Listings

1 **Contest** 2
2 Hash codes 2
3 Test on random inputs 2
4 **GRAPHS** 2
5 Bridges and Cuts 2
6 Block Vertex Tree 3
7 Bridge Tree 3
8 Centroid 3
9 Dijkstra 4
10 Floyd Warshall 4
11 HLD 4
12 Hopcroft Karp 5
13 LCA 5
14 SCC 6
15 **RANGE DATA STRUCTURES** 6
16 Segment Tree 6
17 BIT 7
18 RMQ 8
19 Implicit Lazy Segment Tree 8
20 Range Updates, Point Queries 8
21 Kth Smallest 9
22 Number Distinct Elements 9
23 Buckets 10
24 Persistent Lazy Segment Tree 10
25 Merge Sort Tree 11
26 **STRINGS** 12
27 Suffix Array 12
28 LCP 13
29 Prefix Function 13
30 KMP 13
31 Trie 14
32 Binary Trie 14
33 Longest Common Prefix Query 15
34 **MATH** 15
35 BIN EXP MOD 15
36 Fibonacci 15
37 Matrix Mult and Pow 15
38 N Choose K MOD 16
39 Partitions 16
40 Derangements 16
41 Prime Sieve Mobius 17
42 Row Reduce 17
43 Solve Linear Equations MOD 17
44 Matrix Inverse 18
45 Euler's Totient Phi Function 18
46 **MAX FLOW** 18
47 Dinic 18
48 Hungarian 19
49 Min Cost Max Flow 19
50 **MISC** 20
51 DSU 20
52 PBDS 20
53 Monotonic Stack 20
54 Count Rectangles 21
55 LIS 21
56 Safe Hash 21

Listing 1: **Contest**

Listing 2: Hash codes

```
#!/usr/bin/env bash
#Hashes a file, ignoring all:
#   - whitespace
#   - comments
#   - asserts
#   - includes
#   - pragmas
#Use to verify that code was correctly typed.

#usage:
#   chmod +x hash.sh
#   cat <file> | ./hash.sh
#or just copy this command:
#   cat <file> | sed -r ’/(assert|include|pragma)/d’ | cpp -fpreprocessed -P | tr -d
    ↪ ’[:space:]’ | md5sum | cut -c-6
sed -r ’/(assert|include|pragma)/d’ | cpp -fpreprocessed -P | tr -d ’[:space:]’ | md5sum
    ↪ | cut -c-6
```

Listing 3: Test on random inputs

```
#!/usr/bin/env bash
#runs 2 programs against each other on random inputs until they output different results
#usage:
#   chmod +x test.sh
#   ./test.sh
for((i = 1; ; ++i)); do
    echo $i
    ./test.out > in
    diff --ignore-all-space <(./a.out < in) <(./brute.out < in) || break
done
```

Listing 4: **GRAPHS**

Listing 5: Bridges and Cuts

```
//cat bridges_and_cuts.h | ./hash.sh
//34dc49
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/biconnected_components,
    ↪ https://judge.yosupo.jp/problem/two_edge_connected_components
//with asserts checking correctness of is_bridge and is_cut
//O(n+m) time & space
//2 edge cc and bcc stuff doesn’t depend on each other, so delete whatever is not needed
//handles multiple edges
//
//example initialization of ‘adj‘:
//for (int i = 0; i < m; i++) {
//  int u, v;
//  cin >> u >> v;
//  u--, v--;
//  adj[u].emplace_back(v, i);
//  adj[v].emplace_back(u, i);
//}
struct info {
    //2 edge connected component stuff (e.g. components split by bridge edges)
        ↪ https://cp-algorithms.com/graph/bridge-searching.html
    int num_2_edge_ccs;
    vector<bool> is_bridge;//edge id -> true iff bridge edge
    vector<int> two_edge_ccid;//node -> id of 2 edge component (which are labeled 0, 1,
        ↪ ..., ‘num_2_edge_ccs‘-1)
    //bi connected component stuff (e.g. components split by cut/articulation nodes)
        ↪ https://cp-algorithms.com/graph/cutpoints.html
    int num_bccs;
    vector<bool> is_cut;//node -> true iff cut node
    vector<int> bcc_id;//edge id -> id of bcc (which are labeled 0, 1, ..., ‘num_bccs‘-1)
};
info bridge_and_cut(const vector<vector<pair<int/*neighbor*/, int/*edge id*/>>>&
    ↪ adj/*undirected graph*/, int m/*number of edges*/) {
    //stuff for both (always keep)
    int n = adj.size(), timer = 1;
    vector<int> tin(n, 0);
    //2 edge cc stuff (delete if not needed)
    int num_2_edge_ccs = 0;
    vector<bool> is_bridge(m, false);
    vector<int> two_edge_ccid(n), node_stack;
    //bcc stuff (delete if not needed)
    int num_bccs = 0;
    vector<bool> is_cut(n, false);
    vector<int> bcc_id(m), edge_stack;
    auto dfs = [&](auto self, int v, int p_id) -> int {
        int low = tin[v] = timer++, deg = 0;
        node_stack.push_back(v);
        for (auto [to, e_id] : adj[v]) {
            if (e_id == p_id) continue;
            if (!tin[to]) {
                edge_stack.push_back(e_id);
                int low_ch = self(self, to, e_id);
                if (low_ch >= tin[v]) {
                    is_cut[v] = true;
                    while (true) {
                        int edge = edge_stack.back();
                        edge_stack.pop_back();
                        bcc_id[edge] = num_bccs;
                        if (edge == e_id) break;
                    }
                    num_bccs++;
                }
                low = min(low, low_ch);
                deg++;
            } else if (tin[to] < tin[v]) {
                edge_stack.push_back(e_id);
                low = min(low, tin[to]);
            }
        }
        if (p_id == -1) is_cut[v] = (deg > 1);
        if (tin[v] == low) {
            if (p_id != -1) is_bridge[p_id] = true;
            while (true) {
                int node = node_stack.back();
                node_stack.pop_back();
                two_edge_ccid[node] = num_2_edge_ccs;
                if (node == v) break;
            }
            num_2_edge_ccs++;
        }
        return low;
    };
    for (int i = 0; i < n; i++)
        if (!tin[i])
            dfs(dfs, i, -1);
    return {num_2_edge_ccs, is_bridge, two_edge_ccid, num_bccs, is_cut, bcc_id};
}
```

Listing 6: Block Vertex Tree

```
//cat block_vertex_tree.h | ./hash.sh
//a28ab2
#pragma once
#include "bridges_and_cuts.h"
//library checker tests: https://judge.yosupo.jp/problem/biconnected_components
//(asserts checking correctness of commented-example-usage-loops)
//returns adjacency list of block vertex tree
//usage:
//  info cc = bridge_and_cut(adj, m);
//  vector<vector<int>> bvt = block_vertex_tree(adj, cc);
//to loop over each *unique* bcc containing a node v:
//  for(int bccid : bvt[v]) {
//      bccid -= n;
//      ...
//  }
//to loop over each *unique* node inside a bcc:
//  for(int v : bvt[bccid + n]) {
//      ...
//  }
vector<vector<int>> block_vertex_tree(const vector<vector<pair<int, int>>>& adj, const
     ↪ info& cc) {
    int n = adj.size();
    vector<vector<int>> tree(n + cc.num_bccs);
    vector<int> cnt(cc.num_bccs, 0);
    for (int v = 0; v < n; v++) {
        for (auto [_, e_id] : adj[v]) {
            int bccid = cc.bcc_id[e_id];
            if (cnt[bccid]++ == 0) {
                tree[v].push_back(bccid + n);// add edge between original node, and bcc
                     ↪ node
                tree[bccid + n].push_back(v);
            }
        }
        for (auto [_, e_id] : adj[v])
            cnt[cc.bcc_id[e_id]]--;
    }
    return tree;
}
```

Listing 7: Bridge Tree

```
//cat bridge_tree.h | ./hash.sh
//85f56b
#pragma once
#include "bridges_and_cuts.h"
//library checker tests: https://judge.yosupo.jp/problem/two_edge_connected_components
//never adds multiple edges as bridges_and_cuts.h correctly marks them as non-bridges
//usage:
//  info cc = bridge_and_cut(adj, m);
//  vector<vector<int>> bt = bridge_tree(adj, cc);
vector<vector<int>> bridge_tree(const vector<vector<pair<int, int>>>& adj, const info&
     ↪ cc) {
    vector<vector<int>> tree(cc.num_2_edge_ccs);
    for (int i = 0; i < (int)adj.size(); i++)
        for (auto [to, e_id] : adj[i])
            if (cc.is_bridge[e_id])
                tree[cc.two_edge_ccid[i]].push_back(cc.two_edge_ccid[to]);
    return tree;
}
```

Listing 8: Centroid

```
//cat centroid.h | ./hash.sh
//4ba5e4
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/frequency_table_of_tree_distance
//with asserts checking depth of tree <= log2(n)
//returns array `par` where `par[i]` = parent of node `i` in centroid tree
//`par[root]` is -1
//0-based nodes
//O(n log n)
//example usage:
//  vector<int> parent = get_centroid_tree(adj);
//  vector<vector<int>> childs(n);
//  int root;
//  for (int i = 0; i < n; i++) {
//      if (parent[i] == -1)
//          root = i;
//      else
//          childs[parent[i]].push_back(i);
//  }
vector<int> get_centroid_tree(const vector<vector<int>>& adj/*unrooted tree*/) {
    int n = adj.size();
    vector<int> sizes(n);
    vector<bool> vis(n, false);
    auto dfs_sz = [&](auto self, int node, int par) -> void {
        sizes[node] = 1;
        for (int to : adj[node]) {
            if (to != par && !vis[to]) {
                self(self, to, node);
                sizes[node] += sizes[to];
            }
        }
    };
    auto find_centroid = [&](int node) -> int {
        dfs_sz(dfs_sz, node, node);
        int size_cap = sizes[node] / 2, par = -1;
        while (true) {
            bool found = false;
            for (int to : adj[node]) {
                if (to != par && !vis[to] && sizes[to] > size_cap) {
                    found = true;
                    par = node;
                    node = to;
                    break;
                }
            }
            if (!found) return node;
        }
    };
    vector<int> parent(n);
    auto dfs = [&](auto self, int node, int par) -> void {
        node = find_centroid(node);
        parent[node] = par;
        vis[node] = true;
        for (int to : adj[node]) {
            if (!vis[to])
                self(self, to, node);
        }
    };
    dfs(dfs, 0, -1);
    return parent;
}
```

Listing 9: Dijkstra

```
//cat dijkstra.h | ./hash.sh
//6b6195
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/shortest_path
//returns array ‘len‘ where ‘len[i]‘ = shortest path from node v to node i
//For example len[v] will always = 0
const long long inf = 1e18;
vector<long long> dijkstra(const vector<vector<pair<int, long long>>>& adj /*directed or
    ↪ undirected, weighted graph*/, int v) {
    vector<long long> len(adj.size(), inf);
    len[v] = 0;
    set<pair<long long/*weight*/, int/*node*/>> q;
    q.insert({0LL, v});
    while (!q.empty()) {
        auto it = q.begin();
        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];
                q.insert({len[to], to});
            }
    }
    return len;
}
```

Listing 10: Floyd Warshall

```
//cat floyd_warshall.h | ./hash.sh
//84799a
#pragma once
//status: not tested
//**for directed graphs only** if you initialize len[i][i] to infinity, then
//afterward floyds, len[i][i] = length of shortest cycle including node ‘i‘
//
//another trick: change ‘len‘ to 2d array of *bools* where len[i][j] = true if
//there exists an edge from i -> j in initial graph. Also do:
//‘len[i][j] = len[i][j] | (len[i][k] & len[k][j])‘
//Then after floyds, len[i][j] = true iff there's exists some path from node
//‘i‘ to node ‘j‘
//
//Changing the order of for-loops to i-j-k (instead of the current k-i-j)
//results in min-plus matrix multiplication. If adjacency matrix is ‘mat‘, then
//after computing mat^k (with binary exponentiation), mat[i][j] = min length path
//from i to j with at most k edges.
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]);
```

Listing 11: HLD

```
//cat hld.h | ./hash.sh
//103dab
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/lca,
    ↪ https://judge.yosupo.jp/problem/vertex_add_path_sum,
    ↪ https://judge.yosupo.jp/problem/vertex_add_subtree_sum
//source: https://codeforces.com/blog/entry/53170
//assumes a single tree, 1-based nodes is possible by passing in ‘root‘ in range [1, n]
//mnemonic: Heavy Light Decomposition
//NOLINTNEXTLINE(readability-identifier-naming)
struct HLD {
    int n;
    vector<int> sub_sz, par, time_in, next;
    HLD(vector<vector<int>>& adj /*single unrooted tree*/, int root) :
        n(adj.size()), sub_sz(n, 1), par(n, root), time_in(n), next(n, root) {
        dfs1(root, adj);
        int timer = 0;
        dfs2(root, adj, timer);
    }
    void dfs1(int node, vector<vector<int>>& adj) {
        for (int& to : adj[node]) {
            if (to == par[node]) continue;
            par[to] = node;
            dfs1(to, adj);
            sub_sz[node] += sub_sz[to];
            if (sub_sz[to] > sub_sz[adj[node][0]] || adj[node][0] == par[node])
                swap(to, adj[node][0]);
        }
    }
    void dfs2(int node, const vector<vector<int>>& adj, int& timer) {
        time_in[node] = timer++;
        for (int to : adj[node]) {
            if (to == par[node]) continue;
            next[to] = (timer == time_in[node] + 1 ? next[node] : to);
            dfs2(to, adj, timer);
        }
    }
    // Returns intervals (of time_in'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
        ↪ time_in[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 (time_in[v] < time_in[u]) swap(u, v);
            if (time_in[next[v]] <= time_in[u]) {
                res.emplace_back(time_in[u], time_in[v]);
                return res;
            }
            res.emplace_back(time_in[next[v]], time_in[v]);
        }
    }
    // Returns interval (of time_in'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 time_in[x] is in this interval
    pair<int, int> subtree(int i) const {
        return {time_in[i], time_in[i] + sub_sz[i] - 1};
    }
    // Returns lca of nodes u and v
    int lca(int u, int v) const {
        for (;; v = par[next[v]]) {
            if (time_in[v] < time_in[u]) swap(u, v);
            if (time_in[next[v]] <= time_in[u]) return u;
        }
    }
};
```

Listing 12: Hopcroft Karp

```
//cat hopcroft_karp.h | ./hash.sh
//8b3f52
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/bipartitematching
//with asserts checking correctness of min vertex cover
//source:
    ↪ 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
struct match {
    //# of edges in matching (which = size of min vertex cover by öKnig's theorem)
    int size_of_matching;
    //an arbitrary max matching is found. For this matching:
    //if l_to_r[node_left] == -1:
    //  node_left is not in matching
    //else:
    //  the edge ‘node_left‘ <=> l_to_r[node_left] is in the matching
    //
    //similarly for r_to_l with edge r_to_l[node_right] <=> node_right in matching if
        ↪ r_to_l[node_right] != -1
    //matchings stored in l_to_r and r_to_l are the same matching
    //provides way to check if any node is in matching
    vector<int> l_to_r, r_to_l;
    //an arbitrary min vertex cover is found. For this mvc: mvc_l[node_left] is true iff
        ↪ node_left is in the min vertex cover (same for mvc_r)
    //if mvc_l[node_left] is false, then node_left is in the corresponding maximal
        ↪ independent set
    vector<bool> mvc_l, mvc_r;
};
//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 node_left <=> node_right, do:
    ↪ adj[node_left].push_back(node_right)
match hopcroft_karp(const vector<vector<int>>& adj/*bipartite graph*/, int rsz/*number
    ↪ of nodes on right side*/) {
    int size_of_matching = 0, lsz = adj.size();
    vector<int> l_to_r(lsz, -1), r_to_l(rsz, -1);
    while (true) {
        queue<int> q;
        vector<int> level(lsz, -1);
        for (int i = 0; i < lsz; i++) {
            if (l_to_r[i] == -1) level[i] = 0, q.push(i);
        }
        bool found = false;
        vector<bool> mvc_l(lsz, true), mvc_r(rsz, false);
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            mvc_l[u] = false;
            for (int x : adj[u]) {
                mvc_r[x] = true;
                int v = r_to_l[x];
                if (v == -1) found = true;
                else if (level[v] < 0) {
                    level[v] = level[u] + 1;
                    q.push(v);
                }
            }
        }
        if (!found) return {size_of_matching, l_to_r, r_to_l, mvc_l, mvc_r};
        auto dfs = [&](auto self, int u) -> bool {
            for (int x : adj[u]) {
                int v = r_to_l[x];
                if (v == -1 || (level[u] + 1 == level[v] && self(self, v))) {
                    l_to_r[u] = x;
                    r_to_l[x] = u;
                    return true;
                }
            }
            level[u] = 1e9; //acts as visited array
            return false;
        };
        for (int i = 0; i < lsz; i++)
            size_of_matching += (l_to_r[i] == -1 && dfs(dfs, i));
    }
}
```

Listing 13: LCA

```
//cat lca.h | ./hash.sh
//90ab04
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/lca
//https://codeforces.com/blog/entry/74847
//assumes a single tree, 1-based nodes is possible by passing in ‘root‘ in range [1, n]
//mnemonic: Least/Lowest Common Ancestor
//NOLINTNEXTLINE(readability-identifier-naming)
struct LCA {
    int n;
    vector<int> jmp, jmp_edges, par, depth;
    vector<long long> dist;
    LCA(const vector<vector<pair<int, long long>>>& adj, int root) :
        n(adj.size()), jmp(n, root), jmp_edges(n, 1), par(n, root), depth(n, 0), dist(n,
            ↪ 0LL) {
        dfs(root, adj);
    }
    void dfs(int node, const vector<vector<pair<int, long long>>>& adj) {
        for (auto [ch, w] : adj[node]) {
            if (ch == par[node]) continue;
            par[ch] = node;
            depth[ch] = 1 + depth[node];
            dist[ch] = w + dist[node];
            if (depth[node] > 0 && jmp_edges[node] == jmp_edges[jmp[node]])
                jmp[ch] = jmp[jmp[node]], jmp_edges[ch] = 2 * jmp_edges[node] + 1;
            else
                jmp[ch] = node;
            dfs(ch, adj);
        }
    }
    //traverse up k edges in O(log(k)). So with k=1 this returns ‘node‘’s parent
    int kth_par(int node, int k) const {
        k = min(k, depth[node]);
        while (k > 0) {
            if (jmp_edges[node] <= k) {
                k -= jmp_edges[node];
                node = jmp[node];
            } else {
                k--;
                node = par[node];
            }
        }
        return node;
    }
    int get_lca(int x, int y) const {
        if (depth[x] < depth[y]) swap(x, y);
        x = kth_par(x, depth[x] - depth[y]);
        while (x != y) {
            if (jmp[x] != jmp[y])
                x = jmp[x], y = jmp[y];
            else
                x = par[x], y = par[y];
        }
        return x;
    }
    int dist_edges(int x, int y) const {
        return depth[x] + depth[y] - 2 * depth[get_lca(x, y)];
    }
    long long dist_weight(int x, int y) const {
        return dist[x] + dist[y] - 2 * dist[get_lca(x, y)];
    }
};
```

Listing 14: SCC

```
//cat scc.h | ./hash.sh
//8fa337
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/scc
//mnemonic: Strongly Connected Component
struct scc_info {
    int num_sccs;
    //scc's are labeled 0,1,...,‘num_sccs-1‘
    //scc_id[i] is the id of the scc containing node ‘i‘
    //for each edge i -> j: scc_id[i] >= scc_id[j] (topo order of scc's)
    vector<int> scc_id;
};
//NOLINTNEXTLINE(readability-identifier-naming)
scc_info SCC(const vector<vector<int>>& adj /*directed, unweighted graph*/) {
    int n = adj.size(), timer = 1, num_sccs = 0;
    vector<int> tin(n, 0), scc_id(n, -1), node_stack;
    auto dfs = [&](auto self, int v) -> int {
        int low = tin[v] = timer++;
        node_stack.push_back(v);
        for (int to : adj[v]) {
            if (scc_id[to] < 0)
                low = min(low, tin[to] ? tin[to] : self(self, to));
        }
        if (tin[v] == low) {
            while (true) {
                int node = node_stack.back();
                node_stack.pop_back();
                scc_id[node] = num_sccs;
                if (node == v) break;
            }
            num_sccs++;
        }
        return low;
    };
    for (int i = 0; i < n; i++) {
        if (!tin[i])
            dfs(dfs, i);
    }
    return {num_sccs, scc_id};
}
```

Listing 15: **RANGE DATA STRUCTURES**

Listing 16: Segment Tree

```
//cat seg_tree.h | ./hash.sh
//644265
#pragma once
//stress tests: tests/stress_tests/range_data_structures/seg_tree.cpp
const long long inf = 1e18;
struct seg_tree {
    struct node {
        long long sum, mx, mn;
        long long lazy;
        int l, r;
        int len() const {
            return r - l + 1;
        }
        //returns 1 + (# of nodes in left child's subtree)
        //https://cp-algorithms.com/data_structures/segment_tree.html#memory-efficient-imple
        int rch() const { //right child
            return (r - l + 2) & ~1;
        }
    };
    vector<node> tree;
    seg_tree(const vector<long long>& arr) : tree(2 * (int)arr.size() - 1) {
        int timer = 0;
        build(arr, timer, 0, (int)arr.size() - 1);
    }
    node build(const vector<long long>& arr, int& timer, int tl, int tr) {
        node& curr = tree[timer++];
        if (tl == tr) {
            curr = {
                arr[tl],
                arr[tl],
                arr[tl],
                0,
                tl,
                tr
            };
        } else {
            int tm = tl + (tr - tl) / 2;
            const node& l = build(arr, timer, tl, tm);
            const node& r = build(arr, timer, tm + 1, tr);
            curr = combine(l, r);
        }
        return curr;
    }
    static node combine(const node& l, const node& r) {
        return {
            l.sum + r.sum,
            max(l.mx, r.mx),
            min(l.mn, r.mn),
            0,
            l.l,
            r.r
        };
    }
    //what happens when `add` is applied to every index in range [tree[v].l, tree[v].r]?
    void apply(int v, long long add) {
        tree[v].sum += tree[v].len() * add;
        tree[v].mx += add;
        tree[v].mn += add;
        if (tree[v].len() > 1) {
            tree[v + 1].lazy += add;
            tree[v + tree[v].rch()].lazy += add;
        }
    }
    void push(int v) {
        if (tree[v].lazy) {
            apply(v, tree[v].lazy);
            tree[v].lazy = 0;
        }
    }
    //update range [l,r] with `add`
    void update(int l, int r, long long add) {
        update(0, l, r, add);
    }
    void update(int v, int l, int r, long long add) {
        push(v);
        if (tree[v].r < l || r < tree[v].l)
            return;
        if (l <= tree[v].l && tree[v].r <= r)
            return apply(v, add);
        update(v + 1, l, r, add);
        update(v + tree[v].rch(), l, r, add);
        tree[v] = combine(tree[v + 1], tree[v + tree[v].rch()]);
    }
    //range [l,r]
    node query(int l, int r) {
        return query(0, l, r);
    }
    node query(int v, int l, int r) {
        if (tree[v].r < l || r < tree[v].l)
            return {0, -inf, inf, 0, 0, 0};
        push(v);
        if (l <= tree[v].l && tree[v].r <= r)
            return tree[v];
        return combine(query(v + 1, l, r),
                       query(v + tree[v].rch(), l, r));
    }
};
```

Listing 17: BIT

```
//cat bit.h | ./hash.sh
//516197
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/point_add_range_sum,
   ↪ https://judge.yosupo.jp/problem/vertex_add_path_sum,
   ↪ https://judge.yosupo.jp/problem/vertex_add_subtree_sum,
   ↪ https://judge.yosupo.jp/problem/predecessor_problem
//mnemonic: Binary Indexed Tree
template<class T>
struct BIT { //NOLINT(readability-identifier-naming)
    vector<T> bit;
    BIT(int n) : bit(n, 0) {}
    BIT(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);
    }
    //Returns min pos such that sum of [0, pos] >= sum
    //Returns bit.size() if no sum is >= sum, or -1 if empty sum is.
    //Doesn't work with negatives (since it's greedy), counterexample: array: {1, -1},
       ↪ sum: 1, this returns 2, but should return 0
    int lower_bound(T sum) const {
        if (sum <= 0) return -1;
        int pos = 0;
        for (int pw = 1 << (31 - __builtin_clz(bit.size() | 1)); pw; pw >>= 1) {
            if (pos + pw <= (int)bit.size() && bit[pos + pw - 1] < sum)
```

```
                pos += pw, sum -= bit[pos - 1];
        }
        return pos;
    }
};
```

Listing 18: RMQ

```
//cat rmq.h | ./hash.sh
//0266df
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/staticrmq,
    ↪ https://judge.yosupo.jp/problem/zalgorithm,
    ↪ https://judge.yosupo.jp/problem/enumerate_palindromes,
    ↪ https://judge.yosupo.jp/problem/cartesian_tree
//usage:
//  vector<long long> arr;
//  ...
//  RMQ<long long> st(arr, [&](auto x, auto y) { return min(x,y); });
//
//to also get index of min element, do:
//  RMQ<pair<T, int>> st(arr, [&](auto x, auto y) { return min(x,y); });
//and initialize arr[i].second = i (0<=i<n)
//If there are multiple indexes of min element, it'll return the smallest
//(left-most) one
//mnemonic: Range Min/Max Query
template <class T>
struct RMQ { //NOLINT(readability-identifier-naming)
    vector<vector<T>> dp;
    function<T(const T&, const T&)> func;
    RMQ(const vector<T>& arr, const function<T(const T&, const T&)>& a_func) : dp(1,
        ↪ arr), func(a_func) {
        int n = arr.size();
        for (int pw = 1, k = 1; 2 * pw <= n; pw *= 2, k++) {
            dp.emplace_back(n - 2 * pw + 1);
            for (int j = 0; j + 2 * pw - 1 < n; j++)
                dp[k][j] = func(dp[k - 1][j], dp[k - 1][j + pw]);
        }
    }
    //inclusive range [l, r]
    T query(int l, int r) const {
        int lg = 31 - __builtin_clz(r - l + 1);
        return func(dp[lg][l], dp[lg][r - (1 << lg) + 1]);
    }
};
```

Listing 19: Implicit Lazy Segment Tree

```
//cat implicit_seg_tree.h | ./hash.sh
//6fe6ae
#pragma once
//stress tests: tests/stress_tests/range_data_structures/implicit_seg_tree.cpp
//see TODO for lines of code which usually need to change (not a complete list)
const int sz = 1.5e7; //TODO
struct node {
    long long val;//could represent max, sum, etc
    long long lazy;
    int lch, rch; // children, indexes into 'tree', -1 for null
} tree[sz];
struct implicit_seg_tree {
    int ptr, root_l, root_r;//[root_l, root_r] defines range of root node; handles
        ↪ negatives
    implicit_seg_tree(int l, int r) : ptr(0), root_l(l), root_r(r) {
        tree[ptr++] = {0, 0, -1, -1}; //TODO
    }
    static long long combine(long long val_l, long long val_r) {
        return val_l + val_r; //TODO
    }
    void apply(int v, int tl, int tr, long long add) {
        tree[v].val += (tr - tl + 1) * add; //TODO
        if (tl != tr) {
            tree[tree[v].lch].lazy += add; //TODO
            tree[tree[v].rch].lazy += add;
        }
    }
    void push(int v, int tl, int tr) {
        if (tl != tr && tree[v].lch == -1) {
            assert(ptr + 1 < sz);
            tree[v].lch = ptr;
            tree[ptr++] = {0, 0, -1, -1}; //TODO
            tree[v].rch = ptr;
            tree[ptr++] = {0, 0, -1, -1};
        }
        if (tree[v].lazy) {
            apply(v, tl, tr, tree[v].lazy);
            tree[v].lazy = 0;
        }
    }
    //update range [l,r] with 'add'
    void update(int l, int r, long long add) {
        update(0, root_l, root_r, l, r, add);
    }
    void update(int v, int tl, int tr, int l, int r, long long add) {
        push(v, tl, tr);
        if (tr < l || r < tl)
            return;
        if (l <= tl && tr <= r)
            return apply(v, tl, tr, add);
        int tm = tl + (tr - tl) / 2;
        update(tree[v].lch, tl, tm, l, r, add);
        update(tree[v].rch, tm + 1, tr, l, r, add);
        tree[v].val = combine(tree[tree[v].lch].val, tree[tree[v].rch].val);
    }
    //query range [l,r]
    long long query(int l, int r) {
        return query(0, root_l, root_r, l, r);
    }
    long long query(int v, int tl, int tr, int l, int r) {
        if (tr < l || r < tl)
            return 0; //TODO
        push(v, tl, tr);
        if (l <= tl && tr <= r)
            return tree[v].val;
        int tm = tl + (tr - tl) / 2;
        return combine(query(tree[v].lch, tl, tm, l, r),
                       query(tree[v].rch, tm + 1, tr, l, r));
    }
};
```

Listing 20: Range Updates, Point Queries

```
//cat fenwick_inv.h | ./hash.sh
//e1114e
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/vertex_add_subtree_sum,
    ↪ https://judge.yosupo.jp/problem/point_add_range_sum
#include "../bit.h"
template<class T>
struct fenwick_inv {
    BIT<T> ft;
    fenwick_inv(int n) : ft(n) {}
    fenwick_inv(const vector<T>& arr) : ft(init(arr)) {}
    BIT<T> init(vector<T> arr/*intentional pass by value*/) const {
        for (int i = (int)arr.size() - 1; i >= 1; i--)
            arr[i] -= arr[i - 1];
        return BIT<T>(arr);
    }
    //add ‘add‘ to inclusive range [l, r]
    void update(int l, int r, const T& add) {
        ft.update(l, add);
        if (r + 1 < (int)ft.bit.size())
            ft.update(r + 1, -add);
    }
    //get value at index ‘idx‘
    T query(int idx) const {
        return ft.sum(idx);
    }
};
```

Listing 21: Kth Smallest

```
//cat kth_smallest.h | ./hash.sh
//4e859c
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/range_kth_smallest
//source:
    ↪ https://cp-algorithms.com/data_structures/segment_tree.html#preserving-the-history-of-its-values-persistent-segment-tree
struct kth_smallest {
    struct node {
        int sum;
        int lch, rch;//children, indexes into ‘tree‘
    };
    int mn, mx;
    vector<int> roots;
    deque<node> tree;
    kth_smallest(const vector<int>& arr) : mn(INT_MAX), mx(INT_MIN), roots(arr.size() +
        ↪ 1, 0) {
        tree.push_back({0, 0, 0}); //acts as null
        for (int val : arr) mn = min(mn, val), mx = max(mx, val);
        for (int i = 0; i < (int)arr.size(); i++)
            roots[i + 1] = update(roots[i], mn, mx, arr[i]);
    }
    int update(int v, int tl, int tr, int idx) {
        if (tl == tr) {
            tree.push_back({tree[v].sum + 1, 0, 0});
            return tree.size() - 1;
        }
        int tm = tl + (tr - tl) / 2;
        int lch = tree[v].lch;
        int rch = tree[v].rch;
        if (idx <= tm)
            lch = update(lch, tl, tm, idx);
        else
            rch = update(rch, tm + 1, tr, idx);
        tree.push_back({tree[lch].sum + tree[rch].sum, lch, rch});
        return tree.size() - 1;
    }
    /* find (k+1)th smallest number among arr[l], arr[l+1], ..., arr[r]
     * k is 0-based, so query(l,r,0) returns the min
     */
    int query(int l, int r, int k) const {
        assert(0 <= k && k < r - l + 1); //note this condition implies l <= r
        assert(0 <= l && r + 1 < (int)roots.size());
        return query(roots[l], roots[r + 1], mn, mx, k);
    }
    int query(int vl, int vr, int tl, int tr, int k) const {
        if (tl == tr)
            return tl;
        int tm = tl + (tr - tl) / 2;
        int left_count = tree[tree[vr].lch].sum - tree[tree[vl].lch].sum;
        if (left_count > k) return query(tree[vl].lch, tree[vr].lch, tl, tm, k);
        return query(tree[vl].rch, tree[vr].rch, tm + 1, tr, k - left_count);
    }
};
```

Listing 22: Number Distinct Elements

```
//cat distinct_query.h | ./hash.sh
//6bdf2f
#pragma once
//stress tests: tests/stress_tests/range_data_structures/distinct_query.cpp
//source:
    ↪ https://cp-algorithms.com/data_structures/segment_tree.html#preserving-the-history-o
//works with negatives
//O(n log n) time and space
struct distinct_query {
    struct node {
        int sum;
        int lch, rch;//children, indexes into ‘tree‘
    };
    vector<int> roots;
    deque<node> tree;
    distinct_query(const vector<int>& arr) : roots(arr.size() + 1, 0) {
        tree.push_back({0, 0, 0}); //acts as null
        map<int, int> last_idx;
        for (int i = 0; i < (int)arr.size(); i++) {
            roots[i + 1] = update(roots[i], 0, arr.size(), last_idx[arr[i]]);
            last_idx[arr[i]] = i + 1;
        }
    }
    int update(int v, int tl, int tr, int idx) {
        if (tl == tr) {
            tree.push_back({tree[v].sum + 1, 0, 0});
            return tree.size() - 1;
        }
        int tm = (tl + tr) / 2;
        int lch = tree[v].lch;
        int rch = tree[v].rch;
        if (idx <= tm)
            lch = update(lch, tl, tm, idx);
        else
            rch = update(rch, tm + 1, tr, idx);
        tree.push_back({tree[lch].sum + tree[rch].sum, lch, rch});
        return tree.size() - 1;
    }
    //returns number of distinct elements in range [l,r]
    int query(int l, int r) const {
        return query(roots[l], roots[r + 1], 0, (int)roots.size() - 1, l + 1);
    }
    int query(int vl, int vr, int tl, int tr, int idx) const {
        if (tree[vr].sum == 0 || idx <= tl)
            return 0;
        if (tr < idx)
            return tree[vr].sum - tree[vl].sum;
        int tm = (tl + tr) / 2;
        return query(tree[vl].lch, tree[vr].lch, tl, tm, idx) +
               query(tree[vl].rch, tree[vr].rch, tm + 1, tr, idx);
    }
};
```

Listing 23: Buckets

```
//cat buckets.h | ./hash.sh
//435b76
#pragma once
//stress tests: tests/stress_tests/range_data_structures/buckets.cpp
//this code isn't the best. It's meant as a rough start for sqrt_decomposition, and to be modified
//doesn't handle overflow
struct buckets {
    const int bucket_size = 300;//TODO
    struct node {
        int sum_lazy = 0;
        int sum_bucket = 0;
        int l, r;//inclusive range of bucket
        int len() const {
            return r - l + 1;
        }
    };
    vector<int> values;
    vector<node> bucket;
    buckets(const vector<int>& initial) : values(initial) {
        int numbucket = ((int)values.size() + bucket_size - 1) / bucket_size;
        bucket.resize(numbucket);
        for (int i = 0; i < numbucket; i++) {
            bucket[i].sum_lazy = 0;
            bucket[i].sum_bucket = 0;
            bucket[i].l = i * bucket_size;
            bucket[i].r = min((i + 1) * bucket_size, (int)values.size()) - 1;
            for (int j = bucket[i].l; j <= bucket[i].r; j++)
                bucket[i].sum_bucket += values[j];
        }
    }
    void push(int b_idx) {
        node& b = bucket[b_idx];
        if (!b.sum_lazy) return;
        for (int i = b.l; i <= b.r; i++)
            values[i] += b.sum_lazy;
        b.sum_lazy = 0;
    }
    //update range [l,r]
    void update(int l, int r, int diff) {
        int start_bucket = l / bucket_size;
        int end_bucket = r / bucket_size;
        if (start_bucket == end_bucket) { //range contained in same bucket case
            for (int i = l; i <= r; i++) {
                values[i] += diff;
                bucket[start_bucket].sum_bucket += diff;
            }
            return;
        }
        for (int b_idx : {
                    start_bucket, end_bucket
                }) { //handle "endpoint" buckets
            node& b = bucket[b_idx];
            for (int i = max(b.l, l); i <= min(b.r, r); i++) {
                values[i] += diff;
                b.sum_bucket += diff;
            }
        }
        for (int i = start_bucket + 1; i < end_bucket; i++) { //handle all n/bucket_size buckets in middle
            node& b = bucket[i];
            b.sum_lazy += diff;
            b.sum_bucket += b.len() * diff;
        }
    }
    //sum of range [l,r]
    int query(int l, int r) {
        int start_bucket = l / bucket_size;
        int end_bucket = r / bucket_size;
        if (start_bucket == end_bucket) { //range contained in same bucket case
            push(start_bucket);
            int sum = 0;
            for (int i = l; i <= r; i++)
                sum += values[i];
            return sum;
        }
        int sum = 0;
        for (int b_idx : {
                    start_bucket, end_bucket
                }) { //handle "endpoint" buckets
            node& b = bucket[b_idx];
            push(b_idx);
            for (int i = max(b.l, l); i <= min(b.r, r); i++)
                sum += values[i];
        }
        for (int i = start_bucket + 1; i < end_bucket; i++) //handle all n/bucket_size buckets in middle
            sum += bucket[i].sum_bucket;
        return sum;
    }
};
```

Listing 24: Persistent Lazy Segment Tree

```
//cat persistent_lazy_seg_tree.h | ./hash.sh
//5f187b
#pragma once
//status: not tested
```

```
struct persistent_lazy_seg_tree {
    struct node {
        int lch, rch;//children, indexes into ‘tree‘
        int sum;
        bool lazy_tog;
    };
    int sz;
    deque<node> tree;
    vector<int> roots;
    //implicit
    persistent_lazy_seg_tree(int a_sz) : sz(a_sz) {
        tree.push_back({0, 0, 0, 0}); //acts as null
        roots.push_back(0);
    }
    void push(int v, int tl, int tr) {
        if (tl != tr) {
            tree.push_back(tree[tree[v].lch]);
            tree[v].lch = tree.size() - 1;
            tree.push_back(tree[tree[v].rch]);
            tree[v].rch = tree.size() - 1;
        }
        if (tree[v].lazy_tog) {
            tree[v].sum = (tr - tl + 1) - tree[v].sum;
            tree[v].lazy_tog = false;
            if (tl != tr) {
                tree[tree[v].lch].lazy_tog ^= 1;
                tree[tree[v].rch].lazy_tog ^= 1;
            }
        }
    }
    void set(int idx, int new_val) {
        tree.push_back(tree[roots.back()]);//allocate top down
        roots.push_back(tree.size() - 1);
        set(roots.back(), 0, sz - 1, idx, new_val);
    }
    void set(int v, int tl, int tr, int idx, int new_val) {
        push(v, tl, tr);
        if (tr < idx || idx < tl)
            return;
        if (idx <= tl && tr <= idx) {
            tree[v].sum = new_val;
            return;
        }
        int tm = (tl + tr) / 2;
        int lch = tree[v].lch;
        int rch = tree[v].rch;
        set(lch, tl, tm, idx, new_val);
        set(rch, tm + 1, tr, idx, new_val);
        tree[v].sum = tree[lch].sum + tree[rch].sum;
    }
    void toggle_range(int l, int r) {
        tree.push_back(tree[roots.back()]);//allocate top down
        roots.push_back(tree.size() - 1);
        toggle_range(roots.back(), 0, sz - 1, l, r);
    }
    void toggle_range(int v, int tl, int tr, int l, int r) {
        push(v, tl, tr);
        if (tr < l || r < tl)
            return;
        int lch = tree[v].lch;
        int rch = tree[v].rch;
        if (l <= tl && tr <= r) {
            tree[v].sum = (tr - tl + 1) - tree[v].sum;
            if (tl != tr) {
                tree[lch].lazy_tog ^= 1;
                tree[rch].lazy_tog ^= 1;
            }
            return;
        }
        int tm = (tl + tr) / 2;
        toggle_range(lch, tl, tm, l, r);
        toggle_range(rch, tm + 1, tr, l, r);
        tree[v].sum = tree[lch].sum + tree[rch].sum;
    }
    //let's use implementation trick described here
        ↪ https://codeforces.com/blog/entry/72626
    //so that we don't have to propagate lazy vals and thus we don't have to allocate
        ↪ new nodes
    int query(int l, int r) const {
        int version = roots.size() - 1;
        int root = roots[version];
        return query(root, 0, sz - 1, l, r, tree[root].lazy_tog);
    }
    int query(int v, int tl, int tr, int l, int r, bool tog) const {
        if (v == 0 || tr < l || r < tl)
            return 0;
        if (l <= tl && tr <= r) {
            int sum = tree[v].sum;
            if (tree[v].lazy_tog) sum = (tr - tl + 1) - sum;
            return sum;
        }
        int tm = (tl + tr) / 2;
        tog ^= tree[v].lazy_tog;
        return query(tree[v].lch, tl, tm, l, r, tog) +
               query(tree[v].rch, tm + 1, tr, l, r, tog);
    }
};
```

Listing 25: Merge Sort Tree

```
//cat merge_sort_tree.h | ./hash.sh
//9b42ef
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/static_range_frequency,
    ↪ https://judge.yosupo.jp/problem/range_kth_smallest
//For point updates: either switch to policy based BST, or use sqrt decomposition
struct merge_sort_tree {
    struct node {
        vector<int> vals;
        int l, r;
        //returns 1 + (# of nodes in left child's subtree)
        //https://cp-algorithms.com/data_structures/segment_tree.html#memory-efficient-imple
        int rch() const {
            return (r - l + 2) & ~1;
        }
    };
    vector<node> tree;
    //RTE's when ‘arr‘ is empty
    merge_sort_tree(const vector<int>& arr) : tree(2 * (int)arr.size() - 1) {
        int timer = 0;
        build(arr, timer, 0, (int)arr.size() - 1);
    }
    node build(const vector<int>& arr, int& timer, int tl, int tr) {
        node& curr = tree[timer++];
        if (tl == tr) {
            curr = {
                {arr[tl]},
                tl,
                tr
            };
        } else {
            int tm = tl + (tr - tl) / 2;
            const vector<int>& l = build(arr, timer, tl, tm).vals;
            const vector<int>& r = build(arr, timer, tm + 1, tr).vals;
            merge(l.begin(), l.end(), r.begin(), r.end(), back_inserter(curr.vals));
            curr.l = tl, curr.r = tr;
        }
        return curr;
    }
    //How many of arr[l], arr[l+1], ..., arr[r] are < x?
    //O(log^2(n))
    int query(int l, int r, int x) const {
        return query(0, l, r, x);
    }
    int query(int v, int l, int r, int x) const {
        if (tree[v].r < l || r < tree[v].l)
            return 0;
        if (l <= tree[v].l && tree[v].r <= r) {
            const vector<int>& vals = tree[v].vals;
            return lower_bound(vals.begin(), vals.end(), x) - vals.begin();
        }
        return query(v + 1, l, r, x) +
               query(v + tree[v].rch(), l, r, x);
    }
};
```

Listing 26: STRINGS

Listing 27: Suffix Array

```
//cat suffix_array.h | ./hash.sh
//46840a
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/suffixarray,
    ↪ https://judge.yosupo.jp/problem/zalgorithm,
    ↪ https://judge.yosupo.jp/problem/number_of_substrings,
    ↪ https://judge.yosupo.jp/problem/enumerate_palindromes
//source: https://judge.yosupo.jp/submission/37410
//O(n)
//mnemonic: Suffix Array Induced Sorting
template<class T>
vector<int> sa_is(const T& s, int upper/*max element of `s`; for std::string, pass in
    ↪ 255*/) {
    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;
}
```

Listing 28: LCP

```
//cat lcp.h | ./hash.sh
//064842
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/zalgorithm,
    ↪ https://judge.yosupo.jp/problem/number_of_substrings,
    ↪ https://judge.yosupo.jp/problem/enumerate_palindromes
//source: https://judge.yosupo.jp/submission/37410
//mnemonic: Longest Common Prefix
template<class T>
vector<int> LCP(const T& s, const vector<int>& sa) {
    ↪ //NOLINT(readability-identifier-naming)
    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;
}
```

Listing 29: Prefix Function

```
//cat prefix_function.h | ./hash.sh
//aa0518
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/zalgorithm
//stress tests: tests/stress_tests/strings/kmp.cpp
//source: https://cp-algorithms.com/string/prefix-function.html#implementation
template <class T>
vector<int> prefix_function(const T& s) {
    int n = s.size();
    vector<int> pi(n, 0);
    for (int i = 1; i < n; i++) {
        int j = pi[i - 1];
        while (j > 0 && s[i] != s[j]) j = pi[j - 1];
        pi[i] = j + (s[i] == s[j]);
    }
    return pi;
}
```

Listing 30: KMP

```
//cat kmp.h | ./hash.sh
//9d70ad
#pragma once
//stress tests: tests/stress_tests/strings/kmp.cpp
//mnemonic: Knuth Morris Pratt
#include "prefix_function.h"
//usage:
//  string needle;
//  ...
//  KMP kmp(needle);
//or
//  vector<int> needle;
//  ...
//  KMP kmp(needle);
//kmp doubling trick: to check if 2 arrays are rotationally equivalent: run kmp
//with one array as the needle and the other array doubled (excluding the first
//& last characters) as the haystack or just use kactl's min rotation code
template <class T>
struct KMP { //NOLINT(readability-identifier-naming)
    KMP(const T& a_needle) : pi(prefix_function(a_needle)), needle(a_needle) {}
    // if haystack = "bananas"
    // needle = "ana"
    //
    // then we find 2 matches:
    // bananas
    // _ana___
    // ___ana_
    // 0123456 (indexes)
    // and KMP::find returns {1,3} - the indexes in haystack where
    // each match starts.
    //
    // You can also pass in false for "all" and KMP::find will only
    // return the first match: {1}. Useful for checking if there exists
    // some match:
    //
    // KMP::find(<haystack>,false).size() > 0
    vector<int> find(const T& haystack, bool all = true) const {
        vector<int> matches;
        for (int i = 0, j = 0; i < (int)haystack.size(); i++) {
            while (j > 0 && needle[j] != haystack[i]) j = pi[j - 1];
            if (needle[j] == haystack[i]) j++;
            if (j == (int)needle.size()) {
                matches.push_back(i - (int)needle.size() + 1);
                if (!all) return matches;
                j = pi[j - 1];
            }
        }
        return matches;
    }
    vector<int> pi;//prefix function
    T needle;
};
```

Listing 31: Trie

```
//cat trie.h | ./hash.sh
//6c97ea
#pragma once
//status: not tested
//source: https://cp-algorithms.com/string/aho_corasick.html#construction-of-the-trie
//intended to be a base template and to be modified
const int k = 26;//alphabet size
struct trie {
    const char min_ch = 'a';//'A' for uppercase, '0' for digits
    struct node {
        int next[k], id, p = -1;
        char ch;
        bool leaf = 0;
        node(int a_p = -1, char a_ch = '#') : p(a_p), ch(a_ch) {
            fill(next, next + k, -1);
        }
    };
    vector<node> t;
    trie() : t(1) {}
    void add_string(const string& s, int id) {
        int c = 0;
        for (char ch : s) {
            int v = ch - min_ch;
            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 - min_ch;
            if (t[c].next[v] == -1)
                return;
            c = t[c].next[v];
        }
        t[c].leaf = 0;
    }
    int find_string(const string& s) const {
        int c = 0;
        for (char ch : s) {
            int v = ch - min_ch;
            if (t[c].next[v] == -1)
                return -1;
            c = t[c].next[v];
        }
        if (!t[c].leaf) return -1;
        return t[c].id;
    }
};
```

Listing 32: Binary Trie

```
//cat binary_trie.h | ./hash.sh
//874a75
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/set_xor_min
struct binary_trie {
    const int mx_bit = 62;
    struct node {
        long long val = -1;
        int sub_sz = 0;//number of inserted values in subtree
        int next[2] = {-1, -1};
    };
    vector<node> t;
    binary_trie() : t(1) {}
    //delta = 1 to insert val, -1 to remove val, 0 to get the # of val's in this data
        ↪ structure
    int update(long long val, int delta) {
        int c = 0;
        t[0].sub_sz += delta;
        for (int bit = mx_bit; bit >= 0; bit--) {
            bool v = (val >> bit) & 1;
            if (t[c].next[v] == -1) {
                t[c].next[v] = t.size();
                t.emplace_back();
            }
            c = t[c].next[v];
            t[c].sub_sz += delta;
        }
        t[c].val = val;
        return t[c].sub_sz;
    }
    int size() const {
        return t[0].sub_sz;
    }
    //returns x such that:
    //  x is in this data structure
    //  value of (x ^ val) is minimum
    long long min_xor(long long val) const {
        assert(size() > 0);
        int c = 0;
        for (int bit = mx_bit; bit >= 0; bit--) {
            bool v = (val >> bit) & 1;
            int ch = t[c].next[v];
            if (ch != -1 && t[ch].sub_sz > 0)
                c = ch;
            else
                c = t[c].next[!v];
        }
        return t[c].val;
    }
};
```

Listing 33: Longest Common Prefix Query

```
//cat lcp_queries.h | ./hash.sh
//a4013c
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/zalgorithm,
    ↪ https://judge.yosupo.jp/problem/enumerate_palindromes
#include "suffix_array.h"
#include "lcp.h"
#include "../range_data_structures/rmq.h"
//computes suffix array, lcp array, and then sparse table over lcp array
//O(n log n)
struct lcp_queries {
    lcp_queries(const string& s) : sa(sa_is(s, 255)), inv_sa(s.size()), lcp(LCP(s, sa)),
        ↪ st(lcp, [](int x, int y) {
        return min(x, y);
    }) {
        for (int i = 0; i < (int)s.size(); i++)
            inv_sa[sa[i]] = i;
    }
    //length of longest common prefix of suffixes s[idx1 ... n-1], s[idx2 ... n-1],
        ↪ 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 && 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);
        return st.query(idx1, idx2 - 1);
    }
    //returns true if suffix s[idx1 ... n-1] < s[idx2 ... n-1]
    //(so false if idx1 == idx2)
    bool less(int idx1, int idx2) const {
        return inv_sa[idx1] < inv_sa[idx2];
    }
    vector<int> sa, inv_sa, lcp;
    RMQ<int> st;
};
```

Listing 34: **MATH**

Listing 35: BIN EXP MOD

```
//cat exp_mod.h | ./hash.sh
//3be256
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/system_of_linear_equations,
    ↪ https://judge.yosupo.jp/problem/binomial_coefficient,
    ↪ https://judge.yosupo.jp/problem/matrix_det,
    ↪ https://judge.yosupo.jp/problem/inverse_matrix
//stress tests: tests/stress_tests/math/exp_mod.cpp
```

```
//returns (base^pw)%mod in O(log(pw)), but returns 1 for 0^0
//
//What if base doesn't fit in long long?
//Since (base^pw)%mod == ((base%mod)^pw)%mod we can calculate base under mod of `mod`
//
//What if pw doesn't fit in long long?
//case 1: mod is prime
//(base^pw)%mod == (base^(pw%(mod-1)))%mod (from Fermat's little theorem)
//so calculate pw under mod of `mod-1`
//note `mod-1` is not prime, so you need to be able to calculate `pw%(mod-1)` without
    ↪ division
//
//case 2: non-prime mod
//let t = totient(mod)
//if pw >= log2(mod) then (base^pw)%mod == (base^(t+(pw%t)))%mod (proof
    ↪ https://cp-algorithms.com/algebra/phi-function.html#generalization)
//so calculate pw under mod of `t`
//incidentally, totient(p) = p - 1 for every prime p, making this a more generalized
    ↪ version of case 1
int pow(long long base, long long pw, int mod) {
    assert(0 <= pw && 0 <= base && 1 <= mod);
    int res = 1;
    base %= mod;
    while (pw > 0) {
        if (pw & 1) res = res * base % mod;
        base = base * base % mod;
        pw >>= 1;
    }
    return res;
}
```

Listing 36: Fibonacci

```
//cat fib.h | ./hash.sh
//9ac293
#pragma once
//stress tests: tests/stress_tests/math/fib_matrix_expo.cpp
//https://codeforces.com/blog/entry/14516
//O(log(n))
unordered_map<long long, int> table;
int fib(long long n, int mod) {
    if (n < 2) return 1;
    if (table.find(n) != table.end()) return table[n];
    table[n] = (1LL * fib((n + 1) / 2, mod) * fib(n / 2, mod) + 1LL * fib((n - 1) / 2,
        ↪ mod) * fib((n - 2) / 2, mod)) % mod;
    return table[n];
}
```

Listing 37: Matrix Mult and Pow

```
//cat matrix_expo.h | ./hash.sh
//2edd34
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/matrix_product
//stress tests: tests/stress_tests/math/fib_matrix_expo.cpp
//empty matrix -> RTE
vector<vector<int>> mult(const vector<vector<int>>& a, const vector<vector<int>>& b, int
    ↪ mod) {
    assert(a[0].size() == b.size());
    int n = a.size(), m = b[0].size(), inner = b.size();
    vector<vector<int>> prod(n, vector<int>(m, 0));
    for (int i = 0; i < n; i++) {
        for (int k = 0; k < inner; k++) {
            for (int j = 0; j < m; j++)
                prod[i][j] = (prod[i][j] + 1LL * a[i][k] * b[k][j]) % mod;
        }
    }
    return prod;
}
vector<vector<int>> power(vector<vector<int>> mat/*intentional pass by value*/, long
    ↪ long pw, int mod) {
    int n = mat.size();
    vector<vector<int>> prod(n, vector<int>(n, 0));
    for (int i = 0; i < n; i++)
        prod[i][i] = 1;
    while (pw > 0) {
        if (pw % 2 == 1) prod = mult(prod, mat, mod);
        mat = mult(mat, mat, mod);
        pw /= 2;
    }
    return prod;
}
```

Listing 38: N Choose K MOD

```
//cat n_choose_k_mod.h | ./hash.sh
//1e5548
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/binomial_coefficient
//only the tests with prime mod
//for mod inverse
#include "exp_mod.h"
// usage:
//     n_choose_k nk(n, 1e9+7) to use `choose`, `inv` with inputs < n
// or:
//     n_choose_k nk(mod, mod) to use `choose_with_lucas_theorem` with arbitrarily large
    ↪ inputs
struct n_choose_k {
    n_choose_k(int n, int a_mod) : mod(a_mod), fact(n, 1), inv_fact(n, 1) {
        //this implementation doesn't work if n > mod because n! % mod = 0 when n >=
            ↪ mod. So `inv_fact` array will be all 0's
        assert(max(n, 2) <= 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
        for (int i = 2; i * i <= mod; i++) assert(mod % i);
        for (int i = 2; i < n; i++)
            fact[i] = 1LL * fact[i - 1] * i % mod;
        inv_fact.back() = pow(fact.back(), mod - 2, mod);
        for (int i = n - 2; i >= 2; i--)
            inv_fact[i] = 1LL * inv_fact[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] * inv_fact[k] % mod * inv_fact[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 prime mod (< 1e6 maybe)
    //handles n >= mod correctly
    int choose_with_lucas_theorem(long long n, long long k) const {
        if (k < 0 || k > n) return 0;
        if (k == 0 || k == n) return 1;
        return 1LL * choose_with_lucas_theorem(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] * inv_fact[n] % mod;
    }
    int mod;
    vector<int> fact, inv_fact;
};
```

Listing 39: Partitions

```
//cat partitions.h | ./hash.sh
//3356f6
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/partition_function
//https://oeis.org/A000041
//O(n sqrt n) time, but small-ish constant factor (there does exist a O(n log n)
    ↪ solution too)
vector<int> partitions(int n/*size of dp array*/, int mod) {
    vector<int> dp(n, 1);
    for (int i = 1; i < n; i++) {
        long long sum = 0;
        for (int j = 1, pent = 1, sign = 1; pent <= i; j++, pent += 3 * j - 2, sign =
            ↪ -sign) {
            if (pent + j <= i) sum += dp[i - pent - j] * sign + mod;
            sum += dp[i - pent] * sign + mod;
        }
        dp[i] = sum % mod;
    }
    return dp;
}
```

Listing 40: Derangements

```
//cat derangements.h | ./hash.sh
//c221bb
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/montmort_number_mod
//https://oeis.org/A000166
//
//for a permutation of size i:
//there are (i-1) places to move 0 to not be at index 0. Let's say we moved 0 to index j
    ↪ (j>0).
//If we move value j to index 0 (forming a cycle of length 2), then there are dp[i-2]
    ↪ derangements of the remaining i-2 elements
//else there are dp[i-1] derangements of the remaining i-1 elements (including j)
vector<int> derangements(int n/*size of dp array*/, int mod) {
    vector<int> dp(n, 0);
    dp[0] = 1;
    for (int i = 2; i < n; i++)
        dp[i] = 1LL * (i - 1) * (dp[i - 1] + dp[i - 2]) % mod;
    return dp;
}
```

Listing 41: Prime Sieve Mobius

```
//cat prime_sieve_mobius.h | ./hash.sh
//4986da
#pragma once
//stress tests: tests/stress_tests/math/prime_sieve_mobius.cpp
//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 sz = 2e6 + 10;
int mobius[sz];
void calc_mobius() {
    mobius[1] = 1;
    for (int i = 1; i < sz; i++)
        for (int j = i + i; j < sz; j += i)
            mobius[j] -= mobius[i];
}
//a_prime[val] = some random prime factor of `val`
//
//to check if `val` is prime:
//  if (a_prime[val] == val)
//
//to get all prime factors of a number `val` in O(log(val)):
//  while(val > 1) {
//      int p = a_prime[val];
//      //p is some prime factor of val
//      val /= p;
//  }
int a_prime[sz];
void calc_seive() {
    iota(a_prime, a_prime + sz, 0);
    for (int i = 2; i * i < sz; i++)
        if (a_prime[i] == i)
            for (int j = i * i; j < sz; j += i)
                a_prime[j] = i;
}
```

Listing 42: Row Reduce

```
//cat row_reduce.h | ./hash.sh
//1d7c3e
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/system_of_linear_equations,
    ↪ https://judge.yosupo.jp/problem/matrix_det,
    ↪ https://judge.yosupo.jp/problem/inverse_matrix
//for mod inverse
#include "exp_mod.h"
//First `cols` columns of mat represents a matrix to be left in reduced row echelon form
//Row operations will be performed to all later columns
//
//example usage:
//  row_reduce(mat, mat[0].size(), mod) //row reduce matrix with no extra columns
pair<int/*rank*/, int/*determinant*/> row_reduce(vector<vector<int>>& mat, int cols, int
    ↪ mod) {
    int n = mat.size(), m = mat[0].size(), rank = 0, det = 1;
    assert(cols <= m);
    for (int col = 0; col < cols && rank < n; col++) {
        //find arbitrary pivot and swap pivot to current row
        for (int i = rank; i < n; i++)
            if (mat[i][col] != 0) {
                if (rank != i) det = det == 0 ? 0 : mod - det;
                swap(mat[i], mat[rank]);
                break;
            }
        if (mat[rank][col] == 0) {
            det = 0;
            continue;
        }
        det = (1LL * det * mat[rank][col]) % mod;
        //make pivot 1 by dividing row by inverse of pivot
        int a_inv = pow(mat[rank][col], mod - 2, mod);
        for (int j = 0; j < m; j++)
            mat[rank][j] = (1LL * mat[rank][j] * a_inv) % mod;
        //zero-out all numbers above & below pivot
        for (int i = 0; i < n; i++)
            if (i != rank && mat[i][col] != 0) {
                int val = mat[i][col];
                for (int j = 0; j < m; j++) {
                    mat[i][j] -= 1LL * mat[rank][j] * val % mod;
                    if (mat[i][j] < 0) mat[i][j] += mod;
                }
            }
        rank++;
    }
    assert(rank <= min(n, cols));
    return {rank, det};
}
```

Listing 43: Solve Linear Equations MOD

```
//cat solve_linear_mod.h | ./hash.sh
//44cc6e
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/system_of_linear_equations
#include "row_reduce.h"
struct matrix_info {
    int rank, det;
    vector<int> x;
};
//Solves mat * x = b under prime mod.
//mat is a n (rows) by m (cols) matrix, b is a length n column vector, x is a length m
    ↪ vector.
//assumes n,m >= 1, else RTE
//Returns rank of mat, determinant of mat, and x (solution vector to mat * x = b).
//x is empty if no solution. If rank < m, there are multiple solutions and an arbitrary
    ↪ one is returned.
//Leaves mat in reduced row echelon form (unlike kactl) with b appended.
//O(n * m * min(n,m))
matrix_info solve_linear_mod(vector<vector<int>>& mat, const vector<int>& b, int mod) {
    assert(mat.size() == b.size());
    int n = mat.size(), m = mat[0].size();
    for (int i = 0; i < n; i++)
        mat[i].push_back(b[i]);
    auto [rank, det] = row_reduce(mat, m, mod);//row reduce not including the last column
    //check if solution exists
    for (int i = rank; i < n; i++) {
        if (mat[i].back() != 0) return {rank, det, {} }; //no solution exists
    }
    //initialize solution vector (‘x‘) from row-reduced matrix
    vector<int> x(m, 0);
    for (int i = 0, j = 0; i < rank; i++) {
        while (mat[i][j] == 0) j++; //find pivot column
        x[j] = mat[i].back();
    }
    return {rank, det, x};
}
```

Listing 44: Matrix Inverse

```
//cat matrix_inverse.h | ./hash.sh
//3056ad
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/inverse_matrix
#include "row_reduce.h"
//returns inverse of square matrix mat, empty if no inverse
vector<vector<int>> matrix_inverse(vector<vector<int>> mat/*intentional pass by value*/,
    ↪ int mod) {
    int n = mat.size();
    assert(n == (int)mat[0].size());
    //append identity matrix
    for (int i = 0; i < n; i++) {
        mat[i].resize(2 * n, 0);
        mat[i][i + n] = 1;
    }
    auto [rank, det] = row_reduce(mat, n, mod);//row reduce first n columns, leaving
        ↪ inverse in last n columns
    if (rank < n) return {}; //no inverse
    for (int i = 0; i < n; i++)
        mat[i].erase(mat[i].begin(), mat[i].begin() + n);
    return mat;
}
```

Listing 45: Euler’s Totient Phi Function

```
//cat totient.h | ./hash.sh
//36bd41
#pragma once
//stress tests: tests/stress_tests/math/totient.cpp
//Euler's totient function counts the positive integers
//up to a given integer n that are relatively prime to n.
//
//To improve, use Pollard-rho to find prime factors
int totient(int n) {
    int res = n;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            while (n % i == 0) n /= i;
            res -= res / i;
        }
    }
    if (n > 1) res -= res / n;
    return res;
}
```

Listing 46: **MAX FLOW**

Listing 47: Dinic

```
//cat dinic.h | ./hash.sh
//23e871
#pragma once
//status: not tested
struct max_flow {
    typedef long long ll;
    ll n, s, t;
    max_flow(int a_n, int a_s, int a_t) : n(a_n), s(a_s), t(a_t), d(n), ptr(n), q(n),
        ↪ g(n) {}
    void add_edge(ll a, ll b, ll cap) {
        edge_map[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 get_flow() {
        ll flow = 0;
        for (;;) {
            if (!bfs()) break;
            ptr.assign(ptr.size(), 0);
            while (ll pushed = dfs(s, inf))
                flow += pushed;
        }
        return flow;
    }
    ll get_flow_for_edge(ll a, ll b) {
        return e[edge_map[a * n + b]].flow;
    }
    const ll inf = 1e18;
    struct edge {
        ll a, b, cap, flow;
    };
    unordered_map<int, ll> edge_map;
    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 48: Hungarian

```
//cat hungarian.h | ./hash.sh
//c1ba31
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/assignment
//source: https://e-maxx.ru/algo/assignment_hungary
//
//input: cost[1...n][1...m] with 1 <= n <= m
//n workers, indexed 1, 2, ..., n
//m jobs, indexed 1, 2, ..., m
//it costs `cost[i][j]` to assign worker i to job j (1<=i<=n, 1<=j<=m)
//this returns *min* total cost to assign each worker to some distinct job
//O(n^2 * m)
//
//trick 1: set `cost[i][j]` to inf to say: "worker `i` cannot be assigned job `j`"
//trick 2: `cost[i][j]` can be negative, so to instead find max total cost over all
    ↪ matchings: set all `cost[i][j]` to `-cost[i][j]`.
//Now max total cost = - hungarian(cost).min_cost
const long long inf = 1e18;
struct match {
    long long min_cost;
    vector<int> matching;//worker `i` (1<=i<=n) is assigned to job `matching[i]`
        ↪ (1<=matching[i]<=m)
};
match hungarian(const vector<vector<long long>>& cost) {
    int n = cost.size() - 1, m = cost[0].size() - 1;
    assert(n <= m);
    vector<int> p(m + 1), way(m + 1);
    vector<long long> u(n + 1), v(m + 1);
    for (int i = 1; i <= n; i++) {
        p[0] = i;
        int j0 = 0;
        vector<long long> minv(m + 1, inf);
        vector<bool> used(m + 1, false);
        do {
            used[j0] = true;
            int i0 = p[j0], j1 = 0;
            long long delta = inf;
            for (int j = 1; j <= m; j++)
                if (!used[j]) {
                    long long 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 (int 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 {
            int j1 = way[j0];
            p[j0] = p[j1];
            j0 = j1;
        } while (j0);
    }
    vector<int> ans(n + 1);
    for (int j = 1; j <= m; j++)
        ans[p[j]] = j;
    return {-v[0], ans};
}
```

Listing 49: Min Cost Max Flow

```
//cat min_cost_max_flow.h | ./hash.sh
//805596
#pragma once
//status: not tested
const long long inf = 1e18;
struct min_cost_max_flow {
    typedef long long ll;
    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; // max amount of flow allowed
    min_cost_max_flow(int a_n, int a_s, int a_t) : n(a_n), s(a_s), t(a_t) {
        g.resize(n);
    }
    void add_edge(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> get_flow() {
        ll flow = 0, cost = 0;
        while (flow < k) {
            vector<ll> id(n, 0), d(n, inf), q(n), 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 50: **MISC**

Listing 51: DSU

```
//cat dsu.h | ./hash.sh
//9b3c97
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/unionfind,
    ↪ https://judge.yosupo.jp/problem/two_edge_connected_components
//stress tests: tests/stress_tests/misc/disjoint_set.cpp
//mnemonic: Disjoint Set Union
//NOLINTNEXTLINE(readability-identifier-naming)
struct DSU {
    int num_sets;
    vector<int> par;
    DSU(int n) : num_sets(n), par(n, -1) {}
    DSU(const DSU& rhs) : num_sets(rhs.num_sets), par(rhs.par) {}
    int find(int x) {
        return par[x] < 0 ? x : par[x] = find(par[x]);
    }
    int size_of_set(int x) {
        return -par[find(x)];
    }
    bool join(int x, int y) {
        if ((x = find(x)) == (y = find(y))) return false;
        if (par[y] < par[x]) swap(x, y);
        par[x] += par[y];
        par[y] = x;
        num_sets--;
        return true;
    }
};
```

Listing 52: PBDS

```
//cat policy_based_data_structures.h | ./hash.sh
//807de9
#pragma once
//status: not tested
//place these includes *before* the '#define int long long' else compile error
//not using <bits/extc++.h> as it compile errors on codeforces c++20 compiler
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
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<pair<long long, int>> is;
//hash table (apparently faster than unordered_map):
    ↪ https://codeforces.com/blog/entry/60737
//example initialization:
gp_hash_table<string, long long> ht;
```

Listing 53: Monotonic Stack

```
//cat monotonic_stack.h | ./hash.sh
//90f107
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/cartesian_tree
//stress tests: tests/stress_tests/misc/count_rectangles.cpp
//calculates array 'left' with:
//for every index j with left[i] < j < i: arr[j] > arr[i]
//and
//arr[left[i]] <= arr[i] if left[i] != -1
//
//trick: pass in vector<pair<T/*value*/, int/*index*/>> with arr[i].second = i (0<=i<n)
    ↪ to simulate arr[j] >= arr[i]
//
//O(n)
```

```
template<class T>
vector<int> monotonic_stack(const vector<T>& arr) {
    int n = arr.size();
    vector<int> left(n);
    for (int i = 0; i < n; i++) {
        int& j = left[i] = i - 1;
        while (j >= 0 && arr[j] > arr[i]) j = left[j];
    }
    return left;
}
```

Listing 54: Count Rectangles

```
//cat count_rectangles.h | ./hash.sh
//9873d2
#pragma once
#include "monotonic_stack.h"
//stress tests: tests/stress_tests/misc/count_rectangles.cpp
//given a 2D boolean matrix, calculate cnt[i][j]
//cnt[i][j] = the number of times an i-by-j rectangle appears in the matrix such that
    ↪ all i*j cells in the rectangle are true
//Note cnt[0][j] and cnt[i][0] will contain garbage values
//O(n*m)
vector<vector<int>> count_rectangles(const vector<vector<bool>>& grid) {
    int n = grid.size(), m = grid[0].size();
    vector<vector<int>> cnt(n + 1, vector<int>(m + 1, 0));
    vector<int> arr(m, 0);
    auto rv = [&](int j) -> int {//reverse
        return m - 1 - j;
    };
    for (int i = 0; i < n; i++) {
        vector<pair<int, int>> arr_rev(m);
        for (int j = 0; j < m; j++) {
            arr[j] = grid[i][j] * (arr[j] + 1);
            arr_rev[rv(j)] = {arr[j], j};
        }
        vector<int> left = monotonic_stack(arr);
        vector<int> right = monotonic_stack(arr_rev);
        for (int j = 0; j < m; j++) {
            int l = j - left[j] - 1, r = rv(right[rv(j)]) - j - 1;
            cnt[arr[j]][l + r + 1]++;
            cnt[arr[j]][l]--;
            cnt[arr[j]][r]--;
        }
    }
    for (int i = 1; i <= n; i++)
        for (int k = 0; k < 2; k++)
            for (int j = m; j > 1; j--)
                cnt[i][j - 1] += cnt[i][j];
    for (int j = 1; j <= m; j++)
        for (int i = n; i > 1; i--)
            cnt[i - 1][j] += cnt[i][j];
    return cnt;
}
```

Listing 55: LIS

```
//cat lis.h | ./hash.sh
//a243e1
#pragma once
//library checker tests: https://judge.yosupo.jp/problem/static_range_lis_query
//returns array of indexes representing the longest *strictly* increasing subsequence
//for non-decreasing: pass in a vector<pair<T, int>> with arr[i].second = i (0<=i<n)
//alternatively, there's this https://codeforces.com/blog/entry/13225
//mnemonic: Longest Increasing Subsequence
template<class T>
vector<int> LIS(const vector<T>& arr) { //NOLINT(readability-identifier-naming)
    if (arr.empty()) return {};
    vector<int> dp{0}/*array of indexes into `arr`*/, prev(arr.size(), -1);
    for (int i = 1; i < (int)arr.size(); i++) {
        auto it = lower_bound(dp.begin(), dp.end(), i, [&](int x, int y) -> bool {
            return arr[x] < arr[y];
        });
        if (it == dp.end()) {
            prev[i] = dp.back();
            dp.push_back(i);
        } else {
            prev[i] = it == dp.begin() ? -1 : *(it - 1);
            *it = i;
        }
        //here, dp.size() = length of LIS of prefix of arr ending at index i
    }
    vector<int> res(dp.size());
    for (int i = dp.back(), j = dp.size(); i != -1; i = prev[i])
        res[--j] = i;
    return res;
}
```

Listing 56: Safe Hash

```
//cat safe_hash.h | ./hash.sh
//e837ee
#pragma once
//status: not tested
//source: https://codeforces.com/blog/entry/62393
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);
    }
};
//usage:
unordered_map<long long, int, custom_hash> safe_map;
#include "policy_based_data_structures.h"
gp_hash_table<long long, int, custom_hash> safe_hash_table;
```