Segment tree

```
ll update(int pos, int val, int node, int x, int y) {
	if (pos<x || pos>y)return seg[node];
	if (x == y)return seg[node] = val;
	int mid = (x + y) >> 1;
	return seg[node] = update(pos, val, node * 2, x, mid) +
update(pos, val, node * 2 + 1, mid + 1, y);
}
ll query(int lo, int hi, int node, int x, int y) {
	if (lo > y || hi < x)return 0;
	if (lo <= x && y <= hi)return seg[node];
	int mid = (x + y) >> 1;
	return query(lo, hi, node * 2, x, mid) + query(lo, hi, node * 2
+ 1, mid + 1, y);
}
```

lazy

```
ll seg[100001 * 4], lazy[100001 * 4];

void propagate(ll lo, ll hi, ll node) {
	if (lazy[node]) {
		if (lo < hi) {
			lazy[node * 2] += lazy[node];
			lazy[node * 2 + 1] += lazy[node];
		}
		seg[node] += lazy[node] * (hi - lo + 1);
		lazy[node] = 0;
	}
}
void update(ll lo, ll hi, ll node, ll x, ll y, ll val) {
	propagate(lo, hi, node);
	if (x > hi || lo > y)return;
	if (x <= lo && hi <= y) {
		lazy[node] = val;
		propagate(lo, hi, node);
		return;
	}
	ll mid = (lo + hi) >> 1;
	update(lo, mid, node << 1, x, y, val);
	update(mid + 1, hi, node << 1 | 1, x, y, val);
	seg[node] = seg[node * 2] + seg[node + 2 + 1];
}
ll query(ll lo, ll hi, ll node, ll x, ll y) {
	propagate(lo, hi, node);
	if (hi < x || y < lo)return 0;
	if (x <= lo && hi <= y)return seg[node];
	ll mid = (lo + hi) >> 1;
	ll left = query(lo, mid, node << 1, x, y);
	ll right = query(mid + 1, hi, node << 1 | 1, x, y);
	return left + right;
}
```

ett + lazy

```
int Visit[100005], check[100005], seg[100005 * 4], lazy[100005 * 4];
vector<vector<int>> v;

void dfs(int cur) {
	Visit[cur] = ++cnt;
	for (auto& i : v[cur])
		if (!Visit[i])dfs(i);
	check[cur] = cnt;
}
void propagate(int lo, int hi, int node) {
	if (!lazy[node])return;
	else {
		if (lo != hi) {
			lazy[node * 2] += lazy[node];
			lazy[node * 2 + 1] += lazy[node];
		}
	}
	seg[node] += lazy[node] * (hi - lo + 1);
	lazy[node] = 0;
}
```

```
int update(int lo, int hi, int val, int node, int x, int y) {
	propagate(x, y, node);
	if (hi < x || y < lo)return seg[node];
	if (lo <= x && y <= hi) {
		lazy[node] += val;
		propagate(x, y, node);
		return seg[node];
	}
	int mid = (x + y) >> 1;
	return seg[node] = update(lo, hi, val, node * 2, x, mid) +
update(lo, hi, val, node * 2 + 1, mid + 1, y);
}
int query(int lo, int hi, int node, int x, int y) {
	propagate(x, y, node);
	if (hi < x || y < lo)return 0;
	if (lo <= x && y <= hi)return seg[node];
	int mid = (x + y) >> 1;
	return query(lo, hi, node * 2, x, mid) + query(lo, hi, node * 2
+ 1, mid + 1, y);
}
```

sqrt decomposition

```
bool check(int x, int y) { return (x <= y) ? true : false; }
void make_dcmp(){
	bucket_size = sqrt(n);
	for (int i = 0; i < n; i++)
		comp[i / csize].push_back(s[i]);
	for (int i = 0; i < n / csize; i++)
		sort(comp[i].begin(), comp[i].end());//depends on
logic
}
void update(int pos, int val);//update element in bucket
int query(int lo, int hi, int val){
	int cnt = 0;
	while (lo % csize && check(lo, hi))//adjustment to bucket
		if (s[lo++] > val)//depends on logic
			cnt++;
	while ((hi + 1) % csize && check(lo, hi))//adjustment to bucket
		if (s[hi--] > val)//depends on logic
			cnt++;
	while (check(lo, hi)) {//bucket by bucket
		cnt += comp[lo / csize].end() - upper_bound(comp[lo /
csize].begin(), comp[lo / csize].end(), val);//depends on logic
		lo += bucket_size;
	}
}
```

mo's

```
struct make_dcmp {
	int lo, hi, id;
	bool operator<(const make_dcmp& d) {
		if (lo / sz != d.lo / sz)return (lo / sz < d.lo /
sz);
		else return hi < d.hi;
	}
};
make_dcmp dcmp[100005];
void add(int x);//logic
void erase(int x);//logic
sort(dcmp, dcmp + q);
int x = 0, y = 0;
for (int i = 0; i < q; i++) {
	int lo = dcmp[i].lo, hi = dcmp[i].hi, idx = dcmp[i].id;
	if (!i) {
		for (int j = lo; j < hi + 1; j++)add(j);
		result[idx] = ans, x = lo, y = hi;
		continue;
	}
	while (x < lo)erase(x++);
	while (lo < x)add(--x);
	while (hi < y)erase(y--);
	while (y < hi)add(++y);
	result[idx] = ans, x = lo, y = hi;
}
```