# 数据结构

# *并查集*

## 并查集的优化方式：

```
[1]路径压缩  O(logn)
return fa[x]=find(fa[x])

[2]按秩合并  O(logn)
将偏树合并到主树上

当两个优化都加上时，并查集的时间复杂度为 O(n*Alpha(n))

其中Alpha()为Ackerman函数的某个反函数，对于 N<2^2^10^19727都有 Alpha(N)<5
```

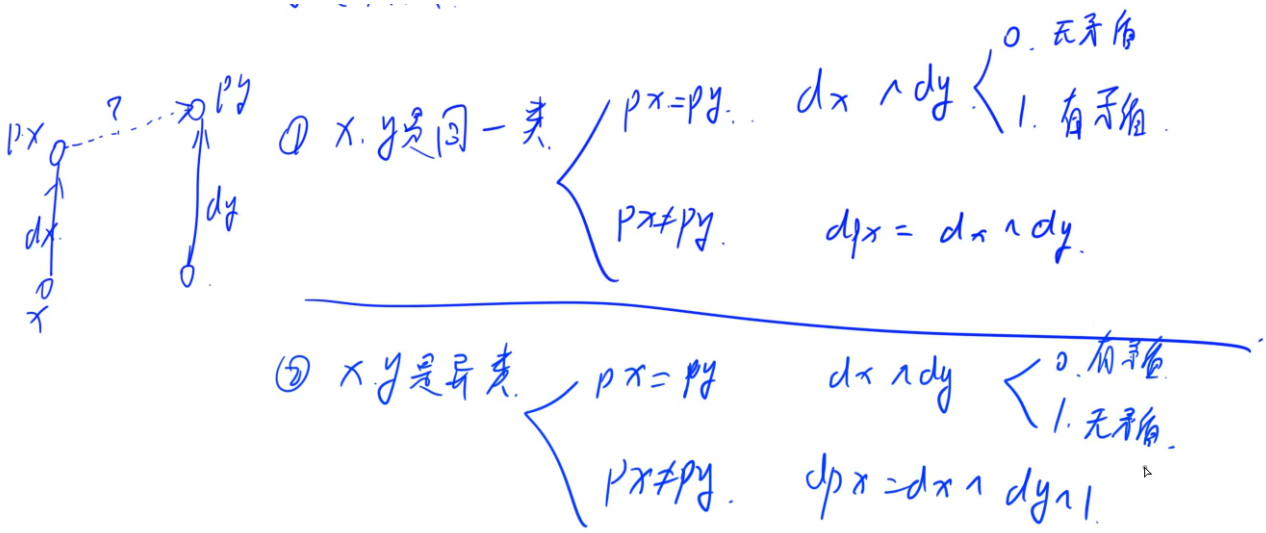
AND MORE

```
(1)记录每个集合大小，并绑定在根节点上
(2)记录每个点到根节点的距离
(3)维护多类关系，维护到根节点的距离mod某个数来分类：
    [1]带权并查集
    [2]扩展域并查集
```

对于如何维护到根节点的距离：

```
int fa[N],n;
int num[N]; //序号位置
int length[N];//所在序列长度

int find(int x){
    if(fa[x]!=x){
        int u=find(fa[x]);  //先做一遍遍历，把前面点距根节点的距离都维护好
        num[x]+=num[fa[x]]; //更新此点到根节点的距离
        fa[x]=u;  //路径压缩
    }
    return fa[x];
}
```

带权并查集的权值维护：




## 种类并查集

用于维护多个种类的并查集,通过对集合总数的模数进行维护。有两个关键：

1.find函数

```
int find(int x){
    if(x!=fa[x]){
        int t = find(fa[x]);
        d[x] += d[fa[x]];
        fa[x] = t;
    }
    return fa[x];
}
```

2.距离修正

```
距离修正 = 对应模数
```

## *树状数组*

```
int tr[200010];
int lowbit(int x){
    return x&-x;
}

void add(int x,int d){
    for(int i=x;i<=N-10;i+=lowbit(i))
        tr[i]+=d;
}

int ask(int x){
    int ans=0;
    for(int i=x;i;i-=lowbit(i))
        ans+=tr[i];
```

```
    return ans;
}

struct BIT{
    #define lowbit(x) ((x)&(-x))
    int tr[N],n;
    void resize(int _n){n=_n;}
    inline void add(int x,int d){for(;x<=n;x+=lowbit(x)) tr[x]+=d;}
    inline int ask(int x){int ans=0;for(;x;x-=lowbit(x)) ans+=tr[x];return ans;}
}T;

struct BIT{
    #define lowbit(x) ((x)&(-x))
    int tr[N];
    inline void add(int x,int d){for(;x<=N-10;x+=lowbit(x)) tr[x]+=d;}
    inline int ask(int x){int ans=0;for(;x;x-=lowbit(x)) ans+=tr[x];return ans;}
    int query(LL s){ //BIT上二分
        int pos=0;
        LL t=0;
        for(int j=18;j>=0;--j){
            if((pos|(1<<j))<=n && t + tr[pos|(1<<j)] <= s){
                pos |= 1<<j;
                t += tr[pos];
            }
        }
        return pos;
    }
}T;
```

是在单调不增的数组中找到最后一个>val的位置.如果要找第一个，则需要二分+树状数组，即$nlog^2n$

# *线段树*

## 带懒标记的线段树

```
#include<iostream>
#include<algorithm>
#include<cstring>
#include<string.h>
#include<cstdio>
#include<vector>
#include<queue>
#include<stack>
#define M 100010
using namespace std;
typedef long long LL;
//read(x)
template <typename T>void read(T &x){x=0;int f=1;char ch=getchar();while(!isdigit(ch)){if(ch=='-
')f=-1;ch=getchar();}while(isdigit(ch)){x=x*10+(ch^48);ch=getchar();}x*=f;return;}
//write(x)
template <typename T>void write(T x){if(x<0){putchar('-');x=-
x;}if(x>9)write(x/10);putchar(x%10+'0');return;}

struct Node{
```

```
    int l,r;
    LL add,mul;
    LL sum;
}tr[M*4];
int w[M],n,m,p;

void val(Node &root,LL add,LL mul){
    root.sum=((LL)(root.sum*mul)%p+(LL)(root.r-root.l+1)*add%p)%p;
    root.add=((LL)root.add*mul%p+(LL)add)%p;
    root.mul=(LL)(root.mul*mul)%p;
}

void push_up(int u){
    tr[u].sum=(LL)(tr[u<<1].sum+tr[u<<1|1].sum)%p;
    tr[u].add=0,tr[u].mul=1;
}

void push_down(int u){
    val(tr[u<<1],tr[u].add,tr[u].mul);
    val(tr[u<<1|1],tr[u].add,tr[u].mul);
    tr[u].add=0,tr[u].mul=1;
}

void build(int u,int l,int r){
    tr[u]={l,r};
    if(l==r){
        tr[u].sum=w[r];
        return ;
    }
    int mid=l+r>>1;
    build(u<<1,l,mid),build(u<<1|1,mid+1,r);
    push_up(u);
}

void modify(int u,int l,int r,int add,int mul){
    if(tr[u].l>=l&&tr[u].r<=r){
        val(tr[u],add,mul);
        return ;
    }
    push_down(u);
    int mid=tr[u].l+tr[u].r>>1;
    if(l<=mid) modify(u<<1,l,r,add,mul);
    if(r>mid) modify(u<<1|1,l,r,add,mul);
    push_up(u);
}

int query(int u,int l,int r){
    if(tr[u].l>=l&&tr[u].r<=r) return tr[u].sum;
    push_down(u);
    int mid=tr[u].l+tr[u].r>>1;
    int res=0;
    if(l<=mid) res=(LL)(res+query(u<<1,l,r))%p;
    if(r>mid) res=(LL)(res+query(u<<1|1,l,r))%p;
    return res;
}

int main(){
    read(n),read(m),read(p);
```

```
    for(int i=1;i<=n;i++)
        read(w[i]);

    build(1,1,n);

    for(int i=1;i<=m;i++){
        int op,l,r,k;
        read(op),read(l),read(r);
        if(op==1){
            read(k);
            modify(1,l,r,0,k);
        }
        else if(op==2){
            read(k);
            modify(1,l,r,k,1);
        }
        else{
            printf("%d\n",query(1,l,r));
        }
    }
    return 0;
}
```

当有多个需要维护的懒标记时，即有多个懒标记的时候：

```
void cal(Node &root,int add,int mul){
    root.sum=root.sum*mul+add*(root.r-root,l+1);
    root.mul=root.mul*mul;
    root.add=root.add*mul+add;
}
//用子节点来更新父节点
void push_up(int u){
    tree[u].sum=tree[u<<1].sum+tr[u<<1|1].sum;
    tree[u].add=0,tree[u].mul=1;
}
//用父节点来更新子节点
void push_down(int u){
    cal(tree[u<<1],tree[u].add,tree[u].mul);
    cal(tree[u<<1|1],tree[u].add,tree[u].mul);
    //恢复
    root.add=0,root.mul=1;
}
```

## 势能线段树

- 以区间开方为例

```
#include<bits/stdc++.h>
using namespace std;
#define N 200010
#define int long long
typedef long long LL;
#define int long long

int n,m;
```

```
int w[N];
struct Node{
    int l,r;
    LL sum,max,min;
    LL lazy,set;   //set是区间置数
}tr[N<<4];

void push_up(int u){
    tr[u].sum=tr[u<<1].sum+tr[u<<1|1].sum;
    tr[u].min=min(tr[u<<1].min,tr[u<<1|1].min);
    tr[u].max=max(tr[u<<1].max,tr[u<<1|1].max);
    tr[u].lazy=tr[u].set=0;
}

void val(Node &u,int lazy,int set){
    if(lazy){
        u.sum+=lazy*(u.r-u.l+1);
        u.max+=lazy;
        u.min+=lazy;
        if(u.set)
            u.set+=lazy;
        else
            u.lazy+=lazy;
    }
    if(set){
        u.sum=set*(u.r-u.l+1);
        u.max=set;
        u.min=set;
        u.lazy=0;
        u.set=set;
    }
}

void push_down(int u){
    val(tr[u<<1], tr[u].lazy, tr[u].set);
    val(tr[u<<1|1], tr[u].lazy, tr[u].set);
    tr[u].lazy=0,tr[u].set=0;
}

void build(int u,int l,int r){
    tr[u]={l,r};
    if(l==r){
        tr[u].sum=tr[u].max=tr[u].min=w[l];
        tr[u].lazy=tr[u].set=0;
        return ;
    }
    int mid= l+r >> 1;
    build(u<<1,l,mid),build(u<<1|1,mid+1,r);
    push_up(u);
}

void tadd(int u,int l,int r,int x){
    if(tr[u].l>=l&&tr[u].r<=r){
        val(tr[u],x,0);
        return ;
    }
    push_down(u);
    int mid=tr[u].l+tr[u].r>>1;
```

```
    if(l<=mid) tadd(u<<1,l,r,x);
    if(r>mid) tadd(u<<1|1,l,r,x);
    push_up(u);
}

void tsqrt(int u,int l,int r){
    if(tr[u].l>=l&&tr[u].r<=r){
        if(tr[u].min==tr[u].max){
            val(tr[u],0,sqrt(tr[u].max));
            return ;
        }
        push_down(u);
        int mid=tr[u].l+tr[u].r>>1;
        if(l<=mid) tsqrt(u<<1,l,r);
        if(r>mid) tsqrt(u<<1|1,l,r);
        push_up(u);
        return ;
    }
    push_down(u);
    int mid=tr[u].l+tr[u].r>>1;
    if(l<=mid) tsqrt(u<<1,l,r);
    if(r>mid) tsqrt(u<<1|1,l,r);
    push_up(u);
}

int query(int u,int l,int r){
    if(tr[u].l>=l&&tr[u].r<=r) return tr[u].sum;
    push_down(u);
    int mid=tr[u].l+tr[u].r>>1;
    int ans=0;
    if(l<=mid) ans+=query(u<<1,l,r);
    if(r>mid) ans+=query(u<<1|1,l,r);
    return ans;
}

signed main(){
    scanf("%lld%lld",&n,&m);
    for(int i=1;i<=n;i++)
        scanf("%lld",&w[i]);

    build(1,1,n);

    for(int i=1;i<=m;i++){
        int op,l,r,x;
        scanf("%lld%lld%lld",&op,&l,&r);
        if(op==1){
            tsqrt(1, l, r);
        }
        else if(op==3){
            printf("%lld\n",query( 1, l, r));
        }
        else if(op==2){
            scanf("%lld",&x);
            tadd( 1, l, r, x);
        }
    }
    return 0;
}
```

## 动态开点

最初只建立出根节点代表整个区间，等到需要访问某个子区间的时候再建立出子区间的结点。

```
struct Node{
    int ls,rs;  //注意，这里维护的是左右子节点的编号
    int dat;    //维护信息，例如区间最大值
}tr[N<<2];
int root,idx;

void push_up(int u){
    tr[u].dat = mxa(tr[tr[u].ls].dat, tr[tr[u].rs].dat);
}

int build(){
    tr[++idx].ls = tr[idx].rs = tr[idx].dat = 0;
    return tot; //返回当前新区间的结点编号
}

//单点修改，把pos位置加上delta
void insert(int u,int l,int r,int pos,int delta){
    if(l==r){
        tr[u].dat += delta;
        return ;
    }
    int mid = l+r >> 1;
    if(pos <= mid) {
        if(!tr[u].ls) tr[u].ls = build(); //动态开点
        insert(tr[u].ls, l, mid, pos, delta);
    }
    else {
        if(!tr[u].rs) tr[u].rs = build(); //动态开点
        insert(tr[u].rs, mid+1, r, pos, delta);
    }
    push_up(u);
}

int main(){
    idx=0;
    root = build();
    insert(root,1,n,pos,delta);
}
```

## 线段树合并

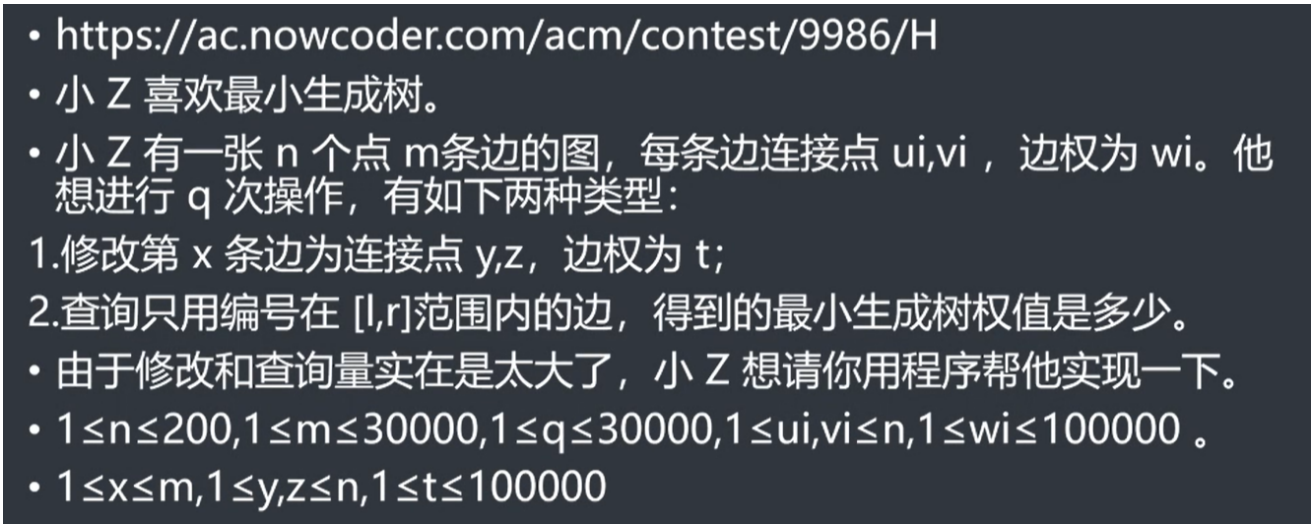
倘若有若干棵都维护[1,n] ,所有操作完成后希望把这些线段树对应位置的值相加，同时维护区间最大值，这时需要使用线段树的合并算法

> 用两个指针p,q从两个根节点出发，以递归的方式同步遍历两棵线段树，即p和q总是指向相同的子区间
>
> 如果p,q之一为空，则以非空的那个作为合并后的节点
>
> 若p,q均不为空，则递归合并两棵左子树和右子树，然后删除结点q，以p为合并后的结点。自底向上更新最值信息。若已到达叶子节点，则直接将两个最值相加即可。

```
int merge(int p,int q,int l,int r){
    if(!p) return q;
    if(!q) return p;
    if(l==r){
        tr[p].dat += tr[q].dat;
        return p;
    }
    int mid = l+r >> 1;
    tr[p].ls = merge(tr[p].ls,tr[q].ls,l,mid);
    tr[p].rs = merge(tr[p].rs,tr[q].rs,mid+1,r);
    tr[p].dat = max(tr[tr[p].ls].dat, tr[tr[p].rs].dat);
    return p;
}
```

### 例子：动态最小生成树

动态最小生成树




```
#include<bits/stdc++.h>
using namespace std;
struct ty
{
    int x, y, z;
}edge[30100];
int n, m, q;
int tree[4 * 30010][420]; //当前区间内的边在这个区间的边组成的最小生成树里的有哪些（一条一条从小到大存下来）（也可能边还不够没有联通）
int fa[30100];
int ans[420];
int find(int x){
    return fa[x] == x ? x : fa[x] = find(fa[x]);
}
void merge(int x, int y)
{
    fa[find(x)] = find(y);
}
void  pushup(int x, int l, int r)
{
    for (int i = 1; i <= n; i++)
        fa[i] = i;
    for (int i = 1; i <= n; i++)
        tree[x][i] = 0;
```

```
    int p = 1, q = 1;
    for(int i = 0; i < n ; ) //最多有n-1条生成树上的边
    {
        int e1 = tree[l][p];
        int e2 = tree[r][q];
        if (e1 == 0 && e2 ==0) break;
        if(e2 == 0  || (e1 != 0 && edge[e1].z <= edge[e2].z))
        {
            int tx = edge[e1].x;
            int ty = edge[e1].y;
            if (find(tx) != find(ty) )
            {
                merge(tx, ty);
                i++;
                tree[x][i] = e1;
            }
            p++;
        }
        else{
            int tx = edge[e2].x;
            int ty = edge[e2].y;
            if (find(tx) != find(ty) )
            {
                merge(tx, ty);
                i++;
                tree[x][i] = e2;
            }
            q++;
        }
    }
}

int tmp[220];
void  hebin(int l)
{
    for (int i = 1; i <= n; i++)
        fa[i] = i;
    for (int i = 1; i <= n; i++)
        tmp[i] = 0;

    int p = 1, q = 1;
    for(int i = 0; i < n; ) //最多有n-1条生成树上的边
    {
        int e1 = tree[l][p];
        int e2 = ans[q];
        if (e1 == 0 && e2 ==0) break;
        if(e2 == 0  || (e1 != 0 && edge[e1].z <= edge[e2].z))
        {
            int tx = edge[e1].x;
            int ty = edge[e1].y;
            if (find(tx) != find(ty) )
            {
                merge(tx, ty);
                i++;
                tmp[i] = e1;
            }
            p++;
```

```
        }
        else{
            int tx = edge[e2].x;
            int ty = edge[e2].y;
            if (find(tx) != find(ty) )
            {
                merge(tx, ty);
                i++;
                tmp[i] = e2;
            }
            q++;
        }
    }
    for (int i = 1; i <= n; i++)
        ans[i] = tmp[i];
}

void build(int p, int l, int r)
{
    if (l ==r)
    {
        tree[p][1] = l;
        return;
    }
    int mid = (l + r)/ 2;
    build(p*2, l, mid);
    build(p*2+1, mid+1, r);
    pushup(p, p*2, p*2+1);
}
void change(int p, int l, int r, int pos)
{
    if (l == r)
    {
       // tree[p][1] = l;
        return ;
    }
    int mid = (l + r)/ 2;
    if (pos <= mid) change(p*2, l, mid, pos);
    else change(p*2+1, mid + 1, r, pos);
    pushup(p, p*2, p*2+1);
}

void query(int p, int l, int r, int x, int y)
{
    if (x <= l && r <= y)
    {
        hebin(p);//把区间p的答案合并到ans数组里面去
        return ;
    }
    int mid = (l + r)/ 2;
    if (x <= mid) query(p*2, l, mid, x, y);
    if (y > mid) query(p*2+1, mid + 1, r, x, y);
}

int main()
{
    scanf("%d%d%d", &n, &m, &q);
    for (int i = 1; i <= m; i++)
```

```
        scanf("%d%d%d", &edge[i].x, &edge[i].y, &edge[i].z);
    build(1, 1, m);
    while (q--)
    {
        int op;
        scanf("%d", &op);
        if (op== 1)
        {
            int num, x, y, z;
            scanf("%d%d%d%d", &num, &x, &y, &z);
            edge[num].x = x;
            edge[num].y = y;
            edge[num].z = z;
            change(1, 1, m, num);
        }
        else
        {
            int x, y;
            scanf("%d%d", &x, &y);

            memset(ans, 0, sizeof(ans));
            query(1, 1, m, x, y);

            long long sum = 0;
            if (ans[n - 1] == 0) printf("Impossible\n");
            else{
                for (int i = 1; i < n; i++) sum += edge[ans[i]].z;
                printf("%lld\n", sum);
            }
        }
    }


    return 0;
}
```

## 可持久化线段树 (主席树)

```
//====================================
struct HJT_Tree{
    struct Node{
        int ls,rs;
        int cnt;
    }tr[N<<5];
    int root[N],idx=0;
    #define push_up(rt) tr[rt].cnt=tr[tr[rt].ls].cnt+tr[tr[rt].rs].cnt
    int build(int l,int r){
        int rt=++idx;
        if(l==r) return rt;
        int mid=l+r>>1;
        tr[rt].ls=build(l,mid); tr[rt].rs=build(mid+1,r);
        return rt;
    }

    int insert(int pre,int l,int r,int x){
```

```
        int rt=++idx;
        tr[rt]=tr[pre];
        if(l==r) {tr[rt].cnt++; return rt;}
        int mid=l+r>>1;
        if(x<=mid) tr[rt].ls=insert(tr[pre].ls,l,mid,x);
        else tr[rt].rs=insert(tr[pre].rs,mid+1,r,x);
        push_up(rt);
        return rt;
    }

    int query(int now,int pre,int l,int r,int k){
        if(l==r) return l;
        int cnt=tr[tr[now].ls].cnt-tr[tr[pre].ls].cnt;
        int mid=l+r>>1;
        if(cnt>=k) return query(tr[now].ls, tr[pre].ls, l, mid, k);
        else return query(tr[now].rs, tr[pre].rs, mid+1, r, k-cnt);
    }

}T;

//区间mex,主席树维护当前区间出现的下标最小值
struct HJT_Tree{
    struct Node{
        int ls,rs;
        int mn;
    }tr[N<<5]; int root[N],idx=0;

    void push_up(int u){
        tr[u].mn=min(tr[tr[u].ls].mn, tr[tr[u].rs].mn);
    }

    void build(int &u,int l,int r){
        u=++idx;
        if(l==r) return ;
        int mid=l+r>>1;
        build(tr[u].ls, l, mid); build(tr[u].rs, mid+1, r);
        push_up(u);
    }

    int insert(int pre,int l,int r,int x,int id){
        int u=++idx;
        tr[u]=tr[pre];
        if(l==r){
            tr[u].mn=id;
            return u;
        }
        int mid=l+r>>1;
        if(x<=mid) tr[u].ls=insert(tr[pre].ls, l, mid, x, id);
        else tr[u].rs=insert(tr[pre].rs, mid+1, r, x, id);
        push_up(u);
        return u;
    }

    int query(int u,int L,int l,int r){
        if(l==r) return l;
        int mid=l+r>>1;
        if(tr[tr[u].ls].mn<L) return query(tr[u].ls, L, l, mid);
        else if(tr[tr[u].rs].mn<L) return query(tr[u].rs, L, mid+1, r);
```

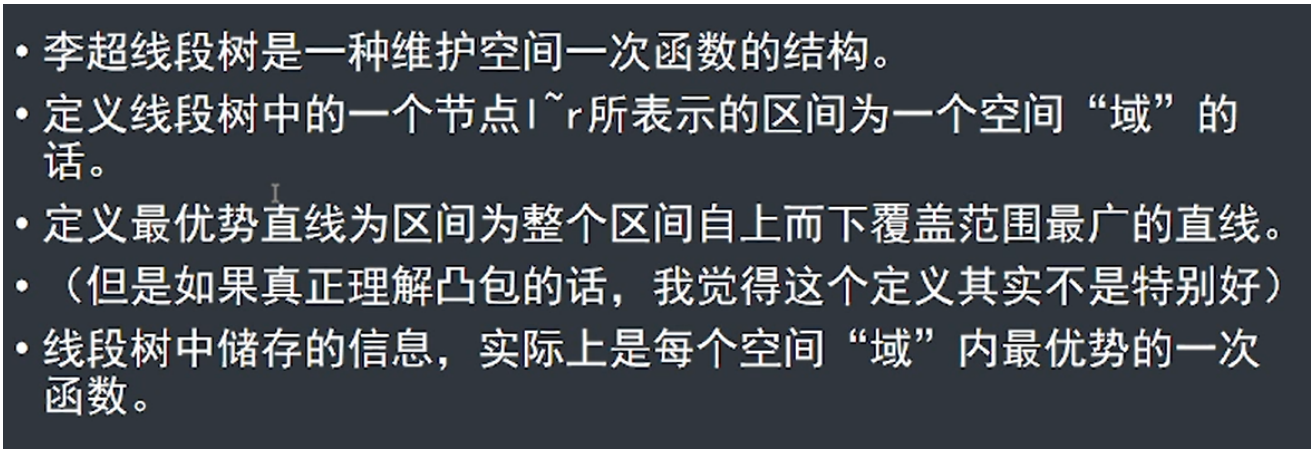
```
        else return r+1;
    }
}T;
//=====================================
//read(x)
template <typename T>void read(T &x){x=0;int f=1;char ch=getchar();while(!isdigit(ch)){if(ch=='-
')f=-1;ch=getchar();}while(isdigit(ch)){x=x*10+(ch^48);ch=getchar();}x*=f;return;}
//write(x)
template <typename T>void write(T x){if(x<0){putchar('-');x=-
x;}if(x>9)write(x/10);putchar(x%10+'0');return;}

struct Node{
    int l,r;  //left tree point&righht tree point
    int cnt;
}tr[N*30];
int root[N],idx=0;
int a[N];
vector<int> alls;
int n,m;

int get_id(int x){return lower_bound(alls.begin(),alls.end(),x)-alls.begin();}

int build(int l,int r){
    int now=++idx;
    if(l==r) return now;
    int mid=l+r>>1;
    tr[now].l=build(l,mid),tr[now].r=build(mid+1,r);
    return now;
}

int insert(int pre,int l,int r,int x){
    int now=++idx;
    tr[now]=tr[pre]; //paste previos
    if(l==r){
        tr[now].cnt++;
        return now;
    }
    int mid=l+r>>1;
    if(x<=mid) tr[now].l=insert(tr[pre].l,l,mid,x);
    else tr[now].r=insert(tr[pre].r,mid+1,r,x);
    tr[now].cnt=tr[tr[now].l].cnt+tr[tr[now].r].cnt;
    return now;
}

int query(int now,int pre,int l,int r,int k){
    if(l==r) return r;
    int cnt=tr[tr[now].l].cnt-tr[tr[pre].l].cnt;
    int mid=l+r>>1;
    if(k<=cnt) return query(tr[now].l,tr[pre].l,l,mid,k);
    else return query(tr[now].r,tr[pre].r,mid+1,r,k-cnt); //k should minus cnt of left
}

int main(){
    read(n),read(m);
    for(int i=1;i<=n;i++){
        read(a[i]);
        alls.push_back(a[i]);
    }
```

```
    sort(alls.begin(),alls.end());
    alls.erase(unique(alls.begin(),alls.end()),alls.end());

    root[0]=build(0,alls.size()-1);

    for(int i=1;i<=n;i++)
        root[i]=insert(root[i-1],0,alls.size()-1,get_id(a[i]));

    for(int i=1;i<=m;i++){
        int l,r,k;
        read(l),read(r),read(k);
        write(alls[query(root[r],root[l-1],0,alls.size()-1,k)]);
        puts("");
    }
    return 0;
}
```

**主席树的区间修改**

```
void change(int &root,int pre,int ul,int ur,int l,int r,int v){  //区间修改
    root=++idx;
    cpy(pre, root);
    if(ul>=l&&ur<=r){
        tr[root].lazy += v;
        return ;
    }
    int mid=ul+ur>>1;
    if(l<=mid) change(tr[root].ls, tr[root].ls, ul, mid, l, r, v);
    if(r>mid) change(tr[root].rs, tr[root].rs, mid+1, ur, l, r, v);
    return ;
}
```

## 李超树




区间[l,r]中存放的是一次函数，存放最优势直线


mid'左侧蓝色线为最优势直线，而右侧为红色直线更占优

```
struct Line{
    int k, b;
    Line(int _k=0,int _b=0){
        k=_k,b=_b;
    }
    int val(int x){
        return k*x+b;
    }
};

double L_cross_x(Line &A,Line &B){
    return 1.0*(1.0*B.b-A.b)/(1.0*A.k-B.k);
}

struct LiChao_SegTree{
    int cross_x(const Line &A,const Line &B){
        return (B.b-A.b)/(A.k-B.k);
    }
    struct Node{
        int l,r;
        bool vis,has_line;
        Line line;
    }tr[N<<2];

    void build(int u,int l,int r){
        tr[u].l=l,tr[u].r=r;
        tr[u].vis=false;
        tr[u].has_line=false;
        if(l==r) return ;
        int mid=l+r>>1;
        if(l<=mid) build(u<<1, l, mid);
        if(r>mid) build(u<<1|1, mid+1, r);
    }

    void insert(int u,int l,int r,Line s){
        tr[u].vis=true;
        if(tr[u].l>=l&&tr[u].r<=r){
            if(!tr[u].has_line){
                tr[u].line=s;
                tr[u].has_line=true;
                return ;
            }
```

```
            if(tr[u].line.val(tr[u].l)>=s.val(tr[u].l)&&tr[u].line.val(tr[u].r)>=s.val(tr[u].r))
{
                return ;
            }
            if(tr[u].line.val(tr[u].l)<s.val(tr[u].l)&&tr[u].line.val(tr[u].r)<s.val(tr[u].r)){
                tr[u].line=s;
                return ;
            }
            int mid=tr[u].l+tr[u].r>>1;
            if(cross_x(s,tr[u].line)<=mid) {
                if(s.k<tr[u].line.k) insert(u<<1,l,r,s);
                else{
                    insert(u<<1,l,r,tr[u].line);
                    tr[u].line=s;
                }
            }
            else{
                if(s.k>tr[u].line.k)   insert(u<<1|1,l,r,s);
                else{
                    insert(u<<1|1,l,r,tr[u].line);
                    tr[u].line=s;
                }
            }
            return ;
        }
        insert(u<<1,l,r,s);
        insert(u<<1|1,l,r,s);
    }

    void clear(int u=1){
        tr[u].vis=false;
        tr[u].has_line=false;
        if(tr[u].l==tr[u].r) return ;
        int mid=tr[u].l+tr[u].r>>1;
        if(tr[u].l<=mid) clear(u<<1);
        if(tr[u].r>mid) clear(u<<1|1);
    }

    int get_val(int u,int x){
        if(!tr[u].vis) return -INF;
        int ret=-INF;
        if(!tr[u].has_line) ret=-INF;
        else ret=tr[u].line.val(x);
        if(tr[u].l==tr[u].r) return ret;

        int mid=tr[u].l+tr[u].r>>1;
        if(x<=mid) return max(ret, get_val(u<<1,x));
        else return max(ret, get_val(u<<1|1,x));
    }
}tr_mx,tr_mn;
```

# 数点问题

**一维数点问题**

1.给定一个数组a，q次询问，每次求数组中[l,r]内小于h的个数

**1. 离线+树状数组统计下标**

先对数组a[]按照值域进行排序，然后对于每个询问的h从小到大进行排序。每次依次扫描每个询问，把所有小于h的数全部加入，每次询问的答案是query(r)-query(l-1)

```
int n,m,tr[N],ans[N];
pii a[N];
struct Query{
    int l,r,h,id;
    bool operator<(const Query& A){
        return h<A.h;
    }
}ques[N];
#define lowbit(x) ((x)&(-x))
int ask(int x){
    int ans=0;
    for(int i=x;i;i-=lowbit(i)) ans+=tr[i];
    return ans;
}
void add(int x,int d){for(int i=x;i<=n;i+=lowbit(i)) tr[i]+=d;}
void solve(){
    n=read(),m=read();
    memset(tr,0,(n+2)*sizeof(int));
    rep(i,1,n) a[i].x=read(),a[i].y=i;
    sort(a+1,a+1+n,[](pii &A,pii &B){
        if(A.x!=B.x) return A.x<B.x;
        return A.y<B.y;
    });
    rep(i,1,m){
        ques[i].l=read(),ques[i].r=read(),ques[i].h=read(),ques[i].id=i;
    }
    sort(ques+1,ques+1+m,[](Query& A,Query& B){
        if(A.h!=B.h) return A.h<B.h;
        return A.id<B.id;
    });
    int j=1;
    for(int i=1;i<=m;++i){
        int id=ques[i].id,h=ques[i].h,l=ques[i].l,r=ques[i].r;
        while(j<=n&&a[j].x<=h){
            add(a[j].y, 1);
            j++;
        }
        if(l==1) ans[id]=ask(r);
        else ans[id]=ask(r)-ask(l-1);
    }
    for(int i=1;i<=m;++i) printf("%d ",ans[i]);
    printf("\n");
}
```

**2.莫队+值域分块**

用莫队算法处理区间，然后对于值域可以进行分块（即使1e9也可以离散化后分块）

P4396 [AHOI2013]作业

```
const int N=200010,M=N*2,mod=1e9+7;
int n,m,V,T,K;
int blockv[N],block[N];
int sumv[N],blocksum[N],sum[N]; //sum[i]表示i出现的个数，sumv[i]表示值i是否出现
int a[N];
struct Q{
    int l,r,a,b,id;
}q[N];
int ans[N],res[N];

int get_ans(int l,int r){
    int ans=0;
    r=min(r,V);
    if(l>V) return 0;
    int ll=blockv[l], rr=blockv[r];
    ll++,rr--;
    if(blockv[l]==blockv[r]){
        for(int i=l;i<=r;++i)
            ans += sum[i];
        return ans;
    }
    for(int i=ll;i<=rr;++i) ans+=blocksum[i];
    for(int i=l;blockv[i]==blockv[l]&&l<=V;++i) ans+=sum[i];
    for(int i=r;blockv[i]==blockv[r]&&r>=0;--i) ans+=sum[i];
    return ans;
}

int get_res(int l,int r){
    int res=0;
    r=min(r,V);
    if(l>V) return 0;
    int ll=blockv[l],rr=blockv[r];
    ll++,rr--;
    if(blockv[l]==blockv[r]){
        for(int i=l;i<=r;++i) res+=(bool)sum[i];
        return res;
    }
    for(int i=ll;i<=rr;++i) res+=sumv[i];
    for(int i=l;blockv[i]==blockv[l]&&l<=V;++i) res+=(bool)sum[i];
    for(int i=r;blockv[i]==blockv[r]&&r>=0;--i) res+=(bool)sum[i];
    return res;
}

void add(int pos){
    if(sum[a[pos]]==0) ++sumv[blockv[a[pos]]];
    ++sum[a[pos]];
    ++blocksum[blockv[a[pos]]];
}

void del(int pos){
    sum[a[pos]] --;
    blocksum[blockv[a[pos]]] --;
```

```
    if(sum[a[pos]]<=0) sumv[blockv[a[pos]]]--;
}

void solve(){
    n=read(),m=read();
    T=1.0*n/sqrt(m)+1;
    rep(i,1,n) a[i]=read(),V=max(V,a[i]),block[i]=i/T;
    K=sqrt(V+1.0); //值域分块
    rep(i,0,V) blockv[i]=i/K;
    rep(i,1,m) {
        q[i].l=read(),q[i].r=read();
        q[i].a=read(),q[i].b=read();
        q[i].id=i;
    }
    sort(q+1,q+1+m,[](Q &A,Q &B){
        int al=block[A.l],bl=block[B.l];
        // return  (al^bl)?(al<bl):(A.r<B.r);
        if(al!=bl) return al<bl;
        return A.r<B.r;
    });
    for(int i=1,l=1,r=0;i<=m;++i){
        int ll=q[i].l,rr=q[i].r,aa=q[i].a,bb=q[i].b,id=q[i].id;
        while(r<rr) add(++r);
        while(r>rr) del(r--);
        while(l<ll) del(l++);
        while(l>ll) add(--l);
        ans[id]=get_ans(aa,bb);
        res[id]=get_res(aa,bb);
    }
    for(int i=1;i<=m;++i) printf("%d %d\n",ans[i],res[i]);
    printf("\n");
}
```

# 二维数点问题

## 2.离线+树状数组

这种做法就是按照横坐标排序后对纵坐标离散化后用树状数组维护位置。

```
#define N 1000010
struct BIT{
    int n,tr[N];
    #define lowbit(x) ((x)&(-x))
    void resize(int _n){n=_n;}
    void add(int x,int d){
        for(;x<=n;x+=lowbit(x)) tr[x]+=d;
    }
    LL ask(int x){
        LL ans=0;
        for(;x;x-=lowbit(x)) ans+=tr[x];
        return ans;
    }
}T;
int n,m,k,ans[N];
vector<int> alls;
int get(int x){
    return lower_bound(alls.begin(), alls.end(), x) - alls.begin()+1;
```

```
}
struct  Pos{
    int x,y,z,p;
    bool operator<(const Pos& W)const{
        if(x!=W.x) return x < W.x;
        return z<W.z;
    }
};
vector<Pos> q;
//================================
signed main(){
    n=read(),m=read();
    rep(i,1,n){
        int x=read(),y=read();
        q.push_back({x,y,0,0});
        alls.push_back(y);
    }
    rep(i,1,m){
        int x1=read(),x2=read(),y1=read(),y2=read();
        q.push_back({x2,y2,i,1});
        q.push_back({x1-1,y1-1,i,1});
        q.push_back({x1-1,y2,i,-1});
        q.push_back({x2,y1-1,i,-1});
        alls.push_back(y1);
        alls.push_back(y2);
        alls.push_back(y1-1);
    }
    sort(alls.begin(), alls.end());
    alls.erase(unique(alls.begin(),alls.end()), alls.end());
    sort(q.begin(), q.end());
    T.resize(alls.size()+1);
    for(auto u:q){
        int x=u.x, y=u.y,z=u.z,p=u.p;
        y=get(y);
        if(z==0){
            T.add(y, 1);
        }
        else{
            int tmp=T.ask(y);
            ans[z]+=tmp*p;
        }
    }
    for(int i=1;i<=n;++i) print(ans[i]);
    return 0;
}
```

## CDQ分治

模板：（使用归并排序的分治解法）

给定 $n$ 个元素（编号 $1 \sim n$），其中第 $i$ 个元素具有 $a_i, b_i, c_i$ 三种属性。

设 $f(i)$ 表示满足以下 4 个条件：

1. $a_j \leq a_i$
2. $b_j \leq b_i$
3. $c_j \leq c_i$
4. $j \neq i$

的 $j$ 的数量。

对于 $d \in [0, n)$，求满足 $f(i) = d$ 的 $i$ 的数量。

**输入格式**

第一行两个整数 $n, k$，表示元素数量和最大属性值。

接下来 $n$ 行，其中第 $i$ 行包含三个整数 $a_i, b_i, c_i$，分别表示第 $i$ 个元素的三个属性值。

**输出格式**

共 $n$ 行，每行输出一个整数，其中第 $d+1$ 行的整数表示满足 $f(i) = d$ 的 $i$ 的数量。

```
const int N=200010,M=N*2,mod=1e9+7,tot=200000;
struct BIT{
    #define N 200010
    #define lowbit(x) ((x)&(-x))
    const int tot=200000;
    int tr[N];
    void add(int x,int d){for(;x<=tot;x+=lowbit(x))tr[x]+=d;}
    int ask(int x){
        int ans=0;
        for(;x;x-=lowbit(x)) ans+=tr[x];
        return ans;
    }
}B;
int n,m,k,ans[N];
struct Poi{
    int a,b,c,s,res;
    bool operator<(const Poi &W)const{
        if(a!=W.a) return a<W.a;
        if(b!=W.b) return b<W.b;
        return c<W.c;
    }
    bool operator == (const Poi& W)const{
        return a==W.a && b==W.b&& c==W.c;
    }
}p[N];

Poi tmp[N];
void merge(int l,int r){
    if(l>=r) return ;
    int mid=l+r>>1;
    merge(l,mid); merge(mid+1,r);
    int i=l,j=mid+1,cnt=0;
    while(i<=mid&&j<=r){
        if(p[i].b<=p[j].b) B.add(p[i].c, p[i].s), tmp[++cnt]=p[i++];
        else p[j].res+=B.ask(p[j].c), tmp[++cnt]=p[j++];
    }
```

```
    while(i<=mid) B.add(p[i].c, p[i].s), tmp[++cnt]=p[i++];
    while(j<=r) p[j].res+=B.ask(p[j].c), tmp[++cnt]=p[j++];
    for(int i=l;i<=mid;++i) B.add(p[i].c, -p[i].s);
    for(int i=l,j=1;i<=r;++i,++j) p[i]=tmp[j];
}

void solve(){
    n=read(),m=read();
    rep(i,1,n){
        p[i].a=read(),p[i].b=read(),p[i].c=read();
        p[i].s=1;
    }
    sort(p+1,p+1+n);  //先按照三关键字排序
    int k=1;
    for(int i=2;i<=n;++i){
        if(p[i]==p[k]) p[k].s++;
        else p[++k]=p[i];
    }
    merge(1, k);
    for(int i=1;i<=k;++i){
        ans[p[i].res+p[i].s-1]+=p[i].s;
    }
    for(int i=0;i<n;++i) print(ans[i]);
}
```

CDQ分治的应用：

1.二维数点问题

对于要求统计的二维平面的数点问题分拆成三个维度：x,y,z其中z表示是否是查询的的点

```
    rep(i,1,n){
        int x=read(),y=read(),P=read();
        p[++idx]={x,y,0,P};
    }
    rep(i,1,m){
        int x1=read(),y1=read(),x2=read(),y2=read();
        p[++idx]={x1-1, y1-1, 1, 0, 1, i};
        p[++idx]={x1-1, y2, 1, 0, -1, i};
        p[++idx]={x2, y1-1, 1, 0, -1, i};
        p[++idx]={x2, y2, 1, 0, 1, i};
    }
    sort(p+1,p+1+idx);
```

然后在归并的过程中算出结果即可。

```
void merge(int l,int r){
    if(l>=r) return ;
    int mid=l+r>>1;
    merge(l,mid), merge(mid+1,r);
    int i=l,j=mid+1,cnt=0;
    LL sum=0;
    while(i<=mid&&j<=r){
        if(p[i].y<=p[j].y) sum +=(!p[i].z)*p[i].p, tmp[++cnt]=p[i++];
        else p[j].sum+=sum,tmp[++cnt]=p[j++];
    }
    while(i<=mid) sum += (!p[i].z)*p[i].p, tmp[++cnt]=p[i++];
```

```
    while(j<=r) p[j].sum+=sum,tmp[++cnt]=p[j++];
    for(int i=l,j=1;i<=r;) p[i++]=tmp[j++];
}
```

2.统计LIS的数量：

```
void CDQ(int l,int r){
    if(l>=r) return ;
    int mid=l+r>>1;
    CDQ(l,mid);
    int idx=0;
    rep(k,l,r) tmp[++idx]=k;
    sort(tmp+1, tmp+1+idx,[](int &x,int &y){
        return a[x]==a[y]?x>y:a[x]<a[y];
    });
    int mx=0; LL num=0;
    for(int i=1;i<=idx;++i){
        int x=tmp[i];
        if(x>mid){
            if(mx+1>f[x])
                f[x]=mx+1,cnt[x]=num;
            else if(mx+1==f[x])
                cnt[x]=Mod(cnt[x]+num,mod);
        }
        else{
            if(f[x]>mx)
                mx=f[x], num=cnt[x];
            else if(f[x]==mx)
                num=Mod(num+cnt[x],mod);
        }
    }
    CDQ(mid+1, r);
}
```

# *分块&Mo's Algorithm*

## 分块

一个简单的整数问题：

> 两个指令来维护一个序列：
>
> `C l r d`把`[l,r]`加上`d`
>
> `Q l r` 询问`[l,r]`的数的和

可以使用分块的思想，假设修改和查询的数据量几乎相当，根据均值不等式可以将块的数量设置为$\sqrt{n}$,然后将每次查询分成两类，分别是直接是大段的和拆分成的小段

修改操作：

- 完整段：

```
add = add + d

sum = sum + d * len
```

- 小段内

```
sum = sum + d

w[i] = w[i] + d
```

## 块状链表

- 在某个位置插入一段
- 将某一段删除
- 将某一段翻转

- 插入一段
  - 分裂节点
  - 在分裂点插入序列

- 删除一段
  - 删除开头结点的后半部分
  - 删除中间的完整结点
  - 删除结尾结点的前半部分

- 合并(防止块数太多影响复杂度)
  - 遍历整个链表，判断当前节点时候能和下一个节点合并，如果可以就将下一个节点合并进当前结点

**不写块状链表，用STL的Rope来代替：**

## Rope的使用

```
#include<ext/rope>
using namespace __gnu_cxx;
```

定义方式：

```
rope<类型> 变量名;
```

当定义rope<char> str时

```
str.substr(pos, len); //返回rope从下标pos开始的len个字符
str.at(x);                //访问下标为x的元素
str.erase(pos, num);      //从rope的小标pos开始删除num个字符
str.copy(int pos, int len, string &s);   //从str下标开始的len个字符用字符串s代替，如果pos后的位数不足就补足
replace(int pos, string &x);    //从str的下标pos开始替换成字符串x，x的长度为从pos开始替换的位数，如果pos后的位数不够就补足
```

## NOI 2003 文本编辑器

文本编辑器：由一段文本和该文本中的一个光标组成的，支持如下操作的数据结构。如果这段文本为空，我们就说这个文本编辑器是空的。

| 操作名称 | 输入文件中的格式 | 功能 |
| --- | --- | --- |
| MOVE(k) | Move k | 将光标移动到第 $k$ 个字符之后，如果 $k$=0，将光标移到文本开头 |
| INSERT(n, s) | Insert n s | 在光标处插入长度为 $n$ 的字符串 $s$，光标位置不变，$n \geq 1$ |
| DELETE(n) | Delete n | 删除光标后的 $n$ 个字符，光标位置不变，$n \geq 1$ |
| GET(n) | Get n | 输出光标后的 $n$ 个字符，光标位置不变，$n \geq 1$ |
| PREV() | Prev | 光标前移一个字符 |
| NEXT() | Next | 光标后移一个字符 |

## rope 版本

```
const int N = 25e5;
int n,t;
char s[N];

inline void reads(char *s,int len){
    s[len]='\0'; len --;
    for(int i=0;i<=len;++i){
        s[i]='\0';
        while(s[i]<32 || s[i]>126) s[i]=getchar();
    }
}
rope<char> ans;

void solve(){
    cin >> n;
    int cur = 0 ;

    while(n--){
        string op; cin >> op;
        if(op=="Move"){
            cin >> t;
            cur = t;
        }
        else if(op=="Insert"){
            cin >> t; reads(s,t);
            ans.insert(cur,s);
        }
        else if(op=="Delete"){
            cin >> t;
            ans.erase(cur,t);
```

```
        }
        else if(op=="Get"){
            cin >> t;
            cout << ans.substr(cur,t) << endl;
        }
        else if(op=="Prev") cur --;
        else if(op=="Next") cur ++;
    }
}
```

## 块状链表版本

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

using namespace std;

const int N = 2000, M = 2010;

int n, x, y;
struct Node
{
    char s[N + 1];
    int c, l, r;
}p[M];
char str[2000010];
int q[M], tt;  // 内存回收

void move(int k)  // 移到第k个字符后面
{
    x = p[0].r;
    while (k > p[x].c) k -= p[x].c, x = p[x].r;
    y = k - 1;
}

void add(int x, int u)  // 将节点u插到节点x的右边
{
    p[u].r = p[x].r, p[p[u].r].l = u;
    p[x].r = u, p[u].l = x;
}

void del(int u)  // 删除节点u
{
    p[p[u].l].r = p[u].r;
    p[p[u].r].l = p[u].l;
    p[u].l = p[u].r = p[u].c = 0;  // 清空节点u
    q[ ++ tt] = u;  // 回收节点u
}

void insert(int k)  // 在光标后插入k个字符
{
    if (y < p[x].c - 1)  // 从光标处分裂
    {
        int u = q[tt -- ];  // 新建一个节点
        for (int i = y + 1; i < p[x].c; i ++ )
```

```
            p[u].s[p[u].c ++ ] = p[x].s[i];
        p[x].c = y + 1;
        add(x, u);
    }
    int cur = x;
    for (int i = 0; i < k;)
    {
        int u = q[tt -- ];  // 创建一个新的块
        while (p[u].c < N && i < k)
            p[u].s[p[u].c ++ ] = str[i ++ ];
        add(cur, u);
        cur = u;
    }
}

void remove(int k)  // 删除光标后的k个字符
{
    if (p[x].c - 1 - y >= k)  // 节点内删
    {
        for (int i = y + k + 1, j = y + 1; i < p[x].c; i ++, j ++ ) p[x].s[j] = p[x].s[i];
        p[x].c -= k;
    }
    else
    {
        k -= p[x].c - y - 1;  // 删除当前节点的剩余部分
        p[x].c = y + 1;
        while (p[x].r && k >= p[p[x].r].c)
        {
            int u = p[x].r;
            k -= p[u].c;
            del(u);
        }
        int u = p[x].r;  // 删除结尾节点的前半部分
        for (int i = 0, j = k; j < p[u].c; i ++, j ++ ) p[u].s[i] = p[u].s[j];
        p[u].c -= k;
    }
}

void get(int k)  // 返回从光标开始的k个字符
{
    if (p[x].c - 1 - y >= k)  // 节点内返回
    {
        for (int i = 0, j = y + 1; i < k; i ++, j ++ ) putchar(p[x].s[j]);
    }
    else
    {
        k -= p[x].c - y - 1;
        for (int i = y + 1; i < p[x].c; i ++ ) putchar(p[x].s[i]);  // 输出当前节点的剩余部分
        int cur = x;
        while (p[cur].r && k >= p[p[cur].r].c)
        {
            int u = p[cur].r;
            for (int i = 0; i < p[u].c; i ++ ) putchar(p[u].s[i]);
            k -= p[u].c;
            cur = u;
        }
        int u = p[cur].r;
        for (int i = 0; i < k; i ++ ) putchar(p[u].s[i]);
```

```
    }
    puts("");
}

void prev()  // 光标向前移动一位
{
    if (!y)
    {
        x = p[x].l;
        y = p[x].c - 1;
    }
    else y -- ;
}

void next()  // 光标向后移动一位
{
    if (y < p[x].c - 1) y ++ ;
    else
    {
        x = p[x].r;
        y = 0;
    }
}

void merge()  // 将长度较短的相邻节点合并，保证块状链表时间复杂度的核心
{
    for (int i = p[0].r; i; i = p[i].r)
    {
        while (p[i].r && p[i].c + p[p[i].r].c < N)
        {
            int r = p[i].r;
            for (int j = p[i].c, k = 0; k < p[r].c; j ++, k ++ )
                p[i].s[j] = p[r].s[k];
            if (x == r) x = i, y += p[i].c;  // 更新光标的位置
            p[i].c += p[r].c;
            del(r);
        }
    }
}

int main()
{
    for (int i = 1; i < M; i ++ ) q[ ++ tt] = i;
    scanf("%d", &n);
    char op[10];

    str[0] = '>';
    insert(1);  // 插入哨兵
    move(1);  // 将光标移动到哨兵后面

    while (n -- )
    {
        int a;
        scanf("%s", op);
        if (!strcmp(op, "Move"))
        {
            scanf("%d", &a);
            move(a + 1);
```

```
        }
        else if (!strcmp(op, "Insert"))
        {
            scanf("%d", &a);
            int i = 0, k = a;
            while (a)
            {
                str[i] = getchar();
                if (str[i] >= 32 && str[i] <= 126) i ++, a -- ;
            }
            insert(k);
            merge();
        }
        else if (!strcmp(op, "Delete"))
        {
            scanf("%d", &a);
            remove(a);
            merge();
        }
        else if (!strcmp(op, "Get"))
        {
            scanf("%d", &a);
            get(a);
        }
        else if (!strcmp(op, "Prev")) prev();
        else next();
    }

    return 0;
}
```

莫队算法是一种离线算法，能够高效的完成序列的暴力查询和暴力修改。目前的理解是通过对所有查询离线排序后，通过分块来提高效率。

## 普通莫队

### 排序方式

通过对区间[l,r]以l所在块为第一关键字，r为第二关键字进行从小到大的排序

### 算法操作

排序后按照顺序应对每个查询，由于每次左右指针只在一个块内移动，因此最大修改次数为$2\sqrt{n}$，因此莫队的时间复杂度为$n\sqrt{n}$

```
for(int k=1,i=0,j=1,res=0;k<=m;++k){
    while(i<r) add(a[++i], res);
    while(i>r) del(a[i--], res);
    while(j<l) del(a[j++], res);
    while(j>l) add(a[--j], res);
    ans[id]=res;
}
```

# 带修莫队

由于带修改操作，因此每个询问需要多记录一条时间轴，表征第几次询问。

## 排序方式

对于区间[l,r]和修改次数t进行以l所在块为第一关键字，r所在块为第二关键字，t为第三关键字进行排序。

## 算法操作

排序后对于每个查询，与原来一样，只是多加一个维度，最后处理不同修改的影响。有一个小trick，对于每次修改，可以将修改的数与修改数组记录的进行交换，无需新开数组记录。

```
for(int k=1,i=0,j=1,t=0,res=0;k<=q1;++k){
    int id=q[k].id,l=q[k].l,r=q[k].r,tn=q[k].t;
    while(i<r) add(w[++i], res);
    while(i>r) del(w[i--], res);
    while(j<l) del(w[j++], res);
    while(j>l) add(w[--j], res);
    while(t < tn){
        t ++;
        if(mo[t].pos>=j&&mo[t].pos<=i){
            del(w[mo[t].pos], res);
            add(mo[t].c, res);
        }
        swap(w[mo[t].pos], mo[t].c);
    }
    while(t > tn){
        if(mo[t].pos>=j&&mo[t].pos<=i){
            del(w[mo[t].pos], res);
            add(mo[t].c, res);
        }
        swap(w[mo[t].pos], mo[t].c);
        t--;
    }
    ans[id]=res;
}
```

# 回滚莫队

回滚莫队主要解决单个操作无法实现的问题，例如增加或者删除不好直接实现，例如求最大值等操作

## 排序方式

此处正常按照区间[l,r]的l所在块为第一关键字，r的大小为第二关键字进行排序

## 算法实现

按照排序顺序处理

- 初始化莫队区间为l=询问左端点所在块的最右结点+1,r=询问左端点所在块的最右
  - 如果询问的左右端点在一个块内，直接暴力求解，时间复杂度最多$\sqrt{n}$
  - 对于所有左端点在一个块内的，莫队区间右指针向右移动并更新答案。备份答案后左指针向左移动并记录答案。然后左指针回溯到原来初始化的位置，当前答案值回溯至备份答案。 $2\sqrt{n}$

**第一种实现方式**

```
for(int nw=1;nw<=m;){
    int af=nw;
    //对于所有左端点在一个块内的
    int right = len*get(q[nw].l)+len-1;
    //暴力处理块内
    while(af<=m&&get(q[af].l) == get(q[nw].l)) af++;
    while(nw<af&&q[nw].r<=right){
        LL res=0;
        int id=q[nw].id,l=q[nw].l,r=q[nw].r;
        for(int i=l;i<=r;++i) add(a[i], res);
        ans[id]=res;
        for(int i=l;i<=r;++i) cnt[a[i]]--;
        nw++;
    }

    //莫队处理跨区间的
    LL res=0;
    int j=right+1,i=right;
    while(nw<af){
        int id=q[nw].id,l=q[nw].l,r=q[nw].r;
        while(i<r) add(a[++i], res);
        LL res_backup = res; //存储回溯点
        while(j>l) add(a[--j], res);
        ans[id]=res;
        while(j<right+1) cnt[a[j++]]--;
        res = res_backup;
        nw++;
    }
    // rep(i,lmx,rmx) cnt[i]=0;
    memset(cnt,0,sizeof cnt);
}
```

**第二种实现方式**

```
int left=1,right=0,last_block=-1,__l; //左指针，右指针，上一个块
LL res=0,backup=0;  //答案和备份
for(int i=1;i<=m;++i){
    int l=q[i].l,r=q[i].r,id=q[i].id;
    //如果在一块内，在单独开的数组内直接暴力
    if(get(l) == get(r)){
        rep(j,l,r) __cnt[a[j]] ++;
        rep(j,l,r) ans[id]=max(ans[id], 1ll*__cnt[a[j]]*alls[a[j]]);
        rep(j,l,r) __cnt[a[j]] --;
        continue;
    }

    //访问到新的块则初始化莫队区间
    //初始化莫队左结点为当前询问区间的左边所属块的右节点+1
    if(get(l)!=last_block){
        while(right>get_right(l)) --cnt[a[right--]];
        while(left<get_right(l)+1) --cnt[a[left++]];
        last_block=get(l);
        res = 0;
    }
    //扩展右端点
```

```
    while(right<r) add(a[++right], res);
    backup = res;
    __l=left;
    //暴力扩展左端点，统计答案后回滚
    while(__l>l) --__l,add(a[__l], res);
    ans[id] = res;
    res=backup;

    while(__l<left) --cnt[a[__l]],__l++; //回滚
}
```

## 板子*带修莫队*

带修莫队

墨墨购买了一套 $N$ 支彩色画笔（其中有些颜色可能相同），摆成一排，你需要回答墨墨的提问。

墨墨会像你发布如下指令：

1. `Q L R` 代表询问你从第 $L$ 支画笔到第 $R$ 支画笔中共有几种不同颜色的画笔。
2. `R P Col` 把第 $P$ 支画笔替换为颜色 $Col$。

为了满足墨墨的要求，你知道你需要干什么了吗？

```
// #define int LL
const int N=10010,M=1000010,mod=1e9+7;
int n,k,w[N],cnt[M],len,ans[N];
struct Query{
    int id,l,r,t;
}ques[N];int q=0;
struct Modify{
    int x,y;
}modi[N];int m=0;

int get(int x){
    return x/len;
}

void add(int x,int &res){
    if(!cnt[x]) res++;
    cnt[x]++;
}

void del(int x,int &res){
    cnt[x]--;
    if(!cnt[x]) res--;
}

void solve(){
    n=read(),k=read();
    rep(i,1,n) w[i]=read();
    rep(i,1,k) {
        char op[2];int l,r;
        scanf("%s%d%d",op,&l,&r);
        if(*op=='Q'){
            ques[++q]={q,l,r,m};
        }
        else{
```

```
            modi[++m]={l,r};
        }
    }

    len=cbrt((double)n*max(1,m)+1);

    sort(ques+1,ques+1+q,[](Query& A,Query& B){
        if(get(A.l)!=get(B.l)) return get(A.l)<get(B.l);
        else if(get(A.r)!=get(B.r)) return get(A.r)<get(B.r);
        return A.t<B.t;
    });

    for(int k=1,i=0,j=1,t=0,res=0;k<=q;++k){
        int id=ques[k].id, l=ques[k].l,r=ques[k].r,tn=ques[k].t;
        while(i<r) add(w[++i], res);
        while(i>r) del(w[i--], res);
        while(j<l) del(w[j++], res);
        while(j>l) add(w[--j], res);
        while(t<tn){
            t++;
            if(modi[t].x>=j&&modi[t].x<=i){
                del(w[modi[t].x], res);
                add(modi[t].y, res);
            }
            swap(w[modi[t].x], modi[t].y);
        }
        while(t>tn){
            if(modi[t].x>=j&&modi[t].x<=i){
                del(w[modi[t].x], res);
                add(modi[t].y, res);
            }
            swap(w[modi[t].x], modi[t].y);
            t--;
        }
        ans[id]=res;
    }
    rep(i,1,q) printf("%d\n",ans[i]);
}
```

## 根号分治

三种类型：

1.普通的分块：区间加减

2.图上分块：按照点的度数进行分块

3.$x_1+x_2+\ldots+x_n=k$中最多有$\sqrt{k}$个不同的取值

## 整体二分

核心函数：

```
void solve(int vl,int vr,int ql,int qr);//在值域[vl,vr]上二分处理[ql,qr]这些操作。
```

以区间第k大数为例，我们可以对每一个询问做二分，复杂度是$nq\sqrt{\text{值域}}$的，但是显然会超时，因此我们对二分进行优化，每次同时进行多次二分。

对于二分的值mid，我们按照下标记录到树状数组中，以你对于询问[l,r]我们可以直接在树状数组中直接查到

```
typedef pair<int,int> pii;
#define N 200010
#define M 400010
#define VL -1e9
#define VR 1e9
int n,m;
struct Data{
    int op; //op=1 (insert) op=2 (query)
    int x,y,k,id;
    Data(int _op=0,int _x=0,int _y=0,int _k=0,int _id=0):op(_op),x(_x),y(_y),k(_k),id(_id){};
}q[M],lq[M],rq[M];
int ans[N];
struct BIT{
    #define lowbit(x) ((x)&(-x))
    int tr[N];
    inline void add(int x,int d){for(;x<=N-10;x+=lowbit(x)) tr[x]+=d;}
    inline int ask(int x){int ans=0;for(;x;x-=lowbit(x)) ans+=tr[x];return ans;}
}T;

void solve(int vl,int vr,int ql,int qr){
    if(vl>vr||ql>qr) return ;
    if(vl == vr){
        for(int i=ql;i<=qr;++i)
            if(q[i].op==2) ans[q[i].id]=vl;
        return ;
    }
    int mid=vl+vr>>1;
    int nl=0,nr=0;
    for(int i=ql;i<=qr;++i){
        if(q[i].op==1){ //如果是一个修改，
            if(q[i].x<=mid){
                T.add(q[i].y, 1);
                lq[++nl]=q[i];
            }
            else rq[++nr]=q[i];
        }
        else{
            int cnt=T.ask(q[i].y)-T.ask(q[i].x-1);
            if(cnt>=q[i].k) lq[++nl]=q[i];
            else {
                q[i].k-=cnt;
                rq[++nr]=q[i];
            }
        }
    }
    //clean BIT
    for(int i=ql;i<=qr;++i){
        if(q[i].op==1)
            if(q[i].x<=mid) T.add(q[i].y, -1);
    }
    //conquer
    for(int i=1;i<=nl;++i) q[ql+i-1]=lq[i];
    for(int i=1;i<=nr;++i) q[ql+nl+i-1]=rq[i];

    //solve
    solve(vl, mid, ql, ql+nl-1);
```

```
    solve(mid+1, vr, ql+nl, qr);
}

void solve(){
    n=read(),m=read();
    rep(i,1,n){
        int x=read();
        q[i]=Data(1,x,i);
    }
    rep(i,1,m){
        int l=read(),r=read(),x=read();
        q[n+i]=Data(2,l,r,x,i);
    }
    solve(VL,VR,1,n+m);
    rep(i,1,m){
        print(ans[i]);
    }
}
```

# *字典树*

## 常规字典树

```
//trie每个点的所有儿子son[],cnt[]存储以cnt点结尾的点的数量，idx与单链表类似，表示当前用到的点
int son[MAXN][26],cnt[MAXN],idx=0;//下标是0的点即是根节点又是空节点
char str[MAXN],op;int n;
//插入
void insert(char str[]){
    int p=0;
    for(int i=0;str[i];i++){
        int u=str[i]-'a';
        if(!son[p][u]) son[p][u]=++idx;
        p=son[p][u];
    }
    cnt[p]++;
}
//查询
int query(char str[]){
    int p=0;
    for(int i=0;str[i];i++){
        int u=str[i]-'a';
        if(!son[p][u]) return 0;
        p=son[p][u];
    }
    return cnt[p];
}
```

## 可持久化字典树

```
const int N = 600010,M=N*25;
int n,m,k,a[N],idx=0,max_id[M];
int tr[M][2],root[N], cnt[M];

void insert(int i,int k,int p,int q){
    if(k<0){
        max_id[q]=i;
        return ;
    }
    int u=a[i]>>k&1;
    if(p) tr[q][u^1]=tr[p][u^1];
    tr[q][u]=++idx;
    insert(i, k-1, tr[p][u], tr[q][u]);
    max_id[q]=max(max_id[tr[q][0]], max_id[tr[q][1]]);
}

int query(int u,int x,int L){
    int p=root[u];
    for(int i=23;i>=0;--i){
        int v=x>>i&1;
        if(max_id[tr[p][v^1]]>=L) p=tr[p][v^1];
        else p=tr[p][v];
    }
    return x^a[max_id[p]];
}
//=========================================

struct Trie{
    int root[N],tr[M][2],Cnt[M],idx=0,cnt;
    void insert(int k, int pre, int t, int x)
    {
        if (t < 0)return;
        signed int i = (x >> t) & 1;
        tr[k][!i] = tr[pre][!i];
        tr[k][i] = ++cnt;
        Cnt[tr[k][i]] = Cnt[tr[pre][i]] + 1;
        insert(tr[k][i], tr[pre][i], t - 1, x);
    }
    int query(int l, int r, int t, int x)
    {
        if (t < 0)return 0;
        signed int i = (x >> t) & 1;
        if (Cnt[tr[r][!i]] > Cnt[tr[l][!i]])
            return (1 << t) | query(tr[l][!i], tr[r][!i], t - 1, x);
        else return query(tr[l][i], tr[r][i], t - 1, x);
    }
}T;


struct Trie{
    int tr[M][K],cnt[M],idx,rt[N];
    void insert(int nw,int pre,int x){
        for(int i=23;~i;--i){
            int v=x>>i&1; tr[nw][v^1]=tr[pre][v^1];
            tr[nw][v]=++idx,nw=tr[nw][v],pre=tr[pre][v],cnt[nw]=cnt[pre]+1;
```

```
        }
    }
    int query(int nw,int pre,int x){
        int ans=0;
        for(int i=23;~i;--i){
            int v=x>>i&1;
            if(cnt[tr[nw][v^1]]-cnt[tr[pre][v^1]]) nw=tr[nw][v^1],pre=tr[pre][v^1],ans|=1<<i;
            else nw=tr[nw][v],pre=tr[pre][v];
        }return ans;
    }
}tr;
```

## 反建字典树&字典树合并

主要是方便完成所有数+1的操作

- 例题

大意是树上每个点有一个权值，需要你维护三种操作：
1 x : 给距离x为1的所有的点+1
2 x y: x点值-y
3 x:查询所有距离x点=1的所有值的异或和
一道非常经典的题目，用到了两个非常经典的套路：
1.从低到高建立01Trie来维护每个点的+1操作和所有点的异或和
2.分别单独维护儿子和父亲

```
const int N=600010,M=N*19,mod=1e9+7;
int n,m,k,e[N*2],ne[N*2],h[N],idx,fa[N];
void add(int a,int b){e[idx]=b,ne[idx]=h[a],h[a]=idx++;}
#define MAX_H 17
int root[N];
int w[M],tr[M][2],xorv[M],cnt,val[N]; //trie维护的是某个节点的子树
int ad[N]; //维护x对x周围节点的增加量
//w[u]维护的是u->fa这条边的个数,xorv[u]维护以u为根的子树的异或和
void push_up(int u){
    w[u]=xorv[u]=0;
    if(tr[u][0]){
        w[u] += w[tr[u][0]];
        xorv[u] ^= xorv[tr[u][0]]<<1;
    }
    if(tr[u][1]){
        w[u] += w[tr[u][1]];
        xorv[u] ^= (xorv[tr[u][1]]<<1) | (w[tr[u][1]]&1);
    }
}

void insert(int &u,int x,int bit){
    if(!u) {
        u=++cnt,xorv[u]=0,w[u]=0;
    }
    if(bit>MAX_H) {
        w[u] ++;
        return ;
    }
    insert(tr[u][x&1], x>>1, bit+1);
    push_up(u);
}
```

```
void erase(int u,int x,int bit){
    if(bit>MAX_H) {
        w[u] --;
        return ;
    }
    erase(tr[u][x&1], x>>1, bit+1);
    push_up(u);
}
//trie维护+1的逻辑就是:找到第一个0把它变成1然后把后面所有的1都变成0
void add_one(int u){
    swap(tr[u][0], tr[u][1]);
    if(tr[u][0]) add_one(tr[u][0]);
    push_up(u);
}

int merge(int p,int q){
    if(!p||!q) return p+q;
    w[p] = w[p]+w[q];
    xorv[p] ^= xorv[q];
    tr[p][0]=merge(tr[p][0], tr[q][0]);
    tr[p][1]=merge(tr[p][1], tr[q][1]);
    return p;
}

void dfs(int u,int pre){
    fa[u]=pre;
    for(int i=h[u];~i;i=ne[i]){
        int j=e[i];
        if(j==pre) continue;
        dfs(j, u);
    }
}

void solve(){
    n=read(),m=read();
    rep(i,1,n-1){
        int u=read(),v=read();
        add(u,v),add(v,u);
    }
    dfs(1,0);
    rep(i,1,n){
        val[i]=read();
        if(i!=1) {
            insert(root[fa[i]], val[i], 0);
        }
    }
    while(m--){
        int ty=read();
        if(ty==1){
            int x=read();
            ad[x] ++;
            add_one(root[x]);
            if(x!=1){
                if(fa[x])
                    erase(root[fa[fa[x]]], val[fa[x]]+ad[fa[fa[x]]], 0);
                val[fa[x]] ++;
                if(fa[x])
```

```
                    insert(root[fa[fa[x]]], val[fa[x]]+ad[fa[fa[x]]], 0);
            }
        }
        else if(ty==2){
            int x=read(),y=read();
            if(fa[x])
                erase(root[fa[x]], val[x]+ad[fa[x]], 0);
            val[x] -= y;
            if(fa[x])
                insert(root[fa[x]], val[x]+ad[fa[x]], 0);
        }
        else{
            int x=read();
            if(x!=1){
                if(fa[x]&&fa[fa[x]]){
                    print(xorv[root[x]]^(val[fa[x]]+ad[fa[fa[x]]]));
                }
                else{
                    print(xorv[root[x]]^val[fa[x]]);
                }
            }
            else{
                print(xorv[root[x]]);
            }
        }
    }
}
```

# *平衡树*

平衡树的用途：在只考虑前四种用途的情况下，我们可以借助STL中的set容器来代替平衡树（实际上set就是一颗红黑树）

1.插入：递归找到位置并插入 ___STL set insert()

2.删除：将删除的位置变成叶子节点之后删除 __STL erase

3.找前驱/后继：中序遍历中的前一个位置和后一个位置STL_ set ++

```
//找前驱伪代码
if(p->left):
    p=p->left
    while(p->right) p=p->right
    return p;
else:
    while(p->father>p) p=p->father
    return p;
```

4.找最大值和最小值：**STL_set_/begin()_**/end()-1

5.求某一个值得排名

6.求排名是k的数是哪个

7.比某个数小的最大值

8.比某个数大的最小值

- Splay

```
struct Splay{
    int root,idx;

    struct Node{
       int s[2];
       int sz,p,v;
       bool flag;

       void init(int _v=0,int _p=0){
           p=_p,v=_v;
           sz=1;
       }
    }tr[100010];

    Splay(){
      root=idx=0;
    }

    void push_up(int u){
       tr[u].sz=tr[tr[u].s[0]].sz+tr[tr[u].s[1]].sz+1;
    }

    void push_down(int u){
        if(tr[u].flag){
           swap(tr[u].s[0],tr[u].s[1]);
           tr[tr[u].s[0]].flag^=1;
           tr[tr[u].s[1]].flag^=1;
           tr[u].flag=false;
        }
    }

   void rotate(int x){
        int y=tr[x].p,z=tr[y].p;
        int k=tr[y].s[1]==x;
        tr[z].s[tr[z].s[1]==y]=x, tr[x].p=z;
        tr[y].s[k]=tr[x].s[k^1], tr[tr[x].s[k^1]].p=y;
        tr[x].s[k^1]=y, tr[y].p=x;
        push_up(y); push_up(x);
   }

    void splay(int x,int k){
       while(tr[x].p!=k){
            int y=tr[x].p,z=tr[y].p;
            if(z!=k)
                (tr[y].s[1]==x)^(tr[z].s[1]==y)?rotate(x):rotate(y);
            rotate(x);
       }
       if(!k) root=x;
    }
```

```
    void insert(int v){
        int u=root,p=0;
        while(u) p=u,u=tr[u].s[v>tr[u].v];
        u=++idx;
        if(p) tr[p].s[v>tr[p].v]=u;
        tr[u].init(v,p);
        splay(u,0);
    }

    int find(int v){
        int u=root,res;
        while(u){
            if(tr[u].v>=v) res=u,u=tr[u].s[0];
            else u=tr[u].s[1];
        }
        return res;
    }

    int find_kth(int x){
        int u=root;
        while(true){
            push_down(u);
            if(tr[tr[u].s[0]].sz>=x) u=tr[u].s[0];
            else if(tr[tr[u].s[0]].sz+1==x) return u;
            else x-=tr[tr[u].s[0]].sz+1,u=tr[u].s[1];
        }
        return -1;
    }
}tree;
```

- Treap

```
struct Treap{
    const int N = 100010;
    const int INF=1e9;
    int idx=0,root=0;

    struct Node{
        int l,r;
        int key,val;
        int cnt,sz;
    }tr[100010];

    void push_up(int &u){
        tr[u].sz=tr[tr[u].l].sz+tr[tr[u].r].sz+tr[u].cnt;
    }

    int get_node(int key){
        int u=++idx;
        tr[u].key=key;
        tr[u].val=rand();
        tr[u].cnt=tr[u].sz=1;
        return u;
    }

    void zig(int &p){
        int q=tr[p].l;
```

```
    tr[p].l=tr[q].r;
    tr[q].r=p;
    p=q;
    push_up(tr[p].r);
    push_up(p);
}

void zag(int &p){
    int q=tr[p].r;
    tr[p].r=tr[q].l;
    tr[q].l=p;
    p=q;
    push_up(tr[p].l);
    push_up(p);
}

Treap(){  //初始化函数，初始化两个边界
    get_node(-INF); get_node(INF);
    root=1; tr[root].r=2;
    push_up(root);
    if(tr[1].val < tr[2].val) zag(root);
}

void insert(int &p,int key){
    if(!p) p=get_node(key);
    else if(tr[p].key==key) tr[p].cnt++;
    else if(tr[p].key>key){
        insert(tr[p].l,key);

        if(tr[tr[p].l].val>tr[p].val) zig(p);
    }
    else{
        insert(tr[p].r,key);

        if(tr[tr[p].r].val>tr[p].val) zag(p);
    }
    push_up(p);
}

void remove(int &p,int key){
    if(!p) return ;
    if(tr[p].key==key){
        if(tr[p].cnt>1) tr[p].cnt--;
        else if(tr[p].l||tr[p].r){
            if(!tr[p].r||tr[tr[p].l].val>tr[tr[p].r].val){
                zig(p);
                remove(tr[p].r, key);
            }
            else{
                zag(p);
                remove(tr[p].l, key);
            }
        }
        else p=0; //直接是叶子节点
    }
    else if(tr[p].key>key) remove(tr[p].l,key);
    else remove(tr[p].r,key);
    push_up(p);
```

```
    }

    int get_rank_by_key(int p,int key){
        if(!p) return 0;
        if(tr[p].key==key) return tr[tr[p].l].sz+1;
        else if(tr[p].key>key) return get_rank_by_key(tr[p].l,key);
        else return get_rank_by_key(tr[p].r,key)+tr[tr[p].l].sz+tr[p].cnt;
    }

    int get_key_by_rank(int p,int rk){
        if(!p) return INF;
        if(rk<=tr[tr[p].l].sz) return get_key_by_rank(tr[p].l,rk);
        else if(rk<=tr[tr[p].l].sz+tr[p].cnt) return tr[p].key;
        else return get_key_by_rank(tr[p].r,rk-tr[tr[p].l].sz-tr[p].cnt);
    }

    int get_pre(int p,int key){
        if(!p) return -INF;
        if(key<=tr[p].key) return get_pre(tr[p].l,key);
        else return max(tr[p].key,get_pre(tr[p].r,key));
    }

    int get_nxt(int p,int key){
        if(!p) return INF;
        if(key<tr[p].key) return min(tr[p].key, get_nxt(tr[p].l, key));
        else return get_nxt(tr[p].r,key);
    }
}treap;
```

**---------分割线---------不太常用数据结构**

# *左偏树*

```
struct Leftist_Tree{
    int v[200010],l[200010],r[200010],dist[200010];
    bool cmp(int x,int y){return v[x]!=v[y]?v[x]<v[y]:x<y;}
    void insert(int x,int d){v[x]=d; dist[x]=1;}
    int merge(int x,int y){
        if(!x||!y) return x+y;
        if(cmp(y,x)) swap(x,y);
        r[x]=merge(r[x],y);
        if(dist[r[x]]>dist[l[x]]) swap(r[x], l[x]);
        dist[x]=dist[r[x]]+1;
        return x;
    }
}LT;
```

## 应用：

左偏树的应用主要是：

1.快速找最小值和合并两棵树（略）

2.快速维护中位数

维护中位数常用左偏树+单调栈：

左偏树为维护一段数中较小的一半的大根堆，单调栈维护的是每一段数的信息，比如大根堆维护的前一半的最大值，即中位数。由于中位数在每次加进来一个数的情况下只改变一位。（注意当都是奇数的情况下需要加完后退一个，因为奇数+奇数所得的中位数）。

https://www.acwing.com/problem/content/2727/ 数字序列

给定一个整数序列 $a_1, a_2, \cdots, a_n$。

请你求出一个递增序列 $b_1 < b_2 < \cdots < b_n$，使得序列 $a_i$ 和 $b_i$ 的各项之差的绝对值之和 $|a_1 - b_1| + |a_2 - b_2| + \cdots + |a_n - b_n|$ 最小。

有一个非常妙的贪心就是顺序扫描a[i]，每次扫到一个数与之前的最优清空做比较，如果是小的话就都取中位数。用左偏树维护一段区间内的合并并找中位数。

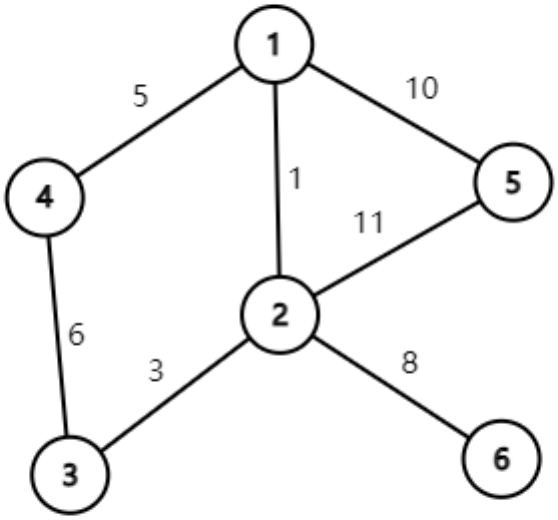
```
const int N=1000010,M=N*2,mod=1e9+7;
int n,m,top,ans[N],a[N];
struct Seg{
    int ed,rt,sz;
}stk[N];
int l[N],r[N],v[N],dist[N],idx;

int merge(int x,int y){
    if(!x||!y) return x+y;
    if(v[y]>v[x]) swap(x,y);
    r[x]=merge(r[x],y);
    if(dist[r[x]]>dist[l[x]]) swap(l[x], r[x]);
    dist[x]=dist[r[x]]+1;
    return x;
}
signed main(){
    n=read();
    rep(i,1,n ) v[i]=read()-i;
    for(int i=1;i<=n;++i){
        Seg nw={i,i,1};
        dist[i]=1;
        while(top&&v[nw.rt]<v[stk[top].rt]){
            nw.rt=merge(nw.rt, stk[top].rt);
            if(nw.sz%2 && stk[top].sz%2)
                nw.rt=merge(l[nw.rt], r[nw.rt]);
            nw.sz+=stk[top].sz,top--;
        }
        stk[++top]=nw;
    }
    LL res=0;
```

```
    for(int i=1,j=1;i<=top;++i)
        for(;j<=stk[i].ed;++j)
            ans[j]=v[stk[i].rt];
    rep(i,1,n) res+=abs(ans[i]-v[i]);
    print(res);
    for(int i=1;i<=n;++i) printf("%lld ",ans[i]+i);
    return 0;
}
```

# *克鲁斯卡尔重构树*

克鲁斯卡尔重构树是一颗根据克鲁斯卡尔算法的性质构造的一棵树。构造方式如下:

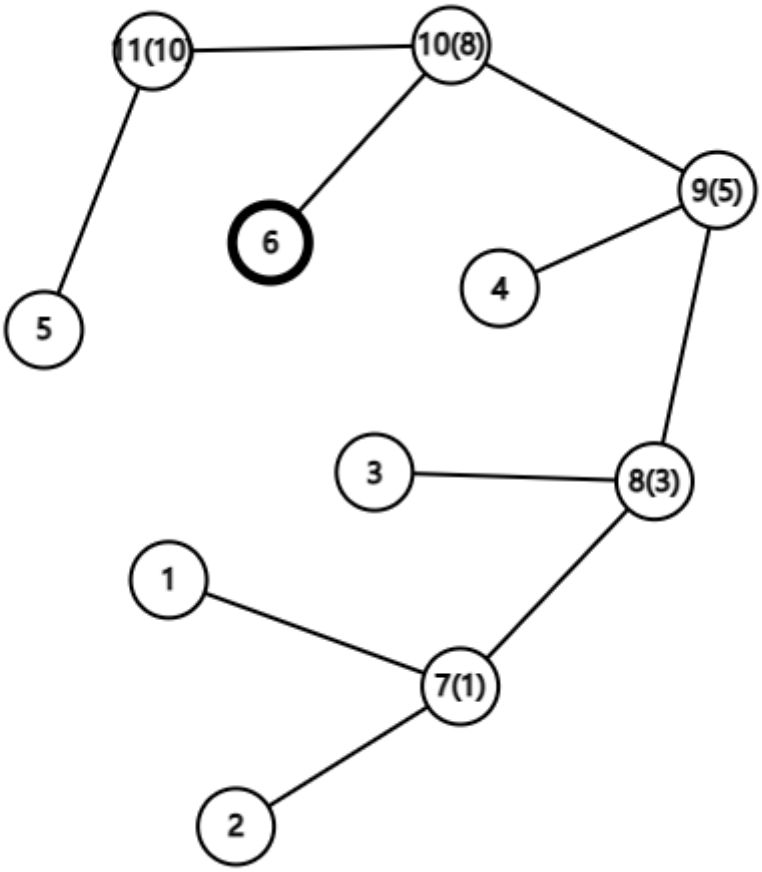
给定一个n个点的图




我们在求其最小生成树的过程中会用到并查集连接两个连通块。在连接两个连通块的时候，我们假设此时连接边权为$w$,两个连通块的根节点为$root_u$和$root_v$,我们新建一个节点$root_{new}$分别向$root_u$和$root_v$连(无向)边,并给$root_{new}$附上点权$w_i$,其中$w_i = w$，在这之后新节点$root_{new}$也作为整个连通块的根节点。

按照此方法直到在原图中构造出一棵最小生成树。

样例如图所示:




## 性质：

1.Kruskal重构树有助于求一个图中u到v节点的边权最大值最小路径上的边权最大值

> 最大值就是克鲁斯卡尔树中u和v点的lca对应的点权

2.同理于1，将边权按照从大到小排序之后求出来的LCA的点权就是u到v路径上最小值最大的值

3.从u出发只经过边权不超过k的边能到达的节点

> 由于克鲁斯卡尔树中一个点的祖先点权是升序排列的(以最后一个new点为根),因此我们通过倍增找到u的点权小于k的深度最小的祖先，其子树就是满足要求的节点的集合。

## 例题：

P4197 Peaks

> 在 Bytemountains 有 n 座山峰，每座山峰有他的高度$h_i$。有些山峰之间有双向道路相连，共 m$m$ 条路径，每条路径有一个困难值，这个值越大表示越难走。
>
> 现在有 q 组询问，每组询问询问从点 v 开始只经过困难值小于等于x的路径所能到达的山峰中第k高的山峰，如果无解输出 -1。

**做法**

> 1.按照边权排序建立克鲁斯卡尔重构树
>
> 2.构建dfs序和倍增预处理lca

3.建立动态开点的主席树，将所有原树中节点的高度值插入树中

4.对于每次查询先倍增查询到对应的根节点，然后在主席树上求解区间第K大

```
#define L 0
#define R 1000000000
const int N=200010,M=500010,mod=1e9+7;
int n,m,k,q,w[N],idx,h[N],sz[N];
vector<int> edge[N];
int f[26][N];
struct Edge{
    int a,b,w;
    bool operator<(const Edge&W)const{
        return w<W.w;
    }
}e[M];
int fa[N];
int find(int x){
    return x==fa[x]?fa[x]:fa[x]=find(fa[x]);
}
int id[N],val[N],cnt=0,l[N],r[N];

void dfs(int u,int pre){
    f[0][u]=pre;
    if(u<=n) sz[u]=1;
    l[u]=++cnt,id[cnt]=u,val[cnt]=h[u];
    for(auto v:edge[u]){
        if(v==pre) continue;
        dfs(v, u);
        sz[u] += sz[v];
    }
    r[u]=cnt;
}
struct HJT{
    struct Node{
        int ls,rs;
        int sum;
    }tr[N<<5];
    int root[N],idx=0;
    #define push_up(u) tr[u].sum=(tr[tr[u].ls].sum+tr[tr[u].rs].sum)
    void insert(int &u,int pre,int l,int r,int pos){
        u=++idx;
        tr[u]=tr[pre];
        if(l==r){
            tr[u].sum ++;
            return ;
        }
        int mid=l+r>>1;
        if(pos<=mid) insert(tr[u].ls, tr[pre].ls, l, mid, pos);
        else insert(tr[u].rs, tr[pre].rs, mid+1, r, pos);
        push_up(u);
    }
    int query(int now,int pre,int l,int r,int k){
        if(l==r) return l;
        int mid=l+r>>1;
        int sum=tr[tr[now].rs].sum-tr[tr[pre].rs].sum;
        if(sum>=k) return query(tr[now].rs, tr[pre].rs, mid+1, r, k);
        else return query(tr[now].ls, tr[pre].ls, l, mid, k-sum);
```

```
    }
}T;

void solve(){
    n=read(),m=read(),q=read();
    idx=n;
    rep(i,1,2*n) fa[i]=i;
    rep(i,1,n)  h[i]=read();
    rep(i,1,m) e[i].a=read(),e[i].b=read(),e[i].w=read();
    sort(e+1,e+1+m);
    for(int i=1;i<=m;++i){
        int a=e[i].a,b=e[i].b;
        int pa=find(a),pb=find(b);
        if(pa!=pb){
            int node=++idx;
            w[node]=e[i].w;
            edge[pa].push_back(node); edge[node].push_back(pa);
            edge[pb].push_back(node); edge[node].push_back(pb);
            fa[pa]=fa[pb]=fa[node];
        }
    }
    dfs(idx,0);
    for(int i=1;i<=25;++i)
        for(int j=1;j<=idx;++j)
            f[i][j]=f[i-1][f[i-1][j]];

    for(int i=1;i<=cnt;++i)
        T.insert(T.root[i], T.root[i-1], L, R, val[i]);

    // w[0]=2e9;
    while(q--){
        int v=read(),x=read(),k=read();
        for(int i=25;i>=0;--i)
            if(f[i][v]&&w[f[i][v]]<=x)
                v=f[i][v];
        if(sz[v]<k){
            puts("-1");
            continue;
        }
        int pl=T.query(T.root[r[v]], T.root[l[v]-1], L, R, k);
        print(pl);
    }
}
```

## *珂朵莉树ODT*

可以较快地实现：(得保证数据随机！！！)

- 区间加
- 区间赋值
- 求区间第k大值
- 求区间n次方和

> 珂朵莉树的思想在于随机数据下的区间赋值操作很可能让大量元素变为同一个数。所以我们以三元组<l,r,v>的形式保存数据（区间 [l,r] 中的元素的值都是v）

存储形式：

```
struct Node{
    int l,r;
    mutable int c;
    node(int l,int r,int v):l(l),r(r),v(v){};
    bool operator<(const Node& W)const{return l<W.l;}
};
set<Node> ds;
```

1.split操作，区间分裂

```
//断开”的操作，把<l,r,v>断成<l,pos-1,v>和<pos,r,v>:
set<Node>::iterator split(int pos){
    auto now=ds.lower_bound(Node(pos,0,0));// 寻找左端点大于等于pos的第一个节点
    if(now!=ds.end()&&now->l == pos)
        return now;
    now --; //往前数一个节点
    int l=now->l,r=now->r,v=now->v;
    ds.erase(now);
    ds.insert(Node(l, pos-1, v));
    return ds.insert(Node(pos,r,v)).x; //nsert默认返回值是一个pair，第一个成员是以pos开头的那个节点的迭代器
}
```

2.assign操作，区间赋值

```
void assign(LL l,LL r,LL v){ //将[l,r]赋值为v
    auto end=split(r+1),begin=split(l);
    ds.erase(begin, end);
    ds.insert(Node(l,r,v));
}
```

例题：

> 从现在到学期结束还有 n 天(从 1 到 n 编号)，他们一开始都是工作日。接下来学校的工作人员会**依次**发出 q 个指令，每个指令可以用三个参数 l,r,k 描述：
> 如果 k=1 ，那么从 l 到 r （包含端点）的所有日子都变成**非**工作日。
> 如果 k=2 ，那么从 l 到 r （包含端点）的所有日子都变成**工作日**。

将assign()修改一下，暴力统计即可

```
void assign(int l,int r,int v){
    int tot = 0;
    auto end=split(r+1),begin=split(l),now=begin;
    for(;now!=end;now++){
        tot += (now->v)*(now->r-now->l+1);
    }
    ds.erase(begin, end);
    ds.insert(Node(l,r,v));
    sum -= tot;
    sum += (r - l + 1) * v;
}
```

3.add()区间加

直接暴力加法即可

```
void add(int l,int r,int v){
    auto begin=split(l),end=split(r+1);
    for(;begin!=end;begin++)
        begin->v += v;
}
```

4.rank()区间第k小

```
int rank(int l,int r,int k){
    vector<pair<int,int>> tmp;
    auto begin=split(l),end=split(r+1);
    for(;begin!=end;begin++)
        tmp.push_back({begin->v, begin->r-begin->l+1});
    sort(tmp.begin(), tmp.end());
    for(auto u:tmp){
        k -= u.y;
        if(k<=0) return u.x;
    }
    return -1;
}
```

5.sum_of_pow()区间n次方的和

```
int sum_of_pow(int l,int r,int x,int y){
    int tot=0;
    auto begin=split(l),end=split(r+1);
    for(;begin!=end;begin++){
        tot = (tot + fpower(begin->v, x, y) * (begin->r-begin->l+1) % y)%y;
    }
    return tot;
}
```

ODT经典题：

- 区间加
- 区间赋值
- 求区间第k大值

- 求区间n次方和

```
#define int LL
const int N=200010,M=N*2,mod=1e9+7;
struct Node{
    int l,r;
    mutable int v;
    Node(int _l,int _r,int _v):l(_l),r(_r),v(_v){};
    bool operator<(const Node &W)const{return l<W.l;}
};
set<Node> ds;
int n,m,seed,vmx;
int sum=n;

set<Node>::iterator split(int pos){
    auto now=ds.lower_bound(Node(pos,0,0));
    if(now!=ds.end() && now->l == pos)
        return now;
    now --;
    int l=now->l,r=now->r,v=now->v;;
    ds.erase(now);
    ds.insert(Node(l,pos-1,v));
    return ds.insert(Node(pos, r, v)).x;
}

void assign(int l,int r,int v){
    int tot = 0;
    auto end=split(r+1),begin=split(l);
    ds.erase(begin, end);
    ds.insert(Node(l,r,v));
}

void add(int l,int r,int v){
    auto begin=split(l),end=split(r+1);
    for(;begin!=end;begin++)
        begin->v += v;
}

int kth(int l,int r,int k){
    vector<pair<int,int>> tmp;
    auto begin=split(l),end=split(r+1);
    for(;begin!=end;begin++)
        tmp.push_back({begin->v, begin->r-begin->l+1});
    sort(tmp.begin(), tmp.end());
    for(auto u:tmp){
        k -= u.y;
        if(k<=0) return u.x;
    }
    return -1;
}

int sum_of_pow(int l,int r,int x,int y){
    int tot=0;
    auto begin=split(l),end=split(r+1);
    for(;begin!=end;begin++){
        tot = (tot + fpower(begin->v, x, y) * (begin->r-begin->l+1) % y)%y;
    }
    return tot;
```

```
}
LL rnd()
{
    LL ret = seed;
    seed = (seed * 7 + 13) % 1000000007;
    return ret;
}
void solve(){
    n=read(),m=read(),seed=read(),vmx=read();
    rep(i,1,n){
        int r=rnd();
        ds.insert(Node(i, i, r%vmx+1));
    }
    while(m--){
        //input
        LL ty=rnd()%4+1,l=rnd()%n+1,r=rnd()%n+1,x,y;
        if(l>r) swap(l,r);
        if(ty==3) x=rnd()%(r-l+1)+1;
        else x=rnd()%vmx+1;
        if(ty==4) y=rnd()%vmx+1;

        if(ty==1){
            add(l,r,x);
        }
        else if(ty==2){
            assign(l,r,x);
        }
        else if(ty==3){
            print(kth(l, r, x));
        }
        else{
            print(sum_of_pow(l ,r, x, y));
        }
    }
}
```

# *动态树__Link Cut Tree*

**解决的问题：**

动态的维护一个森林，可以添加一条边或者删除一条边。并维护树上路径的一些信息。

维护一棵树，支持如下操作：

- 修改两点间路径权值。
- 查询两点间路径权值和。
- 修改某点子树权值。
- 查询某点子树权值和。

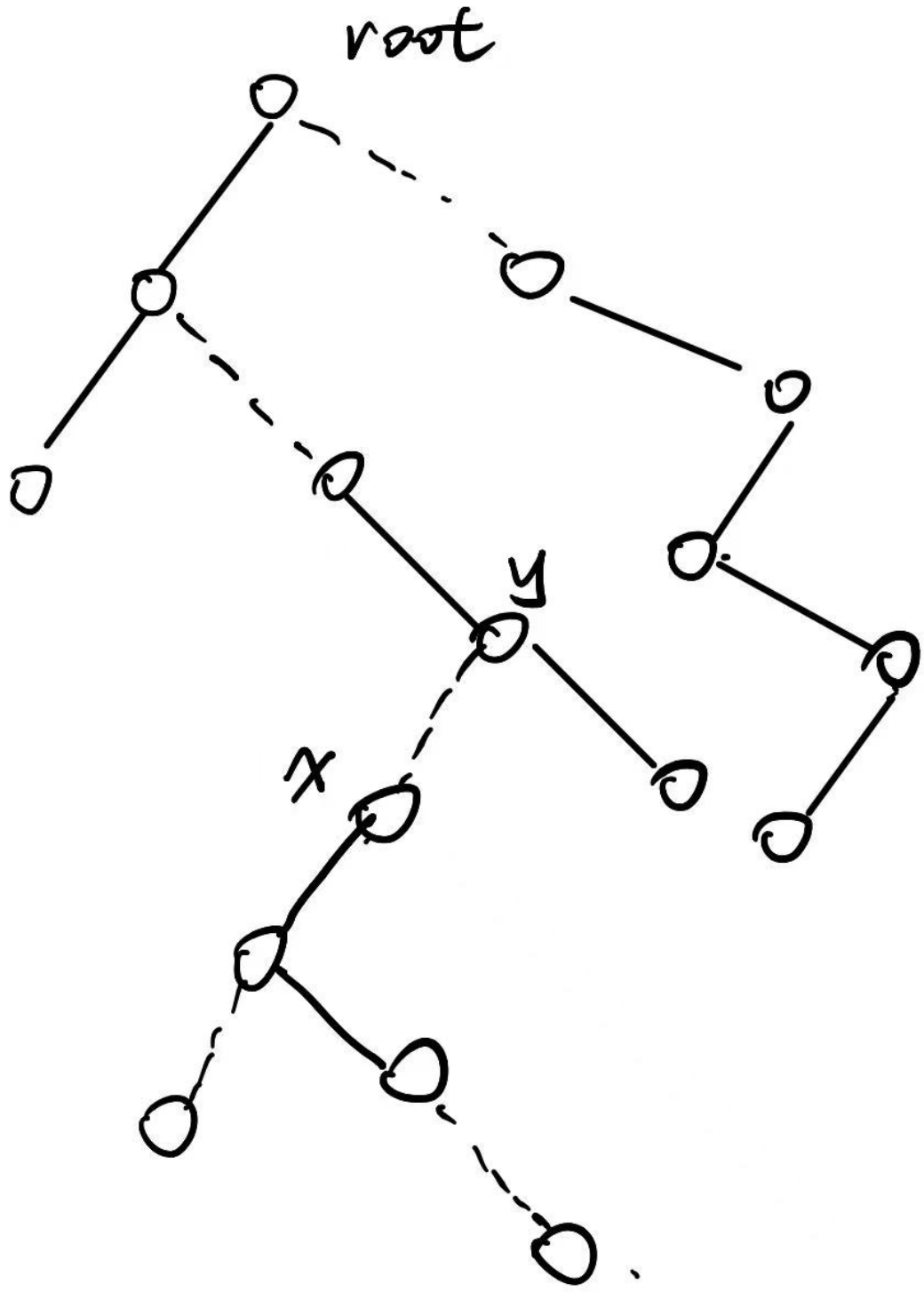
## 组成

所有边可以分成虚边和实边,每个节点最多只有一条实边，也可能没有实边。通过维护实边之间的关系来维护一棵树。

## 方法：

- 用splay维护所有实边路径：splay的中序遍历就是路径
  - 用splay的后继和前驱来维护原树的父子关系
  - 虚边用splay的根节点来维护

每一条实边路径可看作一棵树，每一棵树都是用splay来维护，每一个虚边通过对应的两个splay的节点来维护，即某个splay的某个点指向另外一个splay的根节点



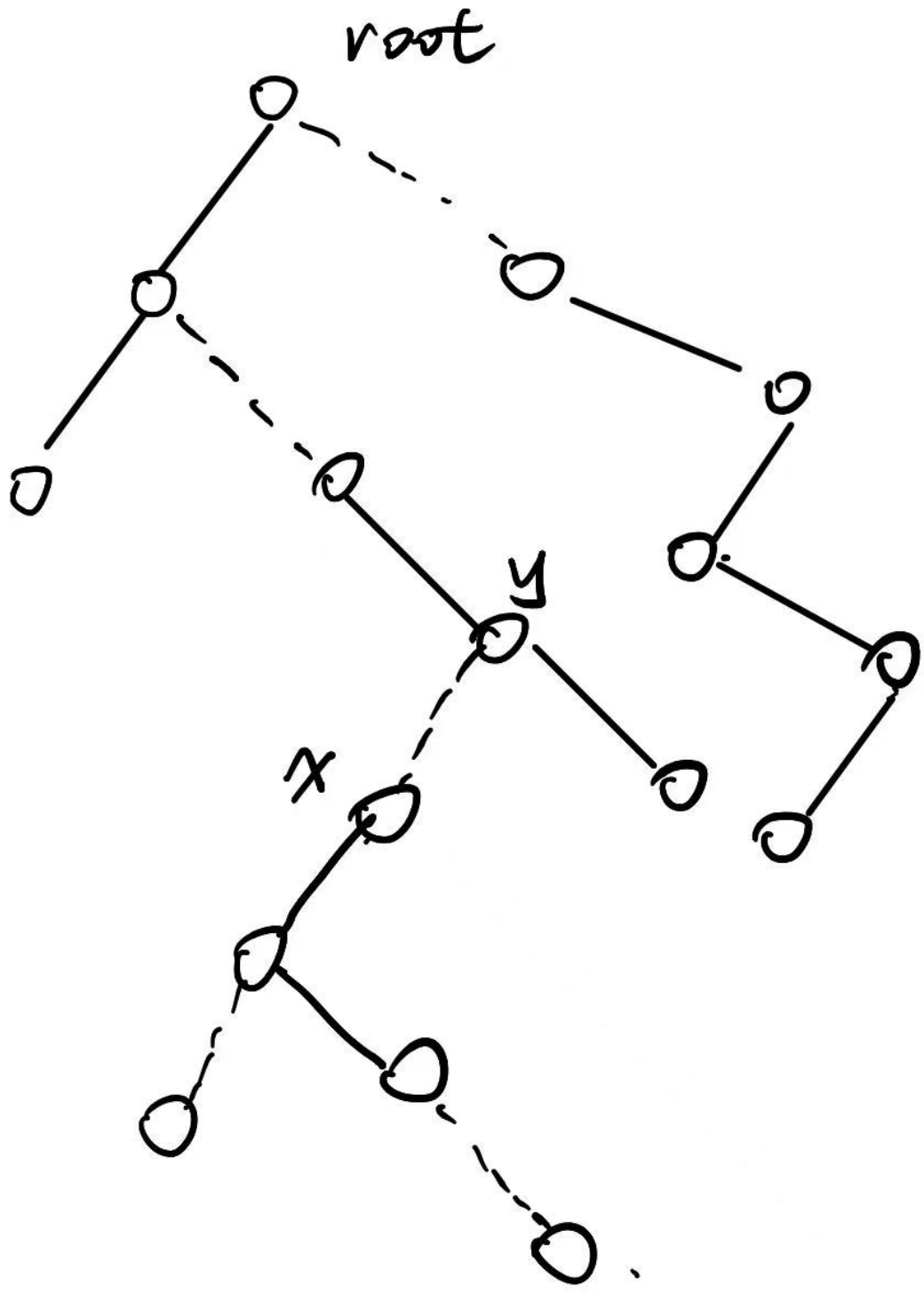

虚边：子节点知道父节点，父节点不知道子节点，即父节点未指向自己。

实边：父节点指向自己。

越在上面的点位于序列的前面，即splay的左子树

**操作**

1.access(x)：建立一条从根节点到x的实边路径




- splay(x,0)将x转到根节点
- splay(y,0)将y转到根节点
- 将以x为根的树接到y的右子树
- 把y直接插入z的右子树（修改后继）

2.make_root(int x)将x变成根节点

- 先建立从根节点到x的实边路径

- 将x转到根节点，然后翻转整条路经(借助于懒标记swap)

3.`find_root(int x)`找到x所在原树的根节点，再将原树的根节点转到splay的根节点

- 建立从根到x的路径
- 将x旋转到根节点
- 一直向左走到尽头便是根节点

> 副作用：调用access(x)会自动调用将x转到根节点

4.`split(int x,int y)`将从x到y的路径建成一棵splay(变成实边路径)

- `make_root(x)`
- `access(y)`

> 此时splay的根节点为

5.`link(x,y)`如果x,y不连通，则假如<x,y>

- `make_root(x)`
- `find_root(y)`

6.`cut(x,y)`如果x和y之间有边，则删除该边

7.`isroot(x)`判断x是不是所在splay的根节点

等价于x是其父节点的左儿子或者右儿子

```
const int N = 300010,M=N*2,mod=1e9+7;
struct Node{
    int s[2],p,v;
    int sum,rev;
}tr[N<<2];
int n,m,stk[N];

void rev(int u){
    swap(tr[u].s[0], tr[u].s[1]);
    tr[u].rev^=1;
}

void push_up(int u){
    tr[u].sum=tr[tr[u].s[0]].sum^tr[u].v^tr[tr[u].s[1]].sum;
}

void push_down(int u){
    if(tr[u].rev){
        rev(tr[u].s[0]); rev(tr[u].s[1]);
        tr[u].rev=0;
    }
}

bool isroot(int x) {
    return ((tr[tr[x].p].s[0]!=x) && (tr[tr[x].p].s[1]!=x));
}

void rotate(int x){
    int y=tr[x].p, z=tr[y].p;
    int k=tr[y].s[1]==x;
    if(!isroot(y)) tr[z].s[tr[z].s[1]==y] = x;
```

```
    tr[x].p=z;
    tr[y].s[k]=tr[x].s[k^1], tr[tr[x].s[k^1]].p=y;
    tr[x].s[k^1]=y,tr[y].p=x;
    push_up(y); push_up(x);
}

void splay(int x){
    int top=0,r=x;
    stk[++top]=r;
    while(!isroot(r)) r=tr[r].p, stk[++top]=r;
    while(top) push_down(stk[top--]);

    while(!isroot(x)){
        int y=tr[x].p, z=tr[y].p;
        if(!isroot(y))
            ((tr[z].s[1]==y)^(tr[y].s[1]==x))?rotate(x):rotate(y);
        rotate(x);
    }
}

void access(int x){ //建立一条从原树根节点到x的路径，同时将x变成所在辅助树的根节点
    int z=x;
    for(int y=0; x; y=x,x=tr[x].p){
        splay(x);
        tr[x].s[1]=y,push_up(x);
    }
    splay(z);
}

void makeroot(int x){ //将x变成原树的根
    access(x);
    rev(x);
}

int findroot(int x){ //找到x所在原树的根节点,再将原树的根节点转到所在辅助树的根节点
    access(x);
    while(tr[x].s[0]) push_down(x), x=tr[x].s[0];
    splay(x);
    return x;
}

void split(int x,int y){
    makeroot(x);
    access(y);
}

void link(int x,int y){
    makeroot(x);
    if(findroot(y)!=x) tr[x].p=y;
}

void cut(int x,int y){
    makeroot(x);
    if(findroot(y) == x && tr[y].p == x && !tr[y].s[0]) {
        tr[x].s[1]=tr[y].p=0;
        push_up(x);
    }
}
```

```
void solve(){
    n=read(); m=read();
    rep(i,1,n) tr[i].v=read();
    while(m--){
        int op=read(),x=read(),y=read();
        if(op==0){
            split(x,y);
            print(tr[y].sum);
        }
        else if(op==1){
            link(x,y);
        }
        else if(op==2){
            cut(x,y);
        }
        else{
            splay(x);
            tr[x].v=y;
            push_up(x);
        }
    }
}
```

## 笛卡尔树

```
#define N 200010
int stk[N],top,a[N],ls[N],rs[N],n;

void build(){
    top=0;
    rep(i,1,n+1) ls[i]=rs[i]=0;
    rep(i,1,n+1){
        int tmp=top;
        while(top&&a[stk[tmp]]>a[i]) --tmp;
        if(tmp) rs[stk[tmp]] = i;
        if(tmp<top) ls[i]=stk[tmp+1];
        stk[++tmp]=i;
        top=tmp;
    }
    /* 建树 */
    // rep(i,1,n+1){
    //  if(l[i]) add(i, l[i]);
    //  if(r[i]) add(i, r[i]);
    // }
}
```