

.bashrc	1	structures/hull_online.h	45
.vimrc	2	structures/max_queue.h	46
template.cpp	3	structures/pairing_heap.h	47
geometry/convex_hull.h	4	structures/rmq.h	48
geometry/convex_hull_dist.h	5	structures/segment_tree.h	49
geometry/convex_hull_sum.h	6	structures/segment_tree_beats.h	50
geometry/line2.h	7	structures/segment_tree_point.h	51
geometry/rmst.h	8	structures/treap.h	52
geometry/segment2.h	9	structures/ext/hash_table.h	53
geometry/vec2.h	10	structures/ext/rope.h	54
graphs/2sat.h	11	structures/ext/tree.h	55
graphs/bellman_inequalities.h	12	structures/ext/trie.h	56
graphs/bridges_online.h	13	text/aho_corasick.h	57
graphs/dense_dfs.h	14	text/kmp.h	58
graphs/edmonds_karp.h	15	text/kmr.h	59
graphs/push_relabel.h	16	text/lcp.h	60
graphs/scc.h	17	text/lyndon_factorization.h	61
graphs/turbo_matching.h	18	text/main_lorentz.h	62
math/bit_gauss.h	19	text/manacher.h	63
math/bit_matrix.h	20	text/min_rotation.h	64
math/crt.h	21	text/palindromic_tree.h	65
math/discrete_logarithm.h	22	text/suffix_array_linear.h	66
math/fft_complex.h	23	text/z_function.h	67
math/fft_mod.h	24	trees/centroid_decomp.h	68
math/ gauss.h	25	trees/heavylight_decomp.h	69
math/miller_rabin.h	26	trees/lca.h	70
math/modinv_precompute.h	27	trees/link_cut_tree.h	71
math/modular.h	28	util/bit_hacks.h	72
math/modular64.h	29	util/bump_alloc.h	73
math/montgomery.h	30	util/compress_vec.h	74
math/nimber.h	31	util/inversion_vector.h	75
math/phi_large.h	32	util/longest_increasing_sub.h	76
math/phi_precompute.h	33	util/max_rects.h	77
math/pi_large_precomp.h	34	util/mo.h	78
math/pollard_rho.h	35	util/parallel_binsearch.h	79
math/polynomial_interp.h	36	util/radix_sort.h	80
math/sieve.h	37		
math/sieve_factors.h	38		
math/sieve_segmented.h	39		
structures/bitset_plus.h	40		
structures/fenwick_tree.h	41		
structures/fenwick_tree_2d.h	42		
structures/find_union.h	43		
structures/hull_offline.h	44		

## .bashrc 1

```
b() {
    g++ $@ -o $1.e -DLOC -O2 -g -std=c++11 \
        -Wall -Wextra -Wfatal-errors -Wshadow \
        -Wlogical-op -Wconversion -Wfloat-equal
}

d() { b $@ -fsanitize=address,undefined \
        -D_GLIBCXX_DEBUG }

cmp() {
    set -e; $1 $2; $1 $3; $1 $4
    for ((;)) {
        ./$4.e > gen.in;          echo -n 0
        ./$2.e < gen.in > p1.out; echo -n 1
        ./$3.e < gen.in > p2.out; echo -n 2
        diff p1.out p2.out;      echo -n Y
    }
}

# Other flags:
# -Wformat=2 -Wshift-overflow=2 -Wcast-qual
# -Wcast-align -Wduplicated-cond
# -D_GLIBCXX_DEBUG_PEDANTIC -D_FORTIFY_SOURCE=2
# -fno-sanitize-recover -fstack-protector
```

## .vimrc 2

```
se ai aw cin cul ic is nocp nohls nu rnu sc scs
se bg=dark so=7 sw=4 ttm=9 ts=4
sy on
colo delek
```

## template.cpp 3

```
#include <bits/stdc++.h>
using namespace std;

using ll = long long;
using Vi = vector<int>;
using Pii = pair<int,int>;

#define pb push_back
#define x first
#define y second

#define rep(i,b,e) for(int i=(b); i<(e); i++)
#define each(a,x) for(auto& a : (x))
#define all(x) (x).begin(), (x).end()
#define sz(x) int((x).size())

int main() {
    cin.sync_with_stdio(0); cin.tie(0);
    cout << fixed << setprecision(18);
    return 0;
}

// > Debug printer

#define tem template<class t,class u,class...w>
#define pri(x,y)tem auto operator<<(t&o,u a) \
    ->decltype(x,o) { o << y; return o; }

pri(a.print(), "{"; a.print(); o << "}")
pri(a.y, "{ " << a.x << ", " << a.y << "}")

pri(all(a), "{"; auto d=""; for (auto i : a)
```

```
(o << d << i, d = ", "); o << "]" )

void DD(...) {}
tem void DD(t s, u a, w... k) {
    int b = 44;
    while (*s && *s != b) {
        b += (*s == 40 ? 50 : *s == 41 ? -50 : 0);
        cerr << *s++;
    }
    cerr << ": " << a << *s++; DD(s, k...);
}

tem vector<t> span(const t* a, u n) {
    return {a, a+n};
}

#ifdef LOC
#define deb(...) (DD("#", #__VA_ARGS__, \
    __LINE__, __VA_ARGS__), cerr << endl)
#else
#define deb(...)
#endif

#define DBP(...) void print() { \
    DD(__VA_ARGS__, __VA_ARGS__); }

// > Utils

// #pragma GCC optimize("Ofast,unroll-loops,
// no-stack-protector")
// #pragma GCC target("avx")

// while (clock() < time*CLOCKS_PER_SEC)
// using namespace rel_ops;

// Return smallest k such that 2^k > n
// Undefined for n = 0!
int uplg(int n) { return 32-__builtin_clz(n); }
int uplg(ll n){ return 64-__builtin_clzll(n); }

// Compare with certain epsilon (branchless)
// Returns -1 if a < b; 1 if a > b; 0 if equal
// a and b are assumed equal if |a-b| <= eps
int cmp(double a, double b, double eps=1e-10) {
    return (a > b+eps) - (a+eps < b);
}

geometry/convex_hull.h 4
#include "vec2.h"

// Translate points such that lower-left point
// is (0, 0). Returns old point location; O(n)
vec2 normPos(vector<vec2>& points) {
    auto q = points[0].yxPair();
    each(p, points) q = min(q, p.yxPair());
    vec2 ret{q.y, q.x};
    each(p, points) p = p-ret;
    return ret;
}

// Find convex hull of points; time: O(n lg n)
// Points are returned counter-clockwise.
vector<vec2> convexHull(vector<vec2> points) {
    vec2 pivot = normPos(points);
    sort(all(points));
    vector<vec2> hull;
```

```
each(p, points) {
    while (sz(hull) >= 2) {
        vec2 a = hull.back() - hull[sz(hull)-2];
        vec2 b = p - hull.back();
        if (a.cross(b) > 0) break;
        hull.pop_back();
    }
    hull.pb(p);
}

// Translate back, optional
each(p, hull) p = p+pivot;
return hull;
}
```

## geometry/convex\_hull\_dist.h 5

```
#include "vec2.h"

// Check if p is inside convex polygon. Hull
// must be given in counter-clockwise order.
// Returns 2 if inside, 1 if on border,
// 0 if outside; time: O(n)
int insideHull(vector<vec2>& hull, vec2 p) {
    int ret = 1;
    rep(i, 0, sz(hull)) {
        auto v = hull[(i+1)%sz(hull)] - hull[i];
        auto t = v.cross(p-hull[i]);
        ret = min(ret, cmp(t, 0)); // For doubles
        //ret = min(ret, (t>0) - (t<0)); // Ints
    }
    return int(max(ret+1, 0));
}

#include "segment2.h"

// Get distance from point to hull; time: O(n)
double hullDist(vector<vec2>& hull, vec2 p) {
    if (insideHull(hull, p)) return 0;
    double ret = 1e30;
    rep(i, 0, sz(hull)) {
        seg2 seg{hull[(i+1)%sz(hull)], hull[i]};
        ret = min(ret, seg.distTo(p));
    }
    return ret;
}

// Compare distance from point to hull
// with sqrt(d2); time: O(n)
// -1 if smaller, 0 if equal, 1 if greater
int cmpHullDist(vector<vec2>& hull,
    vec2 p, ll d2) {
    if (insideHull(hull,p)) return (d2<0)-(d2>0);
    int ret = 1;
    rep(i, 0, sz(hull)) {
        seg2 seg{hull[(i+1)%sz(hull)], hull[i]};
        ret = min(ret, seg.cmpDistTo(p, d2));
    }
    return ret;
}
```

## geometry/convex\_hull\_sum.h 6

```
#include "vec2.h"

// Get edge sequence for given polygon
// starting from lower-left vertex; time: O(n)
```

```
// Returns start position.
vec2 edgeSeq(vector<vec2> points,
    vector<vec2>& edges) {
    int i = 0, n = sz(points);
    rep(j, 0, n) {
        if (points[i].yxPair()>points[j].yxPair())
            i = j;
    }
    rep(j, 0, n) edges.pb(points[(i+j+1)%n] -
        points[(i+j)%n]);
    return points[i];
}

// Minkowski sum of given convex polygons.
// Vertices are required to be in
// counter-clockwise order; time: O(n+m)
vector<vec2> hullSum(vector<vec2> A,
    vector<vec2> B) {
    vector<vec2> sum, e1, e2, es(sz(A) + sz(B));
    vec2 pivot = edgeSeq(A, e1) + edgeSeq(B, e2);
    merge(all(e1), all(e2), es.begin());

    sum.pb(pivot);
    each(e, es) sum.pb(sum.back() + e);
    sum.pop_back();
    return sum;
}
```

## geometry/line2.h 7

```
#include "vec2.h"

// 2D line structure; PARTIALLY TESTED

// Base class of versions for ints and doubles
template<class T, class P, class S>
struct bline2 { // norm*point == off
    P norm; // Normal vector [A; B]
    T off; // Offset (C parameter of equation)

    // Line through 2 points
    static S through(P a, P b) {
        return { (b-a).perp(), b.cross(a) };
    }

    // Parallel line through point
    static S parallel(P a, S b) {
        return { b.norm, b.norm.dot(a) };
    }

    // Perpendicular line through point
    static S perp(P a, S b) {
        return { b.norm.perp(), b.norm.cross(a) };
    }

    // Distance from point to line
    double distTo(P a) {
        return fabs(norm.dot(a)-off) / norm.len();
    }
};

// Version for integer coordinates (long long)
struct line2i : bline2<ll, vec2i, line2i> {
    line2i() : bline2({}, 0) {}
    line2i(vec2i n, ll c) : bline2{n, c} {}

    int side(vec2i a) {
```

```

    ll d = norm.dot(a);
    return (d > off) - (d < off);
}
};

// Version for double coordinates
// Requires cmp() from template
struct line2d : bline2<double, vec2d, line2d> {
    line2d() : bline2({}, 0) {}
    line2d(vec2d n, double c) : bline2{n, c} {}

    int side(vec2d a) {
        return cmp(norm.dot(a), off);
    }

    bool intersect(line2d a, vec2d& out) {
        double d = norm.cross(a.norm);
        if (cmp(d, 0) == 0) return false;
        out = (norm*a.off-a.norm*off).perp() / d;
        return true;
    }
};

using line2 = line2d;
```

## geometry/rmst.h

8

```

#include "../structures/find_union.h"

// Rectilinear Minimum Spanning Tree
// (MST in Manhattan metric); time: O(n lg n)
// Returns MST weight. Outputs spanning tree
// to G, vertex indices match point indices.
// Edge in G is pair (target, weight).
ll rmst(vector<Pii>& points,
        vector<vector<Pii>&& G) {
    int n = sz(points);
    vector<pair<int, Pii>> edges;
    vector<Pii> close;
    Vi ord(n), merged(n);
    iota(all(ord), 0);

    function<void(int,int)> octant =
        [&](int begin, int end) {
            if (begin+1 >= end) return;

            int mid = (begin+end) / 2;
            octant(begin, mid);
            octant(mid, end);

            int j = mid;
            Pii best = {INT_MAX, -1};
            merged.clear();

            rep(i, begin, mid) {
                int v = ord[i];
                Pii p = points[v];

                while (j < end) {
                    int e = ord[j];
                    Pii q = points[e];
                    if (q.x-q.y > p.x-p.y) break;
                    best = min(best, make_pair(q.x+q.y, e));
                    merged.pb(e);
                    j++;
                }
            }
        }
    }
```

```

    if (best.y != -1) {
        int alt = best.x-p.x-p.y;
        if (alt < close[v].x)
            close[v] = {alt, best.y};
    }
    merged.pb(v);
}

while (j < end) merged.pb(ord[j++]);
copy(all(merged), ord.begin()+begin);
};

rep(i, 0, 4) {
    rep(j, 0, 2) {
        sort(all(ord), [&](int l, int r) {
            return points[l] < points[r];
        });
        close.assign(n, {INT_MAX, -1});
        octant(0, n);
        rep(k, 0, n) {
            Pii p = close[k];
            if (p.y != -1) edges.pb({p.x, {k, p.y}});
            points[k].x *= -1;
        }
    }
    each(p, points) p = {p.y, -p.x};
}

ll sum = 0;
FAU fau(n);
sort(all(edges));
G.assign(n, {});

each(e, edges) if (fau.join(e.y.x, e.y.y)) {
    sum += e.x;
    G[e.y.x].pb({e.y.y, e.x});
    G[e.y.y].pb({e.y.x, e.x});
}
return sum;
}
```

## geometry/segment2.h

9

```

#include "vec2.h"

// 2D segment structure; NOT HEAVILY TESTED

// Base class of versions for ints and doubles
template<class P, class S> struct bseg2 {
    P a, b; // Endpoints

    // Distance from segment to point
    double distTo(P p) const {
        if ((p-a).dot(b-a) < 0) return (p-a).len();
        if ((p-b).dot(a-b) < 0) return (p-b).len();
        return double(abs((p-a).cross(b-a)))
            / (b-a).len();
    }
};

// Version for integer coordinates (long long)
struct seg2i : bseg2<vec2i, seg2i> {
    seg2i() {}
    seg2i(vec2i c, vec2i d) : bseg2{c, d} {}

    // Check if segment contains point p
    bool contains(vec2i p) {
```

```

    return (a-p).dot(b-p) <= 0 &&
        (a-p).cross(b-p) == 0;
}

// Compare distance to p with sqrt(d2)
// -1 if smaller, 0 if equal, 1 if greater
int cmpDistTo(vec2i p, ll d2) const {
    if ((p-a).dot(b-a) < 0) {
        ll l = (p-a).len2();
        return (l > d2) - (l < d2);
    }
    if ((p-b).dot(a-b) < 0) {
        ll l = (p-b).len2();
        return (l > d2) - (l < d2);
    }
}

ll c = abs((p-a).cross(b-a));
d2 *= (b-a).len2();
return (c*c > d2) - (c*c < d2);
}

// Version for double coordinates
// Requires cmp() from template
struct seg2d : bseg2<vec2d, seg2d> {
    seg2d() {}
    seg2d(vec2d c, vec2d d) : bseg2{c, d} {}

    bool contains(vec2d p) {
        return cmp((a-p).dot(b-p), 0) <= 0 &&
            cmp((a-p).cross(b-p), 0) == 0;
    }
};

using seg2 = seg2d;
```

## geometry/vec2.h

10

```

// 2D point/vector structure; PARTIALLY TESTED

// Base class of versions for ints and doubles
template<class T, class S> struct bvec2 {
    T x, y;
    S operator+(S r) const {return{x+r.x, y+r.y};}
    S operator-(S r) const {return{x-r.x, y-r.y};}
    S operator*(T r) const {return{x*r, y*r};}
    S operator/(T r) const {return{x/r, y/r};}

    T dot(S r) const {return x*r.x+y*r.y;}
    T cross(S r) const {return x*r.y-y*r.x;}
    T len2() const {return x*x+y*y;}
    double len() const {return sqrt(len2());}
    S perp() const {return{-y, x};} //90deg

    pair<T, T> yxPair() const {return{y, x};}

    double angle() const { // [0; 2*PI] CCW from OX
        double a = atan2(y, x);
        return (a < 0 ? a+2*M_PI : a);
    }
};

// Version for integer coordinates (long long)
struct vec2i : bvec2<ll, vec2i> {
    vec2i() : bvec2{0, 0} {}
    vec2i(ll a, ll b) : bvec2{a, b} {}
}
```

```

bool operator==(vec2i r) const {
    return x == r.x && y == r.y;
}

// Sort by angle, length if angles equal
bool operator<(vec2i r) const {
    if (upper() != r.upper()) return upper();
    auto t = cross(r);
    return t > 0 || (!t && len2() < r.len2());
}

bool upper() const {
    return y > 0 || (y == 0 && x >= 0);
}

// Version for double coordinates
// Requires cmp() from template
struct vec2d : bvec2<double, vec2d> {
    vec2d() : bvec2{0, 0} {}
    vec2d(double a, double b) : bvec2{a, b} {}

    vec2d unit() const {return*this/len();}
    vec2d rotate(double a) const { // CCW
        return {x*cos(a) - y*sin(a),
            x*sin(a) + y*cos(a)};
    }

    bool operator==(vec2d r) const {
        return !cmp(x, r.x) && !cmp(y, r.y);
    }

    // Sort by angle, length if angles equal
    bool operator<(vec2d r) const {
        int t = cmp(angle(), r.angle());
        return t < 0 || (!t && len2() < r.len2());
    }
};
```

using vec2 = vec2d;

## graphs/2sat.h

11

```

// 2-SAT solver; time: O(n+m), space: O(n+m)
// Variables are indexed from 1 and
// negative indices represent negations!
// Usage: SAT2 sat(variable_count);
// (add constraints...)
// bool solution_found = sat.solve();
// sat[i] = value of i-th variable, 0 or 1
// (also indexed from 1!)
// (internally: positive = i*2-1, neg. = i*2-2)
struct SAT2 : Vi {
    vector<Vi> G;
    Vi order, flags;

    // Init n variables, you can add more later
    SAT2(int n = 0) : G(n*2) {}

    // Add new var and return its index
    int addVar() {
        G.resize(sz(G)+2); return sz(G)/2;
    }

    // Add (i => j) constraint
    void imply(int i, int j) {
        i = max(i*2-1, -i*2-2);
```

```

    j = max(j*2-1, -j*2-2);
    G[i].pb(j); G[j*1].pb(i^1);
}

// Add (i v j) constraint
void either(int i, int j) { imply(-i, j); }

// Constraint at most one true variable
void atMostOne(Vi& vars) {
    int x = addVar();
    each(i, vars) {
        int y = addVar();
        imply(x, y); imply(i, -x); imply(i, y);
        x = y;
    }
}

// Solve and save assignments in 'values'
bool solve() { // O(n+m), Kosaraju is used
    assign(sz(G)/2+1, -1);
    flags.assign(sz(G), 0);
    rep(i, 0, sz(G)) dfs(i);
    while (!order.empty()) {
        if (!propag(order.back()^1, 1)) return 0;
        order.pop_back();
    }
    return 1;
}

void dfs(int i) {
    if (flags[i]) return;
    flags[i] = 1;
    each(e, G[i]) dfs(e);
    order.pb(i);
}

bool propag(int i, bool first) {
    if (!flags[i]) return 1;
    flags[i] = 0;
    if (at(i/2+1) >= 0) return first;
    at(i/2+1) = i&1;
    each(e, G[i]) if (!propag(e, 0)) return 0;
    return 1;
}
};

```

## graphs/bellman\_inequalities.h 12

```

struct Ineq {
    ll a, b, c; // a - b >= c
};

// Solve system of inequalities of form a-b>=c
// using Bellman-Ford; time: O(n*m)
bool solveIneq(vector<Ineq>& edges,
    vector<ll>& vars) {
    rep(i, 0, sz(vars)) each(e, edges)
        vars[e.b] = min(vars[e.b], vars[e.a]-e.c);
    each(e, edges)
        if (vars[e.a]-e.c < vars[e.b]) return 0;
    return 1;
}

```

## graphs/bridges\_online.h 13

```

// Dynamic 2-edge connectivity queries
// Usage: Bridges bridges(vertex_count);
// - bridges.addEdge(u, v); - add edge (u, v)

```

```

// - bridges.cc[v] = connected component ID
// - bridges.bi(v) = 2-edge connected comp ID
struct Bridges {
    vector<Vi> G; // Spanning forest
    Vi cc, size, par, bp, seen;
    int cnt{0};

    // Initialize structure for n vertices; O(n)
    Bridges(int n = 0) : G(n), cc(n), size(n, 1),
        par(n, -1), bp(n, -1),
        seen(n) {
        iota(all(cc), 0);
    }

    // Add edge (u, v); time: amortized O(lg n)
    void addEdge(int u, int v) {
        if (cc[u] == cc[v]) {
            int r = lca(u, v);
            while ((v = root(v)) != r)
                v = bp[bi(v)] = par[v];
            while ((u = root(u)) != r)
                u = bp[bi(u)] = par[u];
        } else {
            G[u].pb(v); G[v].pb(u);
            if (size[cc[u]] > size[cc[v]]) swap(u, v);
            size[cc[v]] += size[cc[u]];
            dfs(u, v);
        }
    }

    // Get 2-edge connected component ID
    int bi(int v) { // amortized time: < O(lg n)
        return bp[v] == -1 ? v : bp[v] = bi(bp[v]);
    }

    int root(int v) {
        return par[v] == -1 || bi(par[v]) != bi(v)
            ? v : par[v] = root(par[v]);
    }

    void dfs(int v, int p) {
        par[v] = p; cc[v] = cc[p];
        each(e, G[v]) if (e != p) dfs(e, v);
    }

    int lca(int u, int v) { // Don't use this!
        for (cnt++; swap(u, v)) if (u != -1) {
            if (seen[u = root(u)] == cnt) return u;
            seen[u] = cnt; u = par[u];
        }
    }
};

```

## graphs/dense\_dfs.h 14

```

#include "../math/bit_matrix.h"

// DFS over adjacency matrix; time: O(n^2/64)
// G = graph, V = not visited vertices masks
// UNTESTED
struct DenseDFS {
    BitMatrix G, V; // space: O(n^2/64)

    DenseDFS(int n = 0) : G(n, n), V(1, n) {
        reset();
    }
}

```

```

void reset() { each(x, V.M) x = -1; }
void setVisited(int i) { V.set(0, i, 0); }
bool isVisited(int i) { return !V(0, i); }

// DFS step: func is called on each unvisited
// neighbour of i. You need to manually call
// setVisited(child) to mark it visited.
template<class T> // Single step: O(n/64)
void step(int i, T func) {
    ull* E = G.row(i);
    for (int w = 0; w < G.stride(); ) {
        ull x = E[w] & V.row(0)[w];
        if (x) func((w<<6) | __builtin_ctzll(x));
        else w++;
    }
}
};

```

## graphs/edmonds\_karp.h 15

```

constexpr int INF = 1e9+10;

// Edmonds-Karp algorithm for finding
// maximum flow in graph; time: O(V*E^2)
// NOT HEAVILY TESTED
struct MaxFlow {
    using T = int;

    struct Edge {
        int dst, inv;
        T flow, cap;
    };

    vector<vector<Edge>> G;
    vector<T> add;
    Vi prev;

    // Initialize for n vertices
    MaxFlow(int n = 0) : G(n) {}

    // Add new vertex
    int addVert() {
        G.emplace_back(); return sz(G)-1;
    }

    // Add edge between u and v with capacity cap
    // and reverse capacity rcap
    void addEdge(int u, int v, T cap, T rcap=0) {
        G[u].pb({ v, sz(G[v]), 0, cap });
        G[v].pb({ u, sz(G[u])-1, 0, rcap });
    }

    // Compute maximum flow from src to dst.
    // Flow values can be found in edges,
    // vertices with 'add' >= 0 belong to
    // cut component containing 's'.
    T maxFlow(int src, int dst) {
        T f = 0;

        do {
            queue<int> Q;
            Q.push(src);
            prev.assign(sz(G), -1);
            add.assign(sz(G), -1);
            add[src] = INF;

            while (!Q.empty()) {

```

```

                int i = Q.front();
                T m = add[i];
                Q.pop();

                if (i == dst) {
                    while (i != src) {
                        auto& e = G[i][prev[i]];
                        e.flow -= m;
                        G[e.dst][e.inv].flow += m;
                        i = e.dst;
                    }
                    f += m;
                    break;
                }

                each(e, G[i]) if (add[e.dst] < 0) {
                    if (e.flow < e.cap) {
                        Q.push(e.dst);
                        prev[e.dst] = e.inv;
                        add[e.dst] = min(m, e.cap-e.flow);
                    }
                }
                while (prev[dst] != -1);

            return f;
        }
    };
}

```

## graphs/push\_relabel.h 16

```

constexpr int64_t INF = 1e18;

// Push-relabel algorithm with global relabel
// heuristic for finding maximum flow; O(V^3),
// but very fast in practice.
// Preflow is not converted to flow!
struct MaxFlow {
    using T = int64_t;

    struct Vert {
        int head{0}, cur{0}, label;
        T excess;
    };

    struct Edge {
        int dst, nxt;
        T avail, cap;
    };

    vector<Vert> V;
    vector<Edge> E;
    queue<int> que, bfs;

    // Initialize for n vertices
    MaxFlow(int n = 0) {
        V.assign(n, {});
        E.resize(2);
    }

    // Add new vertex
    int addVert() {
        V.emplace_back();
        return sz(V)-1;
    }

    // Add edge between u and v with capacity cap

```

```
// and reverse capacity rcap
void addEdge(int u, int v, T cap, T rcap=0) {
    E.pb({ v, V[u].head, 0, cap });
    E.pb({ u, V[v].head, 0, rcap });
    V[u].head = sz(E)-2;
    V[v].head = sz(E)-1;
}

void push(int v, int e) {
    T f = min(V[v].excess, E[e].avail);
    E[e].avail -= f;
    E[e^1].avail += f;
    V[v].excess -= f;
    if ((V[E[e].dst].excess += f) == f)
        que.push(E[e].dst);
}

// Compute maximum flow from src to dst
T maxFlow(int src, int dst) {
    each(v, V) v.excess = v.label = v.cur = 0;
    each(e, E) e.avail = max(e.cap, T(0));

    int cnt, n = cnt = V[src].label = sz(V);
    V[src].excess = INF;
    for (int e = V[src].head; e; e = E[e].nxt)
        push(src, e);

    for (; !que.empty(); que.pop()) {
        if (cnt >= n/2) {
            each(v, V) v.label = n;
            V[dst].label = 0;
            bfs.push(dst);
            cnt = 0;

            for (; !bfs.empty(); bfs.pop()) {
                auto& v = V[bfs.front()];
                for (int e=v.head; e; e = E[e].nxt) {
                    int x = E[e].dst;
                    if (E[e^1].avail &&
                        V[x].label > v.label+1) {
                        V[x].label = v.label+1;
                        bfs.push(x);
                    }
                }
            }

            int v = que.front(), &l = V[v].label;
            if (v == dst) continue;

            while (V[v].excess && l < n) {
                if (!V[v].cur) {
                    l = n;
                    for (int e=V[v].head; e; e=E[e].nxt){
                        if (E[e].avail)
                            l = min(l, V[E[e].dst].label+1);
                    }
                    V[v].cur = V[v].head;
                    cnt++;
                }

                int e = V[v].cur;
                V[v].cur = E[e].nxt;
                if (E[e].avail &&
                    l == V[E[e].dst].label+1) push(v, e);
            }
        }
    }
}
```

```
}
    return V[dst].excess;
}

// Get if v belongs to cut component with src
bool cutSide(int v) {
    return V[v].label >= sz(V);
}
};
```

## graphs/scc.h 17

```
// Tarjan's SCC algorithm; time: O(n+m)
// Usage: SCC scc(graph);
// scc[v] = index of SCC for vertex v
// scc.comps[i] = vertices of i-th SCC
struct SCC : Vi {
    vector<Vi> comps;
    Vi S;
    int cnt{0};

    SCC() {}

    SCC(vector<Vi>& G) : Vi(sz(G),-1), S(sz(G)) {
        rep(i, 0, sz(G)) if (!S[i]) dfs(G, i);
    }

    int dfs(vector<Vi>& G, int v) {
        int low = S[v] = ++cnt, t = -1;
        S.pb(v);

        each(e, G[v]) if (at(e) < 0)
            low = min(low, S[e] ? dfs(G, e));

        if (low == S[v]) {
            comps.emplace_back();
            for (; t != v; S.pop_back()) {
                at(t = S.back()) = sz(comps) - 1;
                comps.back().pb(t);
            }
        }

        return low;
    }
};
```

## graphs/turbo\_matching.h 18

```
// Find maximum bipartite matching; time: ?
// G must be bipartite graph!
// Returns matching size (edge count).
// match[v] = vert matched to v or -1
int matching(vector<Vi>& G, Vi& match) {
    vector<bool> seen;
    int n = 0, k = 1;
    match.assign(sz(G), -1);

    function<int(int)> dfs = [&](int i) {
        if (seen[i]) return 0;
        seen[i] = 1;
        each(e, G[i]) {
            if (match[e] < 0 || dfs(match[e])) {
                match[i] = e; match[e] = i;
                return 1;
            }
        }
        return 0;
    };
}
```

```
};

while (k) {
    seen.assign(sz(G), 0);
    k = 0;
    rep(i, 0, sz(G)) if (match[i] < 0)
        k += dfs(i);
    n += k;
}
return n;
}

// Convert maximum matching to vertex cover
// time: O(n+m)
Vi vertexCover(vector<Vi>& G, Vi& match) {
    Vi ret, col(sz(G)), seen(sz(G));

    function<void(int, int)> dfs =
        [&](int i, int c) {
            if (col[i]) return;
            col[i] = c+1;
            each(e, G[i]) dfs(e, !c);
        };

    function<void(int)> aug = [&](int i) {
        if (seen[i] || col[i] != 1) return;
        seen[i] = 1;
        each(e, G[i]) seen[e] = 1, aug(match[e]);
    };

    rep(i, 0, sz(G)) dfs(i, 0);
    rep(i, 0, sz(G)) if (match[i] < 0) aug(i);
    rep(i, 0, sz(G))
        if (seen[i] == col[i]-1) ret.pb(i);
    return ret;
}
```

## math/bit\_gauss.h 19

```
constexpr int MAX_COLS = 2048;

// Solve system of linear equations over Z_2
// time: O(n^2*m/W), where W is word size
// - A - extended matrix, rows are equations,
//       columns are variables,
//       m-th column is equation result
//       (A[i][j] - i-th row and j-th column)
// - ans - output for variables values
// - m - variable count
// Returns 0 if no solutions found, 1 if one,
// 2 if more than 1 solution exist.
int bitGauss(vector<bitset<MAX_COLS>&& A,
              vector<bool>& ans, int m) {
    Vi col;
    ans.assign(m, 0);

    rep(i, 0, sz(A)) {
        int c = int(A[i]._Find_first());
        if (c >= m) {
            if (c == m) return 0;
            continue;
        }

        rep(k, i+1, sz(A)) if (A[k][c]) A[k]^=A[i];
        swap(A[i], A[sz(col)]);
        col.pb(c);
    }
}
```

```
for (int i = sz(col); i--;) if (A[i][m]) {
    ans[col[i]] = 1;
    rep(k,0,i) if(A[k][col[i]]) A[k][m].flip();
}

return sz(col) < m ? 2 : 1;
}
```

## math/bit\_matrix.h 20

```
using ull = uint64_t;

// Matrix over Z_2 (bits and xor)
// UNTESTED and UNFINISHED
struct BitMatrix {
    vector<ull> M;
    int rows, cols, stride;

    BitMatrix(int n = 0, int m = 0) {
        rows = n; cols = m;
        stride = (m+63)/64;
        M.resize(n*stride);
    }

    ull* row(int i) { return &M[i*stride]; }

    bool operator()(int i, int j) {
        return (row(i)[j/64] >> (j%64)) & 1;
    }

    void set(int i, int j, bool val) {
        ull &w = row(i)[j/64], m = 1 << (j%64);
        if (val) w |= m;
        else w &= ~m;
    }
};
```

## math/crt.h 21

```
using Pll = pair<ll, ll>;

ll egcd(ll a, ll b, ll& x, ll& y) {
    if (!a) return x=0, y=1, b;
    ll d = egcd(b%a, a, y, x);
    x -= b/a*y;
    return d;
}

// Chinese Remainder Theoerem; time: O(lg lcm)
// Solves x = a.x (mod a.y), x = b.x (mod b.y)
// Returns pair (x mod lcm, lcm(a.y, b.y))
// or (-1, -1) if there's no solution.
// WARNING: a.x and b.x are assumed to be
// in [0;a.y) and [0;b.y) respectively.
// Works properly if lcm(a.y, b.y) < 2^63.
Pll crt(Pll a, Pll b) {
    if (a.y < b.y) swap(a, b);
    ll x, y, g = egcd(a.y, b.y, x, y);
    ll c = b.x-a.x, d = b.y/g, p = a.y*d;
    if (c % g) return {-1, -1};
    ll s = (a.x + c/g*x % d * a.y) % p;
    return {s < 0 ? s+p : s, p};
}
```

## math/discrete\_logarithm.h 22

```
#include "../modular.h"
```

```
// Baby-step giant-step algorithm; O(sqrt(p))
// Finds smallest x such that a^x = b (mod p)
// or returns -1 if there's no solution.
ll dlog(ll a, ll b, ll p) {
    int m = int(min(llround(sqrt(p))+1, p-1));
    unordered_map<ll, int> small;
    ll t = 1;
```

```
    rep(i, 0, m) {
        int& k = small[t];
        if (!k) k = i+1;
        t = t*a % p;
    }

    t = modInv(t, p);

    rep(i, 0, m) {
        int j = small[b];
        if (j) return i*m + j - 1;
        b = b*t % p;
    }

    return -1;
}
```

## math/fft\_complex.h

23

```
using cmpl = complex<double>;
using Vfft = vector<cmpl>;

// Compute DFT over complex numbers; O(n lg n)
// DFT is in bit-reversed order!
// Input size must be power of 2!
template<int dir> // 1 for DFT, -1 for inverse
void fft(Vfft& buf) {
    assert(__builtin_popcount(sz(buf)) == 1);
    int n = sz(buf), bits = 31-__builtin_clz(n);
    int i = (dir > 0 ? 0 : bits-1);

    Vfft bases = {1};
    auto c = exp(cmpl(0, 2*M_PI/n));
    rep(k, 0, n) bases.pb(bases.back()*c);

    for (; i >= 0 && i < bits; i += dir) {
        int shift = 1 << (bits-i-1);
        rep(j, 0, 1 << i) rep(k, 0, shift) {
            int a = (j << (bits-i)) | k, b = a|shift;
            auto v1 = buf[a], v2 = buf[b];
            auto base = bases[(dir*(k<i)) & (n-1)];
            if (dir > 0) {
                buf[b] = (v1-v2) * base;
            } else {
                v2 *= base;
                buf[b] = v1 - v2;
            }
            buf[a] = v1 + v2;
        }
    }

    if (dir < 0) each(x, buf) x /= n;
}
```

// Convolve a and b, store result in a;  
// time: O(n lg n)

```
void convolve(Vfft& a, Vfft b) {
    int len = sz(a) + sz(b) - 1;
```

```
    int n = 1 << (32 - __builtin_clz(len));
    a.resize(n); b.resize(n);
    fft<1>(a); fft<1>(b);
    rep(i, 0, n) a[i] *= b[i];
    fft<-1>(a);
    a.resize(len);
}
```

## math/fft\_mod.h

24

```
constexpr ll MOD = 15*(1<<27)+1; // ~2e9
constexpr ll ROOT = 440564289; // order = 1<<27

ll modInv(ll a, ll m) {
    if (a == 1) return 1;
    return ((a - modInv(m%a, a))*m + 1) / a;
}

// Compute DFT over integers mod MOD; O(n lg n)
// DFT is in bit-reversed order!
// Input size must be power of 2!
template<int dir> // 1 for DFT, -1 for inverse
void fft(vector<ll>& buf) {
    assert(__builtin_popcount(sz(buf)) == 1);
    int n = sz(buf), bits = 31-__builtin_clz(n);
    int i = (dir > 0 ? 0 : bits-1);

    vector<ll> bases = {1};
    ll c = ROOT;
    rep(k, bits, 27) c = c*c % MOD;
    rep(k, 0, n) bases.pb(bases.back()*c % MOD);

    for (; i >= 0 && i < bits; i += dir) {
        int shift = 1 << (bits-i-1);
        rep(j, 0, 1 << i) rep(k, 0, shift) {
            int a = (j << (bits-i)) | k, b = a|shift;
            auto v1 = buf[a], v2 = buf[b];
            auto base = bases[(dir*(k<i)) & (n-1)];
            if (dir > 0) {
                buf[b] = (v1-v2)*base % MOD;
            } else {
                v2 *= base;
                buf[b] = (v1-v2) % MOD;
            }
            buf[a] = (v1+v2) % MOD;
        }
    }

    if (dir < 0) {
        ll y = modInv(n, MOD);
        each(x, buf) x = x*y % MOD;
    }

    // Convolve a and b, store result in a;
    // time: O(n lg n)
    void convolve(vector<ll>& a, vector<ll> b) {
        int len = sz(a) + sz(b) - 1;
        int n = 1 << (32 - __builtin_clz(len));
        a.resize(n); b.resize(n);
        fft<1>(a); fft<1>(b);
        rep(i, 0, n) a[i] = a[i]*b[i] % MOD;
        fft<-1>(a);
        a.resize(len);
    }
```

## math/gauss.h

25

```
// Solve system of linear equations; O(n^2*m)
// - A - extended matrix, rows are equations,
//     columns are variables,
//     m-th column is equation result
//     (A[i][j] - i-th row and j-th column)
// - ans - output for variables values
// - m - variable count
// Returns 0 if no solutions found, 1 if one,
// 2 if more than 1 solution exist.
int gauss(vector<vector<double>>& A,
          vector<double>& ans, int m) {
    Vi col;
    ans.assign(m, 0);

    rep(i, 0, sz(A)) {
        int c = 0;
        while (c <= m && !cmp(A[i][c], 0)) c++;
        // For Zp:
        //while (c <= m && !A[i][c].x) c++;

        if (c >= m) {
            if (c == m) return 0;
            continue;
        }

        rep(k, i+1, sz(A)) {
            auto mult = A[k][c] / A[i][c];
            rep(j, 0, m+1) A[k][j] -= A[i][j]*mult;
        }

        swap(A[i], A[sz(col)]);
        col.pb(c);
    }

    for (int i = sz(col); i--;) {
        ans[col[i]] = A[i][m] / A[i][col[i]];
        rep(k, 0, i)
            A[k][m] -= ans[col[i]] * A[k][col[i]];
    }

    return sz(col) < m ? 2 : 1;
}
```

## math/miller\_rabin.h

26

```
#include "modular64.h"

// Miller-Rabin primality test
// time O(k*lg^2 n), where k = number of bases

// Deterministic for p <= 10^9
// constexpr ll BASES[] = {
//     336781006125, 9639812373923155
// };

// Deterministic for p <= 2^64
constexpr ll BASES[] = {
    2,325,9375,28178,450775,9780504,1795265022
};

bool isPrime(ll p) {
    if (p == 2) return true;
    if (p <= 1 || p%2 == 0) return false;

    ll d = p-1, times = 0;
    while (d%2 == 0) d /= 2, times++;
```

```
    each(a, BASES) if (a%p) {
        // ll a = rand() % (p-1) + 1;
        ll b = modPow(a%p, d, p);
        if (b == 1 || b == p-1) continue;

        rep(i, 1, times) {
            b = modMul(b, b, p);
            if (b == p-1) break;
        }

        if (b != p-1) return false;
    }

    return true;
}
```

## math/modinv\_precompute.h

27

```
constexpr ll MOD = 234567899;
vector<ll> modInv(MOD); // You can lower size

// Precompute modular inverses; time: O(MOD)
void initModInv() {
    modInv[1] = 1;
    rep(i, 2, sz(modInv)) modInv[i] =
        (MOD - (MOD/i) * modInv[MOD%i]) % MOD;
}
```

## math/modular.h

28

```
// Big prime number, about 2*10^9
constexpr int MOD = 15*(1<<27)+1;

ll modInv(ll a, ll m) { // a^(-1) mod m
    if (a == 1) return 1;
    return ((a - modInv(m%a, a))*m + 1) / a;
}

ll modPow(ll a, ll e, ll m) { // a^e mod m
    ll t = 1 % m;
    while (e) {
        if (e % 2) t = t*a % m;
        e /= 2; a = a*a % m;
    }
    return t;
}

// Wrapper for modular arithmetic
struct Zp {
    ll x; // Contained value, in range [0;MOD-1]
    Zp() {}
    Zp(ll a) : x(a%MOD) { if (x < 0) x += MOD; }

    #define OP(c,d) Zp& operator c##=(Zp r) { \
        x = x d; return *this; } \
        Zp operator c(Zp r) const { \
            Zp t = *this; return t c##= r; }

    OP(+, +.x - MOD*(x+r.x >= MOD));
    OP(-, -.x + MOD*(x-r.x < 0));
    OP(*, *.x % MOD);
    OP(/, *.inv().x % MOD);

    // For composite modulus use modInv, not pow
    Zp inv() const { return pow(MOD-2); }
    Zp pow(ll e) const { return modPow(x,e,MOD); }
    void print() { cerr << x; } // For deb()
```

```
};

// Extended Euclidean Algorithm
11 egcd(11 a, 11 b, 11& x, 11& y) {
    if (!a) return x=0, y=1, b;
    11 d = egcd(b%a, a, y, x);
    x -= b/a*y;
    return d;
}
```

math/modular64.h29

```
// Modular arithmetic for modulus < 2^62

11 modAdd(11 x, 11 y, 11 m) {
    x += y;
    return x < m ? x : x-m;
}

11 modSub(11 x, 11 y, 11 m) {
    x -= y;
    return x >= 0 ? x : x+m;
}

11 modMul(11 a, 11 b, 11 m) {
    11 c = 11((long double) a * b / m);
    11 r = (a*b - c*m) % m;
    return r < 0 ? r+m : r;
}

11 modPow(11 x, 11 e, 11 m) {
    11 t = 1;
    while (e) {
        if (e & 1) t = modMul(t, x, m);
        e >>= 1;
        x = modMul(x, x, m);
    }
    return t;
}
```

math/montgomery.h30

```
#include "modular.h"

// Montgomery modular multiplication
// MOD < MG_MULT, gcd(MG_MULT, MOD) must be 1
// Don't use if modulo is constexpr; UNTESTED

constexpr 11 MG_SHIFT = 32;
constexpr 11 MG_MULT = 1LL << MG_SHIFT;
constexpr 11 MG_MASK = MG_MULT - 1;
const 11 MG_INV = MG_MULT-modInv(MOD, MG_MULT);

// Convert to Montgomery form
11 MG(11 x) { return (x*MG_MULT) % MOD; }

// Montgomery reduction
// redc(mg * mg) = Montgomery-form product
11 redc(11 x) {
    11 q = (x * MG_INV) & MG_MASK;
    x = (x + q*MOD) >> MG_SHIFT;
    return (x >= MOD ? x-MOD : x);
}
```

math/nimber.h31

```
// Nimbers are defined as sizes of Nim heaps.
// Operations on nimbers are defined as:
```

```
// a+b = mex({a'+b : a' < a} u {a+b' : b' < b})
// ab = mex({a'b+ab'+a'b' : a' < a, b' < b})
// Nimbers smaller than M = 2^2^k form a field.
// Addition is equivalent to xor, meanwhile
// multiplication can be evaluated
// in O(lg^2 M) after precomputing.

using ull = uint64_t;
ull nbuf[64][64]; // Nim-products for 2^i * 2^j

// Multiply nimbers; time: O(lg^2 M)
// WARNING: Call initNimMul() before using.
ull nimMul(ull a, ull b) {
    ull ret = 0;
    for (ull s = a; s; s &= (s-1))
        for (ull t = b; t; t &= (t-1))
            ret ^= nbuf[__builtin_ctzll(s)]
                    [__builtin_ctzll(t)];
    return ret;
}

// Initialize nim-products lookup table
void initNimMul() {
    rep(i, 0, 64)
        nbuf[i][0] = nbuf[0][i] = 1ull << i;
    rep(b, 1, 64) rep(a, 1, b+1) {
        int i = 1 << (63 - __builtin_clzll(a));
        int j = 1 << (63 - __builtin_clzll(b));
        ull t = nbuf[a-i][b-j];
        if (i < j)
            t = nimMul(t, 1ull << i) << j;
        else
            t = nimMul(t, 1ull << (i-1)) ^ (t << i);
        nbuf[a][b] = nbuf[b][a] = t;
    }

// Compute a^e under nim arithmetic; O(lg^3 M)
// WARNING: Call initNimMul() before using.
ull nimPow(ull a, ull e) {
    ull t = 1;
    while (e) {
        if (e % 2) t = nimMul(t, a);
        e /= 2; a = nimMul(a, a);
    }
    return t;
}

// Compute inverse of a in 2^64 nim-field;
// time: O(lg^3 M)
// WARNING: Call initNimMul() before using.
ull nimInv(ull a) {
    return nimPow(a, ull(-2));
}

// If you need to multiply many nimbers by
// the same value you can use this to speedup.
// WARNING: Call initNimMul() before using.
struct NimMult {
    ull M[64] = {0};

// Initialize lookup; time: O(lg^2 M)
NimMult(ull a) {
    for (ull t=a; t; t &= (t-1)) rep(i, 0, 64)
        M[i] ^= nbuf[__builtin_ctzll(t)][i];
}
```

```
// Multiply by b; time: O(lg M)
ull operator()(ull b) {
    ull ret = 0;
    for (ull t = b; t; t &= (t-1))
        ret ^= M[__builtin_ctzll(t)];
    return ret;
}
};
```

math/phi\_large.h32

```
#include "pollard_rho.h"

// Compute Euler's totient of large numbers
// time: O(n^(1/4)) <- factorization
11 phi(11 n) {
    each(p, factorize(n)) n = n / p.x * (p.x-1);
    return n;
}
```

math/phi\_precompute.h33

```
Vi phi(10e6+1);

// Precompute Euler's totients; time O(n lg n)
void calcPhi() {
    iota(all(phi), 0);
    rep(i, 2, sz(phi)) if (phi[i] == i)
        for (int j = i; j < sz(phi); j += i)
            phi[j] = phi[j] / i * (i-1);
}
```

math/pi\_large\_precomp.h34

```
#include "sieve.h"

// Count primes in given interval
// using precomputed table.
// Set MAX_P to sqrt(MAX_N) and run sieve()!
// Precomputed table will contain N_BUCKETS
// elements - check source size limit.

constexpr 11 MAX_N = 1e11+1;
constexpr 11 N_BUCKETS = 10000;
constexpr 11 BUCKET_SIZE = (MAX_N/N_BUCKETS)+1;
constexpr 11 precomputed[] = { /* ... */ };

11 sieveRange(11 from, 11 to) {
    bitset<BUCKET_SIZE> elems;
    from = max(from, 2LL);
    to = max(from, to);
    each(p, primesList) {
        11 c = max((from+p-1) / p, 2LL);
        for (11 i = c*p; i < to; i += p)
            elems.set(i-from);
    }
    return to-from-elems.count();
}

// Run once on local computer to precompute
// table. Takes about 10 minutes for n = 1e11.
// Sanity check (for default params):
// 664579, 606028, 587253, 575795, ...
void localPrecompute() {
    for (11 i = 0; i < MAX_N; i += BUCKET_SIZE) {
        11 to = min(i+BUCKET_SIZE, MAX_N);
        cout << sieveRange(i, to) << ', ' << flush;
```

```
}
    cout << endl;
}

// Count primes in [from;to) using table.
// O(N_BUCKETS + BUCKET_SIZE*lg lg n + sqrt(n))
11 countPrimes(11 from, 11 to) {
    11 bFrom = from/BUCKET_SIZE+1,
        bTo = to/BUCKET_SIZE;
    if (bFrom > bTo) return sieveRange(from, to);
    11 ret = accumulate(precomputed+bFrom,
                        precomputed+bTo, 0);
    ret += sieveRange(from, bFrom*BUCKET_SIZE);
    ret += sieveRange(bTo*BUCKET_SIZE, to);
    return ret;
}
```

math/pollard\_rho.h35

```
#include "modular64.h"
#include "miller_rabin.h"

using Factor = pair<11, int>;

void rho(vector<11>& out, 11 n) {
    if (n <= 1) return;
    if (isPrime(n)) out.pb(n);
    else if (n%2 == 0) rho(out,2), rho(out,n/2);
    else for (11 a = 2; a++) {
        11 x = 2, y = 2, d = 1;
        while (d == 1) {
            x = modAdd(modMul(x, x, n), a, n);
            y = modAdd(modMul(y, y, n), a, n);
            y = modAdd(modMul(y, y, n), a, n);
            d = __gcd(abs(x-y), n);
        }
        if (d != n) {
            rho(out, d);
            rho(out, n/d);
            return;
        }
    }
}
```

```
// Pollard's rho factorization algorithm
// Las Vegas version; time: n^(1/4)
// Returns pairs (prime, power), sorted
vector<Factor> factorize(11 n) {
    vector<Factor> ret;
    vector<11> raw;
    rho(raw, n);
    sort(all(raw));
    each(f, raw) {
        if (ret.empty() || ret.back().x != f)
            ret.pb({ f, 1 });
        else
            ret.back().y++;
    }
    return ret;
}
```

math/polynomial\_interp.h36

```
// Interpolates set of points (i, vec[i])
// and returns it evaluated at x; time: O(n^2)
// TODO: Improve to linear time
template<typename T>
T polyExtend(vector<T>& vec, T x) {
```

```

T ret = 0;
rep(i, 0, sz(vec)) {
    T a = vec[i], b = 1;
    rep(j, 0, sz(vec)) if (i != j) {
        a *= x-j; b *= i-j;
    }
    ret += a/b;
}
return ret;
}

```

**math/sieve.h**

37

```

constexpr int MAX_P = 1e6;
bitset<MAX_P+1> primes;
Vi primesList;

```

```

// Erathostenes sieve; time: O(n lg lg n)
void sieve() {
    primes.set();
    primes.reset(0);
    primes.reset(1);

```

```

    for (int i = 2; i*i <= MAX_P; i++)
        if (primes[i])
            for (int j = i*i; j <= MAX_P; j += i)
                primes.reset(j);

```

```

    rep(i, 0, MAX_P+1) if (primes[i])
        primesList.pb(i);
}

```

**math/sieve\_factors.h**

38

```

constexpr int MAX_P = 1e6;
Vi factor(MAX_P+1);

```

```

// Erathostenes sieve with saving smallest
// factor for each number; time: O(n lg lg n)
void sieve() {
    for (int i = 2; i*i <= MAX_P; i++)
        if (!factor[i])
            for (int j = i*i; j <= MAX_P; j += i)
                if (!factor[j])
                    factor[j] = i;

    rep(i, 0, MAX_P+1) if (!factor[i]) factor[i]=i;
}

```

```

// Factorize n <= MAX_P; time: O(lg n)
// Returns pairs (prime, power), sorted
vector<Pii> factorize(ll n) {
    vector<Pii> ret;
    while (n > 1) {
        int f = factor[n];
        if (ret.empty() || ret.back().x != f)
            ret.pb({ f, 1 });
        else
            ret.back().y++;
        n /= f;
    }
    return ret;
}

```

**math/sieve\_segmented.h**

39

```

constexpr int MAX_P = 1e9;
bitset<MAX_P/2+1> primes; // Only odd numbers

```

```

// Cache-friendly Erathostenes sieve
// ~1.5s on Intel Core i5 for MAX_P = 10^9
// Memory usage: MAX_P/16 bytes
void sieve() {
    constexpr int SEG_SIZE = 1<<18;
    int pSqrt = int(sqrt(MAX_P)+0.5);
    vector<Pii> dels;
    primes.set();
    primes.reset(0);

```

```

    for (int i = 3; i <= pSqrt; i += 2) {
        if (primes[i/2]) {
            int j;
            for (j = i*i; j <= pSqrt; j += i*2)
                primes.reset(j/2);
            dels.pb({ i, j/2 });
        }
    }

    for (int seg = pSqrt/2;
         seg <= sz(primes); seg += SEG_SIZE) {
        int lim = min(seg+SEG_SIZE, sz(primes));
        each(d, dels) for (;d.y < lim; d.y += d.x)
            primes.reset(d.y);
    }
}

```

```

bool isPrime(int x) {
    return x == 2 || (x%2 && primes[x/2]);
}

```

**structures/bitset\_plus.h**

40

```

// Undocumented std::bitset features:
// - _Find_first() - returns first bit = 1 or N
// - _Find_next(i) - returns first bit = 1
//               after i-th bit
//               or N if not found

```

```

// Bitwise operations for vector<bool>
// UNTESTED

```

```

#define OP(x) vector<bool>& operator x##=( \
    vector<bool>& l, const vector<bool>& r) { \
    assert(sz(l) == sz(r)); \
    auto a = l.begin(); auto b = r.begin(); \
    while (a<l.end()) *a._M_p++ x##= *b._M_p++; \
    return l; }

```

```
OP(&)OP(!)OP(^)
```

**structures/fenwick\_tree.h**

41

```

// Fenwick tree (BIT tree); space: O(n)
// Default version: prefix sums
struct Fenwick {
    using T = int;
    static const T ID = 0;
    T f(T a, T b) { return a+b; }

```

```

    vector<T> s;
    Fenwick(int n = 0) : s(n, ID) {}

```

```

    // A[i] = f(A[i], v); time: O(lg n)
    void modify(int i, T v) {
        for (; i < sz(s); i |= i+1) s[i]=f(s[i],v);
    }

```

```

// Get f(A[0], ..., A[i-1]); time: O(lg n)
T query(int i) {
    T v = ID;
    for (; i > 0; i &= i-1) v = f(v, s[i-1]);
    return v;
}

```

```

// Find smallest i such that
// f(A[0],...,A[i-1]) >= val; time: O(lg n)
// Prefixes must have non-decreasing values.
int lowerBound(T val) {
    if (val <= ID) return 0;
    int i = -1, mask = 1;
    while (mask <= sz(s)) mask *= 2;
    T off = ID;

```

```

    while (mask /= 2) {
        int k = mask+i;
        if (k < sz(s)) {
            T x = f(off, s[k]);
            if (val > x) i=k, off=x;
        }
    }
    return i+2;
}

```

};

**structures/fenwick\_tree\_2d.h**

42

```

// Fenwick tree 2D (BIT tree 2D); space: O(n*m)
// Default version: prefix sums 2D
// Change s to hashmap for O(q lg^2 n) memory
struct Fenwick2D {
    using T = int;
    static constexpr T ID = 0;
    T f(T a, T b) { return a+b; }

```

```

    vector<T> s;
    int w, h;

```

```

    Fenwick2D(int n = 0, int m = 0)
        : s(n*m, ID), w(n), h(m) {}

```

```

    // A[i,j] = f(A[i,j], v); time: O(lg^2 n)
    void modify(int i, int j, T v) {
        for (; i < w; i |= i+1)
            for (int k = j; k < h; k |= k+1)
                s[i*h+k] = f(s[i*h+k], v);
    }

```

```

    // Query prefix; time: O(lg^2 n)
    T query(int i, int j) {
        T v = ID;
        for (; i>0; i&=i-1)
            for (int k = j; k > 0; k &= k-1)
                v = f(v, s[i*h+k-h-1]);
        return v;
    }
}

```

**structures/find\_union.h**

43

```

// Disjoint set data structure; space: O(n)
// Operations work in amortized O(alfa(n))
struct FAU {
    Vi G;
    FAU(int n = 0) : G(n, -1) {}

```

```

// Get size of set containing i
int size(int i) { return -G[find(i)]; }

// Find representative of set containing i
int find(int i) {
    return G[i] < 0 ? i : G[i] = find(G[i]);
}

```

```

// Union sets containing i and j
bool join(int i, int j) {
    i = find(i); j = find(j);
    if (i == j) return 0;
    if (G[i] > G[j]) swap(i, j);
    G[i] += G[j]; G[j] = i;
    return 1;
}
};

```

**structures/hull\_offline.h**

44

```

constexpr ll INF = 2e18;
// constexpr double INF = 1e30;
// constexpr double EPS = 1e-9;

```

```

// MAX of linear functions; space: O(n)
// Use if you add lines in increasing 'a' order
// Default uncommented version is for int64
struct Hull {
    using T = ll; // Or change to double

```

```

    struct Line {
        T a, b, end;
        T intersect(const Line& r) const {
            // Version for double:
            //if (r.a-a < EPS) return b>r.b?INF:-INF;
            //return (b-r.b) / (r.a-a);
            if (a==r.a) return b > r.b ? INF : -INF;
            ll u = b-r.b, d = r.a-a;
            return u/d + ((u^d) >= 0 || !(u&d));
        }
    };

```

```

    vector<Line> S;
    Hull() { S.pb({ 0, -INF, INF }); }

```

```

    // Insert f(x) = ax+b; time: amortized O(1)
    void push(T a, T b) {
        Line l{a, b, INF};
        while (true) {
            T e = S.back().end=S.back().intersect(l);
            if (sz(S) < 2 || S[sz(S)-2].end < e)
                break;
            S.pop_back();
        }
        S.pb(l);
    }

```

```

    // Query max(f(x) for each f); time: O(lg n)
    T query(T x) {
        auto t = *upper_bound(all(S), x,
            [](int l, const Line& r) {
                return l < r.end;
            });
        return t.a*x + t.b;
    }
};

```



**structures/hull\_online.h**

45

```
constexpr ll INF = 2e18;

// MAX of linear functions online; space: O(n)
struct Hull {
    static bool modeQ; // Toggles operator< mode

    struct Line {
        mutable ll a, b, end;

        ll intersect(const Line& r) const {
            if (a==r.a) return b > r.b ? INF : -INF;
            ll u = b-r.b, d = r.a-a;
            return u/d + ((u^d) >= 0 || !(u^d));
        }

        bool operator<(const Line& r) const {
            return modeQ ? end < r.end : a < r.a;
        }
    };

    multiset<Line> S;
    Hull() { S.insert({ 0, -INF, INF }); }

    // Updates segment end
    bool update(multiset<Line>::iterator it) {
        auto cur = it++; cur->end = INF;
        if (it == S.end()) return false;
        cur->end = cur->intersect(*it);
        return cur->end >= it->end;
    }

    // Insert f(x) = ax+b; time: O(lg n)
    void insert(ll a, ll b) {
        auto it = S.insert({ a, b, INF });
        while (update(it)) it = --S.erase(++it);
        rep(i, 0, 2)
            while (it != S.begin() && update(--it))
                update(it = --S.erase(++it));
    }

    // Query max(f(x) for each f): time: O(lg n)
    ll query(ll x) {
        modeQ = 1;
        auto l = *S.upper_bound({ 0, 0, x });
        modeQ = 0;
        return l.a*x + l.b;
    }
};
```

bool Hull::modeQ = false;

**structures/max\_queue.h**

46

```
// Queue with max query on contained elements
struct MaxQueue {
    using T = int;
    deque<T> Q, M;

    // Add v to the back; time: amortized O(1)
    void push(T v) {
        while (!M.empty() && M.back() < v)
            M.pop_back();
        M.pb(v); Q.pb(v);
    }

    // Pop from the front; time: O(1)
```

```
void pop() {
    if (M.front() == Q.front()) M.pop_front();
    Q.pop_front();
}

// Get max element value; time: O(1)
T max() const { return M.front(); }
};
```

**structures/pairing\_heap.h**

47

```
// Pairing heap implementation; space O(n)
// Elements are stored in vector for faster
// allocation. It's MINIMUM queue.
// Allows to merge heaps in O(1)
template<class T, class Cmp = less<T>>
struct PHeap {
    struct Node {
        T val;
        int child{-1}, next{-1}, prev{-1};

        Node(T x = T()) : val(x) {}
    };

    using Vnode = vector<Node>;
    Vnode& M;
    int root{-1};

    int unlink(int& i) {
        if (i >= 0) M[i].prev = -1;
        int x = i; i = -1;
        return x;
    }

    void link(int host, int& i, int val) {
        if (i >= 0) M[i].prev = -1;
        i = val;
        if (i >= 0) M[i].prev = host;
    }

    int merge(int l, int r) {
        if (l < 0) return r;
        if (r < 0) return l;
        if (Cmp()(M[l].val, M[r].val)) swap(l, r);

        link(l, M[l].next, unlink(M[r].child));
        link(r, M[r].child, l);
        return r;
    }

    int mergePairs(int v) {
        if (v < 0 || M[v].next < 0) return v;
        int v2 = unlink(M[v].next);
        int v3 = unlink(M[v2].next);
        return merge(merge(v, v2), mergePairs(v3));
    }

    // ---

    // Initialize heap with given node storage
    // Just declare 1 Vnode and pass it to heaps
    PHeap(Vnode& mem) : M(mem) {}

    // Add given key to heap, returns index; O(1)
    int push(const T& x) {
        int index = sz(M);
        M.emplace_back(x);
```

```
        root = merge(root, index);
        return index;
    }

    // Change key of i to smaller value; O(1)
    void decrease(int i, T val) {
        assert(!Cmp()(M[i].val, val));
        M[i].val = val;

        int prev = M[i].prev;
        if (prev < 0) return;

        auto& p = M[prev];
        link(prev, (p.child == i ? p.child
            : p.next), unlink(M[i].next));

        root = merge(root, i);
    }

    bool empty() { return root < 0; }
    const T& top() { return M[root].val; }

    // Merge with other heap. Must use same vec.
    void merge(PHeap& r) { // time: O(1)
        assert(&M == &r.M);
        root = merge(root, r.root); r.root = -1;
    }

    // Remove min element; time: O(lg n)
    void pop() {
        root = mergePairs(unlink(M[root].child));
    }
};
```

**structures/rmq.h**

48

```
// Range Minimum Query; space: O(n lg n)
struct RMQ {
    using T = int;
    static constexpr T ID = INT_MAX;
    T f(T a, T b) { return min(a, b); }

    vector<vector<T>> s;

    // Initialize RMQ structure; time: O(n lg n)
    RMQ(const vector<T>& vec = {}) {
        s = {vec};
        for (int h = 1; h <= sz(vec); h *= 2) {
            s.emplace_back();
            auto& prev = s[sz(s)-2];
            rep(i, 0, sz(vec)-h*2+1)
                s.back().pb(f(prev[i], prev[i+h]));
        }
    }

    // Query f(s[b], ... ,s[e-1]); time: O(1)
    T query(int b, int e) {
        if (b >= e) return ID;
        int k = 31 - __builtin_clz(e-b);
        return f(s[k][b], s[k][e - (1<<k)]);
    }
};
```

**structures/segment\_tree.h**

49

```
// Optionally dynamic segment tree with lazy
// propagation. Configure by modifying:
// - T - data type for updates (stored type)
```

```
// - ID - neutral element for extra
// - Node - details in comments
struct SegmentTree {
    using T = int;
    static constexpr T ID = 0; // +
    // static constexpr T ID = INT_MIN; // max/=

    struct Node {
        T extra{ID}; // Lazy propagated value
        // Aggregates: sum, max, count of max
        T sum{0}, great{INT_MIN}, nGreat{0};

        // Initialize node with default value x
        void init(T x, int size) {
            sum = x*size; great = x; nGreat = size;
        }

        // Merge with node R on the right
        void merge(const Node& R) {
            if (great < R.great) nGreat = R.nGreat;
            else if (great == R.great) nGreat += R.nGreat;

            sum += R.sum;
            great = max(great, R.great);
        }

        // + version
        // Apply modification to node, return
        // value to be applied to node on right
        T apply(T x, int size) {
            extra += x;
            sum += x*size;
            great += x;
            return x;
        }
    };

    // MAX
    // T apply(T x, int size) {
    //     if (great <= x) nGreat = size;
    //     extra = max(extra, x);
    //     great = max(great, x);
    //     // sum doesn't work here
    //     return x;
    // }

    // =
    // T apply(T x, int size) {
    //     extra = x;
    //     sum = x*size;
    //     great = x;
    //     nGreat = size;
    //     return x;
    // }

    vector<Node> V;
    int len;
    // vector<array<int, 3>> links; // [DYNAMIC]
    // T defVal; // [DYNAMIC]

    SegmentTree(int n=0, T def=ID) { init(n, def); }

    void init(int n, T def) {
        for (len = 1; len < n; len *= 2);

        // [STATIC] version
```

```

V.assign(len*2, {});
rep(i, len, len+n) V[i].init(def, 1);
for (int i = len-1; i > 0; i--) update(i);

// [DYNAMIC] version
// defVal = def;
// links.assign(2, {-1, -1, len});
// V.assign(2, {});
// V[1].init(def, len);
}

// [STATIC] version
int getChild(int i, int j) { return i*2+j; }

// [DYNAMIC] version
// int getChild(int i, int j) {
// if (links[i][j] < 0) {
// int size = links[i][2] / 2;
// links[i][j] = sz(V);
// links.push_back({ -1, -1, size });
// V.emplace_back();
// V.back().init(defVal, size);
// }
// return links[i][j];
// }

int L(int i) { return getChild(i, 0); }
int R(int i) { return getChild(i, 1); }

void update(int i) {
    int a = L(i), b = R(i);
    V[i] = {};
    V[i].merge(V[a]);
    V[i].merge(V[b]);
}

void push(int i, int size) {
    T e = V[i].extra;
    if (e != ID) {
        e = V[L(i)].apply(e, size/2);
        V[R(i)].apply(e, size/2);
        V[i].extra = ID;
    }
}

// Modify [vBegin;end) with x; time: O(lg n)
T modify(int vBegin, int vEnd, T x,
        int i = 1,
        int begin = 0, int end = -1) {
    if (end < 0) end = len;
    if (vEnd <= begin || end <= vBegin)
        return x;

    if (vBegin <= begin && end <= vEnd) {
        return V[i].apply(x, end-begin);
    }

    int mid = (begin + end) / 2;
    push(i, end-begin);
    x = modify(vBegin, vEnd, x, L(i), begin, mid);
    x = modify(vBegin, vEnd, x, R(i), mid, end);
    update(i);
    return x;
}

// Query [vBegin;vEnd); time: O(lg n)

```

```

// Returns base nodes merged together
Node query(int vBegin, int vEnd, int i = 1,
        int begin = 0, int end = -1) {
    if (end < 0) end = len;
    if (vEnd <= begin || end <= vBegin)
        return {};
    if (vBegin <= begin && end <= vEnd)
        return V[i];

    int mid = (begin + end) / 2;
    push(i, end-begin);
    Node x = query(vBegin, vEnd, L(i), begin, mid);
    x.merge(query(vBegin, vEnd, R(i), mid, end));
    return x;
}

// TODO: generalize?
// Find longest suffix of given interval
// such that max value is smaller than val.
// Returns suffix begin index; time: O(lg n)
T search(int vBegin, int vEnd, int val,
        int i=1, int begin=0, int end=-1) {
    if (end < 0) end = len;
    if (vEnd <= begin || end <= vBegin)
        return begin;

    if (vBegin <= begin && end <= vEnd) {
        if (V[i].great < val) return begin;
        if (begin+1 == end) return end;
    }

    int mid = (begin+end) / 2;
    push(i, end-begin);

    int ind = search(vBegin, vEnd, val,
        R(i), mid, end);

    if (ind > mid) return ind;
    return search(vBegin, vEnd, val,
        L(i), begin, mid);
}

```

## structures/segment\_tree\_beats.h 50

```

constexpr ll INF = 1e18;

// Segment tree with min/+ update and
// sum/max query; time: amortized O(n lg^2 n)
// or O(n lg n) if not using + operation
struct SegmentTree {
    using T = ll;
    vector<T> sum, plus, max1, max2, cnt1;
    int len;

    SegmentTree(int n = 0) {
        for (len = 1; len < n; len *= 2);
        sum.resize(len*2);
        plus.resize(len*2);
        max1.resize(len*2);
        max2.assign(len*2, -INF);
        cnt1.assign(len*2, 1);
        for (int i = len-1; i > 0; i--) update(i);
    }

    void apply(int i, T m, T p, int size) {
        plus[i] += p; sum[i] += p*size;
        max1[i] += p; max2[i] += p;
    }
}

```

```

if (m < max1[i]) {
    sum[i] -= (max1[i]-m)*cnt1[i];
    max1[i] = m;
}

void update(int i) {
    int a = i*2, b = i*2+1;
    sum[i] = sum[a] + sum[b];
    plus[i] = 0;
    max1[i] = max1[a];
    max2[i] = max2[a];
    cnt1[i] = cnt1[a];

    if (max1[b] > max1[i]) {
        max2[i] = max1[i];
        max1[i] = max1[b];
        cnt1[i] = cnt1[b];
    } else if (max1[b] == max1[i]) {
        cnt1[i] += cnt1[b];
    } else if (max1[b] > max2[i]) {
        max2[i] = max1[b];
    }
    max2[i] = max(max2[i], max2[b]);
}

void push(int i, int s) {
    rep(j, 0, 2)
        apply(i*2+j, max1[i], plus[i], s/2);
    plus[i] = 0;
}

// Apply min with x on [vBegin;vEnd)
// time: amortized O(lg n) or O(lg^2 n)
void setMin(int vBegin, int vEnd, T x,
        int i = 1,
        int begin = 0, int end = -1) {
    if (end < 0) end = len;
    if (vEnd <= begin || end <= vBegin ||
        max1[i] < x) return;

    if (begin >= vBegin && end <= vEnd &&
        max2[i] < x)
        return apply(i, x, 0, end-begin);

    int mid = (begin+end) / 2;
    push(i, end-begin);
    setMin(vBegin, vEnd, x, i*2, begin, mid);
    setMin(vBegin, vEnd, x, i*2+1, mid, end);
    update(i);
}

// Add x on [vBegin;vEnd); time: O(lg n)
void add(int vBegin, int vEnd, T x,
        int i = 1,
        int begin = 0, int end = -1) {
    if (end < 0) end = len;
    if (vEnd <= begin || end <= vBegin) return;
    if (begin >= vBegin && end <= vEnd)
        return apply(i, INF, x, end-begin);

    int mid = (begin+end) / 2;
    push(i, end-begin);
    add(vBegin, vEnd, x, i*2, begin, mid);
    add(vBegin, vEnd, x, i*2+1, mid, end);
    update(i);
}

int mid = (begin+end) / 2;
push(i, end-begin);
add(vBegin, vEnd, x, i*2, begin, mid);
add(vBegin, vEnd, x, i*2+1, mid, end);
update(i);

```

```

}

// Query sum of [vBegin;vEnd); time: O(lg n)
T getSum(int vBegin, int vEnd, int i = 1,
        int begin = 0, int end = -1) {
    if (end < 0) end = len;
    if (vEnd<=begin || end<=vBegin) return 0;
    if (vBegin <= begin && end <= vEnd)
        return sum[i];
    int mid = (begin+end) / 2;
    push(i, end-begin);
    return getSum(vBegin, vEnd, i*2, begin, mid) +
        getSum(vBegin, vEnd, i*2+1, mid, end);
}

// Query max of [vBegin;vEnd); time: O(lg n)
T getMax(int vBegin, int vEnd, int i = 1,
        int begin = 0, int end = -1) {
    if (end < 0) end = len;
    if (vEnd <= begin || end <= vBegin)
        return -INF;
    if (vBegin <= begin && end <= vEnd)
        return max1[i];
    int mid = (begin+end) / 2;
    push(i, end-begin);
    return max(
        getMax(vBegin, vEnd, i*2, begin, mid),
        getMax(vBegin, vEnd, i*2+1, mid, end)
    );
}
};

```

## structures/segment\_tree\_point.h 51

```

// Segment tree (point, interval)
// Configure by modifying:
// - T - stored data type
// - ID - neutral element for query operation
// - merge(a, b) - combine results
struct SegmentTree {
    using T = int;
    static constexpr T ID = INT_MIN;
    static T merge(T a, T b) { return max(a,b); }

    vector<T> V;
    int len;

    SegmentTree(int n = 0, T def = ID) {
        for (len = 1; len < n; len *= 2);
        V.resize(len*2, ID);
        rep(i, 0, n) V[len+i] = def;
        for (int i = len-1; i > 0; i--)
            V[i] = merge(V[i*2], V[i*2+1]);
    }

    void set(int i, T val) {
        V[i+=len] = val;
        while ((i/=2) > 0)
            V[i] = merge(V[i*2], V[i*2+1]);
    }

    T query(int begin, int end) {
        begin += len; end += len-1;
        if (begin > end) return ID;
        if (begin == end) return V[begin];
        T x = merge(V[begin], V[end]);
    }
}

```

```

while (begin/2 < end/2) {
    if (~begin&1) x = merge(x, V[begin^1]);
    if (end&1) x = merge(x, V[end^1]);
    begin /= 2; end /= 2;
}
return x;
};

```

**constexpr** SegmentTree::T SegmentTree::ID;

## structures/treap.h 52

```

// "Set" of implicit keyed treaps; space: O(n)
// Treaps are distinguished by roots indices
// Put any additional data in Node struct.
struct Treap {
    struct Node {
        int E[2] = {-1, -1}, weight{rand()};
        int size[1], par{-1};
        bool flip{false}; // Is interval reversed?
    };

    vector<Node> G;

    // Initialize structure for n nodes; O(n)
    // Each node is separate treap,
    // use join() to construct sequence.
    Treap(int n = 0) : G(n) {}

    int size(int x) { // Returns subtree size
        return (x >= 0 ? G[x].size : 0);
    }

    void push(int x) { // Propagates down stuff
        if (x >= 0 && G[x].flip) {
            G[x].flip = 0;
            swap(G[x].E[0], G[x].E[1]);
            each(e, G[x].E) if (e>=0) G[e].flip ^= 1;
        } // + any other lazy operations
    }

    void update(int x) { // Updates aggregates
        if (x >= 0) {
            int& s = G[x].size = 1;
            G[x].par = -1;
            each(e, G[x].E) if (e >= 0) {
                s += G[e].size;
                G[e].par = x;
            }
        } // + any other aggregates
    }

    // Split treap x by index i into l and r
    // average time: O(lg n)
    void split(int x, int& l, int& r, int i) {
        push(x); l = r = -1;
        if (x < 0) return;
        int key = size(G[x].E[0]);
        if (i <= key) {
            split(G[x].E[0], l, G[x].E[0], i);
            r = x;
        } else {
            split(G[x].E[1], G[x].E[1], r, i-key-1);
            l = x;
        }
        update(x);
    }

```

```

}

// Join two treaps in given order; O(lg n)
int join(int l, int r) {
    push(l); push(r);
    if (l < 0 || r < 0) return max(l, r);

    if (G[l].weight < G[r].weight) {
        G[l].E[1] = join(G[l].E[1], r);
        update(l);
        return l;
    }

    G[r].E[0] = join(l, G[r].E[0]);
    update(r);
    return r;
}

// Find node with index i in treap x; O(lg n)
int find(int x, int i) {
    while (x >= 0) {
        push(x);
        int key = size(G[x].E[0]);
        if (key == i) return x;
        x = G[x].E[key < i];
        if (key < i) i -= key+1;
    }
    return x;
}

// Reverse interval [l;r] in treap x; O(lg n)
int reverse(int x, int l, int r) {
    int a, b, c;
    split(x, b, c, r);
    split(b, a, b, l);
    if (b >= 0) G[b].flip ^= 1;
    return join(join(a, b), c);
}

// Find root of treap containing x; O(lg n)
int root(int x) {
    while (G[x].par >= 0) x = G[x].par;
    return x;
}

```

## structures/ext/hash\_table.h 53

```

#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
// gp_hash_table<K, V> = faster unordered_set

// Anti-anti-hash
const size_t HXOR = mt19937_64(time(0))();
template<class T> struct SafeHash {
    size_t operator()(const T& x) const {
        return hash<T>(x ^ T(HXOR));
    }
};

```

## structures/ext/rope.h 54

```

#include <ext/rope>
using namespace __gnu_cxx;
// rope<T> = implicit cartesian tree

```

## structures/ext/tree.h 55

```

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

template<class T, class Cmp = less<T>>
using ordered_set = tree<
    T, null_type, Cmp, rb_tree_tag,
    tree_order_statistics_node_update
>;

// Standard set functions and:
// t.order_of_key(key) - index of first >= key
// t.find_by_order(i) - find i-th element
// t1.join(t2) - assuming t1<>t2 merge t2 to t1

structures/ext/trie.h 56
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/trie_policy.hpp>
using namespace __gnu_pbds;

using pref_trie = trie<
    string, null_type,
    trie_string_access_traits<>, pat_trie_tag,
    trie_prefix_search_node_update
>;

```

## text/aho\_corasick.h 57

```

constexpr char AMIN = 'a'; // Smallest letter
constexpr int ALPHA = 26; // Alphabet size

// Aho-Corasick algorithm for linear-time
// multiple pattern matching.
// Add patterns using add(), then call build().
struct Aho {
    vector<array<int, ALPHA>> nxt{1};
    Vi suf = {-1}, accLink = {-1};
    vector<Vi> accept{1};

    // Add string with given ID to structure
    // Returns index of accepting node
    int add(const string& str, int id) {
        int i = 0;
        each(c, str) {
            if (!nxt[i][c-AMIN]) {
                nxt[i][c-AMIN] = sz(nxt);
                nxt.pb({}); suf.pb(-1);
                accLink.pb(1); accept.pb({});
            }
            i = nxt[i][c-AMIN];
        }
        accept[i].pb(id);
        return i;
    }
}

```

```

// Build automata; time: O(V*ALPHA)
void build() {
    queue<int> que;
    each(e, nxt[0]) if (e) {
        suf[e] = 0; que.push(e);
    }
    while (!que.empty()) {
        int i = que.front(), s = suf[i], j = 0;
        que.pop();
        each(e, nxt[i]) {
            if (e) que.push(e);
            (e ? suf[e] : e) = nxt[s][j++];
        }
    }
}

```

```

}
accLink[i] = (accept[s].empty() ?
    accLink[s] : s);
}

// Append 'c' to state 'i'
int next(int i, char c) {
    return nxt[i][c-AMIN];
}

// Call 'f' for each pattern accepted
// when in state 'i' with its ID as argument.
// Return true from 'f' to terminate early.
// Calls are in decreasing length order.
template<class F> void accepted(int i, F f) {
    while (i != -1) {
        each(a, accept[i]) if (f(a)) return;
        i = accLink[i];
    }
}
};

```

## text/kmp.h 58

```

// Computes pfsuf array; time: O(n)
// ps[i] = max pfsuf of [0;i]; ps[0] := -1
template<class T> Vi kmp(const T& str) {
    Vi ps; ps.pb(-1);
    each(x, str) {
        int k = ps.back();
        while (k >= 0 && str[k] != x) k = ps[k];
        ps.pb(k+1);
    }
    return ps;
}

// Finds occurrences of pat in vec; time: O(n)
// Returns starting indices of matches.
template<class T>
Vi match(const T& str, T pat) {
    int n = sz(pat);
    pat.pb(-1); // SET TO SOME UNUSED CHARACTER
    pat.insert(pat.end(), all(str));
    Vi ret, ps = kmp(pat);
    rep(i, 0, sz(ps)) {
        if (ps[i] == n) ret.pb(i-2*n-1);
    }
    return ret;
}

```

## text/kmr.h 59

```

// KMR algorithm for O(1) lexicographical
// comparison of substrings.
struct KMR {
    vector<Vi> ids;

    KMR() {}

    // Initialize structure; time: O(n lg^2 n)
    // You can change str type to Vi freely.
    explicit KMR(const string& str) {
        ids.clear();
        ids.pb(Vi(all(str)));

        for (int h = 1; h <= sz(str); h *= 2) {
            vector<pair<Pii, int>> tmp;
        }
    }
}

```

```

rep(j, 0, sz(str)) {
    int a = ids.back()[j], b = -1;
    if (j+h < sz(str)) b = ids.back()[j+h];
    tmp.pb({ {a, b}, j });
}

sort(all(tmp));
ids.emplace_back(sz(tmp));

int n = 0;
rep(j, 0, sz(tmp)) {
    if (j > 0 && tmp[j-1].x != tmp[j].x)
        n++;
    ids.back()[tmp[j].y] = n;
}
}

// Get representative of [begin;end); O(1)
Pii get(int begin, int end) {
    if (begin >= end) return {0, 0};
    int k = 31 - __builtin_clz(end-begin);
    return {ids[k][begin], ids[k][end-(1<<k)]};
}

// Compare [b1;e1) with [b2;e2); O(1)
// Returns -1 if <, 0 if ==, 1 if >
int cmp(int b1, int e1, int b2, int e2) {
    int l1 = e1-b1, l2 = e2-b2;
    int l = min(l1, l2);
    Pii x = get(b1, b1+l), y = get(b2, b2+l);

    if (x == y) return (l1 > l2) - (l1 < l2);
    return (x > y) - (x < y);
}

// Compute suffix array of string; O(n)
Vi sufArray() {
    Vi sufs(sz(ids.back()));
    rep(i, 0, sz(ids.back()))
        sufs[ids.back()[i]] = i;
    return sufs;
}
};

```

## text/lcp.h 60

```

// Compute Longest Common Prefix array for
// given string and it's suffix array; O(n)
// In order to compute suffix array use kmr.h
// or suffix_array_linear.h
template<class T>
Vi lcpArray(const T& str, const Vi& sufs) {
    int n = sz(str), k = 0;
    Vi pos(n, lcp(n-1));
    rep(i, 0, n) pos[sufs[i]] = i;
    rep(i, 0, n) {
        if (pos[i] < n-1) {
            int j = sufs[pos[i]+1];
            while (i+k < n && j+k < n &&
                    str[i+k] == str[j+k]) k++;
            lcp[pos[i]] = k;
        }
        if (k > 0) k--;
    }
    return lcp;
}

```

## text/lyndon\_factorization.h 61

```

// Compute Lyndon factorization for s; O(n)
// Word is simple iff it's stricly smaller
// than any of it's nontrivial suffixes.
// Lyndon factorization is division of string
// into non-increasing simple words.
// It is unique.
vector<string> duval(const string& s) {
    int n = sz(s), i = 0;
    vector<string> ret;
    while (i < n) {
        int j = i+1, k = i;
        while (j < n && s[k] <= s[j])
            k = (s[k] < s[j] ? i : k+1), j++;
        while (i <= k)
            ret.pb(s.substr(i, j-k)), i += j-k;
    }
    return ret;
}

```

## text/main\_lorentz.h 62

```

#include "z_function.h"

struct Sqr {
    int begin, end, len;
};

// Main-Lorentz algorithm for finding
// all squares in given word; time: O(n lg n)
// Results are in compressed form:
// (b, e, l) means that for each b <= i < e
// there is square at position i of size 2l.
// Each square is present in only one interval.
vector<Sqr> lorentz(const string& s) {
    int n = sz(s);
    if (n <= 1) return {};
    auto a = s.substr(0, n/2), b = s.substr(n/2);

    auto ans = lorentz(a);
    each(p, lorentz(b))
        ans.pb({p.begin+n/2, p.end+n/2, p.len});

    string ra(a.rbegin(), a.rend());
    string rb(b.rbegin(), b.rend());
}

```

```

rep(j, 0, 2) {
    Vi z1 = prefPref(ra), z2 = prefPref(b+a);
    z1.pb(0); z2.pb(0);

    rep(c, 0, sz(a)) {
        int l = sz(a)-c;
        int x = c - min(l-1, z1[l]);
        int y = c - max(l-z2[sz(b)+c], j);
        if (x > y) continue;

        if (j)
            ans.pb({n-y-l*2, n-x-l*2+1, l});
        else
            ans.pb({x, y+1, l});
    }

    a.swap(rb);
    b.swap(ra);
}

```

```

return ans;
}

```

## text/manacher.h 63

```

// Manacher algorithm; time: O(n)
// Finds largest radiuses for palindromes:
// r[2*i] = for center at i (single letter = 1)
// r[2*i+1] = for center between i and i+1
template<class T> Vi manacher(const T& str) {
    int n = sz(str)*2, c = 0, e = 1;
    Vi r(n, 1);
    auto get = [&](int i) { return i%2 ? 0 :
        (i >= 0 && i < n ? str[i/2] : i); };

    rep(i, 0, n) {
        if (i < e) r[i] = min(r[c*2-i], e-i);
        while (get(i-r[i]) == get(i+r[i])) r[i]++;
        if (i+r[i] > e) c = i, e = i+r[i]-1;
    }

    rep(i, 0, n) r[i] /= 2;
    return r;
}

```

## text/min\_rotation.h 64

```

// Find lexicographically smallest
// rotation of s; time: O(n)
// Returns index where shifted word starts.
// You can use std::rotate to get the word:
// rotate(s.begin(), v.begin()+minRotation(v),
//        v.end());
int minRotation(string s) {
    int a = 0, n = sz(s); s += s;
    rep(b, 0, n) rep(i, 0, n) {
        if (a+i == b || s[a+i] < s[b+i]) {
            b += max(0, i-1); break;
        }
        if (s[a+i] > s[b+i]) {
            a = b; break;
        }
    }
    return a;
}

```

## text/palindromic\_tree.h 65

```

constexpr int ALPHA = 26; // Set alphabet size

// Tree of all palindromes in string,
// constructed online by appending letters.
// space: O(n*ALPHA); time: O(n)
// Code marked with [EXT] is extension for
// calculating minimal palindrome partition
// in O(n lg n). Can also be modified for
// similar dynamic programmings.
struct PalTree {
    Vi txt; // Text for which tree is built

    // Node 0 = empty palindrome (root of even)
    // Node 1 = "-1" palindrome (root of odd)
    Vi len{0, -1}; // Lengths of palindromes
    Vi link{1, 0}; // Suffix palindrome links
    // Edges to next palindromes
    vector<array<int, ALPHA>> to{ {}, {} };
    int last{0}; // Current node (max suffix pal)
}

```

```

Vi diff{0, 0}; // len[i]-len[link[i]] [EXT]
Vi slink{0, 0}; // Serial links [EXT]
Vi series{0, 0}; // Series DP answer [EXT]
Vi ans{0}; // DP answer for prefix[EXT]

```

```

int ext(int i) {
    while (len[i]+2 > sz(txt) ||
            txt[sz(txt)-len[i]-2] != txt.back())
        i = link[i];
    return i;
}

// Append letter from [0;ALPHA); time: O(1)
// (or O(lg n) if [EXT] is enabled)
void add(int x) {
    txt.pb(x);
    last = ext(last);

    if (!to[last][x]) {
        len.pb(len[last]+2);
        link.pb(to[ext(link[last])][x]);
        to[last][x] = sz(to);
        to.emplace_back();

        // [EXT]
        diff.pb(len.back() - len[link.back()]);
        slink.pb(diff.back() == diff[link.back()]
                ? slink[link.back()] : link.back());
        series.pb(0);
        // [/EXT]
    }
    last = to[last][x];

    // [EXT]
    ans.pb(INT_MAX);
    for (int i=last; len[i] > 0; i=slink[i]) {
        series[i] = ans[sz(ans) - len[slink[i]]
                - diff[i] - 1];

        if (diff[i] == diff[link[i]])
            series[i] = min(series[i],
                    series[link[i]]);

        ans.back() = min(ans.back(), series[i]+1);
    }
    // [/EXT]
}
};

```

## text/suffix\_array\_linear.h 66

```

#include "../util/radix_sort.h"

// KS algorithm for suffix array; time: O(n)
// Input values are assumed to be in [1;k]
Vi sufArray(Vi str, int k) {
    int n = sz(str);
    Vi suf(n);
    str.resize(n+15);

    if (n < 15) {
        iota(all(suf), 0);
        rep(j, 0, n) countSort(suf,
                [&](int i) { return str[i+n-j-1]; }, k);
        return suf;
    }

    // Compute triples codes
}

```

```

Vi tmp, code(n+2);
rep(i, 0, n) if (i % 3) tmp.pb(i);

rep(j, 0, 3) countSort(tmp,
    [&](int i) { return str[i-j+2]; }, k);

int mc = 0, j = -1;

each(i, tmp) {
    code[i] = mc += (j == -1 ||
        str[i] != str[j] ||
        str[i+1] != str[j+1] ||
        str[i+2] != str[j+2]);
    j = i;
}

// Compute suffix array of 2/3
tmp.clear();
for (int i=1; i < n; i += 3) tmp.pb(code[i]);
tmp.pb(0);
for (int i=2; i < n; i += 3) tmp.pb(code[i]);
tmp = sufArray(move(tmp), mc);

// Compute partial suffix arrays
Vi third;
int th = (n+4) / 3;
if (n%3 == 1) third.pb(n-1);

rep(i, 1, sz(tmp)) {
    int e = tmp[i];
    tmp[i-1] = (e < th ? e*3+1 : (e-th)*3+2);
    code[tmp[i-1]] = i;
    if (e < th) third.pb(e*3);
}

tmp.pop_back();
countSort(third,
    [&](int i) { return str[i]; }, k);

// Merge suffix arrays
merge(all(third), all(tmp), suf.begin(),
    [&](int l, int r) {
    while (l%3 == 0 || r%3 == 0) {
        if (str[l] != str[r])
            return str[l] < str[r];
        l++; r++;
    }
    return code[l] < code[r];
});

return suf;
}

// KS algorithm for suffix array; time: O(n)
Vi sufArray(const string& str) {
    return sufArray(Vi(all(str)), 255);
}

```

## text/z\_function.h 67

```

// Computes Z function array; time: O(n)
// zf[i] = max common prefix of str and str[i:]
template<class T> Vi prefPref(const T& str) {
    int n = sz(str), b = 0, e = 1;
    Vi zf(n);
    rep(i, 1, n) {
        if (i < e) zf[i] = min(zf[i-b], e-i);

```

```

        while (i+zf[i] < n &&
            str[zf[i]] == str[i+zf[i]]) zf[i]++;
        if (i+zf[i] > e) b = i, e = i+zf[i];
    }
    zf[0] = n;
    return zf;
}

```

## trees/centroid\_decomp.h 68

```

// Centroid decomposition; space: O(n lg n)
// UNTESTED
struct CentroidTree {
    // child[v] = children of v in centroid tree
    // ind[v][i] = index of vertex v in
    //             i-th centroid from root
    // subtree[v] = vertices in centroid subtree
    // dists[v] = distances from v to vertices
    //             in centroid subtree
    // par[v] = parent of v in centroid tree
    // depth[v] = depth of v in centroid tree
    // size[v] = size of centroid subtree of v
    vector<Vi> child, ind, dists, subtree;
    Vi par, depth, size;
    int root; // Root centroid

    CentroidTree() {}

    CentroidTree(vector<Vi>& G)
        : child(sz(G)), ind(sz(G)), dists(sz(G)),
          subtree(sz(G)), par(sz(G), -2),
            depth(sz(G)), size(sz(G)) {
        root = decomp(G, 0, 0);
    }

    int dfs(vector<Vi>& G, int i, int p) {
        size[i] = 1;
        each(e, G[i]) if (e != p && par[e] == -2)
            size[i] += dfs(G, e, i);
        return size[i];
    }

    void layer(vector<Vi>& G, int i,
        int p, int c, int d) {
        ind[i].pb(sz(subtree[c]));
        subtree[c].pb(i);
        dists[c].pb(d);
        each(e, G[i]) if (e != p && par[e] == -2)
            layer(G, e, i, c, d+1);
    }

    int decomp(vector<Vi>& G, int v, int d) {
        int p = -1, s = dfs(G, v, -1);
        bool ok = 1;
        while (ok) {
            ok = 0;
            each(e, G[v]) {
                if (e != p && par[e] == -2 &&
                    size[e] > s/2) {
                    p = v; v = e; ok = 1;
                    break;
                }
            }
        }

        par[v] = -1;
        size[v] = s;

```

```

        depth[v] = d;
        layer(G, v, -1, v, 0);

        each(e, G[v]) if (par[e] == -2) {
            int j = decomp(G, e, d+1);
            child[v].pb(j);
            par[j] = v;
        }
        return v;
    }
};

```

## trees/heavylight\_decomp.h 69

```

#include "../structures/segment_tree_point.h"

// Heavy-Light Decomposition of tree
// with subtree query support; space: O(n)
struct HLD {
    // Subtree of v = [pos[v]; pos[v]+size[v])
    // Chain with v = [chBegin[v]; chEnd[v])
    Vi par; // Vertex parent
    Vi size; // Vertex subtree size
    Vi depth; // Vertex distance to root
    Vi pos; // Vertex position in "HLD" order
    Vi chBegin; // Begin of chain with vertex
    Vi chEnd; // End of chain with vertex
    Vi order; // "HLD" preorder of vertices
    SegmentTree tree; // Verts are in HLD order

    HLD() {}

    // Initialize structure for tree G
    // and root r; time: O(n lg n)
    // MODIFIES ORDER OF EDGES IN G!
    HLD(vector<Vi>& G, int r)
        : par(sz(G)), size(sz(G)),
          depth(sz(G)), pos(sz(G)),
            chBegin(sz(G)), chEnd(sz(G)) {
        dfs(G, r, -1);
        decomp(G, r, -1, 0);
        tree = {sz(order)};
    }

    void dfs(vector<Vi>& G, int i, int p) {
        par[i] = p;
        size[i] = 1;
        depth[i] = p < 0 ? 0 : depth[p]+1;

        int& fs = G[i][0];
        if (fs == p) swap(fs, G[i].back());

        each(e, G[i]) if (e != p) {
            dfs(G, e, i);
            size[i] += size[e];
            if (size[e] > size[fs]) swap(e, fs);
        }
    }

    void decomp(vector<Vi>& G,
        int i, int p, int chb) {
        pos[i] = sz(order);
        chBegin[i] = chb;
        chEnd[i] = pos[i]+1;
        order.pb(i);

        each(e, G[i]) if (e != p) {

```

```

            if (e == G[i][0]) {
                decomp(G, e, i, chb);
                chEnd[i] = chEnd[e];
            } else {
                decomp(G, e, i, sz(order));
            }
        }

        // Get root of chain containing v
        int chRoot(int v) {return order[chBegin[v]];}

        // Level Ancestor Query; time: O(lg n)
        int laq(int i, int level) {
            while (true) {
                int k = pos[i] - depth[i] + level;
                if (k >= chBegin[i]) return order[k];
                i = par[chRoot(i)];
            }

            // Lowest Common Ancestor; time: O(lg n)
            int lca(int a, int b) {
                while (chBegin[a] != chBegin[b]) {
                    int ha = chRoot(a), hb = chRoot(b);
                    if (depth[ha] > depth[hb]) a = par[ha];
                    else b = par[hb];
                }
                return depth[a] < depth[b] ? a : b;
            }

            // Call func(chBegin, chEnd) on each path
            // segment; time: O(lg n * time of func)
            template<class T>
            void iterPath(int a, int b, T func) {
                while (chBegin[a] != chBegin[b]) {
                    int ha = chRoot(a), hb = chRoot(b);
                    if (depth[ha] > depth[hb]) {
                        func(chBegin[a], pos[a]+1);
                        a = par[ha];
                    } else {
                        func(chBegin[b], pos[b]+1);
                        b = par[hb];
                    }
                }

                if (pos[a] > pos[b]) swap(a, b);
                // Remove +1 from pos[a]+1 for vertices
                // queries (with +1 -> edges).
                func(pos[a]+1, pos[b]+1);
            }

            // Query path between a and b; O(lg^2 n)
            SegmentTree::T queryPath(int a, int b) {
                auto ret = SegmentTree::ID;
                iterPath(a, b, [&](int i, int j) {
                    ret = SegmentTree::merge(ret,
                        tree.query(i, j));
                });
                return ret;
            }

            // Query subtree of v; time: O(lg n)
            SegmentTree::T querySubtree(int v) {
                return tree.query(pos[v], pos[v]+size[v]);
            }

```

```

};

trees/lca.h 70
// LAQ and LCA using jump pointers
// space: O(n lg n)

struct LCA {
    vector<Vi> jumps;
    Vi level, pre, post;
    int cnt{0}, depth;

    LCA() {}

    // Initialize structure for tree G
    // and root r; time: O(n lg n)
    LCA(vector<Vi>& G, int r)
        : jumps(sz(G)), level(sz(G)),
          pre(sz(G)), post(sz(G)) {
        dfs(G, r, -1);
        depth = int(log2(sz(G))) + 2;
        rep(j, 0, depth) rep(i, 0, sz(G))
            jumps[i].pb(jumps[jumps[i][j]][j]);
    }

    void dfs(vector<Vi>& G, int i, int p) {
        level[i] = p < 0 ? 0 : level[p]+1;
        jumps[i].pb(p < 0 ? i : p);
        pre[i] = ++cnt;
        each(e, G[i]) if (e != p) dfs(G, e, i);
        post[i] = ++cnt;
    }

    // Check if a is ancestor of b; time: O(1)
    bool isAncestor(int a, int b) {
        return pre[a] <= pre[b] &&
            post[b] <= post[a];
    }

    // Lowest Common Ancestor; time: O(lg n)
    int operator()(int a, int b) {
        for (int j = depth; j >= 0; j--) {
            if (!isAncestor(jumps[a][j], b))
                a = jumps[a][j];
        }
        return isAncestor(a, b) ? a : jumps[a][0];
    }

    // Level Ancestor Query; time: O(lg n)
    int laq(int a, int lvl) {
        for (int j = depth; j >= 0; j--) {
            if (lvl < level[jumps[a][j]])
                a = jumps[a][j];
        }
        return level[a] <= lvl ? a : jumps[a][0];
    }

    // Get distance from a to b; time: O(lg n)
    int distance(int a, int b) {
        return level[a] + level[b] -
            level[operator()(a, b)]*2;
    }
};

trees/link_cut_tree.h 71
// Link/cut tree; space: O(n)
// Represents forest of (un)rooted trees.

```

```

struct LinkCutTree {
    vector<array<int, 2>> child;
    Vi par, prev, flip;

    // Initialize structure for n vertices; O(n)
    // At first there's no edges.
    LinkCutTree(int n = 0)
        : child(n, {-1, -1}), par(n, -1),
          prev(n, -1), flip(n, -1) {}

    void auxLink(int p, int i, int ch) {
        child[p][i] = ch;
        if (ch >= 0) par[ch] = p;
    }

    void push(int x) {
        if (x >= 0 && flip[x]) {
            flip[x] = 0;
            swap(child[x][0], child[x][1]);
            each(e, child[x]) if (e>=0) flip[e] ^= 1;
        }
    }

    void rot(int p, int i) {
        int x = child[p][i], g = par[x] = par[p];
        if (g >= 0) child[g][child[g][1] == p] = x;
        auxLink(p, i, child[x][!i]);
        auxLink(x, !i, p);
        swap(prev[x], prev[p]);
    }

    void splay(int x) {
        while (par[x] >= 0) {
            int p = par[x], g = par[p];
            push(g); push(p); push(x);
            bool f = (child[p][1] == x);

            if (g >= 0) {
                if (child[g][f] == p) { // zig-zig
                    rot(g, f); rot(p, f);
                } else { // zig-zag
                    rot(p, f); rot(g, !f);
                }
            } else { // zig
                rot(p, f);
            }
            push(x);
        }

    }

    void access(int x) {
        while (true) {
            splay(x);
            int p = prev[x];
            if (p < 0) break;

            prev[x] = -1;
            splay(p);

            int r = child[p][1];
            if (r >= 0) swap(par[r], prev[r]);
            auxLink(p, 1, x);
        }
    }

    void makeRoot(int x) {

```

```

        access(x);
        int& l = child[x][0];
        if (l >= 0) {
            swap(par[l], prev[l]);
            flip[l] ^= 1;
            l = -1;
        }
    }

    // Find representative of tree containing x
    int find(int x) { // time: amortized O(lg n)
        access(x);
        while (child[x][0] >= 0)
            push(x = child[x][0]);
        splay(x);
        return x;
    }

    // Add edge x-y; time: amortized O(lg n)
    void link(int x, int y) {
        makeRoot(x); prev[x] = y;
    }

    // Remove edge x-y; time: amortized O(lg n)
    void cut(int x, int y) {
        makeRoot(x); access(y);
        par[x] = child[y][0] = -1;
    }
};

util/bit_hacks.h 72
// __builtin_popcount - count number of 1 bits
// __builtin_clz - count most significant 0s
// __builtin_ctz - count least significant 0s
// __builtin_ffs - like ctz, but indexed from 1
// returns 0 for 0
// For ll version add ll to name

using ull = uint64_t;

#define T64(s,up) \
    for (ull i=0; i<64; i+=s*2) \
        for (ull j = i; j < i+s; j++) { \
            ull a = (M[j] >> s) & up; \
            ull b = (M[j+s] & up) << s; \
            M[j] = (M[j] & up) | b; \
            M[j+s] = (M[j+s] & (up<<s)) | a; \
        }

// Transpose 64x64 bit matrix
void transpose64(array<ull, 64>& M) {
    T64(1, 0x5555555555555555);
    T64(2, 0x3333333333333333);
    T64(4, 0xF0F0F0F0F0F0F0F);
    T64(8, 0xFF00FF00FF00FF);
    T64(16, 0xFFFF0000FFFF);
    T64(32, 0xFFFFFFFFFLL);
}

// Lexicographically next mask with same
// amount of ones.
int nextSubset(int v) {
    int t = v | (v - 1);
    return (t + 1) | (((~t & -~t) - 1) >>
        (__builtin_ctz(v) + 1));
}

```

```

util/bump_alloc.h 73
// Allocator, which doesn't free memory.

char mem[400<<20]; // Set memory limit
size_t nMem;

void* operator new(size_t n) {
    nMem += n; return &mem[nMem-n];
}
void operator delete(void*) {}

util/compress_vec.h 74
// Compress integers to range [0;n) while
// preserving their order; time: O(n lg n)
// Returns mapping: compressed -> original
Vi compressVec(vector<int*>& vec) {
    sort(all(vec),
        [](int* l, int* r) { return *l < *r; });
    Vi old;
    each(e, vec) {
        if (old.empty() || old.back() != *e)
            old.pb(*e);
        *e = sz(old)-1;
    }
    return old;
}

util/inversion_vector.h 75
// Get inversion vector for sequence of
// numbers in [0;n); ret[i] = count of numbers
// smaller than perm[i] to the left; O(n lg n)
Vi encodeInversions(Vi perm) {
    Vi odd, ret(sz(perm));
    int cont = 1;

    while (cont) {
        odd.assign(sz(perm)+1, 0);
        cont = 0;

        rep(i, 0, sz(perm)) {
            if (perm[i] % 2) odd[perm[i]]++;
            else ret[i] += odd[perm[i]+1];
            cont += perm[i] /= 2;
        }
        return ret;
    }

    // Count inversions in sequence of numbers
    // in [0;n); time: O(n lg n)
    ll countInversions(Vi perm) {
        ll ret = 0, cont = 1;
        Vi odd;

        while (cont) {
            odd.assign(sz(perm)+1, 0);
            cont = 0;

            rep(i, 0, sz(perm)) {
                if (perm[i] % 2) odd[perm[i]]++;
                else ret += odd[perm[i]+1];
                cont += perm[i] /= 2;
            }
        }
        return ret;
    }
}

```

```

}

util/longest_increasing_sub.h 76
// Longest Increasing Subsequence; O(n lg n)
int lis(const V& seq) {
    Vi dp(sz(seq), INT_MAX);
    each(c, seq) *lower_bound(all(dp), c) = c;
    return int(lower_bound(all(dp), INT_MAX)
        - dp.begin());
}

```

```

util/max_rects.h 77
struct MaxRect {
    // begin = first column of rectangle
    // end = first column after rectangle
    // hei = height of rectangle
    // touch = columns of height hei inside
    int begin, end, hei;
    Vi touch; // sorted increasing
};

```

```

// Given consecutive column heights find
// all inclusion-wise maximal rectangles
// contained in "drawing" of columns; time O(n)
vector<MaxRect> getMaxRects(Vi hei) {
    hei.insert(hei.begin(), -1);
    hei.pb(-1);
    Vi reach(sz(hei), sz(hei)-1);
    vector<MaxRect> ans;

    for (int i = sz(hei)-1; --i;) {
        int j = i+1, k = i;
        while (hei[j] > hei[i]) j = reach[j];
        reach[i] = j;

        while (hei[k] > hei[i-1]) {
            ans.pb({ i-1, 0, hei[k], {} });
            auto& rect = ans.back();

            while (hei[k] == rect.hei) {
                rect.touch.pb(k-1);
                k = reach[k];
            }
            rect.end = k-1;
        }
    }
    return ans;
}

```

```

util/mo.h 78
// Modified MO's queries sorting algorithm,
// slightly better results than standard.
// Allows to process q queries in O(n*sqrt(q))

struct Query {
    int begin, end;
};

```

```

// Get point index on Hilbert curve
ll hilbert(int x, int y, int s, ll c = 0) {
    if (s <= 1) return c;
    s /= 2; c *= 4;
    if (y < s)
        return hilbert(x&(s-1), y, s, c+(x>=s)+1);
    if (x < s)

```

```

        return hilbert(2*s-y-1, s-x-1, s, c);
    return hilbert(y-s, x-s, s, c+3);
}

// Get good order of queries; time: O(n lg n)
Vi moOrder(vector<Query>& queries, int maxN) {
    int s = 1;
    while (s < maxN) s *= 2;

    vector<ll> ord;
    each(q, queries)
        ord.pb(hilbert(q.begin, q.end, s));

    Vi ret(sz(ord));
    iota(all(ret), 0);
    sort(all(ret), [&](int l, int r) {
        return ord[l] < ord[r];
    });
    return ret;
}

```

```

util/parallel_binsearch.h 79
// Run 'count' binary searches on [begin;end),
// 'cmp' arguments:
// 1) vector<Pii>& - pairs (value, index)
//    which are queries if value of index is
//    greater or equal to value,
//    sorted by value
// 2) vector<bool>& - true at index i means
//    value of i-th query is >= queried value
// Returns vector of found values.
// Time: O((n+c) lg n), where c is cmp time.
template<class T>
Vi multiBS(int begin, int end, int count, T cmp) {
    vector<Pii> ranges(count, {begin, end});
    vector<Pii> queries(count);
    vector<bool> answers(count);

    rep(i, 0, count) queries[i] = { (begin+end)/2, i };

    for (int k = uplg(end-begin); k > 0; k--) {
        int last = 0, j = 0;
        cmp(queries, answers);

        rep(i, 0, sz(queries)) {
            Pii &q = queries[i], &r = ranges[q.y];
            if (q.x != last) last = q.x, j = i;

            (answers[i] ? r.x : r.y) = q.x;
            q.x = (r.x+r.y) / 2;

            if (!answers[i])
                swap(queries[i], queries[j++]);
        }
    }

    Vi ret;
    each(p, ranges) ret.pb(p.x);
    return ret;
}

```

```

util/radix_sort.h 80
Vi buf, cnt;

// Stable countingsort; time: O(k+sz(vec))
// See example usage in radixSort for pairs.

```

```

template<class F>
void countSort(Vi& vec, F key, int k) {
    vec.swap(buf);
    vec.resize(sz(buf));
    cnt.assign(k+1, 0);
    each(e, buf) cnt[key(e)]++;
    rep(i, 1, k+1) cnt[i] += cnt[i-1];
    for (int i = sz(vec)-1; i >= 0; i--)
        vec[--cnt[key(buf[i])]] = buf[i];
}

// Compute order of elems, k is max key; O(n)
Vi radixSort(const vector<Pii>& elems, int k) {
    Vi order(sz(elems));
    iota(all(order), 0);
    countSort(order,
        [&](int i) { return elems[i].y; }, k);
    countSort(order,
        [&](int i) { return elems[i].x; }, k);
    return order;
}

```