可持久化线段树

算法简介

一种可以维护历史版本的线段树，时间复杂度和空间复杂度均为 $O\left(n+m\log n\right)$

算法思想

显然如果对每个历史版本都重建线段树保存时间复杂度和空间复杂度均为 $O\left(nm\right)$

但如果可以重复利用部分结点信息，则可以降低时间复杂度和空间复杂度。

考虑仅对线段树修改路径上的点进行复制、修改，每次修改的时间复杂度和空间复杂度可降为$O\left(\log n\right)$

算法应用

可持久化数组

洛谷p3919

题意

维护一个长度为 n 的数组，支持下列操作：

1. 在某个历史版本上修改某一个位置上的值。

2. 访问某个历史版本上的某一位置的值。

每进行一次操作就会生成一个新的版本，版本编号即为当前操作的编号。

题解

用可持久化线段树维护数组，线段树叶子节点记录对应对应位置数组数值，时间复杂度和空间复杂度均为 $O\left(n+m\log n\right)$

```
const int MAXN=1e6+5;
struct Node{
    int lef,rig,val;
};
Node node[MAXN*20];
int cnt,root[MAXN],a[MAXN];
int nodecopy(int k){
    node[++cnt]=node[k];
    return cnt;
}
```

```
void build(int &k,int lef,int rig){
    k=++cnt;
    int mid=lef+rig>>1;
    if(lef==rig)
    return node[k].val=a[mid],void();
    build(node[k].lef,lef,mid);
    build(node[k].rig,mid+1,rig);
}
void update(int &k,int p,int lef,int rig,int pos,int v){
    k=nodecopy(p);
    if(lef==rig)
    return node[k].val=v,void();
    int mid=lef+rig>>1;
    if(mid<pos)
    update(node[k].rig,node[p].rig,mid+1,rig,pos,v);
    else
    update(node[k].lef,node[p].lef,lef,mid,pos,v);
}
int query(int k,int lef,int rig,int pos){
    if(lef==rig)
    return node[k].val;
    int mid=lef+rig>>1;
    if(mid<pos)
    return query(node[k].rig,mid+1,rig,pos);
    else
    return query(node[k].lef,lef,mid,pos);
}
int main()
{
    int n=read_int(),q=read_int();
    _rep(i,1,n)
    a[i]=read_int();
    build(root[0],1,n);
    int rt,pos,v,opt;
    _rep(i,1,q){
        rt=read_int(),opt=read_int(),pos=read_int();
        if(opt==1){
            v=read_int();
            update(root[i],root[rt],1,n,pos,v);
        }
        else{
            root[i]=root[rt];
            enter(query(root[rt],1,n,pos));
        }
    }
    return 0;
}
```

静态区间第 k 小

洛谷p3834

题意

维护一个长度为 n 的数组，m 次查询，每次查询给定闭区间内第 k 小的数。

题解

关于第 k 小问题，首先想到使用类似名次树的方法查询。

记数组元素为 $a[1\sim n]$，建立空线段树，依次在线段树第 $a[i]$ 位置插入一个元素，线段树需维护区间所含元素个数，并记录历史版本。

设查询区间为 $[l,r]$ ，这历史版本 r 与 历史版本 $l-1$ 的对应节点差值即为 元素 $a[l\sim r]$ 在节点维护范围内所含元素个数。

在此基础上，便可以使用名次树查询第 k 小元素，同时考虑到数据范围需要对数组进行离散化，时间复杂度和空间复杂度均为 $O\left(n+m\log n\right)$。

```
const int MAXN=1e6+5;
struct Node{
    int lef,rig,val;
};
Node node[MAXN*20];
int cnt,root[MAXN],a[MAXN],b[MAXN];
inline int nodecopy(int k){
    node[++cnt]=node[k];
    return cnt;
}
void build(int &k,int lef,int rig){
    k=++cnt;
    int mid=lef+rig>>1;
    if(lef==rig)
    return;
    build(node[k].lef,lef,mid);
    build(node[k].rig,mid+1,rig);
}
void update(int &k,int p,int lef,int rig,int pos){
    k=nodecopy(p);
    node[k].val++;
    if(lef==rig)
    return;
    int mid=lef+rig>>1;
    if(mid<pos)
    update(node[k].rig,node[p].rig,mid+1,rig,pos);
    else
    update(node[k].lef,node[p].lef,lef,mid,pos);
}
int query(int k1,int k2,int lef,int rig,int rk){
```

```
    int mid=lef+rig>>1;
    if(lef==rig)
    return mid;
    int lc1=node[k1].lef,lc2=node[k2].lef,lz=node[lc2].val-node[lc1].val;
    if(rk>lz)
    return query(node[k1].rig,node[k2].rig,mid+1,rig,rk-lz);
    else
    return query(node[k1].lef,node[k2].lef,lef,mid,rk);
}
int main()
{
    int n=read_int(),q=read_int(),m;
    build(root[0],1,n);
    _rep(i,1,n)
    a[i]=read_int();
    memcpy(b+1,a+1,sizeof(int)*n);
    sort(b+1,b+n+1);
    m=unique(b+1,b+n+1)-b;
    _rep(i,1,n)
    update(root[i],root[i-1],1,n,lower_bound(b+1,b+m+1,a[i])-b);
    int l,r,k,ans;
    while(q--){
        l=read_int(),r=read_int(),k=read_int();
        ans=query(root[l-1],root[r],1,n,k);
        enter(b[ans]);
    }
    return 0;
}
```

可持久化并查集

洛谷p3402

题意

维护 n 个集合，每个集合初始时只有一个元素 i。

接下来 m 操作，操作分三种：

1. 合并 a 和 b 所在集合。

2. 回到第 k 次操作状态。

3. 询问 a 和 b 是否在同一集合。

题解

显然不能用路径压缩，可以考虑启发式合并，将深度小的集合并到深度大的里面。

考虑将普通版本的启发式合并的数组改成可持久化数组即可，时间复杂度 $O\left(n+m\log^2 n\right)$，空间复杂度为 $O\left(n+m\log n\right)$。

```
const int MAXN=1e5+5;
struct Node{
    int lef,rig,fa,d;
};
Node node[MAXN*20];
int cnt,root[MAXN<<1],n,q;
inline int nodecopy(int k){
    node[++cnt]=node[k];
    return cnt;
}
void build(int &k,int lef,int rig){
    k=++cnt;
    int mid=lef+rig>>1;
    if(lef==rig)
    return node[k].fa=mid,void();
    build(node[k].lef,lef,mid);
    build(node[k].rig,mid+1,rig);
}
void update_fa(int &k,int p,int lef,int rig,int pos,int F){
    k=nodecopy(p);
    if(lef==rig){
        node[k].fa=F;
        return;
    }
    int mid=lef+rig>>1;
    if(mid<pos)
    update_fa(node[k].rig,node[p].rig,mid+1,rig,pos,F);
    else
    update_fa(node[k].lef,node[p].lef,lef,mid,pos,F);
}
void update_dep(int k,int lef,int rig,int pos){
    if(lef==rig){
        node[k].d++;
        return;
    }
    int mid=lef+rig>>1;
    if(mid<pos)
    update_dep(node[k].rig,mid+1,rig,pos);
    else
    update_dep(node[k].lef,lef,mid,pos);
}
int query_id(int k,int lef,int rig,int pos){
    int mid=lef+rig>>1;
    if(lef==rig)
    return k;
    if(mid<pos)
    return query_id(node[k].rig,mid+1,rig,pos);
    else
    return query_id(node[k].lef,lef,mid,pos);
}
```

```
int Find(int rt,int pos){
    int id=query_id(rt,1,n,pos);
    return node[id].fa==pos?id:Find(rt,node[id].fa);
}
void Union(int &rt,int p,int a,int b){
    int x=Find(p,a),y=Find(p,b);
    if(node[x].fa==node[y].fa)
    return rt=p,void();
    if(node[x].d>node[y].d)
    swap(x,y);
    update_fa(rt,p,1,n,node[x].fa,node[y].fa);
    if(node[x].d==node[y].d)
    update_dep(rt,1,n,node[y].fa);
}
int main()
{
    n=read_int(),q=read_int();
    build(root[0],1,n);
    int opt,a,b;
    _rep(i,1,q){
        opt=read_int();
        if(opt==1){
            a=read_int(),b=read_int();
            Union(root[i],root[i-1],a,b);
        }
        else if(opt==2)
        root[i]=root[read_int()];
        else{
            root[i]=root[i-1];
            a=read_int(),b=read_int();
            int x=Find(root[i],a),y=Find(root[i],b);
            if(node[x].fa==node[y].fa)
            puts("1");
            else
            puts("0");
        }
    }
    return 0;
}
```