ACM template

fangzeyu @ ZJUT

2021 年 3 月 15 日

目录

1 基础 **3**
1.1 快读 . . . 3
1.2 bitset . . . 4
1.3 int128 . . . 4
1.4 vector 去重 . . . 4

2 动态规划 **5**
2.1 背包 DP . . . 5
2.1.1 树形背包 . . . 5
2.2 概率 DP . . . 6
2.2.1 概率 DP . . . 6
2.2.2 树上概率 DP . . . 9
2.3 数位 DP . . . 11
2.3.1 模版 . . . 11
2.4 树形 DP . . . 12
2.4.1 树上背包 . . . 12
2.4.2 换根 DP . . . 13
2.5 动态 DP . . . 14
2.6 插头 DP . . . 19
2.6.1 路径模型 . . . 19
2.6.2 多条回路 . . . 21
2.7 状压 DP . . . 22
2.7.1 状压 DP . . . 22
2.8 四边形不等式优化 . . . 23
2.8.1 总结 . . . 23
2.8.2 1D1D 分治 . . . 25
2.8.3 1D1D 二分栈 . . . 26
2.8.4 区间类 2D1D . . . 27
2.9 单调栈单调队列优化 . . . 28
2.9.1 一维 . . . 28
2.9.2 二维 . . . 28
2.10 矩阵优化 . . . 30
2.10.1 矩阵乘法 . . . 30
2.10.2 广义矩阵乘法 . . . 31
2.11 GarsiaWachs 算法 . . . 32

3 字符串 **33**
3.1 KMP . . . 33
3.2 扩展 KMP . . . 34
3.3 最小表示法 . . . 35
3.4 哈希 . . . 35
3.5 马拉车 . . . 36
3.6 字典树 . . . 37
3.6.1 Trie . . . 37
3.6.2 01-Trie . . . 37
3.7 后缀数组 . . . 39
3.7.1 总结 . . . 39

3.7.2 倍增 . . . 39
3.7.3 DC3 . . . 41
3.8 单串 SAM . . . 42
3.9 多串 SAM . . . 49
3.10 后缀树 . . . 52
3.11 Lyndon 分解 . . . 54

4 数据结构 55
4.1 ST 表 . . . 55
4.2 线段树 . . . 56
4.3 二维线段树 . . . 59
4.4 主席树 . . . 62
4.5 主席树 . . . 63
4.6 线性基 . . . 63
4.7 舞蹈链 . . . 64
4.8 划分树 . . . 67
4.9 单调栈 . . . 68
4.10 并查集 . . . 69
4.10.1 带权并查集 . . . 69
4.10.2 可撤销并查集 . . . 69
4.10.3 可撤销种类并查集 . . . 71

5 计算几何 72
5.1 判两条线段相交 . . . 72

6 数论 72
6.1 埃式筛法 . . . 72
6.2 分数类 . . . 73
6.3 三分 . . . 74
6.4 组合数 . . . 75
6.5 exgcd . . . 76
6.6 高斯消元 . . . 77
6.7 计数 . . . 82
6.8 数列 . . . 82
6.8.1 oeis . . . 82
6.8.2 Bell 数 . . . 83
6.8.3 Catalan 数 . . . 83
6.8.4 超级 Catalan 数 . . . 84
6.8.5 stirling . . . 85

7 图论 86
7.1 有向图判环 . . . 86
7.2 HK 算法 . . . 86
7.3 点分治 . . . 88
7.4 树链剖分 . . . 90
7.5 矩阵树定理 . . . 93
7.6 一般图最大匹配 . . . 96
7.7 最近公共祖先 . . . 98
7.8 最小树形图 . . . 99
7.9 二分图 . . . 103

7.10 判无向图是否为二分图 103
7.11 KM 算法 103
7.12 染色法判二分图 105
7.13 匈牙利算法 106
7.14 矩阵树定理 107
7.15 启发式合并 107
7.16 2-SAT 108
7.17 网络流 109
7.17.1 网络流 109
7.17.2 最大费用流 109
7.17.3 最大费用最大流 111
7.17.4 zkw 费用流 112
7.17.5 费用流 114
7.17.6 dinic 115
7.18 最短路 117
7.18.1 Floyd 117
7.18.2 Bellman-Ford 118
7.18.3 dijkstra 119
7.18.4 bfs 全源最短路径 120
7.18.5 Johnson 全源最短路径算法 122

8 杂项 124

8.1 随机化 124
8.1.1 模拟退火 124

1 基础

1.1 快读

```
/*
* 普通快读
*/
inline int read() {
    int res = 0; bool bo = 0; char c;
    while (((c = getchar()) < '0' || c > '9') && c != '-');
    if (c == '-') bo = 1; else res = c - 48;
    while ((c = getchar()) >= '0' && c <= '9')
        res = (res << 3) + (res << 1) + (c - 48);
    return bo ? ~res + 1 : res;
}

/*
* 注意:
* 1. 结束读入输ctrl+Z!
* 2. 一旦用了这个读入优化。getchar, scanf都不能用了(存到buf里了), 所有读入都必须自己写了。所以说数据流不是太大的时候(如1*10^6), 可以考虑不用这个读入优化。
* 原理: fread这个函数的原理就是先把数据流中的一整段都存下来, 然后从这个数组里读取, 直到数组读空了再重新从数据流中读取, 由于是整段整段读取, 所以自然比getchar()要快的多。
*/
inline char nc() {
    static char buf[100000], *p1 = buf, *p2 = buf;
    if (p1 == p2) {
        p2 = (p1 = buf) + fread(buf, 1, 100000, stdin);
        if (p1 == p2)
            return EOF;
    }
    return *p1++;
}
inline void read(int &x) {
    char c = nc(), b = 1;
    for (; !(c >= '0' && c <= '9'); c = nc())
        if (c == '-')
            b = -1;
    for (x = 0; c >= '0' && c <= '9'; x = x * 10 + c - '0', c = nc())
        ;
    x *= b;
}

/*
* 泛型模版, 支持多参数读入(整形), 不支持读入实数
*/
template <typename T> inline void read (T &t){t = 0;char c = getchar();int f = 1;while (c < '0' || c > '9'){if (c == '-') f = -f;c = getchar();}while (c >= '0' && c <= '9'){t = (t << 3) + (t << 1) + c - '0';c = getchar();} t *= f;}
template <typename T,typename ... Args> inline void read (T &t,Args&... args){read (t);read (args...);}
template <typename T> inline void write (T x){if (x < 0){x = -x;putchar ('-');}if (x > 9) write (x / 10);putchar (x % 10 + '0');}
```

1.2 bitset

bitset 用法 1、二进制数的低位存储在 bitset 的低位，高位用 0 填充。

```
// 构造函数
bitset<4> bitset1;    //无参构造，长度为4，默认每一位为0

bitset<8> bitset2(12);    //长度为8，二进制保存，前面用0补充

string s = "100101";
bitset<10> bitset3(s);    //长度为10，前面用0补充

char s2[] = "10101";
bitset<13> bitset4(s2);    //长度为13，前面用0补充

cout << bitset1 << endl;    //0000
cout << bitset2 << endl;    //00001100
cout << bitset3 << endl;    //0000100101
cout << bitset4 << endl;    //0000000010101
```

1.3 int128

```
std::ostream& operator<<(std::ostream& os, __int128 t) {
    if (t==0) return os << "0";
    if (t<0) {
        os<<"-";
        t=-t;
    }
    int a[50],ai=0;
    memset(a,0,sizeof a);
    while (t!=0){
        a[ai++]=t%10;
        t/=10;
    }
    for (int i=1;i<=ai;i++) os<<abs(a[ai-i]);
    return os<<"";
}

void print(__int128 x)
{
    if (x>9) print(x/10);
    putchar('0'+x%10);
}
```

1.4 vector 去重

```
#include <bits/stdc++.h>
using namespace std;
template<typename T>
inline void deduplication(T& c) {
    sort(c.begin(), c.end());
```

```cpp
    T::iterator new_end = unique(c.begin(), c.end());//"删除"相邻的重复元素
    c.erase(new_end, c.end());//删除(真正的删除)重复的元素
}

int main() {
    int ary[] = {1, 1, 2, 3, 2, 4, 3};
    vector<int> vec(ary, ary + sizeof(ary) / sizeof(int));
    //
    deduplication(vec);
    //
    copy(vec.begin(), vec.end(), ostream_iterator<int>(cout, " "));
}
```

2 动态规划

2.1 背包 DP

2.1.1 树形背包

```cpp
/*
* 题目: Monster Hunter
* URL: https://ac.nowcoder.com/acm/contest/10272/M
* 说明: 填表TLE, 刷表AC
*/
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N = 2e3 + 5;
int pa[N], hp[N], sz[N];
vector<int> g[N];
ll dp[N][N][2], tmp[N][2];
template<typename T>
void Min(T &x, T y) {
    if(x > y) {
        x = y;
    }
}

void dfs(int u) {
    sz[u] = 1;

    dp[u][0][1] = hp[u]; dp[u][1][0] = 0;

    for (auto v : g[u]) {
        dfs(v);

        for (int i = 0; i <= sz[u] + sz[v]; i++) {
            tmp[i][0] = tmp[i][1] = 1e15;
        }

        for (int i = 0; i <= sz[u]; i++) {
            for (int j = 0; j <= sz[v]; j++) {
```

```
                Min(tmp[i+j][0], dp[u][i][0] + min(dp[v][j][0], dp[v][j][1]));
                Min(tmp[i+j][1], dp[u][i][1] + dp[v][j][0]);
                Min(tmp[i+j][1], dp[u][i][1] + dp[v][j][1] + hp[v]);
            }
        }

        sz[u] += sz[v];

        for (int i = 0; i <= sz[u]; i++) {
            dp[u][i][0] = tmp[i][0];
            dp[u][i][1] = tmp[i][1];
        }
    }
}

int main() {
    int T;
    scanf("%d", &T);
    while(T--) {

        int n;
        scanf("%d", &n);
        for (int i = 1; i <= n; i++) {
            g[i].clear();
            sz[i] = 0;
            for (int j = 0; j <= n; j++) dp[i][j][0] = dp[i][j][1] = 1e15;
        }

        for (int i = 2; i <= n; i++) {
            scanf("%d", &pa[i]);
            g[pa[i]].push_back(i);
        }

        for (int i = 1; i <= n; i++) {
            scanf("%d", &hp[i]);
        }

        dfs(1);

        for (int i = 0; i <= n; i++) {
            printf("%lld%c", min(dp[1][i][0], dp[1][i][1]), i==n?'\n':' ');
        }
    }
}
```

2.2 概率 DP

2.2.1 概率 DP

概率—期望系统的定义

概率—期望系统是一个带权的有向图。这个图中的点代表一个事件，而如果点 A 与点 B 之间有一条权为 p 的边，就表示 A 发生后，B 紧接着发生的概率是 p 。初始的时候，有一个点

（叫做初始点）代表的事件发生了，其他事件根据概率依次发生，每次只发生一个。求其他各个事件发生次数的期望。记时间 A 的发生次数期望为 E_A ，A 到 B 的边权为 P_{AB}

限制:

- 对任意的 AB ，$P_{AB} \le 1$
- 对于任意点 A ，$\sum_{(A,B)\in E} P_{AB} \le 1$，且对于系统中的所有点，至少有一个点使等号不成立。如果等号都成立的话这些事件将无穷无尽的发生下去，而概率—期望系统则变得没有意义（此时期望或者是无穷大，或者是 0 ）。
- 不能有指向初始点的边，这是因为求解时我们把初始顶点的概率设为 1 。但是如果真的有这样的边，可以添加一个假点作为初始点，这个假点到真正的初始点有一条概率为 1 的边。

概率—期望系统的求解

可以根据是否为 DAG 图，将问题分为两类。

有向无环图的概率—期望系统。这种系统是很简单的，因为它没有后效性，所以可以通过动态规划的方法在 O(E) 的时间内解决。许多使用动态规划解决的概率—期望问题都是基于这类系统的。

在有些问题中，我们需要解决更一般的概率—期望系统。这时图中含有圈，因而造成了后效性。

高斯消元解决后效性概率 DP

$$E_A = \sum_{(B,A)\in E} P_{BA} E_B$$

求线性方程组的解，我们更常用的稳定算法是高斯消元法，完全可以在这里使用。这样就得到了一种稳定而精确的解法：首先根据概率—期望系统建立方程组，然后用高斯消元法去解，得到的结果就是我们要求的期望。这种算法的时间复杂度是 $O(n^3)$ 。

一种可以消除后效性的特例

限制:

- 图为有向无环图 (线性递推、树)，非 DAG 图需要用 Tarjan 算法缩点
- 存在无后效性可以转移的节点，即只从前置状态转移而来
- 每个状态只从常数个数的状态转移而来

解法:

根据 DAG 图的逆拓扑序，将所有节点的转移方程依次变为无后效性的。原理等价于高斯消元，

```

/*
* URL: https://zoj.pintia.cn/problem-sets/91827364500/problems/91827368253
```

```
* 题意:
* 有三个骰子，分别有k1,k2,k3个面。
* 每次掷骰子，如果三个面分别为a,b,c则分数置0，否则加上三个骰子的分数之和。
* 当分数大于n时结束。求游戏的期望步数。初始分数为0
* 分析:
* 假设dp[i]表示拥有分数i到游戏结束的期望步数
* 则
* (1):dp[i]=SUM(p[k]*dp[i+k])+p[0]*dp[0]+1;//p[k]表示增加分数为k的概率,p[0]表示分数变为0
  的概率
* 假定
* (2):dp[i]=A[i]*dp[0]+B[i];
* 则
* (3):dp[i+k]=A[i+k]*dp[0]+B[i+k];
* 将(3)代入(1)得:
* (4):dp[i]=(SUM(p[k]*A[i+k])+p[0])*dp[0]+SUM(p[k]*B[i+k])+1;
* 将4与2做比较得:
* A[i]=(SUM(p[k]*A[i+k])+p[0]);
* B[i]=SUM(p[k]*B[i+k])+1;
* 当i+k>n时A[i+k]=B[i+k]=0可知
* 所以dp[0]=B[0]/(1-A[0])可求出
*************************************************************************
总结下这类概率DP:
既DP[i]可能由DP[i+k]和DP[i+j]需要求的比如DP[0]决定
相当于概率一直递推下去会回到原点
比如
(1):DP[i]=a*DP[i+k]+b*DP[0]+d*DP[i+j]+c;
但是DP[i+k]和DP[0]都是未知
这时候根据DP[i]的方程式假设一个方程式:
比如:
(2):DP[i]=A[i]*DP[i+k]+B[i]*DP[0]+C[i];
因为要求DP[0],所以当i=0的时候但是A[0],B[0],C[0]未知
对比(1)和(2)的差别
这时候对比(1)和(2)发现两者之间的差别在于DP[i+j]
所以根据(2)求DP[i+j]然后代入(1)消除然后对比(2)就可以得到A[i],B[i],C[i]
然后视具体情况根据A[i],B[i],C[i]求得A[0],B[0],C[0]继而求DP[0]
请看这题: http://acm.hdu.edu.cn/showproblem.php?pid=4035
*************************************************************************
*/
#include <bits/stdc++.h>
using namespace std;
const int N = 505;
const double eps = 1e-9;
double p[40], A[N], B[N];
int main() {
    int T;
    scanf("%d", &T);
    while (T--) {
        memset(p, 0, sizeof(p));
        memset(A, 0, sizeof(A));
        memset(B, 0, sizeof(B));
        int n, k1, k2, k3, a, b, c;
        scanf("%d%d%d%d%d%d%d", &n, &k1, &k2, &k3, &a, &b, &c);
        p[0] = 1.0 / (k1 * k2 * k3);
```

```
        for (int i = 1; i <= k1; i++) {
            for (int j = 1; j <= k2; j++) {
                for (int k = 1; k <= k3; k++) {
                    int sum = i + j + k;
                    p[sum] += p[0];
                }
            }
        }
        p[a+b+c] -= p[0];


        for (int i = n; i >= 0; i--) {
            A[i] = p[0];
            B[i] = 1;
            for (int k = 3; k <= k1 + k2 + k3; k++) {
                if(i+k > n) {
                    break; // 注意精度
                }
                A[i] += p[k] * A[i+k];
                B[i] += p[k] * B[i+k];
            }
        }
        printf("%.15f\n", B[0]/(1-A[0]));
    }
}
```

2.2.2 树上概率 DP

```
/*
HDU 4035

    dp求期望的题。
    题意:
    有n个房间，由n-1条隧道连通起来，实际上就形成了一棵树，
    从结点1出发，开始走，在每个结点i都有3种可能:
        1.被杀死，回到结点1处（概率为ki）
        2.找到出口，走出迷宫 （概率为ei）
        3.和该点相连有m条边，随机走一条
    求：走出迷宫所要走的边数的期望值。

    设 E[i]表示在结点i处，要走出迷宫所要走的边数的期望。E[1]即为所求。

    叶子结点:
    E[i] = ki*E[1] + ei*0 + (1-ki-ei)*(E[father[i]] + 1);
         = ki*E[1] + (1-ki-ei)*E[father[i]] + (1-ki-ei);

    非叶子结点：（m为与结点相连的边数）
    E[i] = ki*E[1] + ei*0 + (1-ki-ei)/m*( E[father[i]]+1 + ∑( E[child[i]]+1 ) );
         = ki*E[1] + (1-ki-ei)/m*E[father[i]] + (1-ki-ei)/m*∑(E[child[i]]) + (1-ki-ei);

    设对每个结点：E[i] = Ai*E[1] + Bi*E[father[i]] + Ci;

```

```
    对于非叶子结点i, 设j为i的孩子结点, 则
     (E[child[i]]) = E[j]
                   = (Aj*E[1] + Bj*E[father[j]] + Cj)
                   = (Aj*E[1] + Bj*E[i] + Cj)
    带入上面的式子得
    (1 - (1-ki-ei)/m* Bj)*E[i] = (ki+(1-ki-ei)/m* Aj)*E[1] + (1-ki-ei)/m*E[father[i]] +
        (1-ki-ei) + (1-ki-ei)/m* Cj;
    由此可得
    Ai = (ki+(1-ki-ei)/m* Aj) / (1 - (1-ki-ei)/m* Bj);
    Bi = (1-ki-ei)/m / (1 - (1-ki-ei)/m* Bj);
    Ci = ( (1-ki-ei)+(1-ki-ei)/m* Cj ) / (1 - (1-ki-ei)/m* Bj);

    对于叶子结点
    Ai = ki;
    Bi = 1 - ki - ei;
    Ci = 1 - ki - ei;

    从叶子结点开始, 直到算出 A1,B1,C1;

    E[1] = A1*E[1] + B1*0 + C1;
    所以
    E[1] = C1 / (1 - A1);
    若 A1趋近于1则无解...

*/
#include <bits/stdc++.h>
using namespace std;
const int N = 1e4 + 5;
const double eps = 1e-9;
vector<int> g[N];
double k[N], e[N];
double A[N], B[N], C[N];
bool dfs(int u, int fa) {
    int m = g[u].size();
    A[u] = k[u];
    B[u] = (1 - k[u] - e[u]) / m;
    C[u] = 1 - k[u] - e[u];
    double tmp = 1;
    for (auto v : g[u]) {
        if(v == fa) continue;
        if(!dfs(v, u)) return false;
        A[u] += (1-k[u]-e[u])/m*A[v];
        C[u] += (1-k[u]-e[u])/m*C[v];
        tmp -= (1-k[u]-e[u])/m*B[v];
    }
    if(fabs(tmp)<eps) return false;
    A[u] /= tmp;
    B[u] /= tmp;
    C[u] /= tmp;
    return true;
}
int main() {
    int T, kase = 0;
```

```
    scanf("%d", &T);
    while (T--) {
        int n;
        scanf("%d", &n);
        for (int i = 1; i <= n; i++) g[i].clear();
        for (int i = 1; i < n; i++) {
            int u, v;
            scanf("%d%d", &u, &v);
            g[u].push_back(v);
            g[v].push_back(u);
        }

        for (int i = 1; i <= n; i++) {
            int ki, ei;
            scanf("%d%d", &ki, &ei);
            k[i] = ki / 100.0;
            e[i] = ei / 100.0;
        }
        printf("Case %d: ", ++kase);
        if(dfs(1, 0) && fabs(1-A[1])>eps) {
            printf("%.6lf\n", C[1]/(1-A[1]));
        }
        else {
            puts("impossible");
        }
    }
}
```

2.3 数位 DP

2.3.1 模版

```
typedef long long ll;
int a[20];
ll dp[20][state];//不同题目状态不同
ll dfs(int pos,/*state变量*/,bool lead/*前导零*/,bool limit/*数位上界变量*/)//不是每个题都要判断前导零
{
    //递归边界，既然是按位枚举，最低位是0，那么pos==-1说明这个数我枚举完了
    if(pos==-1) return 1;/*这里一般返回1，表示你枚举的这个数是合法的，那么这里就需要你在枚举时必须每一位都要满足题目条件，也就是说当前枚举到pos位，一定要保证前面已经枚举的数位是合法的。不过具体题目不同或者写法不同的话不一定要返回1 */
    //第二个就是记忆化(在此前可能不同题目还能有一些剪枝)
    if(!limit && !lead && dp[pos][state]!=-1) return dp[pos][state];
    /*常规写法都是在没有限制的条件记忆化，这里与下面记录状态是对应，具体为什么是有条件的记忆化后面会讲*/
    int up=limit?a[pos]:9;//根据limit判断枚举的上界up;这个的例子前面用213讲过了
    ll ans=0;
    //开始计数
    for(int i=0;i<=up;i++)//枚举，然后把不同情况的个数加到ans就可以了
    {
        if() ...
```

```
        else if()...
        ans+=dfs(pos-1,/*状态转移*/,lead && i==0,limit && i==a[pos]) //最后两个变量传参都是这样写的
        /*这里还算比较灵活，不过做几个题就觉得这里也是套路了
        大概就是说，我当前数位枚举的数是i，然后根据题目的约束条件分类讨论
        去计算不同情况下的个数，还有要根据state变量来保证i的合法性，比如题目
        要求数位上不能有62连续出现,那么就是state就是要保存前一位pre,然后分类，
        前一位如果是6那么这意味就不能是2，这里一定要保存枚举的这个数是合法*/
    }
    //计算完，记录状态
    if(!limit && !lead) dp[pos][state]=ans;
    /*这里对应上面的记忆化，在一定条件下时记录，保证一致性，当然如果约束条件不需要考虑lead，这里就是lead就完全不用考虑了*/
    return ans;
}
ll solve(ll x)
{
    int pos=0;
    while(x)//把数位都分解出来
    {
        a[pos++]=x%10;//个人老是喜欢编号为[0,pos),看不惯的就按自己习惯来，反正注意数位边界就行
        x/=10;
    }
    return dfs(pos-1/*从最高位开始枚举*/,/*一系列状态 */,true,true);//刚开始最高位都是有限制并且有前导零的，显然比最高位还要高的一位视为0嘛
}
int main()
{
    ll le,ri;
    while(~scanf("%lld%lld",&le,&ri))
    {
        //初始化dp数组为-1,这里还有更加优美的优化,后面讲
        printf("%lld\n",solve(ri)-solve(le-1));
    }
}
```

2.4 树形 DP

2.4.1 树上背包

```
/*
* 题目：[CTSC1997]选课
* URL: https://www.luogu.com.cn/problem/P2014
* 题意：k[i]表示第i门课的直接先修课，s[i]表示第i门课的学分。
* 时间复杂度：O(nk)（树上有n个点，第二维限制为k（最多选k个))
*/
#include <bits/stdc++.h>
using namespace std;
int f[305][305], s[305], n, m;
vector<int> e[305];
int dfs(int u) {
```

```
  int p = 1;
  f[u][1] = s[u];
  for (auto v : e[u]) {
    int siz = dfs(v);
    // 注意下面两重循环的上界和下界
    // 只考虑已经合并过的子树，以及选的课程数超过 m+1 的状态没有意义
    for (int i = min(p, m + 1); i; i--)
      for (int j = 1; j <= siz && i + j <= m + 1; j++)
        f[u][i + j] = max(f[u][i + j], f[u][i] + f[v][j]);
    p += siz;
  }
  return p;
}
int main() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= n; i++) {
    int k;
    scanf("%d%d", &k, &s[i]);
    e[k].push_back(i);
  }
  dfs(0);
  printf("%d", f[0][m + 1]);
}
```

2.4.2 换根 DP

```
/*
* 题目: [POI2008]STA-Station
* URL: https://www.luogu.com.cn/problem/P3478
* 树形 DP 中的换根 DP 问题又被称为二次扫描，通常不会指定根结点，并且根结点的变化会对一些值，
  例如子结点深度和、点权和等产生影响。
* 通常需要两次 DFS，第一次 DFS 预处理诸如深度，点权和之类的信息，在第二次 DFS 开始运行换根
  动态规划。
* 换根解决的是“不定根”的树形dp问题。该类题目的特点是: 给定一个树形结构，需要以每个节点为根
  进行一系列统计。
* 方法为两次扫描来求解:
* 第一次扫描时，任选一个点为根，在“有根树”上执行一次树形dp，在回溯时，自底向上的状态转移。
* 第二次扫描时，从第一次选的根出发，对整棵树执行一个dfs，在每次递归前进行自顶向下的转移，计
  算出换根后的解。
*/
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N = 1e6 + 5;
vector<int> e[N];
ll sz[N], sum[N], f[N];
int n;
void dfs1(int u, int fa) {
    sz[u] = 1; sum[u] = 0;
    for (auto v : e[u]) {
        if(v == fa) continue;
        dfs1(v, u);
```

```
        sz[u] += sz[v];
        sum[u] += sum[v] + sz[v];
    }
}
void dfs2(int u, int fa) {
    for (auto v : e[u]) {
        if(v == fa) continue;
        f[v] = f[u] + n - 2 * sz[v];
        dfs2(v, u);
    }
}
int main() {
    scanf("%d", &n);
    for (int i = 1; i < n; i++) {
        int u, v;
        scanf("%d%d", &u, &v);
        e[u].push_back(v);
        e[v].push_back(u);
    }
    dfs1(1, 0);
    f[1] = sum[1];
    dfs2(1, 0);
    int ans = 1;
    for (int i = 1; i <= n; i++) if(f[i] > f[ans]) ans = i;
    printf("%d\n", ans);
}
```

2.5 动态 DP

广义矩阵乘法

定义广义矩阵乘法 $A \times B = C$ 为:

$$C_{i,j} = \max_{k=1}^{n}(A_{i,k} + B_{k,j})$$

相当于将普通的矩阵乘法中的乘变为加，加变为 max 操作。

同时广义矩阵乘法满足结合律，所以可以使用矩阵快速幂。

不带修改操作

令 $f_{i,0}$ 表示不选择 i 的最大答案，$f_{i,1}$ 表示选择 i 的最大答案。

则有 DP 方程:

$$\begin{cases} f_{i,0} = \sum_{son} \max(f_{son,0}, f_{son,1}) \\ f_{i,1} = w_i + \sum_{son} f_{son,0} \end{cases}$$

答案就是 $\max(f_{root,0}, f_{root,1})$.

带修改操作

设 $g_{i,0}$ 表示不选择 i 且只允许选择 i 的轻儿子所在子树的最大答案，$g_{i,1}$ 表示选择 i 的最大答案，son_i 表示 i 的重儿子。

假设我们已知 $g_{i,0/1}$ 那么有 DP 方程：

$$\begin{cases} f_{i,0} = g_{i,0} + \max(f_{son_i,0}, f_{son_i,1}) \\ f_{i,1} = g_{i,1} + f_{son_i,0} \end{cases}$$

答案是 $\max(f_{root,0}, f_{root,1})$.

可以构造出矩阵：

$$\begin{bmatrix} g_{i,0} & g_{i,0} \\ g_{i,1} & -\infty \end{bmatrix} \times \begin{bmatrix} f_{son_i,0} \\ f_{son_i,1} \end{bmatrix} = \begin{bmatrix} f_{i,0} \\ f_{i,1} \end{bmatrix}$$

注意，我们这里使用的是广义乘法规则。

可以发现，修改操作时只需要修改 $g_{i,1}$ 和每条往上的重链即可。

具体思路

- DFS 预处理求出 $f_{i,0/1}$ 和 $g_{i,0/1}$.
- 对这棵树进行树剖（注意，因为我们对一个点进行询问需要计算从该点到该点所在的重链末尾的区间矩阵乘，所以对于每一个点记录 End_i 表示 i 所在的重链末尾节点编号），每一条重链建立线段树，线段树维护 g 矩阵和 g 矩阵区间乘积。
- 修改时首先修改 $g_{i,1}$ 和线段树中 i 节点的矩阵，计算 top_i 矩阵的变化量，修改到 fa_{top_i} 矩阵。
- 查询时就是 1 到其所在的重链末尾的区间乘，最后取一个 max 即可。

```
/*
* 模版: 动态DP
* 一般用来解决树上的 DP 问题，同时支持点权（边权）修改操作。
*/
#include <bits/stdc++.h>
using namespace std;

#define REP(i, a, b) for (int i = (a), _end_ = (b); i <= _end_; ++i)
#define mem(a) memset((a), 0, sizeof(a))
#define str(a) strlen(a)
#define lson root << 1
#define rson root << 1 | 1
typedef long long LL;

const int maxn = 500010;
const int INF = 0x3f3f3f3f;

int Begin[maxn], Next[maxn], To[maxn], e, n, m;
int size[maxn], son[maxn], top[maxn], fa[maxn], dis[maxn], p[maxn], id[maxn],
```

```
    End[maxn];
// p[i]表示i树剖后的编号，id[p[i]] = i
int cnt, tot, a[maxn], f[maxn][2];

struct matrix
{
    int g[2][2];
    matrix() { memset(g, 0, sizeof(g)); }
    matrix operator*(const matrix &b) const // 重载矩阵乘
    {
        matrix c;
        REP(i, 0, 1)
        REP(j, 0, 1)
        REP(k, 0, 1)
        c.g[i][j] = max(c.g[i][j], g[i][k] + b.g[k][j]);
        return c;
    }
} Tree[maxn], g[maxn]; // Tree[]是建出来的线段树，g[]是维护的每个点的矩阵

inline void PushUp(int root) { Tree[root] = Tree[lson] * Tree[rson]; }

inline void Build(int root, int l, int r)
{
    if (l == r)
    {
        Tree[root] = g[id[l]];
        return;
    }
    int Mid = l + r >> 1;
    Build(lson, l, Mid);
    Build(rson, Mid + 1, r);
    PushUp(root);
}

inline matrix Query(int root, int l, int r, int L, int R)
{
    if (L <= l && r <= R) return Tree[root];
    int Mid = l + r >> 1;
    if (R <= Mid) return Query(lson, l, Mid, L, R);
    if (Mid < L) return Query(rson, Mid + 1, r, L, R);
    return Query(lson, l, Mid, L, R) * Query(rson, Mid + 1, r, L, R);
    // 注意查询操作的书写
}

inline void Modify(int root, int l, int r, int pos)
{
    if (l == r)
    {
        Tree[root] = g[id[l]];
        return;
    }
    int Mid = l + r >> 1;
    if (pos <= Mid) Modify(lson, l, Mid, pos);
```

```
    else Modify(rson, Mid + 1, r, pos);
    PushUp(root);
}

inline void Update(int x, int val)
{
    g[x].g[1][0] += val - a[x];
    a[x] = val;
    // 首先修改x的g矩阵
    while (x)
    {
        // 查询top[x]的原本g矩阵
        matrix last = Query(1, 1, n, p[top[x]], End[top[x]]);

        // 进行修改(x点的g矩阵已经进行修改但线段树上的未进行修改)
        Modify(1, 1, n, p[x]);

        // 查询top[x]的新g矩阵
        matrix now = Query(1, 1, n, p[top[x]], End[top[x]]);

        // 根据变化量修改fa[top[x]]的g矩阵
        x = fa[top[x]];
        g[x].g[0][0] += max(now.g[0][0], now.g[1][0]) - max(last.g[0][0], last.g[1][0]);
        g[x].g[0][1] = g[x].g[0][0];
        g[x].g[1][0] += now.g[0][0] - last.g[0][0];
    }
}

inline void add(int u, int v)
{
    To[++e] = v;
    Next[e] = Begin[u];
    Begin[u] = e;
}

inline void DFS1(int u)
{
    size[u] = 1;
    int Max = 0;
    f[u][1] = a[u];
    for (int i = Begin[u]; i; i = Next[i])
    {
        int v = To[i];
        if (v == fa[u]) continue;
        dis[v] = dis[u] + 1;
        fa[v] = u;
        DFS1(v);
        size[u] += size[v];
        if (size[v] > Max)
        { // 求重儿子
            Max = size[v];
            son[u] = v;
        }
```

```
        // DFS1过程中同时求出f[i][0/1]
        f[u][1] += f[v][0];
        f[u][0] += max(f[v][0], f[v][1]);
    }
}

// 树链剖分
inline void DFS2(int u, int t)
{
    top[u] = t;

    // 统计dfs序
    p[u] = ++cnt;
    id[cnt] = u;
    End[t] = cnt;

    // DFS2过程中预处理g[i][0/1]
    g[u].g[1][0] = a[u];
    g[u].g[1][1] = -INF;

    if (!son[u]) return;
    DFS2(son[u], t);

    for (int i = Begin[u]; i; i = Next[i])
    {
        int v = To[i];
        if (v == fa[u] || v == son[u]) continue;
        DFS2(v, v);
        // g矩阵根据f[i][0/1]求出
        g[u].g[0][0] += max(f[v][0], f[v][1]);
        g[u].g[1][0] += f[v][0];
    }
    g[u].g[0][1] = g[u].g[0][0];
}

int main()
{
    scanf("%d%d", &n, &m);
    REP(i, 1, n)
    scanf("%d", &a[i]);
    REP(i, 1, n - 1)
    {
        int u, v;
        scanf("%d%d", &u, &v);
        add(u, v);
        add(v, u);
    }
    dis[1] = 1;
    DFS1(1);
    DFS2(1, 1);
    Build(1, 1, n);
    REP(i, 1, m)
    {
```

```
    int x, val;
    scanf("%d%d", &x, &val);
    Update(x, val);
    matrix ans = Query(1, 1, n, 1, End[1]); // 查询1所在重链的矩阵乘
    printf("%d\n", max(ans.g[0][0], ans.g[1][0]));
  }
  return 0;
}
```

2.6 插头 DP

2.6.1 路径模型

```
/*
* 状态编码：括号表示和最小表示
*/

/* 最小表示
* 长度m+1的整形数组，记录轮廓线上每个插头的状态
* 0表示没有插头，并约定连通的插头用相同的数字进行标记
* b[]：轮廓线上插头的状态。
* bb[]：在最小表示的编码的过程中，每个数字被映射到的最小数字。
* 注意：0表示插头不存在，不能被映射成其他值。
*/
#include <bits/stdc++.h>
using namespace std;
#define REP(i, n) for (int i = 0; i < n; ++i)
const int M = 10;
const int offset = 3, mask = (1 << offset) - 1;
int n, m;
long long ans, d;
const int MaxSZ = 16796, Prime = 9973;
// MaxSz：表示合法状态的上界，可以估计，也可以预处理出较为精确的值。
// Prime：一个小于 MaxSZ 的大素数。
struct hashTable {
  int head[Prime], next[MaxSZ], sz;
  int state[MaxSZ];
  long long key[MaxSZ];
  // 初始化函数，和手写邻接表类似，我们只需要初始化表头节点的指针。
  inline void clear() {
    sz = 0;
    memset(head, -1, sizeof(head));
  }
  // 状态转移函数，其中d是一个全局变量，表示每次状态转移所带来的增量。如果找到的话就+=，否则
      就创建一个状态为s，关键字为d的新节点。
  inline void push(int s) {
    int x = s % Prime;
    for (int i = head[x]; ~i; i = next[i]) {
      if (state[i] == s) {
        key[i] += d;
        return;
      }
```

```
    }
    state[sz] = s, key[sz] = d;
    next[sz] = head[x];
    head[x] = sz++;
  }
  void roll() { REP(i, sz) state[i] <<= offset; }
} H[2], *H0, *H1;
int b[M + 1], bb[M + 1];
int encode() {
  int s = 0;
  memset(bb, -1, sizeof(bb));
  int bn = 1;
  bb[0] = 0;
  for (int i = m; i >= 0; --i) {
#define bi bb[b[i]]
    if (!~bi) bi = bn++;
    s <<= offset;
    s |= bi;
  }
  return s;
}
void decode(int s) {
  REP(i, m + 1) {
    b[i] = s & mask;
    s >>= offset;
  }
}
void push(int j, int dn, int rt) {
  b[j] = dn;
  b[j + 1] = rt;
  H1->push(encode());
}
int main() {
  cin >> n >> m;
  if (m > n) swap(n, m);
  H0 = H, H1 = H + 1;
  H1->clear();
  d = 1;
  H1->push(0);
  REP(i, n) {
    REP(j, m) {
      swap(H0, H1);
      H1->clear();
      REP(ii, H0->sz) {
        decode(H0->state[ii]);  // 取出状态，并解码
        d = H0->key[ii];  // 得到增量 delta
        int lt = b[j], up = b[j + 1]; // 左插头，上插头
        bool dn = i != n - 1, rt = j != m - 1; // 下插头，右插头
        if (lt && up) { // 如果左、上均有插头
          if (lt == up) { // 来自同一个连通块
            if (i == n - 1 && j == m - 1) { // 只有在最后一个格子时，才能合并，封闭回路。
              push(j, 0, 0);
            }
```

```
          } else { // 否则，必须合并这两个连通块，因为本题中需要回路覆盖
            REP(i, m + 1) if (b[i] == lt) b[i] = up;
            push(j, 0, 0);
          }
        } else if (lt || up) { // 如果左、上之中有一个插头
          int t = lt | up; // 得到这个插头
          if (dn) { // 如果可以向下延伸
            push(j, t, 0);
          }
          if (rt) { // 如果可以向右延伸
            push(j, 0, t);
          }
        } else { // 如果左、上均没有插头
          if (dn && rt) { // 生成一对新插头
            push(j, m, m);
          }
        }
      }
    }
    H1->roll();
  }
  assert(H1->sz <= 1);
  cout << (H1->sz == 1 ? H1->key[0] : 0) << endl;
}
```

2.6.2 多条回路

```
/*
* 多条回路问题并不属于插头 DP，因为我们只需要和上面的骨牌覆盖问题一样，记录插头是否存在，然后成对的合并和生成插头就可以了。
*/
#include <bits/stdc++.h>
using namespace std;
const int N = 11;
long long f[2][1 << (N + 1)], *f0, *f1;
int n, m;
int main() {
  int T;
  cin >> T;
  for (int Case = 1; Case <= T; ++Case) {
    cin >> n >> m;
    f0 = f[0];
    f1 = f[1];
    fill(f1, f1 + (1 << m + 1), 0);
    f1[0] = 1;
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < m; ++j) {
        bool bad;
        cin >> bad;
        bad ^= 1;
        swap(f0, f1);
        fill(f1, f1 + (1 << m + 1), 0);
```

```
#define u f0[s]
        for (int s = 0; s < 1 << m + 1; ++s)
          if (u) {
            bool lt = s >> j & 1, up = s >> j + 1 & 1;
            if (bad) {
              if (!lt && !up) f1[s] += u;
            } else {
              f1[s ^ 3 << j] += u;
              if (lt != up) f1[s] += u;
            }
          }
      }
      swap(f0, f1);
      fill(f1, f1 + (1 << m + 1), 0);
      for (int s = 0; s < 1 << m; ++s) f1[s << 1] = u;
    }
    printf("Case %d: There are %lld ways to eat the trees.\n", Case, f1[0]);
  }
}
```

2.7 状压 DP

2.7.1 状压 DP

技巧: 1. 用位运算优化判断，如判合法，或者有无相邻的 1 可以用式子 $s1\&(s1 << 1)$

空间优化: 1. 使用滚动数组，而非 swap，只要记录当前状态和上一个状态，节省一维空间

时间优化: 1. 先将所有合法状态过滤出来

算法: 1. 初始化 2. 状态转移 3. 统计答案

```
#include <bits/stdc++.h>
using namespace std;
#define rep(a,b,c) for (int a=b;a<=c;a++)
#define per(a,b,c) for (int a=b;a>=c;a--)
#define fi first
#define se second // pair
#define hav(s,p) (s>>(p-1)&1)
#define ins(s,p) (s|1<<(p-1))
#define del(s,p) (s^1<<(p-1)) // state compression
const int N = 12;
const int P = 1e8;
int s[N], sta[1<<N], cnt;
int f[2][1<<N];
int main() {
    int n, m;
    scanf("%d%d", &n, &m);
    rep(i, 1, n) rep(j, 1, m) {
        int t; scanf("%d", &t);
        if(!t) s[i] = ins(s[i], j);
    }
    int MX = (1<<m) -1;
    rep(st, 0, MX) {
        if(st & (st<<1)) continue;
```

```
        sta[++cnt] = st;
    }
    f[0&1][0] = 1;
    rep(i, 1, n) {
        memset(f[i&1], 0, sizeof(f[i&1]));
        rep(j, 1, cnt) {
            if(sta[j] & s[i-1]) continue;
            rep(k, 1, cnt) {
                if(sta[k] & s[i]) continue;
                if(sta[j] & sta[k]) continue;
                f[i&1][sta[k]] = (f[i&1][sta[k]] + f[(i-1)&1][sta[j]]) % P;
            }
        }
    }
    int ans = 0;
    rep(st, 0, MX) ans = (ans + f[n&1][st]) % P;
    printf("%d\n", ans);
}
```

2.8 四边形不等式优化

2.8.1 总结

一、DP 时间复杂度

$$\text{时间复杂度} = \text{状态总数} \times \text{每个状态转移的状态数} \times \text{每次状态转移的时间}$$

二、各类优化方式

1. 决策单调性

四边形不等式的性质在一类 1D1D 动态规划中得出决策单调性，从而优化状态转移的复杂度。

1D1D 动态规划： DP 方程形如 $f_r = \min_{l=1}^{r-1}\{f_l + w(l,r)\} \quad (1 \le r \le n)$, 状态数为 $O(n)$, 每一步决策量为 $O(n)$ 。

决策单调性： 设 k_i 表示 $f[i]$ 转移的最优决策点，那么决策单调性可描述为 $\forall i \le j, k_i \le k_j$。也就是说随着 i 的增大，所找到的最优决策点是递增态（非严格递增）。

定理：若函数 $w(l,r)$ 满足四边形不等式，记 $h_{l,r} = f_l + w(l,r)$ 表示从 l 转移过来的状态 r , $k_r = \min\{l | f_r = h_{l,r}\}$ 表示最优决策点，则有

$$\forall r_1 \le r_2 : k_{r_1} \le k_{r_2}$$

四边形不等式： 如果对于任意 $l_1 \le l_2 \le r_1 \le r_2$ ，均有 $w(l_1,r_1) + w(l_2,r_2) \le w(l_1,r_2) + w(l_2,r_1)$ 成立，则称函数 w 满足四边形不等式（简记为“交叉小于包含”）。若等号永远成立，则称函数 w 满足四边形恒等式。

我们根据决策单调性只能得出每次枚举 l 时的下界，而无法确定其上界。因此，简单实现该状态转移方程仍然无法优化最坏时间复杂度。

2. 决策单调性（分治）

先考虑一种简单的情况，转移函数的值在动态规划前就已完全确定。即如下所示状态转移方程：

$$f_r = \min_{l=1}^{r-1} w(l,r) \qquad (1 \le r \le n)$$

在这种情况下，我们定义过程 $\mathsf{DP}(l,r,k_l,k_r)$ 表示求解 $f_l \sim f_r$ 的状态值，并且已知这些状态的最优决策点必定位于 $[k_l,k_r]$ 中，然后使用分治算法如 **单调性决策（分治）** 中所诉。

使用递归树的方法，容易分析出该分治算法的复杂度为 $O(n\log n)$ ，因为递归树每一层的决策区间总长度不超过 $2n$ ，而递归层数显然为 $O(\log n)$ 级别。

3. 决策单调性（二分栈）

处理一般情况，即转移函数的值是在动态规划的过程中按照一定的拓扑序逐步确定的。此时我们需要改变思维方式，由“确定一个状态的最优决策”转化为“确定一个决策是哪些状态的最优决策”。

用栈维护单调的决策点，二分找到是哪些状态的最优决策，时间复杂度为 $O(n\log n)$。

4. 区间类（2D1D）动态规划

在区间类动态规划（如石子合并问题）中，我们经常遇到以下形式的 2D1D 状态转移方程：

$$f_{l,r} = \min_{k=l}^{r-1}\{f_{l,k} + f_{k+1,r}\} + w(l,r) \qquad (1 \le l \le r \le n)$$

直接简单实现状态转移，总时间复杂度将会达到 $O(n^3)$ ，但当函数 $w(l,r)$ 满足一些特殊的性质时，我们可以利用决策的单调性进行优化。

区间包含单调性： 如果对于任意 $l \le l' \le r' \le r$ ，均有 $w(l',r') \le w(l,r)$ 成立，则称函数 w 对于区间包含关系具有单调性。

四边形不等式： 如果对于任意 $l_1 \le l_2 \le r_1 \le r_2$ ，均有 $w(l_1,r_1) + w(l_2,r_2) \le w(l_1,r_2) + w(l_2,r_1)$ 成立，则称函数 w 满足四边形不等式（简记为“交叉小于包含”）。若等号永远成立，则称函数 w 满足四边形恒等式。

引理 1 ：若满足关于区间包含的单调性的函数 $w(l,r)$ 满足四边形不等式，则状态 $f_{l,r}$ 也满足四边形不等式。

定理 1 ：若状态 f 满足四边形不等式，记 $m_{l,r} = \min\{k : f_{l,r} = g_{k,l,r}\}$ 表示最优决策点，则有

$$m_{l,r-1} \le m_{l,r} \le m_{l+1,r}$$

因此，如果在计算状态 $f_{l,r}$ 的同时将其最优决策点 $m_{l,r}$ 记录下来，那么我们对决策点 k 的总枚举量将降为

$$\sum_{1\le l<r\le n} m_{l+1,r} - m_{l,r-1} = \sum_{i=1}^{n} m_{i,n} - m_{1,i} \le n^2$$

5. 满足四边形不等式的函数类

为了更方便地证明一个函数满足四边形不等式，我们有以下几条性质：

性质 1 ： 若函数 $w_1(l,r), w_2(l,r)$ 均满足四边形不等式（或区间包含单调性），则对于任意 $c_1, c_2 \geq 0$ ， 函数 $c_1w_1 + c_2w_2$ 也满足四边形不等式（或区间包含单调性）。

性质 2 ： 若存在函数 $f(x), g(x)$ 使得 $w(l,r) = f(r) - g(l)$ ， 则函数 w 满足四边形恒等式。当函数 f, g 单调增加时，函数 w 还满足区间包含单调性。

性质 3 ： 设 $h(x)$ 是一个单调增加的凸函数，若函数 $w(l,r)$ 满足四边形不等式并且对区间包含关系具有单调性，则复合函数 $h(w(l,r))$ 也满足四边形不等式和区间包含单调性。

性质 4 ： 设 $h(x)$ 是一个凸函数，若函数 $w(l,r)$ 满足四边形恒等式并且对区间包含关系具有单调性，则复合函数 $h(w(l,r))$ 也满足四边形不等式。

2.8.2 1D1D 分治

```cpp
/*
* 题目: [POI2011]Lightning Conductor
* URL: https://www.luogu.com.cn/problem/P3515
* 说明: 此题中没有限制决策点j的范围, 故需正反都来一遍
*/
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N = 5e5 + 5;
int a[N];
double sq[N], f[N], g[N];
double w(int l, int r) {
    return a[l] + sq[r-l] - a[r];
}
void solve(int l, int r, int kl, int kr, double dp[]) {
    int mid = (l+r) >> 1, k = kl;
    for (int i = kl; i <= min(kr, mid); i++) {
        if(w(i, mid) > w(k, mid)) k = i;
    }
    dp[mid] = w(k, mid);
    if(l < mid) solve(l, mid-1, kl, k, dp);
    if(r > mid) solve(mid+1, r, k, kr, dp);
}
int main() {
    int n;
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) scanf("%d", &a[i]);
    for (int i = 0; i <= n; i++) sq[i] = sqrt(i);
    solve(1, n, 1, n, f);
    reverse(a+1, a+n+1);
    solve(1, n, 1, n, g);
    for (int i = 1; i <= n; i++) {
        printf("%lld\n", (ll)ceil(max(f[i], g[n-i+1])));
    }
}
```

2.8.3 1D1D 二分栈

```
/*
* 题目: [HNOI2008]玩具装箱
* URL: https://www.luogu.com.cn/problem/P3195
*/
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N = 5e4 + 5;
int c[N];
ll sum[N], f[N];
int n, L;
struct node {
    int l, r, x;
} stk[N];
int stk_top;
inline ll w(int i, int j) {
    ll res = j - i - 1 - L;
    res += sum[j] - sum[i];
    return res * res;
}
inline ll calc(int i, int j) {
    return f[i] + w(i, j);
}
inline int find(int i) {
    int l = stk[stk_top].l, r = stk[stk_top].r;
    while(l <= r) {
        int mid = (l + r) >> 1;
        if(calc(i, mid) < calc(stk[stk_top].x, mid)) {
            r = mid - 1;
        }
        else {
            l = mid + 1;
        }
    }
    return l;
}
int main() {
    scanf("%d%d", &n, &L);
    for (int i = 1; i <= n; i++) {
        scanf("%d", &c[i]);
        sum[i] = sum[i-1] + c[i];
    }
    int cur = 0;
    stk[cur] = {1, n, 0};
    for (int i = 1; i <= n; i++) {
        f[i] = calc(stk[cur].x, i);
        while(i < stk[stk_top].l && calc(i, stk[stk_top].l) < calc(stk[stk_top].x , stk[
            stk_top].l)) stk_top--;
        int l = find(i); stk[stk_top].r = l - 1;
        if(l <= n) stk[++stk_top] = {l, n, i};
        if(i == stk[cur].r) cur++;
```

```
    }
    printf("%lld\n", f[n]);
}
```

2.8.4 区间类 2D1D

```
/*
* 题目: [NOI1995] 石子合并
* URL: https://www.luogu.com.cn/problem/P1880
*/
#include <bits/stdc++.h>
using namespace std;
const int N = 1e3 + 5;
const int inf = 0x3f3f3f3f;
int a[N], sum[N], f[N][N], m[N][N], g[N][N];
int w(int l, int r) {
    return sum[r] - sum[l-1];
}
int main() {
    int n;
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
        scanf("%d", &a[i]);
        sum[i] = sum[i-1] + a[i];
        m[i][i] = i;
    }

    for (int i = n+1; i <= 2*n; i++) {
        a[i] = a[i-n];
        sum[i] = sum[i-1] + a[i];
        m[i][i] = i;
    }
    for (int len = 2; len <= n; ++len) {
        for (int l = 1, r = len; r <= 2*n; l++, r++) {
            f[l][r] = inf;
            g[l][r] = max(g[l+1][r], g[l][r-1]) + w(l, r);
            for (int k = m[l][r-1]; k <= m[l+1][r]; ++k) {
                int tmp = f[l][k] + f[k+1][r] + w(l, r);
                if(f[l][r] > tmp) {
                    f[l][r] = tmp;
                    m[l][r] = k;
                }
            }
        }
    }
    int mx = 0, mn = inf;
    for (int i = 1; i <= n; i++) {
        mn = min(mn, f[i][i+n-1]);
        mx = max(mx, g[i][i+n-1]);
    }
    printf("%d\n", mn);
    printf("%d\n", mx);
```

```

}
```

2.9 单调栈单调队列优化

2.9.1 一维

```
/*
* 题目: P1725 琪露诺
* URL: https://www.luogu.com.cn/problem/P1725
* 状态转移公式: f[i] = max{f[j] + a[i]} (i-r <= j <= i-l)
*/
#include <bits/stdc++.h>
using namespace std;
const int inf = 0x3f3f3f3f;
const int N = 2e5 + 5;
int a[N], dp[N];
int Q[N], h, t; // 队列, head, tail
int main() {
    fill(dp, dp+N, -inf);
    int n, l, r;
    scanf("%d%d%d", &n, &l, &r);
    for (int i = 0; i <= n; i++) {
        scanf("%d", &a[i]);
    }
    h = 0, t = -1;
    Q[++t] = 0; dp[0] = a[0];
    for (int i = l; i <= n; i++) {
        while(h <= t && dp[Q[t]] <= dp[i-l]) t--;
        Q[++t] = i-l;
        while(h <= t && Q[h] < i-r) h++;
        dp[i] = dp[Q[h]] + a[i];
    }
    int mx = -inf;
    for (int i = n-r+1; i <= n; i++) mx = max(mx, dp[i]);
    printf("%d\n", mx);
}
```

2.9.2 二维

```
/*
* 模版: dp值从 (大小固定的) 二维区间的最大值转移过来
* 适用: 只要转移的区间满足双指针的性质, 都可以用单调队列优化
*/
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N = 1e3 + 5;
int a[N][N];
int head[N], tail[N], head2, tail2;
struct {
    ll data;
```

```
    int id;
}row[N], col[N][N];

ll dp[N][N];
void Max(ll &x, ll y) {
    if(x < y) x = y;
}
int main() {
    memset(dp, -1, sizeof dp);
    int n, m, k, u;
    while(~scanf("%d%d%d%d", &n, &m, &k, &u)) {

        for (int i = 1; i <= m; i++) head[i] = tail[i] = 0;

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                scanf("%d", &a[i][j]);
                dp[i][j] = -1;
            }
        }

        dp[1][1] = a[1][1];

        for (int i = 1; i <= n; i++) {
            head2 = tail2 = 0;
            for (int j = 1; j <= m; j++) {
                while(head[j] < tail[j] && col[j][head[j]].id < i - k) head[j]++;
                while(head2 < tail2 && row[head2].id < j - k) head2++;
                if(a[i][j] > 0) {
                    if(dp[i-1][j-1]!=-1) Max(dp[i][j], dp[i-1][j-1] + a[i][j]);
                    if(dp[i][j-1]!=-1) Max(dp[i][j], dp[i][j-1] + a[i][j]);
                    if(dp[i-1][j]!=-1) Max(dp[i][j], dp[i-1][j] + a[i][j]);
                    if(head[j] < tail[j]) {
                        ll now = col[j][head[j]].data;
                        while(head2 < tail2 && row[tail2-1].data <= now) tail2--;
                        row[tail2++] = {now, j};
                    }
                    if(head2 < tail2) {
                        Max(dp[i][j], row[head2].data + a[i][j] - u);
                    }
                    if(head[j] < tail[j]) {
                        tail2--;
                    }
                }
                if(dp[i][j] >= u) {
                    while(head[j] < tail[j] && dp[i][j] >= col[j][tail[j]-1].data)
                        tail[j]--;
                    col[j][tail[j]++] = {dp[i][j], i};
                }
                if(head[j] < tail[j]) {
                    ll now = col[j][head[j]].data;
                    while(head2 < tail2 && row[tail2-1].data <= now) tail2--;
                    row[tail2++] = {now, j};
```

```
            }
        }
    }
    printf("%lld\n", dp[n][m]);
  }
}
```

2.10 矩阵优化

2.10.1 矩阵乘法

```
/*
* 模版: n*n矩阵
* 矩阵乘法: 快速幂加速
*/
#include <stdio.h>
#include <algorithm>
using namespace std;
#define rep(a,b,c) for(int a=b;a<=c;a++)
#define per(a,b,c) for(int a=b;a>=c;a--)
template <typename T> inline void read (T &t){t = 0;char c = getchar();int f = 1;while (c
     < '0' || c > '9'){if (c == '-') f = -f;c = getchar();}while (c >= '0' && c <= '9'){t =
    (t << 3) + (t << 1) + c - '0';c = getchar();} t *= f;}
template <typename T,typename ... Args> inline void read (T &t,Args&... args){read (t);
    read (args...);}
template <typename T> inline void write (T x){if (x < 0){x = -x;putchar ('-');}if (x > 9)
    write (x / 10);putchar (x % 10 + '0');}

struct Matrix {
    double mat[2][2], n;
    Matrix() {
        n = 2;
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++) mat[i][j] = 0;
    }
    Matrix operator * (const Matrix& rhs) const {
        Matrix ret;
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++) {
                ret.mat[i][j] = 0;
                for (int k = 0; k < n; k++)
                    ret.mat[i][j] += mat[i][k] * rhs.mat[k][j];
            }
        return ret;
    }
};

Matrix qpow(Matrix x, int k) {
    Matrix res;
    res.n = x.n;
    rep (i, 0, res.n-1) res.mat[i][i] = 1;
    while (k) {
```

```
        if (k & 1) res = res * x;
        x = x * x;
        k >>= 1;
    }
    return res;
}

int main() {
    int n; double p;
    while(scanf("%d %lf", &n, &p) != EOF) {
        int x[15];
        rep (i, 1, n) read(x[i]);
        sort(x+1, x+n+1);
        double ans = 1;
        Matrix t;
        t.mat[0][0] = p; t.mat[0][1] = 1-p;
        t.mat[1][0] = 1; t.mat[1][1] = 0;

        x[0] = 0;
        rep(i, 1, n) {
            if(x[i] == x[i-1]) continue;
            Matrix tmp = qpow(t, x[i]-x[i-1]-1);
            ans *= (1 - tmp.mat[0][0]);
        }
        printf("%.7f\n", ans);
    }
}
```

2.10.2 广义矩阵乘法

```
/*
* 模版：广义矩阵乘法
* URL: https://www.luogu.com.cn/problem/P2886
* c[i][j] = Max/Min {a[i][k] + a[j][k]}
*/
#include <bits/stdc++.h>
using namespace std;
#define rep(a,b,c) for(int a=b;a<=c;a++)
#define per(a,b,c) for(int a=b;a>=c;a--)
template <typename T> inline void read (T &t){t = 0;char c = getchar();int f = 1;while (c < '0' || c > '9'){if (c == '-') f = -f;c = getchar();}while (c >= '0' && c <= '9'){t = (t << 3) + (t << 1) + c - '0';c = getchar();} t *= f;}
template <typename T,typename ... Args> inline void read (T &t,Args&... args){read (t);read (args...);}
template <typename T> inline void write (T x){if (x < 0){x = -x;putchar ('-');}if (x > 9) write (x / 10);putchar (x % 10 + '0');}
const int N = 250;
const int inf = 0x3f3f3f3f;
void Min(int &x, int y) {if(x > y) x = y;}
struct Matrix {
    int mat[N][N], n;
    Matrix() {
```

```
        memset(mat, inf, sizeof mat);
    }
    inline int* operator [](const int i) { //暴力重载运算符
        return mat[i];
    }
    Matrix operator *(const Matrix& rhs) const {
        Matrix ret; ret.n = n;
        rep (i, 0, n-1) rep (j, 0, n-1) rep (k, 0, n-1) {
            Min(ret.mat[i][j], mat[i][k] + rhs.mat[k][j]);
        }
        return ret;
    }
};
Matrix qpow(Matrix x, int k) {
    assert(k > 0);
    Matrix res = x; k--;
    while (k) {
        if (k & 1) res = res * x;
        x = x * x;
        k >>= 1;
    }
    return res;
}
int re[1005], cnt; // 离散化
int main() {
    int n, t, s, e;
    read(n, t, s, e);
    Matrix g;
    memset(re, -1, sizeof re);
    rep (i, 1, t) {
        int u, v, w;
        read(w, u, v);
        if(~re[u]) u = re[u]; else {re[u] = cnt++; u = re[u];}
        if(~re[v]) v = re[v]; else {re[v] = cnt++; v = re[v];}
        g[u][v] = g[v][u] = min(w, g[u][v]);
    }
    g.n = cnt;
    g = qpow(g, n);
    s = re[s]; e = re[e];
    write(g[s][e]);
}
```

2.11 GarsiaWachs 算法

```
/*
* GarsiaWachs算法专门解决石子合并问题
*/
#include <bits/stdc++.h>
using namespace std;

int n, k, j;
int ans, sum;
```

```cpp
vector<int> v;

int read()
{
    int x = 0, f = 1;
    char c = getchar();
    while (c < '0' || c > '9') {if (c == '-') f = -1; c = getchar();}
    while (c >= '0' && c <= '9') x = x * 10 + c - '0', c = getchar();
    return x * f;
}

int main()
{
    n = read();
    v.push_back(INT_MAX - 1);
    for (int i = 1; i <= n; i++)
        v.push_back(read());
    v.push_back(INT_MAX);

    while (n-- > 1)
    {
        for (k = 1; k <= n; k++)
            if (v[k - 1] < v[k + 1])
                break;
        sum = v[k] + v[k - 1];
        for (j = k - 1; j >= 0; j--)
            if (v[j] > v[k] + v[k - 1])
                break;
        v.erase(v.begin() + k - 1);
        v.erase(v.begin() + k - 1);
        v.insert(v.begin() + j + 1, sum);
        ans += sum;
    }

    printf("%d", ans);
    return 0;
}
```

3 字符串

3.1 KMP

```cpp
/*
 模版: kmp字符串匹配
 */

const int N = 1e6 + 5;
int nxt[N];
char s1[N], s2[N];
void get_next(char* s, int* nxt) {
    int i = 0, j = -1, l = strlen(s);
    nxt[0] = -1;
```

```
    while(i < l) {
        if(j == -1 || s[i] == s[j])
            nxt[++i] = ++j;
        else j = nxt[j];
    }
}
void kmp(char* s, char* p, int* nxt) {
    int i = 0, j = 0, n = strlen(p), m = strlen(s);
    while (i < n) {
        if(j == -1 || p[i] == s[j]) i++, j++;
        else j = nxt[j];
        if(j == m) printf("%d\n", i-m+1), j = nxt[j];
    }
}
int main() {
    scanf("%s%s", s1, s2);
    get_next(s2, nxt);
    kmp(s2, s1, nxt);
    for (int i = 1; i <= strlen(s2); i++)
        printf("%d ", nxt[i]);
}
```

3.2 扩展 KMP

```
/*
扩展KMP求的是对于原串S1的每一个后缀子串与模式串S2的最长公共前缀。
extend[i]表示为以字符串S1中以i为起点的后缀字符串和模式串S2的最长公共前缀长度.
*/
const int N = 2e7 + 7;
int n, m, z[N], p[N];
char a[N], b[N];

inline void Z(char *s, int n) {
    for (int i = 1; i <= n; i++) z[i] = 0;
    z[1] = n;
    for (int i = 2, l = 0, r = 0; i <= n; i++) {
        if (i <= r) z[i] = min(z[i-l+1], r - i + 1);
        while (i + z[i] <= n && s[i+z[i]] == s[z[i]+1]) ++z[i];
        if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
    }
}

inline void exkmp(char *s, int n, char *t, int m) {
    Z(t, m);
    for (int i = 1; i <= n; i++) p[i] = 0;
    for (int i = 1, l = 0, r = 0; i <= n; i++) {
        if (i <= r) p[i] = min(z[i-l+1], r - i + 1);
        while (i + p[i] <= n && s[i+p[i]] == t[p[i]+1]) ++p[i];
        if (i + p[i] - 1 > r) l = i, r = i + p[i] - 1;
    }
}

```

```
int main() {
    rds(a, n), rds(b, m);
    exkmp(a, n, b, m);
    ll ans = 0;
    for (int i = 1; i <= m; i++) ans ^= 1ll * i * (z[i] + 1); // next
    print(ans);
    ans = 0;
    for (int i = 1; i <= n; i++) ans ^= 1ll * i * (p[i] + 1); // extend
    print(ans);
    return 0;
}
```

3.3 最小表示法

```
/*
 模版: 字符串循环同构 (最小表示法)
 返回: 字典序最小的起始位置
 */

const int N = 1e4 + 5;
int Minrp(char* s, int l) {
    int i = 0, j = 1, k = 0;
    while (i < l && j < l && k < l) {
        int d = s[(i+k)%l] - s[(j+k)%l];
        if(!d) k++;
        else {
            if(d > 0) i = i + k + 1;
            else j = j + k + 1;
            if(i == j) j++;
            k = 0;
        }
    }
    return i;
}
char str[N];
int main() {
    int t; scanf("%d", &t);
    while (t--) {
        int n;
        scanf("%d", &n);
        scanf("%s", str);
        printf("%d\n", Minrp(str, n));
    }
}
```

3.4 哈希

```
/*
* 前缀hash: 求子串的hash值
* 注意: string的size()要强转int类型!
*/
```

```cpp
const int MAXLEN = 2e6 + 5;
typedef unsigned long long ull;
ull base = 1331;
ull h[MAXLEN], p[MAXLEN];
ull get_hash(int l, int r) {
    return h[r] - h[l-1] * p[r-l+1];
}
ull hashs(string& s) {
    p[0] = 1; h[0] = 0;
    int sz = s.size();
    rep(i, 0, sz-1) {
        p[i+1] = p[i] * base;
        h[i+1] = h[i] * base + s[i];
    }
    return h[sz];
}
```

3.5 马拉车

```cpp
/*
    Manacher算法：最长回文子串
*/
const int N = 10010;
char s[N];
bool mp[300];
int len, str[2*N], Len[2*N];
void getstr(int l, int r) {//重定义字符串
    int k = 0;
    str[k++] = '@';//开头加个特殊字符防止越界
    for (int i = l; i <= r; i++) {
        str[k++] = '#';
        str[k++] = s[i];
    }
    str[k++] = '#';
    len = k;
    str[k] = 0;//字符串尾设置为0，防止越界
}
int manacher() {
    int mx = 0, id;//mx为最右边，id为中心点
    int maxx = 0;
    for (int i = 1; i < len; i++) {
        if (mx > i) Len[i] = min(mx - i, Len[2 * id - i]);//判断当前点超没超过mx
        else Len[i] = 1;//超过了就让他等于1，之后再进行查找
        while (str[i + Len[i]] == str[i - Len[i]]) Len[i]++;//判断当前点是不是最长回文子串，不断的向右扩展
        if (Len[i] + i > mx) {//更新mx
            mx = Len[i] + i;
            id = i;//更新中间点
            maxx = max(maxx, Len[i]);//最长回文字串长度
        }
    }
    return (maxx - 1);
```

```
}

getstr(l, r);
ans = max(ans, manacher());
// cout << l << " " << r << " " << manacher() << endl;
```

3.6 字典树

3.6.1 Trie

```
struct trie {
  int nex[MAX_NODE][CHAR_NUM], cnt;
  bool exist[MAX_NODE]; // 该结点结尾的字符串是否存在

  void insert(char *s, int l) { // 插入字符串
    int p = 0;
    for (int i = 0; i < l; i++) {
      int c = s[i] - 'a';
      if (!nex[p][c]) nex[p][c] = ++cnt; // 如果没有，就添加结点
      p = nex[p][c];
    }
    exist[p] = 1;
  }
  bool find(char *s, int l) { // 查找字符串
    int p = 0;
    for (int i = 0; i < l; i++) {
      int c = s[i] - 'a';
      if (!nex[p][c]) return 0;
      p = nex[p][c];
    }
    return exist[p];
  }
};
```

3.6.2 01-Trie

```
/*
- 维护亦或和，支持插入、删除、全局加一操作
- 这里的 `MAXH` 指 trie 的深度，也就是强制让每一个叶子节点到根的距离为 `MAXH` 。
*/
const int _ = 526010;
int n;
int V[_];
int debug = 0;
namespace trie {
const int MAXH = 21;
int ch[_ * (MAXH + 1)][2], w[_ * (MAXH + 1)], xorv[_ * (MAXH + 1)];
int tot = 0;
int mknode() {
  ++tot;
  ch[tot][1] = ch[tot][0] = w[tot] = xorv[tot] = 0;
  return tot;
```

```cpp
}
void maintain(int o) {
  w[o] = xorv[o] = 0;
  if (ch[o][0]) {
    w[o] += w[ch[o][0]];
    xorv[o] ^= xorv[ch[o][0]] << 1;
  }
  if (ch[o][1]) {
    w[o] += w[ch[o][1]];
    xorv[o] ^= (xorv[ch[o][1]] << 1) | (w[ch[o][1]] & 1);
  }
  w[o] = w[o] & 1;
}
void insert(int &o, int x, int dp) {
  if (!o) o = mknode();
  if (dp > MAXH) return (void)(w[o]++);
  insert(ch[o][x & 1], x >> 1, dp + 1);
  maintain(o);
}
int marge(int a, int b) {
  if (!a) return b;
  if (!b) return a;
  w[a] = w[a] + w[b];
  xorv[a] ^= xorv[b];
  ch[a][0] = marge(ch[a][0], ch[b][0]);
  ch[a][1] = marge(ch[a][1], ch[b][1]);
  return a;
}
void addall(int o) {
  swap(ch[o][0], ch[o][1]);
  if (ch[o][0]) addall(ch[o][0]);
  maintain(o);
}
} // namespace trie
int rt[_];
long long Ans = 0;
vector<int> E[_];
void dfs0(int o) {
  for (int i = 0; i < E[o].size(); i++) {
    int node = E[o][i];
    dfs0(node);
    rt[o] = trie::marge(rt[o], rt[node]);
  }
  trie::addall(rt[o]);
  trie::insert(rt[o], V[o], 0);
  Ans += trie::xorv[rt[o]];
}
int main() {
  n = read();
  for (int i = 1; i <= n; i++) V[i] = read();
  for (int i = 2; i <= n; i++) E[read()].push_back(i);
  dfs0(1);
  printf("%lld", Ans);
```

```
  return 0;
}
```

3.7 后缀数组

3.7.1 总结

应用

比较一个字符串的两个子串的大小关系

假设需要比较的是 $A = S[a..b]$ 和 $B = S[c..d]$ 的大小关系。

若 $lcp(a, c) \geq \min(|A|, |B|)$，$A < B \iff |A| < |B|$。

否则，$A < B \iff rk[a] < rk[b]$。

不同子串的数目

子串就是后缀的前缀，所以可以枚举每个后缀，计算前缀总数，再减掉重复。

"前缀总数"其实就是子串个数，为 $n(n+1)/2$。

如果按后缀排序的顺序枚举后缀，每次新增的子串就是除了与上一个后缀的 LCP 剩下的前缀。这些前缀一定是新增的，否则会破坏 $lcp(sa[i], sa[j]) = \min\{height[i+1..j]\}$ 的性质。只有这些前缀是新增的，因为 LCP 部分在枚举上一个前缀时计算过了。

所以答案为：$\frac{n(n+1)}{2} - \sum\limits_{i=2}^{n} height[i]$

3.7.2 倍增

```
/*
后缀数组：把字符串S的每个后缀按字典序排序
LCP：最长公共前缀

定义：
- sa[i]：排名为i的后缀首字母的下标
- rk[i]：首字母下标为i的后缀的排名
- LCP(i,j)：Suffix(sa[i])和Suffix(sa[j])的最长公共前缀

定理：
height[i] = LCP(i,i-1)：排名相邻的两个后缀数组的公共前缀长度
令h[i] = height[rk[i]]，可证**h[i]>=h[i-1]-1**（递推条件，时间复杂度收敛）

注意：
1. 不要在main函数内定义n和m变量，两者为全局变量
2. s数组内的元素最小值为1
*/
char s[N];
int y[N], x[N], c[N], sa[N], rk[N], height[N], ans[N];
int f[N][20];
int n, m;
void get_SA() {
    for (int i = 1; i <= m; i++) c[i] = 0;
```

```
    for (int i = 1; i <= n; i++) ++c[x[i]=s[i]];
    for (int i = 2; i <= m; i++) c[i] += c[i-1];
    for (int i = n; i >= 1; i--) sa[c[x[i]]--] = i;
    for (int k = 1; k <= n; k <<= 1) {
        int num = 0;
        for (int i = n - k + 1; i <= n; i++) y[++num] = i;
        for (int i = 1; i <= n; i++) if(sa[i] > k) y[++num] = sa[i] - k;
        for (int i = 1; i <= m; i++) c[i] = 0;
        for (int i = 1; i <= n; i++) ++c[x[i]];
        for (int i = 2; i <= m; i++) c[i] += c[i-1];
        for (int i = n; i >= 1; i--) sa[c[x[y[i]]]--] = y[i], y[i] = 0;
        swap(x, y);
        x[sa[1]] = 1;
        num = 1;
        for (int i = 2; i <= n; i++)
            x[sa[i]] = (y[sa[i]] == y[sa[i-1]] && y[sa[i]+k] == y[sa[i-1]+k]) ? num : ++
                num;
        if(num == n) break;
        m = num;
    }
}
void get_height() {
    int k = 0;
    for (int i = 1; i <= n; i++) rk[sa[i]] = i;
    for (int i = 1; i <= n; i++) {
        if(rk[i] == 1) continue;
        if(k) --k;
        int j = sa[rk[i] - 1];
        while(j + k <= n && i + k <= n && s[i+k] == s[j+k]) ++k;
        height[rk[i]] = k;
    }
}
void init() {
    for (int i = 1; i <= n; i++) {
        f[i][0] = height[i];
    }

    for (int j = 1; (1<<j) <= n; j++) {
        for (int i = 1; i + (1<<j) - 1 <= n; i++) {
            f[i][j] = min(f[i][j-1], f[i + (1<<(j-1))][j-1]);
        }
    }
}
int lcp(int l, int r) {
    int a = rk[l], b = rk[r];
    if(a > b) swap(a, b);
    ++a;
    int k = log2(b - a + 1);
    return min(f[a][k], f[b-(1<<k)+1][k]);
}

scanf("%s", s+1);
n = strlen(s+1);
```

```
m = 123;
get_SA();
get_height();
init();
```

3.7.3 DC3

```
/*
DC3实现SA复杂度为O(n), 常数大, 考虑在数据量大的时候使用。
注意:
1.MAXN开n的十倍大小;
2.dc3(r,sa,n+1,mx+1);r为待后缀处理的数组,sa为存储排名位置的数组,n+1和mx+1 都和倍增一样
3.calheight(r,sa,n);和倍增一样
*/
#define F(x) ((x) / 3 + ((x) % 3 == 1 ? 0 : tb))
#define G(x) ((x) < tb ? (x)*3 + 1 : ((x)-tb) * 3 + 2)
const int MAXN = 1e7 + 5; //n*10
int wa[MAXN], wb[MAXN], wv[MAXN], wws[MAXN];
int sa[MAXN], rk[MAXN], height[MAXN], r[MAXN];
int n;
char s[MAXN];
void sort(int *r, int *a, int *b, int n, int m)
{
    int i;
    for (i = 0; i < n; i++) wv[i] = r[a[i]];
    for (i = 0; i < m; i++) wws[i] = 0;
    for (i = 0; i < n; i++) wws[wv[i]]++;
    for (i = 1; i < m; i++) wws[i] += wws[i - 1];
    for (i = n - 1; i >= 0; i--) b[--wws[wv[i]]] = a[i];
    return;
}
int c0(int *r, int a, int b)
{
    return r[a] == r[b] && r[a + 1] == r[b + 1] && r[a + 2] == r[b + 2];
}
int c12(int k, int *r, int a, int b)
{
    if (k == 2) return r[a] < r[b] || r[a] == r[b] && c12(1, r, a + 1, b + 1);
    else return r[a] < r[b] || r[a] == r[b] && wv[a + 1] < wv[b + 1];
}

void dc3(int *r, int *sa, int n, int m)
{
    int i, j, *rn = r + n, *san = sa + n, ta = 0, tb = (n + 1) / 3, tbc = 0, p;
    r[n] = r[n + 1] = 0;
    for (i = 0; i < n; i++) if (i % 3 != 0) wa[tbc++] = i;
    sort(r + 2, wa, wb, tbc, m);
    sort(r + 1, wb, wa, tbc, m);
    sort(r, wa, wb, tbc, m);
    for (p = 1, rn[F(wb[0])] = 0, i = 1; i < tbc; i++)
        rn[F(wb[i])] = c0(r, wb[i - 1], wb[i]) ? p - 1 : p++;
    if (p < tbc) dc3(rn, san, tbc, p);
```

```
    else for (i = 0; i < tbc; i++) san[rn[i]] = i;
    for (i = 0; i < tbc; i++)
        if (san[i] < tb) wb[ta++] = san[i] * 3;
    if (n % 3 == 1) wb[ta++] = n - 1;
    sort(r, wb, wa, ta, m);
    for (i = 0; i < tbc; i++)
        wv[wb[i] = G(san[i])] = i;
    for (i = 0, j = 0, p = 0; i < ta && j < tbc; p++)
        sa[p] = c12(wb[j] % 3, r, wa[i], wb[j]) ? wa[i++] : wb[j++];
    for (; i < ta; p++) sa[p] = wa[i++];
    for (; j < tbc; p++) sa[p] = wb[j++];
    return;
}
void calheight(int *r, int *sa, int n)
{
    int i, j, k = 0;
    for (i = 1; i <= n; ++i)
        rk[sa[i]] = i;
    for (i = 0; i < n; height[rk[i++]] = k)
        for (k ? k-- : 0, j = sa[rk[i] - 1]; r[i + k] == r[j + k]; ++k);
    return;
}

scanf("%s", s);
int mx = -1;
n = strlen(s);
for (int i = 0; i < n; i++)
{
    r[i] = s[i];
    if (r[i] > mx) mx = r[i];
}
r[n] = 0;
dc3(r, sa, n + 1, mx + 1);
calheight(r, sa, n);
```

3.8 单串 SAM

概述

直观上，字符串的 Suffix Automation(SAM) 可以理解为给定字符串的 **所有子串**的压缩形式。值得注意的事实是，SAM 将所有的这些信息以高度压缩的形式储存。对于一个长度为 n 的字符串，它的空间复杂度仅为 $O(n)$ 。此外，构造 SAM 的时间复杂度仅为 $O(n)$ 。准确地说，一个 SAM 最多有 $2n-1$ 个节点和 $3n-4$ 条转移边。

性质

裸的后缀自动机仅仅是一个可以接收子串的自动机，在它的状态结点上维护的性质才是解题的关键。

子串可以根据它们结束的位置 $endpos$ 被划分为多个等价类，SAM 由初始状态 t_0 和与每一个 $endpos$ 等价类对应的每个状态组成。

一个构造好的 SAM 实际上包含了两个图：

1. 由 $next$ 数组组成的 DAG 图；
2. 由 $link$ 指针构成的 $parent$ 树。

SAM 的状态结点包含了很多重要的信息：

- $maxlen$：即代码中 len 变量，它表示该状态能够接受的最长的字符串长度。
- $minlen$：表示该状态能够接受的最短的字符串长度。实际上等于该状态的 $link$ 指针指向的结点的 $len+1$。
- $maxlen-minlen+1$：表示该状态能够接受的不同的字符串数。
- $right$：即结束位置的集合 $endpos-set$ 的个数，表示这个状态在字符串中出现了多少次，该状态能够表示的所有字符串均出现过 $right$ 次。
- $link$：$link$ 指向了一个能够表示当前状态表示的所有字符串的最长公共后缀的结点。所有的状态的 $link$ 指针构成了一个 $parent$ 树，恰好是字符串的逆序的后缀树。
- $parent$ 树的拓扑序：序列中第 i 个状态的子结点必定在它之后，父结点必定在它之前。

如果我们从任意状态 v_0 开始顺着后缀链接遍历，总会到达初始状态 t_0 。这种情况下我们可以得到一个互不相交的区间 $[\operatorname{minlen}(v_i),\operatorname{len}(v_i)]$ 的序列，且它们的并集形成了连续的区间 $[0,\operatorname{len}(v_0)]$ 。

拓展

设字符串的长度为 n ，考虑 $extend$ 操作中 cur 变量的值，这个节点对应的状态是执行 $extend$ 操作时的当前字符串，即字符串的一个前缀，每个前缀有一个终点。这样得到的 n 个节点，对应了 n 个不同的终点。设第 i 个节点为 v_i ，对应的是 $S_{1\dots i}$ ，终点是 i 。姑且把这些节点称之为“终点节点”。

考虑给 SAM 赋予树形结构，树的根为 0，且其余节点 v 的父亲为 $\operatorname{link}(v)$ 。则这棵树与原 SAM 的关系是：

- 每个节点的终点集合等于其子树内所有终点节点对应的终点的集合。

在此基础上可以给每个节点赋予一个最长字符串，是其终点集合中任意一个终点开始往前取 len 个字符得到的字符串。每个这样的字符串都一样，且 len 恰好是满足这个条件的最大值。

这些字符串满足的性质是：

- 如果节点 A 是 B 的祖先，则节点 A 对应的字符串是节点 B 对应的字符串的后缀。

这条性质把字符串所有前缀组成了一棵树，且有许多符合直觉的树的性质。例如，$S_{1\dots p}$ 和 $S_{1\dots q}$ 的最长公共后缀对应的字符串就是 v_p 和 v_q 对应的 LCA 的字符串。实际上，这棵树与将字符串 S 翻转后得到字符串的压缩后缀树结构相同。

构造

构造 SAM 是在线算法，逐个加入字符串的字符过程中，每一步对应的维护 SAM。

为了保证线性的空间复杂度，状态节点中将只保存 len 和 link 的值和每个状态的转移列表。

一开始 SAM 只包含一个状态 t_0 ，编号为 0 （其它状态的编号为 $1, 2, \ldots$ ）。为了方便，对于状态 t_0 我们指定 $\text{len} = 0$ 、$\text{link} = -1$ （-1 表示虚拟状态）。

现在，任务转化为实现给当前字符串添加一个字符 c 的过程。算法流程如下：

1. 令 $last$ 为添加字符 c 之前，整个字符串对应的状态（一开始我们设 $last = 0$ ，算法的最后一步更新 $last$ ）。
2. 创建一个新的状态 cur ，并将 $\text{len}(cur)$ 赋值为 $\text{len}(last) + 1$ ，在这时 $\text{link}(cur)$ 的值还未知。
3. 现在我们按以下流程进行（从状态 $last$ 开始）。如果还没有到字符 c 的转移，我们就添加一个到状态 cur 的转移，遍历后缀链接。如果在某个点已经存在到字符 c 的转移，我们就停下来，并将这个状态标记为 p 。
4. 如果没有找到这样的状态 p ，我们就到达了虚拟状态 -1 ，我们将 $\text{link}(cur)$ 赋值为 0 并退出。
5. 假设现在我们找到了一个状态 p ，其可以通过字符 c 转移。我们将转移到的状态标记为 q 。
6. 现在我们分类讨论两种状态，要么 $\text{len}(p) + 1 = \text{len}(q)$ ，要么不是。
7. 如果 $\text{len}(p) + 1 = \text{len}(q)$ ，我们只要将 $\text{link}(cur)$ 赋值为 q 并退出。
8. 否则就会有些复杂。需要复制状态 q ：我们创建一个新的状态 $clone$ ，复制 q 的除了 len 的值以外的所有信息（后缀链接和转移）。我们将 $\text{len}(clone)$ 赋值为 $\text{len}(p) + 1$ 。复制之后，我们将后缀链接从 cur 指向 $clone$ ，也从 q 指向 $clone$ 。最终我们需要使用后缀链接从状态 p 往回走，只要存在一条通过 p 到状态 q 的转移，就将该转移重定向到状态 $clone$ 。
9. 以上三种情况，在完成这个过程之后，我们将 $last$ 的值更新为状态 cur 。

标记终止状态在构造完完整的 SAM 后找到所有的终止状态。

为此，我们从对应整个字符串的状态（存储在变量 $last$ 中），遍历它的后缀链接，直到到达初始状态。我们将所有遍历到的节点都标记为终止节点。容易理解这样做我们会准确地标记字符串 s 的所有后缀，这些状态都是终止状态。

```
/*
* 后缀自动机 Suffix Automation (SAM)
* 功能:
* 1. 多个串的最长公共子串
* 2. 统计不同子串数量
* 3. 统计子串出现次数
* 4. dfs2: 查询字典序第k大的子串 (不考虑重复情况)
* 5. dfs3: 查询字典序第k大的子串 (考虑重复情况)
*/
```

```cpp

#include <bits/stdc++.h>
using namespace std;

namespace SAM {
    const int MAXLEN = 1e6 + 5;
    struct state {
        int len, link;
        int next[26]; // 字符集很小时，常数优化
    } st[MAXLEN * 2];
    int sz, last;
    // sz: 状态总数 [0,sz)
    // last: 最后插入的状态编号

    int cnt[MAXLEN * 2], topo[MAXLEN * 2], d[MAXLEN * 2];
    // cnt: 从起点到节点表示的字符串在子串中出现的次数
    // topo: parent树中的拓扑序

    void sam_init() { // 初始化
        st[0].len = 0;
        st[0].link = -1;
        sz++;
        last = 0;
    }

    void sam_extend(int c) { // 插入时是 ch-'a' 这种形式
        int cur = sz++;
        st[cur].len = st[last].len + 1;
        cnt[cur] = 1; // 次数统计

        int p = last;
        while (p != -1 && !st[p].next[c]) {
            st[p].next[c] = cur;
            p = st[p].link;
        }

        if (p == -1) {
            st[cur].link = 0;
        } else {
            int q = st[p].next[c];

            if (st[p].len + 1 == st[q].len) {
                st[cur].link = q;
            }
            else {
                int clone = sz++;
                st[clone].len = st[p].len + 1;
                for (int i = 0; i < 26; i++)
                    st[clone].next[i] = st[q].next[i];
                st[clone].link = st[q].link;

                while (p != -1 && st[p].next[c] == q) {
                    st[p].next[c] = clone;
```

```
                p = st[p].link;
            }

            st[q].link = st[cur].link = clone;
        }
    }

    last = cur;
    // ans += st[cur].len - st[st[cur].link].len; 统计不同子串的个数
}

void Topo_init() { // 求出parent树的Topo序
    int mx = 0;
    memset(d, 0, sizeof d);
    for (int i = 1; i < sz; i++) mx = max(mx, st[i].len), d[st[i].len]++;
    for (int i = 1; i <= mx; i++) d[i] += d[i-1];
    for (int i = 1; i < sz; i++) topo[d[st[i].len]--] = i;

    // 求出现次数
    for (int i = sz-1; i > 0; i--) cnt[st[topo[i]].link] += cnt[topo[i]];
}
}
using namespace SAM;

void debug() {
    for (int i = 0; i < sz; i++) {
        for (int c = 0; c < 26; c++) {
            if(st[i].next[c]) {
                cout << i << " -- " << char(c+'a') << " --> " << st[i].next[c] << endl;
            }
        }
    }
}
int nlcs[MAXLEN * 2], lcs[MAXLEN * 2];
void calc(char t[]) { // 多个字符串的最长公共子串
    int m = strlen(t+1), now = 0, len = 0;
    for (int i = 0; i < sz; i++) nlcs[i] = 0;
    for (int i = 1; i <= m; i++) {
        if(st[now].next[t[i]-'a']) {
            len++;
            now = st[now].next[t[i]-'a'];
        }
        else {
            while(now && !st[now].next[t[i]-'a']) now = st[now].link;
            len = st[now].len;
            if(st[now].next[t[i]-'a']) {
                now = st[now].next[t[i]-'a'];
                len++;
            }
        }
        nlcs[now] = max(nlcs[now], len);
    }
    for (int i = sz-1; i > 0; i--) {
```

```
        int u = topo[i], v = st[topo[i]].link;
        nlcs[v] = max(nlcs[v], nlcs[u]);
    }
    for (int i = 0; i < sz; i++) lcs[i] = min(lcs[i], nlcs[i]);
}
int dp[MAXLEN * 2][2];
void dfs(int u) {
    if(dp[u][0]) return ;
    if(u > 0) {
        dp[u][0] = 1;
        dp[u][1] = cnt[u];
    }
    for (char ch = 'a'; ch <= 'z'; ch++) {
        if(!st[u].next[ch-'a']) continue;
        int v = st[u].next[ch-'a'];
        // printf("u=%d, v=%d, ch=%c\n", u, v, ch);
        dfs(v);
        dp[u][0] += dp[v][0];
        dp[u][1] += dp[v][1];
    }
}
char s[MAXLEN], t[MAXLEN];
int main() {
    scanf("%s", s+1);
    int n = strlen(s+1);

    sam_init();
    for (int i = 1; i <= n; i++) sam_extend(s[i]-'a');
    for (int i = 0; i < sz; i++) lcs[i] = st[i].len;
    Topo_init();
    while(~scanf("%s", t+1)) {
        calc(t);
    }
    int ans = 0;
    for (int i = 0; i < sz; i++) ans = max(ans, lcs[i]);
    printf("%d\n", ans);
}

/*
void dfs(int u) {
    if(dp[u][0]) return ;
    if(u > 0) {
        dp[u][0] = 1;
        dp[u][1] = cnt[u];
    }
    for (char ch = 'a'; ch <= 'z'; ch++) {
        if(!st[u].next[ch-'a']) continue;
        int v = st[u].next[ch-'a'];
        // printf("u=%d, v=%d, ch=%c\n", u, v, ch);
        dfs(v);
        dp[u][0] += dp[v][0];
        dp[u][1] += dp[v][1];
    }
```

```
}
void dfs2(int u, int k) {
    if(u > 0 && k == 1) return ;
    k -= (u > 0);
    for (char ch = 'a'; ch <= 'z'; ch++) {
        if(!st[u].next[ch-'a']) continue;
        int v = st[u].next[ch-'a'];
        // printf("ch=%c,v=%d,k=%d,dp=%d\n", ch, v, k, dp[v][op]);
        if(k > dp[v][0]) k -= dp[v][0];
        else {
            printf("%c", ch);
            dfs2(v, k);
            break;
        }
    }
}
void dfs3(int u, int k) {
    if(u > 0) {
        if(k <= cnt[u]) return ;
        k -= cnt[u];
    }
    for (char ch = 'a'; ch <= 'z'; ch++) {
        if(!st[u].next[ch-'a']) continue;
        int v = st[u].next[ch-'a'];
        if(k > dp[v][1]) k -= dp[v][1];
        else {
            printf("%c", ch);
            dfs3(v, k);
            break;
        }
    }
}
int main() {
    scanf("%s", s + 1);
    int n = strlen(s + 1);
    sam_init();
    for (int i = 1; i <= n; i++) sam_extend(s[i]-'a');
    Topo();
    dfs(0);
    int op, k;
    while(~scanf("%d%d", &op, &k)) {
        if(!op) {
            if(dp[0][0] >= k) dfs2(0, k);
            else printf("-1");
        }
        else {
            if(dp[0][1] >= k) dfs3(0, k);
            else printf("-1");
        }
        printf("\n");
    }
}
*/
```

```
}
```

3.9 多串 SAM

概述

后缀自动机 (suffix automaton, SAM) 是用于处理单个字符串的子串问题的强力工具。

而广义后缀自动机 (General Suffix Automaton) 则是将后缀自动机整合到字典树中来解决对于多个字符串的子串问题

实现

- 由于整个 BFS 的过程得到的顺序，其父节点始终在变化，所以并不需要保存 ‘last‘ 指针。
- 插入操作中，‘int cur = next[last][c];‘ 与正常后缀自动机的 ‘int cur = tot++;‘ 有差异，因为我们插入的节点已经在树型结构中完成了，所以只需要直接获取即可
- 在 *clone* 后的数据拷贝中，有这样的判断 ‘next[clone][i] = len[next[q][i]] != 0 ? next[q][i] : 0;‘ 这与正常的后缀自动机的直接赋值 ‘next[clone][i] = next[q][i];‘ 有一定差异，此次是为了避免更新了 ‘len‘ 大于当前节点的值。由于数组中 ‘len‘ 当且仅当这个值被 BFS 遍历并插入到后缀自动机后才会被赋值

```cpp
/*
在线
出现次数统计：可能会有重复的串，所以每个返回的节点加cnt
*/
#include <bits/stdc++.h>
using namespace std;
#define MAXN 600005
#define CHAR_NUM 26
char s[MAXN];
int rt[MAXN];
int cnt[MAXN], c[MAXN], id[MAXN];
// cnt[i]: 出现次数统计
int tot = 1,link[MAXN],len[MAXN],trans[MAXN][CHAR_NUM];
//link[i]: 后缀链接
//trans[i]: 状态转移数组
inline int insert(int ch, int last){
    if(trans[last][ch]){
        int p=last,x=trans[p][ch];
        if(len[p]+1==len[x])return x;//即最初的特判1
        else{
            int y=++tot;len[y]=len[p]+1;
            for(int i=0;i<26;++i)trans[y][i]=trans[x][i];
            while(p&&trans[p][ch]==x)trans[p][ch]=y,p=link[p];
            link[y]=link[x],link[x]=y;
            return y;//即最初的特判2
        }
    }
    int z=++tot,p=last;len[z]=len[last]+1;
```

```
    while(p&&!trans[p][ch])trans[p][ch]=z,p=link[p];
    if(!p)link[z]=1;
    else{
        int x=trans[p][ch];
        if(len[p]+1==len[x])link[z]=x;
        else{
            int y=++tot;len[y]=len[p]+1;
            for(int i=0;i<26;++i)trans[y][i]=trans[x][i];
            while(p&&trans[p][ch]==x)trans[p][ch]=y,p=link[p];
            link[y]=link[x],link[z]=link[x]=y;
        }
    }
    return z;
}
int main() {
    int n, q;
    scanf("%d%d", &n, &q);
    scanf("%s", s+1);
    rt[1] = insert(s[1] - 'A', 1);
    for (int i = 2; i <= n; i++) {
        int u; scanf("%d", &u);
        rt[i] = insert(s[i] - 'A', rt[u]);
    }

    // 次数统计
    for (int i = 1; i <= n; i++) cnt[rt[i]]++;
    for (int i = 1; i <= tot; i++) c[i] = 0;
    for (int i = 1; i <= tot; i++) c[len[i]]++;
    for (int i = 2; i <= tot; i++) c[i] += c[i-1];
    for (int i = 1; i <= tot; i++) id[c[len[i]]--] = i;
    for (int i = tot; i >= 1; i--) {
        int u = id[i];
        cnt[link[u]] += cnt[u];
    }

    while (q--) {
        int X, L;
        scanf("%d%d", &X, &L);
        int p = rt[X];
        while(len[link[p]] >= L) {
            p = link[p];
        }
        printf("%d\n", cnt[p]);
    }
}

/*
* 题目: P6139 【模板】广义后缀自动机 (广义 SAM)
* URL: https://www.luogu.com.cn/problem/P6139
* 题意: 统计多串的不同子串个数
*/
#include <bits/stdc++.h>
using namespace std;
```

```
#define MAXN 2000000 // 双倍字符串长度
#define CHAR_NUM 30 // 字符集个数，注意修改下方的 (-'a')
struct exSAM {
  int len[MAXN]; // 节点长度
  int link[MAXN]; // 后缀链接，link
  int next[MAXN][CHAR_NUM]; // 转移
  int tot; // 节点总数：[0, tot)
  void init() {
    tot = 1;
    link[0] = -1;
  }
  int insertSAM(int last, int c) {
    int cur = next[last][c];
    if (len[cur]) return cur;
    len[cur] = len[last] + 1;
    int p = link[last];
    while (p != -1) {
      if (!next[p][c])
        next[p][c] = cur;
      else
        break;
      p = link[p];
    }
    if (p == -1) {
      link[cur] = 0;
      return cur;
    }
    int q = next[p][c];
    if (len[p] + 1 == len[q]) {
      link[cur] = q;
      return cur;
    }
    int clone = tot++;
    for (int i = 0; i < CHAR_NUM; ++i)
      next[clone][i] = len[next[q][i]] != 0 ? next[q][i] : 0;
    len[clone] = len[p] + 1;
    while (p != -1 && next[p][c] == q) {
      next[p][c] = clone;
      p = link[p];
    }
    link[clone] = link[q];
    link[cur] = clone;
    link[q] = clone;
    return cur;
  }
  int insertTrie(int cur, int c) {
    if (next[cur][c]) return next[cur][c];
    return next[cur][c] = tot++;
  }
  void insert(const string &s) {
    int root = 0;
    for (auto ch : s) root = insertTrie(root, ch - 'a');
  }
```

```
  void insert(const char *s, int n) {
    int root = 0;
    for (int i = 0; i < n; ++i) root = insertTrie(root, s[i] - 'a');
  }
  void build() {
    queue<pair<int, int>> q;
    for (int i = 0; i < 26; ++i)
      if (next[0][i]) q.push({i, 0});
    while (!q.empty()) {
      auto item = q.front();
      q.pop();
      auto last = insertSAM(item.second, item.first);
      for (int i = 0; i < 26; ++i)
        if (next[last][i]) q.push({i, last});
    }
  }
} T;
int main() {
  int n;
  scanf("%d", &n);
  T.init();
  for (int i = 1; i <= n; i++) {
    string s;
    cin >> s;
    T.insert(s);
  }
  T.build();
  long long ans = 0;
  for (int i = 1; i < T.tot; i++) {
    ans += T.len[i] - T.len[T.link[i]];
  }
  printf("%lld\n", ans);
}
```

3.10 后缀树

```
/*
1.时间复杂度
注意到，在从左到右扫描字符串的过程中，最长后缀回文串的左边界只可能向右移动，并且最多移动n次，与后缀链接边相对应的左边界也只可能向右移动，并且最多移动n词。因此总的时间复杂度是O(|S|)或者说O(N)的。
2.空间复杂度
空间复杂度为O(|alphbet|*N)，还有其他几个数组，可以忽略掉。对于小写英文字母表|alphabet|=26。
3.应用
末尾追加一个字符，会产生多少个新的回文串？
举个例子，如果我们在字符串aba后面添加一个新的字符a，已经存在的回文串有a，b，aba，新产生的回文串为aa。
根据前面的讨论，这个问题的答案只可能是0或者1，当我们更新回文树的时候，插入这个新的字符，如果新产生了新节点，那么答案就是1，否则就是0。
4.回文子串的数目
给定一个字符串，计数这个字符串当中有多少个回文子串。比如，aba有四个：两个a,一个b，一个aba。
这个问题其实就是上面的代码所解决的问题，当我们扫描到一个新字符的时候，将结果累加上以这个字符
```

```
    结尾的后缀回文字符串个数，这个数字就是新节点通过后缀链接边可达的节点个数，为了高效计数，
    可以在每个节点新增一个域num，表示由该节点出发的链接长度。
对于根节点而言，链长为0，对于其他节点，链长为其后续节点的链长 + 1.
这个问题还可以用Manacher’ s algorithm求解，时间复杂度也是O(N)。但回文树相对更好写并且应用的
    范围更广。
5.回文串出现的个数统计
这个问题要求统计出每个回文串各出现了多少次，解决的思路和上面类似，每扫描一个新的字符x时，就
    对新出现的最长后缀回文串以及它可达的所有回文串计数加1。
为了加快更新速度，需要类似于线段树那样采用一个延迟更新的策略，就不多说了。
最后再进行一遍计数值的传播更新，就可以得到所有回文串出现的次数了。
*/
#include <bits/stdc++.h>
using namespace std;

const int MAXN = 1005;

struct node {
    int next[26];
    int len;
    int sufflink;
    int num;
};

int len;
char s[MAXN];
node tree[MAXN];
int num; // node 1 - root with len -1, node 2 - root with len 0
int suff; // max suffix palindrome
long long ans;

bool addLetter(int pos) {
    int cur = suff, curlen = 0;
    int let = s[pos] - 'a';

    while (true) {
        curlen = tree[cur].len;
        if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos])
            break;
        cur = tree[cur].sufflink;
    }
    if (tree[cur].next[let]) {
        suff = tree[cur].next[let];
        return false;
    }

    num++;
    suff = num;
    tree[num].len = tree[cur].len + 2;
    tree[cur].next[let] = num;

    if (tree[num].len == 1) {
        tree[num].sufflink = 2;
        tree[num].num = 1;
```

```
        return true;
    }

    while (true) {
        cur = tree[cur].sufflink;
        curlen = tree[cur].len;
        if (pos - 1 - curlen >= 0 && s[pos - 1 - curlen] == s[pos]) {
            tree[num].sufflink = tree[cur].next[let];
            break;
        }
    }
    tree[num].num = 1 + tree[tree[num].sufflink].num;
    return true;
}

void initTree() {
    num = 2; suff = 2;
    tree[1].len = -1; tree[1].sufflink = 1;
    tree[2].len = 0; tree[2].sufflink = 1;
}

int main() {
    scanf("%s", s);
    len = strlen(s);

    initTree();

    for (int i = 0; i < len; i++) {
        addLetter(i);
        ans += tree[suff].num;
    }
    cout << ans << endl;
    return 0;
}
```

3.11 Lyndon 分解

Lyndon 串: 对于字符串，如果的字典序严格小于的所有后缀的字典序，我们称是简单串，或者 Lyndon 串。举一些例子，‘a‘ , ‘b‘ , ‘ab‘ , ‘aab‘ , ‘abb‘ , ‘ababb‘ , ‘abcd‘ 都是 Lyndon 串。当且仅当的字典序严格小于它的所有非平凡的循环同构串时，才是 Lyndon 串。

Lyndon 分解: 串的 Lyndon 分解记为 $s = w_1 w_2 ... w_n$ ，其中所有 w_i 为简单串，并且他们的字典序按照非严格单减排序，即 $w_1 \geq w_2 \geq ... \geq w_n$。可以发现，这样的分解存在且唯一。

```
/*
 Lyndon 分解
 */
#include<bits/stdc++.h>
using namespace std;
const int N = 5e6 + 5;
char s[N];
```

```cpp
int n, ans;
int main()
{
    scanf("%s", s+1);
    n = (int)strlen(s+1);
    for(int i = 1; i <= n; ) {
        int j = i, k = i + 1;
        while(k <= n && s[j] <= s[k]) {
            if(s[j] < s[k]) j = i;
            else j++;
            k++;
        }
        while(i <= j) {
            ans ^= i + k - j - 1; //右端点的位置
            i += k - j;
        }
    }
    printf("%d\n", ans);
    return 0;
}
```

4 数据结构

4.1 ST 表

```cpp
#include <bits/stdc++.h>
using namespace std;
const int N = 5e4 + 5;
int Max[N][30], Min[N][30];
class Solution {
public:
    inline int mx(int l,int r)
    {
        int k=(int)(log((double)(r-l+1))/log(2.0));
        return max(Max[l][k],Max[r-(1<<k)+1][k]);
    }
    inline int mn(int l,int r)
    {
        int k=(int)(log((double)(r-l+1))/log(2.0));
        return min(Min[l][k],Min[r-(1<<k)+1][k]);
    }

    int Intervalxor(vector<int> &num, vector<vector<int>> &ask) {
        // write your code here
        int n = num.size();
        for(int i = 1; i <= n; i++)
        {
            Max[i][0] = num[i-1];
            Min[i][0] = num[i-1];
        }
        for(int j=1;(1<<j)<=n;j++)
        {
```

```
            for(int i=1;i+(1<<j)-1<=n;i++)
            {
                Max[i][j]=max(Max[i][j-1],Max[i+(1<<(j-1))][j-1]);
                Min[i][j]=min(Min[i][j-1],Min[i+(1<<(j-1))][j-1]);
            }
        }
        int ans = 0;
        for (auto i : ask) {
            int l1 = i[0], r1 = i[1], l2 = i[2], r2 = i[3];
// cout << mx(l1, r1) << " " << mn(l2, r2) << endl;
            ans ^= mx(l1, r1) + mn(l2, r2);
        }
        return ans;
    }
};

int main() {
    while(true) {
        int n; cin >> n;
        vector<int> num(n);
        for (auto& i : num) cin >> i;
        int m; cin >> m;
        vector<vector<int>> ask(m);
        for (auto& i : ask) {
            for (int j = 0; j < 4; j++) {
                int t; cin >> t;
                i.push_back(t);
            }
        }
        cout << Solution().Intervalxor(num, ask) << endl;
    }
}
```

4.2 线段树

```
/*
    区间修改、区间查询区间和
*/
const int N = 1e5 + 5;
#define ls o<<1
#define rs o<<1|1
int sum[N<<2], lz[N<<2], n;
void pushup(int o) {
    sum[o] = sum[ls] + sum[rs];
}
void pushdown(int o, int l, int r) {
    if(lz[o] != -1) {
        int m = (l + r) >> 1;
        sum[ls] = lz[o] * (m - l + 1);
        sum[rs] = lz[o] * (r - m);
        lz[ls] = lz[rs] = lz[o];
        lz[o] = -1;
```

```
    }
}
void build(int l, int r, int o) {
    lz[o] = -1; sum[o] = 0;
    if(l == r) return ;
    int m = (l + r) >> 1;
    build(l, m, ls);
    build(m+1, r, rs);
}
void update(int L, int R, int val, int l = 1, int r = n, int o = 1) {
    if(l > R || r < L) return ;
    if(L <= l && r <= R) {
        sum[o] = val * (r - l + 1);
        lz[o] = val;
        return ;
    }
    int m = (l + r) >> 1;
    pushdown(o, l, r);
    update(L, R, val, l, m, ls);
    update(L, R, val, m+1, r, rs);
    pushup(o);
}
int query(int L, int R, int l = 1, int r = n, int o = 1) {
    if(l > R || r < L) return 0;
    if(L <= l && r <= R) {
        return sum[o];
    }
    int m = (l + r) >> 1, res = 0;
    pushdown(o, l, r);
    res += query(L, R, l, m, ls);
    res += query(L, R, m+1, r, rs);
    return res;
}

/*
    区间修改，区间查询极值
*/
const int inf = 0x3f3f3f3f;
#define ls o<<1
#define rs o<<1|1
int n, f;
int lazy[maxn<<2], mx[maxn<<2], mn[maxn<<2];
void pushdown(int o) {
    if(lazy[o]) {
        mx[ls] = mx[rs] = lazy[o];
        mn[ls] = mn[rs] = lazy[o];
        lazy[ls] = lazy[rs] = lazy[o];
        lazy[o] = 0;
    }
}
void pushup(int o) {
    mx[o] = max(mx[ls], mx[rs]);
    mn[o] = min(mn[ls], mn[rs]);
```

```
}
void update(int L, int R, int x, int l = 1, int r = n, int o = 1) {
    if(l > R || r < L) return ;
    if(L <= l && r <= R) {
        lazy[o] = x;
        mx[o] = mn[o] = x;
        return ;
    }
    pushdown(o);
    int mid = (l + r) >> 1;
    update(L, R, x, l, mid, ls);
    update(L, R, x, mid+1, r, rs);
    pushup(o);
}

int query_max(int L, int R, int l = 1, int r = n, int o = 1) {
    if(l > R || r < L) return 0;
    if(L <= l && r <= R) {
        return mx[o];
    }
    pushdown(o);
    int mid = (l + r) >> 1;
    return max(query_max(L, R, l, mid, ls), query_max(L, R, mid+1, r, rs));
}

int query_min(int L, int R, int l = 1, int r = n, int o = 1) {
    if(l > R || r < L) return inf;
    if(L <= l && r <= R) {
        return mn[o];
    }
    pushdown(o);
    int mid = (l + r) >> 1;

    return min(query_min(L, R, l, mid, ls), query_min(L, R, mid+1, r, rs));
}

/*
    单点修改、区间查询
*/
#define ls o<<1
#define rs o<<1|1
const int N = 2e5 + 5;
int tr[N<<2];
int sz;
void pushup(int o) {
    tr[o] = max(tr[ls], tr[rs]);
}

void update(int i, int x, int l = 1, int r = sz, int o = 1) {
    if(l == r) {
        tr[o] = x;
        return ;
    }
```

```
    int mid = (l + r) >> 1;
    if(i <= mid) update(i, x, l, mid, ls);
    else update(i, x, mid+1, r, rs);
    pushup(o);
}

int query(int L, int R, int l = 1, int r = sz, int o = 1) {
    if(r < L || R < l) return 0;
    if(L <= l && r <= R) {
        return tr[o];
    }
    int mid = (l + r) >> 1;
    int res = 0;
    if(L <= mid) res = max(res, query(L, R, l, mid, ls));
    if(mid < R) res = max(res, query(L, R, mid+1, r, rs));
    return res;
}
```

4.3 二维线段树

```
/*
    注意空间问题
    二维线段树求区间最值问题（可修）。
    给你一个n*n的初始矩阵，支持三种操作
    单点修改某个位置的值（或者加上一个值）
    查询子矩阵最小值和最大值，或者查询子矩阵的和
    测试题目：hdu 4819 && poj 1195
*/
#include <bits/stdc++.h>
#define ll long long
using namespace std;
const int INF = 0x3f3f3f3f;
const int N = 1024 + 5;
ll MAX[N << 2][N << 2], minV, maxV,MIN[N<<2][N<<2];//维护最值
ll a[N<<2][N<<2];//初始矩阵
ll SUM[N<<2][N<<2],sumV;//维护求和
int n;
void pushupX(int deep, int rt)
{
    MAX[deep][rt] = max(MAX[deep << 1][rt], MAX[deep << 1 | 1][rt]);
    MIN[deep][rt] = min(MIN[deep << 1][rt], MIN[deep << 1 | 1][rt]);
    SUM[deep][rt] = SUM[deep<<1][rt]+SUM[deep<<1|1][rt];
}
void pushupY(int deep, int rt)
{
    MAX[deep][rt] = max(MAX[deep][rt << 1], MAX[deep][rt << 1 | 1]);
    MIN[deep][rt] = min(MIN[deep][rt << 1], MIN[deep][rt << 1 | 1]);
    SUM[deep][rt]=SUM[deep][rt<<1]+SUM[deep][rt<<1|1];
}
void buildY(int ly, int ry, int deep, int rt, int flag)
{
    //y轴范围ly,ry;deep,rt;标记flag
```

```
    if (ly == ry){
        if (flag!=-1)
            MAX[deep][rt] = MIN[deep][rt] = SUM[deep][rt] = a[flag][ly];
        else
            pushupX(deep, rt);
        return;
    }
    int m = (ly + ry) >> 1;
    buildY(ly, m, deep, rt << 1, flag);
    buildY(m + 1, ry, deep, rt << 1 | 1, flag);
    pushupY(deep, rt);
}
void buildX(int lx, int rx, int deep)
{
    //建树x轴范围lx,rx;deep
    if (lx == rx){
        buildY(1, n, deep, 1, lx);
        return;
    }
    int m = (lx + rx) >> 1;
    buildX(lx, m, deep << 1);
    buildX(m + 1, rx, deep << 1 | 1);
    buildY(1, n, deep, 1, -1);
}
void updateY(int Y, int val, int ly, int ry, int deep, int rt, int flag)
{
    //单点更新y坐标;更新值val;当前操作y的范围ly,ry;deep,rt;标记flag
    if (ly == ry){
        if (flag) //注意读清楚题意，看是单点修改值还是单点加值
            MAX[deep][rt] = MIN[deep][rt] = SUM[deep][rt] = val;
        else pushupX(deep, rt);
        return;
    }
    int m = (ly + ry) >> 1;
    if (Y <= m)
        updateY(Y, val, ly, m, deep, rt << 1, flag);
    else
        updateY(Y, val, m + 1, ry, deep, rt << 1 | 1, flag);
    pushupY(deep, rt);
}
void updateX(int X, int Y, int val, int lx, int rx, int deep)
{
    //单点更新范围x,y;更新值val;当前操作x的范围lx,rx;deep
    if (lx == rx){
        updateY(Y, val, 1, n, deep, 1, 1);
        return;
    }
    int m = (lx + rx) >> 1;
    if (X <= m)
        updateX(X, Y, val, lx, m, deep << 1);
    else
        updateX(X, Y, val, m + 1, rx, deep << 1 | 1);
    updateY(Y, val, 1, n, deep, 1, 0);
```

```
}
void queryY(int Yl, int Yr, int ly, int ry, int deep, int rt)
{
    //询问区间y轴范围y1,y2;当前操作y的范围ly,ry;deep,rt
    if (Yl <= ly && ry <= Yr)
    {
        minV = min(MIN[deep][rt], minV);
        maxV = max(MAX[deep][rt], maxV);
        sumV += SUM[deep][rt];
        return;
    }
    int m = (ly + ry) >> 1;
    if (Yl <= m)
        queryY(Yl, Yr, ly, m, deep, rt << 1);
    if (m < Yr)
        queryY(Yl, Yr, m + 1, ry, deep, rt << 1 | 1);
}
void queryX(int Xl, int Xr, int Yl, int Yr, int lx, int rx, int rt)
{
    //询问区间范围x1,x2,y1,y2;当前操作x的范围lx,rx;rt
    if (Xl <= lx && rx <= Xr){
        queryY(Yl, Yr, 1, n, rt, 1);
        return;
    }
    int m = (lx + rx) >> 1;
    if (Xl <= m)
        queryX(Xl, Xr, Yl, Yr, lx, m, rt << 1);
    if (m < Xr)
        queryX(Xl, Xr, Yl, Yr, m + 1, rx, rt << 1 | 1);
}
int main()
{
    scanf("%d",&n);
    for(int i=1;i<=n;i++)
        for(int j=1;j<=n;j++)
            scanf("%lld",&a[i][j]);
    buildX(1,n,1);
    int m;
    scanf("%d",&m);
    while(m--){
        int opt;
        scanf("%d",&opt);
        if(opt==1){ //单点修改
            int x,y,val;
            scanf("%d%d%d",&x,&y,&val);
            updateX(x,y,val,1,n,1);
        }
        else if(opt==2){//查询子矩阵最值，以及和
            minV=INF,maxV=-INF,sumV=0;//注意初始化
            int x1,y1,x2,y2;//注意看清楚给范围的方式以及下标是从0开始还是从1开始
            scanf("%d%d%d%d",&x1,&y1,&x2,&y2);
            queryX(x1,x2,y1,y2,1,n,1);
            printf("%lld %lld %lld\n",sumV,minV,maxV);
```

```
        }

    }
    return 0;
}
```

4.4 主席树

```
/*
前缀和思想、公共节点、权值平衡树
*/
typedef long long ll ;
const int oo=0x7f7f7f7f ;
const int maxn=1e5+7;
const int mod=1e9+7;
int n,m,cnt,root[maxn],a[maxn],x,y,k;
struct node{
    int l,r,sum;
}T[maxn*25];
vector<int> v;
int getid(int x){
    return lower_bound(v.begin(),v.end(),x)-v.begin()+1;
}
void update(int l,int r,int &x,int y,int pos){
    T[++cnt]=T[y],T[cnt].sum++,x=cnt;
    if(l==r) return;
    int mid=(l+r)/2;
    if(mid>=pos) update(l,mid,T[x].l,T[y].l,pos);
    else update(mid+1,r,T[x].r,T[y].r,pos);
}
int query(int l,int r,int x,int y,int k){
    if(l==r) return l;
    int mid=(l+r)/2;
    int sum=T[T[y].l].sum-T[T[x].l].sum;
    if(sum>=k) return query(l,mid,T[x].l,T[y].l,k);
    else return query(mid+1,r,T[x].r,T[y].r,k-sum);
}
int main(void){
    scanf("%d%d",&n,&m);
    for(int i=1;i<=n;i++) scanf("%d",&a[i]),v.push_back(a[i]);
    sort(v.begin(),v.end());
    v.erase(unique(v.begin(),v.end()),v.end());

    for(int i=1;i<=n;i++) update(1,n,root[i],root[i-1],getid(a[i]));
    for(int i=1;i<=m;i++){
        scanf("%d%d%d",&x,&y,&k);
        printf("%d\n",v[query(1,n,root[x-1],root[y],k)-1]);
    }

    return 0;
}
```

4.5 主席树

```
/*
前缀和思想、公共节点、权值平衡树
*/
typedef long long ll ;
const int oo=0x7f7f7f7f ;
const int maxn=1e5+7;
const int mod=1e9+7;
int n,m,cnt,root[maxn],a[maxn],x,y,k;
struct node{
    int l,r,sum;
}T[maxn*25];
vector<int> v;
int getid(int x){
    return lower_bound(v.begin(),v.end(),x)-v.begin()+1;
}
void update(int l,int r,int &x,int y,int pos){
    T[++cnt]=T[y],T[cnt].sum++,x=cnt;
    if(l==r) return;
    int mid=(l+r)/2;
    if(mid>=pos) update(l,mid,T[x].l,T[y].l,pos);
    else update(mid+1,r,T[x].r,T[y].r,pos);
}
int query(int l,int r,int x,int y,int k){
    if(l==r) return l;
    int mid=(l+r)/2;
    int sum=T[T[y].l].sum-T[T[x].l].sum;
    if(sum>=k) return query(l,mid,T[x].l,T[y].l,k);
    else return query(mid+1,r,T[x].r,T[y].r,k-sum);
}
int main(void){
    scanf("%d%d",&n,&m);
    for(int i=1;i<=n;i++) scanf("%d",&a[i]),v.push_back(a[i]);
    sort(v.begin(),v.end());
    v.erase(unique(v.begin(),v.end()),v.end());

    for(int i=1;i<=n;i++) update(1,n,root[i],root[i-1],getid(a[i]));
    for(int i=1;i<=m;i++){
        scanf("%d%d%d",&x,&y,&k);
        printf("%d\n",v[query(1,n,root[x-1],root[y],k)-1]);
    }

    return 0;
}
```

4.6 线性基

```
/*
线性基是一个数的集合，并且每个序列都拥有至少一个线性基，取线性基中若干个数异或起来可以得到原
    序列中的任何一个数。
 功能:
```

```
1. 查询异或最值
2. 查询k小值
3. 查询某个数
*/

struct XOR {
    const int MN=62;
    ll a[63], tmp[63];
    bool flag; //可以表示0
    void ins(ll x){
        for(int i = MN; ~i; i--)
            if(x&(1ll<<i)) {
                if(!a[i]) { a[i]=x; return;}
                else x ^= a[i];
            }
        flag=true;
    }
    bool check(ll x){ // 查询某个数
        for(int i = MN; ~i; i--)
            if(x&(1ll<<i)) {
                if(!a[i]) return false;
                else x ^= a[i];
            }
        return true;
    }
    ll qmax(ll res = 0){ // 查询异或最大值
        for(int i = MN; ~i; i--)
            res = max(res, res^a[i]);
        return res;
    }
    ll qmin(){ // 查询异或最小值
        if(flag) return 0;
        for(int i = 0; i <= MN; i++) if(a[i]) return a[i];
        return -1; // not exists
    }
    ll query(ll k){ // 查询k小值
        ll res = 0; int cnt = 0;
        k -= flag;if(!k) return 0;
        for(int i = 0; i <= MN; i++){
            for(int j = i - 1; ~j; j--)
                if(a[i]&(1ll<<j)) a[i] ^= a[j];
            if(a[i]) tmp[cnt++] = a[i];
        }
        if(k >= (1ll<<cnt)) return -1;
        for(int i = 0; i < cnt; i++)
            if(k&(1ll<<i)) res ^= tmp[i];
        return res;
    }
}ret;
```

4.7 舞蹈链

```
#include <bits/stdc++.h>
using namespace std;
typedef pair<int,int> pii;
typedef long long ll;

const int N = 9;
const int MaxN = N * N * N + 10;
const int MaxM = N * N * 4 + 10;
const int maxnode = MaxN * 4 + MaxM + 10;
char g[MaxN];
struct DLX {
    int n, m, size;
    int U[maxnode], D[maxnode], R[maxnode], L[maxnode], Row[maxnode], Col[maxnode];
    // up, down, right, left, 行号, 列号
    int H[MaxN], S[MaxM]; // S是列中1的个数
    int ansd, ans[MaxN]; // 辅助数组
    void init(int _n, int _m) { // n是列数, m是行数
        n = _n;
        m = _m;
        for(int i = 0; i <= m; ++i) { // 初始化列标
            S[i] = 0; // ? ?
            U[i] = D[i] = i; // ? ?
            L[i] = i - 1;
            R[i] = i + 1;
        }
        R[m] = 0;
        L[0] = m; // 循环双向链表
        size = m;
        for(int i = 1; i <= n; ++i)
            H[i] = -1; // 是否是第0列, 及第i列是否有首个元素
    }

    void Link(int r, int c) {
        ++S[Col[++size] = c];
        Row[size] = r;
        D[size] = D[c];
        U[D[c]] = size;
        U[size] = c;
        D[c] = size;
        if(H[r] < 0)
            H[r] = L[size] = R[size] = size;
        else {
            R[size] = R[H[r]];
            L[R[H[r]]] = size;
            L[size] = H[r];
            R[H[r]] = size;
        }
    }

    void remove(int c) { // 删除列标c上有1的行的所有元素
        L[R[c]] = L[c];
        R[L[c]] = R[c];
        for(int i = D[c]; i != c; i = D[i])
```

```
            for(int j = R[i]; j != i; j = R[j]) {
                U[D[j]] = U[j];
                D[U[j]] = D[j];
                --S[Col[j]];
            }
    }

    void resume(int c) { // 恢复列标c上有1的行的所有元素
        for(int i = U[c]; i != c; i = U[i])
            for(int j = L[i]; j != i; j = L[j])
                ++S[Col[U[D[j]] = D[U[j]] = j]];
        L[R[c]] = R[L[c]] = c;
    }

    bool Dance(int d) { // d是深度
        if(R[0] == 0) { // 如果head.right = head
            for(int i = 0; i < d; ++i)
                g[(ans[i] - 1) / 9] = (ans[i] - 1) % 9 + '1';
            for(int i = 0; i < N * N; ++i)
                printf("%c", g[i]);
            printf("\n");
            return true;
        }
        int c = R[0]; // c = head.right
        for(int i = R[0]; i != 0; i = R[i]) {
            if(S[i] < S[c]) c = i; // 最优化，列中1的个数最多
        }
        remove(c); // 删除列标c
        for(int i = D[c]; i != c; i = D[i]) { // 循环列标c中所有的行号
            ans[d] = Row[i]; // 记录答案
            for(int j = R[i]; j != i; j = R[j])
                remove(Col[j]); // 删除列标
            if(Dance(d + 1)) return true;
            for(int j = L[i]; j != i; j = L[j])
                resume(Col[j]); // 恢复列标
        }
        resume(c); // 恢复列标
        return false;
    }
};

void place(int &r, int &c1, int &c2, int &c3, int &c4, int i, int j, int k) {
    r = (i * N + j) * N + k;
    c1 = i * N + j + 1;
    c2 = N * N + i * N + k;
    c3 = N * N * 2 + j * N + k;
    c4 = N * N * 3 + ((i / 3) * 3 + (j / 3)) * N + k;
}

DLX dlx;

int main() {
    while(~scanf("%s", &g)) {
```

```
        if(g[0] == 'e')
            exit(0);
        dlx.init(N * N * N, N * N * 4);
        int r, c1, c2, c3, c4;
        for(int i = 0; i < N; ++i) {
            for(int j = 0; j < N; ++j) {
                for(int k = 1; k <= N; ++k) {
                    if(g[i * N + j] == '.' || g[i * N + j] == '0' + k) {
                        place(r, c1, c2, c3, c4, i, j, k);
                        dlx.Link(r, c1);
                        dlx.Link(r, c2);
                        dlx.Link(r, c3);
                        dlx.Link(r, c4);
                    }

                }
            }
        }
        dlx.Dance(0);
    }
    return 0;
}
```

4.8 划分树

```
/*
划分树是一种来解决区间第k大的一种数据结构，其常数、理解难度都要比主席树低很多。
Tips:非递归版本，常数较小。
*/
const int MAXN = 2e5 + 5;
const int LOG_N = 30;
// tree[dep][i] 第dep层第i个位置的数值
// toleft[p][i] 第p层前i个数中有多少个整数分入下一层
int tree[LOG_N][MAXN], sorted[MAXN], toleft[LOG_N][MAXN];
void build(int l, int r, int dep) {
    if(l == r) return;
    int mid = (l + r) / 2;
    int same = mid - l + 1; // 和中点数相同的数的个数
    for(int i = l; i <= r; i++)
        if(tree[dep][i] < sorted[mid]) same--;
    int lpos = l, rpos = mid + 1;
    for(int i = l; i <= r; i++) {
        if(tree[dep][i] < sorted[mid])
            tree[dep + 1][lpos++] = tree[dep][i];
        else if(tree[dep][i] == sorted[mid] && same) {
            tree[dep + 1][lpos++] = tree[dep][i];
            same--;
        }
        else tree[dep + 1][rpos++] = tree[dep][i];
        toleft[dep][i] = toleft[dep][l - 1] + lpos - l;
    }
    build(l, mid, dep + 1);
```

```
    build(mid + 1, r, dep + 1);
}
// [L,R]里查询子区间[l,r]第k小的数
int query(int L, int R, int l, int r, int dep, int k)
{
    if(l == r) return tree[dep][l];
    int mid = (L + R) / 2;
    // 有多少个查询区间内的节点会进入下一层的左子树
    int cnt = toleft[dep][r] - toleft[dep][l - 1];
    if(cnt >= k) {
        int newl = L + toleft[dep][l - 1] - toleft[dep][L - 1];
        int newr = newl + cnt - 1;
        return query(L, mid, newl, newr, dep + 1, k);
    }
    else {
        int newr = r + toleft[dep][R] - toleft[dep][r];
        int newl = newr - (r - l - cnt);
        return query(mid + 1, R, newl, newr, dep + 1, k - cnt);
    }
}
int main() {
    int n, m;
    while(~scanf("%d%d", &n, &m)) {
        for(int i = 1; i <= n; i++) {
            scanf("%d", &sorted[i]);
            tree[0][i] = sorted[i];
        }
        sort(sorted + 1, sorted + n + 1);
        build(1, n, 0);
        while(m--) {
            int l, r, k;
            scanf("%d%d%d", &l, &r, &k);
            printf("%d\n", query(1, n, l, r, 0, k));
        }
    }
    return 0;
}
```

4.9 单调栈

单调栈的应用

单调栈只能维护栈内元素，以某一特征递增或递减，而其应用无法离开这条性质。

举个简单的例子，存在长度为 n 的数组 $a[i]$，现要求对 $i \in [1,n]$，输出 $\sum_{j=1}^{i-1} Min(a[k], k \in [j,i])$。

朴素的想法是暴力向前扫一遍，维护区间最小值，这样的复杂度是 $O(n^2)$ 的。

依靠单调栈优化，实现方法是维护单调栈自顶向下递减，对栈内每个元素统计其 val 和 cnt，val 意为对答案的贡献，cnt 为贡为 val 的下标个数，栈内所有元素 $\sum cnt * val$ 即为答案。

4.10 并查集

4.10.1 带权并查集

```
/*
1. 路径压缩实现，维护集合内元素之间可量化的关系，如种类并查集（洛谷-食物链）
*/
struct DSU {
    int f[maxn], val[maxn];
    inline void Init(int n) {
        for (int i = 0; i <= n; i++) {
            f[i] = i;
            val[i] = 0;
        }
    }

    inline int Find(int x) {
        if(f[x] == x) return f[x];
        int fx = Find(f[x]); //这里不能够没有，因为在递归的过程中f[x]的值会被改变，会影响val[x]的更新
        val[x] += val[f[x]];
        return f[x] = fx;
    }

    inline bool Union(int a, int b, int v) {
        int fa = Find(a), fb = Find(b);
        if(fa != fb) {
            if(fa > fb) {
                f[fb] = fa;
                val[fb] = -val[b] - v + val[a];
            }
            else {
                f[fa] = fb;
                val[fa] = -val[a] + v + val[b];
            }
            return true;
        }
        else {
            if(val[fa] != -val[a] + v + val[b]) return false; // 不合法
            /*
                if(val[fa] != (-val[a] + v + val[b] + 2) % 2) return false;
                种类并查集可由带权并查集扩展而来，常见的用途有判二分图（其实是判奇环）
            */
            else return true;
        }
    }
}T;
```

4.10.2 可撤销并查集

```
/*
    按秩合并实现，不能路径压缩
*/
```

```
struct UFS {
    stack<pair<int*, int>> stk;
    int fa[N], rnk[N];
    inline void init(int n) {
        for (int i = 0; i <= n; ++i) fa[i] = i, rnk[i] = 0;
    }
    inline int Find(int x) {
        while(x^fa[x]) x = fa[x];
        return x;
    }
    inline void Merge(int x, int y) {
        x = Find(x), y = Find(y);
        if(x == y) return ;
        if(rnk[x] <= rnk[y]) {
            stk.push({fa+x, fa[x]});
            fa[x] = y;
            if(rnk[x] == rnk[y]) {
                stk.push({rnk+y, rnk[y]});
                rnk[y]++;
            }
        }
        else {
            stk.push({fa+y, fa[y]});
            fa[y] = x;
        }
    }
    inline void Undo() {
        *stk.top().fi = stk.top().se;
        stk.pop();
    }
}ufs;

/*
    按大小合并，复杂度无差异
*/
namespace UFS {
    int fa[maxn], sz[maxn], top;
    struct STK {
        int x, y;
    } stk[maxn*2];
    void init(int n) {
        for (int i = 1; i <= n; i++) fa[i] = i, sz[i] = 1;
        top = 0;
    }
    int find(int x) {
        return fa[x] == x ? x : find(fa[x]);
    }
    void merge(int x, int y) {
        int fx = find(x), fy = find(y);
        if(fx == fy) return ;
        if(sz[fx] > sz[fy]) swap(fx, fy);
        sz[fy] += sz[fx];
        stk[++top] = {fx, fy};
```

```
        fa[fx] = fy;
    }
    void back() {
        sz[stk[top].y] -= sz[stk[top].x];
        fa[stk[top].x] = stk[top].x;
        top--;
    }
}
```

4.10.3 可撤销种类并查集

```
/*
CF1444C - 可撤销种类并查集
满足“敌人的敌人是朋友”这个条件时可用，或是相邻节点都是不同种类（总共两个种类）
1. 种类并查集初始化两倍空间，Init(2*n)
2. 只能合并两个敌对关系的节点，Merge(x,y+n) Merge(y,x+n)
3. 判断两个节点的关系，Find(x)==Find(y)表示x和y是朋友，Find(x)==Find(y+n)表示x和y是敌人
4. Undo撤销并不是一次操作，而是每一次数值更改都会入栈
*/

struct UFS {
    stack<pair<int*, int>> stk;
    int fa[maxn*2], rnk[maxn*2];
    inline void Init(int n) {
        for (int i = 0; i <= n; ++i) fa[i] = i, rnk[i] = 0;
    }
    inline int Find(int x) {
        while(x^fa[x]) x = fa[x];
        return x;
    }
    inline void Merge(int x, int y) {
        x = Find(x), y = Find(y);
        if(x == y) return ;
        if(rnk[x] <= rnk[y]) {
            stk.push({fa+x, fa[x]});
            fa[x] = y;
            if(rnk[x] == rnk[y]) {
                stk.push({rnk+y, rnk[y]});
                rnk[y]++;
            }
        }
        else {
            stk.push({fa+y, fa[y]});
            fa[y] = x;
        }
    }

    inline void Undo() {
        *stk.top().first = stk.top().second;
        stk.pop();
    }
}T;
```

5 计算几何

5.1 判两条线段相交

```
struct pt { // 点
    ll x,y;
}p[10];
struct ln{ // 线段
    pt s,e;
};

ll mul(pt a,pt b,pt c) { // 向量叉积
    return (b.x-a.x)*(c.y-a.y)-(c.x-a.x)*(b.y-a.y);
}

int is(ln a,ln b) { // 判断线段是否相交
    if( min(a.s.x,a.e.x) <= max(b.s.x,b.e.x) &&
        max(a.s.x,a.e.x) >= min(b.s.x,b.e.x) &&
        min(a.s.y,a.e.y) <= max(b.s.y,b.e.y) &&
        max(a.s.y,a.e.y) >= min(b.s.y,b.e.y) &&
        mul(b.s,a.s,b.e)*mul(b.s,b.e,a.e) >= 0 &&
        mul(a.s,b.s,a.e)*mul(a.s,a.e,b.e) >= 0 )
    return 1;
    else return 0;
}
```

6 数论

6.1 埃式筛法

```
#include <bits/stdc++.h>
using namespace std;
//埃式筛法
//求n以内的质数的个数
const int maxn = 1e8 + 5;
int p;
int prime[maxn/10];
bool is_prime[maxn];

int GetPrime(int n)
{
    p = 0;
    for (int i = 0; i <= n; ++i) is_prime[i] = true;
    is_prime[0] = is_prime[1] = false;
    for (int i = 2; i*i <= n; ++i)
        if (is_prime[i])
            for (int j = i*i; j <= n; j += i)
                is_prime[j] = false;
    for (int i = 0; i <= n; ++i)
        if (is_prime[i])
            prime[p++] = i;
```

```
    return p;
}



int main() {
    int n, q;
    scanf("%d%d", &n, &q);
    GetPrime(n);
    while (q--) {
        int k;
        scanf("%d", &k);
        printf("%d\n", prime[--k]);
    }
}
```

6.2 分数类

```
namespace Fraction{
    template<typename Int>Int gcd(Int a,Int b){return b?gcd(b,a%b):a;}
    template<typename Int>Int abs(Int a){return a<0?-a:a;}
    template<typename Int>struct Frac{
        Int a,b;
        Frac(){a=(Int)0;b=(Int)1;}
        Frac(Int x){a=x;b=(Int)1;}
        Frac(Int x,Int y){a=x;b=y;}
        Frac &operator =(Int x){a=x;b=1;return *this;}
        double to_double(){return (double)a/b;}
        Frac rec(){assert(a);return Frac(b,a);}
        Frac operator-(){return (Frac){-a,b};}
        Frac&operator++(){a+=b;return*this;}
        Frac&operator--(){a-=b;return*this;}
        Frac&operator+=(Frac x){
            Int g=gcd(b,x.b);
            a=b/g*x.a+x.b/g*a; b*=x.b/g;
            g=gcd(abs(a),b);a/=g;b/=g;
            return*this;
        }
        Frac&operator-=(Frac x){return*this+=-x;}
        Frac&operator*=(Frac x){
            Int g1=gcd(abs(a),x.b),g2=gcd(abs(x.a),b);
            (a/=g1)*=x.a/g2;(b/=g2)*=x.b/g1;
            return*this;
        }
        Frac&operator/=(Frac x){return*this*=x.rec();}
        Frac friend operator +(Frac x,Frac y){return x+=y;}
        Frac friend operator -(Frac x,Frac y){return x-=y;}
        Frac friend operator *(Frac x,Frac y){return x*=y;}
        Frac friend operator /(Frac x,Frac y){return x/=y;}
        int operator !(){return a;}
        int friend operator &&(Frac x,Frac y){return x.a&&y.a;}
        int friend operator ||(Frac x,Frac y){return x.a||y.a;}
```

```
#define logical_operator(op) int friend operator op(Frac x,Frac y){return (x-y).a op 0;}

        logical_operator(<)
        logical_operator(>)
        logical_operator(<=)
        logical_operator(>=)
        logical_operator(==)
        /*
        friend ostream&operator<<(ostream&ostr,Frac x){
            ostr<<x.a;
            assert(x.b);
            if(x.b!=1)ostr<<"/"<<x.b;
            return ostr;
        }
        *///输出一个分数，一般用不上，要用取消注释并使用std::cout即可
    };
#undef logical_operator
}

Frac<int>x;//定义一个初值为0的分数
Frac<int>x(a);//定义一个初值为a的分数
Frac<int>x(a,b);//定义一个初值为a/b的分数
```

6.3 三分

```
/*
凸函数求极大值 int
*/
while(l+1<r)
{
    int lm=(l+r)>>1,rm=(lm+r)>>1;
    if(judge(lm)>judge(rm))
        r=rm;
    else
        l=lm;
}
//答案取 l

/*
凸函数求极大值 double
*/
while(l+eps<r)
{
    double lm=(l+r)/2,rm=(lm+r)/2;
    if(judge(lm)>judge(rm))
        r=rm;
    else
        l=lm;
}
//答案取 l 或 (l+r)/2 (尽管此时 l 和 r 已经相等，但因为精度问题，取 r 可能会错)

/*
```

```
凹函数求极小值: int
*/
while(l+1<r)
{
    int lm=(l+r)>>1,rm=(lm+r)>>1;
    if(judge(lm)>judge(rm))
        l=lm;
    else
        r=rm;
}
//答案取 r

/*
凹函数求极小值 double
*/
while(l+eps<r)
{
    double lm=(l+r)/2,rm=(lm+r)/2;
    if(judge(lm)>judge(rm))
        l=lm;
    else
        r=rm;
}
//答案取 r 或 (l+r)/2 (尽管此时 l 和 r 已经相等, 但因为精度问题, 取 l 可能会错)
```

6.4 组合数

```
/*
    适用于N<=3000【打表】
    c[i][j]表示从i个中选j个的选法。
*/

long long C[N][N];

void get_C(int maxn)
{
    C[0][0] = 1;
    for(int i=1;i<=maxn;i++)
    {
        C[i][0] = 1;
        for(int j=1;j<=i;j++)
            C[i][j] = C[i-1][j]+C[i-1][j-1];
            //C[i][j] = (C[i-1][j]+C[i-1][j-1])%MOD;
    }
}

/*
    n<=200000【求值】
*/

ll fac[maxn],inv[maxn];
ll pow_mod(ll a,ll n)
```

```
{
    ll ret =1;
    while(n)
    {
        if(n&1) ret=ret*a%mod;
          a=a*a%mod;
          n>>=1;
    }
    return ret;
}
void init()
{
    fac[0]=1;
    for(int i=1;i<maxn;i++)
    {
        fac[i]=fac[i-1]*i%mod;
    }
}
ll Cc(ll x, ll y)
{
    return fac[x]*pow_mod(fac[y]*fac[x-y]%mod,mod-2)%mod;
}

int main(){
    ll n,m;
    init();
    while(1){
        cin>>n>>m;
        cout<<Cc(n,m)<<endl;
    }
}
```

6.5 exgcd

```
/*
* 模版：二元一次不定方程 (ax + by = c)
* 功能：
* 1. 有无整数解
* 2. 正整数解的的个数
*/

#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
void print(ll x) {
    if (x>9) print(x/10);
    putchar('0'+x%10);
}
ll exgcd(ll a, ll b, ll &x, ll &y) {
    if(!b) {
        x = 1, y = 0;
        return a;
```

```
    }
    ll d = exgcd(b, a%b, x, y);
    ll t = x; x = y; y = t - a / b * y;
    return d;
}
void calc(ll a, ll b, ll c) {
    ll x0, y0;
    ll d = exgcd(a, b, x0, y0);
    if(c % d) { // 无整数解
        printf("0\n");
        return ;
    }
    ll x1 = x0 * (c / d), y1 = y0 * (c / d), k;
    ll dx = b / d, dy = a / d;
    if(x1 < 0) { // 将x提高到最小正整数
        k = (1-x1)/dx + ((1-x1)%dx>0);
        x1 += k * dx; y1 -= k * dy;
    }
    else { // 将x降低到最小正整数
         k = (x1-1)/dx;
         x1 -= k * dx; y1 += k * dy;
    }
    if(y1 > 0) { // 有正整数解
        print((y1-1)/dy+1);
        puts("");
    }
    else { // 无正整数解
        printf("0\n");
    }
}

int main() {
    int n;
    scanf("%d", &n);
    while(n--) {
        int b, d;
        scanf("%d%d", &b, &d);
        calc(d, b, 1ll*b*b);
    }
}
```

6.6 高斯消元

```
/*
* 模版: 高斯消元法解方程组(Gauss-Jordan elimination).
* 返回值:
* -2表示有浮点数解, 但无整数解
* -1表示无解
* 0表示唯一解
* 大于0表示无穷解, 并返回自由变元的个数
* 初始化: 有equ个方程, var个变元。增广矩阵行数为equ,分别为0到equ-1,列数为var+1,分别为0到var
    .
```

```
* 注意: 参数不能重复使用, 每次都要重新赋值
*/
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 10;
int a[MAXN][MAXN];//增广矩阵
int x[MAXN];//解集
bool free_x[MAXN];//标记是否是不确定的变元
int gcd(int a,int b) {
    return b ? gcd(b, a%b) : a;
}
int lcm(int a,int b) {
    return a/gcd(a,b)*b;//先除后乘防溢出
}

int Gauss(int equ, int var) {
    int i,j,k;
    int max_r;// 当前这列绝对值最大的行.
    int col;//当前处理的列
    int ta,tb;
    int LCM;
    int temp;
    int free_x_num;
    int free_index;
    for(int i=0;i<=var;i++) {
        x[i]=0;
        free_x[i]=true;
    }
    //转换为阶梯阵.
    col=0; // 当前处理的列
    for(k = 0;k < equ && col < var;k++,col++) { // 枚举当前处理的行.
    // 找到该col列元素绝对值最大的那行与第k行交换.(为了在除法时减小误差)
        max_r=k;
        for(i=k+1;i<equ;i++) {
            if(abs(a[i][col])>abs(a[max_r][col])) max_r=i;
        }
        if(max_r!=k) {// 与第k行交换
            for(j=k;j<var+1;j++) swap(a[k][j],a[max_r][j]);
        }
        if(a[k][col]==0) {// 说明该col列第k行以下全是0了, 则处理当前行的下一列.
            k--;
            continue;
        }
        for(i=k+1;i<equ;i++) {// 枚举要删去的行.
            if(a[i][col]!=0)
            {
                LCM = lcm(abs(a[i][col]),abs(a[k][col]));
                ta = LCM/abs(a[i][col]);
                tb = LCM/abs(a[k][col]);
                if(a[i][col]*a[k][col]<0)tb=-tb;//异号的情况是相加
                for(j=col;j<var+1;j++)
                {
                    a[i][j] = a[i][j]*ta-a[k][j]*tb;
```

```
                }
            }
        }
    }
    // 1. 无解的情况: 化简的增广阵中存在(0, 0, ..., a)这样的行(a != 0).
    for (i = k; i < equ; i++)
    { // 对于无穷解来说，如果要判断哪些是自由变元，那么初等行变换中的交换就会影响，则要记录交换.
        if (a[i][col] != 0) return -1;
    }
    // 2. 无穷解的情况: 在var * (var + 1)的增广阵中出现(0, 0, ..., 0)这样的行，即说明没有形成严格的上三角阵.
    // 且出现的行数即为自由变元的个数.
    if (k < var)
    {
        // 首先，自由变元有var - k个，即不确定的变元至少有var - k个.
        for (i = k - 1; i >= 0; i--)
        {
            // 第i行一定不会是(0, 0, ..., 0)的情况，因为这样的行是在第k行到第equ行.
            // 同样，第i行一定不会是(0, 0, ..., a), a != 0的情况，这样的无解的.
            free_x_num = 0; // 用于判断该行中的不确定的变元的个数，如果超过1个，则无法求解，它们仍然为不确定的变元.
            for (j = 0; j < var; j++)
            {
                if (a[i][j] != 0 && free_x[j]) free_x_num++, free_index = j;
            }
            if (free_x_num > 1) continue; // 无法求解出确定的变元.
            // 说明就只有一个不确定的变元free_index，那么可以求解出该变元，且该变元是确定的.
            temp = a[i][var];
            for (j = 0; j < var; j++)
            {
                if (a[i][j] != 0 && j != free_index) temp -= a[i][j] * x[j];
            }
            x[free_index] = temp / a[i][free_index]; // 求出该变元.
            free_x[free_index] = 0; // 该变元是确定的.
        }
        return var - k; // 自由变元有var - k个.
    }
    // 3. 唯一解的情况: 在var * (var + 1)的增广阵中形成严格的上三角阵.
    // 计算出Xn-1, Xn-2 ... X0.
    for (i = var - 1; i >= 0; i--)
    {
        temp = a[i][var];
        for (j = i + 1; j < var; j++)
        {
            if (a[i][j] != 0) temp -= a[i][j] * x[j];
        }
        if (temp % a[i][i] != 0) return -2; // 说明有浮点数解，但无整数解.
        x[i] = temp / a[i][i];
    }
    return 0;
}
```

```cpp
/*
* 模版：解对P取模的方程组
*/
#include <bits/stdc++.h>
using namespace std;
const int N = 105;
typedef long long ll;
int equ, var;
int a[N][N];
int b[N][N];
int x[N];
int free_x[N];
int free_num;
int n;
int gcd(int a, int b) {
    return b ? gcd(b, a%b) : a;
}
int lcm(int a, int b) {
    return a / gcd(a, b) * b;
}
int Gauss(int P) {
    int max_r, col, k;
    free_num = 0;
    for (k = 0, col = 0; k < equ && col < var; k++, col++) {
        max_r = k;
        for (int i = k + 1; i < equ; i++) {
            if(abs(a[i][col]) > abs(a[max_r][col])) max_r = i;
        }
        if(a[max_r][col] == 0) {
            k--;
            free_x[free_num++] = 0;
            continue;
        }
        if(max_r != k) {
            for (int j = col; j < var+1; j++) {
                swap(a[k][j], a[max_r][j]);
            }
        }
        for (int i = k+1; i < equ; i++) {
            if(a[i][col] != 0) {
                int LCM = lcm(abs(a[i][col]), abs(a[k][col]));
                int ta = LCM / abs(a[i][col]);
                int tb = LCM / abs(a[k][col]);
                if(a[i][col] * a[k][col] < 0) tb = -tb;
                for (int j = col; j < var+1; j++) {
                    a[i][j] = (a[i][j]*ta-a[k][j]*tb)%P;
                    a[i][j] = (a[i][j] + P) % P;
                }
            }
        }
    }
    for (int i = k; i < equ; i++) {
        if(a[i][col] != 0) return -1;
```

```
    }
    if(k < var) return var-k;
    for (int i = var-1; i >= 0; i--) {
        ll temp = a[i][var];
        for (int j = i+1; j < var; j++) {
            temp = (temp - a[i][j] * x[j]) % P;
            temp = (temp + P) % P;
        }
        while(temp % a[i][i]) temp += P;
        temp /= a[i][i];
        temp %= P;
        x[i] = temp;
    }
    return 0;
}
void init() {
    memset(a, 0, sizeof a);
    memset(x, 0, sizeof x);
    equ = n;
    var = n;
}
void debug() {
    for(int i = 0; i < n; i++)
    {
        for(int j = 0; j <= n; j++)
            printf("%d ",a[i][j]);
        printf("\n");
    }
}

/*
* 高斯消元实数版本
*/
const double EPS = 1e-8;
typedef vector<double> vec;
typedef vector<vec> mat;
//解 Ax = b
//无解或无穷多解时返回一个长度为0的数组
vec gauss_jordan(const mat& A, const vec& b) {
    int n = A.size();
    mat B(n, vec(n + 1));
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            B[i][j] = A[i][j];
    for (int i = 0; i < n; i++) B[i][n] = b[i];
    for (int i = 0; i < n; i++) {
        int pivot = i;
        for (int j = i; j < n; j++) {
            if (abs(B[j][i]) > abs(B[pivot][i])) pivot = j;
        }
        swap(B[i], B[pivot]);
        if (abs(B[i][i]) < EPS) return vec();//无解或无穷多解
        for (int j = i + 1; j <= n; j++) B[i][j] /= B[i][i];
```

```
        for (int j = 0; j < n; j++) {
            if (i != j) {
                if (!B[j][i]) continue;
                for (int k = i + 1; k <= n; k++)
                    B[j][k] -= B[j][i] * B[i][k];
            }
        }
    }
    vec x(n);
    for (int i = 0; i < n; i++) x[i] = B[i][n];
    return x;
}
```

6.7 计数

分配

- n 个元素分配到 m 个固定大小的容器里的方案数（$cnt[i]$ 为容器 i 的大小）：$\frac{n!}{\prod_{i=1}^{m} cnt[i]!}$

树的种类数

- Cayley 公式：一个完全图 K_n 有 n^{n-2} 棵生成树，n 个节点的带标号的无根树有 n^{n-2} 个
- n 个结点构成的二叉树种类个数:$b_n = \frac{C_{2n}^{n}}{n+1}$

6.8 数列

6.8.1 oeis

```
/*
binomial(2*n,n) = (2*n)!/(n!)^2.
*/
1, 2, 6, 20, 70, 252, 924, 3432, 12870, 48620, 184756, 705432, 2704156, 10400600,
    40116600, 155117520, 601080390, 2333606220, 9075135300, 35345263800, 137846528820,
    538257874440, 2104098963720, 8233430727600, 32247603683100, 126410606437752,
    495918532948104, 1946939425648112

/*
Bell or exponential numbers: number of ways to partition a set of n labeled elements.
*/

1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, 678570, 4213597, 27644437, 190899322,
     1382958545, 10480142147, 82864869804, 682076806159, 5832742205057, 51724158235372,
     474869816156751, 4506715738447323, 44152005855084346, 445958869294805289,
    4638590332229999353, 49631246523618756274

/*
Catalan numbers: C(n) = binomial(2n,n)/(n+1) = (2n)!/(n!(n+1)!).
*/
```

```
1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452, 18367353072152, 69533550916004, 263747951750360, 1002242216651368, 3814986502092304

/*
Large Schröder numbers (or large Schroeder numbers, or big Schroeder numbers).
超级卡特兰数和施罗德数除了f(1)以外都是2倍的关系
*/
1, 2, 6, 22, 90, 394, 1806, 8558, 41586, 206098, 1037718, 5293446, 27297738, 142078746, 745387038, 3937603038, 20927156706, 111818026018, 600318853926, 3236724317174, 17518619320890, 95149655201962, 518431875418926, 2832923350929742, 15521467648875090

/*
Schroeder's second problem (generalized parentheses); also called super-Catalan numbers or little Schroeder numbers.
*/
1, 1, 3, 11, 45, 197, 903, 4279, 20793, 103049, 518859, 2646723, 13648869, 71039373, 372693519, 1968801519, 10463578353, 55909013009, 300159426963, 1618362158587, 8759309660445, 47574827600981, 259215937709463, 1416461675464871
```

6.8.2 Bell 数

```cpp
/*
   贝尔数
*/
const int maxn = 2000 + 5;
int bell[maxn][maxn];
void f(int n) {
  bell[1][1] = 1;
  for (int i = 2; i <= n; i++) {
    bell[i][1] = bell[i - 1][i - 1];
    for (int j = 2; j <= i; j++)
      bell[i][j] = bell[i - 1][j - 1] + bell[i][j - 1];
  }
}
```

6.8.3 Catalan 数

```cpp
/*
Catalan数
*/
int n;
long long f[25];
int main() {
  f[0] = 1;
  cin >> n;
  for (int i = 1; i <= n; i++) f[i] = f[i - 1] * (4 * i - 2) / (i + 1);
  // 这里用的是常见公式2
  cout << f[n] << endl;
```

```
    return 0;
}
```

6.8.4 超级 Catalan 数

```
/*
超级卡特兰数（又称大施罗德数）
*/

const int maxn=3e5;
typedef long long LL;
#define mod 998244353

LL num[maxn];

LL inverse(LL x,LL y)///快速幂加费马小定理求逆元
{
    LL sum=1;

    while(y)
    {

        if(y&1) sum=sum*x%mod;

        y/=2;
        x=x*x%mod;

    }
    return sum%mod;
}


int main()
{
    int n;

    while(~scanf("%d",&n))
    {
        num[1]=num[0]=1;
        if(n==1) {
            printf("1\n");continue;
        }

        for(int i=2;i<=n;i++)
        {
            num[i]=((6*i-3)*num[i-1]%mod-(i-2)*num[i-2]%mod+mod)%mod*inverse(i+1,mod-2)%
                mod;
        }

        printf("%lld\n",num[n-1]*2%mod);
    }
    return 0;
```

```
}
```

6.8.5 stirling

```
/*
第一类斯特林数 S1(n,m) 表示的是将 n 个不同元素构成 m 个圆排列的数目。
*/

const int mod=1e9+7;//取模
LL s[N][N];//存放要求的第一类Stirling数
void init(){
    memset(s,0,sizeof(s));
    s[1][1]=1;
    for(int i=2;i<=N-1;i++){
        for(int j=1;j<=i;j++){
            s[i][j]=s[i-1][j-1]+(i-1)*s[i-1][j];
            if(s[i][j]>=mod)
                s[i][j]%=mod;
        }
    }
}

/*
第二类斯特林数 S2(n,m) 表示的是把 n 个不同元素划分到 m 个集合的方案数。
*/

const int mod=1e9+7;//取模
LL s[N][N];//存放要求的Stirling数
void init(){
    memset(s,0,sizeof(s));
    s[1][1]=1;
    for(int i=2;i<=N-1;i++){
        for(int j=1;j<=i;j++){
            s[i][j]=s[i-1][j-1]+j*s[i-1][j];
            if(s[i][j]>=mod)
                s[i][j]%=mod;
        }
    }
}

/*
S(n,m) 的奇偶性
*/

int calc(int n,int m){
    return (m&n)==n;
}
int main(){
    int n,m;
    scanf("%d%d",&n,&m);
    if(n==0&&m==0)//特判
        printf("1\n");
```

```
    else if(n==0||m==0||n<m)//特判
        printf("0\n");
    else{
        int a=n-m;
        int b=(m+1)/2;
        int res=calc(b-1,a+b-1);
        printf("%d\n",res);
    }
    return 0;
}
```

7 图论

7.1 有向图判环

对一个有向图的节点进行拓扑排序，可以用来判断该有向图是否成环，有环则无拓扑序列，无环则有。

7.2 HK 算法

```
/*
1. 时间复杂度为$O(m\sqrt n)$
2. 点的序号要从0开始!
3. 需要把nx，ny都赋值为n(点数)
*/
const int MAXN = 1010;
const int MAXM = 1010*1010;

struct Edge {
    int v;
    int next;
} edge[MAXM];

struct node {
    double x, y;
    double v;
} a[MAXN], b[MAXN];

int nx, ny;
int cnt;
int t;
int dis;


int first[MAXN];
int xlink[MAXN], ylink[MAXN];
/*xlink[i]表示左集合顶点所匹配的右集合顶点序号，ylink[i]表示右集合i顶点匹配到的左集合顶点序号。*/
int dx[MAXN], dy[MAXN];
/*dx[i]表示左集合i顶点的距离编号，dy[i]表示右集合i顶点的距离编号*/
```

```
int vis[MAXN]; //寻找增广路的标记数组

void init() {
    cnt = 0;
    memset(first, -1, sizeof(first));
    memset(xlink, -1, sizeof(xlink));
    memset(ylink, -1, sizeof(ylink));
}

void read_graph(int u, int v) {
    edge[cnt].v = v;
    edge[cnt].next = first[u], first[u] = cnt++;
}

int bfs() {
    queue<int> q;
    dis = INF;
    memset(dx, -1, sizeof(dx));
    memset(dy, -1, sizeof(dy));
    for(int i = 0; i < nx; i++) {
        if(xlink[i] == -1) {
            q.push(i);
            dx[i] = 0;
        }
    }
    while(!q.empty()) {
        int u = q.front();
        q.pop();
        if(dx[u] > dis) break;
        for(int e = first[u]; e != -1; e = edge[e].next) {
            int v = edge[e].v;
            if(dy[v] == -1) {
                dy[v] = dx[u] + 1;
                if(ylink[v] == -1) dis = dy[v];
                else {
                    dx[ylink[v]] = dy[v]+1;
                    q.push(ylink[v]);
                }
            }
        }
    }
    return dis != INF;
}

int find(int u) {
    for(int e = first[u]; e != -1; e = edge[e].next) {
        int v = edge[e].v;
        if(!vis[v] && dy[v] == dx[u]+1) {
            vis[v] = 1;
            if(ylink[v] != -1 && dy[v] == dis) continue;
            if(ylink[v] == -1 || find(ylink[v])) {
                xlink[u] = v, ylink[v] = u;
                return 1;
```

```
            }
        }
    }
    return 0;
}

int MaxMatch() {
    int ans = 0;
    while(bfs()) {
        memset(vis, 0, sizeof(vis));
        for(int i = 0; i < nx; i++) if(xlink[i] == -1) {
                ans += find(i);
            }
    }
    return ans;
}

// 调用
init();
for(int i = 0; i < m; i++) {
    if(l[edgee[i][0]] && edgee[i][1] != s && !l[edgee[i][1]]) read_graph(edgee[i][0],
         edgee[i][1]);
    if(l[edgee[i][1]] && edgee[i][0] != s && !l[edgee[i][0]]) read_graph(edgee[i][1],
         edgee[i][0]);
}
nx = n;
ny = n;
int ans = MaxMatch();
```

7.3 点分治

```
const int maxn = 1e5 + 10;
struct edge{
    int dis;
    int nxt;
    int to;
}e[maxn<<1];
int head[maxn],tot;
inline void add(int u,int v,int dis){
    e[++tot].nxt=head[u];
    e[tot].dis=dis;
    e[tot].to=v;
    head[u]=tot;
}
int query[1010],rt,maxp[maxn],sum,size[maxn],vis[maxn],judge[10000000],rem[maxn],dis[
    maxn],test[10000000];
int q[maxn];
int n,m;
void getrt(int u,int pa){
    size[u]=1;
    maxp[u]=0;
    for(int i=head[u];i;i=e[i].nxt){
```

```
        int v=e[i].to;
        if(v==pa || vis[v]) continue;
        getrt(v,u);
        size[u]+=size[v];
        maxp[u]=max(maxp[u],size[v]);
    }
    maxp[u]=max(maxp[u],sum-size[u]);
    if(maxp[u]<maxp[rt]) rt=u;
}
void getdis(int u,int fa){
    rem[++rem[0]]=dis[u];
    for(int i=head[u];i;i=e[i].nxt){
        int v=e[i].to;
        if(v==fa|| vis[v]) continue;
        dis[v]=dis[u]+e[i].dis;
        getdis(v, u);
    }
}
void cal(int u){
    int p=0;
    for(int i=head[u];i;i=e[i].nxt){
        int v=e[i].to;
        if(vis[v]) continue;
        rem[0]=0;
        dis[v]=e[i].dis;
        getdis(v,u);
        for(int j=rem[0];j;--j)
            for(int k=1;k<=m;k++){
                if(query[k]>=rem[j]){
                    test[k]|=judge[query[k]-rem[j]];
                }
            }
        for(int j=rem[0];j;--j){
            q[++p]=rem[j];
            judge[rem[j]]=1;
        }
    }
    for(int i=1;i<=p;++i){
        judge[q[i]]=0;
    }
}
void solve(int u){
    vis[u]=judge[0]=1;
    cal(u);
    for(int i=head[u];i;i=e[i].nxt){
        int v=e[i].to;
        if(vis[v]) continue;
        sum=size[v];
        maxp[rt=0]=10000000;
        getrt(v, 0);
        solve(rt);
    }
}
```

```
int main(){
    scanf("%d%d",&n,&m);
    for(int i=1;i<n;i++){
        int u,v,dis;
        scanf("%d%d%d",&u,&v,&dis);
        add(u,v,dis);
        add(v,u,dis);
    }
    for(int i=1;i<=m;i++){
        scanf("%d",query+i);
    }
    maxp[rt]=sum=n;
    getrt(1,0);
    solve(rt);
    for(int i=1;i<=m;i++){
        if(test[i]) puts("AYE");
        else puts("NAY");
    }
}
```

7.4 树链剖分

```
#include <bits/stdc++.h>
#define Rint register int
#define mem(a,b) memset(a,(b),sizeof(a))
#define Temp template<typename T>
using namespace std;
typedef long long LL;
Temp inline void read(T &x){
    x=0;T w=1,ch=getchar();
    while(!isdigit(ch)&&ch!='-')ch=getchar();
    if(ch=='-')w=-1,ch=getchar();
    while(isdigit(ch))x=(x<<3)+(x<<1)+(ch^'0'),ch=getchar();
    x=x*w;
}

#define mid ((l+r)>>1)
#define lson rt<<1,l,mid
#define rson rt<<1|1,mid+1,r
#define len (r-l+1)

const int maxn=200000+10;
int n,m,r,mod;
//见题意
int e,beg[maxn],nex[maxn],to[maxn],w[maxn],wt[maxn];
//链式前向星数组, w[]、wt[]初始点权数组
int a[maxn<<2],laz[maxn<<2];
//线段树数组、lazy操作
int son[maxn],id[maxn],fa[maxn],cnt,dep[maxn],siz[maxn],top[maxn];
//son[]重儿子编号,id[]新编号,fa[]父亲节点,cnt dfs_clock/dfs序,dep[]深度,siz[]子树大小,top[]当前链顶端节点
int res=0;
```

```
//查询答案

inline void add(int x,int y){//链式前向星加边
    to[++e]=y;
    nex[e]=beg[x];
    beg[x]=e;
}
//--------------------------------------- 以下为线段树
inline void pushdown(int rt,int lenn){
    laz[rt<<1]+=laz[rt];
    laz[rt<<1|1]+=laz[rt];
    a[rt<<1]+=laz[rt]*(lenn-(lenn>>1));
    a[rt<<1|1]+=laz[rt]*(lenn>>1);
    a[rt<<1]%=mod;
    a[rt<<1|1]%=mod;
    laz[rt]=0;
}

inline void build(int rt,int l,int r){
    if(l==r){
        a[rt]=wt[l];
        if(a[rt]>mod)a[rt]%=mod;
        return;
    }
    build(lson);
    build(rson);
    a[rt]=(a[rt<<1]+a[rt<<1|1])%mod;
}

inline void query(int rt,int l,int r,int L,int R){
    if(L<=l&&r<=R){res+=a[rt];res%=mod;return;}
    else{
        if(laz[rt])pushdown(rt,len);
        if(L<=mid)query(lson,L,R);
        if(R>mid)query(rson,L,R);
    }
}

inline void update(int rt,int l,int r,int L,int R,int k){
    if(L<=l&&r<=R){
        laz[rt]+=k;
        a[rt]+=k*len;
    }
    else{
        if(laz[rt])pushdown(rt,len);
        if(L<=mid)update(lson,L,R,k);
        if(R>mid)update(rson,L,R,k);
        a[rt]=(a[rt<<1]+a[rt<<1|1])%mod;
    }
}
//----------------------------------以上为线段树
inline int qRange(int x,int y){
    int ans=0;
```

```
    while(top[x]!=top[y]){//当两个点不在同一条链上
        if(dep[top[x]]<dep[top[y]])swap(x,y);//把x点改为所在链顶端的深度更深的那个点
        res=0;
        query(1,1,n,id[top[x]],id[x]);//ans加上x点到x所在链顶端 这一段区间的点权和
        ans+=res;
        ans%=mod;//按题意取模
        x=fa[top[x]];//把x跳到x所在链顶端的那个点的上面一个点
    }
    //直到两个点处于一条链上
    if(dep[x]>dep[y])swap(x,y);//把x点深度更深的那个点
    res=0;
    query(1,1,n,id[x],id[y]);//这时再加上此时两个点的区间和即可
    ans+=res;
    return ans%mod;
}

inline void updRange(int x,int y,int k){//同上
    k%=mod;
    while(top[x]!=top[y]){
        if(dep[top[x]]<dep[top[y]])swap(x,y);
        update(1,1,n,id[top[x]],id[x],k);
        x=fa[top[x]];
    }
    if(dep[x]>dep[y])swap(x,y);
    update(1,1,n,id[x],id[y],k);
}

inline int qSon(int x){
    res=0;
    query(1,1,n,id[x],id[x]+siz[x]-1);//子树区间右端点为id[x]+siz[x]-1
    return res;
}

inline void updSon(int x,int k){//同上
    update(1,1,n,id[x],id[x]+siz[x]-1,k);
}

inline void dfs1(int x,int f,int deep){//x当前节点，f父亲，deep深度
    dep[x]=deep;//标记每个点的深度
    fa[x]=f;//标记每个点的父亲
    siz[x]=1;//标记每个非叶子节点的子树大小
    int maxson=-1;//记录重儿子的儿子数
    for(Rint i=beg[x];i;i=nex[i]){
        int y=to[i];
        if(y==f)continue;//若为父亲则continue
        dfs1(y,x,deep+1);//dfs其儿子
        siz[x]+=siz[y];//把它的儿子数加到它身上
        if(siz[y]>maxson)son[x]=y,maxson=siz[y];//标记每个非叶子节点的重儿子编号
    }
}

inline void dfs2(int x,int topf){//x当前节点，topf当前链的最顶端的节点
    id[x]=++cnt;//标记每个点的新编号
```

```
    wt[cnt]=w[x];//把每个点的初始值赋到新编号上来
    top[x]=topf;//这个点所在链的顶端
    if(!son[x])return;//如果没有儿子则返回
    dfs2(son[x],topf);//按先处理重儿子，再处理轻儿子的顺序递归处理
    for(Rint i=beg[x];i;i=nex[i]){
        int y=to[i];
        if(y==fa[x]||y==son[x])continue;
        dfs2(y,y);//对于每一个轻儿子都有一条从它自己开始的链
    }
}

int main(){
    read(n);read(m);read(r);read(mod);
    for(Rint i=1;i<=n;i++)read(w[i]);
    for(Rint i=1;i<n;i++){
        int a,b;
        read(a);read(b);
        add(a,b);add(b,a);
    }
    dfs1(r,0,1);
    dfs2(r,r);
    build(1,1,n);
    while(m--){
        int k,x,y,z;
        read(k);
        if(k==1){
            read(x);read(y);read(z);
            updRange(x,y,z);
        }
        else if(k==2){
            read(x);read(y);
            printf("%d\n",qRange(x,y));
        }
        else if(k==3){
            read(x);read(y);
            updSon(x,y);
        }
        else{
            read(x);
            printf("%d\n",qSon(x));
        }
    }
}
```

7.5 矩阵树定理

```
/*
取模（不用逆元版本）
1. 图中节点的下标从0开始计数!
2. 不存在自环，允许存在重边
3. 求行列式参数为n，求生成树计数参数为n-1
*/
```

```
struct Matrix {
    ll mat[N][N];
    void init() {
        memset(mat,0,sizeof(mat));
    }
    void addEdge(int u,int v) {
        mat[u][v]--;
        mat[u][u]++;
    }
    ll det(int n){
        ll res=1;
        for(int i=0;i<n;++i){
            if(!mat[i][i]){
                bool flag=false;
                for(int j=i+1;j<n;++j){
                    if(mat[j][i]){
                        flag=true;
                        for(int k=i;k<n;++k) swap(mat[i][k],mat[j][k]);
                        res=-res;
                        break;
                    }
                }
                if(!flag) return 0;
            }
            for(int j=i+1;j<n;++j){
                while(mat[j][i]){
                    ll t=mat[i][i]/mat[j][i];
                    for(int k=i;k<n;++k){
                        mat[i][k]=(mat[i][k]-t*mat[j][k])%mod;
                        swap(mat[i][k],mat[j][k]);
                    }
                    res=-res;
                }
            }
            res*=mat[i][i];
            res%=mod;//模意义下的语句，不是模意义则不加
        }
        if(res<0) res+=mod;
        return res;
    }
}ret;

/*
取逆元 (mod为质数)
1. 图中节点的下标从0开始计数!
2. 不存在自环，允许存在重边
3. 求行列式参数为n，求生成树计数参数为n-1
*/
ll inv(ll a) {
    if(a == 1)return 1;
    return inv(mod%a)*(mod-mod/a)%mod;
}

```

```
struct Matrix {
    ll mat[N][N];
    void init() {
        memset(mat,0,sizeof(mat));
    }
    void addEdge(int u,int v) {
        mat[u][v]--;
        mat[u][u]++;
    }
    ll det(int n) { //求行列式的值模上MOD，需要使用逆元
        for(int i = 0; i < n; i++)
            for(int j = 0; j < n; j++)
                mat[i][j] = (mat[i][j]%mod+mod)%mod;
        ll res = 1;
        for(int i = 0; i < n; i++) {
            for(int j = i; j < n; j++)
                if(mat[j][i]!=0) {
                    for(int k = i; k < n; k++)
                        swap(mat[i][k],mat[j][k]);
                    if(i != j)
                        res = (-res+mod)%mod;
                    break;
                }
            if(mat[i][i] == 0) {
                res = 0;//不存在(也就是行列式值为0)
                break;
            }
            for(int j = i+1; j < n; j++) {
                //int mut = (mat[j][i]*INV[mat[i][i]])%MOD;//打表逆元
                ll mut = (mat[j][i]*inv(mat[i][i]))%mod;
                for(int k = i; k < n; k++)
                    mat[j][k] = (mat[j][k]-(mat[i][k]*mut)%mod+mod)%mod;
            }
            res = (res * mat[i][i])%mod;
        }
        return res;
    }
}ret;

/*
不取模
1. 图中节点的下标从1开始计数!
2. 不存在自环，允许存在重边
3. 求行列式参数为n，求生成树计数参数为n-1
*/

struct Matrix {
    ll mat[N][N];
    void init() {
        memset(mat, 0, sizeof mat);
    }
    ll gauss(int n) {
        ll res = 1;
```

```
        for (int i = 1; i <= n; i++) {
            for (int j = i + 1; j <= n; j++) {
                while (mat[j][i]) {
                    ll t = mat[i][i] / mat[j][i];
                    for (int k = i; k <= n; k++)
                        mat[i][k] = (mat[i][k] - t * mat[j][k]);
                    swap(mat[i], mat[j]);
                    res = -res;
                }
            }
            if(mat[i][i] == 0) return 0;
            res = res * mat[i][i];
        }
        if(res < 0) res = -res;
        return res;
    }
    void add(int u, int v) {
        mat[u][u]++;
        mat[v][v]++;
        mat[u][v]--;
        mat[v][u]--;
    }
}ret;
```

7.6 一般图最大匹配

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e3 + 5;
const int M = 5e4 + 5;

int n, m, m1, ti;
int fs[N], pre[N], match[N], f[N], bz[N], bp[N];
int nt[2*M], dt[2*M], d[N*N];

void link(int x, int y) {
    nt[++m1] = fs[x];
    dt[fs[x] = m1] = y;
}
int find(int x) {
    return f[x] == x ? x : f[x] = find(f[x]);
}
int lca(int x, int y) {
    ti++;
    x = find(x); y = find(y);
    while(bp[x] != ti) {
        bp[x] = ti;
        x = find(pre[match[x]]);
        if(y) swap(x, y);
    }
    return x;
}
```

```
void make(int x, int y, int w) {
    while(find(x) != w) {
        pre[x] = y; y = match[x];
        if(bz[y] == 2) {
            bz[y] = 1;
            d[++d[0]] = y;
        }
        if(find(x) == x) f[x] = w;
        if(find(y) == y) f[y] = w;
        x = pre[y];
    }
}
bool solve(int st) {
    for (int i = 1; i <= n; i++) {
        f[i] = i;
        pre[i] = bz[i] = 0;
    }
    d[d[0] = 1] = st;
    bz[st] = 1;
    int l = 0;
    while(l < d[0]) {
        int k = d[++l];
        for (int i = fs[k]; i; i = nt[i]) {
            int p = dt[i];
            if(find(p) == find(k) || bz[p] == 2) continue;
            if(!bz[p]) {
                bz[p]=2; pre[p]=k;
                if(!match[p]) { //找到增广路
                    for(int x=p,y; x; x=y) {
                        y = match[pre[x]];
                        match[x] = pre[x];
                        match[pre[x]] = x;//返回修改匹配
                    }
                    return 1;
                }
                bz[match[p]] = 1;
                d[++d[0]] = match[p];//否则将其匹配点加入队列
            }
            else {
                int w = lca(k,p);
                make(k, p, w);
                make(p, k, w);//以上分别修改k到lca的路径以及p到lca的路径（环的两半）
            }
        }
    }
    return 0;
}
int main() {
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= m; i++) {
        int x, y;
        scanf("%d%d", &x, &y);
        link(x, y); link(y, x);
```

```
    }
    int ans = 0;
    for (int i = 1; i <= n; i++) {
        if(!match[i]) ans += solve(i);
    }
    printf("%d\n", ans);
    for (int i = 1; i <= n; i++) {
        printf("%d ", match[i]);
    }
}
```

7.7 最近公共祖先

```
#include <bits/stdc++.h>
using namespace std;
typedef pair<int,int> pi;
typedef long long ll;
const int maxn = 1e6 + 5;
int n,m,s,cnt=0;
int head[maxn],dep[maxn],fa[maxn][64],lg[maxn];
struct node
{
    int to,nex;
}e[maxn];
inline void add(int x,int y)
{
    e[++cnt].nex=head[x];
    e[cnt].to=y;
    head[x]=cnt;
}
void dfs(int u,int f)
{
    dep[u]=dep[f]+1;
    fa[u][0]=f;
    for(int i=1;(1<<i)<=dep[u];++i)
        fa[u][i]=fa[fa[u][i-1]][i-1];
    for(int i=head[u];i;i=e[i].nex)
    {
        if(e[i].to==f) continue;
        dfs(e[i].to,u);
    }
}
int lca(int x,int y)
{
    if(dep[x]<dep[y]) swap(x,y);
    while(dep[x]>dep[y])
        x=fa[x][lg[dep[x]-dep[y]]-1];
    if(x!=y)
    {
        for(int i=lg[dep[x]];i>=0;--i)
            if(fa[x][i]!=fa[y][i])
            {
```

```
                x=fa[x][i];
                y=fa[y][i];
            }
        x=fa[x][0];
    }
    return x;
}
int main()
{
    cin >> n >> m >> s;
    for (int i = 1; i < n; i++)
    {
        int x,y;
        cin >> x >> y;
        add(x,y);add(y,x);
    }
    dfs(s,0);
    for (int i = 1; i <= n; i++)
        lg[i]=lg[i-1]+(1<<lg[i-1]==i);
    for (int i = 1; i <= m; i++)
    {
        int x,y;
        cin >> x >> y;
        printf("%d\n",lca(x,y));
    }
    return 0;
}
```

7.8 最小树形图

```
/*
最小树形图，就是给出一个带权有向图，从中指定一个特殊的结点 root.求一棵以 root 为根的有向生成树 T，且使得 T 中所有边权值最小。
*/

const int M = 1e4 + 5;
const int N = 105;
struct Edge {
    int u, v, w;
}e[M];
const int inf = 2e9;
int n, m, root;
int pre[N], ine[N];
int vis[N], id[N];
int zhuliu() {
    int ans = 0;
    while(1) {
        for (int i = 1; i <= n; i++) ine[i] = inf;
        for (int i = 1; i <= m; i++) {
            if(e[i].u != e[i].v && e[i].w < ine[e[i].v]) {
                ine[e[i].v] = e[i].w;
                pre[e[i].v] = e[i].u;
```

```
            }
        }
        for (int i = 1; i <= n; i++) {
            if(i != root && ine[i] == inf) return -1;
        }
        int cnt = 0;
        for (int i = 1; i <= n; i++) vis[i] = id[i] = 0;
        for (int i = 1; i <= n; i++) {
            if(i == root) continue;
            ans += ine[i];
            int v = i;
            while(vis[v] != i && !id[v] && v != root) {
                vis[v] = i;
                v = pre[v];
            }
            if(!id[v] && v != root) {
                id[v] = ++cnt;
                for (int u = pre[v]; u != v; u = pre[u]) {
                    id[u] = cnt;
                }
            }
        }
        if(cnt == 0) break;
        for (int i = 1; i <= n; i++) if(!id[i]) id[i] = ++cnt;
        for (int i = 1; i <= m; i++) {
            int u = e[i].u, v = e[i].v;
            e[i].u = id[u], e[i].v = id[v];
            if(id[u] != id[v]) e[i].w -= ine[v];
        }
        root = id[root];
        n = cnt;
    }
    return ans;
}
int main() {
    scanf("%d%d%d", &n, &m, &root);
    for (int i = 1; i <= m; i++) {
        scanf("%d%d%d", &e[i].u, &e[i].v, &e[i].w);
    }
    printf("%d\n", zhuliu());
}

/*
无根树最小树形图
*/
#include <bits/stdc++.h>
#define lowbit(x) ( x&(-x) )
#define INF 0x3f3f3f3f
using namespace std;
typedef unsigned long long ull;
typedef long long ll;
const int maxN = 1005;
int N, M;
```

```
ll sum;
struct Eddge //存边
{
    int u, v;
    ll val;
    Eddge(int a=0, int b=0, ll c=0):u(a), v(b), val(c) {}
}edge[maxN*maxN];
int pre[maxN], id[maxN], vis[maxN], pos;
ll in[maxN]; //最小入边权，pre[]为其前面的点（该边的起点）
ll Dir_MST(int root, int V, int E) //root是此时的根节点，我们最初的时候将0（万能节点作为根节点进入），V是点的个数（包括之后要收缩之后点的剩余个数），E是边的条数（不会改变）
{
    ll ans = 0;
    while(true) //如果还是可行的话
    {
        for(int i=0; i<V; i++) in[i] = INF; //给予每个点进行初始化
        /* (1)、最短弧集合E0 */
        for(int i=1; i<=E; i++) //通过这么多条单向边，确定的是每个点的指向边的最小权值
        {
            int u = edge[i].u, v = edge[i].v;
            if(edge[i].val < in[v] && u!=v) //顶点v有更小的入边，记录下来 更新操作，u!=v是为了确保缩点之后，我们的环将会变成点的形式
            {
                pre[v] = u; //节点u指向v
                in[v] = edge[i].val; //最小入边
                if(u == root) pos = i; //这个点就是实际的起点
            }
        }
        /* (2)、检查E0 */
        for(int i=0; i<V; i++) //判断是否存在最小树形图
        {
            if(i == root) continue; //是根节点，不管
            if(in[i] == INF) return -1; //除了根节点以外，有点没有入边，则根本无法抵达它，说明是独立的点，一定不能构成树形图
        }
        /* (3)、收缩图中的有向环 */
        int cnt = 0; //接下来要去求环，用以记录环的个数 找环开始!
        memset(id, -1, sizeof(id));
        memset(vis, -1, sizeof(vis));
        in[root] = 0;
        for(int i=0; i<V; i++) //标记每个环
        {
            ans += in[i]; //加入每个点的入边（既然是最小入边，所以肯定符合最小树形图的思想）
            int v = i; //v一开始先从第i个节点进去
            while(vis[v] != i && id[v] == -1 && v != root) //退出的条件有“形成了一个环，即vis回归”、“到了一个环，此时就不要管了，因为那边已经建好环了”、“到了根节点，就是条链，不用管了”
            {
                vis[v] = i;
                v = pre[v];
            }
            if(v != root && id[v] == -1) //如果v是root就说明是返回到了根节点，是条链，没环；又或者，它已经是进入了对应环的编号了，不需要再跑一趟了
```

```
            {
                for(int u=pre[v]; u!=v; u=pre[u]) //跑这一圈的环
                {
                    id[u] = cnt; //标记点u是第几个环
                }
                id[v] = cnt++; //如果再遇到，就是下个点了
            }
        }
        if(cnt == 0) return ans; //无环的情况，就说明已经取到了最优解，直接返回，或者说是环已经收缩到没有环的情况了
        for(int i=0; i<V; i++) if(id[i] == -1) id[i] = cnt++; //这些点是环外的点，是链上的点，单独再给他们赋值
        for(int i=1; i<=E; i++) //准备开始建立新图 缩点，重新标记
        {
            int u = edge[i].u, v = edge[i].v;
            edge[i].u = id[u]; edge[i].v = id[v]; //建立新图，以新的点进入
            if(id[u] != id[v]) edge[i].val -= in[v]; //为了不改变原来的式子，使得展开后还是原来的式子
        }
        V = cnt; //之后的点的数目
        root = id[root]; //新的根节点的序号，因为id[]的改变，所以根节点的序号也改变了
    }
    return ans;
}
int main()
{
    while(scanf("%d%d", &N, &M)!=EOF)
    {
        sum = 0;
        for(int i=1; i<=M; i++)
        {
            scanf("%d%d%lld", &edge[i].u, &edge[i].v, &edge[i].val);
            edge[i].u++; edge[i].v++; //把‘0’号节点空出来，用以做万能节点，留作之后用
            sum += edge[i].val;
        }
        sum++; //一定要把sum给扩大，这就意味着，除去万能节点以外的点锁构成的图的权值和得在(sum-1)之内(包含)
        for(int i=M+1; i<=M+N; i++) //这就是万能节点了，就是从0这号万能节点有通往所有其他节点的路，而我们最后的最小树形图就是从这个万能节点出发所能到达的整幅图
        {
            edge[i] = Eddge(0, i-M, sum); //对于所有的N个其他节点都要建有向边
        } //此时N+1为总的节点数目，M+N为总的边数
        ll ans = Dir_MST(0, N + 1, M+N); //ans代表以超级节点0为根的最小树形图的总权值
        if(ans == -1 || ans - sum >= sum) printf("impossible\n"); //从万能节点的出度只能是1，所以最后的和必须是小于sum的，而万能节点的出度就由“ans - sum >= sum”保证
        else printf("%lld %d\n", ans - sum, pos - M - 1); //pos-M得到的是1~N的情况，所以“-1”的目的就在于这里
        printf("\n");
    }
    return 0;
}
```

7.9 二分图

性质

一般图:

1. 对于不存在孤立点的图，| 最大匹配 |+| 最小边覆盖 |=|E|

2.| 最大独立集 |+| 最小顶点覆盖 |=|V|

二分图:

| 最大匹配 |=| 最小顶点覆盖 |

7.10 判无向图是否为二分图

判断无向图是否为二分图，等价于判无向图内是否有奇环，有两种方式:

1. 染色法 dfs/bfs 遍历整个图，对相邻节点染不同颜色，遇矛盾则说明不是二分图。

2. 用种类并查集实现，原理与染色法相似，但支持并查集的可持久化、可撤销、可删边等操作。

7.11 KM 算法

```cpp
/*
> 最优匹配——**定理**: 设$M$是一个带权完全二分图$G$的一个完备匹配，给每个顶点一个可行顶标（第$i$个$x$顶点的可行顶标用 $x_i$表示，第$j$个y顶点的可行顶标用$y_j$表示），如果对所有的边$(i,j)$ in $G$，都有$x_i+y_j w_{i,j}$成立（$w_{i,j}$表示边的权），且对所有的边$(i,j)$ in $M$，都有 成立，则$M$是图$G$的$x_i+y_j w_{i,j}$一个最优匹配。
- 时间复杂度$O(n^3)$
- 最小权值匹配，边权取反，答案取反
- $x_i+y_j>=w_{i,j}$，最小顶标和等价于最大权值匹配
  $x_i+y_j<=w_{i,j}$，最大顶标和等价于最小权值匹配
*/
typedef long long ll;
const int N = 405;
const ll INF = LONG_LONG_MAX;
struct KM
{
    int link_x[N], link_y[N], n, nx, ny;
    bool visx[N], visy[N];
    int que[N << 1], top, fail, pre[N];
    ll mp[N][N], hx[N], hy[N], slk[N];
    inline int check(int i)
    {
        visx[i] =true;
        if(link_x[i])
        {
            que[fail++] = link_x[i];
            return visy[link_x[i]] = true;
        }
        while(i)
```

```
    {
        link_x[i] = pre[i];
        swap(i, link_y[pre[i]]);
    }
    return 0;
}
void bfs(int S)
{
    for(int i=1; i<=n; i++)
    {
        slk[i] = INF;
        visx[i] = visy[i] = false;
    }
    top = 0; fail = 1;
    que[0] = S;
    visy[S] = true;
    while(true)
    {
        ll d;
        while(top < fail)
        {
            for(int i = 1, j = que[top++]; i <= n; i++)
            {
                if(!visx[i] && slk[i] >= (d = hx[i] + hy[j] - mp[i][j]))
                {
                    pre[i] = j;
                    if(d) slk[i] = d;
                    else if(!check(i)) return;
                }
            }
        }
        d = INF;
        for(int i=1; i<=n; i++)
        {
            if(!visx[i] && d > slk[i]) d = slk[i];
        }
        for(int i=1; i<=n; i++)
        {
            if(visx[i]) hx[i] += d;
            else slk[i] -= d;
            if(visy[i]) hy[i] -= d;
        }
        for(int i=1; i<=n; i++)
        {
            if(!visx[i] && !slk[i] && !check(i)) return;
        }
    }
}
void init(int cntx, int cnty)
{
    nx = cntx; ny = cnty; n = max(nx, ny);
    top = fail = 0;
    for (int i = 1; i <= n; i++) {
```

```
            link_x[i] = link_y[i] = pre[i] = 0;
            hx[i] = hy[i] = 0;
            for (int j = 1; j <= n; j++) mp[i][j] = 0;
        }
    }
    void solve() {
        for(int i=1; i<=n; i++) for(int j=1; j<=n; j++)
            if(hx[i] < mp[i][j]) hx[i] = mp[i][j];
        ll ans = 0;
        for (int i = 1; i <= n; i++) bfs(i);
        for (int i = 1; i <= nx; i++) ans += mp[i][link_x[i]];
        printf("%lld\n", ans); // 输出最大权值

        /*
        for (int i = 1; i <= nx; i++) if(!mp[i][link_x[i]])
            link_x[i] = 0; // 如果不存在该边则置零
        for (int i = 1; i <= nx; i++) {
            if(i != 1) printf(" ");
            printf("%d", link_x[i]); // 输出每一组匹配
        }
        printf("\n");
         */
    }
}km;

// 调用
int n; scanf("%d", &n);
km.init(n, n);
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        ll w; scanf("%lld", &w);
        km.mp[i][j] = w;
    }
}
km.solve();
```

7.12 染色法判二分图

```
/*
1. 若图G是非连通图，则需要依次判断各连通子图是不是二分图。
2. 无向图加双向边
*/
vector<int> G[maxn];
int color[maxn];

int bfs(int u) {
    queue<int> Q;
    Q.push(u);
    color[u]=1;

    while(!Q.empty()) {
        int t=Q.front();
```

```
        Q.pop();
        for(int i=0; i<G[t].size(); i++) {
            if(color[G[t][i]]==color[t])
                return 0; //Adjacent nodes are the same color, not bipartite graph
            else if(color[G[t][i]]==0) { //no dyeing
                color[G[t][i]]=-color[t];
                Q.push(G[t][i]);
            }
        }
    }

    return 1;
}
```

7.13 匈牙利算法

```
/*
1. 时间复杂度O(nm)
2. 邻接矩阵建双向边
3. ans为最大匹配数
*/
int mp[maxn][maxn]; // 图的存储矩阵
int n, m, ans;
bool vis[maxn]; // 当前搜索过程中是否被访问过
int link[maxn]; // y集合中的点在x集合中的匹配点 -1表示未匹配

bool find_(int x) {
    for (int i=1; i<=n; ++i) {
        if (mp[x][i] && !vis[i]) { // 有边相连
            vis[i] = 1; // 标记该点
            if (link[i] == -1 || find_(link[i])) { //该点未匹配 或者匹配的点能找到增光路
                link[i] = x; // 删掉偶数条边 加进奇数条边
                return true; // 找到增光路
            }
        }
    }
    return false;
}

void match() {
    //初始化
    ans = 0;
    memset(link, -1, sizeof(link));

    for (int i=1; i<=n; ++i) {
        memset(vis, 0, sizeof(vis)); // 从集合的每个点依次搜索
        if (find_(i)) // 如果能搜索到 匹配数加1
            ans++;
    }
    return;
}
```

7.14 矩阵树定理

矩阵树定理

对于生成树的计数，一般采用矩阵树定理 (Matrix-Tree 定理) 来解决。

Matrix-Tree 定理的内容为：对于已经得出的基尔霍夫矩阵，去掉其随意一行一列得出的矩阵的行列式，其绝对值为生成树的个数

因此，对于给定的图 G，若要求其生成树个数，可以先求其基尔霍夫矩阵，然后随意取其任意一个 n-1 阶行列式，然后求出行列式的值，其绝对值就是这个图中生成树的个数。

度数矩阵 D[G]：当 $i \neq j$ 时，$D[i][j] = 0$，当 $i = j$ 时，$D[i][i] = degree(v_i)$

邻接矩阵 A[G]：当 v_i、v_j 有边连接时，$A[i][j] = 1$，当 v_i、v_j 无边连接时，$A[i][j] = 0$

基尔霍夫矩阵 (Kirchhoff) K[G]：也称拉普拉斯算子，其定义为 $K[G] = D[G] - A[G]$，即：$K[i][j] = D[i][j] - A[i][j]$

7.15 启发式合并

```cpp
/*
树上查询问题，满足两个特征：
1. 只有对子树的查询
2. 没有修改
模版：CF-600E 统计子树中个数最多的颜色
*/

#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn = 1e5 + 5;
int c[maxn], sz[maxn], son[maxn];
ll ans[maxn], now;
int cnt[maxn], mx, Son;
vector<int> g[maxn];
void add(int u, int f, int val) {
    cnt[c[u]] += val;
    if(mx < cnt[c[u]]) mx = cnt[c[u]], now = c[u];
    else if(mx == cnt[c[u]]) now += (ll)c[u];
    for (auto v : g[u]) {
        if(v == f || v == Son) continue;
        add(v, u, val);
    }
}
void dfs1(int u, int f) {
    sz[u] = 1;
    for (auto v : g[u]) {
        if(v == f) continue;
        dfs1(v, u);
        if(sz[son[u]] < sz[v]) son[u] = v;
        sz[u] += sz[v];
    }
```

```cpp
}
void dfs2(int u, int f, int opt) {
    for (auto v : g[u]) {
        if(v == f || v == son[u]) continue;
        dfs2(v, u, 0);
    }
    if(son[u]) dfs2(son[u], u, 1), Son = son[u];
    add(u, f, 1); Son = 0;
    ans[u] = now;
    if(!opt) add(u, f, -1), mx = 0, now = 0;
}
int main() {
    int n;
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
        scanf("%d", &c[i]);
    }
    for (int i = 1; i < n; i++) {
        int u, v;
        scanf("%d%d", &u, &v);
        g[u].push_back(v);
        g[v].push_back(u);
    }
    dfs1(1, 0);
    dfs2(1, 0, 0);
    for (int i = 1; i <= n; i++) printf("%lld%c", ans[i], i<n?' ':'\n');
}
```

7.16 2-SAT

```cpp
/*
2-SAT，简单的说就是给出n个集合，每个集合有两个元素，已知若干个<a,b>，表示a与b矛盾（其中a与b
    属于不同的集合）。然后从每个集合选择一个元素，判断能否一共选n个两两不矛盾的元素。
*/
const int N = 2e6 + 5;
int low[N], dfn[N], ins[N], color[N];
stack<int> stk;
vector<int> g[N];
int sccCnt, dfsClock, n, m;
// 注意所有东西都要开两倍空间，因为每个变量存了两次
void tarjan(int u) {
    low[u] = dfn[u] = ++dfsClock;
    stk.push(u); ins[u] = true;
    for (const auto &v : g[u]) {
        if (!dfn[v]) tarjan(v), low[u] = min(low[u], low[v]);
        else if (ins[v]) low[u] = min(low[u], dfn[v]);
    }
    if (low[u] == dfn[u]) {
        ++sccCnt;
        do {
            color[u] = sccCnt;
            u = stk.top(); stk.pop(); ins[u] = false;
```

```cpp
        } while (low[u] != dfn[u]);
    }
}
int main() {
    n = read(), m = read();
    for (int i = 0; i < m; ++i) {
        int a = read(), va = read(), b = read(), vb = read();
        g[a + n * (va & 1)].push_back(b + n * (vb ^ 1));
        g[b + n * (vb & 1)].push_back(a + n * (va ^ 1));
    }
    // Tarjan 找环，得到的 color[x] 是 x 所在的 scc 的拓扑逆序。
    for (int i = 1; i <= (n << 1); ++i) if (!dfn[i]) tarjan(i);
    for (int i = 1; i <= n; ++i)
    if (color[i] == color[i + n]) { // x 与 -x 在同一强连通分量内，一定无解
        puts("IMPOSSIBLE");
        exit(0);
    }
    puts("POSSIBLE");
    for (int i = 1; i <= n; ++i)
        printf("%d%c",(color[i] < color[i + n]),i<n?' ':'\n');
        // 如果不使用 Tarjan 找环，请改成大于号
}
```

7.17 网络流

7.17.1 网络流

最大流

设点数为 n，边数为 m，那么 Dinic 算法的时间复杂度（在应用上面两个优化的前提下）是 $O(n^2m)$，在稀疏图上效率和 EK 算法相当，但在稠密图上效率要比 EK 算法高很多。

最小割

最大流最小割定理 $f(s,t)_{max} = c(s,t)_{min}$

费用流

7.17.2 最大费用流

```cpp
/*
    EK + SPFA
*/
const int maxn = 1e5+10;
struct node {
    int to,w,f,nxt;
} e[maxn];
int n,m,s,t,maxflow,mincost;
int dis[maxn],flow[maxn],via[maxn],pre[maxn],last[maxn];
int head[maxn],cnt=0;
```

```
void init() {
    memset(head,-1,sizeof(head));
    cnt=0;
}
void ade( int u, int v, int f, int w )
{
    e[cnt].to = v;
    e[cnt].f = f;
    e[cnt].w = w;
    e[cnt].nxt = head[u];
    head[u] = cnt++;
}
void add(int u, int v, int f, int w) {
    w = -w; // 最大费用
    ade(u, v, f, w);
    ade(v, u, 0, -w);
}

int spfa()
{
    memset(dis,inf,sizeof(dis));
    memset(flow,inf,sizeof(flow));
    memset(via,0,sizeof(via));
    queue <int> Q;
    Q.push(s); via[s]=1; dis[s]=0; pre[t]=-1;
    while ( !Q.empty() ) {
        int x = Q.front(); Q.pop(); via[x]=0;
        for ( int i=head[x]; i!=-1; i=e[i].nxt ) {
            int y = e[i].to, f=e[i].f, w=e[i].w;
            if ( f && dis[y]>dis[x]+w ) { // 只要最短流能更新就更新
                dis[y] = dis[x] + w;
                pre[y] = x; // y 的父节点是x
                last[y] = i; // y点连接其父节点的边，编号为i
                flow[y] = min(flow[x],f); // 源点到y点的最大流量。会被最小的一个分支限制住
                if ( via[y]==0 ) { // 只有队列中没有当前值才往队列里加。
                    Q.push(y); via[y]=1;
                }
            }
        }
    }
    return pre[t]!=-1; // 判断汇点是否有点连入，即还存不存在增广路。初始化pre[t]=-1.
}

void EK()
{
    maxflow = mincost = 0;
    while ( spfa() ) { // 还存在增广路就进入
        int x = t;
        maxflow += flow[t]; // 源点到t点的最大流量
        mincost += flow[t]*dis[t];
        while ( x!=s ) { // 递归改变边的流量
            e[last[x]].f -= flow[t];
            e[last[x]^1].f += flow[t];
```

```
            x = pre[x];
        }
    }
}

/*
调用:
EK();
cout << -mincost << endl;
 */
```

7.17.3 最大费用最大流

```
/*
    EK + SPFA
*/
const int maxn = 1e5+10;
struct node {
    int to,w,f,nxt;
} e[maxn];
int n,m,s,t,maxflow,mincost;
int dis[maxn],flow[maxn],via[maxn],pre[maxn],last[maxn];
int head[maxn],cnt=0;
void init() {
    memset(head,-1,sizeof(head));
    cnt=0;
}
void ade( int u, int v, int f, int w )
{
    e[cnt].to = v;
    e[cnt].f = f;
    e[cnt].w = w;
    e[cnt].nxt = head[u];
    head[u] = cnt++;
}
void add(int u, int v, int f, int w) {
    w = -w; // 最大费用
    ade(u, v, f, w);
    ade(v, u, 0, -w);
}

int spfa()
{
    memset(dis,inf,sizeof(dis));
    memset(flow,inf,sizeof(flow));
    memset(via,0,sizeof(via));
    queue <int> Q;
    Q.push(s); via[s]=1; dis[s]=0; pre[t]=-1;
    while ( !Q.empty() ) {
        int x = Q.front(); Q.pop(); via[x]=0;
        for ( int i=head[x]; i!=-1; i=e[i].nxt ) {
            int y = e[i].to, f=e[i].f, w=e[i].w;
```

```
            if ( f && dis[y]>dis[x]+w ) { // 只要最短流能更新就更新
                dis[y] = dis[x] + w;
                pre[y] = x; // y 的父节点是x
                last[y] = i; // y点连接其父节点的边，编号为i
                flow[y] = min(flow[x],f); // 源点到y点的最大流量。会被最小的一个分支限制住
                if ( via[y]==0 ) { // 只有队列中没有当前值才往队列里加。
                    Q.push(y); via[y]=1;
                }
            }
        }
    }
    return pre[t]!=-1; // 判断汇点是否有点连入，即还存不存在增广路。初始化pre[t]=-1.
}

void EK()
{
    maxflow = mincost = 0;
    while ( spfa() ) { // 还存在增广路就进入
        int x = t;
        maxflow += flow[t]; // 源点到t点的最大流量
        mincost += flow[t]*dis[t];
        while ( x!=s ) { // 递归改变边的流量
            e[last[x]].f -= flow[t];
            e[last[x]^1].f += flow[t];
            x = pre[x];
        }
    }
}

/*
调用：
EK();
cout << -mincost << endl;
 */
```

7.17.4 zkw 费用流

```
/*
1. zkw费用流适用于稠密图；
2. 使用前调用init()，初始化n、s、t；
*/
const int N = 1e3 + 5;
const int M = 5e5 + 5;
const int inf = 0x3f3f3f3f;
namespace zkw {
    struct Edge {
        int to, nex, f, c;
    } e[M];
    int head[N], tot;
    int n, s, t;
    int level[N], dis[N];
    int maxf, cost;
```

```
    bool flag, vis[N];
    void init() {
        tot = 1;
        memset(head, 0, sizeof head);
    }
    void ade(int u, int v, int f, int c) {
        e[++tot] = {v, head[u], f, c};
        head[u] = tot;
    }
    void add(int u, int v, int f, int c) {
        // c = -c;
        ade(u, v, f, c);
        ade(v, u, 0, -c);
    }
    bool spfa() {
        memset(dis, inf, sizeof dis);
        memset(vis, 0, sizeof vis);
        memset(level, 0, sizeof level);
        dis[s] = 0; level[s] = 1; vis[s] = 1;
        deque<int> Q;
        Q.push_back(s);
        while(!Q.empty()) {
            int x = Q.front();
            Q.pop_front();
            vis[x] = 0;
            for (int i = head[x]; i; i = e[i].nex) {
                int to = e[i].to;
                if(dis[to]>dis[x]+e[i].c && e[i].f>0) {
                    dis[to] = dis[x] + e[i].c;
                    level[to] = level[x] + 1;
                    if(!vis[to]) {
                        vis[to] = 1;
                        if(!Q.empty() && dis[to] < dis[Q.front()])
                            Q.push_front(to);
                        else
                            Q.push_back(to);
                    }
                }
            }
        }
        return dis[t] != inf;
    }
    int dfs(int x, int f) {
        if(x == t) {
            maxf += f;
            flag = true;
            return f;
        }
        int num = 0, flow = 0;
        for (int i = head[x]; i; i=e[i].nex) {
            int to = e[i].to;
            if(f == num) break;
            if(dis[x]+e[i].c==dis[to] && level[x]+1==level[to] && e[i].f>0) {
```

```
                flow = dfs(to, min(f-num, e[i].f));
                num += flow;
                cost += flow * e[i].c;

                e[i].f -= flow;
                e[i^1].f += flow;
            }
        }
        return num;
    }
    void mcmf() {
        maxf = cost = 0;
        while(spfa()) {
            flag = true;
            while(flag) {
                flag = false;
                dfs(s, inf);
            }
        }
    }
}
```

7.17.5 费用流

```
# define pi pair<int, int>
const int N = 5e3 + 5, M = 5e4 + 5;
const int inf = 1e9;

namespace MCMF {
    int Ecnt = 1, first[N], nex[M * 2], arr[M * 2], cap[M * 2], cost[M * 2];
    int dis[N], h[N], pree[N], prev[N], F, C;
    int n;
    template <typename T>
    inline void Min(T &a, T b) {
        if(a > b) a = b;
    }
    inline void Ad(int u, int v, int c, int w) {
        nex[++Ecnt] = first[u], first[u] = Ecnt, arr[Ecnt] = v, cap[Ecnt] = c, cost[Ecnt]
            = w;
    }
    inline void add(int u, int v, int c, int w) { // c:容量 w:费用
        Ad(u, v, c, w), Ad(v, u, 0, -w);
    }
    void init(int node_num) {
        n = node_num;
        F = C = 0;
        Ecnt = 1;
        memset(pree, 0, sizeof(pree));
        memset(prev, 0, sizeof(prev));
        memset(first, 0, sizeof(first));
    }
    void Dijkstra(int s) {
```

```
        static priority_queue<pi, vector<pi>, greater<pi> > q;
        for(; !q.empty(); q.pop());
        fill(dis, dis + 1 + n, -1);
        dis[s] = 0, q.push(pi(0, s));
        // printf("-----------\n");
        while(!q.empty()) {
            pi now = q.top(); q.pop();
            int u = now.second;
            if(dis[u] < now.first) continue;
            for(int i = first[u]; i; i = nex[i]) {
                static int v; v = arr[i];
                if(!cap[i]) continue;
                if(dis[v] < 0 || dis[v] > dis[u] + cost[i] + h[u] - h[v]) {
                    dis[v] = dis[u] + cost[i] + h[u] - h[v];
                    prev[v] = u, pree[v] = i;
                    q.push(pi(dis[v], v));
                }
            }
        }
    }
    pi solve(int s, int t) {
        fill(h, h + 1 + n, 0);
        for(int f = inf; f > 0; ) {
            Dijkstra(s);
            if(dis[t] < 0) break;
            for(register int i = 1; i <= n; ++i) // be careful this for
                h[i] += (dis[i] != -1) ? dis[i] : 0;
            int d = f;
            for(int u = t; u != s; u = prev[u])
                Min(d, cap[pree[u]]);
            f -= d, F += d, C += h[t] * d;
            assert(C >= 0);
            for(int u = t; u != s; u = prev[u]) {
                cap[pree[u]] -= d;
                cap[pree[u] ^ 1] += d;
            }
        } return pi(F, C);
    }
}
```

7.17.6 dinic

```
/*
1. 建图时初始化源点s、汇点t
2. 对于包含二分图最大匹配在内的单位网络，Dinic算法可以在O(m√n)的时间内求出其最大流。
*/

#define inf 0x7fffffff
#define maxn 25000
struct Edge{
    int from,to,cap,flow;
};
```

```cpp
class Dinic{
private:
    int s,t,c;
    vector<Edge>edges;
    vector<int>G[maxn];//结点
    bool vis[maxn];
    int dist[maxn];
    int cur[maxn];
public:
    int n,m;
    void AddEdge(int from,int to,int cap){
        edges.push_back((Edge){from,to,cap,0});
        edges.push_back((Edge){to,from,0,0});
        c=edges.size();
        G[from].push_back(c-2);
        G[to].push_back(c-1);
    }
    bool BFS(){
        queue<int>Q;
        memset(vis,0,sizeof(vis));
        Q.push(s);
        dist[s]=0;
        vis[s]=1;
        while(!Q.empty()){
            int x=Q.front();Q.pop();
            for(int i=0;i<G[x].size();i++){
                Edge& e=edges[G[x][i]];
                if(!vis[e.to]&&e.cap>e.flow){
                    vis[e.to]=1;
                    dist[e.to]=dist[x]+1;
                    Q.push(e.to);
                }
            }
        }
        return vis[t];
    }
    int DFS(int x,int a){
        if(x==t||a==0)return a;
        int flow=0,f;
        for(int& i=cur[x];i<G[x].size();i++){
            Edge& e=edges[G[x][i]];
            if(dist[x]+1==dist[e.to]&&(f=DFS(e.to,min(a,e.cap-e.flow)))>0){
                e.flow+=f;
                edges[G[x][i]^1].flow-=f;
                flow+=f;
                a-=f;
                if(a==0)break;
            }
        }
        return flow;
    }
    int Maxflow(int s,int t){
        this->s=s;this->t=t;
```

```
        int flow=0;
        while(BFS()){
            memset(cur,0,sizeof(cur));
            flow+=DFS(s,inf);
            flow+=DFS(s,inf);
        }
        return flow;
    }
    void init(){
        edges.clear();
        for(int i=0;i<maxn;i++){
            G[i].clear();
            dist[i]=0;
        }
    }
    vector<int> Mincut(){
        BFS();
        vector<int> ans;
        for(int i=0;i<edges.size();i++){
            Edge& e=edges[i];
            if(vis[e.from]&&!vis[e.to]&&e.cap>0)ans.push_back(i);
        }
        return ans;
    }
}Do;
```

7.18 最短路

7.18.1 Floyd

图的传递闭包

已知一个有向图中任意两点之间是否有连边，要求判断任意两点是否连通。

bitset 优化，复杂度可以到 $O(\frac{n^3}{w})$

最小环

给一个正权无向图，找一个最小权值和的环。

想一想这个环是怎么构，考虑环上编号最大的结点 u，$f[u-1][x][y]$ 和 (u,x), (u,y）共同构成了环。

在 Floyd 的过程中枚举 u，计算这个和的最小值即可。

时间复杂度为 $O(n^3)$。

```
/*
* 计算最短路径的基础算法，Floyd算法基础模板
*/
const int inf = 0x3f3f3f3f;
const int maxn = 1050;

int n, m;
int d[maxn][maxn];

```

```
void floyd() {
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (d[i][j] < inf && d[k][j] < inf)
                    d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
            }
        }
    }
}

/*
* 利用floyd可以求传递闭包，边权用1和0表示“连通”和“不连通”
*/

// std::bitset<SIZE> f[SIZE];
for (k = 1; k <= n; k++)
  for (i = 1; i <= n; i++)
    if (f[i][k]) f[i] = f[i] | f[k];
```

7.18.2 Bellman-Ford

应用

给一张有向图，问是否存在负权环。

做法很简单，跑 Bellman-Ford 算法，如果有个点被松弛成功了 n 次，那么就一定存在。

如果 $n-1$ 次之内算法结束了，就一定不存在。

在没有负权边时最好使用 Dijkstra 算法，在有负权边且题目中的图没有特殊性质时，若 SPFA 是标算的一部分，题目不应当给出 Bellman-Ford 算法无法通过的数据范围

```
/*
* Bellman-Ford 算法
* 一种基于松弛（relax）操作的最短路算法,支持负权。
*/

bool inq[510];
int dis[510],sumv[510];
int n,v[510*3],__next[510*3],e,w[510*3],first[510],cnts[510];
void AddEdge(int U,int V,int W) {
    v[++e]=V;
    w[e]=W;
    __next[e]=first[U];
    first[U]=e;
}

bool spfa(const int &s) {
    queue<int>Q;
    memset(dis,0x7f,sizeof(dis));
    dis[s]=0;
    Q.push(s);
    inq[s]=1;
    ++cnts[s];
```

```
    while(!Q.empty()) {
        int U=Q.front();
        for(int i=first[U]; i; i=__next[i])
            if(dis[v[i]]>dis[U]+w[i]) {
                dis[v[i]]=dis[U]+w[i];
                if(!inq[v[i]]) {
                    Q.push(v[i]);
                    inq[v[i]]=1;
                    ++cnts[v[i]];
                    if(cnts[v[i]]>n+1)
                        return 0;
                }
            }
        Q.pop();
        inq[U]=0;
    }
    return 1;
}
```

7.18.3 dijkstra

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N=100002;
const int M=200002;
const int inf=2147483647;
inline int read()
{
    char ch=getchar();
    int x=0;bool f=false;
    while (!isdigit(ch)) f^=!(ch^45),ch=getchar();
    while (isdigit(ch)) x=(x<<1)+(x<<3)+(ch^48),ch=getchar();
    if (f) x=-x;return x;
}
int n,m,s;
int head[N],to[M],nex[M],cnt=0;
ll val[M];
ll dis[N];
bool vis[N];
struct data
{
    int id;
    ll v;
    bool operator <(const data&A)const
    {
        return A.v<v;
    }
};
inline void add(int s,int ed,ll w)
{
    to[++cnt]=ed;
```

```
    val[cnt]=w;
    nex[cnt]=head[s];
    head[s]=cnt;
}
inline void Dijkstra(int st)
{
    priority_queue<data>q;
    for (int i=1;i<=n;++i)
    dis[i]=inf;
    dis[st]=0;
    q.push((data){st,dis[st]});
    while (!q.empty())
    {
        int now=q.top().id;
        q.pop();
        if (vis[now]) continue;
        vis[now]=true;
        for (int i=head[now];i;i=nex[i])
        {
            int v=to[i];
            if (!vis[v]&&dis[v]>dis[now]+val[i])
            {
                dis[v]=dis[now]+val[i];
                q.push((data){v,dis[v]});
            }
        }
    }
}
int main()
{
    n=read(),m=read(),s=read();
    int x,y,z;
    for (int i=1;i<=m;++i)
    {
        x=read(),y=read(),z=read();
        add(x,y,z);
    }
    Dijkstra(s);
    for (int i=1;i<=n;++i)
    printf("%lld ",dis[i]);
    return 0;
}
```

7.18.4 bfs 全源最短路径

```
/*
* 模版: bfs全源最短路径 (边权为1)
*/
#include <bits/stdc++.h>
using namespace std;

const int N = 2e3 + 5; // 节点个数
```

```
const int M = 1e5 + 5; // 边的个数
const int inf = 0x3f3f3f3f;

int head[N], to[M], nex[M], val[M], cnt; // 链式前向星
inline void add(int u, int v, int w) {
    to[++cnt] = v;
    val[cnt] = w;
    nex[cnt] = head[u];
    head[u] = cnt;
}

int Dis[N][N], n, m, q;
bool vis[N];

void bfs(int S) {
    int *dis = Dis[S];
    for (int i = 1; i <= n; i++) vis[i] = 0, dis[i] = inf;
    queue<int> Q;
    Q.push(S);
    vis[S] = 1;
    dis[S] = 0;
    while(!Q.empty()) {
        int now = Q.front();
        Q.pop();
        for (int i = head[now]; i; i = nex[i]) {
            int v = to[i];
            if(vis[v]) continue;
            vis[v] = 1;
            Q.push(v);
            dis[v] = dis[now] + 1;
        }
    }
}

const int Q = 1e6 + 5;
int c[Q], d[Q], t[Q];
int main() {
    freopen("ysys.in", "r", stdin);
    scanf("%d%d%d", &n, &m, &q);
    for (int i = 1; i <= m; i++) {
        int u, v;
        scanf("%d%d", &u, &v);
        add(u, v, 1);
        add(v, u, 1);
    }

    for (int i = 1; i <= n; i++) {
        bfs(i);
    }

    // for (int i = 1; i <= n; i++) {
    //  for (int j = 1; j <= n; j++) {
    //   printf("%d-%d:%d\n", i, j, Dis[i][j]);
```

```
    //  }
    // }
}
```

7.18.5 Johnson 全源最短路径算法

```
/*
* 求多源负权最短路时用，比floyd快，（ +2  ）
*/

const int N = 5007;

struct Edge {
    int nxt, pre, w, from;
} e[N << 1];
int head[N], cntEdge;
inline void add(int u, int v, int w) {
    e[++cntEdge] = (Edge){ head[u], v, w, u}, head[u] = cntEdge;
}

int DIS[N][N], H[N], vis[N];
int qq[N], h, t;
int n, m;
inline void SPFA(int st) {
    for(register int i = 1; i <= n; i += 3){
        H[i] = 0x3f3f3f3f;
        H[i + 1] = 0x3f3f3f3f;
        H[i + 2] = 0x3f3f3f3f;
    }
    H[st] = 0;
    qq[++t] = st;
    while(h != t){
        int u = qq[++h];
        if(h >= N - 5) h = 0;
        vis[u] = false;
        for(register int i = head[u]; i; i = e[i].nxt){
            int v = e[i].pre;
            if(H[v] > H[u] + e[i].w){
                H[v] = H[u] + e[i].w;
                if(!vis[v]){
                    vis[v] = true;
                    qq[++t] = v;
                    if(t >= N - 5) t = 0;
                }
            }
        }
    }
}

struct nod {
    int x, w;
    bool operator < (const nod &com) const {
```

```
        return w > com.w;
    }
};
#include <queue>
priority_queue<nod> q;
int dis[N];
inline void Dijkstra(int st) {
    for(register int i = 1; i <= n; i += 3){
        dis[i] = 0x3f3f3f3f;
        dis[i + 1] = 0x3f3f3f3f;
        dis[i + 2] = 0x3f3f3f3f;
    }
    dis[st] = 0;
    q.push((nod){ st, 0});
    while(!q.empty()){
        int u = q.top().x, w = q.top().w;
        q.pop();
        if(w != dis[u]) continue;
        for(register int i = head[u]; i; i = e[i].nxt){
            int v = e[i].pre;
            if(dis[v] > dis[u] + e[i].w){
                dis[v] = dis[u] + e[i].w;
                q.push((nod){ v, dis[v]});
            }
        }
    }
}

int main() {
    io >> n >> m;

    R(i,1,m){
        int u, v, w;
        io >> u >> v >> w;
        add(u, v, w);
    }

    R(i,1,n){
        add(0, i, 0);
    }

    SPFA(0);

    R(i,1,cntEdge){
        e[i].w += H[e[i].from] - H[e[i].pre];
    }

    R(i,1,n){
        Dijkstra(i);
        R(j,1,n){
            DIS[i][j] = dis[j] - H[i] + H[j];
        }
    }
```

```
    R(i,1,n){
        R(j,1,n){
            printf("%d ", DIS[i][j]);
        }
        putchar('\n');
    }

    return 0;
}
```

8 杂项

8.1 随机化

8.1.1 模拟退火

```
/*
* 题目：[JSOI2004]平衡点 / 吊打XXX
* URL: https://www.luogu.com.cn/problem/P1337
* 温度T的初始值设置问题：
* 1. 初始温度高，则搜索到全局最优解的可能性大，但因此要花费大量的计算时间；
* 2. 反之，则可节约计算时间，但全局搜索性能可能受到影响。
* 退火速度问题：
* 1. 模拟退火算法的全局搜索性能也与退火速度密切相关。同一温度下的“充分”搜索(退火)是相当必要的，但这需要计算时间。
* 温度管理问题：
* 1. 降温系数应为正的略小于1.00的常数。
* 注意：
* 1. 为了使得解更为精确，我们通常不直接取当前解作为答案，而是在退火过程中维护遇到的所有解的最优值。
* 2. 有时为了使得到的解更有质量，会在模拟退火结束后，以当前温度在得到的解附近多次随机状态，尝试得到更优的解（其过程与模拟退火相似）。
* 技巧：
* 1. 分块模拟退火，将值域分为几段，每段跑一遍模拟退火，再取最优解。
* 2. 卡时可以把主程序中的 simulateAnneal(); 换成 while ((double)clock()/CLOCKS_PER_SEC < MAX_TIME) simulateAnneal();。这样子就会一直跑模拟退火，直到用时即将超过时间限制。
*/
#include <bits/stdc++.h>
using namespace std;

const int N = 10005;
const double MAX_TIME = 4.5; // 这里的 MAX_TIME 是一个自定义的略小于时限的数。
int n, x[N], y[N], w[N];
double ansx, ansy, dis;

double Rand() { return (double)rand() / RAND_MAX; }
// 因为物重一定，绳子越短，重物越低，势能越小，势能又与物重成正比。
// 所以，只要使得sum{dist[i]*weight[i]}也就是总的重力势能最小，就可以使系统平衡。
double calc(double xx, double yy)
{
  double res = 0;
```

```
  for (int i = 1; i <= n; ++i)
  {
    double dx = x[i] - xx, dy = y[i] - yy;
    res += sqrt(dx * dx + dy * dy) * w[i];
  }
  if (res < dis)
    dis = res, ansx = xx, ansy = yy;
  return res;
}
void simulateAnneal()
{
  double t = 100000;
  double nowx = ansx, nowy = ansy;
  while (t > 0.001)
  {
    // 随机变化坐标，变化幅度为 T 。
    double nxtx = nowx + t * (Rand() * 2 - 1);
    double nxty = nowy + t * (Rand() * 2 - 1);
    // 计算新解与当前解的差 DE。
    double delta = calc(nxtx, nxty) - calc(nowx, nowy);
    // 如果新解比当前解优(DE > 0)，就用新解替换当前解。
    // 否则以 exp(DE / T) 的概率用新解替换当前解。
    if (exp(-delta / t) > Rand())
      nowx = nxtx, nowy = nxty;
    // 温度乘上一个小于1的系数，即降温。
    t *= 0.97;
  }
  for (int i = 1; i <= 1000; ++i)
  {
    double nxtx = ansx + t * (Rand() * 2 - 1);
    double nxty = ansy + t * (Rand() * 2 - 1);
    calc(nxtx, nxty);
  }
}
int main()
{
  srand(time(0));
  scanf("%d", &n);
  for (int i = 1; i <= n; ++i)
  {
    scanf("%d%d%d", &x[i], &y[i], &w[i]);
    ansx += x[i], ansy += y[i];
  }
  ansx /= n, ansy /= n, dis = calc(ansx, ansy);
  while ((double)clock()/CLOCKS_PER_SEC < MAX_TIME)
  simulateAnneal();
  printf("%.3lf %.3lf\n", ansx, ansy);
  return 0;
}
```