Competitive Programming Notebook

As complexidades temporais são estimadas e simplificadas!

Sumário

Template

```cpp
// #pragma GCC target("popcnt")  // if solution involves bitset
#include <bits/stdc++.h>
using namespace std;

#ifdef croquete  // BEGIN TEMPLATE ----------------------|
#include "dbg/dbg.h"
#else
#define dbg(...)
#endif
#define ll          long long
#define vll         vector<ll>
#define vvll        vector<vll>
#define pll         pair<ll, ll>
#define vpll        vector<pll>
#define all(xs)     xs.begin(), xs.end()
#define rep(i, a, b) for (ll i = (a); i  < (ll)(b); ++i)
#define per(i, a, b) for (ll i = (a); i >= (ll)(b); --i)
#define eb          emplace_back
#define cinj        cin.iword(0)  = 1, cin
#define coutj       cout.iword(0) = 1, cout
template <typename T>  // read vector
istream& operator>>(istream& is, vector<T>& xs) {
    assert(!xs.empty());
    rep(i, is.iword(0), xs.size()) is >> xs[i];
    return is.iword(0) = 0, is;
} template <typename T>  // print vector
ostream& operator<<(ostream& os, vector<T>& xs) {
    rep(i, os.iword(0), xs.size()) os << xs[i] << ' ';
    return os.iword(0) = 0, os;
} void solve();
signed main() {
    cin.tie(0)->sync_with_stdio(0);
    ll t = 1;
    cin >> t;
    while (t--) solve();
}  // END TEMPLATE --------------------------------------|

void solve() {
}
```

Outros defines

```cpp
// BEGIN EXTRAS -----------------------------------------|
#define vvpll vector<vpll>
#define tll   tuple<ll, ll, ll>
#define vtll  vector<tll>
#define pd    pair<double, double>
#define x     first
#define y     second
map<char, pll> ds1 { {'R', {0, 1}}, {'D', {1, 0}}, {'L', {0, -1}}, {'U', {-1, 0}} };
vpll ds2 { {0, 1}, {1, 0}, {0, -1}, {-1, 0}, {1,  1}, {1, -1}, {-1,  1}, {-1, -1} };
vpll ds3 { {1, 2}, {2, 1}, {-1, 2}, {-2, 1}, {1, -2}, {2, -1}, {-1, -2}, {-2, -1} };
// END EXTRAS -------------------------------------------|
```

Flags

```
g++ -g -std=c++20 -fsanitize=undefined -fno-sanitize-recover -Wall -Wextra -Wshadow -Wconversion -Wduplicated-cond -Winvalid-pch -Wno-sign-compare -Wno-sign-conversion -Dcroquete -D_GLIBCXX_ASSERTIONS -fmax-errors=1
```

Debug

```cpp
#pragma once
#include <bits/stdc++.h>
using namespace std;
template <typename T> void p(T x) {
    int f = 0;
    #define D(d) cerr << "\e[94m" << (f++ ? d : "")
    if constexpr (!requires {cout << x;}) {
        cerr << '{';
        if constexpr (requires {get<0>(x);})
            apply([&](auto... args) {((D(","), p(args)), ...);}, x);
        else if constexpr (requires {x.pop();}) while (size(x)) {
            D(",");
            if constexpr (requires {x.top();}) p(x.top());
            else p(x.front());
            x.pop();
        } else for (auto i : x)
            (requires {begin(*begin(x));} ? cerr << "\n\t" : D(",")), p(i);
        cerr << '}';
    } else D("") << x;
} template <typename... A>
void pr(A... a) {int f = 0; ((D(" | "), p(a)), ...); cerr << "\e[m\n";}
#define dbg(...) { cerr << __LINE__ << ": [" << #__VA_ARGS__ << "] = "; pr(__VA_ARGS__); }
```

Algoritmos

Geometria

Ângulo entre segmentos

```
/**
 *  @param  P, Q, R, S  Points.
 *  @return             Smallest angle between segments PQ and RS in radians.
 *  Time complexity: O(1)
*/
template <typename T>
double angle(const pair<T, T>& P, const pair<T, T>& Q,
             const pair<T, T>& R, const pair<T, T>& S) {
    T ux = P.x - Q.x, uy = P.y - Q.y;
    T vx = R.x - S.x, vy = R.y - S.y;
    T num = ux * vx + uy * vy;
    double den = hypot(ux, uy) * hypot(vx, vy);
    assert(den != 0.0);  // degenerate segment
    return acos(num / den);
}
```

Distância entre pontos

```
/**
 *  @param  P, Q  Points.
 *  @return       Distance between points.
 *  Time complexity: O(1)
*/
template <typename T, typename S>
double dist(const pair<T, T>& P, const pair<S, S>& Q) {
    return hypot(P.x - Q.x, P.y - Q.y);
}
```

Envoltório convexo

```
template <typename T>
vector<pair<T, T>> makeHull(const vector<pair<T, T>>& PS) {
    vector<pair<T, T>> hull;
    for (auto& P : PS) {
        ll sz = hull.size();  //           if want collinear < 0
        while (sz >= 2 and D(hull[sz - 2], hull[sz - 1], P) <= 0) {
            hull.pop_back();
            sz = hull.size();
        }
        hull.eb(P);
    }
    return hull;
}

/**
 *  @param  PS  Vector of points.
 *  @return     Convex hull.
 *  Points will be sorted counter-clockwise.
 *  First and last point will be the same.
 *  Be aware of degenerate polygon (line) use D() to check.
 *  Time complexity: O(Nlog(N))
*/
template <typename T>
vector<pair<T, T>> monotoneChain(vector<pair<T, T>> PS) {
    vector<pair<T, T>> lower, upper;
    sort(all(PS));
    lower = makeHull(PS);
    reverse(all(PS));
    upper = makeHull(PS);
    lower.pop_back();
    lower.emplace(lower.end(), all(upper));
    return lower;
}
```

Orientação de ponto

```
/**
 *  @param  A, B, P  Points.
 *  @return          Value that represents orientation of P to segment AB.
 *  If orientation is collinear: zero;
 *  If point is to the left:     positive;
 *  If point is to the right:    negative;
 *  Time complexity: O(1)
*/
template <typename T>
T D(const pair<T, T>& A, const pair<T, T>& B, const pair<T, T>& P) {
    return (A.x * B.y + A.y * P.x + B.x * P.y) - (P.x * B.y + P.y * A.x + B.x * A.y);
}
```

```
/**
 *  @param  P, Q, O  Points.
 *  @return          True if P before Q in counter-clockwise order.
 *  O is the origin point.
 *  Time complexity: O(1)
*/
template <typename T>
bool ccw(pair<T, T> P, pair<T, T> Q, const pair<T, T>& O) {
    static const char qo[2][2] = { { 2, 3 }, { 1, 4 } };
    P.x -= O.x, P.y -= O.y, Q.x -= O.x, Q.y -= O.y, O.x = 0, O.y = 0;
    bool qqx = equals(P.x, 0) or P.x > 0, qqy = equals(P.y, 0) or P.y > 0;
    bool rqx = equals(Q.x, 0) or Q.x > 0, rqy = equals(Q.y, 0) or Q.y > 0;
    if (qqx != rqx || qqy != rqy) return qo[qqx][qqy] > qo[rqx][rqy];
    return equals(D(O, P, Q), 0) ?
           (P.x * P.x - P.y * P.y) < (Q.x * Q.x - Q.y * Q.y) : D(O, P, Q) > 0;
}
```

Mediatriz

```
/**
 *  @param  P, Q  Points.
 *  @return    Perpendicular bisector to segment PQ.
 *  Time complexity: O(1)
*/
template <typename T>
Line<T> perpendicularBisector(const pair<T, T>& P, const pair<T, T>& Q) {
    T a = 2 * (Q.x - P.x), b = 2 * (Q.y - P.y);
    T c = (P.x * P.x + P.y * P.y) - (Q.x * Q.x + Q.y * Q.y);
    return { a, b, c };
}
```

Rotação de ponto

```
/**
 *  @param  P  Point.
 *  @param  a  Angle in radians.
 *  @return    Rotated point.
 *  Time complexity: O(1)
*/
template <typename T>
pd rotate(const pair<T, T>& P, double a) {
    double x = cos(a) * P.x - sin(a) * P.y;
    double y = sin(a) * P.x + cos(a) * P.y;
    return { x, y };
}
```

Árvores

Binary lifting

```
constexpr ll LOG = 31;
vvll parent;
vll depth;

/**
 *  @param  g  Tree/Successor graph.
 *  @param  n  Amount of vertices.
 *  Time complexity: O(Nlog(N))
*/
void populate(const vvll& g) {
    ll n = g.size();
    parent = vvll(n, vll(LOG));
    depth  = vll(n);

    // populate parent
    auto dfs = [&](auto& self, ll u, ll p = 1) -> void {
        parent[u][0] = p;
        depth[u] = depth[p] + 1;
        for (ll v : g[u]) if (v != p)
            self(self, v, u);
    }; dfs(dfs, 1);  // if not tree needs to loop for every vertex

    rep(i, 1, LOG) rep(j, 0, n)
        parent[j][i] = parent[ parent[j][i - 1] ][i - 1];
}

/**
 *  @param  u  Vertex.
 *  @param  k  Number.
 *  @return    k-th ancestor of u.
 *  Requires populate().
 *  k = 0 is me, k = 1 my parent, and so on...
 *  Time complexity: O(log(N))
*/
ll kthAncestor(ll u, ll k) {
    assert(!parent.empty() and 1 <= u and u < parent.size() and k >= 0);
    if (k > depth[u]) return -1;  // no kth ancestor
    rep(i, 0, LOG) if (k & (1LL << i))
        u = parent[u][i];
    return u;
}
```

Centróide

```
vll subtree;

ll subtree_dfs(const vvll& g, ll u, ll p) {
    for (ll v : g[u]) if (v != p)
        subtree[u] += subtree_dfs(g, v, u);
    return subtree[u];
}

/**
 *  @param  g  Tree.
 *  @return    A new root that makes the size of all subtrees be n/2 or less.
 *  Time complexity: O(N)
*/
    ll n = g.size();
    if (p == 0) { subtree = vll(g.size(), 1); subtree_dfs(g, u, p); }
    for (ll v : g[u]) if (v != p and subtree[v] * 2 > g.size())
        return centroid(g, v, u);
    return u;
}
```

Centróide decomposition

```
vll parent, subtree;

ll subtree_dfs(const vvll& g, ll u, ll p) {
    subtree[u] = 1;
    for (ll v : g[u]) if (v != p and !parent[v])
        subtree[u] += subtree_dfs(g, v, u);
    return subtree[u];
}

/**
 *  @param  g  Tree.
 *  Forms a new tree of centroids with height log(N), size of each centroid subtree will
 *  also be kinda like log(N) because it keeps dividing by 2.
 *  Time complexity: O(Nlog(N))
*/
void centroidDecomp(const vvll& g, ll u = 1, ll p = 0, ll sz = 0) {
    if (p == 0) { p = -1; parent = subtree = vll(g.size()); }
    if (sz == 0) sz = subtree_dfs(g, u, 0);
    for (ll v : g[u]) if (!parent[v] and subtree[v] * 2 > sz)
        return subtree[u] = 0, centroidDecomp(g, v, p, sz);
    parent[u] = p;
    for (ll v : g[u]) if (!parent[v]) centroidDecomp(g, v, u);
}
```

Euler Tour

```
ll timer = 0;
vll st, et;

/**
 *  @param  g  Tree.
 *  Populates st and et, vectors that represents intervals of each subtree, with those
 *  we can use stuff like segtrees on the subtrees.
 *  Time complexity: O(N)
*/
void eulerTour(const vvll& g, ll u = 1, ll p = 0) {
    if (p == 0) { timer = 0; st = et = vll(g.size()); }
    st[u] = timer++;
    for (ll v : g[u]) if (v != p)
        eulerTour(g, v, u);
    et[u] = timer++;
}
```

Menor ancestral comum (LCA)

```
/**
 *  @param  u, v  Vertices.
 *  @return       Lowest common ancestor between u and v.
 *  Requires binary lifting pre-processing.
 *  Time complexity: O(log(N))
*/
ll lca(ll u, ll v) {
    assert(1 <= u and u < parent.size() and 1 <= v and v < parent.size());
    if (depth[u] < depth[v]) swap(u, v);
    ll k = depth[u] - depth[v];
    u = kthAncestor(u, k);
    if (u == v) return u;
    per(i, LOG - 1, 0) if (parent[u][i] != parent[v][i])
        u = parent[u][i], v = parent[v][i];
    return parent[u][0];  // could also be parent[v][0]
}
```

Grafos

Bellman-Ford

```
/**
 *  @param  g  Graph (w, v).
 *  @param  s  Starting vertex.
 *  @return    Vectors with smallest distances from every vertex to s and the paths.
 *  Weights can be negative.
 *  Can detect negative cycles.
 *  Time complexity: O(EV)
*/
pair<vll, vll> spfa(const vvpll& g, ll s) {
    ll n = g.size();
    vll ds(n, LLONG_MAX), cnt(n), pre = cnt;
    vll in_queue(n);
    queue<ll> q;
    ds[s] = 0; q.emplace(s);
    in_queue[s] = true;
    while (!q.empty()) {
        ll u = q.front(); q.pop();
        in_queue[u] = false;
        for (auto [w, v] : g[u]) {
            if (ds[u] == LLONG_MIN) {
                if (ds[v] != LLONG_MIN)
                    q.emplace(v);
                ds[v] = LLONG_MIN;
            }
            else if (ds[u] + w < ds[v]) {
                ds[v] = ds[u] + w;
                ++cnt[v], pre[v] = u;
                if (cnt[v] == n) {
                    ds[v] = LLONG_MIN;
                    ds[0] = v;  // ds[0] keeps one vertex that has -inf dist
                }
                if (!in_queue[v]) {
                    q.emplace(v);
                    in_queue[v] = true;
                }
            }
        }
    }
    return { ds, pre };
}
```

BFS 0/1

```
/**
 *  @param  g  Graph (w, v).
 *  @param  s  Starting vertex.
 *  @return    Vector with smallest distances from every vertex to s.
 *  The graph can only have weights 0 and 1.
 *  Time complexity: O(N)
*/
vll bfs01(const vvpll& g, ll s) {
    vll ds(g.size(), LLONG_MAX);
    deque<ll> dq;
    dq.eb(s); ds[s] = 0;
    while (!dq.empty()) {
        ll u = dq.front(); dq.pop_front();
        for (auto [w, v] : g[u])
            if (ds[u] + w < ds[v]) {
                ds[v] = ds[u] + w;
                if (w == 1) dq.eb(v);
                else dq.emplace_front(v);
            }
    }
    return ds;
}
```

Caminho euleriano

```
/**
 *  @param  g     Graph.
 *  @param  d     Directed flag (true if g is directed).
 *  @param  s, e  Start and end vertex.
 *  @return       Vector with the eulerian path. If e is specified: eulerian cycle.
 *  Empty if impossible or no edges.
 *  Eulerian path goes through every edge once, cycle starts and ends at the same node.
 *  Time complexity: O(EVlog(EV))
*/
vll eulerianPath(const vvll& g, bool d, ll s, ll e = -1) {
    ll n = g.size();
    vector<multiset<ll>> h(n);
    vll res, in_degree(n);
    stack<ll> st;
    st.emplace(s);  // start vertex

    rep(u, 0, n) for (auto v : g[u]) {
        ++in_degree[v];
        h[u].emplace(v);
    }

    ll check = (in_degree[s] - (ll)h[s].size()) * (in_degree[e] - (ll)h[e].size());
    if (e != -1 and check != -1) return {};  // impossible

    rep(u, 0, n) {
        if (e != -1 and (u == s or u == e)) continue;
        if (in_degree[u] != h[u].size() or (!d and in_degree[u] & 1))
            return {};  // impossible
    }

    while (!st.empty()) {
        ll u = st.top();
        if (h[u].empty()) { res.eb(u); st.pop(); }
        else {
            ll v = *h[u].begin();
            h[u].erase(h[u].find(v));
            --in_degree[v];
            if (!d) {
                h[v].erase(h[v].find(u));
                --in_degree[u];
            }
            st.emplace(v);
        }
    }

    rep(u, 0, n) if (in_degree[u] != 0) return {};  // impossible
    reverse(all(res));
```

```
    return res;
}
```

Dijkstra

```
/**
 *  @param  g  Graph (w, v).
 *  @param  s  Starting vertex.
 *  @return    Vectors with smallest distances from every vertex to s and the paths.
 *  If want to calculate amount of paths or size of path, notice that when the
 *  distance for a vertex is calculated it probably won't be the best, remember to reset
 *  calculations if a better is found.
 *  Time complexity: O(Elog(V))
*/
pair<vll, vll> dijkstra(const vvpll& g, ll s) {
    vll ds(g.size(), LLONG_MAX), pre(g.size(), -1);
    priority_queue<pll, vpll, greater<>> pq;
    ds[s] = 0; pq.emplace(ds[s], s);
    while (!pq.empty()) {
        auto [t, u] = pq.top(); pq.pop();
        if (t > ds[u]) continue;
        for (auto [w, v] : g[u])
            if (t + w < ds[v]) {
                ds[v] = t + w, pre[v] = u;
                pq.emplace(ds[v], v);
            }
    }
    return { ds, pre };
}

vll getPath(const vll& pre, ll s, ll u) {
    vll p { u };
    do {
        p.eb(pre[u]);
        u = pre[u];
        if (u == 0) return {};
    } while (u != s);
    reverse(all(p));
    return p;
}
```

Floyd Warshall

```
/**
 *  @param  g  Graph (w, v).
 *  @return    Vector with smallest distances between every vertex.
 *  Weights can be negative.
 *  If ds[u][v] == INT_MAX, unreachable
 *  If ds[u][v] == INT_MIN, negative cycle.
 *  Time complexity: O(V^3)
*/
vvll floydWarshall(const vvpll& g) {
    ll n = g.size();
    vvll ds(n, vll(n, INT_MAX));
    rep(u, 0, n) {
        ds[u][u] = 0;
        for (auto [w, v] : g[u]) {
            ds[u][v] = min(ds[u][v], w);
            if (ds[u][u] < 0) ds[u][u] = INT_MIN;  // negative cycle
        }
    }
    rep(k, 0, n) rep(u, 0, n) rep(v, 0, n)
        if (ds[u][k] != INT_MAX and ds[k][v] != INT_MAX) {
            ds[u][v] = min(ds[u][v], ds[u][k] + ds[k][v]);
            if (ds[k][k] < 0) ds[u][v] = INT_MIN;  // negative cycle
        }
    return ds;
}
```

Kosaraju

```
/**
 *  @param  g  Directed graph.
 *  @return    Condensed graph, scc and comp vector.
 *  Condensed graph is a DAG with the scc.
 *  A single vertex is a scc.
 *  The scc is ordered in the sense that if we have {a, b}, then there is a edge from
 *  a to b.
 *  scc is [comp, cc].
 *  comp[u] is the component "leader" of an original vertex.
 *  Time complexity: O(Elog(V))
*/
tuple<vvll, map<ll, vll>, vll> kosaraju(const vvll& g) {
    ll n = g.size();
    vvll inv(n), cond(n);
    map<ll, vll> scc;
    vll vs(n), comp(n), order;
    auto dfs = [&vs](auto& self, const vvll& h, vll& out, ll u) -> void {
        vs[u] = true;
        for (ll v : h[u]) if (!vs[v])
            self(self, h, out, v);
        out.eb(u);
    };
    rep(u, 0, n) {
        for (ll v : g[u]) inv[v].eb(u);
        if (!vs[u])       dfs(dfs, g, order, u);
    }
    vs = vll(n, false);
    reverse(all(order));
    for (ll u : order) if (!vs[u]) {
        vll cc;
        dfs(dfs, inv, cc, u);
        scc[u] = cc;
        for (ll v : cc) comp[v] = u;
    }
    rep(u, 0, n) for (ll v : g[u]) if (comp[u] != comp[v])
        cond[comp[u]].eb(comp[v]);
    return { cond, scc, comp };
}
```

Kruskal

```
/**
 *  @brief         Get min/max spanning tree.
 *  @param  edges  Vector of edges (w, u, v).
 *  @param  n      Amount of vertex.
 *  @return        Edges of mst, or forest if not connected.
 *  Time complexity: O(Nlog(N))
*/
vtll kruskal(vtll& edges, ll n) {
    DSU dsu(n);
    vtll mst;
    ll edges_sum = 0;
    sort(all(edges));  // change order if want maximum
    for (auto [w, u, v] : edges) if (!dsu.sameSet(u, v)) {
        dsu.mergeSetsOf(u, v);
        mst.eb(w, u, v);
        edges_sum += w;
    }
    return mst;
}
```

Ordenação topológica

```cpp
/**
 *  @param  g  Directed graph.
 *  @return    Vector with vertices in topological order or empty if has cycle.
 *  It starts from a vertex with indegree 0, that is no one points to it.
 *  Time complexity: O(EVlog(V))
*/
vll topoSort(const vvll& g) {
    ll n = g.size();
    vll degree(n), res;
    rep(u, 1, n) for (ll v : g[u])
        ++degree[v];

    // lower values bigger priorities
    priority_queue<ll, vll, greater<>> pq;
    rep(u, 1, degree.size())
        if (degree[u] == 0)
            pq.emplace(u);

    while (!pq.empty()) {
        ll u = pq.top();
        pq.pop();
        res.eb(u);
        for (ll v : g[u])
            if (--degree[v] == 0)
                pq.emplace(v);
    }

    if (res.size() != n - 1) return {};  // cycle
    return res;
}
```

Max flow/min cut (Dinic)

```cpp
/**
 *  @param  g  Graph (w, v).
 *  @param  s  Source.
 *  @param  t  Sink.
 *  @return    Max flow/min cut and graph with residuals.
 *  If want the cut edges do a dfs, after, for every visited vertex if it has edge to v
 *  but this is not visited then it was a cut.
 *  If want all the paths from source to sink, make a bfs, only traverse if there is
 *  a path from u to v and w is 0.
 *  When getting the path set each w in the path to 1.
 *  Capacities on edges, to limit a vertex create a new vertex and limit edge.
 *  Time complexity: O(EV^2) but there is cases where it's better (unit capacities).
*/
pair<ll, vector<vtll>> maxFlow(const vvpll& g, ll s, ll t) {
    ll n = g.size();
    vector<vtll> h(n);  // (w, v, rev)
    vll lvl(n), ptr(n), q(n);
    rep(u, 0, n) for (auto [w, v] : g[u]) {
        h[u].eb(w, v, h[v].size());
        h[v].eb(0, u, h[u].size() - 1);
    }
    auto dfs = [&](auto& self, ll u, ll nf) -> ll {
        if (u == t or nf == 0) return nf;
        for (ll& i = ptr[u]; i < h[u].size(); i++) {
            auto& [w, v, rev] = h[u][i];
            if (lvl[v] == lvl[u] + 1)
                if (ll p = self(self, v, min(nf, w))) {
                    auto& [wv, _, __] = h[v][rev];
                    w -= p, wv += p;
                    return p;
                }
        }
        return 0;
    };
    ll f = 0; q[0] = s;
    rep(l, 0, 31)
        do {
            lvl = ptr = vll(n);
            ll qi = 0, qe = lvl[s] = 1;
            while (qi < qe and !lvl[t]) {
                ll u = q[qi++];
                for (auto [w, v, rev] : h[u])
                    if (!lvl[v] and w >> (30 - l))
                        q[qe++] = v, lvl[v] = lvl[u] + 1;
            }
            while (ll nf = dfs(dfs, s, LLONG_MAX)) f += nf;
        } while (lvl[t]);
```

```
    return { f, h };
}
```

Pontes e articulações

```
/**
 *  @param  g  Graph [id of edge, v].
 *  Bridges are edges that when removed increases components.
 *  Articulations are vertices that when removed increases components.
 *  Time complexity: O(E + V)
*/
vll bridgesOrArticulations(const vvpll& g, bool get_bridges) {
    ll n = g.size(), timer = 0;
    vector<bool> vs(n);
    vll st(n), low(n), res;
    auto dfs = [&](auto& self, ll u, ll p) -> void {
        vs[u] = true;
        st[u] = low[u] = timer++;
        ll children = 0;
        bool parent_skipped = false;
        for (auto [i, v] : g[u]) {
            if (v == p && !parent_skipped) {
                parent_skipped = true;
                continue;
            }
            if (vs[v]) low[u] = min(low[u], st[v]);
            else {
                self(self, v, u);
                low[u] = min(low[u], low[v]);
                if (get_bridges and low[v] > st[u]) res.eb(i);
                else if (!get_bridges and p != 0 and low[v] >= st[u]) res.eb(u);
                ++children;
            }
        }
        if (!get_bridges and p == 0 and children > 1) res.eb(u);
    };
    rep(i, 0, g.size()) if (!vs[i]) dfs(dfs, i, 0);
    if (!get_bridges) {
        sort(all(res));
        res.erase(unique(all(res)), res.end());
    }
    return res;
}
```

Outros

Busca ternária

```
/**
 *  @param  lo, hi  Interval.
 *  @param  f       Function (strictly increases, reaches maximum, strictly decreases).
 *  @return         Maximum value of the function in interval [lo, hi].
 *  If it's an integer function use binary search.
 *  Time complexity: O(log(N))
*/
double ternarySearch(double lo, double hi, function<double(double)> f) {
    rep(i, 0, 100) {
        double mi1 = lo + (hi - lo) / 3.0, mi2 = hi - (hi - lo) / 3.0;
        if (f(mi1) < f(mi2)) lo = mi1;
        else                 hi = mi2;
    }
    return f(lo);
}
```

Intervalos com soma S

```
/**
 *  @param  xs   Vector.
 *  @param  sum  Desired sum.
 *  @return      Amount of contiguous intervals with sum S.
 *  Can change to count odd/even sum intervals (hist of even and odd).
 *  Also could change to get contiguos intervals with sum less equal, using an
 *  ordered-set just uncomment and comment the ones with hist.
 *  If want the interval to have an index or value subtract the parts without it.
 *  Time complexity: O(Nlog(N))
*/
template <typename T>
ll countIntervals(const vector<T>& xs, T sum) {
    map<T, ll> hist;
    hist[0] = 1;
    // oset<ll> csums;
    // csums.insert(0);
    ll ans = 0;
    T csum = 0;
    for (T x : xs) {
        csum += x;
        ans += hist[csum - sum];
        ++hist[csum];
        // ans += csums.size() - csums.order_of_key(csum - sum);
        // csums.insert(csum);
    }
    return ans;
}
```

Kadane

```cpp
/**
 *  @param  xs  Vector.
 *  @param  mx  Maximum Flag (true if want max).
 *  @return     Max/min contiguous sum and smallest interval inclusive.
 *  We consider valid an empty sum.
 *  Time complexity: O(N)
*/
template <typename T>
tuple<T, ll, ll> kadane(const vector<T>& xs, bool mx = true) {
    T res = 0, csum = 0;
    ll l = -1, r = -1, j = 0;
    rep(i, 0, xs.size()) {
        csum += xs[i] * (mx ? 1 : -1);
        if (csum < 0) csum = 0, j = i + 1;  //             > if wants biggest interval
        else if (csum > res or (csum == res and i - j + 1 < r - l + 1))
            res = csum, l = j, r = i;
    }
    return { res * (mx ? 1 : -1), l, r };
}
```

Listar combinações

```cpp
/**
 *  @brief      Lists all combinations n choose k.
 *  @param  k   Number.
 *  @param  xs  Target vector (of size n with the elements you want).
 *  When calling try to call on min(k, n - k) if
 *  can make the reverse logic to guarantee efficiency.
 *  Time complexity: O(K(binom(N, K)))
*/
void binom(ll k, const vll& xs) {
    vll ks;
    auto f = [&](auto& self, ll i, ll rem) {
        if (rem == 0) {  // do stuff here
            cout << ks << '\n';
            return;
        }
        if (i == xs.size()) return;
        ks.eb(xs[i]);
        self(self, i + 1, rem - 1);
        ks.pop_back();
        self(self, i + 1, rem);
    }; f(f, 0, k);
}
```

Maior subsequência comum (LCS)

```cpp
/**
 *  @param  xs, ys  Vectors/Strings.
 *  @return         One valid longest common subsequence.
 *  Time complexity: O(NM)
*/
template <typename T>
T lcs(const T& xs, const T& ys) {
    ll n = xs.size(), m = ys.size();
    vvll dp(n + 1, vll(m + 1));
    vvpll pre(n + 1, vpll(m + 1, { -1, -1 }));
    rep(i, 1, n + 1) rep(j, 1, m + 1)
        if (xs[i - 1] == ys[j - 1])
            dp[i][j] = 1 + dp[i - 1][j - 1], pre[i][j] = { i - 1, j - 1};
        else {
            if (dp[i][j - 1] >= dp[i][j])
                dp[i][j] = dp[i][j - 1], pre[i][j] = pre[i][j - 1];
            if (dp[i - 1][j] >= dp[i][j])
                dp[i][j] = dp[i - 1][j], pre[i][j] = pre[i - 1][j];
        }
    T res;
    while (pre[n][m].first != -1) {
        tie(n, m) = pre[n][m];
        res.eb(xs[n]);  // += if T is string.
    }
    reverse(all(res));
    return res;  // dp[n][m] is size of lcs.
}
```

Maior subsequência crescente (LIS)

```
/**
 *  @param  xs      Vector.
 *  @param  values  True if want values, indexes otherwise.
 *  @return         Longest increasing subsequence as values or indexes.
 *  Time complexity: O(Nlog(N))
*/
vll lis(const vll& xs, bool values) {
    assert(!xs.empty());
    vll ss, idx, pre(xs.size()), ys;
    rep(i, 0, xs.size()) {
        // change to upper_bound if want not decreasing
        ll j = lower_bound(all(ss), xs[i]) - ss.begin();
        if (j == ss.size()) idx.eb(), ss.eb();
        if (j == 0) pre[i] = -1;
        else        pre[i] = idx[j - 1];
        idx[j] = i, ss[j]  = xs[i];
    }
    ll i = idx.back();
    while (i != -1) {
        ys.eb((values ? xs[i] : i));
        i = pre[i];
    }
    reverse(all(ys));
    return ys;
}
```

Pares com gcd x

```
/**
 *  @param  xs  Target vector.
 *  @return     Vector with amount of pairs with gcd equals i [1, 1e6].
 *  Time complexity: O(Nlog(N))
*/
vll gcdPairs(const vll& xs) {
    ll MAXN = (ll)1e6 + 1;
    vll dp(MAXN, -1), ms(MAXN), hist(MAXN);
    for (ll x : xs) ++hist[x];
    rep(i, 1, MAXN)
        for (ll j = i; j < MAXN; j += i)
            ms[i] += hist[j];
    per(i, MAXN - 1, 1) {
        dp[i] = ms[i] * (ms[i] - 1) / 2;
        for (ll j = 2 * i; j < MAXN; j += i)
            dp[i] -= dp[j];
    }
    return dp;
}
```

Próximo maior/menor elemento

```
/**
 *  @param  xs  Vector.
 *  @return     Vector of indexes of closest smaller.
 *  Example: c[i] = j where j < i and xs[j] < xs[i] and it's the closest.
 *  If there isn't then c[i] = -1.
 *  Time complexity: O(N)
*/
template <typename T>
vector<T> closests(const vector<T>& xs) {
    vll c(xs.size(), -1);  // n if to the right
    stack<ll> prevs;
    // n - 1 -> 0: to the right
    rep(i, 0, xs.size()) {  //                    <= if want bigger
        while (!prevs.empty() and xs[prevs.top()] >= xs[i])
            prevs.pop();
        if (!prevs.empty()) c[i] = prevs.top();
        prevs.emplace(i);
    }
    return c;
}
```

Soma de todos os intervalos

```
/**
 *  @param  xs  Vector.
 *  @return     Sum of all intervals.
 *  By counting in how many intervals the element appear.
 *  Time complexity: O(N)
*/
template <typename T>
T sumAllIntervals(const vector<T>& xs) {
    T sum = 0;
    ll opens = 0;
    rep(i, 0, xs.size()) {
        opens += xs.size() - 2 * i;
        sum += xs[i] * opens;
    }
    return sum;
}
```

Matemática

Coeficiente binomial

```
/**
 *  @return  Binomial coefficient.
 *  Time complexity: O(N^2)/O(1)
*/
ll binom(ll n, ll k) {
    constexpr ll MAXN = 64;
    static vvll dp(MAXN + 1, vll(MAXN + 1));
    if (dp[0][0] != 1) {
        dp[0][0] = 1;
        rep(i, 1, MAXN + 1) rep(j, 0, i + 1)
            dp[i][j] = dp[i - 1][j] + (j ? (dp[i - 1][j - 1]) : 0);
    }
    if (n < k or n * k < 0) return 0;
    return dp[n][k];
}
```

Coeficiente binomial mod

```
/**
 *  @return  Binomial coefficient mod M.
 *  Time complexity: O(N)/O(1)
*/
ll binom(ll n, ll k) {
    constexpr ll MAXN = (ll)3e6, M = (ll)1e9 + 7;  // check mod value!
    static vll fac(MAXN + 1), inv(MAXN + 1), finv(MAXN + 1);
    if (fac[0] != 1) {
        fac[0] = fac[1] = inv[1] = finv[0] = finv[1] = 1;
        rep(i, 2, MAXN + 1) {
            fac[i] = fac[i - 1] * i % M;
            inv[i] = M - M / i * inv[M % i] % M;
            finv[i] = finv[i - 1] * inv[i] % M;
        }
    }
    if (n < k or n * k < 0) return 0;
    return fac[n] * finv[k] % M * finv[n - k] % M;
}
```

Conversão de base

```
/**
 *  @param  x  Number in base 10.
 *  @param  b  Base.
 *  @return    Vector with coefficients of x in base b.
 *  Example: (x = 6, b = 2): { 1, 1, 0 }
 *  Time complexity: O(log(N))
*/
vll toBase(ll x, ll b) {
    assert(b != 0);
    vll res;
    while (x) { res.eb(x % b); x /= b; }
    reverse(all(res));
    return res;
}
```

Crivo de Eratóstenes

```
/**
 *  @return  Vectors with primes from [1, n] and smallest prime factors.
 *  Time complexity: O(Nlog(N))
*/
pair<vll, vll> sieve(ll n) {
    vll ps, spf(n + 1);
    rep(i, 2, n + 1) if (!spf[i]) {
        ps.eb(i);
        for (ll j = i; j <= n; j += i)
            if (!spf[j]) spf[j] = i;
    }
    return { ps, spf };
}
```

Divisores

```
/**
 *  @return  Unordered vector with all divisors of x.
 *  Time complexity: O(sqrt(N))
*/
vll divisors(ll x) {
    vll ds;
    for (ll i = 1; i * i <= x; ++i)
        if (x % i == 0) {
            ds.eb(i);
            if (i * i != x) ds.eb(x / i);
        }
    return ds;
}
```

Divisores de vários números

```
/**
 *  @param  xs  Target vector.
 *  @param  x   Number.
 *  @return     Vectors with divisors for every number in xs.
 *  Time complexity: O(Nlog(N))
*/
vvll divisors(const vll& xs) {
    ll MAXN = (ll)1e6, mx = 0;
    vector<bool> hist(MAXN);
    for (ll y : xs) {
        mx = max(mx, y);
        hist[y] = true;
    }

    vvll ds(mx + 1);
    rep(i, 1, mx + 1)
        for (ll j = i; j <= mx; j += i)
            if (hist[j]) ds[j].eb(i);
    return ds;
}
```

Exponenciação binária

```
ll mul(ll a, ll b, ll p) { return (__int128)a * b % p; }

/**
 *  @param  a  Number.
 *  @param  b  Exponent.
 *  @param  p  Modulo.
 *  @return    a^b (mod p).
 *  Time complexity: O(log(B))
*/
ll exp(ll a, ll b, ll p) {
    ll res = 1;
    a %= p;
    while (b) {
        if (b & 1) res = mul(res, a, p);
        a = mul(a, a, p);
        b /= 2;
    }
    return res;
}
```

Fatoração

```
/**
 *  @return  Vector with prime factors of x.
 *  Time complexity: O(sqrt(N))
*/
vll factors(ll x) {
    vll fs;
    for (ll i = 2; i * i <= x; ++i)
        while (x % i == 0) {
            fs.eb(i);
            x /= i;
        }
    if (x > 1) fs.eb(x);
    return fs;
}
```

Fatoração com crivo

```
/**
 *  @param  x    Number.
 *  @param  spf  Vector of smallest prime factors
 *  @return      Vector with prime factors of x.
 *  Requires sieve.
 *  Time complexity: O(log(N))
*/
vll factors(ll x, const vll& spf) {
    vll fs;
    while (x != 1) { fs.eb(spf[x]); x /= spf[x]; }
    return fs;
}
```

Fatoração rápida

```
ll rho(ll x) {
    auto f  = [x](ll x) { return mul(x, x, x) + 1; };
    ll init = 0, x = 0, y = 0, prod = 2, i = 0;
    while (i & 63 or gcd(prod, x) == 1) {
        if (x == y) x = ++init, y = f(x);
        if (ll t = mul(prod, (x - y), x); t) prod = t;
        x = f(x), y = f(f(y)), ++i;
    }
    return gcd(prod, x);
}

/**
 *  @param  x  Number.
 *  @return    True if x is prime, false otherwise.
 *  Requires primality test, which requires binary exponentiation.
 *  Time complexity: O(N^(1/4)log(N))
*/
vll factors(ll x) {
    if (x == 1)     return {};
    if (isPrime(x)) return {x};
    ll d  = rho(x);
    vll l = factors(d), r = factors(x / d);
    l.insert(l.end(), all(r));
    return l;
}
```

Permutação com repetição

```
/**
 *  @param  hist  Histogram.
 *  @return       Permutation with repetition mod M.
 *  If it's only two elements and no mod use binom(n, k).
 *  Time complexity: O(N)
*/
template <typename T>
ll rePerm(const map<T, ll>& hist) {
    constexpr ll MAXN = (ll)3e6, M = (ll)1e9 + 7;  // check mod value!
    static vll fac(MAXN + 1), inv(MAXN + 1), finv(MAXN + 1);
    if (fac[0] != 1) {
        fac[0] = fac[1] = inv[1] = finv[0] = finv[1] = 1;
        rep(i, 2, MAXN + 1) {
            fac[i] = fac[i - 1] * i % M;
            inv[i] = M - M / i * inv[M % i] % M;
            finv[i] = finv[i - 1] * inv[i] % M;
        }
    }
    if (hist.empty()) return 0;
    ll res = 1, total = 0;
    for (auto [k, v] : hist) {
        res = res * finv[v] % M;
        total += v;
    }
    return res * fac[total] % M;
}
```

Teste de primalidade

```
/**
 *  @param  x  Number.
 *  @return    True if x is prime, false otherwise.
 *  Requires binary exponentiation.
 *  Time complexity: O(log^2(N))
*/
bool isPrime(ll x) {  // miller rabin
    if (x < 2)      return false;
    if (x <= 3)     return true;
    if (x % 2 == 0) return false;
    ll r = __builtin_ctzll(x - 1), d = x >> r;
    for (ll a : {2, 3, 5, 7, 11, 13, 17, 19, 23}) {
        if (a == x) return true;
        a = exp(a, d, x);
        if (a == 1 or a == x - 1) continue;
        rep(i, 1, r) {
            a = mul(a, a, x);
            if (a == x - 1) break;
        }
        if (a != x - 1) return false;
    }
    return true;
}
```

Totiente de Euler

```
/**
 *  @return  Vector with Euler totient value for every number in [1, n].
 *  Euler totient counts coprimes of x in [1, x].
 *  Time complexity: O(Nlog(log(N)))
*/
vll totient(ll n) {
    vll phi(n + 1);
    iota(all(phi), 0);
    rep(i, 2, n + 1)
        if (phi[i] == i)
            for (ll j = i; j <= n; j += i)
                phi[j] -= phi[j] / i;
    return phi;
}
```

Transformada de Fourier

```
constexpr ll mod     = 998244353;
constexpr ll root    = 15311432;
constexpr ll rootinv = 469870224;
constexpr ll root_pw = 1 << 23;
#define T Mi<mod>

/**
 *  @brief     Fast fourier transform with integers mod.
 *  @param  a  Coefficients of polynomial.
 *  Requires modular arithmetic.
 *  Time complexity: O(Nlog(N))
*/
void ntt(vector<T>& a, bool invert) {
    ll n = a.size();
    for (ll i = 1, j = 0; i < n; i++) {
        ll bit = n >> 1;
        while (j & bit) j ^= bit, bit >>= 1;
        j ^= bit;
        if (i < j) swap(a[i], a[j]);
    }
    for (ll len = 2; len <= n; len <<= 1) {
        T wlen = invert ? rootinv : root;
        for (ll i = len; i < root_pw; i <<= 1)
            wlen *= wlen;
        for (ll i = 0; i < n; i += len) {
            T w = 1;
            for (ll j = 0; j < len / 2; j++, w *= wlen) {
                T u = a[i + j], v = a[i + j + len / 2] * w;
                a[i + j] = u + v, a[i + j + len / 2] = u - v;
            }
        }
    }
    if (invert) {
        T ninv = T(1) / n;
        for (T& x : a) x *= ninv;
    }
}

/**
 *  @param  a, b  Coefficients of both polynomials
 *  @return       Coefficients of the multiplication of both polynomials.
 *  Requires modular arithmetic.
 *  Time complexity: O(Nlog(N))
*/
vector<T> convolution(const vector<T>& a, const vector<T>& b) {
    vector<T> fa(all(a)), fb(all(b));
    ll n = 1;
    while (n < a.size() + b.size()) n <<= 1;
    fa.resize(n), fb.resize(n);
```

```
    ntt(fa, false), ntt(fb, false);
    rep(i, 0, n) fa[i] *= fb[i];
    ntt(fa, true);
    return fa;
}

void print(const vector<T>& a) {
    bool first = true;
    per(i, a.size() - 1, 0) if (a[i].v) {
        cout << (first ? "" : " + ") << a[i] << "x^" << i;
        first = false;
    }
    cout << '\n';
}
```

Strings

Borda de prefixos (KMP)

```
/**
 *  @param  s  String.
 *  @return    Vector with the border size for each prefix size.
 *  Used in KMP to count occurrences.
 *  Time complexity: O(N)
*/
vll prefixBorder(const string& s) {
    vll bs(s.size() + 1, -1);
    ll t = -1;
    rep(i, 0, s.size()) {
        while (t > -1 and s[t] != s[i])
            t = bs[t];
        bs[i + 1] = ++t;
    }
    return bs;
}
```

Comparador de substring

```
/**
 *  @param  i, j  First and second substring start indexes.
 *  @param  m     Size of both substrings.
 *  @param  cs    Equivalence classes from suffix array.
 *  @return       0 if equal, -1 if smaller or 1 if bigger.
 *  Requires suffix array.
 *  Time complexity: O(1)
*/
ll compare(ll i, ll j, ll m, const vector<vector<int>>& cs) {
    ll k = 0;  // move outside
    while ((1 << (k + 1)) <= m) ++k;  // move outside
    pll a = { cs[k][i], cs[k][i + m - (1 << k)] };
    pll b = { cs[k][j], cs[k][j + m - (1 << k)] };
    return a == b ? 0 : (a < b ? -1 : 1);
}
```

Distância de edição

```
/**
 *  @param  s, t  Srings
 *  @return       Edit distance to transform s in t and operations.
 *  Can change costs.
 *  -      Deletion
 *  c      Insertion of c
 *  =      Keep
 *  [c->d] Substitute c to d.
 *  Time complexity: O(MN)
*/
pair<ll, string> edit(const string& s,  string& t) {
    ll ci = 1, cr = 1, cs = 1, m = s.size(), n = t.size();
    vvll dp(m + 1, vll(n + 1)), pre = dp;

    rep(i, 0, m + 1) dp[i][0] = i*cr, pre[i][0] = 'r';
    rep(j, 0, n + 1) dp[0][j] = j*ci, pre[0][j] = 'i';
    rep(i, 1, m + 1)
        rep(j, 1, n + 1) {
            ll ins = dp[i][j - 1] + ci, del = dp[i - 1][j] + cr;
            ll subs = dp[i - 1][j - 1] + cs * (s[i - 1] != t[j - 1]);
            dp[i][j] = min({ ins, del, subs });
            pre[i][j] = (dp[i][j] == ins ? 'i' : (dp[i][j] == del ? 'r' : 's'));
        }

    ll i = m, j = n;
    string ops;

    while (i or j) {
        if (pre[i][j] == 'i')
            ops += t[--j];
        else if (pre[i][j] == 'r') {
            ops += '-';
            --i;
        }
        else {
            --i, --j;
            if (s[i] == t[j]) ops += '=';
            else
                ops += "]", ops += t[j], ops += ">-", ops += s[i], ops += "[";
        }
    }

    reverse(all(ops));
    return { dp[m][n], ops };
}
```

Maior prefixo comum (LCP)

```
/**
 *  @param  s   String.
 *  @param  sa  Suffix array.
 *  @return     Vector with lcp.
 *  Requires suffix array.
 *  lcp[i]: largest common prefix between suffix sa[i]
 *  and sa[i + 1]. To get lcp(i, j), do min({lcp[i], ..., lcp[j - 1]}).
 *  That would be lcp between suffix sa[i] and suffix sa[j].
 *  Time complexity: O(N)
*/
vll getLcp(const string& s, const vll& sa) {
    ll n = s.size(), k = 0;
    vll rank(n), lcp(n - 1);
    rep(i, 0, n) rank[sa[i]] = i;
    rep(i, 0, n) {
        if (rank[i] == n - 1) {
            k = 0;
            continue;
        }
        ll j = sa[rank[i] + 1];
        while (i + k < n and j + k < n and s[i + k] == s[j + k])
            ++k;
        lcp[rank[i]] = k;
        if (k) --k;
    }
    return lcp;
}
```

Manacher (substrings palíndromas)

```
/**
 *  @param  s   String.
 *  @return     Vector of pairs (deven, dodd).
 *  deven[i] and dodd[i] represent the biggest palindrome centered at i,
 *  palindrome of size even and odd respectively, even palindromes centered
 *  at i means that it's centered at both i - 1 and i, because they are equal.
 *  Time complexity: O(N)
*/
vpll manacher(string s) {
    string t;
    for(char c : s) t += string("#") + c;
    t += '#';
    ll n = t.size(), l = 0, r = 1;
    t = "$" + t + "^";
    vll p(n + 2);  // qnt of palindromes centered in i.
    rep(i, 1, n + 1) {
        p[i] = max(0LL, min(r - i, p[l + (r - i)]));
        while(t[i - p[i]] == t[i + p[i]]) p[i]++;
        if(i + p[i] > r) l = i - p[i], r = i + p[i];
    }
    ll m = s.size(), i = 0;
    vpll res(m);
    for (auto& [deven, dodd] : res)
        deven = p[2 * i + 1] - 1, dodd = p[2 * i + 2] - 1, ++i;
    return res;
}
```

Menor rotação

```
/**
 *  @param  s  String.
 *  @return    Index of the minimum rotation.
 *  Time complexity: O(N)
*/
ll minRotation(const string& s) {
    ll n = s.size(), k = 0;
    vll f(2 * n, -1);
    rep(j, 1, 2 * n) {
        ll i = f[j - k - 1];
        while (i != -1 and s[j % n] != s[(k + i + 1) % n]) {
            if (s[j % n] < s[(k + i + 1) % n])
                k = j - i - 1;
            i = f[i];
        }
        if (i == -1 and s[j % n] != s[(k + i + 1) % n]) {
            if (s[j % n] < s[(k + i + 1) % n])
                k = j;
            f[j - k] = -1;
        }
        else
            f[j - k] = i + 1;
    }
    return k;
}
```

Ocorrências de substring (suffix array)

```
/**
 *  @param  s   String.
 *  @param  t   Substring.
 *  @param  sa  Suffix array.
 *  @return     Amount of occurrences.
 *  Requires suffix array.
 *  Time complexity: O(Mlog(N))
*/
ll count(const string& s, const string& t, const vll& sa) {
    auto it1 = lower_bound(all(sa), t, [&](ll i, const string& r) {
        return s.compare(i, r.size(), r) < 0;
    });
    auto it2 = upper_bound(all(sa), t, [&](const string& r, ll i) {
        return s.compare(i, r.size(), r) > 0;
    });
    return it2 - it1;
}
```

Ocorrências de substring (Z-Function)

```
/**
 *  @param  s  String.
 *  @param  t  Substring.
 *  @return    Vector with the first index of occurrences.
 *  Requires Z-Function.
 *  Time complexity: O(N)
*/
vll occur(const string& s, const string& t) {
    vll zs = z(t + ';' + s), is;
    rep(i, 0, zs.size())
        if (zs[i] == t.size())
            is.eb(i - t.size() - 1);
    return is;
}
```

Palíndromo check

```
/**
 *  @param  i, j   Interval of substring.
 *  @param  h, rh  Hash of string and reverse string.
 *  @return        True if substring [i, j] is a palindrome.
 *  Requires hash.
 *  Time complexity: O(1)
*/
bool palindrome(ll i, ll j, Hash& h, Hash& rh) {
    return h(i, j) == rh(h.n - j - 1, h.n - i - 1);
}
```

Períodos

```
/**
 *  @param  s  String.
 *  @return    Vector with the periods.
 *  Requires Z-Function.
 *  Includes period of size n.
 *  Time complexity: O(N)
*/
vll periods(const string& s) {
    ll n = s.size();
    vll zs = z(s), ps;
    rep(i, 0, n) if (zs[i] == n - i)
        ps.eb(i);
    ps.eb(n);
    return ps;
}
```

Suffix array

```
template <typename T>
void cSort(const T& xs, vll& ps, ll alpha) {
    vll hist(alpha + 1);
    for (auto x : xs) ++hist[x];
    rep(i, 1, alpha + 1) hist[i] += hist[i - 1];
    per(i, ps.size() - 1, 0) ps[--hist[xs[i]]] = i;
}

template <typename T>
void updEqClass(vll& cs, const vll& ps, const T& xs) {
    cs[0] = 0;
    rep(i, 1, ps.size())
        cs[ps[i]] = cs[ps[i - 1]] + (xs[ps[i - 1]] != xs[ps[i]]);
}

/**
 *  @param  s  String.
 *  @param  k  log of M (M is size of substring to compare).
 *  @return    Suffix array or equivalence classes.
 *  Suffix array is a vector with the lexographically sorted suffix indexes.
 *  If want to use the compare() function that requires suffix array,
 *  pass k to this function to have the equivalence classes vector.
 *  Time complexity: O(Nlog(N))
*/
vll suffixArray(string s, ll k = LLONG_MAX) {
    s += ';';
    ll n = s.size();
    vll ps(n), rs(n), xs(n), cs(n);
    cSort(s, ps, 256);
    vpll ys(n);
    updEqClass(cs, ps, s);
    for (ll mask = 1; mask < n and k > 0; mask *= 2, --k) {
        rep(i, 0, n) {
            rs[i] = ps[i] - mask + (ps[i] < mask) * n;
            xs[i] = cs[rs[i]];
            ys[i] = {cs[i], cs[i + mask - (i + mask >= n) * n]};
        }
        cSort(xs, ps, cs[ps.back()] + 1);
        rep(i, 0, n) ps[i] = rs[ps[i]];
        updEqClass(cs, ps, ys);
    }
    ps.erase(ps.begin());
    return (k == 0 ? cs : ps);
}
```

Z-Function

```
/**
 *  @param  s  String.
 *  @return    Vector with Z-Function value for every position.
 *  It's how much original prefix this suffix has as prefix.
 *  Time complexity: O(N)
*/
vll z(const string& s) {
    ll n = s.size(), l = 0, r = 0;
    vll zs(n);
    rep(i, 1, n) {
        if (i <= r)
            zs[i] = min(zs[i - l], r - i + 1);
        while (zs[i] + i < n && s[zs[i]] == s[i + zs[i]])
            ++zs[i];
        if (r < i + zs[i] - 1)
            l = i, r = i + zs[i] - 1;
    }
    return zs;
}
```

Estruturas

Árvores

BIT tree 2D

```
template <typename T>
struct BIT2D {
    /**
     *  @param  h, w  Height and width.
    */
    BIT2D(ll h, ll w) : n(h), m(w), bit(n + 1, vector<T>(m + 1)) {}

    /**
     *  @brief        Adds v to position (y, x).
     *  @param  y, x  Position (1-Indexed).
     *  @param  v     Value to add.
     *  Time complexity: O(log(N))
    */
    void add(ll y, ll x, T v) {
        assert(0 < y and y <= n and 0 < x and x <= m)
        for (; y <= n; y += y & -y)
            for (ll i = x; i <= m; i += i & -i)
                bit[y][i] += v;
    }

    T sum(ll y, ll x) {
        T sum = 0;
        for (; y > 0; y -= y & -y)
            for (ll i = x; i > 0; i -= i & -i)
                sum += bit[y][i];
        return sum;
    }

    /**
     *  @param  ly, hy  Vertical   interval
     *  @param  lx, hx  Horizontal interval
     *  @return         Sum in that rectangle.
     *  1-indexed.
     *  Time complexity: O(log(N))
    */
    T sum(ll ly, ll lx, ll hy, ll hx) {
        assert(0 < ly and ly <= hy and hy <= n and 0 < lx and lx <= hx and hx <= m);
        return sum(hy, hx) - sum(hy, lx - 1) - sum(ly - 1, hx) + sum(ly - 1, lx - 1);
    }

    ll n, m;
    vector<vector<T>> bit;
};
```

Disjoint set union

```
struct DSU {
    /**
     *  @param  n  Size.
    */
    DSU(ll n) : parent(n), size(n, 1) { iota(all(parent), 0); }

    /**
     *  @param  x  Element.
     *  Time complexity: ~O(1)
    */
    ll setOf(ll x) {
        assert(0 <= x and x < parent.size());
        return parent[x] == x ? x : parent[x] = setOf(parent[x]);
    }

    /**
     *  @param  x, y  Elements.
     *  Time complexity: ~O(1)
    */
    void mergeSetsOf(ll x, ll y) {
        ll a = setOf(x), b = setOf(y);
        if (size[a] > size[b]) swap(a, b);
        parent[a] = b;
        if (a != b) size[b] += size[a], size[a] = 0;
    }

    /**
     *  @param  x, y  Elements.
     *  Time complexity: ~O(1)
    */
    bool sameSet(ll x, ll y) { return setOf(x) == setOf(y); }

    vll parent, size;
};
```

Heavy-light decomposition

```
template <typename T, typename Op = function<T(T, T)>>
struct HLD {
    /**
    *  @param  g    Tree.
    *  @param  def  Default value.
    *  @param  f    Merge function.
    *  Example: def in sum or gcd should be 0, in max LLONG_MIN, in min LLONG_MAX.
    *  Initialize with setQueryPath(u, u) if values on vertex or setQueryPath(u, v)
    *  if values on edges. The graph will need to be without weights even if there
    *  is on the edges.
    *  Time complexity: O(N)
    */
    HLD(vvll& g, bool values_on_edges, T def, Op f)
            : seg(g.size(), def, f), op(f) {
        idx = subtree = parent = head = vll(g.size());
        auto build = [&](auto& self, ll u = 1, ll p = 0) -> void {
            idx[u] = timer++, subtree[u] = 1, parent[u] = p;
            for (ll& v : g[u]) if (v != p) {
                head[v] = (v == g[u][0] ? head[u] : v);
                self(self, v, u);
                subtree[u] += subtree[v];
                if (subtree[v] > subtree[g[u][0]] or g[u][0] == p)
                    swap(v, g[u][0]);
            }

            if (p == 0) {
                timer = 0;
                self(self, head[u] = u, -1);
            }
        };
        build(build);
    }

    /**
    *  @param  u, v  Vertices.
    *  @param  x     Value to add     (if it's a set).
    *  @return       f of path [u, v] (if it's a query).
    *  It's a query if x is specified.
    *  Time complexity: O(log^2(N))
    */
    ll setQueryPath(ll u, ll v, ll x = INT_MIN) {
        assert(1 <= u and u < idx.size() and 1 <= v and v < idx.size());
        if (idx[u] < idx[v]) swap(u, v);
        if (head[u] == head[v]) return seg.setQuery(idx[v] + values_on_edges, idx[u], x);
        return op(seg.setQuery(idx[head[u]], idx[u], x),
                  setQueryPath(parent[head[u]], v, x));
    }

      /**
```

```
    *  @param  u  Vertex.
    *  @param  x  Value to add (if it's a set).
    *  @return    f of subtree (if it's a query).
    *  It's a query if x is specified.
    *  Time complexity: O(log(N))
    */
    ll setQuerySubtree(ll u, ll x = INT_MIN) {
        assert(1 <= u and u < idx.size());
        return seg.setQuery(idx[u] + values_on_edges, idx[u] + subtree[u] - 1, x);
    }

    Segtree<T> seg;
    vll idx, subtree, parent, head;
    ll timer = 0;
    Op op;
};
```

Ordered-set

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

/**
*  oset<int> = set, oset<int, int> = map.
*  Change less<> to less_equal<> to have a multiset/multimap. (.lower_bound() swaps
*  with .upper_bound(), .erase() will only work with an iterator, .find() breaks).
*  Other methods are the same as the ones in set/map with two new ones:
*  .find_by_order(i) and .order_of_key(T), the first gives the iterator to element in
*  index i and the second gives the index where element T would be inserted (if there
*  is one already, it will be the index of the first), could also interpret as
*  amount of smaller elements.
*/
template <typename T, typename S = null_type>
using oset = tree<T, S, less<>, rb_tree_tag, tree_order_statistics_node_update>;
```

Segment tree

```
template <typename T, typename Op = function<T(T, T)>>
struct Segtree {
    /**
    *  @param  sz   Size.
    *  @param  def  Default value.
    *  @param  f    Merge function.
    *  Example: def in sum or gcd should be 0, in max LLONG_MIN, in min LLONG_MAX
    */
    Segtree(ll sz, T def, Op f) : seg(4 * sz, def), lzy(4 * sz), n(sz), DEF(def), op(f) {}

    /**
    *  @param  xs  Vector.
    *  Time complexity: O(N)
    */
    void build(const vector<T>& xs, ll l = 0, ll r = -1, ll no = 1) {
        if (r == -1) r = n - 1;
        if (l == r) seg[no] = xs[l];
        else {
            ll m = (l + r) / 2;
            build(xs, l, m, 2 * no);
            build(xs, m + 1, r, 2 * no + 1);
            seg[no] = op(seg[2 * no], seg[2 * no + 1]);
        }
    }

    /**
    *  @param  i, j  Interval.
    *  @param  x     Value to add (if it's an update).
    *  @return       f of interval [i, j] (if it's a query).
    *  It's a query if x is specified.
    *  Time complexity: O(log(N))
    */
    T updQry(ll i, ll j, T x = LLONG_MIN, ll l = 0, ll r = -1, ll no = 1) {
        assert(0 <= i and i <= j and j < n);
        if (r == -1) r = n - 1;
        if (lzy[no]) unlazy(l, r, no);
        if (j < l or i > r) return DEF;
        if (i <= l and r <= j) {
            if (x != LLONG_MIN) {
                lzy[no] += x;  // seg[no] if no lazy
                unlazy(l, r, no);
            }
            return seg[no];
        }
        ll m = (l + r) / 2;
        T q = op(updQry(i, j, x, l, m, 2 * no),
                 updQry(i, j, x, m + 1, r, 2 * no + 1));  // [qry]
        seg[no] = op(seg[2 * no], seg[2 * no + 1]);  // [upd] comment if no lazy range upd
        return q;  // [qry] q + seg[no] if no lazy range upd
```

```
    }

private:
    void unlazy(ll l, ll r, ll no) {
        if (seg[no] == DEF) seg[no] = 0;
        seg[no] += (r - l + 1) * lzy[no];  // sum
        // seg[no] += lzy[no];  // min/max
        if (l < r) {
            lzy[2 * no] += lzy[no];
            lzy[2 * no + 1] += lzy[no];
        }
        lzy[no] = 0;
    }

    vector<T> seg, lzy;
    ll n;
    T DEF;
    Op op;
};
```

Máximos

```
// for segment tree
struct Node {
    static constexpr ll n = 2;  // quantity of maxs
    array<ll, n> xs;  // maxs
    Node() = default;
    Node(ll x) { xs.fill(0), xs[0] = x; }
    operator bool() { return xs[0]; }
    Node& operator+=(const Node& v) {
        xs[0] += v.xs[0];
        rep(i, 1, n) if (xs[i])
            xs[i] += v.xs[0];
        return *this;
    }
    bool operator!=(ll x) { return xs[0] != x; }
    static Node f(Node u, const Node& v) {
        vll ys(all(u.xs));
        ys.insert(ys.end(), all(v.xs));
        sort(all(ys), greater<>());
        ys.erase(unique(all(ys)), ys.end());
        rep(i, 0, n)
            u.xs[i] = ys[i];
        return u;
    }
};
```

Primeiro maior

```
/**
*  @param  i, j  Interval;
*  @param  x     Value to comppare.
*  @return       First index with element greater than x.
*  This is a segment tree's method.
*  The segment tree function must be max().
*  Returns -1 if no element is greater.
*  Time complexity: O(log(N))
*/
ll firstGreater(ll i, ll j, T x, ll l = 0, ll r = -1, ll no = 1) {
    assert(0 <= i and i <= j and j < n);
    if (r == -1) r = n - 1;
    if (j < l or i > r or seg[no] <= x) return -1;
    if (l == r) return l;
    ll m = (l + r) / 2;
    ll left = firstGreater(i, j, x, l, m, 2 * no);
    if (left != -1) return left;
    return firstGreater(i, j, x, m + 1, r, 2 * no + 1);
}
```

Treap

```
mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());

typedef char NT;
struct Node {
    Node(NT x) : v(x), s(x), w(rng()) {}
    NT v, s;
    ll w, sz = 1;
    bool lazy_rev = false;
    Node *l = nullptr, *r = nullptr;
};
typedef Node* NP;

ll size(NP t) { return t ? t->sz : 0; }
ll sum(NP t) { return t ? t->s : 0; }

void unlazy(NP t) {
    if (!t or !t->lazy_rev) return;
    t->lazy_rev = false;
    swap(t->l, t->r);
    if (t->l) t->l->lazy_rev ^= true;
    if (t->r) t->r->lazy_rev ^= true;
}

void lazy(NP t) {
    if (!t) return;
    unlazy(t->l), unlazy(t->r);
    t->sz = size(t->l) + size(t->r) + 1;
    t->s  = sum(t->l) + sum(t->r) + t->v;
}

NP merge(NP l, NP r) {
    NP t;
    unlazy(l), unlazy(r);
    if (!l or !r) t = l ? l : r;
    else if (l->w > r->w) l->r = merge(l->r, r), t = l;
    else r->l = merge(l, r->l), t = r;
    lazy(t);
    return t;
}

// splits t into l: [0, val), r: [val, )
void split(NP t, NP& l, NP& r, ll i) {
    unlazy(t);
    if (!t) l = r = nullptr;
    else if (i > size(t->l)) split(t->r, t->r, r, i - size(t->l) - 1), l = t;
    else split(t->l, l, t->l, i), r = t;
    lazy(t);
}

/**
*  @param  t  Node pointer.
*  Time complexity: O(N)
*/
void print(NP t) {
    unlazy(t);
    if (!t) return;
    print(t->l);
    cout << t->v;
    print(t->r);
}

struct Treap {
    NP root = nullptr;

    /**
    *  @brief     Inserts element at index i, pushes from index i inclusive.
    *  @param  i  Index.
    *  @param  x  Value to insert.
    *  Time complexity: O(log(N))
    */
    void insert(ll i, NT x) {
        NP l, r, no = new Node(x);
        split(root, l, r, i);
        root = merge(merge(l, no), r);
    }

    /**
    *  @brief     Erases element at index i, pulls from index i + 1 inclusive.
    *  @param  i  Index.
    *  Time complexity: O(log(N))
    */
    void erase(ll i) {
        NP l, r;
        split(root, l, r, i);
        split(r, root, r, 1);
        root = merge(l, r);
    }

    /**
    *  @brief updates the range [i, j)
    *  @param  i, j  Interval.
    *  @param  f     Function to apply.
    *  Time complexity: O(log(N))
    */
    void upd(ll i, ll j, function<void(NP)> f) {
        NP m, r;
        split(root, root, m, i);
        split(m, m, r, j - i + 1);
        if (m) f(m);
        root = merge(merge(root, m), r);
```

```
    }

    /**
    *  @brief query the range [i, j)
    *  @param  i, j  Interval.
    *  @param  f     Function to query.
    *  Time complexity: O(log(N))
    */
    template <typename R>
    R query(ll i, ll j, function<R(NP)> f) {
        NP m, r;
        split(root, root, m, i);
        split(m, m, r, j - i + 1);
        assert(m);
        R x = f(m);
        root = merge(merge(root, m), r);
        return x;
    }
};
```

Wavelet Tree

```
struct WaveletTree {
    /**
    *  @param  xs  Compressed vector.
    *  @param  sz  Distinct elements amount in xs (mp.size()).
    *  Sorts xs in the process.
    *  Time complexity: O(Nlog(N))
    */
    WaveletTree(vll& xs, ll sz) : wav(2 * n), n(sz) {
        auto build = [&](auto& self, auto b, auto e, ll l, ll r, ll no) {
            if (l == r) return;
            ll m = (l + r) / 2, i = 0;
            wav[no].resize(e - b + 1);
            for (auto it = b; it != e; ++it, ++i)
                wav[no][i + 1] = wav[no][i] + (*it <= m);
            auto p = stable_partition(b, e, [m](ll x) { return x <= m; });
            self(self, b, p, l, m, 2 * no);
            self(self, p, e, m + 1, r, 2 * no + 1);
        };
        build(build, all(xs), 0, n - 1, 1);
    }

    /**
    *  @param  i, j  Interval.
    *  @param  k     Number, starts from 1.
    *  @return       k-th smallest element in [i, j].
    *  Time complexity: O(log(N))
    */
    ll kTh(ll i, ll j, ll k) {
        assert(0 <= i and i <= j and j < (ll)wav[1].size() and k > 0);
        ++j;
        ll l = 0, r = n - 1, no = 1;
        while (l != r) {
            ll m = (l + r) / 2;
            ll leqm_l = wav[no][i], leqm_r = wav[no][j];
            no *= 2;
            if (k <= leqm_r - leqm_l)  i  = leqm_l, j  = leqm_r, r = m;
            else k -= leqm_r - leqm_l, i -= leqm_l, j -= leqm_r, l = m + 1, ++no;
        }
        return l;
    }

    /**
    *  @param  i, j  Interval.
    *  @param  x     Compressed value.
    *  @return       Occurrences of values less than or equal to x in [i, j].
    *  Time complexity: O(log(N))
    */
    ll leq(ll i, ll j, ll x) {
        assert(0 <= i and i <= j and j < (ll)wav[1].size() and 0 <= x and x < n);
```

```
        ++j;
        ll l = 0, r = n - 1, lx = 0, no = 1;
        while (l != r) {
            ll m = (l + r) / 2;
            ll leqm_l = wav[no][i], leqm_r = wav[no][j];
            no *= 2;
            if (x <= m) i = leqm_l, j = leqm_r, r = m;
            else i -= leqm_l, j -= leqm_r, l = m + 1, lx += leqm_r - leqm_l, ++no;
        }
        return j - i + lx;
    }

    vvll wav;
    ll n;
};
```

Geometria

Círculo

```
enum Position { IN, ON, OUT };

template <typename T>
struct Circle {
    /**
    *  @param  P  Origin point.
    *  @param  r  Radius length.
    */
    Circle(const pair<T, T>& P, T r) : C(P), r(r) {}

    /**
    *  Time complexity: O(1)
    */
    double area() { return acos(-1.0) * r * r; }
    double perimeter() { return 2.0 * acos(-1.0) * r; }
    double arc(double radians) { return radians * r; }
    double chord(double radians) { return 2.0 * r * sin(radians / 2.0); }
    double sector(double radians) { return (radians * r * r) / 2.0; }

    /**
    *  @param  a  Angle in radians.
    *  @return    Circle segment.
    *  Time complexity: O(1)
    */
    double segment(double a) {
        double c = chord(a);
        double s = (r + r + c) / 2.0;
        double t = sqrt(s) * sqrt(s - r) * sqrt(s - r) * sqrt(s - c);
        return sector(a) - t;
    }

    /**
    *  @param  P  Point.
    *  @return    Value that represents orientation of P to this circle.
    *  Time complexity: O(1)
    */
    Position position(const pair<T, T>& P) {
        double d = dist(P, C);
        return equals(d, r) ? ON : (d < r ? IN : OUT);
    }

    /**
    *  @param  c  Circle.
    *  @return    Intersection(s) point(s) between c and this circle.
    *  Time complexity: O(1)
    */
    vector<pair<T, T>> intersection(const Circle& c) {
```

```
    double d = dist(c.C, C);

    // no intersection or same
    if (d > c.r + r or d < abs(c.r - r) or (equals(d, 0) and equals(c.r, r)))
        return {};

    double a = (c.r * c.r - r * r + d * d) / (2.0 * d);
    double h = sqrt(c.r * c.r - a * a);
    double x = c.C.x + (a / d) * (C.x - c.C.x);
    double y = c.C.y + (a / d) * (C.y - c.C.y);
    pd p1, p2;
    p1.x = x + (h / d) * (C.y - c.C.y);
    p1.y = y - (h / d) * (C.x - c.C.x);
    p2.x = x - (h / d) * (C.y - c.C.y);
    p2.y = y + (h / d) * (C.x - c.C.x);
    return p1 == p2 ? vector<pair<T, T>> { p1 } : vector<pair<T, T>> { p1, p2 };
}

/**
*  @param  P, Q  Points.
*  @return       Intersection point/s between line PQ and this circle.
*  Time complexity: O(1)
*/
vector<pd> intersection(pair<T, T> P, pair<T, T> Q) {
    P.x -= C.x, P.y -= C.y, Q.x -= C.x, Q.y -= C.y;
    double a(P.y - Q.y), b(Q.x - P.x), c(P.x * Q.y - Q.x * P.y);
    double x0 = -a * c / (a * a + b * b), y0 = -b * c / (a * a + b * b);
    if (c*c > r*r * (a*a + b*b) + 1e-9) return {};
    if (equals(c*c, r*r * (a*a + b*b))) return { { x0, y0 } };
    double d = r * r - c * c / (a * a + b * b);
    double mult = sqrt(d / (a * a + b * b));
    double ax = x0 + b * mult + C.x;
    double bx = x0 - b * mult + C.x;
    double ay = y0 - a * mult + C.y;
    double by = y0 + a * mult + C.y;
    return { { ax, ay }, { bx, by } };
}

/**
*  @return  Tangent points looking from origin.
*  Time complexity: O(1)
*/
pair<pd, pd> tanPoints() {
    double b = hypot(C.x, C.y), th = acos(r / b);
    double d = atan2(-C.y, -C.x), d1 = d + th, d2 = d - th;
    return { {C.x + r * cos(d1), C.y + r * sin(d1)},
             {C.x + r * cos(d2), C.y + r * sin(d2)} };
}

/**
*  @param  P, Q, R  Points.
*  @return          Circle defined by those 3 points.
*  Time complexity: O(1)
*/
static Circle<double> from3(const pair<T, T>& P, const pair<T, T>& Q,
                                                  const pair<T, T>& R) {
    T a = 2 * (Q.x - P.x), b = 2 * (Q.y - P.y);
    T c = 2 * (R.x - P.x), d = 2 * (R.y - P.y);
    double det = a * d - b * c;

    // collinear points
    if (equals(det, 0)) return { { 0, 0 }, 0 };

    T k1 = (Q.x * Q.x + Q.y * Q.y) - (P.x * P.x + P.y * P.y);
    T k2 = (R.x * R.x + R.y * R.y) - (P.x * P.x + P.y * P.y);
    double cx = (k1 * d - k2 * b) / det;
    double cy = (a * k2 - c * k1) / det;
    return { { cx, cy }, dist(P, { cx, cy }) };
}

/**
*  @param  PS  Points
*  @return     Minimum enclosing circle with those points.
*  Time complexity: O(N)
*/
static Circle<double> mec(vector<pair<T, T>>& PS) {
    random_shuffle(all(PS));
    Circle<double> c(PS[0], 0);
    rep(i, 0, PS.size()) {
        if (c.position(PS[i]) != OUT) continue;
        c = { PS[i], 0 };
        rep(j, 0, i) {
            if (c.position(PS[j]) != OUT) continue;
            c = {
                { (PS[i].x + PS[j].x) / 2.0, (PS[i].y + PS[j].y) / 2.0 },
                   dist(PS[i], PS[j]) / 2.0
            };
            rep(k, 0, j)
                if (c.position(PS[k]) == OUT)
                c = from3(PS[i], PS[j], PS[k]);
        }
    }
    return c;
}

pair<T, T> C;
T r;
};
```

Polígono

```cpp
template <typename T>
struct Polygon {
    /**
    *  @param  PS  Clock-wise points.
    */
    Polygon(const vector<pair<T, T>>& PS) : vs(PS), n(vs.size()) { vs.eb(vs.front()); }

    /**
    *  @return  True if is convex.
    *  Time complexity: O(N)
    */
    bool convex() {
        if (n < 3) return false;
        ll P = 0, N = 0, Z = 0;

        rep(i, 0, n) {
            auto d = D(vs[i], vs[(i + 1) % n], vs[(i + 2) % n]);
            d ? (d > 0 ? ++P : ++N) : ++Z;
        }

        return P == n or N == n;
    }

    /**
    *  @return  Area. If points are integer, double the area.
    *  Time complexity: O(N)
    */
    T area() {
        T a = 0;
        rep(i, 0, n) a += vs[i].x * vs[i + 1].y - vs[i + 1].x * vs[i].y;
        if (is_floating_point_v<T>) return 0.5 * abs(a);
        return abs(a);
    }

    /**
    *  @return  Perimeter.
    *  Time complexity: O(N)
    */
    double perimeter() {
        double P = 0;
        rep(i, 0, n) P += dist(vs[i], vs[i + 1]);
        return P;
    }

    /**
    *  @param  P  Point
    *  @return    True if P inside polygon.
    *  Doesn't consider border points.
    *  Time complexity: O(N)
    */
    bool contains(const pair<T, T>& P) {
        if (n < 3) return false;
        double sum = 0;

        rep(i, 0, n) {
            // border points are considered outside, should
            // use contains point in segment to count them
            auto d = D(vs[i], vs[i + 1], P);
            double a = angle(P, vs[i], P, vs[i + 1]);
            sum += d > 0 ? a : (d < 0 ? -a : 0);
        }

        return equals(abs(sum), 2.0 * acos(-1.0));  // check precision
    }

    /**
    *  @param  P, Q  Points.
    *  @return       One of the polygons generated through the cut of the line PQ.
    *  Time complexity: O(N)
    */
    Polygon cut(const pair<T, T>& P, const pair<T, T>& Q) {
        vector<pair<T, T>> points;
        double EPS { 1e-9 };

        rep(i, 0, n) {
            auto d1 = D(P, Q, vs[i]), d2 = D(P, Q, vs[i + 1]);
            if (d1 > -EPS) points.eb(vs[i]);
            if (d1 * d2 < -EPS)
                points.eb(intersection(vs[i], vs[i + 1], P, Q));
        }

        return { points };
    }

    /**
    *  @return  Circumradius length.
    *  Regular polygon.
    *  Time complexity: O(1)
    */
    double circumradius() {
        double s = dist(vs[0], vs[1]);
        return (s / 2.0) * (1.0 / sin(acos(-1.0) / n));
    }

    /**
    *  @return  Apothem length.
    *  Regular polygon.
    *  Time complexity: O(1)
    */
    double apothem() {
```

```
        double s = dist(vs[0], vs[1]);
        return (s / 2.0) * (1.0 / tan(acos(-1.0) / n));
    }

private:
    // lines intersection
    pair<T, T> intersection(const pair<T, T>& P, const pair<T, T>& Q,
                            const pair<T, T>& R, const pair<T, T>& S) {
        T a = S.y - R.y, b = R.x - S.x, c = S.x * R.y - R.x * S.y;
        T u = abs(a * P.x + b * P.y + c), v = abs(a * Q.x + b * Q.y + c);
        return { (P.x * v + Q.x * u) / (u + v), (P.y * v + Q.y * u) / (u + v) };
    }

    vector<pair<T, T>> vs;
    ll n;
};
```

Reta

```
/**
*  A line with normalized coefficients.
*/
template <typename T>
struct Line {
    /**
    *  @param  P, Q  Points.
    *  Time complexity: O(1)
    */
    Line(const pair<T, T>& P, const pair<T, T>& Q)
            : a(P.y - Q.y), b(Q.x - P.x), c(P.x * Q.y - Q.x * P.y) {
        if constexpr (is_floating_point_v<T>) b /= a, c /= a, a = 1;
        else {
            if (a < 0 or (a == 0 and b < 0)) a *= -1, b *= -1, c *= -1;
            T gcd_abc = gcd(a, gcd(b, c));
            a /= gcd_abc, b /= gcd_abc, c /= gcd_abc;
        }
    }

    /**
    *  @param  P  Point.
    *  @return    True if P is in this line.
    *  Time complexity: O(1)
    */
    bool contains(const pair<T, T>& P) { return equals(a * P.x + b * P.y + c, 0); }

    /**
    *  @param  r  Line.
    *  @return    True if r is parallel to this line.
    *  Time complexity: O(1)
    */
    bool parallel(const Line& r) {
        T det = a * r.b - b * r.a;
        return equals(det, 0);
    }

    /**
    *  @param  r  Line.
    *  @return    True if r is orthogonal to this line.
    *  Time complexity: O(1)
    */
    bool orthogonal(const Line& r) { return equals(a * r.a + b * r.b, 0); }

    /**
    *  @param  r  Line.
    *  @return    Point of intersection between r and this line.
    *  Time complexity: O(1)
    */
    pd intersection(const Line& r) {
```

```
        double det = r.a * b - r.b * a;

        // same or parallel
        if (equals(det, 0)) return {};

        double x = (-r.c * b + c * r.b) / det;
        double y = (-c * r.a + r.c * a) / det;
        return { x, y };
    }

    /**
    *  @param  P  Point.
    *  @return    Distance from P to this line.
    *  Time complexity: O(1)
    */
    double dist(const pair<T, T>& P) {
        return abs(a * P.x + b * P.y + c) / hypot(a, b);
    }

    /**
    *  @param  P  Point.
    *  @return    Closest point in this line to P.
    *  Time complexity: O(1)
    */
    pd closest(const pair<T, T>& P) {
        double den = a * a + b * b;
        double x = (b * (b * P.x - a * P.y) - a * c) / den;
        double y = (a * (-b * P.x + a * P.y) - b * c) / den;
        return { x, y };
    }

    bool operator==(const Line& r) {
        return equals(a, r.a) and equals(b, r.b) and equals(c, r.c);
    }

    T a, b, c;
};
```

Segmento

```
template <typename T>
struct Segment {
    /**
    *  @param  P, Q  Points.
    */
    Segment(const pair<T, T>& P, const pair<T, T>& Q) : A(P), B(Q) {}

    /**
    *  @param  P  Point.
    *  @return    True if P is in this segment.
    *  Time complexity: O(1)
    */
    bool contains(const pair<T, T>& P) const {
        T xmin = min(A.x, B.x), xmax = max(A.x, B.x);
        T ymin = min(A.y, B.y), ymax = max(A.y, B.y);
        if (P.x < xmin || P.x > xmax || P.y < ymin || P.y > ymax) return false;
        return equals((P.y - A.y) * (B.x - A.x), (P.x - A.x) * (B.y - A.y));
    }

    /**
    *  @param  r  Segment.
    *  @return    True if r intersects with this segment.
    *  Time complexity: O(1)
    */
    bool intersect(const Segment& r) {
        T d1 = D(A, B, r.A),  d2 = D(A, B, r.B);
        T d3 = D(r.A, r.B, A), d4 = D(r.A, r.B, B);
        d1 /= d1 ? abs(d1) : 1, d2 /= d2 ? abs(d2) : 1;
        d3 /= d3 ? abs(d3) : 1, d4 /= d4 ? abs(d4) : 1;

        if ((equals(d1, 0) and contains(r.A)) or (equals(d2, 0) and contains(r.B)))
            return true;

        if ((equals(d3, 0) and r.contains(A)) or (equals(d4, 0) and r.contains(B)))
            return true;

        return (d1 * d2 < 0) and (d3 * d4 < 0);
    }

    /**
    *  @param  P  Point.
    *  @return    Closest point in this segment to P.
    *  Time complexity: O(1)
    */
    pair<T, T> closest(const pair<T, T>& P) {
        Line<T> r(A, B);
        pd Q = r.closest(P);
        double distA = dist(A, P), distB = dist(B, P);
        if (this->contains(Q)) return Q;
```

```
        if (distA <= distB) return A;
        return B;
    }

    pair<T, T> A, B;
};
```

Triângulo

```
enum Class { EQUILATERAL, ISOSCELES, SCALENE };
enum Angles { RIGHT, ACUTE, OBTUSE };

template <typename T>
struct Triangle {
    /**
    *  @param  P, Q, R  Points.
    */
    Triangle(pair<T, T> P, pair<T, T> Q, pair<T, T> R)
        : A(P), B(Q), C(R), a(dist(A, B)), b(dist(B, C)), c(dist(C, A)) {}

    /**
    *  Time complexity: O(1)
    */
    double perimeter() { return a + b + c; }
    double inradius() { return (2 * area()) / perimeter(); }
    double circumradius() { return (a * b * c) / (4.0 * area()); }

    /**
    *  @return  Area.
    *  Time complexity: O(1)
    */
    T area() {
        T det = (A.x * B.y + A.y * C.x + B.x * C.y) -
                (C.x * B.y + C.y * A.x + B.x * A.y);
        if (is_floating_point_v<T>) return 0.5 * abs(det);
        return abs(det);
    }

    /**
    *  @return  Sides class.
    *  Time complexity: O(1)
    */
    Class sidesClassification() {
        if (equals(a, b) and equals(b, c)) return EQUILATERAL;
        if (equals(a, b) or equals(a, c) or equals(b, c)) return ISOSCELES;
        return SCALENE;
    }

    /**
    *  @return  Angle class.
    *  Time complexity: O(1)
    */
    Angles anglesClassification() {
        double alpha = acos((a * a - b * b - c * c) / (-2.0 * b * c));
        double beta  = acos((b * b - a * a - c * c) / (-2.0 * a * c));
        double gamma = acos((c * c - a * a - b * b) / (-2.0 * a * b));
        double right = acos(-1.0) / 2.0;
        if (equals(alpha, right) || equals(beta, right) || equals(gamma, right))
```

```
            return RIGHT;
        if (alpha > right || beta > right || gamma > right) return OBTUSE;
        return ACUTE;
    }

    /**
    *  @return  Medians intersection point.
    *  Time complexity: O(1)
    */
    pd barycenter() {
        double x = (A.x + B.x + C.x) / 3.0;
        double y = (A.y + B.y + C.y) / 3.0;
        return {x, y};
    }

    /**
    *  @return  Circumcenter point.
    *  Time complexity: O(1)
    */
    pd circumcenter() {
        double D = 2 * (A.x * (B.y - C.y) + B.x * (C.y - A.y) + C.x * (A.y - B.y));
        T A2 = A.x * A.x + A.y * A.y, B2 = B.x * B.x + B.y * B.y,
                                      C2 = C.x * C.x + C.y * C.y;
        double x = (A2 * (B.y - C.y) + B2 * (C.y - A.y) + C2 * (A.y - B.y)) / D;
        double y = (A2 * (C.x - B.x) + B2 * (A.x - C.x) + C2 * (B.x - A.x)) / D;
        return {x, y};
    }

    /**
    *  @return  Bisectors intersection point.
    *  Time complexity: O(1)
    */
    pd incenter() {
        double P = perimeter();
        double x = (a * A.x + b * B.x + c * C.x) / P;
        double y = (a * A.y + b * B.y + c * C.y) / P;
        return {x, y};
    }

    /**
    *  @return  Heights intersection point.
    *  Time complexity: O(1)
    */
    pd orthocenter() {
        Line<T> r(A, B), s(A, C);
        Line<T> u{r.b, -r.a, -(C.x * r.b - C.y * r.a)};
        Line<T> v{s.b, -s.a, -(B.x * s.b - B.y * s.a)};
        double det = u.a * v.b - u.b * v.a;
        double x = (-u.c * v.b + v.c * u.b) / det;
        double y = (-v.c * u.a + u.c * v.a) / det;
        return {x, y};
```

```
    }

    pair<T, T> A, B, C;
    T a, b, c;
};
```

Matemática

Matriz

```
/**
 *  @brief This speeds up constant space dp.
 *  The matrix will be the coefficients of the dp.
 *  Like ndp[i] += dp[j] * m[i][j].
 *  If the dp doesn't look like that it may still work but
 *  probably will need to have a custom product.
*/
template <typename T>
struct Matrix {
    Matrix(const vector<vector<T>>& matrix) : mat(matrix), n(mat.size()) {}
    Matrix(ll m) : n(m) { mat.resize(n, vector<T>(n)); }
    vector<T>& operator[](ll i) { return mat[i]; }

    /**
     *  @param  other  Other matrix.
     *  @return        Product of matrices.
     *  It may happen that this needs to be custom.
     *  Think of it as a transition like on Floyd-Warshall.
     *  Time complexity: O(N^3)
    */
    Matrix operator*(Matrix& other) {
        Matrix res(n);
        rep(i, 0, n) rep(j, 0, n) rep(k, 0, n)
            res[i][k] += mat[i][j] * other[j][k];
        return res;
    }

    /**
     *  @param  b  Exponent.
     *  Time complexity: O(N^3 * log(B))
    */
    void exp(ll b) {
        Matrix &self = *this, res(n);
        rep(i, 0, n) res[i][i] = 1;
        while (b > 0) {
            if (b & 1) res = res * self;
            self = self * self;
            b /= 2;
        }
        self = res;
    }

    vector<vector<T>> mat;
    ll n;
};
```

Strings

Hash

```
constexpr ll M1 = (ll)1e9 + 7, M2 = (ll)1e9 + 9;
#define H pll
ll sum(ll a, ll b, ll m) { return (a += b.x) >= m ? a - m : a; };
ll sub(ll a, ll b, ll m) { return (a -= b.x) >= m ? a + m : a; };
H operator*(H a, H b) { return { a.x * b.x % M1, a.y * b.y % M2 }; }
H operator+(H a, H b) { return { sum(a.x, b.x, M1), sum(a.y, b.y, M2) }; }
H operator-(H a, H b) { return { sub(a.x, b.x, M1), sub(a.y, b.y, M2) }; }
    struct Hash {
    /**
     *  @param  s  String.
     *  p^n + p^n-1 + ... + p^0.
     *  Can use a segtree to update as seen in the commented blocks.
     *  Time complexity: O(N)
    */
    Hash(const string& s) : n(s.size()), ps(n + 1), pw(n + 1) {
        pw[0] = { 1, 1 };
        vector<H> ps_(n);
        rep(i, 0, n) {
            ll v      = s[i] - 'a' + 1;
            ps[i + 1] = ps[i] * p + H(v, v);
            pw[i + 1] = pw[i] * p;
        }
        // rep(i, 0, n) {
        //     ll v    = s[i] - 'a' + 1;
        //     ps_[i] = pw[n - i - 1] * H(v, v);
        // }
        // ps.build(ps_);
    }

    /**
     *  @param  i  Index.
     *  @param  c  Character.
     *  Sets character at index i to c.
     *  Time complexity: O(log(N))
    */
    // void set(ll i, char c) {
    //     ll v = c - 'a' + 1;
    //     ps.setQuery(i, i, pw[n - i - 1] * H(v, v));
    // }

    /**
     *  @param  i, j  Interval.
     *  @return       Pair of integers that represents the substring [i, j].
     *  Time complexity: O(1), If using segtree: O(log(N))
    */
    H operator()(ll i, ll j) {
        assert(0 <= i and i <= j and j < n);
```

```
        return ps[j + 1] - ps[i] * pw[j + 1 - i];
        // return ps.setQuery(i, j) * pw[i];
    }

    ll n;
    // Segtree<H> ps;
    vector<H> ps, pw;
    H p = { 31, 29 };
};
```

Suffix Automaton

```
struct SuffixAutomaton {
    /**
    *  @param  s  String.
    *  Time complexity: O(Nlog(N))
    */
    SuffixAutomaton(const string &s) {
        make_node();
        for (auto c : s) add(c);

        // preprocessing for count of countAndFirst
        vector<pll> order(sz - 1);
        rep(i, 1, sz) order[i - 1] = {len[i], i};
        sort(all(order), greater<>());
        for (auto [_, i] : order) cnt[lnk[i]] += cnt[i];

        // preprocessing for kThSub and kThDSub
        dfs(0);
    }

    /**
    *  @param  t  String.
    *  @return    Pair with how many times substring t
    *             appears and index of first occurrence.
    *  Time complexity: O(M)
    */
    pll countAndFirst(const string &t) {
        ll u = 0;
        for (auto c : t) {
            ll v = next[u][c - 'a'];
            if (!v) return {0, -1};
            u = v;
        }
        return {cnt[u], fpos[u] - t.size() + 1};
    }

    /**
    *  @returns  Amount of distinct substrings.
    *  Time complexity: O(N)
    */
    ll dSubs() {
        ll res = 0;
        rep(i, 1, sz)
            res += len[i] - len[lnk[i]];
        return res;
    }

    /**
    *  @returns  Vector with amount of distinct substrings of each size.
    *  Time complexity: O(N)
```

```
*/
vll dSubsBySz() {
    vll hs(sz, -1);
    hs[0] = 0;
    queue<ll> q;
    q.emplace(0);
    ll mx = 0;
    while (!q.empty()) {
        ll u = q.front();
        q.pop();
        rep(i, 0, alpha) {
            ll v = next[u][i];
            if (!v) continue;
            if (hs[v] == -1) {
                q.emplace(v);
                hs[v] = hs[u] + 1;
                mx    = max(mx, len[v]);
            }
        }
    }

    vll res(mx);
    rep(i, 1, sz) {
        ++res[hs[i] - 1];
        if (len[i] < mx) --res[len[i]];
    }
    rep(i, 1, mx) res[i] += res[i - 1];
    return res;
}

/**
*  @param  k  Number, starts from 0.
*  @return    k-th substring lexographically.
*  Time complexity: O(N)
*/
string kThSub(ll k) {
    k += n;
    string res;
    ll u = 0;
    while (k >= cnt[u]) {
        k -= cnt[u];
        rep(i, 0, alpha) {
            ll v = next[u][i];
            if (!v) continue;
            if (rcnt[v] > k) {
                res += i + 'a', u = v;
                break;
            }
            k -= rcnt[v];
        }
    }
```

```
    return res;
}

/**
*  @param  k  Number, starts from 0.
*  @return    k-th distinct substring lexographically.
*  Time complexity: O(N)
*/
string kThDSub(ll k) {
    string res;
    ll u = 0;
    while (k >= 0) {
        rep(i, 0, alpha) {
            ll v = next[u][i];
            if (!v) continue;
            if (dcnt[v] > k) {
                res += i + 'a', --k, u = v;
                break;
            }
            k -= dcnt[v];
        }
    }
    return res;
}

private:
    ll make_node(ll _len = 0, ll _fpos = -1, ll _lnk = -1, ll _cnt = 0,
                 ll _rcnt = 0, ll _dcnt = 0) {
        next.eb(vll(alpha));
        len.eb(_len), fpos.eb(_fpos), lnk.eb(_lnk);
        cnt.eb(_cnt), rcnt.eb(_rcnt), dcnt.eb(_dcnt);
        return sz++;
    }

    void add(char c) {
        c -= 'a', ++n;
        ll u = make_node(len[last] + 1, len[last], 0, 1);
        ll p = last;
        while (p != -1 and !next[p][c]) {
            next[p][c] = u;
            p = lnk[p];
        }
        if (p == -1) lnk[u] = 0;
        else {
            ll q = next[p][c];
            if (len[p] + 1 == len[q]) lnk[u] = q;
            else {
                ll v = make_node(len[p] + 1, fpos[q], lnk[q]);
                next[v] = next[q];
                while (p != -1 and next[p][c] == q) {
                    next[p][c] = v;
```

```
                p = lnk[p];
            }
            lnk[q] = lnk[u] = v;
        }
    }
    last = u;
}

void dfs(ll u) {  // for kThSub and kThDSub
    dcnt[u] = 1, rcnt[u] = cnt[u];
    rep(i, 0, alpha) {
        ll v = next[u][i];
        if (!v) continue;
        if (!dcnt[v]) dfs(v);
        dcnt[u] += dcnt[v];
        rcnt[u] += rcnt[v];
    }
}

vvll next;
vll len, fpos, lnk, cnt, rcnt, dcnt;
ll sz = 0, last = 0, n = 0, alpha = 26;
};
```

Trie

```
// empty head, tree is made by the prefixs of each string in it.
struct Trie {
    Trie() : n(0), to(MAXN + 1, vll(26)), mark(MAXN + 1), qnt(MAXN + 1) {}

    /**
    *  @param  s  String.
    *  Time complexity: O(N)
    */
    void insert(const string& s) {
        ll u = 0;
        for (auto c : s) {
            ll& v = to[u][c - 'a'];
            if (!v) v = ++n;
            u = v, ++qnt[u];
        }
        ++mark[u], ++qnt[0];
    }

    /**
    *  @param  s  String.
    *  Time complexity: O(N)
    */
    void erase(const string& s) {
        ll u = 0;
        for (char c : s) {
            ll& v = to[u][c - 'a'];
            u = v, --qnt[u];
            if (!qnt[u]) v = 0, --n;
        }
        --mark[u], --qnt[0];
    }

    void dfs(ll u) {
        rep(i, 0, 26) {
            ll v = to[u][i];
            if (v) {
                if (mark[v]) cout << "\e[31m";
                cout << (char)(i + 'a') << " \e[m";
                dfs(to[u][i]);
            }
        }
    }

    constexpr ll MAXN = 5e5;
    ll n;
    vvll to;  // 0 is head
    // mark: quantity of strings that ends in this node.
    // qnt: quantity of strings that pass through this node.
```

```
    vll mark, qnt;
};
```

Outros

RMQ

```
template <typename T>
struct RMQ {
    /**
    *  @param  xs  Target vector.
    *  Time complexity: O(Nlog(N))
    */
    RMQ(const vector<T>& xs) : n(xs.size()), st(LOG, vector<T>(n)) {
        st[0] = xs;
        rep(i, 1, LOG)
            for (ll j = 0; j + (1 << i) <= n; ++j)
                st[i][j] = min(st[i - 1][j], st[i - 1][j + (1 << (i - 1))]);
    }

    /**
    *  @param  i, j  Interval.
    *  @return       Minimum value in interval [i, j].
    *  Time complexity: O(1)
    */
    T query(ll l, ll r) {
        assert(0 <= l and l <= r and r < n);
        ll lg = (ll)log2(r - l + 1);
        return min(st[lg][l], st[lg][r - (1 << lg) + 1]);
    }

    ll n, LOG = 25;
    vector<vector<T>> st;
};
```

Soma de prefixo 2D

```
/**
 *  @brief Make rectangular interval sum queries.
*/
template <typename T>
struct Psum2D {
    /**
     *  @param  xs  Matrix.
     *  Time complexity: O(N^2)
    */
    Psum2D(const vector<vector<T>>& xs)
        : n(xs.size()), m(xs[0].size()), psum(n + 1, vector<T>(m + 1)) {
        rep(i, 0, n) rep(j, 0, m)
            psum[i + 1][j + 1] = psum[i + 1][j] + psum[i][j + 1] + xs[i][j] - psum[i][j];
    }

    /**
     *  @param  ly, hy  Vertical   interval.
     *  @param  lx, hx  Horizontal interval.
     *  @return         Sum in that rectangle.
     *  Time complexity: O(1)
    */
    T query(ll ly, ll lx, ll hy, ll hx) {
        assert(0 <= ly and ly <= hy and hy < n and 0 <= lx and lx <= hx and hx < m);
        ++ly, ++lx, ++hy, ++hx;
        return psum[hy][hx] - psum[hy][lx - 1] - psum[ly - 1][hx] + psum[ly - 1][lx - 1];
    }

    ll n, m;
    vector<vector<T>> psum;
};
```

Soma de prefixo 3D

```cpp
/**
 *  @brief Make cuboid interval sum queries.
*/
template <typename T>
struct Psum3D {
    /**
     *  @param  xs  3D Matrix.
     *  Time complexity: O(N^3)
    */
    Psum3D(const vector<vector<vector<T>>>& xs)
            : n(xs.size()), m(xs[0].size()), o(xs[0][0].size()),
              psum(n + 1, vector<vector<T>>(m + 1, vector<T>(o + 1)) {
        rep(i, 1, n + 1) rep(j, 1, m + 1) rep(k, 1, o + 1) {
            psum[i][j][k] = psum[i - 1][j][k] + psum[i][j - 1][k] + psum[i][j][k - 1];
            psum[i][j][k] -= psum[i][j - 1][k - 1] + psum[i - 1][j][k - 1]
                                                   + psum[i - 1][j - 1][k];
            psum[i][j][k] += psum[i - 1][j - 1][k - 1] + xs[i - 1][j - 1][k - 1];
        }
    }

    /**
     *  @param  ly, hy  First interval.
     *  @param  lx, hy  Second interval.
     *  @param  lz, hz  Third interval
     *  @return         Sum in that cuboid.
     *  Time complexity: O(1)
    */
    T query(ll lx, ll ly, ll lz, ll hx, ll hy, ll hz) {
        assert(0 <= lx and lx <= hx and hx < n);
        assert(0 <= ly and ly <= hy and hy < m);
        assert(0 <= lz and lz <= hz and hz < o);
        ++lx, ++ly, ++lz, ++hx, ++hy, ++hz;
        T res = psum[hx][hy][hz] - psum[lx - 1][hy][hz] - psum[hx][ly - 1][hz]
                                                        - psum[hx][hy][lz - 1];
        res += psum[hx][ly - 1][lz - 1] + psum[lx - 1][hy][lz - 1]
                                        + psum[lx - 1][ly - 1][hz];
        res -= psum[lx - 1][ly - 1][lz - 1];
        return res;
    }

    ll n, m, o;
    vector<vector<vector<T>>> psum;
};
```

Utils

Aritmética modular

```cpp
template <typename T>
T exp(T a, ll b) {
    T res = 1;
    while (b) {
        if (b & 1) res *= a;
        a *= a, b /= 2;
    }
    return res;
}

constexpr ll MOD = (ll)1e9 + 7;
template <ll M = MOD>
struct Mi {
    ll v;
    Mi() : v(0) {}
    Mi(ll x) : v(x) {
        if (v >= M or v < -M) v %= M;
        v += v < 0 ? M : 0;
    }
    friend bool operator==(Mi a, Mi b) { return a.v == b.v; }
    friend bool operator!=(Mi a, Mi b) { return a.v != b.v; }
    friend ostream& operator<<(ostream& os, Mi a) { return os << a.v; }
    Mi& operator+=(Mi b) { return v -= ((v += b.v) >= M ? M : 0), *this; }
    Mi& operator-=(Mi b) { return v += ((v -= b.v)  < 0 ? M : 0), *this; }
    Mi& operator*=(Mi b) { return v = v * b.v % M, *this; }
    Mi& operator/=(Mi b) { return *this *= exp(b, M - 2); }
    friend Mi operator+(Mi a, Mi b) { return a += b; }
    friend Mi operator-(Mi a, Mi b) { return a -= b; }
    friend Mi operator*(Mi a, Mi b) { return a *= b; }
    friend Mi operator/(Mi a, Mi b) { return a /= b; }
};
```

Bits

```cpp
ll msb(ll x) { return (x == 0 ? 0 : 64 - __builtin_clzll(x)); }
ll lsb(ll x) { return __builtin_ffsll(x); }
```

Ceil division

```cpp
ll ceilDiv(ll a, ll b) { assert(b != 0); return a / b + ((a ^ b) > 0 && a % b != 0); }
```

Conversão de índices

```
#define K(i, j) ((i) * w + (j))
#define I(k)    ((k) / w)
#define J(k)    ((k) % w)
```

Compressão de coordenadas

```
/**
 *  @brief      Compress values from a vector.
 *  @param  xs  Vector.
 *  @return     Maps with the compressed values and uncompressed values.
 *  Time complexity: O(Nlog(N))
*/
template <typename T>
pair<map<T, ll>, map<ll, T>> compress(vector<T> xs) {
    ll i = 0;
    sort(all(xs));
    xs.erase(unique(all(xs)), xs.end());
    map<ll, T> pm;
    map<T, ll> mp;
    for (T x : xs) {
        pm[i] = x;
        mp[x] = i++;
    }
    return { mp, pm };
}
```

Fatos

Bitwise

`a + b = (a & b) + (a | b)`.

`a + b = a ^ b + 2 * (a & b)`.

`a ^ b = ~(a & b) & (a | b)`.

Geometria

Quantidade de pontos inteiros num segmento: `gcd(abs(Px - Qx), abs(Py - Qy)) + 1`. `P, Q` *são os pontos extremos do segmento.*

Teorema de Pick: Seja `A` *a área da treliça,* `I` *a quantidade de pontos interiores com coordenadas inteiras e* `B` *os pontos da borda com coordenadas inteiras. Então,* `A = I + B / 2 - 1` *e* `I = (2A + 2 - B) / 2`.

Distância de Chebyshev: `dist(P, Q) = max(Px - Qx, Py - Qy)`. `P, Q` *são dois pontos.*

Manhattam para Chebyshev: Feita a transformação `(x, y) -> (x + y, x - y)`*, temos uma equivalência entre as duas distâncias, podemos agora tratar* `x` *e* `y` *separadamente, fazer bounding boxes, entre outros...*

Matemática

Quantidade de divisores de um número: produtório de `(a + 1)`. `a` *é o expoente do* `i`*-ésimo fator primo.*

Soma dos divisores de um número: produtório de `(p^(a + 1) - 1)/(p - 1)`. `a` *é o expoente do* `i`*-ésimo fator primo* `p`.

Produto dos divisores de um número: `x^(qd(x)/2)`. `x` *é o número,* `qd(x)` *é a quantidade de divisores dele.*

Maior quantidade de divisores de um número: `< 10^3` *é* `32`, `< 10^6` *é* `240`, `< 10^18` *é* `107520`.

Maior diferença entre dois primos consecutivos: `< 10^18` *é* `1476`. *(Podemos concluir que a partir de um número arbitrário a distância para o coprimo mais próximo é bem menor que esse valor).*

Maior quantidade de primos na fatoração de um número: `< 10^3` *é* `9`, `< 10^6` *é* `19`.

Números primos interessantes: `2^31 - 1, 2^31 + 11, 10^16 + 61, 10^18 - 11, 10^18 + 3`.

Quantidade de coprimos de `x` *em* `[1, x]`*: produtório de* `p^(a - 1)(p - 1)`. `a` *é o expoente do* `i`*-ésimo fator primo* `p`.

`gcd(a, b) = gcd(a, a - b)`, `gcd(a, b, c) = gcd(a, a - b, a - c)`*, segue o padrão.*

Para calcular o `lcm` *de um conjunto de números com módulo, podemos fatorizar cada um, cada primo gerado vai ter uma potência que vai ser a maior, o produto desses primos elevados à essa potência será o* `lcm`*, se queremos módulo basta fazer nessas operações.*

Divisibilidade por `3`*: soma dos algarismos divisível por* `3`.

Divisibilidade por `4`*: número formado pelos dois últimos algarismos, divisível por* `4`.

Divisibilidade por `6`*: se divisível por* `2` *e* `3`.

Divisibilidade por `7`*: soma alternada de blocos de três algarismos, divisível por* `7`.

Divisibilidade por `8`*: número formado pelos três últimos algarismos, divisível por* `8`.

Divisibilidade por `9`*: soma dos algarismos divisível por* `9`.

Divisibilidade por `11`*: soma alternada dos algarismos divisível por* `11`.

Divisibilidade por `12`*: se divisível por* `3` *e* `4`.

Soma da progressão geométrica: `(a_n * r - a_1) / (r - 1)`.

Soma de termos ao quadrado: `1^2 + 2^2 + ... + n^2 = n(n + 1)(2n + 1) / 6`.

Ao realizar operações com aritmética modular a paridade (sem módulo) não é preservada, se quer saber a paridade na soma vai checando a paridade do que está sendo somado e do número, na multiplicação e divisão conte e mantenha a quantidade de fatores iguais a dois.

`a^(b % p) % p != a^b % p`*, então é necessário que* `b` *seja sempre menor que* `p`*, mas devido ao Pequeno Teorema de Fermat podemos fazer* `b % (p - 1)` *isso vai garantir que a operação tenha o valor correto.*

Números de Catalan `Cn`*: representa a quantidade de expressões válidas com parênteses de tamanho* `2n`*. Também são relacionados às árvores, existem* `Cn` *árvores binárias de* `n` *vértices e* `Cn-1` *árvores de* `n` *vértices (as árvores são caracterizadas por sua aparência).* `Cn = binom(2n, n)/(n + 1)`*. A intuição boa é imaginar o problema como uma caminhada numa matriz, do ponto* `(0,0)` *até o ponto* `(M, N)`*, onde* `M, N` *é a quantidade de parênteses de abrir e de fechar, aí a quantidade de expressões válidas é o total* `binom(N + M, N)` *menos as inválidas, que dá para interpretar como as que vem do ponto simétrico à reta* `y = x + k = n = m + k + 1` *até* `(M, N)`*, cruzando a reta inválida.* `k` *é a quantidade de parênteses já abertos, aí fica* `binom(N + M, N) - binom(N + M, M + K + 1)`.

Lema de Burnside: o número de combinações em que simétricos são considerados iguais é o somatório de `k` *entre* `[1, n]` *de* `c(k)/n`. `n` *é a quantidade de maneiras de mudar a posição de uma combinação e* `c(k)` *é a quantidade de combinações que são consideradas iguais na* `k`*-ésima maneira.*

Strings

Sejam `p` *e* `q` *dois períodos de uma string* `s`*. Se* `p + q − mdc(p, q) ≤ |s|`*, então* `mdc(p, q)` *também é período de* `s`.

Relação entre bordas e períodos: A sequência `|s| − |border(s)|, |s| − |border^2(s)|, ..., |s| − |border^k(s)|` *é a sequência crescente de todos os possíveis períodos de* `s`.

Outros

> *Princípio da inclusão e exclusão: a união de* n *conjuntos é a soma de todas as interseções de um número ímpar de conjuntos menos a soma de todas as interseções de um número par de conjuntos.*

> *Regra de Warnsdorf: heurística para encontrar um caminho em que o cavalo passa por todas as casas uma única vez, sempre escolher o próximo movimento para a casa com o menor número de casas alcançáveis.*

> *Para utilizar ordenação customizada em sets/maps:* `set<ll, decltype([](ll a, ll b) { ... })`.

Igualdade flutuante

```
/**
 *  @param  a, b  Floats.
 *  @return       True if they are equal.
*/
template <typename T, typename S>
bool equals(T a, S b) { return abs(a - b) < 1e-9; }
```

Overflow check

```
ll mult(ll a, ll b) {
    if (abs(a) >= LLONG_MAX / abs(b))
        return LLONG_MAX;  // overflow
    return a * b;
}

ll sum(ll a, ll b) {
    if (abs(a) >= LLONG_MAX - abs(b))
        return LLONG_MAX;  // overflow
    return a + b;
}
```