Compilation

```
FLAGS=-g -Wall -Wextra -Wshadow -Wno-unused-result -D_GLIBCXX_DEBUG -fsanitize=address -fsanitize=undefined
↪  -fno-sanitize-recover
run_a run_A: clean
	@g++ A.cpp \$(FLAGS) -DJUNCO_DEBUG && ./a.out < z.in
```

Template

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
ios::sync_with_stdio(false); cin.tie(nullptr); cout.tie(nullptr);
cout.precision(20); cout << fixed << ans;
bool is_set(ll x, ll i) {return (x>>i)&1;}
void set_bit(ll &x, ll i) {x |= 1ll<<i;}
```

```
ll LSB(ll x) {return x & (-x);}
void unset_bit(ll &x, ll i) {x = (x | (1ll<<i)) ^ (1ll<<i);}
stringstream ss;
ss << "Hello world";
while(ss >> s) cout << s << endl;
ss.clear();
```

Bitmask

```
// Iterate over all submasks of a mask. CONSIDER SUBMASK = 0/APART
for(submask = mask; submask > 0; submask = (submask-1)&mask) {
}
// With OR and AND you can get XOR.
// (a^b) = (a|b) - (a&b).
```

```
// (a^b) = (a|b) ^ (a&b).
// (a+b) = (a|b) + (a&b).
// (a+b) = (a^b) + 2*(a&b).
// Two complement -x = ~x + 1.
// a^b belongs to [a-b, a+b] and [b-a, a+b].
```

Point

```
template<typename T>
class Point {
    public:
    static const int LEFT_TURN = 1;
    static const int RIGHT_TURN = -1;
    T x = 0, y = 0;
    Point() = default;
    Point(T _x, T _y) {
        x = _x;
        y = _y;
    }
    friend ostream &operator << (ostream &os, Point<T> p) {
        os << "(" << p.x << ", " << p.y << ")";
        return os;
    }
    bool operator == (const Point<T> other) const {
        return x == other.x && y == other.y;
    }
    // Get the (1º) bottom (2º) left point.
    bool operator < (const Point<T> other) const {
        if(y != other.y) return y < other.y;
        return x < other.x;
    }
    T euclidean_distance(Point<T> other) {
        T dx = x - other.x;
        T dy = y - other.y;
        return sqrt(dx*dx + dy*dy);
    }
    T euclidean_distance_squared(Point<T> other) {
        T dx = x - other.x;
        T dy = y - other.y;
        return dx*dx + dy*dy;
    }
    T manhatan_distance(Point<T> other) {
        return abs(other.x - x) + abs(other.y - y);
    }
    // Get the height of the triangle with base b1, b2.
    T height_triangle(Point<T> b1, Point<T> b2) {
        if(b1 == b2 || *this == b1 || *this == b2) return 0; // It's not a triangle.
        T a = euclidean_distance(b1);
        T b = b1.euclidean_distance(b2);
        T c = euclidean_distance(b2);
        T d = (c*c-b*b-a*a)/(2*b);
```

```
        return sqrt(a*a - d*d);
    }
    int get_quadrant() {
        if(x > 0 && y >= 0) return 1;
        if(x <= 0 && y > 0) return 2;
        if(x < 0 && y <= 0) return 3;
        if(x >= 0 && y < 0) return 4;
        return 0; // Point (0, 0).
    }
    // Relative quadrant respect the point other, not the orig
    int get_relative_quadrant(Point<T> other) {
        Point<T> p(other.x - x, other.y - y);
        return p.get_quadrant();
    }
    // Orientation of points *this -> a -> b.
    int get_orientation(Point<T> a, Point<T> b) {
        T prod = (a.x - x)*(b.y - a.y) - (a.y - y)*(b.x - a.x)
        if(prod == 0) return 0;
        return prod > 0? LEFT_TURN : RIGHT_TURN;
    }
    // True if a have less angle than b, if *this->a->b is a l
    bool angle_cmp(Point<T> a, Point<T> b) {
        if(get_relative_quadrant(a) != get_relative_quadrant(b
            return get_relative_quadrant(a) < get_relative_qua
        int ori = get_orientation(a, b);
        if(ori == 0) return euclidean_distance_squared(a) < eu
        return ori == LEFT_TURN;
    }
    // Anticlockwise sort starting at 1º quadrant, respect to
    void polar_sort(vector<Point<T>> &v) {
        sort(v.begin(), v.end(), [&](Point<T> a, Point<T> b)
        {return angle_cmp(a, b);});
    }
    // Convert v to its convex hull, Do a Graham Scan. O(n log
    void convert_convex_hull(vector<Point<T>> &v) {
        if((int)v.size() < 3) return;
        Point<T> bottom_left = v[0], p2;
        for(auto p : v) bottom_left = min(bottom_left, p);
        bottom_left.polar_sort(v);
        vector<Point<T>> v_input = v; v.clear();
        for(auto p : v_input) {
            while(v.size() >= 2) {
                p2 = v.back(); v.pop_back();
                if(v.back().get_orientation(p2, p) == LEFT_TURN) {
                    v.pb(p2);
                    break;
                }
            }
            v.pb(p);
        }
    }
    // Constraint: The points have to be in order p0 -> p1 -> ... -> pn -> p0.
    static ld get_area_polygon(vector<Point<T>> &v) {
        if(v.size() < 3) return 0;
        ll i, sum = 0, n = v.size();
        for(i = 0; i < n; i++) {
            sum += v[i].x*v[(i+1)%n].y - v[(i+1)%n].x*v[i].y;
        }
        return abs(sum)/2.0;
    }
    // Rotate p alpha radians (anti clock wise) respect to this point.
    Point<T> rotate(Point<T> p, ld alpha) {
        Point<T> q(p.x - x, p.y - y); // p shifted.
        return Point<T>(x + q.x*cos(alpha) - q.y*sin(alpha),
                        y + q.x*sin(alpha) + q.y*cos(alpha));
    }
};
template<typename T>
class Line{
    public:
    bool is_vertical = false; // If vertical, line := x = n.
    T m = 0, n = 0; // y = mx + n.
    Line(T _m, T _n) {m = _m; n = _n;}
    Line(Point<T> p1, Point<T> p2) {
        if(p1.x == p2.x) {is_vertical = true; n = p1.x; return;}
        m = (p2.y - p1.y)/(p2.x - p1.x);
        n = m*-p1.x + p1.y;
    }
    friend ostream &operator << (ostream &os, Line<T> l) {
        if(l.is_vertical) os << "x = " << l.n;
        else os << "y = " << l.m << "x + " << l.n;
        return os;
    }
    Point<T> intersection(Line<T> l) {
        if(is_vertical && l.is_vertical) return Point<T>(-inf,
        if(is_vertical) return Point<T>(n, l.m*n + l.n);
        if(l.is_vertical) return Point<T>(l.n, m*l.n + n);
        T new_m = (l.n-n)/(m-l.m);
        return Point<T>(new_m, m*new_m + n);
    }
};
// Point of intersection of two lines formed by (p1, p2), (p3,
Point<ld> intersection4points(Point<ld> p1, Point<ld> p2, Poin
    Line<ld> l1(p1, p2), l2(p3, p4);
    return l1.intersection(l2);
}
// Return random float in [0, 1].
ld rand_float() {
    return rand()/(ld)RAND_MAX;
} // Return a point inside a triangle formed by p1, p2, p3.
Point<ld> random_triangle(Point<ld> p1, Point<ld> p2, Point<ld
    ld u1 = rand_float(), u2 = rand_float();
    if(u1 + u2 > 1) u1 = 1 - u1, u2 = 1 - u2; // rectangle ->
    return Point<ld>(p2.x + (p1.x-p2.x)*u1 + (p3.x-p2.x)*u2,
                     p2.y + (p1.y-p2.y)*u1 + (p3.y-p2.y)*u2);
}
```

Articulation Point

```
// v is an AP if removing v from graph it split into more than one component
// 1- v is the root and v has > 1 child in the DFS.
// 2- v is not the root and has one child u that dont have any back edge.
vector<vi> graph;
class ArticulationPoint{
    int n;
    vi low; // Minimum discover time using a back edge.
    vi discover; // Discover DFS time.
    vi parent;
    int Time = 0;
    void dfs(int u) { // Call dfs(root).
        if(discover[u] != -1) return;
        low[u] = discover[u] = Time++;
        int children = 0;
        for(auto v : graph[u]) {
            if(discover[v] == -1) {
                children++;
                parent[v] = u;
                dfs(v);
                low[u] = min(low[u], low[v]);
                // Every time you set AP[u] = true, the number of components after
                // removing the nodes u or v from the graph increase
                if(parent[u] == -1 && children > 1)
                AP[u] = true;
                if(parent[u] != -1 && low[v] >= discover[u])
                AP[u] = true;
                //if(low[v] > discover[u])
                // {} // edge u->v is a bridge.
            }
            if(v != parent[u]) low[u] =
            min(low[u], discover[v]); // Back edge.
        }
    }
    public:
    vector<bool> AP; // True iff i is an Articulation Point.
    ArticulationPoint() {
        n = graph.size();
        low.assign(n, -1);
        discover.assign(n, -1);
        parent.assign(n, -1);
        AP.assign(n, false);
    }
    void get_AP() {
        for(int i = 0; i < n; i++)
            if(discover[i] == -1) dfs(i);
    }
};
```

Cycle detection

```
ll f(ll x) {
    return (x + 1) % 4; // Example.
}
// mu is the first index of the node in the cycle.
// lambda is the length of the cycle.
pll floyd_cycle_detection(ll x0) {
    ll tortoise = f(x0), hare = f(f(x0)), mu = 0, lambda = 1;
    while(tortoise != hare) tortoise = f(tortoise), hare = f(f(hare));
    tortoise = x0;
    while(tortoise != hare)
    tortoise = f(tortoise), hare = f(hare), mu++;
    hare = f(hare);
    while(tortoise != hare)
    hare = f(hare), lambda++;
    return mp(mu, lambda);
}
```

Toposort

a

```
vector<vi> graph;
class Toposort{
    vector<bool> visited;
    void topo_rec(int u) {
        if(visited[u]) return;
        visited[u] = true;
        for(auto _v : graph[u]) topo_rec(_v);
        vSorted.pb(u);
    }
    public:
    vi vSorted;
    Toposort(int n) {
        visited.assign(n, false);
        for(int i = 0; i < n; i++) topo_rec(i);
        reverse(vSorted.begin(), vSorted.end());
    }
};
```

Flow Dinic

```
class Edge {
    public:
    int u, v;
    int cap, flow = 0; // Capacity and current flow.
    Edge(int _u, int _v, int _cap) : u(_u), v(_v), cap(_cap) { }
};
// O(V^2*E). For unit edge capacity O(sqrt(V)*E).
class Dinic{
    vector<Edge> edge;
    vector<vi> graph;
    int n, n_edges = 0;
    int source, sink, inf_flow = INT_MAX;
    vi lvl; // lvl of the node to the source.
    vi ptr;
    queue<ll> q;
    bool BFS() {
        while(!q.empty()) {
            int u = q.front(); q.pop();
            for(auto el : graph[u]) {
                if(lvl[edge[el].v] != -1) continue;
                if(edge[el].cap - edge[el].flow <= 0) continue;
                lvl[edge[el].v] = lvl[edge[el].u] + 1;
                q.push(edge[el].v);
            }
        }
        return lvl[sink] != -1;
    }
    int dfs(int u, int min_flow) {
        if(u == sink) return min_flow;
        int pushed, el;
        for(;ptr[u] < (int)graph[u].size(); ptr[u]++) {
        //if you can pick ok, else you crop that
            el = graph[u][ptr[u]];
            if(lvl[edge[el].v] != lvl[edge[el].u] + 1
            || edge[el].cap - edge[el].flow <= 0) {
                continue;
            }
            pushed = dfs(edge[el].v, min(min_flow,
            edge[el].cap - edge[el].flow));
            if(pushed > 0) {
                edge[el].flow += pushed;
                edge[el^1].flow -= pushed;
                return pushed;
            }
        }
        return 0;
    }
    public:
    Dinic(int _n, int _source, int _sink) :
    n(_n), source(_source), sink(_sink) {
        graph.assign(_n, vi());
    }
    void add_edge(int u, int v, int flow) { // Add u->v edge.
        Edge uv(u, v, flow), vu(v, u, 0); // Not multiedge.
        edge.pb(uv);
        edge.pb(vu);
        graph[u].pb(n_edges);
        graph[v].pb(n_edges+1);
        n_edges += 2;
    }
    int max_flow() { // It consumes the graph.
        int flow = 0, pushed;
        while(true) {
            lvl.assign(n, -1);
            lvl[source] = 0;
            q.push(source);
            if(!BFS()) break;
            ptr.assign(n, 0);
            while(true) {
                pushed = dfs(source, inf_flow);
                if(!pushed) break;
                flow += pushed;
            }
        }
        return flow;
    }
};
```

LCA

```
const int MAX_N = 1e5+5;
const int MAX_LOG_N = 18;
int parent[MAX_N][MAX_LOG_N]; // Sparse table.
class LCA{ // LCA in O(log n), with O(n log n) preprocess.
    int n;
    vector<vi> graph;
    void dfs_lvl(int u, int p) {
        parent[u][0] = p;
        lvl[u] = lvl[p] + 1;
        for(auto v : graph[u]) {
            if(v == p) continue;
            dfs_lvl(v, u);
        }
    }
    public:
    int lvl[MAX_N];
    LCA() = default;
    LCA(vector<vi> &_graph) {
        int i, j;
        graph = _graph;
        n = graph.size();
        lvl[0] = 0; // The root is 0.
        dfs_lvl(0, 0); // The parent of root is root.
        for(j = 1; j < MAX_LOG_N; j++) {
            for(i = 0; i < n; i++) {
                parent[i][j] = parent[parent[i][j-1]][j-1];
            }
        }
    }
    int lca(int u, int v) { // O(log n).
        if(lvl[u] > lvl[v]) swap(u, v);
        int i, d = lvl[v] - lvl[u];
        v = get_parent(v, d);

        if(u == v) return u;
        for(i = MAX_LOG_N - 1; i >= 0; i--) {
            if(parent[u][i] != parent[v][i]) {
                u = parent[u][i], v = parent[v][i];
            }
        }
        return parent[u][0];
    }
    int dist(int u, int v) { // distance from u to v O(log n).
        return lvl[u] + lvl[v] - 2*lvl[lca(u, v)];
    }
    int get_parent(int u, int dst) { // Calculate the dst pare
        dst = max(dst, 0);
        for(int i = 0; i < MAX_LOG_N; i++) {
            if(is_set(dst, i)) u = parent[u][i];
        }
        return u;
    }
};
```

SCC Kosaraju

```
vector<vi> graph;
class Kosaraju{ // SCC O(n). x2 times slower than Tarjan.
    vi s; // Stack.
    vector<vi> graphT;
    vector<bool> visited;
    void dfs1(int u) {
        visited[u] = true;
        for(auto v : graph[u])
            if(!visited[v]) dfs1(v);
        s.pb(u);
    } // Add to the current component.
    void dfs2(int u) {
        visited[u] = true;
        for(auto v : graphT[u])
            if(!visited[v]) dfs2(v);
        components.back().pb(u);
    }
    public:
    vector<vi> components;
    Kosaraju() {
        int i, n = graph.size();
        visited.assign(n, false);
        for(i = 0; i < n; i++)
            if(!visited[i]) dfs1(i);
        graphT.assign(n, vi());
        for(i = 0; i < n; i++)
            for(auto v : graph[i])
                graphT[v].pb(i);
        visited.assign(n, false);
        while(true) {
            while(!s.empty() && visited[s.back()])
            s.pop_back();
            if(s.empty()) break;
            components.pb(vi());
            dfs2(s.back());
        }
    }
};
```

Mod operations

```
ll mul(ll a, ll b) {
    ll ans = 0, neg = (a < 0) ^ (b < 0);
    a = abs(a); b = abs(b);
    while(b) {
        if(b & 1) ans = (ans + a) % mod;
        b >>= 1;
        a = (a + a) % mod;
    }
    if(neg) return -ans;
    return ans;
}
ll elevate(ll a, ll b) { // b >= 0.
    ll ans = 1;
    while(b) {
        if(b & 1) ans = ans * a % mod;
        b >>= 1;
        a = a * a % mod;
    }
    return ans;
}
// ONLY USE WHEN MOD IS PRIME, ELSE USE GCD.
// a^(mod - 1) = 1, Euler.
ll inv(ll a) {
    return elevate(((a%mod) + mod)%mod, mod - 2);
}
const int MAX = 1e5 + 10;
//inv_fact is fact^-1
ll fact[MAX], inv_fact[MAX];
void init() {
    int i = 0;
    fact[0] = 1;
    inv_fact[0] = 1;
    for(i = 1; i < MAX; i++) {
        fact[i] = fact[i-1]*i;
        fact[i] %= mod;
        inv_fact[i] = inv(fact[i]);
    }
}
```

Catalan Numbers // (2nCn)/(n+1).

Chinese Remainder

```
class ChineseRemainder{
    ll mul(ll a, ll b) {
        ll ans = 0, neg = (a < 0) ^ (b < 0);
        a = abs(a); b = abs(b);
        while(b) {
            if(b & 1) ans = (ans + a)%modulo;
            b >>= 1;
            a = (a + a)%modulo;
        }
        if(neg) return -ans;
        return ans;
    }

    ll gcdEx(ll a, ll b, ll *x1, ll *y1) {
        if(a == 0) {
            *x1 = 0;
            *y1 = 1;
            return b;
        }
        ll x0, y0, g;
        g = gcdEx(b%a, a, &x0, &y0);
        *x1 = y0 - (b/a)*x0;
        *y1 = x0;
        return g;
    }

    ll mod(ll x) {
        return ((x%modulo) + modulo)%modulo;
    }
    ll n = 0;
    vll c, m; // x == c[i] mod m[i], m[i] not need to be coprime.
    void solve_one() { // m[i] <= 1e9
        ll x, y, g;
        modulo = m[n-2]*(m[n-1]/__gcd(m[n-2], m[n-1]));
        g = gcdEx(m[n-1], m[n-2], &x, &y);
        if((c[n-1]-c[n-2])%g != 0) {n = -1; return;}
        x = c[n-1] + mul(m[n-1], mul(-x, (c[n-1]-c[n-2])/g));
        c.pop_back(); m.pop_back(); n--;
        c[n-1] = mod(x); m[n-1] = modulo;
    }
    public:
    ll modulo;
    void insert(ll _c, ll _m) { // _m > 0.
        n++;
        modulo = _m;
        m.pb(_m);
        c.pb(mod(_c));
    }
    ll solve() {
        while(n > 1) solve_one();
        return n <= 0? -1 : c[0];
    }
};
```

Formulas

```
// a/b is truncate function.
ll floor_div(ll a, ll b) {
    return a/b - ((a^b) < 0 && a%b);
}
// Sum_{0, n} i = Sum of i in [0, n] = n*(n+1)/2.
ll formula_1(ll _n) {
    return _n*(_n+1)/2;
}
// Sum_{a, b} i = Sum of i in [a, b].
ll formula_2(ll _a, ll _b) {
    return formula_1(_b) - formula_1(_a-1);
}
// Sum_{n} i/k = Sum of i/k (floor) in [0, n]. n >= 0.
ll formula_3(ll _n, ll _k) {
    ll _ans = 0, r = _n;
    while((r+1)%_k != 0) {_ans += r/_k; r--;}
    return _ans + _k*formula_1(r/_k);
}
// Sum_{0, n} (x + i*d) Arithmetic sum.
ll formula_4(ll x, ll d, ll _n) {
    return _n*x + d*formula_1(_n);
}
// Sum_{0, n} i^2 = Sum of i^2 in [0, n] = (n(n+1)(2n+1))/6.
ll formula_5(ll _n) {
    return _n*(_n+1)*(2*_n+1)/6;
}
// Sum_{0, n} i^3 = Sum of i^3 in [0, n] = ((n(n+1))/2)^2.
ll formula_6(ll _n) {
    ll ans = formula_1(_n);
    return ans*ans;
}
// Sum_{0, inf} x^i = 1/(1-x) if abs(x) < 1, inf abs(x) >= 1.
// Sum_{0, inf} i*x^i = x/(1-x)^2 if abs(x) < 1, inf abs(x) >=
ll elevate(ll a, ll b) { // b >= 0.
    ll ans = 1;
    while(b) {
        if(b & 1) ans = ans * a;
        b >>= 1;
        a = a * a;
    }
    return ans;
}
// Sum_{0, n} x^i = Sum of x^i in [0, n] =
//(Last*Ratio - First)/(Ratio - 1). Geometric sum.
ll formula_7(ll r, ll n) {
    return (elevate(r, n + 1) - 1) / (r - 1);
}
// Number of digits of num in base 10.
ll formula_8(ll num) { // floor(log10(num)) + 1.
    return log10(num)+1;
}
```

Extended GCD

```
// a*x1 + b*y1 = g;
ll gcdEx(ll a, ll b, ll *x1, ll *y1) {
    if(a == 0) {
        *x1 = 0;
        *y1 = 1;
        return b;
    }
    ll x0, y0, g;
    g = gcdEx(b%a, a, &x0, &y0);
    *x1 = y0 - (b/a)*x0;
    *y1 = x0;
    return g;
}
```

All inverses

```
const ll mod = 31;
ll inverse[mod];
// Calculates inverse for all i < mod.
void init() {
    inverse[1] = 1;
    for(ll i = 2; i < mod; i++) {
        inverse[i] = -(mod/i)*inverse[mod%i];
        inverse[i] = (inverse[i]%mod + mod) % mod;
    }
}
```

Linear Sieve

```
const int MAX_PRIME = 1e6+5;
bool num[MAX_PRIME]; // If num[i] = false => i is prime.
int num_div[MAX_PRIME]; // Number of prime divisors of i.
int min_div[MAX_PRIME]; // The smallest prime that divide i.
vector<int> prime;
void linear_sieve(){

    int i, j, prime_size = 0;
    min_div[1] = 1;
    for(i = 2; i < MAX_PRIME; ++i){
        if(num[i] == false) {prime.push_back(i); ++prime_size; num_div[i] = 1; min_div[i] = i;}

        for(j = 0; j < prime_size && i * prime[j] < MAX_PRIME; ++j){}

            num[i * prime[j]] = true;
            num_div[i * prime[j]] = num_div[i] + 1;
            min_div[i * prime[j]] = min(min_div[i], prime[j]);
            if(i % prime[j] == 0) break;
        }
    }
}
bool is_prime(ll n) {
    for(auto el : prime) {
        if(n == el) return true;
        if(n%el == 0) return false;
    }
    return true;
}
vll fact, nfact; // The factors of n and their exponent. n >=
void factorize(int n) { // Up to MAX_PRIME*MAX_PRIME.
    ll cont, prev_p;
    fact.clear(); nfact.clear();
    for(auto p : prime) {
        if(n < MAX_PRIME) break;
        if(n%p == 0) {
            fact.pb(p);
            cont = 0;
            while(n%p == 0) n /= p, cont++;
            nfact.pb(cont);
        }
    }
    if(n >= MAX_PRIME) {
        fact.pb(n);
        nfact.pb(1);
        return;
    }
    while(n != 1) { // When n < MAX_PRIME, factorization in al
        prev_p = min_div[n];
        cont = 0;
        while(n%prev_p == 0) n /= prev_p, cont++;
        fact.pb(prev_p);
        nfact.pb(cont);
    }
}
```

Lazy Segment Tree

```
template<typename T>
class Node { // Only modify this class.
    public:
    int l = -1, r = -1; // Interval [l, r].
    T value = 0;
    static const T lazy_default = -inf; // Don't change.
    T lazy = lazy_default;
    Node() = default;
    Node(T _value) {value = _value;}
    // Merge nodes.
    Node(Node<T> a, Node<T> b) {value = max(a.value, b.value);} // MINMAX, SUM query.
    void actualize_update(T x) {
        if(x == -inf) return;
        if(lazy == -inf) lazy = 0;
        lazy += x; // (= SET update), (+= SUM update).
        value += x; // MINMAX query + (= SET update), (+= SUM update).
        // value = (r-l+1)*x; // SUM query + (= SET update), (+= SUM update).
    }
};
template<typename T>
class LazySegmentTree { // Use lazy propagation.
    vector<Node<T>> tree;
    vector<T> v;
    int n;
    // Value is the real value, and lazy is only for its children.
    void push_lazy(int k, int l, int r) {
        if(l != r) {
            tree[k<<1].actualize_update(tree[k].lazy);
            tree[k<<1|1].actualize_update(tree[k].lazy);
            tree[k] = Node<T>(tree[k<<1], tree[k<<1|1]);
            tree[k].l = l; tree[k].r = r;
        }
        tree[k].lazy = tree[k].lazy_default;
    }
    void build(int k, int l, int r) {
        if(l == r) {
            tree[k] = Node<T>(v[l]);
            tree[k].l = l; tree[k].r = r;
            return;
        }
        int mid = (l + r) >> 1;
        build(k<<1, l, mid);
        build(k<<1|1, mid+1, r);
        tree[k] = Node<T>(tree[k<<1], tree[k<<1|1]);
        tree[k].l = l; tree[k].r = r;
    }
    void update(int k, int l, int r, int ql, int qr, T x) {
        push_lazy(k, l, r);
        if(qr < l || r < ql) return;
        if(ql <= l && r <= qr) {
            tree[k].actualize_update(x);
        } else {
            int mid = (l + r) >> 1;
            update(k<<1, l, mid, ql, qr, x);
            update(k<<1|1, mid+1, r, ql, qr, x);
        }
        push_lazy(k, l, r);
    }
    Node<T> query(int k, int l, int r, int ql, int qr) {
        push_lazy(k, l, r);
        if(ql <= l && r <= qr) return tree[k];
        int mid = (l + r) >> 1;
        if(qr <= mid) return query(k<<1, l, mid, ql, qr);
        if(mid+1 <= ql) return query(k<<1|1, mid+1, r, ql, qr)
        Node<T> a = query(k<<1, l, mid, ql, qr);
        Node<T> b = query(k<<1|1, mid+1, r, ql, qr);
        return Node<T>(a, b);
    }
    public:
    LazySegmentTree() = default;
    LazySegmentTree(vector<T> _v) {
        v = _v;
        n = v.size();
        tree.assign(4*n, {});
        build(1, 0, n-1);
    }
    void update(int ql, int qr, T x) { // [ql, qr].
        if(ql > qr) swap(ql, qr);
        ql = max(ql, 0);
        qr = min(qr, n-1);
        update(1, 0, n-1, ql, qr, x);
    }
    T query(int ql, int qr) { // [ql, qr].
        if(ql > qr) swap(ql, qr);
        ql = max(ql, 0);
        qr = min(qr, n-1);
        Node<T> ans = query(1, 0, n-1, ql, qr);
        return ans.value;
    }
};
```

Dates

```
// Change here and date_to_num.
ll is_leap_year(ll y) {
    // if(y%4 || (y%100==0 && y%400)) return 0; // Complete leap year.
    if(y%4 != 0) return 0; // Restricted leap year.
    return 1;
}
ll days_month[12] =
{31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
ll days_month_accumulate[12] =
{31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
// d 1-index, m 1-index.
ll date_to_num(ll d, ll m, ll y) {
    ll sum = d;
    m -= 2;
    if(m >= 1) sum += is_leap_year(y);
    y--;
    if(m >= 0) sum += days_month_accumulate[m];
    if(y >= 0) {
        sum += 365*y;
        // sum += y/4 -y/100 + y/400; // Complete leap year.
        sum += y/4; // Restricted leap year.
    }
    return sum;
}
// Tiny optimization, binary search the year, month and day.
void num_to_date(ll num, ll &d, ll &m, ll &y) {
    d = 1; m = 1; y = 0; // The date searched is >= this date.
    while(date_to_num(d, m, y) <= num) y++;
    y--;
    while(date_to_num(d, m, y) <= num) m++;
    m--;
    while(date_to_num(d, m, y) <= num) d++;
    d--;
}
void cin_date(ll &d, ll &m, ll &y) {
    char c;
    cin >> d >> c >> m >> c >> y;
}
void cout_date(ll &d, ll &m, ll &y) {
        if(d < 10) cout << "0";
        cout << d << "/";
        if(m < 10) cout << "0";
        cout << m << "/";
        if(y < 10) cout << "000";
        else if(y < 100) cout << "00";
        else if(y < 1000) cout << "0";
        cout << y;
}
```

Time

```
// Read the hour. scanf("%d:%d:%d", &h, &m, &s);
void cin_hour(ll &h, ll &m, ll &s) {
    char c; // Dummy for read ':'.
    cin >> h >> c >> m >> c >> s;
}
// Prints the hour. printf("%02d:%02d:%02d", h, m, s);
void cout_hour(ll h, ll m, ll s) {
    h %= 24; h += 24; h %= 24;
    m %= 60; m += 60; m %= 60;
    s %= 60; s += 60; s %= 60;
    if(h < 10) cout << "0";
    cout << h << ":";
    if(m < 10) cout << "0";
    cout << m << ":";
    if(s < 10) cout << "0";
    cout << s;
}
// One day has 60*60*24 = 86400 seconds.
// Converts the hour to number of seconds since 00:00:00.
ll hours_to_seconds(ll h, ll m, ll s) {
    return 60*60*h + 60*m + s;
}
// From sec seconds, get the hour. Just's for one day.
void seconds_to_hours(ll &h, ll &m, ll &s, ll sec) {
    sec %= 86400; sec += 86400; sec %= 86400;
    h = sec / (60*60);
    sec %= 60*60;
    m = sec / 60;
    sec %= 60;
    s = sec;
}
// Convert grades of the clock hand to hours and minutes. gh i
// return mp(-1, -1) if no solution exists.
pair<ll, ll> grades_to_hour(ld gh, ld gm) {
    ll h = gh/30, m = gm/6;
    if((ld)30*h + (ld)m/2 != gh || (ld)6*m != gm) return mp(-1
    return mp(h, m);
}
// Convert hours and minutes to grades of the clock hand, mp(g
pair<ld, ld> hour_to_grades(ll h, ll m) {
    return mp((ld)30*h + (ld)m/2, (ld)6*m);
}
// Convert hours and minutes to grades of the clock hand, mp(g
// Not tested.
pair<ld, pair<ld, ld>> hour_to_grades(ll h, ll m, ll s) {
    return mp((ld)30*h + (ld)m/2 + (ld)s/120,
    mp((ld)6*m + (ld)s/10, (ld)6*s));
}
```

Knapsack

```
const int MAX_KNAPSACK = 2000005;
ll dp[MAX_KNAPSACK]; // dp[i] is maximum value can get with i weight.
vector<pll> v; // (value, weight).
// Return max value with weight <= max_knapsack. O(n*max_knapsack).
// If can repeat elements, iterate 0->max_knapsack.
ll knapsack(int max_knapsack) {
    int i, j, n = v.size();
    ll ans = 0, g = v[0].se;
    for(i = 1; i < n; i++) g = __gcd(g, v[i].se);
    for(i = 0; i < n; i++) v[i].se /= g;
    max_knapsack /= g;
    for(i = 1; i <= max_knapsack; i++) dp[i] = -inf;
    dp[0] = 0;
    for(i = 0; i < n; i++) {
        for(j = max_knapsack; j >= 0; j--) {
            if(dp[j] != -inf && j+v[i].se < MAX_KNAPSACK)
                dp[j + v[i].se] =
                max(dp[j + v[i].se], dp[j] + v[i].fi);
        }
    }
    for(i = max_knapsack; i >= 0; i--) ans = max(ans, dp[i]);
    return ans;
}
```

LIS

```
vll LIS(vll &v) { // Is >=, but can be transformed to fit >.
    int i, t, n = v.size();
    if(n == 0) return vll();
    vll lis, lis_t(n), ans;
    lis.pb(v[0]); lis_t[0] = 1;
    for(i = 1; i < n; i++) {
        // if(v[i] == lis.back()) continue; // For >.
        if(v[i] >= lis.back())
            {lis.pb(v[i]); lis_t[i] = lis.size(); continue;}
        int pos = upper_bound(lis.begin(), lis.end(), v[i]) } lis.begin();
        // if(pos > 0 && lis[pos - 1] == v[i]) continue; // For >.
        lis[pos] = v[i];
        lis_t[i] = pos+1;
    }
    for(i = n-1, t = lis.size(); i >= 0; i--) {
        if(lis_t[i] == t && (ans.empty() || v[i] <= ans.back()
            ans.pb(v[i]), t--; // v[i] < ans.back() for >.
    }
    reverse(ans.begin(), ans.end());
    return ans;
```

DSU

```
class DSU {
    int n;
    vi parent;
    vi rank;
    vi sz; // Size of the component.
    int find_parent(int a){
        if(parent[a] == a) return a;
        return parent[a] = find_parent(parent[a]);
    }
    public:
    int number_components;
    DSU() = default;
    DSU(int _n) {
        n = _n;
        number_components = n;
        parent.assign(n, 0);
        rank.assign(n, 0);
        sz.assign(n, 1);
        for(int i = 0; i < n; ++i) parent[i] = i;
    }
    bool is_connected(int a, int b){
        return find_parent(a) == find_parent(b);
    }
    void merge(int a, int b){
        a = find_parent(a);
        b = find_parent(b);
        if(a == b) return;
        number_components--;
        if(rank[a] > rank[b]) parent[b] = a, sz[a] += sz[b];
        else if(rank[a] < rank[b]) parent[a] = b,
        sz[b] += sz[a];
        else {parent[a] = b; rank[b]++, sz[b] += sz[a];}
    }
    int size(int a) {return sz[find_parent(a)];}
};
```

Unordered Set

```
// Use unordered_set<pii, pair_hash> us or unordered_map<pii, int, pair_hash> um;
struct pair_hash
{
    template <class T1, class T2>
    size_t operator () (pair<T1, T2> const &pair) const
    {
        size_t h1 = hash<T1>()(pair.first);
        size_t h2 = hash<T2>()(pair.second);
        return (h1 ^ 0b11001001011001101) + (0b011001010011100111 ^ h2);
    }
};
```

HashString

```
// https://www.browserling.com/tools/prime-numbers.
// s = a[i], hash = a[0] + b*a[1] + b^2*a[2] + b^n*a[n].
class HashString {
    char initial = '0'; // change initial for range. 'a', 'A', '0'.
    public:
    string s;
    int n, n_p;
    vector<vll> v; // contain the hash for [0..i].
    vll p = {16532849, 91638611, 83157709};
    // prime numbers. // 15635513  77781229
    vll base = {37, 47, 53}; // base numbers: primes that > alphabet // 49 83
    vector<vll> b; // b[i][j] = (b_i^j) % p_i.
    vector<vll> b_inv; // b_inv[i][j] = (b_i^j)^-1 % p_i.
    ll elevate(ll a, ll _b, ll mod){
        ll ans = 1;
        while(_b){
            if(_b & 1) ans = ans*a % mod;
            _b >>= 1;
            a = a*a % mod;
        }
        return ans;
    }
    // a^(mod - 1) = 1, Euler.
    ll inv(int i, int j){
        if(b_inv[i][j] != -1) return b_inv[i][j];
        return b_inv[i][j] = elevate(b[i][j], p[i] - 2, p[i]);
    }
    HashString() = default;
    HashString(string &_s) { // Not empty strings.
        s = _s;
        n = _s.length();
        n_p = (int)p.size();
        v.assign(n_p, vll(n, 0));
        b.assign(n_p, vll(n, 0));
        b_inv.assign(n_p, vll(n, -1));
        int i, j;
        for(i = 0; i < n_p; i++) {
            b[i][0] = 1;
            for(j = 1; j < n; j++) {
                b[i][j] = (b[i][j-1]*base[i]) % p[i];
            }
            v[i][0] = s[0]-initial+1;
            for(j = 1; j < n; j++) {
                v[i][j] =
                (b[i][j]*(s[j]-initial+1) + v[i][j-1]) % p[i];
            }
        }
    }
    void add(char c) { // Need something previously added.
        int i;
        s += c;
        n++;
        for(i = 0; i < n_p; i++) {
            b[i].pb((b[i][n-2]*base[i]) % p[i]);
            b_inv[i].pb(-1);
            v[i].pb((b[i][n-1]*(c-initial+1) + v[i][n-2]) % p[i]);
        }
    }
    void add(string &_s) {
        for(auto c : _s) add(c);
    }
    vll getHash(int l, int r) {
        ll i, ans;
        vll vans;
        for(i = 0; i < n_p; i++) {
            ans = v[i][r];
            if(l > 0) ans -= v[i][l-1];
            ans *= inv(i, l);
            ans = ((ans%p[i])+p[i])%p[i];
            vans.pb(ans);
        }
        return vans;
    }
    // O(1).
    bool operator == (HashString other) {
        if(n != other.n) return false;
        return getHash(0, n-1) == other.getHash(0, n-1);
    }
    // return the index of the Longest Comon Prefix, -1 if no
    // O(log n).
    int LCP(HashString other) {
        int l = 0, r = min(n, other.n), mid;
        if(s[0] != other.s[0]) return -1;
        if(*this == other) return n-1;
        while(l + 1 < r) {
            mid = (l + r) >> 1;
            if(getHash(0, mid) == other.getHash(0, mid)) l = m
            else r = mid;
        }
        return l;
    }
    bool operator < (HashString other) {
        int id = LCP(other);
        if(id == -1) return s[0] < other.s[0];
        if(*this == other) return false;
        if(id == n) return true; // "ho" < "hol"
        if(id == other.n) return false;

        return s[id+1] < other.s[id+1];
    }
};
```

Z algorithm

```
// Search the ocurrences of t (pattern to search) in s (the text).
// O(n + m). It increases R at most 2n times and decreases at most n times.
// z[i] is the longest string s[i..i+z[i]-1] that is a prefix = s[0..z[i]-1].
void z_algorithm(string &s, string &t) {
    s = t + "$" + s; // "£" is a char not present in s nor t.
    int n = s.length(), m = t.length(), i, L = 0, R = 0;
    vi z(n, 0);
    // s[L..R] = s[0..R-L], [L, R] is the current window.
    for(i = 1; i < n; i++) {
        if(i > R) { // Old window, recalculate.
            L = R = i;
            while(R < n && s[R] == s[R-L]) R++;
            R--;
            z[i] = R - L + 1;
        } else {
            if(z[i-L] < R - i) z[i] = z[i-L]; // z[i] will fall in the window.
            else { // z[i] can fall outside the window, try to increase the window.
                L = i;
                while(R < n && s[R] == s[R-L]) R++;
                R--;
                z[i] = R - L + 1;
            }
        }
        if(z[i] == m) { // Match found.
            //echo("Pattern found at: ", i-m-1);
        }
    }
}
```

Longest Palindromic Substring

```
// LPS Longest Palindromic Substring, O(n).
void Manacher(string &str) {
    char ch = '#'; // '#' a char not contained in str.
    string s(1, ch), ans;
    for(auto c : str) {s += c; s += ch;}
    int i, n = s.length(), c = 0, r = 0;
    vi lps(n, 0);
    for(i = 1; i < n; i++) {
        // lps[i] >= it's mirror, but falling in the interval [L..R]. L = c - (R - c).
        if(i < r) lps[i] = min(r - i, lps[c - (i - c)]);
        // Try to increase.
        while(i-lps[i]-1 >= 0 && i+lps[i]+1 < n && s[i-lps[i]-1] == s[i+lps[i]+1]) lps[i]++;
        // Update the interval [L..R].
        if(i + lps[i] > r) c = i, r = i + lps[i];
    }
    // Get the longest palindrome in ans.
    int pos = max_element(lps.begin(), lps.end()) - lps.begin();
    for(i = pos - lps[pos]; i <= pos + lps[pos]; i++) {
        if(s[i] != ch) ans += s[i];
    }
    //cout << ans.size() << "\n";
}
```

2 SAT

```
vector<vi> graph;
class SAT{ // 2SAT, (xi or xj) and ()... O(n).
    public:
    SAT(int n) {
        graph.assign(2*n, vi());
    }
    int get_pos(int i) {return 2*i;}
    int get_neg(int i) {return 2*i + 1;}
    void add_or(int i, int j) { // Use it with get_pos.
        graph[i^1].pb(j);
        graph[j^1].pb(i);
    }
    void add_value(int i, int val) { // x[i] = val;
        if(val) add_or(get_pos(i), get_pos(i));
        else add_or(get_neg(i), get_neg(i));
    }
    vector<bool> x; // Can add (xi or xi) if you know xi = true.
    bool solve() {
        Kosaraju kosaraju;
        int i, n = graph.size(), n_component = kosaraju.components.size();
        vi el2component(n, 0);
        for(i = 0; i < n_component; i++)
            for(auto u : kosaraju.components[i])
            el2component[u] = i;
        for(i = 0; i < n; i += 2)
            if(el2component[i] == el2component[i + 1])
            return false;
        vector<vi> graph2 = graph;
        graph.assign(n_component, vi());
        for(i = 0; i < n; i++)
            for(auto u : graph2[i])
            if(el2component[i] != el2component[u])
                graph[el2component[i]].pb(el2component[u]);
        Toposort toposort(n_component);
        x.assign(n/2, false);
        vi component_order(n_component, 0);
        for(i = 0; i < n_component; i++)
            component_order[toposort.vSorted[i]] = i;
        for(i = 0; i < n; i += 2)
            x[i/2] = component_order[el2component[i]]
            > component_order[el2component[i + 1]];
        return true;
    }
};
```

Python Template

```
from decimal import Decimal, getcontext
import math, sys
input = sys.stdin.readline
# getcontext().prec = 3 # 3 de precision, trunca el resto.
# n = Decimal(1) / Decimal(3)
# try:
#    x = input() # until EOF.
#    if len(x) == 0:
#        exit(0)
# except:
#    exit(0)
# v = [k for k in map(int, s.split(' '))]
```

```
vector<vector<pll>> graph; // Dijkstra
// Return the vector of minimum distance between s to all other n nodes, inf means unrecheable.
// u = (cost, next node), graph[u] = vector of (next node, cost).
vll dijkstra(ll s) {
    priority_queue<pll, vector<pll>, greater<pll>> p;
    vll dist(graph.size(), inf);
    pll u;
    p.push(mp(0, s));
    while(p.empty() == false) {
        u = p.top(); p.pop();
        if(dist[u.se] != inf) continue;
        dist[u.se] = u.fi;
        for(auto el : graph[u.se]) {
            if(dist[el.fi] != inf) continue;
            p.push(mp(u.fi + el.se, el.fi));
        }
    }
    return dist;
}
```