functions

calcfact

```
auto calc_fact = [mpow]{
        constexpr int N = 1e7;
        vector<mint> fact(N + 1, 1);
        vector<mint> inv(N + 1, 1);
        for(int i = 1; i < N; ++i){
                fact[i + 1] = fact[i] * (i + 1);
                inv[i + 1] = mpow(fact[i + 1], MOD - 2);
        }
        return make_pair(fact, inv);
};
vector<mint> fact, inv;
tie(fact, inv) = calc_fact();
```

divisor

```
auto divisor = [](i64 x){
        int sq = sqrt(x) + 1;
        vector<int> ret;
        for(i64 i = 1; i < sq; ++i)
                if(!(x % i)){
                        ret.emplace_back(i);
                        if(i * i != x)
                                ret.emplace_back(x / i);
                }
        sort(ret.begin(), ret.end());
        return ret;
};
```

eratosthenes

```
auto eratosthenes = []{
        constexpr int N = 2e6;
        bitset<N> not_prime(3);
        for(int i = 2; i < N; ++i)
                if(!not_prime[i])
                        for(int j = 2 * i; j < N; j += i)
                                not_prime.set(j);
        return not_prime;
};
```

factoring

```
auto factoring = [](i64 x){
        int sq = sqrt(x) + 1;
        vector<int> ret;
        if(x == 1){
                ret.emplace_back(1);
                return ret;
        }
        for(i64 i = 2; i < sq; ++i)
                while(x % i == 0){
                        ret.emplace_back(i);
                        x /= i;
                }
        if(x != 1)
                ret.emplace_back(x);
        return ret;
};
```

modpow

```
auto mpow = [](auto x, i64 y){
        auto z = x;
        decltype(x) val = y & 1 ? x : decltype(x)(1);
        while(z *= z, y >>= 1)
                if(y & 1)
                        val *= z;
        return val;
};
```

ncr

```
auto ncr = [&fact, &inv](int n, int r){
        if(n < 0 || r < 0 || n < r)
                return mint(0);
        return fact[n] * inv[r] * inv[n - r];
};
```

nhr

```
auto nhr = [ncr](int n, int r){
        return ncr(n + r - 1, r);
};
```

scc

```
vector<int> scc(vector<vector<int>>& edges){
    int n = edges.size();
    vector<vector<int>> rev(n);

    for(int i = 0; i < n; ++i)
        for(auto& x : edges[i])
            rev[x].emplace_back(i);

    vector<i64> dfs_num(n, -1);
    vector<i64> flag(n, 0);
    int num = 0;
    function<void(int)> dfs = [&](int pos){
        flag[pos] = 1;
        for(auto& xx : edges[pos])
            if(!flag[xx]){
                dfs(xx);
        }
        dfs_num[pos] = num++;
    };

    for(int i = 0; i < n; ++i)
        if(!flag[i])
            dfs(i);

    vector<int> dfs_inv(n);
    for(int i = 0; i < n; ++i)
        dfs_inv[n - 1 - dfs_num[i]] = i;

    num = 0;

    vector<int> scc_vec(n, -1);

    function<void(int)> rdfs = [&](int pos){
        scc_vec[pos] = num;
        for(auto t : rev[pos])
            if(scc_vec[t] == -1)
                rdfs(t);
    };

    for(int i = 0; i < n; ++i)
```

```
        if(scc_vec[dfs_inv[i]] == -1){
            rdfs(dfs_inv[i]);
            ++num;
        }

    return scc_vec;
}
```

classes

BinaryIndexedTree

```
template <typename T>
struct BIT{
    vector<T> elm;
    BIT(int n, T init = T()) : elm(n + 1, init){
    }

    // [0, x)
    T sum(int x){
        T val = 0;
        for(; x > 0; x -= x & -x)
            val += elm[x];
        return val;
    }

    void add(int x, T val){
        for(++x; x < elm.size(); x += x & -x)
            elm[x] += val;
    }
};
```

Compression

```
template<typename T>
struct Compression{
    vector<T> compvec;
    Compression(vector<T>& inp){//圧縮する
        compvec = inp;
        sort(compvec.begin(), compvec.end());
        compvec.erase(unique(compvec.begin(), compvec.end()), compvec.end());
    }
    int Index(T val){//圧縮を元に対応するインデックスを返す
        auto it = lower_bound(compvec.begin(), compvec.end(), val);
        return distance(compvec.begin(), it);
    }
    vector<T>& operator*(){
        return compvec;
    }
};
```

ConvexHullTrick

```
template <typename T, typename U>
struct ConvexHullTrick{
    // 任意の2関数で共有点が高々1個ならElmの中身を適切に変えれば通る

    struct Elm{
        T a, b;
        U operator()(T x){
            return a * x + b;
        }
    };

    struct Node{
        Elm f;
```

```
        Node* l;
        Node* r;
        Node(Elm elm) : f(elm), l(nullptr), r(nullptr){}
    };

    U _min, _max, _inf;
    Node* root;

    ConvexHullTrick(U _min, U _max, U _inf) :
        _min(_min),
        _max(_max),
        _inf(_inf),
        root(nullptr)
    {
    }

    Node* _insert(Node* p, T st, T en, Elm f){
        if(!p)
            return new Node(f);
        if(p->f(st) <= f(st) && p->f(en) <= f(en))
            return p;
        if(p->f(st) >= f(st) && p->f(en) >= f(en)){
            p->f = f;
            return p;
        }
        T mid = (st + en) / 2;
        if(p->f(mid) > f(mid))
            swap(p->f, f);
        if(p->f(st) >= f(st))
            p->l = _insert(p->l, st, mid, f);
        else
            p->r = _insert(p->r, mid, en, f);
        return p;
    }

    U _query(Node* p, T st, T en, T x){
        if(!p)
            return _inf;
        if(st == en)
            return p->f(x);
        T mid = (st + en) / 2;
        if(x <= mid)
            return min(p->f(x), _query(p->l, st, mid, x));
        else
            return min(p->f(x), _query(p->r, mid, en, x));
    }

    void insert(Elm f){
        root = _insert(root, _min, _max, f);
    }

    U query(T x){
        return _query(root, _min, _max, x);
    }
};
```

Dinic

```
template <typename T>
struct Dinic{
    struct Edge{
        int to, rev;
        T cap;
        Edge(int to, T cap, int rev) : to(to), rev(rev), cap(cap){}
    };

    vector<vector<Edge>> edges;
    T _inf;
    vector<T> min_cost;
    vector<int> cnt;

    Dinic(int n) : edges(n), _inf(numeric_limits<T>::max()){}

    void add(int from, int to, T cap){
```

```
        edges[from].emplace_back(to, cap, static_cast<T>(edges[to].size()));
        edges[to].emplace_back(from, 0, static_cast<T>(edges[from].size()) - 1);
    }

    bool bfs(int s, int t){
        min_cost.assign(edges.size(), -1);
        queue<int> que;
        min_cost[s] = 0;
        que.emplace(s);
        while(!que.empty() && min_cost[t] == -1){
            int x = que.front();
            que.pop();
            for(auto& ed : edges[x])
                if(ed.cap > 0 && min_cost[ed.to] == -1){
                    min_cost[ed.to] = min_cost[x] + 1;
                    que.emplace(ed.to);
                }
        }
        return min_cost[t] != -1;
    }

    T dfs(int idx, int t, T flow){
        if(idx == t)
            return flow;
        for(int i = cnt[idx]; i < edges[idx].size(); ++i){
            auto& ed = edges[idx][i];
            if(ed.cap > 0 && min_cost[idx] < min_cost[ed.to]){
                T res = dfs(ed.to, t, min(flow, ed.cap));
                if(res > 0){
                    ed.cap -= res;
                    edges[ed.to][ed.rev].cap += res;
                    return res;
                }
            }
        }
        return 0;
    }

    T solve(int s, int t){
        T flow = 0;
        while(bfs(s, t)){
            cnt.assign(edges.size(), 0);
            T f = 0;
            while((f = dfs(s, t, _inf)) > 0)
                flow += f;
        }
        return flow;
    }

};
```

DynamicLazySegmentTree

```
template<typename T, typename U>
struct Segtree{

    struct SegNode{
        T val;
        U lazy;

        SegNode* l;
        SegNode* r;
        SegNode(T val, U lazy) : val(val), lazy(lazy), l(nullptr), r(nullptr){}
    };

    i64 n;
    function<T(T, T)> f;
    function<T(T, U, int)> g;
    function<U(U, U)> h;
    T op_t;
    U op_u;

    SegNode* root;
```

```
    Segtree(int n_, function<T(T, T)> f, function<T(T, U, int)> g, function<U(U, U)> h, T op_t, U op_u) :
        f(f), g(g), h(h), op_t(op_t), op_u(op_u){
        for(n = 1; n < n_; n <<= 1);
        root = new SegNode(op_t, op_u);
    }

    SegNode* getl(SegNode* node){
        return node->l ? node->l : node->l = new SegNode(op_t, op_u);
    }

    SegNode* getr(SegNode* node){
        return node->r ? node->r : node->r = new SegNode(op_t, op_u);
    }

    void eval(SegNode* node, i64 len){
        node->val = g(node->val, node->lazy, len);
        getl(node);
        node->l->lazy = h(node->l->lazy, node->lazy);
        getr(node);
        node->r->lazy = h(node->r->lazy, node->lazy);
        node->lazy = op_u;
    }

    void update(i64 x, i64 y, U val, SegNode* node = nullptr, i64 l = 0, i64 r = 0){
        if(node == nullptr){
            node = root;
            r = n;
        }
        eval(node, r - l);
        if(r <= x || y <= l)
            return ;
        if(x <= l && r <= y){
            node->lazy = h(node->lazy, val);
            eval(node, r - l);
        }else{
            i64 mid = (l + r) >> 1;
            update(x, y, val, getl(node), l, mid);
            update(x, y, val, getr(node), mid, r);
            node->val = f(node->l->val, node->r->val);
        }
    }

    T get(i64 x, i64 y, SegNode* node = nullptr, i64 l = 0, i64 r = 0){
        if(node == nullptr){
            node = root;
            r = n;
        }

        if(r <= x || y <= l)
            return op_t;
        eval(node, r - l);
        if(x <= l && r <= y)
            return node->val;

        i64 val_l = op_t, val_r = op_t;
        i64 mid = (l + r) >> 1;

        if(node->l)
            val_l = get(x, y, node->l, l, mid);
        if(node->r)
            val_r = get(x, y, node->r, mid, r);
        return f(val_l, val_r);
    }

};
```

DynamicSegmentTree

```
template <typename T>
struct Segtree{

    struct SegNode;

    struct SegNode{
```

```
    T val;
    SegNode* l;
    SegNode* r;

    SegNode(T val) : val(val), l(nullptr), r(nullptr){}
};

i64 n;
function<T(T, T)> f;
T op;
SegNode* root;

Segtree(int n_, function<T(T, T)> f, T op) : f(f), op(op){
    for(n = 1; n < n_; n <<= 1);
    root = new SegNode(op);
}

SegNode* getl(SegNode* node, T val){
    return node->l == nullptr ? node->l = new SegNode(val) : node->l;
}

SegNode* getr(SegNode* node, T val){
    return node->r == nullptr ? node->r = new SegNode(val) : node->r;
}

void eval(SegNode* node){
    node->val = f(node->l == nullptr ? op : node->l->val, node->r == nullptr ? op : node->r->val);
}

void set(i64 x, T val){
    assert(0 <= x && x < n);

    SegNode* node = root;
    stack<SegNode*> nodes;
    i64 l = 0, r = n;

    while(r - l > 1){
        nodes.push(node);
        i64 mid = (l + r) >> 1;

        if(x < mid){
            node = getl(node, x);
            r = mid;
        }else{
            node = getr(node, x);
            l = mid;
        }
    }

    node->val = val;
    while(!nodes.empty()){
        eval(nodes.top());
        nodes.pop();
    }
}

T get(i64 x, i64 y, SegNode* node = nullptr, i64 l = 0,  i64 r = 0){

    if(node == nullptr){
        node = root;
        r = n;
    }

    if(x <= l && r <= y)
        return node->val;

    if(r <= x || y <= l)
        return op;

    T val_l = op, val_r = op;
    i64 mid = (l + r) >> 1;

    if(node->l != nullptr)
        val_l = f(val_l, get(x, y, node->l, l, mid));
    if(node->r != nullptr)
        val_r = f(get(x, y, node->r, mid, r), val_r);
```

```
        return f(val_l, val_r);
    }

};
```

HeavyLightDecomposition

```
class HeavyLightDecomposition{
public:
    int n;
    vector<vector<int>> g;
    vector<int> rev, in, out, nxt, rin, size, depth;

    HeavyLightDecomposition(vector<vector<int>>& inp) :
        n(inp.size()),
        g(n),
        rev(n, 0),
        in(n, 0),
        out(n, 0),
        nxt(n, 0),
        rin(n, 0),
        size(n, 0),
        depth(n, -1)
    {

        function<void(int, int)> dfs_ed = [&](int pos, int dep){
            depth[pos]=dep;
            for(auto ed : inp[pos])
                if(depth[ed] == -1){
                    g[pos].emplace_back(ed);
                    rev[ed] = pos;
                    dfs_ed(ed, dep + 1);
                }
        };
        dfs_ed(0, 0);

        function<void(int)> dfs_sz = [&](int v){
            size[v] = 1;
            for(auto &u: g[v]){
                dfs_sz(u);
                size[v] += size[u];
                if(size[u] > size[g[v][0]])
                    swap(u, g[v][0]);
            }
        };
        dfs_sz(0);

        int t = 0;

        function<void(int)> dfs_hld = [&](int v){
            in[v] = t++;
            rin[in[v]] = v;
            for(auto u: g[v]){
                nxt[u] = (u == g[v][0] ? nxt[v] : u);
                dfs_hld(u);
            }
            out[v] = t;
        };
        dfs_hld(0);
    }

    pair<int,int> subtree(int x){
        return make_pair(in[x], out[x]);
    }

    vector<int> subtree_path(int x){
        return vector<int>(next(rin.begin(), in[x]), next(rin.begin(), out[x]));
    }

    pair<int,int> subsegment(int x){
        return make_pair(in[nxt[x]], in[x] + 1);
    }

    vector<int> subsegment_path(int x){
```

```
        return vector<int>(next(rin.begin(), in[nxt[x]]), next(rin.begin(), in[x] + 1));
    }

    vector<pair<int,int>> root_path_query(int x){
        vector<pair<int,int>> ret;
        ret.emplace_back(subsegment(x));
        while(ret.back().first)
            ret.emplace_back(subsegment(rev[rin[ret.back().first]]));

        return ret;
    }

    int lca(int x, int y){

        int sx = rin[subsegment(x).first];
        int sy = rin[subsegment(y).first];
        while(sx != sy){
            if(depth[sx] > depth[sy])
                x = rev[sx];
            else
                y = rev[sy];
            sx = rin[subsegment(x).first];
            sy = rin[subsegment(y).first];
        }

        return depth[x] < depth[y] ? x : y;
    }

    pair<vector<pair<int,int>>,vector<pair<int,int>>> two_point_path(i64 x, i64 y){
        i64 z = lca(x, y);
        pair<int,int> z_par = subsegment(z);

        vector<pair<int,int>> ret_x;
        ret_x.emplace_back(subsegment(x));

        while(ret_x.back().first != z_par.first)
            ret_x.emplace_back(subsegment(rev[rin[ret_x.back().first]]));

        ret_x.back().first = in[z];

        vector<pair<int,int>> ret_y;
        ret_y.emplace_back(subsegment(y));

        while(ret_y.back().first != z_par.first)
            ret_y.emplace_back(subsegment(rev[rin[ret_y.back().first]]));

        ret_y.back().first = in[z] + 1;

        return make_pair(ret_x, ret_y);
    }


};
```

LazySegmentTree

```
template<typename T, typename U>
struct Segtree{
    int n;
    T op_t;
    U op_u;
    vector<T> elm;
    vector<U> lazy;
    vector<int> length;
    function<T(T, T)> f;
    function<T(T, U, int)> g;
    function<U(U, U)> h;

    Segtree(int n, T init, function<T(T, T)> f, function<T(T, U, int)> g, function<U(U, U)> h,
            T op_t = T(), U op_u = U()) :
        n(n),
        op_t(op_t),
        op_u(op_u),
        elm(2 * n, init),
```

```
    lazy(2 * n, op_u),
    length(2 * n, 0),
    f(f),
    g(g),
    h(h)
{
    for(int i = n - 1; i > 0; --i){
        elm[i] = f(elm[2 * i], elm[2 * i + 1]);
        length[i] = length[2 * i] + 1;
    }
}

Segtree(int n, vector<T> init, function<T(T, T)> f, function<T(T, U, int)> g,
        function<U(U, U)> h, T op_t = T(), U op_u = U()) :
    n(n),
    op_t(op_t),
    op_u(op_u),
    elm(2 * n),
    lazy(2 * n, op_u),
    length(2 * n, 0),
    f(f),
    g(g),
    h(h)
{
    for(int i = 0; i < n; ++i)
        elm[i + n] = init[i];

    for(int i = n - 1; i > 0; --i){
        elm[i] = f(elm[2 * i], elm[2 * i + 1]);
        length[i] = length[2 * i] + 1;
    }
}

vector<int> get_list(int x, int y){

    vector<int> ret_list;
    for(x += n, y += n - 1; x; x >>= 1, y >>= 1){
        ret_list.emplace_back(x);
        if(x != y)
            ret_list.emplace_back(y);
    }

    return ret_list;
}

void eval(int x){

    elm[x] = g(elm[x], lazy[x], 1 << length[x]);
    if(x < n){
        lazy[2 * x] = h(lazy[2 * x], lazy[x]);
        lazy[2 * x + 1] = h(lazy[2 * x + 1], lazy[x]);
    }
    lazy[x] = op_u;
}

void update(int x, int y, U val){

    vector<int> index_list = get_list(x, y);
    for(int i = index_list.size() - 1; i >= 0; --i)
        eval(index_list[i]);

    for(x += n, y += n - 1; x <= y; x >>= 1, y >>= 1){
        if(x & 1){
            lazy[x] = h(lazy[x], val);
            eval(x++);
        }
        if(!(y & 1)){
            lazy[y] = h(lazy[y], val);
            eval(y--);
        }
    }

    for(auto index : index_list){
        if(index < n){
            eval(2 * index);
            eval(2 * index + 1);
            elm[index] = f(elm[2 * index], elm[2 * index + 1]);
```

```
            }
        }
    }

    T get(int x, int y){

        vector<int> index_list = get_list(x, y);
        for(int i = index_list.size() - 1; i >= 0; --i)
            eval(index_list[i]);

        T l = op_t, r = op_t;
        for(x += n, y += n - 1; x <= y; x >>= 1, y >>= 1){
            if(x & 1){
                eval(x);
                l = f(l, elm[x++]);
            }
            if(!(y & 1)){
                eval(y);
                r = f(elm[y--], r);
            }
        }
        return f(l, r);
    }
};
```

Matrix

```
template <typename T>
struct Matrix{
    int h, w;
    vector<T> v;

    Matrix() : h(1), w(1), v(1, 1){}
    Matrix(int n){*this = makeUnit(n);}
    Matrix(int h, int w) : h(h), w(w), v(h * w, 0){}

    Matrix(vector<vector<T>> v_) : h(v_.size()), w(v_[0].size()), v(h * w){
        for(int i = 0; i < h; ++i)
            for(int j = 0; j < w; ++j)
                v[i * w + j] = v_[i][j];
    }

    static Matrix makeUnit(int n){
        Matrix mat(n, n);
        for(int i = 0; i < n; ++i)
            mat.at(i, i) = 1;
        return mat;
    }

    T& at(int i, int j){
        assert(0 <= i && i <= h && 0 <= j && j < w);
        return v[i * h + j];
    };

    Matrix pow(i64 x){
        assert(h == w);
        auto mat = x & 1 ? *this : makeUnit(h);
        auto u = *this;
        while(u = u * u, x >>= 1)
            if(x & 1)
                mat *= u;
        return mat;
    }

    Matrix& operator+=(const Matrix& mat){
        assert(h == mat.h && w == mat.w);
        for(int i = 0; i < h * w; ++i)
            v[i] += mat.v[i];
        return *this;
    }

    Matrix& operator-=(const Matrix& mat){
        assert(h == mat.h && w == mat.w);
        for(int i = 0; i < h * w; ++i)
```

```
            v[i] -= mat.v[i];
        return *this;
    }

    Matrix& operator%=(const T mod){
        for(int i = 0; i < h * w; ++i)
            v[i] %= mod;
        return *this;
    }

    Matrix operator*(const Matrix& mat){
        assert(w == mat.h);
        Matrix ret(h, mat.w);
        for(int i = 0; i < h; ++i)
            for(int k = 0; k < w; ++k)
                for(int j = 0; j < mat.w; ++j)
                    ret.v[i * mat.w + j] += v[i * w + k] * mat.v[k * mat.w + j];
        return ret;
    }

    Matrix operator+(const Matrix& mat){return Matrix(*this) += mat;}
    Matrix operator-(const Matrix& mat){return Matrix(*this) -= mat;}
    Matrix operator%(const T mod){return Matrix(*this) %= mod;}
    Matrix& operator*=(const Matrix& mat){return *this = *this * mat;}
};
```

ModInt

```
template <i64 mod = MOD>
struct ModInt{
    i64 p;

    ModInt() : p(0){}
    ModInt(i64 x){p = x >= 0 ? x % mod : x + (-x + mod - 1) / mod * mod;}

    ModInt& operator+=(const ModInt& y){p = p + *y - ((p + *y) >= mod ? mod : 0); return *this;}
    ModInt& operator-=(const ModInt& y){p = p - *y + (p - *y < 0 ? mod : 0); return *this;}
    ModInt& operator*=(const ModInt& y){p = (p * *y) % mod; return *this;}
    ModInt& operator%=(const ModInt& y){if(y)p %= *y; return *this;}

    ModInt operator+(const ModInt& y) const{ModInt x = *this; return x += y;}
    ModInt operator-(const ModInt& y) const{ModInt x = *this; return x -= y;}
    ModInt operator*(const ModInt& y) const{ModInt x = *this; return x *= y;}
    ModInt operator%(const ModInt& y) const{ModInt x = *this; return x %= y;}

    friend ostream& operator<<(ostream& stream, const ModInt<mod>& x){
        stream << *x;
        return stream;
    }

    friend ostream& operator>>(ostream& stream, const ModInt<mod>& x){
        stream >> *x;
        return stream;
    }

    ModInt& operator++(){p = (p + 1) % mod; return *this;}
    ModInt& operator--(){p = (p - 1 + mod) % mod; return *this;}

    bool operator==(const ModInt& y) const{return p == *y;}
    bool operator!=(const ModInt& y) const{return p != *y;}

    const i64& operator*() const{return p;}
    i64& operator*(){return p;}

};

using mint = ModInt<>;
```

PersistentDynamicLazySegmentTree

```
template<typename T, typename U>
struct Segtree{

    struct SegNode{
        T val;
        U lazy;

        shared_ptr<SegNode> l;
        shared_ptr<SegNode> r;
        SegNode(T val, U lazy) : val(val), lazy(lazy), l(nullptr), r(nullptr){}
    };

    i64 n;
    shared_ptr<SegNode> nil;
    function<T(T, T)> f;
    function<T(T, U, int)> g;
    function<U(U, U)> h;
    T op_t;
    U op_u;

    shared_ptr<SegNode> root;

    Segtree(int n_, function<T(T, T)> f, function<T(T, U, int)> g, function<U(U, U)> h, T op_t, U op_u) :
    f(f), g(g), h(h), op_t(op_t), op_u(op_u){
        for(n = 1; n < n_; n <<= 1);
        root = make_shared<SegNode>(op_t, op_u);
    }

    void eval(shared_ptr<SegNode> node, i64 len, bool make = true){
        node->val = g(node->val, node->lazy, len);
        if(make){
            node->l = node->l ? make_shared<SegNode>(*node->l) : make_shared<SegNode>(op_t, op_u);
            node->r = node->r ? make_shared<SegNode>(*node->r) : make_shared<SegNode>(op_t, op_u);
        }
        node->l->lazy = h(node->l->lazy, node->lazy);
        node->r->lazy = h(node->r->lazy, node->lazy);
        node->lazy = op_u;
    }

    // if root -> make new node      -> eval(make child)
    void update(i64 x, i64 y, U val, shared_ptr<SegNode> node = nullptr, i64 l = -1, i64 r = -1){
        bool root_flag = (node == nullptr);
        if(root_flag){
            root = make_shared<SegNode>(*root);
            node = root;
        }
        if(l == -1){
            l = 0;
            r = n;
        }
        eval(node, r - l);
        if(r <= x || y <= l)
            return ;
        if(x <= l && r <= y){
            node->lazy = h(node->lazy, val);
            eval(node, r - l, false);
        }else{
            eval(node, r - l);
            i64 mid = (l + r) >> 1;
            update(x, y, val, node->l, l, mid);
            update(x, y, val, node->r, mid, r);
            node->val = f(node->l->val, node->r->val);
        }
        return ;
    }

    T get(i64 x, i64 y, shared_ptr<SegNode> node = nullptr, i64 l = -1, i64 r = -1){
        bool root_flag = (node == nullptr);
        if(root_flag){
            root = make_shared<SegNode>(*root);
            node = root;
        }
        if(l == -1){
            l = 0;
            r = n;
        }
```

```
        if(r <= x || y <= l)
            return op_t;
        eval(node, r - l);
        if(x <= l && r <= y)
            return node->val;

        i64 val_l = op_t, val_r = op_t;
        i64 mid = (l + r) >> 1;

        if(node->l)
            val_l = get(x, y, node->l, l, mid);
        if(node->r)
            val_r = get(x, y, node->r, mid, r);

        return f(val_l, val_r);
    }

};
```

RectangleSum

```
struct RectangleSum{//O(HW)で初期化してO(1)で長方形の和を出す(半開区間)
    vector<vector<i64>> sum;
    int h, w;
    RectangleSum(vector<vector<i64>>& v) :
        h(v.size()),
        w(v[0].size()),
        sum(v)
    {}

    // 半開区間で設定する事に注意する
    void set(int sx, int sy, int ex, int ey, i64 val){
        sum[sx][sy] += val;
        sum[sx][ey] -= val;
        sum[ex][sy] -= val;
        sum[ex][ey] += val;
    }

    void run(){

        for(int i = 0; i < h + 1; ++i)
            for(int j = 0; j < w; ++j)
                sum[i][j + 1] += sum[i][j];

        for(int j = 0; j < w + 1; ++j)
            for(int i = 0; i < h; ++i)
                sum[i + 1][j] += sum[i][j];
    }

    i64 getSum(int sx, int sy, int ex, int ey){
        return sum[ex][ey] + sum[sx][sy] - sum[sx][ey] - sum[ex][sy];
    }
};
```

RollingHash

```
template <i64 mod1 = MOD, i64 mod2 = MOD + 2, i64 base = 10007, typename T = string>
struct RollingHash{

    using mint1 = ModInt<mod1>;
    using mint2 = ModInt<mod2>;
    using pair_type = pair<mint1, mint2>;
    int len;
    std::vector<pair_type> v;
    static std::vector<pair_type> power, inv;

    RollingHash(T s) :
    len(s.size())
    {
        v.assign(1, make_pair(mint1(0), mint2(0)));
```

```
        for(int i = 0; i < len; ++i){
            auto c = s[i];
            v.emplace_back(v.back().first + power[i].first * c,
                           v.back().second + power[i].second * c);
            if(static_cast<int>(power.size()) == i + 1){
                power.emplace_back(power.back().first * base,
                                   power.back().second * base);
                inv.emplace_back(mpow<mint1>(power.back().first, mod1 - 2),
                                 mpow<mint2>(power.back().second, mod2 - 2));
            }
        }
    };

    pair_type get(int l = 0, int r = -1){
        if(r == -1)
            r = len;
        assert(l <= r);
        assert(r <= len);
        auto l_cut = make_pair(v[r].first - v[l].first,
                               v[r].second - v[l].second);
        return make_pair(l_cut.first * inv[l].first,
                         l_cut.second * inv[l].second);
    }

    pair_type connect(pair_type l, pair_type r, int l_len){
        return make_pair(l.first + power[l_len].first * r.first,
                         l.second + power[l_len].second * r.second);
    }
};

using RH = RollingHash<MOD, MOD + 2, 10007>;
template<>
vector<pair<ModInt<MOD>, ModInt<MOD + 2>>> RH::power = {make_pair(ModInt<MOD>(1), ModInt<MOD + 2>(1))};
template<>
vector<pair<ModInt<MOD>, ModInt<MOD + 2>>> RH::inv = {make_pair(ModInt<MOD>(1), ModInt<MOD + 2>(1))};
```

SegmentTree

```
template<typename T>
struct Segtree{
    int n;
    T op;
    vector<T> elm;
    function<T(T, T)> f;

    Segtree(int n, T init, function<T(T, T)> f, T op = T()) :
        n(n),
        op(op),
        elm(2 * n, init),
        f(f)
    {
        for(int i = n - 1; i >= 1; --i)
            elm[i] = f(elm[2 * i], elm[2 * i + 1]);
    }

    Segtree(int n, vector<T> init, function<T(T, T)> f, T op = T()) :
        n(n),
        op(op),
        elm(2 * n),
        f(f)
    {
        for(int i = 0; i < n; ++i)
            elm[i + n] = init[i];
        for(int i = n - 1; i >= 1; --i)
            elm[i] = f(elm[2 * i], elm[2 * i + 1]);
    }

    void set(int x, T val){
        x += n;
        elm[x] = val;
        while(x >>= 1)
            elm[x] = f(elm[2 * x], elm[2 * x + 1]);
    }
```

```
    void update(int x, T val){
        x += n;
        elm[x] = f(val, elm[x]);
        while(x >>= 1)
            elm[x] = f(elm[2 * x], elm[2 * x + 1]);
    }

    T get(int x, int y) const{
        T l = op, r = op;
        for(x += n, y += n - 1; x <= y; x >>= 1, y >>= 1){
            if(x & 1)
                l = f(l, elm[x++]);
            if(!(y & 1))
                r = f(elm[y--], r);
        }
        return f(l, r);
    }
};
```

Treap

```
template <typename T, typename U = int>
struct Node{

    using np = Node<T, U>*;

    static np nil;

    T val;
    U lazy;
    uint32_t pri;

    int size;
    T sum;

    np l = nil;
    np r = nil;

    Node(T v, U OU = U()) : val(v), lazy(OU), pri(rndpri()), size(1), sum(v), l(nil), r(nil){}
    Node(T v, U OU, uint32_t p) : val(v), lazy(OU), pri(p), size(1), sum(v), l(nil), r(nil){}

    static uint32_t rndpri() {
        static uint32_t x = 123456789, y = 362436069, z = 521288629, w = time(0);
        uint32_t t = x ^ (x << 11);
        x = y;
        y = z;
        z = w;
        w = (w ^ (w >> 19)) ^ (t ^ (t >> 8));
        return max<uint32_t>(1, w & 0x3FFFFFFF);
    }
};


template <typename T, typename U = int>
class Treap{

    using nt = Node<T, U>;
    using np = nt*;
    using F = function<T(T, T)>;
    using G = function<T(T, U, int)>;
    using H = function<U(U, U)>;

public:

    np root;
    bool is_list;
    F f;
    G g;
    H h;
    T OT;
    U OU;

    Treap(bool is_list, F f, G g, H h, T OT, U OU) : root(nt::nil), is_list(is_list),
    f(f), g(g), h(h), OT(OT), OU(OU){}
```

```
Treap(T val, bool is_list, F f, G g, H h, T OT, U OU) : root(new nt(val)), is_list(is_list),
f(f), g(g), h(h), OT(OT), OU(OU){}

// 配列で初期化する
Treap(vector<T> v, bool is_list, F f, G g, H h, T OT, U OU) : root(nt::nil), is_list(is_list),
f(f), g(g), h(h), OT(OT), OU(OU){
    for(auto& xx : v)
        root = _merge(root, new nt(xx, OU));
}

static Treap make(bool is_list, F f = [](T x, T){return x;}, T OT = T(),
G g = [](auto x, auto, auto){return x;}, H h = [](auto x, auto){return x;}, U OU = U()){
    assert(nt::nil != nullptr);
    return Treap(is_list, f, g, h, OT, OU);
}

static Treap make(T val, bool is_list, F f = [](auto x, auto){return x;}, T OT = T(),
G g = [](auto x, auto, auto){return x;}, H h = [](auto x, auto){return x;}, U OU = U()){
    assert(nt::nil != nullptr);
    return Treap(val, is_list, f, g, h, OT, OU);
}

static Treap make(vector<T> val, bool is_list, F f = [](auto x, auto){return x;}, T OT = T(),
G g = [](auto x, auto, auto){return x;}, H h = [](auto x, auto){return x;}, U OU = U()){
    assert(nt::nil != nullptr);
    return Treap(val, is_list, f, g, h, OT, OU);
}

~Treap(){
    clear();
    if(root != nt::nil)
        delete root;
}

int _size(np x){return x == nt::nil ? 0 : x->size;}
T _sum(np x){return x == nt::nil ? OT : x->sum;}

np _update(np x){

    if(x == nt::nil)
        return x;

    if(is_list){
        _push(x);
        _push(x->l);
        _push(x->r);
    }

            x->sum = f(f(_sum(x->l), x->val), _sum(x->r));
            x->size = _size(x->l) + _size(x->r) + 1;
    return x;
}

void _push(np x){
    if(x->lazy == OU)
        return ;

    x->sum = g(x->sum, x->lazy, x->size);
    x->val = g(x->val, x->lazy, 1);

    if(x->l != nt::nil)
        x->l->lazy = h(x->l->lazy, x->lazy);
    if(x->r != nt::nil)
        x->r->lazy = h(x->r->lazy, x->lazy);

    x->lazy = OU;

}

np _merge(np l, np r){
    if(l == nt::nil || r ==nt::nil)
        return l == nt::nil ? r : l;

    if(l->pri > r->pri){
        l->r = _merge(l->r, r);
        return _update(l);
```

```
    }else{
        r->l = _merge(l, r->l);
        return _update(r);
    }
}

pair<np,np> _split(np x, int k){
    if(x == nt::nil)
        return make_pair(nt::nil, nt::nil);

    assert(0 <= k && k <= _size(x));

    if(k <= _size(x->l)){
        pair<np, np> s = _split(x->l, k);
        x->l = s.second;
        return make_pair(s.first, _update(x));

    }else{
        pair<np, np> s = _split(x->r, k - _size(x->l) - 1);
        x->r = s.first;
        return make_pair(_update(x), s.second);
    }
}

np _insert(np x, int k, T val){
    np l, r;
    tie(l, r) = _split(x, k);
    return _merge(_merge(l, new nt(val, OU)), r);
}

np _erase(np x, int k){
    np l, r, m;
    tie(l, r) = _split(x, k);
    tie(m, r) = _split(r, 1);
    if(m != nt::nil)
        delete m;
    return _merge(l, r);
}

void _set(np x, int k, T val){
    _update(x);

    if(k < _size(x->l))
        _set(x->l, k, val);
    else if(_size(x->l) == k)
        x->val = val;
    else
        _set(x->r, k - _size(x->l) - 1, val);

    _update(x);
}

void _add(np x, int l, int r, U val){
    assert(is_list);
    _update(x);

    if(x == nt::nil)
        return ;
    l = max(l, 0);
    r = min(r, _size(x));

    int sl = _size(x->l);

    if(l >= r)
        return ;

    if (l == 0 && r == _size(x)){
        x->lazy = h(x->lazy, val);
    }
    else{
        if(l <= sl && sl < r)
            x->val = g(x->val, val, 1);

        _add(x->l, l, r, val);
        _add(x->r, l - sl - 1, r - sl - 1, val);
    }
```

```
    _update(x);
}

np _getnode(np x, int k){

    _update(x);

    assert(0 <= k && k < _size(x));

    if(k < _size(x->l))
        return _getnode(x->l, k);
    else if(_size(x->l) == k)
        return x;
    else
        return _getnode(x->r, k - _size(x->l) - 1);
}

T _get(np x, int k){
    return _getnode(x, k)->val;
}

T _rangesum(np x, int l, int r){
    _update(x);

    l = max(l, 0);
    r = min(r, _size(x));
    if(l >= r)
        return OT;
    if(l == 0 && r == _size(x))
        return _sum(x);

    int sl = _size(x->l);
    T ret = (l <= sl && sl < r ? x->val : OT);
    ret = f(_rangesum(x->l, l, r), ret);
    ret = f(ret, _rangesum(x->r, l - sl - 1, r - sl - 1));
    return ret;
}

int _lowerbound(np x, T val){
    _update(x);

    if(x == nt::nil)
        return 0;
    if(val <= x->val)
        return _lowerbound(x->l, val);
    else
        return _lowerbound(x->r,val) + _size(x->l) + 1;
}

int _upperbound(np x, T val){
    _update(x);

    if(x == nt::nil)
        return 0;
    if(val < x->val)
        return _upperbound(x->l, val);
    else
        return _upperbound(x->r,val) + _size(x->l) + 1;
}

np _insert(np x, T val){
    return _insert(x, _lowerbound(x, val), val);
}

void _clear(np x){
    if(x->l != nt::nil){
        _clear(x->l);
        delete(x->l);
        x->l = nt::nil;
    }
    if(x->r != nt::nil){
        _clear(x->r);
        delete(x->r);
        x->r = nt::nil;
    }
}
```

```
void push_front(T val){
    root = _merge(new nt(val, OU), root);
}

void push_back(T val){
    root = _merge(root, new nt(val, OU));
}

void pop_front(){
    root = _split(root, 1).second;
}

void pop_back(){
    root = _split(root, _size(root) - 1).first;
}

// [0, k)と[k, size)に分割して, [k, size)側を返す
Treap split_left(int k){
    np p;
    tie(root, p) = _split(root, k);
    return decltype(this)(f, g, h, p);
}

// [0, k)と[k, size)に分割して, [0, k)側を返す
Treap split_right(int k){
    np p;
    tie(p, root) = _split(root, k);
    return decltype(this)(f, g, h, p);
}

// rootを含めたサイズの出力
int size(){
    return (root == nt::nil ? 0 : root->size);
}

// k番目への代入
void set(int k, T val){
    return _set(root, k, val);
}

// k番目への加算
void add(int k, U val){
    assert(is_list);
    return _add(root, k, k + 1, val);
}

// [l, r)への一様加算
void add(int l, int r, U val){
    assert(is_list);
    return _add(root, l, r, val);
}

// k番目の取得
T get(int k){
    return _get(root, k);
}

// [l, r)の総和 (同様の実装でRMQ等も可能)
T get(int l, int r){
    return _rangesum(root, l, r);
}

// k番目への挿入
void insert(int k, T val){
    assert(is_list);
    root = _insert(root, k, val);
}

// 適切な位置への挿入
void insert(T val){
    root = _insert(root, val);
}

// val <= get(k) となるような最小のk
int lowerbound(T val){
    return _lowerbound(root, val);
}
```

```
    // val < get(k) となるような最小のk
    int upperbound(T val){
        return _upperbound(root, val);
    }

    // k番目の要素削除
    void erase(int k){
        root = _erase(root, k);
    }

    // 要素の全削除
    void clear(){
        if(root != nt::nil){
            _clear(root);
            delete(root);
            root = nt::nil;
        }
    }
};

const i64 val = 0;
const i64 op = -1e9;
using node_type = Node<i64, i64>;
template<> node_type* node_type::nil = new node_type(0, op, 0);
```

TrieTree

```
template <int size = 26, int start = 'a'>
struct Trie{
    struct Node{
        // 値, prefixに含む文字列の数, 文字列の数
        int val, len, cnt, exist_cnt;
        // 子のindex, 子の(indexの)一覧
        vector<int> next, exist;
        Node(int val = -1, int len = 0, bool back = false) : val(val), len(len), cnt(0),
        exist_cnt(back), next(size, -1){}
    };

    vector<Node> nodes;
    Trie() : nodes(1){}

    int insert(string& s, int str_index = 0){
        int pos = 0, idx = str_index;
        while(idx != s.size()){
            ++nodes[pos].cnt;
            int c = s[idx] - start;
            assert(c < size);

            if(nodes[pos].next[c] == -1){
                nodes[pos].next[c] = nodes.size();
                nodes[pos].exist.emplace_back(nodes.size());
                nodes.emplace_back(c, nodes[pos].len + 1);
            }
            pos = nodes[pos].next[c];
            ++idx;
        }
        ++nodes[pos].cnt;
        ++nodes[pos].exist_cnt;
        return pos;
    }

    // (sの部分文字列, s, sを部分文字列に含む文字列)に対して関数を実行する
    // ラムダ内でtrie.nodes[idx].exist_cntを判定する事で, 挿入された文字列そのもの以外判定しなくなる
    void query(string& s, function<void(int, string&)> f, bool from_prefix, bool correct,
               bool to_prefix, int str_index = 0){
        int pos = 0, idx = str_index;
        string str;
        while(idx != s.size()){
            if(from_prefix)
                f(pos, str);
            int c = s[idx] - start;
            assert(c < size);
```

```
            if(nodes[pos].next[c] == -1)
                return ;
            pos = nodes[pos].next[c];
            str += static_cast<char>(nodes[pos].val + start);
            ++idx;
        }
        if(correct)
            f(pos, str);
        function<void(int)> dfs = [&](int pos){
            for(auto& next : nodes[pos].exist){
                char c = nodes[next].val + start;
                if(to_prefix)
                    f(pos, str);
                str += c;
                dfs(next);
                str.pop_back();
            }
        };
        dfs(pos);
    }
};
```

UnionFind

```
struct UnionFind{
    vector<int> par;
    int count;
    UnionFind(int n) : par(n, -1), count(0){}
    int Find(int x){return par[x] < 0 ? x : Find(par[x]);}
    int Size(int x){return par[x] < 0 ? -par[x] : Size(par[x]);}
    bool Unite(int x, int y){
        x = Find(x);
        y = Find(y);
        if(x == y)
            return false;
        if(par[x] > par[y])
            swap(x, y);
        par[x] += par[y];
        par[y] = x;
        return ++count;
    }
};
```