Competitive Programming Notebook

As complexidades temporais são estimadas e simplificadas!

Sumário

- Fatos
  - Bitwise
  - Geometria
- Matemática
  - Strings
  - Outros

Template

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

#ifdef croquete  // BEGIN TEMPLATE ----------------------|
#include "dbg/dbg.h"
#define fio freopen("in.txt", "r", stdin)
#else
#define dbg(...)
#define fio cin.tie(0)->sync_with_stdio(0)
#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() {
    fio;
    ll t = 1;
    cin >> t;
    while (t--) solve();
}  // END TEMPLATE --------------------------------------|

void solve() {
}
```

Outros defines

```
// BEGIN EXTRAS -----------------------------------------|
#define ull unsigned ll
#define vvvll vector<vvll>
#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
```

Pragmas

```
#pragma GCC target("popcnt") // if solution involves bitset
```

Debug

```
#include <bits/stdc++.h>
using namespace std;
template <typename T> void p(T x) {
    int f = 0;
    #define _(a, b) if constexpr (requires {(a);}) { (b); } else
    #define D(d) cerr << "\e[94m" << (f++ ? d : "")
    _(cout << x, cerr << x)
    _(x.pop(), {struct y : T {using T::c;}; p(static_cast<y>(x).c);}) {
        cerr << '{';
        _(get<0>(x), apply([&](auto... a) {((D(","), p(a)), ...);}, x))
        for (auto i : x) (requires {begin(*begin(x));} ? cerr << "\n\t" : D(",")), p(i);
        cerr << '}';
    }
} template <typename... T>
void pr(T... 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)
*/
double angle(pd P, pd Q, pd R, pd S) {
    assert(P != Q && R != S);
    double ux = P.x - Q.x, uy = P.y - Q.y;
    double vx = R.x - S.x, vy = R.y - S.y;
    double cross = ux * vy - uy * vx;
    double dot = ux * vx + uy * vy;
    return atan2(cross, dot);  // oriented
}
```

Distância entre pontos

```
/**
 *  @param  P, Q  Points.
 *  @return       Distance between points.
 *  Time complexity: O(1)
*/
double dist(pd P, pd Q) { return hypot(P.x - Q.x, P.y - Q.y); }
```

Envoltório convexo

```
template <typename T>
vector<pair<T, T>> make_hull(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 && 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>> monotone_chain(vector<pair<T, T>> PS) {
    vector<pair<T, T>> lower, upper;
    sort(all(PS));
    lower = make_hull(PS);
    reverse(all(PS));
    upper = make_hull(PS);
    lower.pop_back();
    lower.emplace(lower.end(), all(upper));
    return lower;
}
```

Orientação de ponto

```
/**
 * @param  A, B     Points defining the line segment AB.
 * @param  P        The point to check.
 * @return          Cross product value (2 * signed area).
 * Positive: P is to the left of AB (Counter-Clockwise).
 * Negative: P is to the right of AB (Clockwise).
 * Zero:     P is collinear with AB.
 * Time complexity: O(1)
*/
template <typename T>
T D(pair<T, T> A, pair<T, T> B, pair<T, T> P) {
    return (B.x - A.x) * (P.y - A.y) - (B.y - A.y) * (P.x - A.x);
}
```

Verificar Quadrado

```
/**
 *  @param  PS  Points.
 *  @return     True if those 4 points forms a square.
 *  Time complexity: ~O(1)
*/
bool is_square(const vpll& ps) {
    if (ps.size() != 4) return false;
    vll ds;
    rep(i, 0, 4) rep(j, i + 1, 4) {
        ll dx = ps[i].x - ps[j].x, dy = ps[i].y - ps[j].y;
        ds.eb(dx * dx + dy * dy);
    }
    sort(all(ds));
    ds.erase(unique(all(ds)), ds.end());
    return ds.size() == 2 && 2 * ds[0] == ds[1];
}
```

Slope

```
/**
 *  @param  P, Q  Points.
 *  @return       The irreducible fraction dy/dx, dy always positive.
 *  Time complexity: O(log(N))
*/
pll slope(pll P, pll Q) {
    ll dy = P.y - Q.y, dx = P.x - Q.x;
    if (dy < 0 || (dy == 0 && dx < 0)) dy *= -1, dx *= -1;
    ll g = gcd(dy, dx);
    return {dy / g, dx / g};
}
```

Mediatriz

```
/**
 *  @param  P, Q  Points.
 *  @return       Perpendicular bisector to segment PQ.
 *  Time complexity: O(1)
*/
template <typename T>
Line<T> perpendicular_bisector(pair<T, T> P, 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)
*/
pd rotate(pd 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 = 22;
vvll parent;

/**
 *  @param  ps  Tree/Successor graph.
 *  Time complexity: O(Nlog(N))
*/
void populate(const vll& ps) {
    ll n = ps.size();
    parent = vvll(n, vll(LOG));
    rep(i, 0, n) parent[i][0] = ps[i];
    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 kth(ll u, ll k) {
    assert(!parent.empty() && 0 <= u && u < parent.size() && k >= 0);
    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 centroid(const vvll& g, ll u, ll p = -1) {
    ll sz = g.size();
    if (p == -1) subtree = vll(sz, 1), subtree_dfs(g, u, p);
    for (ll v : g[u]) if (v != p && subtree[v] * 2 > sz)
        return centroid(g, v, u);
    return u;
}
```

Decomposição de Centróide

```
vll parent, subtree;

ll subtree_dfs(const vvll& g, ll u, ll p) {
    subtree[u] = 1;
    for (ll v : g[u]) if (v != p && parent[v] == -1)
        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 centroid_decomp(const vvll& g, ll u = 0, ll p = -1, ll sz = 0) {
    if (p == -1) p = -2, parent= vll(g.size(), -1), subtree = vll(g.size());
    if (sz == 0) sz = subtree_dfs(g, u, -1);
    for (ll v : g[u]) if (parent[v] == -1 && subtree[v] * 2 > sz)
        return subtree[u] = 0, centroid_decomp(g, v, p, sz);
    parent[u] = p;
    for (ll v : g[u]) if (parent[v] == -1) centroid_decomp(g, v, u);
}
```

Euler Tour

```cpp
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 euler_tour(const vvll& g, ll u, ll p = -1) {
    if (p == -1) timer = 0, st = et = vll(g.size());
    st[u] = ++timer;
    for (ll v : g[u]) if (v != p)
        euler_tour(g, v, u);
    et[u] = timer;
}
```

Menor ancestral comum (LCA)

```cpp
/**
 *  @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 && u < parent.size() && 1 <= v && v < parent.size());
    if (depth[u] < depth[v]) swap(u, v);
    ll k = depth[u] - depth[v];
    u = kth(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

```cpp
/**
 *  @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)
*/
constexpr ll NC = LLONG_MIN;  // negative cycle
pair<vll, vll> spfa(const vvpll& g, ll s) {
    ll n = g.size();
    vll ds(n, LLONG_MAX), cnt(n), pre(n);
    vector<bool> inq(n);
    queue<ll> q;
    ds[s] = 0, q.emplace(s);
    while (!q.empty()) {
        ll u = q.front(); q.pop(), inq[u] = false;
        for (auto [w, v] : g[u]) {
            if (ds[u] == NC) {
                // spread negative cycle
                if (ds[v] != NC)
                    q.emplace(v), inq[v] = true;
                ds[v] = NC;
            } else if (ds[u] + w < ds[v]) {
                ds[v] = ds[u] + w, pre[v] = u;
                if (!inq[v]) {
                    q.emplace(v), inq[v] = true;
                    if (++cnt[v] > n) ds[v] = NC;
                }
            }
        }
    }
    return {ds, pre};
}
```

BFS 0/1

```cpp
/**
 *  @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

```cpp
/**
 *  @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(Nlog(N))
*/
vll eulerian_path(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 && check != -1) return {};  // impossible

    rep(u, 0, n) {
        if (e != -1 && (u == s || u == e)) continue;
        if (in_degree[u] != h[u].size() || (!d && 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;
}
```

Detecção de ciclo

```
/**
 *  @param  g      Graph [id of edge, v].
 *  @param  edges  Edges flag (true if wants edges).
 *  @param  d      Directed flag (true if g is directed).
 *  @return        Vector with cycle vertices or edges.
 *  Empty if no cycle.
 *  Time complexity: O(V + E)
*/
vll cycle(const vvpll& g, bool edges, bool d) {
    ll n = g.size();
    vll color(n + 1), parent(n + 1), edge(n + 1), res;
    auto dfs = [&](auto&& self, ll u, ll p) -> ll {
        color[u] = 1;
        bool parent_skipped = false;
        for (auto [i, v] : g[u]) {
            if (!d && v == p && !parent_skipped)
                parent_skipped = true;
            else if (color[v] == 0) {
                parent[v] = u, edge[v] = i;
                if (ll end = self(self, v, u); end != -1) return end;
            } else if (color[v] == 1) {
                parent[v] = u, edge[v] = i;
                return v;
            }
        }
        color[u] = 2;
        return -1;
    };
    rep(u, 0, n) if (color[u] == 0)
        if (ll end = dfs(dfs, u, -1), start = end; end != -1) {
            do {
                res.eb(edges ? edge[end] : end);
                end = parent[end];
            } while (end != start);
            reverse(all(res));
            return res;
        }
    return {};
}
```

Dijkstra

```
/**
 *  @param  g  Graph (w, v).
 *  @param  s  Starting vertex.
 *  @return    Vectors with smallest distances from every vertex to s and the paths.
 *  The distance is final only on the first time the node is out of the queue.
 *  It doesn't work with negative weights, but if you can find a potential function
 *  we can turn all weights to positive.
 *  A potential function is such that:
 *  new  weight is w' = w + p(u) - p(v) >= 0.
 *  real dist will be dist(u, v) = dist'(u, v) - p(u) + p(v).
 *  Time complexity: O(Elog(V))
*/
pair<vll, vll> dijkstra(const vvpll& g, ll s) {
    vll ds(g.size(), LLONG_MAX), pre = ds;
    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 get_path(const vll& pre, ll u) {
    vll p;
    while (u != LLONG_MAX) p.eb(u), u = pre[u];
    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] == oo, unreachable
 *  If ds[u][v] == -oo, negative cycle.
 *  Time complexity: O(V³)
*/
constexpr ll oo = 4e18;
vvll floyd_warshall(const vvpll& g) {
    ll n = g.size();
    vvll ds(n, vll(n, oo));
    rep(u, 0, n) {
        ds[u][u] = 0;
        for (auto [w, v] : g[u]) ds[u][v] = min(ds[u][v], w);
    }
    rep(k, 0, n) rep(u, 0, n) rep(v, 0, n)
        ds[u][v] = min(ds[u][v], ds[u][k] + ds[k][v]);
    rep(k, 0, n) if (ds[k][k] < 0)
        rep(u, 0, n) rep(v, 0, n)
            if (ds[u][k] != oo && ds[k][v] != oo)
                ds[u][v] = -oo;
    return ds;
}
```

Johnson

```
/**
 *  @param  g  Graph (w, v).
 *  @return    Vector with smallest distances between every vertex.
 *  Weights can be negative.
 *  If ds[u][v] == LLONG_MAX, unreachable
 *  Will return all ds = NC if negative cycle.
 *  Requires Bellman-Ford and Dijkstra.
 *  If complete graph is worse than Floyd-Warshall.
 *  Time complexity: O(EVlog(N))
*/
vvll johnson(vvpll& g) {
    ll n = g.size();
    rep(v, 1, n) g[0].eb(0, v);
    auto [dsb, _] = spfa(g, 0);
    vvll dsj(n, vll(n, NC));
    rep(u, 1, n) {
        if (dsb[u] == NC) return dsj;  // negative cycle
        for (auto& [w, v] : g[u])
            w += dsb[u] - dsb[v];
    }
    rep(u, 1, n) {
        auto [dsd, __] = dijkstra(g, u);
        rep(v, 1, n)
            if (dsd[v] == LLONG_MAX)
                dsj[u][v] = LLONG_MAX;
            else
                dsj[u][v] = dsd[v] - dsb[u] + dsb[v];
    }
    return dsj;
}
```

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 [leader, cc].
 *  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), leader(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) leader[v] = u;
    }
    rep(u, 0, n) for (ll v : g[u]) if (leader[u] != leader[v])
        cond[leader[u]].eb(leader[v]);
    return {cond, scc, leader};
}
```

MST

```
/**
 *  @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 + 1);
    vtll mst;
    sort(all(edges));  // change order if want maximum
    for (auto [w, u, v] : edges)
        if (dsu.merge(u, v))
            mst.eb(w, u, v);
    return mst;
}
```

Ordenação topológica

```
/**
 *  @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(Vlog(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;
}
```

Fluxo

```
// To get path from source to sink, do dfs, only go if edge is real and c is 0.
// When getting the path set each w in the path to 1.
struct Flow {
    struct Edge { bool real; ll c, v, rev, w; };
    ll n, s, t, l = 1;
    vector<vector<Edge>> g;
    vll ds, ptr;

    Flow(ll n, ll source, ll sink)
        : n(n), s(source), t(sink), g(n + 1), ds(n + 1), ptr(n + 1) {}

    void add_edge(ll c, ll u, ll v, ll w = 0) {  // add w if mcmf
        if (c > 1) l = 1 << 31;
        g[u].eb(true, c, v, g[v].size(), w);
        g[v].eb(false, 0, u, g[u].size() - 1, -w);
    }

    // Time complexity: O(E²logC), with unit capacities it's better.
    ll max_flow() {
        ll f = 0;
        while (l >>= 1) while (bfs())
            while (ll nf = dfs(s, LLONG_MAX)) f += nf;
        return f;
    }

    // If bipartite, instead check for each (u, v) if c is 0.
    vpll min_cut() {
        max_flow();
        vpll res;
        rep(u, 0, n + 1) if (ds[u] != 0)
            for (auto e : g[u])
                if (e.real && e.c == 0 && ds[e.v] == 0)
                    res.eb(u, e.v);
        return res;
    }

    // Reuses ptr as parent_edge_idx
    pll min_cost_max_flow(ll k) {
        ll c = 0, f = 0;
        vll pre(n + 1);
        while (f < k && spfa(pre)) {
            ll p = k - f;
            for (ll v = t; v != s; v = pre[v])
                p = min(p, g[pre[v]][ptr[v]].c);
            f += p, c += p * ds[t];
            for (ll v = t; v != s; v = pre[v]) {
                auto& e = g[pre[v]][ptr[v]];
                e.c -= p;
```

```cpp
                g[v][e.rev].c += p;
            }
        }
        return {c, f};
    }

private:
    ll dfs(ll u, ll f) {
        if (u == t || !f) return f;
        for (ll& i = ptr[u]; i < (ll)g[u].size(); ++i) {
            auto& e = g[u][i];
            if (ds[e.v] == ds[u] + 1)
                if (ll p = dfs(e.v, min(f, e.c))) {
                    e.c -= p, g[e.v][e.rev].c += p;
                    return p;
                }
        }
        return 0;
    }

    ll bfs() {
        fill(all(ds), 0), fill(all(ptr), 0);
        queue<ll> q;
        ds[s] = 1, q.emplace(s);
        while (!q.empty() && !ds[t]) {
            ll u = q.front(); q.pop();
            for (auto e : g[u]) if (!ds[e.v] && e.c >= 1)
                ds[e.v] = ds[u] + 1, q.emplace(e.v);
        }
        return ds[t];
    }

    ll spfa(vll& pre) {
        fill(all(ds), LLONG_MAX), fill(all(ptr), -1);
        queue<ll> q;
        vector<bool> inq(n + 1);
        ds[s] = 0, q.emplace(s), inq[s] = true;
        while (!q.empty()) {
            ll u = q.front(); q.pop(), inq[u] = false;
            rep(i, 0, g[u].size()) {
                auto& e = g[u][i];
                if (e.c > 0 && ds[u] + e.w < ds[e.v]) {
                    ds[e.v] = ds[u] + e.w;
                    ptr[e.v] = i, pre[e.v] = u;
                    if (!inq[e.v]) q.emplace(e.v), inq[e.v] = true;
                }
            }
        }
        return ds[t] != LLONG_MAX;
    }
};
```

Pontes e articulações

```cpp
/**
 *  @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 bridges_or_articulations(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 && low[v] > st[u]) res.eb(i);
                else if (!get_bridges && p != 0 && low[v] >= st[u]) res.eb(u);
                ++children;
            }
        }
        if (!get_bridges && p == 0 && 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

Algoritmo de Mo

```cpp
struct Query {
    ll l, r, idx, block;
    bool operator<(const Query& q) const {
        if (block != q.block) return block < q.block;
        return (block & 1 ? (r < q.r) : (r > q.r));
    }
};

template <typename T, typename Tans>
struct Mo {
    vector<T> vs;
    vector<Query> qs;
    const ll block_size;

    Mo(const vector<T>& xs) : vs(xs), block_size((int)ceil(sqrt(xs.size()))) {}

    void add_query(ll l, ll r) {
        qs.emplace_back(l, r, qs.size(), l / block_size);
    }

    // Time complexity: O(N sqrt(N))
    auto solve() {
        vector<Tans> answers(qs.size());
        sort(all(qs));

        ll cur_l = 0, cur_r = -1;
        for (auto q : qs) {
            while (cur_l > q.l) add(--cur_l);
            while (cur_r < q.r) add(++cur_r);
            while (cur_l < q.l) remove(cur_l++);
            while (cur_r > q.r) remove(cur_r--);
            answers[q.idx] = get_answer();
        }

        return answers;
    }

private:
    // add value at idx from data structure
    inline void add(ll idx) {}

    // remove value at idx from data structure
    inline void remove(ll idx) {}

    // extract current answer of the data structure
    inline Tans get_answer() {}
};
```

Busca ternária

```cpp
/**
 *  @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 ternary_search(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);
}
```

Kadane

```cpp
/**
 *  @param  xs  Target 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 || (csum == res && i - j + 1 < r - l + 1))
            res = csum, l = j, r = i;
    }
    return {res * (mx ? 1 : -1), l, r};
}
```

Listar combinações

```
/**
 *  @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)

```
/**
 *  @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      Target Vector.
 *  @return         Longest increasing subsequence as indexes.
 *  Time complexity: O(Nlog(N))
*/
vll lis(const vll& xs) {
    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()) ss.eb(), idx.eb();
        if (j == 0) pre[i] = -1;
        else        pre[i] = idx[j - 1];
        ss[j] = xs[i], idx[j] = i;
    }
    ll i = idx.back();
    while (i != -1) ys.eb(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 gcd_pairs(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  Target vector.
 *  @return     Vector of indexes of closest smaller.
 *  Time complexity: O(N)
*/
template <typename T>
vector<T> closests(const vector<T>& xs) {
    ll n = xs.size();
    vll dp(n, -1);  // n: to right
    // n - 1 -> 0: to right
    rep(i, 0, n) {
        ll j = i - 1;  // i + 1: to the right
        //       <  n          <= strictly bigger
        while (j >= 0 && xs[j] >= xs[i]) j = dp[j];
        dp[i] = j;
    }
    return dp;
}
```

Matemática

Coeficiente binomial

```
/**
 *  @return  Binomial coefficient.
 *  Time complexity: O(N²)/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 || 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 || 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 to_base(ll x, ll b) {
    assert(b > 1);
    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(log(N)))
*/
pair<vll, vll> sieve(ll n) {
    vll ps, spf(n + 1);
    rep(i, 2, n + 1) if (!spf[i]) {
        spf[i] = i;
        ps.eb(i);
        for (ll j = i * 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 rápido

```
/**
 *  @return  Unordered vector with all divisors of x.
 *  Requires pollard rho.
 *  Time complexity: O(faster than sqrt)
*/
vll divisors(ll x) {
    vll fs = factors(x), ds{1};  // use fast factorization (and sort)
    ll prev = 1, sz_prev = 1;
    rep(i, 0, fs.size()) {
        ll f = fs[i];
        if (i > 0 && fs[i] == fs[i - 1])
            prev = f *= prev;
        else prev = fs[i], sz_prev = ds.size();
        rep(j, 0, sz_prev) ds.eb(ds[j] * f);
    }
    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;
}
```

Equações diofantinas

```
/**
 *  @param  a, b, c  Numbers.
 *  @return          (x, y, d) integer solution for aX + bY = c and d is gcd(a, b).
 *  (LLONG_MAX, LLONG_MAX) if no solution is possible.
 *  Time complexity: O(log(N))
*/
tuple<ll, ll, ll> diophantine(ll a, ll b, ll c = 1) {
    if (b == 0) {
        if (c % a != 0) return {LLONG_MAX, LLONG_MAX, a};
        return {c / a, 0, a};
    }
    auto [x, y, d] = diophantine(b, a % b, c);
    if (x == LLONG_MAX) return {x, y, a};
    return {y, x - a / b * y, d};
}
```

Exponenciação rápida

```
/**
 *  @param  a  Number.
 *  @param  b  Exponent.
 *  @return    a^b.
 *  Time complexity: O(log(B))
*/
template <typename T>
T pot(T a, ll b) {
    T res(1);  // T's identity
    while (b) {
        if (b & 1) res = res * a;
        a = a * a, b /= 2;
    }
    return res;
}
```

Fatoração

```cpp
/**
 *  @return  Ordered 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

```cpp
/**
 *  @param  x    Number.
 *  @param  spf  Vector of smallest prime factors
 *  @return      Ordered 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

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

/**
 *  @param  x   Number.
 *  @return     Unordered vector with prime factors of x.
 *  Requires primality test.
 *  Time complexity: O(N^(1/4)log(N))
*/
vll factors(ll x) {
    if (x == 1)     return {};
    if (is_prime(x)) return {x};
    ll d  = rho(x);
    vll l = factors(d), r = factors(x / d);
    l.insert(l.end(), all(r));
    return l;
}
```

Gauss

```cpp
// const double EPS = 1e-9;  // double
const ll EPS = 0;  // mod
#define abs(x) (x).v  // mod

/**
 *  @param  ls  Linear system matrix.
 *  @return     Vector with value of each variable.
 *  Time complexity: O(N³)
*/
template <typename T>
vector<T> gauss(vector<vector<T>>& ls) {
    ll n = ls.size(), m = ls[0].size() - 1;
    vll where(m, -1);
    for (ll col = 0, row = 0; col < m && row < n; ++col) {
        ll sel = row;
        rep(i, row, n) if (abs(ls[i][col]) > abs(ls[sel][col])) sel = i;
        if (abs(ls[sel][col]) <= EPS) continue;
        rep(i, col, m + 1) swap(ls[sel][i], ls[row][i]);
        where[col] = row;
        rep(i, 0, n) if (i != row) {
            T c = ls[i][col] / ls[row][col];
            rep(j, col, m + 1) ls[i][j] -= ls[row][j] * c;
        }
        ++row;
    }
    vector<T> ans(m);
    rep(i, 0, m) if (where[i] != -1) ans[i] = ls[where[i]][m] / ls[where[i]][i];
    rep(i, 0, n) {
        T sum = 0;
        rep(j, 0, m) sum += ans[j] * ls[i][j];
        if (abs(sum - ls[i][m]) > EPS) return {};
    }
    return ans;
}
```

Permutação com repetição

```cpp
/**
 *  @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 rep_perm(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;
}
```

Teorema chinês do resto

```cpp
/**
 *  @param  congruences  Vector of pairs (a, m).
 *  @return              (s, l) (s (mod l) is the answer for the equations)
 *  (LLONG_MAX, LLONG_MAX) if no solution is possible.
 *  s = a0 (mod m0)
 *  s = a1 (mod m1)
 *  ...
 *  congruences vector has pairs (ai, mi).
 *  Requires diophantine equations.
 *  Time complexity: O(Nlog(N))
*/
pll crt(const vpll& congruences) {
    auto [s, l] = congruences[0];
    for (auto [a, m] : congruences) {
        auto [x, y, d] = diophantine(l, -m, a - s);
        if (x == LLONG_MAX) return {x, y};
        s = (a + y % (l / d) * m + l * m / d) % (l * m / d);
        l = l * m / d;
    }
    return {s, l};
}
```

Teste de primalidade

```cpp
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 pot(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;
}

/**
 *  @param  x  Number.
 *  @return    True if x is prime, false otherwise.
 *  Time complexity: O(log²(N))
*/
bool is_prime(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 = pot(a, d, x);
        if (a == 1 || 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;
}
```

Totiente de Euler rápido

```
/**
 *  @return  Euler totient value for x.
 *  Euler totient counts coprimes of x in [1, x].
 *  Requires pollard rho.
 *  Time complexity: O(faster than sqrt)
*/
ll totient(ll x) {
    vll fs = factors(x);  // Pollard rho
    sort(all(fs));
    fs.erase(unique(all(fs)), fs.end());
    ll res = x;
    for (auto f : fs) res = (res / f) * (f - 1);
    return res;
}
```

Transformada de Fourier

```
constexpr ll M = 998244353;  // ntt
constexpr ll G = 3;
// #define T Mi<M>  // ntt
#define T complex<double>  // fft

/**
 *  @brief     Fast fourier transform.
 *  @param  a  Coefficients of polynomial.
 *  Requires modular arithmetic if ntt.
 *  Time complexity: O(Nlog(N))
*/
void fft(vector<T>& a, bool invert) {
    ll n = a.size();
    for (ll i = 1, j = 0; i < n; ++i) {
        ll bit = n >> 1;
        for (; j & bit; bit >>= 1) j ^= bit;
        j ^= bit;
        if (i < j) swap(a[i], a[j]);
    }
    for (ll len = 2; len <= n; len <<= 1) {
        // T wlen = pot<T>(G, (M - 1) / len);  // ntt
        // if (invert) wlen = pot(wlen, M - 2);  // ntt
        double ang = 2 * acos(-1) / len * (invert ? -1 : 1);  // fft
        T wlen(cos(ang), sin(ang));  // fft
        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) / T(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 if ntt.
 *  If normal fft may need to round later: round(.real())
 *  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;
```

```cpp
    while (n < a.size() + b.size()) n <<= 1;
    fa.resize(n), fb.resize(n);
    fft(fa, false), fft(fb, false);
    rep(i, 0, n) fa[i] *= fb[i];
    fft(fa, true);
    return fa;
}
```

Strings

Autômato KMP

```cpp
/**
 *  @param  s  String.
 *  @return    KMP Automaton.
 *  Time complexity: O(26N)
*/
vvll kmp_automaton(const string& s) {
    ll n = s.size();
    vll pi(n);
    vvll aut(n + 1, vll(26));
    rep(i, 0, n + 1) {
        rep(c, 0, 26)
            if (i < n && c == s[i] - 'a') aut[i][c] = i + 1;
            else if (i > 0) aut[i][c] = aut[pi[i - 1]][c];
        if (i > 0 && i < n) pi[i] = aut[pi[i - 1]][s[i] - 'a'];
    }
    return aut;
}
```

Bordas

```cpp
/**
 *  @param  s  String.
 *  @return    Borders.
 *  Time complexity: O(N)
*/
vll borders(const T& s) {
    vll pi = kmp(s), res;
    ll b = pi[s.size() - 1];
    while (b >= 1) res.eb(b), b = pi[b - 1];
    reverse(all(res));
    return res;
}
```

Comparador de substring

```cpp
/**
 *  @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 vvll& 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 || 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};
}
```

KMP

```
/**
 *  @param  s  String.
 *  @return    Vector with the border size for each prefix index.
 *  Time complexity: O(N)
*/
template <typename T>
vll kmp(const T& s) {
    ll n = s.size();
    vll pi(n);
    rep(i, 1, n) {
        ll j = pi[i - 1];
        while (j > 0 && s[i] != s[j])
            j = pi[j - 1];
        pi[i] = j + (s[i] == s[j]);
    }
    return pi;
}
```

Maior prefixo comum (LCP)

```cpp
/**
 *  @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 lcp(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 && j + k < n && s[i + k] == s[j + k])
            ++k;
        lcp[rank[i]] = k;
        if (k) --k;
    }
    return lcp;
}
```

Manacher (substrings palíndromas)

```cpp
/**
 *  @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(const string& s) {
    string t = "#";
    for(char c : s) t += 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 min_rotation(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 && 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 && 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 (FFT)

```
/**
 *  @param  s  String.
 *  @param  t  Substring (can have wildcards '?').
 *  @return    Vector with the first index of occurrences.
 *  Requires FFT.
 *  Time complexity: O(Nlog(N))
*/
vll occur(const string& s, const string& t) {
    ll n = s.size(), m = t.size(), q = 0;
    if (n < m) return {};
    vector<T> a(n), b(m);
    rep(i, 0, n) {
        double ang = acos(-1) * (s[i] - 'a') / 13;
        a[i] = {cos(ang), sin(ang)};
    }
    rep(i, 0, m) {
        if (t[m - i - 1] == '?') ++q;
        else {
            double ang = acos(-1) * (t[m - 1 - i] - 'a') / 13;
            b[i] = {cos(ang), -sin(ang)};
        }
    }
    auto c = convolution(a, b);
    vll res;
    rep(i, 0, n)
        if (abs(c[m - 1 + i].real() - (double)(m - q)) < 1e-3)
            res.eb(i);
    return res;
}
```

Quantidade de ocorrências de substring

```cpp
/**
 *  @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;
}
```

Suffix array

```cpp
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 upd_eq_class(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 suffix_array(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);
    upd_eq_class(cs, ps, s);
    for (ll mask = 1; mask < n && 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]];
        upd_eq_class(cs, ps, ys);
    }
    ps.erase(ps.begin());
    return (k == 0 ? cs : ps);
}
```

Estruturas

Árvores

BIT 2D

```
template <typename T>
struct BIT2D {
    ll n, m;
    vector<vector<T>> bit;

    /**
     *  @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 && y <= n && 0 < x && 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 && ly <= hy && hy <= n && 0 < lx && lx <= hx && hx <= m);
        return sum(hy, hx) - sum(hy, lx - 1) - sum(ly - 1, hx) + sum(ly - 1, lx - 1);
    }
};
```

DSU

```
struct DSU {
    vll parent, size;

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

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

    /**
     *  @param  x, y  Elements.
     *  @return       True if merged, false if already same set.
     *  Time complexity: ~O(1)
    */
    bool merge(ll x, ll y) {
        x = find(x), y = find(y);
        if (x == y) return false;
        if (size[x] > size[y]) swap(x, y);
        parent[x] = y;
        size[y] += size[x], size[x] = 0;
        return true;
    }

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

Heavy-light decomposition

```cpp
template <typename T, typename Op = function<T(T, T)>>
struct HLD {
    Segtree<T> seg;
    Op op;
    bool values_on_edges;
    vll idx, subtree, parent, head;
    ll timer = 0;

    /**
    *  @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 upd_qry_path(u, u) if values on vertex or upd_qry_path(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), values_on_edges(values_on_edges) {
        idx = subtree = parent = head = vll(g.size());
        auto build = [&](auto&& self, ll u = 1, ll p = -1) -> 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]] || 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 an upd).
    *  @return       f of path [u, v] (if it's a qry).
    *  It's a query if x is specified.
    *  Time complexity: O(log²(N))
    */
    ll upd_qry_path(ll u, ll v, ll x = INT_MIN) {
        assert(1 <= u && u < idx.size() && 1 <= v && v < idx.size());
        if (idx[u] < idx[v]) swap(u, v);
        if (head[u] == head[v]) return seg.upd_qry(idx[v] + values_on_edges, idx[u], x);
        return op(seg.upd_qry(idx[head[u]], idx[u], x),
                  upd_qry_path(parent[head[u]], v, x));
    }

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

    /**
    *  @param  u, v  Vertices.
    *  @return       Lowest common ancestor between u and v.
    *  Time complexity: O(log(N))
    */
    ll lca(ll u, ll v) {
        for (; head[u] != head[v]; u = parent[head[u]])
            if (idx[u] < idx[v]) swap(u, v);
        return idx[u] < idx[v] ? u : v;
    }
};
```

Ordered Set

```cpp
#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

```cpp
template <typename T, typename Op = function<T(T, T)>>
struct Segtree {
    ll n;
    T DEF;
    vector<T> seg, lzy;
    Op op;

    /**
    *  @param  sz   Size.
    *  @param  def  Default value.
    *  @param  f    Combine 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) : n(sz), DEF(def), seg(4 * n, DEF), lzy(4 * n), 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 upd).
    *  @return       f of interval [i, j] (if it's a qry).
    *  It's a query if x is specified.
    *  Time complexity: O(log(N))
    */
    T upd_qry(ll i, ll j, T x = LLONG_MIN, ll l = 0, ll r = -1, ll no = 1) {
        assert(0 <= i && i <= j && j < n);
        if (r == -1) r = n - 1;
        if (lzy[no]) unlazy(l, r, no);
        if (j < l || i > r) return DEF;
        if (i <= l && r <= j) {
            if (x != LLONG_MIN) {
                lzy[no] += x;
                unlazy(l, r, no);
            }
            return seg[no];
        }
        ll m = (l + r) / 2;
        T q = op(upd_qry(i, j, x, l, m, 2 * no),
                 upd_qry(i, j, x, m + 1, r, 2 * no + 1));
        seg[no] = op(seg[2 * no], seg[2 * no + 1]);
        return q;
    }

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;
    }
};
```

Primeiro maior

```cpp
/**
*  @param  i, j  Interval;
*  @param  x     Value to compare.
*  @return       First index from i->j 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 first_greater(ll i, ll j, T x, ll l = 0, ll r = -1, ll no = 1) {
    assert(0 <= i && i <= j && j < n);
    if (r == -1) r = n - 1;
    if (j < l || i > r || seg[no] <= x) return -1;
    if (l == r) return l;
    ll m = (l + r) / 2;
    ll left = first_greater(i, j, x, l, m, 2 * no);
    if (left != -1) return left;
    return first_greater(i, j, x, m + 1, r, 2 * no + 1);
}
```

SegTree Lazy (Affine)

```
// fora da seg
template <typename T>
struct Affine {
    T mul = 1, add = 0;
    Affine operator*=(const Affine& other) {
        return *this = {mul * other.mul, add * other.mul + other.add};
    }
    operator bool() { return mul != 1 || add != 0; }
};

// dentro da seg
vector<Affine<T>> lzy;

// dentro do upd_qry
Affine<T> x = Affine<T>()

// troca todos os += por *=

// dentro do unlazy
seg[no] = seg[no] * lzy[no].mul + (r - l + 1) * lzy[no].add;
lzy[no] = Affine<T>();  // em vez de = 0
```

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 || !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 || !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);
}
```

```cpp
/**
*  @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

```cpp
struct WaveletTree {
    ll n;
    vvll wav;

    /**
    *  @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) : n(sz), wav(2 * n) {
        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 && i <= j && j < (ll)wav[1].size() && 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 && i <= j && j < (ll)wav[1].size() && 0 <= x && 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;
    }
};
```

Geometria

Círculo

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

template <typename T>
struct Circle {
    pair<T, T> C;
    T r;

    /**
    *  @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 || d < abs(c.r - r) || (equals(d, 0) && 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> tan_points() {
        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;
    }
};
```

Polígono

```
template <typename T>
struct Polygon {
    vector<pair<T, T>> vs;
    ll n;

    /**
    *  @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 == 0 || N == 0;
    }

    /**
    *  @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;
    }

    /**
```

```cpp
    *  @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;
        bool in = false;
        rep(i, 0, n) {
            if (((vs[i].y > P.y) != (vs[i + 1].y > P.y)) &&
                (P.x < (vs[i + 1].x - vs[i].x) * (P.y - vs[i].y) / (double)(vs[i + 1].y -
vs[i].y) + vs[i].x))
                in = !in;
        }
        return in;
    }

    /**
    *  @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)};
    }
};
```

Reta

```cpp
/**
*  A line with normalized coefficients.
*/
template <typename T>
struct Line {
    T a, b, c;

    /**
    *  @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>) {
            if (abs(a) > abs(b)) b /= a, c /= a, a = 1;
            else a /= b, c /= b, b = 1;
        }
        else {
            if (a < 0 || (a == 0 && 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) && equals(b, r.b) && equals(c, r.c);
    }
};
```

Segmento

```cpp
template <typename T>
struct Segment {
    pair<T, T> A, B;

    /**
    *  @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) && contains(r.A)) || (equals(d2, 0) && contains(r.B)))
            return true;

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

        return (d1 * d2 < 0) && (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;
    }
};
```

Triângulo

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

template <typename T>
struct Triangle {
    pair<T, T> A, B, C;
    T a, b, c;

    /**
    *  @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 class_by_sides() {
        if (equals(a, b) && equals(b, c)) return EQUILATERAL;
        if (equals(a, b) || equals(a, c) || equals(b, c)) return ISOSCELES;
        return SCALENE;
    }

    /**
    *  @return  Angle class.
    *  Time complexity: O(1)
    */
    Angles class_by_angles() {
        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};
    }
};
```

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) {}
    Matrix(ll n, ll m, ll x = 0) : mat(n, vector<T>(m)) {
        if (n == m) rep(i, 0, n) mat[i][i] = x;
    }
    vector<T>& operator[](ll i) { return mat[i]; }
    ll size() const { return mat.size(); }

    /**
     *  @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(N³)
    */
    Matrix operator*(const Matrix& other) const {
        ll N = mat.size(), K = mat[0].size(), M = other[0].size();
        assert(other.size() == K);
        Matrix res(N, M);
        rep(i, 0, N) rep(k, 0, K) rep(j, 0, M)
            res[i][j] += mat[i][k] * other[k][j];
        return res;
    }

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

Strings

Aho-Corasick

```
struct AhoCorasick {
    static constexpr ll MAXN = 5e5 + 1;
    ll n, m;
    vvll to, go, idx;
    vll mark, qnt, p, pc, link, exit;

    AhoCorasick() : n(0), m(0), to(MAXN, vll(26)), go(MAXN, vll(26, -1)), idx(MAXN),
mark(MAXN),
                    qnt(MAXN), p(MAXN), pc(MAXN), link(MAXN, -1), exit(MAXN, -1) {}

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

    /**
     *  @param  s  String.
     *  Time complexity: O(N + Matches)
     */
    vpll occur(const string& s) {
        vll occ(n + 1);
        vpll res;
        ll u = 0;
        for (char ch : s) {
            u = go_to(u, ch - 'a');
            for (ll v = u; v != 0; v = get_exit(v)) ++occ[v];
        }
        rep(v, 0, n + 1) for (auto i : idx[v])
            if (occ[v])
                res.eb(i, occ[v]);
        return res;
    }

    ll get_link(ll u) {
        if (link[u] != -1) return link[u];
        if (u == 0 || p[u] == 0) return link[u] = 0;
```

```cpp
        return link[u] = go_to(get_link(p[u]), pc[u]);
    }

    ll go_to(ll u, ll c) {
        if (go[u][c] != -1) return go[u][c];
        if (to[u][c]) return go[u][c] = to[u][c];
        return go[u][c] = u == 0 ? 0 : go_to(get_link(u), c);
    }

    ll get_exit(ll u) {
        ll v = get_link(u);
        if (exit[u] != -1) return exit[u];
        return exit[u] = (v == 0 || mark[v]) ? v : get_exit(v);
    }
};
```

Hash

```cpp
constexpr ll M1 = (ll)1e9 + 7, M2 = (ll)1e9 + 9;
#define H pll
#define x first
#define y second
mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
H P(
    uniform_int_distribution<ll>(256, 1e9)(rng),
    uniform_int_distribution<ll>(256, 1e9)(rng)
);
ll sum(ll a, ll b, ll m) { return (a += b) >= m ? a - m : a; };
ll sub(ll a, ll b, ll m) { return (a -= b) < 0 ? 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)}; }
template <typename T>
struct Hash {
    ll n;
    // Segtree<H> ps;
    vector<H> ps, pw;

    /**
     *  @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 T& s) : n(s.size()), ps(n + 1), pw(n + 1) {
        pw[0] = {1, 1};
        rep(i, 0, n) {
            ps[i + 1] = ps[i] * P + H(s[i], s[i]);
            pw[i + 1] = pw[i] * P;
        }
        // vector<H> ps_(n);
        // 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) {
    //     ps.upd_qry(i, i, pw[n - i - 1] * H(c, c));
    // }

    /**
     *  @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 && i <= j && j < n);
        return ps[j + 1] - ps[i] * pw[j + 1 - i];
        // return ps.upd_qry(i, j) * pw[i];
    }
};
```

Hash Inverso

```
template <typename T>
struct HashInv {
    Hash<T> h;
    HashInv(T s) : h("") { reverse(all(s)), h = Hash<T>(s); }
    H operator()(ll i, ll j) { return h(h.n - j - 1, h.n - i - 1); }
};
```

Suffix Automaton

```
struct SuffixAutomaton {
    vvll to;  // if mle change to vector<map<ll, ll>>
    vll len, fpos, lnk, cnt, rcnt, dcnt;
    ll sz = 0, last = 0, n = 0, alpha = 26;

    SuffixAutomaton() { make_node(); }

    /**
    *  @param  s  String.
    *  Time complexity: O(N)
    */
    void insert(const string& s) {
        last = 0;
        for (auto c : s) add(c - 'a');
    }

    void finalize() {
        vll order(sz - 1);
        iota(all(order), 1);
        sort(all(order), [&](ll a, ll b) { return len[a] > len[b]; });
        for (ll i : order) cnt[lnk[i]] += cnt[i];

        // preprocessing for kth_sub and kth_dsub
        dfs(0);
    }

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

    /**
    *  @return  Vector with amount of distinct substrings of each size.
    *  Time complexity: O(N)
    */
    vll dsubs_by_size() {
        vll delta(n);
```

```cpp
    rep(i, 1, sz) {
        ll mnlen = len[lnk[i]];
        ++delta[mnlen];
        if (len[i] < n) --delta[len[i]];
    }
    rep(i, 1, n) delta[i] += delta[i - 1];
    return delta;
}

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

```cpp
            }
        }
        return res;
    }

private:
    ll make_node(ll LEN = 0, ll FPOS = -1, ll LNK = -1, ll CNT = 0) {
        to.eb(vll(alpha)), rcnt.eb(0), dcnt.eb(0);
        len.eb(LEN), fpos.eb(FPOS), lnk.eb(LNK), cnt.eb(CNT);
        return sz++;
    }

    void add(ll c) {
        ++n;
        ll p = last;  // if strings have weight cnt = 0, add it later
        ll u = make_node(len[p] + 1, len[p], 0, 1);
        last = u;
        for (; p != -1 && !to[p][c]; p = lnk[p]) to[p][c] = u;
        if (p == -1) lnk[u] = 0;
        else {
            ll v = to[p][c];
            if (len[p] + 1 == len[v]) lnk[u] = v;
            else {
                ll clone = make_node(len[p] + 1, fpos[v], lnk[v]);
                to[clone] = to[v];
                lnk[u] = lnk[v] = clone;
                for (; p != -1 && to[p][c] == v; p = lnk[p])
                    to[p][c] = clone;
            }
        }
    }

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

Trie

```
// empty head, tree is made by the prefixs of each string in it.
struct Trie {
    static constexpr ll MAXN = 5e5;
    ll n;
    vvll to;  // 0 is head
    // term: quantity of strings that ends in this node.
    // qnt: quantity of strings that pass through this node.
    vll term, qnt;

    Trie() : n(0), to(MAXN + 1, vll(26)), term(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];
        }
        ++term[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;
        }
        --term[u], --qnt[0];
    }

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

Bit Trie

```cpp
// empty head, tree is made by the prefixs of each string in it.
struct BitTrie {
    static constexpr ll MAXN = 5e6;
    ll n;
    vvll to;  // 0 is head
    // qnt: quantity of strings that pass through this node.
    vll qnt;

    BitTrie() : n(0), to(MAXN + 1, vll(2)), qnt(MAXN + 1) {}

    /**
    *  @param  s  String.
    *  Time complexity: O(N)
    */
    void insert(ll x) {
        ll u = 0;
        rep(i, 0, 32) {
            ll& v = to[u][(x >> i) & 1];
            if (!v) v = ++n;
            u = v, ++qnt[u];
        }
        ++qnt[0];
    }

    /**
    *  @param  s  String.
    *  Time complexity: O(N)
    */
    void erase(ll x) {
        ll u = 0;
        rep(i, 0, 32) {
            ll& v = to[u][(x >> i) & 1];
            u = v, --qnt[u];
            if (!qnt[u]) v = 0;
        }
        --qnt[0];
    }

    ll mx_or(ll x) {
        ll u = 0;
        ll res = 0;
        rep(i, 0, 32) {
            ll desired = !((x >> i) & 1);
            ll v = to[u][desired];
            if (v) {
                u = v;
                res |= desired << i;
            }
            else {
                u = to[u][!desired];
                res |= (!desired) << i;
            }
        }
        return res;
    }
};
```

Outros

Compressão

```cpp
template <typename T>
struct Compressed {
    Compressed(const vector<T>& xs) : cs(xs) {
        sort(all(cs));
        cs.erase(unique(all(cs)), cs.end());
    }
    ll operator[](T x) { return lower_bound(all(cs), x) - cs.begin(); }
    T ret(ll c) { return cs[c]; }
    ll size() { return cs.size(); }
    vector<T> cs;
};
```

Delta encoding

```
struct Delta {
    vll xs;

    Delta(ll n) : xs(n + 1) {}

    /**
    *  @brief        Adds x to each element in interval [i, j].
    *  @param  i, j  Interval.
    *  @param  x     Value to add.
    *  Time complexity: O(1)
    */
    void upd(ll l, ll r, ll x) { xs[l] += x, xs[r + 1] -= x; }

    /**
    *  @return  Vector after operations.
    *  Time complexity: O(N)
    */
    vll get() {
        vll res(xs.size() - 1);
        res[0] = xs[0];
        rep(i, 1, res.size()) res[i] = res[i - 1] + xs[i];
        return res;
    }
};
```

Fila agregada

```
template <typename T, typename Op = function<T(T, T)>>
struct AggQueue {
    stack<pair<T, T>> in, out;
    Op op;

    /**
    *  @param  f  Function (without inverse, like max(), min(), or, and, gcd...)
    *  Time complexity: ~O(1)
    */
    AggQueue(Op f) : op(f) {}

    /**
    *  @return  f of all elements in queue.
    *  Time complexity: ~O(1)
    */
    T query() {
        if (in.empty()) return out.top().y;
        if (out.empty()) return in.top().y;
        return op(in.top().y, out.top().y);
    }

    /**
    *  @brief     Inserts x in queue.
    *  @param  x  Value to insert.
    *  Time complexity: ~O(1)
    */
    void insert(T x, bool is_user_insertion = true) {
        auto& st = is_user_insertion ? in : out;
        T cur = st.empty() ? x : op(st.top().y, x);
        st.emplace(x, cur);
    }

    /**
    *  @brief  Deletes first element in queue.
    *  Time complexity: ~O(1)
    */
    void pop() {
        if (out.empty())
            while (!in.empty()) {
                insert(in.top().x, false);
                in.pop();
            }
        out.pop();
    }
};
```

Mex

```cpp
struct Mex {
    vll hist;
    set<ll> missing;

    /**
    *  @param  n  Size (max element).
    *  Time complexity: O(Nlog(N))
    */
    Mex(ll n) : hist(n) { rep(i, 0, n + 1) missing.emplace(i); }

    /**
    *  @returns  Mex of current elements.
    *  Time complexity: O(1)
    */
    ll mex() { return *missing.begin(); }

    /**
    *  @param  x  Value to insert.
    *  Time complexity: O(log(N))
    */
    void insert(ll x) { if (x < hist.size() && ++hist[x] == 1) missing.erase(x); }

    /**
    *  @param  x  Value to erase.
    *  Time complexity: O(log(N))
    */
    void erase(ll x) { if (x < hist.size() && --hist[x] == 0) missing.emplace(x); }
};
```

Venice set

```cpp
template<typename T>
struct VeniceSet {
    map<T, ll> xs;
    T delta = 0;

    void emplace(T x) { xs[x + delta]++; }
    void add_all(T x) { delta -= x; }

    ll count(T x) {
        if (xs.find(x + delta) != xs.end())
            return xs[x + delta];
        return 0;
    }

    T pop() {
        auto m = xs.begin()->first - delta;
        if (--xs[m] == 0) xs.erase(m);
        return m;
    }
};
```

Utils

Aritmética modular

```cpp
constexpr ll M1 = (ll)1e9 + 7;
constexpr ll M2 = (ll)998244353;
template <ll M = M1>
struct Mi {
    ll v;
    Mi(ll x = 0) : v(x) {
        if (v >= M || v < -M) v %= M;
        if (v < 0) v += M;
    }
    explicit operator ll() const { return v; }
    Mi& operator+=(Mi o) { if ((v += o.v) >= M) v -= M; return *this; }
    Mi& operator-=(Mi o) { if ((v -= o.v) < 0) v += M; return *this; }
    Mi& operator*=(Mi o) { v = v * o.v % M; return *this; }
    Mi& operator/=(Mi o) { return *this *= pot(o, 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; }
    friend ostream& operator<<(ostream& os, Mi a) { return os << a.v; }
};
```

Bits

```
ll msb(ll x) { return (x == 0 ? 0 : 63 - __builtin_clzll(x)); }
ll lsb(ll x) { return __builtin_ffsll(x); }
```

Ceil division

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

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-7; }
```

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:*** $gcd(abs(P.x - Q.x), abs(P.y - Q.y)) + 1$. ***$P, Q$ 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,*** $A = I + \frac{B}{2} - 1$ ***e*** $I = \frac{2A+2-B}{2}$.

> ***Distância de Chebyshev:*** $dist(P, Q) = max(P.x - Q.x, P.y - Q.y)$. ***$P, Q$ são dois pontos.***

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

> ***Para contar paralelogramos em um conjunto de pontos podemos marcar o centro de cada segmento, coincidências entre dois centros formam um paralelogramo.***

Matemática

> ***Quantidade de divisores de um número:*** $\prod(a_i + 1)$. ***a_i é o expoente do i-ésimo fator primo.***

> ***Soma dos divisores de um número:*** $\prod \frac{p_i^{a_i+1}-1}{p_i-1}$. ***$a_i$ é o expoente do i-ésimo fator primo p_i.***

> ***Produto dos divisores de um número x:*** $x^{\frac{qd(x)}{2}}$. ***$qd(x)$ é a quantidade de divisores dele.***

> ***Maior quantidade de divisores de um número:*** $< 10^3$ ***é*** 32; $< 10^6$ ***é*** 240; $< 10^9$ ***é*** 1344; $< 10^{18}$ ***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 diferença entre dois primos consecutivos:*** $< 10^7$ ***é*** 180.

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

> ***Números primos interessantes:*** $2^{31} - 1; 2^{31} + 11; 10^{16} + 61; 10^{18} - 11; 10^{18} + 3$.

> $gcd(a, b) = gcd(a, a - b)$, $gcd(a, b, c) = gcd(a, a - b, a - c)$, ***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 ($\sum_{k=1}^{n} x^k$ é uma progressão geométrica):*** $\frac{a_n * r - a_1}{r-1}$.

> ***Soma de progressão geométrica da forma:*** $\sum_{k=1}^{n} kx^k = \frac{x(1-x^n)}{(1-x)^2} - \frac{nx^{n+1}}{1-x}$, ***parte da fórmula normal, multiplica por x e subtrai.***

> ***Soma de termos ao quadrado:*** $1^2 + 2^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$.

> ***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.***

> ***Teorema de Euler:*** $a^{\varphi(m)} = 1 \bmod m$. ***Se m é primo se reduz á*** $a^{m-1} = 1 \bmod m$.

> $a^{b^c} \bmod m = a^{b^c \bmod \varphi(m)} \bmod m$. ***Se m é primo se reduz á*** $a^{b^c} \bmod m = a^{b^c \bmod (m-1)} \bmod m$.

> $a^{\varphi(m)-1} = a^{-1} \bmod m$, ***mas precisa da condição que*** $gcd(a, m) = 1$.

> ***Teorema de Wilson:*** $(n-1)! = -1 \bmod n$. ***Dá para calcular*** $x! \bmod n$ ***se*** $n - x \le 10^6$ ***pois*** $x! = -[(x+1)\ldots(m-1)]^{-1} \bmod n$.

> $(a+b)^n = \binom{n}{0}a^n + \binom{n}{1}a^{n-1}b + \binom{n}{2}a^{n-2}b^2 + \cdots + \binom{n}{k}a^{n-k}b^k + \cdots + \binom{n}{n}b^n$

> ***Soma da n-ésima linha do triângulo de Pascal:*** 2^n.

> ***Soma da m-ésima coluna do triângulo de Pascal:*** $\binom{n+1}{m+1}$. ***n é a quantidade de linhas.***

> ***A quantidade de ímpares na n-ésima linha do triângulo de Pascal:*** 2^c. ***c é a quantidade de bits ligados na representação binária de n.***

> ***Números de Catalan C_n: representa a quantidade de expressões válidas com parênteses de tamanho $2n$. Também são relacionados às árvores, existem C_n árvores binárias de n vértices e $C_n - 1$ árvores de n vértices (as árvores são caracterizadas por sua aparência).*** $C_n = \frac{\binom{2n}{n}}{n+1}$.

> ***Lema de Burnside: o número de combinações em que simétricos são considerados iguais é o somatório*** $\frac{1}{n}\sum_{k=1}^{n} c(k)$. ***$n$ é a quantidade de maneiras de mudar a posição de uma combinação e $c(k)$ é a quantidade de combinações que são consideradas iguais à primeira maneira na k-ésima maneira.***

> $min(f, g) = \frac{f+g}{2} - \frac{|f-g|}{2}$. ***Se estamos fazendo para vários valores, para lidar com o módulo, se temos f fixo, tentamos lidar com os g maiores que f e com os menores que f separadamente.***

Strings

> ***Sejam p e q dois períodos de uma string s. Se $p + q - mdc(p, q) \leq |s|$, então $mdc(p, q)$ também é período de s.***

> ***Relação entre bordas e períodos: A sequência $|s| - |border(s)|, |s| - |border^2(s)|, \ldots, |s| - |border^k(s)|$ 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) { ... })`.

> ***Por padrão python faz operações com até*** `4000` ***dígitos, para aumentar:*** `import sys`
> `sys.set_int_max_str_digits(1000001)`

> ***Dado um grafo a quantidade de vértices pesados é $\sqrt{N}$, vértices leves são aqueles com degrau $\leq \sqrt{N}$.***