1 Data Structures

1.1 Segment Tree

1.1.1 Lazy

```cpp
#include "../../headers/headers.h"

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

```
#include "../../headers/headers.h"

// 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);
    }
};

```

```cpp
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);
    }
};
```