Echo's notebook

```
FLAGS=-Wall -Wextra -Wshadow -Wno-unused-result -D_GLIBCXX_DEBUG -fsanitize=address -fsanitize=undefined
↪  -fno-sanitize-recover

@g++ A.cpp $(FLAGS) -DJUNCO_DEBUG && ./a.out < z.in
```

```
// Iterate over all submasks of a mask. CONSIDER SUBMASK = 0 APART.
for(submask = mask; submask > 0; submask = (submask-1)&mask) {}
```

LIS

```
vll v_LIS(vll &v) {
    int i, j, n = v.size();
    vll lis, lis_time(n), ans;
    if(!n) return ans;
    lis.pb(v[0]); lis_time[0] = 1;
    for(i = 1; i < n; i++) {
        if(v[i] > lis.back()) {lis.pb(v[i]); lis_time[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; // USE IF YOU WANT STRICTLY INCREASING.
        lis[pos] = v[i];
        lis_time[i] = pos+1;
    }
    j = lis.size();
    for(i = n-1; i >= 0; i--) {
        if(lis_time[i] == j && (ans.empty() || v[i] <= ans.back())) {ans.pb(v[i]); j--;} // <= or <.
    }
    reverse(ans.begin(), ans.end());
    return ans;
}
```

IO

```
ios::sync_with_stdio(false); cin.tie(nullptr); cout.tie(nullptr);

stringstream ss;
ss << "Hello world";
ss.str("Hello world");
while(ss >> s) cout << s << endl;
ss.clear();
```

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--;
}
```

Geometry

```cpp
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 origin.
    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 left turn.
    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_quadrant(b);
        int ori = get_orientation(a, b);
        if(ori == 0) return euclidean_distance_squared(a) < euclidean_distance_squared(b);
        return ori == LEFT_TURN;
    }
    // Anticlockwise sort starting at 1º quadrant, respect to *this point.
    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 n).
    void convert_convex_hull(vector<Point<T>> &v) {
        if(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);
        }
    }
};
```

Graphs

Articulation points and bridges

```cpp
vector<vi> adyList; // Graph
vi num, low;        // num and low for DFS
int cnt;            // Counter for DFS
int root, rchild;   // Root and number of (DFS) children
vi artic;           // Contains the articulation points at the end
set<pii> bridges;   // Contains the bridges at the end

void dfs(int nparent, int nnode) {
    num[nnode] = low[nnode] = cnt++;
    rchild += (nparent == root);

    for (auto a : adyList[nnode]) {
        if (num[a] == -1) { // Tree edge
            dfs(nnode, a);
            low[nnode] = min(low[nnode], low[a]);
            if (low[a] >= num[nnode]) {
                artic[nnode] = true;
            }
            if (low[a] > num[nnode]) {
                bridges.insert((nnode < a) ? mp(nnode, a) : mp(a, nnode));
            }
        } else if (a != nparent) { // Back edge
            low[nnode] = min(low[nnode], num[a]);
        }
    }
}
void findArticulations(int n) {
    cnt = 0;
    low = num = vi(n, -1);
    artic = vi(n, 0);
    bridges.clear();

    for (int i = 0; i < n; ++i) {
        if (num[i] != -1) {
            continue;
        }
        root = i;
        rchild = 0;
        dfs(-1, i);
        artic[root] = rchild > 1; //Special case
    }
}
```

Max Flow: Edmond Karp's $\mathcal{O}(VE^2)$

```cpp
vector<vector<ll>> adjList;
vector<vector<ll>> adjMat;

void initialize(int n) {
    adjList = decltype(adjList)(n);
    adjMat = decltype(adjMat)(n, vector<ll>(n, 0));
}

map<int, int> p;
bool bfs(int source, int sink) {
    queue<int> q;
    vi visited(adjList.size(), 0);
    q.push(source);
    visited[source] = 1;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        if (u == sink)
            return true;
        for (auto v : adjList[u]) {
            if (adjMat[u][v] > 0 && !visited[v]) {
                visited[v] = true;
                q.push(v);
                p[v] = u;
            }
        }
    }
    return false;
}
int max_flow(int source, int sink) {
    ll max_flow = 0;
    while (bfs(source, sink)) {
        ll flow = inf;
        for (int v = sink; v != source; v = p[v]) {
            flow = min(flow, adjMat[p[v]][v]);
        }
        for (int v = sink; v != source; v = p[v]) {
            adjMat[p[v]][v] -= flow; // Decrease capacity forward edge
            adjMat[v][p[v]] += flow; // Increase capacity backward edge
        }
        max_flow += flow;
    }
    return max_flow;
}
void addedgeUni(int orig, int dest, ll flow) {
    adjList[orig].pb(dest);
    adjMat[orig][dest] = flow;
    adjList[dest].pb(orig); //Add edge for residual flow
}
void addEdgeBi(int orig, int dest, ll flow) {
    adjList[orig].pb(dest);
    adjList[dest].pb(orig);
    adjMat[orig][dest] = flow;
    adjMat[dest][orig] = flow;
}
```

Bellman Ford's

```cpp
for(i = 0; i < n - 1; i++) { // Iterate n - 1 times.
    for(auto e : edge) {
        if(dist[e.fi.fi] != inf)
            dist[e.fi.se] = min(dist[e.fi.se], dist[e.fi.fi] + e.se);
    }
}
```

Floyd cycle detection

```cpp
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);
}
```

Max Flow: Dinic's $\mathcal{O}(V^2E)$

```cpp
// O(V^2*E) max flow algorithm. For bipartite matching O(sqrt(V)*E), always faster than Edmond-Karp.
// Creates layer's graph with a BFS and then it tries all possibles DFS, branching while the path doesn't reach the sink
struct EdgeFlow {
    ll u, v;
    ll cap, flow = 0; //capacity and current flow
    EdgeFlow(ll _u, ll _v, ll _cap) : u(_u), v(_v), cap(_cap) { }
};

struct Dinic {
    vector<EdgeFlow> edge; //keep the edges
    vector<vll> graph; //graph[u] is the list of their edges
    ll n, n_edges = 0;
    ll source, sink, inf_flow = inf;
    vll lvl; //lvl of the node to the source
    vll ptr; //ptr[u] is the next edge you have to take in order to branch the DFS
    queue<ll> q;

    Dinic(ll _n, ll _source, ll _sink) : n(_n), source(_source), sink(_sink) { //n nodes
        graph.assign(_n, vll());
    }

    void add_edge(ll u, ll v, ll flow) { //u->v with cost x
        EdgeFlow uv(u, v, flow), vu(v, u, 0);
        edge.pb(uv);
        edge.pb(vu);
        graph[u].pb(n_edges);
        graph[v].pb(n_edges+1);
        n_edges += 2;
    }

    bool BFS() {
        ll u;
        while(q.empty() == false) {
            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;
    }

    ll dfs(ll u, ll min_flow) {
        if(u == sink) return min_flow;
        ll pushed, el;
        for(;ptr[u] < (int)graph[u].size(); ptr[u]++) { //if you can pick ok, else you crop that edge for the current bfs layer
            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;
    }

    ll max_flow() {
        ll 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;
    }
};
```

Hungarian Algorithm

```cpp
// The rows are jobs, the columns are workers
pair<ll, vl> hungarian(vector<vl> &matrix) {
    int n = matrix.size(), m = matrix[0].size();
    vl jobP(n), workerP(m + 1), matched(m + 1, -1);

    vl dist(m + 1, inf);
    vi from(m + 1, -1), seen(m + 1, 0);

    for (int i = 0; i < n; ++i) {
        int cWorker = m;
        matched[cWorker] = i;
        std::fill(all(dist), inf);
        std::fill(all(from), -1);
        std::fill(all(seen), false);

        while (matched[cWorker] != -1) {
            seen[cWorker] = true;
            int i0 = matched[cWorker];
            int nextWorker = -1;
            ll delta = inf;

            for (int worker = 0; worker < m;
            ↪  ++worker) {
                if (seen[worker])
                    continue;
                ll candidateDistance =
                ↪  matrix[i0][worker];
                candidateDistance += -jobP[i0] -
                ↪  workerP[worker];

                if(candidateDistance<dist[worker]){
                    dist[worker] =
                    ↪  candidateDistance;
                    from[worker] = cWorker;
                }
                if (dist[worker] < delta) {
                    delta = dist[worker];
                    nextWorker = worker;
                }
            }
            for (int j = 0; j <= m; ++j) {
                if (seen[j]) {
                    jobP[matched[j]] += delta;
                    workerP[j] -= delta;
                } else {
                    dist[j] -= delta;
                }
            }
            cWorker = nextWorker;
        }
        while (cWorker != m) {
            int prevWorker = from[cWorker];
            matched[cWorker] = matched[prevWorker];
            cWorker = prevWorker;
        }
    }
    ll ans = -workerP[m];
    vl rowMatchesWith(n);
    for (int j = 0; j < m; ++j) {
        if (matched[j] != -1) {
            rowMatchesWith[matched[j]] = j;
        }
    }
    return {ans, std::move(rowMatchesWith)};
}
```

Floyd - Warshall: k->i->j

Kosaraju

```cpp
vector<vi> adyList;  // Graph
vector<int> visited; // Visited for DFS
vector<vi> sccs;     // Contains the SCCs at the
↪  end

void dfs(int nnode, vector<int> &v, vector<vi>
↪  &adyList) {
    if (visited[nnode]) {
        return;
    }
    visited[nnode] = true;
    for (auto a : adyList[nnode]) {
        dfs(a, v, adyList);
    }
    v.push_back(nnode);
}

void Kosaraju(int n) {
    visited = vi(n, 0);
    stack<int> s = stack<int>();
    sccs = vector<vi>();

    vector<int> postorder;
    for (int i = 0; i < n; ++i) {
        dfs(i, postorder, adyList);
    }
    reverse(all(postorder));

    vector<vi> rAdyList = vector<vi>(n, vi());
    for (int i = 0; i < n; ++i) {
        for (auto v : adyList[i]) {
            rAdyList[v].push_back(i);
        }
    }

    visited = vi(n, 0);
    vi data;
    for (auto a : postorder) {
        if (!visited[a]) {
            data = vi();
            dfs(a, data, rAdyList);
            if (!data.empty())
                sccs.pb(data);
        }
    }
}
```

LCA tree

```cpp
const int MAX_N = 1e5 + 5;
const int MAX_LOG_N = 18;
int n;
vector<vi> graph; // Directed graph, allways reserve memory for it.
vector<vi> bigraph; // Undirected graph, reserve memory only if needed.

int level[MAX_N]; // level of the node rooted.
int parent[MAX_N][MAX_LOG_N]; // parent[i][j] is the parent 2^j of the node i.

vector<bool> visited_bigraph;
// root_graph(u, -1) roots the bigraph at node u.
void root_graph(int u, int p) {
    if(p == -1) visited_bigraph.assign(n, false);
    for(auto v : bigraph[u]) {
        if(v == p) continue;
        graph[u].pb(v);
        root_graph(v, u);
    }
}

// Calcule the level and parent 1. Don't call.
void dfs_level(int u, int p) {
    parent[u][0] = p;
    level[u] = level[p] + 1;
    for(auto v : graph[u]) {
        if(v == p) continue;
        dfs_level(v, u);
    }
}
// Builds the LCA.
void build_lca(int root) {
    int i, j;
    level[root] = -1;
    dfs_level(root, root); // The parent of the root is itself.
    for(j = 1; j < MAX_LOG_N; j++) {
        for(i = 0; i < MAX_N; i++) {
            parent[i][j] = parent[parent[i][j - 1]][j - 1];
        }
    }
}
// Calculates the LCA(u, v) in O(log n).
int lca(int u, int v) {
    if(level[u] > level[v]) swap(u, v);
    int i, d = level[v] - level[u];
    for(i = MAX_LOG_N - 1; i >= 0; i--) {
        if(is_set(d, i)) v = parent[v][i];
    }
    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];
}
// Calculates the distance(u, v) in a tree in O(log n).
int dist(int u, int v) {
    return level[u] + level[v] - 2 * level[lca(u, v)];
}
```

Mathematics

Binary operations

```cpp
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;
}
// a^(mod - 1) = 1, Euler.
ll inv(ll a) {
    return elevate(((a%mod) + mod)%mod, mod - 2);
}

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;
}
```

Catalan numbers: $C_n = \frac{1}{n+1}\binom{2n}{n}$

Combinatoric numbers

```cpp
 const int MAX_C = 1+66; // 66 is the for long long, C(66, x)
ll Comb[MAX_C][MAX_C];

void calc() {
    int i, j;
    for(i = 0; i < MAX_C; i++) {
        Comb[i][0] = 1;
        Comb[0][i] = 1;
    }
    for(i = 1; i < MAX_C; i++) {
        for(j = 1; j < MAX_C; j++) {
            if(i+j >= MAX_C) continue;
            Comb[i][j] = Comb[i-1][j] + Comb[i][j-1];
        }
    }
}
ll C(ll i, ll j) {
    return Comb[i-j][j];
}
```

Chinese Remainder

```cpp
const ll MAX = 10;
ll a[MAX], p[MAX], n;
// Given n x == a[i] mod p[i], find x,
// or -1 if it doesn't exist.
// Let q[i] = (\prod_{i=0}^{n-1} p[j])/p[i].
// x will be = \sum_{i=0}^{n-1} a[i]*q[i]
// *inv(q[i], mod p[i])
ll chinese_remainder() {
    ll i, j, g, ans = 0, inv1, inv2;
    mod = 1;
    for(i = 0; i < n; i++) {
    // If the p[i] are not coprimes, do them coprimes.
        a[i] %= p[i]; a[i] += p[i]; a[i] %= p[i];
        for(j = 0; j < i; j++) {
            g = __gcd(p[i], p[j]);
            if((a[i]%g + g)%g != (a[j]%g + g)%g)
                return -1;
            // Delete the repeated factor at the correct side.
            if (__gcd(p[i]/g, p[j]) == 1) {p[i] /= g; a[i] %= p[i];}
            else {p[j] /= g; a[j] %= p[j];}
        }
    }
    // If you have a supermod, take P = min(P, supermod);
    for(i = 0; i < n; i++) {
        mod *= p[i];
    }
    for(i = 0; i < n; i++) {
        gcdEx(mod/p[i], p[i], &inv1, &inv2);
        ans += mul(a[i], mul(mod/p[i], inv1));
        ans %= mod;
    }
    return (ans%mod + mod) % mod;
}
```

Euclides

```cpp
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;
}
```

Hash Set

```cpp
const int MAX = 2*1e5+5;
ll val[MAX]; // For random numbers and not index use f with random xor.
void ini() { // CALL ME ONCE.
    srand(time(0));
    for(int i = 0; i < MAX; i++) val[i] = rand();
}
// Hash_set contains a set of indices [0..MAX-1] with duplicates.
// a[i] = sum_x{val_x} % mod p[i].
class Hash_set {
    public:
    vll p = {1237273, 1806803, 3279209}; // Prime numbers.
    vll a = {0, 0, 0};
    int n = 3; // n = p.size();

    void insert(int x) {  // Insert index x.
        for(int i = 0; i < n; i++) a[i] = (a[i] + val[x]) % p[i];
    }
    // Insert all the elements of hs.
    void insert (Hash_set hs) {
        for(int i = 0; i < n; i++) a[i] = (a[i] + hs.a[i]) % p[i];
    }
    bool operator == (Hash_set hs) {
        for(int i = 0; i < n; i++) if(a[i] != hs.a[i]) return false;
        return true;
    }
};
```

Hash of pairs

```cpp
// 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);
    }
};
```

Linear Sieve

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

```

```cpp
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;
        }
    }
}
```

```cpp
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.
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 almost O(1).
        prev_p = min_div[n];
        cont = 0;
        while(n%prev_p == 0) n /= prev_p, cont++;
        fact.pb(prev_p);
        nfact.pb(cont);
    }
}
```

Suffix Array

```cpp
class SuffixArray {
    public:
    int n;
    string s;
    vi p; // p[i] is the position in the order array of the ith suffix (s[i..n-1]).
    vi c; // c[i] is the equivalence class of the ith suffix. When build, c[p[i]] = i, inverse.
    // dont use lcp[0] = 0.
    vi lcp; // lcp[i] is the longest common prefix in s[p[i-1]..n-1] and s[p[i]..n-1].
    // To get lcp(s[i..n-1], s[j..n-1) is min(lcp[c[i]+1], lcp[c[j]]) (use SegTree).
    void radix_sort(vector<pair<pii, int>> &v) { // O(n).
        vector<pair<pii, int>> v2(n);
        vi freq(n, 0); // first frequency and then the index of the next item.
        int i, sum = 0, temp;
        for(i = 0; i < n; i++) freq[v[i].fi.se]++; // Sort by second component.
        for(i = 0; i < n; i++) {temp = freq[i]; freq[i] = sum; sum += temp;}
        for(i = 0; i < n; i++) {v2[freq[v[i].fi.se]] = v[i]; freq[v[i].fi.se]++;}
        freq.assign(n, 0); sum = 0;
        for(i = 0; i < n; i++) freq[v2[i].fi.fi]++; // Sort by first component.
        for(i = 0; i < n; i++) {temp = freq[i]; freq[i] = sum; sum += temp;}
        for(i = 0; i < n; i++) {v[freq[v2[i].fi.fi]] = v2[i]; freq[v2[i].fi.fi]++;}
    }
    SuffixArray() = default;
    SuffixArray(string &_s) {
        s = _s;
        s += "$"; // smaller char to end the string.
        n = s.size();
        int i, k;
        p.assign(n, 0);
        c.assign(n, 0);
        vector<pii> v1(n); // temporal vector to sort.
        for(i = 0; i < n; i++) v1[i] = mp(s[i], i);
        sort(v1.begin(), v1.end());
        for(i = 0; i < n; i++) p[i] = v1[i].se;
        c[p[0]] = 0;
        for(i = 1; i < n; i++) {
            if(v1[i].fi == v1[i - 1].fi) c[p[i]] = c[p[i - 1]];
            else c[p[i]] = c[p[i - 1]] + 1;
        }
        k = 0; // in k+1 iterations sort strings of length 2^(k+1).
        while(c[p[n-1]] != n-1) { // At most ceil(log2(n)).
            vector<pair<pii, int>> v2(n); // temporal vector to sort.
            for(i = 0; i < n; i++) v2[i] = mp(mp(c[i], c[(i + (1 << k)) % n]), i);
```

```
            radix_sort(v2);
            for(i = 0; i < n; i++) p[i] = v2[i].se;
            c[p[0]] = 0;
            for(i = 1; i < n; i++) {
                if(v2[i].fi == v2[i - 1].fi)
                ↪ c[p[i]] = c[p[i - 1]];
                else c[p[i]] = c[p[i - 1]] + 1;
            }
            k++;
        }
    }
    void show_suffixes() { // IMPORTANT use this to
    ↪ debug.
        for(int i = 0; i < n; i++) cout << i << " "
        ↪ << p[i] << " " << s.substr(p[i]) <<
        ↪ endl;
        if(!lcp.empty()) cout << "LCP: " << lcp <<
        ↪ endl;
    }
    // cmp s with t. return -1 if s < t, 1 if s >
    ↪ t, 0 if s == t.
    int cmp_string(int pos, string &t) {
        for(int i = p[pos], j = 0; j < (int)
        ↪ t.size(); i++, j++) {
            if(s[i] < t[j]) return -1; // i < n
            ↪ because s[n-1] = '$'.
            if(s[i] > t[j]) return 1;
        }
        return 0;
    }
    // Count the number of times t appears in s.
    int count_substring(string &t) {
        int l = -1, r = n, mid, L, R;
        while(l + 1 < r) { //
        ↪ -1,...,-1=L,0,...,0,1=R...1.
            mid = (l + r) / 2;
            if(cmp_string(mid, t) < 0) l = mid;
            else r = mid;
        }
        L = l;
        l = -1; r = n;
        while(l + 1 < r) {
            mid = (l + r) / 2;
            if(cmp_string(mid, t) <= 0) l = mid;
            else r = mid;
        }
        R = r;
        return R - L - 1;
    }
    // O(n) build. At most 2n lcp++ and n lcp--;
    void build_lcp() {
        lcp.assign(n, 0);
        for(int i = 0; i < n - 1; i++) {
            if(i > 0) lcp[c[i]] = max(lcp[c[i - 1]]
            ↪ - 1, 0);
            while(s[i + lcp[c[i]]] == s[p[c[i] - 1]
            ↪ + lcp[c[i]]]) lcp[c[i]]++;
        }
    }
    ll number_substrings() {
        ll ans = 0, i;
        for(i = 1; i < n; i++) {
            ans += n - p[i-1] - lcp[i]; // Length
            ↪ of the suffix - lcp with the next
            ↪ suffix.
        }
        ans += n - p[n - 1]; // Plus the last
        ↪ suffix.
        return ans - n; // Remove the '$' symbol on
        ↪ n substrings.
    }
};
string LCS(string s, string &t) {
    int mx = 0, mxi = 0, i, n2 = t.length();
    string ans = "";
    s += "@" + t; // Concatenate with a special
    ↪ char.
    SuffixArray sa(s);
    sa.build_lcp();
    for(i = 1; i < sa.n; i++) {
        // Suffix of s and before suffix of t.
        if(sa.n - sa.p[i] > n2 + 2 && sa.n -
        ↪ sa.p[i-1] <= n2 + 1) {
            if(sa.lcp[i] > mx) mx = sa.lcp[i], mxi
            ↪ = i;
        }
        // Suffix of t and before suffix of s.
        if(sa.n - sa.p[i] <= n2 + 1 && sa.n -
        ↪ sa.p[i-1] > n2 + 2) {
            if(sa.lcp[i] > mx) mx = sa.lcp[i], mxi
            ↪ = i;
        }
    }
    return sa.s.substr(sa.p[mxi], mx);
}
```

BIT Fenweick tree

```
template<typename T>
class BIT{
    vector<T> bit;
    int n;
    public:
    BIT(int _n) {
        n = _n;
        bit.assign(n+1, 0);
    }
    BIT(vector<T> v) {
        n = v.size();
        bit.assign(n+1, 0);
        for(int i = 0; i < n; i++) update(i, v[i]);
    }
    // Point update.
    void update(int i, T dx) {
        for(i++; i < n+1; i += LSB(i)) bit[i] +=
        ↪ dx;
    }
    T query(int r) { // query [0, r].
        T ans = 0;
        for(r++; r > 0; r -= LSB(r)) ans += bit[r];
        return ans;
    }
    T query(int l, int r) { // query [l, r].
        return query(r) - query(l-1);
    }
    // k-th smallest element inserted.
    int k_element(ll k) { // k > 0 (1-indexed).
        int l = 0, r = n+1, mid;
        if(query(0) >= k) return 0;
        while(l + 1 < r) {
            mid = (l + r)/2;
            if(query(mid) >= k) r = mid;
            else l = mid;
        }
        return r;
    }
};
```

Strings: KMP

```cpp
template <typename T>
vi prefixFun(const T &s, int n) {
    vi res(n);
    for (int i = 1; i < n; ++i) {
        int j = res[i - 1];
        while (j > 0 && s[i] != s[j]) {
            j = res[j - 1];
        }
        res[i] = j + (s[i] == s[j]);
    }
    return res;
}

template <typename T>
int kmpSearch(const T &text, int n,
              const T &pattern, int m,
              const vi &patternPre) {

    int count = 0;
    int j = 0;
    for (int i = 0; i < n; ++i) {
        while (j > 0 && text[i] != pattern[j]) {
            j = max(0, patternPre[j] - 1);
        }
        j += (text[i] == pattern[j]);
        if (j == m) {
            count++;
            j = patternPre[j - 1];
        }
    }
    return count;
}
```

Longest Palindromic Substring

```cpp
// 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";
}
```

Z-algorithm

```cpp
// 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;
    int 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 {
            // z[i] will fall in the window.
            if(z[i-L] < R - i) z[i] = z[i-L];
            // z[i] can fall outside the window,
            // try to increase the window.
            else {
                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);
        }
    }
}
```

Hours

```cpp
// 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 is grades of hours and gm grades of minutes.
// 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, -1);
    return mp(h, m);
}

// Convert hours and minutes to grades of the clock hand, mp(grade of large hour hand, small minute hand).
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(grade of large hour hand, small minute hand).
// 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));
}
```

Persistant Segment Tree

```cpp
const int MAX_VERSION = 3*1e4+4; // Maximum number of versions.
template<typename T>
struct node {
    node *pl = NULL, *pr = NULL;
    int l, r, mid;
    T value = 0; // Sum query.
    node(int _l, int _r) {l = _l; r = _r; mid = (l+r)>>1;}
    node(int _l, int _r, T _value) {l = _l; r = _r; value = _value; mid = (l+r)>>1;}
    void update() { // Sum query.
        value = 0;
        if(pl) value += pl->value;
        if(pr) value += pr->value;
    }
}; // Declare outside, else static memory gives seg fault.
node<ll> *root[MAX_VERSION]; //it stores the i versions after updates, start at 0.

template<typename T>
class PersistentSegmentTree {
    vector<T> arr; // Copy of the array to build SegmentTree.

    void build(node<T> *n) { // O(n).
        if(n->l == n->r) {n->value = arr[n->l]; return;}
        n->pl = new node<T>(n->l, n->mid);
        n->pr = new node<T>(n->mid+1, n->r);
        build(n->pl);
        build(n->pr);
        n->update();
    }
    node<T>* update(node<T> *n, int q, ll x) { // O(logn).
        if(n->l == n->r) {
            return new node<T>(n->l, n->r, x);
        }
        node<T> *nod = new node<T>(n->l, n->r);
        if(q <= n->mid) {
            nod->pl = update(n->pl, q, x);
            nod->pr = n->pr;
        } else {
            nod->pl = n->pl;
            nod->pr = update(n->pr, q, x);
        }
        nod->update();
        return nod;
    }
    T query(node<T> *n, int l, int r) { // O(logn).
        if(l <= n->l && n->r <= r) return n->value;
        if(r < n->l || n->r < l) return 0; // Sum query.
        return query(n->pl, l, r) + query(n->pr, l, r);
    }
    public:
    PersistentSegmentTree() = default;
    PersistentSegmentTree(int n){ // Build from empty vector of size n.
        arr.assign(n, 0);
        root[0] = new node<T>(0, n-1);
        build(root[0]);
    }
    PersistentSegmentTree(vector<T> &v) { // Build from vector v.
        arr = v;
        root[0] = new node<T>(0, (int)arr.size() - 1);
        build(root[0]);
    }
    T query(int version, int l, int r) {return query(root[version], l, r);} // O(logn).
    // Set v[idx] = x. Set update.
    void update(int version, int new_version, int idx, T x) { // update the segTree version into new_version root.
```

```
        root[new_version] = update(root[version], idx, x); // O(logn).
    }
};

template<typename T>
class NumberDistinctNumbers { // Works for queries online. For offline can check MO's.
    PersistentSegmentTree<T> pst;
    static const int MAX_ELEMENT_VALUE = 1e6+4; // for querying last[el].

    public:
    NumberDistinctNumbers(vector<T> &v) { // O(nlogn).
        T last[MAX_ELEMENT_VALUE]; // last ocurrence of the i-number in the array, updating from left to right.
        int n = v.size();
        fill(last, last+MAX_ELEMENT_VALUE, -1);
        pst = PersistentSegmentTree<T>(n);
        for(int r = 0; r < n; r++) {
            pst.update(max(0, r-1), r, r, 1); // Actualize r.
            if(last[v[r]] != -1) pst.update(r, r, last[v[r]], 0); // Remove last[v[r]].
            last[v[r]] = r; // Actualize last[v[r]].
        }
    }
    T query(int l, int r) { // Return the number of Distinct numbers in [l..r], O(logn).
        return pst.query(r, l, r);
    }
};
```

Euler Circuit

```
// Euler circuit, visit all edges once.
// Condition: for every node in-degree = out-degree. All edges are in the same SCC (connected).
// For a Euler path the condition is all nodes in-degree = out-degree, one out-degree+1 = in-degree (start with this node) and one out-degree = in-degree+1.
vi euler_tour;
void hierholzer(int u) {
    int v;
    while(!graph[u].empty()) {
        v = graph[u].back();
        graph[u].pop_back(); // DESTROYS THE GRAPH.
        hierholzer(v);
    }
    euler_tour.pb(u);
}

void f_euler_tour() {
    hierholzer(0);
    reverse(euler_tour.begin(), euler_tour.end());
}
```

```
vi vSorted;
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);
}

// Topo sort the n_sz first values of graph.
void topo_sort(int n_sz) {
    vSorted.clear();
    visited.assign(n_sz, false);
    for(int i = 0; i < n_sz; i++) topo_rec(i);
    reverse(vSorted.begin(), vSorted.end());
}
```

Hast String

```
// 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 HashStr {
    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 size. // 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]);
    }
    HashStr() = default; // Initialize later.
    HashStr(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];
            }
```

```
        }
        char initial = 'A'; // change initial for range. 'a', 'A'.
        for(i = 0; i < n_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];
            }
        }
    }
    ll getHash(int l, int r, int imod) {
        ll ans = v[imod][r];
        if(l > 0) ans -= v[imod][l-1];
        ans *= inv(imod, l);
        ans = ((ans%p[imod])+p[imod])%p[imod];
        return ans;
    }
    // O(1).
    bool equals(HashStr other, int l, int r) {
        for(int i = 0; i < n_p; i++) {
            if(getHash(l, r, i) != other.getHash(l, r, i)) return false;
        }
        return true;
    }
    // O(1).
    bool operator == (HashStr other) {
        if(n != other.n) return false;
        return equals(other, 0, n-1);
    }
    // return the index of the Longest Comon Prefix, -1 if no Common Prefix.
    // O(log n).
    int LCP(HashStr 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(equals(other, 0, mid)) l = mid;
            else r = mid;
        }

        return l;
    }
    bool operator < (HashStr 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];
    }
};
```

FFT

```
typedef complex<double> cd;
typedef vector<cd> vcd;

void show(vcd &e) { //for debug
    int cont = 0; for(auto el : e) {cout << " +" << (el.real() > eps ? el.real() : 0) << "x^" << cont++;} cout << endl;
}
void convolution(vcd &a) { //insert a_i and get y_i = sum_j(a_j*w_i^j)
    int i, n = a.size(); //n power of 2
    if(n == 1) return;
    vcd a_even, a_odd;
    for(i = 0; i < n; i++) { //divide part of FFT
        if(i%2) a_odd.pb(a[i]);
        else a_even.pb(a[i]);
    }
    convolution(a_even); //recursive part
    convolution(a_odd);
    cd wn = polar(1.0, 2*(double)PI/n), w = 1.0; //wn^i are the n roots of n-unity
    //cd w;
    for(i = 0; i < n/2; i++) {
        //w = polar(1.0, i*2*(double)PI/n); //avoid precission error, but slower
        a[i] = a_even[i] + w*a_odd[i]; //A(wn^k) = Aeven(wn/2^k) + wn^k+Aodd(wn/2^k)
        a[i + n/2] = a_even[i] - w*a_odd[i];
        //A(wn^k) = Aeven(wn/2^(k-n/2)) - wn^(k-n/2)+Aodd(wn/2^(k-n/2))
        w = w*wn;
    }
}
void deconvolution(vcd &a) { //insert y_i and get a_i = sum_j(y_j*w_i^-j)/n
    for(auto &el : a) el = conj(el); //you can conjugate wn and do a[i]/n o can conj(a[i])/n
    convolution(a); // The coefficients of the polynomial have to be are real
    for(auto &el : a) el /= (double)a.size();
}
// Calculate \sum_{i=0}^{n-1} a[i]*b[n-i].
vcd FFT(vcd &a, vcd &b) { //multiply polynomial a*b
    //vcd a = {1.0, 2.0}, b = {3.0}, c;// a and b examples of polynomials to multiply, real coefficients
    vcd c;
    if(a.size() < b.size()) swap(a, b);
    int i, n = a.size();
    while(n - LSB(n)) n++, a.pb(0.0); //add 0.0's to the next power of two of the next power of two, 3->8
    n++, a.pb(0.0);
    while(n - LSB(n)) n++, a.pb(0.0);
    while((int) b.size() < n) b.pb(0.0); //the grade of a and b equal.
    convolution(a);
    convolution(b); //if you want a*a then delete this 2º call
    for(i = 0; i < n; i++) c.pb(a[i]*b[i]);
    deconvolution(c);
    return c;
}
```

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

Matrix

```cpp
// Determinant: https://cp-algorithms.com/linear_algebra/determinant-gauss.html

template<typename T>
class Matrix {
    public:
    int nrow = 0;
    int ncol = 0;
    vector<vector<T>> v;
    Matrix() {}
    // Empty Matrix.
    Matrix(int _nrow, int _ncol) {
        nrow = _nrow;
        ncol = _ncol;
        v.assign(nrow, vector<T>(ncol, 0));
    }
    // Example: Matrix<ll> a({{1, 2}, {3, 4}}); // Can't use for one column vector.
    Matrix(vector<vector<T>> _v) {
        nrow = _v.size();
        ncol = _v[0].size();
        v = _v;
    }
    friend ostream& operator << (ostream &os, Matrix<T> m) {
        int i, j;
        for(i = 0; i < m.nrow; i++) {
            for(j = 0; j < m.ncol; j++) {
                if(j) cout << " ";
                cout << m.v[i][j]; // Becareful with "-0".
            }
            cout << "\n";
        }
        return os;
    }
    Matrix<T> operator + (const Matrix<T> other) {
        int i, j;
        Matrix<T> ans(nrow, ncol);
        for(i = 0; i < nrow; i++) {
            for(j = 0; j < ncol; j++) {
                ans.v[i][j] = v[i][j] + other.v[i][j];
            }
        }
        return ans.delete_negative_cero();
    }
    // Use this for an empty square Matrix to create an identity Matrix.
    Matrix<T> convert_to_identity() {
        for(int i = 0; i < nrow; i++) v[i][i] = 1;
        return *this;
    }
    Matrix<T> operator * (const Matrix<T> other) {
        int i, j, k;
        Matrix<T> ans(nrow, other.ncol);
        for(i = 0; i < nrow; i++) {
            for(j = 0; j < other.ncol; j++) {
                for(k = 0; k < ncol; k++) {
                    ans.v[i][j] += v[i][k]*other.v[k][j];
                }
            }
        }
        return ans.delete_negative_cero();
    }
    Matrix<T> operator ^ (ll ex) {
        if(ex == 0) {
            Matrix<T> ans(nrow, ncol);
            return ans.convert_to_identity();
        }
        Matrix<T> half = (*this) ^ (ex/2);
        if(ex%2) return half * half * (*this);
        else return half * half;
    }
    bool operator == (const Matrix<T> other) {
        int i, j;
        if(nrow != other.nrow || ncol != other.ncol) return false;
        for(i = 0; i < nrow; i++) {
            for(j = 0; j < ncol; j++) {
                if(abs(v[i][j] - other.v[i][j]) > eps) return false;
            }
        }
        return true;
    }
    bool is_null_matrix() {
        return ncol == 0 || nrow == 0;
    }
    // Change "-0" by "0".
    Matrix<T> delete_negative_cero() {
        int i, j;
        for(i = 0; i < nrow; i++) {
            for(j = 0; j < ncol; j++) {
                if(abs(v[i][j]) < eps) v[i][j] = 0;
            }
        }
        return *this;
    }
    static Matrix<T> gaussian_elimination(Matrix<T> mat, Matrix<T> dato) {
        int i, j, k, imx;
        T mx, val;
        for(i = 0; i < mat.ncol; i++) {
            mx = mat.v[i][i];
            imx = i;
            for(j = i+1; j < mat.nrow; j++) {
                if(mat.v[j][i] > mx) {
                    mx = mat.v[j][i];
                    imx = j;
                }
            }
            // If no pivot found, the matrix is not invertible. Its determinant is 0.
            if(mat.v[imx][i] == 0) return Matrix<T>(0, 0);
            // Swap the line with the highest value.
            for(j = i; j < mat.ncol; j++) {
                swap(mat.v[i][j], mat.v[imx][j]);
            }
            for(j = 0; j < dato.ncol; j++) {
                swap(dato.v[i][j], dato.v[imx][j]);
            }
            for(j = i+1; j < mat.nrow; j++) {
                T factor = - mat.v[j][i] / mat.v[i][i]; // Change if using modulus.
                for(k = i; k < mat.ncol; k++) {
                    mat.v[j][k] += factor * mat.v[i][k];
```

```
                }
                for(k = 0; k < dato.ncol; k++) {
                    dato.v[j][k] +=
                    ↪ factor*dato.v[i][k];
                }
            }
        }
        // Solving Ux = dato.
        // For every column of dato.
        for(k = 0; k < dato.ncol; k++) {
            for(i = mat.nrow-1; i >= 0; i--) {
                val = dato.v[i][k];
                for(j = i+1; j < mat.ncol; j++) {
                    val -= mat.v[i][j] *
                    ↪ dato.v[j][k];
                }
                dato.v[i][k] = val / mat.v[i][i];
            }
        }
        return dato.delete_negative_cero();
    }
    // If you are going to *, it loses a lot of
    ↪ precission.
    static Matrix<T> inverse(Matrix<T> mat) {
        Matrix<T> id(mat.nrow, mat.ncol);
        id.convert_to_identity();
        return gaussian_elimination(mat, id);
    }
};
```

Discrete logarithm/root

```
ll elevate(ll a, ll b, ll mod) { // b >= 0.
    ll ans = 1;
    while(b) {
        if(b & 1) ans = ans * a % mod;
        b >>= 1;
        a = a * a % mod;
    }
    return ans;
}

// phi of Euler. O(sqrt(n)).
ll get_phi(ll n) {
    ll ans = n, i;
    for(i = 2; i*i <= n; i++) {
        if(n%i == 0) {
            while(n%i == 0) n /= i;
            ans -= ans/i;
        }
    }
    if(n > 1) ans -= ans/n;
    return ans;
}

// Return g such that for all x coprime with mod
↪ exists k : (g^k == x)% mod.
// g^k generate all the elements.
// If g is the primitive root of mod, you can take
↪ log_g{} in both sides.
// Exists iff mod is 1, 2, 4, (odd p)^k, 2*(odd
↪ p)^k.
// Complexity O(mod log(mod)).
ll primitive_root(ll mod) {
    ll phi = get_phi(mod);
    vll factors; // Factorize phi.
    ll i, num = phi;
    bool ok;
    if(mod == 1) return 0;
    if(mod == 2) return 1;
    for(i = 2; i*i <= num; i++) {
        if(num%i == 0) {
            factors.pb(i);
            while(num%i == 0) num /= i;
        }
    }
    if(num > 1) factors.pb(num);
    // Try every coprime number.
    for(i = 2; i < mod; i++) {
        if(__gcd(i, mod) != 1) continue;
        ok = true;
        for(auto p : factors) {
            if(elevate(i, phi/p, mod) == 1) {
                ok = false;
                break;
            }
        }
        if(ok) return i;
    }
    return -1;
}

// Return the smallest x such as a^x == b % mod, or
↪ -1 if no answer exists.
// x = sqrt(mod)*p - q. Baby step - Giant step
↪ algorithm.
// Complexity O(sqrt(mod)).
ll discrete_logarithm(ll a, ll b, ll mod) {
    a = (a%mod + mod)%mod; b = (b%mod + mod)%mod;
    if(a == 0 && b == 0) return 1;
    if(a == 0) return -1; // 0^0 sometimes is 1.
    if(b == 1) return 0;
    ll k = 1, sq = sqrt(mod) + 1, q, p, g, asq = 1,
    ↪ aq = 1, ap = 1, add = 0;
    unordered_map<ll, ll> value;
    // if a is not coprime con mod then transform
    ↪ it to k*a^x == b%mod.
    while((g = __gcd(a, mod)) != 1) {
        if(b == k) return add; // Stop decreasing
        ↪ x.
        if(b%g) return -1;
        b /= g;
        mod /= g;
        add++;
        k = (k * (a / g))%mod;
    }
    // Meet in the middle, smallest x is high q and
    ↪ low p.
    for(q = 0; q <= sq; q++) {
        value[(b*aq)%mod] = q;
        aq = (aq*a)%mod;
    }
    for(p = 1; p <= sq; p++) asq = (asq*a)%mod;
    for(p = 1; p <= sq; p++) {
        ap = (ap*asq)%mod;
        if(value.count((k*ap)%mod))
            return sq*p - value[(k*ap)%mod] + add;
    }
    return -1;
}

ll p_root = -1;
```

```cpp
// Get x such that x^k == b % mod.
ll discrete_root(ll k, ll b, ll mod) {
    // If you change the mod then calculate again the primitive root.
    if(p_root == -1) p_root = primitive_root(mod);
    if(p_root == -1) return -1;
    ll num = elevate(p_root, k, mod);
    ll y = discrete_logarithm(num, b, mod);
    return elevate(p_root, y, mod);
}
```

Mobius inversion

```cpp
// You can do Mobius inversion, that is Sum_{d|n} mu[d] = [n == 1]. (Maybe n = gcd(a_i) usually).
const int MAX = 1e5;
int mu[MAX]; // mu(n) = 0 if n is square prime and (-1)^t if n = p1...pt.
void mobius_ini() {
    vi prime;
    ll i;
    bool is_composite[MAX];
    fill(is_composite, is_composite+MAX, false);
    mu[1] = 1;
    for(i = 2; i < MAX; i++) {
        if(!is_composite[i]) prime.pb(i), mu[i] = -1;
        for(auto p : prime) {
            if(i * p >= MAX) break;
            is_composite[i * p] = true;
            if(i % p == 0) { // if p divides i.
                // mu[i * p] = 0; // already 0.
                break;
            } else mu[i * p] = mu[i] * mu[p];
        }
    }
}
```

Lazy Segment Tree

```cpp
template<typename T>
class Node { // Only modify this class.
    public:
    T value = numeric_limits<T>::max(); // max for MIN query.
    static const T lazy_default = 0; // Default value for lazy.
    T lazy = lazy_default;
    Node(T _value) {value = _value;}
    // Merge nodes.
    Node(Node<T> a, Node<T> b) {value = min(a.value, b.value);} // MIN query.
    Node() = default;
    void actualize_update(T x) {
        value += x; // MIN query + (= SET update), (+= SUM update).
        lazy += x; // MIN query + (= SET update), (+= SUM update).
    }
};

template<typename T>
class Lazy_SegTree {
    vector<Node<T>> tree;
    vector<T> v_input;
    int v_size;
    // 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].lazy = tree[k].lazy_default;
    }
    void build(int k, int l, int r) {
        if(l == r) {tree[k] = Node<T>(v_input[l]); 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]);
    }
    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:
    Lazy_SegTree(vector<T> v) {
        v_input = v;
        v_size = v_input.size();
        tree.assign(4*v_size, {});
        build(1, 0, v_size-1);
    }
    void update(int ql, int qr, T x) { // [ql, qr].
        update(1, 0, v_size-1, ql, qr, x);
    }
    T query(int ql, int qr) { // [ql, qr].
        Node<T> ans = query(1, 0, v_size-1, ql, qr);
        return ans.value;
    }
};
```

Sparse Table

```cpp
// Sparse Table, table[i][j] = covers [i, i + 2^j - 1], range 2^j.
// CAN'T UPDATE VALUES.
const ll MAX = 1e5;
const int LOG2_MAX = 22; // log2(MAX).
ll table[MAX][LOG2_MAX]; // Outside class.
template<typename T>
class SparseTable {
    int n;
    T f(T a, T b) {
        return min(a, b);
    }
    public:
    SparseTable(vector<T> &v) {
        int i, j;
        n = v.size();
        for(i = 0; i < n; ++i) table[i][0] = v[i];

        for(j = 1; j < LOG2_MAX; ++j){
            for(i = 0; i < n; ++i){
                if(i + (1ll << (j - 1)) >= n)
                    break;
                table[i][j] = f(table[i][j - 1],
                    table[i + (1ll << (j - 1))][j - 1]);
            }
        }
    }
    // [ql..qr], [0..n-1].
    T query(int ql, int qr) {
        int lg2_dif = -1, num = qr - ql;
        if(ql == qr) return table[ql][0];
        while(num) lg2_dif++, num >>= 1;
        return f(table[ql][lg2_dif], table[qr - (1ll << lg2_dif) + 1][lg2_dif]);
    }
};
```

MO's algorithm

```cpp
//MO's algorithm, similar than sqrt decomposition. First sort the queries and then keep adding and
//removing elements until your current interval is the query interval and report the answer
//Usefull when you can compose the answer with a smaller or bigger interval. O((Q+N)sqrt(N))
const int BLOCK = //3; //sqrt(max v.size)

struct Query{
    int l, r, id;
};

//Sort first by block, second by R
bool Query_cmp(Query a, Query b) {
    if(a.l / BLOCK != b.l / BLOCK) {
        return a.l / BLOCK < b.l / BLOCK;
    }
    return a.r < b.r;
}

const int MAX = 1e5+4;
vi v, frec(MAX, 0);
int answer = 0;

//add data to the answer
void add(int i) {
    ++frec[v[i]];
    if(frec[v[i]] == 1) ++answer;
}

//remove data to the answer
void remove(int i) {
    --frec[v[i]];
    if(frec[v[i]] == 0) --answer;
}

void MO() {
    int i, currL = 0, currR = 0;
    v = {2, 3, 1, 1, 2, 1, 2, 3};
    vector<Query> vq = {{0, 5, 0}, {6, 7, 1}, {0, 3, 2}};
    //Sort the queries
    sort(vq.begin(), vq.end(), Query_cmp);

    //The answer contains data of the interval [L..R)
    for(i = 0; i < (int)vq.size(); i++) {
        while(currL < vq[i].l) {
            remove(currL);
            ++currL;
        }
        while(currL > vq[i].l) {
            --currL;
            add(currL);
        }

        while(currR <= vq[i].r) {
            add(currR);
            currR++;
        }
        while(currR -1 > vq[i].r) {
            currR--;
            remove(currR);
        }

        cout << "[" << vq[i].l << " " << vq[i].r << "]:" << answer << endl;
        //ans[vq[i].id] = answer //to sort the answer
    }
}
```

Square Root Decomposition

```cpp
template<typename T>
class SquareRootDecomposition {
    const int B = //3; //size of the bucket, ~ sqrt(N)
    vector<T> bucket;//(N/B + 1, 0T);
    vector<T> v;
    public:
    SquareRootDecomposition(vector<T> &_v) {
        bucket.assign((int)_v.size()/B +1, 0);
        v = _v;
```

```
        for(int i = 0; i < (int)v.size(); i++) {
            bucket[i/B] += v[i];
        }
    }
    // [l..r].
    T query(int l, int r) {
        T ans = 0;
        int i;
        if(l/B == r/B) {
            for(i = l; i <= r; i++) ans += v[i]; // Same block.
            return ans;
        }
        for(i = l/B + 1; i <= r/B -1; i++) ans += bucket[i]; //middle blocks
        for(i = l; i/B == l/B; i++) ans += v[i]; //left block
        for(i = r; i/B == r/B; i--) ans += v[i]; //right block

        return ans;
    }
    // Replace v[x] by dx.
    void update(int x, T dx) {
        bucket[x/B] += dx - v[x];
        v[x] = dx;
    }
};
```

Aho Corasick

```
//construct trie O(m) + automaton O(mk), O(mk) memory, m = sum(len(word_i))
#define next asdfa
//size of alphabet, 26 lowercase
const int k = 26;

struct vertex{
    vi next;
    //number of words ending at current vertex
    int leaf;
    //ancestor p and ch is the transition of p->v
    int p;
    char pch;
    //proper suffix link of the vertex
    int link;
    vi go;
    //how many suffixes there are in the tree;
    int count;

    vertex(int _p, char _pch) {
        next.assign(k, -1);
        leaf = 0;
        this->p = _p;
        this->pch = _pch;
        link = -1;
        go.assign(k, -1);
        count = -1;
    }
};

vector<vertex> t = {{-1, '$'}};
int t_size = 1;

//add string to the trie t
void add_string(string &s) {
    int c, p = 0;
    for(char ch : s) {
        c = ch - 'a';
        if(t[p].next[c] == -1) {
            t.pb({p, ch});
            t[p].next[c] = t_size++;
        }
        p = t[p].next[c];
    }
    t[p].leaf++;
}

//Search for any proper suffix of v that has next[c] transition
//call go(v, ch) for move the automaton from the vertex v using transition ch
int go(int v, char ch);

//get the proper suffix link of v. Once called, don't call anymore add_strings
int get_link(int v) {
    if(t[v].link == -1) {
        if(v == 0 || t[v].p == 0) t[v].link = 0;
        else t[v].link = go(get_link(t[v].p), t[v].pch);
    }
    return t[v].link;
}

int go(int v, char ch) {
    int c = ch - 'a';
    if(t[v].go[c] == -1) {
        if(t[v].next[c] != -1) t[v].go[c] = t[v].next[c];
        //The root doesn't have next[c]
        else if(v == 0) t[v].go[c] = 0;
        else {
            t[v].go[c] = go(get_link(v), ch);
        }
    }
    return t[v].go[c];
}

//get the count of v
int count(int v) {
    if(t[v].count == -1) {
        t[v].count = t[v].leaf;
        if(v != 0) t[v].count += count(get_link(v));
    }
    return t[v].count;
}

//search the number of the strings in the automaton that are in the text
int search_num_string(string &text) {
    int p = 0, ans=0;

    for(auto ch : text) {
        ans += count(p);
        p = go(p, ch);
    }
    ans += count(p);
    return ans;
}
```

LPS

```cpp
// 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";
}
```

Suffix Automaton

```cpp
#define next _42_
//Suffix Automaton, save a directed acyclic graph and a suffix link tree with all the suffix of a word
struct state {
    //length of the longest string in the equivalence classes
    int len;
    //suffix link
    int link = -1;
    map<char, int> next;
    state(int _len) {
        len = _len;
    }
};

vector<state> t = {{0}};
int t_size = 1, last = 0;

//add a character to the automaton
//last is the state of the last char c added, p is the head of the automaton
//q is the state to duplicate
void sa_extend(char c) {
    int p = last, q;
    t.pb({t[last].len + 1});
    last = t_size; t_size++;
    //add c to the previous suffixes
    while(p != -1 && t[p].next.find(c) == t[p].next.end()) {
        t[p].next[c] = last;
        p = t[p].link;
    }
    //first time of c in the string
    if(p == -1) {
        t[last].link = 0;
        return;
    }
    q = t[p].next[c];
    if(t[p].len + 1 == t[q].len) {
        t[last].link = q;
        return;
    }
    //clone state q
    t.pb({t[p].len + 1});
    t_size++;
    t[t_size - 1].next = t[q].next;
    t[t_size - 1].link = t[q].link;

    //add links of last and q
    t[last].link = t_size - 1;
    t[q].link = t_size - 1;

    //point the last suffixes to q cloned
    while(p != -1 && t[p].next.find(c) != t[p].next.end()) {
        t[p].next[c] = t_size - 1;
        p = t[p].link;
    }
}

//O(s.length()) to create the automaton. Be careful adding any char once called another function
void sa_ini(string &s) {
    for(char c : s) sa_extend(c);
}

//A path from root to a terminal node is a suffix of the automaton string
vector<bool> terminal;
void sa_terminal() {
    int p = last;
    if(terminal.empty() == false) return; //previously calculated
    terminal.assign(t_size, false);
    while(p != -1) {
        terminal[p] = true;
        p = t[p].link;
    }
}

//true if w is a substring of the automaton string
//Also s is the longest prefix of w that is in s
//w is a suffix if the last p is a terminal state
bool sa_is_substr(string &w) {
    int p = 0; //string s;
    for(char ch : w) {
        if(t[p].next.find(ch) == t[p].next.end()) return false;
        p = t[p].next[ch];
        //s += c;
    }
    return true;
}

vll dp_num_substr;
```

```cpp
ll num_substr_rec(int i) {
    ll sum = 1;
    if(dp_num_substr[i] != -1) return dp_num_substr[i];
    for(auto el : t[i].next) sum += num_substr_rec(el.se);
    return dp_num_substr[i] = sum;
}
//Number of different substrings of the automaton string (Is the number of different paths in the automaton)
//For the number of the length of all different substring the recursive formula is
// sum of dp_num_substr[i] + dp_num_len_substr[i], the previous answer + 1*number of different substrings
ll sa_num_substr() {
    if(dp_num_substr.empty() == false) return dp_num_substr[0]; //previously calculated
    dp_num_substr.assign(t_size, -1);
    num_substr_rec(0);
    return dp_num_substr[0]; // -1 if you don't want the empty substring
}

//k-th string in the sorted substrings set of the automaton string. It's the k-th path in the graph
//k is [0..sa_num_substr()-1]
string sa_k_substr(int k) {
    int p = 0;
    char prev = '$';
    string ans = "";
    if(k > sa_num_substr()) return ans; //not exists that k-th string, error
    while(k > 0) {
        prev = '$';
        for(auto el : t[p].next) {
            prev = el.fi;
            if(dp_num_substr[el.se] >= k) break;
            k -= dp_num_substr[el.se];
        }
        if(prev == '$') break; //error
        ans += prev;
        p = t[p].next[prev];
        k--;
    }
    return ans;
}

//lexicographically smallest cyclic shift of the string s
string sa_small_cyclic_shift(string &s) {
    int p = 0, cnt = s.length();
    string ans = "";
    sa_ini(s + s); //initialize sa with s+s, the ans is greedy the first path with length s.length()
    while(cnt--) {
        auto el = *(t[p].next.begin()); //take greedy the first edge
        ans += el.fi;
        p = el.se;
    }

    return ans;
}

//int sa_num_ocurrences(string w); //Better use Aho-Corasick


//Test of the automaton string, the number of the substrings and the substrings, sorted
void sa_test1() {
    ll i, n;
    sa_ini("test");
    n = sa_num_substr();
    cout << n << endl;
    for(i = 0; i < n; i++)
        cout << sa_k_substr(i) << endl;
}
```

Python Template

```python
import math, sys
input = sys.stdin.readline
# try:
#    x = input() # until EOF.
# except:
#    exit(0)
# v = [k for k in map(int, s.split(' '))]
```

```cpp
#define mp make_pair
#define pb push_back
#define fi first
#define se second
#define LSB(x) ((x) & (-(x)))
#define is_set(x, i) (((x)>>(i))&1)
#define set_bit(x, i) {(x) |= 1ll<<(i);}
#define unset_bit(x, i) {(x) = ((x) | (1ll<<(i))) ^ (1ll<<(i));}
const long double PI = acos(-1);
const long double eps = 1e-9;
const long long inf = LLONG_MAX / 10;
```

2-SAT

```cpp
// 2-SAT. Check values (xi or xj) and ... and (xk or xz).
// xi will be element 2*i and not xi will be 2*i+1. Change them with xi xor 1.
vector<vi> graph; // size of graph will be 2*(the number of xi).

int get_element(int n) {return 2*n;} // Get pos of xi.
int get_not_element(int n) {return 2*n + 1;} // Get pos of not xi.
// Add (xi or xj), two edges: (not xj => xi) and (not xi => xj).
// inclusive or: 1 or 1 = 1. For exclusive use 2 clausules. (xi or xj) and (not xi or not xj).
void add_or_clausule(int i, int j) {
    int neg_i = i^1, neg_j = j^1;
    graph[neg_i].pb(j);
    graph[neg_j].pb(i);
```

```
}

// Use Kosaraju to find the SCCs.
vector<vi> graphRev;
stack<int> s;
vector<bool> visited; // It will be reutilized in SAT.
vector<vi> components;

void dfs1(ll u){
    visited[u] = true;
    for(auto v : graph[u]){
        if(!visited[v]) dfs1(v);
    }
    s.push(u);
}
void dfs2(ll u){
    visited[u] = true;
    for(auto v : graphRev[u]){
        if(!visited[v]) dfs2(v);
    }
    components.back().pb(u); // One element more to the current component.
}
void Kosaraju(){
    ll i, n = graph.size();
    graphRev.assign(n, vi());
    s = stack<int>();
    //transpose graph to graphRev
    for(i = 0; i < n; ++i){
        for(auto v : graph[i]){
            graphRev[v].pb(i);
        }
    }
    visited.assign(n, false);
    for(i = 0; i < n; i++)
        if(!visited[i])
            dfs1(i);
    visited.assign(n, false);
    components.pb(vi());
    while(true) {
        while(!s.empty() && visited[s.top()] == true) s.pop();
        if(s.empty()) break;

        dfs2(s.top());
        components.pb(vi()); // End of the current component.
    }
}
// Do a topoSort of the SCCs.
vector<vi> graph_topo;
vi vSorted;
void topo_rec(int u) {
    if(visited[u]) return;
    visited[u] = true;
    for(auto _v : graph_topo[u]) topo_rec(_v);
    vSorted.pb(u);
}

// Topo sort the n_sz first values of graph.
void topo_sort(int n_sz) {
    vSorted.clear();
    visited.assign(n_sz, false);
    for(int i = 0; i < n_sz; i++) topo_rec(i);
    reverse(vSorted.begin(), vSorted.end());
}
// xi_value[i] is 0 if xi is false, 1 if true.
vi xi_value;
// If you know in advance elements of xi add a clausule (xi or xi).
// Return true if the base is satisfactible, false otherwise.
// All calls are O(n).
vi node2component; // Index of the component of the node.
vi component2order; // Order of the component in the topological sort.
bool SAT() {
    Kosaraju();
    int n_components = components.size(), i, n = graph.size();
    node2component.assign(n, 0);
    graph_topo.assign(n_components, vi());
    component2order.assign(n_components, 0);
    // All components in the same SCC will have the same truth value.
    for(i = 0; i < n_components; i++) {
        for(auto u : components[i])
            node2component[u] = i;
    }
    // If xi and not xi are in the same component is UNSAT.
    for(i = 0; i < n; i += 2) {
        if(node2component[i] == node2component[i+1]) return false;
    }
    for(i = 0; i < n; i++) {
        for(auto u : graph[i]) {
            if(node2component[i] != node2component[u])

                graph_topo[node2component[i]].pb(node2com
        }
    }
    topo_sort(n_components);
    for(i = 0; i < n_components; i++) {
        component2order[vSorted[i]] = i;
    }
    xi_value.assign(n/2, -1);
    for(i = 0; i < n; i += 2) {
        if(component2order[node2component[i]] > component2order[node2component[i+1]])
            xi_value[i/2] = true;
        else xi_value[i/2] = false;
    }
    return true;
}
```

Convex Hull Trick

```
typedef long double ftype; //NOT USE LONG LONG, complex cast to minor precission
typedef complex<ftype> point;
#define x real
#define y imag

ftype dot(point a, point b) {
    return (conj(a)*b).x();
}

ftype cross(point a, point b) {
    return (conj(a)*b).y();
```

```
}
//get min{k_i * x + b_i}.
//Insert k_i in ascending order. max {} = -min{-()}
//Decreasing k_i then add -k_i and query -x
class ConvexHullTrick {
    point im = {0, 1};
    vector<point> hull, normal;
    public:
    void add_line(ftype k, ftype b) {
        point nw = {k, b};
        //Create lower convex hull, with increasing k (add lines to the right only)
        while(!normal.empty() && dot(normal.back(), (nw - hull.back())) < 0) {
            hull.pop_back();
            normal.pop_back();
        }
        if(!hull.empty()) { //add the normal of vector (nw - hull.back())
            normal.pb(im * (nw - hull.back()));
        }
        hull.pb(nw);
    }
    // normal anti-clockwise, hull[it], normal clockwise
    ftype query(ftype x) {
        point px = {x, 1}; //query is min dot product {k, b} * {x, 1}

        int pos = lower_bound(normal.begin(), normal.end(), px, [](point a, point b) {
            return cross(a, b) > 0;
        }) - normal.begin();
        return dot(px, hull[pos]);
    }
};
```

Find Centroid

```
int subtree_sz[MAX];// Number of nodes in the subtree u, rooted at 1.
int mn = MAX; // Min {Max{sz(T_i)}} and T_i are the trees created when cutting node i.
vi centroid; // The centroids. Always there are 1 centroid or 2: x and y, and edge x-y exist.
vector<pii> subtree[MAX]; // subtree[u] = (fi, se) fi is one child of u and se is the size of that subtree.
void fill_sz(int u, int p) { // Recursive fill subtree array.
    subtree_sz[u] = 1;
    for(auto v : graph[u]) {
        if(v == p) continue;
        fill_sz(v, u);
        subtree_sz[u] += subtree_sz[v];
        subtree[u].pb(mp(v, subtree_sz[v]));
    }
    if(p != -1)
        subtree[u].pb(mp(p, n-subtree_sz[u])); // n is the number of nodes of the graph.
    int mx = 0;
    for(auto el : subtree[u]) mx = max(mx, el.se);
    mn = min(mn, mx);
}

// 1-indexed!
void find_centroid() { // fill vi centroid.
    int mx = 0, i;
    mn = MAX;
    centroid.clear();
    for(i = 1; i <= n; i++) subtree[i].clear();
    fill_sz(1, -1);
    for(i = 1; i <= n; i++) {
        mx = 0;
        for(auto el : subtree[i]) {
            mx = max(mx, el.se);
        }
        if(mx == mn) centroid.pb(i);
    }
}
```

Java Template

```
import java.io.*;
import java.math.*;
// .setScale(m, RoundingMode.FLOOR);
// .divide(denom, m, RoundingMode.FLOOR);
@SuppressWarnings("unused")
public class Main {
    private final static int MAX = 100_005;
    public static void solve() {
    }
    public static void main(String[] args) {
        out = new PrintWriter(new BufferedOutputStream(System.out));
        sc = new MyScanner();
        int zz = 1;//sc.nextInt();
        while (zz-- > 0) solve();
        out.close();
    }
    private static PrintWriter out;
    private static MyScanner sc;
    private static class MyScanner {
        private static final int BUF_SIZE = 2048;
        BufferedReader br;
        private MyScanner() {
            br = new BufferedReader(new InputStreamReader(System.in));
```

```
        }
        private boolean isSpace(char c) {
            return c == '\n' || c == '\r' || c == ' ';
        }
        String next() {
            try {
                StringBuilder sb = new StringBuilder();
                int r;
                while ((r = br.read()) != -1 && isSpace((char) r));
                if (r == -1) {
                    return null;
                }
                sb.append((char) r);
                while ((r = br.read()) != -1 && !isSpace((char) r)) {
                    sb.append((char) r);
                }
                return sb.toString();
            } catch (IOException e) {
                e.printStackTrace();
            }
            return null;
        }
        int nextInt() {
            return Integer.parseInt(next());
        }
        long nextLong() {
            return Long.parseLong(next());
        }
        double nextDouble() {
            return Double.parseDouble(next());
        }
    }
}
```

Treap

```
add srand
//https://cp-algorithms.com/data_structures/treap.html
//not implemented: find_by_order, order_of_key, find/search
template<typename K, typename V> //key (unique) and value (data)
class Treap {
    const pair<V, K> MINVK = mp(numeric_limits<V>::min(), numeric_limits<K>::min());
    const pair<V, K> MAXVK = mp(numeric_limits<V>::max(), numeric_limits<K>::max());
    struct node {
        K key; //unique, time or x-axis for example
        V data; // f[key] = data
        node *l = NULL, *r = NULL;
        int priority;
        pair<V, K> mx, mn; //maximum and minimum values over all subtrees, V 1º to be comparable <
        node(K _key, V _data) {
            key = _key;
            data = _data;
            mx = mn = mp(_data, _key);
            priority = rand();////((ll)rand()<<16)^(ll)rand();
            //hope there is no collision...
        }
    };
    typedef node* pnode;
    pnode root = NULL;
    pair<V, K> getMaximumVK(pnode t) { //O(1)
        if(!t) return MINVK;
        else return t->mx;
    }
    pair<V, K> getMinimumVK(pnode t) { //O(1)
        if(!t) return MAXVK;
        else return t->mn;
    }
    void update (pnode t) {
        if(!t) return;
        t->mx = max({mp(t->data, t->key), getMaximumVK(t->l), getMaximumVK(t->r)});
        t->mn = min({mp(t->data, t->key), getMinimumVK(t->l), getMinimumVK(t->r)});
    }
    //return a subtree l and r such as key(l) < key < key(r), similar to rotations
    void split(pnode t, pnode &l, pnode &r, K key) {
        if(!t) l = r = NULL;
        else if(key < t->key) split(t->l, l, t->l, key), r = t;
        else split(t->r, t->r, r, key), l = t;
        update(t);
    }
    //merge two trees l and r into one, t, allKey(l) < allKey(r)
    void merge(pnode &t, pnode l, pnode r) {
        if(!l || !r) t = l ? l : r;
```

```
        else if(l->priority < r->priority)
        ↪ merge(r->l, l, r->l), t = r;
        else merge(l->r, l->r, r), t = l;
        update(t);
    }
    void insert(pnode &t, pnode it) {
        if(!t) t = it;
        else if(t->priority < it->priority)
        ↪ split(t, it->l, it->r, it->key), t =
        ↪ it;
        else insert(it->key < t->key ? t->l : t->r,
        ↪ it);
        update(t);
    }
    void erase(pnode &t, K key) { //only erase if
    ↪ the element exists
        if(!t) {echo("estas borrando pero no
        ↪ esta:", key); exit(-1);}
        if(t->key == key) merge(t, t->l, t->r);
        else erase(key < t->key ? t->l : t->r,
        ↪ key);
        update(t);
    }
    void showTree(pnode t) { //preOrder, the
    ↪ inOrder is the tree sorted
        if(!t) return;
        cout << "(" << t->key << "," << t->data <<
        ↪ ")";
        showTree(t->l); showTree(t->r);
    }
    // UP CONSTRUCTION, BELOW QUERIES
    pair<V, K> getMaximumAfter(pnode t, K key) {
    ↪ //max node with key >= t, O(log(n))
        if(!t) return MINVK;
        pair<V, K> mid = t->key >= key ?
        ↪ mp(t->data, t->key) : MINVK;
        if(key < t->key) return
        ↪ max({getMaximumAfter(t->l, key),
        ↪ getMaximumVK(t->r), mid}); //maybe
        ↪ equals also here
        else return max({getMaximumAfter(t->r,
        ↪ key), mid});
    }
    pair<V, K> getMinimumBefore(pnode t, K key) {
    ↪ //min node with key <= t, O(log(n))
        if(!t) return MAXVK;
        pair<V, K> mid = t->key <= key ?
        ↪ mp(t->data, t->key): MAXVK;
        if(key > t->key) return
        ↪ min({getMinimumBefore(t->r, key),
        ↪ getMinimumVK(t->l), mid});
        else return min({getMinimumBefore(t->l,
        ↪ key), mid});
    }
    pair<V, K> getMinimumKAll(pnode t) { //is the
    ↪ first node in the inOrder traversal
        if(t->l) return getMinimumKAll(t->l);
        return mp(t->data, t->key);
    }
    public:
    void insert(K key, V data) { //O(log n)
        pnode n = new node(key, data);
        insert(root, n);
    }
    void erase(K key) { //O(log n)
        erase(root, key);
    }
    void showTree() { //debug
        showTree(root);
        cout << endl;
    }
    pair<V, K> getMaximumAfter(K t) { //O(log n)
        return getMaximumAfter(root, t);
    }
    pair<V, K> getMinimumBefore(K t) { //O(log n)
        return getMinimumBefore(root, t);
    }
    pair<V, K> getMaximumVKAll() { //all the tree,
    ↪ O(1)
        return getMaximumVK(root);
    }
    pair<V, K> getMinimumVKAll() { //all the tree,
    ↪ O(1)
        return getMinimumVK(root);
    }
    pair<V, K> getMinimumKAll() { //top of the
    ↪ tree, O(1), check root not null
        return getMinimumKAll(root);
    }
};
```

Factorizator

```
add srand(time(0)); //ADD srand(time(0));
// USE THIS ONLY WHEN NO OTHER OPTION LEFT ...
namespace Factorizator {
    vll primes; //Add primes manually
    const vll fixed_primes = {2, 3, 5, 7, 11, 13,
    ↪ 17, 19, 23, 29, 31, 37, 41}; //47, 53, 59
    //return (a*b)%mod, with numbers up to 1e18
    ↪ (LL)
    ll mult(ll a, ll b, ll mod) {
        //return ((__int128_t)a*b) % mod; //only
        ↪ with 64 bits GCC
        ll ans = 0;
        while(b) {
            if(b&1) ans = (ans+a) % mod;
            b >>=1;
            a = (a+a)%mod;
        }
        return ans;
    }
    ll elevate(ll a, ll b, ll mod){
        ll ans = 1;
        while(b){
            if(b & 1) ans = mult(ans, a, mod);
            b >>= 1;
            a = mult(a, a, mod);
        }
        return ans;
    }
    //a^(mod - 1) = 1, Euler
    ll inv(ll a, ll mod){
        return elevate(((a%mod) + mod)%mod, mod -
        ↪ 2);
    }
    //a^{p-1} = 1 mod p => p divides some factor of
    ↪ (a^{d2^s}+1)*(a^{d2^{s-
    ↪ 1}}+1)*...*(a^d+1)*(a^d-1)
    //return true if the number is composite, false
    ↪ if it is not sure
    bool check_composite(ll num, ll a, ll d, int s)
    ↪ {
```

```
        ll x = elevate(a, d, num);
        if(x == 1 || x == num-1) return false;
        int i;
        for(i = 0; i < s; i++) {
            x = mult(x, x, num);
            if(x == num-1) return false;
        }
        return true;
    }
    //Miller_Rabin DETERMINISTIC Version for num up to 1e18 (all LL)
    bool isPrime(ll num) { //num-1 == d*2^s
        bool flag = true;
        ll d = num-1, s = 0;
        if(num <= 1) return false;
        for(auto p : fixed_primes) {
            if(p == num) return true;
            if(p%num == 0) return false; //optimization
        }
        while(d%2 == 0) {
            d /= 2;
            ++s;
        }
        for(auto p : fixed_primes) flag &= !check_composite(num, p, d, s);
        return flag;
    }
    //a polynomail function modulo mod, it will contain a cycle
    ll f_pollard_rho(ll x, ll c, ll mod) {
        return (mult(x, x, mod)+c) % mod;
    }
    //found a factor (maybe not prime) of num. x0 and c are random, change them if the return is num
    ll pollard_rho(ll num, ll x0, ll c) {
        ll x1 = x0, x2 = x0;
        ll g = 1;
        if(num == 1) return 1;
        if(num%2 == 0) return 2;
        x0 %= num;
        c %= num;
        while(g == 1) { //Floyd cycle detection
            x1 = f_pollard_rho(x1, c, num);
            x2 = f_pollard_rho(x2, c, num);
            x2 = f_pollard_rho(x2, c, num);
            g = __gcd(abs(x1 - x2), num);
        }
        return g;
    }
    vector<pll> factors; // .fi is the prime, .se is the exponent.
    vector<pll> factorize(ll num) {
        factors.clear();
        ll y = num, cont;
        while(num > 1) {
            if(isPrime(num)) y = num;
            else {
                y = pollard_rho(num, rand(), rand());
                while(isPrime(y) == false) y = pollard_rho(y, rand(), rand());
            }
            cont = 0;
            while(num % y == 0) {
                num /= y; cont++;
            }
            factors.pb(mp(y, cont));
        }
        sort(factors.begin(), factors.end());
        return factors;
    }
    vll divisors; //will save all the divisors
    void dfs_div(ll x, ll i) {
        if(i == (int)factors.size()) {divisors.pb(x); return;}
        dfs_div(x, i+1);
        int j;
        for(j = 0; j < factors[i].se; j++) {x*= factors[i].fi; dfs_div(x, i+1);}
    }
    //NOT TESTED
    vector<ll> get_divisors(ll num) { //1 and num inclusive
        if(factors.empty()) factorize(num);
        divisors.clear();
        dfs_div(1, 0);
        sort(divisors.begin(), divisors.end());
        return divisors;
    }
    //factorize knowing that its primes are in primes vector
    vector<pll> factorize_using_primes(ll num) {
        vector<pll> ans;
        ll i, cont;
        for(i = 0; i < (int)primes.size() && primes[i] <= num/primes[i]; i++) {
            if(num%primes[i] == 0) {
                cont = 0;
                while(num%primes[i] == 0) {
                    num /= primes[i];
                    cont++;
                }
                ans.pb(mp(primes[i], cont));
            }
        }
        if(num > 1) ans.pb(mp(num, 1));
        return ans;
    }
    //empty if gcd = 1
    vector<pll> gcd(vector<pll> &va, vector<pll> &vb) {
        vector<pll> ans;
        int l = 0, r = 0, va_sz = (int)va.size(), vb_sz = (int)vb.size();
        while(l < va_sz && r < vb_sz) {
            if(va[l].fi == vb[r].fi)
                {ans.pb(mp(va[l].fi, min(va[l].se, vb[r].se))); ++l; ++r;}
            else if(va[l].fi < vb[r].fi) ++l;
            else ++r;
        }
        return ans;
    }
}
```