Listings

Listing 1: **DATA STRUCTURES**

Listing 2: Count Rectangles

```cpp
#pragma once

//given a 2D boolean matrix, calculate cnt[i][j]
//cnt[i][j] = the number of times an (i * j) rectangle appears in the matrix
//such that all cells in the rectangle are false
//Note cnt[0][j] and cnt[i][0] will contain garbage values
//O(R*C)
//
//status: tested on random inputs
vector<vector<int>> getNumRectangles(const vector<vector<bool>>& grid) {
    vector<vector<int>> cnt;
    const int rows = grid.size(), cols = grid[0].size();
    if (rows == 0 || cols == 0) return cnt;
    cnt.resize(rows + 1, vector<int> (cols + 1, 0));
    vector<vector<int>> arr(rows + 2, vector<int> (cols + 1, 0));
    for (int i = 1; i <= rows; ++i) {
        for (int j = 1; j <= cols; ++j) {
            arr[i][j] = 1 + arr[i][j - 1];
            if (grid[i - 1][j - 1]) arr[i][j] = 0;
        }
    }
    for (int j = 1; j <= cols; ++j) {
        arr[rows + 1][j] = 0;
        stack<pair<int, int>> st;
        st.push({0, 0});
        for (int i = 1; i <= rows + 1; ++i) {
            pair<int, int> curr = {i, arr[i][j]};
            while (arr[i][j] < st.top().second) {
                curr = st.top();
                st.pop();
                cnt[i - curr.first][curr.second]++;
                cnt[i - curr.first][max(arr[i][j], st.top().second)]--;
            }
            st.push({curr.first, arr[i][j]});
        }
    }
    for (int j = 1; j <= cols; ++j) {
        for (int i = rows - 1; i >= 1; --i)
            cnt[i][j] += cnt[i + 1][j];
        for (int i = rows - 1; i >= 1; --i)
            cnt[i][j] += cnt[i + 1][j];
    }
    for (int i = 1; i <= rows; ++i) {
        for (int j = cols - 1; j >= 1; --j)
            cnt[i][j] += cnt[i][j + 1];
    }
    return cnt;
}
```

Listing 3: Disjoint Set

```cpp
#pragma once

//status: tested on random inputs, and on https://judge.yosupo.jp/problem/unionfind
struct disjointSet {
    int numberOfSets;
    vector<int> parent;
    disjointSet(int n) : numberOfSets(n), parent(n, -1) {}
    disjointSet(const disjointSet& rhs) : numberOfSets(rhs.numberOfSets),
        ↪ parent(rhs.parent) {}
    int find(int x) {
        return parent[x] < 0 ? x : parent[x] = find(parent[x]);
    }
    int sizeOfSet(int x) {
        return -parent[find(x)];
    }
    bool merge(int x, int y) {
        if ((x = find(x)) == (y = find(y))) return false;
        if (parent[y] < parent[x]) swap(x, y);
        parent[x] += parent[y];
        parent[y] = x;
        numberOfSets--;
        return true;
    }
};
```

Listing 4: PBDS

```cpp
//status: not tested

//place this include *before* the '#define int long long'
#include <bits/extc++.h>
using namespace __gnu_pbds;

//BST with extra functions https://codeforces.com/blog/entry/11080
//order_of_key - # of elements *strictly* less than given element
//find_by_order - find kth largest element, k is 0 based so find_by_order(0) returns min
    ↪ element
template<class T>
using indexed_set = tree<T, null_type, less<T>, rb_tree_tag,
    ↪ tree_order_statistics_node_update>;
//example initialization:
indexed_set<int> is;


//hash table (apparently faster than unordered_map):
    ↪ https://codeforces.com/blog/entry/60737
//example initialization:
gp_hash_table<int, int> ht;
```

Listing 5: **GRAPHS**

Listing 6: Bridges and Cuts

```cpp
#pragma once

//modified from
    ↪ https://github.com/nealwu/competitive-programming/blob/master/graph_theory/biconnect
//
//status: tested on random graphs
struct biconnected_components {
    //don't pass in a graph with multiple edges between the same pair of nodes - it
        ↪ breaks bridge finding
    biconnected_components(const vector<vector<int>>& adj) :
        is_cut(adj.size(), false),
        n(adj.size()),
        tour_start(n),
        low_link(n) {
        int tour = 0;
        vector<bool> visited(n, false);
        vector<int> stack;
        for (int i = 0; i < n; i++)
            if (!visited[i])
                dfs(i, -1, adj, visited, stack, tour);
    }

    bool is_bridge_edge(int u, int v) const {
        if (u > v) swap(u, v);
        return is_bridge.count(1LL * u * n + v);
    }

    //vector of all bridge edges
    vector<pair<int, int>> bridges;

    //vector of all BCCs. Note a node can be in multiple BCCs (iff it's a cut node)
    vector<vector<int>> components;

    //is_cut['node'] is true iff 'node' is a cut node
    vector<bool> is_cut;


    //use anything below this at your own risk :)
    int n;
    vector<int> tour_start, low_link;
    unordered_set<ll> is_bridge;

    void add_bridge(int u, int v) {
        if (u > v) swap(u, v);
        is_bridge.insert(1LL * u * n + v);
    }

    void dfs(int node, int parent, const vector<vector<int>>& adj, vector<bool>&
        ↪ visited, vector<int>& stack, int& tour) {
        assert(!visited[node]);
        visited[node] = true;
        tour_start[node] = tour++;
        low_link[node] = tour_start[node];
        is_cut[node] = false;
        int parent_count = 0, children = 0;
        for (int next : adj[node]) {
            // Skip the first edge to the parent, but allow multi-edges.
            if (next == parent && parent_count++ == 0)
                continue;
            if (visited[next]) {
                // next is a candidate for low_link.
                low_link[node] = min(low_link[node], tour_start[next]);
                if (tour_start[next] < tour_start[node])
                    stack.push_back(node);
            } else {
                int size = (int) stack.size();
                dfs(next, node, adj, visited, stack, tour);
                children++;
                // next is part of our subtree.
                low_link[node] = min(low_link[node], low_link[next]);
                if (low_link[next] > tour_start[node]) {
                    // This is a bridge.
                    bridges.push_back({node, next});
                    add_bridge(node, next);
                    components.push_back({node, next});
                } else if (low_link[next] == tour_start[node]) {
                    // This is the root of a biconnected component.
                    stack.push_back(node);
                    vector<int> component(stack.begin() + size, stack.end());
                    sort(component.begin(), component.end());
                    component.erase(unique(component.begin(), component.end()),
                        ↪ component.end());
                    components.push_back(component);
                    stack.resize(size);
                } else
                    stack.push_back(node);
                // In general, 'node' is a cut vertex iff it has a child whose subtree
                    ↪ cannot reach above 'node'.
                if (low_link[next] >= tour_start[node])
                    is_cut[node] = true;
            }
        }
        // The root of the tree is a cut vertex iff it has more than one child.
        if (parent < 0) {
            is_cut[node] = children > 1;
            if (children == 0) {
                components.push_back({node});
            }
        }
    }
};


// Note: instead of a block-cut tree this is technically a block-vertex tree, which ends
    ↪ up being much easier to use.
// block-cut tree:
//     nodes for each BCC, and for each cut node
//     edges between a BCC and cut node iff that cut node is in that BCC (so no edges
    ↪ between 2 cut nodes, or 2 BCCs)
//
// block-vertex tree:
//     nodes for each BCC, and for each original node in graph
//     edges between an original node and BCC if that node is inside that BCC
struct block_cut_tree {
    block_cut_tree(const biconnected_components& _bi_comps) :
        n(_bi_comps.n),
        BC(_bi_comps.components.size()),
        T(n + BC),
        block_vertex_tree(T),
        parent(T, -1),
        depth(T) {
        auto add_edge = [&](int a, int b) {
            assert((a < n) ^ (b < n));
            block_vertex_tree[a].push_back(b);
            block_vertex_tree[b].push_back(a);
        };
        for (int bc = 0; bc < BC; bc++)
            for (int x : _bi_comps.components[bc])
                add_edge(x, n + bc);
        for (int root = 0; root < T; root++)
            if (parent[root] < 0)
                dfs(root, -1);
    }

    //If a and b are in the same BCC, this returns the index into
    //biconnected_components::components representing which bcc contains both a,b
    //else returns -1
    //assumes a != b
    int which_bcc(int a, int b) const {
        assert(a != b);
        if (depth[a] > depth[b])
            swap(a, b);
        // Two different nodes are in the same biconnected component iff their distance
          ↪ = 2 in the block-cut tree.
        if ((depth[b] == depth[a] + 2 && parent[parent[b]] == a) || (parent[a] >= 0 &&
          ↪ parent[a] == parent[b]))
            return parent[b] - n;
        return -1;
    }

    //use anything below this at your own risk :)
    int n, BC, T;
    vector<vector<int>> block_vertex_tree;//adjacency list of block vertex tree
    vector<int> parent;
    vector<int> depth;

    void dfs(int node, int par) {
        parent[node] = par;
        depth[node] = par < 0 ? 0 : depth[par] + 1;
        for (int neigh : block_vertex_tree[node])
            if (neigh != par)
                dfs(neigh, node);
    }
};
```

Listing 7: Centroid

```
#pragma once

//status: not tested

const int Max = 2e5 + 2;
vector<int> adj[Max];
int sizes[Max], parent[Max];
bool removed[Max];

void dfs2(int node, int par) {
    sizes[node] = 1;
    for (int to : adj[node]) {
        if (to != par && !removed[to]) {
            dfs2(to, node);
            sizes[node] += sizes[to];
        }
    }
}

int findCentroid(int node) {
    dfs2(node, node);
    bool found = true;
    int sizeCap = sizes[node] / 2;
    int par = node;
    while (found) {
        found = false;
        for (int to : adj[node]) {
            if (to != par && !removed[to] && sizes[to] > sizeCap) {
                found = true;
                par = node;
                node = to;
                break;
            }
        }
    }
    return node;
}

void dfs1(int node, int par) {
    removed[node] = true;
    parent[node] = par;
    for (int to : adj[node]) {
        if (!removed[to])
            dfs1(findCentroid(to), node);
    }
}
//dfs1(findCentroid(1), 0);
```

Listing 8: Count Path Lengths

```
#pragma once

//status: not tested

#include "../math/fft.h"

const int Max = 1e6+10;
int n, sizes[Max];
vector<int> adj[Max], cntPathLength[Max];
ll cntTotalPathLengths[Max];
bool removed[Max];

void dfs2(int node, int par, int root, int currDist) {
    while((int)cntPathLength[root].size() <= currDist) {
        cntPathLength[root].push_back(0);
    }
    cntPathLength[root][currDist]++;
    sizes[node] = 1;
    for(int to : adj[node]) {
        if(to != par && !removed[to]) {
            dfs2(to, node, root, currDist+1);
            sizes[node] += sizes[to];
        }
    }
}

int findCentroid(int node) {
    dfs2(node, node, node, 1);
    bool found = true;
    int sizeCap = sizes[node]/2;
    int par = node;
    while(found) {
        found = false;
        for(int to : adj[node]) {
            if(to != par && !removed[to] && sizes[to] > sizeCap) {
                found = true;
                par = node;
                node = to;
                break;
            }
        }
    }
    return node;
}

void dfs1(int node, int par) {
    removed[node] = true;
    int maxLength = 1;
    for(int to : adj[node]) {
        if(to != par && !removed[to]) {
            cntPathLength[to].clear();
            cntPathLength[to].push_back(0);
            dfs2(to, to, to, 1);
            maxLength = max(maxLength, (int)cntPathLength[to].size());
        }
    }
    vector<int> temp(maxLength, 0);
    temp[0]++;
    for(int to : adj[node]) {
        if(to != par && !removed[to]) {
            vector<ll> prod = multiply(temp, cntPathLength[to]);
            for(int i = 0; i < (int)prod.size(); ++i) {
                cntTotalPathLengths[i] += prod[i];
            }
            for(int i = 0; i < (int)cntPathLength[to].size(); ++i) {
                temp[i] += cntPathLength[to][i];
            }
        }
    }

    for(int to : adj[node]) {
        if(to != par && !removed[to]) {
            dfs1(findCentroid(to), node);
        }
    }
}
```

Listing 9: Dijkstra

```
#pragma once

//returns array `len` where `len[i]` = shortest path from node `startNode` to node i
//For example len[startNode] will always = 0
//
//status: tested on random graphs against floyd warshals, and on
    ↪ https://judge.yosupo.jp/problem/shortest_path

const ll INF = 1e18;

struct dij {
    vector<ll> len;
    vector<int> par;
};

dij dijkstra(const vector<vector<pair<int, ll>>>& adj /*directed or undirected, weighted
    ↪ graph*/, int startNode) {
    vector<ll> len(adj.size(), INF);
    vector<int> par(adj.size(), -1);
    len[startNode] = 0;
    set<pair<ll, int>> q;//weight, node
    q.insert({0, startNode});
    while (!q.empty()) {
        auto it = q.begin();
        const int node = it->second;
        q.erase(it);
        for (auto [to, weight] : adj[node]) {
            if (len[to] > weight + len[node]) {
                q.erase({len[to], to});
                len[to] = weight + len[node];
                par[to] = node;
                q.insert({len[to], to});
            }
        }
    }
    return dij{len, par};
}
```

Listing 10: DSU Tree

```
#pragma once

//status: not tested

const int Max = 1e5 + 3;
int color[Max], Time = 1, timeIn[Max], timeOut[Max], ver[Max], Size[Max], cnt[Max],
    ↪ heavyChild[Max], Depth[Max] = {0}, answer[Max];
vector<int> adj[Max];

void dfs(int node, int prev) {
    timeIn[node] = Time;
    ver[Time] = node;
    Time++;
    Size[node] = 1;
    int largest = heavyChild[node] = -1;
    Depth[node] = 1 + Depth[prev];
    for (int to : adj[node]) {
        if (to == prev) continue;
        dfs(to, node);
        Size[node] += Size[to];
        if (Size[to] > largest) {
            largest = Size[to];
            heavyChild[node] = to;
        }
    }
    timeOut[node] = Time;
}

void dfs1(int node, int prev, bool keep = true) {
    for (int to : adj[node]) {
        if (to == prev || to == heavyChild[node]) continue;
        dfs1(to, node, false);
    }
    if (heavyChild[node] != -1)
        dfs1(heavyChild[node], node, true);
    cnt[color[node]]++;
    for (int to : adj[node]) {
        if (to == prev || to == heavyChild[node]) continue;
        for (int i = timeIn[to]; i < timeOut[to]; ++i)
            cnt[color[ver[i]]]++;
    }
    if (!keep) {
        for (int i = timeIn[node]; i < timeOut[node]; ++i)
            cnt[color[ver[i]]]--;
    }
}
/*
int n;
cin >> n;
dfs(1, 1);
dfs1(1, 1);
for(int i = 1; i <= n; ++i) {
    cout << answer[i] << ' ';
}
cout << '\n';
*/
```

Listing 11: Floyd Warshall

```
#pragma once

//status: tested against output of dijkstras on random graphs

vector<vector<ll>> floydWarshall(const vector<vector<pair<int, ll>>>& adj /*directed or
    ↪ undirected, weighted graph*/) {
    int n = adj.size();
    vector<vector<ll>> len(n, vector<ll> (n, 1e18));
    for (int i = 0; i < n; i++) {
        len[i][i] = 0;//remove this line if you want shortest cycle - len[i][i] will =
            ↪ length of shortest cycle including node i (only for directed graphs)
        for (auto [neighbor, weight] : adj[i])
            len[i][neighbor] = min(len[i][neighbor], weight);
    }
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++)
                len[i][j] = min(len[i][j], len[i][k] + len[k][j]);
        }
    }
    return len;
}
```

Listing 12: HLD

```
#pragma once

//status: all functions tested on random trees; also 'lca' tested on
    ↪ https://judge.yosupo.jp/problem/lca

struct hld {
    vector<int> Size, par, Depth, timeIn, Next, timeInToNode;
    hld(vector<vector<int>>& adj /*forest of trees*/, int root = -1/*pass in to specify
        ↪ root, usually for a single component*/) :
        Size(adj.size(), 1), par(adj.size(), -1), Depth(adj.size(), 1),
            ↪ timeIn(adj.size()), Next(adj.size(), -1), timeInToNode(adj.size()) {
        int Time = 0;
        auto callDfss = [&](int node) -> void {
            Next[node] = par[node] = node;
            dfs1(node, adj);
            dfs2(node, adj, Time);
        };
        if (root != -1)
            callDfss(root);
        for (int i = 0; i < (int) adj.size(); i++) {
            if (par[i] == -1)   //roots each tree by node with min label
                callDfss(i);
        }
    }
    void dfs1(int node, vector<vector<int>>& adj) {
        for (auto& to : adj[node]) {
            if (to == par[node]) continue;
            Depth[to] = 1 + Depth[node];
            par[to] = node;
            dfs1(to, adj);
            Size[node] += Size[to];
            if (Size[to] > Size[adj[node][0]] || adj[node][0] == par[node])
                swap(to, adj[node][0]);
        }
    }
    void dfs2(int node, const vector<vector<int>>& adj, int& Time) {
        timeIn[node] = Time;
        timeInToNode[Time] = node;
        Time++;
        for (auto to : adj[node]) {
            if (to == par[node]) continue;
            Next[to] = (Time == timeIn[node] + 1 ? Next[node] : to);
            dfs2(to, adj, Time);
        }
    }
    // Returns intervals (of timeIn's) corresponding to the path between u and v, not
        ↪ necessarily in order
    // This can answer queries for "is some node 'x' on some path" by checking if the
        ↪ timeIn[x] is in any of these intervals
    vector<pair<int, int>> path(int u, int v) const {
        vector<pair<int, int>> res;
        for (;; v = par[Next[v]]) {
            if (timeIn[v] < timeIn[u]) swap(u, v);
            if (timeIn[Next[v]] <= timeIn[u]) {
                res.push_back({timeIn[u], timeIn[v]});
                return res;
            }
            res.push_back({timeIn[Next[v]], timeIn[v]});
        }
    }
    // Returns interval (of timeIn's) corresponding to the subtree of node i
    // This can answer queries for "is some node 'x' in some other node's subtree" by
        ↪ checking if timeIn[x] is in this interval
    pair<int, int> subtree(int i) const {
        return {timeIn[i], timeIn[i] + Size[i] - 1};
    }
    // Returns lca of nodes u and v
    int lca(int u, int v) const {
        for (;; v = par[Next[v]]) {
            if (timeIn[v] < timeIn[u]) swap(u, v);
            if (timeIn[Next[v]] <= timeIn[u]) return u;
        }
    }
};
```

Listing 13: Hopcroft Karp

```
#pragma once

//Modified from
    ↪ https://github.com/foreverbell/acm-icpc-cheat-sheet/blob/master/src/graph-algorithm/hopcroft-karp.cpp
//Worst case O(E*sqrt(V)) but faster in practice
//status: tested on https://judge.yosupo.jp/problem/bipartitematching with asserts
    ↪ checking correctness of min vertex cover

struct match {
    //# of edges in matching (which = size of min vertex cover by öKnig's theorem)
    int sizeOfMatching;
    //an arbitrary max matching is found. For this matching:
    //if ml[nodeLeft] == -1:
    //    'nodeLeft' is not in matching
    //else:
    //    the edge 'nodeLeft' <=> ml[nodeLeft] is in the matching
    //
    //similarly for mr with edge mr[nodeRight] <=> nodeRight in matching if
        ↪ mr[nodeRight] != -1
    //matchings stored in ml and mr are the same matching
    //provides way to check if any node is in matching
    vector<int> ml, mr;
    //an arbitrary min vertex cover is found. For this MVC: leftMVC['left node'] is true
        ↪ iff 'left node' is in the min vertex cover (same for rightMVC)
    //if leftMVC['left node'] is false, then 'left node' is in the corresponding maximal
        ↪ independent set
    vector<bool> leftMVC, rightMVC;
};

//Think of the bipartite graph as having a left side (with size lSz) and a right side
    ↪ (with size rSz).
//Nodes on left side are indexed 0,1,...,lSz-1
//Nodes on right side are indexed 0,1,...,rSz-1
//
//'adj' is like a directed adjacency list containing edges from left side -> right side:
//To initialize 'adj': For every edge nodeLeft <=> nodeRight, do:
    ↪ adj[nodeLeft].push_back(nodeRight)
match hopcroftKarp(const vector<vector<int>>& adj /*bipartite graph*/) {
    int sizeOfMatching = 0;
    int lSz = adj.size();
    int rSz = 0;
    /****size of mr = 1 + largest right node****/
    for (const vector<int>& v : adj) for (int rhs : v) rSz = max(rSz, rhs + 1);
    vector<int> level(lSz), ml(lSz, -1), mr(rSz, -1);
    vector<bool> visL(lSz, false);
    while (true) {
        queue<int> q;
        for (int i = 0; i < lSz; i++) {
            if (ml[i] == -1) level[i] = 0, q.push(i);
            else level[i] = -1;
        }
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int x : adj[u]) {
                int v = mr[x];
                if (v != -1 && level[v] < 0) {
                    level[v] = level[u] + 1;
                    q.push(v);
                }
            }
        }
        auto dfs = [&](auto&& dfs, int u) -> bool {
            visL[u] = true;
            for (int x : adj[u]) {
                int v = mr[x];
                if (v == -1 || (!visL[v] && level[u] < level[v] && dfs(dfs, v))) {
                    ml[u] = x;
                    mr[x] = u;
                    return true;
                }
            }
            return false;
        };
        visL.assign(lSz, false);
        bool found = false;
        for (int i = 0; i < lSz; i++)
            if (ml[i] == -1 && dfs(dfs, i)) {
                found = true;
                sizeOfMatching++;
            }
        if (!found) break;
    }
    //find min vertex cover
    vector<bool> visR(rSz, false);
    auto dfs = [&](auto&& dfs, int node) -> void {
        for (int to : adj[node]) {
            if (!visR[to] && mr[to] != -1) {
                visR[to] = true;
                dfs(dfs, mr[to]);
            }
        }
    };
    for (int i = 0; i < lSz; i++) {
        visL[i] = !visL[i];
        if (ml[i] == -1)
            dfs(dfs, i);
    }
    return {sizeOfMatching, ml, mr, visL, visR};
}
```

Listing 14: LCA

```
#pragma once

/*
 * Description: Lowest Common Ancestor (LCA) implemented with Binary Lifting.
 * Supports functionality like kth Parent, distance between pair of nodes in
 * number of edges, and distance between pair of nodes in sum of edge weight.
 * Time: O(n * log(n)) for construction and all queries can be answered in
 * O(log(n))
 *
* status: all functions tested on random trees. 'getLca' also tested on
```

```
    ↪ https://judge.yosupo.jp/problem/lca
*/

struct lca {
    typedef long long ll;
    vector<vector<int>> memo;
    vector<int> depth;
    vector<ll> dist;
    int Log;

    // use weights of 1 for unweighted tree
    lca(const vector<vector<pair<int, ll>>>& adj /*connected, weighted tree*/, int root)
        ↪ :
        depth(adj.size()), dist(adj.size()), Log(1) {          //0 - based nodes
        int n = adj.size();
        while ((1 << Log) < n) ++Log;
        memo.resize(n, vector<int> (Log));
        dfs(root, root, adj);
    }

    void dfs(int node, int par, const vector<vector<pair<int, ll>>>& adj) {
        memo[node][0] = par;
        for (int i = 1; i < Log; ++i)
            memo[node][i] = memo[memo[node][i - 1]][i - 1];
        for (auto [to, w] : adj[node]) {
            if (to == par) continue;
            depth[to] = 1 + depth[node];
            dist[to] = w + dist[node];
            dfs(to, node, adj);
        }
    }

    //if k > depth of node, then this returns the root
    int kthPar(int node, int k) const {
        for (int bit = 0; bit < Log; ++bit)
            if (k & (1 << bit))
                node = memo[node][bit];
        return node;
    }

    int getLca(int x, int y) const {
        if (depth[x] < depth[y]) swap(x, y);
        x = kthPar(x, depth[x] - depth[y]);
        if (x == y) return x;
        for (int bit = Log - 1; bit >= 0; --bit)
            if (memo[x][bit] != memo[y][bit]) {
                x = memo[x][bit];
                y = memo[y][bit];
            }
        assert(x != y && memo[x][0] == memo[y][0]);
        return memo[x][0];
    }

    int distEdges(int x, int y) const {
        return depth[x] + depth[y] - 2 * depth[getLca(x, y)];
    }

    ll distWeight(int x, int y) const {
        return dist[x] + dist[y] - 2 * dist[getLca(x, y)];
    }
};
```

Listing 15: SCC

```
#pragma once

//status: tested against floyd warshals on random graphs. also tested on
    ↪ https://judge.yosupo.jp/problem/scc

struct sccInfo {
    //sccId[i] is the id of the scc containing node `i`
    vector<int> sccId;
    //scc's are labeled 0,1,...,`numberOfSCCs-1`
    int numberOfSCCs;
    //adjacency list of "condensation graph", condensation graph is a dag with topo
        ↪ ordering 0,1,...,`numberOfSCCs-1`
    //  - nodes are scc's (labeled by sccId)
    //  - edges: if u -> v exists in original graph, then add edge sccId[u] -> sccId[v]
        ↪ (then remove multiple&self edges)
    vector<vector<int>> adj;
};

sccInfo getSCCs(const vector<vector<int>>& adj /*directed, unweighted graph*/) {
    int n = adj.size();
    sccInfo res;
    res.sccId.resize(n);
    res.numberOfSCCs = 0;
    stack<int> seen;
    {
        vector<bool> vis(n, false);
        auto dfs = [&](auto&& dfsPtr, int curr) -> void {
            vis[curr] = true;
            for (int x : adj[curr]) {
                if (!vis[x])
                    dfsPtr(dfsPtr, x);
            }
            seen.push(curr);
        };
        for (int i = 0; i < n; ++i) {
            if (!vis[i])
                dfs(dfs, i);
        }
    }
    vector<vector<int>> adjInv(n);
    for (int i = 0; i < n; ++i) {
        for (int to : adj[i])
            adjInv[to].push_back(i);
    }
    vector<bool> vis(n, false);
    auto dfs = [&](auto&& dfsPtr, int curr) -> void {
        vis[curr] = true;
        res.sccId[curr] = res.numberOfSCCs;
        for (int x : adjInv[curr]) {
            if (!vis[x])
                dfsPtr(dfsPtr, x);
        }
    };
    while (!seen.empty()) {
        int node = seen.top();
        seen.pop();
        if (vis[node])
            continue;
        dfs(dfs, node);
        res.numberOfSCCs++;
    }
    // This condensation graph building part is not tested
    res.adj.resize(res.numberOfSCCs);
    for (int i = 0; i < n; i++) {
        for (int j : adj[i]) {
            int sccI = res.sccId[i], sccJ = res.sccId[j];
            if (sccI != sccJ) {
                assert(sccI < sccJ);    //sanity check for topo order
                res.adj[sccI].push_back(sccJ);
            }
        }
    }
    for (vector<int>& nexts : res.adj) {
        sort(nexts.begin(), nexts.end());
        nexts.erase(unique(nexts.begin(), nexts.end()), nexts.end());
    }
    return res;
}
```

Listing 16: **HASH**

Listing 17: Safe Hash

```
struct custom_hash {
    static uint64_t splitmix64(uint64_t x) {
        // http://xorshift.di.unimi.it/splitmix64.c
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }

    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM =
            ↪ chrono::steady_clock::now().time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM);
    }
};
```

Listing 18: **MATH**

Listing 19: Count Primes

```
#pragma once

//status: not tested

const int MAX = 1000005;
bool prime[MAX];
int prec[MAX];
vector<int> P;

ll rec(ll N, int K) {
    if (N <= 1 || K < 0) return 0;
    if (N <= P[K]) return N - 1;
    if (N < MAX && ll(P[K]) *P[K] > N) return N - 1 - prec[N] + prec[P[K]];
    const int LIM = 250;
    static int memo[LIM * LIM][LIM];
    bool ok = N < LIM * LIM;
    if (ok && memo[N][K]) return memo[N][K];
    ll ret = N / P[K] - rec(N / P[K], K - 1) + rec(N, K - 1);
    if (ok) memo[N][K] = ret;
    return ret;
}

// init_count_primes();
// count_primes(x);
// Time complexity: Around O(N ^ 0.75)
// Constants to configure:
// - MAX is the maximum value of sqrt(N) + 2
// increase MAX to increase time efficiency
ll count_primes(ll N) {
    if (N < MAX) return prec[N];
    int K = prec[(int) sqrt(N) + 1];
    return N - 1 - rec(N, K) + prec[P[K]];
}

void init_count_primes() {
    prime[2] = true;
    for (int i = 3; i < MAX; i += 2) prime[i] = true;
    for (int i = 3; i * i < MAX; i += 2)
        if (prime[i])
            for (int j = i * i; j < MAX; j += i + i)
                prime[j] = false;
    for (int i = 0; i < MAX; ++i) if (prime[i]) P.push_back(i);
    for (int i = 1; i < MAX; ++i) prec[i] = prec[i - 1] + prime[i];
}
```

Listing 20: BIN EXP MOD

```
#pragma once

//status: tested on random inputs, and used in misc. problems

//returns a^pw % mod in O(log(pw))
ll fastPow(ll a, ll pw, int mod) {
    ll res = 1;
    a %= mod;
    while (pw > 0) {
        if (pw & 1) res = (res * a) % mod;
        a = (a * a) % mod;
        pw >>= 1;
    }
    return res;
}
```

Listing 21: Fibonacci

```
#pragma once

//status: not tested
```

```
const int mod = 1e9 + 7;
unordered_map<ll, ll> table;
int fib(int n) {    //**O(log(n))**
    if (n < 2) return 1;
    if (table.find(n) != table.end()) return table[n];
    table[n] = (fib((n + 1) / 2) * fib(n / 2) + fib((n - 1) / 2) * fib((n - 2) / 2)) %
        ↪ mod;
    return table[n];
}
```

Listing 22: Matrix Mult and Pow

```
#pragma once

//status: not tested, but used on misc. problems

const int mod = 1e9 + 7;

vector<vector<int>> mult(const vector<vector<int>>& a, const vector<vector<int>>& b) {
    if (a.size() == 0) return {};
    if (a[0].size() == 0) return {};
    if (b.size() == 0) return {};
    if (b[0].size() == 0) return {};
    if (a[0].size() != b.size()) return {};
    int resultRow = a.size(), resultCol = b[0].size(), n = a[0].size();
    vector<vector<int>> product(resultRow, vector<int> (resultCol, 0));
    for (int i = 0; i < resultRow; ++i) {
        for (int k = 0; k < n; ++k) {
            for (int j = 0; j < resultCol; ++j)
                product[i][j] = (product[i][j] + 1LL * a[i][k] * b[k][j]) % mod;
        }
    }
    return product;
}

vector<vector<int>> power(vector<vector<int>> matrix, int b) {
    vector<vector<int>> res(matrix.size(), vector<int> (matrix.size(), 0));
    for (int i = 0; i < (int) matrix.size(); i++)
        res[i][i] = 1;
    while (b > 0) {
        if (b % 2 == 1)
            res = mult(res, matrix);
        matrix = mult(matrix, matrix);
        b /= 2;
    }
    return res;
}
```

Listing 23: N Choose K MOD

```
#pragma once

//status: tested on random inputs

#include "exp_mod.h"

// usage:
//     NchooseK nk(n+1, 1e9+7) to use ‘choose‘, ‘inv‘ with inputs <= n
// or:
//     NchooseK nk(mod, mod) to use ‘chooseWithLucasTheorem‘
struct NchooseK {
    // ‘factSz‘ is the size of the factorial array, so only call ‘choose‘, ‘inv‘ with n
        ↪ < factSz
    NchooseK(int factSz, int currMod) : mod(currMod), fact(factSz, 1), invFact(factSz) {
        //this implimentation of doesn’t work if factSz > mod because n! % mod = 0 when
            ↪ n >= mod. So ‘invFact‘ array will be all 0’s
        assert(factSz <= mod);
        //assert mod is prime. mod is intended to fit inside an int so that
        //multiplications fit in a longlong before being modded down. So this
        //will take sqrt(2^31) time
        assert(mod >= 2);
        for (int i = 2; i * i <= mod; i++)
            assert(mod % i);
        for (int i = 1; i < factSz; i++)
            fact[i] = 1LL * fact[i - 1] * i % mod;
        invFact.back() = fastPow(fact.back(), mod - 2, mod);
        for (int i = factSz - 2; i >= 0; i--)
            invFact[i] = 1LL * invFact[i + 1] * (i + 1) % mod;
    }

    //classic n choose k
    //fails when n >= mod
    int choose(int n, int k) const {
        if (k < 0 || k > n) return 0;
        //now we know 0 <= k <= n so 0 <= n
        return 1LL * fact[n] * invFact[k] % mod * invFact[n - k] % mod;
    }

    //lucas theorem to calculate n choose k in O(log(k))
    //need to calculate all factorials in range [0,mod), so O(mod) time&space, so need
        ↪ smallish mod (< 1e6 maybe)
    //handles n >= mod correctly
    int chooseWithLucasTheorem(ll n, ll k) const {
        if (k < 0 || k > n) return 0;
        if (k == 0 || k == n) return 1;
        return 1LL * chooseWithLucasTheorem(n / mod, k / mod) * choose(n % mod, k % mod)
            ↪ % mod;
    }

    //returns inverse of n in O(1)
    int inv(int n) const {
        assert(1 <= n);    //don’t divide by 0 :)
        return 1LL * fact[n - 1] * invFact[n] % mod;
    }

    int mod;
    vector<int> fact, invFact;
};
```

Listing 24: Partition

```
#pragma once

//status: not tested

struct partitionFunction {
    vector<ll> remember;
    //The number of ways you can add to a number
    ll getPartitionsModM(ll n, ll m) {
        if (n < 0) return 0;
        if (n == 0) return 1;
        if ((int) remember.size() <= n) remember.resize(n + 1, -1);
        if (remember[n] != -1) return remember[n];
        ll sum = 0;
        ll val = 1;
        for (ll i = 1; val <= n; i++) {
            ll multiply = 1;
            if (i % 2 == 0) multiply = -1;
            val = ((3 * i * i) + i) / 2;
            sum += getPartitionsModM(n - val, m) * multiply % m;
            val = ((3 * i * i) - i) / 2;
            sum += getPartitionsModM(n - val, m) * multiply % m;
            sum %= m;
            if (sum < 0) sum += m;
        }
        return remember[n] = sum % m;
    }
};
```

Listing 25: Prime Sieve Mobius

```
#pragma once

//status: not tested, but used on various problems

//mobius[i] = 0 iff there exists a prime p s.t. i%(p^2)=0
//mobius[i] = -1 iff i has an odd number of distinct prime factors
//mobius[i] = 1 iff i has an even number of distinct prime factors
const int N = 2e6 + 10;
int mobius[N];
void calcMobius() {
    mobius[1] = 1;
    for (int i = 1; i < N; ++i) {
        for (int j = i + i; j < N; j += i)
            mobius[j] -= mobius[i];
    }
}

int minPrime[N];
void calcSeive() {
    fill(minPrime, minPrime + N, N);
    for (int i = N - 1; i >= 2; --i) {
        for (int j = i; j < N; j += i)
            minPrime[j] = i;
    }
}
```

Listing 26: Solve Linear Equations MOD

```
#pragma once

#include "exp_mod.h"

struct matrixInfo {
    int rank, det;
    vector<int> x;
};

//Solves A * x = b under prime mod.
//A is a n (rows) by m (cols) matrix, b is a length n column vector, x is a length m
    ↪ column vector.
//assumes n,m >= 1, else RTE
//Returns rank of A, determinant of A, and x (solution vector). x is empty if no
    ↪ solution. If multiple solutions, an arbitrary one is returned.
//Leaves A in reduced row echelon form (unlike kactl).
//O(n * m * min(n,m))
//
//status: tested on https://judge.yosupo.jp/problem/system_of_linear_equations and
    ↪ https://judge.yosupo.jp/problem/matrix_det
matrixInfo solve_linear_mod(vector<vector<int>>& A, vector<int>& b, const int mod) {
    assert(A.size() == b.size());
    int n = A.size(), m = A[0].size(), rank = 0, det = 1;
    //start of row reduce
    for (int col = 0; col < m && rank < n; ++col) {
        //find arbitrary pivot and swap pivot to current row
        for (int i = rank; i < n; ++i) if (A[i][col] != 0) {
                if (rank != i) det = det == 0 ? det : mod - det;
                swap(A[i], A[rank]);
                swap(b[i], b[rank]);
                break;
            }
        if (A[rank][col] == 0) {
            det = 0;
            continue;
        }
        det = (1LL * det * A[rank][col]) % mod;
        //make pivot 1 by dividing row by inverse of pivot
        const int aInv = fastPow(A[rank][col], mod - 2, mod);
        for (int j = 0; j < m; ++j)
            A[rank][j] = (1LL * A[rank][j] * aInv) % mod;
        b[rank] = (1LL * b[rank] * aInv) % mod;
        //zero-out all numbers above & below pivot
        for (int i = 0; i < n; ++i) if (i != rank && A[i][col] != 0) {
                const int val = A[i][col];
                for (int j = 0; j < m; ++j) {
                    A[i][j] -= 1LL * A[rank][j] * val % mod;
                    if (A[i][j] < 0) A[i][j] += mod;
                }
                b[i] -= 1LL * b[rank] * val % mod;
                if (b[i] < 0) b[i] += mod;
            }
        ++rank;
    }
    //end of row reduce, start of extracting answer (‘x‘) from ‘A‘ and ‘b‘
    assert(rank <= min(n, m));
    matrixInfo info{rank, det, vector<int>() };
    //check if solution exists
    for (int i = rank; i < n; i++) {
        if (b[i] != 0) return info;   //no solution exists
    }
    info.x.resize(m, 0);
    //initialize solution vector (‘x‘)
    for (int i = 0, j = 0; i < rank; i++) {
        while (A[i][j] == 0) j++;
        assert(A[i][j] == 1);
        info.x[j] = b[i];
    }
    return info;
}
```

Listing 27: Sum Floors of Arithmetic Series

```
#pragma once

//status: used on https://open.kattis.com/problems/itsamodmodmodmodworld

//computes:
//[p/q] + [2p/q] + [3p/q] + ... + [np/q]
//(p, q, n are natural numbers)
//[x] = floor(x)

ll cnt(ll p, ll q, ll n) {
    ll t = __gcd(p, q);
    p = p / t;
    q = q / t;
    ll s = 0;
    ll z = 1;
    while ((q > 0) && (n > 0)) {
        //(point A)
        t = p / q;
        s += z * t * n * (n + 1) / 2;
        p -= q * t;
        //(point B)
        t = n / q;
        s += z * p * t * (n + 1) - z * t * (p * q * t + p + q - 1) / 2;
        n -= q * t;
        //(point C)
        t = n * p / q;
        s += z * t * n;
        n = t;
        swap(p, q);
        z = -z;
    }
    return s;
}
```

Listing 28: Sum of Kth Powers

```
#pragma once

//status: not tested, but used on misc. problems

#define MAX 1000010
#define MOD 1000000007

//Faulhaber'the sum of the k-th powers of the first n positive integers
//1^k + 2^k + 3^k + 4^k + ... + n^k
//O(k*log(k))

//Usage: lgr::lagrange(n, k)

namespace lgr {
short factor[MAX];
int P[MAX], S[MAX], ar[MAX], inv[MAX];

inline int expo(int a, int b) {
    int res = 1;
    while (b) {
        if (b & 1) res = (long long) res * a % MOD;
        a = (long long) a * a % MOD;
        b >>= 1;
    }
    return res;
}

int lagrange(long long n, int k) {
    if (!k) return (n % MOD);
    int i, j, x, res = 0;
    if (!inv[0]) {
        for (i = 2, x = 1; i < MAX; i++) x = (long long) x * i % MOD;
        inv[MAX - 1] = expo(x, MOD - 2);
        for (i = MAX - 2; i >= 0; i--) inv[i] = ((long long) inv[i + 1] * (i + 1)) % MOD;
    }
    k++;
    for (i = 0; i <= k; i++) factor[i] = 0;
    for (i = 4; i <= k; i += 2) factor[i] = 2;
    for (i = 3; (i * i) <= k; i += 2) {
        if (!factor[i]) {
            for (j = (i * i), x = i << 1; j <= k; j += x)
                factor[j] = i;
        }
    }
    for (ar[1] = 1, ar[0] = 0, i = 2; i <= k; i++) {
        if (!factor[i]) ar[i] = expo(i, k - 1);
        else ar[i] = ((long long) ar[factor[i]] * ar[i / factor[i]]) % MOD;
    }
    for (i = 1; i <= k; i++) {
        ar[i] += ar[i - 1];
        if (ar[i] >= MOD) ar[i] -= MOD;
    }
    if (n <= k) return ar[n];
    P[0] = 1, S[k] = 1;
    for (i = 1; i <= k; i++) P[i] = ((long long) P[i - 1] * ((n - i + 1) % MOD)) % MOD;
    for (i = k - 1; i >= 0; i--) S[i] = ((long long) S[i + 1] * ((n - i - 1) % MOD)) %
        ↪ MOD;
    for (i = 0; i <= k; i++) {
        x = (long long) ar[i] * P[i] % MOD * S[i] % MOD * inv[k - i] % MOD * inv[i] %
            ↪ MOD;
        if ((k - i) & 1) {
            res -= x;
            if (res < 0) res += MOD;
        } else {
            res += x;
            if (res >= MOD) res -= MOD;
        }
    }
    return (res % MOD);
}
}
```

Listing 29: Euler's Totient Phi Function

```
#pragma once

//not tested, but used in misc. problems

//  Euler's totient function counts the positive integers
//  up to a given integer n that are relatively prime to n.
ll phi(ll n) {
    ll tempN = n;
    ll result = n;
    for (ll i = 2; i * i <= tempN; i++) {
        if (n % i == 0) {
            while (n % i == 0)
                n /= i;
            result -= result / i;
        }
    }
    if (n > 1) result -= result / n;
    return result;
}
```

Listing 30: **MAX FLOW**

Listing 31: Dinic

```
#pragma once

//status: no tests, but used in various problems

struct maxflow {
public:
    ll n, s, t;
    maxflow(int _n, int _s, int _t) : n(_n), s(_s), t(_t), d(n), ptr(n), q(n), g(n) {}
    void addedge(ll a, ll b, ll cap) {
        edgeMap[a * n + b] = e.size();
        edge e1 = { a, b, cap, 0 };
        edge e2 = { b, a, 0, 0 };
        g[a].push_back((ll) e.size());
        e.push_back(e1);
        g[b].push_back((ll) e.size());
        e.push_back(e2);
    }
    ll getflow() {
        ll flow = 0;
        for (;;) {
            if (!bfs())  break;
            ptr.assign(ptr.size(), 0);
            while (ll pushed = dfs(s, inf))
                flow += pushed;
        }
        return flow;
    }
    ll getFlowForEdge(ll a, ll b) {
        return e[edgeMap[a * n + b]].flow;
    }

private:
    const ll inf = 1e18;
    struct edge {
        ll a, b, cap, flow;
    };
    unordered_map<int, ll> edgeMap;
    vector<ll> d, ptr, q;
    vector<edge> e;
    vector<vector<ll>> g;
    bool bfs() {
        ll qh = 0, qt = 0;
        q[qt++] = s;
        d.assign(d.size(), -1);
        d[s] = 0;
        while (qh < qt && d[t] == -1) {
            ll v = q[qh++];
            for (size_t i = 0; i < g[v].size(); ++i) {
                ll id = g[v][i],
                   to = e[id].b;
                if (d[to] == -1 && e[id].flow < e[id].cap) {
                    q[qt++] = to;
                    d[to] = d[v] + 1;
                }
            }
        }
        return d[t] != -1;
    }
    ll dfs(ll v, ll flow) {
        if (!flow) return 0;
        if (v == t) return flow;
        for (; ptr[v] < (ll) g[v].size(); ++ptr[v]) {
            ll id = g[v][ptr[v]];
            ll to = e[id].b;
            if (d[to] != d[v] + 1)  continue;
            ll pushed = dfs(to, min(flow, e[id].cap - e[id].flow));
            if (pushed) {
                e[id].flow += pushed;
                e[id ^ 1].flow -= pushed;
                return pushed;
            }
        }
        return 0;
    }
};
```

Listing 32: Hungarian

```
#pragma once

const ll inf = 1e18;

// this is one-indexed
// jobs X workers cost matrix
// cost[i][j] is cost of job i done by worker j
// #jobs must be <= #workers
// Default finds min cost; to find max cost set all costs[i][j] to -costs[i][j], set all
     ↪ unused to positive inf, ***set main diagonal (self edges) to 0***

//status: tested on https://judge.yosupo.jp/problem/assignment

struct match {
    ll cost;
    vector<int> matching;
};

match HungarianMatch(const vector<vector<ll>>& cost) {
    ll n = cost.size() - 1;
    ll m = cost[0].size() - 1;
    vector<ll> u(n + 1), v(m + 1), p(m + 1), way(m + 1);
    for (ll i = 1; i <= n; ++i) {
        p[0] = i;
        ll j0 = 0;
        vector<ll> minv(m + 1, inf);
        vector<char> used(m + 1, false);
        do {
            used[j0] = true;
            ll i0 = p[j0], delta = inf, j1 = 0;
            for (ll j = 1; j <= m; ++j)
                if (!used[j]) {
                    ll cur = cost[i0][j] - u[i0] - v[j];
                    if (cur < minv[j])
                        minv[j] = cur, way[j] = j0;
                    if (minv[j] < delta)
                        delta = minv[j], j1 = j;
                }
            for (ll j = 0; j <= m; ++j)
                if (used[j])
                    u[p[j]] += delta, v[j] -= delta;
                else
                    minv[j] -= delta;
            j0 = j1;
        } while (p[j0] != 0);
        do {
            ll j1 = way[j0];
            p[j0] = p[j1];
            j0 = j1;
        } while (j0);
    }
    // For each N, it contains the M it selected
    vector<int> ans(n + 1);
    for (ll j = 1; j <= m ; ++j)
        ans[p[j]] = j;
    return {-v[0], ans};
}
```

Listing 33: Min Cost Max Flow

```
#pragma once

//status: not tested, but used in various problems

const ll inf = 1e18;

struct mincostmaxflow {
    struct edge {
        ll a, b, cap, cost, flow;
        size_t back;
    };

    vector<edge> e;
    vector<vector<ll>> g;
    ll n, s, t;
    ll k = inf; // The maximum amount of flow allowed

    mincostmaxflow(int _n, int _s, int _t) : n(_n), s(_s), t(_t) {
        g.resize(n);
    }

    void addedge(ll a, ll b, ll cap, ll cost) {
        edge e1 = {a, b, cap, cost, 0, g[b].size() };
        edge e2 = {b, a, 0, -cost, 0, g[a].size() };
        g[a].push_back((ll) e.size());
        e.push_back(e1);
        g[b].push_back((ll) e.size());
        e.push_back(e2);
    }

    // Returns {flow,cost}
    pair<ll, ll> getflow() {
        ll flow = 0, cost = 0;
        while (flow < k) {
            vector<ll> id(n, 0);
            vector<ll> d(n, inf);
            vector<ll> q(n);
            vector<ll> p(n);
            vector<size_t> p_edge(n);
            ll qh = 0, qt = 0;
            q[qt++] = s;
            d[s] = 0;
            while (qh != qt) {
                ll v = q[qh++];
                id[v] = 2;
                if (qh == n) qh = 0;
                for (size_t i = 0; i < g[v].size(); ++i) {
                    edge& r = e[g[v][i]];
                    if (r.flow < r.cap && d[v] + r.cost < d[r.b]) {
                        d[r.b] = d[v] + r.cost;
                        if (id[r.b] == 0) {
                            q[qt++] = r.b;
                            if (qt == n) qt = 0;
                        } else if (id[r.b] == 2) {
                            if (--qh == -1) qh = n - 1;
                            q[qh] = r.b;
                        }
                        id[r.b] = 1;
                        p[r.b] = v;
                        p_edge[r.b] = i;
                    }
                }
            }
            if (d[t] == inf) break;
            ll addflow = k - flow;
            for (ll v = t; v != s; v = p[v]) {
                ll pv = p[v];
                size_t pr = p_edge[v];
                addflow = min(addflow, e[g[pv][pr]].cap - e[g[pv][pr]].flow);
            }
            for (ll v = t; v != s; v = p[v]) {
                ll pv = p[v];
                size_t pr = p_edge[v], r = e[g[pv][pr]].back;
                e[g[pv][pr]].flow += addflow;
                e[g[v][r]].flow -= addflow;
                cost += e[g[pv][pr]].cost * addflow;
            }
            flow += addflow;
        }
        return {flow, cost};
    }
};
```

Listing 34: RANGE DATA STRUCTURES

Listing 35: Buckets

```
#pragma once

//this code isn't the best. It's meant as a rough start for sqrt-decomposition, and to
    ↪ be (heavily) modified
//doesn't handle overflow

//status: tested on random inputs, also used in various problems
struct buckets {
    const int BUCKET_SIZE = 50;//TODO: change - small value for testing

    struct bucket {
        int sumLazy = 0;
        int sumBucket = 0;
        int l, r;//inclusive range of bucket
        int len() const {
            return r - l + 1;
        }
    };

    vector<int> values;
    vector<bucket> _buckets;

    buckets(const vector<int>& initial) : values(initial) {
        int numBuckets = ((int) values.size() + BUCKET_SIZE - 1) / BUCKET_SIZE;
        _buckets.resize(numBuckets);
        for (int i = 0; i < numBuckets; i++) {
            _buckets[i].sumLazy = 0;
            _buckets[i].sumBucket = 0;
            _buckets[i].l = i * BUCKET_SIZE;
            _buckets[i].r = min((i + 1) * BUCKET_SIZE, (int) values.size()) - 1;
            for (int j = _buckets[i].l; j <= _buckets[i].r; j++)
                _buckets[i].sumBucket += values[j];
        }
    }

    void pushLazy(int bIdx) {
        bucket& b = _buckets[bIdx];
        if (!b.sumLazy) return;
        for (int i = b.l; i <= b.r; i++)
            values[i] += b.sumLazy;
        b.sumLazy = 0;
    }

    //update range [L,R]
    void update(int L, int R, int diff) {
        int startBucket = L / BUCKET_SIZE;
        int endBucket = R / BUCKET_SIZE;
        if (startBucket == endBucket) {   //range contained in same bucket case
            for (int i = L; i <= R; i++) {
                values[i] += diff;
                _buckets[startBucket].sumBucket += diff;
            }
            return;
        }
        for (int bIdx : {
                    startBucket, endBucket
                }) {  //handle "endpoint" buckets
            bucket& b = _buckets[bIdx];
            for (int i = max(b.l, L); i <= min(b.r, R); i++) {
                values[i] += diff;
                b.sumBucket += diff;
            }
        }
        for (int i = startBucket + 1; i < endBucket; i++) {   //handle all n/B buckets
            ↪ in middle
            bucket& b = _buckets[i];
            b.sumLazy += diff;
            b.sumBucket += b.len() * diff;
        }
    }

    //sum of range [L,R]
    int query(int L, int R) {
        int startBucket = L / BUCKET_SIZE;
        int endBucket = R / BUCKET_SIZE;
        if (startBucket == endBucket) {   //range contained in same bucket case
            pushLazy(startBucket);
            int sum = 0;
            for (int i = L; i <= R; i++)
                sum += values[i];
            return sum;
        }
        int sum = 0;
        for (int bIdx : {
                    startBucket, endBucket
                }) {  //handle "endpoint" buckets
            bucket& b = _buckets[bIdx];
            pushLazy(bIdx);
            for (int i = max(b.l, L); i <= min(b.r, R); i++)
                sum += values[i];
        }
        for (int i = startBucket + 1; i < endBucket; i++)   //handle all n/B buckets in
            ↪ middle
            sum += _buckets[i].sumBucket;
        return sum;
    }
};
```

Listing 36: Fenwick Tree

```
#pragma once

//status: tested on random inputs; also tested on
    ↪ https://judge.yosupo.jp/problem/point_add_range_sum

template<class T>
struct fenwickTree {
    vector<T> bit;
    fenwickTree(int n) : bit(n, 0) {}
    fenwickTree(const vector<T>& a) : bit(a.size()) {
        if (a.empty()) return;
        bit[0] = a[0];
        for (int i = 1; i < (int) a.size(); i++)
```

```
            bit[i] = bit[i - 1] + a[i];
        for (int i = (int) a.size() - 1; i > 0; i--) {
            int lower_i = (i & (i + 1)) - 1;
            if (lower_i >= 0)
                bit[i] -= bit[lower_i];
        }
    }
    void update(int idx, const T& d) {
        for (; idx < (int) bit.size(); idx = idx | (idx + 1))
            bit[idx] += d;
    }
    T sum(int r) const {
        T ret = 0;
        for (; r >= 0; r = (r & (r + 1)) - 1)
            ret += bit[r];
        return ret;
    }
    T sum(int l, int r) const {
        return sum(r) - sum(l - 1);
    }
};

//status: tested on random inputs

template<class T>
struct rangeUpdatesAndPointQueries {
    fenwickTree<T> ft;
    rangeUpdatesAndPointQueries(int n) : ft(n) {}
    rangeUpdatesAndPointQueries(vector<T> arr) : ft(0) {
        for (int i = (int) arr.size() - 1; i >= 1; i--)
            arr[i] -= arr[i - 1];
        ft = fenwickTree<T> (arr);
    }
    void updateRange(int l, int r, const T& diff) {
        ft.update(l, diff);
        if (r + 1 < (int) ft.bit.size())
            ft.update(r + 1, -diff);
    }
    T queryIdx(int idx) const {
        return ft.sum(idx);
    }
};
```

Listing 37: Merge Sort Tree

```
#pragma once

//status: not tested, but used in various problems

struct MergeSortTree {
    struct Node {
        vector<int> vals;
    };
    vector<Node> tree;
    int n, size;

    /*implement these*/
    Node combine(const Node& L, const Node& R) {
        Node par;
        //merge from merge sort
        int ptrL = 0, ptrR = 0;
        while (ptrL < (int) L.vals.size() && ptrR < (int) R.vals.size()) {
            int valL = L.vals[ptrL];
            int valR = R.vals[ptrR];
            if (valL > valR) {
                par.vals.push_back(valR);
                ptrR++;
            } else if (valL < valR) {
                par.vals.push_back(valL);
                ptrL++;
            } else {
                par.vals.push_back(valL);
                par.vals.push_back(valL);
                ptrL++, ptrR++;
            }
        }
        while (ptrL < (int) L.vals.size())
            par.vals.push_back(L.vals[ptrL++]);
        while (ptrR < (int) R.vals.size())
            par.vals.push_back(R.vals[ptrR++]);
        return par;
    }

    MergeSortTree(const vector<int>& arr) : n((int) arr.size()) {
        size = 1;
        while (size < n) size <<= 1;
        size <<= 1;
        tree.resize(size);
        build(arr, 1, 0, n - 1);
    }
    void build(const vector<int>& arr, int node, int start, int end) {
        if (start == end)
            tree[node].vals.push_back(arr[start]);
        else {
            int mid = (start + end) / 2;
            build(arr, 2 * node, start, mid);
            build(arr, 2 * node + 1, mid + 1, end);
            tree[node] = combine(tree[2 * node], tree[2 * node + 1]);
        }
    }
    //returns how many values of arr[l], arr[l+1], ..., arr[r] which are < x
    int query(int l, int r, int x) {
        return query(1, 0, n - 1, l, r, x);
    }
    int query(int node, int start, int end, int l, int r, int x) {
        if (r < start || end < l) return 0;
        if (l <= start && end <= r) {
            auto& v = tree[node].vals;
            return lower_bound(v.begin(), v.end(), x) - v.begin();
        }
        int mid = (start + end) / 2;
        return query(2 * node, start, mid, l, r, x) + query(2 * node + 1, mid + 1, end,
            ↪ l, r, x);
    }
};
```

Listing 38: Mos Algorithm

```
//status: not tested, but used in various problems
```

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

const int Max = 1e6+2;
int block, answer[Max], answerToQuery;

struct query {
    int l, r, index;
};

bool cmp(query x, query y) {
    if(x.l/block == y.l/block) return x.r < y.r;
    return x.l < y.l;
}
void add(int pos) {
}
void remove(int pos) {
}

int main() {
    int q;
    cin >> q;
    vector<query> queries(q);
    for(int i = 0; i < q; ++i) {
        cin >> queries[i].l >> queries[i].r;
        queries[i].index = i;
        answer[i] = 0;
    }
    sort(queries.begin(), queries.end(), cmp);
    int left = 0, right = 0;//store inclusive ranges, start at [0,0]
    add(0);
    answerToQuery = 0;
    for(auto& q : queries) {
        while(left > q.l) {
            left--;
            add(left);
        }
        while(right < q.r) {
            right++;
            add(right);
        }
        while(left < q.l) {
            remove(left);
            left++;
        }
        while(right > q.r) {
            remove(right);
            right--;
        }
        answer[q.index] = answerToQuery;
    }
    for(int i = 0; i < q; ++i) cout << answer[i] << '\n';
    return 0;
}
```

Listing 39: Persistent Segment Tree

```cpp
#pragma once

//modified from k-th smallest section of
    ↪ https://cp-algorithms.com/data_structures/segment_tree.html
//
//status: tested on random inputs, and used in various problems
struct pst {
public:
    /* Persistent seg tree handle a variety of queries
     * - no updates. For point updates, use either merge sort tree, wavelet tree or
         ↪ sqrt-decomp
     * O(nlogn) time and space to build tree
     */
    pst(const vector<int>& arr) : sorted(arr), n(arr.size()) {
        nodes.reserve(4 * n * log(n + 1));
        roots.reserve(n + 1);
        nodes.push_back(Node(Node::Data()));       //acts as nullptr
        sort(sorted.begin(), sorted.end());
        sorted.erase(unique(sorted.begin(), sorted.end()), sorted.end());
        tl = 0, tr = (int) sorted.size() - 1;
        roots.push_back(0);
        for (int val : arr) {
            int idx = lower_bound(sorted.begin(), sorted.end(), val) - sorted.begin();
            roots.push_back(update(roots.back(), tl, tr, idx, val));
        }
    }
    /* find kth smallest number among arr[L], arr[L+1], ..., arr[R]
     * k is 1-based, so find_kth(L,R,1) returns the min
     * O(logn)
     */
    int find_kth(int L, int R, int k) const {
        assert(1 <= k && k <= R - L + 1);    //note this condition implies L <= R
        assert(0 <= L && R < n);
        return sorted[find_kth(roots[L], roots[R + 1], tl, tr, k)];
    }

    /* Among elements arr[L], arr[L+1], ..., arr[R], this returns:
     * the number of elements which is in range [valueL, valueR]
     * */
    int cnt_in_range(int L, int R, int valueL, int valueR) const {
        return calc_in_range(L, R, valueL, valueR).cnt;
    }

    /* Among elements arr[L], arr[L+1], ..., arr[R], this returns:
     * the **sum** of elements which is in range [valueL, valueR]
     * */
    ll sum_in_range(int L, int R, int valueL, int valueR) const {
        return calc_in_range(L, R, valueL, valueR).sum;
    }

    /* Returns sum of min(arr[i], X) for i in range [L,R]
     *
     * assumes -1e9 <= arr[i] <= 1e9 for each i
     * */
    ll sum_in_range_min(int L, int R, int X) const {
        return calc_in_range(L, R, -1e9, X).sum + 1LL * calc_in_range(L, R, X + 1,
            ↪ 1e9).cnt * X;
    }

    /* Returns sum of max(arr[i], X) for i in range [L,R]
     *
     * assumes -1e9 <= arr[i] <= 1e9 for each i
     * */
    ll sum_in_range_max(int L, int R, int X) const {
        return calc_in_range(L, R, X, 1e9).sum + 1LL * calc_in_range(L, R, -1e9, X -
            ↪ 1).cnt * X;
    }
private:
    struct Node {
        int lCh, rCh;
        struct Data {
            int cnt = 0;
            ll sum = 0;
        } data;
        Node(Data _data) : lCh(0), rCh(0), data(_data) {}
        Node(int _lCh, int _rCh, const vector<Node>& nodes) : lCh(_lCh), rCh(_rCh),
            ↪ data(combine(nodes[_lCh].data, nodes[_rCh].data)) { }
    };
    static Node::Data combine(const Node::Data& L, const Node::Data& R) {
        return Node::Data {
            L.cnt + R.cnt,
            L.sum + R.sum
        };
    }
    vector<Node> nodes;
    vector<int> roots, sorted;
    int allocateNode(const Node& v) {
        nodes.push_back(v);
        return (int) nodes.size() - 1;
    }
    int tl, tr, n;
    int update(int v, int l, int r, int pos, int val) {
        if (l == r) {
            return allocateNode(Node({nodes[v].data.cnt + 1, nodes[v].data.sum + val}));
        }
        int m = (l + r) / 2;
        if (pos <= m)
            return allocateNode(Node(update(nodes[v].lCh, l, m, pos, val), nodes[v].rCh,
                ↪ nodes));
        return allocateNode(Node(nodes[v].lCh, update(nodes[v].rCh, m + 1, r, pos, val),
            ↪ nodes));
    }
    int find_kth(int vl, int vr, int l, int r, int k) const {
        if (l == r)
            return l;
        int m = (l + r) / 2, left_count = nodes[nodes[vr].lCh].data.cnt -
            ↪ nodes[nodes[vl].lCh].data.cnt;
        if (left_count >= k) return find_kth(nodes[vl].lCh, nodes[vr].lCh, l, m, k);
        return find_kth(nodes[vl].rCh, nodes[vr].rCh, m + 1, r, k - left_count);
    }
    Node::Data calc_in_range(int L, int R, int valueL, int valueR) const {
        assert(L <= R && valueL <= valueR);
        int compL = lower_bound(sorted.begin(), sorted.end(), valueL) - sorted.begin();
        int compR = (int)(upper_bound(sorted.begin(), sorted.end(), valueR) -
            ↪ sorted.begin()) - 1;
        if (compL > compR) return Node::Data();
        return calc_in_range(roots[L], roots[R + 1], tl, tr, compL, compR);
    }
    Node::Data calc_in_range(int vl, int vr, int l, int r, int valueL, int valueR) const
        ↪ {
        if (valueR < l || r < valueL) return Node::Data();
        if (valueL <= l && r <= valueR) return {nodes[vr].data.cnt - nodes[vl].data.cnt,
            ↪ nodes[vr].data.sum - nodes[vl].data.sum};
        int m = (l + r) / 2;
        return combine(
                    calc_in_range(nodes[vl].lCh, nodes[vr].lCh, l, m, valueL, valueR),
                    calc_in_range(nodes[vl].rCh, nodes[vr].rCh, m + 1, r, valueL, valueR)
                );
    }
};
```

Listing 40: Segment Tree Beats

```
#pragma once

//status: not tested, used in various problems

struct SegTreeBeats {
    struct Node {
        ll sum;
        ll mx;
        ll secondMx;
        ll cntMx;
    };
    vector<Node> tree;
    vector<int> lazy;
    int n, size;
    const ll inf = 1e18;

    /*implement these*/
    const Node zero = {0, -inf, -inf, 0};
    Node combine(const Node& L, const Node& R) {
        Node par;
        par.sum = L.sum + R.sum;
        if (L.mx == R.mx)
            par.cntMx = L.cntMx + R.cntMx;
        else if (L.mx > R.mx)
            par.cntMx = L.cntMx;
        else
            par.cntMx = R.cntMx;
        par.mx = max(L.mx, R.mx);
        par.secondMx = -inf;
        for (ll val : {
                    L.mx, R.mx, L.secondMx, R.secondMx
                }) {
            if (par.mx != val) {
                assert(par.mx > val);
                par.secondMx = max(par.secondMx, val);
            }
        }
        return par;
    }
    void push(int node, int start, int end) {
        if (start == end) return;
        assert(start < end);
        for (int child : {
                    2 * node, 2 * node + 1
                }) {
            if (tree[child].mx <= tree[node].mx) continue;
            tree[child].sum -= (tree[child].mx - tree[node].mx) * tree[child].cntMx;
            tree[child].mx = tree[node].mx;
        }
    }

    SegTreeBeats(const vector<int>& arr) : n((int) arr.size()) {
        size = 1;
        while (size < n) size <<= 1;
        size <<= 1;
        tree.resize(size);
        lazy.resize(size, 0);
        build(arr, 1, 0, n - 1);
    }
    void build(const vector<int>& arr, int node, int start, int end) {
        if (start == end) {
            tree[node].sum = arr[start];
            tree[node].mx = arr[start];
            tree[node].secondMx = -inf;
            tree[node].cntMx = 1;
        } else {
            const int mid = (start + end) / 2;
            build(arr, 2 * node, start, mid);
            build(arr, 2 * node + 1, mid + 1, end);
            tree[node] = combine(tree[2 * node], tree[2 * node + 1]);
        }
    }
    //set a[i] = min(a[i], newMn), for i in range: [l,r]
    void update(int l, int r, int newMn) {
        update(1, 0, n - 1, l, r, newMn);
    }
    void update(int node, int start, int end, int l, int r, int newMn) {
        assert(start <= end);
        push(node, start, end);
        if (start > r || end < l || tree[node].mx <= newMn) return;
        if (start >= l && end <= r && tree[node].secondMx < newMn) {
            tree[node].sum -= (tree[node].mx - newMn) * tree[node].cntMx;
            tree[node].mx = newMn;
            return;
        }
        assert(start < end);
        const int mid = (start + end) / 2;
        update(2 * node, start, mid, l, r, newMn);
        update(2 * node + 1, mid + 1, end, l, r, newMn);
        tree[node] = combine(tree[2 * node], tree[2 * node + 1]);
    }
    //query for sum/max in range [l,r]
    Node query(int l, int r) {
        return query(1, 0, n - 1, l, r);
    }
    Node query(int node, int start, int end, int l, int r) {
        if (r < start || end < l) return zero;
        push(node, start, end);
        if (l <= start && end <= r) return tree[node];
        const int mid = (start + end) / 2;
        return combine(query(2 * node, start, mid, l, r), query(2 * node + 1, mid + 1,
            ↪ end, l, r));
    }
};
```

Listing 41: Segment Tree

```
#pragma once

//status: tested on random inputs

const ll inf = 1e18;

struct SegmentTree {
    struct Node {
        ll sum = 0;
        ll mx = -inf;
        ll mn = inf;

        int l, r;

        ll lazy = 0;
    };

    //change to unordered_map<int, Node> for implicit seg tree. Although I've TLE'd
        ↪ multiple times doing this
    vector<Node> tree;
    int n;

    /***implement these***/
    Node combineChildren(const Node& L, const Node& R) {
        return Node {
            L.sum + R.sum,
            max(L.mx, R.mx),
            min(L.mn, R.mn),
            L.l,
            R.r
            //Note lazy value initializes to 0 here which is ok since we always push
                ↪ lazy down&reset before combining back up
        };
    }
    //what happens when delta is applied to every index in range [start,end]?
    void applyDeltaOnRange(int node, ll delta) {
        int start = tree[node].l, end = tree[node].r;
        tree[node].sum += (end - start + 1) * delta;
        tree[node].mx += delta;
        tree[node].mn += delta;
        if (start != end) {
            tree[2 * node].lazy += delta;
            tree[2 * node + 1].lazy += delta;
        }
    }
    //apply lazy value to range
    void pushLazy(int node) {
        ll& currLazy = tree[node].lazy;
        if (currLazy) {
            applyDeltaOnRange(node, currLazy);
            currLazy = 0;
        }
    }
    /*********************/

    SegmentTree(const vector<ll>& arr) : n((int) arr.size()) {
        auto build = [&](auto&& buildPtr, int node, int start, int end) -> void {
            if (start == end) {
                tree[node] = Node {
                    arr[start],
                    arr[start],
                    arr[start],
                    start,
                    end
                };
            } else {
                int mid = (start + end) / 2;
                buildPtr(buildPtr, 2 * node, start, mid);
                buildPtr(buildPtr, 2 * node + 1, mid + 1, end);
                tree[node] = combineChildren(tree[2 * node], tree[2 * node + 1]);
            }
        };
        int size = 1;
        while (size < n) size <<= 1;
        size <<= 1;
        tree.resize(size);
        build(build, 1, 0, n - 1);
    }
    //inclusive range: [l,r]
    void update(int l, int r, ll diff) {
        auto update = [&](auto&& updatePtr, int node) -> void {
            pushLazy(node);
            int start = tree[node].l, end = tree[node].r;
            if (r < start || end < l) return;
            if (l <= start && end <= r) {
                applyDeltaOnRange(node, diff);
                return;
            }
            updatePtr(updatePtr, 2 * node);
            updatePtr(updatePtr, 2 * node + 1);
            tree[node] = combineChildren(tree[2 * node], tree[2 * node + 1]);
        };
        update(update, 1);
    }
    //inclusive range: [l,r]
    Node query(int l, int r) {
        auto query = [&](auto&& queryPtr, int node) -> Node {
            int start = tree[node].l, end = tree[node].r;
            if (r < start || end < l) return Node();   //l,r is uninitialized -> access
                ↪ means undefined behavior
            pushLazy(node);
            if (l <= start && end <= r) return tree[node];
            return combineChildren(
                queryPtr(queryPtr, 2 * node),
                queryPtr(queryPtr, 2 * node + 1)
            );
        };
        return query(query, 1);
    }
};
```

Listing 42: Sparse Table

```
#pragma once

//usage:
//  sparseTable<ll> st(arr, [](ll x, ll y) { return min(x,y); });
//
//to also get index of min element, do:
//  sparseTable<pair<ll,int>> st(arr, [](auto x, auto y) { return min(x,y); });
//and initialize second to index. If there are multiple indexes of min element,
//it'll return the smallest (left-most) one
//
//status: tested on random inputs, also on https://judge.yosupo.jp/problem/staticrmq
template <class T>
struct sparseTable {
    vector<int> log2;
    vector<vector<T>> dp;
    function<T(const T&, const T&) > func;
    sparseTable(const vector<T>& arr, const function<T(const T&, const T&) >& _func) :
        ↪ func(_func) {
        const int n = arr.size();
        log2.resize(n + 1, -1);
        for (int i = 1; i <= n; ++i) log2[i] = 1 + log2[i / 2];
        dp.resize(log2[n] + 1, arr);
        for (int i = 1; i <= log2[n]; ++i) {
            for (int j = 0; j + (1 << i) - 1 < n; ++j)
                dp[i][j] = func(dp[i - 1][j], dp[i - 1][j + (1 << (i - 1))]);
        }
    }
    //returns func of arr[l], arr[l+1], ..., arr[r]
    T query(int l, int r) const {
        const int x = log2[r - l + 1];
        return func(dp[x][l], dp[x][r - (1 << x) + 1]);
    }
};
```

Listing 43: **RANGE DATA STRUCTURES**

Listing 44: KMP

```
#pragma once

//usage:
//  KMP_Match<string> kmp(needle);
//or
//  KMP_Match<vector<int>> kmp(needle);
//
//status: tested on random inputs
template <class T>
struct KMP_Match {
public:
    KMP_Match(const T& needle_) : prefixFunction(needle_.size() + 1, 0), needle(needle_)
        ↪ {
        for (int i = 1, p = 0; i < (int) needle.size(); i++) {
            update(needle[i], p);
            prefixFunction[i + 1] = p;
        }
    };

    // if haystack = "bananas"
    // needle = "ana"
    //
    // then we find 2 matches:
    // bananas
    // _ana___
    // ___ana_
    // 0123456 (indexes)
    // and KMP_Match::find returns {1,3} - the indexes in haystack where
    // each match starts.
    //
    // You can also pass in false for "all" and KMP_Match::find will only
    // return the first match: {1}. Useful for checking if there exists
    // some match:
```

```
    //
    // KMP_Match::find(<haystack>,false).size() > 0
    vector<int> find(const T& haystack, bool all = true) const {
        vector<int> matches;
        for (int i = 0, p = 0; i < (int) haystack.size(); i++) {
            update(haystack[i], p);
            if (p == (int) needle.size()) {
                matches.push_back(i - (int) needle.size() + 1);
                if (!all) return matches;
                p = prefixFunction[p];
            }
        }
        return matches;
    }
private:
    void update(char val, int& p) const {
        while (p && val != needle[p]) p = prefixFunction[p];
        if (val == needle[p]) p++;
    }
    vector<int> prefixFunction;
    T needle;
};
```

Listing 45: Rolling Hash

```
#pragma once

//usage:
//  Hash<string> h(s);
//or
//  Hash<vector<int>> h(arr);
//  assumes 0 < arr[i] < 1e9+5 = `base`
//  here, for negatives or longlongs, compress values to {1,2,...,n} and make sure
    ↪ `base` is > n
//
//status: tested on random inputs, and on https://judge.yosupo.jp/problem/zalgorithm
template <class T>
struct Hash {
    //`base` should be relatively prime with all mods and *strictly* larger than max
        ↪ element
    //larger/smaller `base` *doesn't* change collision odds
    //ideally `base` would be random to avoid getting hacked
    const int base = 1e9 + 5;
    //From C.R.T., this is the same as having a single mod = product of `mods` = n
    //probability of collision is 1/n
    //probability that k unique strings have k unique hashes = (1/n)^k * (n permute k)
        ↪ from birthday paradox
    const vector<int> mods = { (int) 1e9 + 7, (int) 1e9 + 9, (int) 1e9 + 21, (int) 1e9 +
        ↪ 33, (int) 1e9 + 87};
    vector<vector<int>> prefix, powB;

    Hash(const T& s) :
        prefix(mods.size(), vector<int> (s.size() + 1, 0)),
        powB(mods.size(), vector<int> (s.size() + 1, 1)) {
        //negatives may cause trivial collisions, when s[i]%mod = s[j]%mod, but s[i] !=
            ↪ s[j]
        //0's cause trivial collisions: "0" and "00" both hash to 0
        for (auto val : s) assert(0 < val && val < base);
        for (int i = 0; i < (int) mods.size(); i++) {
            for (int j = 0; j < (int) s.size(); j++) {
                powB[i][j + 1] = 1LL * powB[i][j] * base % mods[i];
                prefix[i][j + 1] = (1LL * base * prefix[i][j] + s[j]) % mods[i];
            }
        }
    }

    //returns hashes of substring/subarray [L,R] inclusive, one hash per mod
    vector<int> getHashes(int L, int R) const {
        assert(0 <= L && L <= R && R + 1 < (int) prefix[0].size());
        vector<int> res(mods.size());
        for (int i = 0; i < (int) mods.size(); i++) {
            res[i] = prefix[i][R + 1] - 1LL * prefix[i][L] * powB[i][R - L + 1] %
                ↪ mods[i];
            if (res[i] < 0) res[i] += mods[i];
        }
        return res;
    }
};
```

Listing 46: Rotational Equivalence

```
#pragma once

// Checks if two arrays are rotationally equivalent
// uses KMP with doubling trick
// usage:
//  rot_eq<string>(s1, s2)
// or
//  rot_eq<vector<int>>(arr1, arr2)
//
//status: tested on random inputs, also on https://open.kattis.com/problems/maze

template <class T>
bool rot_eq(const T& a, const T& b) {
    if (a.size() != b.size()) return false;
    if (a.empty()) return true;
    int n = a.size();
    vector<int> fail(n + 1, 0);
    auto update = [&](int val, int& p) -> void {
        while (p && val != a[p]) p = fail[p];
        if (val == a[p]) p++;
    };
    for (int i = 1, p = 0; i < n; i++) {
        update(a[i], p);
        fail[i + 1] = p;
    }
    for (int i = 0, p = 0; i < 2 * n; i++) {
        update(b[i % n], p);
        if (p == n) return true;
    }
    return false;
}
```

Listing 47: Suffix Array

```
#pragma once

//modified from here: https://judge.yosupo.jp/submission/37410
//
```

```cpp
//status: tested on random inputs, and on
    ↪ https://open.kattis.com/problems/automatictrading
class suffix_array {
public:
    //computes suffix array, lcp array, and then sparse table over lcp array
    //O(n log n)
    suffix_array(const string& s) {
        int n = (int) s.size();
        vector<int> arr(n);
        for (int i = 0; i < n; i++)
            arr[i] = s[i];
        sa_ = sa_is(arr, 255);
        inv_sa_.resize(n);
        for (int i = 0; i < n; i++)
            inv_sa_[sa_[i]] = i;
        lcp_ = lcp_array(arr, sa_);
        init_min_sparse_table(lcp_);
    }

    //length of longest common prefix of suffixes s[idx1..n], s[idx2..n], 0-based
        ↪ indexing
    //You can check if two substrings s[L1..R1], s[L2..R2] are equal in O(1) by:
    //
    //R2-L2 == R1-L1 && suffix_array::longest_common_prefix(L1, L2) >= R2-L2+1
    int longest_common_prefix(int idx1, int idx2) const {
        if (idx1 == idx2) return (int) sa_.size() - idx1;
        idx1 = inv_sa_[idx1];
        idx2 = inv_sa_[idx2];
        if (idx1 > idx2) swap(idx1, idx2);
        int lg = log2_[idx2 - idx1];
        return min(dp_[lg][idx1], dp_[lg][idx2 - (1 << lg)]);
    }

    //returns true if suffix s[idx1..n] < s[idx2..n]
    //(so false if idx1 == idx2)
    //O(1)
    bool less(int idx1, int idx2) const {
        return inv_sa_[idx1] < inv_sa_[idx2];
    }

    vector<int> get_suffix_array() const {
        return sa_;
    }

    vector<int> get_lcp_array() const {
        return lcp_;
    }
private:
    vector<int> sa_, lcp_, inv_sa_, log2_;
    vector<vector<int>> dp_;

    // SA-IS, linear-time suffix array construction
    // Reference:
    // G. Nong, S. Zhang, and W. H. Chan,
    // Two Efficient Algorithms for Linear Time Suffix Array Construction
    vector<int> sa_is(const vector<int>& s, int upper) {
        int n = (int) s.size();
        if (n == 0) return {};
        if (n == 1) return {0};
        if (n == 2) {
            if (s[0] < s[1]) {
                return {0, 1};
            } else {
                return {1, 0};
            }
        }
        vector<int> sa(n);
        vector<bool> ls(n);
        for (int i = n - 2; i >= 0; i--)
            ls[i] = (s[i] == s[i + 1]) ? ls[i + 1] : (s[i] < s[i + 1]);
        vector<int> sum_l(upper + 1), sum_s(upper + 1);
        for (int i = 0; i < n; i++) {
            if (!ls[i])
                sum_s[s[i]]++;
            else
                sum_l[s[i] + 1]++;
        }
        for (int i = 0; i <= upper; i++) {
            sum_s[i] += sum_l[i];
            if (i < upper) sum_l[i + 1] += sum_s[i];
        }
        vector<int> buf(upper + 1);
        auto induce = [&](const vector<int>& lms) {
            fill(sa.begin(), sa.end(), -1);
            fill(buf.begin(), buf.end(), 0);
            copy(sum_s.begin(), sum_s.end(), buf.begin());
            for (auto d : lms) {
                if (d == n) continue;
                sa[buf[s[d]]++] = d;
            }
            copy(sum_l.begin(), sum_l.end(), buf.begin());
            sa[buf[s[n - 1]]++] = n - 1;
            for (int i = 0; i < n; i++) {
                int v = sa[i];
                if (v >= 1 && !ls[v - 1])
                    sa[buf[s[v - 1]]++] = v - 1;
            }
            copy(sum_l.begin(), sum_l.end(), buf.begin());
            for (int i = n - 1; i >= 0; i--) {
                int v = sa[i];
                if (v >= 1 && ls[v - 1])
                    sa[--buf[s[v - 1] + 1]] = v - 1;
            }
        };
        vector<int> lms_map(n + 1, -1);
        int m = 0;
        for (int i = 1; i < n; i++) {
            if (!ls[i - 1] && ls[i])
                lms_map[i] = m++;
        }
        vector<int> lms;
        lms.reserve(m);
        for (int i = 1; i < n; i++) {
            if (!ls[i - 1] && ls[i])
                lms.push_back(i);
        }
        induce(lms);
        if (m) {
            vector<int> sorted_lms;
            sorted_lms.reserve(m);
            for (int v : sa) {
                if (lms_map[v] != -1) sorted_lms.push_back(v);
        }
        vector<int> rec_s(m);
        int rec_upper = 0;
        rec_s[lms_map[sorted_lms[0]]] = 0;
        for (int i = 1; i < m; i++) {
            int l = sorted_lms[i - 1], r = sorted_lms[i];
            int end_l = (lms_map[l] + 1 < m) ? lms[lms_map[l] + 1] : n;
            int end_r = (lms_map[r] + 1 < m) ? lms[lms_map[r] + 1] : n;
            bool same = true;
            if (end_l - l != end_r - r)
                same = false;
            else {
                while (l < end_l) {
                    if (s[l] != s[r])
                        break;
                    l++;
                    r++;
                }
                if (l == n || s[l] != s[r]) same = false;
            }
            if (!same) rec_upper++;
            rec_s[lms_map[sorted_lms[i]]] = rec_upper;
        }
        auto rec_sa =
            sa_is(rec_s, rec_upper);
        for (int i = 0; i < m; i++)
            sorted_lms[i] = lms[rec_sa[i]];
        induce(sorted_lms);
    }
    return sa;
}

// Reference:
// T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park,
// Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its
// Applications
vector<int> lcp_array(const vector<int>& s, const vector<int>& sa) {
    int n = s.size(), k = 0;
    vector<int> lcp(n, 0);
    vector<int> rank(n, 0);
    for (int i = 0; i < n; i++) rank[sa[i]] = i;
    for (int i = 0; i < n; i++, k ? k-- : 0) {
        if (rank[i] == n - 1) {
            k = 0;
            continue;
        }
        int j = sa[rank[i] + 1];
        while (i + k < n && j + k < n && s[i + k] == s[j + k]) k++;
        lcp[rank[i]] = k;
    }
    return lcp;
}

void init_min_sparse_table(const vector<int>& arr) {
    const int n = arr.size();
    log2_.resize(n + 1, -1);
    for (int i = 1; i <= n; ++i) log2_[i] = 1 + log2_[i / 2];
    dp_.resize(log2_[n] + 1, arr);
    for (int i = 1; i <= log2_[n]; ++i) {
        for (int j = 0; j + (1 << i) - 1 < n; ++j)
            dp_[i][j] = min(dp_[i - 1][j], dp_[i - 1][j + (1 << (i - 1))]);
```

```
        }
    }
};
```

Listing 48: Trie

```
#pragma once

//status: not tested, but used on various problems

const int K = 26;//character size

struct node {
    bool leaf = 0;
    int next[K], id, p = -1;
    char pch;
    node(int _p = -1, char ch = '#') : p(_p), pch(ch) {
        fill(next, next + K, -1);
    }
};

vector<node> t(1);    //adj list

void add_string(const string& s, int id) {
    int c = 0;
    for (char ch : s) {
        int v = ch - 'a';
        if (t[c].next[v] == -1) {
            t[c].next[v] = t.size();
            t.emplace_back(c, ch);
        }
        c = t[c].next[v];
    }
    t[c].leaf = 1;
    t[c].id = id;
}

void remove_string(const string& s) {
    int c = 0;
    for (char ch : s) {
        int v = ch - 'a';
        if (t[c].next[v] == -1)
            return;
        c = t[c].next[v];
    }
    t[c].leaf = 0;
}

int find_string(const string& s) {
    int c = 0;
    for (char ch : s) {
        int v = ch - 'a';
        if (t[c].next[v] == -1)
            return -1;
        c = t[c].next[v];
    }
    if (!t[c].leaf) return -1;
    return t[c].id;
}
```