

Listings

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

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
#source: https://github.com/Errichto/youtube/blob/master/testing/s.sh
#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
//1310ef
#pragma once
//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 -> 1 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 -> 1 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, 0);
vector<int> two_edge_ccid(n), node_stack;
//bcc stuff (delete if not needed)
int num_bccs = 0;
vector<bool> is_cut(n, 0);
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] = 1;
                while (1) {
                    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] = 1;
    while (1) {
        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
//ea8ef1
#pragma once
#include "bridges_and_cuts.h"
//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>> bvt(n + cc.num_bccs);
    vector<bool> vis(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 (!vis[bccid]) {
                vis[bccid] = 1;
                bvt[v].push_back(bccid + n); //add edge between original node, and bcc
                ↪ node
                bvt[bccid + n].push_back(v);
            }
        }
        for (int bccid : bvt[v]) vis[bccid - n] = 0;
    }
    return bvt;
}
```

Listing 7: Bridge Tree

```
//cat bridge_tree.h | ./hash.sh
//85f56b
#pragma once
#include "bridges_and_cuts.h"
//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: Frequency Table of Tree Distance

```
//cat tree_freq_dist.h | ./hash.sh
//4e0d39
#pragma once
#include "../kactl/content/numerical/FastFourierTransform.h"
//returns array 'cnt_paths' where 'cnt_paths[i]' = # of paths in tree with 'i' edges
//centroid decomposition + FFT
//O(n log^2 n)
vector<long long> tree_freq_dist(const vector<vector<int>>& adj/*unrooted, connected
    ↪ tree*/) {
    int n = adj.size();
    vector<int> vis(n, 0), sizes(n);
    auto dfs_sz = [&](auto self, int node, int par) -> void {
        sizes[node] = 1;
        for (int ch : adj[node]) {
            if (ch == par || vis[ch]) continue;
            self(self, ch, node);
            sizes[node] += sizes[ch];
        }
    };
    auto find_centroid = [&](int node) -> int {
        dfs_sz(dfs_sz, node, -1);
        int size_cap = sizes[node] / 2, par = -1;
        while (1) {
            bool found = 0;
            for (int ch : adj[node]) {
                if (ch != par && !vis[ch] && sizes[ch] > size_cap) {
                    found = 1;
                    par = node;
                    node = ch;
                    break;
                }
            }
            if (!found) return node;
        }
    };
    vector<long long> cnt_paths(n, 0);
    auto dfs = [&](auto self, int node) -> void {
        node = find_centroid(node);
        vis[node] = 1;
        vector<double> total_depth(1, 1.0);
        for (int to : adj[node]) {
            if (vis[to]) continue;
            vector<double> cnt_depth(1, 0.0);
            for (queue<pair<int, int>> q({{to, node}}); !q.empty();) {
                cnt_depth.push_back(q.size());
                queue<pair<int, int>> new_q;
                while (!q.empty()) {
                    auto [curr, par] = q.front();
                    q.pop();
                    for (int ch : adj[curr]) {
```

```
        if (ch == par || vis[ch]) continue;
        new_q.emplace(ch, curr);
    }
    swap(q, new_q);
}
{
    vector<double> prod = conv(total_depth, cnt_depth);
    for (int i = 1; i < (int)prod.size(); i++) cnt_paths[i] +=
        ⇨ llround(prod[i]);
}
if (total_depth.size() < cnt_depth.size())
    ⇨ total_depth.resize(cnt_depth.size(), 0.0);
for (int i = 1; i < (int)cnt_depth.size(); i++) total_depth[i] +=
    ⇨ cnt_depth[i];
self(self, to);
}
};
dfs(dfs, 0);
return cnt_paths;
}
```

Listing 9: Dijkstra

```
//cat dijkstra.h | ./hash.sh
//56a477
#pragma once
//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: HLD

```
//cat hld.h | ./hash.sh
//499032
#pragma once
//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 {
    struct node {
        int sub_sz, par, time_in, next;
    };
    vector<node> tree;
    HLD(vector<vector<int>>& adj /*single unrooted tree*/, int root) : tree(adj.size(), {
        1, root, (int)adj.size(), root
    }) {
        dfs1(root, adj);
        int timer = 0;
        dfs2(root, adj, timer);
    }
    void dfs1(int v, vector<vector<int>>& adj) {
        auto par = find(adj[v].begin(), adj[v].end(), tree[v].par);
        if (par != adj[v].end()) adj[v].erase(par);
        for (int& to : adj[v]) {
            tree[to].par = v;
            dfs1(to, adj);
            tree[v].sub_sz += tree[to].sub_sz;
            if (tree[to].sub_sz > tree[adj[v][0]].sub_sz)
                swap(to, adj[v][0]);
        }
    }
    void dfs2(int v, const vector<vector<int>>& adj, int& timer) {
        tree[v].time_in = timer++;
        for (int to : adj[v]) {
            tree[to].next = (timer == tree[v].time_in + 1 ? tree[v].next : to);
            dfs2(to, adj, timer);
        }
    }
    // Returns inclusive-exclusive 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
    ⇨ tree[x].time_in is in any of these intervals
    vector<pair<int, int>> path(int u, int v) const {
        vector<pair<int, int>> res;
        for (; v = tree[tree[v].next].par) {
            if (tree[v].time_in < tree[u].time_in) swap(u, v);
            if (tree[tree[v].next].time_in <= tree[u].time_in) {
                res.emplace_back(tree[u].time_in, tree[v].time_in + 1);
                return res;
            }
            res.emplace_back(tree[tree[v].next].time_in, tree[v].time_in + 1);
        }
    }
    // 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 tree[x].time_in is in this interval
    pair<int, int> subtree(int i) const {
        return {tree[i].time_in, tree[i].time_in + tree[i].sub_sz};
    }
    // Returns lca of nodes u and v
    int lca(int u, int v) const {
        for (; v = tree[tree[v].next].par) {
            if (tree[v].time_in < tree[u].time_in) swap(u, v);
            if (tree[tree[v].next].time_in <= tree[u].time_in) return u;
        }
    }
};
```

Listing 11: Hopcroft Karp

```
//cat hopcroft_karp.h | ./hash.sh
//de75d7
#pragma once
//source:
    ↪ https://github.com/foreverbell/acm-icpc-cheat-sheet/blob/master/src/graph-algorithms/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  $\ddot{o}$ Knig'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'  $\Leftrightarrow$  l_to_r[node_left] is in the matching
    //
    //similarly for r_to_l with edge r_to_l[node_right]  $\Leftrightarrow$  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/edge 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 1 iff
        ↪ node_left is in the min vertex cover (same for mvc_r)
    //if mvc_l[node_left] is 0, 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  $\Leftrightarrow$  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 (1) {
        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 = 0;
        vector<bool> mvc_l(lsz, 1), mvc_r(rsz, 0);
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            mvc_l[u] = 0;
            for (int x : adj[u]) {
                mvc_r[x] = 1;
                int v = r_to_l[x];
                if (v == -1) found = 1;
                else if (level[v] == -1) {
                    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 1;
            }
        }
        level[u] = 1e9; //acts as visited array
        return 0;
    };
    for (int i = 0; i < lsz; i++)
        size_of_matching += (l_to_r[i] == -1 && dfs(dfs, i));
}
```

Listing 12: LCA

```
//cat lca.h | ./hash.sh
//22246e
#pragma once
//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 {
    struct node {
        int jmp, jmp_edges, par, depth;
        long long dist;
    };
    vector<node> tree;
    LCA(const vector<vector<pair<int, long long>>& adj, int root) : tree(adj.size(), {
        root, 1, root, 0, 0LL
    }) {
        dfs(root, adj);
    }
    void dfs(int v, const vector<vector<pair<int, long long>>& adj) {
        int jmp, jmp_edges;
        if (tree[v].depth > 0 && tree[v].jmp_edges == tree[tree[v].jmp].jmp_edges)
            jmp = tree[tree[v].jmp].jmp, jmp_edges = 2 * tree[v].jmp_edges + 1;
        else
            jmp = v, jmp_edges = 1;
        for (auto [ch, w] : adj[v]) {
            if (ch == tree[v].par) continue;
            tree[ch] = {
                jmp,
                jmp_edges,
                v,
                1 + tree[v].depth,
                w + tree[v].dist
            };
            dfs(ch, adj);
        }
    }
}
//traverse up k edges in  $O(\log(k))$ . So with k=1 this returns 'v's parent
int kth_par(int v, int k) const {
    k = min(k, tree[v].depth);
```

```
while (k > 0) {
    if (tree[v].jmp_edges <= k) {
        k -= tree[v].jmp_edges;
        v = tree[v].jmp;
    } else {
        k--;
        v = tree[v].par;
    }
}
return v;
}

int get_lca(int x, int y) const {
    if (tree[x].depth < tree[y].depth) swap(x, y);
    x = kth_par(x, tree[x].depth - tree[y].depth);
    while (x != y) {
        if (tree[x].jmp != tree[y].jmp)
            x = tree[x].jmp, y = tree[y].jmp;
        else
            x = tree[x].par, y = tree[y].par;
    }
    return x;
}

int dist_edges(int x, int y) const {
    return tree[x].depth + tree[y].depth - 2 * tree[get_lca(x, y)].depth;
}

long long dist_weight(int x, int y) const {
    return tree[x].dist + tree[y].dist - 2 * tree[get_lca(x, y)].dist;
}
};
```

Listing 13: Kth Node on Path

```
//cat kth_node_on_path.h | ./hash.sh
//7a4c3c
#pragma once
#include "lca.h"
struct kth_node_on_path {
    LCA lca;
    kth_node_on_path(const vector<vector<pair<int, long long>>>& adj, int root) :
        lca(adj, root) {}
    //consider path {u, u's par, ..., LCA(u,v), ..., v's par, v}. This returns the node
    //↪ at index k
    //assumes 0 <= k <= number of edges on path from u to v
    int query(int u, int v, int k) const {
        int lca_uv = lca.get_lca(u, v);
        int u_lca = lca.tree[u].depth - lca.tree[lca_uv].depth;
        int v_lca = lca.tree[v].depth - lca.tree[lca_uv].depth;
        assert(0 <= k && k <= u_lca + v_lca);
        return k <= u_lca ? lca.kth_par(u, k) : lca.kth_par(v, u_lca + v_lca - k);
    }
};
```

Listing 14: SCC

```
//cat scc.h | ./hash.sh
//ee9331
#pragma once
//source:
//↪ https://github.com/kth-competitive-programming/kactl/blob/main/content/graph/SCC.h
```

```
//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 (1) {
                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: Lazy Segment Tree

```
//cat seg_tree.h | ./hash.sh
//4d91a7
#pragma once
//source: https://codeforces.com/blog/entry/18051,
//↪ https://github.com/ecnerwala/cp-book/blob/master/src/seg_tree.hpp,
//↪ https://github.com/yosupo06/Algorithm/blob/master/src/datastructure/segtree.hpp
struct seg_tree {
    using dt = long long;
    using ch = long long;
    static dt combine(const dt& l, const dt& r) {
        return min(l, r);
    }
    static const dt INF = 1e18;
    struct node {
        dt val;
        ch lazy;
        int l, r; //[l, r)
    };
};
```

```
};
const int N, S/*smallest power of 2 >= N*/;
vector<node> tree;
//doesn't work with empty array
seg_tree(const vector<dt>& arr) : N(arr.size()), S(1 << __lg(2 * N - 1)), tree(2 *
    ↪ N) {
    for (int i = 0; i < N; i++)
        tree[i + N] = {arr[i], 0, i, i + 1};
    rotate(tree.rbegin(), tree.rbegin() + S - N, tree.rbegin() + N);
    for (int i = N - 1; i >= 1; i--) {
        tree[i] = {
            combine(tree[2 * i].val, tree[2 * i + 1].val),
            0,
            tree[2 * i].l,
            tree[2 * i + 1].r
        };
    }
}
void apply(int v, ch change) {
    tree[v].val += change;
    tree[v].lazy += change;
}
void push(int v) {
    if (tree[v].lazy) {
        apply(2 * v, tree[v].lazy);
        apply(2 * v + 1, tree[v].lazy);
        tree[v].lazy = 0;
    }
}
void build(int v) {
    tree[v].val = combine(tree[2 * v].val, tree[2 * v + 1].val);
}
int to_leaf(int i) const {
    i += S;
    return i < 2 * N ? i : 2 * (i - N);
}
//update range [l, r)
void update(int l, int r, ch change) {
    assert(0 <= l && l < r && r <= N);
    l = to_leaf(l), r = to_leaf(r);
    int lca_l_r = __lg((l - 1) ^ r);
    for (int lg = __lg(l); lg > __builtin_ctz(l); lg--) push(l >> lg);
    for (int lg = lca_l_r; lg > __builtin_ctz(r); lg--) push(r >> lg);
    for (int x = l, y = r; x < y; x >>= 1, y >>= 1) {
        if (x & 1) apply(x++, change);
        if (y & 1) apply(--y, change);
    }
    for (int lg = __builtin_ctz(r) + 1; lg <= lca_l_r; lg++) build(r >> lg);
    for (int lg = __builtin_ctz(l) + 1; lg <= __lg(l); lg++) build(l >> lg);
}
//query range [l, r)
dt query(int l, int r) {
    assert(0 <= l && l < r && r <= N);
    l = to_leaf(l), r = to_leaf(r);
    int lca_l_r = __lg((l - 1) ^ r);
    for (int lg = __lg(l); lg > __builtin_ctz(l); lg--) push(l >> lg);
    for (int lg = lca_l_r; lg > __builtin_ctz(r); lg--) push(r >> lg);
    dt resl = INF, resr = INF;
    for (; l < r; l >>= 1, r >>= 1) {
        if (l & 1) resl = combine(resl, tree[l++].val);
        if (r & 1) resr = combine(tree[--r].val, resr);
    }
```

```
    }
    return combine(resl, resr);
}
};
```

Listing 17: BIT

```
//cat bit.h | ./hash.sh
//83059d
#pragma once
//mnemonic: Binary Indexed Tree
//NOLINTNEXTLINE(readability-identifier-naming)
template<class T> struct BIT {
    const int N;
    vector<T> bit;
    BIT(int a_n) : N(a_n), bit(N, 0) {}
    BIT(const vector<T>& a) : BIT(a.size()) {
        for (int i = 0; i < N; i++) {
            bit[i] += a[i];
            int j = i | (i + 1);
            if (j < N) bit[j] += bit[i];
        }
    }
    void update(int i, const T& d) {
        assert(0 <= i && i < N);
        for (; i < N; i |= i + 1) bit[i] += d;
    }
    T sum(int r) const { //sum of range [0, r)
        assert(0 <= r && r <= N);
        T ret = 0;
        for (; r > 0; r &= r - 1) ret += bit[r - 1];
        return ret;
    }
    T sum(int l, int r) const { //sum of range [l, r)
        assert(0 <= l && l <= r && r <= N);
        return sum(r) - sum(l);
    }
    //Returns min pos (0<=pos<=N+1) such that sum of [0, pos) >= sum
    //Returns N + 1 if no sum is >= sum, or 0 if empty sum is.
    //Doesn't work with negatives
    int lower_bound(T sum) const {
        if (sum <= 0) return 0;
        int pos = 0;
        for (int pw = 1 << __lg(N | 1); pw; pw >>= 1)
            if (pos + pw <= N && bit[pos + pw - 1] < sum)
                pos += pw, sum -= bit[pos - 1];
        return pos + 1;
    }
};
```

Listing 18: RMQ

```
//cat rmq.h | ./hash.sh
//082180
#pragma once
//source:
    ↪ https://github.com/kth-competitive-programming/kactl/blob/main/content/data-structure
//usage:
// vector<long long> arr;
```



```
// ...
// RMQ<long long> rmq(arr, [0](auto x, auto y) { return min(x,y); });
//
//to also get index of min element, do:
// RMQ<pair<T, int>> rmq(arr, [0](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
//NOLINTNEXTLINE(readability-identifier-naming)
template <class T> struct RMQ {
    vector<vector<T>> dp;
    function<T(const T&, const T&> op;
    RMQ(const vector<T>& arr, const function<T(const T&, const T&>& a_op) : dp(1, arr),
        op(a_op) {
        for (int pw = 1, k = 1, n = arr.size(); 2 * pw <= n; pw *= 2, k++) {
            dp.emplace_back(n - 2 * pw + 1);
            for (int j = 0; j < n - 2 * pw + 1; j++)
                dp[k][j] = op(dp[k - 1][j], dp[k - 1][j + pw]);
        }
        //inclusive-exclusive range [l, r)
        T query(int l, int r) const {
            assert(0 <= l && l < r && r <= (int)dp[0].size());
            int lg = __lg(r - l);
            return op(dp[lg][l], dp[lg][r - (1 << lg)]);
        }
    }
};
```

Listing 19: Implicit Lazy Segment Tree

```
//cat implicit_seg_tree.h | ./hash.sh
//2edb5e
#pragma once
//example initialization:
// implicit_seg_tree<10'000'000> ist(l, r);
template <int N> struct implicit_seg_tree {
    using dt = array<long long, 2>; //min, number of mins
    using ch = long long;
    static dt combine(const dt& l, const dt& r) {
        if (l[0] == r[0]) return {l[0], l[1] + r[1]};
        return min(l, r);
    }
    static constexpr dt UNIT{(long long)1e18, 0LL};
    struct node {
        dt val;
        ch lazy;
        int lch, rch; // children, indexes into 'tree', -1 for null
        node() {}
        node(const dt& a_val) : val(a_val), lazy(0), lch(-1), rch(-1) {}
    };
    array<node, N> tree;
    int ptr, root_l, root_r; // [root_l, root_r) defines range of root node; handles
        op negatives
    implicit_seg_tree(int l, int r) : ptr(0), root_l(l), root_r(r) {
        tree[ptr++] = node(dt{0, r - l});
    }
    void apply(int v, ch add) {
        tree[v].val[0] += add;
        tree[v].lazy += add;
    }
};
```

```
}
void push(int v, int tl, int tr) {
    if (tr - tl > 1 && tree[v].lch == -1) {
        int tm = tl + (tr - tl) / 2;
        assert(ptr + 1 < N);
        tree[v].lch = ptr;
        tree[ptr++] = node(dt{0, tm - tl});
        tree[v].rch = ptr;
        tree[ptr++] = node(dt{0, tr - tm});
    }
    if (tree[v].lazy) {
        apply(tree[v].lch, tree[v].lazy);
        apply(tree[v].rch, tree[v].lazy);
        tree[v].lazy = 0;
    }
}
//update range [l,r)
void update(int l, int r, ch add) {
    update(0, root_l, root_r, l, r, add);
}
void update(int v, int tl, int tr, int l, int r, ch add) {
    if (r <= tl || tr <= l)
        return;
    if (l <= tl && tr <= r)
        return apply(v, add);
    push(v, tl, tr);
    int tm = tl + (tr - tl) / 2;
    update(tree[v].lch, tl, tm, l, r, add);
    update(tree[v].rch, tm, tr, l, r, add);
    tree[v].val = combine(tree[tree[v].lch].val,
        tree[tree[v].rch].val);
}
//query range [l,r)
dt query(int l, int r) {
    return query(0, root_l, root_r, l, r);
}
dt query(int v, int tl, int tr, int l, int r) {
    if (r <= tl || tr <= l)
        return UNIT;
    if (l <= tl && tr <= r)
        return tree[v].val;
    push(v, tl, tr);
    int tm = tl + (tr - tl) / 2;
    return combine(query(tree[v].lch, tl, tm, l, r),
        query(tree[v].rch, tm, tr, l, r));
}
};
```

Listing 20: Kth Smallest

```
//cat kth_smallest.h | ./hash.sh
//f8ce8b
#include <bits/stdc++.h>
using namespace std;
#pragma once
//source:
    op https://cp-algorithms.com/data_structures/segment_tree.html#preserving-the-history-o
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 + 1);
    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 (tr - tl == 1) {
        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, tr, idx);
    tree.push_back({tree[lch].sum + tree[rch].sum, lch, rch});
    return tree.size() - 1;
}
/* find (k+1)th smallest number in range [l, 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); //note this condition implies l < r
    assert(0 <= l && r < (int)roots.size());
    return query(roots[l], roots[r], mn, mx, k);
}
int query(int vl, int vr, int tl, int tr, int k) const {
    assert(tree[vr].sum > tree[vl].sum);
    if (tr - tl == 1)
        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, tr, k - left_count);
}
};
```

Listing 21: Number Distinct Elements

```
//cat distinct_query.h | ./hash.sh
//6dfaad
#pragma once
//source:
⇨ https://cp-algorithms.com/data_structures/segment_tree.html#preserving-the-history-of-it
//works with negatives
//O(n log n) time and space
struct distinct_query {
    struct node {
        int sum;
        int lch, rch; //children, indexes into 'tree'
    };
};
```

```
const int N;
vector<int> roots;
deque<node> tree;
distinct_query(const vector<int>& arr) : N(arr.size()), roots(N + 1, 0) {
    tree.push_back({0, 0, 0}); //acts as null
    map<int, int> last_idx;
    for (int i = 0; i < N; i++) {
        roots[i + 1] = update(roots[i], 0, N, last_idx[arr[i]]);
        last_idx[arr[i]] = i + 1;
    }
}
int update(int v, int tl, int tr, int idx) {
    if (tr - tl == 1) {
        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, 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 {
    assert(0 <= l && l <= r && r <= N);
    return query(roots[l], roots[r], 0, N, 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 - tl) / 2;
    return query(tree[vl].lch, tree[vr].lch, tl, tm, idx) +
        query(tree[vl].rch, tree[vr].rch, tm, tr, idx);
}
};
```

Listing 22: Merge Sort Tree

```
//cat merge_sort_tree.h | ./hash.sh
//a84032
#pragma once
//For point updates: either switch to policy based BST, or use sqrt decomposition
struct merge_sort_tree {
    const int N, S; //smallest power of 2 >= N*/;
    vector<vector<int>> tree;
    //doesn't work with empty array
    merge_sort_tree(const vector<int>& arr) : N(arr.size()), S(1 << __lg(2 * N - 1)),
        ⇨ tree(2 * N) {
        for (int i = 0; i < N; i++)
            tree[i + N] = {arr[i]};
        rotate(tree.rbegin(), tree.rbegin() + S - N, tree.rbegin() + N);
        for (int i = N - 1; i >= 1; i--) {
            const auto& l = tree[2 * i];
            const auto& r = tree[2 * i + 1];
```

```
        tree[l].reserve(l.size() + r.size());
        merge(l.begin(), l.end(), r.begin(), r.end(), back_inserter(tree[l]));
    }
}

int value(int v, int x) const {
    return lower_bound(tree[v].begin(), tree[v].end(), x) - tree[v].begin();
}

int to_leaf(int i) const {
    i += S;
    return i < 2 * N ? i : 2 * (i - N);
}

//How many values in range [l, r) are < x?
//O(log^2(n))
int query(int l, int r, int x) const {
    int res = 0;
    for (l = to_leaf(l), r = to_leaf(r); l < r; l >= 1, r >= 1) {
        if (l & 1) res += value(l++, x);
        if (r & 1) res += value(--r, x);
    }
    return res;
}

};
```

Listing 23: STRINGS

Listing 24: Suffix Array and LCP Array

```
//cat string.hpp | ./hash.sh
//67378f
#ifndef ATCODER_STRING_HPP
#define ATCODER_STRING_HPP 1

#include <algorithm>
#include <cassert>
#include <numeric>
#include <string>
#include <vector>

namespace atcoder {

namespace internal {

std::vector<int> sa_naive(const std::vector<int>& s) {
    int n = int(s.size());
    std::vector<int> sa(n);
    std::iota(sa.begin(), sa.end(), 0);
    std::sort(sa.begin(), sa.end(), [&](int l, int r) {
        if (l == r) return false;
        while (l < n && r < n) {
            if (s[l] != s[r]) return s[l] < s[r];
            l++;
            r++;
        }
        return l == n;
    });
    return sa;
}

};
```

```
std::vector<int> sa_doubling(const std::vector<int>& s) {
    int n = int(s.size());
    std::vector<int> sa(n), rnk = s, tmp(n);
    std::iota(sa.begin(), sa.end(), 0);
    for (int k = 1; k < n; k *= 2) {
        auto cmp = [&](int x, int y) {
            if (rnk[x] != rnk[y]) return rnk[x] < rnk[y];
            int rx = x + k < n ? rnk[x + k] : -1;
            int ry = y + k < n ? rnk[y + k] : -1;
            return rx < ry;
        };
        std::sort(sa.begin(), sa.end(), cmp);
        tmp[sa[0]] = 0;
        for (int i = 1; i < n; i++) {
            tmp[sa[i]] = tmp[sa[i - 1]] + (cmp(sa[i - 1], sa[i]) ? 1 : 0);
        }
        std::swap(tmp, rnk);
    }
    return sa;
}

// 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
template <int THRESHOLD_NAIVE = 10, int THRESHOLD_DOUBLING = 40>
std::vector<int> sa_is(const std::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};
        }
    }
    if (n < THRESHOLD_NAIVE) {
        return sa_naive(s);
    }
    if (n < THRESHOLD_DOUBLING) {
        return sa_doubling(s);
    }

    std::vector<int> sa(n);
    std::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]);
    }
    std::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];
    }
}
```

```

auto induce = [&](const std::vector<int>& lms) {
    std::fill(sa.begin(), sa.end(), -1);
    std::vector<int> buf(upper + 1);
    std::copy(sum_s.begin(), sum_s.end(), buf.begin());
    for (auto d : lms) {
        if (d == n) continue;
        sa[buf[s[d]]++] = d;
    }
    std::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;
        }
    }
    std::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;
        }
    }
};

std::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++;
    }
}

std::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) {
    std::vector<int> sorted_lms;
    sorted_lms.reserve(m);
    for (int v : sa) {
        if (lms_map[v] != -1) sorted_lms.push_back(v);
    }
    std::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 - 1 != 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<THRESHOLD_NAIVE, THRESHOLD_DOUBLING>(rec_s, rec_upper);

    for (int i = 0; i < m; i++) {
        sorted_lms[i] = lms[rec_sa[i]];
    }
    induce(sorted_lms);
}
return sa;
}

} // namespace internal

std::vector<int> suffix_array(const std::vector<int>& s, int upper) {
    assert(0 <= upper);
    for (int d : s) {
        assert(0 <= d && d <= upper);
    }
    auto sa = internal::sa_is(s, upper);
    return sa;
}

template <class T> std::vector<int> suffix_array(const std::vector<T>& s) {
    int n = int(s.size());
    std::vector<int> idx(n);
    iota(idx.begin(), idx.end(), 0);
    sort(idx.begin(), idx.end(), [&](int l, int r) { return s[l] < s[r]; });
    std::vector<int> s2(n);
    int now = 0;
    for (int i = 0; i < n; i++) {
        if (i && s[idx[i - 1]] != s[idx[i]]) now++;
        s2[idx[i]] = now;
    }
    return internal::sa_is(s2, now);
}

std::vector<int> suffix_array(const std::string& s) {
    int n = int(s.size());
    std::vector<int> s2(n);
    for (int i = 0; i < n; i++) {
        s2[i] = s[i];
    }
    return internal::sa_is(s2, 255);
}

// Reference:
// T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park,
// Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its
// Applications

```

```
template <class T>
std::vector<int> lcp_array(const std::vector<T>& s,
                        const std::vector<int>& sa) {
    int n = int(s.size());
    assert(n >= 1);
    std::vector<int> rnk(n);
    for (int i = 0; i < n; i++) {
        rnk[sa[i]] = i;
    }
    std::vector<int> lcp(n - 1);
    int h = 0;
    for (int i = 0; i < n; i++) {
        if (h > 0) h--;
        if (rnk[i] == 0) continue;
        int j = sa[rnk[i] - 1];
        for (; j + h < n && i + h < n; h++) {
            if (s[j + h] != s[i + h]) break;
        }
        lcp[rnk[i] - 1] = h;
    }
    return lcp;
}

std::vector<int> lcp_array(const std::string& s, const std::vector<int>& sa) {
    int n = int(s.size());
    std::vector<int> s2(n);
    for (int i = 0; i < n; i++) {
        s2[i] = s[i];
    }
    return lcp_array(s2, sa);
}

// Reference:
// D. Gusfield,
// Algorithms on Strings, Trees, and Sequences: Computer Science and
// Computational Biology
template <class T> std::vector<int> z_algorithm(const std::vector<T>& s) {
    int n = int(s.size());
    if (n == 0) return {};
    std::vector<int> z(n);
    z[0] = 0;
    for (int i = 1, j = 0; i < n; i++) {
        int& k = z[i];
        k = (j + z[j] <= i) ? 0 : std::min(j + z[j] - i, z[i - j]);
        while (i + k < n && s[k] == s[i + k]) k++;
        if (j + z[j] < i + z[i]) j = i;
    }
    z[0] = n;
    return z;
}

std::vector<int> z_algorithm(const std::string& s) {
    int n = int(s.size());
    std::vector<int> s2(n);
    for (int i = 0; i < n; i++) {
        s2[i] = s[i];
    }
    return z_algorithm(s2);
}

} // namespace atcoder
```

```
#endif // ATCODER_STRING_HPP
```

Listing 25: KMP

```
//cat kmp.h | ./hash.sh
//73f1be
#pragma once
//mnemonic: Knuth Morris Pratt
#include "../kactl/content/strings/KMP.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
//8 last characters) as the haystack or just use kactl's min rotation code
//NOLINTNEXTLINE(readability-identifier-naming)
template <class T> struct KMP {
    KMP(const T& a_needle) : needle(a_needle), pf(pi(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 0 for "all" and KMP::find will only
    // return the first match: {1}. Useful for checking if there exists
    // some match:
    //
    // KMP::find(<haystack>,0).size() > 0
    vector<int> find(const T& haystack, bool all = 1) const {
        vector<int> matches;
        for (int i = 0, j = 0; i < (int)haystack.size(); i++) {
            while (j > 0 && needle[j] != haystack[i]) j = pf[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 = pf[j - 1];
            }
        }
        return matches;
    }
    T needle;
    vector<int> pf; //prefix function
};
```

Listing 26: Trie

```
//cat trie.h | ./hash.sh
//65b7dd
#pragma once
//source: https://cp-algorithms.com/string/aho_corasick.html#construction-of-the-trie
const int K = 26; //alphabet size
struct trie {
    const char MIN_CH = 'A'; // 'a' for lowercase, '0' for digits
    struct node {
        array<int, K> next;
        int cnt_words = 0, par = -1;
        char ch;
        node(int a_par = -1, char a_ch = '#') : par(a_par), ch(a_ch) {
            fill(next.begin(), next.end(), -1);
        }
    };
    vector<node> t;
    trie() : t(1) {}
    void add_string(const string& s) {
        int v = 0;
        for (char ch : s) {
            int let = ch - MIN_CH;
            if (t[v].next[let] == -1) {
                t[v].next[let] = t.size();
                t.emplace_back(v, ch);
            }
            v = t[v].next[let];
        }
        t[v].cnt_words++;
    }
    bool find_string(const string& s) const {
        int v = 0;
        for (char ch : s) {
            int let = ch - MIN_CH;
            if (t[v].next[let] == -1) return 0;
            v = t[v].next[let];
        }
        return t[v].cnt_words;
    }
};
```

Listing 27: Binary Trie

```
//cat binary_trie.h | ./hash.sh
//33aa3a
#pragma once
struct binary_trie {
    const int MX_BIT = 62;
    struct node {
        long long val = -1;
        int sub_sz = 0; //number of inserted values in subtree
        array<int, 2> next = {-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 28: Longest Common Prefix Query

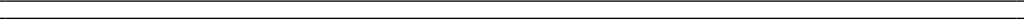
```
//cat lcp_query.h | ./hash.sh
//951676
#pragma once
#include "../ac-library/atcoder/string.hpp"
#include "../range_data_structures/rmq.h"
//computes suffix array, lcp array, and then sparse table over lcp array
//O(n log n)
struct lcp_query {
    const int N;
    vector<int> sa, lcp, inv_sa;
    RMQ<int> st;
    lcp_query(const string& s) : N(s.size()), sa(atcoder::suffix_array(s)),
        lcp(atcoder::lcp_array(s, sa)), inv_sa(N), st(lcp, [](int x, int y) {
            return min(x, y);
        }) {}
    for (int i = 0; i < N; i++) inv_sa[sa[i]] = i;
}
//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:
//r1-l1 == r2-l2 && longest_common_prefix(l1, l2) >= r1-l1
int longest_common_prefix(int idx1, int idx2) const {
    if (idx1 == idx2) return N - idx1;
    idx1 = inv_sa[idx1];
    idx2 = inv_sa[idx2];
```

```
        if (idx1 > idx2) swap(idx1, idx2);
        return st.query(idx1, idx2);
    }
    //returns 1 if suffix s[idx1 ... n] < s[idx2 ... n]
    //(so 0 if idx1 == idx2)
    bool less(int idx1, int idx2) const {
        return inv_sa[idx1] < inv_sa[idx2];
    }
};
```

Listing 29: Palindrome Query

```
//cat palindrome_query.h | ./hash.sh
//7326d0
#pragma once
#include "../kactl/content/strings/Manacher.h"
struct pal_query {
    const int N;
    array<vi, 2> pal_len;
    pal_query(const string& s) : N(s.size()), pal_len(manacher(s)) {}
    //returns 1 if substring s[l...r] is a palindrome
    bool is_pal(int l, int r) const {
        assert(0 <= l && l <= r && r <= N);
        int len = r - l;
        return pal_len[len & 1][l + len / 2] >= len / 2;
    }
};
```

Listing 30: MATH



Listing 31: BIN EXP MOD

```
//cat exp_mod.h | ./hash.sh
//3be256
#pragma once
//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 32: Fibonacci

```
//cat fib.h | ./hash.sh
//9ac293
#pragma once
//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 33: Matrix Mult and Pow

```
//cat matrix_expo.h | ./hash.sh
//2edd34
#pragma once
//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 34: N Choose K MOD

```
//cat n_choose_k_mod.h | ./hash.sh
//f3a1a9
#pragma once
//for mod inverse
#include "exp_mod.h"
// usage:
//      n_choose_k nk(n, 1e9+7) to use 'choose', 'inv' with inputs strictly < 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] = inv_fact[i + 1] * (i + 1LL) % 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 x such that x * n % mod == 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 35: Partitions

```
//cat partitions.h | ./hash.sh
//3356f6
#pragma once
//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, 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 36: Derangements

```
//cat derangements.h | ./hash.sh
//c221bb
#pragma once
//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, 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 37: Prime Sieve

```
//cat prime_sieve.h | ./hash.sh
//f17268
#pragma once
//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;
//  }
const int N = 1e6 + 10;
array<int, N> a_prime;
void calc_seive() {
    iota(a_prime.begin(), a_prime.end(), 0);
    for (int i = 2; i * i < N; i++)
        if (a_prime[i] == i)
            for (int j = i * i; j < N; j += i)
                a_prime[j] = i;
```



```
}

```

Listing 38: Mobius Inversion

```
//cat mobius_inversion.h | ./hash.sh
//55b832
#pragma once
//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 = 1e6 + 10;
array<int, N> mobius;
void calc_mobius() {
    mobius[1] = 1;
    for (int i = 1; i < N; i++)
        for (int j = i + i; j < N; j += i)
            mobius[j] -= mobius[i];
}
```

Listing 39: Row Reduce

```
//cat row_reduce.h | ./hash.sh
//1d7c3e
#pragma once
//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 40: Solve Linear Equations MOD

```
//cat solve_linear_mod.h | ./hash.sh
//44cc6e
#pragma once
#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
    vector 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 41: Matrix Inverse

```
//cat matrix_inverse.h | ./hash.sh
//3056ad
#pragma once
#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 42: Euler’s Totient Phi Function

```
//cat totient.h | ./hash.sh
//36bd41
#pragma once
//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 43: MAX FLOW

Listing 44: Dinic

```
//cat dinic.h | ./hash.sh
//e6a3d5
#pragma once
struct max_flow {
    using ll = long long;
    const ll INF = 1e18;
    struct edge {
        int a, b;
        ll cap, flow;
    };
    vector<edge> e;
    vector<vector<int>> g;
    vector<int> q, d, ptr;
    max_flow(int n) : g(n), q(n), d(n), ptr(n) {}
    void add_edge(int a, int b, ll cap) {
        edge e1 = { a, b, cap, 0 };
        edge e2 = { b, a, 0, 0 };
        g[a].push_back(e.size());
        e.push_back(e1);
        g[b].push_back(e.size());
        e.push_back(e2);
    }
};
```

```
    }
    ll get_flow(int s, int t) {
        ll flow = 0;
        for (;;) {
            if (!bfs(s, t)) break;
            ptr.assign(ptr.size(), 0);
            while (ll pushed = dfs(s, INF, t))
                flow += pushed;
        }
        return flow;
    }
    bool bfs(int s, int t) {
        int qh = 0, qt = 0;
        q[qt++] = s;
        d.assign(d.size(), -1);
        d[s] = 0;
        while (qh < qt && d[t] == -1) {
            int v = q[qh++];
            for (int i = 0; i < (int)g[v].size(); i++) {
                int 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(int v, ll flow, int t) {
        if (!flow) return 0;
        if (v == t) return flow;
        for (; ptr[v] < (int)g[v].size(); ptr[v]++) {
            int id = g[v][ptr[v]], to = e[id].b;
            if (d[to] != d[v] + 1) continue;
            ll pushed = dfs(to, min(flow, e[id].cap - e[id].flow), t);
            if (pushed) {
                e[id].flow += pushed;
                e[id ^ 1].flow -= pushed;
                return pushed;
            }
        }
        return 0;
    }
};
```

Listing 45: Hungarian

```
//cat hungarian.h | ./hash.sh
//625431
#pragma once
//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
//0(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, 0);
        do {
            used[j0] = 1;
            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 46: Min Cost Max Flow

```
//cat min_cost_max_flow.h | ./hash.sh
//6d926c
#pragma once
const long long INF = 1e18;
struct min_cost_max_flow {
    using ll = long long;
    struct edge {
        int a, b;
```

```
ll cap, cost, flow;
int back;
};
const int N;
vector<edge> e;
vector<vector<int>> g;
min_cost_max_flow(int a_n) : N(a_n), g(N) {}
void add_edge(int a, int b, ll cap, ll cost) {
    edge e1 = {a, b, cap, cost, 0, (int)g[b].size() };
    edge e2 = {b, a, 0, -cost, 0, (int)g[a].size() };
    g[a].push_back(e1);
    e.push_back(e1);
    g[b].push_back(e2);
    e.push_back(e2);
}
//returns minimum cost to send 'total_flow' flow through the graph, or -1 if
↳ impossible
ll get_flow(int s, int t, ll total_flow) {
    ll flow = 0, cost = 0;
    while (flow < total_flow) {
        vector<ll> d(N, INF);
        vector<int> p_edge(N), id(N, 0), q(N), p(N);
        int qh = 0, qt = 0;
        q[qt++] = s;
        d[s] = 0;
        while (qh != qt) {
            int v = q[qh++];
            id[v] = 2;
            if (qh == N) qh = 0;
            for (int i = 0; i < (int)g[v].size(); i++) {
                const 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 = total_flow - flow;
        for (int v = t; v != s; v = p[v]) {
            int pv = p[v], pr = p_edge[v];
            addflow = min(addflow, e[g[pv][pr]].cap - e[g[pv][pr]].flow);
        }
        for (int v = t; v != s; v = p[v]) {
            int pv = p[v], 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 < total_flow ? -1 : cost;
}
```

```
    }
};
```

```
    return res;
}
```

Listing 47: MISC

Listing 48: PBDS

```
//cat policy_based_data_structures.h | ./hash.sh
//807de9
#pragma once
//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 49: LIS

```
//cat lis.h | ./hash.sh
//a243e1
#pragma once
//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
//NOLINTNEXTLINE(readability-identifier-naming)
template<class T> vector<int> LIS(const vector<T>& arr) {
    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;
```

Listing 50: Number of Distinct Subsequences DP

```
//cat num_distinct_subsequences.h | ./hash.sh
//9542f5
#pragma once
//returns number of distinct subsequences
//the empty subsequence is counted
int num_subsequences(const vector<int>& arr, int mod) {
    int n = arr.size();
    vector<int> dp(n + 1, 1);
    map<int, int> last;
    for (int i = 0; i < n; i++) {
        int& curr = dp[i + 1] = 2 * dp[i];
        if (curr >= mod) curr -= mod;
        auto it = last.find(arr[i]);
        if (it != last.end()) {
            curr -= dp[it->second];
            if (curr < 0) curr += mod;
            it->second = i;
        } else last[arr[i]] = i;
    }
    return dp[n];
}
```

Listing 51: Safe Hash

```
//cat safe_hash.h | ./hash.sh
//d9ea53
#pragma once
//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;
```

Listing 52: Monotonic Stack

```
//cat monotonic_stack.h | ./hash.sh
//5d2e78
#pragma once
//usages:
```

```
// vector<int> left = monotonic_stack<int>(arr, less()); //(or replace 'less' with:
//    ↪ less_equal, greater, greater_equal
// vector<int> left = monotonic_stack<int>(arr, [0](int x, int y) {return x < y;});
//
//returns array 'left' where 'left[i]' = max index such that:
// 'left[i]' < i && !op(arr[left[i]], arr[i])
//or -1 if no index exists
//O(n)
template<class T> vector<int> monotonic_stack(const vector<T>& arr, const
    ↪ function<T(const T&, const T&)>& op) {
    int n = arr.size();
    vector<int> left(n);
    for (int i = 0; i < n; i++) {
        int& j = left[i] = i - 1;
        while (j >= 0 && op(arr[j], arr[i])) j = left[j];
    }
    return left;
}
```

Listing 53: Count Rectangles

```
//cat count_rectangles.h | ./hash.sh
//b2cccd
#pragma once
#include "monotonic_stack.h"
//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 1
//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 /**reverse*/ = [&](int j) -> int {
        return m - 1 - j;
    };
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++)
            arr[j] = grid[i][j] * (arr[j] + 1);
        vector<int> left = monotonic_stack<int>(arr, greater());
        vector<int> right = monotonic_stack<int>(vector<int>(arr.rbegin(), arr.rend()),
            ↪ greater_equal());
        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 54: Cartesian Tree

```
//cat cartesian_tree.h | ./hash.sh
//0b95bc
#pragma once
#include "monotonic_stack.h"
//min cartesian tree
vector<int> cartesian_tree(const vector<int>& arr) {
    int n = arr.size();
    auto rv /**reverse*/ = [&](int i) -> int {
        return n - 1 - i;
    };
    vector<int> left = monotonic_stack<int>(arr, greater());
    vector<int> right = monotonic_stack<int>(vector<int>(arr.rbegin(), arr.rend()),
        ↪ greater());
    vector<int> par(n);
    for (int i = 0; i < n; i++) {
        int l = left[i], r = rv(right[rv(i)]);
        if (l >= 0 && r < n) par[i] = arr[l] > arr[r] ? l : r;
        else if (l >= 0) par[i] = l;
        else if (r < n) par[i] = r;
        else par[i] = i;
    }
    return par;
}
```

Listing 55: Max Rectangle in Histogram

```
//cat max_rect_histogram.h | ./hash.sh
//4e6291
#pragma once
#include "monotonic_stack.h"
long long max_rect_histogram(const vector<int>& arr) {
    int n = arr.size();
    auto rv /**reverse*/ = [&](int i) -> int {
        return n - 1 - i;
    };
    vector<int> left = monotonic_stack<int>(arr, greater_equal());
    vector<int> right = monotonic_stack<int>(vector<int>(arr.rbegin(), arr.rend()),
        ↪ greater_equal());
    long long max_area = 0;
    for (int i = 0; i < n; i++) {
        int l = left[i], r = rv(right[rv(i)]); //arr[i] is the max of range (l, r)
        max_area = max(max_area, 1LL * arr[i] * (r - l - 1));
    }
    return max_area;
}
```