NOI 全国青少年信息学奥林匹克竞赛

IOI2018 中国国家候选队论文集

教练：张瑞喆

2018年 4月

目 录

浅谈生成函数在掷骰子问题上的应用 杨懋龙 **1**

《后缀树结点数》命题报告及一类区间问题的优化 陈江伦 **11**

浅谈保序回归问题 高睿泉 **23**

《**Fim 4**》命题报告 吴瑾昭 **34**

解决树上连通块问题的一些技巧和工具 任轩笛 **45**

Leafy Tree 及其实现的加权平衡树 王思齐 **75**

《小**H**爱染色》命题报告 陈嘉乐 **87**

一些特殊的数论函数求和问题 朱震霆 **92**

浅谈**DFT**在信息学竞赛中的应用 刘承奥 **113**

《完美的队列》命题报告 林旭恒 **132**

浅谈拟阵的一些拓展及其应用 杨乾澜 **143**

浅谈**Splay**与**Treap**的性质及其应用 董炜隽 **163**

《最小方差生成树》命题报告 何中天 **179**

欧拉图相关的生成与计数问题探究 陈通 **191**

浅谈生成函数在掷骰子问题上的应用

长沙市长郡中学 杨懋龙

摘 要

掷骰子问题是算法竞赛中的一类常见问题。通过对这一类问题的认真研究，作者得出，这类问题都可以使用生成函数来解决。因此本文围绕掷骰子系列问题，阐述如何用生成函数来一步步解决问题，以及相较于传统做法，用生成函数解决问题的优越性。

关键词：生成函数、掷骰子问题、概率与期望

1 引言

掷骰子问题是算法竞赛中的一类常见问题，在近年的竞赛中多次出现。

而生成函数又是解决这类问题一个有效工具，与传统方法相比具有易于计算且扩展性强等优点。但现在OI届对这种方法的研究十分稀少。

第2节中，约定了一些符号。

第3节中，介绍了概率生成函数的定义，以及一些性质。

第4节中，介绍了该算法的基础应用。

第5、6节中，结合题目介绍了一些更复杂的应用。

2 符号约定

符号1. A_i 表示序列 A 的第 i 个数字。

符号2. $A[l,r]$ 表示序列 A 的第 l 个数字到第 r 个数字组成的序列。

符号3. $f^{(k)}(x)$ 为 $f(x)$ 的 k 阶导数。

符号4. $[P]$ 为艾佛森括号，如果方括号内的条件满足则为 1 ，不满足则为 0 。

3 预备知识

3.1 普通生成函数

数列 $a_0, a_1, a_2, \dots$ 的普通生成函数为：

$$A(x) = \sum_{i=0}^{\infty} a_i x^i$$

3.2 概率生成函数

如果对于数列 $a_0, a_1, a_2, \ldots$ ，存在某个离散随机变量 X 满足 $\Pr(X = i) = a_i$ ，那么 $a_n (n \in \mathbb{N})$ 的普通生成函数被称为 X 的概率生成函数。

也就是说，如果 X 是非负整数集 $\mathbb{N}$ 上的离散随机变量，那么 X 的概率生成函数为：

$$F(z) = \mathbb{E}(z^X) = \sum_{i=0}^{\infty} \Pr(X = i) z^i$$

3.2.1 概率生成函数性质

因为是 X 是非负整数集 $\mathbb{N}$ 上的离散随机变量，所以必有

$$F(1) = \sum_{i=0}^{\infty} \Pr(X = i) = 1$$

对 $F(z)$ 求导，得到

$$F'(z) = \sum_{i=0}^{\infty} i \Pr(X = i) \cdot z^{i-1}$$

所以 X 的期望

$$E(x) = F'(1) = \sum_{i=0}^{\infty} i \Pr(X = i)$$

进一步推导可以得到

$$E(X^{\underline{k}}) = F^{(k)}(1), k \neq 0$$

所以 X 的方差

$$Var(X) = F''(1) + F'(1) - (F'(1))^2$$

3.3 border

对于一个长度为 L 的序列 A ，若 $A[1, i] = A[L-i+1, L]$ ，则称 $A[1, i]$ 是 A 的一个border。

4 基本题型

例题1. 歌唱王国[1]

[1]来源：CTSC2006

给定一个长度为 L 的序列 A 。然后每次掷一个标有 1 到 m 的公平骰子并将其上的数字加入到初始为空的序列 B 的末尾，如果序列 B 中已经出现了给定序列 A ，即 A 是 B 的子串，则停止，求序列 B 的期望长度。 $L \leq 10^5$ 。

4.1 分析

掷骰子问题的基础形式是给出一个序列和一个骰子，然后不断地掷骰子生成一个随机序列，直到给定序列在随机序列中出现才停止，并求期望的掷骰子次数。

4.2 一种非生成函数方法

令 E_i 表示当前满足 $A[1,i]$ 是 B 的后缀，到首次满足 $A[1,i+1]$ 是 B 的后缀的期望掷骰子次数。那么需要求的便是 $\sum_{i=0}^{n-1} E_i$ 。

定义 $f_{i,j}$ 为在 $A[1,i]$ 后加入数字 j 组成的新序列的最长非自身border长度。那么可以得到方程

$$E_i = 1 + \frac{1}{m}\sum_{j=1}^{m}[j \neq A_{i+1}]\sum_{k=f_{i,j}}^{i} E_k$$

解方程可以得到

$$E_i = m + (m-1)\sum_{k=1}^{i-1} E_k - \sum_{j=1}^{m}[j \neq A_{i+1}]\sum_{k=1}^{f_{i,j}-1} E_k$$

这样时间复杂度是 $O(nm)$ 的。主要的问题是在求 $\sum_{j=1}^{m}[j \neq A_{i+1}]\sum_{k=1}^{f_{i,j}-1} E_k$ ，而这个可以优化的。根据比较 $A[1,i]$ 与其最长非自身border的式子，可以发现两式子之间只有 $O(1)$ 个 $f_{i,j}$ 不同，所以每次只需要重新维护这 $O(1)$ 个值即可。这样时间复杂度就降为了 $O(n)$ 。

可以发现这个方法十分复杂，并且扩展性低。而下面介绍的生成函数方法则会比这个方法优越许多。

4.3 生成函数方法

令 a_i 表示 $A[1,i]$ 是否是 A 的一个border， a_i 为 1 表示是， 0 为不是。

令 f_i 为结束时随机序列长度为 i 的概率，其概率生成函数为 $F(x)$ 。令辅助数列 g_i 为随机序列长度达到 i 且还未结束的概率，其普通生成函数为 $G(x)$ 。

我们需要求的便是 $F'(1)$ 。

通过分析可以得到如下等式：

$$F(x) + G(x) = 1 + G(x)\cdot x \tag{1}$$

$$G(x)\cdot\left(\frac{1}{m}x\right)^L = \sum_{i=1}^{L} a_i \cdot F(x)\cdot\left(\frac{1}{m}x\right)^{L-i} \tag{2}$$

(1) 的意义是在一个未结束的序列后加一个数字，可能结束也可能没结束， 1 是初始时随机序列为空的情况。

(2) 的意义是在一个未结束的序列后加上给定序列，则必定会结束，但是有可能在未加到 L 个数字时就已经结束了，这时一定要满足已经添加的序列是给定序列的一个border。

将 (1) 求导并代入 $x = 1$ 可得

$$\begin{aligned} F'(x) + G'(x) &= G'(x) \cdot x + G(x) \\ F'(1) &= G(1) \end{aligned} \tag{3}$$

再将 $x = 1$ 代入 (2) 可得

$$\begin{aligned} G(1) \cdot \left(\frac{1}{m}\right)^L &= \sum_{i=1}^{L} a_i \cdot F(1) \cdot \left(\frac{1}{m}\right)^{L-i} \\ G(1) &= \sum_{i=1}^{L} a_i \cdot m^i \end{aligned} \tag{4}$$

由 (3)(4) 可知 $F'(1) = \sum\limits_{i=1}^{L} a_i \cdot m^i$ 。

使用KMP算法或hash便可以在 $O(L)$ 的时间复杂度内求出 a_i 并计算出 $F'(1)$ 。

4.4 进一步思考

我们可以进一步得到求方差的方法。

根据 $Var(X) = F''(1) + F'(1) - (F'(1))^2$ ，可知只需要知道 $F'(1)$ 和 $F''(1)$ 即可， $F'(1)$ 已经知道了，那么就是要求 $F''(1)$ 。

将 (1) 求二阶导并代入 $x = 1$ 可得

$$\begin{aligned} F''(x) + G''(x) &= G'(x) + G''(x) \cdot x + G'(x) \\ F''(1) &= 2G'(1) \end{aligned}$$

再将 (2) 求导并代入 $x = 1$ 可得

$$\begin{aligned} G(x) &= \sum_{i=1}^{L} a_i \cdot m^i \cdot F(x) \cdot x^{-i} \\ G'(x) &= \sum_{i=1}^{L} a_i \cdot m^i \cdot (F'(x) \cdot x^{-i} - i \cdot F(x) \cdot x^{-i-1}) \\ G'(1) &= \sum_{i=1}^{L} a_i \cdot m^i \cdot (F'(1) - i) \end{aligned}$$

所以 $F''(1) = 2 \cdot \sum_{i=1}^{L} a_i \cdot m^i \cdot (F'(1) - i)$ 。

如果再进一步推导并归纳可以得到

$$F^{(k)}(1) = k \cdot \sum_{i=1}^{L} a_i \cdot m^i \sum_{j=0}^{k-1} (-1)^j \binom{k-1}{j} \frac{(i+j-1)!}{(i-1)!} F^{(k-1-j)}(1)$$

4.5 小结

使用生成函数来解决这类问题的方法通常是先定义一个概率生成函数 $F(x)$ 和一个辅助生成函数 $G(x)$ ，然后是在未结束的情况后加入一个数或一个给定序列，并根据实际情况来列出方程。最后通过代值和求导来解出所需要的 $F'(1)$ 。

5 复杂的题型

例题2. 抛骰子游戏

给定 n 个长度分别为 L_i 的序列 A_i 。再给出一个标有 1 到 m 的骰子，其中抛出 i 的概率为 P_i 。然后每次抛一次骰子将骰子上的数字加入到初始为空的序列 B 末尾，如果给定的 n 个序列都是 B 的子串，则停止，求序列 B 的期望长度。保证 $n \le 15$ ， $L \le 20000$ ， $m \le 10^5$ 。

5.1 分析

这个问题本质还是掷骰子问题，但是变成了需要 n 个序列都在其中出现。

5.2 解法

令 T_i 为 S_i 首次出现时随机序列的长度，那么要求的就是 $E(\max_{i\in\{1,2,\ldots,n\}} T_i)$ 。

如果一个序列 A_i 是另一个序列 A_j 的子串，那么必有 $T_i \le T_j$ ，可以将 A_i 去除。

max 不太好求，对其进行Min-Max容斥， $\max_{i=1}^{n} T_i = \sum_{S\subseteq\{1,2,\ldots,n\}} (-1)^{|S|+1} \min_{i\in S} T_i$ 。

根据期望的线性性，可以转换为求 $E(\min_{i\in S} T_i)$ ，即有一个出现就结束。不妨假设 $S = \{1,2,\ldots,n\}$ 。

定义 $P(A) = \prod_{i\in A} P_i$ 。

令 $a_{i,j,k} = \left[A_i[1,k] = A_j[L_j-k+1,L_j]\right]$ 。

令 $f_{i,j}$ 为首次出现的是序列 A_i 且随机序列的长度为 j 的概率，其普通生成函数为 $F_i(x)$ 。令辅助数列 g_i 为随机序列长度达到 i 且还未结束的概率，其普通生成函数为 $G(x)$ 。

可以得到如下等式：

$$\sum_{i=1}^{n} F_i(x) + G(x) = 1 + G(x) \cdot x \tag{5}$$

$$G(x) \cdot P(A_i)x^{L_i} = \sum_{j=1}^{n}\sum_{k=1}^{L_i} a_{i,j,k} \cdot F_j(x) \cdot P(A_i[k+1, L_i]) \cdot x^{L_i-k}, \forall i \in S \tag{6}$$

要求的便是 $\sum_{i=1}^{n} F_i'(1)$ 。

对 (5) 求导并将 $x = 1$ 代入得

$$\sum_{i=1}^{n} F_i'(1) = G(1)$$

将 $x = 1$ 代入 (6) 可得

$$G(1) = \sum_{j=1}^{n}(\sum_{k=1}^{L_i} a_{i,j,k} \cdot \frac{1}{P(A_i[1,k])}) \cdot F_j(1), \forall i \in S$$

现在有 $n+1$ 个未知数，但只有 n 个方程。可以发现 $\sum_{i=1}^{n} F_i(x)$ 为随机序列长度的概率生成函数，所以有 $\sum_{i=1}^{n} F_i(1) = 1$ 。这样就可以高斯消元求出$G(1)$ 了。同时还可以得到 $F_i(1)$ ，即首次出现的序列是第 i 个序列的概率，如[SDOI2017]硬币游戏就是要求 $F_i(1)$ 。

使用hash可以在 $O(n^2L)$ 的时间复杂度内求出 a 数组并对于每组 (i, j) 计算出 $\sum_{k=1}^{L_i} a_{i,j,k} \cdot \frac{1}{P(A_i[1,k])}$ ，对于每个 S 在 $O(n^3)$ 内可以高斯消元解出 $G(1)$ 。所以总时间复杂度为 $O(n^2L+2^n n^3)$ 。

6 一些新的应用

6.1 模糊匹配

例题3. Dice[2]

一个 m 面的公平骰子，求最后 n 次结果相同就结束的期望次数或者求最后 n 次结果全不同就结束的期望次数。保证 $n, m \leq 10^6$ ，且对第二问保证 $n \leq m$ 。

[2]来源：2013 Multi-University Training Contest 5

6.1.1 传统差分方法

第一问

令 f_i 为当前状态满足后 i 次相同，从当前状态到结束的期望掷骰子次数。那么要求的就是 f_0 。

可以列出转移式：

$$f_i = \frac{1}{m}f_{i+1} + \frac{m-1}{m}f_1 + 1$$

先差分一下得到

$$f_{i-1} - f_i = \frac{1}{m}(f_i - f_{i+1})$$

因为 $f_0 - f_1 = 1$ ，所以 $f_{i-1} - f_i = m^{i-1}$ 。同时 $f_n = 0$ ，那么 $f_0 = \sum_{i=0}^{n-1} m^i = \frac{m^n-1}{m-1}$ 。

第二问

令 g_i 为当前状态满足后 i 次不同，从当前状态到结束的期望掷骰子次数。那么要求的就是 g_0 。

转移式：

$$g_i = \frac{m-i}{m}g_{i+1} + \frac{1}{m}\sum_{j=1}^{i} g_j + 1$$

先差分一下

$$g_{i-1} - g_i = \frac{m-i}{m}(g_i - g_{i+1})$$

同样有 $g_0 - g_1 = 1, g_n = 0$ ，所以也可以在 $O(n)$ 时间内求出 g_0 。

6.1.2 生成函数做法

分析

虽然可以把这个问题简单地看成是多序列的掷骰子问题，但合法序列数量太多，而它们都满足同一条件，所以可以将其整合在一起。

第一问

令 f_i 为 i 次结束的概率，其概率生成函数为 $F(x)$ 。令 g_i 为投掷 i 次且还未结束的概率，其普通生成函数为 $G(x)$ 。

那么可以得到：

$$\begin{aligned} F(x)+G(x) &= G(x)\cdot x+1 && (7)\\ G(x)\cdot\left(\frac{1}{m}x\right)^{n-1} &= \sum_{i=1}^{n} F(x)\cdot\left(\frac{1}{m}x\right)^{n-i} && (8)\end{aligned}$$

套用与之前类似的解法，可以解得 $F'(1)=\frac{m^n-1}{m-1}$ 。

第二问

令 f_i 为 i 次结束的概率，其概率生成函数为 $F(x)$ 。令 g_i 为投掷 i 次且还未结束的概率，其普通生成函数为 $G(x)$ 。

那么可以得到：

$$\begin{aligned} F(x)+G(x) &= G(x)\cdot x+1 && (9)\\ G(x)\cdot\left(\frac{1}{m}x\right)^{n}\cdot\frac{m!}{(m-n)!} &= \sum_{i=1}^{n} F(x)\cdot\left(\frac{1}{m}x\right)^{n-i}\cdot\frac{(m-i)!}{(m-n)!} && (10)\end{aligned}$$

可以解得 $F'(1)=\sum\limits_{i=1}^{n} m^i\frac{(m-i)!}{m!}$ 。

6.1.3 对比

虽然两种方法都可以得出一样的结果，但是生成函数做法在运算上显然要比传统方法简单，且方法更易于理解，同时对于不同的题目需要改动的部分极少。

6.2 更复杂的情况

例题4. 一个有趣的概率小问题[3]

一个 n 面的骰子，每一面标号 1 到 n 。有个初始为 0 的计数器，每次扔骰子，如果结果是奇数，那么计数器清零，否则计数器加 1 并且如果结果是 n 则结束。问结束时计数器的期望。保证 n 是偶数。

6.2.1 解法

看起来答案是 $\frac{n}{2}$ ，但其实并不是。

同样还是令 f_i 为计数器为 i 时结束的概率，其概率生成函数为 $F(x)$ 。但辅助数组的定义要改一下，令 g_i 为计数器 $=i$ 的期望次数，其普通生成函数为 $G(x)$ 。

[3]来源：http://roosephu.github.io/2017/12/31/condexp/

可以得到方程：

$$F(x)+G(x) = G(x)\cdot\frac{x}{2}+\frac{G(1)}{2}+1 \quad (11)$$

$$F(x) = G(x)\cdot\frac{x}{n} \quad (12)$$

第一个式子的意思是，对于每个 G 的状态，有 $\frac{1}{2}$ 的概率计数器 +1 ，有 $\frac{1}{2}$ 的概率计数器清零。

因为 $F(1)=1$ ，所以 $G(1)=n$ 。 (11) 式可以变为

$$F(x)+G(x) = G(x)\cdot\frac{x}{2}+\frac{n}{2}+1 \quad (13)$$

将 (12) 代入 (13) 可得：

$$G(x)\cdot(1+\frac{x}{n}-\frac{x}{2}) = \frac{n}{2}+1$$

$$G(x) = \frac{\frac{n}{2}+1}{1+\frac{x}{n}-\frac{x}{2}}$$

$$G(x) = \frac{n^2+2n}{2n-(n-2)x} \quad (14)$$

再将 (14) 代回 (12) 可知：

$$F(x) = \frac{(n+2)x}{2n-(n-2)x}$$

$$F'(x) = \frac{2n\cdot(n+2)}{[2n-(n-2)x]^2}$$

$$F'(1) = \frac{2n}{n+2}$$

7 总结

对于掷骰子这类问题，我们可以构建生成函数并利用题目中的信息来建立生成函数之间的关系，然后解出想要的值。使用这种方法，可以避免传统的高斯消元方法。

本文所提到的这类问题还值得去进行更加深入地研究。因此希望本文能够起到一个抛砖引玉的作用，希望感兴趣的读者，能够进行进一步研究，扩展笔者做法，从而得到更多更有趣的做法与应用。

感谢

- 感谢中国计算机学会提供学习和交流的平台。
- 感谢国家队教练张瑞喆的指导。

- 感谢谢秋锋老师的关心和指导。
- 感谢陈江伦同学、郭晓旭前辈为我提供思路。
- 感谢陈江伦同学、罗雨屏学长、郭晓旭前辈、王修涵同学为本文审稿。
- 感谢家人对我的关心和支持。

参考文献

[1] Wikipedia, Probability-generating function.

[2] Ronald L. Graham, Donald E. Knuth, Oren Patashnik. *Concrete Mathematics* .

[3] 金策，《生成函数的运算与组合计数问题》，2015年信息学奥林匹克中国国家队候选队员论文

[4] 胡渊鸣，《浅析信息学竞赛中概率论的基础和应用》，2013年信息学奥林匹克中国国家队候选队员论文

[5] 魏太云，《一道抛硬币问题的不同解法和比较》

《后缀树结点数》命题报告及一类区间问题的优化

长沙市长郡中学 陈江伦

摘 要

后缀数据结构是当前在算法竞赛中被广泛应用的一个体系。它包含后缀数组、后缀自动机、后缀树等广为人知的内容。这些后缀数据结构能高效地处理很多的问题。

作者对近年来出现的有关后缀数据结构算法进行了研究，并命制了试题《后缀树结点数》，利用后缀树与后缀自动机的性质得到了一个初步的做法。在集训队作业和冬令营营员交流中，作者就此做法与其它同学进行了探讨，并和其他同学一起研究出了更高效、更简洁的算法。

在研究本题的算法时，作者发现，对于后缀树维护动态树这一类字符串问题，可以使用启发式合并来代替动态树，从而让这类模型能够更为简洁地实现。同时，这类方法可以拓展到更多区间问题。

本文详细阐述了命题思路的来源以及一步步优化过程，并提出了一类区间问题的优化方案，希望能帮助更多的人深入了解后缀数据结构的相关性质并学会一类区间问题的优化方法。

1 引言

在 2016 年一场ACM-ICPC区域赛中，出现了一类新颖的动态树维护后缀树的模型，这类模型可以高效解决多次询问一个字符串的某个子串的本质不同子串个数、求一个字符串的某个子串内最长的出现至少两次的子串长度等一类问题，引起了大家对这方面的关注。

在 2017 年的国家候选队互测中，出现了这个问题的加强版：求一个子串内以某个固定字符开头和某个固定字符结尾的本质不同子串的个数。

同时，近年来也频繁出现一类题型：已知解决某个问题的一个算法，求这个算法在运行时的某些信息。例如ZJOI2017Day1T2中，已知一个用于支持单点加法、区间求和的算法，题目要求的不是这个算法得出的答案，而是求按照这个算法的过程计算出正确结果的概率。

后缀树是解决字符串问题的一个有力工具，作者由此想到，假如不要求用后缀树解决一般字符串问题，而是询问对一个串建立后缀树之后得到的结点个数呢？

由此作者展开了对这个问题的研究，并命制了本题。

本文的结构如下：

- 第二到四节是一些基本的定义、对题目信息的描述及得分情况。
- 第五到十节将介绍 3 个部分分算法以及 3 个满分算法。
- 在第十一节，提出了用启发式合并来代替动态树，从而优化动态树维护后缀树等一类区间问题的代码复杂度的方案。并且阐述了这个方案的一些应用。

2 题目描述

给定一个长度为 n 的字符串 P ，有 m 次询问，每次给定两个参数 l, r ，询问子串 $P[l, r]$ 所构成的后缀树的结点数。

2.1 数据范围

本题中用数字来表示字符串，不同的数字代表不同的字符，相同的数字代表相同的字符，数字的范围在 $[0, n]$ 之间。

对于 20% 的数据满足： $n \leq 500, m \leq 500$

对于 40% 的数据满足： $n \leq 3000, m \leq 3000$

对于 50% 的数据满足： $n \times m \leq 3 \times 10^8$

对于另 10% 的数据满足：字符串为全 0 串

对于另 20% 的数据满足：每一个字符是 $[0, n]$ 内的随机非负整数，每一个询问的 l 和 r 都是随机生成。

对于 100% 的数据满足： $n \leq 10^5, m \leq 3 \times 10^5$ ，字符串的每个数字都 $\leq n$

2.2 资源限制

时间限制 $3s$ ，空间限制 $1GB$ 。

2.3 输入格式

第一行两个正整数 n, m ，表示字符串长度和询问个数。

第二行 n 个用空格隔开的非负整数，表示这个字符串。

接下来 m 行，每行两个正整数 $l, r (l \leq r)$ ，表示询问子串 $[l, r]$ 的后缀树的结点数。

2.4 输出格式

输出 m 行，每行一个正整数表示答案。

2.5 样例输入

7 1
2 1 3 1 3 1 4
1 7

2.6 样例输出

10 [4]

2.7 样例说明

令数字 1 表示字符 a，数字 2 表示字符 b，数字 3 表示字符 n，数字 4 表示字符 s，则样例中的串的后缀树如下图：

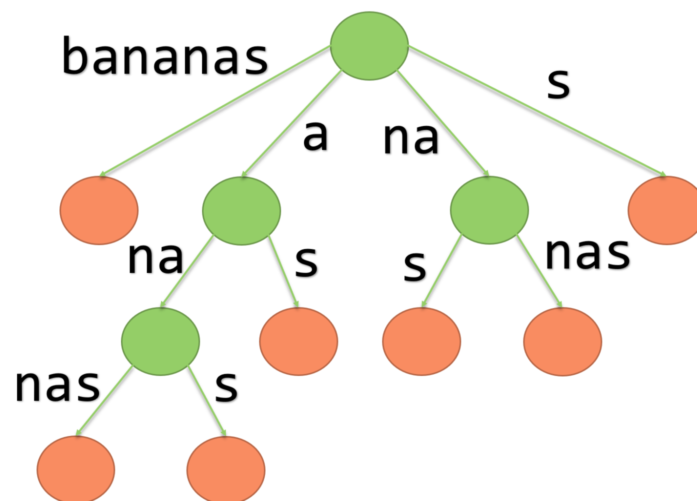


图1 样例中的后缀树

[4] 本题计算时忽略根结点

3 得分分布

共有 59 人参与此次训练，集训队及精英培训成员共有 4 人获得满分， 1 人获得 60 分， 1 人获得 50 分， 2 人获得 40 分， 2 人获得 30 分， 1 人获得 20 分。

4 文中所涉及的概念

对于一个字符串 S ，用 $S[i]$ 或 S_i 表示 S 的第 i 个字符，用 $S[l,r]$ 表示 S 位于区间 $[l,r]$ 的子串，用 $|S|$ 表示字符串 S 的长度。

定义字符串 S 的一个子串 $S[a,b]$ 的后继字符为 $S[b+1]$ 。特别地，如果 $S[b]$ 为字符串的末尾，则 $S[a,b]$ 的后继字符未定义。

4.1 hash

hash是处理字符串问题的一个常用工具，它所做的是一个压缩映射。本文将利用hash把每一个字符串都映射到一个整数上，把判断两个字符串是否相等的问题转化为判断两个整数是否相等。

4.2 后缀字典树

一个具有 $|S|$ 个字母的字符串 S 的后缀字典树为一棵包含根结点的有向树，每个结点代表一个字符串，不同的结点代表不同的字符串，且每一个非根结点所代表的串都是 S 的一个子串，根结点表示空串。在树上，一个结点的祖先所代表的串是这个结点所代表的串的一个前缀。

4.3 后缀树

后缀树是对后缀字典树的一个压缩。

在 S 的后缀字典树中，如果一个结点代表的串是 S 的一个后缀，就称这个结点为**后缀结点**。

在后缀字典树的基础上，把每一个**包含恰好一个孩子的非后缀结点**与它的孩子结点合并后，得到的就是后缀树。

在后缀树中，用fa_k 表示结点 k 在树上的父亲。last_k 表示结点 k 所代表的字符串的最后一次出现的位置。

4.4 后缀自动机

一个串 S 的后缀自动机是一个能接受 S 的所有后缀的有限状态自动机。

后缀自动机的定义及构造方法在算法竞赛中已经十分普及，故本文不再阐述，不了解的读者可以参见参考文献[1]。

后缀自动机的所有结点构成的树结构被称为parent树。

4.5 动态树

动态树是现在算法竞赛中的常用数据结构，用于动态维护一棵树的链剖分。

动态树的基础内容已经十分普及，不了解的读者可以参见参考文献[6]。

5 算法一

根据后缀字典树来构造后缀树：先用一个字符串的所有后缀建立一棵字典树，然后将所有**有且仅有一个孩子的非后缀结点**与它的孩子合并。

该算法的时间复杂度为 $O(n^2m)$ ，空间复杂度为 $O(n^3)$ （因为字符集大小为 n ，所有后缀的字典树结点数为 $O(n^2)$ ）。

期望得分： 20 分。

6 算法二

对于 $n, m \leq 3000$ 的数据，可以对于每一次询问都以较优的复杂度独立求出每个串的答案。

引理 6.1. 后缀自动机的parent树是反串的的后缀树。

这个性质在字符串问题中经常用到，在参考文献[2]中也有所提及。

根据这一性质，可以把串反过来构造后缀自动机。构造后缀自动机可以使用线性的增量法。

该算法时间复杂度 $O(nm\log n)$ ，空间复杂度 $O(n)$ 。因为本题字符集大小很大，所以需要使用数据结构(如C++中的map)来维护每个点的出边。时间复杂度带一个 log 。

期望得分 40 分。

7 算法三

实际上后缀自动机转移边的存储可以采用hash来优化算法二的数据结构，这样可以做到时间复杂度 $O(nm)$ 。空间复杂度仍然是 $O(n)$ 。

期望得分 50 分。

8 算法四

8.1 初步分析

用 S 来表示要求串 S 的的后缀树结点数。

后缀树的结点可以分为两类：后缀结点和非后缀结点。后缀结点恰好有 $|S|$ 个，这一部分只和字符串长度有关。

由于后缀字典树的叶子结点一定是后缀结点，所以非后缀结点一定至少有一个孩子，而如果一个非后缀结点恰好有一个孩子，那么在后缀树中该结点就会和孩子合并。这样，需要被统计进答案的结点满足至少有两个孩子。

后缀字典树上一个结点代表的是 S 的一个子串，如果结点 k 有至少两个孩子，说明至少存在两个不同字符 c_1, c_2，使得结点 k 所代表的子串 t 满足：tc_1 和tc_2 也为 S 的子串。

8.2 复杂的做法

作者在发现上述性质后得到了一个时间复杂度为 $O(n\log^3 n + m\log^2 n)$ 的做法，但是这个做法过于复杂，且效率也很低下，难以理解。同时这个方法对理解之后的做法并无影响，故本文不再阐述。有兴趣的读者可以在参考文献[5]中找到这个做法的详细介绍。

期望得分 100 分。

9 算法五

9.1 动态树维护后缀树模型的应用

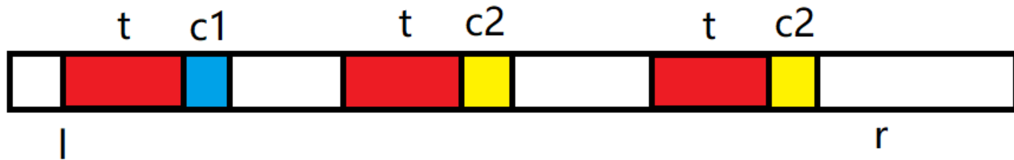
本题的难点就在于多组询问，每组询问的是读入串 P 的一个子串 $P[l,r]$。考虑从右往左枚举询问的左端点 l，对于每一个 r，动态维护串 $P[l,r]$ 的后缀树结点个数。

当 l 往左移动一位时，对于每个串 $P[l,r]$，考虑 $P[l,r]$ 与 $P[l+1,r]$ 相比发生了哪些变化。

根据刚刚发现的性质，对于 $P[l,r]$ 的一个前缀 $P[l,v]$，令这个串为 t，如果在 $[l,r]$ 中，存在两个字符 c_1, c_2 使得 tc_1 和 tc_2 都为字符串 $P[l,r]$ 的子串，且字符串 $P[l+1,r]$ 不满足这个条件，则字符串 t 会对 $P[l,r]$ 的答案新产生 1 的贡献。在左端点从 $l+1$ 移动到 l 之后，需要立即找到满足此要求的 v。

既然 tc_1 和 tc_2 都为 $P[l,r]$ 的子串，说明 t 这个串一定在 $P[l+1,r]$ 中出现过，由于 $P[l+1,r]$ 中，t 没有对答案产生贡献，说明对于 $P[l+1,r]$ 中 t 出现的任意一个位置 $P[a,b](b-$

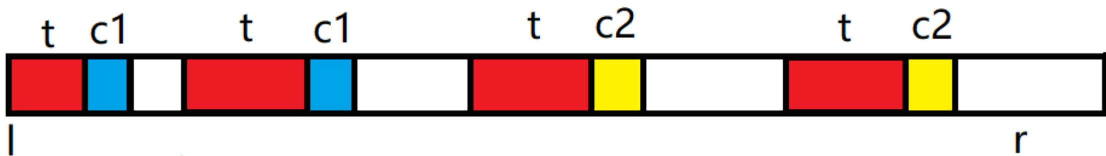
$a+1=|t|, P[a,b]=t)$，满足要么 $b=r$，要么这些串的后继字符（即 $P[b+1]$）都相同。



如图即为上文所描述的情况。

字符串 $P[l,r]$ 的所有前缀 $P[l,v](l \le v \le r)$ 在后缀字典树上对应的是一条从根出发到某个结点的一条链。假设这条链上的结点依次为 $x1, x2, \ldots, x_{r-l+1}$。需要统计的就是有多少个结点 x_i 满足 x_i 存在一个孩子不为 x_{i+1}，如果存在，因为 x_{i+1} 是 x_i 的孩子，所以 x_i 就有了至少两个孩子。

每次加入以当前 l 开头的所有前缀时，就给所有 $x_1, x_2, \ldots, x_{r-l+1}$ 打上一个访问标记。**定义一个结点的实儿子为所有儿子中最后被访问到的儿子。**如果在某次操作后一个结点 k 的实儿子没有发生变化，说明结点 k 所代表的串 t 的后继字符与上一次出现这个串时的后继字符相同，如图所示：

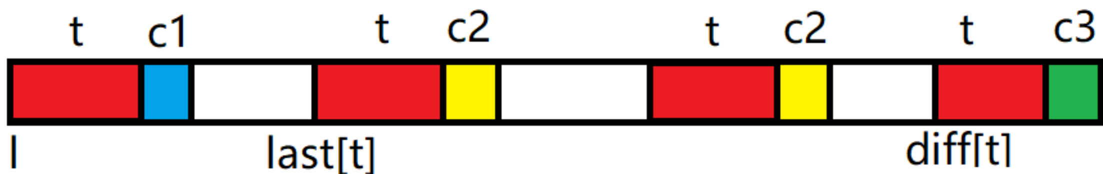


图中当前字符串 $t=P[l,l+|t|-1]$ 的后继字符和上一次出现 t 时的后继字符都是 $c1$。

注意到判断一个串是否对答案产生新贡献的条件是**当前是否出现了至少两种后继字符**，且之前只出现了一种后继字符，那么如果 $P[l,r]$ 的一个前缀的后继字符与最后一次出现时的后继字符相等，则不可能对答案产生新的贡献。

也就是说，只有实儿子发生变化的结点有可能会对答案产生新贡献。注意到这里实儿子的定义和动态树中的实儿子是一样的，且每次打标记就相当于动态树的access操作。根据动态树的复杂度分析，实儿子的切换次数是均摊 $O(\log n)$ 的，所以这就与经典的动态树维护后缀树模型联系在了一起。

具体地，用last_t表示字符串 t 上一次出现的位置，用diff_t表示字符串 t 上一次出现**不同后继**的位置，如图所示：



对于子串 $P[l+1,r]$，当 $r \ge \mathrm{diff}_t + |t|$ 时，t 对 $P[l+1,r]$ 的答案产生了贡献，当 $r < \mathrm{diff}_t + |t|$ 时，t 未对答案产生贡献。

对于子串 $P[l,r]$ ，当 $r \geq \text{last}_t + |t|$ 时， t 对 $P[l,r]$ 的答案产生了贡献，当 $r < \text{last}_t + |t|$ 时， t 未对答案产生贡献。

那么，从 $P[l+1,r]$ 变化到 $P[l,r]$ ，当 r 位于区间 $[\ \text{last}_t + |t|, \text{diff}_t + |t|)$ 中时，答案会新产生 1 的贡献，这就是一次区间加法操作。

这样，就可以使用动态树和线段树来支持access、查找实儿子发生变化的结点、区间加法操作了。

9.2 小结

以上内容就是一类经典模型在本题中的应用。

但这只是本题的一部分，接下来还面临另外一个问题。

9.3 处理重复计算的答案

在计算后缀树结点数时一共把结点分为两类：后缀结点和非后缀结点。而之前统计的是后缀结点+后缀字典树上至少有两个孩子的结点。其中，既是后缀结点又有两个孩子的结点被计算了两次，根据容斥原理，只需要减去同时满足这两个条件的结点个数，就得到了最终答案。

在询问串 $P[l,r]$ 时，如果存在某个 v 使得串 $P[v,r]$ 在 $P[l,r]$ 中出现了至少两个不同的后继，则这个串会被重复计算。

9.4 关键性质

经过分析可以发现这样一个关键性质：**如果一个串 t 在 $P[l,r]$ 中有两个不同的后继，那么 t 的所有后缀都至少有两个不同的后继。**

根据这个性质，被重复计算的串的长度就满足单调性，这样就可以通过二分计算出最长的被重复计算的串的长度。

二分之后，问题就转化为给定一个串，判断这个串在一段区间中是否有两个不同的后继。

9.5 线段树合并

有两个不同的后继意味着后缀字典树上有两个不同的孩子。所以就是要询问一个结点是否有两个孩子所在子树存在 $[l,r]$ 内的后缀结点。

由于实儿子是一个结点的孩子中最后一个被访问到的孩子，如果实儿子代表的串没有在区间 $[l,r]$ 中出现，说明该结点所代表的串在 $[l,r]$ 中没有任何后继，一定不满足条件。如

果实儿子代表的串在区间 $[l, r]$ 中出现，接着就要判断其它儿子所在子树是否也存在一个后缀结点在 $[l, r]$ 内，这个问题可以使用线段树合并解决。

使用线段树合并来维护每棵子树中的后缀结点所代表串的开头位置集合，就能够 $O(\log n)$ 查询一棵子树中，在区间 $[l, r]$ 内的后缀结点数量。判断除了实儿子之外，还有没有其他孩子的子树中存在位于 $[l, r]$ 的后缀结点，这等价于查询当前结点子树内位于 $[l, r]$ 的后缀结点数量 − 实儿子子树内位于 $[l, r]$ 的后缀结点数量是否为 0 。

以上就是算法五的全部内容。一共用到了动态树+线段树和二分+线段树合并，总复杂度是 $O(n\log^2 n + m\log^2 n)$ 。

期望得分 100 分。

10 算法六

该题其实可以进一步优化，注意到，处理问题的过程是枚举询问左端点 l ，然后求以当前左端点为询问左端点的所有询问的答案，那么，对于一个字符串 t ，在 l 固定的情况下，只需要找到一个最小的 r ，使得 t 在 $[l, r]$ 中出现了不同的后继。而这个 r 就是之前维护的$\mathrm{diff}_{\mathrm{node}(t)} + |t|$，node (t) 为字符串 t 在后缀树中对应的结点。

二分一个串后，可以利用hash表来 $O(1)$ 查询一个串在后缀树中的位置，这样就可以 $O(1)$ 判断一个串是否在区间中有两个不同后继。

这样时间复杂度就优化到了 $O(n\log^2 n + m\log n)$ 。

期望得分 100 分。

11 一类区间问题的优化

作者在研究动态树维护后缀树的模型时发现，实儿子的切换本质上就是，对于一个串 t ，把它出现的所有位置找到，按出现位置的右端点从小到大的顺序排成一个序列 $pos_1, pos_2, \ldots,$ pos_v ，对于这个序列任意相邻两个元素pos_i 和pos_{i+1} ，如果这两个位置的后继字符不同，则说明这两个串所对应后缀的叶子结点在后缀树上位于 t 对应的结点的不同孩子的子树中，则在询问左端点从pos_{i+1} 枚举到pos_i 时，后缀树上实儿子会发生一次切换。

既然维护了这么一个序列，也就可以求出last和diff而无需动态树了。

用启发式合并来求每个子树的pos序列。同时启发式合并也可以证明本题实儿子切换的复杂度。复杂度就是满足pos_i 和pos_{i+1} 后继字符不同这种情况的出现次数。

定义所有孩子中子树后缀结点个数最多的为重儿子，其它儿子为轻儿子。如果先只考虑重儿子，pos_1 到pos_v 的后继字符都相同，然后把轻儿子的后缀结点一个一个加入，每加入一个结点插入到pos序列中，最多和前后相邻两个元素产生 2 个新的断点。由于每个数作为轻儿子最多被插入 $O(\log n)$ 次，所以总复杂度就是 $O(n\log n)*$ 区间加法复杂度的。

由于作者在第一步就想到了用启发式合并来代替动态树，所以没意识到之后的部分可以直接用算法六中的方法来判断。当然，由于启发式合并也是可以求出对应的diff和last的，所以算法六仍然可以使用启发式合并来实现。

这样一个方法不仅适用于本题，还可以在很多其它题目中被使用到。

11.1 例题一

11.1.1 题目描述及数据范围

给定一个串 $P[1,n]$ ， m 次询问，每次询问一个子串 $[l,r]$ 中本质不同的子串个数。

$n,m \le 10^5$ 。

11.1.2 分析

根据本文提供的思路，从右往左枚举询问的左端点 l ，然后维护每个右端点的答案。

仍然考虑当询问左端点从 $l+1$ 变成 l 时答案的变化量。对于一个串 $P[l,v]$ ，如果这个串上一次出现的位置的右端点 $> r$ ，而且当前出现位置的右端点 $v \le r$ ，则新对答案产生 1 的贡献。同样用last_t 表示字符串 t 最后一次出现的位置，那么对于任意 $v \in [l,n]$ ，右端点在 $[v, \text{last}_{P[l,v]}-1]$ 的询问的答案就要增加 1 。

接着，如果串 $P[l,v]$ 和串 $P[l,v+1]$ 的 $last$ 相同，则可以放在一起处理，相当于给一个区间加上一个公差为 1 的等差数列。

根据动态树的性质，所有位于同一条实链上的点，最后一次被访问的时间一定是相同的，也就是说这些结点所代表的串的 $last$ 一定相同。这样同一条实链上的点可以一并处理。每次access时访问的实链条数是均摊 $O(\log n)$ 的，那么总时间复杂度就是 $O(n\log^2 n)$ ，空间复杂度 $O(n)$ 。

在access时，会找到一个结点到根的所有实链，实链与实链之间的变化点，就会发生一次实儿子切换。之前本文谈到，启发式合并是可以查询到所有实儿子切换的事件的。那么就可以用启发式合并来代替动态树。

11.2 小结

使用到这个优化的字符串问题通常有如下特点：

(1) 会输入一个大的模板串，每次询问的是这个串的一个子串的信息。

(2) 能够在回答询问前求出整个大模板串，不能在模板串后**强制在线**加入字符。

(3) 所求的答案与每个**本质不同的子串**及其在模板串中的前驱后继有关。

11.3 拓展

进一步的研究这个问题的过程，作者发现这类方法可以拓展到更多的区间问题。

11.4 例题二

11.4.1 题目描述及数据范围

给定一棵 n 个点的树，有 m 组询问，每次询问对编号在 $[l,r]$ 内的所有结点构造虚树之后的结点数。

$n,m \leq 10^5$

11.4.2 分析

对于询问 $[l,r]$，考虑每个点 k 是否会对答案产生贡献。k 位于虚树中，要么满足 $l \leq k \leq r$，要么，k 至少有两个孩子的子树内存在一个结点位于区间 $[l,r]$ 中。

仍然用类似的思路，从右往左枚举左端点 l，然后动态维护每个右端点 r 的答案。每次把当前的左端点 l 进行一次access操作，发生虚实边切换就对应着询问在一段区间中的右端点 r 的答案要新增加 1。

也可以使用启发式合并，把 k 的子树中所有结点按照标号排序，当 $[l,r]$ 中含有两个来自不同孩子子树的结点时 k 就会位于虚树中。在本节的开头已经证明，总段数是 $O(n\log n)$ 的。所以，本题可以在 $O(n\log^2 n) + O(m\log n)$ 的时间内解决。

12 总结

本题是一道以后缀树结点数为模型的题目，考察选手对后缀树性质的分析能力以及对经典模型的应用能力。涉及的后缀自动机，后缀树，线段树，动态树，启发式合并都属于NOI选手的知识范围。

但本题注重考查选手对于后缀数据结构性质的分析，比如后缀字典树与后缀树相比，每一个没有被压缩的结点所对应的子串在原字符串中，要么是一个后缀，要么具有两个不同的后继字符。

本题由两个部分组成，每一部分都要经过深入的思考才能得到高效的解决方案，对于选手而言难度是非常大的。

本题最初是在研究后缀自动机的性质时所作，一开始就把问题直接放到了后缀自动机上，并得到了非常复杂的算法四。把本题放到集训队作业中，则是试图寻找更简单的做法。

之后安徽师范大学附属中学的朱震霆同学发现了一个重要的性质，即本文中提到的，被重复计算的串的长度满足单调性，通过二分极大简化了做法。并在今年冬令营的营员交流上，和成都七中的王修涵同学共同分享了这个思路。也就是本文的算法五。

在营员交流后，作者和北京第八十中学的张宇博同学进行了深入的探讨，发现只需要用last和diff以及hash表就可以 $O(1)$ 判断一个串在一个区间中是否出现过不同后继，这就是本文的算法六。

在研究这些算法的过程中，作者发现，动态树维护后缀树的本质就是把一个字符串出现的所有位置找到，并按照它们的后继字符进行划分，相邻且后继字符相同的串被划分到同一段，不同的段之间就会发现last 和diff的变化。这样就可以使用启发式合并来代替动态树，简化了代码复杂度。同时发现这个方法可以拓展到其它一些区间问题。

希望通过本题，大家能对后缀自动机和后缀树的性质有更深入的了解，并熟练掌握一类区间问题的巧妙解法。

13 感谢

感谢中国计算机学会提供学习和交流的平台。

感谢国家集训队教练张瑞喆的指导。

感谢长郡中学的谢秋锋老师给予的关心和指导。

感谢王修涵、朱震霆和张宇博对本题优化思路给予的帮助。

感谢罗雨屏、郭晓旭、杨懋龙、刘承奥、张晴川、吴航为本文审稿。

感谢家人对我的关心和支持。

参考文献

[1] 陈立杰，后缀自动机，2012年冬令营营员交流。

[2] 张天扬，后缀自动机及其应用，2015年中国国家候选队论文集。

[3] 洪华敦，《基因组重构》命题报告，2017年中国国家候选队论文集。

[4] String，2016 ACM/ICPC Asia Regional Qingdao Online。

[5] 陈江伦，IOI2018国家集训队作业自选题解题报告。

[6] 黄志翱, 浅谈动态树的相关问题及简单拓展, 2014年中国国家候选队论文集。

浅谈保序回归问题

南京外国语学校 高睿泉

摘 要

本文是作者2018年的中国信息学国家集训队论文。本文主要介绍保序回归问题，首先通过特例介绍 $L_p(1 \le p < \infty)$ 问题在信息学竞赛中的常见做法，接着介绍使用整体二分法解决其一般问题的新颖想法及优化，最后简单介绍 $L\infty$ 问题的解法和拓展。

1 引言

保序回归问题是信息学竞赛中讨论较少、难度较高且可以通过不同思路构造解法的一类问题。在我们的实际生活中，保序回归问题常常出现于药剂中毒预测、衣服尺码设定等统计问题中。

本文讨论的保序回归问题，依赖于较强的数学结论。其中一类通过数形结合推导得出一般性结论；另一类则通过构造新问题，利用整体二分法解决，其思路突破竞赛中常规最优值问题的一般思路，能给选手们带来不小的启发。

文中第二节将介绍保序回归问题（ L_p)的基本概念。第三到五节将详细介绍 $L_p(1 \le p < \infty)$ 的做法：其中第三节将从贪心和维护折线算法的角度，介绍一些特殊情形下的解法；第四节将详细介绍利用整体二分法解决一般保序回归问题；第五节将介绍上述做法在特殊偏序结构上的优化。第六节将简单介绍 L_∞ 的解法和拓展。

2 保序回归问题

2.1 偏序关系

定义 2.1. 设 R 是集合 S 上的一个二元关系，若 R 满足：

- 自反性： $\forall x \in S$ ，有 $x\,R\,x$ ；
- 反对称性： $\forall x, y \in S$ ，若有 $x\,R\,y, y\,R\,x$ ，则有 $x = y$ ；
- 传递性： $\forall x, y, z \in S$ ，若有 $x\,R\,y, y\,R\,z$ ，则有 $x\,R\,z$ 。

则称 R 为 S 上的非严格偏序关系，记作 $\leq$ 。

2.2 问题描述

给定正整数 p 、一张点集为 $V = \{v_1, v_2, ..., v_n\}$ 、边集 E （ $|E| = m$ ）的有向无环图 G ，及代价函数 $(y, w)(\forall i, w_i > 0)$ 。如果在 G 中有 v_i 到 v_j 的有向路径，那么就有 $v_i \leq v_j$ 的偏序关系。

求点值序列 f ，满足 $\forall v_i \leq v_j (i, j \in [1, n])$,均有 $f_i \leq f_j$ ，最小化回归代价：

$$\sum_{i=1}^{n} w_i |f_i - y_i|^p, 1 \leq p < \infty$$

$$\max_{i=1}^{n} w_i |f_i - y_i|, p = \infty$$

对于相同的 p ,这一类问题统称为 L_p 问题。

2.3 一些约定

定义 2.2. 将序列 z 中不超过 a 的元素变为 a ，不小于 b 的元素变为 b 称为**序列 z 向集合 $S = \{a, b\}$ 取整**。

定义 2.3. **点集 U 的 L_p 均值**为满足 $\sum_{v_i \in U} w_i |y_i - k|^p (1 \leq p < \infty)$ 或 $\max_{v_i \in U} w_i |y_i - k| (p = \infty)$ 最小的 k 。

3 特殊情形下的算法

3.1 一种贪心算法

例 1. Approximation[5]

给定正整数序列 y, w ，求单调不减的实数序列 f ，最小化 $\sum_{i=1}^{n} w_i (f_i - y_i)^2$ 。

$n \leq 200000$

引理 3.1. 点集 U 的 L_2 均值为其加权平均数 $\frac{\sum_{v_i \in U} w_i y_i}{\sum_{v_i \in U} w_i}$ 。

证明. 可以简单地使用导数证出。 □

引理 3.2. $\forall 1 \leq i < n$ ，如果有 $y_i > y_{i+1}$ ，那么最优解中一定满足 $f_i = f_{i+1}$ 。

[5]题目来源：改自ASC38 Problem A，原题中 w_i 均为 1

证明. 反证在最优解中 $f_i < f_{i+1}$ 。我们选择极小的 $\epsilon > 0$ ，得到的 $f_i' = f_i + \epsilon w_{i+1}, f_{i+1}' = f_{i+1} - \epsilon w_i$ 不会影响偏序关系。那么有

$$\begin{aligned} & w_i(f_i' - y_i)^2 + w_{i+1}(f_{i+1}' - y_{i+1})^2 - w_i(f_i - y_i)^2 - w_{i+1}(f_{i+1} - y_{i+1})^2 \\ = & -2\epsilon w_i w_{i+1}(f_{i+1} + y_i - f_i - y_{i+1}) + \epsilon^2 w_i w_{i+1}(w_i + w_{i+1}) < 0 \end{aligned}$$

□

根据**引理3.1**、**引理3.2**，我们可以使用以下算法流程来解决：

1、维护一个单调栈，栈内每个元素为 (S_i, y_i', w_i') （其中 S_i 为集合， y_i', w_i' 为实数），在这里假设 m 为每一时刻栈的大小。初始时栈为空。

2、从左往右对于每个 $i(1 \le i \le n)$ ，加入元素 $(\{i\}, y_i, w_i)$ 。如果某一时刻有 $m > 1, y_{m-1}' > y_m'$ ，就将 (S_m, y_m', w_m') 和 $(S_{m-1}, y_{m-1}', w_{m-1}')$ 弹栈，并加入元素 $(S_{m-1} \cup S_m, \frac{y_{m-1}' w_{m-1}' + y_m' w_m'}{w_{m-1}' + w_m'}, w_{m-1}' + w_m')$ 。

3、答案 f_i 就为 i 所在集合 S_j 对应的 y_j' 。

上述算法的时间复杂度为 $O(n)$ 。这样的算法可以往 $1 \le p < \infty$ 的问题上拓展，只需把合并得到的加权平均数变成新集合的 L_p 均值即可。

这样的贪心可以向更为一般偏序结构上拓展吗？答案是不能的！因为**引理3.2**中不会影响偏序关系的结论不再成立，也就无法使用后面的算法。

3.2 维护折线算法

维护折线算法作为优化dp的重要方法，可以拓展到 G 构成一棵树的保序回归问题上。

例 2. 有向图[6]

给定一个 n 个点 $n-1$ 条边的有向弱连通图 $G = (V, E)$ ，每个点均有点权 d_i 和修改耗时 w_i 。对于每个 $i(1 \le i \le n)$ ，每次修改可以花费 w_i 的时间把 d_i 加 1 或减 1 ，求最少消耗多少时间，使得 $\forall (u, v) \in E, d_u \le d_v$ 。

$n \le 3 \times 10^5, 1 \le d_i \le 10^9, 1 \le w_i \le 10^4$

容易想到使用树形 dp 解决问题。

容易发现最后所有的取值一定在集合 $D = \{d_1, d_2, ..., d_n\}$ 中。设 D_j 表示 D 中第 j 小的元素， $g_{i,j}$ 为 i 号点调整至 j 时其子树内的最小修改耗时。动态规划的转移可以利用前缀最小值 $L_{i,j}$ 和后缀最小值 $R_{i,j}$ 优化。

对于叶子节点 i ，显然 g_i ， L_i ， R_i 均为斜率单调不减的折线。

[6]2018集训队互测 day2 T3 65分部分分

对于任意节点 i ，其 g_i 可以由其所有子节点的 L_i 或 R_i 以及折线 $y = w_i|x - d_i|$ 叠加得到，而 L_i 为将 g_i 斜率大于 0 的部分变成$y = \min\{g_{i,j}\}$ ， R_i 为将 g_i 斜率小于 0 的部分变成 $y = \min\{g_{i,j}\}$ ，所以可以归纳地证明出任意的 g_i, L_i, R_i 均为斜率单调不减的折线。

所以可以使用线段树维护 g, L, R 构成的折线每一段（ $x = D_j \sim D_{j+1}$ 的显然为一段）的函数式：

在 g_i 的计算中，其子节点的 L 和 R 的叠加可以用线段树合并[7]实现，叠加折线 $y = w_i|x - d_i|$ 则是区间加操作。

由于 g_i 对应的 L_i 和 R_i 不会同时对祖先的 g 产生影响，所以 L_i, R_i 的计算可以直接在 g_i 的线段树上二分并修改。

上述算法的时间复杂度为 $O(n \log n)$ 。

可以发现维护折线的方法仍无法往一般的偏序结构上拓展，其瓶颈在于需要整张图的偏序结构支持自底而上的dp过程。

4 一般问题的算法

4.1 另一种思路——从整体二分谈起

在信息学竞赛中，常常会碰到需要回答多组可以通过二分来回答的询问，如果每一组询问单独使用二分，往往会出现许多重复的计算。所以这时候可以使用整体的思想，通过相同的预处理，快速查询所有答案落在区间内的询问，并将每个询问划分出新的答案区间。这种想法在只知道区间内答案个数的交互题上也是适用的。

由于本文不涉及更为复杂的整体二分做法，具体拓展可以参见参考文献[1]。下面将通过一道例题帮助读者更好地理解整体二分：

例 3. Library[8]

已知桌面上有 n 本书从左往右排成一排，只知道编号为 $1 \sim n$ ，但不知道编号的顺序。管理员每次可以取走连续的若干本书。你每次可以向交互库询问一个编号集合 S ，交互库会返回管理员最少多少次把所有编号在 S 中的书取走。要求通过询问得出这 n 本书编号的顺序，由于左右顺序无法通过询问得出，只要得到从左往右或从右往左依次的编号即可。

$n \le 1000$ ，询问次数限制 20000 次

设 $ask(S)$ 为向交互库询问集合 S 的答案。

可以通过询问每本书相邻的编号得到最后的顺序，容易发现编号为 x 的书和集合 $S(x \notin S)$ 中 $ask(S) - ask(S \cup \{x\}) + 1$ 本书相邻。

[7]对于有懒标记的线段树合并，只需直接叠加标记，并保证线段树内每个节点不考虑祖先懒标记情况下信息正确即可

[8]题目来源：JOI2018 春季合宿 day4 T2

由于只需问出 $n-1$ 组相邻关系，可以考虑询问一个书 x 与哪些比它编号大的书相邻。考虑使用整体二分来解决：当整体二分到区间 $[l,r]$ 时，维护集合 $[l,r]$ 内与 x 相邻的书的数量，通过询问 x 与集合 $[l,mid](mid=\frac{l+r}{2}$, 下文同) 的相邻数，分治计算 $[l,mid]$ 和 $[mid+1,r]$ 中存在与 x 相邻的区间。

可以发现每个相邻限制只会用到 $\lfloor\log n\rfloor$ 次询问，且最多只会有 $\lceil\frac{n}{2}\rceil$ 本书浪费 2 次询问，故最多约 $2n\lfloor\log n\rfloor+n\approx 19000$ 次询问就可以得到答案。

4.2 新问题的构造

考虑在 L_p 问题基础上构造新的 $S=\{a,b\}(a<b)$ 问题：在满足原问题所有限制的同时，还要满足限制 $a\leq f_i\leq b$，最小化回归代价。

4.3 $p=1$ 的情况

引理 4.1. 在 L_1 问题中，如果任意 y_i 均不在区间 (a,b) 内，且存在一个最优解序列 z 满足其元素 z_i 均不在区间 (a,b) 内，若 z^S 为 S 问题的一组最优解，那么一定存在 z 是原问题的一组最优解且 z 可以通过向 S 取整得到 z^S 。

proof. 利用反证法证明。假设原问题最优解为 z' ，其中必有 $z'_i\leq a,z^S_i=b$ 或 $z'_i\geq b,z^S_i=a$ ，由于两类情况对应的点显然不会存在偏序关系，故可以不失一般性地证明不存在$z'_i\leq a,z^S_i=b$ 。

不妨设 $y=\max\{z'_i|z'_i\leq a,z^S_i=b\}$ ，集合 $U=\{v_i|z'_i=y,z^S_i=b\}$ 及 U 的 L_1 均值区间为 $[l,r]$ [9]。容易发现，在区间 $(-\infty,l]$ 内，集合 U 的回归代价为单调减函数，在区间 $[r,+\infty)$ 内，集合 U 的回归代价为单调增函数。

若 $l>a$ ，那么将 U 中的 z'_i 调整为 $\min\{l,b\}$ 后，原问题可以得到更优的解:由于 y 为 $z^S_i=b$ 中最大的，故不存在 $i,j(v_i\in U,v_j\notin U,z'_j\in[y,a])$ 满足 $v_i\leq v_j$ 的偏序关系，所以调整后的解依然合法。故原问题最优解的假设不成立。

若 $l\leq a,r\geq b$ ，同理 $l>a$ 的情况，把U中 z'_i 调整为 b 一定可以得到一组原问题可行解，又由于 $[a,b]$ 在最优区间内，故这组可行解不会比 z' 劣。这样调整之后，会发现 z' 向 S 取整后与 z^S 数值不同的位置数量减少，故一定可以经过有限次操作，将 z' 调整成取整为 z^S 的序列。故无法找到找到对应最优解 z 的假设不成立。

若 $l\leq a,r<b$ ，设 $U'=\{v_i|z'_i\leq a,z^S_i=b\}$ ，由于 $z^S_i=b$ ，故不存在 $i,j(v_i\in U',v_j\notin U',z'_j\in[-\infty,a])$ 满足 $v_i\leq v_j$ 的偏序关系。如果$U=U'$ ，将 U' 中 z^S_i 调整为 a ，显然调整后的序列一定优于 z^S ， z^S 为 S 问题最优解的假设不成立；下假设 $U'\setminus U$ 的 L_1 均值区间为$[l',r']$ ，如果 $r'\leq y$ ，那么 $U'\setminus U$ 中元素经过相同的调整后也可以得到比 z^S 更优的S问题解，故 z^S 为 S 问题最优解的假设不成立；如果 $r'\geq y$ ，可以把$U'\setminus U$ 的 z'_i 调整为y，能得

[9] L_1 均值可能不唯一，但所有值一定构成一个区间

到一组不比 z' 劣的原问题可行解，若这组解更优，z' 为原问题最优解的假设不成立，否则这组最优解一定满足 $U' = U$，重复上述证明过程即可找出矛盾。 □

根据**引理4.1**，通过一组 S 问题的最优解，就可以知道某一组原问题最优解中 z_i 与 a, b 的大小关系。对于 L_1 问题，由于最优解中 z_i 一定在 y_i 构成的集合$\{Y_1, Y_2, ..., Y_m\}(\forall i < m, Y_i < Y_{i+1})$ 中，故只需在该集合上做整体二分：当二分到区间 $Y_{[l,r]}$ 时，只要计算落在此区间内的 v_i 组成的 $S = \{Y_{mid}, Y_{mid+1}\}$ 问题的最优解，并根据 z_i^S，就可以将这些 v_i 分别划入新的选值区间 $Y_{[l,mid]}$ 和 $Y_{[mid+1,r]}$。这样对于每个点 v_i，只要通过 $O(\log n)$ 层计算即可确定选值。

4.4 $1 < p < \infty$ 的情况

引理 4.2. 当 $1 < p < \infty$ 时，任意集合 U 的 L_p 均值是唯一的。

证明. 设 $g(x) = \sum_{v_i \in U} |x - y_i|^p$，则有

$$g(x) = \sum_{v_i \in U, y_i \le x} (x - y_i)^p + \sum_{v_i \in U, y_i > x} (y_i - x)^p$$
$$g'(x) = p(\sum_{v_i \in U, y_i \le x} (x - y_i)^{p-1} - \sum_{v_i \in U, y_i > x} (y_i - x)^{p-1})$$

容易发现，随着 x 的增大，$\sum_{v_i \in U, y_i \le x} (x - y_i)^{p-1}$ 是单调增的，而 $\sum_{v_i \in U, y_i > x} (y_i - x)^{p-1}$ 是单调减的，故 $g'(x) = 0$ 至多有一个解。又由于当 $x > \max\{y_i\}$ 时，有 $g'(x) > 0$；当 $x < \min\{y_i\}$ 时，$g'(x) < 0$，故 $g'(x)$ 至少有一个零点。 □

引理 4.3. 如果 V 的任意非空子集的 L_p 均值均不在区间 $(a, b)(a < b)$ 内，且存在最优解 z 满足其元素 z_i 也均不在 (a, b) 内。若$\tilde{z}$为相同偏序集上代价函数为 (y', w') 的L_1 问题的一组最优解：

$$(y_i', w_i') = \begin{cases} (0, w_i \cdot [(b - y_i)^p - (a - y_i)^p]), y_i \le a \\ (1, w_i \cdot [(y_i - a)^p - (y_i - b)^p]), y_i > a \end{cases}$$

存在 z 是原问题的一组最优解满足$\tilde{z}_i = 0$ 当且仅当 $z_i \le a$。

证明. 由于保证任意子集的 L_p 均值不会落在 (a, b) 中，L_p 问题的 S 问题的最优解中不会有元素在区间 (a, b) 内，故 S 问题与上述 L_1 问题等价。后面的证明与**引理4.1** 相同，在此略去。 □

根据**引理4.2**，最优解中元素和所有的 L_p 均值的并集为有限集，故对于任意 a 一定存在 $\epsilon > 0$，满足区间 $(a, a + \epsilon)$ 中没有上述元素，并将**引理4.3**中的 L_1 问题中的 w_i' 变为位置 a 处的导数。由于信息学竞赛中只需要得出一定精度的最优解，就仍可以套用**小节4.3** 中整体二分的做法，只需将在集合上二分变为在实数中二分。

4.5 网络流模型的建立

对于一般偏序集上的 L_1 问题，其对应的 S 问题就可以看作是简单的 01 决策问题，而 $f_i \le f_j$ 的限制可以看作 f_i 选 1 时 f_j 不能选 0，这样就变成了经典的最小权闭合子图问题，可以用网络流来解决。

4.6 例题

例 4. ExtremeSpanningTrees[10]

给定一张 n 个点 m 条边的无向连通图 $G = (V, E)$ 和边集 $E_1, E_2 (E_1, E_2 \in E)$，每条边有初始的权值 d_i，每次操作可以把一条边的权值加 1 或减 1，求通过最少多少次操作可以满足：

- 边集 E_1 组成的子图是整张图的最小生成树
- 边集 E_2 组成的子图是整张图的最大生成树

$n \le 50, m \le 1000$

可以对 m 条边建立偏序关系，计算出生成树中非树边 i 和其对应树上路径中树边 j 的偏序关系，之后就变成了一般偏序集上的 L_1 问题。

5 特殊偏序结构上的优化

5.1 树上问题

对于**小节3.2**中树上的保序回归问题，也可以在整体二分的基础上套用一样的树形dp来解决 S 问题。

特别的是，这样的算法可以拓展到仙人掌上，在每个环上dp的时候只需枚举一个点的 01 选值，破环为链即可。

无论是树上还是仙人掌上的版本，时间复杂度均为 $O(n \log n)$（这里忽略计算代价中 $p-1$ 次方的复杂度，下文同）。

5.2 多维偏序

定义 5.1. d维偏序：对于 d 维点对 $(a_1, a_2, ..., a_d)$ 和 $(b_1, b_2, ..., b_d)$，如果 $\forall i (1 \le i \le d), a_i \le b_i$，那么就有 $(a_1, a_2, ..., a_d) \le (b_1, b_2, ..., b_d)$。

[10]题目来源：Single Round Match 720 Round 1 - Division I, Level Three

例 5. 给定 $r \times c$ 的网格纸，其中坐标为 (i, j) 的点，将权值加 1 或减 1 的修改代价为 $w_{i,j}$ ，初始权值为 $d_{i,j}$ 。求最小总修改代价使得其满足 $\forall (i_1, j_1) \le (i_2, j_2)$ ，均有 $d_{i_1,j_1} \le d_{i_2,j_2}$ 。

可以沿用整体二分的思路，考虑每层二分时 S 问题的一些性质：

1、每一列 i (x坐标)对应的y坐标都是连续的一段区间 $[l_i, r_i]$ 。

2、 l_i, r_i 均为单调不增序列。

3、如果第 i 列没有元素(即 $l_i > r_i$)，那么 $\forall j, k(j < i < k), l_j > r_k$ 。

根据上述性质，可以发现，每一层二分的时候，只需要考虑相邻两列之间的影响，可以直接设计dp状态： $g_{i,j}$ 表示第 i 列最小选择 b 的y坐标为 j 情况下前 i 列的最小代价总和。

转移为 $g_{i,j} = (\min_{k=j}^{r_{i-1}} g_{i-1,k}) + cost_{i,j}$ （ $cost_{i,j}$ 表示这种决策在第 i 列上的代价）。

容易发现可以用后缀min优化，时间复杂度为 $O(rc \log r)$ 。

例 6. 给定 n 个平面上的点，其中第 i 个点 v_i 坐标为 (x_i, y_i) ，将权值加 1 或减 1 修改代价为 w_i ，初始权值为 d_i 。求最小总修改代价使得其满足 $\forall v_i \le v_j$, 均有 $d_i \le d_j$ 。

与上一题不同的是，在每层的 S 问题中，不能忽视不相邻两列的代价影响，状态需要维护前 i 列中选 b 的y坐标最小为 j 的最小代价 $g_{i,j}$ 。

可以发现 $g_{i,j}$ 会有以下两种转移情况：

1、在第 i 列选择第一个 $\ge j$ 的位置 $next_j$ 作为第一个选 b 位置，转移为 $g_{i,j} = g_{i-1,j} + cost_{i,next_j}$

2、在第 i 列选择位置 j ，转移为 $g_{i,j} = (\min_{k \ge j} g_{i-1,k}) + cost_{i,j}$

容易发现第1种相当于区间加，第2种相当于区间求min、单点修改，均可以用线段树维护。由于整体二分的划分选值过程需要依赖 S 问题最优解的方案，故需要使用可持久化线段树来实现。时间复杂度为 $O(n \log^2 n)$ 。

6 L_∞ 问题的算法与拓展

6.1 简单的二分法

由于 L_∞ 问题要求最小化max，可以想到使用二分的方法来解决。每次二分之后，就可以确定每个点 v_i 的选值范围 $[a_i, b_i]$ ，之后可以在DAG上做dp，计算每个点 v_x 在仅满足所有 $f_j \le f_i (v_j \le v_i)$ 条件下 f_i 的最小值并检查是否可行。这样就可以得到一个一般 $L\infty$ 问题上的 $O(m \log \max\{y_i\})$ 的优秀做法。

6.2 问题的拓展

例 7. 有一个大小为 n 的点集 $V = \{v_1, v_2, ..., v_n\}$ 和代价函数 (y, w) ，对于 $\forall i, j(1 \le i < j \le n)$ ，均有 $v_i \le v_j$ 。

要求决策一个点值序列 f ，满足 $\forall v_i \le v_j (i, j \in [1, n])$ ，都有 $f_i \le f_j$ 。

要求对于每个 $k(1 \le k \le n)$ 求： $\max_{i=1}^{k} w_i|f_i - y_i|$ 最小为多少。

如果套用**小节6.1**中的二分做法，需要处理多组询问，由于不能快速进行预处理，故无法使用整体二分优化，所以我们需要寻找一些性质。

定义 6.1. $err(v_i, r)$ 表示在 v_i 处使用点值 r 的回归代价 $w_i|r - y_i|$ 。

$mean(v_i, v_j)$ 表示 v_i, v_j 的带权平均数 $\frac{w_i y_i + w_j y_j}{w_i + w_j}$ 。

当 $v_i \le v_j, y_i > y_j$ 时， $mean_err(v_i, v_j)$ 为 $err(v_i, mean(v_i, v_j))$ 和 $err(v_i, mean(v_i, v_j))$ 的较大值，否则 $mean_err(v_i, v_j)$ 为 0 。

引理 6.1. $mean_err(v_i, v_j)$ 是仅考虑 v_i 和 v_j 时的最小回归代价。

证明. 当有 $v_i \not\le v_j$ 或 $y_i \le y_j$ 时，显然当 $f_i = y_i, f_j = y_j$ 时，回归代价为0。

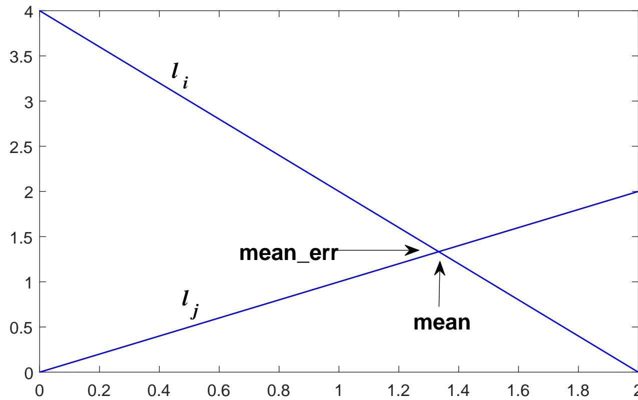


图 1: $y_i = 2, w_i = 2, y_j = 0, w_j = 1$ 下的图像

当有 $v_i \le v_j, y_i > y_j$ 时，根据Figure 1（其中 l_i 为函数 $y = err(v_i, x)$ 的左半支， l_j 为函数 $y = err(v_j, x)$ 的右半支），可以发现 $mean(v_i, v_j)$ 为集合$\{v_i, v_j\}$ 的 $L\infty$ 均值， $mean_err(v_i, v_j)$ 即为最小回归代价。 □

引理 6.2. 对于任意的 $L\infty$ 问题，整张图的最小回归代价为 $\max\{mean_err(v_i, v_j)\}$ 。

证明. 根据**引理6.1**，容易发现最小回归代价不小于 $\max\{mean_err(v_i, v_j)\}$ 。

下证最小回归代价不超过 $\max\{mean_err(v_i, v_j)\}$ 。

对于点 v_k ，点值 f_k 可以选择为 $mean_err(v_i, v_j)$ 最大的一组 $v_i, v_j (v_i \le v_k \le v_j)$ 的 $mean(v_i, v_j)$ 。下面分三步证明这样选值的正确性：

首先，可以发现如果另有一组 $v_{i'}, v_{j'} (v_{i'} \le v_k \le v_{j'})$ 满足 $mean_err(v_{i'}, v_{j'}) = mean_err(v_i, v_j)$ ，那么一定有$mean(v_{i'}, v_{j'}) = mean(v_i, v_j)$ 。否则不妨设 $mean(v_{i'}, v_{j'})$ 更大，根据函数图像的

性质，如果 $mean_err(v_{i'}, v_j)$ 小于等于 $mean_err(v_i, v_j)$ ，那么$l_{i'}$ 的斜率就要大于等于0，与 $l_{i'}$ 为 err 函数的左半支相矛盾。

接着证明 $err(v_k, mean(v_i, v_j))$ 不会超过 $mean_err(v_i, v_j)$ ：如果 $y = err(v_k, x)$ 的图像经过 l_i, l_j 交点以上的部分，那么 v_k 与 v_i 或 v_j 的$mean_err$ 就大于 $mean_err(v_i, v_j)$ ，与之前选择最大的一组相矛盾。

最后证明这样选值可以满足偏序限制。对于 $v_a \le v_b$ 的偏序关系，设 C_a, D_a 分别表示所有 $v_e(v_e \le v_a)$ 的 err 函数左半支构成的射线集合、所有$v_e(v_e \ge v_a)$ 的 err 函数右半支构成的射线集合， C_b, D_b 同理。容易发现 C_b 为从 C_a 中加入若干射线所得， D_b 为从 D_a 中删除若干射线所得。可以利用归纳法，把问题转化为证明在 C_a 中加入一条射线或在 D_a 中删除一条射线后，所有交点中 y 坐标最大点的 x 坐标不会变小。考虑 C_a 构成的下凸壳，y坐标最大交点一定在凸壳上，在 D_a 中删除一条射线只会让最大的y 坐标变小，对应的x坐标右移。同理可以证出在 C_a 中加入一条射线也具有相同的性质。 □

在**例7**中我们可以设 $pre(v_i) = \max\{mean_err(v_j, v_i) | v_j \le v_i\}$ ，可以发现只要算出所有的 $pre(v_i)$ 就可以得到所有 k 的答案。这样就得到了一个 $O(n^2)$ 的算法了。

继续考虑优化计算 $pre(v_i)$ 的过程，可以类似证明中的思路，维护 $v_1 \sim v_i$ 所有err函数的左半支构成的下凸壳，并查询 v_i 的err函数右半支与凸壳的交点。

一种较为简单的维护方法是把形如 $y = kx + b$ 的直线看成平面上的点 (k, b) ，维护这些点构成的上凸壳，这样每次插入就变成了二分位置、暴力删除已经不在凸壳上的点，查询就变成了二分查询直线与凸壳内直线交点与其对应凸壳上折线段的位置关系，这些都可以简单地用c++中的set维护。于是我们就得到了 $O(n \log n)$ 的优秀做法。

$L\infty$ 问题还有不少更为复杂的拓展，这里就不一一阐述了，如有兴趣可以参见参考文献[5]。

7 总结

本文从保序回归问题 L_p 的描述出发，分别介绍了 $1 \le p < \infty$ 和 $p = \infty$ 的做法：

在 $1 \le p < \infty$ 的部分中，先从贪心和维护折线算法的角度介绍了其特殊情况的解法，然后利用全新的整体二分思路，介绍了一般问题的做法。三种方法各有优劣，整体二分做法可以解决一般问题，同时能在特殊偏序结构上进行优化，但很难回答多组不同的询问；而贪心做法可以在特殊的 L_2 问题上做到线性复杂度，维护折线较整体二分有较好的常数，并且它们均可以同时计算出一些子问题的答案，但他们只能解决对应的特殊情况。

在 $p = \infty$ 的部分中，先介绍了简单的二分法，再通过数形结合的推导，得出了 L_∞ 问题的一般性结论，并将其运用到线性偏序结构的问题上。两种方法也是各有优劣，二分法可以解决任意偏序结构上的问题，但是无法同时支持多个问题的解决；而运用一般性的结

论虽然可以同时解决多个问题，但很难在一般偏序上快速计算。这提醒我们要根据题目选用合适的方法。

本文介绍的保序回归问题还值得进行更深入地研究。希望本文能够起到一个抛砖引玉的作用，启发读者进行进一步的研究，得到更多有趣的做法和应用。

感谢

感谢中国计算机学会提供学习和交流的平台。
感谢南京外国语学校的李曙老师的关心与指导。
感谢国家集训队教练张瑞喆和余林韵的指导。
感谢杨骏昭、杨乾澜、徐海珂同学与我讨论这些算法。
感谢王修涵、徐海珂同学为本文验稿。
感谢父母对我的关心和照顾。

参考文献

[1] 许昊然 ,《浅谈数据结构题的几个非经典解法》, 2013年集训队论文

[2] 张恒捷 ,《DP的一些优化技巧》,2015年集训队论文

[3] 维基百科：https://en.wikipedia.org/wiki/Isotonic_regression

[4] Quentin F. Stout , ”Isotonic Regression via Partitioning” , Algorithmica 66 (2013), pp. 93 – 112.

[5] Stout, Q.F. , ”Algorithms for L∞ isotonic regression” , submitted (2011).

《Fim 4》命题报告

浙江省杭州第二中学 吴瑾昭

摘 要

本文将介绍作者在集训队互测时命制的一道以字符串匹配为背景的题目《Fim 4》。题目深度挖掘了有关字符串周期的性质，并且将这个性质与大家常见的KMP算法结合起来。本题需要选手对字符串有比较深刻的理解，同时又需要一定代码能力。是一道考察选手思维深度的好题。

除此之外，题目的部分分设置合理，数据强度高，区分度高，能引导选手思考到正解。无论是只写了暴力的选手，还是稍微进行过思考的选手，以及深度研究找到了所有性质的选手，都能获得适度的部分分。

本文同时提供了一种命题思路：对于一种广为人知的算法，我们充分思考它的本质，找到原本算法中利用得较少的性质，从而把这个算法推广到更一般性的问题上。

1 试题大意及数据范围

1.1 原题目

1.1.1 原题目大意

给定 n 个字符串 s_i 和 m 个 $1 \sim n$ 之间的整数 a_i，令母串为 $s_{a_1} + s_{a_2} + \cdots + s_{a_m}$，回答 q 次询问，每次给出一个字符串 t_i，询问这个串在母串中的出现次数。

保证 s_i 和 t_i 都只由字母a,b组成。

1.1.2 原题目数据范围

对于所有的数据，满足：

$1 \le n, m, q \le 5 \times 10^4$

$\sum_{1 \le i \le n} |s_i|, \sum_{1 \le i \le q} |t_i| \le 10^5$ 。

- 子任务1（20分）：

  $\sum_{1 \le i \le m} \left|s_{a_i}\right| \le 10^6$

- 子任务2（20分）：

  $\max|t_i| \le \min|s_i|$

- 子任务3（30分）：

  $\max|t_i| \le 200$

- 子任务4（30分）：

  无特殊性质。

但实际上，作者给出的解法能解决比原题目更强的题意。加强版题目大意将在下面给出。不难发现原题目大意的询问严格是加强版题目大意的子集。所以我们只需要解决加强版的题目就能同时解决原题目。本论文也将主要介绍对于加强版题目的做法。

1.2 加强版题目

1.2.1 加强版题目大意

给定一棵有 n 个节点的Trie树。令根到第 i 个节点所形成的字符串为 s_i 。

再给定 m 个在 $1 \sim n$ 之间的整数 a_i 。

回答 q 次询问，每次给出一个字符串 t_i 和两个正整数 L_i, R_i 。令 $S_i = s_{a_{L_i}} + s_{a_{L_i+1}} + \cdots + s_{a_{R_i}}$ ，询问 t_i 在 S_i 中的出现次数。

字符集大小 ≤ 10 。

1.2.2 加强版题目数据范围

对于所有的数据，满足：

$n \le 10^6$

$1 \le m, q \le 5 \times 10^4$

$\sum_{1 \le i \le q} |t_i| \le 10^5$

- 子任务1（20分）：

  $\sum_{1 \le i \le m} \left|s_{a_i}\right| \le 10^5$

- 子任务2（30分）：

  $\max|t_i| \le 200$

- 子任务3（20分）：

  $\max|t_i| \le \min|s_{a_i}|$

- 子任务4（30分）：

  无特殊性质。

2 前置技能

这一部分我们将粗略介绍一些解决本题可能需要用到的定义与算法。

2.1 字符串基本概念

设一个长度为 n 的字符串 $s_1 s_2 \dots s_n$ 。对于 $1 \leqslant i \leqslant j \leqslant n$ ，称 $s_i s_{i+1} \dots s_j$ 为 s 的一个子串，记作 $s[i:j]$ ，记 $|s|$ 为 s 的长度，即 $|s| = n$ 。

对于 $1 \leqslant i \leqslant n$ ，称 $s[i:n]$ 为 s 的一个后缀，记为 $s[i:]$ 。类似地，对于 $1 \leqslant i \leqslant n$ ，称 $s[1:i]$ 为 s 的一个前缀，记为 $s[:i]$ 。

对于 $1 \leqslant i < n$ ，如果 $s[:i] = s[n-i+1:]$ ，则称 $s[:i]$ 是 s 的一个 border 。特别地，空串也是 s 的一个 border 。 s 最长的 border 称作 s 的 Lborder 。

如果 $2|Lborder(s)| \geqslant n$ ，则称 s 是周期串，周期的长度即 $|s| - |Lborder(s)|$ 。

2.2 一些记号

我们规定母串 S 为 $s_{a_1} + s_{a_2} + \cdots + s_{a_m}$ ， lp_i 表示 s_{a_i} 在 S 中的左端点位置， rp_i 表示 s_{a_i} 在 S 中的右端点位置。 L 为询问串最长的长度。 Σ 为字符集。

定义 $[cond]$ 的值当 $cond$ 成立时为 1 ，否则为 0 。

定义 $c(T,S)$ 表示字符串 T 在字符串 S 中的出现次数。

定义 $suf(S)$ 表示 S 的全体后缀组成的集合, $pre(S)$ 表示 S 的全体前缀组成的集合。

2.3 后缀自动机

一个串 S 的后缀自动机是一个有限状态自动机（DFA），它能且仅能接受所有 S 的后缀，并且拥有最少的状态和转移。我们简称为SAM。

对于一个串 S ，我们采用增量法构造，时间复杂度为 $O(|S|)$ 。具体构造方法和复杂度证明与本文主题无关我们不在此赘述，具体可见参考文献[5]。

2.4 SAM on Trie

对于一个Trie树，我们定义它的子串集合是它的根节点到任意一个节点所形成的字符串的后缀所形成的集合。

我们给Trie树建立一个有限状态自动机能识别Trie树上的所有子串，并尽可能的缩小自动机的状态数和转移数，我们称之为SAM on Trie。

SAM on Trie的性质与SAM在序列上的性质几乎完全一致。如：SAM on Trie可以识别一个Trie里所有的子串，SAM on Trie的parent树是一棵倒着的后缀树，每个点的父亲是这个点所代表的字符串里最长的本质不同的后缀（本质不同即在这个Trie里的出现次数不同）所代表的节点。

SAM on Trie的构造同序列上类似。每次每个节点以父亲在SAM上的节点为last来继续扩展。如果采用FIFO顺序构造，则时间复杂度为 $O(n|\Sigma|)$ 。其中 n 为Trie的节点数， Σ 为字符集。对于其复杂度的证明因为与本文主题无关我们不在此赘述，具体可见参考文献[1]。

2.5 KMP

使用 KMP 算法可以在 $O(|s| + |t|)$ 的时间内计算出字符串 t 在字符串 s 中的出现次数。除此之外，它还能计算出所有 $Lborder(t[:i])(1 \leqslant i \leqslant n)$ 。具体算法流程如下：

Algorithm 1 KMP Algorithm

```
1: j ← 0
2: np ← 0
3: for i ← 2 to |t| do
4:     while t_i ≠ t_{j+1} and j ≠ 0 do
5:         j ← p_j
6:     end while
7:     if t_i = t_{j+1} then
8:         j ← j + 1
9:     end if
10:    p_i ← j
11: end for
12: j ← 0
13: for i ← 1 to |s| do
14:     while s_i ≠ t_{j+1} and j ≠ 0 do
15:         j ← p_j
16:     end while
17:     if s_i = t_{j+1} then
18:         j ← j + 1
19:     end if
20:     if j = |t| then
21:         pos[np] ← i
```

22: $np \leftarrow np + 1$
23: $j \leftarrow p_j$
24: **end if**
25: **end for**

2.6 快速求两个串的最长公共前后缀

我们要解决一个这样的问题：给定一个Trie和一个字符串 T ，快速回答：给定Trie上某个节点，求一个最长的字符串 T' ，使得 T' 是 T 的前缀，并且 T' 是这个Trie的根节点到这个节点所形成的字符串的后缀。

我们先建立这个Trie的SAM，考虑对于 T 的每个前缀，我们都在它在Trie上对应的节点里打上一个权值为当前前缀长度的Tag。

当我们询问时，我们只需要找到Trie的根节点到这个节点所代表的字符串在SAM上的对应节点中，具有最大权值Tag的parent树上祖先即可。我们只需要用一棵LCT来维护parent树就可以在$O(\log n)$ 的时间内支持这个操作。

3 算法介绍

3.1 算法一

对于Subtask1，注意到母串的总长度在 10^5 范围内。所以直接构造出母串的时间复杂度是可以接受的。

首先我们先构造出母串的后缀自动机。

然后我们用可持久化线段树合并来预处理出parent树上每个节点的Right集合。

每一次询问时，我们只需要找到询问串 t 在parent树上对应的节点，查询它的Right集合里在区间 $[L + |t| - 1, R]$ 的个数即可。因为我们之前已经用可持久化线段树预处理了每个节点的Right 集合，所以我们只需要在这个节点对应的线段树上查询这个区间内的和即可。

时间复杂度 $O(\sum \left|s_{a_i}\right| \log n + \sum |t_i| + q \log n)$ 。

期望得分 20 分。

3.2 算法二

考虑Subtask2，注意到最长的询问串长度只有 ≤ 200 。

我们考虑到询问串在母串中的匹配，要么是询问串在单个字典串内出现，要么询问串横跨了若干个个字典串，我们对两部分分别处理。

3.2.1 Part 1

我们每次给出一个询问串 t ，要计算 $\sum_{i=L}^{R} c(t, S_i)$ 。这是一个比较常见的问题。我们对给定的Trie建出它的SAM on trie。

类似解法一，我们对于SAM上的每个节点，我们用可持久化线段树合并来预处理出parent树上每个节点所代表的字符串在 s_{a_i} 中的出现次数。

每次询问时找到询问串 t 在parent树上对应的节点，在这个节点上对应的线段树上查询区间 $[L, R]$ 的区间和就可以了。

这一部分的复杂度是 $O(n|\Sigma| + n\log n + \sum|t_i|)$ 。

3.2.2 Part 2

形式化地说，计算跨越多个字典串的出现次数也就是要求我们来计算 $\sum_{i=L}^{R-1} c(t, S[\max(lp_i, rp_i - |t| + 2) : \min(rp_R, rp_i + |t| - 1)])$ 。

我们可以把所有 $S[\max(lp_i, rp_i - L + 2) : rp_i + L - 1]$ 拿出来建Trie，并对该Trie建出SAM。对于前缀 $S[\max(lp_i, rp_i - L + 2) : rp_i + k - 1]$ 我们将其标号为 k ，且令它的下标为 $rp_i + k - 1$ 。统计 $\sum_{i=L}^{R-1} c(t, S[\max(lp_i, rp_i - |t| + 2) : \min(rp_R, rp_i + |t| - 1)])$ 也就是相当于在统计 parent树上 t 所代表的节点的子树中有多少节点的标号在 $[2, |t| - 1]$ 中，且下标在 $[lp_L + |t| - 1, rp_R]$ 中，对parent树求dfs序后可以转化为三维数点问题。

我们注意到这个三维数点的“标号”维是 $O(L)$ 级别的。并且总点数是 $O(mL)$ 级别的，询问是 $O(q)$ 级别的。我们可以轻松地通过离线做到加入$O(\log n)$ 询问 $O(L\log n)$ ，因此这一部分的总复杂度是 $O(mL\log n)$ 的。

综上所述，我们可以在 $O(n\log n + \sum|t_i| + mL\log n)$ 的复杂度内解决这个Subtask，期望得分30分。

3.3 算法三

对于Subtask3，注意到最长的询问串长度也小于等于最短的字典串长度。

这给我们带来了一个这样的性质，即询问串最多只会横跨两个字典串。

我们沿用算法二的想法，还是分别统计字典串内的和横跨字典串的。

我们设置一个 B ，对于长度 $\le B$ 的询问我们在用解法二中介绍的算法解决，而对于询问串长度 $> B$ 的算法我们用另一种算法来解决。

我们继续沿用算法二，考虑对于询问串长度 $> B$ 的询问。

定义 $\tau(s, t) = \mathrm{suf}(s) \cap \mathrm{pre}(t)$ 。

定理 3.1. 令 $\tau(s, t) = \{s[p_i :]\}$ ，其中 p 是一个单调上升的序列，则 p 可以被表示成 $O(\log n)$ 个等差数列顺次连接起来。

证明. $s[p_{i+1}:]$ 是 $s[p_i:]$ 的前缀，因此只有两种情形：

1. $\left|s[p_{i+1}:]\right| \le \left|s[p_i:]\right|/2$:长度减半，一共只会减半 $O(\log n)$ 次，不需要处理。

2. $\left|s[p_{i+1}:]\right| > \left|s[p_i:]\right|/2$: $s[p_i:]$ 必然是周期串，我们可以把形如 $p_k = p_i + (k-i)l$ 的位置压成一个等差数列，其中 l 是 period 长度。则处理后 $\left|s\left[p_j:\right]\right| < \left|s[p_i:]\right|/2$ （其中 $j > i$ 是第一个不在等差数列里的下标），同样也只会减半 $O(\log n)$ 次。 □

我们要计算每个串在 s_{a_i} 与 $s_{a_{i+1}}$ 之间的出现次数，可以先算出 $\tau(s_{a_i}, t)$ 和 $\tau(t, s_{a_{i+1}})$ ，然后把两部分结果合并起来。注意到这 $\log n$ 段等差数列的值域一定是互不相交的，因此我们可以用类似归并排序的办法来合并，每次合并复杂度 $O(\log^2 n)$ 。

于是问题来到怎么求 $\tau(s_{a_i}, t)$ 。（求 $\tau(t, s_{a_{i+1}})$ 是对称的，用同样的做法就可以了）

我们对于字典串建出Trie，然后再对这个Trie建出SAM。在 $O(m \log n)$ 的时间内对于每个 i 求出 $\max\{\mathrm{LCP}(s_{a_i}[j:], t)\}$ （具体计算方法在前置技能中有介绍）。这就是我们要计算的 p_1 。我们再用KMP预处理出 t 的next数组。注意到 $s_{a_i}[p_i:]$ 一定是 $s_{a_i}[p_1:]$ 的border。所以我们就可以利用这个next数组来构造出整个 p 数组。

当 B 取到 $\sqrt{m}\log^{1.5} n$ 的时候，整个算法的时间复杂度最小，为 $O(n|\Sigma|+n\log n+m\sqrt{m}\log^{1.5} n)$ 。

期望得分20分,结合算法一，二可以获得70分。

3.4 解法四

我们继续沿用解法二，我们考虑对于询问串 $> B$ 的询问。

我们先介绍两个引理：

引理 3.1. 如果 p, q 都是字符串 S 的周期，且 $p + q \le |S|$,那么 $\gcd(p, q)$ 也是字符串 S 的周期。

证明. 令 $d = q - p(p < q)$ ，则由 $i - p > 0$ 或 $i + q \le |S|$ 均可推出 $S_i = S_{i+d}$ 。 □

引理 3.2. 令 l 是周期长度，对于 $i > 2l, next_i = i - l$ 。

证明. 这等价于对于 $i > 2l$ 的前缀，它的最短周期是 l 。我们使用反证法：如果存在比 l 更短的周期 T ，那么 $T + l < i$,所以由引理3.1可得 $\gcd(T, l)$ 也是这个串的周期，注意到 $\gcd(T, l) | l$ ，这与 l 是整个串的最短周期矛盾，故得证。 □

考虑优化KMP算法。我们考虑把询问串放在母串上跑KMP。我们注意到我们只需要计算串之间的匹配就可以了。

我们令 i 表示当前KMP算法指向字典串的指针， j 表示当前KMP算法指向询问串的指针, L 表示询问串的长度。

如果存在某个 k ，使得 $i \in [lp_k, rp_k]$ 且 $i-j+1 \in [lp_k, rp_k]$,那么说明当前正在匹配的是串内的，是不需要被计算的。所以我们直接令 j 等于最长的询问串前缀的长度，并且这个询问串的前缀是 s_{a_k} 的一个后缀，再把 i 设为那个后缀开始的地方再继续KMP即可。

由于我们可以快速求出两个后缀的 LCP，因此我们可以很容易地计算下一次失配的位置。因此我们只需要优化跳 next 的过程即可。

在失配之后如果 $next_j < \frac{j}{2}$ 我们就直接执行 $j := next_j$ ，否则：

如果执行 $j := next_j$ 后还会失配，根据引理3.2，我们可以直接跳到 $l + j \bmod l$ 的位置（l 是周期长度）(Case (1))。

否则我们统计接下来 $S[i+1:]$ 还能继续匹配多少个 $t[:j]$ 的周期并找到失配时的位置，此时有两种可能的情况，第一种情况是 $S[i+1] = t_j$ (Case (2))，此时我们继续向下匹配，第二种情况同之前 Case (1) 。

有一个特殊情况是我们匹配到了串末。这时候我们可以计算 $S[i+1:]$ 还能匹配多少个 $t[:j]$ 的周期，能匹配的周期数就是对答案的贡献。之后我们和失配情况一样处理即可。

定理 3.2. 之前所述的算法的时间复杂度是 $O(m\log n)$ 的。

证明. 对于Case (1)，考虑 $j-(i-lp_k)$ 的值。显然当 $j-(i-lp_k) \le 0$ 时，我们就会重置 i 和 j 的值。

首先，求LCP时， $j-(i-lp_k)$ 的值并不会变。

而在跳next时，对于 $next_j \le \frac{j}{2}$ 时，执行 $j := next_j$ 会让 j 变小 $\frac{1}{2}$ 。对于 $next_j > \frac{j}{2}$ 时，当我们执行 $j := l + j \bmod l$ 时， $l + j \bmod l \leqslant \lceil\frac{2l}{3}\rceil$ ， j 会变小 $\frac{1}{3}$ 。所以每次Case（1）操作就会让 $j-(i-lp_k) \le 0$ 变小至少 $\frac{1}{3}$ 。

所以我们发现Case (1)的总复杂度是 $O(m\log n)$ 的。

对于Case (2)，我们先引入一个关于层的定义：

我们令 $b_i = \left[next_i \geqslant \frac{i}{2}\right]$ ，将 b 序列中从左到右第 i 段连续的 1 称作第 i 层。

我们再证明一个有关层的性质：

定理 3.3. 令第 i 层的前缀的周期为 per_i ，那么有 $\frac{3}{2}|per_i| \le |per_{i+1}|$

证明. 令 pos_i 表示第 i 层最右边的位置，那么显然 per_i 和 per_{i+1} 都是 $S[:pos_i]$ 的周期。

如果 $|per_i|+|per_{i+1}| \le pos_i$ ，那么根据引理3.2,可以知道 $\gcd(|per_i|, |per_{i+1}|)$ 也是 $S[:pos_i]$ 的周期。注意到$\gcd(|per_i|, |per_{i+1}|)$ 整除 $|per_{i+1}|$ ，这与 per_{i+1} 是第 $i+1$ 层的最短周期矛盾，故我们得到 $|per_i| + |per_{i+1}| > pos_i$

又有 $pos_i \ge 2|per_i|$ ，联立两式可得 $\frac{3}{2}|per_i| \le |per_{i+1}|$ 。

□

那么对于Case (2)，而根据定理3.3显然一共只有 log 层。注意到只有两种操作，而Case 1的操作每次只会倒退一层,Case 2的操作每次只会增加一层。所以Case (2)的操作最多只有 $m\log n$ 次。

因此我们的总失配次数是 $O(m\log n)$ 的，而对于求LCP部分，我们建出字典串的SAM on Trie后直接查询对应他们在SAM上节点的LCA就做到查询他们的LCP。注意到LCA问题可以转化成欧拉序上的RMQ问题，是可以做到 $O(1)$ 回答的。而对于重置 i, j 部分，我们已经在前置技能中已经介绍了如何在 $O(\log n)$ 的时间内求两个串的最长公共前后缀，所以我们总的时间复杂度也是 $O(m\log n)$ 的。 □

综上所述，我们取 $B = \sqrt{m}\log n$ 时复杂度最低，为 $O((q+m)\sqrt{m}\log n)$ 。

4 数据生成方式

因为这是与字符串匹配有关的相关问题，直接随机的话会导致整个串几乎没有period，会让暴力轻松通过，所以需要一种比较精细的构造办法。

4.1 字典串的构造方式

基本思路是先随机出若干个比较小的串，使得它的period尽量小。之后把这个小串复制若干份得到一个大串。然后把这个大串切片切成若干个字典串。注意切的时候要让最长的字典串不能太小，并且字典串个数也不能太少。之后我们再生成一些字典串，它们也是同一个period的若干倍，并且随机加入若干噪点。之后我们把生成的串塞入一个Trie内即可。

4.2 a数组的构造方式

构造a数组时，可以以某个概率来决定接下里的一段是带不带噪点的串。这样可以既避免了一个询问串只会跨越几个相邻的串，也避免了一个串直接匹配到底的情况。

除此之外，也要保证 $\sum_{i=1}^{m}\left|s_{a_i}\right|$ 要足够大。

4.3 询问串的构造方式

之后，对于询问串，我们也让它的period等于我们最开始随机的那若干个个小串之一。然后以一个较低概率的加入一个噪点。

对于询问串的长度。也要做到长的询问串和短的询问串都不能太少。同样的我们有三种随机方式：第一种是全都是比较长的串，第二种是全都是比较短的串，第三种是长度比较均匀，长度不同的串要尽量多。

4.4 其他构造方法

为了更好地覆盖到一些的Corner Case，还有一些手构的数据。比如全A串，随机串等等。

5 命题契机和过程

这道题是我与张宇博同学一起讨论并命题的。

张宇博同学首先想到了这道题最开始的题面。我们认为这样一道题意简洁的题一定也会有一个优美简单的做法。在最开始的思考过程中，我和张宇博同学首先想到的做法是解法三。但后来我们发现如果没有每个询问串都只会跨过两个匹配串这个性质的话，我们并不会在很好的复杂度内求出 $\tau(t, S[i:])$ 。

之后我们又回顾了我们所学过的字符串匹配算法，希望从中得到启发。最终我们发现了KMP算法中next数组与字符串period是互补关系这一良好的性质，这可以极快地加速我们匹配的过程。所以我们在KMP算法的基础上加以优化，添加了若干补丁之后得到了解法四。

在子任务的设置方面，我们也特意地给标算的若干个部分以及我们在思考这题的过程中所拐过的弯路都分别设置了部分分，来引导选手思考到标算。

在集训队互测中，因为考虑到代码复杂程度的原因，我们在把原题削弱了三个版本之后放到了集训队互测里。

对于解法三，我和张宇博同学都只会在每个询问串都只会跨过两个匹配串这个性质下解决这个题目。对于这个解法有没有更好的扩展我们都没有一个较好的结论，欢迎大家和我讨论。

6 总结

题目深度挖掘了有关字符串周期的性质，并且将这个性质应用到了有关字符串匹配中去。

对于题目的四个子任务，子任务一是朴素算法，知识、思维、代码难度都很低。子任务二需要利用SAM on trie的相关知识，但对题目稍加分析后也不难得到这个解法。子任务三则需要选手对于border性质的理解，难度稍加提升了一些。子任务四除了要掌握传统算法之外，还要求了选手能够在观察性质之后在传统算法上创新的能力，对选手的字符串水平是一个比较高的考验，无论是只写了暴力的选手，还是稍微进行过思考的选手，以及深度研究找到了所有性质的选手，都能获得适度的部分分。

而在实际考试中,有4位同学获得了20分，1位同学通过了此题。我在赛后与若干同学交流了他们有关这题的想法。发现有大多数同学想到了算法二，有少数几个同学想到了算法

三。但是因为各种原因只写了最基本的暴力。可见本题具有良好的区分度。

总体来说，这是一道既需要选手掌握了一些字符串中常见性质，又需要选手熟练掌握基本字符串算法。同时，这题还有一个比较大的代码难度，总体难度接近近年的CTSC。

就我个人而言，我非常喜欢算法四的有关想法。KMP是一个经典的算法。我们在仔细挖掘了KMP的性质之后把KMP算法应用在了这样新的场景上。考察了大家对于这个基本算法的掌握和理解。这提醒我们我们对于某个耳熟能详的算法，我们也许可以继续挖掘它的性质，从而出出来一些有趣的题。

7 感谢

感谢张宇博同学对撰写本文提供的巨大帮助，也同时感谢他在OI路上对我一直以来的帮助与陪伴。

感谢中国计算机学会提供学习和交流的平台。

感谢国家集训队教练张瑞喆教练的指导。

感谢李建老师以及父母多年来的关心和指导。

感谢王修涵、吴越、董炜隽、李宁远同学为本文验稿。

参考文献

[1] 刘研绎, 《后缀自动机在字典树上的扩展》, 2015年国家集训队论文。

[2] wikipedia，Knuth－Morris－Pratt algorithm，https://en.wikipedia.org/wiki/KMP_algorithm

[3] 金策，《字符串算法选讲》，2017年NOI冬令营。

[4] 王鉴浩，《浅谈字符串匹配的几种方法》，2015年国家集训队论文。

[5] 张天扬，《后缀自动机及其应用》，2015年国家集训队论文。

解决树上连通块问题的一些技巧和工具

绍兴市第一中学 任轩笛

摘 要

树上连通块问题是信息学竞赛中的一类热门问题，本文介绍了解决这类问题的一些常见技巧和工具。对于树上连通块DP问题，本文介绍了按DFS序转移及点分治、“点数-边数”、用线段树合并进行整体DP、用链分治维护动态DP等常见技巧及其应用；对于树上同色连通块维护问题，本文探究了其一般思路，并介绍了一种基于LCT的、能支持链修改颜色并维护任意DFS序上信息的通用数据结构。

1 引言

树是一种非常特殊的结构，树的任意一个连通子图（树上连通块）同样也是一棵树，若干个树上连通块的交也是一棵树。笔者在学习过程中遇到了形形色色的和树上连通块相关的问题，大致将其分为两大类：对树上连通块计数/求最优值的DP问题、树上同色连通块信息维护问题。

近年来出现的不少和树上连通块相关的问题往往方法繁多，且因题而异，常常使人觉得无从下手。本文对解决树上连通块问题的一些常见方法和思路进行了整理和归纳，并针对树上同色连通块信息维护问题，介绍了一种支持链修改颜色并维护任意DFS序上信息的通用数据结构，旨在让读者遇到这类问题时能够有迹可循。

本文主要分为两个部分，第一部分从几个例题入手，分别介绍了按DFS序转移及点分治、“点数-边数”、用线段树合并进行整体DP、用链分治维护动态DP等方法在树上连通块DP问题中的应用。第二部分围绕树上同色连通块问题展开，通过两个经典例题展示了该类问题的一般思路，并介绍了Link Cut Memphis这一支持链修改颜色并维护DFS序上信息的数据结构。

2 相关定义和说明

无向图 $G = \{V, E\}$ 被称为树当且仅当任意两个点之间有且仅有一条简单路径。任意无环的连通无向图都是树。

树 $T = \{V, E\}$ 的一个树上连通块 $T' = \{V', E'\}$ ，满足 $V' \subseteq V, E' \subseteq E$ 且 T' 连通。易知 T' 的结构也为一棵树。

本文涉及的树上连通块DP问题形式一般为以下两种：

- 计数类问题：给出一棵树 T ，求有多少树上连通块满足性质 A 。
- 最优化问题：给出一棵树 T 和一个价值函数，求 T 的所有满足某性质的树上连通块中的最优价值。

本文涉及的树上同色连通块信息维护问题形式一般为：给出一棵树 T ，每个点有黑白两种颜色。定义一个树上连通块合法当且仅当其所有节点颜色相同。询问的是点 x 所在极大合法连通块的信息。

这两种问题以外可能还有若干较少见的和树上连通块有关的问题，不在本文讨论之列。

3 树上连通块的DP问题

3.1 朴素算法

对于一般的树上连通块DP问题，通用的朴素做法是设 dp_x 表示在 x 子树中选择一个包含点 x 的连通块时的方案数/最优答案。 dp_x 可以由 x 的儿子的 dp 值合并得到，即遍历 x 的所有儿子，决策每个子树内是留空还是选择一个含根的连通块。

如果合并两个儿子的信息是 $O(1)$ 的（例如求有多少本质不同的树上连通块），则总的复杂度为 $O(n)$ ，如果合并两个儿子的复杂度为 $O(size_a * size_b)$ （例如求多少本质不同的 k 个点的树上连通块），则任意两个点都会在它们的LCA处产生1的复杂度，总复杂度为 $O(n^2)$ 。

这种朴素算法对计数类问题和最优化问题均适用。

3.2 按DFS序转移与点分治

对于一类树上连通块DP问题，如果信息的合并不够高效，但加入单点的信息比较高效，往往可以按照DFS序转移，把合并子树变成添加单点。常见的有背包的模型，设体积上限为 m ，合并两个背包的复杂度为 $O(m^2)$ ，而加入一个物品的复杂度为 $O(m)$ 。

先假定选出的连通块必须包含根，求出整棵树的DFS序，一个包含根的树上连通块的结构一定形如整棵树去掉若干个互不相交的子树。在DFS序中，去掉的就是连续的若干段。

设 dp_i 表示考虑了DFS序前 i 个节点时的信息。如果要选择第 i 个节点，则转移到 dp_{i+1} ，否则转移到 dp_j ，其中 j 是第 i 个点子树DFS序的右端点+1，表示去掉这整个子树。

这样每次就只要往背包中添加一个物品，而不是合并两个背包，设添加单点的复杂度为 $O(m)$ ，总的复杂度就是 $O(nm)$ 。

而对于选出的连通块可以不包含根的问题，只要进行点分治，每次把重心作为根，算出强制包含重心的方案数，再把重心删除，对每个连通块递归做。总的复杂度为 $O(n\log n*m)$ 。

例题一 [11]

题意

给出一棵树，每个点有个颜色。给出3种颜色 u,v,w ，求有多少个树上连通块满足里面颜色为 u,v,w 的点的个数分别为 a,b,c 。

做法

朴素做法：自底向上DP，用 $dp_{x,i,j,k}$ 表示处理了以 x 为根的子树，三种颜色个数分别为 i,j,k 的方案数。转移合并两个子树时枚举两个子树中三种颜色各自的点数，复杂度是$O(na^2b^2c^2)$ 。

注意到这里合并两棵子树是一个三维卷积的过程，如果使用三维FFT优化，复杂度能变成 $O(nabc\log abc)$ ，但是常数和码量都较大。

考虑使用上述的点分治算法，配合按DFS序转移，复杂度为 $O(nabc\log n)$ ，常数很小，而且可以方便地推广到最优化问题。

3.3 “点数-边数”

对于一棵非空的树，点数-边数恒为1，我们可以用这个等式来统计满足某性质的树上连通块个数，即枚举每个点/每条边，算出包含这个点/这条边的合法树上连通块个数，用点的答案减去边的答案，就能让所有合法的非空树上连通块被统计到恰好1次。

这个技巧常用于求若干个树上连通块的交的场合，因为若干个树上连通块的交仍然是一棵树（可能为空）。若干个树上连通块交集非空的方案数并不容易直接计算，但如果只考虑一个点/一条边，就变成每个树上连通块都要包含这个点/这条边。树上连通块之间彼此独立，变得相对容易计算。

例题二：完美的集合 [12]

题意

[11] Source:【清华集训2016】连通子树 子问题

[12] Source:集训队互测2018 by 梁晏成

给出一棵树，每个点有重量和价值，每条边有边权，考虑选出一个点的子集 S ，满足这些节点重量之和 $\leq M$ 且构成一个树上连通块，把那些价值和最大的集合 S 称为完美的集合。

如果两个点 x, y 满足 $dist(x, y) * v_y \leq Max$ ，则 x 可以对 y 进行测试，问有多少种方案在所有完美的集合中选出 k 个，使得在它们的交中存在一个点 x ，能对这些集合中所有点进行测试。

答案对 5^{23} 取模。

$n \leq 60, m \leq 10000, k \leq 10^9$ 。

做法

能对一个集合中所有点进行测试的点形成了一个树上连通块。若干个树上连通块的交仍是树上连通块。考虑用"点数- 边数"的技巧，即统计能被某个点 x 测试的完美集合个数，减去能同时被某条边两个端点测试的完美集合个数。

考虑计算能被点 x 测试的完美集合个数，这些集合肯定不能包含任何到 x 距离不合法的点。在去掉非法的点后的树中进行DP，相当于要问有多少树上连通块重量之和 $\leq M$ 且价值和最大。这是一个经典的最优化背包问题，只要在背包记录最优值的同时再记录达到最优值的方案数就可以了。把点 x 当作根，按照DFS序转移，复杂度为 $O(nm)$ 。

最后还涉及到一个组合数取模，和本文关系不大，不再展开。

总的复杂度为 $O(n^2m +$ 组合数取模复杂度$)$ 。

3.4 用线段树合并进行整体DP

有这样一类树上连通块计数问题：每个点需要维护一个大小为 m 的DP数组，合并两个子树时，操作是把 m 个位置对应进行合并，添加一个点时需要对某个位置进行修改。虽然DP状态有$O(nm)$ 个，但大部分状态都仅仅是重复地由儿子状态合并得到。而注意到"添加一个点"的操作是 $O(n)$ 级别的，如果我们能快速地完成按位合并操作，就能很好地解决这类问题。

考虑使用线段树合并，即把每个点的 m 个DP值用一棵线段树维护，添加一个点时在其线段树中进行修改，合并两个子树时用线段树合并来批量完成转移。线段树中插入的总点数是 $O(n \log n)$ 级别的，而线段树合并的复杂度不高于其插入总复杂度。

例题三 [13]

题意

给出一棵树，每个点有颜色，求有多少树上连通块包含不超过2种颜色。

做法

[13] Source:THUWC 2018

设 $f(x,c)$ 表示在 x 的子树内选择一个包含 x 的连通块，且2种颜色分别为 col_x 和 c 时的方案数。

讨论 x 的儿子 y 的颜色，不难得出转移：若 y 和 x 的颜色相同，则直接将所有 $f(y,\sim)+1$ 乘给 $f(x,\sim)$ ，否则将 $f(y,col_x)+1$ 乘给 $f(x,col_y)$ 。

第二种转移总共只需要转移 $O(n)$ 个状态，瓶颈在于第一种转移：每有一个同色儿子，就要花费 $O(\text{子树中颜色数})$ 的复杂度。

观察到不同的颜色是彼此独立的，可以用线段树来维护一个点上的所有DP值，合并子树的时候进行线段树合并就可以了。

复杂度为 $O(n\log n)$ 。

3.5 用链分治维护动态DP

动态DP是指支持对DP的输入进行修改，并在每次修改后快速求出DP值，一般应用于计数类问题。在2017年的集训队论文中，陈俊锟提出了一种具有扩展性的解决树上动态DP问题的算法——链分治，这里考虑将链分治算法应用到树上连通块动态DP问题中。

具体地，我们需要对节点的信息进行修改，并在每次修改后求树上连通块计数类问题的答案。

链分治的思想是将树进行轻重链剖分，按照重链从底向上的顺序依次计算DP值。对于一条重链，其延伸出去的子重链的DP信息已经计算完毕，可以把它们的信息附着在这条重链上，然后就变成了一个序列上的动态DP问题。而序列上往往可以使用数据结构来支持修改一个点的信息、快速维护DP值。

修改一个点的信息时，根据这个分治结构，只会有 $O(\log n)$ 条重链的信息需要修改。从底向上依次在每条重链的数据结构中修改并维护DP值就可以了。

一般地，设 $G(x)$ 表示在 x 的子树中选择一个含根的连通块时的答案，对于一条重链，设上面的节点按深度从小到大排序为 $x_1,x_2,...x_k$ ，对于每个节点 x_i ，都已经求出了它所有轻儿子的 $G()$ 值，把这些信息与 x_i 本身的信息进行合并，就能得到这条重链上选择 x_i 这个点时的贡献。设这个值为 $F(x_i)$ 。

如果选出的树上连通块和这条重链有交，交集必然是一个区间，贡献就是这个区间的 $F(x)$ 之积。亦即要求这条重链上所有区间的 $F(x)$ 积的和。只要使用线段树/平衡树维护区间内信息的积、所有前/后缀信息积的和，就能方便地在 $O(\log n)$ 时间内支持修改一个点的信息、重新计算DP值。

而这条重链对于其父重链的贡献，就是在这条重链上选择一个前缀的方案数，只要用线段树/平衡树维护出的前缀信息去更新其父节点的 $G()$ 值就可以了。

至此这个算法的框架已渐渐清晰：对树进行轻重链剖分，按照重链自底向上的顺序计算DP值。修改一个点的信息时，对于其祖先中每条重链，在线段树/平衡树中重新算出选择一个前缀的方案数，并用这个方案数更新这条重链的父亲的信息。

如果需要维护的信息可减，在修改子重链的信息后，直接减去旧的信息，加上新的信息就可以了。否则还需要对于每个点开一个大小为 O(轻儿子个数) 的线段树或平衡树来快速维护轻儿子们的信息。但是复杂度是不变的。

例题四：切树游戏 [14]

题意

给出一棵树，每个点有点权。有2种操作，形如

1、Change x y ，将编号为 x 的节点的权值改成 y 。

2、Query k ，询问有多少非空连通子树，满足其点权的异或值恰好为 k 。

输出对10007取模。

$n, Q \le 30000, 0 \le A_i, y, k < 128$ 。

做法

首先把异或值全都转成FWT的点值，运算过程中全部用点值计算，最后再FWT回去。设 m 表示权值的最大值，合并信息的复杂度就是 $O(m)$ 。

考虑应用链分治算法，设 $G(x, \sim)$ 表示在 x 子树中选择一个含根的连通块时的点值。对于重链上一个点，算出其轻儿子的 $G(y, \sim) + 1$ 之积，然后用数据结构维护选择一个区间的方案数，再用选择一个前缀的方案数来更新重链父亲的 $G(fa, \sim)$ 。

由于维护的是积，在去掉一个轻儿子的旧的贡献时，可能会面临除以0的问题，一个解决方法是维护模10007非0的值的积和10007的幂次。

用树链剖分维护的复杂度是 $O(qm\log^2 n)$ 。

如果用LCT维护链分治结构，在Access的时候动态地更新轻重链的关系并维护答案，复杂度可以做到 $O(qm\log n)$ ，而且可以更方便地询问一些子结构的DP信息，如询问某个点子树中的答案，或和某条路径 $u \sim v$ 有交的答案等，还能支持Link/Cut等操作。

3.6 总结

树上连通块DP问题一般分为计数类问题和最优化问题。

如果合并两个DP状态的复杂度较高，而添加一个点的复杂度较低，例如大部分的背包问题，往往可以用按DFS序转移代替朴素的自底向上转移。如果所求的树上连通块不一定包含根节点，可以多花费$O(\log n)$ 的复杂度进行点分治，每次把重心定为根，求出强制包含根的连通块信息，再递归每个连通块。

“点数-边数”的技巧常用于选多个树上连通块交非空的计数类问题中。通过做 $O(n)$ 次强制包含某个点/某条边的DP，能让不同的树上连通块彼此独立。特别地，如果题目要求的是树上连通块两两有交，也等价于其全部交起来非空。

[14]Source:SDOI2017 Round2 Day1

对于大部分时间花在合并两个子树对应信息的题目，可以考虑用线段树存储DP信息，用线段树合并来快速完成批量转移。

对于要支持修改节点信息并重新对树上连通块计数的问题，可以用树链剖分或LCT来维护动态DP，本质上利用的是其信息的合并满足结合律的条件。对每条重链求其选一个区间的答案和，再用其选一个前缀的答案和更新其父重链的答案。

4 树上同色连通块维护问题

4.1 一般思路

考虑一棵黑白两色的树，对于每个极大的同色连通块，其最浅点，即整个连通块的 LCA 是唯一的，且可以通过树链剖分或者LCT快速地找到，即求某个点到其最近异色祖先路径上的倒数第二个点。我们不妨把信息维护在这个最浅点处。

最浅点的颜色修改时，连通块的形状会有较大变化——原本与其同色的儿子要变成异色，而原本异色的儿子变成同色。如果直接维护“点 x 子树内与 x 同色的连通块信息”，要花费O(儿子个数) 的时间。

解决的办法是在每个节点上同时记录两种颜色的信息，即如果一个点是黑色/白色时，以它为最浅点的该种颜色连通块的信息。这样修改它的颜色时，它本身的信息不需要修改，只要在父节点处添加/删除这个点的信息就可以了。

如果颜色数不是两种，而是一个较小的常数 k ，可以维护 k 个结构，第 i 个结构中把第 i 种颜色当作黑色，其余颜色当作白色进行处理。这样可以以复杂度乘 k 的代价支持多种颜色的维护。

例题五 [15]

题意

给出一棵树，每个点有黑白两种颜色，要求支持两种操作：

1、询问点 u 所在同色连通块的大小。

2、翻转点 u 的颜色，即黑色变白色，白色变黑色。

$n, q \leq 10^5$ 。

做法

考虑在每个点上维护两个值 $B(x)$ 和 $W(x)$ ，分别表示如果点 x 是黑色/白色时，以它为最浅点的该种颜色同色连通块的大小。

对于询问操作，设询问的是点 u ，且它是白色，只要找到 u 所在的白色连通块的最浅点 v ，输出 $W(v)$ 就行了。

[15] Source:SPOJ QTREE6

考虑翻转点 u 的颜色，假设是从白色改成黑色，只要找到 u 往上第一个黑色节点 v ，把 $(u, v]$ 的 $W()$ 值都减去 $W(u)$ ，再找 u 往上第一个白色节点 v ，把 $(u, v]$ 的 $B()$ 值加上 $B(u)$ ，就维护完成了。

“ u 往上第一个白色/黑色节点”可以用数据结构维护链上区间内两种颜色的点数来进行查找。如果用树链剖分+线段树，复杂度是 $O(n \log^2 n)$ ，如果用LCT，复杂度为 $O(n \log n)$ 。

例题六 [16]

题意

给出一棵树，每个点有黑白两种颜色，还有个点权，要求支持三种操作：

1、询问点 u 所在同色连通块中最大点权。

2、翻转点 u 的颜色。

3、修改点 u 的点权。

$n, q \leq 10^5$ 。

做法

与上一题不同的是，这题要维护最大值，并不能像上一题一样直接把贡献减掉。

观察上一题中维护的信息的本质，如果一个点是白点，其信息就会被加到父节点的 $W()$ 中，若是黑点则被加到父节点的 $B()$ 中。

相当于有一棵黑树一棵白树，黑点在黑树中与其父亲相连，白点在白树中与其父亲相连。每棵树中每个连通块除了最浅点，都为该种颜色。询问一个点所在同色连通块信息时，同样找到其最浅同色祖先，询问其整个子树的信息就可以了。

考虑使用LCT，在每个点上维护一个multiset，记录其所有轻儿子子树中的最大点权。在splay上同时维护重链的最大点权。

在Access切换轻重边时，在multiset里更新信息：插入原先的重儿子的值，并删去新的重儿子的值。修改点权只要先Access就能方便地维护了。

修改一个点的颜色，比如把 u 从白色改为黑色，若 u 非根节点，就在白树中断开 u 和父亲的边，在黑树中连上 u 和父亲的边就可以了。

每次切换轻重边时有一个multiset的 O(轻儿子个数) 的复杂度，根据Top-Tree的复杂度证明，这个做法的复杂度是 $O(n \log n)$ 的。

4.2 维护DFS序上任意信息

对于更复杂的信息，只要其满足结合律且能快速合并，换言之能用数据结构在DFS序上维护，都可以套用上面的做法。

[16]Source:SPOJ QTREE7

观察上面提到的黑树/白树的形态，可以发现其和原树是完全一致的，而且整个过程中都不需要用到换根操作。也就是说可以直接按照原树的DFS序进行维护。

考虑用一种支持提取区间的平衡树，如Splay或Treap来处理，在LCT里Link/Cut时，到平衡树中提出子树拼入父节点DFS序中，或将子树与父节点分离开来。

一次平衡树操作复杂度为 $O(\log n)$ ，于是总的复杂度为 $O(n\log n)$ 。

4.3 链修改颜色

对于链修改颜色，这里介绍一种由陶渊政提出的基于LCT的数据结构[17] 。

把一条链上所有点的颜色都改成黑色/白色，如果将链颜色覆盖看成LCT操作，可以将问题转化为 $O(n\log n)$ 次同色链反色。

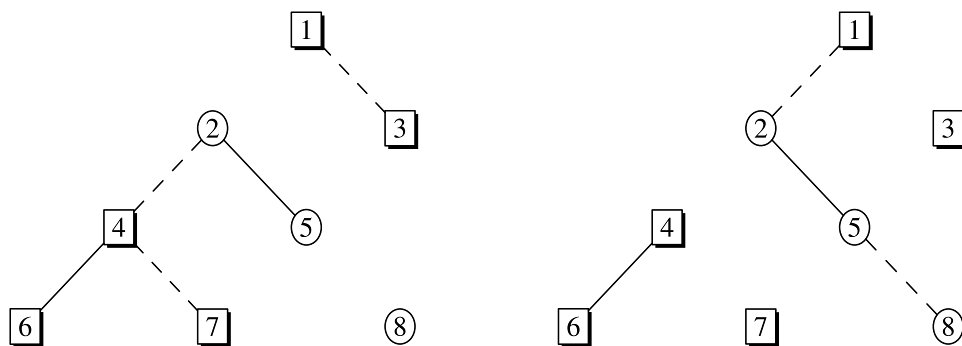
4.3.1 新的黑/白树定义

效仿前面提到的做法，仍然是维护黑树和白树，不过要稍微改一下两者的定义，以一个白点 u 为例，

1、 u 和它的父亲 v 在白树上相连。

2、若 u 和 v 在白树上是以重边相连的，则将它们在黑树上连接。

一种可能的黑树/白树形态：



在新的定义里，黑树中任意一个连通块可视为两部分组成：

1、由黑色连通块组成的若干个连通子树，把这部分称为块树。

2、一条白树中的重链对应过来的白点构成的链，把这部分称为链树。

可以发现，白色链下可能挂着若干个黑色块树，但黑色块树下一定没有白色节点。因为在黑树中白点是不可能跟其父亲连接的。

[17]一般将这种数据结构称作Link Cut Memphis(LCM).

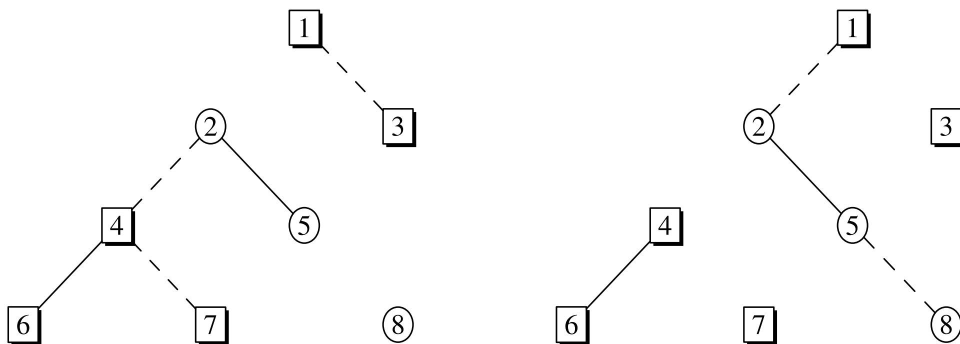
在新的定义中，同色连通块在对应颜色的树中仍然是一段连续的DFS序，可以用平衡树来维护。询问时仍可以找到最浅同色祖先然后询问子树信息。剩下的问题就是在链修改的时候维护黑树和白树的形态。

4.3.2 Expose操作

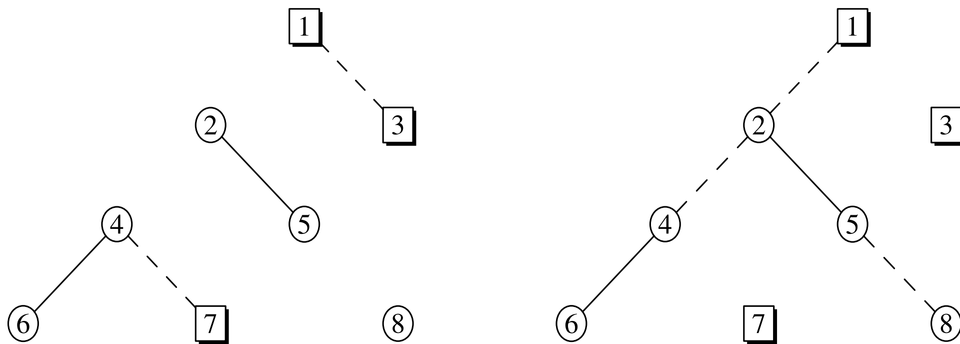
定义Expose操作，为Access操作的变种。设 v 为 u 的最浅同色祖先，Expose(u)的效果为：将 u 和 v 之间的路径改为以重边相连，同时断开 u 下方的重边。在更改轻重边时，到异色的树中相应地进行Link/Cut操作。

考虑将 u 到最浅同色祖先 v 的链进行反色（假设 u 为黑色），首先Expose(u)，此时黑树中 u 到链顶的链已经打通，直接把这部分链当成白色链树即可。而白树中 $u \sim v$ 链以上就是白色块树，直接把这部分链当成块树拼到父节点块树下就可以了。同时要断开黑树中 v 和父亲的边，连上白树中 v 和父亲的一条轻边。

考虑以下例子：



我们要把 $4 \sim 6$ 的黑链改成白色。上图为黑树/白树经过Expose(6)后的状态。



可以发现只要断开黑树中 $(2,4)$ 这条虚边，连上白树中 $(2,4)$ 这条虚边，然后直接把这部分改成白色就可以了。

4.3.3 流程整理

假设要将 $u \sim v$ 的链修改为颜色 c ，先通过讨论LCA处的情况，转化成若干次到根路径改色。

从 u 开始，不停地找到根路径上的待反色同色段，设当前段最浅点为 v 。

用Expose(u)操作打通 $u \sim v$ 的链并在另一棵树中对应维护好形态。

连上/断开 v 和其父亲的边，并令 $u = v$ 的父亲。

重复这个过程直至 u 爬到根。

4.3.4 复杂度分析

链上同色段数，即Expose切换轻重边的次数同LCT的Access操作一样，均摊意义下为 $O(n \log n)$ 。每次切换轻重边时都要到另一棵树中进行一次Link/Cut操作，而Link/Cut 的时候要到平衡树中 $O(\log n)$ 维护好DFS序，因此总的复杂度是 $O(n \log^2 n)$ 。

4.4 子树修改颜色

对于有子树修改颜色操作的问题，由于在菊花树的情况下，每次修改会影响 $O(n)$ 条重链，如果要维护一般的DFS序信息，我并没有找到较为高效的算法。

如果信息较为简单，使得子树修改后信息能够直接计算（比如同色连通块内点数等），可以使用Top-Tree这一数据结构进行维护从而做到 $O(n \log n)$ 的复杂度。

例题七 [18]

题意

给出一棵 n 个点的无根树，每个点有两种颜色，最初所有点均为黑色。每条边有正的权值。

需要维护 m 次操作：

1 u ：询问 u 所在树上同色连通块的直径。

2 $u\ v\ c$ ：将 $u \sim v$ 的链覆盖为颜色 c 。

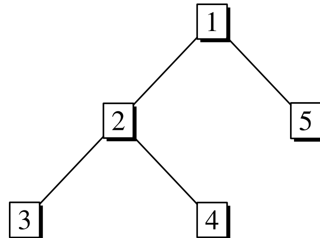
$n, m \leq 10^5$ 。

做法

找树上直径有一种单次 $O(n)$ 的做法：随便找一个起点，bfs一遍找到一个最远点 u ，再从 u 开始bfs一遍找到一个最远点 v ，则 $u \sim v$ 就是直径之一。遗憾的是这一种做法难以高效地维护。

[18] Source:BZOJ 3914

有另一种可以在DFS序上维护的做法——欧拉遍历序，即把DFS过程中正向和反向经过一条边全都记录下来，分别写为左括号和右括号，则两点之间的括号序列化简后的长度就是两点之间的距离。



如上图中的树DFS序为 [1 [2 [3] [4]] [5]] ， $dist(2,4)$ 就等于 $(2,4)$ 间的括号序列化简后的长度，即1。 $dist(3,4) = 2$ 。

化简后的括号对用二元组 (A, B) 表示，即

$$\overbrace{]\,]\,]\,]\,]\,\ldots\,]\,]\,]}^{A}\ \overbrace{[\,[\,[\,[\,[\,[\ldots\,[\,[\,[}^{B}$$

树的直径就是选一个子串，使得化简后 $A + B$ 最大。

两个串合并时在中间会产生 $\min(B_1, A_2)$ 对匹配括号。也就是

$$(A_1, B_1) + (A_2, B_2) = (A_1 + \max(A_2 - B_1, 0), B_2 + \max(B_1 - A_2, 0))$$

$$A' + B' = \max(A_1 + B_1 + B_2 - A_2, A_1 - B_1 + A_2 + B_2)$$

用splay维护DFS序区间最大 $A+B$ ，后缀最大 $A+B$ ，后缀最大 $A-B$ ，前缀最大 $B-A$ ，前缀最大 $A+B$ ，以及整个区间的 A, B ，这些信息就都可以互相计算了。

对于链修改，套用LCM就可以了。复杂度为 $O(n \log^2 n)$ 。

5 后记及展望

本文提到了不少解决树上连通块问题的方法和技巧，但在浩瀚的算法海洋中仍只是冰山一角。希望未来能有更多和树上连通块问题相关的算法被发现和发明，也希望有更多有趣的树上连通块问题出现在信息学竞赛中。

6 致谢

感谢中国计算机学会提供学习和交流的平台。

感谢绍兴一中的陈合力老师和董烨华老师的关心和指导。

感谢国家集训队教练张瑞喆的指导和帮助。

感谢清华大学的陈俊锟学长对于树上动态规划问题提出的许多实用算法和技巧。

感谢北京大学的陶渊政学长、清华大学的任之洲学长对LCM这个数据结构的研究作出的贡献。

感谢北京大学的洪华敦、孙耀峰、冯哲、孙奕灿、叶珈宁等学长和绍兴一中的同学们对我的帮助。

感谢北京大学的孙奕灿学长为本文验稿。

感谢其他对我有过帮助和启发的老师和同学。

感谢父母对我的关心、支持与无微不至的照顾。

7 参考文献

[1] 陈俊锟, 《平凡的圆方树和神奇的(动态)动态规划》, WC2018.

[2] 陈俊锟, 《< 神奇的子图 > 命题报告及其拓展》, 2017年集训队论文.

[3] 陶渊政, “动态树拓展相关”, http://memphis.is-programmer.com/2015/8/7/linkcutmemphis.99293.html.

[4] 任之洲, 《一种基于LCT的树上同色连通块维护算法》, WC2016营员交流.

[5] Wikipedia, Top tree.

《Jellyfish》命题报告及拓展探究

宁波市镇海中学 梁晏成

摘 要

《Jellyfish》是作者在集训队第一轮作业中命制的一道题目“海蜇？海蜇！”。

本文首先介绍试题有关信息和该题的做法，然后在该题的基础上，从分治乘法的角度讨论一些多项式和集合幂级数卷积的问题。

1 引言

近年来，多项式一类的题目层见叠出。这类题目灵活多变，既有思维难度，又考察数学功底，部分题目还能检验选手的代码能力，因此这些题目越来越受到出题人的青睐。

在多项式问题中，“卷积”占有重要的地位。一般在解决卷积问题时，我们常常用变换的思想来加快速度。大家较为熟悉的“快速傅里叶变换(FFT)”，“快速沃尔什变换(FWT)”，“快速莫比乌斯变换(FMT)”都是基于变换的思想来解决卷积问题的典型。

分治乘法是一种从不同的角度解决卷积问题的方法。Karatsuba算法[19] 是最经典的分治乘法，它能在大约$O(N^{1.59})$ 的时间内解决两个次数为 N 的多项式相乘的问题。由于时间复杂度不及FFT算法，因此并不太常见，但是它给了我们一个新的看问题的角度。实际上，用分治乘法的思想去解决一些位运算规则下的卷积，往往能取得不错的效果。

本文第二、三两节介绍一道运用分治乘法思想解决卷积问题的题目《Jellyfish》及其解法，并介绍了Karatsuba算法及其改进。第四节给出了一些分治乘法应用的例子。

为了方便起见，我们做一些约定：

记号 1.1. 对于一个布尔表达式x，$[x] = \begin{cases} 1 \mid x = true \\ 0 \mid x = false \end{cases}$

记号 1.2. 对于一个实数 x， $\lfloor x \rfloor$ 表示不超过 x 的最大整数

[19]https: // en.wikipedia.org / wiki / Karatsuba _ algorithm

2 试题信息

2.1 简要题意

- 给定 20 维空间中 N 个关键点，对每个点将坐标看成 d 进制数后转化为十进制数用 a_i 表示，并将 a_i 称为这个点的位置编码。关键点的每一维坐标值范围是$\{0, .., d-1\}$，
- 现在随机从某个关键点出发，随机走 K 步，每次走到一个和当前点相邻的点上（两个点相邻当且仅当它们恰好存在一维坐标值相差1）。求期望经过的不同的关键点的个数

2.2 数据规模与时空限制

$1 \leq N \leq 200,000, 0 \leq K \leq 500, d \in \{2, 4\}, 0 \leq a_i \leq 250,000$
存在 25% 的数据，满足 $1 \leq N \leq 3000$
另存在 15% 的数据，满足 $d = 2$
另存在 20% 的数据，满足 $1 \leq K \leq 4$，其中 10% 满足 $d = 2$

时间限制：TUOJ上的 $4s$
空间限制：512M

3 试题解法

3.1 简要分析

我们令 $e_{x,y}$ 表示从点 x 出发并在 K 步中经过点 y 的概率（ x, y 可能相同）。那么最后要求的答案就是 $\sum_{i=1}^{N}\sum_{j=1}^{N} e_{i,j}$

根据期望的线性性，我们可以先独立地求出所有的 $e_{x,y}$，然后求和得到答案。

我们令 $ways_{x,y}$ 表示从点 x 出发并在 K 步中经过点 y 的方案数，那么 $e_{x,y} = \frac{ways_{x,y}}{N \times 40^N}$。因此我们只需要对每对 x, y 求出 $ways_{x,y}$，求和后除以 $N \times 40^N$ 即可

3.2 暴力解法

3.2.1 简单暴力

我们考虑首先枚举一个点 x，然后直接枚举接下来的 K 步每一步是移动哪一维，这一维是+1还是-1。在第一次经过某个点时更新答案。

时间复杂度为 $O(N40^K)$

3.2.2 改进1

注意到上面算法的复杂度为 $O(N40^K)$，如果我们能让 K 减小一点，时间复杂度能大大提高。

考虑运用折半搜索的思想。假设在第 s 步从 x 走到了 y，我们可以把这个过程替换为先从 x 出发走 $\lceil \frac{s}{2} \rceil$ 步到达一个点 p，然后再从y 出发倒着走 $\lfloor \frac{s}{2} \rfloor$ 步到达同一个点 p。

这样我们先枚举总步数 s，并将从所有 y 出发倒着走 $\lfloor \frac{s}{2} \rfloor$ 步到达的点全部记录下来，然后每次枚举起点 x 和接下来的 $\lceil \frac{s}{2} \rceil$ 步的情况并统计答案。

注意这么统计可能会重复，但是由于这个做法针对 K 很小的情况，我们简单讨论一下就能把重复的部分扣除。

时间复杂度降为 $O(NK40^{\frac{K}{2}})$，可以通过 20% 的测试数据。

3.2.3 改进2

注意到，对于点对 (x, y)，我们并不关心点 x, y 的具体位置，而只关心他们的相对位置。

因此我们先 $O(N^2)$ 枚举点对 (x, y)，然后把 x 看成原点时 y 对应的位置加入可重集合 S。之后我们可以看成是从原点出发，对可重集合 S 中的每个点求走 K 步经过这个点的方案数。

如果用哈希算法来维护 S 和查询 S 中的某个点的出现次数，时间复杂度可以做到 $O(N^2 + 40^K)$

3.3 预处理部分

暴力算法的改进2给了我们一个方向：如果我们能快速预处理出所有可能的相对位置的答案，就能 $O(N^2)$ 求出最终的答案。

换而言之，我们需要解决这么一个问题：

给你一个关键点，问从原点 $(0, 0, ..., 0)$ 出发走 K 步能经过这个关键点的方案数是多少

我们考虑运用动态规划，对所有可能的情况预处理出答案。

方便起见，我们假设 $d = 4$（$d = 2$ 的情况实际上可以被 $d = 4$ 包含）。

引理 3.1. 交换某个关键点的某两维后，从原点出发走 K 步经过这个关键点的方案数不变。

Proof. 考虑一个关键点 $P = (a_1, ...u, ..., v, ..., a_n)$，以及交换了 u, v 的对应点 $P' = (a_1, ...v, ..., u, ..., a_n)$

构造一个映射 $A(x)$ ，满足：

$$A(x) = \begin{cases} v \mid x = u \\ u \mid x = v \\ x \mid x \neq u, x \neq v \end{cases}$$

我们用 (t_i, d_i) 表示第 i 步走的是第 t_i 维，这一维的变化量是 d_i

那么如果我们通过 $(t_1, d_1), ..., (t_K, d_K)$ 经过了点 P ，那么可以通过 $(A(t_1), d_1), ..., (A(t_K), d_K)$ 经过点 P'

反之亦然

这样我们就证明了经过点 P 的方案和经过点 P' 的方案一一对应，并且方案数相同

引理 3.2. 将某个关键点的某一维取反后，从原点出发走 K 步经过这个关键点的方案数不变。

Proof. 假设将第 k 维取反，只需定义函数 $B(x)$:

$$B(x) = \begin{cases} -1 \mid x = k \\ 1 \mid x \neq k \end{cases}$$

然后 $(t_i, d_i) \Rightarrow (t_i, B(t_i)d_i)$ 即可（其余部分我们沿用上一个引理证明的方法）

构造一个辅助数组 b ，我们用 b_i 表示这个关键点有几个维度的坐标值的绝对值为 i ，那么由引理3.1和引理3.2，如果两个关键点拥有相同的 b 数组，它们的答案是相同的。

同时，因为 $b_0 + b_1 + b_2 + b_3 = 20$ ，所以我们可以用 dp_{b_1,b_2,b_3} 来表示如果一个关键点的 b 数组为 $\{20 - b_1 - b_2 - b_3, b_1, b_2, b_3\}$ ，从原点走 K 步能经过这个点的方案数。

我们考虑如何求出 dp 数组，在此之前，我们需要一些准备工作：

- 首先，我们需要一个数组 $f_{i,j,k}$ ，表示在 j 维空间中从原点走 k 步走到 $(i, i, ..., i)$ 的方案数
- 那么，我们可以枚举在第 j 个维度走了 x 次 -1 和 $x + i$ 次 $+1$ ，这样可以得到转移方程：

$$f_{i,j,k} = \sum_x \binom{k}{x + x + i} \binom{x + x + i}{x} f_{i,j-1,k-x-(x+i)}$$

我们先预处理出所有的 f ，时间复杂度为 $O(pdK^2)$ ， p 表示空间的维度，本题中为20。

现在我们考虑对于某个 b 数组，需要求出对应的 dp 数组:

- 我们先求一个数组 h， $h_{i,j}$ 表示在 $b_0 + ... + b_i$ 维下，
  从原点出发走 j 步到达点 $(0, ..., 0(b_0 \text{ 个 } 0), 1, ..., 1(b_1 \text{ 个 } 1), ..., i, ..., i(b_i \text{ 个 } i))$ 的方案数
- 我们枚举 k 表示 b_i 对应的那些维度走了几步，得到：

$$h_{i,j} = \sum_k \binom{j}{k} h_{i-1,j-k} f_{i,b_i,k}$$

- 现在我们令 g_i 表示走了 i 步恰好到达该关键点（即在第 i 步第一次到达该关键点）的方案数。考虑使用容斥，我们用总方案数 $h_{d,i}$ 减去不合法的部分来得到答案。
- 对于不合法的部分，一定是之前已经经过了该关键点，考虑枚举之前第一次到达该关键点时是第 j 步，得到：

$$g_i = h_{d,i} - \sum_{j<i} g_j f_{0,20,i-j}$$

- 那么

$$dp_{b_1,b_2,b_3} = \sum_i g_i \times 40^{K-i}$$

整个预处理的时间复杂度为 $O(p^3 dK^2)$，但是常数非常小，可以在较短的时间内求出。

如果我们在预处理之后采用 $O(N^2)$ 枚举每个点对并通过 dp 数组计算答案，可以通过 25% 的测试数据。

3.4 进一步分析

有了 dp 数组，我们现在就只需要求出一个存储 N 个点两两之间距离差（距离差定义见下）的数组，求出所有的方案之和，最后除以 $N \times 40^N$ 即可。

定义 3.1. 定义一个运算符 $\oplus$，在 d 进制下表示两个位置编码的距离差函数，具体地：

$$(\overline{a_1 b_1 c_1 ...})_d \oplus (\overline{a_2 b_2 c_2 ...})_d = (\overline{|a_1 - a_2|\, |b_1 - b_2|\, |c_1 - c_2| ...})_d$$

形式化地：

$$x \oplus y = \sum_{k=0} \left| \left\lfloor \frac{x}{d^k} \right\rfloor - \left\lfloor \frac{y}{d^k} \right\rfloor \right| \mod d \times d^k$$

定义 3.2. 对于两个长度为 N 的序列 f, g，定义 $f \oplus g = a$，其中序列 a 满足：

$$a_i = \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} [j \oplus k = i] f_j g_k$$

可以发现，对于初始给定的 a 数组，我们构造一个辅助数组 a' ， a'_i 表示 a 数组中位置编码为 i 的共有多少个，然后求出 $a' \oplus a'$ ，就能得到想要的“距离差数组”了。

当 $d = 2$ 时， $\oplus$ 和 xor 是等价的，可以使用快速沃尔什变换在时间限制内通过 $d = 2$ 的数据。

当 $d = 4$ 时，我们可以考虑将 d 进制扩大为 $2d-1$ 进制，每一维的范围为 $[-(d-1),(d-1)]$ ，这样我们就能去掉距离差函数中的绝对值，并将距离差函数直接改为两个数作差。

将 d 进制扩大为 $2d-1$ 进制后，新的序列的长度扩大为 $M = (\max a_i)^{\log_d 2d-1}$ 。使用快速傅里叶变换得到“距离差数组”的时间复杂度为

$$O(M \log M) = O((\max a_i)^{\log_d 2d-1} \log \max a_i) \approx O((\max a_i)^{1.40} \log \max a_i)$$

使用这种方法比较难在规定的时间限制内通过本题，因此我们考虑使用分治乘法。

3.5 多项式乘法问题

我们来看经典的多项式乘法问题：

给定两个长度为 N 的序列 a,b ，求序列 c ，满足 $c_i = \sum_{j=0}^{i} a_j b_{i-j}$

我们令 $A(x) = \sum a_k x^k$ ， $B(x) = \sum b_k x^k$ ， $C(x) = \sum c_k x^k$ ，相当于求两个多项式 $A(x), B(x)$ 相乘得到的多项式 $C(x)$ 的各项系数。

3.5.1 Karatsuba algorithm

对于多项式乘法问题，一般情况下我们都是先用快速傅里叶变换将 a,b 转化成点值的形式，得到 c 的点值后再插值得到序列 c 。

Karatsuba algorithm则给我们提供了一个不同的思路：利用分治来解决多项式乘法！它的基本流程如下（为了方便，我们假设 N 为2的幂）：

- 我们将 $A(x)$ 拆成两个多项式 $A_0(x), A_1(x)$ ，满足 $A(x) = A_0(x) + A_1(x)x^{\frac{N}{2}}$
- 同理我们将 $B(x)$ 拆成两个多项式 $B_0(x), B_1(x)$ ，满足 $B(x) = B_0(x) + B_1(x)x^{\frac{N}{2}}$
- 那么我们就有

$$C(x) = A(x)B(x) = A_0(x)B_0(x) + (A_0(x)B_1(x) + A_1(x)B_0(x))x^{\frac{N}{2}} + A_1(x)B_1(x)x^N$$

- 接下来需要快速计算 $A_0(x)B_1(x) + A_1(x)B_0(x)$ 这一项

- 注意到这三项系数的和较容易计算，即

$$A_0(x)B_0(x) + A_0(x)B_1(x) + A_1(x)B_0(x) + A_1(x)B_1(x) = (A_0(x) + A_1(x))(B_0(x) + B_1(x))$$

- 那么我们只需要求出 $(A_0(x) + A_1(x))(B_0(x) + B_1(x))$ ，就能得到

$$A_0(x)B_1(x) + A_1(x)B_0(x) = (A_0(x) + A_1(x))(B_0(x) + B_1(x)) - A_0(x)B_0(x) - A_1(x)B_1(x)$$

- 于是我们只用求三个规模降低为原来一半的子问题
  - $A_0(x)B_0(x)$
  - $A_1(x)B_1(x)$
  - $(A_0(x) + A_1(x))(B_0(x) + B_1(x))$
- 就能还原出 $C(x)$

对时间复杂度进行简单分析可以得到：

$$T(N) = 3T\left(\frac{N}{2}\right) + O(N) = O(N^{\log_2 3}) \approx O(N^{1.59})$$

这个算法在 $N = 100000$ ，开启O2优化的情况下在UOJ上大约需要 $0.7s$ 的运行时间。如果考虑模意义下的多项式乘法，运行时间大约是3模数NTT算法[20] 的1.5 ~ 2倍。

3.5.2 改进Karatsuba algorithm

在解决多项式乘法时，Karatsuba算法采取将多项式拆成两部分的方法来加快运算速度。

那么，如果我们将多项式拆成 $\gamma + 1$ 个（ γ 是一个常数），会不会得到更优的时间复杂度呢？

考虑两个 $(\gamma + 1)n - 1$ 次多项式的系数分别为 $a_{0...(\gamma+1)n-1}, b_{0...(\gamma+1)n-1}$ ，我们将 a, b 切割成 $\gamma + 1$ 个长度为 n 的片段 $A_0, ...A_\gamma$ 和$B_0, ..., B_\gamma$ 。令 $p = x^n$ ，且

$$A(p) = \sum_{k=0}^{\gamma} A_k p^k, B(p) = \sum_{k=0}^{\gamma} B_k p^k$$

那么

$$C(p) = A(p)B(p), \quad C_i = \sum_{j=0}^{i} A_j B_{i-j}$$

[20]这方面的快速算法可以见毛啸同学的 2015 年信息学奥林匹克中国国家队候选队员论文《再探快速傅里叶变换》

由于 $C(p)=\sum_{k=0}^{2\gamma}C_kp^k$ 是一个关于 p 的 2γ 次多项式，那么我们考虑取 $2\gamma+1$ 个点值带入后再还原 $C(p)$ 。

当我们取 $p=p_0$ 带入后，得到一个新的长度为 n 的序列 a' ，其中 $a'_i=\sum_{k=0}^{\gamma}a_{i+kn}p_0^k$ 。同理得到 b' ，那么我们需要求出以 a',b' 为系数的两个多项式相乘，问题规模降为原来的 $\frac{1}{\gamma+1}$ 。

除了递归以外的部分我们都可以在 $O(N)$ 的时间复杂度内完成，那么时间复杂度为：

$$T(N)=(2\gamma+1)T\left(\frac{N}{\gamma+1}\right)+O(N)=O(N^{\log_{\gamma+1}2\gamma+1})$$

可以发现，我们证明了以下定理[21]：

定理 3.1. *给定 $\epsilon>0$ ，存在一个与 N 无关的常数 $c(\epsilon)$ 和一个乘法算法，使得 $T(N)<c(\epsilon)N^{1+\epsilon}$*

实际上，Karatsuba算法可以看成 $\gamma=1$ 的特殊情况。

3.6 最终解法

回到原问题，我们尝试将Karatsuba算法的思想运用到本题中。考虑如何求 $a=f\oplus g$ ，f,g 长度为 N 。

在Karatsuba算法中，我们按把序列拆成两个来加快速度。

那么在 d 进制下，我们考虑将序列按照 d 进制下最高位进行拆分。用 a_i 表示序列 a 在 d 进制下最高位为 i 的那部分。我们把 a,f,g 分别进行拆分，可以发现（我们只考虑 $d=4$ 的情况）：

- $a_0=f_0\oplus g_0+f_1\oplus g_1+f_2\oplus g_2+f_3\oplus g_3$
- $a_1=f_0\oplus g_1+f_1\oplus g_0+f_1\oplus g_2+f_2\oplus g_1+f_2\oplus g_3+f_3\oplus g_2$
- $a_2=f_0\oplus g_2+f_2\oplus g_0+f_1\oplus g_3+f_3\oplus g_1$
- $a_3=f_0\oplus g_3+f_3\oplus g_0$

为了简化上面的式子，我们首先将奇数项和偶数项分离，考虑：

$$x'=(f_0+f_1+f_2+f_3)\oplus(g_0+g_1+g_2+g_3),\quad y'=(f_0-f_1+f_2-f_3)\oplus(g_0-g_1+g_2-g_3)$$

然后令 $x=(x'+y')/2,y=(x'-y')/2$ ，那么

$$x=\sum_{i=0}^{3}\sum_{j=0}^{3}[i\equiv j\pmod 2]f_i\oplus g_j,y=\sum_{i=0}^{3}\sum_{j=0}^{3}[i\not\equiv j\pmod 2]f_i\oplus g_j$$

[21]《计算机程序设计艺术卷2》 4.3.3 定理A

也就是说 $x = a_0 + a_2, y = a_1 + a_3$

现在我们考虑直接求出 a_3 ，由

$$u' = (f_0 + f_3) \oplus (g_0 + g_3), v' = (f_0 - f_3) \oplus (g_0 - g_3)$$

然后令 $u = (u' + v')/2, v = (u' - v')/2$ ，那么

$$u = f_0 \oplus g_0 + f_3 \oplus g_3, v = f_0 \oplus g_3 + f_3 \oplus g_0$$

于是 $a_3 = v, a_1 = y - a_3$

现在我们只需要求出 a_0, a_2 中的任意一个，观察上面我们求出的 u ，我们可以考虑求出 $f_1 \oplus g_1$ 和 $f_2 \oplus g_2$ ，那么：

$$a_0 = u + f_1 \oplus g_1 + f_2 \oplus g_2$$

并且 $a_2 = x - a_0$

于是我们将一个规模为 N 的问题拆分成 6 个规模为 $\frac{N}{4}$ 的子问题递归计算，于是我们可以得到时间复杂度为：

$$T(N) = 6T\left(\frac{N}{4}\right) + O(N) = O(N^{\log_4 6}) \approx O(N^{1.29})$$

在原问题中， $N = \max a_i$ ，复杂度可以估计为 $O((\max a_i)^{1.29})$ 。综上，我们完整地解决了该题.

4 分治乘法与位运算卷积

通过《Jellyfish》这道题，可以发现当我们将分治乘法的思想运用到位运算卷积中时，往往能获得不错的效果。

接下来作者将通过一些例子，展示分治乘法在位运算卷积中更多的应用[22]。

4.1 and, or 规则下的卷积

定义 4.1. 给定两个下标为 n 位二进制数的序列 a, b ，定义 $a \cdot b = c$ ，其中序列 c 需要满足：

$$c_i = \sum_{j=0}^{2^n-1} \sum_{k=0}^{2^n-1} [j \ and \ k = i] a_j b_k$$

我们称 c 为序列 a, b 在 and 规则下的卷积

[22] 4.1和4.2的内容在吕凯风同学2015年信息学奥林匹克中国国家队候选队员论文《集合幂级数的性质与应用及其快速算法》中也有讨论

仿照Karatsuba算法，我们将序列 a, b 按照最高位拆成两个序列 a_0, a_1 和 b_0, b_1 ，并且将 c 也拆成两个序列 c_0, c_1 ，那么：

- $c_0 = a_0 \cdot b_0 + a_0 \cdot b_1 + a_1 \cdot b_0$
- $c_1 = a_1 \cdot b_1$
- 根据 $c_0 + c_1 = (a_0 + a_1) \cdot (b_0 + b_1)$ ，得到：

$$c_0 = (a_0 + a_1) \cdot (b_0 + b_1) - c_1$$

这样我们只需要递归计算 $(a_0 + a_1) \cdot (b_0 + b_1)$ 和 $a_1 \cdot b_1$ 即可，时间复杂度为：

$$T(n) = 2T(n-1) + O(2^n) = O(n2^n)$$

or 规则下的卷积和 *and* 在本质上是相同的，只需要在 *and* 的基础上将下标二进制表示中的 $0, 1$ 互换就能得到 *or* 规则下卷积的做法。

4.2 二进制 *xor* 规则下的卷积

定义 4.2. 给定两个下标为 n 位二进制数的序列 a, b ，定义 $a \cdot b = c$ ，其中序列 c 需要满足：

$$c_i = \sum_{j=0}^{2^n-1} \sum_{k=0}^{2^n-1} [j \ xor \ k = i] a_j b_k$$

我们称 c 为序列 a, b 在 *xor* 规则下的卷积

我们同样对三个序列进行拆分，那么：

- $c_0 = a_0 \cdot b_0 + a_1 \cdot b_1$
- $c_1 = a_0 \cdot b_1 + a_1 \cdot b_0$
- 根据 $c_0 + c_1 = (a_0 + a_1) \cdot (b_0 + b_1)$ ，得到：

$$c_1 = (a_0 + a_1) \cdot (b_0 + b_1) - c_0$$

如果这么做，需要递归计算 $a_0 \cdot b_0, a_1 \cdot b_1, (a_0 + a_1) \cdot (b_0 + b_1)$ ，时间复杂度为：

$$T(n) = 3T(n-1) + O(2^n) = O(3^n)$$

当然，我们可以通过构造递归计算的多项式，来得到更优的解法：

- $x = (a_0 + a_1) \cdot (b_0 + b_1)$
- $y = (a_0 - a_1) \cdot (b_0 - b_1)$
- 那么 $c_0 = (x + y)/2, c_1 = (x - y)/2$

这样只需要递归计算 $(a_0 + a_1) \cdot (b_0 + b_1), (a_0 - a_1) \cdot (b_0 - b_1)$ ，时间复杂度为：

$$T(n) = 2T(n-1) + O(2^n) = O(n2^n)$$

4.3 K 进制 xor 规则下的卷积

定义 4.3. 定义 K 进制下的异或 xor_K ，为两个数在 K 进制下进行不进位加法得到的结果。形式化的：

$$x \ xor_K \ y = \sum_{i=0} \left(\left\lfloor \frac{x}{K^i} \right\rfloor + \left\lfloor \frac{y}{K^i} \right\rfloor \right) \mod K \times K^i$$

这样我们定义 K 进制 xor 规则下的卷积如下：

定义 4.4. 给定两个长度为 N 的序列 a, b ，定义 $a \cdot b = c$ ，其中序列 c 需要满足：

$$c_i = \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} [j \ xor_K \ k = i] a_j b_k$$

我们称 c 为序列 a, b 在 K 进制 xor 规则下的卷积

4.3.1 解决方法

通常情况下，我们使用每一维大小为 K 的高维快速傅里叶变换来解决这个问题。同样，我们可以用分治乘法来解决每一维大小为 K 的高维多项式乘法。

我们把序列 a, b 按照 K 进制下第一位拆成 K 个序列 $(a_0, a_1, .., a_{K-1})$ 和 $(b_0, b_1, ..., b_{K-1})$ ，同样对序列 c 进行拆分，那么我们有：

$$c_i = \sum_{j=0}^{K-1} \sum_{k=0}^{K-1} [j + k \equiv i \pmod K] a_j \cdot b_k$$

我们构造 $2K - 1$ 个序列 $d_0, d_1, ..., d_{2K-2}$ ，满足：

$$d_i = \sum_{j=0}^{K-1} \sum_{k=0}^{K-1} [j + k = i] a_j \cdot b_k$$

那么：

$$c_i = \sum_j [j \equiv i \pmod K] d_j$$

于是我们转而求序列 d 。

我们构造多项式 $A(x)=\sum a_k x^k, B(x)=\sum b_k x^k, D(x)=\sum d_k x^k$ ，那么我们只需要求 $D(x)=A(x)B(x)$ 。可以使用Karatsuba算法在将其变为 $t(K)K^{1.59}$ 个子问题递归解决。其中$t(K)$ 表示常数因子。

假设 N 是 K 的幂，那么时间复杂度可以估计为：

$$T(N)\approx t(K)K^{1.59}T\left(\frac{N}{K}\right)+t(K)K^{1.59}O\left(\frac{N}{K}\right)=O(N^{\log_K t(K)K^{1.59}})=O(N^{1.59+\log_K t(K)})$$

当 K 比较大的时候， $\log_K t(K)$ 可以忽略不计，时间复杂度[23]约为 $O(N^{1.59})$ 。

4.3.2 一个优化

由于我们实际上只需要求序列 c 而不需要完全求出序列 d ，因此当 K 为偶数时，我们可以采用一些优化。

注意到，在Karatsuba算法第一次拆分的时候，我们是这么进行的：

$$D(x)=A_0(x)B_0(x)+(A_0(x)B_1(x)+A_1(x)B_0(x))x^{\frac{K}{2}}+A_1(x)B_1(x)x^K$$

考虑下标在模 K 意义下相同的那些 d ，他们会被加到同一个 c_i 中，因此我们可以将 $D(x)$ 对 x^K 取模，也就是：

$$D(x)=A_0(x)B_0(x)+A_1(x)B_1(x)+(A_0(x)B_1(x)+A_1(x)B_0(x))x^{\frac{K}{2}}$$

那么我们构造两个多项式 $u(x),v(x)$ ，满足：

$$u(x)=(A_0(x)+A_1(x))(B_0(x)+B_1(x))$$

$$v(x)=(A_0(x)-A_1(x))(B_0(x)-B_1(x))$$

于是

$$D(x)=\frac{u(x)+v(x)}{2}+\frac{u(x)-v(x)}{2}x^{\frac{K}{2}}$$

这样，我们就把 $t(K)$ 缩小到了原来的 $\frac{2}{3}$ ，当 K 比较小时，具有比较显著的效果。

4.4 更复杂位运算规则下的卷积

即使是更为复杂的位运算规则，分治乘法也常常能得到一个不错的算法。

考虑下面的例子：

[23]这里也可以用3.5.2中提到的方法来改进时间复杂度

定义 4.5. 定义运算 $x*y$ 表示两个十进制数 x,y 按位进行不进位乘方得到的结果。

形式化的：

$$x*y=\sum_{k=0}\left\lfloor\frac{x}{10^k}\right\rfloor^{\left(\left\lfloor\frac{y}{10^k}\right\rfloor \mod 10\right)} \mod 10\times 10^k$$

特别地，令 $0^0=0$

我们定义 $*$ 规则下的卷积如下：

定义 4.6. 给定两个长度为 N 的序列 a,b ，定义 $a\cdot b=c$ ，其中序列 c 需要满足：

$$c_i=\sum_{j=0}^{N-1}\sum_{k=0}^{N-1}[j*k=i]a_jb_k$$

我们称 c 为 a,b 在 $*$ 规则下的卷积。

4.4.1 解决方法

可以发现，运算符 $*$ 几乎没有什么好的性质，无论是交换律，结合律还是分配率，它都不满足。

但是我们仍然可以利用分治乘法的思想，将复杂度降低到 $O(N^2)$ 以下。

考虑两个在 $[0,10)$ 之间的数 x,y ，我们可以把一些 $x^y \mod 10$ 相同的 x,y 合并。例如当 $x=0,1,5,6$ 时， y 取任意正整数， x^y 的值都不会改变。基于“合并”的思想，我们可以尝试降低时间复杂度。

首先我们还是将序列 a,b,c 按照10进制下最高位分别拆成10个序列。

为了方便描述，我们先做一些记号的约定：

记号 4.1. 令 $U=\{0,1,...,9\},V=\{1,2,...,9\}$

记号 4.2. 给定两个集合 $S,T\subseteq U$ ，记

$$g_{S,T}=\sum_{i\in S}\sum_{j\in T}a_i\cdot b_j$$

可以发现：

$$g_{S,T}=\left(\sum_{i\in S}a_i\right)\cdot\left(\sum_{j\in T}b_j\right)$$

也就是给定 S,T ，我们只需要递归做一次规模为原问题 $\frac{1}{10}$ 的子问题就能求出 $g_{S,T}$

记号 4.3. 给定 $x, y \in V$ ，令

$$F(x, y) = \{i \mid i \in V, x^i \equiv y \pmod{10}\}$$

$$h_{x,y} = \sum_{i \in F(x,y)} a_x \cdot b_i$$

显然， $h_{x,y} = g_{\{x\},F(x,y)}$ ，因此我们只需要一次递归就能求出 $h_{x,y}$

注意到，对某一个 x ，满足 $F(x, y) \neq \varnothing$ 的 y 并不会太多，我们对每个 $x \in V$ 进行考虑：

- 当 $x = 1$ 时，只需求 $h_{x,1}$
- 当 $x = 2$ 或 $x = 8$ 时，只需求 $h_{x,2}, h_{x,4}, h_{x,6}, h_{x,8}$
- 当 $x = 3$ 或 $x = 7$ 时，只需求 $h_{x,1}, h_{x,3}, h_{x,7}, h_{x,9}$
- 当 $x = 4$ 时，只需求 $h_{x,4}, h_{x,6}$
- 当 $x = 5$ 时，只需求 $h_{x,5}$
- 当 $x = 6$ 时，只需求 $h_{x,6}$
- 当 $x = 9$ 时，只需求 $h_{x,1}, h_{x,9}$

可以看到，满足 $F(x, y) \neq \varnothing$ 的有序数对 $x, y(x, y \in V)$ 只有上述23个。我们直接暴力递归解23个子问题来求出上面的这些 $h_{x,y}$

现在假设我们得到了 $d_1, ..., d_9$ ，满足

$$d_k = \sum_{x \in V} h_{x,k}$$

接下来我们考虑 $x = 0$ 或 $y = 0$ 的情况：

- 当 $x = 0$ 时， $x^y \equiv 0 \pmod{10}$
- 当 $x \neq 0, y = 0$ 时， $x^y \equiv 1 \pmod{10}$

在序列 d 的基础上加入上述两种情况，我们就能得到最终的 c 数组：

- $c_0 = g_{\{0\},U}$

- $c_1 = d_1 + g_{V,\{0\}}$
- 对于其他 c_k， $c_k = d_k$

这样，我们只需要递归做25个规模为原问题 $\frac{1}{10}$ 的子问题，就能得到序列 c 。

时间复杂度为：

$$T(N) = 25T\left(\frac{N}{10}\right) + O(N) = O(N^{\log_{10} 25}) \approx O(N^{1.40})$$

4.4.2 一些简单的优化

在上面的做法中，我们简单地按照 x 计算 $h_{x,y}$ ，得到了一个算法。

实际上，我们求序列 d 时，是按照 y 将 $h_{x,y}$ 进行求和的。因此，有些 $h_{x,y}$ 是可以被合并的，例如：

- $h_{2,4} + h_{8,4} = g_{\{2,8\},\{2,6\}}$
- $h_{2,6} + h_{8,6} = g_{\{2,8\},\{4,8\}}$
- $h_{3,9} + h_{7,9} = g_{\{3,7\},\{2,6\}}$
- $h_{3,1} + h_{7,1} = g_{\{3,7\},\{4,8\}}$

这样，我们减少了4次递归，时间复杂度降为：

$$T(N) = 21T\left(\frac{N}{10}\right) + O(N) = O(N^{\log_{10} 21}) \approx O(N^{1.32})$$

另外，我们发现：

- $h_{2,2} + h_{8,2} + h_{2,8} + h_{8,8} = g_{\{2,8\},\{1,3,5,7,9\}}$
- $h_{2,2} + h_{8,2} - h_{2,8} - h_{8,8} = (a_2 - a_8) \cdot (b_1 - b_3 + b_5 - b_7 + b_9)$

于是我们只要两次递归就能分别求出 $h_{2,2} + h_{8,2}$ 和 $h_{2,8} + h_{8,8}$ ，可以减少两次递归。时间复杂度进一步降为：

$$T(N) = 19T\left(\frac{N}{10}\right) + O(N) = O(N^{\log_{10} 19}) \approx O(N^{1.28})$$

4.5 多个序列为自变量的函数

一般情况下的卷积都是以两个序列为自变量的。

如果有三个或更多的序列作为自变量，就难以通过某种变换来加速运算了。

但是应用分治乘法，我们还是能够找出一个比直接暴力更加优秀的算法。

来看下面的例子：

定义 4.7. 已知一个函数 $f(x,y,z)$，定义域为 $\{(x,y,z) \mid x,y,z \in \{0,1\}\}$，值域为 $\{0,1\}$，并且满足 $f(0,0,0)=0$

定义一个函数 $g(x,y,z)$ 为 x,y,z 按位求 f 函数得到的结果。

形式化的，

$$g(x,y,z) = \sum_{k=0} 2^k f(x_k, y_k, z_k)$$

其中 x_i 表示 x 二进制表示下从低到高第 i 位

定义 4.8. 给定三个长度为 N 的序列 a,b,c，定义一个函数 $H(a,b,c)$。令 $d=H(a,b,c)$，那么：

$$d_i = \sum_{j=0}^{N-1}\sum_{k=0}^{N-1}\sum_{l=0}^{N-1}[g(j,k,l)=i]a_j b_k c_l$$

我们令 $S=\{(x,y,z) \mid f(x,y,z)=0\}, T=\{(x,y,z) \mid f(x,y,z)=1\}$

将序列 a,b,c,d 按照最高位拆分后，显然有：

$$d_0 = \sum_{x,y,z,(x,y,z)\in S} H(a_x, b_y, c_z)$$

$$d_1 = \sum_{x,y,z,(x,y,z)\in T} H(a_x, b_y, c_z)$$

我们知道 $S \cap T = \varnothing$，并且 $|S|+|T|=8$，那么必然有 $\min(|S|,|T|) \le 4$

因此我们根据 $|S|$ 和 $|T|$ 的大小决定暴力递归计算出 d_0 还是 d_1，然后通过

$$d_0 + d_1 = H(a_0+a_1, b_0+b_1, c_0+c_1)$$

得到另外一项。

由于 $\min(|S|,|T|) \le 4$，因此最多递归计算4+1=5次。

时间复杂度为 $T(N) = 5T\left(\frac{N}{2}\right) + O(N) = O(N^{\log_2 5}) \approx O(N^{2.32})$

5 总结

《Jellyfish》是一道结合了数学、动态规划和多项式的较为综合的题目，具有一定的思维难度和代码实现难度。选手需要通过期望的有关知识简化问题，并通过动态规划和分治乘法来优化算法。

运用分治乘法来解决一些特殊的按位运算规则下的卷积问题的思想，作者最早是在黄致焕学长的校内模拟赛中接触到的。之后作者对这类问题进行了一些研究，发现分治乘法常常能取得不错的效果，因此将《Jellyfish》作为集训队第一轮作业的自选题并与大家交流讨论，希望能对大家有所启发。

6 感谢

感谢CCF提供学习和交流的平台。

感谢父母对我的培养和教育。

感谢学校的栽培，应平安老师的教导和同学们的帮助。

感谢北京大学黄致焕学长对我们的指导。

感谢绍兴一中任轩笛，西工大附中刘承奥与我交流讨论、给我启发，并且为本文审稿。

参考文献

[1] Karatsuba算法在英文维基百科资料：https://en.wikipedia.org/wiki/Karatsuba_algorithm

[2] 吕凯风，《集合幂级数的性质与应用及其快速算法》，2015年信息学奥林匹克中国国家队候选队员论文。

[3] 毛啸，《再探快速傅里叶变换》，2015年信息学奥林匹克中国国家队候选队员论文。

[4] Donald E. Knuth，《计算机程序设计艺术》，国防工业出版社

Leafy Tree 及其实现的加权平衡树

成都市第七中学 王思齐

摘 要

二叉树与平衡树是信息学竞赛中重要的数据结构。其在维护集合或序列时有非常大的用途。

本文介绍了 Leafy Tree，这是一种能实现大多数基于二叉树的数据结构的数据结构。本文描述了其实现二叉搜索树和加权平衡树的方法，并给出了这两种数据结构与其他数据结构的比较。

1 前言

近年来，信息学竞赛中对于数据结构类问题的考察逐渐增加。许多问题需要使用二叉树（如线段树、平衡树、左偏树等）进行维护，而其中平衡树是一个重要的考察内容。

常见的平衡树有 Splay、Treap 等，而他们或多或少存在着一些问题，比如常数大、代码量大、难以可持久化等。针对这些问题，本文引入了一种新的二叉树结构——Leafy Tree，以及一种新的平衡策略——加权平衡。结合这两者，平衡树的大部分问题可以被克服。

本文第二节中，介绍了 Leafy Tree。

第三节中，描述了 Leafy Tree 实现二叉搜索树的方法。

第四节中，引入了加权平衡树，并描述了 Leafy Tree 实现加权平衡树的方法。

2 什么是 Leafy Tree

Leafy Tree 是一种二叉树，其每个节点要么为叶子，要么有两个儿子。其信息完全储存在叶子上面，每个非叶节点存储的信息是其儿子的信息的合并。它的结构和线段树类似，同时也可以将其看作是二叉搜索树的 *kruskal* 重构树（这里 *kruskal* 重构树并不是指最小生成树的 *kruskal* 重构树，而是对一棵二叉搜索树进行类似的操作，每次合并两个节点并把中间的边在新图上变成一个节点，得到的树）。

这是一个纯函数化的数据结构，可以方便高效地实现其他的函数化数据结构，同时也可以方便的实现可持久化。

3 Leafy Tree 实现二叉搜索树

3.1 定义

考虑用一棵 Leafy Tree 维护一个集合。每个叶子节点存储集合中一个值，每个非叶节点保存它右儿子的值（即子树最大值）。同时满足每个非叶节点左儿子的最大值不大于右儿子的最小值，即这棵树的中序遍历中，叶节点的值是有序的。我们把这样的一棵 Leafy Tree 称为 Leafy Tree 实现的二叉搜索树。

3.2 基本操作

为了方便描述，我们用 $A.left, A.right, A.value$ 表示节点 A 的左儿子、右儿子和值。对于叶节点 X，定义 $X.left = X.right = null$。同时定义函数 $newNode(x)$ 用来新建一个值为 x 的节点，并将新建节点的左右儿子设为 $null$， $deleteNode(x)$ 用来回收节点 x。

3.2.1 查询

在节点 T 的子树中查找值 x 的过程为：

若 T 为叶节点，则比较 x 与 T 的值，如果值相等，则查找成功，否则查找失败；

否则若 x 不大于 T 的左子树的值，搜索左子树；

否则搜索右子树。

伪代码见下面算法 1。

算法 1 Leafy Tree 实现二叉搜索树的查询操作

Find 函数参数为节点 T 、查询的数 x，返回查询结果。

```
1:  function Find( T, x )
2:    if T.left = null then
3:      if T.value = x then
4:        return True
5:      else
6:        return False
7:      end if
8:    else
9:      if x ≤ T.left.value then
10:       return Find( T.left, x )
11:     else
12:       return Find( T.right, x )
13:     end if
```

14: **end if**
15: **end function**

3.2.2 插入

在节点 T 的子树中插入值 x 的过程为：

若 T 为叶节点，则比较 x 与 T 的值，如果 x 大于 T 的值，则新建节点 A, B 来存放 T 的值，并将 T 的左右儿子设为 A 和 B； x 不大于 T 的值时操作类似；

否则若 x 不大于 T 的左子树的值，将 x 插入左子树；

否则将 x 插入右子树。

伪代码见下面算法 2。

算法 2 Leafy Tree 实现二叉搜索树的插入操作

Insert 函数参数为节点 T 、插入的数 x 。

1: **function** Insert(T, x)
2: **if** $T.left = null$ **then**
3: $A \leftarrow$ newNode(T.value)
4: $B \leftarrow$ newNode(x)
5: **if** $A.value > B.value$ **then**
6: swap(A, B)
7: **end if**
8: $T.left \leftarrow A$
9: $T.right \leftarrow B$
10: $T.value \leftarrow B.value$
11: **else**
12: **if** $x \le T.left.value$ **then**
13: Insert($T.left, x$)
14: **else**
15: Insert($T.right, x$)
16: **end if**
17: $T.value \leftarrow T.right.value$
18: **end if**
19: **end function**

3.2.3 删除

在节点 T 的子树中删除值 x 的过程为：

若 T 为叶节点，则比较 x 与 T 的值，如果 x 与 T 的值相等，则删除成功，并将 T 父亲的所有属性设为 T 兄弟的，然后删除 T 的兄弟节点与 T 本身；否则删除失败；

否则若 x 不大于 T 的左子树的值，在左子树删除；

否则在右子树删除。

伪代码见下面算法 3。

算法 3 Leafy Tree 实现二叉搜索树的删除操作

Remove 函数参数为节点 T 、删除的数 x ，返回是否删除成功。这个函数要求 T 为非叶节点。

```
1:  function Remove( T, x )
2:    if x ≤ T.left.value then
3:      if T.left.size = 1 then
4:        if T.left.value = x then
5:          temp ← T.right
6:          T = T.right
7:          deleteNode( temp )
8:          deleteNode( T.left )
9:          return True
10:       else
11:         return False
12:       end if
13:     else
14:       return Remove( T.left, x )
15:     end if
16:   else
17:     if T.right.size = 1 then
18:       if T.right.value = x then
19:         temp ← T.left
20:         T = T.left
21:         deleteNode( temp )
22:         deleteNode( T.right )
23:         return True
24:       else
25:         return False
26:       end if
27:     else
28:       temp ← Remove( T.right, x )
```

```
29:         T.value ← T.right.value
30:         return temp
31:       end if
32:     end if
33: end function
```

虽然这里伪代码较长，但是实际实现上，如果保证被删除数存在，并用数组来存储每个节点的两个儿子，可以节省大量代码。

3.3 和二叉搜索树的比较

可以发现，由于用到了只有叶子节点维护信息的性质，Leafy Tree 进行插入删除相当方便。每次插入删除的节点实际上都是叶子节点，大量减少了普通二叉搜索树结构中对插入删除的节点类型的繁杂讨论（即这个节点是叶子节点，还是有一个儿子，还是有两个儿子）。同时由于每个非叶子节点维护了区间信息，通过左儿子的区间信息来进行定位，本质上和二叉搜索树通过与树根节点的值比较进行定位相同，所以是以更加简洁的实现达到了同样的效果。

4 Leafy Tree 实现加权平衡树

加权平衡树（Weight Balanced Tree，也叫 BB[α] 树，重量平衡树）是一种储存子树大小的二叉搜索树。即一个结点包含以下字段：值（value）、左儿子（left）、右儿子（right）、子树大小（size）。

一个结点的权重 $weight$ 取决于它的子树大小或者等于它的子树大小。具体来说，需要满足 $weight[x] = weight[x.left] + weight[x.right]$ 。如果一个节点 x 满足 $min(weight[x.left], weight[x.right]) \geq \alpha \cdot weight[x]$ ，则称这个节点是 α 加权平衡的，显然 $0 < \alpha \leq \frac{1}{2}$ 。一棵含有 n 个元素的加权平衡树的高度 h 满足 $h \leq \log_{\frac{1}{1-\alpha}} n = O(\log n)$ 。

4.1 定义

在 Leafy Tree 实现的加权平衡树中，子树大小为子树中叶节点的个数，非叶节点的值为其右儿子的值。即对于一个叶节点 x ， $x.size = 1$ ；对于一个非叶节点 x ， $x.size = x.left.size + x.right.size, x.value = x.right.value$ 。而一个节点的权重就是它的子树大小，即对于一个节点 x ， $weight[x] = x.size$ 。

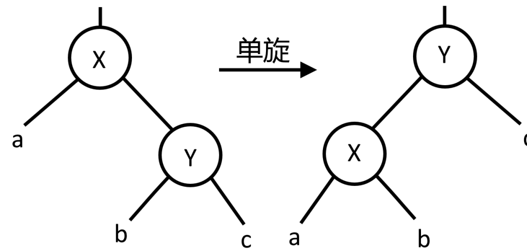
4.2 实现方式

加权平衡树有两种实现方式，重构和旋转。这里只介绍旋转平衡。

对于一棵满足 α 加权平衡的树，在插入或删除一个节点后，不满足 α 加权平衡的节点一定在一条链上。考虑依次处理这条链上的节点。

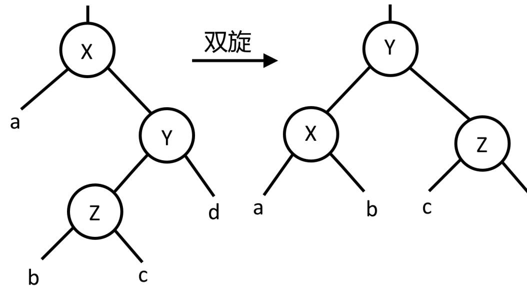
假设这棵树的一个子树 T 中，除根节点外的所有节点均满足 α 加权平衡。我们可以用一次单旋或双旋操作使 T 满足 α 加权平衡。

定义 x 的平衡度表示 $\frac{weight[x.left]}{weight[x]}$ 。



如上图，考虑一次单旋，设 X, Y 旋转前平衡度为 ρ_1, ρ_2 ，旋转后为 γ_1, γ_2 。

可以得到 $\gamma_1 = \frac{\rho_1}{\rho_1+(1-\rho_1)\rho_2}, \gamma_2 = \rho_1 + (1-\rho_1)\rho_2$ 。



如上图，考虑一次双旋，设 X, Y 旋转前平衡度为 ρ_1, ρ_2, ρ_3 ，旋转后为 $\gamma_1, \gamma_2, \gamma_3$ 。

可以得到 $\gamma_1 = \frac{\rho_1}{\rho_1+(1-\rho_1)\rho_2\rho_3}, \gamma_2 = \rho_1 + (1-\rho_1)\rho_2\rho_3, \gamma_3 = \frac{\rho_2(1-\rho_3)}{1-\rho_2\rho_3}$ 。

假设 $\rho_1 < \alpha$ ，在只插入或删除一个节点的情况下， ρ_1 最小为 $\frac{\alpha}{2-\alpha}$ ，可在子树 a 中原有 2 个节点，现在删除其中一个时取到。

在给定的限制下，可以证明，当 $\alpha \le 1 - \frac{\sqrt{2}}{2}$ 时，通过这两种操作，可以使平衡度改变的这几个节点，其新平衡度在 $[\alpha, 1-\alpha]$ 内。证明时可能还需要进一步对 ρ_1 进行放缩，以及小范围的分类讨论。具体证明由于篇幅所限，不再赘述。

具体操作为，当 $\rho_2 < \frac{1-2\alpha}{1-\alpha}$ 时，进行一次单旋，否则进行一次双旋。

4.3 基本操作

为了方便描述，在 $left$ 和 $right$ 之外，我们同时用 $A.child[0], A.child[1]$ 表示节点 A 的左儿子和右儿子。对于叶节点 X ，定义 $X.child[0] = X.child[1] = null$ 。同时定义函数 $newNode(x)$ 用来新建一个值为 x 的节点，并将新建节点的左右儿子设为 $null$ ， $deleteNode(x)$ 用来回收节点 x 。

4.3.1 合并信息

一个节点经过操作后，需要重新计算其子树大小及值。

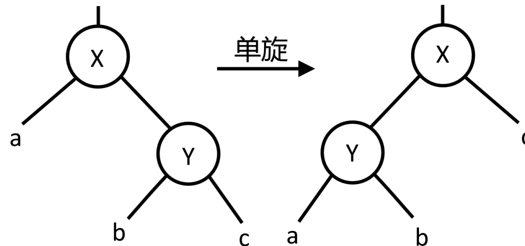
伪代码见下面算法 4。

算法 4 Leafy Tree 实现的加权平衡树的信息合并

Pushup 函数参数为节点 T 。

1: **function** Pushup(T)
2:   **if not** $T.left = null$ **then**
3:     $T.size \leftarrow T.left.size + T.right.size$
4:     $T.value \leftarrow T.right.value$
5:   **end if**
6: **end function**

4.3.2 单旋



如图，这个操作是将 Y 单旋至 X 的位置。这里为了不改变指向 X 的指针，在单旋后交换了它们的位置。

伪代码见下面算法 5。其中 $\oplus$ 表示异或。

算法 5 Leafy Tree 实现的加权平衡树的单旋

Rotate 函数参数为节点 T 和一个 bool 变量 d ，表示是将 $child[d]$ 旋转到 T 的位置。

1: **function** Rotate(T, d)
2:   $temp \leftarrow T.child[d \oplus 1]$
3:   $T.child[d \oplus 1] \leftarrow T.child[d]$
4:   $T.child[d] \leftarrow T.child[d \oplus 1].child[d]$

5: $T.child[d \oplus 1].child[d] \leftarrow T.child[d \oplus 1].child[d \oplus 1]$
6: $T.child[d \oplus 1].child[d \oplus 1] \leftarrow temp$
7: Pushup($T.child[d \oplus 1]$)
8: Pushup(T)
9: **end function**

4.3.3 维护平衡

一个节点经过操作后，需要一次单旋或双旋来维护平衡。

伪代码见下面算法 6。

算法 6 Leafy Tree 实现的加权平衡树的维护平衡

Maintain 函数参数为节点 T 。

1: **function** Maintain(T)
2: **if** $not\ T.left = null$ **then**
3: **if** $T.left.size < T.size \cdot \alpha$ **then**
4: $d \leftarrow 1$
else if $T.right.size < T.size \cdot \alpha$ **then**
5: $d \leftarrow 0$
else $T.right.size < T.size \cdot \alpha$
6: **return**
7: **end if**
8: **if** $T.child[d].child[d \oplus 1].size \geq T.child[d].size \cdot \frac{1-2\alpha}{1-\alpha}$ **then**
9: Rotate($T.child[d], d \oplus 1$)
10: **end if**
11: Rotate(T, d)
12: **end if**
13: **end function**

4.3.4 插入

与 Leafy Tree 实现的二叉搜索树类似。

伪代码见下面算法 7。

算法 7 Leafy Tree 实现的加权平衡树的插入操作

Insert 函数参数为节点 T 、插入的值 x

1: **function** Insert(T, x)
2: **if** $T.size = 1$ **then**

```
3:     T.left ← newNode(x)
4:     T.right ← newNode(T.value)
5:     if x > T.value then
6:       swap( T.left, T.right )
7:     end if
8:   else
9:     Insert( T.child[x > T.left.value], x )
10:  end if
11:  Pushup( T )
12:  Maintain( T )
13: end function
```

4.3.5 删除

与 Leafy Tree 实现的二叉搜索树类似。

伪代码见下面算法 7。

算法 7 Leafy Tree 实现的加权平衡树的删除操作

Remove 函数参数为节点 T 、删除的值 x 。这个函数要求 T 为非叶节点。

```
1: function Remove( T, x )
2:   d ← x > T.left.value
3:   if T.child[d].size = 1 then
4:     if T.child[d].value = x then
5:       deleteNode( T.child[d] )
6:       temp ← T.child[d ⊕ 1]
7:       T.left = T.child[d ⊕ 1].left
8:       T.right = T.child[d ⊕ 1].right
9:       T.value = T.child[d ⊕ 1].value
10:      deleteNode( temp )
11:    else
12:      return
13:    end if
14:  else
15:    Remove( T.child[d] )
16:  end if
17:  Pushup( T )
18:  Maintain( T )
```

19: **end function**

4.3.6 查询

与 Leafy Tree 实现的二叉搜索树相同。

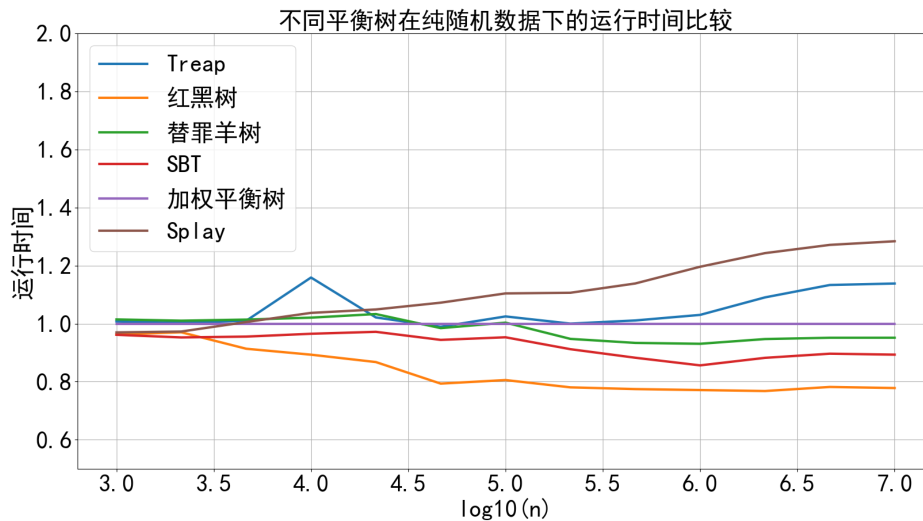
伪代码见上面算法 1。

4.4 运行速度

这里选取了 LOJ #104 普通平衡树一题中，运行较快的一些代码进行测试。

由于不同的数据生成器的测试结果大致相同，这里只展示纯随机数据的测试结果[24]。

这里的结果可能有一定误差，并且代码也并不是保证常数最优秀的，但是能大致反应出这几种平衡树的运行速度。



如图，这里运行时间为该平衡树实际运行时间与加权平衡树实际运行时间的比值。

可以发现，加权平衡树虽然比红黑树和 SBT 慢，但是比 Treap 和 Splay 快，和替罪羊树运行速度相近。

4.5 其他操作

除了基本的插入删除等操作外，加权平衡树还可以用优秀的复杂度实现合并、分裂等操作，以及实现可持久化。

[24]测试代码及原始数据可以在 https://mcfx.us/ctsc 下载。

4.5.1 合并

合并操作是指，给定两棵树 A 和 B ，保证 A 中最大值不大于 B 中的最小值，要求将它们合并为一棵新树。这个操作的复杂度为 $O(\log \frac{max(A.size,B.size)}{min(A.size,B.size)})$ 。

首先，假设 $A.size \ge B.size$ 。如果 $B.size \ge \frac{\alpha}{1-\alpha} \cdot A.size$ ，那么可以直接新建一个节点，其左儿子为 A ，右儿子为 B 。

否则，如果 $A.left.size \ge \alpha \cdot (A.size + B.size)$ ，可以递归合并 A 的右儿子和 B ，再与 A 的左儿子合并。

否则可以递归合并 A 的左儿子与 A 的右儿子的左儿子，以及 A 的右儿子的右儿子和 B ，最后再把这两次合并的结果进行合并。

复杂度证明受篇幅所限，不再赘述。

4.5.2 分裂

分裂操作是指，给定一棵树 T ，要求将其中不大于 k 的部分和大于 k 的部分各自独立为一棵新树。

这个操作可以直接递归进行，并对返回的结果调用上面的合并操作。复杂度为 $O(\log T.size)$ 。

复杂度证明受篇幅所限，不再赘述。

4.5.3 可持久化

对于每次操作，可以复制修改的所有节点，并依次新建一个（即 Path copy）。单次插入或删除修改的节点显然是 $O(\log size)$ 的。

Leafy Tree 实现的加权平衡树中，单次操作期望下修改的节点个数很少，且不需要新建冗余的节点。故其内存占用小，常数也小，与其他平衡树（如非旋转 Treap）相比有巨大优势。

5 总结

本文介绍了 Leafy Tree 及其实现的加权平衡树。

Leafy Tree 和二叉搜索树相比，其插入删除操作更容易实现，但是会使用两倍的节点。

加权平衡树的常数较小，实现较简单，同时还可以方便的实现可持久化。与常数更小的红黑树相比，可以不用记录额外信息（节点大小可以用来截取区间或求第 K 小值），并且实现方便得多。与实现简单的 $Splay$ 相比，代码量基本相同，而常数更小，可以可持久化。与 $Treap$ 相比，不用记录额外信息且常数更小。与替罪羊树相比，复杂度不是均摊的，可以实现可持久化。

而将这两者结合起来的加权平衡的 Leafy Tree，则可以在较短的代码中，实现平衡树的大多数功能，而且常数较小，可以可持久化。

除此之外，Leafy Tree 还可以实现各种基于二叉树的数据结构，如其他平衡树，线段树，堆等。但由于篇幅所限，本文不能一一介绍。

感谢

感谢中国计算机学会提供学习和交流的平台。

感谢成都七中张君亮、蔺洋老师多年以来的关心和指导。

感谢成都七中李欣隆、安师大附中赵雨扬同学为本文提供思路。

感谢其他帮助过我的老师、同学们。

参考文献

[1] Nievergelt, J. and Reingold, E. M. (1972) Binary search trees of bounded balance. In *Proceedings of the Fourth Annual Acm Symposium on Theory of Computing*. ACM, pp. 137 – 142.

[2] Wikipedia, Weight-balanced tree

[3] Wikipedia, Binary search tree

[4] LOJ #104 普通平衡树, https://loj.ac/problem/104

《小H爱染色》命题报告

浙江省杭州第二中学 陈嘉乐

摘 要

本文将介绍笔者2018年集训队互测第五轮中命制的一道题，其具体考察了组合数、斯特林数、二项式反演、多项式等知识以及少量计数技巧，要求选手有扎实的基本功和一定的思维能力。

除此之外，题目的部分分设置合理，区分度高，意在引导选手向正解方向靠拢。同时，解决本题的思路多样，选手可以从不同的方向深入思考，均能获得高分。

1 试题

1.1 题目描述

有排成一列的 n 个球，编号依次为0到 $n-1$ ，初始都为白色。小 H 会重复以下操作共2次：随机取 m 个球染黑（可以重复染黑球）。小 H 对编号最小的黑球情有独钟，她想知道，如果令 A 为它的编号， $F(A)$ 的期望是多少。其中， $F(x)$ 为一个次数不超过 m 的多项式， $F(A)$ 表示 $x=A$ 时多项式的值。

1.2 输入格式

第一行两个整数 n ， m 。第二行 $m+1$ 个整数，第 i 个整数为 $F(i-1)$ 。

1.3 输出格式

一行一个整数，如果令 E 表示 $F(A)$ 的期望，输出 $E\times\binom{n}{m}^2$ 模998244353的值。

1.4 样例输入

8 5
45856608 386378255 106492167 28766400 272276589 93721672

1.5 样例输出

321347828

1.6 数据规模及约定

对于 10% 的数据， $n \leq 10$ ， $m \leq 5$ 。
对于 20% 的数据， $n \leq 100$ ， $m \leq 100$ 。
对于 30% 的数据， $n \leq 1000$ ， $m \leq 1000$ 。
对于另外 5% 的数据， $m \leq 1000000$ ，且保证多项式 $F(x) = 1$ 。
对于另外 5% 的数据， $m \leq 1000000$ ，且保证多项式 $F(x) = x$ 。
对于另外 10% 的数据， $m \leq 5000$ ，且保证多项式 $F(x) = x^m$ 。
对于 70% 的数据， $m \leq 5000$ 。
对于 80% 的数据， $m \leq 20000$ 。
对于 100% 的数据， $n \leq 998244353$ ， $m \leq 1000000$ 。

1.7 时空限制

时间限制：2s
空间限制：512M

2 算法介绍

2.1 算法1

首先插值将多项式转化成系数表示的形式，并求出该多项式在 $0 \cdots n-1$ 的点值。枚举每一种可能的染色方案，找到其中的最小值，将贡献累加。

预处理复杂度 $O(nm)$ 。统计答案复杂度 $O\left(\binom{n}{m}^2 \times poly(n)\right)$ 。

期望得分10分。

2.2 算法2

记 $f_{i,j,k}$ 为从后往前染色染到编号为 i 的球，且第一次染色中染了 j 个球，第二次 k 个球的方案数。有四种转移，分别为，当前球不染色，第一次染色中染黑，第二次中染黑，两次都染.则

$f_{i,j,k} = f_{i+1,j,k} + [j > 0] \times f_{i+1,j-1,k} + [k > 0] \times f_{i+1,j,k-1} + [j > 0] \times [k > 0] \times f_{i+1,j-1,k-1}$;

$ans = \sum_{i=0}^{n-1} F(i) \times (f_{i,m,m} - f_{i+1,m,m})$ 。

复杂度为 $O(nm^2)$ 。

期望得分20分。

2.3 算法3

实际上 $f_{i,m,m}$ 可以直接计算而不用递推得到。

考虑大于等于 i 的数的个数为 $n-i$ ，因此每种颜色的染色方案实际都为 $\binom{n-i}{m}$ 。

即 $f_{i,m,m}=\binom{n-i}{m}^2$ 。

使用同样的方式计算贡献，总复杂度为 $O(nm)$ 。

期望得分30分。

2.4 算法4

当 n 达到了 10^9 的级别，我们难以枚举最小值，不妨尝试转变思路。

部分分在设置时希望提供选手从贡献多项式 $F(x)$ 角度出发，获得灵感。

2.4.1 $F(x)=1$

等于说每种不同的染色方案对答案的贡献相同，都为1，那么答案就是方案数。即 $ans=\binom{n}{m}^2$ 。

由于 $\binom{n}{m}=\binom{n}{m-1}\times\frac{n-m+1}{m}$ ，预处理逆元后可以 $O(m)$ 计算。

期望得分5分。

2.4.2 $F(x)=x$

寻找期望的等价形式。即编号小于 A 的球的数量的期望。再根据期望的线性性，我们要算的就是每个球 i 编号小于 A 的概率的和。

不妨将球 i 与染黑的球共同枚举。令 k 为最终染黑的球的数量，由于两次各染 m 个，那么 $k\in[m,2m]$ 。考虑染色的方案数。第一次随意染，方案数为 $\binom{k}{m}$ 。由于选择的位置都要被染色，因此第一次没染的一定要染黑，而多余的只能染到已经被染黑的球上，方案数为 $\binom{m}{2m-k}$ 。那么总的方案数 $H_k=\binom{k}{m}\times\binom{m}{2m-k}$ 。 H 数组在预处理阶乘和阶乘逆元后可以线性求出。

统计答案时直接选出 $k+1$ 个球，方案数为 $\binom{n}{k+1}$ 。由于球 i 编号比 A 小，因此染色的就是编号较大的k个，是一一对应的。

那么 $ans=\sum_{k=m}^{2m}\binom{n}{k+1}H_k$ 。

总复杂度 $O(m)$ 。

期望得分5分。

2.4.3 $F(x) = x^c$

与之前类似，等价形式为，可重复的选择 c 个球，编号全都小于 A 的期望。根据期望的线性性，要算的是每种可重复选择 c 个球的方案， c 个球编号都小于 A 的概率的和。

注意到被选择的球可重复，我们不妨求去重后选择的球数量为 q 时的方案数 $G_q(q = 0\cdots c)$ 。

考虑二项式反演。随意选择的总次数为 $q^c = \sum_{i=0}^{q}\binom{q}{i}G_i$ ，那么 $G_q = \sum_{i=0}^{q}\binom{q}{i}(-1)^{q-i}i^c$ 。当然，实际上 G_q 就是 $S(c,q)q!$ ，其中 S 表示第二类斯特林数。

G 数组可以在 $O(m^2)$ 时间求出。

令 T_i 表示染色过的球与被选择的球总数为 i 的方案数，由于选择的球的编号一定小于染色的球编号，因此方案一一对应。容易发现 $T_i = \sum_{j=0}^{i}G_jH_{i-j}$ 。 T 数组可以在 $O(m^2)$ 时间求出。

则 $ans = \sum_{i=0}^{3m}\binom{n}{i}T_i$ 。

总复杂度 $O(m^2)$ 。

期望得分10分。

2.4.4 $F(x) = \sum_{i=0}^{m} a_i x^i$

注意到当 $F(x) = x^c$ 时， $G_q = \sum_{i=0}^{q}\binom{q}{i}(-1)^{q-i}i^c$

那么求和之后，现在的 $G_q = \sum_{i=0}^{q}\binom{q}{i}(-1)^{q-i}\sum_{j=0}^{m}a_j i^j = \sum_{i=0}^{q}\binom{q}{i}(-1)^{q-i}F(i)$

其余部分和之前相同。

总复杂度 $O(m^2)$ 。

期望得分70分。

2.5 算法5

在 n 很大的时候可能还有别的处理方法。如果编号最小位置 A 的贡献是关于 A 的多项式，将其求前缀和，得到的也会是一个多项式。那答案就是将 $n-1$ 代入得到的值。

考虑算法3中的计算， $\binom{n-A}{m}^2$ ，是关于 A 的 $2m$ 次多项式。而它要乘上的贡献， $F(x)$ 是一个 m 次多项式。因此最终每个位置的贡献就是一个 $3m$ 次多项式，其前缀和是一个 $3m+1$ 次多项式。我们取 $0\cdots 3m+1$ 算得其点值，使用拉格朗日插值就能线性得到该多项式在 $n-1$ 处的点值。

复杂度 $O(mlogm)$ ，常数较大。

期望得分80-100分。

2.6 算法6

在算法4的基础上进行优化。

对 G 数组的计算推导后可以得到, $\frac{G(q)}{q!} = \sum_{i=0}^{q} \frac{(-1)^{q-i}}{(q-i)!} \times \frac{F(i)}{i!}$ 。可以用NTT求出 G 数组。同样的道理用NTT求出 T 数组。

总复杂度为 $O(mlogm)$ ，常数较小。

期望得分100分。

3 总结

本题题意简洁，需要选手对于贡献多项式有深刻的理解与充分的性质考察，熟练的掌握二项式反演或斯特林数的知识，以及多项式的技巧。在考察选手基本功的同时，也考验选手观察部分分分布、从中寻找思路并总结的能力。同时本题两条思路都有优化的空间，能够获得高分，提高了题目的可做性。在实际互测中共有3位选手获得满分，7位选手获得70分及以上的分数，分数分布均匀，有一定的区分度，总体来说，是一道NOI难度的题。

4 鸣谢

感谢中国计算机学会提供学习和交流的平台。

感谢李建老师以及父母多年来的关心和指导。

感谢杭州二中的同学为本文验稿。

参考文献

[1] Graham, Ronald L.; Knuth, Donald E.; Patashnik, Oren (February 1994). Concrete Mathematics - A foundation for computer science (2nd ed.). Reading, MA, USA: Addison-Wesley Professional.

[2] 彭雨翔, Fast Fourier Transform.

[3] 刘汝佳, 黄亮, 《算法艺术与信息学竞赛》, 清华大学出版社。

[4] 刘汝佳, 《算法竞赛入门经典》, 清华大学出版社。

一些特殊的数论函数求和问题

安徽师范大学附属中学 朱震霆

摘 要

数论函数求和是信息学竞赛中较为重要的一类问题。本文主要分为三个部分：第一部分从埃式筛法的角度介绍了一种积性函数求和方法；第二部分探讨了素数求和问题；第三部分研究了一种特殊的数论函数的求和与拟合平面曲线之间的联系。

引言

在信息学竞赛中，我们常常会遇到对定义域为正整数的函数，也就是数论函数求和的问题。在研究一些特殊的数论函数求和问题时，我们往往会发现，解决这些问题的方法虽然特殊，但具有很大的启发意义。其中一些问题的解决办法还可以扩展到更多更普遍的问题，或是能够给许多问题提供一种新的思路。本文主要对其中的三个问题进行了探讨。

1 基础知识

定义 1.1. 定义域为正整数集，陪域为复数域的函数被称为数论函数。

定义 1.2. 设 $f(x)$ 是一个数论函数，若对任意一对互质的正整数 a, b，有 $f(ab) = f(a)f(b)$，则称 $f(x)$ 是积性函数。

引理 1.3. 对于给定的正整数 n 和所有正整数 m，$\left\lfloor \frac{n}{m} \right\rfloor$ 的取值只有 $O(\sqrt{n})$ 种。

证明. 当 $m \le \sqrt{n}$ 时，满足条件的 m 只有 $O(\sqrt{n})$ 种。当 $m > \sqrt{n}$ 时，$0 \le \left\lfloor \frac{n}{m} \right\rfloor < \sqrt{n}$，于是满足条件的 $\left\lfloor \frac{n}{m} \right\rfloor$ 也只有 $O(\sqrt{n})$ 种，于是得证。 □

定理 1.4 (素数定理)**.** 令 $\pi(x)$ 表示不超过 x 的素数个数，则 $\pi(x) = O(\frac{x}{\ln x})$ [25]。

[25] 关于素数定理的证明，可查阅参考文献 1 和其他相关资料

2 积性函数求和

作为一种特殊的数论函数，积性函数求和问题在信息学竞赛中更为常见。在 2016 年的国家集训队论文中，任之洲同学对该问题进行了深入探讨，并得出了一个适用范围较为广泛的积性函数求和方法，这种算法也在 OI 界中被称为"洲阁筛法"，实际上它就是筛素数时常用的埃式筛法的一种扩展。

但洲阁筛法仍然存在它的不足之处：一是在其从 $O(\frac{n}{\log n})$ 优化至 $O(\frac{n^{\frac{3}{4}}}{\log n})$ 的过程中，很多部分较难理解，初学者很难上手；二是常数因子较大，代码实现较为复杂。但扩展埃筛并不只有这一种实现方法。实际上，存在一种实现难度更低，同时在较小的数据范围下表现更好的方法。它和洲阁筛的思想类似，但理解起来更为容易。因为该算法在 Min_25 使用后才开始普及，因此也被称为 Min_25 筛法。

本文我们讨论的为后一种筛法，即 Min_25 筛法。

2.1 引入问题

定义 $\sigma_0(n)$ 为 n 的正约数个数，求 $S(n,k)=\sum_{i=1}^{n}\sigma_0(i^k)$ ， $n,k\le 10^{10}$ 。

2.2 分析

令 $f(x)=\sigma_0(x^k)$ ，容易发现 $f(x)$ 是积性函数，但找到更好的性质却并不容易，因此通常的想法是使用洲阁筛处理。

我们考虑一个稍微暴力一点的方法，不妨假设我们要求的积性函数为 f ，那么：

$$\sum_{i=1}^{n}f(i)=1+\sum_{\substack{2\le p\le n\\ p\text{是质数}}}\ \sum_{\substack{2\le x\le n\\ x\text{的最小质因子为}p}}f(x)=1+\sum_{\substack{2\le p^e\le n,e\ge 1\\ p\text{是质数}}}f(p^e)\left(1+\sum_{\substack{2\le x\le\left\lfloor\frac{n}{p^e}\right\rfloor\\ x\text{不含}\le p\text{的质因子}}}f(x)\right)$$

实际上也就是枚举 i 的最小质因子。

而我们注意到，对于一个合数 n 来说，其最小质因子必然不会超过 $\sqrt{n}$ ，因此：

$$\sum_{\substack{2\le p^e\le n,e\ge 1\\ p\text{是质数}}}f(p^e)\left(1+\sum_{\substack{2\le x\le\left\lfloor\frac{n}{p^e}\right\rfloor\\ x\text{不含}\le p\text{的质因子}}}f(x)\right)=\sum_{\substack{2\le p\le\sqrt{n},2\le p^e\le n,e\ge 1\\ p\text{是质数}}}f(p^e)\left(1+\sum_{\substack{2\le x\le\left\lfloor\frac{n}{p^e}\right\rfloor\\ x\text{不含}\le p\text{的质因子}}}f(x)\right)+\sum_{\substack{\sqrt{n}<p\le n\\ p\text{是质数}}}f(p)$$

令

$$g_{n,m}=\sum_{\substack{2\le x\le n\\ x\text{不含}\le m\text{的质因子}}}f(x)$$

$$h_n=\sum_{\substack{2\le p\le n\\ p\text{是质数}}}f(p)$$

我们可以写出递推式：

$$\begin{aligned} g_{n,m} &= \sum_{\substack{2\le x\le n \\ x\text{不含}\le m\text{的质因子}}} f(x) \\ &= \sum_{\substack{m<p\le\sqrt{n},p^e\le n,e\ge 1 \\ p\text{是质数}}} f(p^e)\left(1+\sum_{\substack{2\le x\le\left\lfloor\frac{n}{p^e}\right\rfloor \\ x\text{不含}\le p\text{的质因子}}} f(x)\right)+\sum_{\substack{m<p\le n \\ p\text{是质数}}} f(p) \\ &= \sum_{\substack{m<p\le\sqrt{n},p^e\le n,e\ge 1 \\ p\text{是质数}}} f(p^e)\left(1+g_{\left\lfloor\frac{n}{p^e}\right\rfloor,p}\right)+h_n-h_m \end{aligned}$$

而我们求解的答案就是 $g_{n,0}$ 。

于是若我们已经对所有 i 求出了 h_i ，那么我们有两种选择：直接使用该式计算；或是从大到小枚举 m 来计算所有的 g 。其中对于后一种做法来说，只有当 $n>m^2$ 时转移的值不是 0 ，于是直接按照转移式后缀和优化即可[26]。

下面考虑如何求 h 。

容易发现有用的 h_i 一定满足存在正整数 m ，使得 $\left\lfloor\frac{n}{m}\right\rfloor=i$ ，由引理 1.3 可知这样的 i 共有 $O(\sqrt{n})$ 个。而一般来说， $f(p)$ 是一个 p 的低次多项式，可以表示为 $f(p)=\sum a_t p^{b_t}$ ，那么我们只要对每个 p^{b_t} 和 i ，让 h_i 累加上所有不超过 i 的素数的 b_t 次之和的 a_t 倍即可。

于是我们的问题即为：给定 n 和 s ，对所有的 $i=\left\lfloor\frac{n}{m}\right\rfloor$ ，求

$$h_i=\sum_{\substack{p\le i \\ p\text{是质数}}} p^s$$

我们可以从埃式筛法的角度理解：埃式筛法的过程是每次枚举一个 p ，筛去所有不小于 p^2 的 p 的倍数。于是可以令 $h'_{i,j}$ 表示埃式筛法枚举了前 i 个素数后，不超过 j 的所有还剩下的数的 s 次方之和。即

$$h'_{i,j}=\sum_{\substack{1\le p\le j \\ p\text{是}1,\text{质数},\text{或没有}\le p\text{的质因子}}} p^s$$

考虑如何转移，不妨假设第 i 个质数为 p_i ，那么考虑埃式筛法的过程，对于 $j\ge p_i^2$ 我们有：

$$h'_{i,j}=h'_{i-1,j}-p_i^s\left(h'_{i-1,\left\lfloor\frac{j}{p_i}\right\rfloor}-h'_{i-1,p_i-1}\right)$$

在其他情况下， $h_{i,j}$ 不变。

于是我们只要暴力实现即可。实际上，后半部分的实现和洲阁筛中的第一部分完全相同。

[26]从大到小枚举 m 的做法即为任之洲论文中的“另一种做法”

2.3 时间复杂度计算

对于后一部分的计算，复杂度证明和洲阁筛完全相同。考虑每个 $\left\lfloor \frac{n}{m} \right\rfloor = i$ 只会在枚举不超过 $\sqrt{i}$ 的素数时产生贡献。由定理 1.4，复杂度可估计为

$$\begin{aligned}
&\sum_{i=1}^{\sqrt{n}} O\left(\frac{\sqrt{i}}{\log\sqrt{i}}\right) + \sum_{i=1}^{\sqrt{n}} O\left(\frac{\sqrt{\frac{n}{i}}}{\log\sqrt{\frac{n}{i}}}\right) \\
=&O\left(\int_1^{\sqrt{n}} \frac{\sqrt{\frac{n}{x}}\mathrm{d}x}{\log\sqrt{\frac{n}{x}}}\right) \\
=&O\left(\frac{n^{\frac{3}{4}}}{\log n}\right)
\end{aligned}$$

下面考虑分析前一部分计算 $g_{n,0}$ 的时间复杂度，为了便于后面的描述，我们先给出一个定义。

定义 2.1. 对正整数 i，我们定义 big_i 表示 i 的最大素因子， $small_i$ 表示 i 的最小素因子，cnt_i 表示 i 的可重素因子数目。其中当 $i = 1$ 时， $big_i = small_i = \infty, cnt_i = 0$ 。

如果利用 g 的转移式进行 DP，空间复杂度 $O(\sqrt{n})$，时间复杂度证明和后一部分计算类似，也是 $O\left(\frac{n^{\frac{3}{4}}}{\log n}\right)$。

如果直接按照原式暴力计算会怎样？在较小数据的测试情况中，该算法表现得更像是比 $O\left(\frac{n^{\frac{3}{4}}}{\log n}\right)$ 更优秀的算法，于是我们尝试分析该算法的时间复杂度。

考虑我们之前计算 $g_{n,m}$ 时递归的过程，那么我们的时间复杂度就是进入递归的次数，也就是 $h_n - h_m$ 被累加进答案的次数。

对于一次递归，不妨假设我们在这一次递归中，累加上的素数 p 的贡献，实际上是原来的 $F(l \times p)$ 的贡献，那么我们把这一次递归对复杂度的贡献计入 $l \times m$ 上。按照我们的转移， m 显然是 l 的最大质因子。由于我们每个函数值只会被统计一次贡献，因此每次的 $l \times m$ 显然是不同的；同时，每一个 $l \times m$ 只要不超过 n，就一定会出现。于是该算法实际上的复杂度就是 $i \times big_i \leq n$ 的 i 的个数，其中 big_i 为 i 的最大质因子。

引理 2.2. 对于给定的实常数 $0 < \alpha < 1$，令 $Q(n) = \{i \leq n : big_i \leq i^{\alpha}\}$，那么 $|Q(n)| \sim n\rho\left(\frac{1}{\alpha}\right)$，其中 ρ 为 Dickman function。

引理 2.3. 对于任意 $x > 1$， $\rho(x) > 0$ 随 x 增大而迅速衰减，且 $\rho(x) \approx x^{-x}$ 。

关于这两个引理可查阅参考文献 3 。

定理 2.4. 令 $M(n) = \{i : i \times big_i \leq n\}$，那么对任意 $0 < \alpha < 1$，有 $|M(n)| = \Omega(n^{\alpha})$

证明. 不妨取 $P = \left\{i \leq n^{\alpha} : big_i \leq i^{1-\alpha}\right\}$, 由引理 2.3 可知 $|Q(n^{\alpha})| = \Omega(n^{\alpha})$，而该集合 P 中每个元素 x 都满足 $x \times w_x \leq x^{2-\alpha} \leq n^{2\alpha-\alpha^2} \leq n$，且 $P \subseteq M(n)$，于是 $|M(n)| = \Omega(n^{\alpha})$ 。 □

定理 2.5. $|M(n)| = \Theta(n^{1-\epsilon})$

证明. 不妨取前 $\log\log n$ 个素数，令 p_i 表示第 i 个素数，那么最大素因子不超过 p_i 的数的个数不超过 $(\log n)^{\log\log n} \le O\left(\frac{n}{\log\log n}\right)$，而其他的满足 $i \times big_i \le n$ 的 i 都不超过 $\frac{n}{\log\log n}$，于是 $|M(n)| = O\left(\frac{n}{\log\log n}\right)$。

由定理 2.4， $|M(n)| = \Theta(n^{1-\epsilon})$。 □

该算法最终被证明是 $\Theta\left(n^{1-\epsilon}\right)$ 的，但这与我们实际的运行效果相去甚远。如何解释这一情况？

引理 2.6. $\sum_{p\ge n, p\text{是质数}} \frac{1}{p^2} = \Theta\left(\frac{1}{n\ln n}\right)$

证明.

$$
\begin{aligned}
\sum_{\substack{p>n\\ p\text{是质数}}} \frac{1}{p^2} &= \int_n^{\infty} \frac{\mathrm{d}\pi(x)}{x^2} \\
&= \Theta\left(\left.\frac{\pi(x)}{x^2}\right|_n^{\infty} - \int_n^{\infty} \pi(x)\mathrm{d}\left(\frac{1}{x^2}\right)\right) \\
&= \Theta\left(\left.\frac{1}{x\ln x}\right|_n^{\infty} + 2\int_n^{\infty} \frac{\mathrm{d}x}{x^2\ln x}\right) \\
&= \Theta\left(-\frac{1}{n\ln n} + 2\int_n^{\infty} \frac{\mathrm{d}x}{x^2\ln x}\right) \\
&= \Theta\left(-\frac{1}{n\ln n} - 2\int_n^{\infty} \frac{\mathrm{d}\frac{1}{x}}{\ln\frac{1}{x}}\right) \\
&= \Theta\left(-\frac{1}{n\ln n} - 2\int_0^{\frac{1}{n}} \frac{\mathrm{d}y}{\ln y}\right) \\
&= \Theta\left(-\frac{1}{n\ln n} - 2\mathrm{li}\left(\frac{1}{n}\right)\right)
\end{aligned}
$$

其中 $\mathrm{li}(x)$ 为对数积分函数。

查阅资料可知 $\mathrm{li}(n) \sim -\frac{1}{n\ln n}$，于是原命题得证。 □

定理 2.7. 当 $big_i \ge n^{\frac{1}{4}}$ 时，满足条件的 i 至多有 $O\left(\frac{n^{\frac{3}{4}}}{\log n}\right)$ 个。

证明. 当 $big_i \ge n^{\frac{1}{4}}$ 时，我们枚举每个 $p = big_i$，至多有 $\left\lfloor\frac{n}{p^2}\right\rfloor$ 个满足条件的数，于是这一部分的时间复杂度为 $O\left(\frac{n^{\frac{3}{4}}}{\log n}\right)$。 □

而打表可知，当 $n \le 10^{13}$ 时，对于一个素数 $p \le n^{\frac{1}{4}}$，满足 $big_i = p$ 的 i 的个数是 $\sqrt{n}$ 级别的，而一共只有 $\frac{n^{\frac{1}{4}}}{\log n}$ 个素数，于是运算次数的级别为 $\frac{n^{\frac{3}{4}}}{\log n}$ [27]。

[27] 该算法的代码实现: https://ideone.com/0l4aml

3 素数的 k 次幂前缀和

我们之前的算法涉及到了对所有 $\lfloor \frac{n}{i} \rfloor$ 求解素数的 k 次幂前缀和的问题，并给出了一个时间复杂度为 $O\left(\frac{n^{\frac{3}{4}}}{\log n}\right)$ 的解法。但有时候，我们所需要求的只是 $\sum_{i=1}^{n}[i是素数]i^k$ ，此时我们存在复杂度更优的做法，下文复杂度的计算中均不考虑计算 k 次幂所带来的影响。

定义 3.1. 对正整数 n ，定义 $\pi_k(n)$ 为素数的 k 次幂前缀和函数，即 $\pi_k(n) = \sum_{i=1}^{n}[i是素数]i^k$ 。特别地， π_0 即为素数计数函数，即 $\pi_0(n) = \pi(n)$ 。

定义 3.2. 对正整数 n 和正整数 a ，定义部分筛函数 $\phi_k(n,a) = \sum_{i=1}^{n}[big_i > p_a]i^k$ ，其中 p_a 表示第 a 个素数。当 $a = 0$ 时，有 $\phi_k(n,a) = \sum_{i=1}^{n} i^k$ 。

定义 3.3. 对正整数 s,n,a ，定义 $P_{s,k}(n,a) = \sum_{i=1}^{n}[big_i > p_a, cnt_i = s]i^k$ 。

定理 3.4 (Meissel Lehmer 方法)**.** $\phi_k(x,a) = \sum_{s\geq 0} P_{s,k}(x,a)$ 。当 $x^{\frac{1}{3}} \leq B = p_a \leq x^{\frac{1}{2}}$ 时，有 $\phi_k(x,a) = P_{0,k}(x,a) + P_{1,k}(x,a) + P_{2,k}(x,a) = 1 + \pi_k(x) - \pi_k(p_a) + P_{2,k}(x,a)$ 。

上式由定义显然可知，故证明略去。而根据该式移项我们可以得到： $\pi_k(x) = \phi_k(x,a) - P_{2,k}(x,a) + \pi_k(p_a) - 1$ 。

而我们现在要计算的是 $\pi_k(x)$ ， $\pi_k(p_a)$ 可以 $O(B)$ 筛法计算。于是只要考虑计算 $P_{2,k}(x,a)$ 和 $\phi_k(x,a)$ 即可。

3.1 $P_{2,k}(x,a)$ 的计算

根据定义有 $P_{2,k}(x,a) = \sum_{n\in\{n\leq x: n=pq, p_a<p\leq q\leq \frac{x}{p}, p和q是素数\}} n^k$ ，同时有 $B < q < \frac{x}{B}$ ，于是可以枚举 p ，计算所有可行的 q 的贡献，使用线性筛计算，时间复杂度 $O\left(\frac{x}{B}\right)$ 。

3.2 $\phi_k(x,a)$ 的计算

引理 3.5. $\phi_k(x,a) = \phi_k(x,a-1) - p_a^k\phi_k(\lfloor \frac{x}{p_a} \rfloor, a-1)$

利用容斥，该式是显然的。

定理 3.6. $\phi_k(x,a) = \sum_{big_n\leq p_a, small_n > p_b} \mu(n)n^k\phi_k(\lfloor \frac{x}{n} \rfloor, b)$

证明. 考虑我们转移的过程，不妨假设我们在转移的过程中依次除去的素数是 $p_1, p_2. \dots p_l$ 。不妨设 $p_1p_2p_3p_4\dots p_l = n$ ，那么由于数 n 可以被第 $b+1$ 个至第 a 个素数唯一分解，于是当 $big_n \leq p_a, small_n > p_b$ 时会产生 $\mu(n)n^k$ 的贡献，即为该式。 □

由该定理我们可知， $\phi_k(x,a) = \sum_{big_n\leq p_a} \mu(n)n^k \left\lfloor \frac{x}{n} \right\rfloor^k$ 。

首先我们考虑暴力统计 $n \leq p_a = B$ 的贡献，那么接下来需要统计的即为 $\sum_{n>B, big_n\leq p_a} \mu(n)n^k \left\lfloor \frac{x}{n} \right\rfloor^k$ 。

定理 3.7. $\sum_{n>B, big_n \le p_a} \mu(n) n^k \left\lfloor \frac{x}{n} \right\rfloor^k = \sum_{B<n\le B\times small_n} \mu(n) n^k \phi_k\left(\left\lfloor \frac{x}{n} \right\rfloor, \pi_0(small_n) - 1\right)$

证明. 利用定理 3.2 展开右式即可得到两者等价的信息。 □

实际上这相当于在我们 DFS 计算 $\phi_k(x,a)$ 过程中加入了 $n > B$ 时停止递归的剪枝。

不妨令 $p = small_n, m = \frac{n}{small_n}$ ，那么 $\mu(m) = -\mu(n)$ ，所求即为：

$$-\sum_{p\le B} \sum_{small_m>p, m\le B<mp} \mu(m) m^k p^k \phi_k\left(\left\lfloor \frac{x}{mp} \right\rfloor, \pi_0(p) - 1\right)$$

考虑计算该式。

3.2.1 $p \le \sqrt{B}$

对于这一部分，我们可以暴力计算出所有 ϕ 的值，并枚举 p 和 m 进行统计。

由引理 1.3，有效状态只有 $O\left(\frac{\sqrt{xB}}{\log B}\right)$ 种，于是这一部分的贡献可以 $O\left(\frac{\sqrt{xB}}{\log B}\right)$ 求得。

3.2.2 $p > x^{\frac{1}{3}}$

当 $p > \sqrt{B}$ 时，若 m 不为素数，则 $m > p^2 > B$ ，于是不存在可行的 m 。这提示我们只要考虑 m 为质数时的情况，于是求和式即为：

$$\sum_{p\le B} \sum_{q>p, q\le B<qp} (pq)^k \phi_k\left(\left\lfloor \frac{x}{pq} \right\rfloor, \pi_0(p) - 1\right)$$

其中 p 和 q 均为素数。

当 $p > x^{\frac{1}{3}}$ 时，$\left\lfloor \frac{x}{pq} \right\rfloor < x^{\frac{1}{3}} < p$ ，根据定义可知 $\phi_k\left(\left\lfloor \frac{x}{pq} \right\rfloor, \pi_0(p) - 1\right) = 1$ ，于是求和式即为：

$$\sum_{x^{\frac{1}{3}}<p\le B} \sum_{q>p, q\le B<qp} (pq)^k$$

由于素数的 k 次方和可以利用 3.1 节中预处理的信息进行统计，于是只要枚举 p 即可计算出所有的贡献。

于是这一部分的时间复杂度为 $O(B)$ 。

3.2.3 $\sqrt{B} < p \le x^{\frac{1}{3}}$

在这一部分中，我们仍然有 m 为素数的性质，且 $p^2 > B$ 是显然的。因此问题即求：

$$\sum_{\sqrt{B}<p\le x^{\frac{1}{3}}} \sum_{p<q\le B} (pq)^k \phi_k\left(\left\lfloor \frac{x}{pq} \right\rfloor, \pi_0(p) - 1\right)$$

我们利用定义计算 $\phi_k\left(\left\lfloor\frac{x}{pq}\right\rfloor,\pi_0(p)-1\right)$，即：

$$
\begin{aligned}
&\sum_{\sqrt{B}<p\le x^{\frac{1}{3}}}\sum_{p<q\le B}(pq)^k\phi_k\left(\left\lfloor\frac{x}{pq}\right\rfloor,\pi_0(p)-1\right)\\
=&\sum_{\sqrt{B}<p\le x^{\frac{1}{3}}}\sum_{p<q\le B}(pq)^k\sum_{pqr\le x,small_r\ge p}r^k\\
=&\sum_{r=1}^{\lfloor\frac{x}{r}\rfloor}r^k\sum_{\sqrt{B}<p\le x^{\frac{1}{3}}}\sum_{p<q\le B}(pq)^k[pqr\le x,small_r\ge p]\\
=&\sum_{r=1}^{\lfloor\frac{x}{r}\rfloor}r^k\sum_{\sqrt{B}<p\le\min(x^{\frac{1}{3}},small_r)}p^k\sum_{p<q\le\min(\lfloor\frac{x}{pr}\rfloor,B)}q^k
\end{aligned}
$$

我们可以枚举 p，同时用树状数组维护所有 $small_r\ge p$ 的 r。查询时，由引理 1.3，$\frac{x}{pr}$ 的取值只有 $O\left(\sqrt{\frac{x}{p}}\right)$ 种，于是我们只要在枚举 $\frac{x}{pr}$ 时，在树状数组上查询 $O\left(\sqrt{\frac{x}{p}}\right)$ 次即可。

定理 3.8. 对于任意 $0<\alpha<1$，n 以内恰有 $k\ge1$ 个素因子且最小素因子不小于 n^α 的数的个数是 $O\left(\frac{n}{\log^k n}\right)$ 的。

证明. 考虑归纳。

当 $k=1$ 时，即为素数定理。

当 $k\ge2$ 时，由归纳假设，当素因子个数为 $k-1$ 时成立，同时：

$$
\begin{aligned}
&O\left(\int_{n^\alpha}^{\sqrt{n}}\frac{\frac{n}{p}}{\log^{k-1}\frac{n}{p}}\mathrm{d}\pi(p)\right)\\
=&O\left(\frac{n}{\log^{k-1}\frac{n}{p}\log p}\Bigg|_{n^\alpha}^{\sqrt{n}}\right)-O\left(\int_{n^\alpha}^{\sqrt{n}}\frac{p}{\log p}\mathrm{d}\frac{\frac{n}{p}}{\log^{k-1}\frac{n}{p}}\right)\\
=&O\left(\frac{n}{\log^k n}\right)+O\left(\int_{n^\alpha}^{\sqrt{n}}\frac{n}{p\log p\log^{k-1}\frac{n}{p}}\mathrm{d}p\right)\\
=&O\left(\frac{n}{\log^k n}\right)
\end{aligned}
$$

因此恰有 $k\ge1$ 个素因子且最小素因子不小于 n^α 的数的个数是 $O\left(\frac{n}{\log^k n}\right)$ 的。 □

由定理 3.8，我们在树状数组上进行区间修改的次数不会超过 $O\left(\frac{x^{\frac{2}{3}}}{\log x}\right)$，而一次修改的时间复杂度为 $O(\log n)$，于是这一部分修改的时间复杂度为 $O\left(x^{\frac{2}{3}}\right)$。

考虑计算询问的次数：

定理 3.9. $\sum_{p\le m}\sqrt{\frac{x}{p}}=O(\frac{\sqrt{mx}}{\log m})$

证明.

$$\begin{aligned}&\sum_{p\le m}\sqrt{\frac{x}{p}}\\ =&\sqrt{x}\int_1^m\frac{\mathrm{d}\pi(p)}{\sqrt{p}}\\ =&\sqrt{x}\left(\frac{\pi(m)}{\sqrt{m}}-\int_1^m\pi(p)\mathrm{d}\frac{1}{\sqrt{p}}\right)\\ =&\sqrt{x}\left(\frac{\pi(m)}{\sqrt{m}}+\frac{1}{2}\int_1^m\frac{1}{\sqrt{p}\ln p}\mathrm{d}p\right)\\ =&O\left(\frac{\sqrt{mx}}{\log m}\right)\end{aligned}$$

□

由该定理，带入 $m=x^{\frac{1}{3}}$ 可知，询问的次数是 $O\left(\frac{x^{\frac{2}{3}}}{\log x}\right)$ 的，因此询问的时间复杂度也为 $O\left(x^{\frac{2}{3}}\right)$。

考虑总的时间复杂度即为 $O\left(x^{\frac{2}{3}}+\frac{x}{B}+\left(\frac{\sqrt{xB}}{\log B}\right)\right)$，取 $B=x^{\frac{1}{3}}\log^{\frac{2}{3}}x$，后半部分复杂度平衡为 $O\left(\left(\frac{x}{\log x}\right)^{\frac{2}{3}}\right)$，总的时间复杂度仍为 $O\left(x^{\frac{2}{3}}\right)$。

3.3 优化

之前做法的时间复杂度瓶颈并不在于分块部分，而是在于 $\sqrt{B}<p\le x^{\frac{1}{3}}$ 的部分，这启发我们进行优化。

仍然考虑我们要计算的式子，即计算

$$\begin{aligned}&\sum_{\sqrt{B}<p\le x^{\frac{1}{3}}}\sum_{p<q\le B}(pq)^k\phi_k\left(\left\lfloor\frac{x}{pq}\right\rfloor,\pi_0(p)-1\right)\\ =&\sum_{r=1}^{\lfloor\frac{x}{r}\rfloor}r^k\sum_{\sqrt{B}<p\le\min(x^{\frac{1}{3}},small_r)}p^k\sum_{p<q\le\min(\lfloor\frac{x}{pr}\rfloor,B)}q^k\end{aligned}$$

我们初始化树状数组的时间复杂度为 $O\left(\frac{n}{B}\right)$，而我们之后在树状数组上对 r 进行修改，意味着 r 的最小素因子在 $\sqrt{B}$ 和 $x^{\frac{1}{3}}$。

当 r 是素数时，若 $r\ge x^{\frac{1}{3}}$，那么对于所有满足条件的 p 来说，r 都会产生贡献。因此我们可以枚举 p，再统计 r 是素数时的所有贡献即可。时间复杂度 $O\left(\frac{x^{\frac{2}{3}}}{\log x}\right)$。

当 r 是非素数时，由定理 3.8，修改的有效次数只有 $O\left(\frac{x^{\frac{2}{3}}}{\log^2 x}\right)$，不会影响整体的复杂度。而 r 是非素数同样意味着 $r > p^2$，而存在 q 的条件有 $\frac{x}{pr} > p$，即 $p^2 r \le x$，因此有 $p \le x^{\frac{1}{4}}$。

由定理 3.9，询问的次数为 $O\left(\frac{x^{\frac{5}{8}}}{\log x}\right)$，同样不会影响到整体的复杂度。

这样这一部分的时间复杂度就被优化到了 $O\left(\frac{x^{\frac{2}{3}}}{\log x}\right)$，总的时间复杂度即被优化到了 $O\left(\left(\frac{x}{\log x}\right)^{\frac{2}{3}}\right)$。

4 约数函数前缀和

4.1 引入问题

定义 $\sigma(i)$ 表示 i 的约数个数，给定正整数 $n < 2^{63}$，求 $\sum_{i=1}^{n} \sigma(i)$。[28]

4.2 朴素做法

对于这个问题，我们有一个非常显然的朴素做法：

$$\begin{aligned} &\sum_{i=1}^{n} \sigma(i) \\ =&\sum_{i=1}^{n} \sum_{d|n} 1 \\ =&\sum_{d=1}^{n} \left\lfloor \frac{n}{d} \right\rfloor \end{aligned}$$

容易发现该式就是统计 $x, y > 0$ 且 $xy \le n$ 的 (x, y) 对数，而这个曲线关于 $x = y$ 对称，于是有：

$$\begin{aligned} &\sum_{d=1}^{n} \left\lfloor \frac{n}{d} \right\rfloor \\ =&2 \sum_{d=1}^{\lfloor \sqrt{n} \rfloor} \left\lfloor \frac{n}{d} \right\rfloor - \left\lfloor \sqrt{n} \right\rfloor^2 \end{aligned}$$

于是只要枚举 d 就能计算出答案了，时间复杂度 $O(\sqrt{n})$。但本题中 n 可以达到 2^{63} 级别，因此该朴素算法会超时。

[28] spoj.com/problems/DIVCNT1/

4.3 Stern-Brocot Tree和Farey序列

在介绍我们的算法之前，我们先介绍一些我们后面算法会利用到的东西。

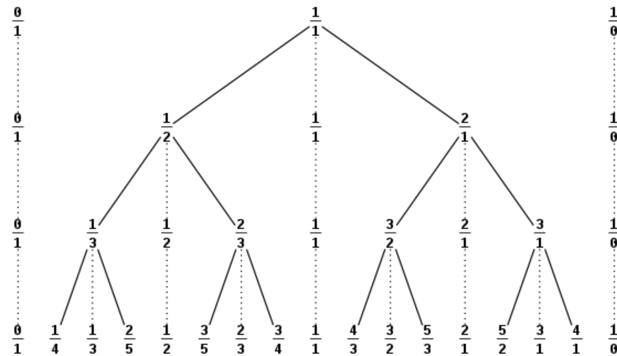
定义 4.1. 我们将分母不超过 n 的在 0 和 1 之间的所有最简分数由小至大排成一排，形成一个序列，那么我们称这个序列为 n 阶的 Farey 序列。若存在一个 Farey 序列使得某两个分数连续出现，那么称这两个数为 Farey neighbour。

引理 4.2. 任意一阶 Farey 序列中相邻连续两项 $\frac{a}{b} > \frac{c}{d}$ ，则有 $ad - bc = 1$ ，且反之也成立。

关于该引理的相关证明可查阅资料。

在本文中，若 $\frac{a}{b}$ 和 $\frac{c}{d}$ 是 Farey neighbour ，则称 $\frac{b}{a}$ 和 $\frac{d}{c}$ 也是 Farey neighbour ，显然在这样的情况下，该引理仍然成立。

定义 4.3 (Stern-Brocot Tree 的构造)**.** 我们构造一个完全二叉树，如图所示：



我们直接给出 Stern-Brocot Tree 的构造方法：不妨假设我们已经得到了 Stern-Brocot Tree 的前 i 层，那么对于第 i 层的一个代表元素 $\frac{x}{y}$ 的节点，我们考虑前 i 层中，比 $\frac{x}{y}$ 小的最大元素 $\frac{a}{b}$ (不存在则为 $\frac{0}{1}$)，和比 $\frac{x}{y}$ 大的最小元素 $\frac{c}{d}$ (不存在则为 $\frac{1}{0}$)，那么该节点的左孩子代表的元素为 $\frac{a+x}{b+y}$ ，右孩子代表的元素为 $\frac{c+x}{d+y}$ 。

Stern-Brocot Tree 具有以下性质：

性质 4.3.1. Stern-Brocot Tree 是一棵无穷大的完全二叉树。

性质 4.3.2. Stern-Brocot Tree 上的节点代表的数均为最简分数，且和有理数一一对应。

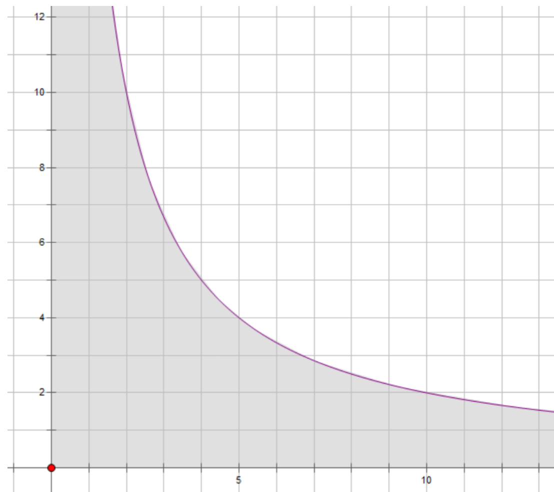
性质 4.3.3. Stern-Brocot Tree 具有搜索树性质，即左子树代表的所有数都小于根代表的数，且右子树的所有数都大于根代表的数。

性质 4.3.4. 仍然使用我们构造 Stern-Brocot Tree 中的定义，那么元素 $\frac{x}{y}$ 和元素 $\frac{a}{b}$ ，元素 $\frac{x}{y}$ 和元素 $\frac{c}{d}$ 均是 Farey neighbour。

以上内容可在参考文献 5 中查阅。

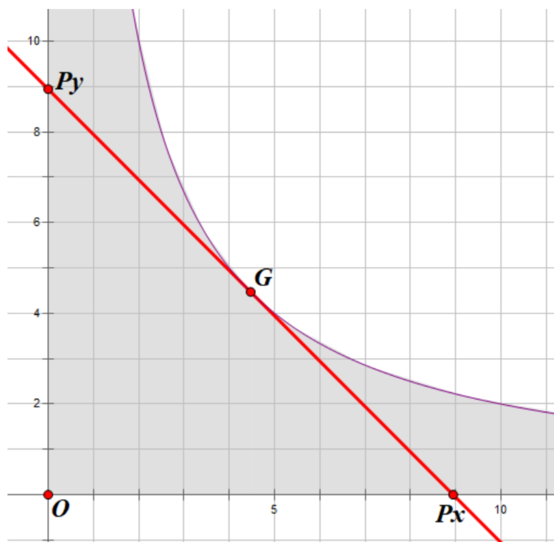
4.4 初步思路

直接统计显然不好解决，我们转化成统计曲线 $xy = n$ 下的整点数：



4.5 利用Stern-Brocot Tree切割曲线

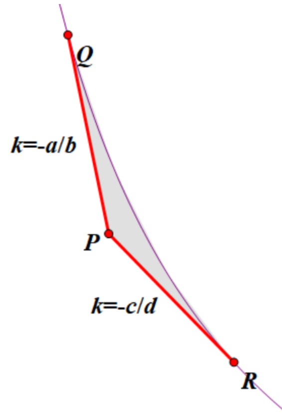
不妨考虑这样一种切割曲线的方法，我们现在要统计的是整个函数曲线下方的整点：



我们找到函数图像上，切线斜率为 -1 的点 G ，并作一条与该函数相切的直线，如图所示，交 x 轴和 y 轴于 P_x 和 P_y 两点。

统计在该切线下方的点数是容易的：不妨设 G 点的坐标为 (x_1, y_1) ， O 点的坐标为 $(0, 0)$ ，那么实际上满足条件的整点数即为满足 $x + y < x_1 + y_1$ 的整点数。一个很自然的想法就是统计完这一部分的点数后，迭代计算两边的整点数。

定义 4.4. 假设 x_0, y_0, a, b, c, d 为非负整数。当 $\frac{a}{b}$ 和 $\frac{c}{d}$ 是 Farey neighbour 时，我们可以根据下图定义 $S(x_0, y_0, a, b, c, d)$：图中 P 点坐标为 (x_0, y_0)，从 P 引出的两条斜率为 $-\frac{a}{b}$ 和 $-\frac{c}{d}$ 的直线与 $xy = n$ 分别交与 Q 点和 R 点，图中曲边三角形 PQR 内的整点个数即为 $S(x_0, y_0, a, b, c, d)$。当 $\frac{a}{b}$ 和 $\frac{c}{d}$ 不是 Farey neighbour 时，我们定义 $S(x_0, y_0, a, b, c, d) = 0$。



由于 y 轴可以被看做一条斜率为 $-\frac{1}{0}$ 的射线，因此我们所求的就是 $S(0, 0, 1, 0, 0, 1)$。

4.5.1 v-u 坐标系和 y-x 坐标系间的转换

考虑如何计算 $S(x_0, y_0, a, b, c, d)$。由于区域在 PR 和 PQ 上方，我们可以考虑以 P 为原点，PR 和 PQ 为轴，建立新的坐标系。

不妨将 PR 定为 u 轴，PQ 定为 v 轴，某个方向上的坐标加一，意味着沿着该方向走到下一个接触到的整点。根据我们之前的定义，有 a 和 b 互质，且 c 和 d 互质，因此有：

$$x = x_0 + ud - vb$$

$$y = y_0 - uc + va$$

同时，假如我们知道了整数 x 和 y，我们也能求出一组对应的整数 u 和 v：将两式分别乘 a 和 b 相加得：

$$ax + by = ax_0 + by_0 + (ad - bc)u$$

由性质 4.2 可知 $ad - bc = 1$，于是 u 也是整数，类似可知 v 也是整数。因此 y - x 坐标系中的点和 v - u 坐标系中的点一一对应。

我们考虑在 $v - u$ 坐标系中与原本的曲线 $xy = n$ 对应的曲线。不妨令

$$w_1 = ax_0 + by_0$$

$$w_2 = cx_0 + dy_0$$

可以解得

$$x_0 = dw_1 - bw_2$$
$$y_0 = aw_2 - cw_1$$

于是有：

$$x = d(u + w_1) - b(v + w_2)$$
$$y = a(v + w_2) - c(u + w_1)$$

于是 $xy = n$ 在 v - u 坐标系中即为：

$$(d(u + w_1) - b(v + w_2))(a(v + w_2) - c(u + w_1)) = n$$

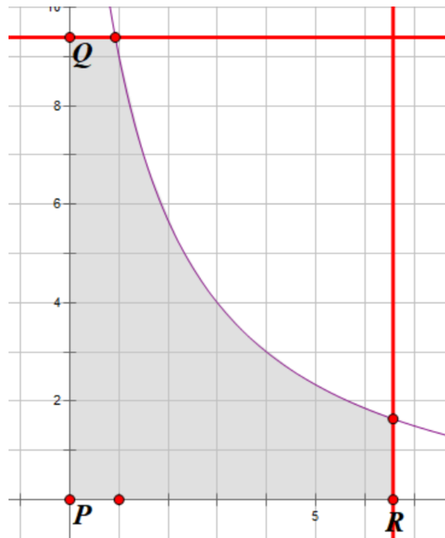
即曲线

$$(u + w_1)(v + w_2) - cd(u + w_1)^2 - ab(v + w_2)^2 = n$$

不妨设该曲线为 $H(u, v)$ 。

由于 $w_1, w_2, ab, cd \geq 0$ ，可知任意 $u > 0$ 对应唯一的 v ，同时任意 $v > 0$ 对应唯一的 u 。不妨设 $v = V(u), u = U(v)$ ，**则 U 和 V 在第一象限中均为凹函数，且性质与 $xy = n$ 类似**。关于该结论的证明可采用暴力求解析式或是采用微分的方法，此处略去。

不妨假设 Q 点的 v 轴坐标为 h ， R 点的 u 轴坐标为 w ，那么我们实际上统计的就是下图中阴影部分的整点数：



4.5.2 利用坐标系的转换统计整点数

回到我们最初的想法，我们希望在统计整点数目的过程中，能够利用之前所提到的，在切线斜率为 -1 的点切割函数曲线，进而分治向下递归的方法。我们考虑对上面转化后

的问题进行计算，当 $\min(w,h)$ 很小时我们可以暴力枚举 u 坐标或是 v 坐标计算，因此先不考虑这部分的情况。

首先我们找到切线斜率为 -1 的点 G ，注意到 v - u 坐标系中斜率为 -1 的直线实际上在 y - x 坐标系中的斜率为 $-\frac{a+c}{b+d}$ ，因此在 $y-x$ 坐标系中 G 存在，因此在 $v-u$ 坐标系中 G 也存在。找到点 G 后，作切线如下图所示。

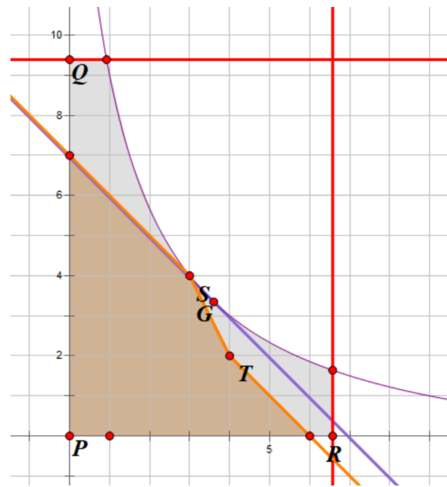
接下来我们令 G_u 表示点 G 的 u 轴坐标， G_v 表示点 G 的 v 轴坐标， G_x 表示点 G 的 x 轴坐标， G_y 表示点 G 的 y 轴坐标。我们不妨找到另外两个整点 S,T ，满足：

$$S_u \le G_u < S_u + 1 = T_u$$

$$S_v = \lfloor V(S_u) \rfloor$$

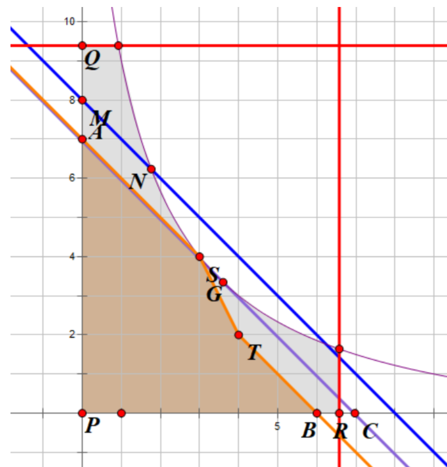
$$T_v = \lfloor V(T_u) \rfloor$$

由于 G 点的切线斜率为 -1 ，因此这两个点 S,T 一定存在，同时也可以保证 S 和 T 不相交。于是我们过 S,T 做 G 的切线的平行线，即：



我们可以先统计图中棕色部分的整点数目(包括边界)，这一部分的图形整体规则，可以方便处理。

由于 v - u 坐标系中斜率为 -1 的直线实际上在 y - x 坐标系中的斜率为 $-\frac{a+c}{b+d}$ ，而 Stern-Brocot Tree 的性质保证了 $\frac{a+c}{b+d}$ 同时和 $\frac{a}{c}, \frac{b}{d}$ 是 Farey neighbour。因此对于其他部分的整点，我们是可以迭代计算的。



我们连接 PG 后，分别对 PG 上方和右侧的阴影区域统计仍未被统计的整点数目。下面我们只考虑上方区域。

令 A 是 v 轴和过 S 所作的斜率为 -1 的线的交点， M 位于将 A 向上平移一格的位置。

定理 4.5. 过 M 作一条斜率为 -1 的直线交曲线于 N， PG 上方未统计的阴影区域内的整点总数，就是 MN 和 PG 上方(含边界)阴影区域内的整点总数，即 $S(B_x, B_y, a, b, a+c, b+d)$ 。

证明. 容易发现 N 在 PG 的上方，否则意味着 $S_v \neq \lfloor V(S_u) \rfloor$，于是 MN 和 PG 上方的阴影部分中，每个整点都未被统计。

同时，由于 $S_u < G_u < S_u + 1 = T_u$ 同时在 MN 下方，且在 PG 上方的每个点一定都在棕色区域内，于是该结论得证。 □

右侧也可以类似进行处理，不妨设 C 是 B 向右平移一格的位置，那么这一部分的整点数即为 $S(C_x, C_y, a+c, b+d, c, d)$ [29]。

4.6 时间复杂度证明

考虑我们整个算法的流程分为两个部分：前 $O(n^{\frac{1}{3}})$ 列的枚举和后 $O(n^{\frac{1}{2}})$ 列的计算。前面的枚举告诉我们该算法的时间复杂度至少是 $O(n^{\frac{1}{3}})$ 的，接下来我们考虑后面计算的复杂度。

由于在之前的迭代过程中， $w < 1$ 或 $h < 1$ 时我们将直接返回答案，我们只需要计算那些不会直接返回答案的迭代次数即可。如果一次迭代 $S(x_0, y_0, a, b, c, d)$ 不会被直接返回，意味着 $w \geq 1$ 且 $h \geq 1$ ，也就意味着在 v - u 坐标系中同时存在 $(1,0)$ 和 $(0,1)$ 两点。而根据我们之前的公式， $x = d(u + w_1) - b(v + w_2)$ 可以知道，当前在 v - u 坐标系内的 x 轴坐标最大的点和 x 轴坐标最小的点的 x 轴坐标差至少为 $b + d$ ，而这个差显然不超过切线斜率为 $-\frac{c}{d}$ 和 $-\frac{a}{b}$ 的两点的横坐标的差。

于是我们可以列出这样的条件式，即：

$$
\begin{aligned}
&O\left(\sum_{ad-bc=1}\left[\sqrt{\frac{n}{\frac{c}{d}}} - \sqrt{\frac{n}{\frac{a}{b}}} \geq b + d\right]\right)\\
=&O\left(\sum_{ad-bc=1}\left[\sqrt{\frac{d}{c}} - \sqrt{\frac{b}{a}} \geq \frac{b+d}{\sqrt{n}}\right]\right)\\
=&O\left(\sum_{ad-bc=1}\left[\frac{\sqrt{ad} - \sqrt{bc}}{\sqrt{ac}} \geq \frac{b+d}{\sqrt{n}}\right]\right)\\
=&O\left(\sum_{ad-bc=1}\left[\frac{\sqrt{bc+1} - \sqrt{bc}}{\sqrt{ac}} \geq \frac{b+d}{\sqrt{n}}\right]\right)
\end{aligned}
$$

引理 4.6. $\sqrt{w+1} - \sqrt{w} = O\left(\frac{1}{\sqrt{w}}\right)$

证明. 令 $f(w) = \sqrt{w+1} - \sqrt{w}$ ，在 $w = \infty$ 处泰勒展开即可得出该结论。 □

由该引理可知 $\sqrt{bc+1} - \sqrt{bc} = O\left(\frac{1}{\sqrt{bc}}\right)$ ，于是：

[29] 在参考文献 6 中，有该算法的具体实现细节。

$$\begin{aligned}
&O\left(\sum_{ad-bc=1}\left[\frac{\sqrt{bc+1}-\sqrt{bc}}{\sqrt{ac}} \geq \frac{b+d}{\sqrt{n}}\right]\right)\\
=&O\left(\sum_{ad-bc=1}\left[\frac{1}{\sqrt{abc^2}} \geq \frac{b+d}{\sqrt{n}}\right]\right)\\
=&O\left(\sum_{ad-bc=1}\left[abc^2(b+d)^2 \leq n\right]\right)
\end{aligned}$$

枚举 $bc = x$，可得：

$$\begin{aligned}
&O\left(\sum_{ad-bc=1}\left[abc^2(b+d)^2 \leq n\right]\right)\\
=&O\left(\sum_{x}\sum_{b|x}\sum_{a|(x+1)}\left[ab\frac{x^2}{b^2}\left(b+\frac{x+1}{a}\right)^2 \leq n\right]\right)\\
=&O\left(\sum_{x}\sum_{b|x}\sum_{a|(x+1)}\left[a\frac{x^2}{b}\left(b^2+\frac{x^2}{a^2}+\frac{bx}{a}\right) \leq n\right]\right)\\
=&O\left(\sum_{x}\sum_{b|x}\sum_{a|(x+1)}\left[abx^2+\frac{x^4}{ab}+x^3 \leq n\right]\right)\\
=&O\left(\sum_{x=1}^{n^{\frac{1}{3}}}\sigma(x)\sigma(x+1)\right)
\end{aligned}$$

由 $xy \leq \frac{x^2+y^2}{2}$ 可得：

$$\begin{aligned}
&\sum_{x=1}^{n^{\frac{1}{3}}}\sigma(x)\sigma(x+1)\\
\leq&\sum_{x=1}^{n^{\frac{1}{3}}}\frac{\sigma^2(x)+\sigma^2(x+1)}{2}\\
=&O\left(\sum_{x=1}^{n^{\frac{1}{3}}}\sigma^2(x)\right)
\end{aligned}$$

定理 4.7. $\sum_{i=1}^{n}\sigma^2(i) = \Theta(n\log^3 n)$

该数列即 OEIS 上的 A061502 数列。

于是我们得到了这一部分时间复杂度不超过 $O(n^{\frac{1}{3}}\log^3 n)$ 的证明。

显然，我们没有必要计算整个图形下的整点数，而是可以统计一部分曲线下的整点数，如下图所示：

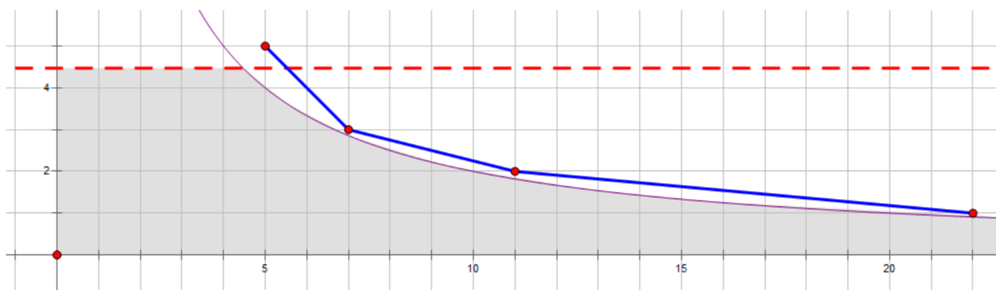
不过，我们之前计算的复杂度只考虑了 $\frac{a}{b},\frac{c}{d}$ 是正有理数的情况。我们没有计算 $a=1,b=0$ 或是 $c=0,d=1$ 时的情况，而这种情况出现的次数已经是 $O(\sqrt{n})$ 的了。

解决方法也很简单，不妨暴力计算前 $O(B)$ 列中的整点数目，而 $a=1,b=0$ 时一定有 $d=1$ 且 c 是整数，而 c 显然不会超过在 $x=B$ 时的切线斜率绝对值。于是这种情况的出现次数为 $O\left(\sqrt{\frac{n}{B}}\right)$。取 $B=n^{\frac{1}{3}}$ 时这一部分的情况总数是 $O\left(n^{\frac{1}{3}}\right)$。

但该证明所证出的复杂度上界并不紧，据说该算法的时间复杂度可以被证到 $O(n^{\frac{1}{3}}\log n)$，但笔者未能找到相关证明，故只给出一个上界，以证明该算法的时间复杂度是 $\tilde{O}\left(n^{\frac{1}{3}}\right)$ 的。

4.7 优化

实际上，我们之前反复切割曲线是没有必要的。我们可以直接利用 Stern-Brocot Tree 来拟合原本的曲线，如下图所示，我们只统计严格在拟合出来的折线下方的所有整点：



在具体的实现中，假设当前拟合到的点是 (x,y)，我们每次拟合一定是对所有不在曲线内的且在当前点右侧的点 (p,q) 选择一个 $\frac{q-y}{p-x}$ 最大且 p 最小的点，并加上这一条折线，以满足在折线下方的所有点都在曲线内的条件。我们只要根据上一次用来拟合的直线的斜率，

在 Stern-Brocot Tree 上二分找到这一次折线的斜率 $-\frac{a}{b}$ ，然后从 (x, y) 移动到 $(x + a, y - b)$ ，统计纵坐标在 $[y - b, y)$ 之间的整点数目即可[30]。

如此一来，这样用折线拟合的方案一定是唯一且存在的，因为点 $(1, n + 1)$ 一定在拟合出的折线上。

接下来考虑时间复杂度，我们可以将折线向下平移一格，容易发现折线的每一段到曲线的距离均不超过 1 ，并且折线与曲线之间没有任意一个点。因此我们可以将折线的每一段平移到与曲线相切的位置。同样按照上一个做法的想法考虑，我们每次相当于在 v-u 坐标系中取一条斜率为 -1 的直线，直到 $w, h \leq 1$ 。因此该算法与之前算法的时间复杂度相同。

4.8 扩展

显然，我们上面的利用 Stern-Brocot Tree 直接拟合曲线的优化算法并非只能解决计算双曲线 $xy = n$ 下方的整点数目的问题，即便是整点带权也不会产生什么影响。同时，在要求计算坐标系上其他具有凹凸性的曲线(如双曲线)与坐标轴围成的内部的整点的数目或权值和时，该算法也往往有着远远优于暴力算法的表现(例如圆内整点计数问题)。可以说，这个算法在解决这一类计数问题中的扩展前途广阔。

5 总结

本文主要介绍了三个特殊的数论函数求和问题，同时也有了三种解决问题的不同思路。我们解决第一个问题的思路是计算积性函数时的一种常见思想，即枚举质因子；解决第二个问题时我们通过分块降低了原本做法的复杂度；解决第三个问题时，我们采用数形结合和坐标系变换的方法得出了一种数点的方法，并通过 Stern-Brocot Tree 上二分得到了一种更优秀的实现。虽然这三个问题目前已经解决，但希望本文解决问题时用到的思想，能够给大家以启发，也希望有兴趣的同学能够继续研究下去，欢迎大家与我讨论。

感谢

感谢中国计算机学会提供交流和学习的平台。

感谢叶国平老师对我学习生活方面的帮助。

感谢刘承奥同学，唐靖哲前辈对本文的帮助。

感谢陈江伦同学，刘承奥同学，钟子谦同学，张晴川前辈，吴航前辈为本文审稿。

感谢我的父母对我的理解和关心。

[30]该算法的代码实现：https://ideone.com/PagVPQ

参考文献

[1] 潘承彪, 《从切比雪夫到爱尔特希: 素数定理的初等证明》。

[2] 任之洲, 《积性函数求和的几种方法》, 国家集训队2016论文集。

[3] http://mathworld.wolfram.com/DickmanFunction.html

[4] 张博航, https://zhuanlan.zhihu.com/p/33544708

[5] https://en.wikipedia.org/wiki/Stern-Brocot_tree, Wikipedia

[6] Richard Sladkey, A Successive Approximation Algorithm for Computing the Divisor Summatory Function(draft)

浅谈DFT在信息学竞赛中的应用

西北工业大学附属中学 刘承奥

摘 要

本文讨论离散傅里叶变换在信息学竞赛中的应用。围绕集合到环的映射这一研究对象，本文先从多项式角度引入了离散傅里叶变换这一工具，并讨论了扩展——在无零因子交换环和有限交换半群上的情况，最后进一步地利用这一工具优化了一般卷积的计算并得到了效率优于朴素算法的方法。

1 概述

1.1 引言

离散傅里叶变换（Discrete Fourier transform,DFT）是一种相当重要和具有许多应用的代数方法。在信息学竞赛中，其可以用于数学演算，或用于快速计算卷积以及一些线性变换，如多项式乘法等。

近年来在竞赛界，各种基于DFT的线性代数算法如雨后春笋般流行起来。DFT可以导出相应的快速卷积算法；至迟到2014年已引入了基于FFT和牛顿迭代的快速多项式初等函数算法 [5]；此后许多算法如Berlekamp-Massey也被引入。这些线性代数算法给竞赛内容的发展提供了极大的便利。

然而，这些形形色色的应用都离不开一种基本运算——DFT。因而，本文将回归DFT本身，探讨它在竞赛中的应用及扩展。

第1节介绍了总的模型、定义；

第2节从多项式角度引入DFT；

第3节探讨DFT的复合以及在下标为幂集时的扩展；

第4节继续扩展，讨论有限交换半群和无零因子交换环上的DFT；

第5节介绍计算DFT的算法；

第6节对更一般的卷积的高效算法做了少许讨论。

1.2 引入

在计算中，我们经常遇到这样的问题：序列之间对应相乘，再按下标的某种运算累加。在下标运算为加减法的时候，我们一般称之为卷积，但更多的情况下，下标并不是加法，比如二进制位运算等。要扩展我们的结论，就要把这些对象的共性——乘积和累加提取出来加以概括，得到卷积更抽象的概念。

我们提到了“下标”和“值”这两个概念。为了更一般和清晰地描述它们，我们把一个“序列”视作下标到值的映射。

1.3 定义和模型

一开始，我们给定定义在集合 R 上的伪环[31] $(R, +, \cdot)$ （也用 R 表示）和定义在集合 G 上的原群[32] $(G, \circ)$ （也用 G 表示）。

设 R 的加法幺元为 1_R ，在不产生歧义时，我们用整数 z 表示 z 个 1_R 的和。

我们有如下定义：

定义 1.1. 记 G 到 R 的**映射**的集合为 R^G 。称 G 是 R^G 的**定义域**， R 是 R^G 的**值域**。

定义 1.2 (加法)**.** $+$ ：对于 $f : G \to R$ 和 $g : G \to R$ ，定义 $f + g$ 为一个映射 $h : G \to R$ ，满足 $\forall x \in G, h(x) = f(x) + g(x)$ 。

定义 1.3 (点积)**.** $\cdot$ ：对于 $f : G \to R$ 和 $g : G \to R$ ，定义 $f \cdot g$ 为一个映射 $h : G \to R$ ，满足 $\forall x \in G, h(x) = f(x) \cdot g(x)$ 。

定义 1.4 (卷积)**.** $*$ ：对于 $f : G \to R$ 和 $g : G \to R$ ，定义 $f * g$ 为一个映射 $h : G \to R$ ，满足 $\forall x \in G, h(x) = \sum_{y,z \in G, y \circ z = x} f(y) \cdot g(z)$ 。[33]

上述定义中的符号在上下文未指明 G 时，用下标记为 $+_G, \cdot_G, *_G$ 。

引理 1.5 (零元)**.** $f : G \to R, f(x) = 0$ 是 R^G 关于 $+$ 的幺元和关于 $\cdot$ 及 $*$ 的零元，记作 $\mathbf{0}$ 。

引理 1.6 (点积幺元)**.** 如果 R 是环，那么 $f : G \to R, f(x) = 1$ 是 R^G 关于 $\cdot$ 的幺元，记作 $\mathbf{1}$ 。

引理 1.7 (卷积幺元)**.** 如果 R 是环且 G 含有幺元 e （即， $\forall a \in G, a \circ e = e \circ a = a$ ），那么：
$f : G \to R, f(x) = \begin{cases} 1 & x = e \\ 0 & \text{otherwise} \end{cases}$ 是 R^G 关于 $*$ 的幺元，记作 $\boldsymbol{e}$ 。

引理 1.8 (卷积结合律)**.** 如果 G 是半群，那么 R^G 的 $*$ 满足结合律，即 $\forall A, B, C \in R^G, (A * B) * C = A * (B * C)$ 。

[31] 即不需要存在乘法幺元的环，rng。

[32] 即定义了一个封闭二元运算的代数系统，参见https://en.wikipedia.org/wiki/Magma_(algebra)

[33] 此处，对于无穷情况， $h(x)$ 可能无定义。

引理 1.9 (卷积交换律)**.** 如果 G 满足交换律，且 R 是交换环，那么 R^G 的 $*$ 满足交换律，即 $\forall A, B \in R^G, A * B = B * A$ 。

引理 1.10 (卷积对加法的分配律)**.** R^G 的 $*$ 对 $+$ 满足分配律，即 $\forall A, B, C \in R^G, A * (B + C) = A * B + A * C, (B + C) * A = B * A + C * A$ 。

我们有如下结论：

引理 1.11. $(R^G, +, \cdot)$ 是一个伪环，或者当 R 是环时是环。

引理 1.12. 如果 $\sum$ 是封闭的，那么 $(R^G, +, *)$ 是一个伪环，或者当 R 是环且 G 是幺半群时是环。

我们设：

记号 1.13 (点积（伪）环)**.** 记 $R^G_\cdot$ 表示 $(R^G, +, \cdot)$ 。

记号 1.14 (卷积（伪）环)**.** 记 R^G_* 表示 $(R^G, +, *)$ ，在不致引起混淆的情况下，也用 R^G 表示。

2 形式幂级数与经典DFT

2.1 定义

本节中，我们令 $G = \mathbb{N}$ ， R^G 即为 R 上的形式幂级数 $R[[x]]$ 。[34]

在本节中，用 $A(x) = \sum_{i\in\mathbb{N}} a_i x^i \in R[[x]]$ 表示 R 上的一个形式幂级数。在不出现歧义的情况下，有时用数列 $\{a_i\}$ 表示形式幂级数 $\sum_{i\in\mathbb{N}} a_i x^i$ 。

2.2 多项式

我们首先讨论较简单的情况：域上的多项式。

本节中，我们要求 R 是复数域 $\mathbb{C}$ 。

这样，第1节的所有性质都成立。

2.2.1 多项式的系数和点值表示

卷积的形式较为复杂，我们尝试用简单形式表示。

我们知道，一个 n 次多项式可以由 $n+1$ 项系数或某 $n+1$ 个 x 及对应的 $f(x)$ 完全确定，我们把这样的 n 项数列 $\{a_i\}(i \in [0, n) \cap \mathbb{N})$ 视作$\mathbb{C}^{\mathbb{Z}_n}$ 中的元素。我们称；

[34]本文中 $\mathbb{N}$ 包含元素0。

定义 2.1 (系数表示)**.** 称数列 $\{a_i\} \in \mathbb{C}^{\mathbb{Z}_n}$ 是至多 $n-1$ 次多项式 $f(x) = \sum_{i=0}^{n-1} a_i x^i$ 的**系数表示**，记作 $[x]f$ 。

定义 2.2 (点值表示)**.** 对于元素两两不同的数列 $\{x_i\} \in \mathbb{C}^{\mathbb{Z}_n}$ ，称数列 $\{y_i : y_i = f(x_i)\} \in \mathbb{C}^{\mathbb{Z}_n}$ 是至多 $n-1$ 次多项式 $f(x)$ 的**点值表示**，记作 $\mathcal{F}_{\{x_i\}}(f)$ ，逆变换记作 $f = \mathcal{F}_{\{x_i\}}^{-1}(\{y_i\})$ 。

有：

引理 2.3 (拉格朗日插值)**.** 对于元素两两不同的数列 $\{x_i\}(i \in [0, n) \cap \mathbb{N})$ 和点值表示 $\{y_i\} = \mathcal{F}_{\{x_i\}}(f)$ ，唯一满足条件的 f 为：

$$f(x) = \sum_{i=0}^{n-1} y_i \prod_{j=0, j\neq i}^{n-1} \frac{x - x_j}{x_i - x_j}$$

另外，**点值的线性运算与多项式的线性运算等价**，也即： $\forall x, y \in \mathbb{C}, f, g \in \mathbb{C}[x], (f + g)(x) = f(x) + g(x), (yf)(x) = y \cdot f(x)$ 。

多项式的卷积是乘法，我们有 $(fg)(x) = f(x) \cdot g(x)$ 。但可以注意到点值的数量与多项式的次数不同。如何用点值表示卷积呢？有：

引理 2.4 (多项式余数定理)**.** $\forall a \in \mathbb{C}, f \in \mathbb{C}[x], f(a) = f \bmod (x - a)$ 。[35]

由此可得：

定理 2.5. $\forall \boldsymbol{x} = \{x_i\}, \boldsymbol{a}, \boldsymbol{b} \in \mathbb{C}^{\mathbb{Z}_n}$ ，且 x_i 两两不同，有 $\mathcal{F}_{\boldsymbol{x}}^{-1}(\boldsymbol{a} \cdot \boldsymbol{b}) = \mathcal{F}_{\boldsymbol{x}}^{-1}(\boldsymbol{a}) \mathcal{F}_{\boldsymbol{x}}^{-1}(\boldsymbol{b}) \bmod \prod_{i=0}^{n-1}(x - x_i)$

2.2.2 卷积

考察点值多项式。我们将其展开，可得：

$$fg \bmod \prod_{i=0}^{n-1}(x - x_i) = \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} ([x]f)_j ([x]g)_k (x^{j+k} \bmod \prod_{i=0}^{n-1}(x - x_i))$$

现在回到开始。我们希望利用点值乘法构造一个卷积。为了简便，首先考虑系数形成卷积的情况，也即要求 $\forall j \in \mathbb{N}, x^j \bmod \prod_{i=0}^{n-1}(x - x_i) = x^k, k \in \mathbb{N}$ 。

这也就意味着 $\prod_{i=0}^{n-1}(x - x_i)$ 只能有两项系数非 0 且分别为 ± 1 。当 $n > 1$ 时， $\prod_{i=0}^{n-1}(x - x_i)$ 至少有两项系数非 0 ，这只有两种解：

$$x^n - 1 \quad (1)$$
$$x^n - x \quad (2)$$

对于 (1) ，根即为 n 次单位根 $\{\omega_n^i : i \in \mathbb{Z}_n\}$ （以下记为 $\mathbb{T}_n$ ），有：

[35] 在此意义下，拉格朗日插值是中国剩余定理的特例。

$$[x]\mathcal{F}_{\mathbb{T}_n}^{-1}(\boldsymbol{a}\cdot\boldsymbol{b}) = [x]\mathcal{F}_{\mathbb{T}_n}^{-1}(\boldsymbol{a}) *_{\mathbb{Z}_n} [x]\mathcal{F}_{\mathbb{T}_n}^{-1}(\boldsymbol{b})$$

也即， $(\bmod\ \prod_{i=0}^{n-1}(x-x_i))$ 意义下的多项式乘法等价于系数列在 $\mathbb{Z}_n$ 下的卷积。

对于 (2)，根即为 $\{0\}\cup\mathbb{T}_{n-1}$，有：

$$[x]\mathcal{F}_{\mathbb{T}_n}^{-1}(\boldsymbol{a}\cdot\boldsymbol{b}) = [x]\mathcal{F}_{\mathbb{T}_n}^{-1}(\boldsymbol{a}) *_{\boldsymbol{Z'}} [x]\mathcal{F}_{\mathbb{T}_n}^{-1}(\boldsymbol{b})$$

其中， $\boldsymbol{Z'} = (\mathbb{Z}_n, \circ), \forall a, b \in \mathbb{Z}_n$

$$a \circ b = \begin{cases} a+b & a = 0 \wedge b = 0 \\ (a+b-1) \bmod (n-1) + 1 & a \neq 0 \vee b \neq 0 \end{cases}$$

这是一个由 $\mathbb{Z}_{n-1}$ 和平凡群[36]构成的系统，这与(1)很相似，且是后文内容的特例，我们不再专门讨论。

2.3 经典DFT

利用多项式的点值表示，我们得到了一种卷积和点积之间的转化方法。现在我们抛去多项式的限制，从第1节所述映射的角度来分析。我们实际上得到了一个变换：

定义 2.6 (经典DFT)**.**

$$\mathcal{F}: \mathbb{C}^{\mathbb{Z}_n} \to \mathbb{C}^{\mathbb{Z}_n}{}', f \to (f' : x \to \sum_{i\in\mathbb{Z}_n} \omega_n^{ix} f(i))$$

其中 ω_n 是一个 n 次本原单位根，如 $e^{\frac{2\pi i}{n}}$ 。

显然其是一个**线性变换**，另外还满足：

定理 2.7 (卷积定理)**.**

$$\mathcal{F}(f * g) = \mathcal{F}(f)\cdot\mathcal{F}(g)$$

这给了我们一个 $\mathbb{C}_*^{\mathbb{Z}_n}$ 到 $\mathbb{C}_\cdot^{\mathbb{Z}_n}$ 的同态。事实上，我们还可以发现：

定理 2.8 (经典IDFT)**.**

$$\mathcal{F}^{-1}: \mathbb{C}^{\mathbb{Z}_n}{}' \to \mathbb{C}^{\mathbb{Z}_n}, f \to (f' : x \to \frac{1}{n}\sum_{i\in\mathbb{Z}_n} \omega_n^{-ix} f(i))$$

即DFT是 $\mathbb{C}_*^{\mathbb{Z}_n}$ 到 $\mathbb{C}_\cdot^{\mathbb{Z}_n}$ 的**同构**映射。并且，可以证明：

定理 2.9. DFT是 $\mathbb{C}_*^{\mathbb{Z}_n}$ 到 $\mathbb{C}_\cdot^{\mathbb{Z}_n}$ 的唯一同构映射。

那么，我们在计算这两个环上的运算时，可以采用DFT进行简化和加速。2-FFT以及Bluestein's Algo.等算法可以以 $O(n\log n)$ 次单位根乘法和 $O(n\log n)$ 次加法的效率快速计算DFT。不过这些算法在竞赛中已相当普及，不再赘述。

[36]即仅由一个元素构成的群

2.4 DFT的条件

上节讨论的DFT适用于 $\mathbb{C}^{\mathbb{Z}_n}$ ，我们考虑对 R 进行扩展，有：

定理 2.10. 对于环 R ，如果 R 满足：

1. R 是无零因子环[37]；
2. R 有 n 次本原单位根 ω_n ，即，存在元素 ω_n 使得 $\omega_n^i (i \in \mathbb{Z}_n)$ 互不相等且 $\omega_n^n = 1$ ；
3. n 的乘法逆元在 R 中存在，即，存在元素 x 使得 $xn = nx = 1$ ；

那么：

$$\mathcal{F} : R^{\mathbb{Z}_n} \to R^{\mathbb{Z}_n}{}', f \to (f' : x \to \sum_{i \in \mathbb{Z}_n} \omega_n^{ix} f(i))$$

和

$$\mathcal{F}^{-1} : R^{\mathbb{Z}_n}{}' \to R^{\mathbb{Z}_n}, f \to (f' : x \to \frac{1}{n} \sum_{i \in \mathbb{Z}_n} \omega_n^{-ix} f(i))$$

是 $R_*^{\mathbb{Z}_n}$ 到 $R_\cdot^{\mathbb{Z}_n}$ 的**同构**映射。

证明过程与 $\mathbb{C}^{\mathbb{Z}_n}$ 上的DFT相同，此处略去。

不过需要注意的是，这是一个充分条件而不是充要条件。

信息学竞赛中最常见的满足上述条件的环有 $\mathbb{C}$ 和 $\mathbb{Z}_p$ （ $n|p-1$ ， p 表示质数），后者的DFT也称为数论变换（NTT）。

有些时候，可以通过扩域来得到合法的环，例如生成树求和[38]中使用的二次域。

为了方便下文，我们再补充一个定义：

定义 2.11. 如果 $\mathcal{F}$ 是 R_*^G 到 $R_\circ^G$ 的同构映射，称 $\mathcal{F}$ 是 R^G 上的**卷积性变换**。

3 简单推广

上节中，我们利用形式幂级数导出和引入了DFT方法。本节中将讨论一些简单推广。

---

[37]实际上，这个条件可以替换成一个适用范围更广的条件：求和引理 $\sum_{j=0}^{n-1} \omega_n^{ji} = [n|i]$ 。无零因子环蕴含此条件。求和引理可以适用于更多的环，例如满足条件的环上的方阵。

[38]https://loj.ac/problem/6271

3.1 复合变换

记号 3.1. 记 A^{B^C} 表示 $(A^B)^C$ 。

引理 3.2 (结合律)**.** $A^{B^C} = A^{(B^C)}$ 。

引理 3.3 (卷积性变换复合)**.** 如果 G, G' 是**半群**，且 $\mathcal{F}$ 是 R^G 上的卷积性变换，且 $\mathcal{F}'$ 是 $R^{G'}$ 上的卷积性变换，那么：

$$\mathcal{H} : R^{G^{G'}}_{*\,*} \to R^{G^{G'}}_{\cdot\,\cdot}, (x \to f(x) \to g(f(x))) \to (x \to \mathcal{F}'(f)(x) \to \mathcal{F}(g)(\mathcal{F}'(f)(x)))$$

是 $R^{G^{G'}}$ 上的卷积性变换。

这相当于，对第一次变换后得到的“点值”独立地再进行一次变换。

记号 3.4. 记上述 $\mathcal{H}$ 为 $\mathcal{F}' \circ \mathcal{F}$ ，称为 $\mathcal{F}'$ 和 $\mathcal{F}$ 的复合。

那么，引理3.3可以表述为：**值域相同的两个卷积性变换的复合是它们定义域的直积**[39]**到值域的卷积性变换**。

我们也有相应的逆定理：

引理 3.5 (卷积性变换复合逆定理)**.** 如果 $H = G \otimes G'$ ， H, G, G' 是半群，且 $\mathcal{H}$ 是 R^H 上的卷积性变换，那么：

$$\mathcal{F}(f(x)) = \mathcal{H}(f(x \circ e'_{G'}))$$

是 R^G 上的卷积性变换。其中 $e'_{G'}$ 是 G' 的一个幂等元，即取与另一个半群中某个幂等元运算结果在 $\mathcal{H}$ 中对应变换结果。同理 $\mathcal{F}'$ 是 $R^{G'}$ 上的卷积性变换。且 $\mathcal{H} = \mathcal{F}' \circ \mathcal{F}$ 。

即，**定义域能写成两个半群直积的卷积性变换一定可以写成这两个半群分别到值域的卷积性变换的复合**。

综上，可以认为**卷积性变换的复合等价于定义域的直积**。

将此应用于经典DFT即可得高维DFT。实际上高维DFT的定义域和值域与多元多项式等价。

3.2 集合幂级数

集合幂级数[40]即下标 G 是某个有限集的幂集。我们记全集为 U （ U 的幂集记为 2^U ），且 U 是一个有限集。那么 U 上的集合幂级数即为 R^{2^U} 。

---

[39] G 与 G' 的**直积**，记作 $G \otimes G'$ ，即在二者中各取一个元素 $a \in G, b \in G'$ 组成的所有二元组 (a, b) 组成的系统，运算时对应项分别在 G 和 G' 中运算，即 $(a, b) \circ (c, d) = (a \circ c, b \circ d)$ 。

[40]注意，数学界并没有这一公认用法，“集合幂级数”一词为竞赛中使用的方言，最早可能出现在吕凯风同学的《集合幂级数的性质及其快速算法》[6] 中。

可以注意到，对于满足元素独立性的集合运算——如交、并、对称差等—— 2^U 可以分解成关于每个元素运算的直积。

其中，对称差运算的 2^U 同构于 $\mathbb{Z}_2^{|U|}$ ，即二进制异或。利用上述高维DFT即可。

交和并每一维的变换（我们用向量 $\begin{bmatrix} f(0) \\ f(1) \end{bmatrix}$ 来表示 f ）分别是交：

$$\mathcal{F} : \boldsymbol{a} \to \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \boldsymbol{a}, \mathcal{F}^{-1} : \boldsymbol{a} \to \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \boldsymbol{a}$$

和并：

$$\mathcal{F} : \boldsymbol{a} \to \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \boldsymbol{a}, \mathcal{F}^{-1} : \boldsymbol{a} \to \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \boldsymbol{a}$$

在后文中我们将看到这是二阶情况下的两个解，在此不深入讨论。最后对每一维变换应用复合即可。

4 有限交换半群上的DFT

本节我们将着重讨论有关 G 的推广。

4.1 引入

上一节中我们已经讨论了DFT在 R 和复合上的推广，但这些推广的作用仍然不足。能否将DFT推广到更一般的 G 上呢？

鉴于讨论难度、作者水平以及计算方面的实际限制，本文不涉及 G 无限以及没有交换律的内容，仅讨论**有限交换半群**。

4.2 条件和准备

我们直接给出下面讨论的条件，在后文中，我们将会看到这些条件对于我们的讨论方法是必要的。

我们要求：

1. G 是有限交换半群

2. R 是无零因子非零[41]交换环

3. 讨论的卷积性变换 $\mathcal{F}$ 是线性映射

[41]即不是零环——只含有一个元素的环。

由于我们已经要求 $\mathcal{F}$ 是线性映射，仅考虑线性运算时已经是 R^G_* 到 $R^G_\cdot$ 的同构映射，故重点关注卷积定理：

$$\mathcal{F}(\boldsymbol{a} * \boldsymbol{b}) = \mathcal{F}(\boldsymbol{a}) \cdot \mathcal{F}(\boldsymbol{b})$$

既然 G 有限，不妨设 $|G| = n$，将其中元素分别标号为 $0, 1, \cdots, n-1$，且约定用列向量 $\begin{bmatrix} f(0) \\ f(1) \\ \vdots \\ f(n-1) \end{bmatrix}$ 表示一个 $f \in R^G$。

可以看到，R^G 是存在线性基 $\{f(x) = [x = i] : i \in G\}$ 的，这个列向量即为这组基下的坐标。既然存在基，线性映射必然可以用基下的矩阵表示，也即：

记号 4.1 (变换矩阵)**.** 记 $\boldsymbol{T} = T(\mathcal{F})$ 表示 $\mathcal{F}$ 在基 $\{f(x) = [x = i] : i \in G\}$ 下对应的变换矩阵，即 $\forall \boldsymbol{a} \in R^G, \mathcal{F}(\boldsymbol{a}) = \boldsymbol{T}\boldsymbol{a}$。

展开卷积定理式，可以注意到 $\boldsymbol{a}$ 的每一维（对应 $\boldsymbol{T}$ 的一行）实际上是独立的。考虑第 i 维有：

$$\begin{aligned} \sum_{j=0}^{n} \boldsymbol{T}_{i,j} \sum_{k,l,k\circ l=j} \boldsymbol{a}_k \boldsymbol{b}_l &= (\sum_{j=0}^{n} \boldsymbol{T}_{i,j} \boldsymbol{a}_j)(\sum_{j=0}^{n} \boldsymbol{T}_{i,j} \boldsymbol{b}_j) \\ \sum_{k,l} \boldsymbol{T}_{i,k\circ l} \boldsymbol{a}_k \boldsymbol{b}_l &= \sum_{k,l} \boldsymbol{T}_{i,k} \boldsymbol{a}_k \boldsymbol{T}_{i,l} \boldsymbol{b}_l \\ \sum_{k,l} \boldsymbol{T}_{i,k\circ l} \boldsymbol{a}_k \boldsymbol{b}_l &= \sum_{k,l} \boldsymbol{T}_{i,k} \boldsymbol{T}_{i,l} \boldsymbol{a}_k \boldsymbol{b}_l \end{aligned} \tag{15}$$

注意到 (1) 中用到了交换律。[42]

我们需要这是 R 上的恒等式。显然，如果两边对应系数相等，即 $\forall k, l, \boldsymbol{T}_{i,k\circ l} = \boldsymbol{T}_{i,k} \boldsymbol{T}_{i,l}$，恒等式一定成立。且 $\forall x, y \in G$ 依次令 $a_i = [x = i], b_j = [y = j]$，即得 $\forall k, l, \boldsymbol{T}_{i,k\circ l} = \boldsymbol{T}_{i,k} \boldsymbol{T}_{i,l}$，故这是恒等的充要条件。

设行向量 $\boldsymbol{x} = \{x_i\} = \boldsymbol{T}_i$，我们实际上得到了一个方程组:

$$\forall i, j, x_{i\circ j} = x_i x_j$$

也即 $\boldsymbol{x}$ 将 G 中的 $\circ$ 运算映射到 R 中的乘法上，或者说，**G 是 R 的乘法半群的子半群**。

为了让 $\mathcal{F}$（即 $\boldsymbol{T}$）可逆，我们至少需要方程组的 n 组线性无关的解。我们称 $\boldsymbol{x} = \boldsymbol{0}$ 为平凡解，不再考虑。

接下来的两节将讨论这一内容。

[42] 事实上，交换环的条件可以更弱一些，只要 $\boldsymbol{T}$ 中的元素对 R 中元素满足交换性，即 $\forall x \in R, i, j, \boldsymbol{T}_{i,j} x = x \boldsymbol{T}_{i,j}$，本节中的论证也都成立。这样的条件可以应用于满足条件的环上的方阵。

4.3 必要性

定义 4.2 (幂)**.** 称 $\underbrace{i \circ i \circ \cdots \circ i}_{j个}$ 为 i 的 j 次幂，记作 i^j 。

定义 4.3 (循环性)**.** 我们称原群 G 满足**循环性**，当且仅当 $\forall i \in G, \exists j \in \mathbb{N}^+$ 使得 $i^{j+1} = i$ 。

引理 4.4. G 满足循环性是 R^G 上存在卷积性变换存在的必要条件。

证. 假设对于某个 $i \in G$ ，不存在 $j \in \mathbb{N}^+$ 使得 $i^{j+1} = i$ ，又由于 G 有限，必然存在一些 $x, y \in \mathbb{N}^+$ 满足 $x \neq y, i^x = i^y$ ，我们考虑其中 x, y 的字典序最小的一对。

那么由假设 $y > x > 1, i^{x-1} \neq i^{y-1}$ 。另由结合律及交换律有 $\forall a \geq x, i^a = i^{x+((a-x) \bmod (y-x))}$ 。

那么， $i^{x-1} \circ i^{y-1} = i^{x+y-2} = i^{x-1} \circ i^{x-1} = i^{2x-2} = i^{y-1} \circ i^{y-1} = i^{2y-2}$ 。

由于 R 非零，有 $-1, 1 \neq 0$ ，那么考虑映射 $f(x) = \begin{cases} -1 & x = i^{x-1} \vee x = i^{y-1} \\ 0 & \text{otherwise} \end{cases}$ ，可以注意到 $f * f = \mathbf{0}$ ，那么$\mathcal{F}(f) = \mathbf{0} = \mathcal{F}(\mathbf{0})$ ，这和 $\mathcal{F}$ 是单射矛盾，证毕。

□

4.4 充分性及构造

下面我们将看到，循环性是DFT扩展到 G 上的关键条件。本节中我们在循环律的前提下讨论。

定义 4.5 (阶)**.** $\forall i \in G$ ，称 i 的正整数次幂的最小正周期为 i 的**阶**，记作 $\mathrm{ord}(i)$ 。定义 $i^0 = i^{\mathrm{ord}(i)}$ ， $\forall x \in \mathbb{Z}$ ，定义$i^x = i^{x \bmod \mathrm{ord}(i)}$ ，将 i 的幂生成的循环子群记作 $G_i = \langle i, \circ \rangle$ 。

引理 4.6. $\forall i \in G, i^0$ 是 G_i 中唯一的幂等元。

证. 若 i^j 是幂等元， $j = 2j, j = 0 \pmod{\mathrm{ord}(i)}$ 。 □

引理 4.7. $\forall i, j \in G, ((\exists x, y \text{ s.t. } i^x = j^y) \Rightarrow (i^0 = j^0))$

证. $(i^x = j^y) \Rightarrow ((i^x)^{\mathrm{ord}(x)\mathrm{ord}(y)} = (j^y)^{\mathrm{ord}(x)\mathrm{ord}(y)}) \Rightarrow (i^0 = j^0)$ □

由引理4.6和4.7，可以按 i^0 划分 G 。

定义 4.8. 设所有 $i^0 = x$ 的 i 组成等价类[43] C_x ；记等价关系为 $i \sim j$ ，当且仅当 $i^0 = j^0$ 。

引理 4.9. $\sim$ 保持运算 $\circ$ ，即 $\forall a, b, c, d \in G, (a \sim b \wedge c \sim d) \Rightarrow ((a \circ c) \sim (b \circ d))$ 。

由4.3和4.4我们定义等价类之间的运算：

[43]实际上，这是循环律下半格分解的特殊情况。

定义 4.10 (等价类运算)**.** 定义 $\circ : C_x \circ C_y = C_{x\circ y}$ ，等价类关于 $\circ$ 构成了一个交换半群，记为 G^C 。[44]

定理 4.11. G^C 的元素全部为幂等元， G^C 是半格[45]。

现在考虑对方程组的影响：

引理 4.12. 对于方程组的一个解 $\boldsymbol{x}$ ， $\forall i, j \in G, (i \sim j) \Leftrightarrow ([x_i = 0] = [x_j = 0])$ ，即同一个等价类中元素对应的 x 要么全为0要么全非0。

证. $x_i^{\mathrm{ord}(x)} = x_{i^{\mathrm{ord}(x)}} = x_{i^0} \Rightarrow ([x_i = 0] = [x_{i^0} = 0])$ □

对于一个等价类 i ，设 $S_i = \{j : j \in G^C, j \circ i = i\}$ ，有：

引理 4.13. S_i 互不相等，即 $\forall i \neq j, i, j \in G^C, S_i \neq S_j$ 。

证. 假设 $S_i = S_j$ ， $i \in S_i = S_j, j \in S_j = S_i$ ，可得 $i \circ j = i = j$ ，矛盾。 □

引理 4.14. 方程组的某个非平凡解 $\boldsymbol{x}$ 中非 0 元素对应的等价类集合必然等于某个 S_i 。

证. 设 x 中所有非 0 元素对应的等价类集合为 S ，设 $a = \mathop{\circ}\limits_{i\in S} i$ ，由于 R 无零因子，由方程组可得 $\forall i \in a, x_i \neq 0$ 。

由等价类幂等律， $\forall i \in S, a \circ i = (\mathop{\circ}\limits_{j\in S} j) \circ i = \mathop{\circ}\limits_{j\in S} j = a$ ，故 $S \subseteq S_a$ ；假设 $\exists i \in S_a, i \notin S$ ，由方程组可得 $\forall i \in a, x_i = 0$ ，矛盾，故 $S_a \subseteq S$ ；综上，有 $S_a = S$ 。 □

引理 4.15. $\forall x, C_x$ 是交换群。

证. 我们令 C_x 的幺元为 x ， i 的逆元为 $i^{\mathrm{ord}(i)-1}$ ，验证即可。 □

引理 4.16. （有限交换群基本定理）有限交换群 G 一定同构于若干质数幂阶循环群的直和[46]，即存在唯一的 $\{p_i\}, \{c_i\}$ 使得 $G \sim \bigoplus\limits_i \mathbb{Z}/p_i^{c_i}$ 。

证明 [4] 限于篇幅从略。

根据引理3.5，将卷积性变换和对应的 $\boldsymbol{x}$ 分为每一维 $\mathbb{Z}/p_i^{c_i}$ 考虑，对于这些循环群利用定理2.7经典DFT的结论即可。对于一个等价类 C_a ，DFT可以给出恰好 $|C_a|$ 组 $\boldsymbol{x}$ ，即 $|C_a|$ 次单位根的范德蒙德矩阵[47]的每一行。

得到等价类内的解 $\boldsymbol{x}_{C_a}$ 后，我们尝试构造一组完整的解 $\boldsymbol{x}$ 。由方程组必有 $x_a = 1$ ，那么 $\forall k(C_{k^0} \in S_{C_a})$ ，有 $x_k = x_k x_a = x_{k\circ a}$ ，由于$k \circ a \in C_a$ ， $x_{k\circ a}$ 已由DFT给定，故其余等价类的取值是唯一的。

现在要证明 $\{x_k = x_{k\circ a}\}$ 是合法解， $\forall i, j$ 分类讨论：

---

[44]注意， G^C 整体作为一个符号表示这个交换半群， C 无实义，请不要与第1 节的定义混淆。

[45]参见https://en.wikipedia.org/wiki/Semilattice

[46]对于交换群而言，直和是有限个交换群的直积。

[47]即为DFT对应的 $\boldsymbol{T}$ 矩阵

1. $i, j \in C_a$：由DFT必然成立；

2. $C_{i^0}, C_{j^0} \in S_{C_a}$ 且不满足1：$x_{i\circ j} = x_{(i\circ j)\circ a} = x_{i\circ j\circ(a\circ a)} = x_{(i\circ a)\circ(j\circ a)} = x_{i\circ a}x_{j\circ a} = x_i x_j$

3. $C_{i^0} \notin S_{C_a} \vee C_{j^0} \notin S_{C_a}$：不妨设 $C_{i^0} \notin S_{C_a}$，则必有 $C_{(i\circ j)^0} \notin S_{C_a}$，对应的方程为 $0x_j = 0$ 一定成立。否则假设 $C_{(i\circ j)^0} \in S_{C_a}$，有 $C_{i^0} \circ C_{j^0} \circ C_a = C_a, C_{i^0} \circ C_a = C_{i^0} \circ (C_{i^0} \circ C_{j^0} \circ C_a) = (C_{i^0} \circ C_{i^0}) \circ C_{j^0} \circ C_a = C_{i^0} \circ C_{j^0} \circ C_a = C_a$，矛盾。

这样，对每个等价类求一次，我们就得到了 $\sum_{C\in G^C} |C| = n$ 组互不相等的解。

我们还需要证明这 n 个解构造出的变换可逆。按照 G^C 的某种半格拓扑序[48]依次取出所有等价类，然后按这个顺序排列 $\boldsymbol{x}$，可得一个如下形式的变换矩阵：

$$\boldsymbol{T} = \begin{bmatrix} T_1 & \cdots & \cdots & \cdots & \cdots \\ 0 & T_2 & \cdots & \cdots & \cdots \\ 0 & 0 & T_3 & \cdots & \cdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & T_{|G^C|} \end{bmatrix}$$

其中 T_i 是第 i 个等价类的DFT变换矩阵。按DFT的性质，T_i 均可逆，那么 $\boldsymbol{T}$ 一定可逆。这样我们就得到了一个线性卷积性变换。

4.5 结论

定理 4.17. 在4.2节给出的前提下：G 满足循环性，且对于 G 按4.4节中分解得的每个循环子群 C，R^C 都存在DFT，是 R^G 上存在一个卷积性变换的充要条件。

5 算法

5.1 主算法

5.1.1 算法流程

[48]有限情况下，可以如下理解：我们将 G^C 排序，排序方法如下：每次取出一个满足 $S_C = C$ 的等价类C，然后删去 C 并重复直到完成。这种顺序一定是存在的：一方面，按照 S_C 的定义，删去 C 后的运算仍封闭，故得到的仍是一个有限半格。另一方面，非空时这样的 C 一定存在，因为若不存在，每个 C 都至少对应一个 X 满足 $X \neq C, X \circ C = C$，考虑图论模型，我们从 X 向 C 连一条有向边，可以发现边数与点数相等，又图是有限的，必定有环。由于每个等价类都不和自身连边，环中至少有两个不同元素。但对于一个环，考虑环中所有元素的 $\circ$ 运算的结果，按照边的定义可得该结果与环上每个元素相等，矛盾。

Algorithm 1 Construct DFT

Require: R, G which satisfie theoremlca 4.11

Ensure: transform matrix $\boldsymbol{T}$,inverse transform matrix $\boldsymbol{T}^{-1}$

```
1: function DFT(R, G)
2:   T ← 0 with order |G|, T^{-1} ← E with order |G|, G^C, C_i ← ∅, it ← 0
3:   for i ∈ G do
4:     insert i into C_{i^0}
5:     if C_{i^0} ∉ G^C then
6:       insert C_{i^0} into G^C
7:     end if
8:   end for
9:   G^C ← Toposort(G^C)
10:  for C_x ∈ G^C do
11:    B ← FiniteAbelGroupBase(C_x), S ← {x}
12:    T_{C_x} ← 0 with order |C_x|, T^{-1}_{C_x} ← 0 with order |C_x|, T_{C_x x,x} ← 1, T^{-1}_{C_x x,x} ← |C_x|^{-1}
13:    for i ∈ B do
14:      for j, k = 0 to ord(i) − 1, c, d ∈ S do
15:        T_{C_x i^j∘a, i^k∘b} ← ω^{jk}_{ord(i)} T_{C_x a,b}, T^{-1}_{C_x i^j∘a, i^k∘b} ← ω^{-jk}_{ord(i)} T^{-1}_{C_x a,b}
16:      end for
17:      S ← {a ∘ i^j : a ∈ S, j = 0 to ord(i) − 1}
18:    end for
19:    for i ∈ C_x do
20:      T_{it++} ← r ← {r_j = T_{C_x i, j∘x}}
21:    end for
22:  end for
23:  for C_x ∈ G^C by inverse order do
24:    T_{C_x} ← T^{-1}_{C_x} T_{C_x}, T^{-1}_{C_x} ← T^{-1}_{C_x} T^{-1}_{C_x}
25:    for C_y ∈ G^C, C_y ≠ C_x do
26:      T_{C_y} ← T_{C_y} − T_{C_y,C_x} T_{C_x}, T^{-1}_{C_y} ← T^{-1}_{C_y} − T^{-1}_{C_y,C_x} T^{-1}_{C_x}
27:    end for
28:  end for
29:  return T, T^{-1}
30: end function
```

5.1.2 时空效率

求 R 的 n 次本原单位根的次数是 $O(|G|)$ 的，以下不再考虑这部分对时间复杂度带来的影响。

下文中的时间效率与 R 中加法和乘法的调用次数均相同。

我们认为求 G 中 $\circ$ 的时间消耗为常数。

第11行调用了后文的交换群分解算法，该算法时空效率均为 $\Theta(|C_x|^2)$ ，由于 $|G| = \sum |C_x|$ ，总时空复杂度均不超过 $O(|G|^2)$ 。

第13到18行中，注意到每次 $|S|$ 的大小是上次的 $\mathrm{ord}(i)$ 倍，由于 $\mathrm{ord}(i) \geq 2$ ，有 $T(n) = O(T(\frac{n}{2})) + \Theta(n^2), T(n) = \Theta(n^2)$ ，故这部分时空复杂度均为 $\Theta(|G|^2)$ 。

第23到28行将 $\boldsymbol{T}$ 看作每个 T 组成的分块矩阵执行了求逆矩阵的算法。这部分使用高斯消元和朴素矩阵乘法的时间复杂度为 $\Theta(|G|^3)$ ，空间复杂度为 $\Theta(|G|^2)$ 。如果改用其它矩阵算法，时间复杂度为 $\Theta(|G|^\omega)$ ，其中 ω 表示矩阵乘法时间复杂度，已知的最优算法中 $\omega = 2.3728639$ 。[2]

其余部分的时空复杂度显然为 $\Theta(|G|^2)$ 。

综上，时间复杂度为 $\Theta(|G|^\omega)$ ，空间复杂度为 $\Theta(|G|^2)$ ，后者达到输入下界。

5.2 有限交换群分解

定义 5.1. 对于有限交换群 $(G, \circ)$ ，称 $B = \{b_i\}$ 是 G 的**基**[49]，当且仅当$G = \bigoplus_i \langle b_i \rangle$ ，其中 $\langle b_i \rangle$ 是 b_i 的生成子群。

下面是求出有限交换群 G 的一组基的算法：

5.2.1 算法描述

引理 5.2. 若一组元素 $B = \{b_i\}$ 满足：

1. b_i 线性无关，即不存在 $\{a_i\} \neq \mathbf{0}$ 使得 $\underset{i}{\circ} b_i^{a_i} = e$ ，其中 e 是 G 的幺元；[50]
2. $\forall x \in G, x$ 与 b_i 线性相关，即存在 $\{a_i\} \neq \mathbf{0}$ 使得 $\underset{i}{\circ} b_i^{a_i} \circ x = e$ 。

则 B 是一组基。

证. 由1， $\bigoplus_i \langle b_i \rangle$ 的元素两两不同且都属于 G ，可得 $\bigoplus_i \langle b_i \rangle \subseteq G$ ；由2， $G \subseteq \bigoplus_i \langle b_i \rangle$ ，故 $\bigoplus_i \langle b_i \rangle = G$ ，证毕。 □

[49]注意，这里的基不要求不可再分，于是阶也可以不为质数幂。另外此说法只对本文有效。

[50]注意这里的指数 b^a 的 a 都是在 $\pmod{\mathrm{ord}(b)}$ 意义下考虑的。

那么，我们从空基开始，每次取出与当前的基线性无关的的元素中阶最大者直到无元素可取即可，显然这样得到的基满足两个条件。

每次向基中加入一个元素 b_i 时，只要把每个元素 x 与 $x \circ b_i$ 划入同一关于当前基线性相关的等价类即可维护线性相关性。

5.2.2 算法流程

Algorithm 2 Decompose Finite Abel Group

Require: finite abel group G

Ensure: a group of base B

```
1: function FiniteAbelGroupBase(G)
2:   B ← ∅, S ← G \ {e}
3:   while S ≠ ∅ do
4:     max_ord ← 0
5:     for i ∈ S do
6:       if max_ord < ord(i) then
7:         b ← i, max_ord ← ord(i)
8:       end if
9:     end for
10:    insert b into B
11:    for i ∈ S do
12:      if ∃j s.t. i ∘ p^j ∉ S then
13:        remove i from S
14:      end if
15:    end for
16:  end while
17:  return B
18: end function
```

5.2.3 时空效率

每次循环时间复杂度是 $\Theta(|G||S|)$ 的，注意到每一次 $|S|$ 的规模除以 $\mathrm{ord}(b_i)$，由于 $\mathrm{ord}(b_i) \geq 2$，有$T(n) = O(T(\frac{n}{2})) + \Theta(n|G|), T(n) = \Theta(n|G|)$，故时空复杂度均为 $\Theta(|G|^2)$。

5.3 简化算法

上述算法按照证明推导过程实现，虽然思路清晰，但比较复杂。实际上我们可以得到一些效率相同的更简单的算法。

注意到 $\boldsymbol{T}$ 中元素的值只可能 0 和 $|G|$ 的约数次单位根，且每个元素是否为 0 可以通过等价类判断。我们定义符号 $\omega_{|G|}$ 表示 $|G|$ 次形式单位根，那么 $\boldsymbol{T}$ 的非零元素都可以写成 $\omega_{|G|}^{x}, x \in \mathbb{Z}_{|G|}$ 的形式。

这样，我们把方程组 $\forall i, j \in G, x_{i\circ j} = x_i x_j$ 转化成了线性方程组，解之即可。时间复杂度为 $\Theta(|G|^{\omega})$，空间复杂度为 $\Theta(|G|^2)$，与上述算法相同。

6 更一般的卷积

本节将讨论一些关于计算更一般的卷积的算法。限于作者的水平，只能做一些不系统的简单探索。

本节中我们依然在4.2节所述前提下讨论。

6.1 引入

在 R^G 上，其它运算（线性运算）都只需要 $\Theta(|G|)$ 的开销，这是输入下界。只有卷积这一非线性的运算有着 $\Theta(|G|^2)$ 的朴素代价。利用卷积性变换将卷积映射到简单的点积上是一个好方法——对于满足定理4.11的情况可以结合FFT等算法在 $O(|G|^2)$ 时间内完成计算，但其存在的条件是比较严格的。在卷积性变换不存在的情况下，如何加速卷积的计算呢？

基于研究内容，以下讨论的主要是**定义域复合**（而且是某个 G 的幂，即“位运算”）的情况。定义域复合的情况的显著特征是递归子问题和乘法开销大。

6.2 模型

在 R^G 中，利用第1节定义的运算——线性运算和卷积（乘法）——经过有限次复合，我们实际上得到的是 R^G_* 上的多项式 $R^G_*[x]$ 。

由于在环上我们不能做一般的除法，本节不考虑除法运算。[51]

在不考虑除法运算的情况下，由于卷积是二次的，计算卷积的过程中所能做的变换都不能超过二次，因此，只能先进行若干线性运算，计算这些运算结果的积（一次多项式乘法）后再将这些结果进行线性运算以求解。

设要计算 $\boldsymbol{a} = \{i \to a_i\}$ 和 $\boldsymbol{b} = \{i \to b_i\}$ 的卷积 $\boldsymbol{a} * \boldsymbol{b}$，计算过程可以由以下模型描述：

[51]尽管如此，有时通过形式除法，或者在需要的元素可逆，甚至 R 是域的情况下，利用除法运算可以得到比仅利用 R^G 上线性运算和乘法效率更高的算法，如一些基于多项式逆元的算法。此处限于水平和篇幅不再赘述。

$$\boldsymbol{a},\boldsymbol{b} \xrightarrow{\text{线性运算}} \{\boldsymbol{f}_i = \boldsymbol{A}_i\boldsymbol{a} + \boldsymbol{B}_i\boldsymbol{b}\} \xrightarrow{\text{卷积和线性运算}} \boldsymbol{a} * \boldsymbol{b} = \sum_{i,j} \boldsymbol{K}_{i,j}(\boldsymbol{f}_i * \boldsymbol{f}_j)$$

其中 $\boldsymbol{a},\boldsymbol{b},\boldsymbol{a}*\boldsymbol{b}$ 表示相应的列向量， $\boldsymbol{f}$ 表示中间值， $\boldsymbol{A},\boldsymbol{B},\boldsymbol{K}$ 表示系数矩阵。下文把第一次的线性运算过程称为**“第一步”**，把第二次的卷积和之后的线性运算过程称为**“第二步”**。

注意到，为了优于朴素算法，中间值不应超过 $|G|^2$ 个。这样依据上述模型，我们可以直接将寻找高效算法归结为一个组合优化问题[52]。这样，我们可以考虑直接使用求解组合优化问题的一般方法来解决问题，如搜索、贪心、结合随机化，机器学习等。

不过这样的做法太过暴力，下面我们试图做一些更深入的讨论。遗憾的是还没有什么好的一般性的结论。

6.3 一些讨论

6.3.1 时间效率

在第二步中卷积的计算方法当前有两种：递归计算，或利用DFT加速。不过，递归计算可以看成平凡群上的DFT，所以可以把运算看成若干次DFT的组合。只利用递归计算的算法被称为**分治乘法**。

设当前定义域的大小是 N ，计算卷积的时间复杂度为 $T(N)$ ，那么DFT变换时间为 $N \le T_{DFT}(N) \le N|G|, T_{DFT}(N) = \Theta(N)$ ，一次定义域大小为 n 的DFT的点值运算的时间为$nT(\frac{N}{|G|})$ 。设第 i 次DFT定义域大小为 n_i ，可得时间复杂度为：

$$T(N) = \sum_i n_i \cdot T(\frac{N}{|G|}) + T_{DFT}(\frac{n_i N}{|G|})$$

有 $\sum_i n_i \ge |G|$ （否则，卷积将有某些项是平凡的，我们可以删除它们来减小 G ）。解得：

$$T(N) = \begin{cases} \Theta(N^{\log_{|G|} \sum_i n_i}) & \sum_i n_i > |G| \\ \Theta(N \log N) & \sum_i n_i = |G| \end{cases}$$

设 $S = \sum_i n_i$ 。

朴素算法相当于 $|G|^2$ 次大小为 1 的DFT（分治乘法），其时间复杂度也可得为 $\Theta(N^2)$ 。

[52]通常状态空间是有限的，或者可以轻易找到一个最优解所在的有限集，不过这点在 $|R|$ 无限的一些情况下未必成立。

6.3.2 简单的优化

我们可以对算法做一个非常强的假设（限制），就是两步线性变换保持原本的元素不变（相当于乘置换），而且只有一次大小 $|G|$ 的DFT，其余都是分治乘法。

这样，我们的操作相当于花费 S 增加1的代价更改 G 的运算表的一个元素，然后再计算和补偿修改造成的影响。尽管这个算法简单且暴力，实际效果还是不错的，在 $|G| = 3, 4$ 时验证有 $S \le 8, 14$ ，优于朴素算法的 $S = 9, 16$ 。也即用这个方法可以得到 $|G| = 3, 4$ 时复杂度分别为 $O(n^{\log_3 8}) \approx O(n^{1.89279}), O(n^{\log_4 14}) \approx O(n^{1.90368})$ 的做法。

6.3.3 猜想

猜想 6.1. 存在一个时间复杂度最低的算法满足 $\forall i, \boldsymbol{A}_i = \boldsymbol{0} \vee \boldsymbol{B}_i = \boldsymbol{0}$ ，即中间值的每个序列都只含原序列 $\boldsymbol{a}$ 或 $\boldsymbol{b}$ 的项。

这个猜想很有可能是错的。不过我们还没有找到一个反例。

如果有这个条件，也就意味着 $\boldsymbol{f}_i$ 可以被分成关于 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 的两种，且卷积只会在两种之间计算。这样，对于组合优化是很有利的。

可惜的是，限于作者水平，没能得到更好的结论，也没有找到更好的资料，这一节的讨论只能暂时到此为止了。

7 总结

至此，我们解决了有限交换半群到环的DFT，得到了 $O(N|G|\log_{|G|} N)$ 的变换算法，由此对于可变换的系统可在线性时间内通过点值计算线性运算以及乘法。另外也对于一般的卷积（以至于二次运算）得到了优于朴素算法的方法。

本文介绍的是DFT在计算方面的一个具体应用，讨论内容比较基础和浅薄，发此妄言，也是希望能起到抛砖引玉的作用。时至今日，数学和计算科学界已经做了很多研究，感兴趣的同学可以参阅相关资料，可惜以作者的水平不能在截稿前学习和探索更多了。

8 后记

产生向竞赛同学们介绍DFT的想法，是半年前的事。当时我对竞赛中用到的FFT和FWT算法的原理非常疑惑：这些算法是如何发现和构造的？经过查阅资料和自己的探究，我得到了经典DFT和FWT算法的方法。我尝试扩展这些内容，后来有了第5节的结论。其中查找关于半群的资料比较困难（相对地，群的资料非常多），而且大多是抽象调和分析等比较偏纯数学的内容，虽然高深，但并不能直接用于计算优化——具体数学也是有必要的。于是我整理了自己的探究内容以及收集的资料，在2018年的冬令营上与大家分享。后来我试图

探究更一般的卷积和线性运算，得到了第6节的内容，无奈囿于数学（尤其是线性代数）水平，没有发现什么太好的结论，不过仍然得到了组合优化的模型和优于朴素的算法。

这些理论也许初看与竞赛关联不多，但抽象和概括毕竟是竞赛知识的繁杂名目缺少的一盏明灯，有了这样一般性的分析方法，我们就可以更好地理解以往所学的FFT、FWT、集合幂级数、莫比乌斯变换等模型和算法。不过也许，要求同学们接受这些理论，以至于在题目中考察，并不是这个已经明显学重于思的中国中学生信息学竞赛氛围下的最佳选择。

无论如何，希望本文的内容能对读者有所帮助和启发。

感谢中国计算机学会提供学习和交流的平台。

感谢国家集训队张瑞喆教练的指导。

感谢本校刘利丽老师和各位学长对我的帮助、培养和教导。

感谢唐珑珂、武弘勋学长以及梁晏成、王成瑞、王泽宇同学关于内容的帮助。

感谢孙奕灿、赵义凯、张晴川、张明远学长以及倪锦涵同学为本文验稿。

参考文献

[1] Donald E. Knuth, “The Art of Computer Programming, Volume 2, Seminumerical Algorithms”. Massachusetts:Addison-Wesley, 1997.

[2] Le Gall, Francois, “Powers of tensors and fast matrix multiplication”.

[3] Jorg Arndt, “Matters Computational”.

[4] Wikipedia, the free encyclopedia, “Finitely generated abelian group”, https://en.wikipedia.org/wiki/Finitely_generated_abelian_group

[5] 彭雨翔, “Picks’s Blog”, http://picks.logdown.com

[6] 吕凯风,《集合幂级数的性质及其快速算法》, IOI2015中国国家队候选队员论文集。

[7] 毛啸, 《再探快速傅里叶变换》, IOI2016中国国家队候选队员论文集。

[8] 梁晏成,《Jellyfish命题报告及拓展探究》, IOI2018中国国家队候选队员论文集。

[9] 刘承奥,《浅谈卷积定理在OI中的应用及扩展》, 2018年NOI冬令营营员交流。

《完美的队列》命题报告

福建师范大学附属中学 林旭恒

摘 要

本文介绍了作者在此次集训队互测中命制的一道传统数据结构题。

本题题意简单，正解使用到的算法及代码实现都不复杂，却十分考验选手对问题性质的分析能力。本文将介绍其解法和命题背景。

1 试题

1.1 问题描述

有 n 个队列，编号为1到 n 。其中第 i 个队列的容量限制为 a_i ，一开始所有队列均为空。

给出 m 次操作，每次操作用 $l\ r\ x$ 来描述，表示往编号在 $[l,r]$ 之间的队列末尾插入元素 x 。若有队列在插入操作后其中元素个数超过其容量限制，删除该队列最前端的元素。

你需要计算出每次操作结束后所有队列中不同的元素种数。

1.2 输入格式

第一行两个正整数 n,m ，分别表示队列个数和操作个数。

第二行 n 个正整数，第 i 个表示 a_i 。

接下来 m 行，每行三个正整数 $l\ r\ x$ ，其中第 i 行表示第 i 次操作。

1.3 输出格式

输出共 m 行，每行一个非负整数，表示第 i 次操作结束后所有队列中的不同的元素种数。

1.4 数据范围与约定

对于所有数据， $n, m, a_i, x \le 10^5$ ， $l \le r$ 。

共20个测试点，每个测试点5分，其中第 k 个测试点满足 $n, m, a_i, x \le 5000k$ 。

特别地，以下几个测试点满足一些特殊性质：

测试点5： $a_i = 1$ 。

测试点7： $a_i = 2$ 。

测试点9： $a_i = 10$ 。

测试点11： $a_i \le 10$ 。

测试点13,15： $\sum a_i \le 10^6$ 。

对于每个测试点，你需要通过满足该点数据范围及性质的所有数据才能获得该点的分数。

2 朴素算法

2.1 模拟

直接使用数组或者C++提供的std::queue来模拟每个操作。

用一个数组记录每种权值的出现次数，在插入和删除的时候维护这个数组，并在权值的存在性被改变时更新答案。

时间复杂度 $O(nm)$ ，空间复杂度 $O(\sum a_i)$ 。

根据实现的不同，期望得分5~15分。

2.2 初步分析

直接模拟的做法忽视了太多可以用到的性质，现在我们对问题进行初步的分析。

问题并没有要求我们在线处理每次操作，所以我们可以考虑离线处理这个问题。先考虑每个操作对答案的影响，每个操作对答案存在影响的时间显然只在这个操作执行之后，该操作插入的所有元素都被删除之前。我们把一个操作插入的所有元素都被删除的时间称为这个操作的被删除时间。

如果我们能求出每个操作的被删除时间，那么不难计算出我们需要的答案。一种方法是先将每个操作的存在时间抽象成时间轴上的区间，那么对于插入元素相同的操作，这种元素对答案有贡献的时间为这些时间区间的并。我们对每种元素求出区间的并之后利用差分数组即可快速计算答案。另一种方法是将问题转化成维护一个可重集合，原问题的每个操作转化为在执行操作时往集合中加入一个该操作插入的元素，在该操作被删除时删除集合中的一个该元素，那么原问题的答案与这个集合中不同的元素个数相等，用模拟做法中提到的方法不难维护不同的元素个数。第二种方法在实现上会比第一种略微简单一些。

那么接下来我们的问题完全转化为求每个操作的被删除时间。

2.3 基于初步分析的朴素算法

容易想到，一个操作插入的所有元素都被删除的时间就是该操作插入的每个元素被删除的时间（若始终未被删除，视作正无穷）中的最大值。我们考虑分别计算每个元素的出队时间。不难发现，在第 i 个队列中，第 j 次插入的元素会在第 $j + a_i$ 次插入时被删除。我们可以对每个队列分别求出哪些操作往这个队列中加入了元素，然后暴力更新其中每个操作的被删除时间。

时间复杂度 $O(nm)$，空间复杂度 $O(n + m)$。

期望得分20~30分。

3 在线算法

这个部分与正解的思路关联不大，其中的算法以特殊数据的形式作为部分分来考察选手在对正解没有特别清晰的思路时能否通过其他方式来获得更高的分数。以下通过分别介绍各个特殊测试点的解法来介绍此部分的算法。

3.1 测试点5： $a_i = 1$

每个队列的容量限制恰好为1。问题相当于要求维护一个序列，每次操作将序列的一个区间中所有元素修改为 x，并且计算序列中不同的元素种数。

如何维护这个序列呢？传统的维护区间修改操作往往要用到线段树，但是这个问题中要求的不同的元素种数难以在线段树上直接维护。

注意到一个事实：由于我们的修改操作是将整段区间全部修改为 x，我们维护的序列中往往会出现大段的重复元素。我们能否针对这个性质设计一个算法来维护这个序列呢？

答案是肯定的。我们可以将相邻的相同元素合并成一整段，在修改操作时，不难根据之前每一段元素与修改区间的位置关系来维护这些元素段，答案也可以通过维护每种元素出现了多少段来计算。显然，在修改区间均匀随机的情况下，不同的元素段始终不会太多，这个做法的效率十分可观。

然而，在构造数据中，不同的元素段数很容易就能达到 $O(m)$ 的级别（我们不妨假设 m 与 n 同级），暴力维护元素段的时间复杂度将不能接受。

我们可以用数据结构来优化这个维护的过程。首先，只有与修改区间相交的元素段才会被修改，如果我们能快速找到这些元素段，就能进一步优化我们的算法。

不难想到，我们可以用平衡树（或是C++提供的std::set）按序列中的位置顺序来维护这些元素段，那么每次我们可以用 $O(\log m)$ 的时间快速找到一个与修改区间相交的元素段，

并处理修改操作带来的影响。

然而，与一个修改区间相交的元素段的数量仍然可能达到 $O(m)$ ，似乎这个优化本质上并没有降低我们的时间复杂度。但我们注意到，一个元素段如果被修改区间包含，这个元素段会被直接删除，又由于我们维护的元素段之间互不相交，与一个修改区间相交但不被包含的元素段最多只有2个。那么除了这至多2个元素段，其他与修改区间相交的元素段都被我们删除。而每次操作新增的元素段也至多只有2个（修改区间被某个元素段包含时，元素段被分成两段，又新增一段，相较原来多2段，否则被修改的元素段要么被缩短要么被删除，只新增一段），故所有操作能产生的元素段只有 $O(m)$ 个，那么删除元素段的次数也不会超过 $O(m)$ 次。于是我们证明了这个算法的时间复杂度为 $O(m\log m)$ ，足以通过该测试点的数据。

实现时，我们只需要维护这些元素段的左端点。为了方便实现，我们可以假设开始时存在一个覆盖整个序列的空元素段，并且只将同一个操作修改出的相同元素合并成一段，显然这样并不影响复杂度。那么在修改时，修改区间的左端点以及右端点的下一个位置必然成为修改后某个元素段的左端点，直接加入平衡树即可。而原来在这两个端点之间的元素段端点可以直接删除。

直接维护左端点的做法省去了许多不必要的特判，思路也更为清晰。而从这个做法的过程来看，每次只加入2个新端点和删除旧端点，总修改次数为 $O(m)$ 的结论也更加明显。

3.2 测试点7： $a_i = 2$

当容量限制增加到2时，问题不再是简单的维护一个序列。但我们仍然可以将其转化成一个序列问题：我们需要维护两个序列，分别表示每个队列的两个元素，每次操作我们要将第一个序列的 $[l,r]$ 区间修改为 x ，而每有一个元素被修改，第二个序列上的对应位置就要被替换为这个元素。

仍然考虑使用测试点5的算法，维护第一个序列的元素段，当第一个序列有元素段被修改时，被缩短或是删除的部分就要到第二个序列对应位置上进行区间修改。由于我们在维护第一个序列的元素段时，对元素段的总修改次数为 $O(m)$ ，那么我们对第二个序列进行的区间修改次数也为 $O(m)$ ，使用一样的算法即可维护。

这个算法的总时间复杂度同为 $O(m\log m)$ ，可以通过该测试点。

3.3 测试点9： $a_i = 10$

现在我们考虑容量限制均为 K 的情况。不难想到，我们可以用与容量限制均为2时类似的方法。我们维护 K 个序列，其中第 i 个序列表示每个队列中的第 i 个元素，每一个序列的元素段在被修改的同时对下一个序列进行区间修改。如果一个序列的修改次数为 $O(m)$ ，那么下一个序列的修改次数也为 $O(m)$ 。于是我们似乎理所当然地得到了一个时间复杂度为 $O(Km\log m)$ 的算法。

但是注意到一点，在计算维护元素段的复杂度时，我们得出的 $O(m)$ 次修改实际上忽略了其中的常数因子。而这个常数因子很可能会在我们将修改由一个序列转到下一个序列时不断扩大，甚至以指数级增长。例如，是否有可能第一个序列需要 $2m$ 次，第二个需要 $4m$ 次，第三个需要 $8m$ 次，第 K 个需要 $2^K m$ 次呢？

我们不妨换一个角度分析这个算法的复杂度。我们考虑所有元素段端点。事实上，在每次操作中，所有序列中新增的端点只会在操作的左右端点（前后，下略）处出现，也就是最多增加 $O(K)$ 个。但这并不意味着我们的总时间复杂度为 $O(Km\log m)$ ，因为我们并没有在所有修改时都至少删去一个端点。

那么如何计算复杂度呢？我们可以这么考虑：对于每个位置，定义其深度为其作为元素段端点出现的第一个序列的编号。那么每次操作时，操作的左右端点的深度会被更新为1，而其他所有被我们影响到的端点位置，深度都会被加1，如果超过 K 则被完全删除。也就是说，每次操作加入的所有新端点至多只会被我们处理 $O(K)$ 次，那么这个算法真正的时间复杂度为 $O(K^2 m\log m)$ 。

实际上，这个算法也可以进一步优化。注意到每次操作我们做的修改实际上等同于先在修改区间的左右端点处插入元素段端点，然后将修改区间中在第 K 个序列的元素段删除，在其他序列的元素段移动到下一个序列。我们可以用支持区间插入删除的平衡树（如splay）来维护元素段端点，这样就可以快速地将元素段移动到下一个序列。

优化后，每个端点只会在被从第 K 个序列中删除时被我们处理一次，每次将各个序列的元素段移动到下一序列的时间复杂度是 $O(K\log m)$ ，所以总时间复杂度为 $O(Km\log m)$ 。

3.4 测试点11： $a_i \le 10$

当容量限制不一定相等时，问题变得更加复杂。一种想法是对每种容量限制都用测试点9的优化后的算法分开维护，这样做的时间复杂度为 $O((\max a_i)^2 m\log m)$ 。

实际上我们有一个更简单的做法。考虑将所有队列一起维护，维护 $\max a_i$ 个序列，此时序列可能因为某个队列的容量不足而被“拦腰截断”，但是如果我们也把对这些序列的修改截断来强行维护，复杂度显然是错误的。

我们换一个思路，把序列中被截断的位置补上，在被补上的位置的元素没有实际意义，只是用来辅助我们维护序列。而这些序列中某个元素段有至少一个元素真实存在于队列中当且仅当这个元素段对应的所有队列中容量限制 a_i 的最大值大等于这个元素段所在序列的编号。预处理之后用RMQ不难 $O(1)$ 判断一个元素段是否真实存在。我们先不考虑用高级平衡树来优化。根据之前的复杂度分析，这个做法的时间复杂度也为 $O((\max a_i)^2 m\log m)$ 。

如果用支持区间插入删除的平衡树优化，相较测试点9的做法，此时元素段可能在移动到下一个序列的时候就失去对答案的贡献。我们可以对每个仍然真实存在的元素段用RMQ求出其移动到哪一个序列时会失去贡献，然后在平衡树上维护这个值的最小值。那么我们每次移动元素段时不难在平衡树上找到失去贡献的颜色段并处理对答案的影响。

可以证明，优化后的时间复杂度为 $O(\max a_i m\log m)$ 。

3.5 测试点13,15： $\sum a_i \le 10^6$

在 a_i 的总和有限制时，不容易出现太多容量较大的队列。

我们可以按 a_i 分类， $a_i \le K$ 的队列用之前的方法一起维护， $a_i > K$ 的队列直接模拟。总时间复杂度为 $O(Km\log m + \frac{\sum a_i}{K}m)$ 。选取合适的 K 值，我们得到一个时间复杂度为 $O(m\sqrt{\sum a_i \log m})$ 的做法，配合一些常数优化可以通过该部分数据。

4 二分法

下面介绍此题的一类离线解法，该类解法通过二分或一些其他类似的方法找到每个操作的被删除时间来计算答案，能获得可观的分数。

4.1 判断合法性

之前我们已经提到，如果能求出每个操作的被删除时间，就可以得到答案。现在我们的问题是如何快速求出每个操作的被删除时间。

我们很自然地想到对每一个操作通过二分法来求其被删除时间，那么我们就将问题转化为了一个判定性问题：第 x 个操作插入的元素在第 mid 个操作结束后是否被完全删除？

首先，我们知道，在第 i 个队列中，第 j 次操作插入的元素会在第 $j + a_i$ 次，即其往后第 a_i 次操作时被删除。也就是说，只有第 x 次操作之后的操作才会对它有影响。我们用 b_i 表示第 x 次操作在第 i 个队列插入的元素还需要多少次在 i 号队列的插入才会被删除。那么在第 x 个操作执行时，其影响的区间 $[l, r]$ 中所有 b_i 都会被初始化为 a_i 。而之后的每个操作，都会使这个操作对应的区间中所有 b_i 减1，一旦有 b_i 被减为 0 ，说明第 x 次操作在这个队列里插入的元素被删除了。当所有 b_i 都被减为0时，第 x 次操作插入的所有元素都被删除。

为了方便，我们允许 b_i 小于0，那么每次操作相当于区间减1，当 x 号操作插入的区间 $[l, r]$ 中 b_i 的最大值小等于0 时， x 号操作插入的元素均被删除。

区间减和区间最大值可以用线段树维护。于是我们得到一个判定方法：以 a_i 为初始值建线段树，将时间在 $(x, mid]$ 内的每个操作对应的 $[l, r]$ 区间减1，再判断 x 号操作插入的区间 $[l, r]$ 的最大值是否小等于0。

如果对于每次判定都从头建线段树并执行这些操作来判断，总时间复杂度为 $O(n^2\log^2 n)$ （我们认为 n 与 m 同级，不作区分），甚至不如朴素做法。

但我们注意到，如果我们已经得到了时间在 $[l, r]$ 内的每个操作的区间都减1后的线段树，再用 $O(\log n)$ 的时间进行一次区间加1或减1操作就可以得到时间在 $[l-1, r]$ ， $[l+1, r]$ ， $[l, r-1]$ ， $[l, r+1]$ 四者之一的线段树。

显然，我们可以利用这个性质设计一些算法来优化时间复杂度。

4.2 整体二分

我们设计一个类似分治的算法。用 $solve(S, l, r)$ 表示我们在时间 $[l, r]$ 中寻找操作集合 S 中每个操作的被删除时间。我们先取 $[l, r]$ 的中点 mid ，然后依次判断 S 中的每个操作在第 mid 个操作后是否被完全删除，并依此将 S 分成两部分，分别递归到 $solve(S_1, l, mid)$ 和 $solve(S_2, mid + 1, r)$ 。

为了优化我们的判断时间，我们只维护一棵线段树，并记录其对应的时间区间（即对哪些操作进行了区间减1）的左右端点，在需要查询时，用每次 $O(\log n)$ 时间的代价将左端点或右端点往相邻位置移一步，一直移动到我们需要的位置。那么我们的复杂度就只跟我们的左右端点进行了多少次移动有关。特别地，第一次查询相当进行了 $O(n)$ 次端点的移动，以下分析中略。

先考虑右端点，我们可以这么考虑：在 $solve(S, l, r)$ 时，右端点在 mid ，调用 $solve(S_1, l, mid)$ 时，我们将右端点从 mid 移动到 $[l, mid]$ 的中点，结束调用时移动回来；调用 $solve(S_2, mid + 1, r)$ 时，我们将右端点从 mid 移动到 $[mid + 1, r]$ 的中点，同样在结束调用时移动回来。事实上我们不难发现，右端点的实际移动次数还要更少一些，但这么分析已经足够了。这样右端点的移动次数与 $O(r - l)$ 同级，故右端点的总移动次数不会超过 $O(n\log n)$ 。

然而，对于左端点，我们无法保证其总移动次数在可以接受的范围内。由于所有操作的被删除时间分布并没有特别的性质，每个 $solve$ 函数的调用中，左端点的移动次数都可能达到 $O(n)$ 次，极端情况下，左端点的总移动次数可以达到 $O(n^2)$ 次，难以接受。

我们注意到，左端点的移动只会在 S 集合中编号最小到编号最大的操作的范围内进行。我们可以将操作按照编号分成 K 个块，每个块调用一遍 $solve(S_i, 1, m)$ ，其中 S_i 表示这个块内所有操作构成的集合，那么我们每次判断时进行的左端点移动都不会超过 $O(\frac{n}{K})$ 次。而每个操作都会被递归并判断 $O(\log n)$ 次，故左端点的总移动次数不会超过 $O(\frac{n^2\log n}{K})$ 次。又因为我们总共调用了 K 次 $solve(S_i, 1, m)$ ，右端点的总移动次数为 $O(Kn\log n)$ 。再算上线段树的复杂度，这个做法的总时间复杂度为 $O((\frac{n^2}{K} + Kn)\log^2 n)$ 。

K 取 $O(\sqrt{n})$ 时，时间复杂度为 $O(n\sqrt{n}\log^2 n)$ 。由于实际中这个算法的时间复杂度是不满的，配合一些优化可以得到相当可观的分数。

4.3 类莫队算法

我们回到直接二分的做法。

注意到我们进行左右端点移动的操作恰好与莫队算法类似。常规的莫队算法是将询问按左端点分块，每一块按右端点的顺序依次处理，左端点只在块内移动，右端点不断向右移动，以此保证复杂度。而我们可以设计一个类似的算法来满足我们二分时判断合法性的需求。

我们按判断时需要查询的时间区间（即之前提到的 $(x, mid]$ ）的右端点分块，然后为每个块建一棵线段树，并记录其对应的左右端点。查询时我们找到右端点所在的块，将对

应的线段树左右端点移动到查询的位置上。如果我们按时间顺序依次对每个操作进行二分，不难发现，所有线段树的左端点只会向右移动，如果共 K 个块，左端点的总移动次数就是 $O(Kn)$ 的。而右端点只在块内移动，又因为我们二分时总共要进行 $O(n\log n)$ 次判断，故右端点的总移动次数为 $O(\frac{n^2\log n}{K})$ 。算上线段树的复杂度，总时间复杂度为 $O((\frac{n^2\log n}{K}+Kn)\log n)$ 。

K 取 $O(\sqrt{n\log n})$ 时，时间复杂度为 $O(n\sqrt{n}\log^{1.5}n)$ 。

4.4 类莫队算法的改进

在之前的算法中，我们将所有线段树左端点不断向右移的过程中实际上得到了以每个操作为左端点，右端点在每个块内的信息，而二分的做法并没有完全利用到这些信息，我们不妨改进一下这个做法。

我们在时间轴上均匀设置 K 个关键点，相当于之前的块。然后分别以每个关键点为右端点，计算每个左端点的信息。在计算的同时，我们判断每个操作到每个关键点时是否被删除，以此确定每个操作的被删除时间在哪两个关键点之间。这部分的时间复杂度为 $O(Kn\log n)$ 。

最后，我们对每对相邻关键点，将被删除时间在其之间的操作按顺序通过依次判断每个时间点的方式找到被删除时间。这样对于每对相邻关键点，左端点不断向右移，总移动次数为 $O(Kn)$ 。对于每个操作，其在对应的相邻关键点之间暴力判断所有右端点，故右端点的总移动次数为 $O(\frac{n^2}{K})$ 。算上线段树的复杂度，总时间复杂度为 $O((\frac{n^2}{K}+Kn)\log n)$ 。

K 取 $O(\sqrt{n})$ 时，这个算法的时间复杂度为 $O(n\sqrt{n}\log n)$ 。

5 序列分块

此部分为本题的标准解法，相较之前的算法更好地利用了本题的性质。

5.1 单调性

我们注意到一个显然的事实：在每个队列中，每个元素的被删除时间是随插入时间单调递增的。可是对于所有操作，它们的被删除时间却并不以操作时间单调递增。其原因是不同的操作插入的元素可能在不同的队列中，而不同的队列之间是没有关联的。

我们从中产生一个想法：如果有若干个操作插入的队列区间完全相同，它们之间的被删除时间是否按时间顺序单调递增呢？

答案是显然的。在每个队列中，元素的被删除时间随时间递增，那么插入的队列完全相同的操作，它们的被删除时间，也就是插入的每个元素被删除时间的最大值，自然也随时间递增。

我们可以设计这样一个算法来计算插入队列完全相同的操作的被删除时间：先用与上一部分相同的方式，对 a_i 建线段树，然后以第一个要求的操作为左端点，右端点不断向右推，直到我们要求的操作的插入区间中最大值小等于0，我们就求出了第一个要求的操作的被删除时间。接下来我们按时间顺序求每一个要求的操作的被删除时间，由于它们的被删除时间单调递增，每次我们只需要将左端点移到对应位置，右端点继续往右推即可。因为我们的左右端点都是不断往右推的，总移动次数为 $O(n)$ ，所以求一次插入队列为某一区间的所有操作的被删除时间的时间复杂度为 $O(n\log n)$ 。

可是两个不同的操作插入的区间很可能并不相同，如何才能利用这个性质呢？对于每个操作，我们只关心其插入的每个元素被删除时间中的最大值，我们可以将每个操作通过将插入区间划分成若干个子区间的方式分成若干个子操作，然后分别求出这些子操作的被删除时间，再取其中的最大值即可。

我们想到可以将所有队列按编号分块，那么如果每一块的大小设为 K ，每个操作就可以被分成 $O(\frac{n}{K})$ 个整块操作和 $O(1)$ 个非整块操作。由于总共只有 $O(\frac{n}{K})$ 个不同的块，我们可以对每个块用前面提到的算法求出所有这个块的整块操作的被删除时间，这部分的总时间复杂度为 $O(\frac{n^2\log n}{K})$ 。

而对于非整块操作，总共只有 $O(n)$ 个，也就是最多只插入 $O(nK)$ 个元素，我们考虑直接求出其中每个元素的被删除时间。由于第 i 个队列中第 j 个插入的元素会在第 $j + a_i$ 次插入时被删除，我们可以先求出一个操作在第 i 个队列中是第几个操作，再求出其在第 i 个队列中往后第 a_i 个操作是哪个操作。

我们可以按队列的编号顺序依次求出每个队列对应的操作集合，对于每个操作，在其插入区间的左端点处加入集合，右端点处删出集合。要支持查询信息，只需要按操作时间维护一个树状数组，支持前缀求和以及在树状数组上二分即可。总时间复杂度为 $O(nK\log n)$ 。

故 K 取 $O(\sqrt{n})$ 时，我们得到了一个 $O(n\sqrt{n}\log n)$ 的做法。

5.2 对整块操作的优化

在对每个块计算这个块的整块操作的被删除时间时，我们注意到，实际用到的 a_i 只有块内的部分。

我们可以只对块内的 a_i 建线段树，所有操作对块外的影响直接无视。此时我们发现，绝大多数的区间加1和减1操作，都是直接对整块进行的。事实上，在分块时，每个操作对每个块的影响也被分成了 $O(\frac{n}{K})$ 个整块和 $O(1)$ 个不完整块。也就是说，我们总共要进行 $O(\frac{n^2}{K})$ 次整块加减和 $O(n)$ 次区间加减。直接打标记就可以 $O(1)$ 完成一次整块加减，时间复杂度降为 $O(\frac{n^2}{K} + n\log n)$ 。事实上，我们可以直接用数组模拟代替线段树来完成区间加减和维护最值，整块加减仍然使用标记，复杂度虽然变为 $O(\frac{n^2}{K} + nK)$ ，但是更易实现并且不会影响总复杂度。

对非整块操作，我们用与之前相同的做法。适当选取 K 值，时间复杂度降为 $O(n\sqrt{n\log n})$，可以通过全部数据。

5.3 对非整块操作的优化

在之前通过维护树状数组来计算非整块操作的被删除时间的做法中，并没有利用到同一队列中元素被删除时间单调递增的性质。我们能否对非整块操作也设计一个与整块类似的算法呢？

我们可以将每个块分开考虑。对于一个块，我们将这个块的整块操作集合称为 A，这个块内的非整块操作集合称为 B。我们需要对 B 中每个操作求出其在该块中插入的每个元素的被删除时间。对于块中某个队列 i，我们将 B 中对这个队列有影响的操作集合称为 B_i。对于 B_i 中的每个操作 x，我们求出 B_i 中最早的操作 y，满足在 y 执行前，x 在 i 中插入的元素已被删除。根据同一队列中元素被删除时间单调递增的性质，y 显然也单调不降，那么我们仍然可以在 B_i 中用左右端点不断向右推的方法求出，问题是如何判断 x 插入的元素是否已被删除，即计算两个操作之间有多少个对 i 有影响的操作。由于对 i 有影响的只有 B_i 和 A 中的操作，我们可以对每个块预处理出 B 中每个操作的前一个 A 中操作在 A 中是第几个，即可 $O(1)$ 计算。在求出 y 后，通过一些简单的计算，我们不难得出操作 x 在 i 中插入的元素的实际被删除时间是 y 往前第几个对 i 有影响的操作，通过预处理的信息可以 $O(1)$ 求出这个操作。那么除了预处理的时间，我们只用 $O(|B_i|)$ 的时间就能求出 B_i 中所有操作在 i 中的被删除时间。

不难证明，用这个算法优化后，总时间复杂度为 $O(\frac{n^2}{K}+nK)$。K 取 $O(\sqrt{n})$ 时，我们得到了一个时间复杂度为 $O(n\sqrt{n})$ 的优秀算法。

6 命题背景和得分情况

本题最初的原型是这样一道题目：要求维护一个序列，支持区间修改为 x 和查询区间中不同权值种数。原题利用了本文在线算法部分提到的维护元素段的方法，再将每个元素分别以其下标和前一个相同权值的下标抽象成二维平面上的点，将询问转化为矩形求和解决。在偶然中，我产生了通过计算每个修改的被删除时间来得出答案的想法，并专门为这个想法设计了这道题目。在此之后，我不断改进自己的算法，最终将此题放在集训队互测与大家分享。

近来，随着广大OI选手对数据结构的研究不断深入，许多命题人常用复杂的模型和高级的数据结构及算法来提升题目的难度，并形成了许多常见的“套路”。这不仅使得做数据结构题往往需要掌握许多难度较大的算法和大量零碎的知识点，还需要具备很强的代码能力，与竞赛锻炼人思维的初衷相违背。

相较之下，本题的标准解法仅使用到了分块以及two pointer（指左右端点不断向右推的

做法）的知识点。最终标算的一个实现中，只用到了最简单的数组，没有用到包括线段树在内的任何较高级的数据结构，且代码长度仅1.4kb。

在此基础上，本题仍然具有一定的思维难度和区分度。在命题时，我预测候选队中会有1~3人通过这道题目，一半左右的人能通过各种做法拿到较高的分数。而在互测中，候选队中无人通过该题，3人拿到了60以上的分数，略微少于预期，说明大家对这种类型的题目还不够擅长。此外，在LOJ的同步赛中，来自北京大学的吉如一学长现场用 $O(n\sqrt{n})$ 的做法通过了该题。

7 总结

本题考察了选手对问题性质的分析，以及针对性质设计算法的能力。特殊的部分分设计方式能根据选手不同的算法与实现，让选手获得不同的分数。而要拿到高分，不仅需要对性质的分析，还需要对分块算法和基础数据结构的充分理解与应用。

希望本题能对大家有所启发，加强对基础数据结构的灵活运用。

致谢

感谢中国计算机学会提供交流和学习的平台。

感谢张瑞喆教练的指导。

感谢周成老师对我的指导和帮助。

感谢黄哲威学长在信息竞赛路上对我的启发与帮助。

感谢所有在赛前赛后与我交流讨论本题的同学们，其中特别感谢吉如一学长分享他的做法。

感谢胡学浚同学和陈喆桓同学为本文审稿。

感谢各位抽出宝贵时间阅读这篇文章。

感谢所有帮助过我的人。

参考文献

[1] 徐明宽，《非常规大小分块算法初探》，2017年国家集训队论文。

浅谈拟阵的一些拓展及其应用

江苏省淮阴中学 杨乾澜

摘 要

拟阵是一个信息学竞赛中大家熟知但一直被忽略的概念。本文将介绍一些关于拟阵的重要的拓展操作，使得拟阵可以表达出大量新的组合问题，在组合优化问题的求解中更加有力。希望本文能够让读者深入了解拟阵，体会到拟阵在组合优化问题中的重要作用。

1 引言

拟阵 (matroid) 是研究组合优化问题的一种重要手段，由 Hassler Whitney 最早开始研究，在组合优化问题的研究中有着重要的地位。

信息学竞赛中，早在 2007 年就已经出现系统引入拟阵的论文 [1] 。然而遗憾的是，多年以来，大部分选手对于拟阵的认识简单止步于拟阵可以用来证明一类贪心的正确性。

拟阵是对“独立”这一概念的抽象，它将广泛存在于图论，线性代数中的很多共通性质抽象出来，单独形成一个概念，使之不依赖于具体图论，线性代数等的性质。在有了关于拟阵的一些运算之后，更是可以将很多概念归于拟阵，使用拟阵解决很多问题。

本文将从拟阵的定义概念开始，着重介绍拟阵的一些运算，并引入一些拟阵的研究方法。希望能让大家感受到拟阵的重要性和魅力。

本文第二节讲介绍拟阵的一些基础定义，第三节将证明拟阵上的最优化问题的正确性，第四节第五节将分别引入拟阵的交和拟阵的并，第七节会补充一些拟阵的重要特性和一些研究拟阵的方法。

2 引入

2.1 定义

定义 2.1. 记 $M = (S, \mathcal{I})$ 表示一个定义在一个有限集 S 上，独立集的集合是 $\mathcal{I}$ 的拟阵。其中 $\mathcal{I} \subseteq 2^S$ [53]，即 $\mathcal{I}$ 是由一些 S 的子集组成的集合。拟阵 M 需要满足以下公理：

[53] 设 S 是一个集合，2^S 表示 S 的幂集，即 S 的所有子集组成的集合

1. 如果 $I \in \mathcal{I}, J \subseteq I$ 那么 $J \in \mathcal{I}$

2. 如果 $I, J \in \mathcal{I}$ ，且 $|J| > |I|$ ，那么存在元素 $z \in J \backslash I$ 满足 $I \cup z \in \mathcal{I}$ 。

定义 2.2. 如果 $I \in \mathcal{I}$ ，我们称 I 是独立的，也称 I 是独立集，否则称 I 是不独立的。

通常为了方便，我们认为空集 $\varnothing$ 是独立的。

简单来说，拟阵是这样一种数学结构，它拥有一个有限的定义集合 S ，一些 S 的子集被认为是独立的，他们共同构成了 $\mathcal{I}$ 。拟阵的公理规定了独立集的条件，是拟阵的基础。通常来说，第一条公理被称为遗传性公理，即每个独立集的所有子集都是独立的。第二条公理通常被称为交换性公理或扩张性公理，即如果独立集 A 存在更大的独立集 B ，那么存在元素只在 B 不在 A 中，可以使得 A 扩张到更大的独立集。

这两条公理也是拟阵的定义的一种[54]，是抽象出来的很多问题拥有的一般的性质，一个组合优化问题一旦满足这两条公理，那么所有的拟阵的定理都在这个组合优化问题上成立。

为了更好的理解，我们可以看几个简单的拟阵或非拟阵的例子。

例子 2.1 (均匀拟阵[55]). 令拟阵 $U_n^k = (S, \mathcal{I})$ ，其中 $|S| = n$ ， $\mathcal{I} = \{I \subseteq S : |I| \le k\}$ 。

这个例子非常容易证明两条公理，在此略去。

例子 2.2 (图拟阵). 令无向图 $G = (V, E)$ ，令图拟阵 $M = (E, \mathcal{I})$ ，其中 $\mathcal{I} = \{F \subseteq E : F\text{无环}\}$ 。

这个例子就是拟阵中非常重要的图拟阵，定义集合是边集，独立集是一个生成森林。我们可以证明一下这个结构是一个拟阵结构。

首先遗传性的存在比较显然，如果一个边集的子集 F 无环，那么 F 的子集肯定是无环的。

交换性的证明也不难，考虑假设存在两个边集的子集 A, B ，满足 $|A| < |B|$ ，那么 A 形成的图的联通块数量必然比 B 形成的图的联通块数量多，所以必然存在一个 B 形成的图中的联通块， A 形成的图中不连通，所以必然存在一条 B 中的边可以加入 A ，且不会产生环。

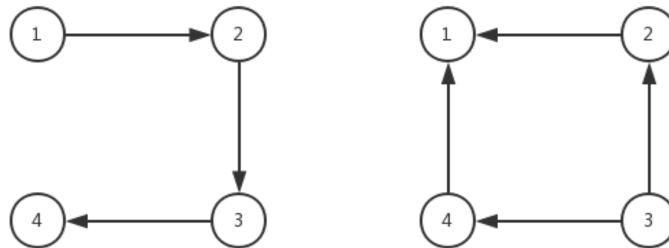
例子 2.3 (有向图“拟阵”). 令有向图 $G = (V, E)$ ，令拟阵 $M = (E, \mathcal{I})$ ，其中 $\mathcal{I} = \{F \subseteq E : F\text{无环}\}$ 。

既然无向图的生成树结构是拟阵结构，自然猜想有向图上也是如此，然而遗憾的是这个例子是错的，有向图组成的组合结构并不是一个拟阵。可以通过一个简单的反例证明。

注意到右图的任意一条不同于左图的边加入左图都会导致产生环，所以在这个结构上交换性是不存在的。

[54]后文将讨论另外一种重要的定义

[55]Uniform matroid，作者未找到标准翻译



例子 2.4 (匹配拟阵)**.** 令无向图 $G = (V, E)$ ，匹配拟阵 $M = (V, \mathcal{I})$ ，其中一个点集是独立集，当且仅当存在一个匹配能够覆盖点集中的所有点（可以覆盖点集外的点）。

这个例子中，遗传性的证明比较显然，而交换性的证明比较困难，一种比较传统的方法是通过增广路来证明，但这与本文的主题无关，在此略去。后文我们会从拟阵并的角度来证明二分图匹配问题是一个拟阵。

2.2 基与环

定义 2.3. 对于拟阵 $M = (S, \mathcal{I})$ 中的一个独立集 I ，如果 I 加入任何一个属于 $S \setminus I$ 的元素都会变成非独立集，则称 I 是拟阵的一个基，也叫极大独立集。

定义 2.4. 对于拟阵 $M = (S, \mathcal{I})$ 中的一个非独立集 I' ，如果 I' 去掉任何一个属于 I' 的元素都会变成独立集，则称 I' 是拟阵的一个环，也叫极小非独立集。

在图拟阵中，一个基就是一个极大生成森林，一个环就是一个简单环。特别的，如果这个图联通，那么一个基就是一个生成树。

在线性代数中，一个基就是一个线性空间的基，一个环就是一个最小的线性相关的集合。

之前我们说空集 $\varnothing$ 一定是独立集，所以这里可以得到空集 $\varnothing$ 一定不是环。

关于基有一些简单的结论。

引理 2.1. 对于任意拟阵 M ，所有基的大小相同。

证明. 如果存在两个基 A, B ， $|A| < |B|$ 。那么根据交换性， A 可以继续扩大，命题得证。

定理 2.2 (基交换定理)**.** 对于任意拟阵 M ，如果存在两个不同的基 A, B ，那么对于任意一个 $z \in A \setminus B$ ，都存在一个 $y \in B \setminus A$ ，满足 $A - \{z\} + \{y\}$ 是独立集。

证明. $|A - \{z\}| < |B|$ ，根据交换性公理，一定存在一个元素 y 满足 $y \in B \setminus (A - \{z\}) = B \setminus A$ ，命题得证。

我们令 $C(M)$ 表示拟阵 M 的所有的环，关于环有如下性质。

推论 2.3. 如果存在环 $X, Y \in C(M)$ ，且 $X \subseteq Y$ ，那么 $X = Y$ 。

证明. 由定义可得。

推论 2.4. 如果存在环 $X, Y \in C(M), e \in X \cap Y$ ，且 $X \neq Y$ 。那么存在环 $C \in C(M)$ 满足 $C \subseteq X \cup Y - \{e\}$ 。

证明. 由推论 2.3 可得， $X \backslash Y$ 非空，设 $f \in X \backslash Y$ 。假设 $X \cup Y - \{e\}$ 是独立集。由于 X 是环，所以 $X - \{f\}$ 是独立集，假设 $X \cup Y$ 中最大的包含 $X - \{f\}$ 的独立集是 Z 。由于 Y 不是独立集，所以 $Y \subsetneq Z$ 。由此可得 $|Z| \le |X \cup Y - \{f\}| - 1 \le |X \cup Y| - 2 < |X \cup Y - \{e\}|$ 。因为 Z 已经是最大的独立集了，所以 $X \cup Y - \{e\}$ 不可能是独立集，命题得证。

推论 2.5. 令 I 是拟阵 $M = (S, \mathcal{I})$ 的一个基， $e \notin I$ 。那么 $I + e$ 包含一个唯一的环。

证明. 假设 $I + e$ 存在两个不同的环 C_1, C_2 ，由于 e 的加入导致环的产生，所以 $e \in C_1 \cap C_2$ ，由推论 2.4 可得， $C_1 \cup C_2 - \{e\}$ 包含环，这与 I 是独立集矛盾。命题得证。

2.3 秩函数

定义 2.5. 对于拟阵 $M = (S, \mathcal{I})$ ，基的元素的个数称为拟阵的秩，对于任意一个 S 的子集 U ，定义秩函数 $r(U)$ 表示 U 中极大独立集的大小。

秩函数拥有如下几种性质。

定理 2.6 (有界性)**.** $\forall U \subseteq S, 0 \le r(U) \le |U|$

定理 2.7 (单调性)**.** $\forall A \subseteq B \subseteq S, r(A) \le r(B)$

定理 2.8 (次模性)**.** $\forall A, B \subseteq S, r(A \cup B) + r(A \cap B) \le r(A) + r(B)$

对于引理 2.6 和 2.7 都可以由定义直接得来，我们重点证明引理 2.8。

证明. 令 $A \cap B$ 中的最大独立集是 Z ,由交换性可得，从 Z 开始，可以扩张出 A 中的最大独立集 Z_A ， B 中的最大独立集 Z_B ，和 $A \cup B$ 中的最大独立集 Z_{AB} 。 Z_{AB} 我们可以直接将其划分为分别属于 $A \cap B, A \backslash B, B \backslash A$ 的三部分，由遗传性得到，任意部分的组合都是独立集。令 Z_{AB} 在 $A \backslash B$ 中的部分为 E_A ， $B \backslash A$ 中的部分是 E_B ，可以得到 $r(A \cap B) + r(A \cup B) = |E_A| + |E_B| + |Z| + |Z| = (|Z| + |E_A|) + (|Z| + |E_B|) \le |Z_A| + |Z_B| = r(A) + r(B)$ 。

这里我们给出了秩函数的几个重要性质。事实上，通过这几个性质，我们可以直接定义出唯一的函数 $r : 2^S \to \mathbb{Z}_+$ 。通过这个函数，我们甚至可以直接定义出拟阵，即 $M = (S, \mathcal{I}), \mathcal{I} = \{I : r(I) = |I|\}$ 。

如果我们证明这个结论的正确性，那么我们就得到了一条重要的研究拟阵的手段，代数化。我们将拟阵的组合问题转化为秩函数上的代数问题，通过研究秩函数的性质来求得

拟阵的性质。如果我们定义的一个函数满足秩函数的条件，那么对应的组合问题就满足拟阵的条件。

我们下面将给出通过秩函数的性质推得拟阵公理的过程。

证明. 我们首先证明遗传性，令 B 是任意独立集， A 是 B 的任意子集，由次模性可得 $|B| = r(B) \le r(A)+r(B-A)$,由有界性可得 $r(A) \le |A|, r(B-A) \le |B-A|$ 。由此可得 $r(A) = |A|, r(B-A) = |B-A|$ ，即 A 是独立集。

交换性的证明稍微复杂一些，令 A, B 是任意独立集，满足 $|A| < |B|$ 。我们尝试通过反证证明命题。假设 A 加入 B 中任一元素都不是独立集。令 B 中元素分别为 $b_1, b_2, \ldots, b_{|B|}$ ，我们尝试通过归纳推出矛盾。由于 A 加入任一元素都不是独立集，所以 $r(A \cup \{b_1\}) = |A|$ ，假设我们已经有 $r(A \cup \{b_1, b_2, \ldots, b_n\}) = |A|$ ，由次模性可得 $r((A \cup \{b_1, b_2, \ldots, b_n\}) \cap (A \cup \{b_{n+1}\})) + r((A \cup \{b_1, b_2, \ldots, b_n\}) \cup (A \cup \{b_{n+1}\})) \le r(A \cup \{b_1, b_2, \ldots, b_n\}) + r(A \cup \{b_{n+1}\})$ 。由于 $(A \cup \{b_1, b_2, \ldots, b_n\}) \cap (A \cup \{b_{n+1}\}) = A$ ，所以 $r(A \cup \{b_1, b_2, \ldots, b_{n+1}\}) = |A|$ 。依次归纳，我们可以得到 $r(A \cup B) = |A|$ ，而由单调性可得 $|B| = r(B) \le r(A \cup B)$ ，所以矛盾。

3 拟阵上的最优化

拟阵上的最优化问题是这样定义的，给定拟阵 $M = (S, \mathcal{I})$ ，给定函数 $\omega : S \to \mathbb{R}$ ，定义 S 的一个子集的权值 $\omega(I) = \sum_{e \in I} \omega(e)$ 。需要找一个最大的 $\omega(I)$ ，满足 $I \in \mathcal{I}$ 。

拟阵上的最优化问题是拟阵最简单的应用，一个组合优化问题如果可以证明符合拟阵的模型，那么就可以通过简单的贪心解决这个问题。具体来说算法流程如下。

1. 将元素根据 ω 按照不降的顺序排列，得到 $\omega(s_1) \ge \omega(s_2) \ge \cdots \ge \omega(s_{|S|})$
2. 维护一个集合 I ，初始时为空，按照顺序依次考虑每个 s_i ，如果 $I \cup \{s_i\}$ 是独立集，则加入元素 s_i 。

我们考虑证明这个算法是正确的，证明分为两步，第一步我们证明最后得到的 I 是拟阵的一个基，第二步我们证明这个基是最优的解。

我们通过归纳证明第一步，令 $U_i = \{s_1, s_2, \ldots, s_i\}$,我们证明对于任意的时刻 i ，满足 $r(U_i) = |I|$ 。对于 $i = 0$ 命题显然成立，假设命题对于 i 成立，尝试证明对于 $i + 1$ 成立。一共有两种情况：

1. $r(U_{i+1}) = r(U_i)$ ，情况显然。
2. $r(U_{i+1}) > r(U_i)$ ，如果 $I \cup \{s_{i+1}\}$ 是独立集，那么问题得证，否则考虑现在的一组基 I' ，由于 $|I'| > I$ ，根据交换性，必有一元素 $z \in I' \backslash I$ 可以加入 I ，由于这个元素不可能是 s_{i+1} ,所以这个元素必定属于 U_i ，这就说明之前存在一个大小超过 $r(U_i)$ 的独立集，这与事实矛盾。

第二步将证明这个基是最优解。假设真正的最优解第一步与 I 不同的选择是 s_i，即最优解选择了 s'_i，那么由于 $\omega(s'_i) \le \omega(s_i)$，所以之后必然存在某个选择 $\omega(s_j) \le \omega(s'_j)$，但是由于基交换定理，$I - \{s_j\} + \{s'_j\}$ 也一定是独立集，所以贪心算法一定会在遇到 s'_j 时选择的。

由此我们证明了拟阵上的最优化问题，可以通过简单的贪心解决，于是一类具有拟阵结构的问题，我们都可以证明其正确性，例如最小生成树，例如二分图匹配。对于一些新的问题，如果我们能分析出交换性的存在（遗传性一般比较显然存在），那么我们就可以通过简单的贪心解决，例如最小生成环套树森林等。

4 一些补充定义

为了方便引入拟阵的交与并，在此之前，我们需要补充定义一些关于拟阵的基础操作。

4.1 对偶拟阵

定义 4.1. 对于拟阵 $M = (S, \mathcal{I})$，定义 M 的对偶拟阵 $M^* = (S, \mathcal{I}^*)$，其中 $\mathcal{I}^* = \{I : \text{存在}M\text{中的基}B \subseteq S \setminus I\}$。

上述定义中，我们不加证明地断言了一个拟阵的对偶一定还是一个拟阵。为了之后的内容更加严谨，我们还是需要证明这是一个拟阵。

在这里我们尝试换一种证明的方法，我们来证明这个拟阵的秩函数是合法的。令 $r^*(U)$ 表示拟阵 M^* 的秩函数，首先我们需要求出这个 $r^*(U)$：

$$\begin{aligned}
r^*(U) &= \max_{I \subseteq U, I \subseteq \mathcal{I}^*} |I| \\
&= \max_{\text{B 是 M 中的基}} |U \setminus B| \\
&= |U| - \min_{\text{B 是 M 中的基}} |U \cap B| \\
&= |U| - r(S) + \max_{\text{B 是 M 中的基}} |B \cap (S \setminus U)| \\
&= |U| - r(S) + r(S \setminus U)
\end{aligned}$$

现在我们证明这个秩函数满足我们的几条公理[56]：

1. 有界性：由于 $r(S \setminus U) \le r(S)$，所以 $r^*(U) \le |U|$。

[56]如果以秩函数定义拟阵，那么引理 2.6-2.8 将成为公理

2. 单调性：令 $T \subseteq U \subseteq S$ ，那么 $r^*(T) = |T| - r(S) + r(S \setminus T) \leq |T| - r(s) + (r(S \setminus U) + r(U \setminus T)) \leq |T| - r(S) + r(S \setminus U) + (|U| - |T|) = |U| - r(S) + r(S \setminus U) = r^*(U)$ 。

3. 次模性：令 $A, B \subseteq S$ ， $T = S \setminus A, U = S \setminus B$ 。由 r 的次模性可得， $r(T \cup U) + r(T \cap U) = r(S \setminus (A \cap B)) + r(S \setminus (A \cup B)) \leq r(S \setminus A) + r(S \setminus B)$ 。另外 $|A \cup B| + |A \cap B| = |A| + |B|$ ，由于其他几项全部相同，所以 $r^*(A \cup B) + r^*(A \cap B) \leq r^*(A) + r^*(B)$ 。

4.2 删除和收缩

这一节中我们将要规定两个关于拟阵的操作，删除[57]和收缩[58]。

定义 4.2. 对于拟阵 $M = (S, \mathcal{I})$ ，和 $Z \subseteq S$ ，定义拟阵 M 删除子集 Z 的拟阵为 $(S \setminus Z, \mathcal{I}'), \mathcal{I}' = \{I : I \subseteq S \setminus Z, I \in \mathcal{I}\}$ 。记为 $M \setminus Z$ 。

定义 4.3. 对于拟阵 $M = (S, \mathcal{I})$ ，和 $Z \subseteq S$ ，定义拟阵 M 收缩子集 Z 的拟阵为 $(M^* \setminus Z)^*$ ，记为 M/Z 。

删除操作比较好理解，等价于直接不再考虑 Z 这个子集得到的新的拟阵，秩函数也可以直接沿用原拟阵的秩函数。

收缩操作比较难以理解，我们尝试推导出新的拟阵的秩函数以帮助我们理解。

$$\begin{aligned}
r_{M/Z}(U) &= |U| - r_{M^* \setminus Z}(S \setminus Z) + r_{M^* \setminus Z}((S \setminus Z) \setminus U) \\
&= |U| - r_{M^*}(S \setminus Z) + r_{M^*}((S \setminus Z) \setminus U) \\
&= |U| - (|S \setminus Z| - r_M(S) + r_M(Z)) + (|(S \setminus Z) \setminus U| - r_M(S) + r_M(Z \cup U)) \\
&= r_M(Z \cup U) - r_M(Z)
\end{aligned}$$

通过这个秩函数，可以理解到收缩操作的本质就是强制选取了 Z 中的一组基， M/Z 中的独立集的要求就是并上 Z 中的独立集还是 M 的一个独立集，事实上收缩这个名称可以对应到图拟阵中的缩边操作。

对一个拟阵施以以上任意一个操作得到的都是一个拟阵，这点可以通过秩函数证明，这里不做赘述。

4.3 极小元

定义 4.4. 对于拟阵 M ，经过一系列删除和收缩操作得到的任意拟阵 M' ，称作拟阵 M 的极小元。

[57]deletion，作者未找到标准翻译

[58]contraction，作者未找到标准翻译

拟阵的很多性质都可以它的极小元来表现，不严谨的说，拟阵的极小元可以看成原拟阵的一个局部特征。通过研究极小元，我们可以推出更一般的拟阵的性质，后文将使用到这一工具。

5 拟阵的交

在本节之前，拟阵可以刻画的问题比较少，要求比较严格，能够解决的问题也比较少，因此在信息学竞赛中没有收到重视，大部分选手仅仅只有拟阵可以用于证明一类贪心正确性的印象。但从这一节开始，我们将引入拟阵的交与并，通过这两种操作，我们将可以通过拟阵刻画出更多的组合优化问题。我们也可以看到许多的组合问题可以通过拟阵解决。

定义 5.1. 给定两个定义在相同基础集上的拟阵 $M_1 = (S, \mathcal{I}_1), M_2 = (S, \mathcal{I}_2)$ ，定义 M_1 和 M_2 的交是所有的集合 I ，满足 I 同时在两个拟阵内都是独立的。

拟阵的交问题要求的就是同时属于两个拟阵的独立集中最大的独立集，如果两个拟阵的交还是一个拟阵的话，那我们可以通过简单的贪心算法解决这个问题。然而遗憾的是这并不是一个拟阵，反例普遍存在。我们需要考虑其他手段解决拟阵交问题，下面一个定理可以帮助我们解决问题。

定理 5.1 (最小最大定理)**.** $\max\limits_{I \in \mathcal{I}_1 \cap \mathcal{I}_2} |I| = \min\limits_{U \subseteq S}(r_1(U) + r_2(S \setminus U))$

这个定理将成为我们解决拟阵交问题的关键，我们首先考虑证明这个定理比较简单的一面 $\max \le \min$ 。这个问题比较容易解决：

$$\begin{aligned} |I| &= |I \cap U| + |I \cap (S \setminus U)| \\ &\le r_1(U) + r_2(S \setminus U) \end{aligned}$$

由此可得，我们只要成功求出一组 I 和 U ，满足这个等式，我们就解决了拟阵交问题。下文我们将给出一种算法，并构造性的证明这个定理的正确性。在此之前我们还需要一些铺垫工作。

5.1 强基交换定理

我们首先定义一种有益于证明的集合。

定义 5.2. 对于拟阵 $M = (S, \mathcal{I})$ 和 $A \subseteq S$ ，我们定义 A 的闭包算子 $cl(A) = \{e \in S : r(A \cup \{e\}) = r(A)\}$ 。

闭包算子有如下的性质：

引理 5.2. 对于拟阵 $M = (S, \mathcal{I})$ ，如果 $A \subseteq B$ ，那么 $cl(A) \subseteq cl(B)$ 。

证明. 假设 e 是 $cl(A)$ 中的一个元素，由次模性可得 $r(A\cup\{e\})+r(B)\geq r((A\cup\{e\})\cap B)+r(B\cup\{e\})\geq r(A)+r(B\cup\{e\})$ 。左右消去相等的 $r(A\cup\{e\})$ 和 $r(A)$ 可以得到 $r(B)\geq r(B\cup\{e\})$ ，即 $r(B)=r(B\cup\{e\})$ ，所以 $e\in cl(B)$ 。因为任意一个 $cl(A)$ 中的元素都属于 $cl(B)$ ，所以 $cl(A)\subseteq cl(B)$ 。

引理 5.3. 对于拟阵 $M=(S,\mathcal{I})$ 和 $A\subseteq S$ ，如果 $e\in cl(A)$ ，那么 $cl(A)=cl(A\cup\{e\})$ 。

证明. 从引理 5.2 可以得到 $cl(A)\subseteq cl(A\cup\{e\})$ ，我们只要证明 $cl(A\cup\{e\})\subseteq cl(A)$ 即可。类似之前的证明，假设 f 是 $cl(A\cup\{e\})$ 中的一个元素，利用次模性，得到 $r(A\cup\{e\})+r(A\cup\{f\})\geq r(A\cup\{e,f\})+r(A\cup(\{e\}\cap\{f\}))\geq r(A\cup\{e,f\})+r(A)$ ，消去两侧相同的，得到 $r(A\cup\{f\})\geq r(A\cup\{e,f\})$ ，即 $r(A\cup\{f\})=r(A\cup\{e,f\})$ 。由于 $f\in cl(A\cup\{e\})$ ，所以 $r(A\cup\{e,f\})=r(A\cup\{e\})$ 。进一步可以得到 $r(A\cup\{f\})=r(A)$ ，即 $f\in cl(A)$ ，命题得证。

引理 5.4. 对于拟阵 $M=(S,\mathcal{I})$ 和 $A\subseteq S$ ， $cl(A)=cl(cl(A))$ 。

证明. 利用引理 5.3 逐步加入 $cl(A)\backslash A$ 的元素即可。

定理 5.5 (强基交换定理)**.** 对于拟阵 M ，假设存在两个不同的基 A,B ，那么对于任意一个元素 $x\in A\backslash B$ ，都存在一个元素 $y\in B\backslash A$ ，满足 $A-\{x\}+\{y\}$ 和 $B-\{y\}+\{x\}$ 都是拟阵的基。

证明. 既然 B 是基，那么 $B+\{x\}$ 包含一个唯一的环 C ，且 $x\in C$ 。我们可以得到 $x\in cl(C-\{x\})$ ，运用引理 5.2 可以得到 $x\in cl((A\cup C)-\{x\})$ 。由引理 5.3 可以得到 $cl((A\cup C)-\{x\})=cl(A\cup C)$ 。既然 A 是基，那么 $S=cl(A)\subseteq cl(A\cup C)$ 。由此可得 $(A\cup C)-\{x\}$ 含有一个基，令它为 A' 。由于 $A-\{x\}$ 和 A' 都是独立集，且 $|A'|>|A-\{x\}|$,所以必定存在一个元素 $y\in A'\backslash(A-\{x\})$ 使得 $A-\{x\}+\{y\}$ 是基，由于 $A'\backslash(A-\{x\})\subseteq((A\cup C)-\{x\})\backslash(A-\{x\})\subseteq C-\{x\}$ ，所以存在元素 $y\in C-\{x\}$ ，满足 $A-\{x\}+\{y\}$ 是基，另一方面由于 C 是环，所以 $B-\{y\}+\{x\}$ 也是基，命题得证。

5.2 算法

在本节中，我们将提出一种算法。它从初始为空的独立集开始，逐渐拓展，当它停止运行的时候，我们就得到了最大的独立集 I 。同时我们可以构造出 U ，以此证明这就是我们要求的最大独立集。我们的算法全部定义在一种叫交换图的二分图上。

定义 5.3. 对于给定的拟阵 $M=(S,\mathcal{I})$ ，和一个独立集 $I\in\mathcal{I}$ ，定义 M 中 I 的交换图 $D_M(I)$ 是这样一种二分图。它的两部分点集各自由 I 和 $S\backslash I$ 构成，一条连接 $y\in I$ 和 $x\in S\backslash I$ 的边存在，当且仅当 $I-\{y\}+\{x\}\in\mathcal{I}$ 成立。

引理 5.6. 令 I 和 J 都是拟阵 M 的独立集，且 $|I|=|J|$ 。那么在 $D_M(I)$ 中存在一个关于 $I\backslash J$ 和 $J\backslash I$ 的完美匹配。

证明. 首先为了方便使用强基交换定理，我们令新的拟阵 M' 中独立集为 $\{I' : I' \in \mathcal{I}, |I'| \leq |I|\}$ 。那么 I 和 J 都是新的拟阵的基。任取一个 $J \backslash I$ 中的元素 x ，根据强基交换定理，一定存在一个 $I \backslash J$ 中的元素 y ，满足 $I - \{y\} + \{x\}$ 和 $J - \{x\} + \{y\}$ 都是独立集。根据定义，边 (y, x) 一定存在，我们取这条边为一个匹配，并把 J 变成 $J - \{x\} + \{y\}$ 。如此递归下去，可以证明命题成立。

定理 5.7. 令 I 是拟阵 M 的独立集， J 是一个大小与 I 相同的集合。如果在二分图 $D_M(I)$ 中存在一个**唯一的** $I \backslash J$ 到 $J \backslash I$ 的完美匹配，那么 J 也是独立集。

证明. 令 N 是唯一的匹配，将 N 中的边定向为从 $S \backslash I$ 到 I ，将其他边定向为从 I 到 $S \backslash I$ 。匹配的唯一性，表明了图中不存在有向的环。那么这一定是个拓扑图，我们可以给每个点一个标号，使得所有的边都从小的编号指向大的编号。我们将 N 中的点重标号，令其为 $N = \{(y_1, , x_1), (y_2, x_2), \ldots, (y_t, x_t)\}$ ，其中 $x_i \in J \backslash I$ 。不失一般性，我们规定，当 $i < j$ 时， (y_i, x_j) 不存在边。使用反证法，假设 J 不是独立集。那么 J 中必定存在一个环 C ，取最小的 i 满足 $x_i \in C$ 。由于之前的构造，对所有 $x \in C - \{x_i\}$, (y_i, x) 都不存在边，即 $C - \{x_i\} \subseteq cl(I - \{y_i\})$ 。所以 $cl(C - \{x_i\}) \subseteq cl(cl(I - \{y_i\})) = cl(I - \{y_i\})$ 。因为 C 是环，所以 $x_i \in cl(C - \{x_i\})$, 由此可得 $x_i \in cl(I - \{y_i\})$ ，但是这与 (y_i, x_i) 是交换图的边矛盾，所以 J 必定是独立集。

定义 5.4. 给定两个拟阵 M_1, M_2 ， 和一个集合 $I \in \mathcal{I}_1 \cap \mathcal{I}_2$ ， 定义关于 I 的交换图 $D_{M_1,M_2}(I)$ 是这样一个二分图。它的两部分点集各自由 I 和 $S \backslash I$ 构成，一条从 $y \in I$ 指向 $x \in S \backslash I$ 的有向边存在，当且仅当 $I - \{y\} + \{x\} \in \mathcal{I}_1$ ，一条从 $x \in S \backslash I$ 指向 $y \in I$ 的有向边存在，当且仅当 $I - \{y\} + \{x\} \in \mathcal{I}_2$ 。

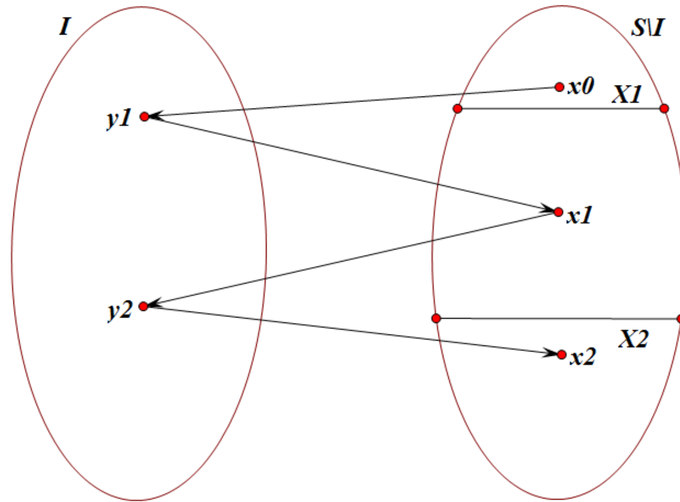
所有的铺垫工作已经做完，我们可以叙述算法流程了。

令初始的 I 为 $\varnothing$ 。我们定义集合 $X_1 = \{x \notin I : I + \{x\} \in \mathcal{I}_1\} X_2 = \{x \notin I : I + \{x\} \in \mathcal{I}_2\}$ 。算法将会每次在当前的交换图 D_{M_1,M_2} 中找到一条从 X_1 到 X_2 的路径，满足没有任何一条边能够缩短这条路径的长度，令其为 P 。然后我们令 I 变为 $I \Delta P$ [59]。我们称这样的一次过程为一次增广过程，重复增广过程，直到找不到路径。我们就得到了最大的 I ，并找到了 $U = \{z \in S : z\text{可以到达}X_2\}$ 。

注意如果 $X_1 \cap X_2$ 不是空集，那么算法将会直接拓展一个属于交集的元素。

[59] Δ 表示两个集合的对称差

下图描述了算法的一次增广过程。



其中所有的 (y_i, x_i) 均表示 $I - \{y_i\} + \{x_i\} \in \mathcal{I}_1$，所有的 (x_i, y_{i+1}) 均表示 $I - \{y_i\} + \{x_i\} \in \mathcal{I}_2$。

该算法用伪代码表示如下：

Algorithm 1 求拟阵交的最大独立集

Require: 定义在相同基础集 S 上的两个拟阵 M_1 和 M_2

Ensure: 最大公共独立集 I

1: $I \leftarrow \varnothing$
2: **repeat**
3: 构建 $D_{M_1,M_2}(I)$
4: $X_1 \leftarrow \{z \in S \setminus I : I + \{z\} \in \mathcal{I}_1\}$
5: $X_2 \leftarrow \{z \in S \setminus I : I + \{z\} \in \mathcal{I}_2\}$
6: 求出 $D_{M_1,M_2}(I)$ 中 X_1 到 X_2 的最短路 P
7: $I \leftarrow I \Delta P$
8: **until** P 不存在

为了证明这个算法是正确的，我们需要证明两点。一是最后算法运行结束之后，$|I| = r_1(U) + r_2(S \setminus U)$，二是算法运行中的任意一个时刻 $I\Delta P$ 都是独立集。

首先证明 $|I| = r_1(U) + r_2(S \setminus U)$，我们先证明 $r_1(U) \le |I \cap U|$。如果 $r_1(U) > |I \cap U|$，那么存在元素 $x \in U \setminus I$，使得 $(I \cap U) + \{x\} \in \mathcal{I}_1$。但如果 $I + \{x\}$ 也是 M_1 中的独立集的话，那么 $x \in X_1$，即存在一条从 X_1 到 X_2 的路径，所以 $I + \{x\}$ 不是 M_1 的独立集。综上可得，存

在一个元素 $y \in I \backslash U$ ，满足 $I - \{y\} + \{x\} \in \mathcal{I}_1$ ，但这就导致了 y 也可以到达 X_2 ，即 y 也应该是 U 的元素，与事实矛盾，命题得证。

另外一侧的证明 $r_2(S \backslash U) = |I \cap (S \backslash U)|$ 类似上述的证明，这里就不再重复。

最后考虑如何证明对于每一步 $I \Delta P$ 都是独立集。

令最短路 $P = (x_0, y_1, x_1, y_1, \ldots, x_t, y_t)$ ，令 $J = \{x_1, x_2, \ldots, x_t\} \cup (I \backslash \{y_1, y_2, \ldots, y_t\})$ 。那么可以得到 $|J| = |I|$ ，且 $I \backslash J$ 与 $J \backslash I$ 有唯一的完美的匹配。根据定理 5.7 可以得到 J 是 M_1 中的独立集。接下来考虑 x_0 的加入，既然 P 是最短路，那么 $x_1, x_2, \ldots, x_t$ 都不是 X_1 的元素，那么 $r_1(I \cup J) = r_1(I) = r_1(J)$ 。由于 $I + \{x_0\}$ 是独立集，沿用拟阵最优化问题时的证明方法，$J + \{x_0\}$ 也是独立集，命题得证。

我们证明了上述算法的正确性，同时证明的最小最大定理的正确性，并给出了一个解决拟阵交的算法。

5.3 复杂度

令 $r = \max(r_1(S), r_2(S))$ ，我们每次构建出来的图都是 n 个点， rn 条边的图。在无权图上找最短路的复杂度是 $O(n + m)$ 的，因此单次增广的复杂度就是 $O(rn)$ 的。考虑到每次增广独立集中必然增加一个元素，所以增广次数不会超过 r 次。因为拟阵交算法的复杂度是 $O(r^2 n)$ 的。

5.4 带权拟阵的交

带权拟阵的交问题类似于拟阵上的最优化问题。给定一个权值函数 $\omega : S \rightarrow \mathbb{R}$ ，需要求的是 $\max\limits_{I \in \mathcal{I}_1 \cap \mathcal{I}_2} \sum_{e \in I} \omega(e)$ 。

带权问题的解决和一般问题类似，通过拓展的最小最大定理:

$$\max_{I \in \mathcal{I}_1 \cap \mathcal{I}_2} \sum_{e \in I} \omega(e) = \min_{\omega_1, \omega_2 : \forall x, \omega_1(x) + \omega_2(x) = \omega(w)} (\max_{I \in \mathcal{I}_1} \omega_1(I)) + (\max_{I \in \mathcal{I}_2} \omega_2(I))$$

而实际的算法中，我们只要在交换图中定义点权：

$$f(x) = \begin{cases} \omega(x) & x \in S \backslash I \\ -\omega(x) & x \in I \end{cases} \tag{16}$$

求最短路的过程改为求一条从 X_1 到 X_2 的路径，第一关键字是点权最小，第二关键字是经过的边数最小即可。我们可以证明交换图上不存在负权环，所以这个算法一定有解。

5.5 多个拟阵的交

类比拟阵的交，我们可以定义多个拟阵的交，即最大的同时属于所有拟阵的独立集的集合。不过遗憾的是，大于等于三个拟阵的交是 NP-Hard 的，我们可以通过规约哈密顿路

径问题来证明。考虑有向图 $G = (V, E)$ ，求 s 到 t 的哈密顿路径，我们构造三个拟阵。第一个拟阵 $M_1 = (E, \mathcal{I}_1)$,其中的独立集需要满足将有向边看做无向边，整个图无环，这是经典的图拟阵。第二个拟阵 $M_2 = (E, \mathcal{I}_2)$,其中的独立集需要满足对于任意一个非 s 的点，入度不超过 1 ，对于 s ，入度不超过 0 。这是一个非常简单的拟阵，很好证明。第三个拟阵 $M_3 = (E, \mathcal{I}_3)$ ，其中的独立集需要满足对于任意一个非 t 的点，出度不超过 1 ，对于 t ，出度不超过 0 。我们求出三个拟阵的交 I ，如果 $|I| = n - 1$ ，那么我们就求得了一条哈密顿路径，否则哈密顿路径不存在。

5.6 应用

有了拟阵交之后，我们可以通过拟阵交结束很多组合模型。因为交这个操作对应逻辑上的同时满足若干条件，所以很多组合问题，我们可以将若干条件拆分开来，使得每一组单独只看这一个条件都是符合拟阵公理的条件。然后我们再把这几个拟阵交在一起，就可以解决原问题了。

我们来看几个例子：

例子 5.1 (二分图匹配)**.** 给定二分图 $G = (V, E)$ ，其中 $V = V_1 + V_2$ ，分别表示左部和右部，求一个最大的匹配。

二分图匹配问题如果我们将点集看成基础集，那么本身就是一个拟阵问题。然而加入一个点之后是否还是独立集这一问题不好直接判断，常规的方法是通过匈牙利算法解决。但如果我们定义这样两个拟阵， $M_1 = (E, \mathcal{I}_1)$ ，其中 $\mathcal{I}_1 = \{I : V_1\text{中的每个点度数不超过}1\}$ ，第二个拟阵类似的是 V_2 中的每个度数不超过 1 。二分图匹配就可以被看做两个拟阵的交。并且这两个拟阵判断是否独立都可以简单的 $O(1)$ 判断。

事实上，我们仔细分析一下就可以发现，匈牙利算法所找到的增广路，和拟阵的交找到的增广路是一样的。

例子 5.2 (最小树形图)**.** 给定带权有向图 $G = (V, E)$ ，求图 G 以 x 为根的最小树形图[60]。

最小树形图我们只要取两个拟阵，第一个限制将边看作无向图时无环，第二个限制除了 x 点之外每个点入度不超过 1 ,然后进行带权的拟阵交即可。

例子 5.3 (Colourful tree)**.** 给定带权无向图 $G(V, E)$ ，每条边拥有一个 1 到 $n - 1$ 中的颜色，求一个权最大的生成树，满足每种颜色只出现一次。

令 $M_1 = (E, \mathcal{I}_1)$ ，其中 $\mathcal{I}_1 = \{I : I\text{无环}\}$ ， $M_2 = (E, \mathcal{I}_2)$ ，其中 $\mathcal{I}_2 = \{I : I\text{中每种颜色出现不超过一次}\}$ 。求 M_1 和 M_2 的交即可。

[60] https://en.wikipedia.org/wiki/Arborescence_(graph_theory)

例子 5.4. Rainbow Graph[61]

给定一张带权无向图 $G = (V, E)$ ，每条边是三种颜色红绿蓝之一。要求对于从 1 到 $|E|$ 的每个 k ，都求出来选出 k 条边并使得不含红色边的图和不看蓝色边的图都是联通的的最小权值，如果不存在则输出 -1 。

$|V|, |E| \leq 100$ 。

选取一些边并使得图是联通的并不是一个合法的拟阵的条件，我们转而考虑删去哪些边。由于删去边后图联通等价于图拟阵的对偶，所以这是一个合法的拟阵条件。那么问题就很简单了，对于每个 k 我们只要考虑删去 $|E| - k$ 条之后，在两个图中都联通的方案，这显然是两个拟阵的交。至于要求每个 k 的答案，我们的拟阵交算法是从空集开始每步逐渐拓展一个元素的，自然可以得到每个 k 的答案。

6 拟阵的并

6.1 定义

定义 6.1. 对于给定的 k 个拟阵 $M_i = (S_i, \mathcal{I}_i), 1 \leq i \leq k$ 。定义这 k 个拟阵的并为 $M = (S, \mathcal{I})$ ，其中 $S = \bigcup_{i=1}^{k} S_i, \mathcal{I} = \{\bigcup_{i=1}^{k} I_i : I_i \in \mathcal{I}_i\}$ 。

上述定义我们不加证明的断言了若干拟阵的并是一个拟阵，之后我们将证明这个结论。首先我们考虑一下拟阵的并是怎么样一种概念。如果这 k 个拟阵的基础集互不相交，那么定义比较显然，新的拟阵的基础集就是原基础集的并，一个集合是独立集的条件当且仅当他在每个拟阵的部分都是独立集，这种情况也比较容易证明这是个拟阵。但如果基础集有相交的时候，情况不是那么显然。直接从组合意义解释的话，一个集合是独立集的条件当且仅当其存在一个拆分成 k 个子集的方法，使得这 k 个部分各自在对应的拟阵中有定义且是独立集。我们尝试通过推导出它的秩函数来证明它是一个拟阵。

首先我们不加证明的给出秩函数的定义。

定理 6.1. 对于给定的 k 个拟阵 M_i 的并的拟阵 M 的秩函数如下：

$$r_M(U) = \min_{T \subseteq U}(|U \backslash T| + \sum_{i=1}^{k} r_i(T \cap S_i))$$

为了证明这个秩函数的正确性，我们先要证明一个引理。

引理 6.2. 令 $\hat{M} = (\hat{S}, \hat{\mathcal{I}})$ 为一个任意的拟阵，且是 k 个基础集互不相交的拟阵 $\hat{M}_i = (\hat{S}_i, \hat{\mathcal{I}}_i)$ 的并。对于任意一个函数 $f : \hat{S} \to S$ ，我们定义拟阵 $M = (S, \mathcal{I})$ ，其中 $\mathcal{I} = \{f(\hat{I}) : \hat{I} \in \hat{\mathcal{I}}\}$ [62]。

[61] North American Invitational Programming Contest 2018 G

[62] $f(\hat{I})$ 表示 $\bigcup f(\hat{x}), x \in \hat{I}$

我们有 M 的秩函数如下[63]：

$$r_M(U) = \min_{T\subseteq U}(r_{\hat{M}}(f^{-1}(T)) + |U\backslash T|)$$

证明. 首先证明 M 是一个拟阵。考虑任一独立集 I，一定存在一些 $\hat{I} \in \hat{\mathcal{I}}$，满足 $I = f(\hat{I})$，并且 $|I| = |\hat{I}|$。由于 I 的每个子集都是 f 下 $\hat{I}$ 一些子集的像，I 的每个子集都是必定是独立的。

考虑一个独立集 $J = f(\hat{J}) \in \mathcal{I}$，存在一些 $\hat{J} \in \hat{\mathcal{I}}$，满足 $|\hat{I}| < |\hat{J}| = |J|$。这说明存在 $e \in \hat{J}\backslash\hat{I}$ 满足 $\hat{I} + \{e\} \in \hat{\mathcal{I}}$。假设 $\hat{I}$ 是从所有满足 $f(\hat{I}) = \hat{I}, |I| = |\hat{I}|$ 的独立集中选取出的，满足最大化 $|\hat{I} \cap \hat{J}|$ 的一个。我们将证明 $f(e) \notin I$，以此说明交换性的存在。假如 $f(e) \in I \cap J$，那么存在一个 $e' \in \hat{I}\backslash\hat{J}$，满足 $f(e') = f(e)$。那么 $\hat{I}' = \hat{I} + \{e\} - \{e'\}$ 也是独立集，那么我们就可以拓展 $|\hat{I} \cap \hat{J}|$ 的大小，这与假设不符。

为了证明秩函数的正确性，我们需要关于 S 的一个子集 U 定义一个划分拟阵，即 $(\hat{S}, \mathcal{I}_p)$，其中 $\mathcal{I}_p = \{I \subseteq \hat{S} : |f^{-1}(e) \cap I| \le 1, \forall e \in U, |f^{-1}(e) \cap I| = 0 \forall e \notin U\}$。考虑这拟阵的秩函数，$r_{M_p}(T) = |\{e \in U : f^{-1}(e) \cap T \neq \varnothing\}|$。那么 M 中 U 的一个子集是独立的，当且仅当存在一个子集 $\hat{I} \subseteq \hat{U} = f^{-1}(U)$ 同时在 M_p 和 $\hat{M}$ 中独立。由此可得 $\max_{I\subseteq U, I\in\mathcal{I}} |I| = \max_{\hat{I}\subseteq\hat{U}, \hat{I}\in\hat{\mathcal{I}}\cap\mathcal{I}_p} |\hat{I}|$。根据最小最大定理，可以得到引理成立。

根据引理证明定理不是一件困难的事情，我们只要令任意一个 S_i 中的元素 e 变为二元组 (i, e) 作为 $\hat{S}_i$ 的元素即可，映射 f 取 $f((i, e)) = e$ 即可，由于新构造的所有的 $\hat{S}_I$ 互不相同，所以运用引理 6.2 就可以证明。

6.2 独立集判定问题

虽然我们证明了拟阵的并还是一个拟阵，但是想要判定一个子集是否是独立集并不是一件容易的事情，即使在单独任意一个拟阵中都很容易判断。

判断一个元素是否是独立集有两种常用办法。

6.2.1 拟阵交

第一种方法是利用拟阵交算法。考虑构造 $\hat{S}_i = \{(e, i) : e \in S_i\}$，构造拟阵 $(\bigcup \hat{S}_i, \hat{\mathcal{I}})$，其中一个子集 U 是独立集当且仅当对于每个拟阵 M_i 都满足 $f^{-1}(\{e : (e, i) \in U\})$ 是 M_i 中的独立集。构造第二个拟阵 $M_p = (\hat{S}, \mathcal{I}_p)$，其中 $\mathcal{I}_p = \{I \subseteq \hat{S} : \forall e \in S, |I \cap \{(e, i) : e \in S_i\}| \le 1\}$。第二个拟阵限制了相同的元素只出现了一次，而第一个拟阵限制了每一个拟阵中都是单独独立的，我们求出要判断的集合的子集是否是这两个拟阵的交就可以判断是否是独立集了。

[63] $f^{-1}(e)$ 表示所有满足 $f(x) = e$ 的元素的集合

6.2.2 拟阵划分

我们考虑直接解决这个问题。假如存在一种划分算法，它满足输入一个 $M_1 \cup M_2 \cup \cdots \cup M_k$ 的独立集 I ，一个关于 I 的划分 $I = I_1 \cup I_2 \cup \cdots \cup I_k$ ，其中 $I_i \in \mathcal{I}_i$ ，和一个不属于 I 的元素 s ，判断 $I + \{s\}$ 是否还是独立集，并给出一个新的划分。那么我们就可以通过逐步加入元素来判断这个子集是否是独立集。

这个算法与拟阵的交的算法有些类似，我们定义关于拟阵 M_i 上独立集 I_i 的交换图 $D_{M_i}(I_i)$ 是一个有向二分图，左部是点集 I_i ，右部是 $S_i \backslash I_i$ ，一条从 $y \in I_i$ 指向 $x \in S_i \backslash I_i$ 的有向边存在，当且仅当 $I_i - \{y\} + \{x\}$ 是 M_i 中的独立集。

我们定义当前的交换图 D 是所有的 $D_{M_i}(I_i)$ 的并，对于每个拟阵定义 $F_i = \{x \in S_i \backslash I_i : I_i + \{x\} \in \mathcal{I}_i\}$ ，以此定义 $F = \bigcup F_i$ 。

下面我们将给出判断 $I + \{s\}$ 是否是独立集的定理，并在证明中给出一个算法流程。

定理 6.3. 对于任一元素 $s \in S \backslash I$ ， $I + \{s\} \in \mathcal{I}$ 当且仅当存在一条从 F 到 s 的有向路径。

证明. 我们首先证明条件的充分性。假设存在一条从 F 到 s 的有向路径，我们令 P 为其中最短的一条。假设 P 由 $\{s_0, s_1, \ldots .s_p\}$ 构成。不失一般性，我们令 $s_0 \in F_1$ 。对于 1 到 k 中的每一个 j 我们令 $S_j = \{s_i, s_{i+1} : (s_i, s_{i+1}) \in D_{M_j}(I_j)\}$ 。

我们令新的 $I'_1 = (I_1 \Delta S_1) \cup \{s_0\}$ ，其余的 j ，新的 $I'_j = I_j \Delta S_j$ 。由于除了端点，其余点都出现了两次，所以现在的划分就是多出了点 s 。现在我们只要证明每个 I'_j 依然独立即可。类似于拟阵交中的证明，由于我们选取了最短路，所以对于 > 1 的每个 j ，都满足在 $D_{M_j}(I_j)$ 中存在唯一的一个被删除的点和新加入的点的完美匹配，所以 $I'_j \in \mathcal{I}_j$ 。同样类似拟阵的交，由于 $s_0 \in F_1$ ，我们可以证明 I'_1 也是独立集。

考虑如何证明充分性，令 T 为所有可以到达 s 的点组成的点集，我们考虑反正，假设不存在路径，即 $F \cap T = \varnothing$ 。首先需要证明的是 $|I_i \cap T| = r(S_i \cap T)$ 。假设存在某个 i ，不符合这个要求，即 $|I_i \cap T| < r_i(S_i \cap T)$ 。那么说明存在一个 $x \in T \cap (S_i \backslash I - i)$,满足 $(I_i \cap T) + \{x\}$ 是独立集。但由于 $T \cap F = \varnothing$ ，所以 $x \notin F$ ，由此可能必然存在一个元素 $y \in I_i \backslash T$ ，满足 $I_i - \{y\} + \{x\}$ 是独立集，说明 $y \in T$ ，但这与 y 的选择不符。所以 $|I_i \cap T| = r_i(T \cap S_i)$ ，这说明 I_i 已经包含了 T 中的最大的独立集了。

注意到 $s \in T$ ，所以 $(I + \{s\}) \cap T = \bigcup (I_1 \cap T) + \{s\}$ 。根据拟阵并的秩函数，我们可以得到：

$$
\begin{aligned}
r_M(I + \{s\}) &\leq |(I + \{s\}) \backslash T| + \sum_{i=1}^{k} r_i(T \cap S_i) \\
&\leq |(I + \{s\}) \backslash T| + \sum_{i=1}^{k} r_i(T \cap I_i) = |I \backslash T| + \sum_{i=1}^{k} |I_i \cap T| = |I|
\end{aligned}
$$

由此可得， $I + \{s\}$ 并不是独立集，命题得证。

6.3 应用

如果我们特殊化拟阵的并，令 $M^k = M \cup M \cup \cdots \cup M$ ，即 k 个 M 的并。那么我们可以得到判断一个子集是否能够被划分成不超过 k 个独立集。

考虑 M^k 的秩函数：

$$r_{M^k} = \min_{T \subseteq U}(|U \backslash T| + kr_M(T))$$

注意到上式的最小值只会在满足 $cl(T) = T$ 时取得，否则我们可以通过扩大 T 来减小答案。

通过秩函数，我们可以得到一些简单的结论。

定理 6.4 (拟阵基覆盖定理)**.** 一个拟阵 M 可以被不超过 k 个基覆盖，当且仅当 $\forall T \subseteq S, |T| \leq kr_M(T)$ 。

证明. 直接带入秩函数可以得到这个结论。

定理 6.5 (拟阵基包含定理)**.** 一个拟阵 M 可以包含至少 k 个互不相交的基，当且仅当 $\forall T \subseteq S, |S \setminus T| \geq k(r_M(S) - r_M(T))$ 。

证明. 证明合并之后的拟阵的秩至少是 k 倍原拟阵的秩即可。

通过上述两个定理，可以解决很多推论问题，这里只举两个例子为例。

定理 6.6. 一张无向图 $G = (V, E)$ 可以被不超过 k 个生成树覆盖，当且仅当 $\forall U \subseteq V, |E(U)| \leq k(|U| - 1)$ 。

证明. 考虑如何证明正向，直接利用定理 6.4 是对于任意一个边集满足要求，考虑到边集根据连通性分成了很多联通块，如果不满足要求的话，必定存在一个联通块不符合要求，我们只考虑这一个联通块。在只考虑一个联通块的情况下，子集的秩一定是点数-1，且所有不改变秩的边全部加入是最坏情况，即 $E(U)$ ，由此我们只要保证 $E(U) \leq k(|U| - 1)$ 即可。反向证明类似。

定理 6.7. 一张有向图包含至少 k 个互不相交生成树，当且仅当对于原图的任意一个点集划分 $(V_1, V_2, \ldots, V_p)$ ，都满足 $\delta(V_1, V_2, \ldots, V_p) \geq k(p - 1)$ ，其中 $\delta(V_1, V_2, \ldots, V_p)$ 表示所有跨划分的边的数量。

证明. 证明类似上述证明。

例子 6.1 (二分图匹配)**.** 给定无向二分图 $G(V, E)$ ，其中 $V = V_1 + V_2$ ，分别表示左部和右部，求最大匹配。

之前我们已经用拟阵交的模型来解释过二分图匹配了，这里我们发现二分图匹配还可以通过拟阵并解释。令左部的点数为 n ，右部的点数为 m ,我们考虑建立 m 个拟阵，每个拟阵的基础集都是 V_1 ，第 i 个拟阵的独立集条件是只有一个点，且与右部的第 i 个点右边，则二分图匹配的答案就是拟阵并的秩。

我们仔细观察一下，发现求解二分图匹配的匈牙利算法，和拟阵并算法也是类似的。

例子 6.2. 元旦老人与丛林[64]

给定一张 n 个点， m 条边的有向图 G ，判断是否存在一种选出边的方案，使得无论保留你选择的边还是你没选择的边，图都无环。

$n \le 2000, m \le 4000$ 。

注意到本题的条件等价于是否存在不超过两个生成树就能覆盖全图的情况，所以我们只要判断是否对于每个点集 U ，都满足 $E(U) \le 2|U| - 2$ 即可。这个问题是经典的网络流问题，可以通过最大闭合权子图解决。本题由于会出现负答案的情况，所以需要强制选择一个点，需要使用退流优化复杂度，但不是本文讨论的重点，这里不做赘述。

7 拟阵的可表示性

关于拟阵还有很多重要的内容，由于作者水平有限，这里只能讲述一个。

7.1 定义

定义 7.1. 对于拟阵 $M = (S, \mathcal{I})$ ，定义 M 在域 $\mathbb{F}$ 上可表示，当且仅当存在一个定义在域 $\mathbb{F}$ 上的线性拟阵与 M 同构。

拟阵可表示等价于存在一个映射 $f : S \to \mathbb{F}^k$ ，使得一个集合是独立集当且仅当他们对应的向量线性无关。

我们来看一个简单的例子。

例子 7.1. 均匀拟阵 U_4^2 在域 $GF(2)$ [65]上是不可表示的。

我们需要构造四个 $GF(2)$ 上的向量，使得他们两两线性无关，且任意三个都是线性相关的。由于拟阵的秩是 2 ，所以构造出的拟阵组成的矩阵的维度也必定是 2 。然而考虑到 $GF(2)$ 上的二维向量只有 4 个，而向量 $(0, 0)$ 一定是线性相关的，所以不存在 4 个向量，所以不可表示。

[64] Universal Online Judge Round #11 B,http://uoj.ac/contest/23/problem/168

[65] 只有两个元素的域是平凡的

7.2 正则拟阵

定义 7.2. 定义一个可以表示在 $GF(2)$ 上的拟阵是二元拟阵，定义一个可以表示在任意域上的拟阵是正则拟阵。

定理 7.1. 图拟阵是正则拟阵。

证明. 对于图 $G=(V,E)$，考虑构造一个 $|E|\times|V|$ 的矩阵 A，其中一行表示一条边，也表示一个向量。假设第 i 条边是 (u,v)，则 $A_{i,u}=1, A_{i,v}=-1$。

如此构造之后，如果存在一个环，我们可以通过调整正负形，使得一个点刚好是一个 1 一个 -1，必然是线性相关的。

这里的构造考虑的是 $\mathbb{Z}$ 中的构造，但对于任意的域，我们只要取其中的乘法单位元，和乘法单位元的加法逆元分别对应 1 和 -1 即可[66]。

通过研究拟阵的可表示性，可以提供一个分类拟阵的方法。而正则拟阵是其中一类性质很强的拟阵，而图拟阵是正则拟阵的一部分，这说明图拟阵在除了拥有拟阵的一般性之外，有着更强的特殊性。

在研究中作者发现了任意的正则拟阵的基的计数都可以通过计算一个关联矩阵的行列式求得，而矩阵树定理是其中的特例。但作者没有找到合适的资料，如果有同学得出了一些结论欢迎作者讨论。

7.3 一些结论

极小元是表示一个拟阵特征的重要结构，在可表示性中，很多拟阵是否可表示可以用极小元来判断，在研究过程中我发现了一些比较重要的结论。

定理 7.2. 一个拟阵是二元拟阵，当且仅当不存在极小元是 U_4^2。

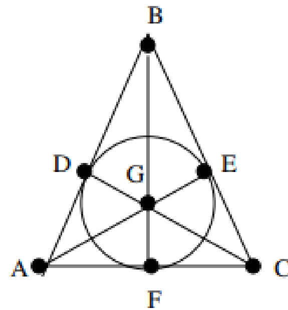
定理 7.3. 一个拟阵可以在 $GF(3)$ 上被表示，当且仅当不存在极小元是 U_5^2, U_5^3 和 F_7, F_7^*。其中 F_7 表示一个三阶的大小为 7 的法诺拟阵。

法诺拟阵是如下的一个图，他的独立集是所有的大小为 3 且不共线的点集。

8 总结

拟阵是组合优化中的强力工具，它抽象了一类组合优化问题的共同性质，研究了“独立”这一概念所推出的诸多性质。在引入了拟阵交拟阵并之后，有很多经典的组合优化问题都可以通过拟阵来解决。

[66]特征为 2 的域中，1 和 -1 是一个元素，但这并不影响构造



拟阵理论博大精深，但作者水平万分有限，因此希望本文能够起到抛砖引玉的作用，能让更多的同学感受到拟阵的魅力，希望感兴趣的读者，能够进一步研究。

9 致谢

感谢中国计算机学会提供交流和学习的平台。
感谢国家集训队教练张瑞喆的指导。
感谢薛志坚老师和学长多年对我的指导和帮助。
感谢王修涵同学对我的启发和关于本文的帮助。
感谢王修涵同学、钟子谦同学、吴航学长和张晴川学长为本文验稿。

参考文献

[1] 刘雨辰，《对拟阵的初步研究》，IOI2007 中国国家队候选队员论文

[2] W. T. Tutte. A homotopy theorem for matroids, i, ii. Trans. Amer. Math Soc., 88:144–174, 1958.

[3] J.G. Oxley. Matroid Theory. Oxford University Press, 1992.

[4] H. Whitney. On the abstract properties of linear dependence. Amer. J. Math., 57:509–533, 1935.

[5] P. Seymour. Matroid representation over GF(3). J. Combin. Theory Ser. B, 26:159–173, 1979.

[6] A. Schrijver. Combinatorial Optimization: Polyhedra and Efficiency, volume B. Springer, 2003.

浅谈Splay与Treap的性质及其应用

广州市第六中学 董炜隽

摘 要

Splay和Treap是两种常用的平衡树。本文先简要介绍了这两种平衡树的基础操作，之后着重地分析了Splay与Treap上的finger search以及平衡树带权的情况，并展示它们的一些实际应用。

引言

平衡树被引入OI界已有将近二十年的时间。在这些年里，许多平衡树都已经得到了广泛的应用。其中，Splay和Treap都兼具良好的泛用性以及较小的代码难度，因此不少选手都已经掌握了它们的基础操作。

实际上，Splay和Treap还拥有一些特殊的性质，这使得它们具有比一般的平衡树更为强大的功能。例如，Splay与Treap的启发式合并都可以做到一个 log 的时间复杂度，两者都可以支持带权的情况，等等。目前OI界对这部分内容的介绍并不多，部分选手对其有一定了解，但往往也不知道如何证明其时间复杂度。因此，本文将会详细地介绍Splay与Treap的这些性质，并且对大部分的结论给出证明，希望能帮助填补这一方面的空白，同时也向更多的选手推荐这两种优秀的平衡树。

本文的前两节首先会对Splay和Treap的基础操作进行介绍，为后续的内容做铺垫。之后在第三节中，本文会介绍如何在Splay和Treap上实现finger search，并展示它们的一些具体应用。最后在第四节中，本文将讨论Splay与Treap带权的情况，并考虑将其应用在一类树的多层嵌套问题上。

1 Splay

Splay（伸展树）是一种实用的平衡树。

在一般的二叉查找树中，操作的耗时与节点的深度有关。假如对一个深度较大的节点进行很多次操作，那么时间上的开销就会变得非常大。一个自然的想法是，在每次对某个节点进行操作后就将其旋转至根，那么下次访问该节点的耗时就非常小了。然而直接把该

节点从下往上旋转到根并不可行，我们可以轻易地构造出数据将其卡掉。而Splay就使用了一种特殊的旋转策略，从而保证了操作的时间复杂度。

1.1 splay操作

当访问Splay中的某个节点 x 时，我们需要对 x 进行一次splay操作，将其移动到根节点。splay操作的过程中共有三种情况：

1. 当 x 的父节点为根时，直接上旋 x 即可。
2. 当 x 及其父节点均为左儿子，或 x 及其父节点均为右儿子时，进行一次Zig-Zig操作或者Zag-Zag操作：先上旋 x 的父节点，再上旋节点 x 。
3. 当 x 及其父节点其中一个为左儿子，而另一个为右儿子时，进行一次Zig-Zag操作或者Zag-Zig操作：直接上旋两次节点 x 。

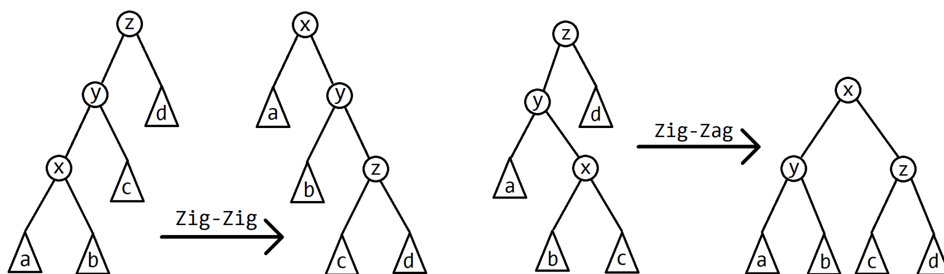


图 2: splay操作

可以发现，进行一次splay操作后，在splay路径上的节点的深度都会减少约一半，这使得Splay具有良好的自调整的性质。有了感性的认识之后，我们再来严谨地证明一下splay操作的时间复杂度。

定理 1.1. 在一棵 n 个节点的 Splay 中，对一个节点进行 splay 操作的时间复杂度为均摊 $O(\log n)$ 。

证明. 设 $size(x)$ 表示节点 x 的子树大小。考虑势能分析，定义节点 x 的势能为 $r(x) = \log size(x)$ ，整棵树的势能为所有节点的势能之和。令节点 y 为节点 x 的父节点，节点 z 为节点 y 的父节点（如果存在的话）。

1. 当 x 的父节点为根时，我们对其进行了一次上旋操作，势能的增量为 $r'(x) + r'(y) - r(x) - r(y) = r'(x) - r(x)$ 。

2. 对于 Zig - Zig 或者 Zag - Zag 的情况，势能的增量为

$$\begin{aligned}&r'(x)+r'(y)+r'(z)-r(x)-r(y)-r(z)\\=&r'(y)+r'(z)-r(x)-r(y)\\\leq&r'(x)+r'(z)-2r(x)\\\leq&3(r'(x)-r(x))-2\end{aligned}$$

   最后一步的时候我们给式子加上了 $2r'(x)-r(x)-r'(z)-2$ 。注意到 $size'(x)=size(x)+size'(z)+1$ ，因此 $2r'(x)-r(x)-r'(z)\geq 2$ 。

3. 对于 Zig - Zag 或者 Zag - Zig 的情况，我们可以类似地证明势能的增量不超过 $2(r'(x)-r(x))-2$ 。

综上，一次进行了 k 次旋转的 splay 操作后，势能的增量不超过 $3(r'(x)-r(x))-k+1$ 。整棵树初始的势能为 $O(n\log n)$ ，且势能任意时刻均非负，因此对一棵 n 个节点的 Splay 进行 m 次 splay 操作的时间复杂度就是 $O((n+m)\log n)$ 。也就是说 splay 操作的时间复杂度为均摊 $O(\log n)$ 。

1.2 各项操作

我们可以借由splay操作来完成插入、删除操作。具体的实现方式有很多种，这里仅提供一种参考的写法：在插入一个元素时，先把它插入到叶子上，之后将其splay到根。删除一个元素时，先把它splay到根并删掉，再用下面介绍的连接两棵Splay的方法将其左右子树拼接起来即可。

Splay还可以应用在维护序列的问题上。在这类问题中，每棵Splay表示的是其中序遍历对应的序列，并且通常会需要用到以下两种操作：

- 把两棵Splay T_1,T_2 连接为一棵Splay T 。假设 T_1,T_2 表示的序列分别为 $a_1,a_2,\dots,a_n$ 和 $b_1,b_2,\dots,b_m$ ，那么 T 表示的序列应为$a_1,a_2,\dots,a_n,b_1,b_2,\dots,b_m$ 。
- 把一棵Splay T 分裂为两棵Splay T_1,T_2 。假设 T 表示的序列为 $a_1,a_2,\dots,a_n$ ，那么 T_1,T_2 表示的序列应分别为 $a_1,a_2,\dots,a_k$ 和$a_{k+1},a_{k+2},\dots,a_n$ 。

这两种操作同样可以用splay操作完成。连接两棵Splay时，先把 T_1 的最后一个元素splay至根，然后把 T_2 作为其右子树即可。分裂操作时，把第 k 个节点splay到根，切断它与右儿子的边就能得到所需的两棵Splay了。

在这些操作的过程中，除了splay操作以外的步骤造成的势能变化量都是 $O(\log n)$ 的，因此以上操作的时间复杂度均为均摊 $O(\log n)$ 。

需要注意的是，只要遍历了Splay的根节点到某个节点的路径，我们就必须对该节点进行一次splay操作，否则就无法保证时间复杂度。

2 Treap

Treap是一种常用的随机二叉搜索树。顾名思义，Treap就是树+堆。Treap中的每个节点都被分配了一个随机的优先级，它既满足二叉搜索树的性质，同时节点的优先级又满足堆的性质。在下文的叙述中，我们默认Treap中节点的优先级满足大根堆的性质。

2.1 节点的深度

在引入了随机优先级后，Treap就相当于以随机顺序插入得到的二叉搜索树，因此其形态会趋于平衡。下面我们来严谨地证明节点深度的渐进上界。

定理 2.1. 在一棵 n 个节点的 Treap 中，节点的深度为期望 $O(\log n)$ 。

证明. 考虑每个节点 i 在什么情况下会成为节点 x 的祖先。当 i 为 x 的祖先时，说明在 i 到 x 之间的节点的优先级都小于 i 。因此节点 x 的期望深度为

$$\begin{aligned}&1+\sum_{i=1}^{x-1}\frac{1}{x-i}+\sum_{i=x+1}^{n}\frac{1}{i-x}\\=&1+\sum_{i=1}^{x-1}\frac{1}{i}+\sum_{i=1}^{n-x}\frac{1}{i}\\=&O(\log n)\end{aligned}$$

注意在此处及后文的证明中，我们默认节点的优先级互不相同。

2.2 插入与删除

插入一个节点时，我们可以先把该节点插入到一个叶子上，之后再对其进行若干次上旋以维护堆的性质。

删除是插入的逆操作，因此可以直接把要删除的节点下旋至叶节点处，之后直接将其删除。

插入和删除的耗时都不会超过节点的深度，即期望 $O(\log n)$ 。

2.3 连接与分裂

Treap同样可以应用在维护序列的问题中，这时我们可能需要把两棵Treap连接为一棵Treap，或者把一棵Treap分裂为两棵Treap。下面给出连接操作及分裂操作的伪代码：

Algorithm 1 Treap-Join

```
1: function Join(T_1, T_2)
2:     if T_1 = NULL then
```

3: **return** T_2
4: **end if**
5: **if** $T_2 = NULL$ **then**
6: **return** T_1
7: **end if**
8: **if** $T_1 \to priority > T_2 \to priority$ **then**
9: $T_1 \to rchild \leftarrow$ Join $(T_1 \to rchild, T_2)$
10: **return** T_1
11: **else**
12: $T_2 \to lchild \leftarrow$ Join $(T_1, T_2 \to lchild)$
13: **return** T_2
14: **end if**
15: **end function**

Algorithm 2 Treap-Split

1: **procedure** Split($T, k, \&T_1, \&T_2$)
2: **if** $T \to maxkey \leq k$ **then**
3: $[T_1, T_2] \leftarrow [T, NULL]$
4: **return**
5: **end if**
6: **if** $T \to minkey > k$ **then**
7: $[T_1, T_2] \leftarrow [NULL, T]$
8: **return**
9: **end if**
10: **if** $T \to key \leq k$ **then**
11: $T_1 \leftarrow T$
12: Split($T \to rchild, k, T_1 \to rchild, T_2$)
13: **else**
14: $T_2 \leftarrow T$
15: Split($T \to lchild, k, T_1, T_2 \to lchild$)
16: **end if**
17: **end procedure**

可以发现连接/分裂操作的耗时不会超过两棵树的深度之和。假设连接前/分裂后的两棵Treap大小分别为 n, m ，那么该操作的时间复杂度为期望 $O(\log n + \log m)$ 。

实际上我们也可以仅通过旋转操作来实现连接或分裂操作。连接两棵Treap时，先新建一个辅助节点，把两棵Treap作为它的左右子树，然后把该节点下旋至叶子处删掉；分裂两棵Treap时，先在分裂点处插入一个辅助节点，之后将其上旋至根，此时它的左右子树就是

分裂后的两棵Treap了。可以发现这种实现方式与之前的做法本质上是相同的，因此时间复杂度同样为期望 $O(\log n + \log m)$ 。

2.4 重量平衡

Treap在插入或删除一个元素时树的整体形态变化很小，旋转的次数为期望 $O(1)$ ，且旋转的子树大小为期望 $O(\log n)$ 。

陈立杰学长已经在其集训队论文中详细介绍过这部分内容[67]，这里不再赘述。

2.5 可持久化

Treap的时间复杂度是期望的，因此可以直接可持久化。

需要注意的是，假如在可持久化的过程中涉及到复制一棵Treap，复制出来的Treap的随机优先级会与原本的Treap完全一样，这可能会导致后续操作的Treap不再平衡。

目前OI界中常用的解决方法是，不存储节点的随机优先级，而是在需要比较优先级时再依据节点的子树大小随机地返回结果。例如，当连接两棵Treap T_1, T_2 时， T_1 的优先级大于 T_2 的优先级的概率为 $\frac{size(T_1)}{size(T_1)+size(T_2)}$ ，那么直接按照这个概率返回优先级的比较结果即可。

3 Finger Search

一般情况下，在数据结构中查找某个元素的时间复杂度会表示成关于 n （即数据结构中的元素个数）的函数。设 $d(x, y)$ 表示两个元素 x, y 在集合中的排名之差。在一些数据结构中，如果查找元素 y 时从节点 x 开始查找，那么时间复杂度就可以优化为关于 $d(x, y)$ 的函数。finger search指的就是在数据结构中从某些特定的节点（也就是“finger”）开始查找其他元素。假如被查找的元素与“finger”的排名距离很近，那么查询的效率就会有很大的提升。

例如在一个有序数组中，我们可以通过倍增来实现finger search，时间复杂度为 $O(\log(d(x, y)+1))$ 。（此处以及下文的一些地方将 $d(x, y) + 1$ 是为了避免 $\log 0$ 的情况）

3.1 Splay上的Finger Search

Splay是一种原生地支持finger search的数据结构。我们可以把根节点视为一个“finger”，那么Splay的所有操作其实都是用finger search完成的。

这里给出Splay上进行finger search的时间复杂度：

[67]详见参考文献[5]

定理 3.1 (Dynamic Finger Theorem)**.** 在一棵 n 个节点上的 Splay 上进行 m 次的插入、删除或者查找操作的总时间为 $O\left(n + m + \sum_{i=1}^{m} \log(d_i + 1)\right)$ 。其中 d_i 表示第 i 次操作的元素与第 $i-1$ 次操作的元素在该操作时的排名之差。第 0 次操作的元素可以视为初始的根节点。

该定理的证明较为复杂。限于水平，笔者暂时还未能读懂其证明。感兴趣的读者可以自行查阅参考文献[1]。

3.2 Treap上的Finger Search

定理 3.2. 两个元素 x, y 在 Treap 上的路径长度是期望 $O(\log(d(x, y) + 1))$ 的。

证明. 先假设 $x < y$ 。考虑每个节点 $i(i \neq x, y)$ 成为 x 的祖先且不是 y 的祖先的概率，那么显然有 $i < y$ 。

- 若 $i < x$ ，那么节点 i 的优先级一定大于节点 $i+1..x$ ，并且 $x+1..y$ 中至少有一个节点的优先级大于节点 i ；
- 若 $x < i < y$ ，那么节点 i 的优先级一定大于节点 $x..i-1$ ，并且 $i+1..y$ 中至少有一个节点的优先级大于节点 i 。

那么节点 x 到 x, y 的最近公共祖先的期望路径长度就是

$$
\begin{aligned}
& O(1) + \sum_{i=1}^{x-1} \left(\frac{1}{x-i} - \frac{1}{y-i} \right) + \sum_{i=x+1}^{y-1} \left(\frac{1}{i-x} - \frac{1}{y-x} \right) \\
=& O(1) + \sum_{i=1}^{x-1} \frac{1}{i} - \left(\sum_{i=1}^{y-1} \frac{1}{i} - \sum_{i=1}^{y-x} \frac{1}{i} \right) + \sum_{i=1}^{y-x-1} \frac{1}{i} - \frac{y-x-1}{y-x} \\
\leq& O(1) + \sum_{i=1}^{y-x} \frac{1}{i} + \sum_{i=1}^{y-x-1} \frac{1}{i} \\
=& O(\log(y-x))
\end{aligned}
$$

节点 y 到 x, y 的最近公共祖先的期望路径长度同理。那么我们就证明了该定理。

因此，在Treap上实现finger search时，只需要从给定的节点 x 往上走到 x, y 的最近公共祖先处，然后再往下查找节点 y 。这个过程中需要判断当前节点是否为 x, y 的最近公共祖先，可以像线段树一样维护出每个节点表示的区间，若当前节点表示的区间已经包含了 y 时就不需要再向上走了。

3.2.1 Treap的快速连接

本文第二节中介绍了如何在期望 $O(\log n + \log m)$ 的时间内对Treap进行连接或者分裂操作。实际上，我们可以利用finger search的思想将复杂度进一步地优化到期望 $O(\log\min(n, m))$ 。

可以发现，连接两棵Treap T_1, T_2 的时候，其实就是自顶向下地把 T_1 的右链和 T_2 的左链按照优先级归并了起来。然而在极端情况下，例如， T_1 是一棵很大的Treap， T_2 则仅有一个节点，那么 T_2 的那个节点很大概率会被放到接近底层的位置，然而我们却需要遍历 T_1 的几乎整条右链，这就导致了很多冗余的代价。

我们不妨改为自底向上地归并 T_1 的右链和 T_2 的左链，当归并完其中一棵树的根节点时就结束这个过程。假设我们先归并到的是 T_1 的根节点，那么这个过程就相当于在新树中遍历了T_1 的最后一个节点或者 T_2 的第一个节点到 T_1 的根节点的路径。因此这样连接两棵Treap的复杂度就是期望 $O(\log\min(n, m))$ 。

3.2.2 Treap的快速分裂

可以发现，把一棵Treap分裂成 T_1, T_2 的过程其实就是遍历了根节点到 T_1 的最后一个节点 y 的路径，然后把路径上的节点分别划分到了 T_1, T_2 中。当 T_1, T_2 的其中一棵包含的元素很多而另一棵包含的元素很少时，这条路径顶部的很长一段都会被划分到较大的那棵Treap中。因此我们需要尽可能地避免遍历这段路径。

我们假设 T_1 中包含的元素个数少于 T_2 ，另一种情况同理。考虑从 T_1 的第一个节点 x 开始进行一次finger search，找到它与节点 y 的最近公共祖先 z 。显然在 z 的子树外的节点都会被划分到 T_2 中，这时再从节点 z 开始进行一次普通的分裂操作即可。整个过程只遍历了节点 x 到节点 y 的路径，因此时间复杂度就被优化到了期望 $O(\log\min(n, m))$ 。

有时候在分裂Treap之前可能无法知道 T_1 与 T_2 的大小。我们可以同时从第一个节点和最后一个节点开始进行finger search，直到其中一侧找到与节点 y 的最近公共祖先为止。时间复杂度仍然不变。

3.3 应用

3.3.1 启发式合并

合并平衡树时通常采用启发式合并的策略，也就是把较小的平衡树的每个元素暴力插入到较大的平衡树中。每个元素被插入到另一个平衡树中时，它所处的平衡树的大小至少增加一倍，那么每个元素至多只会被插入 $O(\log n)$ 次。而在平衡树中插入一个元素的复杂度通常是 $O(\log n)$ ，因此普通的平衡树启发式合并的时间复杂度就是 $O(n\log^2 n)$ 。

考虑使用finger search来优化启发式合并：

对于Splay，只需要把较小的一棵Splay中的元素按升序/降序的顺序中插入到另一棵Splay中。

对于Treap，同样把较小的一棵Treap中的元素按升序/降序的顺序插入到另一棵Treap中，每次插入时从上一个插入的位置开始进行一次finger search，找到新元素应插入的位置。具体实现时可以自顶向下地实现，这样就不需要记录每个节点的父指针。文献[7]中还介绍了另外一种合并Treap的方式，感兴趣的读者可以自行查阅。

假设合并的两棵平衡树大小分别为 $n, m(n \geq m)$ 。那么这样合并时，除了第一次插入的耗时为 $O(\log n)$ 外，其余插入的总耗时为 $O\left(\sum_{i=2}^{m} \log(d_i + 1)\right)$ ，并且满足$\sum_{i=2}^{m} d_i \leq n$ 。由于 log 是凸函数，最坏情况为所有 d_i 相等，即 $d_i = \frac{n}{m}$ 。因此一次合并的复杂度就是 $O\left(m \log \frac{n}{m}\right)$ 。实际上这也达到了理论下界，因为合并两个集合时要区分 $\binom{n+m}{n}$ 种情况，那么就至少需要 $\left\lceil \log \binom{n+m}{n} \right\rceil$ 次比较。

接下来考虑合并的总时间复杂度。

定理 3.3. 设使用 finger search 以任意顺序将若干棵 Splay 或者 Treap 合并成了一棵包含 n 个元素的树，那么合并的总时间为 $O(n \log n)$ 。

证明. 分别考虑每个元素的贡献，设其在合并的过程中所处的集合大小依次为 $s_1, s_2, s_3, ..., s_k$ ，那么其贡献为 $O\left(\sum_{i=2}^{k} \log \frac{s_i}{s_{i-1}}\right) = O(\log n)$ 。

根据定理3.1，Splay合并的复杂度里还有一个初始的 n 。不过我们可以把在每个Splay中进行的所有插入视为一个连续的过程，因此并不会影响最终的复杂度。

例 1. Tree[68]

有一棵 n 个节点的树，给出每条边的长度，问有多少个点对的距离不超过 k 。

$n \leq 10000$ ，边权为不超过 1000 的正整数。

此题的一个经典解法是点分治，但与本文主题无关，在此不再赘述。

考虑任意选择一个节点作为根开始dfs，并对每个节点维护一棵平衡树表示它到子树内每个节点的距离。那么就需要在每次合并两棵平衡树之前计算它们之间有多少个元素对的和小于 k ，同样可以用finger search完成。时间复杂度为 $O(n \log n)$ 。

3.3.2 维护序列

在一般的维护序列问题中使用Treap的快速连接与分裂并不能带来很好的优化效果，但是在一些特殊的情况下却可以优化问题的时间复杂度。

例 2. tree[69]

通过交互的方式，要求用二叉树维护若干个序列。

[68]POJ1741

[69]2018年候选队互测第五场

初始时有 n 个序列，每个序列仅有一个元素。接下来会发生 m 个事件，事件有如下三种类型：

- (join)把两个序列拼接成一个序列。
- (split)把某个序列拆成两个序列。
- (visit)询问某个序列的信息和。

交互库会对每棵树提供两个指针，分别指向树上的某个节点，需要通过对指针进行操作来维护这些事件。允许修改某个指针指向节点的儿子、更新某个指针指向节点的子树信息和以及把某个指针移动到相邻节点。交互库把这三种操作封装在了move函数中，并限制调用该函数的总次数不能超过 $2 * 10^6$ 次，在每次事件中调用该函数的次数不能超过 $maxcnt$ 。另外还限制查询的二叉树的最大深度不能超过 $maxdep$ 。

$1 \le n, m \le 2{*}10^5$ ， $maxdep \ge 60$ ， $maxcnt \ge 250$ ， $k \le 20000$ ，其中 k 表示split和visit事件的总个数。

考虑使用Treap来维护这些序列。注意到join事件的个数远高于其他事件，因此可以使用前面介绍的Treap的快速连接来优化这部分的效率。然而为了保持复杂度不变，合并后就不能再往上更新节点的子树信息和了。解决方法也很简单，显然每棵树未更新的节点一定都在左链或右链顶部的一段上，那么我们可以直接延迟到下次split或者visit事件时再更新。

我们来分析一下整个做法的复杂度。定义一个长度为 l 的序列的势能函数为 $\log l$ ，那么拼接两个长度分别为 $l1, l2$ 的序列时势能的增量为

$$\begin{aligned}&\log(l1 + l2) - \log l1 - \log l2\\=&\log(l1 + l2) - \log\max(l1, l2) - \log\min(l1, l2)\\\le&1 - \log\min(l1, l2)\end{aligned}$$

初始时总势能为0，且最终总势能非负，因此只有join事件时复杂度是 $O(n)$ 的。而一次split事件只会额外增加 $O(\log n)$ 的势能。那么，这个算法的时间复杂度就是 $O(n + m + k\log n)$ 。

类似地，如果本题改成只有split事件的话，我们也可以使用Treap的快速分裂来做到线性的复杂度。

另外，当不限制 $maxdep$ 和 $maxcnt$ 时，本题也可以用Splay来完成。设两棵Splay的大小分别为 a, b ，如果沿用普通的Splay的势能函数的话，分析出来连接操作的复杂度会是 $O(\log(a + b))$ 的。考虑对其稍作修改，定义Splay的势能函数为除根节点外所有节点 x 的 $\log size(x)$ 的和。连接两棵Splay时先把较小的一棵Splay的端点旋转到根，这里会使势能增加 $O(\log\min(a, b))$ ；之后把另一棵Splay 接到这个点下面，势能的增量同样是 $O(\log\min(a, b))$ 的。这样我们就证明了使用Splay的时间复杂度同样是 $O(n + m + k\log n)$ 的。

4 带权的二叉查找树

在一些情况下，我们会给二叉查找树的每个节点附上一个权重 w_i（也可以视为访问的频率），那么就需要尽可能地降低访问权重较大的节点时的代价。

4.1 静态的情况

先来考虑静态的情况。设 W 表示所有元素的权重之和。我们可以在建树时，找到最小的 k 使得 $\sum_{i=1}^{k} w_i > \frac{W}{2}$ ，把节点 k 作为根，接着再递归两侧进行建树。

定理 4.1. 在上述算法建出的树中，节点 x 的深度为 $O\left(\log \frac{W}{w_x}\right)$ 。

证明. 显然从任意节点向下走到一个儿子时子树内的权重和至少减半。因此从根节点向下走至多 $\left\lceil \log \frac{W}{w_x} \right\rceil$ 步就能到达节点 x 。

那么建树的时间复杂度为 $O(n \log W)$ 。

4.2 带权的Splay

Splay优秀的自调整性质使其可以直接支持带权的情况，我们甚至不需要知道每个节点对应的权重。在带权的Splay中，对某个节点进行splay操作的时间复杂度同样与其权重有关：

定理 4.2. 在一棵权重之和为 W 的 Splay 中，对节点 x 进行 splay 操作的均摊时间复杂度为 $O\left(\log \frac{W}{w_x}\right)$ 。

证明. 把 $size(x)$ 重新定义为节点 x 的子树内所有节点的权重之和，之后套用定理1.1的证明即可。

4.3 带权的Treap

我们可以对Treap做出一些修改，使其适用于带权的情况。我们希望权重较大的节点能处在更靠近根的位置，那么就需要给权重较大的节点分配一个较大的随机优先级。

4.3.1 分配节点的随机优先级

对于一个权重为 w 的节点，我们可以随机出 w 个数，并将这 w 个数的最大值作为该节点的优先级。节点的权重可能会经常改变，因此直接 $O(w)$ 计算一次优先级显然是无法接受的，我们需要找到一个更快的计算方法。

假设随机数是在 $[0, 1]$ 中均匀产生。考虑找出一个函数 f ，每次需要计算优先级时，我们就在 $[0, 1]$ 之间均匀地选取一个随机数 x ，然后把 $f(x)$ 作为该节点的优先级。原本的

方法中 w 个随机数的最大值小于 k 的概率为 k^w ，假设 $f(x)$ 单调递增，那么 k^w 同时也是 $x < f^{-1}(k)$ 的概率[70]。因此我们可以选择 $f(x) = x^{\frac{1}{w}}$ 。

考虑到优先级只需要比较大小，我们也可以直接记录 $\frac{\log(x)}{w}$ 。

4.3.2 节点的深度

Treap中节点的深度也有类似的性质：

定理 4.3. 在一棵权重之和为 W 的 Treap 中，节点 x 的期望深度为 $O\left(\log \frac{W}{w_x}\right)$ 。

证明. 考虑每个节点 $i(i \neq x)$ 成为节点 x 的祖先的概率，记 $sum(l, r) = \sum_{i=l}^{r} w_i$ ，那么节点 x 的期望深度为

$$\begin{aligned}
&1 + \sum_{i=1}^{x-1} \frac{w_i}{sum(i, x)} + \sum_{i=x+1}^{n} \frac{w_i}{sum(x, i)} \\
\leq& 1 + \sum_{i=1}^{x-1} \sum_{j=1}^{w_i} \frac{1}{sum(i+1, x) + j} + \sum_{i=x+1}^{n} \sum_{j=1}^{w_i} \frac{1}{sum(x, i-1) + j} \\
=& 1 + \sum_{i=w_x+1}^{sum(1,x)} \frac{1}{i} + \sum_{i=w_x+1}^{sum(x,n)} \frac{1}{i} \\
=& O\left(\log \frac{W}{w_x}\right)
\end{aligned}$$

Treap的各项操作的时间复杂度都可以表示为某些节点的深度，此处不再一一分析。

4.3.3 修改权重

当一个节点的权重被修改时，我们可以给它重新分配一个随机优先级。此时优先级可能不满足堆的性质，那么就需要对该节点进行一些上旋或下旋操作。可以发现旋转的次数其实就是修改前后该节点的深度之差。然而之前计算的是节点的深度上界，并不能直接以此得到深度差的上界。因此我们需要单独计算这部分的时间复杂度：

定理 4.4. 若节点 x 的权重由 w_x 修改为 w'_x ，那么修改前后该节点的深度之差为 $O\left(\log \frac{\max(w_x, w'_x)}{\min(w_x, w'_x)}\right)$ 。

证明. 令 $\triangle = w'_x - w_x$ ，这里仅证明 $\triangle > 0$ 的情况， $\triangle < 0$ 的情况同理。记 $sum(l, r) = \sum_{i=l}^{r} w_i$ ，那么深度差的期望为

$$\sum_{i=1}^{x-1} \left(\frac{w_i}{sum(i, x)} - \frac{w_i}{sum(i, x) + \triangle} \right) + \sum_{i=x+1}^{n} \left(\frac{w_i}{sum(x, i)} - \frac{w_i}{sum(x, i) + \triangle} \right)$$

[70] f^{-1} 表示 f 的反函数

考虑式子左半部分的上界：

$$\begin{aligned}&\sum_{i=1}^{x-1}\left(\frac{w_i}{sum(i,x)}-\frac{w_i}{sum(i,x)+\triangle}\right)\\ \le&\sum_{i=1}^{x-1}\sum_{j=1}^{w_i}\left(\frac{1}{sum(i+1,x)+j}-\frac{1}{sum(i+1,x)+\triangle+j}\right)\\ =&\sum_{i=w_x+1}^{sum(1,x)}\frac{1}{i}-\sum_{i=w_x+\triangle+1}^{sum(1,x)+\triangle}\frac{1}{i}\\ =&O\left(\log\frac{w'_x}{w_x}\right)\end{aligned}$$

同理可得式子右半部分也是 $O\left(\log\frac{w'_x}{w_x}\right)$ 的。证毕。

4.4 一类树的多层嵌套问题

在常见的树链剖分题目中，我们往往需要用线段树或者其他数据结构来维护每条重链的信息。考虑把每条重链的线段树视为套在该重链顶端的父节点对应的线段树节点下面，那么树链剖分+线段树就可以看作是一个 $O(\log n)$ 层的树套树套树...套树的结构。类似地，Link-Cut Tree、点分治+线段树也可以视为这样的一类树的多层嵌套问题。

在这类问题中，当需要访问某个节点时，我们通常需要遍历顶层根节点到该节点的路径。如果使用线段树或平衡树单独地维护每棵树，假设共有 k 层，那么访问一个节点的时间复杂度就会是 $O(k\log n)$ 。尽管这样做每棵树内部都是平衡的，但是并没有利用到整体结构的性质，因而得不到很好的复杂度。我们来尝试对其进行一些改进，以达到一个全局的平衡效果。

4.4.1 静态的情况

考虑定义一个节点的权重为套在它下面的树的大小之和+1，然后用4.1节中的方法把每棵树建成一棵带权的二叉树。

定理 4.5. 设节点 x 处于第 k 层，使用上述算法建树，那么节点 x 在整体结构中的深度为 $O(k+\log n)$ 。

证明. 假设节点 x 套在第 i 层的一个权重为 s_i 的节点下方（ s_k 即为 w_x ），那么深度为

$$\begin{aligned}&1+O\left(\log\frac{n}{s_1}\right)+\sum_{i=2}^{k}\left(1+O\left(\log\frac{s_{i-1}}{s_i}\right)\right)\\ =&k+O(\log n-\log w_x)\\ =&O(k+\log n)\end{aligned}$$

大部分的树链剖分或点分治+线段树的问题都可以通过这种方法优化到一个 log 。

4.4.2 动态的情况

当树变成动态时，我们可以用Splay或者Treap来维护每棵树，同样可以做到在 $O(k+\log n)$ 的时间内访问一个节点。

例 3. 鏖战表达式[71]

这是一道交互题。

我们需要处理这样一类表达式：它由 n 个元素和 $n-1$ 个运算符构成，两个相邻元素之间有且仅有一个运算符，且表达式中没有括号。这些运算符一共有 k 种，每种的优先级都不同。为了方便，我们用 1 到 k 的整数来表示这些运算符，并且数字越大的优先级越高。运算符都满足交换律和结合律。

记一开始的表达式为第 0 个版本。有 m 个操作，每个操作为以下几种之一：

- 元素修改：对某一个版本，修改某个位置上的元素；
- 运算符修改：对某一个版本，修改某两个元素之间的运算符；
- 翻转：对某一个版本，将第 l 个元素到第 r 个元素之间的所有元素（包括第 l 个和第 r 个元素）和运算符翻转。

我们记第 i 次操作之后得到的表达式为第 i 个版本。在每个操作之后，你需要求出当前表达式的值。

你只能通过调用一个函数 $F(a,b,op)$ 来得到 $a\ op\ b$ 的结果，且调用这个函数的次数不能超过 10^7 次。

$n,m\le 20000$ ， $k\le 100$ 。

考虑用平衡树来维护表达式。为了保证运算的正确性，我们要使每个节点代表的运算符的优先级比祖先的大，可以发现这样就形成了一个树的多层嵌套的结构。由于需要支持可持久化，我们使用Treap来维护每一棵树。

本题的所有操作都可以借由Treap的分裂以及连接操作来完成。我们来仔细地分析一下分裂操作的时间复杂度。假设某棵Treap中节点的权重之和为 W ，我们在第 p 个节点和第 $p+1$ 个节点之间将其分裂开，那么此次分裂的时间复杂度就是 $O\left(\log\frac{W}{\max(w_p,w_{p+1})}\right)$ ，总的时间复杂度不会超过最底层分裂点的深度。需要注意的是，一些较差的实现会使分裂的时间复杂度变为 $O\left(\log\frac{W}{\min(w_p,w_{p+1})}\right)$ ，这会导致复杂度的退化。在分裂结束之后，路径上的一些节点的权重会降低，这时我们需要对其进行一些下旋操作。这部分的时间复杂度同样不会超过最终最底层分裂点的深度，即 $O(k+\log n)$ 。

连接操作是分裂操作的逆操作，其时间复杂度也是一样的。因此，整个算法的时空复杂度为 $O(k+\log n)$ 。

[71] WC2016

4.4.3 Link-Cut Tree

通常我们会使用Splay来维护Link-Cut Tree的每条实链，其复杂度证明在许多资料中都可以找到[72]，在此不再赘述。那么我们是否可以用Treap来替代Link-Cut Tree中的Splay呢？

考虑Link-Cut Tree中虚实边切换的过程。我们首先需要将节点 x 右侧的部分分裂出来变成虚子树，之后将原本的某个虚子树拼接到 x 右侧，最后还需要调整节点 x 的权重。前两步的复杂度都受到最初节点 x 的深度的限制。然而在第三步中，节点 x 可能会因为权重减小而需要下旋。由于操作后原本access的节点已经不在 x 的虚子树中了，下旋的总复杂度就无法像之前一样限制为某个节点的深度。因此用这种方法实现的Link-Cut Tree的时间复杂度仍是两个 log 的。可见，Treap在一些形态变化较复杂的树的多层嵌套问题中仍有一定的局限性。

此处笔者并没有想到很好的方法来解决这个问题，如果读者有好的想法的话也欢迎来与我讨论。

5 总结

本文介绍的两种平衡树都通过简洁而优美的平衡策略，在许多问题上取得了非常优秀的时间复杂度。Splay的自调整实际上是一个非常强的性质。它的发明者Sleator和Tarjan还提出了一个猜想（Dynamic Optimality Conjecture）：假设 A 是某一种二叉搜索树算法，这种算法中访问节点 x 时需要以 $depth(x)+1$ 的代价遍历根节点到 x 的路径，并且在两次访问之间可以进行若干次代价为1 的旋转操作。考虑某个访问序列 S ，使用这种算法依次访问 S 中的元素的耗时为 $A(S)$ ，那么使用Splay访问 S 的时间复杂度为 $O(n+A(S))$ 。这个猜想至今仍未被证明或证伪，这也意味着Splay 仍有巨大的潜力等待我们去挖掘。而尽管Treap在一些问题上的表现可能不如Splay，但是它的优势也是非常明显的：在插入删除元素时树的整体形态变化较小，并且可以方便地支持可持久化。

事实上学术界早已对各种各样的平衡树进行了深入的研究，并得到了许多有趣的结果。但如何将这些结果与OI界的内容进行有机的结合，仍然是我们需要去思考的问题。希望本文能起到抛砖引玉的作用，吸引更多的读者来研究这一类问题。

致谢

- 感谢中国计算机学会提供交流和学习的平台；
- 感谢父母对我的关心和照顾；
- 感谢严开明老师多年以来的关心和指导；

[72] Link-Cut Tree的时间复杂度分析可以参考文献[4]

- 感谢王逸松学长与我分享了本文中涉及到的一些算法，并且为撰写本文提供了帮助；
- 感谢武弘勋、吴瑾昭、王修涵、麦骏同学为本文验稿；
- 感谢各位读者抽出时间阅读本文。

参考文献

[1] R. Cole. On the dynamic finger conjecture for Splay trees. part II: The proof. SIAM Journal of Computing, 30(1):44 – 85, 2000.

[2] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. Journal of the ACM, 32(3):652 – 686, 1985.

[3] R. Seidel and C. R. Aragon. Randomized search trees. Algorithmica, 16(4/5):464 – 497, 1996.

[4] 杨哲，《SPOJ375 QTREE解法的一些研究》，2007年国家集训队作业。

[5] 陈立杰，《重量平衡树和后缀平衡树在信息学奥赛中的应用》，2013年信息学奥林匹克中国国家队候选队员论文集。

[6] 陈立杰，《可持久化数据结构研究》，2012年国家集训队作业。

[7] Guy E. Blelloch and Margaret Reid-Miller. Fast set operations using treaps. In 10th Annual ACM Symposium on Parallel Algorithms and Architectures, Puerto Vallarta, Mexico, June - July 1998. ACM.

[8] 王逸松，《鏖战表达式》讲评课件，2016年全国冬令营。

《最小方差生成树》命题报告

北京市第八十中学 何中天

摘 要

本文将介绍作者在2018年集训队互测出的一道题目。这道题目涉及图论、数学和数据结构的知识，以图论为主。需要选手深入的挖掘性质，构建模型，并结合图像分析问题。此题设置了多档有价值的子任务，子任务的做法就能让读者耳目一新。标算不超出NOI选手的知识范围。

1 题目

1.1 题目描述

给定一个 n 个点 m 条边的带权图，每条边的边权为 w_i，有两种询问。

1.求其最小方差生成树。

2.对于每条边，问如果删除它，残余图（包含 n 个点 $m-1$ 条边）的最小方差生成树。

你只需要求出最小的方差值。如果图不连通，输出-1。

一个生成树的方差定义为它的所有边的权值的方差。

对于 N 个变量 $x_1, x_2 ... x_N$，其方差计算方式如下

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{n}(x_i - \mu)^2$$

其中 σ^2 为方差，μ 为平均值，由于是生成树，所以 $N = n - 1$。

你需要将方差乘 N^2 后输出，可以证明这是一个整数。

1.2 输入格式

第 1 行包含 3 个整数 n, m, T，表示点数，边数和询问类型。

接下来 m 行，每行包含 3 个正整数 u_i, v_i, w_i，表示第 i 条边连接 u_i 和 v_i，权值为 w_i，保证无自环，但可能有重边。

1.3 输出格式

当 $T = 1$，输出一个数表示答案。

当 $T = 2$，输出 m 行，每行一个数表示删除第 i 条边的答案。

如果图不连通，输出-1。

1.4 样例输入

4 4 2
1 2 1
2 3 3
1 3 2
3 4 5

1.5 样例输出

14
26
24
-1

1.6 数据规模[73]

对于 100% 的数据 $2 \le n \le m \le 10^5, 1 \le u_i, v_i \le n, 1 \le w_i \le 10^{18}, u_i \ne v_i$

<table>
<tr><th>子任务</th><th>分值</th><th>T</th><th>$n \le$</th><th>$m \le$</th><th>$w_i \le$</th><th>时间限制</th></tr>
<tr><td>1</td><td>5</td><td rowspan="6">1</td><td rowspan="6">m</td><td>20</td><td>10^6</td><td rowspan="6">$2s$</td></tr>
<tr><td>2</td><td>10</td><td rowspan="2">200</td><td>200</td></tr>
<tr><td>3</td><td>10</td><td rowspan="2">10^6</td></tr>
<tr><td>4</td><td>10</td><td>1000</td></tr>
<tr><td>5</td><td>10</td><td rowspan="2">100000</td><td>10^9 并且满足特性1</td></tr>
<tr><td>6</td><td>15</td><td>10^9</td></tr>
<tr><td>7</td><td>20</td><td rowspan="2">2</td><td rowspan="2">300</td><td rowspan="2">100000</td><td>10^6</td><td>$3s$</td></tr>
<tr><td>8</td><td>20</td><td>10^{18}</td><td>$5s$</td></tr>
</table>

特性1：第 i 条边连接点 $((i \bmod n) + 1)$ 和点 $(((i + 1) \bmod n) + 1)$，且 $w_i \le w_{i+1}$

[73]数据规模与原题相比进行了微调。

2 简单做法

2.1 暴力做法

暴力枚举所有所有生成树，更新答案。可以通过子任务1。

2.2 针对特性数据的做法

我们把方差的公式变形一下：

$$\begin{aligned}\sigma^2 &= \frac{1}{N}\sum_{i=1}^{N}(x_i-\mu)^2\\ &= \frac{1}{N}\sum_{i=1}^{N}\left(x_i^2-2\cdot x_i\cdot\mu+\mu^2\right)\\ &= \frac{1}{N}\left(\sum_{i=1}^{N}x_i^2-2\cdot\mu\cdot\sum_{i=1}^{N}x_i+N\cdot\mu^2\right)\\ &= \frac{1}{N}\left(\sum_{i=1}^{N}x_i^2-2\cdot\mu\cdot N\cdot\mu+N\cdot\mu^2\right)\\ &= \frac{1}{N}\sum_{i=1}^{N}x_i^2-\mu^2\end{aligned}$$

记 $s_1=\sum_{i=1}^{N}x_i, s_2=\sum_{i=1}^{N}x_i^2$ 。

最后也可以写成：

$$\sigma^2=\frac{s_2}{N}-\frac{s_1^2}{N^2} \tag{2.1}$$

由于要将答案乘 N^2 ，所以

$$answer=\sigma^2\cdot N^2=s_2\cdot N-s_1^2 \tag{2.2}$$

我们知道方差是描述数据分散程度的，所以这 $n-1$ 个数越密集，方差越小。对于子任务5，编号连续的 $n-1$ 条边一定是一个生成树，又由于边权不降，所以一定存在一种最优解是编号连续的 $n-1$ 条边。从小到大考虑，每次删去最前面的边，加入后面的边，维护 s_1 和 s_2 ，由公式(2.2)即可求出答案。

可以通过子任务5。

3 初步分析

3.1 寻找切入点

看上去没有一个显然的多项式做法。我们观察方差的式子中，最棘手的就是 μ ，如果

知道 μ 的值就很好求，把边权替换为 $(w_i-\mu)^2$ ，问题就变成了普通的最小生成树。对于 C 比较小的数据，由于 $1\le s_1\le nC$ ，所以 μ 的取值只有 $O(nC)$ 个，枚举其所有可能的取值，求一下对应的最小生成树，用这棵树更新答案，下面给出这个算法正确性的证明。

考虑这样一个问题：如果求出的最小生成树真实的 μ 和我们枚举的不一样怎么办？实际上这不影响我们求最优解。首先要证明方差关于 μ 的函数在真实的平均值处取到最小值。

$$\begin{aligned}\sigma^2&=\frac{1}{N}\left(\sum_{i=1}^{N}x_i^2-2\cdot\mu\cdot\sum_{i=1}^{N}x_i+N\cdot\mu^2\right)\\&=\mu^2-\frac{2}{N}\cdot\sum_{i=1}^{N}x_i\cdot\mu+\frac{1}{N}\sum_{i=1}^{N}x_i^2\end{aligned}\tag{3.1}$$

可以看到 σ^2 是关于 μ 的二次函数，最小值在 $\mu=\frac{1}{N}\sum_{i=1}^{N}x_i$ ，即平均值处取到。

假设 μ 的值是可以由我们定的，令 $T(x)$ 为生成树 T ， μ 的值取 x 时方差函数的值。令 T_x 是 $\mu=x$ 下求的最小生成树，对于任意满足 $\mu_T=x$ 的T ，我们有

$$T(\mu_T)\ge T_x(\mu_T)\ge T_x(\mu_{T_x})\tag{3.2}$$

不等式的前一半是因为 T_x 是 x 下求的最小生成树，并且有 $\mu_T=x$ 。后一半是因为方差关于 μ 的函数在真实的平均值处取到最小值。我们的算法相当于求 $\min_x\{T_x(\mu_{T_x})\}$ 。由上式可得

$$\begin{aligned}answer=\min_T\{T(\mu_T)\}&=\min_x\left\{\min_{\mu_T=x}\{T(\mu_T)\}\right\}\\&\ge\min_x\left\{\min_{\mu_T=x}\{T_x(\mu_T)\}\right\}\\&\ge\min_x\left\{\min_{\mu_T=x}\{T_x(\mu_{T_x})\}\right\}\ge\min_x\{T_x(\mu_{T_x})\}\end{aligned}$$

又因为

$$\min_x\{T_x(\mu_{T_x})\}\ge answer$$

所以 $answer=\min_x\{T_x(\mu_{T_x})\}$ ，正确性得证。

由于要做 $O(nC)$ 次最小生成树，时间复杂度 $O(nmC\log m)$ ，可以通过子任务2。

3.2 挖掘最小生成树算法的性质

对于 C 更大的数据， μ 的值太多了，不能枚举。于是我们需要挖掘最小生成树算法的性质，常用的Kruskal算法首先对边排序，当边的顺序确定，最小生成树也就唯一确定了。

发现正是 μ 的变化导致了边的顺序的变化，经过观察可以发现，两条边 i 和 j 大小关系改变当 μ 跨过二者的平均值。

下面证明过程不考虑边权相同的边界情况（在边界情况下算法依然成立）。

对于两条不同的边 i, j ，不失一般性设 $w_i < w_j$ ，假设边 i 优于边 j ，我们有

$$(w_i - \mu)^2 < (w_j - \mu)^2 \iff \mu < (w_i + w_j)/2 \tag{3.3}$$

于是可以取出每两条边的平均值，将它们排序并离散化，记排好序的这 k 个值为 t_k ，它们将数轴划分成 $k+1$ 个区间，也就是 $(-\infty, t1], (t1, t2], (t2, t3]...(tk, +\infty)$ ，记第 i 个区间为 $invl_i$ 。

对于某个区间 i 中的任意 $\mu \in invl_i$ ，边的顺序不变，我们在区间中选一个代表值（实现时任取区间内不在边界上的一值作为 μ 会比较方便），与上一个算法相似，对于每个区间求出其最小生成树，并更新答案。下面给出证明。

令 $T(x)$ 为生成树 T ， μ 的值取 x 时方差函数的值。令 T_i 是区间 i 下求的最小生成树。对于任意满足 $\mu_T \in invl_i$ 的 T ，我们有

$$T(\mu_T) \geq T_i(\mu_T) \geq T_i(\mu_{T_i}) \tag{3.4}$$

证明与上一小节类似。

我们的算法相当于求 $\min_i \{T_i(\mu_{T_i})\}$ 。由上式可得

$$\begin{aligned} answer = \min_T \{T(\mu_T)\} &= \min_i \left\{ \min_{\mu_T \in invl_i} \{T(\mu_T)\} \right\} \\ &\geq \min_i \left\{ \min_{\mu_T \in invl_i} \{T_i(\mu_T)\} \right\} \\ &\geq \min_i \left\{ \min_{\mu_T \in invl_i} \{T_i(\mu_{T_i})\} \right\} \geq \min_i \{T_i(\mu_{T_i})\} \end{aligned}$$

又因为

$$\min_i (T_i(\mu_{T_i})) \geq answer$$

所以 $answer = \min_i (T_i(\mu_{T_i}))$ 。

由于要做 $O(m^2)$ 次最小生成树，时间复杂度 $O(m^3 \log m)$ ，可以通过子任务3。

如果按区间从小到大来求，在两条边大小关系改变时，考察能否用新的小边替换大边使得还是一棵树，也就是不断调整最小生成树，用LCT维护，由于改变次数 $O(m^2)$ ，所以时间复杂度 $O(m^2 \log m)$ ，LCT常数很大，预计不能通过之后的子任务。

例题 1. Topcoder SRM 611 Egalitarianism2(Div1 550pts)

n 个点的完全图，边权为欧几里得距离，求最小标准差生成树， $n \leq 20$ 。

使用子任务3的的算法，只需要改用对稠密图更优的prim算法，时间复杂度 $O(n^6)$ 。

4 进一步优化

我们已经有了一个多项式算法了，现在要优化它。在上一节中对每一个区间独立计算，然后取最小值，我们没有充分利用区间之间的关系。

定理 4.1. 一条边 i 存在于连续一段区间的最小生成树中。

下面给出证明。(边界情况的证明较为繁琐，以下过程不考虑)

观察一条边关于 μ 的代价函数 $c_i = (\mu - w_i)^2$ ，它是一个开口向上的二次函数，并在 $\mu = w_i$ 处取极小值 0 ，所以在 $\mu = w_i$ 处边 i 一定在最小生成树中，则有 $l_i \leq w_i \leq r_i$ 。对于给定的 μ_0 求最小生成树，相当于用 $\mu = \mu_0$ 这条线去截那些代价函数曲线，交点由低到高的顺序就是对应边的顺序。

记 $E_{p(e)}$ 为所有满足条件 $p(e)$ 的边 e 的集合。

引理 4.1. 对于边 i ，若 u_i 和 v_i 在图 $G(V, E_{c_e<c_i})$ 中连通，则边 i 不在最小生成树中。否则边 i 在最小生成树中。

我们来研究这个代价的二次函数在 $\mu < w_i$ 的一半，从而分析 l_i 满足的性质。

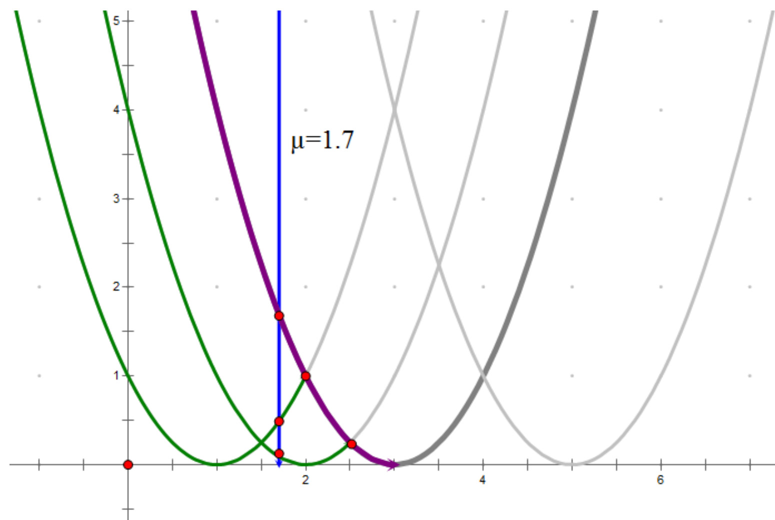


图 3: 样例输入的函数图像。紫线是我们要分析的边 i 的函数的一半，绿线表示这条边比边 i 优，绿线在横轴上的投影就是这条边比边 i 优的范围。

对于 $w_j > w_i$ 的边 j ， $c_j > c_i$ 在 $\mu < w_i$ 上恒成立，所以对边 i 在不在最小生成树中无影响。

对于 $w_j < w_i$ 的边 j ， $c_j < c_i$ 当且仅当 $\mu < (w_i + w_j)/2$ 。所以 $E_{c_e<c_i}$ 在 μ 减小时其包含的边集增大，所以存在某一值 l_i ，使得当 $\mu < l_i$ 时 u_i 和 v_i 在图 $G(V, E_{c_e<c_i})$ 中连通，当 $\mu > l_i$ 时 u_i 和 v_i 在图 $G(V, E_{c_e<c_i})$ 中不连通。如果在无穷小处 u_i 和 v_i 在图 $G(V, E_{c_e<c_i})$ 中不连通，那么 $l_i = -infty$ 。

同理分析代价函数在 $\mu > w_i$ 的另一半，可以得到 r_i 向相反的方向满足同样的性质。

以上就完成了定理4.1的证明。

对于边 i ，我们要求出它存在于最小生成树中的区间的左右 l_i 和 r_i 。以 l_i 为例，将 $w_j < w_i$ 的边 j ，按 $(w_i + w_j)/2$ 从大到小排序，逐一加入$E_{c_e<c_i}$ ，用并查集维护，直到 u_i 和 v_i 连通，设最后加入的边为 j ，则 $l_i = (w_i + w_j)/2$ 。同理向相反的方向可以求出 r_i 。

把所有 l_i 和 r_i 排序并离散化，即可得到 $O(m)$ 个时间点，它们将数轴划分成 $O(m)$ 个区间，从小到大扫这些区间，在 l_i 处加入边 i ， r_i 处删除边 i 。不用维护这棵树的形态，只需要用2.2 节的方法求方差的值即可。

时间复杂度 $O(m^2 \log m)$ ，可以通过子任务4。

5 $T = 1$ 的标准解法

观察上一节中 l_i 和 r_i 的求法。以 l_i 为例，按 $(w_i + w_j)/2$ 排序实际上可以简化为按 w_j 排序，因为 w_i 与边 j 无关。排序后逐一加入边，用并查集维护的过程就是一个求最大生成树（森林）的过程。所以我们求出 $G(V, E_{w_e<w_i})$ 的最大生成树（森林），找出 u_i 和 v_i 的路径上边权最小的边 j ，则 $l_i = (w_i + w_j)/2$ ，如果 u_i 和 v_i 不连通，那么$l_i = -infty$ 。按边权 w_i 从小到大来求，动态维护 $G(V, E_{w_e<w_i})$ 的最大生成树，可以用LCT实现。同理求 r_i 动态维护最小生成树即可。

求出 l_i 和 r_i 后的做法与上一节相同。

时间复杂度 $O(m \log m)$ ，可以通过子任务6。

5.1 优化版标准解法

定理 5.1. 对于边 i 和边 j ，若 $G(V, E_{w_e<w_i})$ 的最大生成树上 u_i 和 v_i 的路径上边权最小的边 j ，则 $l_i = r_j = (w_i + w_j)/2$ 。

证明：. $l_i = (w_i + w_j)/2$ 在上文已经给出理由。我们只需证 $r_j = (w_i + w_j)/2$ 。

由于 j 是 $G(V, E_{w_e<w_i})$ 的最大生成树上的边，所以 $G(V, E_{w_j<w_e<w_i})$ 中 u_j 和 v_j 不连通（引理4.1）。而在 $E_{w_j<w_e<w_i}$ 中再加入边 i ， u_j 和 v_j 就连通了，因为在 $G(V, E_{w_e<w_i})$ 的最大生成树上，边 i 与 u_i 和 v_i 的路径构成了环，边 j 又是这个环上最小的边，所以 $E_{w_j<w_e<w_i}$ 中再加入边 i 就包含了环上所有除了边 j 的边，因此 u_j 和 v_j 就连通了。根据上文介绍的 l 和 r 的求法，边 i 是最小的使 u_j 和 v_j 连通的边，所以$r_j = (w_i + w_j)/2$ 。 □

从这个定理出发，在求出 l_i 的时候顺便也就求出了 r_i ，一遍动态树即可求出。

时间复杂度 $O(m \log m)$ ，可以通过子任务6。

6 最小乘积生成树

我们来看另一个问题，最小乘积生成树，并尝试使用最小方差生成树的做法。

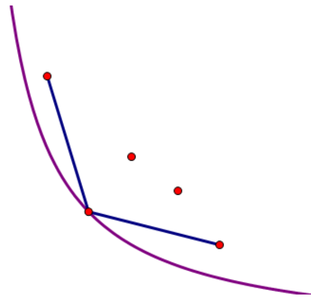
例题 2. 经典问题 [74]

给定一张无向图，每条边都有两个正边权 a_i 和 b_i 。一棵生成树的权值定义为所有边 a 的权值和乘上 b 的权值和。求权值最小的生成树。 $n \le 200, m \le 10000.$

6.1 问题分析

令 $x = \sum_{i \in T} a_i, y = \sum_{i \in T} b_i$ 。那么生成树 T 就对应二维平面上的点 (x, y) 。

观察图像，考虑权值相同的点构成的曲线 $c = xy$ ，我们要找到最左下方的曲线使得有点在曲线上，可以看出这个点一定在左下凸壳上，其原因在于这个代价曲线是凸的。



下面证明刚才观察出的结论，即凸包上的点一定比凸包内的点优。

考虑线段 $(a, b) - (a + c, b + d)(c > 0, d < 0)$ 。

最小化 $(a + ct)(b + dt) = ab + (ad + bc)t + cdt^2(0 \le t \le 1)$ 。

因为 $cd < 0$ ，所以这是一个关于 t 的开口向下的二次函数，最小值一定在区间的左边界或右边界即 $t = 0$ 或 $t = 1$ 时取到。凸性得证。

6.2 标准解法

接下来我们需要求出所有左下凸壳上的点。首先找到 x 最小的点 A 和 y 最小的点 B 。找到 AB 左下方最远的点 C 。然后 AB 被分成 AC 和 CB 两段，对 AC 和 CB 递归求左下方最远的点，直到该点与两端点共线为止。

怎么求 AB 左下方最远的点 C 呢？实际上是最小化和 AB 同斜率的直线在 y 轴上的截距。设 AB 的斜率为 $-k$ ，则需最小化 $kx + y$ ，只需要将每条边的边权替换为$ka_i + b_i$ 求最小生成树即可。

[74]Balkan OI 2011 Timeismoney

总点数不超过 C_m^{n-1} 。如果把点视为随机的话，凸包上的点数为 $O(n\log m)$ 。时间复杂度 $O(n^3\log n)$ 。可以通过此题。

6.3 与最小方差生成树的类比

对于某个下凸壳上的点 (x,y) ，我们发现一定存在 k 使得 (x,y) 是 $kx+y$ 最小的点。将 $kx+y$ 变形为 $xk+y$ ，一条边的贡献是 a_ik+b_i 。于是我们建立了与最小方差生成树类似的模型，只是代价函数不同，这里是关于 k 的一次函数，方差是关于 μ 是二次函数。这里的 k 与方差中的 μ 效果类似。于是可以用类似的做法求解。两条边的大小关系在两条直线的交点处变化，有 m^2 个相对顺序，用子任务3的做法求解，就能将下凸壳上的所有点求出。时间复杂度 $O(m^3\log m)$ ，若用LCT 优化，复杂度 $O(m^2\log m)$ ，虽然这种方法复杂度更高，但它证明了最坏情况下下凸壳上的点数不超过 m^2 个。

能否用第4、5节所介绍的方法来优化呢？这取决于定理4.1是否还成立。遗憾的是可以举出反例：

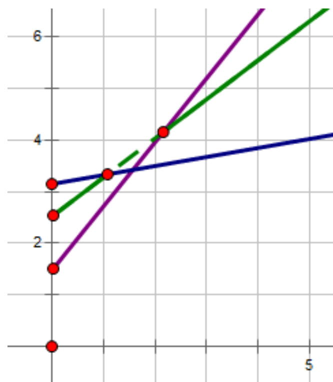


图 4: 三个一次函数分别对应三条端点为(1,2)、(1,3)和(2,3)的边，绿色的实线部分表示在最小生成树中，虚线表示不在，可以看出存在区间不连续。

虽然继续优化失败了，但我们对这个算法的模型与适用条件有了更深的认识。观察发现，当边的代价函数是单谷函数（也就是开口向上的单峰函数），并且每条边的函数图像都可以由第一条边的图像沿 x 轴平移得到，这一类情形都满足定理4.1，可以用第4、5节的方法来优化。

有兴趣的读者可以寻找定理4.1还适用于哪些函数。

7 $T=2$ 的算法

这部分的算法是基于第一问算法的拓展。

7.1 问题分析

首先求出第一问不删边的最小方差生成树 T_0 。

如果删掉的不是 T_0 中的边，答案不变，所以只需要对 $O(n)$ 条边求第二问的答案。

如果删掉一条边 k ，求出 $G(V, E_{w_e<w_k})$ 的最大生成树（或森林） T_l ，和 $G(V, E_{w_e>w_k})$ 的最小生成树（或森林） T_r 。

定理 7.1. 删去边 k 后， l_i 变化的边一定在 T_r 中， r_i 变化的边一定在 T_l 中。

证明：. 由于对称性，只需证 l_i 变化的边一定在 T_r 中。对于边 i 不在 T_r 中，以下分两种情况讨论。

若 $w_i < w_k$ ，根据上文 l 的求解过程可知， l_i 不受删除边 k 的影响。

若 $w_i > w_k$ ，在图 $G(V, E_{w_k<w_e<w_i})$ 中， u_i 和 v_i 连通（根据引理4.1，把其中的边集 E 替换为 $E_{w_k<w_e}$ ，如果 u_i 和 v_i 在图$G(V, E_{w_k<w_e<w_i})$ 中不连通，边 i 在 $G(V, E_{w_k<w_e})$ 的最小生成树也就是 T_r 中，与假设矛盾）。由上文所介绍的 l 的求解过程可知，因为在图 $G(V, E_{w_k<w_e<w_i})$ 中 u_i 和 v_i 连通，所以 $l_i > (w_i + w_k)/2$ ，且 l_i 不受删除边 k 的影响。 □

为了求出这些变化的 l 和 r ，我们还是要动态维护生成树（不是最大生成树，而是有某种性质的生成树）。初始设 $T = T_l$ ，把 T_r 上的边从小到大逐一加到 T 上，当加入边 i 时，替换掉 T 中 u_i 和 v_i 的路径上边权最小的边，记加入边 i 时的 T 为 T_i 。通过下面的定理7.2即可求出变化的 l 和 r 。

定理 7.2. 对于 T_r 上的一条边 i ， T_i 中 u_i 和 v_i 的路径上边权最小的边为边 j ，边 j 属于 T_l ，并且 $l_i = r_j = (w_i + w_j)/2$ 。

注意这个定理与定理5.1的区别在于 T_i 不是最大生成树。此定理的证明与定理5.1的证明过程类似，不再赘述。

求出新的 l 和 r 后的做法与之前相同，这一步的时间复杂度为 $O(m)$ 。直接排序是 $O(m \log m)$ 的，由定理7.1可知，变化的 l_i 和 r_i 只有 $O(n)$ 个，我们可以对它们排序，再和原始的 l 和 r 归并（归并时要跳过原始的 l 和 r 中发生变化的元素），排序复杂度降至 $O(m + n \log n)$ 。

由于 n 比较小，所以维护最大生成树可以暴力实现，当然用LCT也可以。这一步的时间复杂度为 $O(n^2)$ 或 $O(n \log n)$ 。

以上过程要做 $O(n)$ 次，所以总时间复杂度 $O(n^3 + nm)$ 或 $O(n^2 \log n + nm)$ 。可以通过子任务7。由于高精度常数大，无法通过子任务8，我们需要复杂度更优的算法。

7.2 $T = 2$ 的标准解法

我们回顾之前的做法。

把所有 l_i 和 r_i 排序并离散化，即可得到 $O(m)$ 个时间点，它们将数轴划分成 $O(m)$ 个区间，我们从小到大扫这些区间，在 l_i 处加入边 i ， r_i 处删除边i 。

而求答案是通过公式2.2即

$$answer = \sigma^2 \cdot N^2 = s_2 \cdot N - s_1^2$$

我们将在 l_i 处加入边 i ， r_i 处删除边 i 的过程看做求前缀和。在 l_i 处加 w_i ， r_i 处减 w_i ，我们称这个序列为原序列。对其求前缀和即可得到s_1 ， s_2 同理。

由定理7.1可知，变化的 l_i 和 r_i 只有 $O(n)$ 个，原序列的改变只有 $O(n)$ 处，它们将前缀和序列分成 $O(n)$ 个区间。每个区间的 Δs_1 和 Δs_2 是固定的，可以通过对变化的位置求前缀和得到。设前缀和序列的第 i 项为 s_{i1} 和 s_{i2} ，第 i 项的答案为

$$answer = (s_{i2} + \Delta s_2) \cdot N - (s_{i1} + \Delta s_1)^2 \tag{7.1}$$

展开得

$$answer = (s_{i2} + \Delta s_2) \cdot N - (s_{i1}^2 + 2 \cdot s_{i1} \cdot \Delta s_1 + \Delta^2 s_1) \tag{7.2}$$

可以用斜率优化，将第 i 项看做二维平面上的点 $(2 \cdot s_{i1}, s_{i2} \cdot N - s_{i1}^2)$ ，求这些点的下凸壳，询问看做斜率为 $-\Delta s_1$ 的直线，答案最优在直线与下凸壳相切时得到。

每次查询需要其中一段区间的点的下凸壳。建立线段树，每个线段树节点上求出对应区间的点的下凸壳。由于要对 $O(n)$ 条边求第二问的答案，每次又分为 $O(n)$ 个区间，总计有 $O(n^2)$ 个区间。直接在线段树节点上二分的时间复杂度是 $O(n^2 \log^2 m + m \log m)$ 的，不够优秀。把询问离线按斜率排好序，在线段树节点上记录一个指针指向上一次询问的最优解，因为询问关于斜率单调，所以最优解在下凸壳上也满足单调性，指针只会向一个方向移动，均摊复杂度是 $O(m \log m)$ 的[75]，这部分总时间复杂度是 $O(n^2 \log m + m \log m)$ 。

由于 n 比较小，所以维护最大生成树可以暴力实现，当然用LCT也可以。

这也是唯一需要高精度的子任务（子任务6可以使用int128）。为什么这道题要求高精度而不直接输出实数呢？因为那样不能区分出最后这个斜率优化做法，由于LCT的常数大，耗时瓶颈在LCT的部分，所以增大数据范围也不能区分这两个做法。

设高精度时间复杂度为常数 P 。时间复杂度 $O(P \cdot (n^2 + m) \log m)$ 。可以通过子任务8。

8 总结

这道题的知识点包括：图论方面的最小生成树，数学方面考察了函数的分析、斜率优化，数据结构用到线段树、动态树。都属于NOI选手的知识范围。需要选手有一定的思维能力和建模能力。在第6节中，笔者将本题的模型加以推广，将方差函数换成一类单谷函数本题的方法依然适用，也能够让读者对这个算法的模型有更深的认识。

[75]CTSC2016时空旅行出现过这种优化方法。

9 致谢

感谢中国计算机学会提供学习和交流的平台。
感谢贾志勇老师多年来给予的关怀和教诲。
感谢国家集训队教练张瑞喆的指导。
感谢罗剑桥学长对我的帮助。
感谢张宇博、王子健、高睿泉、陈通同学对我的帮助。
感谢所有其他帮助过我的老师和同学们。
最后，感谢我的父母对我的无微不至的关心和照顾。

参考文献

[1] Thomas H.Cormen, Charles E.Leiserson, Ronald L.Rivest, Clifford Stein, Introduction to Algorithms.

[2] 唐文斌，《图论专题之生成树》，2012年NOI冬令营。

[3] 吉如一，《最小乘积问题》

[4] TopCoder Algorithm Problem Set Analysis，apps.topcoder.com/wiki/display/tc/SRM+611

[5] Balkan OI 2011 Timeismoney官方题解，www.boi2011.ro/resurse/tasks/timeismoney-sol.pdf

欧拉图相关的生成与计数问题探究

北京师范大学附属实验中学 陈通

摘 要

本文着重从生成与计数的角度对欧拉图相关的问题进行研究。介绍了欧拉图的判定方法、重要性质以及欧拉回路生成算法。在欧拉图性质的基础上，归纳了欧拉图生成问题的经典模型和转化方法，探究了欧拉图相关计数问题的一些思路，并给出了一些结论。

引 言

著名的哥尼斯堡七桥问题是18世纪著名的古典数学问题之一，该问题在相当长的时间里无人能解。欧拉经过研究，于1736年发表了论文《哥尼斯堡的七座桥》。这篇论文不仅圆满地回答了哥尼斯堡居民提出的问题，而且给出并证明了更为广泛的一般性结论。从那时起，图论作为数学的一个新的分支而诞生。因此，欧拉图问题是图论的起源，也是图论研究中十分重要的一部分。

在信息学竞赛中，图论考察的重点是生成问题与计数问题。而欧拉图因具有特殊性质而成为这类题目的基本模型之一。这类问题需要我们要根据欧拉图的特点归纳出针对的方法加以解决。我们将在本文中对欧拉图相关的生成与计数问题进行探究。

本文在第1节中，对一些基本概念给出了定义。在第2节中，介绍了欧拉图的判定方法。第3节中，介绍了欧拉回路构造算法以及扩展应用。第4节中，介绍了一些与欧拉图相关的常用结论与几类经典试题。在第5节中，介绍了欧拉图生成问题的一些模型与转化方法。在第6节中，介绍了一些与欧拉图相关的计数问题。在第7节中进行了总结。

1 基本概念

欧拉图问题是图论中的一类特殊的问题。在本文的介绍过程中，我们将会使用一些图论术语。为了使本文叙述准确，本节将给出一些术语的定义。

定义 1.1. 图 G 中与顶点 v 关联的边数（自环统计两次）称为图 G 中顶点 v 的**度**。特别地，对于有向图 G ，进入顶点 v 的边的条数称为顶点 v 的**入度**；从顶点v 引出的边的条数称为顶点 v 的**出度**。

定义 1.2. 图 G 中度为奇数的点称为**奇顶点**，度为偶数的点称为**偶顶点**，度为 0 的点称为**孤立点**。

定义 1.3. 对于无向图 G 中的两点 u 和 v ，若存在 u 到 v 的路径，则称 u 和 v 是**连通**的。如果图 G 中任意两点都是连通的，则称图 G 为**连通图**。对于有向图 G ，将所有的有向边替换为无向边得到图 G 的基图，若图 G 的基图是连通的，则称图 G 是**弱连通图**。

定义 1.4. 对于图 G 中边 e ，若删去 e 后图 G 的连通分量的数量增加，则称边 e 为 G 的**桥**。

定义 1.5. 图 G 中一条**路径**指一个顶点与边的交替序列 $v_0, e_1, v_1, ..., e_m, v_m$ ，其中 e_i 为 v_{i-1} 到 v_i 的一条边。**回路**指满足 $v_0 = v_m$ 的一条路径，一般不区分起点。

定义 1.6. 图 G 中经过每条边恰一次的回路称为**欧拉回路**，经过每条边恰一次的路径称为**欧拉路径**。

定义 1.7. 存在欧拉回路的图称为**欧拉图**，存在欧拉路径但不存在欧拉回路的图称为**半欧拉图**。

定义 1.8. 不含平行边（也称重边）也不含自环的图称为**简单图**。

2 欧拉图的判定

2.1 无向欧拉图

一笔画问题是生活中常见的一类欧拉图问题，例如汉字 “日” 和 “中” 字都可一笔画，而 “田” 和 “目” 则不能。那么究竟具有什么性质的图可以一笔画出。画出一张图的最少笔画数又是多少。早在十八世纪，欧拉研究七桥问题时就提出了欧拉图问题的实质：

定理 2.1. 无孤立点的无向图 G 为欧拉图，当且仅当图 G **连通**且**所有顶点的度都是偶数**。

证明. 首先证明必要性。因为图G存在欧拉路径且没有孤立点，所以任意两个点都可以相互到达，图G一定是连通图。如果图G存在回路 $v_0, v_1, ..., v_{m-1}, v_0$ ，那么对于顶点 $v_i(i = 0, 1, 2, 3, ..., m-1)$ 来说，有一条进入 v_i 的边就有一条从 v_i 引出的边，所以与 v_i 相邻的边总是成对的，得到 v_i 的度为偶数。因此，如果图 G 存在欧拉回路，那么图 G 为连通图且所有顶点的度都是偶数。

接着我们来说明充分性。从 G 中任意顶点 v_0 出发，经关联的边 e_1 进入 v_1 ，因为 v_1 的度数是偶数，一定可以由 v_1 再经关联的边 e_2 进入 v_2 ，如此继续下去，每条边仅经过一次，经过若干步后必可回到 v_0 ，于是得到了一个圈 c_1 。如果 c_1 恰是图 G ，则命题得证。否则在 G 中去掉 c_1 后得子图 G_1 ， G_1 中每个顶点也是偶顶点，又因为图 G 是连通的，所以在 G_1 中必定存在一个和 c_1 公共的顶点 u ，在 G_1 中存在一个从 u 出发，到 u 结束的圈 c_2 ，于

是 c_1 和 c_2 合起来仍是一个回路。重复上述过程，因为G 中总共只有有限条边，总有一个时候得到的图恰好是 G 。

对于存在奇顶点的图，我们可以利用以下结论求出最少笔画数：

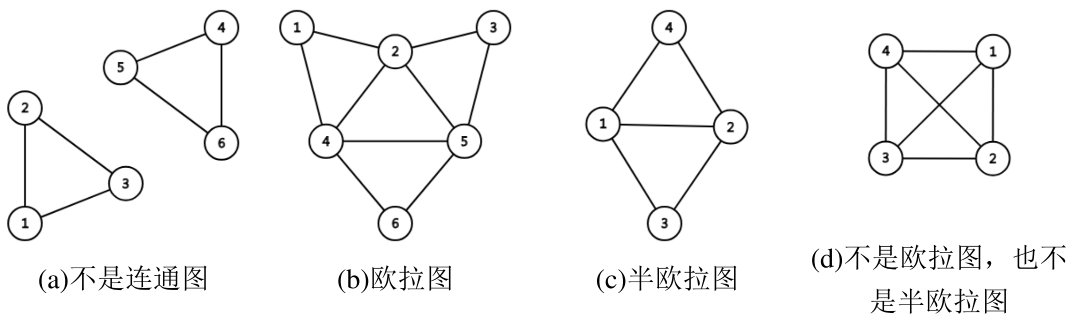
定理 2.2. 如果无向连通图有 $2k$ 个奇顶点，则图 G 可以用 k 条路径将图 G 的每一条边经过一次，且至少要使用 k 条路径。

证明. 我们先来证明路径条数的下界。设图 G 可以分解成 h 条链，每条链上至多有两个奇顶点，所以有 $2h \geq 2k$ ，即 $h \geq k$ 。因此 k 是路径数量的下限。

接下来我们只需要构造一组方案。把这 $2k$ 个奇顶点分成 k 对 $(v_1, v_1'), (v_2, v_2'), \ldots, (v_k, v_k')$ 。在每对点 (v_i, v_i') 之间添加一条边，得到图 G' 。图 G' 连通且没有奇顶点，所以 G' 存在欧拉回路。再把这 k 条新添的边从回路中去掉，这个圈至多被分为 k 段，我们就得到了 k 条链。这说明图 G 是可以用 k 条路径将图 G 的每一条边经过一次的。

综合上述两个定理，我们已经了解了欧拉回路与欧拉路径存在的充要条件。我们可以由此推导出半欧拉图的判定条件：

定理 2.3. 无孤立点的无向图 G 为半欧拉图，当且仅当图 G 连通且 G 的奇顶点个数为 2 。此时两个奇顶点分别为欧拉路径的起点和终点。



(a)不是连通图 (b)欧拉图 (c)半欧拉图 (d)不是欧拉图，也不是半欧拉图

图1：欧拉图与半欧拉图的图示

2.2 有向欧拉图

对于有向图我们可以通过同样的思路得到类似的结论。因其证明思路与无向图基本相同，证明过程从略。

定理 2.4. 无孤立点的有向图 G 为欧拉图，当且仅当图 G 弱连通且所有顶点的入度等于出度。

定理 2.5. 对于连通有向图，所有顶点入度与出度差的绝对值之和为 $2k$ ，则图 G 可以用 k 条路径将图 G 的每一条边经过一次，且至少要使用 k 条路径。

定理 2.6. 无孤立点的有向图 G 为半欧拉图，当且仅当图 G 弱连通，且恰有一个顶点 u 入度比出度小 1 ，一个顶点 v 入度比出度大 1 ，其余顶点入度等于出度。此时存在 u 作为起点， v 作为终点的欧拉路径。

3 欧拉回路的生成

既然我们已经了解了欧拉图与半欧拉图的判定方法，那如何在欧拉图中求出一条欧拉回路呢？我们接下来考虑如下问题：

问题 3.1. 给定无向欧拉图 $G = (V, E)$ ，求出图 G 的一条欧拉回路。

本节中我们将介绍Fluery算法与Hierholzer算法来解决此问题。 [4]

3.1 Fluery算法

3.1.1 算法简介

想要求出一条欧拉回路，最直接的想法就是依次确定回路上的每一个点。那么我们需要保证的是，在未访问边构成的子图中，当前路径的终点到起点是存在欧拉路径的。

我们用 p_k 表示经过 k 条边时的当前路径，未访问边构成的子图为 G_k 。图 G_k 由图 G 删除路径 p_k 得到，除了 v_0 与 v_k 之外所有点度均为偶数。所以只要图 G_k 除去孤立点是连通的，那么 G_k 一定存在一个 v_k 到 v_0 的欧拉路径。

我们接下来要考虑如何选择路径中的下一个点，使得未访问边构成的子图是连通的。初始时，图 G 存在欧拉回路。假设 G_k 是连通的，因为 v_k 是欧拉路径的端点，所以 v_k 的关联边中最多只有一条桥边。若 v_k 有多条关联的未访问边，一定可以找到一条非桥边作为下一条访问边。图 G_{k+1} 由图 G_k 删除一条非桥边（或 v_k 的最后一条边）得到，图 G_{k+1} 除去孤立点外一定连通。

换句话说，若当前点的出边中未访问的边不只一条，我们应该选择一条不是桥的边，作为下一条访问边。如此反复，最终就能得到一条欧拉回路。

3.1.2 算法流程

1. 任取 G 中的一个顶点 v_0 ，令 $p_0 : v_0$ 。

2. 记当前求出的路径 $p_k: v_0, v_1, ..., v_k$ 。若 v_k 只有一条未删除的出边 (v_k, u) ，则令 $v_{k+1} = u$ 。否则找到一条未删除的边 (v_k, u) ，使得在删去 (v_k, u) 后， u 仍然能到达 v_k ，令 $v_{k+1} = u$ 。删去边 (v_k, v_{k+1}) ，将 v_{k+1} 加入当前路径得到 p_{k+1} 。

3. 重复(2)直到不能进行为止，此时的路径 p_m 即为一条欧拉回路。

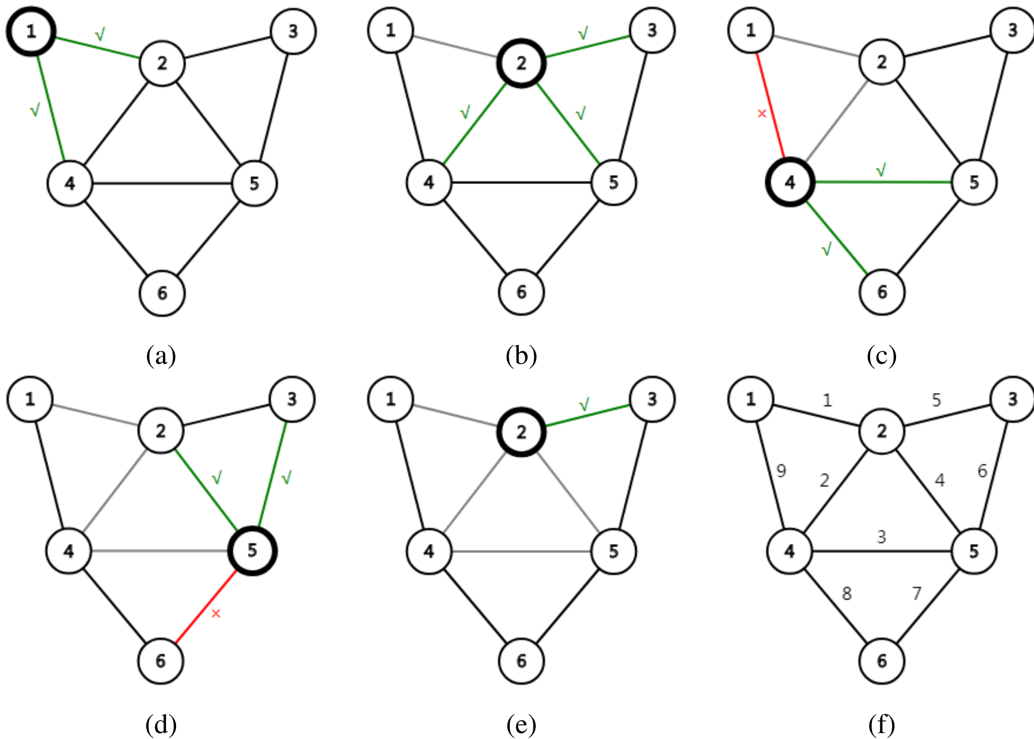


图2：Fleury算法的执行过程

图2(a)-(e)展现了Fleury算法的执行过程，其中绿色表示可以作为下一条访问边，红色表示不可以作为下一条访问边。图2(f)表示在这个例子上Fleury算法得到的最终结果。

3.1.3 伪代码

根据上面的描述，这个算法的实现如算法1所示。其中 u 为给定的起点， $path$ 为求出的欧拉回路。

我们可以使用链表维护边（无向边用两条反向的有向边表示），加入（或删除）时同时加（或删）两条边，并通过DFS（或BFS）判断连通性。在这种实现方式下，Fluery算法的时间复杂度为 $O(m^2)$ 。

算法 1 FLEURY(u)

```
1:  size ← 0
2:  while u 存在未被删除的无向边 (u, v) do
3:      path[size] ← u
4:      size ← size + 1
5:      删除无向边 (u, v)
6:      if u 还有未被删除的边且 v不能到达u then
7:          加入无向边 (u, v)
8:          continue
9:      end if
10:     u ← v
11: end while
12: path[size] ← u
13: size ← size + 1
```

3.1.4　算法分析与扩展

虽然该算法的效率不及下文将要介绍的同类算法，但由于该算法求解路径的方式十分直接，所以可以用来处理一些其他算法不能求解的问题。例如：

问题 3.2. 给定无向欧拉图（半欧拉图） $G = (V, E)$ ，求出图G的所有不同的欧拉路径。

解法. 在确定起点后，进行递归搜索，若当前点有多条未访问的关联边，则在回溯时依次尝试所有非桥边作为下一条访问边。直到所有边都已访问过，我们就得到了一条欧拉路径。最终我们就不重复地枚举出了所有欧拉路径。

在一些情况下，欧拉路径的数量很多，所以这一方法并不适合解决欧拉回路的计数问题。我们将在后面章节介绍的欧拉回路计数问题的计算方法。

3.2　Hierholzer算法

3.2.1　算法简介

该算法也被称作“套圈算法”或“DFS法”。因为效率高、代码短等优势，该算法成为信息竞赛中最常用的欧拉路径算法。

该算法的思路与定理2.1的构造思路基本相同。任选一起点，沿任意未访问的边走到相邻节点，直至无路可走。此时必然回到起点形成了一个回路，此时图中仍有部分边未被访问。在退栈的时候找到仍有未访问边的点，从该点为起点求出另一个回路，将该回路与之前求出的回路拼接。如此反复，直至所有的边都被访问。

3.2.2 算法流程

1. 任取 G 中的一个顶点 v_0，将 v_0 加入栈 S 。
2. 记栈 S 顶端元素为 u，若 u 不存在未访问的出边，将 u 从栈 S 中弹出，并将 u 插入路径 P 的**前端**。否则任选一条未访问的出边 (u, v)，将 v 加入栈 S 。
3. 重复(2)直到栈 S 为空，此时 P 为所求得的欧拉回路。

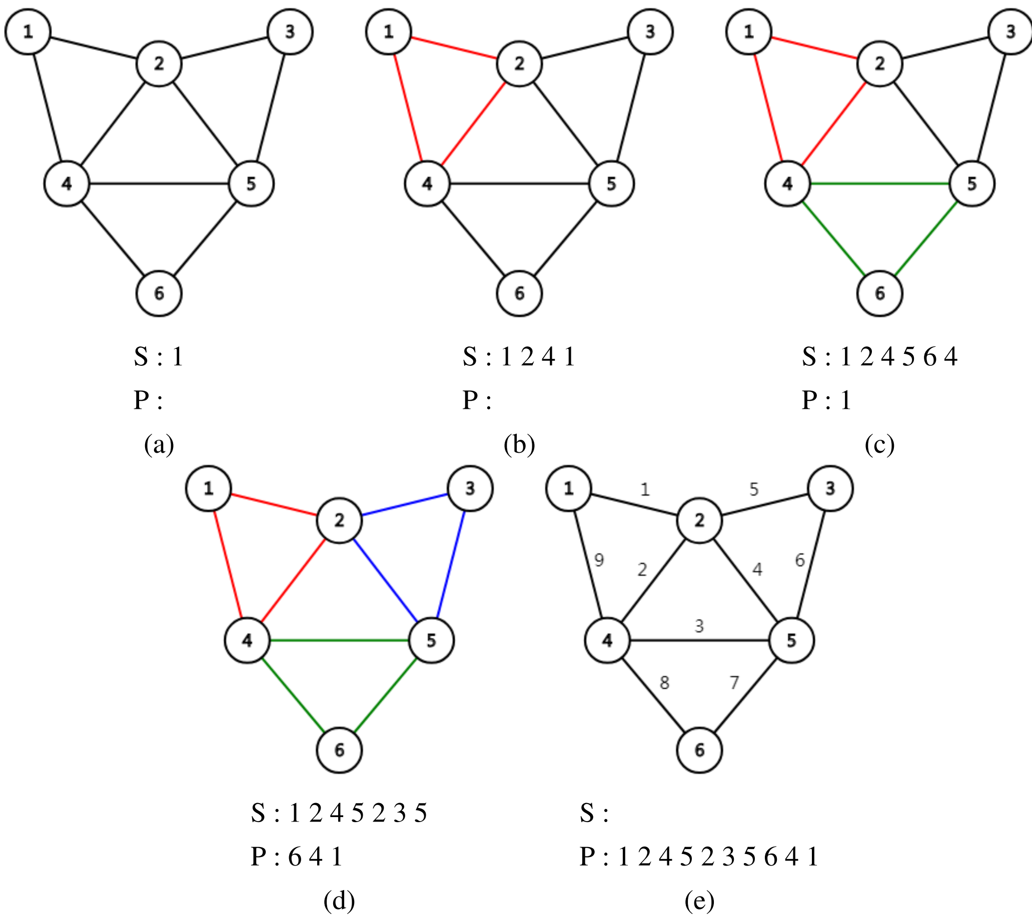


图3：Hierholzer算法的执行过程

图3(a)-(d)展现了Hierholzer算法的执行过程，不同颜色表示在执行过程中找到的不同回路。图3(e)表示在这个例子上Hierholzer算法得到的最终结果。

3.2.3 伪代码

根据上面的描述，这个算法的递归实现由算法2所示。其中 u 为给定的起点， $path$ 的**反序**为求出的欧拉回路。

算法 2 HIERHOLZER(u)

```
1: while u 存在未被删除的无向边 (u, v) do
2:     删除无向边 (u, v)
3:     HIERHOLZER( v )
4: end while
5: path[size] ← x
6: size ← size + 1
```

我们可以使用链表维护边（无向边用两条反向的有向边表示），删除时同时删去两条边。Hierholzer算法的时间复杂度为 $O(m)$ 。

这个算法也可以很容易地按照算法流程中描述的方式改成非递归形式。

3.2.4 算法分析与扩展

该算法同样可以解决有向图的情况与求解欧拉路径：

问题 3.3. 给定有向欧拉图 $G = (V, E)$ ，求出图 G 的一条欧拉回路。

解法. 对于有向图，只需用链表维护有向边即可，删除时只需删除一条边。

问题 3.4. 给定无向（或有向）半欧拉图 $G = (V, E)$ ，求出图 G 的一条欧拉路径。

解法. 对于包含两个奇顶点的无向图，以任意一个奇顶点作为起点。对于有向图，找到入度比出度小 1 的顶点作为起点。使用上述算法就可以得到一条欧拉路径。此时，算法的执行过程就是将一条路径与多个回路依次合并。

我们还可以使用该算法求解对字典序有要求的问题：

问题 3.5. 给定无向（或有向）欧拉图（或半欧拉图） $G = (V, E)$ ，求出图 G 的一条字典序最小的欧拉回路（路径）。

解法. 我们对每个点 u 关联的出边 (u, v) 按照 v 排序，找到图中编号最小的可作为起点的点出发，每一次走到编号最小的相邻节点 v 。在算法执行过程中，先会找到包含起点的一个字典序最小的回路（或路径）。从后向前找到一个仍有未访问边的节点，将一个包含该点的字典序最小的回路拼接到已求得的回路（或路径）中。如此反复就能得到字典序最小的回路（或路径）。

问题 3.6. 给定无向（或有向）连通图 $G = (V, E)$ ，求出最少的路径将图 G 的每一条边经过一次。

解法. 这一问题在定理2.2中已经给出了构造方案，对奇点两两配对，在每对点之间加入一条边构成欧拉图。求出新图的欧拉回路，将回路从新加入的边处断开，就得到了 k 条路径。

4 欧拉图相关的性质

通过2、3两节的介绍，我们对欧拉图的判定方法和欧拉回路的生成有了一定了解。下面我们通过解决一些实际问题，分析欧拉图的相关性质。

定理 4.1. 对于任意无向连通图 G ，一定存在回路使得每条边经过恰好两次。进一步地，存在回路使得每条边的两个方向各经过一次。

证明. 我们将图 G 的每一条边重复两次，得到无向图 G_1 。 G_1 是连通图，且所有点的度都是图 G 中对应点的度的两倍。因此 G_1 是欧拉图，存在欧拉回路。

同理，若把图 G 的每条无向边变为两条反向的有向边，得到有向图 G_2 。 G_2 也存在欧拉回路，满足图 G 每条边的两个方向各经过一次。

例题 4.1. 给定 n 个点， m 条边的无向连通图。其中 $m-2$ 条边经过两次，余下的两条边各经过一次，经过次数为一次的两条边不同就认为是不同的路径，问有多少种不同的路径。满足 $n, m \leq 10^6$ 。[76]

解法. 将图 G 中所有的边重复两次得到图 G' 。现在需要从 G' 中删去两条不同的边，使其存在欧拉路径，满足奇顶点个数为 0 或 2 。满足条件的只有以下三种组合方式：（1）删去的两条边都是自环。（2）删去的两条边中恰有一条为自环。（3）删去的两条边均不是自环，但有公共点。分别计算方案即可求出答案。时间复杂度为 $O(n+m)$ 。

定理 4.2. 对于无向图 G ，所有顶点的度都是偶数等价于图 G 有圈分解。圈分解即为用若干个圈（不重复经过顶点的回路）使图G的每一条边恰经过一次。

证明. 这一定理我们其实在一开始介绍的定理2.1中就已经证明了，只是因为章节的侧重不同没有将这一定理提出。该定理也可以描述成点度均为偶数等价于存在回路分解。该定理揭示了所有点度为偶数的等价性表述，虽然看似简单，但往往在解题中经常用到。

例题 4.2. 给定 n 个点， m 条边的无向图，边有黑白两种颜色。现在你可以进行若干次回路反色操作，每次操作从任意点出发，每经过一条边，将其颜色反转，最后回到起点。判断能否通过若干次操作，使这张图所有边都变成白色。若可以，求出最少操作次数。满足 $n, m \leq 10^6$ 。[77]

[76]题目来源：Codeforces Round #407(Div. 1). B

[77]题目来源：BZOJ 3706

解法. 给定的无向图记为 G 。我们事先决定每条边的经过次数，将每条边按照经过次数重复，构成图 G' 。图 G' 是由若干条回路构成的，所以图 G' 的所有点必为偶顶点。为了使每条黑边变为白边，每条白边仍是白边，黑边必然经过奇数次，而白边经过偶数次。因此，图 G 中黑边构成的子图必然所有点为偶顶点，此时图 G' 才可以分解成若干条回路。若图 G 中黑边构成的子图存在奇顶点，则一定无解。

我们如何求出最少的操作次数呢？因为图 G' 的每个连通分量都是欧拉图，操作次数就是图 G' 的连通分量数（不考虑孤立点）。我们要在黑边构成的子图中加入白边，减少连通分量数。最直接的办法就是对于图 G 中包含黑边的连通分量，我们让路径对于每条黑边经过一次，白边经过两次。其余连通分量不经过。最少操作次数就是图 G 中包含黑边的连通分量数。时间复杂度为 $O(n+m)$ 。

定理 4.3. 对于不存在欧拉回路的图 G ，若最少用 a 条路径将图 G 的每一条边经过一次，若最少在图 G 中加入 b 条有向边使之成为欧拉图，则 a 一定等于 b 。

证明. 这一定理与定理2.2证明方法相同。加入 a 条有向边使路径收尾相接，可以得到欧拉回路，推得 $a \geq b$ 。在图 G 加入 b 条边得到的欧拉回路中删除这 b 条边，就得到了 b 条路径，推得 $b \geq a$ 。因此， $a = b$ 。这一巧妙的转换往往可以使解题容易不少。

例题 4.3. 给定 n 个点， m 条边的无向图。求最少添加多少条无向边后，使得图存在从 1 号点出发又回到 1 号点的欧拉回路。满足 $n, m \leq 10^6$ 。 [78]

解法. 如果从构造加边的方案入手，就会发现既要考虑消除奇顶点，又要考虑将不同连通分量连通，十分麻烦。我们就转换角度，考虑这个图需要几条路径才能使每一条边经过一次。不包含奇顶点的连通分量一定存在欧拉回路，计一条路径。包含奇顶点的连通分量，需要奇顶点个数除以 2 条路径。记奇顶点个数为 a ，不包含奇顶点的连通分量（不考虑孤立点）个数为 b 。路径总数为$\frac{a}{2}+b$ 条。我们还需要对原图为欧拉图， 1 号点为孤立点的情况做特殊处理。其余情况的答案为 $\frac{a}{2}+b$ 。时间复杂度为 $O(n+m)$ 。

5 欧拉图的生成问题

在解决实际的应用问题时，我们需要先在欧拉图性质的引导下，把问题转换为图论模型。再借助一些其他知识与算法将图论模型生成出我们所需要的欧拉图。在本节中，我们将通过几个例子介绍在欧拉图生成问题中的思路与技巧。

[78]题目来源：VK Cup 2012 Finals, Practice Session. C

5.1 De Bruijn序列

问题 5.1. 求出一个 2^n 位环形 0/1 串，满足所有 2^n 个长度为 n 的子串恰为所有 2^n 个的 n 位的 0/1 串。

解法. 每次将一个子串（顺时针）移动到相邻的下一个子串，相当于将原来的子串去掉最左边一位，并在最后加上 0 或者 1 。于是对于 0/1 串 $x_1x_2...x_n$ ，它的下一个子串可以变成$x_2x_3...x_n0$ 或者 $x_2x_3...x_n1$ 。

一个容易想到的思路是，将所有的0/1串看作 2^n 个点，每个点连出有向边 $(x_1x_2...x_n, x_2x_3...x_n0)$ 与 $(x_1x_2...x_n, x_2x_3...x_n1)$ 。此时需要找到一条路径经过所有点恰一次，即哈密尔顿路。而求解哈密尔顿问题是是十分困难的。那么这个方法无法解决此问题，我们需要改变模型。

在之前的模型中我们是用点来表示 0/1 串，所以需要经过每一个点恰一次。如果我们用边来表示每一个 0/1 串呢？我们构造一个包含 2^{n-1} 个点和 2^n 条有向边的图。每个点表示一个 $n-1$ 位 0/1 串，即相邻两个子串相同的 $n-1$ 位。每一条有向边表示一个 n 位 0/1 串 $x_1x_2...x_n$ ，连接 $(x_1x_2...x_{n-1}, x_2x_3...x_n)$ 。这样，该图中的一条回路就对应了一个环形串，而我们希望能求出一条经过所有边的回路。容易发现，每一个点一定恰有两条入边和两条出边，且每两个点之间一定可以到达。所以该图是欧拉图，一定存在欧拉回路，也就存在满足要求的环形 0/1 串。

这一问题还可以扩展， 2 可以替换成任意正整数 k ， 0/1 串替换成 k 进制串。对于给定的 n 与 k ，满足上述条件的长度为 k^n 的环形k进制串称作De Bruijn序列。

例题 5.1. 一个保险箱有 n 位数字密码，正确输入密码后保险箱就会打开，目前输入的最后 n 位数字与密码相同就算正确输入密码。请求出一个长度为 10^n+n-1 字典序最小的数字序列，满足依次输入序列的每一位数字，一定可以打开保险箱。满足 $n \le 6$ 。[79]

解法. 可以发现该本题与上述问题的本质是一样的，唯一不同的是环形串改为了数字串，需求从欧拉回路变为了欧拉路径。建模方式基本不变，我们需要 10^{n-1} 个点，表示相邻数字串的连接点。共 10^n 个需要被包含的数字串，每个数字串对应一条有向边。因为该图是欧拉图，我们需要求出该图的一条欧拉路径，满足产生的数字序列字典序最小。对于本题来说，就是求出字典序最小的欧拉回路。我们选择数值最小的点作为起点，使用求字典序最小的欧拉路径的方法即可求出满足题意的答案。时间复杂度为 $O(10^n)$ 。

5.2 混合图欧拉回路

问题 5.2. 给定包含有向边与无向边的弱连通图 $G = (V, E)$ ，判断图G是否为欧拉图。若存在，请求出一条欧拉回路。

[79]题目来源：POJ 1780

解法. 判定混合图是否为欧拉图不是一个容易的问题，如果能够把问题转化为熟悉的有向图就容易解决了。在混合图 G 中，为每一条边 (u, v) 确定一个方向，就得到了一个有向图。如果该有向图每个点的入得均等于出度，那么该图一定存在欧拉回路。

而随意定向显然不能满足度数要求，但我们可以在此基础上进行调整。我们先对每个点 u 求出出度减去入度的值 $s(u)$ 。反转一条已指定方向 $u \to v$ 的无向边 (u, v) ，会使 $s(u)$ 减小 2 ， $s(v)$ 增大 2 。最终我们希望使每个点入度等于出度，即对于每个点 u ，满足 $s(u) = 0$ 。

这一问题可以看做多源汇的最大流模型。把 $s(u)$ 大于 0 的点看做供给，而 $s(u)$ 小于 0 的点看做需求，每条边有容量上限，我们可以通过网络流算法求解。具体的建图方式如下：对于点 u ，若 $s(u) > 0$ ，则从源点向 u 连容量为 $|\frac{s(u)}{2}|$ 的边；若 $s(u) < 0$ ，则从u向汇点连容量为 $|\frac{s(u)}{2}|$ 的边。对于每条无向边，按照定向的方向连边，容量为 1 。求出该图源点到汇点的最大流。若在源点连出的边中存在未满流的边，根据最大流的性质，说明不存在方案使得所有点入度等于出度，故原图不是欧拉图。

若原图为欧拉图，若无向边 (u, v) 对应的边在最大流中流量为1，则将原来指定的方向 $u \to v$ 改为 $v \to u$ 。这样，我们得到了图 G 的欧拉定向，满足所有点的入度等于出度。利用有向图欧拉回路生成算法即可求出图 G 的欧拉回路。

5.3 中国邮递员问题

问题 5.3. 给定**有向带权**连通图 $G = (V, E)$ ，求出一条总边权和最小的回路，使得经过每一条边至少一次。

解法. 这道题目看起来比较复杂，我们可以先考虑一些简单情况。如果给定的图 G 是欧拉图，显然不需要重复走任意一条边，欧拉回路就是符合要求的路线。如果图G为半欧拉图，存在从 v_1 到 v_2 的一条欧拉路径 P ，那么我们需要加入一些重边使得图中的所有点入度等于出度。最优方案是找到一条从 v_2 到 v_1 的最短路 Q ，将 P 与 Q 拼接就得到了一条符合要求的路线。

通过以上的分析可以看出，本题的实质是在图 G 中增加一些重复边，使新图的每个点入度等于出度，并且增加的重复边总边权和最小。我们先对每个点 u 求出入度减去出度的值 $s(u)$ 。如果将一条边 (u, v) 重复一次，就会使得 $s(u)$ 减小 1 ，而 $s(v)$ 增加 1 ，同时产生边权的代价。这就是一个多源汇的最小费用最大流模型。具体的建图方式如下：对于点 u ，若$s(u) > 0$ ，则从源点向 v 连容量为 $|s(u)|$ ，费用为 0 的边；若 $s(u) < 0$ ，则从 u 向汇点连容量为 $|s(u)|$ ，费用为 0 的边。对于每条图 G 中的有向边 (u, v) ，则从 u 向 v 连一条容量为正无穷，费用为该边边权的边。求出该图源点到汇点的最小费用最大流。若在源点连出的边中存在未满流的边，则图 G 不存在每条边至少经过一次的回路。

若存在方案，图G的边权和加上求出的最小费用就是该方案的总边权。对于每条边 (u, v) ，对应的边在最大流中的流量，就是该边需要额外重复的次数。可以利用有向图欧拉回路

算法求出符合要求的路径。

我们可以对应地给出该问题的无向图版本，可以通过同样的方式分析此题。

问题 5.4. 给定**无向带权**连通图 $G = (V, E)$，求出一条总边权和最小的回路，使得经过每一条边至少一次。

解法. 无向图可以类比有向图的解法。我们要在图 G 中增加一些重复边，使新图的每个点都成为偶顶点，并且增加的重复边总边权和最小。若图中有 $2k$ 个奇顶点，则我们要加入 k 条以奇顶点为端点路径。每个点最多只会为一条路径的端点，否则我们可以将两条路径合并为一条。每一条路径必然选取两个端点间的最短路。因此，我们需要将 $2k$ 个奇顶点分为 k 对，使得每对点的最短路长度之和最小。我们要求出一个最小权完美匹配，这可以通过一般图最小权完美匹配算法求解。

6 欧拉图相关的计数

计数问题是图论中一类重要的问题。欧拉图，代表了点度为偶以及连通这两个条件。两个简单的条件却产生了很多有趣的计数问题。在本节的介绍过程中，我们将会利用上文中得到的欧拉图的相关性质，构造有利于我们求解的模型，最后使用我们熟知的方法得到答案。我们将通过几个经典的例子，探究欧拉图相关的计数问题的常用方法，并提出一些有用的结论。

6.1 欧拉图计数

问题 6.1. 给定 n，求包含 n 个点的所有点度为**偶数**的有标号简单无向图个数。

解法. 为了方便表示，记 s_n 表示 n 个点所有点度为偶数的有标号简单无向图个数。除去 n 号点以及 n 号点关联的边，此时其余这 $n-1$ 个点中必然有偶数个奇顶点，那么 n 号点就必须与这偶数个奇顶点各连一条边。我们只需求 $n-1$ 个点包含偶数个奇顶点的简单无向图的个数。而任意一个无向图中奇顶点的个数一定是偶数，s_n 就是 $n-1$ 个点简单无向图的数量，$n-1$ 个点之间有 C_{n-1}^2 个点对可以连边，所以 $s_n = 2^{C_{n-1}^2}$。

问题 6.2. 给定 n，求包含 n 个点的有标号简单连通无向**欧拉图**个数。

解法. 记 f_n 为问题所求的 n 个点的有标号简单连通无向欧拉图个数。同时处理连通与度为偶数两个约束较为困难，但我们在前面已经解决了度为偶数的问题。接下来，我们利用容斥原理的思路解决连通这一限制。f_n 等于 s_n 减去不连通的方案数。而不连通的方案数可以通过枚举 1 号点所在连通分量大小计算。当连通分量大小为 i 时，该连通分量内点的集

合有 C_{n-1}^{i-1} 种不同组合方式，这些点之间的连边方法有 f_i 种，剩余点之间就不需要保证连通性了，有 s_{n-i} 种连边方法。公式如下：

$$f_n = s_n - \sum_{i=1}^{n-1} C_{n-1}^{i-1} f_i s_{n-i}$$

运用该递推公式可以在 $O(n^2)$ 的时间内求出答案。当然，也可以运用CDQ分治+FFT或是多项式求逆等优化技巧得到 $O(n\log^2 n)$ 或 $O(n\log n)$ 的更高效率的解法。

6.2 欧拉子图计数

问题 6.3. 给定一个**无向连通**图 $G=(V,E)$ ，求有多少个支撑子图 $G'=(V,E'),E'\subseteq E$ ，满足每个顶点的度都是偶数。

解法. 这个图论问题不好直接入手，我们可以将这个问题可以转化为代数问题。用变量表示每条边选或不选，再把度的约束转化为等式，这样可以列出一组方程描述此问题。对于标号为 i 的边，这条边是否出现在子图中用 0/1 变量 x_i 表示。对于点 v ，与 v 关联的边分别为 $e_1,e_2,...,e_d$ ，这些边在子图中出现的条数必为偶数，得到 $x_{e_1}\text{ xor }x_{e_2}\text{ xor ... xor }x_{e_d}=0$ ，其中 xor 表示异或运算。由此我们得到了 n 个等式， m 个变量的异或方程组。通过高斯消元算法，我们可以求出该方程组的自由元的数量s ，则满足题意的子图数量为 2^s 。这一解法的时间复杂度为 $O(n^2m)$ 。

上述方法虽然已经解决了本问题，但在转化为代数模型的过程中，我们忽略了一些图自身的性质，例如图 G 是连通的，每一条边所代表的变量只在方程组中出现了两次。那么，我们能否利用这些性质得到更简便的解法呢？

我们任意求出图 G 的一棵生成树，将该生成树上的边对应的变量作为主元，此时非树边所对应的变量恰为自由元。我们可以找到这一过程的组合意义。依次决定每条非树边是否选择，容易发现，对于一组非树边的选法，树边只有唯一的选法。我们任意指定一个点为根，按照从叶子到树根的顺序依次决定每个点与其父亲间的树边是否选择。所以，一组非树边变量的取值，就对应了一组方程的解。

事实上，若把一条非树边与其两端点在树上的路径看做该树边所代表的环，将所有选择的非树边代表的环异或（一条边出现奇数次则选择，否则不选择），就得到了一个满足条件的子图。我们可以很容易表示出满足条件的子图的数量为 2^{m-n+1} 。

我们还可以将该问题扩展到一般的无向图：

问题 6.4. 给定一个**无向**图 $G=(V,E)$ ，求有多少个支撑子图 $G'=(V,E'),E'\subseteq E$ ，满足每个顶点的度都是偶数。

解法. 容易发现，不同的连通分量之间是独立的。若图的连通分量数量为 c ，对每一个连通分量使用上述公式，得到满足顶点为度为偶数的支撑子图数量为 2^{m-n+c} 。可以发现这一

数值与每一连通分量内部边的连接方式无关，在 $O(n+m)$ 的时间内就可以求出 c ，计算出本题的答案。

6.3 欧拉回路计数

问题 6.5. 给定一个**有向**欧拉图 $G=(V,E)$ ，求以 1 号点为起点的欧拉路径的数量。

解法. 我们先来提出一种构造方案。找到一棵以 1 号点为根的内向树（即每个点有唯一的一条路径到达 1 号点），对于一个点的所有不在树上的出边指定一个顺序。接下来，我们来证明上述的方案与欧拉路径一一对应。

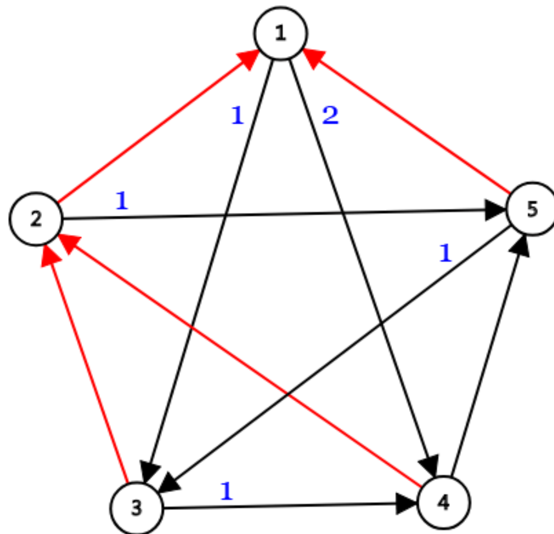


图4：一个构造方案的例子

先证明一个方案唯一对应一条欧拉路径。对于一个方案，我们从 1 号点出发，对每一个点，按照非树边指定的顺序走，若所有的非树边都走过，则走树边。这样走出了一条路径，我们现在来证明，这条路径就是一条欧拉路径。回顾Fleury算法的执行过程，只需证明走一条非树边 (u,v) 的时候 u 与 v 在树上是弱连通的。如果一个点有未访问的出边，则这个点出发的树边一定未访问过。走一条非树边 (u,v) 的时候， u 与 v 的树边都未访问过，它们的父亲节点出发的树边一定也未访问过。以此类推，它们到根节点的路径的每一条边都未访问过。因此，走一条非树边 (u,v) 的时候 u 与 v 在树上是弱连通的。按照上述方式得到的路径一定是欧拉路径。

再证明一条欧拉路径唯一对应一个方案。对于以 1 号点为起点和终点的欧拉路径，除 1 号点外的每个点最后访问的出边是“树边”，其余出边按照访问次序确定顺序，便得到一个上述方案。需要证明得到 $n-1$ 条“树边”中不存在环。我们可以反证，假设“树边”形成了环，由于 1 号节点不选择树边，环上不存在一号节点。我们任选环上一点出发，沿着“树

边”走，最后又回到了这个点。因为树边为最后访问的一条边，因此欧拉路径终止于该点了，这与欧拉路径的终点为 1 号点矛盾。

这样我们就证明了方案与欧拉路径一一对应。接下来，我们来考虑“方案”的数量。对于图 G ，记 T_i 为以 i 为根的内向树的数量， d_i 为 i 号点的出度。对于一棵内向树，与1 号点关联的出边有 $d_1!$ 种，其余节点 i 对非树边指定顺序，有 $d_i!$ 种。以 1 号点为起点和终点的欧拉路径的数量为：

$$T_1 d_1! \prod_{i=2}^{n} (d_i - 1)!$$

其中 T_1 可以利用矩阵树定理求出。即出度矩阵减去邻接矩阵去掉第一行第一列后行列式的值。时间复杂度为 $O(n^3)$ 。

问题 6.6. （BEST 定理）给定一个有向欧拉图 $G = (V, E)$ ，求欧拉回路的数量。

解法. 我们可以通过同样的方式求出不同欧拉回路的数量，即不考虑起点，路径循环同构。为了避免重复计算同一种欧拉回路，我们需要将一个欧拉回路转化为唯一的一条欧拉路径，即从 1 号点出发的标号最小的边作为第一条访问边。我们就能够得到不同的欧拉回路的数量：

$$T_1 \prod_{i=1}^{n} (d_i - 1)!$$

这一数量关系由de Bruijn、van Aardenne-Ehrenfest、Smith和Tutte四人提出，称为BEST定理 [5] 。从这个公式中，我们还可以看出有向欧拉图具有 $T_1 = T_2 = ... = T_n$ 的性质。

问题 6.7. 给定一个有向半欧拉图 $G = (V, E)$ ，求欧拉路径的数量。

解法. 对于不好处理的半欧拉图问题，最直接的办法就是通过加边转化为欧拉图问题。我们在半欧拉图中添加一条有向边，将半欧拉图 G 变为欧拉图 G' 。那么，图 G' 中的欧拉回路就与图 G 中的欧拉路径一一对应。利用BEST定理给出的公式，计算图 G' 的欧拉回路的数量，就可以算出图 G 的欧拉路径的数量。

那么我们能否求出无向欧拉图中的欧拉回路数量呢？遗憾的是这一问题十分困难，已经被证明为`#P-complete`问题，目前没有多项式复杂度的解法。

7 总结

欧拉图问题是图论中十分重要的一类问题。本文从欧拉图本质分析着手，介绍了欧拉图的判定方法和欧拉回路的生成算法。在分析性质和设计算法的过程中，图的连通性与顶点的度起到了关键的作用，这两个要素是我们分析欧拉图问题的基础。以这两个要素入手，我们总结出了更一般的结论与解题思路。

运用欧拉图模型可以解决一些应用问题。本文分析了两种常见的解题思路。一是对于表面上看似是哈密尔顿问题的题目，不妨考虑一下将边与点的关系对调，转化为欧拉图问题。二是对于构造或贪心难以处理的欧拉图生成问题，可以考虑将约束转化为网络流模型或其它经典问题。模型转化往往是这类题目的突破口。

欧拉图的计数问题与生成问题类似，需要灵活运用模型转化，基于欧拉图性质分析，构造组合意义，抽象出代数模型，并且善用计数技巧进行解决。

总而言之，深入理解欧拉图相关性质，灵活运用模型转化，巧妙借助其他知识与算法，是解决欧拉图相关问题的关键。希望本文可以起到抛砖引玉的作用，使更多人了解并继续发掘欧拉图问题的奥秘。

致谢

- 感谢中国计算机学会提供学习和交流的平台。
- 感谢北师大实验中学的胡伟栋老师多年来的关心与指导。
- 感谢为本文审稿和提出建议的同学们。
- 感谢所有对我有过帮助的老师和同学。

参考文献

[1] 仇荣琦，《欧拉回路性质与应用探究》，IOI2007国家集训队论文

[2] 熊斌, 郑仲义，《数学奥林匹克小丛书：图论》，华东师范大学出版社

[3] Herbert Fleischner, Eulerian Graphs and Related Topics. Part 1, Vol.2

[4] 欧拉路径的英文维基百科，
`https://en.wikipedia.org/wiki/Eulerian_path`

[5] BEST定理的英文维基百科，
`https://en.wikipedia.org/wiki/BEST_theorem`