Estruturas

1. Segment Tree

Lazy Propagation, Query de soma entre os ranges

1. Construa a árvore a partir do índice idx=1, com Left=0 e Right=n-1. (Indexando do zero).
    - Se `Left==Right`, a seg_tree no índice idx terá como resultado arr[Left]. Então retorne.
    - `Left>Right` é proibido, retorne se ocorrer.
    - Construa a árvore a esquerda: `idx=idx*2, Left=Left, Right=(Left+Right)/2`
    - Construa a árvore a direita: `idx=1+idx*2, Left=1+(Left+Right)/2, Right`
    - Atualize `seg_tree[idx]` com os resultados a esquerda e a direita:

`st[idx] = st[idx*2]+st[1+idx*2];`

2. Para atualizar a árvore com a soma de um elemento (a partir de idx=1):
    - Se uma condição proibida ocorrer, retorne (`L>R`, por exemplo)
    - Se o nó atual tiver a flag de lazy setado, propague a flag de lazy para os filhos a direita e a esquerda:

```
st[idx] += lazy[idx]*(R-L+1);
if (L!=R) lazy[idx*2]+=lazy[idx], lazy[1+idx*2]+=lazy[id
lazy[idx] = 0;
```

- Se o range encontrado é o desejado `L>=l && R<=r`.

3. Algoritmo

```
void build_st(int idx=1, int L=0, int R=n-1)
{
        if (L>R)
                return;

        if (L==R) {
                st[idx] = arr[L];
                return;
        }

        int mid = (L+R)/2;
        build_st(idx*2, L, mid);
        build_st(1+idx*2, mid+1, R);
        st[idx] = st[idx*2]+st[1+idx*2];
}

ll query_st(int idx, int a, int b, int i, int j)
{
        if(a>b||a>j||b<i) return 0;
        if (lazy[idx] !=0 )
        {
                st[idx]+=lazy[idx]*(b-a+1);
                if (a!=b)
                {
                        lazy[idx*2]+=lazy[idx];
                        lazy[idx*2+1]+=lazy[idx];
                }
                lazy[idx]=0;
        }

        if (a>=i && b<=j) return st[idx];
```

```
        ll q1=query_st(idx*2, a, (a+b)/2, i, j);
        ll q2=query_st(idx*2+1, (a+b)/2+1, b, i, j);

        return q1+q2;
}

void update_st(int idx, int a, int b, int i, int j, int inc)
{
        if(a>b) return;
        if (lazy[idx]!=0)
        {
                st[idx]+=lazy[idx]*(b-a+1);
                if (a!=b)
                {
                        lazy[idx*2]+=lazy[idx];
                        lazy[idx*2+1]+=lazy[idx];
                }
                lazy[idx]=0;
        }
        if(a>b||a>j||b<i) return;

        if (a>=i && b<=j)
        {
                st[idx]+=inc*(b-a+1);
                if (a!=b)
                {
                        lazy[idx*2]+=inc;
                        lazy[idx*2+1]+=inc;
                }
                return;
        }

        update_st(idx*2, a, (a+b)/2, i, j, inc);
        update_st(idx*2+1, (a+b)/2+1, b,i, j, inc);
        st[idx] = st[idx*2] + st[idx*2+1];
}
```

2. Fenwick Tree

Encontrar soma no range [l; r]

- Algoritmo

```
struct FenwickTree {
    vector<int> bit; // binary indexed tree
    int n;

    void init(int n) {
        this->n = n;
        bit.assign(n, 0);
    }
    int sum(int r) {
        int ret = 0;
        for (; r >= 0; r = (r & (r+1)) - 1)
            ret += bit[r];
        return ret;
    }
    void add(int idx, int delta) {
        for (; idx < n; idx = idx | (idx+1))
            bit[idx] += delta;
    }
    int sum(int l, int r) {
        return sum(r) - sum(l-1);
    }
    void init(vector<int> a) {
        int len = a.size();
        init(len);
        for (size_t i = 0; i < len; i++)
            add(i, a[i]);
    }
};
```

Encontrar o mínimo no range [0;r]

- Algoritmo

```
struct FenwickTreeMin {
    vector<int> bit;
    int n;
    const int INF = (int)1e9;
    void init (int n) {
        this->n = n;
        bit.assign (n, INF);
    }
    int getmin (int r) {
        int ret = INF;
        for (; r >= 0; r = (r & (r+1)) - 1)
            ret = min(ret, bit[r]);
        return ret;
    }
    void update (int idx, int val) {
        for (; idx < n; idx = idx | (idx+1))
            bit[idx] = min(bit[idx], val);
    }
    void init (vector<int> a) {
        init (a.size());
        for (size_t i = 0; i < a.size(); i++)
            update(i, a[i]);
    }
};
```

Soma em duas dimensões

- Algoritmo

```
struct FenwickTree2D {
    vector <vector <int> > bit;
    int n, m;
```

```
    // init(...) { ... }
    int sum (int x, int y) {
        int ret = 0;
        for (int i = x; i >= 0; i = (i & (i+1)) - 1)
            for (int j = y; j >= 0; j = (j & (j+1)) - 1)
                ret += bit[i][j];
        return ret;
    }
    void add(int x, int y, int delta) {
        for (int i = x; i < n; i = i | (i+1))
            for (int j = y; j < m; j = j | (j+1))
                bit[i][j] += delta;
    }
};
```