2014 年信息学奥林匹克

中国国家队候选队员论文集

教练：胡伟栋

中国计算机学会

目录

MSS 命题报告

东营市胜利第一中学 王子昱

1 题目描述

1.1 问题描述

平面上有N个点，每个点有权值，开始时每个点属于一个不同的集合。不妨设点P_i属于集合S_i。维护数据结构支持以下四种操作：

- "Merge x y"：将集合S_y中的点移动到S_x中；
- "Split i d v"（$d \in \{0,1\}$）：创建新集合S_{c+1}，S_{c+2}，之后将集合S_i中的所有点中，第d维坐标不超过v的移动到集合S_{c+1}中，超过v的移动到集合S_{c+2}中，其中c为现有集合数（包括之前合并和分割中产生的空集）；
- "Query i"：查询集合S_i中点权值的最大值，最小值及和；
- "Add i d"：将集合S_i中所有点的权值增加d。

方便起见，对于$d \in \{0,1\}$，输入的所有点的第d维坐标不重复。所有操作涉及的集合非空。

1.2 输入格式

第一行一个整数N表示点数。
接下来N行，每行3个整数分别表示P_i的坐标与权值。
接下来一个整数Q表示操作数。
接下来Q行，每行描述一个操作，格式如上所述。

1.3 输出格式

对于每个Q操作，输出三个整数，依次表示询问集合权值的最大值，最小值与和。

1.4 样例输入

```
5
4 7 3
18 16 2
14 13 1
10 2 10
3 8 5
11
Merge 2 3
Merge 4 5
Split 2 1 13
Query 4
Query 6
Add 6 2
Query 6
Merge 6 4
Split 6 0 10
Query 9
Split 8 0 3
```

1.5 样例输出

```
10 5 15
1 1 1
3 3 3
3 3 3
```

1.6 数据规模和约定

数据分为5类：

- A类：占10%，$N, Q \leq 500$；
- B类：占20%，数据中没有分割操作；
- C类：占10%，在第一次分割操作出现之后，不再有合并操作；
- D类：占20%，所有点满足$x_i = y_i$；
- E类：无任何特征。

B到E类中均有$N \leq 50000, Q \leq 100000$。
题目的时间限制为3s；内存限制为512MB。

2 题目分析

2.1 简单的算法

2.1.1 模拟

对于A类数据，我们可以直接模拟，时间复杂度$O(NQ)$。

2.1.2 启发式合并

对于B类数据，问题可以用启发式合并解决：用平衡树维护每个集合。合并两个集合时，将元素数较小的一个集合中的所有点暴力插入较大的集合的平衡树中。由于每个点被重新插入时，它所在的集合的点数至少增加一倍，所以每个点至多被插入$O(\log N)$次，算法时间复杂度为$O(N \log^2 N + Q)$。

2.1.3 “启发式分割”

对于C类数据，我们开始时用启发式合并处理所有的合并操作。合并结束后，如果我们能在$O(T \times [\text{分割后较小集合的大小}])$的时间内完成一次分割[1]，算

[1]T是任意与“较小集合大小”无关的量。

法的复杂度就是$O(NT\log N+Q)$的了，因为这一算法的逆过程与启发式合并相同。

如果每次只按一维坐标分割，只需要以该维坐标为序建立平衡树，之后可以容易地完成分割操作；回到本题，我们可以对每个集合建立两棵平衡树，分别以横纵坐标为序。分割时，我们用分割所用维的平衡树得到分割后较小的集合，在另一平衡树中将对应的点删除并重建。于是$T=O(\log N)$，算法复杂度$O(N\log^2 N+Q)$。

2.2 一维算法

在D类数据中，所有点的横纵坐标分别相等。由于对纵坐标的分割可以转化为对横坐标的分割，可以认为我们维护的是一维的数的集合。

2.2.1 基于平衡树

我们尝试继续用平衡树维护每个集合中的值。

平衡树支持$O(\log N)$的分割和（对值的范围不相交的集合的）合并；考虑值的范围可以相交时的合并。

考虑以下算法[2]：

考查上述算法的复杂度。接下来我们要说明，Q_1次分割操作和Q_2次合并操作的时间代价为$O((N+Q_1)\log^2 N)$。

定义

$$W(X,Y)=\sum_{a\in X,b\in Y}[|rank(X\cup Y,a)-rank(X\cup Y,b)|=1]$$

，其中$rank(S,x)$为x在S中的大小排名。不严格地说，$W(X,Y)$表示将X和Y合并，并将原先属于同一集合的点压缩为一段后的段数（减1）。于是一次合并的复杂度为$O(W(X,Y)\log N)$。

[2]代码中`merge0(a, b)`在a，b范围不相交时，返回平衡树a和b的并；
`splitL(a, v)`返回a中值不超过v的元素构成的平衡树；
`splitR(a, v)`返回a中值超过v的元素构成的平衡树。

Algorithm 1 merge(x, y): return the union of x and y

```
if x = nil then
    return y
else if y = nil then
    return x
else
    if x.min > y.min then
        swap(x, y)
    end if
    return merge(merge0(splitL(x, y.min), y), splitR(x, y.min))
end if
```

定义

$$g_0(x) = rank(U, x) - rank(U, prev(S, x))$$
$$g_1(x) = rank(U, succ(S, x)) - rank(U, x)$$

其中S为x所属的集合；U为所有集合的并集，$prev(S, x)$和$succ(S, x)$分别表示x在集合S中的前趋和后继，如果不存在则为$\min(U)$或$\max(U)$。

最后，定义第i次操作后，数据结构的势能

$$\Phi_i = c \log N \sum_{x \in U} (\log g_0(x) + \log g_1(x))$$

，其中c是之后确定的常数。于是$\Phi_0 = cN \log^2 N$。

分割操作的实际代价为$O(\log N)$；由于只有一个g_0和一个g_1增加，带来的势能增量不大于$2c \log^2 N$；

考虑对X和Y的合并。定义

$$P_X = \{x \in X \mid prev(X, x) \neq prev(X \cup Y, x)\} \cup \{\min(X)\}$$
$$S_X = \{x \in X \mid succ(X, x) \neq succ(X \cup Y, x)\} \cup \{\max(X)\}$$

。于是将X和Y合并，并将原先属于同一集合的点压缩为一段后，P_X包含来自X的所有段的左端点，S_X包含了来自X的所有段的右端点。类似地定义P_Y和S_Y。于是$|P_X| + |P_Y| = W(X, Y) + 1$。

于是对所有满足$a \in S_X, b \in P_X, rank(S_X, a) + 1 = rank(P_X, b)$的有序对$(a, b)$，$g_0(a)$和$g_1(b)$中总有一个至少减少到之前值的1/2；对集合$Y$同类。每对$(a, b)$对势能增量的贡献不大于$-c \log N$，而这样的有序对共有$W(X, Y) - 1$对，因此合并带来的势能增量不大于$-c(W(X, Y) - 1) \log N$。

由此可得，通过选择适当的c，可以得到所有操作的代价之和不超过

$$\begin{aligned}&\Phi_0 + \sum_{i=1}^{Q_1}(2c \log^2 N + O(\log N)) + \\ &\sum_{i=1}^{Q_2}(-c(W(X_i, Y_i) - 1) \log N + O(W(X_i, Y_i) \log N)) \\ =&O((Q_1 + N) \log^2 N)\end{aligned}$$

。[3]

2.2.2 基于线段树

考虑用动态存储的线段树维护集合。动态线段树支持合并操作；对坐标离散化后，分割操作的复杂度为$O(\log N)$。接下来我们要说明，Q_1次分割操作和Q_2次合并操作的时间代价为$O((N + Q_1) \log N)$。

定义$size(T)$为集合T对应的动态线段树的结点数，于是合并集合X, Y的时间代价为$O(size(X) + size(Y) - size(X \cup Y))$。[4]定义第$i$次操作后，数据结构的势能

$$\Phi_i = c \sum_{S \in I_i} size(S)$$

，其中I_i为包含第i次操作后所有集合的集合族，c为之后确定的常数。

对于分割操作，其实际代价为$O(\log N)$，操作后势函数的增量不超过$c \log N$；对于对集合X, Y的合并操作，其实际代价为$O(size(X)+size(Y)-size(X \cup Y))$，势函数的增量为$-c(size(X)+size(Y)-size(X \cup Y))$；数据结构的初始势能$\Phi_0 = cN \log N$。

[3]由于$Q_2 < N + Q_1$，我们可以消去O记号中可能存在的$Q_2 \log N$项。

[4]考虑合并X, Y时访问的线段树上的结点。除去最下一层结点外，其它的每个结点都同时在X, Y中存在，在合并后都会有一个被删去。

通过选择适当的c，我们可以得到所有操作的总代价不超过

$$\Phi_0 + \sum_{i=1}^{Q_1}(O(\log N) + c\log N)+$$
$$\sum_{i=1}^{Q_2}(O(size(X) + size(Y) - size(X \cup Y)) - c(size(X) + size(Y) - size(X \cup Y)))$$
$$= O((N + Q_1)\log N)$$

。

至此，D类数据得到解决，算法复杂度$O((N + Q)\log N)$。

2.2.3 替代算法

这是一个经典问题，参考文献[3]中给出了基于biased skip list 的均摊$O(\log N)$算法，不过分析和实现都较为复杂。有兴趣的读者可以自行参阅。

2.3 通用算法

考虑一维算法适用的范围。

定义点集上的偏序关系≤，使得$(x_1, y_1) \le (x_2, y_2)$等价于$x_1 \le x_2$且$y_1 \le y_2$。于是在所有集合的并$U$关于关系≤是一个链时，一维算法是适用的：由于$y$坐标关于$x$坐标单调递增，利用$y$坐标进行的划分都可以转化为利用$x$坐标的划分。进一步地，当$U$是一个反链，也就是$y$关于$x$单减时，一维算法也是适用的。

在一般的情况下，我们将所有集合的并U划分为若干集合$U_{1..k}$，使得在上述偏序意义下U_i或者为链，或者为反链。对每个i，令$I_i = \{s \cap U_i \mid s \in I\}$，其中$I$为包含当前所有集合的集合族。于是合并、分割和查询都可以在每个I_i上分别处理，而每个I_i可以用一维算法维护。

这一算法的复杂度为$O((N + Q)k\log N)$。接下来我们要证明$k = O(\sqrt{N})$。

引理1. *大小为N的偏序集P中，或者存在大小为$L = \lceil\sqrt{N}\rceil$的链，或者存在大小为L的反链。*

证明. 假设P中不包含长度为L的链。根据Dilworth定理，P可以被划分为不超过$L - 1$个反链，于是其中存在大小至少为$\lceil\frac{N}{L-1}\rceil \ge L$的反链。 □

引理2. 大小为N的偏序集P可以被划分为$k = O(\sqrt{N})$个链或反链。

证明. 考虑递归完成划分，每次从P中删去最长的链或反链。于是大小为N的集合的划分数上界

$$T(N) = T(N - \sqrt{N}) + 1 \leq T(N/2) + O(\sqrt{N}) = O(\sqrt{N})$$

。 □

至此，问题得到解决，算法复杂度$O((N + Q)\sqrt{N}\log N)$。

3 总结

本题涉及到的知识只有基础数据结构和均摊分析，都属于NOI选手的知识范围。然而得出通用算法对选手的分析能力有一定要求。

对于最后40分，虽然也有其它解法存在，但作者认为这里给出的划分偏序集的方法更有趣，也更有启发性。当题目涉及的对象满足其它偏序关系（例如区间的包含关系）时，类似的做法值得考虑。

参考文献

[1] 刘汝佳, 黄亮, 《算法艺术与信息学竞赛》, 清华大学出版社.

[2] Richard A. Brualdi, “Introductory Combinatorics”, 5th Ed, Prentice Hall.

[3] John Iacono, Ozgur Ozkan, “Mergeable Dictionaries”.

矩阵命题报告

长沙市长郡中学 余行江

1 试题

1.1 题目描述

给你一个 $N*N$ 的非负实数矩阵 A，你需要求出一个 $N*N$ 的实数矩阵 B，使得 B 的第 i 行的元素值和 + 第 j 列的元素值和 $\geq A_{i,j}$。

在此前提下，最小化 B 的元素值和。

1.2 输入格式

你有二个任务（详见数据范围）。

对于每个任务，第一行一个整数 N。

接下来 N 行，每行 N 个非负实数，描述给出的矩阵 A。

1.3 输出格式

二行。每行输出一个保留 4 位的非负实数，分别描述二个任务的答案。

1.4 输入样例

```
3
1 7 3
3 2 1
5 4 1
2
1 1
1 1
```

1. 5 输出样例

6.5000
1.0000

1. 6 子任务与约定

Task0 : A 内的所有元素值相等。
Task1 : 任务一、二的 $N \leq 50$ 。
Task2 : A 的元素随机。
Task3 : 任务二的 $N \leq 50$ 。
Task4 : 任务一的 $N \leq 300$，任务二的 $N \leq 80$ 。
对于任务一，构造的矩阵 B 没有任何限制。
对于任务二，构造的矩阵 B 的所有元素值应非负。
所有输入的实数均保留 3 位小数，且绝对值≤ 1000

1.7 时间限制

3s。

1.8 空间限制

512MB。

2 解题思路

2. 1 Task0

容易知道，矩阵 B 内的所有元素值也都相同。
可以直接手算答案。

2. 2 建立线性规划模型

根据题意，列出线性规划模型：

$$\begin{cases} \sum_{i=1}^{N} B_{i,1} + \sum_{j=1}^{N} B_{1,j} \geq A_{1,1} \\ \sum_{i=1}^{N} B_{i,2} + \sum_{j=1}^{N} B_{1,j} \geq A_{1,2} \\ \cdots \\ \sum_{i=1}^{N} B_{i,1} + \sum_{j=1}^{N} B_{2,j} \geq A_{2,1} \\ \sum_{i=1}^{N} B_{i,2} + \sum_{j=1}^{N} B_{2,j} \geq A_{2,2} \\ \cdots \\ \sum_{i=1}^{N} B_{i,n} + \sum_{j=1}^{N} B_{n,j} \geq A_{N,N} \end{cases}$$

$$\min \ z = \sum_{i=1}^{N} \sum_{j=1}^{N} B_{i,j}$$

这样建模的变量个数是 $O(N^2)$ 的，需要优化。考虑题面即提示了可以用行和与列和作为变量，不妨尝试。

定义 R_i 表示第 i 行的权值和， C_j 表示第 j 行的权值和。则我们可以得到一个新的线性规划模型：

$$\begin{cases} R_1 + C_1 \geq A_{1,1} \\ R_1 + C_2 \geq A_{1,2} \\ \cdots \\ R_2 + C_1 \geq A_{2,1} \\ R_2 + C_2 \geq A_{2,2} \\ \cdots \\ R_N + C_N \geq A_{N,N} \\ \sum_{i=1}^{N} R_i - \sum_{j=1}^{N} C_j = 0 \end{cases}$$

$$\min \ z = \sum_{i=1}^{N} R_i + \sum_{j=1}^{N} C_j$$

相比原来的，这个模型的变量缩减为 $O(N)$ 个。但是，这个模型也带来了一个问题：一定存在一个矩阵对应得到的解？

对于任务一，由于权值可以任取，所以一定存在。

对于任务二，考虑当前的限制条件为：

$$\sum_{i=1}^{N} R_i = \sum_{j=1}^{N} C_j \tag{1}$$

$$R_i \geq 0 \tag{2}$$

$$C_j \geq 0 \tag{3}$$

不妨任取一个 R_i，然后从 C 中抽取一部分值放到第 i 行中。由于 $\sum_{j=1}^{N} C_j \geq R_i$，故一定存在一种方案，使得抽取后，任意 $C_j \geq 0$。

可以发现，将 R_i 处理完毕后，剩下的子问题的限制条件不变。故一定存在一个矩阵对应上述模型得到的解。

2. 4 Task1

任务一、二的 $N \leq 50$，可以用单纯形等算法暴力解线性规划。

对于任务二，模型正好是线性规划的标准形式。

对于任务一，我们要将每个 R_i [1]拆成 $Ra_i - Rb_i$，并且 $Ra_i, Rb_i \geq 0$。

在随机数据下，单纯形的效率很好[2]。

如果实现得好[3]，也可以通过 $Task3$ 的数据。

时间复杂度：未知

2. 5 对偶原理

原线性规划似乎不好进一步处理了，尝试对偶转化。

2. 5. 1 任务一

可以预见答案矩阵的元素值的绝对值都不是很大。我们给每一个元素加上一个很大的值，利用 $Task0$ 的结论，就转变成了任务二。

[1] C_i 同理。

[2]甚至比正解还要快

[3]比如说加上链表剪枝

当然，这样做肯定会 TLE 。

直接用对偶原理：

$$\begin{cases} Y_{1,1}+Y_{1,2}+Y_{1,3}+...+Y_{1,N}+P=1 \\ Y_{2,1}+Y_{2,2}+Y_{2,3}+...+Y_{2,N}+P=1 \\ \cdots \\ Y_{1,1}+Y_{2,1}+Y_{3,1}+...+Y_{N,1}-P=1 \\ Y_{1,2}+Y_{2,2}+Y_{3,2}+...+Y_{N,2}-P=1 \\ \cdots \\ Y_{N,1}+Y_{N,2}+Y_{N,3}+...+Y_{N,N}-P=1 \\ Y_{1,N}+Y_{2,N}+Y_{3,N}+...+Y_{N,N}+P=1 \\ \\ Y_{i,j}\geq 0 \end{cases}$$

$$\max \quad z=0*P+\sum_{i=1}^{N}\sum_{j=1}^{N}A_{i,j}*Y_{i,j}$$

如果没有 P ，这就是一个很常见的二分图最大费用流的模型。

注意到，所有的限制条件都是取等号的。这意味着对于这张费用流， S 连出的边的容量和 = 连入 T 的边的容量和，我们可以直接得出 $P=0$。

之后就是简单的费用流问题了。

2. 6. 2 任务二 - 1

还是按照上述方法对偶：

$$\begin{cases} Y_{1,1}+Y_{1,2}+Y_{1,3}+...+Y_{1,N}+P\le 1 \\ Y_{2,1}+Y_{2,2}+Y_{2,3}+...+Y_{2,N}+P\le 1 \\ \cdots \\ Y_{1,1}+Y_{2,1}+Y_{3,1}+...+Y_{N,1}-P\le 1 \\ Y_{1,2}+Y_{2,2}+Y_{3,2}+...+Y_{N,2}-P\le 1 \\ \cdots \\ Y_{N,1}+Y_{N,2}+Y_{N,3}+...+Y_{N,N}-P\le 1 \\ Y_{1,N}+Y_{2,N}+Y_{3,N}+...+Y_{N,N}+P\le 1 \\ \\ Y_{i,j}\ge 0 \end{cases}$$

$$\max \quad z=0*P+\sum_{i=1}^{N}\sum_{j=1}^{N}A_{i,j}*Y_{i,j}$$

由于 $Y_{i,j}\ge 0$，可以发现 P 的取值范围为 $[-1,1]$。

这个模型的意义实际上是一个完全二分图的最大费用流。只不过我们可以给一边的所有点的出流量限制 $+P$，同时给另一边的所有点的入流量限制 $-P$。

直觉告诉我们，对于 P 在区间 $[-1,0],[0,1]$ 内，答案是一个单峰函数。

尝试证明？唔……似乎有些困难[4]。

不妨先打个表试一试吧。于是乎，我们发现这确实是个单峰函数。

令这个单峰函数为 $f(P)$，其有如下的一些性质：

1、当且仅当 $f'(P)=0$ 时，存在多个 P，满足 $f'(P)$ 相等。也就是说这个函数可以三分。

2、在 $[-1,1]$ 内的单峰函数，我们只需要做一次三分就可以了。

可以解决 $Task4$。

2.6.3 任务二 - 2

这是我最初的想法。虽然复杂度较高，但我觉得思想还是很巧妙的。

[4]证明仍在思考中……

注意到原问题的答案有明显的可二分性。不妨直接二分答案，假设答案为 K。

这样，对于原来的限制：

$$\sum_{i=1}^{N} R_i - \sum_{j=1}^{N} C_j = 0$$

变成了：

$$\sum_{i=1}^{N} R_i = K \tag{4}$$

$$\sum_{i=1}^{N} C_i = K \tag{5}$$

再对偶，得到：

$$\begin{cases} Y_{1,1} + Y_{1,2} + Y_{1,3} + ... + Y_{1,N} + A \le 1 \\ Y_{2,1} + Y_{2,2} + Y_{2,3} + ... + Y_{2,N} + A \le 1 \\ \cdots \\ Y_{1,1} + Y_{2,1} + Y_{3,1} + ... + Y_{N,1} + B \le 1 \\ Y_{1,2} + Y_{2,2} + Y_{3,2} + ... + Y_{N,2} + B \le 1 \\ \cdots \\ Y_{N,1} + Y_{N,2} + Y_{N,3} + ... + Y_{N,N} + A \le 1 \\ Y_{1,N} + Y_{2,N} + Y_{3,N} + ... + Y_{N,N} + B \le 1 \\ \\ Y_{i,j} \ge 0 \end{cases}$$

$$\max \quad z = K * A + K * B + \sum_{i=1}^{N} \sum_{j=1}^{N} A_{i,j} * Y_{i,j}$$

这个线性规划的流模型也很明显——可以用 K 的代价，给某一边的所有点的流量限制 $+K$。

根据对偶原理的性质，当且仅当该模型有极大解时，原问题无解。即二分的答案 K 小于正确答案。

显然，在该模型取极大解时，一定有 $A \to -\infty$ 及 $B \to -\infty$ 。

但我们当然不能直接在 $[-\infty, 1]$ 的范围中寻找 A 、 B 。可以预见到，当 A 、 B 足够小时，对于任意的 A ，每减少 ΔA ， B 应当减少的值 ΔB 是趋于一个定值的。

假设此时我们已知 A 、 B。

我们假设 $\Delta A \geq \Delta B$ 。不妨令 $\Delta A = 1$ ，然后求 ΔB 相对 ΔA 的值。明显， ΔB 在区间 $[0, 1]$ 内，并且 ΔB 是可以三分的——若 ΔB 过小，则 ΔA 有冗余；若 ΔB 过大，则 ΔA 不足以弥补 ΔB 的花费 $-\Delta B * K$ 。

对于 $\Delta A \leq \Delta B$ ，我们令 $\Delta B = 1$ ，再类似地做一次即可。

再考虑初始的 A 、 B 如何确定。明显， A/B 最后是趋近于 $\Delta A/\Delta B$ 的。那么我们只需要将 $\Delta A = 1$ 修改为 $\Delta A = P$ 即可，其中 P 足够大。

由于 K 越接近答案， A/B 要求的精度也就越高。明显，如果我们要精确地得到答案 K ，需要设定的 P 值是非常大的。

不过，注意到我们的答案只要求 4 位的精度。也就是说，我们的答案没有必要那么精确。 P 完全可以取比较小的值。

经过测试，发现在一般情况下，令 $P = N^2$ 即可得到最优解。

相对于算法一，这样做多了一个二分，故复杂度增加一个 log 。

不过很可惜，这个算法只能够勉强通过 $N \leq 50$ 的数据，也就是 $Task3$。

2. 7 Task2

在任务二中，对于大部分的随机数据， $P = 0$ 就是最优解了。有极少的数据[5] 会令 P 在 $[-1e-4, 1e-4]$ 内波动，基本可以忽略不计。

将任务一的答案输出到第二问即可。

0.1 其他

如果是一个 $N * M$ 的矩阵，当然也是可以做的。不过任务一的 P 就不为 0

[5]在我生成的 10000 组数据中，只有不到 50 组

了， $P = (M - N)/(N + M)$ 。

本题是一张完全二分图，如果用 *SPFA* 跑费用流，效率会比较低。需要用 *zkw* 或原始对偶来做。

单纯形在很多情况下的效率都很不错，接近 *zkw* 费用流的效率。但是如果充分发挥人的智慧，还是能卡掉的。

任务二的二分图有一定的特殊性。如果有人能去掉三分，或者是用其他的方法优化每次的费用流[6]，欢迎与我讨论。

3 总结

本题有很多的衍生版本。不过其他版本要么题面太复杂，要么我没有找到比较优秀的算法。如果有关于这类问题的更好想法的话，欢迎与我联系。

本题的灵感来自于我曾经出过的一道题 *Chessboard* 。初期模型是在一个棋盘上放车，要求每个格子只能放一个，并且格子 (i, j) 需要被至少 $W(i, j)$ 个车控制。这个模型虽然描述很简单，不过似乎是 *NP* 的——它要求一个线性规划的整数解，但这个线性规划的最优解基本上都是实数。最后我不断地削弱限制条件[7]，最终得到了本题。

本题考查了选手对线性规划和网络流建模的基础知识，并且运用了对偶原理。算法一的性质很优美，而证明还有待思考。对于算法二，套上了二分与三分，并用费用流来判断是否有解。我认为这是一个非常不错的思想，将最优化问题转变为判断可行性问题，并且实现在网络流上。只可惜，对于本题它并不是最优秀的算法。

希望本题能对大家有所启发，进一步地认知线性规划和网络流问题。

4 致谢

感谢 *CCF* 提供学习和交流的平台。

感谢向期中老师对我的指导。

感谢帮助过我的同学们。

[6]比如说先预处理出 $P = (M - N)/(N + M)$ 时的最大费用流，每次三分 P 时再利用其快速求值

[7]削弱过程惨不忍睹....

-- 空 --

多变的多边形

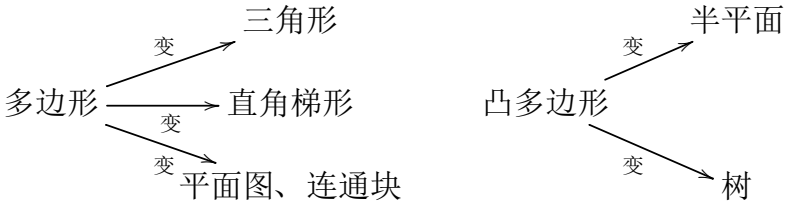
绍兴市第一中学 董宏华

摘 要

本文总结了多边形的常用变法，生动展示了多边形的多变，并结合例题演示了各种变法的作用，为解决与多边形有关的题目提供了思路。同时结合其中一种变法深入研究了多边形内部点信息统计问题，得到了一个较优秀的算法，为更高效地解决该问题提供了途径。

1 引言

近年来，多边形在各省省选及各类OI竞赛，以及ACM WorldFinals等比赛中频繁出现，而且出现形式复杂，变化多样。



作者对其进行了一些总结，并呈现如下：

在本文第二节，介绍了一些之后要用到的简单的基础知识。

在本文第三节，列举了多边形的多种变法，同时配备了例题，便于读者理解和拓展。

在本文第四节，由GIS（地理信息系统）引入多边形内部点信息统计问题，并由特殊到一般，由易到难地讨论了该问题，最终得到了一个复杂度较优秀的通用的算法。

2 基础

2.1 多边形定义

2.1.1 多边形

由三条及以上的线段首尾顺次连接所组成的平面封闭图形。

2.1.2 简单多边形

不相邻的边不相交的多边形。

2.1.3 凸多边形

所有内角均不超过 180° 的简单多边形。

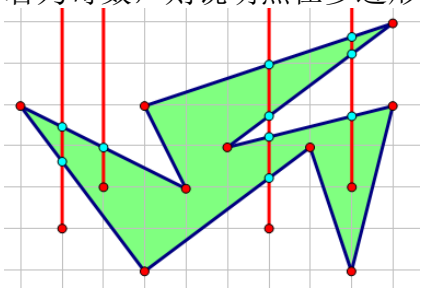
2.2 叉积

两个二维向量 $(x_1, y_1), (x_2, y_2)$ 叉积结果为$(x_1 \times y_2 - x_2 \times y_1)$，其表示的意义为两个向量所构成的平行四边形的有向面积。

2.3 射线法

用射线法判断一个点是否在多边形内。

以当前点为起点，向某一方向作一条射线，计算多边形与该射线的交点个数。若为奇数，则说明点在多边形内部。若为偶数，则点在外面。



为了计算方便，射线的方向可以选为正上方稍偏右（避免交点在多边形顶点上）。

该方法同样适用于会自交的多边形。

【例1】围豆豆[1]

试题大意

在 $n \times m$ 的网格中有 $D(D \leq 9)$ 个带权点，求一条从某点出发又回到该点的路径，使 其包住的点权和减去步数 最大。

分析

路径构成了一个（会自交的）多边形，判断每个带权点是否被包住就变为了点是否在多边形内的问题。

使用射线法来判断，每个带权点向上作射线，则经过与该射线有交的边时，该带权点的状态取反。

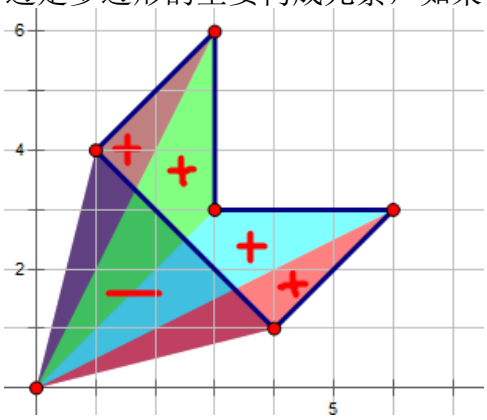
用二进制来记录每个带权点的状态，带权点总价值可在最后状态中得出。

那么只需要枚举起点，计算达到其他二进制状态且回到该点的最短路，由于边权为1，bfs即可。

3 各种变法

3.1 结合原点

边是多边形的主要构成元素，如果将每条边与原点结合



则得到了若干三角形。

[1]Source: [bzoj 1294]SCOI2009 Bean

用射线法的判断方式可以证明 多边形的面积等于三角形带正负的面积和，每个三角形的面积可以直接通过叉积得到。

要注意的是叉积的顺序不同，得到的面积正负不同，而这恰好能说明多边形的绕行方向。

【例2】Pollution Solution [2]

试题大意

求多边形和圆的交的面积。

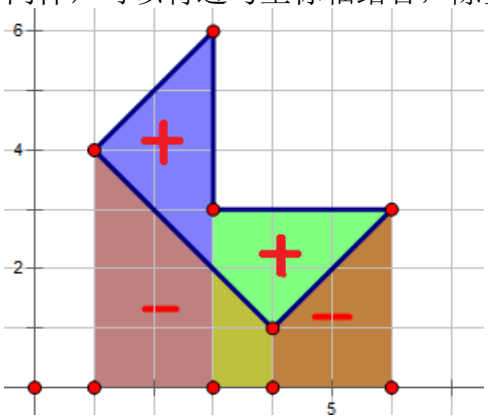
分析

多边形面积可以拆分为若干个三角形的有向面积，则问题可以化简为求三角形与圆的交的面积。若按圆心拆分可以保证三角形其中一顶点位于圆心。

然后分4种情况讨论三角形与圆的位置关系，分别计算即可。

3.2 沿坐标轴拆分

同样，可以将边与坐标轴结合，除竖直边外每条边都向 x 轴投影

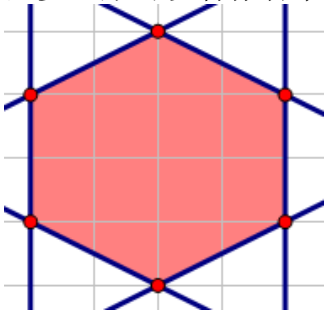


边与投影构成直角梯形（或矩形）。

[2]Source: World Finals 2013 Problem J

3.3 半平面

凸多边形可以看作若干个半平面的交集。



【例3】凸多边形[3]

试题大意

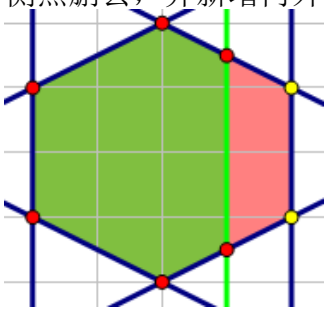
求 n 个凸多边形的交集的面积。

分析

凸多边形的交集就是所有半平面的交集。

这里介绍一种复杂度为平方，但比较好写的半平面求交算法：

使用增量法，对于新增的半平面，求出原交集中在半平面内侧和外侧的点，将外侧点删去，并新增内外侧点连线和半平面的交点，就形成了新的交集。

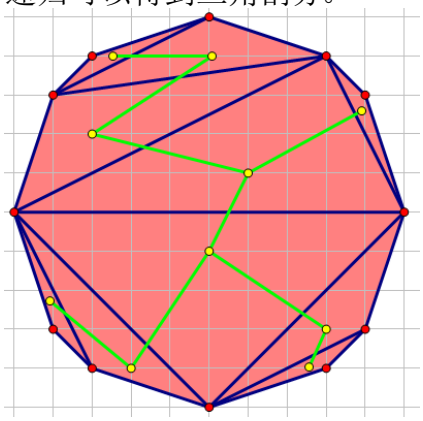


3.4 树

凸多边形对角线都在内部，按对角线分割成两个凸多边形。

[3]Source: [bzoj 2618]Cqoi2006

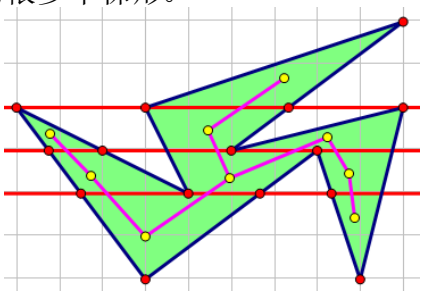
递归可以得到三角剖分。



所有分解出的三角形会组成一棵树。

简单多边形也可以变成树!

所有端点左右水平延伸直到第一个交点（延伸为直线也可以），将多边形分解为很多个梯形。



所有梯形组成了一棵树。

【例4】旅游[4]

试题大意

给出一个凸多边形的三角剖分，求一条对角线，使其穿过的三角形数最多。

分析

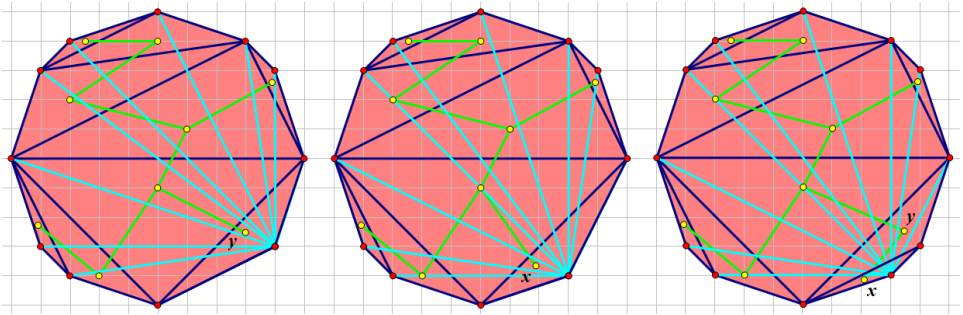
三角剖分构成了树结构，则尝试将对角线转变为树上路径。

[4]Source: [bzoj 2657]ZJOI2012 journey

结论为凸 n 边形的（穿过三角形个数不为0的）对角线与树上的所有路径一一对应。

首先 $n=3$ 时显然成立。假设 $n=k$ 时成立，接下来证明 $n=k+1$ 时成立。

凸 $k+1$ 边形删去一个点之后可以转变为凸 k 边形，只考虑树上增加叶子节点的情况，即树上新增节点 x，其父亲为 y。



删去 y 中一点后，凸 $k+1$ 边形变为凸 k 边形，且使 x 顶替了 y 的位置，也就是说，x 中新增的点与其他点的连线对应了树上从 x 到其他点的路径（不含 $x-y$ 的边）。

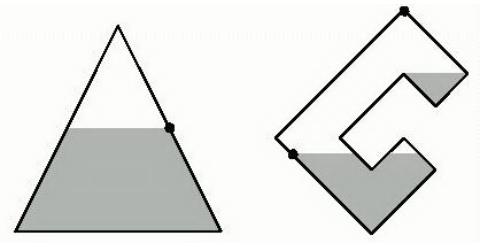
现在重新把 y 加回来，由于 y 是 x 的父亲，所以 x 中新增的点到其他点的连线必定穿过连接 $x-y$ 的边，所以 x 中新增的点到其他点的连线对应了树上从 x 出发的所有路径。即结论对于 $n=k+1$ 成立。

那么求穿过三角形个数最多的对角线，变为了求树上最长链，使用树形DP或两次bfs即可解决。

【例5】Toil for Oil [5]

试题大意

给出一个 n 边形，边界上有洞，求该多边形内部能容纳的油的面积，如图中灰色部分。



[5]Source: World Finals 2002 Problem F

分析

用所有顶点左右水平延伸的直线把多边形切开，形成若干梯形，梯形内要么全部有油，要么完全没有。

从下到上扫出每个区域（用当前区间的中位线去切，若多边形的边有交则加入上边界和下边界，最后对两者排序可得到对应关系），若当前连通块下已有洞，那么上面部分就不可能有油，把连通块标记为不可行。而上面部分对当前区间无影响，直接将当前区间内的可行面积累加到答案。

并查集维护树的连通性，同时用下边界（在当前区间统计前）和上边界（在统计后）上的洞来更新连通块的状态。

4 Architect系列：多边形内部点信息统计问题

4.1 GIS中的应用

在GIS（地理信息系统）中，常常需要统计在某片区域内的特征值。

往往可以用多边形来表示区域，那么这些问题就是多边形内部点信息统计问题了。

在大多数GIS中，使用的是射线法来依次判断点是否在多边形内部，但该算法效率较低，且没有很好地利用地图的性质。

4.2 Architect [6]

4.2.1 试题大意

支持动态加点、询问一个内角均为 45° 的倍数的简单多边形内的点数。其中点在网格中心，多边形在网格边界上。

4.2.2 暴力

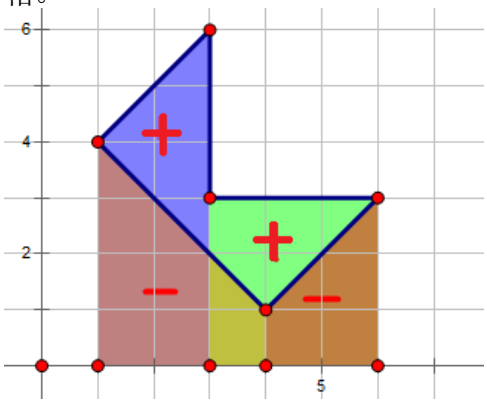
用射线法依次判断每个点是否在内部。

[6]Source: POJ Challenge Round5 绍兴一中邀请赛

4.2.3 多边形拆分

由于多边形较复杂，考虑拆分。把计算多边形面积时每条边变为原点与该边构成的三角形的方法运用到此题上，发现该方法多出了一些与坐标轴既不平行，也不呈 45° 的边，破坏了题目的条件。

则考虑另一种拆分，除竖直边外每条边都向 x 轴投影，该边与投影构成的 45° 直角梯形和矩形。所有 45° 直角梯形和矩形内点数通过加减（沿顺时针遍历则向右的边 +，向左的边 –）即可得到原多边形内的点数，这样仍保留了题目的条件。要注意由于点在多边形边界上也算，所以 – 的 45° 直角梯形需要向下移一格。

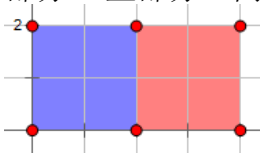


4.2.4 问题转化

多边形拆分后题目变为求 45° 直角梯形和矩形内的点数。

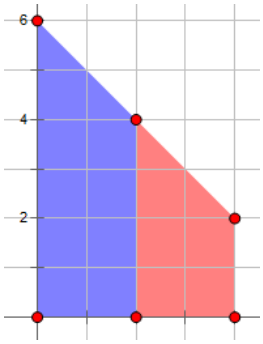
先不考虑动态加点询问，假设所有点已经加完，然后再进行询问。

对于询问某个一边在 x 轴上的矩形内的点数，可以进行转换，即用（红 + 蓝）部分 – 蓝部分。两部分都是求询问点的左下方的点数。



然后使用扫描线算法，即把加点和询问分别排序按 x 坐标(y 坐标为第二关键字)排序，扫描询问，把在询问点左侧的点都加入数据结构中（比如树状数组），然后根据 y 坐标在数据结构中询问区间和（前缀和）。

而对于 45° 直角梯形也可以用类似方法：



此时 y 关键字变为了 $x+y$。

4.2.5 cdq分治

最后由于题目给出的是一边加点一边询问，而点对询问的贡献又是独立的，所以可以使用按时间分治（cdq分治），即所有事件开始按时间排序，然后找到中点，将序列划分为两部分，分别递归左右部分，然后计算左侧点对右侧询问的贡献，此时由于左侧的时间都小于右侧，就变成了刚才先加点再询问的情况，使用刚才所述的方法即可。

【时间复杂度】$O(n\log^2 n)$。

4.3 Architext1 [7]

4.3.1 试题大意

给出初始平面上n个点的点权，Q个操作，每个操作为修改点权或询问一个简单多边形内的点权和。

保证所有询问多边形的边构成一个连通平面图。

4.3.2 无修改

先考虑无修改的情况。

点权和是可以加减的，将多边形沿坐标轴拆分，每条边与其向x轴的投影构成梯形，求梯形内的点权和，在统计时根据边的方向求和。

[7] Source: 2014集训队互测第一问

改为用点对梯形产生贡献，用扫描线优化该算法，用平衡树维护还与扫描线有交的线段，遇到候选点时给在其上方的线段都加上一个值。

最后统计所有线段上的点权和，并根据方向加减。

【时间复杂度】$O(S \log S)$。

4.3.3 cdq分治

使用上述算法，如果要增加或减小点权怎么办？

加减还是可以拆分的，可以套用cdq分治，将一边修改一边询问变为先修改再询问。

即对于当前序列，选定序列中点，计算左侧修改对右侧询问的影响，这部分即为先修改再询问，回到上述问题。

然后左侧右侧分别递归处理。

【时间复杂度】$O(S \log S \log Q)$。

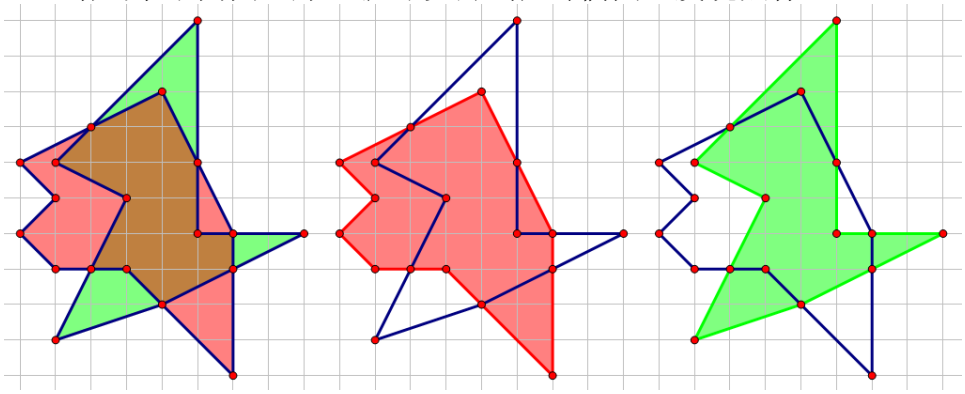
4.4 Architext2 [8]

4.4.1 试题大意

在上一题基础上，求点权和的同时要求最大点权。

4.4.2 问题转化

已经给出了平面图，那么就可以尝试在对偶图上发现点什么。



[8]Source: 2014集训队互测

在对偶图中，每个询问多边形都围出了一个连通块，而其在对偶图中也是连通块。而多边形的每条边就是对偶图中的边，也就是说，问题转化为了删去一些边，询问某个连通块内的信息。（多边形退化为线段时有点特殊，内部没有任何域，这个可以直接特判。）

接下来引入动态图问题。

4.4.3 动态图问题

问题简述：多次询问删去一些边后某个连通块内的信息。

使用cdq分治，任何时刻保证当前区间内所有询问中没有涉及到的边都已合并。并记录每条边在区间内所有询问中出现的次数。

递归左侧时，将右侧询问中出现，而左侧询问中未出现的边加入图中，用并查集维护连通块内信息。

在递归回来后，需要把并查集恢复到递归前的状态，也就是把左侧中添进去边删掉。

由于删的是一些最后添进去的一些边，只需对添边的栈进行退栈操作，即撤销并查集内指定父亲的操作，还原维护的信息。

递归右侧时也一样，退回来时也要恢复。

当区间内只有一个询问时，直接得到所在连通块内的信息。

4.4.4 信息对应

现在的难点变为如何把平面图中点的信息放到对偶图中的点上，即平面图中的域。

询问为一个多边形内部，需要找出至少一个内部的域（顺时针方向，某条边右侧的域），询问其所在的连通块内的信息。

4.4.5 无修改

在计算max时可以对一个点重复计算多次，无修改时，直接把域上所有点权的max作为域的权值，然后套用动态图求一个连通块内的点权最大值。

【时间复杂度】$O(S \log S \log n)$。

4.4.6 数点数

再看下点权均为1时的特殊情况，也就是说求的是内部点数。这个可以利用连通平面图上的欧拉定理，即

$$\text{点数} = \text{边数} - \text{域数} - 1$$

连通块内的域数和边数可以较容易地求出，从而求得点数。

【时间复杂度】$O(S \log S \log n)$。

4.4.7 一般情况

在一般情况时怎么办？

我们需要把原来的每一个点对应到某一个域上。

继续分析询问，可以发现所有在内部的点的邻域也都在对偶图的连通块内，所以对于内部的点，直接随便选一个域作为对应即可。

虽然边界上的点不满足，但这些点可以直接通过询问的信息得出。

此外需要强制边界上的点在询问时不添加到域中，在cdq分治时要多记录一个点出现次数的信息。

即递归左侧时，需要把右侧出现，而左侧未出现的点按对应域放入其中。

【时间复杂度】$O(S \log S \log n)$。

4.4.8 加入修改

在有修改操作时，由于已经有了统计一个点涉及到的操作数信息，直接把修改也算入其中。

在递归到只有修改时进行修改。

【时间复杂度】$O(S \log S \log n)$。

4.5 ArchitEXT

4.5.1 试题大意

在上一题基础上，不给出所有询问多边形构成的连通平面图。

4.5.2 自力更生

不给出平面图，但仍要使用上述算法，所以只能自力更生，求出平面图。

用扫描线法可以求出所有多边形的边的交点，这些交点和多边形的边就构成了平面图。

这一步的复杂度为 $O(\text{交点个数}\times \log S)$。

4.5.3 不连通

注意到这样求出的平面图不一定连通。

所以就要使其连通，套用作者在WC2014营员交流中的方法，可以转化为连通平面图。

在得到连通平面图后，就变为上一题了。

4.5.4 优化

复杂度中的瓶颈为求所有交点这一步，即 $O(\text{交点个数})$，但把所有交点求出来显得不太科学，而且太浪费了。

考虑分块，每 T 个询问分成一块，这一块中需要建出平面图，而其余的点利用扫描线点定位可以放入一个域中。

这样建平面图复杂度为 $O(\frac{S}{T}\times T^2\log S)$，其余点进行点定位复杂度 $O(\frac{S}{T}\times S\log S)$，对T个询问求解的复杂度为 $O(\frac{S}{T}\times T\log^2 T)$。

综合发现 T 取 $\sqrt{S}$ 时最优，最终复杂度为 $O(S^{1.5}\log S)$。

但这样还有一个问题，如果询问多边形边数太多，使得序列无法按 $\sqrt{S}$ 分割怎么办？

注意到这样的多边形数量不会太多，而且内部只有一个域，可以直接使用点定位来判断，总复杂度 $O((nQ+S)\log S)$。

将所有边数 $>\sqrt{S}$ 的询问（不会超过 $\sqrt{S}$ 个）提出，每个单独求解，这样就可以保证复杂度为 $O(S^{1.5}\log S)$。

4.6 小结

在常用算法求max遇挫时，作者另辟蹊径，以多个多边形组合为平面图，将其转化为了图论问题。

在该问题中还频繁使用了按时间分治（cdq分治）的算法，化繁为简。

并在最后用到了分块以及分类讨论的思想，降低了最坏复杂度，最终得到了一个较优秀的算法。

5 总结

虽然多边形的边数不定，且形状复杂，给解决题目造成了很多困难。

但通过巧妙的拆分，多边形便能**变**为若干部分，为逐个击破提供了可能。

而在分析题目中多边形的特点后，套用常用拆分方法，往往能收到不错的效果。

感谢

感谢中国计算机学会提供这个学习与交流的平台。

感谢绍兴一中的陈合力老师、邵红祥老师、游光辉老师、董烨华老师多年来给予的关心和指导。

感谢国家集训队教练胡伟栋和余林韵的指导。

感谢清华大学的徐捷、鲁逸沁、裘捷中、梁佳文和上海交通大学的蒋舜宁学长对我的帮助。

感谢绍兴一中的何奇正、郑钟屹、徐正杰同学对我的帮助和启发。

感谢绍兴一中的俞鼎力、张恒捷、王鉴浩、郭雨等同学对我的帮助和启发。

感谢其他对我有过帮助和启发的老师和同学。

感谢父母家人的支持和无微不至的关心、照顾。

参考文献

[1] 顾昱洲，《浅谈一类分治算法》，WC2013授课。

[2] 许昊然，《浅谈数据结构题的几个非经典解法》，国家集训队2013论文。

[3] 董宏华，《扫描线在计算几何中的应用》，WC2014营员交流。

[4] 董宏华，《Architext命题报告》，国家集训队2014作业。

-- 空 --

对置换群有关算法的初步研究

浙江省镇海中学 岑若虚

摘 要

置换群是信息学竞赛中的常见模型。本文介绍了有关置换群的基本知识，并从一般化的置换群成员性判定问题出发，对Schreier－Sims算法进行了初步研究。

1 预备知识

1.1 群

设G是一个非空集合，$\circ$是定义在G上的二元运算。如果满足

- 封闭性：$\forall a,b \in G, a \circ b \in G$；
- 结合律：$\forall a,b,c \in G, (a \circ b) \circ c = a \circ (b \circ c)$；
- 单位元：$\exists e \in G, \forall a \in G, e \circ a = a \circ e = a$；
- 逆元：$\forall a \in G, \exists a^{-1} \in G, a \circ a^{-1} = a^{-1} \circ a = e.$

则称$(G,\circ)$是一个**群**，简称G是群。以下把$a \circ b$简记为ab。群中元素的个数称为群的**阶**，由此分为有限群和无限群。群是对众多代数系统的抽象。例如全体整数对于加法运算，$[1,n]$中所有与n互质的整数对于模n意义下的乘法运算都作成群。需要注意的是群不一定满足交换律。

群$(G,\circ)$中，若H是G的子集，且$(H,\circ)$也是群，则称H是G的**子群**，记作$H \leq G$。任意群G都有平凡子群G和$\{e\}$，其它子群称为真子群。

对于G的子集M，所有包含M的子群的交也是一个子群，称为M的**生成子群**，记作$\langle M\rangle$。可以把生成子群看做M中元素任意进行运算得到的群。M称为$\langle M\rangle$ 的**生成集**。

例如一个简单的群$G = \{0, 1, 2, 3, 4, 5\}$，运算是模6意义下的加法，它的单位元是0。它的真子群有$H_1 = \{0, 2, 4\}$和$H_2 = \{0, 3\}$，H_1也是$\{2\}$的生成子群。

1.2 陪集

若H是G的子群，对任意$a \in G$，称

$$aH = \{ah | h \in H\}$$

为子群H的一个**左陪集**，称

$$Ha = \{ha | h \in H\}$$

为子群H的一个**右陪集**。

设G中H的右陪集作成的集合为S_R，左陪集作成的集合为S_L，可以证明映射$\phi : Ha \mapsto a^{-1}H$是$S_R$到$S_L$的一一映射。下面我们只考虑右陪集。

在之前的例子中，H_1的右陪集有$H_10 = H_12 = H_14 = \{0, 2, 4\}, H_11 = H_13 = H_15 = \{1, 3, 5\}$。$H_2$的右陪集有$H_20 = H_23 = \{0, 3\}, H_21 = H_24 = \{1, 4\}, H_22 = H_25 = \{2, 5\}$。从这个例子中，我们可以观察到下面的定理。

定理1. *设H为G的子群，任给H的右陪集Ha, Hb，则要么$Ha = Hb$，要么$Ha \cap Hb = \varnothing$.*

证明. 若存在$x \in Ha \cap Hb$，则存在$h_1, h_2 \in H$，使得$x = h_1a = h_2b$. 因此有

$$\forall ha \in Ha, ha = hh_1^{-1}h_1a = hh_1^{-1}h_2b = h'b \in Hb,$$

故$Ha \subseteq Hb$，同理$Hb \subseteq Ha$. 因此$Ha = Hb$，定理得证。 □

这个定理说明了群G可以划分成两两不相交的右陪集的并，给我们一种分析和简化群的结构的思路。

记$|G : H|$表示G中子群H的不同右陪集的个数。用1表示只含单位元的子群，$|G : 1|$表示G的阶。我们可以立即得到如下定理。

定理2. *（拉格朗日定理）若H是有限群G的子群，则$|G : 1| = |G : H||H : 1|$*

1.3 置换

一个有限集合Ω到Ω的一个一一映射称为Ω的一个**置换**。不妨把Ω看成$\{1,2,\ldots,n\}$，则置换可以表示成

$$\begin{pmatrix} 1 & 2 & \ldots & n \\ a_1 & a_2 & \ldots & a_n \end{pmatrix}$$

其中a是一个排列。置换之间的运算是置换的合成，即

$$\begin{pmatrix} 1 & 2 & \ldots & n \\ a_1 & a_2 & \ldots & a_n \end{pmatrix}\begin{pmatrix} 1 & 2 & \ldots & n \\ b_1 & b_2 & \ldots & b_n \end{pmatrix}=\begin{pmatrix} a_1 & a_2 & \ldots & a_n \\ b_1 & b_2 & \ldots & b_n \end{pmatrix}$$

n个元素的置换有$n!$个，容易验证这$n!$个置换对于置换合成运算是一个群，称为n阶对称群，用S_n表示。**置换群**是S_n的子群，它的元素是置换。

元素β在置换g下的象记作β^g。元素β在置换群G中所有置换下的象的集合称为β的**轨道**，记作β^G。置换群G中不改变元素集$A=\{a_1,\ldots,a_m\}$的置换组成的子群称为G中A的**稳定集**，用G_A或$G_{a_1,\ldots,a_m}$表示。

1.4 例题1：Shift it![1]

1.4.1 题目大意

有一个$n\times n$的网格，格子上分别写着$1\sim n^2$的数字，每个数字出现一次，如图(a)所示。每次操作可以把某一行循环左移或右移一格，或把某一列循环上移或下移一格。判断是否能通过这些操作将原始网格变成目标网格的方案，若能求任意不超过10000步的方案。

(a)

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

(b)

2	1	3	4	5	6
13	8	9	10	11	12
19	14	15	16	17	18
25	20	21	22	23	24
31	26	27	28	29	30
7	32	33	34	35	36

[1]**题目来源**：Codeforces 74E

1.4.2 数据规模

$n = 6$.

1.4.3 分析

虽然数据规模看似很小，但是操作序列的数量巨大，难以搜索。我们用置换群的观点分析，发现每种操作可以表示成n^2个数的一个置换，能完成的所有操作就是这些置换组成的集合在S_{n^2}中的生成子群G，询问的就是原始网格到目标网格的置换h是否在G中。

对于题中n是偶数的情况我们可以构造出交换任意相邻两个数的方案。例如要交换(a)中的1和2，可以依次将第一行或第一列向上、向左、向下、向右、向上，变成(b)的情况。可以发现这一系列操作的效果是交换了1和2，并将第一列的其它数循环上移了一格。重复$n-1$次后就只交换了1和2。

既然可以交换任意相邻两个数，显然任意目标网格都可以达到，因为我们可以按顺序将每个数一步步交换到它要去的地方。再结合一些简单的贪心策略可以将步数减小到10000步以内。

有趣的是，n为奇数时就不一定有解了。我们知道，置换可以拆成几个循环的乘积。循环$(a_1, a_2, \ldots, a_n)$表示a_1换成a_2，a_2换成a_3，…，a_n换成a_1。长度为2的循环称为对换，循环可以再拆成对换的乘积。每个置换表示成对换的乘积时，表示法不唯一，但奇偶性不变，由此分为奇置换和偶置换。当n为奇数时，操作都是偶置换，不可能生成奇置换。

2 Schreier–Sims算法

2.1 一般置换游戏可解性判定问题

例题1中我们虽然得出了优美的结论，但是使用了难以想到的构造方法和非常特殊的性质。下面我们考虑一般化的问题：对n个数进行操作，每个操作是一个置换，操作集合为S，问能否将一个状态变成另一个状态。也就是说，给定$S \subseteq S_n, h \in S_n$，判断是否有$h \in \langle S \rangle$。下面假设元素集是$\Omega = \{1, 2, \ldots, n\}, G = \langle S \rangle \subseteq S_n$.

2.2 基和强生成集

G的**基**是一个Ω的元素序列$B = (\beta_1, ..., \beta_m)$，满足$G_B = 1$，即$G$中能让$B$中所有元素都不变的置换只有单位元一个。$B$定义了一个子群链

$$G = G^{[1]} \geq G^{[2]} \geq ... \geq G^{[m]} \geq G^{[m+1]} = 1 \tag{1}$$

其中$G^{[i]} = G_{\beta_1,...,\beta_{i-1}}$，即能让$\{\beta_1, ..., \beta_{i-1}\}$不变的置换构成的子群。如果$\forall 1 \leq i \leq m, G^{[i+1]} \neq G^{[i]}$，那么这个基称为无冗余的。不同的无冗余基可能有不同的大小.

群G的一个生成集T是群G关于基B的**强生成集**, 如果有

$$\forall 1 \leq i \leq m+1, \langle T \cap G^{[i]} \rangle = G^{[i]} \tag{2}$$

也就是说，强生成集中必须包含(1)中所有子群的生成集。

例如，考虑群$G = S_4$，序列$B = (1, 2, 3)$是G的一个无冗余基。这里$G^{[1]}, G^{[2]}, G^{[3]}$分别是$\{1, 2, 3, 4\}, \{2, 3, 4\}$和$\{3, 4\}$的所有置换的集合，$G^{[4]} = 1$，满足(1)式。集合$T_1 = \{(1, 2, 3, 4), (3, 4)\}$和集合$T_2 = \{(1, 2, 3, 4), (2, 3, 4), (3, 4)\}$都是$G$的生成集，但是$T_1$不是关于B的强生成集，因为$\langle T_1 \cap G^{[2]} \rangle \neq G^{[2]}$。而$T_2$是一个关于$B$的强生成集。

2.3 分解过程

假设已经求出了基B和强生成集T，我们来判定h是否属于G。

我们用$G^{[i+1]}$的陪集划分$G^{[i]}$。$G^{[i]}$中β_i可能变成$\beta_i^{G^{[i]}}$中的元素，而$G^{[i+1]}$是β_i的稳定集，因此$\beta_i^{G^{[i]}}$中的每个元素对应$G^{[i+1]}$的一个陪集。对于每个陪集，我们选取一个代表元r来表示陪集$G^{[i+1]}r$。这些代表元的集合记作R_i。R_i可以通过BFS求出：以元素为点，$T \cap G^{[i]}$中的置换为边，从β_i开始BFS。若能到达γ，说明它在$\beta_i^{G^{[i]}}$中。将β_i到γ的路径上的置换依次相乘，得到的就是将β_i变成γ的陪集的代表元。

继续上面的例子，$G = S_4, B = (1, 2, 3), T = \{(1, 2, 3, 4), (2, 3, 4), (3, 4)\}$。$G^{[3]} = \{e, (3, 4)\}$，它把$G^{[2]}$分成三个陪集$G^{[3]}e = \{e, (3, 4)\}, G^{[3]}(2, 3) = \{(2, 3)(2, 3, 4)\}, G^{[3]}(2, 4) = \{(2, 4)(2, 3, 4)\}$。因此$R_2 = \{e, (2, 3), (2, 4)\}$。这三个陪集中的置换分别把$\beta_2 = 2$变为2,3和4。

现在，任意$g \in G$可以唯一表示成$g = r_m r_{m-1} \dots r_1$的形式，其中$r_i \in R_i$。分解过程非常简单：给定$g \in G$，先找出陪集代表元$r_1 \in R_1$满足$\beta_1^g = \beta_1^{r_1}$；然后令$g_2 = g r_1^{-1} \in G^{[2]}$，找出$r_2 \in R_2$满足$\beta_2^{g_2} = \beta_2^{r_2}$；令$g_3 = g_2 r_2^{-1}$，依次类推。

分解过程可以判定h是否属于G。我们尝试用以上方法分解h，若能成功分解显然有$h \in G$，否则$h \notin G$。注意到分解失败的情况有两种，一种是分解过程中求出的$h_i = hr_1^{-1}r_2^{-1}\dots r_{i-1}^{-1}$把$\beta_i$换到了$\beta_i^{G^{[i]}}$之外，导致不能继续分解，我们把这个$h_i$称为剩余置换。另一种是$h_{m+1} \neq e$，这个$h_{m+1}$也称为剩余置换。

2.4 Schreier引理

如果R是G中H的陪集代表元的集合，那么对任意$g \in G$，$Hg \cap R$只有一个元素，用$\overline{g}$表示。

定理3. *设$H \leq G = \langle S \rangle$，$R$为$G$中$H$的陪集代表元的集合，$e \in R$。那么集合*

$$T = \{rs(\overline{rs})^{-1} | r \in R, s \in S\}$$

是H的生成集。

证明. 由定义，T中元素都属于H，$\langle T \rangle \subseteq H$。

任取$h \in H \leq G$，h可以写成$h = s_1 s_2 .. s_k$的形式, 其中$s_i \in S$。我们定义一个G中元素的序列$h_0, h_1, \dots, h_k$使得

$$h_j = t_1 t_2 \dots t_j r_{j+1} s_{j+1} s_{j+2} \dots s_k \tag{3}$$

其中$t_i \in T, r_{j+1} \in R, h_j = h$。首先令$h_0 = es_1 s_2 \dots s_k = h$。如果$h_j$已经定义，令$t_{j+1} = r_{j+1}s_{j+1}(\overline{r_{j+1}s_{j+1}})^{-1}, r_{j+2} = \overline{r_{j+1}s_{j+1}}$。显然，$h_{j+1} = h_j = h$。这就是(3)要求的形式。我们有$h = h_k = t_1 t_2 \dots t_k r_{k+1}$。由于$h \in H$且$t_1 t_2 \dots t_k \in \langle T \rangle \leq H$，一定有$r_{k+1} \in H \cap R = 1$。因此$h \in \langle T \rangle$。

综上所述，$H \subseteq \langle T \rangle$。所以$\langle T \rangle = H$。 □

2.5 算法流程

我们通过不断调整，维护元素序列$B = (\beta_1, \beta_2, \dots, \beta_m)$，生成集序列$T_1, T_2, \dots, T_{m+1}$，保证$T_i$是$\{\beta_1, \dots, \beta_{i-1}\}$的稳定集，并保证$\langle T_i \rangle \geq \langle T_{i+1} \rangle$对$1 \leq i \leq m$成立，$\langle T_1 \rangle = G, T_{m+1} = 1$。还要维护$\langle T_i \rangle$中$\langle T_i \rangle_{\beta_i}$的陪集代表元集合$R_i$。我们使用一个指示器$cur$，保证$i > cur$时有

$$\langle T_i \rangle_{\beta_i} = \langle T_{i+1} \rangle \tag{4}$$

从而$(\beta_{cur+1},\ldots,\beta_m)$是$\langle T_{cur}\rangle$的基，$\bigcup_{cur+1\le j\le m} T_j$是$\langle T_{cur}\rangle$的强生成集。

1. 开始时令$m = 1$，β_1为任意会被S中置换改变的数，$T_1 = S$，$cur = 1$，用BFS求出R_1。

2. 每次判断$i = cur$时(4)是否成立。根据我们的保证，$\langle T_{cur}\rangle_{\beta_{cur}} \supseteq \langle T_{cur+1}\rangle$成立，只要判断$\langle T_{cur}\rangle_{\beta_{cur}} \subseteq \langle T_{cur+1}\rangle$是否成立。我们利用定理3求出$\langle T_{cur}\rangle_{\beta_{cur}}$的生成集$T'$，并利用分解过程判断是否有$T'$中的每个元素都属于$\langle T_{cur+1}\rangle$。

如果(4)成立，可以将cur减1。如果不成立，有T'中的元素$h \notin \langle T_{cur+1}\rangle$。将$h$在分解过程中得到的剩余置换加入$T_{cur+1}$。若$cur = m$，我们要任选会被$S_{cur}$改变的元素作为$\beta_{m+1}$。重新进行BFS更新$R_{cur+1}$。现在$i = cur + 1$时(4)不一定成立，需要将$cur$加1。

3. 重复以上过程，直至$cur = 0$，我们就得到了G的基B和强生成集$T = \bigcup_{1\le j\le m} T_j$。

2.6 复杂度分析

基的大小最多为n。每个T_i最多改变n次，因为每次改变后β_i的轨道都会增加一个元素。这样第2步执行$O(n^2)$次。每次求出$\langle T_{cur}\rangle_{\beta_{cur}}$的强生成集的大小是$O(|R_{cur}||T_{cur}|) = O(n^2)$，对其中每个元素执行分解过程需要$O(n^2)$。这样得出的时间复杂度上界是$O(n^6)$。

注意到如果已经知道$\langle T_{cur}\rangle_{\beta_{cur}}$的强生成集的某个元素属于$\langle T_{cur+1}\rangle$，之后就不必再对它执行分解过程。因此对于每个$1 \le cur \le m$都只要进行$O(|R_{cur}||T_{cur}|) = O(n^2)$次分解操作，总的时间复杂度是$O(n^5)$。如果群的大小为$|G|$，时间复杂度也可以表示成$O(n^2 \log^3 |G|)$。

在实际运行中算法的常数很小，一般不需要这么多操作。我的程序能在2秒内跑出$n = 100$的数据。

空间复杂度是$O(n^3)$。

2.7 例题2：Group[2]

2.7.1 题目大意

给定长度为N的置换集合S，$|S| = M$求满足以下条件的集合G的最小的阶：

[2]**题目来源**：2013 Multi-University Training Contest 10, HDU4702.

1. $S \subset G$

2. $\forall \sigma, \tau \in G, \sigma\tau^{-1} \in G$

2.7.2 数据规模

$N \leq 50, M \leq 100.$

2.7.3 分析

G中结合律和逆元由定义成立，封闭性也成立，单位元$e = \sigma\sigma_{-1} \in G$，因此$G$是群，包含$S$的最小的群即为$S$生成的群。因此$G = \langle S \rangle$，要求的就是$|\langle S \rangle|$。

我们用Schreier－Sims算法求出$\langle S \rangle$的基B。我们知道B定义了一个子群链

$$G = G^{[1]} \geq G^{[2]} \geq ... \geq G^{[m]} \geq G^{[m+1]} = 1$$

由拉格朗日定理，

$$|G| = \prod_{i=1}^{m} |G^{[i]} : G^{[i+1]}|$$

而$G^{[i]}$中$G^{[i+1]}$的陪集个数就是$|R_i|$。Schreier－Sims算法求出了R_i，我们直接求$\prod_{i=1}^{m} |R_i|$即可。需要用到高精度。

3 总结

Schreier－Sims算法是计算群论的基本算法。还有一些更优秀的类似算法以及对该算法的优化，由于本人水平有限，以及考虑到在OI中的实现难度不作介绍。该算法实际上清晰地表示出了置换群的结构，因此可以方便地完成成员性判定、求阶等任务。但是该算法是个一般化的算法，并没有利用具体的置换群的特殊性质，因此时间复杂度较大。我们在面对具体问题的时候要挖掘题目的特殊性质再设计算法。

致谢

感谢杜瑜皓同学对本文写作的帮助。

感谢李建老师的指导。

参考文献

[1] Ákos Seress, “Permutation Group Algorithms”, Cambridge University Press.

[2] Donald E. Knuth, “Efficient Representation of Perm Groups”, Combinatorica 11(1991).

[3] 同济大学应用数学系《离散数学》编写组, 《离散数学》, 同济大学出版社。

-- 空 --

浅谈线性相关

长沙市雅礼中学 匡正非

摘 要

很多同学在OI生涯、大学甚至更高级的学习当中，都会或多或少的接触到线性代数。线性相关是线性代数中重要的一员，它本身并不复杂，但在许多重要的知识中都能看到它的影子。本文将对线性相关进行介绍，并讨论它在线性代数中的拓展，以及在信息学竞赛中的一些经典应用。

1 预备知识

在正题开始之前，我们首先要介绍一些线性代数的知识。

1.1 矩阵的行列式

行列式是线性代数中的一个函数，将一个n阶方块矩阵A映射到一个标量。它是由数学公式定义的，大家可以去翻阅相关的资料，这里不再赘述。本文中，我们用$det(A)$或者$|A|$表示矩阵A的行列式。

1.2 转置矩阵

线性代数中，一个$n \times m$的矩阵A的转置是另一个矩阵A^T。其中A的每个横行均对应A^T的一个纵列。举个例子，对于一个3×2的矩阵：

$$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}$$

它的转置即为：

$$\begin{bmatrix} a & c & e \\ b & d & f \end{bmatrix}$$

1.3 逆矩阵

对于一个n阶方阵A，如果存在另一个n阶方阵B，使得$AB = BA = I_n$，其中I_n为n阶单位矩阵，则称方阵A是可逆的，而B为A的逆矩阵。

引理 1：一个方阵A可逆的充要条件是$det(A) \neq 0$。（克莱姆法则）

1.4 线性空间

线性空间的定义很长，这里不再详细说明，只解释由k维向量（有序的k元数组）组成的线性空间是什么。

设V是一个由k维向量组成的集合，$\mathbb{F}$是一个域（通常是数域）。如果满足：

$$\forall a, b \in V \qquad a + b \in V$$

$$\forall a \in V,\ k \in \mathbb{F} \qquad ka \in V$$

那么就称V是一个在域$\mathbb{F}$上的线性空间。特别要注意的是，不论线性空间里面装的实际是什么，我们都把它们叫做向量。

1.5 线性组合

设V是一个在域$\mathbb{F}$上的线性空间，$S = \{v_1, v_2, \cdots, v_n\}$为$V$的一个子集。对于一个$V$中的元素$v$，若存在一组系数$\{a_1, a_2, \cdot, a_n\} \in \mathbb{F}$使得：

$$v = a_1 v_1 + a_2 v_2 + \cdots + a_n v_n$$

则称v是S的一个线性组合。

2 线性相关

2.1 什么是线性相关

令V是在域$\mathbb{F}$上的一个线性空间。对于V的一个子集$S = \{v_1, v_2, \cdots, v_n\}$，如果有一组域$\mathbb{F}$中的非全零系数$a_1, a_2, \cdots, a_n$使得：

$$a_1 v_1 + a_2 v_2 + \cdots + a_n v_n = 0$$

则称S是线性相关的。注意这里的0是一个向量。反之，如果找不到这样一组系数，则称这组向量线性无关。

或者说，如果0向量是S的一个线性组合，则称S线性相关。否则称S线性无关。同时，一组线性相关的向量被称为线性相关组，反之亦然。

举几个例子：向量$[1,2,0],[0,-1,1],[1,1,1]$是线性相关的，因为后一个向量是前两个的和；而向量$[1,0,0],[0,1,0],[0,0,1]$是线性无关的。

有一点必须提到：如果上式中的所有运算都换成异或的话，我们也可以说它们之间有类似的关系。而异或与线性相关的联系在OI中相对更多一些。

2.2 线性相关的基本定理

线性相关有很多基本的定理：

定理 1：$k+1$个k维向量之间一定线性相关。

定理 2：任一线性相关组的超集均线性相关。

定理 3：任一线性无关组的子集均线性无关。

定理 4：如果一组向量S中有一个0向量或有两个向量相等，则其线性相关。

定理 5：任一线性相关组都能被一个线性无关组的线性组合表示。

线性相关本身并没有什么太多值得叙述，但是它在线性代数中出现的频率却比比皆是，这就是我们接下来要讨论的方向。

3 线性空间的基

线性空间的定义之前已经提到过。显然，不同维度的欧几里得空间$\mathbb{R}^n$都是在实数域$\mathbb{R}$上的线性空间。在二维欧几里得空间$\mathbb{R}^2$中，所有的向量都可以被两个线性无关的向量$[0,1],[1,0]$的线性组合唯一表示。如果把这种现象进行扩展，可以得到所有的线性空间都具有这种性质。那么，我们把这些类似于$[0,1],[1,0]$的向量成为线性空间的基。更标准的定义是：

设V为一个线性空间，$\mathfrak{B}$为V的一个子集。如果V中的任意一个向量都能唯一地表示成$\mathfrak{B}$中向量的线性组合，则称$\mathfrak{B}$是V的一组基。

根据$\mathbb{R}^2$的情况，不难想到可能所有线性空间的基都是线性无关的。事实上也是如此，因为一组线性相关的基会有无数种线性组合来表示线性空间内的向量。所以我们可以用另一种方法来描述线性空间的基：

线性空间的基是其满足线性无关的极大子集。

显然，线性空间的基可能有很多个，但它们的元素个数（下称基数）都是相等的，称为线性空间的维度。反过来，一个线性无关组只能成为一个线性空间的基。因此，我们可以用线性空间的基来表示线性空间。同样，我们也可以用一组向量S（不一定线性无关）来构造线性空间，构造出来的线性空间称为$span(S)$。

另外，有些线性空间的基的大小是无限的，这里只是提到不再细讲。有兴趣的同学可以去翻阅一下其他资料。

4 矩阵的秩

矩阵可以看成是由很多个向量组成的向量，具有很多优美的性质，因此在线性代数中具有很重要的地位。

4.1 什么是矩阵的秩

正因为矩阵可以看成是一个行（列）向量组，研究这些向量之间是否线性相关也便是一个值得探讨的问题。也就是这样，矩阵的秩出现了。

一个矩阵A有两种秩：行秩和列秩。行秩表示矩阵A的线性无关的横行的极大数目，列秩则是满足条件的纵列的极大数目。显然A的行秩等于A^T的列秩。但是更优美的是：

定理 6：矩阵的行秩等于列秩。

证明的方法有很多很多种，这里只给出一个。不过在证明之前，我们还要探讨一下其他的东西。

4.2 秩和基的关系

矩阵有秩，而线性空间有基。我们之前提到过一组向量可以构造出一个线性空间，而矩阵正好就是一组这样的向量。不难得出，矩阵的行秩对应的集合就是由它的横向量构造出来的线性空间（下称行空间）的基，列秩则对应其列空间的基。

4.3 行秩等于列秩

有了上一节的探讨，我们就可以证明之前的定理了。

对于一个$n \times m$的矩阵A，我们设它的行秩为r，那么A中的行空间V的维度为r。在V中找到任意一组向量$S = \{v_1, v_2, \cdots, v_r\}$，并用这些向量作为横行构成一个$r \times m$的矩阵$R$。显然$A$中的每个行向量都能被$S$的一个线性组合表示，用矩阵乘法来说，一定存在一个$n \times r$的矩阵$C$ 使得$A = CR$。

因为$A = CR$，所以A中的每个列向量都能被C中的列向量的线性组合表示。显然A的列空间能被C的列空间完全包含，因此A的列秩$\leq C$的列秩。又因为C只有r列，所以C的列秩$\leq r = A$的行秩，因此A的列秩$\leq A$的行秩。根据对称性同样有A的行秩$\leq A$的列秩，所以A的行秩$= A$的列秩。证毕。

既然行秩等于列秩，我们把这两者统一起来，统称为矩阵的秩，用$R(A)$来表示。显然对于一个$n \times m$的矩阵A，$R(A) \leq \min(n, m)$。当$R(A) = \min(n, m)$时，我们称A满秩，否则称其欠秩。

4.4 秩和行列式

当一个$n \times m$的矩阵A的行列式不为0时，不难证明A的n个行向量之间是线性无关的。也就是说这个矩阵满秩。所以如果用行列式去理解秩，新的定义就会出现：

> 当一个矩阵A中存在r阶子式（即任选出r个横向量组成矩阵求行列式）为0，而所有的$r + 1$阶子式均为0时，我们规定矩阵A的秩$R(A) = r$。

到了这里，基、秩、行列式之间的关系我们都已经进行了分析。现在只剩一个问题了：怎么求秩。

4.5 如何求秩

如果要求秩，最普遍的方法便是高斯消元。相信所有学习OI的同学们都有写过高斯消元，并且认为这是求解一组线性方程的普遍方法。实际上，对一个矩阵（或者说横向量组）A进行高斯消元后得到的就是$span(A)$里的一组基。也就是说，做了一遍高斯消元后，剩下的非0元素个数即为$R(A)$。

一般来说，高斯消元的时间复杂度为$O(n^3)$，但是如果把向量之间的线性组合变为元素的异或和的话，这时一组元素对应的线性空间维度最多为$O(log$(元素大小))，高斯消元的时间复杂度就变为了$O(nlog$(元素大小))。

4.6 最大异或和

说道高斯消元，就不得不提最大异或和问题。最大异或和是OI中比较经典的问题：给定一组自然数，求其中异或和最大的子集。显然，将这一组数字进行异或高斯消元后，会形成一个倒立的三角形（元素从上至下递减）。这时我们从上至下依次枚举每个元素，将它和当前答案的异或和与当前答案的最大值作为新的答案。不难证明，这样产生的答案一定是最优的。

5 线性代数之外的联系

之前几章讲的都是线性相关在线性代数中的一些扩展，更像是数学。其实线性相关和很多其他的东西也有联系。

5.1 线性相关与拟阵

很多学过贪心的同学都知道拟阵，也就是贪心算法的证明来源。下文中我们用二元组$M = \{S, L\}$来表示拟阵。我们设A是一个行向量组，可以证明：

定理 7：A（S）与A的所有线性无关子集（L）构成一个拟阵。

拟阵拥有4个性质，如果4个性质都能被满足，上述定理就能得证。其中的两个是显然的，只要证明其遗传性和交换性即可。

遗传性：由定理3[2.2]可以得到。

交换性：利用反证法。设$X, Y \in L, |X| > |Y|$。如果对于任意$v \in X - Y$都满足$Y \cup \{v\} \notin L$，则X中的所有向量都能被Y中向量的线性组合表示。又因为X线性

无关，不可能被元素个数更小的集合完全表示，这与$|X| > |Y|$矛盾。因此，交换性一定满足。证毕。

有了这个性质，利用贪心就可以带权最大线性无关组问题了。再来看看这题：

5.2 CQOI2013 Problem 1 Nim

5.2.1 题目大意

现有A，B两人进行博弈，游戏规则如下：

游戏中有N堆火柴。第一轮由A从中取出若干堆火柴（不可全取），随后B进行同样的操作。接着，若剩余的各堆火柴个数的异或和为0，则B胜；否则A胜。

问A至少要取多少火柴才能获胜。

5.2.2 数据范围与约定

$N \leq 100$，各堆火柴个数均$\leq 10^9$。

5.2.3 问题分析

如果在A取了一些火柴之后，剩下的元素线性无关的话，那么即为A胜。

那么问题就变为了：给定一些自然数，要求选择一个线性无关的子集，使得子集中所有数的和最大。根据之前所论，利用贪心即可将其解决。

5.3 线性相关与图

如果把一个无向图用点边矩阵表示，那么矩阵（向量组）中的一个线性无关组对应的就是图中的一个生成森林。

再深入一些，如果我们可以求得一个向量组有多少个线性无关组，那么就能解决求一个图中生成森林个数的问题了。不幸的是，后者是一个sharp-P问题（类似于最优解问题中的NP问题），所以前者也肯定是sharp-P问题。如果同学们对这方面有兴趣，可以试着进行探索。

5.4 线性相关与计算几何

在$\mathbb{R}^2$中，两个向量共线的充要条件是它们之间线性相关，$\mathbb{R}^3$中则对应共面关系，以此类推。如果细心一点就可以发现，这个性质与利用叉积判断（求行列式）是等价的。

6 例题

俗话说实践出真知，为了更好体现线性代数在OI中的作用，这里列举出几道线性相关的例题。

6.1 XXor（原创题）

6.1.1 题目大意

定义一个函数$f(S)$，它把一个自然数的集合映射到一个数字上。如果S中一共有k个子集满足其中的所有数异或和为0，那么$f(S) = k^2$。给你一个含有n个自然数的集合L，求：

$$\sum_{T \subset L} f(T)$$

6.1.2 数据规模与约定

$n \leq 24$，S中的所有元素均$\leq 10^{18}$。

6.1.3 题目分析

不难发现，$f(S)$实际对应的就是$2^{2(|S|-span(S)\text{的维度})}$。我们只要得到所有线性空间的维度，问题就能得解。

之前已经提到过，对一组元素进行异或高斯消元的复杂度为$O(nlog(\text{元素大小}))$。这题可以直接对所有集合都进行异或高斯消元，但是总复杂度还是太慢。我们联想到利用*lowbit*函数在$O(2^n)$ 的时间复杂度内来计算每个长度为n的二进制串中1的个数，这题也能用类似的方法。

对于任一集合S，我们都用一个链表来记录$span(S)$的基按大小排序之后形成的序列。我们从按S的元素个数从大到小枚举S，如果$S = \emptyset$，则其基数为0。

否则我们选出S中的任一元素x，用$span(S-\{x\})$的基从大到小依次对x进行消元操作（能使x变小就将其异或）。如果最后$x=0$，则$span(S)=span(S-\{x\})$；否则$span(S)$的基等于$span(S-\{x\})$的基并上最后生成的x，将其用链表存起来。

至此问题得到了解决。设P等于L中最大的数，算法的时间、空间复杂度均为$O(2^n logP)$。

6.2 Codeforces Round #228 Div.1 D

6.2.1 题目大意

设S为一个在自然数域$\mathbb{Z}^+$上的集合。如果对于任意$a,b\in S$有$a\oplus b\in S$，则称S为一个完美的集合。

给定一个数n，问有多少个完美的集合满足集合中最大的数不超过n。

6.2.2 数据规模与约定

$n\le 10^9$

6.2.3 题目分析

为了简化题意，我们把本题中所有的数字均视为32位。那么一个数字实际对应的是一个32维的向量。

不难发现，一个完美的集合实际上对应的就是一个线性空间，我们可以用基来将其表示。也就是说，如果每个满足条件的线性空间我们都能找出唯一一组对应的基的话，那么问题就能得到解决。

利用高斯消元的思想，我们可以把一个完美集合对应的基用一个倒置的三角形（或者是梯形）来表示。又因为集合中最大的元素不能超过n，我们可以考虑使用数位DP从高到低逐位处理，一个一个加入基向量。

我们先不考虑不大于n的限制。当处理到第i位，已经加入了j个基向量时：

1. 考虑新加入一个基向量2^i。这时第i位的数字我们可以随意控制，所以可以把之前所有加入的向量的第i位都设为0。这时有1种取法。

2. 如果不加入新向量，可以证明无论之前的向量第i位如何取值，对应的线性空间都不会有重复。这时有2^j种取法。

使用DP即可解决问题，但这时还要考虑不大于n的限制。

设$g_{i,j}$表示已经处理了后i位，且产生了j个基向量时，有多少个可能的线性空间其中的最大值的后i位与n的后i位相等。同样设$f_{i,j}$表示有多少个线性空间中最大值的后i位比n的后i位小。可以发现，$f_{i,j}$转移的过程和上述过程完全一致，我们只需考虑如何转移$g_{i,j}$。当处理第i位，并有j个基向量时：

1. 当n的第i位等于1时：
   - 如果新加入一个基向量2^i，那么这个线性空间中最大值的后i位和n的后i位还是相等（对应$f_{i+1,j+1}$）。
   - 否则，我们有2^{j-1}种赋值方法使得后i位仍然相等（对应$g_{i+1,j}$），同样有2^{j-1}种取法使得后i位小于n（对应$f_{i+1,j}$）。
2. 否则，我们不能重新加入一个向量，且只有2^{j-1}种赋值方法使得后i位保持相等（对应$g_{i+1,j}$）。注意一个特例——当$j = 0$时也会有1种方法。

以上就是所有转移的过程。本题的答案即为：

$$\sum_{i=0}^{32} g_{0,i} + f_{0,i}$$

算法时间复杂度为$O(log^2 n)$，空间复杂度为$O(log^2 n)$。

6.3 HEOI2013 钙铁锌硒维生素

6.3.1 题目大意

给你n个n维向量（A类），再给你n个n维向量（B类）。保证A类向量之间线性无关。现你要给每个A类向量都从B类向量中选择一个备用向量，要求该向量被备用向量替换后新的n 个向量之间仍然线性无关。任两个A类向量的备用向量都不能相同。请找出这样的方案。

6.3.2 数据规模与约定

$n \leq 300$，所有出现的数字均为在$[0, 10000]$之间的整数。

6.3.3 题目分析

我们设A类向量的集合为S_A，同样B类向量的集合为S_B。

问题实际就是询问任两个A, B向量之间是否能替换，剩下的部分用二分图匹配即可。显然$span(S_A) = \mathbb{R}^n$，所有的B类向量都能被S_A的唯一一个线性组合表示。事实上可以证明：

> **定理 8**：一个B类向量x能替换一个A类向量y的充要条件是在S_A的元素表示x的线性组合中，y的系数不为0。

证明是显然的：如果y的系数不为0，那么y就能被x和S_A中的其他元素表示。也就是说，替换之后仍然满足$span(S_A) = \mathbb{R}^n$，S_A线性无关。否则，S_A中的其他元素可以直接组合出x，这些向量之间一定是线性相关的。证毕。

现在问题变成了怎么求x对应的线性组合的各项系数。这个利用类似高斯消元的方法就能解决。

至此问题得解。算法的时间复杂度为$O(N^3)$，空间复杂度为$O(N^2)$（**注意**，这并不是原题的复杂度）。

7 总结

线性相关虽说在OI中出现的频率并不算特别多，但是有很多与之有关的好题。与现在热门的数论不同，线性相关的题目往往会结合一些数学之外的东西，因为它本身并不复杂，但要求选手有较强的理解能力与分析能力才能将其运用到位。线性相关是美妙的，希望在未来的OI 界中能出现更多有关线性相关的题目，让其得到更多的应用。

8 鸣谢

- 感谢父母和学校对我的培养。
- 感谢CCF给我这次宝贵的机会。
- 感谢屈运华老师对我的支持与辅导。
- 感谢刘研绎，黄志翱等同学提供的积极帮助。

9 参考文献

[1] http://www.wikipedia.org ——维基百科。

[2] 同济大学数学系，《线性代数》，同济大学出版社。

[3] 刘雨辰，《对拟阵的初步研究》，2007年集训队论文集。

[4] Heidi Gebauer, Yoshio Okamoto, “Fast Exponential-Time Algorithms for the Forest Counting and the Tutte Polynomial Computation in Graph Classes”.

关于三维最小乘积生成树的一些研究

绍兴市第一中学 张恒捷

摘 要

本文在二维最小乘积模型的基础上，根据3D卷包裹求凸包原理，对最小乘积模型扩展到三维时的情况进行了一些研究。

1 三维最小乘积生成树的模型

n个点，m条边，每条边有a,b,c三种属性。求一个边集使得n个点通过选出的边能够两两到达。要求最小化边集中边的a属性之和与b属性之和与c属性之和的乘积。

注意其中的a,b,c必须大于等于0。

2 一维的情况

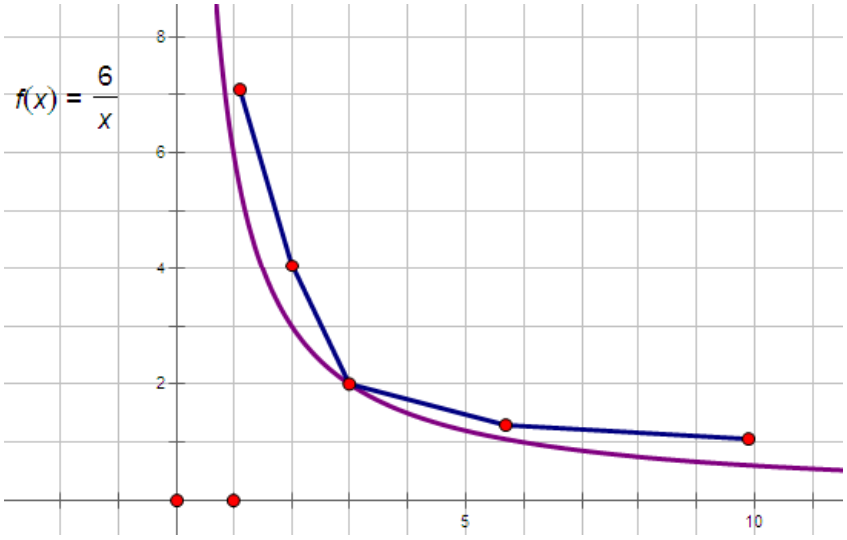
即普通的情况

3 二维的情况

对于一个可行的方案，把此方案下的a属性之和与b属性之和当做二维笛卡尔平面上的坐标，一个方案就对应为一个点。

一个性质

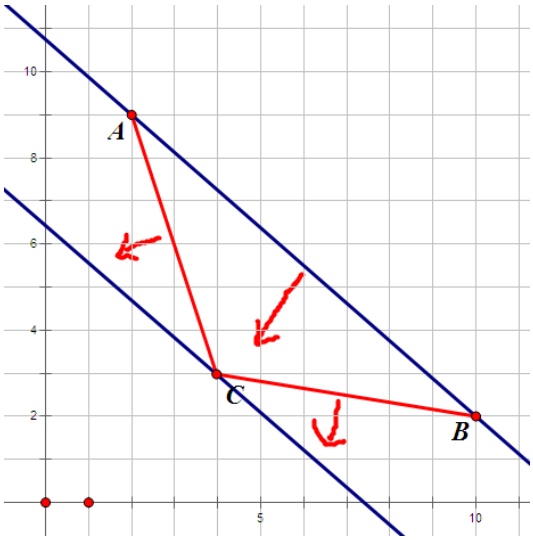
在所有点中，只有在下凸壳上的点会成为答案。



证明

证明不是下凸壳上的点一定不会是答案，只要证明在下凸壳边界处的点（不包括顶点），不会比答案优。取凸壳上相邻两个顶点研究，假如这两个顶点连线上的某个点比这两个顶点答案优，做一条经过此点的反比例函数，就会出现两个顶点都在此函数的上方的情况。由于不可能存在一条反比例函数使得存在在其上方的两个点，这两个点连线与此函数有交，因此凸壳相邻两个顶点连线上的所有点都劣与其中较优的顶点，证毕。

实现方法



定义work(A,B)为寻找在与AB连线垂直，方向与原点相反的向量上投影最靠近原点的点。假设work(A,B) 找到了点C，那么递归调用work(A,C)与work(C,B)就可以找到下凸壳在A与B之间的所有点了。假如A与B都是无穷远处且所有点都在这两个点连线往原点方向的半平面内,那么work(A,B)就可以找到所有下凸壳上的点了。

怎么找在某个向量上投影最近的点呢？设这个向量为u，d(v)为v这条向量在u上的投影长度，可以证明$\sum(d(v_i)) = d(\sum v_i)$,因此只需要把第i条边的权值设为向量$(a_i, b_i)$在u上的投影长度，再做普通的最小生成树就可以求出方案了。

4 三维的情况

与二维类似，在所有点中，只有在下凸壳上的点会成为答案。

4.1 乱搞

在数据随机的情况下，考虑到极值点在凸包上可能非常突出（尖），因此随机一条向量找最优值或者模拟退火的方法命中率非常高。在随机的情况下，乱搞是一种非常有力的算法。

4.2 三维向量

先介绍三维向量的各类运算以及其几何意义。

A向量加B向量：与二维类似，把B的开头接在A的结尾后，A的开头到B的结尾即为所求向量。用数学语言来描述为$(A.x + B.x, A.y + B.y, A.z + B.z)$

A向量减B向量：相当于加上B反方向的向量，用数学语言来描述为$(A.x - B.x, A.y - B.y, A.z - B.z)$

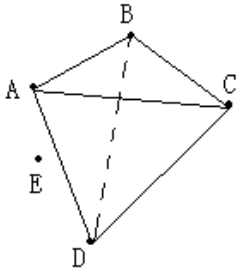
A向量点积B向量：结果是一个数字，表示A在B向量上的投影长度与B向量模长的乘积，也可以表示为A向量模长乘B向量模长乘cos夹角的值。用数学语言来描述为$(A.x \times B.x + A.y \times B.y + A.z \times B.z)$

A向量叉积B向量：结果是一个向量，它垂直于A向量与B向量且它的模长等于A与B组成的平行四边形的面积。方向用右手螺旋定则判断。用数学语言来描述为$(A.y \times B.z - A.z \times B.y, A.z \times B.x - A.x \times B.z, A.x \times B.y - A.y \times B.x)$

一个平面的法向量：一个垂直于平面的向量。

4.3 一个“显然”的做法

类比二维的方法，定义work(A,B,C)为寻找在平面ABC的法向量上投影最小的点。假设work(A,B,C)找到了点D，再递归调用 work(A,B,D) 与 work(A,D,C) 与 work(D,B,C)。很可惜，这种做法是错误的。因为在二维空间中，直线AC 向原点一侧的半平面与直线BC向原点一侧的半平面的交集不会再成为凸包上的点了。但是三维的情况下不能保证这一点。比如下图：



点E藏在边AD的左侧。work(A,B,D)与work(A,C,D)可能会同时找到E。

4.4 一个正确的做法

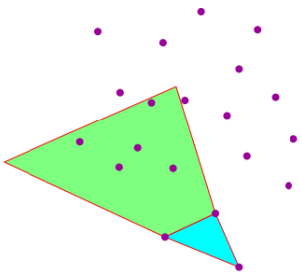
考虑另一个问题：

三维空间中有n个点，求出它们的凸包

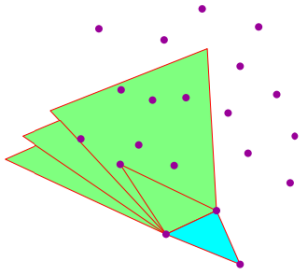
其实上一个做法错误的本质是那个方法本身便无法在三维空间中求出一个完整的凸包。所以作者选用了3D卷包裹法作为本算法的基础。

普通的3D凸包的卷包裹做法：一开始选择一条必定在凸包上的边，然后找到一个点满足这条边与这个点构成的平面的另一侧没有任何一个点，这样便找到了凸包上的一个面，递归这个面的另外两条边做同样的事情即可，直到没有新的边加入为止。

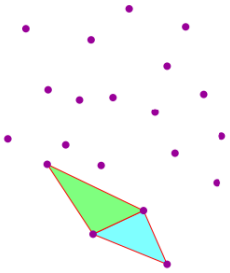
3D: Gift wrapping



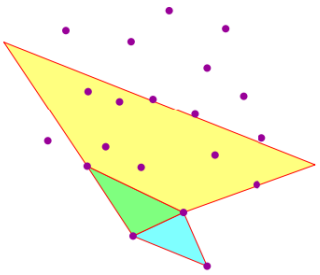
3D: Gift wrapping



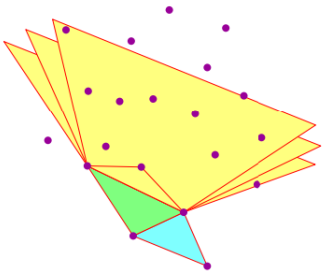
3D: Gift wrapping



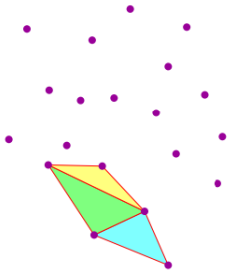
3D: Gift wrapping



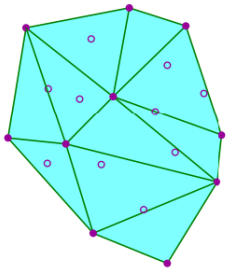
3D: Gift wrapping



3D: Gift wrapping



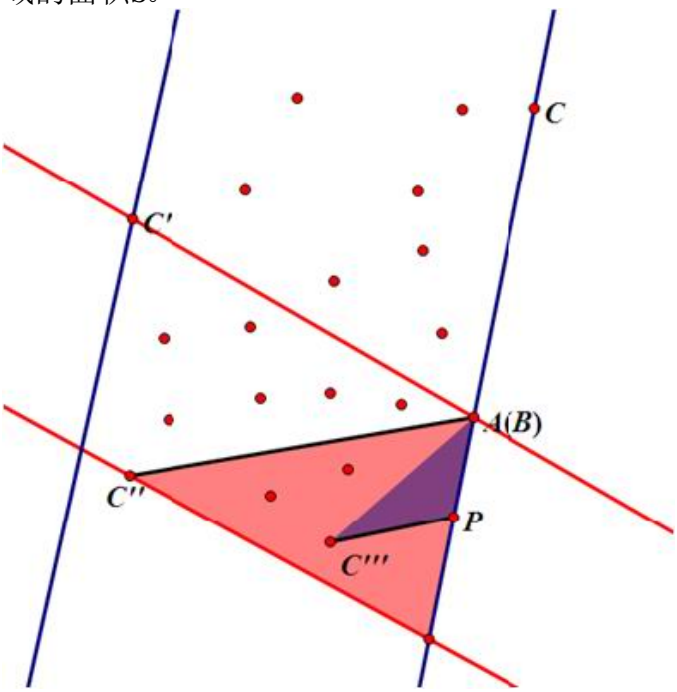
3D: Gift wrapping



类比卷包裹做法，假设已经找到了凸壳上的一个面ABC，要找点D使得ABD的一侧没有一个点。直接找可能比较困难，介绍一个作者找到的方法：找到距离面ABC最远的点C'，再找距离面ABC'最远的点C''，不停地迭代即可找到D。

复杂度证明

为了方便，将所有点投影到以向量AB为法向量的平面上，考虑D可能存在的区域的面积S。



假设现在已经找到了C''，S''为红色三角形的面积，新找到C'''后，新的S'''应该为紫色三角形的面积。由于直线AC''与直线PC'''平行，所以S'''至

少在S''的基础上缩小一半以上。

在假设所有点都是格点的情况下，最终$S < 1$时，D点必定已经被找到了。因此复杂度不会超过$\log_2 area$

因此，只有在所有点都是在a,b,c属性均为整数时，此算法复杂度为$O(h * \log_2 area)$,其中h为凸包上的点数。

5 扩展

其实不只是生成树可以套最小乘积的模型，其他经典算法也可以套。下面介绍最小割模型成功与最小乘积结合的例子。

n个点，m条边的无向图，每条边有三个属性a,b,c，求一个边集使得删掉这个边集后，s 无法到达t。要求最小化边集中边的a属性之和与b属性之和与c属性之和的乘积。

起初，做法都和生成树相同；维护凸包上的面，找投影长度最短的点时，将边重新设为另一个权值，直接做最大流。但是当提起方案时，用最大流做最小割好像很奇怪。

这里给出一种求方案的做法：将点集分为两个部分，分别是在残余网络中s能走到的点的集合(设为A) 与走不到的点的集合(设为B)，连接这两个集合的边就是一个合法的方案。简要证明一下：设s 等于这些边的容量之和。跑最大流算法时一条增广路会不停在两个集合间穿梭，不管怎么样从A 走入B的次数一定等于B 走进A的次数+1，所以s会减少本次增广路的流量。又因为最终s=0，所以可知原来的s 等于最大流的流量。又因为最大流=最小割，所以证出该方案是一个可行解。

可能还有其他模型能和最小乘积模型嵌套，只要满足这一类模型能够求出方案即可。

6 参考文献

唐文斌2012年冬令营讲课

7 感谢

感谢我的指导老师陈合力，游光辉，董烨华，邵红祥长期以来的帮助。
感谢俞鼎力，董宏华对我的帮助。

浅谈回文子串问题

江苏省常州高级中学 徐毅

摘 要

本文首先对用后缀数组求回文半径的算法进行了简单回顾，紧接着对Manacher算法进行了详细介绍，在此基础上对几种类型的回文子串问题分别举例分析，最后对解题的一般思路进行了总结。

1 引言

近年来，随着处理字符串的算法和数据结构在国内的不断推广，与字符串相关的问题越来越多地进入大家的视线。

回文子串问题，是与字符串相关的诸多问题中比较热门的问题之一。由于回文子串有许多优美的性质，此类问题往往灵活性较强。

为此，本文对此类问题进行了一定的分析和总结，希望能起到抛砖引玉的作用。

2 基本概念

- 字符的非空有限集，称为**字母表（alphabet）**。
- 字母表中字符的有限序列，称为**字符串（string）**。
- 字符串中字符的个数，称为该字符串的**长度（length）**。
- 字符串中连续的一段，称为该字符串的**子串（substring）**。
- 字符串中反转后与反转前相同的子串，称为该字符串的**回文子串（palindromic substring）**。

- 字符串中长度最大的回文子串，称为该字符串的**最长回文子串（longest palindromic substring）**。
- 以字符串第i位为中心的回文子串的最大长度的一半，称为该字符串第i位的**回文半径（radius of palindrome）**。

3 常用算法

在处理回文子串问题时，一般需要求出字符串每一位的回文半径。

为了避免奇偶讨论和边界问题，从而降低代码复杂度，我们在字符串每一位两侧都添加同一个特殊字符，在字符串的首位前添加一个不同的特殊字符，在字符串的末位后再添加一个不同的特殊字符，如将abbabcba变为$#a#b#b#a#b#c#b#a#@。

由此我们得到了一个长度为n的新字符串$S[1..n]$，而要求出该字符串每一位的回文半径$R[1..n]$，通常可以采用以下两种算法。

Table 1: S和R的对应关系

S	#	a	#	b	#	b	#	a	#	b	#	c	#	b	#	a	#
R	1	2	1	2	5	2	1	4	1	2	1	6	1	2	1	2	1

3.1 后缀数组

后缀数组的构造大家应该都很熟悉，在这里就不赘述了。而利用后缀数组来求字符串每一位的回文半径，前人也有过介绍，我们再简单回顾一下。

$R[i]$显然等于$S[i..n]$和反转后的$S[1..i]$的最长公共前缀。因此，我们将反转后的原字符串写在原字符串后面，中间用一个特殊字符隔开，这样就把问题变为了求这个新字符串某两个后缀的最长公共前缀，可以很方便地用后缀数组来完成。

- 若使用倍增算法来构造后缀数组，RMQ问题用$O(n\log n)$的时间复杂度来预处理，则总时间复杂度为$O(n\log n)$。
- 若使用DC3算法来构造后缀数组，RMQ问题用$O(n)$的时间复杂度来预处理，则总时间复杂度为$O(n)$。

3.2 Manacher算法

要想顺利解决回文子串问题，通常需要用$O(n)$的时间复杂度来求$R[1..n]$，而利用后缀数组来实现的代码复杂度较高。因此，我们应当利用问题的特殊性，熟练使用下述短小精悍的Manacher算法。

3.2.1 算法流程

在Manacher算法中，我们添加两个辅助变量mx和p，分别表示已有回文半径覆盖到的最右边界和对应中心的位置。

计算$R[i]$时，我们先给它一个下界，令i关于p的对称点$j = 2p - i$，分以下三种情况讨论（示意图中，上方线段描述了字符串，中间线段描述了$R[p]$，左下方线段描述了$R[j]$，右下方线段描述了$R[i]$的下界）：

- $mx < i$时，只有$R[i] \geq 1$；
- $mx - i > R[j]$时，以第j位为中心的回文子串包含于以第p位为中心的回文子串，由于i和j关于p对称，以第i位为中心的回文子串必然也包含于以第p位为中心的回文子串，故有$R[i] = R[j]$；
- $mx - i \leq R[j]$时，以第j位为中心的回文子串不一定包含于以第p位为中心的回文子串，但由于i和j关于p对称，以第i位为中心的回文子串向右至少会扩展到mx的位置，故有$R[i] \geq mx - i$。

Figure 1: $mx < i$的情况

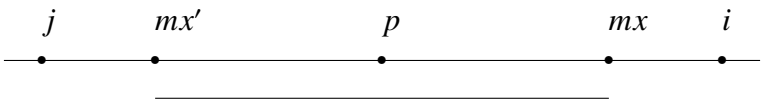


Figure 2: $mx - i > R[j]$的情况

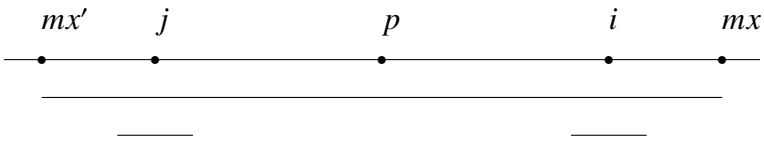
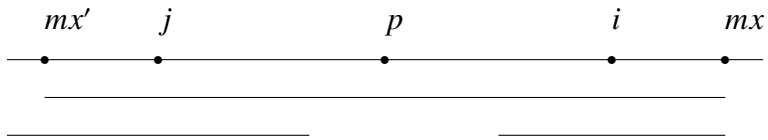


Figure 3: $mx - i \leq R[j]$的情况



对于超过下界的部分，我们只要逐位匹配就可以了。

由于逐位匹配成功必然导致mx右移，而mx右移不超过n次，故Manacher算法的时间复杂度为$O(n)$。

3.2.2 伪代码

MANACHER'S-ALGORITHM(S)

```
 1  p ← 0
 2  mx ← 0
 3  for i ← 1 to n
 4       do if mx > i
 5             then R[i] ← MIN(R[2p − i], mx − i)
 6             else  R[i] ← 1
 7          while S[i + R[i]] = S[i − R[i]]
 8                do R[i] ← R[i] + 1
 9          if i + R[i] > mx
10             then p ← i
11                  mx ← i + R[i]
12  return R
```

4 例题分析

4.1 例题一：拉拉队排练[1]

4.1.1 题目大意

给出一个长度为$n(1 \le n \le 10^6)$的字符串，求前$k(1 \le k \le 10^{12})$长的长度为奇数的回文子串长度的乘积除以19930726的余数或判断长度为奇数的回文子串不足k个。

4.1.2 算法分析

首先，我们使用Manacher算法求出该字符串每一位的回文半径。

显然，回文子串的长度不超过n种，因此我们可以想办法统计出每种长度回文子串的个数$cnt[1..n]$。设以第i位为中心的回文子串的可能长度区间为$[l_i, r_i]$，则我们要实现对$cnt[l_i..r_i]+1$，这个子问题非常经典，差分后每次只要对$cnt[l_i]+1$，对$cnt[r_i+1]-1$，最后再求一遍前缀和就可以了。

下面，我们只要按回文子串长度从大到小依次处理（当然只处理长度为奇数的），每次利用快速幂将相应长度的乘积累乘到答案中，直到长度为奇数的回文子串总个数达到k。

时间复杂度为$O(n \log n)$。

4.2 例题二：Palisection[2]

4.2.1 题目大意

给出一个长度为$n(1 \le n \le 2 \times 10^6)$的字符串，求相交的回文子串对数。

4.2.2 算法分析

首先，我们使用Manacher算法求出该字符串每一位的回文半径，同时求出回文子串的总个数，进而得到回文子串的总对数。

[1]题目来源：2011年国家集训队互测

[2]题目来源：Codeforces Beta Round #17

利用补集转化思想，我们可以将问题转化求不相交的回文子串对数，显然用回文子串的总对数减去不相交的回文子串对数就得到了相交的回文子串对数。

令$S_1[i]$表示以第i位结尾的回文子串个数，$S_2[i]$表示以第$j(1 \le j \le i)$位结尾的回文子串个数和，显然$S_2[i] = S_2[i-1] + S_1[i]$。设当前处理到以第$i$位为中心的回文子串与左侧回文子串的不相交对数，我们要求的就是

$$\sum_{j=i-R[i]}^{i-1} S_2[j]$$

。为了快速得到上式的值，我们再求前缀和$S_3[i] = S_3[i-1] + S_2[i]$，则上式就为$S_3[i-1] - S_3[i-R[i]-1]$。至此，单次处理就能在$O(1)$的时间复杂度内完成。

时间复杂度为$O(n)$。

4.3 小结一

对于例题一和例题二这样与回文子串相关的简单计数问题，我们通常可以在使用Manacher算法求出字符串每一位回文半径的基础上，利用前缀和等常用手段将核心问题在$O(n)$的时间复杂度内解决。

4.4 例题三：最长双回文串[3]

4.4.1 题目大意

给出一个长度为$n(2 \le n \le 10^5)$的字符串，求最长双回文子串T（双回文指可将T分为非空连续两部分X, Y，且X和Y都是回文串）的长度。

4.4.2 算法分析

首先，我们使用Manacher算法求出该字符串每一位的回文半径。

考虑枚举分界点i，我们要使得以第i位为右边界和左边界的回文子串都最长，也就是要分别找到在第i位左右两侧回文半径可覆盖至第i位的离第i位最远的位置。

[3]题目来源：2012年国家集训队清华集训

那么，以找左侧最优位置为例，我们从左往右扫描，同时维护一个当前处理到的右边界。扫描到第i位时，我们只要更新从当前右边界的后一位到$i + R[i]$这段范围里每一位的左侧最优位置，同时更新右边界为$i + R[i]$。

其正确性是显然的，因为如果左右两个位置同时覆盖到一个位置，那么选左边那个必然更优。找右侧点同理。

得到了左侧和右侧的最优位置后，我们也就得到了以第i位为分界点的最长双回文子串长度，取最大值即为最后的答案。

时间复杂度为$O(n)$。

4.5 例题四：双倍回文[4]

4.5.1 题目大意

给出一个长度为$n(1 \le n \le 5 \times 10^5)$的字符串，求最长双倍回文子串$T$（双倍回文指$T$是长度为4的倍数的回文串，且$T$的前一半和后一半也是回文串）的长度。

4.5.2 算法分析

首先，我们使用Manacher算法求出该字符串每一位的回文半径。

考虑从左往右枚举中心i，我们需要找到最小的子中心$j(j < i)$，使得第j位的回文半径能覆盖到第i位，并且第i位到第j位的距离的2倍不能超过$R[i]$。

我们可以很容易得到满足“第i位到第j位的距离的2倍不超过$R[i]$”的j的左边界$k = i - \lfloor \frac{R[i]}{2} \rfloor$。那么，我们要找的就是在第$k$位右侧离第$k$位最近的回文半径能覆盖到第$i$位的位置$j$。

我们维护一个可能的最优位置表，在表中找到第k位右侧离第k位最近的未被删除的位置后，若该位置的回文半径不能覆盖到第i位，就把该位置从表中删除（因为该位置不可能再被右侧的中心所用），继续重复该操作，直到找到的位置的回文半径能覆盖到第i位或已经找到第i位为止。

对于一个表，我们要支持删除操作和查询某个位置右侧最近的未被删除的位置，这个就很容易用并查集来维护了。

对以每一位为中心的最长双倍回文子串长度取最大值即为最后的答案。

[4]**题目来源**：Shanghai Team Selection Contest 2011

时间复杂度为$O(n\alpha(n))$。

4.6 例题五：Tricky and Clever Password[5]

4.6.1 题目大意

一个长度为l（l为奇数）的回文串T可以被加密为$A+T[1..x]+B+T[x+1..l-x]+C+T[l-x+1..l]$（+表示字符串连接，$x$为满足$2x<l$的任意非负整数，$A,B,C$为可以为空的任意字符串）。给出一个长度为$n(1\le n\le 10^5)$的密文$S$，求$T$的最大可能长度。

4.6.2 算法分析

首先，我们使用Manacher算法求出该字符串每一位的回文半径。

我们注意到，$T[x+1..l-x]$也是一个长度为奇数的回文串，且其中心就是T的中心。

考虑枚举中心i，将$T[x+1..l-x]$设为$S[i-R[i]+1..i+R[i]-1]$必然是最优的，这很容易通过讨论B,C是否为空来证明。

接下来，我们要最大化x，也即要求反转后的$S[i+R[i]..n]$在$S[1..i-R[i]]$中的最大匹配长度。令反转后的S为S'，我们可以先用KMP算法预处理出$S_1[i]$表示满足$S[i-k+1..i]=S'[1..k]$的最大的k，再求前缀最大值$S_2[i]=\max\{S_2[i-1],S_1[i]\}$。这样，以第$i$位为中心的$T$的最大可能长度就为$2(R[i]+\min\{S_2[i-R[i]],n-i-R[i]+1\})-1$。

对以每一位为中心的T的最大可能长度取最大值即为最后的答案。

时间复杂度为$O(n)$。

4.7 小结二

对于例题三、例题四和例题五这样与回文子串相关的求最优字符串问题，我们通常可以在使用Manacher算法求出字符串每一位回文半径的基础上，用$O(n)$的时间复杂度枚举所求字符串的中心（或分界点），分析题中的约束条件后结合一些数据结构或是辅助算法来加速单次处理。

[5]**题目来源：** Codeforces Beta Round #30

4.8 例题六：Palindromic Substring[6]

4.8.1 题目大意

给出一个长度为$n(1 \le n \le 10^5)$的小写字母字符串，进行$m(1 \le m \le 20)$次询问，第i次询问给每个字母一个$[0,25]$的权值，求权值第k_i小的回文子串（保证存在）的权值（回文子串的权值指将该回文子串的前一半提取出来，每一位用字母对应的权值替代后形成的二十六进制数除以777777777的余数）。

4.8.2 算法分析

定理1. 一个字符串只有$O(n)$个**本质不同**的回文子串。

Proof. 我们注意到，在Manacher算法中，只有mx右移时才会产生新的回文子串（否则一定存在对称的回文子串），而mx右移不超过n次，故一个字符串只有$O(n)$个本质不同的回文子串。 □

据此，我们可以用Manacher算法在$O(n)$的时间复杂度内得到所有本质不同的回文子串的位置。

对于每次询问，我们都可以用类似字符串Hash的计算方法来得到每种回文子串的权值。

而为了求出第k小的，我们需要得到每种回文子串在该字符串中出现的次数，这个子问题可以通过构造后缀数组后二分来解决。

接下来，我们只要将每种回文子串的权值从小到大排序，依次累加出现次数直到不小于k，则对应的权值就是答案。

时间复杂度为$O(mn \log n)$。

4.9 例题七：Palindromic Equivalence[7]

4.9.1 题目大意

给出一个长度为$n(1 \le n \le 10^6)$的小写字母字符串S，求和S回文性质相同的字符串T（回文性质相同指$T[i..j]$是回文串当且仅当$S[i..j]$是回文串）的个数。

[6]**题目来源：** Asia Changchun Regional Contest 2012
[7]**题目来源：** 11th POI Training Camp

4.9.2 算法分析

我们发现，S和T的回文性质可以用一些相等和不等关系来表示。对于以第i位为中心的回文子串，显然有$T[i-k] = T[i+k](1 \le k < R[i])$，以及$T[i-R[i]] \neq T[i+R[i]]$。

为了进行统计，我们肯定要得到这些相等和不等关系。

定理2. *一个字符串的回文性质可以用$O(n)$个相等和不等关系来表示。*

Proof. 我们注意到，在Manacher算法中，只有mx右移时才会产生新的相等关系（否则一定存在对称的相等关系），而mx右移不超过n次，故相等关系的数量是$O(n)$的。

由于以每一位为中心都只会产生一个不等关系，故不等关系的数量同样是$O(n)$的。

综上，一个字符串的回文性质可以用$O(n)$个相等和不等关系来表示。 □

据此，我们可以用Manacher算法在$O(n)$的时间复杂度内完成将相等的位置合并只保留最前的一个，并在不等的位置间连边，相当于形成了一个新字符串，边代表不等关系。

下面的问题相当于给出一个无向图，要求用26种颜色给点染色，使得每条边的两个点颜色不同，求方案数。

然而，对于一般的无向图，这个问题是没有有效算法的，因此我们还要进一步挖掘这个图的特殊性。

定理3. *将一个字符串T的相等位置合并只保留最前的一个后得到的新字符串T'，若存在不等关系$T'[i] \neq T'[j](j < i), T'[i] \neq T'[k](k < i)$，则一定有不等关系$T'[j] \neq T'[k]$。*

Proof. 我们仍然从Manacher算法的流程来思考，计算$R[i]$时，令i关于p的对称点$j = 2p - i$，分以下情况讨论：

- $mx < i$时，mx发生右移，相当于向T'末尾添加字符$T[mx]$，使之成为当前活跃字符，并有初始不等关系$T[i - R[i]] \neq T[mx]$；
- $mx - i > R[j]$时，所得的不等关系$T[i - R[i]] \neq T[i + R[i]]$是冗余的（一定存在对称的不等关系），故无需考虑；

- $mx - i \leq R[j]$时，若mx发生右移，则同$mx < i$的情况；若mx不发生右移，则我们使得当前活跃字符又多了一个不等关系$T[i - R[i]] \neq T[mx]$。

设当前活跃字符为$T[i]$，存在不等关系$T[i] \neq T[j](j < i), T[i] \neq T[k](k < i)$，不妨设$k < j$，可得$T[k + 1..i - 1]$和$T[j + 1..i - 1]$均为回文子串，由对称进一步得到$T[j] = T[k + i - j]$，$T[k + 1..k + i - j - 1]$是回文子串。

那么，在Manacher算法中一定会比较$T[k]$与$T[k + i - j]$，又因为$T[j] = T[k + i - j]$，故$T[j]$与$T[k]$要么一定相等，要么一定不等。而一定相等时，$T[k]$与$T[j]$在T'中是同一个位置。

综上，若存在不等关系$T'[i] \neq T'[j](j < i), T'[i] \neq T'[k](k < i)$，则一定有不等关系$T'[j] \neq T'[k]$。 □

得到该性质后，$T'[i]$的可能字母数就是26减去不等关系$T'[i] \neq T'[j](j < i)$的个数，故可以很方便地完成计算。

时间复杂度为$O(n)$。

4.10 小结三

对于例题六和例题七这样与回文子串相关却又很难直接下手的问题，我们往往需要分析问题的瓶颈，利用Manacher算法的流程来证明某种对象的数量级（通常是$O(n)$的），并得到一些有用的性质从而进一步简化问题。

5 总结

对于回文子串问题，我们首先要能熟练掌握求回文半径的基本算法，尤其是深入理解Manacher算法，这对分析回文子串的诸多性质来说是必不可少的。

在此基础上，我们要抓住题中的条件仔细分析，善于利用回文子串的性质来简化问题，并能够结合多种算法和数据结构解决问题。

致谢

- 感谢CCF提供的学习和交流平台。
- 感谢学校的支持和曹文老师的栽培。

- 感谢许多同学在论文完成过程中对我的帮助。
- 最后，感谢父母的养育之恩。

参考文献

[1] Maxime Crochemore, Wojciech Rytter Mokhtar, “Text Algorithms”, Oxford University Press.

[2] 罗穗骞, 《后缀数组——处理字符串的有力工具》, 2009年国家集训队论文。

浅谈维护多维数组的方法在数据结构题中的应用

合肥一中 梁泽宇

摘 要

近几年来，信息学竞赛中的数据结构领域快速发展，出现了很多新颖且有用的解决数据结构题的方法和思想，如分块思想、嵌套数据结构、可持久化和可撤销数据结构、对时间分治、整体二分、标号法等。然而，这些方法有时也会体现出一些弊端，如代码量太大、易写错、常数太大等。本文提出一种基于维护多维数组来解决数据结构题的方法，介绍这种方法的基本原理、应用、和其它方法的比较，并举了一些例题。希望本文能引起各位读者的深入研究，起到抛砖引玉的效果。

1 引入

1.1 可修改区间第K小问题[1]

给出一个正整数序列$A[1..n]$，两种操作：

- 将$A[i]$修改为x；
- 询问$A[l..r]$中的第K小值；

这是一个经典问题，目前已有**权值线段树套平衡树**、**权值树状数组套平衡树**、**重量平衡树套线段树**、**权值块状数组套平衡树**等多种解法。这些解法的共同点是需要使用嵌套数据结构，而嵌套数据结构的代码量和常数往往都是惊人的，实用效果并不好。

我们换一个角度审视这些嵌套数据结构。如果在外层数据结构的结点i套的内层数据结构中存在结点j，则这可以用一个pair$\langle i, j\rangle$来表示，因此，我们可以

[1]经典问题

用若干个pair来表示出整个数据结构中的所有元素。进一步，由于任何数据结构都可以用数组来存储，所以这里的i和j都可以是整数，表示数组下标，这样，我们就可以将这些pair表示到二维数组里，然后，对这个嵌套数据结构的操作就变成了对这个二维数组的操作。

当然，我们还可以做得更直接一些，直接建立一个二维数组$A'[1..n][1..W]$（其中W为序列值的范围），满足：

$$A'[i][j] = \begin{cases} 1, & A[i] = j \\ 0, & A[i] \neq j \end{cases}$$

我们称A'为A的**权值二维数组**。

对于询问操作，假设有一个值W_r，则区间和$\sum_{i=l}^{r}\sum_{j=1}^{W_r} A'[i][j]$即为$A[l..r]$中值在$[1..W_r]$之间的数的个数，二分这个$W_r$即可得到询问结果。对于修改操作，直接将$A'[i][A[i]]$减1、$A'[i][x]$加1，并将$A[i]$的值改为$x$即可。

问题是，A'数组不可能全部存下来，因为它的大小是$O(nW)$的，无法承受。但是，注意到A'数组中的值只有0和1两种，且在操作的时候并不需要知道每一个$A'[i][j]$的值，只需要知道A'数组的一些区间和，且本题对于区间和的操作是“改点求段[2]”的，因此，可以用二维树状数组来存储A'的部分和，建立二维树状数组$FT(A')$，满足：

$$FT(A')[i][j] = \sum_{i'=i-\text{lowbit}(i)+1}^{i}\ \sum_{j'=j-\text{lowbit}(j)+1}^{j} A'[i][j]$$

$FT(A')$数组的大小也是$O(nW)$，因此也不能全部存下来。不过，我们可以发现一个性质：

性质1.1. *若序列的长度为n，之前已进行了Q次修改操作，则$FT(A')$中值非0的元素个数为$O((n+Q)\log n\log W)$个。*

证明：一开始，A数组为空，则A'和$FT(A')$数组全部为0。先依次加入A中的n个初始元素，每加入一个元素，相当于A'中有一个元素被修改，引起$FT(A')$中的$O(\log n\log W)$个元素被修改。对于一次修改操作，由于在A'中只修改$O(1)$个元素，所以$FT(A')$中也只有$O(\log n\log W)$个元素被修改。综上，$FT(A')$中最多只有$O((n+Q)\log n\log W)$个元素值非0，证毕。

[2]即“修改操作只修改一个元素，询问操作询问一个区间”

因此，我们可以只记录$FT(A')$中的这些值非0的元素，然后在引用$FT(A')$的元素时，在这些记录的元素里找，如果找到就引用其值，否则引用0。修改的时候，如果某个元素的值从0变为非0，就记录这个元素，若从非0变为0，就将其从记录元素中删除。一种比较好的方法是用hash表存储这些记录的元素，这样可以保证每次查找元素的时间复杂度为$O(1)$。

下面分析整个算法的时间复杂度。对于修改操作，由于最多涉及$FT(A')$中的$O(\log n \log W)$个元素，因此时间复杂度显然是$O(\log n \log W)$的。对于询问操作，如果暴力二分W_r再求区间和，时间复杂度将升到$O(\log n \log^2 W)$，不过注意到，树状数组的本质是位运算，因此可以在二分的时候这样做：设上次二分的值为W'_r，本次是倒数第s次二分，则显然有$\text{lowbit}(W'_r + 2^{s-1}) = 2^{s-1}$[3]，因此，用上次二分时得到的$\sum_{i=l}^{r}\sum_{j=1}^{W'_r} A'[i][j]$的值加上$FT(A')[\,][W'_r + 2^{s-1}]$中的对应值，就是$\sum_{i=l}^{r}\sum_{j=1}^{W'_r+2^{s-1}} A'[i][j]$的值。这样，每次二分时只会涉及$FT(A')$中的$O(\log n)$个元素，一次询问的总时间复杂度为$O(\log n \log W)$。加上预处理的时间，我们可以利用这种方法在$O((n+Q)\log n \log W)$的时间复杂度内解决本题。

1.2　维护多维数组

从可修改区间第K小问题的解决可以看出，利用这种维护多维数组的方法往往可以免去嵌套数据结构，改用树状数组等简单的数据结构，大大降低了代码量和常数。其实，对于很多数据结构题目，我们都可以建立一个多维数组（一般是二维，有时可能达到三维或更多维），然后利用这个数组的区间和或其它信息得到原题所需求出的量，维护这些信息即可解决原题。一般来说，这个数组本身的规模都很大，超过了题目允许的范围，因此不能将它全部存下来，只能存储它的部分元素，对于其它的元素，根据它们自身的特点（如全是0或其它同一个值、和别的元素相同或者有其它明显规律），在需要引用的时候直接算出其值。

对于这个多维数组，一般需要使用基本数据结构来维护它，基本数据结构主要有以下两种（均为对数组的某一维而言，设这一维规模为n）：

[3]因为倒数第s次二分时，右$s-1$位还没有被决定，均视为0，只要在此次二分中将右起第s位置为1即可得到此式。

树状数组或线段树

D维数组，一个区间拆成$O(\log^D n)$个区间，一个元素关联$O(\log^D n)$个区间。

（分层）块状数组

对于一维数组（序列），设有正整数K，建立K层块状数组，上起第$i(1 \le i \le K)$层将整个这一维分成$n^{\frac{i}{K}}$块，每块的大小为$n^{\frac{K-i}{K}}$。如果能保证上层的分点一定也是下层的分点，则该序列的任意一个区间都可以拆成每一层各$O(\sqrt[K]{n})$个块，一共是$O(K\sqrt[K]{n})$个。每个元素都关联K个块。

对于多维数组的情况较为复杂。以二维数组为例，如果仅仅建立K层二维块状数组，第i层的每块为$n^{\frac{K-i}{K}} * n^{\frac{K-i}{K}}$的正方形，则一个二维区间不能拆成每一层上只有$O(\sqrt[K]{n^2})$个块，时间复杂度无法保证。正确做法是对于所有满足$0 \le i, j < K$的二元组$\langle i, j\rangle$，都建立每块大小为$n^{\frac{i}{K}} * n^{\frac{j}{K}}$的一层，这样就可以保证任意一个二维区间在每层上拆成的块数都是$O(\sqrt[K]{n^2})$个了[4]，此时总的层数为K^2。这里的K，已经不再表示层数，而是表示决定分块大小的指数，所以这种分层块状数组应该称为**指数为K的分层块状数组**。对于更多维数组，可推出类似的结论，即

性质1.2. 对于D维数组建立指数为K的分层块状数组，则总共需要K^D层，每个D维区间都可以拆成$O(K^D N^{\frac{D}{K}})$个块，每个元素关联$O(K^D)$个块。

实际应用时，对于维数不同的数组，K的取值也不一样，一般一维数组K取$3 \sim 4$，二维数组K取$5 \sim 6$效果最好。

下面来看一个例题：

【例1】middle[5]

给出一个序列$A[0..n-1]$，设$midv[i][j]$表示$A[i..j]$的中位数，即$A[i..j]$中第$\lceil \frac{j-i}{2} \rceil + 1$小的元素。每次询问给出四个整数$a$、$b$、$c$、$d(0 \le a < b < c < d < n)$，表示求$max_{a \le l \le b, c \le r \le d}\ midv[l][r]$。强制在线。

[4]每一维都在一维区间上考虑，拆成$O(\sqrt[K]{n})$个，行列结合成$O(\sqrt[K]{n^2})$个。

[5]Author：陈立杰

解法

注意到中位数的本质也是第K小，因此我们可以借鉴之前的可修改区间第K小的做法。先二分一个值W_r，然后建立数组$C[W_r][0..n-1]$，若$A[i] \geq W_r$，则$C[W_r][i] = 1$，否则$C[W_r][i] = -1$。容易发现，

$$max_{a \leq l \leq b, c \leq r \leq d} \ midv[l][r] \geq W_r$$

当且仅当

$$\begin{aligned} & max_{a \leq l \leq b, c \leq r \leq d} \sum_{i=l}^{r} C[W_r][i] \\ = & \sum_{i=b+1}^{c-1} C[W_r][i] + max_{a \leq l \leq b} \sum_{i=l}^{b} C[W_r][i] + max_{c \leq r \leq d} \sum_{i=c}^{r} C[W_r][i] \\ \geq & 0 \end{aligned}$$

这样，问题转化为求一个序列的区间最大前/后缀和，可以使用线段树维护。

上述的数组C是一个二维数组，大小是$O(nW)$，不可能全部存下来。注意到对于一个区间$[l..r]$，$C[x][l..r]$和$C[x+1][l..r]$有至少一个元素不同当且仅当$\exists_{l<=i<=r} A[i] = x$，因此，对于线段树上的区间$[l..r]$，最多只有$r-l+1$个$x$值满足$C[x][l..r]$的最大前、后缀和是需要记录的，因为其它值都可以通过记录的值中不小于它的最小x值得到结果。根据线段树的性质易得，一共需要记录$O(n \log n)$个值，用hash表存储这些记录的值，就可以在$O(n \log n)$时空复杂度内解决本题。

1.3 动态序列的多维数组维护方法

在有的题目中，需要维护一个动态（有插入删除操作的）序列，使用动态标号法 [3]可以在每次插入操作均摊$O(\log n)$的时间复杂度内解决此类题目。以前，为了存储区间中元素的值的分布，需要使用标号线段树套值线段树，现在有了多维数组这个工具，我们可以发现一种更简单的方法：建立以标号为第一维，值为第二维的二维数组，当目前序列中标号为i的元素值为j时，该二维数组的元素$[i][j]$值为1，否则为0。然后，对于标号这一维仍然用线段树维护，值这一维则用树状数组维护，每次重新分配标号的时候，将待重新分配的原有标

号信息全部在二维数组中删去，新的标号信息在二维数组中加入即可。同样地，我们也只用hash表记录维护该二维数组的“线段树套树状数组”中所有值非0的元素。

我们也可以用分层块状数组来维护关于动态序列的多维数组，只是其时间复杂度一般较高，效果不好。

1.4 树的处理

下面将这种方法的应用从序列上扩展到树上。很容易想到，利用树的DFS序将树上的问题转化为序列上的问题。

定义1.1 (DFS序)**.** DFS遍历整棵树，每个结点在入栈的时候记录一次，得到的序列称为该树的**DFS序**。设$in[i]$为结点i在序列中出现的位置。

定义1.2 (欧拉DFS序)**.** DFS遍历整棵树，每个结点在入栈的时候记录一次，出栈的时候再记录一次，得到的序列称为该树的**欧拉DFS序**，又叫**括号序**。设$in[i]$为结点i的入栈记录在序列中出现的位置（又叫**左括号位置**），$out[i]$为结点i的出栈记录在序列中出现的位置（又叫**右括号位置**）。

树上记录的信息，主要有树的路径和子树两种，对于子树，可以使用DFS序（因为在DFS序中，任意一棵子树都是一个连续的区间），对于路径，则需要使用欧拉DFS序，关于欧拉DFS序有两个很重要的性质：

性质1.3. 对于一棵树上任意两个无前后代关系的结点u和v，若$out[u] <= in[v]$，则该树的欧拉DFS序中的$[out[u]..in[v]]$区间内所有**出现且仅出现一次**的结点以及$\mathrm{LCA}(u,v)$，就是树上路径$u \to v$中的所有结点。

性质1.4. 对于一棵树上的两个结点u和v，若u是v的祖先，则该树的欧拉DFS序中的$[in[u]..in[v]]$区间内所有**出现且仅出现一次**的结点，就是树上路径$u \to v$中的所有结点。

根据这两个性质，要处理树上的一条路径的信息，只需找到这棵树的欧拉DFS序的一个区间内，所有出现且仅出现一次的结点即可（对于LCA可以特殊处理）。如果操作是可反的，可以使用“反操作抵消”的方法，即一个结点第二次出现时就抵消它第一次出现的效果值，一种常见的方法是利用性质1.4，将路径$u \to v$拆成$[1, in[u]] + [1, in[v]] - [1, in[LCA(u,v)]] - [1, in[LCA(u,v).pr]]$，其中$x.pr$表示结点$x$的父结点。

2 应用

2.1 在一类效果值与出现次数有关的题目中的应用

有一类题目，它们需要求出一个区间（或一段路径、一棵子树等）的总效果值，然而其元素的效果值是与该元素本身的值在这个区间（路径、子树等）内的出现次数有关的。这时，普通的线段树、平衡树等数据结构往往无能为力，即使采用分块一类的方法也不是很好处理，然而，维护多维数组的方法就出现奇效了。

先给出一个定义，之后会用到：

定义2.1 (前k次出现位置序列)**.** 对于一个序列$A[1..n]$，定义序列$L_k(i) = max\{0, x \mid k = \sum_{\substack{j=x \\ A[j]=A[i]}}^{i-1} 1\}$，即$A[i]$往前第$k$次出现和$A[i]$相同的元素时的位置，如果不存在就为0。

【例2】homework[6]

给出一个序列$A[1..n]$和若干次询问，每次询问给出l、r、a、$b(1 \le l \le r \le n, 0 \le a \le b)$，要求回答

1．$A[l..r]$内有多少个元素的值在$[a..b]$之间；

2．$[a..b]$内有多少个数在$A[l..r]$内出现过至少一次；

$1 \le n \le 10^5$，时限10s。

解法

先将序列A离散化，这样元素值的范围就是$[0..n-1]$了。对于第一问，直接按照可修改区间第K小的方法，建立序列A的权值二维数组A'，求区间和$\sum_{i=l}^{r}\sum_{j=a}^{b} A'[i][j]$即可，用二维树状数组维护。然后，注意到第二问的结果实际上就是第一问的结果减去$[l..r]$**内前1次出现位置也在**$[l..r]$**内的元素的个数**，先求出序列A的$L_1[\,]$序列，问题转化为对于询问$[l..r]$，求出

$$\sum_{\substack{i=l \\ L_1 i \ge l}}^{r} 1$$

[6]Source: AHOI2013 Round 2 Day 2

建立以序列A的下标为第一维，A的L_1序列为第二维的二维数组，求相应区间和即可。每个操作两问的时间复杂度均为$O(\log^2 n)$（树状数组）。

【例3】糖果公园[7]

给出一棵有n个结点的树，每个结点i上有一个权值$V[i]$，另外给出一个序列$W[1\,..\,n]$。两种操作

- 修改一个点的权值；
- 询问树上的路径$u \to v$上所有结点的效果值之和，对于该路径上的一个结点i，如果它的权值是在该路径上第k次出现的（即前面已有$k-1$个结点的权值等于它），则它的效果值为$V[i]W[k]$。

$1 \le n$、询问次数$Q \le 10^5$，时限30s。

解法

本题是树上的问题，所以要先求出这棵树的欧拉DFS序，然后一条路径可以转化为一个区间内出现且仅出现一次的数的集合（对于LCA可以特判）。

建立数组$C[1\,..\,2n][1\,..\,2n]$，$C[i][j](i \le j)$表示**欧拉DFS序中位置i的存在对位置j的效果值的影响（增量）**，则询问区间$[l\,..\,r]$时，我们只需要求出$C[l\,..\,r][l\,..\,r]$（或$C[1\,..\,r][l\,..\,2n]$）的区间和即可。设$V'[i]$为欧拉DFS序位置i对应结点的权值，则对于这个数组C，容易发现当且仅当$i \le j$且$V'[i] = V'[j]$时，$C[i][j]$才有意义。因此，对于原树中出现次数超过$n^{\frac{1}{3}}$的权值，将其预存下来，在询问时特殊处理（这样的权值只有$O(n^{\frac{2}{3}})$个），其它权值对应的所有元素，在C中算出其值，具体来说有以下几种情况（这里假设$W[0] = 0$）：

- 若位置i和j对应相同结点，则$C[i][j] = -2V'[i]W[1]$，因为i和j的第一次出现的效果值同时被抵消；
- 若在区间$[i+1\,..\,j]$中存在某位置与j对应相同结点（即位置j对应结点在$[i+1\,..\,j]$中出现两次），则$C[i][j] = 0$，因为此时位置j的效果值已被完全抵消，为0，所以位置i的出现并不会改变其效果值；

[7]Source: WC2013

- 若在区间$[i\,..\,j]$中不存在与位置j对应相同结点的位置（即位置j对应结点在$[i\,..\,j]$中出现一次），设在区间$[i\,..\,j]$中有K个权值为$V'[i]$的结点出现且仅出现一次，则当i在区间$[i\,..\,j]$中出现一次时，$C[i][j] = V'[i](W[K] - W[K-1])$，即加上位置$i$对应结点的影响，当$i$在区间$[i\,..\,j]$中出现两次时，$C[i][j] = V'[i](W[K-1] - W[K])$，即抵消位置$i$对应结点的影响。

对于修改操作，如果涉及的权值出现次数不超过$n^{\frac{1}{3}}$，则在C数组中对所有该权值对应的元素全部暴力重算，一共$O(n^{\frac{2}{3}})$个元素；如果出现次数超过$n^{\frac{1}{3}}$，则只要特殊处理一下即可。

时间复杂度：如果使用二维树状数组维护数组C，则一次操作时间复杂度为$O(n^{\frac{2}{3}}\log^2 n)$；如果使用分层块状数组，假设指数取3（本题中指数取3比取大于3的整数效果更好），则一次操作的时间复杂度为$O(9\cdot n^{\frac{2}{3}})$，显然用分层块状数组的方法更优。

2.2 利用多维数组将不可反操作转化为可反操作

数据结构问题中的操作，可以分为两种：某些操作存在反操作，如加上一个值（反操作为减去这个值）、xor（反操作为再次xor）、逆序（反操作为再次逆序）、求区间和（可以通过区间减法得到）等，称为**可反**的操作；某些操作不存在反操作（或者反操作非常难实现），如赋值操作、求区间的最小（大）值或第K小（大）值（不能通过区间减法得到）等，称为**不可反**的操作。

一般来说，可反操作的处理比不可反操作容易，因此当题目中出现不可反的操作时，可以想办法将它转化为可反的操作，利用多维数组往往能实现这一点。比如，一开始的可修改区间第K小问题，就是将“第K小”这个不可反操作通过“加一维变成权值数组”的方式转化为了“区间和”这个可反操作。再比如对于赋值操作，可以将所赋的值作为新的一维，并记录下每个元素赋某个值的最新时间，然后要引用某个元素的值时，找到记录的时间的最大（即最新）的值，再转化为区间和（有时可以直接使用最大值，不必转化为区间和）即可。

下面是相关的例题。

【例4】Tree[8]

给出一棵n个结点的树，每个结点上有一个非负整数权值。有q个询问操作，每个询问(u,v,z)，表示求路径$u \to v$上的所有结点的权值xor z的最大值。$1 \le n,q \le 10^5$，$0 \le z$,结点权值$< 2^{16}$，时限5s。原题允许离线，但这里强制在线。

解法

本题仍是关于树上路径的，因此需要先求出这棵树的欧拉DFS序，并建立其权值二维数组C：当结点i的权值为j时，$C[in[i]][j] = 1, C[out[i]][j] = -1$，$C$数组的其它元素值为0。这样，根据性质1.3，路径$u \to v$可以拆成$[1, in[u]] + [1, in[v]] - [1, in[LCA(u,v)]] - [1, in[LCA(u,v).pr]]$。为求出最大的xor z的值，只需从高到低逐位二分，每位先枚举和z的该位不同的，如果路径$u \to v$中没有权值在这个范围内的，再枚举和z的改为相同的即可。显然，我们可以直接使用区间和来得到路径$u \to v$中权值在某个范围内的结点个数，用二维树状数组维护，结合其自然二分性质，本题可以在$O((n+q)\log n \log W)$时间内解决（$W = 2^{16}$）。

本题中，“最大值”是不可反的，但是我们通过建立权值数组，增加一维，将其转化为区间和这样的可反操作，从而使问题变得容易解决。

【例5】Just for fun EXT[9]

给出n个结点（一开始无边），每个结点有一个非负整数权值，四种可能的操作：

- 在结点u和v间加一条边，若u,v已连通则无视该操作；
- 修改一个结点的权值；
- 询问u到v的路径上权值第k大的结点，若u,v尚未连通则无视该操作；
- 询问u到v的路径上，所有权值小于等于z的结点的权值之积mod 28256292的余数，若u,v尚未连通则无视该操作。

$1 \le n \le 2 * 10^5, 1 \le$操作数$q \le 4 * 10^5$。强制在线。

[8]Source: HDU 4757
[9]Author: 陈立杰

解法

先考虑简化的情形，即这棵树一开始已给出（或者说是允许离线）。这时，只需类似上一题的解法，求欧拉DFS序并建立相应的权值二维数组，只不过要建立5个权值二维数组，其中一个的权值对应位置值为1和-1，树状数组维护区间和，用于回答权值第k大的询问；另三个的权值对应位置的值分别为该权值中2、3、784897[10]的因数个数及其相反数（因数个数为0的可以不存储，不记录），树状数组维护区间和，最后一个的权值对应位置的值为权值本身去掉所有2、3、784897因数后的值及其mod 28256292的逆元，树状数组维护区间积，以上4个数组用于回答权值积的询问。修改时分别在这些数组里修改即可，时间复杂度$O((n+q)\log n\log W)$，其中$W = 28256292$。

回到原题。本题最难的一点在于加边操作——需要合并两棵树，即合并它们的欧拉DFS序和对应的权值二维数组。显然，我们可以使用启发式合并的方法，每次将较小的那棵树整体拆掉并以对应结点(u或v）为根重建成有根树，求出其欧拉DFS序后插入较大的那棵树的欧拉DFS序中的相应位置，并更新权值二维数组。为了实现这一点，一种方法是将其当成动态序列，使用动态标号法解决；另一种方法是使用块状数组来维护这些权值二维数组，每块的标志不是它在欧拉DFS序中的位置，而是它最左边的元素对应结点的编号。在合并的时候，先对重建好的较小树欧拉DFS序的各个权值二维数组分好块，然后可直接将这些块插入较大树的对应块中，唯一需要进行的处理是对于较小树的最后一个块可能要和较大树的插入位置之后的那个块合并（如果它们的总大小不超过$\sqrt{n}$的话）。前者（动态标号）一次加边操作的均摊时间复杂度为$O(\log^2 n\log W)$，后者（分块）为$O((\sqrt{n}+\sqrt{W})\log n)$。

2.3 利用多维数组实现可持久化

可持久化思想在数据结构题中起到了很重要的作用。可持久化思想，其实就是记录下所维护的数据结构在各个时刻的样子（即各个版本），并且利用各个版本间的共用部分进行压缩，以此来实现访问历史版本。实现可持久化数据结构时，一般是将“修改”变为“复制”，即复制修改后的元素，对原来的元素并不覆盖。

[10] $28256292 = 2^2 * 3^2 * 784897$。

任何数据结构都可以用数组方式存储，即用数组下标来表示元素（结点等），通过引用下标的域来表示元素（结点）之间的关系（其实地址方式存储和数组存储在某种意义上是类似的，只是将元素在内存中的地址作为下标而已）。如果以下标作为第一维，以时间轴作为第二维，建立二维数组，如果下标为i的结点在时刻j被修改，就将该二维数组的元素[i][j]置为修改后的值（包括其引用其它结点下标的域），然后，在引用某个结点在某个历史版本中的值的时候，就可以直接从该二维数组里调用——找到在这个历史版本及其之前，该结点最后一次被修改成的值即可。

如果每次只要对单个元素进行引用，可以不用二维数组存储，而使用C++ STL中的set——直接建立一个set数组，存储每个下标的修改历史记录，并按照时间排序。

【例6】Version Controlled IDE[11]

维护一个字符串，三种操作：

- 在某个位置插入一个长度不超过100的字符串；
- 删除一个子串；
- 询问时间tm（即第tm次插入删除操作之前）该字符串的第l到第r个字符组成的子串。

1 ≤操作数$q \le 5 * 10^5$，插入的字符串总长度不超过10^6，询问中输出的字符串总长度不超过$2 * 10^5$。

解法

对于这种动态序列（字符串）维护问题，一般使用伸展树(Splay Tree）进行维护。一个子串在伸展树中可以通过伸展操作形成一棵完整的子树，因此插入删除操作其实就是插入或一棵子树。

本题的难点在于如何询问历史版本。由于伸展树也是可以用数组方式存储的，所以只要用数组存储这棵伸展树，下标表示结点，然后按照上述方法，用二维数组（一维数组套set）存储每个结点的修改记录，询问时从中调用。

[11]Source: UVA 12538

此外还有一个问题，伸展树的时间复杂度是基于均摊分析的，可能在某个版本中伸展树的深度达到$O(n)$级别，此时如果询问操作都是关于这个版本的，时间复杂度就无法保证。所以，在每次插入删除之后，如果目前的伸展树深度太大，则要人为地进行一些伸展操作来进行调整（比如每次将深度最大的结点伸展到根，直到最大深度达到$O(\log n)$级别以内为止），这些伸展操作所造成的结点修改也要进行记录。

其实在这道题的解决过程中，我们只是引用单个元素，使用一维数组套set来实现可持久化，因此没有真正发挥多维数组的强大威力。其实，利用“下标+时间轴”的方法，我们可以借助区间和等操作实现一些比较复杂的应用，比如“询问第l到第r个版本的相关信息”、“找到符合某条件的版本个数”等。

3 总结

一切数据结构的本质都是数组，利用多维数组，往往能简化题目所涉及的复杂结构，更为直观地表示出题目中的各种数量关系，从而使题目变得容易解决。与数据结构题的传统方法相比，这种维护多维数组的方法在复杂度上的优势非常明显。这种方法的得出，其实也就是不满足于已有方法、对已有方法进行改进直至有所创新的过程。其实，任何新方法的出现，本质上都是“已有方法+人类智慧”这一过程。

我们要充分发挥人类智慧，探索、测试、改进解决方案的能力。[12]

参考文献

[1] 陈立杰，《可持久化数据结构研究》，2012。

[2] 许昊然，《数据结构漫谈》，2012。

[3] List Order Maintenance && Monotonic List Labeling Density Maintenance.

[4] http://en.wikipedia.org/wiki/Euler_tour_technique

[12] 出处：许昊然WC2014讲稿。

-- 空 --

线段树在一类分治问题上的应用

杭州学军中学 徐寅展

摘 要

线段树与按时间分治都是信息学竞赛中常用的算法。本文提出利用线段树解决分治问题的思想，给出一个能将区间修改问题转化为分治问题的方法，并将这种思想推广到了简单的可持久化问题中，最后本文介绍了虚树的概念以及构造方法。本文的例题比较基础，希望能起到抛砖引玉的作用。

1 线段树与按时间分治

1.1 线段树

线段树是一种很常用的数据结构。它的根表示的区间为总区间，每一个非叶子节点都有一个左儿子与一个右儿子。如果这个节点表示的区间为$[l,r]$，那么它的左儿子表示的区间为$[l,\lfloor\frac{l+r}{2}\rfloor]$，右儿子表示的区间为$[\lfloor\frac{l+r}{2}\rfloor+1,r]$。

找到一段区间在线段树中的对应位置的时间复杂度是$O(\log n)$的，这里n是总区间的大小。

简单证明：令插入的区间为$[a,b]$，子树i表示的区间为$[l_i,r_i]$，$mid_i=\lfloor\frac{l_i+r_i}{2}\rfloor$。若在某次访问子树$i$时需要同时访问左右两棵子树，那么有$a\le mid_i$以及$mid_i<b$。在$i$的左子树中，若又有一个j使得$a\le mid_j$以及$mid_j<b$，则由$a\le mid_j$以及$r_j\le mid_i<b$，得$[a,b]$肯定完全覆盖$j$的右子树。$i$的右子树中情况类似。于是同时访问两棵子树的次数$\le 1$，每一层的访问次数都是$O(1)$的，总复杂度为$O(\log n)$。

1.2 按时间分治

按时间分治是一种分治方法。它的基本思想是若要处理时间$[l,r]$上的修改与询问，就先处理$[l,\lfloor\frac{l+r}{2}\rfloor]$上的修改对$[\lfloor\frac{l+r}{2}\rfloor+1,r]$上的询问的贡献，之后递归处

理$[l,\lfloor\frac{l+r}{2}\rfloor]$与$[\lfloor\frac{l+r}{2}\rfloor+1,r]$。这种方法要求各个修改对询问的贡献是**互不影响**的，修改是**互相独立**的，且题目还需**允许离线**。若处理k个修改与询问的时间复杂度为$O(f(k))$，那么由主定理易得总时间复杂度为$O(f(n)\log n)$。

1.3 有撤销操作的按时间分治问题

由于对时间分治要求操作是互相独立的，因此如果有形如“撤销某次操作”这样的操作，就不能应用按时间分治的方法。遇到这种情况时，需要采用一些方法来将原问题转化为没有删除操作的新问题。一个优秀的做法是进行一次分治后将时间倒过来，那么删除操作也就变为了原来的加入操作。关于这种**“时光倒流”**的方法可以参考许昊然2013年国家集训队论文。

本文将提出另外一种通用方法，可以解决有撤销操作、修改对询问的贡献互不影响，且允许离线的问题。这也是本文的主题。下面由一道例题来具体阐述这种方法。

【例一】动态半平面交问题[1]

给定一个平面，要求支持以下几种操作：

1.插入一个半平面

2.删除一个还在当前平面的半平面

3.询问某个点是否在当前半平面的交集内

如果没有第二个的操作，那么这就是一道很好的按时间分治练习题。

通过观察上面两节中线段树的定义与按时间分治的方法，可以发现它们有着一些联系：**每次都是从区间$[l,r]$变为[l,mid]和[mid+1,r]进行递归。**

先考虑没有删除操作的子问题。在对时间分治中，每次都处理在左区间中的修改，对在右区间中的询问的贡献，这在线段树中就相当于处理在左孩子中修改对在右孩子中询问的贡献。在分治的最后，剩下的区间是大小为1的，而这也就是把对应的操作插入到了线段树的叶子中。

我们惊讶地发现，**按时间分治等价于线段树**，整个过程就相当于建立了以时间为序的线段树！当然分治与线段树还是有区别的：一个是先处理大区间再处理小区间，一个是先处理小区间，再处理大区间。这时就有一个奇妙的想法：

[1]题目来源：经典问题

如果按线段树的处理顺序来解决分治问题，会不会有一些强大或者方便的方法呢?

具体的做法是，事先建立一棵**大小为操作个数**的线段树，依次把每一个操作插入到线段树中对应的叶子，这个叶子的位置即为这个操作对应的时间。考虑线段树中的一个非叶子节点x，把它左儿子中所有半平面拿出来，求一个上凸壳与一个下凸壳；对于它右子树中的询问点(X,Y)，依次判断每一个是否在上凸壳或者下凸壳中。判断的方法是二分找到在$x=X$上最高（或者最低）的半平面，与Y比较即可。当这个点有一次不在某个凸壳中，就把它标记为永久不在半平面的交集中。这个做法是$O(m\log^2 m)$的，这里m是操作总数。如果用归并将半平面的斜率以及询问点的x坐标排序，可以做到$O(m\log m)$。

问题是，上述做法似乎与直接分治相差无几。换个角度思考，考虑每一个半平面，它对哪些询问点有贡献呢? 当然是他之后的那些嘛。既然知道这一点，何不在线段树中**直接对这个区间打上一个标记**，表示这个区间中的点需要被这个半平面包含。

现在的算法流程变成了：若某个半平面出现的时间为T，那么在线段树中$[T,m]$对应的那些区间标记上这个操作。只要对线段树中的每一个节点，将标记它的那些半平面与它所对应的区间的询问点做一次半平面交与询问即可。

令人欣喜，这样的算法是可以推广到有第二个操作的问题的——**一个半平面依然对应于一段连续的时间**，所以剩下的工作就与上述算法大同小异了。

接下去考虑复杂度。不考虑排序，求一次半平面交，以及判断一些点是否在当前半平面交内是线性的。询问点的排序可以利用归并完成。半平面可以在插入线段树之前事先排序，**先插的半平面一定在后插的半平面之前**，因此每一个节点处理的时候那些半平面都是已经排序好了的。时空复杂度都是$O(m\log m)$。

上述做法的空间复杂度是比经典做法要劣的，怎样才能优化它呢? 问题的关键在于求出线段覆盖的节点后就**直接对这个节点代表的半平面做半平面交**。显然，可以直接把所有半平面一起插入线段树，按线段树中插入区间的方式递归这些半平面；同时，必须把询问点与半平面同时插入，否则空间得不到优化。但是直接这样做，空间复杂度还是不变的，因为若所有半平面都覆盖了某一个叶子，那么从这个叶子到根的路径都要存一遍所有半平面。虽然这种情况是不可能的，但是**依然可以构造出类似的较差的情况**。

似乎我们陷入了困境? 别忘了我们依然是在线段树上讨论。每一个半平面

插入的时候**只会分叉一次**，这同时也说明了线段树的每一层最多只会有4个同样的半平面。因此只要一层一层地执行算法，就能保证空间复杂度了。直观上理解，这就是由bfs取代了上段中的dfs。至此，我们得到了一个空间$O(m)$，时间$O(m\log m)$ 的优秀做法。

前两段的叙述表明，这里所述的线段树与分治的**本质是相同的，只是表现形式不同罢了**。分治每次只考虑了线段树中的一条路径，线段树即为整个分治过程。依靠线段树这个更加具体、直观的模型，可以更清晰地处理问题，更方便地思考出优化问题的方法。但分治依然具有其好写、快捷的优点。

【例二】作业[2]

给定一列n个数，每次询问在第l个数字到第r个数字中，权值在a到b中的数字个数，以及在这些数字中，不同的数字的个数，这样的询问有Q个。

$n \le 100000, Q \le 1000000$。

考虑到不同的数字与权值关系较大，因此对权值建立线段树，在权值不同的地方数字是不会相等的。权值线段树中的每一个点，存下的是权值在它所代表的范围内的那些数字的位置。每一个询问对应的权值是线段树中的一些节点。

在线段树的每一个节点中，问题就变为求$[l,r]$中的数字个数，以及求$[l,r]$中的不同数字个数。这两个问题都可以用树状数组做到$O((n+Q)\log n)$。

这种做法的空间复杂度是$O(n+Q)$，时间复杂度是$O((n+Q)\log^2 n)$。

1.4 一种通用的转化方法

由于在现今信息学竞赛中，有像“撤销某次添加的操作”这种操作的题目是比较少的，因此上文中的方法往往得不到很广泛的应用。下面给出一种方法，能将形如“把$[l,r]$当中的所有元素全部变为某一个相同的元素”这种操作，转化为添加与撤销操作。

具体做法如下：将修改操作依次插入到一棵按**位置**建的线段树中，并对$[l,r]$对应的那些节点添上标记。若在插入的过程中遇到了一个已经有标记的

[2]题目来源：Ahoi2013

节点，就把这个节点上的标记分别记到左右两个孩子上，并将这个节点上的标记清除。当一次修改操作在线段树上找到对应的节点之后，将那些节点的子树中的**所有标记**都取出来（并清除这些标记）。若当前修改操作的插入时间为i，某个取出的标记的插入时间为j，且这个标记在线段树中的位置为$[l,r]$，那么就在转化后的问题中增如下两个操作：

1.**在时间j将位置$[l,r]$上的元素全部改为某一个元素。**

2.**在时间i撤销1这个操作。**

实现时只要记下线段树**某个节点对应的子树中有无标记**。如果没有标记，则不必继续在这棵子树中寻找；否则，继续寻找那些标记。

简单审视一下这种做法的正确性与复杂度。

可以证明，在区间覆盖操作中，任意一个时刻，序列中的第i个元素即为线段树中第i个叶子到根的路径上的最后那个标记所代表的元素。而一个已标记节点所代表的子树中，**其他的标记是没有用的**，因此清除标记并不影响区间覆盖。每插入一个修改操作，都相当于把它对应的那个区间中的所有元素取出，将那些元素修改为这个元素（只不过把一些连续且相同的元素并起来了）。

接下来考虑复杂度。每次插入后，每一个有标记的节点的子树中是**没有其他标记**的。因此每插入到一个节点中时标记只会下传一次（而不会由于下传到的位置有标记而继续下传）。下传一次之后标记个数会增加$O(1)$个，标记总深度会增加$O(\log n)$。如果用m代表修改总数，n代表序列长度，总的插入复杂度是$O(m\log^2 n)$ 的，这包括了插入以及下传的总时间。清除复杂度是不会超过插入复杂度的，因此总的复杂度是$O(m\log^2 n)$ 的，新问题中的操作数不会超过总标记数$O(m\log n)$。

但是这样的复杂度显然不能满足我们的要求。尝试写一下代码后可以发现，虽然构造的时间是$O(m\log^2 n)$的，但是对于$n=m=200000$的数据，还是能在200ms以内出解的；输出操作次数之后，发现与$O(m\log n)$更为接近。

换个角度思考，在原序列中，每次新增一个区间修改操作，都会把一些完整的相同元素段覆盖，并把两端那两个相同元素段截成两半，且覆盖掉其中一半。**每次最多会增加两个新的相同元素段**，因此最后不同的元素段是$O(m)$的。如：将6覆盖到1112334445555的第3个位置到倒数第二个位置，会把“2”，“33”，“444”完全覆盖，“111”会分成“1”与“11”，“5555”会分裂成“55”与“55”。

回归线段树，每次清除的标记有许多都是对应于区间上连续的一段的，**只**

需要将这些标记并到一起，新问题中的总操作数就是$O(m)$的了；同时，查找出所有可以并到一起的标记，实际上是$O(\log n)$的，因为这相当于找到一段区间在线段树中对应的节点。所以构造复杂度为$O(m \log n)$。

【例三】一个简单的区间问题[3]

给定一个长度为n的序列，初始时每一个元素的值都是给定的。要求支持以下几种操作：

1.将序列中第l个到第r个元素全部修改为x。

2.询问序列中第l个到第r个元素中有多少元素的值小于等于x。

一共有m个操作。

为了简化叙述，下列表述中皆认为$m = O(n)$。

先利用上述方法转化，新问题中第一个操作变为在序列的第l个位置到第r个位置中，每一个位置增加一个元素x以及撤销某次增加操作。按照上一节中的方法，可以将问题变为只有区间增加某一个元素的操作，且所有增加操作在询问操作之前的子问题。这个子问题只要按权值大小排序以后，利用线段树维护区间加以及区间询问权值和就可以了。

考虑复杂度。修改操作起初是$O(n)$的，经过第一步转化后有$O(n)$个，再添加到时间线段树之后就有$O(n \log n)$个。这些操作每一个还要在线段树中修改，因此总的时间复杂度是$O(n \log^2 n)$的。考虑询问的部分，在时间线段树上每一个询问都要处理$O(\log n)$次，每一次要询问区间和，所以这部分的时间复杂度是$O(n \log^2 n)$的。总的时间复杂度为$O(n \log^2 n)$。

由于这题的特殊性，在第一步转化之后的第二个操作可以看作一个权值为-1的添加的操作，之后可以采取普通的按时间分治写法来减小常数，时间复杂度不变。

【例四】Robot[4]

有n个机器人，m个插座排成一排，开始时每一个插座都是空的。

每一个时刻，都会有一个机器人登上舞台。机器人i会选择插着插座的某个机器人j，在第k个插座处吸收它的能量（需要保证此时j号机器人插着k号插

[3]题目来源：经典问题

[4]题目来源：原创

头）。如果j机器人的能量为f_j，那么i号机器人的能量为$f_j+(i-j)^2+(i-k)^2+(j-k)^2$，吸收完之后$j$机器人的能量并不会消失。如果没有任何一个能吸收的机器人，i机器人的能量为0。

获得自己的能量之后，i机器人会把自己的插头插到每一个$[l,r]$的插座上（此时插着的插头会被全部拔掉）。

i机器人在$i+M$时刻会把自己的所有插头拔掉并离开。这个M对所有机器人都是相同的。

问每一个机器人能获得的最大能量是多少。

先不考虑能在$i+M$时刻拔掉插头。即使不能预先确定每一个f_i的值，每一个机器人的有效区间是确定的。按照上文中的方法处理过后，线段树中的每一个节点只有添加j号机器人插着一段插头的操作。按照线段树的**中序遍历处理**所有操作，便能得到所有f值了。

由于$f_i = max(f_j+(i-j)^2+(i-k)^2+(j-k)^2)$，在$i, j$确定时，$(i-k)^2+(j-k)^2$必然在端点处取到最大值，因此只需把每一个区间的两个端点取出来，中间那段是无用的。$f_j+(i-j)^2+(i-k)^2+(j-k)^2 = f_j+(j-k)^2+j^2+k^2-2(j+k)i+2i^2$。这个转移方程是可以**利用斜率优化**$O(n)$求解的。此时的做法时间复杂度是$O(n\log n+n\log m)$的。

拔掉插头的操作，这只需要找出p在$i+M$时刻，i号机器人的标记**当前存在了线段树的哪些节点当中**，将它们全部取出来即可，这与普通的撤销操作相同，只需要在时间线段树中打标记。时间复杂度依然是$O(n\log n+n\log m)$。

EXT

题目中其他所有条件都不变，只将最大能量改为最小能量。

改为最小能量后，只将每一个区间的两个端点取出来的做法是不对的，$f_j+(i-j)^2+(i-k)^2+(j-k)^2$的最小值还会在$k=\frac{i+j}{2}$时取到。这时必须有$l\le\frac{i+j}{2}\le r$，也就是$2l-j\le i\le 2r-j$。这可以在原来的基础上，**再进行一次类似的分治**。当然，由于k的定义域是整数范围的，还需要处理好边界情况，但这并不影响复杂度。时间复杂度为$O(n\log^2 n)$。

2 有关树形结构的分治

2.1 序列问题的可持久化

可持久化是各种数据结构的经典应用。这里只考虑“把当前全局变回到某一次操作之后的样子”这样的可持久化。

在每一个时间T，都会有若干次不同的修改操作将它变为新的分支T'，新的这些分支T'会有自己新的后代，但与其他分支中的状态无关。回到某次操作也就是回到某个节点罢了。因此最后修改操作形成的关系是一个树形结构。

每一个修改操作会影响它子树中的所有节点，在dfs一棵树时，这些子树中的节点被访问到的时间是连续的一段。因此，可以在刚进入子树x时，添加上x所代表的操作，离开子树x时再撤销x这个操作，撤销操作的撤销操作亦即一个添加操作。将这棵树中的节点按照dfs序排列之后，就将这个问题转化为了序列问题，且操作数量级不变。

2.2 虚树

在序列问题的分治中，每一个子问题的时间复杂度不能与总序列的长度有关，只能与这个子问题中包含的元素个数有关。这只需将这些元素按照在序列中的位置排序，就可以做到与序列总长无关了。

但由于树的结构比较复杂，不能简单地通过排序来摸清子问题中节点之间的关系，因此需要利用虚树来优化复杂度。

2.2.1 定义

设原树为一棵形态固定[5]的树T，它有n 个节点。它的虚树是它的一棵联通子树[6]，或者是它的某棵联通子树，将一些没有分叉的连续的边缩成一条边之后形成的树。由于这些新形成的边在T中是不存在的，因此形象地称之为虚树。

[5]形态不固定的树也能构造虚树，但与本文没有较大关系，这里不再介绍。感兴趣的读者可以查阅参考文献[3]。

[6]这里所说的子树概念类似与子图，而不是有根树的子树。

2.2.2 构造方法

在实际问题中，构造虚树的条件通常是求出一棵较简的虚树，使其包含$x_1, x_2, \ldots, x_m$这些节点，称这些节点为关键点。

首先，如果一个点是关键点，那么这个点肯定要是最后的虚树中的某个节点。将这些点之间路径上的所有点选出后形成的子树，就是一个合法的虚树（即使很不优）。考虑这些点中的某个非关键点，去掉这个点后整颗树会被分为许多子树，这些子树中至少有两棵包含关键点（否则这个点就不会被选）。如果这些子树中有两棵包含关键点，那么这个点连向这两个子树的那两条边就能缩为同一条边；否则，将这个点也作为最后虚树中的节点。

这样的构造还是有点麻烦的，可以降低一点要求。将整棵树视为有根树。若一个点满足上述的作为虚树节点的条件，那么被它分成的那些子树中，**至少有两个是它的儿子子树**；而它就对应于在那两棵子树中的**关键点的**Lca 。因此只需将关键点两两之间的Lca作为虚树中的节点即可。

树上两个点之间的Lca，即这棵树的Euler tour technique中，那两个点之间深度最小的节点。如果有三个点x, y, z，它们在Euler tour technique 中对应的先后次序为x, y, z，由于区间最小值具有可合并性，那么有$h[Lca(x, z)] = min(h[Lca(x, y)], h[Lca(y, z)])$，而在这一段中深度最小的节点是唯一的[7]，**就有**$Lca(x, z) = Lca(x, y)$**或**$Lca(x, z) = Lca(y, z)$ 。只要先将$x_1, x_2, \ldots, x_m$按照dfs的顺序排序后，再把$Lca(x_i, x_{i+1})(1 \le i < m)$与$x_i(1 \le i \le m)$一起作为虚树中的节点即可。

有了虚树中的节点后，按照dfs序的顺序模拟这棵虚树的dfs，就能得到虚树节点之间的连边了。具体做法是，先把虚树中的节点按照原树dfs序中的顺序排序，按照这个顺序插入这些节点。维护虚树中的根到当前节点的那一条路径。当前插入点x时，依次检查这条路径上最后的那个点y是不是x的祖先[8]。**如果不是，就将**y**踢出当前的路径，继续检查；否则在**x**与**y**之间连一条边**，并将x加入路径。

上面这个构造虚树的方法时间复杂度是$O(m \log n)$的，构造出来的虚树大小是$O(m)$的。

[7]假设有两个深度相同且最小的点，那么它们的Lca也一定在这一段中，且深度更小，矛盾。

[8]可以检查x是否在整棵树中y的子树对应的那一段区间中，这是$O(1)$的。

2.2.3 操作方法

问题中的操作往往不是针对虚树中的节点的，而是针对整颗树中的节点的。在虚树中，可以将不是虚树节点的点权视为无用的。那么对一棵虚树进行某个操作，即为对那些既在这棵虚树中，又与这个操作有关的那些节点进行操作。

一般树的操作有链与子树两种，对子树的操作可以直接对dfs序维护，下面只考虑在全局中对一条链(X,Y)进行操作。这样的链一定能**拆成两条链**(X,Z)，(Y,Z)，Z是X和Y的Lca。找到X的祖先中（包括X），深度最大且为虚树中的点的节点a；找到Y的祖先中（包括Y）深度最大且为虚树中的点的节点b。若Z为虚树中的点，那么(X,Y)对应的虚树中的链为(a,b)；否则，对应的链为a到Z以下（不包括Z）与b到Z以下（不包括Z）的那两条链。可以证明，这两条链中最多只有一条是有用的，因为如果a、b的深度都大于Z，那么Z必然是虚树中的点。知道a后，可以利用倍增，找到a的祖先中，深度大于Z且深度最小的那个点c，那么(a,c)就是所要求的链。

为了找到a节点与b节点，可以事先**按深度从小到大的顺序**将虚树中的每一个节点c对应子树中的所有节点的权值改为c，这样X的权值即为a，Y的权值即为b。这些修改与询问可以利用线段树维护dfs序解决。预处理的复杂度为$O(m\log n)$，对每一个操作找到对应的虚树节点的复杂度为$O(\log n)$。

【例五】Jc的宿舍[9]

一棵n个节点的树，每一个节点有一个权值。

有Q个询问，每次询问是将一条链(x,y)上的权值存到数组w中，并从小到大排序。

若这条链上共有m个节点，那么输出$\sum_{1\le k\le m} kw_k$。

$1\le n,Q\le 50000$，强制在线。

按权值从小到大分块，建立每一个块中的虚树，并预处理出每一个块之中**两两节点之间的答案**。这只要从每一个节点出发，在虚树中dfs，并维护路径的平衡树即可。这里两两之间的答案具体指的是，**只考虑此虚树中的节点**，这两个点之间路径的答案。

[9]Author：吉如一

对于每一个询问，依次对每一个块中的虚树询问链上答案，链上节点个数以及链上权值和。链上答案在预处理时已经得到，链上节点个数与链上权值和可以预处理出节点到根路径上的和，减一下就是路径的值。

假设共有X个块，每一个块的答案，权值和分别为ans_i，w_i，前i个块的节点个数和为S_i。那么最后的答案为

$$\sum_{1\le k\le X} ans_k + w_k S_{k-1}$$

这个式子的正确性是显然的，因为某个块k中的所有节点都又有S_{k-1}个节点放在它之前。

若每个块的大小为T，那么预处理的复杂度为$O(n^2 \log n/T)$，处理每一个询问的复杂度为$O(T \log n)$。取$T = \sqrt{n}$，可以得到复杂度为$O((n+Q)\sqrt{n}\log n)$的做法。

事实上，虚树中两两之间的点数，点权和，以及每个点在虚树中对应的点，都是可以预处理的。这样每次询问复杂度变为$O(n/T)$，取$T = \sqrt{n/\log n}$，总复杂度变为$O((n+Q)\sqrt{n\log n})$。

【例六】CHANGE[10]

给定一棵大小为n的有根树，树上每一个节点有两个属性a, b，初始时所有节点的属性都相同。共有Q个操作。

操作有以下几种：

1.将一棵子树的a值修改为某个值。

2.将一棵子树的b值修改为某个值。

3.询问一条链(x, y)上，$a \in [X, Y]$的节点中，b的最大值。

由于子树修改相当于对dfs序中的某一段进行修改，因此，可以利用1.4中的方法，将其变为增加与撤销操作。但如果只这样变化的话，会给整个算法造成麻烦。考虑到每次都是修改子树，每一次操作完之后，那些转化而成的且标记相同的区间[11] **必然可以并成一个联通子图**。将这些联通子图插入到时间线段树中。

[10]题目来源：原创。

[11]这边标记的定义见1.4。

对于时间线段树中的每一个节点，建立一棵以a值为序的权值线段树，权值线段树中的每一个节点有一些修改一个联通子图b值的操作。现在考虑对权值线段树中的每一个节点怎么做。

对于每一个修改联通子图的操作，都将这个联通子图中深度最小的点，作为关键点[12]，建立包含这些关键点的虚树；并且利用线段树将这个联通子图中点的b值修改为给定值，这只要将利用1.4构造出来那些区间修改一下就可以了。

对于每一个询问，肯定能**拆成两条往祖先方向的链**。这样的有向链要么停留在某个联通子图中，要么穿过了这个联通子图中深度最小的点，到达下一个联通子图。因此，只要求出这条链通过的关键点中b值的最大值，以及这条链最后到达的那个点的b值，两者中较大的那个就是这个权值线段树中这个节点对这个询问的贡献。

简单复述一下算法：先将子树修改操作转化为联通子图增加一对(a,b)的操作，以及撤销某次增加操作的操作。将每个增加操作插入到对应的时间线段树当中。对时间线段树中的每一个节点，将包含它的增加操作取出来，按照a值建立权值线段树，并在权值线段树的每一个节点构造一棵虚树。对于这个时间线段树节点包含的询问，在权值线段树中找到$[X,Y]$对应的那些节点，并在那些节点的虚树中进行询问。

时间线段树将节点数乘上$\log Q$，权值线段树将节点数乘上$\log n$，权值线段树中的操作都是$O(\log n)$的，所以该做法的复杂度为$O(Q\log^2 n\log Q)$，这里忽略了一些在原树中预处理的复杂度。

3 总结

这种结合线段树与分治的方法对一类区间修改问题有着良好的功效。这种方法的优点有代码简单，思路清晰，高效快捷等，在实际问题中有较广的应用。结合虚树后，可以用来解决一些较难的问题。

4 特别感谢

- 感谢父母、学校对我的培养

[12]这样的点必然是唯一的。

- 感谢杜瑜皓，黄嘉泰，周子凯等同学对我的帮助
- 感谢CCF给了我一个展示自己的机会

参考文献

[1] 陈丹琦，《从〈Cash〉谈一类分治算法的应用》

[2] 许昊然，《浅谈数据结构题的几个非经典解法》

[3] 陈立杰，《JustForFun EXT 解题报告》

-- 空 --

根号算法——不只是分块

南京外国语学校　王悦同

摘 要

根号算法逐渐普及，OI题目的复杂度也变得越来越“多样”；不再仅仅是简单的O(N),O(N^2),O(NlogN)这种复杂度了。很多题目复杂度里出现了根号，其中数据结构题中的根号算法最为常见，例如分块维护或者“莫队算法”——去年已经有同学详细介绍过了。但是，根号算法绝不仅限于这些地方，它们在很多其它的算法中都会出现，伴随着不同的新奇思想。而对这些思想深入研究，更是能挖掘出比根号更优的复杂度的应用。本文介绍了几种根号算法出现在非数据结构题中的应用，并试图移植这种思想获得更优的算法。

1　算法组合

1.1　从一道水题说起

【例1】Codeforces #80 Div1 D

给出一个长度为N（⩽300000）的数列a[i]，再给出M（⩽300000）个询问，每个询问是形如(x,y)的形式，你需要输出a[x]+a[x+y]+a[x+2y]+...+a[x+ky]的和，其中x+(k+1)y>N。时限4s。

这题的询问和以往不太一样：大多数同学应该有这样的感觉，我们所学过的大多数数据结构都擅长处理“连续区间”的询问，而不擅长间隔的位置之间的询问。对于本题，线段树之类的传统数据结构难以胜任。但我们发现，假设所有询问的y都一样，那么我们就可以把原序列按下标mod y的值分成y组，询问在对应的组里就是连续的了——而且只需要部分和维护。

这样的问题就是，我们对于y=1要维护若干个数组，size总和为N；对

于y=2也要，y=3、4、一直到N都需要，难以接受。但我们也不难想到，如果y很大，那么为什么要预存呢？直接暴力不是很快么？每次暴力扫描所有位置是O(N/y)的。因此，取K= $\sqrt{N}$，我们对于1≤y≤K的情况预存部分和，对于y>K的情况暴力扫描，就可以保证算法的复杂度为O(N $\sqrt{N}$)了。

我们回顾一下这题。这题不能完全说不是“数据结构”，但它体现的是一种和数据结构题中的根号截然不同的思想。对于题目中的某两个约束，它们是互相制约的，并且这种制约是“乘积”关系——步长和项数，总有一个不能太大。而我们针对两个情况分别设计算法，然后将两种算法组合起来，就能获得一个完整的解决问题的算法。下面看另一道这种思路的简单应用：

【例2】Topcoder SRM589 Level-3 FlippingBits

给一个长度为N（N≤300）的01字符串，再给一个正整数M。每次操作，你可以将一个位置取反，或者将一个长度为M的倍数的前缀取反。问最少需要多少次操作，才能使字符串成为一个循环节为M的循环串。

循环节长度为M定义为，对于任意i，如果i+M这个位置存在，那么i和i+M上的字符应该相等。

本题也不是很难。我们考虑循环节的长度和循环节的个数——这两个乘积（大约）是原串的长度，也就是他们是互相制约的，总有一个不超过 $\sqrt{N}$。因此这启示我们分别针对两种情况设计“专杀”算法。首先考虑循环节长度不是很大的情况。这里指的是循环节长度不超过17（也就是300的平方根的近似值）。这个时候我们发现，因为答案要求每一个循环节都一样，而循环节又不长，所以直接枚举：2^{17}。枚举出答案以后，每一段必须和答案相等、或者和答案的反串相等。这里很简单，只需要一个动态规划即可，因为注意到记原串每一段是否整体反转，那么操作“将一个长度为M的倍数的前缀取反”的执行次数就是有多少段连续的0和1。

下一种情况就是循环节个数不是很多。这个时候，由于操作“将一个长度为M的倍数的前缀取反”只会在不超过 $\sqrt{N}$个位置进行，故仍然可以枚举。这个时候，我们已知每一段是否反转，要构造一个答案来“迎合”它们——这也很简单，对于答案串的每一位，如果所有循环节在这一位的0比较多，这一位就是0；1比较多就是1。因此整个题目就解决了，复杂度为O($N * 2^{\sqrt{N}}$)。

我们再来简单回顾这题，和上一题还是比较相似的。通过组合两个“朴素”的算法，得到一个相对高效的做法。更重要的是，这样的“组合算法”解题，几乎是无可替代的——有些题唯一的做法就是通过组合各种算法；或者只有这样才能降低复杂度。这样的做法能扩展到更加复杂的题目上去。

1.2 进一步应用

【例3】IOI2009 Regions

N个节点的树，有R种属性，每个点属于一种属性。有Q次询问，每次询问r1,r2，回答有多少对(e1,e2)满足e1属性是r1，e2属性是r2，e1是e2的祖先。

N,Q≤200000，R≤25000，时限5s

我初看这题感觉是很正统静态树上的询问（幸好题目要求在线，不然我应该会想的更偏）。不过我想了很久类似O($Nlog^2N$)之类的算法却毫无进展。后来我看了一眼题解，只看了一点点，立即恍然大悟——题解中告诉我，考虑针对询问的点的个数不同设计不同的算法。

不妨设询问中r1的属性的点有A个，r2属性的点有B个。如果A不大、而B很大呢？我们可以允许扫描A，但是不能扫描B。一种可行的做法如下：考虑dfs序，A中每个节点的子树在dfs序中都对应了一段区间[l,r]，而B中每个节点都对应了一个dfs序中的位置p。如果l≤p≤r，那么可以知道A是B的祖先，反之就不是。所以，对于每种属性的节点开一个Vector，将这种属性的所有点的dfs序标号按顺序存进来，那么我们枚举A中的每个点，只要在B对应的vector里二分查找即可。这个算法并不难实现，但是却可以在O(AlogB)的时间内完成一个询问。

那么如果A很大但B不大呢？从直觉来看，我们应该设计一个O(BlogA)的算法。这样的算法确实也是能设计出来的，尽管有点复杂。例如，大概可以这么做：模仿刚刚AlogB的做法，我们扫描B，然后看这个点覆盖了多少区间。因此可以先将该组A个区间全部预处理好，把数组中对应的一段+1，例如有一个区间[5,10]，就把数组5至10这一段都+1。这样我们可以直接看这个位置的数而确定这个点被多少区间覆盖了。A个区间端点可以离散化、求和也可以用差分等

方法进行，总之也能在O(BlogA)内完成。

但是我设计出O(BlogA)的做法后发现了更严重的问题。

第一个问题就是，如果询问的A和B都很大，那么无论如何也不可能优于O(A+B)了；A和B都最大可以是N，如果每个询问都这么大，不就直接超时没有优化的余地了么？

但实际上解决这个问题的方案却异常直观。想象一下，一个询问涉及的点数非常多，那么这样的询问不可能有太多；如果有两个询问一样我们必然可以直接查询以前的答案。所以关键就是，同一个询问不做两次。那么不同的询问呢？直接做！复杂度将在后面给出证明。

第二个问题就是，现在即便已经有O(AlogB)和O(BlogA)的做法了，但仍然有这种情况：（假设N和Q是一样的，因为它们是一个数量级），有$\sqrt{N}$种属性，每种有$\sqrt{N}$个，然后N（也就是Q）个询问，把两两属性之间都问了一遍。这个时候不难发现算法复杂度退化到了O(N$\sqrt{N}$logN)，对于N=200000来说一看就是不可接受的。

通过之前的分析，我们发现，A和B一个很大一个很小的情况是不难办的。但是近似大小的时候呢？我们应该有这样一种感觉，如果我们允许A和B都扫一遍，应该可以设法避免这个log的存在。实际上确实想下去就发现不难了——把A视为区间，B视为点，要求每个点被多少区间包含了，这个实际上可以吧“区间首段点”和点一起做归并，然后顺便维护一个栈就行了，因为A中的区间互不包含，有很多显然的单调性。

最后就是复杂度分析。我们设一个阈值C，对于A≥C的情况，使用O(BlogA)的做法，对于B≥C的情况使用O(AlogB)的做法，复杂度最坏O(N^2logN/C)；对于其它情况用O(A+B)的做法，复杂度是O(NC)。解出来C=$\sqrt{NlogN}$。这里把N和Q混在一起分析了，但他们是一个数量级，所以这么分析没啥问题。整个算法复杂度是O(N$\sqrt{NlogN}$)，可以接受。

刚刚这题就比以往复杂多了；即便是放在当时IOI赛场，也绝非容易。多种算法的取长补短以保证复杂度的思想，在刚刚那题中得到了很好的体现。

2　另类平衡

2.1　分类维护

“分类维护”是一种数据结构题中的另类思想。同样是根号算法，它和“分块维护”在思想上还是有很大差异的。我们通过一道例题来简要的说明一下这种做法的含义和应用。

【例4】　给出一个N个点M条边的无向图，N,M≤100000。一开始所有的点都是黑色。每次你可以进行两种操作：将一个点反色，或者查询某个点直接连边的点中有多少个黑点。时间限制1s。

本题确实不难，但体现了一种很好的思想。我们发现，在大多数情况下（随机情况下），直接暴力扫描一个点的出边是很快的。为什么？因为出边少——虽然理论上最坏是O(N)的，但是很难达到。如果一个点出边不多，那么我们就直接询问到的时候枚举即可。

如果出边很多咋办呢？我们应该有这样的反应——一个多，另一个就少；出边多，这样的点就少。我们不妨以$\sqrt{N}$为界来讨论（一般都可以以$\sqrt{N}$为界讨论，等到最后卡复杂度的时候再仔细估算阈值大小）。如果一个点度数≤$\sqrt{N}$，那么直接维护，复杂度是O(M$\sqrt{N}$)；如果度数大了，则这样的点最多$\sqrt{N}$个，我们可以直接每次修改后都暴力维护一遍这些点的答案——很简单，一旦修改，就扫描这$\sqrt{N}$个点，看看这次修改和它是否有关。这个也显然能在O($\sqrt{N}$)时间内维护。因此本题通过组合这两种情况的维护方法，就可以实现O(N$\sqrt{N}$)的复杂度。

2.2　轻松一刻

下面我们来看一个很简单有趣的题目放松一下。

【例5】烧桥计划（JSOI2013互测）

给一个长度为N的数列（N≤100000）$A_1,A_2...A_n$ (1000≤A_i≤2000)。你需要从中选出若干个数（个数不限，也可以不选），记为K 个，分别为$A_{p_1},A_{p_2}...A_{p_K}$（p数列严格递增）。这次选数的代价是$A_{p_1}$*1+$A_{p_2}$*2+...+$A_{p_K}$*K。

这些选出来的数把剩下的N-K个数分成了K+1 段（可能有空段）。记每一段中A数列的和为T_i,对于所有T_i>M的段，产生额外的T_i的代价；否则没有任何额

外代价。对所有的T_i，M都是一样的，输入的数≤$2*10^8$。求一种选数方案，使得总代价最小。时限7s。

我们不难设计出一个二维的状态f[i][j]，表示当前最后一个选择的位置是i，且这个位置是从左到右第j个选择的位置，此时j个位置和j段数生成的总代价，最少是多少。因此可以写出状态转移方程：记sum[i]表示前i个A数组的和，

F[i][j]=min(F[k][j-1]+(A_i)*j) for sum[i-1]-sum[k]≤M

F[i][j]=min(F[k][j-1]+(A_i)*j+sum[i-1]-sum[k]) for sum[i-1]-sum[k]>M

其中0≤k<i

对这个动态规划稍加分析，就可以发现利用单调队列可以将复杂度降到O(N^2)——剥离变量后直接维护，因为比较显然，所以这里不详细说明了。关键是原题是N=100000，这个做法仍然太慢。哪儿还能优化呢？

这题是我在JSOI2013的省队互测中出的一道“搞笑”题，现场没有人做出来；其实不是难题，而是没有反应过来“脑筋急转弯”。原题中有一个条件，1000≤A_i≤2000。上界正常，下界有什么用呢？注意到题目中选一个A_i的代价其实是越来越大的，所以我们不能选太多！

对于一个数都不选的情况，总代价≤2000*N（只有一段）

对于选了i个数的情况，总代价≥i*(i+1)/2*1000≥500*i*i

要使得选i个数时答案可能成为最优解，必须有500*i*i≤2000*N

即i≤$\sqrt{N}$*2

因此状态个数是O(N$\sqrt{N}$)的，即可通过本题了。

我把这题放在这儿也并非和其它题没有关联；毕竟，它也是一个O(N$\sqrt{N}$)的算法，根号的来源也同样奇特——来源于对题目中下界的分析。我当时出这题时受到topcoder某道题的启发：那题是说，给出K个点分成N层，相邻层之间有些点有边，第一层和最后一层也有一些边。求这个图的最大独立集。K≤100，10≤N≤100。这题中“最大独立集”明显是需要利用二分图的性质进行匹配，但如果N是奇数则不是二分图，怎么办？注意到题目中N不小于10，而点数不多于100，所以一定有一层点数不超过10。我们枚举这一层的所有情况，就可以“破环为链”了。这题和主题关系不大，但还是顺便介绍一下；毕竟如果上界和下界可以视为一个数量级（差小常数倍，例如原题差2倍），则经常可以通过上下界的分析得到和根号有关的性质，作为解题的突破口。我们再回到原主题：

2.3 内存优化

也许卡内存的情况在OI中并不常见。但既然时间可以有这样的优化，内存为何不可以呢？通过类比的扩展，内存也可以得到类似的优化，而且幅度并不小——所以其实也是一种很有用的思路。我们通过一道题目来看一下这种应用。

【例6】Codeforces #79 Div1 E

你现在有N（⩽20000）颗石子和M（⩽20000）颗糖果，准备把它们吃完。现在有N个参数A[i]和M个参数B[j]，每次你可以选择吃一个糖果或者一个石子，吃完之后，如果剩下来x颗石子和y颗糖果，你就可以得到(A[x]+B[y])mod P的奖励（P也是给定的），然后你继续吃。你要最大化M+N次得到的所有奖励之和。你需要输出方案：每次是吃糖果还是石子。时间限制15s，内存限制45MB。

如果不考虑内存限制的问题，这题还是很容易的。一个动态规划：F[i][j]表示吃了i个糖果、j个石子时的最优答案。F[i][j]=(A[N-i]+B[M-j])mod P+max(F[i-1][j],F[i][j-1])。这应该大家都会；至于记录方案，只要另一个数组G记录每个位置从哪儿转移来的就行了。

那么回来看内存限制呢？大家应该都能想到F数组可以滚动数组、而G可以压位。不过算了一下发现这样也需要五十多兆内存……所以考虑另辟蹊径。

直接存下这个表内存是不够的。但我们为啥要全部存下来呢？例如，我们如果记录了倒数第K行的答案，那么从第K+1行到第N行的空间就可以和前面“公用”，因为我把这段的答案得出来后直接输出，以后就规约为从第K行往前的答案的问题了。受此启发，我们考虑适当存一些行的答案，以尽可能公用空间。

想到这里就不难想到根号了；凭直觉来也应该是每$\sqrt{N}$行存一次答案（也就是F数组，这里不滚动了）。这样我们只需要O(N$\sqrt{N}$)的空间存储这些信息，就可以分段恢复答案了！我们要f[M][N]的答案，但我们既然有了M-$\sqrt{N}$这一行的答案，就可以从这一行开始dp，然后得出这一段的“最优路径”；然后同样的空间还可以恢复上面$\sqrt{N}$行的答案，再往上，以此类推。这是一种另类的平衡。虽然理解起来不困难，但思维还是很新奇的，是根号思想的一种扩展应用。整个算法复杂度仍然是O(N^2)，但空间是O(N$\sqrt{N}$)了。

3　自然出现

也有一些时候，我们并不是刻意去让根号出现在算法里，但是它却会无意中就出现了。举个例子：N div i（整除）当i取遍1至N时，只有O($\sqrt{N}$)种不同的取值。这个并不难证明：当i≤$\sqrt{N}$时，最多也就$\sqrt{N}$个不同的答案；而当i>$\sqrt{N}$时，答案落在[1, $\sqrt{N}$]的区间内，也最多$\sqrt{N}$个。这个性质有一些应用；下一个例子介绍了一下这一类“自然出现”的$\sqrt{N}$如何应用起来。

【例7】POI2007 Queries

有N（≤50000）组询问，对于给定的整数a,b和d，有多少正整数对x,y，满足x≤a，y≤b，并且gcd(x,y)=d。

本题是一个数学类问题。首先分析一下题目，gcd(x,y)=d这个可以直接转化为gcd(x,y)=1，并且把a和b的范围同时缩小为a/d和b/d。也就是说，我们要求出有多少互质的数对，一个不超过a，一个不超过b。这和NOI2010的能量采集十分像，可以直接通过容斥原理解决。具体做法就是，设F(x)表示有多少对数，一个不超过a，一个不超过b，并且它们的gcd 是x的倍数。这很简单——F(x)=(a div x)*(b div x)。接下来用容斥即可，答案=F(1)-F(2)-F(3)-F(5)+F(6)-F(7)……，其中F(x)加到答案中前面有一个系数p(x)，如果x中某种质因子含有超过一个，则p(x)=0，否则p(x)=$(-1)^G$，其中G表示x中不同质因子的个数。直接这么做是每个询问O(N)的，相比O(N^2)的暴力确实好得多，但仍然复杂度不满足要求。

我们观察式子，发现我们要做的其实就是，尽快求出∑F(i)*p(i)。p(i)规律并不明显，但F(i)呢？之前我们说过，N div i的取值个数是O($\sqrt{N}$)级别的，那么F(x)和它有直接的关联——a div x和b div x都只有O($\sqrt{N}$)级别的不同取值，所以F(x)取值也是同一个数量级的!

对于连续一段相邻的F(x)，如果他们相等，就可以一并计算。这里仅仅需要维护一个p(x)数组的部分和就行了。通过维护p(x)的部分和和将相同的F(x)一起计算，我们成功的做到了在O($\sqrt{N}$)的时间内回答一次询问，因此可以通过本题。

本题中所说的p(x)这个函数还有一个名字叫“Mobius函数”，去年集训队王迪同学的论文里讲到过。这里并没有深入挖掘它在数学上的更多性质和广泛应用，只是以此为例，指出了一类利用“自然”存在的性质对算法进行优化的思路。

参考文献

[1] 《21-st International Olympiad In Informatics TASKS AND SOLUTIONS》from www.ioi2009.org

[2] 《POI.XIV.sol》2008国家集训队作业，成都七中-周梦宇

[3] 《浅谈容斥原理》2013国家集训队论文，成都七中-王迪

-- 空 --

浅谈动态树的相关问题及简单拓展

长沙市雅礼中学 黄志翱

摘 要

动态树是在OI中很常见的一种问题。解决各种动态树问题的算法：树链剖分，树分治和link/cut tree已被广泛了解和学习。同时，用树的dfs序或者ETT[1]维护子树信息的方法也已经众所周知。本文将对相关问题进行总结，并提出一些原创性的解法。在最后，本文将介绍LCT的两个拓展：top tree和link/cut cactus，用来解决相关问题。

1 动态树问题

动态树问题(Dynamic Trees)是指在树上动态维护相关信息的问题。不过有时候在提到动态树时，会专指LCT。

在一般的动态树问题中，会出现如下操作：

1. 添加一条边或者删除一条边，维护树的形态。
2. 将树上的一条路径上的权值进行一定操作，维护每个点的权值。
3. 查询树上一条路径上的信息。
4. 给树的一棵子树的权值进行一定操作，维护树上每个节点的权值。
5. 查询子树信息。

[1]欧拉遍历树

2 古老的动态树问题

算法竞赛中出现过各种各样的和树有关的题目。解决这些问题的方法通常也是使用各种树型结构——比如说平衡树。

在一些比较古老的题目中，树的形态是相对固定的，这使得解题者不用将注意力花费在维护树的形态，更多地是如何将树上的信息按照某种形式组织，从而实现维护。

2.1 树链剖分

树链剖分的核心是将树上的边按照一定规则划分为轻边和重边。将重边形成的链用相关数据结构进行维护。可以证明一个点到根的路径至多被分成$O(\log n)$条重边和$O(\log n)$条轻边。于是就可以在$O(\log n *$单次询问复杂度)完成所有操作。

2.2 点分治

点分治是通过寻找树的重心将树进行分开，从而做到了将树的大小在每次分治后能够减少一半。

通常点分治解决的，是和树上的路径，或者点对有关的问题。在朴素的做法中，经常能够通过枚举路径两个端点之间的LCA，从而很方便地统计相关路径信息。而在点分治的做法中，只需要枚举在分治时选择的重心，可达到同样的效果。

3 Link/cut tree

3.1 什么是LCT

可以参考4中对LCT的介绍。

具体来讲，动态树就像是用Splay维护所有实边的树链剖分。利用动态树的access操作，可以取出一个点到根的链。通过对Splay 的翻转操作，可以实现换根。利用两次access可以取出两个点的LCA。

在均摊分析下，单次操作是均摊$O(\log n)$。于是可以解决相当多的问题。

3.2 边的信息与点的信息

在有些题目需要对边信息和点信息进行维护，解决这个问题的方法通常有两种：

第一种是将一条边(u,v)，新建一个点u'，将这条边变成连接$(u,u'),(u',v)$的两条边，将相应的边的信息存到点上。

第二种方法是同时记录一条链与每个点关联的两条边。在换根的时候进行相应的维护。

3.3 如何使用LCT

考虑一个最简单的问题：

1. 每次可以添加一条边，或者删除一条边。
2. 给一条链上每个点加上一个数。
3. 询问一条链的权值和。

考虑怎么用动态树实现：

1. 加边：将一个点作为它所在的树的根，并设置一条它连往其父亲的虚边。
2. 删边：对边的一个端点进行access操作，将这条边变成虚边，直接删除。
3. 给链加上一个数：将链上一个端点设为根，access另一个点，使得整条链在一个Splay中。利用Splay将整棵树加上一个权值。
4. 查询链上的权值：同上得到链所对应的Splay，在平衡树中查询整棵树的权值和即可。

可以发现，只要是能够用平衡树维护的序列问题，很容易通过LCT推广到树上。比如说将相应的区间加减设为区间赋值。查询的也不一定是链上的和，还可以是链上的最大值或者许多其它信息。

3.4 LCT的简单应用

3.4.1 明显的动态树问题

很多问题直接使用动态树即可。比如说BZOJ 3091 城市旅行。

此题要求维护树，支持删边加边，链上的权值加减，询问最大子段和。

显然可以使用LCT维护出树的形态，接下来只需要考虑区间赋值和查询子段和的操作了。

最大子段和可以使用维护最大前缀和以及最大后缀和的方法，利用相关信息合并解决。做法很经典，就不细说了。

3.4.2 隐藏的树结构

有很多问题并没有显式地将树告诉解题者，但经过分析之后可以发现明显的树结构，故而可以用动态树进行解决。比如IOI 2011 elephant 一题。只要注意到了每个点只有一个唯一的父亲等类似的树的特性，这些问题就迎刃而解了。

3.5 LCT在一些简单图论问题中的应用

树是描述图的重要工具，具体讨论可以参考3。动态树的灵活使得可以用其来解决一些图论问题。

3.5.1 只有加边的最小生成树

最小生成树是经典问题。而动态维护最小生成树则是一个相当困难的问题。

如果使用分治算法，可以做到$O(m \log m)$的复杂度。具体参考5的解题报告。但是这个做法是基于离线的，强制在线的话，只有一些复杂度不是很优美，相当复杂的算法。

但是如果只存在加边操作，且强制在线维护的话，就可以用动态树维护了。最小生成树的环切性质保证了只要每次在加入一条边之后，删除环上权值最大的边，则仍然是最小生成树。而查询路径最大值是动态树基本操作，问题解决。

3.5.2 最小极差路径

给定一个带权无向图，求从S到T的一条路径，使得路径上的最大权值减最小权值最小。

这个问题可以用动态树解决：将边排序，从小到大枚举边的最大值，实际上要求的是小于当前最大值的边中最小权值最大的一条路径。最小权值最大其实是最小瓶颈生成树的问题的一个子集，而最小生成树一定是最小瓶颈生成树，故而可以套用之前的做法解决。

3.5.3 动态维护图是否是二分图

给定一张图，可以加边或删边。每次求它是不是二分图。可以离线。

此题类似离线动态维护图的连通性，可以用分治做，具体不再赘述。

离线处理出每条边的删除时间，维护关于删除时间的最大生成树，也就是加一条边之后若形成环，则删掉环上的最早删除的边。删除边时还要做一件事：询问这条边与树边是否形成奇环，如果是则加入一个集合。删边时如果是树边直接删，否则若在集合中则从集合中删除。如果某个时刻集合为空则是二分图，否则不是二分图。正确性不是很显然，但可以证明。

而所有操作直接用动态树维护即可。

4 dfs序和ETT

可以发现，之前提到的动态树问题几乎都是和链上信息有关的。因为LCT本身就是用链维护树的结构。不过有很多问题是和树上的子树信息有关，单纯地用动态树解决会有一定困难。

这时候就有了另一种强大的工具，树的dfs序列——在dfs中，只有遍历完了一个点的子树才会继续访问其它点。故而，将点按照dfs 中的顺序写出，一个点的子树总是一个连续的区间。这样就可以使用各种数据结构进行维护了。

4.1 简单的子树维护问题

很多问题都是具有很明显的子树查询操作，可以用dfs序直接解决。

4.1.1 树

题目来源：IOI2012中国国家队训练罗雨屏

题目大意是给定一棵树，要求支持：

1. 换根
2. 修改点权
3. 查询子树最小值

经过简单分析后可以发现，这题从LCT入手并不明智。虽然有换根这个很灵活的操作，但树的形态并不会改变。

不妨使用树的dfs序。修改点权使用线段树维护区间最小值。

那么换根和查询怎么办呢？很容易发现，无论根在哪儿，查询的，要么是这个点在根为1时的子树，或者这棵子树以外的其它点。对于前者，直接查询区间最小值，否则，查询这个区间的补集的最小值，问题解决。

4.2 平衡树对dfs序的维护

得到一棵树的dfs序之前必须对整棵树进行dfs，如果树的形态会改变会怎么样呢？

可以发现，如果固定根，在加入一个点或者删除一个点后，在dfs序中相当于插入和删除一个数，只要能够找到这个数的位置就可以用平衡树维护，而这是非常容易的事情。

同理，如果是将一棵子树加入和删除，对应在dfs序中的改变只不过是相应的区间插入和删除，使用Splay或者没有旋转的Treap即可。

有了平衡树。就可以解决一些相对不是那么简单的题目。

4.2.1 Mashmokh's Designed Problem

题目来源：Codeforces Round 240 Div1 E

题目给定一棵有根树，且每个点连出去的边都有顺序。操作有：1.查询两个点之间的距离，2.以点u为根的子树从树中分开，并添加一条其与其某个祖先的连边，作为该祖先的最后一个儿子，3.查询从一个点出发按边的顺序进行dfs，深度为k的最后遍历的点。

这题使用LCT不是特别好解决，但是从dfs序的角度思考问题非常显然。删边和加边操作等价于区间操作。接下来只需要解决这么些问题：

1. 查找一个深度为k的在dfs序中的最后面的元素：注意到在dfs序中点的深度一定是连续变化的。维护一段区间中的深度最深的点和深度最浅的点，即可判断出该段区间是否存在深度为k的点，故而可以二分查找最后出现的深度为k的位置。

2. 利用上述方法亦可以查询一个点任意深度的祖先。

3. 在第一种询问中要查询u, v两个点的LCA，使用倍增可以做到$\log^2 n$的复杂度。考虑dfs序中在u和v之间的深度最小的点，其父亲就一定是LCA，转化为区间最小值，在$O(\log n)$解决。

所以本题就解决了。

4.2.2 dfs序与可持久化

由于使用了平衡树维护dfs序。而平衡树是可以可持久化的。故而有了这题：

题目来源：毛啸

给定一棵以0号点为根的树，要求支持以下操作：

1. 在树上添加一个点，权值为0

2. 将一棵子树的权值统一加上一个值

3. 询问一个点的权值

4. 询问一棵子树的权值和

5. 将当前树的状态变为某次操作后的树的状态

子树权值加减，加点和子树查询即区间查询都是非常基础的平衡树操作。

但是一旦要求可持久化后，就给问题制造了困难。如果要查询一个点对应的区间，就必须沿着这个点的父亲进行查询，而如果在平衡树中维护父亲，是

不可能做到可持久化的。除非将内存池进行持久化，但这仅限理论分析，在实际中无论是常数还是编程复杂度都无法接受。

必须使用不维护父亲的平衡树，但又必须查询到指定的点。为了解决这个问题，故在解法中引入了动态标号的概念：

给平衡树中的每个点维护一个标号，再对标号和平衡树同时进行可持久化。为了维护标号，需要使用到重量平衡树，参见6。具体来讲，如果选择使用Treap来维护持久化平衡树，在插入一个点之后，修改该子树中所有的标号。由于重量平衡树的性质，子树的期望大小为$O(\log n)$。为了将该标号持久化，需要再使用一个可持久化数组，如果用线段树实现，时间复杂度为$O(n \log^2 n)$。

4.3 欧拉遍历

下面将介绍一下欧拉遍历树，即ETT。

4.3.1 什么是欧拉遍历

设T为一个无向树。将T中的每条边拆成两条有向边，然后任选一个节点S作为起点进行DFS，以上过程称为对树T的欧拉遍历。注意，以任一节点作为起点进行欧拉遍历都可能的到相同的结果，因为欧拉遍历对应的是一个环。

4.3.2 欧拉遍历序列

在欧拉遍历中经过的有向边形成的序列。

4.3.3 欧拉遍历树

维护欧拉遍历序列的二叉搜索树。一般用Splay 或Treap实现。

欧拉遍历树注重的是对一棵树整体的信息进行维护。与DFS序不同的是，它可以维护以任意节点作为根时所有子树的信息。故而，可以用来实现换根这个对于dfs序有点困难的问题。

4.3.4 一些实现的细节

实际上只要支持两个操作即可：加边和删边。

1. 加边（连接两颗树）：可以看做是将两个环拼在一起。先把两个环拆开再拼到一起即可。注意两环之间拼接的位置。

2. 删边（将一棵树分成两颗树）：可以看做是将一个环分成两个环。这个操作也不难。

4.4 ETT的简单应用

给定一张图，每次可以加入一条边，删除一条边，查询任意两个点的连通性[2]。

这个问题的做法比较复杂，这儿只做一些简单描述。

因为要求的是两点之间是否联通，不由而然会联想到生成森林。但是这个做法无法解决删边的问题。

故而使用分治算法：给每条边一个标号e_i，使得标号小于等于e的边形成的连通块的大小小于等于2^e。可见$\max(e)$为$\log n$。用ETT对于所有e，维护标号小于等于e的边关于标号的最小生成树。

加边的话，将边的标号设为$\log n$，加入该层的最小生成树，时间复杂度为$\log n$。

询问连通性，在第$\log n$层查询两点的连通性，时间复杂度$\log n$。

最关键是删边。首先找到这条边的标号，如果它不在当前标号的最小生成树中，则直接删除，复杂度依旧为$O(\log n)$。

否则，设边的两个端点为u, v，在所有大于等于这条边的标号的图中寻找连接u, v所对应的子树的边：找到u, v所对应的子树，设u的子树大小小于v的子树大小。暴力扫描u这棵子树向外连接的所有当前标号的边，一旦和v所对应的子树相连，就找到了替代边，否则，将这些边的标号减1，加入当前标号-1的图中。因为u子树的大小必然是原来子树大小的一半以下，所以不会破坏分治的性质。

这样，标号减1的操作次数就是总时间复杂度，而标号的总和为$O(m \log n)$，总时间复杂度为$O(m \log^2 n)$。

因为要维护子树大小，进行删边和加边操作，以及查询一棵树中连接的当前标号的边，故而可以使用ETT实现。

[2]经典问题

5 LCT的一些扩展

LCT不仅仅可以完成简单的链操作，还可以实现一些比较有趣的问题。这儿举两个例子进行说明。

5.1 LCT与子树信息维护

在前面的内容中，用dfs序和ETT解决了子树询问问题，但是是否LCT就完全不能解决子树问题呢？

比如说4.1.1这题，难道就没法使用LCT实现么？其实是可以的。

不妨考虑在动态树对于每个点维护这样的信息：这个点连出去的所有虚边所对应的子树中权值的最大值（为了方便，通常会包括自己）。如果知道了每条虚边对应的子树中的权值的最大值，就可以使用堆进行维护。记这个值为A值。

同样的，可以定义一条链上的最大值为链上所有点连出去的虚边的最大值，即所有点的A值的最大值，这个在Splay中可以很容易维护。这样，查询以链上某个点为根的子树中的权值最大值，只需要在Splay中查询区间中最大的A值。

接下来考虑A值是如何变化的，动态树最关键的操作是access。在access的一次操作中，会将一条实边切换成虚边，将一条虚边切换为实边。在实边变成虚边时将儿子对应的子树最大值加入父亲的A值中，反之则删除。随着access的进行，相应地会修改点到根的路径上的所有信息。不过由于LCT保证了均摊$O(\log n)$的复杂度，再乘上维护A值所使用的堆的复杂度，问题在均摊$O(n\log^2 n)$的复杂度内解决。

关于换根操作，在此做些额外的说明，A值的维护是和根完全无关的，故而无需担心因为换根操作而使得这个做法失效。

5.2 LCT与链翻转

这个问题来自于这么一道题：

5.2.1 Lord

题目来源：IOI2012中国国家队训练陈文潇

求一个子树中的数的和。要求支持以下操作:

1. 查询以x为根，y子树中的所有数的和。

2. 将x,y的路径上的数翻转。

3. 将x 的值改为y。

4. 将x,y相连。

5. 将以x为根,把y向其父亲的边删除。

这题只有询问子树和的操作。但是由于链翻转的操作，从而不能够使用ETT。只能考虑动态树。

使用之前提到的做法，确实可以维护子树信息，但是怎么实现将路径上的权值进行翻转呢？

不妨考虑这么一种做法：并不在动态树上维护权值，而是对于动态树上的每一条链用一棵新的Splay维护其权值，这样翻转操作就可以实现了。再在动态树上对每个点维护其连出去的虚边的子树权值和即可解决问题。

初看之下复杂度是$O(n\log^2 n)$，但实际上两种Splay操作是等价的，不会破坏均摊分析，总时间复杂度应该依旧是动态树的$O(n\log n)$。

6 sone1

6.1 题目大意

sxyz里有一群神犇。要求对树维护：

1. 将某棵子树中所有点的权值加上或变成指定的数。

2. 将某条链上所有点的权值加上或变成指定的数。

3. 查询子树中的权值最大值，最小值，或者权值和。

4. 查询某条链上所有点的权值的最大值，最小值，或者权值和。

5. 加边和删边。

6.2 ETT的做法

这个问题有链操作，ETT是不可能对链进行操作的。LCT中一旦涉及子树标记，也会变得相当复杂。

但是还是有办法将这个问题解决。考虑对一棵树进行欧拉遍历，在ETT中一个点第一次出现之后的连续一段就是从这个点开始不断往第一个儿子移动的路径。这意味着在ETT中将路径表示出来是完全可行的。

那么就可以这么做：结合LCT，将每个点的第一个儿子设为其在LCT中实边连出的唯一的儿子——改变子树顺序在ETT中只不过是改变区间顺序而已。

如果在LCT进行了一次access操作，会依次将一些边变成实边，那么就在ETT中进行相应的修改。对于换根操作，由于LCT和ETT对其都有良好的支持，故而可以解决。考虑这样做的时间复杂度：LCT的时间复杂度为$O(n\log n)$，意味着在ETT中进行的相应操作也是$O(n\log n)$ 次，总时间复杂度为$O(n\log^2 n)$。

接下来考虑查询和修改操作。由于链和子树都对应到了相关的区间，故而可以在ETT中用平衡树进行维护了。

6.3 LCT的扩展

只是利用动态树的话，是没法做子树操作的——而这题的关键就在于子树修改和子树询问，由于有了这个，本题的难度增加了几个级别。但是依旧可以使用LCT做到$O(n\log n)$。

6.3.1 一个简单技巧

本题的修改操作要么是将一些点的权值加上一个数，要么是赋值，故可以将标记设为$a*x+b$的形式，记下a和b的值即可，这样写程序会异常的方便。

6.3.2 一张好看的图

很明显，如果套用传统的动态树的话，问题就在于标记的下传和统计信息的上传上。

考虑一条链：

其中实线是实边，虚线对应于虚边。点是按照深度从高到低。

6.3.3 信息的合并

因为用一个Splay维护实线连起来的链，那么该链里的最大和最小值信息很容易统计。

接下来考虑虚边，由于需要合并子树信息，需用每条虚边连出去的点对应的子树信息[3]来更新实链中的信息。

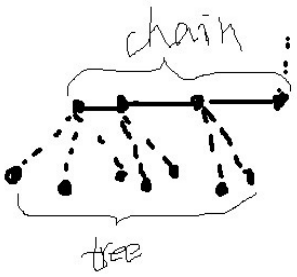
考虑对于一条链需要维护哪些信息：

1. chain: 对于每条链我们统计链内部的点的信息。
2. tree: 所有从这条链上的点连出的虚边对应的子树的信息的和（不包括这条链）。
3. all: 以上两个信息的和，也就是这条链最顶端的点所对应的子树的信息。

注意链是用Splay维护的，也就是说，要在Splay 的每个节点都维护对应的值，实现和普通的动态树没什么区别。

为了看起来更像区间，将树链横放：

[3]因为子树信息可以通过统计链的信息得到，故而之后不区分子树信息和链上的信息



chain的话使用Splay即可维护。如果统计出tree 的话，all只是简单地将其加起来。

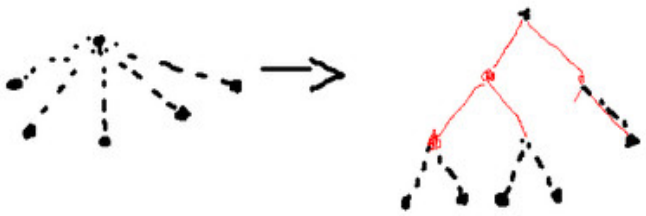
那怎么统计tree呢？

如果对于每个点暴力记下其连出去的所有虚边，就可以直接枚举这些虚边对应的链，用其all来更新该点的tree即可。

但是这个显然是不现实的，因为——太慢了。故而需要在原有的LCT上进行修改：

6.3.4 AAA树

AAA树[4]为此而存在了：对于每个点连出的虚边，同样使用Splay维护这些点，为了写程序方便，这步是通过增加一些内部节点（称之为inner）实现，如图：



其中红色节点就是新增的内部节点。并保证所有的叶子（即黑色节点）就是原树中虚边上的对应的点，内部节点不会超过叶子节点的个数，所以这样做

[4]这只是作者对该结构的称呼，单纯为了叙述方便

是不会改变复杂度的。

显然每条链只会属于一个AAA树，每个点的AAA树包含了这个点连出去的所有虚边。

通过AAA树，可以实现两种操作：

1. add：将一条链的根接到另一条链的某个点上，那么只需将这条链对应的根按照Splay的普通插入插到那个点的AAA树中即可（注意为了保证一定为叶子节点，需要新增内部节点）。

2. del：将一条从虚边连出的链砍掉，使用普通的Splay删除即可。（同样注意这样有可能导致某个黑色儿子个数少于1 的内部节点，则也要将该内部点删除，有一个儿子的话，需要将这个儿子代替自己连到自己的父亲上）

这样，对于实链上的某点的信息进行更新的话，直接使用AAA树的根的信息来更新即可。

6.3.5 AAA树与access操作

接下来考虑在这样的结构下的access——也就是动态树最重要的操作了，有了这个一切动态树的问题基本都解决了。

可以发现，有了add和del两个操作后，对于access操作并没有影响。

在access的每一个循环中，只做了这样几件事：

1. 找到现在的链的父亲（这个直接在该链的AAA树中找到根的父亲即可）。

2. 将一条实边变成虚边，这其实就是add 操作。

3. 将某条虚边变成实边，这其实就是del 操作。

这样在LCT中维护信息的合并也就成为了可能。

6.3.6 标记

怎么实现修改操作，也就是如何实现标记和标记的下传。

很明显有两种标记:

首先是子树标记treeflag，表示这条链对应的子树中的修改（注意不能包含这条链的修改）。其次是链标记chainflag，也就是这条链的修改。

链标记的下传也是简单且常规的。

而treeflag的下传则有两种：

1. 在实链的Splay中（或者是下传到内部节点），直接下传即可，无需修改。
2. 若从AAA树下传到外部节点，需将treeflag拆分成chainflag+treeflag的形式。因为treeflag不包含在链上的修改

6.3.7 实现细节

怎么写呢？看起来AAA树好麻烦啊。

其实不然，只需将LCT中Node的两个儿子改成四个儿子$ch[4]$。

这样只需要传递一个标记type即可决定是考虑在LCT中进行操作还是在AAA树中。

并给内部节点一个inner标记，方便标记。

6.3.8 时间复杂度

在不严谨的分析下，这个做法的复杂度至多不会超过$O(\log^2 n)$。在相关论文9 中证明了这样做的复杂度在均摊意义下单次操作是$O(\log n)$的，虽然有常数为96的因子。但这个做法在实际中也有着较快的运行速度。

6.3.9 名字?

在9中称为Self-Adjusting Top trees。但如果只是看做LCT的一种简单拓展将会更易理解。

6.4 Rake和Compress方法

在11中袁昕颢提到了一种有意思的数据结构：RC tree。

具体来讲，对于一棵树，可以使用两种方式将其划分成更小的子树：Rake节点，代表了以原树中以某个点为根的子树。Compress 节点，代表了原树中一条链上所有点的子树。

利用这两种节点对树进行剖分，可以达到和动态树相类似的效果。加上引入了随机的策略，最终得到期望$O(\log n)$的深度。但在top tree的论文9中提到了这个做法的局限，它要求一棵树的节点个数为常数。若牺牲一定的时间复杂度，也是有可能解决最初的动态树问题。作者并未实现，不敢妄下断言。

7 动态仙人掌

这部分内容是向吕凯风同学致敬。在此会对动态仙人掌问题和相关解法做一定介绍。

7.1 问题描述

定义仙人掌图为每条边最多属于一个简单环的图。

动态仙人掌问题是要求动态地维护一个仙人掌图。由于科技水平不发达，故而目前只限于路径查询和路径修改操作。

对于每条边有一个固定的权值A_i，则路径(u,v)为u到v的所有路径中$\sum A_i$最小的路径。

动态仙人掌这题分为三个难度递增的子问题：

1. 动态仙人掌I：加边和删边，查询最短路径的长度。
2. 动态仙人掌II：在I的基础上对于每条边还有一个固定的权值B，要求查询一条路径上最小的B值。
3. 动态仙人掌III：在II的基础上要求支持对一段路径的B值进行操作。

7.2 一个优秀的算法

最显然的做法就是用LCT维护仙人掌的生成树。可以做到时间复杂度$O(n \log n)$，达到理论最优。

不过该做法的具体实现过于复杂，故而略去对此的讨论。有兴趣的读者请参见吕凯风同学的相关题解。

7.3 Link/cut cactus

为何维护生成树会如此繁琐？因为将一个树上的算法直接套到仙人掌图上本身就无法避免各种复杂的讨论。为了得到一个很好写的做法就不得不跳出树的框架。不妨思考怎么修改Link/cut tree这个数据结构以使之适应于仙人掌图。于是就得到了一个崭新的数据结构Link/cut cactus。

7.4 仙人掌图剖分

仿照LCT，为了得到LCC的做法，必须先考虑如何将仙人掌图分成若干条不相交的链。指定任意一个点为根。

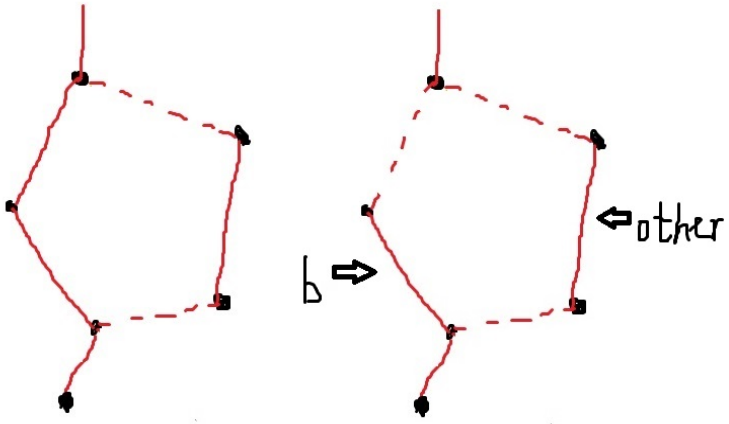
如果不存在环，一切都可以按照LCT进行定义了，否则？

首先必须保证剖分成的链的一定是一条合法的路径（这条链为两个端点的最短路）。如果将环看成LCT中的一个点，还要保证对于一个环，至多只会往下（远离根的方向）连出一条边。

其次，虽然一个点可以属于任意多的环，但只能属于一条链。

将链中的边称为实边，不在链中的边称为虚边。

令一个环中离根最近的点a为这个环的根，设点b有往下连出的边。则可能存在如下两种剖分方式：



其中实线和虚线表示剖分中实边和虚边。

注意到，除去根，一个环最多分成两条链，且仅有一部分的一个端点会存在向下的实边，称这部分为b链，另一部分为other链。

两者的不同点在于根与环中其它点是否相连。可以看出，一个点可能成为许多环的根，但至多在一个环的b链或者other链中。这意味着可以直接维护所有的链。

7.5 access

下面考虑access，对于动态树来说，只要能够实现access操作问题就基本解决了。

首先考虑其意义：将一个点到根的最短路径连成一条链。

假设在access中当前的点为u，查找u在*LCC*上的往前连的边（可以使用链和最短路径定义）。如果这条边不在环中，则做法类似于LCT。

否则，找到这条边所在的环。可以肯定u不是这个环的根（设为a）。故而可以实现操作：将b链和other链连接起来，从u处断开，使得点u 位于较短的那条链上，得到新的b和other链，而且点u刚好为b的端点。将这个操作称为Expand

将u往下的链与新的b链连接即完成了对点*u*的操作。

注意点a：如果a与本来的b链相连，需在最后将a与新的b链连接起来。否则，先对点a进行了Expand操作，再与新的b链相连。

7.6 换根

很幸运的是，换根操作非常简单：对新的根节点进行access操作，得到路径，将其翻转。

注意，LCC要求对每个环维护相应的a,b,other。经过翻转之后会发生相应变化。解决方法很简单：每次访问到一个环，在路径中查询a,b的顺序，一旦顺序不同，则将环中相应的链进行翻转，相应的值对调。

7.7 其它操作

有了access和换根操作，其它操作就非常简单了。

7.8 一些小细节

因为此题的权值都在边上，故而要对每个点维护链中与之相连的两条边，使得在翻转操作后依然可以得到正确的边的集合。

此题要求判断最短路径是否唯一，解决方法是对于每个环记录b链和other链长度是否相等，从而可以对相应的路径进行标记。

7.9 时间复杂度

由于LCC的中实边和虚边的切换与LCT相类似，总次数不会超过$n \log n$，考虑到Splay，时间复杂度为$O(n \log^2 n)$[5]。经实测，效率也是挺不错的。

7.10 一些推广方向

LCC是和LCT一样非常灵活的数据结构，可做如下探讨：

1. 是否可以可持久化。
2. 是否可以将self adjusting top tree和LCC结合，使得能够解决子仙人掌的操作和查询。
3. 是否可以将dfs序推广到仙人掌图上。
4. 如何使用LCC来优化仙人掌图上的统计和最优化问题。

欢迎有兴趣的同学继续思考相关的问题。

后记

本文对信息学竞赛中涉及到的动态树问题进行了简单的介绍。

选择这个简单的题目，原因可能是因为个人对这方面的问题比较感兴趣。近两年来动态树成为各种模拟比赛中的热门题目，题目难度越来也大，类型越来越多。故而有了对这类问题进行总结的想法。

由于时间仓促，以及作者能力有限，很多问题都没有涉及。不过也希望这篇论文能够帮助到将要或正在学习动态树的OIer们，这样的话，本文也就能有所意义了。

[5]怀疑时间复杂度为$O(n \log n)$，但是无法证明

感谢

感谢廖晓刚老师的在学习生活上的关心和照顾。

感谢朱全民老师和屈运华老师的教导。

感谢国家集训队教练的辛勤付出。

感谢CCF提供的平台和机会。

感谢匡正非，刘研绎，毛啸，彭雨翔，杨定澄，刘剑成，谭博文，杨卓林，苏雨峰等同学提供的意见和帮助。

感谢陈立杰，我从他那儿学到了top tree及其它各种各样有趣的算法。

感谢吕凯风在动态仙人掌问题上做出的开创性的研究。

感谢唐翔昊，魏子昆，陈思琪，仇知，徐诗雨，黄骏翔，王子骏等同学陪我度过了快乐的时光。

参考文献

[1] 刘汝佳, 黄亮,《算法艺术与信息学竞赛》, 清华大学出版社。

[2] 刘汝佳,《算法竞赛入门经典》, 清华大学出版社。

[3] 黄志翱,《浅谈算法竞赛中的简单构造题》

[4] 杨哲,《SPOJ375 QTREE解法的一些研究》, 2007年国家集训队作业。

[5] 何朴藩,《HNOI 2010 解题报告》, 2011年国家集训队作业。

[6] 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年国家集训队论文。

[7] 6.851: Advanced Data Structures L20

[8] 匡正非, 黄志翱,《两个冷门图论算法》, 2014年信息学奥林匹克冬令营。

[9] Robert E. Tarjan, Renato F. Werneck, Self-Adjusting Top Trees.

[10] 陈首元, 维护森林连通性——动态树

[11] 袁昕颢, Dynamic Trees Problem, and its applications.

[12] 吕凯风，《动态仙人掌系列题解》

随机化算法在信息学竞赛中的应用

湖南师大附中 胡泽聪

摘 要

随机化算法是一类十分有用的算法。通过引入随机函数，许多随机化算法可以在期望的意义下达到较之于确定性算法更优的时间复杂度。本文将简要介绍随机化算法的定义与分类，针对一些信息学竞赛中的例题分析随机化算法的应用与优越性，并总结出设计随机化算法的一般思路。

引 言

在信息学竞赛中，需要选手设计随机化算法的题目似乎比较少见，但是一旦出现了，往往会让不少选手束手无策。许多选手对随机化算法的了解仅限于各类骗分算法，因此在本文的前三章中，我们将简要介绍随机化算法的定义与分类。

由于随机化算法包含的范围较广，并没有一些通用的解决问题的方法，只能具体问题具体分析。故在本文的第四章中，我们将通过对于例题的分析，探究随机化算法的应用以及效果，并总结设计随机化算法解决实际问题的一般思路与分析方法。

1 定义与分类

随机化算法，也称概率算法，是这样一种算法，在算法中使用了随机函数，而且随机函数的返回值直接或者间接地影响了算法的执行流程或者执行结果。算法的一些决策可能依赖于随机的选择，并且可以通过随机因素改善算法的期望时间复杂度，或者得到可以接受的正确率。

随机化算法可以被分为以下三类：

- 数值概率算法;
- Monte Carlo算法;
- Las Vegas算法。

这三类算法都有着各自的特点与区别:

- **数值概率算法** 是通过随机选取元素从而求得在数值上的近似解。较之于传统方法，其运行速度更快，而且随着运行时间的增加，近似解的精度也会提高。在不可能或者不必要求出问题的精确解时，使用数值概率算法可以得到相当满意的近似解。常见的数值概率算法有**随机撒点法** [1]。
- Monte Carlo**算法** 总是能在确定的的运行时间内出解，但是得到的解有一定概率是错的。通常出错的概率比较小，因此我们可以通过反复运行算法来得到可以接受的正确率。常见的Monte Carlo算法有Miller-Rabin**素性测试算法** 。
- Las Vegas**算法** [2]总是能返回正确的结果，但是其运行时间不确定[3]。对于一些平均时间复杂度优秀，但是最坏情况下的复杂度较高的确定性算法，通过引入随机函数，尝试减小出现最坏情况的可能性，改造成Las Vegas算法，可以在期望意义下达到优秀的时间复杂度。常见的Las Vegas算法有**快速排序算法** [4]。

本文将侧重于介绍Monte Carlo算法与Las Vegas算法，而对于数值概率算法将不会过多提及，有兴趣的同学可以自行研究。

2 Monte Carlo算法

在信息学竞赛中，Monte Carlo算法应该是使用得相对较多的一类随机化算法。相比起Las Vegas算法，我们有时可以利用题目的性质，将一个确定性算法

[1]随机撒点法的一个主要应用是求不方便直接计算的图形的面积。

[2]在少数一些资料中，此类算法被称为Sherwood算法，并将运行时间确定但可能得不到解的随机化算法（如Pollard rho算法）称为Las Vegas算法。参考了许多其它资料后，作者决定将此类算法称为Las Vegas算法；对于前面提到的可能得不到解的算法，则直接将其归类为Monte Carlo算法，不单独分为一类。

[3]这里我们要求算法必须能在有限的时间内结束运行。运行时间有概率为无限的算法，比如低效的Bogo排序算法，不算作Las Vegas 算法。

[4]快速排序算法的随机性来源于划分过程中“基准”(pivot)的选择。

直接改造成Monte Carlo算法。

2.1 Monte Carlo算法的分类

在介绍例题之前，我们先对Monte Carlo算法进行分类。根据问题的性质，Monte Carlo算法可以分为以下两种：

- 求解最优化问题的Monte Carlo算法；
- 求解判定性问题的Monte Carlo算法。

而对于求解判定性问题的Monte Carlo算法，我们可以再进一步地分类为以下三种：

- **假倾向**(false-biased) [5]Monte Carlo算法。当这类算法的返回值为**假** 时，结果一定正确，但当返回值为**真** 时则有一定概率错误。
- **真倾向**(true-biased) Monte Carlo算法。当这类算法的返回值为**真** 时，结果一定正确，但当返回值为**假** 时则有一定概率错误。
- **产生双侧错误**(two-sided error) 的Monte Carlo算法。这类算法无论是返回真或假都有概率错误。

其中假倾向Monte Carlo算法和真倾向Monte Carlo算法并称为**产生单侧错误**(one-sided error) 的Monte Carlo算法。

我们在信息学竞赛中遇到的Monte Carlo算法基本上都是产生单侧错误的算法，因此下面的讨论中我们将忽略产生双侧错误的Monte Carlo算法。

2.2 Monte Carlo算法的正确率与复杂度

Monte Carlo算法的正确率与复杂度很好计算。假设我们有一个正确率为p，时间复杂度为$O(f(n))$的Monte Carlo算法。设我们运行这个Monte Carlo算法k次，那么显然有

- 正确率为$1-(1-p)^k$；
- 时间复杂度为$O(k \cdot f(n))$。

[5]由于没有找到对应的中文翻译，此处的中文为作者自己对括号内英文的翻译。以下带括号的词语亦同。

3 Las Vegas算法

相比起Monte Carlo算法，Las Vegas在信息学竞赛中则使用得相对较少。我们用到的Las Vegas算法大多是已有的算法，如**快速排序算法**、**随机增量算法**和Treap，需要选手自行设计Las Vegas算法的题目很少。在下面的例题中，也只有一道需要使用Las Vegas算法。

Las Vegas算法并不像Monte Carlo算法那样有各种类别，因此我们直接分析其复杂度的计算。

3.1 Las Vegas算法的复杂度

不同的Las Vegas算法之间的差别比较大，因此不存在通用的复杂度分析方法。我们将选取两个Las Vegas算法作为示范，来分析其复杂度。

3.1.1 快速排序算法

想必大家对快速排序算法都非常熟悉，并且熟知其最优复杂度为$O(n\log n)$，最差复杂度为$O(n^2)$，期望复杂度为$O(n\log n)$。这里我们尝试证明其期望复杂度。

设$T(n)$为对长度为n的序列运行快速排序算法所需的期望时间，我们有

$$T(0)=0$$

以及

$$T(n)=n+\frac{1}{n}\sum_{i=0}^{n-1}(T(i)+T(n-i-1))$$

我们通过放缩来获得对$T(n)$上界的一个估计

$$\begin{aligned}T(n) &= n+\frac{1}{n}\sum_{i=0}^{n-1}(T(i)+T(n-i-1))\\ &= n+\frac{2}{n}\sum_{i=n/2}^{n-1}(T(i)+T(n-i-1))\\ &= n+\frac{2}{n}\sum_{i=n/2}^{3n/4}(T(i)+T(n-i-1))+\frac{2}{n}\sum_{i=3n/4}^{n-1}(T(i)+T(n-i-1))\end{aligned}$$

由于$T(n) \geq n$，所以对于$\frac{n}{2} \leq i \leq j$我们显然有

$$T(i) + T(n-i) \leq T(j) + T(n-j)$$

因此

$$\begin{aligned} T(n) &\leq n + \frac{2}{n}\sum_{i=n/2}^{3n/4}(T(3n/4) + T(n/4)) + \frac{2}{n}\sum_{i=3n/4}^{n-1}(T(n-1) + T(0)) \\ &\leq n + \frac{1}{2}(T(3n/4) + T(n/4)) + \frac{1}{2}T(n-1) \end{aligned}$$

我们要证明$T(n) = O(n\log n)$，就需要证明存在常数c满足$T(n) \leq cn\log n$。我们考虑用数学归纳法证明。$n = 0$时定理显然成立。现假设对于任意$m \leq n$定理皆成立，那么

$$\begin{aligned} T(n) &\leq n + \frac{1}{2}(T(3n/4) + T(n/4)) + \frac{1}{2}T(n-1) \\ &\leq n + \frac{1}{2}(c(3n/4)\log(3n/4) + c(n/4)\log(n/4)) + \frac{1}{2}c(n-1)\log(n-1) \\ &\leq n + c\left(\frac{3n}{8}\log(n) - \frac{3n}{8}\log(4/3) + \frac{n}{8}\log(n)\right) - \frac{n}{8}\log(4) + \frac{n}{2}\log(n)\Big) \\ &= cn\log(n) + n\left(1 - \frac{3c}{8}\log(4/3) - \frac{c}{4}\right) \end{aligned}$$

当$1 - \frac{3c}{8}\log(4/3) - \frac{c}{4} \leq 0$时，也即约$c \geq \frac{5}{2}$时，我们有

$$T(n) \leq cn\log n$$

故归纳成立，$T(n) = O(n\log n)$。

3.1.2 一类由Monte Carlo算法改造而成的算法

对于一类一定有解的构造性问题，假设我们有一个正确率为p，时间复杂度为$O(f(n))$的产生单侧错误的Monte Carlo算法，我们可以通过不断运行该算法，直到找到解为止，来将其改造为Las Vegas算法。我们关注的是该算法的期望运行次数k。

我们可以列出k的计算式：

$$k = \sum_{i=1}^{\infty} p(1-p)^{i-1}i \tag{1}$$

两边乘上$1-p$得

$$(1-p)k = \sum_{i=1}^{\infty} p(1-p)^i i = \sum_{i=2}^{\infty} p(1-p)^{i-1}(i-1) \tag{2}$$

令式(1)减去式(2)得

$$p \cdot k = p + \sum_{i=2}^{\infty} p(1-p)^{i-1} = p \sum_{i=0}^{\infty} (1-p)^i \tag{3}$$

而右侧的合式是等比数列的求和，因此可以得出

$$k = \sum_{i=0}^{\infty} (1-p)^i = \frac{1-(1-p)^{\infty}}{1-(1-p)} = \frac{1}{p} \tag{4}$$

结果十分简洁。

因此，这个算法的期望运行时间为$O(p^{-1}f(n))$。

4 例题与分析

下面我们通过一些例题来体会如何使用随机化算法解决信息学竞赛中的问题。

例题一：MSTONE[6]

平面上有n个互不重合的点，已知存在不超过7条直线可以覆盖全部的点，问在平面上作一条直线，最多能覆盖多少个点。

$n \leq 10000$。

题目分析

我们不妨先考虑一个最朴素的算法：枚举两个点，确定一条直线，然后判断有多少个点在这条直线上。这个算法的复杂度为$O(n^3)$，无法在时间限制内求出问题的解。

我们考虑向这个朴素算法中加入随机，将其改造成Monte Carlo算法。问题就在于，在哪一个部分随机?

[6]**题目来源：** CodeChef。

注意到题目有一个很关键的条件：存在不超过7条直线可以覆盖全部的点。我们先来分析一下这个条件能给我们来带什么。

引理1.1. 覆盖了最多点的直线覆盖了至少$\left\lceil \frac{n}{7} \right\rceil$个点。

从而引出下面的定理

定理1.2. 从给定的n个点中随机选择两个点，过这两个点的直线与覆盖了最多点的直线重合的概率为$\frac{1}{49}$。

证明比较显然。

那么我们将“枚举两个点”的步骤改为“随机选择两个点”，从而将算法改造成Monte Carlo算法。运行这个算法一次的复杂度为$O(n)$，正确率为$\frac{1}{49}$。设算法运行k次，取$k = 1000$，可以求出算法的正确率为

$$1 - \left(1 - \frac{1}{49}\right)^k \approx 1 - 10^{-9}$$

这个正确率相当令人满意，而且运行时间方面也可以接受。 □

例题二： Ghd[7]

令一个集合S的“GHD”为

$$\max\left\{GCD(S') \middle| |S'| \geq \frac{1}{2}|S|\right\}$$

给定长度为n的序列a，求$GHD(\{a_1, a_2, \ldots, a_n\})$。

$n \leq 10^6$，$1 \leq a_i \leq 10^{12}$。时间限制为4秒。

题目分析

我们依然先考虑不加入随机的算法。一个朴素的算法是枚举GHD，然后检验是枚举的GHD的倍数的数有多少个。一个稍微优化一些的算法是枚举每个数的每个因子，当然这样也是没法通过这道题的。

我们不妨直接枚举每个数，然后直接考虑当这个数在选出的集合S'中时，可以取到的最大的GHD。设这个数为a_x，我们求出a_x和其它每个数a_i的GCD，记

[7]**题目来源：** Codeforces Round #213 (Div. 1) - Problem D。

为g_i。如果a_i也在集合S'中，能成为GHD的只能是g_i的因子。因此我们对于a_x的每个因子d记录cnt_d，代表有多少个数加入集合S'后GHD可以为d。那么对于每个g_i，枚举其所有因子p，并令cnt_p加1。

设$\tau(n)$为n的因子个数，这个算法的复杂度为$O(n(\sqrt{a_x}+\tau^2(a_x)+n\log n))$。在不超过$10^{12}$的所有数中，因子最多的数为$963,761,198,400$，有6720个因子，如果去掉枚举的部分，复杂度是可以承受的。

现在我们有了一个去掉枚举后时间复杂度可以接受的算法。我们仿照上一题的做法，考虑用随机来替代枚举。我们有如下定理：

定理2.1. 从给定的n个数中随机选择一个数，这个数在最终选出的集合中的概率为$\frac{1}{2}$。

证明十分显然。

那么我们将“枚举一个数”改成“随机选择一个数”，从而将算法改造成Monte Carlo算法。这个算法的正确率为$\frac{1}{2}$，可以说是相当之高，只需运行数十次即可得到满意的正确率。 □

算法设计思路1

我们可以从这两道例题中总结出一个大致的算法设计思路，与这两道题类似的题都可以使用下面的思路来设计一个可行的Monte Carlo 算法：

1. 设计一个能解决问题的确定性的算法。
   - 这个算法需要枚举一些元素；
   - 设这个算法的复杂度为$O(f(n)g(n))$，其中$f(n)$为枚举部分的复杂度，$g(n)$为单次枚举中计算所需的复杂度。应当保证运行若干次$O(g(n))$的算法不会导致超时。
2. 向算法中引入随机。将算法的枚举部分改成随机选择。
   - 你需要证明随机算法的正确率，并计算为了达到一个可以接受的正确率需要运行该算法多少次；
   - 通常情况下，需要使用随机算法的题目都有较强的约束条件，可以保证在随机选择的步骤中，选到所需要的元素的概率不会太低；

- 即使正确率不尽人意，我们也可以使用这个随机算法获得一定的部分分。在遇到提交答案题的时候尤为有效。 □

我们尝试使用上面的思路来解决下面两道例题：

例题三：Couriers[8]

给定长度为n的序列a，有m次询问，每次给定l和r，问$a[l..r]$中是否存在一个元素出现次数严格大于$\frac{1}{2}(r-l+1)$，如果有则输出该元素，否则输出-1。

$n, m \le 500000$，$1 \le a_i \le n$。

题目分析

我们还是先设计一个朴素的算法。我们可以枚举区间内的每个数，然后统计这个数在区间内出现的次数，并判断是否是答案。直接这么做的复杂度是$O(mn^2)$。

我们可以预处理每种数字出现的位置的序列，统计区间中出现次数的时候在序列中二分，这样把复杂度降到了$O(mn\log n)$，但还是会超时。

现在我们应用设计思路1，用随机来代替枚举。我们将“枚举区间内的每个数”改为“随机选择区间内的一个数”，从而将算法改造为随机化算法。我们有如下定理：

定理3.1. 如果询问的答案不是-1，那么一次随机选到答案的概率不小于$\frac{1}{2}$；否则无论怎么随机都无法找到答案。

故这个算法是产生单边错误的Monte Carlo算法，且正确率为$\frac{1}{2}$。

经实测，我们只需对每个询问运行这个算法20次即可通过这道题。当然为了保险起见，可以在时间允许的范围内运行更多次。算法的复杂度为$O(km\log n)$，其中$k = 20$。 □

例题四：TKCONVEX[9]

给定n条线段的长度，求一个从n条线段中选出$2k$条的方案，使得能用这些线段构成两个有k条边的凸多边形，或者指出问题无解。

[8]**题目来源：** POI2014。

[9]**题目来源：** CodeChef。此处的数据范围与原题相比有所不同。

$n \le 1000$，$3 \le k \le 6$。

题目分析

我们先考虑一个基础的问题：给定n条线段，判断这些线段是否能构成一个凸多边形。我们有如下引理：

引理4.1. 假设n条边的边长为$a_1, \ldots, a_n$，这些边能构成一个简单多边形当且仅当对于任意$1 \le i \le n$有

$$a_i \le \frac{1}{2}(a_1 + \cdots + a_n)$$

这个条件也等价于

$$\max(a_1, \ldots, a_n) \le \frac{1}{2}(a_1 + \cdots + a_n)$$

这个引理的证明与本文主题无关，故在这里略去。有兴趣的读者可以自行思考。

注意这个引理只能判断边是否能构成一个简单多边形，而并非凸多边形。不过，我们还有一个引理：

引理4.2. 任意非凸多边形都可以通过对其边的平移和旋转得到一个凸多边形。

出于与上面相同的理由，这里略去该引理的证明。那么我们可以用引理4.1来判断给定的线段是否能构成凸多边形。

在本题中，可以选择的集合有$\binom{n}{2k}\binom{2k}{k}$个之多，我们当然不能一个一个地去检验。我们从上面的引理出发，继续探究问题的性质，尝试缩小可选集合的范围。我们可以得出下面的定理：

定理4.3. 假设n条线段中存在可以构成凸多边形的大小为k的子集，那么一定存在一个这样的子集满足，对所有线段按长度排序后，这个子集中的元素对应排序后序列中的一个区间。

证明： 任选一个可以构成凸多边形的大小为k的子集，记为S。设排序后的序列为p，假如S中的元素对应到p上并非一个区间，那么一定存在至少两个$i(i < n)$，满足

$$p_i \in S \text{ 且} p_{i+1} \notin S$$

我们找到满足条件的第二大的i，并从S中删去p_i，再加入p_{i+1}，由引理4.1可知此时集合S中的线段仍然可以构成凸多边形。重复上述操作直到找不到满足条件的i，此时S中的元素对应到p中一定是一个区间。 □

注意到我们的定理只是用来处理选出一个大小为k的集合的情况的。那么我们怎么将其应用到原问题上呢?

一个很直接的想法就是，把线段长度的序列分成两部分，然后在两部分中各找一个长度为k的区间，使得区间中的线段可以构成凸多边形。但是如果要枚举子集，无疑会超时。

既然这里是“枚举”，我们尝试套用上面的设计思路，将其改造为Monte Carlo算法。设两个集合分别为S和$\bar{S}$，对于每条线段，我们有0.5的概率将其放入S中，有0.5的概率将其放入$\bar{S}$中。这样我们选出任意一个子集的概率是相同的。接下来我们对S和$\bar{S}$分别运行上面的算法。运行一次的复杂度可以做到$O(n)$。

我们来分析这个算法的正确率。划分的总方案数为2^n。考虑一个极端的情况，有且仅有两个不相交集合X和Y可以构成凸多边形。那么一个划分方案有解，当且仅当$S \cap X = X$且$S \cap Y = \varnothing$，或者$S \cap X = \varnothing$且$S \cap Y = Y$。这样的方案数为$2^{n-2k} \times 2$。因此在最坏情况下，选出一个合法划分的概率，也即算法的正确率，约为$\frac{1}{2^{2k-1}}$。

设我们运行这个算法t次，则其复杂度为$O(n \log n + tn)$。在不超时的情况下，t最多可以取到约20000，此时算法的正确率在$k = 6$时约为

$$1 - \left(1 - \frac{1}{2^{11}}\right)^t \approx 1 - 10^{-5}$$

已经相当令人满意了。 □

事实上，例题一、例题三与例题四均存在确定性算法[10]。这两道题的确定性算法需要进一步分析题目的性质，与这里介绍的随机化算法相比，思维难度更高。

由此可见，引入随机化可以在一定程度上降低思维难度、简化算法，同时对于一些题目来说，随机化算法还可以达到更优的复杂度。

[10]例题四的确定性算法的大致思路是，证明对于任意大小不小于约70的集合均有解，从而缩小数据范围。接着再证明定理4.3的一个加强，即选出的集合对应一个长度为$2k$的区间，或者两个长度为k的不相交区间。至此就可以直接枚举枚举区间然后用确定性算法解决了。

例题五：向量内积[11]

定义两个d维向量$A = [a_1\ a_2\ \ldots\ a_d]$和$B = [b_1\ b_2\ \ldots\ b_d]$的内积为其对应维度的权值的乘积和，即

$$(A, B) = \sum_{i=1}^{d} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_d b_d$$

现有n个d维向量$x_1, x_2, \ldots, x_n$，问是否存在两个向量的内积为k的倍数。如果存在，输出任意一组解。

数据分为两类：

- $k = 2$，$n \le 10000$，$d \le 100$；
- $k = 3$，$n \le 100000$，$d \le 30$。

题目分析

$k = 3$的情况处理起来较为复杂，而且其难点与本文主题无关，故下面只讨论$k = 2$的情况。

我们将这个n个向量组成一个$n \times d$的矩阵，记这个矩阵为A。令A^T为A的转置，设$B = A \times A^T$，那么$B_{i,j}$的值实际上就是第i个向量和第j的向量的内积。而我们实际上要判断的就是，在B中是否存在一个不在主对角线上的位置(i, j)，满足$B_{i,j} \equiv 0 \pmod 2$。直接计算矩阵乘法与暴力无异，我们需要想别的方法。

为了处理方便，我们把所有向量的每一维都对2取模，计算中也全程对2取模。我们构造一个$n \times n$的矩阵C，满足C的主对角线与B的主对角线相同，而且主对角线以外的元素都是1。那么问题就变成了判断$A \times A^T = C$是否成立，如果不成立则找到一个不同的位置。B的主对角线可以在$O(nd)$的时间内求出来，故构造矩阵C是没有问题的。

分析到了现在，我们似乎无法继续前进了。确定性算法应该是无法避免矩阵乘法的，而矩阵乘法的复杂度无法接受。我们考虑引入随机化，但是这个问题似乎也无法套用思路1，因为可以构造出矩阵B只有很少几个元素不为1的数据。我们需要一种新的随机化思路。

假设$A \times A^T = C$成立，那么对于任意$m \times n$的矩阵X都应满足$X \times A \times A^T = X \times C$，其中$m$为任意正整数。如果取$m = 1$，那么$X$就是一个向量。我们能不能随机生

[11]**题目来源：** NOI2013 Day 1

成X，然后用这个随机的X来检验?

对于$X \times A \times A^T$，计算一次的复杂度为$O(nd)$。对于$X \times C$，由于矩阵C具有很大的特殊性，我们可以直接推出其乘出来的向量。设$Y = X \times C$，那么有

$$Y_i = (C_{i,i} - 1)X_i + \sum_{j=1}^{n} X_j$$

因此我们可以在$O(n)$的时间内求出$X \times C$。在时间复杂度方面是没有问题的。

而在算法正确率方面，我们有如下定理:

定理5.1. 如果$A \times A^T = C$，一定会返回相等。而如果$A \times A^T \neq C$，算法返回相等的概率不超过$\frac{1}{2}$。

证明： 由矩阵乘法运算的结合律可知，定理的第一部分成立。我们考虑如何证明定理的第二部分。

令$D = A \times A^T - C$，由假设$A \times A^T \neq C$得，D中存在至少一个不为0的元素，设其中一个为$D_{i,j}$。我们假设算法返回了相等，现在我们需要求出X的哪些取值会导致这一错误，并求出X取到这样的值的概率。

令$E = X \times D$，考虑E_j。算法会返回相等，当且仅当E的每个元素均为0，因此

$$E_j = \sum_k X_k D_{k,j} = 0$$

由于我们假设$D_{i,j} \neq 0$，因此$E_j = 0$当且仅当

$$X_i = -\frac{1}{D_{i,j}} \sum_{k \neq i} X_k D_{k,j}$$

因此，当X中其它元素都已确定时，要使得E的每个元素均为0，X_i只有一种取值。而$X_i \in \{0, 1\}$，所以X_i取到会导致错误的值的概率为$\frac{1}{2}$。

由此可知，算法返回相等的概率不超过$\frac{1}{2}$，定理的第二部分得证。故定理得证。 □

我们求出的正确率十分的可观，这意味着我们只需要运行算法数十次即可判断相等，在时间上完全可以接受。

现在我们还需找到一个为0的元素的位置。我们找到一个不为0的E_i，那么一定存在一组解包含了第i个向量，我们只需枚举另一个向量并计算检验即可。这一部分的复杂度为$O(nd)$。

至此$k = 2$的情况已经被我们完美地解决了。 □

算法设计思路2

我们试着从上一道题中总结一个具有一般性的思路。对于类似上一题的一些题，可以使用下面这一思路来设计一个可行的Monte Carlo 算法：

1. 针对问题设计一个确定性的算法，这个算法需要用到一个传入的向量。
   - 算法接受的向量中的值不应当依赖于输入数据。换句话说，我们并非从输入数据中抽出一个向量传入到算法的函数中——我们应当将这样的算法归类到设计思路1；
   - 通常这一类算法都直接是为了随机化而设计的，因此比起设计思路1，拥有更高的思考难度。
2. 向算法中引入随机。随机生成一个向量，并传入到上面算法的函数中。
   - 可以使用这一类算法通过，或者是标准算法就是这一类算法的题目不多。但这并不代表这一类算法没有意义；
   - 对于一些有多维代价的最优化问题，可以用这一类算法获得一定的部分分。具体做法是随机一个向量代表权重，并重新定义代价为原代价向量与权重向量的内积。以新的代价求一遍只有一维代价的最优化问题，并构造方案，计算出实际代价；
   - 而对于一些和计算几何有关的题目，也可以使用这个方法。 □

例题六：Graph Reconstruction[12]

给定一个含有n个顶点和m条边的无向图，其中每个顶点的度数不超过2，且图中无自环与重边。构造一个新图，满足以下条件：

- 新图含有n个顶点与m条边；
- 新图中每个顶点的度数不超过2，且不含有自环与重边；
- 假设在原图中顶点u和v之间存在一条边，则在新图中顶点u和v之间不得有边。

输出任意一个满足条件的新图，或指出无解。

$1 \le m \le n \le 100000$。

[12]**题目来源：** Codeforces Round #192 (Div. 1) - Problem C

题目分析

我们先分析题目中给出的图的性质。由于每个顶点的度数不超过2，因此原图应该是一些环和链，而且在原图的补图中，每个顶点的度数不小于$n-3$。问题实际上就是在原图的补图中找出m条边，满足给出的条件。

先考虑，在什么情况下问题会无解。我们考虑一个较强的约束：求出的新图应为一个哈密尔顿回路。对于哈密尔顿回路的存在性，我们有如下定理：

定理6.1 (Ore's Theorem)**.** 对于一个含有n个顶点的无向图，其中存在哈密尔顿回路的充分条件是，对于任意两个不相邻的顶点u和v，满足

$$\deg u + \deg v \geq n$$

其中$\deg u$代表顶点u的度数。

由上面的分析可知，$\deg u \geq n-3$对于每个顶点都成立，故定理6.1的条件实际上就是

$$2(n-3) \geq n$$

也即$n \geq 6$。因此，对于任意$n \geq 6$的情况，问题始终有解。那么在下面的分析中，我们将只考虑$n \geq 6$的情况。

如果我们求出了原图的补图的一个哈密尔顿回路，那么我们只需在哈密尔顿回路上任选m条边作为新图即可，易知这样得出的新图满足题目的要求。现在问题就在于如何求出哈密尔顿回路了。

众所周知，在给定的图中求一个哈密尔顿回路是NPC问题，不存在高效的算法。而且我们似乎也不好设计一个可以按照设计思路1改造成Monte Carlo算法的确定性算法。这里我们介绍一个Las Vegas算法，并尝试求出其期望时间复杂度。

算法流程如下：

1. 随机一个$1 \sim n$的排列$p_1, \ldots, p_n$；
2. 对于$1 \leq i < n$，检查无向边(p_i, p_{i+1})是否存在于原图中，同时检查无向边(p_n, p_1)是否存在于原图中；
3. 如果检查的所有边都不存在于原图中，那么我们成功地找到了一个哈密尔顿回路；否则则返回第一步。

这个算法的步骤一和二的时间复杂度可以做到$O(n\log n)$或者$O(n)$，现在我们关注的就是期望意义下，算法需要运行几遍?

我们可以发现，这是一个由Monte Carlo算法改造而成的Las Vegas算法，因此计算期望复杂度的关键在于求出算法的正确率。不过对于这个算法而言，求出精确的正确率相当麻烦，因此我们考虑求出近似的正确率。

我们假设原图是一个长度为n的环。考虑逐一确定排列的每一位，令f_i^n表示，排列长度为n且确定了前i位时，尚未产生错误[13]的概率。特别地，我们直接用f^n表示f_n^n。由于只是近似，我们不考虑具体哪些点已经加入排列，只考虑一次选择中不产生错误的概率。

显然我们有$f_1^n = 1$，对于任意$i(i > 1)$，我们可以做如下的估计

$$\begin{aligned} f_i^n &= f_{i-1}^n \frac{1}{\binom{n-1}{2}} \left(\binom{i-2}{2} + (i-2)(n-i+1)\frac{n-i}{n-i+1} + \binom{n-i+1}{2}\frac{n-i-1}{n-i+1} \right) \\ &= \frac{n-3}{n-1} f_{i-1}^n \\ &= \left(\frac{n-3}{n-1}\right)^{i-1} \end{aligned}$$

结果十分简洁。那么对于f^n有

$$f^n = \left(\frac{n-3}{n-1}\right)^{n-1}$$

对其求极限可知

$$\lim_{n\to\infty} f^n = \frac{1}{e^2} \approx 0.135335$$

而当$n \geq 9$时，已经有$f^n \geq 0.1$。根据3.1.2小节中的分析可知，这个算法的期望运行次数$k = p^{-1} = (f^n)^{-1}$。对于足够大的n，我们可以认为$k \approx 10$，在运行时间上可以说是绰绰有余。至于$n < 9$的情况，我们可以直接用暴力算法求解。 □

由此我们总结出本文的最后一个设计思路：

算法设计思路3

我们尝试从上一道题中总结一个相对通用的思路。对于一些操作上类似的题，都可以使用下面的思路设计一个可行的Las Vegas算法：

[13]此处的“产生错误”即发现了一条属于原图的边。

1. 设计一个能解决问题的确定性算法。
    - 这个算法需要枚举所有元素的一个排列；
    - 通常能够使用这个思路解决的问题，要么是只求一个可行方案而不要求最优，要么是最优方案特别多。总之，需要保证“有用”的排列个数不会太少。
2. 向算法中引入随机。将算法枚举排列的部分改为随机一个排列。
    - 这一部分的分析往往是问题的关键。正如你在上一道题中所看到的一样，这一类随机算法的复杂度分析一般比较复杂，通常情况下无法做出精确的计算；
    - 如果你在比赛的时候想到了这样一个算法，与其花时间给出严格的证明，不如通过实践来检验真理；
    - 值得一提的是，可以通过限制运行次数（俗称“卡时”）来将这个算法改造成Monte Carlo算法。对于根据解的优劣程度给部分分的题目来说，即使算法正确率不高也可以出奇迹；
    - 或许对于有些题目而言，这样的算法真的很不靠谱。但如果你坚信这道题的数据很不好出，而且一时半会找不到别的更好的确定性算法，不妨就用这个方法。 □

5 总结

通过对两类算法的简要介绍以及对五道例题的分析，可以看出，随机化算法可以运用在各种类型的题目中。不过，随机化算法本身涵盖的范围就很广，因而不存在一些通用的处理问题的方法，只能具体问题具体分析。

随机化算法相比起确定性算法有如下的优势：

- 不需要对问题的性质做过多分析；
- 编程复杂度较低；
- 一些情况下，可以达到更优的时间复杂度；
- 对于有些题目而言，随机化算法是唯一的正确算法。

当然，也有一些劣势：

- 需要比较严谨的复杂度与正确率的证明，有些时候证明会相当复杂；

- 并非所有问题都存在随机化的做法，大多数时候随机化只能作为获得部分分的工具。

本文针对具有不同操作的一些随机化算法进行了分类，并提出了3个相对通用的设计算法的思路。当然这些并不是设计随机化算法仅有的几个思路。我们在面对需要设计随机化算法的题目时，应当发挥创造新思维，并力求严谨。希望本文能对大家有所启发。

6 致谢

感谢中国计算机学会提供学习和交流的平台。
感谢我的教练李淑平老师对我的指导与关心。
感谢父母对我学习信息学竞赛的支持与鼓励。
感谢一路陪我走来的许许多多的同学的帮助。

参考文献

[1] Wikipedia. “*Randomized algorithm*”.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms (Second Edition)*. Massachusetts, USA: The MIT Press.

[3] Motwani Rajeev (1995). *Randomized algorithms*. Cambridge, UK: Cambridge University Press.

精细地实现程序——浅谈OI竞赛中的常数优化

天津南开中学 何琦

摘 要

程序的运行时间主要由算法复杂度决定，算法复杂度描述了随着数据规模增大时程序运行时间的变化情况。程序常数指的是乘在瓶颈复杂度前的系数，不随数据规模增大而变化，对程序运行时间的影响相对较小。然而，对程序常数的优化，在相同数据规模的情况下可以导致程序运行时间成倍缩短，从而使得程序更加高效。本文主要介绍OI竞赛中对算法复杂度并无影响，但对程序运行时间有较为明显影响的常数处理问题，涉及相同复杂度的算法选择，实现细节优化等方面。

1 同一类型的算法常数比较与选择

对于同一道题目，往往存在多种不同算法，它们的时间复杂度都是最优或很优的，但它们的实现方式不尽相同，从而程序常数也有所不同。由于常数受实现方式和细节影响较大，以下我们采用部分不会变化的算法骨架作为常数的比较方式，会与实际情况有所出入。

1.1 复杂度相同的算法

以树状数组，线段树，平衡树等为例进行比较。复杂度均为为$O(n\log n)$。

1.1.1 使用2-分治的理由

注意到描述复杂度时我们叙述的是$O(n\log n)$而不是$O(n\log_2 n)$，这是因为对于任意常数x,y，

$$\frac{n\log_x n}{n\log_y n}=\log_x y=C$$

由此，底数不影响算法复杂度，但会对常数产生影响。

通常情况下，分治型数据结构通常采用二分，此时合并常数可以看作2，而若采用三分，合并常数为3，而$\log_2 3 > \frac{3}{2}$，似乎采用三分常数更小一些。

然而，事实上三分实现效果不如二分好，这是因为在计算均分位置时，除法运算占了瓶颈，若采p用三分，需要执行两次÷3操作，而由于操作系统为二进制，二分时只需要采用位运算即可。从而，二分常数实际上远小于三分

同时，平衡树系列数据结构也均采用二分作为旋转操作的基础，若采用三分，单是从分析角度就已经无法承受。

1.1.2 线段树

通常情况下，线段树采用自顶向下的实现方式，并可以支持动态建点，可持久化等操作。

为了通过位运算加速寻找子结点的过程，通常编号x的两个子结点分别为$2x$和$2x + 1$。

这种线段树对单点进行操作时，每层只会访问到一个节点，由于需要递归返回，常数可以看作2。而对区间进行操作时，每层最多访问4个节点，最多提取两个节点，我们以访问节点数量为常数，则这种线段树常数为4。

而一种自底向上线段树[1]，将整个数据段补齐至长度$2^k - 2$，并按照之前的编号规则，从$2^k + 1$位置填入底层数据，并向上合并。对区间进行操作时，转为开区间，直接取出底层的两个节点向上递归。

这种数据结构的优势在于，如果没有修改型懒标记的存在，每一层只需要考虑被提取的两个节点，常数为2。而如果有修改型懒标记的存在，就必须考虑到开区间的两个点的懒标记，常数增大为4。

可以看到，使用这种线段树，没有懒标记时比之前的自顶向下线段树快了一倍，究其原因则是不必自顶向下查找所需的节点位置，省去了前一半的时间。

然而，即便有修改型懒标记，这种线段树依然比自顶向下的线段树要快。这则是因为我们分析复杂度时只考虑了访问节点数量，而没有考虑自顶向下时进行的包含判断。p 综合考虑，自底向上线段树常数较优。

[1]最早由清华大学的张昆伟提出，参见参考文献[1]

而自底向上线段树的局限性在于不能处理“无法直接找到底端节点”的题目，例如在线段树上二分。所以，在允许的情况下，使用自底向上线段树效果更佳。

1.1.3 树状数组

树状数组结构与二项树较为类似，但修改和查询方式比较奇怪，也没有明显的“分治”算法的特征，实现效果通常比线段树好。现在仔细分析一下它的原理。

考虑大小为2^{k+1}的树状数组，实质是由两个大小为2^k的树状数组，其中一个的根接到另一个的根上所形成的。

再考虑线段树，大小为2^{k+1}的线段树是由两个大小为2^k的树状数组，都将根接到一个新节点上形成的。

从而，每次合并时树状数组都比线段树少用了一个节点，而少的那个节点，实质是右子树的根

实际上我们注意到，树状数组支持的操作是查询区间[1, x]的值，若包含某个右区间，则一定包含整个区间，只需调用父亲节点即可，所以右节点其实永远不会被用到。

树状数组修改时的位运算，作用是跳到它的最近一个存在的父亲，而查询时的位运算，作用是跳到它父亲的左兄弟以继续查询。

整个算法比线段树节约了一半的空间，但由于省去的节点不均匀，最坏常数仍是1 (这里依然是以访问节点数量为常数，对修改而言最坏情况是修改1号节点，查询而言最坏情况是查询[1, $2^k - 1$])

而树状数组对随机数据的期望常数为0.5，较线段树而言更优一些。

1.1.4 平衡树

平衡树的实现方式多种多样，大致分为基于旋转的平衡树和基于重构的平衡树两种。这里，我们以访问的节点数量(可以近似看做深度)+修改的指针数量定义常数。

基于旋转的平衡树，若记录父亲，每次旋转重构6个指针。

AVL深度常数为1，每平衡一层需要旋转最多两次，常数可以看作$2 \times 6 + 1 =$

12。

splay势能分析中自带3的常数，每1单位势能支持旋转1次，访问次数和旋转次数相同，常数可以看作21。

基于旋转的treap，其旋转次数有一个期望的常数界[2]，所以瓶颈在于深度，而treap的实际深度常数约在1.5到2之间。因而看起来非常优秀。实际上，由于非瓶颈部分常数较大(第二小节会描述这部分的影响)，且需要处理随机数发生器的问题(第二节会讲述)，实现效果并没有预期的好，但这仍是一个常数不大的优秀算法。

基于重构的平衡树，不必记录父亲指针，重构常数为2×重构大小。

替罪羊树，以平衡因子0.7计，深度常数大致为$\log_{\frac{10}{7}} 2$，重构常数约为$2\log_{1.2} 2$，势能总常数=深度常数×重构常数，约为16。

基于重构的treap，每次重构期望大小为深度，常数较基于旋转的treap增大2，一般只有不支持旋转时才会使用。

特别的，即便按照这种方式计算常数，与实际运行时间的比例差距还是很大，因为访问和修改指针并不是平衡树的全部操作。

而我们可以看到，平衡树系列算法常数都不小，其中属splay算法常数最大。因此，可行时最好采用预留位置的静态线段树代替平衡树。

1.2 复杂度相近的算法

有些情况下，同一个问题的不同算法中，有一些算法，它们的复杂度不是最优的，但与最优复杂度的算法相近，但常数较小，使得在现有能承受的数据规模下实际效果可以与最优算法相比。

1.2.1 堆

堆的实现有很多种，应用比较广的是数组实现的二叉堆，需要支持合并时常用的有左偏树，二项堆，还有一种时间复杂度较优的算法“斐波那契堆”。

这里同样采用访问节点数量和修改指针数量来描述常数，其中数组实现的

[2]期望旋转不超过2次，见参考文献[2]

二叉堆只需要用到前一部分。

数组实现的二叉堆，时间复杂度除Min之外全部为$O(\log n)$，但由于其采用了和线段树类似的父亲——儿子对应关系(x号节点子结点为2x和2x+1)，从而通过位运算大大加速，常数只有1。

左偏树，支持合并的二叉堆，时间复杂度与数组实现的二叉堆基本相同。由于有交换儿子操作，常数增大为2。而虽然不平衡，但操作只涉及较浅一侧的深度(DecreaseKey除外)，深度常数为1。

二项堆，记录Min时复杂度与之前的算法相同，同样采用指针记录，采用左儿子右兄弟结构，相同大小二项树合并时常数为2，最坏情况下需要将两个大小为$2^k - 1$的二项堆合并，需合并$2\log n$次，总常数为4。在平均情况下只需涉及$\log n$棵二项树的合并(类似于树状数组)，期望常数为2。

斐波那契堆除Extract-Min和DeleteMin外复杂度均为O(1)[3]，其中每一次合并或分割带来的双向链表修改常数为4，而度数常数为$\log_{\frac{1+\sqrt{5}}{2}} 2$，从而，整个算法常数约为5.5。

综上，斐波那契堆的优越性是不可否认的，但在实际应用中，insert和extract通常差距不大，这导致另外几种实现甚至比它更快，因此通常我们使用前三种堆算法，而只有当insert比extract系列操作多达一定数量级时，才使用斐波那契堆。

1.2.2 后缀数组

后缀数组的构造分为倍增算法和DC3算法。由于这部分算法主要用到基数排序，而基数排序又由循环构成，因此以循环数量定义常数。(一次基数排序为2 个长度为n的循环和2个长度为c的循环)。

倍增算法，倍增内包含两个基数排序(常数4)+ 新rank生成(常数2)，最坏情况下要倍增全部$\log n$次才能出解，复杂度$O(n \log n)$，常数10。后续的RMQ运用经典的$O(n \log n) - O(1)$算法，预处理常数为1，查询常数为一个log运算的复杂度。

DC3算法，递归常数3，每轮递归前三关键字排序合并常数$3\times2\times(1+\frac{2}{3}) = 10$，递归后一次双关键字排序常数$2 \times 2 \times (1 + \frac{1}{3}) = \frac{16}{3}$，归并常数3，总复杂度O(n)，常数高达55。同时，后续RMQ需要采用标准的O(n)-O(1)算法，因为过于繁琐难

[3]见参考文献[2]

以适应之前的常数定义，不予计算。

后缀数组构造的两种算法复杂度相差一个log，而两种算法常数都不小，实际运行中，DC3算法略胜一筹，但差距不大。

1.2.3 树上链系操作

对树上的某条链进行操作，通常可以采用树链剖分和动态树两种做法。[4]

这两种做法的骨架不同，所以常数单位也不同，只能大致估算，偏离实际情况较远。

动态树分析复杂度与splay类似，常数和splay 相同，以21计。

树链剖分，找LCA复杂度$O(\log n)$常数为2，线段树复杂度$O(\log n)$常数为4，总复杂度$O(\log^2 n)$常数以8计。但是注意到两个log很难同时取到，最坏情况下为$1 + 2 + ... + \log_2 \frac{n}{2}$，有常数$\frac{1}{2}$，最坏情况下总常数为4，平均情况下甚至远小于1。

综上，虽然树链剖分复杂度多一个$\log n$，但实现效果与动态树不相上下。

2 实现常数优化

在同一种算法中，也有一些细节可以决定整个程序常数的大小。这部分主要介绍在算法实现过程中进行的常数优化。

2.1 整块中的无用部分

在一些算法中，需要用到数组和循环，但并不是数组的每个位置都有意义。我们可以借此优化算法。

2.1.1 动态规划与记忆化搜索

在高维动态规划中，常常有一些状态是不合法或没有意义的，我们可以通过不计算它们来优化算法常数。

例题：NOIP2008 传纸条方格中要求找出两条不相交的左上到右下的最短路

[4]2007年集训队杨哲曾提出过一种全局平衡二叉树，复杂度为$O(n \log n)$且常数较小，见参考文献[3]

径使得路径经过点权值之和最小。方格200×200。

最容易想到的方法：dp[x1][y1][x2][y2]表示两条路径分别走到(x1,y1)和(x2,y2)时的最小权值和。时间复杂度$O(n^4)$，本来无法通过全部数据。但是注意到不合法的状态其实有很多，采用记忆化搜索能够避开这些不合法状态，从而通过这题的全部数据。

实际上，只有到起点距离相同的两个节点才能成为dp状态，所以标程的做法为dp[距离][x1][x2]，时间复杂度为$O(n^3)$。而实际上记忆化搜索访问的合法状态也为$O(n^3)$，这个算法为我们省去了一些思考难度。

2.1.2 矩阵表示的变化量

在一些题目中，我们常常采用矩阵乘法或倍增的方式优化算法，而整个矩阵也并不是每个位置都有意义或计算的必要。

例题：NOI2013 Day2 P1 要求按一定顺序迭代y=ax+b和y=cx+d，求最终答案对p取模的值。

很容易想到利用矩阵描述迭代并倍增加速：

$$\begin{pmatrix} x & 1 \end{pmatrix} \times \begin{pmatrix} a & 0 \\ b & 1 \end{pmatrix} = \begin{pmatrix} y & 1 \end{pmatrix}$$

p而采用这个算法，矩阵乘法的常数为$2 \times 2 \times 2 = 8$次带模乘法运算，会超时。而注意到这个矩阵无论如何做乘法，右侧一列的0和1都不会变，所以我们可以把常数优化至2次带模乘法和2次带模加法，可以通过该题。

本题标算提供了这种算法[5]和一种采用等比数列模意义求和的算法。采用这种算法减少了思维难度。

2.2 自定义常数选择

在一些算法中，有一些自定义常数，它们对整个程序复杂度并无影响，但对常数有较大影响。

[5]命题人提到“若你采用这种算法，需要实现得非常精细”，这也是本文标题的来源。

2.2.1 分块算法

分块算法中，经常需要微调块的大小以减小常数。

例题：小Z的袜子给定一个长度为n的序列，m 个询问，从区间$[l_i, r_i]$中任选2个数，求相同的概率。

解法：区间端点移动1需要花费1的常数，按左端点分x块，每块按右端点排序后依次调整。左端点总移动长度$m \times n \div x$，右端点总移动长度$x \times n$。所以应当选择块数$x = \sqrt{m}$而不是通常情况下的$\sqrt{n}$。

实际测试表明，n=65536，m=262144时，选择512分块比选择256分块快了1.5倍。

2.2.2 树的分治

树上分治中比较通用的算法为基于重心二分。那么涉及两个常数问题：1、如何划分子树。2、分治到多大时应当转而使用暴力法。

常见划分算法：按任意顺序划分，过半时的那一棵子树划归到较小的一半。由于重心性质保证最大子树不超过总点数一半，这种方法最坏情况划分比例为$1:3(\frac{1}{4}+\frac{1}{2}+\frac{1}{4})$。

改进：按任意顺序划分，可分成三份，均不过半，此时将较小两份合做一半。显然合成的这一份不会超过$\frac{2}{3}$，而另一份不超过$\frac{1}{2}$，最坏情况下划分比例为1 : 2。而注意到有三棵相同大小子树的情况，不存在比1:2更优的算法，因此这种算法常数足够小。

分治大小下界：假设s是树的大小，首先一遍BFS求出size和重心，之后子树划分，两部分各BFS一遍求出各自的列表，最后合并，大约要进行6～8次遍历，并需要将子树按最坏1：2比例进行暴力。暴力时需要进行s 次遍历，则暴力优于分治的情况：$s^2 \le \left(\frac{s}{3}\right)^2 + (\frac{2s}{3})^2 + 8s$，解得$s \le 18$。

p实测结果：约20规模时暴力，程序效果最佳。

2.3 一些实现细节

实现代码时，养成注意细节的习惯，可以减小程序常数。

2.3.1 位运算

对布尔变量的整体操作，可以采用位运算加速，注意到即使是64位整型，位运算速度也是相当快的，这通常能够给程序带来$\frac{1}{64}$的常数优化，有时能使得算法能力提升一个数量级。

采用状态压缩动态规划时，尽可能采用2，4，8进制表示状态，因为通过位运算可以很快的取出需要用到的位，从而加速算法。若有多余状态，参考“整块中的无用部分”优化。

乘法和除法操作通常比加法和减法慢，于是将常用的×2，÷2转为位运算能够小幅提速。[6]

2.3.2 读入

仔细分析后我们发现，读入时需要将十进制转换为二进制，这导致读入时需要做$n \log C$次乘法，这在一些复杂度为$O(n)$或常数不大的$O(n \log n)$算法中尤其明显，注意读入一个包含10^6个整数的文件，即使使用scanf也需要1秒，输入流就更不用说了。

实际上，系统的读入函数由于考虑了全部可能出现的情况，所以确保稳定，但常数相对较大。竞赛中可以认为输入数据是符合题目叙述的，则可以忽略其中的一些情况，所以可以考虑自写读入代码，利用系统的读单个字符函数来完成读入。实现效果通常比系统读入快60 % 到80 % 。

2.3.3 随机数

系统的随机生成函数很慢，通常随机5×10^5次就需要大约1s。

自己写随机函数时，可以考虑线性迭代，二次迭代，多个函数轮流取值等方法，在基本不影响随机性的前提下尽可能减小随机函数的常数。

2.3.4 动态内存静态化

系统的内存分配函数多次调用时速度非常慢，对于主席树等动态内存开销较大的数据结构，使用动态内存可能会使得程序变慢10～20倍。解决方案是估

[6]注意运算符优先级!

计需要的内存量并提前开辟内存池，手工实现内存分配。若有必要，可以采用栈进行内存回收等操作。

2.3.5 内存访问连续性

这种问题通常在大规模矩阵乘法时体现，主要原因是数组的第一维连续变化导致调用内存位置变动较大，看以下一段代码：

```
1 for(int i=1; i<=n; ++i)
2     for(int j=1; j<=n; ++j)
3         for(int k=1; k<=n; ++k)
4             c[i][j]+=a[i][k]*b[k][j];
```

这段代码在内层的k循环时b数组的调用就出现了上述问题，导致该类型矩阵乘法在200×200规模的矩阵就达到了1s。改进方案：

```
1 for(int i=1; i<=n; ++i)
2     for(int j=1; j<=n; ++j)
3         for(int k=1; k<=n; ++k)
4             c[i][k]+=a[i][j]*b[j][k];
```

这段代码可以处理400×400规模矩阵的乘法。

结论：尽可能防止数组非最后一维的连续变动，如果需要，可以交换数组的两维。

3 一道经典题目的算法比较

题意：n个整数，和不超过C，现要将其分成两部分使得和尽可能接近，求最小差值。n和C规模相同。

3.1 算法一：01背包算法

$dp[i][j]$表示用前i个数能否凑出和为j的一部分。

$$dp[0][0] = 1$$

$$dp[i][j] = dp[i-1][j] | (j \geq c_i \ \&\& \ dp[i-1][j-c_i])$$

时间复杂度$O(nC)$，可以支持10^4规模的数据。

3.2 算法二：位运算优化

注意到dp值实质为布尔变量，采用bitset将每个dp值压至一个二进制位。

$$dp[0] = 1$$

$$dp[i] = dp[i-1] | (dp[i-1] << c_i)$$

时间复杂度依然为$O(nC)$，常数缩小为$\frac{1}{32}$至$\frac{1}{64}$，可以支持5×10^4甚至10^5规模的数据。

3.3 算法三：分块FFT

该算法由上海交通大学的郭晓旭提出。

n个数中，超过$\sqrt{C}$的数至多$\sqrt{C}$个。

不超过$\sqrt{C}$的数，可以分成至多$2\sqrt{C}$块，每块和不超过$\sqrt{C}$显然每一块的个数也不超过$\sqrt{C}$，应用算法1，可以在$O(C)$的时间计算出某一块可以凑出的数，则可以在$O(C\sqrt{C})$的时间内计算出所有块可以凑出的数。

试图合并两块时，设a_i和b_i分别表示两部分能否凑出i，c_i表示总体能否凑出i，则有

$$c_i = OR \ \ a_j \ \&\& \ b_{i-j} (0 \leq j \leq i)$$

将OR换成σ，则可以采用快速傅里叶变换(FFT)[7]在$O(ClogC)$时间内计算，总合并时间复杂度$O(C\sqrt{C}logC)$

之后，再将超过$\sqrt{C}$的部分按算法1转移，复杂度$O(C\sqrt{C})$总复杂度$O(C\sqrt{clogc})$[8]瓶颈在于快速傅里叶变换(FFT)。由于不能采用位运算优化，只能做到2^15左右的数据规模，实现效果甚至不如算法二。

[7]见参考文献[2]

[8]通过合理分配块的大小可以做到$O(C\sqrt{ClogC})$，快速傅里叶的常数也可进入根号下，但实现效果仍不理想。

3.4 算法四：分治FFT

注意到上述算法中应用了两部分的合并，而支持合并时我们可以采用分治法解决。

和为C的若干个数，一定可以分为三部分，其中两部分的和不超过C/2，而第三部分只有一个数。递归求解两部分，合并时采用大小为C的FFT，并暴力将第三部分那一个数用O(C)的时间进行转移。

复杂度计算应用主定理，$T(C) = 2T(\frac{C}{2}) + O(C\log C) + O(C)$，得$T(C) = O(n\log^2 n)$

从而，我们得到了一个复杂度相当优秀的算法。

瓶颈依然在于快速傅里叶变换，因此常数依然很大，实现效果没有算法二好。即使在高达$5*10^5$的数据规模时，对n=C的情况该算法用时6s，算法2用时15s，依然没有拉开差距。

3.5 算法五：多重背包

在n很大时，有相当一部分数是相同的。那么，我们可能可以采用多重背包的算法进行加速。

最坏情况下，n个不同的数的和最小为$1+2+...+n = O(n^2)$，于是我们得出：至多有$O(\sqrt{C})$个不同的数(常数为2)。

采用多重背包的经典算法，令$dp[i][j]$表示用前i种数凑出j时，第i种数最少的使用数量。-1表示不能凑出。

$$dp[i][j] = \begin{cases} 0 & dp[i-1][j]! = -1 \\ dp[i][j-c_i]+1 & j \geq c_i \&\& 0 \leq dp[i][j-c_i] \leq num_i \\ p-1 & else \end{cases}$$

复杂度$O(C\sqrt{C})$，由于不能压位，常数为3(转移)*2(数量常数)=6，实现效果可以达到10^5的数据规模。

3.6 算法六：改进01背包

多重背包的另一个处理方式是将同样大小的包按二进制拆分，使得原来的x个相同数变为至多$O(\log x)$个不同的数。于是我们成功将n的规模缩减

为$O(\sqrt{C}logC)$[9]

而我们注意到，只要一个数的个数超过2，我们就可以继续拆分。于是我们从小到大进行拆分，最终保证每个数的个数不超过2，则最多有$O(\sqrt{C})$个数。

p进行01背包，最终复杂度$O(C\sqrt{C})$，采用算法二的位运算优化，常数为$\frac{1}{16}$到$\frac{1}{32}$，可以做到10^6规模的数据。

3.7 小结

该题目的各种算法中，复杂度最优的为算法四(分治FFT)，仅为$O(n\log^2 n)$，而实际效果最优的为算法六，复杂度为$O(n\sqrt{n})$。这是因为两种算法的常数相差了将近200倍。可见，关注算法常数可以大幅提高程序效率。

4 总结

做一个注重常数的OIer。

参考文献

[1] 张昆伟，《统计的力量——线段树全接触》

[2] Thomas H.Cormen、Charles E.Leiserson等，《Introduction to Algorithms》

[3] 杨哲，《对QTREE解法的一些研究》

[4] 刘汝佳, 黄亮, 《算法艺术与信息学竞赛》

[9]这里的复杂度实际不可能达到，所以常数非常小，已经与接下来的算法差距不大。

-- 空 --

回归本源——位运算及其应用

镇海中学 沈洋

摘 要

在电子计算机中，位运算可以说是比四则运算更基本的运算，但由于它们不是数学的基本运算且不常在生活中应用，容易被部分选手，特别是初学信息学的选手忽视。本文希望通过对位运算及其应用的介绍，来使这一情况得到一定的改善。

本文大致可分为三个部分。第一部分（第1节）对位运算基本概念进行了介绍，第二部分（第2至4节）对位运算的几个应用进行了介绍，第三部分（第5节）展示了一些例题。

几点说明

- 约定二进制最高位为最左边，最低位为最右边，最低位记为第0位。
- 本文中的所有程序均以C++语言描述，所涉及的语言细节、内建函数等均以GNU C++（GCC编译器）为标准。
- 若无特别说明，本文中的位运算使用C++运算符表示。
- 本文中汇编指x86汇编。

1 基本运算

1.1 与、或、非和异或

对于逻辑运算的“与”、“或”、“非”和“异或”，想必大家都耳熟能详了。其中“与”（$\wedge$）、“或”（$\vee$）、“异或”（$\oplus$）的真值表如下：

p	q	$p \wedge q$	$p \vee q$	$p \oplus q$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

“非”（¬）的真值表如下：

p	$\neg p$
0	1
1	0

位运算的“与”、“或”、“非”和“异或”可以看作逻辑运算在整数上的扩展。它们的运算法则如下：

与 运算结果的每个二进制位的值等于两个运算数的对应二进制位进行逻辑“与”运算的结果。

或 运算结果的每个二进制位的值等于两个运算数的对应二进制位进行逻辑“或”运算的结果。

非 运算结果的每个二进制位的值等于运算数的对应二进制位进行逻辑“非”运算的结果。

异或 运算结果的每个二进制位的值等于两个运算数的对应二进制位进行逻辑“异或”运算的结果。

这四种位运算对应的汇编指令如下：

与	或	非	异或
and	or	not	xor

由此可以看出，对于这四种运算，带符号整数和无符号整数并没有对应不同的汇编指令，它们的行为是相同的。也就是说，对于带符号整数，符号位也会与其它位按照同样的方式处理。

1.2 移位运算

移位运算分为逻辑移位、算数移位、循环移位、带进位循环移位四种。其中带进位循环移位涉及到CF标志位，因此不在我们的讨论范围内。为了准确区分各种移位运算，在这一部分中我们使用汇编指令来表示它们。

当我们说将x左移或右移y位时，默认y非负且小于x的位宽[1]。例如对于32位整数，y只能在0到31之间取值。

逻辑移位

逻辑左移对应的汇编指令为shl x y，它的功能是把x的每个二进制位向左移动y位，移动造成的最右边的空位由0补足，最左边的数溢出[2]。

逻辑右移与逻辑左移完全相反，它对应的汇编指令为shr x y，它的功能是把x 的每个二进制位向右移动y位，移动造成的最左边的空位由0 补足，最右边的数溢出。

算术移位

算术左移的汇编指令为sal x y，它与逻辑左移完全相同。

算术右移的汇编指令为sar x y，它与逻辑右移大体相同，唯一的区别在于移动造成的最左边的空位由符号位（最高位）补足而不是由0 补足。

循环移位

循环左移对应的汇编指令为rol x y，它的功能是把x的每个二进制位向左移动y位，移动造成的最右边的空位由最左边溢出的位补足。

循环右移与循环左移完全相反，它对应的汇编指令为ror x y，它的功能是把x 的每个二进制位向右移动y位，移动造成的最左边的空位由最右边溢出的位补足。

移位运算的数学意义

在逻辑移位、算术移位这两个看似复杂的定义背后实际上是有其数学含义的。逻辑移位被设计用于处理无符号整数，它的左移和右移分别与无符号整数的$x \times 2^y$ 和$x \div 2^y$ 具有相同的效果；算术移位被设计用于处理带符号整数，它的

[1]若超出这个范围，汇编指令只考虑y的后几位，C语言将产生未定义行为。

[2]实际上，最后一个溢出的位会被放入标志位CF中，其他标志位也会相应发生变化，但这不在我们的讨论范围内，故在本文中略去对这些部分的描述。

左移和右移分别与带符号整数的$x \times 2^y$和$x \div 2^y$具有相同的效果（舍入方式为向下舍入）。这一点根据补码的定义比较容易证明，在此就不赘述了。

C++中的移位运算

在C++中，无符号整数的移位使用逻辑移位，有符号整数的移位使用算术移位。这样一来，无论是无符号整数还是带符号整数，我们都可以放心的使用左移和右移来代替乘以二的幂或除以二的幂的操作。

C++中没有专门的对带符号整数进行逻辑右移的运算符，不过我们可以通过强制类型转换将它转换成无符号整数后再进行运算。

C++中没有提供循环移位操作符，但我们可以通过其他运算的组合来实现它。例如对于32位无符号整数x，表达式`(x << y) | (x >> (32 - y))`可以实现循环左移的功能，而`(x >> y) | (x << (32 - y))`则可以实现循环右移的功能。

2 二进制位的修改和查询

借由汇编指令、内建函数以及基本位运算的组合，我们可以在二进制位的层面上对整数进行一些修改和查询。本节将介绍一些常用操作并给出一些易于理解和扩展的实现[3]。由于带符号整数的性质较为复杂，在这里我们只讨论无符号整数的相关操作，并以常用的32位无符号整数为例。

2.1 读某些位

读取x的第pos个二进制位是一个很常用的操作，它的实现方式是先将x右移pos位，使要读取的位移到最低位，再通过 `& 1` 将其取出。代码如下[4]：

```
bool readBit(u32 x, int pos) {
    return (x >> pos) & 1;
}
```

[3]一些“黑技术”如利用64位指令处理32位操作数、利用浮点数进行计算等并未涉及。对此感兴趣的读者可以参考http://graphics.stanford.edu/%7Eseander/bithacks.html

[4]为简化代码，我们使用u32来表示32位无符号整数(unsigned int)。

有时候我们会将多个整数“打包”在一个整数中，一个典型的应用是将四个字节打包为一个32位整型，那么读取的时候就需要形如“读x的第pos 位开始的cnt位”这样的操作。它的实现方式与上面别无二致：首先将x右移pos位，再通过`& mask` 来取出最后cnt位，其中`mask`的后cnt个位为1，其余位为0，可以由`(1 << cnt) - 1`得到。代码如下：

```
u32 readBits(u32 x, int pos, int cnt) {
    return (x >> pos) & ((1u << cnt) - 1);
}
```

通过上面两个例子，不难发现读取某个或某些二进制位的关键实际上是与运算。通过将原数和一个遮罩进行与运算，可以达到保留指定一些位（将遮罩的对应位设为1），清零其它位（遮罩的对应位设为0）的目的。

2.2 改某些位

对二进制位的修改实际上是“与”、“或”和“异或”运算的实际应用。

将某些位置为1

要实现这个功能，我们的办法是将原数与一个遮罩进行或运算。对于要改为1 的位，我们将遮罩的对应位设为1，否则将对应位设为0，这样，原数与遮罩进行或运算的结果就是答案。举例来说，将x的第pos 位置为1的代码如下：

```
u32 setBit(u32 x, int pos) {
    return x | (1u << pos);
}
```

将某些位置为0

这个操作与上一个操作正好相反，这一次我们要利用的是与运算。我们构造一个遮罩，对于要修改的位，我们将遮罩的对应位设为0，否则将其设为1（注意，这与上一个操作中的遮罩完全相反），随后将原数与这个遮罩进行与运算即可得到答案。将x的第pos位置为0 的代码如下：

```
u32 clearBit(u32 x, int pos) {
    return x & ~(1u << pos);
}
```

取反某些位

这一次是异或运算的应用。同样构造一个遮罩，如果我们要取反某位，则将遮罩的对应位设为1，否则将其设为0，随后将原数与这个遮罩进行异或运算即可得到答案。将x的第pos位取反的代码如下：

```
u32 flipBit(u32 x, int pos) {
    return x ^ (1u << pos);
}
```

2.3 求1的个数

分治法

求二进制位中1的个数与求各个二进制位的和是等价的，因此我们转而思考如何求各个二进制位的和。我们可以使用分治来解决这个问题：每次将整个数分成两个部分，分别求出每个部分的和，再将它们相加。利用位运算，我们可以并行地完成每一层的工作，像下面这样：

```
int bitCount_1(u32 x) {
    x = ((x & 0xAAAAAAAAu) >>  1) + (x & 0x55555555u);
    x = ((x & 0xCCCCCCCCu) >>  2) + (x & 0x33333333u);
    x = ((x & 0xF0F0F0F0u) >>  4) + (x & 0x0F0F0F0Fu);
    x = ((x & 0xFF00FF00u) >>  8) + (x & 0x00FF00FFu);
    x = ((x & 0xFFFF0000u) >> 16) + (x & 0x0000FFFFu);
    return x;
}
```

它是这样工作的：

- 第一步，有32个项需要相加，每一项占1 bit。我们将其中的奇数项和偶数项分别取出来（利用2.1 小节中提到的方法），并将奇数项右移1位和偶数项“对齐”，然后将他们相加。这一步过后我们实际上将16对1 bit的项分别相加，并将结果存放在了原来这两项所在的2 bit空间上。

- 第二步，有16个项需要相加，每一项占2 bit。我们将奇偶项分别相加，形成8个4 bit的项。

  ……

- 第五步，有两项需要相加，每一项占16 bit。将这两项相加我们便得到了答案。

那么，还能再优化么？答案是肯定的。看下面这段代码：

```
int bitCount_2(u32 x) {
    x-= ((x & 0xAAAAAAAAu) >> 1);
    x = ((x & 0xCCCCCCCCu) >> 2) + (x & 0x33333333u);
    x = ((x >>  4) + x) & 0x0F0F0F0Fu;
    x = ((x >>  8) + x) & 0x00FF00FFu;
    x = ((x >> 16) + x) & 0x0000FFFFu;
    return x;
}
```

这段代码较上一段的第一个差别是第一步的实现。我们知道第一步的作用是将奇数位和偶数位相加，根据第一个程序它的结果应为：

```
((x & 0xAAAAAAAAu) >> 1) + (x & 0x55555555u)
```

而x的值等于：

```
((x & 0xAAAAAAAAu) + (x & 0x55555555u)
```

将两式作差得到：

```
(x & 0xAAAAAAAAu) >> 1
```

因而将x减去`(x & 0xAAAAAAAAu) >> 1`便可以得到我们所求的结果。

另一个差别是第三步之后的实现，我们以第三步为例来说明。第三步的作用是将4对4 bit整数相加，但实际上此时每个整数最大只可能是4，这就表示，即使是两个数相加的结果也能在4 bit的空间存下。因此`(x >> 4) + x`可以正

确地依次求出第0个数加第1个数，第1个数加第2个数，第2个数加第3个数……我们从中取出我们需要的结果即可。

还能继续优化么？答案依然是肯定的：

```
int bitCount_3(u32 x) {
    x-= ((x & 0xAAAAAAAAu) >> 1);
    x = ((x & 0xCCCCCCCCu) >> 2) + (x & 0x33333333u);
    x = ((x >> 4) + x) & 0x0F0F0F0Fu;
    x = (x * 0x01010101u) >> 24;
    return x;
}
```

这段代码与上一段的不同点在于，上一段代码的最后两步操作被语句：

```
x = (x * 0x01010101u) >> 24
```

代替了。我们知道上一段代码中第四、第五步的作用是将x中的4个8 bit数相加，那么，如果我们令：

```
x = (a<<24)+(b<<16)+(c<<8)+(d<<0)
```

则我们所求的答案便是：

```
a + b + c + d
```

我们来看看计算x * 0x01010101u时发生了什么：

```
  x * 0x01010101
=(x<<24)+(x<<16)+(x<<8)+(x<<0)
=(a<<48)+(b<<40)+(c<<32)+(d<<24)+
         (a<<40)+(b<<32)+(c<<24)+(d<<16)+
                 (a<<32)+(b<<24)+(c<<16)+(d<< 8)+
                         (a<<24)+(b<<16)+(c<< 8)+(d<< 0)
=((a)       << 48) +
 ((a+b)     << 40) +
 ((a+b+c)   << 32) +
 ((a+b+c+d)<< 24) +
 ((b+c+d)   << 16) +
 ((c+d)     << 8 ) +
 ((d)       << 0 )
```

由于x是32位整数，因此上式的前三项应当溢出，于是x * 0x01010101的

结果为((a+b+c+d)<<24) + ((b+c+d)<<16) + ((c+d)<<8) + (d)考虑到a,b,c的实际意义，它们的值都不会超过8，因此上式的后三项的值必定小于1 << 24，也就是说它们不会对结果的最高8位造成任何影响。因此，x * 0x01010101的结果的高8位便是我们所求的a+b+c+d，我们将它右移24位即可得到答案。

查表法

如果我们要求0到n中每一个数的答案的话，那么可以使用如下递推：

```
f(0) = 0,
f(i) = f(i >> 1) + (i & 1)
```

显然对于所有32位整数保存答案（至少在目前看来）是一件不太现实的事情，不过好在这个问题是可以分割的，我们可以预处理所有16位整数的答案，在询问时将被询问的整数拆成高16位和低16位分别计算答案。于是预处理的代码如下：

```
int cnt_tbl[65537];
void bitCountPre() {
    cnt_tbl[0] = 0;
    for (int i=1; i<65536; ++i)
        cnt_tbl[i] = cnt_tbl[i >> 1] + (i & 1);
}
```

回答询问的代码如下：

```
int bitCount_4(u32 x) {
    return cnt_tbl[x >> 16] + cnt_tbl[x & 65535u];
}
```

内建函数

GCC中提供了如下内建函数[5]来实现这个功能：

```
int __builtin_popcount (unsigned int x);
```

对于支持SSE4.2的机器，如果在编译时开启相应开关，则该函数会被翻译成汇编指令popcnt，否则该函数使用类似上面“查表法”的方法进行计算。

2.4 翻转位序

对于一个32位整数来说，翻转位序是指将它的第0位与第31位交换，第1位与第30位交换，……，第i位与第31 – i位交换，……，第15 位与第16 位交换。例如对于32 位整数5，对它进行翻转位序将得到2684354560。

分治法

我们可以采用分治法来解决这个问题：首先将整个数分割成两个部分，分别翻转这两个部分，再将这两个部分对调。同样地，借由位运算，我们可以把每一层的工作并行完成，实现如下：

```
u32 bitRev_1(u32 x) {
    x = ((x & 0xAAAAAAAAu) >>  1) | ((x & 0x55555555u) <<  1);
    x = ((x & 0xCCCCCCCCu) >>  2) | ((x & 0x33333333u) <<  2);
    x = ((x & 0xF0F0F0F0u) >>  4) | ((x & 0x0F0F0F0Fu) <<  4);
    x = ((x & 0xFF00FF00u) >>  8) | ((x & 0x00FF00FFu) <<  8);
    x = ((x & 0xFFFF0000u) >> 16) | ((x & 0x0000FFFFu) << 16);
    return x;
}
```

这段代码是这样工作的：

- 第一步，交换相邻两位。这样就形成了16个长度为2的已经翻转的组。
- 第二步，每两位为一组，交换相邻两组。这就将16个长度为2的组变成了8个长度为4的已翻转的组。

[5]关于本文中提到的GCC内建函数的具体说明及其他相关内建函数，请参阅GCC手册：http://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html

……

- 第五步，每16位为一组，交换相邻两组。整个数翻转完成。

查表法

我们首先预处理出所有16位整数翻转后的结果，这可以由如下递推得到：

```
f(0) = 0,
f(i) = (f(i >> 1) >> 1) | ((i & 1) << 15)
```

接下来，我们将待翻转的32位整数拆成两个16位整数，通过查表来得到这两个16位整数的答案，再将它们对调便可以完成整个32位整数的翻转。预处理的代码如下：

```
u32 rev_tbl[65537];
void bitRevPre() {
    rev_tbl[0] = 0;
    for (int i=1; i<65536; ++i)
        rev_tbl[i] = (rev_tbl[i>>1]>>1) | ((i&1)<<15);
}
```

查询的代码如下：

```
u32 bitRev_2(u32 x) {
    return (rev_tbl[x & 65535] << 16) | rev_tbl[x >> 16];
}
```

2.5 求前缀/后缀0的个数

二分法

求前缀0与求后缀0是类似的，我们不妨以求前缀0为例。首先给出代码：

```
int countLeadingZeros_1(u32 x) {
    int ans = 0;
    if (x >> 16) x >>= 16; else ans |= 16;
    if (x >>  8) x >>=  8; else ans |=  8;
    if (x >>  4) x >>=  4; else ans |=  4;
    if (x >>  2) x >>=  2; else ans |=  2;
    if (x >>  1) x >>=  1; else ans |=  1;
    return ans + !x;;
}
```

这段代码的实质是二分查找。它是这样工作的：

- 第一步，判断高16位是否为空。若是，则高16位必然均为前缀0，我们给答案加上16，并在第0至15位上继续二分；若不是，则前缀0只出现在则高16位上，我们将它们右移到低16位上以便对它继续二分。
- 第二步，判断此时的高8位是否均为前缀0，并选择一边继续二分下去。

  ……

- 第六步，判断仅剩的那一位是否为0，若是，则给答案加1。

求后缀0个数的方式与此别无二致，因此根据上一段代码炮制一段求后缀0个数的代码并不是件难事：

```
int countTrailingZeros_1(u32 x) {
    int ans = 0;
    if (!(x & 65535u)) x >>= 16, ans |= 16;
    if (!(x & 255u  )) x >>=  8, ans |=  8;
    if (!(x & 15u   )) x >>=  4, ans |=  4;
    if (!(x & 3u    )) x >>=  2, ans |=  2;
    if (!(x & 1u    )) x >>=  1, ans |=  1;
    return ans + !x;
}
```

这段代码的工作原理与上一段基本相同，就不再赘述了。

查表法

我们依然以前缀0个数为例。这个问题与同样是可以分割的。我们可以先确定第一个1出现在高16位中还是低16位中，再通过查表来得到这个16 位整数的答案。这张表可以通过如下递推得到：

```
    f(0) = 16,
    f(i) = f(i >> 1) - 1
```

于是我们就不难写出预处理的代码：

```
int clz_tbl[65537];
void countLeadingZerosPre() {
    clz_tbl[0] = 16;
    for (int i=1; i<65536; ++i)
        clz_tbl[i] = clz_tbl[i >> 1] - 1;
}
```

以及查询的代码：

```
int countLeadingZeros_2(u32 x) {
    if (x >> 16)
        return clz_tbl[x >> 16]; else
        return clz_tbl[x & 65535u] + 16;
}
```

求后缀0个数也可以通过几乎相同的方法实现。

内建函数

对于这两个问题，GCC也提供相应的内建函数。

求前缀0的内建函数为：

```
    int __builtin_clz (unsigned int x);
```

它对应的汇编指令为bsr (Bit Scan Reverse)外加一个异或。

求后缀0的内建函数为：

```
    int __builtin_ctz (unsigned int x);
```

它对应的汇编指令为bsf (Bit Scan Forward)。

需要注意的是，当参数x为0时，这两个函数的行为是未定义的。

2.6 求第k个1的位置

这个问题可以转化为求最大的w，使得第0位到第$w-1$中1的个数小于k。这显然是可以二分的。于是，借助2.3小节中的cnt_tbl，我们便能实现一个简单的二分：

```
int kthBit_1(u32 x, int k) {
    int ans = 0;
    if (cnt_tbl[x & 65535u] < k)
        k -= cnt_tbl[x & 65535u], ans |=16, x >>=16;
    if (cnt_tbl[x & 255u  ] < k)
        k -= cnt_tbl[x & 255u  ], ans |= 8, x >>= 8;
    if (cnt_tbl[x & 15u   ] < k)
        k -= cnt_tbl[x & 15u   ], ans |= 4, x >>= 4;
    if (cnt_tbl[x & 3u    ] < k)
        k -= cnt_tbl[x & 3u    ], ans |= 2, x >>= 2;
    if (cnt_tbl[x & 1u    ] < k)
        k -= cnt_tbl[x & 1u    ], ans |= 1, x >>= 1;
    return ans;
}
```

不借助cnt_tbl自然也是可以的。我们可以发现，我们在二分的过程中所需要的区间和都是2.3小节中分治法计算过的区间，因此我们可以将分治的中间值存下来供二分时使用，像下面这样：

```
int kthBit_2(u32 x, int k) {
    int s[5], ans = 0, t;
    s[0] = x;
    s[1] = x - ((x & 0xAAAAAAAAu) >> 1);
    s[2] = ((s[1] & 0xCCCCCCCCu) >> 2) + (s[1] & 0x33333333u);
    s[3] = ((s[2] >>  4) + s[2]) & 0x0F0F0F0Fu;
    s[4] = ((s[3] >>  8) + s[3]) & 0x00FF00FFu;
    t = s[4] & 65535u;
    if (t < k) k -= t, ans |=16, x >>=16;
    t = (s[3] >> ans) & 255u;
    if (t < k) k -= t, ans |= 8, x >>= 8;
    t = (s[2] >> ans) & 15u;
    if (t < k) k -= t, ans |= 4, x >>= 4;
    t = (s[1] >> ans) & 3u;
    if (t < k) k -= t, ans |= 2, x >>= 2;
    t = (s[0] >> ans) & 1u;
    if (t < k) k -= t, ans |= 1, x >>= 1;
    return ans;
}
```

2.7 提取末尾连续的1

我们知道对x加1之后，最右边连续的1会变为0，最右边的0则变为1，而其它位不变。利用这一点我们可以通过下面这个公式来提取x末尾连续的1：

```
    x & (x ^ (x + 1))
```

2.8 提取lowbit

正整数x的lowbit是指x在二进制下最右边一个1开始至最低位的那部分，记为lowbit(x)。例如28 (11100B)[6]的lowbit 为4 (100B)，16 (10000B) 的lowbit则为16 (10000B)。lowbit的典型应用是树状数组和遍历集合。

有如下三个公式能帮我们求解lowbit：

```
lowbit(x) = x & (x ^ (x - 1))
lowbit(x) = x ^ (x & (x - 1))
lowbit(x) = x & -x
```

前两个公式利用了“将x减去1后，x中最右侧的1变为0，其之后的0变为1”这一点，第三个公式则主要利用了补码的特性。读者可以自行完成这三个公式的证明。

2.9 遍历所有1

一般情况下，我们可以通过不断求lowbit并将它从原数中删去（通过异或）来遍历每一个1。当需要用到这个1所在的位置时，我们可以通过2.5小节中介绍的求后缀0个数的方法来获知。

3 将整数用作集合

集合的种类有很多，但将它们的元素标号之后，都能映射到整数集合上。因此在这里我们只讨论全集为$[0,n)$的整数集合。

二进制的每个位有0和1两种状态，这正好可以对应集合中某个元素是否存在，因此我们可以用二进制数来表示集合——如果某位为1就表示对应元素存在于这个集合中，否则不存在。然而计算机中单个二进制数的位数是有限的，不妨设其能处理的单个二进制数的最大位宽为w，那么，表示上述全集为$[0,n)$的整数集合就需要至少$\lceil n/w \rceil$个二进制数。我们把每个二进制数叫做一“块”，并规定第i块记录第wi至$w(i+1)-1$这个w 个数是否存在（$0 \le i < \lceil n/w \rceil$，最后一块略有不同）。这样，我们便得到了一个维护集合的数据结构——bitset。我们将某个bitset能表示的全集的大小称为该bitset的大小。

[6]我们用后缀B来表示一个二进制数。

从本质上来说，bitset实际上是压w位的二进制高精度整数，因此也可以进行与、或、非、异或、左移、右移这些位运算，甚至可以进行加减乘除。

3.1 加入某个元素

首先计算出被加入元素所属的块，再使用2.2小节所述的方法将对应位置为1。

时间复杂度：O(1)

3.2 删除某个元素

首先计算出被删除元素所属的块，再使用2.2小节所述的方法将对应位置为0。

时间复杂度：O(1)

3.3 查询某个元素是否存在

首先计算出待查询元素所属的块，再利用2.1小节所述的方法检查对应位是否为1。

时间复杂度：O(1)

3.4 交集

集合的交和bitset的与运算相对应。`x`与`y`的交为`x & y`。

时间复杂度：$O(n/w)$

3.5 并集

集合的交和bitset的或运算相对应。`x`与`y`的并为`x | y`。

时间复杂度：$O(n/w)$

3.6 差集

记A、B是两个集合，则所有属于A且不属于B的元素构成的集合叫做A与B的差。集合的差可以通过bitset的与运算和异或运算实现。`x`与`y`的差为`x ^ (x & y)`。

时间复杂度：O(n/w)

3.7 补集

集合的补与bitset的非运算相对应。x的补为~x。

时间复杂度：O(n/w)

3.8 统计元素个数

对于每个块分别使用2.3小节中介绍的方法计算1的个数，再将所有块的答案相加。

时间复杂度：O(n/w)

3.9 遍历集合元素

遍历所有块，对每个块使用2.9小节中介绍的方法遍历元素。

时间复杂度：O($n/w + cnt$)，其中cnt表示集合的元素个数。

3.10 求集合中第k小元素

朴素做法

首先从小到大依次遍历每个块，找出第k小元素所在的块，再利用2.6 小节中介绍的方法确定具体是哪个元素。

时间复杂度：O($n/w + \log w$)

更加高效的做法

在某些应用中，求第k小元素可能比其它集合操作的使用频率更大，此时我们就会需要一个复杂度更优的查询第k小元素操作，同时要尽可能小地影响其它集合操作的复杂度。

一种改进方式是利用线段树。我们使用一颗辅助线段树来维护bitset中每个块的元素个数。这样一来，我们就可以通过在线段树上二分的方式快速确定第k小元素所属的块。这是一个经典的线段树操作，在这里就不再赘述。找出

第k小元素所在的块之后，我们就可以利用2.6小节中介绍的方法确定具体是哪个元素了。这样改进之后，求第k小元素的时间复杂度降为了$O(\log n)$。

添加了辅助线段树之后，其他的集合操作也要做出相应修改。在加入、删除集合元素时，该元素所属的块的元素个数会发生改变，我们需要同时维护辅助线段树的信息，因此这两个操作的时间复杂度变为了$O(\log n)$。在集合求交、并、差、补的时候，我们可以直接暴力重建辅助线段树，时间复杂度维持不变为$O(n/w)$。

3.11 求集合中大于x的最小元素

朴素做法

首先判断x所在块中是否存在大于x的元素，若是则直接返回这个元素，否则从x所在的块开始向后遍历，找到x之后第一个包含元素的块，然后返回该块的第一个元素。其中，某块的第一个元素以及某块中比x大的第一个元素均可以利用2.5小节中介绍的统计后缀0个数的方法求出。

时间复杂度：$O(n/w)$

更加高效的做法

类似于求第k小元素，我们也可以利用辅助数据结构来加速大于x的最小元素的查询。

这次我们利用的是bitset本身。我们使用一个大小为n/w的辅助bitset[7]来保存每个块是否有元素，这样一来，我们可以通过如下流程来完成该查询：

1. 若x所在块中存在大于x的元素，则直接返回这个元素，否则转2。
2. 利用辅助线段树查询x所在块之后第一个含有元素的块，转3。
3. 在该块中找第一个元素并返回结果。

改进之后，该查询的时间复杂度降为了$O(\log_w n)$。

同样地，其他的集合操作也要做出相应修改。在加入、删除集合元素时，我们需要同时维护辅助bitset中该元素所在块的信息，因此这两个操作的时间复

[7]这里的bitset是递归定义的，因此这个辅助bitset也支持快速求集合中大于x的最小元素。

杂度变为了$O(\log_w n)$。在集合求交、并、差、补的时候，我们可以直接暴力重建辅助bitset，时间复杂度维持不变为$O(n/w)$。

3.12 将集合的全部或部分元素加上/减去x

将集合的全部元素加上/减去x

对集合a的每个元素都加上x只需要将a左移x位即可。同样的，对集合a的每个元素都减去x 只需要将a右移x位即可。

时间复杂度：$O(n/w)$

将集合的部分元素加上/减去x

设原集合为a，要求改动的那部分元素组成的集合为b。在这次修改中，a与b的差这部分元素是不会改变的，记为集合p；a与b的交这部分元素是需要加上/减去x的，记为集合q。之后我们利用移位操作对集合q作相应修改，再与集合p求并即可。

时间复杂度：$O(n/w)$

3.13 枚举子集

枚举子集的关键在于，对于某个集合的某个子集，我们需要求出字典序排在它前一位的集合或后一位的集合。对于集合x和它的一个子集y，我们可以通过(y - 1) & x来求出字典序排在它前一位的子集。因此，若要枚举x的子集，我们只需从x本身开始，不断求前一个子集，直到枚举到空集为止即可。

3.14 枚举含有k个元素的集合

与枚举子集类似，枚举含有k个元素的集合的关键同样是对于某个含有k个元素的集合求出字典序排在它前一位或后一位的集合。这一次我们选择的是构造字典序后一位的集合。

朴素的做法如下：设当前集合中的数从小到大依次为$a_0, a_2, \ldots, a_{k-1}$，其中最小的可以增大的数为$a_j$，那么我们把$a_j$ 加上1，并将a_0到a_{j-1}重置为0到$j-1$即可得到下一个包含k 个元素的集合。

这个做法反映到bitset上是这样的：设最右边连续的1是第i位到第j位，那么我们要做的就是将第j位上的1向左移一位，同时将第i位到第$j-1$位上的1向右移到最右边。这可以由下面这段代码完成：

```
u32 nextComb(u32 x) {
    u32 l = x & -x, y = x + l;
    return y | (((x ^ y) / l) >> 2);
}
```

它是这样工作的：

- 第一步，使用`l = x & -x`求出`x`的lowbit。
- 第二步，`y = x + l`将第j位上的1向左移动一位，同时将第i位到第$j-1$位置为0（i和j的含义见上文）。
- 第三步，`((x ^ y) / l) >> 2`将第i位到第$j-1$位上的1提取出来并移到最右边。
- 第四步，将这些1与`y`合并得到答案。

4 其他应用

除上两节所介绍的应用外，我们还可以利用位运算做一些其他事情。

4.1 判断奇偶性

根据奇偶性的定义，二进制最低位为0的数为偶数，最低位为1的数为奇数，因此我们可以直接使用`x & 1`来判断`x`的奇偶性。若`x & 1`等于`1`则表明`x`为奇数，否则`x`为偶数。

4.2 乘以或除以二的幂

根据移位运算的实际意义，我们可以直接利用左移和右移实现，参见1.2 小节中的相关介绍。

4.3 对二的幂取模

$x \bmod 2^y$ 相当于取出x的后y位，因此可以用`x & ((1 << y) - 1)`来实现相同的效果。

4.4 求$\log_2 x$的整数部分

设x的位宽为w，使用2.5小节中介绍的方法计算出的前缀0个数为l，那么$\lfloor \log_2 x \rfloor = w - 1 - l$。

4.5 交换两个数

这是异或的经典应用。我们可以通过如下代码来交换`a,b`：

```
a ^= b, b ^= a, a ^= b
```

4.6 比较两个数是否相等

我们可以计算两个数的异或值，若等于零则说明这两个数相等，否则说明这两个数不相等。

4.7 取绝对值

对于一个w位带符号整数x，它的符号位`sign = x >> (w - 1)`，我们可以通过表达式`(x ^ -sign) + sign`来得到x的绝对值。这一点根据补码的定义比较容易证明，在此就不赘述了。

4.8 选择

我们可以利用表达式`y ^ ((x ^ y) & -cond)`来实现选择的功能：

- 若$cond = 1$，其结果为x
- 若$cond = 0$，其结果为y

5 例题

5.1 筷子

试题来源

经典问题

试题大意

有$2n+1$个整数，其中某个数出现了奇数次，其他数出现了偶数次。求出现了奇数次的那个数。

算法介绍

由于异或运算满足交换律，并且满足$x \oplus x = 0$，因此将所有数异或起来即为答案。

5.2 Robot in Basement

试题来源

Codeforces 97D[8]

Yandex.Algorithm 2011 Finals

试题大意

在一个$n \times m$ $(n, m \leq 150)$的网格上，有一些格子是障碍，并且网格边界上的格子均是障碍，另有一个非障碍的格子是出口。有一个机器人可以根据程序在该网格上行走。一段程序是一个由UDLR 四种指令组成的字符串，机器人会依次执行每个指令，一个指令会使机器人向指定的方向移动一格，如果对应格子为障碍则不动。

现在给定一个长度为l $(l \leq 10^5)$的程序，求它的一个最短前缀p，使得对于一开始在网格图上任意非障碍位置的机器人，在执行完程序p之后都停在出口上。

[8]http://codeforces.com/problemset/problem/97/D

算法介绍

首先不难想到一个O(nml)的朴素算法。我们直接顺序执行给定的程序，并按照题目指定的移动方式暴力维护网格上哪些格子有机器人，直到执行完某一步后发现网格上只有出口位置有机器人时结束。

实际上我们可以利用bitset来维护网格上哪些格子有机器人。一种表示方式是利用bitset的第$i \cdot m + j$个位置表示格子(i, j) $(0 \le i < n, 0 \le j < m)$上是否有机器人。这样一来一个向左或向右移动的指令就对应了该bitset的左移或右移1位，一个向上或向下的指令就对应了该bitset的左移或右移m位。障碍的问题可以这样解决：预处理cL, cR, cU, cD四个bitset，分别表示向左、向右、向上及向下走一步会碰到障碍的位置。以向左走为例，设原bitset为`x`，那么执行一个指令L之后会撞到障碍的位置为`x & cL`，能正常向左走的位置为`x ^ (x & cL)`，我们知道能正常向左走的那些机器人应该被左移一位，而那些会撞到障碍的机器人则应留在原地，因此最终的bitset应变为`(x & cL) | (x ^ (x & cL))`。改进后的算法的时间复杂度为$O\left(\frac{nml}{w}\right)$，可以通过本题。

5.3 Quick Tortoise

试题来源

Codeforces 232E[9]

试题大意

在一个$n \times m$ $(n, m \le 500)$的网格上，有一些格子是障碍。共有q $(q \le 6 \cdot 10^5)$个询问，每次询问是否能只通过向下走和向右走从格子(x_1, y_1)走到格子(x_2, y_2)。

算法介绍

首先考虑所有询问满足$x_1 \le p \le x_2$的情况。对于所有在第p行上方的格子(x, y)，求它能走到第p行的哪些格子，记这个点集为$f_{x,y}$，它可以通过递推式$f_{x,y} = f_{x+1,y} \cup f_{x,y+1}$求出。类似地，对于所有在第$p$行下方的格子$(x, y)$，求第$p$行的哪些格子能走到它，记这个点集为$g_{x,y}$，它可以通过$g_{x,y} = g_{x-1,y} \cup g_{x,y-1}$求

[9]http://codeforces.com/problemset/problem/232/E

出。求出$f_{x,y}$和$g_{x,y}$之后，我们就可以来处理询问了。对于询问$(x_1, y_1), (x_2, y_2)$，若$f_{x_1,y_1} \cap g_{x_2,y_2} \neq \varnothing$，则表示可以从$(x_1, y_1)$走到$(x_2, y_2)$，反之则不能。

对于一般情况，我们可以使用分治转化成上面的情况进行求解。例如我们正在处理所有满足$l \le x_1 \le x_2 \le r$的询问，令$mid = \left\lfloor \frac{l+r}{2} \right\rfloor$，我们可以先按照上一段所述的方法处理所有满足$l \le x_1 \le mid \le x_2 \le r$的询问，再递归地处理剩下的满足$l \le x_1 \le x_2 < mid$的询问和满足$mid < x_1 \le x_2 \le r$的询问。如果我们只对$x$轴进行分治，那么这个算法时间复杂度为$\mathrm{O}\left(\frac{n \cdot m^2}{w} \log n + q \log n\right)$。如果我们交替对$x$轴和$y$轴进行分治，它的时间复杂度可以进一步降为$\mathrm{O}\left(\frac{(nm)^{1.5}}{w} + q \log nm\right)$

5.4 Bags and Coins

试题来源

Codeforces 356D[10]

试题大意

有n ($n \le 70000$)个包，每个包可以直接放在地上，也可以放在其他包里面，并且允许多层嵌套。有s ($s \le 70000$)个硬币分布在这n个包里。如果拿出某个硬币必须要打开第i个包，我们就称该硬币在第i个包里。现在已知每个包里的硬币数，第i个包里有a_i ($a_i \le 70000$)个硬币。求一种满足条件的包的嵌套方式以及硬币分布方式，或确定问题无解。

算法介绍

不妨设$a_1 \ge a_2 \ge \ldots \ge a_n$。

我们首先考虑$s = a_1$的情况。在这种情况下，我们只需要将第1个包放在地上，并在第i ($1 \le i < n$)个包中放入$a_i - a_{i+1}$ 个硬币以及第$i+1$个包，最后在第n个包中放入a_n个硬币即可。

于是对于一般情况，即$s \ge a_1$的情况，问题就转化成了选一些包放在地上(第1个包是必选的)，让它们的硬币个数之和等于s。这是一个经典的背包问题。我们可以用一个bitset来表示当前背包（通过当前添加的物品能够达到哪

[10]http://codeforces.com/problemset/problem/356/D

些总体积），设当前bitset为S，那么添加一个体积为x的物品后的bitset就会变为S | (S << x)。当添加完所有物品之后，我们检查最终的bitset中是否包含s这个元素就能确定是否有解。

如何记录方案呢？我们引入一个$from$数组，其中$from[i]$表示在某种体积达到i的方案中最后加入的物品。这样一来，如果问题有解，我们就可以通过$from$数组不断追溯到上一个加进来的物品，也就可以恢复方案了。$from$ 数组的维护也很容易，在添加第i个物品的时候，设添加之前的bitset为S，之后为Sn，记Sn与S的差为D，我们枚举D的每个元素j，并将$from[j]$ 设为i即可。由于D的实际意义是加入第i个物品时新产生的可行体积的集合，因此$from$数组的每个元素至多被设置一次，也就保证了复杂度。

该算法的时间复杂度为$O\left(\frac{ns}{w}\right)$。

参考文献

[1] Sean Eron Anderson, “Bit Twiddling Hacks”.

[2] Christer Ericson, “Advanced bit manipulation-fu”.

[3] 刘汝佳, 黄亮, 《算法艺术与信息学竞赛》, 清华大学出版社。

寻找第 k 优解的几种方法

绍兴市第一中学 俞鼎力

摘 要

求第 k 优解是信息学竞赛中十分常见的题目类型。本文对这类题目的解法进行了分类归纳，使得此类问题可以一般化地转化成较普通、较简单的问题。此外，本文给出了 k 短路的 $O(n\log n+m+k\log k)$ 算法和 k 小生成树的 $O(m\log n+k^2)$ 算法以及 k 小简单路径的 $O(kn(m+n\log n))$ 算法。对于文中的每一项内容（除去算法介绍的章节），都给出了例题，便于读者理解和归纳。

1 引言

好奇心是学者的第一美德。——居里夫人

在信息学竞赛中，解决完某个问题之后，我们往往会考虑原问题的一些变种，比如改变题目中某个条件，改变题目中的目标函数，甚至寻求该问题的第 k 优解。

信息学竞赛中，求解第 k 大值或第 k 小值的题型已经非常普遍。对于这类题型，我们称之为求第 k 优解。求第 k 优解可能需要用到的算法很多：比如数据结构、动态规划、图论……

我们有一些已知的方法来寻找第 k 优解或者将问题转化得较为简单。比如二分、权值线段树、字母树上统计等等。也有一些大家并不是非常熟悉的算法，比方说 k 短路算法、k 小生成树算法、k 小简单路径算法。本文对这些内容进行了全面的描述，对大家熟悉的部分进行了总结归纳，对大家并不熟悉的部分进行了详细的讲述。希望能给所有参加信息学竞赛的同学带来或多或少的帮助。

在本文第二节，作者对第 k 优解进行了数学化的定义，为之后的论述打下基础。紧接着，作者证明了使用二分法求解第 k 优解的正确性，并给出了两道有趣的例题供读者深入研究。

在本文第三节，作者首先引用了 2013 年许昊然对整体二分的论述[3]，着重讲了权值线段树与整体二分的联系，以及它在求第 k 优解时的两种使用方法，对这两种方法分别给出了一道例题。

在本文第四节，作者归纳了在字母树上统计来求第 k 优解的方法。

在本文第五节，作者详细地讲述了优先队列在求第 k 优解时的应用，并对辅助用的图 P 下了定义并进行了扩展。之后的所有内容都利用了优先队列，并且都会使用到辅助用的图 P.

在本文第六节，作者对 k 短路算法进行了详细的讲解，并提及了堆的可持久化。k 短路算法的应用在本文第七节中提到，即一类动态规划问题第 k 优解的通用做法。

在本文第八节，作者耗费大量的篇幅介绍了 k 小生成树算法。刘汝佳在《算法艺术与信息学竞赛》一书中提及过 k 小生成树，但是只给出了一个不加证明的定理，且没有给出具体做法。作者查阅了诸多国外的文献资料，得出了本文中 k 小生成树的 $O(m\log n + k^2)$ 算法。

在本文第九节，作者对 k 小生成树的构图 P 的方法进行总结，并在此基础上提出了 k 短简单路径的 $O(kn(m + n\log n))$ 算法。

2 二分法求第 k 优解

2.1 第 k 优解的定义

首先，我们需要给第 k 优解一个数学化的定义。

先给出最优化问题的数学描述：

定义2.1. *最优化问题即在一给定集合上求某函数的极值(极大化或极小化)问题. 对于极小化情形，即为求*

$$\min_{u\in U} J(u),$$

其中 u 表示问题的一个解，$u \in U$ 表示解 u 需要满足给定的约束(组成集合 U 的条件称为约束条件)，函数 J 表示对解的优劣判断的指标(称为问题的目标函数).

类似的我们可以给出第 k 优解的数学描述：

定义2.2. *第 k 优解问题即在一给定集合上求某函数的第 k 极值(极大化或极小化)问题. 对于极小化情形，即为求*

$$\max_{u\in K} J(u),$$

其中解集 K 满足

$$|K| = k \quad 且 \quad \max_{u\in K} J(u) \le \min_{v\in U-K} J(v).$$

2.2 一个转化

定理2.1. *第 k 优解问题在只需求 $\max_{u\in K} J(u)$ 而不需要求解 u 的具体方案时，可以花费 $O(\log(\sup J(U) - \inf J(U)))$[1] 的代价转化为一个计数问题。其中 $J(U)$ 表示函数 J 在定义域 U 下的值域，且 $J(U) \subseteq \mathbb{Z}$.*

Proof. 这里只证明第 k 优解的极小化情形。设集合

$$S(x) = \{u \in U : J(u) \le x\},$$

则

$$\max_{u\in K} J(u) = \min_{|S(x)|\ge k} x.$$

由于 $|S(x)|$ 关于 x 是一个单调函数，所以我们可以二分 x 求出 $\min_{|S(x)|\ge k} x$.

由于 $J(U) \subseteq \mathbb{Z}$，根据整数的二分复杂度可以得到，问题能通过花费 $O(\log(\sup J(U) - \inf J(U)))$ 的代价转化为：给定 x, U, J，求

$$\sum_{u\in U, J(u)\le x} 1.$$

第 k 优解的极大化情形同理。 □

定理 2.1 中要求 $J(U) \subseteq \mathbb{Z}$，如果 $J(U) \subseteq \mathbb{R}$ 且题目要求四舍五入到 d 位小数，那么我们就可以令 $J'(u) = \text{round}(J(u) \cdot 10^d)$[2] 将其转成满足条件的函数。

注意到定理 2.1 中所求的计数问题为

$$\sum_{u\in U, J(u)\le x} 1,$$

[1] $\sup S$ 和 $\inf S$ 分别表示集合 S 的上界和下界

[2] $\text{round}(x)$ 表示 x 四舍五入之后的值

它是在 $u \in U$ 的基础上添加了一个条件 $J(u) \le x$，因此往往在 $J(u)$ 比较简单的情况下，可以使用定理 2.1 来简化问题；而对于 $J(u)$ 计算非常复杂的情况，我们需要简化 $J(u) \le x$ 这个条件(如例题 2)，如果无法简化，运用定理 2.1 可能会使得问题变得更加复杂。

2.3 例题一

例1 (COCI 2011/2012 4^{th} round BROJ). 给定一个质数 p，求最小质因子等于 p 的第 k 小正整数。若答案大于 10^9 则输出 0.

此时 $J(u) = u$，$U = \{u \in \mathbb{Z}^+ : u \le 10^9 \text{ 且 } u \text{ 的最小质因数为 } p\}$. 可以发现 $U = \{vp : v \le \frac{10^9}{p} \text{ 且 } v \text{ 的最小质因数不小于 } p\} \subseteq \{vp : v \le \frac{10^9}{p}\}$.

当 $p \ge 67$ 时，$|\{vp : v \le \frac{10^9}{p}\}|$ 不大，可以直接枚举其中的所有元素，然后判断其是否在 U 中，找到满足条件的第 k 小即可。

当 $p \le 61$ 时，根据定理 2.1，可以在 $[0, 10^9]$ 中二分 x，并统计

$$\sum_{v=1}^{\lfloor \frac{x}{p} \rfloor} (1 - [v\text{的最小质因数小于}p])^{3}.$$

然后根据容斥原理可以得到

$$\sum_{v=1}^{\lfloor \frac{x}{p} \rfloor} \left(1 - [v\text{的最小质因数小于}p]\right) = \sum_{d=1}^{\lfloor \frac{x}{p} \rfloor} \left([d\text{的质因子均小于}p] \cdot \mu(d) \left\lfloor \frac{x}{pd} \right\rfloor \right).$$

而质因子均小于 p，且 $\mu(d) \ne 0$ 的 d 的个数不超过 $2^{\pi(p)}$ 个[4]，可以顺利通过此题。

复杂度 $O\left(\min\left\{\frac{L}{p}, 2^{\pi(p)} \log L\right\}\right)$，此题中 $L = 10^9$.

2.4 例题二

例2 (TopCoder SRM 607 Div. 1 - Level 3). 桌面上有两个钉子和 n 个滑轮。两个钉子的坐标分别为 $(0, 0)$ 和 $((n+1)d, 0)$. 每个滑轮的半径均为 r，滑轮的圆心坐标为 $(d, 0), (2d, 0), \ldots, (n \cdot d, 0)$. 保证滑轮不会重叠。

[3] $[S]$ 当命题 S 为真时等于1，否则等于0

[4] $\pi(n)$ 表示不超过 n 的素数个数

要将两个钉子用一个**紧绷**的绳子连起来，绳子可以在滑轮上绕任意圈。现给定 n, d, r, k，问在所有方案中，第 k 短的绳子长度。要求与答案误差不超过 10^{-9}.

数据范围：$1 \le r \le 499\,999\,999, 2r+1 \le d \le 1\,000\,000\,000, 1 \le n \le 50, 1 \le k \le 10^{18}$.

同样套用定理 2.1，首先二分绳子的长度 *bound*，统计绳子长度不超过 *bound* 的方案数。

将绳子的每一段分成 4 类：e, A, B, R，如图 1。四者的长度均能通过简单几何得出。

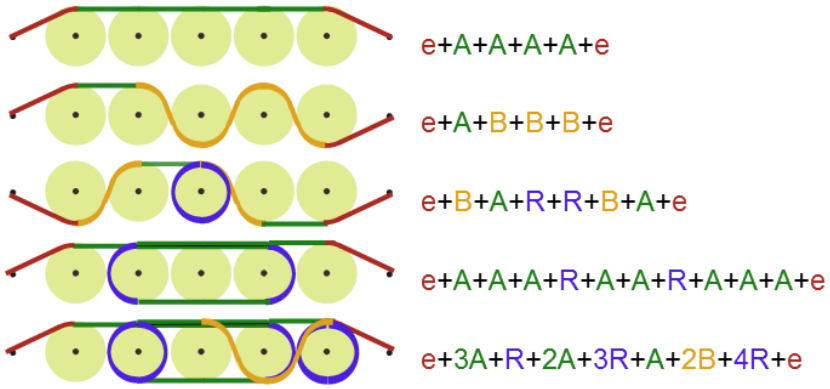


Figure 1

所以，我们需要统计的就是所有满足 $2e + xA + yB + zR \le bound$ 的 (x, y, z) 三元组的方案数。

这样，我们可以设计一个比较暴力的动态规划：用六元组 (x, y, z, p, d, b) 来表示当前用了 x 个 A、y 个 B、z 个 R，在第 p 个滑轮上，方向是 d(0 表示向右，1 表示向左)，在轮子上的位置为 b(0 表示在上方，1 表示在下方)时的方案数。转移显然。

第一点，可以发现方向 d 完全是由 R 的个数 z 决定的，即 $d = z \mod 2$.

其次有：b 的取值不会影响转移，并且两种情况始终是对称的，如图 2。

在注意到 b 的取值无所谓后，可以发现 A 和 B 的转移完全相同，所以在动态规划时，我们可以只记录 $w = x + y$，然后在计算 (x, y, z) 三元组的方案数时，我们需要在动态规划的基础上乘上 $\binom{x+y}{y}$.

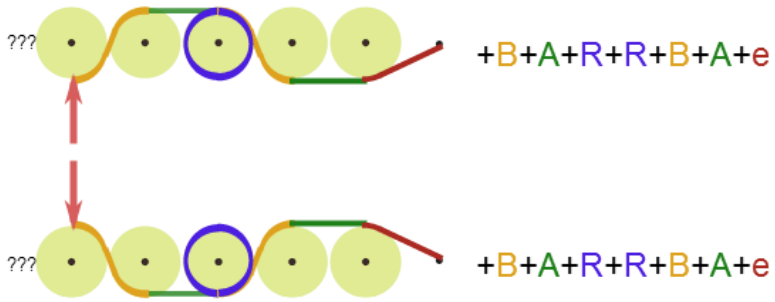


Figure 2

因此，在统计时，可以先枚举 w,z，再枚举 x，如果 (x,y,z) 满足 $2e+xA+yB+zR\le bound$，就将 (w,z) 的方案数乘上 $\binom{x+y}{y}$ 加入答案。所以 w 枚举的范围不大，枚举到 $O(\log k)$ 约 70 已经足够。

但是由于 z 仍然可能很大，无法记录，考虑进一步优化：在 A 或 B 之后出现多个 R 是没有意义的，因为转两次相当于没转。而去掉的多余的 R，可以在最后乘上一个组合数来分配。即，枚举 w,z,x 之后$(z\le w)$，计算最多能再加入的 R 的个数，即

$$rest=\frac{bound-(2e+xA+yB+zR)}{2R},$$

将 (w,z) 的方案数乘上 $\binom{x+y}{y}$ 再乘上 $\binom{w+1+rest}{w+1}$[5] 加入答案。

最后一个小细节，在计算组合数 $\binom{n}{m}$ 时$(m\le 70)$：对于 $m<4$ 的情况，我们可以 $O(m)$ 暴力计算；而对于 $m\ge 4$ 的情况，我们可以预处理 $n\le 70000$ 的所有组合数，因为如果 $n>70000$，则 $\binom{n}{m}$ 一定大于 10^{18}.

至此，问题得到很好的解决。对于二分的上界，可以取 $(n-1)A+2kR$，时间复杂度为 $O(\log^3 k\log(dn+rk)+k^{\frac{1}{4}}\log k)$.

3 整体二分和权值线段树

3.1 整体二分

在数据结构题中，往往题目要求回答多个询问，比如询问 $[l_i,r_i]$ 区间中第

[5]这个组合数可以使用隔板法得到，也可以直接借助生成函数。

k_i 极值。我们仍然能够使用二分来解决，但是单纯的二分是不够的，这时我们需要使用**整体二分**。整体二分的具体做法请参见许昊然的2013集训队论文[3]。

这里引用它需要满足的性质：

1. 询问的答案具有可二分性；
2. 修改对判定答案的贡献互相独立，修改之间互不影响效果；
3. 修改如果对判定答案有贡献，则贡献为一确定的与判定标准无关的值；
4. 贡献满足交换律、结合律，具有可加性；
5. 题目允许离线算法。

其中在求第 k 优解时，第一条性质已经被定理 2.1 证明。

3.2 使用权值线段树求第 k 优解

整体二分需要满足的性质中，第二条和第五条显得非常苛刻。因为很多问题都需要在线进行，而且修改某个点的权值。对于这种情况，我们可以运用**权值线段树**将问题转化为计数问题，同样花费 $O(\log W)$ 的代价($W = \sup J(U) - \inf J(U)$)。

建以权值为关键字的线段树，在线段树节点上维护 $J(u)$ 在 $[l, r]$ 之间的 u 的数据结构 $T_{l,r}$.

询问时，在 $T_{l,r}$ 中询问 $u \in U_i$ 的解 u 的个数 s (U_i 表示第 i 个询问解的约束条件，例如解在某个区间内)，若 $s \leq k$，则在权值线段树中往右走，否则往左走，并将 k 减去 s.

插入一个新的解 v，就在 $\log W$ 个满足 $l \leq J(v) \leq r$ 的 $T_{l,r}$ 中插入 v. 删除同理。修改一个点的权值就是先删除再加入。

但是此方法不能修改 $u \in U_j$ 的所有 $J(u)$ (U_j 表示第 j 个修改操作中需要修改的解的约束条件)。

可以发现整体二分时，我们也利用了这个模型，但是整体二分是离线地在权值线段树上走，不维护这些数据结构，所以其无法支持可能会影响的修改；但是好处是它可以继续在线段树的节点上套用分治等离线做法，并且节省内存。

3.3 例题三

例3 (BZOJ 3065 带插入区间 k 小值). 开始 n 个数：$a_1, \ldots, a_n$. 执行 q 个操作，操作有三种：询问从左至右第 x 个数到从左至右第 y 个数中，第 k 小的数；将从左至右第 x 个数改为 val；在从左至右第 x 个数之前插入一个值为 val 的数。

要求强制在线。

数据范围：$n \leq 35000$，插入个数 ≤ 35000，修改个数 ≤ 70000，询问个数 ≤ 70000，$0 \leq$ 每时每刻权值 ≤ 70000.

维护权值线段树，每个节点是一棵平衡树 $T_{l,r}$，按照原来顺序维护权值在区间 $[l, r]$ 内的数，如果一个数权值 $\leq m = \left\lfloor \frac{l+r}{2} \right\rfloor$，那么它在平衡树中的特征值为 0，否则特征值为 1.

询问时，首先找到 x 和 y 在 $T_{0,W}$ 中的位置，这样可以得到 x 和 y 之间特征值为 0 的数的个数 s，如果 $s \leq k$，则走到 $T_{l,m}$ 中，否则走到 $T_{m+1,r}$ 中，并将 k 减去 s. 而 x, y 在下一层平衡树中的位置可以通过做类似这样的询问得到：在 x (包含 x)之前(之后)的第一个特征值为 c 的数在下一层平衡树的位置。因此需要维护每个数在权值线段树的每一层的平衡树中的位置。递归直到 $l = r$ 即可。

插入和修改的方法与询问类似。

总复杂度为 $O((n+q) \log W \log n)$. 但与其他复杂度相同的算法相比代码复杂度较低。

3.4 权值线段树的另一种用法

如果换一个思路，将权值线段树作为某一数据结构的点，来维护某一区间的解的权值分布，将会产生一个问题：很多数据结构都需要合并点权，而权值线段树无法高效地合并。我们需要一系列手段来解决这个问题。

对于答案的合并，可以这样做：由于只需要询问第 k 极值，所以我们可以假设已经合并好了，并得到了答案所代表的权值线段树。在权值线段树上走的时候，再去计算当前需要统计的节点的值。

对于标记也是权值线段树的情况：由于不能下传标记了，所以应该将标记永久化，详见例 4。

对于数据结构中需要用子节点更新父节点的情况，如平衡树左旋或右旋之后需要更新时，可以选择重量平衡的数据结构，例如 Treap，替罪羊树等等，暴力重构整棵子树做到期望或均摊较优的复杂度。

3.5 例题四

例4 (ZJOI2013 K 大数查询). 有 n 个位置，m 个操作。操作有两种：在第 a 个位置到第 b 个位置，每个位置加入一个数 c；或者询问从第 a 个位置到第 b 个位置，第 k 大的数是多少。

数据规模：$n, m \leq 50000, |c| \leq n$.

此题做法很多，作者在考场上写了线段树套权值线段树的做法，即对位置维护线段树，线段树上每一个节点维护两棵权值线段树，一棵设为 $C_{l,r}$ 表示当前节点的懒标记。它的意义在于：当前节点对应的区间上每个位置都加了这些权值的数。另一棵设为 $T_{l,r}$ 维护了当前节点对应的区间上所有数的权值。

对于询问操作，如果 $a = l$ 且 $b = r$，那么就直接返回 $T_{l,r}$，否则返回 $[a, b]$ 递归到左儿子和右儿子的结果加上 $(b - a + 1) \cdot C_{l,r}$.

对于修改操作，在 $T_{l,r}$ 中加入 $(b - a + 1)$ 个 c，如果 $a = l$ 且 $b = r$，那么在 $C_{l,r}$ 中加入 c，否则递归到左儿子和右儿子。

时间复杂度 $O(m \log^2 n)$.

值得一提的是，这个做法可以再花费 $O(\log n)$ 的代价来支持区间加一个值，同样也是利用懒标记。

在线段树上多维护一个标记 D，表示当前节点对应的区间全部加了 D. 修改时，一旦进入一个节点，c 就要减去当前节点的 D. 询问时，由于答案是 $\sum s_i(T_i + d_i)$ 的形式(其中 T_i 表示一棵权值线段树)，所以在最后的权值线段树上走的时候需要额外付出 $O(\log n)$ 的代价。复杂度为 $O(m \log^3 n)$.

4 字母树上的统计

信息学竞赛中还有一类题目，它往往要求满足某一条件的字典序第 k 小字符串。

当然对这类问题，我们可以套用二分，将问题转化为满足某一条件且字典序小于某个串的字符串个数。但是这样反而使问题变得复杂，不如平时我们在说的**逐位确定**来得方便、高效。

而逐位确定的实质就是在字母树上走，时刻保持当前子树中所有满足条件的方案个数不小于 k.

每次可以尝试着按字典序从小到大，向当前节点的儿子走，设其儿子对应的子树方案数为 s，如果 $s \geq k$，那么就确定走到该儿子，否则就将 k 减去 s.

而某一节点的子树内满足条件的字符串个数，相当于求前缀确定的满足条件的字符串个数。

不光是字符串，比如求第 k 小的排列，也是利用这种方法来做的。

4.1 例题五

例5. 给一个字符串 str，求其第 k 小子串。两个子串的起始位置不同或结束位置不同均视为不同的子串。

数据范围：$|str| \leq 100\,000$.

建字符串 str 的后缀树，由于后缀树也是字母树，所以可以在后缀树上走得到答案。而后缀树上某一子树内子串个数可以用树形动态规划求得，这里不再赘述。

也可以尝试使用其他做法，但是其本质不变，都是在后缀树上走。如果选手只会后缀数组，可以用后缀数组加单调栈建后缀仙人掌来做，由于后缀仙人掌即为后缀树的“左儿子右兄弟”形式，在本题的实现上非常方便。

4.2 例题六

例6. 对于一个包含 W 个单词的语句，相邻两个单词之间用一个空格隔开，单词只包含小写字母，其信息量为所有字符的信息量之和。空格信息量为 1，‘a’信息量为 2,……，‘z’信息量为 27. 问信息量为 V 且字典序第 k 小的语句。

数据规模：$1 \leq V \leq 1000, 1 \leq W \leq 300, 1 \leq k \leq 10^{18}$.

可以直接套用之前所述，在字母树上走，并询问前缀确定的满足条件的字符串个数。

但是如果每次暴力做动态规划无法通过，因此，我们需要预处理来优化。总复杂度为 $O(VW)$.

5 优先队列在求第 k 优解中的应用

5.1 一个使用堆的例子

首先，引用算法导论中的一道习题：

例7 (Introduction to Algorithms 6.5-8). 请给出一个时间为 $O(n \log m)$、用来将 m 个已排序链表合并为一个排序链表的算法。此处 n 为所有输入链表中元素总数。

做法非常显然。首先，我们将 m 个链表头放入一个小根堆中。进行 n 次操作，第 i 次操作，将堆中最小节点的从堆中取出，作为答案链表的第 i 个，然后在堆中加入该节点在原链表中的下一个。

例8. 给出 n 个数 $a_1, a_2, \ldots, a_n$ 和 m 个数 $b_1, b_2, \ldots, b_m$，问对于所有 $1 \le i \le n, 1 \le j \le m$，$a_i + b_j$ 的第 k 小值。

数据范围：$n, m, k \le 500000$.

将 b 排序，并设 $c_{ij} = a_i + b_j$，那么 $c_1, c_2, \ldots, c_n$ 均为长度为 m 的有序表。运用例 7 的做法，我们执行 k 次操作即可得到前 k 小值。时间复杂度 $O(m \log m + k \log n)$.

当然也可以二分答案来得到，每次枚举 i，维护可行的最大的 j，就可以计算满足 $a_i + b_j \le x$ 的 (i, j) 对数。复杂度为 $O(n \log W)$. 但是二分只能得到第 k 小值，而无法得到前 k 小值。

5.2 使用优先队列求第 k 优解的一般方法

构造解的一张图 P，满足所有解均是图中的一个或多个点，图中每个点都是一个合法的解，且这张图满足堆性质：即如果 u 连向 v，那么 $J(u) \le J(v)$.

新建一个节点 *source*，设其权值为 $-\infty$，从 *source* 向所有没有入度的点连边。那么从 *source* 出发能到达所有解。如图 3，即为例 8 对应的图 P 的其中一种。

用一个优先队列维护已扩展的点，初始只有一个点 *source*，进行 $k+1$ 次操作，第 i 次弹出当前优先队列中的最优解 u，作为第 $i-1$ 优解，将所有 u 到 v 有边，且之前还没被遍历到的 v 放入优先队列中。注意到这张图 P 并不需要

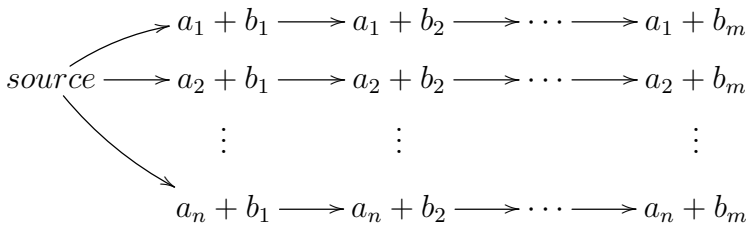


Figure 3

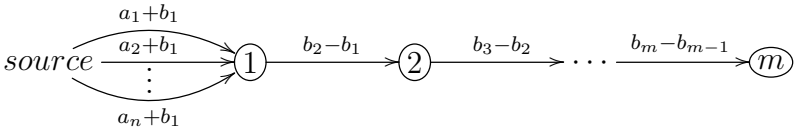


Figure 4

完全构出，可以在需要的时候再建边。要判断是否多次被遍历，可以使用散列表。设最终遍历到的解个数为 $State$，那么整个算法复杂度为 $O(State \log k)$.

注意到图的堆性质，我们可以设 $c(u,v) = J(v) - J(u)$，即 (u,v) 拥有边权 $J(v) - J(u)$. 在边有权值的意义下，对图 P 的定义进行扩展：从起点 $source$ 开始的路径与问题的解**一一对应**，解的值为路径上所有边权和。此时图 P 不一定是拓扑图。图 4 即为例 8 对应的图 P 的另一种。

例 8 还有一个 $O(k \log k + n + m)$ 的做法(如图 5)：首先对 b 建小根堆 h，建堆复杂度为 $O(m)$，详细请参见算法导论[5]；堆中父节点 u 向儿子 v 连边权为 $h_v - h_u$ 的边；$source$ 向堆的根连 n 条边，第 i 条边的边权为 $a_i + \min\{b\}$. 这样建得 P，由于除 $source$ 外所有点的出度不超过 2，对于 $source$，可以只加入权值前 k 小的边，这样复杂度即为 $O(k \log k + n + m)$.

在之后的章节中我们将看到优先队列在求第 k 优解时的应用，以及更多构建 P 的巧妙方法。

6 k 短路算法

定理6.1. *在一张有向带权图 G 中，从起点 s 到终点 t 的可重复经过同一点的不严格递增的第 k 短路的长度，可以在 $O(n \log n + m + k \log k)$ 的复杂度内得到。*

Proof. 对于一条边 e，定义它的边权(长度)为 $\ell(e)$，定义它的起始点为 $head(e)$、终止点为 $tail(e)$，用 $d(u,v)$ 表示 u 到 v 的最短距离。

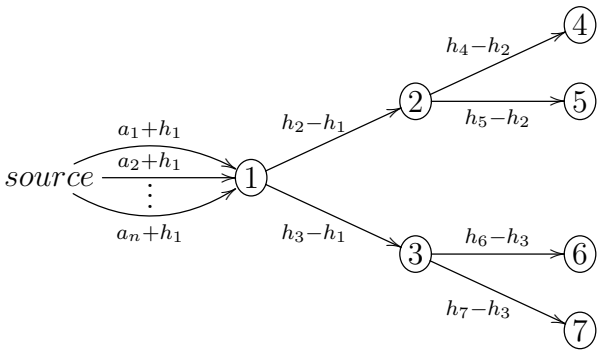


Figure 5

6.1 问题转化

首先定义 T 为 G 中以 t 为终点的最短路径构成的最短路径树。对于一条边$e \in G$，定义：

$$\delta(e) = \ell(e) + d(tail(e), t) - d(head(e), t),$$

那么有：

$$\forall e \in G, \delta(e) \geq 0; \qquad \forall e \in T, \delta(e) = 0.$$

例如图 6，A 为起点，L 为终点；右图中，所有实箭头构成最短路径树 $\delta = 0$，而虚箭头上所标即为该边的 δ 值。

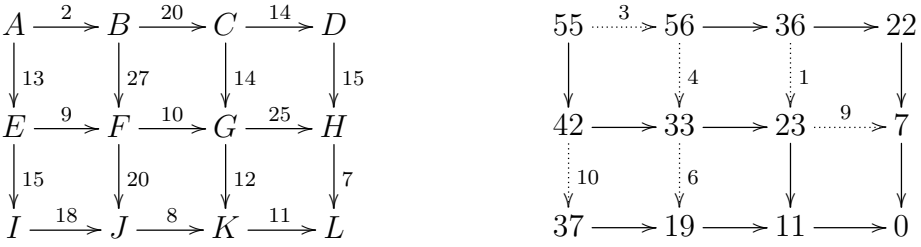


Figure 6

但是最短路径树可能并不一定是一棵树，在此我们要将 T 严格定义为一棵树；对于一个节点 $v \neq t$，定义 $next_T(v)$ 为 $v \to t$ 的最短路径（不唯一则任选一条）上 v 的后继节点。那么 G 中所有的点就以 $next_T$ 形成了一个严格的树结构 T。

对于一条从 s 到 t 的路径 p，去掉 p 中所有在 T 中的边，将其定义为 $sidetracks(p)$.

对于 $sidetracks$ 我们有以下三条性质：

性质6.1. 对于 $sidetracks(p)$ 中任意一条边 e，都有 $e \in G-T$；对于 $sidetracks(p)$ 中任意相邻的两条边 e, f，都满足 $head(f)$ 是 $tail(e)$ 在 T 上的祖先或 $head(f) = tail(e)$.

性质6.2. p 的路径长度 $l(p) = d(s,t) + \sum_{e \in sidetracks(p)} \delta(e) = d(s,t) + \sum_{e \in p} \delta(e)$.

性质6.3. 对于一个满足性质 6.1的边的序列 q，有且仅有一条 s 到 t 的路径 p 满足 $sidetracks(p) = q$.

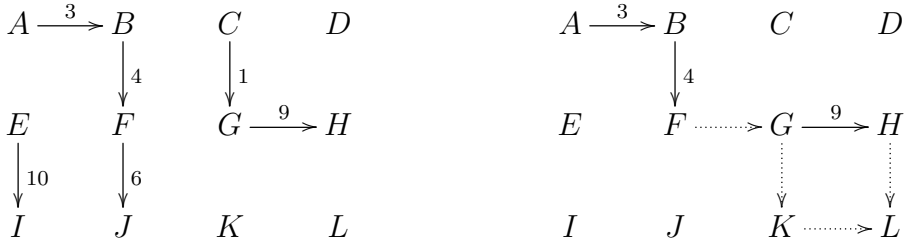


Figure 7

以上三条性质比较显然。例如图 7 中 $A \to B \to F \to G \to H \to L$ 这条路径，它的 $sidetracks$ 为 δ 值为 $3, 4, 9$ 的三条边。它的长度为 $dist(s,t)+3+4+9 = 71$.

至此，我们将问题转化为：求第 k 小的满足性质 6.1 的边的序列。

即，序列中任意一条边 e 都有 $e \in G - T$；任意相邻的两条边 e, f，都满足 $head(f)$ 是 $tail(e)$ 在 T 上的祖先或 $head(f) = tail(e)$.

序列 q 的权值定义为

$$w(q) = \sum_{e \in q} \delta(e).$$

6.2 构建 P

算法一 初始解为空序列。对于一个序列 q，令 q 最后一条边的 $tail$ 为 v (若是空序列则 $v = s$)。在 q 之后加入一条边 e 得到新序列 $q'(head(e)$ 在 T 中 v 到 t 的路径上且 $e \in G - T)$。q 向 q' 连边权为 $\delta(e)$ 的边。如图 8。

这样构成的图 P，每个点的出度最多为 m，所以复杂度为 $O(n\log n + km(\log k + \log m))$[6].

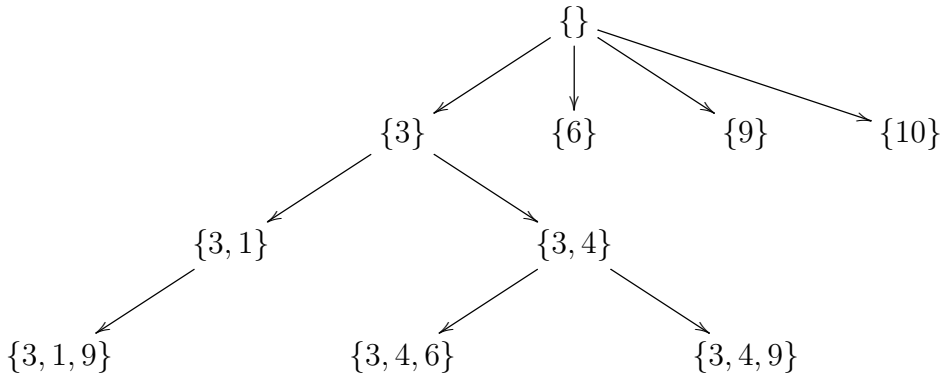


Figure 8

算法二 上图可以对每个点 v 的所有边排序，然后用左儿子有兄弟的方法将每个点的出度减小为 2，兄弟之间的边的边权为两者之差。如图 9。

它的意义在于建立一张有序表 $g(v)$，按权值从小到大记录所有在 $G-T$ 中且满足 $head(e)$ 在 T 中 v 到 t 的路径上的边 e.

对于序列 q，令其最后一条边为 e、$v = tail(e)$、倒数第二条边的 $tail$ 为 u. 将 e 替换为 $g(u)$ 中 e 的后一条边或者在 q 中新加入 $g(v)$ 中的第一条边得到新序列 q'.

复杂度为 $O(n\log n + nm\log m + k\log k)$.

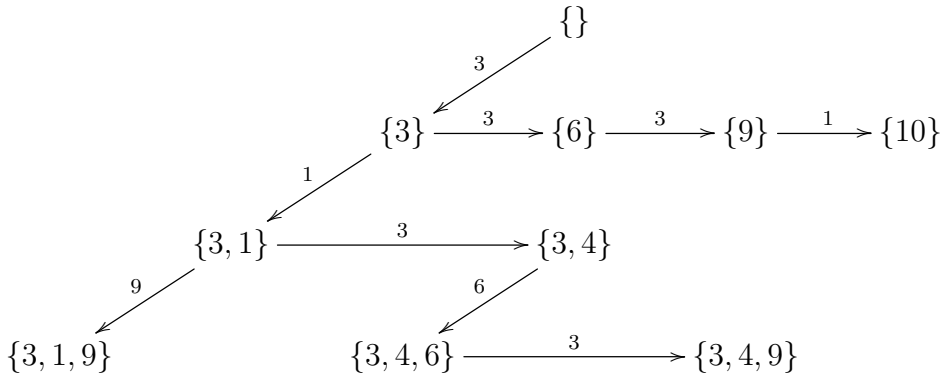


Figure 9

[6]这里默认第一步的最短路算法复杂度为 $O(n\log n + m)$.

算法三 上一个算法的瓶颈在于对每个点建立有序表。注意到 $g(v)$ 和 $g(next_T(v))$ 是有大部分相同的。事实上我们是在 $g(next_T(v))$ 中添上所有在 $G-T$ 的由 v 连出去的边来求得 $g(v)$ 的。但是有碍于有序表的添加复杂度，我们不能有效得来求得 g.

如果我们对于点 v 不建立 $g(v)$ 转而建立一个堆 $H(v)$ (类似例 8 的 $O(k\log k+n+m)$ 做法)，并通过在 $H(next_T(v))$ 中**可持久地**加入所有在 $G-T$ 的由 v 连出去的边来得到 $H(v)$，那么我们就有能解决上面的问题。

这里可持久化的原因是我们时刻需要用到插入之前的堆。

而对于上一算法中“将 e 替换为 $g(u)$ 中 e 的后一条边”这一步，我们直接将 e 在 $H(u)$ 中的两个儿子替换 e 即可。

复杂度为 $O(n\log n+m\log m+k\log k)$.

算法四 例 8 的 $O(k\log k+n+m)$ 做法中，我们用到了建堆的优越性，这里我们也可以利用。

对于点 v 用堆 $h(v)$ 维护所有在 $G-T$ 的由 v 连出去的边。并将 $h(v)$ 中最小元素的权值作为 $h(v)$ 的权值。

把 $h(v)$ 当作一个节点，可持久地插入 $H(next_T(v))$ 得到堆的堆 $H(v)$.

这样建 $h(v)$ 的时间为 $O(m)$，建 $H(v)$ 减少为$O(n\log n)$. 所以总复杂度为 $O(n\log n+m+k\log k)$.

□

关于可持久化堆的实现，作者在冬令营营员讨论[7]上讲解了二叉堆和二项堆的可持久方法，但是作者当时误以为左偏树和斜堆都是均摊复杂度的，所以没有介绍左偏树的可持久化，其实左偏树的复杂度与斜堆不同，是单次操作严格 $O(\log n)$. 所以左偏树是可以可持久化的。

左偏树的可持久化并不困难，设 $\textsc{Merge}(a,b)$ 返回将左偏树 a 和 b 合并得到的左偏树。如果 a 的权值大于 b，那么 $\textsc{Merge}(a,b)$ 返回 $\textsc{Merge}(b,a)$，否则新建一个节点，将其左儿子设为 $a.left$，右儿子设为 $\textsc{Merge}(a.right,b)$，并调整维护左右儿子及其键值，最后返回该节点即可。

7 一类动态规划问题

众所周知，拓扑图的最短路可以通过动态规划来实现。反过来说，如果某一动态规划能转成拓扑图，且它构出的图的边 (a,b) 的意义为 $b\le a+\ell(a,b)(a,b$

均为动态规划时的变量(状态)，对应于动态规划转移方程即为 $b = \min\{b, a + \ell(a,b)\}$)，那么此动态规划的最优解就可以用最短路来求解。

定理7.1. *如果某一动态规划能转成拓扑图，且它构出的图的边 (a,b) 的意义为 $b \le a + \ell(a,b)$(对应于动态规划转移方程即为 $b = \min\{b, a + \ell(a,b)\}$)，且此动态规划的所有决策方案与该拓扑图中所有起始点到终止点的路径一一对应，那么此动态规划的第 k 优解就可以用 k 短路来求解。*

设该动态规划的状态数为 n，转移数为 m，则其第 k 优解可以在 $O(n \log n + m + k \log k)$ 的复杂度内求出。

Proof. 由于动态规划的初始状态和终止状态可能不止一个，所以其构成的拓扑图中初始点和终止点也可能不止一个，需要新建节点 s, t，从 s 向所有初始点连长度为 0 的边，所有终止点向 t 连长度为 0 的边，此时此动态规划的第 k 优解即为拓扑图上 s 到 t 的 k 短路。

设该动态规划的状态数为 n，转移数为 m，则此时拓扑图上点数为 $O(n)$，边数为 $O(n+m)$，根据定理 6.1 可以得出在此拓扑图上求解 k 短路的复杂度 $O(n \log n + m + k \log k)$. □

如果动态规划转移方程为 $b = \max\{b, a + \ell(a,b)\}$，那么也是可以用 k 短路解的，只需将所有边权取原来的相反数，最后答案也取相反数。这也是求解最长路的方法。

7.1 例题九

例9. *给出 n 个一维上的点 $x_i (1 \le i \le n)$，要将这些点划分成两个集合 S_A, S_B. 对于一个集合 $S = \{p_1, p_2, \ldots, p_{|S|}\}$，其中 $p_1 < p_2 < \cdots < p_{|S|}$，设这个集合的代价*

$$\ell(S) = \sum_{k=0}^{|S|-1} \left| x_{p_k} - x_{p_{k+1}} \right|.$$

其中默认 $x_{p_0} = 0$.

现在要求所有方案中第 k 小的

$$\ell(S_A) + \ell(S_B).$$

数据范围：$n \le 10000, k \le 500000$.

解法一

如果 k 等于 1，那么我们可以设计动态规划来解决这个问题：

用 $f(i,j)$ 来表示两个集合的末尾分别为 i 和 j 时的最小代价。这里，可以默认 $i \geq j$，因为 $f(i,j) = f(j,i)$.

转移时，枚举 $i+1$ 会放到哪个集合中。即

$$f(i,j) + |x_{i+1} - x_i| \to f(i+1,j) \quad [7]$$
$$f(i,j) + |x_{i+1} - x_j| \to f(i+1,i).$$

初始化 $f(0,0) = 0$，其他均为 ∞. 答案等于 $\min_{j<n} f(n,j)$.

这样，我们就可以通过定理 7.1 来建拓扑图，然后用 k 短路来得到第 k 优解，由于图的点数和边数均为 $O(n^2)$，复杂度为 $O(n^2 \log n + k \log k)$.

容易看出 $f(i+1,j) - f(i,j)$ 只有当 $j = i$ 时不确定，对于其他的 $j \neq i$ 都有

$$f(i+1,j) = f(i,j) + |x_{i+1} - x_i|.$$

而对于 $f(i+1,i)$ 我们可以写成

$$f(i+1,i) = \min\{f(i,j) + |x_{i+1} - x_j| - |x_{i+1} - x_i|\} + |x_{i+1} - x_i|.$$

如果我们求出 $M_{i+1} = \min\{f(i,j) + |x_{i+1} - x_j|\}$，就可以这样转移：$M_{i+1} - |x_{i+1} - x_i| \to f(i,i)$，然后将整个 $f(i)$ 加上 $|x_{i+1} - x_i|$ 得到 $f(i+1)$.

这样我们优化点数至 $O(n)$，并且仍然满足定理 7.1 的所有条件，所以复杂度为 $O(n \log n + n^2 + k \log k)$.

解法二

可以发现，求 M_i，可以用线段树来做：设 $g(i,x_j) = f(i,j) - x_j, h(i,x_j) = f(i,j) + x_j$，那么有

$$M_i = \min\left\{x_i + \min_{j<x_i} g(i,j), \min_{j>x_i} h(i,j) - x_i\right\}$$

这样一来，每次只会在线段树中新建 $\log n$ 个节点(需要离散)，边数也优化至 $O(n \log n)$，复杂度为 $O(n \log^2 n + k \log k)$.

[7]这里 $a + x \to b$ 表示用 $a + x$ 去更新 b 这个变量的值，等同于 $b = \min\{a + x, b\}$.

但是这并非能做到的最好复杂度，可以这样做：在线段树上，每个节点的儿子个数增加为 d，也就是说，我们不再平分一段区间，而是将它 d 等分，这样修改时，修改的点数会减少至 $O(\log_d n)$，而询问时的区间个数会增加至 $O(d\log_d n)$. 这样，我们的点数为 $O(n\log_d n)$，而边数为 $O(nd\log_d n)$，总复杂度就是

$$O\left(\frac{n\log^2 n}{\log d}+\frac{nd\log n}{\log d}+k\log k\right).$$

如果取 $d=\log n$，那么复杂度为

$$O\left(\frac{n\log^2 n}{\log\log n}+k\log k\right).$$

其实 d 取 $O(\log n)$ 并非最优，可以证明再将 d 除一个朗伯 W 函数[8]，即 $d=O\left(\frac{\log n}{W(O(\log n))}\right)$ 时取到最优，详细请见[8]。

8 k 小生成树

8.1 最小生成树

Prim 和 Kruskal 算法是信息学竞赛中喜闻乐见的最小生成树做法，其复杂度分别为 $O(n\log n+m)$ 和 $O(m\log n+m\alpha(n))$.

但是也有一些比两者更优的算法，可以做到期望线性，或者最坏 $O(m\alpha(m,n))$.

8.2 次小生成树

首先引用唐文斌在冬令营 2012 上的讲课：

- 先求出最小生成树 MST；
- 然后枚举不在 MST 上的边 (u,v)，若将 (u,v) 替换掉 MST 上节点u 与节点 v 之间权值最大的边，那么得到的生成树的权值为 $w(MST)+w(u,v)-maxw(u,v)$;
- 取最小值即得到次小生成树。

[8]朗伯 W 函数满足：$z=W(z)e^{W(z)}$.

意思就是说，我们使用一条非树边去替换最小生成树上的树边，并使其仍是生成树。次小生成树一定是这些生成树之一。具体的证明请见下文。

而 $maxw(u,v)$ 可以使用倍增在 $O(m\log n)$ 的复杂度内求得。所以求次小生成树的复杂度为 $O(m\log n)$.

8.3 k 小生成树

这里先给出一个 $O(m\log n+k^2)$ 的做法。

8.3.1 收缩必要边

对于一张无向图 G 和 G 上的一条边 $e=(x,y)$，定义收缩图 $G\cdot e$ 为点集为 $V(G)-y$、边集为 $c_e(E(G))$ 的图。其中函数 c_e 表示去掉所有 (x,y) 边，并将所有 (y,z) 边改为 (x,z).

引理8.1. *T 为 G 的一棵生成树，e 是 T 上的任意一条边，$c_e(T)$ 是 $G\cdot e$ 的最小生成树当且仅当 T 是 G 的最小的包含 e 的生成树。*

Proof. 可以直接根据定义得出。 □

引理8.2. *v 是 G 中任意一个点，e 是所有与 v 相连的边中最小的一条。令 T 为 $G\cdot e$ 的最小生成树，那么 $T+e$ 即为 G 的最小生成树。*

Proof. 令 $e=(u,v)$，若 e 不在 G 的最小生成树中，那么 v 到 u 必有路径，设其与 v 相连的边为 f，那么 $\ell(f)\geq\ell(e)$，由于将 f 换成 e 之后仍为生成树，所以 e 必在 G 的其中一棵最小生成树中。

根据引理 8.1，$T+e$ 为 G 的最小的包含 e 的生成树，所以 $T+e$ 不大于 G 的最小生成树，所以 $T+e$ 是 G 的最小生成树。 □

对于一条生成树上的边 e，它将生成树分成 T_1,T_2，定义 $r_G(e)$ 为除 e 以外最小的跨越 T_1 和 T_2 的边。

引理8.3. *对于任意一条最小生成树 T 上的边 e，如果 $G-e$ 连通，那么 $T-e+r_G(e)$ 是 $G-e$ 的最小生成树。*

Proof. 可以通过对点数数学归纳得出。每次收缩一个叶节点。 □

引理8.4. 给出 G 的最小生成树 T，所有 T 上的边 e 的 $r_G(e)$ 能在 $O(m\log n)$ 的时间复杂度内计算得到。

Proof. 枚举所有非树边 $e=(u,v)$，在 T 上 u 到 v 的路径覆盖 e，最后求每条树边上被覆盖的最小非树边即可。

可以暴力使用动态树来做，也可以将所有非树边按边长从小到大排序，暴力覆盖路径，并将已覆盖的路径使用并查集合并，每次跳到已覆盖的最高点。

□

引理8.5. 给出 G 的最小生成树和所有树边的 r_G，我们可以在线性时间内得出 $n-k$ 条一定在 k 小生成树中的边。

Proof. 对于一条树边 e，令 $\ell'(e)=\ell(r_G(e))-\ell(e)$. 换句话说，$\ell'$ 即为将 e 删掉对图的最小生成树的额外代价。然后，我们可以在 $O(n)$ 时间内得到前 $k-1$ 小的 ℓ'，设其余的树边的集合为 S.

对于 S 中任意一条边 e，至少存在 $k-1$ 条边 e' 满足 $\ell(T-e'+r_G(e'))\le\ell(T-e+r_G(e))$，所以至少有 k 棵生成树大小不劣于 $\ell(T-e+r_G(e))$. 根据引理 8.3 得到，至少有 k 棵生成树大小不劣于任意一棵不包含边 e 的生成树。所以 e 必在 k 小生成树中。 □

引理8.6. 给出图 G 的最小生成树 T，在 $O(m\log n)$ 的时间内，我们可以找到一个边集 S 和一张 k 个点的图 G'，使得 G 的 k 小生成树恰好为 G' 的 k 小生成树加上边集 S.

Proof. 令 S 为引理 8.5 中所建的边集，令 $G'=G\cdot S$，即 G 收缩 S 中所有的边。

□

8.3.2 去掉无用边

与 $r_G(e)$ 类似的，对非树边 $e=(u,v)$，定义 $R_G(e)$ 为树上 u 到 v 路径上的最大边。

引理8.7. 如果 T 是图 G 的最小生成树，那么 $c_e(T-R_G(e))$ 为 $G\cdot e$ 的最小生成树。

Proof. 对点数进行数学归纳。每次收缩一条非 $R_G(e)$ 的树边，根据引理 8.1 可得。 □

同样，R_G 也可以被快速算出。

引理8.8. *给出 G 的最小生成树 T，所有非树边 e 的替换边 $R_G(e)$，能在 $O(m\log n)$ 的复杂度内求出。*

Proof. 即求树上 u 到 v 路径最大值，使用倍增在 $O(m\log n)$ 的复杂度内得到。 □

引理8.9. *给出 G 的最小生成树和所有非树边的 R_G，我们可以在线性时间内得出 $m-n-k$ 条一定不在在 k 小生成树中的非树边边。*

Proof. 定义 $L(e)$ 为 $\ell(e)-\ell(R_G(e))$. 与之前类似，令 f 为所有非树边中 L 第 $k-1$ 小的边。对于任意一条边 e，若 $L(e)>L(f)$，那么至少有 k 棵生成树的大小不超过 $T-R_G(e)+e$，因此也不超过任意包含边 e 的生成树。所以 e 一定不在 G 的 k 小生成树中。 □

这样，我们就将 n 减少至 k，m 减少至 $2k-2$，所以之后，我们默认 $n\le k, m\le k-1$.

对于 $k=2$ 的情况，n,m 均可减少至 2，所以之前次小生成树的做法正确性可以直接推得。

8.3.3 构建 P

定义 P 中一个节点为 R，它由三部分组成：(生成树 T，强制在生成树中的边集 I，强制不在生成树中的边集 X)。

首先给出第一种建 P 的方法(如果对 P 的定义有些遗忘或模糊，请回顾 5.2 节)：

起始点 *source* 为 $R_1=(T_1,\varnothing,\varnothing)$，$T_1$ 为 G 的最小生成树。

对于一个 $R_i=(T_i,I_i,X_i)$，枚举下一次进行的替换，$T_i-e+r_{G-X_i}(e)$，其中 $e\in T_i-I_i$. 对这些替换按长度进行排序，设排序后被替换边分别为 $e_1,e_2,\cdots\cdots$ 那么 R_i 扩展出节点

$$(T_i-e_j+r_{G-X_i}(e_j),I_i+e_1+e_2+\cdots+e_{j-1},X_i+e_j).$$

根据 P 的定义，需要证明所有 R_i 与所有可行的生成树一一对应，且图满足堆性质。

首先证明所有 R 与所有可行的生成树一一对应：

Proof. 令 R 对应其定义中的生成树 T. 现只需证明一棵生成树有且仅有一个 R 与之对应。

先证明一棵生成树 T 至少存在一个 R 与之对应：设 R_i 为当前的生成树，满足 $I_i \subseteq T$ 且 $X_i \cap T = \varnothing$. 一开始 $i = 1$，即 $R_i = R_1 = (T_1, \varnothing, \varnothing)$，满足条件。如果 $T = T_i$，那么我们就找到了 R_i 与 T 对应，否则由于 $T \neq T_i$，所以 R_i 中至少有一个儿子 R_j 满足 $I_j \subseteq T$ 且 $X_j \cap T = \varnothing$，将 R_i 变为 R_j 重复之前即可。由于 $|I_i| + |X_i|$ 每次至少增加 1，所以这个过程不会无限制地进行下去，所以一定能找到一个 R 与 T 对应。

再证明一棵生成树 T 不会与多个 R 相对应，即每个 R 对应的 T 互不相同：对于一个节点 R，设其儿子按顺序分别为 $R_1, R_2, \cdots\cdots$ 那么由于 $X_i = X + e_i$ 且 $e_i \in T$，所以 T 与 R_i 及其子树内所有点对应的生成树均不相同。对于节点 R_i, R_j，假设 $i < j$，则 $e_i \in X_i$，$e_i \in I_j$，所以 R_i 及其子树内所有点的生成树与 R_j 及其子树内所有点的生成树不相同。对 $|I_i| + |X_i|$ 进行数学归纳，通过 R_i 为根的子树内所有节点的生成树互不相同，加上之前的即可得出 R 为根的子树内所有节点的生成树互不相同。 □

再证明图满足堆性质：

Proof. 对于 I_i 中的所有边，我们视其边权为 $-\infty$；同理对于 X_i 中的所有边，视其边权为 $+\infty$. 那么对于 P 中节点 R，满足 T 是改边权之后图的最小生成树。

数学归纳证明，首先 R_1 满足。考虑 P 中节点 R_i，若 R_i 满足，那么对于 $I_i + e_1 + e_2 + \cdots + e_{j-1}$，$T_i - e_j + r_G(e_j)$ 为次小生成树，所以对于 $I_i + e_1 + e_2 + \cdots + e_{j-1}, X_i + e_j$，$T_i - e_j + r_G(e_j)$ 为最小生成树，所以 R_j 满足。

由于随着边集 I 增大，最小生成树不会变优，而 X 增大会使最小生成树变劣，因此图 P 满足堆性质。 □

第二种方法是对第一种方法的优化，也就是使用左儿子右兄弟的形式来减少单个点的出度，详细见下文。

8.3.4 算法流程

首先，求出最小生成树，设其为 T_1，令 $R_1 = (T_1, \varnothing, \varnothing)$. 如图 10 中，实线部分为最小生成树，虚线部分为非树边。

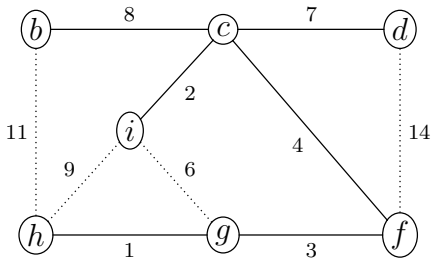


Figure 10

假设现在已经求出前 k 优解：$R_1, R_2, \ldots, R_k$. 找一个最小的替换 (i, e, f)，即 $T_i - e + f$ 仍然是一棵树且 $\ell(T_i - e + f)$ 在所有替换中是最小的，其中 e, f 需要满足 $e \notin I_i$ 且 $f \notin X_i$.

那么 $R_{k+1} = (T_i - e + f, I_i, X_i + e)$，并将 R_i 改为 $(T_i, I_i + e, X_i)$. 如图 11，将 (c, f) 替换成 (i, g)，那么 T_2 中 (c, f) 就被列入 X 集合无法再出现在树中，T_1 中 (c, f) 将强制在树中，所以我们用两条线来表示。

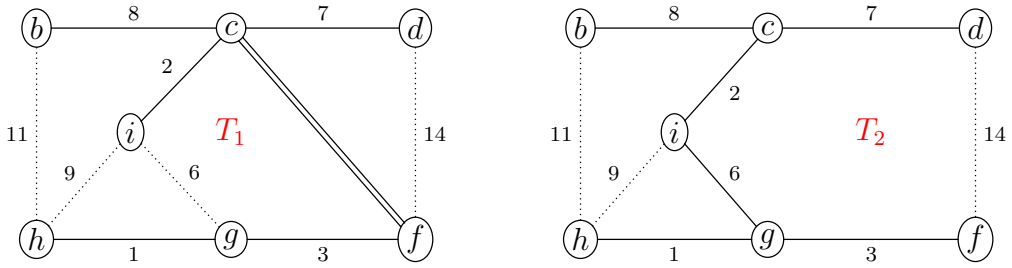


Figure 11

下一步，如图 12，我们将把 T_1 中的 (b, c) 替换成 (b, h) 得到 T_3.

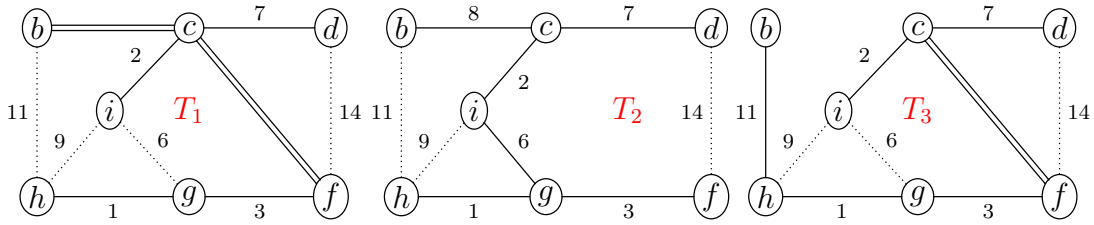


Figure 12

再下一步，如图 13，我们将把 T_1 中的 (g, f) 替换成 (i, g) 得到 T_4.

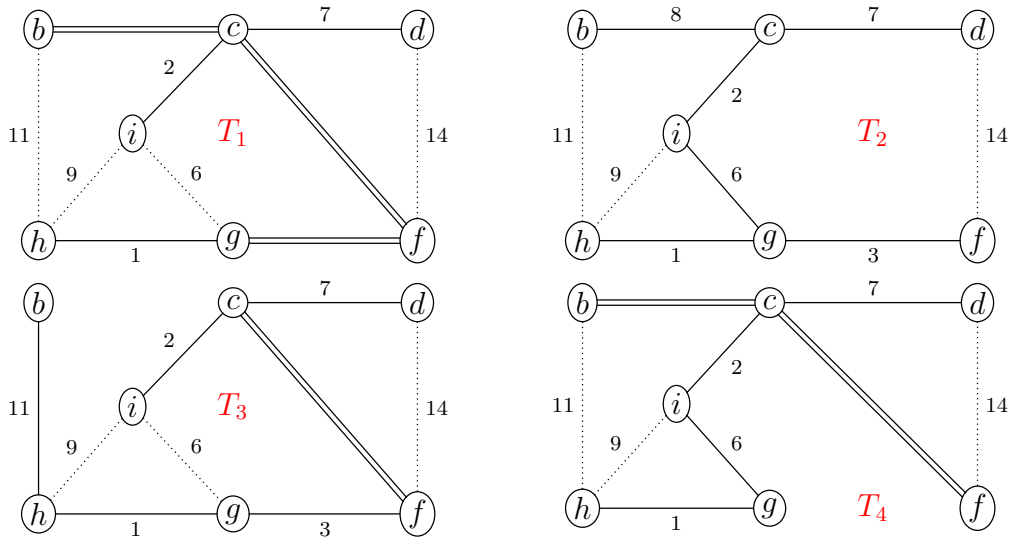


Figure 13

现在问题是：如何快速的得到最小的替换。有两种方法：

第一种：维护所有树边的 r_G 值。如果暴力维护，根据引理 8.4 可知复杂度为 $O(km\log n)$. 可以预先将所有边排序，将其优化至 $O(km\alpha(n))$. 但是无法做到 $O(km)$.

第二种：维护所有非树边的 R_G 值。如果暴力维护，根据引理 8.8 可知复杂度为 $O(km\log n)$，现将其优化至 $O(km)$.

I 集合增加一条边 e 时，只有 u 到 v 树上路径经过 e 的非树边 (u,v) 的 R_G 值才会改变。因此假设以 e 的某个端点为根，那么只有最近公共祖先为根的路径的 R_G 值才会改变。要判断最近公共祖先是否为根，只需把根删掉，判断是否联通即可；如果已知最近公共祖先为根，那么 R_G 值的计算就非常简单了。

将 e 换成 f，并在 X 集合中加入 e 的做法如下：

设 $e=(u_0,v_0)$，去掉 e，树被划分成 T_1,T_2，设 u_0 在 T_1 中，v_0 在 T_2 中。设在 T_1 中 u_0 到 f 的路径为 $u_0,u_1,\ldots,u_p$，在 T_2 中 v_0 到 f 的路径为 $v_0,v_1,\ldots,v_q$，其中 $f=(u_p,v_q)$.

如果将 e 换成 f，那么只有横跨 T_1,T_2 的非树边在树上的路径才会改变。更具体地，横跨 T_1,T_2 的非树边边在树上的路径只改变在环 $u_0\to u_1\to\cdots\to u_p\to v_q\to\cdots\to v_1\to v_0\to u_0$ 上的部分(实际上是取反)。所以，只需更改所有横跨 T_1,T_2 的非树边的 R_G 值。

对树进行一次遍历，求出每个点距离它最近的环上的点。对于一条非树边 $e=(u,v)$，设距离 u,v 最近的环上的点分别为 u',v'，那么 $R_G(e)$ 将由五部分构成：$u\sim u'$, $u'\sim u_p$, f, $v_q\sim v'$, $v'\sim v$. 这些都可以通过维护前缀信息在 $O(n+m)$ 时间内得到。所以复杂度为 $O(km)$.

所以我们有：

定理8.1. *k 小生成树能在时间复杂度 $O(m\log n+k^2)$ 复杂度内求出。*

Proof. 最小生成树能在该复杂度内完成。使用引理 8.6 在该复杂度内将点数和边数都减少为 $O(k)$，再根据本节中的方法，在 $O(k^2)$ 内得出 k 小生成树。 □

最后偷偷说一句：如果存在一种数据结构能支持快速并可持久地更换树边、更改边权，并维护树上最小的替换，那么就可以在该数据结构 $O(k)$ 次操作的复杂度内得出 k 小生成树。在 [15] 中就有这个做法，它使用的数据结构是可持久化 topology tree.

9 k 小简单路径

k 小简单路径，是求在一张有向带权图 G 中，从起点 s 到终点 t 的**不**可重复经过同一点的不严格递增的第 k 短路。下面我们将默认图 G 不包含负环。

虽然它与 k 短路的定义只相差了一个“不”字，但是求解 k 小简单路径与求解 k 短路的方法相差很大，反而与 k 小生成树的解法有异曲同工之妙，因此作者选择在 k 小生成树之后提出 k 小简单路径的做法。

9.1 构建 P

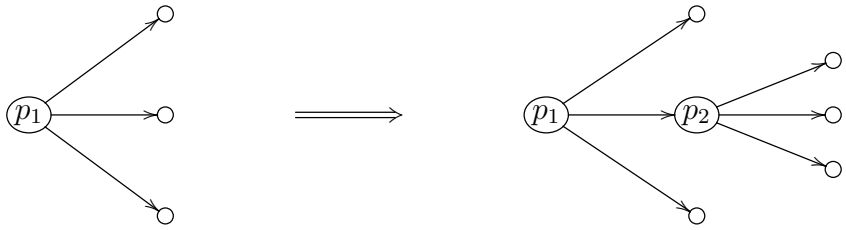
分析之前 k 小生成树的构造，P 是一棵树，P 中节点 v 始终优于所有在其子树内的点。

换一个角度，就可以这样理解：

- 初始有一个集合 $\mathcal{T}$ 表示所有生成树的全集；
- 开始找到 $\mathcal{T}$ 中的最优解，即最小生成树 T_1；
- 将 $\mathcal{T}-\{T_1\}$ 分成 m 个集合 $\mathcal{T}_1,\mathcal{T}_2,\ldots,\mathcal{T}_m$，使得这 m 个集合中每个集合的最优解容易求出；

- 这样得到次小生成树 T_2，假设 $T_2 \in \mathcal{T}_j$，那么将 $\mathcal{T}_j - \{T_2\}$ 分成若干集合与 $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_{j-1}, \mathcal{T}_{j+1}, \ldots, \mathcal{T}_m$ 一起继续之前操作得到第三小生成树 T_3……

所以类似的，可以这样构建 k 小简单路径的图 P：



- 初始有一个集合 $\mathcal{P}$ 表示所有 s 到 t 的简单路径的全集；
- 开始找到 $\mathcal{P}$ 中的最优解，即 s 到 t 的最短路 p_1；
- 将 $\mathcal{P} - \{p_1\}$ 分成 m 个集合 $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_m$，使得这 m 个集合中每个集合的最优解容易求出；
- 这样得到次小简单路径 p_2，假设 $p_2 \in \mathcal{P}_j$，那么将 $\mathcal{P}_j - \{p_2\}$ 分成若干集合与 $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_{j-1}, \mathcal{P}_{j+1}, \ldots, \mathcal{P}_m$ 一起继续之前操作得到第三小生成树 p_3 ……

现在的问题就是如何将某个集合进行划分，使得划分之后所得的集合的最优解容易求出。在 k 小生成树中，我们使用了强制包含和强制排除的方法，在求 k 小简单路径时也是相同的：

令图 P 中一个节点 R 由三部分组成：(当前简单路径 p，强制包含的边集 I，强制不包含的边集 X)。其中令 $p = (v_1 = s, v_2, \ldots, v_{|p|-1}, v_{|p|} = t)$，保证 $I = \{(s, v_2), (v_2, v_3), \ldots, (v_{q-1}, v_q)\}$ 且 $q \leq |p| - 1$，X 中全是从 v_q 出发的边。

R 共扩展出 $|p| - q$ 个点，设其为 $R_i = (p_i, I_i, X_i)$，那么有

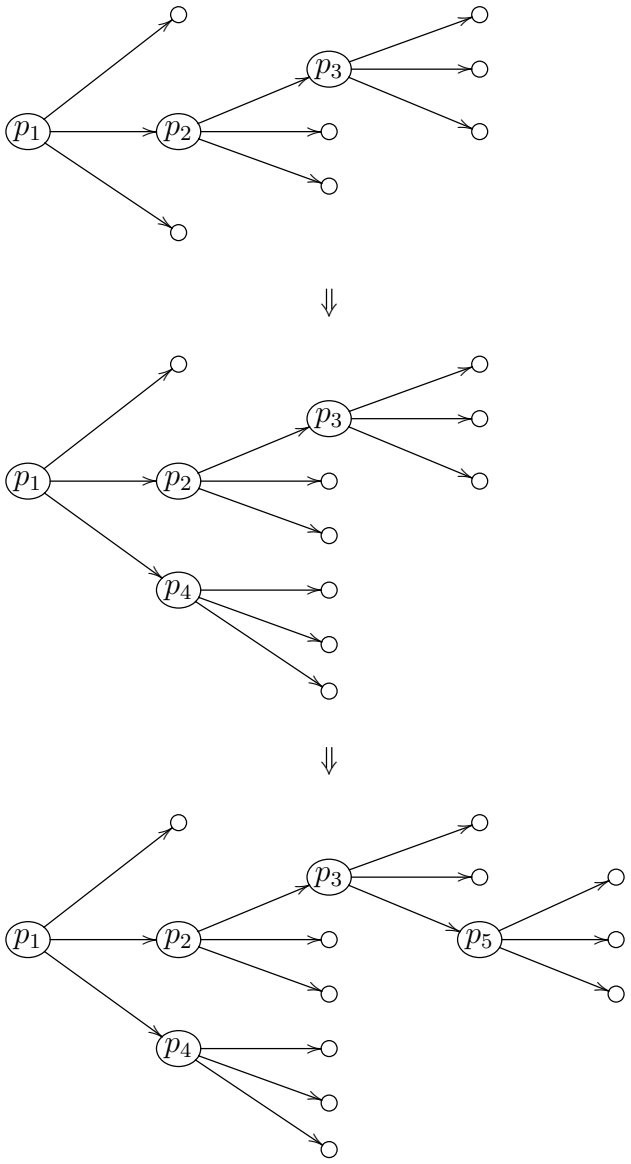
$$I_1 = I, X_1 = X + \{(v_q, v_{q+1})\},$$

对于 $i > 1$ 有

$$I_i = I_{i-1} + \{(v_{q+i-2}, v_{q+i-1})\}, X_i = \{(v_{q+i-1}, v_{q+i})\},$$

p_i 是条件 I_i, X_i 下的最短简单路径。

很可惜，我们无法像 k 小生成树那样使用左儿子右兄弟的形式来减少单个节点的出度，因为我们无法做到快速地判断兄弟之间的大小关系。



9.2 算法流程

首先，如果所有边权非负，就使用 Dijkstra 算法求出 s 到 t 的最短路；否则使用 Bellman-Ford 或 SPFA 算法求出 s 到所有点的最短路，将 (u,v) 的边权 $\ell(u,v)$ 替换为 $\ell(u,v)+dist(s,u)-dist(s,v)$，类似于之前 k 短路算法中的 δ. 此时 s 到 t 的任意一条路径长度需要加上原先 s 到 t 的最短路，与性质 6.2 类似。将 (s 到 t 的最短路,$\varnothing,\varnothing$) 加入优先队列中。

每次从优先队列中弹出最小的点 $R = (p, I, X)$，并将它能扩展出的点 $R_i = (p_i, I_i, X_i)$ 全部加入优先队列中。

条件 I_i, X_i 下的最短简单路径 p_i 可以通过 Dijkstra 算法在 $O(n \log n + m)$ 复杂度内得到。由于 k 个点最多扩展出 kn 个点，所以需要计算 kn 次最短路，所以复杂度为 $kn(n \log n + m)$.

对于优先队列的实现，可以直接使用堆，时间复杂度为 $O(kn \log kn)$，空间复杂度可以优化到 $O(kn)$.

因此我们有

定理9.1. *在一张有向带权图 G 中，从起点 s 到终点 t 的不可重复经过同一点的不严格递增的第 k 短路的长度，可以在 $O(kn(m + n \log n))$ 的复杂度内得到。*

10 感谢

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、邵红祥老师、游光辉老师、董烨华老师多年来给予的关心和指导。

感谢国家集训队教练胡伟栋和余林韵的指导。

感谢清华大学的徐捷、鲁逸沁、裘捷中、梁佳文和上海交通大学的蒋舜宁学长对我的帮助。

感谢绍兴一中的何奇正、郑钟屹、徐正杰同学对我的帮助和启发。

感谢绍兴一中的董宏华、张恒捷、王鉴浩、郭雨等同学对我的帮助和启发。

感谢镇海中学的杜瑜皓同学为本文审稿。

感谢其他对我有过帮助和启发的老师和同学。

感谢我的父母二十年如一日无微不至的关心和照顾。

参考文献

[1] 维基百科，最优化问题。

[2] Wheeler，PulleyTautLine，TopCoder Algorithm Problem Set Analysis.

[3] 许昊然，浅谈数据结构题的几个非经典解法，2013 集训队论文。

[4] 吕凯风，带插入区间 K 小值系列题解。

[5] Thomas H.Cormen, Charles E.Leiserson, Ronald L.Rivest, Clifford Stein, Introduction to Algorithms.

[6] D. Eppstein. Finding the k shortest paths. *SIAM J. Computing*, 28(2):652–673, 1998.

[7] 俞鼎力，堆的可持久化和 k 短路，2014 冬令营营员讨论。

[8] 俞鼎力，无聊的邮递员解题报告，2014 集训队第二次作业。

[9] WIKIPEDIA，Prim's algorithm.

[10] WIKIPEDIA，Kruskal's algorithm.

[11] WIKIPEDIA，Minimum spanning tree.

[12] 唐文斌，图论专题之生成树，2012 冬令营讲课。

[13] 刘汝佳，黄亮，算法艺术与信息学竞赛。

[14] D. Eppstein. Finding the k smallest spanning trees. *BIT* 32 (2): 237 - 248.

[15] Frederickson, Greg N. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees, *SIAM J. Computing*, 26(2):484 - 538, 1997.

[16] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids.*

[17] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17:712-716, 1971.