

void rtt(int x, int ty) {
  int y = f[x], z = f[y], d = dir(x), w = T[x][d ^ 1];
  if (z) T[z][ms(z) == y ? 2 : dir(y)] = x;
  T[x][d ^ 1] = y;
  T[y][d] = w;
  if (w) f[w] = y;
  f[y] = x;
  f[x] = z;
  psu(y, ty);
  psu(x, ty);
}

void spy(int x, int ty, int gl = 0) {
  upd(x, ty);
  for (int y; y = f[x], (!isr(x)) && y != gl; rtt(x, ty)) {
    if (f[y] != gl && (!isr(y))) rtt(dir(x) ^ dir(y) ? x : y, ty);
  }
```

```
}

void cle(int x) {
  ls(x) = ms(x) = rs(x) = ss[x] = r[x] = maxs[x] = f[x] = 0;
  st[++top] = x;
}

void del(int x) {
  stf(ms(x), f[x], 1);
  if (ls(x)) {
    int p = ls(x);
    psd(p, 1);
    while (rs(p)) p = rs(p), psd(p, 1);
    spy(p, 1, x);
    stf(rs(x), p, 1);
    stf(p, f[x], 2);
    psu(p, 1);
    psu(f[x], 0);
  } else
    stf(rs(x), f[x], 2);
  cle(x);
}

void spl(int x) {
  spy(x, 1);
  int y = f[x];
  spy(y, 0);
  psd(x, 1);
  if (rs(y)) {
    swap(f[ms(x)], f[rs(y)]);
    swap(ms(x), rs(y));
  } else
    del(x);
  psu(x, 1);
  psu(y, 0);
  rtt(rs(y), 0);
}

void acs(int x) {
  spy(x, 0);
  if (rs(x)) {
    int y = nnd();
    stf(ms(x), y, 0);
    stf(rs(x), y, 2);
    rs(x) = 0;
    stf(y, x, 2);
    psu(y, 1);
    psu(x, 0);
  }
  while (f[x]) spl(f[x]);
}

void mkr(int x) {
  acs(x);
```

```
  psr(x);
}

void epo(int x, int y) {
  mkr(x);
  acs(y);
}

void lnk(int x, int y) {
  acs(x);
  mkr(y);
  stf(y, x, 1);
  psu(x, 0);
}

void cu(int x, int y) {
  epo(x, y);
  f[x] = ls(y) = 0;
  psu(y, 0);
}

int ans1, ans2;

void non_local_search(int x, int lv, int rv, int op) {
  if (!x) return;
  psd(x, 0);
  if (op == 0) {
    if (maxs[ms(x)] >=
        ss[ms(x)] - maxs[ms(x)] + ss[rs(x)] + ss[ls(x)] + lv + 1 + rv) {
      if (maxs[ms(x)] ==
          ss[ms(x)] - maxs[ms(x)] + ss[rs(x)] + ss[ls(x)] + lv + 1 + rv) {
        if (ans1)
          ans2 = x;
        else
          ans1 = x;
      }
      non_local_search(
          ms(x), ss[ms(x)] - maxs[ms(x)] + ss[rs(x)] + ss[ls(x)] + 1 + lv + rv,
          0, 1);
      return;
    }
    if (ss[rs(x)] + rv >= ss[ms(x)] + ss[ls(x)] + lv + 1) {
      if (ss[rs(x)] + rv == ss[ms(x)] + ss[ls(x)] + lv + 1) {
        if (ans1)
          ans2 = x;
        else
          ans1 = x;
      }
      non_local_search(rs(x), ss[ms(x)] + 1 + ss[ls(x)] + lv, rv, 0);
      return;
    }
    if (ss[ls(x)] + lv >= ss[ms(x)] + ss[rs(x)] + 1 + rv) {
      if (ss[ls(x)] + lv == ss[ms(x)] + ss[rs(x)] + 1 + rv) {
        if (ans1)
```

```
          ans2 = x;
        else
          ans1 = x;
      }
      non_local_search(ls(x), lv, rv + ss[ms(x)] + 1 + ss[rs(x)], 0);
      return;
    }
  } else {
    if (maxs[ls(x)] == maxs[x]) {
      non_local_search(ls(x), lv, rv, 1);
      return;
    }
    if (maxs[rs(x)] == maxs[x]) {
      non_local_search(rs(x), lv, rv, 1);
      return;
    }
    non_local_search(ms(x), lv, rv, 0);
    return;
  }
  if (ans1)
    ans2 = x;
  else
    ans1 = x;
  return;
}

int qu[maxn], qv[maxn];

int main() {
  int i, TT = read(), n, I, U, V, x;
  long long ANS;
  for (I = 1; I <= TT; I++) {
    n = read();
    tot = n;
    ANS = 0;
    for (i = 1; i <= n; i++) ss[i] = 1;
    for (i = 1; i <= n - 1; i++) {
      qu[i] = U = read();
      qv[i] = V = read();
      lnk(U, V);
    }
    for (i = 1; i <= n - 1; i++) {
      cu(qu[i], qv[i]);
      ans1 = 0;
      ans2 = 0;
      non_local_search(qu[i], 0, 0, 0);
      ANS += ans1 + ans2;
      if (ans1) acs(ans1);
      if (ans2) acs(ans2);
      ans1 = 0;
      ans2 = 0;
      non_local_search(qv[i], 0, 0, 0);
      ANS += ans1 + ans2;
      if (ans1) acs(ans1);
```

```
      if (ans2) acs(ans2);
      lnk(qu[i], qv[i]);
    }
    cout << ANS << '\n';
    for (i = 1; i <= tot; i++)
      T[i][0] = T[i][1] = T[i][2] = ss[i] = r[i] = maxs[i] = f[i] = 0;
    tot = top = 0;
  }
  return 0;
}
```

Referrence

1. 《Self-Adjusting Top Trees Robert》E. Tarjan，Renato F. Werneck。

2.negiizhao 的博客[1]

3.zhengrunzhe 的 sone1 题解[2]

参考资料与注释

[1] negiizhao 的博客

[2] zhengrunzhe 的 sone1 题解

10.23 析合树

解释一下本文可能用到的符号：$\wedge$ 逻辑与，$\vee$ 逻辑或。

关于段的问题

我们由一个小清新的问题引入：

> 对于一个 $1-n$ 的排列，我们称一个值域连续的区间为段。问一个排列的段的个数。比如，$\{5,3,4,1,2\}$ 的段有：$[1,1],[2,2],[3,3],[4,4],[5,5],[2,3],[4,5],[1,3],[2,5],[1,5]$。

看到这个东西，感觉要维护区间的值域集合，复杂度好像挺不友好的。线段树可以查询某个区间是否为段，但不太能统计段的个数。

这里我们引入这个神奇的数据结构——析合树！

连续段

在介绍析合树之前，我们先做一些前提条件的限定。鉴于 LCA 的课件中给出的定义不易理解，为方便读者理解，这里给出一些不太严谨（但更容易理解）的定义。

排列与连续段

排列：定义一个 n 阶排列 P 是一个大小为 n 的序列，使得 P_i 取遍 $1,2,\cdots,n$。说得形式化一点，n 阶排列 P 是一个有序集合满足：

1. $|P|=n$.

2. $\forall i, P_i \in [1, n]$.
3. $\nexists i, j \in [1, n], P_i = P_j$.

连续段：对于排列 P，定义连续段 $(P, [l, r])$ 表示一个区间 $[l, r]$，要求 $P_{l\sim r}$ 值域是连续的。说得更形式化一点，对于排列 P，连续段表示一个区间 $[l, r]$ 满足：

$$(\nexists\, x, z \in [l, r], y \notin [l, r],\ P_x < P_y < P_z)$$

特别地，当 $l > r$ 时，我们认为这是一个空的连续段，记作 $(P, \emptyset)$。

我们称排列 P 的所有连续段的集合为 I_P，并且我们认为 $(P, \emptyset) \in I_P$。

连续段的运算

连续段是依赖区间和值域定义的，于是我们可以定义连续段的交并差的运算。

定义 $A = (P, [a, b]), B = (P, [x, y])$，且 $A, B \in I_P$。于是连续段的关系和运算可以表示为：

1. $A \subseteq B \iff x \le a \wedge b \le y$.
2. $A = B \iff a = x \wedge b = y$.
3. $A \cap B = (P, [\max(a, x), \min(b, y)])$.
4. $A \cup B = (P, [\min(a, x), \max(b, y)])$.
5. $A \ \ B = (P, \{i | i \in [a, b] \wedge i \notin [x, y]\})$.

其实这些运算就是普通的集合交并差放在区间上而已。

连续段的性质

连续段的一些显而易见的性质。我们定义 $A, B \in I_P, A \cap B \ne \emptyset, A \notin B, B \notin A$，那么有 $A \cup B, A \cap B, A \ \ B, B \ \ A \in I_P$。

证明？证明的本质就是集合的交并差的运算。

析合树

好的，现在讲到重点了。你可能已经猜到了，析合树正是由连续段组成的一棵树。但是要知道一个排列可能有多达 $O(n^2)$ 个连续段，因此我们就要抽出其中更基本的连续段组成析合树。

本原段

其实这个定义全称叫作**本原连续段**。但笔者认为本原段更为简洁。

对于排列 P，我们认为一个本原段 M 表示在集合 I_P 中，不存在与之相交且不包含的连续段。形式化地定义，我们认为 $X \in I_P$ 且满足 $\forall A \in I_P$，$X \cap A = (P, \emptyset) \vee X \subseteq A \vee A \subseteq X$。

所有本原段的集合为 M_P. 显而易见，$(P, \emptyset) \in M_P$。

显然，本原段之间只有相离或者包含关系。并且你发现**一个连续段可以由几个互不相交的本原段构成**。最大的本原段就是整个排列本身，它包含了其他所有本原段，因此我们认为本原段可以构成一个树形结构，我们称这个结构为**析合树**。更严格地说，排列 P 的析合树由排列 P 的**所有本原段**组成。

前面干讲这么多的定义，不来点图怎么行。考虑排列 $P = \{9, 1, 10, 3, 2, 5, 7, 6, 8, 4\}$. 它的本原段构成的析合树如下：

在图中我们没有标明本原段。而图中**每个结点都代表一个本原段**。我们只标明了每个本原段的值域。举个例子，结点 $[5, 8]$ 代表的本原段就是 $(P, [6, 9]) = \{5, 7, 6, 8\}$。于是这里就有一个问题：**什么是析点合点？**

析点与合点

这里我们直接给出定义，稍候再来讨论它的正确性。

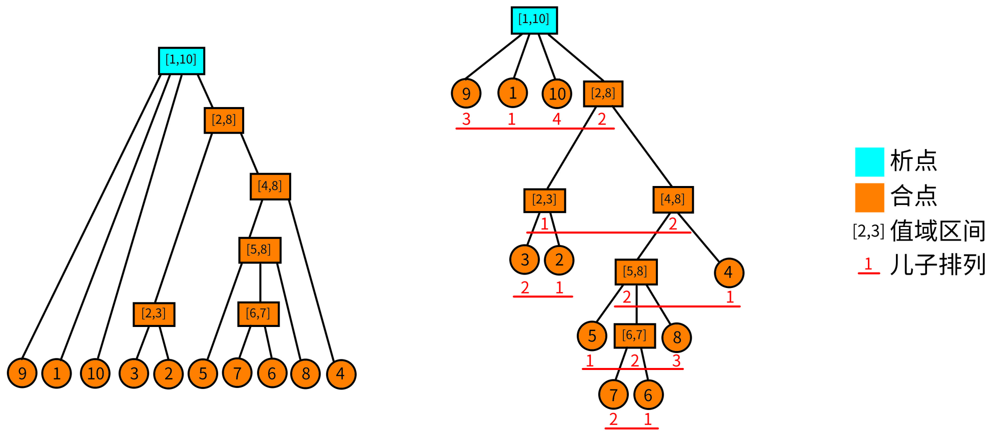


图 10.202 p1

1. **值域区间**：对于一个结点 u，用 $[u_l, u_r]$ 表示该结点的值域区间。
2. **儿子序列**：对于析合树上的一个结点 u，假设它的儿子结点是一个**有序**序列，该序列是以值域区间为元素的（单个的数 x 可以理解为 $[x, x]$ 的区间）。我们把这个序列称为儿子序列。记作 S_u。
3. **儿子排列**：对于一个儿子序列 S_u，把它的元素离散化成正整数后形成的排列称为儿子排列。举个例子，对于结点 $[5, 8]$，它的儿子序列为 $\{[5, 5], [6, 7], [8, 8]\}$，那么把区间排序标个号，则它的儿子排列就为 $\{1, 2, 3\}$；类似的，结点 $[4, 8]$ 的儿子排列为 $\{2, 1\}$。结点 u 的儿子排列记为 P_u。
4. **合点**：我们认为，儿子排列为顺序或者逆序的点为合点。形式化地说，满足 $P_u = \{1, 2, \cdots, |S_u|\}$ 或者 $P_u = \{|S_u|, |S_u - 1|, \cdots, 1\}$ 的点称为合点。**叶子结点没有儿子排列，我们也认为它是合点**。
5. **析点**：不是合点的就是析点。

从图中可以看到，只有 $[1, 10]$ 不是合点。因为 $[1, 10]$ 的儿子排列是 $\{3, 1, 4, 2\}$。

析点与合点的性质

析点与合点的命名来源于他们的性质。首先我们有一个非常显然的性质：对于析合树中任何的结点 u，其儿子序列区间的并集就是结点 u 的值域区间。即 $\bigcup_{i=1}^{|S_u|} S_u[i] = [u_l, u_r]$。

对于一个合点 u：其儿子序列的任意**子区间**都构成一个**连续段**。形式化地说，$\forall S_u[l \sim r]$，有 $\bigcup_{i=l}^{r} S_u[i] \in I_P$。

对于一个析点 u：其儿子序列的任意**长度大于 1（这里的长度是指儿子序列中的元素数，不是下标区间的长度）**的子区间都**不**构成一个**连续段**。形式化地说，$\forall S_u[l \sim r], l < r$，有 $\bigcup_{i=l}^{r} S_u[i] \notin I_P$。

合点的性质不难证明。因为合点的儿子排列要么是顺序，要么是倒序，而值域区间也是首位相接，因此只要是连续的一段子序列（区间）都是一个连续段。

对于析点的性质可能很多读者就不太能理解了：为什么**任意**长度大于 1 的子区间都不构成连续段？

使用反证法。假设对于一个点 u，它的儿子序列中有一个**最长的**区间 $S_u[l \sim r]$ 构成了连续段。那么这个 $A = \bigcup_{i=l}^{r} S_u[i] \in I_P$，也就意味着 A 是一个本原段！（因为 A 是儿子序列中最长的，因此找不到一个与它相交又不包含的连续段）于是你就没有使用所有的本原段构成这个析合树。矛盾。

析合树的构造

对于具体构造析合树，LCA 提供了一种线性构造算法[1]，下面给出一种比较好懂的 $O(n \log n)$ 算法。

增量法

我们考虑增量法。用一个栈维护前 $i-1$ 个元素构成的析合森林。在这里需要**着重强调**，析合森林的意思是，在任何时候，栈中结点要么是析点要么是合点。现在考虑当前结点 P_i。

1. 我们先判断它能否成为栈顶结点的儿子，如果能就变成栈顶的儿子，然后把栈顶取出，作为当前结点。重复上述过程直到栈空或者不能成为栈顶结点的儿子。
2. 如果不能成为栈顶的儿子，就看能不能把栈顶的若干个连续的结点都合并成一个结点（判断能否合并的方法在后面），把合并后的点，作为当前结点。
3. 重复上述过程直到不能进行为止。然后结束此次增量，直接把当前结点压栈。

接下来我们仔细解释一下。

具体的策略

我们认为，如果当前点能够成为栈顶结点的儿子，那么栈顶结点是一个合点。如果是析点，那么你合并后这个析点就存在一个子连续段，不满足析点的性质。因此一定是合点。

如果无法成为栈顶结点的儿子，那么我们就看栈顶连续的若干个点能否与当前点一起合并。设 l 为当前点所在区间的左端点。我们计算 L_i 表示右端点下标为 i 的连续段中，左端点 $< l$ 的最大值。当前结点为 P_i，栈顶结点记为 t。

1. 如果 L_i 不存在，那么显然当前结点无法合并；
2. 如果 $t_l = L_i$，那么这就是两个结点合并，合并后就是一个**合点**；
3. 否则在栈中一定存在一个点 t' 的左端点 ${t'}_l = L_i$，那么一定可以从当前结点合并到 t' 形成一个**析点**；

判断能否合并

最后，我们考虑如何处理 L_i。事实上，一个连续段 $(P, [l, r])$ 等价于区间极差与区间长度 -1 相等。即

$$\max_{l \le i \le r} P_i - \min_{l \le i \le r} P_i = r - l$$

而且由于 P 是一个排列，因此对于任意的区间 $[l, r]$ 都有

$$\max_{l \le i \le r} P_i - \min_{l \le i \le r} P_i \ge r - l$$

于是我们就维护 $\max_{l \le i \le r} P_i - \min_{l \le i \le r} P_i - (r - l)$，那么要找到一个连续段相当于查询一个最小值！

有了上述思路，不难想到这样的算法。对于增量过程中的当前的 i，我们维护一个数组 Q 表示区间 $[j, i]$ 的极差减长度。即

$$Q_j = \max_{j \le k \le i} P_k - \min_{j \le k \le i} P_k - (i - j), \quad 0 < j < i$$

现在我们想知道在 $1 \sim i - 1$ 中是否存在一个最小的 j 使得 $Q_j = 0$。这等价于求 $Q_{1 \sim i-1}$ 的最小值。求得最小的 j 就是 L_i。如果没有，那么 $L_i = i$。

但是当第 i 次增量结束时，我们需要快速把 Q 数组更新到 i+1 的情况。原本的区间从 $[j, i]$ 变成 $[j, i + 1]$，如果 $P_{i+1} > \max$ 或者 $P_{i+1} < \min$ 都会造成 Q_j 发生变化。如何变化？如果 $P_{i+1} > \max$，相当于我们把 Q_j 先减掉 max 再加上 P_{i+1} 就完成了 Q_j 的更新；$P_{i+1} < \min$ 同理，相当于 $Q_j = Q_j + \min - P_{i+1}$.

那么如果对于一个区间 $[x, y]$，满足 $P_{x \sim i}, P_{x+1 \sim i}, P_{x+2 \sim i}, \cdots, P_{y \sim i}$ 的区间 max 都相同呢？你已经发现了，那么相当于我们在做一个区间加的操作；同理，当 $P_{x \sim i}, P_{x+1 \sim i}, \cdots, P_{y \sim i}$ 的区间 min 都想同时也是一个区间加的操作。同时，max 和 min 的更新是相互独立的，因此可以各自更新。

因此我们对 Q 的维护可以这样描述：

1. 找到最大的 j 使得 $P_j > P_{i+1}$，那么显然，$P_{j+1 \sim i}$ 这一段数全部小于 P_{i+1}，于是就需要更新 $Q_{j+1 \sim i}$ 的最大值。由于 $P_i, \max(P_i, P_{i-1}), \max(P_i, P_{i-1}, P_{i-2}), \cdots, \max(P_i, P_{i-1}, \cdots, P_{j+1})$ 是（非严格）单调递增的，因此可以每一段相同的 max 做相同的更新，即区间加操作。
2. 更新 min 同理。
3. 把每一个 Q_j 都减 1。因为区间长度加 1。
4. 查询 L_i：即查询 Q 的最小值的所在的**下标**。

没错，我们可以使用线段树维护 Q！现在还有一个问题：怎么找到相同的一段使得他们的 max / min 都相同？使用单调栈维护！维护两个单调栈分别表示 max / min。那么显然，栈中以相邻两个元素为端点的区间的 max / min 是相同的，于是在维护单调栈的时侯顺便更新线段树即可。

具体的维护方法见代码。

讲这么多干巴巴的想必小伙伴也听得云里雾里的，那么我们就先上图吧。长图警告！

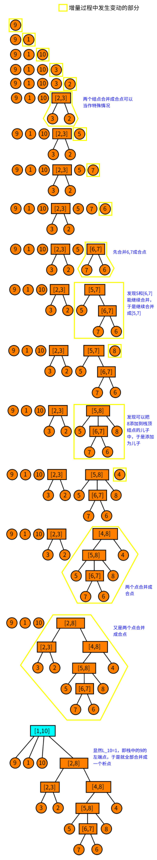


图 10.203 p2

实现

最后放一个实现的代码供参考。代码转自 大米饼的博客[2]，添加了一些注释。

```
#include <bits/stdc++.h>
#define rg register
using namespace std;
const int N = 200010;

int n, m, a[N], st1[N], st2[N], tp1, tp2, rt;
int L[N], R[N], M[N], id[N], cnt, typ[N], bin[20], st[N], tp;

// 本篇代码原题应为 CERC2017 Intrinsic Interval
// a 数组即为原题中对应的排列
// st1 和 st2 分别两个单调栈，tp1、tp2 为对应的栈顶，rt 为析合树的根
// L、R 数组表示该析合树节点的左右端点，M 数组的作用在析合树构造时有提到
// id 存储的是排列中某一位置对应的节点编号，typ 用于标记析点还是合点
// st 为存储析合树节点编号的栈，tp 为其栈顶
struct RMQ {  // 预处理 RMQ (Max & Min)
```

```
  int lg[N], mn[N][17], mx[N][17];

  void chkmn(int& x, int y) {
    if (x > y) x = y;
  }

  void chkmx(int& x, int y) {
    if (x < y) x = y;
  }

  void build() {
    for (int i = bin[0] = 1; i < 20; ++i) bin[i] = bin[i - 1] << 1;
    for (int i = 2; i <= n; ++i) lg[i] = lg[i >> 1] + 1;
    for (int i = 1; i <= n; ++i) mn[i][0] = mx[i][0] = a[i];
    for (int i = 1; i < 17; ++i)
      for (int j = 1; j + bin[i] - 1 <= n; ++j)
        mn[j][i] = min(mn[j][i - 1], mn[j + bin[i - 1]][i - 1]),
        mx[j][i] = max(mx[j][i - 1], mx[j + bin[i - 1]][i - 1]);
  }

  int ask_mn(int l, int r) {
    int t = lg[r - l + 1];
    return min(mn[l][t], mn[r - bin[t] + 1][t]);
  }

  int ask_mx(int l, int r) {
    int t = lg[r - l + 1];
    return max(mx[l][t], mx[r - bin[t] + 1][t]);
  }
} D;

// 维护 L_i

struct SEG {  // 线段树
#define ls (k << 1)
#define rs (k << 1 | 1)
  int mn[N << 1], ly[N << 1];  // 区间加; 区间最小值

  void pushup(int k) { mn[k] = min(mn[ls], mn[rs]); }

  void mfy(int k, int v) { mn[k] += v, ly[k] += v; }

  void pushdown(int k) {
    if (ly[k]) mfy(ls, ly[k]), mfy(rs, ly[k]), ly[k] = 0;
  }

  void update(int k, int l, int r, int x, int y, int v) {
    if (l == x && r == y) {
      mfy(k, v);
      return;
    }
    pushdown(k);
    int mid = (l + r) >> 1;
    if (y <= mid)
```

```
      update(ls, l, mid, x, y, v);
    else if (x > mid)
      update(rs, mid + 1, r, x, y, v);
    else
      update(ls, l, mid, x, mid, v), update(rs, mid + 1, r, mid + 1, y, v);
    pushup(k);
  }

  int query(int k, int l, int r) {  // 询问 0 的位置
    if (l == r) return l;
    pushdown(k);
    int mid = (l + r) >> 1;
    if (!mn[ls])
      return query(ls, l, mid);
    else
      return query(rs, mid + 1, r);
    // 如果不存在 0 的位置就会自动返回当前你查询的位置
  }
} T;

int o = 1, hd[N], dep[N], fa[N][18];

struct Edge {
  int v, nt;
} E[N << 1];

void add(int u, int v) {  // 树结构加边
  E[o] = (Edge){v, hd[u]};
  hd[u] = o++;
}

void dfs(int u) {
  for (int i = 1; bin[i] <= dep[u]; ++i) fa[u][i] = fa[fa[u][i - 1]][i - 1];
  for (int i = hd[u]; i; i = E[i].nt) {
    int v = E[i].v;
    dep[v] = dep[u] + 1;
    fa[v][0] = u;
    dfs(v);
  }
}

int go(int u, int d) {
  for (int i = 0; i < 18 && d; ++i)
    if (bin[i] & d) d ^= bin[i], u = fa[u][i];
  return u;
}

int lca(int u, int v) {
  if (dep[u] < dep[v]) swap(u, v);
  u = go(u, dep[u] - dep[v]);
  if (u == v) return u;
  for (int i = 17; ~i; --i)
    if (fa[u][i] != fa[v][i]) u = fa[u][i], v = fa[v][i];
  return fa[u][0];
```

```
}

// 判断当前区间是否为连续段
bool judge(int l, int r) { return D.ask_mx(l, r) - D.ask_mn(l, r) == r - l; }

// 建树
void build() {
  for (int i = 1; i <= n; ++i) {
    // 单调栈
    // 在区间 [st1[tp1-1]+1,st1[tp1]] 的最小值就是 a[st1[tp1]]
    // 现在把它出栈，意味着要把多减掉的 Min 加回来。
    // 线段树的叶结点位置 j 维护的是从 j 到当前的 i 的
    // Max{j,i}-Min{j,i}-(i-j)
    // 区间加只是一个 Tag。
    // 维护单调栈的目的是辅助线段树从 i-1 更新到 i。
    // 更新到 i 后，只需要查询全局最小值即可知道是否有解

    while (tp1 && a[i] <= a[st1[tp1]])  // 单调递增的栈，维护 Min
      T.update(1, 1, n, st1[tp1 - 1] + 1, st1[tp1], a[st1[tp1]]), tp1--;
    while (tp2 && a[i] >= a[st2[tp2]])
      T.update(1, 1, n, st2[tp2 - 1] + 1, st2[tp2], -a[st2[tp2]]), tp2--;

    T.update(1, 1, n, st1[tp1] + 1, i, -a[i]);
    st1[++tp1] = i;
    T.update(1, 1, n, st2[tp2] + 1, i, a[i]);
    st2[++tp2] = i;

    id[i] = ++cnt;
    L[cnt] = R[cnt] = i;  // 这里的 L,R 是指节点所对应区间的左右端点
    int le = T.query(1, 1, n), now = cnt;
    while (tp && L[st[tp]] >= le) {
      if (typ[st[tp]] && judge(M[st[tp]], i)) {
        // 判断是否能成为儿子，如果能就做
        R[st[tp]] = i, M[st[tp]] = L[now], add(st[tp], now), now = st[tp--];
      } else if (judge(L[st[tp]], i)) {
        typ[++cnt] = 1;  // 合点一定是被这样建出来的
        L[cnt] = L[st[tp]], R[cnt] = i, M[cnt] = L[now];
        // 这里 M 数组是记录节点最右面的儿子的左端点，用于上方能否成为儿子的判断
        add(cnt, st[tp--]), add(cnt, now);
        now = cnt;
      } else {
        add(++cnt, now);  // 新建一个结点，把 now 添加为儿子
        // 如果从当前结点开始不能构成连续段，就合并。
        // 直到找到一个结点能构成连续段。而且我们一定能找到这样
        // 一个结点。
        do add(cnt, st[tp--]);
        while (tp && !judge(L[st[tp]], i));
        L[cnt] = L[st[tp]], R[cnt] = i, add(cnt, st[tp--]);
        now = cnt;
      }
    }
    st[++tp] = now;  // 增量结束，把当前点压栈

    T.update(1, 1, n, 1, i, -1);  // 因为区间右端点向后移动一格，因此整体 -1
```

```
  }

  rt = st[1];  // 栈中最后剩下的点是根结点
}

void query(int l, int r) {
  int x = id[l], y = id[r];
  int z = lca(x, y);
  if (typ[z] & 1)
    l = L[go(x, dep[x] - dep[z] - 1)], r = R[go(y, dep[y] - dep[z] - 1)];
  // 合点这里特判的原因是因为这个合点不一定是最小的包含 l, r 的连续段.
  // 因为合点所代表的区间的子区间也都是连续段，而我们只需要其中的一段就够了。
  else
    l = L[z], r = R[z];
  printf("%d %d\n", l, r);
}  // 分 lca 为析或和，这里把叶子看成析的

int main() {
  scanf("%d", &n);
  for (int i = 1; i <= n; ++i) scanf("%d", &a[i]);
  D.build();
  build();
  dfs(rt);
  scanf("%d", &m);
  for (int i = 1; i <= m; ++i) {
    int x, y;
    scanf("%d%d", &x, &y);
    query(x, y);
  }
  return 0;
}

// 20190612
// 析合树
```

参考文献与链接

大米饼的博客 -【学习笔记】 析合树[2]

[1] 刘承奥。简单的连续段数据结构。WC2019 营员交流。

[2] 大米饼的博客 [2-1] [2-2]

10.24 PQ 树

Authors: isdanni,xyf007

PQ 树是一种基于树的数据结构，代表一组元素上的一系列排列，由 Kellogg S. Booth 和 George S. Lueker 于 1976 年发现命名，用来解决以下问题

> 给出 m 个集合 S_i，你要找到一个 $1 \sim n$ 的排列，使得每个集合内的元素都相邻。

PQ 树可以在 $O(n + \sum |S_i|)$ 时间内构建。本文中介绍的构建方法时间复杂度为 $O(nm)$。

定义

PQ 树有三种结点：**叶子结点、P 结点**和 **Q 结点**。其中叶子结点代表排列中的一个元素，P 结点表示它的子结点可以任意排列，Q 结点表示它的儿子顺序可以反转。所有非叶子结点都是 P 结点或 Q 结点中的一种。P 结点至少有 2 个儿子，Q 结点至少有 3 个儿子。

由于结点的定义，PQ 树本身代表了**所有的**合法方案，其先序遍历就是其中之一。

下图是一棵 PQ 树。

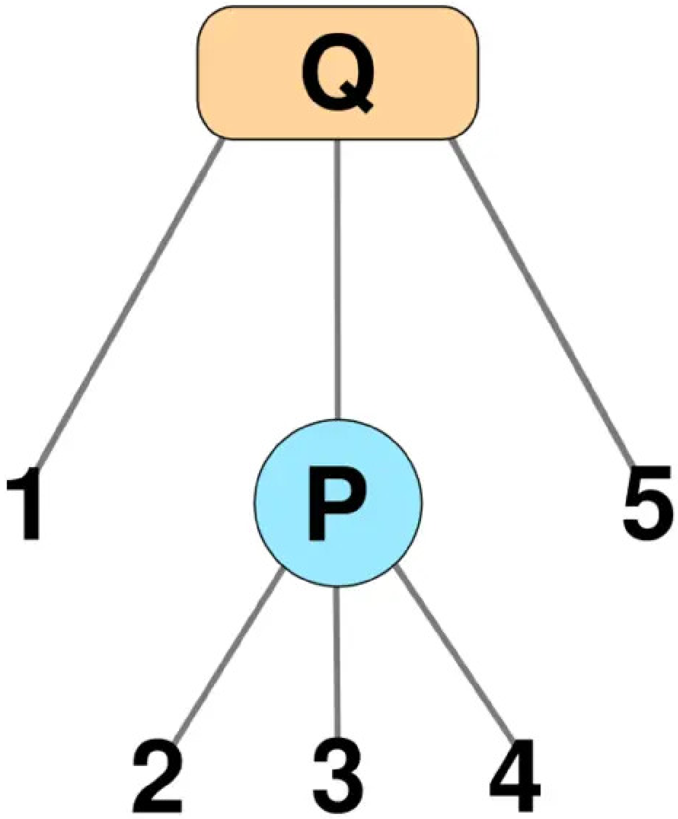


图 10.204

其先序遍历 1,2,3,4,5 代表了一个合法方案。如果 P 结点的儿子重排列为 4,2,3，我们得到了另一个合法方案 1,4,2,3,5。保持 P 结点儿子顺序不变，Q 结点的儿子顺序反转，得到了另一个合法方案 5,3,2,4,1。

构建

PQ 树使用儿子 - 兄弟表示法。

我们增量构建一棵 PQ 树。

首先建立一棵树，其根为 P，总共 n 个儿子，分别是 $1, 2, \dots, n$，代表没有任何限制时的 PQ 树。随着限制的不断加入，我们不断修改这棵树。

当加入一个新的限制集合 S 时，我们把所有属于这个集合的叶子结点标记为**黑色**，不在这个集合内的叶子结点标记为**白色**。对于所有非叶子结点，如果其所有儿子均为黑色，将其也标记为黑色；如果其所有儿子均为白色，将其也标记为白色；否则将其标记为**灰色**。在下面的图中，黑色结点、白色结点、灰色结点分别用黑色、灰色、一半黑一半灰来表示。

我们要求 PQ 树中的结点按照颜色排序。

自底向上法

包含所有黑色结点的最小子树被称为**相关子树**，相关子树的根（不一定是整棵树的根）被称为**相关根**。

添加一个限制的过程被称为 reduction。一次 reduction 分为两个阶段：冒泡阶段和减少阶段。

冒泡阶段

冒泡阶段只处理相关子树。我们将相关子树中的所有结点标记为黑色或灰色，并为每个结点计算其拥有的相关子结点数量。为了高效地完成这个过程，我们从叶子往根处理相关子树。这需要记录每个点的父亲结点，但在减少阶段一个点的父亲结点经常要被修改。为了在线性时间内构造，只有 P 结点的儿子和 **Q 结点的最后一个儿子**始终记录正确的父亲结点。对于 Q 结点的其他儿子，在冒泡阶段用最后一个儿子的父亲更新他们的父亲。

当遇到一个中间的结点时，我们看一下它的兄弟是否已经有合法的父亲结点。如果没有，将其标记为**阻塞**的。如果后面它的兄弟有了合法的父亲，那么修改这个结点的父亲并且取消标记。如果在冒泡阶段结束时，仍然有一段连续的阻塞结点（如下面的情况 Q3），一个没有父结点的「伪结点」成为该块的父结点，并在减少阶段时被去除。

减少阶段

减少阶段用一个队列来处理结点。首先将所有限制内的叶子结点加入队列。每次取出队首的结点 u 并处理。如果 u 的父亲也是相关子树内的结点，那么将 fa_u 入队。

对于每一个结点 u，我们分情况讨论。如果不属于其中任何一种情况，则无解。

叶子结点

将 u 标记为黑色。

P 结点

如果所有儿子均为黑色，将 u 标记为黑色。

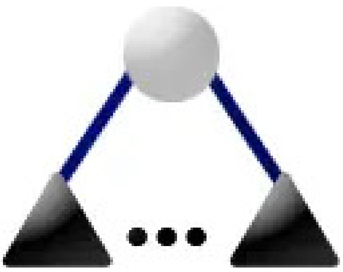


图 10.205

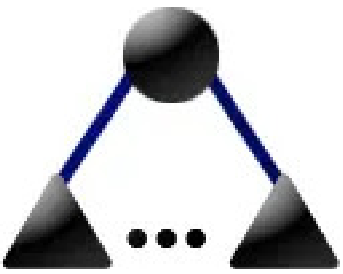


图 10.206

如果 u 有黑色儿子和白色儿子，且 u 是相关根，那么新建一个 P 结点 v 成为它所有黑色儿子的根。

如果 u 有黑色儿子和白色儿子，且 u 不是相关根，那么做以下操作：

- 新建一个 P 结点 f 成为所有黑色儿子的根。
- 新建一个 P 结点 e 成为所有白色儿子的根。
- 如果 e（和/或 f）只有一个儿子，那么不要新建结点，而是将 e（和/或 f）直接赋值成那个儿子。

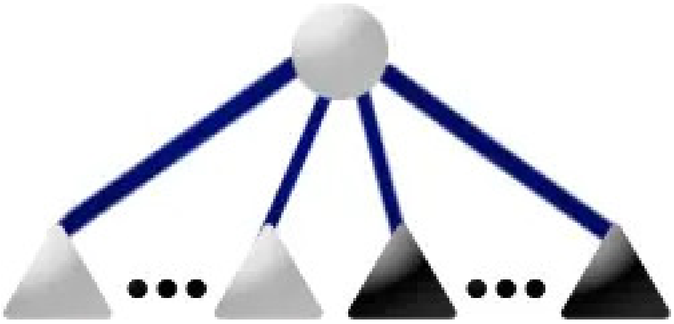


图 10.207

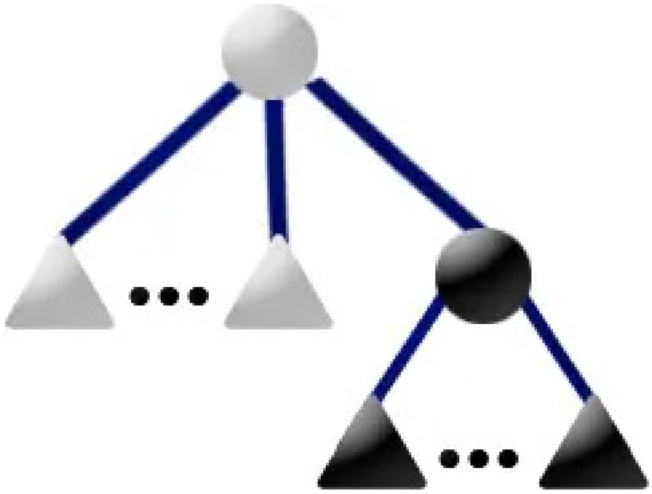


图 10.208

- 将 u 改成 Q 结点并把其儿子设为 e 和 f，将其标记为灰色。

注意到根据之前的定义，Q 结点至少有 3 个儿子，因此这里的 u 被视为一个「伪结点」，并且将在后面被继续处理。

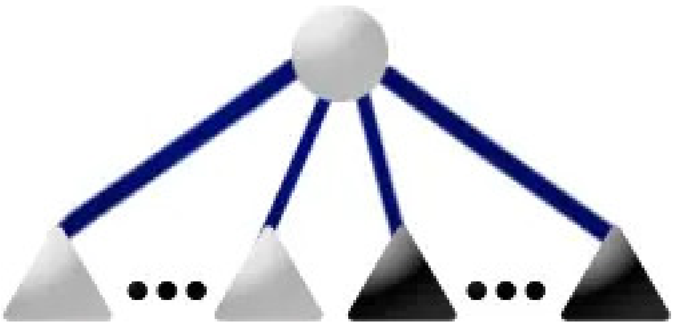


图 10.209

如果 u 有一个灰色儿子 p，且 u 是相关根，那么新建一个 P 结点 v 作为其所有黑色儿子的根，将 v 的兄弟设为 p 最后一个一个黑色儿子，然后把 v 设为 p 的最后一个儿子。

如果 u 有一个灰色儿子 p，且 u 不是相关根，那么进行以下操作：

- 新建一个 P 结点 f 成为所有黑色儿子的根。
- 新建一个 P 结点 e 成为所有白色儿子的根。
- 如果 e（和/或 f）只有一个儿子，那么不要新建结点，而是将 e（和/或 f）直接赋值成那个儿子。
- 将 e 的兄弟设为 p 最后一个白色儿子，然后把 e 设为 p 的最后一个儿子。
- 将 f 的兄弟设为 p 最后一个黑色儿子，然后把 f 设为 p 的最后一个儿子。

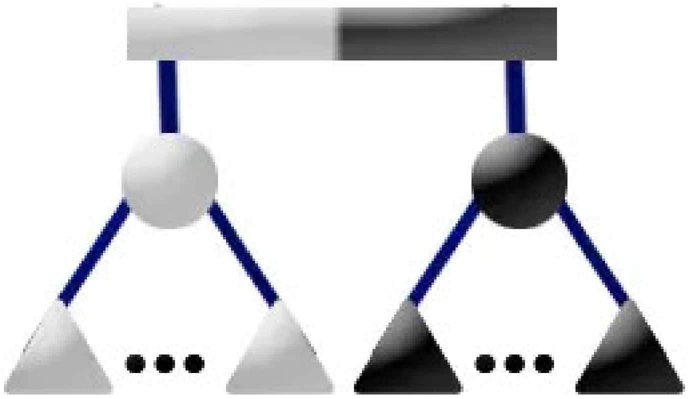


图 10.210

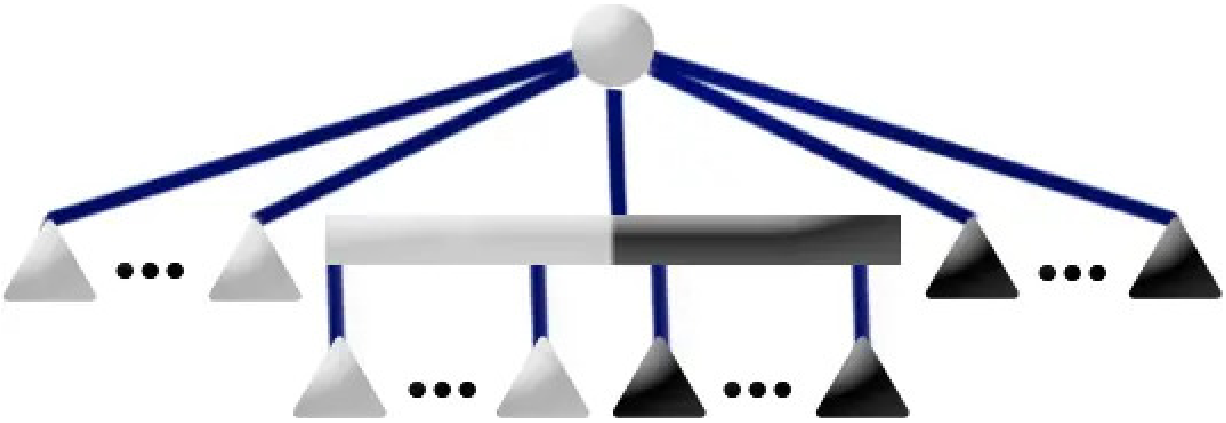


图 10.211

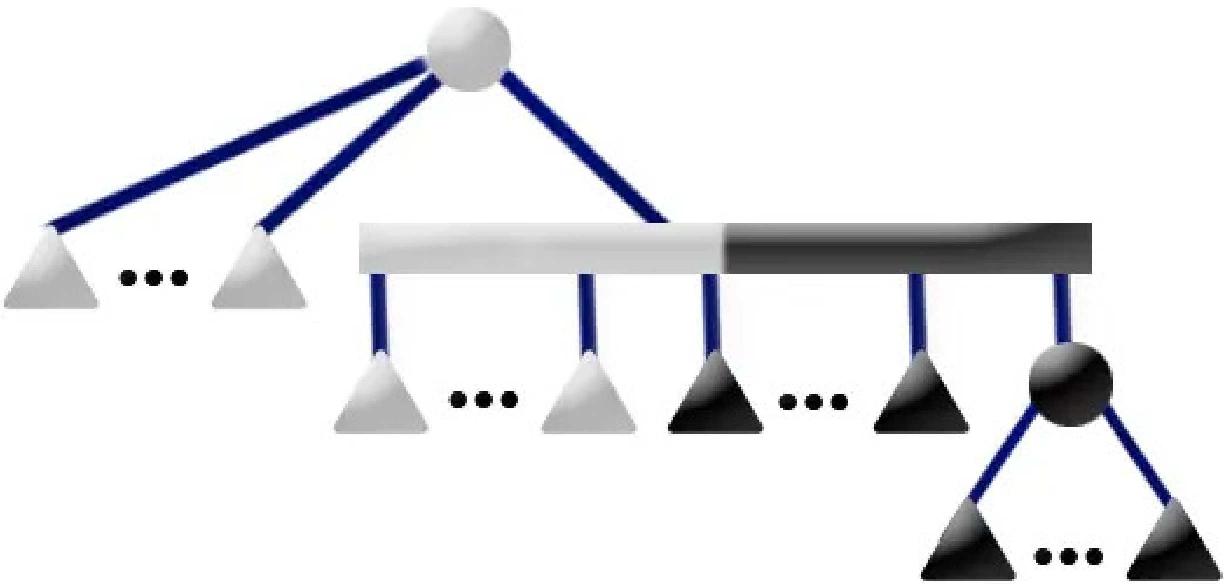


图 10.212

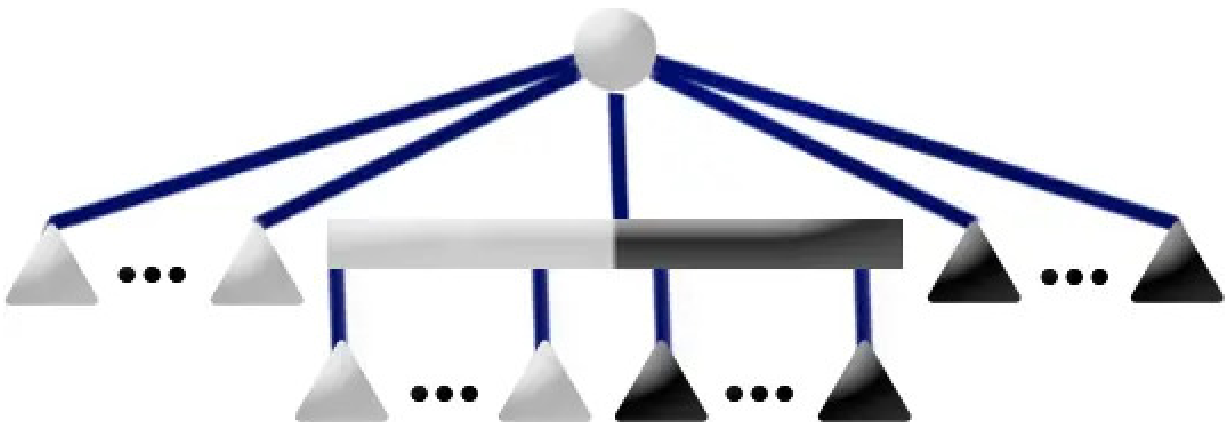


图 10.213

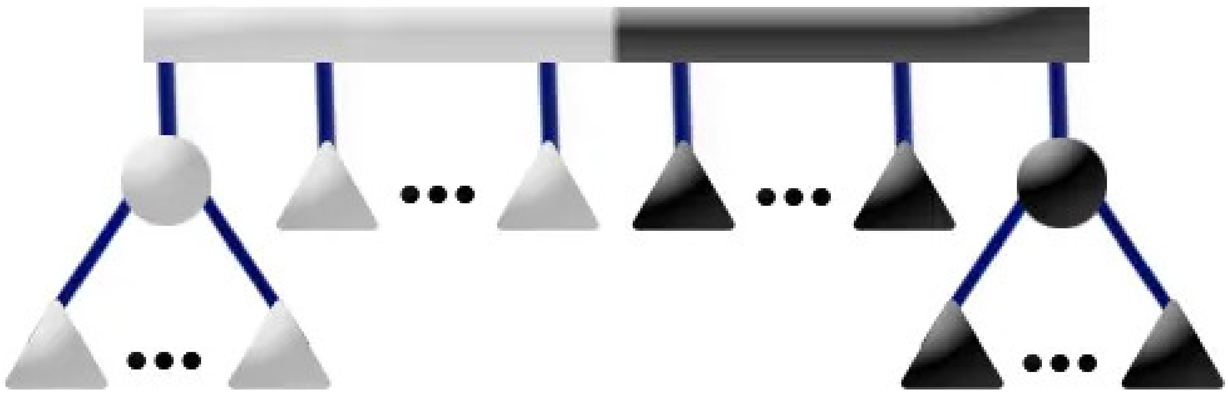


图 10.214

如果 u 恰有两个灰色儿子 p_1, p_2，那么进行以下操作：

- 新建一个 P 结点 f 成为所有黑色儿子的根。
- 如果 f 只有一个儿子，那么不要新建结点，而是将 f 直接赋值成那个儿子。
- 把 p_1 的最后一个黑色儿子的兄弟设为 f。
- 把 f 的兄弟设为 p_2 的最后一个黑色儿子。
- 把 p_2 的最后一个儿子设为 p_2 的最后一个白色儿子。

可以发现这样 p_2 就被合并进了 p_1。

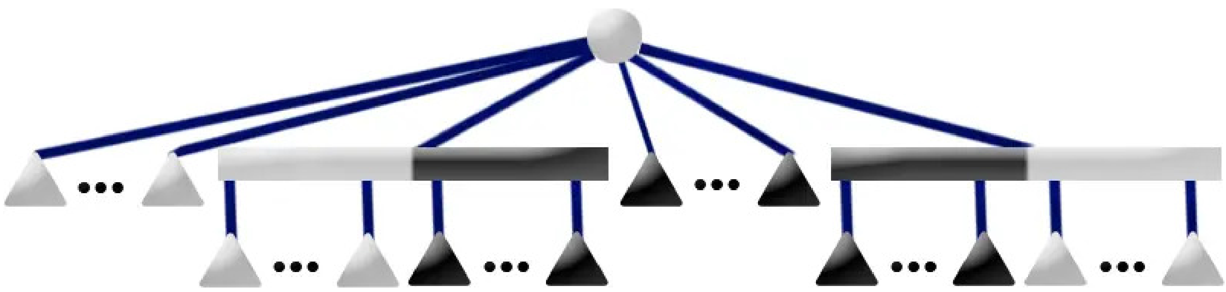


图 10.215

Q 结点

如果 u 只有黑色儿子，那么将 u 标记成黑色。（下面的图的形状错了。）

如果 u 有一个灰色儿子 p，且所有标记相同的儿子均连续出现，那么进行如下操作：

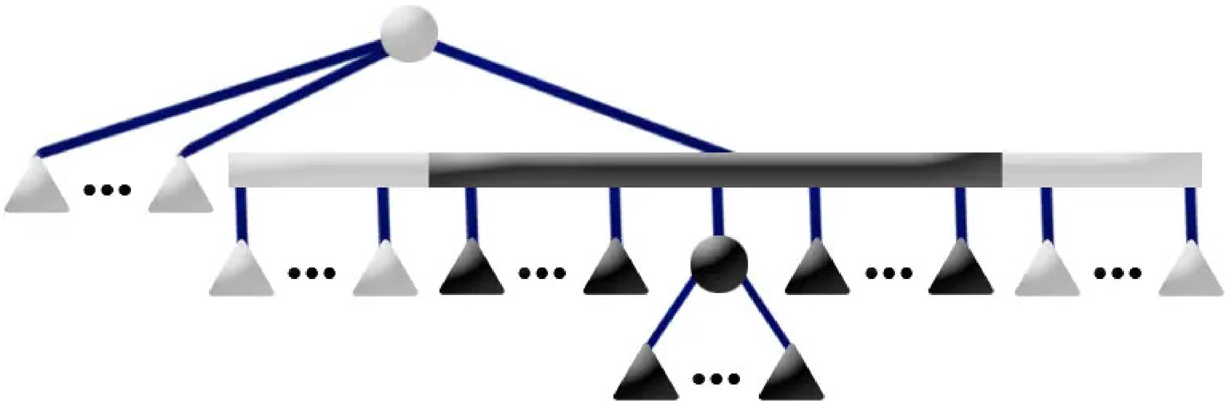


图 10.216

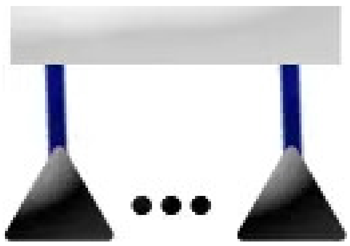


图 10.217

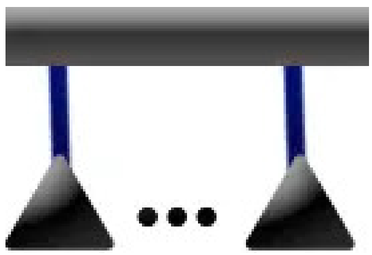


图 10.218

- 设 p_f 为 p 最后一个黑色儿子，p_e 为 p 最后一个白色儿子，f 为 p 的黑色兄弟，e 为 p 的白色兄弟。
- 将 f 的兄弟设为 p_f，e 的兄弟设为 p_e。
- 如果 p 没有一个白色兄弟或黑色兄弟，将 u 的最后一个儿子设成 p 的最后一个儿子。
- 删除 p。

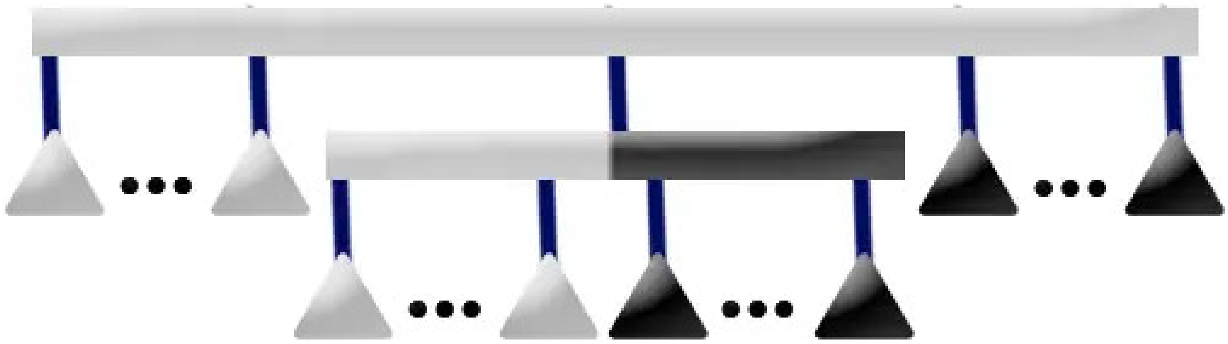


图 10.219

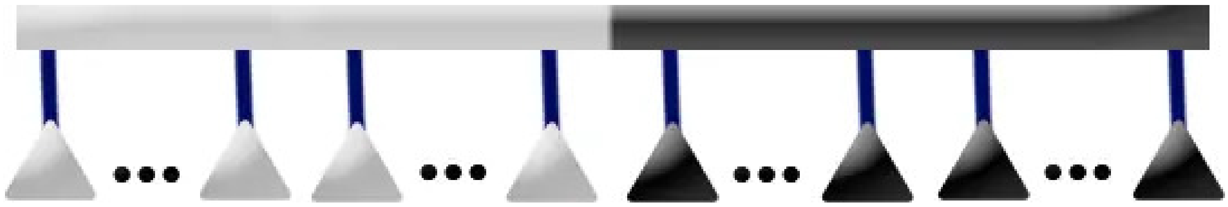


图 10.220

如果 u 恰有两个灰色儿子 p_1,p_2，且所有标记相同的儿子均连续出现，那么对 p_1,p_2 都进行上一种操作即可。

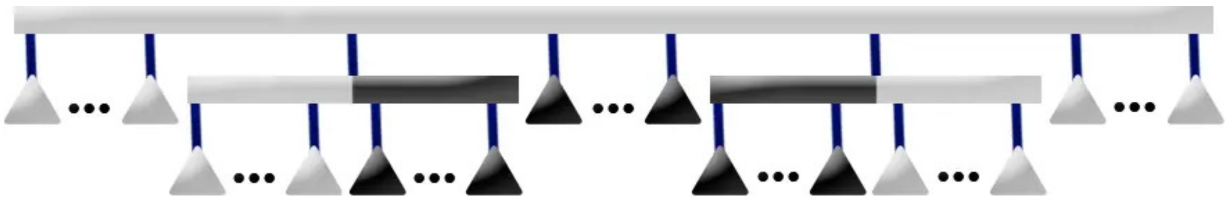


图 10.221

该构建方法是原论文中的，但是实现较为不便。

自顶向下法

目前 OI 中的实现大多采用该方法。其实方法类似，下面出现的情况基本都能在上面找到。

注意到根据之前的染色过程，所有黑色和白色的点都已经满足条件，因此我们**只需要处理灰色结点**。

P 结点

- 如果 u 有多于两个灰色儿子，无解。
- 如果 u 只有一个灰色儿子，且没有黑色儿子，递归处理灰色儿子。
- 否则先清空 u 的儿子，然后加入所有的白色儿子。新建一个 Q 结点 q_1 并成为 u 的儿子。在 q_1 中加入所有的灰色儿子。新建一个 P 结点 p 作为所有黑色儿子的根，将 p 插入 q_1 的中间。（对应了自底向上法 P 结点的所有情况。）

注意到我们会要求两个灰色节点白色全在左侧，黑色全在右侧（或相反），因此我们需要实现一个分裂函数 `split`，可以把这个子树的点分裂成黑白部分，并同时保留分裂成的子树的节点的**所有可能**。

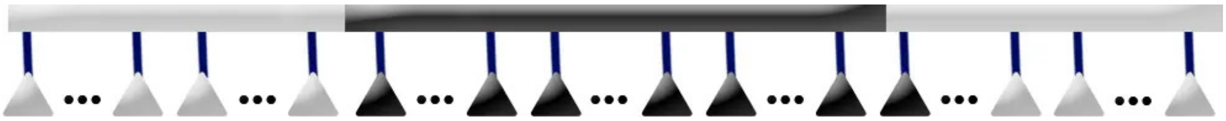


图 10.222

Q 结点

- 找到最左边和最右边的非白色节点位置 l, r。如果 $[l+1, r-1]$ 内有非黑色节点，无解。
- 如果没有黑色节点，只有一个灰色节点，递归处理这个灰色节点，否则只需要将 l 和 r 位置的节点分裂。

分裂函数

令要分裂的点为 u，我们想把 u 分裂成左边全是白色，右边全是黑色的森林。如果 u 不是灰色结点则直接返回子树。只考虑灰色结点的情况。如果 u 是 P 类结点：

- 如果 u 有至少两个灰色儿子，则无解。
- 否则左边是所有白色儿子，中间递归处理灰色儿子，右边是所有黑色儿子。注意到要保留所有的可能，因此要新建两个 P 结点分别作为白色儿子和黑色儿子的根。（对应自底向上法的 P4 情况。）
- 删除 u。

如果 u 是 Q 类结点：

- 如果正序和反序都不满足白 - 灰 - 黑，则无解。
- 如果有至少两个灰色儿子，也无解。
- 否则递归分裂灰色儿子即可。
- 删除 u。

最后把所有多余的结点（只有一个儿子的结点）删除。

代码实现

```
class PQTree {
 public:
  PQTree() {}

  void Init(int n) {
    n_ = n, rt_ = tot_ = n + 1;
    for (int i = 1; i <= n; i++) g_[rt_].emplace_back(i);
  }

  void Insert(const std::string &s) {
    s_ = s;
    Dfs0(rt_);
    Work(rt_);
    while (g_[rt_].size() == 1) rt_ = g_[rt_][0];
    Remove(rt_);
  }

  std::vector<int> ans() {
    DfsAns(rt_);
    return ans_;
  }
```

```cpp
  ~PQTree() {}

 private:
  int n_, rt_, tot_, pool_[100001], top_, typ_[100001] /* 0-P 1-Q */,
      col_[100001] /* 0-black 1-white 2-grey */;
  std::vector<int> g_[100001], ans_;
  std::string s_;

  void Fail() {
    std::cout << "NO\n";
    std::exit(0);
  }

  int NewNode(int ty) {
    int x = top_ ? pool_[top_--] : ++tot_;
    typ_[x] = ty;
    return x;
  }

  void Delete(int u) { g_[u].clear(), pool_[++top_] = u; }

  void Dfs0(int u) {  // get color of each node
    if (u >= 1 && u <= n_) {
      col_[u] = s_[u] == '1';
      return;
    }
    bool c0 = false, c1 = false;
    for (auto &&v : g_[u]) {
      Dfs0(v);
      if (col_[v]) c1 = true;
      if (col_[v] != 1) c0 = true;
    }
    if (c0 && !c1)
      col_[u] = 0;
    else if (!c0 && c1)
      col_[u] = 1;
    else
      col_[u] = 2;
  }

  bool Check(const std::vector<int> &v) {
    int p2 = -1;
    for (int i = 0; i < static_cast<int>(v.size()); i++)
      if (col_[v[i]] == 2) {
        if (p2 != -1) return false;
        p2 = i;
      }
    if (p2 == -1)
      for (int i = 0; i < static_cast<int>(v.size()); i++)
        if (col_[v[i]]) {
          p2 = i;
          break;
        }
```

```
  for (int i = 0; i < p2; i++)
    if (col_[v[i]]) return false;
  for (int i = p2 + 1; i < static_cast<int>(v.size()); i++)
    if (col_[v[i]] != 1) return false;
  return true;
}

std::vector<int> Split(int u) {
  if (col_[u] != 2) return {u};
  std::vector<int> ng;
  if (typ_[u]) {  // Q
    if (!Check(g_[u])) {
      std::reverse(g_[u].begin(), g_[u].end());
      if (!Check(g_[u])) Fail();
    }
    for (auto &&v : g_[u])
      if (col_[v] != 2) {
        ng.emplace_back(v);
      } else {
        auto s = Split(v);
        ng.insert(ng.end(), s.begin(), s.end());
      }
  } else {  // P
    std::vector<int> son[3];
    for (auto &&x : g_[u]) son[col_[x]].emplace_back(x);
    if (son[2].size() > 1) Fail();
    if (!son[0].empty()) {
      int n0 = NewNode(0);
      g_[n0] = son[0];
      ng.emplace_back(n0);
    }
    if (!son[2].empty()) {
      auto s = Split(son[2][0]);
      ng.insert(ng.end(), s.begin(), s.end());
    }
    if (!son[1].empty()) {
      int n1 = NewNode(0);
      g_[n1] = son[1];
      ng.emplace_back(n1);
    }
  }
  Delete(u);
  return ng;
}

void Work(int u) {
  if (col_[u] != 2) return;
  if (typ_[u]) {  // Q
    int l = 1e9, r = -1e9;
    for (int i = 0; i < static_cast<int>(g_[u].size()); i++)
      if (col_[g_[u][i]]) checkmin(l, i), checkmax(r, i);
    for (int i = l + 1; i < r; i++)
      if (col_[g_[u][i]] != 1) Fail();
    if (l == r && col_[g_[u][l]] == 2) {
```

```
      Work(g_[u][l]);
      return;
    }
    std::vector<int> ng;
    for (int i = 0; i < l; i++) ng.emplace_back(g_[u][i]);
    auto s = Split(g_[u][l]);
    ng.insert(ng.end(), s.begin(), s.end());
    for (int i = l + 1; i < r; i++) ng.emplace_back(g_[u][i]);
    if (l != r) {
      s = Split(g_[u][r]);
      std::reverse(s.begin(), s.end());
      ng.insert(ng.end(), s.begin(), s.end());
    }
    for (int i = r + 1; i < static_cast<int>(g_[u].size()); i++)
      ng.emplace_back(g_[u][i]);
    g_[u] = ng;
  } else {  // P
    std::vector<int> son[3];
    for (auto &&x : g_[u]) son[col_[x]].emplace_back(x);
    if (son[1].empty() && son[2].size() == 1) {
      Work(son[2][0]);
      return;
    }
    g_[u].clear();
    if (son[2].size() > 2) Fail();
    g_[u] = son[0];
    int n1 = NewNode(1);
    g_[u].emplace_back(n1);
    if (son[2].size() >= 1) {
      auto s = Split(son[2][0]);
      g_[n1].insert(g_[n1].end(), s.begin(), s.end());
    }
    if (son[1].size()) {
      int n2 = NewNode(0);
      g_[n1].emplace_back(n2);
      g_[n2] = son[1];
    }
    if (son[2].size() >= 2) {
      auto s = Split(son[2][1]);
      std::reverse(s.begin(), s.end());
      g_[n1].insert(g_[n1].end(), s.begin(), s.end());
    }
  }
}

void Remove(int u) {  // remove the nodes with only one child
  for (auto &&v : g_[u]) {
    int tv = v;
    while (g_[tv].size() == 1) {
      int t = tv;
      tv = g_[tv][0];
      Delete(t);
    }
    v = tv, Remove(v);
```

```
    }
  }

  void DfsAns(int u) {
    if (u >= 1 && u <= n_) {
      ans_.emplace_back(u);
      return;
    }
    for (auto &&v : g_[u]) DfsAns(v);
  }
} T;
```

习题

- CF243E Matrix[1]
- CF1552I Organizing a Music Festival[2]

参考资料

- Booth, Kellogg S. & Lueker, George S. (1976).”Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms”[3].*Journal of Computer and System Sciences*[4].**13**(3): 335–379.doi[5]:10.1016/S0022-0000(76)80045-1[6].
- PQ Tree Algorithm and Consecutive Ones Problem[7]
- CF243E Matrix PQTree - RainAir’s Blog[8]

参考资料与注释

[1] CF243E Matrix

[2] CF1552I Organizing a Music Festival

[3] ”Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms”

[4] Journal of Computer and System Sciences

[5] doi

[6] 10.1016/S0022-0000(76)80045-1

[7] PQ Tree Algorithm and Consecutive Ones Problem

[8] CF243E Matrix PQTree - RainAir’s Blog

10.25 手指树

Authors: isdanni

> " 注意"
>
> 此章是选读内容，在阅读前请确定你对函数式编程（Functional Programming）有一定了解。

简介

手指树（Finger Tree）是一种**纯函数式**数据结构，由 Ralf Hinze 和 Ross Paterson 提出。

为什么需要手指树

在函数式编程中，列表是十分常见的数据类型。对于基于序列的操作，包括在两端添加和删除元素（双端队列操作），在任意节点插入、连接、删除，查找某个满足要求的元素，将序列拆分为子序列，几乎所有的函数型语言都支持。但是对于高效的更多操作，这些语言很难做到。即使有相对应的实现，通常也都非常复杂，实际很难使用。

而指状树提供了一种纯函数式的序列数据结构，它可以在均摊常量时间（amortized constant time）内完成访问，添加到序列的前端和末尾等操作，以及在对数时间（logarithmic time）内完成串联和随机访问。除了良好的渐近运行时边界外，手指树还非常灵活：当与元素上的幺半群标记（monoidal tag[2]）结合时，指状树可用于实现高效的随机访问序列、有序序列、间隔树和优先级队列。

基本结构

手指树在树的「手指」（叶子）的地方存储数据，访问时间为分摊常量。手指是一个可以访问部分数据结构的点。在命令式语言（imperative language）中，这被称做指针。在手指树中，「手指」是指向序列末端或叶节点的结构。手指树还在每个内部节点中存储对其后代应用一些关联操作的结果。存储在内部节点中的数据可用于提供除树类数据结构之外的功能。

1. 手指树的深度由下到上计算。
2. 手指树的第一级，即树的叶节点，仅包含值，深度为 0。第二级为深度 1。第三级为深度 2，依此类推。
3. 离根越近，节点指向的原始树（在它是手指树之前的树）的子树越深。这样，沿着树向下工作就是从叶子到树的根，这与典型的树数据结构相反。为了获得这种的结构，我们必须确保原始树具有统一的深度。在声明节点对象时，必须通过子节点的类型进行参数化。深度为 1 及以上的脊椎上的节点指向树，通过这种参数化，它们可以由嵌套节点表示。

将一棵树变成手指树

> " 注释"
>
> **2-3 树**是一种树状数据结构，其中每个带有子节点（内部节点）的节点具有两个子节点（2 节点）和一个数据元素或三个子节点（3 节点）和两个数据元素。2-3 树是 3 阶 B 树。树外部的节点（叶节点）没有子节点和一两个数据元素。

我们将从平衡 2-3 树开始这个过程。为了使手指树正常工作，所有的叶节点需要是水平的。如下图所示（图片取自手指树论文）：

手指是「一种结构，可以有效地访问靠近特定位置的树的节点。」要制作手指树，我们需要将手指放在树的左右两端，取树的最左边和最右边的内部节点并将它们拉起来，使树的其余部分悬在它们之间，这为我们提供了对序列末尾的均摊常量访问时间。

这种新的数据结构被称为手指树。手指树由沿其树脊（棕色线）分布的几层（下方蓝色框）组成：

```
data FingerTree a = Empty
                  | Single a
                  | Deep (Digit a) (FingerTree (Node a)) (Digit a)
```

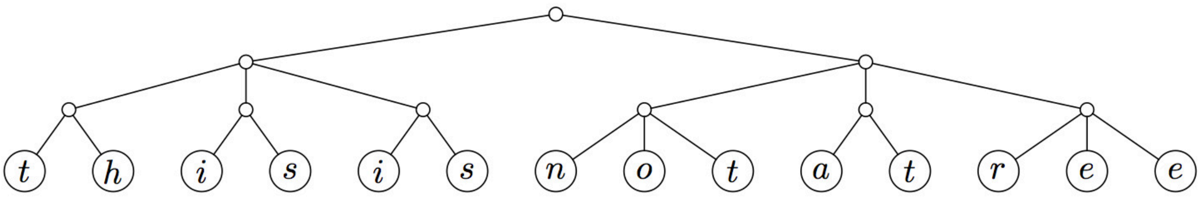


图 10.223

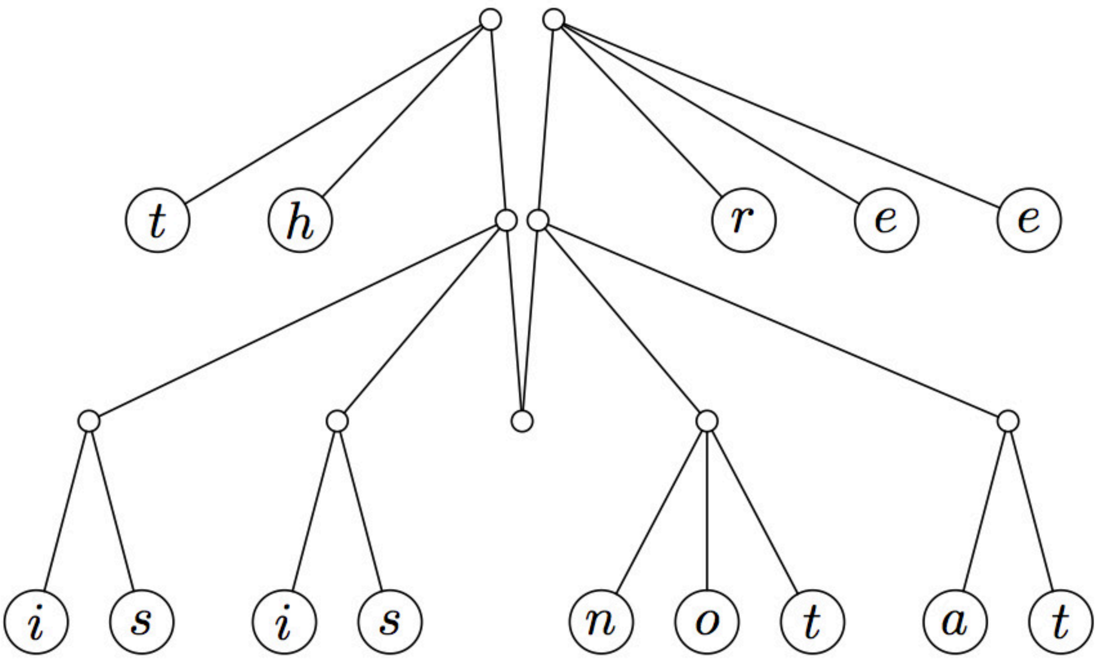


图 10.224

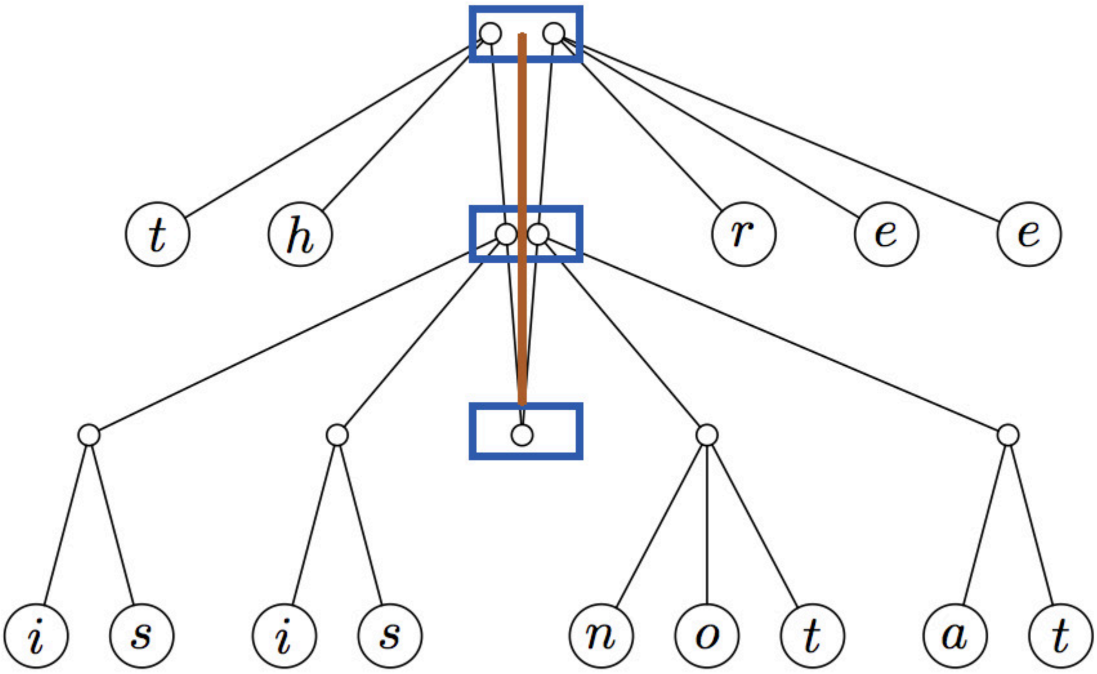


图 10.225

```
data Digit a = One a | Two a a | Three a a a | Four a a a a
data Node a = Node2 a a | Node3 a a a
```

示例中的数字是带有字母的节点。每个列表由树脊上每个节点的前缀或后缀划分。在转换后的 2-3 树中，顶层的数字列表似乎可以有两个或三个长度，而较低级别的长度只有一或两个。为了使手指树的某些应用程序能够如此高效地运行，手指树允许在每个级别上有 1 到 4 个子树。手指树的数字可以转换成一个列表，如：

```
type Digit a = One a | Two a a | Three a a a | Four a a a a
```

顶层具有类型 a 的元素，下一层具有类型节点 a 的元素，因为树脊和叶子之间的节点，这通常意味着树的第 n 层具有元素类型为 $Node^n\ a$，或 2-3 个深度为 n 的树。这意味着 n 个元素的序列由深度为 `Θ(log n)` 的树表示。距离最近端 d 的元素存储在树中 `Θ(log d)` 深度处。

双向队列操作

指状树也可以制作高效的双向队列。无论结构是否持久，所有操作都需要 `Θ(1)` 时间。它可以被看作是的隐式双端队列的扩展[1]：

1. 用 2-3 个节点替换对提供了足够的灵活性来支持有效的串联。（为了保持恒定时间的双端队列操作，必须将 Digit 扩展为四。）
2. 用幺半群（monoid）注释内部节点允许有效的分裂。

```
data ImplicitDeque a = Empty
                     | Single a
                     | Deep (Digit a) (ImplicitDeque (a, a)) (Digit a)

data Digit a = One a | Two a a | Three a a a
```

时间复杂度

手指树提供了对树的「手指」（叶子）的分摊常量时间访问，这是存储数据的地方，以及在较小部分的大小中连接和拆分对数时间。它还在每个内部节点中存储对其后代应用一些关联操作的结果。存储在内部节点中的「摘要」数据可用于提供除树之外的数据结构的功能。

操作	手指树	注释 2-3 树（annotated 2-3 tree）	列表（list）	向量（vector）
`const`,`snoc`	$O(1)$	$O(\log n)$	$O(1)/O(n)$	$O(n)$
`viewl`,`viewr`	$O(1)$	$O(\log n)$	$O(1)/O(n)$	$O(1)$
`measure`/`length`	$O(1)$	$O(1)$	$O(n)$	$O(1)$
`append`	$O(\log\min(l1, l2))$	$O(\log n)$	$O(n)$	$O(m+n)$
`split`	$O(\log\min(n, l-n))$	$O(\log n)$	$O(n)$	$O(1)$
`replicate`	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$
`fromList`,`toList`,`reverse`	$O(l)/O(l)/O(l)$	$O(l)$	$O(1)/O(1)/O(n)$	$O(n)$
`index`	$O(\log\min(n, l-n))$	$O(\log n)$	$O(n)$	$O(1)$

应用

指状树可用于建造其他树。例如，优先级队列可以通过树中子节点的最小优先级标记内部节点来实现，或者索引列表/数组可以通过节点的子节点中叶子的计数来标记节点来实现。其他应用包括随机访问序列（如下所述）、有序序列和区间树。

手指树可以提供平均 $O(1)$ 的推、反转、弹出，$O(\log n)$ 追加和拆分；并且可以适应索引或排序序列。和所有函数式数据结构一样，它本质上是持久的；也就是说，始终保留旧版本的树。

对于代码实现，Haskell 核心库中的有限序列 `Seq` 的实现使用了 2-3 手指树（Data.Sequence[3]），OCaml 中 `BatFingerTree` 模块的 实现[4] 也使用了通用手指树数据结构。手指树可以使用或不使用惰性求值来实现，但惰性允许更简单的实现。

参考资料与拓展阅读

1. Ralf Hinze and Ross Paterson, ”Finger trees: a simple general-purpose data structure[5]”, Journal of Functional Programming 16:2 (2006) pp 197-217.
2. Finger Tree - Wikipedia[6]

[1] Purely Functional Data Structures, Chris Okasaki (1999)

[2] monoidal tag

[3] Data.Sequence

[4] 实现

[5] Finger trees: a simple general-purpose data structure

[6] Finger Tree - Wikipedia

10.26 霍夫曼树

Authors: Alex-McAvoy

树的带权路径长度

设二叉树具有 n 个带权叶结点，从根结点到各叶结点的路径长度与相应叶节点权值的乘积之和称为**树的带权路径长度（Weighted Path Length of Tree，WPL）**。

设 w_i 为二叉树第 i 个叶结点的权值，l_i 为从根结点到第 i 个叶结点的路径长度，则 WPL 计算公式如下：

$$WPL = \sum_{i=1}^{n} w_i l_i$$

如上图所示，其 WPL 计算过程与结果如下：

$$WPL = 2*2+3*2+4*2+7*2 = 4+6+8+14 = 32$$

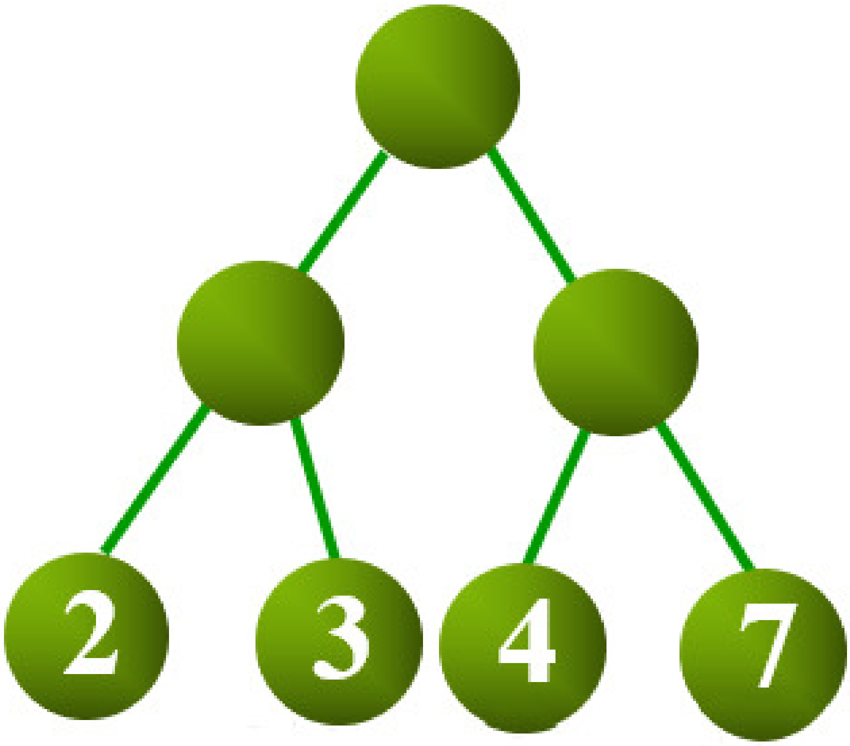


图 10.226

结构

对于给定一组具有确定权值的叶结点，可以构造出不同的二叉树，其中，**WPL 最小的二叉树**称为**霍夫曼树**(**Huffman Tree**)。

对于霍夫曼树来说，其叶结点权值越小，离根越远，叶结点权值越大，离根越近，此外其仅有叶结点的度为 0，其他结点度均为 2。

霍夫曼算法

霍夫曼算法用于构造一棵霍夫曼树，算法步骤如下：

1. **初始化**：由给定的 n 个权值构造 n 棵只有一个根节点的二叉树，得到一个二叉树集合 F。
2. **选取与合并**：从二叉树集合 F 中选取根节点权值**最小的两棵**二叉树分别作为左右子树构造一棵新的二叉树，这棵新二叉树的根节点的权值为其左、右子树根结点的权值和。
3. **删除与加入**：从 F 中删除作为左、右子树的两棵二叉树，并将新建立的二叉树加入到 F 中。
4. 重复 2、3 步，当集合中只剩下一棵二叉树时，这棵二叉树就是霍夫曼树。

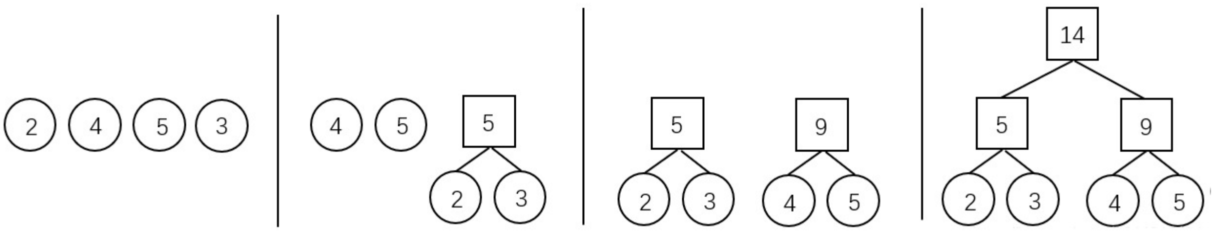


图 10.227

霍夫曼编码

在进行程序设计时，通常给每一个字符标记一个单独的代码来表示一组字符，即**编码**。

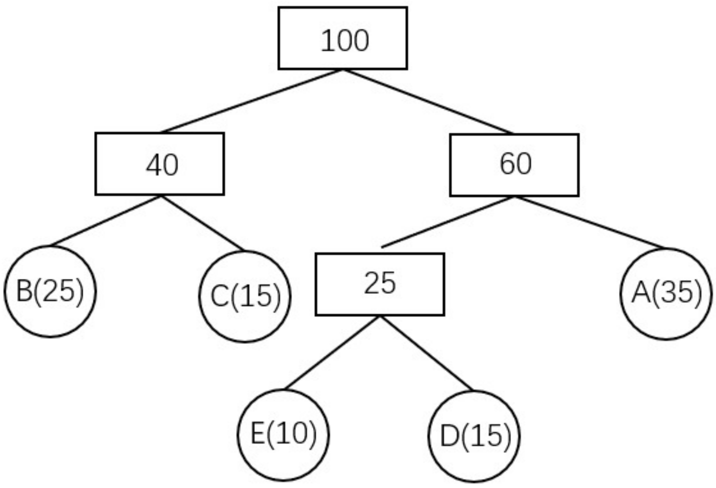
在进行二进制编码时，假设所有的代码都等长，那么表示 n 个不同的字符需要 $\lceil \log_2 n \rceil$ 位，称为**等长编码**。

如果每个字符的**使用频率相等**，那么等长编码无疑是空间效率最高的编码方法，而如果字符出现的频率不同，则可以让频率高的字符采用尽可能短的编码，频率低的字符采用尽可能长的编码，来构造出一种**不等长编码**，从而获得更好的空间效率。

在设计不等长编码时，要考虑解码的唯一性，如果一组编码中任一编码都不是其他任何一个编码的前缀，那么称这组编码为**前缀编码**，其保证了编码被解码时的唯一性。

霍夫曼树可用于构造**最短的前缀编码**，即**霍夫曼编码（Huffman Code）**，其构造步骤如下：

1. 设需要编码的字符集为：$d_1, d_2, \dots, d_n$，他们在字符串中出现的频率为：$w_1, w_2, \dots, w_n$。
2. 以 $d_1, d_2, \dots, d_n$ 作为叶结点，$w_1, w_2, \dots, w_n$ 作为叶结点的权值，构造一棵霍夫曼树。
3. 规定哈夫曼编码树的左分支代表 0，右分支代表 1，则从根结点到每个叶结点所经过的路径组成的 0、1 序列即为该叶结点对应字符的编码。



字符	频率	编码
A	35	11
B	25	00
C	15	01
D	15	101
E	10	100

图 10.228

示例代码

" 霍夫曼树的构建"

```c
typedef struct HNode {
  int weight;
  HNode *lchild, *rchild;
} * Htree;

Htree createHuffmanTree(int arr[], int n) {
  Htree forest[N];
  Htree root = NULL;
  for (int i = 0; i < n; i++) {  // 将所有点存入森林
    Htree temp;
    temp = (Htree)malloc(sizeof(HNode));
    temp->weight = arr[i];
    temp->lchild = temp->rchild = NULL;
```

```
      forest[i] = temp;
    }

    for (int i = 1; i < n; i++) {  // n-1 次循环建哈夫曼树
      int minn = -1, minnSub;  // minn 为最小值树根下标，minnsub 为次小值树根下标
      for (int j = 0; j < n; j++) {
        if (forest[j] != NULL && minn == -1) {
          minn = j;
          continue;
        }
        if (forest[j] != NULL) {
          minnSub = j;
          break;
        }
      }

      for (int j = minnSub; j < n; j++) {  // 根据 minn 与 minnSub 赋值
        if (forest[j] != NULL) {
          if (forest[j]->weight < forest[minn]->weight) {
            minnSub = minn;
            minn = j;
          } else if (forest[j]->weight < forest[minnSub]->weight) {
            minnSub = j;
          }
        }
      }

      // 建新树
      root = (Htree)malloc(sizeof(HNode));
      root->weight = forest[minn]->weight + forest[minnSub]->weight;
      root->lchild = forest[minn];
      root->rchild = forest[minnSub];

      forest[minn] = root;     // 指向新树的指针赋给 minn 位置
      forest[minnSub] = NULL;  // minnSub 位置为空
    }
    return root;
  }
```

" 计算构成霍夫曼树的 WPL"

```
typedef struct HNode {
  int weight;
  HNode *lchild, *rchild;
} * Htree;

int getWPL(Htree root, int len) {  // 递归实现，对于已经建好的霍夫曼树，求 WPL
  if (root == NULL)
    return 0;
  else {
    if (root->lchild == NULL && root->rchild == NULL)  // 叶节点
      return root->weight * len;
    else {
```

```
      int left = getWPL(root->lchild, len + 1);
      int right = getWPL(root->rchild, len + 1);
      return left + right;
    }
  }
}
```

" 对于未建好的霍夫曼树，直接求其 WPL"

```
int getWPL(int arr[], int n) {  // 对于未建好的霍夫曼树，直接求其 WPL
  priority_queue<int, vector<int>, greater<int>> huffman;  // 小根堆
  for (int i = 0; i < n; i++) huffman.push(arr[i]);

  int res = 0;
  for (int i = 0; i < n - 1; i++) {
    int x = huffman.top();
    huffman.pop();
    int y = huffman.top();
    huffman.pop();
    int temp = x + y;
    res += temp;
    huffman.push(temp);
  }
  return res;
}
```

" 对于给定序列，计算霍夫曼编码"

```
typedef struct HNode {
  int weight;
  HNode *lchild, *rchild;
} * Htree;

void huffmanCoding(Htree root, int len, int arr[]) {  // 计算霍夫曼编码
  if (root != NULL) {
    if (root->lchild == NULL && root->rchild == NULL) {
      printf(" 结点为 %d 的字符的编码为: ", root->weight);
      for (int i = 0; i < len; i++) printf("%d", arr[i]);
      printf("\n");
    } else {
      arr[len] = 0;
      huffmanCoding(root->lchild, len + 1, arr);
      arr[len] = 1;
      huffmanCoding(root->rchild, len + 1, arr);
    }
  }
}
```

第 11 章

图论

11.1 图论部分简介

图论 (Graph theory) 是数学的一个分支，图是图论的主要研究对象。**图 (Graph)** 是由若干给定的顶点及连接两顶点的边所构成的图形，这种图形通常用来描述某些事物之间的某种特定关系。顶点用于代表事物，连接两顶点的边则用于表示两个事物间具有这种关系。

参考资料

图论 - 维基百科， 自由的百科全书[1]

参考资料与注释

[1] 图论 - 维基百科，自由的百科全书

11.2 图论相关概念

本页面概述了图论中的一些概念，这些概念并不全是在 OI 中常见的，对于 OIer 来说，只需掌握本页面中的基础部分即可，如果在学习中碰到了不懂的概念，可以再来查阅。

> warning
>
> 图论相关定义在不同教材中往往会有所不同，遇到的时候需根据上下文加以判断。

图

图 (graph) 是一个二元组 $G=(V(G),E(G))$。其中 $V(G)$ 是非空集，称为**点集 (vertex set)**，对于 V 中的每个元素，我们称其为**顶点 (vertex)** 或**节点 (node)**，简称**点**；$E(G)$ 为 $V(G)$ 各结点之间边的集合，称为**边集 (edge set)**。

常用 $G=(V,E)$ 表示图。

当 V,E 都是有限集合时，称 G 为**有限图**。

当 V 或 E 是无限集合时，称 G 为**无限图**。

图有多种，包括**无向图 (undirected graph)**，**有向图 (directed graph)**，**混合图 (mixed graph)** 等。

若 G 为无向图，则 E 中的每个元素为一个无序二元组 (u,v)，称作**无向边 (undirected edge)**，简称**边 (edge)**，其中 $u,v \in V$。设 $e=(u,v)$，则 u 和 v 称为 e 的**端点 (endpoint)**。

若 G 为有向图，则 E 中的每一个元素为一个有序二元组 (u,v)，有时也写作 $u \to v$，称作**有向边 (directed edge)** 或**弧 (arc)**，在不引起混淆的情况下也可以称作**边 (edge)**。设 $e=u \to v$，则此时 u 称为 e 的**起点 (tail)**，v 称为 e 的**终点 (head)**，起点和终点也称为 e 的**端点 (endpoint)**。并称 u 是 v 的直接前驱，v 是 u 的直接后继。

" **为什么起点是** tail，**终点是** head？"

边通常用箭头表示，而箭头是从「尾」指向「头」的。

若 G 为混合图，则 E 中既有**有向边**，又有**无向边**。

若 G 的每条边 $e_k=(u_k,v_k)$ 都被赋予一个数作为该边的**权**，则称 G 为**赋权图**。如果这些权都是正实数，就称 G 为**正权图**。

图 G 的点数 $|V(G)|$ 也被称作图 G 的**阶 (order)**。

形象地说，图是由若干点以及连接点与点的边构成的。

相邻

在无向图 $G=(V,E)$ 中，若点 v 是边 e 的一个端点，则称 v 和 e 是**关联的 (incident)** 或**相邻的 (adjacent)**。对于两顶点 u 和 v，若存在边 (u,v)，则称 u 和 v 是**相邻的 (adjacent)**。

一个顶点 $v \in V$ 的**邻域 (neighborhood)** 是所有与之相邻的顶点所构成的集合，记作 $N(v)$。

一个点集 S 的邻域是所有与 S 中至少一个点相邻的点所构成的集合，记作 $N(S)$，即：

$$N(S)=\bigcup_{v\in S} N(v)$$

简单图

自环 (loop)：对 E 中的边 $e=(u,v)$，若 $u=v$，则 e 被称作一个自环。

重边 (multiple edge)：若 E 中存在两个完全相同的元素（边）e_1,e_2，则它们被称作（一组）重边。

简单图 (simple graph)：若一个图中没有自环和重边，它被称为简单图。具有至少两个顶点的简单无向图中一定存在度相同的结点。(鸽巢原理)

如果一张图中有自环或重边，则称它为**多重图 (multigraph)**。

warning

在无向图中 (u,v) 和 (v,u) 算一组重边，而在有向图中，$u \to v$ 和 $v \to u$ 不为重边。

warning

在题目中，如果没有特殊说明，是可以存在自环和重边的，在做题时需特殊考虑。

度数

与一个顶点 v 关联的边的条数称作该顶点的**度 (degree)**，记作 $d(v)$。特别地，对于边 (v,v)，则每条这样的边要对 $d(v)$ 产生 2 的贡献。

对于无向简单图，有 $d(v)=|N(v)|$。

握手定理（又称图论基本定理）：对于任何无向图 $G=(V,E)$，有 $\sum_{v\in V} d(v)=2|E|$。

推论：在任意图中，度数为奇数的点必然有偶数个。

若 $d(v)=0$，则称 v 为**孤立点 (isolated vertex)**。

若 $d(v)=1$，则称 v 为**叶节点 (leaf vertex)/悬挂点 (pendant vertex)**。

若 $2 \mid d(v)$，则称 v 为**偶点** (**even vertex**)。

若 $2 \nmid d(v)$，则称 v 为**奇点** (**odd vertex**)。图中奇点的个数是偶数。

若 $d(v) = |V| - 1$，则称 v 为**支配点** (**universal vertex**)。

对一张图，所有节点的度数的最小值称为 G 的**最小度** (**minimum degree**)，记作 $\delta(G)$；最大值称为**最大度** (**maximum degree**)，记作 $\Delta(G)$。即：$\delta(G) = \min_{v\in G} d(v)$，$\Delta(G) = \max_{v\in G} d(v)$。

在有向图 $G = (V, E)$ 中，以一个顶点 v 为起点的边的条数称为该顶点的**出度** (**out-degree**)，记作 $d^+(v)$。以一个顶点 v 为终点的边的条数称为该节点的**入度** (**in-degree**)，记作 $d^-(v)$。显然 $d^+(v) + d^-(v) = d(v)$。

对于任何有向图 $G = (V, E)$，有：

$$\sum_{v\in V} d^+(v) = \sum_{v\in V} d^-(v) = |E|$$

若对一张无向图 $G = (V, E)$，每个顶点的度数都是一个固定的常数 k，则称 G 为**k- 正则图** (**k-regular graph**)。

如果给定一个序列 a，可以找到一个图 G，以其为度数列，则称 a 是**可图化**的。

如果给定一个序列 a，可以找到一个简单图 G，以其为度数列，则称 a 是**可简单图化**的。

路径

途径 (**walk**)：途径是连接一连串顶点的边的序列，可以为有限或无限长度。形式化地说，一条有限途径 w 是一个边的序列 $e_1, e_2, \ldots, e_k$，使得存在一个顶点序列 $v_0, v_1, \ldots, v_k$ 满足 $e_i = (v_{i-1}, v_i)$，其中 $i \in [1, k]$。这样的途径可以简写为 $v_0 \to v_1 \to v_2 \to \cdots \to v_k$。通常来说，边的数量 k 被称作这条途径的**长度**（如果边是带权的，长度通常指途径上的边权之和，题目中也可能另有定义）。

迹 (**trail**)：对于一条途径 w，若 $e_1, e_2, \ldots, e_k$ 两两互不相同，则称 w 是一条迹。

路径 (**path**)（又称**简单路径** (**simple path**)）：对于一条迹 w，若其连接的点的序列中点两两不同，则称 w 是一条路径。

回路 (**circuit**)：对于一条迹 w，若 $v_0 = v_k$，则称 w 是一条回路。

环/圈 (**cycle**)（又称**简单回路/简单环** (**simple circuit**)）：对于一条回路 w，若 $v_0 = v_k$ 是点序列中唯一重复出现的点对，则称 w 是一个环。

warning

关于路径的定义在不同地方可能有所不同，如，「路径」可能指本文中的「途径」，「环」可能指本文中的「回路」。如果在题目中看到类似的词汇，且没有「简单路径」/「非简单路径」（即本文中的「途径」）等特殊说明，最好询问一下具体指什么。

子图

对一张图 $G = (V, E)$，若存在另一张图 $H = (V', E')$ 满足 $V' \subseteq V$ 且 $E' \subseteq E$，则称 H 是 G 的**子图** (**subgraph**)，记作 $H \subseteq G$。

若对 $H \subseteq G$，满足 $\forall u, v \in V'$，只要 $(u, v) \in E$，均有 $(u, v) \in E'$，则称 H 是 G 的**导出子图/诱导子图** (**induced subgraph**)。

容易发现，一个图的导出子图仅由子图的点集决定，因此点集为 $V'(V' \subseteq V)$ 的导出子图称为 V' 导出的子图，记作 $G[V']$。

若 $H \subseteq G$ 满足 $V' = V$，则称 H 为 G 的**生成子图/支撑子图** (**spanning subgraph**)。

显然，G 是自身的子图，支撑子图，导出子图；无边图是 G 的支撑子图。原图 G 和无边图都是 G 的平凡子图。

如果一张无向图 G 的某个生成子图 F 为 k- 正则图，则称 F 为 G 的一个**k- 因子** (**k-factor**)。

如果有向图 $G = (V, E)$ 的导出子图 $H = G[V^*]$ 满足 $\forall v \in V^*, (v, u) \in E$，有 $u \in V^*$，则称 H 为 G 的一个**闭合子图** (**closed subgraph**)。

连通

无向图

对于一张无向图 $G=(V,E)$，对于 $u,v\in V$，若存在一条途径使得 $v_0=u,v_k=v$，则称 u 和 v 是**连通的 (connected)**。由定义，任意一个顶点和自身连通，任意一条边的两个端点连通。

若无向图 $G=(V,E)$，满足其中任意两个顶点均连通，则称 G 是**连通图 (connected graph)**，G 的这一性质称作**连通性 (connectivity)**。

若 H 是 G 的一个连通子图，且不存在 F 满足 $H\subsetneq F\subseteq G$ 且 F 为连通图，则 H 是 G 的一个**连通块/连通分量 (connected component)**（极大连通子图）。

有向图

对于一张有向图 $G=(V,E)$，对于 $u,v\in V$，若存在一条途径使得 $v_0=u,v_k=v$，则称 u **可达** v。由定义，任意一个顶点可达自身，任意一条边的起点可达终点。（无向图中的连通也可以视作双向可达。）

若一张有向图的节点两两互相可达，则称这张图是**强连通的 (strongly connected)**。

若一张有向图的边替换为无向边后可以得到一张连通图，则称原来这张有向图是**弱连通的 (weakly connected)**。

与连通分量类似，也有**弱连通分量 (weakly connected component)**（极大弱连通子图）和**强连通分量 (strongly connected component)**（极大强连通子图）。

相关算法请参见 强连通分量。

割

相关算法请参见 割点和桥 以及 双连通分量。

在本部分中，有向图的「连通」一般指「强连通」。

对于连通图 $G=(V,E)$，若 $V'\subseteq V$ 且 $G[V\ \ V']$（即从 G 中删去 V' 中的点）不是连通图，则 V' 是图 G 的一个**点割集 (vertex cut/separating set)**。大小为一的点割集又被称作**割点 (cut vertex)**。

对于连通图 $G=(V,E)$ 和整数 k，若 $|V|\geq k+1$ 且 G 不存在大小为 $k-1$ 的点割集，则称图 G 是**k- 点连通的 (k-vertex-connected)**，而使得上式成立的最大的 k 被称作图 G 的**点连通度 (vertex connectivity)**，记作 $\kappa(G)$。（对于非完全图，点连通度即为最小点割集的大小，而完全图 K_n 的点连通度为 $n-1$。）

对于图 $G=(V,E)$ 以及 $u,v\in V$ 满足 $u\neq v$，u 和 v 不相邻，u 可达 v，若 $V'\subseteq V$，$u,v\notin V'$，且在 $G[V\ \ V']$ 中 u 和 v 不连通，则 V' 被称作 u 到 v 的点割集。u 到 v 的最小点割集的大小被称作 u 到 v 的**局部点连通度 (local connectivity)**，记作 $\kappa(u,v)$。

还可以在边上作类似的定义：

对于连通图 $G=(V,E)$，若 $E'\subseteq E$ 且 $G'=(V,E\ \ E')$（即从 G 中删去 E' 中的边）不是连通图，则 E' 是图 G 的一个**边割集 (edge cut)**。大小为一的边割集又被称作**桥 (bridge)**。

对于连通图 $G=(V,E)$ 和整数 k，若 G 不存在大小为 $k-1$ 的边割集，则称图 G 是**k- 边连通的 (k-edge-connected)**，而使得上式成立的最大的 k 被称作图 G 的**边连通度 (edge connectivity)**，记作 $\lambda(G)$。（对于任何图，边连通度即为最小边割集的大小。）

对于图 $G=(V,E)$ 以及 $u,v\in V$ 满足 $u\neq v$，u 可达 v，若 $E'\subseteq E$，且在 $G'=(V,E\ \ E')$ 中 u 和 v 不连通，则 E' 被称作 u 到 v 的边割集。u 到 v 的最小边割集的大小被称作 u 到 v 的**局部边连通度 (local edge-connectivity)**，记作 $\lambda(u,v)$。

点双连通 (biconnected) 几乎与 2- 点连通完全一致，除了一条边连接两个点构成的图，它是点双连通的，但不是 2- 点连通的。换句话说，没有割点的连通图是点双连通的。

边双连通 (2-edge-connected) 与 2- 边双连通完全一致。换句话说，没有桥的连通图是边双连通的。

与连通分量类似，也有**点双连通分量 (biconnected component)**（极大点双连通子图）和**边双连通分量 (2-edge-connected component)**（极大边双连通子图）。

Whitney 定理：对任意的图 G，有 $\kappa(G) \le \lambda(G) \le \delta(G)$。（不等式中的三项分别为点连通度、边连通度、最小度。）

稀疏图/稠密图

若一张图的边数远小于其点数的平方，那么它是一张**稀疏图 (sparse graph)**。

若一张图的边数接近其点数的平方，那么它是一张**稠密图 (dense graph)**。

这两个概念并没有严格的定义，一般用于讨论 时间复杂度 为 $O(|V|^2)$ 的算法与 $O(|E|)$ 的算法的效率差异（在稠密图上这两种算法效率相当，而在稀疏图上 $O(|E|)$ 的算法效率明显更高）。

补图

对于无向简单图 $G=(V,E)$，它的**补图 (complement graph)** 指的是这样的一张图：记作 $\bar{G}$，满足 $V\left(\bar{G}\right)=V\left(G\right)$，且对任意节点对 (u,v)，$(u,v)\in E\left(\bar{G}\right)$ 当且仅当 $(u,v)\notin E\left(G\right)$。

反图

对于有向图 $G=(V,E)$，它的**反图 (transpose graph)** 指的是点集不变，每条边反向得到的图，即：若 G 的反图为 $G'=(V,E')$，则 $E'=\{(v,u)|(u,v)\in E\}$。

特殊的图

若无向简单图 G 满足任意不同两点间均有边，则称 G 为**完全图 (complete graph)**，n 阶完全图记作 K_n。若有向图 G 满足任意不同两点间都有两条方向不同的边，则称 G 为**有向完全图 (complete digraph)**。

边集为空的图称为**无边图 (edgeless graph)**、**空图 (empty graph)** 或**零图 (null graph)**，n 阶无边图记作 $\overline{K}_n$ 或 N_n。N_n 与 K_n 互为补图。

warning

零图 (null graph) 也可指**零阶图 (order-zero graph)** K_0，即点集与边集均为空的图。

若有向简单图 G 满足任意不同两点间都有恰好一条边（单向），则称 G 为**竞赛图 (tournament graph)**。

若无向简单图 $G=(V,E)$ 的所有边恰好构成一个圈，则称 G 为**环图/圈图 (cycle graph)**，$n(n\ge 3)$ 阶圈图记作 C_n。易知，一张图为圈图的充分必要条件是，它是 2- 正则连通图。

若无向简单图 $G=(V,E)$ 满足，存在一个点 v 为支配点，其余点之间没有边相连，则称 G 为**星图/菊花图 (star graph)**，$n+1(n\ge 1)$ 阶星图记作 S_n。

若无向简单图 $G=(V,E)$ 满足，存在一个点 v 为支配点，其它点之间构成一个圈，则称 G 为**轮图 (wheel graph)**，$n+1(n\ge 3)$ 阶轮图记作 W_n。

若无向简单图 $G=(V,E)$ 的所有边恰好构成一条简单路径，则称 G 为**链 (chain/path graph)**，n 阶的链记作 P_n。易知，一条链由一个圈图删去一条边而得。

如果一张无向连通图不含环，则称它是一棵**树 (tree)**。相关内容详见 树基础。

如果一张无向连通图包含恰好一个环，则称它是一棵**基环树 (pseudotree)**。

如果一张有向弱连通图每个点的入度都为 1，则称它是一棵**基环外向树**。

如果一张有向弱连通图每个点的出度都为 1，则称它是一棵**基环内向树**。

多棵树可以组成一个**森林 (forest)**，多棵基环树可以组成**基环森林 (pseudoforest)**，多棵基环外向树可以组成**基环外向树森林**，多棵基环内向树可以组成**基环内向森林 (functional graph)**。

如果一张无向连通图的每条边最多在一个环内，则称它是一棵**仙人掌 (cactus)**。多棵仙人掌可以组成**沙漠**。

如果一张图的点集可以被分为两部分，每一部分的内部都没有连边，那么这张图是一张**二分图 (bipartite graph)**。如果二分图中任何两个不在同一部分的点之间都有连边，那么这张图是一张**完全二分图 (complete bipartite graph/biclique)**，一张两部分分别有 n 个点和 m 个点的完全二分图记作 $K_{n,m}$。相关内容详见 二分图。

如果一张图可以画在一个平面上，且没有两条边在非端点处相交，那么这张图是一张**平面图 (planar graph)**。一张图的任何子图都不是 K_5 或 $K_{3,3}$ 是其为一张平面图的充要条件。对于简单连通平面图 $G=(V,E)$ 且 $V \geq 3$，$|E| \leq 3|V|-6$。

同构

两个图 G 和 H，如果存在一个双射 $f: V(G) \to V(H)$，且满足 $(u,v) \in E(G)$，当且仅当 $(f(u), f(v)) \in E(H)$，则我们称 f 为 G 到 H 的一个**同构 (isomorphism)**，且图 G 与图 H 是**同构的 (isomorphic)**，记作 $G \cong H$。

从定义可知，若 $G \cong H$，必须满足：

- $|V(G)| = |V(H)|, |E(G)| = |E(H)|$
- G 和 H 结点度的非增序列相同
- G 和 H 存在同构的导出子图

无向简单图的二元运算

对于无向简单图，我们可以定义如下二元运算：

交 (intersection)：图 $G=(V_1,E_1), H=(V_2,E_2)$ 的交定义成图 $G \cap H = (V_1 \cap V_2, E_1 \cap E_2)$。

容易证明两个无向简单图的交还是无向简单图。

并 (union)：图 $G=(V_1,E_1), H=(V_2,E_2)$ 的并定义成图 $G \cup H = (V_1 \cup V_2, E_1 \cup E_2)$。

和 (sum)/直和 (direct sum)：对于 $G=(V_1,E_1), H=(V_2,E_2)$，任意构造 $H' \cong H$ 使得 $V(H') \cap V_1 = \emptyset$($H'$ 可以等于 H)。此时与 $G \cup H'$ 同构的任何图称为 G 和 H 的和/直和/不交并，记作 $G+H$ 或 $G \oplus H$。

若 G 与 H 的点集本身不相交，则 $G \cup H = G+H$。

比如，森林可以定义成若干棵树的和。

> " 并与和的区别"
>
> 可以理解为，「并」会让两张图中「名字相同」的点、边合并，而「和」则不会。

特殊的点集/边集

支配集

对于无向图 $G=(V,E)$，若 $V' \subseteq V$ 且 $\forall v \in (V\ \ V')$ 存在边 $(u,v) \in E$ 满足 $u \in V'$，则 V' 是图 G 的一个**支配集 (dominating set)**。

无向图 G 最小的支配集的大小记作 $\gamma(G)$。求一张图的最小支配集是 NP 困难 的。

对于有向图 $G=(V,E)$，若 $V' \subseteq V$ 且 $\forall v \in (V\ \ V')$ 存在边 $(u,v) \in E$ 满足 $u \in V'$，则 V' 是图 G 的一个**出 - 支配集 (out-dominating set)**。类似地，可以定义有向图的**入 - 支配集 (in-dominating set)**。

有向图 G 最小的出 - 支配集大小记作 $\gamma^+(G)$，最小的入 - 支配集大小记作 $\gamma^-(G)$。

边支配集

对于图 $G=(V,E)$，若 $E' \subseteq E$ 且 $\forall e \in (E\ \ E')$ 存在 E' 中的边与其有公共点，则称 E' 是图 G 的一个**边支配集 (edge dominating set)**。

求一张图的最小边支配集是 NP 困难 的。

独立集

对于图 $G=(V,E)$，若 $V' \subseteq V$ 且 V' 中任意两点都不相邻，则 V' 是图 G 的一个**独立集** (**independent set**)。图 G 最大的独立集的大小记作 $\alpha(G)$。求一张图的最大独立集是 NP 困难 的。

匹配

对于图 $G=(V,E)$，若 $E' \in E$ 且 E' 中任意两条不同的边都没有公共的端点，且 E' 中任意一条边都不是自环，则 E' 是图 G 的一个**匹配** (**matching**)，也可以叫作**边独立集** (**independent edge set**)。如果一个点是匹配中某条边的一个端点，则称这个点是**被匹配的** (**matched**)/**饱和的** (**saturated**)，否则称这个点是**不被匹配的** (**unmatched**)。

边数最多的匹配被称作一张图的**最大匹配** (**maximum-cardinality matching**)。图 G 的最大匹配的大小记作 $\nu(G)$。

如果边带权，那么权重之和最大的匹配被称作一张图的**最大权匹配** (**maximum-weight matching**)。

如果一个匹配在加入任何一条边后都不再是一个匹配，那么这个匹配是一个**极大匹配** (**maximal matching**)。最大的极大匹配就是最大匹配，任何最大匹配都是极大匹配。极大匹配一定是边支配集，但边支配集不一定是匹配。最小极大匹配和最小边支配集大小相等，但最小边支配集不一定是匹配。求最小极大匹配是 NP 困难的。

如果在一个匹配中所有点都是被匹配的，那么这个匹配是一个**完美匹配** (**perfect matching**)。如果在一个匹配中只有一个点不被匹配，那么这个匹配是一个**准完美匹配** (**near-perfect matching**)。

求一张普通图或二分图的匹配或完美匹配个数都是 #P 完全 的。

对于一个匹配 M，若一条路径以非匹配点为起点，每相邻两条边的其中一条在匹配中而另一条不在匹配中，则这条路径被称作一条**交替路径** (**alternating path**)；一条在非匹配点终止的交替路径，被称作一条**增广路径** (**augmenting path**)。

托特定理：n 阶无向图 G 有完美匹配当且仅当对于任意的 $V' \subset V(G)$，$p_{\text{odd}}(G-V') \le |V'|$，其中 p_{odd} 表示奇数阶连通分支数。

托特定理（推论）：任何无桥 3 - 正则图都有完美匹配。

点覆盖

对于图 $G=(V,E)$，若 $V' \subseteq V$ 且 $\forall e \in E$ 满足 e 的至少一个端点在 V' 中，则称 V' 是图 G 的一个**点覆盖** (**vertex cover**)。

点覆盖集必为支配集，但极小点覆盖集不一定是极小支配集。

一个点集是点覆盖的充要条件是其补集是独立集，因此最小点覆盖的补集是最大独立集。求一张图的最小点覆盖是 NP 困难 的。

一张图的任何一个匹配的大小都不超过其任何一个点覆盖的大小。完全二分图 $K_{n,m}$ 的最大匹配和最小点覆盖大小都为 $\min(n,m)$。

边覆盖

对于图 $G=(V,E)$，若 $E' \subseteq E$ 且 $\forall v \in V$ 满足 v 与 E' 中的至少一条边相邻，则称 E' 是图 G 的一个**边覆盖** (**edge cover**)。

最小边覆盖的大小记作 $\rho(G)$，可以由最大匹配贪心扩展求得：对于所有非匹配点，将其一条邻边加入最大匹配中，即得到了一个最小边覆盖。

最大匹配也可以由最小边覆盖求得：对于最小边覆盖中每对有公共点的边删去其中一条。

一张图的最小边覆盖的大小加上最大匹配的大小等于图的点数，即 $\rho(G)+\nu(G)=|V(G)|$。

一张图的最大匹配的大小不超过最小边覆盖的大小，即 $\nu(G) \le \rho(G)$。特别地，完美匹配一定是一个最小边覆盖，这也是上式取到等号的唯一情况。

一张图的任何一个独立集的大小都不超过其任何一个边覆盖的大小。完全二分图 $K_{n,m}$ 的最大独立集和最小边覆盖大小都为 $\max(n,m)$。

团

对于图 $G=(V,E)$，若 $V'\subseteq V$ 且 V' 中任意两个不同的顶点都相邻，则 V' 是图 G 的一个**团 (clique)**。团的导出子图是完全图。

如果一个团在加入任何一个顶点后都不再是一个团，则这个团是一个**极大团 (maximal clique)**。

一张图的最大团的大小记作 $\omega(G)$，最大团的大小等于其补图最大独立集的大小，即 $\omega(G)=\alpha(\bar{G})$。求一张图的最大团是 NP 困难 的。

参考资料

OI 中转站 - 图论概念梳理[1]

Wikipedia[2]（以及相关概念的对应词条）

离散数学（修订版），田文成周禄新编著，天津文学出版社，P184-187

戴一奇，胡冠章，陈卫。图论与代数结构 [M]. 北京：清华大学出版社，1995.

参考资料与注释

[1] OI 中转站 - 图论概念梳理

[2] Wikipedia

11.3 图的存储

Authors: Ir1d, sshwy, Xeonacid, partychicken, Anguei, HeRaNO 在 OI 中，想要对图进行操作，就需要先学习图的存储方式。

约定

本文默认读者已阅读并了解了 图论相关概念 中的基础内容，如果在阅读中遇到困难，也可以在 图论相关概念 中进行查阅。

在本文中，用 n 代指图的点数，用 m 代指图的边数，用 $d^+(u)$ 代指点 u 的出度，即以 u 为出发点的边数。

直接存边

方法

使用一个数组来存边，数组中的每个元素都包含一条边的起点与终点（带边权的图还包含边权）。（或者使用多个数组分别存起点，终点和边权。）

" 参考代码"

```cpp
#include <iostream>
#include <vector>

using namespace std;

struct Edge {
```

```
  int u, v;
};

int n, m;
vector<Edge> e;
vector<bool> vis;

bool find_edge(int u, int v) {
  for (int i = 1; i <= m; ++i) {
    if (e[i].u == u && e[i].v == v) {
      return true;
    }
  }
  return false;
}

void dfs(int u) {
  if (vis[u]) return;
  vis[u] = true;
  for (int i = 1; i <= m; ++i) {
    if (e[i].u == u) {
      dfs(e[i].v);
    }
  }
}

int main() {
  cin >> n >> m;

  vis.resize(n + 1, false);
  e.resize(m + 1);

  for (int i = 1; i <= m; ++i) cin >> e[i].u >> e[i].v;

  return 0;
}

class Edge:
    def __init__(self, u = 0, v = 0):
        self.u = u
        self.v = v

n, m = map(int, input().split())

e = [Edge() for _ in range(m)]; vis = [False] * n

for i in range(m):
    e[i].u, e[i].v = map(int, input().split())

def find_edge(u, v):
    for i in range(m):
        if e[i].u == u and e[i].v == v:
            return True
    return False
```

```python
def dfs(u):
    if vis[u]:
        return
    vis[u] = True
    for i in range(m):
        if e[i].u == u:
            dfs(e[i].v)
```

复杂度

查询是否存在某条边：$O(m)$。

遍历一个点的所有出边：$O(m)$。

遍历整张图：$O(nm)$。

空间复杂度：$O(m)$。

应用

由于直接存边的遍历效率低下，一般不用于遍历图。

在 Kruskal 算法 中，由于需要将边按边权排序，需要直接存边。

在有的题目中，需要多次建图（如建一遍原图，建一遍反图），此时既可以使用多个其它数据结构来同时存储多张图，也可以将边直接存下来，需要重新建图时利用直接存下的边来建图。

邻接矩阵

方法

使用一个二维数组 `adj` 来存边，其中 `adj[u][v]` 为 1 表示存在 u 到 v 的边，为 0 表示不存在。如果是带边权的图，可以在 `adj[u][v]` 中存储 u 到 v 的边的边权。

" 参考代码"

```cpp
#include <iostream>
#include <vector>

using namespace std;

int n, m;
vector<bool> vis;
vector<vector<bool> > adj;

bool find_edge(int u, int v) { return adj[u][v]; }

void dfs(int u) {
  if (vis[u]) return;
  vis[u] = true;
  for (int v = 1; v <= n; ++v) {
    if (adj[u][v]) {
      dfs(v);
    }
  }
}
```

```cpp
int main() {
  cin >> n >> m;

  vis.resize(n + 1, false);
  adj.resize(n + 1, vector<bool>(n + 1, false));

  for (int i = 1; i <= m; ++i) {
    int u, v;
    cin >> u >> v;
    adj[u][v] = true;
  }

  return 0;
}
```

```python
vis = [False] * (n + 1)
adj = [[False] * (n + 1) for _ in range(n + 1)]

for i in range(1, m + 1):
    u, v = map(lambda x:int(x), input().split())
    adj[u][v] = True

def find_edge(u, v):
    return adj[u][v]

def dfs(u):
    if vis[u]:
        return
    vis[u] = True
    for v in range(1, n + 1):
        if adj[u][v]:
            dfs(v)
```

复杂度

查询是否存在某条边：$O(1)$。

遍历一个点的所有出边：$O(n)$。

遍历整张图：$O(n^2)$。

空间复杂度：$O(n^2)$。

应用

邻接矩阵只适用于没有重边（或重边可以忽略）的情况。

其最显著的优点是可以 $O(1)$ 查询一条边是否存在。

由于邻接矩阵在稀疏图上效率很低（尤其是在点数较多的图上，空间无法承受），所以一般只会在稠密图上使用邻接矩阵。

邻接表

方法

使用一个支持动态增加元素的数据结构构成的数组，如 `vector<int> adj[n + 1]` 来存边，其中 `adj[u]` 存储的是点 u 的所有出边的相关信息（终点、边权等）。

" 参考代码"

```cpp
#include <iostream>
#include <vector>

using namespace std;

int n, m;
vector<bool> vis;
vector<vector<int> > adj;

bool find_edge(int u, int v) {
  for (int i = 0; i < adj[u].size(); ++i) {
    if (adj[u][i] == v) {
      return true;
    }
  }
  return false;
}

void dfs(int u) {
  if (vis[u]) return;
  vis[u] = true;
  for (int i = 0; i < adj[u].size(); ++i) dfs(adj[u][i]);
}

int main() {
  cin >> n >> m;

  vis.resize(n + 1, false);
  adj.resize(n + 1);

  for (int i = 1; i <= m; ++i) {
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
  }

  return 0;
}
```

```python
vis = [False] * (n + 1)
adj = [[] for _ in range(n + 1)]

for i in range(1, m + 1):
    u, v = map(lambda x:int(x), input().split())
    adj[u].append(v)
```

```python
def find_edge(u, v):
    for i in range(0, len(adj[u])):
        if adj[u][i] == v:
            return True
    return False

def dfs(u):
    if vis[u]:
        return
    vis[u] = True
    for i in range(0, len(adj[u])):
        dfs(adj[u][i])
```

复杂度

查询是否存在 u 到 v 的边：$O(d^+(u))$（如果事先进行了排序就可以使用 二分查找 做到 $O(\log(d^+(u)))$）。

遍历点 u 的所有出边：$O(d^+(u))$。

遍历整张图：$O(n+m)$。

空间复杂度：$O(m)$。

应用

存各种图都很适合，除非有特殊需求（如需要快速查询一条边是否存在，且点数较少，可以使用邻接矩阵）。

尤其适用于需要对一个点的所有出边进行排序的场合。

链式前向星

方法

本质上是用链表实现的邻接表，核心代码如下：

```cpp
// head[u] 和 cnt 的初始值都为 -1
void add(int u, int v) {
  nxt[++cnt] = head[u];  // 当前边的后继
  head[u] = cnt;         // 起点 u 的第一条边
  to[cnt] = v;           // 当前边的终点
}

// 遍历 u 的出边
for (int i = head[u]; ~i; i = nxt[i]) {  // ~i 表示 i != -1
  int v = to[i];
}
```

```python
# head[u] 和 cnt 的初始值都为 -1
def add(u, v):
    cnt = cnt + 1
    nex[cnt] = head[u] # 当前边的后继
    head[u] = cnt # 起点 u 的第一条边
    to[cnt] = v # 当前边的终点

# 遍历 u 的出边
```

```python
i = head[u]
while ~i: # ~i 表示 i != -1
    v = to[i]
    i = nxt[i]
```

" 参考代码"

```cpp
#include <iostream>
#include <vector>

using namespace std;

int n, m;
vector<bool> vis;
vector<int> head, nxt, to;

void add(int u, int v) {
  nxt.push_back(head[u]);
  head[u] = to.size();
  to.push_back(v);
}

bool find_edge(int u, int v) {
  for (int i = head[u]; ~i; i = nxt[i]) {  // ~i 表示 i != -1
    if (to[i] == v) {
      return true;
    }
  }
  return false;
}

void dfs(int u) {
  if (vis[u]) return;
  vis[u] = true;
  for (int i = head[u]; ~i; i = nxt[i]) dfs(to[i]);
}

int main() {
  cin >> n >> m;

  vis.resize(n + 1, false);
  head.resize(n + 1, -1);

  for (int i = 1; i <= m; ++i) {
    int u, v;
    cin >> u >> v;
    add(u, v);
  }

  return 0;
}
```

复杂度

查询是否存在 u 到 v 的边：$O(d^+(u))$。

遍历点 u 的所有出边：$O(d^+(u))$。

遍历整张图：$O(n+m)$。

空间复杂度：$O(m)$。

应用

存各种图都很适合，但不能快速查询一条边是否存在，也不能方便地对一个点的出边进行排序。

优点是边是带编号的，有时会非常有用，而且如果 `cnt` 的初始值为奇数，存双向边时 `i ^ 1` 即是 `i` 的反边（常用于 网络流）。

11.4 DFS（图论）

> **Authors:** Ir1d, greyqz, yjl9903, partychicken, ChungZH, qq1010903229, Marcythm, Acfboy, shenshuaijie, Craneplayz

引入

DFS 全称是 Depth First Search[1]，中文名是深度优先搜索，是一种用于遍历或搜索树或图的算法。所谓深度优先，就是说每次都尝试向更深的节点走。

该算法讲解时常常与 BFS 并列，但两者除了都能遍历图的连通块以外，用途完全不同，很少有能混用两种算法的情况。

DFS 常常用来指代用递归函数实现的搜索，但实际上两者并不一样。有关该类搜索思想请参阅 DFS（搜索）.

过程

DFS 最显著的特征在于其**递归调用自身**。同时与 BFS 类似，DFS 会对其访问过的点打上访问标记，在遍历图时跳过已打过标记的点，以确保**每个点仅访问一次**。符合以上两条规则的函数，便是广义上的 DFS。

具体地说，DFS 大致结构如下：

```
DFS(v) // v 可以是图中的一个顶点，也可以是抽象的概念，如 dp 状态等。
  在 v 上打访问标记
  for u in v 的相邻节点
    if u 没有打过访问标记 then
      DFS(u)
    end
  end
end
```

以上代码只包含了 DFS 必需的主要结构。实际的 DFS 会在以上代码基础上加入一些代码，利用 DFS 性质进行其他操作。

性质

该算法通常的时间复杂度为 $O(n+m)$，空间复杂度为 $O(n)$，其中 n 表示点数，m 表示边数。注意空间复杂度包含了栈空间，栈空间的空间复杂度是 $O(n)$ 的。在平均 $O(1)$ 遍历一条边的条件下才能达到此时间复杂度，例如用前向星或邻接表存储图；如果用邻接矩阵则不一定能达到此复杂度。

> 备注：目前大部分算法竞赛（包括 NOIP、大部分省选以及 CCF 举办的各项赛事）都支持**无限栈空间**，即：栈空间不单独限制，但总内存空间仍然受题面限制。但大部分操作系统会对栈空间做额外的限制，因此在本地调试时需要一些方式来取消栈空间限制。
>
> - 在 Windows 上，通常的方法是在**编译选项**中加入 `-Wl,--stack=1000000000`，表示将栈空间限制设置为 1000000000 字节。
> - 在 Linux 上，通常的方法是在运行程序前**在终端内**执行 `ulimit -s unlimited`，表示栈空间无限。每个终端只需执行一次，对之后每次程序运行都有效。

实现

栈实现

DFS 可以使用 栈 （Stack） 为遍历中节点的暂存容器来实现；这与用 队列 （Queue） 实现的 BFS 形成高度对应。

```cpp
vector<vector<int>> adj;  // 邻接表
vector<bool> vis;         // 记录节点是否已经遍历

void dfs(int s) {
  stack<int> st;
  st.push(s);
  vis[s] = true;

  while (!st.empty()) {
    int u = st.top();
    st.pop();

    for (int v : adj[u]) {
      if (!vis[v]) {
        vis[v] = true;  // 确保栈里没有重复元素
        st.push(v);
      }
    }
  }
}
```

```python
# adj : List[List[int]] 邻接表
# vis : List[bool] 记录节点是否已经遍历

def dfs(s : int) -> None:
    stack = [s] # 用列表来模拟栈，把起点加入栈中
    vis[s] = True # 起点被遍历

    while(not stack):
        u = stack.pop() # 拿取并丢弃掉最后一个元素（栈顶的元素），可以理解为走到 u 这
个元素

        for v in adj[u]: # 对于与 u 相邻的每个元素 v
            if not vis[v]: # 如果 v 在此前没有走过
                vis[v] = True # 确保栈里没有重复元素
                stack.append(v) # 把 v 加入栈中
```

递归实现

函数在递归调用时的求值如同对栈的添加和删除元素的顺序，故函数调用所占据的虚拟地址被称为函数调用栈（Call Stack），DFS 可用递归的方式实现。

以 邻接表（Adjacency List） 作为图的存储方式：

```
vector<vector<int>> adj;  // 邻接表
vector<bool> vis;         // 记录节点是否已经遍历

void dfs(const int u) {
  vis[u] = true;
  for (int v : adj[u])
    if (!vis[v]) dfs(v)
}
```

```
# adj : List[List[int]] 邻接表
# vis : List[bool] 记录节点是否已经遍历

def dfs(u : int) -> None:
    vis[u] = True
    for v in adj[u]:
        if not vis[v]:
            dfs(v)
```

以 链式前向星 为例：

```
void dfs(int u) {
  vis[u] = 1;
  for (int i = head[u]; i; i = e[i].x) {
    if (!vis[e[i].t]) {
      dfs(v);
    }
  }
}
```

```
public void dfs(int u) {
    vis[u] = true;
    for (int i = head[u]; i != 0; i = e[i].x) {
        if (!vis[e[i].t]) {
            dfs(v);
        }
    }
}
```

```
def dfs(u):
    vis[u] = True
    i = head[u]
    while i:
        if vis[e[i].t] == False:
            dfs(v)
        i = e[i].x
```

DFS 序列

DFS 序列是指 DFS 调用过程中访问的节点编号的序列。

我们发现，每个子树都对应 DFS 序列中的连续一段（一段区间）。

括号序列

DFS 进入某个节点的时候记录一个左括号（，退出某个节点的时候记录一个右括号 ）。

每个节点会出现两次。相邻两个节点的深度相差 1。

一般图上 DFS

对于非连通图，只能访问到起点所在的连通分量。

对于连通图，DFS 序列通常不唯一。

注：树的 DFS 序列也是不唯一的。

在 DFS 过程中，通过记录每个节点从哪个点访问而来，可以建立一个树结构，称为 DFS 树。DFS 树是原图的一个生成树。

DFS 树 有很多性质，比如可以用来求 强连通分量。

参考资料与注释

[1] Depth First Search

11.5 BFS（图论）

Authors: Ir1d, greyqz, yjl9903, Anguei, Marcythm, ChungZH, Xeonacid, ylxmf2005

BFS 全称是 Breadth First Search[1]，中文名是宽度优先搜索，也叫广度优先搜索。

是图上最基础、最重要的搜索算法之一。

所谓宽度优先。就是每次都尝试访问同一层的节点。如果同一层都访问完了，再访问下一层。

这样做的结果是，BFS 算法找到的路径是从起点开始的**最短**合法路径。换言之，这条路径所包含的边数最小。

在 BFS 结束时，每个节点都是通过从起点到该点的最短路径访问的。

算法过程可以看做是图上火苗传播的过程：最开始只有起点着火了，在每一时刻，有火的节点都向它相邻的所有节点传播火苗。

实现

下文中 C++ 与 Python 的代码实现是基于链式前向星的存图方式，其实现可参考 图的存储 页面。

```
bfs(s) {
  q = new queue()
  q.push(s), visited[s] = true
  while (!q.empty()) {
    u = q.pop()
    for each edge(u, v) {
      if (!visited[v]) {
        q.push(v)
        visited[v] = true
      }
    }
  }
```

```
}
```

```cpp
void bfs(int u) {
  while (!Q.empty()) Q.pop();
  Q.push(u);
  vis[u] = 1;
  d[u] = 0;
  p[u] = -1;
  while (!Q.empty()) {
    u = Q.front();
    Q.pop();
    for (int i = head[u]; i; i = e[i].nxt) {
      if (!vis[e[i].to]) {
        Q.push(e[i].to);
        vis[e[i].to] = 1;
        d[e[i].to] = d[u] + 1;
        p[e[i].to] = u;
      }
    }
  }
}

void restore(int x) {
  vector<int> res;
  for (int v = x; v != -1; v = p[v]) {
    res.push_back(v);
  }
  std::reverse(res.begin(), res.end());
  for (int i = 0; i < res.size(); ++i) printf("%d", res[i]);
  puts("");
}
```

```python
from queue import Queue

def bfs(u):
    Q = Queue()
    Q.put(u)
    vis[u] = True
    d[u] = 0
    p[u] = -1
    while Q.qsize() != 0:
        u = Q.get()
        i = head[u]
        while i:
            if vis[e[i].to] == False:
                Q.put(e[i].to)
                vis[e[i].to] = True
                d[e[i].to] = d[u] + 1
                p[e[i].to] = u
            i = e[i].nxt

def restore(x):
    res = []
    v = x
```

```
    while v != -1:
        res.append(v)
        v = p[v]
    res.reverse()
    for i in range(0, len(res)):
        print(res[i])
```

具体来说，我们用一个队列 Q 来记录要处理的节点，然后开一个布尔数组 `vis[]` 来标记是否已经访问过某个节点。

开始的时候，我们将所有节点的 `vis` 值设为 0，表示没有访问过；然后把起点 s 放入队列 Q 中并将 `vis[s]` 设为 1。

之后，我们每次从队列 Q 中取出队首的节点 u，然后把与 u 相邻的所有节点 v 标记为已访问过并放入队列 Q。

循环直至当队列 Q 为空，表示 BFS 结束。

在 BFS 的过程中，也可以记录一些额外的信息。例如上述代码中，d 数组用于记录起点到某个节点的最短距离（要经过的最少边数），p 数组记录是从哪个节点走到当前节点的。

有了 d 数组，可以方便地得到起点到一个节点的距离。

有了 p 数组，可以方便地还原出起点到一个点的最短路径。上述代码中的 `restore` 函数使用该数组依次输出从起点到节点 x 的最短路径所经过的节点。

时间复杂度 $O(n+m)$

空间复杂度 $O(n)$（`vis` 数组和队列）

open-closed 表

在实现 BFS 的时候，本质上我们把未被访问过的节点放在一个称为 open 的容器中，而把已经访问过了的节点放在一个称为 closed 容器中。

在树/图上 BFS

BFS 序列

类似 DFS 序列，BFS 序列是指在 BFS 过程中访问的节点编号的序列。

一般图上 BFS

如果原图不连通，只能访问到从起点出发能够到达的点。

BFS 序列通常也不唯一。

类似的我们也可以定义 BFS 树：在 BFS 过程中，通过记录每个节点从哪个点访问而来，可以建立一个树结构，即为 BFS 树。

应用

- 在一个无权图上求从起点到其他所有点的最短路径。
- 在 $O(n+m)$ 时间内求出所有连通块。（我们只需要从每个没有被访问过的节点开始做 BFS，显然每次 BFS 会走完一个连通块）
- 如果把一个游戏的动作看做是状态图上的一条边（一个转移），那么 BFS 可以用来找到在游戏中从一个状态到达另一个状态所需要的最小步骤。
- 在一个有向无权图中找最小环。（从每个点开始 BFS，在我们即将抵达一个之前访问过的点开始的时候，就知道遇到了一个环。图的最小环是每次 BFS 得到的最小环的平均值。）

- 找到一定在 (a,b) 最短路上的边。（分别从 a 和 b 进行 BFS，得到两个 d 数组。之后对每一条边 (u,v)，如果 $d_a[u]+1+d_b[v]=d_a[b]$，则说明该边在最短路上）
- 找到一定在 (a,b) 最短路上的点。（分别从 a 和 b 进行 BFS，得到两个 d 数组。之后对每一个点 v，如果 $d_a[v]+d_b[v]=d_a[b]$，则说明该点在某条最短路上）
- 找到一条长度为偶数的最短路。（我们需要一个构造一个新图，把每个点拆成两个新点，原图的边 (u,v) 变成 $((u,0),(v,1))$ 和 $((u,1),(v,0))$。对新图做 BFS，$(s,0)$ 和 $(t,0)$ 之间的最短路即为所求）
- 在一个边权为 0/1 的图上求最短路，见下方双端队列 BFS。

双端队列 BFS

如果你不了解双端队列 `deque` 的话，请参阅 deque 相关章节。

双端队列 BFS 又称 0-1 BFS。

适用范围

边权值为可能有，也可能没有（由于 BFS 适用于权值为 1 的图，所以一般权值是 0 或 1），或者能够转化为这种边权值的最短路问题。

例如在走迷宫问题中，你可以花 1 个金币走 5 步，也可以不花金币走 1 步，这就可以用 0-1 BFS 解决。

实现

一般情况下，我们把没有权值的边扩展到的点放到队首，有权值的边扩展到的点放到队尾。这样即可保证像普通 BFS 一样整个队列队首到队尾权值单调不下降。

下面是伪代码：

```
while (队列不为空) {
  int u = 队首;
  弹出队首;
  for (枚举 u 的邻居) {
    更新数据
    if (...)
      添加到队首;
    else
      添加到队尾;
  }
}
```

例题

Codeforces 173B[2]

一个 $n\times m$ 的图，现在有一束激光从左上角往右边射出，每遇到'#'，你可以选择光线往四个方向射出，或者什么都不做，问最少需要多少个'#' 往四个方向射出才能使光线在第 n 行往右边射出。

此题目正解不是 0-1 BFS，但是适用 0-1 BFS，减小思维强度，赛时许多大佬都是这么做的。

做法很简单，一个方向射出不需要花费（0），而往四个方向射出需要花费（1），然后直接来就可以了。

代码

```cpp
#include <bits/stdc++.h>
using namespace std;

#define INF (1 << 29)
int n, m;
char grid[1001][1001];
int dist[1001][1001][4];
int fx[] = {1, -1, 0, 0};
int fy[] = {0, 0, 1, -1};
deque<int> q;  // 双端队列

void add_front(int x, int y, int dir, int d) {  // 向前方加
  if (d < dist[x][y][dir]) {
    dist[x][y][dir] = d;
    q.push_front(dir);
    q.push_front(y);
    q.push_front(x);
  }
}

void add_back(int x, int y, int dir, int d) {  // 向后方加
  if (d < dist[x][y][dir]) {
    dist[x][y][dir] = d;
    q.push_back(x);
    q.push_back(y);
    q.push_back(dir);
  }
}

int main() {
  cin >> n >> m;
  for (int i = 0; i < n; i++) cin >> grid[i];

  for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
      for (int k = 0; k < 4; k++) dist[i][j][k] = INF;

  add_front(n - 1, m - 1, 3, 0);

  while (!q.empty()) {  // 具体搜索的过程，可以参考上面写的题解
    int x = q[0], y = q[1], dir = q[2];
    q.pop_front();
    q.pop_front();
    q.pop_front();
    int d = dist[x][y][dir];
    int nx = x + fx[dir], ny = y + fy[dir];
    if (nx >= 0 && nx < n && ny >= 0 && ny < m)
      add_front(nx, ny, dir, d);  // 判断条件
    if (grid[x][y] == '#')
      for (int i = 0; i < 4; i++)
        if (i != dir) add_back(x, y, i, d + 1);
  }
  if (dist[0][0][3] == INF)
    cout << -1 << endl;
```

```
    else
        cout << dist[0][0][3] << endl;
    return 0;
}
```

优先队列 BFS

优先队列，相当于一个二叉堆，STL 中提供了 `std::priority_queue`，可以方便我们使用优先队列。

在基于优先队列的 BFS 中，我们每次从队首取出代价最小的结点进行进一步搜索。容易证明这个贪心思想是正确的，因为从这个结点开始扩展的搜索，一定不会更新原来那些代价更高的结点。换句话说，其余那些代价更高的结点，我们不回去考虑更新它。

当然，每个结点可能会被入队多次，只是每次入队的代价不同。当该结点第一次从优先队列中取出，以后便无需再在该结点进行搜索，直接忽略即可。所以，优先队列的 BFS 当中，每个结点只会被处理一次。

相对于普通队列的 BFS，时间复杂度多了一个 $\log n$，毕竟要维护这个优先队列嘛。不过普通 BFS 有可能每个结点入队、出队多次，时间复杂度会达到 $O(n^2)$，不是 $O(n)$。所以优先队列 BFS 通常还是快的。

诶？这怎么听起来这么像堆优化的 Dijkstra 算法呢？事实上，堆优化 Dijkstra 就是优先队列 BFS。

习题

- 「NOIP2017」 奶酪[3]

双端队列 BFS：

- CF1063B. Labyrinth[4]
- CF173B. Chamber of Secrets[5]
- 「BalticOI 2011 Day1」 打开灯泡 Switch the Lamp On[6]

参考

https://cp-algorithms.com/graph/breadth-first-search.html[7]

参考资料与注释

[1] Breadth First Search

[2] Codeforces 173B

[3] 「NOIP2017」奶酪

[4] CF1063B. Labyrinth

[5] CF173B. Chamber of Secrets

[6] 「BalticOI 2011 Day1」打开灯泡 Switch the Lamp On

[7] https://cp-algorithms.com/graph/breadth-first-search.html

11.6 树上问题

11.6.1 树基础

引入

图论中的树和现实生活中的树长得一样，只不过我们习惯于处理问题的时候把树根放到上方来考虑。这种数据结构看起来像是一个倒挂的树，因此得名。

定义

一个没有固定根结点的树称为**无根树**（unrooted tree）。无根树有几种等价的形式化定义：

- 有 n 个结点，$n-1$ 条边的连通无向图
- 无向无环的连通图
- 任意两个结点之间有且仅有一条简单路径的无向图
- 任何边均为桥的连通图
- 没有圈，且在任意不同两点间添加一条边之后所得图含唯一的一个圈的图

在无根树的基础上，指定一个结点称为**根**，则形成一棵**有根树**（rooted tree）。有根树在很多时候仍以无向图表示，只是规定了结点之间的上下级关系，详见下文。

有关树的定义

适用于无根树和有根树

- **森林（forest）**：每个连通分量（连通块）都是树的图。按照定义，一棵树也是森林。
- **生成树（spanning tree）**：一个连通无向图的生成子图，同时要求是树。也即在图的边集中选择 $n-1$ 条，将所有顶点连通。
- **无根树的叶结点（leaf node）**：度数不超过 1 的结点。

  " 为什么不是度数恰为 1 ? "

  考虑 $n=1$。

- **有根树的叶结点（leaf node）**：没有子结点的结点。

只适用于有根树

- **父亲（parent node）**：对于除根以外的每个结点，定义为从该结点到根路径上的第二个结点。
  根结点没有父结点。
- **祖先（ancestor）**：一个结点到根结点的路径上，除了它本身外的结点。
  根结点的祖先集合为空。
- **子结点（child node）**：如果 u 是 v 的父亲，那么 v 是 u 的子结点。
  子结点的顺序一般不加以区分，二叉树是一个例外。
- **结点的深度（depth）**：到根结点的路径上的边数。
- **树的高度（height）**：所有结点的深度的最大值。
- **兄弟（sibling）**：同一个父亲的多个子结点互为兄弟。
- **后代（descendant）**：子结点和子结点的后代。
  或者理解成：如果 u 是 v 的祖先，那么 v 是 u 的后代。

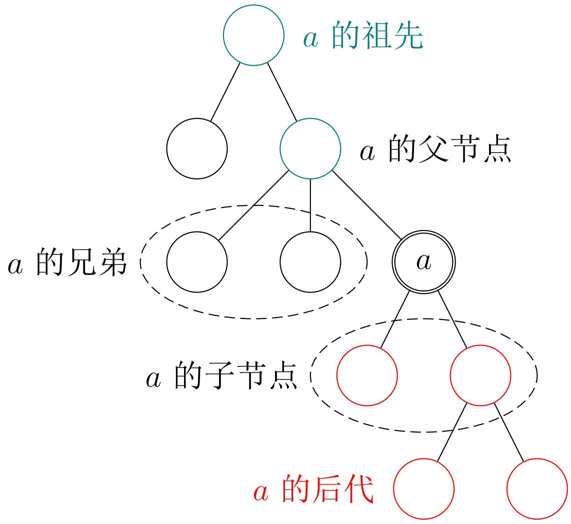


图 11.1 tree-definition.svg

- **子树**（**subtree**）：删掉与父亲相连的边后，该结点所在的子图。

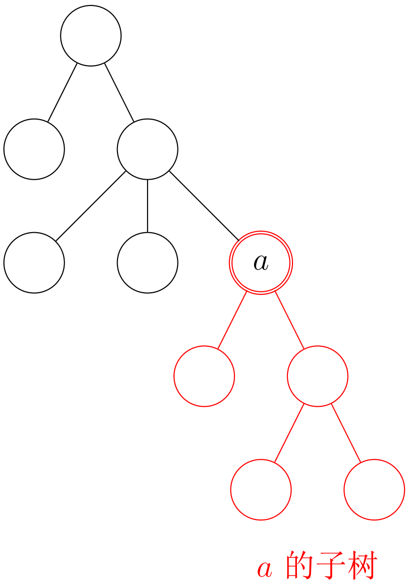
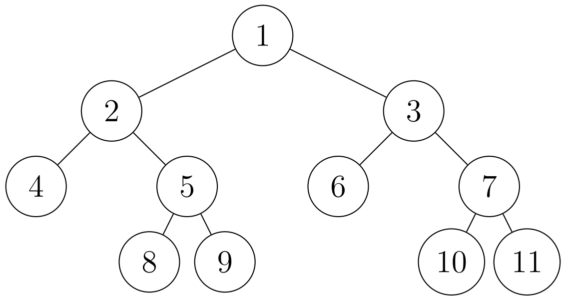


图 11.2 tree-definition-subtree.svg

特殊的树

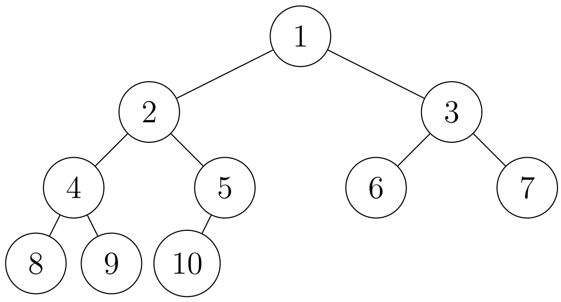
- **链**（**chain/path graph**）：满足与任一结点相连的边不超过 2 条的树称为链。
- **菊花/星星**（**star**）：满足存在 u 使得所有除 u 以外结点均与 u 相连的树称为菊花。
- **有根二叉树**（**rooted binary tree**）：每个结点最多只有两个儿子（子结点）的有根树称为二叉树。常常对两个子结点的顺序加以区分，分别称之为左子结点和右子结点。

  大多数情况下，**二叉树**一词均指有根二叉树。
- **完整二叉树**（**full/proper binary tree**）：每个结点的子结点数量均为 0 或者 2 的二叉树。换言之，每个结点或者是树叶，或者左右子树均非空。
- **完全二叉树**（**complete binary tree**）：只有最下面两层结点的度数可以小于 2，且最下面一层的结点都集中在该层最左边的连续位置上。
- **完美二叉树**（**perfect binary tree**）：所有叶结点的深度均相同，且所有非叶节点的子节点数量均为 2 的二叉树称为完美二叉树。



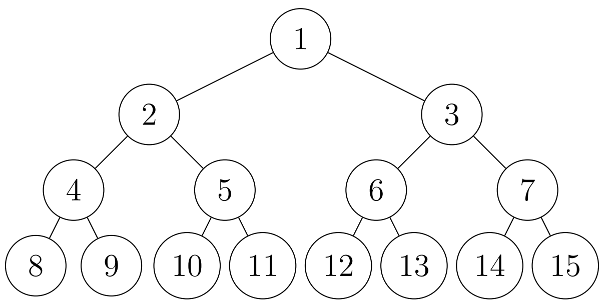
完整二叉树（proper binary tree）

图 11.3



完全二叉树（complete binary tree）

图 11.4



完美二叉树（perfect binary tree）

图 11.5

warning

Proper binary tree 的汉译名称不固定，且完全二叉树和满二叉树的定义在不同教材中定义不同，遇到的时候需根据上下文加以判断。

OIers 所说的「满二叉树」多指完美二叉树。

存储

只记录父结点

用一个数组 `parent[N]` 记录每个结点的父亲结点。

这种方式可以获得的信息较少，不便于进行自顶向下的遍历。常用于自底向上的递推问题中。

邻接表

- 对于无根树：为每个结点开辟一个线性列表，记录所有与之相连的结点。

  ```
  std::vector<int> adj[N];
  ```

- 对于有根树：
  - 方法一：若给定的是无向图，则仍可以上述形式存储。下文将介绍如何区分结点的上下关系。
  - 方法二：若输入数据能够确保结点的上下关系，则可以利用这个信息。为每个结点开辟一个线性列表，记录其所有子结点；若有需要，还可在另一个数组中记录其父结点。

    ```
    std::vector<int> children[N];
    int parent[N];
    ```

    当然也可以用其他方式（如链表）替代 `std::vector`。

左孩子右兄弟表示法

过程

对于有根树，存在一种简单的表示方法。

首先，给每个结点的所有子结点任意确定一个顺序。

此后为每个结点记录两个值：其**第一个子结点** `child[u]` 和其**下一个兄弟结点** `sib[u]`。若没有子结点，则 `child[u]` 为空；若该结点是其父结点的最后一个子结点，则 `sib[u]` 为空。

实现

遍历一个结点的所有子结点可由如下方式实现。

```
int v = child[u];  // 从第一个子结点开始
while (v != EMPTY_NODE) {
  // ...
  // 处理子结点 v
  // ...
  v = sib[v];  // 转至下一个子结点，即 v 的一个兄弟
}
```

也可简写为以下形式。

```
for (int v = child[u]; v != EMPTY_NODE; v = sib[v]) {
  // ...
  // 处理子结点 v
  // ...
```

```
}
```

二叉树

需要记录每个结点的左右子结点。

" 实现"

```
int parent[N];
int lch[N], rch[N];
// -- or --
int child[N][2];
```

树的遍历

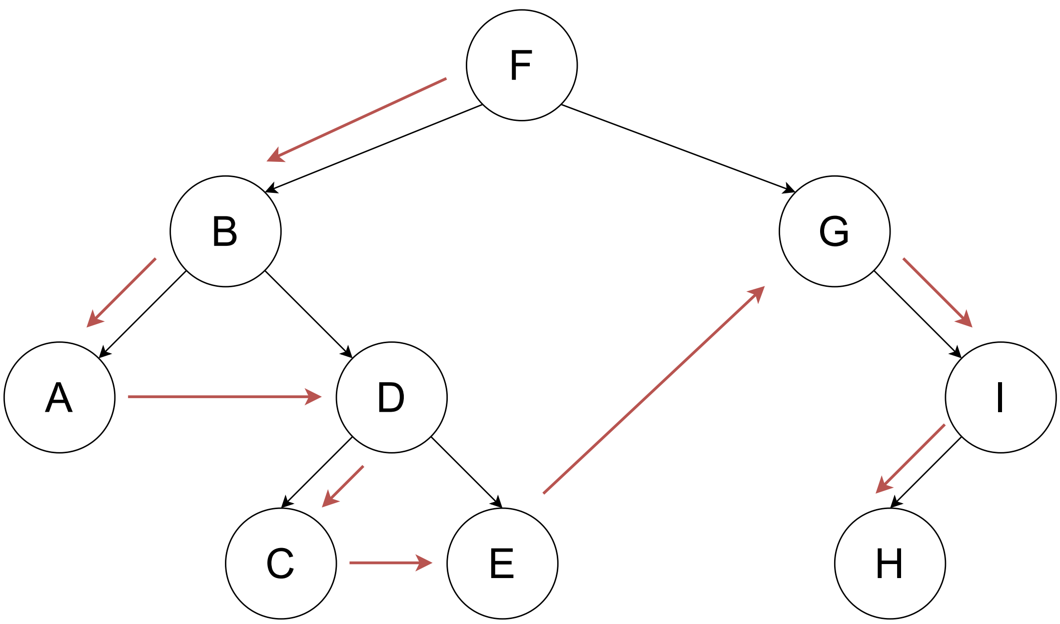
树上 DFS

在树上 DFS 是这样的一个过程：先访问根节点，然后分别访问根节点每个儿子的子树。

可以用来求出每个节点的深度、父亲等信息。

二叉树 DFS 遍历

先序遍历



Preorder:	F	B	A	D	C	E	G	I	H

图 11.6 preorder

按照**根**，**左**，**右**的顺序遍历二叉树。

" 实现"

```
void preorder(BiTree* root) {
  if (root) {
```

```
    cout << root->key << " ";
    preorder(root->left);
    preorder(root->right);
  }
}
```

中序遍历

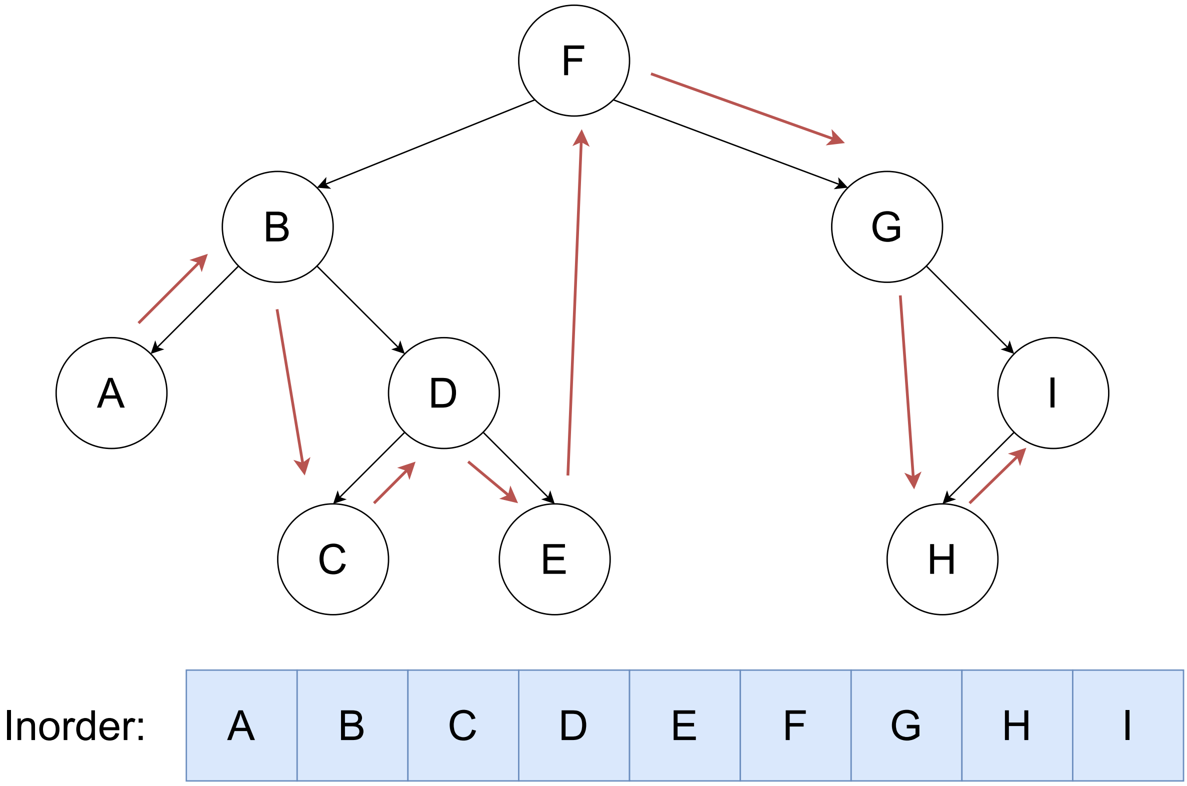


图 11.7 inorder

按照**左**，**根**，**右**的顺序遍历二叉树。

"实现"

```
void inorder(BiTree* root) {
  if (root) {
    inorder(root->left);
    cout << root->key << " ";
    inorder(root->right);
  }
}
```

后序遍历

按照**左**，**右**，**根**的顺序遍历二叉树。

"实现"

```
void postorder(BiTree* root) {
  if (root) {
    postorder(root->left);
    postorder(root->right);
```

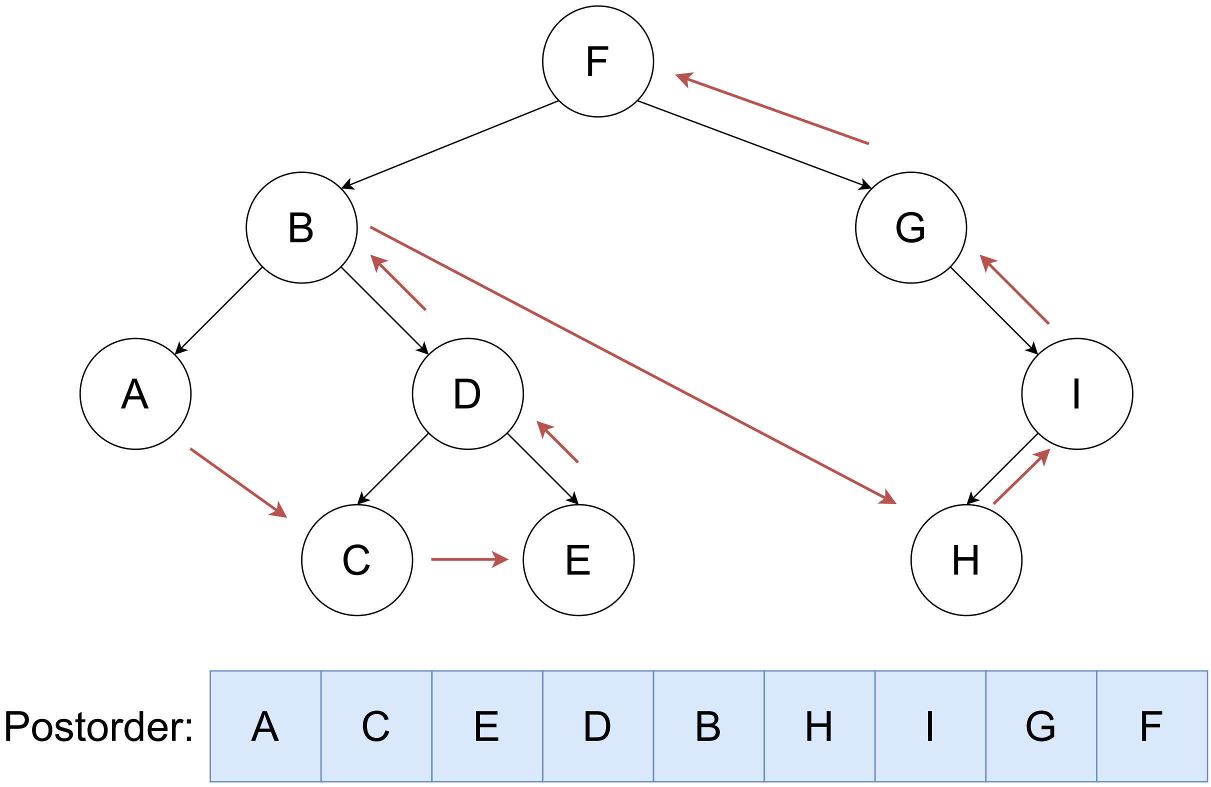


图 11.8 postorder

```
    cout << root->key << " ";
  }
}
```

反推

已知中序遍历序列和另外一个序列可以求第三个序列。

1. 前序的第一个是 `root`，后序的最后一个是 `root`。
2. 先确定根节点，然后根据中序遍历，在根左边的为左子树，根右边的为右子树。
3. 对于每一个子树可以看成一个全新的树，仍然遵循上面的规律。

树上 BFS

从树根开始，严格按照层次来访问节点。

BFS 过程中也可以顺便求出各个节点的深度和父亲节点。

树的层序遍历

树层序遍历是指按照从根节点到叶子节点的层次关系，一层一层的横向遍历各个节点。根据 BFS 的定义可以知道，BFS 所得到的遍历顺序就是一种层序遍历。但层序遍历要求将不同的层次区分开来，所以其结果通常以二维数组的形式表示。

例如，下图的树的层序遍历的结果是 `[[1], [2, 3, 4], [5, 6]]`（每一层从左向右）。

" 实现"

```
vector<vector<int>> levelOrder(Node* root) {
  if (!root) {
    return {};
```

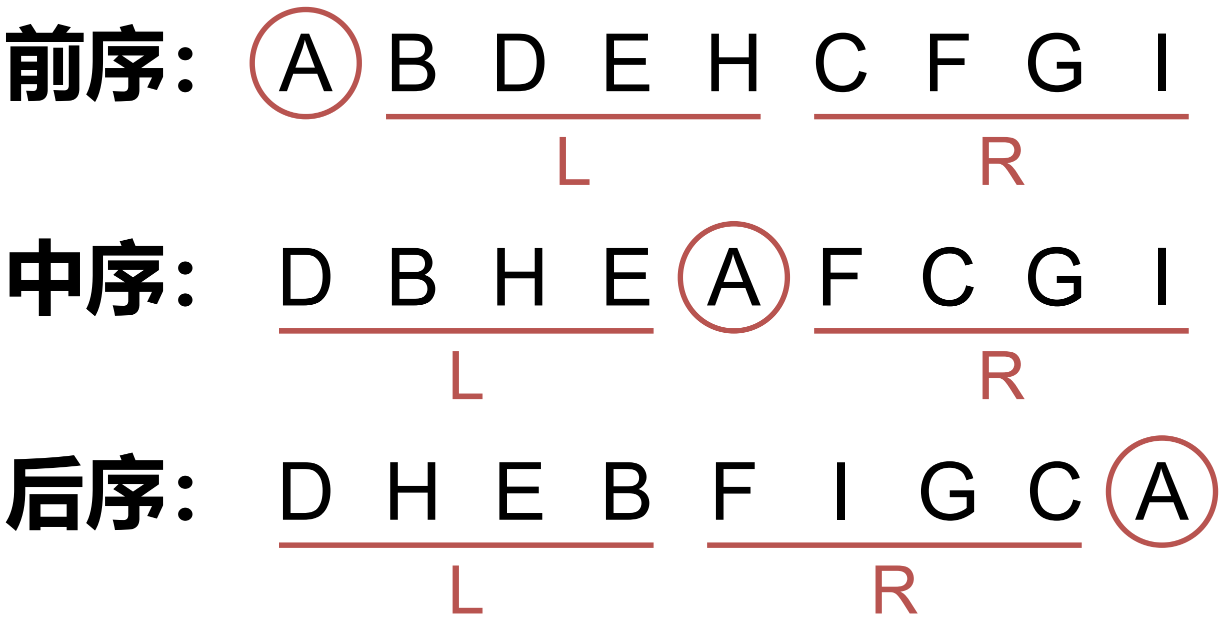


图 11.9 reverse

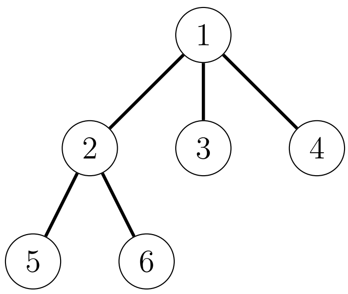


图 11.10 tree-basic-levelOrder

```
  }
  vector<vector<int>> res;
  queue<Node*> q;
  q.push(root);
  while (!q.empty()) {
    int currentLevelSize = q.size();  // 当前层的节点个数
    res.push_back(vector<int>());
    for (int i = 0; i < currentLevelSize; ++i) {
      Node* cur = q.front();
      q.pop();
      res.back().push_back(cur->val);
      for (Node* child : cur->children) {  // 把子节点都加入
        q.push(child);
      }
    }
  }
  return res;
}
```

二叉树 Morris 遍历

二叉树遍历的核心问题是，当遍历当前节点的子节点后，如何返回当前节点并继续遍历。遍历二叉树的递归方法和非递归方法都使用了栈结构，记录返回路径，来实现从下层到上层的移动。其空间复杂度最好时为 $O(\log n)$，最坏时为 $O(n)$（二叉树呈线性）。

Morris 遍历的实质是避免使用栈，利用底层节点空闲的 `right` 指针指回上层的某个节点，从而完成下层到上层的移动。

Morris 遍历的过程

假设来到当前节点 `cur`，开始时来到根节点位置。

1. 如果 `cur` 为空时遍历停止，否则进行以下过程。
2. 如果 `cur` 没有左子树，`cur` 向右移动（`cur = cur->right`）。
3. 如果 `cur` 有左子树，找到左子树上最右的节点，记为 `mostRight`。
   - 如果 `mostRight` 的 `right` 指针指向空，让其指向 `cur`，然后 `cur` 向左移动（`cur = cur->left`）。
   - 如果 `mostRight` 的 `right` 指针指向 `cur`，将其修改为 `null`，然后 `cur` 向右移动（`cur = cur->right`）。

例如，`cur` 从节点 1 开始访问。

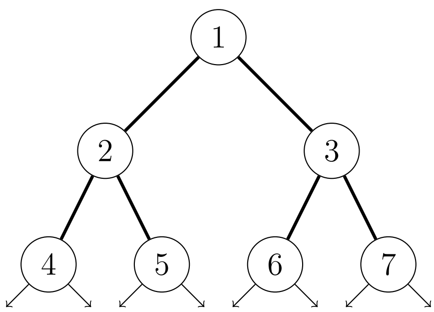


图 11.11 tree-basic-morris-1

`cur` 第一次访问节点 2 时，找到左子树上最右的节点 4，将 4 的 `right` 指针指向 `cur`（节点 2）。

`cur` 通过 4 的 `right` 指针返回上层，第二次访问节点 2 时，找到左子树上最右节点 4，将 4 的 `right` 指针修改为 `null`，然后继续访问右子树。之后的过程省略。

整棵树的访问顺序是 `1242513637`。可以发现有左子树的节点访问两次，没有左子树的节点只访问一次。

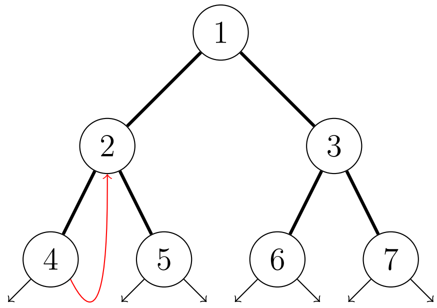


图 11.12 tree-basic-morris-2

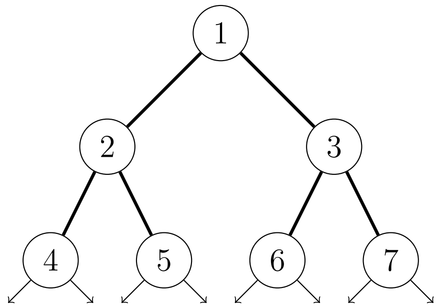


图 11.13 tree-basic-morris-1

" 实现"

```
void morris(TreeNode* root) {
  TreeNode* cur = root;
  while (cur) {
    if (!cur->left) {
      // 如果当前节点没有左子节点，则输出当前节点的值并进入右子树
      std::cout << cur->val << " ";
      cur = cur->right;
      continue;
    }
    // 找到当前节点的左子树的最右节点
    TreeNode* mostRight = cur->left;
    while (mostRight->right && mostRight->right != cur) {
      mostRight = mostRight->right;
    }
    if (!mostRight->right) {
      // 如果最右节点的 right 指针为空，将其指向当前节点，并进入左子树
      mostRight->right = cur;
      cur = cur->left;
    } else {
      // 如果最右节点的 right 指针指向当前节点，说明左子树已经遍历完毕，输出当前节点
的值并进入右子树
      mostRight->right = nullptr;
      std::cout << cur->val << " ";
      cur = cur->right;
    }
  }
}
```

无根树

过程

树的遍历一般为深度优先遍历，这个过程中最需要注意的是避免重复访问结点。

由于树是无环图，因此只需记录当前结点是由哪个结点访问而来，此后进入除该结点外的所有相邻结点，即可避免重复访问。

" 实现"

```cpp
void dfs(int u, int from) {
  // 递归进入除了 from 之外的所有子结点
  // 对于出发结点，from 为空，故会访问所有相邻结点，这与期望一致
  for (int v : adj[u])
    if (v != from) {
      dfs(v, u);
    }
}

// 开始遍历时
int EMPTY_NODE = -1;  // 一个不存在的编号
int root = 0;         // 任取一个结点作为出发点
dfs(root, EMPTY_NODE);
```

有根树

对于有根树，需要区分结点的上下关系。

考察上面的遍历过程，若从根开始遍历，则访问到一个结点时 `from` 的值，就是其父结点的编号。

通过这个方式，可以对于无向的输入求出所有结点的父结点，以及子结点列表。

参考资料与注释

[1] 二叉树：前序遍历、中序遍历、后续遍历

11.6.2 树的直径

树上任意两节点之间最长的简单路径即为树的「直径」。

前置知识：树基础。

引入

显然，一棵树可以有多条直径，他们的长度相等。

可以用两次 DFS 或者树形 DP 的方法在 $O(n)$ 时间求出树的直径。

例题

"SPOJ PT07Z, Longest path in a tree[1]"

给定一棵 n 个节点的树，求其直径的长度。$1 \leq n \leq 10^4$。

做法 1. 两次 DFS

过程

首先从任意节点 y 开始进行第一次 DFS，到达距离其最远的节点，记为 z，然后再从 z 开始做第二次 DFS，到达距离 z 最远的节点，记为 z'，则 $\delta(z,z')$ 即为树的直径。

显然，如果第一次 DFS 到达的节点 z 是直径的一端，那么第二次 DFS 到达的节点 z' 一定是直径的一端。我们只需证明在任意情况下，z 必为直径的一端。

定理：在一棵树上，从任意节点 y 开始进行一次 DFS，到达的距离其最远的节点 z 必为直径的一端。

" 证明"

使用反证法。记出发节点为 y。设真实的直径是 $\delta(s,t)$，而从 y 进行的第一次 DFS 到达的距离其最远的节点 z 不为 t 或 s。共分三种情况：

- 若 y 在 $\delta(s,t)$ 上：

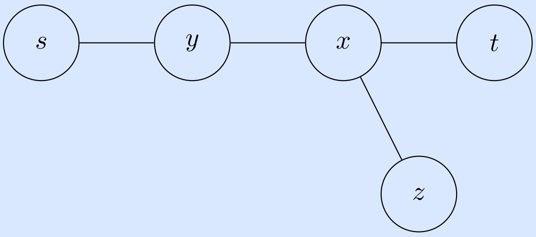


图 11.14 y 在 s-t 上

有 $\delta(y,z)>\delta(y,t)\Longrightarrow\delta(x,z)>\delta(x,t)\Longrightarrow\delta(s,z)>\delta(s,t)$，与 $\delta(s,t)$ 为树上任意两节点之间最长的简单路径矛盾。

- 若 y 不在 $\delta(s,t)$ 上，且 $\delta(y,z)$ 与 $\delta(s,t)$ 存在重合路径：

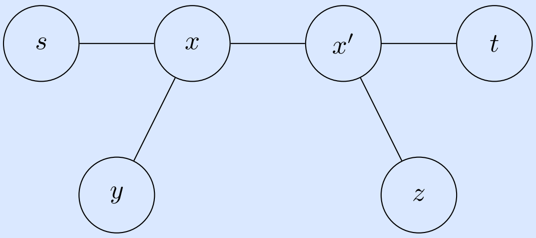


图 11.15 y 不在 s-t 上，y-z 与 s-t 存在重合路径

有 $\delta(y,z)>\delta(y,t)\Longrightarrow\delta(x,z)>\delta(x,t)\Longrightarrow\delta(s,z)>\delta(s,t)$，与 $\delta(s,t)$ 为树上任意两节点之间最长的简单路径矛盾。

- 若 y 不在 $\delta(s,t)$ 上，且 $\delta(y,z)$ 与 $\delta(s,t)$ 不存在重合路径：

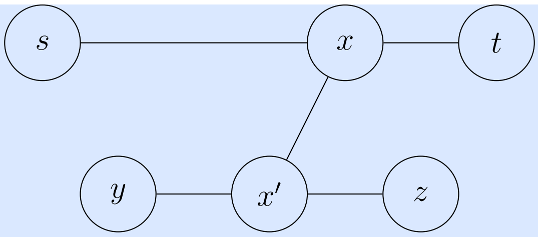


图 11.16 y 不在 s-t 上，y-z 与 s-t 不存在重合路径

有 $\delta(y,z) > \delta(y,t) \Longrightarrow \delta(x',z) > \delta(x',t) \Longrightarrow \delta(x,z) > \delta(x,t) \Longrightarrow \delta(s,z) > \delta(s,t)$，与 $\delta(s,t)$ 为树上任意两节点之间最长的简单路径矛盾。

综上，三种情况下假设均会产生矛盾，故原定理得证。

" 负权边"

上述证明过程建立在所有路径均不为负的前提下。如果树上存在负权边，则上述证明不成立。故若存在负权边，则无法使用两次 DFS 的方式求解直径。

实现

代码实现如下。

```cpp
const int N = 10000 + 10;

int n, c, d[N];
vector<int> E[N];

void dfs(int u, int fa) {
  for (int v : E[u]) {
    if (v == fa) continue;
    d[v] = d[u] + 1;
    if (d[v] > d[c]) c = v;
    dfs(v, u);
  }
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    E[u].push_back(v), E[v].push_back(u);
  }
  dfs(1, 0);
  d[c] = 0, dfs(c, 0);
  printf("%d\n", d[c]);
  return 0;
}
```

如果需要求出一条直径上所有的节点，则可以在第二次 DFS 的过程中，记录每个点的前序节点，即可从直径的一端一路向前，遍历直径上所有的节点。

做法 2. 树形 DP

过程 1

我们记录当 1 为树的根时，每个节点作为子树的根向下，所能延伸的最长路径长度 d_1 与次长路径（与最长路径无公共边）长度 d_2，那么直径就是对于每一个点，该点 $d_1 + d_2$ 能取到的值中的最大值。

树形 DP 可以在存在负权边的情况下求解出树的直径。

实现 1

代码实现如下：

```
const int N = 10000 + 10;

int n, d = 0;
int d1[N], d2[N];
vector<int> E[N];

void dfs(int u, int fa) {
  d1[u] = d2[u] = 0;
  for (int v : E[u]) {
    if (v == fa) continue;
    dfs(v, u);
    int t = d1[v] + 1;
    if (t > d1[u])
      d2[u] = d1[u], d1[u] = t;
    else if (t > d2[u])
      d2[u] = t;
  }
  d = max(d, d1[u] + d2[u]);
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    E[u].push_back(v), E[v].push_back(u);
  }
  dfs(1, 0);
  printf("%d\n", d);
  return 0;
}
```

如果需要求出一条直径上所有的节点，则可以在 DP 的过程中，记录下每个节点能向下延伸的最长路径与次长路径（定义同上）所对应的子节点，在求 d 的同时记下对应的节点 u，使得 $d = d_1[u] + d_2[u]$，即可分别沿着从 u 开始的最长路径的次长路径对应的子节点一路向某个方向（对于无根树，虽然这里指定了 1 为树的根，但仍需记录每点跳转的方向；对于有根树，一路向上跳即可），遍历直径上所有的节点。

过程 2

这里提供一种只使用一个数组进行的树形 DP 方法。

我们定义 $dp[u]$：以 u 为根的子树中，从 u 出发的最长路径。那么容易得出转移方程：$dp[u] = \max(dp[u], dp[v] + w(u, v))$，其中的 &v& 为 u 的子节点，$w(u, v)$ 表示所经过边的权重。

对于树的直径，实际上是可以通过枚举从某个节点出发不同的两条路径相加的最大值求出。因此，在 DP 求解的过程中，我们只需要在更新 $dp[u]$ 之前，计算 $d = \max(d, dp[u] + dp[v] + w(u, v))$ 即可算出直径 d。

实现 2

代码实现如下：

```
const int N = 10000 + 10;

int n, d = 0;
int dp[N];
vector<int> E[N];

void dfs(int u, int fa) {
  for (int v : E[u]) {
    if (v == fa) continue;
    dfs(v, u);
    d = max(d, dp[u] + dp[v] + 1);
    dp[u] = max(dp[u], dp[v] + 1);
  }
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    E[u].push_back(v), E[v].push_back(u);
  }
  dfs(1, 0);
  printf("%d\n", d);
  return 0;
}
```

性质

若树上所有边边权均为正，则树的所有直径中点重合

证明：使用反证法。设两条中点不重合的直径分别为 $\delta(s,t)$ 与 $\delta(s',t')$，中点分别为 x 与 x'。显然，$\delta(s,x)=\delta(x,t)=\delta(s',x')=\delta(x',t')$。

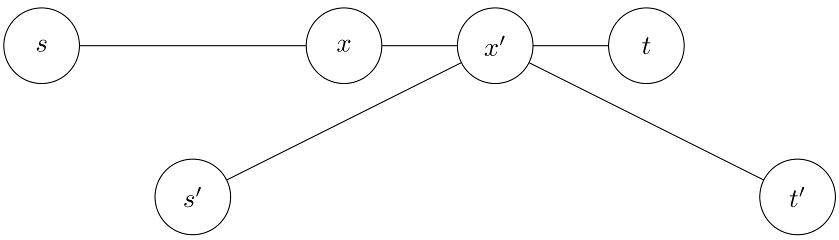


图 11.17 无负权边的树所有直径的中点重合

有 $\delta(s,t')=\delta(s,x)+\delta(x,x')+\delta(x',t')>\delta(s,x)+\delta(x,t)=\delta(s,t)$，与 $\delta(s,t)$ 为树上任意两节点之间最长的简单路径矛盾，故性质得证。

习题

- CodeChef, Diameter of Tree[2]
- Educational Codeforces Round 35, Problem F, Tree Destruction[3]
- ZOJ 3820, Building Fire Stations[4]

- CEOI2019/CodeForces 1192B. Dynamic Diameter[5]
- IPSC 2019 网络赛 ，Lightning Routing I[6]
- NOIP2007 提高组 树网的核[7]
- SDOI2011 消防[8]
- APIO2010 巡逻[9]

参考资料与注释

[1] SPOJ PT07Z, Longest path in a tree

[2] CodeChef, Diameter of Tree

[3] Educational Codeforces Round 35, Problem F, Tree Destruction

[4] ZOJ 3820, Building Fire Stations

[5] CEOI2019/CodeForces 1192B. Dynamic Diameter

[6] IPSC 2019 网络赛，Lightning Routing I

[7] NOIP2007 提高组树网的核

[8] SDOI2011 消防

[9] APIO2010 巡逻

11.6.3 最近公共祖先

author:ouuan, Backl1ght, billchenchina, CCXXXI, ChickenHu, ChungZH, cjsoft, countercurrent-time, diauweb, Early0v0, Enter-tainer, EtaoinWu, H-J-Granger, H-Shen, Henry-ZHR, HeRaNO, hsfzLZH1, huaruoji, iamtwz, imp2002, Ir1d, kenlig, Konano, Lyccrius, Marcythm, Menci, NachtgeistW, PeterlitsZo, psz2007, shuzhouliu, SkqLiao, sshwy, SukkaW, therehello, TrisolarisHD, ttzztztz, vincent-163, WAAutoMaton, Hunter19019

定义

最近公共祖先简称 LCA（Lowest Common Ancestor）。两个节点的最近公共祖先，就是这两个点的公共祖先里面，离根最远的那个。为了方便，我们记某点集 $S=\{v_1,v_2,\ldots,v_n\}$ 的最近公共祖先为 $\text{LCA}(v_1,v_2,\ldots,v_n)$ 或 $\text{LCA}(S)$。

性质

> 本节**性质**部分内容翻译自 wcipeg[1]，并做过修改。

1. $\text{LCA}(\{u\})=u$；
2. u 是 v 的祖先，当且仅当 $\text{LCA}(u,v)=u$；
3. 如果 u 不为 v 的祖先并且 v 不为 u 的祖先，那么 u,v 分别处于 $\text{LCA}(u,v)$ 的两棵不同子树中；
4. 前序遍历中，$\text{LCA}(S)$ 出现在所有 S 中元素之前，后序遍历中 $\text{LCA}(S)$ 则出现在所有 S 中元素之后；

5. 两点集并的最近公共祖先为两点集分别的最近公共祖先的最近公共祖先，即 $\text{LCA}(A\cup B)=\text{LCA}(\text{LCA}(A),\text{LCA}(B))$；
6. 两点的最近公共祖先必定处在树上两点间的最短路上；
7. $d(u,v)=h(u)+h(v)-2h(\text{LCA}(u,v))$，其中 d 是树上两点间的距离，h 代表某点到树根的距离。

求法

朴素算法

过程

可以每次找深度比较大的那个点，让它向上跳。显然在树上，这两个点最后一定会相遇，相遇的位置就是想要求的 LCA。或者先向上调整深度较大的点，令他们深度相同，然后再共同向上跳转，最后也一定会相遇。

性质

朴素算法预处理时需要 dfs 整棵树，时间复杂度为 $O(n)$，单次查询时间复杂度为 $\Theta(n)$。如果树满足随机性质，则时间复杂度与这种随机树的期望高度有关。

倍增算法

过程

倍增算法是最经典的 LCA 求法，他是朴素算法的改进算法。通过预处理 $\text{fa}_{x,i}$ 数组，游标可以快速移动，大幅减少了游标跳转次数。$\text{fa}_{x,i}$ 表示点 x 的第 2^i 个祖先。$\text{fa}_{x,i}$ 数组可以通过 dfs 预处理出来。

现在我们看看如何优化这些跳转：在调整游标的第一阶段中，我们要将 u,v 两点跳转到同一深度。我们可以计算出 u,v 两点的深度之差，设其为 y。通过将 y 进行二进制拆分，我们将 y 次游标跳转优化为「y 的二进制表示所含 `1` 的个数」次游标跳转。在第二阶段中，我们从最大的 i 开始循环尝试，一直尝试到 0（包括 0），如果 $\text{fa}_{u,i}\neq\text{fa}_{v,i}$，则 $u\leftarrow\text{fa}_{u,i},v\leftarrow\text{fa}_{v,i}$，那么最后的 LCA 为 $\text{fa}_{u,0}$。

性质

倍增算法的预处理时间复杂度为 $O(n\log n)$，单次查询时间复杂度为 $O(\log n)$。另外倍增算法可以通过交换 `fa` 数组的两维使较小维放在前面。这样可以减少 cache miss 次数，提高程序效率。

例题

HDU 2586 How far away?[2] 树上最短路查询。

可先求出 LCA，再结合性质 7 进行解答。也可以直接在求 LCA 时求出结果。

" 参考代码"

```
#include <cstdio>
#include <cstring>
#include <vector>

#define MXN 40005
using namespace std;
std::vector<int> v[MXN];
std::vector<int> w[MXN];

int fa[MXN][31], cost[MXN][31], dep[MXN];
int n, m;
int a, b, c;

// dfs，用来为 lca 算法做准备。接受两个参数：dfs 起始节点和它的父亲节点。
void dfs(int root, int fno) {
```

```
  // 初始化：第 2^0 = 1 个祖先就是它的父亲节点，dep 也比父亲节点多 1。
  fa[root][0] = fno;
  dep[root] = dep[fa[root][0]] + 1;
  // 初始化：其他的祖先节点：第 2^i 的祖先节点是第 2^(i-1) 的祖先节点的第
  // 2^(i-1) 的祖先节点。
  for (int i = 1; i < 31; ++i) {
    fa[root][i] = fa[fa[root][i - 1]][i - 1];
    cost[root][i] = cost[fa[root][i - 1]][i - 1] + cost[root][i - 1];
  }
  // 遍历子节点来进行 dfs。
  int sz = v[root].size();
  for (int i = 0; i < sz; ++i) {
    if (v[root][i] == fno) continue;
    cost[v[root][i]][0] = w[root][i];
    dfs(v[root][i], root);
  }
}

// lca。用倍增算法算取 x 和 y 的 lca 节点。
int lca(int x, int y) {
  // 令 y 比 x 深。
  if (dep[x] > dep[y]) swap(x, y);
  // 令 y 和 x 在一个深度。
  int tmp = dep[y] - dep[x], ans = 0;
  for (int j = 0; tmp; ++j, tmp >>= 1)
    if (tmp & 1) ans += cost[y][j], y = fa[y][j];
  // 如果这个时候 y = x，那么 x，y 就都是它们自己的祖先。
  if (y == x) return ans;
  // 不然的话，找到第一个不是它们祖先的两个点。
  for (int j = 30; j >= 0 && y != x; --j) {
    if (fa[x][j] != fa[y][j]) {
      ans += cost[x][j] + cost[y][j];
      x = fa[x][j];
      y = fa[y][j];
    }
  }
  // 返回结果。
  ans += cost[x][0] + cost[y][0];
  return ans;
}

void Solve() {
  // 初始化表示祖先的数组 fa，代价 cost 和深度 dep。
  memset(fa, 0, sizeof(fa));
  memset(cost, 0, sizeof(cost));
  memset(dep, 0, sizeof(dep));
  // 读入树：节点数一共有 n 个，查询 m 次，每一次查找两个节点的 lca 点。
  scanf("%d %d", &n, &m);
  // 初始化树边和边权
  for (int i = 1; i <= n; ++i) {
    v[i].clear();
    w[i].clear();
  }
  for (int i = 1; i < n; ++i) {
```

```
    scanf("%d %d %d", &a, &b, &c);
    v[a].push_back(b);
    v[b].push_back(a);
    w[a].push_back(c);
    w[b].push_back(c);
  }
  // 为了计算 lca 而使用 dfs。
  dfs(1, 0);
  for (int i = 0; i < m; ++i) {
    scanf("%d %d", &a, &b);
    printf("%d\n", lca(a, b));
  }
}

int main() {
  int T;
  scanf("%d", &T);
  while (T--) Solve();
  return 0;
}
```

Tarjan 算法

过程

Tarjan 算法是一种**离线算法**，需要使用 并查集 记录某个结点的祖先结点。做法如下：

1. 首先接受输入边（邻接链表）、查询边（存储在另一个邻接链表内）。查询边其实是虚拟加上去的边，为了方便，每次输入查询边的时候，将这个边及其反向边都加入到 `queryEdge` 数组里。
2. 然后对其进行一次 DFS 遍历，同时使用 `visited` 数组进行记录某个结点是否被访问过、`parent` 记录当前结点的父亲结点。
3. 其中涉及到了**回溯思想**，我们每次遍历到某个结点的时候，认为这个结点的根结点就是它本身。让以这个结点为根节点的 DFS 全部遍历完毕了以后，再将这个结点的根节点设置为这个结点的父一级结点。
4. 回溯的时候，如果以该节点为起点，`queryEdge` 查询边的另一个结点也恰好访问过了，则直接更新查询边的 LCA 结果。
5. 最后输出结果。

性质

Tarjan 算法需要初始化并查集，所以预处理的时间复杂度为 $O(n)$。

朴素的 Tarjan 算法处理所有 m 次询问的时间复杂度为 $O(m\alpha(m+n,n)+n)$,。但是 Tarjan 算法的常数比倍增算法大。存在 $O(m+n)$ 的实现。

" 注意"

并不存在「朴素 Tarjan LCA 算法中使用的并查集性质比较特殊，单次调用 `find()` 函数的时间复杂度为均摊 $O(1)$」这种说法。

以下的朴素 Tarjan 实现复杂度为 $O(m\alpha(m+n,n)+n)$。如果需要追求严格线性，可以参考 Gabow 和 Tarjan 于 1983 年的论文[3]。其中给出了一种复杂度为 $O(m+n)$ 的做法。

实现

" 参考代码"

```cpp
#include <algorithm>
#include <cstring>
#include <iostream>
using namespace std;

class Edge {
 public:
  int toVertex, fromVertex;
  int next;
  int LCA;
  Edge() : toVertex(-1), fromVertex(-1), next(-1), LCA(-1){};
  Edge(int u, int v, int n) : fromVertex(u), toVertex(v), next(n), LCA(-1){};
};

const int MAX = 100;
int head[MAX], queryHead[MAX];
Edge edge[MAX], queryEdge[MAX];
int parent[MAX], visited[MAX];
int vertexCount, queryCount;

int find(int x) {
  if (parent[x] == x) {
    return x;
  } else {
    return parent[x] = find(parent[x]);
  }
}

void tarjan(int u) {
  parent[u] = u;
  visited[u] = 1;

  for (int i = head[u]; i != -1; i = edge[i].next) {
    Edge& e = edge[i];
    if (!visited[e.toVertex]) {
      tarjan(e.toVertex);
      parent[e.toVertex] = u;
    }
  }

  for (int i = queryHead[u]; i != -1; i = queryEdge[i].next) {
    Edge& e = queryEdge[i];
    if (visited[e.toVertex]) {
      queryEdge[i ^ 1].LCA = e.LCA = find(e.toVertex);
    }
  }
}

int main() {
  memset(head, 0xff, sizeof(head));
  memset(queryHead, 0xff, sizeof(queryHead));

  cin >> vertexCount >> queryCount;
```

```cpp
  int count = 0;
  for (int i = 0; i < vertexCount - 1; i++) {
    int start = 0, end = 0;
    cin >> start >> end;

    edge[count] = Edge(start, end, head[start]);
    head[start] = count;
    count++;

    edge[count] = Edge(end, start, head[end]);
    head[end] = count;
    count++;
  }

  count = 0;
  for (int i = 0; i < queryCount; i++) {
    int start = 0, end = 0;
    cin >> start >> end;

    queryEdge[count] = Edge(start, end, queryHead[start]);
    queryHead[start] = count;
    count++;

    queryEdge[count] = Edge(end, start, queryHead[end]);
    queryHead[end] = count;
    count++;
  }

  tarjan(1);

  for (int i = 0; i < queryCount; i++) {
    Edge& e = queryEdge[i * 2];
    cout << "(" << e.fromVertex << "," << e.toVertex << ") " << e.LCA << endl;
  }

  return 0;
}
```

用欧拉序列转化为 RMQ 问题

定义

对一棵树进行 DFS，无论是第一次访问还是回溯，每次到达一个结点时都将编号记录下来，可以得到一个长度为 $2n-1$ 的序列，这个序列被称作这棵树的欧拉序列。

在下文中，把结点 u 在欧拉序列中第一次出现的位置编号记为 $pos(u)$（也称作节点 u 的欧拉序），把欧拉序列本身记作 $E[1..2n-1]$。

过程

有了欧拉序列，LCA 问题可以在线性时间内转化为 RMQ 问题，即 $pos(LCA(u,v)) = \min\{pos(k)|k \in E[pos(u)..pos(v)]\}$。

这个等式不难理解：从 u 走到 v 的过程中一定会经过 $LCA(u,v)$，但不会经过 $LCA(u,v)$ 的祖先。因此，从 u 走到 v 的过程中经过的欧拉序最小的结点就是 $LCA(u,v)$。

用 DFS 计算欧拉序列的时间复杂度是 $O(n)$，且欧拉序列的长度也是 $O(n)$，所以 LCA 问题可以在 $O(n)$ 的时间内转化成等规模的 RMQ 问题。

实现

" 参考代码"

```cpp
int dfn[N << 1], pos[N], tot, st[30][(N << 1) + 2],
    rev[30][(N << 1) + 2];  // rev 表示最小深度对应的节点编号

void dfs(int cur, int dep) {
  dfn[++tot] = cur;
  depth[tot] = dep;
  pos[cur] = tot;
  for (int i = head[t]; i; i = side[i].next) {
    int v = side[i].to;
    if (!pos[v]) {
      dfs(v, dep + 1);
      dfn[++tot] = cur, depth[tot] = dep;
    }
  }
}

void init() {
  for (int i = 2; i <= tot + 1; ++i)
    lg[i] = lg[i >> 1] + 1;  // 预处理 lg 代替库函数 log2 来优化常数
  for (int i = 1; i <= tot; i++) st[0][i] = depth[i], rev[0][i] = dfn[i];
  for (int i = 1; i <= lg[tot]; i++)
    for (int j = 1; j + (1 << i) - 1 <= tot; j++)
      if (st[i - 1][j] < st[i - 1][j + (1 << i - 1)])
        st[i][j] = st[i - 1][j], rev[i][j] = rev[i - 1][j];
      else
        st[i][j] = st[i - 1][j + (1 << i - 1)],
        rev[i][j] = rev[i - 1][j + (1 << i - 1)];
}

int query(int l, int r) {
  int k = lg[r - l + 1];
  return st[k][l] < st[k][r + 1 - (1 << k)] ? rev[k][l]
                                            : rev[k][r + 1 - (1 << k)];
}
```

当我们需要查询某点对 (u,v) 的 LCA 时，查询区间 $[\min\{pos[u],pos[v]\},\max\{pos[u],pos[v]\}]$ 上最小值的所代表的节点即可。

若使用 ST 表来解决 RMQ 问题，那么该算法不支持在线修改，预处理的时间复杂度为 $O(n\log n)$，每次查询 LCA 的时间复杂度为 $O(1)$。

树链剖分

LCA 为两个游标跳转到同一条重链上时深度较小的那个游标所指向的点。

树链剖分的预处理时间复杂度为 $O(n)$，单次查询的时间复杂度为 $O(\log n)$，并且常数较小。

动态树

设连续两次 access 操作的点分别为 `u` 和 `v`，则第二次 access 操作返回的点即为 `u` 和 `v` 的 LCA.

在无 link 和 cut 等操作的情况下，使用 link cut tree 单次查询的时间复杂度为 $O(\log n)$。

标准 RMQ

前面讲到了借助欧拉序将 LCA 问题转化为 RMQ 问题，其瓶颈在于 RMQ。如果能做到 $O(n) \sim O(1)$ 求解 RMQ，那么也就能做到 $O(n) \sim O(1)$ 求解 LCA。

注意到欧拉序满足相邻两数之差为 1 或者 -1，所以可以使用 $O(n) \sim O(1)$ 的 加减 1RMQ 来做。

时间复杂度 $O(n) \sim O(1)$，空间复杂度 $O(n)$，支持在线查询，常数较大。

例题 Luogu P3379【模板】最近公共祖先（LCA）[4]

"参考代码"

```cpp
#include <bits/stdc++.h>
using namespace std;

const int N = 5e5 + 5;

struct PlusMinusOneRMQ {  // RMQ
  // Copyright (C) 2018 Skqliao. All rights served.
  const static int M = 9;

  int blocklen, block, Minv[N], F[N / M * 2 + 5][M << 1], T[N], f[1 << M][M][M],
      S[N];

  void init(int n) {  // 初始化
    blocklen = std::max(1, (int)(log(n * 1.0) / log(2.0)) / 2);
    block = n / blocklen + (n % blocklen > 0);
    int total = 1 << (blocklen - 1);
    for (int i = 0; i < total; i++) {
      for (int l = 0; l < blocklen; l++) {
        f[i][l][l] = l;
        int now = 0, minv = 0;
        for (int r = l + 1; r < blocklen; r++) {
          f[i][l][r] = f[i][l][r - 1];
          if ((1 << (r - 1)) & i) {
            now++;
          } else {
            now--;
            if (now < minv) {
              minv = now;
              f[i][l][r] = r;
            }
          }
        }
      }
    }
    T[1] = 0;
    for (int i = 2; i < N; i++) {
      T[i] = T[i - 1];
      if (!(i & (i - 1))) {
        T[i]++;
      }
    }
  }
```

```
  void initmin(int a[], int n) {
    for (int i = 0; i < n; i++) {
      if (i % blocklen == 0) {
        Minv[i / blocklen] = i;
        S[i / blocklen] = 0;
      } else {
        if (a[i] < a[Minv[i / blocklen]]) {
          Minv[i / blocklen] = i;
        }
        if (a[i] > a[i - 1]) {
          S[i / blocklen] |= 1 << (i % blocklen - 1);
        }
      }
    }
    for (int i = 0; i < block; i++) {
      F[i][0] = Minv[i];
    }
    for (int j = 1; (1 << j) <= block; j++) {
      for (int i = 0; i + (1 << j) - 1 < block; i++) {
        int b1 = F[i][j - 1], b2 = F[i + (1 << (j - 1))][j - 1];
        F[i][j] = a[b1] < a[b2] ? b1 : b2;
      }
    }
  }

  int querymin(int a[], int L, int R) {
    int idl = L / blocklen, idr = R / blocklen;
    if (idl == idr)
      return idl * blocklen + f[S[idl]][L % blocklen][R % blocklen];
    else {
      int b1 = idl * blocklen + f[S[idl]][L % blocklen][blocklen - 1];
      int b2 = idr * blocklen + f[S[idr]][0][R % blocklen];
      int buf = a[b1] < a[b2] ? b1 : b2;
      int c = T[idr - idl - 1];
      if (idr - idl - 1) {
        int b1 = F[idl + 1][c];
        int b2 = F[idr - 1 - (1 << c) + 1][c];
        int b = a[b1] < a[b2] ? b1 : b2;
        return a[buf] < a[b] ? buf : b;
      }
      return buf;
    }
  }
} rmq;

int n, m, s;

struct Edge {
  int v, nxt;
} e[N * 2];

int tot, head[N];

void init(int n) {
```

```cpp
  tot = 0;
  fill(head, head + n + 1, 0);
}

void addedge(int u, int v) {  // 加边
  ++tot;
  e[tot] = (Edge){v, head[u]};
  head[u] = tot;

  ++tot;
  e[tot] = (Edge){u, head[v]};
  head[v] = tot;
}

int dfs_clock, dfn[N * 2], dep[N * 2], st[N];

void dfs(int u, int fa, int d) {
  st[u] = dfs_clock;

  dfn[dfs_clock] = u;
  dep[dfs_clock] = d;
  ++dfs_clock;

  int v;
  for (int i = head[u]; i; i = e[i].nxt) {
    v = e[i].v;
    if (v == fa) continue;
    dfs(v, u, d + 1);
    dfn[dfs_clock] = u;
    dep[dfs_clock] = d;
    ++dfs_clock;
  }
}

void build_lca() {  // like init
  rmq.init(dfs_clock);
  rmq.initmin(dep, dfs_clock);
}

int LCA(int u, int v) {  // 求解 LCA，看题解用 RMQ 的方法
  int l = st[u], r = st[v];
  if (l > r) swap(l, r);
  return dfn[rmq.querymin(dep, l, r)];
}

int main() {
  scanf("%d %d %d", &n, &m, &s);

  init(n);
  int u, v;
  for (int i = 1; i <= n - 1; ++i) {
    scanf("%d %d", &u, &v);
    addedge(u, v);
  }
```

```
  dfs_clock = 0;
  dfs(s, s, 0);

  build_lca();

  for (int i = 1; i <= m; ++i) {
    scanf("%d %d", &u, &v);
    printf("%d\n", LCA(u, v));
  }

  return 0;
}
```

习题

- 祖孙询问[5]
- 货车运输[6]
- 点的距离[7]

参考资料与注释

[1] wcipeg

[2] HDU 2586 How far away?

[3] Gabow 和 Tarjan 于 1983 年的论文

[4] Luogu P3379【模板】最近公共祖先（LCA）

[5] 祖孙询问

[6] 货车运输

[7] 点的距离

11.6.4 树的重心

Authors: Ir1d, Marcythm, LucienShui, Anguei, H-J-Granger, CornWorld, ttzc

定义

如果在树中选择某个节点并删除，这棵树将分为若干棵子树，统计子树节点数并记录最大值。取遍树上所有节点，使此最大值取到最小的节点被称为整个树的重心。

（这里以及下文中的「子树」若无特殊说明都是指无根树的子树，即包括「向上」的那棵子树，并且不包括整棵树自身。）

性质

- 树的重心如果不唯一，则至多有两个，且这两个重心相邻。
- 以树的重心为根时，所有子树的大小都不超过整棵树大小的一半。
- 树中所有点到某个点的距离和中，到重心的距离和是最小的；如果有两个重心，那么到它们的距离和一样。
- 把两棵树通过一条边相连得到一棵新的树，那么新的树的重心在连接原来两棵树的重心的路径上。
- 在一棵树上添加或删除一个叶子，那么它的重心最多只移动一条边的距离。

求法

在 DFS 中计算每个子树的大小，记录「向下」的子树的最大大小，利用总点数 - 当前子树（这里的子树指有根树的子树）的大小得到「向上」的子树的大小，然后就可以依据定义找到重心了。

" 参考代码"

```
// 这份代码默认节点编号从 1 开始，即 i   [1,n]
int size[MAXN],  // 这个节点的「大小」(所有子树上节点数 + 该节点)
    weight[MAXN],  // 这个节点的「重量」，即所有子树「大小」的最大值
    centroid[2];  // 用于记录树的重心 (存的是节点编号)

void GetCentroid(int cur, int fa) {  // cur 表示当前节点 (current)
  size[cur] = 1;
  weight[cur] = 0;
  for (int i = head[cur]; i != -1; i = e[i].nxt) {
    if (e[i].to != fa) {  // e[i].to 表示这条有向边所通向的节点。
      GetCentroid(e[i].to, cur);
      size[cur] += size[e[i].to];
      weight[cur] = max(weight[cur], size[e[i].to]);
    }
  }
  weight[cur] = max(weight[cur], n - size[cur]);
  if (weight[cur] <= n / 2) {  // 依照树的重心的定义统计
    centroid[centroid[0] != 0] = cur;
  }
}
```

例题

"Codeforces Round 359 (Div. 1) B. Kay and Snowflake[1]"

给定一棵有根树，求出每一棵子树（有根树意义下且包含整颗树本身）的重心是哪一个节点。

" 解题思路"

本题中子树无特殊说明指的是有根树意义下且包含整颗树本身的「向下」的子树。

```
根据第四条性质，对于一棵以点 $u$ 为根的子树，其重心一定在所有以 $u$ 的直接子节点为
根的子树的重心到点 $u$ 的路径上。

类似于上文提到的 DFS 求重心方法，对于每棵以节点 $u$ 为根的子树，先求出所有以其直接
子节点为根的子树的重心 (叶子节点的重心是其本身)，然后向上判断路径上的节点是不是重心
即可。
```

```
时间复杂度为 $O(n)$ 可以求出所有子树的重心。
```

" 参考代码"

```cpp
#include <bits/stdc++.h>
using namespace std;

const int N = 3e5 + 5;

int n, q;  // 点数，询问数
int fa[N];
vector<int> son[N];
int siz[N],     // 子树大小
    ans[N],     // 以节点 u 为根的子树重心是 ans[u]
    weight[N];  // 节点重量

void dfs(int u) {
  siz[u] = 1, ans[u] = u;
  for (int v : son[u]) {
    dfs(v);
    siz[u] += siz[v];
    weight[u] = max(weight[u], siz[v]);
  }
  for (int v : son[u]) {
    int p = ans[v];
    while (p != u) {
      if (max(weight[p], siz[u] - siz[p]) <= siz[u] / 2) {
        ans[u] = p;
        break;
      } else
        p = fa[p];
    }
  }
}

signed main() {
  ios::sync_with_stdio(0);
  cin >> n >> q;
  for (int v = 2; v <= n; v++) cin >> fa[v], son[fa[v]].push_back(v);
  dfs(1);
  while (q--) {
    int u;
    cin >> u;
    cout << ans[u] << '\n';
  }
  return 0;
}
```

参考

http://fanhq666.blog.163.com/blog/static/81943426201172472943638/[2](博客园转载[3]，Internet Archive[4])

https://blog.csdn.net/weixin_43810158/article/details/88391828[5]

https://www.cnblogs.com/zinthos/p/3899075.html[6]

https://www.cnblogs.com/suxxsfe/p/13543253.html[7]

《信息学奥林匹克辞典》2.4.7.11 章 1. 树的重心

习题

- POJ 1655 Balancing Art[8]（模板题）
- 洛谷　P1364　医院设置[9]
- Codeforces 1406C Link Cut Centroids[10]
- Codeforces 708C Centroids[11]

参考资料与注释

[1] Codeforces Round 359 (Div. 1) B. Kay and Snowflake

[2] http://fanhq666.blog.163.com/blog/static/81943426201172472943638/

[3] 博客园转载

[4] Internet Archive

[5] https://blog.csdn.net/weixin_43810158/article/details/88391828

[6] https://www.cnblogs.com/zinthos/p/3899075.html

[7] https://www.cnblogs.com/suxxsfe/p/13543253.html

[8] POJ 1655 Balancing Art

[9] 洛谷 P1364 医院设置

[10] Codeforces 1406C Link Cut Centroids

[11] Codeforces 708C Centroids

11.6.5　树链剖分

Authors: GoodCoder666, Ir1d, Marcythm, ouuan, hsfzLZH1, Xeonacid, greyqz, Chrogeek, ftxj, sshwy, LuoshuiTianyi, hyp1231

树链剖分的思想及能解决的问题

树链剖分用于将树分割成若干条链的形式，以维护树上路径的信息。

具体来说，将整棵树剖分为若干条链，使它组合成线性结构，然后用其他的数据结构维护信息。

树链剖分（树剖/链剖）有多种形式，如**重链剖分**，**长链剖分**和用于 Link/cut Tree 的剖分（有时被称作「实链剖分」），大多数情况下（没有特别说明时），「树链剖分」都指「重链剖分」。

重链剖分可以将树上的任意一条路径划分成不超过 $O(\log n)$ 条连续的链，每条链上的点深度互不相同（即是自底向上的一条链，链上所有点的 LCA 为链的一个端点）。

重链剖分还能保证划分出的每条链上的节点 DFS 序连续，因此可以方便地用一些维护序列的数据结构（如线段树）来维护树上路径的信息。

如：

1. 修改**树上两点之间的路径上**所有点的值。
2. 查询**树上两点之间的路径上**节点权值的**和/极值/其它（在序列上可以用数据结构维护，便于合并的信息）**。

除了配合数据结构来维护树上路径信息，树剖还可以用来 $O(\log n)$（且常数较小）地求 LCA。在某些题目中，还可以利用其性质来灵活地运用树剖。

重链剖分

我们给出一些定义：

定义**重子节点**表示其子节点中子树最大的子结点。如果有多个子树最大的子结点，取其一。如果没有子节点，就无重子节点。

定义**轻子节点**表示剩余的所有子结点。

从这个结点到重子节点的边为**重边**。

到其他轻子节点的边为**轻边**。

若干条首尾衔接的重边构成**重链**。

把落单的结点也当作重链，那么整棵树就被剖分成若干条重链。

如图：

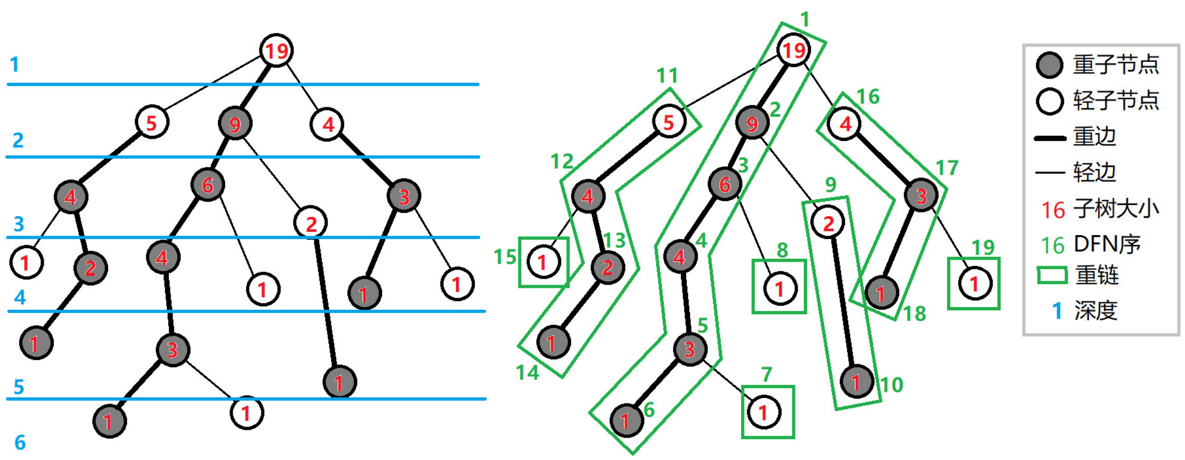


图 11.18 HLD

实现

树剖的实现分两个 DFS 的过程。伪代码如下：

第一个 DFS 记录每个结点的父节点（father）、深度（deep）、子树大小（size）、重子节点（hson）。

```
TREE-BUILD (u, dep)
u.hson ← 0
u.hson.size ← 0
u.deep ← dep
u.size ← 1
for each u's son v
    u.size ← u.size + TREE-BUILD (v, dep + 1)
    v.father ← u
    if v.size > u.hson.size
        u.hson ← v
return u.size
```

第二个 DFS 记录所在链的链顶（top，应初始化为结点本身）、重边优先遍历时的 DFS 序（dfn）、DFS 序对应的节点编号（rank）。

```
TREE-DECOMPOSITION (u, top)
u.top ← top
tot ← tot + 1
u.dfn ← tot
rank(tot) ← u
if u.hson is not 0
    TREE-DECOMPOSITION (u.hson, top)
    for each u's son v
        if v is not u.hson
            TREE-DECOMPOSITION (v, v)
```

以下为代码实现。

我们先给出一些定义：

- $fa(x)$ 表示节点 x 在树上的父亲。
- $dep(x)$ 表示节点 x 在树上的深度。
- $siz(x)$ 表示节点 x 的子树的节点个数。
- $son(x)$ 表示节点 x 的**重儿子**。
- $top(x)$ 表示节点 x 所在**重链**的顶部节点（深度最小）。
- $dfn(x)$ 表示节点 x 的 **DFS 序**，也是其在线段树中的编号。
- $rnk(x)$ 表示 DFS 序所对应的节点编号，有 $rnk(dfn(x)) = x$。

我们进行两遍 DFS 预处理出这些值，其中第一次 DFS 求出 $fa(x)$，$dep(x)$，$siz(x)$，$son(x)$，第二次 DFS 求出 $top(x)$，$dfn(x)$，$rnk(x)$。

```
void dfs1(int o) {
  son[o] = -1;
  siz[o] = 1;
  for (int j = h[o]; j; j = nxt[j])
    if (!dep[p[j]]) {
      dep[p[j]] = dep[o] + 1;
      fa[p[j]] = o;
      dfs1(p[j]);
      siz[o] += siz[p[j]];
      if (son[o] == -1 || siz[p[j]] > siz[son[o]]) son[o] = p[j];
    }
```

```
}

void dfs2(int o, int t) {
  top[o] = t;
  cnt++;
  dfn[o] = cnt;
  rnk[cnt] = o;
  if (son[o] == -1) return;
  dfs2(son[o], t);  // 优先对重儿子进行 DFS，可以保证同一条重链上的点 DFS 序连续
  for (int j = h[o]; j; j = nxt[j])
    if (p[j] != son[o] && p[j] != fa[o]) dfs2(p[j], p[j]);
}
```

重链剖分的性质

树上每个节点都属于且仅属于一条重链。

重链开头的结点不一定是重子节点（因为重边是对于每一个结点都有定义的）。

所有的重链将整棵树**完全剖分**。

在剖分时**重边优先遍历**，最后树的 DFS 序上，重链内的 DFS 序是连续的。按 DFN 排序后的序列即为剖分后的链。

一颗子树内的 DFS 序是连续的。

可以发现，当我们向下经过一条**轻边**时，所在子树的大小至少会除以二。

因此，对于树上的任意一条路径，把它拆分成从 LCA 分别向两边往下走，分别最多走 $O(\log n)$ 次，因此，树上的每条路径都可以被拆分成不超过 $O(\log n)$ 条重链。

常见应用

路径上维护

用树链剖分求树上两点路径权值和，伪代码如下：

TREE-PATH-SUM (u, v)
1 $tot \leftarrow 0$
2 **while** $u.top$ is not $v.top$
3   **if** $u.top.deep < v.top.deep$
4   SWAP(u, v)
5     $tot \leftarrow tot +$ sum of values between u and $u.top$
6     $u \leftarrow u.top.father$
7 $tot \leftarrow tot +$ sum of values between u and v
8 **return** tot

链上的 DFS 序是连续的，可以使用线段树、树状数组维护。

每次选择深度较大的链往上跳，直到两点在同一条链上。

同样的跳链结构适用于维护、统计路径上的其他信息。

子树维护

有时会要求，维护子树上的信息，譬如将以 x 为根的子树的所有结点的权值增加 v。

在 DFS 搜索的时候，子树中的结点的 DFS 序是连续的。

每一个结点记录 bottom 表示所在子树连续区间末端的结点。

这样就把子树信息转化为连续的一段区间信息。

求最近公共祖先

不断向上跳重链，当跳到同一条重链上时，深度较小的结点即为 LCA。

向上跳重链时需要先跳所在重链顶端深度较大的那个。

参考代码：

```
int lca(int u, int v) {
  while (top[u] != top[v]) {
    if (dep[top[u]] > dep[top[v]])
      u = fa[top[u]];
    else
      v = fa[top[v]];
  }
  return dep[u] > dep[v] ? v : u;
}
```

" 怎么有理有据地卡树剖"

一般情况下树剖的 $O(\log n)$ 常数不满很难卡，如果要卡只能建立二叉树深度低。

于是我们可以考虑折中方案。

我们建立一颗 $\sqrt{n}$ 个节点的二叉树。对于每个节点到其儿子的边，我们将其替换成一条长度为 $\sqrt{n}$ 的链。

这样子我们可以将随机询问轻重链切换次数卡到平均 $\frac{\log n}{2}$ 次，同时有 $O(\sqrt{n}\log n)$ 的深度。

加上若干随机叶子看上去可以卡树剖。但是树剖常数小有可能卡不掉。

例题

「ZJOI2008」 树的统计[1]

题目大意

对一棵有 n 个节点，节点带权值的静态树，进行三种操作共 q 次：

1. 修改单个节点的权值；
2. 查询 u 到 v 的路径上的最大权值；
3. 查询 u 到 v 的路径上的权值之和。

保证 $1 \le n \le 30000$，$0 \le q \le 200000$。

解法

根据题面以及以上的性质，你的线段树需要维护三种操作：

1. 单点修改；
2. 区间查询最大值；
3. 区间查询和。

单点修改很容易实现。

由于子树的 DFS 序连续（无论是否树剖都是如此），修改一个节点的子树只用修改这一段连续的 DFS 序区间。

问题是如何修改/查询两个节点之间的路径。

考虑我们是如何用**倍增法求解 LCA** 的。首先我们**将两个节点提到同一高度，然后将两个节点一起向上跳**。对于树链剖分也可以使用这样的思想。

在向上跳的过程中，如果当前节点在重链上，向上跳到重链顶端，如果当前节点不在重链上，向上跳一个节点。如此直到两节点相同。沿途更新/查询区间信息。

对于每个询问，最多经过 $O(\log n)$ 条重链，每条重链上线段树的复杂度为 $O(\log n)$，因此总时间复杂度为

$O(n\log n+q\log^2 n)$。实际上重链个数很难达到 $O(\log n)$（可以用完全二叉树卡满），所以树剖在一般情况下常数较小。

给出一种代码实现：

```
// st 是线段树结构体
int querymax(int x, int y) {
  int ret = -inf, fx = top[x], fy = top[y];
  while (fx != fy) {
    if (dep[fx] >= dep[fy])
      ret = max(ret, st.query1(1, 1, n, dfn[fx], dfn[x])), x = fa[fx];
    else
      ret = max(ret, st.query1(1, 1, n, dfn[fy], dfn[y])), y = fa[fy];
    fx = top[x];
    fy = top[y];
  }
  if (dfn[x] < dfn[y])
    ret = max(ret, st.query1(1, 1, n, dfn[x], dfn[y]));
  else
    ret = max(ret, st.query1(1, 1, n, dfn[y], dfn[x]));
  return ret;
}
```

" 参考代码"

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#define lc o << 1
#define rc o << 1 | 1
const int maxn = 60010;
const int inf = 2e9;
int n, a, b, w[maxn], q, u, v;
int cur, h[maxn], nxt[maxn], p[maxn];
int siz[maxn], top[maxn], son[maxn], dep[maxn], fa[maxn], dfn[maxn], rnk[maxn],
    cnt;
char op[10];

void add_edge(int x, int y) {  // 加边
  cur++;
  nxt[cur] = h[x];
  h[x] = cur;
  p[cur] = y;
}

struct SegTree {
  int sum[maxn * 4], maxx[maxn * 4];

  void build(int o, int l, int r) {
    if (l == r) {
      sum[o] = maxx[o] = w[rnk[l]];
      return;
    }
    int mid = (l + r) >> 1;
    build(lc, l, mid);
    build(rc, mid + 1, r);
```

```
    sum[o] = sum[lc] + sum[rc];
    maxx[o] = std::max(maxx[lc], maxx[rc]);
  }

  int query1(int o, int l, int r, int ql, int qr) {  // 查询 max
    if (l > qr || r < ql) return -inf;
    if (ql <= l && r <= qr) return maxx[o];
    int mid = (l + r) >> 1;
    return std::max(query1(lc, l, mid, ql, qr), query1(rc, mid + 1, r, ql, qr));
  }

  int query2(int o, int l, int r, int ql, int qr) {  // 查询 sum
    if (l > qr || r < ql) return 0;
    if (ql <= l && r <= qr) return sum[o];
    int mid = (l + r) >> 1;
    return query2(lc, l, mid, ql, qr) + query2(rc, mid + 1, r, ql, qr);
  }

  void update(int o, int l, int r, int x, int t) {  // 更新
    if (l == r) {
      maxx[o] = sum[o] = t;
      return;
    }
    int mid = (l + r) >> 1;
    if (x <= mid)
      update(lc, l, mid, x, t);  // 左右分别更新
    else
      update(rc, mid + 1, r, x, t);
    sum[o] = sum[lc] + sum[rc];
    maxx[o] = std::max(maxx[lc], maxx[rc]);
  }
} st;

void dfs1(int o) {
  son[o] = -1;
  siz[o] = 1;
  for (int j = h[o]; j; j = nxt[j])
    if (!dep[p[j]]) {
      dep[p[j]] = dep[o] + 1;
      fa[p[j]] = o;
      dfs1(p[j]);
      siz[o] += siz[p[j]];
      if (son[o] == -1 || siz[p[j]] > siz[son[o]]) son[o] = p[j];
    }
}

void dfs2(int o, int t) {
  top[o] = t;
  cnt++;
  dfn[o] = cnt;
  rnk[cnt] = o;
  if (son[o] == -1) return;
  dfs2(son[o], t);
  for (int j = h[o]; j; j = nxt[j])
```

```
    if (p[j] != son[o] && p[j] != fa[o]) dfs2(p[j], p[j]);
}

int querymax(int x, int y) {  // 查询，看 main 函数理解一下
  int ret = -inf, fx = top[x], fy = top[y];
  while (fx != fy) {
    if (dep[fx] >= dep[fy])
      ret = std::max(ret, st.query1(1, 1, n, dfn[fx], dfn[x])), x = fa[fx];
    else
      ret = std::max(ret, st.query1(1, 1, n, dfn[fy], dfn[y])), y = fa[fy];
    fx = top[x];
    fy = top[y];
  }
  if (dfn[x] < dfn[y])
    ret = std::max(ret, st.query1(1, 1, n, dfn[x], dfn[y]));
  else
    ret = std::max(ret, st.query1(1, 1, n, dfn[y], dfn[x]));
  return ret;
}

int querysum(int x, int y) {
  int ret = 0, fx = top[x], fy = top[y];
  while (fx != fy) {
    if (dep[fx] >= dep[fy])
      ret += st.query2(1, 1, n, dfn[fx], dfn[x]), x = fa[fx];
    else
      ret += st.query2(1, 1, n, dfn[fy], dfn[y]), y = fa[fy];
    fx = top[x];
    fy = top[y];
  }
  if (dfn[x] < dfn[y])
    ret += st.query2(1, 1, n, dfn[x], dfn[y]);
  else
    ret += st.query2(1, 1, n, dfn[y], dfn[x]);
  return ret;
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i < n; i++)
    scanf("%d%d", &a, &b), add_edge(a, b), add_edge(b, a);
  for (int i = 1; i <= n; i++) scanf("%d", w + i);
  dep[1] = 1;
  dfs1(1);
  dfs2(1, 1);
  st.build(1, 1, n);
  scanf("%d", &q);
  while (q--) {
    scanf("%s%d%d", op, &u, &v);
    if (!strcmp(op, "CHANGE")) st.update(1, 1, n, dfn[u], v);
    if (!strcmp(op, "QMAX")) printf("%d\n", querymax(u, v));
    if (!strcmp(op, "QSUM")) printf("%d\n", querysum(u, v));
  }
  return 0;
}
```

Nauuo and Binary Tree[2]

这是一道交互题，也是树剖的非传统应用。

题目大意

有一棵以 1 为根的二叉树，你可以询问任意两点之间的距离，求出每个点的父亲。

节点数不超过 3000，你最多可以进行 30000 次询问。

解法

首先可以通过 $n-1$ 次询问确定每个节点的深度。

然后考虑按深度从小到大确定每个节点的父亲，这样的话确定一个节点的父亲时其所有祖先一定都是已知的。

确定一个节点的父亲之前，先对树已知的部分进行重链剖分。

假设我们需要在子树 u 中找节点 k 所在的位置，我们可以询问 k 与 u 所在重链的尾端的距离，就可以进一步确定 k 的位置，具体见图：

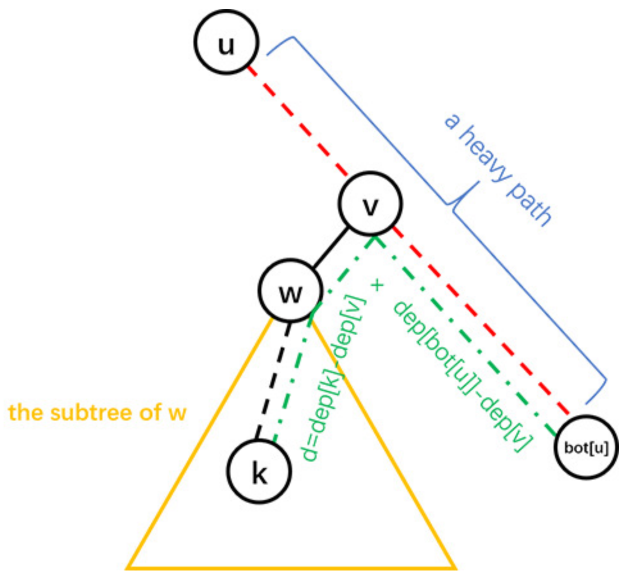


图 11.19

其中红色虚线是一条重链，d 是询问的结果即 $dis(k, bot[u])$，v 的深度为 $(dep[k]+dep[bot[u]]-d)/2$。

这样的话，如果 v 只有一个儿子，k 的父亲就是 v，否则可以递归地在 w 的子树中找 k 的父亲。

时间复杂度 $O(n^2)$，询问复杂度 $O(n\log n)$。

具体地，设 $T(n)$ 为最坏情况下在一棵大小为 n 的树中找到一个新节点的位置所需的询问次数，可以得到：

$$T(n) \le \begin{cases} 0 & n = 1 \\ T\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + 1 & n \ge 2 \end{cases}$$

$2999+\sum_{i=1}^{2999} T(i) \le 29940$，事实上这个上界是可以通过构造数据达到的，然而只要进行一些随机扰动（如对深度进行排序时使用不稳定的排序算法），询问次数很难超过 21000 次。

" 参考代码"

```cpp
#include <algorithm>
#include <cstdio>
#include <iostream>

using namespace std;
```

```cpp
const int N = 3010;

int n, fa[N], ch[N][2], dep[N], siz[N], son[N], bot[N], id[N];

int query(int u, int v) {
  printf("? %d %d\n", u, v);
  fflush(stdout);
  int d;
  scanf("%d", &d);
  return d;
}

void setFather(int u, int v) {
  fa[v] = u;
  if (ch[u][0])
    ch[u][1] = v;
  else
    ch[u][0] = v;
}

void dfs(int u) {
  if (ch[u][0]) dfs(ch[u][0]);
  if (ch[u][1]) dfs(ch[u][1]);

  siz[u] = siz[ch[u][0]] + siz[ch[u][1]] + 1;

  if (ch[u][1])
    son[u] = int(siz[ch[u][0]] < siz[ch[u][1]]);
  else
    son[u] = 0;

  if (ch[u][son[u]])
    bot[u] = bot[ch[u][son[u]]];
  else
    bot[u] = u;
}

void solve(int u, int k) {
  if (!ch[u][0]) {
    setFather(u, k);
    return;
  }
  int d = query(k, bot[u]);
  int v = bot[u];
  while (dep[v] > (dep[k] + dep[bot[u]] - d) / 2) v = fa[v];
  int w = ch[v][son[v] ^ 1];
  if (w)
    solve(w, k);
  else
    setFather(v, k);
}

int main() {
  int i;
```

```
  scanf("%d", &n);

  for (i = 2; i <= n; ++i) {
    id[i] = i;
    dep[i] = query(1, i);
  }

  sort(id + 2, id + n + 1, [](int x, int y) { return dep[x] < dep[y]; });

  for (i = 2; i <= n; ++i) {
    dfs(1);
    solve(1, id[i]);
  }

  printf("!");
  for (i = 2; i <= n; ++i) printf(" %d", fa[i]);
  printf("\n");
  fflush(stdout);

  return 0;
}
```

长链剖分

长链剖分本质上就是另外一种链剖分方式。

定义**重子节点**表示其子节点中子树深度最大的子结点。如果有多个子树最大的子结点，取其一。如果没有子节点，就无重子节点。

定义**轻子节点**表示剩余的子结点。

从这个结点到重子节点的边为**重边**。

到其他轻子节点的边为**轻边**。

若干条首尾衔接的重边构成**重链**。

把落单的结点也当作重链，那么整棵树就被剖分成若干条重链。

如图（这种剖分方式既可以看成重链剖分也可以看成长链剖分）：

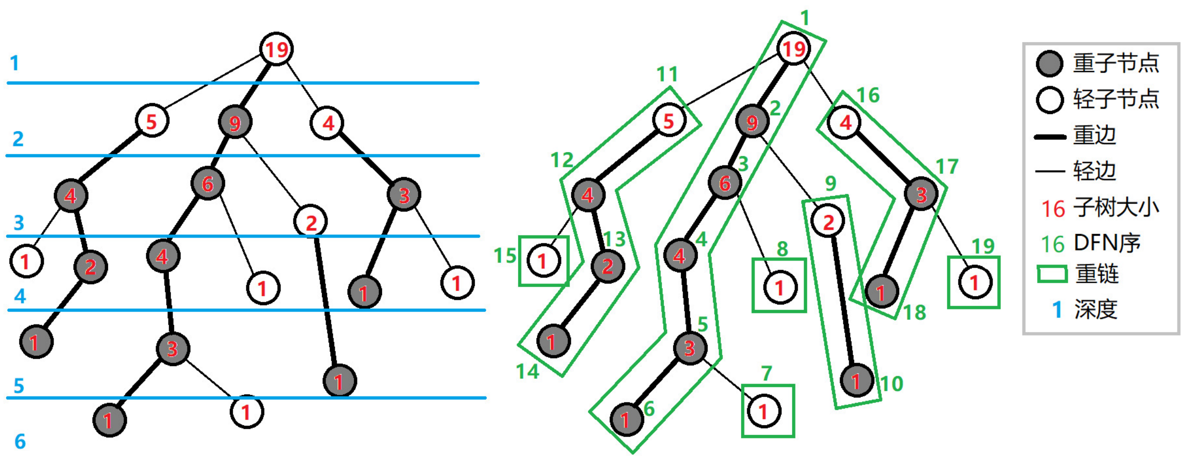


图 11.20 HLD

长链剖分实现方式和重链剖分类似，这里就不再展开。

常见应用

首先，我们发现长链剖分从一个节点到根的路径的轻边切换条数是 $\sqrt{n}$ 级别的。

" 如何构造数据将轻重边切换次数卡满"

我们可以构造这么一颗二叉树 T：

假设构造的二叉树参数为 D。

若 $D \neq 0$, 则在左儿子构造一颗参数为 $D-1$ 的二叉树，在右儿子构造一个长度为 $2D-1$ 的链。

若 $D = 0$, 则我们可以直接构造一个单独叶节点，并且结束调用。

这样子构造一定可以将单独叶节点到根的路径全部为轻边且需要 D^2 级别的节点数。

取 $D = \sqrt{n}$ 即可。

长链剖分优化 DP

一般情况下可以使用长链剖分来优化的 DP 会有一维状态为深度维。

我们可以考虑使用长链剖分优化树上 DP。

具体的，我们每个节点的状态直接继承其重儿子的节点状态，同时将轻儿子的 DP 状态暴力合并。

"CF 1009F[3]"

我们设 $f_{i,j}$ 表示在子树 i 内，和 i 距离为 j 的点数。

直接暴力转移时间复杂度为 $O(n^2)$

我们考虑每次转移我们直接继承重儿子的 DP 数组和答案，并且考虑在此基础上进行更新。

首先我们需要将重儿子的 DP 数组前面插入一个元素 1, 这代表着当前节点。

然后我们将所有轻儿子的 DP 数组暴力和当前节点的 DP 数组合并。

注意到因为轻儿子的 DP 数组长度为轻儿子所在重链长度，而所有重链长度和为 n。

也就是说，我们直接暴力合并轻儿子的总时间复杂度为 $O(n)$。

注意，一般情况下 DP 数组的内存分配为一条重链整体分配内存，链上不同的节点有不同的首位置指针。

DP 数组的长度我们可以根据子树最深节点算出。

例题参考代码：

```cpp
#include <bits/stdc++.h>
using namespace std;
const int N = 1000005;

struct edge {
  int to, next;
} e[N * 2];

int head[N], tot, n;
int d[N], fa[N], mx[N];
int *f[N], g[N], mxp[N];
int dfn[N];

void add(int x, int y) {
  e[++tot] = (edge){y, head[x]};
  head[x] = tot;
}

void dfs1(int x) {  // 第一次插入一个 1
```

```
  d[x] = 1;
  for (int i = head[x]; i; i = e[i].next)
    if (e[i].to != fa[x]) {
      fa[e[i].to] = x;
      dfs1(e[i].to);
      d[x] = max(d[x], d[e[i].to] + 1);
      if (d[e[i].to] > d[mx[x]]) mx[x] = e[i].to;
    }
}

void dfs2(int x) {  // 第二次合并
  dfn[x] = ++*dfn;
  f[x] = g + dfn[x];
  if (mx[x]) dfs2(mx[x]);
  for (int i = head[x]; i; i = e[i].next)
    if (e[i].to != fa[x] && e[i].to != mx[x]) dfs2(e[i].to);
}

void getans(int x) {  // 暴力合并算答案
  if (mx[x]) {
    getans(mx[x]);
    mxp[x] = mxp[mx[x]] + 1;
  }
  f[x][0] = 1;
  if (f[x][mxp[x]] <= 1) mxp[x] = 0;
  for (int i = head[x]; i; i = e[i].next)
    if (e[i].to != fa[x] && e[i].to != mx[x]) {
      getans(e[i].to);
      int len = d[e[i].to];
      for (int j = 0; j <= len - 1; j++) {
        f[x][j + 1] += f[e[i].to][j];
        if (f[x][j + 1] > f[x][mxp[x]]) mxp[x] = j + 1;
        if (f[x][j + 1] == f[x][mxp[x]] && j + 1 < mxp[x]) mxp[x] = j + 1;
      }
    }
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i < n; i++) {
    int x, y;
    scanf("%d%d", &x, &y);
    add(x, y);
    add(y, x);
  }
  dfs1(1);
  dfs2(1);
  getans(1);
  for (int i = 1; i <= n; i++) printf("%d\n", mxp[i]);
}
```

当然长链剖分优化 DP 技巧非常多，包括但是不仅限于打标记等等。这里不再展开。

参考 租酥雨的博客[4]。

长链剖分求 k 级祖先

即询问一个点向父亲跳 k 次跳到的节点。

首先我们假设我们已经预处理了每一个节点的 2^i 级祖先。

现在我们假设我们找到了询问节点的 2^i 级祖先满足 $2^i \le k < 2^{i+1}$。

我们考虑求出其所在重链的节点并且按照深度列入表格。假设重链长度为 d。

同时我们在预处理的时候找到每条重链的根节点的 1 到 d 级祖先，同样放入表格。

根据长链剖分的性质，$k - 2^i \le 2^i \le d$, 也就是说，我们可以 $O(1)$ 在这条重链的表格上求出的这个节点的 k 级祖先。

预处理需要倍增出 2^i 次级祖先，同时需要预处理每条重链对应的表格。

预处理复杂度 $O(n \log n)$, 询问复杂度 $O(1)$。

练习

「洛谷 P3379」【模板】最近公共祖先（LCA）[5]（树剖求 LCA 无需数据结构，可以用作练习）

「JLOI2014」松鼠的新家[6]（当然也可以用树上差分）

「HAOI2015」树上操作[7]

「洛谷 P3384」【模板】重链剖分 / 树链剖分[8]

「NOI2015」软件包管理器[9]

「SDOI2011」染色[10]

「SDOI2014」旅行[11]

「POI2014」Hotel 加强版[12]（长链剖分优化 DP）

攻略[13]（长链剖分优化贪心）

参考资料与注释

[1] 「ZJOI2008」树的统计

[2] Nauuo and Binary Tree

[3] CF 1009F

[4] 租酥雨的博客

[5] 「洛谷 P3379」【模板】最近公共祖先（LCA）

[6] 「JLOI2014」松鼠的新家

[7] 「HAOI2015」树上操作

[8] 「洛谷 P3384」【模板】重链剖分/树链剖分

[9] 「NOI2015」软件包管理器

[10] 「SDOI2011」染色

[11] 「SDOI2014」旅行

[12] 「POI2014」Hotel 加强版

[13] 攻略

11.6.6 树上启发式合并

Authors: abc1763613206, cesonic, Ir1d, MingqiHuang, xinchengo

引入

启发式算法是什么呢?

启发式算法是基于人类的经验和直观感觉，对一些算法的优化。

给个例子?

最常见的就是并查集的按秩合并了，有带按秩合并的并查集中，合并的代码是这样的:

```
void merge(int x, int y) {
  int xx = find(x), yy = find(y);
  if (size[xx] < size[yy]) swap(xx, yy);
  fa[yy] = xx;
  size[xx] += size[yy];
}
```

在这里，对于两个大小不一样的集合，我们将小的集合合并到大的集合中，而不是将大的集合合并到小的集合中。

为什么呢? 这个集合的大小可以认为是集合的高度（在正常情况下），而我们将集合高度小的并到高度大的显然有助于我们找到父亲

让高度小的树成为高度较大的树的子树，这个优化可以称为启发式合并算法。

算法内容

树上启发式合并（dsu on tree）对于某些树上离线问题可以速度大于等于大部分算法且更易于理解和实现的算法。

考虑下面的问题：树上数颜色[1]

" 例题引入"

给出一棵 n 个节点以 1 为根的树，节点 u 的颜色为 c_u，现在对于每个结点 u 询问 u 子树里一共出现了多少种不同的颜色。

$n \leq 2 \times 10^5$。

对于这种问题解决方式大多是运用大量的数据结构（树套树等），如果可以离线，询问的量巨大，是不是有更简单的方法?

树上莫队!

不行，莫队带根号，我要 log

过程

既然支持离线，考虑预处理后 $O(1)$ 输出答案。

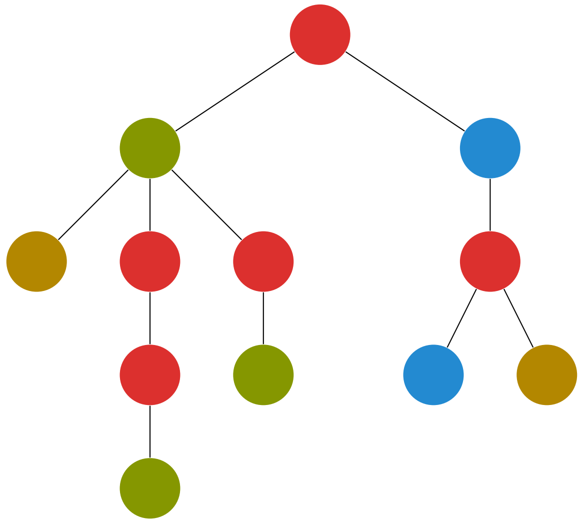


图 11.21 dsu-on-tree-1.png

直接暴力预处理的时间复杂度为 $O(n^2)$，即对每一个子节点进行一次遍历，每次遍历的复杂度显然与 n 同阶，有 n 个节点，故复杂度为 $O(n^2)$。

可以发现，每个节点的答案由其子树和其本身得到，考虑利用这个性质处理问题。

我们可以先预处理出每个节点子树的大小和它的重儿子，重儿子同树链剖分一样，是拥有节点最多子树的儿子，这个过程显然可以 $O(n)$ 完成

我们用 cnt[i] 表示颜色 i 的出现次数，ans[u] 表示结点 u 的答案。

遍历一个节点 u，我们按以下的步骤进行遍历：

1. 先遍历 u 的轻（非重）儿子，并计算答案，但**不保留遍历后它对 cnt 数组的影响**；
2. 遍历它的重儿子，**保留它对 cnt 数组的影响**；
3. 再次遍历 u 的轻儿子的子树结点，加入这些结点的贡献，以得到 u 的答案。

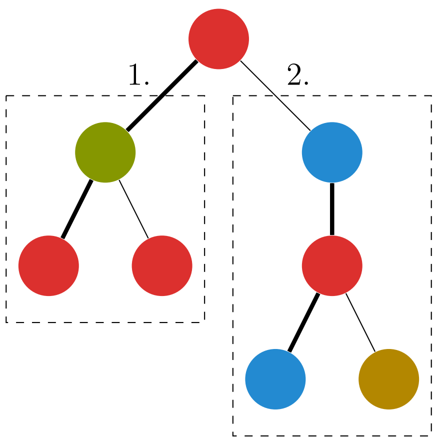


图 11.22 dsu-on-tree-2.png

上图是一个例子。

这样，对于一个节点，我们遍历了一次重子树，两次非重子树，显然是最划算的。

通过执行这个过程，我们获得了这个节点所有子树的答案。

为什么不合并第一步和第三步呢？因为 cnt 数组不能重复使用，否则空间会太大，需要在 $O(n)$ 的空间内完成。

显然若一个节点 u 被遍历了 x 次，则其重儿子会被遍历 x 次，轻儿子（如果有的话）会被遍历 $2x$ 次。

注意除了重儿子，每次遍历完 cnt 要清零。

证明

（对于不关心复杂度证明的，可以跳过不看）

我们像树链剖分一样定义重边和轻边（连向重儿子的为重边，其余为轻边）关于重儿子和重边的定义，可以见下图，对于一棵有 n 个节点的树：

根节点到树上任意节点的轻边数不超过 $\log n$ 条。我们设根到该节点有 x 条轻边该节点的子树大小为 y，显然轻边连接的子节点的子树大小小于父亲的一半（若大于一半就不是轻边了），则 $y < n/2^x$，显然 $n > 2^x$，所以 $x < \log n$。

又因为如果一个节点是其父亲的重儿子，则它的子树必定在它的兄弟之中最多，所以任意节点到根的路径上所有重边连接的父节点在计算答案时必定不会遍历到这个节点，所以一个节点的被遍历的次数等于它到根节点路径上的轻边数 $+1$（之所以要 $+1$ 是因为它本身要被遍历到），所以一个节点的被遍历次数 $= \log n + 1$，总时间复杂度则为 $O(n(\log n + 1)) = O(n \log n)$，输出答案花费 $O(m)$.

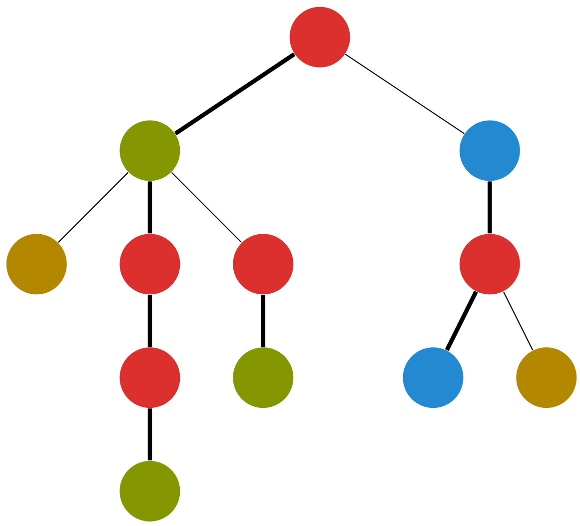


图 11.23 dsu-on-tree-3.png

图中标粗的即为重边，重边连向的子节点为重儿子

实现

" 实现"

```
#include <bits/stdc++.h>
using namespace std;

const int N = 2e5 + 5;

int n;

// g[u]: 存储与 u 相邻的结点
vector<int> g[N];

// sz: 子树大小
// big: 重儿子
// col: 结点颜色
// L[u]: 结点 u 的 DFS 序
// R[u]: 结点 u 子树中结点的 DFS 序的最大值
// Node[i]: DFS 序为 i 的结点
// ans: 存答案
```

```
// cnt[i]: 颜色为 i 的结点个数
// totColor: 目前出现过的颜色个数
int sz[N], big[N], col[N], L[N], R[N], Node[N], totdfn;
int ans[N], cnt[N], totColor;

void add(int u) {
  if (cnt[col[u]] == 0) ++totColor;
  cnt[col[u]]++;
}

void del(int u) {
  cnt[col[u]]--;
  if (cnt[col[u]] == 0) --totColor;
}

int getAns() { return totColor; }

void dfs0(int u, int p) {
  L[u] = ++totdfn;
  Node[totdfn] = u;
  sz[u] = 1;
  for (int v : g[u])
    if (v != p) {
      dfs0(v, u);
      sz[u] += sz[v];
      if (!big[u] || sz[big[u]] < sz[v]) big[u] = v;
    }
  R[u] = totdfn;
}

void dfs1(int u, int p, bool keep) {
  // 计算轻儿子的答案
  for (int v : g[u])
    if (v != p && v != big[u]) {
      dfs1(v, u, false);
    }
  // 计算重儿子答案并保留计算过程中的数据（用于继承）
  if (big[u]) {
    dfs1(big[u], u, true);
  }
  for (int v : g[u])
    if (v != p && v != big[u]) {
      // 子树结点的 DFS 序构成一段连续区间，可以直接遍历
      for (int i = L[v]; i <= R[v]; i++) {
        add(Node[i]);
      }
    }
  add(u);
  ans[u] = getAns();
  if (keep == false) {
    for (int i = L[u]; i <= R[u]; i++) {
      del(Node[i]);
    }
  }
```

```
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) scanf("%d", &col[i]);
  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d%d", &u, &v);
    g[u].push_back(v);
    g[v].push_back(u);
  }
  dfs0(1, 0);
  dfs1(1, 0, false);
  for (int i = 1; i <= n; i++) printf("%d%c", ans[i], " \n"[i == n]);
  return 0;
}
```

运用

1. 某些出题人设置的正解是 dsu on tree 的题

   如 CF741D[2]。给一棵树，每个节点的权值是'a' 到'v' 的字母，每次询问要求在一个子树找一条路径，使该路径包含的字符排序后成为回文串。

   因为是排列后成为回文串，所以一个字符出现了两次相当于没出现，也就是说，这条路径满足**最多有一个字符出现奇数次**。

   正常做法是对每一个节点 dfs，每到一个节点就强行枚举所有字母找到和它异或后结果为 1 的个数大于 1 的路径，再取最长值，这样是 $O(n^2 \log n)$ 的，可以用 dsu on tree 优化到 $O(n \log^2 n)$。关于具体做法，可以参考下面的扩展阅读

2. 可以用 dsu 乱搞的题

   可以水一些树套树的部分分（没有修改操作），而且 dsu 的复杂度优于树上莫队的 $O(n\sqrt{m})$

练习题

CF600E Lomsat gelral[3]

题意翻译：树的节点有颜色，一种颜色占领了一个子树，当且仅当没有其他颜色在这个子树中出现得比它多。求占领每个子树的所有颜色之和。

UOJ284 快乐游戏鸡[4]

CF1709E XOR Tree[5]

参考资料/扩展阅读

CF741D 作者介绍的 dsu on tree[6]

这位作者的题解[7]

参考资料与注释

[1] 树上数颜色

[2] CF741D

[3] CF600E Lomsat gelral

[4] UOJ284 快乐游戏鸡

[5] CF1709E XOR Tree

[6] CF741D 作者介绍的 dsu on tree

[7] 这位作者的题解

11.6.7 虚树

Authors: HeRaNO, Ir1d, konnyakuxzy, ksyx, Xeonacid, konnyakuxzy, greyqz, sshwy

引入

"「SDOI2011」 消耗战[1]"

题目描述

在一场战争中，战场由 n 个岛屿和 $n-1$ 个桥梁组成，保证每两个岛屿间有且仅有一条路径可达。现在，我军已经侦查到敌军的总部在编号为 1 的岛屿，而且他们已经没有足够多的能源维系战斗，我军胜利在望。已知在其他 k 个岛屿上有丰富能源，为了防止敌军获取能源，我军的任务是炸毁一些桥梁，使得敌军不能到达任何能源丰富的岛屿。由于不同桥梁的材质和结构不同，所以炸毁不同的桥梁有不同的代价，我军希望在满足目标的同时使得总代价最小。

侦查部门还发现，敌军有一台神秘机器。即使我军切断所有能源之后，他们也可以用那台机器。机器产生的效果不仅仅会修复所有我军炸毁的桥梁，而且会重新随机资源分布（但可以保证的是，资源不会分布到 1 号岛屿上）。不过侦查部门还发现了这台机器只能够使用 m 次，所以我们只需要把每次任务完成即可。

输入格式

第一行一个整数 n，代表岛屿数量。

接下来 $n-1$ 行，每行三个整数 u,v,w，代表 u 号岛屿和 v 号岛屿由一条代价为 w 的桥梁直接相连，保证 $1 \le u,v \le n$ 且 $1 \le w \le 10^5$。

第 $n+1$ 行，一个整数 m，代表敌方机器能使用的次数。

接下来 m 行，每行一个整数 k_i，代表第 i 次后，有 k_i 个岛屿资源丰富，接下来 k 个整数 $h_1,h_2,\cdots,h_k$，表示资源丰富岛屿的编号。

输出格式

输出有 m 行，分别代表每次任务的最小代价。

数据范围

对于 100% 的数据，$2 \le n \le 2.5 \times 10^5, 1 \le m \le 5 \times 10^5, \sum k_i \le 5 \times 10^5, 1 \le k_i \le n-1$。

过程

对于上面那题，我们不难发现——如果树的点数很少，那么我们可以直接跑 DP。

首先我们称某次询问中被选中的点为——「**关键点**」。

设 $Dp(i)$ 表示——使 i 不与其子树中任意一个关键点连通的**最小代价**。

设 $w(a,b)$ 表示 a 与 b 之间的边的权值。

则枚举 i 的儿子 v:

- 若 v 不是关键点：$Dp(i) = Dp(i) + \min\{Dp(v), w(i, v)\}$；
- 若 v 是关键点：$Dp(i) = Dp(i) + w(i, v)$。

很好，这样我们得到了一份 $O(nq)$ 的代码。

听起来很有意思。

解释

我们不难发现——其实很多点是没有用的。以下图为例：

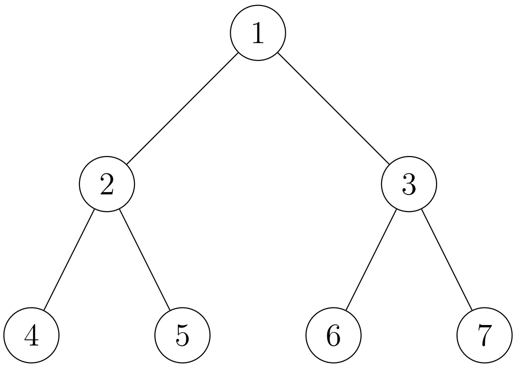


图 11.24 vtree-1

如果我们选取的关键点是：

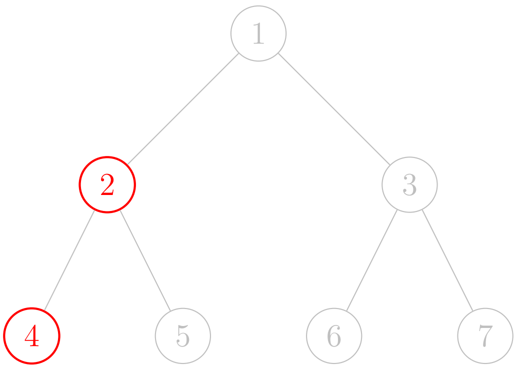


图 11.25 vtree-2

图中只有两个红色的点是**关键点**，而别的点全都是「非关键点」。

对于这题来说，我们只需要保证红色的点无法到达 1 号节点就行了。

通过肉眼观察可以得出结论——1 号节点的右子树（虽然实际上可能有多个子树，但这里只有两个子树，所以暂时这么称呼了）一个红色节点都没有，**所以没必要去 DP 它**。

观察题目给出的条件，红色点（关键点）的总数是与 n 同阶的，也就是说实际上一次询问中红色的点对于整棵树来说是很稀疏的，所以如果我们能让复杂度由红色点的总数来决定就好了。

因此我们需要**浓缩信息，把一整颗大树浓缩成一颗小树**。

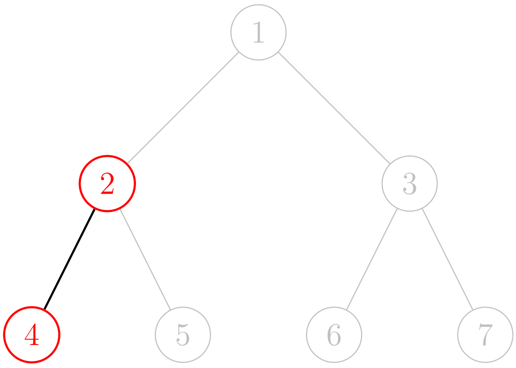
虚树 Virtual Tree

由此我们引出了「**虚树**」这个概念。

我们先直观地来看看虚树的样子。

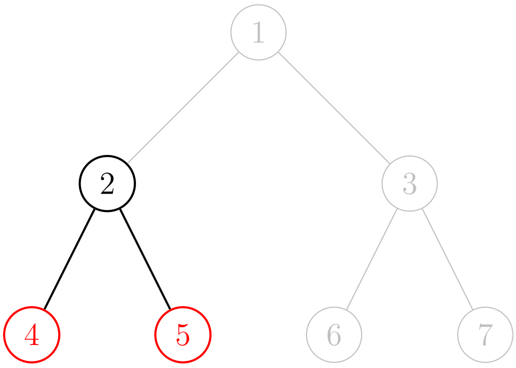
下图中，红色结点是我们选择的关键点。红色和黑色结点都是虚树中的点。黑色的边是虚树中的边。

因为任意两个关键点的 LCA 也是需要保存重要信息的，所以我们需要保存它们的 LCA，因此虚树中不一定只有关键点。



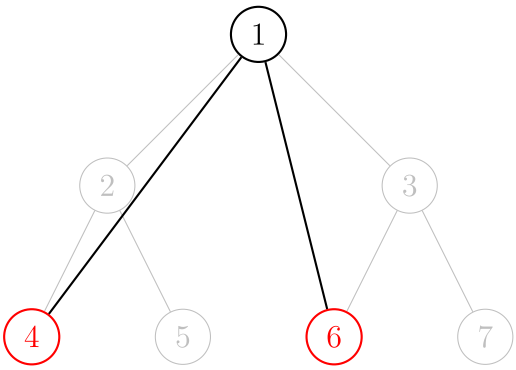
$\{2,4\}$ 的虚树

图 11.26 vtree-3



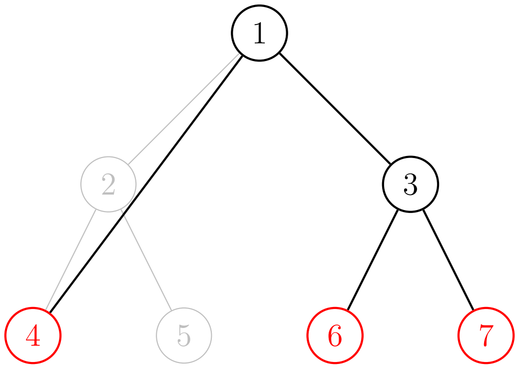
$\{4,5\}$ 的虚树

图 11.27 vtree-4



$\{4,6\}$ 的虚树

图 11.28 vtree-5



$\{4, 6, 7\}$ 的虚树

图 11.29 vtree-6

不难发现虚树中祖先后代的关系并不会改变。（就是不会出现原本 a 是 b 的祖先结果后面 a 变成 b 的后代了之类的鬼事）

但我们不可能 $O(k^2)$ 暴力枚举 LCA，所以我们不难想到——首先将关键点按 DFS 序排序，然后排完序以后相邻的两个关键点（相邻指的是在排序后的序列中下标差值的绝对值等于 1）求一下 LCA，并把它加入虚树。

我们的当务之急就是如何构造虚树。

在提出方案之前，我们先确认一个事实——在虚树里，只要保证祖先后代的关系没有改变，就可以随意添加节点。

也就是，如果我们乐意，我们可以把原树中所有的点都加入虚树中，也不会导致 WA（虽然会导致 TLE）。

因此，我们为了方便，可以首先将 1 号节点加入虚树中，并且并不会影响答案。

第一种构造过程：二次排序 + LCA 连边

因为多个节点的 LCA 可能是同一个，所以我们不能多次将它加入虚树。

非常直观的一个方法是：

- 将关键点按 DFS 序排序；
- 遍历一遍，任意两个相邻的关键点求一下 LCA，并且判重；
- 然后根据原树中的祖先后代关系建树。

具体实现上，在**关键点序列**上，枚举**相邻的两个数**，两两求得 lca 并且加入序列 A 中。

因为 DFS 序的性质，此时的序列 A 已经包含了**虚树中的所有点**，但是可能有重复。

所以我们把序列 A 按照 dfn 序**从小到大排序并去重**。

最后，在序列 A 上，枚举**相邻**的两个**点编号** x, y，求得它们的 lca 并且连接 lca, y，虚树就构造完成了。

为什么连接 lca 和 y 可以做到不重不漏呢？

" 证明"

如果 x 是 y 的祖先，那么 x 直接到 y 连边。因为 dfn 序保证了 x 和 y 的 dfn 序是相邻的，所以 x 到 y 的路径上面没有关键点。

如果 x 不是 y 的祖先，那么就把 $lca(x, y)$ 当作 y 的的祖先，根据上一种情况也可以证明 $lca(x, y)$ 到 y 点的路径上不会有关键点。

所以连接 lca 和 y，不会遗漏，也不会重复。

另外第一个点没有被一个节点连接会不会有影响呢？因为第一个点一定是这棵树的根，所以不会有影响，所以总边数就是 $m - 1$ 条。

因为至少要两个实点才能够召唤出来一个虚点，再加上一个根节点，所以虚树的点数就是实点数量的两倍。

时间复杂度 $O(m \log n)$，其中 m 为关键点数，n 为总点数。

实现

```
int dfn[maxn];
bool valid[maxn];
int h[maxn], m, a[maxn], len;  // 存储关键点

bool cmp(int x, int y) {
  return dfn[x] < dfn[y];  // 按照 dfn 序排序
}

void build_virtual_tree() {
  sort(h + 1, h + m + 1, cmp);  // 把关键点按照 dfn 序排序
  for (int i = 1; i < m; ++i) {
    a[++len] = h[i];
    a[++len] = lca(h[i], h[i + 1]);  // 插入 lca
  }
  a[++len] = h[m];
  sort(a + 1, a + len + 1, cmp);  // 把所有虚树上的点按照 dfn 序排序
  len = unique(a + 1, a + len + 1) - a - 1;  // 去重
  for (int i = 1, lc; i < len; ++i) {
    lc = lca(a[i], a[i + 1]);
    conn(lc, a[i + 1]);  // 连边，如有边权就是 distance(lc,a[i+1])
  }
}
```

其实这样就足以构造一棵虚树了。

第二种构造过程：使用单调栈

如何使用单调栈构造虚树?

首先我们要明确一个目的——我们要用单调栈来维护一条虚树上的链。

也就是一个栈里相邻的两个节点在虚树上也是相邻的，而且栈是从底部到栈首单调递增的（指的是栈中节点 DFS 序单调递增），说白了就是某个节点的父亲就是栈中它下面的那个节点。

首先我们在栈中添加节点 1。

然后接下来按照 DFS 序从小到大添加关键节点。

假如当前的节点与栈顶节点的 LCA 就是栈顶节点的话，则说明它们是在一条链上的。所以直接把当前节点入栈就行了。

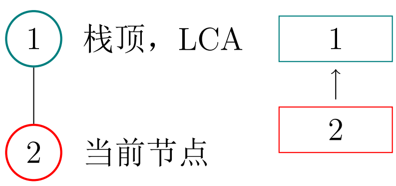


图 11.30 vtree-7

假如当前节点与栈顶节点的 LCA 不是栈顶节点的话：

这时，当前单调栈维护的链是：

而我们需要把链变成：

那么我们就把用虚线标出的结点弹栈即可，在弹栈前别忘了向它在虚树中的父亲连边。

假如弹出以后发现栈首不是 LCA 的话要让 LCA 入栈。

再把当前节点入栈就行了。

下面给出一个具体的例子。假设我们要对下面这棵树的 4，6 和 7 号结点建立虚树：

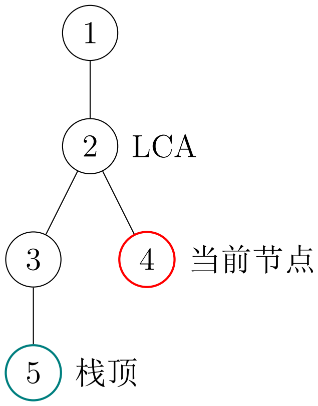


图 11.31 vtree-8

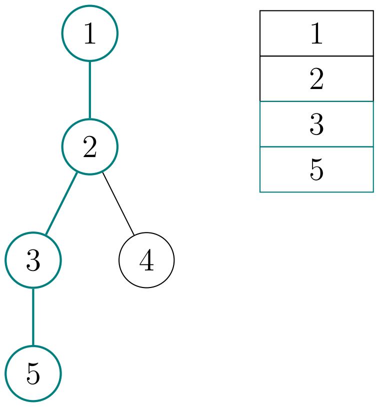


图 11.32 vtree-9

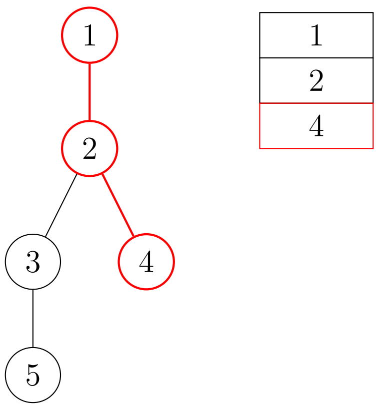


图 11.33 vtree-10

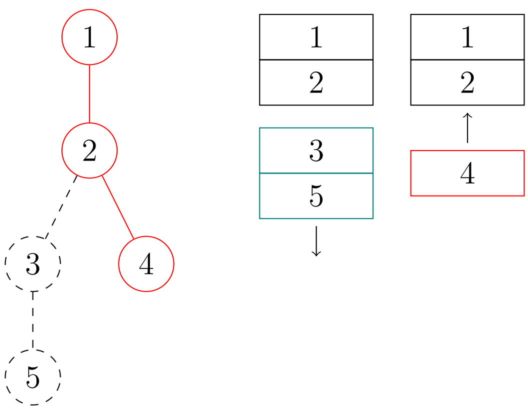


图 11.34 vtree-11

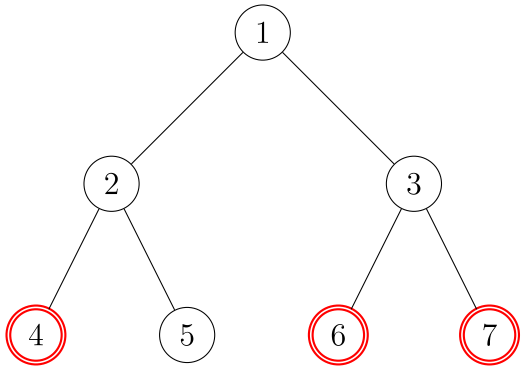


图 11.35 vtree-12

那么步骤是这样的：

- 将 3 个关键点 $6, 4, 7$ 按照 DFS 序排序，得到序列 $[4, 6, 7]$。
- 将 1 入栈。

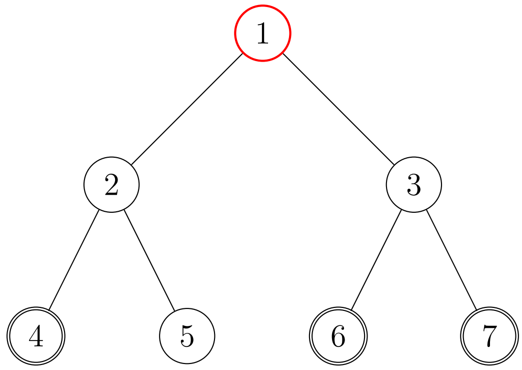


图 11.36 vtree-13

我们用红色的点代表在栈内的点，青色的点代表从栈中弹出的点。

- 取序列中第一个作为当前节点，也就是 4。再取栈顶元素，为 1。求 1 和 4 的 LCA：$LCA(1, 4) = 1$。
- 发现 $LCA(1, 4) =$ 栈顶元素，说明它们在虚树的一条链上，所以直接把当前节点 4 入栈，当前栈为 $4, 1$。

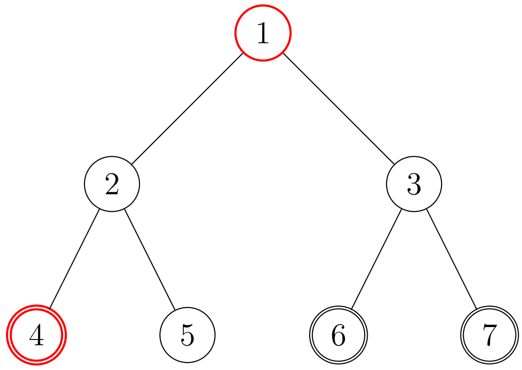


图 11.37 vtree-14

- 取序列第二个作为当前节点，为 6。再取栈顶元素，为 4。求 6 和 4 的 LCA：$LCA(6, 4) = 1$。
- 发现 $LCA(6, 4) \neq$ 栈顶元素，进入判断阶段。
- 判断阶段：发现栈顶节点 4 的 DFS 序是大于 $LCA(6, 4)$ 的，但是次大节点（栈顶节点下面的那个节点）1 的 DFS 序是等于 LCA 的（其实 DFS 序相等说明节点也相等），说明 LCA 已经入栈了，所以直接连接 $1 \to 4$ 的

边，也就是 LCA 到栈顶元素的边。并把 4 从栈中弹出。

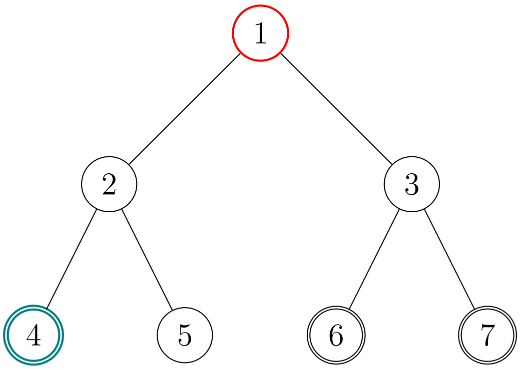


图 11.38 vtree-15

- 结束了判断阶段，将 6 入栈，当前栈为 6, 1。

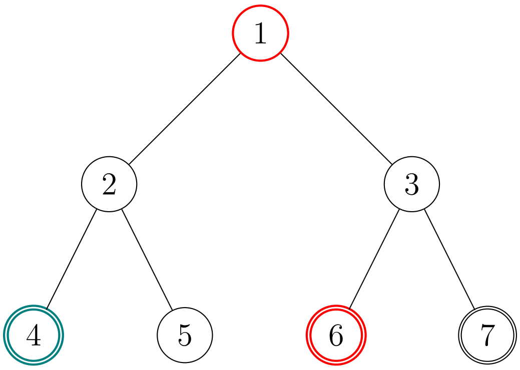


图 11.39 vtree-16

- 取序列第三个作为当前节点，为 7。再取栈顶元素，为 6。求 7 和 6 的 LCA：$LCA(7,6) = 3$。
- 发现 $LCA(7,6) \neq$ 栈顶元素，进入判断阶段。
- 判断阶段：发现栈顶节点 6 的 DFS 序是大于 $LCA(7,6)$ 的，但是次大节点（栈顶节点下面的那个节点）1 的 DFS 序是小于 LCA 的，说明 LCA 还没有入过栈，所以直接连接 $3 \to 6$ 的边，也就是 LCA 到栈顶元素的边。把 6 从栈中弹出，并且把 $LCA(6,7)$ 入栈。
- 结束了判断阶段，将 7 入栈，当前栈为 1, 3, 7。

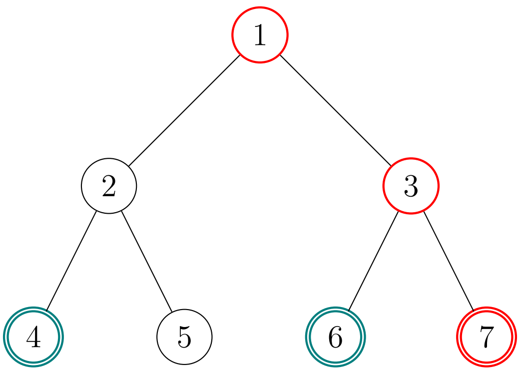


图 11.40 vtree-17

- 发现序列里的 3 个节点已经全部加入过栈了，退出循环。
- 此时栈中还有 3 个节点：1, 3, 7，很明显它们是一条链上的，所以直接链接：$1 \to 3$ 和 $3 \to 7$ 的边。

- 虚树就建完啦!

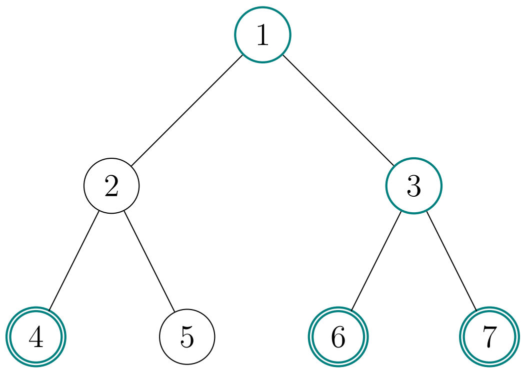


图 11.41 vtree-18

我们接下来将那些没入过栈的点（非青色的点）删掉，对应的虚树长这个样子:

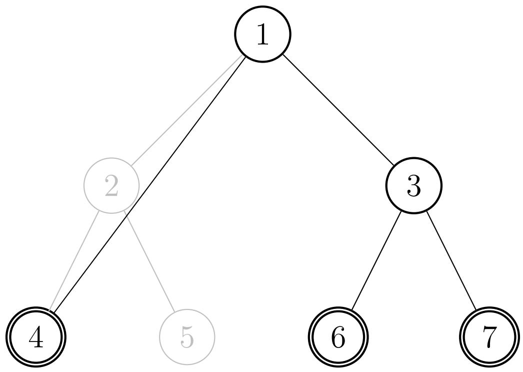


图 11.42 vtree-19

其中有很多细节，比如用邻接表存图的方式存虚树的话，需要清空邻接表。但是直接清空整个邻接表是很慢的，所以我们在**有一个从未入栈的元素入栈的时候清空该元素对应的邻接表**即可。

实现

建立虚树的 C++ 代码大概长这样:

" 代码实现"

```
bool cmp(const int x, const int y) { return id[x] < id[y]; }

void build() {
  sort(h + 1, h + k + 1, cmp);
  sta[top = 1] = 1, g.sz = 0, g.head[1] = -1;
  // 1 号节点入栈，清空 1 号节点对应的邻接表，设置邻接表边数为 1
  for (int i = 1, l; i <= k; ++i)
    if (h[i] != 1) {
      // 如果 1 号节点是关键节点就不要重复添加
      l = lca(h[i], sta[top]);
      // 计算当前节点与栈顶节点的 LCA
      if (l != sta[top]) {
        // 如果 LCA 和栈顶元素不同，则说明当前节点不再当前栈所存的链上
        while (id[l] < id[sta[top - 1]])
          // 当次大节点的 Dfs 序大于 LCA 的 Dfs 序
          g.push(sta[top - 1], sta[top]), top--;
```

```
        // 把与当前节点所在的链不重合的链连接掉并且弹出
        if (id[l] > id[sta[top - 1]])
          // 如果 LCA 不等于次大节点（这里的大于其实和不等于没有区别）
          g.head[l] = -1, g.push(l, sta[top]), sta[top] = l;
        // 说明 LCA 是第一次入栈，清空其邻接表，连边后弹出栈顶元素，并将 LCA
        // 入栈
        else
          g.push(l, sta[top--]);
        // 说明 LCA 就是次大节点，直接弹出栈顶元素
      }
      g.head[h[i]] = -1, sta[++top] = h[i];
      // 当前节点必然是第一次入栈，清空邻接表并入栈
    }
  for (int i = 1; i < top; ++i)
    g.push(sta[i], sta[i + 1]);  // 剩余的最后一条链连接一下
  return;
}
```

于是我们就学会了虚树的建立了！

对于消耗战这题，直接在虚树上跑最开始讲的那个 DP 就行了，我们等于利用了虚树排除了那些没用的非关键节点！仍然考虑 i 的所有儿子 v：

- 若 v 不是关键点：$Dp(i) = Dp(i) + \min\{Dp(v), w(i,v)\}$
- 若 v 是关键点：$Dp(i) = Dp(i) + w(i,v)$

于是这题很简单就过了。

推荐习题

- 「SDOI2011」 消耗战[1]
- 「HEOI2014」 大工程[2]
- CF613D Kingdom and its Cities[3]
- 「HNOI2014」 世界树[4]

参考资料与注释

[1] 「SDOI2011」消耗战 [1-1] [1-2]

[2] 「HEOI2014」大工程

[3] CF613D Kingdom and its Cities

[4] 「HNOI2014」世界树

11.6.8 树分治

点分治

点分治适合处理大规模的树上路径信息问题。

" 例题 1 Luogu P3806【模板】 点分治 1[1]"

给定一棵有 n 个点的带边权树，m 次询问，每次询问给出 k，询问树上距离为 k 的点对是否存在。

$n \le 10000, m \le 100, k \le 10000000$

我们先随意选择一个节点作为根节点 rt，所有完全位于其子树中的路径可以分为两种，一种是经过当前根节点的路径，一种是不经过当前根节点的路径。对于经过当前根节点的路径，又可以分为两种，一种是以根节点为一个端点的路径，另一种是两个端点都不为根节点的路径。而后者又可以由两条属于前者链合并得到。所以，对于枚举的根节点 rt，我们先计算在其子树中且经过该节点的路径对答案的贡献，再递归其子树对不经过该节点的路径进行求解。

在本题中，对于经过根节点 rt 的路径，我们先枚举其所有子节点 ch，以 ch 为根计算 ch 子树中所有节点到 rt 的距离。记节点 i 到当前根节点 rt 的距离为 $dist_i$，tf_d 表示之前处理过的子树中是否存在一个节点 v 使得 $dist_v = d$。若一个询问的 k 满足 $tf_{k-dist_i} = true$，则存在一条长度为 k 的路径。在计算完 ch 子树中所连的边能否成为答案后，我们将这些新的距离加入 tf 数组中。

注意在清空 tf 数组的时候不能直接用 `memset`，而应将之前占用过的 tf 位置加入一个队列中，进行清空，这样才能保证时间复杂度。

点分治过程中，每一层的所有递归过程合计对每个点处理一次，假设共递归 h 层，则总时间复杂度为 $O(hn)$。

若我们**每次选择子树的重心作为根节点**，可以保证递归层数最少，时间复杂度为 $O(n \log n)$。

请注意在重新选择根节点之后一定要重新计算子树的大小，否则一点看似微小的改动就可能会使时间复杂度错误或正确性难以保证。

" 参考代码"

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <queue>
using namespace std;
const int maxn = 20010;
const int inf = 2e9;
int n, m, a, b, c, q[maxn], rt, siz[maxn], maxx[maxn], dist[maxn];
int cur, h[maxn], nxt[maxn], p[maxn], w[maxn];
bool tf[10000010], ret[maxn], vis[maxn];

void add_edge(int x, int y, int z) {
  cur++;
  nxt[cur] = h[x];
  h[x] = cur;
  p[cur] = y;
  w[cur] = z;
}

int sum;

void calcsiz(int x, int fa) {
  siz[x] = 1;
  maxx[x] = 0;
  for (int j = h[x]; j; j = nxt[j])
    if (p[j] != fa && !vis[p[j]]) {
      calcsiz(p[j], x);
      maxx[x] = max(maxx[x], siz[p[j]]);
      siz[x] += siz[p[j]];
    }
```

```
  maxx[x] = max(maxx[x], sum - siz[x]);
  if (maxx[x] < maxx[rt]) rt = x;
}

int dd[maxn], cnt;

void calcdist(int x, int fa) {
  dd[++cnt] = dist[x];
  for (int j = h[x]; j; j = nxt[j])
    if (p[j] != fa && !vis[p[j]])
      dist[p[j]] = dist[x] + w[j], calcdist(p[j], x);
}

queue<int> tag;

void dfz(int x, int fa) {
  tf[0] = true;
  tag.push(0);
  vis[x] = true;
  for (int j = h[x]; j; j = nxt[j])
    if (p[j] != fa && !vis[p[j]]) {
      dist[p[j]] = w[j];
      calcdist(p[j], x);
      for (int k = 1; k <= cnt; k++)
        for (int i = 1; i <= m; i++)
          if (q[i] >= dd[k]) ret[i] |= tf[q[i] - dd[k]];
      for (int k = 1; k <= cnt; k++)
        if (dd[k] < 10000010) tag.push(dd[k]), tf[dd[k]] = true;
      cnt = 0;
    }
  while (!tag.empty()) tf[tag.front()] = false, tag.pop();
  for (int j = h[x]; j; j = nxt[j])
    if (p[j] != fa && !vis[p[j]]) {
      sum = siz[p[j]];
      rt = 0;
      maxx[rt] = inf;
      calcsiz(p[j], x);
      calcsiz(rt, -1);
      dfz(rt, x);
    }
}

int main() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i < n; i++)
    scanf("%d%d%d", &a, &b, &c), add_edge(a, b, c), add_edge(b, a, c);
  for (int i = 1; i <= m; i++) scanf("%d", q + i);
  rt = 0;
  maxx[rt] = inf;
  sum = n;
  calcsiz(1, -1);
  calcsiz(rt, -1);
  dfz(rt, -1);
  for (int i = 1; i <= m; i++)
```

```
    if (ret[i])
      printf("AYE\n");
    else
      printf("NAY\n");
  return 0;
}
```

" 例题 2 Luogu P4178 Tree[2]"

给定一棵有 n 个点的带权树，给出 k，询问树上距离小于等于 k 的点对数量。

$n \leq 40000, k \leq 20000, w_i \leq 1000$

由于这里查询的是树上距离为 $[0, k]$ 的点对数量，所以我们用线段树来支持维护和查询。

" 参考代码"

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <queue>
#define int long long
using namespace std;
const int maxn = 2000010;
const int inf = 2e9;
int n, a, b, c, q, rt, siz[maxn], maxx[maxn], dist[maxn];
int cur, h[maxn], nxt[maxn], p[maxn], w[maxn], ret;
bool vis[maxn];

void add_edge(int x, int y, int z) {
  cur++;
  nxt[cur] = h[x];
  h[x] = cur;
  p[cur] = y;
  w[cur] = z;
}

int sum;

void calcsiz(int x, int fa) {
  siz[x] = 1;
  maxx[x] = 0;
  for (int j = h[x]; j; j = nxt[j])
    if (p[j] != fa && !vis[p[j]]) {
      calcsiz(p[j], x);
      maxx[x] = max(maxx[x], siz[p[j]]);
      siz[x] += siz[p[j]];
    }
  maxx[x] = max(maxx[x], sum - siz[x]);
  if (maxx[x] < maxx[rt]) rt = x;
}

int dd[maxn], cnt;

void calcdist(int x, int fa) {
```

```
  dd[++cnt] = dist[x];
  for (int j = h[x]; j; j = nxt[j])
    if (p[j] != fa && !vis[p[j]])
      dist[p[j]] = dist[x] + w[j], calcdist(p[j], x);
}

queue<int> tag;

struct segtree {
  int cnt, rt, lc[maxn], rc[maxn], sum[maxn];

  void clear() {
    while (!tag.empty()) update(rt, 1, 20000000, tag.front(), -1), tag.pop();
    cnt = 0;
  }

  void print(int o, int l, int r) {
    if (!o || !sum[o]) return;
    if (l == r) {
      printf("%lld %lld\n", l, sum[o]);
      return;
    }
    int mid = (l + r) >> 1;
    print(lc[o], l, mid);
    print(rc[o], mid + 1, r);
  }

  void update(int& o, int l, int r, int x, int v) {
    if (!o) o = ++cnt;
    if (l == r) {
      sum[o] += v;
      if (!sum[o]) o = 0;
      return;
    }
    int mid = (l + r) >> 1;
    if (x <= mid)
      update(lc[o], l, mid, x, v);
    else
      update(rc[o], mid + 1, r, x, v);
    sum[o] = sum[lc[o]] + sum[rc[o]];
    if (!sum[o]) o = 0;
  }

  int query(int o, int l, int r, int ql, int qr) {
    if (!o) return 0;
    if (r < ql || l > qr) return 0;
    if (ql <= l && r <= qr) return sum[o];
    int mid = (l + r) >> 1;
    return query(lc[o], l, mid, ql, qr) + query(rc[o], mid + 1, r, ql, qr);
  }
} st;

void dfz(int x, int fa) {
  // tf[0]=true;tag.push(0);
```

```
  st.update(st.rt, 1, 20000000, 1, 1);
  tag.push(1);
  vis[x] = true;
  for (int j = h[x]; j; j = nxt[j])
    if (p[j] != fa && !vis[p[j]]) {
      dist[p[j]] = w[j];
      calcdist(p[j], x);
      for (int k = 1; k <= cnt; k++)
        if (q - dd[k] >= 0)
          ret += st.query(st.rt, 1, 20000000, max(0ll, 1 - dd[k]) + 1,
                          max(0ll, q - dd[k]) + 1);
      for (int k = 1; k <= cnt; k++)
        st.update(st.rt, 1, 20000000, dd[k] + 1, 1), tag.push(dd[k] + 1);
      cnt = 0;
    }
  st.clear();
  for (int j = h[x]; j; j = nxt[j])
    if (p[j] != fa && !vis[p[j]]) {
      sum = siz[p[j]];
      rt = 0;
      maxx[rt] = inf;
      calcsiz(p[j], x);
      calcsiz(rt, -1);
      dfz(rt, x);
    }
}

signed main() {
  scanf("%lld", &n);
  for (int i = 1; i < n; i++)
    scanf("%lld%lld%lld", &a, &b, &c), add_edge(a, b, c), add_edge(b, a, c);
  scanf("%lld", &q);
  rt = 0;
  maxx[rt] = inf;
  sum = n;
  calcsiz(1, -1);
  calcsiz(rt, -1);
  dfz(rt, -1);
  printf("%lld\n", ret);
  return 0;
}
```

" **例题 3 Luogu P2664 树上游戏**[3] "

一棵每个节点都给定颜色的树，定义 $s(i,j)$ 为 i 到 j 的颜色数量，$sum_i = \sum_{j=1}^{n} s(i,j)$。对所有的 $1 \le i \le n$，求 sum_i。（$1 \le n, c_i \le 10^5$）

这道题很考验对点分治思想的理解和应用，适合作为点分治的难度较高的例题和练习题。

首先，我们需要想明白一个转化。题目定义 sum_i 是 i 到所有节点路径上的颜色数量之和，可是如果用这个方法，在点分治中是不好统计答案的，因为这样很难合并从当前根出发的两棵子树的信息。所以我们想到将 sum_i 的意义转化。对于每个颜色 j，其中一个端点为 i 且含有颜色 j 的路径数量记为 cnt_j，sum_i 其实就是 $\sum cnt_j$。这一步转化其实就是换了个观察对象，考虑的是每个颜色对 sum_i 的贡献。而 cnt_j 其实很好处理出来，只需要每遇到一个新颜色，就 $cnt_{col_u} += size_u$ 即可，其中 $size_u$ 为 u 的子树大小，意味着这个子树里的所有节点都在这个颜色上对 u 的答案有一个

贡献。

考虑到点分治过程中，我们只需要分别考虑统计：

1. 子树中以当前根节点为端点的路径对根的贡献
2. lca 为当前根节点的路径对子树内每个点的贡献

1 部分比较好办，由于点分治中，递归层数不超过 $\log n$，每一层我们都可以遍历全部子树，这个时候就可以使用 sum_i 的定义式来在遍历子树的过程中顺便统计了。

而针对 2 部分，设当前根节点 u 的一个子节点为 d,d 的子树里任取一个点为 v，那么 v 的答案可以分为两部分：

1. (u,v) 路径上出现过的颜色，数量设为 num，u 除了 d 以外的其他所有子树的总大小设为 $siz1$, 那么这些出现过的颜色对 v 的答案贡献为 $num \times siz1$。
2. (u,v) 路径上没有出现过的颜色 j，它们的贡献来自于 u 除了 d 以外的其他所有子树的 cnt_j，这部分答案为 $\sum_{j \notin (u,v)} cnt_j$。

以上是全部统计思路，实现细节详见参考代码。

" 参考代码"

```
#include <bits/stdc++.h>
using namespace std;
#define rep(i, a, b) for (int i = (a); i <= (b); ++i)
const int N = 200005;
int h[N], nxt[N * 2], to[N * 2], c[N], gr;
#define il inline

il void tu(int x, int y) { to[++gr] = y, nxt[gr] = h[x], h[x] = gr; }

typedef long long ll;
int n, nn, siz[N], mn, rt;
bool vis[N];

void get_root(int u, int f) {
  siz[u] = 1;
  int mx = 0;
  for (int i = h[u]; i; i = nxt[i]) {
    int d = to[i];
    if (vis[d] || d == f) continue;
    get_root(d, u);
    siz[u] += siz[d];
    mx = max(mx, siz[d]);
  }
  mx = max(mx, nn - siz[u]);
  if (mx < mn) mn = mx, rt = u;
}

ll ans[N], sum;
int cnt[N], v[N];
// sum 实时统计的是 cnt[i] 的和
int nowrt;

void get_dis(int u, int f, int now) {  // now 为当前树链上的颜色数量 (不含 u)
  siz[u] = 1;
  if (!v[c[u]]) {
```

```
    sum -= cnt[c[u]];  // 减去在之前子树中已经出现过的颜色信息
    now++;
  }
  v[c[u]]++;
  ans[u] += sum + now * siz[nowrt];  // 统计过 u 点的路径对 u 的贡献
  for (int i = h[u]; i; i = nxt[i]) {
    int d = to[i];
    if (d == f || vis[d]) continue;
    get_dis(d, u, now);
    siz[u] += siz[d];
  }
  v[c[u]]--;
  if (!v[c[u]]) {
    sum += cnt[c[u]];  // 回溯
  }
}

void get_cnt(int u, int f) {
  if (!v[c[u]]) {
    cnt[c[u]] += siz[u];
    sum += siz[u];  // 将刚遍历过的子树的信息整合到 cnt[i] 和 sum 上去
  }
  v[c[u]]++;
  for (int i = h[u]; i; i = nxt[i]) {
    int d = to[i];
    if (vis[d] || d == f) continue;
    get_cnt(d, u);
  }
  v[c[u]]--;
}

void clear(int u, int f, int now) {
  if (!v[c[u]]) now++;
  v[c[u]]++;
  ans[u] -= now;
  ans[nowrt] += now;
  for (int i = h[u]; i; i = nxt[i]) {
    int d = to[i];
    if (vis[d] || d == f) continue;
    clear(d, u, now);
  }
  v[c[u]]--;
  cnt[c[u]] = 0;
}

void clear2(int u, int f) {
  cnt[c[u]] = 0;
  for (int i = h[u]; i; i = nxt[i]) {
    int d = to[i];
    if (vis[d] || d == f) continue;
    clear2(d, u);
  }
}
```

```
int son[N];

void divid(int u) {
  vis[u] = 1;
  int tot = 0;
  nowrt = u;
  ans[u]++;
  for (int i = h[u]; i; i = nxt[i]) {
    if (vis[to[i]]) continue;
    son[++tot] = to[i];
  }
  siz[u] = sum = cnt[c[u]] = 1;
  v[c[u]]++;
  rep(i, 1, tot) {  // 统计每个子树和它之前的所有子树中节点组合产生的贡献
    int d = son[i];
    get_dis(d, u, 0);
    get_cnt(d, u);
    siz[u] += siz[d];
    cnt[c[u]] += siz[d];
    sum += siz[d];
  }
  clear2(u, 0);  // 清空数组，记得不可以用 memset
  siz[u] = sum = cnt[c[u]] = 1;
  for (int i = tot; i >= 1;
       --i) {  // 统计每个子树和它之后的所有子树中节点组合产生的贡献
    int d = son[i];
    get_dis(d, u, 0);
    get_cnt(d, u);
    siz[u] += siz[d];
    cnt[c[u]] += siz[d];
    sum += siz[d];
  }
  v[c[u]]--;
  clear(u, 0, 0);                      // 清空的同时统计答案
  for (int i = h[u]; i; i = nxt[i]) {  // 继续向下进行点分治
    int d = to[i];
    if (vis[d]) continue;
    nn = siz[d], mn = n + 1, rt = 0;
    get_root(d, u);
    divid(rt);
  }
}

int main() {
  scanf("%d", &n);
  int u, v;
  rep(i, 1, n) scanf("%d", &c[i]);
  rep(i, 2, n) scanf("%d%d", &u, &v), tu(u, v), tu(v, u);
  rt = 0, nn = n, mn = n + 1;
  get_root(1, 0);
  divid(rt);
  rep(i, 1, n) printf("%lld\n", ans[i]);
  return 0;
}
```

边分治

与上面的点分治类似，我们选取一条边，把树尽量均匀地分成两部分（使边连接的两个子树的 $size$ 尽量接近）。然后递归处理左右子树，统计信息。

但是这是不行的，考虑一个菊花图：

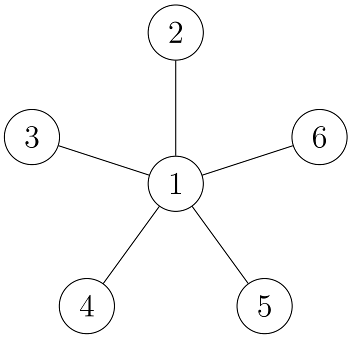


图 11.43 菊花图

我们发现当一个点下有多个 $size$ 接近的儿子时，应用边分治的时间复杂度是无法接受的。

如果这个图是个二叉树，就可以避免上面菊花图中应用边分治的弊端。因此我们考虑把一个多叉树转化成二叉树。

显然，我们只需像线段树那样建树就可以了。就像这样

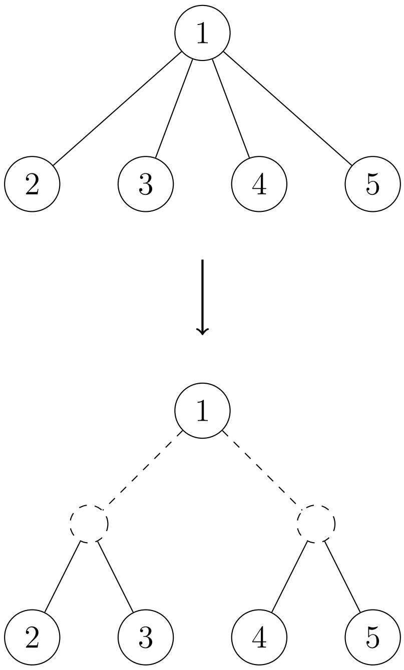


图 11.44 建树

新建出来的点根据题目要求给予恰当的信息即可。例如：统计路径长度时，将原边边权赋为 1, 将新建的边边权赋为 0 即可。

分析复杂度，发现最多会增加 $O(n)$ 个点，则总复杂度为 $O(n\log n)$

几乎所有点分治的题边分都能做（常数上有差距，但是不卡），所以就不放例题了。

点分树

点分树是通过更改原树形态使树的层数变为稳定 $\log n$ 的一种重构树。

常用于解决与树原形态无关的带修改问题。

算法分析

我们通过点分治每次找重心的方式来对原树进行重构。

将每次找到的重心与上一层的重心缔结父子关系，这样就可以形成一棵 $\log n$ 层的树。

由于树是 $\log n$ 层的，很多原来并不对劲的暴力在点分树上均有正确的复杂度。

代码实现

有一个小技巧：每次用递归上一层的总大小 tot 减去上一层的点的重儿子大小，得到的就是这一层的总大小。这样求重心就只需一次 DFS 了。

" 参考代码"

```cpp
#include <bits/stdc++.h>

using namespace std;

typedef vector<int>::iterator IT;

struct Edge {
  int to, nxt, val;

  Edge() {}

  Edge(int to, int nxt, int val) : to(to), nxt(nxt), val(val) {}
} e[300010];

int head[150010], cnt;

void addedge(int u, int v, int val) {
  e[++cnt] = Edge(v, head[u], val);
  head[u] = cnt;
}

int siz[150010], son[150010];
bool vis[150010];

int tot, lasttot;
int maxp, root;

void getG(int now, int fa) {
  siz[now] = 1;
  son[now] = 0;
  for (int i = head[now]; i; i = e[i].nxt) {
    int vs = e[i].to;
    if (vs == fa || vis[vs]) continue;
    getG(vs, now);
    siz[now] += siz[vs];
    son[now] = max(son[now], siz[vs]);
  }
  son[now] = max(son[now], tot - siz[now]);
  if (son[now] < maxp) {
    maxp = son[now];
    root = now;
  }
```

```cpp
}

struct Node {
  int fa;
  vector<int> anc;
  vector<int> child;
} nd[150010];

int build(int now, int ntot) {
  tot = ntot;
  maxp = 0x7f7f7f7f;
  getG(now, 0);
  int g = root;
  vis[g] = 1;
  for (int i = head[g]; i; i = e[i].nxt) {
    int vs = e[i].to;
    if (vis[vs]) continue;
    int tmp = build(vs, ntot - son[vs]);
    nd[tmp].fa = now;
    nd[now].child.push_back(tmp);
  }
  return g;
}

int virtroot;

int main() {
  int n;
  cin >> n;
  for (int i = 1; i < n; i++) {
    int u, v, val;
    cin >> u >> v >> val;
    addedge(u, v, val);
    addedge(v, u, val);
  }
  virtroot = build(1, n);
}
```

参考资料与注释

[1] Luogu P3806【模板】点分治 1

[2] Luogu P4178 Tree

[3] Luogu P2664 树上游戏

11.6.9 动态树分治

动态点分治

动态点分治用来解决**带点权/边权修改**的树上路径信息统计问题。

点分树

回顾点分治的计算过程。

对于一个结点 x 来说，其子树中的简单路径包括两种：经过结点 x 的，由一条或两条从 x 出发的路径组成的；和不经过结点 x 的，即已经包含在其所有儿子结点子树中的路径。

对于一个子树中简单路径的计算，我们选择一个分治中心 rt，计算经过该节点的子树中路径的信息，然后对于其每个儿子结点，将删去 rt 后该点所在连通块作为一个子树，递归计算。选择的分治中心点可以构成一个树形结构，称为**点分树**。我们发现，计算点分树中同一层的结点所代表的连通块（即以该结点为分治中心的连通块）的大小总和是 $O(n)$ 的。这意味着，点分治的时间复杂度是与点分树的深度相关的，若点分树的深度为 h，则点分治的复杂度为 $O(nh)$。

可以证明，当我们每次选择连通块的重心作为分治中心的时候，点分树的深度最小，为 $O(\log n)$ 的。这样，我们就可以在 $O(n\log n)$ 的时间复杂度内统计树上 $O(n^2)$ 条路径的信息了。

由于树的形态在动态点分治的过程中不会改变，所以点分树的形态在动态点分治的过程中也不会改变。

下面给出求点分树的参考代码：

```cpp
void calcsiz(int x, int f) {
  siz[x] = 1;
  maxx[x] = 0;
  for (int j = h[x]; j; j = nxt[j])
    if (p[j] != f && !vis[p[j]]) {
      calcsiz(p[j], x);
      siz[x] += siz[p[j]];
      maxx[x] = max(maxx[x], siz[p[j]]);
    }
  maxx[x] =
      max(maxx[x], sum - siz[x]);  // maxx[x] 表示以 x 为根时的最大子树大小
  if (maxx[x] < maxx[rt])
    rt = x;  // 这里不能写 <= ，保证在第二次 calcsiz 时 rt 不改变
}

void pre(int x) {
  vis[x] = true;  // 表示在之后的过程中不考虑 x 这个点
  for (int j = h[x]; j; j = nxt[j])
    if (!vis[p[j]]) {
      sum = siz[p[j]];
      rt = 0;
      maxx[rt] = inf;
      calcsiz(p[j], -1);
      calcsiz(rt, -1);  // 计算两次，第二次求出以 rt 为根时的各子树大小
      fa[rt] = x;
      pre(rt);  // 记录点分树上的父亲
    }
}

int main() {
  sum = n;
  rt = 0;
  maxx[rt] = inf;
  calcsiz(1, -1);
  calcsiz(rt, -1);
  pre(rt);
}
```

实现修改

在查询和修改的时候，我们在点分树上暴力跳父亲修改。由于点分树的深度最多是 $O(\log n)$ 的，所以这样做复杂度能得到保证。

在动态点分治的过程中，需要一个结点到其点分树上的祖先的距离等其他信息，由于一个点最多有 $O(\log n)$ 个祖先，我们可以在计算点分树时额外计算深度 $dep[x]$ 或使用 LCA，预处理出这些距离或实现实时查询。**注意**：一个结点到其点分树上的祖先的距离不一定递增，不能累加！

在动态点分治的过程中，一个结点在其点分树上的祖先结点的信息中可能会被重复计算，这是我们需要消去重复部分的影响。一般的方法是对于一个连通块用两种方式记录：一个是其到分治中心的距离信息，另一个是其到点分树上分治中心父亲的距离信息。这一部分内容将在例题中得到展现。

" **例题 「ZJOI2007」 捉迷藏**[1]"

给定一棵有 n 个结点的树，初始时所有结点都是黑色的。你需要实现以下两种操作：

1. 反转一个结点的颜色（白变黑，黑变白）；
2. 询问树上两个最远的黑点的距离。

$n \leq 10^5, m \leq 5 \times 10^5$

求出点分树，对于每个结点 x 维护两个**可删堆**。$dist[x]$ 存储结点 x 代表的连通块中的所有黑点到 x 的距离信息，$ch[x]$ 表示结点 x 在点分树上的所有儿子和它自己中的黑点到 x 的距离信息，由于本题贪心的求答案方法，且两个来自于同一子树的路径不能成为一条完成的路径，我们只在这个堆中插入其自己的值和其每个子树中的最大值。我们发现，$ch[x]$ 中最大的两个值（如果没有两个就是所有值）的和就是分治时分支中心为 x 时经过结点 x 的最长黑端点路径。我们可以用可删堆 ans 存储所有结点的答案，这个堆中的最大值就是我们所求的答案。

我们可以根据上面的定义维护 $dist[x], ch[x], ans$ 这些可删堆。当 $dist[x]$ 中的值发生变化时，我们也可以在 $O(\log n)$ 的时间复杂度内维护 $ch[x], ans$。

现在我们来看一下，当我们反转一个点的颜色时，$dist[x]$ 值会发生怎样的改变。当结点原来是黑色时，我们要进行的是删除操作；当结点原来是白色时，我们要进行的是插入操作。

假如我们要反转结点 x 的颜色。对于其所有祖先 u，我们在 $dist[u]$ 中插入或删除 $dist(x, u)$，并同时维护 $ch[x], ans$ 的值。特别的，我们要在 $ch[x]$ 中插入或删除值 0。

参考代码：

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <queue>
using namespace std;
const int maxn = 100010;
const int inf = 2e9;
int n, a, b, m, x, col[maxn];
// 0 off 1 on
char op;
int cur, h[maxn * 2], nxt[maxn * 2], p[maxn * 2];

void add_edge(int x, int y) {
  cur++;
  nxt[cur] = h[x];
  h[x] = cur;
  p[cur] = y;
}

bool vis[maxn];
int rt, sum, siz[maxn], maxx[maxn], fa[maxn], dep[maxn];
```

```
void calcsiz(int x, int f) {
  siz[x] = 1;
  maxx[x] = 0;
  for (int j = h[x]; j; j = nxt[j])
    if (p[j] != f && !vis[p[j]]) {
      calcsiz(p[j], x);
      siz[x] += siz[p[j]];
      maxx[x] = max(maxx[x], siz[p[j]]);
    }
  maxx[x] =
      max(maxx[x], sum - siz[x]);  // maxx[x] 表示以 x 为根时的最大子树大小
  if (maxx[x] < maxx[rt])
    rt = x;  // 这里不能写 <= ，保证在第二次 calcsiz 时 rt 不改变
}

struct heap {
  priority_queue<int> A, B;  // heap=A-B

  void insert(int x) { A.push(x); }

  void erase(int x) { B.push(x); }

  int top() {
    while (!B.empty() && A.top() == B.top()) A.pop(), B.pop();
    return A.top();
  }

  void pop() {
    while (!B.empty() && A.top() == B.top()) A.pop(), B.pop();
    A.pop();
  }

  int top2() {
    int t = top(), ret;
    pop();
    ret = top();
    A.push(t);
    return ret;
  }

  int size() { return A.size() - B.size(); }
} dist[maxn], ch[maxn], ans;

void dfs(int x, int f, int d, heap& y) {
  y.insert(d);
  for (int j = h[x]; j; j = nxt[j])
    if (p[j] != f && !vis[p[j]]) dfs(p[j], x, d + 1, y);
}

void pre(int x) {
  vis[x] = true;  // 不考虑 x
  for (int j = h[x]; j; j = nxt[j])
    if (!vis[p[j]]) {
```

```
      rt = 0;
      maxx[rt] = inf;
      sum = siz[p[j]];
      calcsiz(p[j], -1);
      calcsiz(rt, -1);  // 计算两次，第二次求出以 rt 为根时的各子树大小
      fa[rt] = x;
      dfs(p[j], -1, 1, dist[rt]);
      ch[x].insert(dist[rt].top());
      dep[rt] = dep[x] + 1;
      pre(rt);  // 记录点分树上的父亲
    }
  ch[x].insert(0);
  if (ch[x].size() >= 2)
    ans.insert(ch[x].top() + ch[x].top2());
  else if (ch[x].size())
    ans.insert(ch[x].top());
}

struct LCA {
  int dep[maxn], lg[maxn], fa[maxn][20];

  void dfs(int x, int f) {
    for (int j = h[x]; j; j = nxt[j])
      if (p[j] != f) dep[p[j]] = dep[x] + 1, fa[p[j]][0] = x, dfs(p[j], x);
  }

  void init() {
    dfs(1, -1);
    for (int i = 2; i <= n; i++) lg[i] = lg[i / 2] + 1;
    for (int j = 1; j <= lg[n]; j++)
      for (int i = 1; i <= n; i++) fa[i][j] = fa[fa[i][j - 1]][j - 1];
  }

  int query(int x, int y) {
    if (dep[x] > dep[y]) swap(x, y);
    int k = dep[y] - dep[x];
    for (int i = 0; k; k = k / 2, i++)
      if (k & 1) y = fa[y][i];
    if (x == y) return x;
    k = dep[x];
    for (int i = lg[k]; i >= 0; i--)
      if (fa[x][i] != fa[y][i]) x = fa[x][i], y = fa[y][i];
    return fa[x][0];
  }

  int dist(int x, int y) { return dep[x] + dep[y] - 2 * dep[query(x, y)]; }
} lca;

int d[maxn][20];

int main() {
  scanf("%d", &n);
  for (int i = 1; i < n; i++)
    scanf("%d%d", &a, &b), add_edge(a, b), add_edge(b, a);
```

```
  lca.init();
  rt = 0;
  maxx[rt] = inf;
  sum = n;
  calcsiz(1, -1);
  calcsiz(rt, -1);
  pre(rt);
  // for(int i=1;i<=n;i++)printf("%d ",fa[i]);printf("\n");
  for (int i = 1; i <= n; i++)
    for (int j = i; j; j = fa[j]) d[i][dep[i] - dep[j]] = lca.dist(i, j);
  scanf("%d", &m);
  while (m--) {
    scanf(" %c", &op);
    if (op == 'G') {
      if (ans.size())
        printf("%d\n", ans.top());
      else
        printf("-1\n");
    } else {
      scanf("%d", &x);
      if (!col[x]) {
        if (ch[x].size() >= 2) ans.erase(ch[x].top() + ch[x].top2());
        ch[x].erase(0);
        if (ch[x].size() >= 2) ans.insert(ch[x].top() + ch[x].top2());
        for (int i = x; fa[i]; i = fa[i]) {
          if (ch[fa[i]].size() >= 2)
            ans.erase(ch[fa[i]].top() + ch[fa[i]].top2());
          ch[fa[i]].erase(dist[i].top());
          dist[i].erase(d[x][dep[x] - dep[fa[i]]]);
          if (dist[i].size()) ch[fa[i]].insert(dist[i].top());
          if (ch[fa[i]].size() >= 2)
            ans.insert(ch[fa[i]].top() + ch[fa[i]].top2());
        }
      } else {
        if (ch[x].size() >= 2) ans.erase(ch[x].top() + ch[x].top2());
        ch[x].insert(0);
        if (ch[x].size() >= 2) ans.insert(ch[x].top() + ch[x].top2());
        for (int i = x; fa[i]; i = fa[i]) {
          if (ch[fa[i]].size() >= 2)
            ans.erase(ch[fa[i]].top() + ch[fa[i]].top2());
          if (dist[i].size()) ch[fa[i]].erase(dist[i].top());
          dist[i].insert(d[x][dep[x] - dep[fa[i]]]);
          ch[fa[i]].insert(dist[i].top());
          if (ch[fa[i]].size() >= 2)
            ans.insert(ch[fa[i]].top() + ch[fa[i]].top2());
        }
      }
      col[x] ^= 1;
    }
  }
  return 0;
}
```

" 例题 Luogu P6329【模板】点分树 | 震波[2]"

给定一棵有 n 个结点的树，树上每个结点都有一个权值 $v[x]$。实现以下两种操作：

1. 询问与结点 x 距离不超过 y 的结点权值和；
2. 修改结点 x 的点权为 y，即 $v[x] = y$。

我们用动态开点权值线段树记录距离信息。

类似于上题的思路，对于每个结点，我们维护线段树 $dist[x]$，表示分治块 x 中的所有结点到结点 x 的距离信息，下标为距离，权值加上点权。线段树 $ch[x]$ 表示分治块 x 中所有结点到结点 x 在分治树上的父亲结点的距离信息。

在本题中，所有查询和修改都需要在点分树上对所有祖先进行修改。

以查询操作为例，如果我们要查询距离结点 x 不超过 y 的结点的权值和，我们要先将答案加上线段树 $dist[x]$ 中下标从 0 到 y 的权值和，然后我们遍历 x 的所有祖先 u，设其低一级祖先为 v，令 $d = dist(x, u)$，如果我们不进入包含 x 的子树，即以 v 为根的子树，那么我们要将答案加上线段树 $dist[u]$ 中下标从 0 到 $y - d$ 的权值和。由于我们重复计算了以 v 为根的部分，我们要将答案减去线段树 $ch[v]$ 中下标从 0 到 $y - d$ 的权值和。

在进行修改操作时，我们要同时维护 $dist[x]$ 和 $ch[x]$。

参考代码：

```
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;
const int maxn = 100010;
const int inf = 2e9;
const int ddd = 6000010;

struct Segtree {
  int cnt, rt[maxn], sum[ddd], lc[ddd], rc[ddd];

  void update(int& o, int l, int r, int x, int v) {
    if (!o) o = ++cnt;
    if (l == r) {
      sum[o] += v;
      return;
    }
    int mid = (l + r) >> 1;
    if (x <= mid)
      update(lc[o], l, mid, x, v);
    else
      update(rc[o], mid + 1, r, x, v);
    sum[o] = sum[lc[o]] + sum[rc[o]];
  }

  int query(int o, int l, int r, int ql, int qr) {
    if (!o || r < ql || l > qr) return 0;
    if (ql <= l && r <= qr) return sum[o];
    int mid = (l + r) >> 1;
    return query(lc[o], l, mid, ql, qr) + query(rc[o], mid + 1, r, ql, qr);
  }
} dist, ch;

int n, m, val[maxn], u, v, op, x, y, lstans;
int cur, h[maxn * 2], nxt[maxn * 2], p[maxn * 2];
```

```cpp
void add_edge(int x, int y) {
  cur++;
  nxt[cur] = h[x];
  h[x] = cur;
  p[cur] = y;
}

struct LCA {
  int dep[maxn], lg[maxn], fa[maxn][20];

  void dfs(int x, int f) {
    for (int j = h[x]; j; j = nxt[j])
      if (p[j] != f) dep[p[j]] = dep[x] + 1, fa[p[j]][0] = x, dfs(p[j], x);
  }

  void init() {
    dep[1] = 1;
    dfs(1, -1);
    for (int i = 2; i <= n; i++) lg[i] = lg[i / 2] + 1;
    for (int j = 1; j <= lg[n]; j++)
      for (int i = 1; i <= n; i++) fa[i][j] = fa[fa[i][j - 1]][j - 1];
  }

  int query(int x, int y) {
    if (dep[x] > dep[y]) swap(x, y);
    int k = dep[y] - dep[x];
    for (int i = 0; k; k = k / 2, i++)
      if (k & 1) y = fa[y][i];
    if (x == y) return x;
    k = dep[x];
    for (int i = lg[k]; i >= 0; i--)
      if (fa[x][i] != fa[y][i]) x = fa[x][i], y = fa[y][i];
    return fa[x][0];
  }

  int dist(int x, int y) { return dep[x] + dep[y] - 2 * dep[query(x, y)]; }
} lca;

int rt, sum, siz[maxn], maxx[maxn], fa[maxn];
int d[maxn][20], dep[maxn];
bool vis[maxn];

void calcsiz(int x, int fa) {
  siz[x] = 1;
  maxx[x] = 0;
  for (int j = h[x]; j; j = nxt[j])
    if (p[j] != fa && !vis[p[j]]) {
      calcsiz(p[j], x);
      siz[x] += siz[p[j]];
      maxx[x] = max(maxx[x], siz[p[j]]);
    }
  maxx[x] =
      max(maxx[x], sum - siz[x]);  // maxx[x] 表示以 x 为根时的最大子树大小
```

```
    if (maxx[x] < maxx[rt])
      rt = x;  // 这里不能写 <= ，保证在第二次 calcsiz 时 rt 不改变
}

void dfs1(int x, int fa, int y, int d) {
  ch.update(ch.rt[y], 0, n, d, val[x]);
  for (int j = h[x]; j; j = nxt[j])
    if (p[j] != fa && !vis[p[j]]) dfs1(p[j], x, y, d + 1);
}

void dfs2(int x, int fa, int y, int d) {
  dist.update(dist.rt[y], 0, n, d, val[x]);
  for (int j = h[x]; j; j = nxt[j])
    if (p[j] != fa && !vis[p[j]]) dfs2(p[j], x, y, d + 1);
}

void pre(int x) {
  vis[x] = true;  // 表示在之后的过程中不考虑 x 这个点
  dfs2(x, -1, x, 0);
  for (int j = h[x]; j; j = nxt[j])
    if (!vis[p[j]]) {
      rt = 0;
      maxx[rt] = inf;
      sum = siz[p[j]];
      calcsiz(p[j], -1);
      calcsiz(rt, -1);  // 计算两次，第二次求出以 rt 为根时的各子树大小
      dfs1(p[j], -1, rt, 1);
      fa[rt] = x;
      dep[rt] = dep[x] + 1;
      pre(rt);  // 记录点分树上的父亲
    }
}

int main() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= n; i++) scanf("%d", val + i);
  for (int i = 1; i < n; i++)
    scanf("%d%d", &u, &v), add_edge(u, v), add_edge(v, u);
  lca.init();
  rt = 0;
  maxx[rt] = inf;
  sum = n;
  calcsiz(1, -1);
  calcsiz(rt, -1);
  pre(rt);
  // for(int i=1;i<=n;i++)printf("%d ",fa[i]);printf("\n");
  for (int i = 1; i <= n; i++)
    for (int j = i; j; j = fa[j]) d[i][dep[i] - dep[j]] = lca.dist(i, j);
  while (m--) {
    scanf("%d%d%d", &op, &x, &y);
    x ^= lstans;
    y ^= lstans;
    if (op == 0) {
      lstans = dist.query(dist.rt[x], 0, n, 0, y);
```

```
      int nww = 0;
      for (int i = x; fa[i]; i = fa[i]) {
        nww = d[x][dep[x] - dep[fa[i]]];  // lca.dist(x,fa[i]);
        lstans += dist.query(dist.rt[fa[i]], 0, n, 0, y - nww);
        lstans -= ch.query(ch.rt[i], 0, n, 0, y - nww);
      }
      printf("%d\n", lstans);
    }
    if (op == 1) {
      int nww = 0;
      dist.update(dist.rt[x], 0, n, 0, y - val[x]);
      for (int i = x; fa[i]; i = fa[i]) {
        nww = d[x][dep[x] - dep[fa[i]]];  // lca.dist(x,fa[i]);
        dist.update(dist.rt[fa[i]], 0, n, nww, y - val[x]);
        ch.update(ch.rt[i], 0, n, nww, y - val[x]);
      }
      val[x] = y;
    }
  }
  return 0;
}
```

参考资料与注释

[1] 「ZJOI2007」捉迷藏

[2] Luogu P6329【模板】点分树 | 震波

11.6.10 AHU 算法

Authors: Backl1ght

AHU 算法用于判断两棵有根树是否同构。

判断树同构外还有一种常见的做法是 树哈希。

前置知识：树基础，树的重心

建议配合参考资料里给的例子观看。

树同构的定义

有根树同构

对于两棵有根树 $T_1(V_1,E_1,r_1)$ 和 $T_2(V_2,E_2,r_2)$，如果存在一个双射 $\varphi : V_1 \to V_2$，使得

$$\forall u,v \in V_1, (u,v) \in E_1 \iff (\varphi(u),\varphi(v)) \in E_2$$

且 $\varphi(r_1) = r_2$ 成立，那么称有根树 $T_1(V_1,E_1,r_1)$ 和 $T_2(V_2,E_2,r_2)$ 同构。

无根树同构

对于两棵无根树 $T_1(V_1,E_1)$ 和 $T_2(V_2,E_2)$，如果存在一个双射 $\varphi : V_1 \to V_2$，使得

$$\forall u,v \in V_1, (u,v) \in E_1 \iff (\varphi(u),\varphi(v)) \in E_2$$

成立，那么称无根树 $T_1(V_1,E_1)$ 和 $T_2(V_2,E_2)$ 同构。

简单的说就是，如果能够通过把树 T_1 的所有节点重新标号，使得树 T_1 和树 T_2 **完全相同**，那么称这两棵树同构。

问题的转化

无根树同构问题可以转化为有根树同构问题。具体方法如下：

对于无根树 $T_1(V_1,E_1)$ 和 $T_2(V_2,E_2)$，先分别找出它们的**所有**重心。

- 如果这两棵无根树重心数量不同，那么这两棵树不同构。
- 如果这两颗无根树重心数量都为 1，分别记为 c_1 和 c_2，那么如果有根树 $T_1(V_1,E_1,c_1)$ 和有根树 $T_2(V_2,E_2,c_2)$ 同构，那么无根树 $T_1(V_1,E_1)$ 和 $T_2(V_2,E_2)$ 同构，反之则不同构。
- 如果这两颗无根树重心数量都为 2，分别记为 c_1,c_1' 和 c_2,c_2'，那么如果有根树 $T_1(V_1,E_1,c_1)$ 和有根树 $T_2(V_2,E_2,c_2)$ 同构**或者**有根树 $T_1(V_1,E_1,c_1')$ 和 $T_2(V_2,E_2,c_2)$ 同构，那么无根树 $T_1(V_1,E_1)$ 和 $T_2(V_2,E_2)$ 同构，反之则不同构。

所以，只要解决了有根树同构问题，我们就可以把无根树同构问题根据上述方法转化成有根树同构的问题，进而解决无根树同构的问题。

假设有一个可以 $O(|V|)$ 解决有根树同构问题的算法，那么根据上述方法我们也可以在 $O(|V|)$ 的时间内解决无根树同构问题。

朴素的 AHU 算法

朴素的 AHU 算法是基于括号序的。

原理 1

我们知道一段合法的括号序和一棵有根树唯一对应，而且一棵树的括号序是由它的子树的括号序拼接而成的。如果我们通过改变子树括号序拼接的顺序，从而获得了一段新的括号序，那么新括号序对应的树和原括号序对应的树同构。

原理 2

树的同构关系是传递的。既如果 T_1 和 T_2 同构，T_2 和 T_3 同构，那么 T_1 和 T_3 同构。

推论

考虑求树括号序的递归算法，我们在回溯时拼接子树的括号序。如果在拼接的时候将字典序小的序列先拼接，并将最后的结果记为 $NAME$。

将以节点 r 为根的子树的 $NAME$ 作为节点 r 的 $NAME$，记为 $NAME(r)$，那么对于有根树 $T_1(V_1,E_1,r_1)$ 和 $T_2(V_2,E_2,r_2)$，如果 $NAME(r_1)=NAME(r_2)$，那么 T_1 和 T_2 同构。

命名算法

" 实现"

```
Input. A rooted tree T
Output. The name of rooted tree T
ASSIGN-NAME(u)
    if u is a leaf
        NAME(u) = (0)
    else
        for all child v of u
            ASSIGN-NAME(v)
    sort the names of the children of u
    concatenate the names of all children u to temp
    NAME(u) = (temp)
```

AHU 算法

" 实现"

```
Input. Two rooted trees T1(V1, E1, r1) and T2(V2, E2, r2)
Output. Whether these two trees are isomorphic
AHU(T1(V1, E1, r1), T2(V2, E2, r2))
    ASSIGN-NAME(r1)
    ASSIGN-NAME(r2)
    if NAME(r1) = NAME(r2)
        return true
    else
        return false
```

复杂度证明

对于一颗有 n 个节点的有根树，假设他是链状的，那么节点名字长度最长可以是 n，那么 ASSIGN-NAME 算法的复杂度是 $1+2+\cdots+n$ 的常数倍，即 $\Theta(n^2)$。由此，朴素 AHU 算法的复杂度为 $O(n^2)$。

优化的 AHU 算法

朴素的 AHU 算法的缺点是树的 $NAME$ 的长度可能会过长，我们可以针对这一点做一些优化。

原理 1

对树进行层次划分，第 i 层的节点到根的最短距离为 i。位于第 i 层的节点的 $NAME$ 可以**只**由位于第 $i+1$ 层的节点的 $NAME$ 拼接得到。

原理 2

在同一层内，节点的 $NAME$ 可以由其在层内的排名唯一标识。

注意，这里的排名是对两棵树而言的，假设节点 u 位于第 i 层，那么节点 u 的排名等于所有 T_1 和 T_2 第 i 层的节点中 $NAME$ 比 $NAME(u)$ 小的节点的个数。

推论

我们可以将节点原来的 $NAME$ 用其在层内的排名代替，然后把原来拼接节点 $NAME$ 用向数组加入元素代替。

这样用整数和数组来代替字符串，既不会影响算法的正确性，又很大的降低了算法的复杂度。

复杂度证明

首先注意到第 i 层由拼接得到的 $NAME$ 的总长度为第 i 层点的度数之和，即第 $i+1$ 层的总点数，以下用 L_i 表示。算法的下一步会将这些 $NAME$ 看成字符串（数组）并排序，然后将它们替换为其在层内的排名（即重新映射为一个数）。以下引理表明了对总长为 L 的 m 个字符串排序的复杂度：

1. 我们可以使用基数排序在 $O(L+|\Sigma|)$ 的时间内完成排序，其中 $|\Sigma|$ 为字符集的大小。（有一些实现细节，参见参考资料）
2. 我们可以使用快速排序在 $O(L\log m)$ 的时间内完成排序。证明的大致思路为快排递归树的高度为 $O(\log m)$，且暴力比较长度为 ℓ_1 和 ℓ_2 的两个字符串的复杂度为 $O(\min\{\ell_1,\ell_2\})$。

在 AHU 算法中，第 i 层字符串的字符集大小最多为第 $i+1$ 层的点数，即 L_i，所以基数排序的复杂度是线性的。根据 $\sum_i L_i = O(n)$，并将每层的复杂度相加后可以看出，若使用字符串的基数排序，则算法的总复杂度为 $T(n)=O(n)$。同理，如果使用快排排序字符串，那么 $T(n)=O(n\log n)$。

例题

SPOJ-TREEISO[1]

题意翻译：给你两颗无根树，判断两棵树是否同构。

" 参考代码"

```
// Tree Isomorphism, O(nlogn)
// replace quick sort with radix sort ==> O(n)
// Author: _Backl1ght
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N = 1e5 + 5;
const int maxn = N << 1;

int n;

struct Edge {
  int v, nxt;
} e[maxn << 1];

int head[maxn], sz[maxn], f[maxn], maxv[maxn], tag[maxn], tot, Max;
vector<int> center[2], L[maxn], subtree_tags[maxn];

void addedge(int u, int v) {  // 建图
  e[tot].v = v;
  e[tot].nxt = head[u];
  head[u] = tot++;
  e[tot].v = u;
  e[tot].nxt = head[v];
  head[v] = tot++;
}

void dfs_size(int u, int fa) {  // 找到 size 值
  sz[u] = 1;
  maxv[u] = 0;
  for (int i = head[u]; i; i = e[i].nxt) {
    int v = e[i].v;
    if (v == fa) continue;
    dfs_size(v, u);
    sz[u] += sz[v];
    maxv[u] = max(maxv[u], sz[v]);
  }
}

void dfs_center(int rt, int u, int fa, int id) {
  maxv[u] = max(maxv[u], sz[rt] - sz[u]);
  if (Max > maxv[u]) {
    center[id].clear();
    Max = maxv[u];
  }
  if (Max == maxv[u]) center[id].push_back(u);  // 如果相等就 push_back
  for (int i = head[u]; i; i = e[i].nxt) {
```

```
    int v = e[i].v;
    if (v == fa) continue;
    dfs_center(rt, v, u, id);
  }
}

int dfs_height(int u, int fa, int depth) {  // 递归查找 height
  L[depth].push_back(u);
  f[u] = fa;
  int h = 0;
  for (int i = head[u]; i; i = e[i].nxt) {
    int v = e[i].v;
    if (v == fa) continue;
    h = max(h, dfs_height(v, u, depth + 1));
  }
  return h + 1;
}

void init(int n) {  // 一开始的处理
  for (int i = 1; i <= 2 * n; i++) head[i] = 0;
  tot = 1;
  center[0].clear();
  center[1].clear();

  int u, v;
  for (int i = 1; i <= n - 1; i++) {
    scanf("%d %d", &u, &v);
    addedge(u, v);
  }
  dfs_size(1, -1);
  Max = n;
  dfs_center(1, 1, -1, 0);

  for (int i = 1; i <= n - 1; i++) {
    scanf("%d %d", &u, &v);
    addedge(u + n, v + n);
  }
  dfs_size(1 + n, -1);
  Max = n;
  dfs_center(1 + n, 1 + n, -1, 1);
}

bool cmp(int u, int v) { return subtree_tags[u] < subtree_tags[v]; }

bool rootedTreeIsomorphism(int rt1, int rt2) {
  for (int i = 0; i <= 2 * n + 1; i++) L[i].clear(), subtree_tags[i].clear();
  int h1 = dfs_height(rt1, -1, 0);
  int h2 = dfs_height(rt2, -1, 0);
  if (h1 != h2) return false;
  int h = h1 - 1;
  for (int j = 0; j < (int)L[h].size(); j++) tag[L[h][j]] = 0;
  for (int i = h - 1; i >= 0; i--) {
    for (int j = 0; j < (int)L[i + 1].size(); j++) {
      int v = L[i + 1][j];
```

```
      subtree_tags[f[v]].push_back(tag[v]);
    }

    sort(L[i].begin(), L[i].end(), cmp);

    for (int j = 0, cnt = 0; j < (int)L[i].size(); j++) {
      if (j && subtree_tags[L[i][j]] != subtree_tags[L[i][j - 1]]) ++cnt;
      tag[L[i][j]] = cnt;
    }
  }
  return subtree_tags[rt1] == subtree_tags[rt2];
}

bool treeIsomorphism() {
  if (center[0].size() == center[1].size()) {
    if (rootedTreeIsomorphism(center[0][0], center[1][0])) return true;
    if (center[0].size() > 1)
      return rootedTreeIsomorphism(center[0][0], center[1][1]);
  }
  return false;
}

int main() {
  int T;
  scanf("%d", &T);
  while (T--) {
    scanf("%d", &n);
    init(n);
    puts(treeIsomorphism() ? "YES" : "NO");
  }
  return 0;
}
```

参考资料

本文大部分内容译自 Paper[2] 和 Slide[3]。参考资料里的证明会更加全面和严谨，本文做了一定的简化。

对 AHU 算法的复杂度分析，以及字符串的线性时间基数排序算法可以参见 The Design and Analysis of Computer Algorithms 的 3.2 节 Radix sorting，以及其中的 Example 3.2。

参考资料与注释

[1] SPOJ-TREEISO

[2] Paper

[3] Slide

11.6.11 树哈希

判断一些树是否同构的时，我们常常把这些树转成哈希值储存起来，以降低复杂度。

树哈希是很灵活的，可以设计出各种各样的哈希方式；但是如果随意设计，很有可能是错误的，可能被卡。以下介绍一类容易实现且不易被卡的方法。

方法

这类方法需要一个多重集的哈希函数。以某个结点为根的子树的哈希值，就是以它的所有儿子为根的子树的哈希值构成的多重集的哈希值，即：

$$h_x = f(\{h_i \mid i \in son(x)\})$$

其中 h_x 表示以 x 为根的子树的哈希值，f 是多重集的哈希函数。

以代码中使用的哈希函数为例：

$$f(S) = \left(c + \sum_{x \in S} g(x)\right) \bmod m$$

其中 c 为常数，一般使用 1 即可。m 为模数，一般使用 2^{32} 或 2^{64} 进行自然溢出，也可使用大素数。g 为整数到整数的映射，代码中使用 xor shift，也可以选用其他的函数，但是不建议使用多项式。为了预防出题人对着 xor hash 卡，还可以在映射前后异或一个随机常数。

这种哈希十分好写。如果需要换根，第二次 DP 时只需把子树哈希减掉即可。

例题

UOJ #763. 树哈希[1]

这是一道模板题。不用多说，以 1 为根跑一遍 DFS 就好了。

" 参考代码"

```cpp
#include <cctype>
#include <chrono>
#include <cstdio>
#include <random>
#include <set>
#include <vector>

typedef unsigned long long ull;

const ull mask = std::chrono::steady_clock::now().time_since_epoch().count();

ull shift(ull x) {
  x ^= mask;
  x ^= x << 13;
  x ^= x >> 7;
  x ^= x << 17;
  x ^= mask;
  return x;
}

const int N = 1e6 + 10;

int n;
ull hash[N];
std::vector<int> edge[N];
std::set<ull> trees;

void getHash(int x, int p) {
  hash[x] = 1;
  for (int i : edge[x]) {
    if (i == p) {
```

```
      continue;
    }
    getHash(i, x);
    hash[x] += shift(hash[i]);
  }
  trees.insert(hash[x]);
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d%d", &u, &v);
    edge[u].push_back(v);
    edge[v].push_back(u);
  }
  getHash(1, 0);
  printf("%lu", trees.size());
}
```

[BJOI2015] 树的同构[2]

这道题所说的同构是指无根树的，而上面所介绍的方法是针对有根树的。因此只有当根一样时，同构的两棵无根树哈希值才相同。由于数据范围较小，我们可以暴力求出以每个点为根时的哈希值，排序后比较。

如果数据范围较大，我们也可以使用换根 DP，遍历树两遍，求出以每个点为根时的哈希值。我们还可以利用上面的多重集哈希函数：把以每个结点为根时的哈希值都存进多重集，再把多重集的哈希值算出来，进行比较（做法一）。

还可以通过找重心的方式来优化复杂度。一棵树的重心最多只有两个，只需把以它（们）为根时的哈希值求出来即可。接下来，既可以分别比较这些哈希值（做法二），也可以在有一个重心时取它的哈希值作为整棵树的哈希值，有两个时则取其中较小（大）的。

" 做法一"

```
#include <chrono>
#include <cstdio>
#include <map>
#include <random>
#include <set>
#include <vector>

typedef unsigned long long ull;

const int N = 60, M = 998244353;
const ull mask = std::chrono::steady_clock::now().time_since_epoch().count();

ull shift(ull x) {
  x ^= mask;
  x ^= x << 13;
  x ^= x >> 7;
  x ^= x << 17;
  x ^= mask;
  return x;
}
```

```
std::vector<int> edge[N];
ull sub[N], root[N];
std::map<ull, int> trees;

void getSub(int x) {
  sub[x] = 1;
  for (int i : edge[x]) {
    getSub(i);
    sub[x] += shift(sub[i]);
  }
}

void getRoot(int x) {
  for (int i : edge[x]) {
    root[i] = sub[i] + shift(root[x] - shift(sub[i]));
    getRoot(i);
  }
}

int main() {
  int m;
  scanf("%d", &m);
  for (int t = 1; t <= m; t++) {
    int n, rt = 0;
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
      int fa;
      scanf("%d", &fa);
      if (fa) {
        edge[fa].push_back(i);
      } else {
        rt = i;
      }
    }
    getSub(rt);
    root[rt] = sub[rt];
    getRoot(rt);
    ull hash = 1;
    for (int i = 1; i <= n; i++) {
      hash += shift(root[i]);
    }
    if (!trees.count(hash)) {
      trees[hash] = t;
    }
    printf("%d\n", trees[hash]);
    for (int i = 1; i <= n; i++) {
      edge[i].clear();
    }
  }
}
```

" 做法二"

```cpp
#include <chrono>
#include <cstdio>
#include <map>
#include <random>
#include <set>
#include <vector>

typedef unsigned long long ull;
typedef std::pair<ull, ull> Hash2;

const int N = 60, M = 998244353;
const ull mask = std::chrono::steady_clock::now().time_since_epoch().count();

ull shift(ull x) {
  x ^= mask;
  x ^= x << 13;
  x ^= x >> 7;
  x ^= x << 17;
  x ^= mask;
  return x;
}

int n;
int size[N], weight[N], centroid[2];
std::vector<int> edge[N];
std::map<Hash2, int> trees;

void getCentroid(int x, int fa) {
  size[x] = 1;
  weight[x] = 0;
  for (int i : edge[x]) {
    if (i == fa) {
      continue;
    }
    getCentroid(i, x);
    size[x] += size[i];
    weight[x] = std::max(weight[x], size[i]);
  }
  weight[x] = std::max(weight[x], n - size[x]);
  if (weight[x] <= n / 2) {
    int index = centroid[0] != 0;
    centroid[index] = x;
  }
}

ull getHash(int x, int fa) {
  ull hash = 1;
  for (int i : edge[x]) {
    if (i == fa) {
      continue;
    }
    hash += shift(getHash(i, x));
  }
```

```
  return hash;
}

int main() {
  int m;
  scanf("%d", &m);
  for (int t = 1; t <= m; t++) {
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
      int fa;
      scanf("%d", &fa);
      if (fa) {
        edge[fa].push_back(i);
        edge[i].push_back(fa);
      }
    }
    getCentroid(1, 0);
    Hash2 hash;
    hash.first = getHash(centroid[0], 0);
    if (centroid[1]) {
      hash.second = getHash(centroid[1], 0);
      if (hash.first > hash.second) {
        std::swap(hash.first, hash.second);
      }
    } else {
      hash.second = hash.first;
    }
    if (!trees.count(hash)) {
      trees[hash] = t;
    }
    printf("%d\n", trees[hash]);
    for (int i = 1; i <= n; i++) {
      edge[i].clear();
    }
    centroid[0] = centroid[1] = 0;
  }
}
```

HDU 6647 Bracket Sequences on Tree[3]

题目要求遍历一棵无根树产生的本质不同括号序列方案数。

首先可以注意到，两棵不同构的有根树一定不会生成相同的括号序列。我们先考虑遍历有根树能够产生的本质不同括号序列方案数，假设我们当前考虑的子树根节点为 u，记 $f(u)$ 表示这棵子树的方案数，从 u 开始往下遍历，顺序可以随意选择，产生 $|son(u)|!$ 种排列，遍历每个儿子节点 v，v 的子树内有 $f(v)$ 种方案，因此有 $f(u) = |son(u)|! \cdot \prod_{v \in son(u)} f(v)$。但是，同构的子树之间会产生重复，$f(u)$ 需要除掉每种本质不同子树出现次数阶乘的乘积，类似于多重集合的排列。

通过上述 DP，可以求出根节点的方案数。再通过换根 DP，将父亲节点的哈希值和方案信息转移给儿子，可以求出以每个节点为根时的哈希值和方案数。每种不同的子树只需要计数一次即可。

"参考代码"

```
#include <chrono>
#include <cstdio>
```

```cpp
#include <map>
#include <random>
#include <set>
#include <vector>

typedef unsigned long long ull;

const int N = 1e5 + 10, M = 998244353;
const ull mask = std::chrono::steady_clock::now().time_since_epoch().count();

struct Tree {
  ull hash, deg, ans;
  std::map<ull, ull> son;

  Tree() { clear(); }

  void add(Tree& o);
  void remove(Tree& o);
  void clear();
};

ull inv(ull x) {
  ull y = M - 2, z = 1;
  while (y) {
    if (y & 1) {
      z = z * x % M;
    }
    x = x * x % M;
    y >>= 1;
  }
  return z;
}

ull shift(ull x) {
  x ^= mask;
  x ^= x << 13;
  x ^= x >> 7;
  x ^= x << 17;
  x ^= mask;
  return x;
}

void Tree::add(Tree& o) {
  ull temp = shift(o.hash);
  hash += temp;
  ans = ans * ++deg % M * inv(++son[temp]) % M * o.ans % M;
}

void Tree::remove(Tree& o) {
  ull temp = shift(o.hash);
  hash -= temp;
  ans = ans * inv(deg--) % M * son[temp]-- % M * inv(o.ans) % M;
}
```

```
void Tree::clear() {
  hash = 1;
  deg = 0;
  ans = 1;
  son.clear();
}

std::vector<int> edge[N];
Tree sub[N], root[N];
std::map<ull, ull> trees;

void getSub(int x, int fa) {
  for (int i : edge[x]) {
    if (i == fa) {
      continue;
    }
    getSub(i, x);
    sub[x].add(sub[i]);
  }
}

void getRoot(int x, int fa) {
  for (int i : edge[x]) {
    if (i == fa) {
      continue;
    }
    root[x].remove(sub[i]);
    root[i] = sub[i];
    root[i].add(root[x]);
    root[x].add(sub[i]);
    getRoot(i, x);
  }
  trees[root[x].hash] = root[x].ans;
}

int main() {
  int t, n;
  scanf("%d", &t);
  while (t--) {
    scanf("%d", &n);
    for (int i = 1; i < n; i++) {
      int u, v;
      scanf("%d%d", &u, &v);
      edge[u].push_back(v);
      edge[v].push_back(u);
    }
    getSub(1, 0);
    root[1] = sub[1];
    getRoot(1, 0);
    ull tot = 0;
    for (auto p : trees) {
      tot = (tot + p.second) % M;
    }
    printf("%lld\n", tot);
```

```
      for (int i = 1; i <= n; i++) {
        edge[i].clear();
        sub[i].clear();
        root[i].clear();
      }
      trees.clear();
    }
  }
```

参考资料

文中的哈希方法参考并拓展自博客 一种好写且卡不掉的树哈希[4]。

参考资料与注释

[1] UOJ #763. 树哈希

[2] [BJOI2015] 树的同构

[3] HDU 6647 Bracket Sequences on Tree

[4] 一种好写且卡不掉的树哈希

11.6.12 树上随机游走

给定一棵有根树，树的某个结点上有一个硬币，在某一时刻硬币会等概率地移动到邻接结点上，问硬币移动到邻接结点上的期望距离。

需要用到的定义

- $T = (V, E)$: 所讨论的树
- $d(u)$: 结点 u 的度数
- $w(u, v)$: 结点 u 与结点 v 之间的边的边权
- p_u: 结点 u 的父结点
- $root$: 树的根结点
- son_u: 结点 u 的子结点集合
- $sibling_u$: 结点 u 的兄弟结点集合

向父结点走的期望距离

设 $f(u)$ 代表 u 结点走到其父结点 p_u 的期望距离，则有：

$$f(u) = \frac{w(u, p_u) + \sum\limits_{v \in son_u} (w(u, v) + f(v) + f(u))}{d(u)}$$

分子中的前半部分代表直接走向了父结点，后半部分代表先走向了子结点再由子结点走回来然后再向父结点走；分母 $d(u)$ 代表从 u 结点走向其任何邻接点的概率相同。

化简如下：

$$
\begin{aligned}
f(u) &= \frac{w(u,p_u)+\sum\limits_{v\in son_u}(w(u,v)+f(v)+f(u))}{d(u)} \\
&= \frac{w(u,p_u)+\sum\limits_{v\in son_u}(w(u,v)+f(v))+(d(u)-1)f(u)}{d(u)} \\
&= w(u,p_u)+\sum_{v\in son_u}(w(u,v)+f(v)) \\
&= \sum_{(u,t)\in E} w(u,t)+\sum_{v\in son_u} f(v)
\end{aligned}
$$

对于叶子结点 l，初始状态为 $f(l)=w(p_l,l)$。

当树上所有边的边权都为 1 时，上式可化为：

$$
f(u)=d(u)+\sum_{v\in son_u} f(v)
$$

即 u 子树的所有结点的度数和，也即 u 子树大小的两倍 -1（每个结点连向其父亲的边都有且只有一条，除 u 与 p_u 之间的边只有 1 点度数的贡献外，每条边会产生 2 点度数的贡献）。

向子结点走的期望距离

设 $g(u)$ 代表 p_u 结点走到其子结点 u 的期望距离，则有：

$$
g(u)=\frac{w(p_u,u)+\Big(w(p_u,p_{p_u})+g(p_u)+g(u)\Big)+\sum\limits_{s\in sibling_u}(w(p_u,s)+f(s)+g(u))}{d(p_u)}
$$

分子中的第一部分代表直接走向了子结点 u，第二部分代表先走向了父结点再由父结点走回来然后再向 u 结点走，第三部分代表先走向 u 结点的兄弟结点再由其走回来然后再向 u 结点走；分母 $d(p_u)$ 代表从 p_u 结点走向其任何邻接点的概率相同。

化简如下：

$$
\begin{aligned}
g(u) &= \frac{w(p_u,u)+\Big(w(p_u,p_{p_u})+g(p_u)+g(u)\Big)+\sum\limits_{s\in sibling_u}(w(p_u,s)+f(s)+g(u))}{d(p_u)} \\
&= \frac{w(p_u,u)+w(p_u,p_{p_u})+g(p_u)+\sum\limits_{s\in sibling_u}(w(p_u,s)+f(s))+(d(p_u)-1)g(u)}{d(p_u)} \\
&= w(p_u,u)+w(p_u,p_{p_u})+g(p_u)+\sum_{s\in sibling_u}(w(p_u,s)+f(s)) \\
&= \sum_{(p_u,t)\in E} w(p_u,t)+g(p_u)+\sum_{s\in sibling_u} f(s) \\
&= \sum_{(p_u,t)\in E} w(p_u,t)+g(p_u)+\left(f(p_u)-\sum_{(p_u,t)\in E} w(p_u,t)-f(u)\right) \\
&= g(p_u)+f(p_u)-f(u)
\end{aligned}
$$

初始状态为 $g(\mathrm{root})=0$。

代码实现（以无权树为例）

```cpp
vector<int> G[maxn];

void dfs1(int u, int p) {
  f[u] = G[u].size();
  for (auto v : G[u]) {
    if (v == p) continue;
    dfs1(v, u);
    f[u] += f[v];
  }
}

void dfs2(int u, int p) {
  if (u != root) g[u] = g[p] + f[p] - f[u];
  for (auto v : G[u]) {
    if (v == p) continue;
    dfs2(v, u);
  }
}
```

11.7 矩阵树定理

Authors: pw384, s0cks5, Xeonacid

Kirchhoff 矩阵树定理（简称矩阵树定理）解决了一张图的生成树个数计数问题。

本篇记号声明

本篇中的图，无论无向还是有向，都允许重边，但是不允许自环。

无向图情况

设 G 是一个有 n 个顶点的无向图。定义度数矩阵 $D(G)$ 为：

$$D_{ii}(G) = \deg(i), D_{ij} = 0, i \neq j$$

设 $\#e(i,j)$ 为点 i 与点 j 相连的边数，并定义邻接矩阵 A 为：

$$A_{ij}(G) = A_{ji}(G) = \#e(i,j), i \neq j$$

定义 Laplace 矩阵（亦称 Kirchhoff 矩阵）L 为：

$$L(G) = D(G) - A(G)$$

记图 G 的所有生成树个数为 $t(G)$。

有向图情况

设 G 是一个有 n 个顶点的有向图。定义出度矩阵 $D^{out}(G)$ 为：

$$D^{out}_{ii}(G) = \deg^{\mathrm{out}}(i), D^{out}_{ij} = 0, i \neq j$$

类似地定义入度矩阵 $D^{in}(G)$

设 $\#e(i,j)$ 为点 i 指向点 j 的有向边数，并定义邻接矩阵 A 为：

$$A_{ij}(G) = \#e(i,j), i \neq j$$

定义出度 Laplace 矩阵 L^{out} 为：

$$L^{out}(G) = D^{out}(G) - A(G)$$

定义入度 Laplace 矩阵 L^{in} 为：

$$L^{in}(G) = D^{in}(G) - A(G)$$

记图 G 的以 r 为根的所有根向树形图个数为 $t^{root}(G,r)$。所谓根向树形图，是说这张图的基图是一棵树，所有的边全部指向父亲。

记图 G 的以 r 为根的所有叶向树形图个数为 $t^{leaf}(G,r)$。所谓叶向树形图，是说这张图的基图是一棵树，所有的边全部指向儿子。

定理叙述

矩阵树定理具有多种形式。其中用得较多的是定理 1、定理 3 与定理 4。

定理 1（矩阵树定理，无向图行列式形式） 对于任意的 i，都有

$$t(G) = \det L(G)\begin{pmatrix}1,2,\cdots,i-1,i+1,\cdots,n\\1,2,\cdots,i-1,i+1,\cdots,n\end{pmatrix}$$

其中记号 $L(G)\binom{1,2,\cdots,i-1,i+1,\cdots,n}{1,2,\cdots,i-1,i+1,\cdots,n}$ 表示矩阵 $L(G)$ 的第 $1,\cdots,i-1,i+1,\cdots,n$ 行与第 $1,\cdots,i-1,i+1,\cdots,n$ 列构成的子矩阵。也就是说，无向图的 Laplace 矩阵具有这样的性质，它的所有 $n-1$ 阶主子式都相等。

定理 2（矩阵树定理，无向图特征值形式） 设 $\lambda_1,\lambda_2,\cdots,\lambda_{n-1}$ 为 $L(G)$ 的 $n-1$ 个非零特征值，那么有 $t(G) = \frac{1}{n}\lambda_1\lambda_2\cdots\lambda_{n-1}$

定理 3（矩阵树定理，有向图根向形式） 对于任意的 k，都有

$$t^{root}(G,k) = \det L^{out}(G)\begin{pmatrix}1,2,\cdots,k-1,k+1,\cdots,n\\1,2,\cdots,k-1,k+1,\cdots,n\end{pmatrix}$$

因此如果要统计一张图所有的根向树形图，只要枚举所有的根 k 并对 $t^{root}(G,k)$ 求和即可。

定理 4（矩阵树定理，有向图叶向形式） 对于任意的 k，都有

$$t^{leaf}(G,k) = \det L^{in}(G)\begin{pmatrix}1,2,\cdots,k-1,k+1,\cdots,n\\1,2,\cdots,k-1,k+1,\cdots,n\end{pmatrix}$$

因此如果要统计一张图所有的叶向树形图，只要枚举所有的根 k 并对 $t^{leaf}(G,k)$ 求和即可。

BEST 定理

定理 5（BEST 定理） 设 G 是有向欧拉图，那么 G 的不同欧拉回路总数 $ec(G)$ 是

$$ec(G) = t^{root}(G,k)\prod_{v\in V}(\deg(v)-1)!$$

注意，对欧拉图 G 的任意两个节点 k,k'，都有 $t^{root}(G,k) = t^{root}(G,k')$，且欧拉图 G 的所有节点的入度和出度相等。

定理证明

前置知识：LGV 引理

定义 $[n] = \{1,2,\cdots,n\}$，矩阵 A 的子式 $A_{[S],[T]}$ 为选取 $A_{i,j}$ $(i\in S, j\in T)$ 的元素得到的子矩阵。

定义一条边 e 的权值为 $\omega(e)$。

以下的内向也指根向，表示有向边的方向指向根。

引理 1（Cauchy–Binet） 给定 $n\times m$ 的矩阵 A 和 $m\times n$ 的矩阵 B，则有

$$|AB|=\sum_{|S|=n,S\subseteq[m]}|A_{[n],[S]}||B_{[S],[n]}|$$

证明：考虑建出有向无环图 $G(V,E)$，$V=L\cup R\cup D$，

$L=\{l_1,l_2,\cdots,l_n\}$，$R=\{r_1,r_2,\cdots,r_n\}$，

$D=\{d_1,d_2,\cdots,d_m\}$，$E=E_L\cup E_R$，

$E_L=\{(l_i,d_j)\mid i\in[1,n],j\in[1,m]\}$，$E_R=\{(d_i,r_j)\mid i\in[1,m],j\in[1,n]\}$，

$\omega(l_i,d_j)=a_{i,j}$，$\omega(d_i,r_j)=b_{i,j}$。

与「NOI2021」路径交点[1] 的模型相同，容易发现上式左右两侧计算的都是 L 到 R 的不相交路径组中，交点个数为偶数的方案数减去奇数的方案数，其中 S 表示路径组经过的 D 中的点。

性质 1 Laplace 矩阵 L 的所有代数余子式 $C_{i,j}$ 的值都相等。

证明：删去第 i 行后，设列向量是 $r_1,r_2,\cdots,r_n$，则有 $\sum r_i=0$。

将余子式 $M_{i,j}$ 中除了 r_k 之外的所有 r_i 都加到 r_k 上，得到 $A=[r_1,\cdots,r_{j-1},r_{j+1},\cdots,r_{k-1},-r_j,r_{k+1},\cdots,r_n]$。

将 $-r_j$ 取反并通过交换两列移动到 r_{j+1} 左边，得到 $|A|=M_{i,k}=(-1)^{1+(k-1)-(j+1)+1}M_{i,j}$，所以 $C_{i,j}=C_{i,k}$。

同理，删去第 i 列后行向量之和为 0，得到 $C_{j,i}=C_{k,i}$。

定理 1（Kirchhoff's Matrix Tree） 对于带边权的简单无向图 $G(V,E)$，若 $T(V,E_T)$ 是 G 的生成树，定义 $\omega(T)=\prod_{e\in E_T}\omega(e)$，$\mathcal{T}$ 是 G 所有生成树的集合，则 G 的 Laplace 矩阵的所有代数余子式的值等于 $\sum_{T\in\mathcal{T}}\omega(T)$。

证明：根据性质 1，只需证明 $C_{1,1}=\sum_{T\in\mathcal{T}}\omega(T)$。对于一条边 $e=(u,v)$，定义 $\zeta(e,u)=v$，$\zeta(e,v)=u$。

定义 $u_i<u_j\iff i<j$，$E=\{e_1,e_2,\cdots,e_{|E|}\}$，构造关联矩阵：

$$A_{i,j}=\begin{cases}\sqrt{\omega(e_j)} & u_i\in e_j\wedge u_i<\zeta(e_j,u_i)\\ -\sqrt{\omega(e_j)} & u_i\in e_j\wedge u_i>\zeta(e_j,u_i)\\ 0 & \text{otherwise}\end{cases}$$

容易发现 $L=AA^T$，定义 A 删去第一行得到 B，则 $M_{1,1}=BB^T$。代入 Cauchy–Binet 公式得到：

$$M_{1,1}=\sum_{|S|=n-1,S\subseteq[|E|]}|B_{[n-1],[S]}||(B^T)_{[S],[n-1]}|=\sum_{|S|=n-1,S\subseteq[|E|]}|B_{[n-1],[S]}|^2$$

$|B_{[n-1],[S]}|$ 的组合意义是对点 $u_2,\cdots,u_n$，分别选一条 S 中的边，且每条边都恰好被选一次。若 u_i 选择了 e_j，就看做有向边 $(u_i,\zeta(e_j,u_i))$，所以也相当于给 S 中的边定向，使得 $u_2,\cdots,u_n$ 的出度为 1。

对于存在环的方案，构造对合映射（满足 $f(f(x))=x$ 的映射 f）：取环上点编号最小值最小的环（容易发现环的点不交，所以这样的环唯一），将这个环上的边反向。

若环长为奇数，则排列奇偶性不变，关联矩阵中系数符号变化了奇数个；若环长为偶数，则排列奇偶性改变，关联矩阵中系数符号变化了偶数个。所以贡献值相反，出现环的权值都被两两抵消，对行列值没有贡献。

于是只用考虑不存在环的情况，此时有向图只能是以 1 为根的内向树，此时定向方案唯一（确定了边集和根），也就是每个点选择的出边都唯一，所以 $|B_{[n-1],[S]}|^2$ 即为该树的边权积，求和就得到 $\sum_{T\in\mathcal{T}}\omega(T)$。

性质 2 设 Laplace 矩阵的特征值为 $\lambda_1\geq\lambda_2\geq\cdots\geq\lambda_{|V|}$，则 $\lambda_{|V|}=0$。

证明：因为 $L=AA^T$，所以 L 是半正定矩阵，特征值都非负。而 $|M|=0$，所以必定有 $\lambda=0$。

定义 k- 生成森林是图的一个生成子图 (V,E)，使得这个子图有 k 个连通分量且无环。

定理 2 定义 $\mathcal{T}_k$ 表示无向图 G 的 k- 生成森林的集合，$Q(T)$ 表示森林 T 的每个连通分量的点数之积，L 的特征多项式为 $P(x)$，则有

$$F_k=\sum_{T\in\mathcal{T}_k}\omega(T)Q(T)=(-1)^{|V|-k}[x^k]P(x)$$

证明：考虑 $P(x)=\det(xI-L)$，枚举排列行列式时，贡献到 $[x^k]$ 相当于选择相同编号的 k 行 k 列删去，这些就是每个连通分量的根，其他点选择出边连到这些根（类似定理 1 的证明），$(-1)^{|V|-k}$ 表示将负号去掉。

推论 $F_1=\prod_{i=1}^{|V|-1}\lambda_i$ 是生成树个数的 n 倍。

定理 3 内向生成树计数（见上）

证明：发现定理 1 的证明中用到了两个关联矩阵，于是我们考虑使用两个不同的关联矩阵 A,B 承担不同的功能。

$$A_{i,j}=\begin{cases}\sqrt{\omega(e_j)} & u_i \text{ is } e_j\text{'s head}\\ -\sqrt{\omega(e_j)} & u_i \text{ is } e_j\text{'s tail}\\ 0 & \text{otherwise}\end{cases}$$

$$B_{i,j}=\begin{cases}\sqrt{\omega(e_j)} & u_i \text{ is } e_j\text{'s head}\\ 0 & \text{otherwise}\end{cases}$$

与定理 1 中不同的是，关联矩阵 B 限制了只有边的起点能选择这条边，剩下的讨论均与定理 1 相同。

实现

一个无向图的生成树个数为邻接矩阵度数矩阵去一行一列的行列式，可以使用 Gauss–Jordan 消元法。

例如，一个正方形图的生成树个数

$$\begin{pmatrix}2&0&0&0\\0&2&0&0\\0&0&2&0\\0&0&0&2\end{pmatrix}-\begin{pmatrix}0&1&0&1\\1&0&1&0\\0&1&0&1\\1&0&1&0\end{pmatrix}=\begin{pmatrix}2&-1&0&-1\\-1&2&-1&0\\0&-1&2&-1\\-1&0&-1&2\end{pmatrix}$$

$$\begin{vmatrix}2&-1&0\\-1&2&-1\\0&-1&2\end{vmatrix}=4$$

可以用高斯消元解决，时间复杂度为 $O(n^3)$。

" 实现"

```
#include <algorithm>
#include <cassert>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <iostream>
using namespace std;
#define MOD 100000007
#define eps 1e-7

struct matrix {
  static const int maxn = 20;
  int n, m;
  double mat[maxn][maxn];

  matrix() { memset(mat, 0, sizeof(mat)); }

  void print() {
    cout << "MATRIX " << n << " " << m << endl;
    for (int i = 0; i < n; i++) {
```

```
      for (int j = 0; j < m; j++) {
        cout << mat[i][j] << "\t";
      }
      cout << endl;
    }
  }

  void random(int n) {
    this->n = n;
    this->m = n;
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++) mat[i][j] = rand() % 100;
  }

  void initSquare() {
    this->n = 4;
    this->m = 4;
    memset(mat, 0, sizeof(mat));
    mat[0][1] = mat[0][3] = 1;
    mat[1][0] = mat[1][2] = 1;
    mat[2][1] = mat[2][3] = 1;
    mat[3][0] = mat[3][2] = 1;
    mat[0][0] = mat[1][1] = mat[2][2] = mat[3][3] = -2;
    this->n--;  // 去一行
    this->m--;  // 去一列
  }

  double gauss() {
    double ans = 1;
    for (int i = 0; i < n; i++) {
      int sid = -1;
      for (int j = i; j < n; j++)
        if (abs(mat[j][i]) > eps) {
          sid = j;
          break;
        }
      if (sid == -1) continue;
      if (sid != i) {
        for (int j = 0; j < n; j++) {
          swap(mat[sid][j], mat[i][j]);
          ans = -ans;
        }
      }
      for (int j = i + 1; j < n; j++) {
        double ratio = mat[j][i] / mat[i][i];
        for (int k = 0; k < n; k++) {
          mat[j][k] -= mat[i][k] * ratio;
        }
      }
    }
    for (int i = 0; i < n; i++) ans *= mat[i][i];
    return abs(ans);
  }
};
```

```cpp
int main() {
  srand(1);
  matrix T;
  // T.random(2);
  T.initSquare();
  T.print();
  double ans = T.gauss();
  T.print();
  cout << ans << endl;
}
```

例题

" **例题** 1：「HEOI2015」 **小 Z 的房间**[2]"

解矩阵树定理的裸题。将每个空房间看作一个结点，根据输入的信息建图，得到 Laplace 矩阵后，任意删掉 L 的第 i 行第 i 列，求这个子式的行列式即可。求行列式的方法就是高斯消元成上三角阵然后算对角线积。另外本题需要在模 k 的整数子环 $\mathbb{Z}_k$ 上进行高斯消元，采用辗转相除法即可。

" **例题** 2：「FJOI2007」 **轮状病毒**[3]"

解本题的解法很多，这里用矩阵树定理是最直接的解法。当输入为 n 时，容易写出其 $n+1$ 阶的 Laplace 矩阵为：

$$L_n = \begin{bmatrix} n & -1 & -1 & -1 & \cdots & -1 & -1 \\ -1 & 3 & -1 & 0 & \cdots & 0 & -1 \\ -1 & -1 & 3 & -1 & \cdots & 0 & 0 \\ -1 & 0 & -1 & 3 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ -1 & 0 & 0 & 0 & \cdots & 3 & -1 \\ -1 & -1 & 0 & 0 & \cdots & -1 & 3 \end{bmatrix}_{n+1}$$

求出它的 n 阶子式的行列式即可，剩下的只有高精度计算了。

" **例题** 2+"

将例题 2 的数据加强，要求 $n \leq 100000$，但是答案对 1000007 取模。（本题求解需要一些线性代数知识）

解推导递推式后利用矩阵快速幂即可求得。

推导递推式的过程：

注意到 L_n 删掉第 1 行第 1 列以后得到的矩阵很有规律，因此其实就是在求矩阵

$$M_n = \begin{bmatrix} 3 & -1 & 0 & \cdots & 0 & -1 \\ -1 & 3 & -1 & \cdots & 0 & 0 \\ 0 & -1 & 3 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 3 & -1 \\ -1 & 0 & 0 & \cdots & -1 & 3 \end{bmatrix}_n$$

的行列式。对 M_n 的行列式按第一列展开，得到

$$\det M_n = 3\det\begin{bmatrix} 3 & -1 & \cdots & 0 & 0 \\ -1 & 3 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 3 & -1 \\ 0 & 0 & \cdots & -1 & 3 \end{bmatrix}_{n-1} + \det\begin{bmatrix} -1 & 0 & \cdots & 0 & -1 \\ -1 & 3 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 3 & -1 \\ 0 & 0 & \cdots & -1 & 3 \end{bmatrix}_{n-1} + (-1)^n\det\begin{bmatrix} -1 & 0 & \cdots & 0 & -1 \\ 3 & -1 & \cdots & 0 & 0 \\ -1 & 3 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 3 & -1 \end{bmatrix}_{n-1}$$

上述三个矩阵的行列式记为 $d_{n-1}, a_{n-1}, b_{n-1}$。

注意到 d_n 是三对角行列式，采用类似的展开的方法可以得到 d_n 具有递推公式 $d_n = 3d_{n-1} - d_{n-2}$。类似地，采用展开的方法可以得到 $a_{n-1} = -d_{n-2} - 1$，以及 $(-1)^n b_{n-1} = -d_{n-2} - 1$。

将这些递推公式代入上式，得到：

$$\det M_n = 3d_{n-1} - 2d_{n-2} - 2$$

$$d_n = 3d_{n-1} - d_{n-2}$$

于是猜测 $\det M_n$ 也是非齐次的二阶线性递推。采用待定系数法可以得到最终的递推公式为

$$\det M_n = 3\det M_{n-1} - \det M_{n-2} + 2$$

改写成 $(\det M_n + 2) = 3(\det M_{n-1} + 2) - (\det M_{n-2} + 2)$ 后，采用矩阵快速幂即可求出答案。

” **例题** 3：「BZOJ3659」WHICH DREAMED IT”

解本题是 BEST 定理的直接应用，但是要注意，由于题目规定「两种完成任务的方式算作不同当且仅当使用钥匙的顺序不同」，对每个欧拉回路，1 号房间可以沿着任意一条出边出发，从而答案还要乘以 1 号房间的出度。

” **例题** 4：「**联合省选** 2020 A」**作业题**[4]”

解首先需要用莫比乌斯反演转化成计算所有生成树的边权和，因为与本文关系不大所以略去。

将行列式的项写成 $w_i x + 1$，最后答案是行列式的一次项系数，因为答案实际上是钦定一条边之后的生成树个数 × 这条边的边权之和，那么被乘上一次项系数的边就是被钦定的边。此时可以把高于一次的项忽略掉，复杂度 $O(n^3)$。

「北京省选集训 2019」生成树计数[5] 是较为一般化的情况：计算生成树权值之和的 k 次方之和，用类似方法构造行列式的项即可，具体见洛谷题解。

” **例题** 5：AGC051D C4[6]”

解无向图欧拉回路计数是 NPC 问题，但这题的图较为简单，确定了 $S-T$ 的边中从 S 指向 T 的有多少条，就可以确定其他三条边的定向方案，然后直接套用 BEST 定理就得到 $O(a+b+c+d)$ 的做法。

注释

根向树形图也被称为内向树形图，但因为计算内向树形图用的是出度，为了不引起 in 和 out 的混淆，所以采用了根向这一说法。

参考资料与注释

[1] 「NOI2021」路径交点

[2] 「HEOI2015」小 Z 的房间

[3] 「FJOI2007」轮状病毒

[4] 「联合省选 2020 A」作业题

[5] 「北京省选集训 2019」生成树计数

[6] AGC051D C4

11.8 有向无环图

定义

边有向，无环。

英文名叫 Directed Acyclic Graph，缩写是 DAG。

性质

- 能 拓扑排序 的图，一定是有向无环图；

  如果有环，那么环上的任意两个节点在任意序列中都不满足条件了。

- 有向无环图，一定能拓扑排序；

  （归纳法）假设节点数不超过 k 的有向无环图都能拓扑排序，那么对于节点数等于 k 的，考虑执行拓扑排序第一步之后的情形即可。

判定

如何判定一个图是否是有向无环图呢？

检验它是否可以进行 拓扑排序 即可。

当然也有另外的方法，可以对图进行一遍 DFS，在得到的 DFS 树上看看有没有连向祖先的非树边（返祖边）。如果有的话，那就有环了。

应用

DP 求最长（短）路

在一般图上，求单源最长（短）路径的最优时间复杂度为 $O(nm)$（Bellman–Ford 算法，适用于有负权图）或 $O(m\log m)$（Dijkstra 算法，适用于无负权图）。

但在 DAG 上，我们可以使用 DP 求最长（短）路，使时间复杂度优化到 $O(n+m)$。状态转移方程为 $dis_v = min(dis_v, dis_u + w_{u,v})$ 或 $dis_v = max(dis_v, dis_u + w_{u,v})$。

拓扑排序后，按照拓扑序遍历每个节点，用当前节点来更新之后的节点。

```
struct edge {
  int v, w;
};

int n, m;
vector<edge> e[MAXN];
```

```cpp
vector<int> L;                            // 存储拓扑排序结果
int max_dis[MAXN], min_dis[MAXN], in[MAXN];  // in 存储每个节点的入度

void toposort() {  // 拓扑排序
  queue<int> S;
  memset(in, 0, sizeof(in));
  for (int i = 1; i <= n; i++) {
    for (int j = 0; j < e[i].size(); j++) {
      in[e[i][j].v]++;
    }
  }
  for (int i = 1; i <= n; i++)
    if (in[i] == 0) S.push(i);
  while (!S.empty()) {
    int u = S.front();
    S.pop();
    L.push_back(u);
    for (int i = 0; i < e[u].size(); i++) {
      if (--in[e[u][i].v] == 0) {
        S.push(e[u][i].v);
      }
    }
  }
}

void dp(int s) {  // 以 s 为起点求单源最长（短）路
  toposort();     // 先进行拓扑排序
  memset(min_dis, 0x3f, sizeof(min_dis));
  memset(max_dis, 0, sizeof(max_dis));
  min_dis[s] = 0;
  for (int i = 0; i < L.size(); i++) {
    int u = L[i];
    for (int j = 0; j < e[u].size(); j++) {
      min_dis[e[u][j].v] = min(min_dis[e[u][j].v], min_dis[u] + e[u][j].w);
      max_dis[e[u][j].v] = max(max_dis[e[u][j].v], max_dis[u] + e[u][j].w);
    }
  }
}
```

参见：DAG 上的 DP。

11.9 拓扑排序

定义

拓扑排序的英文名是 Topological sorting。

拓扑排序要解决的问题是给一个有向无环图的所有节点排序。

我们可以拿大学每学期排课的例子来描述这个过程，比如学习大学课程中有：「程序设计」，「算法语言」，「高等数学」，「离散数学」，「编译技术」，「普通物理」，「数据结构」，「数据库系统」等。按照例子中的排课，当我们想要学习「数据结构」的时候，就必须先学会「离散数学」，学习完这门课后就获得了学习「编译技术」的前置条件。当然，「编译技术」还有一个更加前的课程「算法语言」。这些课程就相当于几个顶点 u，顶点之间的有向边 (u, v) 就相当于学习课程的顺序。教务处安排这些课程，使得在逻辑关系符合的情况下排出课表，就是拓扑排序的过程。

但是如果某一天排课的老师打瞌睡了，说想要学习数据结构，还得先学操作系统，而操作系统的前置课程又是数

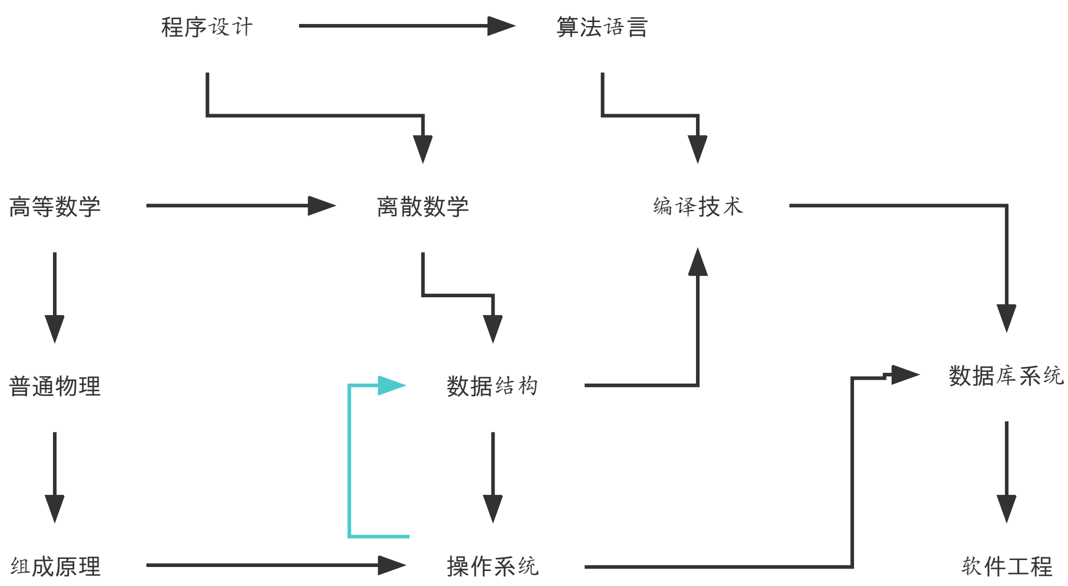


图 11.45 topo

据结构，那么到底应该先学哪一个（不考虑同时学习的情况）？在这里，数据结构和操作系统间就出现了一个环，显然同学们现在没办法弄清楚自己需要先学什么了，也就没办法进行拓扑排序了。因为如果有向图中存在环路，那么我们就没办法进行拓扑排序。

因此我们可以说在一个 DAG（有向无环图）中，我们将图中的顶点以线性方式进行排序，使得对于任何的顶点 u 到 v 的有向边 (u,v)，都可以有 u 在 v 的前面。

还有给定一个 DAG，如果从 i 到 j 有边，则认为 j 依赖于 i。如果 i 到 j 有路径（i 可达 j），则称 j 间接依赖于 i。

拓扑排序的目标是将所有节点排序，使得排在前面的节点不能依赖于排在后面的节点。

AOV 网

日常生活中，一项大的工程可以看作是由若干个子工程组成的集合，这些子工程之间必定存在一定的先后顺序，即某些子工程必须在其他的一些子工程完成后才能开始。

我们用有向图来表现子工程之间的先后关系，子工程之间的先后关系为有向边，这种有向图称为顶点活动网络，即 **AOV 网 (Activity On Vertex Network)**。一个 AOV 网必定是一个有向无环图，即不带有回路。与 DAG 不同的是，AOV 的活动都表示在边上。（上面的例图即为一个 AOV 网）

在 AOV 网中，顶点表示活动，弧表示活动间的优先关系。AOV 网中不应该出现环，这样就能够找到一个顶点序列，使得每个顶点代表的活动的前驱活动都排在该顶点的前面，这样的序列称为拓扑序列（一个 AOV 网的拓扑序列不是唯一的），由 AOV 网构造拓扑序列的过程称为拓扑排序。因此，拓扑排序也可以解释为将 AOV 网中所有活动排成一个序列，使得每个活动的前驱活动都排在该活动的前面（一个 AOV 网中的拓扑排序也不是唯一的）。

- 前驱活动：有向边起点的活动称为终点的前驱活动（只有当一个活动的前驱全部都完成后，这个活动才能进行）。
- 后继活动：有向边终点的活动称为起点的后继活动。

检测 AOV 网中是否带环的方式是构造拓扑序列，看是否包含所有顶点。

构造拓扑序列步骤

1. 从图中选择一个入度为零的点。

2. 输出该顶点，从图中删除此顶点及其所有的出边。

重复上面两步，直到所有顶点都输出，拓扑排序完成，或者图中不存在入度为零的点，此时说明图是有环图，拓扑排序无法完成，陷入死锁。

关键路径和 AOE 网

与 AOV 网对应的是 **AOE 网（Activity On Edge Network)** 即边表示活动的网。AOE 网是一个带权的有向无环图，其中，顶点表示事件，弧表示活动持续的时间。通常，AOE 网可以用来估算工程的完成时间。AOE 网应该是无环的，且存在唯一入度为零的起始顶点（源点），以及唯一出度为零的完成顶点（汇点）。

图 11.46 topo

AOE 网中的有些活动是可以并行进行的，所以完成整个工程的最短时间是从开始点到完成点的最长活动路径长度（这里所说的路径长度是指路径上各活动的持续时间之和，即弧的权值之和，不是路径上弧的数目）。因为一项工程需要完成所有工程内的活动，所以最长的活动路径也是关键路径，它决定工程完成的总时间。

AOE 网的相关基本概念

- 活动：AOE 网中，弧表示活动。弧的权值表示活动持续的时间，活动在事件被触发后开始。
- 事件：AOE 网中，顶点表示事件，事件能被触发。
- 弧（活动）a_j 的最早开始时间：初始点到该弧起点的最长路径长度，记为 $e(j)$。
- 弧（活动）a_j 的最迟开始时间：在不推迟整个工期的前提下，工程达到弧起点所表示的状态最晚能容忍的时间，记为 $l(j)$。
- 顶点（事件）v_j 的最早发生时间：初始点到该顶点的最长路径长度，记为 $ve(j)$，它决定了以该顶点开始的活动的最早发生时间，所以 $ve(j) = e(j)$。
- 顶点（事件）v_j 的最迟发生时间：在不推迟整个工期的前提下，工程达到顶点所表示的状态最晚能容忍的时间，记为 $vl(j)$，它决定了所有以该状态结束的活动的最迟发生时间，所以 $l(j) = vl(j) - dul(aj)$。
- 关键路径：AOE 网中从源点到汇点的最长路径的长度。
- 关键活动：关键路径上的活动，最早开始时间和最迟开始时间相等。

最早和最迟发生时间的递推关系

$$ve(j) = \max\{ve(k) + dut(\langle k, j\rangle)\}, \quad \langle k, j\rangle \in T, 2 \leq j \leq n$$

$$vl(j) = \min\{vl(k) - dut(\langle j,k\rangle)\}, \quad \langle j,k\rangle \in S, 1 \le j \le n-1$$

按拓扑顺序求，最早是从前往后，前驱顶点的最早开始时间与边的权重之和最大者，最迟是从后往前，后继顶点的最迟开始时间与边的权重之差的最小者。

关键路径算法

1. 输入 e 条弧 (j,k)，建立 AOE 网；
2. 从源点 v_0 出发，令 $ve[0] = 0$, 按照拓扑排序求其余各个顶点的最早发生时间 $ve[i]$, $(i \le i \le n-1)$。如果得到的拓扑有序序列中顶点的个数小于网中的顶点数 n，则说明网中存在环，不能求关键路径，算法终止；否则执行步骤 3；
3. 从汇点 v_n 出发，令 $vl[n-1] = ve[n-1]$，按照逆拓扑有序求其余各顶点的最迟发生时间 $vl[i]$, $(n-2 \ge i \ge 2)$;
4. 根据各顶点的 ve 和 vl 值，求每条弧 s 的最早开始时间 $e(s)$ 和最迟开始时间 $l(s)$。若某条弧满足条件 $e(s) = l(s)$，则为关键活动。

Kahn 算法

过程

初始状态下，集合 S 装着所有入度为 0 的点，L 是一个空列表。

每次从 S 中取出一个点 u（可以随便取）放入 L, 然后将 u 的所有边 $(u,v_1),(u,v_2),(u,v_3)\cdots$ 删除。对于边 (u,v)，若将该边删除后点 v 的入度变为 0，则将 v 放入 S 中。

不断重复以上过程，直到集合 S 为空。检查图中是否存在任何边，如果有，那么这个图一定有环路，否则返回 L，L 中顶点的顺序就是构造拓扑序列的结果。

首先看来自 Wikipedia[1] 的伪代码

" 实现"

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edges
while S is not empty do
    remove a node n from S
    insert n into L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
if graph has edges then
    return error (graph has at least one cycle)
else
    return L (a topologically sorted order)
```

代码的核心是维持一个入度为 0 的顶点的集合。

可以参考该图

对其排序的结果就是：2 -> 8 -> 0 -> 3 -> 7 -> 1 -> 5 -> 6 -> 9 -> 4 -> 11 -> 10 -> 12

时间复杂度

假设这个图 $G = (V,E)$ 在初始化入度为 0 的集合 S 的时候就需要遍历整个图，并检查每一条边，因而有 $O(E+V)$ 的复杂度。然后对该集合进行操作，显然也是需要 $O(E+V)$ 的时间复杂度。

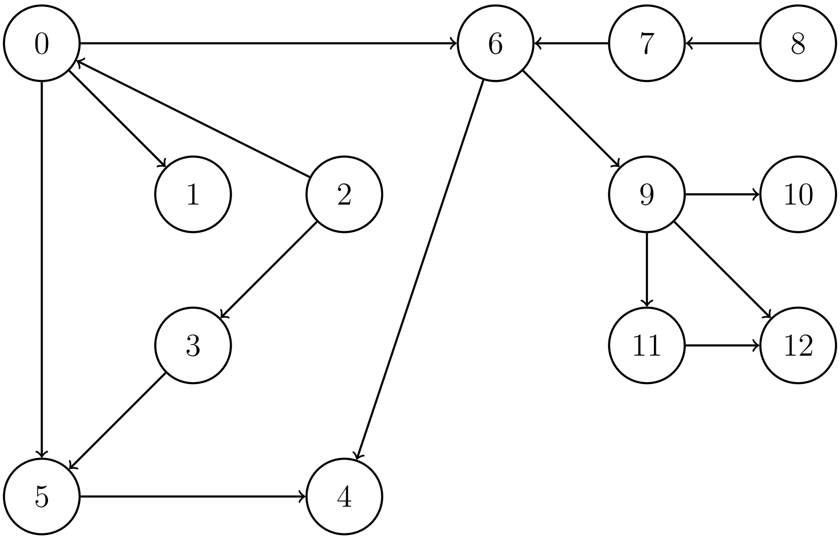


图 11.47 topo

因而总的时间复杂度就有 $O(E+V)$

实现

```
int n, m;
vector<int> G[MAXN];
int in[MAXN];  // 存储每个结点的入度

bool toposort() {
  vector<int> L;
  queue<int> S;
  for (int i = 1; i <= n; i++)
    if (in[i] == 0) S.push(i);
  while (!S.empty()) {
    int u = S.front();
    S.pop();
    L.push_back(u);
    for (auto v : G[u]) {
      if (--in[v] == 0) {
        S.push(v);
      }
    }
  }
  if (L.size() == n) {
    for (auto i : L) cout << i << ' ';
    return true;
  } else {
    return false;
  }
}
```

DFS 算法

实现

```cpp
using Graph = vector<vector<int>>;  // 邻接表

struct TopoSort {
  enum class Status : uint8_t { to_visit, visiting, visited };

  const Graph& graph;
  const int n;
  vector<Status> status;
  vector<int> order;
  vector<int>::reverse_iterator it;

  TopoSort(const Graph& graph)
      : graph(graph),
        n(graph.size()),
        status(n, Status::to_visit),
        order(n),
        it(order.rbegin()) {}

  bool sort() {
    for (int i = 0; i < n; ++i) {
      if (status[i] == Status::to_visit && !dfs(i)) return false;
    }
    return true;
  }

  bool dfs(const int u) {
    status[u] = Status::visiting;
    for (const int v : graph[u]) {
      if (status[v] == Status::visiting) return false;
      if (status[v] == Status::to_visit && !dfs(v)) return false;
    }
    status[u] = Status::visited;
    *it++ = u;
    return true;
  }
};
```

```python
from enum import Enum, auto


class Status(Enum):
    to_visit = auto()
    visiting = auto()
    visited = auto()


def topo_sort(graph: list[list[int]]) -> list[int] | None:
    n = len(graph)
    status = [Status.to_visit] * n
    order = []
```

```
    def dfs(u: int) -> bool:
        status[u] = Status.visiting
        for v in graph[u]:
            if status[v] == Status.visiting:
                return False
            if status[v] == Status.to_visit and not dfs(v):
                return False
        status[u] = Status.visited
        order.append(u)
        return True

    for i in range(n):
        if status[i] == Status.to_visit and not dfs(i):
            return None

    return order[::-1]
```

时间复杂度：$O(E+V)$ 空间复杂度：$O(V)$

合理性证明

考虑一个图，删掉某个入度为 0 的节点之后，如果新图可以拓扑排序，那么原图一定也可以。反过来，如果原图可以拓扑排序，那么删掉后也可以。

应用

拓扑排序可以判断图中是否有环，还可以用来判断图是否是一条链。拓扑排序可以用来求 AOE 网中的关键路径，估算工程完成的最短时间。

求字典序最大/最小的拓扑排序

将 Kahn 算法中的队列替换成最大堆/最小堆实现的优先队列即可，此时总的时间复杂度为 $O(E+V\log V)$。

习题

CF 1385E[2]：需要通过拓扑排序构造。

Luogu P1347[3]: 拓扑排序模板。

参考

1. 离散数学及其应用。ISBN:9787111555391
2. https://blog.csdn.net/dm_vincent/article/details/7714519[4]
3. Topological sorting,https://en.wikipedia.org/w/index.php?title=Topological_sorting&oldid=854351542[5]
4. https://blog.csdn.net/time_money/article/details/109857779[6]
5. https://zhuanlan.zhihu.com/p/164751109[7]

参考资料与注释

[1] Wikipedia

[2] CF 1385E

[3] Luogu P1347

[4] https://blog.csdn.net/dm_vincent/article/details/7714519

[5] https://en.wikipedia.org/w/index.php?title=Topological_sorting&oldid=854351542

[6] https://blog.csdn.net/time_money/article/details/109857779

[7] https://zhuanlan.zhihu.com/p/164751109

11.10 最小生成树

Authors: Chrogeek, Enter-tainer, HeRaNO, Ir1d, Marcythm, ShadowsEpic, StudyingFather, Xeonacid, bear-good, billchenchina, diauweb, diauweb, greyqz, kawa-yoiko, ouuan, partychicken, sshwy, stevebraveman, zhouyuyang2002, renbaoshuo, Hszzzx, y-kx-b

定义

在阅读下列内容之前，请务必阅读 图论相关概念 与 树基础 部分，并了解以下定义：

1. 生成子图
2. 生成树

我们定义无向连通图的**最小生成树**（Minimum Spanning Tree，MST）为边权和最小的生成树。

注意：只有连通图才有生成树，而对于非连通图，只存在生成森林。

Kruskal 算法

Kruskal 算法是一种常见并且好写的最小生成树算法，由 Kruskal 发明。该算法的基本思想是从小到大加入边，是个贪心算法。

前置知识

并查集、贪心、图的存储。

实现

图示：

伪代码：

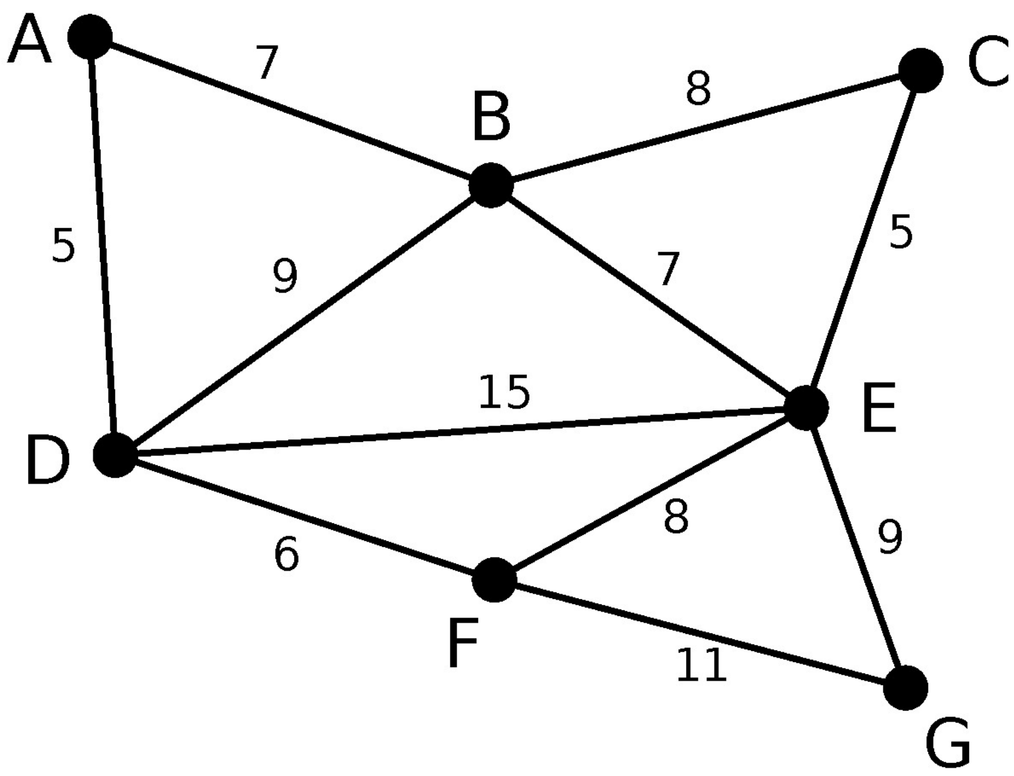


图 11.48

Input. The edges of the graph e, where each element in e is (u, v, w) denoting that there is an edge between u and v weighted w.
Output. The edges of the MST of the input graph.
Method.
$result \leftarrow \emptyset$
sort e into nondecreasing order by weight w
for each (u, v, w) in the sorted e
    if u and v are not connected in the union-find set
        connect u and v in the union-find set
        $result \leftarrow result \bigcup \{(u, v, w)\}$
return *result*

算法虽简单，但需要相应的数据结构来支持……具体来说，维护一个森林，查询两个结点是否在同一棵树中，连接两棵树。

抽象一点地说，维护一堆**集合**，查询两个元素是否属于同一集合，合并两个集合。

其中，查询两点是否连通和连接两点可以使用并查集维护。

如果使用 $O(m \log m)$ 的排序算法，并且使用 $O(m\alpha(m, n))$ 或 $O(m \log n)$ 的并查集，就可以得到时间复杂度为 $O(m \log m)$ 的 Kruskal 算法。

证明

思路很简单，为了造出一棵最小生成树，我们从最小边权的边开始，按边权从小到大依次加入，如果某次加边产生了环，就扔掉这条边，直到加入了 $n-1$ 条边，即形成了一棵树。

证明：使用归纳法，证明任何时候 K 算法选择的边集都被某棵 MST 所包含。

基础：对于算法刚开始时，显然成立（最小生成树存在）。

归纳：假设某时刻成立，当前边集为 F，令 T 为这棵 MST，考虑下一条加入的边 e。

如果 e 属于 T，那么成立。

否则，$T+e$ 一定存在一个环，考虑这个环上不属于 F 的另一条边 f（一定只有一条）。

首先，f 的权值一定不会比 e 小，不然 f 会在 e 之前被选取。

然后，f 的权值一定不会比 e 大，不然 $T+e-f$ 就是一棵比 T 还优的生成树了。

所以，$T+e-f$ 包含了 F，并且也是一棵最小生成树，归纳成立。

Prim 算法

Prim 算法是另一种常见并且好写的最小生成树算法。该算法的基本思想是从一个结点开始，不断加点（而不是 Kruskal 算法的加边）。

实现

图示：

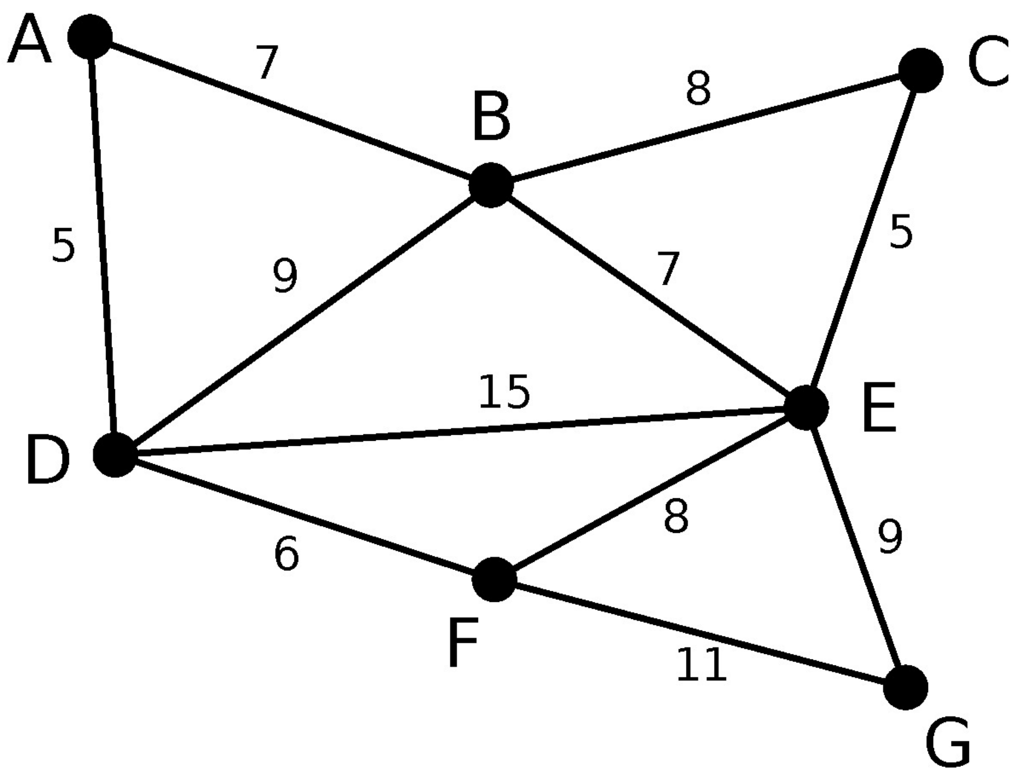


图 11.49

具体来说，每次要选择距离最小的一个结点，以及用新的边更新其他结点的距离。

其实跟 Dijkstra 算法一样，每次找到距离最小的一个点，可以暴力找也可以用堆维护。

堆优化的方式类似 Dijkstra 的堆优化，但如果使用二叉堆等不支持 $O(1)$ decrease-key 的堆，复杂度就不优于 Kruskal，常数也比 Kruskal 大。所以，一般情况下都使用 Kruskal 算法，在稠密图尤其是完全图上，暴力 Prim 的复杂度比 Kruskal 优，但**不一定**实际跑得更快。

暴力：$O(n^2+m)$。

二叉堆：$O((n+m)\log n)$。

Fib 堆：$O(n\log n+m)$。

伪代码：

Input. The nodes of the graph V ; the function $g(u, v)$ which means the weight of the edge (u, v); the function $adj(v)$ which means the nodes adjacent to v.
Output. The sum of weights of the MST of the input graph.
Method.
$result \leftarrow 0$
choose an arbitrary node in V to be the *root*
$dis(root) \leftarrow 0$
for each node $v \in (V - \{root\})$
  $dis(v) \leftarrow \infty$
$rest \leftarrow V$
while $rest \neq \emptyset$
  $cur \leftarrow$ the node with the minimum dis in $rest$
  $result \leftarrow result + dis(cur)$
  $rest \leftarrow rest - \{cur\}$
  for each node $v \in adj(cur)$
    $dis(v) \leftarrow \min(dis(v), g(cur, v))$
return *result*

注意：上述代码只是求出了最小生成树的权值，如果要输出方案还需要记录每个点的 dis 代表的是哪条边。

" 代码实现"

```cpp
// 使用二叉堆优化的 Prim 算法。
#include <cstring>
#include <iostream>
#include <queue>
using namespace std;
const int N = 5050, M = 2e5 + 10;

struct E {
  int v, w, x;
} e[M * 2];

int n, m, h[N], cnte;

void adde(int u, int v, int w) { e[++cnte] = E{v, w, h[u]}, h[u] = cnte; }

struct S {
  int u, d;
};

bool operator<(const S &x, const S &y) { return x.d > y.d; }

priority_queue<S> q;
int dis[N];
bool vis[N];

int res = 0, cnt = 0;
```

```cpp
void Prim() {
  memset(dis, 0x3f, sizeof(dis));
  dis[1] = 0;
  q.push({1, 0});
  while (!q.empty()) {
    if (cnt >= n) break;
    int u = q.top().u, d = q.top().d;
    q.pop();
    if (vis[u]) continue;
    vis[u] = 1;
    ++cnt;
    res += d;
    for (int i = h[u]; i; i = e[i].x) {
      int v = e[i].v, w = e[i].w;
      if (w < dis[v]) {
        dis[v] = w, q.push({v, w});
      }
    }
  }
}

int main() {
  cin >> n >> m;
  for (int i = 1, u, v, w; i <= m; ++i) {
    cin >> u >> v >> w, adde(u, v, w), adde(v, u, w);
  }
  Prim();
  if (cnt == n)
    cout << res;
  else
    cout << "No MST.";
  return 0;
}
```

证明

从任意一个结点开始，将结点分成两类：已加入的，未加入的。

每次从未加入的结点中，找一个与已加入的结点之间边权最小值最小的结点。

然后将这个结点加入，并连上那条边权最小的边。

重复 $n-1$ 次即可。

证明：还是说明在每一步，都存在一棵最小生成树包含已选边集。

基础：只有一个结点的时候，显然成立。

归纳：如果某一步成立，当前边集为 F，属于 T 这棵 MST，接下来要加入边 e。

如果 e 属于 T，那么成立。

否则考虑 $T+e$ 中环上另一条可以加入当前边集的边 f。

首先，f 的权值一定不小于 e 的权值，否则就会选择 f 而不是 e 了。

然后，f 的权值一定不大于 e 的权值，否则 $T+e-f$ 就是一棵更小的生成树了。

因此，e 和 f 的权值相等，$T+e-f$ 也是一棵最小生成树，且包含了 F。

Boruvka 算法

接下来介绍另一种求解最小生成树的算法——Boruvka 算法。该算法的思想是前两种算法的结合。它可以用于求解**边权互不相同**的无向图的最小生成森林。（无向连通图就是最小生成树。）

为了描述该算法，我们需要引入一些定义：

1. 定义 E' 为我们当前找到的最小生成森林的边。在算法执行过程中，我们逐步向 E' 加边，定义**连通块**表示一个点集 $V' \subseteq V$，且这个点集中的任意两个点 u，v 在 E' 中的边构成的子图上是连通的（互相可达）。
2. 定义一个连通块的**最小边**为它连向其它连通块的边中权值最小的那一条。

初始时，$E' = \emptyset$，每个点各自是一个连通块：

1. 计算每个点分别属于哪个连通块。将每个连通块都设为「没有最小边」。
2. 遍历每条边 (u, v)，如果 u 和 v 不在同一个连通块，就用这条边的边权分别更新 u 和 v 所在连通块的最小边。
3. 如果所有连通块都没有最小边，退出程序，此时的 E' 就是原图最小生成森林的边集。否则，将每个有最小边的连通块的最小边加入 E'，返回第一步。

下面通过一张动态图来举一个例子（图源自 维基百科[1]）：

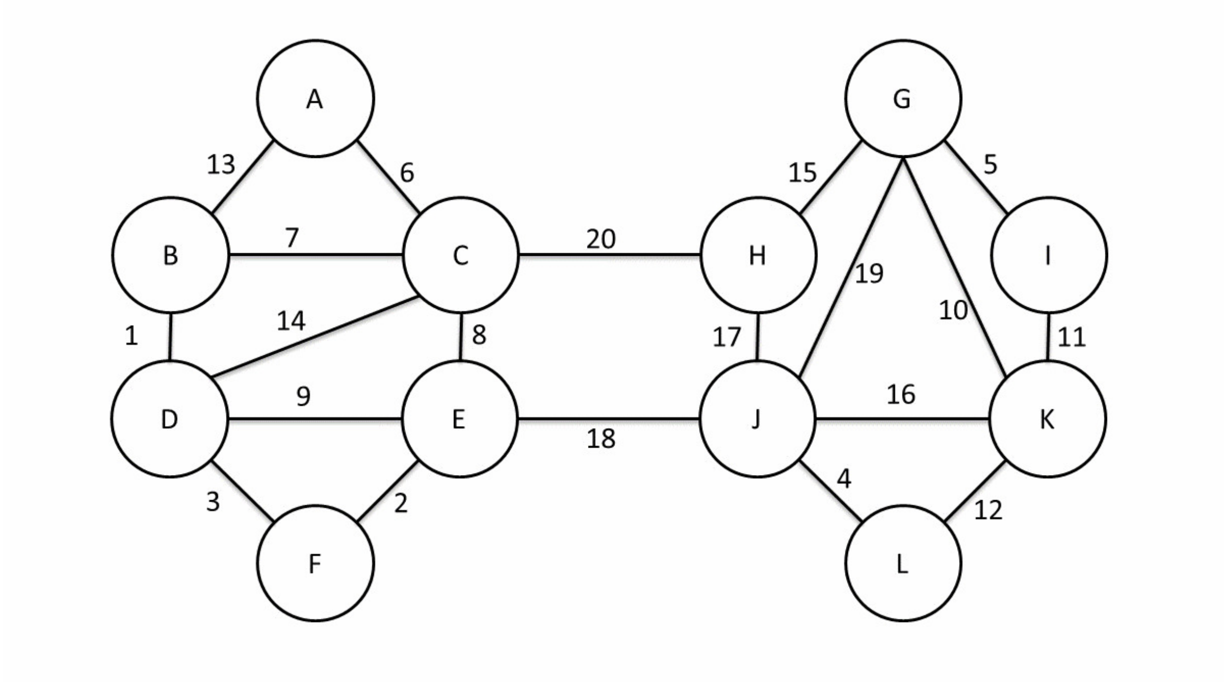


图 11.50 eg

当原图连通时，每次迭代连通块数量至少减半，算法只会迭代不超过 $O(\log V)$ 次，而原图不连通时相当于多个

子问题，因此算法复杂度是 $O(E\log V)$ 的。给出算法的伪代码：（修改自 维基百科[1]）

```
Input. A graph G whose edges have distinct weights.
Output. The minimum spanning forest of G.
Method.
Initialize a forest F to be a set of one-vertex trees
while True
    Find the components of F and label each vertex of G by its component
    Initialize the cheapest edge for each component to "None"
    for each edge (u, v) of G
        if u and v have different component labels
            if (u, v) is cheaper than the cheapest edge for the component of u
                Set (u, v) as the cheapest edge for the component of u
            if (u, v) is cheaper than the cheapest edge for the component of v
                Set (u, v) as the cheapest edge for the component of v
    if all components'cheapest edges are "None"
        return F
    for each component whose cheapest edge is not "None"
        Add its cheapest edge to F
```

习题

- 「HAOI2006」 聪明的猴子[2]
- 「SCOI2005」 繁忙的都市[3]

最小生成树的唯一性

考虑最小生成树的唯一性。如果一条边**不在最小生成树的边集中**，并且可以替换与其**权值相同、并且在最小生成树边集**的另一条边。那么，这个最小生成树就是不唯一的。

对于 Kruskal 算法，只要计算为当前权值的边可以放几条，实际放了几条，如果这两个值不一样，那么就说明这几条边与之前的边产生了一个环（这个环中至少有两条当前权值的边，否则根据并查集，这条边是不能放的），即最小生成树不唯一。

寻找权值与当前边相同的边，我们只需要记录头尾指针，用单调队列即可在 $O(\alpha(m))$（m 为边数）的时间复杂度里优秀解决这个问题（基本与原算法时间相同）。

" **例题**：POJ 1679[4]"

```cpp
#include <algorithm>
#include <cstdio>

struct Edge {
  int x, y, z;
};

int f[100001];
Edge a[100001];

int cmp(const Edge& a, const Edge& b) { return a.z < b.z; }

int find(int x) { return f[x] == x ? x : f[x] = find(f[x]); }
```

```
int main() {
  int t;
  scanf("%d", &t);
  while (t--) {
    int n, m;
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i++) f[i] = i;
    for (int i = 1; i <= m; i++) scanf("%d%d%d", &a[i].x, &a[i].y, &a[i].z);
    std::sort(a + 1, a + m + 1, cmp);  // 先排序
    int num = 0, ans = 0, tail = 0, sum1 = 0, sum2 = 0;
    bool flag = 1;
    for (int i = 1; i <= m + 1; i++) {  // 再并查集加边
      if (i > tail) {
        if (sum1 != sum2) {
          flag = 0;
          break;
        }
        sum1 = 0;
        for (int j = i; j <= m + 1; j++) {
          if (a[j].z != a[i].z) {
            tail = j - 1;
            break;
          }
          if (find(a[j].x) != find(a[j].y)) ++sum1;
        }
        sum2 = 0;
      }
      if (i > m) break;
      int x = find(a[i].x);
      int y = find(a[i].y);
      if (x != y && num != n - 1) {
        sum2++;
        num++;
        f[x] = f[y];
        ans += a[i].z;
      }
    }
    if (flag)
      printf("%d\n", ans);
    else
      printf("Not Unique!\n");
  }
  return 0;
}
```

次小生成树

非严格次小生成树

定义

在无向图中，边权和最小的满足边权和**大于等于**最小生成树边权和的生成树

求解方法

- 求出无向图的最小生成树 T，设其权值和为 M

- 遍历每条未被选中的边 $e = (u, v, w)$，找到 T 中 u 到 v 路径上边权最大的一条边 $e' = (s, t, w')$，则在 T 中以 e 替换 e'，可得一棵权值和为 $M' = M + w - w'$ 的生成树 T'.
- 对所有替换得到的答案 M' 取最小值即可

如何求 u, v 路径上的边权最大值呢?

我们可以使用倍增来维护，预处理出每个节点的 2^i 级祖先及到达其 2^i 级祖先路径上最大的边权，这样在倍增求 LCA 的过程中可以直接求得。

严格次小生成树

定义

在无向图中，边权和最小的满足边权和**严格大于**最小生成树边权和的生成树

求解方法

考虑刚才的非严格次小生成树求解过程，为什么求得的解是非严格的?

因为最小生成树保证生成树中 u 到 v 路径上的边权最大值一定**不大于**其他从 u 到 v 路径的边权最大值。换言之，当我们用于替换的边的权值与原生成树中被替换边的权值相等时，得到的次小生成树是非严格的。

解决的办法很自然：我们维护到 2^i 级祖先路径上的最大边权的同时维护**严格次大边权**，当用于替换的边的权值与原生成树中路径最大边权相等时，我们用严格次大值来替换即可。

这个过程可以用倍增求解，复杂度 $O(m \log m)$。

" 代码实现"

```
#include <algorithm>
#include <iostream>

const int INF = 0x3fffffff;
const long long INF64 = 0x3fffffffffffffffLL;

struct Edge {
  int u, v, val;

  bool operator<(const Edge &other) const { return val < other.val; }
};

Edge e[300010];
bool used[300010];

int n, m;
long long sum;

class Tr {
 private:
  struct Edge {
    int to, nxt, val;
  } e[600010];

  int cnt, head[100010];

  int pnt[100010][22];
  int dpth[100010];
```

```
  // 到祖先的路径上边权最大的边
  int maxx[100010][22];
  // 到祖先的路径上边权次大的边，若不存在则为 -INF
  int minn[100010][22];

 public:
  void addedge(int u, int v, int val) {
    e[++cnt] = (Edge){v, head[u], val};
    head[u] = cnt;
  }

  void insedge(int u, int v, int val) {
    addedge(u, v, val);
    addedge(v, u, val);
  }

  void dfs(int now, int fa) {
    dpth[now] = dpth[fa] + 1;
    pnt[now][0] = fa;
    minn[now][0] = -INF;
    for (int i = 1; (1 << i) <= dpth[now]; i++) {
      pnt[now][i] = pnt[pnt[now][i - 1]][i - 1];
      int kk[4] = {maxx[now][i - 1], maxx[pnt[now][i - 1]][i - 1],
                   minn[now][i - 1], minn[pnt[now][i - 1]][i - 1]};
      // 从四个值中取得最大值
      std::sort(kk, kk + 4);
      maxx[now][i] = kk[3];
      // 取得严格次大值
      int ptr = 2;
      while (ptr >= 0 && kk[ptr] == kk[3]) ptr--;
      minn[now][i] = (ptr == -1 ? -INF : kk[ptr]);
    }

    for (int i = head[now]; i; i = e[i].nxt) {
      if (e[i].to != fa) {
        maxx[e[i].to][0] = e[i].val;
        dfs(e[i].to, now);
      }
    }
  }

  int lca(int a, int b) {
    if (dpth[a] < dpth[b]) std::swap(a, b);

    for (int i = 21; i >= 0; i--)
      if (dpth[pnt[a][i]] >= dpth[b]) a = pnt[a][i];

    if (a == b) return a;

    for (int i = 21; i >= 0; i--) {
      if (pnt[a][i] != pnt[b][i]) {
        a = pnt[a][i];
        b = pnt[b][i];
      }
```

```
    }
    return pnt[a][0];
  }

  int query(int a, int b, int val) {
    int res = -INF;
    for (int i = 21; i >= 0; i--) {
      if (dpth[pnt[a][i]] >= dpth[b]) {
        if (val != maxx[a][i])
          res = std::max(res, maxx[a][i]);
        else
          res = std::max(res, minn[a][i]);
        a = pnt[a][i];
      }
    }
    return res;
  }
} tr;

int fa[100010];

int find(int x) { return fa[x] == x ? x : fa[x] = find(fa[x]); }

void Kruskal() {
  int tot = 0;
  std::sort(e + 1, e + m + 1);
  for (int i = 1; i <= n; i++) fa[i] = i;

  for (int i = 1; i <= m; i++) {
    int a = find(e[i].u);
    int b = find(e[i].v);
    if (a != b) {
      fa[a] = b;
      tot++;
      tr.insedge(e[i].u, e[i].v, e[i].val);
      sum += e[i].val;
      used[i] = 1;
    }
    if (tot == n - 1) break;
  }
}

int main() {
  std::ios::sync_with_stdio(0);
  std::cin.tie(0);
  std::cout.tie(0);

  std::cin >> n >> m;
  for (int i = 1; i <= m; i++) {
    int u, v, val;
    std::cin >> u >> v >> val;
    e[i] = (Edge){u, v, val};
  }
```

```
  Kruskal();
  long long ans = INF64;
  tr.dfs(1, 0);

  for (int i = 1; i <= m; i++) {
    if (!used[i]) {
      int _lca = tr.lca(e[i].u, e[i].v);
      // 找到路径上不等于 e[i].val 的最大边权
      long long tmpa = tr.query(e[i].u, _lca, e[i].val);
      long long tmpb = tr.query(e[i].v, _lca, e[i].val);
      // 这样的边可能不存在，只在这样的边存在时更新答案
      if (std::max(tmpa, tmpb) > -INF)
        ans = std::min(ans, sum - std::max(tmpa, tmpb) + e[i].val);
    }
  }
  // 次小生成树不存在时输出 -1
  std::cout << (ans == INF64 ? -1 : ans) << '\n';
  return 0;
}
```

瓶颈生成树

定义

无向图 G 的瓶颈生成树是这样的一个生成树，它的最大的边权值在 G 的所有生成树中最小。

性质

最小生成树是瓶颈生成树的充分不必要条件。即最小生成树一定是瓶颈生成树，而瓶颈生成树不一定是最小生成树。

关于最小生成树一定是瓶颈生成树这一命题，可以运用反证法证明：我们设最小生成树中的最大边权为 w，如果最小生成树不是瓶颈生成树的话，则瓶颈生成树的所有边权都小于 w，我们只需删去原最小生成树中的最长边，用瓶颈生成树中的一条边来连接删去边后形成的两棵树，得到的新生成树一定比原最小生成树的权值和还要小，这样就产生了矛盾。

例题

"POJ 2395 Out of Hay"

给出 n 个农场和 m 条边，农场按 1 到 n 编号，现在有一人要从编号为 1 的农场出发到其他的农场去，求在这途中他最多需要携带的水的重量，注意他每到达一个农场，可以对水进行补给，且要使总共的路径长度最小。题目要求的就是瓶颈树的最大边，可以通过求最小生成树来解决。

最小瓶颈路

定义

无向图 G 中 x 到 y 的最小瓶颈路是这样的一类简单路径，满足这条路径上的最大的边权在所有 x 到 y 的简单路径中是最小的。

性质

根据最小生成树定义，x 到 y 的最小瓶颈路上的最大边权等于最小生成树上 x 到 y 路径上的最大边权。虽然最小生成树不唯一，但是每种最小生成树 x 到 y 路径的最大边权相同且为最小值。也就是说，每种最小生成树上的 x 到 y 的路径均为最小瓶颈路。

但是，并不是所有最小瓶颈路都存在一棵最小生成树满足其为树上 x 到 y 的简单路径。

例如下图：

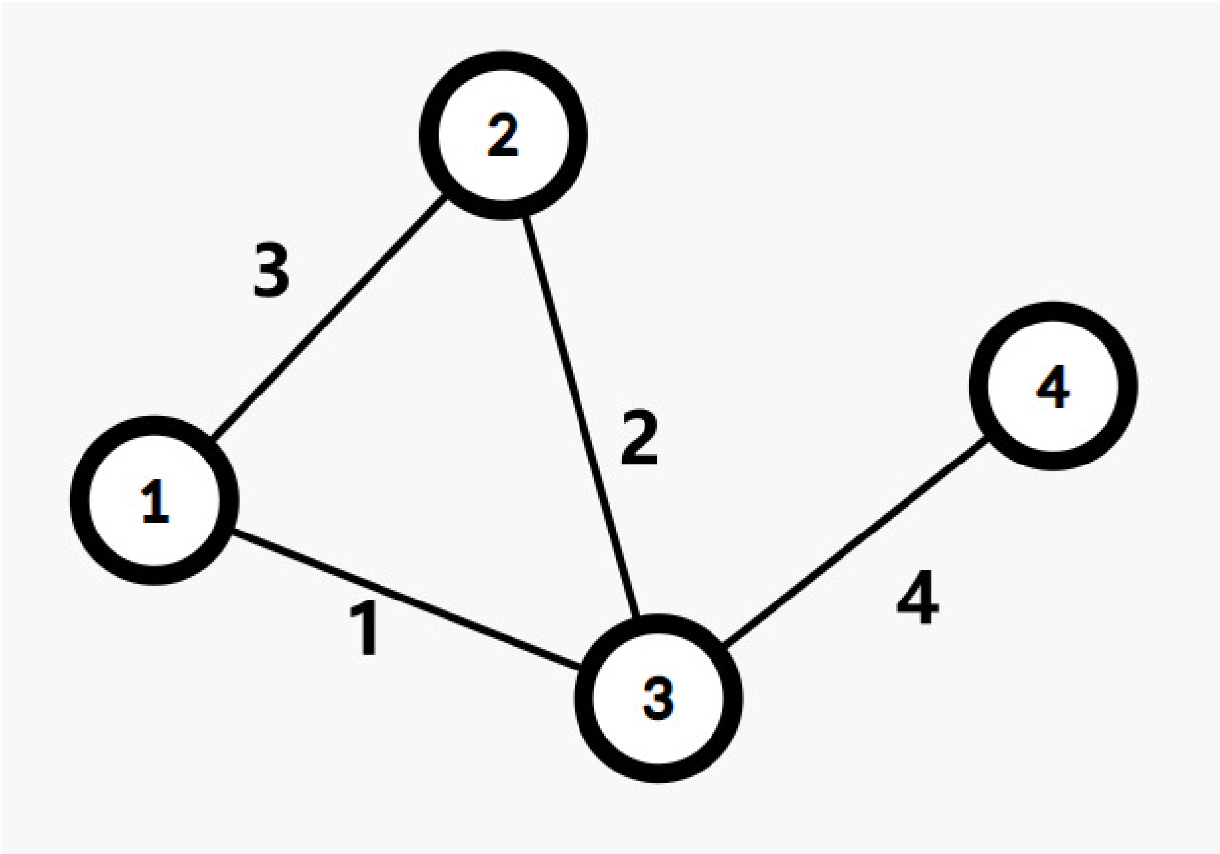


图 11.51

1 到 4 的最小瓶颈路显然有以下两条：1-2-3-4。1-3-4。

但是，1-2 不会出现在任意一种最小生成树上。

应用

由于最小瓶颈路不唯一，一般情况下会询问最小瓶颈路上的最大边权。

也就是说，我们需要求最小生成树链上的 max。

倍增、树剖都可以解决，这里不再展开。

Kruskal 重构树

定义

在跑 Kruskal 的过程中我们会从小到大加入若干条边。现在我们仍然按照这个顺序。

首先新建 n 个集合，每个集合恰有一个节点，点权为 0。

每一次加边会合并两个集合，我们可以新建一个点，点权为加入边的边权，同时将两个集合的根节点分别设为新建点的左儿子和右儿子。然后我们将两个集合和新建点合并成一个集合。将新建点设为根。

不难发现，在进行 $n-1$ 轮之后我们得到了一棵恰有 n 个叶子的二叉树，同时每个非叶子节点恰好有两个儿子。这棵树就叫 Kruskal 重构树。

举个例子：

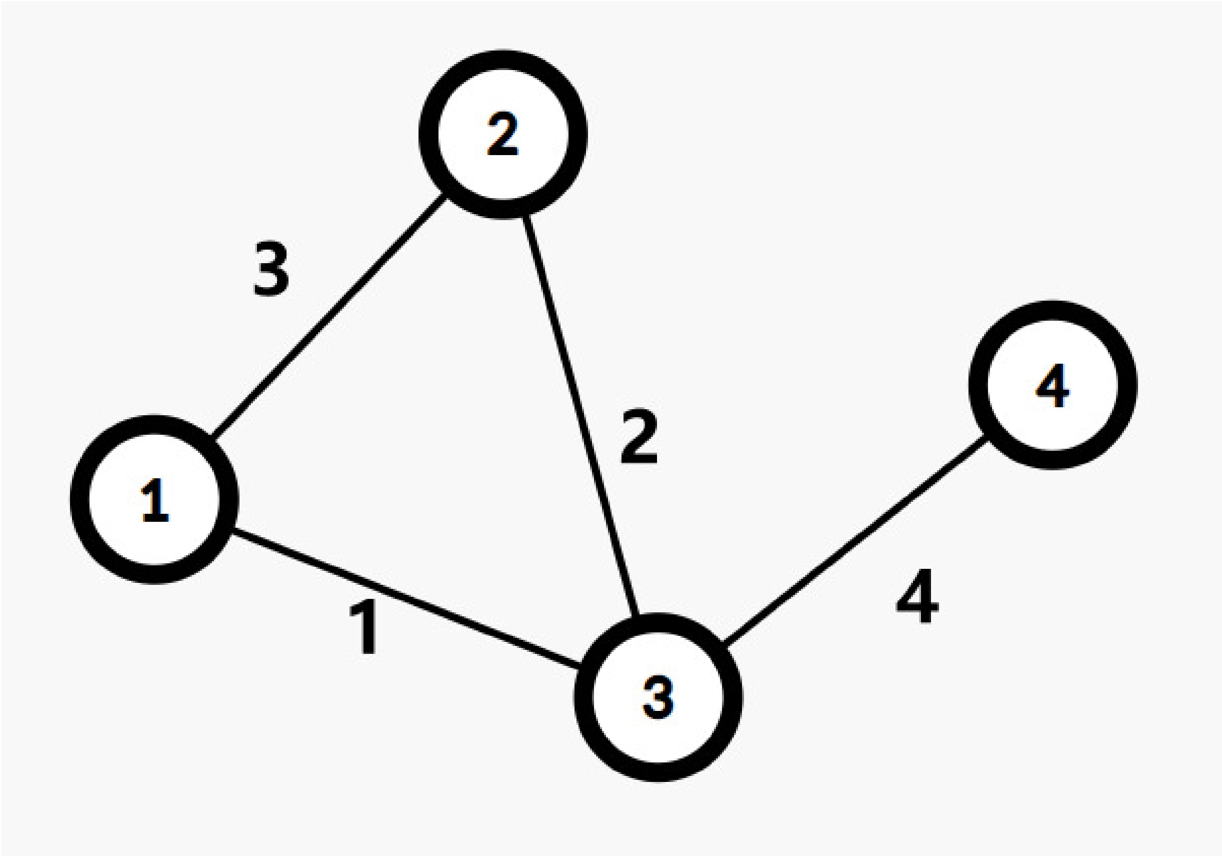


图 11.52

这张图的 Kruskal 重构树如下：

性质

不难发现，原图中两个点之间的所有简单路径上最大边权的最小值 = 最小生成树上两个点之间的简单路径上的最大值 = Kruskal 重构树上两点之间的 LCA 的权值。

也就是说，到点 x 的简单路径上最大边权的最小值 $\leq val$ 的所有点 y 均在 Kruskal 重构树上的某一棵子树内，且恰好为该子树的所有叶子节点。

我们在 Kruskal 重构树上找到 x 到根的路径上权值 $\leq val$ 的最浅的节点。显然这就是所有满足条件的节点所在的子树的根节点。

如果需要求原图中两个点之间的所有简单路径上最小边权的最大值，则在跑 Kruskal 的过程中按边权大到小的顺序加边。

"「LOJ 137」 最小瓶颈路　加强版[5]"

```
#include <bits/stdc++.h>

using namespace std;

const int MAX_VAL_RANGE = 280010;
```

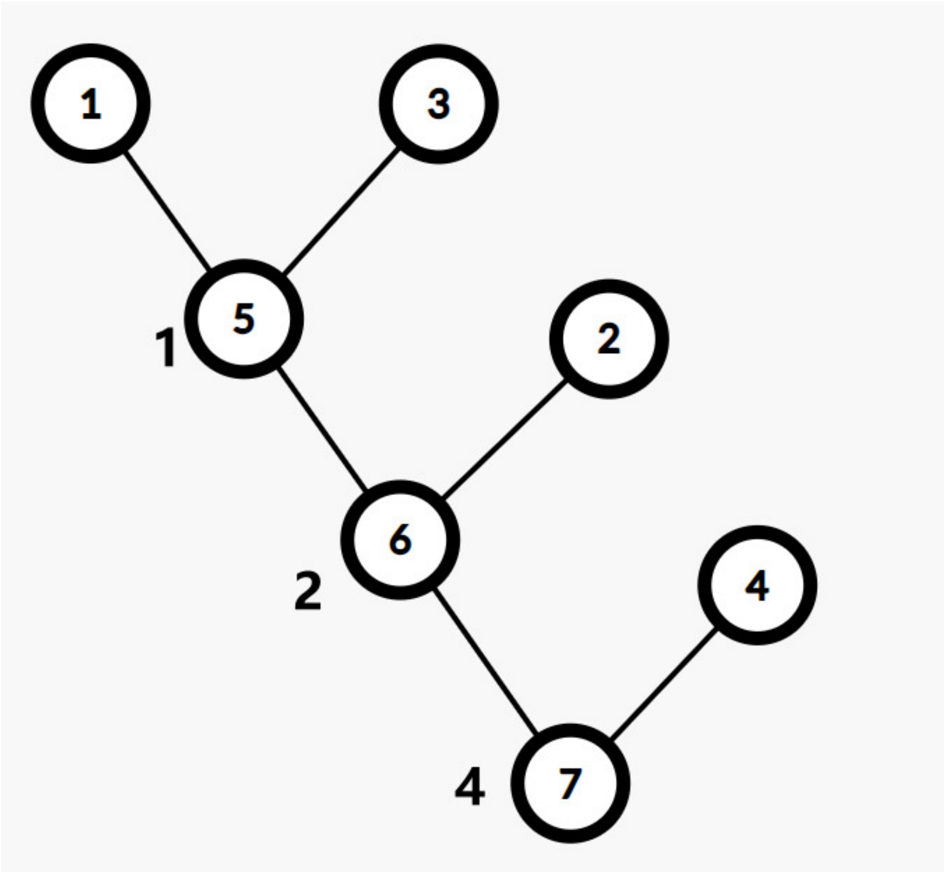


图 11.53

```
int n, m, log2Values[MAX_VAL_RANGE + 1];

namespace TR {
struct Edge {
  int to, nxt, val;
} e[400010];

int cnt, head[140010];

void addedge(int u, int v, int val = 0) {
  e[++cnt] = (Edge){v, head[u], val};
  head[u] = cnt;
}

int val[140010];

namespace LCA {
int sec[280010], cnt;
int pos[140010];
int dpth[140010];

void dfs(int now, int fa) {
  dpth[now] = dpth[fa] + 1;
  sec[++cnt] = now;
  pos[now] = cnt;
```

```
  for (int i = head[now]; i; i = e[i].nxt) {
    if (fa != e[i].to) {
      dfs(e[i].to, now);
      sec[++cnt] = now;
    }
  }
}

int dp[280010][20];

void init() {
  dfs(2 * n - 1, 0);
  for (int i = 1; i <= 4 * n; i++) {
    dp[i][0] = sec[i];
  }
  for (int j = 1; j <= 19; j++) {
    for (int i = 1; i + (1 << j) - 1 <= 4 * n; i++) {
      dp[i][j] = dpth[dp[i][j - 1]] < dpth[dp[i + (1 << (j - 1))][j - 1]]
                     ? dp[i][j - 1]
                     : dp[i + (1 << (j - 1))][j - 1];
    }
  }
}

int lca(int x, int y) {
  int l = pos[x], r = pos[y];
  if (l > r) {
    swap(l, r);
  }
  int k = log2Values[r - l + 1];
  return dpth[dp[l][k]] < dpth[dp[r - (1 << k) + 1][k]]
             ? dp[l][k]
             : dp[r - (1 << k) + 1][k];
}
}  // namespace LCA
}  // namespace TR

using TR::addedge;

namespace GR {
struct Edge {
  int u, v, val;

  bool operator<(const Edge &other) const { return val < other.val; }
} e[100010];

int fa[140010];

int find(int x) { return fa[x] == 0 ? x : fa[x] = find(fa[x]); }

void kruskal() {  // 最小生成树
  int tot = 0, cnt = n;
  sort(e + 1, e + m + 1);
  for (int i = 1; i <= m; i++) {
```

```
    int fau = find(e[i].u), fav = find(e[i].v);
    if (fau != fav) {
      cnt++;
      fa[fau] = fa[fav] = cnt;
      addedge(fau, cnt);
      addedge(cnt, fau);
      addedge(fav, cnt);
      addedge(cnt, fav);
      TR::val[cnt] = e[i].val;
      tot++;
    }
    if (tot == n - 1) {
      break;
    }
  }
}
}  // namespace GR

int ans;
int A, B, C, P;

int rnd() { return A = (A * B + C) % P; }

void initLog2() {
  for (int i = 2; i <= MAX_VAL_RANGE; i++) {
    log2Values[i] = log2Values[i >> 1] + 1;
  }
}

int main() {
  initLog2();  // 预处理
  cin >> n >> m;
  for (int i = 1; i <= m; i++) {
    int u, v, val;
    cin >> u >> v >> val;
    GR::e[i] = (GR::Edge){u, v, val};
  }
  GR::kruskal();
  TR::LCA::init();
  int Q;
  cin >> Q;
  cin >> A >> B >> C >> P;

  while (Q--) {
    int u = rnd() % n + 1, v = rnd() % n + 1;
    ans += TR::val[TR::LCA::lca(u, v)];
    ans %= 1000000007;
  }
  cout << ans;
  return 0;
}
```

"NOI 2018 归程[6]"

首先预处理出来每一个点到根节点的最短路。

我们构造出来根据海拔的最大生成树。显然每次询问可以到达的节点是在最大生成树中和询问点的路径上最小边权 $> p$ 的节点。

根据 Kruskal 重构树的性质，这些节点满足均在一棵子树内同时为其所有叶子节点。

也就是说，我们只需要求出 Kruskal 重构树上每一棵子树叶子的权值 min 就可以支持子树询问。

询问的根节点可以使用 Kruskal 重构树上倍增的方式求出。

时间复杂度 $O((n+m+Q)\log n)$。

参考资料与注释

[1] 维基百科 [1-1] [1-2]

[2] 「HAOI2006」聪明的猴子

[3] 「SCOI2005」繁忙的都市

[4] POJ 1679

[5] 「LOJ 137」最小瓶颈路加强版

[6] NOI 2018 归程

11.11 斯坦纳树

斯坦纳树问题是组合优化问题，与最小生成树相似，是最短网络的一种。最小生成树是在给定的点集和边中寻求最短网络使所有点连通。而最小斯坦纳树允许在给定点外增加额外的点，使生成的最短网络开销最小。

问题引入

19 世纪初叶，柏林大学几何方面的著名学者斯坦纳，研究了一个非常简单却很有启示性的问题：将三个村庄用总长为极小的道路连接起来。从数学上说，就是在平面内给定三个点 A、B、C 找出平面内第四个点 P，使得和数 $a+b+c$ 为最短，这里 a、b、c 分别表示从 P 到 A、B、C 的距离。

问题的答案是：如果三角形 ABC 的每个内角都小于 120°，那么 P 就是使边 AB、BC、AC 对该点所张的角都是 120° 的点。如果三角形 ABC 的有一个角，例如 C 角，大于或等于 120°，那么点 P 与顶点 C 重合。

问题推广

1. 在斯坦纳问题中，给定了三个固定点 A, B, C。很自然地可以把这个问题推广到给定 n 个点 $A_1, A_2, \dots, A_n$ 的情形；我们要求出平面内的点 P，使距离和 $a_1+a_2+\cdots+a_n$ 为极小，其中 a_i 是距离 PA_i。
2. 考虑到点的其他相关因素，加入了权重的表示。n 个点的其他相关因素可以换算成一个权重表示，求出平面内的点 P，使距离与权重的乘积的总和 $a_1\cdot w_1+a_2\cdot w_2+\cdots+a_n\cdot w_n$ 为极小，其中 w_i 是每个点的权重。
3. 库朗（R.Courant）和罗宾斯（H.Robbins）提出第一个定义的推广是肤浅的。为了求得斯坦纳问题真正有价值的推广，必须放弃寻找一个单独的点 P，而代之以具有最短总长的 " 道路网 "。数学上表述成：给定 n 个点 $A_1, A_2, \cdots, A_n$，试求连接此 n 个点，总长最短的直线段连接系统，并且任意两点都可由系统中的直线段组成的折线连接起来。他们将此新问题称为**斯坦纳树问题**。在给定 n 个点的情形，最多将有 $n-2$ 个复接点（斯坦纳

点）。过每一斯坦纳点，至多有三条边通过。若为三条边，则它们两两交成 120° 角；若为两条边，则此斯坦纳点必为某一已给定的点，且此两条边交成的角必大于或等于 120°。

连接三个以上的点的最短网络

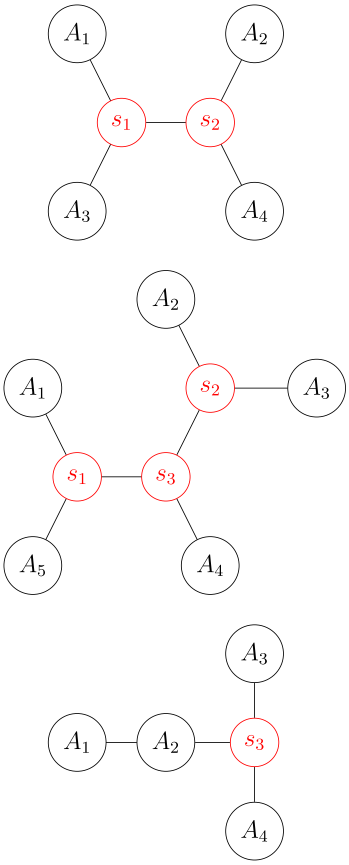


图 11.54 steiner-tree1

在第一种情形，解是由五条线段组成的，其中有两个斯坦纳点（红色 s_1, s_2），在那里有三条线段相交且相互间的交角为 120°。第二种情形的解含有三个斯坦纳点。第三种情形，一个或几个斯坦纳点可能退化，或被一个或几个给定的点所代替。

我们将斯坦纳树的问题模型以图论形式呈现。

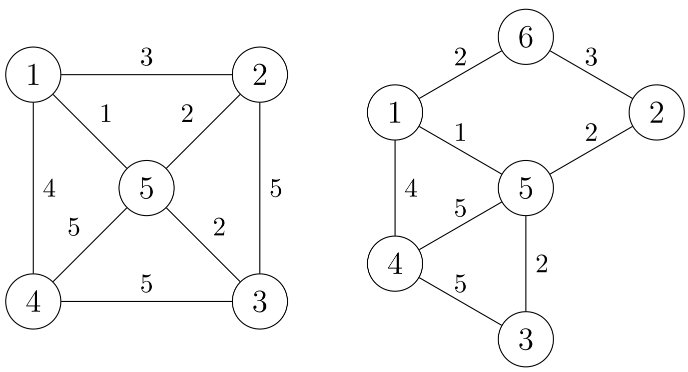


图 11.55 steiner-tree2

对于形式一，如果令关键点为 $\{1, 2, 3, 4\}$，可以发现若直接将这四个关键点相连的最小边权和是 12，显然这不是最优的。如果考虑使用 5 号节点那么最小边权和就会是 9，得到一个更优的答案。

对于形式二，如果令关键点为 $\{1, 2, 3, 4\}$，可以发现这四个关键点中的一些点甚至没有直接相连的边，必须考虑使用复接点（斯坦纳点）。这时将 5 号考虑进去可以得到最小边权和 9。

并且我们可以发现在两张图中 1 号和 4 号的斯坦纳点是退化的，被 1 号或 4 号代替了。

例题

首先以一道模板题来带大家熟悉最小斯坦纳树问题。见 【模板】 最小斯坦纳树[1]。

题意已经很明确了，给定连通图 G 中的 n 个点与 k 个关键点，连接 k 个关键点，使得生成树的所有边的权值和最小。

结合上面的知识我们可以知道直接连接这 k 个关键点生成的权值和不一定是最小的，或者这 k 个关键点不会直接（相邻）连接。所以应当使用剩下的 $n-k$ 个点。

我们使用状态压缩动态规划来求解。用 $f(i,S)$ 表示以 i 为根的一棵树，包含集合 S 中所有点的最小边权值和。

考虑状态转移：

- 首先对连通的子集进行转移，$f(i,S) \leftarrow \min(f(i,S), f(i,T)+f(i,S-T))$。
- 在当前的子集连通状态下进行边的松弛操作，$f(i,S) \leftarrow \min(f(i,S), f(j,S)+w(j,i))$。在下面的代码中用一个 `tree[tot]` 来记录两个相连节点 i,j 的相关信息。

" 参考实现"

```cpp
#include <bits/stdc++.h>

using namespace std;

const int maxn = 510;
const int INF = 0x3f3f3f3f;
typedef pair<int, int> P;
int n, m, k;

struct edge {
  int to, next, w;
} e[maxn << 1];

int head[maxn << 1], tree[maxn << 1], tot;
int dp[maxn][5000], vis[maxn];
int key[maxn];
priority_queue<P, vector<P>, greater<P> > q;

void add(int u, int v, int w) {
  e[++tot] = edge{v, head[u], w};
  head[u] = tot;
}

void dijkstra(int s) {  // 求解最短路
  memset(vis, 0, sizeof(vis));
  while (!q.empty()) {
    P item = q.top();
    q.pop();
    if (vis[item.second]) continue;
    vis[item.second] = 1;
    for (int i = head[item.second]; i; i = e[i].next) {
      if (dp[tree[i]][s] > dp[item.second][s] + e[i].w) {
        dp[tree[i]][s] = dp[item.second][s] + e[i].w;
        q.push(P(dp[tree[i]][s], tree[i]));
```

```
      }
    }
  }
}

int main() {
  memset(dp, INF, sizeof(dp));
  scanf("%d %d %d", &n, &m, &k);
  int u, v, w;
  for (int i = 1; i <= m; i++) {
    scanf("%d %d %d", &u, &v, &w);
    add(u, v, w);
    tree[tot] = v;
    add(v, u, w);
    tree[tot] = u;
  }
  for (int i = 1; i <= k; i++) {
    scanf("%d", &key[i]);
    dp[key[i]][1 << (i - 1)] = 0;
  }
  for (int s = 1; s < (1 << k); s++) {
    for (int i = 1; i <= n; i++) {
      for (int subs = s & (s - 1); subs;
           subs = s & (subs - 1))  // 状压 dp 可以看下题解里写的比较详细
        dp[i][s] = min(dp[i][s], dp[i][subs] + dp[i][s ^ subs]);
      if (dp[i][s] != INF) q.push(P(dp[i][s], i));
    }
    dijkstra(s);
  }
  printf("%d\n", dp[key[1]][(1 << k) - 1]);
  return 0;
}
```

另外一道经典例题 [WC2008] 游览计划[2]。

这道题是求点权和最小的斯坦纳树，用 $f(i,S)$ 表示以 i 为根的一棵树，包含集合 S 中所有点的最小点权值和。a_i 表示点权。

考虑状态转移：

- $f(i,S) \leftarrow \min(f(i,S), f(i,T)+f(i,S-T)-a_i)$。由于此处合并时同一个点 a_i，会被加两次，所以减去。
- $f(i,S) \leftarrow \min(f(i,S), f(j,S)+w(j,i))$。

可以发现状态转移与上面的模板题是类似的，麻烦的是对答案的输出，在 DP 的过程中还要记录路径。

用 `pre[i][s]` 记录转移到 i 为根，连通状态集合为 s 时的点与集合的信息。在 DP 结束后从 `pre[root][S]` 出发，寻找与集合里的点相连的那些点并逐步分解集合 S，用 ans 数组来记录被使用的那些点，当集合分解完毕时搜索也就结束了。

" 参考实现"

```
#include <bits/stdc++.h>

using namespace std;

#define mp make_pair
typedef pair<int, int> P;
```

```
typedef pair<P, int> PP;
const int INF = 0x3f3f3f3f;
const int dx[] = {0, 0, -1, 1};
const int dy[] = {1, -1, 0, 0};
int n, m, K, root;
int f[101][1111], a[101], ans[11][11];
bool inq[101];
PP pre[101][1111];
queue<P> q;

bool legal(P u) {
  if (u.first >= 0 && u.second >= 0 && u.first < n && u.second < m) {
    return true;
  }
  return false;
}

int num(P u) { return u.first * m + u.second; }

void spfa(int s) {
  memset(inq, 0, sizeof(inq));
  while (!q.empty()) {
    P u = q.front();
    q.pop();
    inq[num(u)] = 0;
    for (int d = 0; d < 4; d++) {
      P v = mp(u.first + dx[d], u.second + dy[d]);
      int du = num(u), dv = num(v);
      if (legal(v) && f[dv][s] > f[du][s] + a[dv]) {
        f[dv][s] = f[du][s] + a[dv];
        if (!inq[dv]) {
          inq[dv] = 1;
          q.push(v);
        }
        pre[dv][s] = mp(u, s);
      }
    }
  }
}

void dfs(P u, int s) {
  if (!pre[num(u)][s].second) return;
  ans[u.first][u.second] = 1;
  int nu = num(u);
  if (pre[nu][s].first == u)
    dfs(u, s ^ pre[nu][s].second);  // 通过 dfs 来找到答案
  dfs(pre[nu][s].first, pre[nu][s].second);
}

int main() {
  memset(f, INF, sizeof(f));
  scanf("%d %d", &n, &m);
  int tot = 0;
  for (int i = 0; i < n; i++) {
```

```
    for (int j = 0; j < m; j++) {
      scanf("%d", &a[tot]);
      if (!a[tot]) {
        f[tot][1 << (K++)] = 0;
        root = tot;
      }
      tot++;
    }
  }
  for (int s = 1; s < (1 << K); s++) {
    for (int i = 0; i < n * m; i++) {
      for (int subs = s & (s - 1); subs; subs = s & (subs - 1)) {
        if (f[i][s] > f[i][subs] + f[i][s ^ subs] - a[i]) {
          f[i][s] = f[i][subs] + f[i][s ^ subs] - a[i];  // 状态转移
          pre[i][s] = mp(mp(i / m, i % m), subs);
        }
      }
      if (f[i][s] < INF) q.push(mp(i / m, i % m));
    }
    spfa(s);
  }
  printf("%d\n", f[root][(1 << K) - 1]);
  dfs(mp(root / m, root % m), (1 << K) - 1);
  for (int i = 0, tot = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      if (!a[tot++])
        putchar('x');
      else
        putchar(ans[i][j] ? 'o' : '_');
    }
    if (i != n - 1) printf("\n");
  }
  return 0;
}
```

习题

- 【模板】 最小斯坦纳树[1]
- [WC2008] 游览计划[2]
- [JLOI2015] 管道连接[3]
- [APIO2013] 机器人[4]

参考资料与注释

[1] 【模板】最小斯坦纳树 [1-1] [1-2]

[2] [WC2008] 游览计划 [2-1] [2-2]

[3] [JLOI2015] 管道连接

[4] [APIO2013] 机器人

11.12 最小树形图

定义

有向图上的最小生成树（Directed Minimum Spanning Tree）称为最小树形图。

常用的算法是朱刘算法（也称 Edmonds 算法），可以在 $O(nm)$ 时间内解决最小树形图问题。

过程

1. 对于每个点，选择指向它的边权最小的那条边。
2. 如果没有环，算法终止；否则进行缩环并更新其他点到环的距离。

实现

```
bool solve() {
  ans = 0;
  int u, v, root = 0;
  for (;;) {
    f(i, 0, n) in[i] = 1e100;
    f(i, 0, m) {
      u = e[i].s;
      v = e[i].t;
      if (u != v && e[i].w < in[v]) {
        in[v] = e[i].w;
        pre[v] = u;
      }
    }
    f(i, 0, m) if (i != root && in[i] > 1e50) return 0;
    int tn = 0;
    memset(id, -1, sizeof id);
    memset(vis, -1, sizeof vis);
    in[root] = 0;
    f(i, 0, n) {
      ans += in[i];
      v = i;
      while (vis[v] != i && id[v] == -1 && v != root) {
        vis[v] = i;
        v = pre[v];
      }
      if (v != root && id[v] == -1) {
        for (int u = pre[v]; u != v; u = pre[u]) id[u] = tn;
        id[v] = tn++;
      }
    }
    if (tn == 0) break;
    f(i, 0, n) if (id[i] == -1) id[i] = tn++;
    f(i, 0, m) {
      u = e[i].s;
      v = e[i].t;
      e[i].s = id[u];
      e[i].t = id[v];
```

```
      if (e[i].s != e[i].t) e[i].w -= in[v];
    }
    n = tn;
    root = id[root];
  }
  return ans;
}
```

Tarjan 的 DMST 算法

Tarjan 提出了一种能够在 $O(m + n\log n)$ 时间内解决最小树形图问题的算法。

这里的算法描述以及参考代码基于 Uri Zwick 教授的课堂讲义，更多的细节可以参考原文。

过程

Tarjan 的算法分为**收缩**与**伸展**两个过程。接下来先介绍**收缩**的过程。

我们需要假设输入的图是满足强连通的，如果不满足那么就加入 $O(n)$ 条边使其满足，并且这些边的边权是无穷大的。

我们需要一个堆存储结点的入边编号，入边权值，结点总代价等相关信息，由于后续过程中会有堆的合并操作，这里采用 左偏树 与 并查集 实现。算法的每一步都选择一个任意结点 v，需要保证 v 不是根节点，并且在堆中没有它的入边。再将 v 的最小入边加入到堆中，如果新加入的这条边使堆中的边形成了环，那么将构成环的那些结点收缩，我们不妨将这些已经收缩的结点命名为**超级结点**，再继续这个过程，如果所有的顶点都缩成了一个超级结点，那么收缩过程就结束了。整个收缩过程结束后会得到一棵收缩树，之后将对它进行伸展操作。

堆中的边总是会形成一条路径 $v_0 \leftarrow v_1 \leftarrow \cdots \leftarrow v_k$，由于图是强连通的，这个路径必然存在，并且其中的 v_i 可能是最初的单一结点，也可能是压缩后的超级结点。

最初有 $v_o = a$，其中 a 是图中任意的一个结点，每一次选择一条最小入边 $v_k \leftarrow u$，如果 u 不是 $v_0, v_1, \dots, v_k$ 中的一个结点，那么就将结点扩展到 $v_{k+1} = u$。如果 u 是他们其中的一个结点 v_i，那么就找到了一个关于 $v_i \leftarrow \cdots \leftarrow v_k \leftarrow v_i$ 的环，再将他们收缩为一个超级结点 c。

向队列 P 中放入所有的结点或超级结点，并初始选择任意一节点 a，只要队列不为空，就进行以下步骤：

1. 选择 a 的最小入边，保证不存在自环，并找到另一头的结点 b。如果结点 b 没有被记录过说明未形成环，令 $a \leftarrow b$，继续当前操作寻找环。
2. 如果 b 被记录过了，就说明出现了环。总结点数加一，并将环上的所有结点重新编号，对堆进行合并，以及结点/超级结点的总权值的更新。更新权值操作就是将环上所有结点的入边都收集起来，并减去环上入边的边权。

以图片为例，左边的强连通图在收缩后就形成了右边的一棵收缩树，其中 a 是结点 1 与结点 2 收缩后的超级结点，b 是结点 3，结点 4，结点 5 收缩后的超级结点，A 是两个超级结点 a 与 b 收缩后形成的。

伸展过程是相对简单的，以原先要求的根节点 r 为起始点，对 r 到收缩树的根上的每一个环进行伸展。再以 r 的祖先结点 f_r 为起始点，将其到根的环展开，直到遍历完所有的结点。

实现

```
#include <bits/stdc++.h>

using namespace std;

typedef long long ll;
#define maxn 102
#define INF 0x3f3f3f3f
```

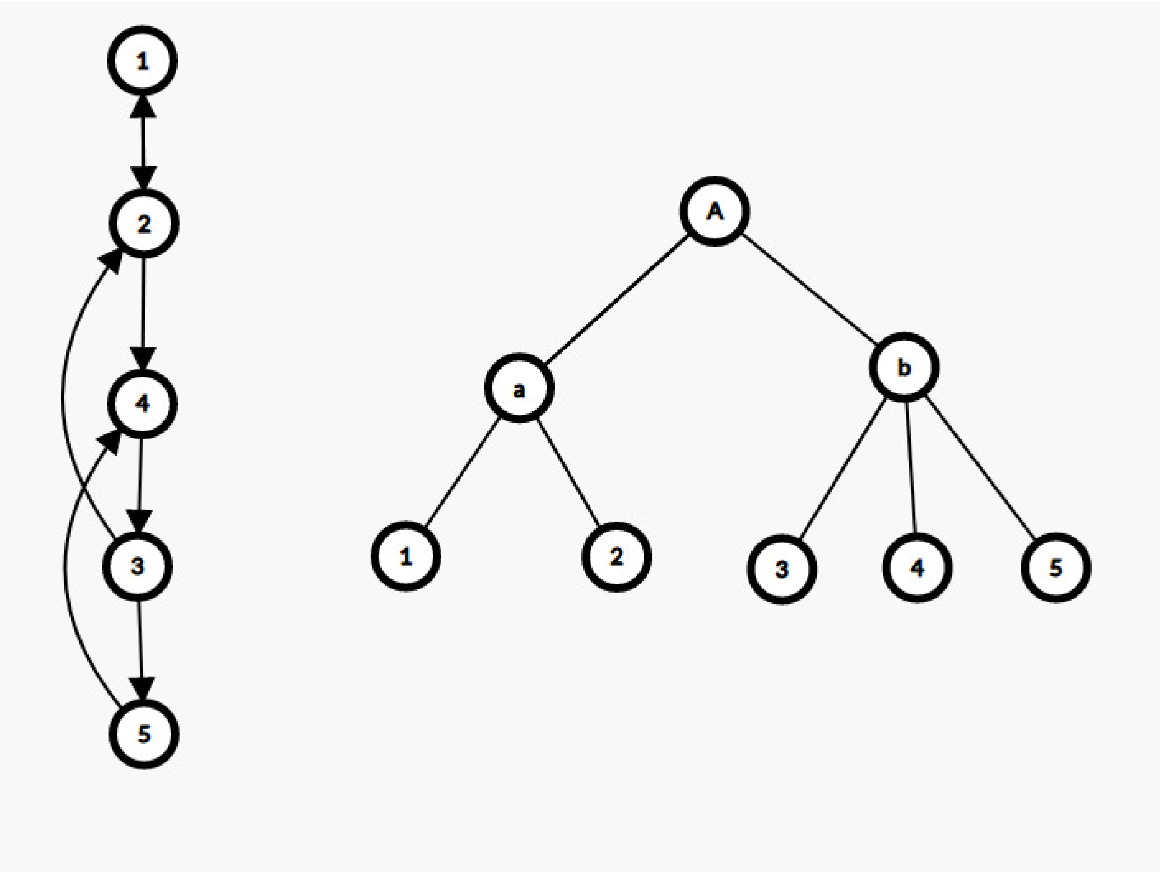


图 11.56 dmst1

```
struct UnionFind {
  int fa[maxn << 1];

  UnionFind() { memset(fa, 0, sizeof(fa)); }

  void clear(int n) { memset(fa + 1, 0, sizeof(int) * n); }

  int find(int x) { return fa[x] ? fa[x] = find(fa[x]) : x; }

  int operator[](int x) { return find(x); }
};

struct Edge {
  int u, v, w, w0;
};

struct Heap {
  Edge *e;
  int rk, constant;
  Heap *lch, *rch;

  Heap(Edge *_e) : e(_e), rk(1), constant(0), lch(NULL), rch(NULL) {}

  void push() {
    if (lch) lch->constant += constant;
    if (rch) rch->constant += constant;
```

```
    e->w += constant;
    constant = 0;
  }
};

Heap *merge(Heap *x, Heap *y) {
  if (!x) return y;
  if (!y) return x;
  if (x->e->w + x->constant > y->e->w + y->constant) swap(x, y);
  x->push();
  x->rch = merge(x->rch, y);
  if (!x->lch || x->lch->rk < x->rch->rk) swap(x->lch, x->rch);
  if (x->rch)
    x->rk = x->rch->rk + 1;
  else
    x->rk = 1;
  return x;
}

Edge *extract(Heap *&x) {
  Edge *r = x->e;
  x->push();
  x = merge(x->lch, x->rch);
  return r;
}

vector<Edge> in[maxn];
int n, m, fa[maxn << 1], nxt[maxn << 1];
Edge *ed[maxn << 1];
Heap *Q[maxn << 1];
UnionFind id;

void contract() {
  bool mark[maxn << 1];
  // 将图上的每一个结点与其相连的那些结点进行记录。
  for (int i = 1; i <= n; i++) {
    queue<Heap *> q;
    for (int j = 0; j < in[i].size(); j++) q.push(new Heap(&in[i][j]));
    while (q.size() > 1) {
      Heap *u = q.front();
      q.pop();
      Heap *v = q.front();
      q.pop();
      q.push(merge(u, v));
    }
    Q[i] = q.front();
  }
  mark[1] = true;
  for (int a = 1, b = 1, p; Q[a]; b = a, mark[b] = true) {
    // 寻找最小入边以及其端点，保证无环。
    do {
      ed[a] = extract(Q[a]);
      a = id[ed[a]->u];
    } while (a == b && Q[a]);
```

```
    if (a == b) break;
    if (!mark[a]) continue;
    // 对发现的环进行收缩，以及环内的结点重新编号，总权值更新。
    for (a = b, n++; a != n; a = p) {
      id.fa[a] = fa[a] = n;
      if (Q[a]) Q[a]->constant -= ed[a]->w;
      Q[n] = merge(Q[n], Q[a]);
      p = id[ed[a]->u];
      nxt[p == n ? b : p] = a;
    }
  }
}

ll expand(int x, int r);

ll expand_iter(int x) {
  ll r = 0;
  for (int u = nxt[x]; u != x; u = nxt[u]) {
    if (ed[u]->w0 >= INF)
      return INF;
    else
      r += expand(ed[u]->v, u) + ed[u]->w0;
  }
  return r;
}

ll expand(int x, int t) {
  ll r = 0;
  for (; x != t; x = fa[x]) {
    r += expand_iter(x);
    if (r >= INF) return INF;
  }
  return r;
}

void link(int u, int v, int w) { in[v].push_back({u, v, w, w}); }

int main() {
  int rt;
  scanf("%d %d %d", &n, &m, &rt);
  for (int i = 0; i < m; i++) {
    int u, v, w;
    scanf("%d %d %d", &u, &v, &w);
    link(u, v, w);
  }
  // 保证强连通
  for (int i = 1; i <= n; i++) link(i > 1 ? i - 1 : n, i, INF);
  contract();
  ll ans = expand(rt, n);
  if (ans >= INF)
    puts("-1");
  else
    printf("%lld\n", ans);
  return 0;
}
```

参考文献

Uri Zwick. (2013),Directed Minimum Spanning Trees[1], Lecture notes on ”Analysis of Algorithms”

https://riteme.site/blog/2018-6-18/mdst.html#_3[2]

参考资料与注释

[1] Directed Minimum Spanning Trees

[2] https://riteme.site/blog/2018-6-18/mdst.html#_3

11.13 最小直径生成树

在学习最小直径生成树（Minimum Diameter Spanning Tree）前建议先阅读 树的直径 的内容。

定义

在无向图的所有生成树中，直径最小的那一棵生成树就是最小直径生成树。

图的绝对中心

求解直径最小生成树，首先需要找到**图的绝对中心**，**图的绝对中心**可以存在于一条边上或某个结点上，该中心到所有点的最短距离的最大值最小。

根据**图的绝对中心**的定义可以知道，到绝对中心距离最远的结点至少有两个。

令 $d(i,j)$ 为顶点 i,j 间的最短路径长，通过多源最短路算法求出所有结点的最短路。

$rk(i,j)$ 记录点 i 到其他所有结点中第 j 小的那个结点。

图的绝对中心可能在某条边上，枚举每一条边 $w=(u,v)$，并且假设图的绝对中心 c 就在这条边上。那么距离 u 的长度为 x（$x \leq w$），距离 v 的长度就是 $w-x$。

对于图中的任意一点 i，图的绝对中心 c 到 i 的距离为 $d(c,i)=\min(d(u,i)+x, d(v,i)+(w-x))$。

举例一个结点 i，该结点与图的绝对中心的位置关系如下图。

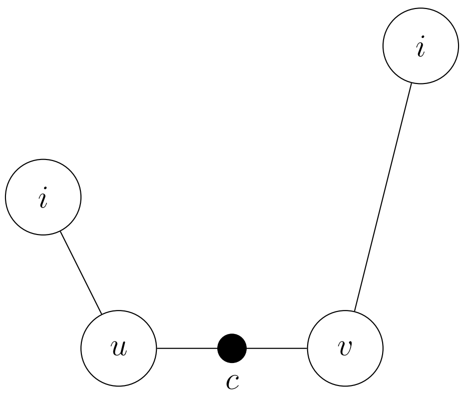


图 11.57 mdst1

随着图的绝对中心 c 在边上的改变会生成一个距离与 c 位置的函数图像。显然的，当前的 $d(c,i)$ 的函数图像是一个两条斜率相同的线段构成的折线段。

对于图上的任意一结点，图的绝对中心到最远距离结点的函数就写作 $f=\max\{d(c,i)\}, i\in[1,n]$，其函数图像如下。

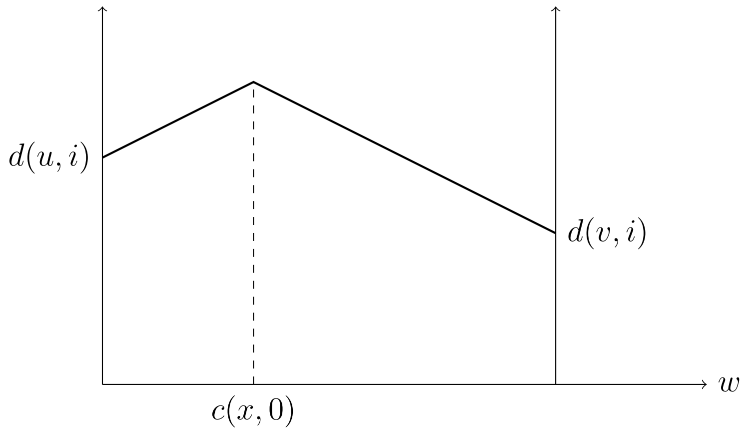


图 11.58 mdst2

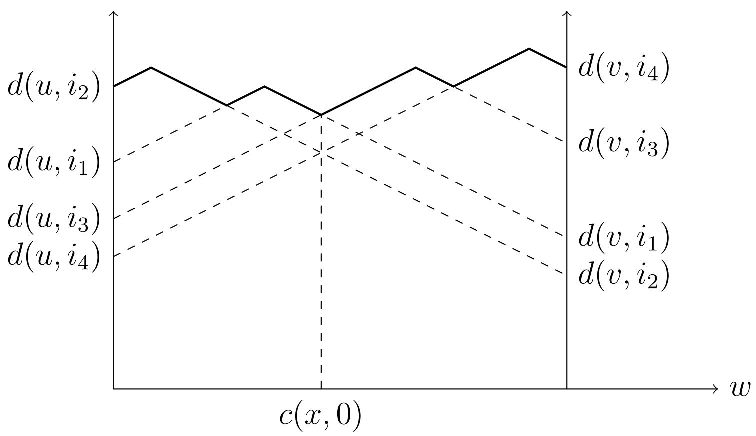


图 11.59 mdst3

并且这些折线交点中的最低点，横坐标就是图的绝对中心的位置。

图的绝对中心可能在某个结点上，用距离预选结点最远的那个结点来更新，即 $ans \leftarrow \min(ans, d(i, rk(i,n)) \times 2)$。

过程

1. 使用多源最短路算法（Floyd，Johnson 等），求出 d 数组；
2. 求出 $rk(i,j)$，并将其升序排序；
3. 图的绝对中心可能在某个结点上，用距离预选结点最远的那个结点来更新，遍历所有结点并用 $ans \leftarrow \min(ans, d(i, rk(i,n)) \times 2)$ 更新最小值。
4. 图的绝对中心可能在某条边上，枚举所有的边。对于一条边 $w(u,v)$ 从距离 u 最远的结点开始更新。当出现 $d(v, rk(u,i)) > \max_{j=i+1}^{n} d(v, rk(u,j))$ 的情况时，用 $ans \leftarrow \min(ans, d(u, rk(u,i)) + \max_{j=i+1}^{n} d(v, rk(u,j)) + w(u,v))$ 来更新。因为这种情况会使图的绝对中心改变。

" 实现"

```cpp
bool cmp(int a, int b) { return val[a] < val[b]; }

void Floyd() {
  for (int k = 1; k <= n; k++)
    for (int i = 1; i <= n; i++)
      for (int j = 1; j <= n; j++) d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}

void solve() {
  Floyd();
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      rk[i][j] = j;
      val[j] = d[i][j];
    }
    sort(rk[i] + 1, rk[i] + 1 + n, cmp);
  }
  int ans = INF;
  // 图的绝对中心可能在结点上
  for (int i = 1; i <= n; i++) ans = min(ans, d[i][rk[i][n]] * 2);
  // 图的绝对中心可能在边上
  for (int i = 1; i <= m; i++) {
    int u = a[i].u, v = a[i].v, w = a[i].w;
    for (int p = n, i = n - 1; i >= 1; i--) {
      if (d[v][rk[u][i]] > d[v][rk[u][p]]) {
        ans = min(ans, d[u][rk[u][i]] + d[v][rk[u][p]] + w);
        p = i;
      }
    }
  }
}
```

例题

- CodeForce 266D BerDonalds[1]

最小直径生成树

根据图的绝对中心的定义，容易得知图的绝对中心是最小直径生成树的直径的中点。

求解最小直径生成树首先需要找到图的绝对中心。以图的绝对中心为起点，生成一个最短路径树，那么就可以得到最小直径生成树了。

" 实现"

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 502;
typedef long long ll;
typedef pair<int, int> pii;
ll d[MAXN][MAXN], dd[MAXN][MAXN], rk[MAXN][MAXN], val[MAXN];
const ll INF = 1e17;
int n, m;

bool cmp(int a, int b) { return val[a] < val[b]; }

void floyd() {
  for (int k = 1; k <= n; k++)
    for (int i = 1; i <= n; i++)
      for (int j = 1; j <= n; j++) d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}

struct node {
  ll u, v, w;
} a[MAXN * (MAXN - 1) / 2];

void solve() {
  // 求图的绝对中心
  floyd();
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      rk[i][j] = j;
      val[j] = d[i][j];
    }
    sort(rk[i] + 1, rk[i] + 1 + n, cmp);
  }
  ll P = 0, ansP = INF;
  // 在点上
  for (int i = 1; i <= n; i++) {
    if (d[i][rk[i][n]] * 2 < ansP) {
      ansP = d[i][rk[i][n]] * 2;
      P = i;
    }
  }
  // 在边上
  int f1 = 0, f2 = 0;
  ll disu = INT_MIN, disv = INT_MIN, ansL = INF;
  for (int i = 1; i <= m; i++) {
    ll u = a[i].u, v = a[i].v, w = a[i].w;
    for (int p = n, i = n - 1; i >= 1; i--) {
      if (d[v][rk[u][i]] > d[v][rk[u][p]]) {
```

```
        if (d[u][rk[u][i]] + d[v][rk[u][p]] + w < ansL) {
          ansL = d[u][rk[u][i]] + d[v][rk[u][p]] + w;
          f1 = u, f2 = v;
          disu = (d[u][rk[u][i]] + d[v][rk[u][p]] + w) / 2 - d[u][rk[u][i]];
          disv = w - disu;
        }
        p = i;
      }
    }
  }
  cout << min(ansP, ansL) / 2 << '\n';
  // 最小路径生成树
  vector<pii> pp;
  for (int i = 1; i <= 501; ++i)
    for (int j = 1; j <= 501; ++j) dd[i][j] = INF;
  for (int i = 1; i <= 501; ++i) dd[i][i] = 0;
  if (ansP <= ansL) {
    for (int j = 1; j <= n; j++) {
      for (int i = 1; i <= m; ++i) {
        ll u = a[i].u, v = a[i].v, w = a[i].w;
        if (dd[P][u] + w == d[P][v] && dd[P][u] + w < dd[P][v]) {
          dd[P][v] = dd[P][u] + w;
          pp.push_back({u, v});
        }
        u = a[i].v, v = a[i].u, w = a[i].w;
        if (dd[P][u] + w == d[P][v] && dd[P][u] + w < dd[P][v]) {
          dd[P][v] = dd[P][u] + w;
          pp.push_back({u, v});
        }
      }
    }
    for (auto [x, y] : pp) cout << x << ' ' << y << '\n';
  } else {
    d[n + 1][f1] = disu;
    d[f1][n + 1] = disu;
    d[n + 1][f2] = disv;
    d[f2][n + 1] = disv;
    a[m + 1].u = n + 1, a[m + 1].v = f1, a[m + 1].w = disu;
    a[m + 2].u = n + 1, a[m + 2].v = f2, a[m + 2].w = disv;
    n += 1;
    m += 2;
    floyd();
    P = n;
    for (int j = 1; j <= n; j++) {
      for (int i = 1; i <= m; ++i) {
        ll u = a[i].u, v = a[i].v, w = a[i].w;
        if (dd[P][u] + w == d[P][v] && dd[P][u] + w < dd[P][v]) {
          dd[P][v] = dd[P][u] + w;
          pp.push_back({u, v});
        }
        u = a[i].v, v = a[i].u, w = a[i].w;
        if (dd[P][u] + w == d[P][v] && dd[P][u] + w < dd[P][v]) {
          dd[P][v] = dd[P][u] + w;
          pp.push_back({u, v});
```

```
      }
    }
  }
  cout << f1 << ' ' << f2 << '\n';
  for (auto [x, y] : pp)
    if (x != n && y != n) cout << x << ' ' << y << '\n';
  }
}

void init() {
  for (int i = 1; i <= 501; ++i)
    for (int j = 1; j <= 501; ++j) d[i][j] = INF;
  for (int i = 1; i <= 501; ++i) d[i][i] = 0;
}

int main() {
  init();
  cin >> n >> m;
  for (int i = 1; i <= m; ++i) {
    ll u, v, w;
    cin >> u >> v >> w;
    w *= 2;
    d[u][v] = w, d[v][u] = w;
    a[i].u = u, a[i].v = v, a[i].w = w;
  }
  solve();
  return 0;
}
```

例题

SPOJ MDST[2]

timus 1569. Networking the "Iset"[3]

SPOJ PT07C - The GbAaY Kingdom[4]

参考文献

Play with Trees Solutions The GbAaY Kingdom[5]

参考资料与注释

[1] CodeForce 266D BerDonalds

[2] SPOJ MDST

[3] timus 1569. Networking the "Iset"

[4] SPOJ PT07C - The GbAaY Kingdom

[5] Play with Trees Solutions The GbAaY Kingdom

11.14 最短路

Authors: du33169

定义

（还记得这些定义吗？在阅读下列内容之前，请务必了解 图论相关概念 中的基础部分。）

- 路径
- 最短路
- 有向图中的最短路、无向图中的最短路
- 单源最短路、每对结点之间的最短路

性质

对于边权为正的图，任意两个结点之间的最短路，不会经过重复的结点。

对于边权为正的图，任意两个结点之间的最短路，不会经过重复的边。

对于边权为正的图，任意两个结点之间的最短路，任意一条的结点数不会超过 n，边数不会超过 $n-1$。

记号

为了方便叙述，这里先给出下文将会用到的一些记号的含义。

- n 为图上点的数目，m 为图上边的数目；
- s 为最短路的源点；
- $D(u)$ 为 s 点到 u 点的**实际**最短路长度；
- $dis(u)$ 为 s 点到 u 点的**估计**最短路长度。任何时候都有 $dis(u) \geq D(u)$。特别地，当最短路算法终止时，应有 $dis(u) = D(u)$。
- $w(u, v)$ 为 (u, v) 这一条边的边权。

Floyd 算法

是用来求任意两个结点之间的最短路的。

复杂度比较高，但是常数小，容易实现（只有三个 `for`）。

适用于任何图，不管有向无向，边权正负，但是最短路必须存在。（不能有个负环）

实现

我们定义一个数组 `f[k][x][y]`，表示只允许经过结点 1 到 k（也就是说，在子图 $V' = 1, 2, \ldots, k$ 中的路径，注意，x 与 y 不一定在这个子图中），结点 x 到结点 y 的最短路长度。

很显然，`f[n][x][y]` 就是结点 x 到结点 y 的最短路长度（因为 $V' = 1, 2, \ldots, n$ 即为 V 本身，其表示的最短路径就是所求路径）。

接下来考虑如何求出 `f` 数组的值。

`f[0][x][y]`：x 与 y 的边权，或者 0，或者 $+\infty$（`f[0][x][y]` 什么时候应该是 $+\infty$？当 x 与 y 间有直接相连的边的时候，为它们的边权；当 $x = y$ 的时候为零，因为到本身的距离为零；当 x 与 y 没有直接相连的边的时候，为 $+\infty$）。

f[k][x][y] = min(f[k-1][x][y], f[k-1][x][k]+f[k-1][k][y])（f[k-1][x][y]，为不经过 k 点的最短路径，而 f[k-1][x][k]+f[k-1][k][y]，为经过了 k 点的最短路）。

上面两行都显然是对的，所以说这个做法空间是 $O(N^3)$，我们需要依次增加问题规模（k 从 1 到 n），判断任意两点在当前问题规模下的最短路。

```
for (k = 1; k <= n; k++) {
  for (x = 1; x <= n; x++) {
    for (y = 1; y <= n; y++) {
      f[k][x][y] = min(f[k - 1][x][y], f[k - 1][x][k] + f[k - 1][k][y]);
    }
  }
}
```

```
for k in range(1, n + 1):
    for x in range(1, n + 1):
        for y in range(1, n + 1):
            f[k][x][y] = min(f[k - 1][x][y], f[k - 1][x][k] + f[k - 1][k][y])
```

因为第一维对结果无影响，我们可以发现数组的第一维是可以省略的，于是可以直接改成 f[x][y] = min(f[x][y], f[x][k]+f[k][y])。

" 证明第一维对结果无影响"

我们注意到如果放在一个给定第一维 k 二维数组中，f[x][k] 与 f[k][y] 在某一行和某一列。而 f[x][y] 则是该行和该列的交叉点上的元素。

现在我们需要证明将 f[k][x][y] 直接在原地更改也不会更改它的结果：我们注意到 f[k][x][y] 的涵义是第一维为 k-1 这一行和这一列的所有元素的最小值，包含了 f[k-1][x][y]，那么在原地进行更改也不会改变最小值的值，因为如果将该三维矩阵压缩为二维，则所求结果 f[x][y] 一开始即为原 f[k-1][x][y] 的值，最后依然会成为该行和该列的最小值。

故可以压缩。

```
for (k = 1; k <= n; k++) {
  for (x = 1; x <= n; x++) {
    for (y = 1; y <= n; y++) {
      f[x][y] = min(f[x][y], f[x][k] + f[k][y]);
    }
  }
}
```

```
for k in range(1, n + 1):
    for x in range(1, n + 1):
        for y in range(1, n + 1):
            f[x][y] = min(f[x][y], f[x][k] + f[k][y])
```

综上时间复杂度是 $O(N^3)$，空间复杂度是 $O(N^2)$。

应用

" 给一个正权无向图，找一个最小权值和的环。"

首先这一定是一个简单环。

想一想这个环是怎么构成的。

考虑环上编号最大的结点 u。

`f[u-1][x][y]` 和 (u,x),(u,y) 共同构成了环。

在 Floyd 的过程中枚举 u，计算这个和的最小值即可。

时间复杂度为 $O(n^3)$。

" 已知一个有向图中任意两点之间是否有连边，要求判断任意两点是否连通。"

该问题即是求**图的传递闭包**。

我们只需要按照 Floyd 的过程，逐个加入点判断一下。

只是此时的边的边权变为 1/0，而取 min 变成了**或**运算。

再进一步用 bitset 优化，复杂度可以到 $O(\frac{n^3}{w})$。

```
// std::bitset<SIZE> f[SIZE];
for (k = 1; k <= n; k++)
  for (i = 1; i <= n; i++)
    if (f[i][k]) f[i] = f[i] | f[k];
```

Bellman–Ford 算法

Bellman–Ford 算法是一种基于松弛（relax）操作的最短路算法，可以求出有负权的图的最短路，并可以对最短路不存在的情况进行判断。

在国内 OI 界，你可能听说过的「SPFA」，就是 Bellman–Ford 算法的一种实现。

过程

先介绍 Bellman–Ford 算法要用到的松弛操作（Dijkstra 算法也会用到松弛操作）。

对于边 (u,v)，松弛操作对应下面的式子：$dis(v)=\min(dis(v),dis(u)+w(u,v))$。

这么做的含义是显然的：我们尝试用 $S\to u\to v$（其中 $S\to u$ 的路径取最短路）这条路径去更新 v 点最短路的长度，如果这条路径更优，就进行更新。

Bellman–Ford 算法所做的，就是不断尝试对图上每一条边进行松弛。我们每进行一轮循环，就对图上所有的边都尝试进行一次松弛操作，当一次循环中没有成功的松弛操作时，算法停止。

每次循环是 $O(m)$ 的，那么最多会循环多少次呢？

在最短路存在的情况下，由于一次松弛操作会使最短路的边数至少 $+1$，而最短路的边数最多为 $n-1$，因此整个算法最多执行 $n-1$ 轮松弛操作。故总时间复杂度为 $O(nm)$。

但还有一种情况，如果从 S 点出发，抵达一个负环时，松弛操作会无休止地进行下去。注意到前面的论证中已经说明了，对于最短路存在的图，松弛操作最多只会执行 $n-1$ 轮，因此如果第 n 轮循环时仍然存在能松弛的边，说明从 S 点出发，能够抵达一个负环。

" 负环判断中存在的常见误区"

需要注意的是，以 S 点为源点跑 Bellman–Ford 算法时，如果没有给出存在负环的结果，只能说明从 S 点出发不能抵达一个负环，而不能说明图上不存在负环。

因此如果需要判断整个图上是否存在负环，最严谨的做法是建立一个超级源点，向图上每个节点连一条权值为 0 的边，然后以超级源点为起点执行 Bellman–Ford 算法。

实现

" 参考实现"

```
struct Edge {
  int u, v, w;
};

vector<Edge> edge;

int dis[MAXN], u, v, w;
const int INF = 0x3f3f3f3f;

bool bellmanford(int n, int s) {
  memset(dis, 0x3f, sizeof(dis));
  dis[s] = 0;
  bool flag = false;  // 判断一轮循环过程中是否发生松弛操作
  for (int i = 1; i <= n; i++) {
    flag = false;
    for (int j = 0; j < edge.size(); j++) {
      u = edge[j].u, v = edge[j].v, w = edge[j].w;
      if (dis[u] == INF) continue;
      // 无穷大与常数加减仍然为无穷大
      // 因此最短路长度为 INF 的点引出的边不可能发生松弛操作
      if (dis[v] > dis[u] + w) {
        dis[v] = dis[u] + w;
        flag = true;
      }
    }
    // 没有可以松弛的边时就停止算法
    if (!flag) {
      break;
    }
  }
  // 第 n 轮循环仍然可以松弛时说明 s 点可以抵达一个负环
  return flag;
}
```

```
class Edge:
    def __init__(self, u = 0, v = 0, w = 0):
        self.u = u
        self.v = v
        self.w = w

INF = 0x3f3f3f3f
edge = []
dis = [INF] * MAXN

def bellmanford(n, s):
    dis[s] = 0
    for i in range(1, n + 1):
        flag = False
        for e in edge:
            u, v, w = e.u, e.v, e.w
            if (dis[u] == INF):
                continue
            # 无穷大与常数加减仍然为无穷大
```

```python
            # 因此最短路长度为 INF 的点引出的边不可能发生松弛操作
            if dis[v] > dis[u] + w:
                dis[v] = dis[u] + w
                flag = True
        # 没有可以松弛的边时就停止算法
        if flag == False:
            break
    # 第 n 轮循环仍然可以松弛时说明 s 点可以抵达一个负环
    return flag
```

队列优化：SPFA

即 Shortest Path Faster Algorithm。

很多时候我们并不需要那么多无用的松弛操作。

很显然，只有上一次被松弛的结点，所连接的边，才有可能引起下一次的松弛操作。

那么我们用队列来维护「哪些结点可能会引起松弛操作」，就能只访问必要的边了。

SPFA 也可以用于判断 s 点是否能抵达一个负环，只需记录最短路经过了多少条边，当经过了至少 n 条边时，说明 s 点可以抵达一个负环。

"实现"

```cpp
struct edge {
  int v, w;
};

vector<edge> e[maxn];
int dis[maxn], cnt[maxn], vis[maxn];
queue<int> q;

bool spfa(int n, int s) {
  memset(dis, 63, sizeof(dis));
  dis[s] = 0, vis[s] = 1;
  q.push(s);
  while (!q.empty()) {
    int u = q.front();
    q.pop(), vis[u] = 0;
    for (auto ed : e[u]) {
      int v = ed.v, w = ed.w;
      if (dis[v] > dis[u] + w) {
        dis[v] = dis[u] + w;
        cnt[v] = cnt[u] + 1;  // 记录最短路经过的边数
        if (cnt[v] >= n) return false;
        // 在不经过负环的情况下，最短路至多经过 n - 1 条边
        // 因此如果经过了多于 n 条边，一定说明经过了负环
        if (!vis[v]) q.push(v), vis[v] = 1;
      }
    }
  }
  return true;
}
```

```python
from collections import deque
class Edge:
```

```python
    def __init__(self, v = 0, w = 0):
        self.v = v
        self.w = w

e = [[Edge() for i in range(maxn)] for j in range(maxn)]
dis = [0x3f3f3f3f] * maxn; cnt = [0] * maxn; vis = [False] * maxn

q = deque()
def spfa(n, s):
    dis[s] = 0
    vis[s] = True
    q.append(s)
    while q:
        u = q.popleft()
        vis[u] = False
        for ed in e[u]:
            v, w = ed.v, ed.w
            if dis[v] > dis[u] + w:
                dis[v] = dis[u] + w
                cnt[v] = cnt[u] + 1 # 记录最短路经过的边数
                if cnt[v] >= n:
                    return False
                # 在不经过负环的情况下，最短路至多经过 n - 1 条边
                # 因此如果经过了多于 n 条边，一定说明经过了负环
                if not vis[v]:
                    q.append(v)
                    vis[v] = True
```

虽然在大多数情况下 SPFA 跑得很快，但其最坏情况下的时间复杂度为 $O(nm)$，将其卡到这个复杂度也是不难的，所以考试时要谨慎使用（在没有负权边时最好使用 Dijkstra 算法，在有负权边且题目中的图没有特殊性质时，若 SPFA 是标算的一部分，题目不应当给出 Bellman–Ford 算法无法通过的数据范围）。

"Bellman–Ford 的其他优化"

除了队列优化（SPFA）之外，Bellman–Ford 还有其他形式的优化，这些优化在部分图上效果明显，但在某些特殊图上，最坏复杂度可能达到指数级。

- 堆优化：将队列换成堆，与 Dijkstra 的区别是允许一个点多次入队。在有负权边的图可能被卡成指数级复杂度。
- 栈优化：将队列换成栈（即将原来的 BFS 过程变成 DFS），在寻找负环时可能具有更高效率，但最坏时间复杂度仍然为指数级。
- LLL 优化：将普通队列换成双端队列，每次将入队结点距离和队内距离平均值比较，如果更大则插入至队尾，否则插入队首。
- SLF 优化：将普通队列换成双端队列，每次将入队结点距离和队首比较，如果更大则插入至队尾，否则插入队首。
- D´Esopo–Pape 算法：将普通队列换成双端队列，如果一个节点之前没有入队，则将其插入队尾，否则插入队首。

更多优化以及针对这些优化的 Hack 方法，可以看 fstqwq 在知乎上的回答[3]。

Dijkstra 算法

Dijkstra（/ dikstr /或/ d ikstr /）算法由荷兰计算机科学家 E. W. Dijkstra 于 1956 年发现，1959 年公开发表。是一种求解**非负权图**上单源最短路径的算法。

过程

将结点分成两个集合：已确定最短路长度的点集（记为 S 集合）的和未确定最短路长度的点集（记为 T 集合）。一开始所有的点都属于 T 集合。

初始化 $dis(s)=0$，其他点的 dis 均为 $+\infty$。

然后重复这些操作：

1. 从 T 集合中，选取一个最短路长度最小的结点，移到 S 集合中。
2. 对那些刚刚被加入 S 集合的结点的所有出边执行松弛操作。

直到 T 集合为空，算法结束。

时间复杂度

有多种方法来维护 1 操作中最短路长度最小的结点，不同的实现导致了 Dijkstra 算法时间复杂度上的差异。

- 暴力：不使用任何数据结构进行维护，每次 2 操作执行完毕后，直接在 T 集合中暴力寻找最短路长度最小的结点。2 操作总时间复杂度为 $O(m)$，1 操作总时间复杂度为 $O(n^2)$，全过程的时间复杂度为 $O(n^2+m)=O(n^2)$。
- 二叉堆：每成功松弛一条边 (u,v)，就将 v 插入二叉堆中（如果 v 已经在二叉堆中，直接修改相应元素的权值即可），1 操作直接取堆顶结点即可。共计 $O(m)$ 次二叉堆上的插入（修改）操作，$O(n)$ 次删除堆顶操作，而插入（修改）和删除的时间复杂度均为 $O(\log n)$，时间复杂度为 $O((n+m)\log n)=O(m\log n)$。
- 优先队列：和二叉堆类似，但使用优先队列时，如果同一个点的最短路被更新多次，因为先前更新时插入的元素不能被删除，也不能被修改，只能留在优先队列中，故优先队列内的元素个数是 $O(m)$ 的，时间复杂度为 $O(m\log m)$。
- Fibonacci 堆：和前面二者类似，但 Fibonacci 堆插入的时间复杂度为 $O(1)$，故时间复杂度为 $O(n\log n+m)$，时间复杂度最优。但因为 Fibonacci 堆较二叉堆不易实现，效率优势也不够大[1-1]，算法竞赛中较少使用。
- 线段树：和二叉堆原理类似，不过将每次成功松弛后插入二叉堆的操作改为在线段树上执行单点修改，而 1 操作则是线段树上的全局查询最小值。时间复杂度为 $O(m\log n)$。

在稀疏图中，$m=O(n)$，使用二叉堆实现的 Dijkstra 算法较 Bellman–Ford 算法具有较大的效率优势；而在稠密图中，$m=O(n^2)$，这时候使用暴力做法较二叉堆实现更优。

正确性证明

下面用数学归纳法证明，在**所有边权值非负**的前提下，Dijkstra 算法的正确性[2]。

简单来说，我们要证明的，就是在执行 1 操作时，取出的结点 u 最短路均已经被确定，即满足 $D(u)=dis(u)$。

初始时 $S=\emptyset$，假设成立。

接下来用反证法。

设 u 点为算法中第一个在加入 S 集合时不满足 $D(u)=dis(u)$ 的点。因为 s 点一定满足 $D(u)=dis(u)=0$，且它一定是第一个加入 S 集合的点，因此将 u 加入 S 集合前，$S\neq\emptyset$，如果不存在 s 到 u 的路径，则 $D(u)=dis(u)=+\infty$，与假设矛盾。

于是一定存在路径 $s\to x\to y\to u$，其中 y 为 $s\to u$ 路径上第一个属于 T 集合的点，而 x 为 y 的前驱结点（显然 $x\in S$）。需要注意的是，可能存在 $s=x$ 或 $y=u$ 的情况，即 $s\to x$ 或 $y\to u$ 可能是空路径。

因为在 u 结点之前加入的结点都满足 $D(u)=dis(u)$，所以在 x 点加入到 S 集合时，有 $D(x)=dis(x)$，此时边 (x,y) 会被松弛，从而可以证明，将 u 加入到 S 时，一定有 $D(y)=dis(y)$。

下面证明 $D(u)=dis(u)$ 成立。在路径 $s\to x\to y\to u$ 中，因为图上所有边边权非负，因此 $D(y)\leq D(u)$。从而 $dis(y)\leq D(y)\leq D(u)\leq dis(u)$。但是因为 u 结点在 1 过程中被取出 T 集合时，y 结点还没有被取出 T 集合，因此此时有 $dis(u)\leq dis(y)$，从而得到 $dis(y)=D(y)=D(u)=dis(u)$，这与 $D(u)\neq dis(u)$ 的假设矛盾，故假设不成立。

因此我们证明了，1 操作每次取出的点，其最短路均已经被确定。命题得证。

注意到证明过程中的关键不等式 $D(y)\leq D(u)$ 是在图上所有边边权非负的情况下得出的。当图上存在负权边时，

这一不等式不再成立，Dijkstra 算法的正确性将无法得到保证，算法可能会给出错误的结果。

实现

这里同时给出 $O(n^2)$ 的暴力做法实现和 $O(m \log m)$ 的优先队列做法实现。

" 暴力实现"

```
struct edge {
  int v, w;
};

vector<edge> e[maxn];
int dis[maxn], vis[maxn];

void dijkstra(int n, int s) {
  memset(dis, 63, sizeof(dis));
  dis[s] = 0;
  for (int i = 1; i <= n; i++) {
    int u = 0, mind = 0x3f3f3f3f;
    for (int j = 1; j <= n; j++)
      if (!vis[j] && dis[j] < mind) u = j, mind = dis[j];
    vis[u] = true;
    for (auto ed : e[u]) {
      int v = ed.v, w = ed.w;
      if (dis[v] > dis[u] + w) dis[v] = dis[u] + w;
    }
  }
}
```

```
class Edge:
    def __init(self, v = 0, w = 0):
        self.v = v
        self.w = w
e = [[Edge() for i in range(maxn)] for j in range(maxn)]
dis = [0x3f3f3f3f] * maxn; vis = [0] * maxn
def dijkstra(n, s):
    dis[s] = 0
    for i in range(1, n + 1):
        u = 0
        mind = 0x3f3f3f3f
        for j in range(1, n + 1):
            if vis[j] == False and dis[v] < mind:
                u = j
                mind = dis[j]
        vis[u] = True
        for ed in e[u]:
            v, w = ed.v, ed.w
            if dis[v] > dis[u] + w:
                dis[v] = dis[u] + w
```

" 优先队列实现"

```cpp
struct edge {
  int v, w;
};

struct node {
  int dis, u;

  bool operator>(const node& a) const { return dis > a.dis; }
};

vector<edge> e[maxn];
int dis[maxn], vis[maxn];
priority_queue<node, vector<node>, greater<node> > q;

void dijkstra(int n, int s) {
  memset(dis, 63, sizeof(dis));
  dis[s] = 0;
  q.push({0, s});
  while (!q.empty()) {
    int u = q.top().u;
    q.pop();
    if (vis[u]) continue;
    vis[u] = 1;
    for (auto ed : e[u]) {
      int v = ed.v, w = ed.w;
      if (dis[v] > dis[u] + w) {
        dis[v] = dis[u] + w;
        q.push({dis[v], v});
      }
    }
  }
}
```

```python
def dijkstra(e,s):
  '''
  输入:
  e: 邻接表
  s: 起点
  返回:
  dis: 从 s 到每个顶点的最短路长度
  '''
  dis = defaultdict(lambda:float("inf"))
  dis[s] = 0
  q = [(0,s)]
  vis = set()
  while q:
      _, u = heapq.heappop(q)
      if u in vis: continue
      vis.add(u)
      for v,w in e[u]:
          if dis[v] > dis[u] + w:
              dis[v] = dis[u] + w
              heapq.heappush(q,(dis[v],v))
```

```
    return dis
```

Johnson 全源最短路径算法

Johnson 和 Floyd 一样，是一种能求出无负环图上任意两点间最短路径的算法。该算法在 1977 年由 Donald B. Johnson 提出。

任意两点间的最短路可以通过枚举起点，跑 n 次 Bellman–Ford 算法解决，时间复杂度是 $O(n^2m)$ 的，也可以直接用 Floyd 算法解决，时间复杂度为 $O(n^3)$。

注意到堆优化的 Dijkstra 算法求单源最短路径的时间复杂度比 Bellman–Ford 更优，如果枚举起点，跑 n 次 Dijkstra 算法，就可以在 $O(nm\log m)$（取决于 Dijkstra 算法的实现）的时间复杂度内解决本问题，比上述跑 n 次 Bellman–Ford 算法的时间复杂度更优秀，在稀疏图上也比 Floyd 算法的时间复杂度更加优秀。

但 Dijkstra 算法不能正确求解带负权边的最短路，因此我们需要对原图上的边进行预处理，确保所有边的边权均非负。

一种容易想到的方法是给所有边的边权同时加上一个正数 x，从而让所有边的边权均非负。如果新图上起点到终点的最短路经过了 k 条边，则将最短路减去 kx 即可得到实际最短路。

但这样的方法是错误的。考虑下图：

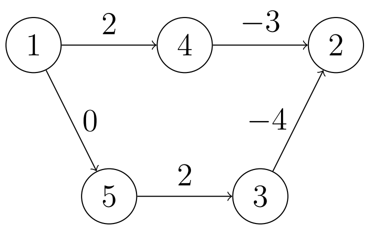


图 11.60

$1 \to 2$ 的最短路为 $1 \to 5 \to 3 \to 2$，长度为 -2。

但假如我们把每条边的边权加上 5 呢?

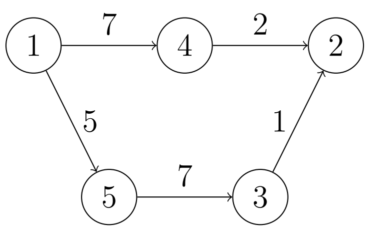


图 11.61

新图上 $1 \to 2$ 的最短路为 $1 \to 4 \to 2$，已经不是实际的最短路了。

Johnson 算法则通过另外一种方法来给每条边重新标注边权。

我们新建一个虚拟节点（在这里我们就设它的编号为 0）。从这个点向其他所有点连一条边权为 0 的边。

接下来用 Bellman–Ford 算法求出从 0 号点到其他所有点的最短路，记为 h_i。

假如存在一条从 u 点到 v 点，边权为 w 的边，则我们将该边的边权重新设置为 $w + h_u - h_v$。

接下来以每个点为起点，跑 n 轮 Dijkstra 算法即可求出任意两点间的最短路了。

一开始的 Bellman–Ford 算法并不是时间上的瓶颈，若使用 `priority_queue` 实现 Dijkstra 算法，该算法的时间复杂度是 $O(nm\log m)$。

正确性证明

为什么这样重新标注边权的方式是正确的呢?

在讨论这个问题之前，我们先讨论一个物理概念——势能。

诸如重力势能，电势能这样的势能都有一个特点，势能的变化量只和起点和终点的相对位置有关，而与起点到终点所走的路径无关。

势能还有一个特点，势能的绝对值往往取决于设置的零势能点，但无论将零势能点设置在哪里，两点间势能的差值是一定的。

接下来回到正题。

在重新标记后的图上，从 s 点到 t 点的一条路径 $s \to p_1 \to p_2 \to \dots \to p_k \to t$ 的长度表达式如下：

$(w(s,p_1)+h_s-h_{p_1})+(w(p_1,p_2)+h_{p_1}-h_{p_2})+\cdots+(w(p_k,t)+h_{p_k}-h_t)$

化简后得到：

$w(s,p_1)+w(p_1,p_2)+\cdots+w(p_k,t)+h_s-h_t$

无论我们从 s 到 t 走的是哪一条路径，h_s-h_t 的值是不变的，这正与势能的性质相吻合！

为了方便，下面我们就把 h_i 称为 i 点的势能。

上面的新图中 $s \to t$ 的最短路的长度表达式由两部分组成，前面的边权和为原图中 $s \to t$ 的最短路，后面则是两点间的势能差。因为两点间势能的差为定值，因此原图上 $s \to t$ 的最短路与新图上 $s \to t$ 的最短路相对应。

到这里我们的正确性证明已经解决了一半——我们证明了重新标注边权后图上的最短路径仍然是原来的最短路径。接下来我们需要证明新图中所有边的边权非负，因为在非负权图上，Dijkstra 算法能够保证得出正确的结果。

根据三角形不等式，图上任意一边 (u,v) 上两点满足：$h_v \leq h_u + w(u,v)$。这条边重新标记后的边权为 $w'(u,v)=w(u,v)+h_u-h_v \geq 0$。这样我们证明了新图上的边权均非负。

这样，我们就证明了 Johnson 算法的正确性。

不同方法的比较

最短路算法	Floyd	Bellman–Ford	Dijkstra	Johnson
最短路类型	每对结点之间的最短路	单源最短路	单源最短路	每对结点之间的最短路
作用于	任意图	任意图	非负权图	任意图
能否检测负环?	能	能	不能	能
时间复杂度	$O(N^3)$	$O(NM)$	$O(M\log M)$	$O(NM\log M)$

注：表中的 Dijkstra 算法在计算复杂度时均用 `priority_queue` 实现。

输出方案

开一个 `pre` 数组，在更新距离的时候记录下来后面的点是如何转移过去的，算法结束前再递归地输出路径即可。

比如 Floyd 就要记录 `pre[i][j] = k;`，Bellman–Ford 和 Dijkstra 一般记录 `pre[v] = u`。

参考资料与注释

[1] Worst case of fibonacci heap - Wikipedia [1-1] [1-2]

[2]《算法导论（第 3 版中译本）》，机械工业出版社，2013 年，第 384 - 385 页。

[3] fstqwq 在知乎上的回答

11.15 拆点

Authors: Anguei, sshwy, Xeonacid, Ir1d, MonkeyOliver, hsfzLZH1

拆点是一种图论建模思想，常用于 网络流，用来处理**点权或者点的流量限制**的问题，也常用于**分层图**。

结点有流量限制的最大流

如果把结点转化成边，那么这个问题就可以套板子解决了。

我们考虑把有流量限制的结点转化成这样一种形式：由两个结点 u, v 和一条边 $\langle u, v\rangle$ 组成的部分。其中，结点 u 承接所有从原图上其他点的出发到原图上该点的边，结点 v 引出所有从原图上该点出发到达原图上其他点的边。边 $\langle u, v\rangle$ 的流量限制为原图该点的流量限制，再套板子就可以解决本题。这就是拆点的基本思想。

如果原图是这样：

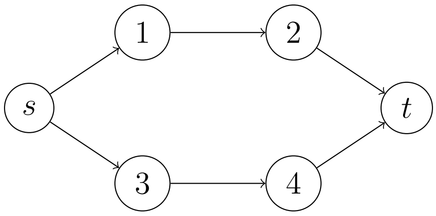


图 11.62

拆点之后的图是这个样子：

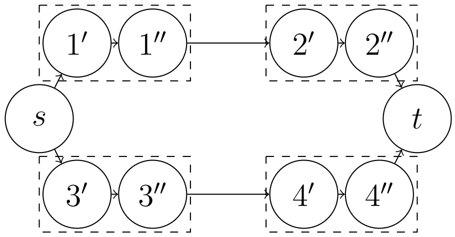


图 11.63

分层图最短路

分层图最短路，如：有 k 次零代价通过一条路径，求总的最小花费。对于这种题目，我们可以采用 DP 相关的思想，设 $\text{dis}_{i,j}$ 表示当前从起点 i 号结点，使用了 j 次免费通行权限后的最短路径。显然，dis 数组可以这么转移：

$\text{dis}_{i,j} = \min\{\min\{\text{dis}_{from,j-1}\}, \min\{\text{dis}_{from,j} + w\}\}$

其中，$from$ 表示 i 的父亲节点，w 表示当前所走的边的边权。当 $j - 1 \geq k$ 时，$\text{dis}_{from,j}=\infty$。

事实上，这个 DP 就相当于把每个结点拆分成了 $k + 1$ 个结点，每个新结点代表使用不同多次免费通行后到达的原图结点。换句话说，就是每个结点 u_i 表示使用 i 次免费通行权限后到达 u 结点。

"「JLOI2011」 飞行路线[1]"

题意：有一个 n 个点 m 条边的无向图，你可以选择 k 条道路以零代价通行，求 s 到 t 的最小花费。

参考核心代码：

```
struct State {    // 优先队列的结点结构体
  int v, w, cnt;  // cnt 表示已经使用多少次免费通行权限
```

```
  State() {}

  State(int v, int w, int cnt) : v(v), w(w), cnt(cnt) {}

  bool operator<(const State &rhs) const { return w > rhs.w; }
};

void dijkstra() {
  memset(dis, 0x3f, sizeof dis);
  dis[s][0] = 0;
  pq.push(State(s, 0, 0));  // 到起点不需要使用免费通行权，距离为零
  while (!pq.empty()) {
    const State top = pq.top();
    pq.pop();
    int u = top.v, nowCnt = top.cnt;
    if (done[u][nowCnt]) continue;
    done[u][nowCnt] = true;
    for (int i = head[u]; i; i = edge[i].next) {
      int v = edge[i].v, w = edge[i].w;
      if (nowCnt < k && dis[v][nowCnt + 1] > dis[u][nowCnt]) {  // 可以免费通行
        dis[v][nowCnt + 1] = dis[u][nowCnt];
        pq.push(State(v, dis[v][nowCnt + 1], nowCnt + 1));
      }
      if (dis[v][nowCnt] > dis[u][nowCnt] + w) {  // 不可以免费通行
        dis[v][nowCnt] = dis[u][nowCnt] + w;
        pq.push(State(v, dis[v][nowCnt], nowCnt));
      }
    }
  }
}

int main() {
  n = read(), m = read(), k = read();
  // 笔者习惯从 1 到 n 编号，而这道题是从 0 到 n - 1，所以要处理一下
  s = read() + 1, t = read() + 1;
  while (m--) {
    int u = read() + 1, v = read() + 1, w = read();
    add(u, v, w), add(v, u, w);  // 这道题是双向边
  }
  dijkstra();
  int ans = std::numeric_limits<int>::max();  // ans 取 int 最大值为初值
  for (int i = 0; i <= k; ++i)
    ans = std::min(ans, dis[t][i]);  // 对到达终点的所有情况取最优值
  println(ans);
}
```

参考资料与注释

[1] 「JLOI2011」飞行路线

11.16 差分约束

Authors: Ir1d, Anguei, hsfzLZH1

定义

差分约束系统是一种特殊的 n 元一次不等式组，它包含 n 个变量 $x_1, x_2, \dots, x_n$ 以及 m 个约束条件，每个约束条件是由两个其中的变量做差构成的，形如 $x_i - x_j \le c_k$，其中 $1 \le i, j \le n, i \ne j, 1 \le k \le m$ 并且 c_k 是常数（可以是非负数，也可以是负数）。我们要解决的问题是：求一组解 $x_1 = a_1, x_2 = a_2, \dots, x_n = a_n$，使得所有的约束条件得到满足，否则判断出无解。

差分约束系统中的每个约束条件 $x_i - x_j \le c_k$ 都可以变形成 $x_i \le x_j + c_k$，这与单源最短路中的三角形不等式 $dist[y] \le dist[x] + z$ 非常相似。因此，我们可以把每个变量 x_i 看做图中的一个结点，对于每个约束条件 $x_i - x_j \le c_k$，从结点 j 向结点 i 连一条长度为 c_k 的有向边。

注意到，如果 $\{a_1, a_2, \dots, a_n\}$ 是该差分约束系统的一组解，那么对于任意的常数 d，$\{a_1 + d, a_2 + d, \dots, a_n + d\}$ 显然也是该差分约束系统的一组解，因为这样做差后 d 刚好被消掉。

过程

设 $dist[0] = 0$ 并向每一个点连一条权重为 0 边，跑单源最短路，若图中存在负环，则给定的差分约束系统无解，否则，$x_i = dist[i]$ 为该差分约束系统的一组解。

性质

一般使用 Bellman–Ford 或队列优化的 Bellman–Ford（俗称 SPFA，在某些随机图跑得很快）判断图中是否存在负环，最坏时间复杂度为 $O(nm)$。

常用变形技巧

例题 luogu P1993 小 K 的农场[1]

题目大意：求解差分约束系统，有 m 条约束条件，每条都为形如 $x_a - x_b \ge c_k$，$x_a - x_b \le c_k$ 或 $x_a = x_b$ 的形式，判断该差分约束系统有没有解。

题意	转化	连边
$x_a - x_b \ge c$	$x_b - x_a \le -c$	`add(a, b, -c);`
$x_a - x_b \le c$	$x_a - x_b \le c$	`add(b, a, c);`
$x_a = x_b$	$x_a - x_b \le 0, x_b - x_a \le 0$	`add(b, a, 0), add(a, b, 0);`

跑判断负环，如果不存在负环，输出 `Yes`，否则输出 `No`。

" 参考代码"

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <queue>
using namespace std;

struct edge {
```

```
  int v, w, next;
} e[40005];

int head[10005], vis[10005], tot[10005], cnt;
long long ans, dist[10005];
queue<int> q;

void addedge(int u, int v, int w) {  // 加边
  e[++cnt].v = v;
  e[cnt].w = w;
  e[cnt].next = head[u];
  head[u] = cnt;
}

int main() {
  int n, m;
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= m; i++) {
    int op, x, y, z;
    scanf("%d", &op);
    if (op == 1) {
      scanf("%d%d%d", &x, &y, &z);
      addedge(y, x, z);
    } else if (op == 2) {
      scanf("%d%d%d", &x, &y, &z);
      addedge(x, y, -z);
    } else {
      scanf("%d%d", &x, &y);
      addedge(x, y, 0);
      addedge(y, x, 0);
    }
  }
  for (int i = 1; i <= n; i++) addedge(0, i, 0);
  memset(dist, -0x3f, sizeof(dist));
  dist[0] = 0;
  vis[0] = 1;
  q.push(0);
  while (!q.empty()) {  // 判负环，看上面的
    int cur = q.front();
    q.pop();
    vis[cur] = 0;
    for (int i = head[cur]; i; i = e[i].next)
      if (dist[cur] + e[i].w > dist[e[i].v]) {
        dist[e[i].v] = dist[cur] + e[i].w;
        if (!vis[e[i].v]) {
          vis[e[i].v] = 1;
          q.push(e[i].v);
          tot[e[i].v]++;
          if (tot[e[i].v] >= n) {
            puts("No");
            return 0;
          }
        }
      }
```

```
  }
  puts("Yes");
  return 0;
}
```

例题 P4926[1007] 倍杀测量者[2]

不考虑二分等其他的东西，这里只论述差分系统 $\frac{x_i}{x_j} \le c_k$ 的求解方法。

对每个 x_i, x_j 和 c_k 取一个 log 就可以把乘法变成加法运算，即 $\log x_i - \log x_j \le \log c_k$，这样就可以用差分约束解决了。

Bellman–Ford 判负环代码实现

下面是用 Bellman–Ford 算法判断图中是否存在负环的代码实现，请在调用前先保证图是连通的。

" 实现"

```
bool Bellman_Ford() {
  for (int i = 0; i < n; i++) {
    bool jud = false;
    for (int j = 1; j <= n; j++)
      for (int k = h[j]; ~k; k = nxt[k])
        if (dist[j] > dist[p[k]] + w[k])
          dist[j] = dist[p[k]] + w[k], jud = true;
    if (!jud) break;
  }
  for (int i = 1; i <= n; i++)
    for (int j = h[i]; ~j; j = nxt[j])
      if (dist[i] > dist[p[j]] + w[j]) return false;
  return true;
}

def Bellman_Ford():
    for i in range(0, n):
        jud = False
        for j in range(1, n + 1):
            while ~k:
                k = h[j]
                if dist[j] > dist[p[k]] + w[k]:
                    dist[j] = dist[p[k]] + w[k]; jud = True
                k = nxt[k]
        if jud == False:
            break
    for i in range(1, n + 1):
        while ~j:
            j = h[i]
            if dist[i] > dist[p[j]] + w[j]:
                return False
            j = nxt[j]
    return True
```

习题

Usaco2006 Dec Wormholes 虫洞[3]

「SCOI2011」 糖果[4]

POJ 1364 King[5]

POJ 2983 Is the Information Reliable?[6]

参考资料与注释

[1] luogu P1993 小 K 的农场

[2] P4926[1007] 倍杀测量者

[3] Usaco2006 Dec Wormholes 虫洞

[4] 「SCOI2011」糖果

[5] POJ 1364 King

[6] POJ 2983 Is the Information Reliable?

11.17 k 短路

问题描述

给定一个有 n 个结点，m 条边的有向图，求从 s 到 t 的所有不同路径中的第 k 短路径的长度。

A * 算法

A * 算法定义了一个对当前状态 x 的估价函数 $f(x)=g(x)+h(x)$，其中 $g(x)$ 为从初始状态到达当前状态的实际代价，$h(x)$ 为从当前状态到达目标状态的最佳路径的估计代价。每次取出 $f(x)$ 最优的状态 x，扩展其所有子状态，可以用**优先队列**来维护这个值。

在求解 k 短路问题时，令 $h(x)$ 为从当前结点到达终点 t 的最短路径长度。可以通过在反向图上对结点 t 跑单源最短路预处理出对每个结点的这个值。

由于设计的距离函数和估价函数，对于每个状态需要记录两个值，为当前到达的结点 x 和已经走过的距离 $g(x)$，将这种状态记为 $(x,g(x))$。

开始我们将初始状态 $(s,0)$ 加入优先队列。每次我们取出估价函数 $f(x)=g(x)+h(x)$ 最小的一个状态，枚举该状态到达的结点 x 的所有出边，将对应的子状态加入优先队列。当我们访问到一个结点第 k 次时，对应的状态的 $g(x)$ 就是从 x 到该结点的第 k 短路。

优化：由于只需要求出从初始结点到目标结点的第 k 短路，所以已经取出的状态到达一个结点的次数大于 k 次时，可以不扩展其子状态。因为之前 k 次已经形成了 k 条合法路径，当前状态不会影响到最后的答案。

当图的形态是一个 n 元环的时候，该算法最坏是 $O(nk\log n)$ 的。但是这种算法可以在相同的复杂度内求出从起始点 s 到每个结点的前 k 短路。

实现

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <queue>
using namespace std;
const int maxn = 5010;
const int maxm = 400010;
const int inf = 2e9;
int n, m, s, t, k, u, v, ww, H[maxn], cnt[maxn];
int cur, h[maxn], nxt[maxm], p[maxm], w[maxm];
int cur1, h1[maxn], nxt1[maxm], p1[maxm], w1[maxm];
bool tf[maxn];

void add_edge(int x, int y, double z) {
  cur++;
  nxt[cur] = h[x];
  h[x] = cur;
  p[cur] = y;
  w[cur] = z;
}

void add_edge1(int x, int y, double z) {
  cur1++;
  nxt1[cur1] = h1[x];
  h1[x] = cur1;
  p1[cur1] = y;
  w1[cur1] = z;
}

struct node {
  int x, v;

  bool operator<(node a) const { return v + H[x] > a.v + H[a.x]; }
};

priority_queue<node> q;

struct node2 {
  int x, v;

  bool operator<(node2 a) const { return v > a.v; }
} x;

priority_queue<node2> Q;

int main() {
  scanf("%d%d%d%d%d", &n, &m, &s, &t, &k);
  while (m--) {
    scanf("%d%d%d", &u, &v, &ww);
    add_edge(u, v, ww);
    add_edge1(v, u, ww);
  }
  for (int i = 1; i <= n; i++) H[i] = inf;
  Q.push({t, 0});
```

```
  while (!Q.empty()) {
    x = Q.top();
    Q.pop();
    if (tf[x.x]) continue;
    tf[x.x] = true;
    H[x.x] = x.v;
    for (int j = h1[x.x]; j; j = nxt1[j]) Q.push({p1[j], x.v + w1[j]});
  }
  q.push({s, 0});
  while (!q.empty()) {
    node x = q.top();
    q.pop();
    cnt[x.x]++;
    if (x.x == t && cnt[x.x] == k) {
      printf("%d\n", x.v);
      return 0;
    }
    if (cnt[x.x] > k) continue;
    for (int j = h[x.x]; j; j = nxt[j]) q.push({p[j], x.v + w[j]});
  }
  printf("-1\n");
  return 0;
}
```

可持久化可并堆优化 k 短路算法

最短路树与任意路径

定义

在反向图上从 t 开始跑最短路，设在原图上结点 x 到 t 的最短路长度为 $dist_x$，建出**任意**一棵以 t 为根的最短路树 T。

所谓最短路径树，就是满足从树上的每个结点 x 到根节点 t 的简单路径都是 x 到 t 的**其中**一条最短路径。

性质

设一条从 s 到 t 的路径经过的边集为 P，去掉 P 中与 T 的交集得到 P'。

P' 有如下性质：

1. 对于一条不在 T 上的边 e，其为从 u 到 v 的一条边，边权为 w，定义其代价 $\Delta e = dist_v + w - dist_u$，即为选择该边后路径长度的增加量。则路径 P 的长度 $L_P = dist_s + \sum_{e\in P'} \Delta e$。
2. 将 P 和 P' 中的所有边按照从 s 到 t 所经过的顺序依次排列，则对于 P' 中相邻的两条边 e_1, e_2，有 u_{e_2} 与 v_{e_1} 相等或为其在 T 上的祖先。因为在 P 中 e_1, e_2 直接相连或中间都为树边。
3. 对于一个确定存在的 P'，有且仅有一个 S，使得 $S' = P'$。因为由于性质 2，P' 中相邻的两条边的起点和终点之间在 T 上只有一条路径。

问题转化

性质 1 告诉我们知道集合 P' 后，如何求出 L_P 的值。

性质 2 告诉我们所有 P' 一定满足的条件，所有满足这个条件的边集 P' 都是合法的，也就告诉我们生成 P' 的方法。

性质 3 告诉我们对于每个合法的 P' 有且仅有一个边集 P 与之对应。

那么问题转化为：求 L_P 的值第 k 小的满足性质 2 的集合 P'。

过程

由于性质 2，我们可以记录按照从 s 到 t 的顺序排列的最后一条边和 L_P 的值，来表示一个边集 P'。

我们用一个小根堆来维护这样的边集 P'。

初始我们将起点为 1 或 1 在 T 上的祖先的所有的边中 Δe 最小的一条边加入小根堆。

每次取出堆顶的一个边集 S，有两种方法可以生成可能的新边集：

1. 替换 S 中的最后一条边为满足相同条件的 Δe 更大的边。
2. 在最后一条边后接上一条边，设 x 为 S 中最后一条边的终点，由性质 2 可得这条边需要满足其起点为 x 或 x 在 T 上的祖先。

将生成的新边集也加入小根堆。重复以上操作 $k-1$ 次后求出的就是从 s 到 t 的第 k 短路。

对于每个结点 x，我们将以其为起点的边的 Δe 建成一个小根堆。为了方便查找一个结点 x 与 x 在 T 上的祖先在小根堆上的信息，我们将这些信息合并在一个编号为 x 的小根堆上。回顾以上生成新边集的方法，我们发现只要我们把紧接着可能的下一个边集加入小根堆，并保证这种生成方法可以覆盖所有可能的边集即可。记录最后选择的一条边在堆上对应的结点 t，有更优的方法生成新的边集：

1. 替换 S 中的最后一条边为 t 在堆上的左右儿子对应的边。
2. 在最后一条边后接上一条新的边，设 x 为 S 中最后一条边的终点，则接上编号为 x 的小根堆的堆顶结点对应的边。

用这种方法，每次生成新的边集只会扩展出最多三个结点，小根堆中的结点总数是 $O(n+k)$。

所以此算法的瓶颈在合并一个结点与其在 T 上的祖先的信息，如果使用朴素的二叉堆，时间复杂度为 $O(nm\log m)$，空间复杂度为 $O(nm)$；如果使用可并堆，每次仍然需要复制堆中的全部结点，时间复杂度同样无法承受。

可持久化可并堆优化

在阅读本内容前，请先了解 可持久化可并堆 的相关知识。

使用可持久化可并堆优化合并一个结点与其在 T 上的祖先的信息，

每次将一个结点与其在 T 上的父亲合并，时间复杂度为 $O((n+m)\log m+k\log k)$，空间复杂度为 $O(m+n\log m+k)$。这样在求出一个结点对应的堆时，无需复制结点且之后其父亲结点对应的堆仍然可以正常访问。

注意的是，如上文所言，最终询问时不需要可并堆的合并操作。

询问时使用优先队列维护可并堆的根，对于可并堆堆顶的删除，直接将其左右儿子加入优先队列中，

就只需要 $O(k)$ 而非 $O(k\log m)$ 的空间。

实现

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <queue>
using namespace std;
const int maxn = 200010;
int n, m, s, t, k, x, y, ww, cnt, fa[maxn];

struct Edge {
  int cur, h[maxn], nxt[maxn], p[maxn], w[maxn];

  void add_edge(int x, int y, int z) {
```

```
    cur++;
    nxt[cur] = h[x];
    h[x] = cur;
    p[cur] = y;
    w[cur] = z;
  }
} e1, e2;

int dist[maxn];
bool tf[maxn], vis[maxn], ontree[maxn];

struct node {
  int x, v;

  node* operator=(node a) {
    x = a.x;
    v = a.v;
    return this;
  }

  bool operator<(node a) const { return v > a.v; }
} a;

priority_queue<node> Q;

void dfs(int x) {
  vis[x] = true;
  for (int j = e2.h[x]; j; j = e2.nxt[j])
    if (!vis[e2.p[j]])
      if (dist[e2.p[j]] == dist[x] + e2.w[j])
        fa[e2.p[j]] = x, ontree[j] = true, dfs(e2.p[j]);
}

struct LeftistTree {
  int cnt, rt[maxn], lc[maxn * 20], rc[maxn * 20], dist[maxn * 20];
  node v[maxn * 20];

  LeftistTree() { dist[0] = -1; }

  int newnode(node w) {
    cnt++;
    v[cnt] = w;
    return cnt;
  }

  int merge(int x, int y) {
    if (!x || !y) return x + y;
    if (v[x] < v[y]) swap(x, y);
    int p = ++cnt;
    lc[p] = lc[x];
    v[p] = v[x];
    rc[p] = merge(rc[x], y);
    if (dist[lc[p]] < dist[rc[p]]) swap(lc[p], rc[p]);
    dist[p] = dist[rc[p]] + 1;
```

```
    return p;
  }
} st;

void dfs2(int x) {
  vis[x] = true;
  if (fa[x]) st.rt[x] = st.merge(st.rt[x], st.rt[fa[x]]);
  for (int j = e2.h[x]; j; j = e2.nxt[j])
    if (fa[e2.p[j]] == x && !vis[e2.p[j]]) dfs2(e2.p[j]);
}

int main() {
  scanf("%d%d%d%d%d", &n, &m, &s, &t, &k);
  for (int i = 1; i <= m; i++)
    scanf("%d%d%d", &x, &y, &ww), e1.add_edge(x, y, ww), e2.add_edge(y, x, ww);
  Q.push({t, 0});
  while (!Q.empty()) {
    a = Q.top();
    Q.pop();
    if (tf[a.x]) continue;
    tf[a.x] = true;
    dist[a.x] = a.v;
    for (int j = e2.h[a.x]; j; j = e2.nxt[j]) Q.push({e2.p[j], a.v + e2.w[j]});
  }
  if (k == 1) {
    if (tf[s])
      printf("%d\n", dist[s]);
    else
      printf("-1\n");
    return 0;
  }
  dfs(t);
  for (int i = 1; i <= n; i++)
    if (tf[i])
      for (int j = e1.h[i]; j; j = e1.nxt[j])
        if (!ontree[j])
          if (tf[e1.p[j]])
            st.rt[i] = st.merge(
                st.rt[i],
                st.newnode({e1.p[j], dist[e1.p[j]] + e1.w[j] - dist[i]}));
  for (int i = 1; i <= n; i++) vis[i] = false;
  dfs2(t);
  if (st.rt[s]) Q.push({st.rt[s], dist[s] + st.v[st.rt[s]].v});
  while (!Q.empty()) {
    a = Q.top();
    Q.pop();
    cnt++;
    if (cnt == k - 1) {
      printf("%d\n", a.v);
      return 0;
    }
    if (st.lc[a.x])  // 可并堆删除直接把左右儿子加入优先队列中
      Q.push({st.lc[a.x], a.v - st.v[a.x].v + st.v[st.lc[a.x]].v});
    if (st.rc[a.x])
```

```
      Q.push({st.rc[a.x], a.v - st.v[a.x].v + st.v[st.rc[a.x]].v});
    x = st.rt[st.v[a.x].x];
    if (x) Q.push({x, a.v + st.v[x].v});
  }
  printf("-1\n");
  return 0;
}
```

习题

「SDOI2010」 魔法猪学院[1]

参考资料与注释

[1] 「SDOI2010」魔法猪学院

11.18 同余最短路

当出现形如「给定 n 个整数，求这 n 个整数能拼凑出多少的其他整数（n 个整数可以重复取）」，以及「给定 n 个整数，求这 n 个整数不能拼凑出的最小（最大）的整数」，或者「至少要拼几次才能拼出模 K 余 p 的数」的问题时可以使用同余最短路的方法。

同余最短路利用同余来构造一些状态，可以达到优化空间复杂度的目的。

类比 差分约束 方法，利用同余构造的这些状态可以看作单源最短路中的点。同余最短路的状态转移通常是这样的 $f(i+y)=f(i)+y$，类似单源最短路中 $f(v)=f(u)+edge(u,v)$。

例题

例题一

"P3403 跳楼机[1]"

题目大意：给定 x y z h，对于 $k \in [1,h]$，有多少个 k 能够满足 $ax+by+cz=k$。($0 \le a,b,c$，$1 \le x,y,z \le 10^5$，$h \le 2^{63}-1$)

不妨假设 $x<y<z$。

令 d_i 为只通过**操作 2** 和**操作 3**，需满足 $p \bmod x = i$ 能够达到的最低楼层 p，即**操作 2** 和**操作 3** 操作后能得到的模 x 下与 i 同余的最小数，用来计算该同余类满足条件的数个数。

可以得到两个状态：

- $i \xrightarrow{y} (i+y) \bmod x$
- $i \xrightarrow{z} (i+z) \bmod x$

注意通常选取一组 a_i 中最小的那个数对它取模，也就是此处的 x，这样可以尽量减小空间复杂度（剩余系最小）。

那么实际上相当于执行了最短路中的建边操作：

`add(i, (i+y) % x, y)`

`add(i, (i+z) % x, z)`

接下来只需要求出 $d_0,d_1,d_2,\dots,d_{x-1}$，只需要跑一次最短路就可求出相应的 d_i。

与差分约束问题相同，当存在一组解 $\{a_1,a_2,\cdots,a_n\}$ 时，$\{a_1+d,a_2+d,\cdots,a_n+d\}$ 同样为一组解，因此在该题让 $i=1$ 作为源点，此时源点处的 $dis_1=1$ 在已知范围内最小，因此得到的也是一组最小的解。

答案即为：

$$\sum_{i=0}^{x-1}\left(\frac{h-d_i}{x}+1\right)$$

加 1 是由于 d_i 所在楼层也算一次。

" 参考实现"

```
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;
const int maxn = 100010;
const int INF = 0x3f3f3f3f;

ll h, x, y, z;
ll head[maxn << 1], tot;
ll dis[maxn], vis[maxn];
queue<int> q;

struct edge {
  ll to, next, w;
} e[maxn << 1];

void add(ll u, ll v, ll w) {
  e[++tot] = edge{v, head[u], w};
  head[u] = tot;
}

void spfa() {  // spfa 算法，可看最短路部分
  dis[1] = 1;
  vis[1] = 1;
  q.push(1);
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    vis[u] = 0;
    for (int i = head[u]; i; i = e[i].next) {
      int v = e[i].to, w = e[i].w;
      if (dis[v] > dis[u] + w) {
        dis[v] = dis[u] + w;
        if (!vis[v]) {
          q.push(v);
          vis[v] = 1;
        }
      }
    }
  }
}

int main() {
  memset(dis, INF, sizeof(dis));
  scanf("%lld", &h);
  scanf("%lld %lld %lld", &x, &y, &z);
  if (x == 1 || y == 1 || z == 1) {
```

```
    printf("%lld\n", h);
    return 0;
  }
  for (int i = 0; i < x; i++) {
    add(i, (i + z) % x, z);
    add(i, (i + y) % x, y);
  }
  spfa();
  ll ans = 0;
  for (int i = 0; i < x; i++) {
    if (h >= dis[i]) ans += (h - dis[i]) / x + 1;
  }
  printf("%lld\n", ans);
  return 0;
}
```

例题二

"ARC084B Small Multiple[2]"

题目大意：给定 n，求 n 的倍数中，数位和最小的那一个的数位和。（$1 \le n \le 10^5$）

本题可以使用循环卷积优化完全背包在 $O(n\log^2 n)$ 的时间内解决，但我们希望得到线性的算法。

观察到任意一个正整数都可以从 1 开始，按照某种顺序执行乘 10、加 1 的操作，最终得到，而其中加 1 操作的次数就是这个数的数位和。这提示我们使用最短路。

对于所有 $0 \le k \le n-1$，从 k 向 $10k$ 连边权为 0 的边；从 k 向 $k+1$ 连边权为 1 的边。（点的编号均在模 n 意义下）

每个 n 的倍数在这个图中都对应了 1 号点到 0 号点的一条路径，求出 1 到 0 的最短路即可。某些路径不合法（如连续走了 10 条边权为 1 的边），但这些路径产生的答案一定不优，不影响答案。

时间复杂度 $O(n)$。

习题

洛谷　P3403　跳楼机[1]

洛谷　P2662　牛场围栏[3]

［国家集训队］墨墨的等式[4]

「NOIP2018」 货币系统[5]

AGC057D - Sum Avoidance[6]

「THUPC 2023　初赛 」 背包[7]

参考资料与注释

[1] P3403 跳楼机 [1-1] [1-2]

[2] ARC084B Small Multiple

[3] 洛谷 P2662 牛场围栏

[4] [国家集训队] 墨墨的等式

[5] 「NOIP2018」货币系统

[6] AGC057D - Sum Avoidance

[7] 「THUPC 2023 初赛」背包

11.19 连通性相关

11.19.1 强连通分量

简介

在阅读下列内容之前，请务必了解 图论相关概念 中的基础部分。

强连通的定义是：有向图 G 强连通是指，G 中任意两个结点连通。

强连通分量（Strongly Connected Components，SCC）的定义是：极大的强连通子图。

这里要介绍的是如何来求强连通分量。

Tarjan 算法

引入

Robert E. Tarjan（罗伯特 · 塔扬，1948~），生于美国加州波莫纳，计算机科学家。

Tarjan 发明了很多算法和数据结构。不少他发明的算法都以他的名字命名，以至于有时会让人混淆几种不同的算法。比如求各种连通分量的 Tarjan 算法，求 LCA（Lowest Common Ancestor，最近公共祖先）的 Tarjan 算法。并查集、Splay、Toptree 也是 Tarjan 发明的。

我们这里要介绍的是在有向图中求强连通分量的 Tarjan 算法。

DFS 生成树

在介绍该算法之前，先来了解 **DFS 生成树**，我们以下面的有向图为例：

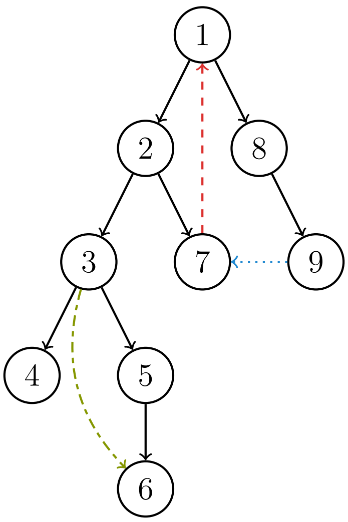


图 11.64 DFS 生成树

有向图的 DFS 生成树主要有 4 种边（不一定全部出现）：

1. 树边（tree edge）：示意图中以黑色边表示，每次搜索找到一个还没有访问过的结点的时候就形成了一条树边。

2. 反祖边（back edge）：示意图中以红色边表示（即 $7 \to 1$），也被叫做回边，即指向祖先结点的边。
3. 横叉边（cross edge）：示意图中以蓝色边表示（即 $9 \to 7$），它主要是在搜索的时候遇到了一个已经访问过的结点，但是这个结点**并不是**当前结点的祖先。
4. 前向边（forward edge）：示意图中以绿色边表示（即 $3 \to 6$），它是在搜索的时候遇到子树中的结点的时候形成的。

我们考虑 DFS 生成树与强连通分量之间的关系。

如果结点 u 是某个强连通分量在搜索树中遇到的第一个结点，那么这个强连通分量的其余结点肯定是在搜索树中以 u 为根的子树中。结点 u 被称为这个强连通分量的根。

反证法：假设有个结点 v 在该强连通分量中但是不在以 u 为根的子树中，那么 u 到 v 的路径中肯定有一条离开子树的边。但是这样的边只可能是横叉边或者反祖边，然而这两条边都要求指向的结点已经被访问过了，这就和 u 是第一个访问的结点矛盾了。得证。

Tarjan 算法求强连通分量

在 Tarjan 算法中为每个结点 u 维护了以下几个变量：

1. dfn_u：深度优先搜索遍历时结点 u 被搜索的次序。
2. low_u：在 u 的子树中能够回溯到的最早的已经在栈中的结点。设以 u 为根的子树为 $Subtree_u$。low_u 定义为以下结点的 dfn 的最小值：$Subtree_u$ 中的结点；从 $Subtree_u$ 通过一条不在搜索树上的边能到达的结点。

一个结点的子树内结点的 dfn 都大于该结点的 dfn。

从根开始的一条路径上的 dfn 严格递增，low 严格非降。

按照深度优先搜索算法搜索的次序对图中所有的结点进行搜索，维护每个结点的 `dfn` 与 `low` 变量，且让搜索到的结点入栈。每当找到一个强连通元素，就按照该元素包含结点数目让栈中元素出栈。在搜索过程中，对于结点 u 和与其相邻的结点 v（v 不是 u 的父节点）考虑 3 种情况：

1. v 未被访问：继续对 v 进行深度搜索。在回溯过程中，用 low_v 更新 low_u。因为存在从 u 到 v 的直接路径，所以 v 能够回溯到的已经在栈中的结点，u 也一定能够回溯到。
2. v 被访问过，已经在栈中：根据 low 值的定义，用 dfn_v 更新 low_u。
3. v 被访问过，已不在栈中：说明 v 已搜索完毕，其所在连通分量已被处理，所以不用对其做操作。

将上述算法写成伪代码：

" 实现"

```
TARJAN_SEARCH(int u)
    vis[u]=true
    low[u]=dfn[u]=++dfncnt
    push u to the stack
    for each (u,v) then do
        if v hasn't been searched then
            TARJAN_SEARCH(v) // 搜索
            low[u]=min(low[u],low[v]) // 回溯
        else if v has been in the stack then
            low[u]=min(low[u],dfn[v])
```

对于一个连通分量图，我们很容易想到，在该连通图中有且仅有一个 u 使得 $dfn_u = low_u$。该结点一定是在深度遍历的过程中，该连通分量中第一个被访问过的结点，因为它的 dfn 和 low 值最小，不会被该连通分量中的其他结点所影响。

因此，在回溯的过程中，判定 $dfn_u = low_u$ 是否成立，如果成立，则栈中 u 及其上方的结点构成一个 SCC。

实现

```cpp
int dfn[N], low[N], dfncnt, s[N], in_stack[N], tp;
int scc[N], sc;  // 结点 i 所在 SCC 的编号
int sz[N];       // 强连通 i 的大小

void tarjan(int u) {
  low[u] = dfn[u] = ++dfncnt, s[++tp] = u, in_stack[u] = 1;
  for (int i = h[u]; i; i = e[i].nex) {
    const int &v = e[i].t;
    if (!dfn[v]) {
      tarjan(v);
      low[u] = min(low[u], low[v]);
    } else if (in_stack[v]) {
      low[u] = min(low[u], dfn[v]);
    }
  }
  if (dfn[u] == low[u]) {
    ++sc;
    while (s[tp] != u) {
      scc[s[tp]] = sc;
      sz[sc]++;
      in_stack[s[tp]] = 0;
      --tp;
    }
    scc[s[tp]] = sc;
    sz[sc]++;
    in_stack[s[tp]] = 0;
    --tp;
  }
}
```

```python
dfn = [0] * N; low = [0] * N; dfncnt = 0; s = [0] * N; in_stack  = [0] * N; tp =
 0
scc = [0] * N; sc = 0 # 结点 i 所在 SCC 的编号
sz = [0] * N # 强连通 i 的大小
def tarjan(u):
    low[u] = dfn[u] = dfncnt; s[tp] = u; in_stack[u] = 1
    dfncnt = dfncnt + 1; tp = tp + 1
    i = h[u]
    while i:
        v = e[i].t
        if dfn[v] == False:
            tarjan(v)
            low[u] = min(low[u], low[v])
        elif in_stack[v]:
            low[u] = min(low[u], dfn[v])
        i = e[i].nex
    if dfn[u] == low[u]:
        sc = sc + 1
        while s[tp] != u:
            scc[s[tp]] = sc
            sz[sc] = sz[sc] + 1
            in_stack[s[tp]] = 0
```

```
            tp = tp - 1
        scc[s[tp]] = sc
        sz[sc] = sz[sc] + 1
        in_stack[s[tp]] = 0
        tp = tp - 1
```

时间复杂度 $O(n+m)$。

Kosaraju 算法

引入

Kosaraju 算法最早在 1978 年由 S. Rao Kosaraju 在一篇未发表的论文上提出，但 Micha Sharir 最早发表了它。

过程

该算法依靠两次简单的 DFS 实现：

第一次 DFS，选取任意顶点作为起点，遍历所有未访问过的顶点，并在回溯之前给顶点编号，也就是后序遍历。

第二次 DFS，对于反向后的图，以标号最大的顶点作为起点开始 DFS。这样遍历到的顶点集合就是一个强连通分量。对于所有未访问过的结点，选取标号最大的，重复上述过程。

两次 DFS 结束后，强连通分量就找出来了，Kosaraju 算法的时间复杂度为 $O(n+m)$。

实现

```cpp
// g 是原图，g2 是反图

void dfs1(int u) {
  vis[u] = true;
  for (int v : g[u])
    if (!vis[v]) dfs1(v);
  s.push_back(u);
}

void dfs2(int u) {
  color[u] = sccCnt;
  for (int v : g2[u])
    if (!color[v]) dfs2(v);
}

void kosaraju() {
  sccCnt = 0;
  for (int i = 1; i <= n; ++i)
    if (!vis[i]) dfs1(i);
  for (int i = n; i >= 1; --i)
    if (!color[s[i]]) {
      ++sccCnt;
      dfs2(s[i]);
    }
}
```

```python
def dfs1(u):
    vis[u] = True
    for v in g[u]:
        if vis[v] == False:
```

```python
            dfs1(v)
    s.append(u)

def dfs2(u):
    color[u] = sccCnt
    for v in g2[u]:
        if color[v] == False:
            dfs2(v)

def kosaraju(u):
    sccCnt = 0
    for i in range(1, n + 1):
        if vis[i] == False:
            dfs1(i)
    for i in range(n, 0, -1):
        if color[s[i]] == False:
            sccCnt = sccCnt + 1
            dfs2(s[i])
```

Garbow 算法

过程

Garbow 算法是 Tarjan 算法的另一种实现，Tarjan 算法是用 dfn 和 low 来计算强连通分量的根，Garbow 维护一个节点栈，并用第二个栈来确定何时从第一个栈中弹出属于同一个强连通分量的节点。从节点 w 开始的 DFS 过程中，当一条路径显示这组节点都属于同一个强连通分量时，只要栈顶节点的访问时间大于根节点 w 的访问时间，就从第二个栈中弹出这个节点，那么最后只留下根节点 w。在这个过程中每一个被弹出的节点都属于同一个强连通分量。

当回溯到某一个节点 w 时，如果这个节点在第二个栈的顶部，就说明这个节点是强连通分量的起始节点，在这个节点之后搜索到的那些节点都属于同一个强连通分量，于是从第一个栈中弹出那些节点，构成强连通分量。

实现

```cpp
int garbow(int u) {
  stack1[++p1] = u;
  stack2[++p2] = u;
  low[u] = ++dfs_clock;
  for (int i = head[u]; i; i = e[i].next) {
    int v = e[i].to;
    if (!low[v])
      garbow(v);
    else if (!sccno[v])
      while (low[stack2[p2]] > low[v]) p2--;
  }
  if (stack2[p2] == u) {
    p2--;
    scc_cnt++;
    do {
      sccno[stack1[p1]] = scc_cnt;
      // all_scc[scc_cnt] ++;
    } while (stack1[p1--] != u);
  }
  return 0;
}
```

```cpp
void find_scc(int n) {
  dfs_clock = scc_cnt = 0;
  p1 = p2 = 0;
  memset(sccno, 0, sizeof(sccno));
  memset(low, 0, sizeof(low));
  for (int i = 1; i <= n; i++)
    if (!low[i]) garbow(i);
}
```

```python
def garbow(u):
    stack1[p1] = u
    stack2[p2] = u
    p1 = p1 + 1; p2 = p2 + 1
    low[u] = dfs_clock
    dfs_clock = dfs_clock + 1
    i = head[u]
    while i:
        v = e[i].to
        if low[v] == False:
            garbow(v)
        elif sccno[v] == False:
            while low[stack2[p2]] > low[v]:
                p2 = p2 - 1
    if stack2[p2] == u:
        p2 = p2 - 1
        scc_cnt = scc_cnt + 1
        while stack1[p1] != u:
            p1 = p1 - 1
            sccno[stack1[p1]] = scc_cnt

def find_scc(n):
    dfs_clock = scc_cnt = 0
    p1 = p2 = 0
    sccno = []; low = []
    for i in range(1, n + 1):
        if low[i] == False:
            garbow(i)
```

应用

我们可以将一张图的每个强连通分量都缩成一个点。

然后这张图会变成一个 DAG，可以进行拓扑排序以及更多其他操作。

举个简单的例子，求一条路径，可以经过重复结点，要求经过的不同结点数量最多。

习题

USACO Fall/HAOI 2006 受欢迎的牛[1]

POJ1236 Network of Schools[2]

参考资料与注释

[1] USACO Fall/HAOI 2006 受欢迎的牛

[2] POJ1236 Network of Schools

11.19.2 双连通分量

简介

在阅读下列内容之前，请务必了解 图论相关概念 部分。

相关阅读：割点和桥

定义

割点和桥更严谨的定义参见 图论相关概念。

在一张连通的无向图中，对于两个点 u 和 v，如果无论删去哪条边（只能删去一条）都不能使它们不连通，我们就说 u 和 v **边双连通**。

在一张连通的无向图中，对于两个点 u 和 v，如果无论删去哪个点（只能删去一个，且不能删 u 和 v 自己）都不能使它们不连通，我们就说 u 和 v **点双连通**。

边双连通具有传递性，即，若 x, y 边双连通，y, z 边双连通，则 x, z 边双连通。

点双连通**不**具有传递性，反例如下图，A, B 点双连通，B, C 点双连通，而 A, C **不**点双连通。

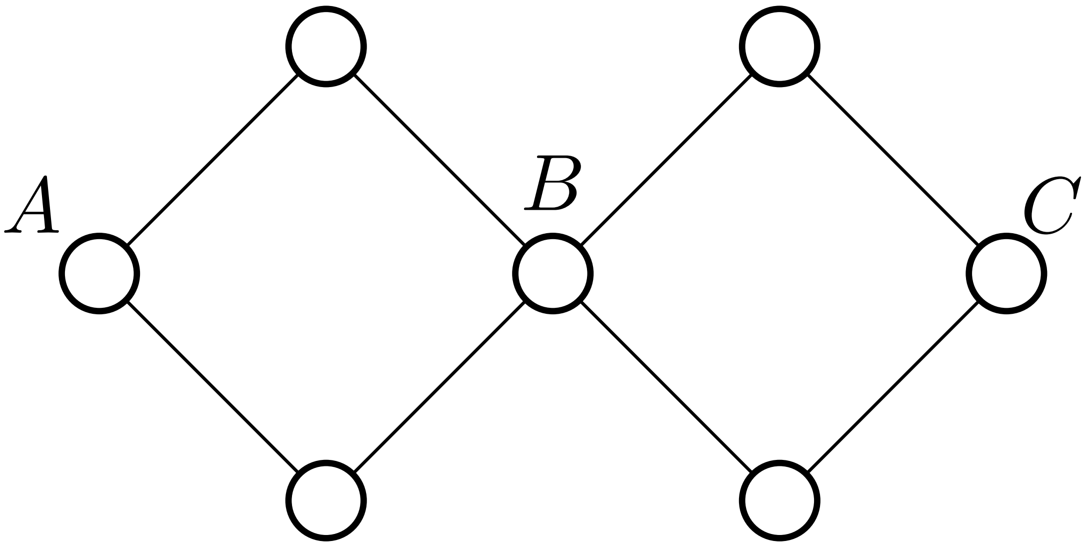


图 11.65 bcc-counterexample.png

DFS

对于一张连通的无向图，我们可以从任意一点开始 DFS，得到原图的一棵生成树（以开始 DFS 的那个点为根），这棵生成树上的边称作**树边**，不在生成树上的边称作**非树边**。

由于 DFS 的性质，我们可以保证所有非树边连接的两个点在生成树上都满足其中一个是另一个的祖先。

DFS 的代码如下：

" 实现"

```
void DFS(int p) {
  visited[p] = true;
  for (int to : edge[p])
    if (!visited[to]) DFS(to);
```

```
    }

def DFS(p):
    visited[p] = True
    for to in edge[p]:
        if visited[to] == False:
            DFS(to)
```

过程

DFS 找桥并判断边双连通

首先，对原图进行 DFS。

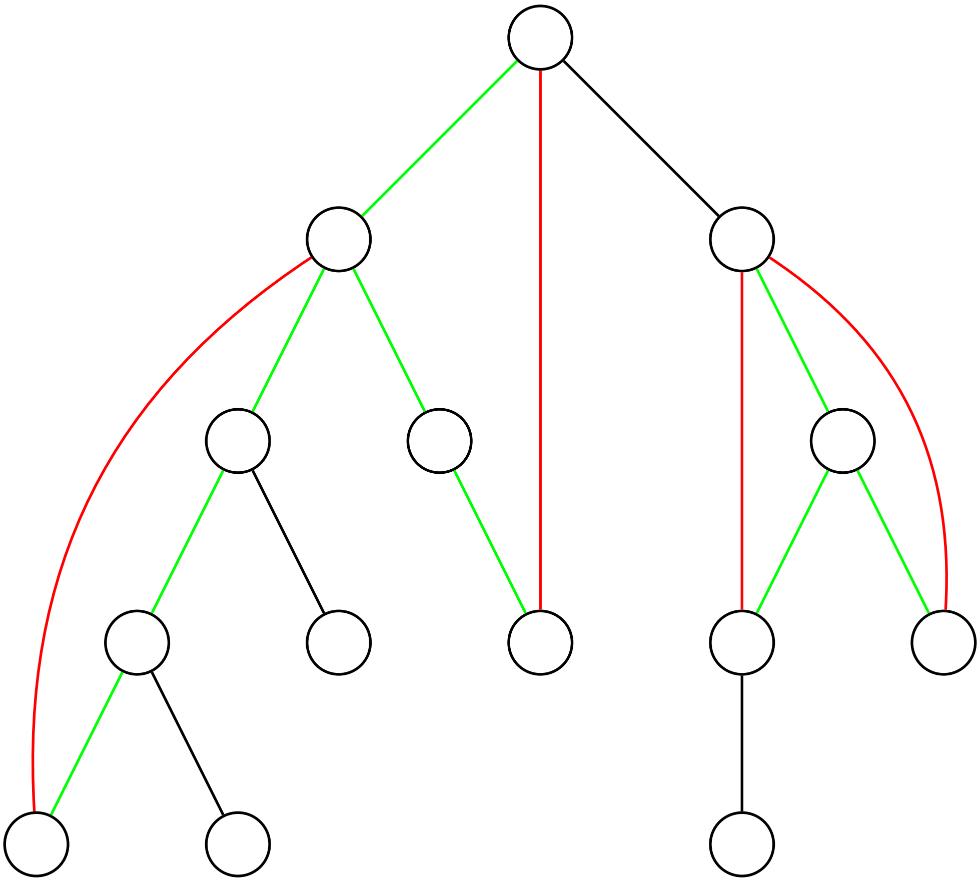


图 11.66 bcc-1.png

如上图所示，黑色与绿色边为树边，红色边为非树边。每一条非树边连接的两个点都对应了树上的一条简单路径，我们说这条非树边**覆盖**了这条树上路径上所有的边。绿色的树边**至少**被一条非树边覆盖，黑色的树边不被**任何**非树边覆盖。

我们如何判断一条边是不是桥呢？显然，非树边和绿色的树边一定不是桥，黑色的树边一定是桥。

如何用算法去实现以上过程呢？首先有一个比较暴力的做法，对于每一条非树边，都逐个地将它覆盖的每一条树边置成绿色，这样的时间复杂度为 $O(nm)$。

怎么优化呢？可以用差分。对于每一条非树边，在其树上深度较小的点处打上 `-1` 标记，在其树上深度较大的点处打上 `+1` 标记。然后 $O(n)$ 求出每个点的子树内部的标记之和。对于一个点 u，其子树内部的标记之和等于覆盖了 u 和 u 的父亲之间的树边的非树边数量。若这个值非 0，则 u 和 u 的父亲之间的树边不是桥，否则是桥。

用以上的方法 $O(n+m)$ 求出每条边分别是否是桥后，两个点是边双连通的，当且仅当它们的树上路径中**不**包含桥。

DFS 找割点并判断点双连通

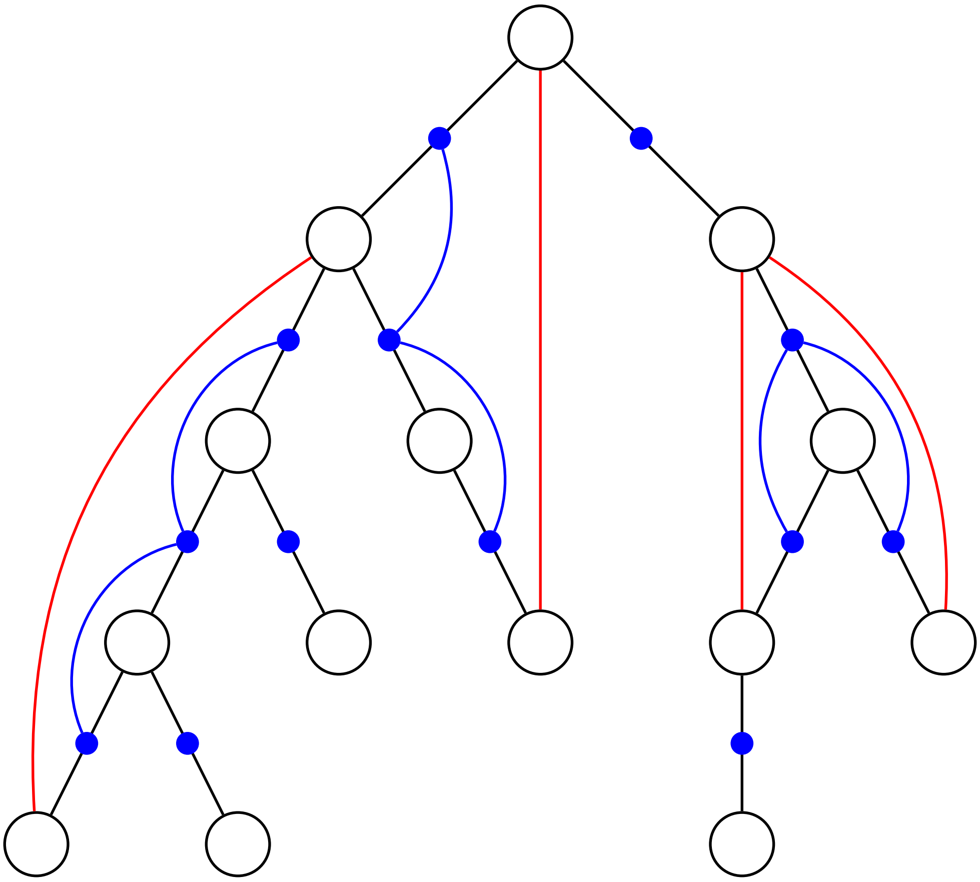


图 11.67 bcc-2.png

如上图所示，黑色边为树边，红色边为非树边。每一条非树边连接的两个点都对应了树上的一条简单路径。

考虑一张新图，新图中的每一个点对应原图中的每一条树边（在上图中用蓝色点表示）。对于原图中的每一条非树边，将这条非树边对应的树上简单路径中的所有边在新图中对应的蓝点连成一个连通块（这在上图中也用蓝色的边体现出来了）。

这样，一个点不是割点，当且仅当与其相连的所有边在新图中对应的蓝点都属于同一个连通块。两个点点双连通，当且仅当它们在原图的树上路径中的所有边在新图中对应的蓝点都属于同一个连通块。

蓝点间的连通关系可以用与求边双连通时用到的差分类似的方法维护，时间复杂度 $O(n+m)$。

11.19.3 割点和桥

Authors: Ir1d, sshwy, GavinZhengOI, Planet6174, ouuan, Marcythm, ylxmf2005, 0xis-cn

相关阅读：双连通分量，

割点和桥更严谨的定义参见 图论相关概念。

割点

> 对于一个无向图，如果把一个点删除后这个图的极大连通分量数增加了，那么这个点就是这个图的割点（又称割顶）。

过程

如果我们尝试删除每个点，并且判断这个图的连通性，那么复杂度会特别的高。所以要介绍一个常用的算法：Tarjan。

首先，我们上一个图：

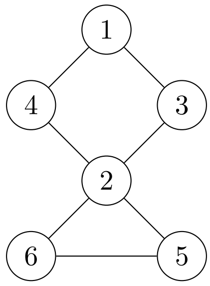


图 11.68

很容易的看出割点是 2，而且这个图仅有这一个割点。

首先，我们按照 DFS 序给他打上时间戳（访问的顺序）。

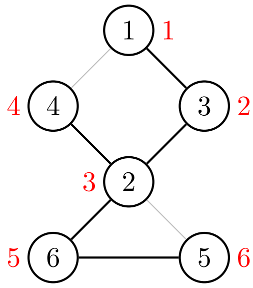


图 11.69

这些信息被我们保存在一个叫做 `dfn` 的数组中。

还需要另外一个数组 `low`，用它来存储不经过其父亲能到达的最小的时间戳。

例如 `low[2]` 的话是 1，`low[5]` 和 `low[6]` 是 3。

然后我们开始 DFS，我们判断某个点是否是割点的根据是：对于某个顶点 u，如果存在至少一个顶点 v（u 的儿子），使得 $low_v \geq dfn_u$，即不能回到祖先，那么 u 点为割点。

此根据惟独不适用于搜索的起始点，其需要特殊考虑：若该点不是割点，则其他路径亦能到达全部结点，因此从起始点只「向下搜了一次」，即在搜索树内仅有一个子结点。如果在搜索树内有两个及以上的儿子，那么他一定是割点了（设想上图从 2 开始搜索，搜索树内应有两个子结点：3 或 4 及 5 或 6）。如果只有一个儿子，那么把它删掉，不会有任何的影响。比如下面这个图，此处形成了一个环。

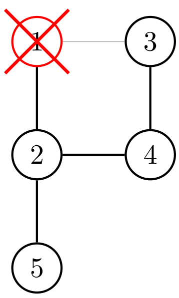


图 11.70

我们在访问 1 的儿子时候，假设先 DFS 到了 2，然后标记用过，然后递归往下，来到了 4，4 又来到了 3，当递归回溯的时候，会发现 3 已经被访问过了，所以不是割点。

更新 low 的伪代码如下：

```
如果 v 是 u 的儿子 low[u] = min(low[u], low[v]);
否则
low[u] = min(low[u], dfn[v]);
```

例题

洛谷 P3388【模板】割点（割顶）[1]

" 例题代码"

```
/*
洛谷 P3388 【模板】割点（割顶）
*/
#include <bits/stdc++.h>
using namespace std;
int n, m;  // n：点数 m：边数
int dfn[100001], low[100001], inde, res;
// dfn：记录每个点的时间戳
// low：能不经过父亲到达最小的编号，inde：时间戳，res：答案数量
bool vis[100001], flag[100001];  // flag: 答案 vis：标记是否重复
vector<int> edge[100001];        // 存图用的

void Tarjan(int u, int father) {  // u 当前点的编号，father 自己爸爸的编号
  vis[u] = true;                  // 标记
  low[u] = dfn[u] = ++inde;  // 打上时间戳
  int child = 0;             // 每一个点儿子数量
  for (auto v : edge[u]) {   // 访问这个点的所有邻居 （C++11）

    if (!vis[v]) {
      child++;                       // 多了一个儿子
      Tarjan(v, u);                  // 继续
      low[u] = min(low[u], low[v]);  // 更新能到的最小节点编号
      if (father != u && low[v] >= dfn[u] &&
          !flag
              [u])  // 主要代码
                    // 如果不是自己，且不通过父亲返回的最小点符合割点的要求，并且没
有被标记过
                    // 要求即为：删了父亲连不上去了，即为最多连到父亲
      {
        flag[u] = true;
        res++;  // 记录答案
      }
    } else if (v != father)
      low[u] =
          min(low[u], dfn[v]);  // 如果这个点不是自己，更新能到的最小节点编号
  }
  if (father == u && child >= 2 &&
      !flag[u]) {  // 主要代码，自己的话需要 2 个儿子才可以
    flag[u] = true;
    res++;  // 记录答案
  }
}
```

```
int main() {
  cin >> n >> m;                  // 读入数据
  for (int i = 1; i <= m; i++) {  // 注意点是从 1 开始的
    int x, y;
    cin >> x >> y;
    edge[x].push_back(y);
    edge[y].push_back(x);
  }                               // 使用 vector 存图
  for (int i = 1; i <= n; i++)    // 因为 Tarjan 图不一定连通
    if (!vis[i]) {
      inde = 0;        // 时间戳初始为 0
      Tarjan(i, i);    // 从第 i 个点开始，父亲为自己
    }
  cout << res << endl;
  for (int i = 1; i <= n; i++)
    if (flag[i]) cout << i << " ";  // 输出结果
  return 0;
}
```

割边

和割点差不多，叫做桥。

> 对于一个无向图，如果删掉一条边后图中的连通分量数增加了，则称这条边为桥或者割边。严谨来说，就是：假设有连通图 $G=\{V,E\}$，e 是其中一条边（即 $e\in E$），如果 $G-e$ 是不连通的，则边 e 是图 G 的一条割边（桥）。

比如说，下图中，

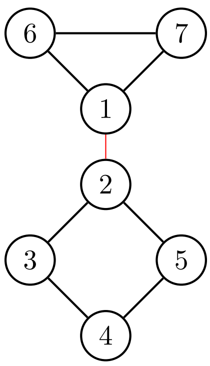


图 11.71 割边示例图

红色的边就是割边。

过程

和割点差不多，只要改一处：$low_v>dfn_u$ 就可以了，而且不需要考虑根节点的问题。

割边是和是不是根节点没关系的，原来我们求割点的时候是指点 v 是不可能不经过父节点 u 为回到祖先节点（包括父节点），所以顶点 u 是割点。如果 $low_v=dfn_u$ 表示还可以回到父节点，如果顶点 v 不能回到祖先也没有另外一条回到父亲的路，那么 $u-v$ 这条边就是割边。

实现

下面代码实现了求割边，其中，当 `isbridge[x]` 为真时，`(father[x],x)` 为一条割边。

```cpp
int low[MAXN], dfn[MAXN], dfs_clock;
bool isbridge[MAXN];
vector<int> G[MAXN];
int cnt_bridge;
int father[MAXN];

void tarjan(int u, int fa) {
  father[u] = fa;
  low[u] = dfn[u] = ++dfs_clock;
  for (int i = 0; i < G[u].size(); i++) {
    int v = G[u][i];
    if (!dfn[v]) {
      tarjan(v, u);
      low[u] = min(low[u], low[v]);
      if (low[v] > dfn[u]) {
        isbridge[v] = true;
        ++cnt_bridge;
      }
    } else if (dfn[v] < dfn[u] && v != fa) {
      low[u] = min(low[u], dfn[v]);
    }
  }
}
```

```python
low = [0] * MAXN; dfn = [0] * MAXN; dfs_clock = 0
isbridge = [False] * MAXN
G = [[0 for i in range(MAXN)] for j in range(MAXN)]
cnt_bridge = 0
father = [0] * MAXN

def tarjan(u, fa):
    father[u] = fa
    low[u] = dfn[u] = dfs_clock
    dfs_clock = dfs_clock + 1
    for i in range(0, len(G[u])):
        v = G[u][i]
        if dfn[v] == False:
            tarjan(v, u)
            low[u] = min(low[u], low[v])
            if low[v] > dfn[u]:
                isbridge[v] = True
                cnt_bridge = cnt_bridge + 1
        elif dfn[v] < dfn[u] and v != fa:
            low[u] = min(low[u], dfn[v])
```

练习

- P3388【模板】割点（割顶）[1]
- POJ2117 Electricity[2]
- HDU4738 Caocao's Bridges[3]
- HDU2460 Network[4]
- POJ1523 SPF[5]

Tarjan 算法还有许多用途，常用的例如求强连通分量，缩点，还有求 2-SAT 的用途等。

参考资料与注释

[1] 洛谷 P3388【模板】割点（割顶） [1-1] [1-2]

[2] POJ2117 Electricity

[3] HDU4738 Caocao's Bridges

[4] HDU2460 Network

[5] POJ1523 SPF

11.19.4 圆方树

Authors: GitPinkRabbit, Early0v0, Backl1ght, mcendu, ksyx, iamtwz, Xeonacid, kenlig, Menci, Enter-tainer, CCXXXI

在阅读下列内容之前，请务必了解 图论相关概念 部分。

相关阅读：割点和桥。

引入

众所周知，树（或森林）有很好的性质，并且容易通过很多常见数据结构维护。

而一般图则没有那么好的性质，所幸有时我们可以把一般图上的某些问题转化到树上考虑。

而圆方树（Block forest 或 Round-square tree）[1] 就是一种将图变成树的方法。本文将介绍圆方树的构建，性质和一些应用。

限于篇幅，本文中有一些结论未经证明，读者可以自行理解或证明。

定义

圆方树最初是处理「仙人掌图」（每条边在不超过一个简单环中的无向图）的一种工具，不过发掘它的更多性质，有时我们可以在一般无向图上使用它。

要介绍圆方树，首先要介绍**点双连通分量**。

一个**点双连通图**的一个定义是：图中任意两不同点之间都有至少两条点不重复的路径。

点不重复既指路径上点不重复（简单路径），也指两条路径的交集为空（当然，路径必然都经过出发点和到达点，这不在考虑范围内）。

可以发现对于只有一个点的图比较难定义它是不是一个点双，这里先不考虑节点数为 1 的图。

一个近乎等价的定义是：不存在割点的图。

这个定义只在图中只有两个点，一条连接它们的边时失效。它没有割点，但是并不能找到两条不相交的路径，因为只有一条路径。

（也可以理解为那一条路径可以算两次，的确没有交，因为不经过其他点）

虽然原始的定义的确是前者，但是为了方便，我们规定点双图的定义采用后者。

而一个图的**点双连通分量**则是一个**极大点双连通子图**。

与强连通分量等不同，一个点可能属于多个点双，但是一条边属于恰好一个点双（如果定义采用前者则有可能不属于任何点双）。

在圆方树中，原来的每个点对应一个**圆点**，每一个点双对应一个**方点**。

所以共有 $n+c$ 个点，其中 n 是原图点数，c 是原图点双连通分量的个数。

而对于每一个点双连通分量，它对应的方点向这个点双连通分量中的每个点连边。

每个点双形成一个「菊花图」，多个「菊花图」通过原图中的割点连接在一起（因为点双的分隔点是割点）。

显然，圆方树中每条边连接一个圆点和一个方点。

下面的图显示了一张图对应的点双和圆方树形态。[2]

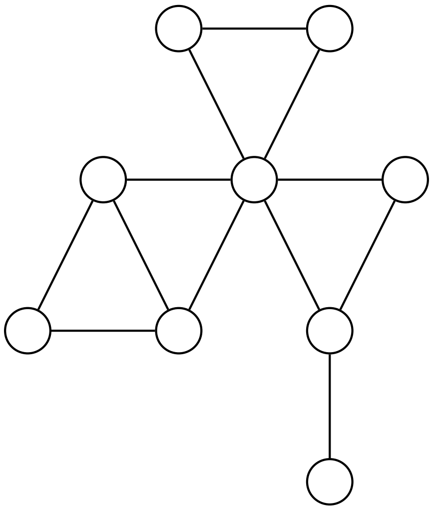


图 11.72

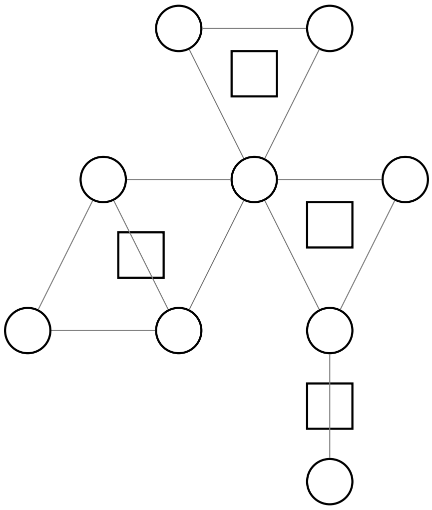


图 11.73

圆方树的点数小于 $2n$，这是因为割点的数量小于 n，所以请注意各种数组大小要开两倍。

其实，如果原图连通，则「圆方树」才是一棵树，如果原图有 k 个连通分量，则它的圆方树也会形成 k 棵树形成的森林。

如果原图中某个连通分量只有一个点，则需要具体情况具体分析，我们在后续讨论中不考虑孤立点。

过程

对于一个图，如何构造出它的圆方树呢？首先可以发现如果图不连通，可以拆分成每个连通子图考虑，所以我们只考虑连通图。

因为圆方树是基于点双连通分量的，而点双连通分量又基于割点，所以只需要用类似求割点的方法即可。

求割点的常用算法是 Tarjan 算法，如果你会了理解下面的内容就很简单了，如果你不会也没关系。

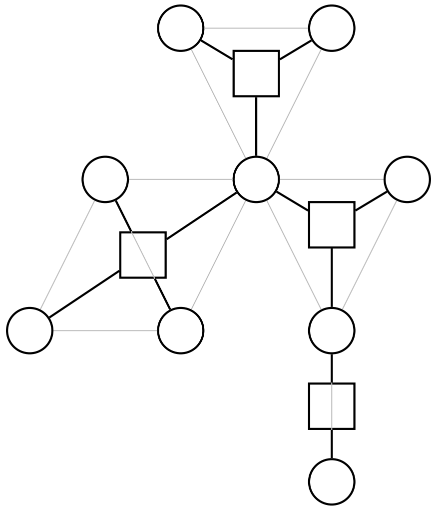


图 11.74

我们跳过 Tarjan 求割点，直接介绍圆方树使用的算法（其实是 Tarjan 的变体）：

对图进行 DFS，并且中间用到了两个关键数组 `dfn` 和 `low`（类似于 Tarjan）。

`dfn[u]` 存储的是节点 u 的 DFS 序，即第一次访问到 u 时它是第几个被访问的节点。

`low[u]` 存储的是节点 u 的 DFS 树中的子树中的某个点 v 通过**最多一次返祖边或向父亲的树边**能访问到的点的**最小** DFS 序。

如果没有听说过 Tarjan 算法可能会有点难理解，让我们举个例子吧：

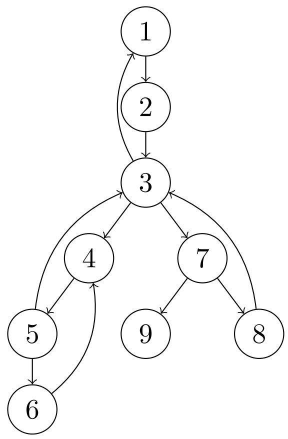


图 11.75

（可以发现这张图其实和上面图片中的图等价）

这里树边从上至下用直线画出，返祖边从下至上用曲线画出。节点的编号便是它的 DFS 序。

则有 `low` 数组如下：

i	1	2	3	4	5	6	7	8	9
low[i]	1	1	1	3	3	4	3	3	7

并不是很难理解吧，注意这里 9 的 `low` 是 7，与一些求割点的做法有差异，因为为了方便，我们规定了可以通过父边向上，但主要思想是相同的。

我们可以很容易地写出计算 `dfn` 和 `low` 的 DFS 函数（初始时 `dfn` 数组清零）：

" 实现"

```
void Tarjan(int u) {
  low[u] = dfn[u] = ++dfc;              // low 初始化为当前节点 dfn
  for (int v : G[u]) {                  // 遍历 u 的相邻节点
    if (!dfn[v]) {                      // 如果未访问过
```

```
            Tarjan(v) # 递归
            low[u] = min(low[u], low[v]) # 未访问的和 low 取 min
        else:
            low[u] = min(low[u], dfn[v]) # 已访问的和 dfn 取 min
```

接下来，我们考虑点双和 DFS 树以及这两个数组之间的关联。

可以发现，每个点双在 DFS 树上是一棵连通子树，并至少包含两个点；特别地，最顶端节点仅往下接一个点。

同时还可以发现每条树边恰好在一个点双内。

我们考虑一个点双在 DFS 树中的最顶端节点 u，在 u 处确定这个点双，因为 u 的子树包含了整个点双的信息。

因为至少有两个点，考虑这个点双的下一个点 v，则有 u，v 之间存在一条树边。

不难发现，此时一定有 $\mathrm{low}[v] = \mathrm{dfn}[u]$。

更准确地说，对于一条树边 $u \to v$，u, v 在同一个点双中，且 u 是这个点双中深度最浅的节点**当且仅当** $\mathrm{low}[v] = \mathrm{dfn}[u]$。

那么我们可以在 DFS 的过程中确定哪些地方存在点双，但是还不能准确确定一个点双所包含的点集。

这并不难处理，我们可以在 DFS 过程中维护一个栈，存储还未确定所属点双（可能有多个）的节点。

在找到点双时，点双中除了 u 以外的其他的点都集中在栈顶端，只需要不断弹栈直到弹出 v 为止即可。

当然，我们可以同时处理被弹出的节点，只要将其和新建的方点连边即可。最后还要让 u 和方点连边。

这样就很自然地完成了圆方树的构建，我们可以给方点标号为 $n+1$ 开始的整数，这样可以有效区分圆点和方点。

这部分可能讲述得不够清晰，下面贴出一份代码，附有详尽注释以及帮助理解的输出语句和一份样例，建议读者复制代码并自行实践理解，毕竟代码才是最能帮助理解的（不要忘记开 `c++11`）。

" 实现"

```cpp
#include <algorithm>
#include <cstdio>
#include <vector>

const int MN = 100005;

int N, M, cnt;
std::vector<int> G[MN], T[MN * 2];

int dfn[MN], low[MN], dfc;
int stk[MN], tp;

void Tarjan(int u) {
  printf("  Enter : #%d\n", u);
  low[u] = dfn[u] = ++dfc;                // low 初始化为当前节点 dfn
  stk[++tp] = u;                          // 加入栈中
  for (int v : G[u]) {                    // 遍历 u 的相邻节点
    if (!dfn[v]) {                        // 如果未访问过
      Tarjan(v);                          // 递归
      low[u] = std::min(low[u], low[v]);  // 未访问的和 low 取 min
      if (low[v] == dfn[u]) {  // 标志着找到一个以 u 为根的点双连通分量
        ++cnt;                 // 增加方点个数
        printf("  Found a New BCC #%d.\n", cnt - N);
        // 将点双中除了 u 的点退栈，并在圆方树中连边
        for (int x = 0; x != v; --tp) {
          x = stk[tp];
          T[cnt].push_back(x);
          T[x].push_back(cnt);
```

```
          printf("    BCC #%d has vertex #%d\n", cnt - N, x);
        }
        // 注意 u 自身也要连边 (但不退栈)
        T[cnt].push_back(u);
        T[u].push_back(cnt);
        printf("    BCC #%d has vertex #%d\n", cnt - N, u);
      }
    } else
      low[u] = std::min(low[u], dfn[v]);  // 已访问的和 dfn 取 min
  }
  printf("  Exit : #%d : low = %d\n", u, low[u]);
  printf("  Stack:\n    ");
  for (int i = 1; i <= tp; ++i) printf("%d, ", stk[i]);
  puts("");
}

int main() {
  scanf("%d%d", &N, &M);
  cnt = N;  // 点双 / 方点标号从 N 开始
  for (int i = 1; i <= M; ++i) {
    int u, v;
    scanf("%d%d", &u, &v);
    G[u].push_back(v);  // 加双向边
    G[v].push_back(u);
  }
  // 处理非连通图
  for (int u = 1; u <= N; ++u)
    if (!dfn[u]) Tarjan(u), --tp;
  // 注意到退出 Tarjan 时栈中还有一个元素即根，将其退栈
  return 0;
}
```

提供一个测试用例:

```
13 15
1 2
2 3
1 3
3 4
3 5
4 5
5 6
4 6
3 7
3 8
7 8
7 9
10 11
11 10
11 12
```

这个例子对应的图（包含了重边和孤立点的情况）:

例题

我们讲一些可以使用圆方树求解的例题。

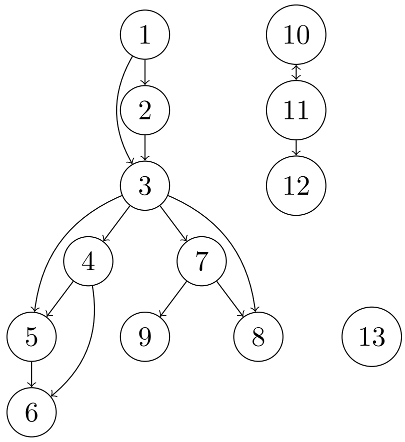


图 11.76

"「APIO2018」 铁人两项[3]"

" 题意简述"

给定一张简单无向图，问有多少对三元组 $\langle s,c,f\rangle$（s,c,f 互不相同）使得存在一条简单路径从 s 出发，经过 c 到达 f。

" 题解"

说到简单路径，就必须提一个关于点双很好的性质：对于一个点双中的两点，它们之间简单路径的并集，恰好完全等于这个点双。

即同一个点双中的两不同点 u,v 之间一定存在一条简单路径经过给定的在同一个点双内的另一点 w。

这个性质的证明：

- 显然如果简单路径出了点双，就不可能再回到这个点双中，否则会和点双的定义冲突。
- 所以我们只需考虑证明一个点双连通图中任意三不同点 u,v,c，必存在一条从 u 到 v 的简单路径经过 c。
- 首先排除点数为 2 的情况，它满足这个性质，但是无法取出 3 个不同点。
- 对于余下的情况，考虑建立网络流模型，源点向 c 连容量为 2 的边，u 和 v 向汇点连容量为 1 的边。
- 原图中的双向边 $\langle x,y\rangle$，变成 x 向 y 连一条容量为 1 的边，y 也向 x 连一条容量为 1 的边。
- 最后，给除了源点，汇点和 c 之外的每个点赋上 1 的容量，这可以通过拆点实现。
- 因为源点到 c 的边的容量为 2，那么如果这个网络最大流为 2，则证明一定有路径经过 c。
- 考虑最大流最小割定理，显然最小割小于等于 2，接下来只要证最小割大于 1。
- 这等价于证明割掉任意一条容量为 1 的边，是无法使源点和汇点不连通的。
- 考虑割掉 u 或 v 与汇点连接的点，根据点双的第一种定义，必然存在简单路径从 c 到另一个没割掉的点。
- 考虑割掉一个节点拆点形成的边，这等价于删除一个点，根据点双的第二种定义，余下的图仍然连通。
- 考虑割掉一条由原先的边建出的边，这等价于删除一条边，这比删除一个点更弱，显然存在路径。
- 所以我们证明了最小割大于 1，即最大流等于 2。证毕。

这个结论能告诉我们什么呢？它告诉了我们：考虑两圆点在圆方树上的路径，与路径上经过的方点相邻的圆点的集合，就等于原图中两点简单路径上的点集。

回到题目，考虑固定 s 和 f，求合法的 c 的数量，显然有合法 c 的数量等于 s,f 之间简单路径的并集的点数减 2（去掉 s,f 本身）。

那么，对原图建出圆方树后，两点之间简单路径的点数，就和它们在圆方树上路径经过的方点（点双）和圆点的个数有关。

接下来是圆方树的一个常用技巧：路径统计时，点赋上合适的权值。

本题中，每个方点的权值为对应点双的大小，而每个圆点权值为 -1。

这样赋权后则有两圆点间圆方树上路径点权和，恰好等于原图中简单路径并集大小减 2。

问题转化为统计圆方树上 $\sum$ 两圆点路径权值和。

换个角度考虑，改为统计每一个点对答案的贡献，即权值乘以经过它的路径条数，这可以通过简单的树形 DP 求出。

最后，不要忘记处理图不连通的情况。下面是对应代码：

" 参考代码"

```
#include <algorithm>
#include <cstdio>
#include <vector>

const int MN = 100005;

int N, M, cnt;
std::vector<int> G[MN], T[MN * 2];
long long Ans;

int dfn[MN], low[MN], dfc, num;
int stk[MN], tp;

int wgh[MN * 2];

void Tarjan(int u) {  // 求点双
  low[u] = dfn[u] = ++dfc;
  stk[++tp] = u;
  ++num;
  for (int v : G[u]) {
    if (!dfn[v]) {
      Tarjan(v);
      low[u] = std::min(low[u], low[v]);
      if (low[v] == dfn[u]) {
        wgh[++cnt] = 0;
        for (int x = 0; x != v; --tp) {
          x = stk[tp];
          T[cnt].push_back(x);
          T[x].push_back(cnt);
          ++wgh[cnt];
        }
        T[cnt].push_back(u);
        T[u].push_back(cnt);
        ++wgh[cnt];
      }
    } else
      low[u] = std::min(low[u], dfn[v]);
  }
}

int vis[MN * 2], siz[MN * 2];
```

```
void DFS(int u, int fz) {  // dfs 求值
  vis[u] = 1;
  siz[u] = (u <= N);
  for (int v : T[u])
    if (v != fz) {
      DFS(v, u);
      Ans += 2ll * wgh[u] * siz[u] * siz[v];
      siz[u] += siz[v];
    }
  Ans += 2ll * wgh[u] * siz[u] * (num - siz[u]);
}

int main() {
  scanf("%d%d", &N, &M);
  for (int u = 1; u <= N; ++u) wgh[u] = -1;
  cnt = N;
  for (int i = 1; i <= M; ++i) {
    int u, v;
    scanf("%d%d", &u, &v);
    G[u].push_back(v);
    G[v].push_back(u);
  }
  for (int u = 1; u <= N; ++u)
    if (!dfn[u]) {
      num = 0;
      Tarjan(u), --tp;
      DFS(u, 0);
    }
  printf("%lld\n", Ans);
  return 0;
}
```

顺带一提，刚刚的测试用例在这题的答案是 212。

"Codeforces #487 E. Tourists[4]"

" 题意简述"

给定一张简单无向连通图，要求支持两种操作：

1. 修改一个点的点权。
2. 询问两点之间所有简单路径上点权的最小值。

" 题解"

同样地，我们建出原图的圆方树，令方点权值为相邻圆点权值的最小值，问题转化为求路径上最小值。

路径最小值可以使用树链剖分和线段树维护，但是修改呢？

一次修改一个圆点的点权，需要修改所有和它相邻的方点，这样很容易被卡到 $\mathcal{O}(n)$ 个修改。

这时我们利用圆方树是棵树的性质，令方点权值为自己的儿子圆点的权值最小值，这样的话修改时只需要修改父亲方点。

对于方点的维护，只需要对每个方点开一个 `multiset` 维护权值集合即可。

需要注意的是查询时若 LCA 是方点，则还需要查 LCA 的父亲圆点的权值。

注意：圆方树点数要开原图的两倍，否则会数组越界。

" 参考代码"

```cpp
#include <algorithm>
#include <cstdio>
#include <set>
#include <vector>

const int MN = 100005;
const int MS = 524288;
const int Inf = 0x7fffffff;

int N, M, Q, cnt;
int w[MN * 2];
std::vector<int> G[MN], T[MN * 2];
std::multiset<int> S[MN * 2];

int dfn[MN * 2], low[MN], dfc;
int stk[MN], tp;

void Tarjan(int u) {
  low[u] = dfn[u] = ++dfc;
  stk[++tp] = u;
  for (int v : G[u]) {
    if (!dfn[v]) {
      Tarjan(v);
      low[u] = std::min(low[u], low[v]);
      if (low[v] == dfn[u]) {
        ++cnt;
        for (int x = 0; x != v; --tp) {
          x = stk[tp];
          T[cnt].push_back(x);
          T[x].push_back(cnt);
        }
        T[cnt].push_back(u);
        T[u].push_back(cnt);
      }
    } else
      low[u] = std::min(low[u], dfn[v]);
  }
}

int idf[MN * 2], faz[MN * 2], siz[MN * 2], dep[MN * 2], son[MN * 2],
    top[MN * 2];

void DFS0(int u, int fz) {
  faz[u] = fz, dep[u] = dep[fz] + 1, siz[u] = 1;
  for (int v : T[u])
    if (v != fz) {
      DFS0(v, u);
      siz[u] += siz[v];
      if (siz[son[u]] < siz[v]) son[u] = v;
    }
}
```

```
void DFS1(int u, int fz, int tp) {
  dfn[u] = ++dfc, idf[dfc] = u, top[u] = tp;
  if (son[u]) DFS1(son[u], u, tp);
  for (int v : T[u])
    if (v != fz && v != son[u]) DFS1(v, u, v);
}

#define li (i << 1)
#define ri (i << 1 | 1)
#define mid ((l + r) >> 1)
#define ls li, l, mid
#define rs ri, mid + 1, r

int dat[MS];

void Build(int i, int l, int r) {  // 建树
  if (l == r) {
    dat[i] = w[idf[l]];
    return;
  }
  Build(ls), Build(rs);
  dat[i] = std::min(dat[li], dat[ri]);
}

void Mdf(int i, int l, int r, int p, int x) {  // 获取最小值
  if (l == r) {
    dat[i] = x;
    return;
  }
  if (p <= mid)
    Mdf(ls, p, x);
  else
    Mdf(rs, p, x);
  dat[i] = std::min(dat[li], dat[ri]);
}

int Qur(int i, int l, int r, int a, int b) {  // 查询
  if (r < a || b < l) return Inf;
  if (a <= l && r <= b) return dat[i];
  return std::min(Qur(ls, a, b), Qur(rs, a, b));
}

int main() {
  scanf("%d%d%d", &N, &M, &Q);
  for (int i = 1; i <= N; ++i) scanf("%d", &w[i]);
  cnt = N;
  for (int i = 1; i <= M; ++i) {
    int u, v;
    scanf("%d%d", &u, &v);
    G[u].push_back(v);
    G[v].push_back(u);
  }
  Tarjan(1), DFS0(1, 0), dfc = 0, DFS1(1, 0, 1);
  for (int i = 1; i <= N; ++i)
```

```
    if (faz[i]) S[faz[i]].insert(w[i]);
  for (int i = N + 1; i <= cnt; ++i) w[i] = *S[i].begin();
  Build(1, 1, cnt);
  for (int q = 1; q <= Q; ++q) {
    char opt[3];
    int x, y;
    scanf("%s%d%d", opt, &x, &y);
    if (*opt == 'C') {
      Mdf(1, 1, cnt, dfn[x], y);
      if (faz[x]) {
        int u = faz[x];
        S[u].erase(S[u].lower_bound(w[x]));
        S[u].insert(y);
        if (w[u] != *S[u].begin()) {
          w[u] = *S[u].begin();
          Mdf(1, 1, cnt, dfn[u], w[u]);
        }
      }
      w[x] = y;
    } else {
      int Ans = Inf;
      while (top[x] != top[y]) {
        if (dep[top[x]] < dep[top[y]]) std::swap(x, y);
        Ans = std::min(Ans, Qur(1, 1, cnt, dfn[top[x]], dfn[x]));
        x = faz[top[x]];
      }
      if (dfn[x] > dfn[y]) std::swap(x, y);
      Ans = std::min(Ans, Qur(1, 1, cnt, dfn[x], dfn[y]));
      if (x > N) Ans = std::min(Ans, w[faz[x]]);
      printf("%d\n", Ans);
    }
  }
  return 0;
}
```

"「SDOI2018」 战略游戏[5]"

" 题意简述"

给出一个简单无向连通图。有 q 次询问：

每次给出一个点集 S（$2 \le |S| \le n$），问有多少个点 u 满足 $u \notin S$ 且删掉 u 之后 S 中的点不全在一个连通分量中。

每个测试点有多组数据。

" 题解"

先建出圆方树，则变为询问 S 在圆方树上对应的连通子图中的圆点个数减去 $|S|$。

如何计算连通子图中的圆点个数？有一个方法：

把圆点的权值放到它和它的父亲方点的边上，问题转化为求边权和，这个问题可以参考「SDOI2015」寻宝游戏[6] 的一种解法。

即把 S 中的点按照 DFS 序排序，计算排序后相邻两点的距离和（还包括首尾两点之间的距离），答案就是距离和的一半，因为每条边只被经过两次。

最后，如果子图中的深度最浅的节点是圆点，答案还要加上 1，因为我们没有统计到它。

因为有多组数据，要注意初始化数组。

" 参考代码"

```
#include <algorithm>
#include <cstdio>
#include <vector>

const int MN = 100005;

int N, M, Q, cnt;
std::vector<int> G[MN], T[MN * 2];

int dfn[MN * 2], low[MN], dfc;
int stk[MN], tp;

void Tarjan(int u) {  // 求点双，准备建树
  low[u] = dfn[u] = ++dfc;
  stk[++tp] = u;
  for (int v : G[u]) {
    if (!dfn[v]) {
      Tarjan(v);
      low[u] = std::min(low[u], low[v]);
      if (low[v] == dfn[u]) {
        ++cnt;
        for (int x = 0; x != v; --tp) {
          x = stk[tp];
          T[cnt].push_back(x);
          T[x].push_back(cnt);
        }
        T[cnt].push_back(u);
        T[u].push_back(cnt);
      }
    } else
      low[u] = std::min(low[u], dfn[v]);
  }
}

int dep[MN * 2], faz[MN * 2][18], dis[MN * 2];

void DFS(int u, int fz) {
  dfn[u] = ++dfc;
  dep[u] = dep[faz[u][0] = fz] + 1;
  dis[u] = dis[fz] + (u <= N);
  for (int j = 0; j < 17; ++j) faz[u][j + 1] = faz[faz[u][j]][j];
  for (int v : T[u])
    if (v != fz) DFS(v, u);
}

int LCA(int x, int y) {  // 最近公共祖先
  if (dep[x] < dep[y]) std::swap(x, y);
  for (int j = 0, d = dep[x] - dep[y]; d; ++j, d >>= 1)
```

```
      if (d & 1) x = faz[x][j];
    if (x == y) return x;
    for (int j = 17; ~j; --j)
      if (faz[x][j] != faz[y][j]) x = faz[x][j], y = faz[y][j];
    return faz[x][0];
}

int main() {
  int Ti;
  scanf("%d", &Ti);
  while (Ti--) {
    scanf("%d%d", &N, &M);
    for (int i = 1; i <= N; ++i) {
      G[i].clear();
      dfn[i] = low[i] = 0;
    }
    for (int i = 1; i <= N * 2; ++i) T[i].clear();
    for (int i = 1, x, y; i <= M; ++i) {
      scanf("%d%d", &x, &y);
      G[x].push_back(y);
      G[y].push_back(x);
    }
    cnt = N;
    dfc = 0, Tarjan(1), --tp;
    dfc = 0, DFS(1, 0);
    scanf("%d", &Q);
    while (Q--) {
      static int S, A[MN];
      scanf("%d", &S);
      int Ans = -2 * S;
      for (int i = 1; i <= S; ++i) scanf("%d", &A[i]);
      std::sort(A + 1, A + S + 1, [](int i, int j) { return dfn[i] < dfn[j]; });
      for (int i = 1; i <= S; ++i) {
        int u = A[i], v = A[i % S + 1];
        Ans += dis[u] + dis[v] - 2 * dis[LCA(u, v)];
      }
      if (LCA(A[1], A[S]) <= N) Ans += 2;
      printf("%d\n", Ans / 2);
    }
  }
  return 0;
}
```

外部链接

immortalCO，圆方树——处理仙人掌的利器[7]，Universal OJ。

参考资料与注释

[1] 2017 年陈俊锟同学在他的 IOI2017 中国国家集训队论文《〈神奇的子图〉命题报告及其拓展》中定义并命名了圆方树这一结构。

[2] 陈俊锟，《平凡的圆方树和神奇的（~~动态~~）动态规划》，NOI2018 冬令营，第 4 页。

[3] 「APIO2018」铁人两项

[4] Codeforces #487 E. Tourists

[5] 「SDOI2018」战略游戏

[6] 「SDOI2015」寻宝游戏

[7] 圆方树——处理仙人掌的利器

11.19.5 点/边连通度

Authors: jifbt

定义

以下内容的定义，请参见 图论相关概念：

- 边连通度、边割集；
- 点连通度、点割集；
- 团。

性质

Whitney 不等式

Whitney 不等式（1932）给出了点连通度 λ、边连通度 κ 和最小度 δ 之间的关系：

$$\kappa \le \lambda \le \delta$$

" 证明"

直觉上，如果有一个大小为 λ 的边割集，其中每一条边任选一个端点，就可以得到一个大小为 λ 的点割集，所以第一个不等式成立。

与度最小的结点（如有多个，任选一个）相邻的所有边构成大小为 δ 的边割集，所以第二个不等式也成立。

这个不等式不能改进；换言之，对每个满足它的三元组，均可以找出满足这个三元组的图。

" 构造"

把两个大小为 $\delta+1$ 的团用 λ 条边连起来，使两个团分别有 λ 和 κ 个不同的结点被连在这些边上。

Menger 定理

由 最大流最小割定理（又名 Ford–Fulkerson 定理）可推出，两点间的不相交（指两两没有公共边）路径的最大数量等于割集的最小大小（这个推论又叫 **Menger 定理**——译者注）。

计算

以下图的边权均为 1。

用最大流计算边连通度

枚举点对 (s,t)，以 s 为源点，t 为汇点跑边权为 1 的最大流。需要 $O(n^2)$ 次最大流，如果使用 Edmonds–Karp 算法，复杂度为 $O(|V|^3|E|^2)$。使用 Dinic 算法可以更优，复杂度为 $O(|V|^2|E|\min(|V|^{2/3},|E|^{1/2}))$。

全局最小割

使用 Stoer–Wagner 算法 只需跑一次无源汇最小割即可。复杂度为 $O(|V||E|+|V|^2\log|V|)$，一般可近似看作 $O(|V|^3)$。

点连通度

仍然枚举点对，这次把每个非源汇的点 x 拆成两个点 x_1 和 x_2，并连边 (x_1,x_2)。把原图中所有边 (u,v) 换成两条边 (u_2,v_1) 和 (v_2,u_1)。此时最大流等于 s、t 之间的最小点割集大小（又称局部点连通度）。复杂度与用最大流计算边连通度相同。

本页面译自博文 .[1]、 .[2] 与其英文翻译版 Edge connectivity/Vertex connectivity[3]。其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0。

延伸阅读

- 论文 *Connectivity Algorithms*[4] 介绍了近年来连通度计算算法的进展。感兴趣的读者可以自行浏览。

参考资料与注释

[1] .

[2] .

[3] Edge connectivity/Vertex connectivity

[4] *Connectivity Algorithms*

11.20 环计数问题

普通环计数

"例题 1：Codeforces Beta Round 11 D. A Simple Task[1]"

给定一个简单图，求图中简单环的数目。简单环是指没有重复顶点或边的环。

结点数目 $1 \le n \le 19$。

" 解题思路"

考虑状态压缩动态规划。记 $f(s,i)$ 表示满足当前经过结点集合为 s，且现在在结点 i 上，且第一个结点为结点集合 s 中**编号最小的那个**的路径条数。

对于状态 $f(s,i)$，枚举下一个结点 u。若 u 在集合 s 中且是编号最小的那个（即起点），就将答案 A 加上 $f(s,i)$。若 u 不在 s 中，就将 $f(s,i)$ 加上 $f(s\cup\{u\},u)$。

这样会把二元环（即重边）也算上，并且每个非二元环会被计算两次（因为固定起点可以向两个方向走），所以答案为 $\dfrac{A-m}{2}$，其中 m 表示边数。时间复杂度 $O(2^n m)$。

" 示例代码"

```cpp
#include <bits/stdc++.h>
using namespace std;

int n, m;

struct Edge {
  int to, nxt;
} edge[500];

int cntEdge, head[20];

void addEdge(int u, int v) {
  edge[++cntEdge] = {v, head[u]}, head[u] = cntEdge;
}

long long answer, dp[1 << 19][20];

int main() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= m; i++) {
    int u, v;
    scanf("%d%d", &u, &v);
    addEdge(u, v);
    addEdge(v, u);
  }
  for (int i = 1; i <= n; i++) dp[1 << i - 1][i] = 1;
  for (int s = 1; s < (1 << n); s++)
    for (int i = 1; i <= n; i++) {
      if (!dp[s][i]) continue;
      for (int j = head[i]; j; j = edge[j].nxt) {
        int u = i, v = edge[j].to;
        if ((s & -s) > (1 << v - 1)) continue;
        if (s & (1 << v - 1)) {
          if ((s & -s) == (1 << v - 1)) answer += dp[s][u];
        } else
          dp[s | (1 << v - 1)][v] += dp[s][u];
      }
    }
  printf("%lld\n", (answer - m) / 2);
  return 0;
}
```

三元环计数

三元环指的是一个简单图 G 中的一个无序三元组 (u, v, w) 满足存在三条边分别连接 (u, v)，(v, w) 和 (w, u)。而**三元环计数问题**要求计算出图中所有三元环的数量。

首先给所有边定向。我们规定从度数小的点指向度数大的点，度数相同就从编号小的点指向编号大的点。那么此时此图是一张有向无环图（DAG）。

" 该图没有环的证明"

反证法，假设存在环，那么环中的点度数一个比一个大，要形成环，所有点的度数必须相等，但是编号必定不同，矛盾。

所以定向后图肯定不存在环。

事实上，可以根据上述定向规则构造一个 偏序，所以按此规则构造的图（也即该偏序的 Hasse 图）一定是一个 DAG。

枚举 u 和 u 指向的点 v，再在 v 指向的点中枚举 w，检验 u 是否与 w 相连即可。

这个算法的时间复杂度为 $O(m\sqrt{m})$。

" 时间复杂度证明"

对于定向部分，遍历了所有的边，时间复杂度 $O(n+m)$。

对于每一对 (v, w)，u 的数量都不超过 v 的入度 $d^{-}(v)$。

若 $d^{-}(v) \le \sqrt{m}$，由于 w 的个数至多为 n，所以这部分时间复杂度为 $O(n\sqrt{m})$。

若 $d^{-}(v) > \sqrt{m}$，由于 v 指向 w，所以 $d(v) \le d(w)$，得出 $d(w) > \sqrt{m}$，但是总边数只有 m，所以这样的 w 的个数至多为 $\sqrt{m}$，故时间复杂度为 $O(m\sqrt{m})$。

总时间复杂度为 $O(n+m+n\sqrt{m}+m\sqrt{m}) = O(m\sqrt{m})$。

事实上，如果定向时从度数大的点指向度数小的点，复杂度也正确，只需要交换 u，w 两个点，上述证明也成立。

" 示例代码（洛谷 P1989 无向图三元环计数[2]）"

```cpp
#include <bits/stdc++.h>
using namespace std;

int n, m, total;
int deg[200001], u[200001], v[200001];
bool vis[200001];

struct Edge {
  int to, nxt;
} edge[200001];

int cntEdge, head[100001];

void addEdge(int u, int v) {
  edge[++cntEdge] = {v, head[u]}, head[u] = cntEdge;
}

int main() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= m; i++)
    scanf("%d%d", u + i, v + i), deg[u[i]]++, deg[v[i]]++;
  for (int i = 1; i <= m; i++) {
    if ((deg[u[i]] == deg[v[i]] && u[i] > v[i]) || deg[u[i]] < deg[v[i]])
      swap(u[i], v[i]);
    addEdge(u[i], v[i]);
  }
  for (int u = 1; u <= n; u++) {
    for (int i = head[u]; i; i = edge[i].nxt) vis[edge[i].to] = true;
    for (int i = head[u]; i; i = edge[i].nxt) {
      int v = edge[i].to;
      for (int j = head[v]; j; j = edge[j].nxt) {
        int w = edge[j].to;
```

```
        if (vis[w]) total++;
      }
    }
    for (int i = head[u]; i; i = edge[i].nxt) vis[edge[i].to] = false;
  }
  printf("%d\n", total);
  return 0;
}
```

例题 2

"HDU 6184 Counting Stars[3]"

给定一张有 n 个点和 m 条边的无向图，求下面图形的出现次数。

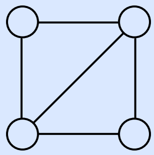


图 11.77

$2 \le n \le 10^5$，$1 \le m \le \min\left\{2 \times 10^5, \dfrac{n(n-1)}{2}\right\}$。

" 解题思路"

这个图形是两个三元环共用了一条边形成的。所以我们先跑一遍三元环计数，统计出一条边上三元环的数量，然后枚举共用的那条边，设有 x 个三元环中有此边，那么对答案的贡献就是 $\dbinom{x}{2}$。

时间复杂度 $O(m\sqrt{m})$。

" 示例代码"

```
#include <bits/stdc++.h>
using namespace std;

int n, m, total;
int deg[200001], u[200001], v[200001];
int edgeId[200001], cnt[200001];

struct Edge {
  int to, nxt;
} edge[200001];

int cntEdge, head[100001];

void addEdge(int u, int v) {
  edge[++cntEdge] = {v, head[u]}, head[u] = cntEdge;
}

int main() {
  while (cin >> n >> m) {
    cntEdge = total = 0;
```

```
    memset(deg, 0, sizeof deg);
    memset(head, 0, sizeof head);
    for (int i = 1; i <= m; i++)
      scanf("%d%d", u + i, v + i), deg[u[i]]++, deg[v[i]]++;
    for (int i = 1; i <= m; i++) {
      if ((deg[u[i]] == deg[v[i]] && u[i] > v[i]) || deg[u[i]] < deg[v[i]])
        swap(u[i], v[i]);
      addEdge(u[i], v[i]);
    }

    for (int u = 1; u <= n; u++) {
      for (int i = head[u]; i; i = edge[i].nxt) edgeId[edge[i].to] = i;
      for (int i = head[u]; i; i = edge[i].nxt) {
        int v = edge[i].to;
        for (int j = head[v]; j; j = edge[j].nxt) {
          int w = edge[j].to;
          if (edgeId[w]) cnt[i]++, cnt[j]++, cnt[edgeId[w]]++;
        }
      }
      for (int i = head[u]; i; i = edge[i].nxt) edgeId[edge[i].to] = 0;
    }
    for (int i = 1; i <= m; i++) total += cnt[i] * (cnt[i] - 1) / 2, cnt[i] = 0;

    printf("%d\n", total);
  }
  return 0;
}
```

四元环计数

类似地，**四元环**就是指四个点 a，b，c，d 满足 $(a,\ b)$，$(b,\ c)$，$(c,\ d)$ 和 $(d,\ a)$ 均有边连接。

考虑先对点进行排序。度数小的排在前面，度数大的排在后面。

考虑枚举排在最后面的点 a，此时只需要对于每个比 a 排名更前的点 c，都求出有多少个排名比 a 前的点 b 满足 $(a,\ b)$，$(b,\ c)$ 有边。然后只需要从这些 b 中任取两个都能成为一个四元环。求 b 的数量只需要遍历一遍 b 和 c 即可。

注意到我们枚举的复杂度本质上与枚举三元环等价，所以时间复杂度也是 $O(m\sqrt{m})$（假设 n，m 同阶）。

值得注意的是，$(a,\ b,\ c,\ d)$ 和 $(a,\ c,\ b,\ d)$ 可以是两个不同的四元环。

另外，度数相同的结点的排名将不相同，并且需要注意判断 $a \neq c$。

" 示例代码（LibreOJ P191 无向图四元环计数[4]）"

```
#include <bits/stdc++.h>
using namespace std;

int n, m, deg[100001], cnt[100001];
vector<int> E[100001], E1[100001];

long long total;

int main() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= m; i++) {
    int u, v;
```

```
    scanf("%d%d", &u, &v);
    E[u].push_back(v);
    E[v].push_back(u);
    deg[u]++, deg[v]++;
  }
  for (int u = 1; u <= n; u++)
    for (int v : E[u])
      if (deg[u] > deg[v] || (deg[u] == deg[v] && u > v)) E1[u].push_back(v);
  for (int a = 1; a <= n; a++) {
    for (int b : E1[a])
      for (int c : E[b]) {
        if (deg[a] < deg[c] || (deg[a] == deg[c] && a <= c)) continue;
        total += cnt[c]++;
      }
    for (int b : E1[a])
      for (int c : E[b]) cnt[c] = 0;
  }
  printf("%lld\n", total);
  return 0;
}
```

例题 3

"Gym 102028L Connected Subgraphs[5]"

给定一张有 n 个点和 m 条边的无向图，求四条边的导出子图连通的情况数。

$4 \le n \le 10^5$，$4 \le m \le 2 \times 10^5$。

" 解题思路"

容易把情况分为五种：菊花图、四元环、三元环上一个点连出一条边、四个点构成的链中间一个点连出一条边以及五个点构成的链。

菊花图直接枚举点的度数，用组合数解决即可。四元环可以直接按照上述算法求得。三元环部分只需枚举三元环 $(u,\ v,\ w)$，那么对答案的贡献就是 $[d(u)-2]+[d(v)-2]+[d(w)-2]$。

下面考虑第四种情况。考虑枚举度数为 2 的点 x，再枚举与它相邻的一个结点 y 作为度数为 3 的那个点。此时对答案的贡献为 $[d(x)-1] \cdot \dbinom{d(y)-1}{2}$。但是注意到 y 的相邻节点可能会和 x 的相邻结点重合，此时的图形等价于第三种情况。但是每种多算的第三种情况都会被多算两次（因为有两个度数为 3 的点），所以应该减去第三种情况数目的两倍。

对于最后一种情况，先枚举中间的点 x，那么容易发现对答案的贡献是

$$\sum_{y \in son_x} \sum_{z \in son_x} [d(y)-1] \cdot [d(z)-1].$$

同样地，这其中有多算的部分。设 y 的相邻结点为 s，z 的相邻结点为 t，那么思考后发现多算的有如下几种情况：

1. y 与 t 重合，但是 s 与 z 不重合时，等价于第三种情况；
2. s 与 z 重合，但是 y 与 t 不重合时，同样等价于第三种情况；
3. y 与 t，s 与 z 都重合时，等价于一个三元环；
4. s 与 t 重合时，等价于一个四元环（第二种情况）。

考虑到第三种情况中两个度数 2 的点作为 x 时正好分别对应上述多算情况的 1 和 2，所以要额外减去第三种

情况数目的两倍。对于一个三元环，三个结点都可以作为 x，多算了 3 次。同样的，四元环的情况被多算了 4 次。

于是我们就得出了所有情况的算法，时间复杂度为 $O(n + m\sqrt{m})$。

" 示例代码"

```
#include <bits/stdc++.h>
using namespace std;
const int mod = 1000000007;

long long power(long long a, long long n = mod - 2) {
  long long res = 1;
  while (n) {
    if (n & 1) res = res * a % mod;
    a = a * a % mod;
    n >>= 1;
  }
  return res;
}

const long long power24 = power(24), power2 = power(2);

int n, m;
vector<int> E[100001], E1[100001], E2[100001];
bool vis[100001];
int cnt1[100001], cnt2[100001];

long long solve1();
long long solve2();
long long solve3();
long long solve4();
long long solve5();
long long ans[6];

long long solve1() {
  if (ans[1] != -1) return ans[1];
  ans[1] = 0;
  for (int i = 1; i <= n; i++) {
    int x = E[i].size();
    ans[1] +=
        1ll * x * (x - 1) % mod * (x - 2) % mod * (x - 3) % mod * power24 % mod;

  }
  return ans[1] %= mod;
}

long long solve2() {
  if (ans[2] != -1) return ans[2];
  ans[2] = 0;
  for (int i = 1; i <= n; i++) {
    for (int j : E1[i])
      for (int k : E1[j]) cnt1[k]++;
    for (int j : E2[i])
      for (int k : E1[j])
```

```
        if (k != i) cnt2[k]++;
      for (int j : E1[i])
        for (int k : E1[j])
          ans[2] +=
              (1ll * cnt1[k] * (cnt1[k] - 1) / 2 + 1ll * cnt1[k] * cnt2[k]) % mod,

              cnt1[k] = 0;
      for (int j : E2[i])
        for (int k : E1[j])
          if (k != i)
            ans[2] += 1ll * cnt2[k] * (cnt2[k] - 1) / 2 % mod * power2 % mod,
                cnt2[k] = 0;
    }
    return ans[2];
  }

  long long solve3() {
    if (ans[3] != -1) return ans[3];
    ans[0] = ans[3] = 0;
    for (int i = 1; i <= n; i++) {
      for (int j : E1[i]) vis[j] = true;
      for (int j : E1[i])
        for (int k : E1[j])
          if (vis[k])
            ans[3] =
                (1ll * ans[3] + E[i].size() + E[j].size() + E[k].size() - 6) %
                mod,
            ans[0]++;
      for (int j : E1[i]) vis[j] = false;
    }
    return ans[3];
  }

  long long solve4() {
    if (ans[4] != -1) return ans[4];
    ans[4] = 0;
    for (int i = 1; i <= n; i++)
      for (int j : E[i])
        (ans[4] += 1ll * (E[j].size() - 1) * (E[j].size() - 2) / 2 *
                   (E[i].size() - 1) % mod) %= mod;
    return ans[4] = (ans[4] - 2 * solve3()) % mod;
  }

  long long solve5() {
    if (ans[5] != -1) return ans[5];
    ans[5] = 0;
    for (int i = 1; i <= n; i++) {
      long long sum = 0;
      for (int j : E[i]) {
        ans[5] += sum * (E[j].size() - 1) % mod;
        sum += E[j].size() - 1;
      }
    }
    solve3();
```

```
  return ans[5] =
             (ans[5] % mod - 2 * solve3() - 4 * solve2() - 3 * ans[0]) % mod;
}

int main() {
  int T;
  scanf("%d", &T);
  while (T--) {
    ans[5] = ans[1] = ans[2] = ans[4] = ans[3] = -1;
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i++) E[i].clear(), E1[i].clear(), E2[i].clear();
    while (m--) {
      int x, y;
      scanf("%d%d", &x, &y);
      E[x].push_back(y), E[y].push_back(x);
    }
    for (int i = 1; i <= n; i++)
      for (int j : E[i]) {
        if (make_pair(E[i].size(), i) < make_pair(E[j].size(), j))
          E1[i].push_back(j);
        else
          E2[i].push_back(j);
      }
    printf(
        "%lld\n",
        ((solve5() + solve1() + solve2() + solve4() + solve3()) % mod + mod) %
            mod);
  }
  return 0;
}
```

习题

洛谷 P3547 [POI2013] CEN-Price List[6]

CodeForces 985G Team Players[7]（容斥原理）

参考资料与注释

[1] 例题 1：Codeforces Beta Round 11 D. A Simple Task

[2] 洛谷 P1989 无向图三元环计数

[3] HDU 6184 Counting Stars

[4] LibreOJ P191 无向图四元环计数

[5] Gym 102028L Connected Subgraphs

[6] 洛谷 P3547 [POI2013] CEN-Price List

[7] CodeForces 985G Team Players

11.21 2-SAT

> SAT 是适定性（Satisfiability）问题的简称。一般形式为 k - 适定性问题，简称 k-SAT。而当 $k > 2$ 时该问题为 NP 完全的。所以我们只研究 $k = 2$ 的情况。

定义

2-SAT，简单的说就是给出 n 个集合，每个集合有两个元素，已知若干个 $\langle a,b\rangle$，表示 a 与 b 矛盾（其中 a 与 b 属于不同的集合）。然后从每个集合选择一个元素，判断能否一共选 n 个两两不矛盾的元素。显然可能有多种选择方案，一般题中只需要求出一种即可。

现实意义

比如邀请人来吃喜酒，夫妻二人必须去一个，然而某些人之间有矛盾（比如 A 先生与 B 女士有矛盾，C 女士不想和 D 先生在一起），那么我们要确定能否避免来人之间有矛盾，有时需要方案。这是一类生活中常见的问题。

使用布尔方程表示上述问题。设 a 表示 A 先生去参加，那么 B 女士就不能参加（$\neg a$）；b 表示 C 女士参加，那么 $\neg b$ 也一定成立（D 先生不参加）。总结一下，即 $(a \vee b)$（变量 a,b 至少满足一个）。对这些变量关系建有向图，则有：$\neg a \to b \wedge \neg b \to a$（$a$ 不成立则 b 一定成立；同理，b 不成立则 a 一定成立）。建图之后，我们就可以使用缩点算法来求解 2-SAT 问题了。

常用解决方法

Tarjan SCC 缩点

算法考究在建图这点，我们举个例子来讲：

假设有 $a1,a2$ 和 $b1,b2$ 两对，已知 $a1$ 和 $b2$ 间有矛盾，于是为了方案自洽，由于两者中必须选一个，所以我们就要拉两条有向边 $(a1,b1)$ 和 $(b2,a2)$ 表示选了 $a1$ 则必须选 $b1$，选了 $b2$ 则必须选 $a2$ 才能够自洽。

然后通过这样子建边我们跑一遍 Tarjan SCC 判断是否有一个集合中的两个元素在同一个 SCC 中，若有则输出不可能，否则输出方案。构造方案只需要把几个不矛盾的 SCC 拼起来就好了。

输出方案时可以通过变量在图中的拓扑序确定该变量的取值。如果变量 x 的拓扑序在 $\neg x$ 之后，那么取 x 值为真。应用到 Tarjan 算法的缩点，即 x 所在 SCC 编号在 $\neg x$ 之前时，取 x 为真。因为 Tarjan 算法求强连通分量时使用了栈，所以 Tarjan 求得的 SCC 编号相当于反拓扑序。

显然地，时间复杂度为 $O(n+m)$。

暴搜

就是沿着图上一条路径，如果一个点被选择了，那么这条路径以后的点都将被选择，那么，出现不可行的情况就是，存在一个集合中两者都被选择了。

那么，我们只需要枚举一下就可以了，数据不大，答案总是可以出来的。

" 暴搜模板"

```
// 来源：刘汝佳白书第 323 页
struct Twosat {
  int n;
  vector<int> g[maxn * 2];
  bool mark[maxn * 2];
  int s[maxn * 2], c;
```

```
  bool dfs(int x) {
    if (mark[x ^ 1]) return false;
    if (mark[x]) return true;
    mark[x] = true;
    s[c++] = x;
    for (int i = 0; i < (int)g[x].size(); i++)
      if (!dfs(g[x][i])) return false;
    return true;
  }

  void init(int n) {
    this->n = n;
    for (int i = 0; i < n * 2; i++) g[i].clear();
    memset(mark, 0, sizeof(mark));
  }

  void add_clause(int x, int y) {  // 这个函数随题意变化
    g[x].push_back(y ^ 1);         // 选了 x 就必须选 y^1
    g[y].push_back(x ^ 1);
  }

  bool solve() {
    for (int i = 0; i < n * 2; i += 2)
      if (!mark[i] && !mark[i + 1]) {
        c = 0;
        if (!dfs(i)) {
          while (c > 0) mark[s[--c]] = false;
          if (!dfs(i + 1)) return false;
        }
      }
    return true;
  }
};
```

例题

HDU3062 Party[1]

> 题面：有 n 对夫妻被邀请参加一个聚会，因为场地的问题，每对夫妻中只有 1 人可以列席。在 $2n$ 个人中，某些人之间有着很大的矛盾（当然夫妻之间是没有矛盾的），有矛盾的 2 个人是不会同时出现在聚会上的。有没有可能会有 n 个人同时列席？

这是一道多校题，裸的 2-SAT 判断是否有方案，按照我们上面的分析，如果 $a1$ 中的丈夫和 $a2$ 中的妻子不合，我们就把 $a1$ 中的丈夫和 $a2$ 中的丈夫连边，把 $a2$ 中的妻子和 $a1$ 中的妻子连边，然后缩点染色判断即可。

" 参考代码"

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <iostream>
#define maxn 2018
```

```
#define maxm 4000400
using namespace std;
int Index, instack[maxn], DFN[maxn], LOW[maxn];
int tot, color[maxn];
int numedge, head[maxn];

struct Edge {
  int nxt, to;
} edge[maxm];

int sta[maxn], top;
int n, m;

void add(int x, int y) {
  edge[++numedge].to = y;
  edge[numedge].nxt = head[x];
  head[x] = numedge;
}

void tarjan(int x) {  // 缩点看不懂请移步强连通分量上面有一个链接可以点。
  sta[++top] = x;
  instack[x] = 1;
  DFN[x] = LOW[x] = ++Index;
  for (int i = head[x]; i; i = edge[i].nxt) {
    int v = edge[i].to;
    if (!DFN[v]) {
      tarjan(v);
      LOW[x] = min(LOW[x], LOW[v]);
    } else if (instack[v])
      LOW[x] = min(LOW[x], DFN[v]);
  }
  if (DFN[x] == LOW[x]) {
    tot++;
    do {
      color[sta[top]] = tot;  // 染色
      instack[sta[top]] = 0;
    } while (sta[top--] != x);
  }
}

bool solve() {
  for (int i = 0; i < 2 * n; i++)
    if (!DFN[i]) tarjan(i);
  for (int i = 0; i < 2 * n; i += 2)
    if (color[i] == color[i + 1]) return 0;
  return 1;
}

void init() {
  top = 0;
  tot = 0;
  Index = 0;
  numedge = 0;
  memset(sta, 0, sizeof(sta));
```

```
  memset(DFN, 0, sizeof(DFN));
  memset(instack, 0, sizeof(instack));
  memset(LOW, 0, sizeof(LOW));
  memset(color, 0, sizeof(color));
  memset(head, 0, sizeof(head));
}

int main() {
  while (~scanf("%d%d", &n, &m)) {
    init();
    for (int i = 1; i <= m; i++) {
      int a1, a2, c1, c2;
      scanf("%d%d%d%d", &a1, &a2, &c1, &c2);  // 自己做的时候别用 cin 会被卡
      add(2 * a1 + c1, 2 * a2 + 1 - c2);
      // 对于第 i 对夫妇，我们用 2i+1 表示丈夫，2i 表示妻子。
      add(2 * a2 + c2, 2 * a1 + 1 - c1);
    }
    if (solve())
      printf("YES\n");
    else
      printf("NO\n");
  }
  return 0;
}
```

2018-2019 ACM-ICPC Asia Seoul Regional K TV Show Game[2]

> 题面：有 $k(k>3)$ 盏灯，每盏灯是红色或者蓝色，但是初始的时候不知道灯的颜色。有 n 个人，每个人选择 3 盏灯并猜灯的颜色。一个人猜对两盏灯或以上的颜色就可以获得奖品。判断是否存在一个灯的着色方案使得每个人都能领奖，若有则输出一种灯的着色方案。

这道题在判断是否有方案的基础上，在有方案时还要输出一个可行解。

根据 伍昱 -《由对称性解 2-sat 问题》[3]，我们可以得出：如果要输出 2-SAT 问题的一个可行解，只需要在 tarjan 缩点后所得的 DAG 上自底向上地进行选择和删除。

具体实现的时候，可以通过构造 DAG 的反图后在反图上进行拓扑排序实现；也可以根据 tarjan 缩点后，所属连通块编号越小，节点越靠近叶子节点这一性质，优先对所属连通块编号小的节点进行选择。

下面给出第二种实现方法的代码。

" 参考代码"

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1e4 + 5;
const int maxk = 5005;

int n, k;
int id[maxn][5];
char s[maxn][5][5], ans[maxk];
bool vis[maxn];

struct Edge {
  int v, nxt;
```

```
} e[maxn * 100];

int head[maxn], tot = 1;

void addedge(int u, int v) {
  e[tot].v = v;
  e[tot].nxt = head[u];
  head[u] = tot++;
}

int dfn[maxn], low[maxn], color[maxn], stk[maxn], ins[maxn], top, dfs_clock, c;

void tarjan(int x) {  // tarjan 算法求强联通
  stk[++top] = x;
  ins[x] = 1;
  dfn[x] = low[x] = ++dfs_clock;
  for (int i = head[x]; i; i = e[i].nxt) {
    int v = e[i].v;
    if (!dfn[v]) {
      tarjan(v);
      low[x] = min(low[x], low[v]);
    } else if (ins[v])
      low[x] = min(low[x], dfn[v]);
  }
  if (dfn[x] == low[x]) {
    c++;
    do {
      color[stk[top]] = c;
      ins[stk[top]] = 0;
    } while (stk[top--] != x);
  }
}

int main() {
  scanf("%d %d", &k, &n);
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= 3; j++) scanf("%d%s", &id[i][j], s[i][j]);

    for (int j = 1; j <= 3; j++) {
      for (int k = 1; k <= 3; k++) {
        if (j == k) continue;
        int u = 2 * id[i][j] - (s[i][j][0] == 'B');
        int v = 2 * id[i][k] - (s[i][k][0] == 'R');
        addedge(u, v);
      }
    }
  }

  for (int i = 1; i <= 2 * k; i++)
    if (!dfn[i]) tarjan(i);

  for (int i = 1; i <= 2 * k; i += 2)
    if (color[i] == color[i + 1]) {
      puts("-1");
```

```
      return 0;
    }

  for (int i = 1; i <= 2 * k; i += 2) {
    int f1 = color[i], f2 = color[i + 1];
    if (vis[f1]) {
      ans[(i + 1) >> 1] = 'R';
      continue;
    }
    if (vis[f2]) {
      ans[(i + 1) >> 1] = 'B';
      continue;
    }
    if (f1 < f2) {
      vis[f1] = 1;
      ans[(i + 1) >> 1] = 'R';
    } else {
      vis[f2] = 1;
      ans[(i + 1) >> 1] = 'B';
    }
  }
  ans[k + 1] = 0;
  printf("%s\n", ans + 1);
  return 0;
}
```

练习题

洛谷 P5782 和平委员会[4]

POJ3683 牧师忙碌日[5]

参考资料与注释

[1] Party

[2] TV Show Game

[3] 伍昱 -《由对称性解 2-sat 问题》

[4] 洛谷 P5782 和平委员会

[5] 牧师忙碌日

11.22 欧拉图

本页面将简要介绍欧拉图的概念、实现和应用。

定义

- **欧拉回路**：通过图中每条边恰好一次的回路

- **欧拉通路**：通过图中每条边恰好一次的通路
- **欧拉图**：具有欧拉回路的图
- **半欧拉图**：具有欧拉通路但不具有欧拉回路的图

性质

欧拉图中所有顶点的度数都是偶数。

若 G 是欧拉图，则它为若干个环的并，且每条边被包含在奇数个环内。

判别法

1. 无向图是欧拉图当且仅当：
   - 非零度顶点是连通的
   - 顶点的度数都是偶数
2. 无向图是半欧拉图当且仅当：
   - 非零度顶点是连通的
   - 恰有 2 个奇度顶点
3. 有向图是欧拉图当且仅当：
   - 非零度顶点是强连通的
   - 每个顶点的入度和出度相等
4. 有向图是半欧拉图当且仅当：
   - 非零度顶点是弱连通的
   - 至多一个顶点的出度与入度之差为 1
   - 至多一个顶点的入度与出度之差为 1
   - 其他顶点的入度和出度相等

求欧拉回路或欧拉路

Fleury 算法

也称避桥法，是一个偏暴力的算法。

算法流程为每次选择下一条边的时候优先选择不是桥的边。

一个广泛使用但是错误的实现方式是先 Tarjan 预处理桥边，然后再 DFS 避免走桥。但是由于走图过程中边会被删去，一些非桥边会变为桥边导致错误。最简单的实现方法是每次删除一条边之后暴力跑一遍 Tarjan 找桥，时间复杂度是 $\Theta(m(n+m)) = \Theta(m^2)$。复杂的实现方法要用到动态图等，实用价值不高。

Hierholzer 算法

也称逐步插入回路法。

过程

算法流程为从一条回路开始，每次任取一条目前回路中的点，将其替换为一条简单回路，以此寻找到一条欧拉回路。如果从路开始的话，就可以寻找到一条欧拉路。

实现

Hierholzer 算法的暴力实现如下：

```
Input. The edges of the graph e, where each element in e is (u, v)
Output. The vertex of the Euler Road of the input graph.
Method.
Function Hierholzer (v)
    circle ← Find a Circle in e Begin with v
    if circle = ∅
        return v
    e ← e − circle
    for each v ∈ circle
        v ← Hierholzer(v)
    return circle
Endfunction
return Hierholzer(any vertex)
```

性质

这个算法的时间复杂度约为 $O(nm+m^2)$。实际上还有复杂度更低的实现方法，就是将找回路的 DFS 和 Hierholzer 算法的递归合并，边找回路边使用 Hierholzer 算法。

如果需要输出字典序最小的欧拉路或欧拉回路的话，因为需要将边排序，时间复杂度是 $\Theta(n+m\log m)$（计数排序或者基数排序可以优化至 $\Theta(n+m)$）。如果不需要排序，时间复杂度是 $\Theta(n+m)$。

应用

有向欧拉图可用于计算机译码。

设有 m 个字母，希望构造一个有 m^n 个扇形的圆盘，每个圆盘上放一个字母，使得圆盘上每连续 n 位对应长为 n 的符号串。转动一周（m^n 次）后得到由 m 个字母产生的长度为 n 的 m^n 个各不相同的符号串。

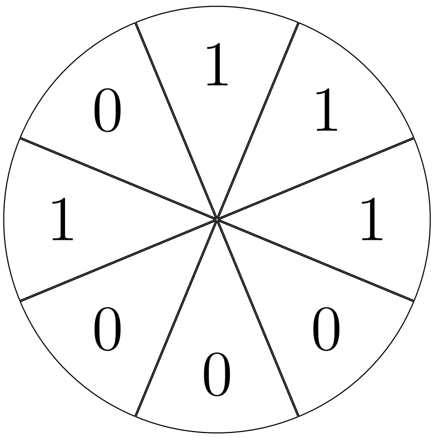


图 11.78

构造如下有向欧拉图：

设 $S=\{a_1,a_2,\cdots,a_m\}$，构造 $D=\langle V,E\rangle$，如下：

$V=\{a_{i_1}a_{i_2}\cdots a_{i_{n-1}}|a_i\in S,1\le i\le n-1\}$

$E=\{a_{j_1}a_{j_2}\cdots a_{j_{n-1}}|a_j\in S,1\le j\le n\}$

规定 D 中顶点与边的关联关系如下：

顶点 $a_{i_1}a_{i_2}\cdots a_{i_{n-1}}$ 引出 m 条边：$a_{i_1}a_{i_2}\cdots a_{i_{n-1}}a_r, r=1,2,\cdots,m$。

边 $a_{j_1}a_{j_2}\cdots a_{j_{n-1}}$ 引入顶点 $a_{j_2}a_{j_3}\cdots a_{j_n}$。

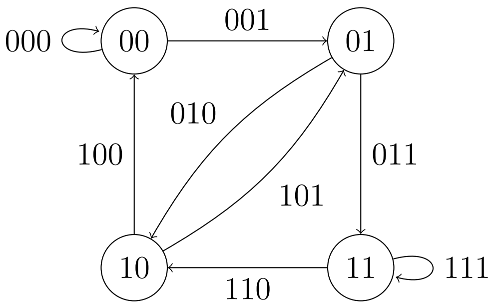


图 11.79

这样的 D 是连通的，且每个顶点入度等于出度（均等于 m），所以 D 是有向欧拉图。

任求 D 中一条欧拉回路 C，取 C 中各边的最后一个字母，按各边在 C 中的顺序排成圆形放在圆盘上即可。

例题

"洛谷 P2731 骑马修栅栏[1]"

给定一张有 500 个顶点的无向图，求这张图的一条欧拉路或欧拉回路。如果有多组解，输出最小的那一组。

在本题中，欧拉路或欧拉回路不需要经过所有顶点。

边的数量 m 满足 $1 \leq m \leq 1024$。

" 解题思路"

用 Fleury 算法解决本题的时候只需要再贪心就好，但是由于复杂度不对，所以要换用 Hierholzer 算法。

保存答案可以使用 `std::stack<int>`，因为如果找的不是回路的话必须将那一部分放在最后。

注意，不能使用邻接矩阵存图，否则时间复杂度会退化为 $\Theta(nm)$。由于需要将边排序，建议使用前向星或者 `std::vector` 存图。示例代码使用 `std::vector`。

" 示例代码"

```
#include <algorithm>
#include <cstdio>
#include <stack>
#include <vector>
using namespace std;

struct edge {
  int to;
  bool exists;
  int revref;

  bool operator<(const edge& b) const { return to < b.to; }
};

vector<edge> beg[505];
int cnt[505];

const int dn = 500;
stack<int> ans;
```

```
void Hierholzer(int x) {  // 关键函数
  for (int& i = cnt[x]; i < (int)beg[x].size();) {
    if (beg[x][i].exists) {
      edge e = beg[x][i];
      beg[x][i].exists = 0;
      beg[e.to][e.revref].exists = 0;
      ++i;
      Hierholzer(e.to);
    } else {
      ++i;
    }
  }
  ans.push(x);
}

int deg[505];
int reftop[505];

int main() {
  for (int i = 1; i <= dn; ++i) {
    beg[i].reserve(1050);  // vector 用 reserve 避免动态分配空间，加快速度
  }

  int m;
  scanf("%d", &m);
  for (int i = 1; i <= m; ++i) {
    int a, b;
    scanf("%d%d", &a, &b);
    beg[a].push_back((edge){b, 1, 0});
    beg[b].push_back((edge){a, 1, 0});
    ++deg[a];
    ++deg[b];
  }

  for (int i = 1; i <= dn; ++i) {
    if (!beg[i].empty()) {
      sort(beg[i].begin(), beg[i].end());  // 为了要按字典序贪心，必须排序
    }
  }

  for (int i = 1; i <= dn; ++i) {
    for (int j = 0; j < (int)beg[i].size(); ++j) {
      beg[i][j].revref = reftop[beg[i][j].to]++;
    }
  }

  int bv = 0;
  for (int i = 1; i <= dn; ++i) {
    if (!deg[bv] && deg[i]) {
      bv = i;
    } else if (!(deg[bv] & 1) && (deg[i] & 1)) {
      bv = i;
    }
```

```
  }

  Hierholzer(bv);

  while (!ans.empty()) {
    printf("%d\n", ans.top());
    ans.pop();
  }
}
```

习题

- SGU 101 Domino[2]
- POJ 1780 Code[3]
- 洛谷 P1127 词链[4]
- 洛谷 P1333 瑞瑞的木棍[5]
- 洛谷 P1341 无序字母对[6]
- 洛谷 P6066 [USACO05JAN]Watchcow S[7]
- 洛谷 P6628 [省选联考 2020 B 卷] 丁香之路[8]

参考资料与注释

[1] 洛谷 P2731 骑马修栅栏

[2] SGU 101 Domino

[3] POJ 1780 Code

[4] 洛谷 P1127 词链

[5] 洛谷 P1333 瑞瑞的木棍

[6] 洛谷 P1341 无序字母对

[7] 洛谷 P6066 [USACO05JAN]Watchcow S

[8] 洛谷 P6628 [省选联考 2020 B 卷] 丁香之路

11.23 哈密顿图

定义

通过图中所有顶点一次且仅一次的通路称为哈密顿通路。

通过图中所有顶点一次且仅一次的回路称为哈密顿回路。

具有哈密顿回路的图称为哈密顿图。

具有哈密顿通路而不具有哈密顿回路的图称为半哈密顿图。

性质

设 $G=\langle V,E\rangle$ 是哈密顿图，则对于 V 的任意非空真子集 V_1，均有 $p(G-V_1)\le |V_1|$。其中 $p(x)$ 为 x 的连通分支数。

推论：设 $G=\langle V,E\rangle$ 是半哈密顿图，则对于 V 的任意非空真子集 V_1，均有 $p(G-V_1)\le |V_1|+1$。其中 $p(x)$ 为 x 的连通分支数。

完全图 $K_{2k+1}(k\ge 1)$ 中含 k 条边不重的哈密顿回路，且这 k 条边不重的哈密顿回路含 K_{2k+1} 中的所有边。

完全图 $K_{2k}(k\ge 2)$ 中含 $k-1$ 条边不重的哈密顿回路，从 K_{2k} 中删除这 $k-1$ 条边不重的哈密顿回路后所得图含 k 条互不相邻的边。

充分条件

设 G 是 $n(n\ge 2)$ 的无向简单图，若对于 G 中任意不相邻的顶点 v_i,v_j，均有 $d(v_i)+d(v_j)\ge n-1$，则 G 中存在哈密顿通路。

推论 1：设 G 是 $n(n\ge 3)$ 的无向简单图，若对于 G 中任意不相邻的顶点 v_i,v_j，均有 $d(v_i)+d(v_j)\ge n$，则 G 中存在哈密顿回路，从而 G 为哈密顿图。

推论 2：设 G 是 $n(n\ge 3)$ 的无向简单图，若对于 G 中任意顶点 v_i，均有 $d(v_i)\ge \frac{n}{2}$，则 G 中存在哈密顿回路，从而 G 为哈密顿图。

设 D 为 $n(n\ge 2)$ 阶竞赛图，则 D 具有哈密顿通路。

若 D 含 $n(n\ge 2)$ 阶竞赛图作为子图，则 D 具有哈密顿通路。

强连通的竞赛图为哈密顿图。

若 D 含 $n(n\ge 2)$ 阶强连通的竞赛图作为子图，则 D 具有哈密顿回路。

11.24 二分图

定义

二分图，又称二部图，英文名叫 Bipartite graph。

二分图是什么？节点由两个集合组成，且两个集合内部没有边的图。

换言之，存在一种方案，将节点划分成满足以上性质的两个集合。

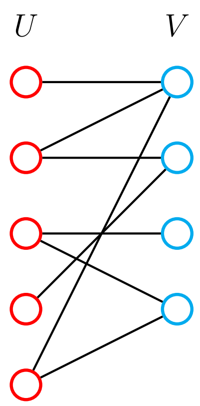


图 11.80

性质

- 如果两个集合中的点分别染成黑色和白色，可以发现二分图中的每一条边都一定是连接一个黑色点和一个白色点。
- 二分图不存在长度为奇数的环

  note

  因为每一条边都是从一个集合走到另一个集合，只有走偶数次才可能回到同一个集合。

判定

如何判定一个图是不是二分图呢？

换言之，我们需要知道是否可以将图中的顶点分成两个满足条件的集合。

显然，直接枚举答案集合的话实在是太慢了，我们需要更高效的方法。

考虑上文提到的性质，我们可以使用 DFS（图论） 或者 BFS 来遍历这张图。如果发现了奇环，那么就不是二分图，否则是。

应用

二分图最大匹配

详见 二分图最大匹配 页面。

二分图最大权匹配

详见 二分图最大权匹配 页面。

一般图最大匹配

详见 一般图最大匹配 页面。

一般图最大权匹配

详见 一般图最大权匹配 页面。

11.25 最小环

引入

"问题"

给出一个图，问其中的由 n 个节点构成的边权和最小的环 $(n \geq 3)$ 是多大。

图的最小环也称围长。

过程

暴力解法

设 u 和 v 之间有一条边长为 w 的边，$dis(u, v)$ 表示删除 u 和 v 之间的连边之后，u 和 v 之间的最短路。

那么无向图中的最小环是 $dis(u, v) + w$。

注意若是在有向图中求最小环，相对应的公式要修改，最小环是 $dis(v, u) + w$。

总时间复杂度 $O(n^2m)$。

Dijkstra

相关链接：最短路 /Dijkstra

过程

枚举所有边，每一次求删除一条边之后对这条边的起点跑一次 Dijkstra，道理同上。

性质

时间复杂度 $O(m(n + m) \log n)$。

Floyd

相关链接：最短路 /Floyd

过程

记原图中 u, v 之间边的边权为 $val\,(u, v)$。

我们注意到 Floyd 算法有一个性质：在最外层循环到点 k 时（尚未开始第 k 次循环），最短路数组 dis 中，$dis_{u,v}$ 表示的是从 u 到 v 且仅经过编号在 $[1, k)$ 区间中的点的最短路。

由最小环的定义可知其至少有三个顶点，设其中编号最大的顶点为 w，环上与 w 相邻两侧的两个点为 u, v，则在最外层循环枚举到 $k = w$ 时，该环的长度即为 $dis_{u,v} + val\,(v, w) + val\,(w, u)$。

故在循环时对于每个 k 枚举满足 $i < k, j < k$ 的 (i, j)，更新答案即可。

性质

时间复杂度：$O(n^3)$

实现

下面给出 C++ 的参考实现：

```
int val[maxn + 1][maxn + 1];  // 原图的邻接矩阵

int floyd(const int &n) {
  static int dis[maxn + 1][maxn + 1];  // 最短路矩阵
  for (int i = 1; i <= n; ++i)
    for (int j = 1; j <= n; ++j) dis[i][j] = val[i][j];  // 初始化最短路矩阵
  int ans = inf;
  for (int k = 1; k <= n; ++k) {
    for (int i = 1; i < k; ++i)
      for (int j = 1; j < i; ++j)
        ans = std::min(ans, dis[i][j] + val[i][k] + val[k][j]);  // 更新答案
    for (int i = 1; i <= n; ++i)
```

```
      for (int j = 1; j <= n; ++j)
        dis[i][j] = std::min(
            dis[i][j], dis[i][k] + dis[k][j]);  // 正常的 floyd 更新最短路矩阵
  }
  return ans;
}
```

```
val = [[0 for i in range(maxn + 1)] for j in range(maxn + 1)] # 原图的邻接矩阵

def floyd(n):
    dis = [[0 for i in range(maxn + 1)] for j in range(maxn + 1)] # 最短路矩阵
    for i in range(1, n + 1):
        for j in range(1, n + 1):
            dis[i][j] = val[i][j] # 初始化最短路矩阵
    ans = inf
    for k in range(1, n + 1):
        for i in range(1, k):
            for j in range(1, i):
                ans = min(ans, dis[i][j] + val[i][k] + val[k][j]) # 更新答案
        for i in range(1, n + 1):
            for j in range(1, n + 1):
                dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]) # 正常的 floyd
 更新最短路矩阵
    return ans
```

例题

GDOI2018 Day2 巡逻

给出一张 n 个点的无负权边无向图，要求执行 q 个操作，三种操作

1. 删除一个图中的点以及与它有关的边
2. 恢复一个被删除点以及与它有关的边
3. 询问点 x 所在的最小环大小

对于 50% 的数据，有 $n, q \le 100$

对于每一个点 x 所在的简单环，都存在两条与 x 相邻的边，删去其中的任意一条，简单环将变为简单路径。

那么枚举所有与 x 相邻的边，每次删去其中一条，然后跑一次 Dijkstra。

或者直接对每次询问跑一遍 Floyd 求最小环，$O(qn^3)$

对于 100% 的数据，有 $n, q \le 400$。

还是利用 Floyd 求最小环的算法。

若没有删除，删去询问点将简单环裂开成为一条简单路。

然而第二步的求解改用 Floyd 来得出。

那么答案就是要求出不经过询问点 x 的情况下任意两点之间的距离。

怎么在线？

强行离线，利用离线的方法来避免删除操作。

将询问按照时间顺序排列，对这些询问建立一个线段树。

每个点的出现时间覆盖所有除去询问该点的时刻外的所有询问，假设一个点被询问 x 次，则它的出现时间可以视为 $x+1$ 段区间，插入到线段树上。

完成之后遍历一遍整棵线段树，在经过一个点时存储一个 Floyd 数组的备份，然后加入被插入在这个区间上的所有点，在离开时利用备份数组退回去即可。

这个做法的时间复杂度为 $O(qn^2 \log q)$。

还有一个时间复杂度更优秀的在线做法。

对于一个对点 x 的询问，我们以 x 为起点跑一次最短路，然后把最短路树建出来，顺便处理出每个点是在 x 的哪棵子树内。

那么一定能找出一条非树边，满足这条非树边的两个端点在根的不同子树中，使得这条非树边 + 两个端点到根的路径就是最小环。

证明：

显然最小环包含至少两个端点在根的不同子树中一条非树边。

假设这条边为 (u,v)，那么最短路树上 x 到 u 的路径是所有 x 到 u 的路径中最短的那条，x 到 v 的路径也是最短的那条，那么 $x \to u \to v \to x$ 这个环肯定不会比最小环要长。

那么就可以枚举所有非树边，更新答案。

每次询问的复杂度为跑一次单源最短路的复杂度，为 $O(n^2)$。

总时间复杂度为 $O(qn^2)$。

11.26 平面图

定义

如果图 G 能画在平面 S 上，即除顶点处外无边相交，则称 G 可平面嵌入 S，G 为可平面图或平面图。画出的没有边相交的图称为 G 的平面表示或平面嵌入。

$K_{3,3}$ 和 K_5 不是平面图。

设 G 是平面图，由 G 的边将 G 所在的平面划分成若干个区域，每个区域称为 G 的一个面，其中面积无限的面称为无限面或外部面，面积有限的称为有限面或内部面。包围每个面的所有边组成的回路称为该面的边界，边界的长度称为该面的次数。

平面图中所有面的次数之和等于边数 m 的 2 倍。

若在简单平面图 G 的任意不相邻顶点间添加边，所得图为非平面图，称 G 为极大平面图。

若 G 为 $n(n \geq 3)$ 阶简单的连通平面图，G 为极大平面图当且仅当 G 的每个面的次数均为 3。

欧拉公式

对于任意的连通的平面图 G，有：

$n - m + r = 2$

其中，n, m, r，分别为 G 的阶数，边数和面数。

推论：对于有 $p(p \geq 2)$ 个连通分支的平面图 G，有

$n - m + r = p + 1$

可推出其他性质：

设 G 是连通的平面图，且 G 的各面的次数至少为 $l(l \geq 3)$，则有：

$m \leq \frac{l}{l-2}(n-2)$

推论：对于有 $p(p \geq 2)$ 个连通分支的平面图 G，有

$m \leq \frac{l}{l-2}(n-p-1)$

推论：设 G 是 $n \geq 3$ 阶 m 条边的简单平面图，则 $m \leq 3n - 6$

判断

若两个图 G_1 与 G_2 同构，或通过反复插入或消去 2 度顶点后是同构的，则称二者是同胚的。

库拉图斯基定理

图 G 是平面图当且仅当 G 不含与 K_5 或 $K_{3,3}$ 同胚的子图。

图 G 是平面图当且仅当 G 中没有可以收缩到 K_5 或 $K_{3,3}$ 的子图。

对偶图

设 G 是平面图的某一个平面嵌入，构造图 G^*：

1. 在 G 的每个面 R_i 中放置 G^* 的一个顶点 v_i^*
2. 设 e 为 G 的一条边，若 e 在 G 的面 R_i 和 R_j 的公共边界上，做 G^* 的边 e^* 与 e 相交，且 e^* 关联 G^* 的顶点 v_i^*, v_j^*，即 $e^* = (v_i^*, v_j^*)$，e^* 不与其他任何边相交。若 e 为 G 中桥且在 R_i 的边界上，则 e^* 是以 R_i 中顶点 v_i^* 为端点的环，即 $e^* = (v_i^*, v_j^*)$

称 G^* 为 G 的对偶图。

性质

1. G^* 为平面图，且是平面嵌入。
2. G 中自环在 G^* 中对应桥，G 中桥在 G^* 中对应自环。
3. G^* 是连通的。
4. 若 G 的面 R_i, R_j 的边界上至少有两条公共边，则关联 v_i^*, v_j^* 的边有平行边，G^* 多半是多重图。
5. 同构的图的对偶图不一定是同构的。
6. G^{**} 与 G 同构当且仅当 G 是连通图。

应用

平面图最小割转对偶图最短路：BZOJ 1001 狼抓兔子

外平面图

设 G 为平面图，若 G 存在平面嵌入 $\tilde{G}$，使得 G 中所有顶点都在 $\tilde{G}$ 的一个面的边界上，则称 G 为外可平面图，简称外平面图。

设 G 是简单的外平面图，若对于 G 中任二不相邻顶点 u, v，令 $G' = G \cup (u, v)$，则 G' 不是外平面图，称 G 为极大外平面图。

性质

所有顶点都在外部面边界上的 $n(n \geq 3)$ 阶外可平面图是极大外可平面图当且仅当 G 的每个外部面的边界都是长为 3 的圈，外部面的边界是一个长为 n 的圈。

$n(n \geq 3)$ 阶极大外平面图有 $n-2$ 个内部面。

设 G 是 $n(n \geq 3)$ 阶极大外平面图，则：

1. $m = 2n - 3$
2. G 中至少有 3 个顶点的度数小于等于 3
3. G 中至少有 2 个顶点的度数为 2
4. G 的点连通度 κ 为 2

一个图 G 是外平面图有当且仅当 G 中不含与 K_4 或 $K_{2,3}$ 同胚的子图。

任何 4 - 连通平面图都是哈密顿图。

11.27 图的着色

点着色

（讨论的是无自环无向图）

对无向图顶点着色，且相邻顶点不能同色。若 G 是 k- 可着色的，但不是 $(k-1)$- 可着色的，则称 k 是 G 的色数，记为 $\chi(G)$。

对任意图 G，有 $\chi(G) \leq \Delta(G)+1$，其中 $\Delta(G)$ 为最大度。

Brooks 定理

设连通图不是完全图也不是奇圈，则 $\chi(G) \leq \Delta(G)$。

证明

" 证明"

设 $|V(G)|=n$，考虑数学归纳法。

首先，$n \leq 3$ 时，命题显然成立。

接下来，假设对于 $n-1$ 时的命题成立，下面我们要逐步强化命题。

不妨只考虑 $\Delta(G)$- 正则图，因为对于非正则图来说，可以看作在正则图里删去一些边构成的，而这一过程并不会影响结论。

对于任意不是完全图也不是奇圈的正则图 G，任取其中一点 v，考虑子图 $H:=G-v$，由归纳假设知 $\chi(H) \leq \Delta(H)=\Delta(G)$，接下来我们只需证明在 H 中插入 v 不会影响结论即可。

令 $\Delta:=\Delta(G)$，设 H 染的 Δ 种颜色分别为 $c_1, c_2, \ldots, c_\Delta$，v 的 Δ 个邻接点为 $v_1, v_1, \ldots, v_\Delta$。不妨假设 v 的这些邻接点颜色两两不同，否则命题得证。

接下来我们设所有在 H 中染成 c_i 或 c_j 的点以及它们之间的所有边构成子图 $H_{i,j}$。不妨假设任意 2 个不同的点 v_i，v_j 一定在 $H_{i,j}$ 的同一个连通分量中，否则若在两个连通分量中的话，可以交换其中一个连通分量所有点的颜色，从而 v_i，v_j 颜色相同。

> **" 证明"**
>
> 这里的交换颜色指的是若图中只有两种颜色 a，b，那么把图中原来染成颜色 a 的点全部染成颜色 b，把图中原来染成颜色 b 的点全部染成颜色 a。

我们设上述连通分量为 $C_{i,j}$，那么 $C_{i,j}$ 一定只能是 v_i 到 v_j 的路。因为 v_i 在 H 中的度为 $\Delta-1$，所以 v_i 在 H 中的邻接点颜色一定两两不同，否则可以给 v_i 染别的颜色，从而和 v 的其他邻接点颜色重复，所以 v_i 在 $C_{i,j}$ 中邻接点数量为 1，v_j 同理。然后我们在 $C_{i,j}$ 中取一条 v_i 到 v_j 的路，令其为 P，若 $C_{i,j} \neq P$，那么我们沿着 P 顺次给路上的点染色，设遇到的第一个度数大于 2 的点为 u，注意到 u 的邻接点最多只用了 $\Delta-2$ 种颜色，所以 u 可以重新染色，从而使 v_i，v_j 不连通。

然后我们不难发现，对任意 3 个不同的点 v_i，v_j，v_k，$V(C_{i,j}) \cap V(C_{j,k})=\{v_j\}$。

到这里我们对命题的强化工作就已经做完了。

接下来就很简单。首先，如果 v 的邻接点两两相邻，那么命题得证。不妨设 v_1，v_2 不相邻，在 $C_{1,2}$ 中取 v_1 的邻接点 w，交换 $C_{1,3}$ 中的颜色。得到的新图中，$w \in V(C_{1,2}) \cap V(C_{2,3})$，矛盾。

至此命题证明完毕。

Welsh–Powell 算法

Welsh–Powell 算法是一种在**不限制最大着色数**时寻找着色方案的贪心算法。

对于无自环无向图 G，设 $V(G) := \{v_1, v_2, \dots, v_n\}$ 满足。

$\deg(v_i) \geq \deg(v_{i+1})$，$\forall 1 \leq i \leq n-1$

按 Welsh–Powell 算法着色后的颜色数至多为 $\max_{i=1}^{n} \min\{\deg(v_i)+1, i\}$，该算法的时间复杂度为 $O\left(n \max_{i=1}^{n} \min\{\deg(v_i) +\right.$ $O(n^2)$。

过程

1. 将当前未着色的点按度数降序排列。
2. 将第一个点染成一个未被使用的颜色。
3. 顺次遍历接下来的点，若当前点和所有与第一个点颜色**相同**的点**不相邻**，则将该点染成与第一个点相同的颜色。
4. 若仍有未着色的点，则回到步骤 1，否则结束。

示例如下：

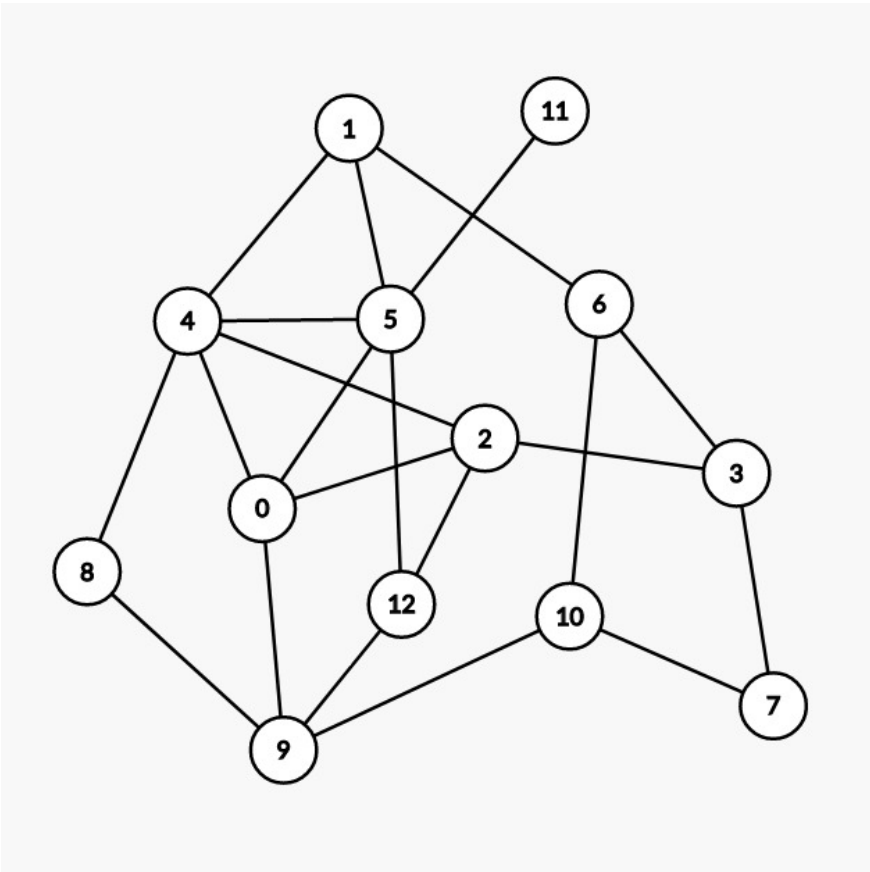


图 11.81 Orignal

（由 Graph Editor[1] 生成）

我们先对点按度数降序排序，得：

次序	1	2	3	4	5	6	7	8	9	10	11	12	13
点的编号	4	5	0	2	9	1	3	6	10	12	7	8	11
度数	5	5	4	4	4	3	3	3	3	3	2	2	1
$\min\{\deg(v_i)+1, i\}$	1	2	3	4	5	4	4	4	4	4	3	3	2

所以 Welsh–Powell 算法着色后的颜色数最多为 5。

另外因为该图有子图 C_3, 所以色数一定大于等于 3。

- 第一次染色:

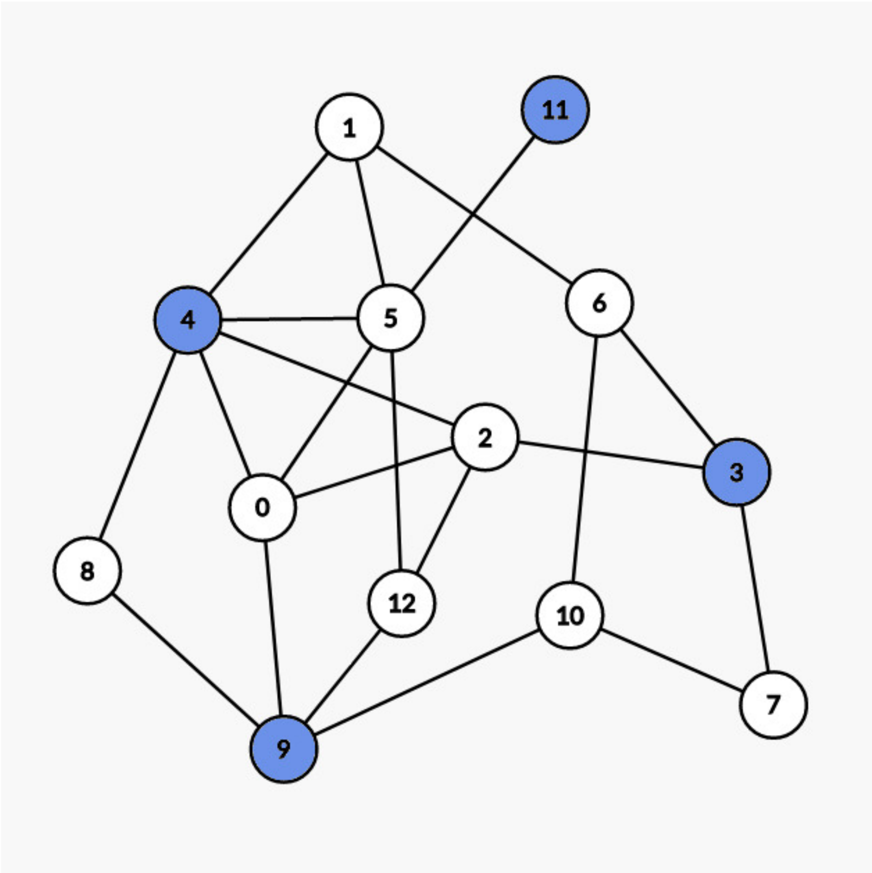


图 11.82 Colored 1

  染 `4 9 3 11` 号点。
- 第二次染色:
  染 `5 2 6 7 8` 号点。
- 第三次染色:
  染 `0 1 10 12` 号点。

证明

" 证明"

对于无自环无向图 G，设 $V(G) := \{v_1, v_2, \dots, v_n\}$ 满足

$\deg(v_i) \geq \deg(v_{i+1})$, $\forall 1 \leq i \leq n-1$

令 $V_0 = \emptyset$, 我们取 $V(G) \ \bigcup_{i=0}^{m-1} V_i$ 中的子集 V_m, 其中的元素满足

1. $v_{k_m} \in V_m$, 其中 $k_m = \min\{k : v_k \notin \bigcup_{i=0}^{m-1} V_i\}$
2. 若
   $\{v_{i_{m,1}}, v_{i_{m,2}}, \dots, v_{i_{m,l_m}}\} \subset V_m$, $i_{m,1} < i_{m,2} < \dots < i_{m,l_m}$
   则 $v_j \in V_m$ 当且仅当
   (a) $j > i_{m,l_m}$
   (b) v_j 与 $v_{i_{m,1}}, v_{i_{m,2}}, \dots, v_{i_{m,l_m}}$ 均不相邻

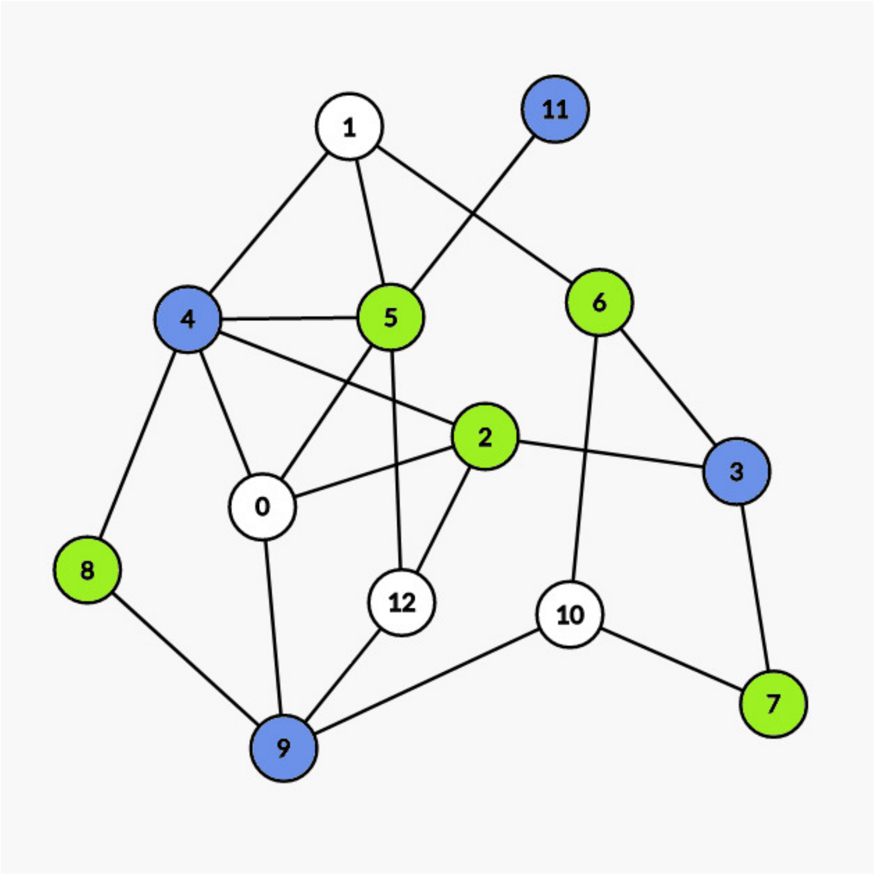


图 11.83 Colored 2

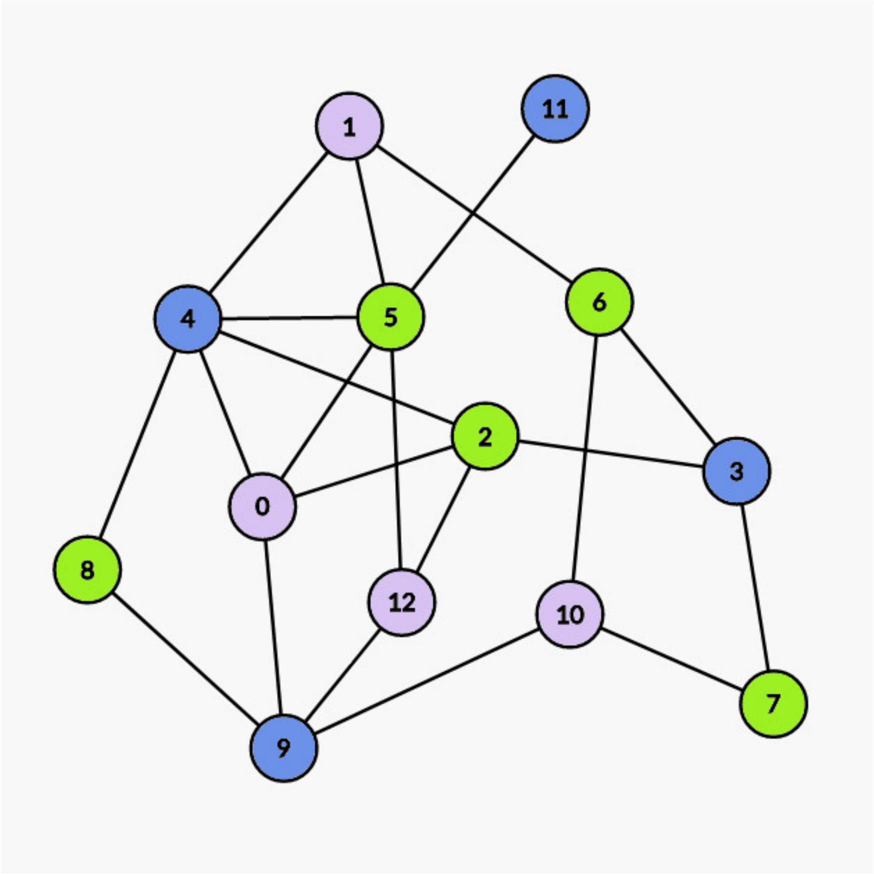


图 11.84 Colored 3

显然若将 V_i 中的点染成第 i 种颜色，则该染色方案即为 Welsh–Powell 算法给出的方案，显然有

- $V_1 \neq \emptyset$
- $V_i \cap V_j = \emptyset \iff i \neq j$
- $\exists \alpha(G) \in \mathbb{N}^*, \forall i > \alpha(G),\ s.t.\ V_i = \emptyset$

我们只需要证明：
$\bigcup_{i=1}^{\alpha(G)} V_i = V(G)$
其中
$\chi(G) \leq \alpha(G) \leq \max_{i=1}^{n} \min\{\deg(v_i) + 1, i\}$
上式左边的不等号显然成立，我们考虑右边。
首先我们不难得出：
若 $v \notin \bigcup_{i=1}^{m} V_i$，则 v 与 $V_1, V_2, \dots, V_m$ 中分别至少有一个点相邻，从而有 $\deg(v) \geq m$
进而
$v_j \in \bigcup_{i=1}^{\deg(v_j)+1} V_i$
另一方面，基于序列 $\{V_i\}$ 的构造方法，我们不难发现
$v_j \in \bigcup_{i=1}^{j} V_i$
两式结合即得证。

边着色

对无向图的边着色，要求相邻的边涂不同种颜色。若 G 是 k- 边可着色的，但不是 $(k-1)$- 边可着色的，则称 k 是 G 的边色数，记为 $\chi'(G)$。

Vizing 定理

设 G 是简单图，则 $\Delta(G) \leq \chi'(G) \leq \Delta(G) + 1$

若 G 是二部图，则 $\chi'(G) = \Delta(G)$

当 n 为奇数（$n \neq 1$）时，$\chi'(K_n) = n$; 当 n 为偶数时，$\chi'(K_n) = n - 1$

二分图 Vizing 定理的构造性证明

" 证明"

按照顺序在二分图中加边。

我们在尝试加入边 (x, y) 的时候，我们尝试寻找对于 x 和 y 的编号最小的尚未被使用过的颜色，假设分别为 l_x 和 l_y。

如果 $l_x = l_y$ 此时我们可以直接将这条边的颜色设置为 l_x。

否则假设 $l_x < l_y$, 我们可以尝试将节点 y 连出去的颜色为 l_x 的边的颜色修改为 l_y。

修改的过程可以被近似的看成是一条从 y 出发，依次经过颜色为 $l_x, l_y, \cdots$ 的边的有限唯一增广路。

因为增广路有限所以我们可以将增广路上所有的边反色，即原来颜色为 l_x 的修改为 l_y，原来颜色为 l_y 的修改为 l_x。

根据二分图的性质，节点 x 不可能为增广路节点，否则与最小未使用颜色为 l_x 矛盾。

所以我们可以在增广之后直接将连接 x 和 y 的边的颜色设为 l_x。

总构造时间复杂度为 $O(nm)$。

" 示例代码 UVa10615 Rooks[2] "

```
#include <iostream>

typedef char chr;

const int maxN = 100;

int t;
int n;
chr a[maxN + 10][maxN + 10];
int c[maxN + 10][maxN + 10];
int ans;

namespace graph {
struct Vertex {
  int head;
  int deg;
  int vis[maxN + 10];
} vertex[2 * maxN + 10], e;

struct Edge {
  int head;
  int next;
  int col;
} edge[2 * maxN * maxN + 10];

int ecnt;

void init() {
  std::fill(c[0], c[0] + sizeof(c) / 4, 0);
  for (int i = 1; i <= 2 * maxN; i++) vertex[i] = e;
  for (int i = 0; i <= 2 * maxN * maxN; i++) edge[i].col = 0;
  ecnt = 1;
  ans = 0;
  return;
}

void addEdge(int tail, int head) {
  ecnt++;
  edge[ecnt].head = head;
  edge[ecnt].next = vertex[tail].head;
  vertex[tail].head = ecnt;
  vertex[tail].deg++;
  return;
}

int get(int u) {
  for (int i = 1; i <= n; i++)
    if (!vertex[u].vis[i]) return i;
}

void DFS(int u, int ori, int upd) {
  if (vertex[u].vis[ori] == 0) {
```

```
    vertex[u].vis[upd] = 0;
    return;
  }
  int e = vertex[u].vis[ori];
  int v = edge[e].head;
  DFS(v, upd, ori);
  edge[e].col = upd;
  edge[e ^ 1].col = upd;
  vertex[u].vis[upd] = e;
  vertex[v].vis[upd] = e ^ 1;
  return;
}

void AddEdge(int u, int v) {
  addEdge(u, v);
  addEdge(v, u);
  return;
}

void solve() {
  for (int u = 1; u <= n; u++) {
    for (int e = vertex[u].head; e; e = edge[e].next) {
      int v = edge[e].head;
      if (edge[e].col) continue;
      int lu = get(u);
      int lv = get(v);
      if (lu < lv) DFS(v, lu, lv);
      if (lu > lv) DFS(u, lv, lu);
      int l = std::min(lu, lv);
      vertex[u].vis[l] = e;
      vertex[v].vis[l] = e ^ 1;
      edge[e].col = l;
      edge[e ^ 1].col = l;
    }
  }
}

void generate() {
  for (int u = 1; u <= n; u++) {
    for (int e = vertex[u].head; e; e = edge[e].next) {
      int v = edge[e].head;
      c[u][v - n] = edge[e].col;
    }
  }
  return;
}
}  // namespace graph

void mian() {
  graph::init();
  std::cin >> n;
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++) std::cin >> a[i][j];
  for (int i = 1; i <= n; i++)
```

```
    for (int j = 1; j <= n; j++)
      if (a[i][j] == '*') graph::AddEdge(i, j + n);
  for (int i = 1; i <= n * 2; i++) ans = std::max(ans, graph::vertex[i].deg);
  graph::solve();
  graph::generate();
  std::cout << ans << '\n';
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) std::cout << c[i][j] << ' ';
    std::cout << '\n';
  }
  return;
}

int main() {
  std::cin >> t;
  while (t--) mian();
  return 0;
}
```

" 一道很不简单的例题 uoj 444 二分图[3]"

本题为笔者于 2018 年命制的集训队第一轮作业题。

首先我们可以发现答案下界为度数不为 k 倍数的点的个数。

下界的构造方法是对二分图进行拆点。

若 $degree \bmod k \neq 0$, 我们将其拆为 $degree/k$ 个度数为 k 的节点和一个度数为 $degree \bmod k$ 的节点。

若 $degree \bmod k = 0$, 我们将其拆为 $degree/k$ 个度数为 k 的节点。

拆出来的点在原图中的意义相同，也就是说，在满足度数限制的情况下，一条边端点可以连接任意一个拆出来的点。

根据 Vizing 定理，我们显然可以构造出该图的一种 k 染色方案。

删边部分由于和 Vizing 定理关系不大这里不再展开。

有兴趣的读者可以自行阅读笔者当时写的题解。

色多项式

$P(G,k)$ 表示 G 的不同 k 着色方式的总数。

$P(K_n,k) = k(k-1)\cdots(k-n+1)$

$P(N_n,k) = k^n$

在无向无环图 G 中，

1. $e = (v_i, v_j) \notin E(G)$，则 $P(G,k) = P(G \cup e, k) + P(G \quad e, k)$
2. $e = (v_i, v_j) \in E(G)$，则 $P(G,k) = P(G - e, k) - P(G \quad e, k)$

定理：设 V_1 是 G 的点割集，且 $G[V_1]$ 是 G 的 $|V_1|$ 阶完全子图，$G - V_1$ 有 $p(p \geq 2)$ 个连通分支，则：

$P(G,k) = \frac{\Pi_{i=1}^{p}(P(H_i,k))}{P(G[V_1],k)^{p-1}}$

其中 $H_i = G[V_1 \cup V(G_i)]$

参考资料

1. Graph coloring - Wikipedia[4]

2. Welsh, D. J. A.; Powell, M. B. (1967), "An upper bound for the chromatic number of a graph and its application to timetabling problems[5]", The Computer Journal, 10 (1): 85–86

参考资料与注释

[1] Graph Editor

[2] UVa10615 Rooks

[3] uoj 444 二分图

[4] Graph coloring - Wikipedia

[5] An upper bound for the chromatic number of a graph and its application to timetabling problems

11.28 网络流

11.28.1 网络流简介

本页面主要介绍网络流相关的基本知识。

概述

网络（network）是指一个特殊的有向图 $G = (V, E)$，其与一般有向图的不同之处在于有容量和源汇点。

- E 中的每条边 (u, v) 都有一个被称为容量（capacity）的权值，记作 $c(u, v)$。当 $(u, v) \notin E$ 时，可以假定 $c(u, v) = 0$。
- V 中有两个特殊的点：源点（source）s 和汇点（sink）t（$s \neq t$）。

对于网络 $G = (V, E)$，流（flow）是一个从边集 E 到整数集或实数集的函数，其满足以下性质。

1. 容量限制：对于每条边，流经该边的流量不得超过该边的容量，即 $0 \leq f(u, v) \leq c(u, v)$；
2. 流守恒性：除源汇点外，任意结点 u 的净流量为 0。其中，我们定义 u 的净流量为 $f(u) = \sum_{x \in V} f(u, x) - \sum_{x \in V} f(x, u)$。

对于网络 $G = (V, E)$ 和其上的流 f，我们定义 f 的流量 $|f|$ 为 s 的净流量 $f(s)$。作为流守恒性的推论，这也等于 t 的净流量的相反数 $-f(t)$。

对于网络 $G = (V, E)$，如果 $\{S, T\}$ 是 V 的划分（即 $S \cup T = V$ 且 $S \cap T = \emptyset$），且满足 $s \in S, t \in T$，则我们称 $\{S, T\}$ 是 G 的一个 s-t 割（cut）。我们定义 s-t 割 $\{S, T\}$ 的容量为 $||S, T|| = \sum_{u \in S} \sum_{v \in T} c(u, v)$。

常见问题

常见的网络流问题包括但不限于以下类型问题。

- 最大流问题：对于网络 $G = (V, E)$，给每条边指定流量，得到合适的流 f，使得 f 的流量尽可能大。此时我们称 f 是 G 的最大流。
- 最小割问题：对于网络 $G = (V, E)$，找到合适的 s-t 割 $\{S, T\}$，使得 $\{S, T\}$ 的容量尽可能小。此时我们称 $\{S, T\}$ 是 G 的最小割。
- 最小费用最大流问题：在网络 $G = (V, E)$ 上，对每条边给定一个权值 $w(u, v)$，称为费用（cost），含义是单位流量通过 (u, v) 所花费的代价。对于 G 所有可能的最大流，我们称其中总费用最小的一者为最小费用最大流。

我们将在稍后的章节中对它们进行详细介绍。

例题：网络流 24 题

网络流 24 题是中文互联网上广泛流传的一个题单（LibreOJ[1]/洛谷[2]），至少在 2010 年前后就已经存在。该题单引入了一些经典的将其他问题建模为网络流问题的技巧。由于时代的局限性，这些问题未必是最具代表性的网络流问题，但仍值得有志于算法竞赛的读者一阅。

参考资料与注释

[1] LibreOJ

[2] 洛谷

11.28.2 最大流

本页面主要介绍最大流问题相关的算法知识。

概述

网络流基本概念参见 网络流简介。

令 $G=(V,E)$ 是一个有源汇点的网络，我们希望在 G 上指定合适的流 f，以最大化整个网络的流量 $|f|$（即 $\sum_{x\in V} f(s,x) - \sum_{x\in V} f(x,s)$），这一问题被称作最大流问题（Maximum flow problem）。

Ford–Fulkerson 增广

Ford–Fulkerson 增广是计算最大流的一类算法的总称。该方法运用贪心的思想，通过寻找增广路来更新并求解最大流。

概述

给定网络 G 及 G 上的流 f，我们做如下定义。

对于边 (u,v)，我们将其容量与流量之差称为剩余容量 $c_f(u,v)$（Residual Capacity），即 $c_f(u,v)=c(u,v)-f(u,v)$。

我们将 G 中所有结点和剩余容量大于 0 的边构成的子图称为残量网络 G_f（Residual Network），即 $G_f=(V,E_f)$，其中 $E_f=\left\{(u,v) \mid c_f(u,v)>0\right\}$。

warning

正如我们马上要提到的，流量可能是负值，因此，E_f 的边有可能并不在 E 中。引入增广的概念后，下文将具体解释这一点。

我们将 G_f 上一条从源点 s 到汇点 t 的路径称为增广路（Augmenting Path）。对于一条增广路，我们给每一条边 (u,v) 都加上等量的流量，以令整个网络的流量增加，这一过程被称为增广（Augment）。由此，最大流的求解可以被视为若干次增广分别得到的流的叠加。

此外，在 Ford–Fulkerson 增广的过程中，对于每条边 (u,v)，我们都新建一条反向边 (v,u)。我们约定 $f(u,v)=-f(v,u)$，这一性质可以通过在每次增广时引入退流操作来保证，即 $f(u,v)$ 增加时 $f(v,u)$ 应当减少同等的量。

tip

在最大流算法的代码实现中，我们往往需要支持快速访问反向边的操作。在邻接矩阵中，这一操作是 trivial 的（$g_{u,v} \leftrightarrow g_{v,u}$）。但主流的实现是更加优秀的链式前向星。其中，一个常用的技巧是，我们令边从偶数（通常为

0）开始编号，并在加边时总是紧接着加入其反向边使得它们的编号相邻。由此，我们可以令编号为 i 的边和编号为 $i \oplus 1$ 的边始终保持互为反向边的关系。

初次接触这一方法的读者可能察觉到一个违反直觉的情形——反向边的流量 $f(v,u)$ 可能是一个负值。实际上我们可以注意到，在 Ford–Fulkerson 增广的过程中，真正有意义的是剩余容量 c_f，而 $f(v,u)$ 的绝对值是无关紧要的，我们可以将反向边流量的减少视为反向边剩余容量 $c_f(v,u)$ 的增加——这也与退流的意义相吻合——反向边剩余容量的增加意味着我们接下来可能通过走反向边来和原先正向的增广抵消，代表一种「反悔」的操作。

以下案例有可能帮助你理解这一过程。假设 G 是一个单位容量的网络，我们考虑以下过程：

- G 上有多条增广路，其中，我们选择进行一次先后经过 u,v 的增广（如左图所示），流量增加 1。
- 我们注意到，如果进行中图上的增广，这个局部的最大流量不是 1 而是 2。但由于指向 u 的边和从 v 出发的边在第一次增广中耗尽了容量，此时我们无法进行中图上的增广。这意味着我们当前的流是不够优的，但局部可能已经没有其他（只经过原图中的边而不经过反向边的）增广路了。
- 现在引入退流操作。第一次增广后，退流意味着 $c_f(v,u)$ 增加了 1 剩余容量，即相当于新增 (v,u) 这条边，因此我们可以再进行一次先后经过 p,v,u,q 的增广（如右图橙色路径所示）。无向边 (u,v) 上的流量在两次增广中抵消，我们惊奇地发现两次增广叠加得到的结果实际上和中图是等价的。

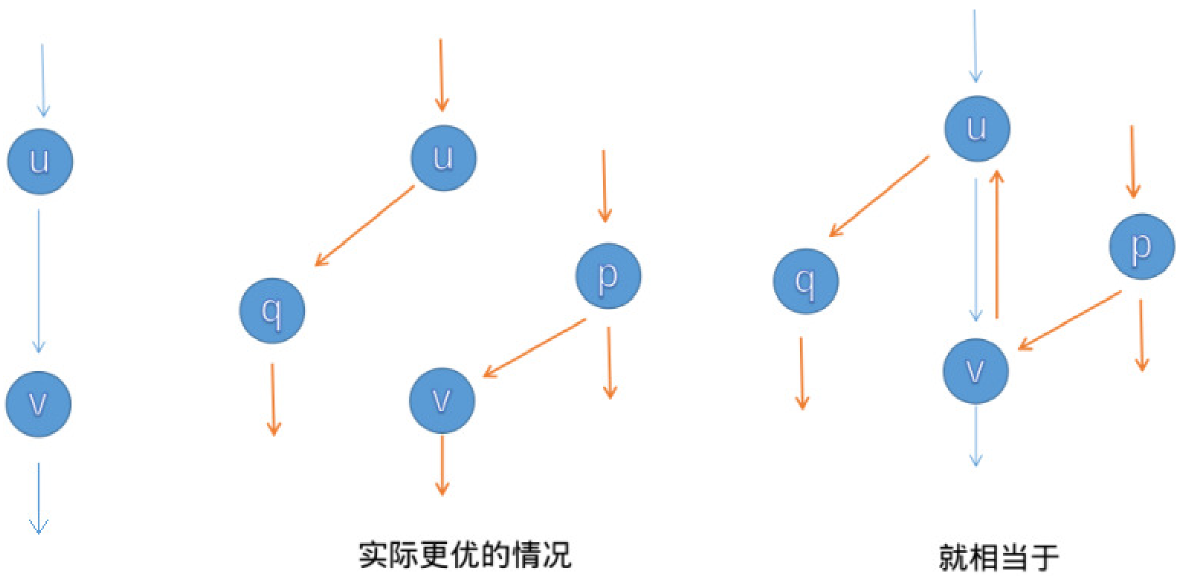


图 11.85

以上案例告诉我们，退流操作带来的「抵消」效果使得我们无需担心我们按照「错误」的顺序选择了增广路。

容易发现，只要 G_f 上存在增广路，那么对其增广就可以令总流量增加；否则说明总流量已经达到最大可能值，求解过程完成。这就是 Ford–Fulkerson 增广的过程。

最大流最小割定理

我们大致了解了 Ford–Fulkerson 增广的思想，可是如何证明这一方法的正确性呢？为什么增广结束后的流 f 是一个最大流？

实际上，Ford–Fulkerson 增广的正确性和最大流最小割定理（The Maxflow-Mincut Theorem）等价。这一定理指出，对于任意网络 $G=(V,E)$，其上的最大流 f 和最小割 $\{S,T\}$ 总是满足 $|f|=||S,T||$。

为了证明最大流最小割定理，我们先从一个引理出发：对于网络 $G=(V,E)$，任取一个流 f 和一个割 $\{S,T\}$，总是有 $|f|\le||S,T||$，其中等号成立当且仅当 $\{(u,v)|u\in S,v\in T\}$ 的所有边均满流，且 $\{(u,v)|u\in T,v\in S\}$ 的所有边均空流。

" 证明"

$$
\begin{aligned}
|f| &= f(s) \\
&= \sum_{u \in S} f(u) \\
&= \sum_{u \in S} \left(\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) \right) \\
&= \sum_{u \in S} \left(\sum_{v \in T} f(u, v) + \sum_{v \in S} f(u, v) - \sum_{v \in T} f(v, u) - \sum_{v \in S} f(v, u) \right) \\
&= \sum_{u \in S} \left(\sum_{v \in T} f(u, v) - \sum_{v \in T} f(v, u) \right) + \sum_{u \in S} \sum_{v \in S} f(u, v) - \sum_{u \in S} \sum_{v \in S} f(v, u) \\
&= \sum_{u \in S} \left(\sum_{v \in T} f(u, v) - \sum_{v \in T} f(v, u) \right) \\
&\le \sum_{u \in S} \sum_{v \in T} f(u, v) \\
&\le \sum_{u \in S} \sum_{v \in T} c(u, v) \\
&= ||S, T||
\end{aligned}
$$

```
为了取等，第一个不等号需要 $\{(u, v) \mid u \in T, v \in S\}$ 的所有边均空流，第二
个不等号需要 $\{(v, u) \mid u \in S, v \in T\}$ 的所有边均满流。原引理得证。
```

那么，对于任意网络，以上取等条件是否总是能被满足呢？如果答案是肯定的，则最大流最小割定理得证。以下我们尝试证明。

" 证明"

假设某一轮增广后，我们得到流 f 使得 G_f 上不存在增广路，即 G_f 上不存在 s 到 t 的路径。此时我们记从 s 出发可以到达的结点组成的点集为 S，并记 $T = V \ S$。

显然，$\{S, T\}$ 是 G_f 的一个割，且 $||S, T|| = \sum_{u \in S} \sum_{v \in T} c_f(u, v) = 0$。由于剩余容量是非负的，这也意味着对于任意 $u \in S, v \in T, (u, v) \in E_f$，均有 $c_f(u, v) = 0$。以下我们将这些边分为存在于原图中的边和反向边两种情况讨论：

- $(u, v) \in E$：此时，$c_f(u, v) = c(u, v) - f(u, v) = 0$，因此有 $c(u, v) = f(u, v)$，即 $\{(u, v) \mid u \in S, v \in T\}$ 的所有边均满流；
- $(v, u) \in E$：此时，$c_f(u, v) = c(u, v) - f(u, v) = 0 - f(u, v) = f(v, u) = 0$，即 $\{(v, u) \mid u \in S, v \in T\}$ 的所有边均空流。

因此，增广停止后，上述流 f 满足取等条件。根据引理指出的大小关系，自然地，f 是 G 的一个最大流，$\{S, T\}$ 是 G 的一个最小割。

容易看出，Kőnig 定理是最大流最小割定理的特殊情形。实际上，它们都和线性规划中的对偶有关。

时间复杂度分析

在整数流量的网络 $G = (V, E)$ 上，平凡地，我们假设每次增广的流量都是整数，则 Ford–Fulkerson 增广的时间复杂度的一个上界是 $O(|E||f|)$，其中 f 是 G 上的最大流。这是因为单轮增广的时间复杂度是 $O(|E|)$，而增广会导致总流量增加，故增广轮数不可能超过 $|f|$。

对于 Ford–Fulkerson 增广的不同实现，时间复杂度也各不相同。其中较主流的实现有 Edmonds–Karp, Dinic, SAP, ISAP 等算法，我们将在下文中分别介绍。

Edmonds–Karp 算法

算法思想

如何在 G_f 中寻找增广路呢？当我们考虑 Ford–Fulkerson 增广的具体实现时，最自然的方案就是使用 BFS。此时，Ford–Fulkerson 增广表现为 Edmonds–Karp 算法。其具体流程如下：

- 如果在 G_f 上我们可以从 s 出发 BFS 到 t，则我们找到了新的增广路。
- 对于增广路 p，我们计算出 p 经过的边的剩余容量的最小值 $\Delta = \min_{(u,v)\in p} c_f(u,v)$。我们给 p 上的每条边都加上 Δ 流量，并给它们的反向边都退掉 Δ 流量，令最大流增加了 Δ。
- 因为我们修改了流量，所以我们得到新的 G_f，我们在新的 G_f 上重复上述过程，直至增广路不存在，则流量不再增加。

以上算法即 Edmonds–Karp 算法。

时间复杂度分析

接下来让我们尝试分析 Edmonds–Karp 算法的时间复杂度。

显然，单轮 BFS 增广的时间复杂度是 $O(|E|)$。

增广总轮数的上界是 $O(|V||E|)$。这一论断在网络资料中常被伪证（或被含糊其辞略过）。以下我们尝试给出一个较正式的证明[1]。

" 增广总轮数的上界的证明"

首先，我们引入一个引理——最短路非递减引理。具体地，我们记 $d_f(u)$ 为 G_f 上结点 u 到源点 s 的距离（即最短路长度，下同）。对于某一轮增广，我们用 f 和 f' 分别表示增广前的流和增广后的流，我们断言，对于任意结点 u，增广总是使得 $d_{f'}(u) \geq d_f(u)$。我们将在稍后证明这一引理。

不妨称增广路上剩余容量最小的边是饱和边（存在多条边同时最小则取任一）。如果一条有向边 (u,v) 被选为饱和边，增广会清空其剩余容量导致饱和边的消失，并且退流导致反向边的新增（如果原先反向边不存在），即 $(u,v)\notin E_{f'}$ 且 $(v,u)\in E_{f'}$。以上分析使我们知道，对于无向边 (u,v)，其被增广的两种方向总是交替出现。

在 G_f 上沿 (u,v) 增广时，$d_f(u)+1=d_f(v)$，此后残量网络变为 $G_{f'}$。在 $G_{f'}$ 上沿 (v,u) 增广时，$d_{f'}(v)+1=d_{f'}(u)$。根据最短路非递减引理又有 $d_{f'}(v)\geq d_f(v)$，我们连接所有式子，得到 $d_{f'}(u)\geq d_f(u)+2$。换言之，如果有向边 (u,v) 被选为饱和边，那么与其上一次被选为饱和边时相比，u 到 s 的距离至少增加 2。

s 到任意结点的距离不可能超过 $|V|$，结合上述性质，我们发现每条边被选为饱和边的次数是 $O(|V|)$ 的，与边数相乘后得到增广总轮数的上界 $O(|V||E|)$。

接下来我们证明最短路非递减引理，即 $d_{f'}(u)\geq d_f(u)$。这一证明并不难，但可能稍显绕口，读者可以停下来认真思考片刻。

" 最短路非递减引理的证明"

考虑反证。对于某一轮增广，我们假设存在若干结点，它们在该轮增广后到 s 的距离较增广前减小。我们记 v 为其中到 s 的距离最小的一者（即 $v=\arg\min_{x\in V, d_{f'}(x)<d_f(x)} d_{f'}(x)$）。注意，根据反证假设，此时 $d_{f'}(v)<d_f(v)$ 是已知条件。

```
在 $G_{f'}$ 中 $s$ 到 $v$ 的最短路上，我们记 $u$ 是 $v$ 的上一个结点，即 $d_{f'}(u
) + 1 = d_{f'}(v)$。

为了不让 $u$ 破坏 $v$ 的「距离最小」这一性质，$u$ 必须满足 $d_{f'}(u) \geq d_f(u)$。

对于上式，我们令不等号两侧同加，得 $d_{f'}(v) \geq d_f(u) + 1$。根据反证假设进行放缩
，我们得到 $d_f(v) > d_f(u) + 1$。

以下我们尝试讨论 $(u, v)$ 上的增广方向。
```

```
- 假设有向边 $(u, v) \in E_f$。根据 BFS「广度优先」的性质，我们有 $d_f(u) + 1 \geq d
_f(v)$。该式与放缩结果冲突，导出矛盾。
- 假设有向边 $(u, v) \not \in E_f$。根据 $u$ 的定义我们已知 $(u, v) \in E_{f'}$，因
此这条边的存在必须是当前轮次的增广经过了 $(v, u)$ 并退流产生反向边的结果，也即 $d
_f(v) + 1 = d_f(u)$。该式与放缩结果冲突，导出矛盾。

由于 $(u, v)$ 沿任何方向增广都会导出矛盾，我们知道反证假设不成立，最短路非递减引理
得证。
```

将单轮 BFS 增广的复杂度与增广轮数的上界相乘，我们得到 Edmonds–Karp 算法的时间复杂度是 $O(|V||E|^2)$。

代码实现

Edmonds–Karp 算法的可能实现如下。

" 参考代码"

```cpp
#define maxn 250
#define INF 0x3f3f3f3f

struct Edge {
  int from, to, cap, flow;

  Edge(int u, int v, int c, int f) : from(u), to(v), cap(c), flow(f) {}
};

struct EK {
  int n, m;             // n：点数，m：边数
  vector<Edge> edges;   // edges：所有边的集合
  vector<int> G[maxn];  // G：点 x -> x 的所有边在 edges 中的下标
  int a[maxn], p[maxn];  // a：点 x -> BFS 过程中最近接近点 x 的边给它的最大流
                         // p：点 x -> BFS 过程中最近接近点 x 的边

  void init(int n) {
    for (int i = 0; i < n; i++) G[i].clear();
    edges.clear();
  }

  void AddEdge(int from, int to, int cap) {
    edges.push_back(Edge(from, to, cap, 0));
    edges.push_back(Edge(to, from, 0, 0));
    m = edges.size();
    G[from].push_back(m - 2);
    G[to].push_back(m - 1);
  }

  int Maxflow(int s, int t) {
    int flow = 0;
    for (;;) {
      memset(a, 0, sizeof(a));
      queue<int> Q;
      Q.push(s);
      a[s] = INF;
      while (!Q.empty()) {
        int x = Q.front();
```

```
      Q.pop();
      for (int i = 0; i < G[x].size(); i++) {  // 遍历以 x 作为起点的边
        Edge& e = edges[G[x][i]];
        if (!a[e.to] && e.cap > e.flow) {
          p[e.to] = G[x][i];  // G[x][i] 是最近接近点 e.to 的边
          a[e.to] =
              min(a[x], e.cap - e.flow);  // 最近接近点 e.to 的边赋给它的流
          Q.push(e.to);
        }
      }
      if (a[t]) break;  // 如果汇点接受到了流，就退出 BFS
    }
    if (!a[t])
      break;  // 如果汇点没有接受到流，说明源点和汇点不在同一个连通分量上
    for (int u = t; u != s;
         u = edges[p[u]].from) {  // 通过 u 追寻 BFS 过程中 s -> t 的路径
      edges[p[u]].flow += a[t];      // 增加路径上边的 flow 值
      edges[p[u] ^ 1].flow -= a[t];  // 减小反向路径的 flow 值
    }
    flow += a[t];
  }
  return flow;
}
};
```

Dinic 算法

算法思想

考虑在增广前先对 G_f 做 BFS 分层，即根据结点 u 到源点 s 的距离 $d(u)$ 把结点分成若干层。令经过 u 的流量只能流向下一层的结点 v，即删除 u 向层数标号相等或更小的结点的出边，我们称 G_f 剩下的部分为层次图（Level Graph）。形式化地，我们称 $G_L = (V, E_L)$ 是 $G_f = (V, E_f)$ 的层次图，其中 $E_L = \left\{(u,v) \mid (u,v) \in E_f, d(u)+1 = d(v)\right\}$。

如果我们在层次图 G_L 上找到一个最大的增广流 f_b，使得仅在 G_L 上是不可能找出更大的增广流的，则我们称 f_b 是 G_L 的阻塞流（Blocking Flow）。

> warning
>
> 尽管在上文中我们仅在单条增广路上定义了增广/增广流，广义地，「增广」一词不仅可以用于单条路径上的增广流，也可以用于若干增广流的并——后者才是我们定义阻塞流时使用的意义。

定义层次图和阻塞流后，Dinic 算法的流程如下。

1. 在 G_f 上 BFS 出层次图 G_L。
2. 在 G_L 上 DFS 出阻塞流 f_b。
3. 将 f_b 并到原先的流 f 中，即 $f \leftarrow f + f_b$。
4. 重复以上过程直到不存在从 s 到 t 的路径。

此时的 f 即为最大流。

在分析这一算法的复杂度之前，我们需要特别说明「在 G_L 上 DFS 出阻塞流 f_b」的过程。尽管 BFS 层次图对于本页面的读者应当是 trivial 的，但 DFS 阻塞流的过程则稍需技巧——我们需要引入当前弧优化。

注意到在 G_L 上 DFS 的过程中，如果结点 u 同时具有大量入边和出边，并且 u 每次接受来自入边的流量时都遍历出边表来决定将流量传递给哪条出边，则 u 这个局部的时间复杂度最坏可达 $O(|E|^2)$。为避免这一缺陷，如果某一时刻我们已经知道边 (u,v) 已经增广到极限（边 (u,v) 已无剩余容量或 v 的后侧已增广至阻塞），则 u 的流量没有必要再尝试流向出边 (u,v)。据此，对于每个结点 u，我们维护 u 的出边表中第一条还有必要尝试的出边。习惯上，我们称维护的这个指针为当前弧，称这个做法为当前弧优化。

" 多路增广"

多路增广是 Dinic 算法的一个常数优化——如果我们在层次图上找到了一条从 s 到 t 的增广路 p，则接下来我们未必需要重新从 s 出发找下一条增广路，而可能从 p 上最后一个仍有剩余容量的位置出发寻找一条岔路进行增广。考虑到其与回溯形式的一致性，这一优化在 DFS 的代码实现中也是自然的。

" 常见误区"

可能是由于大量网络资料的错误表述引发以讹传讹的情形，相当数量的选手喜欢将当前弧优化和多路增广并列称为 Dinic 算法的两种优化。实际上，当前弧优化是用于保证 Dinic 时间复杂度正确性的一部分，而多路增广只是一个不影响复杂度的常数优化。

时间复杂度分析

应用当前弧优化后，对 Dinic 算法的时间复杂度分析如下。

首先，我们尝试证明单轮增广中 DFS 求阻塞流的时间复杂度是 $O(|V||E|)$。

" 单轮增广的时间复杂度的证明"

考虑阻塞流 f_b 中的每条增广路，它们都是在 G_L 上每次沿当前弧跳转而得到的结果，其中每条增广路经历的跳转次数不可能多于 $|V|$。

每找到一条增广路就有一条饱和边消失（剩余容量清零）。考虑阻塞流 f_b 中的每条增广路，我们将被它们清零的饱和边形成的边集记作 E_1。考虑到 G_L 分层的性质，饱和边消失后其反向边不可能在同一轮增广内被其他增广路经过，因此，E_1 是 E_L 的子集。

此外，对于沿当前弧跳转但由于某个位置阻塞所以没有成功得到增广路的情形，我们将这些不完整的路径上的最后一条边形成的边集记作 E_2。E_2 的成员不饱和，所以 E_1 与 E_2 不交，且 $E_1 \cup E_2$ 仍是 E_L 的子集。

由于 $E_1 \cup E_2$ 的每个成员都没有花费超过 $|V|$ 次跳转（且在使用多路增广优化后一些跳转将被重复计数），因此，综上所述，DFS 过程中的总跳转次数不可能多于 $|V||E_L|$。

" 常见伪证一则"

对于每个结点，我们维护下一条可以增广的边，而当前弧最多变化 $|E|$ 次，从而单轮增广的最坏时间复杂度为 $O(|V||E|)$。

bug

「当前弧最多变化 $|E|$ 次」并不能推得「每个结点最多访问其出边 $|E|$ 次」。这是因为，访问当前弧并不一定耗尽上面的剩余容量，结点 u 可能多次访问同一条当前弧。

注意到层次图的层数显然不可能超过 $|V|$，如果我们可以证明层次图的层数在增广过程中严格单增，则 Dinic 算法的增广轮数是 $O(|V|)$ 的。接下来我们尝试证明这一结论[2]。

" 层次图层数单调性的证明"

我们需要引入预流推进类算法（另一类最大流算法）中的一个概念——高度标号。为了更方便地结合高度标号表述我们的证明，在证明过程中，我们令 $d_f(u)$ 为 G_f 上结点 u 到**汇点** t 的距离，从**汇点**而非源点出发进行分层（这并没有本质上的区别）。对于某一轮增广，我们用 f 和 f' 分别表示增广前的流和增广后的流。在该轮增广中求解并加入阻塞流后，记层次图由 $G_L = (V, E_L)$ 变为 $G'_L = (V, E'_L)$。

我们给高度标号一个不严格的临时定义——在网络 $G = (V, E)$ 上，令 h 是点集 V 到整数集 N 上的函数，h 是 G 上合法的高度标号当且仅当 $h(u) \leq h(v) + 1$ 对于 $(u, v) \in E$ 恒成立。

考察所有 $E_{f'}$ 的成员 (u, v)，我们发现 $(u, v) \in E_{f'}$ 的原因是以下二者之一。

- $(u, v) \in E_f$，且剩余容量在该轮增广过程中未耗尽——根据最短路的定义，此时我们有 $d_f(u) \leq d_f(v) + 1$；
- $(u, v) \notin E_f$，但在该轮增广过程中阻塞流经过 (v, u) 并退流产生反向边——根据层次图和阻塞流的定义，此

时我们有 $d_f(u)+1=d_f(v)$。

以上观察让我们得出一个结论——d_f 在 $G_{f'}$ 上是一个合法的高度标号。当然，在 $G_{f'}$ 的子图 G'_L 上也是。

现在，对于一条 G'_L 上的增广路 $p=(s,\dots,u,v,\dots,t)$，按照 p 上结点的反序（从 t 到 s 的顺序）考虑从空路径开始每次添加一个结点的过程。假设结点 v 已加入，结点 u 正在加入，我们发现，加入结点 u 后，根据层次图的定义，$d_{f'}(u)$ 的值较 $d_{f'}(v)$ 增加 1；与此同时，由于 d_f 是 G'_L 上的高度标号，$d_f(u)$ 的值既可能较 $d_f(v)$ 增加 1，也可能保持不变或减少。因此，在整条路径被添加完成后，我们得到 $d_{f'}(s)\geq d_f(s)$，其中取等的充要条件是 $d_f(u)=d_f(v)+1$ 对于 $(u,v)\in p$ 恒成立。如果该不等式不能取等，则有 $d_{f'}(s)>d_f(s)$——即我们想要的结论「层次图的层数在增广过程中严格单增」。以下我们尝试证明该不等式不能取等。

考虑反证，我们假设 $d_{f'}(s)=d_f(s)$ 成立，并尝试导出矛盾。现在我们断言，在 G'_L 上，p 至少包含一条边 (u,v) 满足 (u,v) 在 G_L 上不存在。如果没有这样的边，考虑到 $d_f(s)=d_{f'}(s)$，结合层次图和阻塞流的定义，G_L 上的增广应尚未完成。为了不产生以上矛盾，我们的断言只好是正确的。

令 (u,v) 是满足断言条件的那条边，其满足断言的原因只能是以下二者之一。

- $(u,v)\in E_f$ 但 $d_f(u)\leq d_f(v)+1$ 未取等，故根据层次图的定义可知 $(u,v)\notin E_L$，并在增广后新一轮重分层中被加入到 E'_L 中；
- $(u,v)\notin E_f$，这意味着 (u,v) 这条边的产生是当前轮次增广中阻塞流经过 (v,u) 并退流产生反向边的结果，也即 $d_f(u)=d_f(v)-1$。

由于我们无论以何种方式满足断言均得到 $d_f(u)\neq d_f(v)+1$，也即 $d_{f'}(s)\geq d_f(s)$ 取等的充要条件无法被满足，这与反证假设 $d_{f'}(s)=d_f(s)$ 冲突，原命题得证。

" 常见伪证另一则"

考虑反证。假设层次图的层数在一轮增广结束后较原先相等，则层次图上应仍存在至少一条从 s 到 t 的增广路满足相邻两点间的层数差为 1。这条增广路未被增广说明该轮增广尚未结束。为了不产生上述矛盾，原命题成立。

bug

「一轮增广结束后新的层次图上 s-t 最短路较原先相等」并不能推得「旧的层次图上该轮增广尚未结束」。这是因为，没有理由表明两张层次图的边集相同，新的层次图上的 s-t 最短路有可能经过旧的层次图上不存在的边。

将单轮增广的时间复杂度 $O(|V||E|)$ 与增广轮数 $O(|V|)$ 相乘，Dinic 算法的时间复杂度是 $O(|V|^2|E|)$。

如果需要令 Dinic 算法的实际运行时间接近其理论上界，我们需要构造有特殊性质的网络作为输入。由于在算法竞赛实践中，对于网络流知识相关的考察常侧重于将原问题建模为网络流问题的技巧。此时，我们的建模通常不包含令 Dinic 算法执行缓慢的特殊性质；恰恰相反，Dinic 算法在大部分图上效率非常优秀。因此，网络流问题的数据范围通常较大，「将 $|V|,|E|$ 的值代入 $|V|^2|E|$ 以估计运行时间」这一方式并不适用。实际上，进行准确的估计需要选手对 Dinic 算法的实际效率有一定的经验，读者可以多加练习。

特殊情形下的时间复杂度分析

在一些性质良好的图上，Dinic 算法有更好的时间复杂度。

对于网络 $G=(V,E)$，如果其所有边容量均为 1，即 $c(u,v)\in\{0,1\}$ 对于 $(u,v)\in E$ 恒成立，则我们称 G 是单位容量（Unit Capacity）的。

在单位容量的网络中，Dinic 算法的单轮增广的时间复杂度为 $O(|E|)$。

" 证明"

这是因为，每次增广都会导致增广路上的所有边均饱和并消失，故单轮增广中每条边只能被增广一次。

在单位容量的网络中，Dinic 算法的增广轮数是 $O(|E|^{\frac{1}{2}})$ 的。

" 证明"

以源点 s 为中心分层，记 $d_f(u)$ 为 G_f 上结点 u 到源点 s 的距离。另外，我们定义将点集 $\{u \mid u \in V, d_f(u) = k\}$ 定义为编号为 k 的层次 D_k，并记 $S_k = \cup_{i \le k} D_i$。

假设我们已经进行了 $|E|^{\frac{1}{2}}$ 轮增广。根据鸽巢原理，至少存在一个 k 满足边集 $\{(u,v) \mid u \in D_k, v \in D_{k+1}, (u,v) \in E_f\}$ 的大小不超过 $\frac{|E|}{|E|^{\frac{1}{2}}} \approx |E|^{\frac{1}{2}}$。显然，$\{S_k, V - S_k\}$ 是 G_f 上的 s-t 割，且其割容量不超过 $|E|^{\frac{1}{2}}$。根据最大流最小割定理，G_f 上的最大流不超过 $|E|^{\frac{1}{2}}$，也即 G_f 上最多还能执行 $|E|^{\frac{1}{2}}$ 轮增广。因此，总增广轮数是 $O(|E|^{\frac{1}{2}})$ 的。

在单位容量的网络中，Dinic 算法的增广轮数是 $O(|V|^{\frac{2}{3}})$ 的。

" 证明"

假设我们已经进行了 $2|V|^{\frac{2}{3}}$ 轮增广。由于至多有半数的（$|V|^{\frac{2}{3}}$ 个）层次包含多于 $|V|^{\frac{1}{3}}$ 个点，故无论我们如何分配所有层次的大小，至少存在一个 k 满足相邻两个层次同时包含不多于 $|V|^{\frac{1}{3}}$ 个点，即 $|D_k| \le |V|^{\frac{1}{3}}$ 且 $|D_{k+1}| \le |V|^{\frac{1}{3}}$。

为最大化 D_k 和 D_{k+1} 之间的边数，我们假定这是一个完全二分图，此时边集 $\{(u,v) \mid u \in D_k, v \in D_{k+1}, (u,v) \in E_f\}$ 的大小不超过 $|V|^{\frac{2}{3}}$。显然，$\{S_k, V - S_k\}$ 是 G_f 上的 s-t 割，且其割容量不超过 $|V|^{\frac{2}{3}}$。根据最大流最小割定理，G_f 上的最大流不超过 $|V|^{\frac{2}{3}}$，也即 G_f 上最多还能执行 $|V|^{\frac{2}{3}}$ 轮增广。因此，总增广轮数是 $O(|V|^{\frac{2}{3}})$ 的。

在单位容量的网络中，如果除源汇点外每个结点 u 都满足 $deg_{in}(u) = 1$ 或 $deg_{out}(u) = 1$，则 Dinic 算法的增广轮数是 $O(|V|^{\frac{1}{2}})$ 的。其中，$deg_{in}(u)$ 和 $deg_{out}(u)$ 分别代表结点 u 的入度和出度。

" 证明"

我们引入以下引理——对于这一形式的网络，其上的任意流总是可以分解成若干条单位流量的、**点不交**的增广路。

假设我们已经进行了 $|V|^{\frac{1}{2}}$ 轮增广。根据层次图的定义，此时任意新的增广路的长度至少为 $|V|^{\frac{1}{2}}$。

考虑 G_f 上的最大流的增广路分解，我们得到的增广路的数量不能多于 $\frac{|V|}{|V|^{\frac{1}{2}}} \approx |V|^{\frac{1}{2}}$，这意味着 G_f 上最多还能执行 $|V|^{\frac{1}{2}}$ 轮增广。因此，总增广轮数是 $O(|V|^{\frac{1}{2}})$ 的。

综上，我们得出一些推论。

- 在单位容量的网络上，Dinic 算法的总时间复杂度是 $O(|E| \min(|E|^{\frac{1}{2}}, |V|^{\frac{2}{3}}))$。
- 在单位容量的网络上，如果除源汇点外每个结点 u 都满足 $deg_{in}(u) = 1$ 或 $deg_{out}(u) = 1$，Dinic 算法的总时间复杂度是 $O(|E||V|^{\frac{1}{2}})$。对于二分图最大匹配问题，我们常使用 Hopcroft–Karp 算法解决，而这一算法实际上是 Dinic 算法在满足上述度数限制的单位容量网络上的特例。

代码实现

" 参考代码"

```
struct MF {
  struct edge {
    int v, nxt, cap, flow;
  } e[N];

  int fir[N], cnt = 0;

  int n, S, T;
  ll maxflow = 0;
  int dep[N], cur[N];
```

```
void init() {
  memset(fir, -1, sizeof fir);
  cnt = 0;
}

void addedge(int u, int v, int w) {
  e[cnt] = {v, fir[u], w, 0};
  fir[u] = cnt++;
  e[cnt] = {u, fir[v], 0, 0};
  fir[v] = cnt++;
}

bool bfs() {
  queue<int> q;
  memset(dep, 0, sizeof(int) * (n + 1));

  dep[S] = 1;
  q.push(S);
  while (q.size()) {
    int u = q.front();
    q.pop();
    for (int i = fir[u]; ~i; i = e[i].nxt) {
      int v = e[i].v;
      if ((!dep[v]) && (e[i].cap > e[i].flow)) {
        dep[v] = dep[u] + 1;
        q.push(v);
      }
    }
  }
  return dep[T];
}

int dfs(int u, int flow) {
  if ((u == T) || (!flow)) return flow;

  int ret = 0;
  for (int& i = cur[u]; ~i; i = e[i].nxt) {
    int v = e[i].v, d;
    if ((dep[v] == dep[u] + 1) &&
        (d = dfs(v, min(flow - ret, e[i].cap - e[i].flow)))) {
      ret += d;
      e[i].flow += d;
      e[i ^ 1].flow -= d;
      if (ret == flow) return ret;
    }
  }
  return ret;
}

void dinic() {
  while (bfs()) {
    memcpy(cur, fir, sizeof(int) * (n + 1));
    maxflow += dfs(S, INF);
  }
```

```
  }
} mf;
```

MPM 算法

MPM(Malhotra, Pramodh-Kumar and Maheshwari) 算法得到最大流的方式有两种：使用基于堆的优先队列，时间复杂度为 $O(n^3\log n)$；常用 BFS 解法，时间复杂度为 $O(n^3)$。注意，本章节只专注于分析更优也更简洁的 $O(n^3)$ 算法。

MPM 算法的整体结构和 Dinic 算法类似，也是分阶段运行的。在每个阶段，在 G 的残量网络的分层网络中找到增广路。它与 Dinic 算法的主要区别在于寻找增广路的方式不同：MPM 算法中寻找增广路的部分的只花了 $O(n^2)$，时间复杂度要优于 Dinic 算法。

MPM 算法需要考虑顶点而不是边的容量。在分层网络 L 中，如果定义点 v 的容量 $p(v)$ 为其传入残量和传出残量的最小值，则有：

$$p_{in}(v) = \sum_{(u,v)\in L}(c(u,v) - f(u,v))$$

$$p_{out}(v) = \sum_{(v,u)\in L}(c(v,u) - f(v,u))$$

$$p(v) = \min(p_{in}(v), p_{out}(v))$$

我们称节点 r 是参考节点当且仅当 $p(r) = \min p(v)$。对于一个参考节点 r，我们一定可以让经过 r 的流量增加 $p(r)$ 以使其容量变为 0。这是因为 L 是有向无环图且 L 中节点容量至少为 $p(r)$，所以我们一定能找到一条从 s 经过 r 到达 t 的有向路径。那么我们让这条路上的边流量都增加 $p(r)$ 即可。这条路即为这一阶段的增广路。寻找增广路可以用 BFS。增广完之后所有满流边都可以从 L 中删除，因为它们不会在此阶段后被使用。同样，所有与 s 和 t 不同且没有出边或入边的节点都可以删除。

时间复杂度分析

MPM 算法的每个阶段都需要 $O(V^2)$，因为最多有 V 次迭代（因为至少删除了所选的参考节点），并且在每次迭代中，我们删除除最多 V 之外经过的所有边。求和，我们得到 $O(V^2+E)=O(V^2)$。由于阶段总数少于 V，因此 MPM 算法的总运行时间为 $O(V^3)$。

阶段总数小于 V 的证明

MPM 算法在少于 V 个阶段内结束。为了证明这一点，我们必须首先证明两个引理。

引理 1：每次迭代后，从 s 到每个点的距离不会减少，也就是说，$level_{i+1}[v] \ge level_i[v]$。

证明：固定一个阶段 i 和点 v。考虑 G_i^R 中从 s 到 v 的任意最短路径 P。P 的长度等于 $level_i[v]$。注意 G_i^R 只能包含 G_i^R 的后向边和前向边。如果 P 没有 G_i^R 的后边，那么 $level_{i+1}[v] \ge level_i[v]$。因为 P 也是 G_i^R 中的一条路径。现在，假设 P 至少有一个后向边且第一个这样的边是 (u,w)，那么 $level_{i+1}[u] \ge level_i[u]$（因为第一种情况）。边 (u,w) 不属于 G_i^R，因此 (u,w) 受到前一次迭代的增广路的影响。这意味着 $level_i[u] = level_i[w]+1$。此外，$level_{i+1}[w] = level_{i+1}[u]+1$。从这两个方程和 $level_{i+1}[u] \ge level_i[u]$ 我们得到 $level_{i+1}[w] \ge level_i[w]+2$。路径的剩余部分也可以使用相同思想。

引理 2：$level_{i+1}[t] > level_i[t]$。

证明：从引理一我们得出，$level_{i+1}[t] \ge level_i[t]$。假设 $level_{i+1}[t] = level_i[t]$，注意 G_i^R 只能包含 G_i^R 的后向边和前向边。这意味着 G_i^R 中有一条最短路径未被增广路阻塞。这就形成了矛盾。

实现

" 参考代码"

```
struct MPM {
  struct FlowEdge {
    int v, u;
    long long cap, flow;
```

```
  FlowEdge() {}

  FlowEdge(int _v, int _u, long long _cap, long long _flow)
      : v(_v), u(_u), cap(_cap), flow(_flow) {}

  FlowEdge(int _v, int _u, long long _cap)
      : v(_v), u(_u), cap(_cap), flow(0ll) {}
};

const long long flow_inf = 1e18;
vector<FlowEdge> edges;
vector<char> alive;
vector<long long> pin, pout;
vector<list<int> > in, out;
vector<vector<int> > adj;
vector<long long> ex;
int n, m = 0;
int s, t;
vector<int> level;
vector<int> q;
int qh, qt;

void resize(int _n) {
  n = _n;
  ex.resize(n);
  q.resize(n);
  pin.resize(n);
  pout.resize(n);
  adj.resize(n);
  level.resize(n);
  in.resize(n);
  out.resize(n);
}

MPM() {}

MPM(int _n, int _s, int _t) {
  resize(_n);
  s = _s;
  t = _t;
}

void add_edge(int v, int u, long long cap) {
  edges.push_back(FlowEdge(v, u, cap));
  edges.push_back(FlowEdge(u, v, 0));
  adj[v].push_back(m);
  adj[u].push_back(m + 1);
  m += 2;
}

bool bfs() {
  while (qh < qt) {
    int v = q[qh++];
```

```
      for (int id : adj[v]) {
        if (edges[id].cap - edges[id].flow < 1) continue;
        if (level[edges[id].u] != -1) continue;
        level[edges[id].u] = level[v] + 1;
        q[qt++] = edges[id].u;
      }
    }
    return level[t] != -1;
  }

  long long pot(int v) { return min(pin[v], pout[v]); }

  void remove_node(int v) {
    for (int i : in[v]) {
      int u = edges[i].v;
      auto it = find(out[u].begin(), out[u].end(), i);
      out[u].erase(it);
      pout[u] -= edges[i].cap - edges[i].flow;
    }
    for (int i : out[v]) {
      int u = edges[i].u;
      auto it = find(in[u].begin(), in[u].end(), i);
      in[u].erase(it);
      pin[u] -= edges[i].cap - edges[i].flow;
    }
  }

  void push(int from, int to, long long f, bool forw) {
    qh = qt = 0;
    ex.assign(n, 0);
    ex[from] = f;
    q[qt++] = from;
    while (qh < qt) {
      int v = q[qh++];
      if (v == to) break;
      long long must = ex[v];
      auto it = forw ? out[v].begin() : in[v].begin();
      while (true) {
        int u = forw ? edges[*it].u : edges[*it].v;
        long long pushed = min(must, edges[*it].cap - edges[*it].flow);
        if (pushed == 0) break;
        if (forw) {
          pout[v] -= pushed;
          pin[u] -= pushed;
        } else {
          pin[v] -= pushed;
          pout[u] -= pushed;
        }
        if (ex[u] == 0) q[qt++] = u;
        ex[u] += pushed;
        edges[*it].flow += pushed;
        edges[(*it) ^ 1].flow -= pushed;
        must -= pushed;
        if (edges[*it].cap - edges[*it].flow == 0) {
```

```
          auto jt = it;
          ++jt;
          if (forw) {
            in[u].erase(find(in[u].begin(), in[u].end(), *it));
            out[v].erase(it);
          } else {
            out[u].erase(find(out[u].begin(), out[u].end(), *it));
            in[v].erase(it);
          }
          it = jt;
        } else
          break;
        if (!must) break;
      }
    }
  }

  long long flow() {
    long long ans = 0;
    while (true) {
      pin.assign(n, 0);
      pout.assign(n, 0);
      level.assign(n, -1);
      alive.assign(n, true);
      level[s] = 0;
      qh = 0;
      qt = 1;
      q[0] = s;
      if (!bfs()) break;
      for (int i = 0; i < n; i++) {
        out[i].clear();
        in[i].clear();
      }
      for (int i = 0; i < m; i++) {
        if (edges[i].cap - edges[i].flow == 0) continue;
        int v = edges[i].v, u = edges[i].u;
        if (level[v] + 1 == level[u] && (level[u] < level[t] || u == t)) {
          in[u].push_back(i);
          out[v].push_back(i);
          pin[u] += edges[i].cap - edges[i].flow;
          pout[v] += edges[i].cap - edges[i].flow;
        }
      }
      pin[s] = pout[t] = flow_inf;
      while (true) {
        int v = -1;
        for (int i = 0; i < n; i++) {
          if (!alive[i]) continue;
          if (v == -1 || pot(i) < pot(v)) v = i;
        }
        if (v == -1) break;
        if (pot(v) == 0) {
          alive[v] = false;
          remove_node(v);
```

```
          continue;
        }
        long long f = pot(v);
        ans += f;
        push(v, s, f, false);
        push(v, t, f, true);
        alive[v] = false;
        remove_node(v);
      }
    }
    return ans;
  }
};
```

ISAP

在 Dinic 算法中，我们每次求完增广路后都要跑 BFS 来分层，有没有更高效的方法呢?

答案就是下面要介绍的 ISAP 算法。

过程

和 Dinic 算法一样，我们还是先跑 BFS 对图上的点进行分层，不过与 Dinic 略有不同的是，我们选择在反图上，从 t 点向 s 点进行 BFS。

执行完分层过程后，我们通过 DFS 来找增广路。

增广的过程和 Dinic 类似，我们只选择比当前点层数少 1 的点来增广。

与 Dinic 不同的是，我们并不会重跑 BFS 来对图上的点重新分层，而是在增广的过程中就完成重分层过程。

具体来说，设 i 号点的层为 d_i，当我们结束在 i 号点的增广过程后，我们遍历残量网络上 i 的所有出边，找到层最小的出点 j，随后令 $d_i \leftarrow d_j + 1$。特别地，若残量网络上 i 无出边，则 $d_i \leftarrow n$。

容易发现，当 $d_s \geq n$ 时，图上不存在增广路，此时即可终止算法。

和 Dinic 类似，ISAP 中也存在**当前弧优化**。

而 ISAP 还存在另外一个优化，我们记录层数为 i 的点的数量 num_i，每当将一个点的层数从 x 更新到 y 时，同时更新 num 数组的值，若在更新后 $num_x = 0$，则意味着图上出现了断层，无法再找到增广路，此时可以直接终止算法（实现时直接将 d_s 标为 n），该优化被称为 **GAP 优化**。

实现

" 参考代码"

```
struct Edge {
  int from, to, cap, flow;

  Edge(int u, int v, int c, int f) : from(u), to(v), cap(c), flow(f) {}
};

bool operator<(const Edge& a, const Edge& b) {
  return a.from < b.from || (a.from == b.from && a.to < b.to);
}

struct ISAP {
  int n, m, s, t;
  vector<Edge> edges;
  vector<int> G[maxn];
  bool vis[maxn];
```

```
  int d[maxn];
  int cur[maxn];
  int p[maxn];
  int num[maxn];

  void AddEdge(int from, int to, int cap) {
    edges.push_back(Edge(from, to, cap, 0));
    edges.push_back(Edge(to, from, 0, 0));
    m = edges.size();
    G[from].push_back(m - 2);
    G[to].push_back(m - 1);
  }

  bool BFS() {
    memset(vis, 0, sizeof(vis));
    queue<int> Q;
    Q.push(t);
    vis[t] = 1;
    d[t] = 0;
    while (!Q.empty()) {
      int x = Q.front();
      Q.pop();
      for (int i = 0; i < G[x].size(); i++) {
        Edge& e = edges[G[x][i] ^ 1];
        if (!vis[e.from] && e.cap > e.flow) {
          vis[e.from] = 1;
          d[e.from] = d[x] + 1;
          Q.push(e.from);
        }
      }
    }
    return vis[s];
  }

  void init(int n) {
    this->n = n;
    for (int i = 0; i < n; i++) G[i].clear();
    edges.clear();
  }

  int Augment() {
    int x = t, a = INF;
    while (x != s) {
      Edge& e = edges[p[x]];
      a = min(a, e.cap - e.flow);
      x = edges[p[x]].from;
    }
    x = t;
    while (x != s) {
      edges[p[x]].flow += a;
      edges[p[x] ^ 1].flow -= a;
      x = edges[p[x]].from;
    }
    return a;
```

```
  }

  int Maxflow(int s, int t) {
    this->s = s;
    this->t = t;
    int flow = 0;
    BFS();
    memset(num, 0, sizeof(num));
    for (int i = 0; i < n; i++) num[d[i]]++;
    int x = s;
    memset(cur, 0, sizeof(cur));
    while (d[s] < n) {
      if (x == t) {
        flow += Augment();
        x = s;
      }
      int ok = 0;
      for (int i = cur[x]; i < G[x].size(); i++) {
        Edge& e = edges[G[x][i]];
        if (e.cap > e.flow && d[x] == d[e.to] + 1) {
          ok = 1;
          p[e.to] = G[x][i];
          cur[x] = i;
          x = e.to;
          break;
        }
      }
      if (!ok) {
        int m = n - 1;
        for (int i = 0; i < G[x].size(); i++) {
          Edge& e = edges[G[x][i]];
          if (e.cap > e.flow) m = min(m, d[e.to]);
        }
        if (--num[d[x]] == 0) break;
        num[d[x] = m + 1]++;
        cur[x] = 0;
        if (x != s) x = edges[p[x]].from;
      }
    }
    return flow;
  }
};
```

Push-Relabel 预流推进算法

该方法在求解过程中忽略流守恒性，并每次对一个结点更新信息，以求解最大流。

通用的预流推进算法

首先我们介绍预流推进算法的主要思想，以及一个可行的暴力实现算法。

预流推进算法通过对单个结点的更新操作，直到没有结点需要更新来求解最大流。

算法过程维护的流函数不一定保持流守恒性，对于一个结点，我们允许进入结点的流超过流出结点的流，超过的部分被称为结点 $u(u \in V - \{s,t\})$ 的**超额流** $e(u)$：

$$e(u) = \sum_{(x,u)\in E} f(x,u) - \sum_{(u,y)\in E} f(u,y)$$

若 $e(u) > 0$，称结点 u **溢出**[6]，注意当我们提到溢出结点时，并不包括 s 和 t。

预流推进算法维护每个结点的高度 $h(u)$，并且规定溢出的结点 u 如果要推送超额流，只能向高度小于 u 的结点推送；如果 u 没有相邻的高度小于 u 的结点，就修改 u 的高度（重贴标签）。

高度函数[7]

准确地说，预流推进维护以下的一个映射 $h : V \to \mathbf{N}$：

- $h(s) = |V|, h(t) = 0$
- $\forall (u, v) \in E_f, h(u) \leq h(v) + 1$

称 h 是残量网络 $G_f = (V_f, E_f)$ 的高度函数。

引理 1：设 G_f 上的高度函数为 h，对于任意两个结点 $u, v \in V$，如果 $h(u) > h(v)+1$，则 (u, v) 不是 G_f 中的边。

算法只会在 $h(u) = h(v) + 1$ 的边执行推送。

推送（Push）

适用条件：结点 u 溢出，且存在结点 $v((u, v) \in E_f, c(u, v) - f(u, v) > 0, h(u) = h(v) + 1)$，则 push 操作适用于 (u, v)。

于是，我们尽可能将超额流从 u 推送到 v，推送过程中我们只关心超额流和 $c(u, v) - f(u, v)$ 的最小值，不关心 v 是否溢出。

如果 (u, v) 在推送完之后满流，将其从残量网络中删除。

重贴标签（Relabel）

适用条件：如果结点 u 溢出，且 $\forall (u, v) \in E_f, h(u) \leq h(v)$，则 relabel 操作适用于 u。

则将 $h(u)$ 更新为 $\min_{(u,v)\in E_f} h(v) + 1$ 即可。

初始化

$$\forall (u, v) \in E, \quad f(u, v) = \begin{cases} c(u, v), & u = s \\ 0, & u \neq s \end{cases}$$

$$\forall u \in V, \quad h(u) = \begin{cases} |V|, & u = s \\ 0, & u \neq s \end{cases}$$

$$e(u) = \sum_{(x,u)\in E} f(x, u) - \sum_{(u,y)\in E} f(u, y)$$

上述将 $(s, v) \in E$ 充满流，并将 $h(s)$ 抬高，使得 $(s, v) \notin E_f$，因为 $h(s) > h(v)$，而且 (s, v) 毕竟满流，没必要留在残量网络中；上述还将 $e(s)$ 初始化为 $\sum_{(s,v)\in E} f(s, v)$ 的相反数。

过程

我们每次扫描整个图，只要存在结点 u 满足 push 或 relabel 操作的条件，就执行对应的操作。

如图，每个结点中间表示编号，左下表示高度值 $h(u)$，右下表示超额流 $e(u)$，结点颜色的深度也表示结点的高度；边权表示 $c(u, v) - f(u, v)$，绿色的边表示满足 $h(u) = h(v) + 1$ 的边 (u, v)（即残量网络的边 E_f）：

整个算法我们大致浏览一下过程，这里笔者使用的是一个暴力算法，即暴力扫描是否有溢出的结点，有就更新

最后的结果

可以发现，最后的超额流一部分回到了 s，且除了源点汇点，其他结点都没有溢出；这时的流函数 f 满足流守恒性，为最大流，流量即为 $e(t)$。

但是实际上论文[3] 指出只处理高度小于 n 的溢出节点也能获得正确的最大流值，不过这样一来算法结束的时候预流还不满足流函数性质，不能知道每条边上真实的流量。

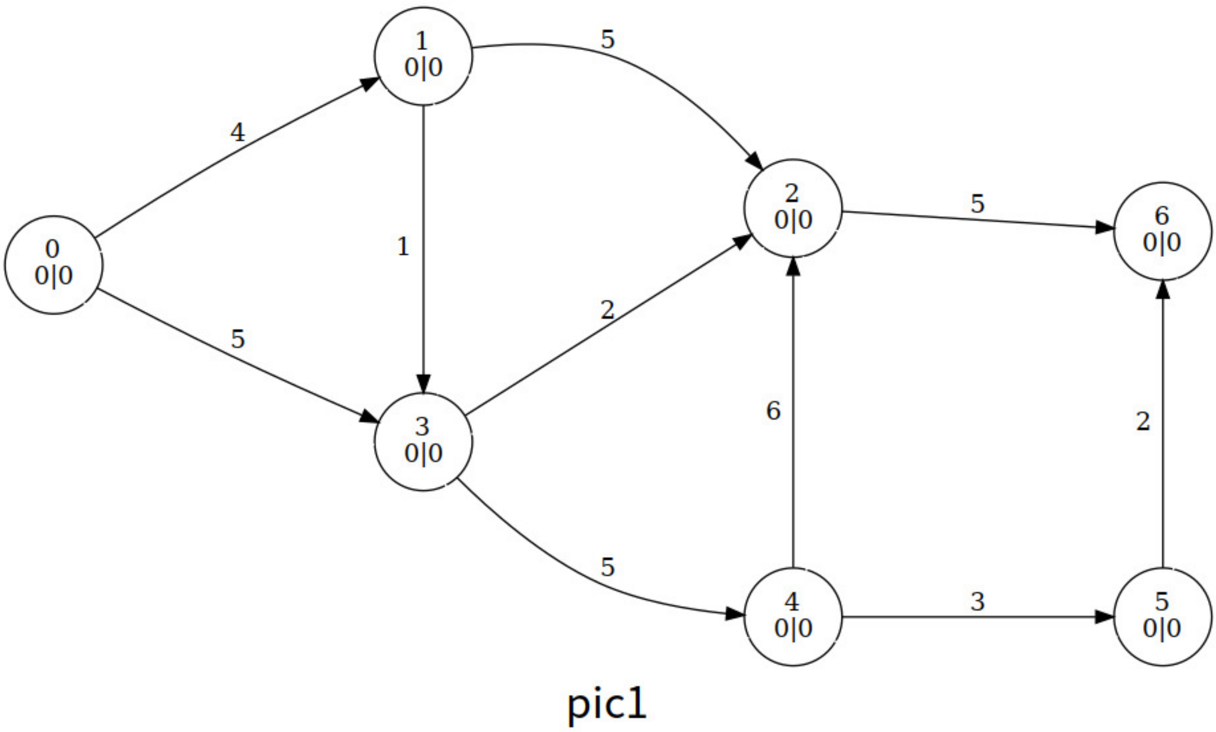


图 11.86 p1

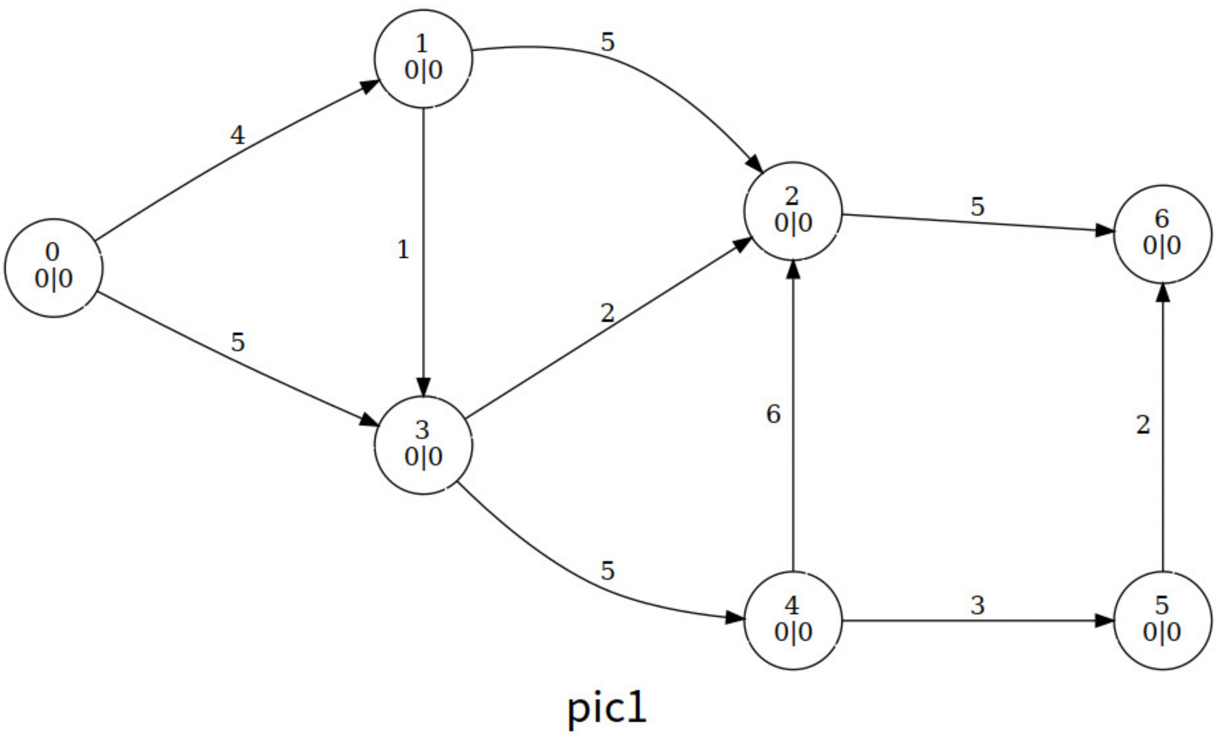


图 11.87 p2

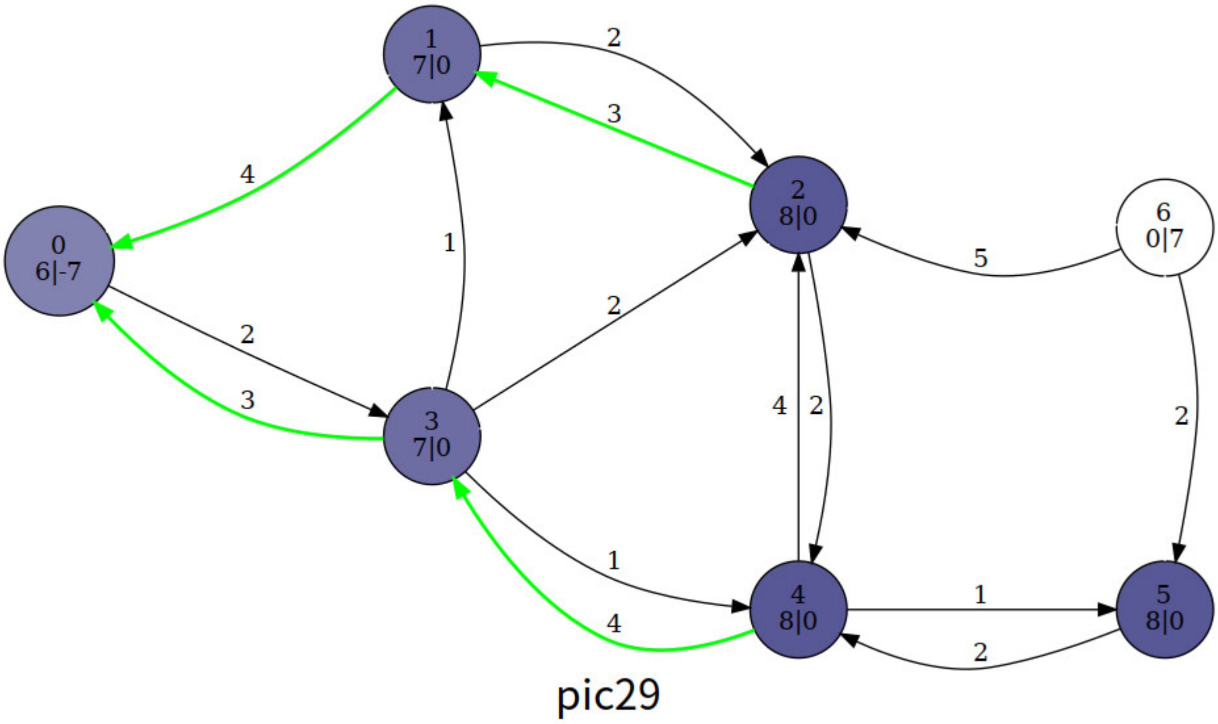


图 11.88 p3

实现

" 核心代码"

```
const int N = 1e4 + 4, M = 1e5 + 5, INF = 0x3f3f3f3f;
int n, m, s, t, maxflow, tot;
int ht[N], ex[N];

void init() {  // 初始化
  for (int i = h[s]; i; i = e[i].nex) {
    const int &v = e[i].t;
    ex[v] = e[i].v, ex[s] -= ex[v], e[i ^ 1].v = e[i].v, e[i].v = 0;
  }
  ht[s] = n;
}

bool push(int ed) {
  const int &u = e[ed ^ 1].t, &v = e[ed].t;
  int flow = min(ex[u], e[ed].v);
  ex[u] -= flow, ex[v] += flow, e[ed].v -= flow, e[ed ^ 1].v += flow;
  return ex[u];  // 如果 u 仍溢出，返回 1
}

void relabel(int u) {
  ht[u] = INF;
  for (int i = h[u]; i; i = e[i].nex)
    if (e[i].v) ht[u] = min(ht[u], ht[e[i].t]);
  ++ht[u];
}
```

HLPP 算法

最高标号预流推进算法（Highest Label Preflow Push）在上述通用的预流推送算法中，在每次选择结点时，都优先选择高度最高的溢出结点，其算法算法复杂度为 $O(n^2\sqrt{m})$。

过程

具体地说，HLPP 算法过程如下：

1. 初始化（基于预流推进算法）；
2. 选择溢出结点中高度最高的结点 u，并对它所有可以推送的边进行推送；
3. 如果 u 仍溢出，对它重贴标签，回到步骤 2；
4. 如果没有溢出的结点，算法结束。

一篇对最大流算法实际表现进行测试的论文[4-1] 表明，实际上基于预流的算法，有相当一部分时间都花在了重贴标签这一步上。以下介绍两种来自论文[5] 的能显著减少重贴标签次数的优化。

BFS 优化

HLPP 的上界为 $O(n^2\sqrt{m})$，但在使用时卡得比较紧；我们可以在初始化高度的时候进行优化。具体来说，我们初始化 $h(u)$ 为 u 到 t 的最短距离；特别地，$h(s) = n$。

在 BFS 的同时我们顺便检查图的连通性，排除无解的情况。

GAP 优化

HLPP 推送的条件是 $h(u) = h(v) + 1$，而如果在算法的某一时刻，存在某个 k，使得 $h(u) = k$ 的结点个数为 0，那么对于 $h(u) > k$ 的结点就永远无法推送超额流到 t，因此只能送回 s，那么我们就在这时直接让他们的高度变成至少 $n + 1$，以尽快推送回 s，减少重贴标签的操作。

以下的实现采取论文[4-2] 中的实现方法，使用 $N * 2 - 1$ 个桶 `B`，其中 `B[i]` 中记录所有当前高度为 i 的溢出节点。加入了以上提到的两种优化，并且只处理了高度小于 n 的溢出节点。

值得注意的是论文[4-3] 中使用的桶是基于链表的栈，而 STL 中的 `stack` 默认的容器是 `deque`。经过简单的测试发现 `vector`，`deque`，`list` 在本题的实际运行过程中效率区别不大。

实现

"LuoguP4722【模板】最大流加强版/预流推进"

```
#include <cstdio>
#include <cstring>
#include <queue>
#include <stack>
using namespace std;
const int N = 1200, M = 120000, INF = 0x3f3f3f3f;
int n, m, s, t;

struct qxx {
  int nex, t;
  long long v;
};

qxx e[M * 2 + 1];
int h[N + 1], cnt = 1;

void add_path(int f, int t, long long v) {
  e[++cnt] = qxx{h[f], t, v}, h[f] = cnt;
```

```
}

void add_flow(int f, int t, long long v) {
  add_path(f, t, v);
  add_path(t, f, 0);
}

int ht[N + 1];         // 高度;
long long ex[N + 1];   // 超额流;
int gap[N];            // gap 优化. gap[i] 为高度为 i 的节点的数量
stack<int> B[N];       // 桶 B[i] 中记录所有 ht[v]==i 的 v
int level = 0;         // 溢出节点的最高高度

int push(int u) {      // 尽可能通过能够推送的边推送超额流
  bool init = u == s;  // 是否在初始化
  for (int i = h[u]; i; i = e[i].nex) {
    const int &v = e[i].t;
    const long long &w = e[i].v;
    // 初始化时不考虑高度差为 1
    if (!w || (init == false && ht[u] != ht[v] + 1) || t[v] == INF) continue;
    long long k = init ? w : min(w, ex[u]);
    // 取到剩余容量和超额流的最小值，初始化时可以使源的溢出量为负数。
    if (v != s && v != t && !ex[v]) B[ht[v]].push(v), level = max(level, ht[v]);
    ex[u] -= k, ex[v] += k, e[i].v -= k, e[i ^ 1].v += k;  // push
    if (!ex[u]) return 0;  // 如果已经推送完就返回
  }
  return 1;
}

void relabel(int u) {  // 重贴标签（高度）
  ht[u] = INF;
  for (int i = h[u]; i; i = e[i].nex)
    if (e[i].v) ht[u] = min(ht[u], ht[e[i].t]);
  if (++ht[u] < n) {  // 只处理高度小于 n 的节点
    B[ht[u]].push(u);
    level = max(level, ht[u]);
    ++gap[ht[u]];  // 新的高度，更新 gap
  }
}

bool bfs_init() {
  memset(ht, 0x3f, sizeof(ht));
  queue<int> q;
  q.push(t), ht[t] = 0;
  while (q.size()) {  // 反向 BFS, 遇到没有访问过的结点就入队
    int u = q.front();
    q.pop();
    for (int i = h[u]; i; i = e[i].nex) {
      const int &v = e[i].t;
      if (e[i ^ 1].v && ht[v] > ht[u] + 1) ht[v] = ht[u] + 1, q.push(v);
    }
  }
  return ht[s] != INF;  // 如果图不连通，返回 0
}
```

```cpp
// 选出当前高度最大的节点之一，如果已经没有溢出节点返回 0
int select() {
  while (B[level].size() == 0 && level > -1) level--;
  return level == -1 ? 0 : B[level].top();
}

long long hlpp() {          // 返回最大流
  if (!bfs_init()) return 0;  // 图不连通
  memset(gap, 0, sizeof(gap));
  for (int i = 1; i <= n; i++)
    if (ht[i] != INF) gap[ht[i]]++;  // 初始化 gap
  ht[s] = n;
  push(s);  // 初始化预流
  int u;
  while ((u = select())) {
    B[level].pop();
    if (push(u)) {  // 仍然溢出
      if (!--gap[ht[u]])
        for (int i = 1; i <= n; i++)
          if (i != s && ht[i] > ht[u] && ht[i] < n + 1)
            ht[i] = n + 1;  // 这里重贴成 n+1 的节点都不是溢出节点
      relabel(u);
    }
  }
  return ex[t];
}

int main() {
  scanf("%d%d%d%d", &n, &m, &s, &t);
  for (int i = 1, u, v, w; i <= m; i++) {
    scanf("%d%d%d", &u, &v, &w);
    add_flow(u, v, w);
  }
  printf("%lld", hlpp());
  return 0;
}
```

感受一下运行过程

其中 pic13 到 pic14 执行了 Relabel(4)，并进行了 GAP 优化。

脚注

[1] http://pisces.ck.tp.edu.tw/~peng/index.php?action=showfile&file=f6cdf7ef750d7dc79c7d599b942acbaaee86a2e3e

[2] https://people.orie.cornell.edu/dpw/orie633/LectureNotes/lecture9.pdf

[3] Cherkassky B V, Goldberg A V. On implementing push-relabel method for the maximum flow problem[C]//International Conference on Integer Programming and Combinatorial Optimization. Springer, Berlin, Heidelberg, 1995: 157-171.

[4] Ahuja R K, Kodialam M, Mishra A K, et al. Computational investigations of maximum flow algorithms[J]. European Journal of Operational Research, 1997, 97(3): 509-542. [4-1] [4-2] [4-3]

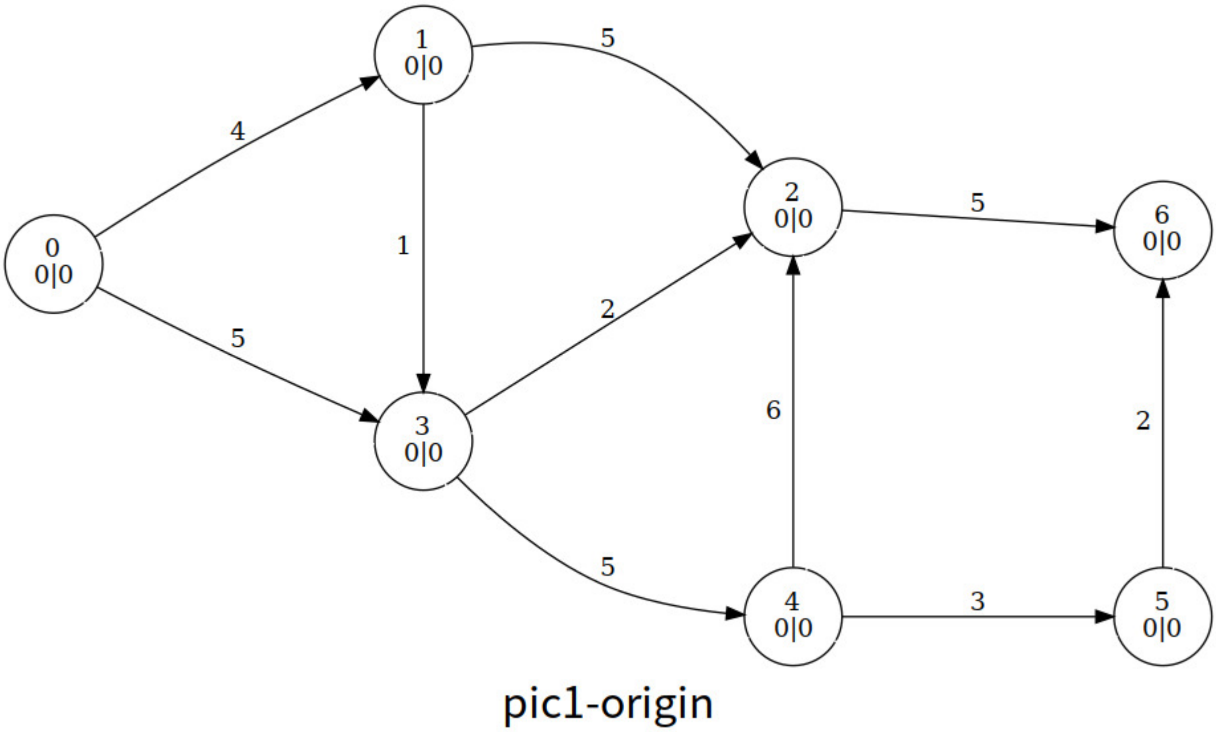


图 11.89 HLPP

[5] Derigs U, Meier W. Implementing Goldberg's max-flow-algorithm—A computational investigation[J]. Zeitschrift für Operations Research, 1989, 33(6): 383-403.

[6] 英语文献中通常称为「active」。

[7] 在英语文献中，一个结点的高度通常被称为「distance label」。此处使用的「高度」这个术语源自算法导论中的相关章节。你可以在机械工业出版社算法导论（原书第 3 版）的 P432 脚注中找到这么做的理由。

11.28.3 最小割

概念

割

对于一个网络流图 $G=(V,E)$，其割的定义为一种**点的划分方式**：将所有的点划分为 S 和 $T=V-S$ 两个集合，其中源点 $s\in S$，汇点 $t\in T$。

割的容量

我们的定义割 (S,T) 的容量 $c(S,T)$ 表示所有从 S 到 T 的边的容量之和，即 $c(S,T)=\sum_{u\in S,v\in T}c(u,v)$。当然我们也可以用 $c(s,t)$ 表示 $c(S,T)$。

最小割

最小割就是求得一个割 (S,T) 使得割的容量 $c(S,T)$ 最小。

证明

最大流最小割定理

参见 最大流 页面最大流最小割定理一节。

代码

最小割

通过**最大流最小割定理**，我们可以直接得到如下代码：

" 参考代码"

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <queue>

const int N = 1e4 + 5, M = 2e5 + 5;
int n, m, s, t, tot = 1, lnk[N], ter[M], nxt[M], val[M], dep[N], cur[N];

void add(int u, int v, int w) {
  ter[++tot] = v, nxt[tot] = lnk[u], lnk[u] = tot, val[tot] = w;
}

void addedge(int u, int v, int w) { add(u, v, w), add(v, u, 0); }

int bfs(int s, int t) {
  memset(dep, 0, sizeof(dep));
  memcpy(cur, lnk, sizeof(lnk));
  std::queue<int> q;
  q.push(s), dep[s] = 1;
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    for (int i = lnk[u]; i; i = nxt[i]) {
      int v = ter[i];
      if (val[i] && !dep[v]) q.push(v), dep[v] = dep[u] + 1;
    }
  }
  return dep[t];
}

int dfs(int u, int t, int flow) {
  if (u == t) return flow;
  int ans = 0;
  for (int &i = cur[u]; i && ans < flow; i = nxt[i]) {
    int v = ter[i];
    if (val[i] && dep[v] == dep[u] + 1) {
      int x = dfs(v, t, std::min(val[i], flow - ans));
      if (x) val[i] -= x, val[i ^ 1] += x, ans += x;
    }
  }
  if (ans < flow) dep[u] = -1;
  return ans;
}

int dinic(int s, int t) {
  int ans = 0;
  while (bfs(s, t)) {
```

```
    int x;
    while ((x = dfs(s, t, 1 << 30))) ans += x;
  }
  return ans;
}

int main() {
  scanf("%d%d%d%d", &n, &m, &s, &t);
  while (m--) {
    int u, v, w;
    scanf("%d%d%d", &u, &v, &w);
    addedge(u, v, w);
  }
  printf("%d\n", dinic(s, t));
  return 0;
}
```

方案

我们可以通过从源点 s 开始 DFS，每次走残量大于 0 的边，找到所有 S 点集内的点。

```
void dfs(int u) {
  vis[u] = 1;
  for (int i = lnk[u]; i; i = nxt[i]) {
    int v = ter[i];
    if (!vis[v] && val[i]) dfs(v);
  }
}
```

割边数量

如果需要在最小割的前提下最小化割边数量，那么先求出最小割，把没有满流的边容量改成 ∞，满流的边容量改成 1，重新跑一遍最小割就可求出最小割边数量；如果没有最小割的前提，直接把所有边的容量设成 1，求一遍最小割就好了。

问题模型 1

有 n 个物品和两个集合 A,B，如果一个物品没有放入 A 集合会花费 a_i，没有放入 B 集合会花费 b_i；还有若干个形如 u_i,v_i,w_i 限制条件，表示如果 u_i 和 v_i 同时不在一个集合会花费 w_i。每个物品必须且只能属于一个集合，求最小的代价。

这是一个经典的二**者选其**一的最小割题目。我们对于每个集合设置源点 s 和汇点 t，第 i 个点由 s 连一条容量为 a_i 的边、向 t 连一条容量为 b_i 的边。对于限制条件 u,v,w，我们在 u,v 之间连容量为 w 的双向边。

注意到当源点和汇点不相连时，代表这些点都选择了其中一个集合。如果将连向 s 或 t 的边割开，表示不放在 A 或 B 集合，如果把物品之间的边割开，表示这两个物品不放在同一个集合。

最小割就是最小花费。

问题模型 2

最大权值闭合图，即给定一张有向图，每个点都有一个权值（可以为正或负或 0），你需要选择一个权值和最大的子图，使得子图中每个点的后继都在子图中。

做法：建立超级源点 s 和超级汇点 t，若节点 u 权值为正，则 s 向 u 连一条有向边，边权即为该点点权；若节点 u 权值为负，则由 u 向 t 连一条有向边，边权即为该点点权的相反数。原图上所有边权改为 ∞。跑网络最大流，将所有正权值之和减去最大流，即为答案。

几个小结论来证明：

1. 每一个符合条件的子图都对应流量网络中的一个割。因为每一个割将网络分为两部分，与 s 相连的那部分满足没有边指向另一部分，于是满足上述条件。这个命题是充要的。
2. 最小割所去除的边必须与 s 和 t 其中一者相连。因为否则边权是 ∞，不可能成为最小割。
3. 我们所选择的那部分子图，权值和 $=$ 所有正权值之和 $-$ 我们未选择的正权值点的权值之和 $+$ 我们选择的负权值点的权值之和。当我们不选择一个正权值点时，其与 s 的连边会被断开；当我们选择一个负权值点时，其与 t 的连边会被断开。断开的边的边权之和即为割的容量。于是上述式子转化为：权值和 $=$ 所有正权值之和 $-$ 割的容量。
4. 于是得出结论，最大权值和 $=$ 所有正权值之和 $-$ 最小割 $=$ 所有正权值之和 $-$ 最大流。

习题

- 「USACO 4.4」Pollutant Control[1]
- 「USACO 5.4」Telecowmunication[2]
- 「Luogu 1361」 小 M 的作物[3]
- 「SHOI 2007」 善意的投票[4]
- 太空飞行计划问题[5]

参考资料与注释

[1] 「USACO 4.4」Pollutant Control

[2] 「USACO 5.4」Telecowmunication

[3] 「Luogu 1361」小 M 的作物

[4] 「SHOI 2007」善意的投票

[5] 太空飞行计划问题

11.28.4 费用流

在看这篇文章前请先看 网络流简介 这篇 wiki 的定义部分。

费用流

给定一个网络 $G=(V,E)$，每条边除了有容量限制 $c(u,v)$，还有一个单位流量的费用 $w(u,v)$。

当 (u,v) 的流量为 $f(u,v)$ 时，需要花费 $f(u,v)\times w(u,v)$ 的费用。

w 也满足斜对称性，即 $w(u,v)=-w(v,u)$。

则该网络中总花费最小的最大流称为**最小费用最大流**，即在最大化 $\sum_{(s,v)\in E} f(s,v)$ 的前提下最小化 $\sum_{(u,v)\in E} f(u,v)\times w(u,v)$。

SSP 算法

SSP（Successive Shortest Path）算法是一个贪心的算法。它的思路是每次寻找单位费用最小的增广路进行增广，直到图上不存在增广路为止。

如果图上存在单位费用为负的圈，SSP 算法无法正确求出该网络的最小费用最大流。此时需要先使用消圈算法消去图上的负圈。

证明

我们考虑使用数学归纳法和反证法来证明 SSP 算法的正确性。

设流量为 i 的时候最小费用为 f_i。我们假设最初的网络上**没有负圈**，这种情况下 $f_0=0$。

假设用 SSP 算法求出的 f_i 是最小费用，我们在 f_i 的基础上，找到一条最短的增广路，从而求出 f_{i+1}。这时 $f_{i+1}-f_i$ 是这条最短增广路的长度。

假设存在更小的 f_{i+1}，设它为 f'_{i+1}。因为 $f_{i+1}-f_i$ 已经是最短增广路了，所以 $f'_{i+1}-f_i$ 一定对应一个经过**至少一个负圈**的增广路。

这时候矛盾就出现了：既然存在一条经过至少一个负圈的增广路，那么 f_i 就不是最小费用了。因为只要给这个负圈添加流量，就可以在不增加 s 流出的流量的前提下，使 f_i 对应的费用更小。

综上，SSP 算法可以正确求出无负圈网络的最小费用最大流。

时间复杂度

如果使用 Bellman–Ford 算法 求解最短路，每次找增广路的时间复杂度为 $O(nm)$。设该网络的最大流为 f，则最坏时间复杂度为 $O(nmf)$。事实上，SSP 算法是 伪多项式时间 的。

" 为什么 SSP 算法是伪多项式时间的？"

SSP 算法的时间复杂度有 $O(nmf)$ 的上界，这是一个关于值域的多项式，所以是伪多项式时间的。

可以构造 $m=n^2, f=2^{n/2}$ 的网络[1] 使得 SSP 算法的时间复杂度达到 $O(n^3 2^{n/2})$，所以 SSP 算法不是多项式时间的。

实现

只需将 EK 算法或 Dinic 算法中找增广路的过程，替换为用最短路算法寻找单位费用最小的增广路即可。

" 基于 EK 算法的实现"

```cpp
struct qxx {
  int nex, t, v, c;
};

qxx e[M];
int h[N], cnt = 1;

void add_path(int f, int t, int v, int c) {
  e[++cnt] = (qxx){h[f], t, v, c}, h[f] = cnt;
}

void add_flow(int f, int t, int v, int c) {
  add_path(f, t, v, c);
  add_path(t, f, 0, -c);
}

int dis[N], pre[N], incf[N];
bool vis[N];

bool spfa() {
  memset(dis, 0x3f, sizeof(dis));
```

```
  queue<int> q;
  q.push(s), dis[s] = 0, incf[s] = INF, incf[t] = 0;
  while (q.size()) {
    int u = q.front();
    q.pop();
    vis[u] = 0;
    for (int i = h[u]; i; i = e[i].nex) {
      const int &v = e[i].t, &w = e[i].v, &c = e[i].c;
      if (!w || dis[v] <= dis[u] + c) continue;
      dis[v] = dis[u] + c, incf[v] = min(w, incf[u]), pre[v] = i;
      if (!vis[v]) q.push(v), vis[v] = 1;
    }
  }
  return incf[t];
}

int maxflow, mincost;

void update() {
  maxflow += incf[t];
  for (int u = t; u != s; u = e[pre[u] ^ 1].t) {
    e[pre[u]].v -= incf[t], e[pre[u] ^ 1].v += incf[t];
    mincost += incf[t] * e[pre[u]].c;
  }
}

// 调用: while(spfa())update();
```

" 基于 Dinic 算法的实现"

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <queue>

const int N = 5e3 + 5, M = 1e5 + 5;
const int INF = 0x3f3f3f3f;
int n, m, tot = 1, lnk[N], cur[N], ter[M], nxt[M], cap[M], cost[M], dis[N], ret;
bool vis[N];

void add(int u, int v, int w, int c) {
  ter[++tot] = v, nxt[tot] = lnk[u], lnk[u] = tot, cap[tot] = w, cost[tot] = c;
}

void addedge(int u, int v, int w, int c) { add(u, v, w, c), add(v, u, 0, -c); }

bool spfa(int s, int t) {
  memset(dis, 0x3f, sizeof(dis));
  memcpy(cur, lnk, sizeof(lnk));
  std::queue<int> q;
  q.push(s), dis[s] = 0, vis[s] = 1;
  while (!q.empty()) {
    int u = q.front();
```

```cpp
    q.pop(), vis[u] = 0;
    for (int i = lnk[u]; i; i = nxt[i]) {
      int v = ter[i];
      if (cap[i] && dis[v] > dis[u] + cost[i]) {
        dis[v] = dis[u] + cost[i];
        if (!vis[v]) q.push(v), vis[v] = 1;
      }
    }
  }
  return dis[t] != INF;
}

int dfs(int u, int t, int flow) {
  if (u == t) return flow;
  vis[u] = 1;
  int ans = 0;
  for (int &i = cur[u]; i && ans < flow; i = nxt[i]) {
    int v = ter[i];
    if (!vis[v] && cap[i] && dis[v] == dis[u] + cost[i]) {
      int x = dfs(v, t, std::min(cap[i], flow - ans));
      if (x) ret += x * cost[i], cap[i] -= x, cap[i ^ 1] += x, ans += x;
    }
  }
  vis[u] = 0;
  return ans;
}

int mcmf(int s, int t) {
  int ans = 0;
  while (spfa(s, t)) {
    int x;
    while ((x = dfs(s, t, INF))) ans += x;
  }
  return ans;
}

int main() {
  int s, t;
  scanf("%d%d%d%d", &n, &m, &s, &t);
  while (m--) {
    int u, v, w, c;
    scanf("%d%d%d%d", &u, &v, &w, &c);
    addedge(u, v, w, c);
  }
  int ans = mcmf(s, t);
  printf("%d %d\n", ans, ret);
  return 0;
}
```

Primal-Dual 原始对偶算法

用 Bellman–Ford 求解最短路的时间复杂度为 $O(nm)$，无论在稀疏图上还是稠密图上都不及 Dijkstra 算法[2]。但网络上存在单位费用为负的边，因此无法直接使用 Dijkstra 算法。

Primal-Dual 原始对偶算法的思路与 Johnson 全源最短路径算法 类似，通过为每个点设置一个势能，将网络上所有边的费用（下面简称为边权）全部变为非负值，从而可以应用 Dijkstra 算法找出网络上单位费用最小的增广路。

首先跑一次最短路，求出源点到每个点的最短距离（也是该点的初始势能）h_i。接下来和 Johnson 算法一样，对于一条从 u 到 v，单位费用为 w 的边，将其边权重置为 $w+h_u-h_v$。

可以发现，这样设置势能后新网络上的最短路径和原网络上的最短路径一定对应。证明在介绍 Johnson 算法时已经给出，这里不再展开。

与常规的最短路问题不同的是，每次增广后图的形态会发生变化，这种情况下各点的势能需要更新。

如何更新呢？先给出结论，设增广后从源点到 i 号点的最短距离为 d'_i（这里的距离为重置每条边边权后得到的距离），只需给 h_i 加上 d'_i 即可。下面我们证明，这样更新边权后，图上所有边的边权均为非负。

容易发现，在一轮增广后，由于一些 (i,j) 边在增广路上，残量网络上会相应多出一些 (j,i) 边，且一定会满足 $d'_i+(w(i,j)+h_i-h_j)=d'_j$（否则 (i,j) 边就不会在增广路上了）。稍作变形后可以得到 $w(j,i)+(h_j+d'_j)-(h_i+d'_i)=0$。因此新增的边的边权非负。

而对于原有的边，在增广前，$d'_i+(w(i,j)+h_i-h_j)-d'_j\geq 0$，因此 $w(i,j)+(d'_i+h_i)-(d'_j+h_j)\geq 0$，即用 $h_i+d'_i$ 作为新势能并不会使 (i,j) 的边权变为负。

综上，增广后所有边的边权均非负，使用 Dijkstra 算法可以正确求出图上的最短路。

"参考代码"

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <queue>
#define INF 0x3f3f3f3f
using namespace std;

struct edge {
  int v, f, c, next;
} e[100005];

struct node {
  int v, e;
} p[10005];

struct mypair {
  int dis, id;

  bool operator<(const mypair& a) const { return dis > a.dis; }

  mypair(int d, int x) { dis = d, id = x; }
};

int head[5005], dis[5005], vis[5005], h[5005];
int n, m, s, t, cnt = 1, maxf, minc;

void addedge(int u, int v, int f, int c) {
  e[++cnt].v = v;
  e[cnt].f = f;
  e[cnt].c = c;
  e[cnt].next = head[u];
  head[u] = cnt;
}

bool dijkstra() {
  priority_queue<mypair> q;
```

```
  for (int i = 1; i <= n; i++) dis[i] = INF;
  memset(vis, 0, sizeof(vis));
  dis[s] = 0;
  q.push(mypair(0, s));
  while (!q.empty()) {
    int u = q.top().id;
    q.pop();
    if (vis[u]) continue;
    vis[u] = 1;
    for (int i = head[u]; i; i = e[i].next) {
      int v = e[i].v, nc = e[i].c + h[u] - h[v];
      if (e[i].f && dis[v] > dis[u] + nc) {
        dis[v] = dis[u] + nc;
        p[v].v = u;
        p[v].e = i;
        if (!vis[v]) q.push(mypair(dis[v], v));
      }
    }
  }
  return dis[t] != INF;
}

void spfa() {
  queue<int> q;
  memset(h, 63, sizeof(h));
  h[s] = 0, vis[s] = 1;
  q.push(s);
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    vis[u] = 0;
    for (int i = head[u]; i; i = e[i].next) {
      int v = e[i].v;
      if (e[i].f && h[v] > h[u] + e[i].c) {
        h[v] = h[u] + e[i].c;
        if (!vis[v]) {
          vis[v] = 1;
          q.push(v);
        }
      }
    }
  }
}

int main() {
  scanf("%d%d%d%d", &n, &m, &s, &t);
  for (int i = 1; i <= m; i++) {
    int u, v, f, c;
    scanf("%d%d%d%d", &u, &v, &f, &c);
    addedge(u, v, f, c);
    addedge(v, u, 0, -c);
  }
  spfa();  // 先求出初始势能
  while (dijkstra()) {
```

```
    int minf = INF;
    for (int i = 1; i <= n; i++) h[i] += dis[i];
    for (int i = t; i != s; i = p[i].v) minf = min(minf, e[p[i].e].f);
    for (int i = t; i != s; i = p[i].v) {
      e[p[i].e].f -= minf;
      e[p[i].e ^ 1].f += minf;
    }
    maxf += minf;
    minc += minf * h[t];
  }
  printf("%d %d\n", maxf, minc);
  return 0;
}
```

习题

- 「Luogu 3381」【模板】最小费用最大流[3]
- 「Luogu 4452」航班安排[4]
- 「SDOI 2009」晨跑[5]
- 「SCOI 2007」修车[6]
- 「HAOI 2010」订货[7]
- 「NOI 2012」美食节[8]

参考资料与注释

[1] 详细构造方法可以参考 min_25 的博客。

[2] 在稀疏图上使用堆优化可以做到 $O(m \log n)$ 的时间复杂度，而在稠密图上不使用堆优化，可以做到 $O(n^2)$ 的时间复杂度。

[3] 「Luogu 3381」【模板】最小费用最大流

[4] 「Luogu 4452」航班安排

[5] 「SDOI 2009」晨跑

[6] 「SCOI 2007」修车

[7] 「HAOI 2010」订货

[8] 「NOI 2012」美食节

11.28.5 上下界网络流

在阅读这篇文章之前请先阅读 最大流 并确保自己熟练掌握最大流算法。

概述

上下界网络流本质是给流量网络的每一条边设置了流量上界 $c(u,v)$ 和流量下界 $b(u,v)$。也就是说，一种可行的流必须满足 $b(u,v) \le f(u,v) \le c(u,v)$。同时必须满足除了源点和汇点之外的其余点流量平衡。

根据题目要求，我们可以使用上下界网络流解决不同问题。

无源汇上下界可行流

给定无源汇流量网络 G。询问是否存在一种标定每条边流量的方式，使得每条边流量满足上下界同时每一个点流量平衡。

不妨假设每条边已经流了 $b(u,v)$ 的流量，设其为初始流。同时我们在新图中加入 u 连向 v 的流量为 $c(u,v)-b(u,v)$ 的边。考虑在新图上进行调整。

由于最大流需要满足初始流量平衡条件（最大流可以看成是下界为 0 的上下界最大流），但是构造出来的初始流很有可能不满足初始流量平衡。假设一个点初始流入流量减初始流出流量为 M。

若 $M=0$，此时流量平衡，不需要附加边。

若 $M>0$，此时入流量过大，需要新建附加源点 S'，S' 向其连流量为 M 的附加边。

若 $M<0$，此时出流量过大，需要新建附加汇点 T'，其向 T' 连流量为 $-M$ 的附加边。

如果附加边满流，说明这一个点的流量平衡条件可以满足，否则这个点的流量平衡条件不满足。（因为原图加上附加流之后才会满足原图中的流量平衡。）

在建图完毕之后跑 S' 到 T' 的最大流，若 S' 连出去的边全部满流，则存在可行流，否则不存在。

有源汇上下界可行流

给定有源汇流量网络 G。询问是否存在一种标定每条边流量的方式，使得每条边流量满足上下界同时除了源点和汇点每一个点流量平衡。

假设源点为 S，汇点为 T。

则我们可以加入一条 T 到 S 的上界为 ∞，下界为 0 的边转化为无源汇上下界可行流问题。

若有解，则 S 到 T 的可行流流量等于 T 到 S 的附加边的流量。

有源汇上下界最大流

给定有源汇流量网络 G。询问是否存在一种标定每条边流量的方式，使得每条边流量满足上下界同时除了源点和汇点每一个点流量平衡。如果存在，询问满足标定的最大流量。

我们找到网络上的任意一个可行流。如果找不到解就可以直接结束。

否则我们考虑删去所有附加边之后的残量网络并且在网络上进行调整。

我们在残量网络上再跑一次 S 到 T 的最大流，将可行流流量和最大流流量相加即为答案。

"一个非常易错的问题"

> S 到 T 的最大流直接在跑完有源汇上下界可行的残量网络上跑。
>
> 千万不可以在原来的流量网络上跑。

有源汇上下界最小流

给定有源汇流量网络 G。询问是否存在一种标定每条边流量的方式，使得每条边流量满足上下界同时除了源点和汇点每一个点流量平衡。如果存在，询问满足标定的最小流量。

类似的，我们考虑将残量网络中不需要的流退掉。

我们找到网络上的任意一个可行流。如果找不到解就可以直接结束。

否则我们考虑删去所有附加边之后的残量网络。

我们在残量网络上再跑一次 T 到 S 的最大流，将可行流流量减去最大流流量即为答案。

"AHOI 2014 支线剧情[1]"

对于每条 x 到 y 花费 v 的剧情边设上界为 ∞, 下界为 1。

对于每个点，向 T 连边权 c, 上界 ∞, 下界为 1。

S 点为 1 号节点。

跑一次上下界带源汇最小费用可行流即可。

因为最小费用可行流解法与最小可行流类似，这里不再展开。

参考资料与注释

[1] AHOI 2014 支线剧情

11.29 Stoer–Wagner 算法

Authors: DanJoshua, opsiff, yzy-1

定义

由于取消了**源汇点**的定义，我们需要对**割**的概念进行重定义。

（其实是网络流部分有关割的定义与维基百科不符，只是由于一般接触到的割都是「有源汇的最小割问题」，因此这个概念也就约定俗成了。）

割

去掉其中所有边能使一张网络流图不再连通（即分成两个子图）的边集称为图的割。

即：在无向图 $G=(V,E)$ 中，设 C 为图 G 中一些弧的集合，若从 G 中删去 C 中的所有弧能使图 G 不是连通图，称 C 图 G 的一个割。

有源汇点的最小割问题

同 最小割 中的定义。

无源汇点的最小割问题

包含的弧的权和最小的割。也称为全局最小割。

显然，直接跑网络流的复杂度是行不通的。

Stoer–Wagner 算法

引入

Stoer–Wagner 算法在 1995 年由 *Mechthild Stoer* 与 *Frank Wagner* 提出，是一种通过**递归**的方式来解决**无向正权图**上的全局最小割问题的算法。

性质

算法复杂度 $O(|V||E|+|V|^2\log|V|)$ 一般可近似看作 $O(|V|^3)$。

它的实现基于以下基本事实：设图 G 中有任意两点 S,T。那么任意一个图 G 的割 C，或者有 S,T 在同一连通块中，或者有 C 是一个 $S-T$ 割。

过程

1. 在图 G 中任意指定两点 s,t，并且以这两点作为源汇点求出图 G 的 $S-T$ 最小割（定义为 *cut of phase*），更新当前答案。
2. 「合并」点 s,t，如果图 G 中 $|V|$ 大于 1，则回到第一步。
3. 输出所有 *cut of phase* 的最小值。

合并两点 s,t：删除 s,t 之间的连边 (s,t)，对于 $G/\{s,t\}$ 中任意一点 k，删除 (t,k)，并将其边权 $d(t,k)$ 加到 $d(s,k)$ 上

解释：如果 s,t 在同一连通块，对于 $G/\{s,t\}$ 中的一点 k，假如 $(k,s)\in C_{\min}$，那么 $(k,t)\in C_{\min}$ 也一定成立，否则因为 s,t 连通，k,t 连通，导致 s,k 在同一连通块，此时 $C=C_{\min}/s$ 将比 $C_{\min}$ 更优。反之亦然。所以 s,t 可以看作同一点。

步骤 1 考虑了 s,t 不在同一连通块的情形，步骤 2 考虑了剩余的情况。由于每次执行步骤 2 都会使 $|V|$ 减小 1，因此算法将在进行 $|V|-1$ 后结束。

S-T 最小割的求法

（显然不是网络流。）

假设进行若干次合并以后，当前图 $G'=(V',E')$，执行步骤 1。

我们构造一个集合 A，初始时令 $A=\emptyset$。

我们每次将 V' 中所有点中，满足 $i\notin A$，且权值函数 $w(A,i)$ 最大的节点加入集合 A，直到 $|A|=|V'|$。

其中权值函数的定义：

$w(A,i)=\sum_{j\in A} d(i,j)$

（若 $(i,j)\notin E'$，则 $d(i,j)=0$）。

容易知道所有点加入 A 的顺序是固定的，令 $\mathrm{ord}(i)$ 表示第 i 个加入 A 的点，$t=\mathrm{ord}(|V'|)$；$\mathrm{pos}(v)$ 表示 v 被加入 A 后 $|A|$ 的大小，即 v 被加入的顺序。

则对任意点 s，一个 s 到 t 的割即为 $w(t)$。

证明

定义一个点 v 被激活，当且仅当 v 在加入 A 中时，发现在 A 此时最后一个点 u 早于 v 加入集合，并且在图 $G''=(V',E'/C)$ 中，u 与 v 不在同一连通块。

如图，蓝色区域和黄色区域为两个不同的连通块，方括号中的数字为加入 A 的顺序。灰色节点为活跃节点，白色节点则不是活跃节点。

定义 $A_v=\{u\mid \mathrm{pos}(u)<\mathrm{pos}(v)\}$，也就是严格早于 v 加入 A 的点，令 E_v 为 E' 的诱导子图（点集为 $A_v\cup\{v\}$）的边集。（注意包含点 v。）

定义诱导割 C_v 为 $C\cap E_v$。$w(C_v)=\sum_{(i,j)\in C_v} d(i,j)$。

"Lemma 1"

对于任何被激活的点 v，$w(A_v,v)\le w(C_v)$。

证明：使用数学归纳法。

对于第一个被激活的点 v_0，由定义可知 $w(A_{v_0},v_0)=w(C_{v_0})$。

对于之后两个被激活的点 u,v，假设 $\mathrm{pos}(v)<\mathrm{pos}(u)$，则有：

$w(A_u,u)=w(A_v,u)+w(A_u-A_v,u)$

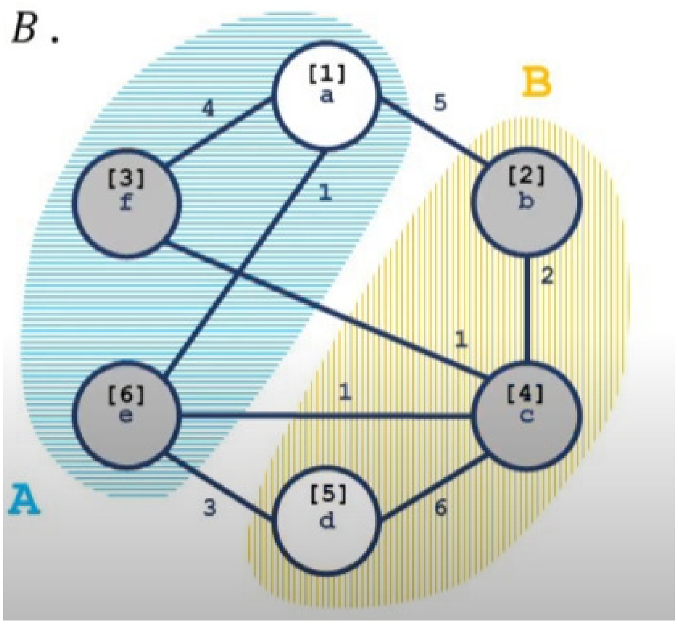


图 11.90 Stoer-Wagner1

又，已知：

$w(A_v, u) \le w(A_v, v)$ 并且 $w(A_v, v) \le w(C_v)$ 联立可得：

$w(A_u, u) \le w(C_v) + w(A_u - A_v, u)$

由于 $w(A_u - A_v, u)$ 对 $w(C_u)$ 有贡献而对 $w(C_v)$ 没有贡献，在所有边均为正权的情况下，可导出：

$w(A_u, u) \le w(C_u)$

由归纳法得证。

由于 $\text{pos}(s) < \text{pos}(t)$，并且 s, t 不在同一连通块，因此 t 会被激活，由此可以得出 $w(A_t, t) \le w(C_t) = w(C)$。

"P5632【模板】Stoer–Wagner 算法[1]"

```
#include <bits/stdc++.h>
using namespace std;
const int N = 601;
int fa[N], siz[N], edge[N][N];

int find(int x) { return fa[x] == x ? x : fa[x] = find(fa[x]); }

int dist[N], vis[N], bin[N];
int n, m;

int contract(int &s, int &t) {  // Find s,t
  memset(dist, 0, sizeof(dist));
  memset(vis, false, sizeof(vis));
  int i, j, k, mincut, maxc;
  for (i = 1; i <= n; i++) {
    k = -1;
    maxc = -1;
    for (j = 1; j <= n; j++)
      if (!bin[j] && !vis[j] && dist[j] > maxc) {
        k = j;
        maxc = dist[j];
```

```cpp
      }
    if (k == -1) return mincut;
    s = t;
    t = k;
    mincut = maxc;
    vis[k] = true;
    for (j = 1; j <= n; j++)
      if (!bin[j] && !vis[j]) dist[j] += edge[k][j];
  }
  return mincut;
}

const int inf = 0x3f3f3f3f;

int Stoer_Wagner() {
  int mincut, i, j, s, t, ans;
  for (mincut = inf, i = 1; i < n; i++) {
    ans = contract(s, t);
    bin[t] = true;
    if (mincut > ans) mincut = ans;
    if (mincut == 0) return 0;
    for (j = 1; j <= n; j++)
      if (!bin[j]) edge[s][j] = (edge[j][s] += edge[j][t]);
  }
  return mincut;
}

int main() {
  ios::sync_with_stdio(0), cin.tie(0);
  cin >> n >> m;
  if (m < n - 1) {
    cout << 0;
    return 0;
  }
  for (int i = 1; i <= n; ++i) fa[i] = i, siz[i] = 1;
  for (int i = 1, u, v, w; i <= m; ++i) {
    cin >> u >> v >> w;
    int fu = find(u), fv = find(v);
    if (fu != fv) {
      if (siz[fu] > siz[fv]) swap(fu, fv);
      fa[fu] = fv, siz[fv] += siz[fu];
    }
    edge[u][v] += w, edge[v][u] += w;
  }
  int fr = find(1);
  if (siz[fr] != n) {
    cout << 0;
    return 0;
  }
  cout << Stoer_Wagner();
  return 0;
}
```

复杂度分析与优化

contract 操作的复杂度为 $O(|E|+|V|\log|V|)$。

一共进行 $O(|V|)$ 次 *contract*，总复杂度为 $O(|E||V|+|V|^2\log|V|)$。

根据 最短路 的经验，算法瓶颈在于找到权值最大的点。

在一次 *contract* 中需要找 $|V|$ 次堆顶，并递增地修改 $|E|$ 次权值。

斐波那契堆可以胜任 $O(\log|V|)$ 查找堆顶和 $O(1)$ 递增修改权值的工作，理论复杂度可以达到 $O(|E|+|V|\log|V|)$，但是由于斐波那契堆常数过大，码量高，实际应用价值偏低。

（实际测试中开 O2 还要卡评测波动才能过。）

参考资料与注释

[1] P5632【模板】Stoer–Wagner 算法

11.30 图的匹配

11.30.1 图匹配

Authors: accelsao, StudyingFather, t4rf9, wlbksy, yuhuoji

匹配或是**独立边集**是一张图中不具有公共端点的边的集合。在二分图中求匹配等价于网路流问题。

图匹配算法是信息学竞赛中常用的算法，总体分为最大匹配以及最大权匹配，先从二分图开始介绍，再进一步提出一般图的作法。

图的匹配

在图论中，假设图 $G=(V,E)$，其中 V 是点集，E 是边集。

一组两两没有公共点的边集 $M(M\in E)$ 称为这张图的**匹配**。

定义匹配的大小为其中边的数量 $|M|$，其中边数最大的 M 为**最大匹配**。

当图中的边带权的时候，边权和最大的为**最大权匹配**。

匹配中的边称为**匹配边**，反之称为**未匹配边**。

一个点如果属于 M 且为至多一条边的端点，称为**匹配点**，反之称为**未匹配点**。

- 极大匹配（maximal matching）：无法再增加匹配边的匹配。不一定是最大匹配。
- 最大匹配（maximum matching or maximum cardinality matching）：匹配边数量最多的匹配。最大匹配可能有不止一个，但最大匹配的边数是确定的，而且不可能超过图中定点数的一半。
- 最大权匹配（maximum weight matching）：加权图中，权值和最大的匹配。
- 最大权最大匹配（maximum weight maximum cardinality matching）：匹配数最多的前提下，边权和最大的匹配。即所有最大匹配中，边权和最大的匹配。
- 完美匹配（perfect matching）：所有点都属于匹配，同时也符合最大匹配。若图 G 为完全图且顶点数为偶数时，必然存在完美匹配。
- 近完美匹配（near-perfect matching）：发生在图的点数为奇数，刚好只有一个点不在匹配中，扣掉此点以后的图称为 factor-critical graph。

  极大匹配

  最大匹配

  最大权匹配

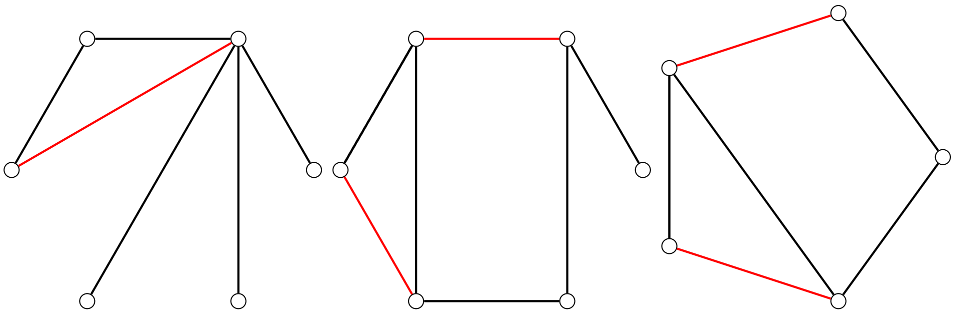


图 11.91 maximal matching

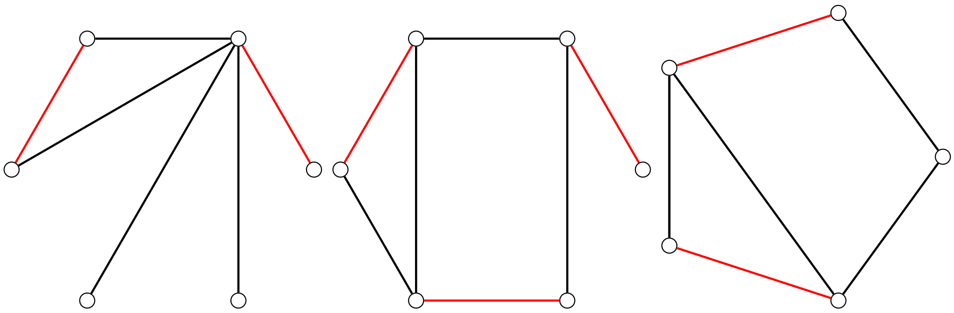


图 11.92 maximum cardinality matching

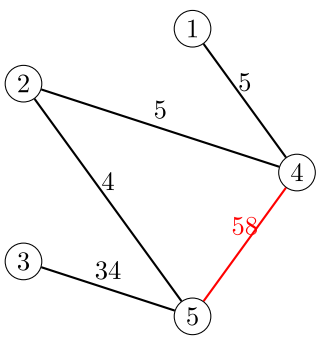


图 11.93 maximum weight matching

最大权最大匹配

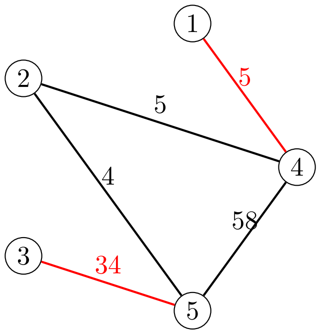


图 11.94 maximum weight maximum cardinality matching

二分图匹配

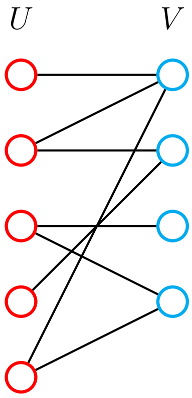


图 11.95 bi-graph

一张二分图上的匹配称作二分匹配

设 G 为二分图，若在 G 的子图 M 中，任意两条边都没有公共节点，那么称 M 为二分图 G 的一个匹配，且 M 的边数为匹配数。

完美匹配

设 $G=\langle V_1,V_2,E\rangle$ 为二分图，$|V_1|\le|V_2|$，M 为 G 中一个最大匹配，且 $|M|=|V_1|$，则称 M 为 V_1 到 V_2 的完美匹配。

霍尔定理

设二分图 $G=\langle V_1,V_2,E\rangle,|V_1|\le|V_2|$，则 G 中存在 V_1 到 V_2 的完美匹配当且仅当对于任意的 $S\subset V_1$，均有 $|S|\le|N(S)|$，其中 $N(S)=\Cup_{v_i\in S}N(V_i)$，是 S 的邻域。

最大匹配

寻找二分图边数最大的匹配称为最大匹配问题。

算法

组合优化中的一个基本问题是求**最大匹配（maximum matching）**。

二分图最大匹配

详见 二分图最大匹配 页面。

在无权二分图中，Hopcroft–Karp 算法可在 $O(\sqrt{V}E)$ 解决。

二分图最大权匹配

详见 二分图最大权匹配 页面。

在带权二分图中，可用 Hungarian 算法解决。如果在最短路搜寻中用 Bellman–Ford 算法，时间复杂度为 $O(V^2E)$，如果用 Dijkstra 算法或 Fibonacci heap，可用 $O(V^2 \log V + VE)$ 解决。

一般图最大匹配

详见 一般图最大匹配 页面。

无权一般图中，Edmonds' blossom 算法可在 $O(V^2E)$ 解决。

一般图最大权匹配

详见 一般图最大权匹配 页面。

带权一般图中，Edmonds' blossom 算法可在 $O(V^2E)$ 解决。

匹配算法的转换

用最大权最大匹配求最大权匹配

最大权最大匹配允许负边权 ($w(e) < 0$)，但是**最大权匹配**不会有负边权。

若一张图 G 其所有的边皆为负权，则其**最大权匹配** $M = \emptyset$。

调整边的权重

先将图 G 的所有负权边其权重 $w(e)$ 设为 0，再进行接下来的步骤。

完全图性质

当图 G 为完全图且没有负边权时，**最大权最大匹配** = **最大权匹配**。

所以把图 G 铺成完全图，铺上的边其权重为 0，计算**最大权最大匹配**后再把权重为 0 的边去除即可。如下图所示。

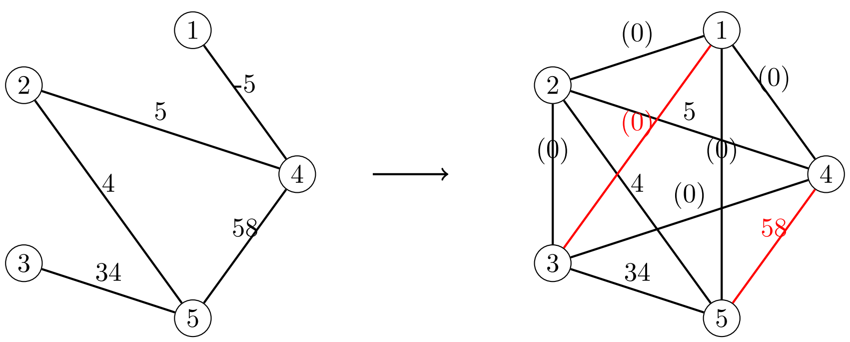


图 11.96 graph-match

用最大权匹配求最大权最大匹配

最大权匹配不会有负边权，且零边 ($w(e) = 0$) 可选可不选，但是**最大权最大匹配**允许负边权和零边。

调整边的权重

令 $K = \max\{|w(e)| : e \in E, w(e) \le 0\} + 1$，若没有负边权和零边则 $K = 0$。把图 G 中所有的边其权重 $w(e)$ 加上 K 产生一张新图 $G' = (V, E')$。得到的新图 G 不存在负边权和零边。

最大权最大匹配不一定等于**最大权匹配**，但如果把所有边的边权加上一个足够大的数 P，**最大权匹配**的结果就是**最大权最大匹配**。这样的 P 应该取多大？令 $P = \sum w(e) : e \in E'$，把图 G 中所有的边其权重 $w(e)$ 加上 P，产生一张新图 $G'' = (V, E'')$。

此时对图 G 进行**最大权匹配**，其结果可以对应原图 G 的**最大权最大匹配**。如下图所示。

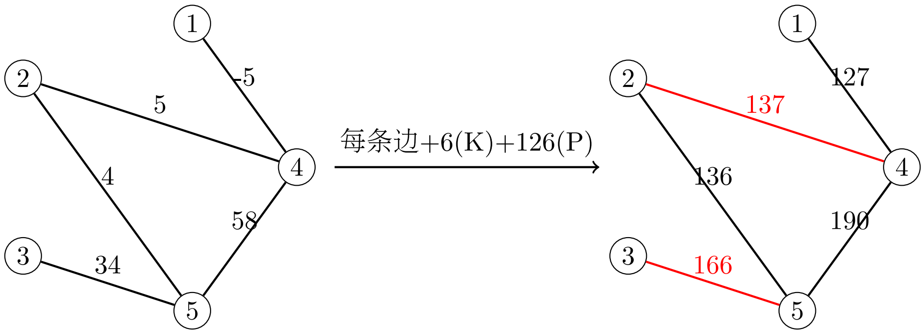


图 11.97 graph-match

参考资料

1. Wikiwand - Matching (graph theory)[1]
2. Wikiwand - Blossom algorithm[2]
3. 2015 年《浅谈图的匹配算法及其应用》- 陈胤伯
4. 演算法笔记 - Matching[3]
5. the-tourist/algo[4]
6. Bill Yang's Blog - 带花树学习笔记[5]
7. 二分图的最大匹配 、 完美匹配和匈牙利算法[6]
8. Wikiwand - Hopcroft–Karp algorithm[7]

参考资料与注释

[1] Wikiwand - Matching (graph theory)

[2] Wikiwand - Blossom algorithm

[3] 演算法笔记 - Matching

[4] the-tourist/algo

[5] Bill Yang's Blog - 带花树学习笔记

[6] 二分图的最大匹配、完美匹配和匈牙利算法

[7] Wikiwand - Hopcroft–Karp algorithm

11.30.2 增广路

Authors: accelsao, Chrogeek, t4rf9, yuhuoji

增广路定理 Berge's lemma

这是最大匹配的一个重要理论。

定义

- 交错路（alternating path）始于非匹配点且由匹配边与非匹配边交错而成。
- 增广路（augmenting path）是始于非匹配点且终于非匹配点（除了起始的点）的交错路。增广路中边的数量是奇数。

增广路上非匹配边比匹配边数量多 1，如果将增广路上的匹配边和未匹配边反转，则匹配数量会增加 1 且依然是交错路。

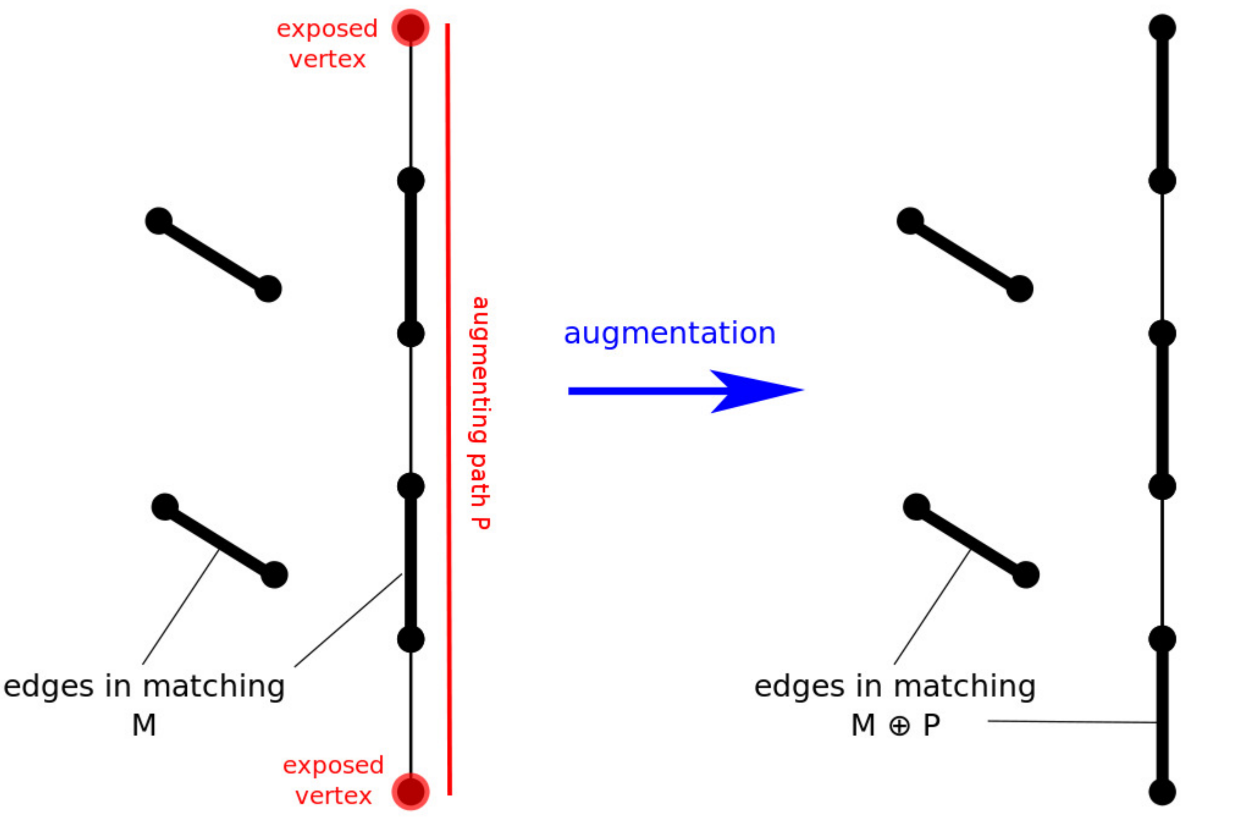


图 11.98 augment-1

如上图，匹配数从 2 增加为 3，匈牙利算法中只通过这样的方式增加匹配数量，称为**增广（Augment）**。

根据 Berge's lemma 当找不到增广路的时候，得到最大匹配。

过程

由此定理可知我们求最大匹配的核心思路。

核心思路

枚举所有未匹配点，找增广路径，直到找不到增广路径。

证明

事实上，对于每个点只要枚举一次就好，证明如下：

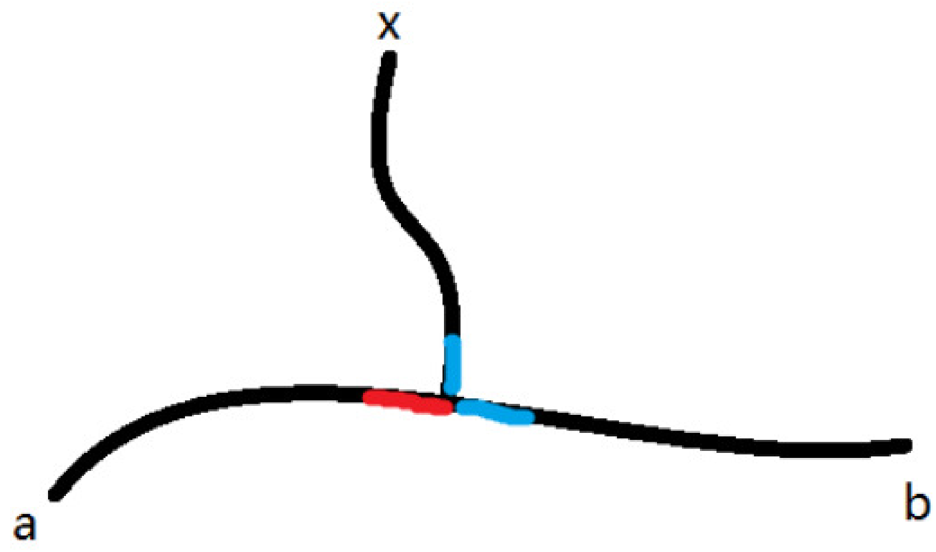


图 11.99 augment-2

假设某一轮沿着增广路 $a-b$ 增广后，新增了以未匹配点 x 为起点的增广路 P_x，则 P_x 必与 $a-b$ 有公共边（否则 P_x 不可能是因此次增广而新增的）。在 P_x 与 $a-b$ 取得公共边时，由于 $a-b$ 是交错路，意味着相交点在 $a-b$ 内的两邻边是不同类型的（图中以红和蓝表示）；因而增广前 x 就能走到 $a-b$ 中的某个未匹配点，说明此前已存在从 x 出发的增广路，即已枚举过的未匹配点不再可能作为增广路起点。

交错树

从未匹配点 r 进行 DFS 或 BFS 寻找增广路的过程中产生的树称为交错树，r 是交错树的根。设 $T=(V_t,E_t)$ 为再寻找增广路时产生的交错树。定义：

- 偶点（黑点）为树上深度为偶数的点。
- 奇点（白点）为树上深度为奇数的点。

下图展示了一个二分图和从未匹配点 1 开始寻找增广路时，形成的以 1 为根的交错树。

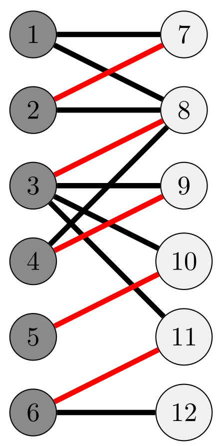


图 11.100 augment-3

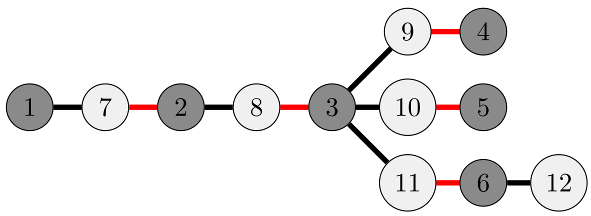


图 11.101 augment-4

11.30.3 二分图最大匹配

Authors: accelsao, thallium, Chrogeek, Enter-tainer, ksyx, StudyingFather, H-J-Granger, Henry-ZHR, countercurrent-time, william-song-shy, 5ab-juruo, XiaoQuQuSD

为了描述方便将两个集合分成左和右两个部分，所有匹配边都是横跨左右两个集合，可以假想成男女配对。

假设图有 n 个顶点，m 条边。

题目描述

给定一个二分图 G，即分左右两部分，各部分之间的点没有边连接，要求选出一些边，使得这些边没有公共顶点，且边的数量最大。

增广路算法 Augmenting Path Algorithm

因为增广路长度为奇数，路径起始点非左即右，所以我们先考虑从左边的未匹配点找增广路。注意到因为交错路的关系，增广路上的第奇数条边都是非匹配边，第偶数条边都是匹配边，于是左到右都是非匹配边，右到左都是匹配边。于是我们给二分图**定向**，问题转换成，有向图中从给定起点找一条简单路径走到某个未匹配点，此问题等价给定起始点 s 能否走到终点 t。那么只要从起始点开始 DFS 遍历直到找到某个未匹配点，$O(m)$。未找到增广路时，我们拓展的路也称为**交错树**。

性质

因为要枚举 n 个点，总复杂度为 $O(nm)$。

实现

```
struct augment_path {
  vector<vector<int> > g;
  vector<int> pa;  // 匹配
  vector<int> pb;
  vector<int> vis;  // 访问
  int n, m;         // 两个点集中的顶点数量
  int dfn;          // 时间戳记
  int res;          // 匹配数

  augment_path(int _n, int _m) : n(_n), m(_m) {
    assert(0 <= n && 0 <= m);
    pa = vector<int>(n, -1);
    pb = vector<int>(m, -1);
    vis = vector<int>(n);
    g.resize(n);
```

```
    res = 0;
    dfn = 0;
  }

  void add(int from, int to) {
    assert(0 <= from && from < n && 0 <= to && to < m);
    g[from].push_back(to);
  }

  bool dfs(int v) {
    vis[v] = dfn;
    for (int u : g[v]) {
      if (pb[u] == -1) {
        pb[u] = v;
        pa[v] = u;
        return true;
      }
    }
    for (int u : g[v]) {
      if (vis[pb[u]] != dfn && dfs(pb[u])) {
        pa[v] = u;
        pb[u] = v;
        return true;
      }
    }
    return false;
  }

  int solve() {
    while (true) {
      dfn++;
      int cnt = 0;
      for (int i = 0; i < n; i++) {
        if (pa[i] == -1 && dfs(i)) {
          cnt++;
        }
      }
      if (cnt == 0) {
        break;
      }
      res += cnt;
    }
    return res;
  }
};
```

转为网络最大流模型

二分图最大匹配可以转换成网络流模型。

将源点连上左边所有点，右边所有点连上汇点，容量皆为 1。原来的每条边从左往右连边，容量也皆为 1，最大流即最大匹配。

如果使用 Dinic 算法 求该网络的最大流，可在 $O(\sqrt{n}m)$ 求出。

Dinic 算法分成两部分，第一部分用 $O(m)$ 时间 BFS 建立网络流，第二步是 $O(nm)$ 时间 DFS 进行增广。

但因为容量为 1，所以实际时间复杂度为 $O(m)$。

接下来前 $O(\sqrt{n})$ 轮，复杂度为 $O(\sqrt{n}m)$。$O(\sqrt{n})$ 轮以后，每条增广路径长度至少 $\sqrt{n}$，而这样的路径不超过 $\sqrt{n}$，所以此时最多只需要跑 $\sqrt{n}$ 轮，整体复杂度为 $O(\sqrt{n}m)$。

代码可以参考 Dinic 算法 的参考实现，这里不再给出。

补充

二分图最小点覆盖（König 定理）

最小点覆盖：选最少的点，满足每条边至少有一个端点被选。

二分图中，最小点覆盖 = 最大匹配。

" 证明"

将二分图点集分成左右两个集合，使得所有边的两个端点都不在一个集合。

考虑如下构造：从左侧未匹配的节点出发，按照匈牙利算法中增广路的方式走，即先走一条未匹配边，再走一条匹配边。由于已经求出了最大匹配，所以这样的「增广路」一定以匹配边结束，即增广路是不完整的。在所有经过这样「增广路」的节点上打标记。则最后构造的集合是：所有左侧未打标记的节点和所有右侧打了标记的节点。

首先，这个集合的大小等于最大匹配。左边未打标记的点都一定对应着一个匹配边（否则会以这个点为起点开始标记），右边打了标记的节点一定在一条不完整的增广路上，也会对应一个匹配边。假设存在一条匹配边左侧标记了，右侧没标记，左边的点只能是通过另一条匹配边走过来，此时左边的点有两条匹配边，不符合最大匹配的规定；假设存在一条匹配边左侧没标记，右侧标记了，那就会从右边的点沿着这条匹配边走过来，从而左侧也有标记。因此，每一条匹配的边两侧一定都有标记（在不完整的增广路上）或都没有标记，匹配边的两个节点中必然有一个被选中。

其次，这个集合是一个点覆盖。由于我们的构造方式是：所有左侧未打标记的节点和所有右侧打了标记的节点。假设存在左侧打标记且右侧没打标记的边，对于匹配边，上一段已经说明其不存在，对于非匹配边，右端点一定会由这条非匹配边经过，从而被打上标记。因此，这样的构造能够覆盖所有边。

同时，不存在更小的点覆盖。为了覆盖最大匹配的所有边，至少要有最大匹配边数的点数。

二分图最大独立集

最大独立集：选最多的点，满足两两之间没有边相连。

因为在最小点覆盖中，任意一条边都被至少选了一个顶点，所以对于其点集的补集，任意一条边都被至多选了一个顶点，所以不存在边连接两个点集中的点，且该点集最大。因此二分图中，最大独立集 $= n-$ 最小点覆盖。

习题

"UOJ #78. 二分图最大匹配[1]"

模板题

```cpp
#include <bits/stdc++.h>
using namespace std;

struct augment_path {
  vector<vector<int> > g;
  vector<int> pa;  // 匹配
  vector<int> pb;
  vector<int> vis;  // 访问
  int n, m;         // 顶点和边的数量
  int dfn;          // 时间戳记
  int res;          // 匹配数
```

```
  augment_path(int _n, int _m) : n(_n), m(_m) {
    assert(0 <= n && 0 <= m);
    pa = vector<int>(n, -1);
    pb = vector<int>(m, -1);
    vis = vector<int>(n);
    g.resize(n);
    res = 0;
    dfn = 0;
  }

  void add(int from, int to) {
    assert(0 <= from && from < n && 0 <= to && to < m);
    g[from].push_back(to);
  }

  bool dfs(int v) {
    vis[v] = dfn;
    for (int u : g[v]) {
      if (pb[u] == -1) {
        pb[u] = v;
        pa[v] = u;
        return true;
      }
    }
    for (int u : g[v]) {
      if (vis[pb[u]] != dfn && dfs(pb[u])) {
        pa[v] = u;
        pb[u] = v;
        return true;
      }
    }
    return false;
  }

  int solve() {
    while (true) {
      dfn++;
      int cnt = 0;
      for (int i = 0; i < n; i++) {
        if (pa[i] == -1 && dfs(i)) {
          cnt++;
        }
      }
      if (cnt == 0) {
        break;
      }
      res += cnt;
    }
    return res;
  }
};

int main() {
```

```
  int n, m, e;
  cin >> n >> m >> e;
  augment_path solver(n, m);
  int u, v;
  for (int i = 0; i < e; i++) {
    cin >> u >> v;
    u--, v--;
    solver.add(u, v);
  }
  cout << solver.solve() << "\n";
  for (int i = 0; i < n; i++) {
    cout << solver.pa[i] + 1 << " ";
  }
  cout << "\n";
}
```

"P1640 [SCOI2010] 连续攻击游戏[2]"

None

"Codeforces 1139E - Maximize Mex[3]"

None

参考资料

1. http://www.matrix67.com/blog/archives/116[4]

参考资料与注释

[1] UOJ #78. 二分图最大匹配

[2] P1640 [SCOI2010] 连续攻击游戏

[3] Codeforces 1139E - Maximize Mex

[4] http://www.matrix67.com/blog/archives/116

11.30.4 二分图最大权匹配

Authors: accelsao, Enter-tainer, guodong2005, StudyingFather, Backl1ght, Chrogeek, H-J-Granger, Henry-ZHR

二分图的最大权匹配是指二分图中边权和最大的匹配。

Hungarian Algorithm（Kuhn–Munkres Algorithm）

匈牙利算法又称为 **KM** 算法，可以在 $O(n^3)$ 时间内求出二分图的**最大权完美匹配**。

考虑到二分图中两个集合中的点并不总是相同，为了能应用 KM 算法解决二分图的最大权匹配，需要先作如下处理：将两个集合中点数比较少的补点，使得两边点数相同，再将不存在的边权重设为 0，这种情况下，问题就转换成求**最大权完美匹配问题**，从而能应用 KM 算法求解。

" 可行顶标"

给每个节点 i 分配一个权值 $l(i)$，对于所有边 (u,v) 满足 $w(u,v) \le l(u)+l(v)$。

" 相等子图"

在一组可行顶标下原图的生成子图，包含所有点但只包含满足 $w(u,v)=l(u)+l(v)$ 的边 (u,v)。

" 定理 1：对于某组可行顶标，如果其相等子图存在完美匹配，那么，该匹配就是原二分图的最大权完美匹配。"

证明 1.

考虑原二分图任意一组完美匹配 M，其边权和为

$val(M)=\sum_{(u,v)\in M} w(u,v) \le \sum_{(u,v)\in M} l(u)+l(v) \le \sum_{i=1}^{n} l(i)$

任意一组可行顶标的相等子图的完美匹配 M' 的边权和

$val(M')=\sum_{(u,v)\in M} l(u)+l(v)=\sum_{i=1}^{n} l(i)$

即任意一组完美匹配的边权和都不会大于 $val(M')$，那个 M' 就是最大权匹配。

有了定理 1，我们的目标就是透过不断的调整可行顶标，使得相等子图是完美匹配。

因为两边点数相等，假设点数为 n，$lx(i)$ 表示左边第 i 个点的顶标，$ly(i)$ 表示右边第 i 个点的顶标，$w(u,v)$ 表示左边第 u 个点和右边第 v 个点之间的权重。

首先初始化一组可行顶标，例如

$lx(i)=\max_{1\le j\le n}\{w(i,j)\},\ ly(i)=0$

然后选一个未匹配点，如同最大匹配一样求增广路。找到增广路就增广，否则，会得到一个交错树。

令 S，T 表示二分图左边右边在交错树中的点，S'，T' 表示不在交错树中的点。

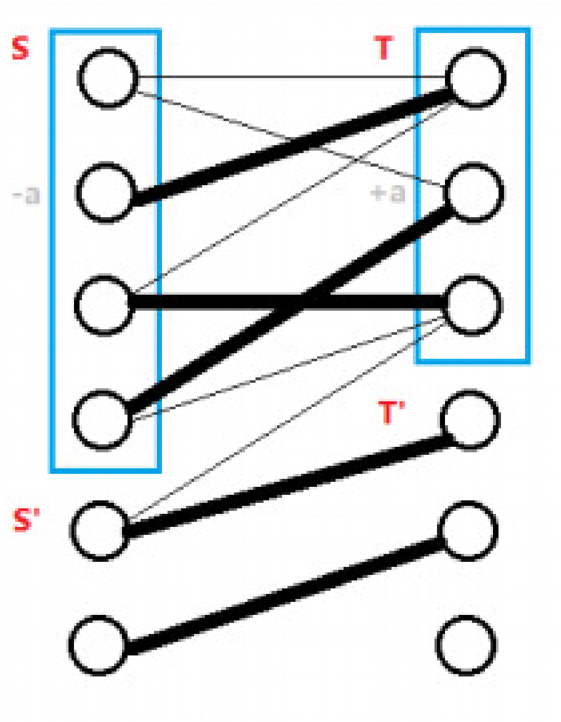


图 11.102 bigraph-weight-match-1

在相等子图中：

- $S-T'$ 的边不存在，否则交错树会增长。
- $S'-T$ 一定是非匹配边，否则他就属于 S。

假设给 S 中的顶标 $-a$，给 T 中的顶标 $+a$，可以发现

- $S-T$ 边依然存在相等子图中。
- $S'-T'$ 没变化。
- $S-T'$ 中的 $lx+ly$ 有所减少，可能加入相等子图。
- $S'-T$ 中的 $lx+ly$ 会增加，所以不可能加入相等子图。

所以这个 a 值的选择，显然得是 $S-T'$ 当中最小的边权，

$a=\min\{lx(u)+ly(v)-w(u,v)|u\in S,v\in T'\}$。

当一条新的边 (u,v) 加入相等子图后有两种情况

- v 是未匹配点，则找到增广路
- v 和 S' 中的点已经匹配

这样至多修改 n 次顶标后，就可以找到增广路。

每次修改顶标的时候，交错树中的边不会离开相等子图，那么我们直接维护这棵树。

我们对 T 中的每个点 v 维护

$slack(v)=\min\{lx(u)+ly(v)-w(u,v)|u\in S\}$。

所以可以在 $O(n)$ 算出顶标修改值 a

$a=\min\{slack(v)|v\in T'\}$

交错树新增一个点进入 S 的时候需要 $O(n)$ 更新 $slack(v)$。修改顶标需要 $O(n)$ 给每个 $slack(v)$ 减去 a。只要交错树找到一个未匹配点，就找到增广路。

一开始枚举 n 个点找增广路，为了找增广路需要延伸 n 次交错树，每次延伸需要 n 次维护，共 $O(n^3)$。

" 参考代码"

```
template <typename T>
struct hungarian {  // km
  int n;
  vector<int> matchx;  // 左集合对应的匹配点
  vector<int> matchy;  // 右集合对应的匹配点
  vector<int> pre;     // 连接右集合的左点
  vector<bool> visx;   // 拜访数组左
  vector<bool> visy;   // 拜访数组右
  vector<T> lx;
  vector<T> ly;
  vector<vector<T> > g;
  vector<T> slack;
  T inf;
  T res;
  queue<int> q;
  int org_n;
  int org_m;

  hungarian(int _n, int _m) {
    org_n = _n;
    org_m = _m;
    n = max(_n, _m);
    inf = numeric_limits<T>::max();
    res = 0;
    g = vector<vector<T> >(n, vector<T>(n));
```

```
    matchx = vector<int>(n, -1);
    matchy = vector<int>(n, -1);
    pre = vector<int>(n);
    visx = vector<bool>(n);
    visy = vector<bool>(n);
    lx = vector<T>(n, -inf);
    ly = vector<T>(n);
    slack = vector<T>(n);
  }

  void addEdge(int u, int v, int w) {
    g[u][v] = max(w, 0);  // 负值还不如不匹配因此设为 0 不影响
  }

  bool check(int v) {
    visy[v] = true;
    if (matchy[v] != -1) {
      q.push(matchy[v]);
      visx[matchy[v]] = true;  // in S
      return false;
    }
    // 找到新的未匹配点更新匹配点 pre 数组记录着" 非匹配边" 上与之相连的点
    while (v != -1) {
      matchy[v] = pre[v];
      swap(v, matchx[pre[v]]);
    }
    return true;
  }

  void bfs(int i) {
    while (!q.empty()) {
      q.pop();
    }
    q.push(i);
    visx[i] = true;
    while (true) {
      while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int v = 0; v < n; v++) {
          if (!visy[v]) {
            T delta = lx[u] + ly[v] - g[u][v];
            if (slack[v] >= delta) {
              pre[v] = u;
              if (delta) {
                slack[v] = delta;
              } else if (check(v)) {  // delta=0 代表有机会加入相等子图找增广路
                                      // 找到就 return 重建交错树
                return;
              }
            }
          }
        }
      }
```

```
      // 没有增广路修改顶标
      T a = inf;
      for (int j = 0; j < n; j++) {
        if (!visy[j]) {
          a = min(a, slack[j]);
        }
      }
      for (int j = 0; j < n; j++) {
        if (visx[j]) {  // S
          lx[j] -= a;
        }
        if (visy[j]) {  // T
          ly[j] += a;
        } else {  // T'
          slack[j] -= a;
        }
      }
      for (int j = 0; j < n; j++) {
        if (!visy[j] && slack[j] == 0 && check(j)) {
          return;
        }
      }
    }
  }

  void solve() {
    // 初始顶标
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        lx[i] = max(lx[i], g[i][j]);
      }
    }

    for (int i = 0; i < n; i++) {
      fill(slack.begin(), slack.end(), inf);
      fill(visx.begin(), visx.end(), false);
      fill(visy.begin(), visy.end(), false);
      bfs(i);
    }

    // custom
    for (int i = 0; i < n; i++) {
      if (g[i][matchx[i]] > 0) {
        res += g[i][matchx[i]];
      } else {
        matchx[i] = -1;
      }
    }
    cout << res << "\n";
    for (int i = 0; i < org_n; i++) {
      cout << matchx[i] + 1 << " ";
    }
    cout << "\n";
  }
};
```

Dynamic Hungarian Algorithm

原论文 The Dynamic Hungarian Algorithm for the Assignment Problem with Changing Costs[1]

伪代码更清晰的论文 A Fast Dynamic Assignment Algorithm for Solving Resource Allocation Problems[2]

相关 OJ 问题 DAP[3]

" 算法思路"

1. 修改单点 u_i 和所有 v_j 之间的权重，即权重矩阵中的一行
   - 修改顶标 $lx(u_i) = max(w_{ij} - v_j), \forall j$
   - 删除 u_i 相关的匹配
2. 修改所有 u_i 和单点 v_j 之间的权重，即权重矩阵中的一列
   - 修改顶标 $ly(v_j) = max(w_{ij} - u_i), \forall i$
   - 删除 v_j 相关的匹配
3. 修改单点 u_i 和单点 v_j 之间的权重，即权重矩阵中的单个元素
   - 做 1 或 2 两种操作之一即可
4. 添加某一单点 u_i，或者某一单点 v_j，即在权重矩阵中添加或者删除一行或者一列
   - 对应地做 1 或 2 即可，注意此处加点操作仅为加点，不额外设定权重值，新加点与其他点的权重为 0.

" 算法证明"

- 设原图为 G，左右两边的顶标为 α^i 和 β^j，可行顶标为 l，那 G_l 是 G 的一个子图，包含图 G 中满足 $w_{ij} = alpha_i + beta_j$ 的点和边。
- 在上面匈牙利算法的部分，定理一证明了：对于某组可行顶标，如果其相等子图存在完美匹配，那么，该匹配就是原二分图的最大权完美匹配。
- 假设原来的最优匹配是 M^*，当一个修改发生的时候，我们会根据规则更新可行顶标，更新后的顶标设为 α^{i^*}，或者 β^{j^*}，会出现以下情况：
  1. 权重矩阵的一整行被修改了，设被修改的行为 i^* 行，即 v_{i^*} 的所有边被修改了，所以 v_{i^*} 原来的顶标可能不满足条件，因为我们需要 $w_{i^*j} \leq alpha_{i^*} + beta_j$，但对于其他的 u_j 来说，除了 i^* 相关的边，他们的边权是不变的，因此他们的顶标都是合法的，所以算法中修改了 v_{i^*} 相关的顶标使得这组顶标是一组可行顶标。
  2. 权重矩阵的一整列被修改了，同理可得算法修改顶标使得这组顶标是一组可行顶标。
  3. 修改权重矩阵某一元素，任意修改其中一个顶标即可满足顶标条件
- 每一次权重矩阵被修改，都关系到一个特定节点，这个节点可能是左边的也可能是右边的，因此我们直接记为 x，这个节点和某个节点 y 在原来的最优匹配中匹配上了。每一次修改操作，最多让这一对节点 unpair，因此我们只要跑一轮匈牙利算法中的搜索我们就能得到一个新的 match，而根据定理一，新跑出来的 match 是最优的。

以下代码应该为论文 2 作者提交的代码（以下代码为最大化权重版本，原始论文中为最小化 cost）

" 动态匈牙利算法参考代码"

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <list>
```

```
using namespace std;
typedef long long LL;
const LL INF = (LL)1e15;
const int MAXV = 105;

int N, mateS[MAXV], mateT[MAXV], p[MAXV];
LL u[MAXV], v[MAXV], slack[MAXV];
LL W[MAXV][MAXV];
bool m[MAXV];
list<int> Q;

void readMatrix() {
  scanf("%d", &N);
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++) scanf("%lld", &W[i][j]);
}

void initHungarian() {
  memset(mateS, -1, sizeof(mateS));
  memset(mateT, -1, sizeof(mateT));
  for (int i = 0; i < N; i++) {
    u[i] = -INF;
    for (int j = 0; j < N; j++) u[i] = max(u[i], W[i][j]);
    v[i] = 0;
  }
}

void augment(int j) {
  int i, next;
  do {
    i = p[j];
    mateT[j] = i;
    next = mateS[i];
    mateS[i] = j;
    if (next != -1) j = next;
  } while (next != -1);
}

LL hungarian() {
  int nres = 0;
  for (int i = 0; i < N; i++)
    if (mateS[i] == -1) nres++;

  while (nres > 0) {
    for (int i = 0; i < N; i++) {
      m[i] = false;
      p[i] = -1;
      slack[i] = INF;
    }
    bool aug = false;
    Q.clear();
    for (int i = 0; i < N; i++)
      if (mateS[i] == -1) {
        Q.push_back(i);
```

```
      break;
    }

  do {
    int i, j;
    i = Q.front();
    Q.pop_front();
    m[i] = true;
    j = 0;

    while (aug == false && j < N) {
      if (mateS[i] != j) {
        LL minSlack = u[i] + v[j] - W[i][j];
        if (minSlack < slack[j]) {
          slack[j] = minSlack;
          p[j] = i;
          if (slack[j] == 0) {
            if (mateT[j] == -1) {
              augment(j);
              aug = true;
              nres--;
            } else
              Q.push_back(mateT[j]);
          }
        }
      }
      j++;
    }

    if (aug == false && Q.size() == 0) {
      LL minSlack = INF;
      for (int k = 0; k < N; k++)
        if (slack[k] > 0) minSlack = min(minSlack, slack[k]);
      for (int k = 0; k < N; k++)
        if (m[k]) u[k] -= minSlack;

      int x = -1;
      bool X[MAXV];
      for (int k = 0; k < N; k++)
        if (slack[k] == 0)
          v[k] += minSlack;
        else {
          slack[k] -= minSlack;
          if (slack[k] == 0 && mateT[k] == -1) x = k;
          if (slack[k] == 0)
            X[k] = true;
          else
            X[k] = false;
        }

      if (x == -1) {
        for (int k = 0; k < N; k++)
          if (X[k]) Q.push_back(mateT[k]);
      } else {
```

```
          augment(x);
          aug = true;
          nres--;
        }
      }
    } while (aug == false);
  }

  LL ans = 0;
  for (int i = 0; i < N; i++) ans += (u[i] + v[i]);
  return ans;
}

void dynamicHungarian() {
  char type[2];
  LL w;
  int i, j;

  scanf("%s", type);
  if (type[0] == 'C') {
    scanf("%d%d%lld", &i, &j, &w);
    if ((w < W[i][j]) && (mateS[i] == j)) {
      W[i][j] = w;
      if (mateS[i] != -1) {
        mateT[mateS[i]] = -1;
        mateS[i] = -1;
      }
    } else if ((w > W[i][j]) && (u[i] + v[j] < w)) {
      W[i][j] = w;
      u[i] = -INF;
      for (int c = 0; c < N; c++) u[i] = max(u[i], W[i][c] - v[c]);
      if (mateS[i] != j) {
        mateT[mateS[i]] = -1;
        mateS[i] = -1;
      }
    } else
      W[i][j] = w;
  } else if (type[0] == 'X') {
    scanf("%d", &i);
    for (int c = 0; c < N; c++) scanf("%lld", &W[i][c]);
    if (mateS[i] != -1) {
      mateT[mateS[i]] = -1;
      mateS[i] = -1;
    }
    u[i] = -INF;
    for (int c = 0; c < N; c++) u[i] = max(u[i], W[i][c] - v[c]);
  } else if (type[0] == 'Y') {
    scanf("%d", &j);
    for (int r = 0; r < N; r++) scanf("%lld", &W[r][j]);
    if (mateT[j] != -1) {
      mateS[mateT[j]] = -1;
      mateT[j] = -1;
    }
    v[j] = -INF;
```

```
      for (int r = 0; r < N; r++) v[j] = max(v[j], W[r][j] - u[r]);
    } else if (type[0] == 'A') {
      i = j = N++;
      u[i] = -INF;
      for (int c = 0; c < N; c++) u[i] = max(u[i], W[i][c] - v[c]);
      v[j] = -INF;
      for (int r = 0; r < N; r++) v[j] = max(v[j], W[r][j] - u[r]);
    } else if (type[0] == 'Q')
      printf("%lld\n", hungarian());
}

int main() {
  readMatrix();
  initHungarian();
  LL ans = hungarian();
  int M;
  scanf("%d", &M);
  while (M--) dynamicHungarian();
  return 0;
}
```

转化为费用流模型

在图中新增一个源点和一个汇点。

从源点向二分图的每个左部点连一条流量为 1，费用为 0 的边，从二分图的每个右部点向汇点连一条流量为 1，费用为 0 的边。

接下来对于二分图中每一条连接左部点 u 和右部点 v，边权为 w 的边，则连一条从 u 到 v，流量为 1，费用为 w 的边。

求这个网络的 最大费用最大流 即可得到答案。

习题

"UOJ #80. 二分图最大权匹配[4]"

模板题

```
#include <bits/stdc++.h>
using namespace std;

template <typename T>
struct hungarian {  // km
  int n;
  vector<int> matchx;
  vector<int> matchy;
  vector<int> pre;
  vector<bool> visx;
  vector<bool> visy;
  vector<T> lx;
  vector<T> ly;
  vector<vector<T> > g;
  vector<T> slack;
  T inf;
  T res;
```

```
  queue<int> q;
  int org_n;
  int org_m;

  hungarian(int _n, int _m) {
    org_n = _n;
    org_m = _m;
    n = max(_n, _m);
    inf = numeric_limits<T>::max();
    res = 0;
    g = vector<vector<T> >(n, vector<T>(n));
    matchx = vector<int>(n, -1);
    matchy = vector<int>(n, -1);
    pre = vector<int>(n);
    visx = vector<bool>(n);
    visy = vector<bool>(n);
    lx = vector<T>(n, -inf);
    ly = vector<T>(n);
    slack = vector<T>(n);
  }

  void addEdge(int u, int v, int w) {
    g[u][v] = max(w, 0);  // 负值还不如不匹配因此设为 0 不影响
  }

  bool check(int v) {
    visy[v] = true;
    if (matchy[v] != -1) {
      q.push(matchy[v]);
      visx[matchy[v]] = true;
      return false;
    }
    while (v != -1) {
      matchy[v] = pre[v];
      swap(v, matchx[pre[v]]);
    }
    return true;
  }

  void bfs(int i) {
    while (!q.empty()) {
      q.pop();
    }
    q.push(i);
    visx[i] = true;
    while (true) {
      while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int v = 0; v < n; v++) {
          if (!visy[v]) {
            T delta = lx[u] + ly[v] - g[u][v];
            if (slack[v] >= delta) {
              pre[v] = u;
```

```
            if (delta) {
              slack[v] = delta;
            } else if (check(v)) {
              return;
            }
          }
        }
      }
    }
    // 没有增广路修改顶标
    T a = inf;
    for (int j = 0; j < n; j++) {
      if (!visy[j]) {
        a = min(a, slack[j]);
      }
    }
    for (int j = 0; j < n; j++) {
      if (visx[j]) {  // S
        lx[j] -= a;
      }
      if (visy[j]) {  // T
        ly[j] += a;
      } else {  // T'
        slack[j] -= a;
      }
    }
    for (int j = 0; j < n; j++) {
      if (!visy[j] && slack[j] == 0 && check(j)) {
        return;
      }
    }
  }
}

void solve() {
  // 初始顶标
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      lx[i] = max(lx[i], g[i][j]);
    }
  }

  for (int i = 0; i < n; i++) {
    fill(slack.begin(), slack.end(), inf);
    fill(visx.begin(), visx.end(), false);
    fill(visy.begin(), visy.end(), false);
    bfs(i);
  }

  // custom
  for (int i = 0; i < n; i++) {
    if (g[i][matchx[i]] > 0) {
      res += g[i][matchx[i]];
    } else {
```

```
            matchx[i] = -1;
          }
        }
        cout << res << "\n";
        for (int i = 0; i < org_n; i++) {
          cout << matchx[i] + 1 << " ";
        }
        cout << "\n";
      }
    };

    int main() {
      ios::sync_with_stdio(0), cin.tie(0);
      int n, m, e;
      cin >> n >> m >> e;

      hungarian<long long> solver(n, m);

      int u, v, w;
      for (int i = 0; i < e; i++) {
        cin >> u >> v >> w;
        u--, v--;
        solver.addEdge(u, v, w);
      }
      solver.solve();
    }
```

参考资料与注释

[1] The Dynamic Hungarian Algorithm for the Assignment Problem with Changing Costs

[2] A Fast Dynamic Assignment Algorithm for Solving Resource Allocation Problems

[3] DAP

[4] UOJ #80. 二分图最大权匹配

11.30.5 一般图最大匹配

Authors: H-J-Granger, accelsao, Ir1d, Early0v0, Henry-ZHR, HeliumOI, AntiLeaf

带花树算法（Blossom Algorithm）

开花算法(Blossom Algorithm，也被称做带花树)可以解决一般图最大匹配问题(maximum cardinality matchings)。此算法由 Jack Edmonds 在 1961 年提出。经过一些修改后也可以解决一般图最大权匹配问题。此算法是第一个给出证明说最大匹配有多项式复杂度。

一般图匹配和二分图匹配（bipartite matching）不同的是，图可能存在奇环。

以此图为例，若直接取反（匹配边和未匹配边对调），会使得取反后的 M 不合法，某些点会出现在两条匹配上，而问题就出在奇环。

下面考虑一般图的增广算法。从二分图的角度出发，每次枚举一个未匹配点，设出发点为根，标记为「**o**」，接下

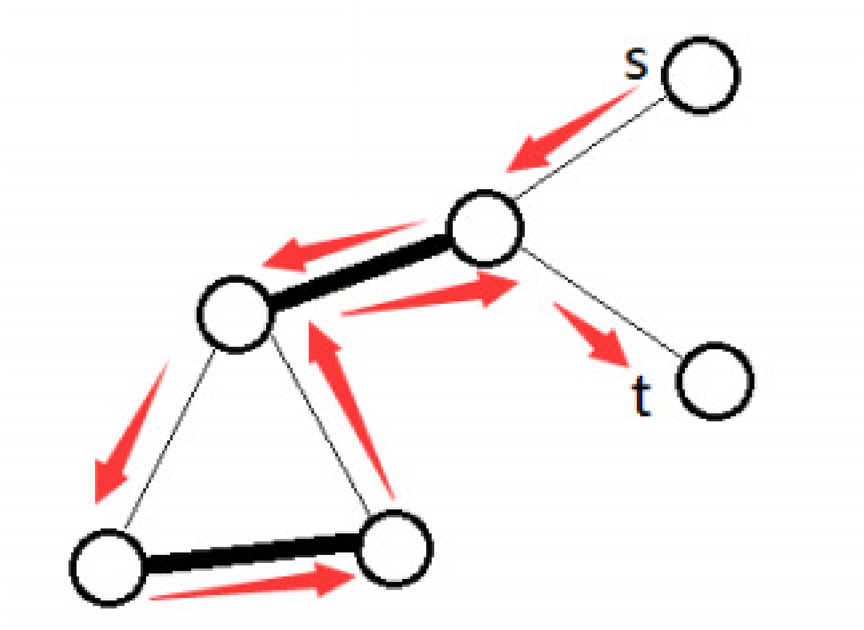


图 11.103 general-matching-1

来交错标记「**o**」和「**i**」，不难发现「**i**」到「**o**」这段边是匹配边。

假设当前点是 v，相邻点为 u，可以分为以下两种情况：

1. u 未拜访过，当 u 是未匹配点，则找到增广路径，否则从 u 的配偶找增广路。
2. u 已拜访过，遇到标记「o」代表需要**缩花**，否则代表遇到偶环，跳过。

遇到偶环的情况，将他视为二分图解决，故可忽略。**缩花**后，再新图中继续找增广路。

设原图为 G，**缩花**后的图为 G'，我们只需要证明：

1. 若 G 存在增广路，G' 也存在。
2. 若 G' 存在增广路，G 也存在。

设非树边（形成环的那条边）为 (u,v)，定义花根 $h = LCA(u,v)$。奇环是交替的，有且仅有 h 的两条邻边类型相同，都是非匹配边。那么进入 h 的树边肯定是匹配边，环上除了 h 以外其他点往环外的边都是非匹配边。

观察可知，从环外的边出去有两种情况，顺时针或逆时针。

于是**缩花**与**不缩花**都不影响正确性。

实作上找到**花**以后我们不需要真的**缩花**，可以用数组纪录每个点在以哪个点为根的那朵花中。

复杂度分析 Complexity Analysis

每次找增广路，遍历所有边，遇到**花**会维护**花**上的点，$O(|E|^2)$。

枚举所有未匹配点做增广路，总共 $O(|V||E|^2)$。

参考代码

" 参考代码"

```
// graph
template <typename T>
class graph {
```

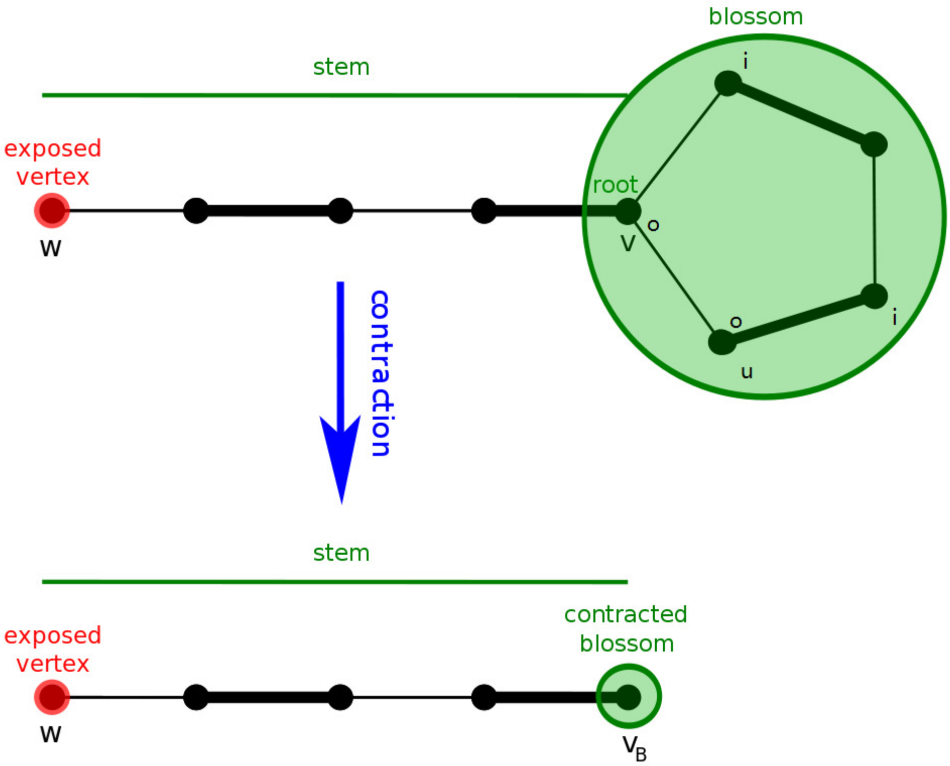


图 11.104 general-matching-2

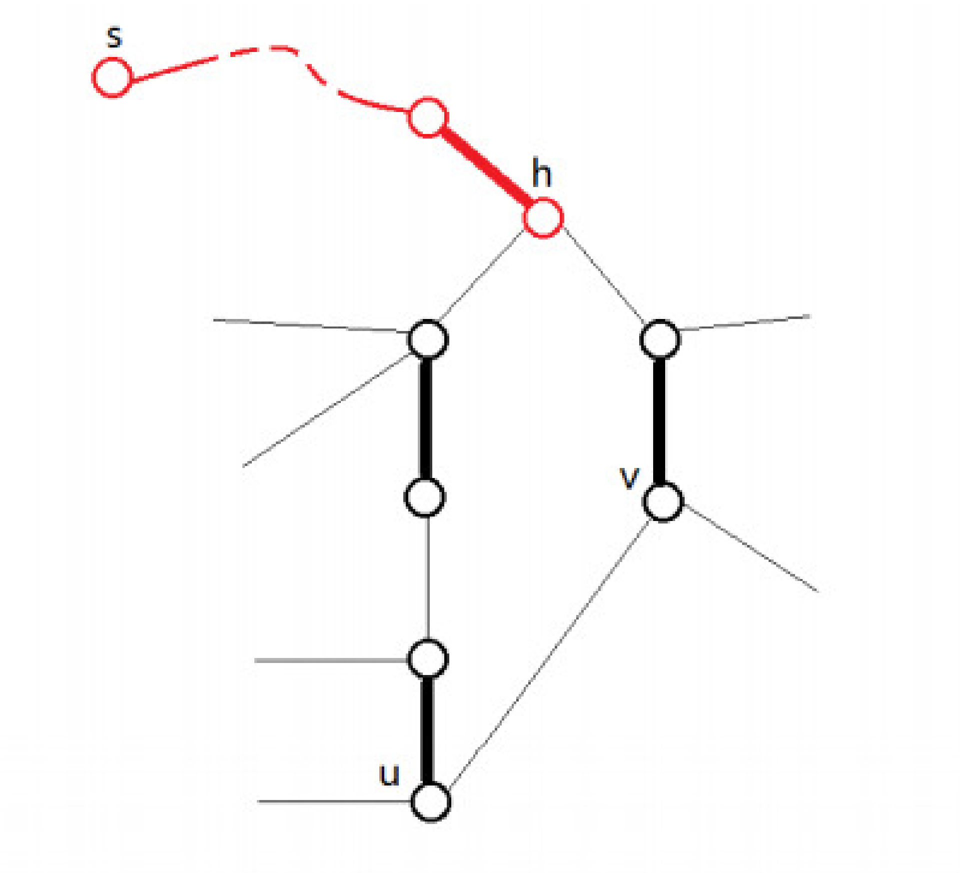


图 11.105 general-matching-3

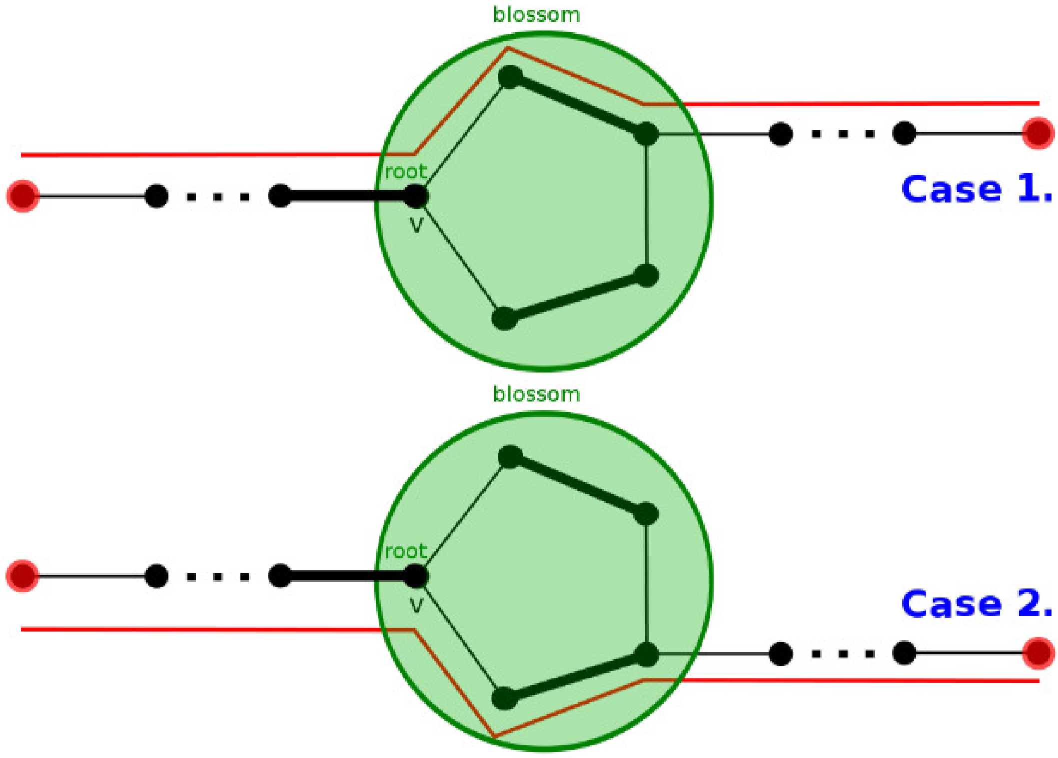


图 11.106 general-matching-4

```
 public:
  struct edge {
    int from;
    int to;
    T cost;
  };

  vector<edge> edges;
  vector<vector<int> > g;
  int n;

  graph(int _n) : n(_n) { g.resize(n); }

  virtual int add(int from, int to, T cost) = 0;
};

// undirectedgraph
template <typename T>
class undirectedgraph : public graph<T> {
 public:
  using graph<T>::edges;
  using graph<T>::g;
  using graph<T>::n;

  undirectedgraph(int _n) : graph<T>(_n) {}

  int add(int from, int to, T cost = 1) {
    assert(0 <= from && from < n && 0 <= to && to < n);
    int id = (int)edges.size();
```

```
    g[from].push_back(id);
    g[to].push_back(id);
    edges.push_back({from, to, cost});
    return id;
  }
};

// blossom / find_max_unweighted_matching
template <typename T>
vector<int> find_max_unweighted_matching(const undirectedgraph<T> &g) {
  std::mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
  vector<int> match(g.n, -1);   // 匹配
  vector<int> aux(g.n, -1);     // 时间戳记
  vector<int> label(g.n);       // 「o」或「i」
  vector<int> orig(g.n);        // 花根
  vector<int> parent(g.n, -1);  // 父节点
  queue<int> q;
  int aux_time = -1;

  auto lca = [&](int v, int u) {
    aux_time++;
    while (true) {
      if (v != -1) {
        if (aux[v] == aux_time) {  // 找到拜访过的点也就是 LCA
          return v;
        }
        aux[v] = aux_time;
        if (match[v] == -1) {
          v = -1;
        } else {
          v = orig[parent[match[v]]];  // 以匹配点的父节点继续寻找
        }
      }
      swap(v, u);
    }
  };  // lca

  auto blossom = [&](int v, int u, int a) {
    while (orig[v] != a) {
      parent[v] = u;
      u = match[v];
      if (label[u] == 1) {  // 初始点设为「o」找增广路
        label[u] = 0;
        q.push(u);
      }
      orig[v] = orig[u] = a;  // 缩花
      v = parent[u];
    }
  };  // blossom

  auto augment = [&](int v) {
    while (v != -1) {
      int pv = parent[v];
      int next_v = match[pv];
```

```
      match[v] = pv;
      match[pv] = v;
      v = next_v;
    }
  };  // augment

  auto bfs = [&](int root) {
    fill(label.begin(), label.end(), -1);
    iota(orig.begin(), orig.end(), 0);
    while (!q.empty()) {
      q.pop();
    }
    q.push(root);
    // 初始点设为「o」，这里以「0」代替「o」，「1」代替「i」
    label[root] = 0;
    while (!q.empty()) {
      int v = q.front();
      q.pop();
      for (int id : g.g[v]) {
        auto &e = g.edges[id];
        int u = e.from ^ e.to ^ v;
        if (label[u] == -1) {  // 找到未拜访点
          label[u] = 1;        // 标记「i」
          parent[u] = v;
          if (match[u] == -1) {  // 找到未匹配点
            augment(u);          // 寻找增广路径
            return true;
          }
          // 找到已匹配点将与她匹配的点丢入 queue 延伸交错树
          label[match[u]] = 0;
          q.push(match[u]);
          continue;
        } else if (label[u] == 0 && orig[v] != orig[u]) {
          // 找到已拜访点且标记同为「o」代表找到「花」
          int a = lca(orig[v], orig[u]);
          // 找 LCA 然后缩花
          blossom(u, v, a);
          blossom(v, u, a);
        }
      }
    }
    return false;
  };  // bfs

  auto greedy = [&]() {
    vector<int> order(g.n);
    // 随机打乱 order
    iota(order.begin(), order.end(), 0);
    shuffle(order.begin(), order.end(), rng);

    // 将可以匹配的点匹配
    for (int i : order) {
      if (match[i] == -1) {
        for (auto id : g.g[i]) {
```

```
          auto &e = g.edges[id];
          int to = e.from ^ e.to ^ i;
          if (match[to] == -1) {
            match[i] = to;
            match[to] = i;
            break;
          }
        }
      }
    }
  };  // greedy

  // 一开始先随机匹配
  greedy();
  // 对未匹配点找增广路
  for (int i = 0; i < g.n; i++) {
    if (match[i] == -1) {
      bfs(i);
    }
  }
  return match;
}
```

"UOJ #79. **一般图最大匹配**[1]"

```
#include <bits/stdc++.h>
using namespace std;

// graph
template <typename T>
class graph {
 public:
  struct edge {
    int from;
    int to;
    T cost;
  };

  vector<edge> edges;
  vector<vector<int> > g;
  int n;

  graph(int _n) : n(_n) { g.resize(n); }

  virtual int add(int from, int to, T cost) = 0;
};

// undirectedgraph
template <typename T>
class undirectedgraph : public graph<T> {
 public:
  using graph<T>::edges;
  using graph<T>::g;
```

```
  using graph<T>::n;

  undirectedgraph(int _n) : graph<T>(_n) {}

  int add(int from, int to, T cost = 1) {
    assert(0 <= from && from < n && 0 <= to && to < n);
    int id = (int)edges.size();
    g[from].push_back(id);
    g[to].push_back(id);
    edges.push_back({from, to, cost});
    return id;
  }
};

// blossom / find_max_unweighted_matching
template <typename T>
vector<int> find_max_unweighted_matching(const undirectedgraph<T> &g) {
  std::mt19937 rng(114514);  // 这里随机种子是无关紧要的
  // 也可以用 chrono::steady_clock::now().time_since_epoch().count()
  // 获取当前时间
  vector<int> match(g.n, -1);   // 匹配
  vector<int> aux(g.n, -1);     // 时间戳记
  vector<int> label(g.n);       // "o" or "i"
  vector<int> orig(g.n);        // 花根
  vector<int> parent(g.n, -1);  // 父节点
  queue<int> q;
  int aux_time = -1;

  auto lca = [&](int v, int u) {
    aux_time++;
    while (true) {
      if (v != -1) {
        if (aux[v] == aux_time) {  // 找到拜访过的点也就是 LCA
          return v;
        }
        aux[v] = aux_time;
        if (match[v] == -1) {
          v = -1;
        } else {
          v = orig[parent[match[v]]];  // 以匹配点的父节点继续寻找
        }
      }
      swap(v, u);
    }
  };  // lca

  auto blossom = [&](int v, int u, int a) {
    while (orig[v] != a) {
      parent[v] = u;
      u = match[v];
      if (label[u] == 1) {  // 初始点设为"o" 找增广路
        label[u] = 0;
        q.push(u);
      }
```

```
      orig[v] = orig[u] = a;  // 缩花
      v = parent[u];
    }
  };  // blossom

  auto augment = [&](int v) {
    while (v != -1) {
      int pv = parent[v];
      int next_v = match[pv];
      match[v] = pv;
      match[pv] = v;
      v = next_v;
    }
  };  // augment

  auto bfs = [&](int root) {
    fill(label.begin(), label.end(), -1);
    iota(orig.begin(), orig.end(), 0);
    while (!q.empty()) {
      q.pop();
    }
    q.push(root);
    // 初始点设为 "o", 这里以"0" 代替"o", "1" 代替"i"
    label[root] = 0;
    while (!q.empty()) {
      int v = q.front();
      q.pop();
      for (int id : g.g[v]) {
        auto &e = g.edges[id];
        int u = e.from ^ e.to ^ v;
        if (label[u] == -1) {  // 找到未拜访点
          label[u] = 1;        // 标记 "i"
          parent[u] = v;
          if (match[u] == -1) {  // 找到未匹配点
            augment(u);          // 寻找增广路径
            return true;
          }
          // 找到已匹配点将与她匹配的点丢入 queue 延伸交错树
          label[match[u]] = 0;
          q.push(match[u]);
          continue;
        } else if (label[u] == 0 && orig[v] != orig[u]) {
          // 找到已拜访点且标记同为"o" 代表找到" 花"
          int a = lca(orig[v], orig[u]);
          // 找 LCA 然后缩花
          blossom(u, v, a);
          blossom(v, u, a);
        }
      }
    }
    return false;
  };  // bfs

  auto greedy = [&]() {
```

```
    vector<int> order(g.n);
    // 随机打乱 order
    iota(order.begin(), order.end(), 0);
    shuffle(order.begin(), order.end(), rng);

    // 将可以匹配的点匹配
    for (int i : order) {
      if (match[i] == -1) {
        for (auto id : g.g[i]) {
          auto &e = g.edges[id];
          int to = e.from ^ e.to ^ i;
          if (match[to] == -1) {
            match[i] = to;
            match[to] = i;
            break;
          }
        }
      }
    }
  };  // greedy

  // 一开始先随机匹配
  greedy();
  // 对未匹配点找增广路
  for (int i = 0; i < g.n; i++) {
    if (match[i] == -1) {
      bfs(i);
    }
  }
  return match;
}

int main() {
  ios::sync_with_stdio(false);

  int n, m;
  cin >> n >> m;

  undirectedgraph<int> g(n);

  while (m--) {
    int u, v;
    cin >> u >> v;
    g.add(u - 1, v - 1);  // 0-based
  }

  auto match = find_max_unweighted_matching(g);

  cout << count_if(match.begin(), match.end(), [](int x) { return x != -1; }) /
              2
       << endl;
  for (int i = 0; i < n; i++) cout << match[i] + 1 << " \n"[i == n - 1];

  return 0;
}
```

基于高斯消元的一般图匹配算法

" 提示"

在阅读以下内容前，你可能需要先阅读「线性代数」部分中关于矩阵的内容：

- 矩阵
- 行列式
- 高斯消元

这一部分将介绍一种基于高斯消元的一般图匹配算法。与传统的带花树算法相比，它的优势在于更易于理解与编写，同时便于解决「最大匹配中的必须点」等问题；缺点在于常数比较大，因为高斯消元的 $O(n^3)$ 基本是跑满的，而带花树一般跑不满。

前置知识：Tutte 矩阵

定义：对于一张 n 个点的无向图 $G=(V,E)$，其 Tutte 矩阵 $\tilde{A}(G)$ 为一个 $n\times n$ 的矩阵，其中：

$$\tilde{A}(G)_{i,j}=\begin{cases}x_{i,j}, & i<j,\ (v_i,v_j)\in E\\ -x_{i,j}, & i>j,\ (v_i,v_j)\in E\\ 0, & \text{otherwise}\end{cases}$$

其中 $x_{i,j}$ 是一个变量，因此 $\tilde{A}(G)$ 中共有 $|E|$ 个变量。

在无歧义的情况下，以下将 $\tilde{A}(G)$ 简写为 $\tilde{A}$。

定理（Tutte 定理）：G 存在完美匹配当且仅当 $\det\tilde{A}\neq 0$。

" 证明"

这里引入「偶环覆盖」的概念：一个无向图 G 的偶环覆盖指用若干偶环（包括二元环）不重不漏地覆盖所有的点。

易证 G 存在完美匹配当且仅当 G 存在偶环覆盖。

- 如果 G 存在偶环覆盖，我们只需要在每个环都隔一条取一条边，就可以得到一个完美匹配。
- 如果 G 存在完美匹配，我们只需要将匹配边对应的二元环取出，就可以得到一个偶环覆盖。

然后证明 G 存在偶环覆盖当且仅当 $\tilde{A}\neq 0$。

考虑行列式的定义

$$\det A=\sum_{\pi}(-1)^{\pi}\prod_{i}A_{i,\pi_i}$$

其中 π 是任意排列，$(-1)^{\pi}$ 表示若 π 中的逆序对数为奇数，则取 -1，否则取 1。

不难看出每个排列都可以被看作 G 的一个环覆盖。如果这个环覆盖中存在奇环，则将这个环翻转后的和一定为 0，因此只有偶环覆盖才能使行列式不为 0，证毕。

定理：$\operatorname{rank}\tilde{A}$ 一定为偶数，并且 G 的最大匹配的大小等于 $\operatorname{rank}\tilde{A}$ 的一半。

" 证明"

反对称矩阵的秩只能是偶数；后者请读者自行思考。

实际应用中不可能带着 $|E|$ 个变量进行计算，不过可以取一个数域，例如取某个素数 p 的剩余系 $\mathcal{Z}_p$，将变量分别随机替换为 $\mathcal{Z}_p$ 中的数，再进行计算。方便起见，在无歧义的情况下，以下用 $\tilde{A}$ 直接指代替换后的矩阵。

定理：$\operatorname{rank}\tilde{A}$ 至多为 G 的最大匹配大小的两倍，并且二者相等的概率至少为 $1-\frac{n}{p}$。

考虑到一般图最大匹配中 n 基本不会超过 10^3，实际中 p 取 10^9 数量级的素数就足够了。

由定理可知，如果只需要求最大匹配数，而无需匹配方案，那么只需要用一次高斯消元求出 $\operatorname{rank}\tilde{A}$ 即可，远比带花树简洁。不过如果需要输出方案，会稍微复杂一些，需要用到下面介绍的算法。

构造完美匹配

由 Tutte 定理和上面的定理可知，如果 G 存在完美匹配，那么 $\tilde{A}$ 有很大概率满秩。方便起见，以下叙述中均省略「有很大概率」。

记 G 中标号为 i 的点为 v_i，进一步地我们有如下定理：

定理：$\tilde{A}^{-1}_{j,i} \neq 0 \iff G - \{v_i, v_j\}$ 有完美匹配。

" 逆矩阵与伴随矩阵"

对任意 n 阶方阵 A，定义其伴随矩阵为 $A^*_{i,j} = (-1)^{i+j} M_{j,i}$，其中 $M_{j,i}$ 为删去第 j 行第 i 列的余子式。换言之，设 A 的代数余子式矩阵为 M，则 $A^* = M^T$。

定理：如果 A 可逆，那么 $A^{-1} = \frac{1}{\det A} A^*$。

所以这里的 $A^{-1}_{j,i} \neq 0 \iff M_{i,j} \neq 0$，也就是 A 删去第 i 行第 j 列后的部分满秩。

换言之，如果 $(v_i, v_j) \in E$，并且 $\tilde{A}^{-1}_{j,i} \neq 0$，就表明存在一个完美匹配方案包含 (v_i, v_j) 这条边。以下将这种边称为**可行边**。

由如上定理，对于一个有完美匹配的无向图 G，我们可以得到一个比较显然的暴力算法来寻找一组完美匹配：每次枚举 i, j，如果 (v_i, v_j) 是一条可行边（连边存在，并且 $\tilde{A}^{-1}_{j,i} \neq 0$），就将 (v_i, v_j) 加入匹配方案，并在 G 中都删掉这两个点，再重新计算新的 $\tilde{A}^{-1}$。

总共要做 $\frac{n}{2}$ 轮，每轮都是 $O(n^3)$ 的，总的复杂度是 $O(n^4)$，有点慢了。实际上我们在重新计算 $\tilde{A}^{-1}$ 时，不必每次都重新用高斯消元求逆矩阵，而是可以利用如下定理：

定理（消去定理）：令

$$A = \begin{bmatrix} a_{1,1} & v^T \\ u & B \end{bmatrix} \quad A^{-1} = \begin{bmatrix} \hat{a}_{1,1} & \hat{v}^T \\ \hat{u} & \hat{B} \end{bmatrix}$$

并且 $\hat{a}_{1,1} \neq 0$, 那么就有

$$B^{-1} = \hat{B} - \frac{\hat{u}\hat{v}^T}{\hat{a}_{1,1}}$$

定理中描述的是消去第一行第一列的情况。实际上，它可以非常显然地推广到消去任意一行一列的情况，因此我们只需在算法最开始计算一次 $\tilde{A}^{-1}$，后面每次删除两个点时，只需执行两次 $O(n^2)$ 的消去过程即可。

" 描述有些抽象，可以参考 C++ 代码"

```cpp
void eliminate(int A[][MAXN], int r, int c) {  // 消去第 r 行第 c 列
  row_marked[r] = col_marked[c] = true;        // 已经被消掉

  int inv = quick_power(A[r][c], p - 2);  // 逆元

  for (int i = 1; i <= n; i++)
    if (!row_marked[i] && A[i][c]) {
      int tmp = (long long)A[i][c] * inv % p;

      for (int j = 1; j <= n; j++)
        if (!col_marked[j] && A[r][j])
          A[i][j] = (A[i][j] - (long long)tmp * A[r][j]) % p;
    }
}
```

总共要做 $\frac{n}{2}$ 轮，每轮复杂度为 $O(n^2)$，因此上述算法可以在 $O(n^3)$ 的时间内找到一组完美匹配。

构造最大匹配

我们刚刚已经解决了构造一组完美匹配的问题，但是求解问题时一般需要最大匹配。

前面已经提到，G 的最大匹配大小等于 rank $\tilde{A}$ 的一半。如果我们能找到 $\tilde{A}$ 的一个极大满秩子矩阵，那么对子矩阵对应的导出子图求出一组完美匹配，即可找到 G 的一组完美匹配。

换一个角度考虑，如果 G 有完美匹配，那么 $\tilde{A}$ 满秩，换言之，$\tilde{A}$ 是线性无关的。那么如果 $\tilde{A}$ 不是满秩的，我们可以求出 $\tilde{A}$ 的一组线性基，然后只保留线性基对应的行列，就可以得到 $\tilde{A}$ 的一个极大满秩子矩阵。

求出极大满秩子矩阵之后，再用上面的算法找出导出子图的一组完美匹配，即可得到原图的一组最大匹配。注意由于高斯消元中可能会有行的交换，因此实现时要注意维护好点的编号。

"UOJ #79. **一般图最大匹配**[1]"

```
#include <bits/stdc++.h>
using namespace std;

const int maxn = 505, p = (int)1e9 + 7;

int qpow(int a, int b) {
  int ans = 1;
  while (b) {
    if (b & 1) ans = (long long)ans * a % p;
    a = (long long)a * a % p;
    b >>= 1;
  }
  return ans;
}

int A[maxn][maxn], B[maxn][maxn], t[maxn][maxn], id[maxn];

// 高斯消元 O(n^3)
// 在传入 B 时表示计算逆矩阵，传入 nullptr 则只需计算矩阵的秩
void Gauss(int A[][maxn], int B[][maxn], int n) {
  if (B) {
    memset(B, 0, sizeof(t));
    for (int i = 1; i <= n; i++) B[i][i] = 1;
  }

  for (int i = 1; i <= n; i++) {
    if (!A[i][i]) {
      for (int j = i + 1; j <= n; j++)
        if (A[j][i]) {
          swap(id[i], id[j]);
          for (int k = i; k <= n; k++) swap(A[i][k], A[j][k]);

          if (B)
            for (int k = 1; k <= n; k++) swap(B[i][k], B[j][k]);
          break;
        }

      if (!A[i][i]) continue;
    }

    int inv = qpow(A[i][i], p - 2);
```

```
    for (int j = 1; j <= n; j++)
      if (i != j && A[j][i]) {
        int t = (long long)A[j][i] * inv % p;

        for (int k = i; k <= n; k++)
          if (A[i][k]) A[j][k] = (A[j][k] - (long long)t * A[i][k]) % p;

        if (B) {
          for (int k = 1; k <= n; k++)
            if (B[i][k]) B[j][k] = (B[j][k] - (long long)t * B[i][k]) % p;
        }
      }
  }

  if (B)
    for (int i = 1; i <= n; i++) {
      int inv = qpow(A[i][i], p - 2);

      for (int j = 1; j <= n; j++)
        if (B[i][j]) B[i][j] = (long long)B[i][j] * inv % p;
    }
}

bool row_marked[maxn] = {false}, col_marked[maxn] = {false};

int sub_n;  // 极大满秩子矩阵的大小

// 消去一行一列 O(n^2)
void eliminate(int r, int c) {
  row_marked[r] = col_marked[c] = true;  // 已经被消掉

  int inv = qpow(B[r][c], p - 2);

  for (int i = 1; i <= sub_n; i++)
    if (!row_marked[i] && B[i][c]) {
      int t = (long long)B[i][c] * inv % p;

      for (int j = 1; j <= sub_n; j++)
        if (!col_marked[j] && B[r][j])
          B[i][j] = (B[i][j] - (long long)t * B[r][j]) % p;
    }
}

int vertices[maxn], girl[maxn];  // girl 是匹配点，用来输出方案

int main() {
  auto rng = mt19937(chrono::steady_clock::now().time_since_epoch().count());

  int n, m;
  scanf("%d%d", &n, &m);  // 点数和边数

  while (m--) {
    int x, y;
    scanf("%d%d", &x, &y);
```

```
    A[x][y] = rng() % p;
    A[y][x] = -A[x][y];  // Tutte 矩阵
  }

  for (int i = 1; i <= n; i++)
    id[i] = i;  // 输出方案用的，因为高斯消元的时候会交换列
  memcpy(t, A, sizeof(t));

  Gauss(A, nullptr, n);

  for (int i = 1; i <= n; i++)
    if (A[id[i]][id[i]]) vertices[++sub_n] = i;  // 找出一个极大满秩子矩阵

  for (int i = 1; i <= sub_n; i++)
    for (int j = 1; j <= sub_n; j++) A[i][j] = t[vertices[i]][vertices[j]];

  Gauss(A, B, sub_n);

  for (int i = 1; i <= sub_n; i++)
    if (!girl[vertices[i]])
      for (int j = i + 1; j <= sub_n; j++)
        if (!girl[vertices[j]] && t[vertices[i]][vertices[j]] && B[j][i]) {
          // 注意上面那句 if 的写法，现在 t 是邻接矩阵的备份，
          // 逆矩阵 j 行 i 列不为 0 当且仅当这条边可行
          girl[vertices[i]] = vertices[j];
          girl[vertices[j]] = vertices[i];

          eliminate(i, j);
          eliminate(j, i);
          break;
        }

  printf("%d\n", sub_n / 2);
  for (int i = 1; i <= n; i++) printf("%d ", girl[i]);

  return 0;
}
```

习题

- UOJ #79. 一般图最大匹配[1]
- UOJ#171.【WC2016】 挑战 NPC[2]

参考资料

1. Mucha M, Sankowski P.Maximum matchings via Gaussian elimination[3]
2. 周子鑫，杨家齐《基于线性代数的一般图匹配》
3. ZYQN 《 基于线性代数的一般图匹配算法 》[4]

参考资料与注释

[1] UOJ #79. 一般图最大匹配 [1-1] [1-2] [1-3]

[2] UOJ#171.【WC2016】挑战 NPC

[3] Maximum matchings via Gaussian elimination

[4] 《基于线性代数的一般图匹配算法》

11.30.6 一般图最大权匹配

Authors: accelsao, Henry-ZHR, yuhuoji

本页从一般图最大权完美匹配到一般图最大权匹配（最大权匹配可以通过增加零边变成最大权完美匹配）。

预备知识

花（blossom）

一般图匹配和二分图匹配不同的是，图可能存在奇环。可以将偶环视为二分图。

带花树算法（Blossom Algorithm）的处理方式时是遇到奇环就把它缩成一个**花（Blossom）**，并把花中所有的点设为偶点。既然花上的点都可以成为偶点，那么可以把整个花直接缩成一个偶点。注意，一个花可以包含其它花。

这也可以变成线性规划和对偶问题，但是要对花进行一些处理。

顶标（vertex labeling）和等边（Equality Edge）

定义 z_u 是点 u 的顶标（vertex labeling），与 KM 算法中定义的顶标含义相同。定义边 $e(u,v)$ 为”等边”当且仅当点 u 和点 v 的标号和等于边 e 的权值（$z_u+z_v=w(e)$），此时边的标号 $z_e=z_u+z_v-w(e)=0$。

一般图最大权完美匹配的线性规划

定义

因为一朵花最少有三个点，缩花后成为一个点。设 O 为大小为 ≥ 3 奇数的集合的集合（包含所有花），$\gamma(S)$ 表示 S 集合中的边。

$$
\begin{aligned}
&\text{设} S \subseteq V \\
&\gamma(S)=\{(u,v)\in E : u\in S, v\in S\} \\
&O=\{B\subseteq V : |B|\text{是奇数且}|B|\geq 3\}
\end{aligned}
$$

对偶问题

” 原问题”

$$
\begin{aligned}
&\max \sum_{e\in E} w(e)x_e \\
&\text{限制:} \\
&x(\delta(u))=1 : \forall u\in V \\
&x(\gamma(B))\leq \lfloor\frac{|B|}{2}\rfloor : \forall B\in O \\
&x_e\geq 0 : \forall e\in E
\end{aligned}
$$

然后通过原始对偶（Primal-Dual）将问题转换为对偶问题。

" 对偶问题"

$$\min \sum_{u\in V} z_u + \sum_{B\in O} \left\lfloor \frac{|B|}{2} \right\rfloor z_B$$

限制：

$z_B \geq 0 : \forall B \in O$

$z_e \geq 0 : \forall e \in E$

设$e = (u, v)$ 这里

$$z_e \quad = \quad z_u + z_v - w(e) + \sum_{\substack{B\in O \\ u,v\in\gamma(B)}} z_B$$

$x_e = 1$ 的边是匹配边，$x_e = 0$ 的边是非匹配边。和二分图一样，我们必须满足 $x_e \in \{0, 1\} : \forall e \in E$。因此必须在最大权完美匹配的时候，让所有匹配边都是**等边**的。

和二分图不同的是，一般图多了 z_B 要处理。下面考虑 z_B 什么时候大于 0。

可以看出，尽量使 $z_B = 0$ 是最好的做法，但在不得已时还是要让 $z_B > 0$。在 $x(\gamma(B)) = \left\lfloor \frac{|B|}{2} \right\rfloor$ 且$x(\delta(B)) = 1$ 时，让 $z_B > 0$ 即可。因为除了在这种情况下，$z_B > 0$ 是无意义的。

根据互补松弛条件，有以下的对应关系：

- 对于选中的边 e，必有 $z_e = 0$。

$$x_e > 0 \longrightarrow z_e = 0, \quad \forall e \in E$$

- 对于选中的集合 B，$z_B > 0 \longrightarrow x(\gamma(B)) = \left\lfloor \frac{|B|}{2} \right\rfloor$，即所有 $z_B > 0$ 的集合 B，都被选了集合大小一半的边，也即集合 B 是一朵花，选中花中的一条边进行增广。同时，我们加入一个条件：$x(\delta(B)) = 1$，即只有花 B 向外连了一条边的时候，$z_B > 0$ 才是有意义的。

$$z_B > 0 \longrightarrow x(\gamma(B)) = \left\lfloor \frac{|B|}{2} \right\rfloor, x(\delta(B)) = 1 \quad \forall B \in O$$

以「**等边**」的概念，结合之前的带花树算法：用「等边」构成的增广路不断进行扩充，由于用来扩充的边全是「等边」，最后得到的最大权完美匹配仍然全是「等边」。

处理花的问题

当遇到花的时候，要将它缩成一个偶点。将花中所有点都设为偶点，并让它的 $z_B = 0$。

由于缩花后会把花保存起来，直到满足某些条件才会拆开，所以不能用之前的方法记录花。

如果没有特殊说明，之前提到的点，都包含缩花形成的偶点。

由于花也有可能缩成点被加入队列中，并且花的数量是不固定的，因此不能像之前一样枚举每个点来检查是否有增广路。因此，在进行广度优先搜索（BFS）时，必须将所有未匹配的点都放入队列中。

这样会同时产生很多棵交错树。

算法的四个步骤

这个算法可以分成四个步骤。

1. GROW（等边）：用" 等边" 构成交错树。
2. AUGMENT（增广）：找出增广路并扩充匹配。
3. SHRINK（缩花）：把花缩成一个点。
4. EXPAND（展开）：把花拆开。

在 AUGMENT 阶段时，因为所有未匹配点都会在不同的交错树上，所以当增广时两棵交错树的偶点连在一起，就表示找到了一条增广路。

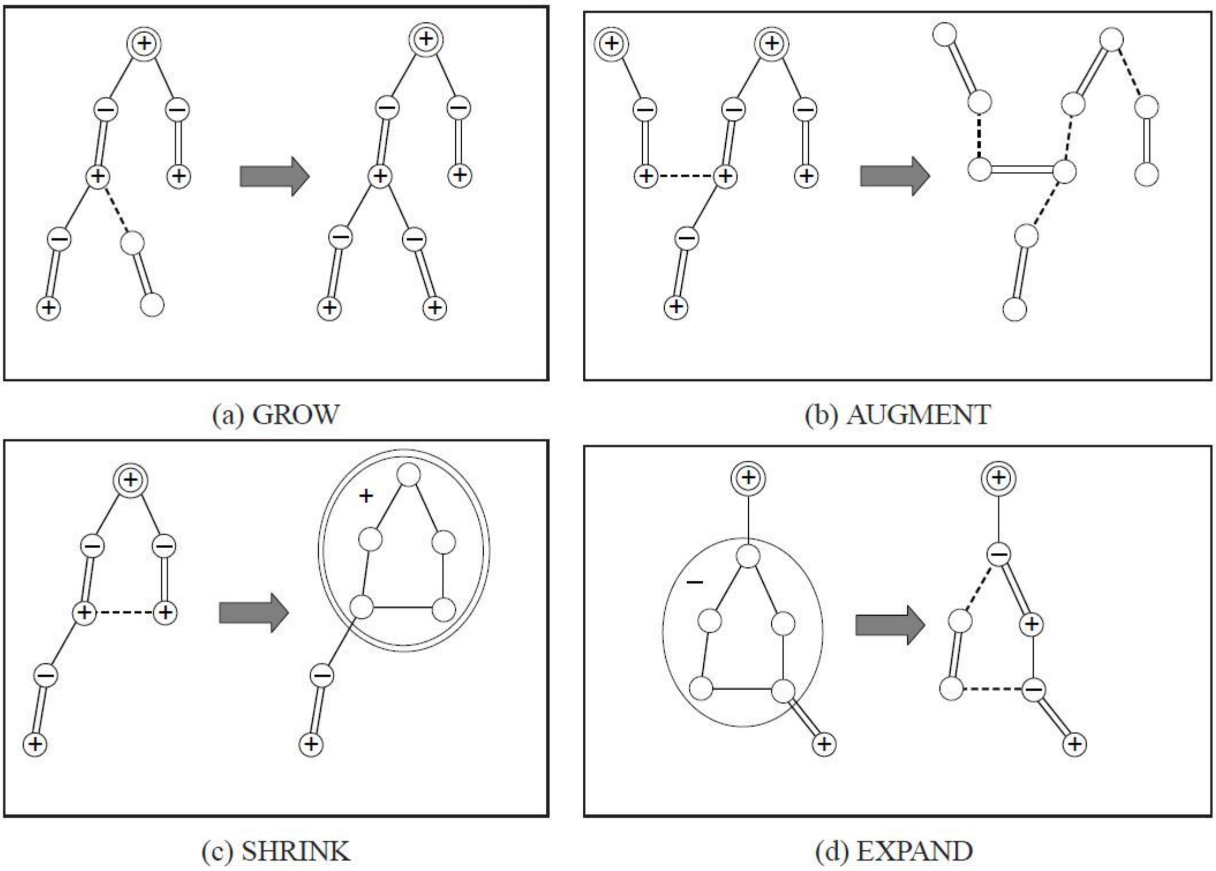


图 11.107 general-weight-match-1

找不到等边扩充

和二分图一样，也会有找不到「等边」扩充的问题。这时就需要调整 vertex labeling。

调整 VERTEX LABELING

vertex labeling 仍要维持大于等于的性质，而且既有的「等边」不能被改变，还要让 z_B 尽量的小。

" 定义符号奇偶点"

以 u^- 来表示 u 在交错树上为奇点。

以 u^+ 来表示 u 在交错树上为偶点。

以 $u^\emptyset$ 来表示 u 不在任何一棵交错树上。

之后所有提到的 B 预设都是花，并同时代表缩花之后的点。

花也可以有奇花偶花之分，因此也适用 B^+、B^-、$B^\emptyset$ 等符号。

设目前有 r 棵交错树 $T_i = (U_{t_i}, V_{t_i}) : 1 \le i \le r$，令

$$
\begin{aligned}
d1 &= \min(\{z_e : e = (u^+, v^\emptyset)\}) \\
d2 &= \min(\{z_e : e = (u^+, v^+),\ u^+ \in T_i,\ v^+ \in T_j,\ i \neq j\})/2 \\
d3 &= \min(\{z_{B^-} : B^- \in O\})/2
\end{aligned}
$$

注意这里 B 是缩花之后的点，所以可以有奇偶性。

设 $d = min(d1, d2, d3)$，让

$$z_{u^+} -= d$$
$$z_{v^-} += d$$
$$z_{B^+} += 2d$$
$$z_{B^-} -= 2d$$

如果出现 $z_B = 0 (d = d3)$，为了防止 $z_B < 0$ 的情况，所以要把这朵花拆了 (EXPAND)。拆花后只留下花里的交替路径，并把花里不在交替路径上的点设为未走访 ($\emptyset$)。

如此便制造了一条（以上）的等边，既有等边保持不动，并维持了 $z_e \geq 0 : \forall e \in E$ 的性质，且最低限度增加了 z_B，可以继续找增广路了。

一般图最大权匹配

以上求的是最大权完美匹配，求最大权匹配需要在 vertex labeling 额外增加一个限制：对于所有匹配点 u，$z_u > 0$。

开始时先设所有的 $z_u = max(\{w(e) : e \in E\})/2$。

vertex labeling 为 0 的点最后将成为未匹配点。

参考代码

这里为了方便实现，使用边权乘 2 来计算 z_e 的值，这样就不会出现浮点数误差了。

" 存储"

```
#define INF INT_MAX
#define MAXN 400

struct edge {
  int u, v, w;

  // 表示 (u,v) 为一条边其权重为 w
  edge() {}

  edge(int u, int v, int w) : u(u), v(v), w(w) {}
};

int n, n_x;
// 有 n 个点，编号为 1 ~ n
// n_x 表示当前点加上花的数量，编号从 n+1 到 n_x 为花的节点
edge g[MAXN * 2 + 1][MAXN * 2 + 1];
// 图用邻接矩阵存储，因为最多有 n-1 朵花，所以大小为 MAXN*
vector<int> flower[MAXN * 2 + 1];
// flower[b] 记录了花 b 中有哪些点
// 我们记录花中的点的方式是只记录花里面的最外层花
```

下面是嵌套花的例子。

其中 $\{6, 5, 8\} \in b1, \{b1, 4, 3, 2, 11, 10, 9\} \in b2$。存储为：

```
flower[b2] = {b1, 4, 3, 2, 11, 10, 9}
flower[b1] = {6, 5, 8}
```

```
flower[b2] = {9, b1, 4, 3, 2, 11, 10}
flower[b1] = {5, 8, 6}
```

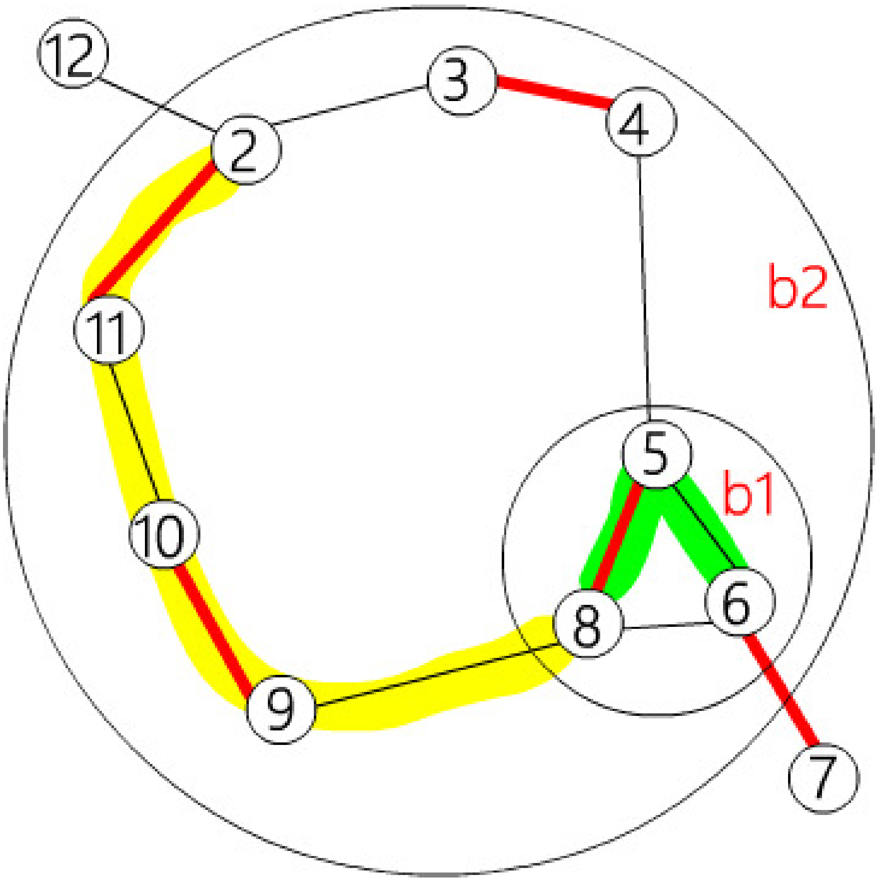


图 11.108 general-weight-match-2

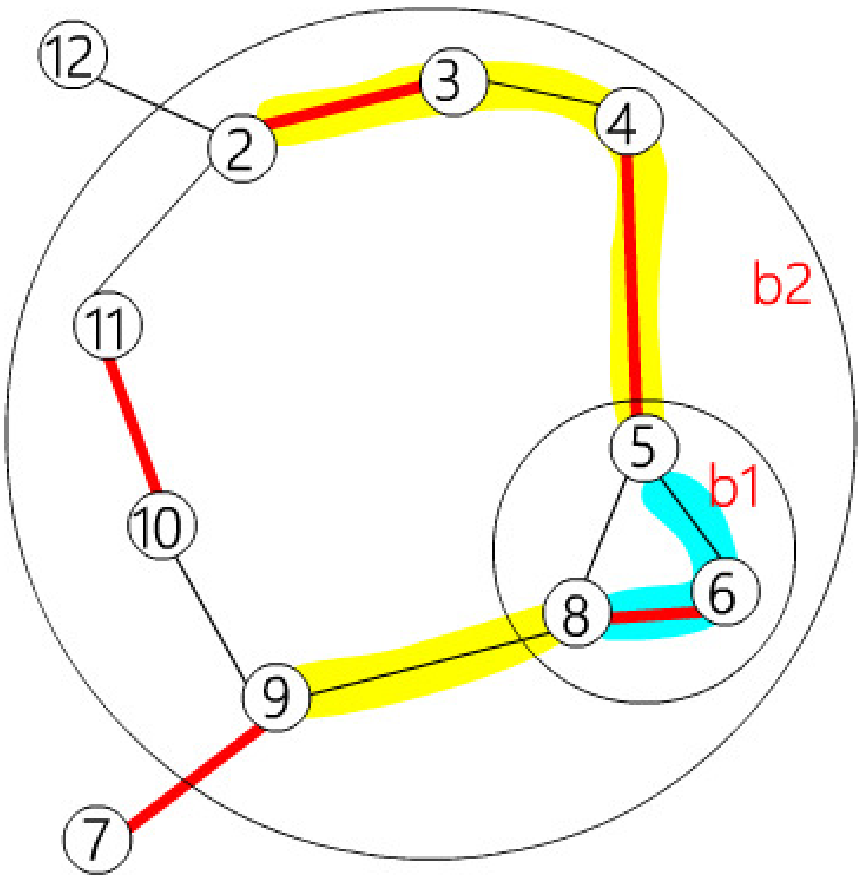


图 11.109 general-weight-match-3

```
int lab[MAXN * 2 + 1];
// lab[u] 用来记录 z_u, lab[b] 用来记录 z_B
int match[MAXN * 2 + 1], slack[MAXN * 2 + 1], st[MAXN * 2 + 1],
    pa[MAXN * 2 + 1];
// match[x]=y 表示 (x,y) 是匹配，这里 x、y 可能是花
// slack[x]=u 表示 z(x,u) 是所有和 x 相邻的边中最小的那条边
// 表示节点 x 所在的花是 b。如果 x=b 且 b<=n，则表示 x
// 是一个普通节点（不属于任何花）表示在交错树中，节点 v 的父节点是 u
int flower_from[MAXN * 2 + 1][MAXN + 1], S[MAXN * 2 + 1], vis[MAXN * 2 + 1];
/*
flower_from[b][x]=xs 表示最大的包含 x 的 b 的子花是 xs
x 是 b 里面的一个点，xs 是 b 里面的一朵花或一个点，同时 x=xs 或 x 是 xs 的其中一个点
*/
// S[u]={-1: 没走过 0: 偶点 1: 奇点}
// vis 只用在找 lca 的时候检查是不是走过了
queue<int> q;
// BFS 找增广路用的 queue
```

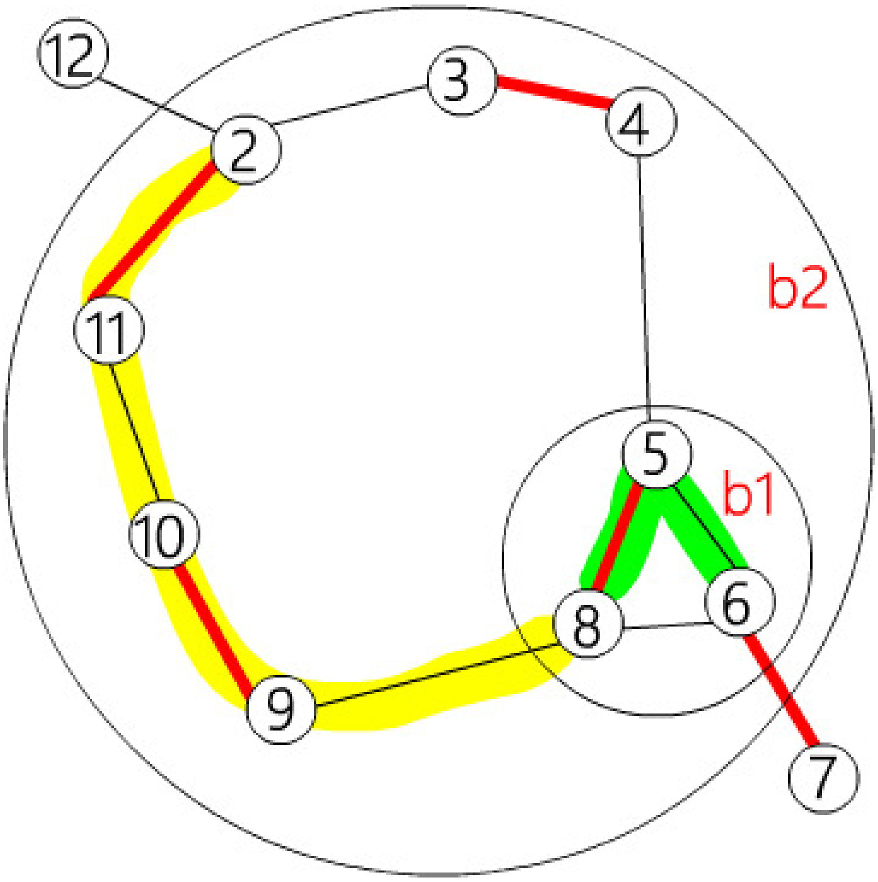


图 11.110 general-weight-match-4

```
flower_from[b2][6] = b1
flower_from[b2][5] = b1
flower_from[b2][9] = 9
flower_from[b1][6] = 6
以此类推
```

```cpp
inline int e_delta(const edge &e) {
  // 计算 ze，为了方便起见先把所有边的权重乘二
  // 在花里面直接计算 e_delta 值会导致错误
  return lab[e.u] + lab[e.v] - g[e.u][e.v].w * 2;
}

inline void update_slack(int u, int x) {
  // 以 u 更新 slack[x] 的值
  if (!slack[x] || e_delta(g[u][x]) < e_delta(g[slack[x]][x])) {
    slack[x] = u;
  }
}

inline void set_slack(int x) {
  // 算出 slack[x] 的值，slack[x]=0 表示 x 是交错树中的节点
  slack[x] = 0;
  for (int u = 1; u <= n; ++u) {
    if (g[u][x].w > 0 && st[u] != x && S[st[u]] == 0) {
      update_slack(u, x);
    }
  }
}
```

```cpp
void q_push(int x) {
  // 把 x F到 queue 里面，我们设定 queue 不能直接 push 一朵花
  if (x <= n)
    q.push(x);
  else {
    // 若要 push 花必须将花里面原图的点都添加到 queue 中
    for (size_t i = 0; i < flower[x].size(); i++) {
      q_push(flower[x][i]);
    }
  }
}

inline void set_st(int x, int b) {
  // 将 x 所在的花设为 b
  st[x] = b;
  if (x > n) {
    // 若 x 也是花的话，就必须要把 x 里面的点其所在的花也设为 b
    for (size_t i = 0; i < flower[x].size(); ++i) {
      set_st(flower[x][i], b);
    }
  }
}
```

```cpp
inline int get_pr(int b, int xr) {
  // xr 是 flower[b] 中的一个点，返回值 pr 是它的位置
  // 为了方便程序运行，我们让 flower[b][0]~flower[b][pr] 为花里的交替路
  int pr = find(flower[b].begin(), flower[b].end(), xr) - flower[b].begin();
  if (pr % 2 == 1) {
    // 检查他在花里的位置，如果 flower[b][0]~flower[b][pr] 不是交替路
    // 就把整朵花反转，重新计算 pr
    // 让 flower[b][0]~flower[b][pr] 为花里的交替路
```

```
    reverse(flower[b].begin() + 1, flower[b].end());
    return (int)flower[b].size() - pr;
  } else
    return pr;
}
```

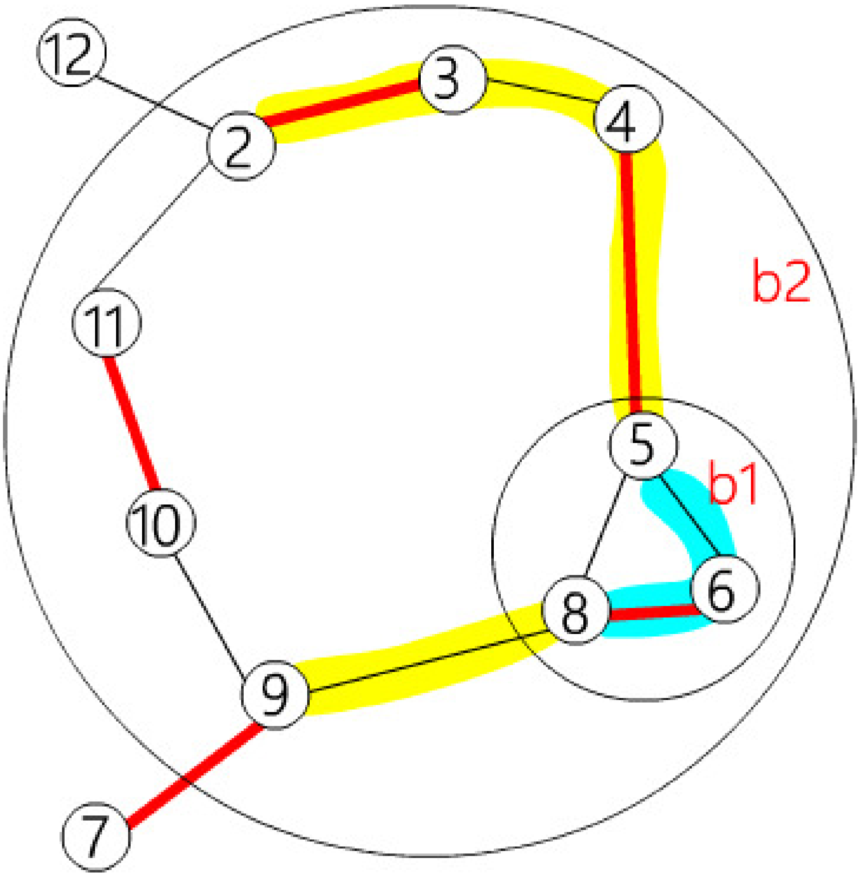


图 11.111 general-weight-match-5

如果使用 get_pr(b2,11)，flower[b2] 会变成 {9,10,11,2,3,4,b1}，并返回 2。

如果使用 get_pr(b2,2)，flower[b2] 会变成 {9,b1,4,3,2,11,10}，并返回 4。

```
inline void set_match(int u, int v) {
  // 设置 u 和 v 为匹配边，u 和 v 有可能是花
  match[u] = g[u][v].v;
  if (u > n) {
    // 如果 u 是花的话
    edge e = g[u][v];
    int xr = flower_from[u][e.u];  // 找出 e.u 在 flower[u] 里的哪朵花上
    int pr = get_pr(u, xr);  // 找出 xr 的位置并让 0~pr 为花里的交替路径
    for (int i = 0; i < pr; ++i) {  // 把花里的交替路上的匹配边和非匹配边反转
      set_match(flower[u][i], flower[u][i ^ 1]);
    }
    set_match(xr, v);  // 设置 (xr,v) 为匹配边
    rotate(flower[u].begin(), flower[u].begin() + pr, flower[u].end());
    // 最后把 pr 设为花托，因为花的存法是 flower[u][0] 会是 u 的花托
    // 所以要把 flower[u][pr] rotate 到最前面
  }
}
```

```cpp
inline void augment(int u, int v) {
  // 把 u 和 u 的祖先全部增广，并设 (u,v) 为匹配边
  for (;;) {
    int xnv = st[match[u]];
    set_match(u, v);
    if (!xnv) return;
    set_match(xnv, st[pa[xnv]]);
    u = st[pa[xnv]];
    v = xnv;
  }
}

inline int get_lca(int u, int v) {
  // 找出 u,v 在交错树上的 lca
  static int t = 0;
  for (++t; u || v; swap(u, v)) {
    if (u == 0) continue;
    if (vis[u] == t) return u;
    vis[u] = t;  // 这种方法可以不用清空 vis 数组
    u = st[match[u]];
    if (u) u = st[pa[u]];
  }
  return 0;
}
```

" 增加一朵奇花"

```cpp
inline void add_blossom(int u, int lca, int v) {
  // 将 u,v,lca 这朵花缩成一个点 b
  // 交错树上 u,v 的 lca 即为花托
  int b = n + 1;
  while (b <= n_x && st[b]) ++b;
  if (b > n_x) ++n_x;
  // 找出目前未使用的花的编号
  lab[b] = 0;              // 设置 zB=0
  S[b] = 0;                // 整朵花为一个偶点
  match[b] = match[lca];  // 设置花的匹配边为花托的匹配边
  flower[b].clear();
  flower[b].push_back(lca);
  for (int x = u, y; x != lca; x = st[pa[y]]) {
    flower[b].push_back(x);
    y = st[match[x]];
    flower[b].push_back(y);
    q_push(y);
  }
  reverse(flower[b].begin() + 1, flower[b].end());
  for (int x = v, y; x != lca; x = st[pa[y]]) {
    flower[b].push_back(x);
    y = st[match[x]];
    flower[b].push_back(y);
    q_push(y);
  }
  // b 中所有点以环形的方式加入 flower[b]，并设花托为首个元素
  set_st(b, b);  // 把整朵花里所有的元素其所在的花设为 b
```

```
  for (int x = 1; x <= n_x; ++x) {
    g[b][x].w = 0;
    g[x][b].w = 0;
  }
  for (int x = 1; x <= n; ++x) {
    flower_from[b][x] = 0;
  }
  for (size_t i = 0; i < flower[b].size(); ++i) {
    int xs = flower[b][i];
    for (int x = 1; x <= n_x; ++x) {
      // 设置 b 和 x 相邻的边为 b 里面和 x 相邻的边 e_delta 最小的那条
      if (g[b][x].w == 0 || e_delta(g[xs][x]) < e_delta(g[b][x])) {
        g[b][x] = g[xs][x];
        g[x][b] = g[x][xs];
      }
    }
    for (int x = 1; x <= n; ++x) {
      if (flower_from[xs][x]) {
        // 如果 b 里面的点 xs 有包含 x
        // 那 flower_from[b][x] 就会是 xs
        flower_from[b][x] = xs;
      }
    }
  }
  set_slack(b);
  // 最后必须要设置 b 的 slack 值
}
```

" 拆花"

```
inline void expand_blossom(int b) {
  // b 是奇花且 zB=0 时，必须要把 b 拆开
  // 因为只拆开 b 而已，所以如果 b 里面有包含其他的花
  // 不需要把他们拆开
  for (size_t i = 0; i < flower[b].size(); ++i) {
    set_st(flower[b][i], flower[b][i]);
    // 先把 flower[b] 里每个元素所在的花设为自己
  }
  int xr = flower_from[b][g[b][pa[b]].u];
  // xr 表示交错路上 b 的父母节点在 flower[b] 里的哪朵花上
  int pr = get_pr(b, xr);  // 找出 xr 的位置并让 0~pr 为花里的交替路径
  for (int i = 0; i < pr; i += 2) {
    // 把交替路径拆开到交错树中
    // 并把交替路中的偶点丢到 queue 里
    int xs = flower[b][i];
    int xns = flower[b][i + 1];
    pa[xs] = g[xns][xs].u;
    S[xs] = 1;
    S[xns] = 0;
    slack[xs] = 0;
    set_slack(xns);
    q_push(xns);
  }
```

```
    S[xr] = 1;  // 这时 xr 会是奇点或奇花
    pa[xr] = pa[b];
    for (size_t i = pr + 1; i < flower[b].size(); ++i) {
      // 把花中所有不再交替路径上的点设为未走访
      int xs = flower[b][i];
      S[xs] = -1;
      set_slack(xs);
    }
    st[b] = 0;
  }
```

" 尝试增广一条等边"

```
inline bool on_found_edge(const edge &e) {
  // BFS 时找到一条等边 e
  // 要对它进行以下的处理
  // 这里 u 一定是偶点
  int u = st[e.u], v = st[e.v];
  if (S[v] == -1) {
    // v 是未走访节点
    pa[v] = e.u;
    S[v] = 1;
    int nu = st[match[v]];
    slack[v] = 0;
    slack[nu] = 0;
    S[nu] = 0;
    q_push(nu);
  } else if (S[v] == 0) {
    // v 是偶点
    int lca = get_lca(u, v);
    if (!lca) {  // lca=0 表示 u,v 在不同的交错树上，有增广路
      augment(u, v);
      augment(v, u);
      return true;  // 找到增广路
    } else
      add_blossom(u, lca, v);
    // 否则 u,v 在同棵树上就会是一朵花，要缩花
  }
  return false;
}
```

" 增广"

```
inline bool matching() {
  memset(S + 1, -1, sizeof(int) * n_x);
  memset(slack + 1, 0, sizeof(int) * n_x);
  q = queue<int>();  // 把 queue 清空
  for (int x = 1; x <= n_x; ++x) {
    if (st[x] == x && !match[x]) {
      // 把所有非匹配点加入 queue 里面，并设为偶点
      pa[x] = 0;
      S[x] = 0;
      q_push(x);
```

```
    }
  }
  if (q.empty()) return false;  // 所有点都有匹配了
  for (;;) {
    while (q.size()) {
      // BFS
      int u = q.front();
      q.pop();
      if (S[st[u]] == 1) continue;
      for (int v = 1; v <= n; ++v) {
        if (g[u][v].w > 0 && st[u] != st[v]) {
          if (e_delta(g[u][v]) == 0) {
            if (on_found_edge(g[u][v])) return true;
          } else
            update_slack(u, st[v]);
        }
      }
    }
    // 修改 lab 值
    int d = INF;
    for (int u = 1; u <= n; ++u) {
      // 这是为了防止出现 lab<0 的情况发生
      // 只要有任何一个 lab[u]=0 就结束程序
      if (S[st[u]] == 0) d = min(d, lab[u]);
    }
    for (int b = n + 1; b <= n_x; ++b) {
      if (st[b] == b && S[b] == 1) d = min(d, lab[b] / 2);
    }
    for (int x = 1; x <= n_x; ++x)
      if (st[x] == x && slack[x]) {
        if (S[x] == -1)
          d = min(d, e_delta(g[slack[x]][x]));
        else if (S[x] == 0)
          d = min(d, e_delta(g[slack[x]][x]) / 2);
      }
    for (int u = 1; u <= n; ++u) {
      if (S[st[u]] == 0) {
        if (lab[u] == d) return 0;
        // 如果 lab[u]=0 就直接结束程序
        lab[u] -= d;
      } else if (S[st[u]] == 1)
        lab[u] += d;
    }
    for (int b = n + 1; b <= n_x; ++b) {
      if (st[b] == b) {
        if (S[st[b]] == 0)
          lab[b] += d * 2;
        else if (S[st[b]] == 1)
          lab[b] -= d * 2;
      }
    }
    q = queue<int>();  // 把 queue 清空
    for (int x = 1; x <= n_x; ++x) {
      // 检查看看有没有增广路径产生
```

```
      if (st[x] == x && slack[x] && st[slack[x]] != x &&
          e_delta(g[slack[x]][x]) == 0)
        if (on_found_edge(g[slack[x]][x])) return true;
    }
    for (int b = n + 1; b <= n_x; ++b) {
      // EXPAND 的操作，把所有 lab[b]=0 的奇花拆开
      if (st[b] == b && S[b] == 1 && lab[b] == 0) expand_blossom(b);
    }
  }
  return false;
}
```

" 主函数"

```
inline pair<long long, int> weight_blossom() {
  // 主函数，一开始先初始化
  memset(match + 1, 0, sizeof(int) * n);
  n_x = n;  // 一开始没有花
  int n_matches = 0;
  long long tot_weight = 0;
  for (int u = 0; u <= n; ++u) {
    // 先把自己所在的花设为自己
    st[u] = u;
    flower[u].clear();
  }
  int w_max = 0;
  for (int u = 1; u <= n; ++u)
    for (int v = 1; v <= n; ++v) {
      // u 是一个点时，里面所包含的点只有自己
      flower_from[u][v] = (u == v ? u : 0);
      w_max = max(w_max, g[u][v].w);
      // 找出最大的边权
    }
  for (int u = 1; u <= n; ++u) lab[u] = w_max;
  // 让所有的 lab= 最大的边权
  // 因为这里实现是用边权乘二来计算 ze 的值所以不用除以二
  while (matching()) ++n_matches;
  for (int u = 1; u <= n; ++u)
    if (match[u] && match[u] < u) tot_weight += g[u][match[u]].w;
  return make_pair(tot_weight, n_matches);
}
```

" 初始化"

很重要使用前一定要初始化

```
inline void init_weight_graph() {
  // 在把边输入到图里面前必须要初始化
  // 因为是最大权匹配所以把不存在的边设为 0
  for (int u = 1; u <= n; ++u)
    for (int v = 1; v <= n; ++v) g[u][v] = edge(u, v, 0);
}
```

复杂度分析

每朵花在一次 BFS 中只会被缩花或拆花一次。每次缩花或拆花的时间复杂度为 $O(|V|)$。最多总共有 $O(|V|)$ 朵花，所以花的处理花费 $O(|V|^2)$ 的时间。而 BFS 花费 $O(|V|+|E|)$ 的时间复杂度。因此，找增广路花费 $O(|V|+|E|)+O(|V|^2)=O(|V|^2)$ 的时间复杂度。

最多做 $|V|$ 次 BFS。所以，总时间复杂度为 $O(|V|^3)$。

习题

- UOJ #81. 一般图最大权匹配[1]

参考资料

1. Kolmogorov, Vladimir (2009), "Blossom V: A new implementation of a minimum cost perfect matching algorithm"[2]
2. 从匈牙利算法到带权带花树 —— 详解对偶问题在图匹配上的应用[3]

参考资料与注释

[1] UOJ #81. 一般图最大权匹配

[2] Kolmogorov, Vladimir (2009), "Blossom V: A new implementation of a minimum cost perfect matching algorithm"

[3] 从匈牙利算法到带权带花树——详解对偶问题在图匹配上的应用

11.31 Prüfer 序列

> note
>
> 本文翻译自 e-maxx Prüfer Code[1]。另外解释一下，原文的结点是从 0 开始标号的，本文按照大多数人的习惯改成了从 1 标号。

这篇文章介绍 Prüfer 序列 (Prüfer code)，这是一种将带标号的树用一个唯一的整数序列表示的方法。

使用 Prüfer 序列可以证明凯莱公式 (Cayley's formula)。并且我们也会讲解如何计算在一个图中加边使图连通的方案数。

注意：我们不考虑含有 1 个结点的树。

Prüfer 序列

引入

Prüfer 序列可以将一个带标号 n 个结点的树用 $[1,n]$ 中的 $n-2$ 个整数表示。你也可以把它理解为完全图的生成树与数列之间的双射。常用组合计数问题中。

Heinz Prüfer 于 1918 年发明这个序列来证明凯莱公式。

对树建立 Prüfer 序列

Prüfer 是这样建立的：每次选择一个编号最小的叶结点并删掉它，然后在序列中记录下它连接到的那个结点。重复 $n-2$ 次后就只剩下两个结点，算法结束。

显然使用堆可以做到 $O(n\log n)$ 的复杂度

" 实现"

```cpp
// 代码摘自原文，结点是从 0 标号的
vector<vector<int>> adj;

vector<int> pruefer_code() {
  int n = adj.size();
  set<int> leafs;
  vector<int> degree(n);
  vector<bool> killed(n, false);
  for (int i = 0; i < n; i++) {
    degree[i] = adj[i].size();
    if (degree[i] == 1) leafs.insert(i);
  }

  vector<int> code(n - 2);
  for (int i = 0; i < n - 2; i++) {
    int leaf = *leafs.begin();
    leafs.erase(leafs.begin());
    killed[leaf] = true;
    int v;
    for (int u : adj[leaf])
      if (!killed[u]) v = u;
    code[i] = v;
    if (--degree[v] == 1) leafs.insert(v);
  }
  return code;
}
```

```python
# 结点是从 0 标号的
adj = [[]]

def pruefer_code():
    n = len(adj)
    leafs = set()
    degree = [0] * n
    killed = [False] * n
    for i in range(1, n):
        degree[i] = len(adj[i])
        if degree[i] == 1:
            leafs.intersection(i)
    code = [0] * (n - 2)
    for i in range(1, n - 2):
        leaf = leafs[0]
        leafs.pop()
        killed[leaf] = True
        for u in adj[leaf]:
            if killed[u] == False:
```

```
            v = u
        code[i] = v
        if degree[v] == 1:
            degree[v] = degree[v] - 1
            leafs.intersection(v)
    return code
```

例如，这是一棵 7 个结点的树的 Prüfer 序列构建过程：

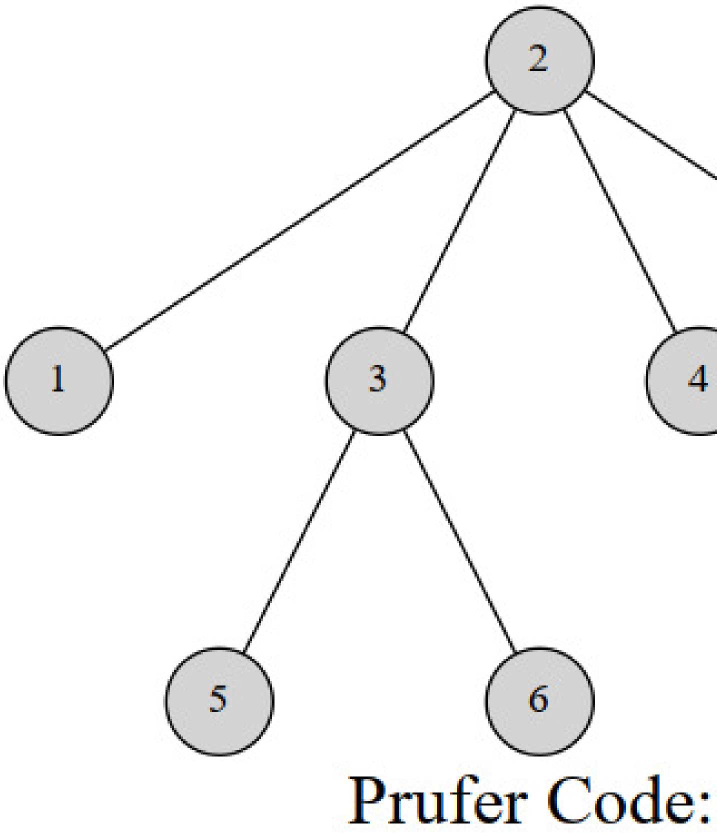


图 11.112 Prüfer

最终的序列就是 $2, 2, 3, 3, 2$。

当然，也有一个线性的构造算法。

Prüfer 序列的线性构造算法

线性构造的本质就是维护一个指针指向我们将要删除的结点。首先发现，叶结点数是非严格单调递减的，删去一个叶结点，叶结点总数要么不变要么减 1。

于是我们考虑这样一个过程：维护一个指针 p。初始时 p 指向编号最小的叶结点。同时我们维护每个结点的度数，方便我们知道在删除结点的时侯是否产生新的叶结点。操作如下：

1. 删除 p 指向的结点，并检查是否产生新的叶结点。
2. 如果产生新的叶结点，假设编号为 x，我们比较 p, x 的大小关系。如果 $x > p$，那么不做其他操作；否则就立刻删除 x，然后检查删除 x 后是否产生新的叶结点，重复 2 步骤，直到未产生新节点或者新节点的编号 $> p$。
3. 让指针 p 自增直到遇到一个未被删除叶结点为止；

正确性

循环上述操作 $n-2$ 次，就完成了序列的构造。接下来考虑算法的正确性。

p 是当前编号最小的叶结点，若删除 p 后未产生叶结点，我们就只能去寻找下一个叶结点；若产生了叶结点 x：

- 如果 $x > p$，则反正 p 往后扫描都会扫到它，于是不做操作；
- 如果 $x < p$，因为 p 原本就是编号最小的，而 x 比 p 还小，所以 x 就是当前编号最小的叶结点，优先删除。删除 x 继续这样的考虑直到没有更小的叶结点。

算法复杂度分析，发现每条边最多被访问一次（在删度数的时侯），而指针最多遍历每个结点一次，因此复杂度是 $O(n)$ 的。

实现

```cpp
// 从原文摘的代码，同样以 0 为起点
vector<vector<int>> adj;
vector<int> parent;

void dfs(int v) {
  for (int u : adj[v]) {
    if (u != parent[v]) parent[u] = v, dfs(u);
  }
}

vector<int> pruefer_code() {
  int n = adj.size();
  parent.resize(n), parent[n - 1] = -1;
  dfs(n - 1);

  int ptr = -1;
  vector<int> degree(n);
  for (int i = 0; i < n; i++) {
    degree[i] = adj[i].size();
    if (degree[i] == 1 && ptr == -1) ptr = i;
  }

  vector<int> code(n - 2);
  int leaf = ptr;
  for (int i = 0; i < n - 2; i++) {
    int next = parent[leaf];
    code[i] = next;
    if (--degree[next] == 1 && next < ptr) {
      leaf = next;
    } else {
      ptr++;
      while (degree[ptr] != 1) ptr++;
      leaf = ptr;
    }
  }
  return code;
}
```

```python
# 同样以 0 为起点
adj = [[]]
parent = [0] * n

def dfs()v:
```

```
        for u in adj[v]:
            if u != parent[v]:
                parent[u] = v
                dfs(u)

def pruefer_code():
    n = len(adj)
    parent[n - 1] = -1
    dfs(n - 1)

    ptr = -1
    degree = [0] * n
    for i in range(0, n):
        degree[i] = len(adj[i])
        if degree[i] == 1 and ptr == -1:
            ptr = i

    code = [0] * (n - 2)
    leaf = ptr
    for i in range(0, n - 2):
        next = parent[leaf]
        code[i] = next
        if degree[next] == 1 and next < ptr:
            degree[next] = degree[next] - 1
            leaf = next
        else:
            ptr = ptr + 1
            while degree[ptr] != 1:
                ptr = ptr + 1
            leaf = ptr
    return code
```

Prüfer 序列的性质

1. 在构造完 Prüfer 序列后原树中会剩下两个结点，其中一个一定是编号最大的点 n。
2. 每个结点在序列中出现的次数是其度数减 1。（没有出现的就是叶结点）

用 Prüfer 序列重建树

重建树的方法是类似的。根据 Prüfer 序列的性质，我们可以得到原树上每个点的度数。然后你也可以得到编号最小的叶结点，而这个结点一定与 Prüfer 序列的第一个数连接。然后我们同时删掉这两个结点的度数。

讲到这里也许你已经知道该怎么做了。每次我们选择一个度数为 1 的最小的结点编号，与当前枚举到的 Prüfer 序列的点连接，然后同时减掉两个点的度。到最后我们剩下两个度数为 1 的点，其中一个是结点 n。就把它们建立连接。使用堆维护这个过程，在减度数的过程中如果发现度数减到 1 就把这个结点添加到堆中，这样做的复杂度是 $O(n \log n)$ 的。

" 实现"

```
// 原文摘代码
vector<pair<int, int>> pruefer_decode(vector<int> const& code) {
  int n = code.size() + 2;
  vector<int> degree(n, 1);
  for (int i : code) degree[i]++;
```

```
  set<int> leaves;
  for (int i = 0; i < n; i++)
    if (degree[i] == 1) leaves.insert(i);

  vector<pair<int, int>> edges;
  for (int v : code) {
    int leaf = *leaves.begin();
    leaves.erase(leaves.begin());

    edges.emplace_back(leaf, v);
    if (--degree[v] == 1) leaves.insert(v);
  }
  edges.emplace_back(*leaves.begin(), n - 1);
  return edges;
}
```

线性时间重建树

同线性构造 Prüfer 序列的方法。在删度数的时侯会产生新的叶结点，于是判断这个叶结点与指针 p 的大小关系，如果更小就优先考虑它。

实现

```
// 原文摘代码
vector<pair<int, int>> pruefer_decode(vector<int> const& code) {
  int n = code.size() + 2;
  vector<int> degree(n, 1);
  for (int i : code) degree[i]++;

  int ptr = 0;
  while (degree[ptr] != 1) ptr++;
  int leaf = ptr;

  vector<pair<int, int>> edges;
  for (int v : code) {
    edges.emplace_back(leaf, v);
    if (--degree[v] == 1 && v < ptr) {
      leaf = v;
    } else {
      ptr++;
      while (degree[ptr] != 1) ptr++;
      leaf = ptr;
    }
  }
  edges.emplace_back(leaf, n - 1);
  return edges;
}
```

通过这些过程其实可以理解，Prüfer 序列与带标号无根树建立了双射关系。

Cayley 公式 (Cayley's formula)

完全图 K_n 有 n^{n-2} 棵生成树。

怎么证明？方法很多，但是用 Prüfer 序列证是很简单的。任意一个长度为 $n-2$ 的值域 $[1,n]$ 的整数序列都可以通过 Prüfer 序列双射对应一个生成树，于是方案数就是 n^{n-2}。

图连通方案数

Prüfer 序列可能比你想得还强大。它能创造比凯莱公式更通用的公式。比如以下问题：

> 一个 n 个点 m 条边的带标号无向图有 k 个连通块。我们希望添加 $k-1$ 条边使得整个图连通。求方案数。

证明

设 s_i 表示每个连通块的数量。我们对 k 个连通块构造 Prüfer 序列，然后你发现这并不是普通的 Prüfer 序列。因为每个连通块的连接方法很多。不能直接淦就设啊。于是设 d_i 为第 i 个连通块的度数。由于度数之和是边数的两倍，于是 $\sum_{i=1}^{k} d_i = 2k-2$。则对于给定的 d 序列构造 Prüfer 序列的方案数是

$$\binom{k-2}{d_1-1, d_2-1, \cdots, d_k-1} = \frac{(k-2)!}{(d_1-1)!(d_2-1)!\cdots(d_k-1)!}$$

对于第 i 个连通块，它的连接方式有 ${s_i}^{d_i}$ 种，因此对于给定 d 序列使图连通的方案数是

$$\binom{k-2}{d_1-1, d_2-1, \cdots, d_k-1} \cdot \prod_{i=1}^{k} {s_i}^{d_i}$$

现在我们要枚举 d 序列，式子变成

$$\sum_{d_i \ge 1,\ \sum_{i=1}^{k} d_i = 2k-2} \binom{k-2}{d_1-1, d_2-1, \cdots, d_k-1} \cdot \prod_{i=1}^{k} {s_i}^{d_i}$$

好的这是一个非常不喜闻乐见的式子。但是别慌！我们有多元二项式定理：

$$(x_1 + \cdots + x_m)^p = \sum_{c_i \ge 0,\ \sum_{i=1}^{m} c_i = p} \binom{p}{c_1, c_2, \cdots, c_m} \cdot \prod_{i=1}^{m} {x_i}^{c_i}$$

那么我们对原式做一下换元，设 $e_i = d_i - 1$，显然 $\sum_{i=1}^{k} e_i = k-2$，于是原式变成

$$\sum_{e_i \ge 0,\ \sum_{i=1}^{k} e_i = k-2} \binom{k-2}{e_1, e_2, \cdots, e_k} \cdot \prod_{i=1}^{k} {s_i}^{e_i+1}$$

化简得到

$$(s_1 + s_2 + \cdots + s_k)^{k-2} \cdot \prod_{i=1}^{k} s_i$$

即

$$n^{k-2} \cdot \prod_{i=1}^{k} s_i$$

这就是答案啦

习题

- UVa #10843 - Anne's game[2]
- Timus #1069 - Prufer Code[3]
- Codeforces - Clues[4]
- Topcoder - TheCitiesAndRoadsDivTwo[5]

参考资料与注释

[1] e-maxx Prüfer Code

[2] UVa #10843 - Anne's game

[3] Timus #1069 - Prufer Code

[4] Codeforces - Clues

[5] Topcoder - TheCitiesAndRoadsDivTwo

11.32 LGV 引理

简介

Lindström–Gessel–Viennot lemma，即 LGV 引理，可以用来处理有向无环图上不相交路径计数等问题。

前置知识：图论相关概念 中的基础部分、矩阵、高斯消元求行列式。

LGV 引理仅适用于**有向无环图**。

定义

$\omega(P)$ 表示 P 这条路径上所有边的边权之积。（路径计数时，可以将边权都设为 1）（事实上，边权可以为生成函数）

$e(u,v)$ 表示 u 到 v 的**每一条**路径 P 的 $\omega(P)$ 之和，即 $e(u,v)=\sum\limits_{P:u\to v}\omega(P)$。

起点集合 A，是有向无环图点集的一个子集，大小为 n。

终点集合 B，也是有向无环图点集的一个子集，大小也为 n。

一组 $A\to B$ 的不相交路径 S：S_i 是一条从 A_i 到 $B_{\sigma(S)_i}$ 的路径（$\sigma(S)$ 是一个排列），对于任何 $i\neq j$，S_i 和 S_j 没有公共顶点。

$t(\sigma)$ 表示排列 σ 的逆序对个数。

引理

$$M=\begin{bmatrix}e(A_1,B_1) & e(A_1,B_2) & \cdots & e(A_1,B_n)\\ e(A_2,B_1) & e(A_2,B_2) & \cdots & e(A_2,B_n)\\ \vdots & \vdots & \ddots & \vdots\\ e(A_n,B_1) & e(A_n,B_2) & \cdots & e(A_n,B_n)\end{bmatrix}$$

$$\det(M)=\sum_{S:A\to B}(-1)^{t(\sigma(S))}\prod_{i=1}^{n}\omega(S_i)$$

其中 $\sum\limits_{S:A\to B}$ 表示满足上文要求的 $A\to B$ 的每一组不相交路径 S。

证明

由行列式定义可得

$$\det(M) = \sum_{\sigma}(-1)^{t(\sigma)}\prod_{i=1}^{n} e(a_i, b_{\sigma(i)}) \tag{11.1}$$

$$= \sum_{\sigma}(-1)^{t(\sigma)}\prod_{i=1}^{n}\sum_{P:a_i\to b_{\sigma(i)}} \omega(P) \tag{11.2}$$

观察到 $\prod\limits_{i=1}^{n}\sum\limits_{P:a_i\to b_{\sigma(i)}} \omega(P)$，实际上是所有从 A 到 B 排列为 σ 的路径组 P 的 $\omega(P)$ 之和。

$$\sum_{\sigma}(-1)^{t(\sigma)}\prod_{i=1}^{n}\sum_{P:a_i\to b_{\sigma(i)}} \omega(P) \tag{11.3}$$

$$= \sum_{\sigma}(-1)^{t(\sigma)}\sum_{P=\sigma}\omega(P) \tag{11.4}$$

$$= \sum_{P:A\to B}(-1)^{t(\sigma)}\prod_{i=1}^{n}\omega(P_i) \tag{11.5}$$

此处 P 为任意路径组。

设 U 为不相交路径组，V 为相交路径组，

$$\sum_{P:A\to B}(-1)^{t(\sigma)}\prod_{i=1}^{n}\omega(P_i) \tag{11.6}$$

$$= \sum_{U:A\to B}(-1)^{t(U)}\prod_{i=1}^{n}\omega(U_i) + \sum_{V:A\to B}(-1)^{t(V)}\prod_{i=1}^{n}\omega(V_i) \tag{11.7}$$

设 P 中存在一个相交路径组 $P_i : a_1 \to u \to b_1, P_j : a_2 \to u \to b_2$，则必然存在和它相对的一个相交路径组 $P'_i = a_1 \to u \to b_2, P'_j = a_2 \to u \to b_1$，$P'$ 的其他路径与 P 相同。可得 $\omega(P) = \omega(P'), t(P) = t(P') \pm 1$。

因此我们有 $\sum\limits_{V:A\to B}(-1)^{t(\sigma)}\prod\limits_{i=1}^{n}\omega(V_i) = 0$。

则 $\det(M) = \sum\limits_{U:A\to B}(-1)^{t(U)}\prod\limits_{i=1}^{n}\omega(U_i) = 0$。

证毕[1-1]。

例题

" 例 1 CF348D Turtles[2]"

题意：有一个 $n\times m$ 的格点棋盘，其中某些格子可走，某些格子不可走。有一只海龟从 (x,y) 只能走到 $(x+1,y)$ 和 $(x,y+1)$ 的位置，求海龟从 $(1,1)$ 到 (n,m) 的不相交路径数对 10^9+7 取模之后的结果。$2 \le n,m \le 3000$。

比较直接的 LGV 引理的应用。考虑所有合法路径，发现从 $(1,1)$ 出发一定要经过 $A = \{(1,2),(2,1)\}$，而到达终点一定要经过 $B = \{(n-1,m),(n,m-1)\}$，则 A, B 可立即选定。应用 LGV 引理可得答案为：

$$\begin{vmatrix} f(a_1,b_1) & f(a_1,b_2) \\ f(a_2,b_1) & f(a_2,b_2) \end{vmatrix} = f(a_1,b_1)\times f(a_2,b_2) - f(a_1,b_2)\times f(a_2,b_1)$$

其中 $f(a,b)$ 为图上 $a \to b$ 的路径数，带有障碍格点的路径计数问题可以直接做一个 $O(nm)$ 的 dp，则 f 易求。最终复杂度 $O(nm)$。

"参考代码"

```cpp
#include <cstring>
#include <iostream>
#include <vector>

using namespace std;

using ll = long long;
const int MOD = 1e9 + 7;
const int SIZE = 3010;

char board[SIZE][SIZE];
int dp[SIZE][SIZE];

int f(int x1, int y1, int x2, int y2) {
  memset(dp, 0, sizeof dp);

  dp[x1][y1] = board[x1][y1] == '.';
  for (int i = 1; i <= x2; i++) {
    for (int j = 1; j <= y2; j++) {
      if (board[i][j] == '#') {
        continue;
      }
      dp[i][j] = (dp[i][j] + dp[i - 1][j]) % MOD;
      dp[i][j] = (dp[i][j] + dp[i][j - 1]) % MOD;
    }
  }
  return dp[x2][y2] % MOD;
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);
  cout.tie(nullptr);

  int n, m;
  cin >> n >> m;

  for (int i = 1; i <= n; i++) {
    cin >> (board[i] + 1);
  }

  ll f11 = f(1, 2, n - 1, m);
  ll f12 = f(1, 2, n, m - 1);
  ll f21 = f(2, 1, n - 1, m);
  ll f22 = f(2, 1, n, m - 1);

  ll ans = ((f11 * f22) % MOD - (f12 * f21) % MOD + MOD) % MOD;
  cout << ans << '\n';

  return 0;
}
```

" **例 2 hdu5852 Intersection is not allowed!**[3] "

题意：有一个 $n \times n$ 的棋盘，一个棋子从 (x, y) 只能走到 $(x, y+1)$ 或 $(x+1, y)$，有 k 个棋子，一开始第 i 个棋子放在 $(1, a_i)$，最终要到 (n, b_i)，路径要两两不相交，求方案数对 10^9+7 取模。$1 \le n \le 10^5$，$1 \le k \le 100$，保证 $1 \le a_1 < a_2 < \cdots < a_n \le n$，$1 \le b_1 < b_2 < \cdots < b_n \le n$。

观察到如果路径不相交就一定是 a_i 到 b_i，因此 LGV 引理中一定有 $\sigma(S)_i = i$，不需要考虑符号问题。边权设为 1，直接套用引理即可。

从 $(1, a_i)$ 到 (n, b_j) 的路径条数相当于从 $n-1+b_j-a_i$ 步中选 $n-1$ 步向下走，所以 $e(A_i, B_j) = \binom{n-1+b_j-a_i}{n-1}$。

行列式可以使用高斯消元求。

复杂度为 $O(n + k(k^2 + \log p))$，其中 $\log p$ 是求逆元复杂度。

" **参考代码** "

```
#include <algorithm>
#include <cstdio>

typedef long long ll;

const int K = 105;
const int N = 100005;
const int mod = 1e9 + 7;

int T, n, k, a[K], b[K], fact[N << 1], m[K][K];

int qpow(int x, int y) {
  int out = 1;
  while (y) {
    if (y & 1) out = (ll)out * x % mod;
    x = (ll)x * x % mod;
    y >>= 1;
  }
  return out;
}

int c(int x, int y) {
  return (ll)fact[x] * qpow(fact[y], mod - 2) % mod *
         qpow(fact[x - y], mod - 2) % mod;
}

int main() {
  fact[0] = 1;
  for (int i = 1; i < N * 2; ++i) fact[i] = (ll)fact[i - 1] * i % mod;

  scanf("%d", &T);

  while (T--) {
    scanf("%d%d", &n, &k);

    for (int i = 1; i <= k; ++i) scanf("%d", a + i);
    for (int i = 1; i <= k; ++i) scanf("%d", b + i);

    for (int i = 1; i <= k; ++i) {
```

```
      for (int j = 1; j <= k; ++j) {
        if (a[i] <= b[j])
          m[i][j] = c(b[j] - a[i] + n - 1, n - 1);
        else
          m[i][j] = 0;
      }
    }

    for (int i = 1; i < k; ++i) {
      if (!m[i][i]) {
        for (int j = i + 1; j <= k; ++j) {
          if (m[j][i]) {
            std::swap(m[i], m[j]);
            break;
          }
        }
      }
      if (!m[i][i]) continue;
      int inv = qpow(m[i][i], mod - 2);
      for (int j = i + 1; j <= k; ++j) {
        if (!m[j][i]) continue;
        int mul = (ll)m[j][i] * inv % mod;
        for (int p = i; p <= k; ++p) {
          m[j][p] = (m[j][p] - (ll)m[i][p] * mul % mod + mod) % mod;
        }
      }
    }

    int ans = 1;

    for (int i = 1; i <= k; ++i) ans = (ll)ans * m[i][i] % mod;

    printf("%d\n", ans);
  }

  return 0;
}
```

参考资料

[1] 证明来源于 知乎 - LGV 引理证明 [1-1] [1-2]

[2] CF348D Turtles

[3] hdu5852 Intersection is not allowed!

11.33 弦图

弦图是一种特殊的图，很多在一般图上的 NP-Hard 问题在弦图上都有优秀的线性时间复杂度算法。

一些定义与性质

子图：点集和边集均为原图点集和边集子集的图。

导出子图（诱导子图）：点集为原图点集子集，边集为所有满足**两个端点均在选定点集中**的图。

团：完全子图。

极大团：不是其他团子图的图。

最大团：点数最大的团。

团数：最大团的点数，记为 $\omega(G)$。

最小染色：用最少的颜色给点染色使得所有边连接的两点颜色不同。

色数：最小染色的颜色数，记为 $\chi(G)$。

最大独立集：最大的点集使得点集中任意两点都没有边直接相连。该集合的大小记为 $\alpha(G)$。

最小团覆盖：用最少的团覆盖所有的点。使用团的数量记为 $\kappa(G)$。

弦：连接环中不相邻两点的边。

弦图：任意长度大于 3 的环都有一个弦的图称为弦图。

Lemma 1：团数 $\omega(G) \le \chi(G)$ 色数

证明：考虑单独对最大团的导出子图进行染色，至少需要 $\omega(G)$ 种颜色。

Lemma 2：最大独立集数 $\alpha(G) \le \kappa(G)$ 最小团覆盖数

证明：每个团中至多选择一个点。

Lemma 3：弦图的任意导出子图一定是弦图。

证明：如果弦图有导出子图不是弦图，说明在这个导出子图上存在大于 3 的无弦环，则无论原图如何（怎么加边）都不会使得原图是弦图，矛盾。

Lemma 4：弦图的任意导出子图一定不可能是一个点数大于 3 的环。

证明：一个点数大于 3 的环不是弦图，用以上定理即可。

弦图的判定

问题描述

给定一个无向图，判断其是否为弦图。

点割集

对于图 G 上的两点 u,v，定义这两点间的**点割集**为满足删除这一集合后，u,v 两点之间不连通。如果关于 u,v 两点间的一个点割集的任意子集都不是点割集，则称这个点割集为**极小点割集**。

Lemma 5：图关于 u,v 的极小点割集将原图分成了若干个连通块，设包含 u 的连通块为 V_1，包含 v 的连通块为 V_2，则对于极小点割集上的任意一点 a，$N(a)$ 一定包含 V_1 和 V_2 中的点。

证明：若 $N(a)$ 只包含 V_1 或 V_2 中的至多一个连通块中的点，从点割集中删去 a 点，仍不连通，则原点割集不是最小点割集。

Lemma 6：弦图上任意两点间的极小点割集的导出子图一定为一个团。

证明：极小点割集大小 ≤ 1 时，导出子图一定为一个团。

否则，设极小点割集上有两点为 x,y，由 **Lemma 5** 得，$N(x)$ 中有 V_1,V_2 中的点，设为 x_1,x_2，同样的，设 y_1,y_2，注意，可能有 $x_1=y_1,x_2=y_2$。

由于 V_1,V_2 均为连通块，则在 x_1,y_1 和 x_2,y_2 两个点对之间存在最短路径。设 x,y 在 V_1,V_2 内部的最短路为 $x-x_1 \sim y_1-y, x-x_2 \sim y_2-y$，则图上存在一个环 $x-x_1 \sim y_1-y-y_2 \sim x_2-x$，该环的大小一定 ≥ 4，根据弦图的定义，此时该环上一定存在一条弦。

若这条弦连接了 V_1, V_2 两个连通块，则点集不是点割集。若这条弦连接了单个连通块内部的两个点或一个连通块内部的一个点和一个点割集上的点，都不满足最短路的性质。所以这条弦只能连接 x, y 两点。

由此，可证弦图中每个极小点割集中的两点都有边直接相连，故性质得证。

单纯点

设 $N(x)$ 表示与点 x 相邻的点集。若点集 $\{x\} + N(x)$ 的导出子图为一个团，则称点 x 为单纯点。

Lemma 7：任何一个弦图都至少有一个单纯点，不是完全图的弦图至少有两个不相邻的单纯点。

证明：数学归纳法。单独考虑每一连通块。

归纳基底：当图与完全图同构时，图上任意一点都是单纯点。当图的点数 ≤ 3 时，引理成立。

若图上的点数 ≥ 4 且图不为完全图，可知必然存在 u, v 使得 $(u, v) \notin E$。设 I 是图关于 u, v 的极小点割集。设 A, B 分别是删去 I 后的导出子图上 u, v 所在的连通块。由于问题的对称性，我们只考虑 A 一侧的情况，设 $L = A + I$。若 L 为完全图，则 u 为单纯点；若不是，因为 L 是原图的导出子图，一定也是弦图，所以有两个不相邻的单纯点，因为 I 是一个团，其上两点都相邻，所以 A 中一定有一个单纯点。该单纯点扩展到全图也为单纯点。

由于每次将整个图分成若干个连通块证明，大小一定减小，且都满足性质，故归纳成立。

完美消除序列

令 $n = |V|$，完美消除序列 $v_1, v_2, \ldots, v_n$ 为 $1, 2, \ldots, n$ 的一个排列，满足 v_i 在 $\{v_i, v_{i+1}, \ldots, v_n\}$ 的导出子图中为单纯点。

Lemma 8：一个无向图是弦图当且仅当其有一个完全消除序列。

充分性：点数为 1 的弦图有完全消除序列。由 **Lemma 3** 和 **Lemma 7**，点数为 n 的弦图的完美消除序列可以由点数为 $n-1$ 的弦图的完美消除序列加上一个单纯点得到。

必要性：假设有无向图存在结点数 > 3 的环且拥有完美消除序列，设在完美消除序列中出现的第一个环上的点为 v，设 v 在环上与 v_1, v_2 相连，则有完美消除序列的性质即单纯点的定义可得 v_1, v_2 直接有边相连，矛盾。

朴素算法

每次找到一个**单纯点** v，加入到完美消除序列中。

将点 v 与其相邻的边从图上删除。

重复以上过程，若所有点都被删除，则原图是弦图且求得了一个完美消除序列；若图上不存在单纯点，则原图不是弦图。

时间复杂度 $O(n^4)$。

MCS 算法

最大势算法（Maximum Cardinality Search）是一种可以在 $O(n+m)$ 的时间复杂度内求出无向图的完美消除序列的方法。

逆序给结点编号，即按从 n 到 1 的顺序给点标号。

设 $label_x$ 表示第 x 个点与多少个已经标号的点相邻，每次选择 $label$ 值最大的未标号结点进行标号。

用链表维护对于每个 i，满足 $label_x = i$ 的 x。

由于每条边对 $\sum_{i=1}^{n} label_i$ 的贡献最多是 2，时间复杂度 $O(n+m)$。

正确性证明：

设 $\alpha(x)$ 为 x 在这个序列中的位置。我们需要证明对于任何一个弦图，算法求出的序列一定是一个完美消除序列，即在序列中位于某个点后面且与这个点相连的所有点两两相连。

Lemma 9：考虑三个点 u, v, w 满足 $\alpha(u) < \alpha(v) < \alpha(w)$，如果 uw 相连，vw 不相连，则 w 只给 u 的 $label$ 贡献，不给 v 贡献。为了让 v 比 u 先加入序列，需要一个 x 满足 $\alpha(v) < \alpha(x)$ 且 vx 相连，ux 不相连，即 x 只给 v 贡

献而不给 u 贡献。

Lemma 10：任意一个弦图一定不存在一个序列 $v_0, v_1, \dots, v_k (k \geq 2)$ 满足下列性质：

1. $v_i v_j$ 相连当且仅当 $|i-j| = 1$。
2. $\alpha(v_0) > \alpha(v_i) (i \in [1, k])$。
3. 存在 $i \in [1, k-1]$，满足 $\alpha(v_i) < \alpha(v_{i+1}) < \cdots < \alpha(v_k)$ 且 $\alpha(v_i) < \alpha(v_{i-1}) < \cdots < \alpha(v_1) < \alpha(v_k) < \alpha(v_0)$。

证明：

由于 $\alpha(v_1) < \alpha(v_k) < \alpha(v_0)$，且 $v_1 v_0$ 相连，$v_k v_0$ 不相连，所以由性质一，存在 x 满足 $\alpha(v_k) < \alpha(x)$ 且 $v_k x$ 相连，$v_1 x$ 不相连。

考虑最小的 $j \in (1, k]$ 满足 $v_j x$ 相连，我们可以推出 $v_0 x$ 不相连，否则 $v_0 v_1 \cdots v_j x$ 构成了一个长度 ≥ 4 且无弦的环。

如果 $x < v_0$，则 $v_0, v_1, \dots, v_j, x$ 也是一个满足性质的序列；如果 $v_0 < x$ 则 $x, v_j, \dots, v_1, v_0$ 也是一个满足性质的序列。

在上面的推导中，我们扩大了 $\min(v_0, v_k)$，于是一直推下去，一定会产生矛盾。

Theorem 1：对于任何一个弦图，最大势算法求出的序列一定是一个完美消除序列。

证明：考虑任意三个点 u, v, w 满足 $\alpha(u) < \alpha(v) < \alpha(w)$，我们需要证明若 uv 相连，uw 相连，则 vw 一定相连。

考虑反证法，假设不相连，那么 w, u, v 就是一个满足 **Lemma 10** 中性质的序列，我们证明了这样序列不存在，所以矛盾，vw 相连。

参考代码：

```
while (cur) {
  p[cur] = h[nww];
  rnk[p[cur]] = cur;
  h[nww] = nxt[h[nww]];
  lst[h[nww]] = 0;
  lst[p[cur]] = nxt[p[cur]] = 0;
  tf[p[cur]] = true;
  for (vector<int>::iterator it = G[p[cur]].begin(); it != G[p[cur]].end();
       it++)
    if (!tf[*it]) {
      if (h[deg[*it]] == *it) h[deg[*it]] = nxt[*it];
      nxt[lst[*it]] = nxt[*it];
      lst[nxt[*it]] = lst[*it];
      lst[*it] = nxt[*it] = 0;
      deg[*it]++;
      nxt[*it] = h[deg[*it]];
      lst[h[deg[*it]]] = *it;
      h[deg[*it]] = *it;
    }
  cur--;
  if (h[nww + 1]) nww++;
  while (nww && !h[nww]) nww--;
}
```

如果此时原图是弦图，此时求出的就是完美消除序列；但是由于原图可能不是弦图，此时求出的一定不是完美消除序列，所以问题转化为**判断求出的序列是否是原图的完美消除序列**。

判断一个序列是否是完美消除序列

朴素算法

根据定义，依次判断完美消除序列 v 上 $\{v_i, v_{i+1}, \dots, v_n\}$ 中与 v_i 相邻的点是否构成了一个团。时间复杂度 $O(nm)$。

优化后的算法

根据完美消除序列的定义，设 v_i 在 $v_i, v_{i+1}, \cdots, v_n$ 中相邻的点从小到大为 $\{v_{c_1}, v_{c_2}, \cdots, v_{c_k}\}$，则只需判断 v_{c_1} 与其他点是否直接连通即可。时间复杂度 $O(n+m)$。

```
jud = true;
for (int i = 1; i <= n; i++) {
  cur = 0;
  for (vector<int>::iterator it = G[p[i]].begin(); it != G[p[i]].end(); it++)
    if (rnk[p[i]] < rnk[*it]) {
      s[++cur] = *it;
      if (rnk[s[cur]] < rnk[s[1]]) swap(s[1], s[cur]);
    }
  for (int j = 2; j <= cur; j++)
    if (!st[s[1]].count(s[j])) {
      jud = false;
      break;
    }
}
if (!jud)
  printf("Imperfect\n");
else
  printf("Perfect\n");
```

至此，**弦图判定问题**可以在 $O(n+m)$ 的时间复杂度内解决。

弦图的极大团

令 $N(x)$ 为满足与 x 直接有边相连且在完美消除序列上的 x 之后的序列。则弦图的极大团一定为 $\{x\}+N(x)$。

证明：考虑弦图的一个极大团 V，其中的点在完美消除序列中出现的第一个点 x，一定有 $V \subseteq \{x\}+N(x)$，又因为 V 是极大团，所以 $V=\{x\}+N(x)$。

弦图最多有 n 个极大团。求出弦图的每个极大团，可以判断每个 $\{x\}+N(x)$ 是否为极大团。

设 $A=\{x\}+N(x), B=\{y\}+N(y)$，若 $A \quad B$，则 A 不是极大团。此时在完美消除序列上显然有 y 在 x 前。

设 nxt_x 表示 $N(x)$ 中在完美消除序列上最靠前的点，$y*$ 表示所有满足 $A \subseteq B$ 的 y 中的最靠后的点。此时必然有 $nxt_{y*}=x$，否则 $y*$ 不是最靠后的，令 $y*=nxt_{y*}$ 仍然满足条件。

$A \quad B$ 当且仅当 $|A|+1 \le |B|$。

问题转化为判断是否存在 y，满足 $nxt_y=x$ 且 $|N(x)|+1 \le |N(y)|$。时间复杂度 $O(n+m)$。

```
for (int i = 1; i <= n; i++) {
  cur = 0;
  for (vector<int>::iterator it = G[p[i]].begin(); it != G[p[i]].end(); it++)
    if (rnk[p[i]] < rnk[*it]) {
      s[++cur] = *it;
      if (rnk[s[cur]] < rnk[s[1]]) swap(s[1], s[cur]);
    }
  fst[p[i]] = s[1];
  N[p[i]] = cur;
}
for (int i = 1; i <= n; i++) {
  if (!vis[p[i]]) ans++;
  if (N[p[i]] >= N[fst[p[i]]] + 1) vis[fst[p[i]]] = true;
}
```

弦图的色数/弦图的团数

一种构造方法：按完美消除序列从后往前依次给每个点染色，给每个点染上可以染的最小颜色。时间复杂度 $O(m+n)$。

正确性证明：设以上方法使用了 t 种颜色，则 $t \geq \chi(G)$。由于团上每个点都是不同的颜色，所以 $t=\omega(G)$，由 **Lemma 1**，$t=\omega(G) \leq \chi(G)$。综上，可得 $t=\chi(G)=\omega(G)$。

无需染色方案，只需求出弦图的色数/团数时，可以取 $|\{x\}+N(x)|$ 的最大值得到。

```
for (int i = 1; i <= n; i++) ans = max(ans, deg[i] + 1);
```

弦图的最大独立集/最小团覆盖

最大独立集：完美消除序列从前往后，选择所有没有与已经选择的点有直接连边的点。

最小团覆盖：设最大独立集为 $\{v_1, v_2, \dots, v_t\}$，则团的集合 $\{\{v_1+N(v_1)\}, \{v_2+N(v_2)\}, \dots, \{v_t+N(v_t)\}\}$ 为图的最小团覆盖。时间复杂度均为 $O(n+m)$。

正确性证明：设以上方案独立集数和团覆盖数为 t，由定义得 $t \leq \alpha(G), t \geq \kappa(G)$，由 **Lemma 2** 得，$\alpha(G) \leq \kappa(G)$，所以 $t=\alpha(G)=\kappa(G)$。

```
for (int i = 1; i <= n; i++)
  if (!vis[p[i]]) {
    ans++;
    for (vector<int>::iterator it = G[p[i]].begin(); it != G[p[i]].end(); it++)
      vis[*it] = true;
  }
```

习题

SP5446 FISHNET - Fishing Net[1]

P3196[HNOI2008] 神奇的国度[2]

P3852[TJOI2007] 小朋友[3]

参考资料

弦图相关[4]

2009 WC 讲稿[5]

弦图总结 - 租酥雨[6]

R. E. Tarjan and M. Yannakakis, Simple linear-time algorithms to test chordality of graphs,test acyclicity of hypergraphs,and selectively reduce acyclic hypergraphs, SIAM J. Comput., 13 (1984), pp. 566–579.[7]

参考资料与注释

[1] SP5446 FISHNET - Fishing Net

[2] P3196[HNOI2008] 神奇的国度

[3] P3852[TJOI2007] 小朋友

[4] 弦图相关

[5] 2009 WC 讲稿

[6] 弦图总结 - 租酥雨

[7] R. E. Tarjan and M. Yannakakis, Simple linear-time algorithms to test chordality of graphs,test acyclicity of hypergraphs,and selectively reduce acyclic hypergraphs, SIAM J. Comput., 13 (1984), pp. 566–579.

11.34 最大团搜索算法

Authors: Persdre

前置知识：团

引入

在计算机科学中，团问题指的是在给定的图中找到团（顶点的子集，都彼此相邻，也称为完全子图）的计算问题。

团的问题在现实生活中也有体现。例如我们考虑一个社交网络，其中图的点代表用户，图的边代表其所连接的两个用户互相认识。那么我们找到了一个团，也就找到了一群互相认识的人。

我们如果想要找到这个社交网络中最大的一群互相认识的人，那么就需要用到最大团搜索算法。

我们已经介绍了 极大团 的概念，最大团指的是点数量最多的极大团。

解释

想法是利用递归和回溯，用一个列表存储点，每次加入点进来都检查这些点是否仍在一个团中。如果加入进来这个点后就无法还是一个团了，就回溯到满足条件的位置，重新加入别的点。

采用回溯策略的原因是，我们并不知道某个顶点 v **最终**是否是最大团中的成员。如果递归算法选择 v 作为最大团的成员时，并没有找到最大团，那么应该回溯，并查找最大团中没有 v 的解。

过程

Bron–Kerbosch 算法对于这种想法进行了优化实现。它的基础形式是通过给定三个集合：R、P、X 来递归地进行搜索。步骤如下：

1. 初始化集合 R, X 分别为空，集合 P 是图中所有点的集合。
2. 每次从集合 P 中取顶点 v，当集合中没有顶点时，有两种情况：
   (a) 集合 R 是最大团，此时集合 X 为空
   (b) 无最大团，此时回溯
3. 对于每一个从集合 P 中取得的顶点 v，有如下处理：
   (a) 将顶点 v 加到集合 R 中，之后递归集合 R, P, X
   (b) 从集合 P 中删除顶点 v，并将顶点 v 添加到集合 X 中
   (c) 若集合 P, X 都为空，则集合 R 即为最大团

此方法也可继续优化。为了节省时间让算法更快的回溯，可以通过设定关键点（pivot vertex）来进行搜索。另一种优化思路是在开始时把所有点排序，枚举时按照下标顺序，防止重复。

实现

伪代码

```
R := {}
P := node set of G
X := {}

BronKerbosch1(R, P, X):
    if P and X are both empty:
        report R as a maximal clique
    for each vertex v in P:
        BronKerbosch1(R   {v}, P   N(v), X   N(v))
        P := P \ {v}
        X := X   {v}
```

C++ 实现

" 实现代码"

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <iostream>
using namespace std;
const int MAXN = 105;

struct MaxClique {
  bool g[MAXN][MAXN];
  int n, dp[MAXN], st[MAXN][MAXN], ans;

  // dp[i] 表示第 i 个点之后能组成的最大团的大小，
  // st[i][j] 表示算法中第 i 层 dfs 所需要的点的集合，保存有可能是最大团其中之一的点

  void init(int n) {
    this->n = n;
    memset(g, false, sizeof(g));
  }

  void addedge(int u, int v, int w) { g[u][v] = w; }

  bool dfs(int sz, int num) {
    if (sz == 0) {
      if (num > ans) {
        ans = num;
        return true;
      }
      return false;
    }
    for (int i = 0; i < sz; i++) {  // 在第 num 层的集合中枚举一个点 i
      if (sz - i + num <= ans) return false;  // 剪枝 1
      int u = st[num][i];
```

```
      if (dp[u] + num <= ans) return false;  // 剪枝 2
      int cnt = 0;
      for (
          int j = i + 1; j < sz;
          j++) {  // 在第 num 层遍历在 i 之后的且与 i 所相连的点，并且加入第 num+1 层集合
        if (g[u][st[num][j]]) st[num + 1][cnt++] = st[num][j];
      }
      if (dfs(cnt, num + 1)) return true;
    }
    return false;
  }

  int solver() {
    ans = 0;
    memset(dp, 0, sizeof(dp));
    for (int i = n; i >= 1; i--) {
      int cnt = 0;
      for (int j = i + 1; j <= n; j++) {  // 初始化第 1 层集合
        if (g[i][j]) st[1][cnt++] = j;
      }
      dfs(cnt, 1);
      dp[i] = ans;
    }
    return ans;
  }

} maxclique;

int main() {
  int n;
  while (scanf("%d", &n), n) {
    maxclique.init(n);
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++) {
        int x;
        scanf("%d", &x);
        maxclique.addedge(i, j, x);
      }
    }
    printf("%d\n", maxclique.solver());
  }
  return 0;
}
```

例题

"POJ 2989: All Friends[1]"

题目大意：给出 n 个人，其中有 m 对朋友，求最大团数量。

思路：模版题，要用 Bron–Kerbosch 算法

伪代码：

```
BronKerbosch(All, Some, None):
    if Some and None are both empty:
        report All as a maximal clique // 所有点已选完，且没有不能选的点，累加答案

    for each vertex v in Some: // 枚举 Some 中的每一个元素
        BronKerbosch1(All   {v}, Some   N(v), None   N(v))
        // 将 v 加入 All，显然只有与 v 为朋友的人才能作为备选，None 中也只有与 v
为朋友的才会对接下来造成影响
        Some := Some - {v} // 已经搜过，从 Some 中删除，加入 None
        None := None   {v}
```

为了节省时间和让算法更快的回溯，我们可以通过设定关键点（pivot vertex）v 进行优化。

我们知道在上述的算法中必然有许多重复计算之前计算过的极大团，然后回溯的过程。

以前文提到的 R、P、X 三个集合为例：

我们考虑如下问题，取集合 $P \cup X$ 中的一个点 u，要与 R 集合构成极大团，那么取的点必然是 $P \cap N(u)$ 中一个点（$N(u)$ 代表与 u 相邻的点）。

如果取完 u 之后我们再取与 u 相邻的点 v 也能加入到极大团，那么我们只取 u 就好了。这样做可以减少之后对 v 的重复计算。我们之后只需要取与 u 不相邻的点。

加入优化后的 C++ 代码实现：

" 实现代码"

```cpp
#include <cstdio>
#include <cstring>
using namespace std;
const int maxn = 130;
bool mp[maxn][maxn];
int some[maxn][maxn], none[maxn][maxn], all[maxn][maxn];
int n, m, ans;

void dfs(int d, int an, int sn, int nn) {
  if (!sn && !nn) ++ans;
  int u = some[d][0];
  for (int i = 0; i < sn; ++i) {
    int v = some[d][i];
    if (mp[u][v]) continue;
    for (int j = 0; j < an; ++j) all[d + 1][j] = all[d][j];
    all[d + 1][an] = v;
    int tsn = 0, tnn = 0;
    for (int j = 0; j < sn; ++j)
      if (mp[v][some[d][j]]) some[d + 1][tsn++] = some[d][j];
    for (int j = 0; j < nn; ++j)
      if (mp[v][none[d][j]]) none[d + 1][tnn++] = none[d][j];
    dfs(d + 1, an + 1, tsn, tnn);
    some[d][i] = 0, none[d][nn++] = v;
    if (ans > 1000) return;
  }
}

int work() {
  ans = 0;
  for (int i = 0; i < n; ++i) some[1][i] = i + 1;
  dfs(1, 0, n, 0);
```

```
  return ans;
}

int main() {
  while (~scanf("%d %d", &n, &m)) {
    memset(mp, 0, sizeof mp);
    for (int i = 1; i <= m; ++i) {
      int u, v;
      scanf("%d %d", &u, &v);
      mp[u][v] = mp[v][u] = 1;
    }
    int tmp = work();
    if (tmp > 1000)
      puts("Too many maximal sets of friends.");
    else
      printf("%d\n", tmp);
  }
  return 0;
}
```

习题

- ZOJ 1492 最大团[2]
- POJ 1419 无向图最大团[3]
- POJ 1129 广播电台[4]

参考资料

- 团问题 - 维基百科[5]
- 无向图的极大团 、 最大团 （Bron–Kerbosch 算法 ）[6]
- 最大团问题 ——Bron–Kerbosch 算法[7]
- 最大团问题[8]

参考资料与注释

[1] POJ 2989: All Friends

[2] ZOJ 1492 最大团

[3] POJ 1419 无向图最大团

[4] POJ 1129 广播电台

[5] 团问题 - 维基百科

[6] 无向图的极大团、最大团（Bron–Kerbosch 算法）

[7] 最大团问题——Bron–Kerbosch 算法

[8] 最大团问题

11.35 支配树

前言

1959 年，「支配」这一概念由 Reese T. Prosser 在 一篇关于网络流的论文[1] 中提出，但并未提出具体的求解算法；直到 1969 年，Edward S. Lowry 和 C. W. Medlock 才首次提出了 有效的求解算法[2]。而目前使用最为广泛的 Lengauer–Tarjan 算法则由 Lengauer 和 Tarjan 于 1979 年在 一篇论文[3] 中提出。

在 OI 界中，支配树的概念最早在 ZJOI2012 灾难[4] 中被引入，当时也被称为「灭绝树」；陈孙立也在 2020 年的国家集训队论文中介绍了这一算法。

目前支配树在竞赛界并不流行，其相关习题并不多见；但支配树在工业上，尤其是编译器相关领域，已有广泛运用。

本文将介绍支配树的概念及几种求解方法。

支配关系

我们在任意的一个有向图上钦定一个入口结点 s，对于一个结点 u，若从 s 到 u 的每一条路径都经过某一个结点 v，那么我们称 v **支配** u，也称 v 是 u 的一个**支配点**，记作 $v\ dom\ u$。

对于从 s 出发无法到达的结点，讨论其支配关系是没有意义的，因此在没有特殊说明的情况下，本文默认 s 能到达图上任何一个结点。

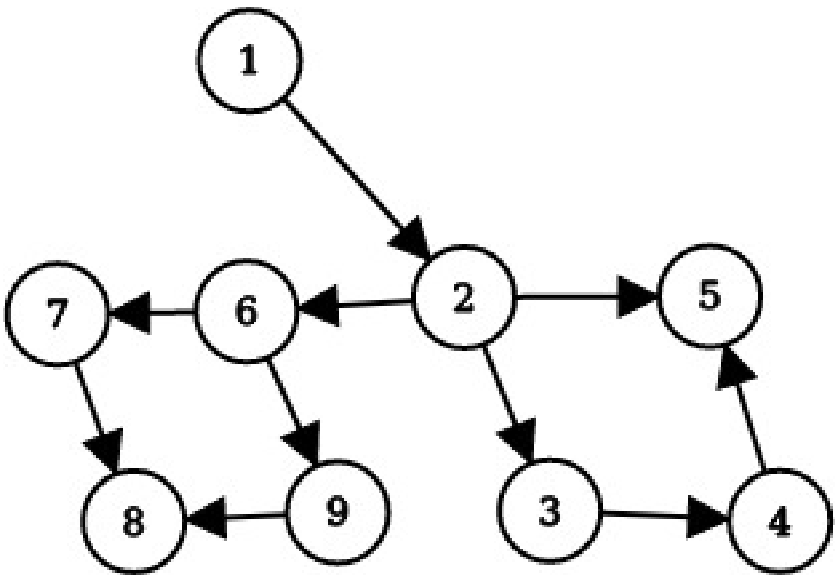


图 11.113

例如这张有向图中，2 被 1 支配，3 被 1,2 支配，4 被 1,2,3 支配，5 被 1,2 支配，etc。

引理

在下文的引理中，默认 $u, v, w \neq s$

引理 1： s 是其所有结点的支配点；任意一个结点都是其自身的支配点。

证明： 显然任何一条从 s 到 u 的路径都必须经过 s 和 u 这两个结点。

引理 2： 仅考虑简单路径得出的支配关系与考虑所有路径得出的关系相同。

证明： 对于非简单路径，我们设两次经过某个结点之间经过的所有结点的点集为 S，若将 S 中的结点删去，便能将每个非简单路径与一个简单路径对应。

在 S 中，在非简单路径而不在简单路径上的点一定不可能成为支配点，因为至少有一条 s 到 u 的简单路径不包括这个点；同时在简单路径和非简单路径上的点只需在简单路径上讨论即可。

综上，删去非简单路径对支配关系没有影响。

引理 3： 如果 $u\ dom\ v$，$v\ dom\ w$，则 $u\ dom\ w$。

证明： 经过 w 的路径必定经过 v，经过 v 的路径必定经过 u，因此经过 w 的路径必定经过 u，即 $u\ dom\ w$。

引理 4： 如果 $u\ dom\ v$，$v\ dom\ u$，则 $u = v$。

证明： 假设 $u \neq v$，则任意一个到达 v 的路径都已经到达过 u，同时任意一个到达 u 的路径都已经到达过 v，矛盾。

引理 5： 若 $u \neq v \neq w$,$u\ dom\ w$ 且 $v\ dom\ w$，则有 $u\ dom\ v$ 或 $v\ dom\ u$。

证明： 考虑一条 $s \to \cdots \to u \to \cdots \to v \to \cdots \to w$ 的路径，若 u,v 不存在支配关系，则一定存在一条不经过 u 的从 s 到 v 的路径，即存在一条 $s \to \cdots \to v \to \cdots \to w$ 的路径，与 $u\ dom\ w$ 矛盾。

求解支配关系

结点删除法

一个和定义等价的结论：如果我们删去图中的某一个结点后，有一些结点变得不可到达，那么这个被删去的结点支配这些变得不可到达的结点。

因此我们只要尝试将每一个结点删去后 dfs 即可，代码复杂度为 $O(n^3)$。下面给出核心代码。

```
// 假设图中有 n 个结点，起始点 s = 1
std::bitset<N> vis;
std::vector<int> edge[N];
std::vector<int> dom[N];

void dfs(int u, int del) {
  vis[u] = true;
  for (int v : edge[u]) {
    if (v == del or vis[v]) {
      continue;
    }
    dfs(v, del);
  }
}

void getdom() {
  for (int i = 2; i <= n; ++i) {
    vis.reset();
    dfs(1, i);
    for (int j = 1; j <= n; ++j) {
      if (!vis[j]) {
        dom[j].push_back(i);
      }
    }
  }
}
```

数据流迭代法

数据流迭代法也是 OI 中不常见的一个知识点，这里先做简要介绍。

数据流分析是编译原理中的概念，用于分析数据如何在程序执行路径上的流动；而数据流迭代法是在程序的流程图的结点上列出方程并不断迭代求解，从而求得程序的某些点的数据流值的一种方法。这里我们就是把有向图看成了一个程序流程图。

这个问题中，方程为：

$$dom(u) = \{u\} \cup \left(\bigcap_{v \in pre(u)} dom(v) \right)$$

其中 $pre(u)$ 定义为 u 的前驱结点组成的点集。这个方程可以通过引理 3 得到。

翻译成人话就是，一个点的支配点的点集为它所有前驱结点的支配点集的交集，再并上它本身。根据这个方程将每个结点上的支配点集不断迭代直至答案不变即可。

为了提高效率，我们希望每轮迭代时，当前迭代的结点的所有前驱结点尽可能都已经执行完了这次迭代，因此我们要利用深度有限排序得出这个图的逆后序，根据这个顺序进行迭代。

下面给出核心代码的参考实现。这里需要预先处理每个点的前驱结点集和图的逆后序，但这不是本文讨论的主要内容，故这里不提供参考实现。

```cpp
std::vector<int> pre[N];  // 每个结点的前驱结点
std::vector<int> ord;     // 图的逆后序
std::bitset<N> dom[N];
std::vector<int> Dom[N];

void getdom() {
  dom[1][1] = true;
  flag = true;
  while (flag) {
    flag = false;
    for (int u : ord) {
      std::bitset<N> tmp;
      tmp[u] = true;
      for (int v : pre[u]) {
        tmp &= dom[v];
      }
      if (tmp != dom[u]) {
        dom[u] = tmp;
        flag = true;
      }
    }
  }
  for (int i = 2; i <= n; ++i) {
    for (int j = 1; j <= n; ++j) {
      if (dom[i][j]) {
        Dom[i].push_back(j);
      }
    }
  }
}
```

不难看出上述算法的复杂度为 $O(n^2)$。

支配树

上一节我们发现，除 s 外，一个点的支配点至少有两个，s 和其自身。

我们将任意一个结点 u 的支配点中，除自身外与自己距离最近的结点 v 称作 u 的直接支配点，记作 $idom(u) = v$。显然除了 s 没有直接支配点外，每个结点都有唯一一个直接支配点。

我们考虑对于除 s 外每一个结点 u 从 $idom(u)$ 向 u 连边，便构成了一个有 n 个结点，$n-1$ 条边的有向图。根据引理 3 和引理 4，我们知道支配关系一定不会构成循环，也就是这些边一定不构成环，因此我们得到的图事实上是一棵树。我们称这颗树为原图的**支配树**。

求解支配树

根据 dom 求解

不妨考虑某个结点的支配点集 $\{s_1, s_2, \dots, s_k\}$，则一定存在一条路径 $s \to \dots \to s_1 \to \dots \to s_2 \to \dots \to \dots \to s_k \to \dots \to u$。显然 u 的直接支配点为 s_k。因此直接支配点的定义等价于：

对于一个结点 u 的支配点集 S，若 $v \in S$ 满足 $\forall w \in S \quad \{u, v\}, w\ dom\ v$，则 $idom(u) = v$。

因此，利用前文所述的算法得到每个结点的支配点集之后，我们根据上述定义便能很轻松地得到每个点的直接支配点，从而构造出支配树。下面给出参考代码。

```
std::bitset<N> dom[N];
std::vector<int> Dom[N];
int idom[N];

void getidom() {
  for (int u = 2; u <= n; ++u) {
    for (int v : Dom[u]) {
      std::bitset<N> tmp = (dom[v] & dom[u]) ^ dom[u];
      if (tmp.count() == 1 and tmp[u]) {
        idom[u] = v;
        break;
      }
    }
  }
  for (int u = 2; u <= n; ++u) {
    e[idom[u]].push_back(u);
  }
}
```

树上的特例

显然树型图的支配树就是它本身。

DAG 上的特例

我们发现 DAG 有一个很好的性质：根据拓扑序求解，先求得的解不会对后续的解产生影响。我们可以利用这个特点快速求得 DAG 的支配树。

引理 6： 在有向图上，$v\ dom\ u$ 当且仅当 $\forall w \in pre(u), v\ dom\ w$。

证明： 首先来证明充分性。考虑任意一条从 s 到 u 的路径都一定经过一个结点 $w \in pre(u)$，而 v 支配这个结点，因此任意一条从 s 到 u 的路径都一定经过 v，因此我们得到 $v\ dom\ u$。

然后是必要性。如果 $\exists w \in pre(u)$，v 不支配 w，则一定有一条不经过 v 的路径 $s \to \dots \to w \to \dots \to u$，因此 v 不支配 u。

我们发现，u 的支配点一定是其所有前驱结点在支配树上的公共祖先，那么显然 u 的直接支配点是所有前驱结点在支配树上的 LCA。考虑倍增求解 LCA 可以支持每次添加一个结点，上述算法显然是可行的。

下面给出参考实现：

```
std::stack<int> sta;
vector<int> e[N], g[N], tree[N];  // g 是原图的反图，tree 是支配树
int in[N], tpn[N], dep[N], idom[N];
int fth[N][17];

void topo() {
  for (int i = 1; i <= n; ++i) {
    if (!in[i]) {
      sta.push(i);
    }
  }
  while (!sta.empty()) {
    int u = sta.top();
    sta.pop();
    tpn[++tot] = u;
    for (int v : e[u]) {
      --in[v];
      if (!in[v]) {
        sta.push(v);
      }
    }
  }
}

int lca(int u, int v) {
  if (dep[u] < dep[v]) {
    std::swap(u, v);
  }
  for (int i = 15; i >= 0; --i) {
    if (dep[fth[u][i]] >= dep[v]) {
      u = fth[u][i];
    }
  }
  if (u == v) {
    return u;
  }
  for (int i = 15; i >= 0; --i) {
    if (fth[u][i] != fth[v][i]) {
      u = fth[u][i];
      v = fth[v][i];
    }
  }
  return fth[u][0];
}

void build() {
  topo();
  for (int i = 1; i <= n; ++i) {
    int u = tpn[i], v = g[u][0];
    for (int j = 1, q = g[u].size(); j < q; ++j) {
      v = lca(v, g[u][j]);
    }
    idom[u] = v;
    tree[v].push_back(u);
```

```
    fth[u][0] = v;
    dep[u] = dep[v] + 1;
    for (int i = 1; i <= 15; ++i) {
      fth[u][i] = fth[fth[u][i - 1]][i - 1];
    }
  }
}
```

Lengauer–Tarjan 算法

Lengauer–Tarjan 算法是求解支配树最有名的算法之一，可以在 $O(n\alpha(n,m))$ 的时间复杂度内求出一个有向图的支配树。这一算法引入了**半支配点**的概念，并通过半支配点辅助求得直接支配点。

约定

首先，我们从 s 出发对这个有向图进行 dfs，所经过的点和边形成了一颗树 T。我们称走过的边为树边，其余的为非树边；令 $dfn(u)$ 表示结点 u 被第几个遍历到；定义 $u<v$ 当且仅当 $dfn(u)<dfn(v)$。

半支配点

一个结点 u 的半支配点，是满足从这个结点 v 出发有一条路径，路径上除了 u,v 之外每个结点都大于 u 的结点中最小的那一个。形式化的说，u 的半支配点 $sdom(u)$ 定义为：

$sdom(u)=\min(v|\exists v=v_0\to v_1\to\cdots\to v_k=u,\forall 1\le i\le k-1,v_i>u)$

我们发现半支配点有一些有用的性质：

引理 7： 对于任意结点 u，$sdom(u)<u$。

证明： 根据定义不难发现，u 在 T 上的父亲 $fa(u)$ 也满足成为半支配点的条件，且 $fa(u)<u$，因此任何大于 u 的结点都不可能成为其半支配点。

引理 8： 对于任意结点 u，$idom(u)$ 是其在 T 上的祖先。

证明： T 上从 s 到 u 的路径对应了原图上的一条路径，则 $idom(u)$ 必定在这个路径上。

引理 9： 对于任意结点 u，$sdom(u)$ 是其在 T 上的祖先。

证明： 假设 $sdom(u)$ 不是 u 的祖先，那么 $sdom(u)$ 不可能连向任何 dfs 序大于等于 u 的结点（否则这个点应在 $sdom(u)$ 的子树内而非其他子树内），矛盾。

引理 10： 对于任意结点 u，$idom(u)$ 是 $sdom(u)$ 的祖先。

证明： 考虑可以从 s 到 $sdom(u)$ 再从定义中的路径走到 u。根据定义，$sdom(u)$ 到 u 的路径上的点均不支配 u，故 $idom(u)$ 一定是 $sdom(u)$ 的祖先。

引理 11： 对于任意结点 $u\neq v$ 满足 v 是 u 的祖先，则要么有 v 是 $idom(u)$ 的祖先，要么 $idom(u)$ 是 $idom(v)$ 的祖先。

证明： 对于任意在 v 和 $idom(v)$ 之间的结点 w，根据直接支配点的定义，一定存在一条不经过 w 的，从 s 到 $idom(v)$ 再到 v 的路径。因此这些结点 w 一定不是 $idom(u)$，因此 $idom(u)$ 要么是 v 的后代，要么是 $idom(v)$ 的祖先。

根据以上引理，我们可以得到以下定理：

定理 1： 一个点 u 的半支配点是其前驱与其支配点在 T 上的，大于 u 的所有祖先的半支配点中最小的节点。形式化地说，$sdom(u)=\min(\{v|\exists v\to u,v<u\}\cup\{sdom(w)|w>u\ and\ \exists w\to\cdots\to v\to u\})$。

证明： 令 x 等于上式右侧。

我们首先证明 $sdom(u)\le x$。根据引理 7 我们知道这个命题等价于证明上述的两种都满足成为半支配点的条件。x 是 u 的前驱时的情况是显然的，对于后半部分，我们考虑将半支配点定义中所述路径 $x=v_0\to\cdots\to v_j=w$ 和 T 上的一条满足 $\forall i\in[j,k-1],v_i\ge w>u$ 的路径 $w=v_j\to\cdots\to v_k=v$ 以及路径 $v\to u$ 拼接，从而我们构造出一条满足半支配点定义的路径。

然后我们证明 $sdom(u)\ge x$。考虑 u 到其半支配点的定义中所述路径 $sdom(u)=v_0\to v_1\to\cdots\to v_k=u$。不难

看出 $k=1$ 和 $k>1$ 分别对应了定义中的两个选取方法。若 $k=1$，则存在有向边 $sdom(u)\to u$，根据引理 7 即可得证；若 $k>1$，令 j 是满足 $j\ge 1$ 且 v_j 是 v_{k-1} 在 T 上祖先的最小数。考虑到 k 满足上述条件，这样的 j 一定存在。

考虑证明 $v_0\to\cdots\to v_j$ 是满足成为 v_j 半支配点条件的一条路径，即证明 $\forall i\in[1,j), v_i>v_j$。若不是，则令 i 为满足 $v_i<v_j$ 中使 v_i 最小的数，根据引理 11 我们知道 v_i 是 v_j 的祖先，这和 j 的定义矛盾。于是 $sdom(v_j)\le sdom(u)$。综上 $sdom(u)\le x$，故 $x=sdom(u)$。

根据定理 1 我们便可以求出每个点的半支配点了。不难发现计算半支配点的复杂度瓶颈在第二种情况上，我们考虑利用带权并查集优化，每次路径压缩时更新最小值即可。

```
void dfs(int u) {
  dfn[u] = ++dfc;
  pos[dfc] = u;
  for (int i = h[0][u]; i; i = e[i].x) {
    int v = e[i].v;
    if (!dfn[v]) {
      dfs(v);
      fth[v] = u;
    }
  }
}

int find(int x) {
  if (fa[x] == x) {
    return x;
  }
  int tmp = fa[x];
  fa[x] = find(fa[x]);
  if (dfn[sdm[mn[tmp]]] < dfn[sdm[mn[x]]]) {
    mn[x] = mn[tmp];
  }
  return fa[x];
}

void getsdom() {
  dfs(1);
  for (int i = 1; i <= n; ++i) {
    mn[i] = fa[i] = sdm[i] = i;
  }
  for (int i = dfc; i >= 2; --i) {
    int u = pos[i], res = INF;
    for (int j = h[1][u]; j; j = e[j].x) {
      int v = e[j].v;
      if (!dfn[v]) {
        continue;
      }
      find(v);
      if (dfn[v] < dfn[u]) {
        res = std::min(res, dfn[v]);
      } else {
        res = std::min(res, dfn[sdm[mn[v]]]);
      }
    }
    sdm[u] = pos[res];
    fa[u] = fth[u];
```

```
    }
  }
```

求解直接支配点

转化为 DAG

可是我还是不知道半支配点有什么用！

我们考虑在 T 上对每一个 u 加入 $sdom(u) \to u$ 的有向边。根据引理 9，新得到的这张图 G 一定是有向无环图；又根据引理 10，我们还发现这样加边不会改变支配关系，因此我们把原图转化为了一张 DAG，利用上文的算法求解即可。

通过半支配点求解

建一堆图也太不优雅了！

定理 2： 对于任意节点 u，若 T 上从 $sdom(u)$ 到 w 的路径上的任意节点 v 都满足 $sdom(v) \geq sdom(w)$，则 $idom(u) = sdom(u)$。

证明： 根据引理 10 我们知道 $idom(u)$ 是 $sdom(u)$ 或其祖先，因此只需证明 $sdom(u)\ dom\ u$。

考虑任意一条 s 到 u 的路径 P，我们需要证明 $sdom(u)$ 一定在 P 中。令 v 为 P 中最后一个满足 $v < sdom(u)$ 的节点。如果 v 不存在则必有 $sdom(u) = idom(u) = s$，否则令 w 是 P 中 v 之后在 DFS 树中从 $sdom(u)$ 到 u 的路径上的第一个点。

我们接下来证明 $sdom(w) \leq v < sdom(v)$。考虑 T 上 v 到 w 的路径 $v = v_0 \to \dots v_k = w$，若不成立，则存在 $i \in [1, k-1], v_i < w$。此时一定存在某个 $j \in [i, k-1]$ 满足 v_j 是 w 的祖先。由 v 的取值可知 $sdom(u) \leq v_j$，于是 v_j 也在 DFS 树中从 $sdom(u)$ 到 u 的路径上，与 w 的定义矛盾，因此 $sdom(w) \leq v < sdom(v)$，结合定理的条件有 $y = sdom(u)$，即路径 P 包含 $sdom(u)$。

定理 3： 对于任意节点 u，T 上从 $sdom(u)$ 到 u 的路径上的所有节点中半支配点最小的节点 v 一定满足 $sdom(v) \leq sdom(u)$ 和 $idom(v) = idom(u)$。

证明： 考虑到 u 本身也满足 v 的条件，因此 $sdom(v) \leq sdom(u)$。

由于 $idom(u)$ 是 v 在 T 上的祖先，由引理 11 可知 $idom(u)$ 也是 $idom(v)$ 的祖先，因此只需证明 $idom(v)$ 支配 u。

考虑任意一条 s 到 u 的路径 P，我们需要证明 $sdom(u)$ 一定在 P 中。令 x 为 P 中最后一个满足 $x < sdom(u)$ 的节点。如果 x 不存在则必有 $sdom(u) = idom(u) = s$，否则令 y 是 P 中 x 之后在 DFS 树中从 $sdom(u)$ 到 u 的路径上的第一个点。

与定理 2 的证明过程同理，我们可以得到 $sdom(y) \leq x$。根据引理 10 有 $sdom(y) \leq x < idom(v) \leq sdom(v)$。至此，由 v 的定义可知 y 不能是 $sdom(u)$ 的后代；另一方面，y 不能既是 $idom(v)$ 的后代也是 v 的祖先，否则沿 DFS 树从 s 到 $sdom(y)$ 再沿 P 走到 y，最后沿 DFS 树走到 v 的这条路径不经过 $idom(v)$，与支配点的定义矛盾。因此 $y = idom(v)$，即 P 包含 $idom(v)$。

根据以上两个定理我们能够得到 $sdom(u)$ 与 $idom(u)$ 之间的关系。

令 v 是满足 v 在 $sdom(u)$ 与 u 之间的结点的所有节点中，$sdom(v)$ 最小的一个节点，那么：

$$idom(u) = \begin{cases} sdom(u), & \text{if } sdom(u) = sdom(v) \\ idom(v), & \text{otherwise} \end{cases}$$

只要对上面求解半支配点的代码稍作修改即可。

```
struct E {
  int v, x;
} e[MAX * 4];

int h[3][MAX * 2];
```

```cpp
int dfc, tot, n, m, u, v;
int fa[MAX], fth[MAX], pos[MAX], mn[MAX], idm[MAX], sdm[MAX], dfn[MAX],
    ans[MAX];

void add(int x, int u, int v) {
  e[++tot] = {v, h[x][u]};
  h[x][u] = tot;
}

void dfs(int u) {
  dfn[u] = ++dfc;
  pos[dfc] = u;
  for (int i = h[0][u]; i; i = e[i].x) {
    int v = e[i].v;
    if (!dfn[v]) {
      dfs(v);
      fth[v] = u;
    }
  }
}

int find(int x) {
  if (fa[x] == x) {
    return x;
  }
  int tmp = fa[x];
  fa[x] = find(fa[x]);
  if (dfn[sdm[mn[tmp]]] < dfn[sdm[mn[x]]]) {
    mn[x] = mn[tmp];
  }
  return fa[x];
}

void tar(int st) {
  dfs(st);
  for (int i = 1; i <= n; ++i) {
    fa[i] = sdm[i] = mn[i] = i;
  }
  for (int i = dfc; i >= 2; --i) {
    int u = pos[i], res = INF;
    for (int j = h[1][u]; j; j = e[j].x) {
      int v = e[j].v;
      if (!dfn[v]) {
        continue;
      }
      find(v);
      if (dfn[v] < dfn[u]) {
        res = std::min(res, dfn[v]);
      } else {
        res = std::min(res, dfn[sdm[mn[v]]]);
      }
    }
    sdm[u] = pos[res];
    fa[u] = fth[u];
```

```
    add(2, sdm[u], u);
    u = fth[u];
    for (int j = h[2][u]; j; j = e[j].x) {
      int v = e[j].v;
      find(v);
      if (sdm[mn[v]] == u) {
        idm[v] = u;
      } else {
        idm[v] = mn[v];
      }
    }
    h[2][u] = 0;
  }
  for (int i = 2; i <= dfc; ++i) {
    int u = pos[i];
    if (idm[u] != sdm[u]) {
      idm[u] = idm[idm[u]];
    }
  }
}
```

例题

洛谷 P5180【模板】支配树[5]

可以仅求解支配关系，求解过程中记录各个点支配了多少节点，也可以建出支配树求解每个节点的 size。

这里给出后一种解法的代码。

" 参考代码"

```
#include <bits/stdc++.h>
using std::cin;
using std::cout;
const int MAX = 3e5 + 5;
const int INF = 0x5ffffff;

struct E {
  int v, x;
} e[MAX * 4];

int n, m, u, v, tot, dfc;
int ans[MAX], dfn[MAX], pos[MAX], sdm[MAX], idm[MAX], fa[MAX], mn[MAX],
    fth[MAX];
int h[3][MAX * 2];

void add(int x, int u, int v) {
  e[++tot] = {v, h[x][u]};
  h[x][u] = tot;
}

void dfs(int u) {
  dfn[u] = ++dfc;
  pos[dfc] = u;
```

```
  for (int i = h[0][u]; i; i = e[i].x) {
    int v = e[i].v;
    if (!dfn[v]) {
      dfs(v);
      fth[v] = u;
    }
  }
}

int find(int x) {
  if (fa[x] == x) {
    return x;
  }
  int tmp = fa[x];
  fa[x] = find(fa[x]);
  if (dfn[sdm[mn[tmp]]] < dfn[sdm[mn[x]]]) {
    mn[x] = mn[tmp];
  }
  return fa[x];
}

void tar(int st) {
  dfs(st);
  for (int i = 1; i <= n; ++i) {
    mn[i] = fa[i] = sdm[i] = i;
  }
  for (int i = dfc; i >= 2; --i) {
    int u = pos[i], res = INF;
    for (int j = h[1][u]; j; j = e[j].x) {
      int v = e[j].v;
      if (!dfn[v]) {
        continue;
      }
      find(v);
      if (dfn[v] < dfn[u]) {
        res = std::min(res, dfn[v]);
      } else {
        res = std::min(res, dfn[sdm[mn[v]]]);
      }
    }
    sdm[u] = pos[res];
    fa[u] = fth[u];
    add(2, sdm[u], u);
    u = fth[u];
    for (int j = h[2][u]; j; j = e[j].x) {
      int v = e[j].v;
      find(v);
      if (u == sdm[mn[v]]) {
        idm[v] = u;
      } else {
        idm[v] = mn[v];
      }
    }
    h[2][u] = 0;
```

```
  }
  for (int i = 2; i <= dfc; ++i) {
    int u = pos[i];
    if (idm[u] != sdm[u]) {
      idm[u] = idm[idm[u]];
    }
  }
  for (int i = dfc; i >= 2; --i) {
    ++ans[pos[i]];
    ans[idm[pos[i]]] += ans[pos[i]];
  }
  ++ans[1];
}

int main() {
  cin >> n >> m;
  for (int i = 1; i <= m; ++i) {
    cin >> u >> v;
    add(0, u, v);
    add(1, v, u);
  }
  tar(1);
  for (int i = 1; i <= n; ++i) {
    cout << ans[i] << ' ';
  }
}
```

ZJOI2012 灾难[4]

在 DAG 上求支配树然后求节点 size 即可。

" 参考代码"

```
#include <bits/stdc++.h>

using std::cin;
using std::cout;
using std::stack;
using std::vector;

const int MAX = 65536;
int n, x, tot;
int d[MAX], w[MAX], siz[MAX], p[MAX], f[MAX][17];
vector<int> e[MAX], g[MAX], h[MAX];
stack<int> s;

void topo() {
  s.push(0);
  for (int i = 1; i <= n; ++i) {
    if (!w[i]) {
      e[0].push_back(i);
      g[i].push_back(0);
      ++w[i];
```

```
      }
    }
    while (!s.empty()) {
      int x = s.top();
      s.pop();
      p[++tot] = x;
      for (int i : e[x]) {
        --w[i];
        if (!w[i]) {
          s.push(i);
        }
      }
    }
  }

  int lca(int u, int v) {
    if (d[u] < d[v]) {
      std::swap(u, v);
    }
    for (int i = 15; i >= 0; --i) {
      if (d[f[u][i]] >= d[v]) {
        u = f[u][i];
      }
    }
    if (u == v) {
      return u;
    }
    for (int i = 15; i >= 0; --i) {
      if (f[u][i] != f[v][i]) {
        u = f[u][i];
        v = f[v][i];
      }
    }
    return f[u][0];
  }

  void dfs(int x) {
    siz[x] = 1;
    for (int i : h[x]) {
      dfs(i);
      siz[x] += siz[i];
    }
  }

  void build() {
    for (int i = 2; i <= n + 1; ++i) {
      int x = p[i], y = g[x][0];
      for (int j = 1, q = g[x].size(); j < q; ++j) {
        y = lca(y, g[x][j]);
      }
      h[y].push_back(x);
      d[x] = d[y] + 1;
      f[x][0] = y;
      for (int i = 1; i <= 15; ++i) {
```

```
      f[x][i] = f[f[x][i - 1]][i - 1];
    }
  }
}

int main() {
  cin >> n;
  for (int i = 1; i <= n; ++i) {
    while (true) {
      cin >> x;
      if (!x) {
        break;
      }
      e[x].push_back(i);
      g[i].push_back(x);
      ++w[i];
    }
  }
  topo();
  build();
  dfs(0);
  for (int i = 1; i <= n; ++i) {
    cout << siz[i] - 1 << '\n';
  }
  return 0;
}
```

参考资料与注释

[1] 一篇关于网络流的论文

[2] 有效的求解算法

[3] 一篇论文

[4] ZJOI2012 灾难 [4-1] [4-2]

[5] 洛谷 P5180【模板】支配树

11.36 图上随机游走

本页介绍图上随机游走问题。主要从网格图、稀疏图、一般图三个角度进行探究，介绍了解决这类问题的各种方法，并对比了它们在解决各种问题时的优缺点。

定义

给定一张有向简单图 $G=(V,E)(V=\{v_1,v_2,\cdots,v_{|V|}\})$ 和起点 $s\in V$，终点 $t\in V$，每条边 $e=(x,y)$ 有正权值 w_e，满足 $\forall x\in V\setminus\{t\}$，$\sum_{(x,y)\in E} w_{(x,y)}=1$，且对于任意点 x 都存在一条从 x 出发到达 t 的路径。有一枚棋子从起点出发，每秒从当前所在点 x 以 $w_{(x,y)}$ 的概率选择出边 (x,y) 并走向 y，到达终点则停止，求期望花费时间。

事实上，这个问题也可以写成矩阵的形式。定义矩阵 P：

$$P_{x,y}=\begin{cases}w_{(x,y)} & \text{if } (x,y)\in E \text{ and } x\neq t\\ 0 & \text{if } (x,y)\notin E \text{ or } x=t\end{cases}$$

要求的答案即为：

$$\sum_{k\geq 0} k\times\left(P^k\right)_{s,t}$$

其中 $\left(P^k\right)_{s,t}$ 表示走了 k 步第一次到达终点的概率。当图有限且所有点都能到达终点时，由 P 的定义可以证明其特征值都小于 1，所以答案一定是收敛的。

为了方便描述，在本页中如无特殊说明，均用 n 代指 $|V|$，m 代指 $|E|$。

另外，在本页中，稀疏图指边数和点数同阶的图。

网格图

" 例题 1 Circles of Waiting[1]"

有一枚棋子起始被放在平面直角坐标系的 $(0,0)$ 点。每秒棋子会随机移动。假设它当前在 (x,y)，它下一秒有 p_1 的概率移动到 $(x-1,y)$，p_2 的概率移动到 $(x,y-1)$，p_3 的概率移动到 $(x+1,y)$，p_4 的概率移动到 $(x,y+1)$。保证 $p_1+p_2+p_3+p_4=1$。求期望经过多少时间它会移动到一个离原点的欧几里得距离大于 R 的位置。$0\leq R\leq 50$，$p_1,p_2,p_3,p_4>0$，答案对 10^9+7 取模。

朴素做法

记 $f(i,j)$ 表示在棋子在 (i,j) 时移动到一个离原点的欧几里得距离大于 R 的位置的期望时间，转移方程为：

$$f(i,j)=\begin{cases}p_1f(i-1,j)+p_2f(i,j-1)+p_3f(i+1,j)+p_4f(i,j+1)+1 & i^2+j^2\leq R\\ 0 & i^2+j^2>R\end{cases}$$

由于转移并不存在拓扑序，需要使用高斯消元求解。时间复杂度 $O\left(R^6\right)$，无法通过本题。

直接消元法

注意到需要消元的方程的系数大多数都是 0，消元时只对值非 0 的位置进行计算就可以降低复杂度。

考虑消元的过程，将方程按照在坐标系中从上到下，同一层中从左到右的顺序进行消元，将已经消元过的方程染成黄色，与黄色点相邻的点染成绿色，其余点染成黑色，如下图：

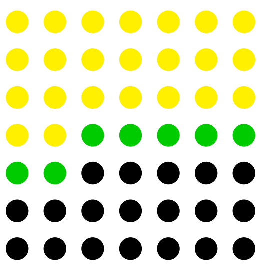


图 11.114 graph-random-walk-1

接下来要对下一个绿色格子对应的方程进行消元。在这个方程中，只有绿色格子和它下方的第一个黑色格子对应的变量系数可能不为 0；而只有绿色格子和它下方的第一个黑色格子对应的方程中，当前格子对应的变量系数可能不为 0。

注意到绿色格子只有 $O(R)$ 个，所以单个方程消元的时间复杂度为 $O\left(R^2\right)$。一共只有 $O\left(R^2\right)$ 个方程，所以时间复杂度降低为 $O\left(R^4\right)$，可以通过本题。

主元法

方程和变量都有 $O(R^2)$ 个，如果能将规模缩小至 $O(R)$，那么朴素的高斯消元就能通过了。

将每行从左到右第一个格子对应的变量设为主元，共 $2R+1$ 个，设法将其他格子对应的变量用关于这些主元的线性函数表示。从左到右逐列考虑，对于当前列的每个格子 (i,j)，注意到 $f(i,j)$ $f(i-1,j)$ $f(i,j-1)$ $f(i,j+1)$ 都是已知的关于主元的线性函数，将转移方程移项，有：

$$f(i+1,j)=\frac{f(i,j)-p_1f(i-1,j)-p_2f(i,j-1)-p_4f(i,j+1)-1}{p_3}$$

这样我们就能得到 $f(i+1,j)$ 关于主元的线性函数表示。如果 $(i+1,j)$ 已经离原点欧几里得距离超过 R 了，则可以得到一个方程：$f(i+1,j)=0$。最终，会得到 $2R+1$ 个方程，对这些方程进行高斯消元即可。

在递推关于主元的线性函数的阶段，共有 $O(R^2)$ 个变量，递推单个变量需要花费 $O(R)$ 的时间；之后，将问题的规模缩减到了 $O(R)$。两部分的时间复杂度均为 $O(R^3)$，总的时间复杂度也为 $O(R^3)$，可以通过本题。

两种做法的对比

下面从多种方面对比两种做法：

从时间复杂度方面，主元法在网格图上的最坏时间复杂度为 $O(n\sqrt{n})$（当网格图长和宽都为 $O(\sqrt{n})$ 级别时时间复杂度最高），直接消元法在网格图上最坏时间复杂度为 $O(n^2)$，主元法较优。

从精度方面，对于一些需要进行实数计算而不是取模的题目，直接消元法的精度优于主元法。

从适用性方面，两种做法适用于不同的方面。

当网格图中存在障碍或者走某些边的概率为 0 时，对于每个障碍或者概率为 0 的边主元法需要增加一个主元，当障碍或者概率为 0 的边的数量多于 $O(R)$ 时，主元法的时间复杂度会增加，而直接消元法的时间复杂度仍然不变。

但主元法还可以做类似于网格图的转移方程的消元，例如 $f(i,j)=p_1f(i+1,j)+p_2f(i,j+1)+p_3f(\mathrm{pre}(i,j))+1$，其中 $\mathrm{pre}(i,j)=(x,y)(x\le i,y\le j)$ 是问题给定的值，而直接消元法的复杂度分析在这种模型中并不适用。

除此以外，网格图邻接矩阵行列式的计算，也不能使用主元法，只能用直接消元法来优化时间复杂度。

综上，两种做法各有所长，需要根据具体题目分析采用不同的做法。

稀疏图

" **例题** 2 Expected Value"

给定一张简单无向连通稀疏图 $G=(V,E)$，有一枚棋子起始被放在 v_1，每秒棋子会从与当前点相连的边中等概率选择一条走到出边指向的点，求到达 v_n 的期望时间。$n\le 2000$，答案对 p 取模，p 是在区间 $[10^9,1.01\times 10^9]$ 内随机生成的一个质数。

基础知识

定义 4.1. 所有满足 $p(A)=0$ 的多项式 $p(\lambda)$ 称为矩阵 A 的零化多项式。

定义 4.2. 记 I_n 表示 n 阶单位矩阵，定义一个 $n\times n$ 矩阵 A 的特征多项式为 $p(\lambda)=\det(\lambda I_n-A)$，其中 det 表示一个矩阵的行列式。不难发现，一个 n 阶矩阵 A 的特征多项式的次数不超过 n。

定理 4.2. (Cayley–Hamilton 定理) 任意矩阵的特征多项式是它的零化多项式。

所以，一个 n 阶矩阵的次数最小的零化多项式的次数也不超过 n。

求解原问题

注意到，期望走的时间 $E(t)=\sum_{i\ge 0}\Pr[t>i]$，如果我们能求出走了 i 步还没有结束的概率，对所有 $i\ge 0$ 求和即为答案。

记 $f(i,j)$ 表示走了 i 步，当前停留在 j，且没有走到过 n 的概率，那么有：

$$f(i,j) = \sum_{(k,j)\in E} \frac{f(i-1,k)}{\deg_k}(j \neq n)$$

其中 $\deg_k$ 表示 k 的度数。

注意到 f 的转移与 i 无关，可以认为一次转移是乘上了一个矩阵，即 $fi+1 = f_i M$。由于 M 的最小零化多项式次数不超过 n，所以 f 的最短递推式长度也不超过 n，故 $\Pr[t > i] = \sum_{j=1}^{n-1} f(i,j)$ 的最短递推式长度也不超过 n。我们可以在 $O(nm)$ 的时间求出 $\Pr[t > 0], \Pr[t > 1], \cdots, \Pr[t > 3n]$，然后使用 *Berlekamp–Massey* 算法，在 $O(n^2)$ 的时间内求解出 $\Pr[t > i]$ 的最短递推式。

考虑求一个 k 阶线性递推序列 a 的生成函数。不妨设 $i \geq i_0$ 时 $a_i = \sum_{j=1}^{k} c_j a_{i-j}$，记 a 和 c 的生成函数为 $A(x)$ 和 $C(x)$，那么 $A(x) = A(x)C(x) + A_0(x)$，其中 $A_0(x)$ 是由 $i < i_0$ 的项决定的。

回到原问题，由于我们能求出 $\Pr[t > i]$ 的最短递推式，则我们可以求出 $C(x)$ 和 $A_0(x)$（定义与上一段相同），移项得 $A(x) = \frac{A_0(x)}{1-C(x)}$。我们要求的是 $\sum_{i\geq 0}[x^i]A(x)$，不难发现这个值就等于 $A(1)$，将 $x = 1$ 带入原问题求解即可。由于模数是随机质数，可以认为分母不会为 0。

这样，我们就在 $O(nm + n^2)$ 的时间复杂度内解决了本题。如果图 G 的点数与边数同阶，本题中时间复杂度可以认为是 $O(n^2)$。

一般图

" 例题 3 Frank"

给定一张简单强连通有向图 $G = (V, E)$，对于所有 $1 \leq s \leq n$，$1 \leq t \leq n$，$s \neq t$。回答下面的问题：有一枚棋子起始被放在 v_s，每秒棋子会从当前点的出边中等概率选择一条走到出边指向的点，求到达 v_t 的期望时间。$3 \leq n \leq 400$。

分析和转化

记 $p_{i,j}$ 表示棋子在 i 时，选择出边 (i,j) 走到 j 的概率，特别地，当出边不存在时概率为 0。记 $f_{i,j}$ 表示 i 随机游走到 j 的期望时间，特别地，$f_{i,i} = 0$。当 $i \neq j$ 时，转移方程为：

$$f_{i,j} = 1 + \sum_{1\leq k\leq n} p_{i,k} f_{k,j}$$

当 $i = j$ 时，记 g_i 表示从 i 开始随机游走，第一次回到 i 的期望时间，那么：

$$f_{i,i} = 1 - g_i + \sum_{1\leq k\leq n} p_{i,k} f_{k,i}$$

为了方便观察，我们将转移方程写成矩阵的形式。记 P 表示这个图的转移矩阵，F 表示答案矩阵，I 表示 n 阶单位矩阵，J 表示 n 阶全 1 矩阵，G 是一个 n 阶矩阵，满足 $G_{i,i} = g_i$，其他位置为 0，则：

$$F = J - G + PF$$

如果我们能求出 G，那么我们只需要解方程：

$$(I - P)F = J - G$$

G 的求法

定义 5.1. 定义一个 n 阶转移矩阵 P 的稳态分布为一个 n 维向量 π，满足 $\sum_{i=1}^{n} \pi_i = 1$，$\pi P = \pi$。其中，π 每一维的值都在区间 $[0,1]$ 内。

我们很容易找出稳态分布的实际意义。如果某个时刻棋子有 π_i 的概率停留在 v_i，则在之后的任意时刻，棋子仍然满足这个概率分布。我们可以在 $O(n^3)$ 的时间通过高斯消元解方程来求出 π，那么 π 与 G 有什么关系呢？

定理 5.1. 对于任意 $1 \leq i \leq n$，有 $\pi_i g_i = 1$。

"证明"

由 $F=J-G+PF$，移项得：

$$G=PF+J-F$$

两边同时在左边乘上 π 有：

$$\pi G=\pi PF+\pi J-\pi F$$

由 π 的定义有 $\pi P=\pi$，故：

$$\pi G=\pi J$$

所以：

$$\pi_i g_i=\sum_{j=1}^{n}\pi_j=1$$

原命题得证。

所以，通过引入稳态分布，我们可以在 $O(n^3)$ 的时间内求解 G。

求解原问题

在解方程的过程中，我们发现一个问题：$(I-P)$ 并不满秩，不能通过乘逆矩阵的方法求解。

定义 5.2. 定义一个有向图 $G=(V,E)$ 的以 $r\in V$ 为根的有向生成树是 G 的一个子图 $T=(V,A)$，满足：

1. 对于任意 $i\neq r$，i 的出度为 1。
2. r 的出度为 0。
3. T 中不存在环。

引理 5.1. (有向图上的矩阵树定理) 对于一个有向图 G，记 D 表示其出度矩阵，即 $D_{i,i}=d_i$，$D_{i,j}=0(i\neq j)$，其中 d_i 表示 i 的出度，记 A 表示其邻接矩阵，则其以 r 为根的有向生成树个数为 $D-A$ 去掉第 r 行第 r 列后的行列式。

定理 5.2. 对于一个强连通图 $G=(V,E)$ 的转移矩阵 P，$(I-P)$ 的秩为 $n-1$。

"证明"

因为对矩阵某一行乘上一个非零常数其秩不改变，所以我们将 $(I-P)$ 的第 i 行乘上 v_i 的出度，得到一个新的矩阵 L，只需证明 L 的秩为 $n-1$ 即可。

由于 L 每行的和均为 0，对 L 的所有列向量求和，会得到零向量，即这些向量线性相关，所以 L 的秩不为 n。

不难发现 L 等于图 G 的出度矩阵减去其邻接矩阵，由引理 5.1 得 L 去掉第 i 行第 i 列后行列式表示以 v_i 为根的有向生成树个数。

由于 G 是强连通的，所以以任意点为根的有向生成树个数均不为 0，即 L 去掉第 i 行第 i 列之后仍然满秩。

因为加上一列秩不会变小，所以 L 去掉第 i 行后所有行向量线性无关。故 L 的秩为 $n-1$。

回到原问题，考虑求解原问题中的方程。为了方便，我们将方程写成 $AX=B$ 的形式，其中 A，B 已知，需要求解 X。由于 A 不满秩，解有无数个，我们首先求出一组特解。将 A 和 B 一起做高斯消元。把 A 的前 $n-1$ 行消成只有主对角线和第 n 列有值的形式，最后一行消成全 0，即下列形式：

$$\begin{bmatrix}1&0&0&\cdots&0&a_1\\0&1&0&\cdots&0&a_2\\0&0&1&\cdots&0&a_3\\\vdots&\vdots&\vdots&\ddots&\vdots&\vdots\\0&0&0&\cdots&1&a_{n-1}\\0&0&0&\cdots&0&0\end{bmatrix}X=\begin{bmatrix}b_{1,1}&b_{1,2}&b_{1,3}&\cdots&b_{1,n-1}&b_{1,n}\\b_{2,1}&b_{2,2}&b_{2,3}&\cdots&b_{2,n-1}&b_{2,n}\\b_{3,1}&b_{3,2}&b_{3,3}&\cdots&b_{3,n-1}&b_{3,n}\\\vdots&\vdots&\vdots&\ddots&\vdots&\vdots\\b_{n-1,1}&b_{n-1,2}&b_{n-1,3}&\cdots&b_{n-1,n-1}&b_{n-1,n}\\0&0&0&\cdots&0&0\end{bmatrix}$$

令 $X_{n,i}=0$，可以解出一组特解，记为 Y。接下来将特解调整为真正的解。

注意到 $X_{n,i}=0$，考虑组合意义有 $Y_{i,j}=1+Y_{j,j}+P_{i,k}X_{k,j}$，不难解出 $X_{i,j}=Y_{i,j}-Y_{j,j}$。最终在 $O(n^3)$ 的时间复杂度内解决了这个问题。

参考

1. 浅谈图模型上的随机游走问题。IOI2019 中国国家候选队论文集（pp. 17-26)

参考资料与注释

[1] Circles of Waiting

第 12 章

计算几何

12.1 计算几何部分简介

利用计算机建立数学模型解决几何问题。

12.2 二维计算几何基础

我们将需要解决的几何问题的范围限制在二维平面内，这样就用到了二维计算几何。

要用电脑解平面几何题？数学好的同学们笑了。

我们并不是用计算机算数学卷子上的几何题去了，而是解决一些更加复杂的几何相关问题。

为了解决复杂且抽象的问题，我们一定要选择合适的研究方法。对于计算机来说，给它看几何图形……

我们可以把要研究的图形放在平面直角坐标系或极坐标系下，这样解决问题就会方便很多。

前置技能

如并不了解：

- 几何基础
- 平面直角坐标系
- 向量（包括向量积）
- 极坐标与极坐标系

请先阅读 向量 和 极坐标。

图形的记录

点

在平面直角坐标系下，点用坐标表示，比如点 $(5,2)$，点 $(-1,0)$ 什么的。

我们记录其横纵坐标值即可。用 `pair` 或开结构体记录均可。

在极坐标系下，用极坐标表示即可。记录其极径与极角。

向量

由于向量的坐标表示与点相同，所以只需要像点一样存向量即可（当然点不是向量）。

在极坐标系下，与点同理。

线

直线与射线

一般在解数学题时，我们用解析式表示一条直线。有一般式 $Ax+By+C=0$，还有斜截式 $y=kx+b$，还有截距式 $\frac{x}{a}+\frac{y}{b}=1\cdots\cdots$用哪种？

这些式子最后都逃不过最后的结果——代入解方程求值。

解方程什么的最讨厌了，有什么好一点的方法吗？

考虑我们只想知道这条直线在哪，它的倾斜程度怎么样。于是用直线上的一个点先大致确定位置，用一个向量表示它的倾斜程度，好了，这条直线确定了。

因此我们记录的是：直线上一点和直线的方向向量。

线段

线段很好记录：只需要记录左右端点即可。

在极坐标系下，记录线是比较麻烦的，因此大多数直线问题都在平面直角坐标系下解决。

多边形

开数组按一定顺序记录多边形的每个顶点即可。

特殊地，如果矩形的各边均与某坐标轴平行的话，我们只记录左下角和右上角的顶点即可。

曲线

一些特殊曲线，如函数图像等一般记录其解析式。对于圆，直接记录其圆心和半径即可。

基本公式

正弦定理

在三角形 $\triangle\mathrm{ABC}$ 中，若角 A,B,C 所对边分别为 a,b,c，则有：

$$\frac{a}{\sin A}=\frac{b}{\sin B}=\frac{c}{\sin C}=2R$$

其中，R 为 $\triangle\mathrm{ABC}$ 的外接圆半径。

余弦定理

在三角形 $\triangle\mathrm{ABC}$ 中，若角 A,B,C 所对边分别为 a,b,c，则有：

$$\begin{aligned}a^2&=b^2+c^2-2bc\cos A\\ b^2&=a^2+c^2-2ac\cos B\\ c^2&=a^2+b^2-2ab\cos C\end{aligned}$$

上述公式的证明略。均为人教版高中数学 A 版必修二内容（旧教材为必修五）。

基本操作

判断一个点在直线的哪边

我们有直线上的一点 P 的直线的方向向量 $\mathbf{v}$，想知道某个点 Q 在直线的哪边。

我们利用向量积的性质，算出 $\overrightarrow{PQ} \times \mathbf{v}$。如果向量积为负，则 Q 在直线上方，如果向量积为 0，则 Q 在直线上，如果向量积为正，则 Q 在直线下方。

可以画一下图，用右手定则感受一下。

快速排斥实验与跨立实验

我们现在想判断两条线段是否相交。

首先特判一些特殊情况。如果两线段平行，自然不能相交。这种情况通过判断线段所在直线的斜率是否相等即可。

当然，如果两线段重合或部分重合，只需要判断是否有三点共线的情况即可。

如果两线段的交点为其中一条线段的端点，仍然判断是否有三点共线的情况即可。

还有些显然不相交的情况，我们口头上称之为「两条线段离着太远了」。可什么是「离着远」，怎么判断它呢?

规定「一条线段的区域」为以这条线段为对角线的，各边均与某一坐标轴平行的矩形所占的区域，那么可以发现，如果两条线段没有公共区域，则这两条线段一定不相交。

比如有以下两条线段:

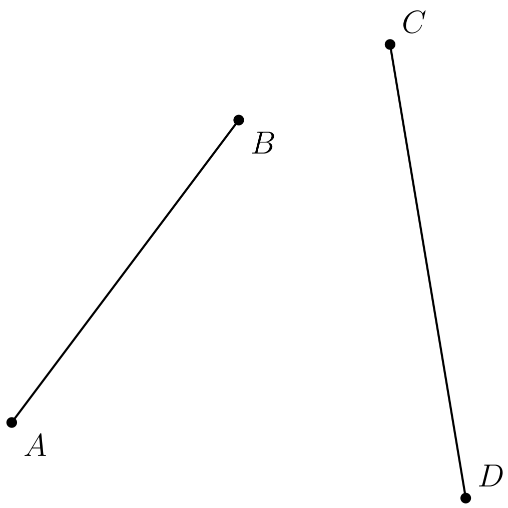


图 12.1 Seg1

它们占用的区域是这样的:

于是可以快速地判断出来这两条线段不相交。

这就是**快速排斥实验**。上述情况称作**未通过快速排斥实验**。

未通过快速排斥实验是两线段无交点的**充分不必要条件**，我们还需要进一步判断。

因为两线段 a,b 相交，b 线段的两个端点一定分布在 a 线段所在直线两侧；同理，a 线段的两个端点一定分布在 b 线段所在直线两侧。我们可以直接判断一条线段的两个端点相对于另一线段所在直线的位置关系，如果不同，则两线段相交，反之则不相交。如上一节所说，直线与点的位置关系我们可以利用向量积判断。

这就是**跨立实验**，如果对于两线段 a,b，b 线段的两个端点分布在 a 线段所在直线的两侧，**且** a 线段的两个端点分布在 b 线段所在直线的两侧，我们就说 a,b 两线段**通过了跨立实验**，即两线段相交。

注意到当两条线段共线但不相交时也可以通过跨立实验，因此想要准确判断还需要与快速排斥实验结合。

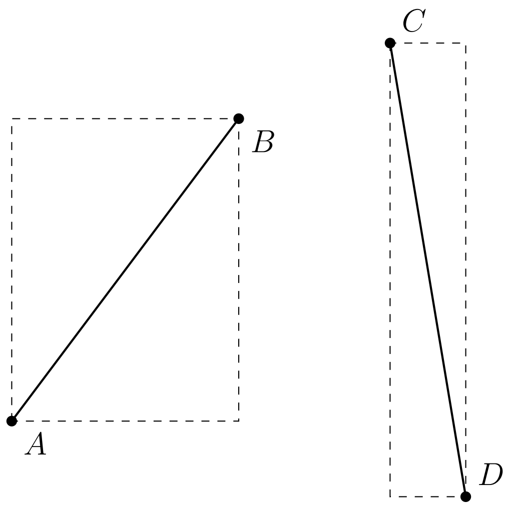


图 12.2 Seg2

判断一点是否在任意多边形内部

在计算几何中，这个问题被称为 PIP 问题[1]，已经有一些成熟的解决方法，下面依次介绍。

光线投射算法 (Ray casting algorithm)

在 这里[2] 可以看到最原始的思路。

我们先特判一些特殊情况，比如「这个点离多边形太远了」。考虑一个能够完全覆盖该多边形的最小矩形，如果这个点不在这个矩形范围内，那么这个点一定不在多边形内。这样的矩形很好求，只需要知道多边形横坐标与纵坐标的最小值和最大值，坐标两两组合成四个点，就是这个矩形的四个顶点了。

还有点在多边形的某一边或某顶点上，这种情况十分容易判断（留作课后作业）。

我们考虑以该点为端点引出一条射线，如果这条射线与多边形有奇数个交点，则该点在多边形内部，否则该点在多边形外部，我们简记为**奇内偶外**。这个算法同样被称为奇偶规则 (Even-odd rule)。

由于 Jordan curve theorem[3]，我们知道，这条射线每次与多边形的一条边相交，就切换一次与多边形的内外关系，所以统计交点数的奇偶即可。

这样的射线怎么取？可以随机取这条射线所在直线的斜率，建议为无理数以避免出现射线与多边形某边重合的情况。

在原版代码中，使用的是记录多边形的数组中最后一个点作为射线上一点，这样统计时，如果出现射线过多边形某边或某顶点时，可以规定射线经过的点同在射线一侧，进而做跨立实验即可。

回转数算法 (Winding number algorithm)

回转数是数学上的概念，是平面内闭合曲线逆时针绕过该点的总次数。很容易发现，当回转数等于 0 的时候，点在曲线外部。这个算法同样被称为非零规则 (Nonzero-rule)。

如何计算呢？我们把该点与多边形的所有顶点连接起来，计算相邻两边夹角的和。注意这里的夹角是**有方向的**。如果夹角和为 0，则这个点在多边形外，否则在多边形内。

求两条直线的交点

首先，我们需要确定两条直线相交，只需判断一下两条直线的方向向量是否平行即可。如果方向向量平行，则两条直线平行，交点个数为 0。进一步地，若两条直线平行且过同一点，则两直线重合。

那么，问题简化为我们有直线 AB,CD 交于一点，想求出交点 E。

如果两直线相交，则交点只有一个，我们记录了直线上的一个点和直线的方向向量，所以我们只需要知道这个点与交点的距离 l，再将这个点沿方向向量平移 l 个单位长度即可。

考虑构造三角形，利用正弦定理求解 l，可以利用向量积构造出正弦定理。

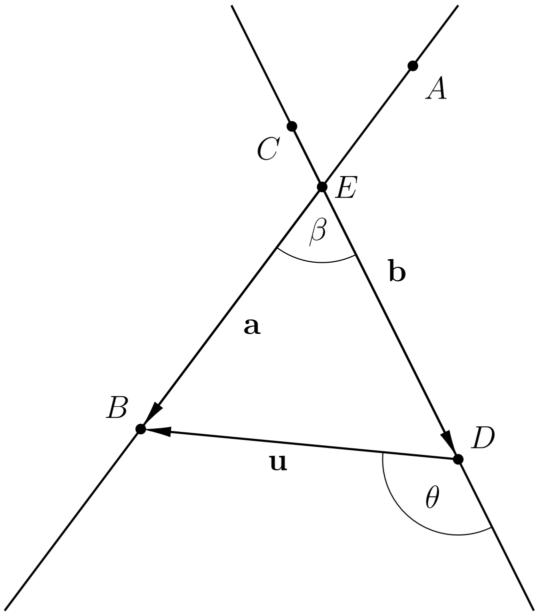


图 12.3 Intersection

由上图可知，$|\mathbf{a}\times\mathbf{b}| = |\mathbf{a}||\mathbf{b}|\sin\beta$，$|\mathbf{u}\times\mathbf{b}| = |\mathbf{u}||\mathbf{b}|\sin\theta$。

作商得：

$$T = \frac{|\mathbf{u}\times\mathbf{b}|}{|\mathbf{a}\times\mathbf{b}|} = \frac{|\mathbf{u}|\sin\theta}{|\mathbf{a}|\sin\beta}$$

可以看出，$\left|\dfrac{|\mathbf{u}|\sin\theta}{\sin\beta}\right| = l$。若绝对值内部式子取值为正，代表沿 $\mathbf{a}$ 方向平移，反之则为反方向。

同时，我们将 T 直接乘上 $\mathbf{a}$，就自动出现了直线的单位向量，不需要进行其他消去操作了。

于是，只需要将点 B 减去 $T\mathbf{a}$ 即可得出交点。

求任意多边形的周长和面积

求任意多边形的周长

直接计算即可，简洁即美德。

求任意多边形的面积

考虑向量积的模的几何意义，我们可以利用向量积完成。

将多边形上的点逆时针标记为 $p_1, p_2, \cdots, p_n$，再任选一个辅助点 O，记向量 $\mathbf{v}_i = p_i - O$，那么这个多边形面积 S 可以表示为：

$$S = \frac{1}{2}\left|\sum_{i=1}^{n} \mathbf{v}_i \times \mathbf{v}_{(i \bmod n)+1}\right|$$

圆与直线相关

求直线与圆的交点

首先判断直线与圆的位置关系。如果直线与圆相离则无交点，若相切则可以利用切线求出切点与半径所在直线，之后转化为求两直线交点。

若有两交点，则可以利用勾股定理求出两交点的中点，然后沿直线方向加上半弦长即可。

求两圆交点

首先我们判断一下两个圆的位置关系，如果外离或内含则无交点，如果相切，可以算出两圆心连线的方向向量，然后利用两圆半径计算出平移距离，最后将圆心沿这个方向向量进行平移即可。

如果两圆相交，则必有两个交点，并且关于两圆心连线对称。因此下面只说明一个交点的求法，另一个交点可以用类似方法求出。

我们先将一圆圆心与交点相连，求出两圆心连线与该连线所成角。这样，将两圆心连线的方向向量旋转这个角度，就是圆心与交点相连形成的半径的方向向量了。

最后还是老套路——沿方向向量方向将圆心平移半径长度。

极角序

一般来说，这类题需要先枚举一个极点，然后计算出其他点的极坐标，在极坐标系下按极角的顺序解决问题。

例题 「JOI Spring Camp 2014 Day4」 两个人的星座[4]

平面内有 n 个点，有三种颜色，每个点的颜色是三种中的一种。求不相交的三色三角形对数。$6 \le n \le 3000$。

题解

如果两个三角形不相交，则一定可以做出两条内公切线，如果相交或内含是做不出内公切线的。三角形的公切线可以类比圆的公切线。

先枚举一个原点，记为 O，以这个点为极点，过这个点且与 x 轴平行的直线作为极轴，建立极坐标系，把剩余点按极角由小到大排序。然后统计出在极轴上方和下方的每种点的个数。

然后根据点枚举公切线，记枚举到的点为 P，初始时公切线为极轴。开始统计。那么一定存在一条公切线过点 O 和点 P。因为公切线与三角形不相交，所以一方选择公切线上方的点，另一方一定选择下方的点。然后利用乘法原理统计方案数即可。

统计完后转公切线，那么点 P 一定改变了相对于公切线的上下位置，而其他点不动，应该只将它的位置信息改变。

这样，可以发现，同一对三角形最终被统计了 4 次，就是同一条公切线会被枚举两次，最后做出的答案应除以 4。

分析一下算法复杂度，我们枚举了一个原点，然后对于每一个原点将剩余点排序后线性统计。于是时间复杂度为 $O(n^2 \log n)$。

代码编写注意事项

由于计算几何经常进行 `double` 类型的浮点数计算，因此带来了精度问题和时间问题。

有些问题，例如求点坐标均为整数的三角形面积，可以利用其特殊性进行纯整数计算，避免用浮点数影响精度。

由于浮点数计算比整数计算慢，所以需要注意程序的常数因子给时间带来的影响。

参考资料与注释

[1] PIP 问题

[2] 这里

[3] Jordan curve theorem

[4] 「JOI Spring Camp 2014 Day4」两个人的星座

12.3 三维计算几何基础

Authors: shuzhouliu

三维几何的很多概念与 二维几何 是相通的，我们可以用与解决二维几何问题相同的方法来解决三维几何问题。

基本概念

点，向量，直线这些概念和二维几何是相似的，这里不再展开。

平面

我们可以用平面上的一点 $P_0(x_0, y_0, z_0)$ 和该平面的法向量（即垂直于该平面的向量）n 来表示一个平面。

因为 n 垂直于平面，所以 n 垂直于该平面内的所有直线。换句话说，设 $n = (A, B, C)$，则该平面上的点 $P(x, y, z)$ 都满足 $n \cdot \overrightarrow{PP_0} = 0$。

根据向量点积的定义，上式等价于：

$$A(x - x_0) + B(y - y_0) + C(z - z_0) = 0$$

整理后得到：

$$Ax + By + Cz - (Ax_0 + By_0 + Cz_0) = 0$$

令 $D = -(Ax_0 + By_0 + Cz_0)$，则上式变成 $Ax + By + Cz + D = 0$。我们称这个式子为平面的**一般式**。

基本操作

直线、平面之间的夹角

运用空间向量的知识，空间中直线、平面之间的夹角可以很快求出。

对于两条异面直线 a，b，过空间中一点 P，作 $a' \parallel a$，$b' \parallel b'$，则 a' 与 b' 所成的锐角或直角被称为 a 和 b 两条**异面直线所成的角**。

对于直线 a 和平面 α，若 a 与 α 相交于 A，过 a 上一点 P 引平面 α 的垂线交 α 于 O，则 a 与 PO 所成的角被称为**直线与平面所成的角**。特别地，若 $a \parallel \alpha$ 或 $a \subset \alpha$，则它们之间的夹角为 $0°$。

对于两个平面 α，β，它们的夹角被定义为与两条平面的交线 l 垂直的两条直线 a, b（其中 $a \subset \alpha$，$b \subset \beta$）所成的角。

两直线夹角定义与关系充要条件

- 两直线的方向向量的夹角，叫做两直线的夹角。

有了这个命题，我们就可以得出以下结论：已知两条直线 l_1, l_2，它们的方向向量分别是 $s_1(m_1, n_1, p_1)$，$s_2(m_2, n_2, p_2)$，设 φ 为两直线夹角，我们可以得到 $\cos\varphi = \dfrac{|m_1m_2 + n_1n_2 + p_1p_2|}{\sqrt{m_1^2 + n_1^2 + p_1^2}\sqrt{m_2^2 + n_2^2 + p_2^2}}$.

- $l_1 \perp l_2 \iff m_1m_2 + n_1n_2 + p_1p_2 = 0$
- $l_1 \parallel l_2 \iff \dfrac{m_1}{m_2} = \dfrac{n_1}{n_2} = \dfrac{p_1}{p_2}$.

三维向量与平面的夹角

当直线与平面不垂直时，直线和它在平面上的投影直线的夹角 φ（$\varphi \in [0, \frac{\pi}{2}]$）称为直线与平面的夹角。

设直线向量 $s(m, n, p)$，平面法线向量 $f(a, b, c)$，那么以下命题成立：

- 角度的正弦值：$\sin\varphi = \dfrac{|am+bn+cp|}{\sqrt{a^2+b^2+c^2}\sqrt{m^2+n^2+p^2}}$
- 直线与平面平行 $\iff am+bn+cp=0$
- 直线与平面垂直 $\iff \dfrac{a}{m}=\dfrac{b}{n}=\dfrac{c}{p}$

点到平面的距离

直线与平面的交点

直接联立直线方程和平面方程即可。

立体几何定理

三正弦定理

设二面角 $M\ AB\ N$ 的度数为 α，在平面 M 上有一条射线 AC，它和棱 AB 所成角为 β，和平面 N 所成的角为 γ，则 $\sin\gamma = \sin\alpha\cdot\sin\beta$。

三余弦定理

设 O 为平面上一点，过平面外一点 B 的直线 BO 在面上的射影为 AO，OC 为面上的一条直线，那么 $\angle COB\ \angle AOC\ \angle AOB$ 三角的余弦关系为：$\cos\angle BOC = \cos\angle AOB\cdot\cos\angle AOC$（$\angle AOC$，$\angle AOB$ 只能是锐角）。

参考资料

- 3D 空间基础概念之一：点、向量（矢量）和齐次坐标[1]

参考资料与注释

[1] 3D 空间基础概念之一：点、向量（矢量）和齐次坐标

12.4 距离

Authors: Chrogeek, frank-xjh, ChungZH, hsfzLZH1, Marcythm, Planet6174, partychicken, i-Yirannn

欧氏距离

二维空间

定义

欧氏距离，一般也称作欧几里得距离。在平面直角坐标系中，设点 A,B 的坐标分别为 $A(x_1,y_1),B(x_2,y_2)$，则两点间的欧氏距离为：

$$|AB| = \sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$$

解释

举个例子，若在平面直角坐标系中，有两点 $A(6,5),B(2,2)$，通过公式，我们很容易得到 A,B 两点间的欧氏距离：

$$|AB| = \sqrt{(2-6)^2+(2-5)^2} = \sqrt{4^2+3^2} = 5$$

除此之外，$P(x,y)$ 到原点的欧氏距离可以用公式表示为：

$$|P| = \sqrt{x^2 + y^2}$$

n 维空间

引入

那么，三维空间中两点的欧氏距离公式呢？我们来观察下图。

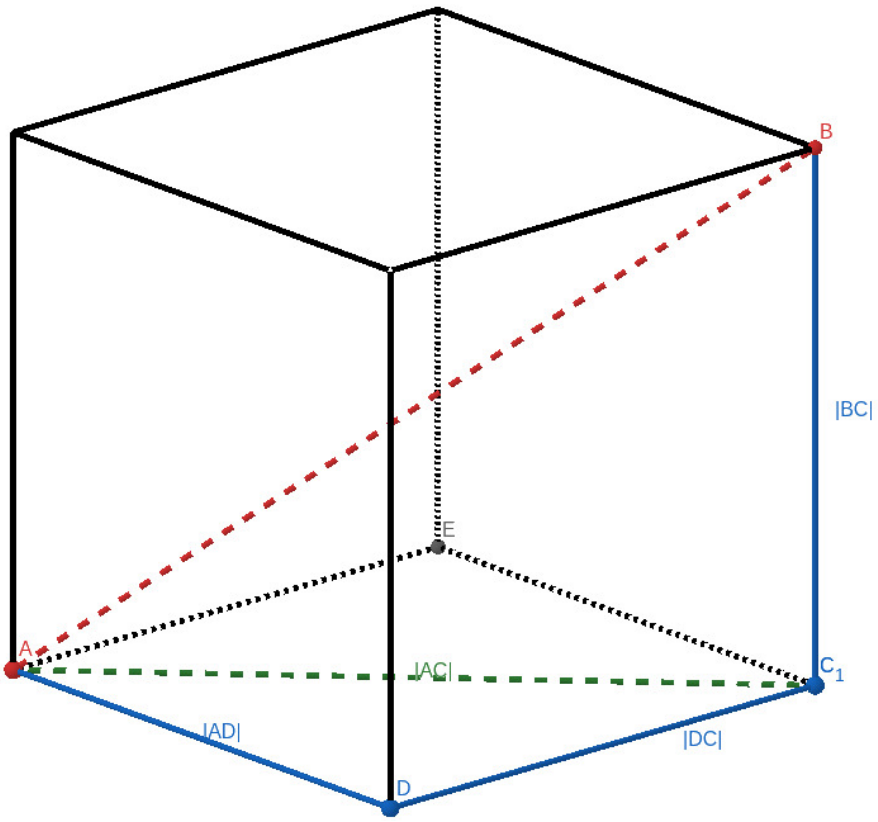


图 12.4 dis-3-dimensional

我们很容易发现，在 $\triangle ADC$ 中，$\angle ADC = 90^\circ$；在 $\triangle ACB$ 中，$\angle ACB = 90^\circ$。

$$\begin{aligned} \therefore |AB| &= \sqrt{|AC|^2 + |BC|^2} \\ &= \sqrt{|AD|^2 + |CD|^2 + |BC|^2} \end{aligned}$$

定义

由此可得，三维空间中欧氏距离的距离公式为：

$$|AB| = \sqrt{{(x_2 - x_1)}^2 + {(y_2 - y_1)}^2 + {(z_2 - z_1)}^2}$$
$$|P| = \sqrt{x^2 + y^2 + z^2}$$

解释

NOIP2017 提高组 奶酪[2] 就运用了这一知识，可以作为欧氏距离的例题。

以此类推，我们就得到了 n 维空间中欧氏距离的距离公式：对于 $\vec{A}(x_{11}, x_{12}, \cdots, x_{1n})$，$\vec{B}(x_{21}, x_{22}, \cdots, x_{2n})$，有

$$\begin{aligned} \|\overrightarrow{AB}\| &= \sqrt{{(x_{11} - x_{21})}^2 + {(x_{12} - x_{22})}^2 + \cdots + {(x_{1n} - x_{2n})}^2} \\ &= \sqrt{\sum_{i=1}^{n} (x_{1i} - x_{2i})^2} \end{aligned}$$

欧氏距离虽然很有用，但也有明显的缺点。两个整点计算其欧氏距离时，往往答案是浮点型，会存在一定误差。

曼哈顿距离

定义

在二维空间内，两个点之间的曼哈顿距离（Manhattan distance）为它们横坐标之差的绝对值与纵坐标之差的绝对值之和。设点 $A(x_1, y_1), B(x_2, y_2)$，则 A, B 之间的曼哈顿距离用公式可以表示为：

$$d(A, B) = |x_1 - x_2| + |y_1 - y_2|$$

解释

观察下图：

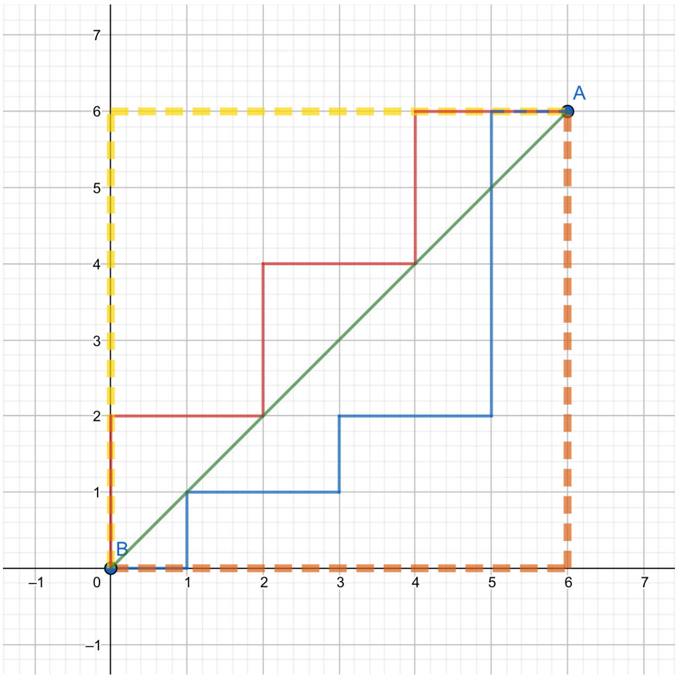


图 12.5 manhattan-dis-diff

在 A, B 间，黄线、橙线都表示曼哈顿距离，而红线、蓝线表示等价的曼哈顿距离，绿线表示欧氏距离。

同样的例子，在下图中 A, B 的坐标分别为 $A(25, 20), B(10, 10)$。

通过公式，我们很容易得到 A, B 两点间的曼哈顿距离：

$$d(A, B) = |20 - 10| + |25 - 10| = 10 + 15 = 25$$

经过推导，我们得到 n 维空间的曼哈顿距离公式为：

$$\begin{aligned} d(A, B) &= |x_1 - y_1| + |x_2 - y_2| + \cdots + |x_n - y_n| \\ &= \sum_{i=1}^{n} |x_i - y_i| \end{aligned}$$

性质

除了公式之外，曼哈顿距离还具有以下数学性质：

- **非负性**

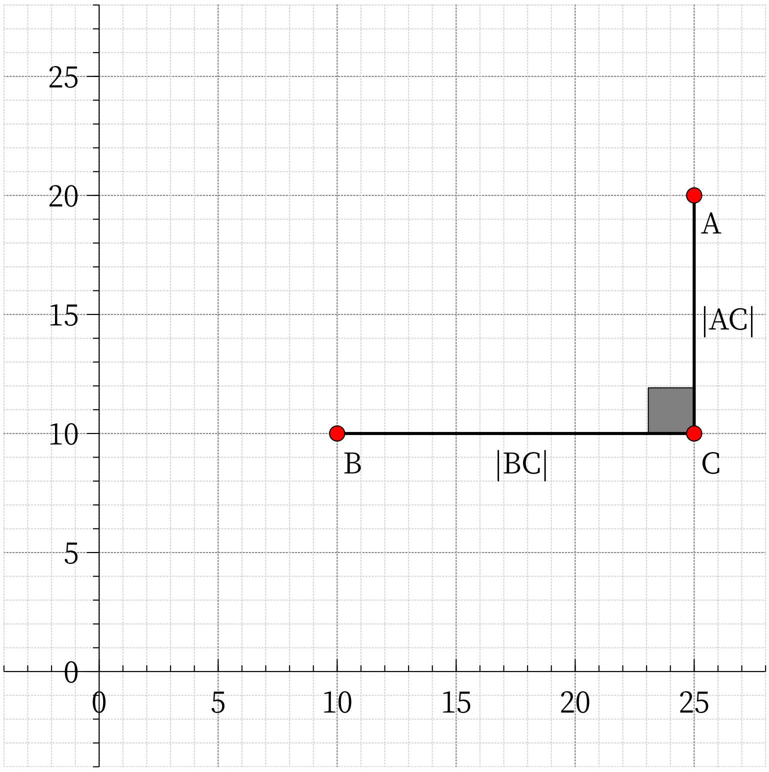


图 12.6 manhattan-dis

曼哈顿距离是一个非负数。

$d(i,j) \geq 0$

- **统一性**

  点到自身的曼哈顿距离为 0。

  $d(i,i) = 0$

- **对称性**

  A 到 B 与 B 到 A 的曼哈顿距离相等，且是对称函数。

  $d(i,j) = d(j,i)$

- **三角不等式**

  从点 i 到 j 的直接距离不会大于途经的任何其它点 k 的距离。

  $d(i,j) \leq d(i,k) + d(k,j)$

例题

P5098「USACO04OPEN」Cave Cows 3[3]

根据题意，对于式子 $|x_1 - x_2| + |y_1 - y_2|$，我们可以假设 $x_1 - x_2 \geq 0$，根据 $y_1 - y_2$ 的符号分成两种情况：

- $(y_1 - y_2 \geq 0) \rightarrow |x_1 - x_2| + |y_1 - y_2| = x_1 + y_1 - (x_2 + y_2)$
- $(y_1 - y_2 < 0) \rightarrow |x_1 - x_2| + |y_1 - y_2| = x_1 - y_1 - (x_2 - y_2)$

只要分别求出 $x + y, x - y$ 的最大值和最小值即能得出答案。

" 参考代码"

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
  int n, x, y, minx = 0x7fffffff, maxx = 0, miny = 0x7fffffff, maxy = 0;
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) {
    scanf("%d%d", &x, &y);
    minx = min(minx, x + y), maxx = max(maxx, x + y);
    miny = min(miny, x - y), maxy = max(maxy, x - y);
  }
  printf("%d\n", max(maxx - minx, maxy - miny));
  return 0;
}
```

```python
minx = 0x7fffffff; maxx = 0; miny = 0x7fffffff; maxy = 0
n = int(input())
for i in range(1, n + 1):
    x, y = map(lambda x:int(x), input().split())
    minx = min(minx, x + y); maxx = max(maxx, x + y)
    miny = min(miny, x - y); maxy = max(maxy, x - y)
print(max(maxx - minx, maxy - miny))
```

其实还有第二种做法，那就是把曼哈顿距离转化为切比雪夫距离求解，最后部分会讲到。

切比雪夫距离

定义

切比雪夫距离（Chebyshev distance）是向量空间中的一种度量，二个点之间的距离定义为其各坐标数值差的最大值。[1]

在二维空间内，两个点之间的切比雪夫距离为它们横坐标之差的绝对值与纵坐标之差的绝对值的最大值。设点 $A(x_1,y_1),B(x_2,y_2)$，则 A,B 之间的切比雪夫距离用公式可以表示为：

$$d(A,B)=\max(|x_1-x_2|,|y_1-y_2|)$$

n 维空间中切比雪夫距离的距离公式可以表示为：

$$\begin{aligned}d(x,y)&=\max\Big\{|x_1-y_1|,|x_2-y_2|,\cdots,|x_n-y_n|\Big\}\\&=\max\Big\{|x_i-y_i|\Big\}\,(i\in[1,n])\end{aligned}$$

解释

仍然是这个例子，下图中 A,B 的坐标分别为 $A(25,20),B(10,10)$。

$$d(A,B)=\max(|20-10|,|25-10|)=\max(10,15)=15$$

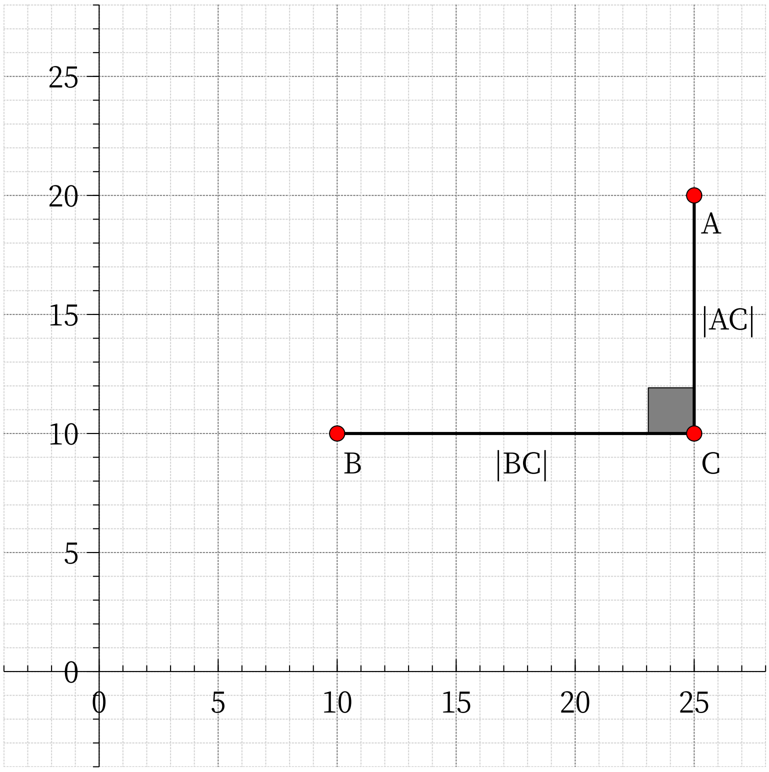


图 12.7 Chebyshev-dis

曼哈顿距离与切比雪夫距离的相互转化

过程

首先，我们考虑画出平面直角坐标系上所有到原点的曼哈顿距离为 1 的点。

通过公式，我们很容易得到方程 $|x|+|y|=1$。

将绝对值展开，得到 4 个一次函数，分别是：

$$\begin{aligned} y &= -x+1 \quad & (x \geq 0, y \geq 0) \\ y &= x+1 \quad & (x \leq 0, y \geq 0) \\ y &= x-1 \quad & (x \geq 0, y \leq 0) \\ y &= -x-1 \quad & (x \leq 0, y \leq 0) \end{aligned}$$

将这 4 个函数画到平面直角坐标系上，得到一个边长为 $\sqrt{2}$ 的正方形，如下图所示：

正方形边界上所有的点到原点的曼哈顿距离都是 1。

同理，我们再考虑画出平面直角坐标系上所有到原点的切比雪夫距离为 1 的点。

通过公式，我们知道 $\max(|x|,|y|)=1$。

我们将式子展开，也同样可以得到可以得到 4 条线段，分别是：

$$\begin{aligned} y &= 1 \quad & (-1 \leq x \leq 1) \\ y &= -1 \quad & (-1 \leq x \leq 1) \\ x &= 1, \quad & (-1 \leq y \leq 1) \\ x &= -1, \quad & (-1 \leq y \leq 1) \end{aligned}$$

画到平面直角坐标系上，可以得到一个边长为 2 的正方形，如下图所示：

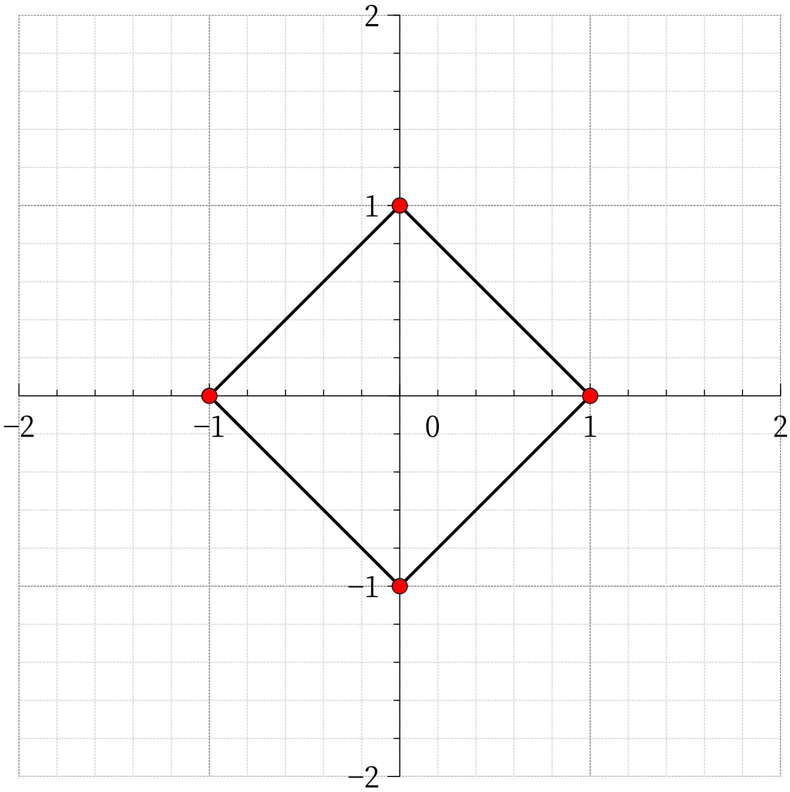


图 12.8 dis-diff-square-1

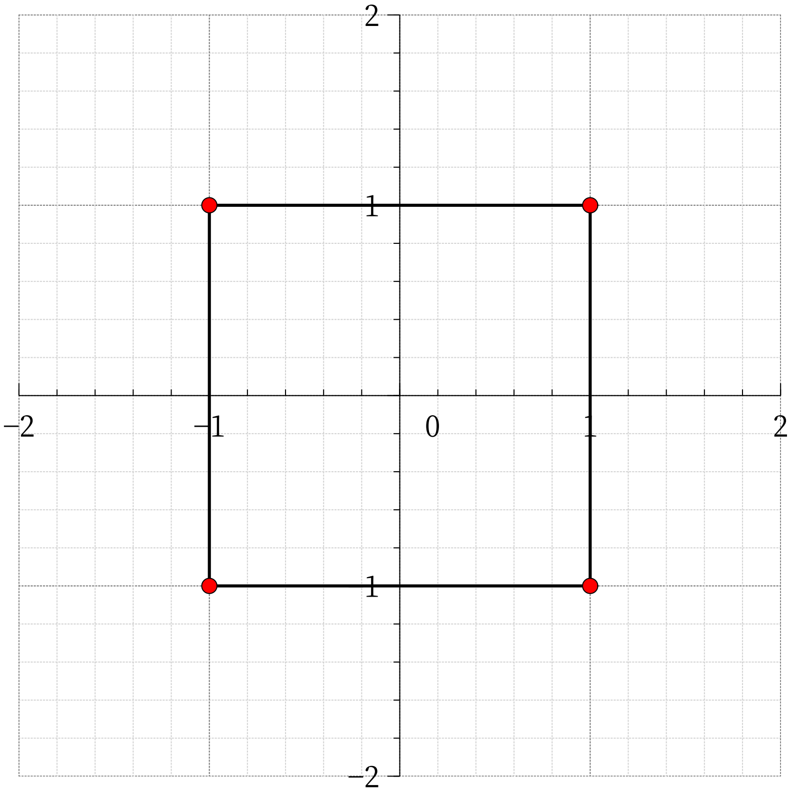


图 12.9 dis-diff-square-2

正方形边界上所有的点到原点的切比雪夫距离都是 1。

将这两幅图对比，我们会神奇地发现：

这 2 个正方形是相似图形。

证明

所以，曼哈顿距离与切比雪夫距离之间会不会有联系呢？

接下来我们简略证明一下：

假设 $A(x_1,y_1),B(x_2,y_2)$，

我们把曼哈顿距离中的绝对值拆开，能够得到四个值，这四个值中的最大值是两个非负数之和，即曼哈顿距离。则 A,B 两点的曼哈顿距离为：

$$\begin{aligned}d(A,B)&=|x_1-x_2|+|y_1-y_2|\\&=\max\left\{x_1-x_2+y_1-y_2,x_1-x_2+y_2-y_1,x_2-x_1+y_1-y_2,x_2-x_1+y_2-y_1\right\}\\&=\max(|(x_1+y_1)-(x_2+y_2)|,|(x_1-y_1)-(x_2-y_2)|)\end{aligned}$$

我们很容易发现，这就是 $(x_1+y_1,x_1-y_1),(x_2+y_2,x_2-y_2)$ 两点之间的切比雪夫距离。

所以将每一个点 (x,y) 转化为 $(x+y,x-y)$，新坐标系下的切比雪夫距离即为原坐标系下的曼哈顿距离。

同理，A,B 两点的切比雪夫距离为：

$$\begin{aligned}d(A,B)&=\max\left\{|x_1-x_2|,|y_1-y_2|\right\}\\&=\max\left\{\left|\frac{x_1+y_1}{2}-\frac{x_2+y_2}{2}\right|+\left|\frac{x_1-y_1}{2}-\frac{x_2-y_2}{2}\right|\right\}\end{aligned}$$

而这就是 $(\frac{x_1+y_1}{2},\frac{x_1-y_1}{2}),(\frac{x_2+y_2}{2},\frac{x_2-y_2}{2})$ 两点之间的曼哈顿距离。

所以将每一个点 (x,y) 转化为 $(\frac{x+y}{2},\frac{x-y}{2})$，新坐标系下的曼哈顿距离即为原坐标系下的切比雪夫距离。

结论

- 曼哈顿坐标系是通过切比雪夫坐标系旋转 45° 后，再缩小到原来的一半得到的。
- 将一个点 (x,y) 的坐标变为 $(x+y,x-y)$ 后，原坐标系中的曼哈顿距离等于新坐标系中的切比雪夫距离。
- 将一个点 (x,y) 的坐标变为 $(\frac{x+y}{2},\frac{x-y}{2})$ 后，原坐标系中的切比雪夫距离等于新坐标系中的曼哈顿距离。

碰到求切比雪夫距离或曼哈顿距离的题目时，我们往往可以相互转化来求解。两种距离在不同的题目中有不同的优缺点，应该灵活运用。

例题

P4648「IOI2007」pairs 动物对数[4]（曼哈顿距离转切比雪夫距离）

P3964「TJOI2013」松鼠聚会[5]（切比雪夫距离转曼哈顿距离）

最后给出 P5098「USACO04OPEN」Cave Cows 3[3] 的第二种解法：

我们考虑将题目所求的曼哈顿距离转化为切比雪夫距离，即把每个点的坐标 (x,y) 变为 $(x+y,x-y)$。

所求的答案就变为 $\max\limits_{i,j\in n}\left\{\max\left\{|x_i-x_j|,|y_i-y_j|\right\}\right\}$。

现要使得横坐标之差和纵坐标之差最大，只需要预处理出 x,y 的最大值和最小值即可。

" 参考代码"

```
#include <bits/stdc++.h>
using namespace std;
```

```cpp
int main() {
  int n, x, y, a, b, minx = 0x7fffffff, maxx = 0, miny = 0x7fffffff, maxy = 0;
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) {
    scanf("%d%d", &a, &b);
    x = a + b, y = a - b;
    minx = min(minx, x), maxx = max(maxx, x);
    miny = min(miny, y), maxy = max(maxy, y);
  }
  printf("%d\n", max(maxx - minx, maxy - miny));
  return 0;
}
```

```python
minx = 0x7fffffff; maxx = 0; miny = 0x7fffffff; maxy = 0
n = int(input())
for i in range(1, n + 1):
    a, b = map(lambda x:int(x), input().split())
    x = a + b; y = a - b
    minx = min(minx, x); maxx = max(maxx, x)
    miny = min(miny, y); maxy = max(maxy, y)
print(max(maxx - minx, maxy - miny))
```

对比两份代码，我们又能够发现，两种不同的思路，写出来的代码却是完全等价的，是不是很神奇呢？当然，更高深的东西需要大家另行研究。

闵可夫斯基距离

我们定义 n 维空间中两点 $X(x_1, x_2, \dots, x_n)$，$Y(y_1, y_2, \dots, y_n)$ 之间的闵可夫斯基距离为：

$$D(X,Y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{\frac{1}{p}}.$$

特别的：

1. 当 $p = 1$ 时，$D(X,Y) = \sum_{i=1}^{n} |x_i - y_i|$ 即为曼哈顿距离；
2. 当 $p = 2$ 时，$D(X,Y) = \left(\sum_{i=1}^{n} (x_i - y_i)^2\right)^{1/2}$ 即为欧几里得距离；
3. 当 $p \to \infty$ 时，$D(X,Y) = \lim_{p\to\infty} \left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{1/p} = \max_{i=1}^{n} |x_i - y_i|$ 即为切比雪夫距离。

注意：当 $p \geq 1$ 时，闵可夫斯基距离才是度量，具体证明参见 Minkowski distance - Wikipedia[6]。

汉明距离

汉明距离是两个字符串之间的距离，它表示两个长度相同的字符串对应位字符不同的数量

我们可以简单的认为对两个串进行异或运算，结果为 1 的数量就是两个串的汉明距离。

参考资料与链接

1. 浅谈三种常见的距离算法[7]，感谢作者 xuxing 的授权。

[1] 切比雪夫距离 - 维基百科，自由的百科全书

[2] NOIP2017 提高组奶酪

[3] P5098「USACO04OPEN」Cave Cows 3 [3-1] [3-2]

[4] P4648「IOI2007」pairs 动物对数

[5] P3964「TJOI2013」松鼠聚会

[6] Minkowski distance - Wikipedia

[7] 浅谈三种常见的距离算法

12.5 Pick 定理

Pick 定理

Pick 定理：给定顶点均为整点的简单多边形，皮克定理说明了其面积 A 和内部格点数目 i、边上格点数目 b 的关系：$A = i + \frac{b}{2} - 1$。

具体证明：Pick's theorem[1]

它有以下推广：

- 取格点的组成图形的面积为一单位。在平行四边形格点，皮克定理依然成立。套用于任意三角形格点，皮克定理则是 $A = 2 \times i + b - 2$。
- 对于非简单的多边形 P，皮克定理 $A = i + \frac{b}{2} - \chi(P)$，其中 $\chi(P)$ 表示 P 的**欧拉特征数**。
- 高维推广：Ehrhart 多项式
- 皮克定理和**欧拉公式**（$V - E + F = 2$）等价。

一道例题 (POJ 1265[2])

题目大意

在直角坐标系中，一个机器人从任意点出发进行 n 次移动，每次向右移动 dx，向上移动 dy，最后会形成一个平面上的封闭简单多边形，求边上的点的数量，多边形内的点的数量，多边形面积。

题解

这道题目其实用了以下三个知识：

- 以整点为顶点的线段，如果边 dx 和 dy 都不为 0，经过的格点数是 $\gcd(dx, dy) + 1$，当然，如果要算一整个图形的，多加的点会被上一条边计算，也就不需要加了。那么一条边覆盖的点的个数为 $\gcd(dx, dy)$，其中，dx, dy 分别为线段横向占的点数和纵向占的点数。如果 dx 或 dy 为 0，则覆盖的点数为 dy **或** dx。
- Pick 定理：平面上以整点为顶点的简单多边形的面积 = 边上的点数/2 + 内部的点数 + 1。
- 任意一个多边形的面积等于按顺序求相邻两个点与原点组成的向量的叉积之和的一半（这个也可以通过顺时针定积分求得）。

" 参考代码"

```
#include <cmath>
#include <cstdio>
#include <iostream>
using namespace std;
const int MAXN = 110;

struct node {
  int x, y;
} p[MAXN];

int gcd(int x, int y) { return y == 0 ? x : gcd(y, x % y); }  // 求最大公约数

int area(int a, int b) { return p[a].x * p[b].y - p[a].y * p[b].x; }  // 求区域

int main() {
  int t, ncase = 1;
  scanf("%d", &t);
  while (t--) {
    int n, dx, dy, x, y, num = 0, sum = 0;
    scanf("%d", &n);
    p[0].x = 0, p[0].y = 0;
    for (int i = 1; i <= n; i++) {
      scanf("%d%d", &x, &y);
      p[i].x = x + p[i - 1].x, p[i].y = y + p[i - 1].y;
      dx = x, dy = y;
      if (x < 0) dx = -x;
      if (y < 0) dy = -y;
      num += gcd(dx, dy);
      sum += area(i - 1, i);
    }
    if (sum < 0) sum = -sum;
    printf("Scenario #%d:\n", ncase++);
    printf("%d %d %.1f\n\n", (sum - num + 2) >> 1, num, sum * 0.5);
  }
  return 0;
}
```

参考资料与注释

[1] Pick's theorem

[2] POJ 1265

12.6 三角剖分

Authors: xehoth

在几何中，三角剖分是指将平面对象细分为三角形，并且通过扩展将高维几何对象细分为单纯形。对于一个给定的点集，有很多种三角剖分，如：

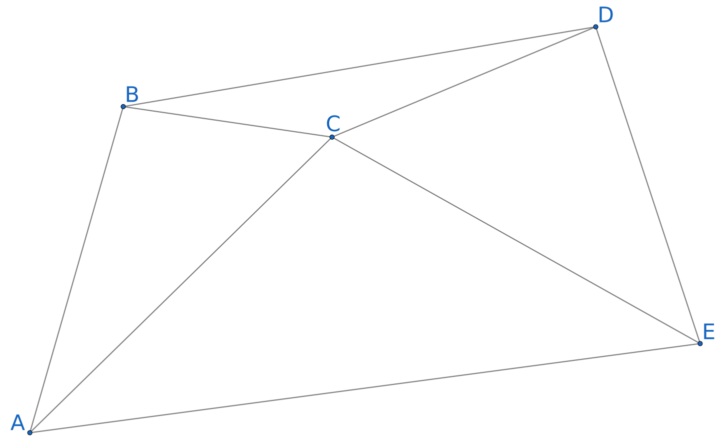


图 12.10 三种三角剖分

OI 中的三角剖分主要指二维几何中的完美三角剖分（二维 Delaunay 三角剖分，简称 DT）。

Delaunay 三角剖分

定义

在数学和计算几何中，对于给定的平面中的离散点集 P，其 Delaunay 三角剖分 $\mathrm{DT}(P)$ 满足：

1. 空圆性：$\mathrm{DT}(P)$ 是**唯一**的（任意四点不能共圆），在 $\mathrm{DT}(P)$ 中，**任意**三角形的外接圆范围内不会有其它点存在。
2. 最大化最小角：在点集 P 可能形成的三角剖分中，$\mathrm{DT}(P)$ 所形成的三角形的最小角最大。从这个意义上讲，$\mathrm{DT}(P)$ 是**最接近于规则化**的三角剖分。具体的说是在两个相邻的三角形构成凸四边形的对角线，在相互交换后，两个内角的最小角不再增大。

性质

1. 最接近：以最接近的三点形成三角形，且各线段（三角形的边）皆不相交。
2. 唯一性：不论从区域何处开始构建，最终都将得到一致的结果（点集中任意四点不能共圆）。
3. 最优性：任意两个相邻三角形构成的凸四边形的对角线如果可以互换的话，那么两个三角形六个内角中最小角度不会变化。
4. 最规则：如果将三角剖分中的每个三角形的最小角进行升序排列，则 Delaunay 三角剖分的排列得到的数值最大。
5. 区域性：新增、删除、移动某一个顶点只会影响邻近的三角形。
6. 具有凸边形的外壳：三角剖分最外层的边界形成一个凸多边形的外壳。

构造 DT 的分治算法

DT 有很多种构造算法，在 $O(n\log n)$ 的构造算法中，分治算法是最易于理解和实现的。

分治构造 DT 的第一步是将给定点集按照 x 坐标**升序**排列，如下图是排好序的大小为 10 的点集。

一旦点集有序，我们就可以不断地将其分成两个部分（分治），直到子点集大小不超过 3。然后这些子点集可以立刻剖分为一个三角形或线段。

然后在分治回溯的过程中，已经剖分好的左右子点集可以依次合并。合并后的剖分包含 LL-edge（左侧子点集的边）。RR-edge（右侧子点集的边），LR-edge（连接左右剖分产生的新的边），如图 LL-edge（灰色），RR-edge（红色），

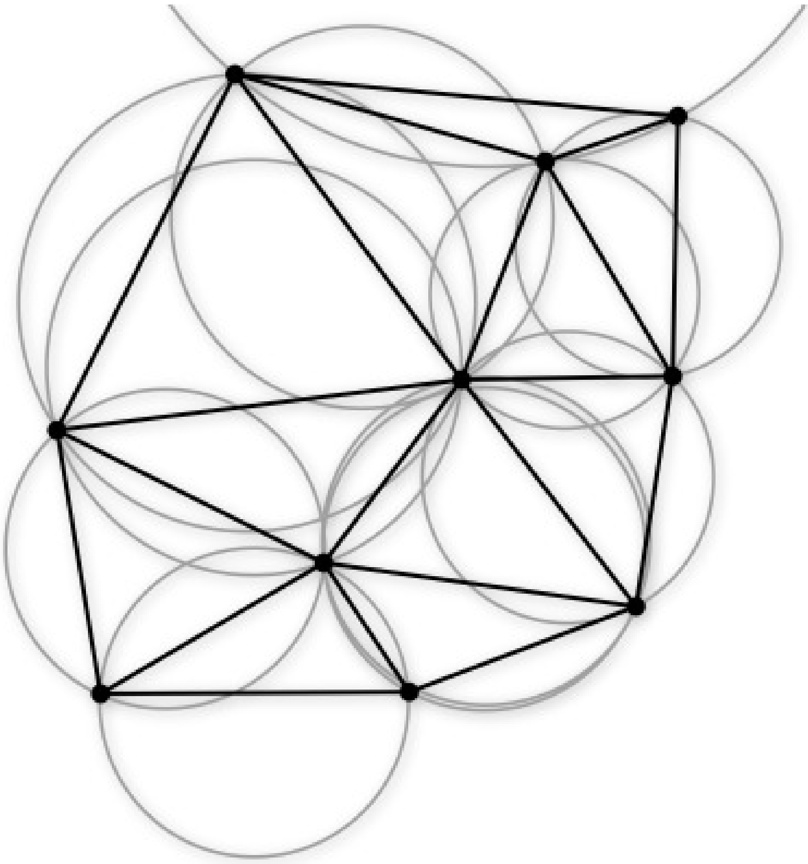


图 12.11 一个显示了外接圆的 Delaunay 三角剖分

图 12.12 排好序的大小为 10 的点集

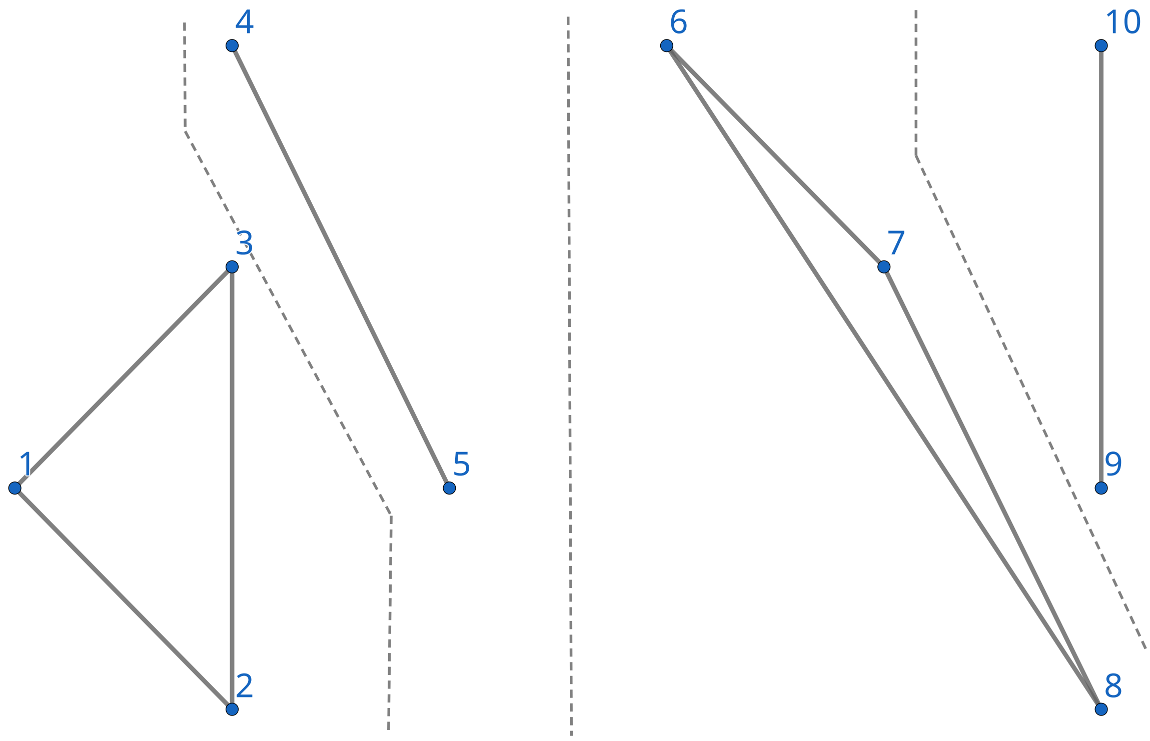


图 12.13 分治为包含 2 或 3 个点的点集

LR-edge（蓝色）。对于合并后的剖分，为了维持 DT 性质，我们**可能**需要删除部分 LL-edge 和 RR-edge，但我们在合并时**不会**增加 LL-edge 和 RR-edge。

合并左右两个剖分的第一步是插入 base LR-edge，base LR-edge 是**最底部**的不与**任何** LL-edge 及 RR-edge 相交的 LR-edge。

然后，我们需要确定下一条**紧接在** base LR-edge 之上的 LR-edge。比如对于右侧点集，下一条 LR-edge 的可能端点（右端点）为与 base LR-edge 右端点相连的 RR-edge 的另一端点（6, 7, 9 号点），左端点即为 2 号点。

对于可能的端点，我们需要按以下两个标准检验：

1. 其对应 RR-edge 与 base LR-edge 的夹角小于 180 度。
2. base LR-edge 两端点和这个可能点三点构成的圆内不包含任何其它**可能点**。

如上图，6 号可能点所对应的绿色圆包含了 9 号可能点，而 7 号可能点对应的紫色圆则不包含任何其它可能点，故 7 号点为下一条 LR-edge 的右端点。

对于左侧点集，我们做镜像处理即可。

当左右点集都不再含有符合标准的可能点时，合并即完成。当一个可能点符合标准，一条 LR-edge 就需要被添加，对于与需要添加的 LR-edge 相交的 LL-edge 和 RR-edge，将其删除。

当左右点集均存在可能点时，判断左边点所对应圆是否包含右边点，若包含则不符合；对于右边点也是同样的判断。一般只有一个可能点符合标准（除非四点共圆）。

当这条 LR-edge 添加好后，将其作为 base LR-edge 重复以上步骤，继续添加下一条，直到合并完成。

代码

" 实现"

```
#include <algorithm>
```

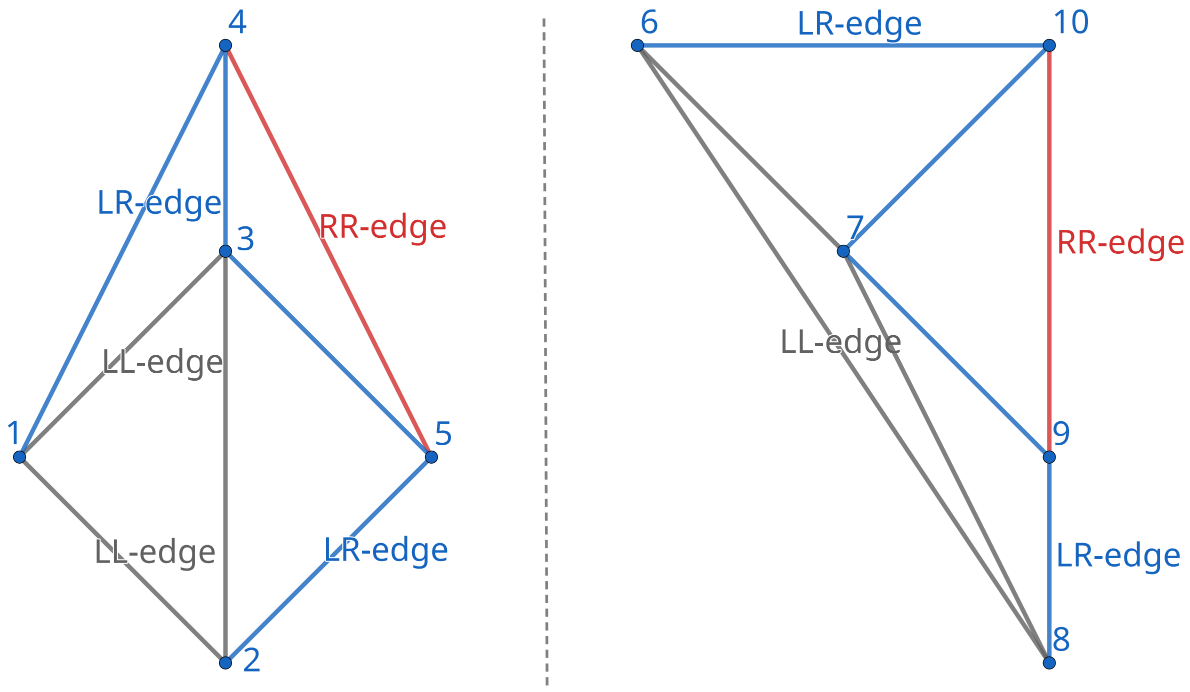


图 12.14 edge

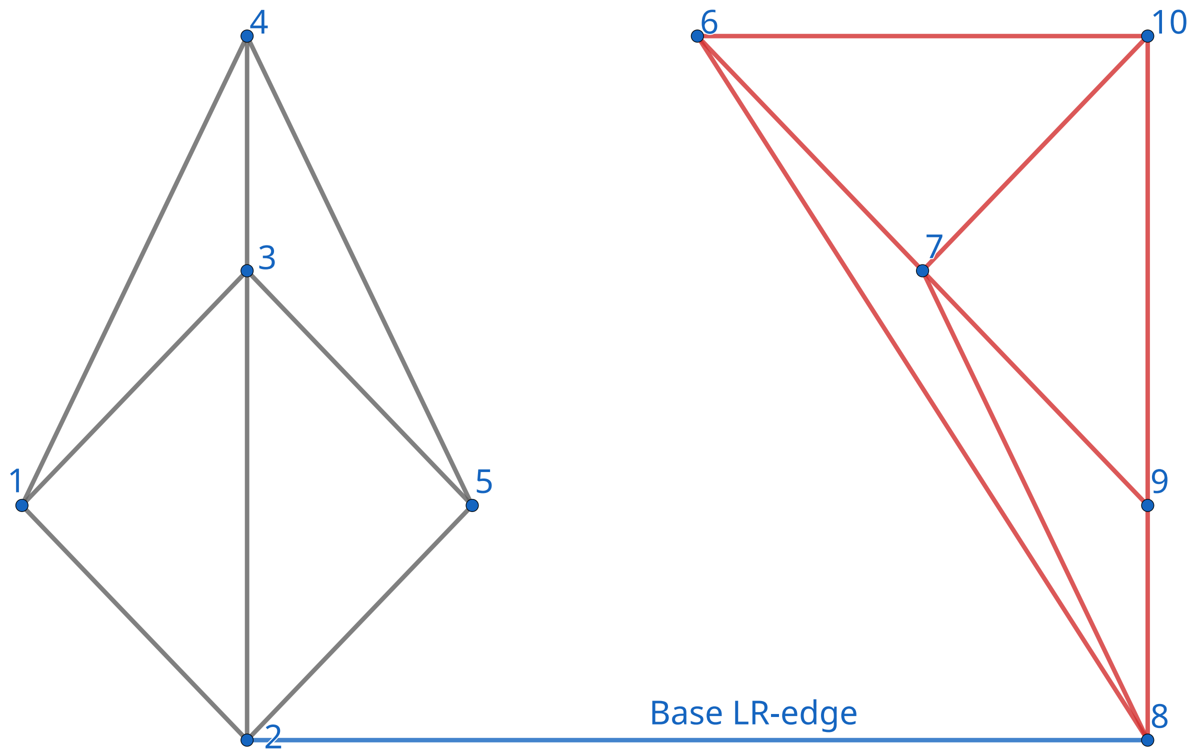


图 12.15 合并左右剖分

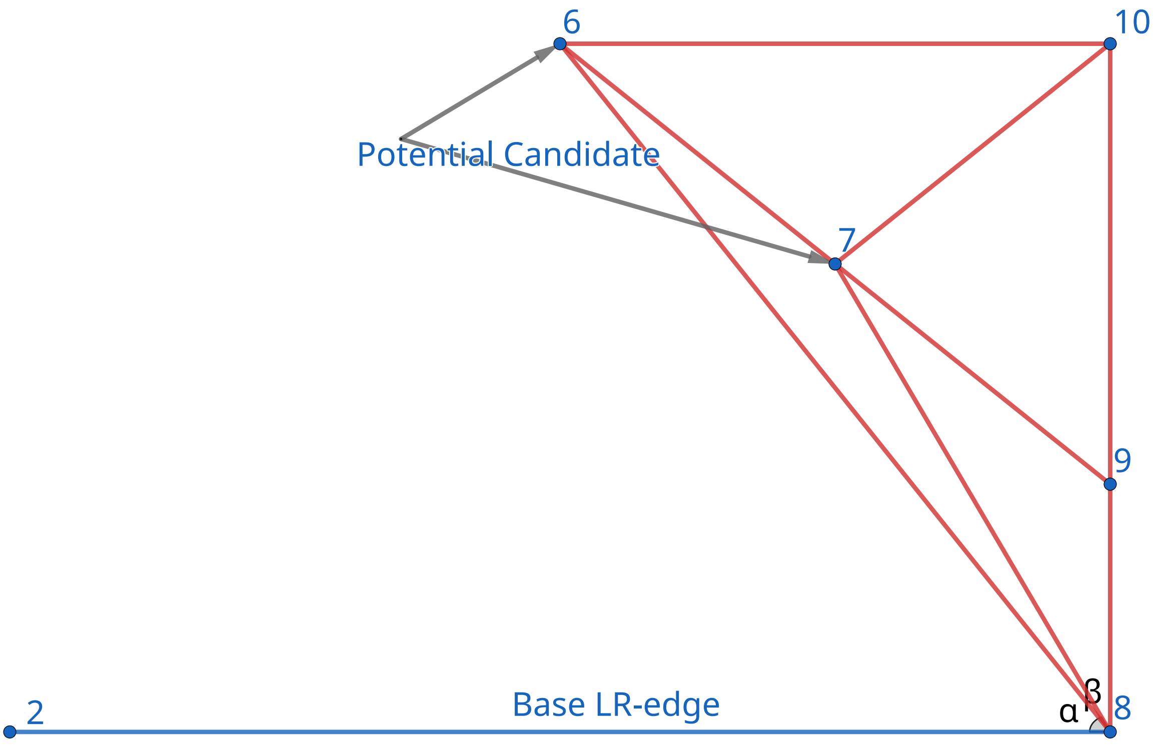


图 12.16 下一条 LR-edge

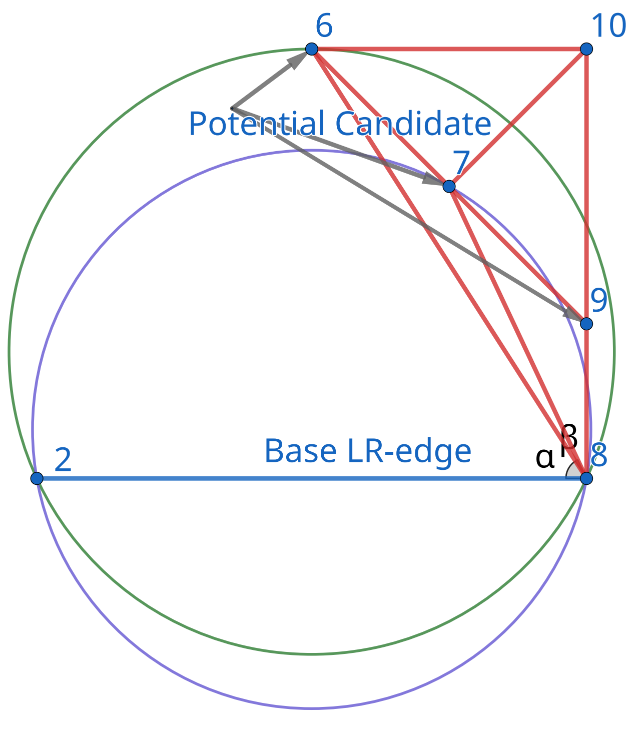


图 12.17 检验可能点

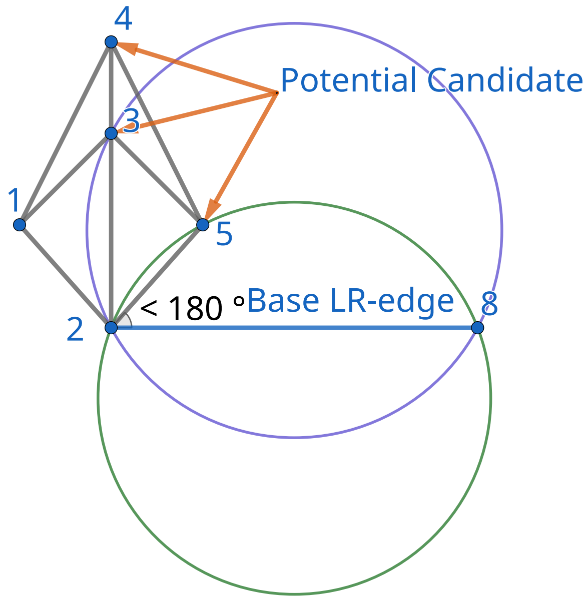


图 12.18 检验左侧可能点

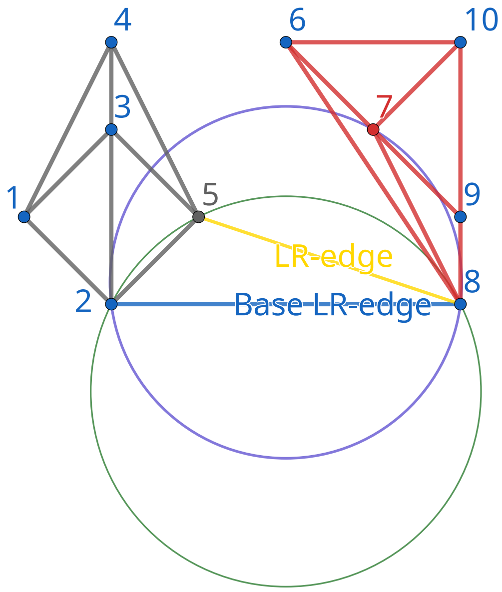


图 12.19 下一条 LR-edge

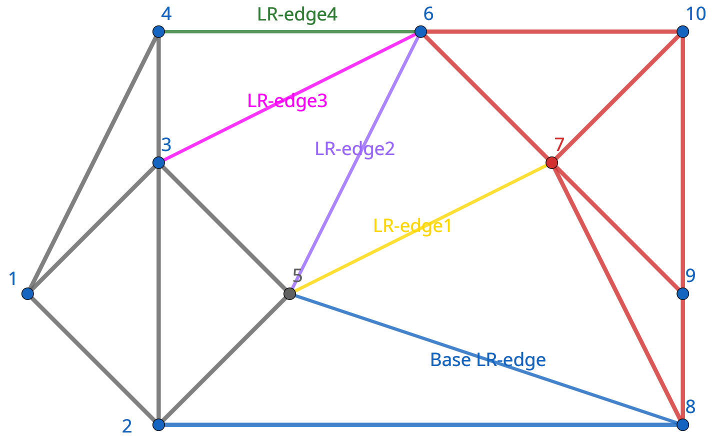


图 12.20 合并

```
#include <cmath>
#include <cstring>
#include <list>
#include <utility>
#include <vector>

const double EPS = 1e-8;
const int MAXV = 10000;

struct Point {
  double x, y;
  int id;

  Point(double a = 0, double b = 0, int c = -1) : x(a), y(b), id(c) {}

  bool operator<(const Point &a) const {
    return x < a.x || (fabs(x - a.x) < EPS && y < a.y);
  }

  bool operator==(const Point &a) const {
    return fabs(x - a.x) < EPS && fabs(y - a.y) < EPS;
  }

  double dist2(const Point &b) {
    return (x - b.x) * (x - b.x) + (y - b.y) * (y - b.y);
  }
};

struct Point3D {
  double x, y, z;

  Point3D(double a = 0, double b = 0, double c = 0) : x(a), y(b), z(c) {}
```

```
  Point3D(const Point &p) { x = p.x, y = p.y, z = p.x * p.x + p.y * p.y; }

  Point3D operator-(const Point3D &a) const {
    return Point3D(x - a.x, y - a.y, z - a.z);
  }

  double dot(const Point3D &a) { return x * a.x + y * a.y + z * a.z; }
};

struct Edge {
  int id;
  std::list<Edge>::iterator c;

  Edge(int id = 0) { this->id = id; }
};

int cmp(double v) { return fabs(v) > EPS ? (v > 0 ? 1 : -1) : 0; }

double cross(const Point &o, const Point &a, const Point &b) {
  return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
}

Point3D cross(const Point3D &a, const Point3D &b) {
  return Point3D(a.y * b.z - a.z * b.y, -a.x * b.z + a.z * b.x,
                 a.x * b.y - a.y * b.x);
}

int inCircle(const Point &a, Point b, Point c, const Point &p) {
  if (cross(a, b, c) < 0) std::swap(b, c);
  Point3D a3(a), b3(b), c3(c), p3(p);
  b3 = b3 - a3, c3 = c3 - a3, p3 = p3 - a3;
  Point3D f = cross(b3, c3);
  return cmp(p3.dot(f));  // check same direction, in: < 0, on: = 0, out: > 0
}

int intersection(const Point &a, const Point &b, const Point &c,
                 const Point &d) {  // seg(a, b) and seg(c, d)
  return cmp(cross(a, c, b)) * cmp(cross(a, b, d)) > 0 &&
         cmp(cross(c, a, d)) * cmp(cross(c, d, b)) > 0;
}

class Delaunay {
 public:
  std::list<Edge> head[MAXV];  // graph
  Point p[MAXV];
  int n, rename[MAXV];

  void init(int n, Point p[]) {
    memcpy(this->p, p, sizeof(Point) * n);
    std::sort(this->p, this->p + n);
    for (int i = 0; i < n; i++) rename[p[i].id] = i;
    this->n = n;
    divide(0, n - 1);
```

```
  }

  void addEdge(int u, int v) {
    head[u].push_front(Edge(v));
    head[v].push_front(Edge(u));
    head[u].begin()->c = head[v].begin();
    head[v].begin()->c = head[u].begin();
  }

  void divide(int l, int r) {
    if (r - l <= 2) {  // #point <= 3
      for (int i = l; i <= r; i++)
        for (int j = i + 1; j <= r; j++) addEdge(i, j);
      return;
    }
    int mid = (l + r) / 2;
    divide(l, mid);
    divide(mid + 1, r);

    std::list<Edge>::iterator it;
    int nowl = l, nowr = r;

    for (int update = 1; update;) {
      // find left and right convex, lower common tangent
      update = 0;
      Point ptL = p[nowl], ptR = p[nowr];
      for (it = head[nowl].begin(); it != head[nowl].end(); it++) {
        Point t = p[it->id];
        double v = cross(ptR, ptL, t);
        if (cmp(v) > 0 || (cmp(v) == 0 && ptR.dist2(t) < ptR.dist2(ptL))) {
          nowl = it->id, update = 1;
          break;
        }
      }
      if (update) continue;
      for (it = head[nowr].begin(); it != head[nowr].end(); it++) {
        Point t = p[it->id];
        double v = cross(ptL, ptR, t);
        if (cmp(v) < 0 || (cmp(v) == 0 && ptL.dist2(t) < ptL.dist2(ptR))) {
          nowr = it->id, update = 1;
          break;
        }
      }
    }

    addEdge(nowl, nowr);  // add tangent

    for (int update = 1; true;) {
      update = 0;
      Point ptL = p[nowl], ptR = p[nowr];
      int ch = -1, side = 0;
      for (it = head[nowl].begin(); it != head[nowl].end(); it++) {
        if (cmp(cross(ptL, ptR, p[it->id])) > 0 &&
            (ch == -1 || inCircle(ptL, ptR, p[ch], p[it->id]) < 0)) {
```

```
          ch = it->id, side = -1;
        }
      }
      for (it = head[nowr].begin(); it != head[nowr].end(); it++) {
        if (cmp(cross(ptR, p[it->id], ptL)) > 0 &&
            (ch == -1 || inCircle(ptL, ptR, p[ch], p[it->id]) < 0)) {
          ch = it->id, side = 1;
        }
      }
      if (ch == -1) break;  // upper common tangent
      if (side == -1) {
        for (it = head[nowl].begin(); it != head[nowl].end();) {
          if (intersection(ptL, p[it->id], ptR, p[ch])) {
            head[it->id].erase(it->c);
            head[nowl].erase(it++);
          } else {
            it++;
          }
        }
        nowl = ch;
        addEdge(nowl, nowr);
      } else {
        for (it = head[nowr].begin(); it != head[nowr].end();) {
          if (intersection(ptR, p[it->id], ptL, p[ch])) {
            head[it->id].erase(it->c);
            head[nowr].erase(it++);
          } else {
            it++;
          }
        }
        nowr = ch;
        addEdge(nowl, nowr);
      }
    }
  }

  std::vector<std::pair<int, int> > getEdge() {
    std::vector<std::pair<int, int> > ret;
    ret.reserve(n);
    std::list<Edge>::iterator it;
    for (int i = 0; i < n; i++) {
      for (it = head[i].begin(); it != head[i].end(); it++) {
        if (it->id < i) continue;
        ret.push_back(std::make_pair(p[i].id, p[it->id].id));
      }
    }
    return ret;
  }
};
```

Voronoi 图

Voronoi 图由一组由连接两邻点直线的垂直平分线组成的连续多边形组成，根据 n 个在平面上不重合种子点，把平面分成 n 个区域，使得每个区域内的点到它所在区域的种子点的距离比到其它区域种子点的距离近。

Voronoi 图是 Delaunay 三角剖分的对偶图，可以使用构造 Delaunay 三角剖分的分治算法求出三角网，再使用最

左转线算法求出其对偶图实现在 $O(n \log n)$ 的时间复杂度下构造 Voronoi 图。

题目

SGU 383 Caravans[1] 三角剖分 + 倍增

ContestHunter. 无尽的毁灭[2] 三角剖分求对偶图建 Voronoi 图

Codeforces Gym 103485M. Constellation collection[3] 三角剖分之后建图进行 Floodfill

参考资料与拓展阅读

1. Wikipedia - Triangulation (geometry)[4]
2. Wikipedia - Delaunay triangulation[5]
3. Samuel Peterson -Computing Constrained Delaunay Triangulations in 2-D (1997-98)[6]

参考资料与注释

[1] SGU 383 Caravans

[2] ContestHunter. 无尽的毁灭

[3] Codeforces Gym 103485M. Constellation collection

[4] Wikipedia - Triangulation (geometry)

[5] Wikipedia - Delaunay triangulation

[6] Computing Constrained Delaunay Triangulations in 2-D (1997-98)

12.7 凸包

二维凸包

定义

凸多边形

凸多边形是指所有内角大小都在 $[0, \pi]$ 范围内的**简单多边形**。

凸包

在平面上能包含所有给定点的最小凸多边形叫做凸包。

其定义为：对于给定集合 X，所有包含 X 的凸集的交集 S 被称为 X 的**凸包**。

实际上可以理解为用一个橡皮筋包含住所有给定点的形态。

凸包用最小的周长围住了给定的所有点。如果一个凹多边形围住了所有的点，它的周长一定不是最小，如下图。根据三角不等式，凸多边形在周长上一定是最优的。

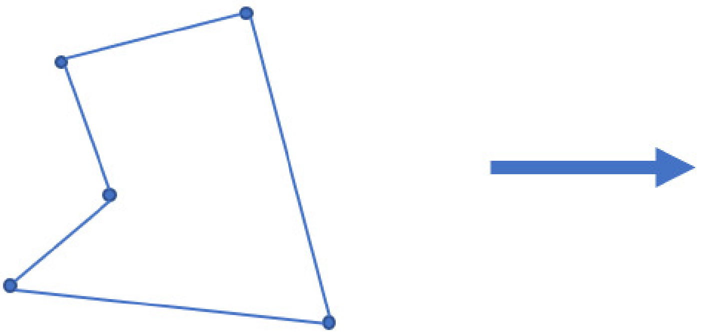


图 12.21

Andrew 算法求凸包

常用的求法有 Graham 扫描法和 Andrew 算法，这里主要介绍 Andrew 算法。

性质

该算法的时间复杂度为 $O(n \log n)$，其中 n 为待求凸包点集的大小，复杂度的瓶颈在于对所有点坐标的双关键字排序。

过程

首先把所有点以横坐标为第一关键字，纵坐标为第二关键字排序。

显然排序后最小的元素和最大的元素一定在凸包上。而且因为是凸多边形，我们如果从一个点出发逆时针走，轨迹总是「左拐」的，一旦出现右拐，就说明这一段不在凸包上。因此我们可以用一个单调栈来维护上下凸壳。

因为从左向右看，上下凸壳所旋转的方向不同，为了让单调栈起作用，我们首先**升序枚举**求出下凸壳，然后**降序**求出上凸壳。

求凸壳时，一旦发现即将进栈的点（P）和栈顶的两个点（S_1, S_2，其中 S_1 为栈顶）行进的方向向右旋转，即叉积小于 0：$\overrightarrow{S_2S_1} \times \overrightarrow{S_1P} < 0$，则弹出栈顶，回到上一步，继续检测，直到 $\overrightarrow{S_2S_1} \times \overrightarrow{S_1P} \geq 0$ 或者栈内仅剩一个元素为止。

通常情况下不需要保留位于凸包边上的点，因此上面一段中 $\overrightarrow{S_2S_1} \times \overrightarrow{S_1P} < 0$ 这个条件中的「$<$」可以视情况改为 $\leq$，同时后面一个条件应改为 $>$。

实现

"代码实现"

```
// stk[] 是整型，存的是下标
// p[] 存储向量或点
tp = 0;                       // 初始化栈
std::sort(p + 1, p + 1 + n);  // 对点进行排序
stk[++tp] = 1;
// 栈内添加第一个元素，且不更新 used，使得 1 在最后封闭凸包时也对单调栈更新
for (int i = 2; i <= n; ++i) {
  while (tp >= 2  // 下一行 * 操作符被重载为叉积
         && (p[stk[tp]] - p[stk[tp - 1]]) * (p[i] - p[stk[tp]]) <= 0)
    used[stk[tp--]] = 0;
  used[i] = 1;  // used 表示在凸壳上
  stk[++tp] = i;
```

```
}
int tmp = tp;  // tmp 表示下凸壳大小
for (int i = n - 1; i > 0; --i)
  if (!used[i]) {
    //   求上凸壳时不影响下凸壳
    while (tp > tmp && (p[stk[tp]] - p[stk[tp - 1]]) * (p[i] - p[stk[tp]]) <= 0)
      used[stk[tp--]] = 0;
    used[i] = 1;
    stk[++tp] = i;
  }
for (int i = 1; i <= tp; ++i)  // 复制到新数组中去
  h[i] = p[stk[i]];
int ans = tp - 1;
```

```
stk = [] # 是整型，存的是下标
p = [] # 存储向量或点
tp = 0 # 初始化栈
p.sort() # 对点进行排序
tp = tp + 1
stk[tp] = 1
# 栈内添加第一个元素，且不更新 used，使得 1 在最后封闭凸包时也对单调栈更新
for i in range(2, n + 1):
    while tp >= 2 and (p[stk[tp]] - p[stk[tp - 1]]) * (p[i] - p[stk[tp]]) <= 0:
        # 下一行 * 操作符被重载为叉积
        used[stk[tp]] = 0
        tp = tp - 1
    used[i] = 1 # used 表示在凸壳上
    tp = tp + 1
    stk[tp] = i
tmp = tp # tmp 表示下凸壳大小
for i in range(n - 1, 0, -1):
    if used[i] == False:
        #        求上凸壳时不影响下凸壳
        while tp > tmp and (p[stk[tp]] - p[stk[tp - 1]]) * (p[i] - p[stk[tp]]) <
= 0:
            used[stk[tp]] = 0
            tp = tp - 1
        used[i] = 1
        tp = tp + 1
        stk[tp] = i
for i in range(1, tp + 1):
    h[i] = p[stk[i]]
ans = tp - 1
```

根据上面的代码，最后凸包上有 ans 个元素（额外存储了 1 号点，因此 h 数组中有 $ans+1$ 个元素），并且按逆时针方向排序。周长就是

$$\sum_{i=1}^{ans} \left|\overrightarrow{h_i h_{i+1}}\right|$$

Graham 扫描法

性质

与 Andrew 算法相同，Graham 扫描法的时间复杂度为 $O(n \log n)$，复杂度瓶颈也在于对所有点排序。

过程

首先找到所有点中，纵坐标最小的一个点 P。根据凸包的定义我们知道，这个点一定在凸包上。然后将所有的点以相对于点 P 的极角大小为关键字进行排序。

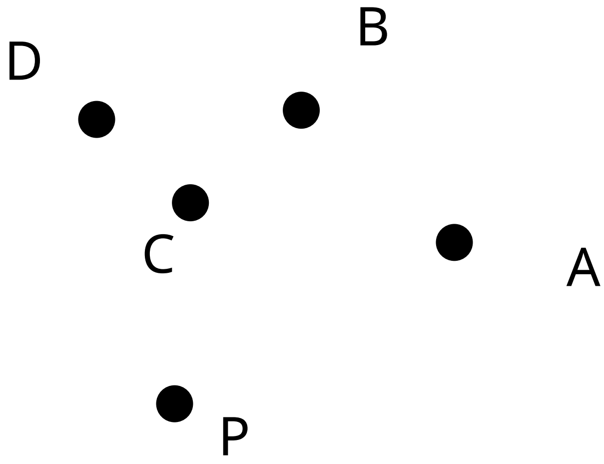


图 12.22

和 Andrew 算法类似地，我们考虑从点 P 出发，在凸包上逆时针走，那么我们经过的所有节点一定都是「左拐」的。形式化地说，对于凸包逆时针方向上任意连续经过的三个点 P_1, P_2, P_3，一定满足 $\overrightarrow{P_1 P_2} \times \overrightarrow{P_2 P_3} \geq 0$。

新建一个栈用于存储凸包的信息，先将 P 压入栈中，然后按照极角序依次尝试加入每一个点。如果进栈的点 P_0 和栈顶的两个点 P_1, P_2（其中 P_1 为栈顶）行进的方向「右拐」了，那么就弹出栈顶的 P_1，不断重复上述过程直至进栈的点与栈顶的两个点满足条件，或者栈中仅剩下一个元素，再将 P_0 压入栈中。

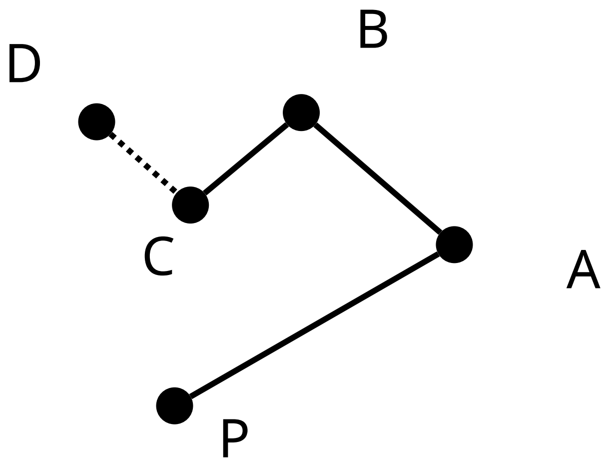


图 12.23

" 代码实现"

```
struct Point {
  double x, y, ang;

  Point operator-(const Point& p) const { return {x - p.x, y - p.y, 0}; }
} p[MAX];

double dis(Point p1, Point p2) {
  return sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y));
}
```

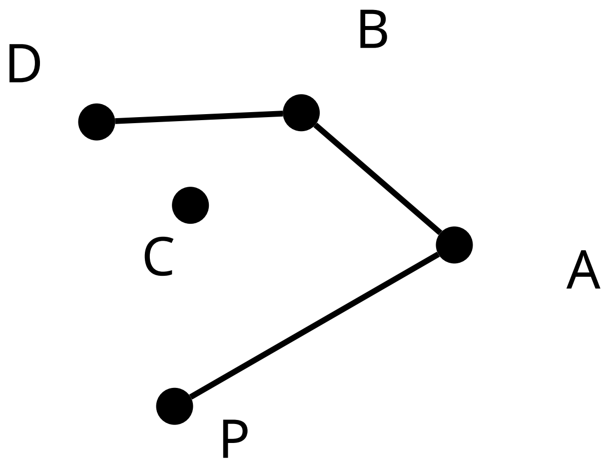


图 12.24

```
bool cmp(Point p1, Point p2) {
  if (p1.ang == p2.ang) {
    return dis(p1, p[1]) < dis(p2, p[1]);
  }
  return p1.ang < p2.ang;
}

double cross(Point p1, Point p2) { return p1.x * p2.y - p1.y * p2.x; }

int main() {
  for (int i = 2; i <= n; ++i) {
    if (p[i].y < p[1].y || (p[i].y == p[1].y && p[i].x < p[1].x)) {
      std::swap(p[1], p[i]);
    }
  }
  for (int i = 2; i <= n; ++i) {
    p[i].ang = atan2(p[i].y - p[1].y, p[i].x - p[1].x);
  }
  std::sort(p + 2, p + n + 1, cmp);
  sta[++top] = 1;
  for (int i = 2; i <= n; ++i) {
    while (top >= 2 &&
           cross(p[sta[top]] - p[sta[top - 1]], p[i] - p[sta[top]]) < 0) {
      top--;
    }
    sta[++top] = i;
  }
  return 0;
}
```

三维凸包

基础知识

> 圆的反演：反演中心为 O，反演半径为 R，若经过 O 的直线经过 P,P'，且 $OP \times OP' = R^2$，则称 P、P' 关于 O 互为反演。

过程

求凸包的过程如下：

- 首先对其微小扰动，避免出现四点共面的情况。
- 对于一个已知凸包，新增一个点 P，将 P 视作一个点光源，向凸包做射线，可以知道，光线的可见面和不可见面一定是由若干条棱隔开的。
- 将光的可见面删去，并新增由其分割棱与 P 构成的平面。重复此过程即可，由 Pick 定理、欧拉公式（在凸多面体中，其顶点 V、边数 E 及面数 F 满足 $V-E+F=2$）和圆的反演，复杂度 $O(n^2)$。[1]

模板题

P4724【模板】三维凸包[2]

重复上述过程即可得到答案。

" 代码实现"

```
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;
const int N = 2010;
const double eps = 1e-9;
int n, cnt, vis[N][N];
double ans;

double Rand() { return rand() / (double)RAND_MAX; }

double reps() { return (Rand() - 0.5) * eps; }

struct Node {
  double x, y, z;

  void shake() {
    x += reps();
    y += reps();
    z += reps();
  }

  double len() { return sqrt(x * x + y * y + z * z); }

  Node operator-(Node A) { return {x - A.x, y - A.y, z - A.z}; }

  Node operator*(Node A) {
    return {y * A.z - z * A.y, z * A.x - x * A.z, x * A.y - y * A.x};
  }

  double operator&(Node A) { return x * A.x + y * A.y + z * A.z; }
} A[N];

struct Face {
  int v[3];
```

```
  Node Normal() { return (A[v[1]] - A[v[0]]) * (A[v[2]] - A[v[0]]); }

  double area() { return Normal().len() / 2.0; }
} f[N], C[N];

int see(Face a, Node b) { return ((b - A[a.v[0]]) & a.Normal()) > 0; }

void Convex_3D() {
  f[++cnt] = {1, 2, 3};
  f[++cnt] = {3, 2, 1};

  for (int i = 4, cc = 0; i <= n; i++) {
    for (int j = 1, v; j <= cnt; j++) {
      if (!(v = see(f[j], A[i]))) C[++cc] = f[j];

      for (int k = 0; k < 3; k++) vis[f[j].v[k]][f[j].v[(k + 1) % 3]] = v;
    }

    for (int j = 1; j <= cnt; j++)
      for (int k = 0; k < 3; k++) {
        int x = f[j].v[k], y = f[j].v[(k + 1) % 3];

        if (vis[x][y] && !vis[y][x]) C[++cc] = {x, y, i};
      }

    for (int j = 1; j <= cc; j++) f[j] = C[j];

    cnt = cc;
    cc = 0;
  }
}

int main() {
  cin >> n;

  for (int i = 1; i <= n; i++) cin >> A[i].x >> A[i].y >> A[i].z, A[i].shake();

  Convex_3D();

  for (int i = 1; i <= cnt; i++) ans += f[i].area();

  printf("%.3f\n", ans);
  return 0;
}
```

练习

- UVa11626 Convex Hull[3]
- 「USACO5.1」 圈奶牛 Fencing the Cows[4]
- POJ1873 The Fortified Forest[5]
- POJ1113 Wall[6]
- 「SHOI2012」 信用卡凸包[7]

参考资料与注释

[1] 三维凸包学习小记

[2] P4724【模板】三维凸包

[3] UVa11626 Convex Hull

[4] 「USACO5.1」圈奶牛 Fencing the Cows

[5] POJ1873 The Fortified Forest

[6] POJ1113 Wall

[7] 「SHOI2012」信用卡凸包

12.8 扫描线

引入

扫描线一般运用在图形上面，它和它的字面意思十分相似，就是一条线在整个图上扫来扫去，它一般被用来解决图形面积，周长，以及二维数点等问题。

Atlantis 问题

题意

在二维坐标系上，给出多个矩形的左下以及右上坐标，求出所有矩形构成的图形的面积。

解法

根据图片可知总面积可以直接暴力即可求出面积，如果数据大了怎么办？这时就需要讲到**扫描线**算法。

过程

现在假设我们有一根线，从下往上开始扫描：

- 如图所示，我们可以把整个矩形分成如图各个颜色不同的小矩形，那么这个小矩形的高就是我们扫过的距离，那么剩下了一个变量，那就是矩形的长一直在变化。
- 我们的线段树就是为了维护矩形的长，我们给每一个矩形的上下边进行标记，下面的边标记为 1，上面的边标记为 -1，每遇到一个矩形时，我们知道了标记为 1 的边，我们就加进来这一条矩形的长，等到扫描到 -1 时，证明这一条边需要删除，就删去，利用 1 和 -1 可以轻松的到这种状态。
- 还要注意这里的线段树指的并不是线段的一个端点，而指的是一个区间，所以我们要计算的是 $r+1$ 和 $r-1$。
- 需要 离散化。

实现

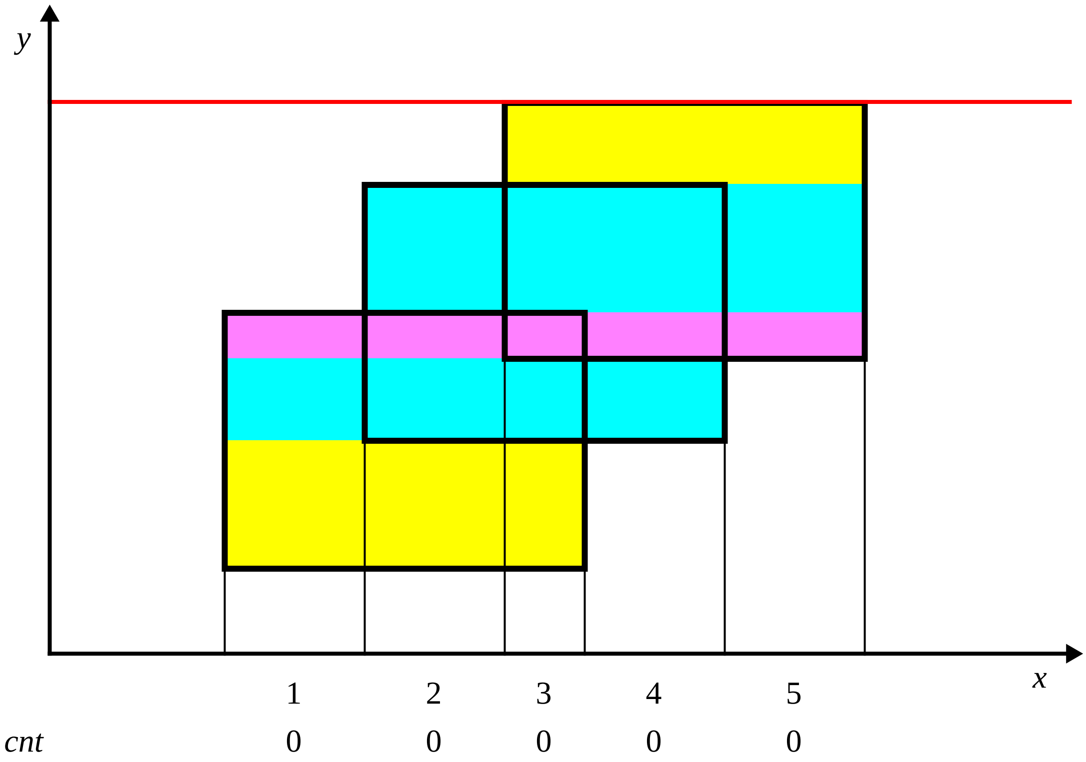


图 12.25

" 代码实现"

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#define maxn 300
using namespace std;

int lazy[maxn << 3];  // 标记了这条线段出现的次数
double s[maxn << 3];

struct node1 {
  double l, r;
  double sum;
} cl[maxn << 3];  // 线段树

struct node2 {
  double x, y1, y2;
  int flag;
} p[maxn << 3];  // 坐标

// 定义 sort 比较
bool cmp(node2 a, node2 b) { return a.x < b.x; }

// 上传
void pushup(int rt) {
```

```
  if (lazy[rt] > 0)
    cl[rt].sum = cl[rt].r - cl[rt].l;
  else
    cl[rt].sum = cl[rt * 2].sum + cl[rt * 2 + 1].sum;
}

// 建树
void build(int rt, int l, int r) {
  if (r - l > 1) {
    cl[rt].l = s[l];
    cl[rt].r = s[r];
    build(rt * 2, l, (l + r) / 2);
    build(rt * 2 + 1, (l + r) / 2, r);
    pushup(rt);
  } else {
    cl[rt].l = s[l];
    cl[rt].r = s[r];
    cl[rt].sum = 0;
  }
  return;
}

// 更新
void update(int rt, double y1, double y2, int flag) {
  if (cl[rt].l == y1 && cl[rt].r == y2) {
    lazy[rt] += flag;
    pushup(rt);
    return;
  } else {
    if (cl[rt * 2].r > y1) update(rt * 2, y1, min(cl[rt * 2].r, y2), flag);
    if (cl[rt * 2 + 1].l < y2)
      update(rt * 2 + 1, max(cl[rt * 2 + 1].l, y1), y2, flag);
    pushup(rt);
  }
}

int main() {
  int temp = 1, n;
  double x1, y1, x2, y2, ans;
  while (scanf("%d", &n) && n) {
    ans = 0;
    for (int i = 0; i < n; i++) {
      scanf("%lf %lf %lf %lf", &x1, &y1, &x2, &y2);
      p[i].x = x1;
      p[i].y1 = y1;
      p[i].y2 = y2;
      p[i].flag = 1;
      p[i + n].x = x2;
      p[i + n].y1 = y1;
      p[i + n].y2 = y2;
      p[i + n].flag = -1;
      s[i + 1] = y1;
      s[i + n + 1] = y2;
    }
```

```
    sort(s + 1, s + (2 * n + 1));  // 离散化
    sort(p, p + 2 * n, cmp);  // 把矩形的边的横坐标从小到大排序
    build(1, 1, 2 * n);       // 建树
    memset(lazy, 0, sizeof(lazy));
    update(1, p[0].y1, p[0].y2, p[0].flag);
    for (int i = 1; i < 2 * n; i++) {
      ans += (p[i].x - p[i - 1].x) * cl[1].sum;
      update(1, p[i].y1, p[i].y2, p[i].flag);
    }
    printf("Test case #%d\nTotal explored area: %.2lf\n\n", temp++, ans);
  }
  return 0;
}
```

练习

- 「POJ1151」Atlantis[1]
- 「POJ1177」Picture[2]
- 「POJ3832」Posters[3]
- 洛谷 P1856[IOI1998][USACO5.5] 矩形周长 Picture[4]

B 维正交范围

B 维正交范围指在一个 B 维直角坐标系下，第 i 维坐标在一个整数范围 $[l_i, r_i]$ 间，内部的点集。

一般来说，一维正交范围简称区间，二维正交范围简称矩形，三维正交范围简称立方体（我们常说的二维数点就是二维正交范围）。

对于一个静态的二维问题，我们可以使用扫描线扫一维，数据结构维护另一维。在扫描线从左到右扫的过程中，会在数据结构维护的那一维上产生一些修改与查询。如果查询的信息可差分的话直接使用差分，否则需要使用分治。差分一般用树状数组和线段树维护，但因为树状数组好写而且常数小，所以大部分人会选择用树状数组来维护。分治一般是 CDQ 分治（但是这里不涉及分治）。

另一种比较容易理解的看待问题的角度是站在序列角度，而不站在二维平面角度。如果我们这样看待问题，则扫描线实际上是枚举了右端点 $r = 1 \cdots n$，维护一个数据结构，支持查询对于当前的 r，给定一个值 l，l 到 r 的答案是什么。即扫描线扫询问右端点，数据结构维护所有左端点的答案，或者说遍历一维，数据结果维护另一维。

复杂度一般为 $\mathcal{O}((n+m)\log n)$。

二维数点

给一个长为 n 的序列，有 m 次查询，每次查区间 $[l, r]$ 中值在 $[x, y]$ 内的元素个数。

这个问题就叫做二维数点。我们可以发现等价于我们要查询一个二维平面上矩形内的点的数量和。这里讲一下这个问题最简单的处理方法，扫描线 + 树状数组。

很显然，这个问题是一个静态的二维问题，我们通过扫描线可以将静态的二维问题转换为动态的一维问题。维护动态的一维问题就使用数据结构维护序列，这里可以使用树状数组。

先将所有的询问离散化，用树状数组维护权值，对于每次询问的 l 和 r，我们在枚举到 $l-1$ 时统计当前位于区间 $[x, y]$ 内的数的数量 a，继续向后枚举，枚举到 r 时统计当前位于区间 $[x, y]$ 内的数的数量 b，$b-a$ 即为该次询问的答案。

可以用 洛谷 P2163[SHOI2007] 园丁的烦恼[5] 这道题进行练习。

例题

"洛谷 P1908 逆序对[6]"

没错，逆序对也可以用扫描线的思维来做。考虑将求逆序对的个数转化为从后向前枚举每个位置 i，求在区间 $[i+1,n]$ 中，大小在区间 $[0,a_i]$ 中的点的个数。题目中数据范围为 10^9，很显然要先进行离散化，我们可以考虑从后向前遍历数组，每次遍历到一个数时更新数组数组（线段树），之后统计当前一共有多少个数小于当前枚举的数，因为我们是从后向前遍历的，所以比当前值小的数的个数就是他的逆序对的个数，可以用树状数组或线段树进行单点修改和区间查询。

" 代码"

```cpp
#include <bits/stdc++.h>
#define ll long long
using namespace std;

struct node {
  ll data;
  ll num;
} f[5000001];

ll n, c[5000001], ans, a[5000001];

bool cmp(node a, node b) {
  if (a.data == b.data) {
    return a.num < b.num;
  }
  return a.data < b.data;
}

ll lowbit(ll i) { return i & (-i); }

ll sum(ll x) {
  ll s = 0;
  for (; x > 0; x -= lowbit(x)) {
    s += c[x];
  }
  return s;
}

int main() {
  cin >> n;
  for (ll i = 1; i <= n; i++) {
    cin >> f[i].data;
    f[i].num = i;
  }
  sort(f + 1, f + 1 + n, cmp);
  for (int i = 1; i <= n; i++) {
    a[f[i].num] = i;
  }
  for (ll i = n; i > 0; i--) {
    ans += sum(a[i]);
    for (ll j = a[i]; j <= n; j += lowbit(j)) {
      c[j]++;
    }
  }
```

```
  cout << ans;
  return 0;
}
```

"洛谷 P1972 [SDOI2009] HH 的项链[7]"

简要题意：给定一个长为 n 的序列，m 次查询区间中有多少不同的数。

这类问题我们可以考虑推导性质，之后使用扫描线枚举所有右端点，数据结构维护每个左端点的答案的方法来实现，我们也可以将问题转换到二维平面上，变为一个矩形查询信息的问题。

在这道题中，对于每个位置 i，考虑预处理出 i 左边离 i 最近的 j 满足 $a_i = a_j$，用一个数组记录，即 $pre_i = j$。

然后查询区间中的不同数，我们可以只把每个数在区间中最后一次出现时统计进去。

若这个数在当前区间中是第一次出现，那么这个数肯定满足 $pre_i < l$。如果不是第一次出现，那么 $l \le pre_i$。这样问题就转变成了求区间 $[l, r]$ 中，满足 $pre_i < l$ 的 i 的个数。

我们可以考虑差分，将区间 $[l, r]$ 差分为前缀 $[1, r]$ 减去前缀 $[1, l-1]$。考虑将询问离线处理，假设有一个询问是对于区间 $[l, r]$ 的，我们在 $l-1$ 位置上和 r 位置上分别记录一下，答案为在 r 处记录的 $pre_i < l$ 的个数减去在 $l-1$ 处记录的 $pre_i < l$ 的 i 的个数。

每次查询可以用值域线段树或值域树状数组来维护，注意一个位置上可能有多个询问，但总共的查询次数是 m 次。总时间复杂度 $\mathcal{O}((n+m)\log n)$。

" 代码"

```cpp
#include <bits/stdc++.h>
#define ls x << 1
#define rs x << 1 | 1
#define N 1000010
using namespace std;

struct node {
  int l, r, ans;
} q[N];

struct t {
  int num, s;
};

vector<t> p[N];
int n, a[N], m, now[N];
int siz[N << 2];

void update(int x, int l, int r, int ad) {
  if (l == r && l == ad) {
    siz[x]++;
    return;
  }
  int mid = l + r >> 1;
  if (ad <= mid) {
    update(ls, l, mid, ad);
  } else {
    update(rs, mid + 1, r, ad);
  }
```

```
  siz[x] = siz[ls] + siz[rs];
}

int query(int x, int l, int r, int L, int R) {
  if (l >= L && r <= R) {
    return siz[x];
  }
  int mid = l + r >> 1;
  int res = 0;
  if (L <= mid) {
    res += query(ls, l, mid, L, R);
  }
  if (R > mid) {
    res += query(rs, mid + 1, r, L, R);
  }
  return res;
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) {
    scanf("%d", &a[i]);
  }
  scanf("%d", &m);
  for (int i = 1; i <= m; i++) {
    int l, r;
    scanf("%d%d", &l, &r);
    p[l - 1].push_back(t{i, -1});
    p[r].push_back(t{i, 1});
    q[i] = node{l, r, 0};
  }
  for (int i = 1; i <= n; i++) {
    update(1, 0, n, now[a[i]]);
    now[a[i]] = i;
    for (auto x : p[i]) {
      int num = x.num;
      q[num].ans += x.s * query(1, 0, n, 0, q[num].l - 1);
    }
  }
  for (int i = 1; i <= m; i++) {
    printf("%d\n", q[i].ans);
  }
  return 0;
}
```

例题

- 洛谷 P8593「KDOI-02」一个弹的投[8] 逆序对的应用。
- AcWing 4709. 三元组[9] 上题的弱化版，同样为逆序对的应用。
- 洛谷 P8773[蓝桥杯 2022 省 A] 选数异或[10] HH 的项链魔改版。
- 洛谷 P8844[传智杯 #4 初赛] 小卡与落叶[11] 树上问题转序列问题然后进行二维数点。

总而言之，二维数点的主要思路就是数据结构维护一维，然后枚举另一维。

参考资料

- https://www.cnblogs.com/yangsongyi/p/8378629.html[12]
- https://blog.csdn.net/riba2534/article/details/76851233[13]
- https://blog.csdn.net/winddreams/article/details/38495093[14]
- https://dregen-yor.cf/2022/10/01/sao-miao-xian[15]

参考资料与注释

[1] 「POJ1151」Atlantis

[2] 「POJ1177」Picture

[3] 「POJ3832」Posters

[4] 洛谷 P1856[IOI1998][USACO5.5] 矩形周长 Picture

[5] 洛谷 P2163[SHOI2007] 园丁的烦恼

[6] 洛谷 P1908 逆序对

[7] 洛谷 P1972 [SDOI2009] HH 的项链

[8] 洛谷 P8593「KDOI-02」一个弹的投

[9] AcWing 4709. 三元组

[10] 洛谷 P8773[蓝桥杯 2022 省 A] 选数异或

[11] 洛谷 P8844[传智杯 #4 初赛] 小卡与落叶

[12] https://www.cnblogs.com/yangsongyi/p/8378629.html

[13] https://blog.csdn.net/riba2534/article/details/76851233

[14] https://blog.csdn.net/winddreams/article/details/38495093

[15] https://dregen-yor.cf/2022/10/01/sao-miao-xian

12.9 旋转卡壳

本页面将主要介绍旋转卡壳。

引入

旋转卡壳（Rotating Calipers，也称「旋转卡尺」）算法，在凸包算法的基础上，通过枚举凸包上某一条边的同时维护其他需要的点，能够在线性时间内求解如凸包直径、最小矩形覆盖等和凸包性质相关的问题。

" 算法中文名称"

该算法比较常见的中文名是「旋转卡壳」。可以理解为：根据我们枚举的边，可以从每个维护的点画出一条或平行或垂直的直线，为了确保对于当前枚举的边的最优性，我们的任务就是使这些直线能将凸包正好卡住。而边通常是按照向某一方向旋转的顺序来枚举，所以整个过程就是在边「旋转」，边「卡壳」。

其英文名「rotating calipers」的直译应为「旋转卡尺」，其中「calipers」的意思是「卡尺」。第一次提出该术语的论文[1] 原意为：使用一个可动态调整的「卡尺」夹住凸包后，绕凸包「旋转」该「卡尺」。

求凸包直径

" 例题 1 :Luogu P1452 Beauty Contest G[2]"

给定平面上 n 个点，求所有点对之间的最长距离。($2 \le n \le 50000, |x|, |y| \le 10^4$)

过程

首先使用任何一种凸包算法求出给定所有点的凸包，有着最长距离的点对一定在凸包上。而由于凸包的形状，我们发现，逆时针地遍历凸包上的边，对于每条边都找到离这条边最远的点，那么这时随着边的转动，对应的最远点也在逆时针旋转，不会有反向的情况，这意味着我们可以在逆时针枚举凸包上的边时，记录并维护一个当前最远点，并不断计算、更新答案。

求出凸包后的数组自然地是按照逆时针旋转的顺序排列，不过要记得提前将最左下角的 1 节点补到数组最后，这样在挨个枚举边 $(i, i+1)$ 时，才能把所有边都枚举到。

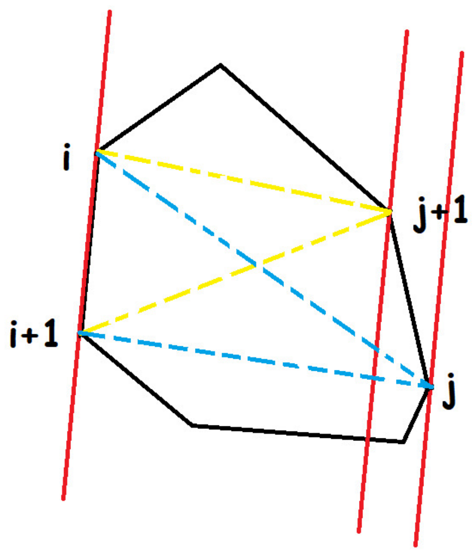


图 12.26

枚举过程中，对于每条边，都检查 $j+1$ 和边 $(i, i+1)$ 的距离是不是比 j 更大，如果是就将 j 加一，否则说明 j 是此边的最优点。判断点到边的距离大小时可以用叉积分别算出两个三角形的面积（如图，黄、蓝两个同底三角形的面积）并直接比较。

实现

" 核心代码"

```cpp
int sta[N], top;  // 将凸包上的节点编号存在栈里，第一个和最后一个节点编号相同
bool is[N];

ll pf(ll x) { return x * x; }

ll dis(int p, int q) { return pf(a[p].x - a[q].x) + pf(a[p].y - a[q].y); }

ll sqr(int p, int q, int y) { return abs((a[q] - a[p]) * (a[y] - a[q])); }

ll mx;

void get_longest() {  // 求凸包直径
  int j = 3;
  if (top < 4) {
    mx = dis(sta[1], sta[2]);
    return;
  }
  for (int i = 1; i <= top; ++i) {
    while (sqr(sta[i], sta[i + 1], sta[j]) <=
           sqr(sta[i], sta[i + 1], sta[j % top + 1]))
      j = j % top + 1;
    mx = max(mx, max(dis(sta[i + 1], sta[j]), dis(sta[i], sta[j])));
  }
}
```

```python
sta = [0] * N; top = 0 # 将凸包上的节点编号存在栈里，第一个和最后一个节点编号相同
def pf(x):
    return x * x
def dis(p, q):
    return pf(a[p].x - a[q].x) + pf(a[p].y - a[q].y)
def sqr(p, q, y):
    return abs((a[q] - a[p]) * (a[y] - a[q]))
def get_longest(): # 求凸包直径
    j = 3
    if top < 4:
        mx = dis(sta[1], sta[2])
        return
    for i in range(1, top + 1):
        while sqr(sta[i], sta[i + 1], sta[j]) <=\
            sqr(sta[i], sta[i + 1], sta[j % top + 1]):
            j = j % top + 1
        mx = max(mx, max(dis(sta[i + 1], sta[j]), dis(sta[i], sta[j])))
```

求最小矩形覆盖

Luogu P3187 最小矩形覆盖[3]

给定一些点的坐标，求能够覆盖所有点的最小面积的矩形。（$3 \leq n \leq 50000$）

过程

有了上一道题做铺垫，这道题比较直观的想法仍然是使用旋转卡壳法，不过这次要求的是面积，像上一题一样只维护一个最优点就只能找到一对距离最小的平行线，我们还需要确定矩形的左右边界。所以这次我们需要维护三个点：一个在所枚举的直线对面的点、两个在不同侧面的点。对面的最优点仍然是用叉积算面积来比较，此时比较面积就是在比较这个矩形的一个边长。侧面的最优点则是用点积来比较，因为比较点积就是比较投影的长度，左右两个投影长度相加可以代表这个矩形的另一个边长。这两个边长的最优性相互独立，因此找到三个最优点的位置就能够确定以当前边所在直线为矩阵的一条边时，能覆盖所有点的矩形最小面积。

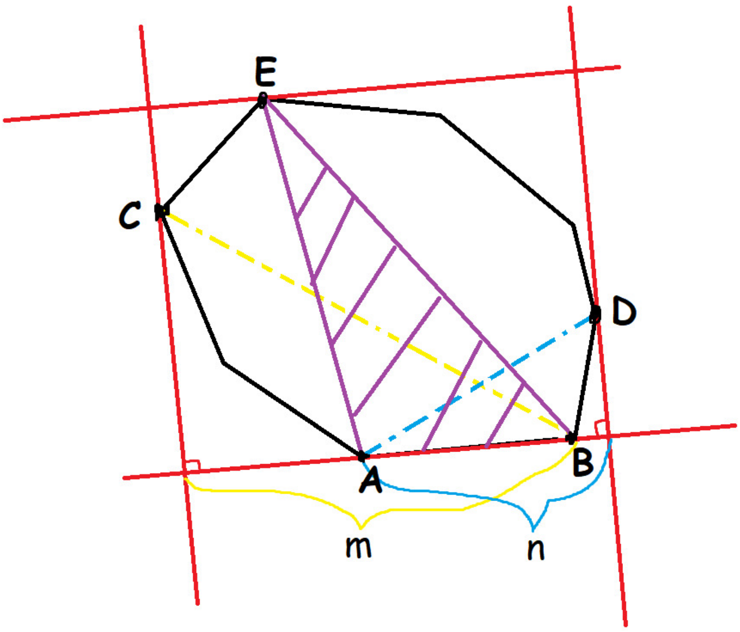


图 12.27

最后统计答案时，如果题目没有要求将四个顶点都求出来，其实有一种较为巧妙的利用叉积和点积的方式直接算出矩阵的面积。设紫色部分面积的两倍为 S，最后的面积就是

$$S \times (|\overrightarrow{AD} \cdot \overrightarrow{AB}| + |\overrightarrow{BC} \cdot \overrightarrow{BA}| - |\overrightarrow{AB} \cdot \overrightarrow{BA}|)/|\overrightarrow{AB} \cdot \overrightarrow{BA}|$$

实现

必要的求凸包过程略去，这里贴出本题核心代码：

" 核心代码"

```
void get_biggest() {
  int j = 3, l = 2, r = 2;
  double t1, t2, t3, ans = 2e10;
  for (int i = 1; i <= top; ++i) {
    while (sqr(sta[i], sta[i + 1], sta[j]) <=
           sqr(sta[i], sta[i + 1], sta[j % top + 1]))
      j = j % top + 1;
    while (dot(sta[i + 1], sta[r % top + 1], sta[i]) >=
           dot(sta[i + 1], sta[r], sta[i]))
      r = r % top + 1;
    if (i == 1) l = r;
    while (dot(sta[i + 1], sta[l % top + 1], sta[i]) <=
```

```
            dot(sta[i + 1], sta[l], sta[i]))
        l = l % top + 1;
      t1 = sqr(sta[i], sta[i + 1], sta[j]);
      t2 = dot(sta[i + 1], sta[r], sta[i]) + dot(sta[i + 1], sta[l], sta[i]);
      t3 = dot(sta[i + 1], sta[i + 1], sta[i]);
      ans = min(ans, t1 * t2 / t3);
  }
}
```

```python
def get_biggest():
    j = 3; l = 2; r = 2
    ans = 2e10
    for i in range(1, top + 1):
        while sqr(sta[i], sta[i + 1], sta[j]) <=\
            sqr(sta[i], sta[i + 1], sta[j % top + 1]):
            j = j % top + 1
        while dot(sta[i + 1], sta[r % top + 1], sta[i]) >=\
            dot(sta[i + 1], sta[r], sta[i]):
            r = r % top + 1
        if i == 1:
            l = r
        while dot(sta[i + 1], sta[l % top + 1], sta[i]) <=\
            dot(sta[i + 1], sta[l], sta[i]):
            l = l % top + 1
        t1 = sqr(sta[i], sta[i + 1], sta[j])
        t2 = dot(sta[i + 1], sta[r], sta[i]) + dot(sta[i + 1], sta[l], sta[i])
        t3 = dot(sta[i + 1], sta[i + 1], sta[i])
        ans = min(ans, t1 * t2 / t3)
```

练习

- POJ 3608. Bridge Across Islands[4]
- 2011 ACM-ICPC World Finals, Problem K. Trash Removal[5]
- ICPC WF Moscow Invitational Contest - Online Mirror, Problem F. Framing Pictures[6]

参考资料与注释

- https://en.wikipedia.org/wiki/Rotating_calipers[7]
- http://www-cgrl.cs.mcgill.ca/~godfried/research/calipers.html[8]
- Shamos, Michael (1978). "Computational Geometry" (PDF). Yale University. pp. 76–81.

[1] Toussaint, Godfried T. (1983). "Solving geometric problems with the rotating calipers". Proc. MELECON '83, Athens. CiteSeerX 10.1.1.155.5671

[2] Luogu P1452 Beauty Contest G

[3] Luogu P3187 最小矩形覆盖

[4] POJ 3608. Bridge Across Islands

[5] 2011 ACM-ICPC World Finals, Problem K. Trash Removal

[6] ICPC WF Moscow Invitational Contest - Online Mirror, Problem F. Framing Pictures

[7] https://en.wikipedia.org/wiki/Rotating_calipers

[8] http://www-cgrl.cs.mcgill.ca/~godfried/research/calipers.html

12.10 半平面交

Authors: wjy-yy, Ir1d, Xeonacid

定义

半平面

一条直线和直线的一侧。半平面是一个点集，因此是一条直线和直线的一侧构成的点集。当包含直线时，称为闭半平面；当不包含直线时，称为开半平面。

解析式一般为 $Ax+By+C\geq 0$。

在计算几何中用向量表示，整个题统一以向量的左侧或右侧为半平面。

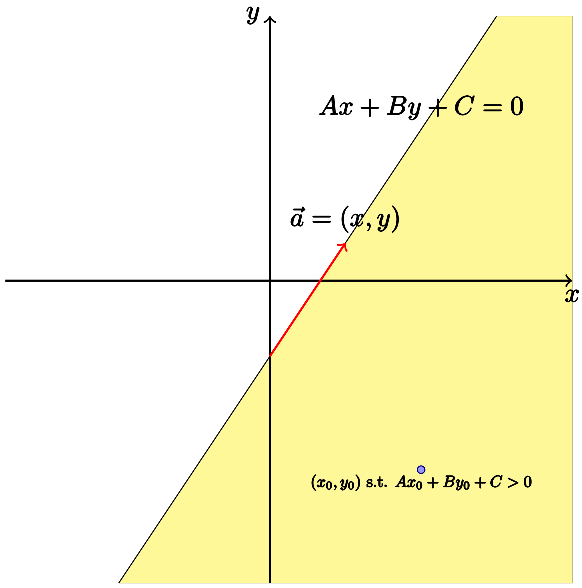


图 12.28 半平面

半平面交

半平面交是指多个半平面的交集。因为半平面是点集，所以点集的交集仍然是点集。在平面直角坐标系围成一个区域。

这就很像普通的线性规划问题了，得到的半平面交就是线性规划中的可行域。一般情况下半平面交是有限的，经常考察面积等问题的解决。

它可以理解为向量集中每一个向量的右侧的交，或者是下面方程组的解。

$$\begin{cases} A_1x + B_1y + C \geq 0 \\ A_2x + B_2y + C \geq 0 \\ \dots \end{cases}$$

多边形的核

如果一个点集中的点与多边形上任意一点的连线与多边形没有其他交点，那么这个点集被称为多边形的核。

把多边形的每条边看成是首尾相连的向量，那么这些向量在多边形内部方向的半平面交就是多边形的核。

解法 - S&I 算法

极角排序

C 语言有一个库函数叫做 `atan2(double y,double x)`，可以返回 $\theta \in (-\pi, \pi]$，$\theta = \arctan\frac{y}{x}$。

直接以向量为自变量，调用这个函数，以返回值为关键字排序，得到新的边（向量）集。

排序时，如果遇到共线向量（且方向相同），则取靠近可行域的一个。比如两个向量的极角相同，而我们要的是向量的左侧半平面，那么我们只需要保留左侧的向量。判断方法是取其中一个向量的起点或终点与另一个比较，检查是在左边还是在右边。

维护单调队列

因为半平面交是一个凸多边形，所以需要维护一个凸壳。因为后来加入的只可能会影响最开始加入的或最后加入的边（此时凸壳连通），只需要删除队首和队尾的元素，所以需要用单调队列。

我们遍历排好序了的向量，并维护另一个交点数组。当单队中元素超过 2 个时，他们之间就会产生交点。

对于当前向量，如果上一个交点在这条向量表示的半平面交的**异侧**，那么上一条边就没有意义了。

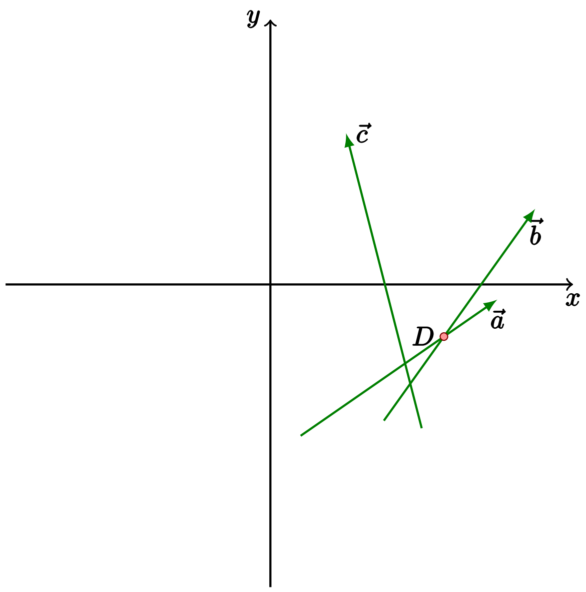


图 12.29 单调队列

如上图，假设取向量左侧半平面。极角排序后，遍历顺序应该是 $\vec{a} \to \vec{b} \to \vec{c}$。当 $\vec{a}$ 和 $\vec{b}$ 入队时，在交点数组里会产生一个点 D（交点数组保存队列中相同下标的向量与前一向量的交点）。

接下来枚举到 $\vec{c}$ 时，发现 D 在 $\vec{c}$ 的右侧。而因为**产生 D 的向量的极角一定比 $\vec{c}$ 要小**，所以产生 D 的向量（指 $\vec{b}$）就对半平面交没有影响了。

还有一种可能的情况是快结束的时候，新加入的向量会从队首开始造成影响。

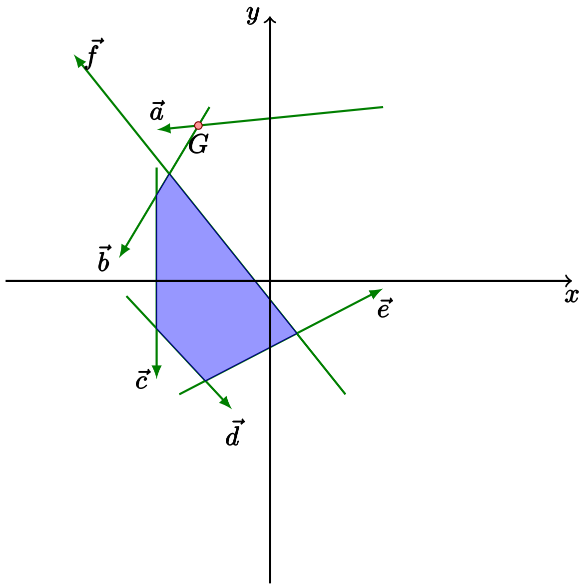


图 12.30 队首影响

仍然假设取向量左侧半平面。加入向量 $\vec{f}$ 之后，第一个交点 G 就在 $\vec{f}$ 的右侧，我们把上面的判断标准逆过来看，就知道此时应该删除向量 $\vec{a}$，也即**队首**的向量。

最后用队首的向量排除一下队尾多余的向量。因为队首的向量会被后面的约束，而队尾的向量不会。此时它们围成了一个环，因此队首的向量就可以约束队尾的向量。

得到半平面交

如果半平面交是一个凸 n 边形，最后在交点数组里会得到 n 个点。我们再把它们首尾相连，就是一个统一方向（顺或逆时针）的 n 多边形。

此时就可以用三角剖分求面积了。（求面积是最基础的考法）

偶尔会出现半平面交不存在或面积为 0 的情况，注意考虑边界。

注意事项

当出现一个可以把队列里的点全部弹出去的向量（即所有队列里的点都在该向量的右侧），则我们**必须**先处理队尾，再处理队首。因此在循环中，我们先枚举 `--r;` 的部分，再枚举 `++l;` 的部分，才不会错。原因如下。

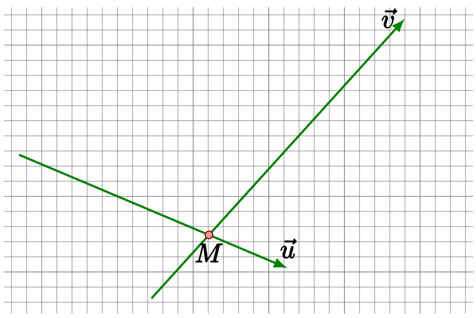


图 12.31

一般情况下，我们在队列（队列顺序为 $\{\vec{u}, \vec{v}\}$）后面加一条边（向量 $\vec{w}$），会产生一个交点 N，缩小 $\vec{v}$ 后面的范围。

但是毕竟每次操作都是一般的，因此可能会有把 M 点「挤出去」的情况。

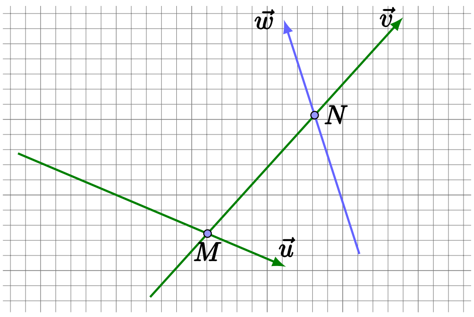


图 12.32

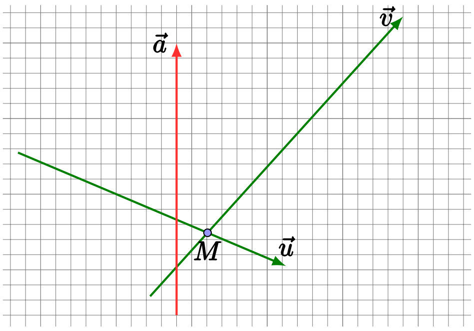


图 12.33

如果此时出现了向量 $\vec{a}$，使得 M 在 $\vec{a}$ 的右侧，那么 M 就要出队了。此时如果从队首枚举 `++l`，显然是扩大了范围。实际上 M 点是由 $\vec{u}$ 和 $\vec{v}$ 共同构成的，因此需要考虑影响到现有进程的是 $\vec{u}$ 还是 $\vec{v}$。而因为我们在极角排序后，向量是逆时针顺序，所以 $\vec{v}$ 的影响要更大一些。

就如上图，如果 M 确认在 $\vec{a}$ 的右侧，那么此时 $\vec{v}$ 的影响一定不会对半平面交的答案作出任何贡献。

而我们排除队首的原因是**当前向量的限制比队首向量要大**，这个条件的前提是队列里有不止两个线段（向量），不然就会出现上面的情况。

所以一定要先排除队尾再排除队首。

" 代码 - 比较部分"

```
friend bool operator<(seg x, seg y) {
  db t1 = atan2((x.b - x.a).y, (x.b - x.a).x);
  db t2 = atan2((y.b - y.a).y, (y.b - y.a).x);  // 求极角
  if (fabs(t1 - t2) > eps)                       // 如果极角不等
    return t1 < t2;
  return (y.a - x.a) * (y.b - x.a) >
         eps;  // 判断向量 x 在 y 的哪边，令最靠左的排在最左边
}
```

" 代码 - 增量部分"

```
// pnt its(seg a,seg b) 表示求线段 a,b 的交点
// s[] 是极角排序后的向量
// q[] 是向量队列
// t[i] 是 s[i-1] 与 s[i] 的交点
// 【码风】队列的范围是 (l,r]
// 求的是向量左侧的半平面
int l = 0, r = 0;
```

```
for (int i = 1; i <= n; ++i)
  if (s[i] != s[i - 1]) {
    // 注意要先检查队尾
    while (r - l > 1 && (s[i].b - t[r]) * (s[i].a - t[r]) >
                           eps)  // 如果上一个交点在向量右侧则弹出队尾
      --r;
    while (r - l > 1 && (s[i].b - t[l + 2]) * (s[i].a - t[l + 2]) >
                           eps)  // 如果第一个交点在向量右侧则弹出队首
      ++l;
    q[++r] = s[i];
    if (r - l > 1) t[r] = its(q[r], q[r - 1]);  // 求新交点
  }
while (r - l > 1 &&
       (q[l + 1].b - t[r]) * (q[l + 1].a - t[r]) > eps)  // 注意删除多余元素
  --r;
t[r + 1] = its(q[l + 1], q[r]);  // 再求出新的交点
++r;
// 这里不能在 t 里面 ++r 需要注意一下……
```

练习

POJ 2451 Uyuw's Concert[1] 注意边界

POJ 1279 Art Gallery[2] 求多边形的核

「CQOI2006」 凸多边形[3]

参考资料与注释

[1] POJ 2451 Uyuw's Concert

[2] POJ 1279 Art Gallery

[3] 「CQOI2006」凸多边形

12.11 平面最近点对

引入

给定 n 个二维平面上的点，求一组欧几里得距离最近的点对。

下面我们介绍一种时间复杂度为 $O(n\log n)$ 的分治算法来解决这个问题。该算法在 1975 年由 Franco P. Preparata[1] 提出，Preparata 和 Michael Ian Shamos[2] 证明了该算法在决策树模型下是最优的。

过程

与常规的分治算法一样，我们将这个有 n 个点的集合拆分成两个大小相同的集合 S_1, S_2，并不断递归下去。但是我们遇到了一个难题：如何合并？即如何求出一个点在 S_1 中，另一个点在 S_2 中的最近点对？这里我们先假设合并操作的时间复杂度为 $O(n)$，可知算法总复杂度为 $T(n) = 2T(\frac{n}{2}) + O(n) = O(n\log n)$。

我们先将所有点按照 x_i 为第一关键字、y_i 为第二关键字排序，并以点 $p_m (m = \lfloor\frac{n}{2}\rfloor)$ 为分界点，拆分点集为

A_1, A_2:

$$A_1 = \{p_i \mid i = 0 \dots m\}$$
$$A_2 = \{p_i \mid i = m+1 \dots n-1\}$$

并递归下去，求出两点集各自内部的最近点对，设距离为 h_1, h_2，取较小值设为 h。

现在该合并了！我们试图找到这样的一组点对，其中一个属于 A_1，另一个属于 A_2，且二者距离小于 h。因此我们将所有横坐标与 x_m 的差小于 h 的点放入集合 B:

$$B = \{p_i \mid |x_i - x_m| < h\}$$

结合图像，直线 m 将点分成了两部分。m 左侧为 A_1 点集，右侧为为 A_2 点集。

再根据 $B = \{p_i \mid |x_i - x_m| < h\}$ 规则，得到绿色点组成的 B 点集。

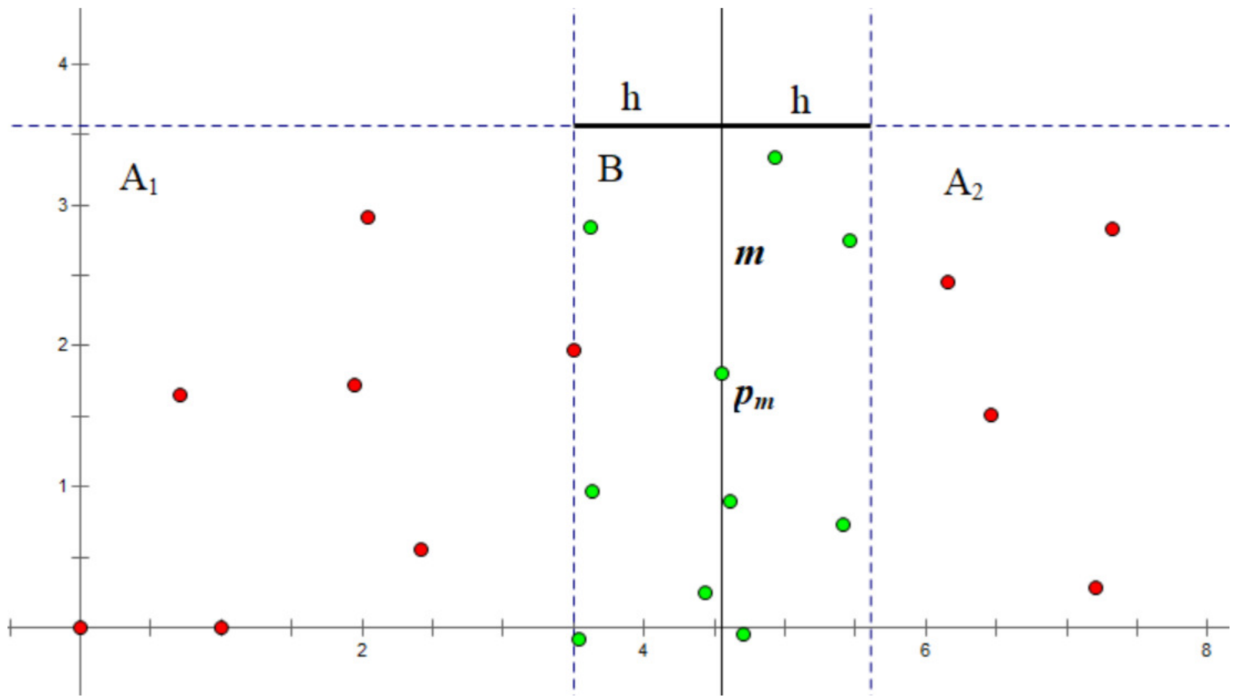


图 12.34 nearest-points1

对于 B 中的每个点 p_i，我们当前目标是找到一个同样在 B 中、且到其距离小于 h 的点。为了避免两个点之间互相考虑，我们只考虑那些纵坐标小于 y_i 的点。显然对于一个合法的点 p_j，$y_i - y_j$ 必须小于 h。于是我们获得了一个集合 $C(p_i)$:

$$C(p_i) = \{p_j \mid p_j \in B,\ y_i - h < y_j \le y_i\}$$

在点集 B 中选一点 p_i，根据 $C(p_i) = \{p_j \mid p_j \in B,\ y_i - h < y_j \le y_i\}$ 的规则，得到了由红色方框内的黄色点组成的 C 点集。

如果我们将 B 中的点按照 y_i 排序，$C(p_i)$ 将很容易得到，即紧邻 p_i 的连续几个点。

由此我们得到了合并的步骤:

1. 构建集合 B。
2. 将 B 中的点按照 y_i 排序。通常做法是 $O(n \log n)$，但是我们可以改变策略优化到 $O(n)$（下文讲解）。
3. 对于每个 $p_i \in B$ 考虑 $p_j \in C(p_i)$，对于每对 (p_i, p_j) 计算距离并更新答案（当前所处集合的最近点对）。

注意到我们上文提到了两次排序，因为点坐标全程不变，第一次排序可以只在分治开始前进行一次。我们令每次递归返回当前点集按 y_i 排序的结果，对于第二次排序，上层直接使用下层的两个分别排序过的点集归并即可。

似乎这个算法仍然不优，$|C(p_i)|$ 将处于 $O(n)$ 数量级，导致总复杂度不对。其实不然，其最大大小为 7，我们给出它的证明:

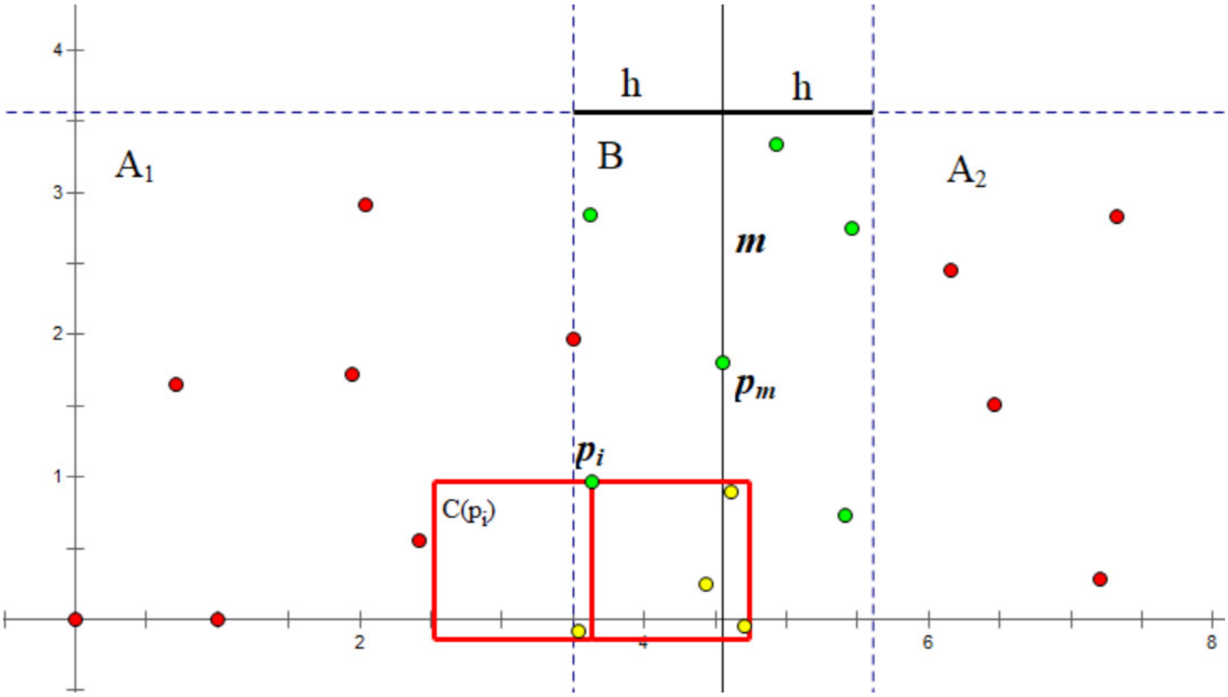


图 12.35 nearest-points2

复杂度证明

我们已经了解到，$C(p_i)$ 中的所有点的纵坐标都在 $(y_i-h, y_i]$ 范围内；且 $C(p_i)$ 中的所有点，和 p_i 本身，横坐标都在 (x_m-h, x_m+h) 范围内。这构成了一个 $2h\times h$ 的矩形。

我们再将这个矩形拆分为两个 $h\times h$ 的正方形，不考虑 p_i，其中一个正方形中的点为 $C(p_i)\cap A_1$，另一个为 $C(p_i)\cap A_2$，且两个正方形内的任意两点间距离大于 h。（因为它们来自同一下层递归）

我们将一个 $h\times h$ 的正方形拆分为四个 $\frac{h}{2}\times\frac{h}{2}$ 的小正方形。可以发现，每个小正方形中最多有 1 个点：因为该小正方形中任意两点最大距离是对角线的长度，即 $\frac{h}{\sqrt{2}}$，该数小于 h。

由此，每个正方形中最多有 4 个点，矩形中最多有 8 个点，去掉 p_i 本身，$\max(C(p_i))=7$。

实现

我们使用一个结构体来存储点，并定义用于排序的函数对象：

" 结构体定义"

```
struct pt {
  int x, y, id;
};

struct cmp_x {
  bool operator()(const pt& a, const pt& b) const {
    return a.x < b.x || (a.x == b.x && a.y < b.y);
  }
};

struct cmp_y {
  bool operator()(const pt& a, const pt& b) const { return a.y < b.y; }
};
```

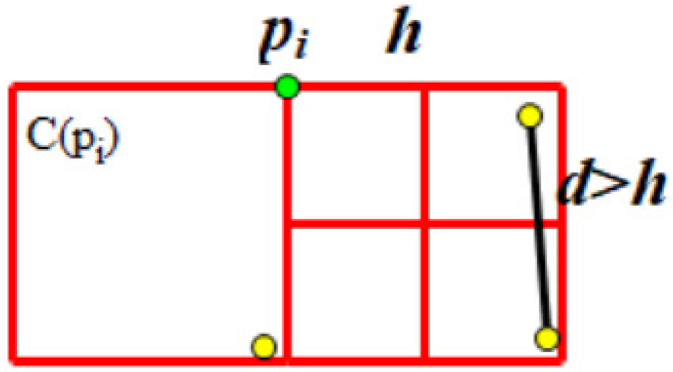


图 12.36 nearest-points3

```
int n;
vector<pt> a;
```

为了方便实现递归，我们引入 `upd_ans()` 辅助函数来计算两点间距离并尝试更新答案：

" 答案更新函数"

```
double mindist;
int ansa, ansb;

void upd_ans(const pt& a, const pt& b) {
  double dist =
      sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y) + .0);
  if (dist < mindist) mindist = dist, ansa = a.id, ansb = b.id;
}
```

下面是递归本身：假设在调用前 `a[]` 已按 x_i 排序。如果 $r-l$ 过小，使用暴力算法计算 h，终止递归。

我们使用 `std::inplace_merge()` 来执行归并排序，并创建辅助缓冲区 `t[]`，B 存储在其中。

" 主体函数"

```
void rec(int l, int r) {
  if (r - l <= 3) {
    for (int i = l; i <= r; ++i)
      for (int j = i + 1; j <= r; ++j) upd_ans(a[i], a[j]);
    sort(a + l, a + r + 1, &cmp_y);
    return;
  }

  int m = (l + r) >> 1;
  int midx = a[m].x;
```

```
  rec(l, m), rec(m + 1, r);
  inplace_merge(a + l, a + m + 1, a + r + 1, &cmp_y);

  static pt t[MAXN];
  int tsz = 0;
  for (int i = l; i <= r; ++i)
    if (abs(a[i].x - midx) < mindist) {
      for (int j = tsz - 1; j >= 0 && a[i].y - t[j].y < mindist; --j)
        upd_ans(a[i], t[j]);
      t[tsz++] = a[i];
    }
}
```

在主函数中，这样开始递归即可：

" 调用接口"

```
sort(a, a + n, &cmp_x);
mindist = 1E20;
rec(0, n - 1);
```

推广：平面最小周长三角形

上述算法有趣地推广到这个问题：在给定的一组点中，选择三个点，使得它们两两的距离之和最小。

算法大体保持不变，每次尝试找到一个比当前答案周长 d 更小的三角形，将所有横坐标与 x_m 的差小于 $\frac{d}{2}$ 的点放入集合 B，尝试更新答案。（周长为 d 的三角形的最长边小于 $\frac{d}{2}$）

非分治算法

过程

其实，除了上面提到的分治算法，还有另一种时间复杂度同样是 $O(n\log n)$ 的非分治算法。

我们可以考虑一种常见的统计序列的思想：对于每一个元素，将它和它的左边所有元素的贡献加入到答案中。平面最近点对问题同样可以使用这种思想。

具体地，我们把所有点按照 x_i 为第一关键字、y_i 为第二关键字排序，并建立一个以 y_i 为第一关键字、x_i 为第二关键字排序的 multiset。对于每一个位置 i，我们执行以下操作：

1. 将所有满足 $x_i - x_j >= d$ 的点从集合中删除。它们不会再对答案有贡献。
2. 对于集合内满足 $|y_i - y_j| < d$ 的所有点，统计它们和 p_i 的距离。
3. 将 p_i 插入到集合中。

由于每个点最多会被插入和删除一次，所以插入和删除点的时间复杂度为 $O(n\log n)$，而统计答案部分的时间复杂度证明与分治算法的时间复杂度证明方法类似，读者不妨一试。

实现

" 参考代码"

```
#include <algorithm>
#include <cmath>
#include <cstdio>
```

```
#include <set>
const int N = 200005;
int n;
double ans = 1e20;

struct point {
  double x, y;

  point(double x = 0, double y = 0) : x(x), y(y) {}
};

struct cmp_x {
  bool operator()(const point &a, const point &b) const {
    return a.x < b.x || (a.x == b.x && a.y < b.y);
  }
};

struct cmp_y {
  bool operator()(const point &a, const point &b) const { return a.y < b.y; }
};

void upd_ans(const point &a, const point &b) {
  double dist = sqrt(pow((a.x - b.x), 2) + pow((a.y - b.y), 2));
  if (ans > dist) ans = dist;
}

point a[N];
std::multiset<point, cmp_y> s;

int main() {
  scanf("%d", &n);
  for (int i = 0; i < n; i++) scanf("%lf%lf", &a[i].x, &a[i].y);
  std::sort(a, a + n, cmp_x());
  for (int i = 0, l = 0; i < n; i++) {
    while (l < i && a[i].x - a[l].x >= ans) s.erase(s.find(a[l++]));
    for (auto it = s.lower_bound(point(a[i].x, a[i].y - ans));
         it != s.end() && it->y - a[i].y < ans; it++)
      upd_ans(*it, a[i]);
    s.insert(a[i]);
  }
  printf("%.4lf", ans);
  return 0;
}
```

期望线性做法

其实，除了上面提到的时间复杂度为 $O(n\log n)$ 的做法，还有一种**期望**复杂度为 $O(n)$ 的算法。

首先将点对 随机打乱，我们将维护前缀点集的答案。考虑从前 $i-1$ 个点求出第 i 个点的答案。

记前 $i-1$ 个点的最近点对距离为 s，我们将平面以 s 为边长划分为若干个网格，并存下每个网格内的点（使用 哈希表），然后检查第 i 个点所在网格的周围九个网格中的所有点，并更新答案。注意到需检查的点的个数是 $O(1)$ 的，因为前 $i-1$ 个点的最近点对距离为 s，从而每个网格不超过 4 个点。

如果这一过程中，答案被更新，我们就重构网格图，否则不重构。在前 i 个点中，最近点对包含 i 的概率为 $O\left(\frac{1}{i}\right)$，而重构网格的代价为 $O(i)$，从而第 i 个点的期望代价为 $O(1)$。于是对于 n 个点，该算法期望为 $O(n)$。

习题

- UVa 10245 ”The Closest Pair Problem”[难度： 低][3]
- SPOJ #8725 CLOPPAIR ”Closest Point Pair”[难度： 低][4]
- CODEFORCES Team Olympiad Saratov - 2011 ”Minimum amount”[难度： 中][5]
- SPOJ #7029 CLOSEST ”Closest Triple”[难度： 中][6]
- Google Code Jam 2009 Final ”Min Perimeter”[难度： 中][7]

参考资料与拓展阅读

本页面中的分治算法部分主要译自博文 [8] 与其英文翻译版 Finding the nearest pair of points[9]。其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0。

知乎专栏： 计算几何 - 最近点对问题[10]

参考资料与注释

[1] Franco P. Preparata

[2] Michael Ian Shamos

[3] UVa 10245 ”The Closest Pair Problem”[难度：低]

[4] SPOJ #8725 CLOPPAIR ”Closest Point Pair”[难度：低]

[5] CODEFORCES Team Olympiad Saratov - 2011 ”Minimum amount”[难度：中]

[6] SPOJ #7029 CLOSEST ”Closest Triple”[难度：中]

[7] Google Code Jam 2009 Final ”Min Perimeter”[难度：中]

[8]

[9] Finding the nearest pair of points

[10] 知乎专栏：计算几何 - 最近点对问题

12.12 随机增量法

Authors: Ir1d, TianyiQ

引入

随机增量算法是计算几何的一个重要算法，它对理论知识要求不高，算法时间复杂度低，应用范围广大。

增量法 (Incremental Algorithm) 的思想与第一数学归纳法类似，它的本质是将一个问题化为规模刚好小一层的

子问题。解决子问题后加入当前的对象。写成递归式是：

$$T(n) = T(n-1) + g(n)$$

增量法形式简洁，可以应用于许多的几何题目中。

增量法往往结合随机化，可以避免最坏情况的出现。

最小圆覆盖问题

题意描述

在一个平面上有 n 个点，求一个半径最小的圆，能覆盖所有的点。

过程

假设圆 O 是前 $i-1$ 个点的最小覆盖圆，加入第 i 个点，如果在圆内或边上则什么也不做。否则，新得到的最小覆盖圆肯定经过第 i 个点。

然后以第 i 个点为基础（半径为 0），重复以上过程依次加入第 j 个点，若第 j 个点在圆外，则最小覆盖圆必经过第 j 个点。

重复以上步骤。（因为最多需要三个点来确定这个最小覆盖圆，所以重复三次）

遍历完所有点之后，所得到的圆就是覆盖所有点得最小圆。

性质

时间复杂度 $O(n)$，证明详见参考资料。

空间复杂度 $O(n)$

实现

" 代码实现"

```
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <iostream>

using namespace std;

int n;
double r;

struct point {
  double x, y;
} p[100005], o;

double sqr(double x) { return x * x; }

double dis(point a, point b) { return sqrt(sqr(a.x - b.x) + sqr(a.y - b.y)); }

bool cmp(double a, double b) { return fabs(a - b) < 1e-8; }
```

```
point geto(point a, point b, point c) {
  double a1, a2, b1, b2, c1, c2;
  point ans;
  a1 = 2 * (b.x - a.x), b1 = 2 * (b.y - a.y),
  c1 = sqr(b.x) - sqr(a.x) + sqr(b.y) - sqr(a.y);
  a2 = 2 * (c.x - a.x), b2 = 2 * (c.y - a.y),
  c2 = sqr(c.x) - sqr(a.x) + sqr(c.y) - sqr(a.y);
  if (cmp(a1, 0)) {
    ans.y = c1 / b1;
    ans.x = (c2 - ans.y * b2) / a2;
  } else if (cmp(b1, 0)) {
    ans.x = c1 / a1;
    ans.y = (c2 - ans.x * a2) / b2;
  } else {
    ans.x = (c2 * b1 - c1 * b2) / (a2 * b1 - a1 * b2);
    ans.y = (c2 * a1 - c1 * a2) / (b2 * a1 - b1 * a2);
  }
  return ans;
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) scanf("%lf%lf", &p[i].x, &p[i].y);
  for (int i = 1; i <= n; i++) swap(p[rand() % n + 1], p[rand() % n + 1]);
  o = p[1];
  for (int i = 1; i <= n; i++) {
    if (dis(o, p[i]) < r || cmp(dis(o, p[i]), r)) continue;
    o.x = (p[i].x + p[1].x) / 2;
    o.y = (p[i].y + p[1].y) / 2;
    r = dis(p[i], p[1]) / 2;
    for (int j = 2; j < i; j++) {
      if (dis(o, p[j]) < r || cmp(dis(o, p[j]), r)) continue;
      o.x = (p[i].x + p[j].x) / 2;
      o.y = (p[i].y + p[j].y) / 2;
      r = dis(p[i], p[j]) / 2;
      for (int k = 1; k < j; k++) {
        if (dis(o, p[k]) < r || cmp(dis(o, p[k]), r)) continue;
        o = geto(p[i], p[j], p[k]);
        r = dis(o, p[i]);
      }
    }
  }
  printf("%.10lf\n%.10lf %.10lf", r, o.x, o.y);
  return 0;
}
```

练习

最小圆覆盖[1]

「HNOI2012」 射箭[2]

CodeForces 442E[3]

参考资料与扩展阅读

http://www.doc88.com/p-007257893177.html[4]

https://www.cnblogs.com/aininot260/p/9635757.html[5]

https://wenku.baidu.com/view/162699d63186bceb19e8bbe6.html[6]

https://blog.csdn.net/u014609452/article/details/62039612[7]

参考资料与注释

[1] 最小圆覆盖

[2] 「HNOI2012」射箭

[3] CodeForces 442E

[4] http://www.doc88.com/p-007257893177.html

[5] https://www.cnblogs.com/aininot260/p/9635757.html

[6] https://wenku.baidu.com/view/162699d63186bceb19e8bbe6.html

[7] https://blog.csdn.net/u014609452/article/details/62039612

12.13 反演变换

Authors: hyp1231, 383494

引入

反演变换适用于题目中存在多个圆/直线之间的相切关系的情况。利用反演变换的性质，在反演空间求解问题，可以大幅简化计算。

定义

给定反演中心点 O 和反演半径 R。若平面上点 P 和 P' 满足：

- 点 P' 在射线 $\overrightarrow{OP}$ 上
- $|OP| \cdot |OP'| = R^2$

则称点 P 和点 P' 互为反演点。

解释

下图所示即为平面上一点 P 的反演：

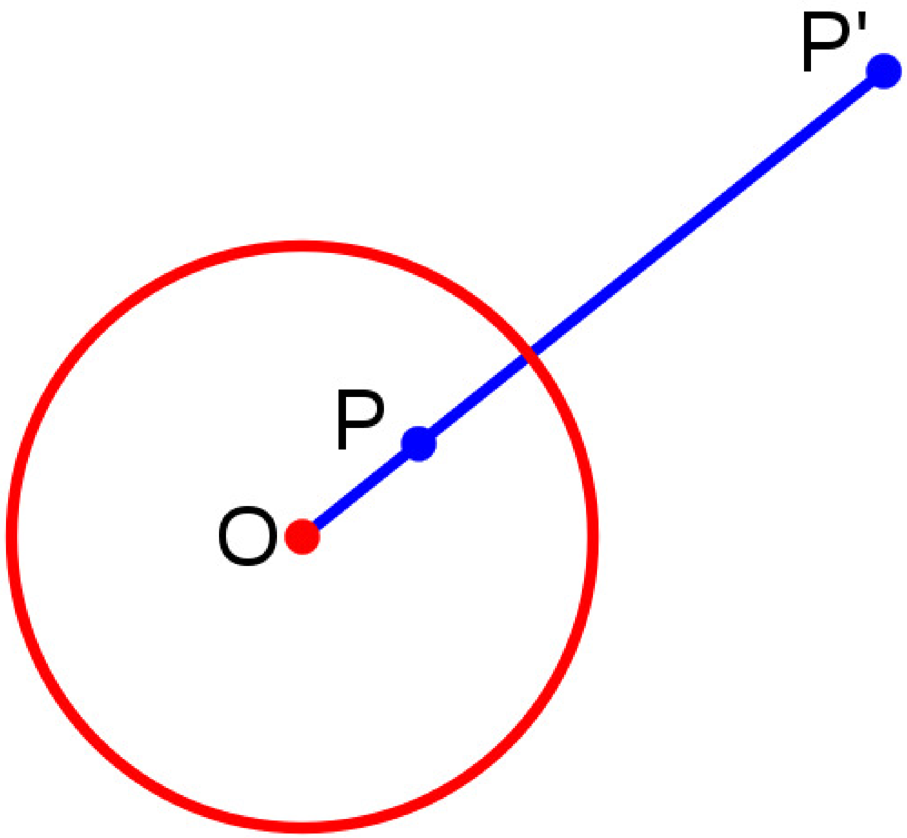


图 12.37 Inv1

性质

1. 圆 O 外的点的反演点在圆 O 内，反之亦然；圆 O 上的点的反演点为其自身。
2. 不过点 O 的圆 A，其反演图形也是不过点 O 的圆。
   - 记圆 A 半径为 r_1，其反演图形圆 B 半径为 r_2，则有：

     $$r_2 = \frac{1}{2}\left(\frac{1}{|OA| - r_1} - \frac{1}{|OA| + r_1}\right) R^2$$

     证明：

     根据反演变换定义：

     $$|OC| \cdot |OC'| = (|OA| + r_1) \cdot (|OB| - r_2) = R^2$$
     $$|OD| \cdot |OD'| = (|OA| - r_1) \cdot (|OB| + r_2) = R^2$$

     消掉 $|OB|$，解方程即可。
   - 记点 O 坐标为 (x_0, y_0)，点 A 坐标为 x_1, y_1，点 B 坐标为 x_2, y_2，则有：

     $$x_2 = x_0 + \frac{|OB|}{|OA|}(x_1 - x_0)$$
     $$y_2 = y_0 + \frac{|OB|}{|OA|}(y_1 - y_0)$$

     其中 $|OB|$ 可在上述求 r_2 的过程中计算得到。
3. 过点 O 的圆 A，其反演图形是不过点 O 的直线。

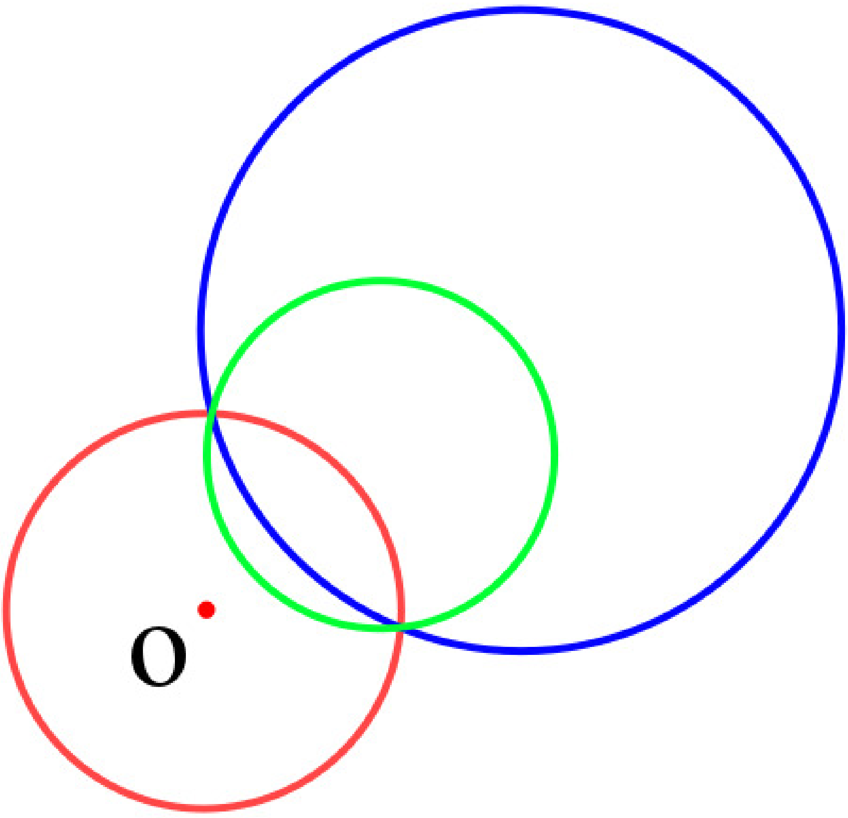


图 12.38 Inv2

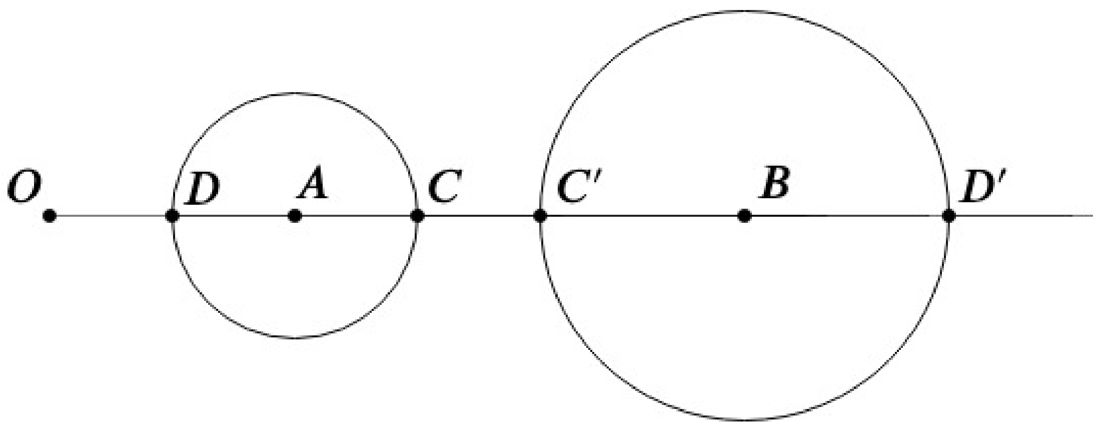


图 12.39 Inv3

> note
>
> 为什么是一条直线呢？因为圆 A 上无限接近点 O 的一点，其反演点离点 O 无限远。

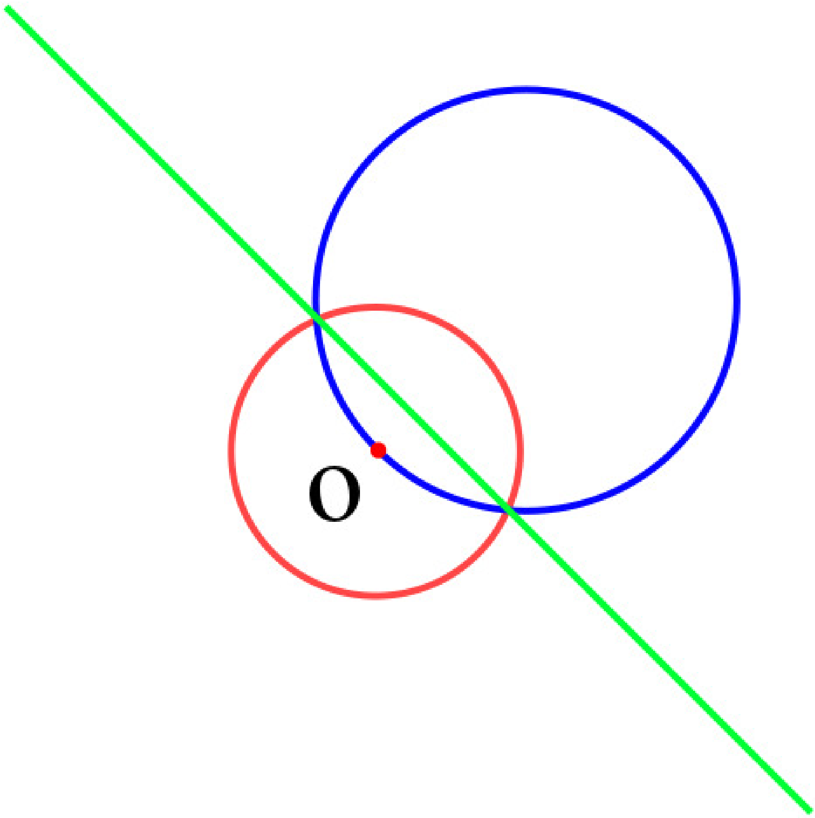


图 12.40 Inv4

4. 两个图形相切且存在不为点 O 的切点，则他们的反演图形也相切。

例题

「ICPC 2013 杭州赛区 」Problem of Apollonius[1]

题目大意

求过两圆外一点，且与两圆相切的所有的圆。

解法

首先考虑解析几何解法，似乎很难求解。

考虑以需要经过的点为反演中心进行反演（反演半径任意），所求的圆的反演图形是一条直线（应用性质 3），且与题目给出两圆的反演图形（性质 2）相切（性质 4）。

于是题目经过反演变换后转变为：求两圆的所有公切线。

求出公切线后，反演回原平面即可。

" 示例代码"

```
#include <algorithm>
#include <cmath>
```

```
#include <cstdio>
#include <cstring>
#include <iostream>
#include <vector>
using namespace std;

const double EPS = 1e-8;       // 精度系数
const double PI = acos(-1.0);  // π
const int N = 4;

struct Point {
  double x, y;

  Point(double x = 0, double y = 0) : x(x), y(y) {}

  const bool operator<(Point A) const { return x == A.x ? y < A.y : x < A.x; }
};  // 点的定义

typedef Point Vector;  // 向量的定义

Vector operator+(Vector A, Vector B) {
  return Vector(A.x + B.x, A.y + B.y);
}  // 向量加法

Vector operator-(Vector A, Vector B) {
  return Vector(A.x - B.x, A.y - B.y);
}  // 向量减法

Vector operator*(Vector A, double p) {
  return Vector(A.x * p, A.y * p);
}  // 向量数乘

Vector operator/(Vector A, double p) {
  return Vector(A.x / p, A.y / p);
}  // 向量数除

int dcmp(double x) {
  if (fabs(x) < EPS)
    return 0;
  else
    return x < 0 ? -1 : 1;
}  // 与 0 的关系

double Dot(Vector A, Vector B) { return A.x * B.x + A.y * B.y; }  // 向量点乘

double Length(Vector A) { return sqrt(Dot(A, A)); }  // 向量长度

double Cross(Vector A, Vector B) { return A.x * B.y - A.y * B.x; }  // 向量叉乘

Point GetLineProjection(Point P, Point A, Point B) {
  Vector v = B - A;
  return A + v * (Dot(v, P - A) / Dot(v, v));
}  // 点在直线上投影
```

```
struct Circle {
  Point c;
  double r;

  Circle() : c(Point(0, 0)), r(0) {}

  Circle(Point c, double r = 0) : c(c), r(r) {}

  Point point(double a) {
    return Point(c.x + cos(a) * r, c.y + sin(a) * r);
  }  // 输入极角返回点坐标
};  // 圆

// a[i] 和 b[i] 分别是第 i 条切线在圆 A 和圆 B 上的切点
int getTangents(Circle A, Circle B, Point* a, Point* b) {
  int cnt = 0;
  if (A.r < B.r) {
    swap(A, B);
    swap(a, b);
  }
  double d2 =
      (A.c.x - B.c.x) * (A.c.x - B.c.x) + (A.c.y - B.c.y) * (A.c.y - B.c.y);
  double rdiff = A.r - B.r;
  double rsum = A.r + B.r;
  if (dcmp(d2 - rdiff * rdiff) < 0) return 0;  // 内含

  double base = atan2(B.c.y - A.c.y, B.c.x - A.c.x);
  if (dcmp(d2) == 0 && dcmp(A.r - B.r) == 0) return -1;  // 无限多条切线
  if (dcmp(d2 - rdiff * rdiff) == 0) {  // 内切，一条切线
    a[cnt] = A.point(base);
    b[cnt] = B.point(base);
    ++cnt;
    return 1;
  }
  // 有外公切线
  double ang = acos(rdiff / sqrt(d2));
  a[cnt] = A.point(base + ang);
  b[cnt] = B.point(base + ang);
  ++cnt;
  a[cnt] = A.point(base - ang);
  b[cnt] = B.point(base - ang);
  ++cnt;
  if (dcmp(d2 - rsum * rsum) == 0) {  // 一条内公切线
    a[cnt] = A.point(base);
    b[cnt] = B.point(PI + base);
    ++cnt;
  } else if (dcmp(d2 - rsum * rsum) > 0) {  // 两条内公切线
    double ang = acos(rsum / sqrt(d2));
    a[cnt] = A.point(base + ang);
    b[cnt] = B.point(PI + base + ang);
    ++cnt;
    a[cnt] = A.point(base - ang);
    b[cnt] = B.point(PI + base - ang);
    ++cnt;
```

```
  }
  return cnt;
} // 两圆公切线返回切线的条数，-1 表示无穷多条切线

Circle Inversion_C2C(Point O, double R, Circle A) {
  double OA = Length(A.c - O);
  double RB = 0.5 * ((1 / (OA - A.r)) - (1 / (OA + A.r))) * R * R;
  double OB = OA * RB / A.r;
  double Bx = O.x + (A.c.x - O.x) * OB / OA;
  double By = O.y + (A.c.y - O.y) * OB / OA;
  return Circle(Point(Bx, By), RB);
} // 点 O 在圆 A 外，求圆 A 的反演圆 B，R 是反演半径

Circle Inversion_L2C(Point O, double R, Point A, Vector v) {
  Point P = GetLineProjection(O, A, A + v);
  double d = Length(O - P);
  double RB = R * R / (2 * d);
  Vector VB = (P - O) / d * RB;
  return Circle(O + VB, RB);
} // 直线反演为过 O 点的圆 B，R 是反演半径

bool theSameSideOfLine(Point A, Point B, Point S, Vector v) {
  return dcmp(Cross(A - S, v)) * dcmp(Cross(B - S, v)) > 0;
} // 返回 true 如果 A B 两点在直线同侧

int main() {
  int T;
  scanf("%d", &T);
  while (T--) {
    Circle A, B;
    Point P;
    scanf("%lf%lf%lf", &A.c.x, &A.c.y, &A.r);
    scanf("%lf%lf%lf", &B.c.x, &B.c.y, &B.r);
    scanf("%lf%lf", &P.x, &P.y);
    Circle NA = Inversion_C2C(P, 10, A);
    Circle NB = Inversion_C2C(P, 10, B);
    Point LA[N], LB[N];
    Circle ansC[N];
    int q = getTangents(NA, NB, LA, LB), ans = 0;
    for (int i = 0; i < q; ++i)
      if (theSameSideOfLine(NA.c, NB.c, LA[i], LB[i] - LA[i])) {
        if (!theSameSideOfLine(P, NA.c, LA[i], LB[i] - LA[i])) continue;
        ansC[ans++] = Inversion_L2C(P, 10, LA[i], LB[i] - LA[i]);
      }
    printf("%d\n", ans);
    for (int i = 0; i < ans; ++i) {
      printf("%.8f %.8f %.8f\n", ansC[i].c.x, ansC[i].c.y, ansC[i].r);
    }
  }

  return 0;
}
```

练习

「ICPC 2017 南宁赛区网络赛」Finding the Radius for an Inserted Circle[2]

「CCPC 2017 网络赛」The Designer[3]

参考资料与拓展阅读

- Inversive geometry - Wikipedia[4]
- 圆的反演变换 - ACdreamers 的博客[5]

参考资料与注释

[1] 「ICPC 2013 杭州赛区」Problem of Apollonius

[2] 「ICPC 2017 南宁赛区网络赛」Finding the Radius for an Inserted Circle

[3] 「CCPC 2017 网络赛」The Designer

[4] Inversive geometry - Wikipedia

[5] 圆的反演变换 - ACdreamers 的博客

12.14 计算几何杂项

Authors: Ir1d

第 13 章

杂项

13.1 杂项简介

这个板块主要介绍的是一些难以分类的算法及 OI 相关知识。

13.2 离散化

> **Authors:** GavinZhengOI, PlanariaIce

简介

离散化是一种数据处理的技巧，本质上可以看成是一种 哈希，其保证数据在哈希以后仍然保持原来的 全 / 偏序 关系。

通俗地讲就是当有些数据因为本身很大或者类型不支持，自身无法作为数组的下标来方便地处理，而影响最终结果的只有元素之间的相对大小关系时，我们可以将原来的数据按照排名来处理问题，即离散化。

用来离散化的可以是大整数、浮点数、字符串等等。

实现

将一个数组离散化，并进行查询是比较常用的应用场景。

方法一

通常原数组中会有重复的元素，一般把相同的元素离散化为相同的数据。

方法如下：

1. 创建原数组的副本。
2. 将副本中的值从小到大排序。
3. 将排序好的副本去重。
4. 查找原数组的每一个元素在副本中的位置，位置即为排名，将其作为离散化后的值。

```
// arr[i] 为初始数组，下标范围为 [1, n]
```

```
for (int i = 1; i <= n; ++i)  // step 1
  tmp[i] = arr[i];
std::sort(tmp + 1, tmp + n + 1);                          // step 2
int len = std::unique(tmp + 1, tmp + n + 1) - (tmp + 1);  // step 3
for (int i = 1; i <= n; ++i)                              // step 4
  arr[i] = std::lower_bound(tmp + 1, tmp + len + 1, arr[i]) - tmp;
```

参考实现中使用的 STL 算法可参考 STL 算法。

同样地，我们也可以对 std::vector 进行离散化：

```
// std::vector<int> arr;
std::vector<int> tmp(arr);  // tmp 是 arr 的一个副本
std::sort(tmp.begin(), tmp.end());
tmp.erase(std::unique(tmp.begin(), tmp.end()), tmp.end());
for (int i = 0; i < n; ++i)
  arr[i] = std::lower_bound(tmp.begin(), tmp.end(), arr[i]) - tmp.begin();
```

方法二

根据题目要求，有时候会把相同的元素跟据输入顺序离散化为不同的数据。

此时再用 `std::lower_bound()` 函数实现就有些困难了，需要换一种思路：

1. 创建原数组的副本，同时记录每个元素出现的位置。
2. 将副本按值从小到大排序，当值相同时，按出现顺序从小到大排序。
3. 将离散化后的数字放回原数组。

```
struct Data {
  int idx, val;

  bool operator<(const Data& o) const {
    if (val == o.val)
      return idx < o.idx;  // 当值相同时，先出现的元素离散化后的值更小
    return val < o.val;
  }
} tmp[maxn];  // 也可以使用 std::pair

for (int i = 1; i <= n; ++i) tmp[i] = (Data){i, arr[i]};
std::sort(tmp + 1, tmp + n + 1);
for (int i = 1; i <= n; ++i) arr[tmp[i].idx] = i;
```

复杂度

对于方法一，去重复杂度为 $O(n)$，排序复杂度为 $O(n\log n)$，最后的 n 次查找复杂度为 $O(n\log n)$。

对于方法二，排序复杂度为 $O(n\log n)$。

故两种方法的总时间复杂度都为 $O(n\log n)$。

空间复杂度为 $O(n)$。

习题

- [HAOI2014] 贴海报[1]
- [NOI2015] 程序自动分析[2]

参考资料与注释

[1] [HAOI2014] 贴海报

[2] [NOI2015] 程序自动分析

13.3 双指针

本页面将简要介绍双指针。

引入

双指针是一种简单而又灵活的技巧和思想，单独使用可以轻松解决一些特定问题，和其他算法结合也能发挥多样的用处。

双指针顾名思义，就是同时使用两个指针，在序列、链表结构上指向的是位置，在树、图结构中指向的是节点，通过或同向移动，或相向移动来维护、统计信息。

接下来我们来看双指针的几个具体使用方法。

维护区间信息

如果不和其他数据结构结合使用，双指针维护区间信息的最简单模式就是维护具有一定单调性，新增和删去一个元素都很方便处理的信息，就比如正数的和、正整数的积等等。

例题 1

" **例题 1** leetcode 713. **乘积小于 K 的子数组**[1]"

给定一个长度为 n 的正整数数组 $nums$ 和整数 k，找出该数组内乘积小于 k 的连续子数组的个数。$1 \leq n \leq 3 \times 10^4, 1 \leq nums[i] \leq 1000, 0 \leq k \leq 10^6$

过程

设两个指针分别为 l, r，另外设置一个变量 tmp 记录 $[l, r]$ 内所有数的乘积。最开始 l, r 都在最左面，先向右移动 r，直到第一次发现 $tmp \geq k$，这时就固定 r，右移 l，直到 $tmp < k$。那么对于每个 r，l 是它能延展到的左边界，由于正整数乘积的单调性，此时以 r 为右端点的满足题目条件的区间个数为 $r - l + 1$ 个。

实现

```
int numSubarrayProductLessThanK(vector<int>& nums, int k) {
  long long ji = 1ll, ans = 0;
  int l = 0;
  for (int i = 0; i < nums.size(); ++i) {
    ji *= nums[i];
    while (l <= i && ji >= k) ji /= nums[l++];
    ans += i - l + 1;
  }
  return ans;
}
```

使用双指针维护区间信息也可以与其他数据结构比如差分、单调队列、线段树、主席树等等结合使用。另外将双指针技巧融入算法的还有莫队，莫队中将询问离线排序后，一般也都是用两个指针记录当前要处理的区间，随着指针一步步移动逐渐更新区间信息。

例题 2

接下来看一道在树上使用双指针并结合树上差分的例题：

> " **例题 2** luogu P3066 Running Away From the Barn G[2]"
>
> 给定一颗 n 个点的有根树，边有边权，节点从 1 至 n 编号，1 号节点是这棵树的根。再给出一个参数 t，对于树上的每个节点 u，请求出 u 的子树中有多少节点满足该节点到 u 的距离不大于 t。数据范围：$1 \le n \le 2\times 10^5, 1 \le t \le 10^{18}, 1 \le p_i < i, 1 \le w_i \le 10^{12}$

过程

从根开始用 dfs 遍历整棵树，使用一个栈来记录根到当前节点的树链，设一个指针 u 指向当前节点，另一个指针 p 指向与 u 距离不大于 t 的节点中深度最小的节点。记录到根的距离，每次二分查找确定 p。此时 u 对 p 到 u 路径上的所有节点都有一个贡献，可以用树上差分来记录。

注意不能直接暴力移动 p，否则时间复杂度可能会退化至 $O(n^2)$。

子序列匹配

> " **例题 3** leetcode 524. **通过删除字母匹配到字典里最长单词**[3]"
>
> 给定一个字符串 s 和一个字符串数组 $dictionary$ 作为字典，找出并返回字典中最长的字符串，该字符串可以通过删除 s 中的某些字符得到。

过程

此类问题需要将字符串 s 与 t 进行匹配，判断 t 是否为 s 的子序列。解决这种问题只需先将两个指针一个 i 放在 s 开始位置，一个 j 放在 t 开始位置，如果 $s[i]=t[j]$ 说明 t 的第 j 位已经在 s 中找到了第一个对应，可以进而检测后面的部分了，那么 i 和 j 同时加一。如果上述等式不成立，则 t 的第 j 位仍然没有被匹配上，所以只给 i 加一，在 s 的后面部分再继续寻找。最后，如果 j 已经移到了超尾位置，说明整个字符串都可以被匹配上，也就是 t 是 s 的一个子序列，否则不是。

实现

```cpp
string findLongestWord(string s, vector<string>& dictionary) {
  sort(dictionary.begin(), dictionary.end());
  int mx = 0, r = 0;
  string ans = "";
  for (int i = dictionary.size() - 1; i >= 0; i--) {
    r = 0;
    for (int j = 0; j < s.length(); ++j) {
      if (s[j] == dictionary[i][r]) r++;
    }
    if (r == dictionary[i].length()) {
      if (r >= mx) {
        mx = r;
        ans = dictionary[i];
```

```
      }
    }
  }
  return ans;
}
```

这种两个指针指向不同对象然后逐步进行比对的方法还可以用在一些 dp 中。

利用序列有序性

很多时候在序列上使用双指针之所以能够正确地达到目的，是因为序列的某些性质，最常见的就是利用序列的有序性。

" **例题** 4 leetcode 167. **两数之和 II - 输入有序数组**[4]"

给定一个已按照**升序排列**的整数数组 `numbers`，请你从数组中找出两个数满足相加之和等于目标数 `target`。

过程

这种问题也是双指针的经典应用了，虽然二分也很方便，但时间复杂度上多一个 $\log n$，而且代码不够简洁。

接下来介绍双指针做法：既然要找到两个数，且这两个数不能在同一位置，那其位置一定是一左一右。由于两数之和固定，那么两数之中的小数越大，大数越小。考虑到这些性质，那我们不妨从两边接近它们。

首先假定答案就是 1 和 n，如果发现 $num[1]+num[n] > target$，说明我们需要将其中的一个元素变小，而 $num[1]$ 已经不能再变小了，所以我们把指向 n 的指针减一，让大数变小。

同理如果发现 $num[1]+num[n] < target$，说明我们要将其中的一个元素变大，但 $num[n]$ 已经不能再变大了，所以将指向 1 的指针加一，让小数变大。

推广到一般情形，如果此时我们两个指针分别指在 l, r 上，且 $l < r$，如果 $num[l]+num[r] > target$，就将 r 减一，如果 $num[l]+num[r] < target$，就将 l 加一。这样 l 不断右移，r 不断左移，最后两者各逼近到一个答案。

实现

```
vector<int> twoSum(vector<int>& numbers, int target) {
  int r = numbers.size() - 1, l = 0;
  vector<int> ans;
  ans.clear();
  while (l < r) {
    if (numbers[l] + numbers[r] > target)
      r--;
    else if (numbers[l] + numbers[r] == target) {
      ans.push_back(l + 1), ans.push_back(r + 1);
      return ans;
    } else
      l++;
  }
  return ans;
}
```

在归并排序中，在 $O(n+m)$ 时间内合并两个有序数组，也是保证数组的有序性条件下使用的双指针法。

在单向链表中找环

过程

在单向链表中找环也是有多种办法，不过快慢双指针方法是其中最为简洁的方法之一，接下来介绍这种方法。

首先两个指针都指向链表的头部，令一个指针一次走一步，另一个指针一次走两步，如果它们相遇了，证明有环，否则无环，时间复杂度 $O(n)$。

如果有环的话，怎么找到环的起点呢？

我们列出式子来观察一下，设相遇时，慢指针一共走了 k 步，在环上走了 l 步（快慢指针在环上相遇时，慢指针一定没走完一圈）。快指针走了 $2k$ 步，设环长为 C，则有

$$2k = n \times C + l + (k - l) \tag{13.1}$$
$$k = n \times C \tag{13.2}$$
$$\tag{13.3}$$

第一次相遇时 n 取最小正整数 1。也就是说 $k = C$。那么利用这个等式，可以在两个指针相遇后，将其中一个指针移到表头，让两者都一步一步走，再度相遇的位置即为环的起点。

习题

leetcode 15. 三数之和[5]

leetcode 1438. 绝对差不超过限制的最长连续子数组[6]

参考资料与注释

[1] leetcode 713. 乘积小于 K 的子数组

[2] luogu P3066 Running Away From the Barn G

[3] leetcode 524. 通过删除字母匹配到字典里最长单词

[4] leetcode 167. 两数之和 II - 输入有序数组

[5] leetcode 15. 三数之和

[6] leetcode 1438. 绝对差不超过限制的最长连续子数组

13.4 离线算法

13.4.1 离线算法简介

13.4.2 CDQ 分治

本页面将介绍 CDQ 分治。

简介

CDQ 分治是一种思想而不是具体的算法，与 动态规划 类似。目前这个思想的拓展十分广泛，依原理与写法的不同，大致分为三类：

- 解决和点对有关的问题。
- 1D 动态规划的优化与转移。
- 通过 CDQ 分治，将一些动态问题转化为静态问题。

CDQ 分治的思想最早由 IOI2008 金牌得主陈丹琦在高中时整理并总结，它也因此得名。[1]

解决和点对有关的问题

这类问题多数类似于「给定一个长度为 n 的序列，统计有一些特性的点对 (i,j) 的数量/找到一对点 (i,j) 使得一些函数的值最大」。

CDQ 分治解决这类问题的算法流程如下：

1. 找到这个序列的中点 mid；
2. 将所有点对 (i,j) 划分为 3 类：
   (a) $1 \leq i \leq mid, 1 \leq j \leq mid$ 的点对；
   (b) $1 \leq i \leq mid, mid+1 \leq j \leq n$ 的点对；
   (c) $mid+1 \leq i \leq n, mid+1 \leq j \leq n$ 的点对。
3. 将 $(1,n)$ 这个序列拆成两个序列 $(1,mid)$ 和 $(mid+1,n)$。此时第一类点对和第三类点对都在这两个序列之中；
4. 递归地处理这两类点对；
5. 设法处理第二类点对。

可以看到 CDQ 分治的思想就是不断地把点对通过递归的方式分给左右两个区间。

在实际应用时，我们通常使用一个函数 `solve(l,r)` 处理 $l \leq i \leq r, l \leq j \leq r$ 的点对。上述算法流程中的递归部分便是通过 `solve(l,mid)` 与 `solve(mid,r)` 来实现的。剩下的第二类点对则需要额外设计算法解决。

例题

"三维偏序[2]"

给定一个序列，每个点有 a_i, b_i, c_i 三个属性，试求：这个序列里有多少对点对 (i,j) 满足 $a_j \leq a_i$ 且 $b_j \leq b_i$ 且 $c_j \leq c_i$ 且 $j \neq i$。

解题思路

三维偏序是 CDQ 分治的经典问题。

题目要求统计序列里点对的个数，那试一下用 CDQ 分治。

首先将序列按 a 排序。

假设我们现在写好了 `solve(l,r)`，并且通过递归搞定了 `solve(l,mid)` 和 `solve(mid+1,r)`。现在我们要做的，就是统计满足 $l \leq i \leq mid, mid+1 \leq j \leq r$ 的点对 (i,j) 中，有多个点对还满足 $a_i < a_j, b_i < b_j, c_i < c_j$ 的限制条件。

稍微思考一下就会发现，那个 $a_i < a_j$ 的限制条件没啥用了：已经将序列按 a 排序，则 $a_i < a_j$ 可转化为 $i < j$。既然 i 比 mid 小，j 比 mid 大，那 i 肯定比 j 要小。现在还剩下两个限制条件：$b_i < b_j$ 与 $c_i < c_j$，根据这个限制条件我们就可以枚举 j，求出有多少个满足条件的 i。

为了方便枚举，我们把 (l,mid) 和 $(mid+1,r)$ 中的点全部按照 b 的值从小到大排个序。之后我们依次枚举每一个 j，把所有 $b_i < b_j$ 的点 i 全部插入到某种数据结构里（这里我们选择 树状数组）。此时只要查询树状数组

里有多少个点的 c 值是小于 c_j 的，我们就求出了对于这个点 j，有多少个 i 可以合法匹配它了。

当我们插入一个 c 值等于 x 的点时，我们就令树状数组的 x 这个位置单点 $+1$，而查询树状数组里有多少个点小于 x 的操作实际上就是在求 前缀和，只要我们事先对于所有的 c 值做了 离散化，我们的复杂度就是对的。

对于每一个 j，我们都需要将所有 $b_i < b_j$ 的点 i 插入树状数组中。由于所有的 i 和 j 都已事先按照 b 值排好序，这样的话只要以双指针的方式在树状数组里插入点，则对树状数组的插入操作就能从 $O(n^2)$ 次降到 $O(n)$ 次。

通过这样一个算法流程，我们就用 $O(n\log n)$ 的时间处理完了关于第二类点对的信息了。此时算法的时间复杂度是 $T(n)=T(\lfloor\frac{n}{2}\rfloor)+T(\lceil\frac{n}{2}\rceil)+O(n\log n)=O(n\log^2 n)$。

示例代码

```cpp
#include <algorithm>
#include <cstdio>

const int maxN = 1e5 + 10;
const int maxK = 2e5 + 10;

int n, k;

struct Element {
  int a, b, c;
  int cnt;
  int res;

  bool operator!=(Element other) {
    if (a != other.a) return true;
    if (b != other.b) return true;
    if (c != other.c) return true;
    return false;
  }
};

Element e[maxN];
Element ue[maxN];
int m, t;
int res[maxN];

struct BinaryIndexedTree {
  int node[maxK];

  int lowbit(int x) { return x & -x; }

  void Add(int pos, int val) {
    while (pos <= k) {
      node[pos] += val;
      pos += lowbit(pos);
    }
    return;
  }

  int Ask(int pos) {
    int res = 0;
    while (pos) {
```

```
      res += node[pos];
      pos -= lowbit(pos);
    }
    return res;
  }
} BIT;

bool cmpA(Element x, Element y) {
  if (x.a != y.a) return x.a < y.a;
  if (x.b != y.b) return x.b < y.b;
  return x.c < y.c;
}

bool cmpB(Element x, Element y) {
  if (x.b != y.b) return x.b < y.b;
  return x.c < y.c;
}

void CDQ(int l, int r) {
  if (l == r) return;
  int mid = (l + r) / 2;
  CDQ(l, mid);
  CDQ(mid + 1, r);
  std::sort(ue + l, ue + mid + 1, cmpB);
  std::sort(ue + mid + 1, ue + r + 1, cmpB);
  int i = l;
  int j = mid + 1;
  while (j <= r) {
    while (i <= mid && ue[i].b <= ue[j].b) {
      BIT.Add(ue[i].c, ue[i].cnt);
      i++;
    }
    ue[j].res += BIT.Ask(ue[j].c);
    j++;
  }
  for (int k = l; k < i; k++) BIT.Add(ue[k].c, -ue[k].cnt);
  return;
}

int main() {
  scanf("%d%d", &n, &k);
  for (int i = 1; i <= n; i++) scanf("%d%d%d", &e[i].a, &e[i].b, &e[i].c);
  std::sort(e + 1, e + n + 1, cmpA);
  for (int i = 1; i <= n; i++) {
    t++;
    if (e[i] != e[i + 1]) {
      m++;
      ue[m].a = e[i].a;
      ue[m].b = e[i].b;
      ue[m].c = e[i].c;
      ue[m].cnt = t;
      t = 0;
    }
  }
```

```
  CDQ(1, m);
  for (int i = 1; i <= m; i++) res[ue[i].res + ue[i].cnt - 1] += ue[i].cnt;
  for (int i = 0; i < n; i++) printf("%d\n", res[i]);
  return 0;
}
```

"CQOI2011 动态逆序对[3]"

对于序列 a，它的逆序对数定义为集合 $\{(i,j)|i<j \wedge a_i > a_j\}$ 中的元素个数。

现在给出 $1 \sim n$ 的一个排列，按照某种顺序依次删除 m 个元素，你的任务是在每次删除一个元素之前统计整个序列的逆序对数。

示例代码

```
// 仔细推一下就是和三维偏序差不多的式子了，基本就是一个三维偏序的板子
#include <algorithm>
#include <cstdio>
using namespace std;
typedef long long ll;
int n;
int m;

struct treearray {
  int ta[200010];

  void ub(int& x) { x += x & (-x); }

  void db(int& x) { x -= x & (-x); }

  void c(int x, int t) {
    for (; x <= n + 1; ub(x)) ta[x] += t;
  }

  int sum(int x) {
    int r = 0;
    for (; x > 0; db(x)) r += ta[x];
    return r;
  }
} ta;

struct data_ {
  int val;
  int del;
  int ans;
} a[100010];

int rv[100010];
ll res;

bool cmp1(const data_& a, const data_& b) {
  return a.val < b.val;
}  // 重写两个比较
```

```
bool cmp2(const data_& a, const data_& b) { return a.del < b.del; }

void solve(int l, int r) {  // 底下是具体的式子，套用
  if (r - l == 1) {
    return;
  }
  int mid = (l + r) / 2;
  solve(l, mid);
  solve(mid, r);
  int i = l + 1;
  int j = mid + 1;
  while (i <= mid) {
    while (a[i].val > a[j].val && j <= r) {
      ta.c(a[j].del, 1);
      j++;
    }
    a[i].ans += ta.sum(m + 1) - ta.sum(a[i].del);
    i++;
  }
  i = l + 1;
  j = mid + 1;
  while (i <= mid) {
    while (a[i].val > a[j].val && j <= r) {
      ta.c(a[j].del, -1);
      j++;
    }
    i++;
  }
  i = mid;
  j = r;
  while (j > mid) {
    while (a[j].val < a[i].val && i > l) {
      ta.c(a[i].del, 1);
      i--;
    }
    a[j].ans += ta.sum(m + 1) - ta.sum(a[j].del);
    j--;
  }
  i = mid;
  j = r;
  while (j > mid) {
    while (a[j].val < a[i].val && i > l) {
      ta.c(a[i].del, -1);
      i--;
    }
    j--;
  }
  sort(a + l + 1, a + r + 1, cmp1);
  return;
}

int main() {
  scanf("%d%d", &n, &m);
```

```
  for (int i = 1; i <= n; i++) {
    scanf("%d", &a[i].val);
    rv[a[i].val] = i;
  }
  for (int i = 1; i <= m; i++) {
    int p;
    scanf("%d", &p);
    a[rv[p]].del = i;
  }
  for (int i = 1; i <= n; i++) {
    if (a[i].del == 0) a[i].del = m + 1;
  }
  for (int i = 1; i <= n; i++) {
    res += ta.sum(n + 1) - ta.sum(a[i].val);
    ta.c(a[i].val, 1);
  }
  for (int i = 1; i <= n; i++) {
    ta.c(a[i].val, -1);
  }
  solve(0, n);
  sort(a + 1, a + n + 1, cmp2);
  for (int i = 1; i <= m; i++) {
    printf("%lld\n", res);
    res -= a[i].ans;
  }
  return 0;
}
```

CDQ 分治优化 1D/1D 动态规划的转移

1D/1D 动态规划指的是一类特定的 DP 问题，该类题目的特征是 DP 数组是一维的，转移是 $O(n)$ 的。如果条件良好的话，有时可以通过 CDQ 分治来把它们的时间复杂度由 $O(n^2)$ 降至 $O(n\log^2 n)$。

例如，给定一个序列，每个元素有两个属性 a，b。我们希望计算一个 DP 式子的值，它的转移方程如下：

$dp_i = 1 + \max_{j=1}^{i-1} dp_j[a_j < a_i][b_j < b_i]$

这是一个二维最长上升子序列的 DP 方程，即只有 $j < i, a_j < a_i$,$b_j < b_i$ 的点 j 可以更新点 i 的 DP 值。

直接转移显然是 $O(n^2)$ 的。以下是使用 CDQ 分治优化转移过程的讲解。

我们发现 dp_j 转移到 dp_i 这种转移关系也是一种点对间的关系，所以我们用类似 CDQ 分治处理点对关系的方式来处理它。

这个转移过程相对来讲比较套路。假设现在正在处理的区间是 (l, r)，算法流程大致如下：

1. 如果 $l = r$，说明 dp_r 值已经被计算好了。直接令 dp_r++ 然后返回即可；
2. 递归使用 `solve(l,mid)`；
3. 处理所有 $l \le j \le mid$，$mid + 1 \le i \le r$ 的转移关系；
4. 递归使用 `solve(mid+1,r)`。

第三步的做法与 CDQ 分治求三维偏序差不多。处理 $l \le j \le mid$，$mid + 1 \le i \le r$ 的转移关系的时候，我们会发现已经不用管 $j < i$ 这个限制条件了。因此，我们依然先将所有的点 i 和点 j 按 a 值进行排序处理，然后用双指针的方式将 j 点插入到树状数组里，最后查一下前缀最大值更新一下 dp_i 就可以了。

转移过程的正确性证明

该 CDQ 写法和处理点对间关系的 CDQ 写法最大的不同就是处理 $l \le j \le mid$，$mid+1 \le i \le r$ 的点对这一部分。处理点对间关系的 CDQ 写法中，这一部分放到哪里都是可以的。但是，在用 CDQ 分治优化 DP 的时候，这个流程却必须夹在 $solve(l, mid)$,$solve(mid+1, r)$ 的中间。原因是 DP 的转移是**有序的**，它必须满足两个条件，否则就是不对的：

1. 用来计算 dp_i 的所有 dp_j 值都必须是已经计算完毕的，不能存在「半成品」；
2. 用来计算 dp_i 的所有 dp_j 值都必须能更新到 dp_i，不能存在没有更新到的 dp_j 值。

上述两个条件可能在 $O(n^2)$ 暴力的时候是相当容易满足的，但是使用 CDQ 分治后，转移顺序很显然已经乱掉了，所以有必要考察转移的正确性。

CDQ 分治的递归树如下所示。

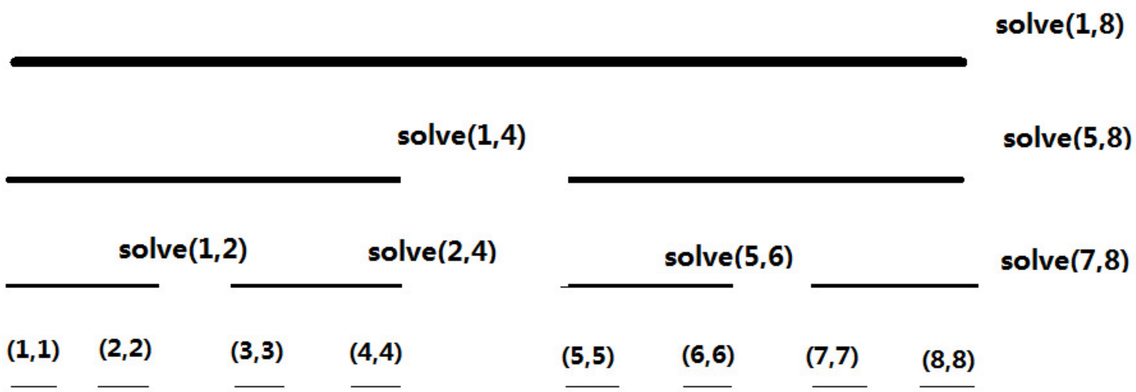


图 13.1 CDQ 分治的递归树

执行刚才的算法流程的话，以 8 这个点为例，它的 DP 值是在 `solve(1,8)`、`solve(5,8)`、`solve(7,8)` 这 3 个函数中更新完成的，而三次用来更新它的点分别是 $(1,4)$、$(5,6)$、$(7,7)$ 这三个不相交的区间；又以 5 这个点为例，它的 DP 值是在 `solve(1,4)` 函数中解决的，更新它的区间是 $(1,4)$。仔细观察就会发现，一个 i 点的 DP 值被更新了 log 次，而且，更新它的区间刚好是 $(1,i)$ 在线段树上被拆分出来的 log 个区间。因此，我们的确保证了所有合法的 j 都更新过点 i，满足第 2 个条件。

接着分析我们算法的执行流程：

1. 第一个结束的函数是 `solve(1,1)`。此时我们发现 dp_1 的值已经计算完毕了；
2. 第一个执行转移过程的函数是 `solve(1,2)`。此时我们发现 dp_2 的值已经被转移好了；
3. 第二个结束的函数是 `solve(2,2)`。此时我们发现 dp_2 的值已经计算完毕了；
4. 接下来 `solve(1,2)` 结束，$(1,2)$ 这段区间的 dp 值均被计算好；
5. 下一个执行转移流程的函数是 `solve(1,4)`。这次转移结束之后我们发现 dp_3 的值已经被转移好了；
6. 接下来结束的函数是 `solve(3,3)`。我们会发现 dp_3 的 dp 值被计算好了；
7. 接下来执行的转移是 `solve(2,4)`。此时 dp_4 在 `solve(1,4)` 中被 $(1,2)$ 转移了一次，这次又被 $(3,3)$ 转移了，因此 dp_4 的值也被转移好了；
8. `solve(4,4)` 结束，dp_4 的值计算完毕；
9. `solve(3,4)` 结束，$(3,4)$ 的值计算完毕；
10. `solve(1,4)` 结束，$(1,4)$ 的值计算完毕。
11. ……

通过模拟函数流程，我们发现一件事：每次 `solve(l,r)` 结束的时候，(l,r) 区间的 DP 值会被全部计算好。由

于我们每一次执行转移函数的时候，`solve(l,mid)` 已经结束，因此我们每一次执行的转移过程都是合法的，满足第 1 个条件。

在刚才的过程我们发现，如果将 CDQ 分治的递归树看成一颗线段树，那么 CDQ 分治就是这个线段树的**中序遍历函数**，因此我们相当于按顺序处理了所有的 DP 值，只是转移顺序被拆开了而已，所以算法是正确的。

例题

"SDOI2011 **拦截导弹**[4]"

某国为了防御敌国的导弹袭击，发展出一种导弹拦截系统。但是这种导弹拦截系统有一个缺陷：虽然它的第一发炮弹能够到达任意的高度、并且能够拦截任意速度的导弹，但是以后每一发炮弹都不能高于前一发的高度，其拦截的导弹的飞行速度也不能大于前一发。某天，雷达捕捉到敌国的导弹来袭。由于该系统还在试用阶段，所以只有一套系统，因此有可能不能拦截所有的导弹。

在不能拦截所有的导弹的情况下，我们当然要选择使国家损失最小、也就是拦截导弹的数量最多的方案。但是拦截导弹数量的最多的方案有可能有多个，如果有多个最优方案，那么我们会随机选取一个作为最终的拦截导弹行动蓝图。

我方间谍已经获取了所有敌军导弹的高度和速度，你的任务是计算出在执行上述决策时，每枚导弹被拦截掉的概率。

参考代码

```
// 一道二维最长上升子序列的题
// 为了确定某一个元素是否在最长上升子序列中可以正反跑两遍 CDQ
#include <algorithm>
#include <cstdio>
using namespace std;
typedef double db;
const int N = 1e6 + 10;

struct data_ {
  int h;
  int v;
  int p;
  int ma;
  db ca;
} a[2][N];

int n;
bool tr;

// 底下是重写比较
bool cmp1(const data_& a, const data_& b) {
  if (tr)
    return a.h > b.h;
  else
    return a.h < b.h;
}

bool cmp2(const data_& a, const data_& b) {
  if (tr)
    return a.v > b.v;
  else
```

```cpp
    return a.v < b.v;
}

bool cmp3(const data_& a, const data_& b) {
  if (tr)
    return a.p < b.p;
  else
    return a.p > b.p;
}

bool cmp4(const data_& a, const data_& b) { return a.v == b.v; }

struct treearray {
  int ma[2 * N];
  db ca[2 * N];

  void c(int x, int t, db c) {
    for (; x <= n; x += x & (-x)) {
      if (ma[x] == t) {
        ca[x] += c;
      } else if (ma[x] < t) {
        ca[x] = c;
        ma[x] = t;
      }
    }
  }

  void d(int x) {
    for (; x <= n; x += x & (-x)) {
      ma[x] = 0;
      ca[x] = 0;
    }
  }

  void q(int x, int& m, db& c) {
    for (; x > 0; x -= x & (-x)) {
      if (ma[x] == m) {
        c += ca[x];
      } else if (m < ma[x]) {
        c = ca[x];
        m = ma[x];
      }
    }
  }
} ta;

int rk[2][N];

void solve(int l, int r, int t) {  // 递归跑
  if (r - l == 1) {
    return;
  }
  int mid = (l + r) / 2;
  solve(l, mid, t);
```

```
  sort(a[t] + mid + 1, a[t] + r + 1, cmp1);
  int p = l + 1;
  for (int i = mid + 1; i <= r; i++) {
    for (; (cmp1(a[t][p], a[t][i]) || a[t][p].h == a[t][i].h) && p <= mid;
         p++) {
      ta.c(a[t][p].v, a[t][p].ma, a[t][p].ca);
    }
    db c = 0;
    int m = 0;
    ta.q(a[t][i].v, m, c);
    if (a[t][i].ma < m + 1) {
      a[t][i].ma = m + 1;
      a[t][i].ca = c;
    } else if (a[t][i].ma == m + 1) {
      a[t][i].ca += c;
    }
  }
  for (int i = l + 1; i <= mid; i++) {
    ta.d(a[t][i].v);
  }
  sort(a[t] + mid, a[t] + r + 1, cmp3);
  solve(mid, r, t);
  sort(a[t] + l + 1, a[t] + r + 1, cmp1);
}

void ih(int t) {
  sort(a[t] + 1, a[t] + n + 1, cmp2);
  rk[t][1] = 1;
  for (int i = 2; i <= n; i++) {
    rk[t][i] = (cmp4(a[t][i], a[t][i - 1])) ? rk[t][i - 1] : i;
  }
  for (int i = 1; i <= n; i++) {
    a[t][i].v = rk[t][i];
  }
  sort(a[t] + 1, a[t] + n + 1, cmp3);
  for (int i = 1; i <= n; i++) {
    a[t][i].ma = 1;
    a[t][i].ca = 1;
  }
}

int len;
db ans;

int main() {
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) {
    scanf("%d%d", &a[0][i].h, &a[0][i].v);
    a[0][i].p = i;
    a[1][i].h = a[0][i].h;
    a[1][i].v = a[0][i].v;
    a[1][i].p = i;
  }
  ih(0);
```

```
  solve(0, n, 0);
  tr = 1;
  ih(1);
  solve(0, n, 1);
  tr = 1;
  sort(a[0] + 1, a[0] + n + 1, cmp3);
  sort(a[1] + 1, a[1] + n + 1, cmp3);
  for (int i = 1; i <= n; i++) {
    len = max(len, a[0][i].ma);
  }
  printf("%d\n", len);
  for (int i = 1; i <= n; i++) {
    if (a[0][i].ma == len) {
      ans += a[0][i].ca;
    }
  }
  for (int i = 1; i <= n; i++) {
    if (a[0][i].ma + a[1][i].ma - 1 == len) {
      printf("%.5lf ", (a[0][i].ca * a[1][i].ca) / ans);
    } else {
      printf("0.00000 ");
    }
  }
  return 0;
}
```

将动态问题转化为静态问题

前两种情况使用 CDQ 分治的目的是将序列折半之后递归处理点对间的关系，来获得良好的复杂度。不过在本节中，折半的不是一般的序列，而是时间序列。

它适用于一些「需要支持做 xxx 修改然后做 xxx 询问」的数据结构题。该类题目有两个特点：

- 如果把询问 离线，所有操作会按照时间自然地排成一个序列。
- 每一个修改均与之后的询问操作息息相关。而这样的「修改 - 询问」关系一共会有 $O(n^2)$ 对。

我们可以使用 CDQ 分治对于这个操作序列进行分治，处理修改和询问之间的关系。

与处理点对关系的 CDQ 分治类似，假设正在分治的序列是 (l, r)，我们先递归地处理 (l, mid) 和 (mid, r) 之间的修改 - 询问关系，再处理所有 $l \leq i \leq mid$，$mid + 1 \leq j \leq r$ 的修改 - 询问关系，其中 i 是一个修改，j 是一个询问。

注意，如果各个修改之间是**独立**的话，我们无需处理 $l \leq i \leq mid$ 和 $mid + 1 \leq j \leq r$，以及 `solve(l,mid)` 和 `solve(mid+1,r)` 之间的时序关系（比如普通的加减法问题）。但是如果各个修改之间并不独立（比如说赋值操作），做完这个修改后，序列长什么样可能依赖于之前的序列。此时处理所有跨越 mid 的修改 - 询问关系的步骤就必须放在 `solve(l,mid)` 和 `solve(mid+1,r)` 之间。理由和 CDQ 分治优化 1D/1D 动态规划的原因是一样的：按照中序遍历序进行分治才能保证每一个修改都是严格按照时间顺序执行的。

例题

" 矩形加矩形求和"

维护一个二维平面，然后支持在一个矩形区域内加一个数字，每次询问一个矩形区域的和。

解题思路

对于这个问题的静态版本，即「二维平面里有一堆矩形，我们希望询问一个矩形区域的和」，有一个经典做法叫线段树 + 扫描线。具体的做法是先将每个矩形拆成插入和删除两个操作，接着将每个询问拆成两个前缀和相减的形式，最后离线。然而，原题目是动态的，不能直接使用这种做法。

尝试对其使用 CDQ 分治。我们将所有的询问和修改操作全部离线。这些操作形成了一个序列，并且有 $O(N^2)$ 对修改 - 询问的关系。依然使用 CDQ 分治的一般流程，将所有的关系分成三类，在这一层分治过程当中只处理跨越 mid 的修改 - 询问关系，剩下的修改 - 询问关系通过递归的的方式来解决。

我们发现，所有的修改在询问之前就已完成。这时，原问题等价于「平面上有静态的一堆矩形，不停地询问一个矩形区域的和」。

使用一个扫描线在 $O(n\log n)$ 的时间内处理好所有跨越 mid 的修改 - 询问关系，剩下的事情就是递归地分治左右两侧的修改 - 询问关系了。

在这样实现的 CDQ 分治中，同一个询问被处理了 $O(\log n)$ 次。不过没有关系，因为每次贡献这个询问的修改是互不相交的。全套流程的时间复杂度为 $T(n)=T(\lfloor\frac{n}{2}\rfloor)+T(\lceil\frac{n}{2}\rceil)+O(n\log n)=O(n\log^2 n)$。

观察上述的算法流程，我们发现一开始我们只能解决静态的矩形加矩形求和问题，但只是简单地使用 CDQ 分治后，我们就可以离线地解决一个动态的矩形加矩形求和问题了。将动态问题转化为静态问题的精髓就在于 CDQ 分治每次仅仅处理跨越某一个点的修改和询问关系，这样的话我们就只需要考虑「所有询问都在修改之后」这个简单的问题了。也正是因为这一点，CDQ 分治被称为「动态问题转化为静态问题的工具」。

"[Ynoi2016] **镜中的昆虫**[5]"

维护一个长为 n 的序列 a_i，有 m 次操作。

1. 将区间 $[l,r]$ 的值修改为 x；
2. 询问区间 $[l,r]$ 出现了多少种不同的数，也就是说同一个数出现多次只算一个。

解题思路

一句话题意：区间赋值区间数颜色。

维护一下每个位置左侧第一个同色点的位置，记为 pre_i，此时区间数颜色就被转化为了一个经典的二维数点问题。

通过将连续的一段颜色看成一个点的方式，可以证明 pre 的变化量是 $O(n+m)$ 的，即单次操作仅仅引起 $O(1)$ 的 pre 值变化，那么我们可以用 CDQ 分治来解决动态的单点加矩形求和问题。

pre 数组的具体变化可以使用 `std::set` 来进行处理。这个用 set 维护连续的区间的技巧也被称之为 old driver tree。

参考代码

```
#include <algorithm>
#include <cstdio>
#include <map>
#include <set>
#define SNI set<nod>::iterator
#define SDI set<data>::iterator
using namespace std;
const int N = 1e5 + 10;
int n;
int m;
int pre[N];
int npre[N];
```

```
int a[N];
int tp[N];
int lf[N];
int rt[N];
int co[N];

struct modi {
  int t;
  int pos;
  int pre;
  int va;

  friend bool operator<(modi a, modi b) { return a.pre < b.pre; }
} md[10 * N];

int tp1;

struct qry {
  int t;
  int l;
  int r;
  int ans;

  friend bool operator<(qry a, qry b) { return a.l < b.l; }
} qr[N];

int tp2;
int cnt;

bool cmp(const qry& a, const qry& b) { return a.t < b.t; }

void modify(int pos, int co)  // 修改函数
{
  if (npre[pos] == co) return;
  md[++tp1] = (modi){++cnt, pos, npre[pos], -1};
  md[++tp1] = (modi){++cnt, pos, npre[pos] = co, 1};
}

namespace prew {
int lst[2 * N];
map<int, int> mp;  // 提前离散化

void prew() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= n; i++) scanf("%d", &a[i]), mp[a[i]] = 1;
  for (int i = 1; i <= m; i++) {
    scanf("%d%d%d", &tp[i], &lf[i], &rt[i]);
    if (tp[i] == 1) scanf("%d", &co[i]), mp[co[i]] = 1;
  }
  map<int, int>::iterator it, it1;
  for (it = mp.begin(), it1 = it, ++it1; it1 != mp.end(); ++it, ++it1)
    it1->second += it->second;
  for (int i = 1; i <= n; i++) a[i] = mp[a[i]];
  for (int i = 1; i <= n; i++)
```

```
    if (tp[i] == 1) co[i] = mp[co[i]];
  for (int i = 1; i <= n; i++) pre[i] = lst[a[i]], lst[a[i]] = i;
  for (int i = 1; i <= n; i++) npre[i] = pre[i];
}
}  // namespace prew

namespace colist {
struct data {
  int l;
  int r;
  int x;

  friend bool operator<(data a, data b) { return a.r < b.r; }
};

set<data> s;

struct nod {
  int l;
  int r;

  friend bool operator<(nod a, nod b) { return a.r < b.r; }
};

set<nod> c[2 * N];
set<int> bd;

void split(int mid) {  // 将一个节点拆成两个节点
  SDI it = s.lower_bound((data){0, mid, 0});
  data p = *it;
  if (mid == p.r) return;
  s.erase(p);
  s.insert((data){p.l, mid, p.x});
  s.insert((data){mid + 1, p.r, p.x});
  c[p.x].erase((nod){p.l, p.r});
  c[p.x].insert((nod){p.l, mid});
  c[p.x].insert((nod){mid + 1, p.r});
}

void del(set<data>::iterator it) {  // 删除一个迭代器
  bd.insert(it->l);
  SNI it1, it2;
  it1 = it2 = c[it->x].find((nod){it->l, it->r});
  ++it2;
  if (it2 != c[it->x].end()) bd.insert(it2->l);
  c[it->x].erase(it1);
  s.erase(it);
}

void ins(data p) {  // 插入一个节点
  s.insert(p);
  SNI it = c[p.x].insert((nod){p.l, p.r}).first;
  ++it;
  if (it != c[p.x].end()) {
```

```
    bd.insert(it->l);
  }
}

void stv(int l, int r, int x) {  // 区间赋值
  if (l != 1) split(l - 1);
  split(r);
  int p = l;  // split 两下之后删掉所有区间
  while (p != r + 1) {
    SDI it = s.lower_bound((data){0, p, 0});
    p = it->r + 1;
    del(it);
  }
  ins((data){l, r, x});  // 扫一遍 set 处理所有变化的 pre 值
  for (set<int>::iterator it = bd.begin(); it != bd.end(); ++it) {
    SDI it1 = s.lower_bound((data){0, *it, 0});
    if (*it != it1->l)
      modify(*it, *it - 1);
    else {
      SNI it2 = c[it1->x].lower_bound((nod){0, *it});
      if (it2 != c[it1->x].begin())
        --it2, modify(*it, it2->r);
      else
        modify(*it, 0);
    }
  }
  bd.clear();
}

void ih() {
  int nc = a[1];
  int ccnt = 1;  // 将连续的一段插入到 set 中
  for (int i = 2; i <= n; i++)
    if (nc != a[i]) {
      s.insert((data){i - ccnt, i - 1, nc}),
          c[nc].insert((nod){i - ccnt, i - 1});
      nc = a[i];
      ccnt = 1;
    } else {
      ccnt++;
    }
  s.insert((data){n - ccnt + 1, n, a[n]}),
      c[a[n]].insert((nod){n - ccnt + 1, n});
}
}  // namespace colist

namespace CDQ {
struct treearray  // 树状数组
{
  int ta[N];

  void c(int x, int t) {
    for (; x <= n; x += x & (-x)) ta[x] += t;
  }
```

```
  void d(int x) {
    for (; x <= n; x += x & (-x)) ta[x] = 0;
  }

  int q(int x) {
    int r = 0;
    for (; x; x -= x & (-x)) r += ta[x];
    return r;
  }

  void clear() {
    for (int i = 1; i <= n; i++) ta[i] = 0;
  }
} ta;

int srt[N];

bool cmp1(const int& a, const int& b) { return pre[a] < pre[b]; }

void solve(int l1, int r1, int l2, int r2, int L, int R) {  // CDQ
  if (l1 == r1 || l2 == r2) return;
  int mid = (L + R) / 2;
  int mid1 = l1;
  while (mid1 != r1 && md[mid1 + 1].t <= mid) mid1++;
  int mid2 = l2;
  while (mid2 != r2 && qr[mid2 + 1].t <= mid) mid2++;
  solve(l1, mid1, l2, mid2, L, mid);
  solve(mid1, r1, mid2, r2, mid, R);
  if (l1 != mid1 && mid2 != r2) {
    sort(md + l1 + 1, md + mid1 + 1);
    sort(qr + mid2 + 1, qr + r2 + 1);
    for (int i = mid2 + 1, j = l1 + 1; i <= r2; i++) {  // 考虑左侧对右侧贡献
      while (j <= mid1 && md[j].pre < qr[i].l) ta.c(md[j].pos, md[j].va), j++;
      qr[i].ans += ta.q(qr[i].r) - ta.q(qr[i].l - 1);
    }
    for (int i = l1 + 1; i <= mid1; i++) ta.d(md[i].pos);
  }
}

void mainsolve() {
  colist::ih();
  for (int i = 1; i <= m; i++)
    if (tp[i] == 1)
      colist::stv(lf[i], rt[i], co[i]);
    else
      qr[++tp2] = (qry){++cnt, lf[i], rt[i], 0};
  sort(qr + 1, qr + tp2 + 1);
  for (int i = 1; i <= n; i++) srt[i] = i;
  sort(srt + 1, srt + n + 1, cmp1);
  for (int i = 1, j = 1; i <= tp2; i++) {  // 初始化一下每个询问的值
    while (j <= n && pre[srt[j]] < qr[i].l) ta.c(srt[j], 1), j++;
    qr[i].ans += ta.q(qr[i].r) - ta.q(qr[i].l - 1);
  }
```

```
  ta.clear();
  sort(qr + 1, qr + tp2 + 1, cmp);
  solve(0, tp1, 0, tp2, 0, cnt);
  sort(qr + 1, qr + tp2 + 1, cmp);
  for (int i = 1; i <= tp2; i++) printf("%d\n", qr[i].ans);
}
}  // namespace CDQ

int main() {
  prew::prew();
  CDQ::mainsolve();
  return 0;
}
```

"[HNOI2010] 城市建设[6]"

PS 国是一个拥有诸多城市的大国。国王 Louis 为城市的交通建设可谓绞尽脑汁。Louis 可以在某些城市之间修建道路，在不同的城市之间修建道路需要不同的花费。

Louis 希望建造最少的道路使得国内所有的城市连通。但是由于某些因素，城市之间修建道路需要的花费会随着时间而改变。Louis 会不断得到某道路的修建代价改变的消息。他希望每得到一条消息后能立即知道使城市连通的最小花费总和。Louis 决定求助于你来完成这个任务。

解题思路

一句话题意：给定一张图支持动态的修改边权，要求在每次修改边权之后输出这张图的最小生成树的最小代价和。

事实上，有一个线段树分治套 lct 的做法可以解决这个问题，但是这个实现方式的常数过大，可能需要精妙的卡常技巧才可以通过本题，因此不妨考虑 CDQ 分治来解决这个问题。

和一般的 CDQ 分治解决的问题不同，此时使用 CDQ 分治的时候并没有修改和询问的关系来让我们进行分治，因为无法单独考虑「修改一个边对整张图的最小生成树有什么贡献」。传统的 CDQ 分治思路似乎不是很好使。

通过刚才的例题可以发现，一般的 CDQ 分治和线段树有着特殊的联系：我们在 CDQ 分治的过程中其实隐式地建了一棵线段树出来（因为 CDQ 分治的递归树就是一颗线段树）。通常的 CDQ 是考虑线段树左右儿子之间的联系。而对于这道题，我们需要考虑的是父亲和孩子之间的关系；换句话来讲，我们在 `$solve(l,r)$` 这段区间的时候，如果可以想办法使图的规模变成和区间长度相关的一个变量的话，就可以解决这个问题了。

那么具体来讲如何设计算法呢?

假设我们正在构造 (l,r) 这段区间的最小生成树边集，并且我们已知它父亲最小生成树的边集。我们将在 (l,r) 这段区间中发生变化的边分别赋与 $+\infty$ 和 $-\infty$ 的边权，并各跑一边 kruskal，求出在最小生成树里的那些边。

对于一条边来讲：

- 如果最小生成树里所有被修改的边权都被赋成了 $+\infty$，而它未出现在树中，则证明它不可能出现在 (l,r) 这些询问的最小生成树当中。所以我们仅仅在 (l,r) 的边集中加入最小生成树的树边。
- 如果最小生成树里所有被修改的边权都被赋成了 $-\infty$，而它未出现在树中，则证明它一定会出现 (l,r) 这段的区间的最小生成树当中。这样的话我们就可以使用并查集将这些边对应的点缩起来，并且将答案加上这些边的边权。

这样我们就将 (l,r) 这段区间的边集构造出来了。用这些边求出来的最小生成树和直接求原图的最小生成树等价。

那么为什么我们的复杂度是对的呢?

首先，修改过的边一定会加进我们的边集，这些边的数目是 $O(len)$ 级别的。

接下来我们需要证明边集当中不会有过多的未被修改的边。我们只会加入所有边权取 $+\infty$ 最小生成树的树边，因此我们加入的边数目不会超过当前图的点数。

现在我们只需证明每递归一层图的点数是 $O(len)$ 级别的，就可以说明图的边数是 $O(len)$ 级别的了。

证明点数是 $O(len)$ 几倍就变得十分简单了。我们每次向下递归的时侯缩掉的边是在 $-\infty$ 生成树中出现的未被修改边，反过来想就是，我们割掉了出现在 $-\infty$ 生成树当中的所有的被修改边。显然我们最多割掉 len 条边，整张图最多分裂成 $O(len)$ 个连通块，这样的话新图点数就是 $O(len)$ 级别的了。所以我们就证明了每次我们用来跑 kruskal 的图都是 $O(len)$ 级别的了，从而每一层的时间复杂度都是 $O(n\log n)$ 了。

时间复杂度是 $T(n)=T(\lfloor\frac{n}{2}\rfloor)+T(\lceil\frac{n}{2}\rceil)+O(n\log n)=O(n\log^2 n)$。

代码实现上可能会有一些难度。需要注意的是并查集不能使用路径压缩，否则就不支持回退操作了。执行缩点操作的时候也没有必要真的执行，而是每一层的 kruskal 都在上一层的并查集里直接做就可以了。

示例代码

```cpp
#include <algorithm>
#include <cstdio>
#include <stack>
#include <vector>
using namespace std;
typedef long long ll;
int n;
int m;
int ask;

struct bcj {
  int fa[20010];
  int size[20010];

  struct opt {
    int u;
    int v;
  };

  stack<opt> st;

  void ih() {
    for (int i = 1; i <= n; i++) fa[i] = i, size[i] = 1;
  }

  int f(int x) { return (fa[x] == x) ? x : f(fa[x]); }

  void u(int x, int y) {  // 带撤回
    int u = f(x);
    int v = f(y);
    if (u == v) return;
    if (size[u] < size[v]) swap(u, v);
    size[u] += size[v];
    fa[v] = u;
    opt o;
    o.u = u;
    o.v = v;
    st.push(o);
```

```
  }

  void undo() {
    opt o = st.top();
    st.pop();
    fa[o.v] = o.v;
    size[o.u] -= size[o.v];
  }

  void clear(int tim) {
    while (st.size() > tim) {
      undo();
    }
  }
} s, s1;

struct edge  // 静态边
{
  int u;
  int v;
  ll val;
  int mrk;

  friend bool operator<(edge a, edge b) { return a.val < b.val; }
} e[50010];

struct moved {
  int u;
  int v;
};  // 动态边

struct query {
  int num;
  ll val;
  ll ans;
} q[50010];

bool book[50010];  // 询问
vector<edge> ve[30];
vector<moved> vq;
vector<edge> tr;
ll res[30];
int tim[30];

void pushdown(int dep)  // 缩边
{
  tr.clear();  // 这里要复制一份，以免无法回撤操作
  for (int i = 0; i < ve[dep].size(); i++) {
    tr.push_back(ve[dep][i]);
  }
  sort(tr.begin(), tr.end());
  for (int i = 0; i < tr.size(); i++) {  // 无用边
    if (s1.f(tr[i].u) == s1.f(tr[i].v)) {
      tr[i].mrk = -1;
```

```
      continue;
    }
    s1.u(tr[i].u, tr[i].v);
  }
  s1.clear(0);
  res[dep + 1] = res[dep];
  for (int i = 0; i < vq.size(); i++) {
    s1.u(vq[i].u, vq[i].v);
  }
  vq.clear();
  for (int i = 0; i < tr.size(); i++) {  // 必须边
    if (tr[i].mrk == -1 || s1.f(tr[i].u) == s1.f(tr[i].v)) continue;
    tr[i].mrk = 1;
    s1.u(tr[i].u, tr[i].v);
    s.u(tr[i].u, tr[i].v);
    res[dep + 1] += tr[i].val;
  }
  s1.clear(0);
  ve[dep + 1].clear();
  for (int i = 0; i < tr.size(); i++) {  // 缩边
    if (tr[i].mrk != 0) continue;
    edge p;
    p.u = s.f(tr[i].u);
    p.v = s.f(tr[i].v);
    if (p.u == p.v) continue;
    p.val = tr[i].val;
    p.mrk = 0;
    ve[dep + 1].push_back(p);
  }
  return;
}

void solve(int l, int r, int dep) {
  tim[dep] = s.st.size();
  int mid = (l + r) / 2;
  if (r - l == 1) {  // 终止条件
    edge p;
    p.u = s.f(e[q[r].num].u);
    p.v = s.f(e[q[r].num].v);
    p.val = q[r].val;
    e[q[r].num].val = q[r].val;
    p.mrk = 0;
    ve[dep].push_back(p);
    pushdown(dep);
    q[r].ans = res[dep + 1];
    s.clear(tim[dep - 1]);
    return;
  }
  for (int i = l + 1; i <= mid; i++) {
    book[q[i].num] = true;
  }
  for (int i = mid + 1; i <= r; i++) {  // 动转静
    if (book[q[i].num]) continue;
    edge p;
```

```
    p.u = s.f(e[q[i].num].u);
    p.v = s.f(e[q[i].num].v);
    p.val = e[q[i].num].val;
    p.mrk = 0;
    ve[dep].push_back(p);
  }
  for (int i = l + 1; i <= mid; i++) {  // 询问转动态
    moved p;
    p.u = s.f(e[q[i].num].u);
    p.v = s.f(e[q[i].num].v);
    vq.push_back(p);
  }
  pushdown(dep);  // 下面的是回撤
  for (int i = mid + 1; i <= r; i++) {
    if (book[q[i].num]) continue;
    ve[dep].pop_back();
  }
  for (int i = l + 1; i <= mid; i++) {
    book[q[i].num] = false;
  }
  solve(l, mid, dep + 1);
  for (int i = 0; i < ve[dep].size(); i++) {
    ve[dep][i].mrk = 0;
  }
  for (int i = mid + 1; i <= r; i++) {
    book[q[i].num] = true;
  }
  for (int i = l + 1; i <= mid; i++) {  // 动转静
    if (book[q[i].num]) continue;
    edge p;
    p.u = s.f(e[q[i].num].u);
    p.v = s.f(e[q[i].num].v);
    p.val = e[q[i].num].val;
    p.mrk = 0;
    ve[dep].push_back(p);
  }
  for (int i = mid + 1; i <= r; i++) {  // 询问转动
    book[q[i].num] = false;
    moved p;
    p.u = s.f(e[q[i].num].u);
    p.v = s.f(e[q[i].num].v);
    vq.push_back(p);
  }
  pushdown(dep);
  solve(mid, r, dep + 1);
  s.clear(tim[dep - 1]);
  return;  // 时间倒流至上一层
}

int main() {
  scanf("%d%d%d", &n, &m, &ask);
  s.ih();
  s1.ih();
  for (int i = 1; i <= m; i++) {
```

```
    scanf("%d%d%lld", &e[i].u, &e[i].v, &e[i].val);
  }
  for (int i = 1; i <= ask; i++) {
    scanf("%d%lld", &q[i].num, &q[i].val);
  }
  for (int i = 1; i <= ask; i++) {  // 初始动态边
    book[q[i].num] = true;
    moved p;
    p.u = e[q[i].num].u;
    p.v = e[q[i].num].v;
    vq.push_back(p);
  }
  for (int i = 1; i <= m; i++) {
    if (book[i]) continue;
    ve[1].push_back(e[i]);
  }  // 初始静态
  for (int i = 1; i <= ask; i++) {
    book[q[i].num] = false;
  }
  solve(0, ask, 1);
  for (int i = 1; i <= ask; i++) {
    printf("%lld\n", q[i].ans);
  }
  return 0;
}
```

参考资料与注释

[1] 从 《Cash》 谈一类分治算法的应用

[2] 三维偏序

[3] CQOI2011 动态逆序对

[4] SDOI2011 拦截导弹

[5] [Ynoi2016] 镜中的昆虫

[6] [HNOI2010] 城市建设

13.4.3 整体二分

引入

在信息学竞赛中，有一部分题目可以使用二分的办法来解决。但是当这种题目有多次询问且我们每次查询都直接二分可能导致 TLE 时，就会用到整体二分。整体二分的主体思路就是把多个查询一起解决。(所以这是一个离线算法)

> 可以使用整体二分解决的题目需要满足以下性质：
>
> 1. 询问的答案具有可二分性

> 2. **修改对判定答案的贡献互相独立**，修改之间互不影响效果
> 3. 修改如果对判定答案有贡献，则贡献为一确定的与判定标准无关的值
> 4. 贡献满足交换律，结合律，具有可加性
> 5. 题目允许使用离线算法
>
> ——许昊然《浅谈数据结构题几个非经典解法》

解释

记 $[l,r]$ 为答案的值域，$[L,R]$ 为答案的定义域。（也就是说求答案时仅考虑下标在区间 $[L,R]$ 内的操作和询问，这其中询问的答案在 $[l,r]$ 内）

- 我们首先把所有操作**按时间顺序**存入数组中，然后开始分治。
- 在每一层分治中，利用数据结构（常见的是树状数组）统计当前查询的答案和 mid 之间的关系。
- 根据查询出来的答案和 mid 间的关系（小于等于 mid 和大于 mid）将当前处理的操作序列分为 $q1$ 和 $q2$ 两份，并分别递归处理。
- 当 $l=r$ 时，找到答案，记录答案并返回即可。

需要注意的是，在整体二分过程中，若当前处理的值域为 $[l,r]$，则此时最终答案范围不在 $[l,r]$ 的询问会在其他时候处理。

过程

注：

1. 为可读性，文中代码或未采用实际竞赛中的常见写法。
2. 若觉得某段代码有难以理解之处，请先参考之前题目的解释，因为节省篇幅解释过的内容不再赘述。

从普通二分说起：

查询全局第 k 小

> **题 1** 在一个数列中查询第 k 小的数。

当然可以直接排序。如果用二分法呢？可以用数据结构记录每个大小范围内有多少个数，然后用二分法猜测，利用数据结构检验。

> **题 2** 在一个数列中多次查询第 k 小的数。

可以对于每个询问进行一次二分；但是，也可以把所有的询问放在一起二分。

先考虑二分的本质：假设要猜一个 $[l,r]$ 之间的数，猜测之后会知道是猜大了，猜小了还是刚好。当然可以从 l 枚举到 r，但更优秀的方法是二分：猜测答案是 $m=\lfloor\frac{l+r}{2}\rfloor$，然后去验证 m 的正确性，再调整边界。这样做每次询问的复杂度为 $O(\log n)$，若询问次数为 q，则时间复杂度为 $O(q\log n)$。

回过头来，对于当前的所有询问，可以去猜测所有询问的答案都是 mid，然后去依次验证每个询问的答案应该是小于等于 mid 的还是大于 mid 的，并将询问分为两个部分（不大于/大于），对于每个部分继续二分。注意：如果一个询问的答案是大于 mid 的，则在将其划至右侧前需更新它的 k，即，如果当前数列中小于等于 mid 的数有 t 个，则将询问划分后实际是在右区间询问第 $k-t$ 小数。如果一个部分的 $l=r$ 了，则结束这个部分的二分。利用线段树的相关知识，我们每次将整个答案可能在的区间 $[1,maxans]$ 划分成了若干个部分，这样的划分共进行了 $O(\log maxans)$ 次，一次划分会将整个操作序列操作一次。若对整个序列进行操作，并支持对应的查询的时间复杂度为 $O(T)$，则整体二分的时间复杂度为 $O(T\log n)$。

试试完成以下代码：

```
struct Query {
  int id, k;  // 这个询问的编号，这个询问的 k
};

int ans[N];       // ans[i] 表示编号为 i 的询问的答案
int check(int x);  // 返回原数列中小于等于 x 的数的个数

void solve(int l, int r, vector<Query> q)
// 请补全这个函数
{
  int m = (l + r) / 2;
  vector<Query> q1, q2;  // 将被划到左侧的询问和右侧的询问
  if (l == r) {
    // ...
    return;
  }
  // ...
  solve(l, m, q1), solve(m + 1, r, q2);
  return;
}
```

参考代码如下

" 实现"

```
void solve(int l, int r, vector<Query> q) {
  int m = (l + r) / 2;
  if (l == r) {
    for (unsigned i = 0; i < q.size(); i++) ans[q[i].id] = l;
    return;
  }
  vector<int> q1, q2;
  for (unsigned i = 0; i < q.size(); i++)
    if (q[i].k <= check(m))
      q1.push_back(q[i]);
    else
      q[i].k -= check(m), q2.push_back(q[i]);
  solve(l, m, q1), solve(m + 1, r, q2);
  return;
}
```

查询区间第 k 小

> **题 3** 在一个数列中多次查询区间第 k 小的数。

涉及到给定区间的查询，再按之前的方法进行二分就会导致 check 函数的时间复杂度爆炸。仍然考虑询问与值域中点 m 的关系：若询问区间内小于等于 m 的数有 t 个，询问的是区间内的 k 小数，则当 $k \leq t$ 时，答案应小于等于 m；否则，答案应大于 m。（注意边界问题）此处需记录一个区间小于等于指定数的数的数量，即单点加，求区间和，可用树状数组快速处理。为提高效率，只对数列中值在值域区间 $[l, r]$ 的数进行统计，即，在进一步递归之前，不仅将询问划分，将当前处理的数按值域范围划为两半。

参考代码（关键部分）

" 实现"

```
struct Num {
  int p, x;
};  // 位于数列中第 p 项的数的值为 x

struct Query {
  int l, r, k, id;
};  // 一个编号为 id, 询问 [l,r] 中第 k 小数的询问

int ans[N];
void add(int p, int x);  // 树状数组, 在 p 位置加上 x
int query(int p);        // 树状数组, 求 [1,p] 的和
void clear();            // 树状数组, 清空

void solve(int l, int r, vector<Num> a, vector<Query> q)
// a 中为给定数列中值在值域区间 [l,r] 中的数
{
  int m = (l + r) / 2;
  if (l == r) {
    for (unsigned i = 0; i < q.size(); i++) ans[q[i].id] = l;
    return;
  }
  vector<Num> a1, a2;
  vector<Query> q1, q2;
  for (unsigned i = 0; i < a.size(); i++)
    if (a[i].x <= m)
      a1.push_back(a[i]), add(a[i].p, 1);
    else
      a2.push_back(a[i]);
  for (unsigned i = 0; i < q.size(); i++) {
    int t = query(q[i].r) - query(q[i].l - 1);
    if (q[i].k <= t)
      q1.push_back(q[i]);
    else
      q[i].k -= t, q2.push_back(q[i]);
  }
  clear();
  solve(l, m, a1, q1), solve(m + 1, r, a2, q2);
  return;
}
```

下面提供 【 模板 】 可持久化线段树 2[1] 一题使用整体二分的，偏向竞赛风格的写法。

" 参考代码"

```
#include <bits/stdc++.h>
using namespace std;
const int N = 200020;
const int INF = 1e9;
int n, m;
int ans[N];
// BIT begin
int t[N];
```

```
int a[N];

int sum(int p) {
  int ans = 0;
  while (p) {
    ans += t[p];
    p -= p & (-p);
  }
  return ans;
}

void add(int p, int x) {
  while (p <= n) {
    t[p] += x;
    p += p & (-p);
  }
}

// BIT end
int tot = 0;

struct Query {
  int l, r, k, id, type;  // set values to -1 when they are not used!
} q[N * 2], q1[N * 2], q2[N * 2];

void solve(int l, int r, int ql, int qr) {
  if (ql > qr) return;
  if (l == r) {
    for (int i = ql; i <= qr; i++)
      if (q[i].type == 2) ans[q[i].id] = l;
    return;
  }
  int mid = (l + r) / 2, cnt1 = 0, cnt2 = 0;
  for (int i = ql; i <= qr; i++) {
    if (q[i].type == 1) {
      if (q[i].l <= mid) {
        add(q[i].id, 1);
        q1[++cnt1] = q[i];
      } else
        q2[++cnt2] = q[i];
    } else {
      int x = sum(q[i].r) - sum(q[i].l - 1);
      if (q[i].k <= x)
        q1[++cnt1] = q[i];
      else {
        q[i].k -= x;
        q2[++cnt2] = q[i];
      }
    }
  }
  // rollback changes
  for (int i = 1; i <= cnt1; i++)
    if (q1[i].type == 1) add(q1[i].id, -1);
  // move them to the main array
```

```
  for (int i = 1; i <= cnt1; i++) q[i + ql - 1] = q1[i];
  for (int i = 1; i <= cnt2; i++) q[i + cnt1 + ql - 1] = q2[i];
  solve(l, mid, ql, cnt1 + ql - 1);
  solve(mid + 1, r, cnt1 + ql, qr);
}

pair<int, int> b[N];
int toRaw[N];

int main() {
  scanf("%d%d", &n, &m);
  // read and discrete input data
  for (int i = 1; i <= n; i++) {
    int x;
    scanf("%d", &x);
    b[i].first = x;
    b[i].second = i;
  }
  sort(b + 1, b + n + 1);
  int cnt = 0;
  for (int i = 1; i <= n; i++) {
    if (b[i].first != b[i - 1].first) cnt++;
    a[b[i].second] = cnt;
    toRaw[cnt] = b[i].first;
  }
  for (int i = 1; i <= n; i++) {
    q[++tot] = {a[i], -1, -1, i, 1};
  }
  for (int i = 1; i <= m; i++) {
    int l, r, k;
    scanf("%d%d%d", &l, &r, &k);
    q[++tot] = {l, r, k, i, 2};
  }
  solve(0, cnt + 1, 1, tot);
  for (int i = 1; i <= m; i++) printf("%d\n", toRaw[ans[i]]);
}
```

带修区间第 k 小

> **题 4** Dynamic Rankings[2] 给定一个数列，要支持单点修改，区间查第 k 小。

修改操作可以直接理解为从原数列中删去一个数再添加一个数，为方便起见，将询问和修改统称为「操作」。因后面的操作会依附于之前的操作，不能如题 3 一样将统计和处理询问分开，故可将所有操作存于一个数组，用标识区分类型，依次处理每个操作。为便于处理树状数组，修改操作可分拆为擦除操作和插入操作。

优化

1. 注意到每次对于操作进行分类时，只会更改操作顺序，故可直接在原数组上操作。具体实现，在二分时将记录操作的 q, a 数组换为一个大的全局数组，二分时记录信息变为 L, R，即当前处理的操作是全局数组上的哪个区间。利用临时数组记录当前的分类情况，进一步递归前将临时数组信息写回原数组。
2. 树状数组每次清空会导致时间复杂度爆炸，可采用每次使用树状数组时记录当前修改位置（这已由 1 中提到的临时数组实现），本次操作结束后在原位置加 -1 的方法快速清零。
3. 一开始对于数列的初始化操作可简化为插入操作。

参考代码（关键部分）

" 实现"

```
struct Opt {
  int x, y, k, type, id;
  // 对于询问, type = 1, x, y 表示区间左右边界, k 表示询问第 k 小
  // 对于修改, type = 0, x 表示修改位置, y 表示修改后的值,
  // k 表示当前操作是插入 (1) 还是擦除 (-1), 更新树状数组时使用.
  // id 记录每个操作原先的编号, 因二分过程中操作顺序会被打散
};

Opt q[N], q1[N], q2[N];
// q 为所有操作,
// 二分过程中, 分到左边的操作存到 q1 中, 分到右边的操作存到 q2 中.
int ans[N];
void add(int p, int x);
int query(int p);  // 树状数组函数, 含义见题 3

void solve(int l, int r, int L, int R)
// 当前的值域范围为 [l,r], 处理的操作的区间为 [L,R]
{
  if (l > r || L > R) return;
  int cnt1 = 0, cnt2 = 0, m = (l + r) / 2;
  // cnt1, cnt2 分别为分到左边, 分到右边的操作数
  if (l == r) {
    for (int i = L; i <= R; i++)
      if (q[i].type == 1) ans[q[i].id] = l;
    return;
  }
  for (int i = L; i <= R; i++)
    if (q[i].type == 1) {  // 是询问: 进行分类
      int t = query(q[i].y) - query(q[i].x - 1);
      if (q[i].k <= t)
        q1[++cnt1] = q[i];
      else
        q[i].k -= t, q2[++cnt2] = q[i];
    } else
      // 是修改: 更新树状数组 & 分类
      if (q[i].y <= m)
        add(q[i].x, q[i].k), q1[++cnt1] = q[i];
      else
        q2[++cnt2] = q[i];
  for (int i = 1; i <= cnt1; i++)
    if (q1[i].type == 0) add(q1[i].x, -q1[i].k);  // 清空树状数组
  for (int i = 1; i <= cnt1; i++) q[L + i - 1] = q1[i];
  for (int i = 1; i <= cnt2; i++)
    q[L + cnt1 + i - 1] = q2[i];  // 将临时数组中的元素合并回原数组
  solve(l, m, L, L + cnt1 - 1), solve(m + 1, r, L + cnt1, R);
  return;
}
```

针对静态序列的优化

> **题 5** 【模板】 可持久化线段树 2[1] 给定一个序列，区间查询第 k 小。

树套树和整体二分实现带修区间第 k 小问题的复杂度都为 $O(n\log^2 n)$，但静态区间第 k 小问题可以使用可持久化线段树在 $O(n\log n)$ 时间复杂度内解决，而几乎所有整体二分实现的静态区间第 k 小问题代码时间复杂度都是 $O(n\log^2 n)$，面对大数据范围时存在 TLE 的风险。（这里默认值域与序列长度同阶，值域与序列长不同阶的情况可以通过离散化转化为同阶情况）

优化

1. 对于每一轮划分，如果当前数列中小于等于 mid 的数有 t 个，则将询问划分后实际是在右区间询问第 $k-t$ 小数，因此对划分到右区间的询问做出了修改。如果答案的原始值域为 $[L,R]$，某次划分的答案值域为 $[l,r]$，那么对于参与此次划分的询问，$[L,l)$ 中所有数值对它们的影响已经在之前被消除了。
2. 由于需要使每轮划分都仅和当前答案值域 $[l,r]$ 有关，树状数组需要多次载入和清空。

如果划分不仅仅和当前答案值域有关呢?

由此可以得到一个与全局序列有关的优化方法：维护一个指针 pos 追踪每轮划分的 mid（分治中心），将所有 $\le pos$ 的元素对应的下标在树状数组中置为 1，树状数组的其余位置置为 0。每次划分之前移动 pos 并更新树状数组。指针 pos 移动的次数与 $n\log n$ 同阶。划分时对每一个询问查询树状数组中对应区间的值，满足则划分至左区间，否则划分至右区间，**不需要对询问做出修改**。

由于要追踪分治中心，需要让 pos 准确地更新树状数组。在整体二分之前将序列按元素大小排序并记录元素对应下标，指针移动时在树状数组中对下标进行相应修改。对于绝大多数**可以用整体二分解决并且不带修改的问题**，都可以应用此种优化以大幅降低数据结构的使用次数。

由于减少了很多树状数组的载入和清空操作，应用这种优化通常情况下会明显提升整体二分的效率（即使只是常数优化），对于静态区间第 k 小值问题而言效率完全不差于时间复杂度更优的可持久化线段树。值得注意的是，对于静态区间第 k 小值问题也存在时间复杂度 $O(n\log n)$ 的整体二分实现。

参考代码（关键部分）

" 实现"

```
struct Query {
  int i, l, r, k;
};  // 第 i 次询问查询区间 [l,r] 的第 k 小值

Query s[200005], t1[200005], t2[200005];
int n, m, cnt, pos, p[200005], ans[200005];
pair<int, int> a[200005];

void add(int x, int y);  // 树状数组位置 x 加 y
int sum(int x);          // 树状数组 [1,x] 前缀和

// 当前处理的询问为 [l,r]，答案值域为 [ql,qr]
void overall_binary(int l, int r, int ql, int qr) {
  if (l > r) return;
  if (ql == qr) {
    for (int i = l; i <= r; i++) ans[s[i].i] = ql;
    return;
  }
  int cnt1 = 0, cnt2 = 0, mid = (ql + qr) >> 1;
  // 追踪分治中心，认为 [1,pos] 的值已经载入树状数组
  while (pos <= n - 1 && a[pos + 1].first <= mid)
    add(a[pos + 1].second, 1), ++pos;
  while (pos >= 1 && a[pos].first > mid) add(a[pos].second, -1), --pos;

  for (int i = l; i <= r; i++) {
    int now = sum(s[i].r) - sum(s[i].l - 1);
```

```
    if (s[i].k <= now)
      t1[++cnt1] = s[i];
    else
      t2[++cnt2] = s[i];  // 注意不应修改询问信息
  }
  for (int i = 1; i <= cnt1; i++) s[l + i - 1] = t1[i];
  for (int i = 1; i <= cnt2; i++) s[l + cnt1 + i - 1] = t2[i];

  overall_binary(l, l + cnt1 - 1, ql, mid);
  overall_binary(l + cnt1, r, mid + 1, qr);
}

int main() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= n; i++) {
    scanf("%d", &a[i].first);
    a[i].second = i;
    p[++cnt] = a[i].first;
  }
  sort(a + 1, a + n + 1);  // 对序列排序离散化
  sort(p + 1, p + n + 1);
  cnt = unique(p + 1, p + n + 1) - p - 1;
  for (int i = 1; i <= n; i++)
    a[i].first = lower_bound(p + 1, p + cnt + 1, a[i].first) - p;
  // 省略读入询问
  overall_binary(1, m, 1, cnt);
  for (int i = 1; i <= n; i++) printf("%d\n", p[ans[i]]);
  return 0;
}
```

区间前驱后继

> **题 6** 在一个数列中多次查询 k 在区间中的前驱（严格小于 k，且最大的数）或后继（严格大于 k，且最小的数），保证存在这样的数。

以前驱为例，使用数据结构解决此种问题的方法一般是先查询区间内有多少严格小于 k 的数（设它们的数量为 x），再查询区间第 x 小的数。后继则是查询区间内有多少不大于 k 的数（数量为 x），然后查询区间第 $x+1$ 小的数。

考虑使用整体二分解决这个问题：整体二分是一种高效求解区间第 k 小的离线算法，而 CDQ 分治[3] 可以离线高效求解区间内的排名。先跑一遍 CDQ 分治求出排名就可以使用整体二分得到区间内部的前驱和后继了。

此问题还可以用 CDQ 分治套线段树离线一遍解决，但效率远低于跑两遍的 CDQ 分治 + 整体二分。

构造单调性序列

> **题 7** Sequence[4] 给定一个序列，每次操作可以把某个数 $+1$ 或 -1。要求把序列变成单调不降的，并且修改后的数列只能出现修改前的数，输出最小操作次数。

此类题目也可以使用动态规划或反悔贪心解决。

在满足操作次数最小化的前提下，一定存在一种方案使得最后序列中的每个数都是序列修改前存在的，这个结论可以使用数学归纳法证明。由于题目并不需要最终序列的信息，问题转化为求出最小操作次数。

由于要求最终的序列单调不降，可以使用整体二分。每轮整体二分判定最终序列区间 $[l,r]$ 的值域，此时答案的值域为 $[ql,qr]$。令 $mid=\lfloor\frac{ql+qr}{2}\rfloor$，每轮二分开始时默认将所有数划分至 $[mid+1,qr]$（要划分到 $[ql,mid]$ 的数设为 0 个），初始代价设为将序列区间 $[l,r]$ 全部置为 $mid+1$ 的操作次数。依次枚举区间 $[l,r]$ 中的数 i 并且计算将 $[l,i]$ 置为

mid、将 $[i+1, r]$ 置为 $mid+1$ 的操作次数之和，如果优于之前的操作次数则更新最少操作次数和要划分到 $[ql, mid]$ 的数的个数。

划分时已经保证了最终序列的单调性不被破坏，同时因为每次都取最小操作次数，最终被划分至左区间的数取 mid 一定比取 $mid+1$ 更优，故整体二分得到的序列一定是单调不降且操作次数最小的。计算操作次数输出即可。

参考代码（关键部分）

" 实现"

```cpp
int a[500005], ans[500005];  // a: 原序列 ans: 构造的序列

void overall_binary(int l, int r, int ql, int qr) {
  if (l > r) return;
  if (ql == qr) {
    for (int i = l; i <= r; i++) ans[i] = ql;
    return;
  }
  int cnt = 0,
      mid = ql + ((qr - ql) >> 1);  // 默认开始都填 mid+1 全部划分到右区间
  long long res = 0ll, sum = 0ll;
  for (int i = l; i <= r; i++) sum += abs(a[i] - (mid + 1));
  res = sum;
  for (int i = l; i <= r;
       i++) {  // 尝试把 [l,i] 从 mid+1 换成 mid 并且划分到左区间
    sum -= abs(a[i] - (mid + 1));
    sum += abs(a[i] - mid);
    if (sum < res) cnt = i - l + 1, res = sum;  // 发现 [l,i] 取 mid 更优，更新
  }
  overall_binary(l, l + cnt - 1, ql, mid);
  overall_binary(l + cnt, r, mid + 1, qr);
}
```

参考习题

「国家集训队」矩阵乘法[5]

「POI2011 R3 Day2」流星 Meteors[6]

二逼平衡树[7]

[BalticOI 2004] Sequence 数字序列[8]

参考资料

- 许昊然《浅谈数据结构题几个非经典解法》

参考资料与注释

[1] 【模板】可持久化线段树 2 [1-1] [1-2]

[2] Dynamic Rankings

[3] CDQ 分治

[4] Sequence

[5] 「国家集训队」矩阵乘法

[6] 「POI2011 R3 Day2」流星 Meteors

[7] 二逼平衡树

[8] [BalticOI 2004] Sequence 数字序列

13.4.4 莫队算法

莫队算法简介

Authors: StudyingFather, Backl1ght, countercurrent-time, Ir1d, greyqz, MicDZ, ouuan

莫队算法是由莫涛提出的算法。在莫涛提出莫队算法之前，莫队算法已经在 Codeforces 的高手圈里小范围流传，但是莫涛是第一个对莫队算法进行详细归纳总结的人。莫涛提出莫队算法时，只分析了普通莫队算法，但是经过 OIer 和 ACMer 的集体智慧改造，莫队有了多种扩展版本。

莫队算法可以解决一类离线区间询问问题，适用性极为广泛。同时将其加以扩展，便能轻松处理树上路径询问以及支持修改操作。

普通莫队算法

Authors: StudyingFather, Backl1ght, countercurrent-time, Ir1d, greyqz, MicDZ, ouuan

形式

假设 $n=m$，那么对于序列上的区间询问问题，如果从 $[l,r]$ 的答案能够 $O(1)$ 扩展到 $[l-1,r],[l+1,r],[l,r+1],[l,r-1]$（即与 $[l,r]$ 相邻的区间）的答案，那么可以在 $O(n\sqrt{n})$ 的复杂度内求出所有询问的答案。

解释

离线后排序，顺序处理每个询问，暴力从上一个区间的答案转移到下一个区间答案（一步一步移动即可）。

排序方法

对于区间 $[l,r]$, 以 l 所在块的编号为第一关键字，r 为第二关键字从小到大排序。

实现

```
void move(int pos, int sign) {
  // update nowAns
}

void solve() {
  BLOCK_SIZE = int(ceil(pow(n, 0.5)));
  sort(querys, querys + m);
  for (int i = 0; i < m; ++i) {
    const query &q = querys[i];
```

```
    while (l > q.l) move(--l, 1);
    while (r < q.r) move(++r, 1);
    while (l < q.l) move(l++, -1);
    while (r > q.r) move(r--, -1);
    ans[q.id] = nowAns;
  }
}
```

复杂度分析

以下的情况在 n 和 m 同阶的前提下讨论。

首先是分块这一步，这一步的时间复杂度是 $O(\sqrt{n}\cdot\sqrt{n}\log\sqrt{n}+n\log n)=O(n\log n)$;

接着就到了莫队算法的精髓了，下面我们用通俗易懂的初中方法来证明它的时间复杂度是 $O(n\sqrt{n})$;

" 证明"

证：令每一块中 L 的最大值为 $\max_1,\max_2,\max_3,\cdots,\max_{\lceil\sqrt{n}\rceil}$。

由第一次排序可知，$\max_1\le\max_2\le\cdots\le\max_{\lceil\sqrt{n}\rceil}$。

显然，对于每一块暴力求出第一个询问的时间复杂度为 $O(n)$。

考虑最坏的情况，在每一块中，R 的最大值均为 n，每次修改操作均要将 L 由 $\max_{i-1}$ 修改至 $\max_i$ 或由 $\max_i$ 修改至 $\max_{i-1}$。

考虑 R：因为 R 在块中已经排好序，所以在同一块修改完它的时间复杂度为 $O(n)$。对于所有块就是 $O(n\sqrt{n})$。

重点分析 L：因为每一次改变的时间复杂度都是 $O(\max_i-\max_{i-1})$ 的，所以在同一块中时间复杂度为 $O(\sqrt{n}\cdot(\max_i-\max_{i-1}))$。

将每一块 L 的时间复杂度合在一起，可以得到：

对于 L 的总时间复杂度为

$$\begin{aligned}&O(\sqrt{n}(\max{}_1-1)+\sqrt{n}(\max{}_2-\max{}_1)+\sqrt{n}(\max{}_3-\max{}_2)+\cdots+\sqrt{n}(\max{}_{\lceil\sqrt{n}\rceil}-\max{}_{\lceil\sqrt{n}\rceil-1}))\\
=&O(\sqrt{n}\cdot(\max{}_1-1+\max{}_2-\max{}_1+\max{}_3-\max{}_2+\cdots+\max{}_{\lceil\sqrt{n}\rceil-1}-\max{}_{\lceil\sqrt{n}\rceil-2}+\max{}_{\lceil\sqrt{n}\rceil}-\max{}_{\lceil\sqrt{n}\rceil-1}))\\
=&O(\sqrt{n}\cdot(\max{}_{\lceil\sqrt{n}\rceil-1}))\end{aligned}$$

（裂项求和）

由题可知 $\max_{\lceil\sqrt{n}\rceil}$ 最大为 n，所以 L 的总时间复杂度最坏情况下为 $O(n\sqrt{n})$。

综上所述，莫队算法的时间复杂度为 $O(n\sqrt{n})$;

但是对于 m 的其他取值，如 $m<n$，分块方式需要改变才能变的更优。

怎么分块呢?

我们设块长度为 S，那么对于任意多个在同一块内的询问，挪动的距离就是 n，一共 $\frac{n}{S}$ 个块，移动的总次数就是 $\frac{n^2}{S}$，移动可能跨越块，所以还要加上一个 mS 的复杂度，总复杂度为 $O\left(\frac{n^2}{S}+mS\right)$，我们要让这个值尽量小，那么就要将这两个项尽量相等，发现 S 取 $\frac{n}{\sqrt{m}}$ 是最优的，此时复杂度为 $O\left(\frac{n^2}{\frac{n}{\sqrt{m}}}+m\left(\frac{n}{\sqrt{m}}\right)\right)=O(n\sqrt{m})$。

事实上，如果块长度的设定不准确，则莫队的时间复杂度会受到很大影响。例如，如果 m 与 $\sqrt{n}$ 同阶，并且块长误设为 $\sqrt{n}$，则可以很容易构造出一组数据使其时间复杂度为 $O(n\sqrt{n})$ 而不是正确的 $O(n)$。

莫队算法看起来十分暴力，很大程度上是因为莫队算法的分块排序方法看起来很粗糙。我们会想到通过看上去更精细的排序方法对所有区间排序。一种方法是把所有区间 $[l,r]$ 看成平面上的点 (l,r)，并对所有点建立曼哈顿最小生成树，每次沿着曼哈顿最小生成树的边在询问之间转移答案。这样看起来可以改善莫队算法的时间复杂度，但是实际上对询问分块排序的方法的时间复杂度上界已经是最优的了。

假设 n,m 同阶且 n 是完全平方数。我们考虑形如 $[a\sqrt{n},b\sqrt{n}](1\le a,b\le\sqrt{n})$ 的区间，这样的区间一共有 n 个。

如果把所有的区间看成平面上的点，则两点之间的曼哈顿距离恰好为两区间的转移代价，并且任意两个区间之间的最小曼哈顿距离为 $\sqrt{n}$，所以处理所有询问的时间复杂度最小为 $O(n\sqrt{n})$。其它情况的数据构造方法与之类似。

莫队算法还有一个特点：当 n 不变时，m 越大，处理每次询问的平均转移代价就越小。一些其他的离线算法也具有同样的特点（如求 LCA 的 Tarjan 算法），但是莫队算法的平均转移代价随 m 的变化最明显。

例题 & 代码

" 例题 「 国家集训队 」 小 Z 的袜子[1]"

题目大意：

有一个长度为 n 的序列 $\{c_i\}$。现在给出 m 个询问，每次给出两个数 l,r，从编号在 l 到 r 之间的数中随机选出两个不同的数，求两个数相等的概率。

过程

思路：莫队算法模板题。

对于区间 $[l,r]$，以 l 所在块的编号为第一关键字，r 为第二关键字从小到大排序。

然后从序列的第一个询问开始计算答案，第一个询问通过直接暴力算出，复杂度为 $O(n)$，后面的询问在前一个询问的基础上得到答案。

具体做法：

对于区间 $[i,i]$，由于区间只有一个元素，我们很容易就能知道答案。然后一步一步从当前区间（已知答案）向下一个区间靠近。

我们设 $col[i]$ 表示当前颜色 i 出现了多少次，ans 当前共有多少种可行的配对方案（有多少种可以选到一双颜色相同的袜子），表示然后每次移动的时候更新答案——设当前颜色为 k，如果是增长区间就是 ans 加上 $\binom{col[k]+1}{2}-\binom{col[k]}{2}$，如果是缩短就是 ans 减去 $\binom{col[k]}{2}-\binom{col[k]-1}{2}$。

而这个询问的答案就是 $\dfrac{ans}{\binom{r-l+1}{2}}$。

这里有个优化：$\binom{a}{2}=\dfrac{a(a-1)}{2}$。

所以 $\binom{a+1}{2}-\binom{a}{2}=\dfrac{(a+1)a}{2}-\dfrac{a(a-1)}{2}=\dfrac{a}{2}\cdot(a+1-a+1)=\dfrac{a}{2}\cdot 2=a$。

所以 $\binom{col[k]+1}{2}-\binom{col[k]}{2}=col[k]$。

算法总复杂度：$O(n\sqrt{n})$

下面的代码中 `deno` 表示答案的分母 (denominator)，`nume` 表示分子 (numerator)，`sqn` 表示块的大小：$\sqrt{n}$，`arr` 是输入的数组，`node` 是存储询问的结构体，`tab` 是询问序列（排序后的），`col` 同上所述。

注意：由于 `++l` 和 `--r` 的存在，下面代码中的移动区间的 4 个 `while` 循环的位置很关键，不能随意改变它们之间的位置关系。

" 关于四个循环位置的讨论"

莫队区间的移动过程，就相当于加入了 $[1,r]$ 的元素，并删除了 $[1,l-1]$ 的元素。因此，

- 对于 $l\le r$ 的情况，$[1,l-1]$ 的元素相当于被加入了一次又被删除了一次，$[l,r]$ 的元素被加入一次，$[r+1,+\infty)$ 的元素没有被加入。这个区间是合法区间。
- 对于 $l=r+1$ 的情况，$[1,r]$ 的元素相当于被加入了一次又被删除了一次，$[r+1,+\infty)$ 的元素没有被加入。这时这个区间表示空区间。
- 对于 $l>r+1$ 的情况，那么 $[r+1,l-1]$（这个区间非空）的元素被删除了一次但没有被加入，因此这个元素被加入的次数是负数。

因此，如果某时刻出现 $l > r+1$ 的情况，那么会存在一个元素，它的加入次数是负数。这在某些题目会出现问题，例如我们如果用一个 `set` 维护区间中的所有数，就会出现「需要删除 `set` 中不存在的元素」的问题。

代码中的四个 while 循环一共有 $4! = 24$ 种排列顺序。不妨设第一个循环用于操作左端点，就有以下 12 种排列（另外 12 种是对称的）。下表列出了这 12 种写法的正确性，还给出了错误写法的反例。

循环顺序	正确性	反例或注释
`l--,l++,r--,r++`	错误	$l<r<l'<r'$
`l--,l++,r++,r--`	错误	$l<r<l'<r'$
`l--,r--,l++,r++`	错误	$l<r<l'<r'$
`l--,r--,r++,l++`	正确	证明较繁琐
`l--,r++,l++,r--`	正确	
`l--,r++,r--,l++`	正确	
`l++,l--,r--,r++`	错误	$l<r<l'<r'$
`l++,l--,r++,r--`	错误	$l<r<l'<r'$
`l++,r++,l--,r--`	错误	$l<r<l'<r'$
`l++,r++,r--,l--`	错误	$l<r<l'<r'$
`l++,r--,l--,r++`	错误	$l<r<l'<r'$
`l++,r--,r++,l--`	错误	$l<r<l'<r'$

全部 24 种排列中只有 6 种是正确的，其中有 2 种的证明较繁琐，这里只给出其中 4 种的证明。

这 4 种正确写法的共同特点是，前两步先扩大区间（`l--` 或 `r++`），后两步再缩小区间（`l++` 或 `r--`）。这样写，前两步是扩大区间，可以保持 $l \le r+1$；执行完前两步后，$l \le l' \le r' \le r$ 一定成立，再执行后两步只会把区间缩小到 $[l', r']$，依然有 $l \le r+1$，因此这样写是正确的。

实现

参考代码

```cpp
#include <algorithm>
#include <cmath>
#include <cstdio>
using namespace std;
const int N = 50005;
int n, m, maxn;
int c[N];
long long sum;
int cnt[N];
long long ans1[N], ans2[N];

struct query {
  int l, r, id;

  bool operator<(const query &x) const {  // 重载 < 运算符
    if (l / maxn != x.l / maxn) return l < x.l;
```

```
    return (l / maxn) & 1 ? r < x.r : r > x.r;
  }
} a[N];

void add(int i) {
  sum += cnt[i];
  cnt[i]++;
}

void del(int i) {
  cnt[i]--;
  sum -= cnt[i];
}

long long gcd(long long a, long long b) { return b ? gcd(b, a % b) : a; }

int main() {
  scanf("%d%d", &n, &m);
  maxn = sqrt(n);
  for (int i = 1; i <= n; i++) scanf("%d", &c[i]);
  for (int i = 0; i < m; i++) scanf("%d%d", &a[i].l, &a[i].r), a[i].id = i;
  sort(a, a + m);
  for (int i = 0, l = 1, r = 0; i < m; i++) {  // 具体实现
    if (a[i].l == a[i].r) {
      ans1[a[i].id] = 0, ans2[a[i].id] = 1;
      continue;
    }
    while (l > a[i].l) add(c[--l]);
    while (r < a[i].r) add(c[++r]);
    while (l < a[i].l) del(c[l++]);
    while (r > a[i].r) del(c[r--]);
    ans1[a[i].id] = sum;
    ans2[a[i].id] = (long long)(r - l + 1) * (r - l) / 2;
  }
  for (int i = 0; i < m; i++) {
    if (ans1[i] != 0) {
      long long g = gcd(ans1[i], ans2[i]);
      ans1[i] /= g, ans2[i] /= g;
    } else
      ans2[i] = 1;
    printf("%lld/%lld\n", ans1[i], ans2[i]);
  }
  return 0;
}
```

普通莫队的优化

过程

我们看一下下面这组数据

```
// 设块的大小为 2 (假设)
1 1
2 100
3 1
4 100
```

手动模拟一下可以发现，r 指针的移动次数大概为 300 次，我们处理完第一个块之后，$l = 2, r = 100$，此时只需要移动两次 l 指针就可以得到第四个询问的答案，但是我们却将 r 指针移动到 1 来获取第三个询问的答案，再移动到 100 获取第四个询问的答案，这样多了九十几次的指针移动。我们怎么优化这个地方呢？这里我们就要用到奇偶化排序。

什么是奇偶化排序？奇偶化排序即对于属于奇数块的询问，r 按从小到大排序，对于属于偶数块的排序，r 从大到小排序，这样我们的 r 指针在处理完这个奇数块的问题后，将在返回的途中处理偶数块的问题，再向 n 移动处理下一个奇数块的问题，优化了 r 指针的移动次数，一般情况下，这种优化能让程序快 30% 左右。

实现

排序代码：

```
// 这里有个小细节等下会讲
int unit;  // 块的大小

struct node {
  int l, r, id;

  bool operator<(const node &x) const {
    return l / unit == x.l / unit
               ? (r == x.r ? 0 : ((l / unit) & 1) ^ (r < x.r))
               : l < x.l;
  }
};
```

```
struct node {
  int l, r, id;

  bool operator<(const node &x) const {
    if (l / unit != x.l / unit) return l < x.l;
    // 注意下面两行不能写小于（大于）等于，否则会出错（详见下面的小细节）
    if ((l / unit) & 1) return r < x.r;
    return r > x.r;
  }
};
```

" 小细节"

如果使用 `sort` 比较两个结构体，不能出现 $a < b$ 和 $b < a$ 同时为真的情况，否则会运行错误，详见 常见错误。

对于压行版，如果没有 `r == x.r` 的特判，当 l 属于同一奇数块且 r 相等时，会出现上面小细节中的问题（自己手动模拟一下），对于不压行版，如果写成小于（大于）等于，则也会出现同样的问题。

参考资料

- 莫队算法学习笔记 | Sengxian's Blog[2]

参考资料与注释

[1] 「国家集训队」小 Z 的袜子

[2] 莫队算法学习笔记 | Sengxian's Blog

带修改莫队

Authors: StudyingFather, Backl1ght, countercurrent-time, Ir1d, greyqz, MicDZ, ouuan, renbaoshuo, Lixuannan

请确保您已经会普通莫队算法了。如果您还不会，请先阅读前面的 普通莫队算法。

特点

普通莫队是不能带修改的。

我们可以强行让它可以修改，就像 DP 一样，可以强行加上一维**时间维**, 表示这次操作的时间。

时间维表示经历的修改次数。

即把询问 $[l,r]$ 变成 $[l,r,\text{time}]$。

那么我们的坐标也可以在时间维上移动，即 $[l,r,\text{time}]$ 多了一维可以移动的方向，可以变成：

- $[l-1,r,\text{time}]$
- $[l+1,r,\text{time}]$
- $[l,r-1,\text{time}]$
- $[l,r+1,\text{time}]$
- $[l,r,\text{time}-1]$
- $[l,r,\text{time}+1]$

这样的转移也是 $O(1)$ 的，但是我们排序又多了一个关键字，再搞搞就行了。

可以用和普通莫队类似的方法排序转移，做到 $O(n^{5/3})$。

这一次我们排序的方式是以 $n^{2/3}$ 为一块，分成了 $n^{1/3}$ 块，第一关键字是左端点所在块，第二关键字是右端点所在块，第三关键字是时间。

" 最优块长以及时间复杂度分析"

我们设序列长为 n，m 个询问，t 个修改。

带修莫队排序的第二关键字是右端点所在块编号，不同于普通莫队。

想一想，如果不把右端点分块：

- 乱序的右端点对于每个询问会移动 n 次。
- 有序的右端点会带来乱序的时间，每次询问会移动 t 次。

无论哪一种情况，带来的时间开销都无法接受。

接下来分析时间复杂度。

设块长为 s，则有 $\frac{n}{s}$ 个块。对于块 i 和块 j，记有 $q_{i,j}$ 个询问的左端点位于块 i，右端点位于块 j。

每「组」左右端点不换块的询问 (i,j)，端点每次移动 $O(s)$ 次，时间单调递增，$O(t)$。

左右端点换块的时间忽略不计。

表示一下就是：

$$
\begin{aligned}
&\sum_{i=1}^{n/s}\sum_{j=i+1}^{n/s}(q_{i,j}\cdot s+t)\\
=&ms+\left(\frac{n}{s}\right)^2 t\\
=&ms+\frac{n^2t}{s^2}
\end{aligned}
$$

考虑求导求此式极小值。设 $f(s)=ms+\frac{n^2t}{s^2}$。那 $f'(s)=m-\frac{2n^2t}{s^3}=0$。

得 $s=\sqrt[3]{\frac{2n^2t}{m}}=\frac{2^{1/3}n^{2/3}t^{1/3}}{m^{1/3}}=s_0$。

也就是当块长取 $\frac{n^{2/3}t^{1/3}}{m^{1/3}}$ 时有最优时间复杂度 $O\left(n^{2/3}m^{2/3}t^{1/3}\right)$。

常说的 $O\left(n^{5/3}\right)$ 便是把 n, m, t 当做同数量级的时间复杂度。

实际操作中还是推荐设定 $n^{2/3}$ 为块长。

例题

" 例题 「 国家集训队 」 数颜色 / 维护队列[1]"

题目大意：给你一个序列，M 个操作，有两种操作：

1. 修改序列上某一位的数字
2. 询问区间 $[l, r]$ 中数字的种类数（多个相同的数字只算一个）

我们不难发现，如果不带操作 1（修改）的话，我们就能轻松用普通莫队解决。

但是题目还带单点修改，所以用**带修改的莫队**。

过程

先考虑普通莫队的做法：

- 每次扩大区间时，每加入一个数字，则统计它已经出现的次数，如果加入前这种数字出现次数为 0，则说明这是一种新的数字，答案 $+1$。然后这种数字的出现次数 $+1$。
- 每次减小区间时，每删除一个数字，则统计它删除后的出现次数，如果删除后这种数字出现次数为 0，则说明这种数字已经从当前的区间内删光了，也就是当前区间减少了一种颜色，答案 -1。然后这种数字的出现次数 -1。

现在再来考虑修改：

- 单点修改，把某一位的数字修改掉。假如我们是从一个经历修改次数为 i 的询问转移到一个经历修改次数为 j 的询问上，且 $i<j$ 的话，我们就需要把第 $i+1$ 个到第 j 个修改强行加上。
- 假如 $j<i$ 的话，则需要把第 i 个到第 $j+1$ 个修改强行还原。

怎么强行加上一个修改呢？假设一个修改是修改第 pos 个位置上的颜色，原本 pos 上的颜色为 a，修改后颜色为 b，还假设当前莫队的区间扩展到了 $[l, r]$。

- 加上这个修改：我们首先判断 pos 是否在区间 $[l, r]$ 内。如果是的话，我们等于是从区间中删掉颜色 a，加上颜色 b，并且当前颜色序列的第 pos 项的颜色改成 b。如果不在区间 $[l, r]$ 内的话，我们就直接修改当前颜色序列的第 pos 项为 b。
- 还原这个修改：等于加上一个修改第 pos 项、把颜色 b 改成颜色 a 的修改。

因此这道题就这样用带修改莫队轻松解决啦！

实现

参考代码

```cpp
#include <algorithm>
#include <cmath>
#include <iostream>

#define int long long
#define endl '\n'
```

```
using namespace std;

int qsize;

struct query {
  int id, t, l, r;

  inline bool operator<(query b) {
    if (l / qsize != b.l / qsize) {
      return l / qsize > b.l / qsize;
    } else if (r / qsize != b.r / qsize) {
      return r / qsize > b.r / qsize;
    } else {
      return t > b.t;
    }
  }
} q[150009];

struct operation {
  int p, x;
} r[150009];

char op;
int n, m, x, y, cur, qcnt, rcnt, mp[1500009], a[150009], ans[150009];

inline void add(int x) {
  if (!mp[x]) {
    cur += 1;
  }
  mp[x] += 1;
}

inline void del(int x) {
  mp[x] -= 1;
  if (!mp[x]) {
    cur -= 1;
  }
}

inline void process() {
  sort(q + 1, q + qcnt + 1);
  int L = 1, R = 0, last = 0;
  for (int i = 1; i <= qcnt; i++) {
    while (R < q[i].r) {
      add(a[++R]);
    }
    while (R > q[i].r) {
      del(a[R--]);
    }
    while (L > q[i].l) {
      add(a[--L]);
    }
    while (L < q[i].l) {
      del(a[L++]);
```

```
    }
    while (last < q[i].t) {
      last += 1;
      if (r[last].p >= L && r[last].p <= R) {
        add(r[last].x);
        del(a[r[last].p]);
      }
      swap(a[r[last].p], r[last].x);
    }
    while (last > q[i].t) {
      if (r[last].p >= L && r[last].p <= R) {
        add(r[last].x);
        del(a[r[last].p]);
      }
      swap(a[r[last].p], r[last].x);
      last -= 1;
    }
    ans[q[i].id] = cur;
  }
}

signed main() {
  cin.tie(0);
  cout.tie(0);
  ios::sync_with_stdio(false);
  cin >> n >> m;
  qsize = pow(n, 2.0 / 3.0);
  for (int i = 1; i <= n; i++) {
    cin >> a[i];
  }
  for (int i = 1; i <= m; i++) {
    cin >> op >> x >> y;
    if (op == 'Q') {
      q[++qcnt] = {qcnt, rcnt, x, y};
    } else if (op == 'R') {
      r[++rcnt] = {x, y};
    }
  }
  process();
  for (int i = 1; i <= qcnt; i++) {
    cout << ans[i] << endl;
  }
}
```

参考资料与注释

[1] 「国家集训队」数颜色/维护队列

树上莫队

Authors: StudyingFather, Backl1ght, countercurrent-time, Ir1d, greyqz, MicDZ, ouuan, Linky

括号序树上莫队

一般的莫队只能处理线性问题，我们要把树强行压成序列。

我们可以将树的括号序跑下来，把括号序分块，在括号序上跑莫队。

具体怎么做呢?

过程

dfs 一棵树，然后如果 dfs 到 x 点，就 `push_back(x)`，dfs 完 x 点，就直接 `push_back(-x)`，然后我们在挪动指针的时候，

- 新加入的值是 x —>`add(x)`
- 新加入的值是 - x —>`del(x)`
- 新删除的值是 x —>`del(x)`
- 新删除的值是 - x —>`add(x)`

这样的话，我们就把一棵树处理成了序列。

例题

" 例题 「WC2013」 糖果公园[1]"

题意：给你一棵树，树上第 i 个点颜色为 c_i，每次询问一条路径 u_i,v_i，求这条路径上的

$\sum_c val_c \sum_{i=1}^{cnt_c} w_i$

其中：val 表示该颜色的价值，cnt 表示颜色出现的次数，w 表示该颜色出现 i 次后的价值

过程

先把树变成序列，然后每次添加/删除一个点，这个点的对答案的的贡献是可以在 $O(1)$ 时间内获得的，即 $val_c \times w_{cnt_{c+1}}$

发现因为他会把起点的子树也扫了一遍，产生多余的贡献，怎么办呢?

因为扫的过程中起点的子树里的点肯定会被扫两次，但贡献为 0。

所以可以开一个 vis 数组，每次扫到点 x，就把 vis_x 异或上 1。

如果 $vis_x = 0$，那这个点的贡献就可以不计。

所以可以用树上莫队来求。

修改的话，加上一维时间维即可，变成带修改树上莫队。

然后因为所包含的区间内可能没有 LCA，对于没有的情况要将多余的贡献删除，然后就完事了。

实现

参考代码

```
#include <algorithm>
#include <cmath>
#include <cstdio>
using namespace std;

const int maxn = 200010;

int f[maxn], g[maxn], id[maxn], head[maxn], cnt, last[maxn], dep[maxn],
    fa[maxn][22], v[maxn], w[maxn];
int block, index, n, m, q;
int pos[maxn], col[maxn], app[maxn];
```

```
bool vis[maxn];
long long ans[maxn], cur;

struct edge {
  int to, nxt;
} e[maxn];

int cnt1 = 0, cnt2 = 0;  // 时间戳

struct query {
  int l, r, t, id;

  bool operator<(const query &b) const {
    return (pos[l] < pos[b.l]) || (pos[l] == pos[b.l] && pos[r] < pos[b.r]) ||
           (pos[l] == pos[b.l] && pos[r] == pos[b.r] && t < b.t);
  }
} a[maxn], b[maxn];

void addedge(int x, int y) {
  e[++cnt] = (edge){y, head[x]};
  head[x] = cnt;
}

void dfs(int x) {
  id[f[x] = ++index] = x;
  for (int i = head[x]; i; i = e[i].nxt) {
    if (e[i].to != fa[x][0]) {
      fa[e[i].to][0] = x;
      dep[e[i].to] = dep[x] + 1;
      dfs(e[i].to);
    }
  }
  id[g[x] = ++index] = x;  // 括号序
}

int lca(int x, int y) {
  if (dep[x] < dep[y]) swap(x, y);
  if (dep[x] != dep[y]) {
    int dis = dep[x] - dep[y];
    for (int i = 20; i >= 0; i--)
      if (dis >= (1 << i)) dis -= 1 << i, x = fa[x][i];
  }  // 爬到同一高度
  if (x == y) return x;
  for (int i = 20; i >= 0; i--) {
    if (fa[x][i] != fa[y][i]) x = fa[x][i], y = fa[y][i];
  }
  return fa[x][0];
}

void add(int x) {
  if (vis[x])
    cur -= (long long)v[col[x]] * w[app[col[x]]--];
  else
    cur += (long long)v[col[x]] * w[++app[col[x]]];
```

```
  vis[x] ^= 1;
}

void modify(int x, int t) {
  if (vis[x]) {
    add(x);
    col[x] = t;
    add(x);
  } else
    col[x] = t;
}  // 在时间维上移动

int main() {
  scanf("%d%d%d", &n, &m, &q);
  for (int i = 1; i <= m; i++) scanf("%d", &v[i]);
  for (int i = 1; i <= n; i++) scanf("%d", &w[i]);
  for (int i = 1; i < n; i++) {
    int x, y;
    scanf("%d%d", &x, &y);
    addedge(x, y);
    addedge(y, x);
  }
  for (int i = 1; i <= n; i++) {
    scanf("%d", &last[i]);
    col[i] = last[i];
  }
  dfs(1);
  for (int j = 1; j <= 20; j++)
    for (int i = 1; i <= n; i++)
      fa[i][j] = fa[fa[i][j - 1]][j - 1];  // 预处理祖先
  int block = pow(index, 2.0 / 3);
  for (int i = 1; i <= index; i++) {
    pos[i] = (i - 1) / block;
  }
  while (q--) {
    int opt, x, y;
    scanf("%d%d%d", &opt, &x, &y);
    if (opt == 0) {
      b[++cnt2].l = x;
      b[cnt2].r = last[x];
      last[x] = b[cnt2].t = y;
    } else {
      if (f[x] > f[y]) swap(x, y);
      a[++cnt1] = (query){lca(x, y) == x ? f[x] : g[x], f[y], cnt2, cnt1};
    }
  }
  sort(a + 1, a + cnt1 + 1);
  int L, R, T;  // 指针坐标
  L = R = 0;
  T = 1;
  for (int i = 1; i <= cnt1; i++) {
    while (T <= a[i].t) {
      modify(b[T].l, b[T].t);
      T++;
```

```
    }
    while (T > a[i].t) {
      modify(b[T].l, b[T].r);
      T--;
    }
    while (L > a[i].l) {
      L--;
      add(id[L]);
    }
    while (L < a[i].l) {
      add(id[L]);
      L++;
    }
    while (R > a[i].r) {
      add(id[R]);
      R--;
    }
    while (R < a[i].r) {
      R++;
      add(id[R]);
    }
    int x = id[L], y = id[R];
    int llca = lca(x, y);
    if (x != llca && y != llca) {
      add(llca);
      ans[a[i].id] = cur;
      add(llca);
    } else
      ans[a[i].id] = cur;
  }
  for (int i = 1; i <= cnt1; i++) {
    printf("%lld\n", ans[i]);
  }
  return 0;
}
```

真 · 树上莫队

上面的树上莫队只是将树转化成了链，下面的才是真正的树上莫队。

由于莫队相关的问题都是模板题，因此实现部分不做太多解释

询问的排序

首先我们知道莫队的是基于分块的算法，所以我们需要找到一种树上的分块方法来保证时间复杂度。

条件：

- 属于同一块的节点之间的距离不超过给定块的大小
- 每个块中的节点不能太多也不能太少
- 每个节点都要属于一个块
- 编号相邻的块之间的距离不能太大

了解了这些条件后，我们看到这样一道题 「SCOI2005」 王室联邦[2]。

在这道题的基础上我们只要保证最后一个条件就可以解决分块的问题了。

思路

令 lim 为希望块的大小，首先，对于整个树 dfs，当子树的大小大于 lim 时，就将它们分在一块，容易想到：对于根，可能会剩下一些点，于是将这些点分在最后一个块里。

做法：用栈维护当前节点作为父节点访问它的子节点，当从栈顶到父节点的距离大于希望块的大小时，弹出这部分元素分为一块，最后剩余的一块单独作为一块。

最后的排序方法：若第一维时间戳大于第二维，交换它们，按第一维所属块为第一关键字，第二维时间戳为第二关键字排序。

指针的移动

过程

容易想到，我们可以标记被计入答案的点，让指针直接向目标移动，同时取反路径上的点。

但是，这样有一个问题，若指针一开始都在 x 上，显然 x 被标记，当两个指针向同一子节点移动（还有许多情况）时，x 应该不被标记，但实际情况是 x 被标记，因为两个指针分别标记了一次，抵消了。

如何解决呢?

有一个很显然的性质：这些点肯定是某些 LCA，因为 LCA 处才有可能被重复撤销导致撤销失败。

所以我们每次不标记 LCA，到需要询问答案时再将 LCA 标记，然后再撤销。

实现

```
// 取反路径上除 LCA 以外的所有节点
void move(int x, int y) {
  if (dp[x] < dp[y]) swap(x, y);
  while (dp[x] > dp[y]) update(x), x = fa[x];
  while (x != y) update(x), update(y), x = fa[x], y = fa[y];
  // x!=y 保证 LCA 没被取反
}
```

对于求 LCA，我们可以用树剖，然后我们就可以把分块的步骤放到树剖的第一次 dfs 里面，时间戳也可以直接用第二次 dfs 的 dfs 序。

```
int bl[100002], bls = 0;  // 属于的块，块的数量
unsigned step;            // 块大小
int fa[100002], dp[100002], hs[100002] = {0}, sz[100002] = {0};
// 父节点，深度，重儿子，大小
stack<int> sta;

void dfs1(int x) {
  sz[x] = 1;
  unsigned ss = sta.size();
  for (int i = head[x]; i; i = nxt[i])
    if (ver[i] != fa[x]) {
      fa[ver[i]] = x;
      dp[ver[i]] = dp[x] + 1;
      dfs1(ver[i]);
      sz[x] += sz[ver[i]];
      if (sz[ver[i]] > sz[hs[x]]) hs[x] = ver[i];
      if (sta.size() - ss >= step) {
        bls++;
        while (sta.size() != ss) bl[sta.top()] = bls, sta.pop();
      }
    }
```

```
    sta.push(x);
  }

// main
if (!sta.empty()) {
  bls++;  // 这一行可写可不写
  while (!sta.empty()) bl[sta.top()] = bls, sta.pop();
}
```

时间复杂度

重点到了，这里关系到块的大小取值。

设块的大小为 $unit$:

- 对于 x 指针，由于每个块中节点的距离在 $unit$ 左右，每个块中 x 指针移动 $unit^2$ 次（$unit \times dis_{\max}$），共计 $n \times unit$ 次（$unit^2 \times (\frac{n}{unit})$）;
- 对于 y 指针，每个块中最多移动 $O(n)$ 次，共计 $\frac{n^2}{unit}$ 次（$n \times (\frac{n}{unit})$）。

加起来大概在根号处取得最小值（由于树上莫队块的大小不固定，所以不一定要严格按照）。

例题「WC2013」糖果公园

由于多了时间维，块的大小取到 $n^{0.6}$ 的样子就差不多了。

参考代码

```
#include <bits/stdc++.h>
using namespace std;

int gi() {
  int x, c, op = 1;
  while (c = getchar(), c < '0' || c > '9')
    if (c == '-') op = -op;
  x = c ^ 48;
  while (c = getchar(), c >= '0' && c <= '9')
    x = (x << 3) + (x << 1) + (c ^ 48);
  return x * op;
}

int head[100002], nxt[200004], ver[200004], tot = 0;

void add(int x, int y) {
  ver[++tot] = y, nxt[tot] = head[x], head[x] = tot;
  ver[++tot] = x, nxt[tot] = head[y], head[y] = tot;
}

int bl[100002], bls = 0;
unsigned step;
int fa[100002], dp[100002], hs[100002] = {0}, sz[100002] = {0}, top[100002],
                            id[100002];
stack<int> sta;

void dfs1(int x) {
  sz[x] = 1;
  unsigned ss = sta.size();
  for (int i = head[x]; i; i = nxt[i])
```

```
    if (ver[i] != fa[x]) {
      fa[ver[i]] = x, dp[ver[i]] = dp[x] + 1;
      dfs1(ver[i]);
      sz[x] += sz[ver[i]];
      if (sz[ver[i]] > sz[hs[x]]) hs[x] = ver[i];
      if (sta.size() - ss >= step) {
        bls++;
        while (sta.size() != ss) bl[sta.top()] = bls, sta.pop();
      }
    }
  sta.push(x);
}

int cnt = 0;

void dfs2(int x, int hf) {
  top[x] = hf, id[x] = ++cnt;
  if (!hs[x]) return;
  dfs2(hs[x], hf);
  for (int i = head[x]; i; i = nxt[i])
    if (ver[i] != fa[x] && ver[i] != hs[x]) dfs2(ver[i], ver[i]);
}

int lca(int x, int y) {
  while (top[x] != top[y]) {
    if (dp[top[x]] < dp[top[y]]) swap(x, y);
    x = fa[top[x]];
  }
  return dp[x] < dp[y] ? x : y;
}

struct qu {
  int x, y, t, id;

  bool operator<(const qu a) const {
    return bl[x] == bl[a.x] ? (bl[y] == bl[a.y] ? t < a.t : bl[y] < bl[a.y])
                            : bl[x] < bl[a.x];
  }
} q[100001];

int qs = 0;

struct ch {
  int x, y, b;
} upd[100001];

int ups = 0;
long long ans[100001];
int b[100001] = {0};
int a[100001];
long long w[100001];
long long v[100001];
long long now = 0;
bool vis[100001] = {0};
```

```
void back(int t) {
  if (vis[upd[t].x]) {
    now -= w[b[upd[t].y]--] * v[upd[t].y];
    now += w[++b[upd[t].b]] * v[upd[t].b];
  }
  a[upd[t].x] = upd[t].b;
}

void change(int t) {
  if (vis[upd[t].x]) {
    now -= w[b[upd[t].b]--] * v[upd[t].b];
    now += w[++b[upd[t].y]] * v[upd[t].y];
  }
  a[upd[t].x] = upd[t].y;
}

void update(int x) {
  if (vis[x])
    now -= w[b[a[x]]--] * v[a[x]];
  else
    now += w[++b[a[x]]] * v[a[x]];
  vis[x] ^= 1;
}

void move(int x, int y) {
  if (dp[x] < dp[y]) swap(x, y);
  while (dp[x] > dp[y]) update(x), x = fa[x];
  while (x != y) update(x), update(y), x = fa[x], y = fa[y];
}

int main() {
  int n = gi(), m = gi(), k = gi();
  step = (int)pow(n, 0.6);
  for (int i = 1; i <= m; i++) v[i] = gi();
  for (int i = 1; i <= n; i++) w[i] = gi();
  for (int i = 1; i < n; i++) add(gi(), gi());
  for (int i = 1; i <= n; i++) a[i] = gi();
  for (int i = 1; i <= k; i++)
    if (gi())
      q[++qs].x = gi(), q[qs].y = gi(), q[qs].t = ups, q[qs].id = qs;
    else
      upd[++ups].x = gi(), upd[ups].y = gi();
  for (int i = 1; i <= ups; i++) upd[i].b = a[upd[i].x], a[upd[i].x] = upd[i].y;
  for (int i = ups; i; i--) back(i);
  fa[1] = 1;
  dfs1(1), dfs2(1, 1);
  if (!sta.empty()) {
    bls++;
    while (!sta.empty()) bl[sta.top()] = bls, sta.pop();
  }
  for (int i = 1; i <= n; i++)
    if (id[q[i].x] > id[q[i].y]) swap(q[i].x, q[i].y);
  sort(q + 1, q + qs + 1);
```

```
  int x = 1, y = 1, t = 0;
  for (int i = 1; i <= qs; i++) {
    if (x != q[i].x) move(x, q[i].x), x = q[i].x;
    if (y != q[i].y) move(y, q[i].y), y = q[i].y;
    int f = lca(x, y);
    update(f);
    while (t < q[i].t) change(++t);
    while (t > q[i].t) back(t--);
    ans[q[i].id] = now;
    update(f);
  }
  for (int i = 1; i <= qs; i++) printf("%lld\n", ans[i]);
  return 0;
}
```

参考资料与注释

[1] 「WC2013」糖果公园

[2] 「SCOI2005」王室联邦

回滚莫队

> **Authors:** StudyingFather, Backl1ght, countercurrent-time, Ir1d, greyqz, MicDZ, ouuan, YOYO-UIAT

引入

有些题目在区间转移时，可能会出现增加或者删除无法实现的问题。在只有增加不可实现或者只有删除不可实现的时候，就可以使用回滚莫队在 $O(n\sqrt{m})$ 的时间内解决问题。回滚莫队的核心思想就是：既然只能实现一个操作，那么就只使用一个操作，剩下的交给回滚解决。

回滚莫队分为只使用增加操作的回滚莫队和只使用删除操作的回滚莫队。以下仅介绍只使用增加操作的回滚莫队，只使用删除操作的回滚莫队和只使用增加操作的回滚莫队只在算法实现上有一点区别，故不再赘述。

例题 JOISC 2014 Day1 历史研究[1]

给你一个长度为 n 的数组 A 和 m 个询问 $(1 \leq n, m \leq 10^5)$，每次询问一个区间 $[L, R]$ 内重要度最大的数字，要求**输出其重要度**。一个数字 i 重要度的定义为 i 乘上 i 在区间内出现的次数。

在这个问题中，在增加的过程中更新答案是很好实现的，但是在删除的过程中更新答案是不好实现的。因为如果增加会影响答案，那么新答案必定是刚刚增加的数字的重要度，而如果删除过后区间重要度最大的数字改变，我们很难确定新的重要度最大的数字是哪一个。所以，普通的莫队很难解决这个问题。

过程

- 对原序列进行分块，对询问按以左端点所属块编号升序为第一关键字，右端点升序为第二关键字的方式排序。
- 按顺序处理询问：
  - 如果询问左端点所属块 B 和上一个询问左端点所属块的不同，那么将莫队区间的左端点初始化为 B 的右端点加 1, 将莫队区间的右端点初始化为 B 的右端点；
  - 如果询问的左右端点所属的块相同，那么直接扫描区间回答询问；
  - 如果询问的左右端点所属的块不同：
    * 如果询问的右端点大于莫队区间的右端点，那么不断扩展右端点直至莫队区间的右端点等于询问的右端点；

* 不断扩展莫队区间的左端点直至莫队区间的左端点等于询问的左端点；
* 回答询问；
* 撤销莫队区间左端点的改动，使莫队区间的左端点回滚到 B 的右端点加 1。

复杂度证明

假设回滚莫队的分块大小是 b：

- 对于左、右端点在同一个块内的询问，可以在 $O(b)$ 时间内计算；
- 对于其他询问，考虑左端点在相同块内的询问，它们的右端点单调递增，移动右端点的时间复杂度是 $O(n)$，而左端点单次询问的移动不超过 b，因为有 $\frac{n}{b}$ 个块，所以总复杂度是 $O(mb+\frac{n^2}{b})$，取 $b=\frac{n}{\sqrt{m}}$ 最优，时间复杂度为 $O(n\sqrt{m})$。

实现

参考代码

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N = 1e5 + 5;
int n, q;
int x[N], t[N], m;

struct Query {
  int l, r, id;
} Q[N];

int pos[N], L[N], R[N], sz, tot;
int cnt[N], __cnt[N];
ll ans[N];

bool cmp(const Query& A, const Query& B) {
  if (pos[A.l] == pos[B.l]) return A.r < B.r;
  return pos[A.l] < pos[B.l];
}

void build() {
  sz = sqrt(n);
  tot = n / sz;
  for (int i = 1; i <= tot; i++) {
    L[i] = (i - 1) * sz + 1;
    R[i] = i * sz;
  }
  if (R[tot] < n) {
    ++tot;
    L[tot] = R[tot - 1] + 1;
    R[tot] = n;
  }
}

void Add(int v, ll& Ans) {
  ++cnt[v];
```

```
  Ans = max(Ans, 1LL * cnt[v] * t[v]);
}

void Del(int v) { --cnt[v]; }

int main() {
  scanf("%d %d", &n, &q);
  for (int i = 1; i <= n; i++) scanf("%d", &x[i]), t[++m] = x[i];
  for (int i = 1; i <= q; i++) scanf("%d %d", &Q[i].l, &Q[i].r), Q[i].id = i;

  build();

  // 对询问进行排序
  for (int i = 1; i <= tot; i++)
    for (int j = L[i]; j <= R[i]; j++) pos[j] = i;
  sort(Q + 1, Q + 1 + q, cmp);

  // 离散化
  sort(t + 1, t + 1 + m);
  m = unique(t + 1, t + 1 + m) - (t + 1);
  for (int i = 1; i <= n; i++) x[i] = lower_bound(t + 1, t + 1 + m, x[i]) - t;

  int l = 1, r = 0, last_block = 0, __l;
  ll Ans = 0, tmp;
  for (int i = 1; i <= q; i++) {
    // 询问的左右端点同属于一个块则暴力扫描回答
    if (pos[Q[i].l] == pos[Q[i].r]) {
      for (int j = Q[i].l; j <= Q[i].r; j++) ++__cnt[x[j]];
      for (int j = Q[i].l; j <= Q[i].r; j++)
        ans[Q[i].id] = max(ans[Q[i].id], 1LL * t[x[j]] * __cnt[x[j]]);
      for (int j = Q[i].l; j <= Q[i].r; j++) --__cnt[x[j]];
      continue;
    }

    // 访问到了新的块则重新初始化莫队区间
    if (pos[Q[i].l] != last_block) {
      while (r > R[pos[Q[i].l]]) Del(x[r]), --r;
      while (l < R[pos[Q[i].l]] + 1) Del(x[l]), ++l;
      Ans = 0;
      last_block = pos[Q[i].l];
    }

    // 扩展右端点
    while (r < Q[i].r) ++r, Add(x[r], Ans);
    __l = l;
    tmp = Ans;

    // 扩展左端点
    while (__l > Q[i].l) --__l, Add(x[__l], tmp);
    ans[Q[i].id] = tmp;

    // 回滚
    while (__l < l) Del(x[__l]), ++__l;
  }
```

```
    for (int i = 1; i <= q; i++) printf("%lld\n", ans[i]);
    return 0;
}
```

参考资料

- 回滚莫队及其简单运用 | Parsnip's Blog[2]

参考资料与注释

[1] JOISC 2014 Day1 历史研究

[2] 回滚莫队及其简单运用 | Parsnip's Blog

二维莫队

二维莫队，顾名思义就是每个状态有四个方向可以扩展。

二维莫队每次移动指针要操作一行或者一列的数，具体实现方式与普通的一维莫队类似，这里不再赘述。这里重点讲块长选定部分。

块长选定

记询问次数为 q，当前矩阵的左上角坐标为 (x_1, y_1)，右下角坐标为 (x_2, y_2)，取块长为 B。

那么指针 x_1 移动了 $\Theta(q \cdot B)$ 次，而指针 y_2 移动了 $\Theta(n^4 \cdot B^{-3})$ 次。

所以只需令 $q \cdot B = n^4 \cdot B^{-3}$，即 $B = n \cdot q^{-\frac{1}{4}}$ 即可。

注意这样计算 B 的结果**可能为** 0，**注意特判**。

最终，计算部分时间复杂度是 $\Theta(n^2 \cdot q^{\frac{3}{4}})$，加上对询问的排序过程，总时间复杂度为 $\Theta(n^2 \cdot q^{\frac{3}{4}} + q \log q)$。

例题 1

"BZOJ 2639 **矩形计算**[1]"

输入一个 $n \times m$ 的矩阵，矩阵的每一个元素都是一个整数，然后有 q 个询问，每次询问一个子矩阵的权值。矩阵的权值是这样定义的，对于一个整数 x，如果它在该矩阵中出现了 p 次，那么它给该矩阵的权值就贡献 p^2。

数据范围：$1 \leq n, m \leq 200$，$0 \leq q \leq 10^5$，$|$ 矩阵元素大小 $| \leq 2 \times 10^9$。

" **解题思路**"

先离散化，二维莫队时用一个数组记录每个数当前出现的次数即可。

" **示例代码**"

```
#include <bits/stdc++.h>
using namespace std;

int n, m, q, a[201][201];
long long ans[100001];
int disc[250001], cntdisc;  // 离散化用

int blocklen, counts[40001];
long long now;
```

```cpp
struct Question {
  int x1, y1, x2, y2, qid;

  bool operator<(Question tmp) const {
    if (x1 / blocklen != tmp.x1 / blocklen) return x1 < tmp.x1;
    if (y1 / blocklen != tmp.y1 / blocklen) return y1 < tmp.y1;
    if (x2 / blocklen != tmp.x2 / blocklen) return x2 < tmp.x2;
    return y2 < tmp.y2;
  }
} Q[100001];

int Qcnt;

void mo_algo_row(int id, int val, int Y1, int Y2) {
  for (int i = Y1; i <= Y2; i++)
    now -= (long long)counts[a[id][i]] * counts[a[id][i]],
        counts[a[id][i]] += val,
        now += (long long)counts[a[id][i]] * counts[a[id][i]];
}

void mo_algo_column(int id, int val, int X1, int X2) {
  for (int i = X1; i <= X2; i++)
    now -= (long long)counts[a[i][id]] * counts[a[i][id]],
        counts[a[i][id]] += val,
        now += (long long)counts[a[i][id]] * counts[a[i][id]];
}

void mo_algo() {
  blocklen = pow(n * m, 0.5) / pow(q, 0.25);
  if (blocklen < 1) blocklen = 1;
  sort(Q + 1, Q + 1 + Qcnt);

  int X1 = 1, Y1 = 1, X2 = 0, Y2 = 0;
  for (int i = 1; i <= Qcnt; i++) {
    while (X1 > Q[i].x1) mo_algo_row(--X1, 1, Y1, Y2);
    while (X2 < Q[i].x2) mo_algo_row(++X2, 1, Y1, Y2);
    while (Y1 > Q[i].y1) mo_algo_column(--Y1, 1, X1, X2);
    while (Y2 < Q[i].y2) mo_algo_column(++Y2, 1, X1, X2);
    while (X1 < Q[i].x1) mo_algo_row(X1++, -1, Y1, Y2);
    while (X2 > Q[i].x2) mo_algo_row(X2--, -1, Y1, Y2);
    while (Y1 < Q[i].y1) mo_algo_column(Y1++, -1, X1, X2);
    while (Y2 > Q[i].y2) mo_algo_column(Y2--, -1, X1, X2);
    ans[Q[i].qid] = now;
  }
}

int main() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++)
      scanf("%d", a[i] + j), disc[++cntdisc] = a[i][j];
  sort(disc + 1, disc + 1 + cntdisc);
  cntdisc = unique(disc + 1, disc + cntdisc + 1) - disc - 1;
```

```
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++)
      a[i][j] = lower_bound(disc + 1, disc + 1 + cntdisc, a[i][j]) - disc;
  scanf("%d", &q);
  for (int i = 1; i <= q; i++) {
    int x1, y1, x2, y2;
    scanf("%d%d%d%d", &x1, &y1, &x2, &y2);
    if (x1 > x2) swap(x1, x2);
    if (y1 > y2) swap(y1, y2);
    Q[++Qcnt] = {x1, y1, x2, y2, i};
  }

  mo_algo();
  for (int i = 1; i <= q; ++i) printf("%lld\n", ans[i]);
  return 0;
}
```

例题 2

"洛谷 P1527 [国家集训队] 矩阵乘法[2]"

给你一个 $n \times n$ 的矩阵，q 次询问，每次询问一个子矩形的第 k 小数。

数据范围：$1 \le n \le 500$，$1 \le q \le 6 \times 10^4$，$0 \le a_{i,j} \le 10^9$。

首先和上一题一样，需要离散化整个矩阵。但是需要注意，本题除了需要对数值进行分块，还需要对数值的值域进行分块，才能求出答案。

这里还需要用到奇偶化排序进行优化，具体内容请见 普通莫队算法。

对于本题而言，时间限制不那么宽，注意代码常数的处理。取的块长计算值普遍较小，n，q 都取最大值时块长大约在 11 左右，可以直接设定为常数来节约代码耗时。

" 示例代码"

```
#include <bits/stdc++.h>
using namespace std;

void read(int& a) {
  a = 0;
  char c;
  while ((c = getchar()) < 48)
    ;
  do a = (a << 3) + (a << 1) + (c ^ 48);
  while ((c = getchar()) > 47);
}

void write(int x) {
  if (x > 9) write(x / 10);
  putchar(x % 10 + '0');
}

int n, q, a[501][501], ans[60001];
int disc[250001], cntdisc;  // 离散化用
int nn;
```

```cpp
int blockId[501], blocklen;             // 分块
int rangeblockId[250001], rangeblocklen;  // 值域分块
int counts[250001], countsum[501];       // 该值次数及值域块总和

struct Position {
  int x, y;
};

vector<Position> pos[250001];

struct Question {
  int x1, y1, x2, y2, k, qid;

  bool operator<(Question tmp) const {
    if (blockId[x1] != blockId[tmp.x1]) return blockId[x1] < blockId[tmp.x1];
    if (blockId[y1] != blockId[tmp.y1])
      return blockId[x1] & 1 ? y1 < tmp.y1 : y1 > tmp.y1;
    if (blockId[y2] != blockId[tmp.y2])
      return blockId[y1] & 1 ? y2 < tmp.y2 : y2 > tmp.y2;
    else
      return blockId[y2] & 1 ? x2 < tmp.x2 : x2 > tmp.x2;
  }
} Q[60001];

int Qcnt;

void mo_algo() {
  blocklen = 11;
  for (int i = 1; i <= n; ++i) blockId[i] = (i - 1) / blocklen + 1;
  rangeblocklen = n + 1;
  for (int i = 1; i <= nn; ++i) rangeblockId[i] = (i - 1) / rangeblocklen + 1;
  counts[a[1][1]] = countsum[rangeblockId[a[1][1]]] = 1;
  sort(Q + 1, Q + 1 + Qcnt);

  int L = 1, R = 1, D = 1, U = 1;
  for (int i = 1; i <= q; ++i) {
    while (R < Q[i].y2) {
      ++R;
      for (int i = U; i <= D; ++i)
        ++counts[a[i][R]], ++countsum[rangeblockId[a[i][R]]];
    }
    while (L > Q[i].y1) {
      --L;
      for (int i = U; i <= D; ++i)
        ++counts[a[i][L]], ++countsum[rangeblockId[a[i][L]]];
    }
    while (D < Q[i].x2) {
      ++D;
      for (int i = L; i <= R; ++i)
        ++counts[a[D][i]], ++countsum[rangeblockId[a[D][i]]];
    }
    while (U > Q[i].x1) {
      --U;
      for (int i = L; i <= R; ++i)
```

```
        ++counts[a[U][i]], ++countsum[rangeblockId[a[U][i]]];
    }
    while (R > Q[i].y2) {
      for (int i = U; i <= D; ++i)
        --counts[a[i][R]], --countsum[rangeblockId[a[i][R]]];
      --R;
    }
    while (L < Q[i].y1) {
      for (int i = U; i <= D; ++i)
        --counts[a[i][L]], --countsum[rangeblockId[a[i][L]]];
      ++L;
    }
    while (D > Q[i].x2) {
      for (int i = L; i <= R; ++i)
        --counts[a[D][i]], --countsum[rangeblockId[a[D][i]]];
      --D;
    }
    while (U < Q[i].x1) {
      for (int i = L; i <= R; ++i)
        --counts[a[U][i]], --countsum[rangeblockId[a[U][i]]];
      ++U;
    }
    int res = 1, cnt = 0;
    while (cnt + countsum[res] < Q[i].k && res <= rangeblockId[nn])
      cnt += countsum[res], ++res;
    res = (res - 1) * rangeblocklen + 1;
    while (cnt + counts[res] < Q[i].k && res <= nn) cnt += counts[res], ++res;
    ans[Q[i].qid] = disc[res];
  }
}

int main() {
  read(n);
  read(q);
  nn = n * n;
  for (int i = 1; i <= n; ++i)
    for (int j = 1; j <= n; ++j) {
      int x;
      read(x);
      a[i][j] = disc[++cntdisc] = x;
    }
  sort(disc + 1, disc + 1 + cntdisc);
  cntdisc = unique(disc + 1, disc + cntdisc + 1) - disc - 1;
  for (int i = 1; i <= n; ++i)
    for (int j = 1; j <= n; ++j)
      a[i][j] = lower_bound(disc + 1, disc + 1 + cntdisc, a[i][j]) - disc;
  for (int i = 1; i <= q; ++i) {
    int x1, y1, x2, y2, k;
    read(x1);
    read(y1);
    read(x2);
    read(y2);
    read(k);
    Q[++Qcnt] = {x1, y1, x2, y2, k, i};
```

```
    }

    mo_algo();
    for (int i = 1; i <= q; ++i) write(ans[i]), puts("");
    return 0;
}
```

参考资料与注释

[1] BZOJ 2639 矩形计算

[2] 洛谷 P1527 [国家集训队] 矩阵乘法

莫队二次离线

Authors: Lyccrius

例题 1

"Luogu P5047 [Ynoi2019 **模拟赛**] Yuno loves sqrt technology II[1]"

给你一个长为 n 的序列 a，m 次询问，每次查询一个区间的逆序对数。

数据范围：$1 \le n, m \le 10^5$，$0 \le a_i \le 10^9$。

查询区间逆序对数，在使用莫队的同时维护一颗权值线段树或权值树状数组，可以在 $O(n\sqrt{n}\log n)$ 的时间复杂度内解决该问题。当然，取块长 $T = \sqrt{n\log n}$ 更优。

可是这样的复杂度仍然无法达到毒瘤出题人的要求，我们需要在此算法上进一步优化。

考虑该题与其它使用莫队的题的差异性，由于需要在维护值域的数据结构上查询，故单次端点的移动是 $O(\log n)$ 而非 $O(1)$。

众所周知，莫队是一种离线算法，它通过将询问离线处理的方式来优化复杂度。我们在将原问题的查询离线的基础上，尝试将端点移动时在数据结构上进行的修改和查询操作离线下来统一处理，最后用 $O(n\sqrt{n} + n\log n)$ 的时间复杂度解决问题。由于前后进行了两次离线操作，故称为「莫队二次离线」。

例题 2

"Luogu P5501 [LnOI2019] **来者不拒，去者不追**[2]"

给定一个长度为 n 的序列 a。给定 m 个询问，每次询问一个区间中 $[l, r]$ 中所有数的「Abbi 值」之和。

Abbi 值定义为：若 a_i 在询问区间 $[l, r]$ 中是第 k 小，那么它的「Abbi 值」等于 $k \times a_i$。

数据范围：$1 \le a_i \le 100000$，$1 \le l \le r \le n$，$1 \le n, m \le 500000$。

" **示例代码**"

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

typedef long long lxl;

const int maxN = 5e5;
```

```
const int maxM = 5e5;
const int maxA = 1e5;
const int sqrN = 708;
const int sqrA = 317;

int n, m;
int a[maxN + 10];
int b[maxN + 10];
int l, r;
lxl f[maxN + 10];
lxl g[maxN + 10];
lxl ans[maxM + 10];

typedef struct SegmentTree {
  struct Node {
    lxl val;
    lxl tag;
  } node[4 * maxA + 10];

  void MakeTag(int u, int l, int r, lxl val) {
    node[u].val += val * (r - l + 1);
    node[u].tag += val;
    return;
  }

  void PushDown(int u, int l, int r) {
    if (!node[u].tag) return;
    int mid = (l + r) / 2;
    MakeTag(2 * u, l, mid, node[u].tag);
    MakeTag(2 * u + 1, mid + 1, r, node[u].tag);
    node[u].tag = 0;
    return;
  }

  void PushUp(int u) {
    node[u].val = node[2 * u].val + node[2 * u + 1].val;
    return;
  }

  void Add(int u, int l, int r, int s, int t, lxl val) {
    if (s > t) return;
    if (s <= l && r <= t) {
      MakeTag(u, l, r, val);
      return;
    }
    PushDown(u, l, r);
    int mid = (l + r) / 2;
    if (s <= mid) Add(2 * u, l, mid, s, t, val);
    if (t >= mid + 1) Add(2 * u + 1, mid + 1, r, s, t, val);
    PushUp(u);
    return;
  }

  void Add(int u, int l, int r, int pos, lxl val) {
```

```
    Add(u, l, r, pos, pos, val);
    return;
  }

  lxl Ask(int u, int l, int r, int s, int t) {
    if (s > t) return 0;
    if (s <= l && r <= t) {
      return node[u].val;
    }
    PushDown(u, l, r);
    int mid = (l + r) / 2;
    if (t <= mid) return Ask(2 * u, l, mid, s, t);
    if (s >= mid + 1) return Ask(2 * u + 1, mid + 1, r, s, t);
    return Ask(2 * u, l, mid, s, t) + Ask(2 * u + 1, mid + 1, r, s, t);
  }
} sgt;

typedef struct BlockArray {
  struct Block {
    int l, r;
    lxl tag;
  } block[sqrA + 10];

  struct Array {
    int bel;
    lxl val;
  } array[maxA + 10];

  void Build() {
    for (int i = 1; i <= maxA; i++) array[i].bel = (i - 1) / sqrA + 1;
    for (int i = 1; i <= maxA; i++) block[array[i].bel].r = i;
    for (int i = maxA; i >= 1; i--) block[array[i].bel].l = i;
    return;
  }

  void Add(int pos, lxl val) {
    for (int i = array[pos].bel + 1; i <= array[maxA].bel; i++)
      block[i].tag += val;
    for (int i = pos; i <= block[array[pos].bel].r; i++) array[i].val += val;
    return;
  }

  lxl Ask(int pos) { return array[pos].val + block[array[pos].bel].tag; }

  lxl Ask(int l, int r) {
    if (l > r) return 0;
    return Ask(r) - Ask(l - 1);
  }
} dba;

namespace captainMoSecondaryOffline {
namespace offline2 {
struct Query {
  int i;
```

```
  int l, r;
  int k;
};

std::vector<Query> query[maxN + 10];

dba sum, cnt;

void solve() {
  sum.Build();
  cnt.Build();
  for (int i = 1; i <= n; i++) {
    sum.Add(a[i], a[i]);
    cnt.Add(a[i], 1);
    for (int j = 0; j < query[i].size(); j++) {
      for (int k = query[i][j].l; k <= query[i][j].r; k++) {
        ans[query[i][j].i] +=
            1ll * query[i][j].k *
            (sum.Ask(a[k] + 1, maxA) + cnt.Ask(1, a[k] - 1) * a[k]);
      }
    }
  }
  return;
}
}  // namespace offline2

namespace offline1 {
struct Query {
  int i;
  int l, r;

  bool operator<(const Query &other) const {
    if (b[l] != b[other.l]) return l < other.l;
    return r < other.r;
  }
};

std::vector<Query> query;

sgt sum, cnt;

void solve() {
  std::sort(query.begin(), query.end());
  for (int i = 1; i <= n; i++) {
    f[i] = sum.Ask(1, 1, maxA, a[i] + 1, maxA);
    g[i] = cnt.Ask(1, 1, maxA, 1, a[i] - 1);
    sum.Add(1, 1, maxA, a[i], a[i]);
    cnt.Add(1, 1, maxA, a[i], 1);
  }
  for (int i = 0, l = 1, r = 0; i < query.size(); i++) {
    if (l > query[i].l) {
      offline2::query[r].push_back(
          (offline2::Query){query[i].i, query[i].l, l - 1, 1});
      while (l > query[i].l) {
```

```
        l--;
        ans[query[i].i] -= f[l] + (g[l] - 1) * a[l];
      }
    }
    if (r < query[i].r) {
      offline2::query[l - 1].push_back(
          (offline2::Query){query[i].i, r + 1, query[i].r, -1});
      while (r < query[i].r) {
        r++;
        ans[query[i].i] += f[r] + (g[r] + 1) * a[r];
      }
    }
    if (l < query[i].l) {
      offline2::query[r].push_back(
          (offline2::Query){query[i].i, l, query[i].l - 1, -1});
      while (l < query[i].l) {
        ans[query[i].i] += f[l] + (g[l] - 1) * a[l];
        l++;
      }
    }
    if (r > query[i].r) {
      offline2::query[l - 1].push_back(
          (offline2::Query){query[i].i, query[i].r + 1, r, 1});
      while (r > query[i].r) {
        ans[query[i].i] -= f[r] + (g[r] + 1) * a[r];
        r--;
      }
    }
  }
  return;
}
}  // namespace offline1

void solve() {
  offline1::solve();
  offline2::solve();
  for (int i = 0; i < m; i++)
    ans[offline1::query[i].i] += ans[offline1::query[i - 1].i];
  return;
}
}  // namespace captainMoSecondaryOffline

int main() {
  std::cin >> n >> m;
  for (int i = 1; i <= n; i++) std::cin >> a[i];
  for (int i = 1; i <= n; i++) b[i] = (i - 1) / sqrN + 1;
  for (int i = 1; i <= m; i++)
    std::cin >> l >> r,
        captainMoSecondaryOffline::offline1::query.push_back(
            (captainMoSecondaryOffline::offline1::Query){i, l, r});
  captainMoSecondaryOffline::solve();
  for (int i = 1; i <= m; i++) std::cout << ans[i] << '\n';
  return 0;
}
```

参考资料与注释

[1] Luogu P5047 [Ynoi2019 模拟赛] Yuno loves sqrt technology II

[2] Luogu P5501 [LnOI2019] 来者不拒，去者不追

莫队配合 bitset

Authors: StudyingFather, Backl1ght, countercurrent-time, Ir1d, greyqz, MicDZ, ouuan

bitset 常用于常规数据结构难以维护的的判定、统计问题，而莫队可以维护常规数据结构难以维护的区间信息。把两者结合起来使用可以同时利用两者的优势。

例题 「Ynoi2016」 掉进兔子洞[1]

本题刚好符合上面提到的莫队配合 bitset 的特征。不难想到我们可以分别用 bitset 存储每一个区间内的出现过的所有权值，一组询问的答案即所有区间的长度和减去三者的并集元素个数 $\times 3$。

但是在莫队中使用 bitset 也需要针对 bitset 的特性调整算法：

1. bitset 不能很好地处理同时出现多个权值的情况。我们可以把当前元素离散化后的权值与当前区间的的出现次数之和作为往 bitset 中插入的对象。
2. 我们平常使用莫队时，可能会不注意 4 种移动指针的方法顺序，所以指针移动的过程中可能会出现区间的左端点在右端点右边，区间长度为负值的情况，导致元素的个数为负数。这在其他情况下并没有什么影响，但是本题中在 bitset 中插入的元素与元素个数有关，所以我们需要注意 4 种移动指针的方法顺序，将左右指针分别往左边和右边移动的语句写在前面，避免往 bitset 中插入负数。
3. 虽然 bitset 用空间小，但是仍然难以承受 $10^5 \times 10^5$ 的数据规模。所以我们需要将询问划分成常数块分别处理，保证空间刚好足够的情况下时间复杂度不变。

参考代码

```
#include <algorithm>
#include <bitset>
#include <cmath>
#include <cstdio>
#include <cstring>
using namespace std;
const int N = 100005, M = N / 3 + 10;
int n, m, maxn;
int a[N], ans[M], cnt[N];
bitset<N> sum[M], now;

struct query {
  int l, r, id;

  bool operator<(const query& x) const {
    if (l / maxn != x.l / maxn) return l < x.l;
    return (l / maxn) & 1 ? r < x.r : r > x.r;
  }
} q[M * 3];

void static_set() {
  static int tmp[N];
```

```
  memcpy(tmp, a, sizeof(a));
  sort(tmp + 1, tmp + n + 1);
  for (int i = 1; i <= n; i++)
    a[i] = lower_bound(tmp + 1, tmp + n + 1, a[i]) - tmp;
}

void add(int x) {
  now.set(x + cnt[x]);
  cnt[x]++;
}

void del(int x) {
  cnt[x]--;
  now.reset(x + cnt[x]);
}

void solve() {
  int cnt = 0, tot = 0;
  now.reset();
  for (tot = 0; tot < M - 5 && m; tot++) {
    m--;
    ans[tot] = 0;
    sum[tot].set();
    for (int j = 0; j < 3; j++) {
      scanf("%d%d", &q[cnt].l, &q[cnt].r);
      q[cnt].id = tot;
      ans[tot] += q[cnt].r - q[cnt].l + 1;
      cnt++;
    }
  }
  sort(q, q + cnt);
  for (int i = 0, l = 1, r = 0; i < cnt; i++) {
    while (l > q[i].l) add(a[--l]);
    while (r < q[i].r) add(a[++r]);
    while (l < q[i].l) del(a[l++]);
    while (r > q[i].r) del(a[r--]);
    sum[q[i].id] &= now;
  }
  for (int i = 0; i < tot; i++)
    printf("%d\n", ans[i] - (int)sum[i].count() * 3);
}

int main() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= n; i++) scanf("%d", &a[i]);
  static_set();
  maxn = sqrt(n);
  solve();
  memset(cnt, 0, sizeof(cnt));
  solve();
  memset(cnt, 0, sizeof(cnt));
  solve();
  return 0;
}
```

习题

- 小清新人渣的本愿[2]
- 「Ynoi2017」 由乃的玉米田[3]
- 「Ynoi2011」WBLT[4]

参考资料与注释

[1] 「Ynoi2016」掉进兔子洞

[2] 小清新人渣的本愿

[3] 「Ynoi2017」由乃的玉米田

[4] 「Ynoi2011」WBLT

13.5 分数规划

Authors: greyqz, Ir1d, hsfzLZH1, huaruoji

分数规划用来求一个分式的极值。

形象一点就是，给出 a_i 和 b_i，求一组 $w_i \in \{0,1\}$，最小化或最大化

$$\frac{\sum\limits_{i=1}^{n} a_i \times w_i}{\sum\limits_{i=1}^{n} b_i \times w_i}$$

另外一种描述：每种物品有两个权值 a 和 b，选出若干个物品使得 $\frac{\sum a}{\sum b}$ 最小/最大。

一般分数规划问题还会有一些奇怪的限制，比如『分母至少为 W』。

求解

二分法

分数规划问题的通用方法是二分。

假设我们要求最大值。二分一个答案 mid，然后推式子（为了方便少写了上下界）：

$$\begin{aligned}
& \frac{\sum a_i \times w_i}{\sum b_i \times w_i} > mid \\
\Longrightarrow & \sum a_i \times w_i - mid \times \sum b_i \cdot w_i > 0 \\
\Longrightarrow & \sum w_i \times (a_i - mid \times b_i) > 0
\end{aligned}$$

那么只要求出不等号左边的式子的最大值就行了。如果最大值比 0 要大，说明 mid 是可行的，否则不可行。

求最小值的方法和求最大值的方法类似，读者不妨尝试着自己推一下。

Dinkelbach 算法

Dinkelbach 算法的大概思想是每次用上一轮的答案当做新的 L 来输入，不断地迭代，直至答案收敛。

分数规划的主要难点就在于如何求 $\sum w_i \times (a_i - mid \times b_i)$ 的最大值/最小值。下面通过一系列实例来讲解该式子的最大值/最小值的求法。

实例

模板

> 有 n 个物品，每个物品有两个权值 a 和 b。求一组 $w_i \in \{0, 1\}$，最大化 $\dfrac{\sum a_i \times w_i}{\sum b_i \times w_i}$ 的值。

把 $a_i - mid \times b_i$ 作为第 i 个物品的权值，贪心地选所有权值大于 0 的物品即可得到最大值。

为了方便初学者理解，这里放上完整代码：

参考代码

```cpp
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <iostream>
using namespace std;

int read() {
  int X = 0, w = 1;
  char c = getchar();
  while (c < '0' || c > '9') {
    if (c == '-') w = -1;
    c = getchar();
  }
  while (c >= '0' && c <= '9') X = X * 10 + c - '0', c = getchar();
  return X * w;
}

const int N = 100000 + 10;
const double eps = 1e-6;

int n;
double a[N], b[N];

bool check(double mid) {
  double s = 0;
  for (int i = 1; i <= n; ++i)
    if (a[i] - mid * b[i] > 0)  // 如果权值大于 0
      s += a[i] - mid * b[i];   // 选这个物品
  return s > 0;
}

int main() {
  // 输入
  n = read();
  for (int i = 1; i <= n; ++i) a[i] = read();
```

```
  for (int i = 1; i <= n; ++i) b[i] = read();
  // 二分
  double L = 0, R = 1e9;
  while (R - L > eps) {
    double mid = (L + R) / 2;
    if (check(mid))  // mid 可行，答案比 mid 大
      L = mid;
    else  // mid 不可行，答案比 mid 小
      R = mid;
  }
  // 输出
  printf("%.6lf\n", L);
  return 0;
}
```

为了节省篇幅，下面的代码只保留 check 部分。主程序和本题是类似的。

POJ2976 Dropping tests[1]

> 有 n 个物品，每个物品有两个权值 a 和 b。
>
> 你可以选 $n-k$ 个物品 $p_1,p_2,\cdots,p_{n-k}$，使得 $\dfrac{\sum a_{p_i}}{\sum b_{p_i}}$ 最大。
>
> 输出答案乘 100 后四舍五入到整数的值。

把第 i 个物品的权值设为 $a_i - mid\times b_i$，然后选最大的 $n-k$ 个即可得到最大值。

```
bool cmp(double x, double y) { return x > y; }

bool check(double mid) {
  int s = 0;
  for (int i = 1; i <= n; ++i) c[i] = a[i] - mid * b[i];
  sort(c + 1, c + n + 1, cmp);
  for (int i = 1; i <= n - k; ++i) s += c[i];
  return s > 0;
}
```

洛谷 4377 Talent Show[2]

> 有 n 个物品，每个物品有两个权值 a 和 b。
>
> 你需要确定一组 $w_i\in\{0,1\}$，使得 $\dfrac{\sum w_i\times a_i}{\sum w_i\times b_i}$ 最大。
>
> 要求 $\sum w_i\times b_i\geq W$。

本题多了分母至少为 W 的限制，因此无法再使用上一题的贪心算法。

可以考虑 01 背包。把 b_i 作为第 i 个物品的重量，$a_i - mid\times b_i$ 作为第 i 个物品的价值，然后问题就转化为背包了。

那么 $dp[n][W]$ 就是最大值。

一个要注意的地方：$\sum w_i\times b_i$ 可能超过 W，此时直接视为 W 即可。（想一想，为什么？）

```
double f[1010];

bool check(double mid) {
  for (int i = 1; i <= W; i++) f[i] = -1e9;
  for (int i = 1; i <= n; i++)
    for (int j = W; j >= 0; j--) {
      int k = min(W, j + b[i]);
      f[k] = max(f[k], f[j] + a[i] - mid * b[i]);
    }
  return f[W] > 0;
}
```

POJ2728 Desert King[3]

> 每条边有两个权值 a_i 和 b_i，求一棵生成树 T 使得 $\frac{\sum_{e\in T} a_e}{\sum_{e\in T} b_e}$ 最小。

把 $a_i - mid \times b_i$ 作为每条边的权值，那么最小生成树就是最小值，
代码就是求最小生成树，故省略。

[HNOI2009] 最小圈[4]

> 每条边的边权为 w，求一个环 C 使得 $\frac{\sum_{e\in C} w}{|C|}$ 最小。

把 $a_i - mid$ 作为边权，那么权值最小的环就是最小值。
因为我们只需要判最小值是否小于 0，所以只需要判断图中是否存在负环即可。
另外本题存在一种复杂度 $O(nm)$ 的算法，如果有兴趣可以阅读 这篇文章[5]。

```
int SPFA(int u, double mid) {  // 判负环
  vis[u] = 1;
  for (int i = head[u]; i; i = e[i].nxt) {
    int v = e[i].v;
    double w = e[i].w - mid;
    if (dis[u] + w < dis[v]) {
      dis[v] = dis[u] + w;
      if (vis[v] || SPFA(v, mid)) return 1;
    }
  }
  vis[u] = 0;
  return 0;
}

bool check(double mid) {  // 如果有负环返回 true
  for (int i = 1; i <= n; ++i) dis[i] = 0, vis[i] = 0;
  for (int i = 1; i <= n; ++i)
    if (SPFA(i, mid)) return 1;
  return 0;
}
```

总结

分数规划问题是一类既套路又灵活的题目，一般使用二分解决。

分数规划问题的主要难点在于推出式子后想办法求出 $\sum w_i \times (a_i - mid \times b_i)$ 的最大值/最小值，而这个需要具体情况具体分析。

习题

- JSOI2016 最佳团体[6]
- SDOI2017 新生舞会[7]
- UVa1389 Hard Life[8]

参考资料与注释

[1] POJ2976 Dropping tests

[2] 洛谷 4377 Talent Show

[3] POJ2728 Desert King

[4] [HNOI2009] 最小圈

[5] 这篇文章

[6] JSOI2016 最佳团体

[7] SDOI2017 新生舞会

[8] UVa1389 Hard Life

13.6 随机化

13.6.1 随机函数

概述

要想使用随机化技巧，前提条件是能够快速生成随机数。本文将介绍生成随机数的常见方法。

随机数与伪随机数

说一个单独的数是「随机数」是无意义的，所以以下我们都默认讨论「随机数列」，即使提到「随机数」，指的也是「随机数列中的一个元素」。

现有的计算机的运算过程都是确定性的，因此，仅凭借算法来生成真正**不可预测、不可重复**的随机数列是不可能的。

然而在绝大部分情况下，我们都不需要如此强的随机性，而只需要所生成的数列在统计学上具有随机数列的种种特征（比如均匀分布、互相独立等等）。这样的数列即称为**伪随机数**序列。

随机数与伪随机数在实际生活和算法中的应用举例：

- 抽样调查时往往只需使用伪随机数。这是因为我们本就只关心统计特征。
- 网络安全中往往要用到（比刚刚提到的伪随机数）更强的随机数。这是因为攻击者可能会利用可预测性做文章。
- OI/ICPC 中用到的随机算法，基本都只需要伪随机数。这是因为，这些算法往往是通过引入随机数来把概率引入复杂度分析，从而降低复杂度。这本质上依然只利用了随机数的统计特征。
- 某些随机算法（例如 Moser 算法[5]）用到了随机数的熵相关的性质，因此必须使用真正的随机数。

实现

rand

用于生成伪随机数，缺点是比较慢，使用时需要 `#include<cstdlib>`。

调用 `rand()` 函数会返回一个 `[0,RAND_MAX]` 中的随机非负整数，其中 `RAND_MAX` 是标准库中的一个宏，在 Linux 系统下 `RAND_MAX` 等于 $2^{31}-1$。可以用取模来限制所生成的数的大小。

使用 `rand()` 需要一个随机数种子，可以使用 `srand(seed)` 函数来将随机种子更改为 `seed`，当然不初始化也是可以的。

同一程序使用相同的 `seed` 两次运行，在同一机器、同一编译器下，随机出的结果将会是相同的。

有一个选择是使用当前系统时间来作为随机种子：`srand(time(0))`。

> warning
>
> 在 `Windows` 系统下 `rand()` 返回值的取值范围为 $[0,2^{15})$（即 `RAND_MAX` 等于 $2^{15}-1$），当需要生成的数不小于 2^{15} 时建议使用 `(rand() << 15 | rand())` 来生成更大的随机数。

关于 `rand()` 和 `rand()%n` 的随机性：

- C/C++ 标准并未关于 `rand()` 所生成随机数的任何方面的质量做任何规定。
- GCC 编译器对 `rand()` 所采用的实现方式，保证了分布的均匀性等基本性质，但具有低位周期长度短等明显缺陷。（例如在笔者的机器上，`rand()%2` 所生成的序列的周期长约 $2\cdot10^6$）
- 即使假设 `rand()` 是均匀随机的，`rand()%n` 也不能保证均匀性，因为 `[0,n)` 中的每个数在 `0%n,1%n,...,RAND_MAX%n` 中的出现次数可能不相同。

预定义随机数生成器

定义了数个特别的流行算法。如没有特别说明，均定义于头文件 `<random>`。

> warning
>
> 预定义随机数生成器仅在于 C++11 标准[2] 中开始使用。

mt19937

是一个随机数生成器类，效用同 `rand()`，随机数的范围同 `unsigned int` 类型的取值范围。

其优点是随机数质量高（一个表现为，出现循环的周期更长；其他方面也都至少不逊于 `rand()`），且速度比 `rand()` 快很多。使用时需要 `#include<random>`。

`mt19937` 基于 32 位梅森缠绕器，由松本与西村设计于 1998 年[3]，使用时用其定义一个随机数生成器即可：`std::mt19937 myrand(seed)`，`seed` 可不填，不填 `seed` 则会使用默认随机种子。

`mt19937` 重载了 `operator ()`，需要生成随机数时调用 `myrand()` 即可返回一个随机数。

另一个类似的生成器是 `mt19937_64`，基于 64 位梅森缠绕器，由松本与西村设计于 2000 年，使用方式同 `mt19937`，但随机数范围扩大到了 `unsigned long long` 类型的取值范围。

> " 代码示例"

```
#include <ctime>
#include <iostream>
#include <random>

using namespace std;

int main() {
  mt19937 myrand(time(0));
  cout << myrand() << endl;
  return 0;
}
```

minstd_rand0

线性同余算法由 Lewis、Goodman 及 Miller 发现于 1969，由 Park 与 Miller 于 1988 采纳为「最小标准」。

计算公式如下，其中 A, C, M 为预定义常数。

$$s_i \equiv s_{i-1} \times A + C \mod M$$

minstd_rand() 是较新的「最小标准」，为 Park、Miller 和 Stockmeyer 于 1993 推荐。

对于 minstd_rand0()，s 的类型取 32 位无符号整数，A 取 16807，C 取 0，M 取 2147483647。

对于 minstd_rand()，s 的类型取 32 位无符号整数，A 取 48271，C 取 0，M 取 2147483647。

random_shuffle

用于随机打乱指定序列。使用时需要 #include<algorithm>。

使用时传入指定区间的首尾指针或迭代器（左闭右开）即可：std::random_shuffle(first, last) 或 std::random_shuffle(first, last, myrand)

内部使用的随机数生成器默认为 rand()。当然也可以传入自定义的随机数生成器。

关于 random_shuffle 的随机性：

- C++ 标准中要求 random_shuffle 在所有可能的排列中**等概率**随机选取，但 GCC[4-1] 编译器**并未**严格执行。
- GCC 中 random_shuffle 随机性上的缺陷的原因之一，是因为它使用了 rand()%n 这样的写法。如先前所述，这样生成的不是均匀随机的整数。
- 原因之二，是因为 rand() 的值域有限。如果所传入的区间长度超过 RAND_MAX，将存在某些排列**不可能**被产生[1]。

warning

random_shuffle 已于 C++14 标准中被弃用，于 C++17 标准中被移除。

shuffle

效用同 random_shuffle。使用时需要 #include<algorithm>。

区别在于必须使用自定义的随机数生成器：std::shuffle(first, last, myrand)。

GCC[4-2] 实现的 shuffle 符合 C++ 标准的要求，即在所有可能的排列中等概率随机选取。

下面是用 rand() 及 random_shuffle() 编写的一个数据生成器。生成数据为 「ZJOI2012」 灾难[6] 的随机小数据。

```
#include <algorithm>
#include <cstdlib>
#include <ctime>
```

```
#include <iostream>

int a[100];

int main() {
  srand(time(0));
  int n = rand() % 99 + 1;
  for (int i = 1; i <= n; i++) a[i] = i;
  std::cout << n << '\n';
  for (int i = 1; i <= n; i++) {
    std::random_shuffle(a + 1, a + i);
    int cnt = rand() % i;
    for (int j = 1; j <= cnt; j++) std::cout << a[j] << ' ';
    std::cout << 0 << '\n';
  }
}
```

下面是用 `mt19937` 及 `shuffle()` 编写的同一个数据生成器。

```
#include <algorithm>
#include <ctime>
#include <iostream>
#include <random>

int a[100];

int main() {
  std::mt19937 rng(time(0));
  int n = rng() % 99 + 1;
  for (int i = 1; i <= n; i++) a[i] = i;
  std::cout << n << '\n';
  for (int i = 1; i <= n; i++) {
    std::shuffle(a + 1, a + i, rng);
    int cnt = rng() % i;
    for (int j = 1; j <= cnt; j++) std::cout << a[j] << ' ';
    std::cout << 0 << '\n';
  }
}
```

下面是随机排列前十个正整数的一个实现。

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <random>

int main() {
  std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  std::random_device rd;
  std::mt19937 g(rd());

  std::shuffle(v.begin(), v.end(), g);

  std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " "));
  std::cout << "\n";
```

```
}
```

非确定随机数的均匀分布整数随机数生成器

random_device 是一个基于硬件的均匀分布随机数生成器，**在熵池耗尽**前可以高速生成随机数。该类在 C++11 定义，需要 random 头文件。由于熵池耗尽后性能急剧下降，所以建议用此方法生成 mt19937 等伪随机数的种子，而不是直接生成。

random_device 是非确定的均匀随机位生成器，尽管若不支持非确定随机数生成，则允许实现用伪随机数引擎实现。目前笔者尚未接到报告称 NOIP 评测机不支持基于硬件的均匀分布随机数生成。但出于保守考虑，建议使用该算法生成随机数种子。

参考代码如下。

```cpp
#include <iostream>
#include <map>
#include <random>
#include <string>

int main() {
  std::random_device rd;
  std::map<int, int> hist;
  std::uniform_int_distribution<int> dist(0, 9);
  for (int n = 0; n < 20000; ++n) {
    ++hist[dist(rd)];  // 注意：仅用于演示：一旦熵池耗尽，
                       // 许多 random_device 实现的性能就急剧下滑
                       // 对于实践使用， random_device 通常仅用于
                       // 播种类似 mt19937 的伪随机数生成器
  }
  for (auto p : hist) {
    std::cout << p.first << " : " << std::string(p.second / 100, '*') << '\n';
  }
}
```

可能的输出如下。

```
0 : ********************
1 : *******************
2 : ********************
3 : ********************
4 : ********************
5 : *******************
6 : ********************
7 : ********************
8 : *******************
9 : ********************
```

随机数分布

这里介绍的是要求生成的随机数按照一定的概率出现，如等概率，伯努利分布[7]，二项分布[8]，几何分布[9]，标准正态 （ 高斯 ） 分布[10]。

具体类名请参见 伪随机数生成 —— 随机数分布[11] 的列表。

实现

下面的程序模拟了一个六面体骰子。

```cpp
#include <iostream>
#include <random>

int main() {
  std::random_device rd;   // 将用于为随机数引擎获得种子
  std::mt19937 gen(rd());  // 以播种标准 mersenne_twister_engine
  std::uniform_int_distribution<> dis(1, 6);

  for (int n = 0; n < 10; ++n)
    // 用 dis 变换 gen 所生成的随机 unsigned int 到 [1, 6] 中的 int
    std::cout << dis(gen) << ' ';
  std::cout << '\n';
}
```

其他实现方法

有的时候我们需要实现自己的随机数生成器。下面是一些常用的随机数生成方法。

线性同余随机数生成器

利用下式来生成随机数序列 $\{R_i\}$：

$$R_{i+1} = (A \times R_i + B) \bmod P$$

其中 A, B, P 均为常数。

该方法实现难度低，但生成的随机序列周期长度较短（周期最大为 P，但大多数情况下都会比 P 短）。

" 参考实现"

```cpp
#include <iostream>
using namespace std;

struct myrand {
  int A, B, P, x;

  myrand(int A, int B, int P) {
    this->A = A;
    this->B = B;
    this->P = P;
  }

  int next() { return x = (A * x + B) % P; }  // 生成随机序列的下一个随机数
};

myrand rnd(3, 5, 97);  // 初始化一个随机数生成器

int main() {
  int x = rnd.next();
  cout << x << endl;
  return 0;
}
```

时滞斐波那契随机数生成器

利用下式来生成随机数序列 $\{R_i\}$（其中 $0 < j < k$）：

$$R_i \equiv R_{i-j} \star R_{i-k} \bmod P$$

这里的 P 通常取 2 的幂（常用 2^{32} 或 2^{64}），$\star$ 表示二元运算符，可以使用加法，减法，乘法，异或。该方法较传统的线性同余随机数生成器而言，拥有更长的周期，但随机性受初始条件影响较大。

" 参考实现"

```
#include <iostream>
#include <vector>
using namespace std;

struct myrand {
  vector<unsigned> vec;
  int l, j, k, cur;

  myrand(int l, int j, int k) {
    this->l = l;
    this->j = j;
    this->k = k;
    cur = 0;
    for (int i = 0; i < l; i++) {
      vec.push_back(rand());  // 先用其他方法生成随机序列中的前几个元素
    }
  }

  unsigned next() {
    vec[cur] = vec[(cur - j + l) % l] * vec[(cur - k + l) % l];
    // 这里用 unsigned 类型是为了实现自动对 2^32 取模
    return vec[cur++];
  }
};

myrand rnd(11, 4, 7);

int main() {
  unsigned x = rnd.next();
  cout << x << endl;
  return 0;
}
```

参考资料与注释

[1] Don't use rand(): a guide to random number generators in C++

[2] 伪随机数生成 - cppreference.com

[3] Mersenne Twister algorithm

[4] 版本号为 GCC 9.2.0 [4-1] [4-2]

[5] Moser 算法

[6] 「ZJOI2012」灾难

[7] 伯努利分布

[8] 二项分布

[9] 几何分布

[10] 标准正态（高斯）分布

[11] 伪随机数生成——随机数分布

13.6.2 随机化技巧

Authors: Ir1d, partychicken, ouuan, Marcythm, TianyiQ

概述

前置知识：随机函数 和 概率初步

本文将对 OI/ICPC 中的随机化相关技巧做一个简单的分类，并对每个分类予以介绍。本文也将介绍一些在 OI/ICPC 中很少使用，但与 OI/ICPC 在风格等方面较为贴近的方法，这些内容前将用（*）标注。

这一分类并不代表广泛共识，也必定不能囊括所有可能性，因此仅供参考。

记号和约定：

- $\Pr[A]$ 表示事件 A 发生的概率。
- $\mathrm{E}[X]$ 表示随机变量 X 的期望。
- 赋值号 $:=$ 表示引入新的量，例如 $Y := 1926$ 表示引入值为 1926 的量 Y。

用随机集合覆盖目标元素

庞大的解空间中有一个（或多个）解是我们想要的。我们可以尝试进行多次撒网，只要有一次能够网住目标解就能成功。

例：三部图的判定

"问题"

给定一张 n 个结点、m 条边的简单无向图，用 RGB 三种颜色给每个结点染色满足任意一对邻居都不同色，或者报告无解。

对每个点 v，从 $\{R,G,B\}$ 中等概率独立随机地选一种颜色 C_v，并钦定 v **不**被染成 C_v。最优解恰好符合这些限制的概率，显然是 $\left(\frac{2}{3}\right)^n$。

在这些限制下，对于一对邻居 (u,v)，「u,v 不同色」的要求等价于以下这条「推出」关系：

- 对于所有异于 C_u,C_v 的颜色 X，若 u 被染成 X，则 v 被染成 $\{R,G,B\}\ \ \{X,C_v\}$。

于是我们可以对每个 v 设置布尔变量 B_v，其取值表示 v 被染成两种剩余的颜色中的哪一种。借助 2-SAT 模型即可以 $O(n+m)$ 的复杂度解决这个问题。

这样做，单次的正确率是 $\left(\frac{2}{3}\right)^n$。将算法重复运行 $-\left(\frac{3}{2}\right)^n \log\epsilon$ 次，只要有一次得到解就输出，这样即可保证 $1-\epsilon$ 的正确率。（详见后文中「概率上界的分析」）

回顾：本题中「解空间」就是集合 $\{R, G, B\}^n$，我们每次通过随机施加限制来在一个缩小的范围内搜寻「目标解」——即合法的染色方案。

例：CodeChef SELEDGE[4]

" 简要题意"

给定一张点、边都有非负权值的无向图，找到一个大小 $\le K$ 的边集合 S，以最大化与 S 相连的点的权值和减去 S 的边权和。一个点的权值只被计算一次。

观察：如果选出的边中有三条边构成一条链，则删掉中间的那条一定不劣；如果选出的边中有若干条构成环，则删掉任何一条一定不劣。

推论：最优解选出的边集，一定构成若干个不相交的菊花图（即直径不超过 2 的树）。

推论：最优解选出的边集，一定构成一张二分图。

我们对每个点等概率独立随机地染上黑白两种颜色之一，并要求这一染色方案，恰好也是最优解所对应的二分图的黑白染色方案。

尝试计算最优解符合这一要求的概率：

- 考虑一张 n 个点的菊花图，显然它有 2 种染色方案，所以它被染对颜色的概率是 $\dfrac{2}{2^n} = 2^{1-n}$。
- 假设最优解中每个菊花的结点数分别为 $a_1, \cdots, a_l$，则一定有 $(a_1 - 1) + \cdots + (a_l - 1) \le K$，其中 K 表示最多能够选出的边数。
- 从而所有菊花都被染对颜色的概率是 $2^{1-a_1} \cdots 2^{1-a_l} \ge 2^{-K}$。

在上述要求下，尝试建立费用流模型计算最优答案：

- 建立二分图，白点在左侧并与 S 相连，黑点在右侧并与 T 相连。
  - 对于白点 v，从 S 向它连一条容量为 1、费用为 $-A_v$ 的边，和一条容量为 ∞、费用为 0 的边。
  - 对于黑点 v，从它向 T 连一条容量为 1、费用为 $-A_v$ 的边，和一条容量为 ∞、费用为 0 的边。
- 对于原图中的边 (u, v, B) 满足 u 为白色、v 为黑色，连一条从 u 到 v 的边，容量为 1，费用为 B。
- 在该图中限制流量不超过 K，则最小费用的相反数就是答案。

用 SPFA 费用流求解的话，复杂度是 $O(K^2(n+m))$，证明：

- 首先，显然 SPFA 的运行次数 $\le K$。
- 然后，在一次 SPFA 中，任何一个结点至多入队 $O(K)$ 次。这是因为：
  - 任意时刻有流量的边不会超过 $3K$ 条，否则就意味着在原图中选了超过 K 条边。
  - 对于任何一条长为 L 的增广路，其中至少有 $\dfrac{L}{2} - 2$ 条边是某条有流量的边的反向边，因为正向边都是从图的左侧指向右侧，只有这些反向边才会从右侧指向左侧。
  - 综合以上两条，得到任意一条增广路的长度不超过 $6K + 4$。
- 综上，复杂度是 $O(K^2(n+m))$。

和上一题类似，我们需要把整个过程重复 $-2^K \log \epsilon$ 次以得到 $1-\epsilon$ 的正确率。总复杂度 $O(2^K K^2 (n+m) \cdot -\log \epsilon)$。

用随机元素命中目标集合

我们需要确定一个集合中的任意一个元素，为此我们随机选取元素，以期能够恰好命中这一集合。

例：Gym 101550I[5]

" 简要题意"

有一张图形如：两条平行的链，加上连接两链的两条平行边。给定这张图上的若干条简单路径（每条路径表示一次通话），请你选择尽量少的边放置窃听器，以使得每条给定的路径上都有至少一个窃听器。

整张图可以拆分为一个环加上四条从环伸出去的链。对于这四条链中的任何一条（记作 C），考虑在这条链上如何放置窃听器，容易通过贪心算法得到满足以下条件的方案：

- 在拦截所有 C 内部进行的通话的前提下，用的窃听器数量最少。
- 在上一条的前提下，使得 C 上的窃听器离环的最短距离尽可能小。
  - 作这一要求的目的是尽可能地拦截恰有一个端点在 C 内部的通话。

接着考虑链与环相接处的共计 4 条边，我们暴力枚举这些边上有没有放窃听器。显然，如果想要拦截跨越链和环的通话，在这 4 条边上放窃听器一定是最优的。现在，我们可以把通话线路分为以下几种：

1. 完全在链上的通话线路。这些线路一定已经被拦截，故可以忽略。
2. 跨越链和环，且已经被拦截的通话线路。它们可以忽略。
3. 跨越链和环，且未被拦截的通话线路。我们可以直接截掉它在链上的部分（因为链上的窃听器放置方案已经固定了），只保留环上的部分。
4. 完全在环上的通话线路。

至此，问题转化成了环上的问题。

设最优解中在环上的边集 S 上放置了窃听器，如果我们已经确定了 S 中的任何一个元素 e，就可以：

- 先在 e 处断环为链。
- 然后从 e 开始贪心，不断找到下一个放置窃听器的边。注意到如果经过合适的预处理，贪心的每一步可以做到 $O(1)$ 的复杂度。
- 从而以 $O(|S|)$ 的复杂度解决问题。

我们考虑随机选取环上的一条边 e'，并钦定 $e' \in S$ 再执行上述过程，重复多次取最优。

分析单次复杂度：

- 观察：记 S' 表示所有选取了 e' 的方案中的最优解，则 $|S'| \le |S| + 1$。
- 从而单次复杂度 $O(|S'|) = O(|S|)$。

分析正确率：

- 显然单次正确率 $\dfrac{|S|}{n}$，其中 n 表示环长。
- 所以需要重复 $-\dfrac{n}{|S|}\log\epsilon$ 次以得到 $1-\epsilon$ 的正确率。

综上，该算法的复杂度 $O(|S| \cdot -\dfrac{n}{|S|}\log\epsilon) = O(-n\log\epsilon)$。

例：CSES 1685 New Flight Routes[6]

" 简要题意"

给定一张有向图，请你加最少的边使得该图强连通，需**输出方案**。

先对原图进行强连通缩点。我们的目标显然是使每个汇点能到达每个源点。

不难证明，我们一定只会从汇点到源点连边，因为任何其他的连边，都能对应上一条不弱于它的、从汇点到源点的连边。

我们的一个核心操作是，取汇点 t 和源点 s（它们不必在同一个弱连通分量里），连边 $t \to s$ 以**使得 s 和 t 都不再是汇点或源点**（记作目标 I）。理想情况下这种操作每次能减少一个汇点和一个源点，那我们不断操作直到只剩一个汇点或只剩一个源点，而这样的情形就很平凡了。由此，我们猜测答案是源点个数与汇点个数的较大值。

不难发现，上述操作能够达到目标 I 的充要条件是：t 拥有 s 以外的前驱、且 s 拥有 t 以外的后继。可以证明（等会会给出证明），对于任意一张有着至少两个源点和至少两个汇点的 DAG，都存在这样的 (s,t)；但存在性的结论无法帮助我们构造方案，还需做其他分析。

- 有了这个充要条件还难以直接得到算法，主要的原因是连边 $t \to s$ 后可能影响其他 (s',t') 二元组的合法性，这个比较难处理。

注意到我们关于源汇点间的关系知之甚少（甚至连快速查询一对 $s-t$ 间是否可达都需要 dfs + bitset 预处理，而时限并不允许这么做），这提示我们需要某种非常一般和强大的性质。

观察：不满足目标 I 的 (s,t) 至多有 $n+m-1$ 对，其中 n 表示源点个数，m 表示汇点个数。

- 理由：对于每一对这样的 (s,t)，若把它看成 s,t 间的一条边，则所有这些边构成的图形如若干条不相交的链，于是边数不超过点数减一。
- 作出这一观察的动机是，要想将存在性结论应用于算法，前置步骤往往是把定性的结果加强为定量的结果。

推论：等概率随机选取 (s,t)，满足前述要求的概率 $\geq \dfrac{(n-1)(m-1)}{nm}$。

- 注意到这个结论严格强于先前给出的存在性结论。

推论：等概率独立随机地连续选取 $\dfrac{\min(n,m)}{2}$ 对不含公共元素的 (s,t)，并对它们**依次**操作（即连边 $t \to s$），则这些操作全部满足目标 I 的概率 $\geq \dfrac{1}{4}$。

- 理由：

$$
\begin{aligned}
&\frac{(n-1)(m-1)}{nm} \cdot \frac{(n-2)(m-2)}{(n-1)(m-1)} \cdots \frac{(n-k)(m-k)}{(n-k+1)(m-k+1)} \\
=& \frac{(n-k)(m-k)}{nm} \\
\geq& \frac{1}{4}
\end{aligned}
$$

而连续选完 k 对 (s,t) 后判断它们是否全部满足目标 I 很简单，只要再跑一遍强连通缩点，判断一下 n,m 是否都减小了 k 即可。注意到若每次减少 $k=\dfrac{\min(n,m)}{2}$，则 $\min(n,m)$ 必在 $O(\log(n+m))$ 轮内变成 1，也就转化到了平凡的情况。

" 算法伪代码"

```
while(n>1 and m>1):
    randomly choose k=min(n,m)/2 pairs (s,t)
    add edge t->s for all these pairs
    if new_n>n-k or new_m>m-k:
        roll_back()
solve_trivial()
```

复杂度 $O((|V|+|E|)\log|V|)$。

回顾：我们需要确定任意一对能够实现目标 I 的二元组 (s,t)，为此我们随机选择 (s,t)。

用随机化获得随机数据的性质

如果一道题的数据随机生成，我们可能可以利用随机数据的性质解决它。而在有些情况下，即使数据并非随机生成，我们也可以通过随机化来给其赋予随机数据的某些特性，从而帮助解决问题。

例：随机增量法

随机生成的元素序列可能具有「前缀最优解变化次数期望下很小」等性质，而随机增量法就通过随机打乱输入的序列来获得这些性质。

详见 随机增量法。

例：TopCoder MagicMolecule[7] 随机化解法

" 简要题意"

给定一张 n 个点、带点权的无向图，在其中所有大小不小于 $\dfrac{2n}{3}$ 的团中，找到点权和最大的那个。

$n \leq 50$

不难想到折半搜索。把点集均匀分成左右两半 V_L, V_R（大小都为 $\dfrac{n}{2}$），计算数组 $f_{L,k}$ 表示点集 $L \subseteq V_L$ 中的所有 $\geq k$ 元团的最大权值和。接着我们枚举右半边的每个团 C_R，算出左半边有哪些点与 C_R 中的所有点相连（这个点集记作 N_L），并用 $f_{N_L, \frac{2}{3}n - |C_R|} + value(C_R)$ 更新答案。

- 注意到可以 $O(1)$ 转移每一个 $f_{L,k}$。具体地说，取 d 为 L 中的任意一个元素，然后分类讨论：
  - 假设最优解中 d 不在团中，则从 $f_{L \setminus \{d\},k}$ 转移而来。
  - 假设最优解中 d 在团中，则从 $f_{L \cap N(d),k} + value(d)$ 转移而来，其中 $N(d)$ 表示 d 的邻居集合。
  - 别忘了还要用 $f_{L,k+1}$ 来更新 $f_{L,k}$。

这个解法会超时。尝试优化：

- 平分点集时均匀随机地划分。这样的话，最优解的点集 C_{res} 以可观的概率也被恰好平分（即 $|C_{res} \cap V_L| = |C_{res} \cap V_R|$）。
  - 当然，$|C_{res}|$ 可能是奇数。简单起见，这里假设它是偶数；奇数的情况对解法没有本质改变。
  - 实验发现，随机尝试约 20 次就能以很大概率有至少一次满足该性质。也就是说，如果我们的算法依赖于「C_{res} 被平分」这一性质，则将算法重复执行 20 次取最优，同样也能保证以很大概率得到正确答案。
- 有了这一性质，我们就可以直接钦定左侧团 L、右侧团 C_R 的大小都 $\geq \dfrac{n}{3}$。这会对复杂度带来两处改进：
  - f 可以省掉记录大小的维度。
  - 因为只需考虑大小 $\geq \dfrac{n}{3}$ 的团，所以需要考虑的左侧团 L 和右侧团 C_R 的数量也大大减少至约 $1.8 \cdot 10^6$。
- 现在的瓶颈变成了求单侧的某一子集的权值和，因为这需要 $O(2^{|V_L|} + 2^{|V_R|})$ 的预处理。
  - 解决方案：在 V_L, V_R 内部再次折半；当查询一个子集的权值和时，将这个子集分成左右两半查询，再把答案相加。
- 这样即可通过本题。

回顾：一个随机的集合有着「在划分出的两半的数量差距不会太悬殊」这一性质，而我们通过随机划分获取了这个性质。

随机化用于哈希

例：UOJ #207 共价大爷游长沙[8]

" 简要题意"

维护一棵动态变化的树，和一个动态变化的结点二元组集合。你需要支持：

- 删边、加边。保证得到的还是一棵树。
- 加入/删除某个结点二元组。
- 给定一条边 e，判断是否对于集合中的每个结点二元组 (s,t)，e 都在 s,t 间的简单路径上。

对图中的每条边 e，我们定义集合 S_e 表示经过该边的关键路径（即题中的 (a,b)）集合。考虑对每条边动态维护集合 S_e 的哈希值，这样就能轻松判定 S_e 是否等于全集（即 e 是否是「必经之路」）。

哈希的方式是，对每个 (a,b) 赋予 2^{64} 以内的随机非负整数 $H_{(a,b)}$，然后一个集合的哈希值就是其中元素的 H 值的异或和。

这样的话，任何一个固定的集合的哈希值一定服从 $R := \{0,1,\cdots,2^{64}-1\}$ 上的均匀分布（换句话说，哈希值的取值范围为 R，且取每一个值的概率相等）。这是因为：

1. 单个 $H_{(a,b)}$ 显然服从均匀分布。
2. 两个独立且服从 R 上的均匀分布的随机变量的异或和，一定也服从 R 上的均匀分布。自证不难。

从而该算法的正确率是有保障的。

至于如何维护这个哈希值，使用 LCT 即可。

例：CodeChef PANIC[9] 及其错误率分析

本题的大致解法：

1. 可以证明[1] $S(N)$ 服从一个关于 N 的 $O(K)$ 阶线性递推式。
2. 用 BM 算法求出该递推式。
3. 借助递推式，用凯莱哈密顿定理计算出 $S(N)$。

这里仅关注第二部分，即如何求一个矩阵序列的递推式。所以我们只需考虑下述问题：

" 问题"

给定一个矩阵序列，该序列在模 $P := 998244353$ 意义下服从一个齐次线性递推式（递推式中的数乘和加法运算定义为矩阵的数乘和加法），求出最短递推式。

如果一系列矩阵服从一个递推式 F，那么它的每一位也一定服从 F。然而，如果对某一位求出最短递推式 F'，则 F' 可能会比 F 更短，从而产生问题。

解决方案：给矩阵的每一位 (i,j) 赋予一个 $< P$ 的随机权值 $x_{i,j}$，然后对于序列中每个矩阵计算其所有位的加权和模 P 的结果，再把每个矩阵算出的这个数连成一个数列，最后我们对所得数列运行 BM 算法。

错误率分析：

- 假设上述做法求得了不同于 F（且显然也不长于 F）的 l 阶递推式 F'。
- 因为矩阵序列不服从 F'，所以一定存在矩阵中的某个位置 (i,j)，满足该位置对应的数列 $S_{i,j}$ 在某个 N 处不服从 F'。也就是说：

$$S(N)_{i,j} - F'_1 S(N-1)_{i,j} - \cdots - F'_l S(N-l)_{i,j} \not\equiv 0 \pmod P$$

- 假设 (i,j) 是唯一的不服从的位置，则一定有：

$$T_{i,j} := \left(x_{i,j} \cdot (S(N)_{i,j} - F'_1 S(N-1)_{i,j} - \cdots - F'_l S(N-l)_{i,j}) \bmod P\right) = 0$$

- 显然这仅当 $x_{i,j} = 0$ 时才成立，概率 P^{-1}。
- 如果有多个不服从的位置呢？
  - 对每个这样的位置 (i,j)，易证 $T_{i,j}$ 服从 $R := \{0,1,\cdots,P-1\}$ 上的均匀分布。
  - 若干个互相独立的、服从 R 上的均匀分布的随机变量，它们在模意义下的和，依然服从 R 上的均匀分布。自证不难。
  - 从而这种情况下的错误率也是 P^{-1}。

例：UOJ #552 同构判定鸭[10] 及其错误率分析

" 简要题意"

给定两张边权为小写字母的有向图 G_0, G_1，你要对这两张图分别算出「所有路径对应的字符串构成的多重集」(可能是无穷集)，并判断这两个多重集是否相等。如果不相等，你要给出一个最短的串，满足它在两个多重集中的出现次数不相等。

令 $f_{K,i,j}$ 表示图 G_K 中从点 i 开始的所有长为 j 的路径，这些路径对应的所有字符串构成的多重集的哈希值。按照 j 升序考虑每个状态，转移时枚举 i 的出边并钦定该边为路径上的第一条边。

要判断是否存在长度 $= L$ 的坏串，只需把 $\{f_{0,*,L}\}$ 和 $\{f_{1,*,L}\}$ 各自「整合」起来再比较即可（通配符 `*` 这里表示每一个结点，例如 $\{f_{0,*,L}\}$ 表示全体 $f_{0,i,L}$ 构成的集合，其中 i 取遍所有结点）。官方题解[2] 中证明了最短坏串（如果存在的话）长度一定不超过 $n_1 + n_2$，所以这个解法的复杂度是可靠的。

接下来考虑具体的哈希方式。注意到常规的哈希方法——即把串 $a_1a_2\cdots a_k$ 映射到 $(a_1 + Pa_2 + P^2a_3 + \cdots + P^{k-1}a_k) \bmod Q$ 上、再把多重集的哈希值定为其中元素的哈希值之和模 Q——在这里是行不通的。一个反例是，集合 `{"ab","cd"}` 与集合 `{"cb","ad"}` 的哈希值是一样的，不论 P, Q 如何取值。

上述做法的问题在于，一个串的哈希值是一个和式，从而其中的每一项可以拆出来并重组。为避免这一问题，我们考虑把哈希值改为一个连乘式。此外，乘法交换律会使得不同的位不可区分，为避免这一点我们要为不同的位赋予不同的权值。

对每一个二元组 (c, j)（其中 c 为字符，j 为整数表示 c 在某个串中的第几位）我们都预先生成一个随机数 $x_{c,j}$。然后我们把串 $a_1a_2\cdots a_k$ 映射到 $x_{a_1,1}x_{a_2,2}\cdots x_{a_k,k} \bmod Q$ 上（其中 Q 为**随机选取**的质数)、再把多重集的哈希值定为其中元素的哈希值之和模 Q。接下来分析它的错误率。

"(*)Schwartz–Zippel 引理"

令 $f \in F[z_1, \cdots, z_k]$ 为域 F 上的 k 元 d 次非零多项式，令 S 为 F 的有限子集，则至多有 $d \cdot |S|^{k-1}$ 组 $(z_1, \cdots, z_k) \in S^k$ 满足 $f(z_1, \cdots, z_k) = 0$。

" 如果你不知道域是什么"

你只需记得这两样东西都是域：

1. 模质数的剩余系，以及其上的各种运算。
2. 实数集，以及其上的各种运算。

推论：若 $z_1, \cdots, z_k$ 都在 S 中等概率独立随机选取，则 $\Pr[f(z_1, \cdots, z_k) = 0] \le \dfrac{d}{|S|}$。

记 F 为模 Q 的剩余系所对应的域，则对于一个 $L \le n_1 + n_2$，$\sum_i f_{0,i,L}$ 和 $\sum_i f_{1,i,L}$ 就分别对应着一个 F 上关于变元集合 $\{x_{*,*}\}$ 的 L 次多元多项式，不妨将这两个多项式记为 P_0, P_1。

假如两个不同的字符串多重集的哈希值相同，则有两种可能：

1. $P_0 \equiv P_1 \pmod Q$，即 P_0, P_1 的每一项系数在模 Q 意义下都对应相等。
2. $P_0 \not\equiv P_1 \pmod Q, P_0(x_{*,*}) \equiv P_1(x_{*,*}) \pmod Q$，即 P_0, P_1 虽然不恒等，但我们选取的这一组 $\{x_{*,*}\}$ 恰好使得它们在此处的点值相等。

分析前者发生的概率：

- 观察：对于任意的 $A \ne B; A, B \le N$ 和随机选取的质数 $Q \le Q_{\max}$，一定有：

$$\Pr[A \equiv B \pmod Q] = O\left(\frac{\log N \log Q_{max}}{Q_{max}}\right)$$

- 这是因为：使 $A \equiv B$ 成立的 Q 一定满足 $Q \mid (A - B)$，这样的 Q 有 $\omega(A - B) \le \log_2 N$ 个；而由质数定理，$Q_{\max}$ 以内不同的质数又有 $\Theta\left(\dfrac{Q_{\max}}{\log Q_{\max}}\right)$ 个。将两者相除即可得到上式。
- 在上述观察中取 A, B（满足 $A \ne B$）为某一特定项在 P_0, P_1 中的系数（也就等于该项对应的串在 G_0, G_1 中的

出现次数)，则易见 $A, B \le (m_1+m_2)^L$，得到：

$$\Pr[A \equiv B \pmod Q] = O\Big(\frac{L\log(m_1+m_2)\log Q_{max}}{Q_{max}}\Big)$$

- 所以取 $Q_{\max} \approx 10^{12}$ 就绰绰有余。如果机器无法支持这么大的整数运算，可以用双哈希代替。

分析后者发生的概率：

- 在 Schwartz–Zippel 引理中：
  - 取域 F 为模 Q 的剩余系对应的域
  - 取 $f(x_{*,*}) = P_0(x_{*,*}) - P_1(x_{*,*})$ 为 L 次非零多项式
  - 取 $S = F$
- 得到：所求概率 $\le \dfrac{L}{Q}$。

注意到我们需要对每个 L 都能保证正确性，所以要想保证严谨的话还需用 Union Bound（见后文）说明一下。

实践上我们不必随机选取模数，因为——比如说——用自己的生日做模数的话，实际上已经相当于随机数了。

例：（*）子矩阵不同元素个数

”问题”

给定 $n \times m$ 的矩阵，q 次询问一个连续子矩阵中不同元素的个数，要求在线算法。

允许 ϵ 的相对误差和 δ 的错误率，换句话说，你要对至少 $(1-\delta)q$ 个询问给出离正确答案相对误差不超过 ϵ 的回答。

$n \cdot m \le 2 \cdot 10^5; q \le 10^6; \epsilon = 0.5, \delta = 0.2$

引理：令 $X_{1\cdots k}$ 为互相独立的随机变量，且取值在 $[0,1]$ 中均匀分布，则 $\mathrm{E}[\min_i X_i] = \dfrac{1}{k+1}$。

- 证明：考虑一个单位圆，其上分布着**相对位置**均匀随机的 $k+1$ 个点，分别在位置 $0, X_1, X_2, \cdots, X_k$ 处。那么 $\min_i X_i$ 就等于 $k+1$ 段空隙中特定的一段的长度。而因为这些空隙之间是「对称」的，所以其中任何一段特定空隙的期望长度都是 $\dfrac{1}{k+1}$。

我们取 k 为不同元素的个数，并借助上述引理来从 $\min_i X_i$ 反推得到 k。

考虑采用某个哈希函数，将矩阵中每个元素都均匀、独立地随机映射到 $[0,1]$ 中的实数上去，且相等的元素会映射到相等的实数。这样的话，一个子矩阵中的所有元素对应的那些实数，在去重后就恰好是先前的集合 $\{X_1, \cdots, X_k\}$ 的一个实例，其中 k 等于子矩阵中不同元素的个数。

于是我们得到了算法：

1. 给矩阵中元素赋 $[0,1]$ 中的哈希值。为保证随机性，哈希函数可以直接用 `map` 和随机数生成器实现，即每遇到一个新的未出现过的值就给它随机一个哈希值。
2. 回答询问时设法求出子矩阵中哈希值的最小值 M，并输出 $\dfrac{1}{M} - 1$。

然而，这个算法并不能令人满意。它的输出值的期望是 $\mathrm{E}\Big[\dfrac{1}{\min_i X_i} - 1\Big]$，但事实上这个值并不等于 $\dfrac{1}{\mathrm{E}[\min_i X_i]} - 1 = k$，而（可以证明）等于 ∞。

也就是说，我们不能直接把 $\min_i X_i$ 的单次取值放在分母上，而要先算得它的期望，再把期望值放在分母上。

怎么算期望值？多次随机取平均。

我们用 C 组不同的哈希函数分别执行前述过程，回答询问时计算出 C 个不同的 M 值，并算出其平均数 $\overline{M}$，然后输出 $(\overline{M})^{-1} - 1$。

实验发现取 $C \approx 80$ 即可满足要求。严格证明十分繁琐，在此略去。

最后，怎么求子矩阵最小值？用二维 S-T 表即可，预处理 $O(nm\log n\log m)$，回答询问 $O(1)$。

随机化在算法中的其他应用

随机化的其他作用还包括：

- 防止被造数据者用针对性数据卡掉。例如在搜索时随机打乱邻居的顺序。
- 保证算法过程中进行的「操作」具有（某种意义上的）均匀性。例如 模拟退火 算法。

在这些场景下，随机化常常（但并不总是）与乱搞、骗分等做法挂钩。

例：「TJOI2015」 线性代数[11]

本题的标准算法是网络流，但这里我们采取这样的乱搞做法：

- 每次随机一个位置，把这个位置取反，判断大小并更新答案。

" 代码"

```cpp
#include <algorithm>
#include <cstdlib>
#include <iostream>

int n;

int a[510], b[510], c[510][510], d[510];
int p[510], q[510];

int maxans = 0;

void check() {
  memset(d, 0, sizeof d);
  int nowans = 0;
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++) d[i] += a[j] * c[i][j];
  for (int i = 1; i <= n; i++) nowans += (d[i] - b[i]) * a[i];
  maxans = std::max(maxans, nowans);
}

int main() {
  srand(19260817);
  std::cin >> n;
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++) std::cin >> c[i][j];
  for (int i = 1; i <= n; i++) std::cin >> b[i];
  for (int i = 1; i <= n; i++) a[i] = 1;
  check();
  for (int T = 1000; T; T--) {
    int tmp = rand() % n + 1;
    a[tmp] ^= 1;
    check();
  }
  std::cout << maxans << '\n';
}
```

例：（*）随机堆[3]

可并堆最常用的写法应该是左偏树了，通过维护树高让树左偏来保证合并的复杂度。然而维护树高有点麻烦，我们希望尽量避开。

那么可以考虑使用随机堆，即不按照树高来交换儿子，而是随机交换。

"代码"

```
struct Node {
  int child[2];
  long long val;
} nd[100010];

int root[100010];

int merge(int u, int v) {
  if (!(u && v)) return u | v;
  int x = rand() & 1, p = nd[u].val > nd[v].val ? u : v;
  nd[p].child[x] = merge(nd[p].child[x], u + v - p);
  return p;
}

void pop(int &now) { now = merge(nd[now].child[0], nd[now].child[1]); }
```

随机堆对堆的形态没有任何硬性或软性的要求，合并操作的期望复杂度对任何两个堆（作为 `merge` 函数的参数）都成立。下证。

"期望复杂度的证明"

将证，对于任意的堆 A，从根节点开始每次随机选左或者右走下去（直到无路可走），路径长度（即路径上的结点数）的期望值 $h(A) \le \log_2(|A|+1)$。

- 注意到在前述过程中合并堆 A, B 的期望复杂度是 $O(h(A)+h(B))$ 的，所以上述结论可以保证随机堆的期望复杂度。

证明采用数学归纳。边界情况是 A 为空图，此时显然。下设 A 非空。

假设 A 的两个子树分别为 L, R，则：

$$
\begin{aligned}
h(A) &= 1 + \frac{h(L)+h(R)}{2} && (13.4)\\
&\le 1 + \frac{\log_2(|L|+1) + \log_2(|R|+1)}{2} && (13.5)\\
&= \log_2 2\sqrt{(|L|+1)(|R|+1)} && (13.6)\\
&\le \log_2 \frac{2((|L|+1)+(|R|+1))}{2} && (13.7)\\
&= \log_2(|A|+1) && (13.8)
\end{aligned}
$$

证毕。

与随机性有关的证明技巧

以下列举几个比较有用的技巧。

自然，这寥寥几项不可能就是全部；如果你了解某种没有列出的技巧，那么欢迎补充。

概率上界的分析

详见 概率不等式 页面。

除了上述页面中提到的各种不等式外，推导过程中还经常会用到以下结论：

自然常数的使用：$\left(1-\dfrac{1}{n}\right)^n \le \dfrac{1}{\mathrm{e}}, \forall n \ge 1$

- 左式关于 $n \ge 1$ 单调递增且在 $+\infty$ 处的极限是 $\dfrac{1}{\mathrm{e}}$，因此有这个结论。
- 这告诉我们，如果 n 个互相独立的事件，每个的发生概率为 $1-\dfrac{1}{n}$，则它们全部发生的概率至多为 $\dfrac{1}{\mathrm{e}}$。

「耦合」思想

「耦合」思想常用于同时处理超过一个有随机性的对象，或者同时处理随机的对象和确定性的对象。

引子：随机图的连通性

"问题"

对于 $n \in \mathbf{N}^*; p,q \in [0,1]$ 且 $q \le p$，求证：随机图 $G_1(n,p)$ 的连通分量个数的期望值不超过随机图 $G_2(n,q)$ 的连通分量个数的期望值。这里 $G(n,\alpha)$ 表示一张 n 个结点的简单无向图 G，其中 $\dfrac{n(n-1)}{2}$ 条可能的边中的每一条都有 α 的概率出现，且这些概率互相独立。

这个结论看起来再自然不过，但严格证明却并不那么容易。

"证明思路"

我们假想这两张图分别使用了一个 01 随机数生成器来获知每条边存在与否，其中 G_1 的生成器 T_1 每次以 p 的概率输出 1，G_2 的生成器 T_2 每次以 q 的概率输出 1。这样，要构造一张图，就只需把对应的生成器运行 $\dfrac{n(n-1)}{2}$ 遍即可。

现在我们把两个生成器合二为一。考虑随机数生成器 T，每次以 q 的概率输出 0，以 $p-q$ 的概率输出 1，以 $1-p$ 的概率输出 2。如果我们将这个 T 运行 $\dfrac{n(n-1)}{2}$ 遍，就能同时构造出 G_1 和 G_2。具体地说，如果输出是 0，则认为 G_1 和 G_2 中都没有当前考虑的边；如果输出是 1，则认为只有 G_1 中有当前考虑的边；如果输出是 2，则认为 G_1 和 G_2 中都有当前考虑的边。

容易验证，这样生成的 G_1 和 G_2 符合其定义，而且在每个实例中，G_2 的边集都是 G_1 边集的子集。因此在每个实例中，G_2 的连通分量个数都不小于 G_1 的连通分量个数；那么期望值自然也满足同样的大小关系。

这一段证明中用到的思想被称为「耦合」，可以从字面意思来理解这种思想。本例中它体现为把两个本来独立的随机过程合二为一。

应用：NERC 2019 Problem G: Game Relics[12]

"简要题意"

有若干个物品，每个物品有一个价格 c_i。你想要获得所有物品，为此你可以任意地进行两种操作：

1. 选择一个未拥有的物品 i，花 c_i 块钱买下来。
2. 花 x 块钱从所有物品（包括已经拥有的）中等概率随机抽取一个。如果尚未拥有该物品，则直接获得它；否则一无所获，但是会返还 $\dfrac{x}{2}$ 块钱。x 为输入的常数。

问最优策略下的期望花费。

观察：如果选择抽物品，就一定会一直抽直到获得新物品为止。

- 理由：如果抽一次没有获得新物品，则新的局面和抽物品之前的局面一模一样，所以如果旧局面的最优行动是「抽一发」，则新局面的最优行动一定也是「再抽一发」。

我们可以计算出 f_k 表示：如果当前已经拥有 k 个不同物品，则期望要花多少钱才能抽到新物品。根据刚才的观察，我们可以直接把 f_k 当作一个固定的代价，即转化为「每次花 f_k 块钱随机获得一个新物品」。

" 期望代价的计算"

显然 $f_k = \frac{x}{2} \cdot (R-1) + x$，其中 R 表示要得到新物品期望的抽取次数。

引理：如果一枚硬币有 p 的概率掷出正面，则首次掷出正面所需的期望次数为 $\frac{1}{p}$。

- 感性理解：$\frac{1}{p} \cdot p = 1$，所以扔这么多次期望得到 1 次正面，看起来就比较对。
- 这种感性理解可以通过 大数定律[13] 严谨化，即考虑 $n \to \infty$ 次「不断抛硬币直到得到正面」的实验。推导细节略。
- 另一种可行的证法是，直接把期望的定义带进去暴算。推导细节略。

显然抽一次得到新物品的概率是 $\frac{n-k}{n}$，那么 $R = \frac{n}{n-k}$。

结论：最优策略一定是先抽若干次，再买掉所有没抽到的物品。

这个结论符合直觉，因为 f_k 是关于 k 递增的，早抽似乎确实比晚抽看起来好一点。

" 证明"

先考虑证明一个特殊情况。将证：

- 随机过程 A：先买物品 x，然后不断抽直到得到所有物品
- ……一定不优于……
- 随机过程 B：不断抽直到得到 x 以外的所有物品，然后如果还没有 x 则买下来

考虑让随机过程 A 和随机过程 B 使用同一个随机数生成器。即，A 的第一次抽取和 B 的第一次抽取会抽到同一个元素，第二次、第三次……也是一样。

显然，此时 A 和 B 抽取的次数必定相等。对于一个被 A 抽到的物品 $y \neq x$，观察到：

- A 中抽到 y 时已经持有的物品数，一定大于等于 B 中抽到 y 时已经持有的物品数。

因此 B 的单次抽取代价不高于 A 的单次抽取代价，进而抽取的总代价也不高于 A。

显然 B 的购买代价同样不高于 A。综上，B 一定不劣于 A。

然后可以通过数学归纳把这一结论推广到一般情况。具体地说，每次我们找到当前策略中的最后一次购买，然后根据上述结论，把这一次购买移到最后一定不劣。细节略。

基于这个结论，我们再次等价地转化问题：把「选一个物品并支付对应价格购买」的操作，改成「随机选一个未拥有的物品并支付对应价格购买」。等价性的理由是，既然购买只是用来扫尾的，那选到哪个都无所谓。

现在我们发现，「抽取」和「购买」，实质上已经变成了相同的操作，区别仅在于付出的价格不同。选择购买还是抽取，对于获得物品的顺序毫无影响，而且每种获得物品的顺序都是等可能的。

观察：在某一时刻，我们应当选择买，当且仅当下一次抽取的代价（由已经抽到的物品数确定）大于剩余物品的平均价格（等于的话则任意）。

- 可以证明，随着时间的推移，抽取代价的增速一定不低于剩余物品均价的增速。这说明从抽到买的「临界点」只有一个，进一步验证了先前结论。

最后，我们枚举所有可能的局面（即已经拥有的元素集合），算出这种局面出现的概率（已有元素的排列方案数除以总方案数），乘上当前局面最优决策的代价（由拥有元素个数和剩余物品总价确定），再加起来即可。这个过程可以用背包式的 DP 优化，即可通过本题。

回顾：可以看到，耦合的技巧在本题中使用了两次。第一次是在证明过程中，令两个随机过程使用同一个随机源；

第二次是把购买转化成随机购买（即引入随机源），从而使得购买和抽取这两种操作实质上「耦合」为同一种操作（即令抽取和购买操作共享一个随机源）。

参考资料

[1] PANIC - Editorial

[2] UOJ NOI Round #4 Day2 题解

[3] Anna Gambin and Adam Malinowski, Randomized Meldable Priority Queues

[4] CodeChef SELEDGE

[5] Gym 101550I

[6] CSES 1685 New Flight Routes

[7] TopCoder MagicMolecule

[8] UOJ #207 共价大爷游长沙

[9] CodeChef PANIC

[10] UOJ #552 同构判定鸭

[11] 「TJOI2015」线性代数

[12] NERC 2019 Problem G: Game Relics

[13] 大数定律

13.6.3 爬山算法

简介

爬山算法是一种局部择优的方法，采用启发式方法，是对深度优先搜索的一种改进，它利用反馈信息帮助生成解的决策。

直白地讲，就是当目前无法直接到达最优解，但是可以判断两个解哪个更优的时候，根据一些反馈信息生成一个新的可能解。

因此，爬山算法每次在当前找到的最优方案 x 附近寻找一个新方案。如果这个新的解 x' 更优，那么转移到 x'，否则不变。

这种算法对于单峰函数显然可行。

Q：都知道是单峰函数了为什么不三分呢？

A：爬山算法的优势在于当正解的写法你并不了解（常见于毒瘤计算几何和毒瘤数学题），或者本身状态维度很多，无法容易地写分治（例 2 就可以用二分完成合法正解）时，可以通过非常暴力的计算得到最优解。

但是对于多数需要求解的函数，爬山算法很容易进入一个局部最优解，如下图（最优解为 $\Uparrow$，而爬山算法可能找

到的最优解为 ⇓）。

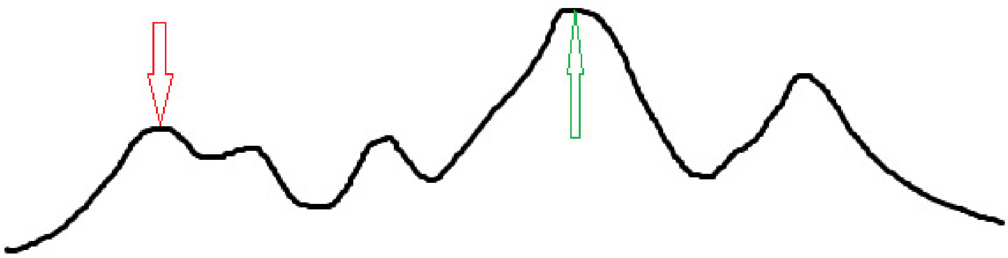


图 13.2

具体实现

爬山算法一般会引入温度参数（类似模拟退火）。类比地说，爬山算法就像是一只兔子喝醉了在山上跳，它每次都会朝着它所认为的更高的地方（这往往只是个不准确的趋势）跳，显然它有可能一次跳到山顶，也可能跳过头翻到对面去。不过没关系，兔子翻过去之后还会跳回来。显然这个过程很没有用，兔子永远都找不到出路，所以在这个过程中兔子冷静下来并在每次跳的时候更加谨慎，少跳一点，以到达合适的最优点。

兔子逐渐变得清醒的过程就是降温过程，即温度参数在爬山的时候会不断减小。

关于降温：降温参数是略小于 1 的常数，一般在 $[0.985, 0.999]$ 中选取。

例 1 「JSOI2008」 球形空间产生器[1]

题意：给出 n 维空间中的 $n+1$ 个点，已知它们在同一个 n 维球面上，求出球心。$n \le 10$，坐标绝对值不超过 20000。

很明显的单峰函数，可以使用爬山解决。本题算法流程：

1. 初始化球心为各个给定点的重心（即其各维坐标均为所有给定点对应维度坐标的平均值），以减少枚举量。
2. 对于当前的球心，求出每个已知点到这个球心欧氏距离的平均值。
3. 遍历所有已知点。记录一个改变值 $cans$（分开每一维度记录）对于每一个点的欧氏距离，如果大于平均值，就把改变值加上差值，否则减去。实际上并不用判断这个大小问题，只要不考虑绝对值，直接用坐标计算即可。这个过程可以形象地转化成一个新的球心，在空间里推来推去，碰到太远的点就往点的方向拉一点，碰到太近的点就往点的反方向推一点。
4. 将我们记录的 $cans$ 乘上温度，更新球心，回到步骤 2
5. 在温度小于某个给定阈值的时候结束。

因此，我们在更新球心的时候，不能直接加上改变值，而是要加上改变值与温度的乘积。

并不是每一道爬山题都可以具体地用温度解决，这只是一个例子。

例题参考代码

```
#include <bits/stdc++.h>
using namespace std;
double ans[10001], cans[100001], dis[10001], tot, f[1001][1001];
int n;

void check() {
```

```
  tot = 0;
  for (int i = 1; i <= n + 1; i++) {
    dis[i] = 0;
    cans[i] = 0;
    for (int j = 1; j <= n; j++)
      dis[i] += (f[i][j] - ans[j]) * (f[i][j] - ans[j]);
    dis[i] = sqrt(dis[i]);  // 欧氏距离
    tot += dis[i];
  }
  tot /= (n + 1);  // 平均
  for (int i = 1; i <= n + 1; i++)
    for (int j = 1; j <= n; j++)
      cans[j] += (dis[i] - tot) * (f[i][j] - ans[j]) /
                 tot;  // 对于每个维度把修改值更新掉，欧氏距离差 * 差值贡献
}

int main() {
  cin >> n;
  for (int i = 1; i <= n + 1; i++)
    for (int j = 1; j <= n; j++) {
      cin >> f[i][j];
      ans[j] += f[i][j];
    }
  for (int i = 1; i <= n; i++) ans[i] /= (n + 1);      // 初始化
  for (double t = 10001; t >= 0.0001; t *= 0.99995) {  // 不断降温
    check();
    for (int i = 1; i <= n; i++) ans[i] += cans[i] * t;  // 修改
  }
  for (int i = 1; i <= n; i++) printf("%.3f ", ans[i]);
}
```

例 2 「BZOJ 3680」 吊打 XXX[2]

题意：求 n 个点的带权类费马点。

框架类似，用了点物理知识。

参考代码

```
#include <cmath>
#include <cstdio>
const int N = 10005;
int n, x[N], y[N], w[N];
double ansx, ansy;

void hillclimb() {
  double t = 1000;
  while (t > 1e-8) {
    double nowx = 0, nowy = 0;
    for (int i = 1; i <= n; ++i) {
      double dx = x[i] - ansx, dy = y[i] - ansy;
      double dis = sqrt(dx * dx + dy * dy);
      nowx += (x[i] - ansx) * w[i] / dis;
      nowy += (y[i] - ansy) * w[i] / dis;
    }
```

```
    ansx += nowx * t, ansy += nowy * t;
    if (t > 0.5)
      t *= 0.5;
    else
      t *= 0.97;
  }
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i <= n; ++i) {
    scanf("%d%d%d", &x[i], &y[i], &w[i]);
    ansx += x[i], ansy += y[i];
  }
  ansx /= n, ansy /= n;
  hillclimb();
  printf("%.3lf %.3lf\n", ansx, ansy);
  return 0;
}
```

优化

很容易想到的是，为了尽可能获取优秀的答案，我们可以多次爬山。方法有修改初始状态/修改降温参数/修改初始温度等，然后开一个全局最优解记录答案。每次爬山结束之后，更新全局最优解。

这样处理可能会存在的问题是超时，在正式考试时请手造大数据测试调参。

劣势

其实爬山算法的劣势上文已经提及：它容易陷入一个局部最优解。当目标函数不是单峰函数时，这个劣势是致命的。因此我们要引进 **模拟退火**。

参考资料与注释

[1] 「JSOI2008」球形空间产生器

[2] 「BZOJ 3680」吊打 XXX

13.6.4 模拟退火

引入

模拟退火是一种随机化算法。当一个问题的方案数量极大（甚至是无穷的）而且不是一个单峰函数时，我们常使用模拟退火求解。

解释

根据 爬山算法 的过程，我们发现：对于一个当前最优解附近的非最优解，爬山算法直接舍去了这个解。而很多情况下，我们需要去接受这个非最优解从而跳出这个局部最优解，即为模拟退火算法。

” 什么是退火？（选自 百度百科[1]）”

退火是一种金属热处理工艺，指的是将金属缓慢加热到一定温度，保持足够时间，然后以适宜速度冷却。目的是降低硬度，改善切削加工性；消除残余应力，稳定尺寸，减少变形与裂纹倾向；细化晶粒，调整组织，消除组织缺陷。准确的说，退火是一种对材料的热处理工艺，包括金属材料、非金属材料。而且新材料的退火目的也与传统金属退火存在异同。

由于退火的规律引入了更多随机因素，那么我们得到最优解的概率会大大增加。于是我们可以去模拟这个过程，将目标函数作为能量函数。

过程

先用一句话概括：如果新状态的解更优则修改答案，否则以一定概率接受新状态。

我们定义当前温度为 T，新状态 S' 与已知状态 S（新状态由已知状态通过随机的方式得到）之间的能量（值）差为 ΔE（$\Delta E \geqslant 0$），则发生状态转移（修改最优解）的概率为

$$P(\Delta E) = \begin{cases} 1, & S' \text{ is better than } S, \\ e^{\frac{-\Delta E}{T}}, & \text{otherwise.} \end{cases}$$

注意：我们有时为了使得到的解更有质量，会在模拟退火结束后，以当前温度在得到的解附近多次随机状态，尝试得到更优的解（其过程与模拟退火相似）。

如何退火（降温）？

模拟退火时我们有三个参数：初始温度 T_0，降温系数 d，终止温度 T_k。其中 T_0 是一个比较大的数，d 是一个非常接近 1 但是小于 1 的数，T_k 是一个接近 0 的正数。

首先让温度 $T = T_0$，然后按照上述步骤进行一次转移尝试，再让 $T = d \cdot T$。当 $T < T_k$ 时模拟退火过程结束，当前最优解即为最终的最优解。

注意为了使得解更为精确，我们通常不直接取当前解作为答案，而是在退火过程中维护遇到的所有解的最优值。

引用一张 Simulated annealing - Wikipedia[2] 的图片（随着温度的降低，跳跃越来越不随机，最优解也越来越稳定）。

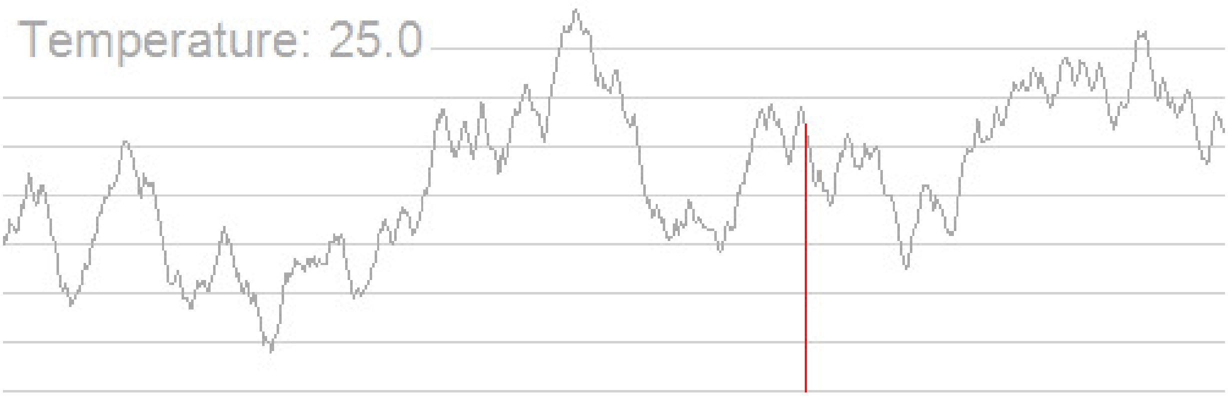


图 13.3

实现

此处代码以「BZOJ 3680」吊打 XXX[3]（求 n 个点的带权类费马点）为例。

```
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <ctime>
```

```cpp
const int N = 10005;
int n, x[N], y[N], w[N];
double ansx, ansy, dis;

double Rand() { return (double)rand() / RAND_MAX; }

double calc(double xx, double yy) {
  double res = 0;
  for (int i = 1; i <= n; ++i) {
    double dx = x[i] - xx, dy = y[i] - yy;
    res += sqrt(dx * dx + dy * dy) * w[i];
  }
  if (res < dis) dis = res, ansx = xx, ansy = yy;
  return res;
}

void simulateAnneal() {
  double t = 100000;
  double nowx = ansx, nowy = ansy;
  while (t > 0.001) {
    double nxtx = nowx + t * (Rand() * 2 - 1);
    double nxty = nowy + t * (Rand() * 2 - 1);
    double delta = calc(nxtx, nxty) - calc(nowx, nowy);
    if (exp(-delta / t) > Rand()) nowx = nxtx, nowy = nxty;
    t *= 0.97;
  }
  for (int i = 1; i <= 1000; ++i) {
    double nxtx = ansx + t * (Rand() * 2 - 1);
    double nxty = ansy + t * (Rand() * 2 - 1);
    calc(nxtx, nxty);
  }
}

int main() {
  srand(0);  // 注意，在实际使用中，不应使用固定的随机种子。
  scanf("%d", &n);
  for (int i = 1; i <= n; ++i) {
    scanf("%d%d%d", &x[i], &y[i], &w[i]);
    ansx += x[i], ansy += y[i];
  }
  ansx /= n, ansy /= n, dis = calc(ansx, ansy);
  simulateAnneal();
  printf("%.3lf %.3lf\n", ansx, ansy);
  return 0;
}
```

一些技巧

分块模拟退火

有时函数的峰很多，模拟退火难以跑出最优解。

此时可以把整个值域分成几段，每段跑一遍模拟退火，然后再取最优解。

卡时

有一个 `clock()` 函数，返回程序运行时间。

可以把主程序中的 `simulateAnneal();` 换成 `while ((double)clock()/CLOCKS_PER_SEC < MAX_TIME) simulateAnneal();`。这样子就会一直跑模拟退火，直到用时即将超过时间限制。

这里的 `MAX_TIME` 是一个自定义的略小于时限的数（单位：秒）。

习题

- 「BZOJ 3680」 吊打 XXX[3]
- 「JSOI 2016」 炸弹攻击[4]
- 「HAOI 2006」 均分数据[5]

参考资料与注释

[1] 百度百科

[2] Simulated annealing - Wikipedia

[3] 「BZOJ 3680」吊打 XXX [3-1] [3-2]

[4] 「JSOI 2016」炸弹攻击

[5] 「HAOI 2006」均分数据

13.7 悬线法

Authors: mwsht, sshwy, ouuan, Ir1d, Henry-ZHR, hsfzLZH1

引入

悬线法的适用范围是单调栈的子集。具体来说，悬线法可以应用于满足以下条件的题目：

- 需要在扫描序列时维护单调的信息；
- 可以使用单调栈解决；
- 不需要在单调栈上二分。

看起来悬线法可以被替代，用处不大，但是悬线法概念比单调栈简单，更适合初学 OI 的选手理解并解决最大子矩阵等问题。

例题

"SP1805 HISTOGRA - Largest Rectangle in a Histogram[1]"

大意：在一条水平线上有 n 个宽为 1 的矩形，求包含于这些矩形的最大子矩形面积。

悬线，就是一条竖线，这条竖线有初始位置和高度两个性质，可以在其上端点不超过当前位置的矩形高度的情况下左右移动。

对于一条悬线，我们在这条上端点不超过当前位置的矩形高度且不移出边界的前提下，将这条悬线左右移动，求出其最多能向左和向右扩展到何处，此时这条悬线扫过的面积就是包含这条悬线的尽可能大的矩形。容易发现，最大子矩形必定是包含一条初始位置为 i，高度为 h_i 的悬线。枚举实现这个过程的时间复杂度为 $O(n^2)$，但是我们可以用悬线法将其优化到 $O(n)$。

我们考虑如何快速找到悬线可以到达的最左边的位置。

过程

定义 l_i 为当前找到的 i 位置的悬线能扩展到的最左边的位置，容易得到 l_i 初始为 i，我们需要进一步判断还能不能进一步往左扩展。

- 如果当前 $l_i = 1$，则已经扩展到了边界，不可以。
- 如果当前 $a_i > a_{l_i-1}$，则从当前悬线扩展到的位置不能再往左扩展了。
- 如果当前 $a_i \le a_{l_i-1}$，则从当前悬线还可以往左扩展，并且 $l_i - 1$ 位置的悬线能向左扩展到的位置，i 位置的悬线一定也可以扩展到，于是我们将 l_i 更新为 l_{l_i-1}，并继续执行判断。

通过摊还分析，可以证明每个 l_i 最多会被其他的 l_j 遍历到一次，因此时间复杂度为 $O(n)$。

实现

参考代码

```
#include <algorithm>
#include <cstdio>
using std::max;
const int N = 100010;
int n, a[N];
int l[N], r[N];
long long ans;

int main() {
  while (scanf("%d", &n) != EOF && n) {
    ans = 0;
    for (int i = 1; i <= n; i++) scanf("%d", &a[i]), l[i] = r[i] = i;
    for (int i = 1; i <= n; i++)
      while (l[i] > 1 && a[i] <= a[l[i] - 1]) l[i] = l[l[i] - 1];
    for (int i = n; i >= 1; i--)
      while (r[i] < n && a[i] <= a[r[i] + 1]) r[i] = r[r[i] + 1];
    for (int i = 1; i <= n; i++)
      ans = max(ans, (long long)(r[i] - l[i] + 1) * a[i]);
    printf("%lld\n", ans);
  }
  return 0;
}
```

"UVa1619 感觉不错 Feel Good[2]"

对于一个长度为 n 的数列，找出一个子区间，使子区间内的最小值与子区间长度的乘积最大，要求在满足舒适值最大的情况下最小化长度，最小化长度的情况下最小化左端点序号。

本题中我们可以考虑枚举最小值，将每个位置的数 a_i 当作最小值，并考虑从 i 向左右扩展，找到满足 $\min\limits_{j=l}^{r} a_j = a_i$ 的尽可能向左右扩展的区间 $[l, r]$。这样本题就被转化成了悬线法模型。

参考代码

```
#include <cstdio>
#include <cstring>
const int N = 100010;
int n, a[N], l[N], r[N];
long long sum[N];
long long ans;
int ansl, ansr;
bool fir = 1;

int main() {
  while (scanf("%d", &n) != EOF) {
    memset(a, -1, sizeof(a));
    if (!fir)
      printf("\n");
    else
      fir = 0;
    ans = 0;
    ansl = ansr = 1;
    for (int i = 1; i <= n; i++) {
      scanf("%d", &a[i]);
      sum[i] = sum[i - 1] + a[i];
      l[i] = r[i] = i;
    }
    for (int i = 1; i <= n; i++)
      while (a[l[i] - 1] >= a[i]) l[i] = l[l[i] - 1];
    for (int i = n; i >= 1; i--)
      while (a[r[i] + 1] >= a[i]) r[i] = r[r[i] + 1];
    for (int i = 1; i <= n; i++) {
      long long x = a[i] * (sum[r[i]] - sum[l[i] - 1]);
      if (ans < x || (ans == x && ansr - ansl > r[i] - l[i]))
        ans = x, ansl = l[i], ansr = r[i];
    }
    printf("%lld\n%d %d\n", ans, ansl, ansr);
  }
  return 0;
}
```

最大子矩形

"P4147 玉蟾宫[3]"

给定一个 $n \times m$ 的包含 'F' 和 'R' 的矩阵，求其面积最大的子矩阵的面积 $\times 3$，使得这个子矩阵中的每一位的值都为 'F'。

我们会发现本题的模型和第一题的模型很像。仔细分析，发现如果我们每次只考虑某一行的所有元素，将位置 (x, y) 的元素尽可能向上扩展的距离作为该位置的悬线长度，那最大子矩阵一定是这些悬线向左右扩展得到的尽可能大的矩形中的一个。

参考代码

```cpp
#include <algorithm>
#include <cstdio>
int m, n, a[1010], l[1010], r[1010], ans;

int main() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
      l[j] = r[j] = j;
    }
    char s[3];
    for (int j = 1; j <= m; j++) {
      scanf("%s", s);
      if (s[0] == 'F')
        a[j]++;
      else if (s[0] == 'R')
        a[j] = 0;
    }
    for (int j = 1; j <= m; j++)
      while (l[j] != 1 && a[l[j] - 1] >= a[j]) l[j] = l[l[j] - 1];
    for (int j = m; j >= 1; j--)
      while (r[j] != m && a[r[j] + 1] >= a[j]) r[j] = r[r[j] + 1];
    for (int j = 1; j <= m; j++) ans = std::max(ans, (r[j] - l[j] + 1) * a[j]);
  }
  printf("%d", ans * 3);
  return 0;
}
```

习题

- P1169「ZJOI2007」 棋盘制作[4]

参考资料与注释

[1] SP1805 HISTOGRA - Largest Rectangle in a Histogram

[2] UVa1619 感觉不错 Feel Good

[3] P4147 玉蟾宫

[4] P1169「ZJOI2007」棋盘制作

13.8 计算理论基础

本部分将介绍基础的计算理论的知识。这部分内容在 OI 中作用不大（但还是略有作用：如果你遇到了一个 NP-hard 问题，你可以认为它是不存在多项式复杂度的解法的），可以作为兴趣了解，或者为以后的学习做准备。

本文中许多结论都是不加证明的，如果有兴趣的话可以自行查阅相关证明。

前置知识：时间复杂度。

问题

语言

一个**字母表（alphabet）**是一个非空有限集合，该集合中的元素称为**符号/字符（symbol）**。

令 Σ^* 表示非负整数个 Σ 中的字符连接而成的串，字母表 Σ 上的一个**语言（language）**是 Σ^* 的一个子集。

需要注意的是，这里的「语言」是一个抽象的概念，通常意义上的字符串是语言，所有的有向无环图也可以是一个语言（01 串与有向图之间可以建立双射，具体方式无需了解）。

由于任何语言都可以转化成 01 串的形式，所以在下文中不加说明时 $\Sigma = \{0, 1\}$。

判定问题

判定问题就是只能用 YES/NO 回答的问题，本质上是判定一个串是否属于一个语言，即：$f : \Sigma^* \to \{0, 1\}, f(x) = 1 \iff x \in L$ 是一个关于字母表 Σ 和语言 L 的判定问题。如，「判定一张图是不是一个有向无环图」就是一个判定问题。

判定问题由于其简洁性而常常被作为计算理论研究的对象。本文中不加说明时，「问题」都指「判定问题」，当然，有时一些命题也能简单地推广到其它问题上。

一个语言也可以代指「判定一个串是否属于这个语言」这个判定问题，因此，「语言」和「问题」可以视作同义词。

功能性问题

功能性问题的回答不止 YES/NO，可以是一个数或是其它。如，「求两个数的和」就是一个功能性问题。

任何功能性问题都可以转化为一个判定问题，如，「求两个数的和」可以转化为「判定两个数的和是否等于第三个数」。

判定问题也可以转化为一个功能性问题：求这个判定问题的指示函数，即上文中判定问题定义里的 f。

图灵机

确定性图灵机

不加说明时，「图灵机」往往指「确定性图灵机」，本文中也是如此。

图灵机有很多不同的定义，这里选取其中一种，其它定义下的图灵机往往与下面这种定义的图灵机计算能力等价。

图灵机是一个在一条可双向无限延伸且被划分为若干格子的纸带上进行操作的机器，其有内部状态，还有一个可以在纸带上进行修改与移动的磁针。

正式地说，图灵机是一个七元组 $M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$，其中：

- Q 是一个有限非空的**状态集合**；
- Γ 是一个有限非空的**磁带字母表**；
- $b \in \Gamma$ 是**空字符**，它是唯一一个在计算过程中可以在磁带上无限频繁地出现的字符；
- $\Sigma \subseteq (\Gamma \quad \{b\})$ 是**输入符号集**，是可以出现在初始磁带（即输入）上的字符；
- $q_0 \in Q$ 是**初始状态**；
- $F \subseteq Q$ 是**接受状态**，如果一个图灵机在某个接受状态停机，则称初始磁带上的内容被这个图灵机**接受**。
- $\delta : (Q \quad F) \times \Gamma \nrightarrow Q \times \Gamma \times \{L, R\}$ 是一个被称作**转移函数**的 partial function（即只对定义域的一个子集有定义的函数）。如果 δ 在当前状态下没有定义，则图灵机停机。

图灵机从初始状态与纸带起点起，每次根据当前的内部状态 x 和当前磁针指向的纸带上的单元格中的字符 y 进行操作：若 $\delta(x, y)$ 没有定义则停机，否则若 $\delta(x, y) = (a, b, c)$，则将内部状态修改为 a，将磁针指向的格子中的字符修改为 b，若 c 为 L 则向左移动一格，为 R 则向右移动一格。

其实，知道图灵机的工作细节是不必要的，只需建立直观理解即可。

图灵机 M 在输入 x 下的输出记作 $M(x)$（$M(x)=1$ 当且仅当 M 接受 x，$M(x)=0$ 当且仅当 M 在输入 x 下在有限步骤内停机且 M 不接受 x），也可以在括号内包含多个参数，用逗号隔开，具体实现时可以向字母表中添加一个元素表示逗号来隔开各个参数。

图灵机与冯 · 诺依曼计算机解决问题的时间复杂度差别在多项式级别内，所以研究复杂度类时可以使用图灵机作为计算模型。

非确定性图灵机

非确定型图灵机是图灵机的一种，它与确定型图灵机的不同在于：确定型图灵机的每一步只能转移到一个状态，而非确定型图灵机可以「同时」转移到多个状态，从而在多个「分支」并行计算，一旦这些「分支」中有一个在接受状态停机，则此非确定性图灵机接受这个输入。

事实上，任何确定型图灵机都可以用类似于迭代加深搜索的方式在指数级时间内模拟一台非确定型图灵机多项式时间内的行为。

在现实生活中，确定型图灵机相当于单核处理器，只支持串行处理；而非确定型图灵机相当于理想的多核处理器，支持无限大小的并行处理。

多带图灵机

标准的图灵机只能在一条纸带上进行操作，但为了方便，本文中研究多带图灵机。对于一个 k 带图灵机，其中一条纸带是只读的输入带，而剩下的 $k-1$ 条纸带可以进行读写，并且这 $k-1$ 条纸带中还有一条纸带用作输出。

多带图灵机的纸带数必须是有限的。

对于一个多带图灵机，它使用的空间是磁头在除输入带外的其它纸带上所访问过的单元格数目。

图灵机的编码

图灵机可以被自然数编码，即存在满射函数 $f:\mathbb{N}\to\mathbb{M}$，使得每个自然数都对应一个图灵机，而每个图灵机都有无数个编码。因此，由若干图灵机构成的集合可以是一个语言。

记由自然数 α 编码的图灵机为 M_α。

通用图灵机

存在一台图灵机 $\mathcal{U}$ 满足：

1. 若 M_α 在输入 x 下在有限时间内停机，则 $\mathcal{U}(x,\alpha)=M_\alpha(x)$，否则 $\mathcal{U}(x,\alpha)$ 不会在有限时间内停机；
2. 如果对于任意 $x\in\{0,1\}^*$，M_α 在输入 x 下在 $T(|x|)$ 时间内停机，则对于任意 $x\in\{0,1\}^*$，$\mathcal{U}(x,\alpha)$ 在 $O(T(|x|)\log T(|x|))$ 时间内停机。

即：存在一台通用图灵机，它能模拟任何一台图灵机，且花费的时间只会比这台被模拟的图灵机慢其运行时间的对数。

可计算性

不可计算问题

对于一个判定问题，若存在一个总是在有限步内停机且能够正确进行判定的图灵机，则这个问题是一个**图灵可计算**的问题，否则这个问题是一个**图灵不可计算**的问题。

由于图灵机可以被自然数编码，所以图灵机的个数是可数无穷，而语言（即二进制串的集合）的个数是不可数无穷，而每个图灵机最多判定一个语言，所以一定存在图灵不可计算的问题。

停机问题

停机问题是一个经典的图灵不可计算问题：给定 α 和 x，判定 M_α 在输入为 x 时是否会在有限步内停机。

"停机问题是图灵不可计算的证明"

定义函数 $\mathsf{UC}:\{0,1\}^*\to\{0,1\}$ 为：

$$\mathsf{UC}(\alpha)=\begin{cases}0 & M_\alpha(\alpha)=1\\ 1 & \text{otherwise}\end{cases}$$

我们先证明 UC 函数是图灵不可计算的：

假设存在一台图灵机 M_β 能够计算 UC，那么根据 UC 的定义可以得到 $\mathsf{UC}(\beta)=1 \iff M_\beta(\beta)\neq 1$，而根据 M_β 能够计算 UC 可以得到 $M_\beta(\beta)=\mathsf{UC}(\beta)$，产生了矛盾，所以假设不成立，不存在可以计算 UC 的图灵机。

令 M_{HALT} 是一个可以解决停机问题的图灵机，$M_{\mathsf{HALT}}(x,\alpha)$ 的值是判定问题 M_α 在输入为 x 时是否会在有限步内停机的解，那么我们可以构造出一台能够计算 UC 函数的图灵机 M_{UC}：

M_{UC} 首先调用 $M_{\mathsf{HALT}}(\alpha,\alpha)$, 如果它输出 0, 则 $M_{\mathsf{UC}}(\alpha)=1$；否则，$M_{\mathsf{UC}}$ 使用通用图灵机模拟计算得到答案。

由于 UC 函数是图灵不可计算的，所以 M_{HALT} 不存在，也就是说停机问题是图灵不可计算的。

丘奇 - 图灵论题

丘奇 - 图灵论题称，若一类问题有一个有效的方法解决，则这类问题可以被某个图灵机解决。

其中，「有效的方法」需要满足：

1. 包含有限条清晰的指令；
2. 当用其解决这类问题的其中一个时，这个方法需要在有限步骤内结束，且得到正确的答案。

这个论题没有被证明，但其是计算理论的一条基本公理。

复杂度类

复杂度类有很多，本文只会介绍其中较为常见的一小部分。

R 和 RE

对于语言 L 和图灵机 M，若 M 在任何输入下都能在有限步骤内停机，且 $M(x)=1 \iff x\in L$，则称 M 能够**判定** L。

对于语言 L 和图灵机 M，若对于任何属于 L 的输入，M 都在有限步骤内停机，且 $M(x)=1 \iff x\in L$，则称 M 能够**识别** L。

复杂度类 R 表示那些可以被某台图灵机判定的语言的集合，即所有图灵可计算的语言。

复杂度类 RE 表示那些可以被某台图灵机识别的语言的集合。RE 也被称作递归可枚举语言。

由定义可以得到 $\mathsf{R}\subseteq\mathsf{RE}$。

DTIME

如果存在一台确定性图灵机能够判定一个语言，且对于任何输入 x，这台图灵机可以在 $O(f(|x|))$ 的时间内停机，那么这个语言属于 $\mathsf{DTIME}(f(n))$ 类。

P

复杂度类 P 表示可以由确定性图灵机在多项式时间内解决的判定问题，即：

$$\mathsf{P} = \bigcup_{k\in\mathbb{N}} \mathsf{DTIME}(n^k)$$

线性规划、计算最大公约数、求图的最大匹配的判定版本都是 P 类问题。

EXPTIME

复杂度类 EXPTIME 表示可以由确定性图灵机在指数级时间内解决的判定问题，即：

$$\mathsf{EXPTIME} = \bigcup_{k\in\mathbb{N}} \mathsf{DTIME}(2^{n^k})$$

停机问题的弱化版——给定一个图灵机的编码以及一个正整数 k，判定这个图灵机是否在 k 步内停机，是一个 EXPTIME 类的问题。因为这个问题的解法需要 $O(k)$ 的时间，而数字 k 可以被编码为长度为 $O(\log k)$ 的二进制串。

NTIME

如果存在一台非确定性图灵机能够判定一个语言，且对于任何输入 x，这台图灵机可以在 $O(f(|x|))$ 的时间内停机，那么这个语言属于 $\mathsf{NTIME}(f(n))$ 类。

NP

复杂度类 NP 表示可以由非确定性图灵机在多项式时间内解决的判定问题，即：

$$\mathsf{NP} = \bigcup_{k\in\mathbb{N}} \mathsf{NTIME}(n^k)$$

所有 P 类问题都是 NP 类问题。更多 NP 类问题请参见下文中的 NPC 问题以及 NP-intermediate 问题。

NP-hard

如果所有 NP 类问题都可以在多项式时间内规约到问题 H，那么问题 H 是 NP-hard 的。

换句话说，如果可以在一单位的时间内解决 NP-hard 的问题 H，那么所有 NP 类问题都可以在多项式单位的时间内解决。

NP-complete

如果一个问题既是 NP 类问题又是 NP-hard 的，那么这个问题是 NP 完全 (NP-complete) 的，或者说这是一个 NPC 问题。

一些经典的 NPC 问题：旅行商问题的判定版本、最大独立集问题的判定版本、最小点覆盖问题的判定版本、最长路问题的判定版本、0-1 整数规划问题的判定版本、集合覆盖问题、图着色问题、背包问题、三维匹配问题、最大割问题的判定版本。

NPC 问题的功能性版本往往是 NP-hard 的，例如：「判定一张图中是否存在大小为 k 的团」既是一个 NP 类问题又是 NP-hard 的，从而它是一个 NPC 问题，而它的功能性版本「求一张图的最大团」不是 NPC 问题，但这个功能性版本依然是 NP-hard 的。

类似地，其它复杂度类也会有「XX-complete」，如所有 EXPTIME 类的问题都能在多项式时间内规约到 EXPTIME-complete 的问题。

co-NP

一个问题是 $\mathsf{co}-\mathsf{NP}$ 类问题，当且仅当它的补集是 NP 类问题。如果将「问题」理解为「语言」，而「语言」是 Σ^* 的子集，就能理解「补集」了。

例如：「给定 n 个子集，判断是否能够从中选取 k 个，覆盖整个集合」是一个 NPC 问题，而其补集「给定 n 个子集，判断是否从中任取 k 个都不能覆盖整个集合」是一个 $\mathsf{co}-\mathsf{NP}$ 类问题。如果第一个问题的答案是「是」，那么相当于找到了第二个问题的一组反例，从而第二个问题的答案是「否」。

NP-intermediate

如果一个问题是 NP 类问题，但它既不是 P 类问题也不是 NPC 问题，则称其为 NP-intermediate 问题。

就人们目前的了解，图同构问题、离散对数问题和因数分解问题可能是 NP-intermediate 的。

Ladner 定理指出，如果 $\mathsf{P} \neq \mathsf{NP}$，则一定存在问题是 NP-intermediate 的。

NEXPTIME

复杂度类 NEXPTIME 表示可以由非确定性图灵机在指数级时间内解决的判定问题，即：

$$\mathsf{NEXPTIME} = \bigcup_{k\in\mathbb{N}} \mathsf{NTIME}(2^{n^k})$$

#P

#P 类问题不是判定问题，而是关于 NP 类问题的计数问题：数一个 NP 类问题的解的个数是一个 #P 类的问题。换句话说，数一个串在一个总是在多项式时间内停机的非确定性图灵机的多少个分支处被接受是一个 #P 类的问题。

求一张普通图或二分图的匹配或完美匹配个数都是 #P 完全的，对应的判定问题为「判定一张图是否存在（完美）匹配」。

DSPACE

如果存在一台确定性图灵机能够在输入为 x 时在 $O(f(|x|))$ 的空间内判定一个语言，那么这个语言属于 $\mathsf{DSPACE}(f(n))$ 类。

- $\mathsf{REG} = \mathsf{DSPACE}(O(1))$，即正则语言，也就是自动机能够判定的语言。
- $\mathsf{L} = \mathsf{DSPACE}(O(\log n))$，需要注意的是图灵机使用的空间不包括输入占用的空间。
- $\mathsf{PSPACE} = \bigcup_{k\in\mathbb{N}} \mathsf{DSPACE}(n^k)$
- $\mathsf{EXPSPACE} = \bigcup_{k\in\mathbb{N}} \mathsf{DSPACE}(2^{n^k})$

NSPACE

如果存在一台非确定性图灵机能够在输入为 x 时在 $O(f(|x|))$ 的空间内判定一个语言，那么这个语言属于 $\mathsf{NSPACE}(f(n))$ 类。

- $\mathsf{REG} = \mathsf{DSPACE}(O(1)) = \mathsf{NSPACE}(O(1))$
- $\mathsf{NL} = \mathsf{NSPACE}(O(\log n))$
- $\mathsf{CSL} = \mathsf{NSPACE}(O(n))$，即上下文相关语言。
- $\mathsf{PSPACE} = \mathsf{NPSPACE} = \bigcup_{k\in\mathbb{N}} \mathsf{NSPACE}(n^k)$
- $\mathsf{EXPSPACE} = \mathsf{NEXPSPACE} = \bigcup_{k\in\mathbb{N}} \mathsf{NSPACE}(2^{n^k})$

多项式时间

简单来说，如果存在正数 k 使得一个算法的时间复杂度为 $O(n^k)$（注意，不是 $\Theta(n^k)$），其中 n 为问题规模（输入的长度），则称这个算法是**多项式时间**的。如果一个问题有（确定性图灵机上的）多项式时间的算法来解决，则这个问题属于复杂度类 P。

多项式时间可分为强多项式时间和弱多项式时间，除此之外还有伪多项式时间。

Strongly polynomial time 强多项式时间

我们先定义一个计算模型，称作算术模型。在算术模型中，数字之间的算术运算（加减乘除、比较大小）可以在单位时间内完成（即 $O(1)$ 时间内完成，与数字大小无关）。

如果一个算法在算术模型下的操作数是输入中的数字个数的多项式，并且空间复杂度是输入规模（而非数字个数）的多项式，则这个算法是**强多项式时间**的。由于算术操作在一般的计算模型下可以在输入规模（即数字大小的对数）的多项式时间内完成，强多项式时间的算法一定是多项式时间的。

一般来说，强多项式时间的算法的时间复杂度与值域无关。

Weakly polynomial time 弱多项式时间

如果一个算法是多项式时间的但不是强多项式时间的，则它是**弱多项式时间**的。

例如，计算最大公约数的欧几里得算法，时间复杂度为 $O(\log a+\log b)$（a 和 b 为输入的数的大小），是弱多项式时间的。

Pseudo-polynomial time 伪多项式时间

如果一个算法的用时是值域的多项式，则称它是**伪多项式时间**的。伪多项式时间的算法可能是多项式时间的也可能不是，可能不是多项式时间是因为表示一个大小为 n 的正整数一般只需要 $O(\log n)$ 个二进制位，所以关于值域多项式时间的算法往往关于输入长度是指数级时间的。虽然从定义上来说伪多项式时间也可能是多项式时间，但当我们说一个算法是伪多项式时间的，一般都是说这个算法不是多项式时间的。

例如，背包问题是 NP-hard 问题，但它有基于动态规划的伪多项式时间的解法。

如果一个 NPC/NP-hard 问题有伪多项式时间的解法，则称这个问题是**弱 NPC/弱 NP-hard** 问题。如果一个 NPC/NP-hard 问题在 $\mathsf{P}\neq\mathsf{NP}$ 的前提下没有伪多项式时间的解法，则称这个问题是**强 NPC/强 NP-hard** 问题。

可构造函数

时间可构造函数

有时，我们想让图灵机知道自己用了多长的时间，例如，强制图灵机在进行 $T(n)$ 步计算后停机。但如果计算 $T(n)$ 的用时就超过了 $T(n)$，这便是不可做到的。为此，定义了时间可构造函数，来避免这样的麻烦。

如果存在图灵机 M，使得输入为 1^n(n 个 1) 时 M 能在 $O(f(n))$ 的时间内停机并且输出 $f(n)$ 的二进制表示（注意，这里的图灵机的输出不是接受/不接受，而是一个串，输出可以在纸带上进行），则 $f(n)$ 是一个**时间可构造函数**。

由于读入需要 $O(n)$ 的时间，$o(n)$ 的非常值函数都不是时间可构造函数。

空间可构造函数

类似地可以定义空间可构造函数。

如果存在图灵机 M，使得输入为 1^n(n 个 1) 时 M 能在 $O(f(n))$ 的空间内停机并且输出 $f(n)$ 的二进制表示，则 $f(n)$ 是一个**空间可构造函数**。

复杂度类之间的关系

时间谱系定理

确定性时间谱系定理

若 $f(n)$ 是一个时间可构造函数，则：

$$\mathsf{DTIME}\left(o\left(\frac{f(n)}{\log f(n)}\right)\right)\subsetneq\mathsf{DTIME}(f(n))$$

由确定性时间谱系定理可以得到 $\mathsf{P} \subsetneq \mathsf{EXPTIME}$。

" 确定性时间谱系定理的证明"

定义语言 $L = \{(x,y)|\mathcal{U}((x,y),x)$ 在$f(|x|+|y|)$ 时间内停机并拒绝$\}$，由于 $f(n)$ 是一个时间可构造函数，可以根据定义进行计算来判定 L，用时为 $O(f(|x|+|y|))$，所以 $L \in \mathsf{DTIME}(f(n))$。

现在假设 $L \in \mathsf{DTIME}(o\left(\frac{f(n)}{\log f(n)}\right))$，设 M_z 就是那台在 $o\left(\frac{f(n)}{\log f(n)}\right)$ 的时间内判定 L 的图灵机。

令通用图灵机 $\mathcal{U}(x,z)$ 关于 x 的用时为 $g(|x|)$，由上文关于通用图灵机的介绍可以得到 $g(n) = o(f(n))$，所以，当 y 足够大时，$g(|z|+|y|) < f(|z|+|y|)$。

令 y' 是一个足够大的 y，那么 $\mathcal{U}((z,y'),z)$ 一定能在 $f(|z|+|y'|)$ 时间内停机，从而 $M_z(z,y') \neq M_z(z,y')$，产生矛盾，所以假设不成立，确定性时间谱系定理证毕。

非确定性时间谱系定理

若 $g(n)$ 是一个时间可构造函数，并且 $f(n+1) = o(g(n))$，则 $\mathsf{NTIME}(f(n)) \subsetneq \mathsf{NTIME}(g(n))$。

由非确定性时间谱系定理可以得到 $\mathsf{NP} \subsetneq \mathsf{NEXPTIME}$。

空间谱系定理

若 $f(n)$ 是一个空间可构造函数且 $f(n) = \Omega(\log n)$，则 $\mathsf{SPACE}(o(f(n))) \subsetneq \mathsf{SPACE}(f(n))$。

其中 SPACE 可以代指 DSPACE 或 NSPACE。

由空间谱系定理可以得到 $\mathsf{PSPACE} \subsetneq \mathsf{EXPSPACE}$。

萨维奇定理

一台确定性图灵机可以在一台非确定性图灵机所消耗空间的平方内模拟它（尽管消耗的时间可能多很多），即：

若 $f(n) = \Omega(\log n)$，则：

$$\mathsf{NSPACE}\left(f\left(n\right)\right) \subseteq \mathsf{DSPACE}\left(\left(f\left(n\right)\right)^{2}\right)$$

推论：$\mathsf{PSPACE} = \mathsf{NPSPACE}$，$\mathsf{EXPSPACE} = \mathsf{NEXPSPACE}$。

P?=NP

复杂度类 P 与 NP 是否相等是计算复杂度理论中一个著名的尚未解决的问题。

若 $\mathsf{P} = \mathsf{NP}$，可以得到 $\mathsf{NP} = \mathsf{co-NP}$，但反之不行（目前没有基于 $\mathsf{NP} = \mathsf{co-NP}$ 证明 $\mathsf{P} = \mathsf{NP}$ 的方法）。

" 为什么 NP?=co-NP 不是显然的？"

由于 NP 问题和与其对应的 co − NP 问题答案相反，很容易有这种想法：对于一个 co − NP 问题，我只要将解决其补集的非确定性图灵机的输出反过来，就解决了该 co − NP 问题，所以 $\mathsf{NP} = \mathsf{co-NP}$。

实际上，上面所说的这种方法确实能够解决该 co − NP 问题，但并没有找到一个非确定性图灵机来解决它：如果一个图灵机所做的事情是将一个非确定性图灵机的输出反过来，该图灵机并不是一个非确定性图灵机。因为，非确定性图灵机接受是在某个分支处接受，而拒绝是在所有分支处拒绝；而将其输出反过来，就变成了接受是在所有分支处，而拒绝是在一个分支处，而这样就不符合非确定性图灵机的定义了，所以能用该图灵机解决这个 co − NP 问题并不能使这个 co − NP 问题变成一个 NP 问题。

若 $\mathsf{P} = \mathsf{NP}$，还可以得到 $\mathsf{EXPTIME} = \mathsf{NEXPTIME}$。

若 $\mathsf{P} \neq \mathsf{NP}$，可以得到 NP-intermediate 不为空。

参考资料

1. 计算复杂性（1）Warming Up：自动机模型[1]；
2. 计算复杂性（2） 图灵机计算模型[2]；
3. Wikipedia[3] 的相关词条以及这些词条的参考资料。

参考资料与注释

[1] 计算复杂性（1）Warming Up: 自动机模型

[2] 计算复杂性（2）图灵机计算模型

[3] Wikipedia

13.9 字节顺序

本页面将简要介绍字节顺序的概念和分类。

简介

字节顺序是跨越多字节的程序对象的存储规则，表示一个对象的字节的排列方法。

分类

字节顺序有两种，分为小端序（little endian）和大端序（big endian）。

为方便介绍，接下来以一个位于 `0x100` 处，类型为 `int`，十六进制值为 `0x01234567` 的变量为例。其中 `0x01` 是最高位有效字节，`0x67` 是最低位有效字节。

小端序

小端序是指机器选择在内存中按照从**最低**有效字节到**最高**有效字节的顺序存储对象。

上文提到的变量表示如下：

....	0x100	0x101	0x102	0x103	....
....	67	45	23	01	....

大端序

大端序是指机器选择在内存中按照从**最高**有效字节到**最低**有效字节的顺序存储对象。

上文提到的变量表示如下：

....	0x100	0x101	0x102	0x103	....
....	01	23	45	67	....

两种顺序的区别

事实上，这两种字节顺序没有孰优孰劣之分。这两种顺序的名字「小端」和「大端」，正是出自《格列佛游记》一书。书中，小人国里两个派别交战不休的原因是无法就从小端还是大端剥鸡蛋达成一致。就和剥鸡蛋的争论一样，选择何种字节顺序的争论是非技术性的。

当然，字节顺序的不一致会导致二进制数据在不同类型的机器之间进行传输时被反序。为了避免这件事情，网络应用程序建立了一套标准，保证发送过程中是使用约定好的网络标准，而不是不同机器的内部表示。

顺序选择惯例

- 小端序：x86, ARM processors running Android, iOS, and Windows
- 大端序：Sun, PPC Mac, Internet

13.10 约瑟夫问题

约瑟夫问题由来已久，而这个问题的解法也在不断改进，只是目前仍没有一个极其高效的算法（log 以内）解决这个问题。

问题描述

> n 个人标号 $0, 1, \cdots, n-1$。逆时针站一圈，从 0 号开始，每一次从当前的人逆时针数 k 个，然后让这个人出局。问最后剩下的人是谁。

这个经典的问题由约瑟夫于公元 1 世纪提出，尽管他当时只考虑了 $k = 2$ 的情况。现在我们可以用许多高效的算法解决这个问题。

过程

朴素算法

最朴素的算法莫过于直接枚举。用一个环形链表枚举删除的过程，重复 $n-1$ 次得到答案。复杂度 $\Theta(n^2)$。

简单优化

寻找下一个人的过程可以用线段树优化。具体地，开一个 $0, 1, \cdots, n-1$ 的线段树，然后记录区间内剩下的人的个数。寻找当前的人的位置以及之后的第 k 个人可以在线段树上二分做。

线性算法

设 $J_{n,k}$ 表示规模分别为 n, k 的约瑟夫问题的答案。我们有如下递归式

$$J_{n,k} = (J_{n-1,k} + k) \bmod n$$

这个也很好推。你从 0 开始数 k 个，让第 $k-1$ 个人出局后剩下 $n-1$ 个人，你计算出在 $n-1$ 个人中选的答案后，再加一个相对位移 k 得到真正的答案。这个算法的复杂度显然是 $\Theta(n)$ 的。

" 实现"

```
int josephus(int n, int k) {
```

```
  int res = 0;
  for (int i = 1; i <= n; ++i) res = (res + k) % i;
  return res;
}
```

对数算法

对于 k 较小 n 较大的情况，本题还有一种复杂度为 $\Theta(k\log n)$ 的算法。

考虑到我们每次走 k 个删一个，那么在一圈以内我们可以删掉 $\lfloor\frac{n}{k}\rfloor$ 个，然后剩下了 $n-\lfloor\frac{n}{k}\rfloor$ 个人。这时我们在第 $\lfloor\frac{n}{k}\rfloor\cdot k$ 个人的位置上。而你发现它等于 $n-n \bmod k$。于是我们继续递归处理，算完后还原它的相对位置。还原相对位置的依据是：每次做一次删除都会把数到的第 k 个人删除，他们的编号被之后的人逐个继承，也即用 $n-\lfloor\frac{n}{k}\rfloor$ 人环算时每 k 个人即有 1 个人的位置失算，因此在得数小于 0 时，用还没有被删去 k 倍数编号的 n 人环的的 n 求模，在得数大于等于 0 时，即可以直接乘 $\frac{k}{k-1}$，于是得到如下的算法：

" 实现"

```
int josephus(int n, int k) {
  if (n == 1) return 0;
  if (k == 1) return n - 1;
  if (k > n) return (josephus(n - 1, k) + k) % n;  // 线性算法
  int res = josephus(n - n / k, k);
  res -= n % k;
  if (res < 0)
    res += n;  // mod n
  else
    res += res / (k - 1);  // 还原位置
  return res;
}
```

可以证明这个算法的复杂度是 $\Theta(k\log n)$ 的。我们设这个过程的递归次数是 x，那么每一次问题规模会大致变成 $n\left(1-\dfrac{1}{k}\right)$，于是得到

$$n\left(1-\frac{1}{k}\right)^x=1$$

解这个方程得到

$$x=-\frac{\ln n}{\ln\left(1-\frac{1}{k}\right)}$$

下面我们证明该算法的复杂度是 $\Theta(k\log n)$ 的。

" 证明"

考虑 $\lim_{k\to\infty} k\log\left(1-\frac{1}{k}\right)$，我们有

$$
\begin{aligned}
\lim_{k\to\infty} k\log\left(1-\frac{1}{k}\right) &= \lim_{k\to\infty}\frac{\log\left(1-\frac{1}{k}\right)}{1/k} \\
&= \lim_{k\to\infty}\frac{\frac{\mathrm{d}}{\mathrm{d}k}\log\left(1-\frac{1}{k}\right)}{\frac{\mathrm{d}}{\mathrm{d}k}\left(\frac{1}{k}\right)} \\
&= \lim_{k\to\infty}\frac{\frac{1}{k^2\left(1-\frac{1}{k}\right)}}{-\frac{1}{k^2}} \\
&= \lim_{k\to\infty} -\frac{k}{k-1} \\
&= -\lim_{k\to\infty}\frac{1}{1-\frac{1}{k}} \\
&= -1
\end{aligned}
$$

所以 $x \sim k\ln n, k\to\infty$，即 $-\frac{\ln n}{\ln\left(1-\frac{1}{k}\right)} = \Theta(k\log n)$

参考资料与注释

[1]

[2] Josephus Problem

13.11 格雷码

Authors: sshwy

格雷码是一个二进制数系，其中两个相邻数的二进制位只有一位不同。举个例子，3 位二进制数的格雷码序列为

$$000, 001, 011, 010, 110, 111, 101, 100$$

注意序列的下标我们以 0 为起点，也就是说 $G(0) = 000, G(4) = 110$。

格雷码由贝尔实验室的 Frank Gray 于 1940 年代提出，并于 1953 年获得专利。

构造格雷码（变换）

格雷码的构造方法很多。我们首先介绍手动构造方法，然后会给出构造的代码以及正确性证明。

手动构造

k 位的格雷码可以通过以下方法构造。我们从全 0 格雷码开始，按照下面策略：

1. 翻转最低位得到下一个格雷码，（例如 $000 \to 001$）；
2. 把最右边的 1 的左边的位翻转得到下一个格雷码，（例如 $001 \to 011$）；

交替按照上述策略生成 2^{k-1} 次，可得到 k 位的格雷码序列。

镜像构造

k 位的格雷码可以从 $k-1$ 位的格雷码以上下镜射后加上新位的方式快速得到，如下图：

$$\begin{array}{ccccc}
k=1 & & k=2 & & k=3 \\
0 & 0 & 00 & 00 & 000 \\
1 & 1 & 01 & 01 & 001 \\
 & 1 & 11 & 11 & 011 \\
 & \to 0 \to & 10 & \to 10 \to & 010 \\
 & & & 10 & 110 \\
 & & & 11 & 111 \\
 & & & 01 & 101 \\
 & & & 00 & 100
\end{array}$$

计算方法

我们观察一下 n 的二进制和 $G(n)$。可以发现，如果 $G(n)$ 的二进制第 i 位为 1，仅当 n 的二进制第 i 位为 1，第 $i+1$ 位为 0 或者第 i 位为 0，第 $i+1$ 位为 1。于是我们可以当成一个异或的运算，即

$$G(n) = n \oplus \left\lfloor \frac{n}{2} \right\rfloor$$

```
int g(int n) { return n ^ (n >> 1); }
```

正确性证明

接下来我们证明一下，按照上述公式生成的格雷码序列，相邻两个格雷码的二进制位有且仅有一位不同。

我们考虑 n 和 $n+1$ 的区别。把 n 加 1，相当于把 n 的二进制下末位的连续的 1 全部变成取反，然后把最低位的 0 变成 1。我们这样表示 n 和 $n+1$ 的二进制位：

$$(n)_2 = \cdots 0\underbrace{11\cdots 11}_{k个}$$

$$(n+1)_2 = \cdots 1\underbrace{00\cdots 00}_{k个}$$

于是我们在计算 $g(n)$ 和 $g(n+1)$ 的时侯，后 k 位都会变成 $\underbrace{100\cdots 00}_{k个}$ 的形式，而第 $k+1$ 位是不同的，因为 n 和 $n+1$ 除了后 $k+1$ 位，其他位都是相同的。因此第 $k+1$ 位要么同时异或 1，要么同时异或 0。两种情况，第 $k+1$ 位都是不同的。而除了后 $k+1$ 位以外的二进制位也是做相同的异或运算，结果是相同的。

证毕。

通过格雷码构造原数（逆变换）

接下来我们考虑格雷码的逆变换，即给你一个格雷码 g，要求你找到原数 n。我们考虑从二进制最高位遍历到最低位（最低位下标为 1，即个位；最高位下标为 k）。则 n 的二进制第 i 位与 g 的二进制第 i 位 g_i 的关系如下：

$$\begin{aligned}
n_k &= g_k \\
n_{k-1} &= g_{k-1} \oplus n_k &&= g_k \oplus g_{k-1} \\
n_{k-2} &= g_{k-2} \oplus n_{k-1} &&= g_k \oplus g_{k-1} \oplus g_{k-2} \\
n_{k-3} &= g_{k-3} \oplus n_{k-2} &&= g_k \oplus g_{k-1} \oplus g_{k-2} \oplus g_{k-3} \\
&\vdots \\
n_{k-i} &= \bigoplus_{j=0}^{i} g_{k-j}
\end{aligned}$$

```
int rev_g(int g) {
  int n = 0;
  for (; g; g >>= 1) n ^= g;
  return n;
}
```

实际应用

格雷码有一些十分有用的应用，有些应用让人意想不到：

- k 位二进制数的格雷码序列可以当作 k 维空间中的一个超立方体（二维里的正方形，一维里的单位向量）顶点的哈密尔顿回路，其中格雷码的每一位代表一个维度的坐标。
- 格雷码被用于最小化数字模拟转换器（比如传感器）的信号传输中出现的错误，因为它每次只改变一个位。
- 格雷码可以用来解决汉诺塔的问题。

  设盘的数量为 n。我们从 n 位全 0 的格雷码 $G(0)$ 开始，依次移向下一个格雷码（$G(i)$ 移向 $G(i+1)$）。当前格雷码的二进制第 i 位表示从小到大第 i 个盘子。

  由于每一次只有一个二进制位会改变，因此当第 i 位改变时，我们移动第 i 个盘子。在移动盘子的过程中，除了最小的盘子，其他任意一个盘子在移动的时侯，只能有一个放置选择。在移动第一个盘子的时侯，我们总是有两个放置选择。于是我们的策略如下：

  如果 n 是一个奇数，那么盘子的移动路径为 $f \to t \to r \to f \to t \to r \to \cdots$，其中 f 是最开始的柱子，t 是最终我们把所有盘子放到的柱子，r 是中间的柱子。

  如果 n 是偶数：$f \to r \to t \to f \to r \to t \to \cdots$
- 格雷码也在遗传算法理论中得到应用。

习题

- CSP S2 2019 D1T1[1] Difficulty: easy
- SGU #249 Matrix[2] Difficulty: medium

> 本页面部分内容译自博文 [3] 与其英文翻译版 Gray code[4]。其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0。

参考资料与注释

[1] CSP S2 2019 D1T1

[2] SGU #249 Matrix

[3]

[4] Gray code

13.12 表达式求值

Authors: Ir1d, Anguei, hsfzLZH1, siger-young, HeRaNO, c8ef

表达式求值要解决的问题一般是输入一个字符串表示的表达式，要求输出它的值。当然也有变种比如表达式中是否包含括号，指数运算，含多少变量，判断多个表达式是否等价，等等。

表达式一般需要先进行语法分析（grammer parsing）再求值，也可以边分析边求值，语法分析的作用是检查输入的字符串是否是一个合法的表达式，一般使用语法分析器（parser）解决。

表达式包含两类字符：运算数和运算符。对于长度为 n 的表达式，借助合适的分析方法，可以在 $O(n)$ 的时间复杂度内完成分析与求值。

表达式树与逆波兰表达式

一种递归分析表达式的方法是，将表达式当成普通的语法规则进行分析，分析后拆分成如图所示的表达式树，然后在树结构上自底向上进行运算。

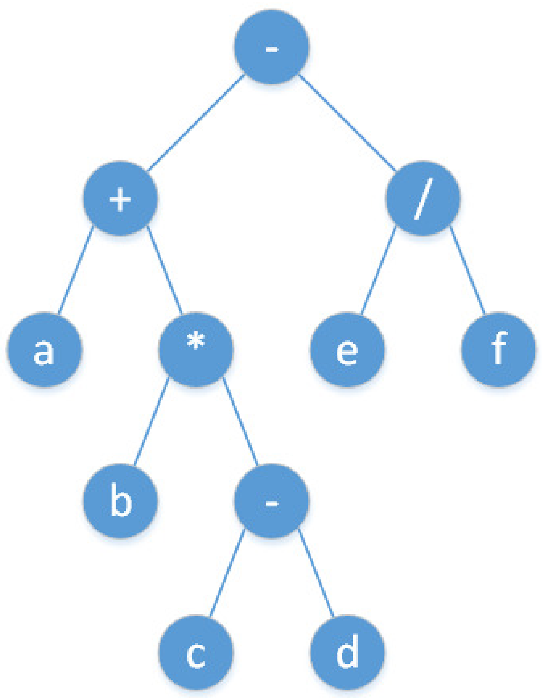


图 13.4

表达式树上进行 树的遍历 可以得到不同类型的表达式。算术表达式分为三种，分别是前缀表达式、中缀表达式、后缀表达式。中缀表达式是日常生活中最常用的表达式；后缀表达式是计算机容易理解的表达式。

- 前序遍历对应前缀表达式（波兰式）
- 中序遍历对应中缀表达式
- 后序遍历对应后缀表达式（逆波兰式）

逆波兰表达式（后缀表达式）是书写数学表达式的一种形式，其中运算符位于其操作数之后。例如，以下表达式：

$$a + b * c * d + (e - f) * (g * h + i)$$

可以用逆波兰表达式书写：

$$abc * d * +ef - gh * i + *+$$

因此，逆波兰表达式与表达式树一一对应。逆波兰表达式不需要括号表示，它的运算顺序是唯一确定的。

逆波兰表达式的方便之处在于很容易在线性时间内计算。举个例子：在逆波兰表达式 3 2 1 − 中，首先计算 $3 \times 2 = 6$（使用最后一个运算符，即栈顶运算符），然后计算 $6 - 1 = 5$。可以看到：对于一个逆波兰表达式，只需要

维护一个数字栈，每次遇到一个运算符，就取出两个栈顶元素，将运算结果重新压入栈中。最后，栈中唯一一个元素就是该逆波兰表达式的运算结果。该算法拥有 $O(n)$ 的时间复杂度。

采用递归的办法分析表达式是否成功，依赖于语法规则的设计是否合理，即，是否能够成功地得到指定的表达式树。例如：

$$a + b * c$$

根据加号与乘号的运算优先级不同，该中缀表达式可能转化为两种不同的表达式树。可见，语法规则的设计高度依赖于运算符的优先级。借助运算符的优先级设计相应递归的语法规则，事实上是一件不容易的事情。

下文介绍的办法将运算符与它的优先级视为一个整体，采用非递归的办法，直接根据运算符的优先级来分析与计算表达式。

只含左结合的二元运算符的含括号表达式

考虑简化的问题。假设所有运算符都是二元的：所有运算符都有两个参数。并且所有运算符都是左结合的：如果运算符的优先级相等，则从左到右执行。允许使用括号。

对于这种类型的中缀表达式的计算，可以将其转化为后缀表达式再进行计算。定义两个 栈 来分别存储运算符和运算数，每当遇到一个数直接放进运算数栈。每个运算符块对应于一对括号，运算符栈只对于运算符块的内部单调。每当遇到一个操作符时，要查找运算符栈中最顶部运算符块中的元素，在运算符块的内部保持运算符按照优先级降序进行适当的弹出操作，弹出的同时求出对应的子表达式的值。

以下部分用「输出」表示输出到后缀表达式，即将该数字放在运算数栈上，或者弹出运算符和两个操作数，运算后再将结果压回运算数栈上。从左到右扫描该中缀表达式：

1. 如果遇到数字，直接输出该数字。
2. 如果遇到左括号，那么将其放在运算符栈上。
3. 如果遇到右括号，不断输出栈顶元素，直至遇到左括号，左括号出栈。换句话说，执行一对括号内的所有运算符。
4. 如果遇到其他运算符，不断输出所有运算优先级大于等于当前运算符的运算符。最后，新的运算符入运算符栈。
5. 在处理完整个字符串之后，一些运算符可能仍然在堆栈中，因此把栈中剩下的符号依次输出，表达式转换结束。

以下是四个运算符 $+$、$-$、$*$、$/$ 的此方法的实现：

" 示例代码"

```cpp
bool delim(char c) { return c == ' '; }

bool is_op(char c) { return c == '+' || c == '-' || c == '*' || c == '/'; }

int priority(char op) {
  if (op == '+' || op == '-') return 1;
  if (op == '*' || op == '/') return 2;
  return -1;
}

void process_op(stack<int>& st, char op) {  // 也可以用于计算后缀表达式
  int r = st.top();                         // 取出栈顶元素，注意顺序
  st.pop();
  int l = st.top();
  st.pop();
  switch (op) {
    case '+':
      st.push(l + r);
      break;
```

```
    case '-':
      st.push(l - r);
      break;
    case '*':
      st.push(l * r);
      break;
    case '/':
      st.push(l / r);
      break;
  }
}

int evaluate(string& s) {  // 也可以改造为中缀表达式转换后缀表达式
  stack<int> st;
  stack<char> op;
  for (int i = 0; i < (int)s.size(); i++) {
    if (delim(s[i])) continue;

    if (s[i] == '(') {
      op.push('(');  // 2. 如果遇到左括号，那么将其放在运算符栈上
    } else if (s[i] == ')') {  // 3. 如果遇到右括号，执行一对括号内的所有运算符
      while (op.top() != '(') {
        process_op(st, op.top());
        op.pop();  // 不断输出栈顶元素，直至遇到左括号
      }
      op.pop();                 // 左括号出栈
    } else if (is_op(s[i])) {  // 4. 如果遇到其他运算符
      char cur_op = s[i];
      while (!op.empty() && priority(op.top()) >= priority(cur_op)) {
        process_op(st, op.top());
        op.pop();  // 不断输出所有运算优先级大于等于当前运算符的运算符
      }
      op.push(cur_op);  // 新的运算符入运算符栈
    } else {             // 1. 如果遇到数字，直接输出该数字
      int number = 0;
      while (i < (int)s.size() && isalnum(s[i]))
        number = number * 10 + s[i++] - '0';
      --i;
      st.push(number);
    }
  }

  while (!op.empty()) {
    process_op(st, op.top());
    op.pop();
  }
  return st.top();
}
```

这种隐式使用逆波兰表达式计算表达式的值的算法的时间复杂度为 $O(n)$。通过稍微修改上述实现，还可以以显式形式获得逆波兰表达式。

一元运算符与右结合的运算符

现在假设表达式还包含一元运算符，即只有一个参数的运算符。一元加号和一元减号是一元运算符的常见示例。

这种情况的一个区别是，需要确定当前运算符是一元运算符还是二元运算符。

注意到，在一元运算符之前一般有另一个运算符或开括号，如果一元运算符位于表达式的最开头则没有。在二元运算符之前，总是有一个运算数或右括号。因此，可以标记下一个运算符是否一元运算符。

此外，需要以不同的方式执行一元运算符和二元运算符，让一元运算符的优先级高于所有二元运算符。应注意，一些一元运算符，例如一元加号和一元减号，实际上是右结合的。

右结合意味着，每当优先级相等时，必须从右到左计算运算符。

如上所述，一元运算符通常是右结合的。右结合运算符的另一个示例是求幂运算符。对于 $a \wedge b \wedge c$，通常被视为 a^{b^c}，而不是 $(a^b)^c$。

为了正确地处理这类运算符，相应的改动是，如果优先级相等，将推迟运算符的出栈操作。

需要改动的代码如下。将：

```
while (!op.empty() && priority(op.top()) >= priority(cur_op))
```

换成

```
while (!op.empty() &&
       ((left_assoc(cur_op) && priority(op.top()) >= priority(cur_op)) ||
        (!left_assoc(cur_op) && priority(op.top()) > priority(cur_op))))
```

其中 left_assoc 是一个函数，它决定运算符是否为左结合的。

这里是二进制运算符 +、−、∗、/ 和一元运算符 + 和 − 的实现：

" 示例代码"

```
bool delim(char c) { return c == ' '; }

bool is_op(char c) { return c == '+' || c == '-' || c == '*' || c == '/'; }

bool is_unary(char c) { return c == '+' || c == '-'; }

int priority(char op) {
  if (op < 0)  // unary operator
    return 3;
  if (op == '+' || op == '-') return 1;
  if (op == '*' || op == '/') return 2;
  return -1;
}

void process_op(stack<int>& st, char op) {
  if (op < 0) {
    int l = st.top();
    st.pop();
    switch (-op) {
      case '+':
        st.push(l);
        break;
      case '-':
```

```
        st.push(-l);
        break;
      }
  } else {  // 取出栈顶元素，注意顺序
    int r = st.top();
    st.pop();
    int l = st.top();
    st.pop();
    switch (op) {
      case '+':
        st.push(l + r);
        break;
      case '-':
        st.push(l - r);
        break;
      case '*':
        st.push(l * r);
        break;
      case '/':
        st.push(l / r);
        break;
    }
  }
}

int evaluate(string& s) {
  stack<int> st;
  stack<char> op;
  bool may_be_unary = true;
  for (int i = 0; i < (int)s.size(); i++) {
    if (delim(s[i])) continue;

    if (s[i] == '(') {
      op.push('(');  // 2. 如果遇到左括号，那么将其放在运算符栈上
      may_be_unary = true;
    } else if (s[i] == ')') {  // 3. 如果遇到右括号，执行一对括号内的所有运算符
      while (op.top() != '(') {
        process_op(st, op.top());
        op.pop();  // 不断输出栈顶元素，直至遇到左括号
      }
      op.pop();  // 左括号出栈
      may_be_unary = false;
    } else if (is_op(s[i])) {  // 4. 如果遇到其他运算符
      char cur_op = s[i];
      if (may_be_unary && is_unary(cur_op)) cur_op = -cur_op;
      while (!op.empty() &&
             ((cur_op >= 0 && priority(op.top()) >= priority(cur_op)) ||
              (cur_op < 0 && priority(op.top()) > priority(cur_op)))) {
        process_op(st, op.top());
        op.pop();  // 不断输出所有运算优先级大于等于当前运算符的运算符
      }
      op.push(cur_op);  // 新的运算符入运算符栈
      may_be_unary = true;
    } else {  // 1. 如果遇到数字，直接输出该数字
```

```
      int number = 0;
      while (i < (int)s.size() && isalnum(s[i]))
        number = number * 10 + s[i++] - '0';
      --i;
      st.push(number);
      may_be_unary = false;
    }
  }

  while (!op.empty()) {
    process_op(st, op.top());
    op.pop();
  }
  return st.top();
}
```

参考资料

本页面主要译自博文 .[1] 与其英文翻译版 Expression parsing[2]。其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0。

延申阅读

1. Operator-precedence_parser[3]
2. Shunting yard algorithm[4]

习题

1. 表达式求值（NOIP2013）[5]
2. 后缀表达式[6]
3. Transform the Expression[7]

参考资料与注释

[1] .

[2] Expression parsing

[3] Operator-precedence_parser

[4] Shunting yard algorithm

[5] 表达式求值（NOIP2013）

[6] 后缀表达式

[7] Transform the Expression

13.13 在一台机器上规划任务

你有 n 个任务，要求你找到一个代价最小的的顺序执行他们。第 i 个任务花费的时间是 t_i，而第 i 个任务等待 t 的时间会花费 $f_i(t)$ 的代价。

形式化地说，给出 n 个函数 f_i 和 n 个数 t_i，求一个排列 p，最小化

$$F(p) = \sum_{i=1}^{n} f_{p_i}\left(\sum_{j=1}^{i-1} t_{p_j}\right)$$

特殊的代价函数

线性代价函数

首先我们考虑所有的函数是线性的函数，即 $f_i(x) = c_i x + d_i$，其中 c_i 是非负整数。显然我们可以事先把常数项加起来，因此函数就转化为了 $f_i(x) = c_i x$ 的形式。

考虑两个排列 p 和 p'，其中 p' 是把 p 的第 i 个位置上的数和 $i+1$ 个位置上的数交换得到的排列。则

$$\begin{aligned} F(p') - F(p) &= c_{p'_i} \sum_{j=1}^{i-1} t_{p'_j} + c_{p'_{i+1}} \sum_{j=1}^{i} t_{p'_j} - \left(c_{p_i} \sum_{j=1}^{i-1} t_{p_j} + c_{p_{i+1}} \sum_{j=1}^{i} t_{p_j}\right) \\ &= c_{p_i} t_{p_{i+1}} - c_{p_{i+1}} t_{p_i} \end{aligned}$$

于是我们使用如果 $c_{p_i} t_{p_{i+1}} - c_{p_{i+1}} t_{p_i} > 0$ 就交换的策略做一下排序就可以了。写成 $\dfrac{c_{p_i}}{t_{p_i}} > \dfrac{c_{p_{i+1}}}{t_{p_{i+1}}}$ 的形式，就可以理解为将排列按 $\dfrac{c_i}{t_i}$ 升序排序。

处理这个问题，我们的思路是考虑微扰后的变换情况，贪心地选取最优解。

指数代价函数

考虑代价函数的形式为 $f_i(x) = c_i \mathrm{e}^{ax}$，其中 $c_i \geq 0, a > 0$。

我们沿用之前的思路，考虑将 i 和 $i+1$ 的位置上的数交换引起的代价变化。最终得到的算法是将排列按照 $\dfrac{1-\mathrm{e}^{at_i}}{c_i}$ 升序排序。

相同的单增函数

我们考虑所有的 $f_i(x)$ 是同一个单增函数。那么显然我们将排列按照 t_i 升序排序即可。

Livshits–Kladov 定理

Livshits–Kladov 定理成立，当且仅当代价函数是以下三种情况：

- 线性函数：$f_i(t) = c_i t + d_i$，其中 $c_i \geq 0$；
- 指数函数：$f_i(t) = c_i \mathrm{e}^{at} + d_i$，其中 $c_i, a > 0$；
- 相同的单增函数：$f_i(t) = \phi(t)$，其中 $\phi(t)$ 是一个单增函数。

定理是在假设代价函数足够平滑（存在三阶导数）的条件下证明的。在这三种情况下，问题的最优解可以通过简单的排序在 $O(n \log n)$ 的时间内解决。

参考资料与注释

[1]

[2] Scheduling jobs on one machine

13.14　主元素问题

Authors: SDLTF

概述

给一个有 n 个元素的数列，保证有一个数 a 出现的次数**超过** $\dfrac{n}{2}$，求这个数。

做法

桶计数做法

桶计数做法是出现一个数，就把这个数出现次数 $+1$，很好懂：

```
for (int i = 0; i < n; i++) {
  cin >> t;
  ans[t]++;
}
for (int i = 0; i < m; i++) {  // m 为桶的大小
  if (ans[i] > n / 2) {
    cout << i;
    break;
  }
}
```

时间复杂度 $O(n+m)$。

但是这个做法很浪费空间，我们不推荐使用。

排序做法

显然，若一个数列存在主元素，那么这个主元素在排序后一定位于 $\dfrac{n}{2}$ 的位置。

那么我们又有想法了：

```
sort(a, a + n);
cout << a[n / 2 - 1];  // 因为这里数组从 0 开始使用，所以需要 -1
```

看起来不错！$O(n\log n)$ 的复杂度可还行？

下面介绍本问题的 $O(n)$ 解法。

主元素数列的特性

由于主元素的出现的次数超过 $\dfrac{n}{2}$，那么在不断的消掉两个不同的元素之后，最后一定剩下主元素。

输入时判断与上一次保存的输入是否相同，如不同则删除两数，这里用栈来实现。

```
while (n--) {
  scanf("%d", &a);
  q[top++] = a;
  top = (top > 1 && (q[top - 1] != q[top - 2])) ? (top - 2) : top;
}
printf("%d", q[top - 1]);
```

再进行优化后，空间复杂度也可以降至 $O(1)$。

```
int val = -1, cnt = 0;
while (n--) {
  scanf("%d", &a);
  if (a != val) {
    if (--cnt <= 0) {
      val = a, cnt = 1;
    }
  } else {
    ++cnt;
  }
}
```

13.15 Garsia–Wachs 算法

简介

Garsia–Wachs 算法（Garsia–Wachs Algorithm）是计算机用来在**线性时间**内构建**最优二叉查找树**和**字母霍夫曼码**的有效算法。它以 Adriano Garsia 和 Michelle L. Wachs 的名字命名，他们于 1977 年发表了相关论文。

问题描述

一个整数 n，对于 $n+1$ 个非负权值 $w_0, w_1, \dots, w_n$，构造一棵有根且 n 个内部节点都有两个子节点的二叉树，这意味着这棵二叉树有 $n+1$ 个叶节点。我们将 $n+1$ 的输入序列与二叉树结点顺序一一映射，目标是在所有具有 n 个内部节点的可能的树结构中找到一棵树，使外部从根到每个叶子的路径长度的权重和最小。

最优二叉查找树

这个问题可以理解为 n 个有序键构造二叉查找树的问题，假设树将仅用于搜索树中不存在的值。在这种情况下，n 个键将搜索值的空间划分为 $n+1$ 个区间，并且这些区间之一的权重可以作为搜索值落在那个区间的概率。外部路径长度的加权和控制了查找的预期时间。

字母霍夫曼码

这个问题也可以用作构建霍夫曼码。这是一种通过使用二进制值的可变长度序列明确编码 $n+1$ 给定值的方法。在这种解释中，值的代码由从树中的根到叶子的路径上从父到子的左步和右步序列给出（例如，左为 0，右为 1）。与标准霍夫曼码不同，以这种方式构造的霍夫曼码是按字母顺序排列的，也就是说这些二进制码的排序顺序与值的输入顺序相同。如果一个值的权重是它在编码消息中的频率，那么 Garsia–Wachs 算法的输出是将消息长度压缩到最短的，按字母顺序排列的霍夫曼代码。

过程

Garsia–Wachs 算法一般包括三个阶段：

1. 构建一个值位于叶子的二叉树，注意顺序可能错误。
2. 计算树中根到每个叶子的距离。
3. 构建另一个二叉树，叶子的距离相同，但顺序正确。

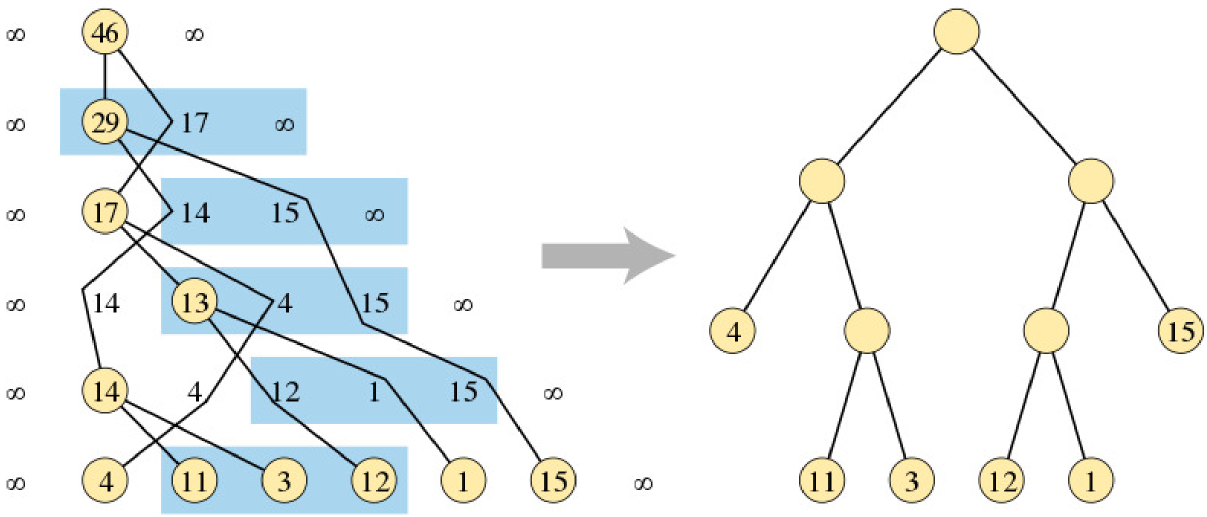


图 13.5

如上图所示，在算法的第一阶段，通过查找合并输入序列的无序三元组构建的二叉树（左侧），和算法输出的正确排序的二叉树，其中叶子高度与另一棵树一样。

如果输入在序列的开始和结束处增加了两个标记值 ∞（或任何足够大的有限值），则算法的第一阶段更容易描述。所以在竞赛题解中使用 Garsia–Wachs 算法时，对于一个长度为 n 的数组 num，我们一般定义 $num[0] = num[n+1] = \infty$。

第一阶段维护了一个由最初为每个非标志（non-sentinel）输入权重创建的单节点树组成的森林。每棵树都与一个值相关联，其叶子的权重之和为每个非标志输入权重构成一个树节点。为了维护这些值的序列，每端会有两个标记值。初始序列只是叶权重作为输入的顺序。然后重复执行以下步骤，每一步都减少输入序列的长度，直到只有一棵树包含了所有叶子：

- 在序列中找到前三个连续的权重值 x，y，z 使得 $x \leq z$。因为序列结尾的标志值大于之前的任意两个有限值，所以总是存在这样的三元组。
- 从序列中移除 x 和 y，并创建一个新的树节点作为 x 和 y 节点的父节点，值为 $x + y$。
- 在原来 x 的位置以前大于或等于 $x + y$ 且距 x 最近的值的右边重新插入新节点。因为左标志值的存在，所以总是存在这样的位置。

为了有效地实现这一阶段，该算法可以在任何平衡二叉查找树结构中维护当前值序列。这样的结构允许我们在对数时间内移除 x 和 y，并重新插入它们的新父节点。在每一步中，数组中位于偶数索引上直到 y 值的权重形成了一个递减序列，位于奇数索引位的权重形成另一个递减序列。因此，重新插入 $x + y$ 的位置可以通过在对数时间内对这两个递减序列使用平衡树执行两次二分查找找到。通过从前一个三元组 z 值开始的线性顺序搜索，我们可以在总线性时间复杂度内执行对满足 $x \leq z$ 的第一个位置的搜索。

Garsia–Wachs 算法的第三阶段的证明，即存在另一棵具有相同距离的树并且这棵树提供了问题的最优解，是很重要的。但是由于其证明方式有多种且过于复杂，此处略去。在第三阶段为正确的前提下，第二和第三阶段很容易在线性时间内实现。因此，在长度为 n 的输入序列上，Garsia–Wachs 算法的总时间复杂度为 $O(n \log n)$。

应用

函数性编程语言 Haskell 的 garsia-wachs package[1] 对 Garsia–Wachs 算法做了函数性实现。它主要用于构建最佳搜索表，或者以最优复杂度平衡 rope[2] 数据结构。

"注释"

rope 是 Haskell 语言中用于操作带有可选注释的字节串（bytestring）手指树 的工具。

例题

"POJ 1738 An old Stone Game[3]"

有一个古老的石头游戏。在游戏开始时，玩家将 $n(1 \leq n \leq 50000)$ 堆石头排成一行。目标是将石头合并成一堆，规则如下：在游戏的每一步，玩家可以将相邻的两个堆合并成一个新的堆。分数是新堆的石头总数。请计算总分中的最小值。

"解题思路"

石子合并的题目很经典，一般我们可以用区间 DP 解答，但是当数据量很大，例如此题中的 $n(1 \leq n \leq 50000)$ 时，用 Garsia–Wachs 算法求解更高效：第一步，初始化一个大小为 n 的数组 $num[n]$，其中 $num[0] = num[n+1] = \infty$。第二步，每次找到一个最小的 i 使 $num[i-1] \leq num[i+1]$，并将 $num[i-1], num[i]$ 合并为 $temp$；找到前面一个最大的 j 使得 $num[j] > temp$，将 $temp$ 移到 j 后面。重复这一步直到剩余堆数为 1。关于每次只能合并相邻石子堆的要求，因为 $num[j] \geq num[i-1] + num[i]$，我们可以将 $num[j+1]$ 到 $num[i-2]$ 看成一个 $num[mid]$ 的整体，所以一定是先合并 sum。因此没有违背题目要求。

"ATCODER N-Slimes[4]"

N 个史莱姆排成一排。最初左边第 i 个史莱姆的大小为 a_i。Taro 试图将所有史莱姆组合成一个更大的史莱姆。他会反复执行以下操作，直到只有一个史莱姆：选择两个相邻的史莱姆，并将它们组合成一个新的史莱姆。新的史莱姆的大小为 $x + y$，其中 x 和 y 是组合之前史莱姆的大小。这一步骤有产生 $x + y$ 的成本。合成史莱姆时史莱姆的位置关系不会改变。找出可能发生的最小总成本。

参考资料与拓展阅读

1. Garsia–Wachs algorithm - Wikipedia[5]
2. Data.Algorithm.GarsiaWachs - Hackage Haskell[6]
3. garsia-wachs: A Functional Implementation of the Garsia-Wachs Algorithm[1]
4. Sentinel value - Wikipedia[7]
5. A new proof of the Garsia-Wachs algorithm[8]

参考资料与注释

[1] garsia-wachs package [1-1] [1-2]

[2] rope

[3] POJ 1738 An old Stone Game

[4] ATCODER N-Slimes

[5] Garsia–Wachs algorithm - Wikipedia

[6] Data.Algorithm.GarsiaWachs - Hackage Haskell

[7] Sentinel value - Wikipedia

[8] A new proof of the Garsia-Wachs algorithm

13.16 15-puzzle

简介

15 - 拼图（英文：15-puzzle, 又名 Gem Puzzle，Boss Puzzle，Game of 15，Mystic Square，N-puzzle, etc）是一个滑块类游戏（英文：sliding puzzle）。滑块方盘的长宽均为 4×4 个方块，其中 15 个位置放序号打乱的方块，剩下一个为空位。与空位同行或同列的方块可以通过水平或垂直滑动来移动。拼图的目标是按编号顺序排列方块。

15 - 拼图常见别称为 **n - 拼图**，其中数字 n 指的是方盘中的方块总数。15 - 拼图的不同尺寸变体亦使用了类似的名称，例如 8 拼图指的是置于 3×3 方盘中的 8 个方块。但 15 拼图也可以称为 16 拼图，此处的 16 指的是方块容量。它的扩展问题有时也包括了 $n \times m$ 的滑动方盘。

15 - 拼图是涉及 启发式算法 建模的经典问题。此问题的常见形式是 曼哈顿距离 和错位方块的数量计算，二者都是可接受启发（英文：admissible heuristic），即它们永远不会高估剩余的移动次数，这确保了某些搜索算法（例如 A * 算法）的最优性。

”注释”

滑块游戏是一类在平面上滑动方块以组成特定排列的智力游戏。常见的滑块游戏包括数字拼图、华容道和塞车时间。其中 15 - 拼图是最古老的滑块类游戏，发明者是 Noyes Chapman，该游戏风靡于 1880 年代。不像其它 tour 类的解谜游戏，滑块游戏禁止任何一个方块离开盘面，这个特性区别于重新排列类的解谜游戏。

定义

给定一个 4×4 的方盘，其中 15 个方块随意排列。我们需要将它按照序号排列成下图所示的样子。移动规则为每次只能交换空方块和和其相邻一个方块的位置。常见问题为找到可解决此问题的最少步骤，计算错位方位的个数，或找出是和否能得到最终的有序排列。

□

可解性证明

Johnson & Story (1879) 证明，如果 m 和 n 都至少为 2，则逆向适用于大小为 $m \times n$ 的棋盘：通过从 $m = n = 2$ 开始对 m 和 n 进行归纳证明，所有偶数排列都是可解的。Archer (1999) 给出了另一个证明，基于通过汉密尔顿路径定义等价类。

算法

寻找数字滑盘游戏的一个解相对容易，但寻找**最优解**是一个 **NP 困难**问题。15-Puzzle 的最优解至多有 80 步；而 8-Puzzle 的最优解至多有 31 步。

N-Puzzle 支持常见的基于图的搜索算法，如广度优先搜索和深度优先搜索，同样我们也可以用 A * 搜索 算法寻找最优解。启发式函数 $h(n)$ 可以是

- 放错的方块的数量。
- 所有放错的方块到各自目标位置的欧几里得距离之和。
- 所有放错的方块到各自目标位置的曼哈顿距离之和。

群理论

因为 15 块的数字推盘游戏组合可以由「3 循环」（英文：3-cycles）产生，所以可以证明 15 块的数字推盘游戏可以用交错群 A_{15} 表示。事实上，任何使用 $2 \times k - 1$ 块相同面积正方形方块的数字滑盘游戏皆可以以交错群 A_{2k-1} 表示。

习题

- N Puzzle[1]
- A. Amity Assessment[2]
- Sliding Puzzle[3]
- POJ 1077 - Eight[4]

参考资料与拓展阅读

1. 15 puzzle - Wikipedia[5]
2. jrdnjacobson,How to Solve the 15 Puzzle - instructables[6]
3. Korf, R. E. (2000),”Recent Progress in the Design and Analysis of Admissible Heuristic Functions”[7], in Choueiry, B. Y.; Walsh, T. (eds.), Abstraction, Reformulation, and Approximation (PDF), SARA 2000. Lecture Notes in Computer Science, vol. 1864, Springer, Berlin, Heidelberg, pp. 45–55, doi:10.1007/3-540-44914-0_3, ISBN 978-3-540-67839-7, retrieved 2010-04-26
4. Welcome to N-Puzzle - web demo[8]

参考资料与注释

[1] N Puzzle

[2] A. Amity Assessment

[3] Sliding Puzzle

[4] POJ 1077 - Eight

[5] 15 puzzle - Wikipedia

[6] How to Solve the 15 Puzzle - instructables

[7] ”Recent Progress in the Design and Analysis of Admissible Heuristic Functions”

[8] Welcome to N-Puzzle - web demo

13.17 Kahan 求和

引入

Kahan 求和算法，又名补偿求和或进位求和算法，是一个用来**降低有限精度浮点数序列累加值误差**的算法。它主要通过保持一个单独变量用来累积误差（常用变量名为 c）来完成的。

该算法主要由 William Kahan 于 1960s 发现。因为 Ivo Babuška 也曾独立提出了一个类似的算法，Kahan 求和算法又名为 Kahan–Babuška 求和算法。

舍入误差

在计算机程序中，我们需要用有限位数对实数做近似表示，如今的大多数计算机都使用 IEEE-754[1] 规定的浮点数来作为这个近似表示。对于 $\frac{1}{3}$，由于我们不能在有限位数内对它进行精准表示，因此在使用 IEEE-754 表示法时，必须四舍五入一部分数值（truncate）。这种**舍入误差**（Rounding off error）是浮点计算的一个特征。

在浮点加法计算中，交换律（commutativity）成立，但结合律（associativity）不成立。也就是说，$a+b=b+a$ 但 $(a+b)+c \neq a+(b+c)$。因此在浮点序列加法计算中，我们可以从左到右一个个累加，也可以在原有顺序上，将他们两两分成一对。第二种算法会相对较慢并需要更多内存，也常被一些语言的特定求和函数使用，但相对结果更准确。

为了得到更准确的浮点累加结果，我们需要使用 Kahan 求和算法。

在计算 $S_{new} = S_{old} + a$（a 为浮点序列的一个数值）时，定义实际计算加入 S 的值为 $a_{eff} = S_{new} - S_{old}$，如果 a_{eff} 比 a 大，则证明有向上舍入误差；如果 a_{eff} 比 a 小，则证明有向下舍入误差。则舍入误差定义为 $E_{roundoff} = a_{eff} - a$。那么用来纠正这部分舍入误差的值就为 $a - a_{eff}$，即 $E_{roundoff}$ 的负值。定义 c 是对丢失的低位进行运算补偿的变量，就可以得到 $c_{new} = c_{old} + (a - a_{eff})$。

过程

Kahan 求和算法主要通过一个单独变量用来累积误差。如下方参考代码所示，sum 为最终返回的累加结果。c 是对丢失的低位进行运算补偿的变量（其被舍去的部分），也是 Kahan 求和算法中的必要变量。

因为 sum 大，y 小，所以 y 的低位数丢失。$(t - sum)$ 抵消了 y 的高阶部分，减去 y 则会恢复负值（y 的低价部分）。因此代数值中 c 始终为零。在下一轮迭代中，丢失的低位部分会被更新添加到 y。

实现

参考代码

```
float kahanSum(vector<float> nums) {
  float sum = 0.0f;
  float c = 0.0f;
  for (auto num : nums) {
    float y = num - c;
    float t = sum + y;
    c = (t - sum) - y;
    sum = t;
  }
  return sum;
}
```

习题

在 OI 中，Kahan 求和主要作为辅助工具存在，为计算结果提供误差更小的值。

" **例题** CodeForces Contest 800 Problem A. Voltage Keepsake[2]"

有 n 个同时使用的设备。第 i 个设备每秒使用 a_i 单位的功率。这种用法是连续的。也就是说，在 λ 秒内，设备将使用 $\lambda \times a_i$ 单位的功率。第 i 个设备当前存储了 b_i 单位的电力。所有设备都可以存储任意数量的电量。有一个可以插入任何单个设备的充电器。充电器每秒会为设备增加 p 个单位的电量。这种充电是连续的。也就是说，如果将设备插入 λ 秒，它将获得 $\lambda \times p$ 单位的功率。我们可以在任意时间单位内（包括实数）切换哪个设备正在充电（切换所需时间忽略不计）。求其中一个设备达到 0 单位功率前，可以使用这些设备的最长时间。

" **例题** CodeForces Contest 504 Problem B. Misha and Permutations Summation[3]"

定义数字 $0, 1, \cdots, (n-1)$ 的两个排列 p 和 q 的和为 $Perm((Ord(p) + Ord(q)) \bmod n!)$，其中 $Perm(x)$ 是数字 $0, 1, \cdots, (n-1)$ 的第 x 个字典排列（从零开始计数），$Ord(p)$ 是字典序排列 p 的个数。例如，$Perm(0) = (0, 1, \cdots, n-2, n-1)$，$Perm(n!-1) = (n-1, n-2, \cdots, 1, 0))$。Misha 有两个排列 p 和 q，找到它们的总和。

编程语言的求和

Python 的标准库指定了精确舍入求和的 fsum[4] 函数可用于返回可迭代对象中值的准确浮点总和，它通过使用 Shewchuk 算法跟踪多个中间部分和来避免精度损失。

Julia 语言中，sum[5] 函数的默认实现是成对求和，以获得高精度和良好的性能。同时外部库函数 sum_kbn[6] 为需要更高精度的情况提供了 Neumaier 变体的实现，具体可见 KahanSummation.jl[7]。

参考资料与注释

1. Kahan_summation_algorithm - Wikipedia[8]
2. Kahan summation - Rosetta Code[9]
3. VK Cup Round 2 + Codeforces Round 409 Announcement[10]
4. Rounding off errors in Java - GeeksforGeeks[11]

参考资料与注释

[1] IEEE-754

[2] CodeForces Contest 800 Problem A. Voltage Keepsake

[3] CodeForces Contest 504 Problem B. Misha and Permutations Summation

[4] fsum

[5] sum

[6] sum_kbn

[7] KahanSummation.jl

[8] Kahan_summation_algorithm - Wikipedia

[9] Kahan summation - Rosetta Code

[10] VK Cup Round 2 + Codeforces Round 409 Announcement

[11] Rounding off errors in Java - GeeksforGeeks

13.18 珂朵莉树/颜色段均摊

名称简介

珂朵莉树（Chtholly Tree），又名老司机树 ODT（Old Driver Tree）。起源自 CF896C[1]。

注意，这种想法的本质是基于数据随机的「颜色段均摊」，而不是一种数据结构，下文介绍的操作是这种想法的具体实现方法。

前置知识

会用 STL 的 set 就行。

核心思想

把值相同的区间合并成一个结点保存在 set 里面。

用处

骗分。只要是有区间赋值操作的数据结构题都可以用来骗分。在数据随机的情况下一般效率较高，但在不保证数据随机的场合下，会被精心构造的特殊数据卡到超时。

如果要保证复杂度正确，必须保证数据随机。详见 Codeforces 上关于珂朵莉树的复杂度的证明[2]。

更详细的严格证明见 珂朵莉树的复杂度分析[3]。对于 add，assign 和 sum 操作，用 set 实现的珂朵莉树的复杂度为 $O(n\log\log n)$，而用链表实现的复杂度为 $O(n\log n)$。

正文

首先，结点的保存方式：

```
struct Node_t {
  int l, r;
  mutable int v;

  Node_t(const int &il, const int &ir, const int &iv) : l(il), r(ir), v(iv) {}

  bool operator<(const Node_t &o) const { return l < o.l; }
};
```

其中，`int v` 是你自己指定的附加数据。

"mutable 关键字的含义是什么？"

mutable 的意思是「可变的」，让我们可以在后面的操作中修改 v 的值。在 C++ 中，mutable 是为了突破 const 的限制而设置的。被 mutable 修饰的变量（mutable 只能用于修饰类中的非静态数据成员），将永远处于可变的状态，即使在一个 const 函数中。

这意味着，我们可以直接修改已经插入 set 的元素的 v 值，而不用将该元素取出后重新加入 set。

然后，我们定义一个 `set<Node_t> odt;` 来维护这些结点。为简化代码，可以 `typedef set<Node_t>::iterator iter`，当然在题目支持 C++11 时也可以使用 `auto`。

split

`split` 是最核心的操作之一，它用于将原本包含点 x 的区间（设为 $[l,r]$）分裂为两个区间 $[l,x)$ 和 $[x,r]$ 并返回指向后者的迭代器。参考代码如下：

```
auto split(int x) {
  if (x > n) return odt.end();
  auto it = --odt.upper_bound(Node_t{x, 0, 0});
  if (it->l == x) return it;
  int l = it->l, r = it->r, v = it->v;
  odt.erase(it);
  odt.insert(Node_t(l, x - 1, v));
  return odt.insert(Node_t(x, r, v)).first;
}
```

这段代码有什么用呢？任何对于 $[l,r]$ 的区间操作，都可以转换成 set 上 $[split(l), split(r+1))$ 的操作。

assign

另外一个重要的操作 `assign` 用于对一段区间进行赋值。对于 ODT 来说，区间操作只有这个比较特殊，也是保证复杂度的关键。如果 ODT 里全是长度为 1 的区间，就成了暴力，但是有了 `assign`，可以使 ODT 的大小下降。参考代码如下：

```
void assign(int l, int r, int v) {
  auto itr = split(r + 1), itl = split(l);
  odt.erase(itl, itr);
  odt.insert(Node_t(l, r, v));
}
```

其他操作

套模板就好了，参考代码如下：

```
void performance(int l, int r) {
  auto itr = split(r + 1), itl = split(l);
  for (; itl != itr; ++itl) {
    // Perform Operations here
  }
}
```

注：**珂朵莉树在进行求取区间左右端点操作时，必须先 split 右端点，再 split 左端点**。若先 split 左端点，返回的迭代器可能在 split 右端点的时候失效，可能会导致 RE。

习题

- ~~\hyref{https://www.luogu.com.cn/problem/P2572}{「SCOI2010」序列操作}\textsuperscript{\label{endnoteref:OIwikidocsmiscodtOutLink_4-1}\hyperref[endnote:OIwikidocsmiscodtOutLink_4]{[4]}}~~（该题目来源已添加 Hack 数据）
- 「SHOI2015」 脑洞治疗仪[5]
- 「Luogu 4979」 矿洞 ： 坍塌[6]
- 「Luogu 8146」risrqnis[7]

参考资料与注释

[1] CF896C

[2] Codeforces 上关于珂朵莉树的复杂度的证明

[3] 珂朵莉树的复杂度分析

[4] 「SCOI2010」序列操作

[5] 「SHOI2015」脑洞治疗仪

[6] 「Luogu 4979」矿洞：坍塌

[7] 「Luogu 8146」risrqnis

第 14 章

专题

14.1 RMQ

简介

RMQ 是英文 Range Maximum/Minimum Query 的缩写，表示区间最大（最小）值。

在接下来的描述中，默认初始数组大小为 n，询问次数为 m。

在接下来的描述中，默认时间复杂度标记方式为 $O(A) \sim O(B)$，其中 $O(A)$ 表示预处理时间复杂度，而 $O(B)$ 表示单次询问的时间复杂度。

单调栈

由于 **OI Wiki** 中已有此部分的描述，本文仅给出 链接。这部分不再展开。

时间复杂度 $O(m \log m) \sim O(\log n)$，空间复杂度 $O(n)$。

ST 表

由于 **OI Wiki** 中已有此部分的描述，本文仅给出 链接。这部分不再展开。

时间复杂度 $O(n \log n) \sim O(1)$，空间复杂度 $O(n \log n)$。

线段树

由于 **OI Wiki** 中已有此部分的描述，本文仅给出 链接。这部分不再展开。

时间复杂度 $O(n) \sim O(\log n)$，空间复杂度 $O(n)$。

Four Russian

Four russian 是一个由四位俄罗斯籍的计算机科学家提出来的基于 ST 表的算法。

在 ST 表的基础上 Four russian 算法对其做出的改进是序列分块。

具体来说，我们将原数组——我们将其称之为数组 A——每 S 个分成一块，总共 n/S 块。

对于每一块我们预处理出来块内元素的最小值，建立一个长度为 n/S 的数组 B，并对数组 B 采用 ST 表的方式预处理。

同时，我们对于数组 A 的每一个零散块也建立一个 ST 表。

询问的时候，我们可以将询问区间划分为不超过 1 个数组 B 上的连续块区间和不超过 2 个数组 A 上的整块内的连续区间。显然这些问题我们通过 ST 表上的区间查询解决。

在 $S = \log n$ 时候，预处理复杂度达到最优，为 $O((n/\log n)\log n + (n/\log n) \times \log n \times \log\log n) = O(n\log\log n)$。

时间复杂度 $O(n\log\log n) \sim O(1)$，空间复杂度 $O(n\log\log n)$。

当然询问由于要跑三个 ST 表，该实现方法的常数较大。

" 一些小小的算法改进"

我们发现，在询问的两个端点在数组 A 中属于不同的块的时候，数组 A 中块内的询问是关于每一块前缀或者后缀的询问。

显然这些询问可以通过预处理答案在 $O(n)$ 的时间复杂度内被解决。

这样子我们只需要在询问的时候进行至多一次 ST 表上的查询操作了。

" 一些玄学的算法改进"

由于 Four russian 算法以 ST 表为基础，而算法竞赛一般没有非常高的时间复杂度要求，所以 Four russian 算法一般都可以被 ST 表代替，在算法竞赛中并不实用。这里提供一种在算法竞赛中更加实用的 Four russian 改进算法。

我们将块大小设为 $\sqrt{n}$，然后预处理出每一块内前缀和后缀的 RMQ，再暴力预处理出任意连续的整块之间的 RMQ，时间复杂度为 $O(n)$。

查询时，对于左右端点不在同一块内的询问，我们可以直接 $O(1)$ 得到左端点所在块的后缀 RMQ，左端点和右端点之间的连续整块 RMQ，和右端点所在块的前缀 RMQ，答案即为三者之间的最值。

而对于左右端点在同一块内的询问，我们可以暴力求出两点之间的 RMQ，时间复杂度为 $O(\sqrt{n})$，但是单个询问的左右端点在同一块内的期望为 $O(\frac{\sqrt{n}}{n})$，所以这种方法的时间复杂度为期望 $O(n)$。

而在算法竞赛中，我们并不用非常担心出题人卡掉这种算法，因为我们可以通过在 $\sqrt{n}$ 的基础上随机微调块大小，很大程度上避免算法在根据特定块大小构造的数据中出现最坏情况。并且如果出题人想要卡掉这种方法，则暴力有可能可以通过。

这是一种期望时间复杂度达到下界，并且代码实现难度和算法常数均较小的算法，因此在算法竞赛中比较实用。

以上做法参考了 P3793 由乃救爷爷[1] 中的题解。

加减 1RMQ

若序列满足相邻两元素相差为 1，在这个序列上做 RMQ 可以成为加减 1RMQ，根究这个特性可以改进 Four Russian 算法，做到 $O(n) \sim O(1)$ 的时间复杂度，$O(n)$ 的空间复杂度。

由于 Four russian 算法的瓶颈在于块内 RMQ 问题，我们重点去讨论块内 RMQ 问题的优化。

由于相邻两个数字的差值为 ± 1，所以在固定左端点数字时长度不超过 $\log n$ 的右侧序列种类数为 $\sum_{i=1}^{\log n} 2^{i-1}$，而这个式子显然不超过 n。

这启示我们可以预处理所有不超过 n 种情况的最小值 - 第一个元素的值。

在预处理的时候我们需要去预处理同一块内相邻两个数字之间的差，并且使用二进制将其表示出来。

在询问的时候我们找到询问区间对应的二进制表示，查表得出答案。

这样子 Four russian 预处理的时间复杂度就被优化到了 $O(n)$。

笛卡尔树在 RMQ 上的应用

不了解笛卡尔树的朋友请移步 笛卡尔树。

不难发现，原序列上两个点之间的 min/max，等于笛卡尔树上两个点的 LCA 的权值。根据这一点就可以借助 $O(n) \sim O(1)$ 求解树上两个点之间的 LCA 进而求解 RMQ。$O(n) \sim O(1)$ 树上 LCA 在 LCA - 标准 RMQ 已经有描

述，这里不再展开。

总结一下，笛卡尔树在 RMQ 上的应用，就是通过将普通 RMQ 问题转化为 LCA 问题，进而转化为加减 1 RMQ 问题进行求解，时间复杂度为 $O(n) \sim O(1)$。当然由于转化步数较多，$O(n) \sim O(1)$ RMQ 常数较大。

如果数据随机，还可以暴力在笛卡尔树上查找。此时的时间复杂度为期望 $O(n) \sim O(\log n)$，并且实际使用时这种算法的常数往往很小。

例题 Luogu P3865【模板】ST 表[2]

基于状压的线性 RMQ 算法

隐性要求

- 序列的长度 n 满足 $\log_2 n \leq 64$。

前置知识

- Sparse Table
- 基本位运算
- 前后缀极值

算法原理

将原序列 $A[1\cdots n]$ 分成每块长度为 $O(\log_2 n)$ 的 $O(\frac{n}{\log_2 n})$ 块。

> 听说令块长为 $1.5 \times \log_2 n$ 时常数较小。

记录每块的最大值，并用 ST 表维护块间最大值，复杂度 $O(n)$。

记录块中每个位置的前、后缀最大值 $Pre[1\cdots n], Sub[1\cdots n]$（$Pre[i]$ 即 $A[i]$ 到其所在块的块首的最大值），复杂度 $O(n)$。

若查询的 l, r 在两个不同块上，分别记为第 bl, br 块，则最大值为 $[bl+1, br-1]$ 块间的最大值，以及 $Sub[l]$ 和 $Pre[r]$ 这三个数的较大值。

现在的问题在于若 l, r 在同一块中怎么办。

将 $A[1\cdots r]$ 依次插入单调栈中，记录下标和值，满足值从栈底到栈顶递减，则 $A[l, r]$ 中的最大值为从栈底往上，单调栈中第一个满足其下标 $p \geq l$ 的值。

由于 $A[p]$ 是 $A[l, r]$ 中的最大值，因而在插入 $A[p]$ 时，$A[l\cdots p-1]$ 都被弹出，且在插入 $A[p+1\cdots r]$ 时不可能将 $A[p]$ 弹出。

而如果用 0/1 表示每个数是否在栈中，就可以用整数状压，则 p 为第 l 位后的第一个 1 的位置。

由于块大小为 $O(\log_2 n)$，因而最多不超过 64 位，可以用一个整数存下（即隐性条件的原因）。

" 参考代码"

```cpp
#include <bits/stdc++.h>

const int MAXN = 1e5 + 5;
const int MAXM = 20;

struct RMQ {
  int N, A[MAXN];
```

```cpp
int blockSize;
int S[MAXN][MAXM], Pow[MAXM], Log[MAXN];
int Belong[MAXN], Pos[MAXN];
int Pre[MAXN], Sub[MAXN];
int F[MAXN];

void buildST() {
  int cur = 0, id = 1;
  Pos[0] = -1;
  for (int i = 1; i <= N; ++i) {
    S[id][0] = std::max(S[id][0], A[i]);
    Belong[i] = id;
    if (Belong[i - 1] != Belong[i])
      Pos[i] = 0;
    else
      Pos[i] = Pos[i - 1] + 1;
    if (++cur == blockSize) {
      cur = 0;
      ++id;
    }
  }
  if (N % blockSize == 0) --id;
  Pow[0] = 1;
  for (int i = 1; i < MAXM; ++i) Pow[i] = Pow[i - 1] * 2;
  for (int i = 2; i <= id; ++i) Log[i] = Log[i / 2] + 1;
  for (int i = 1; i <= Log[id]; ++i) {
    for (int j = 1; j + Pow[i] - 1 <= id; ++j) {
      S[j][i] = std::max(S[j][i - 1], S[j + Pow[i - 1]][i - 1]);
    }
  }
}

void buildSubPre() {
  for (int i = 1; i <= N; ++i) {
    if (Belong[i] != Belong[i - 1])
      Pre[i] = A[i];
    else
      Pre[i] = std::max(Pre[i - 1], A[i]);
  }
  for (int i = N; i >= 1; --i) {
    if (Belong[i] != Belong[i + 1])
      Sub[i] = A[i];
    else
      Sub[i] = std::max(Sub[i + 1], A[i]);
  }
}

void buildBlock() {
  static int S[MAXN], top;
  for (int i = 1; i <= N; ++i) {
    if (Belong[i] != Belong[i - 1])
      top = 0;
    else
      F[i] = F[i - 1];
```

```cpp
      while (top > 0 && A[S[top]] <= A[i]) F[i] &= ~(1 << Pos[S[top--]]);
      S[++top] = i;
      F[i] |= (1 << Pos[i]);
    }
  }

  void init() {
    for (int i = 1; i <= N; ++i) scanf("%d", &A[i]);
    blockSize = log2(N) * 1.5;
    buildST();
    buildSubPre();
    buildBlock();
  }

  int queryMax(int l, int r) {
    int bl = Belong[l], br = Belong[r];
    if (bl != br) {
      int ans1 = 0;
      if (br - bl > 1) {
        int p = Log[br - bl - 1];
        ans1 = std::max(S[bl + 1][p], S[br - Pow[p]][p]);
      }
      int ans2 = std::max(Sub[l], Pre[r]);
      return std::max(ans1, ans2);
    } else {
      return A[l + __builtin_ctz(F[r] >> Pos[l])];
    }
  }
} R;

int M;

int main() {
  scanf("%d%d", &R.N, &M);
  R.init();
  for (int i = 0, l, r; i < M; ++i) {
    scanf("%d%d", &l, &r);
    printf("%d\n", R.queryMax(l, r));
  }
  return 0;
}
```

习题

[BJOI 2020] 封印[3]：SAM+RMQ

参考资料与注释

[1] P3793 由乃救爷爷

[2] Luogu P3865【模板】ST 表

[3] [BJOI 2020] 封印

14.2 并查集应用

Authors: sshwy

并查集，Kruskal 重构树的思维方式是很类似的，他们都能用于处理与连通性有关的问题。本文通过例题讲解的方式给大家介绍并查集思想的应用。

A

"A"

有 n 个点，初始时均为孤立点。

接下来有 m 次加边操作，第 i 次操作在 a_i 和 b_i 之间加一条无向边。设 $L(i,j)$ 表示结点 i 和 j 最早在第 $L(i,j)$ 次操作后连通。

在 m 次操作完后，你要求出 $\sum_{i=1}^{n}\sum_{j=i+1}^{n}L(i,j)$ 的值。

这是基础并查集的应用，并查集记录一下子树的大小。考虑统计每次操作的贡献。如果第 i 次操作 a_i 和 b_i 分属于两个不同子树，就将这两个子树合并，并将两者子树大小的乘积乘上 i 累加到答案里。时间复杂度 $O(n\alpha(n))$。

B

"B"

有 n 个点，初始时均为孤立点。

接下来有 m 次加边操作，第 i 次操作在 a_i 和 b_i 之间加一条无向边。

接下来有 q 次询问，第 i 次询问 u_i 和 v_i 最早在第几次操作后连通。

考虑在并查集合并的时候记录「并查集生成树」，也就是说如果第 i 次操作 a_i 和 b_i 分属于两个不同子树，那么把 (a_i,b_i) 这条边纳入生成树中。边权是 i。那么查询就是询问 u 到 v 路径上边权的最大值，可以使用树上倍增或者树链剖分的方法维护。时间复杂度 $O(n\log n)$。

另外一个方法是维护 Kruskal 重构树，其本质与并查集生成树是相同的。复杂度亦相同。

C

"C"

有 n 个点，初始时均为孤立点。

接下来有 m 次加边操作，第 i 次操作在 a_i 和 b_i 之间加一条无向边。

接下来有 q 次询问，第 i 次询问第 x_i 个点在第 t_i 次操作后所在连通块的大小。

离线算法：考虑将询问按 t_i 从小到大排序。在加边的过程中使用并查集顺便处理询问即可。时间复杂度 $O(q\log q+(n+q)\alpha(n))$。

在线算法：本题的在线算法只能使用 Kruskal 重构树。Kruskal 重构树与并查集的区别是：第 i 次操作 a_i 和 b_i 分属于两个不同子树，那么 Kruskal 会新建一个结点 u，然后让 a_i 所在子树的根和 b_i 所在子树的根分别连向 u，作为 u 的两个儿子。不妨设 u 的点权是 i。对于初始的 n 个点，点权为 0。

对于询问，我们只需要求出 x_i 在重构树中最大的一个连通块使得连通中的点权最大值不超过 t_i，询问的答案就是这个连通块中点权为 0 的结点个数，即叶子结点个数。

由于我们操作的编号是递增的，因此重构树上父结点的点权总是大于子结点的点权。这意味着我们可以在重构树上从 x_i 到根结点的路径上倍增找到点权最大的不超过 t_i 的结点。这样我们就求出了答案。时间复杂度 $O(n\log n)$。

D

"D"

给一个长度为 n 的 01 序列 $a_1, \dots, a_n$，一开始全是 0，接下来进行 m 次操作：

- 令 $a_x = 1$；
- 求 $a_x, a_{x+1}, \dots, a_n$ 中左数第一个为 0 的位置。

建立一个并查集，f_i 表示 $a_i, a_{i+1}, \dots, a_n$ 中第一个 0 的位置。初始时 $f_i = i$。

对于一次 $a_x = 1$ 的操作，如果 a_x 原本就等于 1，就不管。否则我们令 $f_x = f_{x+1}$。

时间复杂度 $O(n \log n)$，如果要使用按秩合并的话实现会较为麻烦，不过仍然可行。也就是说时间复杂度或为 $O(n\alpha(n))$。

E

"E"

给出三个长度为 n 的正整数序列 a，b，c。枚举 $1 \le i \le j \le n$，求 $a_i \cdot b_j \cdot \min_{i \le k \le j} c_k$ 的最大值。

本题同样有许多做法，这里我们重点讲解并查集思路。按权值从大到小考虑 c_k。相当于我们在 k 上加入一个点，然后将 $k-1$ 和 $k+1$ 位置上的点所在的连通块与之合并（如果这两个位置上有点的话）。连通块上记录 a 的最大值和 b 的最大值，即可在合并的时候更新答案。时间复杂度 $O(n \log n)$。

F

"F"

给出一棵 n 个点的树，接下来有 m 次操作：

- 加一条从 a_i 到 b_i 的边。
- 询问两个点 u_i 和 v_i 之间是否有至少两条边不相交的路径。

询问可以转化为：求 u_i 和 v_i 是否在同一个简单环上。按照双连通分量缩点的想法，每次我们在 a_i 和 b_i 间加一条边，就可以把 a_i 到 b_i 树上路径的点缩到一起。如果两条边 (a_i, b_i) 和 (a_j, b_j) 对应的树上路径有交，那么这两条边就会被缩到一起。

换言之，加边操作可以理解为，将 a_i 到 b_i 树上路径的边覆盖一次。而询问就转化为了：判断 u_i 到 v_i 路径上是否存在未被覆盖的边。如果不存在，那么 u_i 和 v_i 就属于同一个双连通分量，也就属于同一个简单环。

考虑使用并查集维护。给树定根，设 f_i 表示 i 到根的路径中第一个未被覆盖的边。那么每次加边操作，我们就暴力跳并查集。覆盖了一条边后，将这条边对应结点的 f 与父节点合并。这样，每条边至多被覆盖一次，总复杂度 $O(n \log n)$。使用按秩合并的并查集同样可以做到 $O(n\alpha(n))$。

本题的维护方式类似于 D 的树上版本。

G

"G"

无向图 G 有 n 个点，初始时均为孤立点（即没有边）。

接下来有 m 次加边操作，第 i 次操作在 a_i 和 b_i 之间加一条无向边。

每次操作后，你均需要求出图中桥的个数。

桥的定义为：对于一条 G 中的边 (x, y)，如果删掉它会使得连通块数量增加，则 (x, y) 被称作桥。

强制在线。

本题考察对并查集性质的理解。考虑用并查集维护连通情况。对于边双树，考虑维护有根树，设 p_i 表示结点 i 的父亲。也就是不带路径压缩的并查集。

如果第 i 次操作 a_i 和 b_i 属于同一个连通块，那么我们就需要将边双树上 a_i 到 b_i 路径上的点缩起来。这可以用并查集维护。每次缩点，边双连通分量的个数减少 1，最多减少 $n-1$ 次，因此缩点部分的并查集复杂度是 $O(n\alpha(n))$。

为了缩点，我们要先求出 a_i 和 b_i 在边双树上的 LCA。对此我们可以维护一个标记数组。然后从 a_i 和 b_i 开始轮流沿着祖先一个一个往上跳，并标记沿途经过的点。一但跳到了某个之前就被标记过的点，那么这个点就是 a_i 和 b_i 的 LCA。这个算法的复杂度与 a_i 到 b_i 的路径长度是线性相关的，可以接受。

如果 a_i 和 b_i 分属于两个不同连通块，那么我们将这两个连通块合并，并且桥的数量加 1。此时我们需要将两个点所在的边双树连起来，也就是加一条 a_i 到 b_i 的边。因此我们需要将其中一棵树重新定根，然后接到另一棵树上。这里运用启发式合并的思想：我们把结点数更小的重新定根。这样的总复杂度是 $O(n\log n)$ 的。

综上，该算法的总复杂度是 $O(n\log n + m\log n)$ 的。

小结

并查集与 Kruskal 重构树有许多共通点，而并查集的优化（按秩合并）正是启发式合并思想的应用。因此灵活运用并查集可以方便地处理许多与连通性有关的图论问题。

参考资料与注释

[1]

[2] Finding Bridges Online

14.3 括号序列

Authors: sshwy

定义一个合法括号序列（balanced bracket sequence）为仅由 (和) 构成的字符串且：

- 空串 ε 是一个合法括号序列。
- 如果 s 是合法括号序列，那么 (s) 也是合法括号序列。
- 如果 s,t 都是合法括号序列，那么 st 也是合法括号序列。

例如 (())() 是合法括号序列，而)() 不是。

有时候会有多种不同的括号，如 [()]{}。这样的变种括号序列与朴素括号序列有相似的定义。

本文将会介绍与括号序列相关的经典问题。

注：英语中一般称左括号为 opening bracket，而右括号是 closing bracket。

判断是否合法

判断 s 是否为合法括号序列的经典方法是贪心思想。该算法同样适用于变种括号序列。

我们维护一个栈，对于 $i=1,2,\ldots,|s|$ 依次考虑：

- 如果 s_i 是右括号且栈非空且栈顶元素是 s_i 对应的左括号，就弹出栈顶元素。

- 若不满足上述条件，则将 s_i 压入栈中。

在遍历整个 s 后，若栈是空的，那么 s 就是合法括号序列，否则就不是。时间复杂度 $O(n)$。

合法括号序列计数

考虑求出长度为 $2n$ 的合法括号序列 s 的个数 f_n。不妨枚举与 s_1 匹配的括号的位置，假设是 $2i+2$。它将整个序列又分成了两个更短的合法括号序列。因此

$$f_n = \sum_{i=0}^{n-1} f_i f_{n-i-1}$$

这同样是卡特兰数的递推式。也就是说 $f_n = \frac{1}{n+1}\binom{2n}{n}$。

当然，对于变种合法括号序列的计数，方法是类似的。假设有 k 种不同类型的括号，那么有 $f'_n = \frac{1}{n+1}\binom{2n}{n}k^n$。

字典序后继

给出合法的括号序列 s，我们要求出按字典序升序排序的长度为 $|s|$ 的所有合法括号序列中，序列 s 的下一个合法括号序列。在本问题中，我们认为左括号的字典序小于右括号，且不考虑变种括号序列。

我们需要找到一个最大的 i 使得 s_i 是左括号。然后，将其变成右括号，并将 $s[i+1,|s|]$ 这部分重构一下。另外，i 必须满足：$s[1,i-1]$ 中左括号的数量**大于**右括号的数量。

不妨设当 s_i 变成右括号后，$s[1,i]$ 中左括号比右括号多了 k 个。那么我们就让 s 的最后 k 个字符变成右括号，而 $s[i+1,|s|-k]$ 则用 $((\dots(())\dots))$ 的形式填充即可，因为这样填充的字典序最小。

该算法的时间复杂度是 $O(n)$。

" 参考实现"

```
bool next_balanced_sequence(string& s) {
  int n = s.size();
  int depth = 0;
  for (int i = n - 1; i >= 0; i--) {
    if (s[i] == '(')
      depth--;
    else
      depth++;

    if (s[i] == '(' && depth > 0) {
      depth--;
      int open = (n - i - 1 - depth) / 2;
      int close = n - i - 1 - open;
      string next =
          s.substr(0, i) + ')' + string(open, '(') + string(close, ')');
      s.swap(next);
      return true;
    }
  }
  return false;
}
```

字典序计算

给出合法的括号序列 s，我们要求出它的字典序排名。

考虑求出字典序比 s 小的括号序列 p 的个数。

不妨设 $p_i < s_i$ 且 $\forall 1 \le j < i, p_j = s_i$。显然 p_i 是左括号而 s_i 是右括号。枚举 i（满足 s_i 为右括号），假设 $p[1,i]$ 中左括号比右括号多 k 个，那么相当于我们要统计长度为 $|s| - i$ 且存在 k 个未匹配的右括号且不存在未匹配的左括号的括号序列的个数。

不妨设 $f(i,j)$ 表示长度为 i 且存在 j 个未匹配的右括号且不存在未匹配的左括号的括号序列的个数。

通过枚举括号序列第一个字符是什么，可以得到 f 的转移：$f(i,j) = f(i-1,j-1) + f(i-1,j+1)$。初始时 $f(0,0) = 1$。其实 f 是 OEIS - A053121[1]。

这样我们就可以 $O(|s|^2)$ 计算字典序了。

对于变种括号序列，方法是类似的，只不过我们需要对每个 s_i 考虑比它小的那些字符进行计算（在上述算法中因为不存在比左括号小的字符，所以我们只考虑了 s_i 为右括号的情况）。

另外，利用 f 数组，我们同样可以求出字典序排名为 k 的合法括号序列。

本页面主要译自博文 http://e-maxx.ru/algo/bracket_sequences[2] 与其英文翻译版 Balanced bracket sequences[3]。其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0。

参考资料与注释

[1] OEIS - A053121

[2] http://e-maxx.ru/algo/bracket_sequences

[3] Balanced bracket sequences

14.4 线段树与离线询问

Authors: xiezheyuan

线段树与离线询问结合的问题在 OI 领域也有出现。这种技巧又被称为线段树分治。

假如你需要维护一些信息，这些信息会在某一个时间段内出现，要求在离线的前提下回答某一个时刻的信息并，则可以考虑使用线段树分治的技巧。

实际上线段树分治常用于不带删的数据结构转成可以带删的数据结构，抑或是对于某一个属性的信息分别计算。

过程

首先我们建立一个线段树来维护时刻，每一个节点维护一个 `vector` 来存储位于这一段时刻的信息。

插入一个信息到线段树中和普通线段树的区间修改是类似的。

然后我们考虑如何处理每一个时间段的信息并。考虑从根节点开始分治，维护当前的信息并，然后每到一个节点的时候将这个节点的所有信息进行合并。回溯时撤销这一部分的贡献。最后到达叶子节点时的信息并就是对应的答案。

如果更改信息的时间复杂度为 $O(T(n))$，可以通过设置一个栈保留更改，以 $O(T(n))$ 的时间复杂度撤销。撤销不维持均摊复杂度。

整个分治流程的总时间复杂度是 $O(n \log n(T(n) + M(n)))$ 的，其中 $O(M(n))$ 为合并信息的时间复杂度，空间复杂度为 $O(n \log n)$。

" 实现"

```
#define ls (i << 1)
#define rs (i << 1 | 1)
```

```
#define mid ((l + r) >> 1)

vector<Object> tree[N << 2];  // 线段树

void update(int ql, int qr, Object obj, int i, int l, int r) {  // 插入
  if (ql <= l && r <= qr) {
    tree[i].push_back(obj);
    return;
  }
  if (ql <= mid) update(ql, qr, obj, ls, l, mid);
  if (qr > mid) update(ql, qr, obj, rs, mid + 1, r);
}

stack<Object> sta;  // 用于撤销的栈
Object now;         // 当前的信息并
Object ans[N];      // 答案

void solve(int i, int l, int r) {
  auto lvl = sta.size();  // 记录一下应当撤销到底几个
  for (Object x : tree[i]) sta.push(now), now = Merge(now, x);  // 合并信息
  if (l == r)
    ans[i] = now;  // 记录一下答案
  else
    solve(ls, l, mid), solve(rs, mid + 1, r);  // 分治
  while (sta.size() != lvl) {                  // 撤销信息
    now = sta.top();
    sta.pop();
  }
}
```

例题

"luogu P5787 二分图 / 【模板】 线段树分治[1]"

你需要维护一个 n 个点 m 条边的无向图。第 i 条边为 (x_i, y_i)，出现的时刻为 $[l_i, r_i)$，其余时刻消失。

对于每一个时刻，若此时该图为二分图，输出 Yes，否则输出 No。

" 解题思路"

使用种类并查集来维护一个图是否是二分图，然后就可以套用线段树分治了。

注意可撤销的并查集不能路径压缩，只能按秩合并。

" 参考代码"

```
#include <bits/stdc++.h>
#define ls (i << 1)
#define rs (i << 1 | 1)
#define mid ((l + r) >> 1)
using namespace std;

int n, m, k;
const int N = 2e5 + 5;
int fa[N << 1], siz[N << 1];
```

```cpp
struct UndoObject {
  int pos, val;

  UndoObject(int p, int v) { pos = p, val = v; }
};

stack<UndoObject> undo_sz, undo_fa;

int find(int x) {
  if (fa[x] == x)
    return x;
  else
    return find(fa[x]);
}

void merge(int u, int v) {
  int x = find(u), y = find(v);
  if (x == y) return;
  if (siz[x] < siz[y]) {
    swap(x, y);
  }
  undo_sz.push(UndoObject(x, siz[x]));
  siz[x] += siz[y];
  undo_fa.push(UndoObject(y, fa[y]));
  fa[y] = x;
}

void undo() {
  fa[undo_fa.top().pos] = undo_fa.top().val;
  undo_fa.pop();
  siz[undo_sz.top().pos] = undo_sz.top().val;
  undo_sz.pop();
}

vector<int> tree[N << 2];

void update(int ql, int qr, int v, int i, int l, int r) {
  if (ql <= l && r <= qr) {
    tree[i].push_back(v);
    return;
  }
  if (ql <= mid) update(ql, qr, v, ls, l, mid);
  if (qr > mid) update(ql, qr, v, rs, mid + 1, r);
}

struct edge {
  int u, v;
} g[N];

vector<bool> ret;

void solve(int i, int l, int r) {
  auto level = undo_fa.size();
```

```
  bool ans = 1;
  for (int u : tree[i]) {
    int a = find(g[u].u);
    int b = find(g[u].v);
    if (a == b) {
      for (int k = l; k <= r; k++) {
        ret.push_back(false);
      }
      ans = 0;
      break;
    }
    merge(g[u].u, g[u].v + n);
    merge(g[u].v, g[u].u + n);
  }
  if (ans) {
    if (l == r) {
      ret.push_back(1);
    } else {
      solve(ls, l, mid);
      solve(rs, mid + 1, r);
    }
  }
  while (undo_fa.size() > level) {
    undo();
  }
}

signed main() {
  cin >> n >> m >> k;
  for (int i = 1; i <= (n << 1); i++) {
    fa[i] = i;
    siz[i] = 1;
  }
  for (int i = 1, l, r; i <= m; i++) {
    cin >> g[i].u >> g[i].v >> l >> r;
    update(l + 1, r, i, 1, 1, k);
  }
  solve(1, 1, k);
  for (bool i : ret) {
    cout << (i ? "Yes" : "No") << '\n';
  }
  return 0;
}
```

" **颜色限制** restriction"

一个 n 点 m 边的无向图，有 k 种颜色编号为 $0 \sim k-1$，每条边有一种颜色。

对于每种颜色，请判断假如删去所有这种颜色的边，得到的图是否连通？是否是一棵树？

输出满足删去后图连通的颜色数和删去后图是树的颜色数。

" 解题思路"

对于每一种颜色，建立一个时间，在这个时间内没有这个颜色的边，其他边都有。用一个并查集维护一下即可。

" 参考代码"

```cpp
#include <bits/stdc++.h>
#define ls (i << 1)
#define rs (i << 1 | 1)
#define mid ((l + r) >> 1)
using namespace std;

int n, m, k;
const int N = 1e5 + 5;

struct edge {
  int u, v, c;
} g[N];

vector<int> t[N << 2];
int fa[N], siz[N], cnt[N];

void update(int ql, int qr, int v, int i, int l, int r) {
  if (ql > qr) return;
  if (ql <= l && r <= qr) {
    t[i].push_back(v);
    return;
  }
  if (ql <= mid) update(ql, qr, v, ls, l, mid);
  if (qr > mid) update(ql, qr, v, rs, mid + 1, r);
}

stack<pair<int, int> > fas, sizs;

int find(int x) { return fa[x] == x ? x : find(fa[x]); }

void merge(int u, int v) {
  int fu = find(u), fv = find(v);
  if (fu == fv) return;
  if (siz[fu] < siz[fv]) swap(fu, fv);
  fas.push(make_pair(fv, fa[fv]));
  fa[fv] = fu;
  sizs.push(make_pair(fu, siz[fu]));
  siz[fu] += siz[fv];
}

bitset<N> ans;

void solve(int i, int l, int r) {
  unsigned lvl = fas.size();
  for (int eg : t[i]) merge(g[eg].u, g[eg].v);
  if (l == r)
    ans[l] = (siz[find(1)] == n);
  else
    solve(ls, l, mid), solve(rs, mid + 1, r);
  while (fas.size() != lvl) {
    auto p1 = fas.top(), p2 = sizs.top();
    fas.pop(), sizs.pop();
```

```cpp
    fa[p1.first] = p1.second;
    siz[p2.first] = p2.second;
  }
}

signed main() {
  ios::sync_with_stdio(false);
  cin.tie(0);
  cout.tie(0);
  cin >> n >> m >> k;
  for (int i = 1; i <= m; i++) {
    cin >> g[i].u >> g[i].v >> g[i].c;
    g[i].c++;
    update(1, g[i].c - 1, i, 1, 1, k);
    update(g[i].c + 1, k, i, 1, 1, k);
    cnt[g[i].c]++;
  }
  for (int i = 1; i <= n; i++) {
    fa[i] = i;
    siz[i] = 1;
  }
  solve(1, 1, k);
  int ans1 = 0, ans2 = 0;
  for (int i = 1; i <= k; i++) {
    ans1 += ans[i];
    ans2 += (ans[i] && (m - cnt[i]) == (n - 1));
  }
  cout << ans1 << ' ' << ans2;
  return 0;
}
```

"luogu P4219 [BJOI2014] **大融合**[2]"

需要维护一个 n 个点的森林，初始时是散点。

有 q 个操作，支持：

- A x y 连边 (x,y)。
- Q x y 输出经过边 (x,y) 的路径数。

允许离线。

" 解题思路"

考虑允许离线，因此可以想到线段树分治。

然后考虑如何支持 Q 操作。如果不存在 (x,y) 这条边，答案就是 x 所在连通块大小乘上 y 所在连通块大小。可以用并查集维护。

因此你可以将 Q 拆成三个时间 $k-1,k,k+1$。其中 $k-1$ 是这条边的终止时刻，$k+1$ 是这条边的起始时刻。这样 k 就没有这条边，正好回答询问。

" 参考代码"

```cpp
#include <bits/stdc++.h>
#define ls (i << 1)
```

```cpp
#define rs (i << 1 | 1)
#define mid ((l + r) >> 1)
using namespace std;

int n, m;
const int N = 1e5 + 5;
int fa[N], siz[N], tim;

struct UndoObject {
  int pos, val;

  UndoObject(int p, int v) { pos = p, val = v; }
};

stack<UndoObject> undo_sz, undo_fa;

int find(int x) {
  if (fa[x] == x)
    return x;
  else
    return find(fa[x]);
}

void merge(int u, int v) {
  int x = find(u), y = find(v);
  if (x == y) return;
  if (siz[x] < siz[y]) {
    swap(x, y);
  }
  undo_sz.push(UndoObject(x, siz[x]));
  siz[x] += siz[y];
  undo_fa.push(UndoObject(y, fa[y]));
  fa[y] = x;
}

void undo() {
  fa[undo_fa.top().pos] = undo_fa.top().val;
  undo_fa.pop();
  siz[undo_sz.top().pos] = undo_sz.top().val;
  undo_sz.pop();
}

vector<pair<int, int> > tree[N << 4];

void update(int ql, int qr, pair<int, int> v, int i, int l, int r) {
  if (ql <= l && r <= qr) {
    tree[i].push_back(v);
    return;
  }
  if (ql <= mid) update(ql, qr, v, ls, l, mid);
  if (qr > mid) update(ql, qr, v, rs, mid + 1, r);
}

map<pair<int, int>, int> tims;
```

```
struct ops {
  int l, r;
  pair<int, int> v;
} opp[N << 3];

int opcnt;
map<int, int> queries;
map<int, pair<int, int> > querylr;
int ans[N << 3];

void solve(int i, int l, int r) {
  auto level = undo_fa.size();
  for (auto u : tree[i]) {
    merge(u.first, u.second);
  }
  if (l == r) {
    if (queries[l]) {
      int x = querylr[l].first;
      int y = querylr[l].second;
      // cout<<siz[find(x)]<<' '<<siz[find(y)]<<'\n';
      ans[l] = siz[find(x)] * siz[find(y)];
    }
  } else {
    solve(ls, l, mid);
    solve(rs, mid + 1, r);
  }
  while (undo_fa.size() > level) {
    undo();
  }
}

signed main() {
  ios::sync_with_stdio(false);
  cin.tie(0);
  cout.tie(0);
  cin >> n >> m;
  for (int i = 1; i <= n; i++) {
    fa[i] = i;
    siz[i] = 1;
  }
  while (m--) {
    char op;
    int x, y;
    cin >> op >> x >> y;
    if (op == 'A') {
      tims[make_pair(x, y)] = ++tim;
    } else {
      pair<int, int> xy = make_pair(x, y);
      opp[++opcnt] = {tims[xy], (++tim), xy};
      queries[++tim] = 1;
      // cout<<tim<<'\n';
      querylr[tim] = xy;
      tims[xy] = ++tim;
```

```cpp
    }
  }
  int tm = ++tim;
  for (auto i : tims) {
    opp[++opcnt] = {tims[i.first], tm, i.first};
  }
  for (int i = 1; i <= opcnt; i++) {
    // cout<<opp[i].l<<' '<<opp[i].r<<' '<<opp[i].v.first<<'
    // '<<opp[i].v.second<<'\n';
    update(opp[i].l, opp[i].r, opp[i].v, 1, 1, tim);
  }
  // cout<<tim<<'\n';
  solve(1, 1, tim);
  for (int i = 1; i <= tim; i++) {
    if (queries[i]) {
      cout << ans[i] << "\n";
    }
  }
  return 0;
}
```

"luogu P2056 [ZJOI2007] **捉迷藏**[3]"

出一个 n 个点的树，每个点有黑白两种颜色。初始时每个点都是黑色的。q 次操作，支持：

- `C x` 将第 x 个点的颜色反转。
- `G` 询问树上两个黑色点的最远距离。特别地，若不存在黑色点，输出 -1。

允许离线。

" 解题思路"

首先考虑如何维护树上点集的直径，有下面的推论：

> "luogu P2056 [ZJOI2007] **捉迷藏**[3]"
>
> 对于一个集合 S 和只有一个点的集合 $\{P\}$。若集合 S 的直径为 (U,V)。则点集 $S \cap \{P\}$ 的直径只可能为 $(U,V),(U,P)$ 或 (V,P)。

然后考虑解决原问题。我们可以考虑维护黑色点集，维护每一个点在黑色点集中的若干个时间段（具体你开一个桶记录一下上一次进入黑色点集的时刻即可）。

然后就自然地想到离线，将所有时间刻插入到线段树中。然后在线段树上分治，每次线段树上的点会记录当前时间段点集新增的点，新增点可以使用上面的推论，找到新点集直径的两个端点。

撤销是平凡的，开一个栈记录一下直径端点的变化即可。

```
??? " 参考代码"
    ```cpp
    #include <bits/stdc++.h>
    #define ls (i << 1)
    #define rs (i << 1 | 1)
    #define mid ((l + r) >> 1)
    using namespace std;

    const int N = 5e5 + 5, M = 5e5 + 5;
```
```

```
int siz[N], dep[N], father[N], top[N], son[N];
int n, q;

struct edge {
  int nxt, to;
} g[N << 1];

int head[N], ec;

void add(int u, int v) {
  g[++ec].nxt = head[u];
  g[ec].to = v;
  head[u] = ec;
}

void dfs1(int u, int fa) {
  dep[u] = dep[fa] + 1;
  siz[u] = 1;
  father[u] = fa;
  for (int i = head[u]; i; i = g[i].nxt) {
    int v = g[i].to;
    if (v == fa) continue;
    dfs1(v, u);
    siz[u] += siz[v];
    if (siz[son[u]] < siz[v]) son[u] = v;
  }
}

void dfs2(int u, int fa) {
  if (son[u]) {
    top[son[u]] = top[u];
    dfs2(son[u], u);
  }
  for (int i = head[u]; i; i = g[i].nxt) {
    int v = g[i].to;
    if (v == fa || v == son[u]) continue;
    top[v] = v;
    dfs2(v, u);
  }
}

int lca(int x, int y) {
  while (top[x] != top[y]) {
    if (dep[top[x]] < dep[top[y]]) swap(x, y);
    x = father[top[x]];
  }
  return dep[x] < dep[y] ? x : y;
}

int dis(int x, int y) { return dep[x] + dep[y] - (dep[lca(x, y)] << 1); }

vector<int> t[N << 2];
```

```
void update(int ql, int qr, int v, int i, int l, int r) {
  if (ql <= l && r <= qr) {
    t[i].push_back(v);
    return;
  }
  if (ql <= mid) update(ql, qr, v, ls, l, mid);
  if (qr > mid) update(ql, qr, v, rs, mid + 1, r);
}

stack<pair<int, int> > stk;
int u, v;
int ans[M];

void solve(int i, int l, int r) {
  auto lvl = stk.size();
  for (int x : t[i]) {
    stk.push(make_pair(u, v));
    if (!u && !v)
      u = x, v = x;
    else {
      vector<int> vct = {dis(u, v), dis(u, x), dis(v, x)};
      sort(vct.begin(), vct.end(), greater<int>());
      if (vct[0] == dis(u, x))
        v = x;
      else if (vct[0] == dis(x, v))
        u = x;
    }
  }
  if (l == r)
    ans[l] = (!u || !v) ? -1 : dis(u, v);
  else
    solve(ls, l, mid), solve(rs, mid + 1, r);
  while (stk.size() != lvl) {
    auto top = stk.top();
    u = top.first, v = top.second;
    stk.pop();
  }
}

int lst[N];
bitset<N> col;
bitset<M> haveq;

signed main() {
  ios::sync_with_stdio(false);
  cin.tie(0);
  cout.tie(0);
  cin >> n;
  for (int i = 1; i < n; i++) {
    int u, v;
    cin >> u >> v;
    add(u, v);
    add(v, u);
  }
```

````
    top[1] = 1;
    dfs1(1, 0);
    dfs2(1, 0);
    for (int i = 1; i <= n; i++) lst[i] = 1;
    cin >> q;
    for (int i = 2; i <= (q + 1); i++) {
      char c;
      int x;
      cin >> c;
      if (c == 'C') {
        cin >> x;
        if (!col[x]) {
          col[x] = 1;
          update(lst[x], i, x, 1, 1, q + 2);
        } else
          col[x] = 0, lst[x] = i;
      } else
        haveq[i] = 1;
    }
    for (int i = 1; i <= n; i++) {
      if (!col[i]) update(lst[i], q + 2, i, 1, 1, q + 2);
    }
    solve(1, 1, q + 2);
    for (int i = 1; i <= (q + 2); i++) {
      if (haveq[i]) cout << ans[i] << '\n';
    }
    return 0;
  }

  ```
````

习题

- CF601E A Museum Robbery[4] 线段树分治 + 背包 dp。
- CF19E Fairy[5] 线段树分治 + 种类并查集。
- luogu P5227 [AHOI2013] 连通图[6] 线段树分治 + 并查集。
- luogu P4319 变化的道路[7] 线段树分治 + Link Cut Tree 维护最小生成树。
- luogu P3733 [HAOI2017] 八纵八横[8] 线段树分治 + 线性基。

参考资料与注释

[1] luogu P5787 二分图/【模板】线段树分治

[2] luogu P4219 [BJOI2014] 大融合

[3] luogu P2056 [ZJOI2007] 捉迷藏

[4] CF601E A Museum Robbery
````

[5] CF19E Fairy

[6] luogu P5227 [AHOI2013] 连通图

[7] luogu P4319 变化的道路

[8] luogu P3733 [HAOI2017] 八纵八横

[9] Deleting from a data structure

第 15 章

关于 Hulu

15.1 关于 Hulu

Hulu 是美国领先的互联网专业视频服务平台，成立于 2006 年，目前由迪士尼控股，在美国拥有超过 2900 万付费用户。通过 Hulu，用户可以在电脑、可联网电视机、手机、平板电脑等多种设备上观看长视频和电视直播。

2017 年，Hulu 自制剧《使女的故事》成为艾美奖和金球奖双料最大赢家，斩获十余奖项；2019 年，Hulu 自制纪录片《滑板少年》获奥斯卡最佳纪录片提名。

Hulu 总部位于美国洛杉矶，在全球一共拥有 8 个办公室。北京办公室是仅次于总部的第二大研发中心，也是从 Hulu 成立伊始就具有重要战略地位的研发中心。

Hulu 北京办公室成立于 2007 年，目前规模为 200 人，其中博士和硕士比例超过百分之八十。Hulu 北京于 2018 和 2019 连续两年当选「大中华区最佳职场 ®」。

正如 Hulu 的产品是技术与娱乐的最佳结合，在 Hulu 工作，既能与一群志同道合的技术达人合作学习，又能享受充满乐趣的工作环境。Hulu 北京面向校园群体开放全年实习生招聘项目。欢迎关注 **Hulu Beijing 微信公众号**了解更多关于 Hulu 的信息！