

Listings

1	CONTEST	1	38	Count Rectangles	12
2	Tips and Tricks	1	39	Max Rectangle in Histogram	13
3	Hash codes	1	40	Monotonic Stack	13
4	Test on random inputs	2	41	GCD Convolution	13
5	MAX FLOW	2	42	Iterate Chooses	13
6	Hungarian	2	43	Iterate Submasks	14
7	Min Cost Max Flow	2	44	Iterate Supermasks	14
8	GRAPHS	3	45	Number of Distinct Subsequences DP	14
9	Block Vertex Tree	3	46	PBDS	14
10	Bridge Tree	3	47	Random	14
11	Bridges and Cuts	3	48	Safe Hash	14
12	Enumerate Triangles	4	49	RANGE DATA STRUCTURES	15
13	Strongly Connected Components	5	50	Number Distinct Elements	15
14	Centroid Decomposition	5	51	Implicit Lazy Segment Tree	15
15	Frequency Table of Tree Distance	5	52	Kth Smallest	16
16	Count Paths Per Node	6	53	Merge Sort Tree	16
17	Dijkstra	6	54	BIT	17
18	HLD	6	55	RMQ	17
19	Hopcroft Karp	7	56	Disjoint RMQ	18
20	Kth Node on Path	8	57	Lazy Segment Tree	18
21	LCA	8	58	STRINGS	19
22	Rooted Tree Isomorphism	9	59	Binary Trie	19
23	MATH	9	60	Prefix Function	19
24	Derangements	9	61	KMP String Matching	20
25	Binary Exponentiation MOD	9	62	Suffix and LCP Arrays	20
26	Fibonacci	9	63	Suffix Array Related Queries	21
27	Matrix Multiplication	10	64	Palindrome Query	21
28	Mobius Inversion	10	65	Trie	22
29	N Choose K MOD	10			
30	Partitions	10			
31	Prime Sieve	11			
32	Row Reduce	11			
33	Solve Linear Equations MOD	11			
34	Euler's Totient Phi Function	12			
35	Tetration MOD	12			
36	MISC	12			
37	Cartesian Tree	12			

CONTEST

Tips and Tricks

```
## Tips and Tricks
- [C++ tips and tricks](https://codeforces.com/blog/entry/74684)
- invokes RTE (Run Time Error) upon integer overflow
...

#pragma GCC optimize "trapv"
...

- invoke RTE for input error (e.g. reading a long long into an int)
...
cin.exceptions(cin.failbit);
...

- use pramgas for C++ speed boost
...

#pragma GCC optimize("O3,unroll-loops")
#pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
...

### Troubleshooting
...

/* stuff you should look for
 * int overflow, array bounds
 * special cases (n=1?)
 * do smth instead of nothing and stay organized
 * WRITE STUFF DOWN
 * DON'T GET STUCK ON ONE APPROACH
 */
...

Author: Benq

- refer to [KACTL
  ↳ Troubleshoot](https://github.com/kth-competitive-programming/kactl/blob/main/content/contest-tips-and-tricks/troubleshoot.txt)

## Sources

- [[Tutorial] GCC Optimization Pragmas](https://codeforces.com/blog/entry/96344)
- [Don't use rand(): a guide to random number generators in
  ↳ C++](https://codeforces.com/blog/entry/61587)
```

Hash codes

```
#!/bin/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
```

Test on random inputs

```
#!/bin/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

Hungarian

```
//cat hungarian.hpp | ./hash.sh
//935a16
const long long INF = 1e18;
struct weighted_match {
    long long min_cost;
    vector<int> matching; //worker `i` (1<=i<=n) is assigned to job `matching[i]`
    ↳ (1<=matching[i]<=m)
};

/**
 * Troubleshoot.txt
 * [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* sum of costs to assign each worker to some distinct job.
 *
 * Set `cost[i][j]` to INF to say: "worker i cannot be assigned job j".
 * This works for negatives, so negating cost array gives max matching.
 *
 * @see https://e-maxx.ru/algorithm/assignment_hungary
 * @time O(n^2 * m)
 * @memory O(n * m)
 */
weighted_match hungarian(const vector<vector<long long>>& cost) {
    int n = ssize(cost) - 1, m = ssize(cost[0]) - 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};
}
```

Min Cost Max Flow

```
//cat min_cost_max_flow.hpp | ./hash.sh
//9dd6b6
const long long INF = 1e18;
/**
 * @see https://e-maxx.ru/algo/min_cost_flow
 */
struct mcmf {
    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;
mcmf(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, ssize(g[b])};
    edge e2 = {b, a, 0, -cost, 0, ssize(g[a])};
    g[a].push_back(ssize(e));
    e.push_back(e1);
    g[b].push_back(ssize(e));
    e.push_back(e2);
}
pair<ll, 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 < ssize(g[v]); 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, cost};
};
```

GRAPHS

Block Vertex Tree

```
//cat block_vertex_tree.hpp | ./hash.sh
//a5c2b9
#include "bridges_and_cuts.hpp"
/**
 * Returns adjacency list of block vertex tree.
 * @code{.cpp}
 * graph_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]) {}
* @endcode
* @time O(n + m)
* @memory O(n + m)
*/
vector<vector<int>> block_vertex_tree(const vector<vector<pair<int, int>>& adj, const
    ↳ graph_info& cc) {
    int n = ssize(adj);
    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;
}
```

Bridge Tree

```
//cat bridge_tree.hpp | ./hash.sh
//8eb014
#include "bridges_and_cuts.hpp"
/**
 * Never adds multiple edges as bridges_and_cuts.hpp correctly marks them as
 * non-bridges.
 * @code{.cpp}
 *     graph_info cc = bridge_and_cut(adj, m);
 *     vector<vector<int>> bt = bridge_tree(adj, cc);
 * @endcode
 * @time O(n + m)
 * @memory O(n + m)
 */
vector<vector<int>> bridge_tree(const vector<vector<pair<int, int>>& adj, const graph_info&
    ↳ cc) {
    vector<vector<int>> tree(cc.num_2_edge_ccs);
    for (int i = 0; i < ssize(adj); 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
//3f21b9
struct graph_info {
    //2 edge connected component stuff (e.g. components split by bridge edges)
    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)
    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)
};
/**
 * @code{.cpp}
 *     //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);
 *     }
 * @endcode
 * @see https://cp-algorithms.com/graph/bridge-searching.html
 * https://cp-algorithms.com/graph/cutpoints.html
 * @time O(n + m)
 * @memory O(n + m)
 */
graph_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 = ssize(adj), 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;
    node_stack.reserve(n);
    //bcc stuff (delete if not needed)
    int num_bccs = 0;
    vector<bool> is_cut(n, 0);
    vector<int> bcc_id(m), edge_stack;
    edge_stack.reserve(m);
    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};
}
```

Enumerate Triangles

```
//cat enumerate_triangles.hpp | ./hash.sh
//689510
/**
 * Given an edge list representing an undirected graph and a function, 'enumerate_triangles'
 *   ↪ will
 * call the given function on every triangle in the undirected graph exactly once
 *
 * @time O(n + m ^ (3/2))
 * @memory O(n + m)
 */
void enumerate_triangles(const vector<pair<int, int>>& edges, int n /*# of nodes*/, const
    ↪ function<void(int, int, int)>& f) {
    vector<int> deg(n);
    for (auto [u, v] : edges)
        deg[u]++, deg[v]++;
    vector<vector<int>> adj(n);
    for (auto [u, v] : edges) {
        if (tie(deg[u], u) > tie(deg[v], v))
            swap(u, v);
        adj[u].push_back(v);
    }
    vector<bool> seen(n);
    for (auto [u, v] : edges) {
        for (auto w : adj[u])
            seen[w] = 1;
        for (auto w : adj[v])
            if (seen[w])
```

```
        f(u, v, w);
        for (auto w : adj[u])
            seen[w] = 0;
    }
}
```

Strongly Connected Components

```
//cat strongly_connected_components.hpp | ./hash.sh
//f6f849
//source: https://github.com/kth-competitive-programming/
// kactl/blob/main/content/graph/SCC.h
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] (reverse 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 = ssize(adj), timer = 1, num_sccs = 0;
    vector<int> tin(n, 0), scc_id(n, -1), node_stack;
    node_stack.reserve(n);
    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};
}
```

Centroid Decomposition

```
//cat centroid_decomposition.hpp | ./hash.sh
//71e2e5
/**
 * Given an unweighted, undirected forest and a function, centroid_decomp runs
 * the function on the centroid of every decomposition.
 * @code{.cpp}
 * //example usage
 * centroid_decomp decomp(adj, [&](const vector<vector<int>>& adj_removed_edges, int cent)
    ↪ -> void {
```

```
*    });
* @endcode
* @time O(n log n)
* @memory O(n)
*/
template <typename F> struct centroid_decomp {
    vector<vector<int>>> adj;
    F func;
    vector<int> sub_sz;
    centroid_decomp(const vector<vector<int>>& a_adj/*undirected forest*/, const F& a_func)
        : adj(a_adj), func(a_func), sub_sz(ssize(adj), -1) {
        for (int i = 0; i < ssize(adj); i++)
            if (sub_sz[i] == -1)
                dfs(i);
    }
    void calc_subtree_sizes(int u, int p = -1) {
        sub_sz[u] = 1;
        for (int v : adj[u]) {
            if (v == p) continue;
            calc_subtree_sizes(v, u);
            sub_sz[u] += sub_sz[v];
        }
    }
    void dfs(int u) {
        calc_subtree_sizes(u);
        for (int p = -1, sz_root = sub_sz[u];) {
            auto big_ch = find_if(adj[u].begin(), adj[u].end(), [&](int v) -> bool {
                return v != p && 2 * sub_sz[v] > sz_root;
            });
            if (big_ch == adj[u].end()) break;
            p = u, u = *big_ch;
        }
        func(adj, u);
        for (int v : adj[u]) {
            //each node is adjacent to O(logn) centroids
            adj[v].erase(find(adj[v].begin(), adj[v].end(), u));
            dfs(v);
        }
    }
};
```

Frequency Table of Tree Distance

```
//cat count_paths_per_length.hpp | ./hash.sh
//182b70
#include "../kactl/content/numerical/FastFourierTransform.h"
#include "centroid_decomposition.hpp"
/**
 * Returns array `num_paths` where `num_paths[i]` = # of paths in tree with `i`
 * edges.
 * @time O(n log^2 n)
 */
vector<long long> count_paths_per_length(const vector<vector<int>>& adj/*unrooted, connected
    ⇨ tree*/) {
    vector<long long> num_paths(ssize(adj), 0);
    centroid_decomp decomp(adj, [&](const vector<vector<int>>& adj_removed_edges, int cent) ->
        ⇨ void {
        vector<vector<double>> child_depths;
```

```
        for (int to : adj_removed_edges[cent]) {
            child_depths.emplace_back(1, 0.0);
            for (queue<pair<int, int>> q{{to, cent}}; !q.empty();) {
                child_depths.back().push_back(ssize(q));
                queue<pair<int, int>> new_q;
                while (!q.empty()) {
                    auto [curr, par] = q.front();
                    q.pop();
                    for (int ch : adj_removed_edges[curr]) {
                        if (ch == par) continue;
                        new_q.emplace(ch, curr);
                    }
                }
                swap(q, new_q);
            }
        }
        sort(child_depths.begin(), child_depths.end(), [&](const auto & x, const auto & y) {
            return x.size() < y.size();
        });
        vector<double> total_depth(1, 1.0);
        for (const auto& cnt_depth : child_depths) {
            auto prod = conv(total_depth, cnt_depth);
            for (int i = 1; i < ssize(prod); i++)
                num_paths[i] += llround(prod[i]);
            total_depth.resize(ssize(cnt_depth), 0.0);
            for (int i = 1; i < ssize(cnt_depth); i++)
                total_depth[i] += cnt_depth[i];
        }
    });
    return num_paths;
}
```

Count Paths Per Node

```
//cat count_paths_per_node.hpp | ./hash.sh
//de045c
#include "centroid_decomposition.hpp"
/**
 * Returns array `num_paths` where `num_paths[i]` = number of paths with k
 * edges where node `i` is on the path. 0-based nodes.
 * @time O(n log n)
 */
vector<long long> count_paths_per_node(const vector<vector<int>>& adj/*unrooted tree*/, int k)
    ⇨ {
    vector<long long> num_paths(ssize(adj));
    centroid_decomp decomp(adj, [&](const vector<vector<int>>& adj_removed_edges, int cent) ->
        ⇨ void {
        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 (ssize(cur_d) <= d) cur_d.push_back(0);
            cur_d[d]++;
            long long cnt = 0;
            if (k - d < ssize(pre_d)) cnt += pre_d[k - d];
            for (int v : adj_removed_edges[u])
                if (v != p)
                    cnt += self(self, v, u, d + 1);
            num_paths[u] += cnt;
        };
    });
}
```

```
        return cnt;
    };
    auto dfs_child = [&](int child) -> long long {
        long long cnt = dfs(dfs, child, cent, 1);
        pre_d.resize(ssize(cur_d));
        for (int i = 1; i < ssize(cur_d) && cur_d[i]; i++)
            pre_d[i] += cur_d[i], cur_d[i] = 0;
        return cnt;
    };
    for (int child : adj_removed_edges[cent])
        num_paths[cent] += dfs_child(child);
    pre_d = vector<int>(1);
    cur_d = vector<int>(1);
    for_each(adj_removed_edges[cent].rbegin(), adj_removed_edges[cent].rend(), dfs_child);
});
return num_paths;
}
```

Dijkstra

```
//cat dijkstra.hpp | ./hash.sh
//aa6eda
const long long INF = 1e18;
/**
 * Returns array `len` where `len[i]` = shortest path from node `start` to node
 * `i`. For example `len[start]` = 0.
 *
 * @time O((n + m) log n) Note log(m) < log(n^2) = 2*log(n), so O(log n) ==
 * O(log m)
 * @memory O(n + m)
 */
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(ssize(adj), 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; //important check: O(n*m) without it
        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
//27b19f
/**
 * @see https://codeforces.com/blog/entry/53170
```

```
*/
//NOLINTNEXTLINE(readability-identifier-naming)
struct HLD {
    struct node {
        int sub_sz = 1, par = -1, time_in = -1, next = -1;
    };
    vector<node> tree;
    /**
     * @time O(n)
     * @memory O(n)
     */
    HLD(vector<vector<int>>& adj /*forest of unrooted trees*/) : tree(ssize(adj)) {
        for (int i = 0, timer = 0; i < ssize(adj); i++) {
            if (tree[i].next == -1) { //lowest indexed node in each tree becomes root
                tree[i].next = i;
                dfs1(i, adj);
                dfs2(i, adj, timer);
            }
        }
    }
    void dfs1(int v, vector<vector<int>>& adj) {
        for (int& to : adj[v]) {
            adj[to].erase(find(adj[to].begin(), adj[to].end(), 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. u, v must be in the
     * same component.
     * @time O(log n)
     * @memory O(log n)
     */
    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);
        }
    }
    pair<int, int> subtree(int i) const {
        return {tree[i].time_in, tree[i].time_in + tree[i].sub_sz};
    }
    /**
```

```
* u, v must be in the same component.
* @time O(log n)
* @memory O(1)
*/
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
//5d1682
struct match {
    //# of edges in matching (which = size of min vertex cover by öKnig's theorem)
    int size_of_matching;
    //an arbitrary max matching is found. For this matching:
    //if l_to_r[node_left] == -1:
    // node_left is not in matching
    //else:
    // the edge `node_left` <=> l_to_r[node_left] is in the matching
    //
    //similarly for r_to_l with edge r_to_l[node_right] <=> node_right in matching if
    //    ↪ r_to_l[node_right] != -1
    //matchings stored in l_to_r and r_to_l are the same matching
    //provides way to check if any node/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.
 * @code{.cpp}
 * //for every edge node_left <=> node_right
 * adj[node_left].push_back(node_right);
 * @endcode
 * @see https://github.com/foreverbell/acm-icpc-cheat-sheet/
 * blob/master/src/graph-algorithm/hopcroft-karp.cpp
 * @time O(m * sqrt(n)) n = lsz + rsz
 * @memory O(n + m)
 */
match hopcroft_karp(const vector<vector<int>>& adj/*bipartite graph*/, int rsz/*number of
    ↪ nodes on right side*/) {
    int size_of_matching = 0, lsz = ssize(adj);
    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
//c59307
#include "lca.hpp"
struct kth_node_on_path {
    LCA lca;
    kth_node_on_path(const vector<vector<pair<int, long long>>&& adj/*forest of weighted
        ↪ trees*/) : lca(adj) {}
};
/**
 * Consider path {u, u's par, ..., LCA(u,v), ..., v's par, v}. This returns
 * the node at index k. So k=0 returns u, k=#path_edges returns v.
 * u, v must be in the same component.
 * @time O(log n)
 */
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
//b28532
/**
 * @see https://codeforces.com/blog/entry/74847
 */
//NOLINTNEXTLINE(readability-identifier-naming)
struct LCA {
    struct node {
        int jmp = -1, jmp_edges = 0, par = -1, depth = 0;
        long long dist = 0LL;
    };
    vector<node> tree;
    /**
     * @time O(n)
     * @memory O(n)
     */
    LCA(const vector<vector<pair<int, long long>>>& adj /*forest of weighted trees*/) :
        tree(ssize(adj)) {
        for (int i = 0; i < ssize(adj); i++) {
            if (tree[i].jmp == -1) { //lowest indexed node in each tree becomes root
                tree[i].jmp = i;
                dfs(i, adj);
            }
        }
    }

    void dfs(int v, const vector<vector<pair<int, long long>>>& adj) {
        int jmp, jmp_edges;
        if (tree[v].jmp != v && 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. So with k=1 this returns v's parent.
     * @time O(log k)
     */
    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;
}
/**
 * x, y must be in the same component.
 * @time O(log n)
 */
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;
}
};
```

Rooted Tree Isomorphism

```
//cat subtree_isomorphism.hpp | ./hash.sh
//455aef
struct iso_info {
    int num_distinct_subtrees; //0 <= id[i] < num_distinct_subtrees
    vector<int> id; //id[u] == id[v] iff rooted subtree u is isomorphic to rooted subtree v
};
/**
 * Classifies each rooted subtree by isomorphism.
 * @time O(n log n)
 * @memory O(n)
 */
iso_info subtree_iso(const vector<vector<int>>>& adj) {
    vector<int> id(ssize(adj), -1);
    map<vector<int>, int> hashes;
    auto dfs = [&](auto self, int u, int p) -> int {
        vector<int> ch_ids;
        ch_ids.reserve(ssize(adj[u]));
        for (int 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 id[u] = hashes[ch_ids] = ssize(hashes);
        return id[u] = it->second;
    };
    for (int i = 0; i < ssize(adj); i++)
        if (id[i] == -1)
            dfs(dfs, i, i);
}
```

```
    return {ssize(hashes), id};
}
```

MATH

Derangements

```
//cat derangements.hpp | ./hash.sh
//64d325
/**
 * @see https://oeis.org/A000166
 * @time O(n)
 */
vector<long long> derangements(int n, long long mod) {
    vector<long long> dp(n, 0);
    dp[0] = 1;
    for (int i = 2; i < n; i++)
        dp[i] = (i - 1) * (dp[i - 1] + dp[i - 2]) % mod;
    return dp;
}
```

Binary Exponentiation MOD

```
//cat binary_exponentiation_mod.hpp | ./hash.sh
//92a3ef
/**
 * Returns (base^pw)%mod; returns 1 for 0^0.
 * @time O(log pw)
 */
long long bin_exp(long long base, long long pw, long long mod) {
    assert(0 <= pw && 0 <= base && 1 <= mod);
    long long res = 1;
    base %= mod;
    while (pw > 0) {
        if (pw & 1) res = res * base % mod;
        base = base * base % mod;
        pw >>= 1;
    }
    return res;
}
```

Fibonacci

```
//cat fibonacci.hpp | ./hash.sh
//78a41f
//https://codeforces.com/blog/entry/14516
//O(log(n))
unordered_map<long long, long long> table;
long long fib(long long n, long long mod) {
    if (n < 2) return 1;
    if (table.find(n) != table.end()) return table[n];
    table[n] = (fib((n + 1) / 2, mod) * fib(n / 2, mod) + fib((n - 1) / 2, mod) * fib((n - 2) / 2, mod)) % mod;
    return table[n];
}
```

Matrix Multiplication

```
//cat matrix_mult.hpp | ./hash.sh
//910018
/**
 * Generic matrix multiplication (not overflow safe). This will RTE if the
 * given matrices are not compatible.
 * @see https://codeforces.com/blog/entry/80195
 * @time O(n * m * inner)
 * @memory O(n * m)
 */
template <typename T> vector<vector<T>> operator * (const vector<vector<T>>& a, const
    vector<vector<T>>& b) {
    assert(ssize(a[0]) == ssize(b));
    int n = ssize(a), m = ssize(b[0]), inner = ssize(b);
    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] += a[i][k] * b[k][j];
    return c;
}
```

Mobius Inversion

```
//cat mobius_inversion.hpp | ./hash.sh
//811515
const int N = 1e6 + 10;
int mobius[N];
/**
 * 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
 * @time O(n log 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
//daaaa8
#include "binary_exponentiation_mod.hpp"
/**
 * @code{.cpp}
 *     n_choose_k nk(n, 1e9+7); // to use `choose`, `inv` with inputs strictly < n
 *     n_choose_k nk(mod, mod); // to use `choose_lucas` with arbitrarily large inputs
 * @endcode
 */
struct n_choose_k {
    long long mod;
    vector<long long> fact, inv_fact;
    /**
     * Only works for `n <= mod` and prime mod.
     */
};
```

```
* @time O(n + sqrt(mod)) The sqrt is only to assert mod is prime.
* @memory O(n)
*/
n_choose_k(int n, long long a_mod) : mod(a_mod), fact(n, 1), inv_fact(n, 1) {
    assert(max(n, 2) <= mod);
    for (int i = 2; i * i <= mod; i++) assert(mod % i);
    for (int i = 2; i < n; i++)
        fact[i] = fact[i - 1] * i % mod;
    inv_fact.back() = bin_exp(fact.back(), mod - 2, mod);
    for (int i = n - 2; i >= 2; i--)
        inv_fact[i] = inv_fact[i + 1] * (i + 1) % mod;
}
/**
 * n choose k for n,k < ssize(fact). Fails when n >= mod.
 * @time O(1)
 * @memory O(n) precomp
 */
long long choose(int n, int k) const {
    if (k < 0 || k > n) return 0;
    //now we know 0 <= k <= n so 0 <= n
    return fact[n] * inv_fact[k] % mod * inv_fact[n - k] % mod;
}
/**
 * Lucas theorem - n choose k for n, k up to LONGLONG_MAX. Handles n>=mod
 * correctly.
 * @time O(log(k))
 * @memory O(mod) precomp, so can't use 1e9 + 7.
 */
long long choose_lucas(long long n, long long k) const {
    if (k < 0 || k > n) return 0;
    if (k == 0 || k == n) return 1;
    return choose_lucas(n / mod, k / mod) * choose(int(n % mod), int(k % mod)) % mod;
}
/**
 * Returns x such that x * n % mod == 1.
 * @time O(1)
 */
long long inv(int n) const {
    assert(1 <= n); //don't divide by 0 :)
    return fact[n - 1] * inv_fact[n] % mod;
}
};
```

Partitions

```
//cat partitions.hpp | ./hash.sh
//e7ae42
//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<long long> partitions(int n, long long mod) {
    vector<long long> 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
//25a877
bool is_prime(int val, const vector<int>& sieve) {
    assert(val < ssize(sieve));
    return val >= 2 && sieve[val] == val;
}
vector<int> get_prime_factors(int val, const vector<int>& sieve) {
    assert(val < ssize(sieve));
    vector<int> factors;
    while (val > 1) {
        int p = sieve[val];
        factors.push_back(p);
        val /= p;
    }
    return factors;
}
/**
 * Returns array `sieve` where `sieve[i]` = some prime factor of `i`.
 * @time O(n * log(logn))
 */
vector<int> get_sieve(int n) {
    vector<int> sieve(n);
    iota(sieve.begin(), sieve.end(), 0);
    for (int i = 2; i * i < n; i++)
        if (sieve[i] == i)
            for (int j = i * i; j < n; j += i)
                sieve[j] = i;
    return sieve;
}
```

Row Reduce

```
//cat row_reduce.hpp | ./hash.sh
//ac160e
#include "binary_exponentiation_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.
 * @code{.cpp}
 *     auto [rank, det] = row_reduce(mat, ssize(mat[0]), mod);
 * @endcode
 * @time O(n * m * min(cols, n))
 * @memory O(n * m)
 */
pair<int/*rank*/, long long/*determinant*/> row_reduce(vector<vector<long long>>& mat, int
  ↪ cols, long long mod) {
    int n = ssize(mat), m = ssize(mat[0]), rank = 0;
    long long det = 1;
    assert(cols <= m);
    for (int col = 0; col < cols && rank < n; col++) {
```

```
//find arbitrary pivot and swap pivot to current row
auto it = find_if(mat.begin() + rank, mat.end(), [&](const auto & v) {return v[col];});
if (it == mat.end()) {
    det = 0;
    continue;
}
if (it != mat.begin() + rank) {
    det = det == 0 ? 0 : mod - det;
    iter_swap(mat.begin() + rank, it);
}
det = det * mat[rank][col] % mod;
//make pivot 1 by dividing row by inverse of pivot
long long a_inv = bin_exp(mat[rank][col], mod - 2, mod);
transform(mat[rank].begin(), mat[rank].end(), mat[rank].begin(), [&](auto val) {
    return val * a_inv % mod;
});
//zero-out all numbers above & below pivot
for (int i = 0; i < n; i++)
    if (i != rank && mat[i][col] != 0) {
        long long val = mat[i][col];
        transform(mat[i].begin(), mat[i].end(), mat[rank].begin(), mat[i].begin(),
            ↳ [&](auto x, auto y) {
                return (x + (mod - y) * val) % mod;
            });
    }
    rank++;
}
assert(rank <= min(n, cols));
return {rank, det};
}
```

Solve Linear Equations MOD

```
//cat solve_linear_mod.hpp | ./hash.sh
//109fff
#include "row_reduce.hpp"
struct matrix_info {
    int rank;
    long long det;
    //solution vector
    //empty if no solution exists
    //if there are multiple solutions, this is an arbitrary one
    vector<long long> 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. Leaves mat in reduced row echelon form with b
 * appended.
 *
 * Number of unique solutions = (size of domain) ^ (# of free variables).
 * (# of free variables) is generally equivalent to n - rank.
 * @time O(n * m * min(n, m))
 * @memory O(n * m)
 */
matrix_info solve_linear_mod(vector<vector<long long>>& mat, const vector<long long>& b, long
    ↳ long mod) {
```

```
assert(ssize(mat) == ssize(b));
int n = ssize(mat), m = ssize(mat[0]);
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
if (any_of(mat.begin() + rank, mat.end(), [](const auto & v) {return v.back();})) {
    return {rank, det, {}}; //no solution exists
}
//initialize solution vector (`x`) from row-reduced matrix
vector<long long> x(m, 0);
int j = 0;
for_each(mat.begin(), mat.begin() + rank, [&](const auto & v) {
    while (v[j] == 0) j++; //find pivot column
    x[j] = v.back();
});
return {rank, det, x};
}
```

Euler’s Totient Phi Function

```
//cat totient.hpp | ./hash.sh
//36bd41
/**
 * 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.
 * @time O(sqrt n)
 */
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;
}
```

Tetration MOD

```
//cat tetration_mod.hpp | ./hash.sh
//e2153e
#include "binary_exponentiation_mod.hpp"
#include "totient.hpp"
/**
 * Returns base ^ (base ^ (base ^ ... )) % mod, where the height of the tower
 * is pw.
 * Let t = totient(mod).
 * If log2(mod) <= pw then (base^pw)%mod == (base^(t+(pw/t)))%mod
 * @see https://cp-algorithms.com/algebra/phi-function.html#generalization
 * @time O(sqrt(mod) * log(mod))
 */
long long tetration(long long base, long long pw, long long mod) {
    if (mod == 1)
        return 0;
```

```
if (base == 0)
    return (pw + 1) % 2 % mod;
if (base == 1 || pw == 0)
    return 1;
if (pw == 1)
    return base % mod;
if (base == 2 && pw == 2)
    return 4 % mod;
if (base == 2 && pw == 3)
    return 16 % mod;
if (base == 3 && pw == 2)
    return 27 % mod;

//need enough base cases such that the following is true
//log2(mod) <= tetration(base, pw - 1) (before modding)
int t = totient(int(mod));
long long exp = tetration(base, pw - 1, t);
return bin_exp(base, exp + t, mod);
}
```

MISC

Cartesian Tree

```
//cat cartesian_tree.hpp | ./hash.sh
//d63975
#include "monotonic_stack.hpp"
/**
 * Min cartesian tree - root stores min.
 * @time O(n)
 * @memory O(n)
 */
vector<int> cartesian_tree(const vector<int>& arr) {
    int n = ssize(arr);
    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);
    transform(left.begin(), left.end(), right.rbegin(), par.begin(), [&](int le, int ri) {
        ri = n - 1 - ri;
        if (le >= 0 && ri < n) return arr[le] > arr[ri] ? le : ri;
        if (le >= 0) return le;
        if (ri < n) return ri;
        return -1; //root
    });
    return par;
}
```

Count Rectangles

```
//cat count_rectangles.hpp | ./hash.sh
//c5179a
#include "monotonic_stack.hpp"
/**
 * Given a n-by-m boolean matrix, calculate cnt[i][j]. cnt[i][j] = the number
 * of times an i-by-j sub rectangle appears in the matrix such that all i*j
 * cells in the sub rectangle are 1.
 *
 */
```

```
* cnt[0][j] and cnt[i][0] will contain garbage values.
* @time O(n * m)
* @memory O(n * m)
*/
vector<vector<int>> count_rectangles(const vector<vector<bool>>& grid) {
    int n = ssize(grid), m = ssize(grid[0]);
    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++) {
        transform(arr.begin(), arr.end(), grid[i].begin(), arr.begin(), [](int a, bool g) {
            return g * (a + 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 le = j - left[j] - 1, ri = rv(right[rv(j)]) - j - 1;
            cnt[arr[j]][le + ri + 1]++;
            cnt[arr[j]][le]--;
            cnt[arr[j]][ri]--;
        }
    }
    for (int i = 1; i <= n; i++)
        for (int j = 0; j < 2; j++)
            partial_sum(cnt[i].rbegin(), cnt[i].rend() - 1, cnt[i].rbegin());
    for (int i = n - 1; i >= 1; i--)
        transform(cnt[i].begin(), cnt[i].end(), cnt[i + 1].begin(), cnt[i].begin(), [](int x,
            ⇨ int y) {return x + y;});
    return cnt;
}
```

Max Rectangle in Histogram

```
//cat max_rect_histogram.hpp | ./hash.sh
//95288f
#include "monotonic_stack.hpp"
/**
 * @time O(n)
 * @memory O(n)
 */
long long max_rect_histogram(const vector<int>& arr) {
    auto rv /*reverse*/ = [&](int i) -> int {
        return ssize(arr) - 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 < ssize(arr); i++) {
        int le = left[i], ri = rv(right[rv(i)]); //arr[i] is the max of range (le, ri)
        max_area = max(max_area, 1LL * arr[i] * (ri - le - 1));
    }
    return max_area;
}
```

Monotonic Stack

```
//cat monotonic_stack.hpp | ./hash.sh
//35c95c
/**
 * Returns array `le` where `le[i]` = max number such that: `le[i]` < i and
 * !op(arr[le[i]], arr[i]). Returns -1 if no number exists.
 * @code{.cpp}
 *     vector<int> le = monotonic_stack<int>(arr, less()); //(or replace `less` with:
 *         ↳ less_equal, greater, greater_equal
 *     vector<int> le = monotonic_stack<int>(arr, [&](int x, int y) {return x < y;});
 * @endcode
 * @time O(n)
 * @memory O(n)
 */
template <typename T> vector<int> monotonic_stack(const vector<T>& arr, const
    ↳ function<bool(const T&, const T&)>& op) {
    vector<int> le(ssize(arr));
    for (int i = 0; i < ssize(arr); i++) {
        le[i] = i - 1;
        while (le[i] >= 0 && op(arr[le[i]], arr[i])) le[i] = le[le[i]];
    }
    return le;
}
```

GCD Convolution

```
//cat gcd_convolution.hpp | ./hash.sh
//d92c44
/**
 * Returns array `c` where `c[k]` = the sum for all pairs where gcd(i,j) == k
 * of a[i] * b[j].
 * @time O(n log n)
 * @memory O(n)
 */
template<int MOD> vector<int> gcd_convolution(const vector<int>& a, const vector<int>& b) {
    assert(ssize(a) == ssize(b));
    int n = ssize(a);
    vector<int> c(n);
    for (int gcd = n - 1; gcd >= 1; gcd--) {
        int sum_a = 0, sum_b = 0;
        for (int i = gcd; i < n; i += gcd) {
            sum_a = (sum_a + a[i]) % MOD, sum_b = (sum_b + b[i]) % MOD;
            c[gcd] = (c[gcd] - c[i] + MOD) % MOD;
        }
        c[gcd] = int((c[gcd] + 1LL * sum_a * sum_b) % MOD);
    }
    return c;
}
```

Iterate Chooses

```
//cat iterate_chooses.hpp | ./hash.sh
//c79083

int next_subset(int mask) {
    int c = mask & -mask, r = mask + c;
    return r | (((r ^ mask) >> 2) / c);
}
```

```
}
/**
 * Iterates over all bitmasks of size n with k bits set.
 * @see https://github.com/kth-competitive-programming/
 *     kactl/blob/main/content/various/chapter.tex
 * @time O(n choose k)
 * @memory O(1)
 */
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
/**
 * Iterates over all submasks of mask.
 * @time O(3^n) to iterate every submask of every mask of size n
 * @memory 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
/**
 * Iterates over all supermasks of mask.
 * @time O(3^n) to iterate every supermask of every mask of size n
 * @memory 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);
}
```

Number of Distinct Subsequences DP

```
//cat num_distinct_subsequences.hpp | ./hash.sh
//d94bdc
/**
 * Returns the number of distinct subsequences of `arr`. The empty subsequence
 * is counted.
 * @time O(n log n)
 * @memory O(n)
 */
int num_subsequences(const vector<int>& arr, int mod) {
    vector<int> dp(ssize(arr) + 1, 1);
    map<int, int> last;
    for (int i = 0; i < ssize(arr); 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.back();
}
```

PBDS

```
//cat policy_based_data_structures.hpp | ./hash.sh
//a777d7
//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 <typename 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
//ab9111

//source: https://codeforces.com/blog/entry/61675

mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());

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

//vector<int> a;
//shuffle(a.begin(), a.end(), rng);
```

Safe Hash

```
//cat safe_hash.hpp | ./hash.sh
//d9ea53
//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;
```

RANGE DATA STRUCTURES

Number Distinct Elements

```
//cat distinct_query.hpp | ./hash.sh
//79534f
/**
 * Can't handle updates; works with negatives.
 * @see https://cp-algorithms.com/data_structures/segment_tree.html
 * #preserving-the-history-of-its-values-persistent-segment-tree
 */
struct distinct_query {
    struct node {
        int sum;
        int lch, rch; //children, indexes into `tree`
        node(int a_sum, int a_lch, int a_rch) : sum(a_sum), lch(a_lch), rch(a_rch) {}
    };
    const int N;
    vector<int> roots;
    deque<node> tree;
    /**
     * @time O(n log n)
     * @memory O(n log n)
     */
    distinct_query(const vector<int>& arr) : N(ssize(arr)), roots(N + 1, 0) {
        tree.emplace_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.emplace_back(tree[v].sum + 1, 0, 0);
            return ssize(tree) - 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.emplace_back(tree[lch].sum + tree[rch].sum, lch, rch);
return ssize(tree) - 1;
}
/**
 * Returns number of distinct elements in range [le, ri).
 * query(i, i) returns 0.
 * @time O(log n)
 */
int query(int le, int ri) const {
    assert(0 <= le && le <= ri && ri <= N);
    return query(roots[le], roots[ri], 0, N, le + 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);
}
};
```

Implicit Lazy Segment Tree

```
//cat implicit_seg_tree.hpp | ./hash.sh
//ab5eb9
//example initialization:
// implicit_seg_tree<10'000'000> ist(le, ri);
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& le, const dt& ri) {
        if (le[0] == ri[0]) return {le[0], le[1] + ri[1]};
        return min(le, ri);
    }
    static constexpr dt UNIT{{(long long)1e18, 0LL}};
    struct node {
        dt val;
        ch lazy = 0;
        int lch = -1, rch = -1; // children, indexes into `tree`, -1 for null
    } tree[N];
    int ptr = 0, root_l, root_r; // [root_l, root_r) defines range of root node; handles
        ⇨ negatives
    implicit_seg_tree(int le, int ri) : root_l(le), root_r(ri) {
        tree[ptr++].val = {0, ri - le};
    }
    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++].val = {0, tm - tl};
tree[v].rch = ptr;
tree[ptr++].val = {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 [le,ri)
void update(int le, int ri, ch add) {
    update(0, root_l, root_r, le, ri, add);
}
void update(int v, int tl, int tr, int le, int ri, ch add) {
    if (ri <= tl || tr <= le)
        return;
    if (le <= tl && tr <= ri)
        return apply(v, add);
    push(v, tl, tr);
    int tm = tl + (tr - tl) / 2;
    update(tree[v].lch, tl, tm, le, ri, add);
    update(tree[v].rch, tm, tr, le, ri, add);
    tree[v].val = combine(tree[tree[v].lch].val,
        tree[tree[v].rch].val);
}
//query range [le,ri)
dt query(int le, int ri) {
    return query(0, root_l, root_r, le, ri);
}
dt query(int v, int tl, int tr, int le, int ri) {
    if (ri <= tl || tr <= le)
        return UNIT;
    if (le <= tl && tr <= ri)
        return tree[v].val;
    push(v, tl, tr);
    int tm = tl + (tr - tl) / 2;
    return combine(query(tree[v].lch, tl, tm, le, ri),
        query(tree[v].rch, tm, tr, le, ri));
}
};
```

Kth Smallest

```
//cat kth_smallest.hpp | ./hash.sh
//1e720f
/**
 * Can't handle updates.
 * @see https://cp-algorithms.com/data_structures/segment_tree.html
 * preserving-the-history-of-its-values-persistent-segment-tree
 */
struct kth_smallest {
    struct node {
        int sum;
        int lch, rch; // children, indexes into `tree`
        node(int a_sum, int a_lch, int a_rch) : sum(a_sum), lch(a_lch), rch(a_rch) {}
    };
    int mn, mx;
```



```
vector<int> roots;
deque<node> tree;
/**
 * @time O(n log max)
 * @memory O(n log max)
 */
kth_smallest(const vector<int>& arr) : roots(ssize(arr) + 1, 0) {
    auto [mn_iter, mx_iter] = minmax_element(arr.begin(), arr.end());
    mn = *mn_iter, mx = *mx_iter + 1;
    tree.emplace_back(0, 0, 0); //acts as null
    for (int i = 0; i < ssize(arr); 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.emplace_back(tree[v].sum + 1, 0, 0);
        return ssize(tree) - 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.emplace_back(tree[lch].sum + tree[rch].sum, lch, rch);
    return ssize(tree) - 1;
}
/**
 * Returns (k+1)th smallest number in range [le, ri]. k is 0-based, so
 * query(le,ri,0) returns the min; query(le,ri,ri-le-1) returns the max.
 * @time O(log max)
 */
int query(int le, int ri, int k) const {
    assert(0 <= k && k < ri - le); //note this condition implies le < ri
    assert(0 <= le && ri < ssize(roots));
    return query(roots[le], roots[ri], 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
//823193
//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;
    merge_sort_tree(const vector<int>& arr) : N(ssize(arr)), S(N ? 1 << __lg(2 * N - 1) : 0),
        ⇨ tree(2 * N) {
        transform(arr.begin(), arr.end(), tree.begin() + N, [](int val) -> vector<int> {
            return {val};
        });
    }
};
```

```
});
rotate(tree.rbegin(), tree.rbegin() + S - N, tree.rbegin() + N);
for (int i = N - 1; i >= 1; i--) {
    const auto& le = tree[2 * i];
    const auto& ri = tree[2 * i + 1];
    tree[i].resize(ssize(le) + ssize(ri));
    merge(le.begin(), le.end(), ri.begin(), ri.end(), tree[i].begin());
}
}
int value(int v, int x) const {
    return int(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);
}
/**
 * Returns the number of values in range [le, ri) which are < x.
 * @time O(log^2(n))
 */
int query(int le, int ri, int x) const {
    int res = 0;
    for (le = to_leaf(le), ri = to_leaf(ri); le < ri; le >>= 1, ri >>= 1) {
        if (le & 1) res += value(le++, x);
        if (ri & 1) res += value(--ri, x);
    }
    return res;
}
};
```

BIT

```
//cat bit.hpp | ./hash.sh
//ab7995
//NOLINTNEXTLINE(readability-identifier-naming)
template <typename T> struct BIT {
    vector<T> bit;
    BIT(int n) : bit(n, 0) {}
    /**
     * @time O(n)
     */
    BIT(const vector<T>& a) : bit(a) {
        for (int i = 0; i < ssize(a); i++) {
            int j = i | (i + 1);
            if (j < ssize(a)) bit[j] += bit[i];
        }
    }
    /**
     * @time O(log n)
     */
    void update(int i, const T& d) {
        assert(0 <= i && i < ssize(bit));
        for (; i < ssize(bit); i |= i + 1) bit[i] += d;
    }
    /**
     * sum of range [0, ri)
     * @time O(log n)
     */
};
```

```
T sum(int ri) const {
    assert(0 <= ri && ri <= ssize(bit));
    T ret = 0;
    for (; ri > 0; ri &= ri - 1) ret += bit[ri - 1];
    return ret;
}
/**
 * sum of range [le, ri)
 * @time O(log n)
 */
T sum(int le, int ri) const {
    assert(0 <= le && le <= ri && ri <= ssize(bit));
    return sum(ri) - sum(le);
}
/**
 * Returns min pos such that sum of range [0, pos) >= sum. Returns
 * ssize(bit) + 1 if no sum is >= sum.
 * Doesn't work if BIT::sum(i, i + 1) < 0
 * @time O(log n)
 */
int lower_bound(T sum) const {
    if (sum <= 0) return 0;
    int pos = 0;
    for (int pw = 1 << __lg(ssize(bit) | 1); pw; pw >>= 1)
        if (pos + pw <= ssize(bit) && bit[pos + pw - 1] < sum)
            pos += pw, sum -= bit[pos - 1];
    return pos + 1;
}
};
```

RMQ

```
//cat rmq.hpp | ./hash.sh
//5adc3f
/**
 * @code{.cpp}
 * vector<long long> arr;
 * RMQ<long long> rmq(arr, [&](auto x, auto y) { return min(x, y); });
 *
 * //To get index of min element:
 * vector<pair<long long, int>> arr; //initialize arr[i].second = i
 * RMQ<pair<long long, int>> rmq(arr, [&](auto x, auto y) { return min(x, y); });
 * @endcode
 * @see https://github.com/kth-competitive-programming/
 * kactl/blob/main/content/data-structures/RMQ.h
 */
//NOLINTNEXTLINE(readability-identifier-naming)
template <typename T> struct RMQ {
    vector<vector<T>> dp;
    function<T(const T&, const T&)> op;
    /**
     * @time O(n log n)
     * @memory O(n log n)
     */
    RMQ(const vector<T>& arr, const function<T(const T&, const T&)>& a_op) : dp(1, arr),
        op(a_op) {
        for (int i = 0; (2 << i) <= ssize(arr); i++) {
            dp.emplace_back(ssize(arr) - (2 << i) + 1);
```

```
                transform(dp[i].begin() + (1 << i), dp[i].end(), dp[i].begin(), dp[i + 1].begin(),
                    op);
        }
    }
    /**
     * range [le, ri)
     * @time O(1)
     */
    T query(int le, int ri) const {
        assert(0 <= le && le < ri && ri <= ssize(dp[0]));
        int lg = __lg(ri - le);
        return op(dp[lg][le], dp[lg][ri - (1 << lg)]);
    }
};
```

Disjoint RMQ

```
//cat disjoint_rmq.hpp | ./hash.sh
//8e3959
/**
 * Disjoint RMQ is like normal RMQ except these ranges never overlap. It is
 * useful for:
 * - min and # of mins.
 * - product under composite mod
 * - matrix multiply
 * - function composition
 * @code{.cpp}
 * //usage for min and # of mins:
 * vector<pair<long long, int>> arr; //initialize arr[i].second = 1
 * disjoint_rmq<pair<long long, int>> rmq(arr, [&](auto x, auto y) {
 *     if (x.first == y.first) return make_pair(x.first, x.second + y.second);
 *     return min(x, y);
 * });
 * @endcode
 * @see https://codeforces.com/blog/entry/87940
 */
template <typename T> struct disjoint_rmq {
    const int N;
    vector<vector<T>> dp;
    function<T(const T&, const T&)> op; // any associative operation
    /**
     * @time O(n log n)
     * @memory O(n log n)
     */
    disjoint_rmq(const vector<T>& arr, const function<T(const T&, const T&)>& a_op) :
        N(ssize(arr)), op(a_op) {
        for (int len = 1; len <= N; len *= 2) {
            dp.emplace_back(N);
            for (int le = 0; le < N; le += 2 * len) {
                int mi = min(N, le + len), ri = min(N, le + 2 * len);
                partial_sum(arr.rend() - mi, arr.rend() - le, dp.back().rend() - mi, [&](const
                    T & x, const T & y) {return op(y, x);});
                partial_sum(arr.begin() + mi, arr.begin() + ri, dp.back().begin() + mi, op);
            }
        }
    }
    /**
     * range [le, ri)
```

```
* @time 0(1)
*/
T query(int le, int ri) const {
    assert(0 <= le && le < ri && ri <= N);
    if (ri - le == 1) return dp[0][le];
    int lg = __lg(le ^ (ri - 1));
    return op(dp[lg][le], dp[lg][ri - 1]);
}
};
```

Lazy Segment Tree

```
//cat lazy_segment_tree.hpp | ./hash.sh
//96535f
/**
 * Internal nodes are [1, n), leaf nodes are [n, 2 * n).
 * Rotating leaves makes it a single complete binary tree (instead of a set of
 * perfect binary trees). So now, even for non-power of 2 size:
 * - recursive seg tree works
 * - recursive tree walks AKA binary search works
 * - root is at tree[1]
 * @see 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& le, const dt& ri) {
        return min(le, ri);
    }
    static const dt UNIT = 1e18;
    struct node {
        dt val;
        ch lazy;
        int le, ri; // [le, ri)
    };
    const int N, S/*smallest power of 2 >= N*/;
    vector<node> tree;
    seg_tree(const vector<dt>& arr) : N(ssize(arr)), 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].le,
                tree[2 * i + 1].ri
            };
        }
    }
    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 [le, ri)
    void update(int le, int ri, ch change) {
        assert(0 <= le && le <= ri && ri <= N);
        le = to_leaf(le), ri = to_leaf(ri);
        int lca_l_r = __lg((le - 1) ^ ri);
        for (int lg = __lg(le); lg > __builtin_ctz(le); lg--) push(le >> lg);
        for (int lg = lca_l_r; lg > __builtin_ctz(ri); lg--) push(ri >> lg);
        for (int x = le, y = ri; x < y; x >>= 1, y >>= 1) {
            if (x & 1) apply(x++, change);
            if (y & 1) apply(--y, change);
        }
        for (int lg = __builtin_ctz(ri) + 1; lg <= lca_l_r; lg++) build(ri >> lg);
        for (int lg = __builtin_ctz(le) + 1; lg <= __lg(le); lg++) build(le >> lg);
    }
    void update(int v/* = 1*/, int le, int ri, ch change) {
        if (ri <= tree[v].le || tree[v].ri <= le) return;
        if (le <= tree[v].le && tree[v].ri <= ri) return apply(v, change);
        push(v);
        update(2 * v, le, ri, change);
        update(2 * v + 1, le, ri, change);
        build(v);
    }
    //query range [le, ri)
    dt query(int le, int ri) {
        assert(0 <= le && le <= ri && ri <= N);
        le = to_leaf(le), ri = to_leaf(ri);
        int lca_l_r = __lg((le - 1) ^ ri);
        for (int lg = __lg(le); lg > __builtin_ctz(le); lg--) push(le >> lg);
        for (int lg = lca_l_r; lg > __builtin_ctz(ri); lg--) push(ri >> lg);
        dt resl = UNIT, resr = UNIT;
        for (; le < ri; le >>= 1, ri >>= 1) {
            if (le & 1) resl = combine(resl, tree[le++].val);
            if (ri & 1) resr = combine(tree[--ri].val, resr);
        }
        return combine(resl, resr);
    }
    dt query(int v/* = 1*/, int le, int ri) {
        if (ri <= tree[v].le || tree[v].ri <= le) return UNIT;
        if (le <= tree[v].le && tree[v].ri <= ri) return tree[v].val;
        push(v);
        return combine(query(2 * v, le, ri), query(2 * v + 1, le, ri));
    }
};
```

STRINGS

Binary Trie

```
//cat binary_trie.hpp | ./hash.sh
//88fa9c
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) {}
    /**
     * Pass delta = 1 to insert val, -1 to remove val, 0 to get the # of val's
     * in this data structure.
     * @time O(MX_BIT)
     */
    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] = ssize(t);
                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, and the value of
     * (x^val) is minimum.
     * @time O(MX_BIT)
     */
    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;
    }
};
```

Prefix Function

```
//cat prefix_function.hpp | ./hash.sh
```

```
//65fea7
/**
 * @see https://cp-algorithms.com/string/prefix-function.html#implementation
 * @time O(n)
 */
template <typename T> vector<int> prefix_function(const T& s) {
    vector<int> pi(ssize(s), 0);
    for (int i = 1; i < ssize(s); i++) {
        int j = pi[i - 1];
        while (j > 0 && s[i] != s[j]) j = pi[j - 1];
        pi[i] = j + (s[i] == s[j]);
    }
    return pi;
}
```

KMP String Matching

```
//cat kmp.hpp | ./hash.sh
//1edfb2
#include "prefix_function.hpp"
/**
 * @code{.cpp}
 *     string s;
 *     KMP kmp(s);
 *     vector<int> a;
 *     KMP kmp(a);
 * @endcode
 * 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 <typename T> struct KMP {
    T needle;
    vector<int> pi;
    /**
     * @time O(|needle|)
     * @memory O(|needle|)
     */
    KMP(const T& a_needle) : needle(a_needle), pi(prefix_function(needle)) {}
    /**
     * Returns array `matches` where:
     * haystack.substr(matches[i], ssize(needle)) == needle
     * @time O(|needle| + |haystack|)
     */
    vector<int> find(const T& haystack) const {
        vector<int> matches;
        for (int i = 0, j = 0; i < ssize(haystack); i++) {
            while (j > 0 && needle[j] != haystack[i]) j = pi[j - 1];
            if (needle[j] == haystack[i]) j++;
            if (j == ssize(needle)) {
                matches.push_back(i - ssize(needle) + 1);
                j = pi[j - 1];
            }
        }
        return matches;
    }
};
```

```
};
```

Suffix and LCP Arrays

```
//cat suffix_array.hpp | ./hash.sh
//3fef7e
/**
 * suffixes of "banana":
 * 0 banana
 * 1 anana
 * 2 nana
 * 3 ana
 * 4 na
 * 5 a
 * sorted, lcp
 * 5 a
 * | 1
 * 3 ana
 * ||| 3
 * 1 anana
 * 0
 * 0 banana
 * 0
 * 4 na
 * || 2
 * 2 nana
 * suffix array = [5, 3, 1, 0, 4, 2]
 * rank array = [3, 2, 5, 1, 4, 0] (sa[rank[i]] == i, rank[sa[i]] == i)
 * lcp array = [1, 3, 0, 0, 2]
 * @code{.cpp}
 * string s;
 * suffix_array info(s, 128);
 * vector<int> arr;
 * suffix_array info(arr, 1e5 + 1);
 * @endcode
 * @see https://github.com/kth-competitive-programming/kactl
 * /blob/main/content/strings/SuffixArray.h
 */
template <typename T> struct suffix_array {
    const int N;
    vector<int> sa, rank, lcp;
    /**
     * Assumes 0 <= s[i] < max_val.
     * @time O((nlogn) + max_val)
     * @memory O(n + max_val)
     */
    suffix_array(const T& s, int max_val) : N(ssize(s)), sa(N), rank(s.begin(), s.end()),
        lcp(max(0, N - 1)) {
        iota(sa.begin(), sa.end(), 0);
        vector<int> tmp(N);
        for (int len = 0; len < N; len = max(1, 2 * len)) { //suffix array
            iota(tmp.begin(), tmp.begin() + len, N - len);
            copy_if(sa.begin(), sa.end(), tmp.begin() + len, [&](int& val) {
                val -= len;
                return val >= 0;
            });
            vector<int> freq(max_val, 0);
            for (int val : rank) freq[val]++;
        }
    }
};
```

```
partial_sum(freq.begin(), freq.end(), freq.begin());
for_each(tmp.rbegin(), tmp.rend(), [&](int t) {
    sa[--freq[rank[t]]] = t;
});
swap(rank, tmp);
max_val = 1, rank[sa[0]] = 0;
auto prev_rank = [&](int i) {return pair(tmp[i], i + len < N ? tmp[i + len] :
    -1);};
for (int i = 1; i < N; i++) {
    max_val += prev_rank(sa[i - 1]) != prev_rank(sa[i]);
    rank[sa[i]] = max_val - 1;
}
if (max_val == N) break;
}
for (int i = 0, k = 0; i < N; i++) { //lcp array
    if (k > 0) k--;
    if (rank[i] == 0) continue;
    int j = sa[rank[i] - 1];
    while (max(i, j) + k < N && s[i + k] == s[j + k]) k++;
    lcp[rank[i] - 1] = k;
}
}
};
```

Suffix Array Related Queries

```
//cat suffix_array_query.hpp | ./hash.sh
//ae4931
#include "suffix_array.hpp"
#include "../range_data_structures/rmq.hpp"
/**
 * Various queries you can do based on suffix array.
 * @see https://github.com/yosupo06/Algorithm/blob
 * /master/src/string/suffixarray.hpp
 */
template <typename T> struct sa_query {
    T s;
    suffix_array<T> info;
    RMQ<int> rmq_lcp, rmq_sa;
    /**
     * Assumes 0 <= s[i] < max_val.
     * @time O((nlogn) + max_val)
     * @memory O((nlogn) + max_val) An O(max_val) size freq array is used
     * temporarily during suffix array construction. Only O(n log n) memory is
     * permanently stored by this struct.
     */
    sa_query(const T& a_s, int max_val) :
        s(a_s),
        info(suffix_array(s, max_val)),
        rmq_lcp(info.lcp, [](int i, int j) -> int {return min(i, j);}),
        rmq_sa(info.sa, [](int i, int j) -> int {return min(i, j);}) {}
    /**
     * Returns length of longest common prefix of suffixes s[idx1...N),
     * s[idx2...N), 0-based indexing.
     *
     * To check if two substrings s[l1..r1), s[l2..r2) are equal:
     * r1-l1 == r2-l2 && get_lcp(l1, l2) >= r1-l1
     * @time O(1)
     */
};
```

```

    */
    int get_lcp(int idx1, int idx2) const {
        if (idx1 == idx2) return ssize(s) - idx1;
        auto [le, ri] = minmax(info.rank[idx1], info.rank[idx2]);
        return rmq_lcp.query(le, ri);
    }
    /**
     * Returns 1 if suffix s[idx1 ... N] < s[idx2 ... N].
     * Returns 0 if idx1 == idx2.
     * @time O(1)
     */
    bool less(int idx1, int idx2) const {
        return info.rank[idx1] < info.rank[idx2];
    }
    /**
     * Returns range [le, ri) such that:
     * - for all i in [le, ri): t == s.substr(info.sa[i], ssize(t))
     * - `ri - le` is the # of matches of t in s.
     * @time O(|t| * log(|s|))
     */
    pair<int, int> find(const T& t) const {
        auto cmp = [&](int i, int cmp_val) -> bool {
            return s.compare(i, ssize(t), t) < cmp_val;
        };
        auto le = lower_bound(info.sa.begin(), info.sa.end(), 0, cmp);
        auto ri = lower_bound(le, info.sa.end(), 1, cmp);
        return {le - info.sa.begin(), ri - info.sa.begin()};
    }
    /**
     * Returns min i such that t == s.substr(i, ssize(t)) or -1. For example,
     * replace RMQ with kth-smallest PST/Wavelet to solve
     * https://open.kattis.com/problems/anothersubstringqueryproblem
     * @time O(|t| * log(|s|))
     */
    int find_first(const T& t) const {
        auto [le, ri] = find(t);
        if (le == ri) return -1;
        return rmq_sa.query(le, ri);
    }
};
```

Palindrome Query

```

//cat palindrome_query.hpp | ./hash.sh
//68c8e1
#include "../kactl/content/strings/Manacher.h"
struct pal_query {
    const int N;
    array<vi, 2> pal_len;
    /**
     * @time O(n)
     * @memory O(n)
     */
    pal_query(const string& s) : N(ssize(s)), pal_len(manacher(s)) {}
    /**
     * Returns 1 if substring s[le...ri] is a palindrome.
     * Returns 1 when le == ri.
     * @time O(1)
     */
};
```

```

    */
    bool is_pal(int le, int ri) const {
        assert(0 <= le && le <= ri && ri <= N);
        int len = ri - le;
        return pal_len[len & 1][le + len / 2] >= len / 2;
    }
};
```

Trie

```

//cat trie.hpp | ./hash.sh
//2aa8c6
//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 insert(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] = ssize(t);
                t.emplace_back(v, ch);
            }
            v = t[v].next[let];
        }
        t[v].cnt_words++;
    }
    int find(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;
    }
};
```