0-1 分数规划

一、分数规划

分数规划的一般形式：

$$\min \lambda = f(\mathbf{x}) = \frac{a(\mathbf{x})}{b(\mathbf{x})} (\mathbf{x} \in S)$$
$$s.t. \quad \forall \mathbf{x} \in S, b(\mathbf{x}) > 0$$

其中，解向量 $\mathbf{x}$ 在解空间 S 内， $a(\mathbf{x}), b(\mathbf{x})$ 都是连续的实值函数。

一般解决分数规划问题的实用方法为参数搜索法，即对答案进行猜测，再验证该猜测值的最优性，将最优化问题转化为判定性问题或其他更容易求解的最优化问题。由于分数规划模型的特殊性，使得能够构造另外一个由猜测值 λ 作为自变量的相关问题，且该问题的解满足一定的单调性，或其他的可以减小参数搜索范围的性质，从而逼近答案。

假设 $\lambda^* = f(\mathbf{x}^*)$ 为该规划的最优解，有：

$$\lambda^* = f(\mathbf{x}^*) = \frac{a(\mathbf{x}^*)}{b(\mathbf{x}^*)}$$
$$\Rightarrow \quad \lambda^* \cdot b(\mathbf{x}^*) = a(\mathbf{x}^*)$$
$$\Rightarrow \quad 0 = a(\mathbf{x}^*) - \lambda^* \cdot b(\mathbf{x}^*)$$

由上面的形式构造一个新函数 $g(\lambda)$：

$$g(\lambda) = \min_{\mathbf{x} \in S} \{(a(\mathbf{x}) - \lambda \cdot b(\mathbf{x})\}$$

这个函数是一个非分式的规划。先来看看它本身的性质：

单调性： $g(\lambda)$ 是一个严格递减函数，即对于 $\lambda_1 < \lambda_2$ ，一定有 $g(\lambda_1) > g(\lambda_2)$

证明：设解向量 $\mathbf{x}_1$ 最小化了 $g(\lambda_1)$。则

$$\begin{aligned} g(\lambda_1) &= \min_{\mathbf{x}\in S}\{(a(\mathbf{x}) - \lambda_1 \cdot b(\mathbf{x})\} \\ &= a(\mathbf{x}_1) - \lambda_1 \cdot b(\mathbf{x}_1) \\ &> a(\mathbf{x}_1) - \lambda_2 \cdot b(\mathbf{x}_1) \\ &\geq \min_{\mathbf{x}\in S}\{(a(\mathbf{x}) - \lambda_2 \cdot b(\mathbf{x})\} = g(\lambda_2) \end{aligned}$$

最后一步说明 $g(\lambda_1)$ 上最小解代入 $g(\lambda_2)$ 后不一定还是最小解，甚至有更小解。

有了单调性，就意味着我们可以采用二分搜索的方法来逼近答案。但我们还不知道我们所求的目标是什么，下面考察构造出的新函数与原目标函数的最优解关系：

Dinkelbach 定理：设 λ^* 为原规划的最优解，则 $g(\lambda) = 0$ 当且仅当 $\lambda = \lambda^*$.

证明：必要性：$\lambda = \lambda^* \Rightarrow g(\lambda) = 0$：

设 $\lambda^* = f(\mathbf{x}^*)$ 为原规划最优解，则 $g(\lambda^*) = 0$

对于 $\forall \mathbf{x} \in S$，都不会比 $\mathbf{x}^*$ 优：

$$\lambda^* = \frac{a(\mathbf{x}^*)}{b(\mathbf{x}^*)} \leq \frac{a(\mathbf{x})}{b(\mathbf{x})} \quad \Rightarrow \quad a(\mathbf{x}) - \lambda^* \cdot b(\mathbf{x}) \geq 0$$

然而 $\mathbf{x}^*$ 可以取到这个下限。

$$\lambda^* = \frac{a(\mathbf{x}^*)}{b(\mathbf{x}^*)} \quad \Rightarrow \quad a(\mathbf{x}^*) - \lambda^* \cdot b(\mathbf{x}^*) = 0$$

故 $g(\lambda^*)$ 的最小值由 $\mathbf{x}^*$ 确定，$g(\lambda^*) = 0$。

充分性：$g(\lambda) = 0 \Rightarrow \lambda = \lambda^*$.

若存在一个解 $\mathbf{x}$ 使得 $g(\lambda) = 0$ 则 $\lambda = f(\mathbf{x})$ 为原规划最优解。

反证法。反设存在一个解 $\lambda' = f(\mathbf{x}')$，它是比 $\lambda = f(\mathbf{x})$ 更优的解。

$$\lambda' = \frac{a(\mathbf{x}')}{b(\mathbf{x}')} < \lambda$$
$$\Rightarrow \quad a(\mathbf{x}') - \lambda \cdot b(\mathbf{x}') < 0$$

这时 $\mathbf{x}'$ 应该使得 $g(\lambda) < 0$，这与 $g(\lambda) = 0$ 矛盾。

由上面的性质与定理容易推得：

设 λ^* 为该规划的最优解，则

$$\begin{cases} g(\lambda) = 0 \Leftrightarrow \lambda = \lambda^* \\ g(\lambda) < 0 \Leftrightarrow \lambda > \lambda^* \\ g(\lambda) > 0 \Leftrightarrow \lambda < \lambda^* \end{cases}$$

有了该推论我们就可以对最优解进行二分查找，每次需要计算的是一个新的非分数规划，这就将原问题简化了，以便我们能设计出其他有效的算法解决这个问题。算法的复杂度是二分迭代的次数与每次解决 $g(\lambda)$ 的复杂度的积。

二、0-1 分数规划

分数规划的一个特例是 0-1 分数规划，就是其解向量 $\mathbf{x}$ 满足 $\forall x_i \in \{0,1\}$.形式化的定义：

$$\min \lambda = f(\mathbf{x}) = \frac{\sum_i a_i x_i}{\sum_i b_i x_i} (\mathbf{x} \in \{0,1\}^n)$$
$$s.t. \quad \mathbf{b} \cdot \mathbf{x} > 0$$

解决 0-1 分数规划与普通的分数规划一样，也可以采用二分搜索，构建新的规划函数求解的算法。

例如：给定 n 个二元组 (a_i, b_i)，求选出 k 个二元组，使得剩下的 $\sum a_i$ 与 $\sum b_i$ 比率最

大。即求 $\max \frac{\sum_i a_i x_i}{\sum_i b_i x_i}, x_i \in \{0,1\}$ 。

分析：二分答案 $r = \frac{\sum a_i x_i}{\sum b_i x_i}$，则最优解满足 $\sum a_i x_i - \sum b_i x_i r = 0$

且任意的 $\sum a_i x_i - \sum b_i x_i \max\{r\} \le 0$，因此我们求解 $g(r) = \sum a_i x_i - \sum b_i x_i r$ 的最优（大）值即可判断出答案的范围：$g(r) > 0$ 则答案偏小，$g(r) < 0$ 答案偏大。

回到原问题中，我们将每个二元组的价值设为 $a_i - b_i \cdot r$，之后贪心取最大的 k 个元素即可求得 $g(r)$ 的最优值，进而利用二分确定答案。

练习题

POJ 2976 ; POJ 2728 ; POJ 3621 ; POJ 3155

上述内容参考：

1.最小割模型在信息学竞赛中的应用，胡伯涛

高斯消元

一、简介

高斯消元法，是线性代数中的一个算法，可用来求解线性方程组，并可以求出矩阵的秩，以及求出可逆方阵的逆矩阵。

高斯（Gauss）消元法的基本思想是：通过一系列的加减消元运算，也就是代数中的加减消去法，将方程组化为上三角矩阵；然后，再逐一回代求解出 x 向量。

二、线性方程组

设含有 n 个未知量、有 m 个方程式组成的方程组

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \end{cases}$$

其中系数 $a_{i,j}$，常数 b_j 都是已知数，x_t 是未知量（也称为未知数）。当右端常数项 b_1，b_2，…，b_m 不全为 0 时，称方程组为非齐次线性方程组；当 $b_1 = b_2 = \cdots = b_m = 0$ 时，即

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = 0 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = 0 \\ \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = 0 \end{cases}$$

称为齐次线性方程组。

由 n 个数 k_1，k_2，…，k_n 组成的一个有序数组（k_1，k_2，…，k_n），如果将它们依次代入方程组中的 x_1，x_2，…，x_n 后，每个方程都变成恒等式，则称这个有序数组（k_1，k_2，…，k_n）为方程组的一个解。显然由 x_1=0，x_2=0，…，x_n=0 组成的有序数组（0, 0, …, 0）是齐次线性方程组的一个解，称之为齐次线性方程组的零解，而当齐次线性方程组的未知量取值不全为零时，称之为非零解。

利用矩阵来讨论线性方程组的解的情况或求线性方程组的解是很方便的。因此，我们先给出线性方程组的矩阵表示形式。

非齐次线性方程组的矩阵表示形式为：

$$AX = B$$

其中

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad B = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

称 A 为方程组的系数矩阵，X 为未知矩阵，B 为常数矩阵。将系数矩阵 A 和常数矩阵 B 放在一起构成的矩阵

$$[A \quad B] = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & b_m \end{bmatrix}$$

称为方程组的增广矩阵。

齐次线性方程组的矩阵表示形式为：$AX = O$

三、算法流程

定理：若用初等行变换将增广矩阵$[A \quad B]$化为$[C \quad D]$，则 $AX = B$ 与 $CX = D$ 是同解方程组。

由定理可知，求方程组的解，可以利用初等行变换将其增广矩阵$[A \quad B]$化简。通过初等行变换可以将$[A \quad B]$化成阶梯形矩阵。因此，我们得到了求解线性方程组的一般方法：

用初等行变换将方程组的增广矩阵$[A \quad B]$化成阶梯形矩阵，再写出该阶梯形矩阵所对应的方程组，逐步回代，求出方程组的解。因为它们为同解方程组，所以也就得到了原方程组的解。这种方法被称为高斯消元法。

初等行变换：1.交换两行，2.将某一行所有元素乘上一个非 0 数，3.将某行所有元素的 k 倍加到另一行对应元素上去。相应的，也有三个初等列变换。

利用初等行变换，我们可以简单的将告诉消元法描述如下：

1.枚举每一行，将行首起第一个非 0 元素当做此行的主元。

2.利用初等行变换，将此行的主元置为 1.

3.利用初等行变换，将其他行中此元素消去，即置为 0.

4.算法结束后，每行的常数项列上的数即为对应行主元变量的解。

实际上这个消元过程与我们手动解方程组的过程无异，即选择某个方程中的某个元素，利用运算将其它方程中这项元素的系数消去。最后将方程组消成 n 个一元一次方程。

例子：

解线性方程组 $\begin{cases} x_1+2x_2-3x_3=4 \\ 2x_1+3x_2-5x_3=7 \\ 4x_1+3x_2-9x_3=9 \\ 2x_1+5x_2-8x_3=8 \end{cases}$

解　利用初等行变换，将方程组的增广矩阵$[A \quad B]$化成阶梯阵，再求解。即

$$[A \quad B]=\begin{bmatrix} 1 & 2 & -3 & 4 \\ 2 & 3 & -5 & 7 \\ 4 & 3 & -9 & 9 \\ 2 & 5 & -8 & 8 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & -3 & 4 \\ 0 & -1 & 1 & -1 \\ 0 & -5 & 3 & -7 \\ 0 & 1 & -2 & 0 \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} 1 & 2 & -3 & 4 \\ 0 & -1 & 1 & -1 \\ 0 & 0 & -2 & -2 \\ 0 & 0 & -1 & -1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & -3 & 4 \\ 0 & -1 & 1 & -1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} 1 & 2 & 0 & 7 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

一般解为

$$\begin{cases} x_1=3 \\ x_2=2 \\ x_3=1 \end{cases}$$

代码：

```
inline void Gauss() {
    int l; double temp;
    for (int i = 0; i < n; ++i) {
        l = i;  // 选取绝对值最大的那个元素当做主元，减小精度问题
        for (int j = i + 1; j < n; ++j)
            if (fabs(f[l][i]) < fabs(f[j][i])) l = j;
        if (i != l)
            for (int j = i; j <= n; ++j)
```

```
                swap(f[l][j], f[i][j]);
        temp = f[i][i];
        for (int j = i; j <= n; ++j) f[i][j] /= temp;
        for (int j = 0; j < n; ++j)
            if (j != i) {
                temp = f[j][i];
                for (int k = i; k <= n; ++k) f[j][k] -= f[i][k] * temp;
            }
    }
    return ;
}
```

若消元过程中出现(0,0,0,...,0,a)的情况，即某行中常数项元素不为 0，但其他系数元素均为 0，则该方程组无解；而若某行出现全 0 的情况，则该行主元元素可取任意值，即无穷解，这些变量称为自由元。

练习题

BZOJ 1013 ; BZOJ 3143 ; BZOJ 2337

BZOJ 1013 代码：

```
#include <algorithm>
#include <iostream>
#include <cstring>
#include <cstdio>
#include <cmath>
using namespace std;

const int N = 13;
int n, m;
double c[N][N], f[N][N], ans[N];

inline void Gauss() {
    for (int i = 1; i <= n; ++i) {
        int l = i;
        for (int j = l + 1; j <= n; ++j)
            if (fabs(f[l][i]) < fabs(f[j][i])) l = j;
```

```
        if (l != i)
            for (int j = i; j <= m; ++j)
                swap(f[l][j], f[i][j]);
        for (int j = i + 1; j <= n; ++j) {
            double temp = f[j][i] / f[i][i];
            for (int k = i; k <= m; ++k)
                f[j][k] = f[j][k] - f[i][k] * temp;
        }
    }
    for (int i = n; i >= 1; --i) {
        double t = f[i][m];
        for (int j = n; j > i; --j)
            t -= ans[j] * f[i][j];
        ans[i] = t / f[i][i];
    }
    return ;
}

int main() {
    scanf("%d", &n); m = n + 1;
    for (int i = 0; i <= n; ++i)
        for (int j = 1; j <= n; ++j)
            scanf("%lf", &c[i][j]);
    for (int i = 1; i <= n; ++i) {
        int j = i - 1; double d = 0;
        for (int k = 1; k <= n; ++k) {
            f[i][k] = (c[i][k] - c[j][k]) * 2;
            d += c[i][k] * c[i][k] - c[j][k] * c[j][k];
        }
        f[i][m] = d;
    }
    Gauss();
    for (int i = 1; i <= n; ++i) {
        if (i < n) printf("%.3lf ", ans[i]);
        else    printf("%.3lf\n", ans[i]);
    }
    return 0;
}
```

虚树

一、引入

题目大意：给定一棵 n 个节点的树，树上边权均为 1。接下来 m 次询问，每次给定树中 k_i 个特殊点，每个点会被离它最近的特殊点所控制（距离一样则被编号小的特殊点控制），求每个特殊点会控制多少个点。$m, \sum k_i \le 3\times 10^5$

分析：如果只有一次询问，我们能利用一个简单的树形 DP 来解决，即 $g[i]$ 是一个二元组 (x,y)，表示子树中离结点 i 最近的特殊点的距离与编号分别是 x,y，求出 $g[i]$ 后利用树形 DP 中换根的技巧求出类似的二元组 $f[i]$，表示整棵树中离结点 i 最近的特殊点的距离与编号分别是 x,y.

但现在询问次数很大，显然不能每次都做这样的 DP.

进一步分析我们发现，实际上每次我们并不需要求出每个点的信息，我们只需要利用在树中相邻两个特殊点的信息，就可求出答案。

例如在一条链上，某一对相邻的特殊点（这里的相邻指的是它们路径上没有其它特殊点），我们肯定能在它们的路径上找到一个分界点，使得分界点两边的点，分属两个不同的特殊点控制。

也就是说我们现在只关心树里这些特殊点的父子关系，也就是，由这些特殊点所组成的简化了的树，这棵树就叫做虚树。

更详细的定义：设原树为一棵形态固定的树 T，它有 n 个节点，它的虚树是它的一棵连通子树，或者是它的某棵连通子树，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。

二、虚树的构建方法

实际问题中，我们会像上述例子一样，通常只用求出一棵较简的虚树，使其包含给定的特殊点，我们称这些点为关键点。

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，去掉这个点后整棵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，因此我们降低一些要求。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。

而对于 dfs 序连续的三个点 x, y, z，我们有 Lca(x, z)=Lca(x, y)或 Lca(y, z)，因此只要将所有关键点按 dfs 排序，再将所有 $Lca(x_i, x_{i+1})$ 与所有 x_i 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，就可得到虚树节点之间的连边。具体的做法是，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（最右链）。当前插入点 x 时，依次检查这条路径上最后的那个点 y 是否是 x 的祖先（利用原树 dfs 序），若不是，就将 y 踢出当前路径，继续检查；否则在 x, y 间连一条边，将 x 加入路径，且 x 在虚树中的父亲即为 y.

事实上，上面这种维护最右链的方法并不需要提前求出所有 dfs 序相邻的关键点的 Lca，只需要在添加新关键点时，求出新关键点与最右链底部节点的 Lca，就可知道当前最右链有哪些节点需要被踢出，而这个新的 Lca 若不在原来的虚树中，则也需要添加进虚树中。

两种方法构造虚树的时间复杂度都为 $O(m \log n)$，构造出的虚树大小均为 $O(m)$，其中 m 是特殊点个数。

代码：

```
inline bool cmp(const int &x, const int &y) { return dfn[x] < dfn[y]; }

void build_vir() {
  top = 0;
  std::sort(vir + 1, vir + vn + 1, cmp);
  vn = std::unique(vir + 1, vir + vn + 1) - vir - 1;
  int tn = vn;
  // vir[]为虚树中节点，par[]为虚树中节点的父亲，stk[]存储当前最右链
  for (int i = 1; i <= tn; ++i) {
    int u = vir[i];
    if (!top) {
      par[u] = 0;
      stk[++top] = u;
```

```
      continue;
    }
    int lca = LCA(stk[top], u);
    while (dep[stk[top]] > dep[lca]) {
      if (dep[stk[top - 1]] < dep[lca]) par[stk[top]] = lca;
      --top;
    }
    if (lca != stk[top]) {
      vir[++vn] = lca;
      par[lca] = stk[top];
      stk[++top] = lca;
    }
    par[u] = lca;
    stk[++top] = u;
  }
  std::sort(vir + 1, vir + vn + 1, cmp);
}
```

三、例题

世界树 (BZOJ 3572)

题目大意：

给定一棵 n 个节点的树，树上边权均为 1。接下来 m 次询问，每次给定树中 k_i 个特殊点，每个点会被离它最近的特殊点所控制（距离一样则被编号小的特殊点控制），求每个特殊点会控制多少个点。 $m, \sum k_i \le 3\times10^5$

分析：

求出虚树后，在虚树上做上述 DP. 之后考虑虚树中一条边上的两个点，若它们同被一个点所控制，则它们这条边上所有被缩掉的点也都被那个点所控制；否则我们能通过计算，找到分界点，将相应的子树大小贡献给两端所属的特殊点。

网上的题解：

1. 首先，虚树中没有父亲的点 p 一定是最高的点，有大小为 n-size[p]的点与他共用最近点。

2. 每个虚树上的点和他没在虚树上的子树肯定会共用一个最近点这个我们只要用其总的 size 减去在虚树中的 size 最后统计答案即可。

3. 对于虚树某一对父子 i,fa，若他们共用一对最近点，则他们在树上的链上点也一定

共用这个最近点。

4. 若并不共用最近点，肯定有某一段与 fa 共用最近点，另一段与 i 共用最近点。因此我们需要找到这个分界点 p。由于 p 到 i 最近点的距离与 p 到 fa 最近点的距离尽可能的接近，所以有 dis(i,p)=(near[fa]-near[i]+dis(i,fa))/2，这样得出来的点是最接近的，并且考虑了不能整除的情况，但是仍然需要特判一些情况——p 到 i 与 p 到 fa 的距离一样近，我们就需要比较他们点的大小，适当的调整 p 的位置。

代码：

```
#include <algorithm>
#include <iostream>
#include <cstring>
#include <cstdio>
#include <cmath>
#include <map>

typedef std::pair<int, int> PII;
#define ST first
#define ND second
#define mp std::make_pair
#define REP(i, a, b) for (int i = (a); i < (b); ++i)
#define PER(i, a, b) for (int i = (a); i > (b); --i)
#define FOR(i, a, b) for (int i = (a); i <= (b); ++i)
#define ROF(i, a, b) for (int i = (a); i >= (b); --i)

inline int read()
{
    static char ch;
    while (ch = getchar(), ch < '0' || ch > '9');
    int res = ch - 48;
    while (ch = getchar(), ch >= '0' && ch <= '9') res = res * 10 + ch - 48;

    return res;
}

const int N = 3e5 + 3, M = 18;
const int INF = 0x3f3f3f3f;
int n, m, sal[N], ans[N];
int num, aux[N], faAux[N], val[N], delta[N];
int dep[N], dfn[N], sze[N], fa[N][M + 1];
int ecnt, adj[N], nxt[N * 2], go[N * 2];
PII f[N];

```

```
inline void addEdge(const int &u, const int &v)
{
    nxt[++ecnt] = adj[u]; adj[u] = ecnt; go[ecnt] = v;
    nxt[++ecnt] = adj[v]; adj[v] = ecnt; go[ecnt] = u;
}

inline bool cmp(const int &u, const int &v)
{
    return dfn[u] < dfn[v];
}

inline void build_tree()
{
    static int qN, que[N];
    que[qN = 1] = 1;
    dep[1] = sze[1] = 1;

    FOR(i, 1, qN)
    {
        int u = que[i];
        for (int e = adj[u]; e; e = nxt[e])
        {
            int v = go[e];
            if (fa[u][0] == v) continue;
            fa[v][0] = u;
            dep[v] = dep[u] + 1;
            que[++qN] = v;
        }
    }
    ROF(i, qN, 2)
        sze[fa[que[i]][0]] += ++sze[que[i]];

    dfn[1] = 1;
    FOR(i, 1, qN)
    {
        int u = que[i], s = dfn[u];
        for (int e = adj[u]; e; e = nxt[e])
        {
            int v = go[e];
            if (fa[u][0] == v) continue;
            dfn[v] = s + 1;
            s += sze[v];
        }
    }
```

```
78.
79.     FOR(j, 1, M) FOR(i, 1, n)
80.         fa[i][j] = fa[fa[i][j - 1]][j - 1];
81. }
82.
83. inline int LCA(int u, int v)
84. {
85.     if (dep[u] < dep[v]) std::swap(u, v);
86.     for (int d = dep[u] - dep[v], i = 0; d; ++i, d >>= 1)
87.         if (d & 1) u = fa[u][i];
88.     if (u == v) return u;
89.     ROF(i, M, 0) if (fa[u][i] != fa[v][i])
90.         u = fa[u][i], v = fa[v][i];
91.     return fa[u][0];
92. }
93.
94. inline int Jump(int u, int d)
95. {
96.     for (int i = 0; d; ++i, d >>= 1)
97.         if (d & 1) u = fa[u][i];
98.     return u;
99. }
100.
101. inline void build_auxTree(const int &aN)
102. {
103.     static int top, stk[N];
104.     stk[top = 0] = 0;
105.
106.     num = aN;
107.     std::sort(&aux[1], &aux[1] + aN, cmp);
108.
109.     FOR(i, 1, aN)
110.     {
111.         int u = aux[i];
112.         if (!top)
113.             faAux[u] = 0, stk[++top] = u;
114.         else
115.         {
116.             int Lca = LCA(stk[top], u);
117.             while (dep[stk[top]] > dep[Lca])
118.             {
119.                 if (dep[stk[top - 1]] <= dep[Lca])
120.                     faAux[stk[top]] = Lca;
121.                 --top;
```

```
            }
            if (stk[top] != Lca)
            {
                faAux[Lca] = stk[top];
                f[Lca] = mp(INF, 0);
                stk[++top] = Lca;
                aux[++num] = Lca;
            }
            faAux[u] = Lca;
            stk[++top] = u;
        }
    }
    std::sort(&aux[1], &aux[1] + num, cmp);
}

inline void solve()
{
    ROF(i, num, 2)
    {
        int u = aux[i], v = faAux[u];
        delta[u] = dep[u] - dep[v];
        PII tmp(f[u].ST + delta[u], f[u].ND);
        if (tmp < f[v]) f[v] = tmp;
    }
    FOR(i, 2, num)
    {
        int u = aux[i], v = faAux[u];
        PII tmp(f[v].ST + delta[u], f[v].ND);
        if (tmp < f[u]) f[u] = tmp;
    }

    FOR(i, 1, num)
    {
        int u = aux[i], v = faAux[u];
        val[u] = sze[u];

        if (i == 1)
        {
            ans[f[u].ND] += n - sze[u];
            continue;
        }

        int son = Jump(u, dep[u] - dep[v] - 1);
        int calc = sze[son] - sze[u];
```

```
166.         val[v] -= sze[son];
167. 
168.         if (f[u].ND == f[v].ND) ans[f[u].ND] += calc;
169.         else
170.         {
171.             int z = f[u].ST - f[v].ST + dep[u] + dep[v] + 1 >> 1;
172.             if (f[v].ND < f[u].ND && f[v].ST + z - dep[v] == f[u].ST + dep[
    u] - z)
173.                 ++z;
174.             z = sze[Jump(u, dep[u] - z)] - sze[u];
175. 
176.             ans[f[u].ND] += z;
177.             ans[f[v].ND] += calc - z;
178.         }
179.     }
180.     FOR(i, 1, num)
181.         ans[f[aux[i]].ND] += val[aux[i]];
182. }
183. 
184. int main()
185. {
186.     n = read();
187.     REP(i, 1, n) addEdge(read(), read());
188.     build_tree();
189. 
190.     for (int Q = read(); Q; --Q)
191.     {
192.         m = read();
193.         FOR(i, 1, m)
194.         {
195.             int u = read();
196.             aux[i] = sal[i] = u;
197.             f[u] = mp(0, u);
198.             ans[u] = 0;
199.         }
200.         build_auxTree(m);
201. 
202.         solve();
203. 
204.         FOR(i, 1, m)
205.             printf("%d ", ans[sal[i]]);
206.         putchar('\n');
207.     }
208.     return 0;
```

```
209. }
```

练习题

BZOJ 3572 ; BZOJ 2286

上述内容参考：

1. 线段树在一类分治问题上的应用，徐寅展

整体二分

一、引入

二分答案是常见的技巧，但在数据结构题中，这个方法往往不奏效，原因是我们常常需要预处理一些东西才能快速回答当前二分点的答案，而任何一个答案都可能被二分到，因此预处理的复杂度就不可接受了。

比如经典问题，给定一个长为n的序列，接下来有m次询问，每次询问给定l,r,k，问区间$[l,r]$内的第k大值是多少，$n,m\le10^5$。

这个问题如果使用二分法来解决，那么对于每个值，我们需要知道每个位置上的数是大于它，还是小于它。如果把大于它的值记为1，小于它的值记为-1，那么每次询问进行二分答案，我们能利用区间和（若干1和-1的和）来判定答案的大小。但询问的区间有很多，它会覆盖到每一个位置，因此每个位置都需要这样处理。这样的话相当于要对n个值建立n个序列长度为n的线段树，这个预处理复杂度是不可接受的。

当然这个预处理其实是有优秀的处理方法的，即主席树。但是我们还有一个不依赖于这个预处理的好的做法来应对这类问题，那就是下面要介绍的整体二分。

二、算法介绍

实际上整体二分不止能做像上述例子那样只有询问的问题，对于一些带有修改操作的数据结构题，整体二分也可以很好的解决。

为了规范问题，我们称当前二分的答案值为“判定标准”，每个修改，结合具体询问和判定标准，可以算出该修改对该询问的判定答案的贡献。询问的判定答案是各个修改的贡献的和，而每个询问都有一个给定的要求的判定答案，根据要求的判定答案与实际的判定答案的关系，我们可以确定询问的真实答案在当前判定标准之上还是之下。

比如上面的区间k大，要求的判定答案即为k。而如果这个问题带修改，即可能会修改某个位置上的数，那么结合判定标准，它会使得比二分的答案大的数变多或变少。

整体二分需要数据结构题满足以下性质：

(1) 询问答案具有可二分性

(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，则贡献为一个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

询问的答案具有可二分性显然是前提。我们发现，因为修改对判定标准的贡献相互独立，且贡献的值（如果有的话）与判定标准无关（这里的无关指的是值的大小无关，而不是说是否贡献无关），所以如果我们已经计算过某一些修改对询问的贡献，那么这个贡献永远不会改变，我们没有必要当判定标准改变时再次计算这部分修改的贡献，只要记录下当前的总贡献，在进一步二分时，直接加上新的贡献即可。

这样的话，我们发现，处理的复杂度可以不再与序列总长度直接相关了，而可以只与当前待处理序列的长度相关。

定义 $T(C,S)$ 表示当前待二分区间长度为 C，待二分序列长度为 S，不妨设单次处理复杂度为 $O(f(n))$，则有

$$T(C,S)=T(\frac{C}{2},S_0)+T(\frac{C}{2},S-S_0)+O(f(S))$$

解之可得 $T(C,n)\le O(f(n)\log C)$

这样一来，复杂度就可以接受了。通过整体二分算法，我们仅以一个 $\log$ 的代价，便实现了在有预处理的限制下的二分查找，与正常情况下二分查找带来的复杂度相同。

三、算法流程

接下来我们考虑用整体二分来解决上述的区间 k 大问题。

假设当前整体二分的答案为 s，则我们只需统计出各个询问区间中大于 s 的数有多少个即可。如果至少有 k 个，那么答案必然大于 s；否则答案必然小于 s，而且，大于 s 的那部分元素始终对这个询问有着相同的贡献，因此，我们直接把这个询问的 k 值减掉 s，然后分治时不再统计大于 s 的元素。

而统计出各个询问区间中大于 s 的元素的个数时，我们的复杂度应该与当前处理序列区间的长度相关，而不能与序列总长 n 线性相关。正确的做法是，对当前处理区间中所有大于 s 的元素的位置排序，然后对每个询问，二分查找其端点所处的位置。这样复杂度为预处理 $O(L\log L)$，每次询问 $O(\log L)$.总复杂度 $O((n+Q)\log n\log C)$， C 是元素的值域。

错误做法的例子：

那么如何统计出各个询问区间中大于s的元素有多少个呢？很多人的第一想法可能是：开一个数组，如果序列中某个元素大于s，就在数组中对应的位置标记为1，然后求一遍部分和即可。$O(n)-O(1)$，完美的复杂度，不是么？很不幸，还真不是。

如果这么做，我们分析一下总复杂度，会惊讶的发现，总复杂度是：（定义$T(C,S)$表示解决待二分区间长度为C，待二分序列长度为S的问题的复杂度）

$$T(C,S) = T\left(\frac{C}{2},S_0\right)+T\left(\frac{C}{2},S-S_0\right)+O(n)$$

$$\text{解之得}\quad T(C,n) = O(nC)$$

四、例题

K 大数查询 (BZOJ 3110)

题目大意：

有 n 个数集（数集中可以有重复元素），要求支持：

1.往第 l 个数集到第 r 个数集中都插入一个数 k $(1\le l\le r\le n,1\le k\le n)$。

2.问你如果把第 l 个数集到第 r 个数集中的数取出来放在一起，那么这些数中第 k 大的数是多少。$1\le k\le 2^{31}-1$.

分析：

考虑应用整体二分，那么问题转变为：

有一个序列，要求支持：

1.对该序列的一段元素加 1

2.查询该序列的一段元素的和

利用线段树或树状数组统计区间和即可。

下面是一个不需要借助数据结构的做法：

由于这个修改满足“修改独立”性质，所以我们可以借助对时间分治，将问题转化为：

开始时给出若干区间，要求回答若干查询：给定一个区间，要求开始时给出的各个区间与该区间的交集的长度和。

这个问题只需要将所有区间排序后扫一遍即可。

时间复杂度 $O(m\log^2 n)$。

代码：

```
#include <algorithm>
#include <iostream>
#include <cstring>
#include <cstdio>
#include <cmath>
using namespace std;

const int N = 5e4 + 3;
int n, m, times, ans[N], sta[N], a[N], b[N], c[N], cur[N], tmp1[N], tmp2[N];

struct BIT {
    int f[N], vst[N];
    inline void Insert(int x, const int &d) {
        for (; x <= n; x += x & -x)
            if (vst[x] != times) vst[x] = times, f[x] = d;
            else    f[x] += d;
        return ;
    }
    inline int Query(int x) {
        int res = 0;
        for (; x; x -= x & -x)
            if (vst[x] == times) res += f[x];
        return res;
    }
} t, ts;

inline int Query(const int &l, const int &r) {
    return t.Query(l) * (r - l + 1) - (ts.Query(r) - ts.Query(l)) + (r + 1) * (t.Query(r) - t.Query(l));
}

inline void Insert(const int &l, const int &r, const int &d) {
    t.Insert(l, d); t.Insert(r + 1, -d);
    ts.Insert(l, l * d); ts.Insert(r + 1, -d * (r + 1));
    return ;
}

inline void Solve(const int &L, const int &R, const int &l, const int &r) {

    if (L > R) return ;
    if (l == r) {
```

```
        for (int i = L; i <= R; ++i)
            if (sta[cur[i]] == 2) ans[cur[i]] = l;
        return ;
    }
    int idx1 = 0, idx2 = 0, mid = l + r >> 1, MID, tmp; ++times;
    for (int i = L; i <= R; ++i) {
        if (sta[cur[i]] == 1) {
            if (c[cur[i]] <= mid) {
                tmp1[idx1++] = cur[i];
                Insert(a[cur[i]], b[cur[i]], 1);
            }
            else    tmp2[idx2++] = cur[i];
        }
        else {
            tmp = Query(a[cur[i]], b[cur[i]]);
            if (tmp < c[cur[i]]) {
                tmp2[idx2++] = cur[i];
                c[cur[i]] -= tmp;
            }
            else    tmp1[idx1++] = cur[i];
        }
    }
    MID = L + idx1;
    for (int i = L; i < MID; ++i) cur[i] = tmp1[i - L];
    for (int i = MID; i <= R; ++i) cur[i] = tmp2[i - MID];
    Solve(L, MID - 1, l, mid); Solve(MID, R, mid + 1, r);
    return ;
}

char ch;
inline int read() {
    while (ch = getchar(), ch < '0' || ch > '9');
    int res = ch - 48;
    while (ch = getchar(), ch >= '0' && ch <= '9') res = res * 10 + ch - 48;

    return res;
}

int main() {
    n = read(); m = read();
    for (int i = 1; i <= m; ++i) {
        sta[i] = read(); a[i] = read(); b[i] = read(); c[i] = read();
        sta[i] == 1 ? c[i] = n - c[i] + 1 : 0; cur[i] = i;
    }
```

```
    Solve(1, m, 1, n);
    for (int i = 1; i <= m; ++i)
        if (sta[i] == 2) printf("%d\n", n - ans[i] + 1);
    return 0;
}
```

练习题

BZOJ 3110 , BZOJ 2527 , BZOJ 2738

上述内容参考：

1.浅谈数据结构题的几个非经典解法，许昊然