

.bashrc	1	structures/segment_tree_beats.h	45
.vimrc	2	structures/segment_tree_point.h	46
template.cpp	3	structures/treap.h	47
geometry/convex_hull.h	4	structures/ext/hash_table.h	48
geometry/convex_hull_dist.h	5	structures/ext/rope.h	49
geometry/convex_hull_sum.h	6	structures/ext/tree.h	50
geometry/line2.h	7	structures/ext/trie.h	51
geometry/rmst.h	8	text/aho_corasick.h	52
geometry/segment2.h	9	text/kmp.h	53
geometry/vec2.h	10	text/kmr.h	54
graphs/2sat.h	11	text/palindromic_tree.h	55
graphs/bellman_inequalities.h	12	text/suffix_array.h	56
graphs/dense_dfs.h	13	trees/centroid_decomp.h	57
graphs/edmonds_karp.h	14	trees/heavylight_decomp.h	58
graphs/push_relabel.h	15	trees/lca.h	59
graphs/turbo_matching.h	16	trees/link_cut_tree.h	60
math/bit_gauss.h	17	util/bit_hacks.h	61
math/bit_matrix.h	18	util/bump_alloc.h	62
math/fft.h	19	util/compress_vec.h	63
math/gauss.h	20	util/inversion_vector.h	64
math/miller_rabin.h	21	util/longest_increasing_sub.h	65
math/modinv_precompute.h	22	util/max_rects.h	66
math/modular.h	23	util/mo.h	67
math/modular64.h	24	util/parallel_binsearch.h	68
math/montgomery.h	25		
math/phi_large.h	26		
math/phi_precompute.h	27		
math/pi_large_precomp.h	28		
math/pollard_rho.h	29		
math/polynomial.h	30		
math/polynomial_interp.h	31		
math/sieve.h	32		
math/sieve_factors.h	33		
math/sieve_segmented.h	34		
structures/bitset_plus.h	35		
structures/fenwick_tree.h	36		
structures/fenwick_tree_2d.h	37		
structures/find_union.h	38		
structures/hull_offline.h	39		
structures/hull_online.h	40		
structures/max_queue.h	41		
structures/pairing_heap.h	42		
structures/rmq.h	43		
structures/segment_tree.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 pivot = points[0].yxPair();
    each(p,points) pivot = min(pivot,p.yxPair());
    vec2 ret{pivot.y, pivot.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 prev = hull.back()-hull[sz(hull)-2];
        vec2 cur = p - hull.back();
        if (prev.cross(cur) > 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)
// Set 'point' for each vertex and run rmst().
// Algorithm will compute RMST edges and save
// them in E for each vertex.
```

```

struct Edge {
    int dst, len;
};

struct Vert {
    Pii point, close;
    vector<Edge> E;
};

vector<Vert> G;
Vi merged;

void octant(Vi& S, int begin, int end) {
    if (begin+1 >= end) return;

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

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

    rep(i, begin, mid) {
        int v = S[i];
        Pii p = G[v].point;

        while (j < end) {
            int e = S[j];
            Pii q = G[e].point;
            if (q.x-q.y > p.x-p.y) break;
```

```

        int alt = q.x+q.y;
        if (alt < best.x) best = {alt, e};
        merged.pb(e);
        j++;
    }

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

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

ll rmst() {
    vector<pair<int, Pii>> edges;
    Vi sorted(sz(G));
    iota(all(sorted), 0);

    rep(i, 0, 4) {
        rep(j, 0, 2) {
            sort(all(sorted), [](int l, int r) {
                return G[l].point < G[r].point;
            });

            each(v, G) v.close = { INT_MAX, -1 };
            octant(sorted, 0, sz(sorted));

            rep(k, 0, sz(G)) {
                auto p = G[k].close;
                if (p.y != -1)
                    edges.pb({ p.x, {k, p.y} });
                G[k].point.x += -1;
            }

            each(v,G) v.point = {v.point.y,-v.point.x};
        }

        ll sum = 0;
        FAU fau(sz(G));
        sort(all(edges));

        each(e, edges) if (fau.join(e.y.x, e.y.y)) {
            sum += e.x;
            G[e.y.x].E.pb({e.y.y, e.x});
            G[e.y.y].E.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!
// Pass negative indices to represent negations
// (internally: positive = i*2-1, neg. = i*2-2)
struct SAT2 {
    vector<Vi> G;
    Vi order, values; // Also indexed from 1!
    vector<bool> flags;

    // Init n variables, you can add more later
    SAT2(int n = 0) { init(n); }
```

```

struct Ineq {
    ll a, b, c; //  $a - b \geq c$ 
    DBP(a, b, c);
};

// Solve system of inequalities of form  $a-b \geq c$ 
// using Bellman-Ford; time:  $O(n \cdot m)$ 
bool solveIneq(vector<Ineq>& edges,
               vector<ll>& vars) {
    rep(i, 0, sz(vars)) each(e, edges)

```

```
// 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) { init(n); }
void init(int n) {
    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/turbo_matching.h 16

```

// Find matching in bipartite graph; time: ?
struct Matching {
    vector<Vi> G; // Both sides together
    Vi match; // Matched vertices, -1 if none
    vector<bool> seen;

    // Initialize for n vertices
    Matching(int n = 0) { init(n); }
    void init(int n) { G.assign(n, {}); }

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

    // Add edge between u and v
    void addEdge(int u, int v) {
        G[u].pb(v); G[v].pb(u);
    }

    bool 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;
}

// Find maximum bipartite matching.
// Returns matching size (edge count)
int solve() {
    int n = 0, k = 1;
    match.assign(sz(G), -1);

    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;
}
};

```

math/bit_gauss.h 17

```

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 18

```

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) { init(n, m); }
    void init(int n, int m) {
        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/fft.h 19

```

#include "modular.h" // Only for Z_p version

// In-place Fast Fourier Transform
// over Z_p or complex; time: O(n lg n)
// DFT is in bit-reversed order!
// Default uncommented version is Z_p

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

// using Vfft = vector<complex<double>>;
using Vfft = vector<Zp>;
Vfft bases;

void initFFT(int n) { // n must be power of 2
    bases.resize(n+1, 1);
    //auto b = exp(complex<double>(0, 2*M_PI/n));
    auto b = Zp(ROOT).pow((1<<27) / n);
    rep(i, 1, n+1) bases[i] = b * bases[i-1];
}

template<int dir> // 1 for DFT, -1 for inverse
void fft(Vfft& buf) {
    int n = sz(buf), bits = 31-__builtin_clz(n);
    int i = (dir > 0 ? 0 : bits-1);

    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;
            int 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 = v2*base;
                buf[b] = v1 - v2;

```

```

        }
        buf[a] = v1 + v2;
    }
}

if (dir < 0) {
    Zp y = Zp(1) / n; // Or change to complex
    each(x, buf) x = x*y;
}

// Compute convolution of a and b; O(n lg n)
// Set a and b size appropriately
Vfft convolve(Vfft a, Vfft b) {
    fft<1>(a); fft<1>(b);
    rep(i, 0, sz(a)) a[i] = a[i]*b[i];
    fft<-1>(a);
    return a;
}

```

math/gauss.h 20

```

// 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 21

```

#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 22

constexpr ll MOD = 234567899;
vector<ll> modInv(MOD);

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

math/modular.h 23

constexpr ll MOD = 15*(1<<27)+1;

// Wrapper for modular arithmetic
struct Zp {
    ll x;
    Zp(ll a = 0) {
        if (a < 0) a = a%MOD + MOD;
        else if (a >= MOD*2) a %= MOD;
        else if (a >= MOD) a -= MOD;
        x = a;
    }

    Zp operator+(Zp r) const{ return x+r.x; }
    Zp operator-(Zp r) const{ return x-r.x+MOD; }
    Zp operator*(Zp r) const{ return x*r.x; }
    Zp operator/(Zp r) const{return x*r.inv().x;}
    Zp operator-() const{ return MOD-x; }
```

```
// Use modInv below for composite modulus
Zp inv() const { return pow(MOD-2); }

Zp pow(ll e) const {
    Zp t = 1, m = *this;
    while (e) {
        if (e & 1) t = t*m;
        e >>= 1; m = m*m;
    }
    return t;
}

#define OP(c) Zp& operator c##=(Zp r){ \
    return *this=*this c r; }
OP(+)OP(-)OP(*)OP(/)
void print() { cerr << x; }
};

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

math/modular64.h 24

// Modular arithmetic for modulus < 2^62

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

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

ll modMul(ll x, ll y, ll m) {
    ll t = 0;
    while (y) {
        if (y & 1) t = modAdd(t, x, m);
        y >>= 1;
        x = modAdd(x, x, m);
    }
    return t;
}

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

math/montgomery.h 25

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

constexpr ll MG_SHIFT = 32;
constexpr ll MG_MULT = 1LL << MG_SHIFT;
```

```
constexpr ll MG_MASK = MG_MULT - 1;
const ll MG_INV = MG_MULT-modInv(MOD, MG_MULT);

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

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

math/phi_large.h 26

#include "pollard_rho.h"

// Compute Euler's totient of large numbers
// time: O(n^(1/3)) <- factorization
// You need to initialize Pollard's rho first!
ll phi(ll n) {
    each(p, factorize(n)) n = n / p.x * (p.x-1);
    return n;
}

math/phi_precompute.h 27

constexpr int MAX_PHI = 10e6;
Vi phi(MAX_PHI+1);

// Precompute Euler's totients; time O(n lg n)
void calcPhi() {
    rep(i, 0, MAX_PHI+1) phi[i] = i;

    for (int i = 2; i <= MAX_PHI; i++)
        if (phi[i] == i)
            for (int j = i; j <= MAX_PHI; j += i)
                phi[j] = phi[j] / i * (i-1);
}

math/pi_large_precomp.h 28

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

ll sieveRange(ll from, ll to) {
    bitset<BUCKET_SIZE> elems;
    from = max(from, 2LL);
    to = max(from, to);
    each(p, primesList) {
        ll c = max((from+p-1) / p, 2LL);
        for (ll 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 (ll i = 0; i < MAX_N; i += BUCKET_SIZE) {
        ll 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))
ll countPrimes(ll from, ll to) {
    ll bFrom = from/BUCKET_SIZE+1,
        bTo = to/BUCKET_SIZE;
    if (bFrom > bTo) return sieveRange(from, to);
    ll ret = accumulate(precomputed+bFrom,
        precomputed+bTo, 0);
    ret += sieveRange(from, bFrom*BUCKET_SIZE);
    ret += sieveRange(bTo*BUCKET_SIZE, to);
    return ret;
}

math/pollard_rho.h 29

#include "sieve.h"
#include "miller_rabin.h"
#include "modular64.h"

using Factor = pair<ll, int>;

// Pollard's rho factorization algorithm
// Las Vegas version; time: n^(1/3)
// Before using, initialize sieve.
// Set MAX_P >= (max input number)^(1/3)
// Returns pairs (prime, power), sorted
vector<Factor> factorize(ll n) {
    vector<Factor> ret;
    each(p, primesList) if (n%p == 0) {
        ret.pb({ p, 0 });
        while (n%p == 0) {
            n /= p;
            ret.back().y++;
        }
    }

    if (n <= 1) return ret;
    if (isPrime(n)) {
        ret.pb({ n, 1 });
        return ret;
    }

    // Now n = p*q for some prime p and q
    for (ll a = 1;; a++) {
        ll 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) {
            n /= d;
        }
    }
}
```

```

    if (d > n) swap(d, n);
    ret.pb({ d, 1 });
    if (d == n) ret.back().y++;
    else ret.pb({ n, 1 });
    return ret;
}
}
}

```

math/polynomial.h 30

```

#include "modular.h"
#include "fft.h"

// Polynomial wrapper class; UNTESTED
struct Poly {
    using T = Zp; // Set appropriate type
    vector<T> C;

    Poly(int n = 0) { C.resize(n); }

    // Cut off trailing zeroes
    void reduce() {
        // Change here '.x' if not using Zp
        while (!C.empty() && C.back().x == 0)
            C.pop_back();
    }

    // Evaluate polynomial at x; time: O(n)
    T eval(T x) {
        T n = 0, y = 1;
        each(a, C) n += a*y, y *= x;
        return n;
    }

    // Add polynomial; time: O(n)
    Poly& operator+=(const Poly& r) {
        C.resize(max(sz(C), sz(r.C)));
        rep(i, 0, sz(r.C)) C[i] += r.C[i];
        reduce();
        return *this;
    }

    // Subtract polynomial; time: O(n)
    Poly& operator-=(const Poly& r) {
        C.resize(max(sz(C), sz(r.C)));
        rep(i, 0, sz(r.C)) C[i] -= r.C[i];
        reduce();
        return *this;
    }

    // Multiply by polynomial
    // time: O(n lg n) if using FFT
    Poly operator*(const Poly& r) const {
        int len = sz(C) + sz(r.C) - 1;
        Poly ret;
        ret.C.resize(len);

        if (sz(C)*sz(r.C) < 200) {
            // If you don't need FFT - use just this
            rep(i, 0, sz(C)) rep(j, 0, sz(r.C)) {
                ret.C[i+j] = ret.C[i+j] + C[i]*r.C[j];
            }
        } else {
            int n = 1 << (32 - __builtin_clz(len));
            Vfft a(n), b(n);

```

```

        rep(i, 0, sz(C)) a[i] = C[i];
        rep(i, 0, sz(r.C)) b[i] = r.C[i];

        initFFT(n);
        fft<1>(a); fft<1>(b);
        rep(i, 0, sz(a)) a[i] = a[i]*b[i];
        fft<-1>(a);

        rep(i, 0, len) ret.C[i] = a[i];
    }

    ret.reduce();
    return ret;
}

Poly operator+(const Poly& r) const {
    Poly l = *this; l += r; return l;
}
Poly operator-(const Poly& r) const {
    Poly l = *this; l -= r; return l;
}
Poly& operator*=(const Poly& r) {
    return *this = *this * r;
}

// Derivate polynomial; time: O(n)
void derivate() {
    rep(i, 1, sz(C)) C[i-1] = C[i]*i;
    C.pop_back();
}

// Integrate polynomial; time: O(n)
void integrate() {
    C.pb(0);
    rep(i, 1, sz(C)) C[i] = C[i-1]/i;
    C[0] = 0;
}
}

```

math/polynomial_interp.h 31

```

// 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 32

```

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 33

```

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 34

```

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 35

```

// 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 36

```

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

    vector<T> s;
    Fenwick(int n = 0) { init(n); }
    void init(int n) { s.assign(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 37

```

// 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) { init(n, m); }
    void init(int n, int m) {
        s.assign(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) {
                T& x = s[i*h+k]; x = f(x, 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-1]);
        return v;
    }
};

```

structures/find_union.h 38

```

// Disjoint set data structure; space: O(n)
// Operations work in amortized O(alfa(n))
struct FAU {
    Vi G;
    FAU(int n = 0) { init(n); }
    void init(int n) { G.assign(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 false;
        if (G[i] > G[j]) swap(i, j);
        G[i] += G[j]; G[j] = i;
        return true;
    }
};

```

structures/hull_offline.h 39

```

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
// TESTED ONLY FOR DOUBLES
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 40

```

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 41

```

// Queue with max query on contained elements
struct MaxQueue {
    using T = ll;
    deque<T> Q;

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

    // Pop from the back (v must be the last one)
    // time: amortized O(1)
    void pop(T v) {
        if (Q.back() == v) Q.pop_back();
    }

    // Get max element value; time: O(1)
    T max() const { return Q.back(); }
};

```

structures/pairing_heap.h 42

```

// 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

43

```

// 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;

    RMQ() {}
    RMQ(const vector<T>& vec) { init(vec); }

    // Initialize RMQ structure; time: O(n lg n)
    void init(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

44

```

// 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) {
        //     extra = x;
        //     sum = x*size;
        //     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

```

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;

    struct Node {
        T sum{0}, add{0}, max1{0}, max2{-INF}, cnt1{1};
        void apply(T m, T p, int size) {
            add += p; sum += p*size;
            max1 += p; max2 += p;
            if (m < max1) {
                sum -= (max1-m)*cnt1;
                max1 = m;
            }
        }
    };

    vector<Node> V;
    int len;

    SegmentTree(int n = 0) { init(n); }
    void init(int n) {
        for (len = 1; len < n; len *= 2);
        V.assign(len*2, {});
        for (int i = len-1; i > 0; i--) update(i);
    }

    void update(int i) {
        auto &v = V[i] = V[i*2], &r = V[i*2+1];
        v.add = 0;
        v.sum += r.sum;
        if (r.max1 > v.max1) {
            v.max2 = v.max1;
            v.max1 = r.max1;

```

```

    v.cnt1 = r.cnt1;
} else if (r.max1 == v.max1) {
    v.cnt1 += r.cnt1;
} else if (r.max1 > v.max2) {
    v.max2 = r.max1;
}
v.max2 = max(v.max2, r.max2);
}

void push(int i, int s) {
    V[i*2].apply(V[i].max1, V[i].add, s/2);
    // time: amortized O(lg n) or O(lg^2 n)
    V[i*2+1].apply(V[i].max1, V[i].add, s/2);
    V[i].add = 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 ||
        V[i].max1 < x) return;

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

    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) {
        V[i].apply(INF, x, end-begin);
        return;
    }

    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 sum(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 V[i].sum;
    int mid = (begin+end) / 2;
    push(i, end-begin);
    return sum(vBegin, vEnd, i*2, begin, mid) +

```

```

    sum(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 V[i].max1;
    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 46
// Simple segment tree (point-interval)
// Configure by modifying:
// - T - stored data type
// - ID - neutral element for QUERY operation
// - merge(a, b) - merge operation
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){init(n,def);}

    void init(int n, T def) {
        for (len = 1; len < n; len *= 2);
        V.assign(len+n, def);
        V.resize(len*2, ID);
        for (int i = len-1; i > 0; i--) update(i);
    }

    void update(int 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) update(i);
    }

    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

47

```

// "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) { init(n); }
    void init(int n) { G.clear(); G.resize(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

48

```

#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

49

```

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

```

structures/ext/tree.h

50

```

#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 51

```

#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 52

```

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 {
    struct Node {
        int nxt[ALPHA], suf{-1}, accLink{-1};
        Vi accept;
    };

    vector<Node> V;
    Aho() { init(); }
    void init() { V.assign(1, {}); }

    // Add string with given ID to structure
    void add(const string& str, int id) {
        int i = 0;
        each(c, str) {
            if (!V[i].nxt[c-AMIN]) {
                V[i].nxt[c-AMIN] = sz(V);
                V.emplace_back();
            }
            i = V[i].nxt[c-AMIN];
        }
        V[i].accept.pb(id);
    }

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

```

```

}

// Append 'c' to state 'i'
int nxt(int i, char c) {
    return V[i].nxt[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, V[i].accept) if (f(a)) return;
        i = V[i].accLink;
    }
}
}
};

```

text/kmp.h 53

```

// Computes prefsuf array; time: O(n)
// ps[i] = max prefsuf 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 54

```

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

    KMR() {}
    explicit KMR(const string& str){ init(str); }

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

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

            rep(j, 0, sz(str)) {
                int a = ids.back()[j], b = -1;

```

```

                if (j+h < sz(str)) b = ids.back()[j+h];
                pairs.pb({a, b}, j);
            }

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

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

```

text/palindromic_tree.h 55

```

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.h 56

```

#include "kmr.h"

// Compute suffix array for KMR-precomputed
// string; time: O(n lg n)
Vi sufArray(const KMR& kmr) {
    Vi sufs(sz(kmr.ids.back()));
    iota(all(sufs), 0);
    sort(all(sufs), [&](int l, int r) {
        return kmr.ids.back()[l]<kmr.ids.back()[r];
    });
    return sufs;
}

// Compute Longest Common Prefix array for
// given string and it's suffix array; O(n)
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;
}

```

trees/centroid_decomp.h 57

```

// Centroid decomposition; space: O(n lg n)
// UNTESTED

struct Vert {
    // cE = edges to children centroids
    // cLinks[i] = vertex index in
    //             i-th centroid from root
    // cSubtree = vertices in centroid subtree
    // cDists = distances to vertices in subtree
    // cParent = parent centroid
    // cDepth = depth in centroid tree
    // cSize = subtree size
    Vi E, cE, cLinks, cSubtree, cDists;
    int cParent{-2}, cDepth{-1}, cSize{-1};
};

vector<Vert> G;

// Check if DFS should go to edge e if it came
// to current node from p. Also ignores
// vertices with cParent != -2 (established
// centroids). Used by functions below
bool can(int e, int p) {
    return e != p && G[e].cParent == -2;
}

// Computes subtree sizes
int computeSize(int i, int p) {
    int& s = G[i].cSize = 1;
    each(e, G[i].E) if (can(e, p))
        s += computeSize(e, i);
    return s;
}

// Finds centroid
int getCentroid(int i) {
    int p = -1, size = computeSize(i, -1);
    bool ok = true;
    while (ok) {
        ok = false;
        each(e, G[i].E) if (can(e, p)) {
            if (G[e].cSize > size/2) {
                p = i; i = e; ok = true;
                break;
            }
        }
    }
    G[i].cSize = size;
    return i;
}

// Calculate centroid subtree data
void dfsLayer(int i, int p, int centr, int d) {
    G[i].cLinks.pb(sz(G[centr].cSubtree));
    G[centr].cSubtree.pb(i);
    G[centr].cDists.pb(d);

    each(e, G[i].E) if (can(e, p))
        dfsLayer(e, i, centr, d+1);
}

```

```

}

// Perform centroid decomposition of tree
// containing 'i'; time: O(n lg n)
// Returns root centroid.
int centroidDecomp(int i, int depth = 0) {
    i = getCentroid(i);
    G[i].cParent = -1;
    G[i].cDepth = depth;
    dfsLayer(i, -1, i, 0);

    each(e, G[i].E) if (can(e, -1)) {
        G[i].cE.pb(centroidDecomp(e, depth+1));
        G[G[i].cE.back()].cParent = i;
    }
    return i;
}

```

trees/heavylight_decomp.h 58

```

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

// Heavy-Light Decomposition of tree
// with subtree query support; space: O(n)

struct Vert {
    // par = parent, size = subtree size
    // depth = distance to root
    // pos = position in hldOrder
    // chBegin = begin of chain in hldOrder
    //           (closest to root)
    // chEnd = end of chain in hldOrder,
    //         EXCLUSIVE (furthest from root)
    Vi E; // Neighbours
    int par, size, depth, pos, chBegin, chEnd;
};

vector<Vert> G; // Graph
Vi hldOrder; // "HLD" preorder of vertices
SegmentTree hldTree; // Verts are in hldOrder

// Subtree of v = [G[v].pos;G[v].pos+G[v].size)
// Chain with v = [G[v].chBegin;G[v].chEnd)

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

void dfsSize(int i, int p, int d) {
    G[i].par = p;
    G[i].size = 1;
    G[i].depth = d;

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

    each(e, G[i].E) if (e != p) {
        dfsSize(e, i, d+1);
        G[i].size += G[e].size;
        if (G[e].size > G[fs].size) swap(e, fs);
    }
}

void dfsHld(int i, int p, int chb) {
    G[i].pos = sz(hldOrder);
    G[i].chBegin = chb;
}

```

```

G[i].chEnd = G[i].pos+1;
hldOrder.pb(i);

each(e, G[i].E) if (e != p) {
    if (e == G[i].E[0]) {
        dfsHld(e, i, chb);
        G[i].chEnd = G[e].chEnd;
    } else {
        dfsHld(e, i, sz(hldOrder));
    }
}

// Initialize decomposition; time: O(n)
void hld(int v) {
    hldOrder.clear();
    dfsSize(v, -1, 0);
    dfsHld(v, -1, 0);
    hldTree.init(sz(hldOrder), 0);
}

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

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

        if (ha.depth > hb.depth) {
            func(G[a].chBegin, G[a].pos+1);
            a = ha.par;
        } else {
            func(G[b].chBegin, G[b].pos+1);
            b = hb.par;
        }
    }

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

// Query path between a and b; time: 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,
        hldTree.query(i, j));
});
return ret;
}

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

```

trees/lca.h 59

```

// LAQ and LCA using jump pointers
// space: O(n lg n)

struct Vert {
    Vi edges, jumps;
    int level, pre, post;
};

int counter;
vector<Vert> G;

// Initialize jump pointers; time: O(n lg n)
void initLCA(int i, int parent=-1, int d=0) {
    G[i].level = d;
    G[i].jumps.pb(parent < 0 ? i : parent);
    G[i].pre = ++counter;
    each(e, G[i].edges) if (e != parent)
        initLCA(e, i, d+1);
    G[i].post = ++counter;

    if (parent < 0) {
        int depth = int(log2(sz(G))) + 2;
        rep(j, 0, depth) each(vert, G)
            vert.jumps.pb(G[vert.jumps[j]].jumps[j]);
    }
}

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

// Level Ancestor Query; time: O(lg n)
int laq(int a, int level) {
    for (int j = sz(G[a].jumps)-1; j >= 0; j--) {
        int k = G[a].jumps[j];
        if (level < G[k].level) a = k;
    }
    return G[a].level<=level ? a : G[a].jumps[0];
}

// Lowest Common Ancestor; time: O(lg n)
int lca(int a, int b) {
    for (int j = sz(G[a].jumps)-1; j >= 0; j--) {
        int k = G[a].jumps[j];
        if (!isAncestor(k, b)) a = k;
    }
    return isAncestor(a,b) ? a : G[a].jumps[0];
}

```

```
// Get distance between a and b; time: O(lg n)
int distance(int a, int b) {
    return G[a].level + G[b].level
        - G[lca(a, b)].level*2;
}
```

trees/link_cut_tree.h 60

```
// Link/cut tree; space: O(n)
// Represents forest of (un)rooted trees.
struct LinkCutTree {
    struct Node {
        int E[2] = {-1, -1}, par{-1}, prev{-1};
        bool flip{0};
    };

    vector<Node> G;

    // Initialize structure for n vertices; O(n)
    // Each vertex is separated.
    LinkCutTree(int n = 0) { init(n); }
    void init(int n) { G.assign(n, {}); }

    void auxLink(int par, int i, int child) {
        G[par].E[i] = child;
        if (child >= 0) G[child].par = par;
    }

    void push(int x) {
        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;
        }
    }

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

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

            if (g >= 0) {
                if (G[g].E[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 = G[x].prev;
```

```
        if (p < 0) break;

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

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

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

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

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

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

util/bit_hacks.h 61

```
// __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
// 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, 0xF0F0F0F0F0F0F0F0);
    T64(8, 0xFF00FF00FF00FF00);
    T64(16, 0xFFFF0000FFFF);
```

```
T64(32, 0xFFFFFFFFLL);
}

util/bump_alloc.h 62

// Allocator, which doesn't free memory.

char mem[512<<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 63

// Compress integers to range [0;n) while
// preserving their order; time: O(n lg n)
// Returns count n of different numbers
int compressVec(vector<int*>& vec) {
    sort(all(vec), [](int* l, int* r) {
        return *l < *r;
    });
    int last = *vec[0], i = 0;
    each(e, vec) {
        if (*e != last) i++;
        last = *e; *e = i;
    }
    return i+1;
}
```

util/inversion_vector.h 64

```
// 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 65

```
// Longest Increasing Subsequence; O(n lg n)
int lis(const Vi& 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 66

```
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 67

```
// 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 68

```
// 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;
}
```