# 数据结构

## 并查集

就是构建n棵树表示n个集合，两个点在同一集合内当且仅当两点的祖先相同。

* 构建并查集

让每个点的祖先等于自己，于是每个集合只包括一个点。

struct Node {  
 int fa, n; //保存自己的父亲，自己有多少个孩子  
}node[MAXN];  
//...  
for (int i = 0; i < n; ++i) node[i].fa = i, node[i].n = 1;

* 查询祖先（路径压缩）

这里要用到一个重要的优化——路径压缩，即让通往祖先的路径上的每一个节点都直接指向祖先。这里介绍递归法的实现方法。

int find(int x) {  
 int ret;  
 if (node[x].fa == x) ret = x;  
 else ret = find(node[x].fa);  
 return node[x].fa = ret; //路径压缩，直接让x指向自己的祖先  
}

使用路径压缩优化，复杂度期望值为O(α(N))，最坏情况下为O(logN)。对于Kruskal算法求解最小生成树，单独使用路径压缩已经足够了，因为Kruskal算法要对边进行排序，复杂度为O(logM)。

* 判断两点是否在一个集合

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

* 合并（按秩合并）

按秩合并的基本思想是将深度小的树的祖先的父亲变更为深度大的树的祖先。这里为了实现简便，把将高度替换为孩子个数。（其实是我不会）

void merge(int x, int y) {  
 int xf = find(x), yf = find(y);  
 if (node[xf].n > node[yf].n) swap(xf, yf); //要把x并到y里面去，所以x小  
 if (xf != yf) {  
 node[xf].fa = yf;  
 node[yf].n += node[xf].n;  
 //只有根节点的n是有意义的，别的地方错了也无所谓  
 }  
}

运用了按秩合并和路径压缩后并查集的复杂度非常优秀，可以认为是O(1)的。

### 并查集维护序列的连通性

**洛谷 P2391 白雪皑皑**：一个常用的套路是，用 $ fa\_x $ 表示 $ x $ 后第一个可以操作的点。

下面的代码可以顺序遍历 $ [l, r] $ 中所有没被修改的点，并将其修改。有一点需要注意的是，初始化时务必 fa[n + 1] = n + 1，因为 n 要合并到 n + 1。

while (l <= r) {  
 int f = find\_fa(l);  
 if (l == f) {  
 ans[l] = i;  
 fa[l] = l + 1;  
 }  
 l = f;  
}

更进一步，**洛谷 P4145 上帝造题的七分钟 2 / 花神游历各国**：需要实现区间开放，并且要维护区间和。

思考一件事，一个数开方到 $ 1 $ 的次数是有限的，并且 $ 1 $ 和 $ 0 $ 开方不会影响区间和，于是可以树状数组+并查集暴力维护， $ fa\_x$ 维护 $ x $ 后第一个 $ > 1 $ 的位置。

while (l <= r) {  
 ll sq = sqrt(a[l]);  
 tr.add(l, sq - a[l]);  
 a[l] = sq;  
  
 if (a[l] <= 1) {  
 fa[l] = l + 1;  
 l = find\_fa(fa[l]);  
 } else {  
 l = l + 1;  
 }  
}

综上所述，第一份代码给出了**只修改一次**的版本，第二份代码给出了**修改多次**的版本。

## 树状数组

struct binary\_indexed\_tree{  
 int n;  
 ll c[N] = {0};  
 int lowbit(int x) {  
 // x 的二进制中，最低位的 1 以及后面所有 0 组成的数。  
 // lowbit(0b01011000) == 0b00001000  
 // ~~~~^~~~  
 // lowbit(0b01110010) == 0b00000010  
 // ~~~~~~^~  
 return x & -x;  
 }  
  
 ll get\_sum(int x) { // a[1]..a[x]的和  
 ll ans = 0;  
 while (x > 0) {  
 ans = ans + c[x];  
 x = x - lowbit(x);  
 }  
 return ans;  
 }  
  
 void add(int x, ll k) { // 在x的下标加上数值k  
 while (x <= n) { // 不能越界  
 c[x] = c[x] + k;  
 x = x + lowbit(x);  
 }  
 }  
}tr;

本质上维护的是前缀和，实现单点修改区间（前缀和）查询。

当然也可以看做维护的是差分数组，修改首尾，再求前缀和，实现单点查询区间修改。

当涉及到带取模的运算时，树状数组或许无法胜任。

## 线段树

* 几个操作以及一些定义

#define ls(xx) ((xx) << 1) + 1  
#define rs(xx) ((xx) << 1) + 2 //根节点编号从0开始  
  
long long tree[666666], tag\_add[666666], tag\_mul[666666], a[666666];  
  
void build(int l, int r, int node);  
void update\_add(int l, int r, int node, int nl, int nr, long long val, int p);  
void update\_mul(int l, int r, int node, int nl, int nr, long long val, int p);  
long long query(int l, int r, int node, int nl, int nr, int p);

* push\_up以及push\_down

void push\_up(int node) { //向上更新节点  
 tree[node] = tree[ls(node)] + tree[rs(node)];  
}  
  
//更新标记  
void fill(int l, int r, int node, long long add, long long mul, int p) {  
 tree[node] \*= mul; tree[node] %= p; tree[node] += (r - l + 1) \* add; tree[node] %= p;  
 tag\_add[node] \*= mul; tag\_add[node] %= p; tag\_add[node] += add; tag\_add[node] %= p;   
 tag\_mul[node] \*= mul; tag\_mul[node] %= p;  
}  
  
//向下释放标记  
void push\_down(int l, int r, int node, int p) {  
 int mid = (l + r) >> 1;  
 fill(l, mid, ls(node), tag\_add[node], tag\_mul[node], p);  
 fill(mid + 1, r, rs(node), tag\_add[node], tag\_mul[node], p);  
 tag\_add[node] = 0; tag\_mul[node] = 1;  
}

* 建树（build）

void build(int l, int r, int node) {  
 int mid = (l + r) >> 1;  
 tag\_mul[node] = 1;  
 if (l == r) {  
 tree[node] = a[l]; return ;  
 }  
 build(l, mid, ls(node));  
 build(mid + 1, r, rs(node));  
 push\_up(node);  
}

建树其实就是把每个节点的值存成区间和，如果该节点区间长度是1，那么区间和就是数组的值。

* 区间加（update\_add）

void update\_add(int l, int r, int node, int nl, int nr, long long val, int p) {  
 int mid = (l + r) >> 1;  
 if (nl <= l && r <= nr) {  
 tree[node] += (r - l + 1) \* val; tree[node] %= p;  
 tag\_add[node] += val; tag\_add[node] %= p; return ;  
 }  
 push\_down(l, r, node, p);  
 if (nl <= mid) update\_add(l, mid, ls(node), nl, nr, val, p);  
 if (mid < nr) update\_add(mid + 1, r, rs(node), nl, nr, val, p);  
 push\_up(node); tree[node] %= p;  
}

* 区间乘（update\_mul）

void update\_mul(int l, int r, int node, int nl, int nr, long long val, int p) {  
 int mid = (l + r) >> 1;  
 if (nl <= l && r <= nr) {  
 tree[node] \*= val; tree[node] %= p;  
 tag\_add[node] \*= val; tag\_add[node] %= p;  
 tag\_mul[node] \*= val; tag\_mul[node] %= p;  
 return ;  
 }  
 push\_down(l, r, node, p);  
 if (nl <= mid) update\_mul(l, mid, ls(node), nl, nr, val, p);  
 if (mid < nr) update\_mul(mid + 1, r, rs(node), nl, nr, val, p);  
 push\_up(node); tree[node] %= p;  
}

区间加和区间乘的操作是类似的，基本可以分为几个步骤：

1. 如果操作区间完全包含了当前区间，直接打上tag并对区间值进行修改。需要注意的是，乘法操作需要修改加法tag；
2. 下放标记（push\_down）；
3. 分别判断操作区间和当前区间的左右部分是否有交集，若有，递归操作；
4. 更新节点（push\_up）。

* 查询

long long query(int l, int r, int node, int nl, int nr, int p) {  
 int mid = (l + r) >> 1;  
 int ret = 0;  
 if (nl <= l && r <= nr) return tree[node];  
 push\_down(l, r, node, p);  
 if (nl <= mid) ret += query(l, mid, ls(node), nl, nr, p), ret %= p;  
 if (mid < nr) ret += query(mid + 1, r, rs(node), nl, nr, p), ret %= p;  
 return ret;  
}

对于线段树的操作，有很多细节。下面针对push\_down和push\_up两个操作说一些有共性的地方。

1. 如果要从某个节点访问其左右儿子，之前一定要下放标记；
2. 如果更新了左右儿子的节点值，一定要push\_up；
3. 不要忘记取模！

* 应用：求逆序对（常数太大，用树状数组好一些）

思路：先对于输入的数离散化，需要特殊处理值相等的情况。如果是严格的逆序对的话，那相等的值不会产生逆序对，下面给出的代码就是这个版本，同时也给出了**只有加法的线段树**。

一定要注意**add**和**query**的参数。

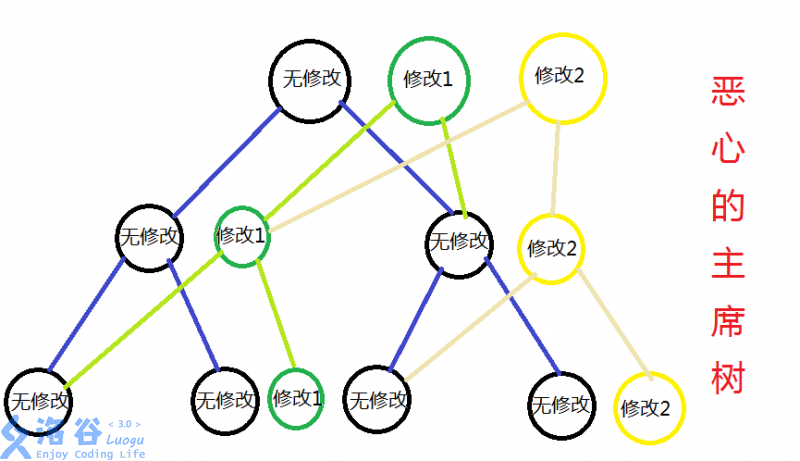
#include <iostream>  
#include <unordered\_map>  
#include <set>  
using namespace std;  
#define MAXN 666666  
#define P 19260817  
#define ls(x) ((x)<<1)  
#define rs(x) (((x)<<1)+1)  
#define mod(x) ((x))  
#define len(l, r) ((r)-(l)+1)  
  
unordered\_map<int, int> mp;  
set<int> s;  
int a[MAXN], aa[MAXN];  
long long su[MAXN\*2], tag[MAXN\*2];  
  
void push\_up(int now) {su[now] = mod(su[ls(now)] + su[rs(now)]); }  
  
void full(int now, int nl, int nr, long long k) {  
 tag[now] = mod(tag[now] + k);  
 su[now] = mod(su[now] + len(nl, nr) \* k);  
}  
  
void push\_down(int now, int nl, int nr) {  
 int mid = (nl + nr) >> 1;  
 full(ls(now), nl, mid, tag[now]);  
 full(rs(now), mid + 1, nr, tag[now]);  
 tag[now] = 0;  
}  
/\* 不用build了  
void build(int now, int nl, int nr) {  
 int mid = (nl + nr) >> 1;  
 if (nl == nr) {su[now] = a[nl]; return; }  
 build(ls(now), nl, mid); build(rs(now), mid + 1, nr);  
 push\_up(now);  
}  
\*/  
void add(int now, int nl, int nr, int l, int r, long long k) {  
 int mid = (nl + nr) >> 1;  
 if (l <= nl && nr <= r) { //完全包含  
 su[now] = mod(su[now] + len(nl, nr) \* k);  
 tag[now] = mod(tag[now] + k); return;  
 }  
 push\_down(now, nl, nr);  
 if (l <= mid) add(ls(now), nl, mid, l, r, k);  
 if (mid + 1 <= r) add(rs(now), mid + 1, nr, l, r, k);  
 push\_up(now);  
}  
  
long long query(int now, int nl, int nr, int l, int r) {  
 int mid = (nl + nr) >> 1; long long ret = 0;  
 if (l <= nl && nr <= r) return su[now];  
 push\_down(now, nl, nr);  
 if (l <= mid) ret = mod(ret + query(ls(now), nl, mid, l, r));  
 if (mid + 1 <= r) ret = mod(ret + query(rs(now), mid + 1, nr, l, r));  
 return ret;  
}  
  
int get\_int(void) {  
 // 快读就省略了  
}  
  
int main() {  
 #ifdef LOCAL  
 freopen("in.txt", "r", stdin);  
 freopen("out.txt", "w", stdout);  
 #endif // LOCAL  
 long long n, ans = 0;  
 // cin >> n;  
 n = get\_int();  
 for (int i = 0; i < n; ++i) {  
 a[i] = get\_int();  
 s.insert(a[i]);  
 }  
 int cnt = 1;  
 for (auto it = s.begin(); it != s.end(); ++it) {  
 mp[\*it] = cnt++; // 离散化，相同的数值离散化成同一值  
 }  
 for (int i = 0; i < n; ++i) {  
 int t = mp[a[i]];  
 ans += query(1, 1, n + 1, t + 1, n + 1);  
 add(1, 1, n + 1, t, t, 1); // 多维护一位  
 }  
 cout << ans;  
 return 0;  
}

由于只支持加法的线段树更加常用，在这里再贴个板子：

struct segment\_tree{  
 #define ls(x) ((x)<<1)  
 #define rs(x) (((x)<<1)+1)  
 #define len(l, r) ((r)-(l)+1)  
 #define mod(x, p) (((x)%(p)+(p))%(p))  
   
 ll su[MAXN\*2], tag[MAXN\*2], a[MAXN\*2];  
  
 void push\_up(int now, int p) {su[now] = mod(su[ls(now)] + su[rs(now)], p); }  
  
 void full(int now, int nl, int nr, ll k, int p) {  
 tag[now] = mod(tag[now] + k, p);  
 su[now] = mod(su[now] + len(nl, nr) \* k, p);  
 }  
  
 void push\_down(int now, int nl, int nr, int p) {  
 int mid = (nl + nr) >> 1;  
 full(ls(now), nl, mid, tag[now], p);  
 full(rs(now), mid + 1, nr, tag[now], p);  
 tag[now] = 0;  
 }  
  
 void build(int now, int nl, int nr, int p) {  
 int mid = (nl + nr) >> 1;  
 if (nl == nr) {su[now] = a[nl]; return; }  
 build(ls(now), nl, mid, p); build(rs(now), mid + 1, nr, p);  
 push\_up(now, p);  
 }  
  
 void add(int now, int nl, int nr, int l, int r, ll k, int p) {  
 int mid = (nl + nr) >> 1;  
 if (l <= nl && nr <= r) { //完全包含  
 su[now] = mod(su[now] + len(nl, nr) \* k, p);  
 tag[now] = mod(tag[now] + k, p); return;  
 }  
 push\_down(now, nl, nr, p);  
 if (l <= mid) add(ls(now), nl, mid, l, r, k, p);  
 if (mid + 1 <= r) add(rs(now), mid + 1, nr, l, r, k, p);  
 push\_up(now, p);  
 }  
  
 ll query(int now, int nl, int nr, int l, int r, int p) {  
 int mid = (nl + nr) >> 1; ll ret = 0;  
 if (l <= nl && nr <= r) return su[now];  
 push\_down(now, nl, nr, p);  
 if (l <= mid) ret = mod(ret + query(ls(now), nl, mid, l, r, p), p);  
 if (mid + 1 <= r) ret = mod(ret + query(rs(now), mid + 1, nr, l, r, p), p);  
 return ret;  
 }  
};

## 可持久化线段树

**主席树**又称可持久化权值线段树。



可持久化线段树

### 可持久化数组

支持：

* 查询某一历史版本单点
* 修改某一历史版本单点

所有的修改和查询都会新建一个版本。

struct HJT\_tree {  
 int n, tot = 0, ver\_cnt = 0, root[N<<5];  
 struct Node{int ls, rs, val; }node[N<<5];  
  
 int create\_node(int now) {  
 node[++tot] = node[now]; // 传递信息  
 return tot;  
 }  
  
 void init(int nn) {  
 n = nn;  
 root[0] = build(1, 1, n);  
 }  
  
 int build(int now, int nl, int nr) {  
 now = ++tot;  
 if (nl == nr) {  
 node[now].val = a[nl];  
 return tot;  
 }  
 int mid = (nl + nr) / 2;  
 node[now].ls = build(now, nl, mid);  
 node[now].rs = build(now, mid + 1, nr);  
 return now;  
 }  
  
 int query(int pos, int ver) {  
 root[++ver\_cnt] = root[ver]; // 新建版本  
 return query\_work(root[ver], 1, n, pos);  
 }  
  
 int query\_work(int now, int nl, int nr, int pos) {  
 if (nl == nr) {  
 return node[now].val;  
 }  
 int mid = (nl + nr) / 2;  
 if (pos <= mid) {  
 return query\_work(node[now].ls, nl, mid, pos);  
 } else {  
 return query\_work(node[now].rs, mid + 1, nr, pos);  
 }  
 }  
  
 void update(int pos, int val, int ver) {  
 root[++ver\_cnt] = update\_work(root[ver], 1, n, pos, val); // 新建版本  
 }  
  
 int update\_work(int now, int nl, int nr, int pos, int val) {  
 now = create\_node(now);  
 if (nl == nr) {  
 node[now].val = val;  
 } else {  
 int mid = (nl + nr) / 2;  
 if (pos <= mid) {  
 node[now].ls = update\_work(node[now].ls, nl, mid, pos, val);  
 } else {  
 node[now].rs = update\_work(node[now].rs, mid + 1, nr, pos, val);  
 }  
 }  
 return now;  
 }  
}tr;

### 静态区间第 k 小

对于原序列的每一个前缀建立出一棵线段树维护值域上每个数出现的次数，则其树是可减的。——HJT

对区间 $ [1, n] $ 查找第 $ k $ 小是容易的，在线段树上只需要看左右子树权值和即可判断答案在哪边。

考虑使用主席树。对于每次对于 $ a\_i $ 的插入操作，我们都可以保存相应的版本，那区间 $ [x, y] $ 对应的信息可以由 $ver\_y - ver\_{x - 1} $ 得到，于是就可以用主席树维护了。

struct Kth\_tree {  
 int n, tot = 0, root[N];  
 struct Node{int ls, rs, sum; }node[N<<5];  
  
public:  
 void init(int nn) {  
 n = nn;  
  
 int cnt = 0;  
 set<int> s;  
 for (int i = 1; i <= n; ++i) {  
 s.insert(a[i]);  
 }  
 for (auto c : s) {  
 mp[c] = ++cnt;  
 origin[cnt] = c; // 离散化  
 }  
 // n = mp.size();  
  
 root[0] = 1;  
 build(1, 1, cnt);  
 for (int i = 1; i <= n; ++i) { // 更新版本  
 root[i] = update(root[i - 1], 1, cnt, mp[a[i]]);  
 }  
 n = mp.size();  
 }  
  
 int query(int x, int y, int k) { // k-th number on [x, y]  
 return origin[query\_work(root[x - 1], root[y], 1, n, k)];  
 }  
  
private:  
 int create\_node(int now) {  
 node[++tot] = node[now]; // 传递信息  
 return tot;  
 }  
  
 int build(int now, int nl, int nr) {  
 now = ++tot;  
 node[now].sum = 0;  
 if (nl == nr) {  
 return now;  
 }  
 int mid = (nl + nr) / 2;  
 node[now].ls = build(now, nl, mid);  
 node[now].rs = build(now, mid + 1, nr);  
 return now;  
 }  
  
 int update(int now, int nl, int nr, int pos) { // a[pos] <- a[pos] + 1  
 now = create\_node(now);  
 node[now].sum += 1;  
 if (nl < nr) {  
 int mid = (nl + nr) / 2;  
 if (pos <= mid) {  
 node[now].ls = update(node[now].ls, nl, mid, pos);  
 } else {  
 node[now].rs = update(node[now].rs, mid + 1, nr, pos);  
 }  
 }  
 return now;  
 }  
  
 int query\_work(int x, int y, int nl, int nr, int k) {  
 if (nl == nr) {  
 return nl;  
 }  
 int t = node[node[y].ls].sum - node[node[x].ls].sum, mid = (nl + nr) / 2;  
 if (t >= k) {  
 return query\_work(node[x].ls, node[y].ls, nl, mid, k);  
 } else {  
 return query\_work(node[x].rs, node[y].rs, mid + 1, nr, k - t);  
 }  
 }  
}tr;

所有功能已经封装到结构体中，调用 privite: 里的相应方法即可。不要忘记查询之前先要 init。

### HH 的项链

题意：给定一个序列，有多次询问，每次询问一个区间内有多少种数字。

* 离线/树状数组

按照 $ r $ 的升序排列所有询问，可以通过下面的方法将 $ r $ 相同的询问转换为前缀和 get\_sum(r) - get\_sum(l - 1)。

我们可以 add(i, 1) 表示 $ i $ 处出现了一个新数字。由于要求前缀和，我们只关心 $ a\_i $ 最后出现的位置，可以用 vis 记录其上次出现的位置。若 $ a\_i $ 出现过，那么在修改之前，首先应 add(vis[a[i]], -1) 消除其之前的影响。

int n;  
cin >> n;  
for (int i = 1; i <= n; ++i) {  
 cin >> a[i];  
}  
  
int m;  
cin >> m;  
for (int i = 1; i <= m; ++i) {  
 int l, r;  
 cin >> l >> r;  
 q[r].push\_back({l, i});  
}  
  
tr.n = n;  
  
for (int i = 1; i <= n; ++i) { // r = i  
 if (vis[a[i]]) {  
 tr.add(vis[a[i]], -1);  
 }  
 tr.add(i, 1);  
 vis[a[i]] = i;  
 for (auto c : q[i]) {  
 int &l = c.first, &id = c.second;  
 ans[id] = tr.get\_sum(i) - tr.get\_sum(l - 1);  
 }  
}  
  
for (int i = 1; i <= m; ++i) {  
 cout << ans[i] << "\n";  
}

* 在线/主席树

显然，上面的修改操作可以用主席树维护，然后就可以在线地求解了。

## 单调队列

譬如最经典的滑动窗口问题：

简单来说，**如果有人比你小并且比你强，那你就没用了**

需要做两件事

1. 维护队首（保证间隔）
2. 从队尾插入元素并向前remove冗余元素（去掉老而菜）

订正我的一个误区，是需要构建一个队列的，并不是在原数组上进行双指针操作。

经典例题：滑动窗口

int n, k, head = 0, tail = 0; scanf("%d %d", &n, &k);  
for (int i = 0; i < n; ++i) scanf("%d", a + i);  
for (int type = 0; type < 2; ++type) {  
 memset(que, 0, sizeof(int)\*MAXN);  
 for (int i = 0; i < n; ++i) {  
 if (i - que[head] >= k) ++head;  
 while ((cmp(a[i], a[que[tail]], type) || a[i] == a[que[tail]])  
 && head <= tail) --tail;  
 que[++tail] = i;  
 if (i + 1 >= k) printf("%d ", a[que[head]]);  
 }  
 printf("\n");  
}