

Listings

1	<b>CONTEST</b>	2	38	Max Rectangle in Histogram	13
2	Hash codes	2	39	Monotonic Stack	13
3	Template	2	40	Iterate Chooses	13
4	Test on random inputs	2	41	Iterate Submasks	14
5	<b>MAX FLOW</b>	2	42	Iterate Supermasks	14
6	Dinic	2	43	LIS	14
7	Hungarian	3	44	Number of Distinct Subsequences DP	14
8	Min Cost Max Flow	3	45	PBDS	14
9	<b>GRAPHS</b>	4	46	Random	14
10	Block Vertex Tree	4	47	Safe Hash	15
11	Bridge Tree	4	48	<b>RANGE DATA STRUCTURES</b>	15
12	Bridges and Cuts	4	49	Number Distinct Elements	15
13	Centroid Decomp	5	50	Implicit Lazy Segment Tree	15
14	Frequency Table of Tree Distance	6	51	Kth Smallest	16
15	Count Paths Per Node	6	52	Merge Sort Tree	17
16	Dijkstra	6	53	BIT	17
17	HLD	7	54	RMQ	17
18	Hopcroft Karp	7	55	Lazy Segment Tree	18
19	Kth Node on Path	8	56	<b>STRINGS</b>	18
20	LCA	8	57	Binary Trie	18
21	SCC	9	58	KMP	19
22	TREE ISOMORPHISM	9	59	Longest Common Prefix Query	19
23	<b>MATH</b>	9	60	Palindrome Query	20
24	Derangements	10	61	Trie	20
25	BIN EXP MOD	10	62	Suffix Array and LCP Array	20
26	Fibonacci	10			
27	Matrix Mult	10			
28	Mobius Inversion	10			
29	N Choose K MOD	11			
30	Partitions	11			
31	Prime Sieve	11			
32	Row Reduce	11			
33	Solve Linear Equations MOD	12			
34	Euler's Totient Phi Function	12			
35	<b>MISC</b>	12			
36	Cartesian Tree	12			
37	Count Rectangles	13			

CONTEST

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 a.cpp | ./hash.sh
#or just copy this command:
#  cat a.cpp | 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
```

Template

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

int main() {
    cin.tie(0)->sync_with_stdio(0);
    return 0;
}
```

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
```

MAX FLOW

Dinic

```
//cat dinic.hpp | ./hash.sh
//2189f1
#pragma once
//source: https://e-maxx.ru/algo/dinic
```

```
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> d, ptr;
    max_flow(int n) : g(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(e1.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;
        while (bfs(s, t)) {
            ptr.assign(ptr.size(), 0);
            while (ll pushed = dfs(s, t, INF))
                flow += pushed;
        }
        return flow;
    }
    bool bfs(int s, int t) {
        queue<int> q({s});
        d.assign(d.size(), -1);
        d[s] = 0;
        while (!q.empty() && d[t] == -1) {
            int v = q.front();
            q.pop();
            for (int id : g[v]) {
                int to = e[id].b;
                if (d[to] == -1 && e[id].flow < e[id].cap) {
                    q.push(to);
                    d[to] = d[v] + 1;
                }
            }
        }
        return d[t] != -1;
    }
    ll dfs(int v, int t, ll flow) {
        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;
            if (ll pushed = dfs(to, t, min(flow, e[id].cap - e[id].flow))) {
                e[id].flow += pushed;
                e[id ^ 1].flow -= pushed;
                return pushed;
            }
        }
        return 0;
    }
}
```

```

        j0 = j1;
    } while (j0);
}
vector<int> ans(n + 1);
for (int j = 1; j <= m; j++)
    ans[p[j]] = j;
return {-v[0], ans};
}

```

## Min Cost Max Flow

```

//cat min_cost_max_flow.hpp | ./hash.sh
//6d926c
#pragma once
//source: https://e-maxx.ru/algo/min_cost_flow
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;
                        }
                    }
                }
            }
            flow += d[t];
            cost += d[t] * total_flow;
        }
    }
};

```

```
        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;
}
};
```

GRAPHS

Block Vertex Tree

```
//cat block_vertex_tree.hpp | ./hash.sh
//ea8ef1
#pragma once
#include "bridges_and_cuts.hpp"
//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;
            }
        }
    }
    return bvt;
}
```

```
        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;
}
```

Bridge Tree

```
//cat bridge_tree.hpp | ./hash.sh
//85f56b
#pragma once
#include "bridges_and_cuts.hpp"
//never adds multiple edges as bridges_and_cuts.hpp 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;
}
```

Bridges and Cuts

```
//cat bridges_and_cuts.hpp | ./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};
}

```

## Centroid Decomp

```

//cat centroid_decomp.hpp | ./hash.sh
//429129
#pragma once

// Time and Space complexity are given in terms of n where n is the number of nodes in
    ↪ the tree
// Time complexity  $O(n \log n)$ 
// Space complexity  $O(n)$ 

// Given an unweighted tree with undirected edges and a function centroid_decomp
// implements the function on every decomposition

// see count_paths_per_node for example usage
struct centroid_decomp {
    vector<vector<int>> adj;
    function<void(const vector<vector<int>>&, int)> func;
    vector<int> subtree_sizes;

    centroid_decomp(const vector<vector<int>>& a_adj,
        const function<void(const vector<vector<int>>&, int)>& a_func)
        : adj(a_adj), func(a_func), subtree_sizes(adj.size()) {
        decomp(find_centroid(0));
    }

    void calc_subtree_sizes(int u, int p = -1) {
        subtree_sizes[u] = 1;
        for (auto v : adj[u]) {
            if (v == p)
                continue;
            calc_subtree_sizes(v, u);
            subtree_sizes[u] += subtree_sizes[v];
        }
    }

    int find_centroid(int root) {
        calc_subtree_sizes(root);
        auto dfs = [&](auto self, int u, int p) -> int {
            int biggest_ch = -1;
            for (auto v : adj[u]) {
                if (v == p)
                    continue;
                if (biggest_ch == -1 ||
                    subtree_sizes[biggest_ch] < subtree_sizes[v])
                    biggest_ch = v;
            }

            if (biggest_ch != -1 &&
                2 * subtree_sizes[biggest_ch] > subtree_sizes[root])
                return self(self, biggest_ch, u);
            return u;
        };
        return dfs(dfs, root, root);
    }

    void decomp(int root) {

```

```
func(adj, root);
for (auto v : adj[root]) {
    adj[v].erase(find(adj[v].begin(), adj[v].end(), root));
    decomp(find_centroid(v));
}
};
```

Frequency Table of Tree Distance

```
//cat count_paths_per_length.hpp | ./hash.sh
//274399
#pragma once
#include "../kactl/content/numerical/FastFourierTransform.h"
#include "centroid_decomp.hpp"
//returns array 'num_paths' where 'num_paths[i]' = # of paths in tree with 'i' edges
//O(n log^2 n)
vector<long long> count_paths_per_length(const vector<vector<int>>& a_adj/*unrooted,
    ↳ connected tree*/) {
    vector<long long> num_paths(a_adj.size(), 0);
    auto func = [&](const vector<vector<int>>& adj, int root) {
        vector<double> total_depth(1, 1.0);
        for (int to : adj[root]) {
            vector<double> cnt_depth(1, 0.0);
            for (queue<pair<int, int>> q({{to, root}}); !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) 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++) num_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];
    }
};
centroid_decomp decomp(a_adj, func);
return num_paths;
}
```

Count Paths Per Node

```
//cat count_paths_per_node.hpp | ./hash.sh
//f6c685
#pragma once
```

```
#include "centroid_decomp.hpp"
//0-based nodes
//returns array 'num_paths' where 'num_paths[i]' = number of paths with k edges where
    ↳ node 'i' is on the path
//O(n log n)
vector<long long> count_paths_per_node(const vector<vector<int>>& a_adj/*unrooted
    ↳ tree*/, int k) {
    vector<long long> num_paths(a_adj.size());
    auto func = [&](const vector<vector<int>>& adj, int root) {
        vector<int> pre_d(1, 1), cur_d(1);
        auto dfs = [&](auto self, int u, int p, int d) -> long long {
            if (d > k)
                return 0;

            if (int(cur_d.size()) <= d)
                cur_d.push_back(0);
            cur_d[d]++;

            long long cnt = 0;
            if (k - d < int(pre_d.size()))
                cnt += pre_d[k - d];

            for (int v : adj[u]) {
                if (v != p)
                    cnt += self(self, v, u, d + 1);
            }

            num_paths[u] += cnt;
            return cnt;
        };
        auto dfs_child = [&](int child) {
            auto cnt = dfs(dfs, child, root, 1);
            pre_d.resize(cur_d.size());
            for (int i = 1; i < int(cur_d.size()) && cur_d[i]; i++) {
                pre_d[i] += cur_d[i];
                cur_d[i] = 0;
            }
            return cnt;
        };
        for (int child : adj[root])
            num_paths[root] += dfs_child(child);
        pre_d = vector<int>(1);
        cur_d = vector<int>(1);
        for (auto it = adj[root].rbegin(); it != adj[root].rend(); it++)
            dfs_child(*it);
    };
    centroid_decomp decomp(a_adj, func);
    return num_paths;
}
```

Dijkstra

```
//cat dijkstra.hpp | ./hash.sh
//8fe9d3
#pragma once
//returns array 'len' where 'len[i]' = shortest path from node 'start' to node 'i'
//For example 'len[start]' 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 start) {
    using node = pair<long long, int>;
    vector<long long> len(adj.size(), INF);
    len[start] = 0;
    priority_queue<node, vector<node>, greater<node>> q;
    q.emplace(0, start);
    while (!q.empty()) {
        auto [curr_len, v] = q.top();
        q.pop();
        if (len[v] < curr_len) continue;
        for (auto [to, weight] : adj[v])
            if (len[to] > weight + len[v]) {
                len[to] = weight + len[v];
                q.emplace(len[to], to);
            }
    }
    return len;
}
```

HLD

```
//cat hld.hpp | ./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;
    }
}
};
```

Hopcroft Karp

```
//cat hopcroft_karp.hpp | ./hash.sh
//de75d7
#pragma once
//source:
    ↳ https://github.com/foreverbell/acm-icpc-cheat-sheet/blob/master/src/graph-algorithms/
//Worst case O(E*sqrt(V)) but faster in practice
struct match {
    // # of edges in matching (which = size of min vertex cover by König'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/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 <=> 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));
    }
}
```

Kth Node on Path

```
//cat kth_node_on_path.hpp | ./hash.sh
//7a4c3c
#pragma once
```

```
#include "lca.hpp"
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);
    }
};
```

LCA

```
//cat lca.hpp | ./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;
}
};
```

SCC

```
//cat scc.hpp | ./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};
}
```

TREE ISOMORPHISM

```
//cat subtree_isomorphism_classification.hpp | ./hash.sh
//0a7feb
#pragma once

// Complexity given in terms of n where n is the number of nodes in the tree
// Time complexity O(n log n)
// Space complexity O(n)

// Given an undirected or directed rooted tree
// rooted_subtree_isomorphism classifies each rooted subtree
struct isomorphic_classifications {
    int k;
    vector<int> ids;
    isomorphic_classifications(int n) : ids(n) {}
};

isomorphic_classifications subtree_isomorphism_classification(
    const vector<vector<int>>& adj, int root) {
    isomorphic_classifications classifications(adj.size());
    map<vector<int>, int> hashes;
    auto dfs = [&](auto self, int u, int p) -> int {
        vector<int> ch_ids;
        ch_ids.reserve(adj[u].size());
        for (auto v : adj[u]) {
            if (v != p)
                ch_ids.push_back(self(self, v, u));
        }
        sort(ch_ids.begin(), ch_ids.end());
        auto it = hashes.find(ch_ids);
        if (it == hashes.end())
            return classifications.ids[u] = hashes[ch_ids] = hashes.size();
        return classifications.ids[u] = it->second;
    };
    dfs(dfs, root, root);
    classifications.k = hashes.size();
    return classifications;
}
```

MATH

Derangements

```
//cat derangements.hpp | ./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;
}
```

BIN EXP MOD

```
//cat exp_mod.hpp | ./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;
}
```

Fibonacci

```
//cat fib.hpp | ./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];
}
```

Matrix Mult

```
//cat matrix_mult.hpp | ./hash.sh
//e4e421
#pragma once
// generic matrix multiplication (not overflow safe)
// will RTE if the given matrices are not compatible
// Time: O(n * m * inner)
// Space: O(n * m)

template<typename T> vector<vector<T>> operator * (const vector<vector<T>>& a, const
    ↪ vector<vector<T>>& b) {
    assert(a[0].size() == b.size());
    int n = a.size(), m = b[0].size(), inner = b.size();
    vector<vector<T>> c(n, vector<T>(m));
    for (int i = 0; i < n; i++) {
        for (int k = 0; k < inner; k++) {
            for (int j = 0; j < m; j++)
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
    return c;
}
```

Mobius Inversion

```
//cat mobius_inversion.hpp | ./hash.sh
//811515
#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;
int mobius[N];
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];
}
```

N Choose K MOD

```
//cat n_choose_k_mod.hpp | ./hash.sh
//f3a1a9
#pragma once
//for mod inverse
#include "exp_mod.hpp"
// 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;
};
```

Partitions

```
//cat partitions.hpp | ./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 as well)
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;
}
```

Prime Sieve

```
//cat prime_sieve.hpp | ./hash.sh
//45fc23
#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;
int a_prime[N];
void calc_seive() {
    iota(a_prime, a_prime + N, 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;
}
```

Row Reduce

```
//cat row_reduce.hpp | ./hash.sh
//1d7c3e
#pragma once
//for mod inverse
#include "exp_mod.hpp"
//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};
}
```

Solve Linear Equations MOD

```
//cat solve_linear_mod.hpp | ./hash.sh
//44cc6e
#pragma once
#include "row_reduce.hpp"
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};
}
```

Euler’s Totient Phi Function

```
//cat totient.hpp | ./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, pre-calc prime factors or 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;
}
```

MISC

Cartesian Tree

```
//cat cartesian_tree.hpp | ./hash.sh
//0b95bc
#pragma once
#include "monotonic_stack.hpp"
//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;
}
```

Count Rectangles

```
//cat count_rectangles.hpp | ./hash.sh
//b2cced
#pragma once
#include "monotonic_stack.hpp"
//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;
}
```

Max Rectangle in Histogram

```
//cat max_rect_histogram.hpp | ./hash.sh
//4e6291
#pragma once
#include "monotonic_stack.hpp"
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;
}
```

Monotonic Stack

```
//cat monotonic_stack.hpp | ./hash.sh
//4c7a40
#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<bool(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;
}
```

Iterate Chooses

```
//cat iterate_chooses.hpp | ./hash.sh
//c79083
#pragma once
// iterates all bitmasks of size n with k bits set
// Time Complexity: O(n choose k)
// Space Complexity: O(1)
int next_subset(int mask) {
    int c = mask & -mask, r = mask + c;
    return r | (((r ^ mask) >> 2) / c);
}
void iterate_chooses(int n, int k, const function<void(int)>& func) {
    for (int mask = (1 << k) - 1; mask < (1 << n); mask = next_subset(mask))
        func(mask);
}
```

Iterate Submasks

```
//cat iterate_submasks.hpp | ./hash.sh
//084c05
#pragma once

// iterates all submasks of mask
// Time Complexity: O(3^n) to iterate every submask of every mask of size n
// Space Complexity: O(1)

void iterate_submasks(int mask, const function<void(int)>& func) {
    for (int submask = mask; submask; submask = (submask - 1) & mask)
        func(submask);
}
```

Iterate Supermasks

```
//cat iterate_supermasks.hpp | ./hash.sh
//76b38f
#pragma once

// iterates all supermasks of mask
// Time Complexity: O(3^n) to iterate every supermask of every mask of size n
// Space Complexity: O(1)

void iterate_supermasks(int mask, int n, const function<void(int)>& func) {
    for (int supermask = mask; supermask < (1 << n); supermask = (supermask + 1) | mask)
        func(supermask);
}
```

LIS

```
//cat lis.hpp | ./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;
    return res;
}
```

Number of Distinct Subsequences DP

```
//cat num_distinct_subsequences.hpp | ./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];
}
```

PBDS

```
//cat policy_based_data_structures.hpp | ./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;
```

Random

```
//cat random.hpp | ./hash.sh
//61293c
#pragma once
```

```
//MUCH RANDOM!!!
seed_seq seed{
    (uint32_t)chrono::duration_cast<chrono::nanoseconds>(chrono::high_resolution_clock::now().time_since_epoch().count()),
    (uint32_t)random_device(),
    (uint32_t)(uintptr_t)make_unique<char>().get(),
    (uint32_t)::_builtin_ia32_rdtsc()
};
mt19937 rng(seed);

//intended types: int, unsigned, long long
//returns a random number in range [l, r)
template<class T> inline T get_rand(T l, T r) {
    assert(l < r);
    return uniform_int_distribution<T>(l, r - 1)(rng);
}
```

Safe Hash

```
//cat safe_hash.hpp | ./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.hpp"
gp_hash_table<long long, int, custom_hash> safe_hash_table;
```

```
int lch, rch; //children, indexes into 'tree'
};
const int N;
vector<int> ch(N);
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);
}
};
```

RANGE DATA STRUCTURES

Number Distinct Elements

```
//cat distinct_query.hpp | ./hash.sh
//6dfaad
#pragma once
//source:
//    https://cp-algorithms.com/data_structures/segment_tree.html#preserving-the-history-of-its-nodes
//works with negatives
//O(n log n) time and space
struct distinct_query {
    struct node {
        int sum;
```

Implicit Lazy Segment Tree

```
//cat implicit_seg_tree.hpp | ./hash.sh
//cbc0c0
#pragma once
//example initialization:
// implicit_seg_tree<10'000'000> ist(l, r);
template <int N> struct implicit_seg_tree {
    using dt = int; //long long, 2e9 //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) {}
} tree[N];
int ptr = 0, root_l, root_r; // [root_l, root_r) defines range of root node; handles
    ↪ negatives
implicit_seg_tree(int l, int r) : 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));
    }
};
```

Kth Smallest

```
//cat kth_smallest.hpp | ./hash.sh
//d28d16
#include <bits/stdc++.h>
using namespace std;
#pragma once
//source:
    ↪ https://cp-algorithms.com/data_structures/segment_tree.html#preserving-the-history-0
struct kth_smallest {
    struct node {
        int sum;
        int lch, rch; //children, indexes into 'tree'
    };
    int mn = INT_MAX, mx = INT_MIN;
    vector<int> roots;
    deque<node> tree;
    kth_smallest(const vector<int>& arr) : 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);
    }
};
```

Merge Sort Tree

```
//cat merge_sort_tree.hpp | ./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[i].reserve(l.size() + r.size());
            merge(l.begin(), l.end(), r.begin(), r.end(), back_inserter(tree[i]));
        }
    }
    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;
    }
};
```

BIT

```
//cat bit.hpp | ./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;
    }
};
```

RMQ

```
//cat rmq.hpp | ./hash.sh
//082180
#pragma once
//source:
⇨ https://github.com/kth-competitive-programming/kactl/blob/main/content/data-structures
//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)]);
}
};
```

Lazy Segment Tree

```
//cat seg_tree.hpp | ./hash.sh
//5fcd48
#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;
    seg_tree(const vector<dt>& arr) : N(arr.size()), S(N ? 1 << __lg(2 * N - 1) : 0),
        ↳ 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);
}
};
```

STRINGS

Binary Trie

```
//cat binary_trie.hpp | ./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;
}

};
```

KMP

```
//cat kmp.hpp | ./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
//& 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(pf(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
};
```

Longest Common Prefix Query

```
//cat lcp_query.hpp | ./hash.sh
//96594e
#pragma once
#include "../ac-library/atcoder/string.hpp"
#include "../range_data_structures/rmq.hpp"
//computes suffix array, lcp array, and then sparse table over lcp array
//O(n log n)
template<typename T> struct lcp_query {
    const int N;
    vector<int> sa, lcp, inv_sa;
    RMQ<int> st;
    lcp_query(const T& 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];
}
};
```

Palindrome Query

```
//cat palindrome_query.hpp | ./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
    //(returns 1 when l == r)
    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;
    }
};
```

Trie

```
//cat trie.hpp | ./hash.sh
//fd9c8d
#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 {
        int next[K], cnt_words = 0, par = -1;
        char ch;
        node(int a_par = -1, char a_ch = '#') : par(a_par), ch(a_ch) {
            fill(next, next + K, -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;
}
};
```

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
```

