ACM校赛模板

> by gemengmeng

线段树

> 扫描线求面积

```
//不同于普通线段树
//每次更新数值会上传到根节点
//不需要query函数 因为每次都是查询整个线段树的信息
//所以每次只需要查看根节点的信息
#include<bits/stdc++.h>
using namespace std;
const int N = 111;
double x[N];
struct Edge{
    double l,r,h;
    int flag;
}e[N<<1];
bool cmp(Edge& a,Edge& b){
    return a.h<b.h;
}
struct Node{//实际上tl tr无用 每次从0-k-1二分可以得到lr
    int l,r;
    int s;
    double len;//真正需要的
}T[N<<3];//线段树空间不够通常会导致tle 而不是re
void build(int rt,int l,int r){
    T[rt].l = l;
    T[rt].r = r;
    T[rt].len = 0;
    T[rt].s = 0;
    if(l==r) return;
    int m = (l+r)>>1;
    build(rt<<1,l,m);
    build(rt<<1|1,m+1,r);
}
void pushup(int rt){
    //去重
    if(T[rt].s)
        T[rt].len = x[T[rt].r+1] - x[T[rt].l];
```

```
    else if(T[rt].l==T[rt].r)
        T[rt].len = 0;
    else
        T[rt].len = T[rt<<1].len + T[rt<<1|1].len;
}
void update(int id,int l,int r,int xx){
    if(T[id].l>=l&&T[id].r<=r){
        T[id].s += xx;
        pushup(id);//不同于普通的区间更新
        return;
    }
    int mid = (T[id].l+T[id].r)>>1;
    if(l<=mid) update(id<<1,l,r,xx);
    if(r>mid) update(id<<1|1,l,r,xx);
    pushup(id);
}
//绝对不要Edge& a=kk,b=k;b会自动识别成一般变量

int main()
{
    int n;int kas = 0;
    while (scanf("%d",&n) == 1&&n)
    {
        int tot = 0;
        for (int i = 0;i < n;++i)
        {
            double x1,x2,y1,y2;
            scanf("%lf %lf %lf %lf",&x1,&y1,&x2,&y2);//输入一个矩形
            Edge &t1 = e[tot];Edge &t2 = e[1+tot];
            t1.l = t2.l = x1,t1.r = t2.r = x2;
            t1.h = y1;t1.flag = 1;
            t2.h = y2;t2.flag = -1;
            x[tot] = x1;x[tot+1] = x2;
            tot += 2;
        }
        sort(e,e+tot,cmp);//边按高度从小到大排序（自下而上扫描）
        sort(x,x+tot);
        //离散化横坐标
        int k = 1;
        for (int i = 1;i < tot;++i)
        {
            if (x[i] != x[i-1]) //去重
            {
                x[k++] = x[i];
            }
        }
```

```
        build(1,0,k-1);//离散化后的区间是[0, k-1]
        double ans = 0.0;
        for (int i = 0;i < tot;++i)
        {
            //因为线段树维护的是横坐标们的下标，所以对每条边求出其两个横坐标对应的下标
            int l = lower_bound(x,x+k,e[i].l) - x;//在横坐标数组里找到这条边的位置
            int r = lower_bound(x,x+k,e[i].r) - x - 1;//小心
            update(1,l,r,e[i].flag);//每扫到一条边就更新横向的覆盖len
            //相当于全树查询函数
            ans += (e[i+1].h - e[i].h)*T[1].len;//q[1]是整个区间,q[1].k=len是整个区间的有效长度
            //计算面积就是用区间横向的有效长度乘以两条边的高度差（面积是两条边里面的部分）
        }
        printf("Test case #%d\n",++kas);
        printf("Total explored area: %.2f\n\n",ans);
    }
    return 0;
}
```

普通线段树 + lazy

```
//cdoj 1057
//learn from qsc
//segment tree--static structure
//block update and query + lazy
#include<bits/stdc++.h>
using namespace std;
const int maxn = 155+7;
struct node{
    int l,r;
    long long sum,lazy;
    void update(long long v){//long long 小心
        sum+=1ll*(r-l+1)*v;
        lazy+=v;
    }
}tree[maxn*4];
int n,q;
int a[maxn];
void push_up(int x){
    tree[x].sum = tree[x<<1].sum + tree[x<<1|1].sum;
}
void push_down(int x){
    //lazy标记小心
    int lazyval = tree[x].lazy;
    if(lazyval!=0){
```

```
        tree[x<<1].update(lazyval);
        tree[x<<1|1].update(lazyval);
        tree[x].lazy=0;//清空
    }
}

void build(int x,int l,int r){
    tree[x].l = l,tree[x].r = r,tree[x].sum=0,tree[x].lazy=0;
    if(l==r){
        tree[x].sum = a[l-1];
        return;
    }
    int mid = (l+r)>>1;
    build(x<<1,l,mid);
    build(x<<1|1,mid+1,r);
    push_up(x);
}
//对于区间更新查询 传入的参数lr表示目标区间 所以lr不会改变
//原树通过自身lr 以及mid决定范围 这里小心不要同建树mid,mid+1作为参数 这里始终是lr
//不同线段树写法决定 如果tree的lr也作为参数 则需要改变tree的lr参数
void update(int x,int l,int r,long long v){
    int ll = tree[x].l,rr = tree[x].r;
    if(ll>=l&&rr<=r)
        tree[x].update(v);
    else{
        push_down(x);
        int mid = (ll+rr)>>1;
        if(mid>=l)
            update(x<<1,l,r,v);
        if(mid<r)
            update(x<<1|1,l,r,v);
    }
}
long long query(int x,int l,int r){
    int ll = tree[x].l,rr = tree[x].r;
    if(ll>=l&&rr<=r)
        return tree[x].sum;
    else{
        push_down(x);
        long long ans = 0;
        int mid = (ll+rr)>>1;
        if(mid>=l)
            ans+=query(x<<1,l,r);
        if(mid<r)
            ans+=query(x<<1|1,l,r);
        return ans;
```

```
    }
}
int main(){
    scanf("%d",&n);
    for(int i=0;i<n;i++){
        scanf("%d",&a[i]);
    }
    build(1,1,n);
    scanf("%d",&q);
    int l,r,v;
    for(int i=0;i<q;i++){
        scanf("%d%d%d",&l,&r,&v);
        update(1,l,r,v);
        printf("%lld\n",query(1,l,r));
    }
    return 0;
}
```

无旋treap

```
/*
FHQ Treap 无旋treap
引用传递实现 更简单 方便树套树
split  merge关键
split可以按照value rnk进行 这里按照value进行
所以合并按照rnk进行 反之按照value进行
维护:  value意义下的二叉搜索树 随机生成的rnk下的小根堆/大根堆
二叉搜索树索具有的基本性质都具有。并且查询第k大 前驱后继的能力都通过split merge实现
*/
//普通平衡树 模板题
#include<bits/stdc++.h>
using namespace std;
const int INF = 0x7fffffff;
const int maxn = 1e6+7;
//读入挂
inline int read(){
    int res = 0;char c = getchar(); bool neg = 0;
    while(!isdigit(c)){
        neg |=(c=='-');//只要有- 永远是-
        c = getchar();
    }
    while(isdigit(c)){
        res = (res<<3) + (res<<1) + (48^int(c));
        c = getchar();
    }
```

```
    return neg ? -res:res;
}


struct node{
    int size,val,rnk,lc,rc;
}treap[maxn];
int root = 0;
int tot = 0;
int seed = 233;
inline int newrand(){
    return seed = int(seed * 48271ll % INF);
}

inline void update(int root){
    treap[root].size = treap[treap[root].lc].size + treap[treap[root].rc].size
+ 1;
}

inline int add_node(int val){
    treap[++tot].val = val;
    treap[tot].lc = 0;
    treap[tot].rc = 0;
    treap[tot].rnk = newrand();
    treap[tot].size = 1;
    return tot;
}
//按照val划分 划分成<=val >val 规则按照二叉树性质划分
inline void split(int r,int& a,int& b,int val){//r不能是引用
    //单独处理
    if(r==0){//不是处理r==1
        a=b=0;
        return;
    }
    if(treap[r].val<=val){
        a = r;
        split(treap[r].rc,treap[a].rc,b,val);
    }else{
        b = r;
        split(treap[r].lc,a,treap[b].lc,val);
    }
    update(r);
}
//合并对象满足二叉树性质 并且一边大 一边小 为了满足两种性质 按照rnk规则合并合适
inline void merge(int& r,int a,int b){
    //单独处理
```

```
    if(a==0||b==0){
        r = a+b;
        return;
    }
    if(treap[a].rnk<treap[b].rnk){
        r = a;
        merge(treap[r].rc,treap[a].rc,b);
    }else{
        r = b;
        merge(treap[r].lc,a,treap[b].lc);
    }
    update(r);
}

inline void insert_node(int &root,int val){
    int newnode = add_node(val);
    int a = 0,b = 0;
    split(root,a,b,val);
    merge(a,a,newnode);
    merge(root,a,b);
}
inline void delete_node(int &root,int val){
    int a = 0,b = 0,c = 0;
    split(root,a,b,val);
    split(a,a,c,val-1);
    merge(c,treap[c].lc,treap[c].rc);
    merge(a,a,c);
    merge(root,a,b);
}


//树上二分
inline int get_kth(int rt,int k){//获得第k大的元素
    while(treap[treap[rt].lc].size+1!=k){
        if(treap[treap[rt].lc].size>=k){
            rt = treap[rt].lc;
        }else{
            k -= treap[treap[rt].lc].size + 1;
            rt = treap[rt].rc;
        }
    }
    return treap[rt].val;
}


//按照value划分就可以
```

```
inline int get_rnk(int &root,int value){
    int a = 0,b = 0;
    split(root,a,b,value-1);
    int res = treap[a].size + 1;
    merge(root,a,b);
    return res;
}
inline int get_pre(int &root,int value){
    int a = 0,b = 0;
    split(root,a,b,value-1);
    int res = get_kth(a,treap[a].size);
    merge(root,a,b);
    return res;
}
int get_scc(int &root,int value){
    int a = 0,b = 0;
    split(root,a,b,value);
    int res = get_kth(b,1);
    merge(root,a,b);
    return res;
}

int n;
int main(){
    n = read();
    //插入根节点
    add_node(INF);
    root = 1;
    tot = 1;
    //为了不影响统计 即便是即便是INF数字 所以size=0
    treap[root].size = 0;
    int op,val;
    for(int i=0;i<n;i++){
        op=read();
        val=read();
        if (op == 1) {
            insert_node(root, val);
        } else if (op == 2) {
            delete_node(root, val);
        } else if (op == 3) {
            printf("%d\n", get_rnk(root, val));
        } else if (op == 4) {
            printf("%d\n", get_kth(root, val));
        } else if (op == 5) {
            printf("%d\n", get_pre(root, val));
        } else {
```

```
            printf("%d\n", get_scc(root, val));
        }

    }
    return 0;
}
```

主席树

```
//学习网络教程qsc
//kth number 问题hdu 2665 poj2104
//解决 区间查询k小num
#include<iostream>
#include<vector>
#include<algorithm>
#include<cstdio>
using namespace std;
const int maxn = 1e5+7;
int a[maxn],root[maxn];
int n,m,cnt;
int x,y,k;
vector<int> v;
//采用静态结构存储
struct{
    int l,r;
    int size;
}T[maxn*32];//主席树 从1开始
//离散化辅助函数
int mygetid(int value){
    return lower_bound(v.begin(),v.end(),value) - v.begin() + 1;
}
void update(int l,int r,int&x,int y,int pos){//注意引用妙用
    //二分
    T[++cnt] = T[y];
    T[cnt].size++;
    x = cnt;
    if(l==r) return;
    int mid = (l+r)>>1;
    if(mid>=pos)
        update(l,mid,T[x].l,T[y].l,pos);
    else
        update(mid+1,r,T[x].r,T[y].r,pos);

}
```

```
int query(int l,int r,int x,int y,int k){
    //二分
    if(l==r) return l;
    int mid = (l+r)>>1;
    if(k <= T[T[x].l].size - T[T[y].l].size)
        return query(l,mid,T[x].l,T[y].l,k);
    else
        return query(mid+1,r,T[x].r,T[y].r,k-T[T[x].l].size + T[T[y].l].size);


}
int main(){
    int times = 0;
    scanf("%d",&times);
    for(int i=0;i<times;i++){
        scanf("%d%d",&n,&m);
        for(int i=0;i<n;i++){
            scanf("%d",&a[i]);
            v.push_back(a[i]);
        }
        sort(v.begin(),v.end());
        v.erase(unique(v.begin(),v.end()),v.end());//去重加删除 使用getid可以正常操
作了
        //for(int i=0;i<n;i++)
        //    printf("%d ",mygetid(a[i]));
        //插入元素
        for(int i=0;i<n;i++){//坑 注意下标 建立的应该是i+1才对 因为查询是默认1作为起点
            update(1,n,root[i+1],root[i],mygetid(a[i]));
        }
        for(int i=0;i<m;i++){
            scanf("%d%d%d",&x,&y,&k);
            printf("%d\n",v[query(1,n,root[y],root[x-1],k)-1]);
        }
        v.clear();//记得清空v
        cnt=0;//初始化 大坑
    }


    return 0;
}
```

ST表

```cpp
/*
使用st标处理rmq问题 思路动态规划
O（nlogn） 预处理 O（1）查询 空间O（nlogn） 比起线段树查询更快空间更大
思路是倍增算法处理rmq问题 树上倍增可以快速处理动态lca tarjan只能处理静态lca
*/
#include<iostream>
#include<algorithm>
#include<cstdio>
using namespace std;
const int maxn = 1e6+7;
const int maxm = 25;
int st[maxn][maxm];
int help[maxn];
int a[maxn];
int n;
//一下注意+1 -1 处理难点 易错
void init_st_rmq(){
    for(int i=1;i<=n;i++)
        st[i][0] = a[i];
    for(int j=1;(1<<j)<=n;j++)//从小区间到大区间进行更新 dp
        for(int i=1;i+(1<<j)-1<=n;i++)
            st[i][j] = min(st[i][j-1],st[i+(1<<(j-1))][j-1]);
    //预处理help
    for(int i=1;i<=n;i++){
        //直接暴力搜索 可以使用二分搜索 复杂度nlogn
        int k=0;
        while((1<<(k+1))<=i){
            k++;
        }
        help[i] = k;
    }
}
int query_st_rmq(int l,int r){
    int k = help[r-l+1];
    return min(st[l][k],st[r-(1<<k)+1][r]);
}

int main(){
    //。。。
    init_st_rmq();
    //。。。
    return 0;
}
```

树状数组

```
#include<iostream>
using namespace std;
const int maxn = 1e5+7;
/*一维树状数组*/
int a[maxn];
int n;

//得到末尾1所对应的数组
//x作为索引 下标从1开始
int lowbit(int x){
    return x&(-x);
}
//单点修改区间查询
void update(int x,int v){
    for(int i=x;i<=n;i+=lowbit(i)){
        a[i]+=v;
    }
}
int getsum(int x){
    int res = 0;
    for(int i=x;i;i-=lowbit(i)){
        res+=a[i];
    }
    return res;
}

int query(int l,int r){
    if(l<1) return -1;
    if(l==1) return getsum(r);
    return getsum(r)-getsum(l-1);
}
//变种 单点查询 区间修改----此时上面的接口外部不能调用 否则语义矛盾 内部调用上面的接口
//对3-6区间+5 相当于3+5 7-5
int query_one_point(int x){
    return getsum(x);
}
void update_block(int l,int r,int v){
    if(l>=1){
        update(l,v);
        if(r<n)
            update(r+1,-v);
    }
}
```

```cpp
/*二维树状数组*/
// 每一个树状数组节点中单独存储一个树状数组
//单点更新区间查询
int c[maxn][maxn];
void update(int x,int y,int v){
    for(int i=x;i<=n;i+lowbit(i))
        for(int j=y;j<=n;j+=lowbit(j))
            c[i][j]+=v;
}
int getsum(int x,int y){
    if(x==0||y==0)
        return 0;
    int res = 0;
    for(int i=x;i;i-=lowbit(i))
        for(int j=y;j;j-=lowbit(j))
            res+=c[i][j];
    return res;
}
//x1 y1 < x2 y2
int query2dblock(int x1,int y1,int x2,int y2){
    //x1==1 y1==1 会+/- 0不影响
    int res = getsum(x2,y2);
    res -= getsum(x1-1,y2);
    res -= getsum(x2,y1-1);
    res += getsum(x1-1,y1-1);
    return res;
}
//对于区间更新 单点查询
//类似上面的思想 但是具体细节有所不同
int query_one_ponit2d(int x,int y){
    return getsum(x,y);
}
void update_block2d(int x1,int y1,int x2,int y2,int v){
    update(x1,y1,v);
    update(x1,y2+1,-v);
    update(x2+1,y1,-v);
    update(x2+1,y2+1,v);
}

/*
Note:
以上去检修盖单点查询 具体实现需要小心
首先把所有数值单独存储 之后拿到空的(0)矩阵/数组 开始进行区间更新
查询只需要拿到区间上单点查询的结果+原始保存的数据就可以
如果不分开存储 则会单点查询的同时把前缀的数值一同带上 产生错误
*/
```

```
int main(){
    return 0;
}
```

dijsktra

```
/*
dijstkra 算法
不同于广度优先搜索 基于优先队列维护 进行更新
每次使用最短的进行更新
入过到达终点可以提前退出 因为这是必然是最短的
spfa到达也不能退出 因为还可能再次被更新
*/
#include<cstdio>
#include<queue>
#include<vector>
#include<algorithm>
#include<iostream>
#include<functional>

using namespace std;
const int INF = 1e9;
const int maxn = 205;
vector<pair<int,int> > E[maxn];
priority_queue<pair<int,int>,vector<pair<int,int> >,greater<pair<int,int> > >
Q;

int n,m;
int d[maxn];
void init(){
    for(int i=0;i<maxn;i++) E[i].clear();
    for(int i=0;i<maxn;i++) d[i] = INF;
}

int main(){
    while(cin>>n>>m){
        init();
        int x,y,z;
        for(int i=0;i<m;i++){
            scanf("%d%d%d",&x,&y,&z);
            E[x].push_back(make_pair(y,z));
            E[y].push_back(make_pair(x,z));
        }
        int s,t;
        scanf("%d%d",&s,&t);
```

```
        d[s] = 0;
        Q.push(make_pair(d[s],s));
        int now;
        while(!Q.empty()){
            now = Q.top().second;
            if(now==t)//只要走到终点就已经到了最短路
                break;
            Q.pop();
            int v;
            for(int i=0;i<E[now].size();i++){
                v = E[now][i].first;
                if(d[v]>d[now]+E[now][i].second){
                    d[v] = d[now]+E[now][i].second;
                    Q.push(make_pair(d[v],v));
                }

            }
        }
        if(d[t]==INF)
            printf("-1\n");
        else
            printf("%d\n",d[t]);
    }
    return 0;
}
```

spfa

```
/*
快速最短路算法 spfa
贝尔曼福德算法优化 使用优先队列
基于bfs 遍历每一条边 使用每一条边更新目标
*/
#include<cstdio>
#include<queue>
#include<vector>
#include<algorithm>
#include<iostream>
using namespace std;
const int INF = 1e9;
```

```
const int maxn = 205;
vector<pair<int,int> > E[maxn];

int n,m;
int d[maxn],inq[maxn];
void init(){
    for(int i=0;i<maxn;i++) E[i].clear();
    for(int i=0;i<maxn;i++) inq[i] = 0;
    for(int i=0;i<maxn;i++) d[i] = INF;
}

int main(){
    while(cin>>n>>m){
        init();
        int x,y,z;
        for(int i=0;i<m;i++){
            scanf("%d%d%d",&x,&y,&z);
            E[x].push_back(make_pair(y,z));
            E[y].push_back(make_pair(x,z));
        }
        int s,t;
        scanf("%d%d",&s,&t);
        queue<int> Q;
        Q.push(s);
        d[s] = 0; inq[s] = 1;
        int now;
        while(!Q.empty()){
            now = Q.front();
            Q.pop();
            inq[now] = 0;
            int v;
            for(int i=0;i<E[now].size();i++){
                v = E[now][i].first;
                if(d[v]>d[now]+E[now][i].second){
                    d[v] = d[now]+E[now][i].second;
                    if(inq[v]==1)
                        continue;
                    inq[v] = 1;
                    Q.push(v);
                }

            }
        }
        if(d[t]==INF)
            printf("-1\n");
        else
```

```
            printf("%d\n",d[t]);
    }
    return 0;
}
```

floyd

```
// #include<bits/stdc++.h>
#include<algorithm>
#include<cstdio>
#include<iostream>

using namespace std;
const int INF = 1e9;
const int maxn = 205;
int n,m;
int mp[maxn][maxn];

void init(int n){
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
            if(i==j)
                mp[i][j]=0;
            else
                mp[i][j]=INF;
}
//mp[k][i][j] = min(mp[k-1][i][k]+mp[k-1][k][j],mp[k-1][i][j])
//mp[k][i][k] = mp[k-1][i][k]   mp[k][k][j] = mp[k-1][k][j]
//可以压缩状态
void floyd(int n){
    for(int k=0;k<n;k++)
        for(int i=0;i<n;i++)
            for(int j=0;j<n;j++)
                mp[i][j] = min(mp[i][j],mp[i][k]+mp[k][j]);
}
int main(){
    while(scanf("%d%d",&n,&m)!=EOF){
        init(n);
        int x,y,z;
        for(int i=1;i<=m;i++){
            scanf("%d%d%d",&x,&y,&z);
            mp[x][y] = min(z,mp[x][y]);
            mp[y][x] = min(z,mp[y][x]);
        }
        int s,t;
```

```
        scanf("%d%d",&s,&t);
        floyd(n);
        if(mp[s][t]==INF)
            printf("-1\n");
        else
            printf("%d\n",mp[s][t]);

    }
    return 0;
}
```

kruskal

```
/*
使用并查集
只需要对边进行排序+判断
unionset通常从1开始初始化 0可能会导致未知错误
*/
#include<cstdio>
#include<iostream>
#include<cstring>
#include<algorithm>
using namespace std;
typedef long long ll;
const int maxn = 111;
//只需要维护边集合 不需要维护领接表

struct edge{
    int from,to;
    long long cost;
    edge(){}
    edge(int x,int y,ll z):from(x),to(y),cost(z){}
}E[maxn*maxn];

bool cmp(const edge& a,const edge& b){
    return a.cost < b.cost;
}

int father[maxn];
int find(int x){
    if(father[x] == x)
        return x;
    return father[x] = find(father[x]);
}
bool judge(int x,int y){
```

```
    return find(x)==find(y);
}
void unionset(int x,int y){
    int xx=find(x),yy=find(y);
    if(xx==yy)
        return;
    father[xx] = yy;
}
void init(int n){
    for(int i=1;i<=n;i++)
        father[i] = i;
}

int n,m;
ll kruskal(int n){
    ll res = 0;
    sort(E,E+n,cmp);
    for(int i=0;i<n;i++){
        if(judge(E[i].from,E[i].to))
            continue;
        res += E[i].cost;
        unionset(E[i].from,E[i].to);
    }
    return res;
}
int main(){
    while(scanf("%d%d",&n,&m)==2){
        //小心
        if(n==0)
            break;
        init(n);
        ll res = 0;
        for(int i=0;i<n;i++){
            scanf("%d%d%lld",&E[i].from,&E[i].to,&E[i].cost);
        }
        res = kruskal(n);

        for(int i=1;i<=m;i++){
            if(!judge(i,1))
                res = -1;
        }
        if(res==-1)
            printf("?\n");
        else
            printf("%lld\n",res);
    }
```

```
    return 0;
}
```

prim

```
/*
使用优先队列维护
每次选出在集合中连接到外边的最小的边 可以证明正确性

*/
#include<iostream>
#include<cstdio>
#include<algorithm>
#include<functional>
#include<queue>
#include<vector>
#include<cstring>
using namespace std;
typedef long long ll;
const int maxn = 111;

struct edge{
    int to;
    ll cost;
    edge(){}
    edge(int x,ll y):to(x),cost(y){}
    bool operator<(const edge& o) const{//从在比较函数 必须重载城const函数类型
operator<(const)const 可以不是引用
        return o.cost < cost;
    }
};

vector<edge> head[maxn];
priority_queue<edge> Q;
bool vis[maxn];

void init(int m){
    memset(vis,0,sizeof(vis));
    while(!Q.empty()) Q.pop();
    for(int i=1;i<=m;i++)
        head[i].clear();
}

ll prim(){
    ll res = 0;
```

```
    vis[1] = 1;
    for(int i=0;i<head[1].size();i++){
        Q.push(head[1][i]);
    }
    edge e;
    while(!Q.empty()){
        //弹出一个进行处理
        e = Q.top();
        Q.pop();
        if(vis[e.to])
            continue;
        vis[e.to] = 1;
        res += e.cost;
        for(int i=0;i<head[e.to].size();i++)
            Q.push(head[e.to][i]);
    }
    return res;
}
int n,m;
int main(){
    while(scanf("%d%d",&n,&m)==2){
            //小心
        if(n==0)
            break;
        init(m);
        int x,y;
        ll z;
        ll res = 0;
        for(int i=0;i<n;i++){
            scanf("%d%d%lld",&x,&y,&z);//小心 点从1开始 坑点
            head[x].push_back(edge(y,z));
            head[y].push_back(edge(x,z));
        }
        res = prim();
        for(int i=1;i<=m;i++)
            if(!vis[i])
                res = -1;

        if(res==-1)
            printf("?\n");
        else
            printf("%lld\n",res);
    }
    return 0;
}
```

lca

```
//三种方法
//离线: tarjan 在线 倍增   /   st_rmq+dfs序
//tarjan O(n+q)
//倍增 O(nlogn+qlogn)
//st_rmq+dfs序 O(nlogn+q)
//两种在线算法都需要nlogn预处理
//poj
//tarjan
#include<iostream>
#include<cstdio>
#include<algorithm>

using namespace std;
const int N = 1e4+10;
int ind[N];
int fa[N];
bool vis[N];
int head[N],nxt[N],to[N];
int mm;
int a,b;
int n;
bool flag = false;
void add_edge(int a,int b){
    nxt[++mm] = head[a];
    head[a] = mm;
    to[mm] = b;
    ind[b]++;
}
int find(int x){
    if(fa[x]==x)
        return x;
    return fa[x] = find(fa[x]);
}
void join(int a,int b){
    int aa=find(a);
    int bb=find(b);
    if(aa==bb) return;
    fa[aa] = bb;
}
void init(int n){
    for(int i=1;i<=n;i++){
        fa[i] = i;
    }
```

```
    flag = false;
    for(int i=1;i<=n;i++)
        ind[i] = 0,vis[i]=0,head[i]=0;
    for(int i=1;i<=mm;i++)
        nxt[i] = 0;
    mm = 0;

}
//由于只需要一个 所以使用tarjan找到就输出 并且退出 更快

void tarjan(int now){
    if(flag)
        return;
    vis[now]=true;
    for(int i=head[now];i;i=nxt[i]){
        int v = to[i];
        if(vis[v])
            continue;
        tarjan(v);
        if(flag)
            return;
        //每当一个子节点全部完成 则合并到同一类 方便子树之间的查找
        join(now,v);
    }
    //每当当前节点子树处理完毕 则对于当前节点本身进行查询结果的得出过程 保证子树内的关联查询能全部正确
    //本题只需要查找与唯一的查询的关系就可以 不需要遍历所有相关查询 及时exit就可以
    if(now==a&&vis[b]){
        printf("%d\n",find(b));
        flag = true;
        return;
    }
    if(now==b&&vis[a]){
        printf("%d\n",find(a));
        flag = true;
        return;
    }
}

int main(){
    int t;
    scanf("%d",&t);
    while(t--){
        scanf("%d",&n);
        init(n);
        for(int i=1;i<n;i++){
```

```
            scanf("%d%d",&a,&b);
            add_edge(a,b);
            // add_edge(b,a);//是树 并且方向性明确 所以不能添加双向边 并且需要积累ind
        }
        scanf("%d%d",&a,&b);
        for(int i=1;i<=n;i++)
            if(ind[i]==0){
                tarjan(i);
                break;//预防有多个根节点符合
            }
    }

    return 0;
}
//tarjan 失败 未知原因

//dfs + lca
//神奇ac
#include<iostream>
#include<cstdio>
#include<algorithm>
#include<cmath>

using namespace std;
const int N = 1e4+10;
int ind[N];
bool vis[N];
int head[N],nxt[N],to[N];
int mm;
int a,b;
int n;
bool flag = false;
void add_edge(int a,int b){
    nxt[++mm] = head[a];
    head[a] = mm;
    to[mm] = b;
    ind[b]++;
}
//多组输入情况下初始化极其重要
void init(int n){
    flag = false;
    for(int i=1;i<=n;i++)
        ind[i] = 0,vis[i]=0,head[i]=0,depth[i]=0;
    for(int i=1;i<=mm;i++)
        nxt[i] = 0;
    mm = 0;
```

```
}
//核心算法 dfs统计深度 并且记录父子关系 同时得到fa表
int fa[N][25];
int depth[N];
int root;
void dfs(int u,int pre){
    fa[u][0] = pre;
    depth[u] = depth[pre]+1;
    for(int i=head[u];i;i=nxt[i]){
        int v = to[i];
        if(v!=pre)
            dfs(v,u);
    }
}
void init_fa(){//获得倍增的fa表
    dfs(root,0);
    for(int j=0;1<<(j+1)<n;j++){
        for(int i=1;i<=n;i++){
            if(fa[i][j]==0) fa[i][j+1]=0;
            else fa[i][j+1] = fa[fa[i][j]][j];
        }
    }
}
//倍增法实现二分查找
int lca(int u,int v){
    if(depth[u]>depth[v])
        swap(u,v);
    int tmp = depth[v] - depth[u];
    //借助树状数组思想 使用二进制消元 找到相同高度--skill
    for(int i=0;(1<<i)<=tmp;i++){
        if((1<<i)&tmp)
            v = fa[v][i];
    }
    //到达相同高度
    if(u==v)
        return u;
    //二者从某一高度的祖先开始共同瞎讲寻找最近的祖先 由于二进制关系 等效于二分查找
    for(int i=int(log2(n*1.0));i>=0;i--){
        //寻找最近的祖先
        //相当于对于目标高度 使用从高位开始的二进制试探方法
        //利用二进制数可以构成任意的数字的性质进行类似二分的对数复杂度搜索
        if(fa[u][i]!=fa[v][i]){
            u = fa[u][i];
            v = fa[v][i];
        }
```

```
    }
    return fa[u][0];
}

int main(){
    int t;
    scanf("%d",&t);
    while(t--){
        scanf("%d",&n);
        init(n);
        for(int i=1;i<n;i++){
            scanf("%d%d",&a,&b);
            add_edge(a,b);
            // add_edge(b,a);//是树 并且方向性明确 所以不能添加双向边 并且需要积累ind
        }
        scanf("%d%d",&a,&b);
        for(int i=1;i<=n;i++)
            if(ind[i]==0){
                root = i;
                break;
            }
        init_fa();
        int ans = lca(a,b);
        printf("%d\n",ans);
    }

    return 0;
}

//dfs序+rmq
/*
类比普通rmq问题 查找区间内的最小值
lca就是哈找在dfs序的表中 dep最小的点 所以可以通过dfs得到dfs序
通过dfs序构建以器为索引 以dep为值得rmq表 在rmq中快速得到lca
难点:
这里st表示基于tot数量的表格 统计depth 而tot在边 点时候更新 共有2*n-1
所以st表通常使用2*n-1作为大小
*/

#include<iostream>
#include<cstdio>
#include<algorithm>
#include<cmath>
#include<cstring>
using namespace std;
const int N = 1e4+10;
```

```cpp
int ind[N];
int head[N],nxt[N],to[N];
int mm;
int a,b;
int n;
bool flag = false;
int root = 0;

int depth[N<<1];
int dfn[N];//每个节点在dfs中出现的实时间 一个节点可能出现多次 但是这里只记录第一次
//dfn的数值在1-2*n-1之间
int dp[N<<1][25];//开小了直接报错 因为这是基于tot的
int id[N<<1];
int tot = 0;//标志dfn起点
//得到深度 得到dfn序 得到id对应关系 方便查找
void dfs(int u,int pre){
    dfn[u] = ++tot;//只需要保存第一次
    // depth[tot] = depth[pre]+1;
    //经过调试 画图 输出depth数组发现了问题 -- 隐藏的很深
    //depth[pre] 不对 应该使用depth[ dfn[pre] ]
    depth[tot] = depth[dfn[pre]]+1;
    id[tot] = u;
    for(int i=head[u];i;i=nxt[i]){
        int v = to[i];
        if(v!=pre){
            dfs(v,u);
            //每次遍历完成一个子分支 回到根节点一次
            id[++tot] = u;
            depth[tot] = depth[dfn[pre]]+1;
        }
    }
}
//使用之前的depth[pre]+1 报错
//基于tot构建st 2*n-1 由于目标不是求解最小的数值 而是求解最小元素的索引
void init_st(int n){
    int k = int(log2(1.0*n));
    for(int i=1;i<=n;i++) dp[i][0] = i;//小心
    for(int j=1;j<=k;j++)
        for(int i=1;i+(1<<j)-1<=n;i++){
            int a = dp[i][j-1];
            int b = dp[i+(1<<(j-1))][j-1];
            if(depth[a]<depth[b]) dp[i][j] = a;
            else dp[i][j] = b;
        }
}
int rmq(int l,int r){
```

```
    int k = int(log2(1.0*r-l+1));//自动下取整
    int a = dp[l][k];
    int b = dp[r-(1<<k)+1][k];
    if(depth[a]<depth[b]) return a;
    else return b;
}
int lca(int x,int y){
    int l = dfn[x];
    int r = dfn[y];
    if(l>r) swap(l,r);
    //小心 返回的应该是当前dfn序号的反向映射 即当前节点
    return id[rmq(l,r)];
}

void add_edge(int a,int b){
    nxt[++mm] = head[a];
    head[a] = mm;
    to[mm] = b;
    ind[b]++;
}
//多组输入情况下初始化极其重要
void init(int n){
    flag = false;
    for(int i=1;i<=n;i++)
        ind[i] = 0,head[i]=0;
    for(int i=1;i<=mm;i++)
        nxt[i] = 0;
    memset(depth,0,sizeof(depth));
    mm = 0;
    root = 0;
    tot=0;
}

int main(){
    int t;
    scanf("%d",&t);
    while(t--){
        scanf("%d",&n);
        init(n);
        for(int i=1;i<n;i++){
            scanf("%d%d",&a,&b);
            add_edge(a,b);
            // add_edge(b,a);//是树 并且方向性明确 所以不能添加双向边 并且需要积累ind
        }
        scanf("%d%d",&a,&b);
        for(int i=1;i<=n;i++)
```

```
            if(ind[i]==0){
                root = i;
                break;
            }
        // dfs(root,0,1);
        dfs(root,0);
        init_st(2*n-1);
        int ans = lca(a,b);
        printf("%d\n",ans);
    }

    return 0;
}
```

匈牙利算法

```
/*
最大匹配数：最大匹配的匹配边的数目
最小点覆盖数：选取最少的点，使任意一条边至少有一个端点被选择
最大独立数：选取最多的点，使任意所选两点均不相连
最小路径覆盖数：对于一个 DAG（有向无环图），选取最少条路径，使得每个顶点属于且仅属于一条路径。
路径长可以为 0（即单个点）。

定理1：最大匹配数 = 最小点覆盖数（这是 Konig 定理）
定理2：最大匹配数 = 最大独立数
定理3：最小路径覆盖数 = 顶点数 - 最大匹配数

匈牙利树一般由 BFS 构造（类似于 BFS 树）
但是匈牙利树要求所有叶子节点均为匹配点
*/
#include<cstdio>
#include<iostream>
#include<algorithm>
#include<cstring>
using namespace std;
const int MAXN = 1010;
const int MAXM = 5050;
int head[MAXN];
int nxt[MAXM];
int to[MAXM];
bool vis[MAXN];//避免当次dfs重复搜索
bool linker[MAXN];
int un,vn;
```

```
//需要把左边的点排成一列
bool dfs(int u){
    for(int i=head[u];i;i=nxt[i]){
        int v = to[i];
        if(!vis[v]){
            vis[v] = true;
            //dfs调用的其实都是u这边的节点
            if(!linker[v]||dfs(linker[v])){//表明对面集合的节点是否被连接过
                linker[v] = u;
                linker[u] = v;//便于剪枝
                return true;
            }
        }
    }
    return false;
}
int hungary(){
    int res = 0;
    memset(linker,0,sizeof(linker));
    for(int u=1;u<=un;u++){
        //遍历所有u集合点
        if(!linker[u]){
            memset(vis,false,sizeof(vis));//由于已有前面节点新建的链接 所以之前的vis需要充重置
            if(dfs(u)) res++;
        }

    }
    return res;
}
int main(){
    return 0;
}
```

dinic

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <queue>
#define inf 0x3f3f3f3f3f3f3f3f
#define ll long long
#define MAXN 10005
using namespace std;
int n,m;//点数、边数
```

```
int sp,tp;//原点、汇点
struct node
{
    int v,next;
    ll cap;
}mp[MAXN*10];
int pre[MAXN],dis[MAXN],cur[MAXN];//cur为当前弧优化，dis存储分层图中每个点的层数（即到原点的最短距离），pre建邻接表
int cnt=0;
void init()//不要忘记初始化
{
    cnt=0;
    memset(pre,-1,sizeof(pre));
}
void add(int u,int v,int w)//加边
{
    mp[cnt].v=v;
    mp[cnt].cap=w;
    mp[cnt].next=pre[u];
    pre[u]=cnt++;
}
bool bfs()//建分层图
{
    memset(dis,-1,sizeof(dis));
    queue<int>q;
    while(!q.empty())
    q.pop();
    q.push(sp);
    dis[sp]=0;
    int u,v;
    while(!q.empty())
    {
        u=q.front();
        q.pop();
        for(int i=pre[u];i!=-1;i=mp[i].next)
        {
            v=mp[i].v;
            if(dis[v]==-1&&mp[i].cap>0)
            {
                dis[v]=dis[u]+1;
                q.push(v);
                if(v==tp)
                break;
            }
        }
    }
```

```
    return dis[tp]!=-1;
}
ll dfs(int u,ll cap)//寻找增广路
{
    if(u==tp||cap==0)
    return cap;
    ll res=0,f;
    for(int &i=cur[u];i!=-1;i=mp[i].next)
    {
        int v=mp[i].v;
        if(dis[v]==dis[u]+1&&(f=dfs(v,min(cap-res,mp[i].cap)))>0)
        {
            mp[i].cap-=f;
            mp[i^1].cap+=f;
            res+=f;
            if(res==cap)
            return cap;
        }
    }
    if(!res)
    dis[u]=-1;
    return res;
}
ll dinic()
{
    ll ans=0;
    while(bfs())
    {
        for(int i=1;i<=n;i++)
        cur[i]=pre[i];
        ans+=dfs(sp,inf);
    }
    return ans;
}
int main()
{

return 0;
}
```

mcmf

```
//最小费用最大流mcmf
//by 趣学算法
//最小费用最大流---满足最大流情况下的最小费用流
```

```
//每次使用spfa找到最小费用路径 在最小费用路径上进行增广一次最短路
//类似于sap isap dinic 中的思想 都是具有导向性的寻找增光路
//dinic基于bfs得到的距离原点的层次进行 isap基于距离汇点的距离进行 ek(sap)导向性不足所以有了isap的导向性优化
//顺便复习spfa用法 以及判断负权环的方法 spfa中需要注意--路径需要满足可憎光的路径条件下的最短路
#include<iostream>
#include<cstring>
#include<cstdio>
#include<algorithm>
#include<queue>
using namespace std;
const int N = 5e4+10;
const int M = 1e6+10;
const int INF = 1e9+7;
int head[N];
struct edge{
    int to,nxt;
    int cap,flow,cost;
}e[M];
int mm;//表示变边数目 01 23 45 为一对
// int que[M];//可能大小不够 感觉需要n*n保险
// int head=1,tail=1;
int cnt[N];//入队次数
int pre[N];//当前点前驱边
bool vis[N];
int dis[N];//到达各个节点的最小距离
int maxflow,mincost;
void _add_edge(int a,int b,int c,int d){//添加的时候只需要顺次添加正向边 反向边  从而形成混合图 -- 正流图 残留图
    e[mm].nxt = head[a];
    head[a] = mm;
    e[mm].to = b;
    e[mm].cap = c;
    e[mm].flow = 0;
    e[mm].cost = d;
    mm++;
}
void add_edges(int a,int b,int c,int d){
    _add_edge(a,b,c,d);
    _add_edge(b,a,0,-d);//小心 cost是负的 cap是0
}
void init(){
    memset(head,-1,sizeof(head));//保证最后的nxt为-1
    mm = 0;
    maxflow = 0;
```

```
    mincost = 0;
}
void spfa_init(int n){
    memset(cnt,0,sizeof(cnt));
    memset(vis,false,sizeof(vis));
    memset(pre,-1,sizeof(pre));//由于从0开始 所以需要初始化成-1
    for(int i=1;i<=n;i++){
        dis[i] = INF;
    }
}
bool spfa(int s,int t,int n){//不怕负边 怕负环
    queue<int> que;
    //首先初始化
    spfa_init(n);
    vis[s] = true;
    cnt[s]++;
    que.push(s);
    dis[s] = 0;
    int top;
    while(!que.empty()){
        top = que.front();
        que.pop();
        //不能缺少
        vis[top] = false;
        for(int i=head[top];~i;i=e[i].nxt){//小心初始化 必须要把head初始化成-1才行
            int v = e[i].to;
            if(dis[v]>dis[top]+e[i].cost && e[i].flow<e[i].cap){
                dis[v] = dis[top]+e[i].cost;
                pre[v] = i;//无论是否松弛 都需要处理
                if(!vis[v]){
                    vis[v] = true;
                    cnt[v]++;
                    que.push(v);
                    if(cnt[v]>n){
                        //printf("存在负环\n");
                        return false;
                    }
                }
            }
        }
    }
    if(dis[t]==INF)
        return false;
    return true;
}
void MCMF(int s,int t,int n){
```

```
    mincost = 0;
    maxflow = 0;
    while(spfa(s,t,n)){
        int minf = INF;
        //反向查找
        for(int i=pre[t];~i;i=pre[e[i^1].to])//拿到反向边的v从而拿到前节点 技巧强 双边存储的优势
            minf = min(minf,e[i].cap - e[i].flow);
        maxflow += minf;
        for(int i=pre[t];~i;i=pre[e[i^1].to]){
            e[i].flow+=minf;
            e[i^1].flow-=minf;
            // e[i^1].cap-=minf;容量一直是0就可以 只要关注流量就行 因为二者cap=0 flow<0才是正常
        }
        mincost+=dis[t]*minf;
    }
}
int main(){
    int n,m,s,t;
    init();//必须有
    scanf("%d%d%d%d",&n,&m,&s,&t);
    int aa,bb,cc,dd;
    for(int i=1;i<=m;i++){
        scanf("%d%d%d%d",&aa,&bb,&cc,&dd);
        add_edges(aa,bb,cc,dd);
    }
    MCMF(s,t,n);
    printf("%d %d\n",maxflow,mincost);
}
```

博弈论

```
// 什么是巴什博弈：只有一堆n个物品，两个人轮流从这堆物品中取物， 规定每次至
// 少取一个，最多取m个。最后取光者得胜。
/*
巴什博奕
只有一堆n个物品，两个人轮流从中取物，规定每次最少取一个，最多取m个，最后取光者为胜
若 n%(m+1)==0 后手必胜
否则先手必胜，若n>m，第一次取n%(m+1)个，若n<=m，则第一次可取n~m间任意个
*/
/*
威佐夫博弈
有两堆各若干的物品，两人轮流从其中一堆取至少一件物品，至多不限，或从两堆中同时取相同件物品，规定最后取完者胜利。
```

```
*/
#include<bits/stdc++.h>
using namespace std;
bool weizuofu(int n1,int n2){
    if(n1>n2)
        swap(n1,n2);
    int temp = floor((n2-n1)*(1+sqrt(5.0))/2.0);
    if(temp == n1) return true;//后手
    else return false;
}
// 尼姆博弈:
// 有任意堆物品，每堆物品的个数是任意的，双方轮流从中取物品，每一次只能从一堆物品中取部分或全部物品，最少取一件，取到最后一件物品的人获胜。
// 结论：把每堆物品数全部异或起来，如果得到的值为0，那么先手必败，否则先手必胜。
void nim(int n){
      int temp=0,ans;
      for(int i=0;i<n;i++){
          cin>>ans;
          temp^=ans;
      }
      if(temp==0)  cout<<"后手必胜"<<endl;
      else cout<<"先手必胜"<<endl;
}
/*
斐波那契博弈:
有一堆物品，两人轮流取物品，先手最少取一个，至多无上限，但不能把物品取完，之后每次取的物品数不能超过上一次取的物品数的二倍且至少为一件，取走最后一件物品的人获胜。
结论：  先手胜当且仅当n不是斐波那契数（n为物品总数）
*/
const int N = 55;
int f[N];
void Init()
{
    f[0] = f[1] = 1;
    for(int i=2;i<N;i++)
        f[i] = f[i-1] + f[i-2];
}
int main()
{
    Init();
    int n;
    while(cin>>n)
    {
        if(n == 0) break;
        bool flag = 0;
        for(int i=0;i<N;i++)
```

```
        {
            if(f[i] == n)
            {
                flag = 1;
                break;
            }
        }
        if(flag) puts("Second win");
        else     puts("First win");
    }
    return 0;
}
/*
公平组合博弈:
（1）两人参与。
（2）游戏局面的状态集合是有限。
（3）对于同一个局面，两个游戏者的可操作集合完全相同
（4）游戏者轮流进行游戏。
（5）当无法进行操作时游戏结束，此时不能进行操作的一方算输。
（6）无论游戏如何进行，总可以在有限步数之内结束。

SG函数:
g(x)=mex{g(y)|y是x的后继}
mex：对集合运算，表示不属于该集合的最小非负整数，如mex{0,1,2,3}=4,mex{2,3,5}=0
SG定理：g(G)=g(G1)⊕g(G2)⊕g(G3)…… 一个游戏的SG值为其子游戏SG值的异或和
结论：g(G)=0时，先手必输，反之，先手必胜

解题模型:
1.把原游戏分解成多个独立的子游戏，则原游戏的SG函数值是它的所有子游戏的SG函数值的异或。
即sg(G)=sg(G1)^sg(G2)^…^sg(Gn)。
2.分别考虑没一个子游戏，计算其SG值。
SG值的计算方法：（重点）
1.可选步数为1~m的连续整数，直接取模即可，SG(x) = x % (m+1);
2.可选步数为任意步，SG(x) = x;
3.可选步数为一系列不连续的数，用模板计算。
*/
//打表
const int N=10005;
int f[N],SG[N],S[N];                    //N要足够大，包含所有情况
void  getSG(int n,int m)          //n表示所求SG长度，m表示f数组长度
{
    int i,j;
    sort(f,f+m);
    memset(SG,0,sizeof(SG));
    //因为SG[0]始终等于0，所以i从1开始
    for(i = 1; i <= n; i++)
```

```
    {
        //每一次都要将上一状态 的 后继集合 重置
        memset(S,0,sizeof(S));
        for(j = 0; f[j] <= i && j<m; j++)
            S[SG[i-f[j]]] = 1;  //将后继状态的SG函数值进行标记
        for(j = 0;; j++) if(!S[j])
        {   //查询当前后继状态SG值中最小的非零值
            SG[i] = j;
            break;
        }
    }
}
//搜索
const int N=10000;
vector<int>e[N];          //邻接表

int sg[N];                          //sg全部初始化为-1
int getsg(int x)
{
    if(sg[x]!=-1)
        return sg[x];
    int mex[N];
    memset(mex,0,sizeof(mex));
    for(int i=0;i<e[x].size();i++)
        mex[getsg(e[x][i])]=1;
    for(int i=0;;i++)
    {
        if(!mex[i])
            return sg[x]=i;             //记忆化搜索
    }
}
```

莫队算法

> 普通

```
//莫队算法模板题
//小z的袜子 组合数技巧
#include<iostream>
#include<algorithm>
#include<cmath>
#include<cstdio>
using namespace std;
const int MAXN = 5e5+10;
int B = 1;
```

```
int color[MAXN];
int cnt[MAXN],ans[MAXN][2];//ans保存结果 cnt保存区间内某个颜色对应的出现次数
int len,son,mom;//中间变量 计算使用

inline int gcd(int a,int b){return b==0?a:gcd(b,a%b);}
//重载函数必须写成const const 最后一个const必不可少
struct Query{
    int l,r,id;
    bool operator < (const Query& other) const {
        return (l/B==other.l/B)?r<(other.r):(l<other.l);
    }

}q[MAXN];
void del(int x){//传入的应该是颜色
    cnt[x]--;
    son-=cnt[x];
    len--;
    mom-=len;
}
void add(int x){
    son+=cnt[x];
    cnt[x]++;
    mom+=len;
    len++;
}

int main(){
    int n,m;
    scanf("%d%d",&n,&m);
    B = int(n/sqrt(m));
    for(int i=1;i<=n;i++){
        scanf("%d",&color[i]);
    }
    for(int i=1;i<=m;i++){
        scanf("%d%d",&q[i].l,&q[i].r);
        q[i].id = i;
    }
    int l=1,r=0;
    sort(q+1,q+m+1);
    for(int i=1;i<=m;i++){
        //剪枝判断
        if(q[i].l==q[i].r){
            ans[q[i].id][0] = 0;
            ans[q[i].id][1] = 1;
            continue;
        }
```

```
        while(l<q[i].l){
            del(color[l++]);
        }
        while(l>q[i].l){
            add(color[--l]);
        }
        while(r<q[i].r){
            add(color[++r]);
        }
        while(r>q[i].r){
            del(color[r--]);
        }
        //更新结束记录result
        int g = gcd(son,mom);
        ans[q[i].id][0] = son/g;
        ans[q[i].id][1] = mom/g;
    }
    for(int i=1;i<=m;i++){
        printf("%d\/%d\n",ans[i][0],ans[i][1]);
    }

    return 0;
}
```

> 带修改的莫队+离散化

```
//cf940f Machine Learning
//类似数颜色 统计在区间内各元素出现的次数的数组中没出现的最小正数字
//由于数据比较大范围 使用离散化 重新映射数值 并且不需要反向映射
//典型的待修改莫队算法 复杂度达到N*N^(5/3) 对于1e5可以接受
/*
总结:  由于离散化不熟悉 导致出现很多错误
分块操作 对于快的大小的更新 init过程 sort(q+1,q+n+1)过程 碎玉c的离散化过程 res索引处理都容易出错
由于离散化需要 并且n+c的数目才是真正的数目 所以需要更大的数组空间 2*N
*/
#include<iostream>
#include<cstdio>
#include<cmath>
#include<algorithm>
#include<cstring>
using namespace std;
const int N = 1e5+1000;
int n,m;
```

```
int cnt[N<<1],sum[N<<1],res[N<<1];//分别用于存放 数字出现的次数 出现对应次数的总和 最终的结果
int B = 1;
int a[N<<1],temp[N<<1];//小心 temp cnt sum需要更大才能装得下
int queryn,changen,tempn;
struct Query{
    int id,l,r,t;
    //注意排序判断
    bool operator <(const Query& o) const{
        return l/B==o.l/B?(r/B==o.r/B ? t<o.t:r<o.r):l<o.l;
    }
}q[N];
struct Change{
    int p,val;
}c[N];
void add(int x){//传入数值
    sum[cnt[x]]--;
    cnt[x]++;
    sum[cnt[x]]++;
}
void sub(int x){
    sum[cnt[x]]--;
    cnt[x]--;
    sum[cnt[x]]++;
}
void chnum(int x,int tim){
    //修改某一区间所对应的数值 传入的是查询区间的编号
    if(q[x].l<=c[tim].p&&q[x].r>=c[tim].p){
        sub(a[c[tim].p]);
        add(c[tim].val);
    }
    //交换就可以了 下一次由于单调性必然回退 交换可以保证正确回退
    swap(a[c[tim].p],c[tim].val);
}
void init(){
    memset(cnt,0,sizeof(cnt));
    memset(sum,0,sizeof(sum));
    memset(res,0,sizeof(res));
    queryn = changen = tempn= 0;
}
int main(){
    //第一次wa了 因为:  离散化需要考虑到c的数值 c未必会在a中出现过 所以离散化需要全部统计
    while(scanf("%d%d",&n,&m)!=EOF){
        init();
        B = pow(n,0.66666);
```

```
        for(int i=1;i<=n;i++)
            scanf("%d",&a[i]),temp[++tempn] = a[i];
        int opt,aa,bb;
        for(int i=1;i<=m;i++){
            scanf("%d%d%d",&opt,&aa,&bb);
            if(opt==1){
                q[++queryn].l = aa;
                q[queryn].r = bb;
                q[queryn].id = queryn;//排序之后还能找得到原始的位置
                q[queryn].t = changen;
            }else if(opt==2){
                c[++changen].p = aa;
                c[changen].val = bb;
                temp[++tempn] = bb;
            }
        }
        sort(q+1,q+queryn+1);

        //数据离散化 由于不需要原始数据 所以只需要借助一个额外的排序拷贝数组就可以 a存放新的
id
        //memcpy(temp,a,sizeof(a));
        sort(temp+1,temp+tempn+1);
        ////unique会返回单一化元素后的最后元素的指针 真正想要删除需要配合erase使用 删除后
面的剩余元素
        tempn = unique(temp+1,temp+tempn+1)-temp;//不需要再-1/+1
        for(int i=1;i<=n;i++)
            a[i] = lower_bound(temp+1,temp+tempn+1,a[i])-temp;
        //c特别小心 因为是用数值部分都需要相应离散化处理
        for(int i=1;i<=changen;i++)
            c[i].val = lower_bound(temp+1,temp+tempn+1,c[i].val)-temp;//减去起始
下表不可以少 因为返回的是int*
        //必不可少 关键
        sort(q+1,q+queryn+1);
        int l=1,r=0,tt=0;
        //正式处理
        for(int i=1;i<=queryn;i++){
            while(l<q[i].l){
                sub(a[l++]);
            }
            while(l>q[i].l){
                add(a[--l]);
            }
            while(r>q[i].r){
                sub(a[r--]);
            }
            while(r<q[i].r){
```

```
                add(a[++r]);
            }
            //应对万区间变化  再去应对时间变化
            while(tt<q[i].t){
                chnum(i,++tt);
            }
            while(tt>q[i].t){
                chnum(i,tt--);
            }
            //记录每一次询问的结果 只需要查找sum数组
            int ans = 1;
            while(sum[ans]!=0){
                ans++;
            }
            res[q[i].id] = ans;
        }
        for(int i=1;i<=queryn;i++){
            printf("%d\n",res[i]);//这里只需要res[i] 因为已经是处理过的正确的顺序
        }

    }
    return 0;
}
```

树上分块

```
//scoi2005 王室联邦
//通过dfs 树上分块 保证每一块的大小在B-3B 之间
#include<iostream>
#include<cstdio>
#include<algorithm>
using namespace std;
const int N = 1010;
//双向边存储 空间不足2n会w掉3个点
int stk[N<<1];
int head[N],nxt[N<<1],to[N<<1];
int tot,top;//tot记录当前划分的省份数量 top记录当前栈顶
int belong[N];
int center[N];
int B=1;
int mm;
//在dfs中进行分块
//遍历完成一颗子树之后，只要栈中元素大于B就出栈 作为一个整体    当前节点dfs结束加入stack   最后dfs结束剩余的元素加入最后一个整体 总体保证[B-3B)
void dfs(int u,int fa){//记录父节点 方便找到center
```

```
    int t = top;
    for(int i=head[u];i;i=nxt[i]){
        int v = to[i];//得到当前指向
        if(v==fa)
            continue;
        dfs(v,u);
        //遍历完成一个子树 就要判断当前栈 才能保证符合要求 复杂度证明基于子树大小分析
        if(top-t>=B){
            tot++;
            while(top!=t){
                belong[stk[top--]]=tot;
            }
            center[tot] = u;
        }
    }
    //遍历过程结束 对于已遍历节点加入堆栈
    stk[++top] = u;//栈从1开始存储
}
//收尾工作
void solve(){
    dfs(1,0);
    while(top>=1){
        belong[stk[top--]] = tot;
    }
}
//加边 使用存图的领接表形式存储 也可以使用左儿子右兄弟方式存储
//小心题目要求是双向边
void add_edge(int a,int b){
    nxt[++mm] = head[a];//从1开始存储
    head[a] = mm;
    to[mm] = b;
}
int main(){
    int n;
    int a,b;
    scanf("%d%d",&n,&B);
    for(int i=1;i<n;i++){
        scanf("%d%d",&a,&b);
        add_edge(a,b);
        add_edge(b,a);
    }

    solve();
    printf("%d\n",tot);
    //、我傻啦
    for(int i=1;i<n;i++){
```

```
        printf("%d ",belong[i]);
    }
    printf("%d\n",belong[n]);

    for(int i=1;i<tot;i++){
        printf("%d ",center[i]);
    }
    printf("%d\n",center[tot]);
    return 0;
}
```

马拉车

```
//learn by qsc
//manacher -- hdu 3068
//注意点 max id什么时候更新 收尾字符必须填充 最终需要对于p[p]-1才是正确的
//暴力过程可以使用一点点技巧
//#include<bits/stdc++.h>
#include<iostream>
#include<cstring>
#include<cstdio>
#include<algorithm>
using namespace std;
const int maxn = 3e5;
char s[maxn],str[maxn];
int p[maxn];
int len1,len2;
int ans;
void init(){
    str[0] = '*';
    str[1] = '#';
    str[2*len1+2] = '$';//千万不要与*一样 否则停不下啦 一直匹配 位置空间可以访问
    for(int i=0;i<len1;i++){
        str[2*i+2] = s[i];
        str[2*i+3] = '#';
    }
    len2 = len1*2+2;
}
void manacher(){
    int id = 0;
    int imax = 0;//不要忘记初始化
    for(int i=1;i<=2*len1+1;i++){
        if(i<imax){
            p[i] = min(p[2*id-i],imax-i+1);
        }else{
```

```
            p[i] = 1;
        }
        //循环外面 **
        for(;str[i-p[i]]==str[i+p[i]];p[i]++);
        //设置更新条件
        if(p[i]+ i - 1 > imax){
            id = i;
            imax = i + p[i] - 1;
        }

    }
}
int main(){
    while(scanf("%s",s)!=EOF){
        len1 = strlen(s);
        init();
        manacher();
        ans = 0;
        for(int i=1;i<=len1*2+1;i++)
            ans = max(ans,p[i]);
        printf("%d\n",ans-1);
    }
    return 0;
}
```

kd-tree

```
//不同于dp 支持动态查询最近元素
#include<cstdio>
#include<cmath>
#include<algorithm>
using namespace std;
#define N 500001
#define INF 2147483647
#define KD 2//Î¬¶ÈÊý
int qp[KD],disn;
int n,root;
bool dn;
struct Node
{
    int minn[KD],maxx[KD],p[KD];
    int ch[2];
    void Init()
      {
        for(int i=0;i<KD;++i)
```

```
            minn[i]=maxx[i]=p[i];
        }
    int Dis()
        {
            int res=0;
            for(int i=0;i<KD;++i)
            {
                res+=max(0,minn[i]-qp[i]);
                res+=max(0,qp[i]-maxx[i]);
            }
            return res;
        }
}T[N<<1];
void Update(int rt)
{
    for(int i=0;i<2;++i)
        if(T[rt].ch[i])
            for(int j=0;j<KD;++j)
            {
                T[rt].minn[j]=min(T[rt].minn[j],T[T[rt].ch[i]].minn[j]);
                T[rt].maxx[j]=max(T[rt].maxx[j],T[T[rt].ch[i]].maxx[j]);
            }
}
int Abs(const int &x)
{
    return x<0 ? (-x) : x;
}
int Dis(int a[],int b[])
{
    return Abs(a[0]-b[0])+Abs(a[1]-b[1]);
}
bool operator < (const Node &a,const Node &b)
{
    return a.p[dn]!=b.p[dn] ? a.p[dn]<b.p[dn] : a.p[dn^1]<b.p[dn^1];
}
int Buildtree(int l=1,int r=n,bool d=0)
{
    dn=d;
    int m=(l+r>>1);
    nth_element(T+l,T+m,T+r+1);
    T[m].Init();
    if(l!=m) T[m].ch[0]=Buildtree(l,m-1,d^1);
    if(m!=r) T[m].ch[1]=Buildtree(m+1,r,d^1);
    Update(m);
    return m;
}
```

```
void Query(int rt=root)
{
    disn=min(disn,Dis(T[rt].p,qp));
    int dd[2];
    for(int i=0;i<2;i++)
      if(T[rt].ch[i])
        dd[i]=T[T[rt].ch[i]].Dis();
      else dd[i]=INF;
    bool f=(dd[0]<=dd[1]);
    if(dd[!f]<disn) Query(T[rt].ch[!f]);
    if(dd[f]<disn) Query(T[rt].ch[f]);
}
void Insert(int rt=root,bool d=0)
{
    bool f=(T[n].p[d]>T[rt].p[d]);
    if(T[rt].ch[f])
      Insert(T[rt].ch[f],d^1);
    else
      T[rt].ch[f]=n;
    Update(rt);
}
int q;
int main()
{
//  freopen("bzoj2716.in","r",stdin);
//  freopen("bzoj2716.out","w",stdout);
    int op,x,y;
    scanf("%d%d",&n,&q);
    for(int i=1;i<=n;++i)
      scanf("%d%d",&T[i].p[0],&T[i].p[1]);
    root=(1+n>>1);
    Buildtree();
    for(int i=1;i<=q;++i)
      {
        scanf("%d",&op);
        if(op==1)
          {
            ++n;
            scanf("%d%d",&T[n].p[0],&T[n].p[1]);
            T[n].Init();
            Insert();
          }
        else
          {
            scanf("%d%d",&qp[0],&qp[1]);
            disn=INF;
```

```
            Query();
            printf("%d\n",disn);
          }
      }
    return 0;
}
```