Data Structures 数据结构

树状数组

树状数组是最为小巧实用的数据结构之一，能在 $\mathcal{O}(\log n)$ 的时间复杂度内进行单点修改+区间查询。通过维护差分数组也可以实现区间修改+单点查询。

树状数组通过前缀和相减来完成区间操作，所以要求维护的信息具有可减性，否则无法使用树状数组维护。

树状数组

```
template<typename T=int,T init=T()> struct Fenwick {
    using F=function<void(T&,const T&)>;
    F add,sub;
    vector<T> tr;

    int lowbit(int x) { return x&(-x); }
    void resize(int n) { tr.resize(n+2,init); }

    void modify(int pos,T val) {
        if(++pos<=0) return;
        while(pos<tr.size()) add(tr[pos],val),pos+=lowbit(pos);
    }

    void reset(int pos) {
        if(++pos<=0) return;
        while(pos<tr.size()) tr[pos]=init,pos+=lowbit(pos);
    }

    T query(int pos) {
        if(++pos<0) return init;
        T res=init;
        while(pos) add(res,tr[pos]),pos-=lowbit(pos);
        return res;
    }

    T range_query(int l,int r) {
        T res=query(r);
        sub(res,query(l-1));
        return res;
    }

    explicit Fenwick(
        int n,
        F add=[](T &x,const T &y) { x+=y; },
        F sub=[](T &x,const T &y) { x-=y; })
        : add(add),sub(sub) {
        resize(n);
    }
};
```

二维树状数组

```
template<typename T=int,T init=T()> struct Fenwick2D {
    using F=function<void(T&,const T&)>;
    F add,sub;
    vector<vector<T>> tr;
```

```
    int lowbit(int x) { return x&(-x); }
    void resize(int r,int c) { tr.resize(r+2,vector<T>(c+2,init)); }

    void modify(int r,int c,T val) {
        if(++r<=0||++c<=0) return;
        for(int i=r;i<tr.size();i+=lowbit(i))
            for(int j=c;j<tr[i].size();j+=lowbit(j))
                add(tr[i][j],val);
    }

    void reset(int r,int c) {
        if(++r<=0||++c<=0) return;
        for(int i=r;i<tr.size();i+=lowbit(i))
            for(int j=c;j<tr[i].size();j+=lowbit(j))
                tr[i][j]=init;
    }

    T query(int r,int c) {
        if(++r<0||++c<0) return init;
        T res=init;
        for(int i=r;i;i-=lowbit(i))
            for(int j=c;j;j-=lowbit(j))
                add(res,tr[i][j]);
        return res;
    }

    T matrix_query(int r,int c,int x,int y) {
        T res=query(x,y);
        sub(res,query(x,c-1));
        sub(res,query(r-1,y));
        add(res,query(r-1,c-1));
        return res;
    }

    explicit Fenwick2D(
        int r,int c,
        F add=[](T &x,const T &y) { x+=y; },
        F sub=[](T &x,const T &y) { x-=y; })
        : add(add),sub(sub) {
        resize(r,c);
    }
};
```

维护区间加区间和

修改 `l,r`

```
Fenwick<LL> dif(n),idif(n);

dif.modify(l, x);
```

```
dif.modify(r+1, -x);
idif.modify(l, 1LL*l*x);
idif.modify(r+1, -1LL*(r+1)*x);
```

查询 `l,r`

```
auto get=[&](int x) {
    return dif.query(x)*(x+1)-idif.query(x);
};
cout<<get(r)-get(l-1)<<endl;
```

维护矩阵加矩阵和

修改 `a,b,c,d`

```
Fenwick2D<LL> dif(n,m),idif(n,m),jdif(n,m),ijdif(n,m);

auto modify=[&](int r,int c,LL x) {
    dif.modify(r, c, x);
    idif.modify(r, c, x*r);
    jdif.modify(r, c, x*c);
    ijdif.modify(r, c, x*r*c);
};

modify(a, b, x);
modify(a, d+1, -x);
modify(c+1, b, -x);
modify(c+1, d+1, x);
```

查询 `a,b,c,d`

```
auto get=[&](int r,int c) {
    return
        (1LL*r*c+r+c+1)*dif.query(r, c)-
        (r+1)*jdif.query(r, c)-
        (c+1)*idif.query(r, c)+
        ijdif.query(r, c);
};

cout<<get(c, d)-get(c, b-1)-get(a-1, d)+get(a-1, b-1)<<endl;
```

树状数组上二分

类似线段树，我们可以在树上数组上进行二分，从高位向低位枚举即可。

权值树状数组求第k大的例子：

```
int kth(int k) {
    int pos=0;
    for(int i=bit;~i;i--)
        if(pos+(1<<i)<N&&tr[pos+(1<<i)]<k)
            pos+=1<<i,k-=tr[pos];
    return pos+1;
}
```

线段树

线段树能够灵活地维护区间信息，区间修改与查询均为 $\mathcal{O}(\log n)$，常数较大。

线段树

```
template<class Info,class Tag,int size> struct SegmentTree {
    #define lch ((u)<<1)
    #define rch ((u)<<1|1)

    int rng_l,rng_r;
    constexpr static int node_size=1<<__lg(size)<<2|1;
    array<Tag, node_size> tag;
    array<Info, node_size> info;
    array<bool, node_size> clean;

    void pushup(int u) {
        info[u]=info[lch]+info[rch];
    }

    void update(int u, const Tag &t) {
        info[u]+=t;
        tag[u]+=t;
        clean[u]=0;
    }

    void pushdn(int u) {
        if(clean[u]) return;
        update(lch, tag[u]);
        update(rch, tag[u]);
        clean[u]=1;
        tag[u].clear();
    }

    Info query(int u,int l,int r,int x,int y) {
        if(l>y||r<x) return {};
        if(l>=x&&r<=y) return info[u];
        pushdn(u);
        int mid=(l+r)/2;
        return query(lch,l,mid,x,y)+query(rch,mid+1,r,x,y);
    }
```

```
    Info query(int l,int r) { return query(1,rng_l,rng_r,l,r); }

    void modify(int u,int l,int r,int x,int y,const Tag &t) {
        if(l>y||r<x) return;
        if(l>=x&&r<=y) update(u, t);
        else {
            pushdn(u);
            int mid=(l+r)/2;
            if(mid>=x) modify(lch,l,mid,x,y,t);
            if(mid<y) modify(rch,mid+1,r,x,y,t);
            pushup(u);
        }
    }
    void modify(int l,int r,const Tag &t) { modify(1,rng_l,rng_r,l,r,t); }

    template<class F>
    int find_first(int u,int l,int r,int x,int y,F check) {
        if(l>y||r<x||l>=x&&r<=y&&!check(info[u])) return -1;
        if(l==r) return l;
        pushdn(u);
        int mid=(l+r)/2;
        int res=find_first(lch,l,mid,x,y,check);
        if(res==-1) res=find_first(rch,mid+1,r,x,y,check);
        return res;
    }
    template<class F> int find_first(int l,int r,F check) {
        return find_first(1,rng_l,rng_r,l,r,check);
    }

    template<class F>
    int find_last(int u,int l,int r,int x,int y,F check) {
        if(l>y||r<x||l>=x&&r<=y&&!check(info[u])) return -1;
        if(l==r) return l;
        pushdn(u);
        int mid=(l+r)/2;
        int res=find_last(rch,mid+1,r,x,y,check);
        if(res==-1) res=find_last(lch,l,mid,x,y,check);
        return res;
    }
    template<class F> int find_last(int l,int r,F check) {
        return find_last(1,rng_l,rng_r,l,r,check);
    }

    void build(int u,int l,int r) {
        clean[u]=1;
        info[u].init(l,r);
        tag[u].clear();
        if(l!=r) {
            int mid=(l+r)/2;
            build(lch,l,mid);
            build(rch,mid+1,r);
            pushup(u);
        }
    }
```

```
    void build(int l=1,int r=size) { build(1,rng_l=l,rng_r=r); }

    #undef lch
    #undef rch
};

struct Tag {

    void clear() {

    }

    Tag &operator+=(const Tag &t) {

        return *this;
    }
};

struct Info {

    void init(int l,int r) {
        if(l!=r) return;

    }

    friend Info operator+(const Info &l,const Info &r) {
        Info res;

        return res;
    }

    Info &operator+=(const Tag &t) {

        return *this;
    }
};

SegmentTree<Info, Tag, N> sgt;
```

单点修改线段树

仅支持单点修改、常数更小更简短的实现。

```
template<class Info,int size> struct SegmentTree {
    #define lch ((u)<<1)
    #define rch ((u)<<1|1)

    int rng_l,rng_r;
    constexpr static int node_size=1<<__lg(size)<<2|1;
    array<Info, node_size> info;
    array<int, size+1> leaf;
```

```
    void pushup(int u) {
        info[u]=info[lch]+info[rch];
    }

    Info query(int u,int l,int r,int x,int y) {
        if(l>y||r<x) return {};
        if(l>=x&&r<=y) return info[u];
        int mid=(l+r)/2;
        return query(lch,l,mid,x,y)+query(rch,mid+1,r,x,y);
    }
    Info query(int l,int r) { return query(1,rng_l,rng_r,l,r); }

    void modify(int p,const Info &v) {
        int u=leaf[p];
        info[u]+=v;
        while(u>>=1) pushup(u);
    }

    template<class F>
    int find_first(int u,int l,int r,int x,int y,F check) {
        if(l>y||r<x||l>=x&&r<=y&&!check(info[u])) return -1;
        if(l==r) return l;
        int mid=(l+r)/2;
        int res=find_first(lch,l,mid,x,y,check);
        if(res==-1) res=find_first(rch,mid+1,r,x,y,check);
        return res;
    }
    template<class F> int find_first(int l,int r,F check) {
        return find_first(1,rng_l,rng_r,l,r,check);
    }

    template<class F>
    int find_last(int u,int l,int r,int x,int y,F check) {
        if(l>y||r<x||l>=x&&r<=y&&!check(info[u])) return -1;
        if(l==r) return l;
        int mid=(l+r)/2;
        int res=find_last(rch,mid+1,r,x,y,check);
        if(res==-1) res=find_last(lch,l,mid,x,y,check);
        return res;
    }
    template<class F> int find_last(int l,int r,F check) {
        return find_last(1,rng_l,rng_r,l,r,check);
    }

    void build(int u,int l,int r) {
        info[u].init(l,r);
        if(l!=r) {
            int mid=(l+r)/2;
            build(lch,l,mid);
            build(rch,mid+1,r);
            pushup(u);
        }
        else leaf[l]=u;
```

```
    }
    void build(int l=1,int r=size) { build(1,rng_l=l,rng_r=r); }

    #undef lch
    #undef rch
};

struct Info {

    void init(int l,int r) {
        if(l!=r) return;

    }

    friend Info operator+(const Info &l,const Info &r) {
        Info res;

        return res;
    }

    Info &operator+=(const Info &v) {

        return *this;
    }
};

SegmentTree<Info, N> sgt;
```

线段树上二分（区间前缀）

```
template<class F>
int find_first(int u,int l,int r,int x,int y,F check,Info &suf) {
    if(l==r&&!check(info[u]+suf)) return -1;
    if(l>=x&&r<=y&&check(info[u]+suf)) return suf=info[u]+suf,l;
    pushdn(u);
    int mid=(l+r)/2;
    if(mid>=x&&mid<y) {
        int res=find_first(rch,mid+1,r,x,y,check,suf);
        if(res==mid+1) {
            int t=find_first(lch,l,mid,x,y,check,suf);
            if(t!=-1) res=t;
        }
        return res;
    }
    else if(mid>=x) return find_first(lch,l,mid,x,y,check,suf);
    return find_first(rch,mid+1,r,x,y,check,suf);
}
template<class F> int find_first(int l,int r,F check,Info suf={}) {
    l=max(l,rng_l),r=min(r,rng_r);
    return l>r?-1:find_first(1,rng_l,rng_r,l,r,check,suf);
}
```

```
template<class F>
int find_last(int u,int l,int r,int x,int y,F check,Info &pre) {
    if(l==r&&!check(pre+info[u])) return -1;
    if(l>=x&&r<=y&&check(pre+info[u])) return pre=pre+info[u],r;
    pushdn(u);
    int mid=(l+r)/2;
    if(mid>=x&&mid<y) {
        int res=find_last(lch,l,mid,x,y,check,pre);
        if(res==mid) {
            int t=find_last(rch,mid+1,r,x,y,check,pre);
            if(t!=-1) res=t;
        }
        return res;
    }
    else if(mid>=x) return find_last(lch,l,mid,x,y,check,pre);
    return find_last(rch,mid+1,r,x,y,check,pre);
}
template<class F> int find_last(int l,int r,F check,Info pre={}) {
    l=max(l,rng_l),r=min(r,rng_r);
    return l>r?-1:find_last(1,rng_l,rng_r,l,r,check,pre);
}
```

可持久化线段树

通过记录每次修改变化的节点，可以在保存历史信息的同时，大幅地压缩空间复杂度。

```
template<class Info,int node_size>
struct PersistentSegmentTree {
    int idx,rng_l,rng_r;
    vector<int> root;
    array<Info, node_size> info;
    array<int, node_size> lch,rch;

    int ver() {
        return root.size()-1;
    }

    int new_node() {
        assert(idx<node_size);
        return ++idx;
    }

    int new_root() {
        root.emplace_back();
        return ver();
    }

    void clone(int u,int v) {
        info[u]=info[v];
        lch[u]=lch[v];
```

```
        rch[u]=rch[v];
    }

    void pushup(int u) {
        info[u]=info[lch[u]]+info[rch[u]];
    }

    Info query(int u,int l,int r,int x,int y) {
        if(l>y||r<x) return {};
        if(l>=x&&r<=y) return info[u];
        int mid=(l+r)/2;
        return query(lch[u],l,mid,x,y)+query(rch[u],mid+1,r,x,y);
    }
    Info query(int u,int l,int r) {
        return query(root[u],rng_l,rng_r,l,r);
    }

    Info range_query(int u,int v,int l,int r,int x,int y) {
        if(l>y||r<x) return {};
        if(l>=x&&r<=y) return info[u]-info[v];
        int mid=(l+r)/2;
        return range_query(lch[u],lch[v],l,mid,x,y)+
               range_query(rch[u],rch[v],mid+1,r,x,y);
    }
    Info range_query(int u,int v,int l,int r) {
        return range_query(root[u],root[v],rng_l,rng_r,l,r);
    }

    void modify(int &u,int v,int l,int r,int p,const Info &val) {
        u=new_node();
        clone(u, v);
        if(l==r) info[u]+=val;
        else {
            int mid=(l+r)/2;
            if(p<=mid) modify(lch[u],lch[v],l,mid,p,val);
            else modify(rch[u],rch[v],mid+1,r,p,val);
            pushup(u);
        }
    }
    void modify(int u,int v,int p,const Info &val) {
        modify(root[u],root[v],rng_l,rng_r,p,val);
    }

    int update(int p,const Info &val) {
        new_root();
        modify(root[ver()],root[ver()-1],rng_l,rng_r,p,val);
        return ver();
    }

    template<class F>
    int find_first(int u,int l,int r,int x,int y,F check) {
        if(l>y||r<x||l>=x&&r<=y&&!check(info[u])) return -1;
        if(l==r) return l;
        int mid=(l+r)/2;
```

```
        int res=find_first(lch[u],l,mid,x,y,check);
        if(res==-1) res=find_first(rch[u],mid+1,r,x,y,check);
        return res;
    }
    template<class F> int find_first(int u,int l,int r,F check) {
        return find_first(root[u],rng_l,rng_r,l,r,check);
    }

    template<class F>
    int find_last(int u,int l,int r,int x,int y,F check) {
        if(l>y||r<x||l>=x&&r<=y&&!check(info[u])) return -1;
        if(l==r) return l;
        int mid=(l+r)/2;
        int res=find_last(rch[u],mid+1,r,x,y,check);
        if(res==-1) res=find_last(lch[u],l,mid,x,y,check);
        return res;
    }
    template<class F> int find_last(int u,int l,int r,F check) {
        return find_last(root[u],rng_l,rng_r,l,r,check);
    }

    void build(int &u,int l,int r) {
        u=new_node();
        info[u].init(l,r);
        if(l!=r) {
            int mid=(l+r)>>1;
            build(lch[u],l,mid);
            build(rch[u],mid+1,r);
            pushup(u);
        }
    }
    void build(int l,int r) {
        build(root[new_root()],rng_l=l,rng_r=r);
    }
};

struct Info {

    void init(int l,int r) {
        if(l!=r) return;

    }

    friend Info operator+(const Info &l,const Info &r) {
        Info res;

        return res;
    }

    friend Info operator-(const Info &l,const Info &r) {
        Info res;

        return res;
    }
```

```
    Info &operator+=(const Info &v) {

        return *this;
    }
};

PersistentSegmentTree<Info, N*__lg(N)*4> sgt;
```

势能线段树

```
struct Info {
    bool final;

    void init(int l,int r) {
        if(l!=r) return;

    }

    friend Info operator+(const Info &l,const Info &r) {
        Info res;

        return res;
    }

    void operator--(int) {

    }

    explicit operator bool() const { return final; }
};

template<class Info,int size> struct SegmentTree {
    #define lch (u<<1)
    #define rch (u<<1|1)

    struct Node {
        int l,r;
        Info info;
        void init(int l,int r) {
            this->l=l;
            this->r=r;
            info.init(l, r);
        }
    };

    array<Node, 1<<__lg(size)<<2|1> tr;

    void pushup(int u) {
        tr[u].info=tr[lch].info+tr[rch].info;
    }
```

```
    Info query(int u,int l,int r) {
        if(tr[u].l>=l&&tr[u].r<=r) { return tr[u].info; }
        else {
            int mid=(tr[u].l+tr[u].r)/2;
            if(mid>=l&&mid<r) return query(lch,l,r)+query(rch,l,r);
            else if(mid>=l) return query(lch,l,r);
            return query(rch,l,r);
        }
    }
    Info query(int l,int r) { return query(1,l,r); }

    void release(int u,int l,int r) {
        if(tr[u].info) return;
        else if(tr[u].l==tr[u].r) tr[u].info--;
        else {
            int mid=(tr[u].l+tr[u].r)/2;
            if(l<=mid) release(lch,l,r);
            if(r>mid) release(rch,l,r);
            pushup(u);
        }
    }
    void release(int l,int r) { release(1,l,r); }

    void build(int u,int l,int r) {
        tr[u].init(l,r);
        if(l!=r) {
            int mid=(l+r)/2;
            build(lch,l,mid);
            build(rch,mid+1,r);
            pushup(u);
        }
    }
    void build(int l=1,int r=size) { build(1,l,r); }

    #undef lch
    #undef rch
};
SegmentTree<Info, N> sgt;
```

线段树合并/分裂

```
namespace sgt {
    #define lch (tr[u].lc)
    #define rch (tr[u].rc)

    struct Node {
        int lc=0,rc=0;
        int cnt=0;
    };
    constexpr int M=N*(__lg(N)+1)*2;
```

```
    vector<Node> tr(M);
    int idx,rng_l,rng_r;

    int new_node() {
        assert(++idx<M);
        tr[idx]={};
        return idx;
    }

    void pushup(int u) {
        tr[u].cnt=tr[lch].cnt+tr[rch].cnt;
    }

    void update(int u) {
        if(!u) return;

    }

    void pushdn(int u) {

    }

    void merge(int &u,int v,int l,int r) {
        if(!u||!v) u=u|v;
        else if(l==r) {
            tr[u].cnt+=tr[v].cnt;
        }
        else {
            pushdn(u);
            int mid=(l+r)/2;
            merge(tr[u].lc, tr[v].lc, l, mid);
            merge(tr[u].rc, tr[v].rc, mid+1, r);
            pushup(u),pushup(v);
        }
    }
    void merge(int &u,int v) { merge(u,v,rng_l,rng_r); }

    pair<int,int> split(int u,int l,int r,int p) {
        if(r<p) return {u,0};
        if(l>=p) return {0,u};
        pushdn(u);
        int v=new_node();
        int mid=(l+r)/2;
        auto [a,b]=split(lch, l, mid, p);
        auto [c,d]=split(rch, mid+1, r, p);
        tr[u].lc=a,tr[u].rc=c;
        tr[v].lc=b,tr[v].rc=d;
        pushup(u),pushup(v);
        return {u,v};
    }
    pair<int,int> split(int u,int p) { return split(u,rng_l,rng_r,p); }

    int extract(int &u,int l,int r,int x,int y) {
        auto [a,b]=split(u, l, r, x);
```

```
        auto [c,d]=split(b, l, r, y+1);
        merge(a, d, l, r);
        return u=a,c;
    }
    int extract(int &u,int l,int r) { return extract(u,rng_l,rng_r,l,r); }

    int query(int u,int l,int r,int x,int y) {
        if(!u||l>y||r<x) return {};
        if(l>=x&&r<=y) {
            return tr[u].cnt;
        }
        pushdn(u);
        int mid=(l+r)/2;
        return query(lch, l, mid, x, y)+query(rch, mid+1, r, x, y);
    }
    int query(int u,int l,int r) { return query(u,rng_l,rng_r,l,r); }

    void modify(int &u,int l,int r,int p,int v) {
        if(!u) u=new_node();
        if(l==r) {
            tr[u].cnt+=v;
        }
        else {
            pushdn(u);
            int mid=(l+r)/2;
            if(p<=mid) modify(lch, l, mid, p, v);
            else modify(rch, mid+1, r, p, v);
            pushup(u);
        }
    }
    void modify(int &u,int p,int v) { modify(u,rng_l,rng_r,p,v); }

    void init(int l,int r) { idx=0,rng_l=l,rng_r=r; }

    #undef lch
    #undef rch
}
```

线段树合并除了合并和分裂外，剩下的和动态开点线段树保持一致，因此也需要考虑线段树共有的问题：

- 信息+信息
- 信息初始化与信息+空信息
- 信息+懒标记
- 懒标记+懒标记
- 懒标记清空

线段树合并更需要考虑是如何在合并/分裂时维护信息的变化，这部分的问题就比一般的线段树灵活很多。

合并子树需要考虑的：

- 是否需要合并叶节点，合并叶子时的信息维护
- 合并时是否需要新建节点来保留子树结构

合并需要保留子树结构最常见于树上线段树合并，此时需要每次都新建节点，需要的空间也会翻倍。

```
int merge(int x,int y) {
    if(!x||!y) return x|y;
    int u=new_node();
    lch=merge(tr[x].lc,tr[y].lc);
    rch=merge(tr[x].rc,tr[y].rc);
    tr[u].cnt=tr[lch].cnt+tr[rch].cnt;
    return u;
}
```

考虑合并的复杂度，由于一棵树中的一个节点至多被合并一次，即一个节点的贡献至多为1，所以无论以何种顺序合并，复杂度都等于总节点数，为 $\mathcal{O}(n\log n)$。分裂为合并的逆过程，复杂度和合并保持一致。

按X维分裂维护Y维信息的线段树

考虑现在有一颗权值线段树，现在要把线段树按照区间的 p 位置分裂。

为了简化问题，不妨设原数组是一个排列。

显然，我们是不能按照区间直接分裂一颗权值线段树的，因此我们需要在线段树上保存值在区间的最左位置 $minp$ 和最右位置 $maxp$。

那么分裂当前节点 u 时：

- 如果 $[minp, maxp]$ 仅在 p 的一侧，那么不需要分裂。
- 否则递归地分裂左右子树，假设左子树分裂为了 a, b 两棵树，右子树分裂为了 c, d 两棵树，u 分裂为 u, v，那么 a, c 归到 u，b, d 归到 v。

```
pair<int,int> split(int u,int k) {
    if(tr[u].minp>=k) return {0,u};
    if(tr[u].maxp<k) return {u,0};
    pushdn(u);
    int v=new_node();
    auto [a,b]=split(lch, k);
    auto [c,d]=split(rch, k);
    tr[u].lc=a,tr[u].rc=c;
    tr[v].lc=b,tr[v].rc=d;
    pushup(u),pushup(v);
    return {u,v};
}
```

同样的，我们可以按照权值分裂区间线段树。

如果原数组不是一个排列，那么和上述做法的区别是需要分裂叶子节点，而排列递归到叶子一定不需要分裂。

分裂叶子节点需要计算新的 $maxp, minp$，可以在 `std::set<pair<int,int>>` 上二分解决。不过这样总归还是比较麻烦，所以尽可能转化为排列来做。

复杂度与一般的分裂一致，为 $\mathcal{O}(n\log n)$。

线段树分治

线段树分治可以将 “增+删” 转化为 “增+撤销/持久化”，代价是多一个log的复杂度，并且要求问题可离线。

```
namespace sd {
    #define lch (u<<1)
    #define rch (u<<1|1)
    using T=int;
    vector<vector<T>> seg;
    int rng_l,rng_r;

    void add(int u,int x,int y,int l,int r,T val) {
        if(x>r||y<l) return;
        if(x<=l&&y>=r) seg[u].emplace_back(val);
        else {
            int mid=(l+r)/2;
            add(lch,x,y,l,mid,val);
            add(rch,x,y,mid+1,r,val);
        }
    }
    void add(int x,int y,T val) {
        add(1,x,y,rng_l,rng_r,val);
    }

    void solve(int u,int l,int r) {
        // apply
        for(auto x:seg[u]) {

        }

        // update ans
        if(l==r) ;
        else {
            int mid=(l+r)/2;
            solve(lch,l,mid);
            solve(rch,mid+1,r);
        }

        // undo
    }
    void solve() {
        solve(1,rng_l,rng_r);
    }

    void init(int l,int r) {
        rng_l=l,rng_r=r;
        seg.clear();
        seg.resize(4<<__lg(r-l+1)|1);
    }
```

```
    #undef lch
    #undef rch
}
```

字典树

字典树

```
struct Trie {
    constexpr static int A=26,B='a';
    struct Node {
        int ch[A];
        int cnt;
    };
    vector<Node> tr;

    int new_node() { tr.push_back({}); return tr.size()-1; }

    int extend(int u,int x) {
        if(!tr[u].ch[x-B]) tr[u].ch[x-B]=new_node();
        tr[tr[u].ch[x-B]].cnt++;
        return tr[u].ch[x-B];
    }

    template<typename T> void insert(const T &s) {
        int u=0;
        for(auto x:s) u=extend(u, x);
    }

    void clear() { tr.clear(); new_node(); }
    Trie() { clear(); }
    Trie(int size) { tr.reserve(size); clear(); }
} trie;
```

01-字典树

```
template<typename I,int H=sizeof(I)*8-1-is_signed<I>()>
struct BinaryTrie {
    struct Node {
        int ch[2];
        int cnt;
    };
    vector<Node> tr;

    int new_node() {
        tr.push_back({});
        return tr.size()-1;
```

```
    }

    void insert(int v) {
        for(int i=H,u=0;i>=0;i--) {
            bool x=v>>i&1;
            if(!tr[u].ch[x]) tr[u].ch[x]=new_node();
            u=tr[u].ch[x];
            tr[u].cnt++;
        }
    }

    void erase(int v) {
        for(int i=H,u=0;i>=0;i--) {
            bool x=v>>i&1;
            u=tr[u].ch[x];
            tr[u].cnt--;
        }
    }

    I xor_max(int v) {
        I res{};
        for(int i=H,u=0;i>=0;i--) {
            bool x=v>>i&1^1;
            if(tr[tr[u].ch[x]].cnt) {
                res|=1<<i;
                u=tr[u].ch[x];
            }
            else u=tr[u].ch[x^1];
        }
        return res;
    }

    I xor_min(int v) {
        I res{};
        for(int i=H,u=0;i>=0;i--) {
            bool x=v>>i&1;
            if(tr[tr[u].ch[x]].cnt) u=tr[u].ch[x];
            else {
                res|=1<<i;
                u=tr[u].ch[x^1];
            }
        }
        return res;
    }

    void clear() {
        tr.clear();
        new_node();
    }

    explicit BinaryTrie(int size=0) {
        tr.reserve(size*(H+1));
        clear();
```

```
    }
};
```

可持久化01-字典树

```
template<typename I> struct PersistentBinaryTrie {
    constexpr static int H=sizeof(I)*8-1;
    struct Node {
        int ch[2];
        int cnt;
    };
    vector<Node> tr;
    vector<int> root;

    int ver() { return root.size()-1; }

    int new_root() {
        root.push_back({});
        return ver();
    }

    int new_node() {
        tr.push_back({});
        return tr.size()-1;
    }

    void insert(int &rt,int v,int val) {
        int u=rt=new_node();
        tr[u]=tr[v];
        for(int i=H;i>=0;i--) {
            bool x=val>>i&1;
            u=tr[u].ch[x]=new_node();
            v=tr[v].ch[x];
            tr[u]=tr[v];
            tr[u].cnt++;
        }
    }
    int insert(int val) {
        new_root();
        insert(root[ver()],root[ver()-1],val);
        return ver();
    }

    I xor_max(int u,int val) {
        u=root[u];
        I res{};
        for(int i=H;i>=0;i--) {
            bool x=val>>i&1^1;
            if(tr[tr[u].ch[x]].cnt) {
                res|=1<<i;
                u=tr[u].ch[x];
```

```
            }
            else u=tr[u].ch[x^1];
        }
        return res;
    }

    I range_xor_max(int u,int v,int val) {
        u=root[u],v=root[v];
        I res{};
        for(int i=H;i>=0;i--) {
            bool x=val>>i&1^1;
            if(tr[tr[u].ch[x]].cnt-tr[tr[v].ch[x]].cnt) {
                res|=1<<i;
                u=tr[u].ch[x];
                v=tr[v].ch[x];
            }
            else u=tr[u].ch[x^1],v=tr[v].ch[x^1];
        }
        return res;
    }

    void clear() {
        tr.clear();
        new_root();
        new_node();
    }

    explicit PersistentBinaryTrie(int size=0) {
        tr.reserve(size*(H+1));
        clear();
    }
};
```

并查集

并查集

并查集能够高效地处理集合信息。

```
struct DisjointUnionSet {
    vector<int> fa,sz;

    void init(int n) {
        fa.resize(n+1);
        sz.assign(n+1,1);
        iota(fa.begin(), fa.end(), 0);
    }

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

```
    }

    bool same(int x,int y) {
        return find(x)==find(y);
    }

    bool join(int x,int y) {
        x=find(x);
        y=find(y);
        if(x==y) return false;
        // if(sz[x]<sz[y]) swap(x,y);
        sz[x]+=sz[y];
        fa[y]=x;
        return true;
    }

    int size(int x) {
        return sz[find(x)];
    }

    DisjointUnionSet() = default;
    DisjointUnionSet(int n) { init(n); }
} dsu;
```

可撤销并查集

通过一个额外的栈保存修改历史来实现撤销操作。由启发式合并保证复杂度。

```
struct DisjointUnionSet {
    vector<int> fa,sz;
    vector<pair<int&,int>> fah,szh;

    void init(int n) {
        fah.clear();
        szh.clear();
        fa.resize(n+1);
        sz.assign(n+1,1);
        iota(fa.begin(), fa.end(), 0);
    }

    int find(int x) {
        while(x!=fa[x]) x=fa[x];
        return x;
    }

    bool same(int x,int y) {
        return find(x)==find(y);
    }

    bool join(int x,int y) {
        x=find(x);
```

```
        y=find(y);
        if(x==y) {
            fah.emplace_back(fa[0],fa[0]);
            szh.emplace_back(sz[0],sz[0]);
            return false;
        }
        if(sz[x]<sz[y]) swap(x,y);
        fah.emplace_back(fa[y],fa[y]);
        szh.emplace_back(sz[x],sz[x]);
        sz[x]+=sz[y];
        fa[y]=x;
        return true;
    }

    void undo() {
        assert(!fah.empty());
        fah.back().first=fah.back().second;
        szh.back().first=szh.back().second;
        fah.pop_back(),szh.pop_back();
    }

    int size(int x) {
        return sz[find(x)];
    }

    DisjointUnionSet() = default;
    DisjointUnionSet(int n) { init(n); }
} dsu;
```

树链剖分

重链剖分能将树上路径转为$\mathcal{O}(\log n)$级别的连续区间，从而将树上问题转化为区间问题。预处理时间复杂度$\mathcal{O}(n)$,单次路径剖分时间复杂度$\mathcal{O}(\log n)$。

关于实现上的易错点：把`id[u]`写成`u`，务必注意。

```
// ! don't confuse dfn id with node id
namespace hpd {
    using PII=pair<int,int>;
    constexpr int N=1e5+10;
    int id[N],w[N],ori[N],cnt;
    int dep[N],sz[N],top[N],p[N],hch[N];
    vector<int> adj[N];

    void dfs1(int u,int fa,int d) {
        dep[u]=d,p[u]=fa,sz[u]=1;
        for(int v:adj[u]) {
            if(v==fa) continue;
            dfs1(v,u,d+1);
            sz[u]+=sz[v];
```

```
            if(sz[hch[u]]<sz[v]) hch[u]=v;
        }
    }

    void dfs2(int u,int t) {
        id[u]=++cnt,ori[id[u]]=u,top[u]=t;
        if(!hch[u]) return;
        dfs2(hch[u],t);
        for(int v:adj[u])
            if(v!=p[u]&&v!=hch[u]) dfs2(v,v);
    }

    int lca(int x,int y) {
        while(top[x]!=top[y]) {
            if(dep[top[x]]<dep[top[y]]) swap(x,y);
            x=p[top[x]];
        }
        if(dep[x]<dep[y]) swap(x,y);
        return y;
    }

    vector<PII> decompose(int x,int y) {
        vector<PII> res;
        while(top[x]!=top[y]) {
            if(dep[top[x]]<dep[top[y]]) swap(x,y);
            res.emplace_back(id[top[x]],id[x]);
            x=p[top[x]];
        }
        if(dep[x]<dep[y]) swap(x,y);
        res.emplace_back(id[y],id[x]);
        return res;
    }

    PII decompose(int x) {
        return { id[x],id[x]+sz[x]-1 };
    }

    void init() {
        dfs1(1,-1,1); dfs2(1,1);
    }

    void clear(int n) {
        cnt=0;
        fill(hch, hch+n+1, 0);
    }
}
```

稀疏表

倍增维护区间最大值。

```
template<int size,typename T=int> struct SparseTable {
    constexpr static int M=__lg(size)+1;
    T st[M][size];

    T merge(const T &x,const T &y) {
        return max(x,y);
    }

    void build(int n) {
        // for(int i=1;i<=n;i++) st[0][i]=arr[i]; // todo
        for(int k=1,t=1<<k;k<M;k++,t<<=1)
            for(int i=1,j=i+t-1,mid=i+t/2;j<=n;i++,j++,mid++)
                st[k][i]=merge(st[k-1][i],st[k-1][mid]);
    }

    T query(int l,int r) {
        if(r<l) return 0;
        int k=__lg(r-l+1);
        return merge(st[k][l],st[k][r-(1<<k)+1]);
    }
};
```

Link Cut Tree

在静态的树结构中，树剖是维护路径/子树修改的实用方法，但树剖无法处理动态的树结构。而LCT正是维护动态树结构的强大工具。LCT擅长维护树链信息，能很容易地实现路径修改+路径查询。

LCT也能处理一些子树相关的问题，稍加修改即可支持单点修改+子树查询，前提是维护的信息可减。但更加通用的子树修改+子树查询，基本只能用LCT的升级版TopTree来解决。

LCT的所有操作都是 $\mathcal{O}(\log n)$ 的，但是由于自带的常数巨大无比，因此可以把LCT的复杂度近似看作 $\mathcal{O}(\log^2 n)$。

LCT

> **三年竞赛一场空，不判自环见祖宗**

```
template<class Info,class Tag,int MAX_SIZE,
         bool CHECK_LINK = 0,bool CHECK_CUT = 0,bool ASSERT = 0>
struct LinkCutTree {
    #define lch tr[u].ch[0]
    #define rch tr[u].ch[1]
    #define wch(u) (tr[tr[u].p].ch[1]==u)

    struct Node {
        int ch[2],p;
        bool rev;
        Info info;
        Tag tag;
```

```
        void update(const Tag &x) {
            info.update(x);
            tag.update(x);
        }
    };
    array<Node, MAX_SIZE> tr;
    array<int, MAX_SIZE> stk;

    bool is_root(int u) {
        return tr[tr[u].p].ch[0]!=u&&tr[tr[u].p].ch[1]!=u;
    }

    void pushup(int u) {
        tr[u].info.pushup(tr[lch].info,tr[rch].info);
    }

    void pushrev(int u) {
        tr[u].info.reverse();
        swap(lch,rch);
        tr[u].rev^=1;
    }

    void pushdn(int u) {
        if(tr[u].rev) {
            if(lch) pushrev(lch);
            if(rch) pushrev(rch);
            tr[u].rev=0;
        }
        if(lch) tr[lch].update(tr[u].tag);
        if(rch) tr[rch].update(tr[u].tag);
        tr[u].tag.clear();
    }

    void rotate(int x) {
        int y=tr[x].p,z=tr[y].p,k=wch(x);
        if(!is_root(y)) tr[z].ch[wch(y)]=x;
        tr[y].ch[k]=tr[x].ch[!k],tr[tr[y].ch[k]].p=y;
        tr[x].ch[!k]=y,tr[y].p=x,tr[x].p=z;
        pushup(y),pushup(x);
    }

    void splay(int u) {
        int top=0,fa=u;
        stk[++top]=fa;
        while(!is_root(fa)) stk[++top]=fa=tr[fa].p;
        while(top) pushdn(stk[top--]);
        for(;!is_root(u);rotate(u))
            if(!is_root(fa=tr[u].p)) rotate(wch(u)==wch(fa)?fa:u);
    }

    int access(int u) {
        int v=0;
        for(;u;v=u,u=tr[u].p)
            splay(u),rch=v,pushup(u);
```

```
        return v;
    }

    void make_root(int u) {
        access(u);
        splay(u);
        pushrev(u);
    }

    int split(int u,int v) {
        make_root(u);
        access(v);
        splay(v);
        return v;
    }

    int find_root(int u) {
        access(u);
        splay(u);
        while(lch) pushdn(u),u=lch;
        splay(u);
        return u;
    }

    bool same(int u,int v) {
        make_root(u);
        return find_root(v)==u;
    }

    bool link(int u,int v) {
        make_root(u);
        if(CHECK_LINK&&find_root(v)==u)
            return assert(!ASSERT),0;
        tr[u].p=v;
        return 1;
    }

    bool cut(int u,int v) {
        make_root(u);
        if(CHECK_CUT&&!(find_root(v)==u&&rch==v&&!tr[v].ch[0]))
            return assert(!ASSERT),0;
        else access(v),splay(u);
        rch=tr[v].p=0;
        pushup(u);
        return 1;
    }

    int lca(int u,int v) {
        access(u);
        return access(v);
    }

    int lca(int rt,int u,int v) {
        make_root(rt);
```

```
        return lca(u,v);
    }

    void modify(int u,const Tag &x) {
        if(!is_root(u)) splay(u);
        tr[u].update(x);
    }

    Info &info(int u) {
        return tr[u].info;
    }

    #undef lch
    #undef rch
    #undef wch
};

struct Tag {

    void update(const Tag &x) {

    }

    void clear() {

    }
};

struct Info {

    //* lch+parent+rch
    void pushup(const Info &l,const Info &r) {

    }

    void update(const Tag &x) {

    }

    void reverse() {}
};

LinkCutTree<Info,Tag,N> lct;
```

调试&卡常

关于常数问题，由于 `Info` 中的 `pushup` 函数通常是LCT中调用最为频繁的函数，因此这部分的细节处理就很关键。具体来说：

- 在维护MST时，手写 `if else` 通常会比开一个 `array/vector` 然后 `sort` 快一倍。然后将边信息改为保存边id也能些微减小常数，但是用起来不如直接暴力存边直观。
- 有时并不需要用的Tag或者存在一些空函数，把这些东西删掉能减少一部分常数。

路径修改+路径查询

这是LCT最基础的用法，没有太多可讲的点。需要注意修改的时候，点必须 `splay` 到根。

```
void modify(int u,const Tag &x) {
    if(!is_root(u)) splay(u);
    tr[u].update(x);
}
```

然后如果维护的信息具有方向性，记得合理实现 `reverse` 函数。

单点修改+子树查询

在 `Info` 中将要维护的信息分离为虚/实两部分，然后添加两个新方法，用来处理添加/删除虚信息。要求信息必须可减。

单点加法+子树求和的例子。常见的用法还有维护子树大小，也是类似的写法。

```
struct Info {
    LL val=0;
    LL sum=0,vsum=0;

    void pushup(const Info &l,const Info &r) {
        // 左右子树的和+所有虚子树的和+自己的value
        sum=l.sum+r.sum+vsum+val;
    }

    // 添加一个新的虚子树
    void add(const Info &x) {
        vsum+=x.sum;
    }

    // 删去一个虚子树
    void sub(const Info &x) {
        vsum-=x.sum;
    }
};
```

然后修改以下几个函数。

`access` 的过程中产生了边的虚实变化，因此需要修改。

```
int access(int u) {
    int v=0;
    for(;u;v=u,u=tr[u].p) {
        splay(u);
        // 实 -> 虚，添加虚子树
        if(rch) tr[u].info.add(info(rch));
```

```
        // 虚 -> 实, 删去虚子树
        if(v) tr[u].info.sub(info(v));
        rch=v,pushup(u);
    }
    return v;
}
```

`link` 添加了一个新的虚儿子 `u` 到 `p`，因此也需要修改。注意必须要先 `make_root` 再做修改。

```
bool link(int p,int u) {
    make_root(u);
    if(CHECK_LINK&&find_root(p)==u)
        return assert(!ASSERT),0;
    make_root(p);
    tr[p].info.add(info(u));
    tr[u].p=p;
    pushup(p);
    return 1;
}
```

而在 `cut` 中，我们断开的是实边，因此不需要做修改，`pushup` 会维护信息的变化。

务必注意，和常规的LCT不同，在修改之前，必须先 `access` 然后再 `splay`，以保证点为整颗树的根。统计子树信息时，也必须保证为树根。

```
// modify
lct.access(u);
lct.splay(u);
lct.info(u)={...};

// query
lct.access(u);
lct.splay(u);
cout<<lct.info(u).sum<<endl;
```

如果要查询指定子树的信息，而不是整棵树的信息，例如 u 点以 p 作为为父节点时 u 的子树和。那么我们先 `cut` 掉 (u,p)，查询完之后再 `link` 回去即可。

```
lct.cut(u,p);
// 这里不需要再转到根了，因为cut函数保证了cut完之后u,p都是根
cout<<lct.info(u).sum<<endl;
lct.link(u,p);
```

维护MST

维护MST几乎可以说是LCT最常见的应用。

由于LCT不方便直接操作边，我们可以使用虚点的技巧来将边信息保存为点信息。即将 (x,y) 边拆分为 $(x,z),(z,y)$ ，其中 z 为新建的虚点，将边权保存为 z 的点权。x,y 为非边点，点权为 ∞ 或 0。

```
for(int i=1;i<=m;i++) {
    int u,v,val;
    cin>>u>>v>>val;
    int w=i+n;
    lct.info(w)={val,val,{u,v,w},{u,v,w}};
    add_edge(u,v,w,val);
}
```

最小生成树

以最小生成树为例，我们在LCT中维护以下信息：

- 节点边权
- 子树最大边权
- 节点所代表的边
- 子树最大边权所代表的边

```
struct MaxInfo {
    int v=0,maxv=0;
    Edge e,maxe;

    void pushup(const MaxInfo &l,const MaxInfo &r) {
        if(l.maxv>=r.maxv) maxv=l.maxv,maxe=l.maxe;
        else maxv=r.maxv,maxe=r.maxe;
        if(v>maxv) maxv=v,maxe=e;
    }
};
```

`Edge` 通常为 `tuple<int,int,int>` 或者 `tuple<int,int,int,int>`，取决于是否要保存边权。

在维护最小生成树的过程，需要根据是否已经连通来分类讨论。

```
// 添加(u,v)边，虚点为w，边权为val
auto add_edge=[&](int u,int v,int w,int val) {
    // 如果已经连通，那么需要判断环上的最大边权是否比val大
    if(lct.same(u, v)) {
        int rt=lct.split(u, v);
        if(lct.info(rt).maxv>val) {
            auto [x,y,z]=lct.info(rt).maxe;
            lct.cut(x, z);
            lct.cut(y, z);
            lct.link(u, w);
            lct.link(v, w);
        }
    }
```

```
    else {
        lct.link(u, w);
        lct.link(v, w);
        cnt++;
    }
};
```

删边则比较简单。

```
// 删除边(u,v)，判断其中一个点是否与虚点连通即可
auto del_edge=[&](int u,int v,int w) {
    if(lct.same(u, w)) {
        lct.cut(u, w);
        lct.cut(v, w);
        cnt--;
    }
};
```

最大生成树

将最小生成树对称过来即可。

```
struct MinInfo {
    int v=INF,minv=INF;
    Edge e,mine;

    void pushup(const MinInfo &l,const MinInfo &r) {
        if(l.minv<r.minv) minv=l.minv,mine=l.mine;
        else minv=r.minv,mine=r.mine;
        if(v<minv) minv=v,mine=e;
    }
};
```

处理加边。

```
auto add_edge=[&](int u,int v,int w,int val) {
    if(lct.same(u, v)) {
        int rt=lct.split(u, v);
        if(lct.info(rt).minv<val) {
            auto [x,y,z]=lct.info(rt).mine;
            lct.cut(x, z);
            lct.cut(y, z);
            lct.link(u, w);
            lct.link(v, w);
        }
    }
    else {
```

```
        lct.link(u, w);
        lct.link(v, w);
        cnt++;
    }
};
```

复杂度 $\mathcal{O}(m\log m)$。

可撤销地维护MST

需要可撤销地维护MST，通常是问题的边有时效性，或者有加删边，需要使用线段树分治。

和DSU不同，LCT本身支持删除，所以实现上不需要特殊的技巧。

```
// 把边丢到线段树上
void add(int u,int x,int y,int l,int r,Edge val) {
    if(x>r||y<l) return;
    if(x<=l&&y>=r) seg[u].emplace_back(val);
    else {
        int mid=(l+r)/2;
        add(lch,x,y,l,mid,val);
        add(rch,x,y,mid+1,r,val);
    }
}

// 维护MST边权和的例子
LL sum;
void dfs(int u,int l,int r) {
    LL bak=sum;
    vector<Edge> del,add;

    for(auto [x,y,z,w]:seg[u]) {
        // 修改MST，把加删的边丢进del和add保存
    }

    if(l==r) ans[l]=sum;
    else {
        int mid=(l+r)/2;
        dfs(lch, l, mid);
        dfs(rch, mid+1, r);
    }

    // 这里务必按照逆序做，否则会导致加删不存在的边而导致RE
    while(add.size()) {
        auto [x,y,z,w]=add.back();
        lct.cut(x, z);
        lct.cut(y, z);
        tie(x,y,z,w)=del.back();
        lct.link(x, z);
        lct.link(y, z);
        add.pop_back();
```

```
        del.pop_back();
    }
    sum=bak;
}
```

复杂度 $\mathcal{O}(m\log m\log t)$。$t$ 为时间跨度。

维护有根树

有些时候，LCT维护的树并不是无根树，我们需要保证每次操作树的根都是不变的。一种简单有效的方法是使用无根树的方式维护有根树，即每次操作前记录一下根，操作完之后再 `make_root` 回去。

不过直接维护有根树能减小因为额外的 `make_root` 带来的常数开销。当固定维护的根时显然不能再使用 `make_root` 函数（也没有必要）。随之而来的，我们需要修改调用了 `make_root` 的函数。

首先是 `link` 和 `cut`。因为是有根树，所以 `cut` 直接cut掉 `u` 和父节点之间的边即可。

```
void link(int u,int p) {
    access(u);
    splay(u);
    tr[u].p=p;
}

void cut(int u) {
    access(u);
    splay(u);
    lch=tr[lch].p=0;
}
```

如果需要知道每个点在原树上的父节点，可以用一个数组维护，或者直接在LCT上查询（感觉不如前者）：

```
int findfa(int u) {
    access(u);
    splay(u);
    u=lch;
    pushdn(u);
    while(rch) u=rch,pushdn(u);
    if(u) splay(u);
    return u;
}
```

然后是 `same`，判断一下根是否相同即可。

```
bool same(int u,int v) {
    return find_root(u)==find_root(v);
}
```

`split` 则完全没有用了，因为根固定只能处理根到子节点的路径。

虚树

能在 $\mathcal{O}(k\log n)$ 时间内提取树上的 k 个关键点建成一棵新树,并且新树的点数不超过 $2k$。

```
namespace vt {
    constexpr int N=1e5+10,M=__lg(N);
    vector<int> vt[N],adj[N];
    int stk[N],top,id[N],idx;
    int fa[N][M+1],dep[N];
    bool key[N];

    void lca_init(int u,int p) {
        dep[u]=dep[p]+1;
        for(int v:adj[u]) {
            if(v==p) continue;
            fa[v][0]=u;
            for(int i=1;i<=M;i++)
                fa[v][i]=fa[fa[v][i-1]][i-1];
            lca_init(v,u);
        }
    }

    int lca(int u,int v) {
        if(dep[u]<dep[v]) swap(u,v);
        for(int k=M;~k;k--)
            if(dep[fa[u][k]]>=dep[v])
                u=fa[u][k];
        if(u==v) return u;
        for(int k=M;~k;k--)
            if(fa[u][k]!=fa[v][k])
                u=fa[u][k],v=fa[v][k];
        return fa[u][0];
    }

    void relabel(int u,int fa) {
        id[u]=++idx;
        for(int v:adj[u]) if(v!=fa) relabel(v, u);
    }

    void build(vector<int> &vec) {
        sort(vec.begin(),vec.end(),[](int x,int y) {
            return id[x]<id[y];
        });

        // TODO clearup dirt memory
        auto clear=[&](int u) {
            vt[u].clear();
            key[u]=0;
        };
```

```
        auto add=[&](int u,int v) {
            vt[u].emplace_back(v);
        };

        clear(1);
        stk[top=0]=1;
        for(int u:vec) {
            if(u==1) continue;
            int p=lca(u,stk[top]);
            if(p!=stk[top]) {
                while(id[p]<id[stk[top-1]])
                    add(stk[top-1],stk[top]),top--;
                if(id[p]!=id[stk[top-1]])
                    clear(p),add(p,stk[top]),stk[top]=p;
                else add(p,stk[top--]);
            }
            clear(u);
            stk[++top]=u;
            key[u]=1;
        }
        for(int i=0;i<top;i++) add(stk[i],stk[i+1]);
    }

    void init() {
        lca_init(1, 0);
        relabel(1, 0);
    }

    void clear(int n) {
        idx=0;
        for(int i=0;i<=n;i++) adj[i].clear();
    }
}
```

Segment Set

线段集（珂朵莉树）。插入删除的均摊复杂度为 $\mathcal{O}(\log n)$。

```
template<class Info=int,typename I=int> struct SegmentSet {
    struct Seg {
        I l;
        mutable I r;
        mutable Info v;
        bool operator<(const Seg &x) const {
            return l<x.l;
        }
        friend ostream &operator<<(ostream &os,const Seg &x) {
            return os<<"["<<x.l<<"->"<<x.r<<"]";
        }
```

```
        Seg(I l,I r={},Info v={}): l(l),r(r),v(v) {}
    };
    set<Seg> st;
    using Iter=typename set<Seg>::iterator;

    // 找最左与[l,r]有交的线段
    // 如果不存在返回[l,r]右侧的第一个线段
    Iter first_inter(I l,I r) {
        Iter it=--st.upper_bound({l});
        if(it->r<l) it++;
        return it;
    }

    // 找最右与[l,r]有交的线段
    // 如果不存在返回[l,r]左侧的第一个线段
    Iter last_inter(I l,I r) {
        return --st.upper_bound({r});
    }

    // 找到包含p点的线段，分裂为 [l,p-1],[p,r]
    // 返回[p,r]，如果没有线段包含p，返回end()
    Iter split(I p) {
        Iter it=st.lower_bound({p});
        if(it->l==p) return it;
        if((--it)->r<p) return st.end();
        auto [l,r,v]=*it;
        st.erase(it);
        st.emplace(l,p-1,v);
        return st.emplace(p,r,v).first;
    }

    // 将与it相接的同色线段拼接，返回拼接后的线段
    Iter merge(Iter it) {
        auto work=[&](Iter lit,Iter rit) {
            if(lit->r+1==rit->l&&lit->v==rit->v) {
                lit->r=rit->r;
                st.erase(rit);
                return true;
            }
            return false;
        };
        auto lit=prev(it),rit=next(it);
        if(work(lit,it)) return work(lit,rit),lit;
        return work(it,rit),it;
    }

    // 删去与[l,r]相交的部分
    Iter erase(I l,I r) {
        split(l),split(r+1);
        auto lit=first_inter(l,r);
        auto rit=++last_inter(l,r);
        return st.erase(lit,rit);
    }
```

```
    // 插入线段[l,r]并删去与[l,r]相交的部分
    Iter insert(I l,I r,Info v) {
        erase(l,r);
        return merge(st.emplace(l,r,v).first);
    }

    // 找到第一个没有被线段覆盖的位置
    I first_uncovered() {
        auto it=st.begin();
        if(it->r+2<next(it)->l) return it->r+2;
        else it=next(it);
        for(;;) {
            if(it->r+1<next(it)->l) return it->r+1;
            else it=next(it);
        }
    }

    SegmentSet(I l,I r) {
        st.emplace(l-2,l-2);
        st.emplace(r+2,r+2);
    }
};
```