1 DS

1.1 bipartite-disjoint-set-union

```
void make_set(int v) {
    parent[v] = make_pair(v, 0);
    rank[v] = 0;
    bipartite[v] = true;
}
pair<int, int> find_set(int v) {
    if (v != parent[v].first) {
        int parity = parent[v].second;
        parent[v] = find_set(parent[v].first);
        parent[v].second ^= parity;
    }
    return parent[v];
}
void add_edge(int a, int b) {
    pair<int, int> pa = find_set(a);
    a = pa.first;
    int x = pa.second;

    pair<int, int> pb = find_set(b);
    b = pb.first;
    int y = pb.second;

    if (a == b) {
        if (x == y)
            bipartite[a] = false;
    } else {
        if (rank[a] < rank[b])
            swap (a, b);
        parent[b] = make_pair(a, x^y^1);
        bipartite[a] &= bipartite[b];
        if (rank[a] == rank[b])
            ++rank[a];
    }
}
bool is_bipartite(int v) {
    return bipartite[find_set(v).first];
}
```

1.2 dsu-rollback

```
struct dsu_save {
    int v, rnkv, u, rnku;
    dsu_save() {}
    dsu_save(int _v, int _rnkv, int _u, int _rnku)
        : v(_v), rnkv(_rnkv), u(_u), rnku(_rnku) {}
};
struct dsu_with_rollbacks {
    vector<int> p, rnk;
    int comps;
    stack<dsu_save> op;
    dsu_with_rollbacks() {}
    dsu_with_rollbacks(int n) {
        p.resize(n);
        rnk.resize(n);
        for (int i = 0; i < n; i++) {
            p[i] = i;
            rnk[i] = 0;
        }
        comps = n;
    }
    int find_set(int v) {
        return (v == p[v]) ? v : find_set(p[v]);
    }
    bool unite(int v, int u) {
        v = find_set(v);
        u = find_set(u);
        if (v == u)
            return false;
        comps--;
        if (rnk[v] > rnk[u])
            swap(v, u);
        op.push(dsu_save(v, rnk[v], u, rnk[u]));
        p[v] = u;
        if (rnk[u] == rnk[v])
            rnk[u]++;
        return true;
    }
    void rollback() {
        if (op.empty())
            return;
        dsu_save x = op.top();
        op.pop();
        comps++;
        p[x.v] = x.v;
        rnk[x.v] = x.rnkv;
        p[x.u] = x.u;
        rnk[x.u] = x.rnku;
    }
};
struct query {
    int v, u;
    bool united;
    query(int _v, int _u) : v(_v), u(_u) {
    }
};
struct QueryTree {
    vector<vector<query>> t;
    dsu_with_rollbacks dsu;
    int T;
    QueryTree() {}
    QueryTree(int _T, int n) : T(_T) {
        dsu = dsu_with_rollbacks(n);
        t.resize(4 * T + 4);
    }
    void add_to_tree(int v, int l, int r, int ul, int
        ur, query& q) {
        if (ul > ur)
            return;
        if (l == ul && r == ur) {
            t[v].push_back(q);
            return;
        }
        int mid = (l + r) / 2;
        add_to_tree(2 * v, l, mid, ul, min(ur, mid), q);
        add_to_tree(2 * v + 1, mid + 1, r, max(ul, mid +
            1), ur, q);
    }
    void add_query(query q, int l, int r) {
        add_to_tree(1, 0, T - 1, l, r, q);
    }
    void dfs(int v, int l, int r, vector<int>& ans) {
        for (query& q : t[v]) {
            q.united = dsu.unite(q.v, q.u);
        }
        if (l == r)
            ans[l] = dsu.comps;
        else {
            int mid = (l + r) / 2;
            dfs(2 * v, l, mid, ans);
            dfs(2 * v + 1, mid + 1, r, ans);
        }
        for (query q : t[v]) {
            if (q.united)
                dsu.rollback();
        }
    }
    vector<int> solve() {
        vector<int> ans(T);
        dfs(1, 0, T - 1, ans);
        return ans;
    }
};
```

1.3 mo

```
struct Query {
    int l, r,k, idx;
    bool operator<(Query other) const
    {
        if(l/block_size!=other.l/block_size) return
            (l<other.l);
        return (l/block_size&1)? (r<other.r) :
            (r>other.r);
    }
};
vector<int> mo_s_algorithm(vector<Query> queries) {
    vector<int> answers(queries.size());
    sort(queries.begin(), queries.end());
    // TODO: initialize data structure
    int cur_l = 0;
    int cur_r = -1;
    // invariant: data structure will always reflect the
        range [cur_l, cur_r]
    for (Query q : queries) {
        while (cur_l > q.l) {
            cur_l--;
            add(cur_l);
        }
        while (cur_r < q.r) {
            cur_r++;
            add(cur_r);
        }
        while (cur_l < q.l) {
            remove(cur_l);
            cur_l++;
        }
        while (cur_r > q.r) {
            remove(cur_r);
            cur_r--;
        }
        answers[q.idx] = get_answer();
    }
    return answers;
}
```

1.4 treap

```
template <class T>
class treap{
    struct item{
        int prior, cnt;
        T key;
        item *l,*r;
        item(T v)
        {
            key=v;
            l=NULL;
            r=NULL;
            cnt=1;
            prior=rand();
        }
    } *root,*node;
    int cnt (item * it){
        return it ? it->cnt : 0;
    }
    void upd_cnt (item * it){
        if (it) it->cnt = cnt(it->l) + cnt(it->r) + 1;
    }
    void split (item * t, T key, item * & l, item * & r){
```

```
    if (!t)
        l = r = NULL;
    else if (key < t->key)
        split (t->l, key, l, t->l), r = t;
    else
        split (t->r, key, t->r, r), l = t;
    upd_cnt(t);
}
void insert (item * & t, item * it){
    if (!t)
        t = it;
    else if (it->prior > t->prior)
        split (t, it->key, it->l, it->r), t = it;
    else
        insert (it->key < t->key ? t->l : t->r, it);
    upd_cnt(t);
}
void merge (item * & t, item * l, item * r){
    if (!l || !r)
        t = l ? l : r;
    else if (l->prior > r->prior)
        merge (l->r, l->r, r), t = l;
    else
        merge (r->l, l, r->l), t = r;
    upd_cnt(t);
}
void erase (item * & t, T key){
    if (t->key == key)
        merge (t, t->l, t->r);
    else
        erase (key < t->key ? t->l : t->r, key);
    upd_cnt(t);
}
T elementAt(item * &t,int key){
    T ans;
    if(cnt(t->l)==key) ans=t->key;
    else if(cnt(t->l)>key) ans=elementAt(t->l,key);
    else ans=elementAt(t->r,key-1-cnt(t->l));
    upd_cnt(t);
    return ans;
}
item * unite (item * l, item * r){
    if (!l || !r) return l ? l : r;
    if (l->prior < r->prior) swap (l, r);
    item * lt, * rt;
    split (r, l->key, lt, rt);
    l->l = unite (l->l, lt);
    l->r = unite (l->r, rt);
    upd_cnt(l);
    upd_cnt(r);
    return l;
}
void heapify (item * t){
    if (!t) return;
    item * max = t;
    if (t->l != NULL && t->l->prior > max->prior)
        max = t->l;
    if (t->r != NULL && t->r->prior > max->prior)
        max = t->r;
    if (max != t)
    {
        swap (t->prior, max->prior);
        heapify (max);
    }
}
item * build (T * a, int n){
    if (n == 0) return NULL;
    int mid = n / 2;
    item * t = new item (a[mid], rand ());
    t->l = build (a, mid);
    t->r = build (a + mid + 1, n - mid - 1);
    heapify (t);
    return t;
}
void output (item * t,vector<T> &arr){
    if (!t) return;
    output (t->l,arr);
    arr.push_back(t->key);
    output (t->r,arr);
}
public:
treap(){
    root=NULL;
}
treap(T *a,int n){
    build(a,n);
}
void insert(T value){
    node=new item(value);
    insert(root,node);
}
void erase(T value){
    erase(root,value);
}
T elementAt(int position){
    return elementAt(root,position);
}
int size(){
    return cnt(root);
}
void output(vector<T> &arr){
    output(root,arr);
}
int range_query(T l,T r){ //(l,r]
    item *previous,*next,*current;
    split(root,l,previous,current);
    split(current,r,current,next);
    int ans=cnt(current);
    merge(root,previous,current);
    merge(root,root,next);
    previous=NULL;
    current=NULL;
    next=NULL;
    return ans;
}
};
template <class T>
class implicit_treap{
struct item{
    int prior, cnt;
    T value;
    bool rev;
    item *l,*r;
    item(T v){
        value=v;
        rev=false;
        l=NULL;
        r=NULL;
        cnt=1;
        prior=rand();
    }
} *root,*node;
int cnt (item * it){
    return it ? it->cnt : 0;
}
void upd_cnt (item * it){
    if (it)
        it->cnt = cnt(it->l) + cnt(it->r) + 1;
}
void push (item * it){
    if (it && it->rev){
        it->rev = false;
        swap (it->l, it->r);
        if (it->l) it->l->rev ^= true;
        if (it->r) it->r->rev ^= true;
    }
}
void merge (item * & t, item * l, item * r){
    push (l);
    push (r);
    if (!l || !r)
        t = l ? l : r;
    else if (l->prior > r->prior)
        merge (l->r, l->r, r), t = l;
    else
        merge (r->l, l, r->l), t = r;
    upd_cnt (t);
}
void split (item * t, item * & l, item * & r, int
     key, int add = 0){
    if (!t)
        return void( l = r = 0 );
    push (t);
    int cur_key = add + cnt(t->l);
    if (key <= cur_key)
        split (t->l, l, t->l, key, add), r = t;
    else
        split (t->r, t->r, r, key, add + 1 +
            cnt(t->l)), l = t;
    upd_cnt (t);
}
void insert(item * &t,item * element,int key){
    item *l,*r;
    split(t,l,r,key);
    merge(l,l,element);
    merge(t,l,r);
    l=NULL;
    r=NULL;
}
T elementAt(item * &t,int key){
    push(t);
    T ans;
    if(cnt(t->l)==key) ans=t->value;
    else if(cnt(t->l)>key) ans=elementAt(t->l,key);
    else ans=elementAt(t->r,key-1-cnt(t->l));
    return ans;
}
void erase (item * & t, int key){
    push(t);
    if(!t) return;
    if (key == cnt(t->l))
        merge (t, t->l, t->r);
    else if(key<cnt(t->l))
        erase(t->l,key);
    else
        erase(t->r,key-cnt(t->l)-1);
    upd_cnt(t);
}
void reverse (item * &t, int l, int r){
    item *t1, *t2, *t3;
    split (t, t1, t2, l);
    split (t2, t2, t3, r-l+1);
    t2->rev ^= true;
    merge (t, t1, t2);
    merge (t, t, t3);
}
void cyclic_shift(item * &t,int L,int R){
    if(L==R) return;
    item *l,*r,*m;
    split(t,t,l,L);
```

```
        split(l,l,m,R-L+1);
        split(l,l,r,R-L);
        merge(t,t,r);
        merge(t,t,l);
        merge(t,t,m);
        l=NULL;
        r=NULL;
        m=NULL;
    }
    void output (item * t,vector<T> &arr){
        if (!t) return;
        push (t);
        output (t->l,arr);
        arr.push_back(t->value);
        output (t->r,arr);
    }
public:
    implicit_treap(){
        root=NULL;
    }
    void insert(T value,int position){
        node=new item(value);
        insert(root,node,position);
    }
    void erase(int position){
        erase(root,position);
    }
    void reverse(int l,int r){
        reverse(root,l,r);
    }
    T elementAt(int position){
        return elementAt(root,position);
    }
    void cyclic_shift(int L,int R){
        cyclic_shift(root,L,R);
    }
    int size(){
        return cnt(root);
    }
    void output(vector<T> &arr){
        output(root,arr);
    }
};
```

2 header

```
#define FastIO ios::sync_with_stdio(false);
    cin.tie(0);cout.tie(0)

#include <ext/pb_ds/assoc_container.hpp> // Common file
using namespace __gnu_pbds;

/*
find_by_order(k) --> returns iterator to the kth largest
    element counting from 0
order_of_key(val) --> returns the number of items in a
    set that are strictly smaller than our item
*/
typedef tree<
int,
null_type,
less<int>,
rb_tree_tag,
tree_order_statistics_node_update>
ordered_set;

#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
gp_hash_table<int, int> table;
struct custom_hash {
    static uint64_t splitmix64(uint64_t x) {
        // http://xorshift.di.unimi.it/splitmix64.c
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }
    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM =
            chrono::steady_clock::now().time_since_epoch().co
        return splitmix64(x + FIXED_RANDOM);
    }
};
gp_hash_table<long long, int, custom_hash>
    safe_hash_table;
```