DJ算法：主要是这个小根堆的写法

struct compare{

    bool operator()(const pair<int,int>&a,const pair<int,int>&b){

        return a.second>b.second;

    }

};

priority\_queue<pair<int,int>,vector<pair<int,int>>,compare>heap;

void dijkstra(){

    for(int i=0;i<=n;i++){

        dis[i]=INF;

    }

    dis[0]=0;

    heap.push({0,0});

    while(!heap.empty()){

        int u=heap.top().first;

        heap.pop();

        if(vis[u]){

            continue;

        }

        vis[u]=1;

        if(u==2\*n+5){

            break;

        }

        for(int i=head[u];i;i=nxt[i]){

            int v=to[i];

            int w=weight[i];

            if(dis[v]>dis[u]+w){

                dis[v]=dis[u]+w;

                heap.push({v,dis[v]});

            }

        }

    }

}

Floyd算法：其实本质上也就是个动态规划

// Floyd算法模版（洛谷）

// 测试链接 : https://www.luogu.com.cn/problem/P2910

int n,m,ans;

int dis[MAXN][MAXN];

int path[MAXM];

void build() {

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < n; j++) {

            dis[i][j] = INT\_MAX;

        }

    }

}

void floyd() {

    // O(N^3)的过程

    // 枚举每个跳板

    // 注意，跳板要最先枚举！跳板要最先枚举！跳板要最先枚举！

    for (int bridge = 0; bridge < n; bridge++) { // 跳板

        for (int i = 0; i < n; i++) {

            for (int j = 0; j < n; j++) {

                // i -> .....bridge .... -> j

                // distance[i][j]能不能缩短

                // distance[i][j] = min ( distance[i][j] , distance[i][bridge] + distance[bridge][j])

                if (dis[i][bridge] != INT\_MAX

                        && dis[bridge][j] != INT\_MAX

                        && dis[i][j] > dis[i][bridge] + dis[bridge][j]) {

                    dis[i][j] = dis[i][bridge] + dis[bridge][j];

                }

            }

        }

    }

}

SPFA算法：可以处理负边权，也可以检测负权环，稀疏图

（n个点，那么最多n-1次；注意有时我们会加入一个源点，此时就是n次）

// 源点出发到每个节点的距离表

int dis[MAXN];

// 节点被松弛的次数

int updateCnt[MAXN];

bool enter[MAXN];//当前在队列中的话  就是true  否则就是false

queue<int >q; //每一轮都弹出最后面的一个  如果一个点可以被优化  那么加入队列

bool spfa() {

    dis[1] = 0;

    updateCnt[1]++;

    q.push(1) ;

    enter[1] = true;

    //这道题目中的源点都是1 保持不变   所以以上的操作都针对1

    //实际上应该理解为是针对源点的

    while (!q.empty()) {

        int u = q.front();

        q.pop();

        enter[u] = false;

        for (int ei = head[u], v, w; ei > 0; ei = Next[ei]) {

            v = to[ei];

            w = weight[ei];

            if (dis[u] + w < dis[v]) {

                //如果值变小了  那么就要改变

                dis[v] = dis[u] + w;

                if (!enter[v]) {

                    //如果不在队列中才会加入

                    // 松弛次数超过n-1就有负环

                    if (++updateCnt[v] > n - 1) {

                        return true;

                    }

                    q.push(v);

                    enter[v] = true;

                }

            }

        }

    }

    return false;

}

Tarjan——缩点

vector<int>vec[MAXN];

//dfn表示的就是dfn序  而low数组表示的是这个节点沿着边可以走到的最小的dfn序

//dfn一旦确定了 就不会改变了  而low数组可能会发生变化

int dfn[MAXN],low[MAXN];

int st[MAXN],top;

int color,Time,col[MAXN];

int in[MAXN],out[MAXN];

bool instack[MAXN];

int n,m;

void tarjan(int x){

    dfn[x]=++Time;

    low[x]=Time;

    st[++top]=x;

    instack[x]=true;

    for(int i=0;i<vec[x].size();i++){

        int v=vec[x][i];

        if(!dfn[v]){

            //表示这个节点没有被访问过

            tarjan(v);

            low[x]=min(low[x],low[v]);

        } else if(instack[v]){

            //表示这个属于是回溯了 一定是同一个环上的

            //这里之所以不用low[v],是因为这里的含义就是dfn序号,其实就算改成low也不影响

            //不过这里最好写成这样 因为其他的要求可能会导致写成low出错

            low[x]=min(low[x],dfn[v]);

        }

    }

    if(dfn[x]==low[x]){

        col[x]=++color;

        //将所有节点按照颜色分类  完成缩点

        while(st[top]!=x){

            //属于同一个强联通分量

            col[st[top]]=color;

            instack[st[top]]=false;

            top--;

        }

        instack[x]=false;

        top--;

    }

}

Tarjan——割点

// 我们判断割点的方法是：

//1.如果这个点是根节点  那么如果他有两个儿子  则他一定是割点

// 这里的两个儿子指的是在深度优先搜索的条件下的两个儿子 即这两个儿子的子树一定是互不相干的

// 否则其中一个儿子的子树一定会包括另外一个儿子

//2.如果这个节点不是根节点，并且不是叶子结点 并且low[他的任意一个孩子]>=dfn[他]则它是割点

// 如果是叶节点  那么不可能是割点

// 并且他的所有孩子不能到达它的上面  即他扼死了孩子向上的通道

vector<int>vec[MAXN];

//在割点的时候  dfn还是最早的序号 可是low却不再是这个节点可以到达的最小编号了

//和缩点的时候不同

int dfn[MAXN],low[MAXN];

int Time,n,m;

bool cut[MAXN],instack[MAXN];

// v:当前点 r：本次搜索树的root

void tarjan(int u,int r) {

    dfn[u] = low[u] = ++Time;

    int child = 0;

    for (int i = 0; i < vec[u].size(); i++) {

        int v = vec[u][i];

        if (!dfn[v]) {

            tarjan(v, r);

            low[u] = min(low[u], low[v]);

            if (low[v] >= dfn[u] && u != r)

                cut[u] = true;//不是根而且他的孩子无法跨越他回到祖先

            if (r == u){

                child++; //如果是搜索树的根，统计孩子数目

                //其实所有的节点都可以统计孩子的数量 但是非根节点统计后没有什么作用

            }

        }

        low[u] = min(low[u], dfn[v]);//这里要特别注意 不能变成low[v]

        // 举个例子 比如a-b-c-d-b  并且还存在b-e-a

        // 那么首先b是一个割点  可是如果改成low[v] 那么low[d]=dfn[a]  导致b无法成为一个割点

    }

    if (child >= 2 && u == r){

        cut[r] = true;//对应第一种情况

    }

}

Tarjan——桥

// 桥其实和tarjan没有什么本质上的区别  都是一样的

// 桥的判断方法是对于一条边  如果下面的点的low大于上面点的dfn  说明是桥

// 如果这条边是回溯的  那不算

int n,m;

vector<pair<int,int>>bridges;

int dfn[MAXN],low[MAXN];

int dfncnt;

void tarjan(int u,int id){

    //表示当前来到了点u  是通过真实编号id这条边来的

    dfn[u]=low[u]=++dfncnt;

    for(int i=head[u];i;i=nxt[i]){

        int v=to[i];

        int w=weight[i];

        if(!dfn[v]){

            tarjan(v,w);

            low[u]=min(low[u],low[v]);

            if(low[v]>dfn[u]){

                bridges.push\_back({u,v});

            }

        }

        else if(w!=id){

            //不能走回头路

            low[u]=min(low[u],dfn[v]);

        }

    }

}

拓扑排序

void topo(){

    for(int i=1;i<=n;i++){

        if(in[i]==0){

            q.push(i);

        }

    }

    while(!q.empty()){

        int u=q.front();

        q.pop();

        for(int i=head1[u];i;i=Next1[i]){

            int v=to1[i];

            in[v]--;

            len[v]=max(len[v],len[u]+weight1[i]);

            if(in[v]==0){

                q.push(v);

            }

        }

    }

}

扩展欧几里得（普通的欧几里得就是gcd）

// 同余方程

// 求关于x的同余方程 ax ≡ 1(mod b) 的最小正整数解

// 题目保证一定有解，也就是a和b互质

// 2 <= a、b <= 2 \* 10^9

// 测试链接 : https://www.luogu.com.cn/problem/P1082

#include<bits/stdc++.h>

using namespace std;

long long d, x, y, px, py;

void exgcd(long long a, long long b) {

    if (b == 0) {

        //表示已经求出了最大公约数d      x y分别是1 0

        d = a;//最大公约数设置为a

        x = 1;

        y = 0;

    } else {

        exgcd(b, a % b);

        //这个过程在向上的时候逐步更新

        //d=x'\*a'+y'\*b'

        // =x'\*b+y'\*(a-(a/b)\*b)

        // =y'\*a+(x'-(a/b)\*y')b

        // =x\*a+y\*b

        //x=y'   y=x'-(a/b)\*y'

        px = x;

        py = y;

        x = py;

        y = px - py \* (a / b);

    }

}

int main()

{

    long long a,b;

    cin>>a>>b;

    exgcd(a,b);

    cout<<((x % b + b) % b);

    return 0;

}

求格点公式：gcd(abs(x1 - x2), abs(y1 - y2)) + 1

中国剩余定理（CRT）、扩展中国剩余定理（exCRT）

// 给出n个同余方程，求满足同余方程的最小正数解x

// 一共n个同余方程，x ≡ ri(% mi) 1 <= n <= 10 0 <= ri、mi <= 10^5

// 所有mi一定互质 所有mi整体乘积 <= 10^18

long long m[MAXN],r[MAXN];

long long d, x, y, px, py;

long long multiply(long long a, long long b, long long mod) {

    // 既然是在%mod的意义下，那么a和b可以都转化成非负的

    // 本题不转化无所谓，但是其他题目可能需要转化

    // 尤其是b需要转化，否则while循环会跑不完

    a = (a % mod + mod) % mod;

    b = (b % mod + mod) % mod;

    long long ans = 0;

    while (b != 0) {

        if ((b & 1) != 0) {

            ans = (ans + a) % mod;

        }

        a = (a + a) % mod;

        b >>= 1;

    }

    return ans;

}

void exgcd(long long a, long long b) {

    if (b == 0) {

        d = a;

        x = 1;

        y = 0;

    } else {

        exgcd(b, a % b);

        px = x;

        py = y;

        x = py;

        y = px - py \* (a / b);

    }

}

// 中国剩余定理模版  只适用于模数互质的情况

long long crt(int n) {

    long long lcm = 1;

    for (int i = 1; i <= n; i++) {

        lcm = lcm \* m[i];

        //因为m是互质的  所以最小公倍数就是他们的累乘

    }

    long long ai, ci, ans = 0;

    for (int i = 1; i <= n; i++) {

        // ai = lcm / m[i]

        // 即 m1 \* m2 \*```\* mi-1 \* mi+1 \*```\* mn-1 \* mn

        ai = lcm / m[i];

        // ai逆元，在%m[i]意义下的逆元

        exgcd(ai, m[i]);

        // ci = (ri \*   ai \* ai逆元(即x)  ) % lcm

        //这里之所以要%lcm  是因为我们要求的是最小的符合要求的正整数  所有符合要求的可以表示为特解加上任意个lcm

        ci = multiply(r[i], multiply(ai, x, lcm), lcm);

        ans = (ans + ci) % lcm;

    }

    return ans;

}

// 扩展中国剩余定理模版 可以用于解决模数不是互质的情况  适用范围更广

long long excrt(int n) {

    long long tail = 0, lcm = 1, tmp, b, c, x0;

    // ans = lcm \* x + tail

    for (int i = 1; i <= n; i++) {

        // ans = m[i] \* y + ri

        // lcm \* x + m[i] \* y = ri - tail

        // a = lcm b = m[i] c = ri - tail

        b = m[i];

        c = ((r[i] - tail) % b + b) % b;

        exgcd(lcm, b);

        if (c % d != 0) {

            return -1;

        }

        // ax + by = gcd(a,b)，特解是，x变量

        // ax + by = c，特解是，x变量 \* (c/d)

        // ax + by = c，最小非负特解x0 = (x \* (c/d)) % (b/d) 取非负余数

        //特解是x0 通解 = x0 + (b/d) \* n

        x0 = multiply(x, c / d, b / d);

        // ans = lcm \* x + tail，带入通解

        // ans = lcm \* (x0 + (b/d) \* n) + tail

        // ans = lcm \* (b/d) \* n + lcm \* x0 + tail

        // tail' = tail' % lcm'

        tmp = lcm \* (b / d);

        tail = (tail + multiply(x0, lcm, tmp)) % tmp;

        lcm = tmp;

    }

    return tail;

}

质数筛

//埃氏筛 欧拉筛 埃氏筛很容易理解欧拉筛的扩展功能很强  可以记录每个数字的最小质因子

bool visit[MAXN];//用于埃氏筛统计是否是质数

int prime[MAXN];//欧拉筛中用于收集质数的数组

int num[MAXN];//记录每个数字的最小质因子

int ehrlich(int n) {

    // 初始时认为0~n所有数都是质数，但0和1不是质数

    for(int i=2;i\*i<=n;i++){

        if(!visit[i]){

            for(int j=i\*i;j<=n;j+=i){

                visit[j]=true;

            }

        }

    }

    int cnt=0;

    for(int i=2;i<=n;i++){

        if(!visit[i]){

            cnt++;

        }

    }

    return cnt;

}

int euler(int n){

    int cnt=0;

    for(int i=2;i<=n;i++){

        if(num[i]==0){

            prime[++cnt]=i;

        }

        //无论是不是质数  都要进行下面的过程

        for(int j=1;j<=cnt&&i\*prime[j]<=n;j++){

            num[i\*prime[j]]=prime[j];

            if(i%prime[j]==0){

                //如果i可以整除prime[j]说明一定含有这个质因子

                //那么如果继续的话  就是将接下来的数字的按照下一个质数作为他的最小质因子排除的 而不是被最小质因子排除的  所以不能继续  要立即跳出

                break; // 每个合数只被其最小的质因数筛去一次

            }

        }

    }

    return cnt;

}

逆元

//这个文件中以下三种逆元： 单个除数的逆元 连续数字的逆元 连续数字阶乘逆元

int n,m,mod;

int inv[MAXN];//连续数字的逆元

int fac[MAXN];//连续数字的阶乘

int facinv[MAXN];//连续数字阶乘的逆元

long long power(long long a,long long b,long long mod){

    long long ans=1;

    while(b){

        if(b&1){

            ans\*=a;

            ans%=mod;

        }

        b>>=1;a\*=a;a%=mod;

    }

    return ans;

}

//单个除数的逆元 费马小定理求(a/b)%mod的值

long long compute(long long a,long long b,long long mod){

    long long inv=power(b,mod-2,mod);//求b在%mod的情况下的逆元

    return (a\*inv)%mod;

}

//连续数字的逆元 预计算模逆元

void build(int n) {

    inv[1] = 1;

    for (int i = 2; i <= n; i++) {

        inv[i]=(mod - (long long)inv[mod % i] \* (mod / i) % mod + mod) % mod;

    }

}

//连续数字阶乘的逆元 初始化阶乘表和逆元表

void build() {

    //先求出阶乘表

    fac[0] = 1; // 0! = 1

    for (int i = 1; i <= n; i++) {

        fac[i] = (fac[i - 1] \* i) % mod;

    }

    // 利用线性递推优化计算逆元

    facinv[n] = power(fac[n], mod - 2,mod);//求出最后一个数字的阶乘逆元

    for (int i = n - 1; i >= 0; i--) {

        facinv[i] = (facinv[i + 1] \* (i + 1)) % mod;//线性递推

    }

}

差分（二维、等差数列）、二维前缀和

// 一开始1~n范围上的数字都是0，一共有m个操作，每次操作为(l,r,s,e,d)

// 表示在l~r范围上依次加上首项为s、末项为e、公差为d的数列

//等差数列差分需要两遍前缀和

int n,m; long long l,s,r,e,maxans,eor;

long long arr[10000005];

void Set(long long l,long long r,long long s,long long e,long long d){

    //以上参数分别表示  左起点  右终点  初始增加值  末尾增加值  公差

    arr[l]+=s;

    arr[l+1]+=(d-s);

    arr[r+1]-=(d+e);

    arr[r+2]+=e;

}

void build(){

    for(int i=1;i<=n;i++){

        arr[i]+=arr[i-1];

    }

    for(int i=1;i<=n;i++){

        arr[i]+=arr[i-1];

    }

}

二维差分

void Set(int a,int b,int c,int d,int v){

    nums[a][b]+=v;

    nums[a][d+1]-=v;

    nums[c+1][b]-=v;

    nums[c+1][d+1]+=v;

}

void build(){

    for(int i=1;i<=n;i++){

        for(int j=1;j<=n;j++){

            nums[i][j]+=nums[i-1][j]+nums[i][j-1]-nums[i-1][j-1];

        }

    }

}

二维前缀和

int fun(int X1,int Y1,int X2,int Y2){

    return sum[X2][Y2]+sum[X1-1][Y1-1]-sum[X2][Y1-1]-sum[X1-1][Y2];

}

博弈论（sg函数）

int sg[MAXN]; // 每个位置的 SG 值

bool appear[MAXV]; // 用于计算 SG 值的辅助数组

// 预处理 SG 值

void build() {

    for (int i = 1; i < MAXN; i++) {

        memset(appear, false, sizeof(appear)); // 重置 appear 数组

        for (int j = i - 1; j >= 0; j--) {

            for (int k = j; k >= 0; k--) {

                appear[sg[j] ^ sg[k]] = true; // 标记 SG 值的组合

            }

        }

        for (int s = 0; s < MAXV; s++) {

            if (!appear[s]) {

                sg[i] = s; // 找到最小的未出现的 SG 值

                break;

            }

        }

    }

}

倍增ST表、树上倍增（更常用）

// ST表查询最大值和最小值 给定一个长度为n的数组arr，一共有m次查询

// 每次查询arr[l~r]上的最大值和最小值

const int MAXN = 70005;

const int LIMIT = 16;

int stmax[MAXN][LIMIT],stmin[MAXN][LIMIT];

int arr[MAXN],Log2[MAXN],n,m;

void build(int n){

    Log2[0]=-1;

    for(int i=1;i<=n;i++){

        Log2[i]=Log2[i>>1]+1;

        stmax[i][0]=arr[i];

        stmin[i][0]=arr[i];

    }

    for(int i=1;i<=Log2[n];i++){

        for(int j=1;j<=n;j++){

            stmax[j][i]=max(stmax[j][i-1],stmax[j+(1<<(i-1))][i-1]);

            stmin[j][i]=min(stmin[j][i-1],stmin[j+(1<<(i-1))][i-1]);

        }

    }

}

int query(int l,int r){

    int p=Log2[r-l+1];

    int maxans=max(stmax[l][p],stmax[r-(1<<p)+1][p]);

    int minans=min(stmin[l][p],stmin[r-(1<<p)+1][p]);

    return maxans-minans;

}

树上倍增

void dfs(int u, int f) {

    deep[u] = deep[f] + 1;

    stjump[u][0] = f;

    for (int p = 1; p <= log\_2[n]; p++) {

        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];

    }

    //完成u的deep  stjump

    for (int e = head[u]; e != 0; e = Next[e]) {

        if (to[e] != f) {

            dfs(to[e], u); //向下递归

        }

    }

}

带权并查集

//寻找i节点的父亲 扁平化处理  并在该过程中更新距离

int find(int i) {

    if (i != fa[i]) {

        int tmp = fa[i];

        fa[i] = find(tmp);

        dist[i] += dist[tmp];//将距离更新正确

    }

    return fa[i];

}

void un(int l, int r, long long v) {

    int lf = find(l), rf = find(r);

    if (lf != rf) {

        fa[lf] = rf;//强制要求后面的节点作为父亲

        dist[lf] = v + dist[r] - dist[l];

    }

}

//加减关系：

//寻找i节点的父亲 扁平化处理  并在该过程中更新距离

int find(int i) {

    if (i != fa[i]) {

        int tmp = fa[i];

        fa[i] = find(tmp);

        dist[i] += dist[tmp];//将距离更新正确

    }

    return fa[i];

}

void un(int l, int r) {

    int lf = find(l), rf = find(r);

    if (lf != rf) {

        fa[lf] = rf;//强制要求后面的节点作为父亲

        dist[lf] += sz[rf];

        sz[rf] += sz[lf];

    }

}

//种类关系

int find(int i) {

    if (i != fa[i]) {

        int tmp = fa[i];

        fa[i] = find(tmp);

        dist[i] = (dist[i] + dist[tmp]) % 3;

        //更新关系比较特殊

    }

    return fa[i];

}

// op == 1, 1 l r，l和r是同类

// op == 2, 2 l r，l吃r

void un(int op, int l, int r) {

    int lf = find(l), rf = find(r), v = op == 1 ? 0 : 1;//判断二者的关系

    if (lf != rf) {

        fa[lf] = rf;

        dist[lf] = (dist[r] - dist[l] + v + 3) % 3;

    }

}

//异或关系

int find(int i){

    if(i!=fa[i]){

        int tmp=fa[i];

        fa[i]=find(tmp);

        dis[i]^=dis[tmp];

    }

    return fa[i];

}

可撤销并查集

//利用数组完成栈的功能

int rollback[MAXN][2];

int opsize = 0;//表示栈的大小

void undo() {

    //撤销一步操作

    int fx = rollback[opsize][0];

    int fy = rollback[opsize--][1];

    father[fy] = fy;

    siz[fx] -= siz[fy];

    edgeCnt[fx] -= edgeCnt[fy] + 1;

}

矩阵乘法

vector<vector<int>> multiply(vector<vector<int>>& a,const vector<vector<int>>& b) {

    int n = a.size();

    int m = b[0].size();

    int k = a[0].size();

    vector<vector<int>> ans(n, vector<int>(m, 0));

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < m; j++) {

            for (int c = 0; c < k; c++) {

                ans[i][j] += a[i][c] \* b[c][j];

                ans[i][j]=((ans[i][j]%MOD)+MOD)%MOD;

            }

        }

    }

    return ans;

}

// 矩阵快速幂

vector<vector<int>> power(vector<vector<int>>& m, int p) {

    int n = m.size();

    vector<vector<int>> ans(n, vector<int>(n, 0));

    for (int i = 0; i < n; i++) {

        ans[i][i] = 1;//单位矩阵 相当于乘法快速幂中的1

    }

    for (; p != 0; p >>= 1) {

        if ((p & 1) != 0) {

            ans = multiply(ans, m);

        }

        m = multiply(m, m);//每次都是倍增

    }

    return ans;

}

高精度

// 移除前导零

void removeLeadingZeros(vector<int>& num) {

    while (num.size() > 1 && num.back() == 0) {

        num.pop\_back();

    }

}

// 比较两个大整数的大小，返回1表示a>b，0表示a==b，-1表示a<b

int compare(const vector<int>& a, const vector<int>& b) {

    if (a.size() != b.size()) {

        return a.size() > b.size() ? 1 : -1;

    }

    for (int i = a.size() - 1; i >= 0; --i) {

        if (a[i] != b[i]) {

            return a[i] > b[i] ? 1 : -1;

        }

    }

    return 0;

}

// 大整数加法

vector<int> add(const vector<int>& a, const vector<int>& b) {

    vector<int> res;

    int carry = 0;

    for (int i = 0; i < (int)a.size() || i <(int) b.size() || carry; ++i) {

        if (i < (int)a.size()) carry += a[i];

        if (i < (int)b.size()) carry += b[i];

        res.push\_back(carry % 10);

        carry /= 10;

    }

    return res;

}

// 大整数减法 (假设a >= b)

vector<int> sub(const vector<int>& a, const vector<int>& b) {

    vector<int> res;

    int carry = 0;

    for (int i = 0; i <(int) a.size(); ++i) {

        carry = a[i] - carry;

        if (i < (int)b.size()) carry -= b[i];

        res.push\_back((carry + 10) % 10);

        carry = carry < 0 ? 1 : 0;//如果carry<0  说明要向上借位

    }

    removeLeadingZeros(res);

    return res;

}

// 大整数乘以小整数

vector<int> mul(const vector<int>& a, int b) {

    vector<int> res;

    int carry = 0;

    for (int i = 0; i <(int) a.size() || carry; ++i) {

        if (i <(int64\_t) a.size()) carry += a[i] \* b;

        res.push\_back(carry % 10);

        carry /= 10;

    }

    removeLeadingZeros(res);

    return res;

}

// 大整数乘以大整数

vector<int> mul(const vector<int>& a, const vector<int>& b) {

    vector<int> res(a.size() + b.size(), 0);

    for (int i = 0; i < a.size(); ++i) {

        for (int j = 0; j < b.size(); ++j) {

            res[i + j] += a[i] \* b[j];

            res[i + j + 1] += res[i + j] / 10;

            res[i + j] %= 10;

        }

    }

    removeLeadingZeros(res);

    return res;

}

// 大整数除以小整数，返回商和余数

pair<vector<int>, int> div(const vector<int>& a, int b) {

    vector<int> res;

    int remainder = 0;

    for (int i = a.size() - 1; i >= 0; --i) {

        remainder = remainder \* 10 + a[i];

        res.push\_back(remainder / b);

        remainder %= b;

    }

    reverse(res.begin(), res.end());

    removeLeadingZeros(res);

    return make\_pair(res, remainder);

}

// 大整数除以大整数 (简单实现，效率不高)

vector<int> div(const vector<int>& a, const vector<int>& b) {

    vector<int> res;

    vector<int> current;

    for (int i = a.size() - 1; i >= 0; --i) {

        current.insert(current.begin(), a[i]);

        removeLeadingZeros(current);

        int quotient = 0;

        while (compare(current, b) >= 0) {

            current = sub(current, b);

            quotient++;

        }

        res.push\_back(quotient);

    }

    reverse(res.begin(), res.end());

    removeLeadingZeros(res);

    return res;

}

// 字符串转大整数 这里有点问题  就是没有考虑负数

vector<int> strToNum(const string& s) {

    vector<int> num;

    for (int i =(int) s.size() - 1; i >= 0; --i) {

        if(s[i]>='0'&&s[i]<='9')

            num.push\_back(s[i] - '0');

    }

    removeLeadingZeros(num);

    return num;

}

// 大整数转字符串

string numToStr(const vector<int>& num) {

    string s;

    for (int i = num.size() - 1; i >= 0; --i) {

        s += to\_string(num[i]);

    }

    return s;

}

树上启发式合并

// 一共有n个节点，编号1~n，给定n-1条边，所有节点连成一棵树，1号节点为树头

// 每个节点给定一种颜色值，一共有m条查询，每条查询给定参数x

// 每条查询打印x为头的子树上，一共有多少种不同的颜色

// 树链剖分

int fa[MAXN];

int siz[MAXN];

int son[MAXN];

// 树上启发式合并 colorCnt[i] = j，表示i这种颜色出现了j次

int colorCnt[MAXN],ans[MAXN];

int diffColors = 0;

// 重链剖分

void dfs1(int u, int f) {

    fa[u] = f;

    siz[u] = 1;

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != f) {

            dfs1(v, u);

        }

    }

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != f) {

            siz[u] += siz[v];

            if (son[u] == 0 || siz[son[u]] < siz[v]) {

                son[u] = v;

            }

        }

    }

}

// 子树u每个节点贡献信息

void effect(int u) {

    if (++colorCnt[arr[u]] == 1) {

        diffColors++;

    }

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != fa[u]) {

            effect(v);

        }

    }

}

// 子树u每个节点取消贡献

void cancle(int u) {

    colorCnt[arr[u]] = 0; // 出现任何颜色，直接把该颜色的计数重置为0

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != fa[u]) {

            cancle(v);

        }

    }

}

// 树上启发式合并的过程

void dfs2(int u, int keep) {

    // 遍历轻儿子的子树，统计子树的答案，然后取消贡献

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != fa[u] && v != son[u]) {

            dfs2(v, 0);

        }

    }

    // 遍历重儿子的子树，统计子树的答案，然后保留贡献

    if (son[u] != 0) {

        dfs2(son[u], 1);

    }

    // 当前节点贡献信息

    if (++colorCnt[arr[u]] == 1) {

        diffColors++;

    }

    // 遍历轻儿子的子树，重新贡献一遍

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != fa[u] && v != son[u]) {

            effect(v);

        }

    }

    // 记录子树u的答案

    ans[u] = diffColors;

    // 如果u是上级节点的轻儿子，子树u的贡献取消，否则保留

    if (keep == 0) {

        diffColors = 0;

        cancle(u);

    }

}

树链剖分（点权）

int fa[MAXN];

int dep[MAXN];

int siz[MAXN];//子树大小

int son[MAXN];//重儿子节点编号

int top[MAXN];//所在重链头节点

int dfn[MAXN];//dfn序号

int seg[MAXN];//dfn序号对应的原始节点编号

int cntd = 0;

// 来到节点u，节点u树上的父节点是f dfs1的过程去设置 fa dep siz son

void dfs1(int u, int f) {

    fa[u] = f;

    dep[u] = dep[f] + 1;

    siz[u] = 1;

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != f) {

            dfs1(v, u);

        }

    }

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != f) {

            siz[u] += siz[v];

            if (son[u] == 0 || siz[son[u]] < siz[v]) {

                son[u] = v;

            }

        }

    }

}

// 来到节点u，节点u所在重链的头节点是t dfs2的过程去设置 top dfn seg

void dfs2(int u, int t) {

    top[u] = t;

    dfn[u] = ++cntd;

    seg[cntd] = u;

    if (son[u] == 0) {

        return;

    }

    dfs2(son[u], t);//去 重儿子继续向下

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != fa[u] && v != son[u]) {

            //轻儿子自己开一条重链 自己做头结点

            dfs2(v, v);

        }

    }

}

// 从x到y的路径上，所有节点的值增加v

void pathAdd(int x, int y, int v) {

    while (top[x] != top[y]) {

        if (dep[top[x]] <= dep[top[y]]) {

            add(dfn[top[y]], dfn[y], v, 1, n, 1);

            y = fa[top[y]];

        } else {

            add(dfn[top[x]], dfn[x], v, 1, n, 1);

            x = fa[top[x]];

        }

    }

    add(min(dfn[x], dfn[y]), max(dfn[x], dfn[y]), v, 1, n, 1);

}

// 从x到y的路径上，查询所有节点的累加和

long long pathSum(int x, int y) {

    long long ans = 0;

    while (top[x] != top[y]) {

        if (dep[top[x]] <= dep[top[y]]) {

            ans = (ans + query(dfn[top[y]], dfn[y], 1, n, 1)) % MOD;

            y = fa[top[y]];

        } else {

            ans = (ans + query(dfn[top[x]], dfn[x], 1, n, 1)) % MOD;

            x = fa[top[x]];

        }

    }

    ans = (ans + query(min(dfn[x], dfn[y]), max(dfn[x], dfn[y]), 1, n, 1)) % MOD;

    return ans;

}

树链剖分（边权）

int fa[MAXN];

int dep[MAXN];

int siz[MAXN];

int son[MAXN];

int top[MAXN];

int dfn[MAXN];

int cntd = 0;

void dfs1(int u, int f) {

    fa[u] = f;

    dep[u] = dep[f] + 1;

    siz[u] = 1;

    for (int e = head[u]; e > 0; e = nxt[e]) {

        int v = to[e];

        if (v != f) {

            dfs1(v, u);

        }

    }

    for (int e = head[u]; e > 0; e = nxt[e]) {

        int v = to[e];

        if (v != f) {

            siz[u] += siz[v];

            if (son[u] == 0 || siz[son[u]] < siz[v]) {

                son[u] = v;

            }

        }

    }

}

void dfs2(int u, int t) {

    top[u] = t;

    dfn[u] = ++cntd;

    if (son[u] == 0) {

        return;

    }

    dfs2(son[u], t);

    for (int e = head[u]; e > 0; e = nxt[e]) {

        int v = to[e];

        if (v != fa[u] && v != son[u]) {

            dfs2(v, v);

        }

    }

}

//更新一条边权  就是更新一个点的点权

void edgeUpdate(int ei, int val) {

    int x = arr[ei][0];

    int y = arr[ei][1];

    int downx = max(dfn[x], dfn[y]);

    update(downx, val, 1, n, 1);

}

//求x到y的之间的路径权值和

int pathSum(int x, int y) {

    int ans = 0;

    while (top[x] != top[y]) {

        if (dep[top[x]] <= dep[top[y]]) {

            ans += querySum(dfn[top[y]], dfn[y], 1, n, 1);

            y = fa[top[y]];

        } else {

            ans += querySum(dfn[top[x]], dfn[x], 1, n, 1);

            x = fa[top[x]];

        }

    }

    //lca不能动

    ans += querySum(min(dfn[x], dfn[y]) + 1, max(dfn[x], dfn[y]), 1, n, 1);

    return ans;

}

//求x到y的之间的路径权值最大值

int pathMax(int x, int y) {

    int ans = INT\_MIN;

    while (top[x] != top[y]) {

        if (dep[top[x]] <= dep[top[y]]) {

            ans = max(ans, queryMax(dfn[top[y]], dfn[y], 1, n, 1));

            y = fa[top[y]];

        } else {

            ans = max(ans, queryMax(dfn[top[x]], dfn[x], 1, n, 1));

            x = fa[top[x]];

        }

    }

    //lca不能动

    ans = max(ans, queryMax(min(dfn[x], dfn[y]) + 1, max(dfn[x], dfn[y]), 1, n, 1));

    return ans;

}

长链剖分

int stjump[MAXN][MAXH];//倍增表

int dep[MAXN];//深度表

int len[MAXN];//长度表

int son[MAXN];//长儿子

int top[MAXN];

int dfn[MAXN];

int cntd = 0;

int high[MAXN];//high[i]记录的是i的二进制最高位对应的数字是多少

int up[MAXN];//记录节点i向上走1 2 3 到达那个节点  只有长链的头结点才可以有

//并且只会填长儿子大小的个数   即使向上走还存在更远的  也不填了

int down[MAXN];//这个和上面一样  记录的是向下走几步

//设置u这个节点向上走i步 达到v节点  u一定是某条长链的头结点

void setUp(int u, int i, int v) {

    up[dfn[u] + i] = v;

}

//查询u节点向上走i步到达那个节点  u一定是某条长链的头结点

int getUp(int u, int i) {

    return up[dfn[u] + i];

}

//设置u这个节点向下走i步 达到v节点  u一定是某条长链的头结点

void setDown(int u, int i, int v) {

    down[dfn[u] + i] = v;

}

//查询u节点向下走i步到达那个节点  u一定是某条长链的头结点

int getDown(int u, int i) {

    return down[dfn[u] + i];

}

void dfs1(int u, int f) {

    stjump[u][0] = f;//没有fa数组 因为倍增表可以直接查询

    for (int p = 1; p < MAXH; p++) {

        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];

    }

    dep[u] = dep[f] + 1;

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != f) {

            dfs1(v, u);

        }

    }

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != f) {

            if (son[u] == 0 || len[son[u]] < len[v]) {

                son[u] = v;

            }

        }

    }

    len[u] = len[son[u]] + 1;//每个节点的大小是自己长儿子加一

}

void dfs2(int u, int t) {

    top[u] = t;

    dfn[u] = ++cntd;

    if (son[u] == 0) return;

    dfs2(son[u], t);

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != stjump[u][0] && v != son[u]) {

            dfs2(v, v);

        }

    }

}

//查询x节点向上走k步能到哪里

int query(int x, int k) {

    if (k == 0) {

        //如果是0步  那么直接就是这个节点

        return x;

    }

    if (k == (1 << high[k])) {

        //如果这个k正好是2的某次方  那么直接st表查询即可

        return stjump[x][high[k]];

    }

    //否则  我们先将其最高位的1用st表查询  找到一个中间节点

    x = stjump[x][high[k]];

    //然后k减去它的最高位

    k -= (1 << high[k]);

    //k剩下的这部分就通过来到的中间节点和他的头结点查询

    k -= dep[x] - dep[top[x]];

    x = top[x];

    return (k >= 0) ? getUp(x, k) : getDown(x, -k);

    //如果k小于0  那么说明根本到不了top[x]  那么向下走一点就好

    //如果k大于0  那么说明top[x]也不够 那么向上走一点就好

    //而且可以肯定的是  top[x]一定可以解决

    //因为既然初始节点走了k的最高位的距离来到了中间节点   至少走了k/2

    //那么说明这个中间节点的头结点的深度至少要大于k/2

}

哈希

const int MAXN = 10001;

const int BASE = 499;

long long power[MAXN];

long long hash\_val[MAXN];

void build(const string& str) {

    power[0] = 1;

    for (int j = 1; j < MAXN; ++j) {

        power[j] = power[j - 1] \* BASE;

    }

    hash\_val[0] = str[0] - 'a' + 1;

    for (int j = 1; j < str.length(); ++j) {

        hash\_val[j] = hash\_val[j - 1] \* BASE + str[j] - 'a' + 1;

    }

}

// 范围是s[l,r]，左闭右开

long long getHash(int l, int r) {

    long long ans = hash\_val[r];

    if (l > 0) {

        ans -= hash\_val[l - 1] \* power[r – l + 1];

    }

    return ans;

}

// 计算一个字符串的哈希值

long long hashString(const string& str) {

    if (str.empty()) {

        return 0;

    }

    int n = str.length();

    long long ans = str[0] - 'a' + 1;

    for (int j = 1; j < n; ++j) {

        ans = ans \* BASE + str[j] - 'a' + 1;

    }

    return ans;

}

KMP

char s1[MAXN],s2[MAXN];int Next[MAXN];

void nextArray(const char \*s, int m) {

    if (m == 1) {

        Next[0] = -1;

        return ;

    }

    Next[0] = -1,Next[1] = 0;

    int i = 2, cn = 0;

    // i表示当前要求next值的位置 cn表示当前要和前一个字符比对的下标

    while (i < m) {

        if (s[i - 1] == s[cn]) {

            Next[i] = ++cn;

            if(s[Next[i]]==s[i]){

                Next[i]=Next[Next[i]];

            }i++;

        } else if (cn > 0) {

            cn = Next[cn];

        } else {

            Next[i] = 0;

            if(s[Next[i]]==s[i]){

                Next[i]=Next[Next[i]];

            }

            i++;

        }

    }

}

int kmp(const char \*s1, const char \*s2) {

    int n = strlen(s1), m = strlen(s2), x = 0, y = 0;

    nextArray(s2, m);

    // s1中当前比对的位置是x s2中当前比对的位置是y

    while (x < n && y < m) {

        if (s1[x] == s2[y]) {

            x++;y++;

        } else if (y == 0) {

            x++;//表示s2已经在第一个位置了

        } else {

            y = Next[y];//表示s2往前跳

        }

    }

    return y == m ? x - y : -1;

}

Lucas定理

long long comb(long long n,long long m){

    if(n<m){

        return 0;

    }

    m=min(m,n-m);

    long long a=1,b=1;

    for(int i=1;i<=m;i++){

        a=(a\*(n-i+1))%p;

        b=(b\*i)%p;

    }

    return a\*power(b,p-2)%p;

}

long long lucas(int n,int m){

    if(m==0){

        return 1;

    }

    else{

        return lucas(n/p,m/p)\*comb(n%p,m%p)%p;

    }

}

BSGS定理

int bsgs(int g, int h, int mod) {

    // 处理特殊情况

    h%=mod;

    g%=mod;

    if(g==0){

        if (h == 0) return 1;  // 0^1 = 0 (x=1)

        if (h == 1) return 0;  // 0^0 = 1 (x=0)

        return -1;             // 无解

    }

    if (h == 1 ) return 0;

    if (h == 0 ) return -1;  // h=0 无解

    int t = ceil(sqrt(mod));

    map<int, int> mp;

    int base = h;

    // 小步：存储 h \* g^j (j=0 到 t-1)

    for (int j = 0; j < t; j++) {

        mp[base] = j;  // 直接覆盖，保留最大j

        base = (base \* g) % mod;

    }

    // 计算大步参数

    int tmp = power(g, t);  // g^t

    int cur = 1;  // 当前大步值 (g^{i\*t})

    // 大步：i 从 0 到 t

    for (int i = 0; i <= t; i++) {

        if (mp.find(cur) != mp.end()) {

            int x = i \* t - mp[cur];

            // 调整解到[0, mod-2]范围内

            //费马小定理：a^(p−1)≡1(modp)(对于任意 a!≡0)

            //这意味着指数x在模(p−1)意义下具有周期性：

            //y^x≡y^(x%(p−1))(mod p)

            x = (x % (mod-1) + (mod-1)) % (mod-1);

            return x;

        }

        cur = (cur \* tmp) % mod;  // 下一个大步

    }

    return -1;

}

欧拉函数

欧拉降幂公式（求2^3^4^…^2023数塔）

int geteuler(int tmp){

    int n=1;

    for(int i=2;i\*i<=tmp;i++){

        if(tmp%i){

            continue;

        }

        n\*=(i-1);

        tmp/=i;

        while(tmp%i==0){

            tmp/=i;

            n\*=i;

        }

    }

    if(tmp>1){

        n\*=(tmp-1);

    }

    return n;

}

int compute(int u,int mod){

    if(u==2023){

        return u;

    }

    return power(u,compute(u+1,geteuler(mod))+geteuler(mod),mod);

}

卡特兰数

long long c(int n, int k) {

    return (((fac[n] \* inv1[k]) % MOD) \* inv1[n - k]) % MOD;

}

// 公式1

long long compute1(int n) {

    build1(2 \* n);

    return (c(2 \* n, n) - c(2 \* n, n - 1) + MOD) % MOD;

}

// 公式2

long long compute2(int n) {

    build1(2 \* n);

    return c(2 \* n, n) \* power(n + 1, MOD - 2) % MOD;

}

// 公式3

long long compute3(int n) {

    build2(n);

    long long f[n + 1];

    f[0] = f[1] = 1;

    for (int i = 2; i <= n; i++) {

        f[i] = f[i - 1] \* (4 \* i - 2) % MOD \* inv2[i + 1] % MOD;

    }

    return f[n];

}

// 公式4

long long compute4(int n) {

    long long f[n + 1];

    for(int i=0;i<=n;i++){

        f[i]=0;

    }

    f[0] = f[1] = 1;

    for (int i = 2; i <= n; i++) {

        for (int l = 0, r = i - 1; l < i; l++, r--) {

            f[i] = (f[i] + f[l] \* f[r] % MOD) % MOD;

        }

    }

    return f[n];

}

康托展开

void add(int i,int v){

    while(i<=n){

        tree[i]+=v;i+=lowbit(i);

    }

}

int sum(int i){

    int ans=0;

    while(i>0){

        ans+=tree[i];

        i-=lowbit(i);

    }

    return ans;

}

int query(int l,int r){

    return sum(r)-sum(l-1);

}

signed main(){

    //将树状数组初始化

    for(int i=1;i<=n;i++){

        add(i,1);

    }

    long long ans=0;

    for(int i=1;i<=n;i++){

        ans=(ans+(fac[n-i]\*sum(arr[i]-1))%MOD)%MOD;

        add(arr[i],-1);//将这一位的信息去除

    }

    ans=(ans+1)%MOD;

    cout<<ans<<endl;

    return 0;

}

逆过程

// 初始化线段树，单点范围的初始累加和为1，认为所有数字都可用

void build(int l, int r, int i) {

    if (l == r) {

        sum[i] = 1;

    } else {

        int mid = (l + r) >> 1;

        build(l, mid, i << 1);build(mid + 1, r, i << 1 | 1);

        sum[i] = sum[i << 1] + sum[i << 1 | 1];

    }

}

// 单点jobi上，增加jobv，因为是单点更新，所以不需要建立懒更新机制

void add(int jobi, int jobv, int l, int r, int i) {

    if (l == r) {

        sum[i] += jobv;

    } else {

        int mid = (l + r) >> 1;

        if (jobi <= mid) {

            add(jobi, jobv, l, mid, i << 1);

        } else {

            add(jobi, jobv, mid + 1, r, i << 1 | 1);

        }

        sum[i] = sum[i << 1] + sum[i << 1 | 1];

    }

}

// 查询jobl~jobr范围的累加和

int Sum(int jobl, int jobr, int l, int r, int i) {

    if (jobl <= l && r <= jobr) {

        return sum[i];

    }

    int mid = (l + r) >> 1;

    int ans = 0;

    if (jobl <= mid) {

        ans += Sum(jobl, jobr, l, mid, i << 1);

    }

    if (jobr > mid) {

        ans += Sum(jobl, jobr, mid + 1, r, i << 1 | 1);

    }

    return ans;

}

// 线段树上找到第k名的是什么，找到后删掉词频，返回的过程修改累加和

// 注意这个排名是在子树中的排名

int getAndDelete(int k, int l, int r, int i) {

    int ans;

    if (l == r) {

        //找到目标 删除词频

        sum[i]--;

        ans = l;

    } else {

        int mid = (l + r) >> 1;

        if (sum[i << 1] >= k) {

            ans = getAndDelete(k, l, mid, i << 1);

        } else {

            ans = getAndDelete(k - sum[i << 1], mid + 1, r, i << 1 | 1);

            //要减去左侧排名的影响

        }

        sum[i] = sum[i << 1] + sum[i << 1 | 1];

        //返回的过程修改累加和

    }

    return ans;

}

void compute() {

    build(1, n, 1);

    // 当前排列转化为阶乘进制的排名

    for (int i = 1, x; i <= n; i++) {

        x = (int) arr[i];

        if (x == 1) {

            arr[i] = 0;

        } else {

            arr[i] = Sum(1, x - 1, 1, n, 1);

        }

        //arr数组表示一开始的排列在阶乘进制下的排名

        add(x, -1, 1, n, 1);

        //消除这个数字的影响

    }

    // 当前排名加上m之后，得到新的排名，用阶乘进制表示

    arr[n] += m; // 最低位获得增加的幅度

    for (int i = n; i >= 1; i--) {

        // 往上进位多少

        arr[i - 1] += arr[i] / (n - i + 1);

        // 当前位是多少

        arr[i] %= n - i + 1;

    }

    //到这里时  arr数组里面是加上m后的排名在阶乘进制下的表示

    // 根据阶乘进制转化为具体的排列

    build(1, n, 1);

    //将线段树重新复原

    for (int i = 1; i <= n; i++) {

        arr[i] = getAndDelete((int) arr[i] + 1, 1, n, 1);

        //此时这个arr数组用来存放排列

    }

}

双向广搜

//这种才是双向广搜的本体用法 即必须要全部展开 但是时间复杂度不允许  eg:2^40(10^12)

//所以我们将左右两边分别展开  最后将左右两边信息合并起来  得到汇总信息

long arr[MAXN];

long lsum[MAXM];//左边展开的信息

long rsum[MAXM];//右边展开的信息

int n;

long w;

//暴力展开 从i开始到e结束  不要超过w  因为超过w没有任何意义

//目前已经有了j个答案  暂时总和是s 递归函数  会一直递归到i==e

//返回值是自己可以得到的最终答案

int f(int i, int e, long s, long w, long\* ans, int j) {

    if (s > w) {

        return j;

    }

    // s <= w

    if (i == e) {

        //表示到了可以选择的最后一个物品了  记录可行答案的值 填入相应的答案记录数组中

        ans[j++] = s;

    } else {

        j = f(i + 1, e, s, w, ans, j); // 不要arr[i]位置的数

        j = f(i + 1, e, s + arr[i], w, ans, j); // 要arr[i]位置的数

    }

    return j;

}

long compute(){

    int lsize = f(0, n >> 1, 0, w, lsum, 0);

    int rsize = f(n >> 1, n, 0, w, rsum, 0);

    sort(lsum, lsum+lsize);

    sort(rsum, rsum+rsize);

    long ans = 0;

    //汇总左右两边信息 双指针

    for (int i = lsize - 1, j = 0; i >= 0; i--) {

        while (j < rsize && lsum[i] + rsum[j] <= w) {

            j++;

        }

        ans += j;

    }

    return ans;

}

高斯消元

加法模板

void swap(int a, int b) {

    //并且手写的时候一定要注意列的范围  不能缺少

    double tmp[MAXN + 1];

    for (int j = 0; j <= n+1; j++) {

        tmp[j] = mat[a][j];

        mat[a][j] = mat[b][j];

        mat[b][j] = tmp[j];

    }

}

//这个函数的作用是求出mat矩阵的解  n是x1 x2    x 如果无解返回0  有解返回1

int gauss(int n) {

    for (int i = 1; i <= n; i++) {

        //每次求解一个xi  也不是最终的值

        //只是将其他式子中的xi的系数消除为0

        //只保留自己的系数  最终将系数化为1

        int max = i;

        for (int j = 1; j <= n; j++) {

            if (j < i && abs(mat[j][j]) >= sml) {

                continue;

            }

            if (abs(mat[j][i]) > abs(mat[max][i])) {

                max = j;

            }

        }

        swap(i, max);//交换改行最大值

        if (abs(mat[i][i]) < sml) {

            //表示xi为0  那么不可能有解了

            return 0;

        }

        if (abs(mat[i][i]) >= sml) {

            //消去其他式子中的xi

            double tmp = mat[i][i];

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

                mat[i][j] /= tmp;

            }

            for (int j = 1; j <= n; j++) {

                if (i != j) {

                    double rate = mat[j][i] / mat[i][i];

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

                        mat[j][k] -= mat[i][k] \* rate;

                    }

                }

            }

        }

    }

    return 1;

}

异或模板

// 高斯消元解决异或方程组模版 需要保证变量有n个，表达式也有n个

void gauss(int n) {

    for (int i = 1; i <= n; i++) {

        for (int j = 1; j <= n; j++) {

            if (j < i && mat[j][j] == 1) {

                continue;

            }

            if (mat[j][i] == 1) {

                swap(mat[i], mat[j]);

                break;

            }

        }

        if (mat[i][i] == 1) {

            for (int j = 1; j <= n; j++) {

                if (i != j && mat[j][i] == 1) {

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

                        mat[j][k] ^= mat[i][k];

                    }

                }

            }

        }

    }

}

同余模板

// 高斯消元解决同余方程组模版，保证初始系数没有负数

void gauss(int n) {

    for (int i = 1; i <= n; i++) {

        for (int j = 1; j <= n; j++) {

            if (j < i && mat[j][j] != 0) {

                continue; // 已经成为主元的行不参与

            }

            if (mat[j][i] != 0) {

                swap(i,j); // 找到系数不等于0的行做主元即可

                break;

            }

        }

        if (mat[i][i] != 0) {

            for (int j = 1; j <= n; j++) {

                if (i != j && mat[j][i] != 0) {

                    int gcd\_val = gcd(mat[j][i], mat[i][i]);

                    int a = mat[i][i] / gcd\_val;

                    int b = mat[j][i] / gcd\_val;

                    if (j < i && mat[j][j] != 0) {

                        // 如果j行有主元，那么从j列到i-1列的所有系数 \* a

                        // 正确更新主元和自由元之间的关系

                        for (int k = j; k < i; k++) {

                            mat[j][k] = (mat[j][k] \* a) % MOD;

                        }

                    }

                    // 正常消元

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

                        mat[j][k] = ((mat[j][k] \* a - mat[i][k] \* b) % MOD + MOD) % MOD;

                    }

                }

            }

        }

    }

    for (int i = 1; i <= n; i++) {

        if (mat[i][i] != 0) {

            // 检查当前主元是否被若干自由元影响

            // 如果当前主元不受自由元影响，那么可以确定当前主元的值

            // 否则保留这种影响，正确显示主元和自由元的关系

            bool flag = false;

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

                if (mat[i][j] != 0) {

                    flag = true;break;

                }

            }

            if (!flag) {

                // 在模意义下应该求逆元，(a / b) % MOD = (a \* b的逆元) % MOD

                mat[i][n + 1] = (mat[i][n + 1] \* inv[mat[i][i]]) % MOD;

                mat[i][i] = 1;

            }

        }

    }

}

线性基

异或高斯消元法

void swap(int a, int b) {

    long long tmp = arr[a];

    arr[a] = arr[b];

    arr[b] = tmp;

}

//这个是高斯消元法  得到的是标准线性基

void compute() {

    len = 1;

    for (int i = BIT; i >= 0; i--) {

        for (int j = len; j <= n; j++) {

            if (arr[j] & (1LL << i)) {

                swap(j, len);//这里是对的 改成swap(arr[j], arr[len]);也可以通过

                break;

            }

        }

        if (arr[len] & (1LL << i)) {

            //表示找到了这一位上是1的数字

            for (int j = 1; j <= n; j++) {

                //将所有这一位置上是1的数字全部异或掉

                if (j != len && (arr[j] & (1LL << i))) {

                    arr[j] ^= arr[len];

                }

            }

            len++;

        }

    }

    len--;//表示有len位可以确定

    zero = len != n;

}

// 返回第k小的异或和

long query(long k) {

    if (zero && k == 1) {

        //如果有0  并且查询第一小的数字

        //那么直接返回0就好了

        return 0;

    }

    if (zero) {

        //如果有0  那么k就要减1

        //因为我们统计的是异或后不产生0的线性基

        k--;

    }

    if (k >= 1L << len) {

        //超出范围

        return -1;

    }

    long ans = 0;

    for (int i = len, j = 0; i >= 1; i--, j++) {

        if ((k & (1L << j)) != 0) {

            ans ^= arr[i];

        }

    }

    return ans;

}

异或普通消元法

// 线性基里插入num，如果线性基增加了返回true，否则返回false

bool insert(long num) {

    for (int i = BIT; i >= 0; i--) {

        if (num >> i == 1) {

            if (basis[i] == 0) {

                basis[i] = num;

                return true;

            }

            num ^= basis[i];

        }

    }

    return false;

}

// 普通消元

// 计算最大异或和

long compute() {

    for (int i = 1; i <= n; i++) {

        insert(arr[i]);

    }

    long ans = 0;

    for (int i = BIT; i >= 0; i--) {

        ans = max(ans, ans ^ basis[i]);

    }

    //如果可以使答案增加  那么就异或上这个数字

    return ans;

}

向量线性基

//向量线性基模版

bool insert(int i) {

    for (int j = 1; j <= m; j++) {

        if (fabs(mat[i][j]) >= sml) {

            if (basis[j] == 0) {

                basis[j] = i;

                return true;

            }

            double rate = mat[i][j] / mat[basis[j]][j];

            for (int k = j; k <= m; k++) {

                mat[i][k] -= rate \* mat[basis[j]][k];

            }

        }

    }

    return false;

}

void compute() {

    cnt = cost = 0;//记录答案

    vector<pair<double, int>> prices;

    for (int i = 1; i <= n; i++) {

        prices.emplace\_back(mat[i][m + 1], i);

    }

    sort(prices.begin(), prices.end());

    //根据价格排序  价格小的优先

    for (const auto& p : prices) {

        int i = p.second;

        if (insert(i)) {

            //如果这个物品可以插入 那么就插入

            cnt++;

            cost += static\_cast<int>(mat[i][m + 1]);

        }

    }

}

前缀树

int cnt=1;//用于赋值新编号   全部从1开始使用

int tree[MAXN][26];//表示最多只有26个字符

//tree[i][j]:表示编号为i的节点通过j（代表一个字符）这条路径到达下一个节点

//如果这个位置上是0  表示没有这条路

int en[MAXN];

int pass[MAXN];

//end  pass表示这个编号上的经过信息和结尾信息

int path;

void insert(string word){

    int cur=1;

    pass[cur]++;

    for (char ch : word){

        path=ch-'a';

        if(tree[cur][path]==0){

            tree[cur][path]=++cnt;

            //没有路就要新建一条路  并赋予编号

        }

        cur=tree[cur][path];

        pass[cur]++;

    }

    en[cur]++;

}

int search(string word){

    int cur=1;

    for (char ch : word){

        path=ch-'a';

        if(tree[cur][path]==0){

            return 0;

            //没有路说明走不通

        }

        cur=tree[cur][path];

    }

    return en[cur];

}

int preword(string word){

    int cur=1;

    for (char ch : word){

        path=ch-'a';

        if(tree[cur][path]==0){

            return 0;

            //没有路说明走不通

        }

        cur=tree[cur][path];

    }

    return pass[cur];

}

void erase(string word){

    if(search(word)>0){

        int cur=1;

        for (char ch : word){

            path=ch-'a';

            if(--pass[tree[cur][path]]==0){

                tree[cur][path]=0;

                //将这个节点设置为0  那么这个节点所连着的路全部失效

                //因为即使后来有结点重新连接上  但是会赋予这个新节点另外一个编号

                return ;

            }

            cur=tree[cur][path];

        }

        en[cur]--;

    }

}

void clear(){

    for(int i=1;i<=cnt;i++){

        pass[i]=0;

        en[i]=0;

        for(int j=0;j<26;j++){

            tree[i][j]=0;

        }

    }

}

可持久化前缀树（01trie）

// 最大异或和，java版

// 非负序列arr的初始长度为n，一共有m条操作，每条操作是如下两种类型中的一种

// A x     : arr的末尾增加数字x，arr的长度n也增加1

// Q l r x : l~r这些位置中，选一个位置p，现在希望

//           arr[p] ^ arr[p+1] ^ .. ^ arr[n] ^ x 这个值最大

//           打印这个最大值

// 1 <= n、m <= 3 \* 10^5

// 0 <= arr[i]、x <= 10^7

// 因为练的就是可持久化前缀树，所以就用在线算法，不要使用离线算法

const int MAXN = 600001;

const int MAXT = MAXN \* 22;

const int BIT = 25;

int n, m, eor;

int root[MAXN];

int tree[MAXT][2];//01trie只有两条路可以走 一条是0 一条是1

int pass[MAXT];//记录这个节点走过了几次

int cnt = 0;

//当前走到了i节点  插入了num  更新并返回节点编号

int insert(int num, int i) {

    int rt = ++cnt;

    tree[rt][0] = tree[i][0];

    tree[rt][1] = tree[i][1];

    pass[rt] = pass[i] + 1;

    for (int b = BIT, path, pre = rt, cur; b >= 0; b--, pre = cur) {

        path = (num >> b) & 1;

        i = tree[i][path];

        cur = ++cnt;

        tree[cur][0] = tree[i][0];

        tree[cur][1] = tree[i][1];

        pass[cur] = pass[i] + 1;

        tree[pre][path] = cur;

    }

    return rt;

}

int query(int num, int u, int v) {

    int ans = 0;

    for (int b = BIT, path, best; b >= 0; b--) {

        path = (num >> b) & 1;

        best = path ^ 1;

        if (pass[tree[v][best]] > pass[tree[u][best]]) {

            //表示可以是best

            ans += 1 << b;

            u = tree[u][best];

            v = tree[v][best];

        } else {

            u = tree[u][path];

            v = tree[v][path];

        }

    }

    return ans;

}

int main() {

    cin >> n >> m;

    eor = 0;//作为所有值的异或和

    root[0] = insert(eor, 0);

    for (int i = 1, num; i <= n; i++) {

        cin >> num;

        eor ^= num;

        root[i] = insert(eor, root[i - 1]);

    }

    string op;

    int x, y, z;

    for (int i = 1; i <= m; i++) {

        cin >> op;

        if (op == "A") {

            cin >> x;

            eor ^= x;

            n++;

            root[n] = insert(eor, root[n - 1]);

        } else {

            cin >> x >> y >> z;

            //首先将要查询的变为eor^z

            if (x == 1) {

                cout << query(eor ^ z, 0, root[y - 1]) << "\n";

            } else {

                cout << query(eor ^ z, root[x - 2], root[y - 1]) << "\n";

            }

        }

    }

    return 0;

}

Manacher

// Manacher算法模版

// 求字符串s中最长回文子串的长度

#include<bits/stdc++.h>

using namespace std;

const int MAXN = 11000001;

char ss[MAXN << 1];//这个数组中间添加了#

char a[MAXN];//原始数组

int p[MAXN << 1];//半径数组

int n;

void manacherss() {

    n = strlen(a) \* 2 + 1;

    for (int i = 0, j = 0; i < n; i++) {

        ss[i] = (i & 1) == 0 ? '#' : a[j++];

    }

}

int manacher() {

    manacherss();

    int maxans = 0;

    for (int i = 0, c = 0, r = 0, len; i < n; i++) {//i是此时来到的中心位置

        len = r > i ? min(p[2 \* c - i], r - i) : 1;//这个值是基本值  如果包住了  那么基本值就是最终值  否则后续再加

        while (i + len < n && i - len >= 0 && ss[i + len] == ss[i - len]) {

            len++;

            //如果被包住了  根本就进不来

        }

        //此时已经求出了以i为中心的回文半径大小

        if (i + len > r) {

            r = i + len;

            c = i;//c表示的是如今的r是以那个点为中心的右边界

            //如果更新了右边界  那么c就是i  r就是i+len

        }

        maxans = max(maxans, len);

        p[i] = len;

        //得到回文半径

    }

    return maxans - 1;

}

可持久化线段树

// 给定一个长度为n的数组arr，下标1~n，一共有m条查询

// 每条查询 l r k : 打印arr[l..r]中第k小的数字

// 1 <= n、m <= 2 \* 10^5 0 <= arr[i] <= 10^9

int n, m, s;

int arr[MAXN]; // 原始数组

int sorted[MAXN]; // 收集权值排序并且去重做离散化

// root[i] : 插入arr[i]之后形成新版本的线段树，记录头节点编号

// 0号版本的线段树代表一个数字也没有时，每种名次的数字出现的次数

// i号版本的线段树代表arr[1..i]范围内，每种名次的数字出现的次数

int root[MAXN],ls[MAXT],rs[MAXT];

// 排名范围内收集了多少个数字

int size[MAXT],cnt;

// 返回num在所有值中排名多少

int kth(int num) {

    int left = 1, right = s, mid, ans = 0;

    while (left <= right) {

        mid = (left + right) / 2;

        if (sorted[mid] <= num) {

            ans = mid;left = mid + 1;

        } else {

            right = mid - 1;

        }

    }

    return ans;

}

// 排名范围l~r，建立线段树，返回头节点编号

int build(int l, int r) {

    int rt = ++cnt;

    size[rt] = 0;

    if (l < r) {

        int mid = (l + r) / 2;

        ls[rt] = build(l, mid);

        rs[rt] = build(mid + 1, r);

    }

    return rt;

}

// 排名范围l~r，信息在i号节点，增加一个排名为jobi的数字 返回新的头节点编号

int insert(int jobi, int l, int r, int i) {

    int rt = ++cnt;

    ls[rt] = ls[i];

    rs[rt] = rs[i];

    size[rt] = size[i] + 1;//新加一个元素  大小加一

    if (l < r) {

        int mid = (l + r) / 2;

        if (jobi <= mid) {

            ls[rt] = insert(jobi, l, mid, ls[rt]);

        } else {

            rs[rt] = insert(jobi, mid + 1, r, rs[rt]);

        }

    }

    return rt;

}

// 排名范围l~r，老版本信息在u号节点，新版本信息在v号节点 返回，第jobk小的数字，排名多少

int query(int jobk, int l, int r, int u, int v) {

    if (l == r) {

        return l;

    }

    int lsize = size[ls[v]] - size[ls[u]];

    int mid = (l + r) / 2;

    if (lsize >= jobk) {

        return query(jobk, l, mid, ls[u], ls[v]); //如果左半边排名足够 那么就在左半边找

    } else {

        return query(jobk - lsize, mid + 1, r, rs[u], rs[v]);

    }

}

// 权值做离散化并且去重 + 生成各版本的线段树

void prepare() {

    cnt = 0;

    for (int i = 1; i <= n; i++) {

        sorted[i] = arr[i];

    }

    sort(sorted + 1, sorted + n + 1);

    s = 1;

    for (int i = 2; i <= n; i++) {

        if (sorted[s] != sorted[i]) {

            sorted[++s] = sorted[i];

        }

    }

    root[0] = build(1, s);//建立第一棵树

    for (int i = 1, x; i <= n; i++) {

        x = kth(arr[i]);

        root[i] = insert(x, 1, s, root[i - 1]);

        //每次基于上一个树插入一个元素  建立新树

    }

}

树上差分

点差分

// 有n个节点形成一棵树，一开始所有点权都是0

// 给定很多操作，每个操作(a,b)表示从a到b路径上所有点的点权增加1

// 所有操作完成后，返回树上的最大点权

//点差分中将从a到b的路径上所有点权加v  相当于a+v b+v  lca-v  lcafa-v

//最后统计的时候 来到一个节点  先遍历他的子节点  然后这个节点加上子节点的点权

//最终得到自己的点权

int deep[MAXN],stjump[MAXN][LIMIT];

int power;

int num[MAXN];//记录点权

//这个函数的作用是建立deep  st的信息

void dfs1(int u, int f) {

    deep[u] = deep[f] + 1;

    stjump[u][0] = f;

    for (int p = 1; p <= power; p++) {

        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];

    }

    //完成u的deep  stjump

    for (int e = head[u]; e != 0; e = Next[e]) {

        if (to[e] != f) {

            dfs1(to[e], u);

        }

        //向下递归

    }

}

int lca(int a, int b) {

    if (deep[a] < deep[b]) {

        int tmp = a;

        a = b;

        b = tmp;

    }

    //确定大小关系

    for (int p = power; p >= 0; p--) {

        if (deep[stjump[a][p]] >= deep[b]) {

            a = stjump[a][p];

        }

    }

    //首先将两者变为同一高度

    if (a == b) {

        return a;

    }

    //如果相同说明就是祖先关系

    for (int p = power; p >= 0; p--) {

        if (stjump[a][p] != stjump[b][p]) {

            a = stjump[a][p];

            b = stjump[b][p];

        }

        //判断跳完后是否符合规则

    }

    return stjump[a][0];

    //我们将头结点的祖先设置为0  实际上没有0

}

//更新修改后的信息

void dfs2(int u,int f){

    for(int i=head[u];i>0;i=Next[i]){

        int v=to[i];

        if(v!=f){

            dfs2(v,u);

        }

    }

    for(int i=head[u];i>0;i=Next[i]){

        int v=to[i];

        if(v!=f){

            num[u]+=num[v];

        }

    }

}

边差分

平衡树

FHQ平衡树

// 实现一种结构，支持如下操作，要求单次调用的时间复杂度O(log n)

// 1，增加x，重复加入算多个词频

// 2，删除x，如果有多个，只删掉一个

// 3，查询x的排名，x的排名为，比x小的数的个数+1

// 4，查询数据中排名为x的数

// 5，查询x的前驱，x的前驱为，小于x的数中最大的数，不存在返回整数最小值

// 6，查询x的后继，x的后继为，大于x的数中最小的数，不存在返回整数最大值

// 所有操作的次数 <= 10^5

// -10^7 <= x <= +10^7

//词频不压缩的一个巨大优势是在删除和加入节点时非常方便

#include <bits/stdc++.h>

using namespace std;

const int MAXN = 100001;

int head = 0;

int cnt = 0;

int key[MAXN];

int ls[MAXN];

int rs[MAXN];

int size[MAXN];

double priority[MAXN];

//没有词频压缩  所以出现次数就是1

void up(int i) {

    size[i] = size[ls[i]] + size[rs[i]] + 1;

}

void split(int l, int r, int i, int num) {

    if (i == 0) {

        rs[l] = ls[r] = 0;

    } else {

        if (key[i] <= num) {

            rs[l] = i;

            split(i, r, rs[i], num);

        } else {

            ls[r] = i;

            split(l, i, ls[i], num);

        }

        up(i);

    }

}

int merge(int l, int r) {

    if (l == 0 || r == 0) {

        return l + r;

    }

    if (priority[l] >= priority[r]) {

        rs[l] = merge(rs[l], r);

        up(l);

        return l;

    } else {

        ls[r] = merge(l, ls[r]);

        up(r);

        return r;

    }

}

void add(int num) {

    split(0, 0, head, num);

    key[++cnt] = num;

    size[cnt] = 1;

    priority[cnt] = (double)rand() / RAND\_MAX;

    head = merge(merge(rs[0], cnt), ls[0]);

}

//删除节点的时候 是将树按照num分裂  然后将<=num 的按照 num-1 分裂

//将那么 >num-1 的树头结点一定是num  只要将这个节点忽略就是删除节点

void remove(int num) {

    split(0, 0, head, num);

    int lm = rs[0];

    int r = ls[0];

    split(0, 0, lm, num - 1);

    int l = rs[0];

    int m = ls[0];

    head = merge(merge(l, merge(ls[m], rs[m])), r);

}

int getRank(int num) {

    split(0, 0, head, num - 1);

    int ans = size[rs[0]] + 1;

    head = merge(rs[0], ls[0]);

    return ans;

}

int index(int i, int x) {

    if (size[ls[i]] >= x) {

        return index(ls[i], x);

    } else if (size[ls[i]] + 1 < x) {

        return index(rs[i], x - size[ls[i]] - 1);

    } else {

        return key[i];

    }

}

int index(int x) {

    return index(head, x);

}

int pre(int i, int num) {

    if (i == 0) {

        return INT\_MIN;

    }

    if (key[i] >= num) {

        return pre(ls[i], num);

    } else {

        return max(key[i], pre(rs[i], num));

    }

}

int pre(int num) {

    return pre(head, num);

}

int post(int i, int num) {

    if (i == 0) {

        return INT\_MAX;

    }

    if (key[i] <= num) {

        return post(rs[i], num);

    } else {

        return min(key[i], post(ls[i], num));

    }

}

int post(int num) {

    return post(head, num);

}

快读

inline int read(){

    int x=0,f=1;

    char ch=getchar();

    while(ch<'0'||ch>'9'){

        if(ch=='-')

            f=-1;

        ch=getchar();

    }

    while(ch>='0' && ch<='9')

        x=x\*10+ch-'0',ch=getchar();

    return x\*f;

}

int n,m,s,t;

int dep[MAXN];//记录深度数组

int iter[MAXN];//当前弧数组 记录这个节点有效访问的第一条边的编号

//这里编号从2开始 为了正反边编号寻找方便

int cnt=2;

int head[MAXN];

int nxt[MAXM];

int to[MAXM];

int now[MAXM];//表示现在的流量

int cap[MAXM];//表示流量限制

void addedge(int u,int v,int w){

    nxt[cnt]=head[u];

    to[cnt]=v;

    cap[cnt]=w;

    head[u]=cnt++;

    nxt[cnt]=head[v];

    to[cnt]=u;

    cap[cnt]=0;

    head[v]=cnt++;

}

//BFS构建分层图，并判断是否存在增广路径

bool bfs(){

    for(int i=1;i<=n;i++){

        dep[i]=-1;

    }

    queue<int>q;

    dep[s]=0;

    q.push(s);

    while(!q.empty()){

        int u=q.front();

        q.pop();

        for(int i=head[u];i;i=nxt[i]){

            int v=to[i];

            int w=cap[i];

            int k=now[i];

            if(dep[v]<0&&(k<w)){

                dep[v]=dep[u]+1;

                q.push(v);

            }

        }

    }

    return dep[t]>=0;// 如果汇点未被访问到，说明无增广路径

}

int dfs(int u,int f){

    if(u==t){

        return f;

    }

    int flow=0;

    for(int &i=iter[u];i;i=nxt[i]){

        //注意这里是引用  iter会随着i发生变化

        int v=to[i];

        int w=cap[i];

        int k=now[i];

        if(dep[u]+1==dep[v]&&k<w){

            int d=dfs(v,min(f,w-k));

            if(d>0){

                now[i]+=d;

                now[i^1]-=d;//更新反向边

                flow+=d;

                f-=d;

                if(f==0){

                    break;

                }

            }

        }

    }

    return flow;

}

// Dinic算法主函数

int maxflow(){

    int flow=0;

    while(bfs()){

        //当前弧全部初始化为最初值

        for(int i=1;i<=n;i++){

            iter[i]=head[i];

        }

        int maxflow;

        while((maxflow=dfs(s,INF))>0){

            flow+=maxflow;

        }

    }

    return flow;

}

//这道题要求的是最小割  也就是最大流   除此之外  要求割边尽量少

int n,m;

int dep[MAXN];int iter[MAXN];

int head[MAXN];int nxt[MAXM];int to[MAXM];int now[MAXM];int cap[MAXM];

int cnt=2;

void addedge(int u,int v,int w){

    nxt[cnt]=head[u];

    to[cnt]=v;

    cap[cnt]=w;

    head[u]=cnt++;

}

//BFS构建分层图，并判断是否存在增广路径

bool bfs(){

    for(int i=1;i<=n;i++){

        dep[i]=-1;

    }

    queue<int>q;

    dep[1]=0;

    q.push(1);

    while(!q.empty()){

        int u=q.front();

        q.pop();

        for(int i=head[u];i;i=nxt[i]){

            int v=to[i];

            int w=cap[i];

            int k=now[i];

            if(dep[v]<0&&(k<w)){

                dep[v]=dep[u]+1;

                q.push(v);

            }

        }

    }

    return dep[n]>=0;// 如果汇点未被访问到，说明无增广路径

}

// DFS寻找增广路径（多路增广）

int dfs(int u,int f){

    //当前来到了u节点 有f的流量可供使用f就是目前这条路最多可以消耗掉的流量

    if(u==n){return f;}

    int flow=0;

    for(int &i=iter[u];i;i=nxt[i]){

        //注意这里是引用  iter会随着i发生变化

        int v=to[i];

        int w=cap[i];

        int k=now[i];

        if(dep[u]+1==dep[v]&&k<w){

            int d=dfs(v,min(f,w-k));

            if(d>0){

                now[i]+=d;

                now[i^1]-=d;//更新反向边

                flow+=d;

                f-=d;

                if(f==0){

                    break;

                }

            }

        }

    }

    return flow;

}

// Dinic算法主函数

int maxflow(){

    int flow=0;

    while(bfs()){

        //当前弧全部初始化为最初值

        for(int i=1;i<=n;i++){

            iter[i]=head[i];

        }

        int maxflow;

        while((maxflow=dfs(1,INF))>0){

            flow+=maxflow;

        }

    }

    return flow;

}

signed main(){

    cin>>n>>m;

    for(int i=1;i<=m;i++){

        int u,v,w;

        cin>>u>>v>>w;

        addedge(u,v,w\*BASE+1);

        addedge(v,u,0);

    }

    int ans=maxflow();

    cout<<(ans/BASE)<<' '<<(ans%BASE)<<endl;

    return 0;

}

回文自动机

// 这道题需要统计出现次数  因为回文自动机上都是本质不同的回文串

// 那么次数就是本质相同的回文串的出现次数  加上是别的回文串的后缀次数

// 本质相同的回文串次数好算  就是在每次添加一个字符后 出现的次数加一

// 如果一个回文串是另外一个的最长子回文后缀  那么他出现的次数应该要加上这个的出现次数

// 因为这个回文串一定是包含了这个最长的子回文后缀的

// 这里其实是一个类似于fail树的操作

int n;

int tree[MAXN][26];

int fail[MAXN];

int len[MAXN];// 节点代表的回文串长度

int num[MAXN];

int cnt;

int last;

char s[MAXN];

long long ans;

// 初始化回文自动机

void init(){

    cnt=1;last=0;

    len[0]=0,len[1]=-1;

    fail[0]=1;//偶根节点的fail指针指向奇根

    fail[1]=1;

    num[0]=0,num[1]=0;

    s[0]='#';// 特殊字符，防止越界

}

// 在fail链上查找满足条件的节点  即回文串前面一个字母是t[i]的位置  最大的位置

int getfail(int p,int i){

    while(s[i-len[p]-1]!=s[i]){

        p=fail[p];

    }

    return p;

}

// 添加字符t[i]到回文自动机

void insert(int i){

    // 当前字符的数值

    int c=s[i]-'a';

    // 查找合适的节点  其实这里找的是一个类似于父亲节点

    int cur=getfail(last,i);

    // 如果该节点不存在，创建新节点

    if(!tree[cur][c]){

        int x=++cnt;

        len[x]=len[cur]+2;

        // 这里要找的是fail指针指向的节点  意思是最长的回文后缀  但不能是整体  即子回文后缀

        // 所以这里必须是cur的fail指针所指向的节点

        int p=getfail(fail[cur],i);

        fail[x]=tree[p][c];

        tree[cur][c]=x;

    }

    last=tree[cur][c];

}

int main()

{

    scanf("%s",s+1);

    n=strlen(s+1);

    init();

    for(int i=1;i<=n;i++){

        insert(i);

        num[last]++;

    }

    for(int i=cnt;i>=1;i--){

        num[fail[i]]+=num[i];

    }

    for(int i=1;i<=cnt;i++){

        ans=max(ans,1ll\*num[i]\*len[i]);

    }

    cout<<ans<<endl;

    return 0;

}

带权二分至少k个

//https://www.luogu.com.cn/problem/P2619

//这种题目  我们称之为带权二分  即边的类型有两种  其中一种必须要选择k条以上（或者k条）

//我们的思路是 二分+kruskal生成树

//如果使用原本的边权  我们不好确定那些边会被使用到

//所以我们将特殊的边设置一个加权  使得在最小生成树中特殊边存在k条

//这道题的二分逻辑是只要符合要求的大于k  那么就是答案 同时加权的左边界变大

//如果说当前白边加上mid后，白边条数temp>need了，然而，如果加上mid+1后，temp<need了

//题目中说到了：保证有解，所以出现上述情况时一定有黑边==白边的边权

//所以我们在排序的时候  加权后边权相同的边  特殊边优先选择

//在白边条数temp>need时更新答案  最终答案ans=sum−mid\*need

int n,m,need;

int mid;

struct node{

    int u,v,w,col;

} nums[MAXM];

int fa[MAXN];

bool cmp(node a,node b){

    if((a.w+(1-a.col)\*mid)!=(b.w+(1-b.col)\*mid)) return (a.w+(1-a.col)\*mid)<(b.w+(1-b.col)\*mid);

    return a.col<b.col;

}

int find(int x){

    return x==fa[x]?x:fa[x]=find(fa[x]);

}

bool check(){

    int cnt=0;

    sort(nums+1,nums+m+1,cmp);

    for(int i=1;i<=n;i++){

        fa[i]=i;

    }

    int k=0;

    for(int i=1;i<=m;i++){

        int x=nums[i].u;

        int y=nums[i].v;

        int fx=find(x),fy=find(y);

        if(fx!=fy){

            fa[fx]=fy;

            if(nums[i].col==0){

                cnt++;

            }

            k++;

            if(k==n-1){

                break;

            }

        }

    }

    return cnt>=need;

}

int main(){

    n=read(),m=read(),need=read();

    for(int i=1;i<=m;i++){

        nums[i].u=read()+1,nums[i].v=read()+1,nums[i].w=read(),nums[i].col=read();

    }

    int l=-105,r=105,ans;

    while(l<=r){

        mid=(l+r)>>1;

        if(check()){

            ans=mid;

            l=mid+1;

        }else{

            r=mid-1;

        }

    }

    mid=ans;

    int all=0;

    sort(nums+1,nums+m+1,cmp);

    for(int i=1;i<=n;i++){

        fa[i]=i;

    }

    for(int i=1;i<=m;i++){

        int x=nums[i].u, y=nums[i].v;

        int fx=find(x),fy=find(y);

        if(fx!=fy){

            fa[fx]=fy;

            all+=((1-nums[i].col)\*mid+nums[i].w);

        }

    }

    all-=mid\*need;

    printf("%d\n",all);

    return 0;

}

带权二分——恰好k个

// https://www.luogu.com.cn/problem/P5633

// 这道题要求的是必须是k个  我们判断的时候  要处理优先选和优先不选的情况  这是连个极值

// 可能会有多个数字都满足条件 但是这些数字最终的答案都是一样的

int find(int x){

    return x==fa[x]?x:fa[x]=find(fa[x]);

}

bool cmp(node a,node b){

    return ((a.w+a.op\*x)==(b.w+b.op\*x))?(pritority?(a.op>b.op):(a.op<b.op)):((a.w+a.op\*x)<(b.w+b.op\*x));

}

int query(int limit,int op){

    all=0;

    x=limit,pritority=op;

    sort(nums+1,nums+m+1,cmp);

    for(int i=1;i<=n;i++){

        fa[i]=i;

    }

    int cnt=0;

    for(int i=1;i<=m;i++){

        int fx=find(nums[i].u);

        int fy=find(nums[i].v);

        if(fx!=fy){

            all+=nums[i].w+nums[i].op\*limit;

            fa[fx]=fy;

            if(nums[i].op){

                cnt++;

            }

        }

    }

    return cnt;

}

int main()

{

    n=read(),m=read(),s=read(),k=read();

    all=0;

    for(int i=1;i<=m;i++){

        nums[i].u=read(),nums[i].v=read(),nums[i].w=read();

        if(nums[i].u==s||nums[i].v==s){

            nums[i].op=1;

            all++;

        }

    }

    if(all<k||!check()){

        cout<<"Impossible"<<endl;

        return 0;

    }

    int l=-30005,r=30005,ans=0;

    bool flag=false;

    while(l<=r){

        int mid=(l+r)>>1;

        int cntmin=query(mid,0);

        int cntmax=query(mid,1);

        if(cntmin<=k&&cntmax>=k){

            flag=true;

            ans=all-k\*mid;

            break;

        }

        else if(cntmin>k){

            l=mid+1;

        }

        else if(cntmax<k){

            r=mid-1;

        }

    }

    if(flag){

        cout<<ans<<endl;

    }

    else{

        cout<<"Impossible"<<endl;

    }

    return 0;

}

笛卡尔树

// 给定一个长度为n的数组arr，下标从1开始

// 构建一棵二叉树，下标按照搜索二叉树组织，值按照小根堆组织

// 建树的过程要求时间复杂度O(n)

// 1 <= n <= 10^7

// 测试链接 : https://www.luogu.com.cn/problem/P5854

// 提交如下代码，可以通过所有测试用例

// 笛卡尔树建立的时候需要单调栈  栈里面比较的东西是按照堆组织的

// 里面存储的是按照二叉排序组织的值  这样建图是O(n)

//如果没有变小的数字  那么建树就是一直往右边节点建立

//也不会有节点从栈中弹出  单调队列比较的是按堆组织的  大压小是小跟堆

//利用单调栈建立笛卡尔树

void build() {

    int top = 0;

    for (int i = 1; i <= n; i++) {

        int pos = top;

        while (pos > 0 && arr[sta[pos]] > arr[i]) {

            //单调栈  弹出不符合的元素

            pos--;

        }

        // 插入一个元素 它是连接在哪个节点的右孩子上的   哪些节点连接到它的左孩子

        if (pos > 0) {

            //那么就是没有弹干净   那么当前节点就会成为栈顶节点的右孩子

            rs[sta[pos]] = i;

        }

        if (pos < top) {

            //表示有元素弹出 那么一定是当前节点的左孩子节点

            ls[i] = sta[pos + 1];

        }

        sta[++pos] = i;//加入栈中

        top = pos;

    }

}

Twosat

// https://www.luogu.com.cn/problem/P5782

// 这里介绍了为什么要选择col[i]<col[i+1]

//输出答案求解

//要使用到拓扑排序   但是实际上拓扑也就是为了防止出现 a->b->a'  即a->a' 这种可能会导致错误情况

//所以我们选择拓扑序尽量大的  即尽量在后面的  a'一定无法推导出a（如果a'的拓扑序更大的话）

//因为后面无法指向前面  否则不就是在一个环里面了吗  那这样就是错误的

//拓扑序尽量大的  其实颜色编号是尽量小的  因为tarjan的时候  先将尾部的那些数字访问到  然后将他们先缩点了

//所以虽然在有的时候  选择较小的颜色编号是对的  但不是最优的

int n,m;

int dfn[MAXN],low[MAXN];

int st[MAXN],top;

int color,Time;

int col[MAXN];

bool instack[MAXN];

void tarjan(int x){

    dfn[x]=++Time;

    low[x]=Time;

    st[++top]=x;

    instack[x]=true;

    for(int i=head[x];i;i=Next[i]){

        int v=to[i];

        if(!dfn[v]){

            //表示这个节点没有被访问过

            tarjan(v);

            low[x]=min(low[x],low[v]);

        }

        else if(instack[v]){

            //表示这个属于是回溯了 一定是同一个环上的

            low[x]=min(low[x],dfn[v]);

        }

    }

    if(dfn[x]==low[x]){

        col[x]=++color;

        //将所有节点按照颜色分类  完成缩点

        while(st[top]!=x){

            //属于同一个强联通分量

            col[st[top]]=color;

            instack[st[top]]=false;

            top--;

        }

        instack[x]=false;

        top--;

    }

}

int main(){

    for(int i=1;i<=2\*n;i++){

        if(!dfn[i]){

            tarjan(i);

        }

    }

    for(int i=1;i<=n;i++){

        if(col[i]==col[i+n]){

            //不符合要求 直接退出

            cout<<"IMPOSSIBLE"<<endl;

            return 0;

        }

    }

    cout<<"POSSIBLE"<<endl;

    for(int i=1;i<=n;i++){

        if(col[i]>col[i+n])

            printf("1 ");

        else

            printf("0 ");

    }

    return 0;

}

左偏树

// 左偏树模版题，C++版

// 依次给定n个非负数字，表示有n个小根堆，每个堆只有一个数

// 实现如下两种操作，操作一共调用m次

// 1 x y : 第x个数字所在的堆和第y个数字所在的堆合并

//         如果两个数字已经在一个堆或者某个数字已经删除，不进行合并

// 2 x   : 打印第x个数字所在堆的最小值，并且在堆里删掉这个最小值

//         如果第x个数字已经被删除，也就是找不到所在的堆，打印-1

//         若有多个最小值，优先删除编号小的

// 1 <= n, m <= 10^5

// 测试链接 : https://www.luogu.com.cn/problem/P3377

// 如下实现是C++的版本，C++版本和java版本逻辑完全一样

// 提交如下代码，可以通过所有测试用例

// 左偏堆是二叉树结构 但并不是完全二叉树结构  是广义堆

#include <bits/stdc++.h>

using namespace std;

const int MAXN = 100001;

int n, m;

int num[MAXN];

int ls[MAXN];

int rs[MAXN];

//表示这个节点需要经过几个节点才会到达空节点

int dist[MAXN];

// 并查集需要father数组，方便快速找到树的头

// father[i]不代表i在树上的父亲节点

// father是并查集找代表节点的路径信息

// 需要保证，并查集最上方的代表节点 = 树的头节点

int fa[MAXN];

//预处理

void prepare() {

    dist[0] = -1;

    for(int i = 1; i <= n; i++) {

        ls[i] = rs[i] = dist[i] = 0;

        fa[i] = i;

    }

}

int find(int i) {

    fa[i] = fa[i] == i ? i : find(fa[i]);//扁平化处理

    return fa[i];

}

//将以i 、 j为头的树合并  并返回头结点编号

int merge(int i, int j) {

    if (i == 0 || j == 0) {

        //遇到了空节点 那么就以非空作为头结点

        return i + j;

    }

    if (num[i] > num[j] || (num[i] == num[j] && i > j)) {

        //谁小谁做头 后面是题目规定  一般都是这个规定

        int tmp = i;

        i = j;

        j = tmp;

    }

    rs[i] = merge(rs[i], j);//将右儿子和它做合并

    //检查dist有没有问题 是否需要交换

    if (dist[ls[i]] < dist[rs[i]]) {

        int tmp = ls[i];

        ls[i] = rs[i];

        rs[i] = tmp;

    }

    dist[i] = dist[rs[i]] + 1;

    fa[ls[i]] = fa[rs[i]] = i;

    return i;

}

// 节点i一定是左偏树的头，在左偏树上删掉节点i，返回新树的头节点编号

int pop(int i) {

    fa[ls[i]] = ls[i];

    fa[rs[i]] = rs[i];

    // 并查集有路径压缩，所以i下方的某个节点x，可能有father[x] = i

    // 现在要删掉i了，所以x往上会找不到正确的头节点

    // 为了任何节点往上都能找到正确的头，所以要有下面这句

    fa[i] = merge(ls[i], rs[i]);

    ls[i] = rs[i] = dist[i] = 0;

    return fa[i];

}

删除任意节点模板

// 节点i是所在左偏树的任意节点，删除节点i，返回整棵树的头节点编号

int remove(int i) {

    //找到这棵树的头结点

    int h = find(i);

    //将左右两个儿子的父亲暂时改为自己

    fa[ls[i]] = ls[i];

    fa[rs[i]] = rs[i];

    //将两个儿子合并  返回新节点

    int s = merge(ls[i], rs[i]);

    //f是i节点的上一层节点

    int f = up[i];

    //相当于删除头结点操作

    fa[i] = s;

    //这里将两个儿子的合并后节点s的父亲改为f

    up[s] = f;

    if (h != i) {

        //并查集的扁平化处理

        fa[s] = h;

        if (ls[f] == i) {

            ls[f] = s;

        } else {

            rs[f] = s;

        }

        //向上修正  逐层更新  直到到达某层之后不会继续影响上面

        for (int d = dist[s], tmp; dist[f] > d + 1; f = up[f], d++) {

            dist[f] = d + 1;

            if (dist[ls[f]] < dist[rs[f]]) {

                tmp = ls[f]; ls[f] = rs[f]; rs[f] = tmp;

            }

        }

    }

    up[i] = ls[i] = rs[i] = dist[i] = 0;//将这个节点的信息修正  消除信息

    return fa[s];

}

差分约束

// 这种形式是值往小的方向更新  初始为最大值  一个式子u-v>=w   --->  v<=u-w v的权值小于等于u的权值-w

int head[MAXN];

int Next[MAXM];

int to[MAXM];

int weight[MAXM];

int cnt=1;

// 源点出发到每个节点的距离表

int dis[MAXN];

// 节点被松弛的次数

int updateCnt[MAXN];

//当前在队列中的话  就是true  否则就是false

bool enter[MAXN];

//每一轮都弹出最后面的一个  如果一个点可以被优化  那么加入队列

queue<int>q;

bool spfa() {

    dis[0] = 0;

    updateCnt[0]++;

    q.push(0) ;

    enter[0] = true;

    //这道题目中的源点都是0 保持不变   所以以上的操作都针对1

    //实际上应该理解为是针对源点的

    while (!q.empty()) {

        int u = q.front();

        q.pop();

        enter[u] = false;

        for (int ei = head[u], v, w; ei > 0; ei = Next[ei]) {

            v = to[ei];

            w = weight[ei];

            if (dis[u] + w < dis[v]) {

                //如果值变小了  那么就要改变

                dis[v] = dis[u] + w;

                if (!enter[v]) {

                    //如果不在队列中才会加入

                    // 松弛次数超过n-1就有负环

                    //但是我们加入了一个超级源点  所以总点数加一

                    if (++updateCnt[v] > n) {

                        return true;

                    }

                    q.push(v);

                    enter[v] = true;

                }

            }

        }

    }

    return false;

}

int main(){

    cin>>n>>m;

    for(int i=1;i<=n;i++){

        dis[i]=INT\_MAX;

    }

    for(int i=1;i<=n;i++){

        //超级源点建边  这个仅仅是为了维护联通性  这种形式是为了判断是否存在负环

        // 所以这里可以使用dis[i]<=dis[0]+0

        addEdge(0,i,0);

    }

    for(int i=1;i<=m;i++){

        int u,v,w;

        cin>>u>>v>>w;

        // 形式1的连边方式  u<=v+w  即v向u连一条权值为w的边

        addEdge(v,u,w);

    }

    if(spfa()){

        cout<<"NO"<<endl;

    }else{

        for(int i=1;i<=n;i++)

            cout<<dis[i]<<' ';

    }

    return 0;

}

第一种：这种形式是值往小的方向更新 初始为最大值 一个式子u-v>=w ---> v<=u-w v的权值小于等于u的权值-w

第二种：// 这种形式是变大才会更新  一个式子比如u-v>=w   ---> u>=v+w  也就是u的权值大于 v的权值加上w

第三种：限制性超级源点 u的权值是5 也就是u-(n+1)>=5 u-(n+1)<=5

普通莫队

bool QueryCmp2(Query &a, Query &b) {

    if (bi[a.l] != bi[b.l]) {

        return bi[a.l] < bi[b.l];

    }

    if ((bi[a.l] & 1) == 1) {

        return a.r < b.r;

    } else {

        return a.r > b.r;

    }

}

void prepare() {

    int blen = (int)sqrt(n);

    for (int i = 1; i <= n; i++) {

        bi[i] = (i - 1) / blen + 1;

    }

    sort(query + 1, query + q + 1, QueryCmp2);

}

void compute() {

    int winl = 1, winr = 0;

    for (int i = 1; i <= q; i++) {

        int jobl = query[i].l;

        int jobr = query[i].r;

        while (winl > jobl) {

            add(arr[--winl]);

        }

        while (winr < jobr) {

            add(arr[++winr]);

        }

        while (winl < jobl) {

            del(arr[winl++]);

        }

        while (winr > jobr) {

            del(arr[winr--]);

        }

        ans[query[i].id] = kind;

    }

}

带修莫队（带修莫队的排序和块长是比较特殊的）

bool QueryCmp(Query &a, Query &b) {

    if (bi[a.l] != bi[b.l]) {

        return bi[a.l] < bi[b.l];

    }

    if (bi[a.r] != bi[b.r]) {

        return bi[a.r] < bi[b.r];

    }

    return a.t < b.t;

}

// 传进来的参数直接就是数值

void del(int num) {

    if (--cnt[num] == 0) {

        kind--;

    }

}

void add(int num) {

    if (++cnt[num] == 1) {

        kind++;

    }

}

// jobl..jobr 数组范围 tim : 生效或者撤销的修改时间点

void moveTime(int jobl, int jobr, int tim) {

    int pos = update[tim].pos;

    int val = update[tim].val;

    if (jobl <= pos && pos <= jobr) {

        del(arr[pos]);

        add(val);

    }

    // 不管生效还是撤销，数据只要在arr和update之间交换即可

    int tmp = arr[pos];

    arr[pos] = val;

    update[tim].val = tmp;

}

void compute() {

    int winl = 1, winr = 0, wint = 0;

    for (int i = 1; i <= cntq; i++) {

        int jobl = query[i].l;

        int jobr = query[i].r;

        int jobt = query[i].t;

        int id = query[i].id;

        while (winl > jobl) {

            add(arr[--winl]);

        }

        while (winr < jobr) {

            add(arr[++winr]);

        }

        while (winl < jobl) {

            del(arr[winl++]);

        }

        while (winr > jobr) {

            del(arr[winr--]);

        }

        while (wint < jobt) {

            moveTime(jobl, jobr, ++wint);

        }

        while (wint > jobt) {

            moveTime(jobl, jobr, wint--);

        }

        ans[id] = kind;

    }

}

void prepare() {

    int blen = max(1, (int)pow(n, 2.0 / 3));

    for (int i = 1; i <= n; i++) {

        bi[i] = (i - 1) / blen + 1;

    }

    sort(query + 1, query + cntq + 1, QueryCmp);

}

int main() {

    cin >> n >> m;

    for (int i = 1; i <= n; i++)

        cin >> arr[i];

    char op; int l, r, pos, val;

    for (int i = 1; i <= m; i++) {

        cin >> op;

        if (op == 'Q') {

            cin >> l >> r;

            cntq++;query[cntq].l = l; query[cntq].r = r;

query[cntq].t = cntu; query[cntq].id = cntq;

        } else {

            cin >> pos >> val;

            cntu++;update[cntu].pos = pos; update[cntu].val = val;

        }

    }

}

只增回滚莫队

// 我们还是按照普通莫队进行排序  我们是按照块进行操作的   对于左右边界都在同一块中的问题  直接暴力维护答案

// 对于右边不和左边在一块的  我们准备一个窗口  首先这些问题的右边界的都是有序的

// 对于每个问题 我们首先先将窗口更新到右边界  记录下此时窗口的状态

// 然后更行左边界  收集答案之后  减窗口信息复原到我们记录  然后重复这个操作

// 对于左边界的话  其实和左右边界在同一块的代价一样 不过就是左右反复 总代价O(q\*根号n)

// 只在窗口扩展的时候更新答案   这里不能使用奇偶优化性排序  任何一个块的右边界都必须是递增的

bool QueryCmp(Query &a, Query &b) {

    if (bi[a.l] != bi[b.l]) {

        return bi[a.l] < bi[b.l];

    }

    return a.r < b.r;

}

// 暴力遍历arr[l..r]得到答案 如果左右边界都在一块中 那么直接暴力维护

long long force(int l, int r) {

    long long ret = 0;

    for (int i = l; i <= r; i++) {

        cnt[arr[i]]++;

    }

    for (int i = l; i <= r; i++) {

        ret = max(ret, 1LL \* cnt[arr[i]] \* sorted[arr[i]]);

    }

    for (int i = l; i <= r; i++) {

        cnt[arr[i]]--;

    }

    return ret;

}

void compute() {

    for (int block = 1, qi = 1; block <= bnum && qi <= m; block++) {

        curAns = 0;

        fill(cnt + 1, cnt + cntv + 1, 0);

        int winl = br[block] + 1, winr = br[block];

        for (; qi <= m && bi[query[qi].l] == block; qi++) {

            int jobl = query[qi].l;

            int jobr = query[qi].r;

            int id = query[qi].id;

            if (jobr <= br[block]) {

                ans[id] = force(jobl, jobr);

            } else {

                while (winr < jobr) {

                    add(arr[++winr]);

                }

// 记录一下此时的答案

                long long backup = curAns;

                while (winl > jobl) {

                    add(arr[--winl]);

                }

                ans[id] = curAns;

                curAns = backup;

                while (winl <= br[block]) {

                    del(arr[winl++]);//必须要先恢复到jobl

                }

            }

        }

    }

}

void prepare() {

    for (int i = 1; i <= n; i++) {

        sorted[i] = arr[i];

    }

    sort(sorted + 1, sorted + n + 1);

    cntv = 1;

    for (int i = 2; i <= n; i++) {

        if (sorted[cntv] != sorted[i]) {

            sorted[++cntv] = sorted[i];

        }

    }

    for (int i = 1; i <= n; i++) {

        arr[i] = kth(arr[i]);

    }

    blen = (int)sqrt(n);

    bnum = (n + blen - 1) / blen;

    for (int i = 1; i <= n; i++) {

        bi[i] = (i - 1) / blen + 1;

    }

    for (int i = 1; i <= bnum; i++) {

        br[i] = min(i \* blen, n);

    }

    sort(query + 1, query + m + 1, QueryCmp);

}

只删回滚莫队

// 只减回滚莫队针对的是增加操作不好维护答案信息 但是在减操作的时候  可以轻松维护答案

// 准备一个beforejob数组 表示进入这个块之前的答案  这个是为了方便在进入下一个块是更好的维护答案

// 然后右边界不断向左收缩 最初时块的左右边界应该是1到n  每次变化块的时候 右边界都要恢复到n  同时答案要变成进入这个块之前的答案

bool QueryCmp(Query &a, Query &b) {

    if (bi[a.l] != bi[b.l]) {

        return bi[a.l] < bi[b.l];

    }

    return b.r < a.r;//注意这里的排序方式

}

void compute() {

    for (int i = 1; i <= n; i++) {

        cnt[arr[i]]++;

    }

    mex = 0;

    while (cnt[mex] != 0) {

        mex++;

    }

    // 最初的窗口范围

    int winl = 1, winr = n;

    for (int block = 1, qi = 1; block <= bnum && qi <= m; block++) {

        // 将左边界修改正确

        while (winl < bl[block]) {

            del(arr[winl++]);

        }

        // 表示进入这个块之前的状态

        int beforeJob = mex;

        for (; qi <= m && bi[query[qi].l] == block; qi++) {

            int jobl = query[qi].l;

            int jobr = query[qi].r;

            int id = query[qi].id;

            while (winr > jobr) {

                del(arr[winr--]);

            }

            int backup = mex;

            while (winl < jobl) {

                del(arr[winl++]);

            }

            ans[id] = mex;

            // 保证每个问题查询时  左边界都是所在块的左边界

            mex = backup;

            while (winl > bl[block]) {

                add(arr[--winl]);

            }

        }

        while (winr < n) {

            add(arr[++winr]);

        }

        mex = beforeJob;

    }

}

void prepare() {

    blen = (int)sqrt(n);

    bnum = (n + blen - 1) / blen;

    for (int i = 1; i <= n; i++) {

        bi[i] = (i - 1) / blen + 1;

    }

    for (int i = 1; i <= bnum; i++) {

        bl[i] = (i - 1) \* blen + 1;

    }

    sort(query + 1, query + m + 1, QueryCmp);

}

int main() {

    cin >> n >> m;

    for (int i = 1; i <= n; i++) {

        cin >> arr[i];

    }

    for (int i = 1; i <= m; i++) {

        cin >> query[i].l >> query[i].r;

        query[i].id = i;

    }

    prepare();

    compute();

    for (int i = 1; i <= m; i++) {

        cout << ans[i] << '\n';

    }

    return 0;

}

树上莫队

// 查询的参数，jobl、jobr、lca、id

// 如果是类型1，那么lca == 0，表示空缺

// 如果是类型2，那么lca > 0，查询结果需要补充这个单点信息

struct Query {

   int l, r, lca, id;

};

// dep是深度 seg是括号序 st是节点开始序 ed是节点结束序

// stjump是倍增表 cntd是括号序列的长度

int dep[MAXN], seg[MAXN << 1];

int st[MAXN], ed[MAXN];

int stjump[MAXN][MAXP];

int cntd;

void dfs(int u, int fa) {

    dep[u] = dep[fa] + 1;

    seg[++cntd] = u;

    st[u] = cntd;

    stjump[u][0] = fa;

    for (int p = 1; p < MAXP; p++) {

        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];

    }

    for (int e = head[u], v; e > 0; e = nxt[e]) {

        v = to[e];

        if (v != fa) {

            dfs(v, u);

        }

    }

    seg[++cntd] = u;

    ed[u] = cntd;

}

bool QueryCmp(Query &a, Query &b) {

    if (bi[a.l] != bi[b.l]) {

        return bi[a.l] < bi[b.l];

    }

    return a.r < b.r;

}

// 窗口不管是加入还是删除node 只要遇到node就翻转信息即可

void invert(int node) {

    int val = color[node];

    if (vis[node]) {

        if (--cnt[val] == 0) {

            kind--;

        }

    } else {

        if (++cnt[val] == 1) {

            kind++;

        }

    }

    vis[node] = !vis[node];

}

void compute() {

    int winl = 1, winr = 0;

    for (int i = 1; i <= m; i++) {

        int jobl = query[i].l;

        int jobr = query[i].r;

        int lca = query[i].lca;

        int id = query[i].id;

        while (winl > jobl) {

            invert(seg[--winl]);

        }

        while (winr < jobr) {

            invert(seg[++winr]);

        }

        while (winl < jobl) {

            invert(seg[winl++]);

        }

        while (winr > jobr) {

            invert(seg[winr--]);

        }

        // 注意这里  如果需要额外添加一个lca  结束的时候不要忘记删除它  防止污染后面的数据

        if (lca > 0) {

            invert(lca);

        }

        ans[id] = kind;

        if (lca > 0) {

            invert(lca);

        }

    }

}

void prepare() {

    for (int i = 1; i <= n; i++) {

        sorted[i] = color[i];

    }

    sort(sorted + 1, sorted + n + 1);

    cntv = 1;

    for (int i = 2; i <= n; i++) {

        if (sorted[cntv] != sorted[i]) {

            sorted[++cntv] = sorted[i];

        }

    }

    for (int i = 1; i <= n; i++) {

        color[i] = kth(color[i]);

    }

    // 括号序列分块

    int blen = (int) sqrt(cntd);

    for (int i = 1; i <= cntd; i++) {

        bi[i] = (i - 1) / blen + 1;

    }

    sort(query + 1, query + m + 1, QueryCmp);

}

int main() {

    cin >> n >> m;

    for (int i = 1; i <= n; i++) {

        cin >> color[i];

    }

    for (int i = 1, u, v; i < n; i++) {

        cin >> u >> v;

        addEdge(u, v); addEdge(v, u);

    }

    dfs(1, 0);

    for (int i = 1, u, v, uvlca; i <= m; i++) {

        cin >> u >> v;

        if (st[v] < st[u]) {

            swap(u, v);

        }

        uvlca = lca(u, v);

        if (u == uvlca) {

            query[i].l = st[u];

            query[i].r = st[v];

            query[i].lca = 0;

        } else {

            query[i].l = ed[u];

            query[i].r = st[v];

            query[i].lca = uvlca;

        }

        query[i].id = i;

    }

}

莫队二次离线

// 第一次离线操作的时候 需要排序

// 第二次离线操作的时候  已经是按顺序执行的了 从1->n  从 n->1

// 二次离线主要需要我们推导公式  一般会变成前缀和、后缀和的形式

// 这是第二次离线操作

// 离线任务，x、l、r、op、id 位置x的任务列表用链式前向星表示

// headl[x]，x在l~r左侧的离线任务列表 headr[x]，x在l~r右侧的离线任务列表

int headl[MAXN],headr[MAXN],nextq[MAXN << 1];

int ql[MAXN << 1],qr[MAXN << 1],qop[MAXN << 1],qid[MAXN << 1];

int cntq;

// cnt[v] : 当前的数字v作为第二个数，之前出现的数字作为第一个数，产生多少k1二元组

int cnt[MAXV];

long long pre[MAXN],suf[MAXN];

bool QueryCmp(Query &a, Query &b) {

    if (bi[a.l] != bi[b.l]) {

        return bi[a.l] < bi[b.l];

    }

    return a.r < b.r;

}

void addLeftOffline(int x, int l, int r, int op, int id) {

    nextq[++cntq] = headl[x];

    headl[x] = cntq;

    ql[cntq] = l;

    qr[cntq] = r;

    qop[cntq] = op;

    qid[cntq] = id;

}

void addRightOffline(int x, int l, int r, int op, int id) {

    nextq[++cntq] = headr[x];

    headr[x] = cntq;

    ql[cntq] = l;

    qr[cntq] = r;

    qop[cntq] = op;

    qid[cntq] = id;

}

void prepare() {

    int blen = (int)sqrt(n);

    for (int i = 1; i <= n; i++) {

        bi[i] = (i - 1) / blen + 1;

    }

    sort(query + 1, query + m + 1, QueryCmp);

    for (int v = 0; v < MAXV; v++) {

        if (countOne(v) == k) {

            kOneArr[++cntk] = v;

        }

    }

}

void compute() {

    for (int i = 1; i <= n; i++) {

        pre[i] = pre[i - 1] + cnt[arr[i]];

        for (int j = 1; j <= cntk; j++) {

            cnt[arr[i] ^ kOneArr[j]]++;

        }

    }

    memset(cnt, 0, sizeof(cnt));

    for (int i = n; i >= 1; i--) {

        suf[i] = suf[i + 1] + cnt[arr[i]];

        for (int j = 1; j <= cntk; j++) {

            cnt[arr[i] ^ kOneArr[j]]++;

        }

    }

    int winl = 1, winr = 0;

    // 处理第一次离线操作  并收集第二次离线操作的任务列表

    for (int i = 1; i <= m; i++) {

        int jobl = query[i].l;

        int jobr = query[i].r;

        int id = query[i].id;

        if (winr < jobr) {

            addLeftOffline(winl - 1, winr + 1, jobr, -1, id);

            ans[id] += pre[jobr] - pre[winr];

        }

        if (winr > jobr) {

            addLeftOffline(winl - 1, jobr + 1, winr, 1, id);

            ans[id] -= pre[winr] - pre[jobr];

        }

        winr = jobr;

        if (winl > jobl) {

            addRightOffline(winr + 1, jobl, winl - 1, -1, id);

            ans[id] += suf[jobl] - suf[winl];

        }

        if (winl < jobl) {

            addRightOffline(winr + 1, winl, jobl - 1, 1, id);

            ans[id] -= suf[winl] - suf[jobl];

        }

        winl = jobl;

    }

    // 处理l类型的离线任务

    memset(cnt, 0, sizeof(cnt));

    for (int x = 0; x <= n; x++) {

        if (x >= 1) {

            for (int j = 1; j <= cntk; j++) {

                cnt[arr[x] ^ kOneArr[j]]++;

            }

        }

        for (int q = headl[x]; q > 0; q = nextq[q]) {

            int l = ql[q], r = qr[q], op = qop[q], id = qid[q];

            for (int j = l; j <= r; j++) {

                ans[id] += 1LL \* op \* cnt[arr[j]];

            }

        }

    }

    // 处理r类型的离线任务

    memset(cnt, 0, sizeof(cnt));

    for (int x = n + 1; x >= 1; x--) {

        if (x <= n) {

            for (int j = 1; j <= cntk; j++) {

                cnt[arr[x] ^ kOneArr[j]]++;

            }

        }

        for (int q = headr[x]; q > 0; q = nextq[q]) {

            int l = ql[q], r = qr[q], op = qop[q], id = qid[q];

            for (int j = l; j <= r; j++) {

                ans[id] += 1LL \* op \* cnt[arr[j]];

            }

        }

    }

}

int main() {

    for (int i = 2; i <= m; i++) {

        ans[query[i].id] += ans[query[i - 1].id];

    }

}

分块——散整块分析

// 这道题主要是散整块之间的关系讨论  主要是分析很难  以及准备预处理信息

// 查询从l到r之间的逆序对

// 1、l~r都是一个组   并且l是左边界  那么直接就是pre[r]

// 2、l~r是一个组  但是l不是左边界  那么答案就是pre[r]-pre[l-1]-f(l[i],l-1,l,r)

// 3、不是一个组  这是普遍情况 答案是（左散块[l..]内部逆序对 + 右散块[..r]内部逆序对 + 左散块 结合 右散块 的逆序对）+左散块对中间的+右散块对中间的+中间的逆序对个数

// pre[i] : 从所在块最左位置到i位置，有多少逆序对

int pre[MAXN];

// suf[i] : 从i位置到所在块最右位置，有多少逆序对

int suf[MAXN];

// cnt[i][j] : 前i块里<=j的数字个数

int cnt[MAXB][MAXN];

// dp[i][j] : 从第i块到第j块有多少逆序对

long long dp[MAXB][MAXB];

// 更靠左的第x块，从xl到xr范围上，选第一个数

// 更靠右的第y块，从yl到yr范围上，选第二个数

// x和y可以相等，但是xl..xr需要在yl..yr的左侧

// 返回逆序对数量

//时间复杂度是根号n

inline int f(int x, int xl, int xr, int y, int yl, int yr) {

    int ans = 0;

    for (int p1 = bl[x], p2 = bl[y] - 1, cnt = 0; p1 <= br[x]; p1++) {

        if (xl <= sortv[p1].i && sortv[p1].i <= xr) {

            while (p2 + 1 <= br[y] && sortv[p1].v > sortv[p2 + 1].v) {

                p2++;

                if (yl <= sortv[p2].i && sortv[p2].i <= yr) {

                    cnt++;

                }

            }

            ans += cnt;

        }

    }

    return ans;

}

long long query(int l, int r) {

    long long ans = 0;

    int lb = bi[l], rb = bi[r];

    if (lb == rb) {

        if (l == bl[lb]) {

            ans = pre[r];

        } else {

            ans = pre[r] - pre[l - 1] - f(lb, bl[lb], l - 1, lb, l, r);

        }

    } else {

        // 左散块[l..]内部逆序对 + 右散块[..r]内部逆序对 + 左散块 结合 右散块 的逆序对

        ans = suf[l] + pre[r] + f(lb, l, br[lb], rb, bl[rb], r);

        // 左散块中的arr[i]，作为第一个数

        // 中间整块中的某个数字，作为第二个数

        // 计算这样的逆序对数量

        // 注意因为题目给定的是排列！所以如下这么写没问题

        for (int i = l; i <= br[lb]; i++) {

            //查询中间的块中小于自己的

            ans += cnt[rb - 1][arr[i]] - cnt[lb][arr[i]];

        }

        // 中间整块中的某个数字，作为第一个数

        // 右散块中的arr[i]，作为第二个数

        // 计算这样的逆序对数量

        for (int i = bl[rb]; i <= r; i++) {

            //查询中间块中大于自己的

            ans += br[rb - 1] - bl[lb + 1] + 1 - (cnt[rb - 1][arr[i]] - cnt[lb][arr[i]]);

        }

        // 中间整块的逆序对

        ans += dp[lb + 1][rb - 1];

    }

    return ans;

}

// 注意调整块长

void prepare() {

    blen = (int)sqrt(n / 4);

    bnum = (n + blen - 1) / blen;

    for (int i = 1; i <= n; i++) bi[i] = (i - 1) / blen + 1;

    for (int i = 1; i <= bnum; i++) {

        bl[i] = (i - 1) \* blen + 1;

        br[i] = min(i \* blen, n);

    }

    for (int i = 1; i <= n; i++) {

        sortv[i].v = arr[i];

        sortv[i].i = i;

    }

    for (int i = 1; i <= bnum; i++) {

        sort(sortv + bl[i], sortv + br[i] + 1, NodeCmp);

    }

    //预处理各种信息

    for (int i = 1; i <= bnum; i++) {

        for (int j = bl[i]; j <= br[i]; j++) {

            cnt[i][arr[j]]++;

            if (j != bl[i]) {

                pre[j] = pre[j - 1] + sum(n) - sum(arr[j]);//前面大于自己的数字个数

            }

            add(arr[j], 1);

        }

        //注意清空树状数组

        for (int j = bl[i]; j <= br[i]; j++) {

            add(arr[j], -1);

        }

        for (int j = br[i]; j >= bl[i]; j--) {

            if (j != br[i]) {

                suf[j] = suf[j + 1] + sum(arr[j]);//后面小于自己的数字的个数

            }

            add(arr[j], 1);

        }

        //注意清空树状数组

        for (int j = bl[i]; j <= br[i]; j++) {

            add(arr[j], -1);

        }

        int tmp = 0;

        for (int j = 1; j <= n; j++) {

            tmp += cnt[i][j];

            cnt[i][j] = tmp + cnt[i - 1][j];

        }

    }

    //利用容斥进行计算

    //dp[l][r]=dp[l+1][r]+dp[l][r-1]-dp[l+1][r-1]+(l这个块和r这个块产生的逆序对)

    for (int l = bnum; l >= 1; l--) {

        dp[l][l] = pre[br[l]];

        for (int r = l + 1; r <= bnum; r++) {

            dp[l][r] = dp[l + 1][r] + dp[l][r - 1] - dp[l + 1][r - 1] + f(l, bl[l], br[l], r, bl[r], br[r]);

        }

    }

}

树上分块——树链剖分

void dfs1(int u, int f) {

    fa[u] = f;

    dep[u] = dep[f] + 1;

    siz[u] = 1;

    for (int e = head[u], v; e; e = nxt[e]) {

        v = to[e];

        if (v != f) {

            dfs1(v, u);

        }

    }

    for (int e = head[u]; e; e = nxt[e]) {

        int v = to[e];

        if (v != f) {

            siz[u] += siz[v];

            if (!son[u] || siz[son[u]] < siz[v]) {

                son[u] = v;

            }

        }

    }

}

void dfs2(int u, int t) {

    top[u] = t;

    dfn[u] = ++cntd;

    val[cntd] = arr[u];

    if (!son[u]) {

        return;

    }

    dfs2(son[u], t);

    for (int e = head[u], v; e; e = nxt[e]) {

        v = to[e];

        if (v != fa[u] && v != son[u]) {

            dfs2(v, v);

        }

    }

}

void query(int l, int r) {

    if (bi[l] == bi[r]) {

        for (int i = l; i <= r; i++) {

            ans[val[i]] = 1;

        }

    } else {

        for (int i = l; i <= br[bi[l]]; i++) {

            ans[val[i]] = 1;

        }

        for (int i = bl[bi[r]]; i <= r; i++) {

            ans[val[i]] = 1;

        }

        //块间操作

        for (int i = bi[l] + 1; i <= bi[r] - 1; i++) {

            ans |= bitSet[i];

        }

    }

}

// 这个其实就是树链剖分中的路径信息查询

void updateAns(int x, int y) {

    while (top[x] != top[y]) {

        if (dep[top[x]] < dep[top[y]]) {

            swap(x, y);

        }

        query(dfn[top[x]], dfn[x]);

        x = fa[top[x]];

    }

    query(min(dfn[x], dfn[y]), max(dfn[x], dfn[y]));

}

void prepare() {

    dfs1(1, 0);

    dfs2(1, 1);

    blen = (int)sqrt(n \* 20);//这道题的块间操作是位图的或   代价比较大

    bnum = (n + blen - 1) / blen;

    //这个是按照dfn序组织的

    for (int i = 1; i <= n; i++) {

        bi[i] = (i - 1) / blen + 1;

    }

    for (int i = 1; i <= bnum; i++) {

        bl[i] = (i - 1) \* blen + 1;

        br[i] = min(i \* blen, n);

        for (int j = bl[i]; j <= br[i]; j++) {

            bitSet[i][val[j]] = 1;

        }

    }

}

树上分块——随机撒点

// 随机撒点并不能被看成是严格的分块   只是设置了一些关键节点

// 通过随机性来保持时间复杂度

// 随机撒点 markNum表示关键点数量

int markNum;

// vis[i]表示i号节点是否已经是关键点

bool vis[MAXN];

// markNode[k] = i 表示第k个关键点是编号为i的节点

int markNode[MAXN];

// kthMark[i] = k 表示i号节点是第k个关键点，kthMark[i] = 0 表示i号节点是非关键点

int kthMark[MAXN];

// up[i] = j，表示i号节点是关键点，它往上跳到最近的关键点是j号节点

int up[MAXN];

// downSet[k]的含义，路径是[第k个的关键点 .. 最近的上方关键点)，沿途所有节点值组成的位图

bitset<MAXV> downSet[MAXB];

//建立倍增表

void dfs(int u, int fa) {

    dep[u] = dep[fa] + 1;

    stjump[u][0] = fa;

    for (int p = 1; p < MAXP; p++) {

        stjump[u][p] = stjump[stjump[u][p - 1]][p - 1];

    }

    for (int e = head[u]; e; e = nxt[e]) {

        if (to[e] != fa) {

            dfs(to[e], u);

        }

    }

}

//查询从x到xylca这条路上（不包括xylca）的所有点权的位图

void query(int x, int xylca) {

    //从起始点到第一个关键点

    while (kthMark[x] == 0 && x != xylca) {

        ans[arr[x]] = 1;

        x = stjump[x][0];

    }

    //从第一个关键点到最后一个关键点

    while (up[x] && dep[up[x]] > dep[xylca]) {

        ans |= downSet[kthMark[x]];

        x = up[x];

    }

    //最后一个关键点到xylca

    while (x != xylca) {

        ans[arr[x]] = 1;

        x = stjump[x][0];

    }

}

void updateAns(int x, int y) {

    int xylca = lca(x, y);

    query(x, xylca);//x到xylca

    query(y, xylca);//y到xylca

    ans[arr[xylca]] = 1;//加上xylca

}

void prepare() {

    dfs(1, 0);

    int len = (int)sqrt(n \* 10);

    markNum = (n + len - 1) / len;

    for (int b = 1, pick; b <= markNum; b++) {

        do {

            pick = rand() % n + 1;

        } while (vis[pick]);

        vis[pick] = true;

        markNode[b] = pick;

        kthMark[pick] = b;

    }

    for (int b = 1, cur; b <= markNum; b++) {

        downSet[b][arr[markNode[b]]] = 1;

        cur = stjump[markNode[b]][0];

        while (cur != 0) {

            if (kthMark[cur] > 0) {

                up[markNode[b]] = cur;

                break;

            } else {

                downSet[b][arr[cur]] = 1;

                cur = stjump[cur][0];

            }

        }

    }

}

int main() {

    srand(time(0));

}

分块——并查集

// 新建节点的编号

int cnt;

// 每个党派的代表节点

int fa[MAXN];

// 每个树上节点的父亲节点

int f[MAXM];

// 子树中叶子结点个数

int sz[MAXM];

// 这个节点所属党派

int col[MAXM];

// 新添加的军队数量

long long tag[MAXM];

// 区间总数量

long long sum;

void prepare(){

    blen=sqrt(n);

    bnum=(n+blen-1)/blen;

    for(int i=1;i<=n;i++){

        bi[i]=(i-1)/blen+1;

    }

    for(int i=1;i<=bnum;i++){

        bl[i]=(i-1)\*blen+1;

        br[i]=min(n,i\*blen);

    }

}

void compute(int l,int r){

    // 所有党派的信息最初代表都是0  表示这个块中没有这个党派

    for(int i=1;i<=m;i++){

        fa[i]=0;

    }

    // 收集这个块中的初始信息

    for(int i=l;i<=r;i++){

        tag[i-l+1]=army[i];

        col[i-l+1]=bel[i];

    }

    int len=r-l+1;

    sum=0;

    // 将以前的信息清空  col信息在设置这个节点的时候 就会被设置  所以不需要管

    for(int i=len+1;i<=cnt;i++){

        f[i]=0;

        tag[i]=0;

    }

    cnt=len;

    // 最初  全部都是叶子结点

    for(int i=1;i<=len;i++){

        f[i]=0;

        sum+=tag[i];

        sz[i]=1;

    }

    for(int i=1;i<=len;i++){

        // 第一次出现

        if(!fa[col[i]]){

            ++cnt;

            fa[col[i]]=cnt;

            sz[cnt]=1;

            col[cnt]=col[i];

            f[i]=cnt;

        }

        else{

            int u=fa[col[i]];

            f[i]=u;

            ++sz[u];

        }

    }

    for(int i=1;i<=q;i++){

        if(ql[i]>r||qr[i]<l){

            continue;

        }

        // 整块包含

        if(ql[i]<=l&&r<=qr[i]){

            if(op[i]==1){

                if(qx[i]==qy[i]||!fa[qx[i]]){

                    continue;

                }

                else if(!fa[qy[i]]){

                    col[fa[qx[i]]]=qy[i];

                    fa[qy[i]]=fa[qx[i]];

                    fa[qx[i]]=0;

                }

                else{

                    // 合并两个根节点  实际上就是将两个党派合并

                    int u=++cnt;

                    f[fa[qx[i]]]=u;

                    f[fa[qy[i]]]=u;

                    col[u]=qy[i];

                    sz[u]=sz[fa[qx[i]]]+sz[fa[qy[i]]];

                    fa[qy[i]]=u;

                    fa[qx[i]]=0;

                }

            }

            else if(op[i]==2){

                if(fa[qx[i]]){

                    // 懒更新  不要忘记更新sum

                    tag[fa[qx[i]]]+=qy[i];

                    sum+=1ll\*qy[i]\*sz[fa[qx[i]]];

                }

            }

            else{

                ans[i]+=sum;

            }

        }

        else{

            // 将所有信息全部下发

            for(int u=cnt;u;u--){

                if(f[u]){

                    tag[u]+=tag[f[u]];

                    col[u]=col[f[u]];

                }

            }

            for(int u=len+1;u<=cnt;u++){

                tag[u]=0,f[u]=0;

            }

            cnt=len;

            for(int j=1;j<=len;j++){

                fa[col[j]]=0;

                f[j]=0;

            }

            int lt=max(ql[i],l)-l+1;

            int rt=min(qr[i],r)-l+1;

            // 暴力维持信息

            if(op[i]==1){

                for(int u=lt;u<=rt;u++){

                    if(col[u]==qx[i]){

                        col[u]=qy[i];

                    }

                }

            }

            else if(op[i]==2){

                for(int u=lt;u<=rt;u++){

                    if(col[u]==qx[i]){

                        sum+=qy[i];

                        tag[u]+=qy[i];

                    }

                }

            }

            else{

                for(int u=lt;u<=rt;u++){

                    ans[i]+=tag[u];

                }

            }

            // 重新建立树

            for(int p=1;p<=len;p++){

                if(!fa[col[p]]){

                    fa[col[p]]=p;

                }

                else{

                    int u=fa[col[p]];

                    if(u<=len){

                        ++cnt;

                        fa[col[p]]=cnt;

                        f[u]=cnt;

                        u=cnt;

                        sz[cnt]=1;

                        col[cnt]=col[p];

                    }

                    f[p]=u;

                    ++sz[u];

                }

            }

        }

    }

}

int main()

{

    for(int i=1;i<=bnum;i++){

        compute(bl[i],br[i]);

    }

}

扩展KMP

// 扩展KMP模版，又称Z算法或Z函数

// 给定两个字符串a、b，求出两个数组

// b与b每一个后缀串的最长公共前缀长度，z数组

// b与a每一个后缀串的最长公共前缀长度，e数组

#include<bits/stdc++.h>

using namespace std;

const int MAXN = 20000001;

int n,m;

char a[MAXN];

char b[MAXN];

// b与b每一个后缀串的最长公共前缀长度

int z[MAXN];

// b与a每一个后缀串的最长公共前缀长度

int e[MAXN];

// 这个过程和Manacher算法大致流程是一样的

// 判断此时的“对称点”和此时扩展到的右边界的位置的关系

// 对于右边界r和扩展到这个右边界的c而言 对称关系有c-0  c+1-1 c+2-2 '''

// 而对于此时的i而言  如果i在r的范围内  我们想要求的是的i-0 i+1-1 i+2-2 ''' i+len-len 的最大len

// 对于c而言  存在i-i-c i+1-i-c+1 '''

// 那么它的对称点是 0-i-c  1-i-c+1 '''  也就是对称点为i-c

void zArray(char\* s, int n) {

    z[0] = n;

    for (int i = 1, c = 1, r = 1, len; i < n; i++) {

        len = r > i ? min(r - i, z[i - c]) : 0;

        while (i + len < n && s[i + len] == s[len]) {

            len++;

        }

        if (i + len > r) {

            r = i + len;

            c = i;

        }

        z[i] = len;

    }

}

void zarray(){

    z[1]=n;

    for(int i=2,c=2,r=2,len;i<=n;i++){

        len=r>i?min(r-i,z[i-c+1]):0;

        while (i + len <= n && s[i + len] == s[len+1]) {

            len++;

        }

        if (i + len > r) {

            r = i + len;

            c = i;

        }

        z[i] = len;

    }

}

// e数组需要在b字符串的z数组已经求出来的情况下进行计算得到

// 这个过程和求z数组大同小异

void eArray(char\* a, char\* b, int n, int m) {

    for (int i = 0, c = 0, r = 0, len; i < n; i++) {

        len = r > i ? min(r - i, z[i - c]) : 0;

        while (i + len < n && len < m && a[i + len] == b[len]) {

            len++;

        }

        if (i + len > r) {

            r = i + len;

            c = i;

        }

        e[i] = len;

    }

}

int main()

{

    scanf("%s",a);

    n=strlen(a);

    scanf("%s",b);

    m=strlen(b);

    zArray(b,m);

    eArray(a,b,n,m);

    cout<<eor(z,m)<<endl;

    cout<<eor(e,n)<<endl;

    return 0;

}

线段树优化建图

// https://www.luogu.com.cn/problem/CF786B

// 这道题是需要线段树优化建图的模板题  我们准备两个线段树

// 第一个线段树父亲节点到儿子节点的距离是0  第二个是儿子节点到父亲节点的距离是0

// 那么第二个操作 就是节点向正向图连边  表示达到这个区间的新边

// 第三个操作就是在反向图上建边  表示这个区间到达这个节点

// 其实两个线段树的叶节点代表的节点是一样的  他们之间距离是0（如果点相同的话）

// 这道题直接将两个树使用同样的一批叶节点  也很巧妙

// P6348也是线段树优化建图

int nodecnt;

int root1,root2;

int ls1[MAXM];

int rs1[MAXM];

int ls2[MAXM];

int rs2[MAXM];

int build1(int l,int r){

    if(l==r){

        return l;

    }

    else{

        int mid=(l+r)>>1;

        int now=++nodecnt;

        ls1[now]=build1(l,mid);

        rs1[now]=build1(mid+1,r);

        addedge(now,ls1[now],0);

        addedge(now,rs1[now],0);

        return now;

    }

}

int build2(int l,int r){

    if(l==r){

        return l;

    }

    else{

        int mid=(l+r)>>1;

        int now=++nodecnt;

        ls2[now]=build2(l,mid);

        rs2[now]=build2(mid+1,r);

        addedge(ls2[now],now,0);

        addedge(rs2[now],now,0);

        return now;

    }

}

void update1(int jobl,int jobr,int jobv,int jobw,int l,int r,int i){

    if(jobl<=l&&r<=jobr){

        addedge(jobv,i,jobw);

    }

    else{

        int mid=(l+r)>>1;

        if(jobl<=mid){

            update1(jobl,jobr,jobv,jobw,l,mid,ls1[i]);

        }

        if(jobr>mid){

            update1(jobl,jobr,jobv,jobw,mid+1,r,rs1[i]);

        }

    }

}

void update2(int jobl,int jobr,int jobv,int jobw,int l,int r,int i){

    if(jobl<=l&&r<=jobr){

        addedge(i,jobv,jobw);

    }

    else{

        int mid=(l+r)>>1;

        if(jobl<=mid){

            update2(jobl,jobr,jobv,jobw,l,mid,ls2[i]);

        }

        if(jobr>mid){

            update2(jobl,jobr,jobv,jobw,mid+1,r,rs2[i]);

        }

    }

}

AC自动机

// 问题的关键就在于如何解决 第x个单词在第y个单词中出现的次数

// 这个问题的推导的逻辑如下：

// 对于AC自动机而言 1、AC自动机的每个节点对应一个字符串前缀

//                 2、节点u的fail指针指向v，表示v对应字符串是u对应字符串的最长后缀

// 对于通过fail指针建立的反向fail树而言：若节点v在节点u的Fail树子树中，则u对应的字符串是v对应字符串的后缀

// 比如u = "ab" → v = "xab"   ---->  则"ab"是"xab"的后缀

// x在y中出现的次数 = y的所有前缀中，以x为后缀的前缀数量(当然 你也可以说是所有后缀中 以x为前缀的数量  但是我们这里用到的是前面这种定义)

// 从自动机的角度来看 我们假设 node\_x是x的结束节点 path\_y是y在Trie树上的路径（根节点到node\_y）

// x在y中出现次数 = path\_y上所有节点v中，满足 v 在 node\_x 的 Fail树的子树中的数量

// 在某个节点的子树中出现的  这一类问题可以转换为  dfn序   在线性数组上处理

// 利用树状数组可以快速查询

// 于是 我们重新走建立AC自动机的过程  在这个过程中统计答案

// 对于fail树而言  这个节点的所有子树所对应的字符串一定包含这个节点对应的字符串  因此常常被用来收集词频

int cnt;

int trie[MAXN][26];

int fail[MAXN];

//这里比一般的AC自动机多出来的部分 是由于建立自动机时的特殊情况所决定的  用于B、P操作

int fa[MAXN];

int endpos[MAXN];//记录这个是为了在统计答案的时候更方便  具体参见代码

void setfail(){

    queue<int>q;

    for(int i=0;i<26;i++){

        if(trie[0][i]){

            q.push(trie[0][i]);

        }

    }

    while(!q.empty()){

        int u=q.front();

        q.pop();

        for(int i=0;i<26;i++){

            if(trie[u][i]==0){

                trie[u][i]=trie[fail[u]][i];

            }

            else{

                fail[trie[u][i]]=trie[fail[u]][i];

                q.push(trie[u][i]);

            }

        }

    }

}

int main(){

    setfail();

    //建立fail树  反向建树

    for(int i=1;i<=cnt;i++){

        addedge(fail[i],i);

    }

    //得到dfn序

    dfs(0);

    cin>>n;

    for(int i=1;i<=n;i++){

        int x,y;

        cin>>x>>y;

        addedge(y,x,i);

    }

    //重新跑一遍trie  统计答案

    cur=0; id=0;

    for(char c:s){

        if(c=='B'){

            add(dfn[cur],-1);//移除当前节点

            cur=fa[cur];

        }

        else if(c=='P'){

            id++;

            for(int i=headq[id];i;i=nxtq[i]){

                int v=toq[i];

                int w=idq[i];

                int node=endpos[v];

                ans[w]=query(dfn[node]+sz[node]-1)-query(dfn[node]-1);

            }

        }

        else{

            int path=c-'a';

            cur=trie[cur][path];

            add(dfn[cur],1);//当前节点加入树状数组中

        }

    }

}

KMP算法——1based

// https://www.luogu.com.cn/problem/P3375

//这是一个1based的kmp算法

//在这里  next数组表示的是从1~i位置最大的前后缀匹配长度

void calcnext(){

    nxt[1]=0;

    for(int i=2,j=0;i<=m;i++){

        // 当不匹配时，通过next数组回溯

        while(j>0&&s2[i]!=s2[j+1]){

            j=nxt[j];

        }

        // 如果当前字符匹配，j增加

        if(s2[i]==s2[j+1]){

            j++;

        }

        // 设置当前位置的next值

        nxt[i]=j;

    }

}

void kmp(){

    n=strlen(s1+1);

    m=strlen(s2+1);

    calcnext();

    for(int i=1,j=0;i<=n;i++){

        // 当不匹配时，通过next数组回溯

        while(j>0&&s1[i]!=s2[j+1]){

            j=nxt[j];

        }

        // 如果当前字符匹配，j增加

        //否则 说明无法匹配上当前的s1字符  只能向下移动一位

        if(s1[i]==s2[j+1]){

            j++;

        }

        // 完全匹配成功

        if (j == m) {

            // 输出匹配位置(1-based)

            cout << i - m + 1 << endl;

            // 继续寻找下一个匹配

            j = nxt[j];

        }

    }

}

网络流（时间复杂度O(V² \* E) 特例（二分图匹配）：O(E \* √V)）

// 如果某一时刻我们已经知道边 (u, v) 已经增广到极限（边 (u, v) 已无剩余容量或 v 的后侧已增广至阻塞），

// 则 u 的流量没有必要再尝试流向出边 (u, v)。据此，对于每个结点 u，我们维护 u 的出边表中第一条还有必要尝试的出边。

// 习惯上，我们称维护的这个指针为当前弧，称这个做法为当前弧优化。

// 多路增广是 Dinic 算法的一个常数优化——如果我们在层次图上找到了一条从 s 到 t 的增广路 p，

// 则接下来我们未必需要重新从 s 出发找下一条增广路，而可能从 p 上最后一个仍有剩余容量的位置出发寻找一条岔路进行增广。

// 考虑到其与回溯形式的一致性，这一优化在 DFS 的代码实现中也是自然的。

int n,m,s,t;

int dep[MAXN];//记录深度数组

int iter[MAXN];//当前弧数组 记录这个节点有效访问的第一条边的编号

int cnt=2;

int head[MAXN];int nxt[MAXM];int to[MAXM];int now[MAXM];//表示现在的流量

int cap[MAXM];//表示流量限制

void addedge(int u,int v,int w){

    nxt[cnt]=head[u]; to[cnt]=v; cap[cnt]=w; now[cnt]=0; head[u]=cnt++;

    nxt[cnt]=head[v]; to[cnt]=u; cap[cnt]=0; now[cnt]=0; head[v]=cnt++;

}

//BFS构建分层图，并判断是否存在增广路径

bool bfs(){

    for(int i=1;i<=n;i++){

        dep[i]=-1;

    }

    queue<int>q;

    dep[s]=0;

    q.push(s);

    while(!q.empty()){

        int u=q.front();

        q.pop();

        for(int i=head[u];i;i=nxt[i]){

            int v=to[i];

            int w=cap[i];

            int k=now[i];

            if(dep[v]<0&&(k<w)){

                dep[v]=dep[u]+1;

                q.push(v);

            }

        }

    }

    return dep[t]>=0;// 如果汇点未被访问到，说明无增广路径

}

// DFS寻找增广路径（多路增广）

//多路增广的含义是来到了一个节点 不仅仅是只去一条边增广  而是去多条边一起增广

int dfs(int u,int f){

    //表示当前来到了u节点 有f的流量可供使用  f就是目前这条路最多可以消耗掉的流量

    if(u==t){

        return f;

    }

    int flow=0;

    for(int &i=iter[u];i;i=nxt[i]){

        //注意这里是引用  iter会随着i发生变化

        int v=to[i];

        int w=cap[i];

        int k=now[i];

        if(dep[u]+1==dep[v]&&k<w){

            int d=dfs(v,min(f,w-k));

            if(d>0){

                now[i]+=d;

                now[i^1]-=d;//更新反向边

                flow+=d;

                f-=d;

                if(f==0){

                    break;

                }

            }

        }

    }

    return flow;

}

int maxflow(){

    int flow=0;

    while(bfs()){

        //当前弧全部初始化为最初值

        for(int i=1;i<=n;i++){

            iter[i]=head[i];

        }

        int maxflow;

        while((maxflow=dfs(s,INF))>0){

            flow+=maxflow;

        }

    }

    return flow;

}

最小费用最大流——spfa（时间复杂度O(F × V × E) f为最大流值）

int head[MAXV];

int nxt[MAXE];

int to[MAXE];

int cap[MAXE];

int cost[MAXE];

int cnt=2;

// 最小费用最大流相关数组

int dist[MAXV], pre[MAXV], preEdge[MAXV];

bool inQueue[MAXV];

// 最小费用最大流

int minCostMaxFlow() {

    int totalFlow = 0, totalCost = 0;

    while (true) {

        // 初始化距离和前驱数组

        memset(dist, 0x3f, sizeof(dist));

        memset(inQueue, false, sizeof(inQueue));

        memset(pre, 0, sizeof(pre));

        memset(preEdge, 0, sizeof(preEdge));

        queue<int> q;

        q.push(s);

        dist[s] = 0;

        inQueue[s] = true;

        // SPFA找最短路

        while (!q.empty()) {

            int u = q.front();

            q.pop();

            inQueue[u] = false;

            for (int i = head[u]; i ; i = nxt[i]) {

                int v = to[i];

                if (cap[i] > 0 && dist[v] > dist[u] + cost[i]) {

                    dist[v] = dist[u] + cost[i];

                    pre[v] = u;

                    preEdge[v] = i;

                    if (!inQueue[v]) {

                        inQueue[v] = true;

                        q.push(v);

                    }

                }

            }

        }

        // 如果无法到达汇点，退出

        if (dist[t] == INF) break;

        // 计算增广路径上的最小容量

        int flow = INF;

        for (int u = t; u != s; u = pre[u]) {

            int edgeIdx = preEdge[u];

            flow = min(flow, cap[edgeIdx]);

        }

        // 更新流量和费用

        totalFlow += flow;

        totalCost += flow \* dist[t];

        // 更新残余网络

        for (int u = t; u != s; u = pre[u]) {

            int edgeIdx = preEdge[u];

            cap[edgeIdx] -= flow;

            cap[edgeIdx ^ 1] += flow; // 反向边

        }

    }

    return -totalCost; // 返回最大费用

}

最小费用最大流——Dijkstra（时间复杂度O(F×(V+E)logV) f为最大流值）

// 最小费用最大流问题是每条边有流量  并且每条边的流量有单位价格

// 要在满足流量最大的情况下  使得花费最小的解决方案

int n, m, s, t;

int mincost, maxflow;

int pot[MAXV];int dis[MAXV];int pre[MAXV];bool vis[MAXV];

queue<int> q;bool in[MAXV];

int head[MAXV], nxt[MAXE], to[MAXE], cap[MAXE], cost[MAXE], cnt = 2;

struct cmp {

    bool operator()(pair<int,int> a, pair<int,int> b) {

        return a.first > b.first;  // 按距离从小到大排序

    }

};

priority\_queue<pair<int,int>, vector<pair<int,int>>, cmp> heap;

inline void addedge(int u, int v, int w, int c) {

    nxt[cnt] = head[u]; to[cnt] = v; cap[cnt] = w; cost[cnt] = c; head[u] = cnt++;

    nxt[cnt] = head[v]; to[cnt] = u; cap[cnt] = 0; cost[cnt] = -c; head[v] = cnt++;

}

void spfa(int s) {

    // 初始化势能为无穷大

    for(int i = 1; i <= n; i++) {

        pot[i] = INF;

    }

    // 源点入队

    q.push(s);

    pot[s] = 0;

    in[s] = true;

    while(!q.empty()) {

        int u = q.front();

        q.pop();

        in[u] = false;

        // 遍历所有出边

        for(int i = head[u]; i; i = nxt[i]) {

            int v = to[i];

            int w = cap[i];     // 剩余容量

            int c = cost[i];    // 单位费用

            // 如果有容量且可以松弛

            if(w > 0 && pot[u] + c < pot[v]) {

                pot[v] = pot[u] + c;  // 更新势能

                // 如果节点v不在队列中，加入队列

                if(!in[v]) {

                    q.push(v);

                    in[v] = true;

                }

            }

        }

    }

}

bool dijkstra() {

    for(int i = 1; i <= n; i++) {

        dis[i] = INF;

        pre[i] = -1;

        vis[i] = false;

    }

    dis[s] = 0;

    heap.push({0, s});

    while(!heap.empty()) {

        int u = heap.top().second;

        heap.pop();

        if(vis[u]) continue;

        vis[u] = true;

        for(int i = head[u]; i; i = nxt[i]) {

            int v = to[i];

            int residual = cap[i];

            int c\_original = cost[i]; // 保存原始费用

            // 跳过容量为0的边

            if(residual <= 0) continue;

            // 使用势函数调整边权（关键步骤）

            int adjusted\_cost = c\_original + pot[u] - pot[v];

            // 如果可以松弛

            if(dis[u] + adjusted\_cost < dis[v]) {

                dis[v] = dis[u] + adjusted\_cost;

                pre[v] = i;  // 记录前驱边索引

                heap.push({dis[v], v});

            }

        }

    }

    return dis[t] < INF;

}

void mincostmaxflow() {

    spfa(s);

    maxflow = 0;

    mincost = 0;

    while(dijkstra()) {

        // 计算增广路径上的最小容量

        int flow = INF;

        for(int u = t; u != s; u = to[pre[u] ^ 1]) {

            flow = min(flow, cap[pre[u]]);

        }

        //  更新流量和费用

        maxflow += flow;

        // 实际费用 = 调整后距离 + 汇点势能

        mincost += flow \* (dis[t] + pot[t]);

        // 更新路径上的边容量

        for(int u = t; u != s; u = to[pre[u] ^ 1]) {

            cap[pre[u]] -= flow;      // 正向边减流量

            cap[pre[u] ^ 1] += flow;  // 反向边加流量

        }

        //  更新势函数（为下一轮准备）

        for(int i = 1; i <= n; i++) {

            if(dis[i] < INF) {

                pot[i] += dis[i];  // 关键：更新势能

            }

        }

    }

}

上下界最大流——无源汇可行流

// 这道题是没有源点、汇点  每条边的流量有一个上下界   查询是否存在可行流

// 这道题的做法是根据每条边的下界处理好in数组  即每个节点净流入量

// 我们根据每条边的上界建图  虚建源点和汇点

// 如果净流量大于0  那么源点向这个点流入流量  否则向虚点流出流量

// 然后跑最大流  存在可行流的条件是最大流是源点流出的流量之和

int n,m,s,t; int in[MAXN];

int main(){

    n=read(),m=read();

    s=n+1,t=s+1;

    for(int i=1;i<=m;i++){

        edge[i].u=read(),edge[i].v=read(),edge[i].lower=read(),edge[i].higher=read();

    }

    for(int i=1;i<=m;i++){

        int u=edge[i].u; int v=edge[i].v;

        int lower=edge[i].lower; int higher=edge[i].higher;

        edgeid[i]=cnt;

        addedge(u,v,higher-lower);

        in[u]-=lower; in[v]+=lower;

    }

    // 添加附加源汇的边  并记录附加边总权值

    int need=0;

    for(int i=1;i<=n;i++){

        if(in[i]>0){

            addedge(s,i,in[i]);

            need+=in[i];

        }

        else{

            addedge(i,t,-in[i]);

        }

    }

    int flow=maxflow();

    if(flow!=need){

        cout<<"NO"<<endl;

    }else{

        cout<<"YES"<<endl;

        for(int i=1;i<=m;i++){

            int actual=now[edgeid[i]]+edge[i].lower;

            cout<<actual<<endl;

        }

    }

}

上下界网络流——有源汇最大流

// 有源汇的最大流的步骤是首先建立出增量网络流图  要加上一条从t到s容量为INF的边

// 根据每个节点的in 添加附加源汇点的边  求出可行流

// 如果可行流存在  那么删除从t到s的那条边 在残余网络上跑最大流

// 此时的最大流加上可行流就是答案

// 最小流就是在残余网络上计算退流 从t到s  用可行流减去退流即可

int n,m,s,t;

int ss,tt;// 附加源点、汇点

int dep[MAXV];

int iter[MAXV];

int cnt=2;

int head[MAXV];

int nxt[MAXE];

int to[MAXE];

int now[MAXE];

int cap[MAXE];

int ts\_id; // 记录t->s边的编号

// 每个节点的下界净流量

int in[MAXV];

int main(){

    n=read(),m=read(),s=read(),t=read();

    ss=n+1,tt=ss+1;

    for(int i=1;i<=m;i++){

        int u,v,l,r;

        u=read(),v=read(),l=read(),r=read();

        addedge(u,v,r-l);

        in[u]-=l;

        in[v]+=l;

    }

    // 记录添加t->s边前的cnt  添加t->s边，容量INF

    ts\_id=cnt;

    addedge(t,s,INF);

    // 建立所有的附加边

    int all=0;

    for(int i=1;i<=n;i++){

        if(in[i]>0){

            addedge(ss,i,in[i]);

            all+=in[i];

        }

        else if(in[i]<0){

            addedge(i,tt,-in[i]);

        }

    }

    int flow1=maxflow(ss,tt);

    if(flow1!=all){

        printf("please go home to sleep");

    }

    else{

        // 记录可行流值

        int f0=now[ts\_id];

        // 删除t->s边：将容量和当前流量设为0

        // 附加边不需要改变 因为他们的剩余流量一定是0

        cap[ts\_id] = 0;

        cap[ts\_id ^ 1] = 0;

        now[ts\_id] = 0;

        now[ts\_id ^ 1] = 0;

        // 在残余网络上从s到t跑最大流

        int flow2 = maxflow(s, t);

        // 如果求的是最小流 那么就是跑一下退流  用可行流减去退流即可

        // // 在残余网络上从t到s跑最大流（退流）

        // int flow\_back = maxflow(t, s);

        // // 最小流 = 可行流 - 退流

        // printf("%lld\n\n", f0 - flow\_back);

        printf("%d\n", f0 + flow2);

    }

    return 0;

}

二分图匹配——HK算法

// https://www.luogu.com.cn/problem/P3386

// HK算法的流程是：

// 1. BFS: 构建分层图，找到所有最短增广路

// 2. DFS: 沿着分层图进行多路增广

// 1、2操作循环进行  直到找不到增广路为止

// 时间复杂度是O(n\*根号m)  n为节点数  m为边数

int matchlt[MAXN];int matchrt[MAXN];

int dislt[MAXN];int disrt[MAXN];bool vis[MAXN];

bool bfs(){

    queue<int>q;

    for(int i=1;i<=n;i++){

        dislt[i]=INF;

    }

    for(int i=1;i<=m;i++){

        disrt[i]=INF;

    }

    for(int i=1;i<=n;i++){

        if(matchlt[i]==0){

            dislt[i]=0;

            q.push(i);

        }

        else{

            dislt[i]=INF;

        }

    }

    bool flag=false;

    while(!q.empty()){

        int u=q.front();

        q.pop();

        for(int i=head[u];i;i=nxt[i]){

            int v=to[i];

            if(disrt[v]==INF){

                disrt[v]=dislt[u]+1;

                if(matchrt[v]==0){

                    flag=true;

                }

                else{

                    dislt[matchrt[v]]=disrt[v]+1;

                    q.push(matchrt[v]);

                }

            }

        }

    }

    return flag;

}

bool dfs(int u){

    for(int i=head[u];i;i=nxt[i]){

        int v=to[i];

        if(disrt[v]==dislt[u]+1){

            disrt[v]=INF;

            if(matchrt[v]==0||dfs(matchrt[v])){

                matchlt[u]=v;

                matchrt[v]=u;

                return true;

            }

        }

    }

    return false;

}

int HK(){

    int match=0;

    while(bfs()){

        for(int i=1;i<=n;i++){

            if(matchlt[i]==0&&dfs(i)){

                match++;

            }

        }

    }

    return match;

}

int main()

{

    n=read(),m=read(),k=read();

    for(int i=1;i<=k;i++){

        int u,v;

        u=read(),v=read();

        addedge(u,v);

    }

}

虚树

// 这道题主要是在虚树上进行树形dp  关键是转移关系

// 关键点数组

int arr[MAXN];

// 标记节点是否是关键点

bool isKey[MAXN];

// 第一种建树方式

int tmp[MAXN << 1];

bool cmp(int x, int y) {

    return dfn[x] < dfn[y];

}

// 二次排序 + LCA连边的方式建立虚树

int buildVirtualTree1() {

    sort(arr + 1, arr + k + 1, cmp);

    int len = 0;

    for (int i = 1; i < k; i++) {

        tmp[++len] = arr[i];

        tmp[++len] = getLca(arr[i], arr[i + 1]);

    }

    tmp[++len] = arr[k];

    sort(tmp + 1, tmp + len + 1, cmp);

    int unique = 1;

    for (int i = 2; i <= len; i++) {

        if (tmp[unique] != tmp[i]) {

            tmp[++unique] = tmp[i];

        }

    }

    cntv = 0;

    // 清空用到的head数组

    for (int i = 1; i <= unique; i++) {

        headv[tmp[i]] = 0;

    }

    for (int i = 1; i < unique; i++) {

        addEdgeV(getLca(tmp[i], tmp[i + 1]), tmp[i + 1]);

    }

    return tmp[1];

}

Tarjan——边双连通分量

// 链式前向星存图

int head[MAXN];     // head[u]表示节点u的第一条边的编号

int Next[MAXM<<1];  // Next[i]表示第i条边的下一条边

int to[MAXM<<1];    // to[i]表示第i条边指向的节点

int cnt = 1;        // 边的计数器，从1开始（方便处理反向边）

// Tarjan算法相关变量

int dfn[MAXN];      // dfn[u]表示节点u的DFS序（时间戳）

int low[MAXN];      // low[u]表示u及其子树能追溯到的最早祖先的dfn

int st[MAXN], top;  // 栈，用于存储当前DFS路径上的节点

int color, Time;    // color记录边双的数量，Time是时间戳计数器

int col[MAXN];      // 可用于染色，但本题未使用

int n, m;           // 节点数和边数

vector<int> ans[MAXN]; // 存储每个边双的节点

// Tarjan算法求边双连通分量

void tarjan(int u, int f, int tot) {

    dfn[u] = low[u] = ++Time; // 初始化dfn和low

    st[++top] = u;            // 节点u入栈

    // 遍历u的所有邻接边

    for(int i = head[u]; i; i = Next[i]) {

        int v = to[i];

        // 如果是父边且不是反向边（处理无向图的重边）

        if(v == f && (i - 1) == ((tot - 1) ^ 1)) {

            continue;

        }

        if(!dfn[v]) {         // 如果v未被访问

            tarjan(v, u, i);  // 递归访问v

            low[u] = min(low[u], low[v]); // 更新low[u]

        } else {              // 如果v已被访问（处理回边）

            low[u] = min(low[u], dfn[v]);

        }

    }

    // 如果dfn[u] == low[u]，说明u是边双的根

    if(dfn[u] == low[u]) {

        color++;              // 新的边双编号

        ans[color].push\_back(u); // 加入根节点

        // 弹出栈中节点，直到u被弹出（构成一个边双）

        while(st[top] != u) {

            ans[color].push\_back(st[top--]);

        }

        top--;                // 弹出u

    }

}

int main() {

    n = read(), m = read();

    // 读入边并建图（无向图）

    for(int i = 1; i <= m; i++) {

        int u = read(), v = read();

        addedge(u, v);

        addedge(v, u); // 无向图加双向边

    }

    // 对所有未访问的节点运行Tarjan

    for(int i = 1; i <= n; i++) {

        if(!dfn[i]) {

            tarjan(i, 0, 0); // 0表示没有父节点和父边

        }

    }

    // 输出边双数量和每个边双的节点

    cout << color << endl;

    for(int i = 1; i <= color; i++) {

        cout << ans[i].size() << ' ';

        for(int j = 0; j < ans[i].size(); j++) {

            cout << ans[i][j] << ' ';

        }

        cout << endl;

    }

    return 0;

}

Tarjan——点双连通分量

// 链式前向星存图

int head[MAXN];     // head[u] 表示节点u的第一条边的编号

int Next[MAXM<<1];  // Next[i] 表示第i条边的下一条边

int to[MAXM<<1];    // to[i] 表示第i条边指向的节点

int cnt=1;          // 边的计数器，从1开始（方便处理反向边）

// Tarjan算法相关变量

int dfn[MAXN];      // dfn[u] 表示节点u的DFS序（时间戳）

int low[MAXN];      // low[u] 表示u及其子树能追溯到的最早祖先的dfn

int st[MAXN], top;  // 栈，用于存储当前DFS路径上的节点

int color, Time;    // color记录点双的数量，Time是时间戳计数器

int n, m;           // 节点数和边数

vector<int> ans[MAXN]; // 存储每个点双的节点

// Tarjan算法求点双连通分量

void tarjan(int u, int f){

    dfn[u] = low[u] = ++Time; // 初始化dfn和low

    st[++top] = u;            // 节点u入栈

    int child = 0;            // 记录u在DFS树中的子节点数量

    // 遍历u的所有邻接节点

    for(int i = head[u]; i; i=Next[i]){

        int v = to[i];

        if(!dfn[v]){          // 如果v未被访问

            child++;

            tarjan(v, u);     // 递归访问v

            low[u] = min(low[u], low[v]); // 更新low[u]

            // 如果low[v] >= dfn[u]，说明u是割点，开始弹出点双

            if(low[v] >= dfn[u]){

                color++;      // 新的点双编号

                // 弹出栈中节点，直到v被弹出（包括v）

                while(st[top] != v){ // 注意：st[top+1]是哨兵

                    ans[color].push\_back(st[top--]);

                }

                ans[color].push\_back(st[top--]);//弹出 v，并显式加入 u

                ans[color].push\_back(u); // 割点u属于当前点双

            }

        }

        else if(v != f){      // 如果v已被访问且不是父节点（处理回边）

            low[u] = min(low[u], dfn[v]);

        }

    }

    // 特判：u是孤立点（DFS树的根且没有子节点）

    if(f == 0 && child == 0){

        ans[++color].push\_back(u);

    }

}

int main(){

    n = read(), m = read();

    // 读入边并建图

    for(int i=1; i<=m; i++){

        int u = read(), v = read();

        addedge(u, v);

        addedge(v, u); // 无向图，加双向边

    }

    // 对所有未访问的节点运行Tarjan

    for(int i=1; i<=n; i++){

        if(!dfn[i]){

            tarjan(i, 0); // 0表示没有父节点（根节点）

        }

    }

    // 输出点双数量和每个点双的节点

    cout << color << endl;

    for(int i=1; i<=color; i++){

        cout << ans[i].size() << ' ';

        for(int j=0; j<ans[i].size(); j++){

            cout << ans[i][j] << ' ';

        }

        cout << endl;

    }

    return 0;

}

最小树形图——朱刘算法

// https://www.luogu.com.cn/problem/P4716

//朱刘算法是一个用于解决有向图的最小生成树算法  时间复杂度是O(VE)

//主要步骤如下：

//1、  找到每个节点的入线的最小权值  记录这个权值以及来源点  也就是这条线

//     如果存在非根节点没有源点  说明无法构成以当前根节点为根的树

//2、将上面所有的线看做有向边  答案加上所有的权值   如果不存在环  说明这就是一棵树  直接结束

//     否则将一个环缩点

//3、  缩完点之后  将原图上的边更新一下  权值全部更新为减去这个v的入度最小权值(因为这个环中多了一条边)

//     然后将u v改成缩完点之后的编号   n变为cnt  root=id[root]

struct node{

    int u,v,w;

}nums[MAXM];

int pre[MAXN];

int in[MAXN];

int id[MAXN];

int vis[MAXN];//这里之所以不是bool类型  主要是为了判断环   如果是bool类型 可能会导致不是环上的点也被计算

void prepare(){

    for(int i=1;i<=n;i++){

        pre[i]=0;

        in[i]=INT\_MAX;

        id[i]=0;

        vis[i]=false;

    }

}

int main()

{

    n=read(),m=read(),root=read();

    for(int i=1;i<=m;i++){

        nums[i].u=read(),nums[i].v=read(),nums[i].w=read();

    }

    long long ans=0;

    while(true){

        prepare();

        for(int i=1;i<=m;i++){

            int u=nums[i].u;

            int v=nums[i].v;

            int w=nums[i].w;

            if(u!=v&&w<in[v]){

                in[v]=w;

                pre[v]=u;

            }

        }

        for(int i=1;i<=n;i++){

            if(in[i]==INT\_MAX){

                cout<<-1<<endl;

                return 0;

            }

        }

        for(int i=1;i<=n;i++){

            if(root==i){

                continue;

            }

            ans+=in[i];

        }

        int cnt=0;

        int idx=0;

        //注意 一个点只能有一条入边  所以如果被访问过了

        //那么如果是环上的点  直接结束    不是环上的点  再次访问也没有什么用

        for(int i=1;i<=n;i++){

            if(vis[i]||i==root){

                //如果已经在环中  或者  是根节点（只能有出边）  那么直接结束

                continue;

            }

            idx++;

            // 这里实际上是想找到一个包含v的最小环（可能不是包含吧  可能是这个环上有一个点出发沿着最短路径可以走到v）

            // 从节点i开始，沿着最小入边的反向路径遍历，检测是否遇到环

            // 遇到了id[v]不等于0的  说明v一定是别的环中的了  一定不可能和自己组成最小环了

            // root不能有入边

            // 只有遇到了vis[v]==idx的  说明来到了环访问过的地方

            int v=i;

            while(vis[v]!=idx&&!id[v]&&v!=root){

                vis[v]=idx;

                v=pre[v];

            }

            //处理环上的节点

            if(v!=root&&!id[v]){

                cnt++;

                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;

            }

        }

        //更新权值

        for(int i=1;i<=m;i++){

            if(nums[i].u!=nums[i].v){

                nums[i].w-=in[nums[i].v];

            }

            nums[i].u=id[nums[i].u];

            nums[i].v=id[nums[i].v];

        }

        //更新变量

        n=cnt;

        root=id[root];

    }

    cout<<ans<<endl;

    return 0;

}