1 Data Structures

1.1 Segment Tree

1.1.1 Lazy

```

struct RSQ // Range sum query
{
    static ll const neutro = 0;
    static ll op(ll x, ll y)
    {
        return x + y;
    }
    static ll
    lazy_op(int i, int j, ll x)
    {
        return (j - i + 1) * x;
    }
};

struct RMinQ // Range minimun query
{
    static ll const neutro = 1e18;
    static ll op(ll x, ll y)
    {
        return min(x, y);
    }
    static ll
    lazy_op(int i, int j, ll x)
    {
        return x;
    }
};

template <class t>
class SegTreeLazy
{
    vector<ll> arr, st, lazy;
    int n;

    void build(int u, int i, int j)
    {
        if (i == j)
        {
            st[u] = arr[i];
            return;
        }
        int m = (i + j) / 2, l = u * 2 + 1, r = u * 2 + 2;
        build(l, i, m);
        build(r, m + 1, j);
        st[u] = t::op(st[l], st[r]);
    }

    void propagate(int u, int i, int j, ll x)
    {
        // nota, las operaciones pueden ser un and, or, ..., etc.
        st[u] += t::lazy_op(i, j, x); // incrementar el valor (+)
        // st[u] = t::lazy_op(i, j, x); // setear el valor
        if (i != j)
        {
            // incrementar el valor
            lazy[u * 2 + 1] += x;
            lazy[u * 2 + 2] += x;
            // setear el valor
            //lazy[u * 2 + 1] = x;
            //lazy[u * 2 + 2] = x;
        }
        lazy[u] = 0;
    }

    ll query(int a, int b, int u, int i, int j)
    {
        if (j < a or b < i)
            return t::neutro;
        int m = (i + j) / 2, l = u * 2 + 1, r = u * 2 + 2;
        if (lazy[u])
            propagate(u, i, j, lazy[u]);
        if (a <= i and j <= b)
            return st[u];
        ll x = query(a, b, l, i, m);
        ll y = query(a, b, r, m + 1, j);
        return t::op(x, y);
    }

    void update(int a, int b, ll value,
                int u, int i, int j)
    {
        int m = (i + j) / 2, l = u * 2 + 1, r = u * 2 + 2;
        if (lazy[u])
            propagate(u, i, j, lazy[u]);
        if (a <= i and j <= b)
            propagate(u, i, j, value);
        else if (j < a or b < i)
            return;
        else
        {
            update(a, b, value, l, i, m);
            update(a, b, value, r, m + 1, j);
            st[u] = t::op(st[l], st[r]);
        }
    }

  public:
    SegTreeLazy(vector<ll> &v)
    {
        arr = v;
        n = v.size();
        st.resize(n * 4 + 5);
        lazy.assign(n * 4 + 5, 0);
```

```
        build(0, 0, n - 1);
    }

    ll query(int a, int b)
    {
        return query(a, b, 0, 0, n - 1);
    }

    void update(int a, int b, ll value)
    {
        update(a, b, value, 0, 0, n - 1);
    }
};
```

1.1.2 Iterative

```

// It requires a struct for a node (e.g. prodsgn)
// A node must have three constructors
//      Arity 0: Constructs the identity of the operation (e.g. 1 for prodsgn)
//      Arity 1: Constructs a leaf node from the input
//      Arity 2: Constructs a node from its children
//
// Building the Segment Tree:
//      Create a vector of nodes (use constructor of arity 1).
//      ST<miStructNode> mySegmentTree(vectorOfNodes);
// Update:
//      mySegmentTree.set_points(index, myStructNode(input));
// Query:
//      mySegmentTree.query(l, r); (It searches on the range [l,r), and returns a node.)

// Logic And Query
struct ANDQ
{
    ll value;
    ANDQ() { value = -1ll; }
    ANDQ(ll x) { value = x; }
    ANDQ(const ANDQ &a,
         const ANDQ &b)
    {
        value = a.value & b.value;
    }
};

// Interval Product (LiveArchive)
struct prodsgn
{
    int sgn;
    prodsgn() { sgn = 1; }
    prodsgn(int x)
    {
        sgn = (x > 0) - (x < 0);
    }
    prodsgn(const prodsgn &a,
            const prodsgn &b)
    {
        sgn = a.sgn * b.sgn;
    }
};

// Maximum Sum (SPOJ)
struct maxsum
{
    int first, second;
    maxsum() { first = second = -1; }
    maxsum(int x)
    {
        first = x;
        second = -1;
    }
    maxsum(const maxsum &a,
           const maxsum &b)
    {
        if (a.first > b.first)
        {
            first = a.first;
            second = max(a.second,
                         b.first);
        }
        else
        {
            first = b.first;
            second = max(a.first,
                         b.second);
        }
    }
    int answer()
    {
        return first + second;
    }
};

// Range Minimum Query
struct rminq
{
    int value;
    rminq() { value = INT_MAX; }
    rminq(int x) { value = x; }
    rminq(const rminq &a,
          const rminq &b)
    {
        value = min(a.value,
                    b.value);
    }
};

template <class node>
class ST
```

```
{
    vector<node> t;
    int n;

public:
    ST(vector<node> &arr)
    {
        n = arr.size();
        t.resize(n * 2);
        copy(arr.begin(), arr.end(), t.begin() + n);
        for (int i = n - 1; i > 0; --i)
            t[i] = node(t[i << 1], t[i << 1 | 1]);
    }

    // 0-indexed
    void set_point(int p, const node &value)
    {
        for (t[p += n] = value; p > 1; p >>= 1)
            t[p >> 1] = node(t[p], t[p ^ 1]);
    }

    // inclusive exclusive, 0-indexed
    node query(int l, int r)
    {
        node ansl, ansr;
        for (l += n, r += n; l < r; l >>= 1, r >>= 1)
        {
            if (l & 1)
                ansl = node(ansl, t[l++]);
            if (r & 1)
                ansr = node(t[--r], ansr);
        }
        return node(ansl, ansr);
    }
};
```

1.2 Wavelet Tree

```

class WaveTree
{
    typedef vector<int>::iterator iter;
    vector<vector<int>> r0;
    int n, s;
    vector<int> arrCopy;

    void build(iter b, iter e, int l, int r, int u)
    {
        if (l == r)
            return;
        int m = (l + r) / 2;
        r0[u].reserve(e - b + 1);
        r0[u].push_back(0);
        for (iter it = b; it != e; ++it)
            r0[u].push_back(r0[u].back() + (*it <= m));
        iter p = stable_partition(b, e, [=](int i) { return i <= m; });
        build(b, p, l, m, u * 2);
        build(p, e, m + 1, r, u * 2 + 1);
    }

    int q, w;
    int range(int a, int b, int l, int r, int u)
    {
        if (r < q or w < l)
            return 0;
        if (q <= l and r <= w)
            return b - a;
        int m = (l + r) / 2, za = r0[u][a], zb = r0[u][b];
        return range(za, zb, l, m, u * 2) +
               range(a - za, b - zb, m + 1, r, u * 2 + 1);
    }

public:
    //arr[i] in [0,sigma)
    WaveTree(vector<int> arr, int sigma)
    {
        n = arr.size();
        s = sigma;
        r0.resize(s * 2);
        arrCopy = arr;
        build(arr.begin(), arr.end(), 0, s - 1, 1);
    }

    //k in [1,n], [a,b) is 0-indexed, -1 if error
    int quantile(int k, int a, int b)
    {
        //extra conditions disabled
        if (/*a < 0 or b > n or*/ k < 1 or k > b - a)
            return -1;
        int l = 0, r = s - 1, u = 1, m, za, zb;
        while (l != r)
        {
            m = (l + r) / 2;
            za = r0[u][a];
            zb = r0[u][b];
            u *= 2;
            if (k <= zb - za)
                a = za, b = zb, r = m;
            else
                k -= zb - za, a -= za, b -= zb,
                    l = m + 1, ++u;
        }
        return r;
    }

    //counts numbers in [x,y] in positions [a,b)
    int range(int x, int y, int a, int b)
    {
        if (y < x or b <= a)
            return 0;
```

```
        q = x;
        w = y;
        return range(a, b, 0, s - 1, 1);
    }

    //count occurrences of x in positions [0,k)
    int rank(int x, int k)
    {
        int l = 0, r = s - 1, u = 1, m, z;
        while (l != r)
        {
            m = (l + r) / 2;
            z = r0[u][k];
            u *= 2;
            if (x <= m)
                k = z, r = m;
            else
                k -= z, l = m + 1, ++u;
        }
        return k;
    }

    //x in [0,sigma)
    void push_back(int x)
    {
        int l = 0, r = s - 1, u = 1, m, p;
        ++n;
        while (l != r)
        {
            m = (l + r) / 2;
            p = (x <= m);
            r0[u].push_back(r0[u].back() + p);
            u *= 2;
            if (p)
                r = m;
            else
                l = m + 1, ++u;
        }
    }

    //doesn't check if empty
    void pop_back()
    {
        int l = 0, r = s - 1, u = 1, m, p, k;
        --n;
        while (l != r)
        {
            m = (l + r) / 2;
            k = r0[u].size();
            p = r0[u][k - 1] - r0[u][k - 2];
            r0[u].pop_back();
            u *= 2;
            if (p)
                r = m;
            else
                l = m + 1, ++u;
        }
    }

    //swap arr[i] with arr[i+1], i in [0,n-1)
    void swap_adj(int i)
    {
        int &x = arrCopy[i], &y = arrCopy[i + 1];
        int l = 0, r = s - 1, u = 1;
        while (l != r)
        {
            int m = (l + r) / 2, p = (x <= m), q = (y <= m);
            if (p != q)
            {
                r0[u][i + 1] ^= r0[u][i] ^ r0[u][i + 2];
                break;
            }
            u *= 2;
            if (p)
                r = m;
            else
                l = m + 1, ++u;
        }
        swap(x, y);
    }
};
```