云某人的数据结构

- 云某人的数据结构
  - 单调栈
    - 左侧第一个比自己小/大元素
  - 单调队列
    - 滑动窗口
  - 并查集
    - 路径压缩
  - 01trie
    - 模板
      - 求异或最大值
  - 可持久化01tire
    - 模板
  - 树状数组
    - 单点修改维护区间和
    - 区间修改维护区间和
    - 维护逆序对
  - 线段树
    - 模板
      - 区间修改线段树
        - 区间和
        - 区间RMQ
      - 单点修改线段树
      - 权值线段树
    - 多种 lazy_tag
      - 乘法tag 和 加法tag
    - 各种区间操作
      - 加上等差数列
    - 各种区间信息
      - 区间平方和
      - 区间乘积和
    - 区间信息性质
      - 开方性质
    - 开多棵线段树
  - 可持久化线段树
    - 主席树
  - 树链剖分
    - 轻重链剖分
    - dsu on tree
  - 珂朵莉树
    - `set`
    - `map`

单调栈

栈内元素始终单调（栈顶 **更** 满足要求）

> 如更大，更小

```
vector<int> stp;
stack<int> st;
for(int i=1;i<=n;++i)
{
    while(st.size() && check(a[i])) st.pop();
    stp.push_back(st.size() ? st.top() : -1);
    st.push(a[i]);
}
```

左侧第一个比自己小/大元素

```
// 左侧小
vector<int> ans;
stack<int> st;
for(int i=1;i<=n;++i)
{
    while(st.size() && st.top() >= a[i])    st.pop();
    ans.push_back(st.size() ? st.top() : -1);
    st.push(a[i]);
}
// 左侧大
for(int i=1;i<=n;++i)
{
    while(st.size() && st.top() <= a[i])    st.pop();
    ans.push_back(st.size() ? st.top() : -1);
    st.push(a[i]);
}
```

单调队列

滑动窗口

```
vector<int> ans;
deque<int> dq;

// 宽度为 k 的窗口中的最大值
for(int i=1;i<=n;++i)
{
    while(dq.size() && dq.front() <= i-k)   dq.pop_front();
    while(dq.size() && a[dq.back()] <= a[i])    dq.pop_back();
    dq.push_back(i);
    if(i >= k)  ans.push_back(a[dq.front()]);
}
// 宽度为 k 的窗口中的最小值
for(int i=1;i<=n;++i)
{
    while(dq.size() && dq.front() <= i-k)   dq.pop_front();
    while(dq.size() && a[dq.back()] <= a[i])    dq.pop_back();
    dq.push_back(i);
    if(i >= k)  ans.push_back(a[dq.front()]);
}
```

并查集

路径压缩

```
int rt[N];

int find(int x)// 找到根节点
{
    return rt[x] = (rt[x] == x ? x : find(rt[x]));
}

void merge(int x,int y)// 合并两联通块
{
    rt[find(x)] = find(y);
}
for(int i=1;i<=N;++i)   rt[i] = i;
```

01trie

模板

```
// 将数组加入trie树
void insert(int z)
{
    int p = 0;
    for(int i=31;i>=0;--i)
    {
        int t = (z>>i)&1;
        if(!nxt[p][t])  nxt[p][t] = ++ idx;
        p = nxt[p][t];
    }
    ext[p] = true;
}
// 判断数字是否在树中
bool find(int z)
{
    int p = 0;
    for(int i=31;i>=0;--i)
    {
        int t = (z>>i)&1;
        if(!nxt[p][t])  return false;
        p = nxt[p][t];
    }
    return ext[p];
}
```

求异或最大值

贪心高位不同的点

```
int get_max(int x)
{
    int res = 0,p = 0;
    for(int i=31;i>=0;--i)
    {
        int t = (x>>i)&1;
        if(nxt[p][t^1]) res += (1<<i), p = nxt[p][t^1];
        else if(nxt[p][t])  p = nxt[p][t];
        else    break;
    }
    return res;
}
```

可持久化01tire

模板

```
int n,idx;
int rt[N],nxt[N<<5][2],cnt[N<<5];
// 添加节点
void insert(int z,int &np,int lp)
{
    // 切记不要用传入值来修改
    np = ++ idx;
    int p = np;
    for(int i=D;i>=0;--i)
    {
        int t = (z>>i)&1;
        cnt[p] = cnt[lp]+1;
        nxt[p][0] = nxt[lp][0], nxt[p][1] = nxt[lp][1];
        nxt[p][t] = ++ idx;
        p = nxt[p][t], lp = nxt[lp][t];
    }
    cnt[p] = cnt[lp] + 1;
}

// 查询区间内匹配最大值
int query(int lp,int rp,int z)
{
    int res = 0;
    for(int i=D;i>=0;--i)
    {
        int t = (z>>i)&1;
        if(cnt[nxt[rp][!t]]-cnt[nxt[lp][!t]] > 0)
        {
            lp = nxt[lp][!t], rp = nxt[rp][!t];
            res += (1<<i);
        }
        else    lp = nxt[lp][t], rp = nxt[rp][t];
    }
    return res;
}

insert(x,rt[i],rt[i-1]);
```

树状数组

单点修改维护区间和

```
int n;
int a[N], t[N];
int lowbit(int z)
{
    return z&-z;
}
void update(int p,int z)// 单点修改
{
    for(int i=p;i<=n;i+=lowbit(i))  t[i] += z;
}
void query(int l,int r)// 区间查询
{
    int res1 = 0, res2 = 0;
    for(int i=l-1;i>=1;i-=lowbit(i))    res1 += t[i];
    for(int i=r;i>=1;i-=lowbit(i))      res2 += t[i];
    return res2-res1;
}
```

区间修改维护区间和

维护差分

其实没啥用了，我会线段树

$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} \overbrace{(n+1)d_i}^{t_1} - \sum_{i=1}^{n} \overbrace{id_i}^{t_2}$

```
int n;
vector<int> a(N), t(N),ti(N);
int lowbit(int i)
{
    return i&-i;
}
void update(int p,int z)
{
    for(int i=p;i<=n;i+=lowbit(i))  t[i] += z, ti[i] += p*z;
}
int query(int l,int r)
{
    int res1 = 0, res2;
    for(int i=l-1;i>=1;i-=lowbit(i))    res1 += (l)*t[i] - ti[i];
    for(int i=r;i>=1;i-=lowbit(i))      res2 += (r+1)*t[i] - ti[i];
    return res2 - res1;
}
```

维护逆序对

按出现顺序依次加入树状数组，每次可得当前小于（大于）等于自己的数目（也就是前面比自己小/大的数目），计算即可得逆序对。

```
int n,ans;
```

```
vector<int> t(N)
vector<int> dis,;// dis离散化

int get(int x)// 寻找离散化下标
{
    return lower_bound(dis.begin(),dis.end(),x)-dis.begin()+1;
    // 下标存入树状数组，保证下标从1开始
}

int lowbit(int z)
{
    return z&-z;
}
void update(int p)
{
    for(int i=p;i<=n;i+=lowbit(i))  t[i] ++;
}
int query(int p)
{
    int res = 0;
    for(int i=p;i>=1;i-=lowbit(i))  res += t[i];
    return res;
}

void func(void)
{
    cin >> n;
    vector<int> a(n);
    for(int i=0;i<n;++i)    cin >> a[i];
    dis = a;
    sort(dis.begin(),dis.end());
    dis.erase(unique(dis.begin(),dis.end()),dis.end());
    for(int i=0;i<n;++i)
    {
        ans += i-query(get(a[i]));
        update(get(a[i]));
    }
    cout << ans << '\n';
}
```

```
void init(void)
{
    for(int i=1;i<=n;++i)   smx[i][0] = smn[i][0] = ds[i];
    for(int i=1;i<=20;++i)
    {
        for(int j=1;j+(1<<i)-1<=n;++j)
        {
            smx[j][i] = max(smx[j][i-1],smx[j+(1<<(i-1))][i-1]);
            smn[j][i] = min(smn[j][i-1],smn[j+(1<<(i-1))][i-1]);
        }
    }
}

PII find(int l,int r)
{
```

```
    int k = log2(r-l+1);
    return {max(smx[l][k],smx[r-(1<<k)+1][k]),min(smn[l][k],smn[r-(1<<k)+1][k])};
}
```

线段树

模板

区间修改线段树

区间和

```
int n;
int a[N],t[N<<2],lz[N<<2];
// 更新节点
void update(int z,int be,int ed,int p)
{
    t[p] += (ed-be+1) * z;
    lz[p] += z;
}
// 合并
void push_up(int p)
{
    t[p] = t[p<<1] + t[p<<1|1];
}
// 下放lazy_tag
void push_down(int be,int ed,int p)
{
    int mid = (be + ed) >> 1;
    update(lz[p],be,mid,p<<1),update(lz[p],mid+1,ed,p<<1|1);
    lz[p] = 0;
}
// 初始化
void build_tree(int be=1,int ed=n,int p=1)
{
    if(be == ed)
    {
        t[p] = a[be];
        return ;
    }
    int mid = (be + ed) >> 1;
    build_tree(be,mid,p<<1), build_tree(mid+1,ed,p<<1|1);
    push_up(p);
}
// 修改
void put(int l,int r,int z,int be=1,int ed=n,int p=1)
{
    if(l <= be && ed <= r)
    {
        update(z,be,ed,p);
        return;
    }
    push_down(be,ed,p);
    int mid = (be + ed) >> 1;
    if(l <= mid)    put(l,r,z,be,mid,p<<1);
```

```
    if(mid+1 <= r)  put(l,r,z,mid+1,ed,p<<1|1);
    push_up(p);
}
// 查询
int query(int l,int r,int be=1,int ed=n,int p=1)
{
    if(l <= be && ed <= r)  return t[p];
    push_down(be,ed,p);
    int mid = (be + ed) >> 1, res = 0;
    if(l <= mid)    res += query(l,r,be,mid,p<<1);
    if(mid+1 <= r)  res += query(l,r,mid+1,ed,p<<1|1);
    return res;
}
```

区间RMQ

```
int n;
int a[N],t[N<<2],lz[N<<2];
// 更新节点
void update(int z,int be,int ed,int p)
{
    t[p] += z;
    lz[p] += z;
}
// 合并
void push_up(int p)
{
    t[p] = max(t[p<<1],t[p<<1|1]);
}
// 下放lazy_tag
void push_down(int be,int ed,int p)
{
    int mid = (be + ed) >> 1;
    update(lz[p],be,mid,p<<1),update(lz[p],mid+1,ed,p<<1|1);
    lz[p] = 0;
}
// 初始化
void build_tree(int be=1,int ed=n,int p=1)
{
    if(be == ed)
    {
        t[p] = a[be];
        return ;
    }
    int mid = (be + ed) >> 1;
    build_tree(be,mid,p<<1), build_tree(mid+1,ed,p<<1|1);
    push_up(p);
}
// 修改
void put(int l,int r,int z,int be=1,int ed=n,int p=1)
{
    if(l <= be && ed <= r)
    {
        update(z,be,ed,p);
        return ;
```

```
    }
    push_down(be,ed,p);
    int mid = (be + ed) >> 1;
    if(l <= mid)    put(l,r,z,be,mid,p<<1);
    if(mid+1 <= r)  put(l,r,z,mid+1,ed,p<<1|1);
    push_up(p);
}
// 查询
int query(int l,int r,int be=1,int ed=n,int p=1)
{
    if(l <= be && ed <= r)  return t[p];
    push_down(be,ed,p);
    int mid = (be + ed) >> 1, res = 0;
    if(l <= mid)    res = max(res,query(l,r,be,mid,p<<1));
    if(mid+1 <= r)  res = max(res,query(l,r,mid+1,ed,p<<1|1));
    return res;
}
```

单点修改线段树

单点修改不再需要lazy_tag

```
int n;
int a[N],t[N<<2],lz[N<<2];
// 更新节点
void update(int z,int be,int ed,int p)
{
    t[p] += (ed-be+1) * z;
    lz[p] += z;
}
// 合并
void push_up(int p)
{
    t[p] = t[p<<1] + t[p<<1|1];
}
// 下放lazy_tag
void push_down(int be,int ed,int p)
{
    int mid = (be + ed) >> 1;
    update(lz[p],be,mid,p<<1),update(lz[p],mid+1,ed,p<<1|1);
    lz[p] = 0;
}
// 初始化
void build_tree(int be=1,int ed=n,int p=1)
{
    if(be == ed)
    {
        t[p] = a[be];
        return ;
    }
    int mid = (be + ed) >> 1;
    build_tree(be,mid,p<<1), build_tree(mid+1,ed,p<<1|1);
    push_up(p);
}
// 修改
```

```
void put(int l,int r,int z,int be=1,int ed=n,int p=1)
{
    if(l <= be && ed <= r)
    {
        update(z,be,ed,p);
        return ;
    }
    push_down(be,ed,p);
    int mid = (be + ed) >> 1;
    if(l <= mid)    put(l,r,z,be,mid,p<<1);
    if(mid+1 <= r)  put(l,r,z,mid+1,ed,p<<1|1);
    push_up(p);
}
// 查询
int query(int l,int r,int be=1,int ed=n,int p=1)
{
    if(l <= be && ed <= r)  return t[p];
    push_down(be,ed,p);
    int mid = (be + ed) >> 1, res = 0;
    if(l <= mid)    res += query(l,r,be,mid,p<<1);
    if(mid+1 <= r)  res += query(l,r,mid+1,ed,p<<1|1);
    return res;
}
```

权值线段树

```
int n,q;
int t[N<<2];

void push_up(int p)
{
    t[p] = t[p<<1] + t[p<<1|1];
}
// 单点修改
void put(int k,int v,int be=1,int ed=n,int p=1)
{
    if(be == ed)
    {
        t[p] += v;
        return ;
    }
    int mid = (be+ed) >> 1;
    if(k <= mid)    put(k,v,be,mid,p<<1);
    else    put(k,v,mid+1,ed,p<<1|1);
    push_up(p);
}
// 查询[l,r]内共有多少个数
int query_cnt(int l,int r,int be=1,int ed=n,int p=1)
{
    if(l <= be && ed <= r)  return t[p];
    int mid = (be+ed) >> 1,cnt = 0;
    if(l <= mid)    cnt += query_cnt(l,r,be,mid,p<<1);
    if(mid+1 <= r)  cnt += query_cnt(l,r,mid+1,ed,p<<1|1);
    return cnt;
}
```

```
// 查询第 k 个数的值
int query_k(int k,int be=1,int ed=n,int p=1)
{
    if(be == ed)    return be;
    int mid = (be+ed) >> 1, lsum = t[p<<1];
    if(lsum >= k)   return query_k(k,be,mid,p<<1);
    else return query_k(k-lsum,mid+1,ed,p<<1|1);
}
```

多种 lazy_tag

乘法tag 和 加法tag

$\sum xa_i + b = x\sum a_i + len \times b$

```
struct node
{
    int sum,mul,add;
}t[N<<2];

int n,m;
int a[N];
// 更新节点
void update(int mul,int add,int be,int ed,int p)
{
    t[p].sum = t[p].sum*mul + (ed-be+1) * add;
    t[p].mul *= mul, t[p].add = t[p].add*mul+add;
}
// 合并
void push_up(int p)
{
    t[p].sum = t[p<<1].sum + t[p<<1|1].sum;
}
// 下放lazy_tag
void push_down(int be,int ed,int p)
{
    int mid = (be + ed) >> 1;
    update(t[p].mul,t[p].add,be,mid,p<<1),
update(t[p].mul,t[p].add,mid+1,ed,p<<1|1);
    t[p].mul = 1, t[p].add = 0;
}
// 初始化
void build_tree(int be=1,int ed=n,int p=1)
{
    t[p] = {0,1,0};
    if(be == ed)
    {
        t[p].sum = a[be];
        return;
    }
    int mid = (be + ed) >> 1;
    build_tree(be,mid,p<<1), build_tree(mid+1,ed,p<<1|1);
    push_up(p);
}
// 区间修改
```

```
void put(int l,int r,int add,int mul,int be=1,int ed=n,int p=1)
{
    if(l <= be && ed <= r)
    {
        update(mul,add,be,ed,p);
        return;
    }
    push_down(be,ed,p);
    int mid = (be + ed) >> 1;
    if(l <= mid)    put(l,r,add,mul,be,mid,p<<1);
    if(mid+1 <= r)  put(l,r,add,mul,mid+1,ed,p<<1|1);
    push_up(p);
}
// 区间查询
int query(int l,int r,int be=1,int ed=n,int p=1)
{
    if(l <= be && ed <= r)  return t[p].sum;
    push_down(be,ed,p);
    int mid = (be + ed) >> 1,res = 0;
    if(l <= mid)    res += query(l,r,be,mid,p<<1);
    if(mid+1 <= r)  res += query(l,r,mid+1,ed,p<<1|1);
    return res;
}
```

各种区间操作

加上等差数列

方案很多，如果只需要单点信息，则可以只维护差分，这里只用一棵线段树维护。

一次增加操作被断开，视为两次增减操作即可。

- $[l, x]$ 区间，$k + i \times d$
- $[x+1, r]$ 区间，$k + i \times (x + 1 - l) + i \times d$

```
struct node
{
    //
    int sum,k,d;
}t[N<<2];

int n,m;
int a[N];

void update(int k,int d,int be,int ed,int p)
{
    int len = ed-be+1;
    t[p].sum += len*k + len*(len-1)/2*d;
    t[p].k += k, t[p].d += d;
}

void push_up(int p)
{
    t[p].sum = t[p<<1].sum + t[p<<1|1].sum;
}
```

```cpp
void push_down(int be=1,int ed=n,int p=1)
{
    int mid = (be + ed) >> 1;
    update(t[p].k,t[p].d,be,mid,p<<1);
    update(t[p].k+t[p].d*(mid+1-be),t[p].d,mid+1,ed,p<<1|1);
    t[p].k = t[p].d = 0;
}

void build_tree(int be=1,int ed=n,int p=1)
{
    if(be == ed)
    {
        t[p].sum = a[be];
        return ;
    }
    int mid = (be + ed) >> 1;
    build_tree(be,mid,p<<1), build_tree(mid+1,ed,p<<1|1);
    push_up(p);
}

void put(int l,int r,int k,int d,int be=1,int ed=n,int p=1)
{
    if(l <= be && ed <= r)
    {
        update(k+(be-l)*d,d,be,ed,p);
        return;
    }
    push_down(be,ed,p);
    int mid = (be + ed) >> 1;
    if(l <= mid)    put(l,r,k,d,be,mid,p<<1);
    if(mid+1 <= r)  put(l,r,k,d,mid+1,ed,p<<1|1);
    push_up(p);
}

int query(int l,int r,int be=1,int ed=n,int p=1)
{
    if(l <= be && ed <= r)  return t[p].sum;
    push_down(be,ed,p);
    int mid = (be + ed) >> 1, res = 0;
    if(l <= mid)    res += query(l,r,be,mid,p<<1);
    if(mid+1 <= r)  res += query(l,r,mid+1,ed,p<<1|1);
    return res;
}

put(l,r,k,d);// 给 l,r 加上以 k 为初始值，方差为 d 的等差数列
query(l,r);// l,r 区间和
```

各种区间信息

区间平方和

$\sum(xa_i+y)=x^2\sum a_i^2+2xy\sum a_i+len\times y^2$

需要维护区间和

```
struct node
{
    // 区间和, 平方和, 加lz, 乘lz
    int sum,sq,add,mul;
}t[N<<2];

int n,m;
int a[N];
// 更新节点
void update(int mul,int add,int be,int ed,int p)
{
    t[p].sq = t[p].sq*mul*mul + t[p].sum*add*2 + (ed-be+1)*add*add;
    t[p].sum = t[p].sum*mul + (ed-be+1) * add;
    t[p].mul *= mul, t[p].add = t[p].add*mul+add;
}
// 合并
void push_up(int p)
{
    t[p].sum = t[p<<1].sum + t[p<<1|1].sum;
    t[p].sq = t[p<<1].sq + t[p<<1|1].sq;
}
// 下放lazy_tag
void push_down(int be,int ed,int p)
{
    int mid = (be + ed) >> 1;
    update(t[p].mul,t[p].add,be,mid,p<<1),
update(t[p].mul,t[p].add,mid+1,ed,p<<1|1);
    t[p].mul = 1, t[p].add = 0;
}
// 初始化
void build_tree(int be=1,int ed=n,int p=1)
{
    t[p] = {0,0,0,1};
    if(be == ed)
    {
        t[p] = {a[be],a[be]*a[be],0,1};
        return;
    }
    int mid = (be + ed) >> 1;
    build_tree(be,mid,p<<1), build_tree(mid+1,ed,p<<1|1);
    push_up(p);
}
// 区间修改
void put(int l,int r,int add,int mul,int be=1,int ed=n,int p=1)
{
    if(l <= be && ed <= r)
    {
```

```
        update(mul,add,be,ed,p);
        return;
    }
    push_down(be,ed,p);
    int mid = (be + ed) >> 1;
    if(l <= mid)    put(l,r,add,mul,be,mid,p<<1);
    if(mid+1 <= r)  put(l,r,add,mul,mid+1,ed,p<<1|1);
    push_up(p);
}
// 区间查询
int query(int l,int r,int be=1,int ed=n,int p=1)
{
    if(l <= be && ed <= r)  return t[p].sq;
    push_down(be,ed,p);
    int mid = (be + ed) >> 1,res = 0;
    if(l <= mid)    res += query(l,r,be,mid,p<<1);
    if(mid+1 <= r)  res += query(l,r,mid+1,ed,p<<1|1);
    return res;
}

// 区间乘
put(l,r,0,x);
// 区间加
put(l,r,x,1);
```

区间乘积和

求区间 l 到 r 之间两两之间数字的乘积和(例如：$2,3,4,5$两两之间乘积和为 $2\times3+2\times4+2\times5+3\times4+3\times5+4\times5$)

$$\sum\sum(xa_i+y)\times(xa_j+y)$$
$$=x^2\sum\sum a_ia_j+xy(len-1)\sum a_i+len(len-1)y^2/2$$

需要维护区间和

```
struct node
{
    // 区间和，区间乘积和，乘法lz，加法lz
    int sum,ab,mul,add;
}t[N<<2];

int n,m,P;
int a[N];
// 更新
void update(int mul,int add,int be,int ed,int p)
{
    int L = ed-be+1;
    // 切记先算乘积和，因为其依托于区间和
    t[p].ab = ((t[p].ab*mul*mul + (L-1)*mul*add*t[p].sum) + L*(L-1)/2*add*add);
    t[p].sum = (t[p].sum*mul + L*add);
    t[p].mul = t[p].mul*mul;
    t[p].add = (t[p].add*mul + add);
}
// 合并
void push_up(int p)
{
```

```
    t[p].sum = (t[p<<1].sum + t[p<<1|1].sum);
    t[p].ab = ((t[p<<1].ab + t[p<<1|1].ab) + t[p<<1].sum*t[p<<1|1].sum);
}
// 下放懒标记
void push_down(int be,int ed,int p)
{
    if(t[p].mul == 1 && t[p].add == 0)  return;
    int mid = (be + ed) >> 1;
    update(t[p].mul,t[p].add,be,mid,p<<1),
update(t[p].mul,t[p].add,mid+1,ed,p<<1|1);
    t[p].mul = 1,t[p].add = 0;
}
// 初始化
void build_tree(int be=1,int ed=n,int p=1)
{
    t[p] = {a[be],0,1,0};
    if(be == ed)
    {
        t[p].sum = a[be];
        return ;
    }
    int mid = (be + ed) >> 1;
    build_tree(be,mid,p<<1), build_tree(mid+1,ed,p<<1|1);
    push_up(p);
}
// 区间修改
void put(int l,int r,int mul,int add,int be=1,int ed=n,int p=1)
{
    if(l <= be && ed <= r)
    {
        update(mul,add,be,ed,p);
        return;
    }
    push_down(be,ed,p);
    int mid = (be + ed) >> 1;
    if(l <= mid)    put(l,r,mul,add,be,mid,p<<1);
    if(mid+1 <= r)  put(l,r,mul,add,mid+1,ed,p<<1|1);
    push_up(p);
}
// 区间查询
PII query(int l,int r,int be=1,int ed=n,int p=1)
{
    if(l <= be && ed <= r)  return {t[p].sum,t[p].ab};
    push_down(be,ed,p);
    int mid = (be + ed) >> 1;
    PII res = {0,0};
    if(l <= mid)
    {
        res = query(l,r,be,mid,p<<1);
        if(mid+1 <= r)
        {
            PII tmp = query(l,r,mid+1,ed,p<<1|1);
            res.Y = (res.Y+tmp.Y) + res.X*tmp.X;
            res.X = res.X+tmp.X;
        }
    }
```

```
    else if(mid+1 <= r) res = query(l,r,mid+1,ed,p<<1|1);
    return res;
}

put(l,r,1,v);// 区间加
put(l,r,v,0);// 区间乘
```

区间信息性质

开方性质

一个 $\leq 10^9$ 的数，最多**五次**开方到 1

> 就是 $\leq 10^18$ 的数，也只需要 **六次**

题面:
第一行一个整数 n，代表数列中数的个数。

第二行 n 个正整数，表示初始状态下数列中的数。

第三行一个整数 m，表示有 m 次操作。

接下来 m 行每行三个整数 `k l r`。

- $k = 0$ 表示给 $[l, r]$ 中的每个数开平方（下取整）。
- $k = 1$ 表示询问 $[l, r]$ 中各个数的和。

思路:
一个 $\geq$ 的数, 最多 5 次开方变成 1, 所以最多对所有数 5 次单点修改. 那么只需要维护**单点修改**和**区间查询**. 并且维护区间最值, 最大值为 1, 则无需再修改.
代码:

```
#include<bits/stdc++.h>
#define Start cin.tie(0), cout.tie(0), ios::sync_with_stdio(false)
#define PII pair<int,int>
#define x first
#define y second
#define ull unsigned long long
#define int long long
using namespace std;

const int M = 1000000007;
const int N = 1e5 + 10;

int n,m;
vector<int> a(N),t(N<<2),tmx(N<<2);

void push_up(int p)
{
    t[p] = t[p<<1] + t[p<<1|1];
    tmx[p] = max(tmx[p<<1],tmx[p<<1|1]);
}

void build_tree(int be=1,int ed=n,int p=1)
{
    if(be == ed)
```

```
    {
        t[p] = a[be];
        tmx[p] = a[be];
        return ;
    }
    int mid = (be + ed) >> 1;
    build_tree(be,mid,p<<1),build_tree(mid+1,ed,p<<1|1);
    push_up(p);
}

void put(int l,int r,int be=1,int ed=n,int p=1)
{
    if(be == ed)
    {
        t[p] = sqrt(t[p]);
        tmx[p] = sqrt(tmx[p]);
        return ;
    }
    int mid = (be + ed) >> 1;
    if(l <= mid && tmx[p] > 1)  put(l,r,be,mid,p<<1);
    if(mid+1 <= r && tmx[p] > 1)    put(l,r,mid+1,ed,p<<1|1);
    push_up(p);
}

int query(int l,int r,int be=1,int ed=n,int p=1)
{
    if(l <= be && ed <= r)  return t[p];
    int mid = (be + ed) >> 1, res = 0;
    if(l <= mid)    res += query(l,r,be,mid,p<<1);
    if(mid+1 <= r)  res += query(l,r,mid+1,ed,p<<1|1);
    return res;
}

void func(void);

signed main(void)
{
    Start;
    int _ = 1;
    // cin >> _;
    while(_--)  func();
    return 0;
}

void func(void)
{
    cin >> n;
    for(int i=1;i<=n;++i)   cin >> a[i];
    build_tree();
    cin >> m;
    while(m--)
    {
        bool op;    cin >> op;
        int l,r;    cin >> l >> r;
        if(l > r)   swap(l,r);
        if(op)  cout << query(l,r) << '\n';
```

```
        else    put(l,r);// 一次结束当然比循环一次要快了
    }
}
```

开多棵线段树

此处是bitset优化
题面:
色板长度为 L，L 是一个正整数，所以我们可以均匀地将它划分成 L 块 1 厘米长的小方格。并从左到右标记为 $1, 2, \dots L$。

现在色板上只有一个颜色，老师告诉阿宝在色板上只能做两件事：

1. `C A B C` 指在 A 到 B 号方格中涂上颜色 C。
2. `P A B` 指老师的提问：A 到 B 号方格中有几种颜色。

学校的颜料盒中一共有 T 种颜料。为简便起见，我们把他们标记为 $1, 2, \dots T$. 开始时色板上原有的颜色就为 1 号色。 面对如此复杂的问题，阿宝向你求助，你能帮助他吗？

输入格式:
第一行有3个整数 $L(1 \le L \le 10^5), T(1 \le T \le 30)$和$O(1 \le O \le 10^5)$。 在这里 O 表示事件数。

接下来 O 行, 每行以 `C A B C` 或 `P A B` 得形式表示所要做的事情（这里 A, B, C 为整数, 可能 $A > B$，这样的话需要你交换 A 和 B）。
输出格式:
对于老师的提问，做出相应的回答。每行一个整数。

代码:

```
#include<bits/stdc++.h>
#define Start cin.tie(0), cout.tie(0), ios::sync_with_stdio(false)
#define PII pair<int,int>
#define x first
#define y second
#define ull unsigned long long
#define ll long long
using namespace std;

const int M = 1000000007;
const int N = 1e5 + 10;

int n,k,q;
vector<bitset<1000>> t(N<<2);
vector<int> lz(N<<2);
// vector<vector<bool>> t(40,vector<bool>(N<<2));
// vector<int> lz(N<<2);

void update(int col,int be,int ed,int p)
{
    // cout << col << ' ' << be << ' ' << ed << '\n';
    t[p].reset();   t[p].set(col);
    lz[p] = col;
}

void push_up(int p)
```

```
{
    int lp = p<<1, rp = p<<1|1;
    t[p] = t[p<<1] | t[p<<1|1];
}

void push_down(int be,int ed,int p)
{
    if(lz[p] == 0)  return ;
    int mid = (be+ed) >> 1;
    update(lz[p],be,mid,p<<1);
    update(lz[p],mid+1,ed,p<<1|1);
    lz[p] = 0;
}

void build_tree(int be=1,int ed=n,int p=1)
{
    if(be == ed)
    {
        t[p].set(1);
        return;
    }
    int mid = (be+ed) >> 1;
    build_tree(be,mid,p<<1);
    build_tree(mid+1,ed,p<<1|1);
    push_up(p);
}

void put(int l,int r,int col,int be=1,int ed=n,int p=1)
{
    if(l <= be && ed <= r)
    {
        update(col,be,ed,p);
        return ;
    }
    push_down(be,ed,p);
    int mid = (be+ed) >> 1;
    if(l <= mid)    put(l,r,col,be,mid,p<<1);
    if(mid+1 <= r)  put(l,r,col,mid+1,ed,p<<1|1);
    push_up(p);
    // cout << t[col][p] << ' ';
}

bitset<1000> query(int l,int r,int be=1,int ed=n,int p=1)
{
    bitset<1000> res = 0;
    if(l <= be && ed <= r)
    {
        return t[p];
    }
    push_down(be,ed,p);
    int mid = (be+ed) >> 1;
    if(l <= mid)    res |= query(l,r,be,mid,p<<1);
    if(mid+1 <= r)  res |= query(l,r,mid+1,ed,p<<1|1);
    return res;
}
```

```
void func(void);

signed main(void)
{
    Start;
    int _  = 1;
    while(_--)  func();
    return 0;
}

void func(void)
{
    cin >> n >> k >> q;
    build_tree();
    while(q--)
    {
        char op;
        int l,r;
        cin >> op >> l >> r;
        if(l > r)    swap(l,r);
        if(op == 'C')
        {
            int col;    cin >> col;
            put(l,r,col);
        }
        else
        {
            int ans = 0;
            cout << query(l,r).count() << '\n';
            // cout << ans << '\n';
        }
    }
}
```

可持久化线段树

主席树

主席树是可持久化的**权值**线段树

```
struct node
{
    int cnt, ls, rs;
};

int n,q,idx,mx;
vector<int> a(N), dis, rt(N);
vector<node> t(N<<5);
int get(int x)
{
    return (lpwer_bound(dis.begin(),dis.end(),x) - dis.begin()+1);
}

void insert(int &p,int pre,int val,int be=1,int ed=mx)
{
```

```
    p = ++ idx;
    t[p] = t[pre];
    t[p].cnt ++;
    if(be == ed)    return ;
    int mid = (be+ed) >> 1;
    if(val <= mid)  insert(t[p].ls,t[pre].ls,val,be,mid);
    else    insert(t[p].rs,t[pre].rs,val,mid+1,ed);
}

int query(int lp,int rp,int k,int be=1,int ed=mx)
{
    if(be == ed)    return be;
    int mid = (be + ed) >> 1, lcnt = t[t[rp].ls].cnt - t[t[lp].ls].cnt;
    if(k <= lcnt)   return query(t[lp].ls,t[rp].ls,k,be,mid);
    else    return query(t[lp].rs,t[rp].rs,k-lcnt,mid+1,ed);

}

void func(void)
{
    cin >> n >> q;
    for(int i=1;i<=n;++i)   cin >> a[i];
    for(int i=1;i<=n;++i)   dis.push_back(a[i]);
    sort(dis.begin(),dis.end());
    dis.erase(unique(dis.begin(),dis.end()),dis.end());
    mx = dis.size();
    for(int i=1;i<=n;++i)   insert(rt[i],rt[i-1],get(a[i]));
    while(q--)
    {
        int l,r,k;
        cin >> l >> r >> k;
        cout << dis[query(rt[l-1],rt[r],k)-1] << '\n';
    }
}
```

树链剖分

轻重链剖分

处理：

求重链 $O(n)$：

- 将子树中总结点最多的视为重儿子，因为轻链的大小 $\leq sum/2$，所以每次走轻儿子等同于抛弃 $sum/2$ 的点，那么可以保证最终复杂度 $\leq n\log n$

链分解 $O(n)$：

- 重儿子继承父节点值，而轻儿子以本身为链头作为重链继续剖分。最终得到若干重链。

操作：

链操作 O ($\log n\times$ 区间修改)：

- dfs序保证每个子树在 `dfn` 上连续，而重链剖分保证每条重链上的点在 `dfn` 上连续。这样就可以对链区间修改

- 将重链缩为一点后，树的深度 $\le \log n$，那么可以用类似暴力的 lca 求两个重链树的父链，每次会操作所在重链头到该节点的数据。最终两点走到同一链，在操作两点见的数据即可。

子树操作 O(区间修改)

- 利用 `dfn`，子树管辖区间是 $[dfn_x, dfn_x + size_x - 1]$

```
int idx;// dfs序辅助变量
vector<int> v[N];
int a[N],ta[N];// 节点值 映射dfn值
int hson[N],top[N],sz[N];// 重儿子 链头 子树大小
int fa[N],dfn[N],dep[N];// 父节点 dfs序 深度

void dfs_size(int p,int lp)// 求重链
{
    fa[p] = lp;
    dep[p] = dep[lp] + 1;
    sz[p] = 1;
    hson[p] = 0;
    for(auto &i : v[p])
    {
        if(i == lp) continue;
        dfs_size(i,p);
        sz[p] += sz[i];
        if(sz[hson[p]] < sz[i]) hson[p] = i;
    }
}

void dfs_chain(int p,int tp)// 链分解
{
    top[p] = tp;
    dfn[p] = ++ idx;
    ta[idx] = a[p];
    if(!hson[p])    return;
    dfs_chain(hson[p],tp);
    for(auto &i : v[p])
    {
        if(i != fa[p] && i != hson[p])  dfs_chain(i,i);
    }
}

void put_path(int x,int y)// 操作 x - y 链
{
    while(top[x] != top[y])
    {
        if(dep[top[x]] < dep[top[y]])   swap(x,y);
        /*
        put(dfn[top[x]],dfn[x]);
        */
        x = fa[top[x]];
    }
    if(dep[x] > dep[y]) swap(x,y);
    // put(dfn[x],dfn[y],z);
}

void put_tree(int x)// 操作 x 的所有子树
```

```
{
    put(dfn[x],dfn[x]+sz[x]-1);
}
```

dsu on tree

理用重链剖分处理离线**子树信息查询**问题

> 类似莫队

处理无法轻松合并的信息

> 线段树做不到或者很麻烦的，比如区间内数的种类

因为不同子树的 `dfn` 只存在包含或者相离，那么父节点是可以利用一个子节点的信息或者说信息数组使用，也就是继承其信息。

> 因为信息比较复杂，无法简单合并，那么多个点的信息只能用一个了。

根据重链剖分的分析，我们必然是使用重儿子的信息，而轻儿子的信息必须清除重新统计。

本质是逆序的重链剖分。

```
int n,idx,
vector<int> v[N];
int ans[N];// 子树答案
int fa[N],sz[N],dfn[N],id[N],hson[N];
int res;// 当前子树答案
// int tmp[N]; 辅助数组

void dfs(int p,int lp)// 求重儿子
{
    fa[p] = lp;
    sz[p] = 1;
    hson[p] = 0;
    // 因为不需要链信息，所以在第一次就可以直接求dfn
    dfn[p] = ++ idx;
    id[idx] = p;
    for(auto &i : v[p])
    {
        if(i == lp) continue;
        dfs(i,p);
        sz[p] += sz[i];
        if(sz[hson[p]] < sz[i]) hson[p] = i;
    }
}

void put(int p)// 操作
{
    for(int i=dfn[p];i<dfn[p]+sz[p];++i)
    {
        if(id[i] == hson[p])
        {
            i = dfn[hson[p]]+sz[hson[p]]-1;
            // 这里如果直接等于 dfn[hson[p]]+sz[hson[p]]，可能恰好超过 dfn[p]，
            // 加上特判的码量差不多，就没必要了。
            continue;
```

```
        }
        // 辅助数组操作
        tmp[id[i]] ++;
        res ++;
    }
    return res;
}

void init(int p)// 清空
{
    for(int i=dfn[p];i<dfn[p]+sz[p];++i)    tmp[id[i]] --;
}

void dsu(int p,bool del)// 求子树信息
{
    for(auto &i : v[p])
    {
        if(i != fa[p] && i != hson[p])  dsu(i,true);
    }
    if(hson[p]) dsu(hson[p],false);
    put(p);
    ans[p] = res;
    if(del)
    {
        init(p);
    }
}
```

珂朵莉树

珂朵莉树（Chtholly Tree），又名老司机树 ODT（Old Driver Tree）。起源自 CF896C。

这个名称指代的是一种「使用平衡树（`set`、`map` 等）或链表（`list`、手写链表等）维护颜色段均摊」的技巧，而不是一种特定的数据结构。其核心思想是将值相同的一段区间合并成一个结点处理。相较于传统的线段树等数据结构，对于含有区间覆盖的操作的问题，珂朵莉树可以更加方便地维护每个被覆盖区间的值。

`set`

`set` 维护各个区间的 l,r 和值（颜色）d

```
// 结点
struct node
{
    int l,r,d;
    bool operator < (const node &i) const
    {
        // 因为一个所有元素组成所有区间，所以不会有重复 l
        return l < i.l;
    }
};

// 将一个区间[l,r]，分割为 [l,x]，[x+1,r]，并返回后者指针
auto split(int x) //
{
```

```
    if(x == n+1)    return st.end(); // assign 最后可能取 n+1
    auto p = st.lower_bound({x,0,0}); // 找 l >= x 值
    if(p != st.end() && p->l == x)  return p; // l = x 情况
    // l > x 情况，p-- 后 l < x
    p --;
    auto &[l,r,d] = *p;
    st.erase(p);
    st.insert({l,x-1,d});// 放回左区间
    return (st.insert({z,r,x-l+d})).first;// insert 返回值pair<T,bool>
}

// 对一段区间进行赋值
void assign(int l,int r,int v)
{
    // 取出两端区间，分成两段并得到下标
    auto pr = split(r+1);   // 先 l 报错
    auto pl = split(l);
    /* 如果要遍历期间区间，用此循环，恰好不访问 pr
    for(auto p=pl;i!=pr;++p) func() */
    st.erase(pl,pr);// erase性质，删除[l,r)
    st.insert({l,r,v});
}
// 新建set
set<node> st;
```

`map`

由于珂朵莉树存储的区间是连续的，我们不一定要记下右端点是什么。不妨使用一个 `map<int, int> mp;` 存储所有区间，其键维护左端点，其值维护其对应的左端点到下一个左端点之前的值。

初始化时，如题目要求维护位置 1 到 n 的信息，则调用 `mp[1] = -1`， `mp[n + 1] = -1` 表示将 `[1,n+1)` 即 `[1, n]` 都设为特殊值 -1， `[n+1, +\infty)` 这个区间当作哨兵使用，也可以对它进行初始化。

```
void split(int x)
{
    // 找到左端点小于等于 x 的区间。
    auto p = prev(mp.upper_bound(x));  // prev 找到该节点上一个位置的迭代器
    mp[x] = p->second;  // 设立新的区间，并将上一个区间储存的值复制给本区间。
}

void assign(int l, int r, int v)
{
    split(l);
    split(r);// 注意，这里的r是区间右端点+1
    auto p = mp.find(l);
    while(p->first != r)  p = mp.erase(it);// erase会返回下个元素的迭代器
    /* 如果要遍历期间区间，用此循环
     while(it->first != r)  p = next(it)*/
    mp[l] = v;
}
```