

Contents			
1 Misc			
1.1 Macros and debug	2	3.4.2 Kosaraju	13
2 Data Structures	3	4 Math	14
2.1 GNU PBDS	3	4.1 SOS DP	14
2.2 2D Partial Sums	3	4.2 Matrix	14
2.3 Sparse Table	4	4.3 Number Theory	15
2.4 Fenwick Tree	5	4.3.1 Miller-Rabin	15
2.5 Longest Increasing Subsequence	5	4.3.2 Linear Sieve	16
2.6 Xobriest	6	4.3.3 Get Factors	17
2.7 Lazy	6	4.3.4 Extended GCD	17
3 Graph	8	4.3.5 Chinese Remainder Theorem	17
3.1 General	8	4.3.6 Pollard Rho	18
3.1.1 Bellman	8	4.3.7 De Bruijn Sequence	18
3.1.2 Floyd	8	4.3.8 Combinatorics	18
3.1.3 DSU	9	5 Geometry	20
3.1.4 Binary Lifting	9		
3.2 Kuhn-Munkres algorithm	10	6 Strings	21
3.3 Strongly Connected Components	12	6.1 Knuth-Morris-Pratt Algorithm	21
3.4 Biconnected Components	12	6.2 Z Value	21
3.4.1 Articulation Points	12	6.3 Manachers Algorithm	21
		6.4 Trie	22
		6.5 Aho-Corasick Automaton	23
		6.6 Hashing	25

1.	Misc	23	- Did you forget to sort or unique? - Generate large and worst "corner" cases. - Check your 'm' / 'n', 'i' / 'j' and 'x' / 'y'. - Are everything initialized or reset properly? - Are you sure about the STL thing you are using? - Read cppreference (should be available). - Print everything and run it on pen and paper.
1	1.1. Macros and debug	25	
1	#pragma GCC optimize("O3", "unroll-loops")	27	
1	- Pre-submit: - Did you make a typo when copying a template? - Test more cases if unsure. - Write a naive solution and check small cases. - Submit the correct file.	29	
3		31	- Time Limit Exceeded: - Calculate your time complexity again.
5		33	- Does the program actually end? - Check for `while(q.size())` etc.
7	- General Debugging: - Read the whole problem again. - Have a teammate read the problem. - Have a teammate read your code. - Explain your solution to them (or a rubber duck).	35	- Test the largest cases locally.
9		37	- Did you do unnecessary stuff? - e.g. pass vectors by value
11		39	- e.g. `memset` for every test case
13	- Print the code and its output / debug output. - Go to the toilet.	41	- Is your constant factor reasonable?
15	- Wrong Answer: - Any possible overflows? - > `__int128` ? - Try `-ftrapv` or `#pragma GCC optimize("trapv")`	43	- Runtime Error: - Check memory usage.
17		45	- Forget to clear or destroy stuff? - > `vector::shrink_to_fit()`
19	- Floating point errors? - > `long double` ?	45	- Stack overflow?
21	- turn off math optimizations - check for `==`, `>=`, `acos(1.000000001)`, etc.		- Bad pointer / array access? - Division by zero? NaN's?

2. Data Structures

2.1. GNU PBDS

```

1 // #include <ext/pb_ds/assoc_container.hpp>
2 // #include <ext/pb_ds/tree_policy.hpp>
3 using namespace __gnu_pbds;
4 template <typename T>
5 using ordered_set =
6     tree<T, null_type, std::less<T>, rb_tree_tag,
7         tree_order_statistics_node_update>;
8 ordered_set<int> os;
9 os.find_by_order(k); // iterator to k-th element (0-indexed)
10 os.order_of_key(x); // number of elements < x
11 template <typename T>
12 using ordered_multiset_base =
13     tree<pair<T, int>, null_type, less<pair<T, int>>,
14         rb_tree_tag, tree_order_statistics_node_update>;
15 template <typename T> struct ordered_multiset {
16     ordered_multiset_base<T> os;
17     int timer = 0;
18     void insert(T x) { os.insert({x, timer++}); }
19     void erase_one(T x) {
20         auto it = os.lower_bound({x, -1});
21         if (it != os.end() && it->first == x) os.erase(it);
22     }
23     int order_of_key(T x) { return os.order_of_key({x, -1}); }
24     T find_by_order(int k) {
25         return os.find_by_order(k)->first;
26     }
27     int size() { return (int)os.size(); }
28 };

```

```

25     return os.find_by_order(k)->first;
26 }
27 int size() { return (int)os.size(); }
28 };

```

2.2. 2D Partial Sums

```

1 struct PrefixSum2D {
2     vvl pref;           // 0-based 2-D prefix sum
3     void build(const vvl &v) { // creates a copy
4         int n = v.size(), m = v[0].size();
5         pref.assign(n, vll(m, 0));
6         for (int i = 0; i < n; i++)
7             for (int j = 0; j < m; j++)
8                 pref[i][j] = v[i][j] + (i ? pref[i - 1][j] : 0) +
9                     (j ? pref[i][j - 1] : 0) -
10                     (i && j ? pref[i - 1][j - 1] : 0);
11    }
12    ll query(int ulx, int uly, int brx, int bry) const {
13        ll ans = pref[brx][bry];
14        if (ulx) ans -= pref[ulx - 1][bry];
15        if (uly) ans -= pref[brx][uly - 1];
16        if (ulx && uly) ans += pref[ulx - 1][uly - 1];
17        return ans;
18    }
19    ll query(int ulx, int uly, int size) const {
20        return query(ulx, uly, ulx + size - 1, uly + size - 1);
21    }

```

```

};  

23 struct PartialSum2D : PrefixSum2D {  

    vll diff; // 0 based  

25     int n, m;  

    PartialSum2D(int _n, int _m) : n(_n), m(_m) {  

        diff.assign(n + 1, vll(m + 1, 0));  

    }  

29     // add c from [ulx,uly] to [brx,bry]  

    void update(int ulx, int uly, int brx, int bry, ll c) {  

31         diff[ulx][uly] += c;  

        diff[ulx][bry + 1] -= c;  

33         diff[brx + 1][uly] -= c;  

        diff[brx + 1][bry + 1] += c;  

35     }  

    void update(int ulx, int uly, int size, ll c) {  

37         int brx = ulx + size - 1, bry = uly + size - 1;  

        update(ulx, uly, brx, bry, c);  

39     }  

    void process() {  

41         this->build(diff);  

        } // process grid using prefix sum  

43 };  

// usage  

45 PrefixSum2D pref;  

pref.build(v); // takes 2d 0-based vector as input  

47 pref.query(x1, y1, x2, y2); // sum of region  

PartialSum2D part(n, m); // dimension of grid 0 based

```

```

49 part.update(x1, y1, x2, y2, 1); // add 1 in region  

    // must run after all updates  

51 part.process(); // prefix sum on diff array  

    // only exists after processing  

53 vll &grid = part.pref; // processed diff array  

part.query(x1, y1, x2, y2); // gives sum of region

```

2.3. Sparse Table

```

1 struct SparseTable {  

    vector<vector<ll>> sparse;  

3     vector<int> Log;  

    int n, max_log;  

5     ll IDENTITY_VAL; // e.g., 0 for Sum, 1 for Product,  

    ll func(ll a, ll b) { return (a + b); } // Example: Sum  

7     void build(const vector<ll> &a, ll identity) {  

        n = a.size();  

9         IDENTITY_VAL = identity;  

        max_log = 32 - __builtin_clz(n);  

11         sparse.assign(n, vector<ll>(max_log));  

        Log.assign(n + 1, 0);  

13         for (int i = 2; i <= n; ++i) Log[i] = Log[i / 2] + 1;  

15         for (int i = 0; i < n; ++i) sparse[i][0] = a[i];  

for (int j = 1; (1 << j) <= n; ++j) {  

17             for (int i = 0; i + (1 << j) <= n; ++i)  

                sparse[i][j] =  

                    func(sparse[i][j - 1],  

                        sparse[i + (1 << (j - 1))][j - 1]);
19

```

```

21    }
22 }
23 ll query_idempotent(int l, int r) {
24     int k = Log[r - l + 1];
25     return func(sparse[l][k], sparse[r - (1 << k) + 1][k]);
26 }
27 // Use for: Sum, Product, XOR, Matrix Multiplication
28 ll query_non_idempotent(int l, int r) {
29     ll res = IDENTITY_VAL;
30     for (int j = max_log - 1; j >= 0; --j)
31         if ((1 << j) <= r - l + 1) {
32             res = func(res, sparse[l][j]);
33             l += (1 << j); // Move L forward by 2^j
34         }
35     return res;
36 }
37 
```

```
9 |     for (size_t i = 0; i < a.size(); i++) add(i, a[i]);
| }
11 | long long sum(int idx) {
12 |     long long ret = 0;
13 |     for (++idx; idx > 0; idx -= idx & -idx) ret += bit[idx];
14 |     return ret;
15 | }
16 | long long sum(int l, int r) {
17 |     if (l > r) return 0; // Guard clause
18 |     return sum(r) - sum(l - 1);
19 | }
20 | void add(int idx, int delta) {
21 |     for (++idx; idx < n; idx += idx & -idx)
22 |         bit[idx] += delta;
23 | }
};
```

2.4. Fenwick Tree

```
1 struct fwt {
2     vector<long long> bit;
3     int n;
4     fwt(int n) {
5         this->n = n + 1;
6         bit.assign(n + 1, 0);
7     }
8     fwt(const vector<int> &a) : fwt(a.size()) {
```

2.5. Longest Increasing Subsequence

```
1 int lis(vector<int> const &a) {
2     int n = a.size();
3     const int INF = 1e9;
4     vector<int> d(n + 1, INF); // min possible ending value
5     // of inc subseq of length l, that we have seen
6     d[0] = -INF;
7     // user lower bound for non-decreasing
8     for (int i = 0; i < n; i++) {
9         int l =
```

```

11    upper_bound(d.begin(), d.end(), a[i]) - d.begin());
12    if (d[l - 1] < a[i] && a[i] < d[l]) d[l] = a[i];
13 }
14 int ans = 0;
15 for (int l = 0; l <= n; l++) {
16     if (d[l] < INF) ans = l;
17 }
18 return ans;
19 }
```

2.6. Xobriest

```

1 static uint64_t seed =
2 chrono::steady_clock::now().time_since_epoch().count();
3 uint64_t splitmix64() {
4     seed += 0x9e3779b97f4a7c15;
5     uint64_t x = seed;
6     x = (x ^ (x >> 30)) * 0xbff58476d1ce4e5b9;
7     x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
8     return x ^ (x >> 31);
9 }
10 long long randInRange(long long l, long long r) {
11     assert(l <= r);
12     return l + splitmix64() % (r - l + 1);
13 }
14 void solve() {
15     map<ll, pll> mp;
16     vector<pll> pref;
```

```

17 for (int i = 1; i <= n; i++) {
18     mp[a[i]] = {randInRange(0LL, (1LL << 64) - 1),
19                 randInRange(0LL, (1LL << 64) - 1)};
20     pref[i].ff = pref[i - 1].ff ^ mp[a[i]].ff;
21     pref[i].ss = pref[i - 1].ss ^ mp[a[i]].ss;
22 }
23 }
```

2.7. Lazy

```

1 template <class Node, class Update> struct LazySegTree {
2     int n;
3     vector<Node> st;
4     vector<Update> lz;
5     LazySegTree(int n, vector<ll> &a) : n(n) {
6         st.assign(4 * n, Node());
7         lz.assign(4 * n, Update());
8         build(1, 0, n - 1, a);
9     }
10    void build(int p, int l, int r, vector<ll> &a) {
11        if (l == r) return void(st[p] = Node(a[l]));
12        int m = (l + r) / 2;
13        build(p << 1, l, m, a);
14        build(p << 1 | 1, m + 1, r, a);
15        st[p].merge(st[p << 1], st[p << 1 | 1]);
16    }
17    void push(int p, int l, int r) {
18        if (lz[p].is_identity()) return;
```

```

19 int m = (l + r) / 2;
apply(p << 1, l, m, lz[p]);
apply(p << 1 | 1, m + 1, r, lz[p]);
lz[p] = Update();
}
23
void apply(int p, int l, int r, Update &u) {
    u.apply(st[p], l, r);
    if (l != r) lz[p].combine(u);
}
27
void update(int p, int l, int r, int i, int j,
            Update &u) {
    if (r < i || l > j) return;
    if (i <= l && r <= j) return apply(p, l, r, u);
    push(p, l, r);
    int m = (l + r) / 2;
    update(p << 1, l, m, i, j, u);
    update(p << 1 | 1, m + 1, r, i, j, u);
    st[p].merge(st[p << 1], st[p << 1 | 1]);
}
37
Node query(int p, int l, int r, int i, int j) {
    if (r < i || l > j) return Node();
    if (i <= l && r <= j) return st[p];
    push(p, l, r);
    int m = (l + r) / 2;
    Node res;
    res.merge(query(p << 1, l, m, i, j),
              query(p << 1 | 1, m + 1, r, i, j));
}
45
47
return res;
}
49 void update(int l, int r, ll v) {
    Update u(v);
    update(1, 0, n - 1, l, r, u);
}
51
Node query(int l, int r) {
    return query(1, 0, n - 1, l, r);
}
53
55
struct Node {
    ll val = 0;
    Node(ll v = 0) : val(v) {}
    void merge(const Node &l, const Node &r) {
        val = l.val + r.val;
    }
}
57
61
63
struct Update {
    ll val = 0;
    Update(ll v = 0) : val(v) {}
    bool is_identity() const { return val == 0; }
    void apply(Node &a, int l, int r) {
        a.val = val * (r - l + 1);
    }
    void combine(Update &u) { val = u.val; }
}
65
67
69
71

```

3. Graph

3.1. General

3.1.1. Bellman

```

1 const long long INF = 1e18;
vector<array<int, 3>> e; // {u, v, w}
3 vector<long long> d, par;
5 int main() {
6     int n, m;
7     cin >> n >> m;
8     e.resize(m);
9     d.assign(
10        n, 0); // use 0 to detect neg-cycle in disconnected graph
11    par.assign(n, -1);
12    int x = -1;
13    for (int i = 0; i <= n; i++) {
14        x = -1;
15        for (auto [u, v, w] : e) {
16            if (d[v] > d[u] + w) {
17                d[v] = d[u] + w;
18                par[v] = u;
19                x = v;
20            }
21        }
22        if (x == -1) {
23            // no negative cycle

```

```

25    } else {
26        // negative cycle exists
27        for (int i = 0; i < n; i++) x = par[x];
28        vector<int> cyc;
29        for (int v = x;; v = par[v]) {
30            cyc.push_back(v);
31            if (v == x && cyc.size() > 1) break;
32        }
33        reverse(cyc.begin(), cyc.end());
34    }
35

```

3.1.2. Floyd

```

1 const int INF = 1e9;
vector<vector<int>> d, p;
3 void path(int i, int j) {
4     if (i != j) path(i, p[i][j]);
5     cout << j << " ";
6 }
7 int main() {
8     int n, m;
9     cin >> n >> m;
10    d.assign(n, vector<int>(n, INF));
11    p.assign(n, vector<int>(n));
12    for (int i = 0; i < n; i++) {
13        d[i][i] = 0;
14        for (int j = 0; j < n; j++) p[i][j] = i;
15    }
16

```

```

15  }
16  while (m--) {
17    int u, v, w;
18    cin >> u >> v >> w;
19    --u;
20    --v;
21    d[u][v] = min(d[u][v], w);
22  }
23  for (int k = 0; k < n; k++)
24    for (int i = 0; i < n; i++)
25      for (int j = 0; j < n; j++)
26        if (d[i][k] < INF && d[k][j] < INF &&
27            d[i][j] > d[i][k] + d[k][j]) {
28          d[i][j] = d[i][k] + d[k][j];
29          p[i][j] = p[k][j];
30        }
31  for (int i = 0; i < n; i++)
32    if (d[i][i] < 0) {
33      // negative cycle exists
34    }
35  }

```

3.1.3. DSU

```

1 struct DSU {
2   vector<int> p, sz;
3   DSU(int n = 0) { init(n); }
4   void init(int n) {

```

```

5   p.resize(n);
6   sz.assign(n, 1);
7   iota(p.begin(), p.end(), 0);
8 }
9 int find(int x) {
10   if (p[x] == x) return x;
11   return p[x] = find(p[x]); // path compression
12 }
13 bool unite(int a, int b) {
14   a = find(a);
15   b = find(b);
16   if (a == b) return false;
17   if (sz[a] < sz[b]) swap(a, b);
18   p[b] = a;
19   sz[a] += sz[b];
20   return true;
21 }
22 bool same(int a, int b) { return find(a) == find(b); }
23 int size(int x) { return sz[find(x)]; }
24

```

3.1.4. Binary Lifting

```

1 int n;
2 vvi jump, g, adj;
3 vi depth, val;
4 void dfs(int root, int parent, int currDepth) {
5   depth[root] = parent == -1 ? 0 : depth[parent] + 1;

```

```

7   jump[root][0] = parent;
g[root][0] = val[root];
for (int i = 1; i < 20;
9     i++) { // always start from 1, since i depends on i-1
10    if (jump[root][i - 1] != -1) {
11      jump[root][i] = jump[jump[root][i - 1]][i - 1];
12      g[root][i] =
13        gcd(g[root][i - 1],
14            g[jump[root][i - 1]][i - 1]); // combination logic
15    } else {
16      jump[root][i] = -1;
17      g[root][i] =
18      g[root]
19      [i - 1]; // when the i-1th ancestor doesn't exist,
// then store same aggregate as i-1th
20    }
21  }
22 // do normal DFS from here
23 }
24 int path_gcd(int u, int v) {
25   if (depth[u] < depth[v]) swap(u, v);
26   int GCD = 0; // always use the identity value
27   for (int i = 19; i >= 0; i--) {
28     if (((depth[u] - depth[v]) >> i) & 1) {
29       {
30         GCD = gcd(g[u][i], GCD);
31         u = jump[u][i];
32       }
33     }
34   }
35 }
36 if (u == v)
37   return gcd(
38   GCD,
39   val[u]); // g[u][i], doesn't contain val at jump[u][i]
40   for (int i = 19; i >= 0; i--) {
41     if (jump[u][i] != jump[v][i]) {
42       GCD = gcd(GCD, g[u][i]);
43       GCD = gcd(GCD, g[v][i]);
44       u = jump[u][i];
45       v = jump[v][i];
46     }
47   }
48 // since both u and v are immediate child of lca
49 // neither u, v nor lca is included in the computation
50 // add them explicitly
51   GCD = gcd(GCD, val[u]);
52   GCD = gcd(GCD, val[v]);
53   return gcd(GCD, val[jump[u][0]]);
54 }
```

3.2. Kuhn-Munkres algorithm

```

1 // Maximum Weight Perfect Bipartite Matching
2 // Detect non-perfect-matching: // 1. set all edge[i][j] as
3 // INF
```

```

// 2. if solve() >= INF, it is not perfect matching.
5 struct KM {
    static const int MAXN = 1050;
7     static const ll INF = 1LL << 60;
    int n, match[MAXN], vx[MAXN], vy[MAXN];
9     ll edge[MAXN][MAXN], lx[MAXN], ly[MAXN], slack[MAXN];
    void init(int _n) {
11        n = _n;
12        for (int i = 0; i < n; i++)
13            for (int j = 0; j < n; j++) edge[i][j] = 0;
14    }
15    void add_edge(int x, int y, ll w) { edge[x][y] = w; }
16    bool DFS(int x) {
17        vx[x] = 1;
18        for (int y = 0; y < n; y++) {
19            if (vy[y]) continue;
20            if (lx[x] + ly[y] > edge[x][y]) {
21                slack[y] =
22                    min(slack[y], lx[x] + ly[y] - edge[x][y]);
23            } else {
24                vy[y] = 1;
25                if (match[y] == -1 || DFS(match[y])) {
26                    match[y] = x;
27                    return true;
28                }
29            }
}
31    return false;
}
32    ll solve() {
33        fill(match, match + n, -1);
34        fill(lx, lx + n, -INF);
35        fill(ly, ly + n, 0);
36        for (int i = 0; i < n; i++)
37            for (int j = 0; j < n; j++)
38                lx[i] = max(lx[i], edge[i][j]);
39        for (int i = 0; i < n; i++) {
40            fill(slack, slack + n, INF);
41            while (true) {
42                fill(vx, vx + n, 0);
43                fill(vy, vy + n, 0);
44                if (DFS(i)) break;
45                ll d = INF;
46                for (int j = 0; j < n; j++)
47                    if (!vy[j]) d = min(d, slack[j]);
48                for (int j = 0; j < n; j++) {
49                    if (vx[j]) lx[j] -= d;
50                    if (vy[j]) ly[j] += d;
51                    else slack[j] -= d;
52                }
53            }
54        }
55        ll res = 0;
56        for (int i = 0; i < n; i++) {
57
}

```

```

59     res += edge[match[i]][i];
}
return res;
61 }
} graph;

```

3.3. Strongly Connected Components

```

1 struct TarjanScc {
2     int n, step;
3     vector<int> time, low, instk, stk;
4     vector<vector<int>> e, scc;
5     TarjanScc(int n_) : n(n_), step(0), time(n), low(n), instk(n), e(n) {}
6     void add_edge(int u, int v) { e[u].push_back(v); }
7     void dfs(int x) {
8         time[x] = low[x] = ++step;
9         stk.push_back(x);
10        instk[x] = 1;
11        for (int y : e[x])
12            if (!time[y]) {
13                dfs(y);
14                low[x] = min(low[x], low[y]);
15            } else if (instk[y]) {
16                low[x] = min(low[x], time[y]);
17            }
18        if (time[x] == low[x]) {
19            scc.emplace_back();

```

```

21    for (int y = -1; y != x;) {
22        y = stk.back();
23        stk.pop_back();
24        instk[y] = 0;
25        scc.back().push_back(y);
26    }
27 }
28
29 void solve() {
30     for (int i = 0; i < n; i++)
31         if (!time[i]) dfs(i);
32     reverse(scc.begin(), scc.end());
33     // scc in topological order
34 }
35 };

```

3.4. Biconnected Components

3.4.1. Articulation Points

```

1 class Solution {
2     int cnt;
3     int dfs(int u, int p, vvi &adj, vi &vis, vi &low,
4             vvi &ans) {
5         if (vis[u] != -1) return low[u];
6         vis[u] = cnt, low[u] = cnt;
7         cnt++;
8         for (auto v : adj[u]) {
9             if (v == p) continue;

```

```

11   int temp = dfs(v, u, adj, vis, low, ans);
12   low[u] = min(low[u], low[v]);
13   if (temp > vis[u]) ans.push_back({u, v});
14   else low[u] = min(low[u], vis[v]);
15 }
16 return low[u];
}
17 vvi tarjanAlgorithm(int n, vvi &edges) {
18   vector<vector<int>> adj(n);
19   for (int i = 0; i < edges.size(); i++) {
20     int u = edges[i][0], v = edges[i][1];
21     adj[u].pb(v), adj[v].pb(u);
22   }
23   vi vis(n, -1), low(n, -1);
24   vector<vector<int>> ans;
25   cnt = 1;
26   dfs(0, -1, adj, vis, low, ans);
27   return ans;
}
28 };

```

3.4.2. Kosaraju

```

1 void SCC(vvi const &adj, vvi &components, vvi &adj_cond) {
2   int n = adj.size();
3   components.clear(), adj_cond.clear();
4   vector<int> order; // sorted (exit) list of G's vertices
5   visited.assign(n, false);

```

```

7   for (int i = 0; i < n; i++) // first dfs series
8     if (!visited[i]) dfs(i, adj, order);
9   // create adjacency list of G^T
10  vector<vector<int>> adj_rev(n);
11  for (int v = 0; v < n; v++)
12    for (int u : adj[v]) adj_rev[u].push_back(v);
13  visited.assign(n, false);
14  reverse(order.begin(), order.end());
15  vector<int> roots(n,
16                    0); // gives root vertex of vertex's SCC
17  for (auto v : order) // second dfs series
18    if (!visited[v]) {
19      std::vector<int> component;
20      dfs(v, adj_rev, component);
21      components.push_back(component);
22      int root = *component.begin();
23      for (auto u : component) roots[u] = root;
24    }
25  // add edges to condensation graph
26  adj_cond.assign(n, {});
27  for (int v = 0; v < n; v++)
28    for (auto u : adj[v])
29      if (roots[v] != roots[u])
30        adj_cond[roots[v]].push_back(roots[u]);
31  }

```

4. Math

4.1. SOS DP

```

1 const long long LOG = 20;
const long long sz = (1 << LOG);
3 void forward1(
4   vector<long long> &dp) { // subSet contribution to superset
5   for (int b = 0; b <= LOG; b++)
6     for (int i = 0; i <= sz; i++)
7       if (i & (1 << b)) dp[i] += dp[i ^ (1 << b)];
8 }
9 void backward1(vector<long long> &dp) { // undo offorward 1
10   for (int b = LOG; b >= 0; b--)
11     for (int i = sz; i >= 0; i--)
12       if (i & (1 << b)) dp[i] -= dp[i ^ (1 << b)];
13 }
14 void forward2(
15   vector<long long> &dp) { // superset contributes to subset
16     for (int b = 0; b <= LOG; b++)
17       for (int i = 0; i <= sz; i++)
18         if (i & (1 << b)) dp[i ^ (1 << b)] += dp[i];
19 }
20 void backward2(vector<long long> &dp) { // undo offorward 2
21   for (int b = LOG; b >= 0; b--)
22     for (int i = sz; i >= 0; i--)
23       if (i & (1 << b)) dp[i ^ (1 << b)] += dp[i];
}

```

4.2. Matrix

```

1 // Matrix A(n, n, 1e9+7); // Modular Arithmetic
2 // Matrix B(n, n); // Standard Arithmetic (mod = 0)
3 // Complexity: Multiplication O(N^3), Power O(N^3 log Exp)
4 struct Matrix {
5   vector<vector<ll>> mat;
6   int rows, cols;
7   ll mod; // mod = 0 implies Standard Arithmetic (No Modulo)
8   // Constructor: Default mod is 0 (No Mod)
9   Matrix(int r, int c, ll m = 0)
10    : rows(r), cols(c), mod(m) {
11      mat.assign(rows, vector<ll>(cols, 0));
12    }
13   void input() {
14     for (int i = 0; i < rows; ++i)
15       for (int j = 0; j < cols; ++j) cin >> mat[i][j];
16   }
17   Matrix operator+(const Matrix &other) const {
18     Matrix result(rows, cols, mod);
19     for (int i = 0; i < rows; ++i) {
20       for (int j = 0; j < cols; ++j) {
21         result.mat[i][j] = mat[i][j] + other.mat[i][j];
22         if (mod) result.mat[i][j] %= mod;
23       }
24     }
25     return result;
}

```

```

27 Matrix operator-(const Matrix &other) const {
28     Matrix result(rows, cols, mod);
29     for (int i = 0; i < rows; ++i) {
30         for (int j = 0; j < cols; ++j) {
31             result.mat[i][j] = mat[i][j] - other.mat[i][j];
32             if (mod)
33                 result.mat[i][j] =
34                     (result.mat[i][j] % mod + mod) % mod;
35         }
36     }
37     return result;
38 }
39 // Multiplication O(N^3)
40 Matrix operator*(const Matrix &other) const {
41     Matrix result(rows, other.cols, mod);
42     for (int i = 0; i < rows; ++i) {
43         for (int k = 0; k < cols;
44              ++k) { // Optimized loop order (i-k-j is cache
45                         // friendly)
46             if (mat[i][k] == 0)
47                 continue; // Optimization for sparse matrices
48             for (int j = 0; j < other.cols; ++j) {
49                 if (mod) {
50                     result.mat[i][j] =
51                         (result.mat[i][j] +
52                          mat[i][k] * other.mat[k][j]) %
53                         mod;
54                 } else {
55                     result.mat[i][j] += mat[i][k] * other.mat[k][j];
56                 }
57             }
58         }
59     }
60     return result;
61 }
62 // Matrix Exponentiation O(N^3 log Exp)
63 Matrix power(ll exp) const {
64     // Identity Matrix
65     Matrix result(rows, cols, mod);
66     for (int i = 0; i < rows; ++i) result.mat[i][i] = 1;
67     Matrix base = *this;
68     while (exp > 0) {
69         if (exp & 1) result = result * base;
70         base = base * base;
71         exp >>= 1;
72     }
73     return result;
74 };
75

```

4.3. Number Theory

4.3.1. Miller-Rabin

```

1 using ull = __uint128_t;
using ull = unsigned long long;

```

```

3 ull mod_mul(ull a, ull b, ull mod) {
4     return (u128)a * b % mod;
5 }
6 ull mod_pow(ull a, ull d, ull mod) {
7     ull res = 1;
8     while (d) {
9         if (d & 1) res = mod_mul(res, a, mod);
10        a = mod_mul(a, a, mod);
11        d >= 1;
12    }
13    return res;
14 }
15 bool isPrime(ull n) {
16     if (n < 2) return false;
17     for (ull p : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37})
18         if (n % p == 0) return n == p;
19     ull d = n - 1;
20     int s = __builtin_ctzll(d);
21     d >= s;
22     auto check = [&](ull a) {
23         ull x = mod_pow(a, d, n);
24         if (x == 1 || x == n - 1) return true;
25         for (int r = 1; r < s; r++) {
26             x = mod_mul(x, x, n);
27             if (x == n - 1) return true;
28         }
29         return false;
30     };
31     };
32     // Proven deterministic bases for 64-bit
33     // {2, 7, 61} for 32 bits
34     for (ull a :
35         {2, 325, 9375, 28178, 450775, 9780504, 1795265022})
36         if (a < n && !check(a)) return false;
37     }

```

4.3.2. Linear Sieve

```

1 constexpr ll MAXN = 1000000;
2 bitset<MAXN> is_prime;
3 vector<ll> primes;
4 ll mpf[MAXN], phi[MAXN], mu[MAXN];
5 void sieve() {
6     is_prime.set();
7     is_prime[1] = 0;
8     mu[1] = phi[1] = 1;
9     for (ll i = 2; i < MAXN; i++) {
10         if (is_prime[i]) {
11             mpf[i] = i;
12             primes.push_back(i);
13             phi[i] = i - 1;
14             mu[i] = -1;
15         }
16         for (ll p : primes) {
17             if (p > mpf[i] || i * p >= MAXN) break;
18             is_prime[i * p] = 0;
19             mpf[i * p] = i * p;
20             phi[i * p] = phi[i] * (p - 1);
21             mu[i * p] = -mu[i];
22         }
23     }
24 }

```

```

19    is_prime[i * p] = 0;
20    mpf[i * p] = p;
21    mu[i * p] = -mu[i];
22    if (i % p == 0)
23        phi[i * p] = phi[i] * p, mu[i * p] = 0;
24    else phi[i * p] = phi[i] * (p - 1);
25
26}
27
28}

```

4.3.3. Get Factors

```

1 vector<ll> all_factors(ll n) {
2     vector<ll> fac = {1};
3     while (n > 1) {
4         const ll p = mpf[n];
5         vector<ll> cur = {1};
6         while (n % p == 0) {
7             n /= p;
8             cur.push_back(cur.back() * p);
9         }
10        vector<ll> tmp;
11        for (auto x : fac)
12            for (auto y : cur) tmp.push_back(x * y);
13        tmp.swap(fac);
14    }
15    return fac;
}

```

4.3.4. Extended GCD

```

1 // returns (p, q, g): p * a + q * b == g == gcd(a, b)
2 // g is not guaranteed to be positive when a < 0 or b < 0
3 tuple<ll, ll, ll> extgcd(ll a, ll b) {
4     ll s = 1, t = 0, u = 0, v = 1;
5     while (b) {
6         ll q = a / b;
7         swap(a -= q * b, b);
8         swap(s -= q * t, t);
9         swap(u -= q * v, v);
10    }
11    return {s, u, a};
}

```

4.3.5. Chinese Remainder Theorem

Requires: Extended GCD

```

1 // for 0 <= a < m, 0 <= b < n, returns the smallest x >= 0
2 // such that x % m == a and x % n == b
3 ll crt(ll a, ll m, ll b, ll n) {
4     if (n > m) swap(a, b), swap(m, n);
5     auto [x, y, g] = extgcd(m, n);
6     assert((a - b) % g == 0); // no solution
7     x = ((b - a) / g * x) % (n / g) * m + a;
8     return x < 0 ? x + m / g * n : x;
}

```

4.3.6. Pollard Rho

```

1 ll f(ll x, ll mod) { return (x * x + 1) % mod; }
// n should be composite
3 ll pollard_rho(ll n) {
    if (!(n & 1)) return 2;
    while (1) {
        ll y = 2, x = RNG() % (n - 1) + 1, res = 1;
        for (int sz = 2; res == 1; sz *= 2) {
            for (int i = 0; i < sz && res <= 1; i++) {
                x = f(x, n);
                res = __gcd(abs(x - y), n);
            }
            y = x;
        }
        if (res != 0 && res != n) return res;
    }
}

```

4.3.7. De Bruijn Sequence

```

1 int res[kN], aux[kN], a[kN], sz;
void Rec(int t, int p, int n, int k) {
3     if (t > n) {
        if (n % p == 0)
            for (int i = 1; i <= p; ++i) res[sz++] = aux[i];
        } else {
7         aux[t] = aux[t - p];
        Rec(t + 1, p, n, k);
    }
}

```

```

9     for (aux[t] = aux[t - p] + 1; aux[t] < k; ++aux[t])
    Rec(t + 1, t, n, k);
11 }
13 }
13 int DeBruijn(int k, int n) {
// return cyclic string of length k^n such that every
// string of length n using k character appears as a
// substring.
15     if (k == 1) return res[0] = 0, 1;
fill(aux, aux + k * n, 0);
19     return sz = 0, Rec(1, 1, n, k), sz;
}

```

4.3.8. Combinatorics

```

1 struct Combinatorics {
    const int MOD;
3     vector<long long> fact, invFact;
5     // Constructor
    Combinatorics(int maxN, int mod)
        : MOD(mod), fact(maxN + 1), invFact(maxN + 1) {
        precompute(maxN);
7     }
9
11     // Function to perform modular exponentiation: a^b % MOD
11     long long modpow(long long a, long long b) const {
13         long long res = 1;
}

```

```

15   while (b) {
16     if (b & 1) res = res * a % MOD;
17     a = a * a % MOD;
18     b >>= 1;
19   }
20   return res;
21 }

// Precomputing factorials and modular inverses
23 void precompute(int maxN) {
24   fact[0] = 1;
25   for (int i = 1; i <= maxN; i++) {
26     fact[i] = fact[i - 1] * i % MOD;
27   }
28   invFact[maxN] =
29   modpow(fact[maxN], MOD - 2); // Fermat's Little Theorem
30   for (int i = maxN - 1; i >= 0; i--) {
31     invFact[i] = invFact[i + 1] * (i + 1) % MOD;
32   }
33 }

// Function to calculate nCk % MOD
35 long long nCk(int n, int k) const {
36   if (k > n || k < 0) return 0;
37   return fact[n] * invFact[k] % MOD * invFact[n - k] %
38         MOD;
39 }

41 // Function to calculate nPk % MOD
42 long long nPk(int n, int k) const {
43   if (k > n || k < 0) return 0;
44   return fact[n] * invFact[n - k] % MOD;
45 }

47 // Function to calculate n! % MOD
48 long long factorial(int n) const { return fact[n]; }
49 }

51 // Combinatorics comb(maxN,mod)

```

5. Geometry

```

1 struct Point {
2     ll x, y;
3     Point(ll _x = 0, ll _y = 0) : x(_x), y(_y) {}
4     bool operator==(const Point &other) const {
5         return x == other.x && y == other.y;
6     }
7     bool operator<(const Point &other) const {
8         if (x != other.x) return x < other.x;
9         return y < other.y;
10    }
11 };
12 ll cross_product(Point &A, Point &B, Point &C) {
13     // cross(A, B, C) tells you how the angle turns when you
14     // go A → B → C. If cross > 0 → left turn If cross < 0 →
15     // right turn (clockwise) If cross = 0 → collinear
16     return (B.x - A.x) * (C.y - A.y) -
17             (B.y - A.y) * (C.x - A.x);
18 }
19 long long dot_product(Point &A, Point &B, Point &C) {
20     // computes (B - A) · (C - A)
21     return (B.x - A.x) * (C.x - A.x) +
22             (B.y - A.y) * (C.y - A.y);
23 }
24 vector<Point> ConvexHullAndrowChain(vector<Point> pts) {
25     sort(pts);
26     pts.erase(unique(pts.begin(), pts.end()), pts.end());

```

```

27     int n = pts.size();
28     if (n <= 1) return pts;
29     vector<Point> lr, up;
30     for (int i = 0; i < n; ++i) { // Build lr hull
31         const Point &p = pts[i];
32         while (lr.size() >= 2 &&
33                 cross_product(lr[lr.size() - 2],
34                             lr[lr.size() - 1], p) <= 0) {
35             lr.pop_back();
36         }
37         lr.push_back(p);
38     }
39     for (int i = n - 1; i >= 0; --i) { // Build up hull
40         const Point &p = pts[i];
41         while (up.size() >= 2 &&
42                 cross_product(up[up.size() - 2],
43                             up[up.size() - 1], p) <= 0) {
44             up.pop_back();
45         }
46         up.push_back(p);
47     }
48     vector<Point> hull = lr;
49     for (int i = 1; i + 1 < (int)up.size(); ++i)
50         hull.push_back(up[i]);
51     return hull; // CCW order
52 }
```

6. Strings

6.1. Knuth-Morris-Pratt Algorithm

```

1 vector<int> pi(const string &s) {
2     vector<int> p(s.size());
3     for (int i = 1; i < s.size(); i++) {
4         int g = p[i - 1];
5         while (g && s[i] != s[g]) g = p[g - 1];
6         p[i] = g + (s[i] == s[g]);
7     }
8     return p;
9 }
10 vector<int> match(const string &s, const string &pat) {
11     vector<int> p = pi(pat + '\0' + s), res;
12     for (int i = p.size() - s.size(); i < p.size(); i++)
13         if (p[i] == pat.size())
14             res.push_back(i - 2 * pat.size());
15     return res;
16 }
```

6.2. Z Value

```

1 int z[n];
2 void zval(string s) {
3     // z[i] => longest common prefix of s and s[i:], i > 0
4     int n = s.size();
5     z[0] = 0;
6     for (int b = 0, i = 1; i < n; i++) {
```

```

7         if (z[b] + b <= i) z[i] = 0;
8         else z[i] = min(z[i - b], z[b] + b - i);
9         while (s[i + z[i]] == s[z[i]]) z[i]++;
10        if (i + z[i] > b + z[b]) b = i;
11    }
12 }
```

6.3. Manachers Algorithm

```

1 // d1[i] = number of odd-length palindromes centered at i
2 // d2[i] = number of even-length palindromes centered
3 // between i-1 and i
4 vector<int> d1, d2;
5 void manacher(const string &s) {
6     int n = s.size();
7     d1.assign(n, 0);
8     d2.assign(n, 0);
9     // Odd length palindromes
10    for (int i = 0, l = 0, r = -1; i < n; i++) {
11        int k = (i > r) ? 1 : min(d1[l + r - i], r - i + 1);
12        while (0 <= i - k && i + k < n && s[i - k] == s[i + k])
13            k++;
14        d1[i] = k;
15        if (i + k - 1 > r) {
16            l = i - k + 1;
17            r = i + k - 1;
18        }
19    }
20 }
```

```

// Even length palindromes
for (int i = 0, l = 0, r = -1; i < n; i++) {
    int k = (i > r) ? 0 : min(d2[l + r - i + 1], r - i + 1);
    while (0 <= i - k - 1 && i + k < n &&
           s[i - k - 1] == s[i + k])
        k++;
    d2[i] = k;
    if (i + k - 1 > r) {
        l = i - k;
        r = i + k - 1;
    }
}
}

13    }
14    };
15    vector<Node> nodes;
16    int
17    distWords; // Count of distinct words currently in Trie
18    int maxChars; // Alphabet size (usually 26)
19    int getBase(char c) {
20        return c - 'A';
21    }

22    Trie(int maxChars = 26) {
23        this->maxChars = maxChars;
24        nodes.clear();
25    }

```

6.4. Trie

```
1 class Trie {
2     public:
3         struct Node {
4             vector<int> next; // Indices of children nodes
5             int pfxCnt = 0; // How many words pass through this node
6             int wordCnt =
7                 0; // How many words end exactly at this node
8
9             Node(int maxChars) {
10                 next.assign(maxChars, -1);
11                 pfxCnt = 0;
12                 wordCnt = 0;
13             }
14         };
15
16         vector nodes;
17
18         void insert(string s) {
19             int curr = 0;
20             for (char &ch : s) {
21                 int base = getBase(ch);
22                 if (nodes[curr].next[base] == -1) {
23                     nodes[curr].next[base] = nodes.size();
24                     nodes.emplace_back(maxChars);
25                 }
26                 curr = nodes[curr].next[base];
27             }
28             nodes[curr].wordCnt++;
29         }
30
31         // Insert string s into Trie
32         void search(string s) {
33             int curr = 0;
34             for (char &ch : s) {
35                 int base = getBase(ch);
36                 if (nodes[curr].next[base] == -1) {
37                     cout << "No such word" << endl;
38                     return;
39                 }
40                 curr = nodes[curr].next[base];
41             }
42             cout << "Word found" << endl;
43         }
44
45         void startsWith(string s) {
46             int curr = 0;
47             for (char &ch : s) {
48                 int base = getBase(ch);
49                 if (nodes[curr].next[base] == -1) {
50                     cout << "No such prefix" << endl;
51                     return;
52                 }
53                 curr = nodes[curr].next[base];
54             }
55             cout << "Prefix found" << endl;
56         }
57
58         int getBase(char ch) {
59             if (ch >= 'a' && ch <= 'z') return ch - 'a';
60             if (ch >= 'A' && ch <= 'Z') return ch - 'A';
61             return -1;
62         }
63
64         int countWords() {
65             return nodes.size();
66         }
67
68         int countPrefixes() {
69             return countWords() - wordCnt;
70         }
71
72         int countMatched() {
73             return wordCnt;
74         }
75
76         int countPfxMatched() {
77             return pfxCnt;
78         }
79
80         int countPfxMatched(string s) {
81             int curr = 0;
82             for (char &ch : s) {
83                 int base = getBase(ch);
84                 if (nodes[curr].next[base] == -1) {
85                     return 0;
86                 }
87                 curr = nodes[curr].next[base];
88             }
89             return nodes[curr].pfxCnt;
90         }
91
92         int countMatched(string s) {
93             int curr = 0;
94             for (char &ch : s) {
95                 int base = getBase(ch);
96                 if (nodes[curr].next[base] == -1) {
97                     return 0;
98                 }
99                 curr = nodes[curr].next[base];
100            }
101            return nodes[curr].wordCnt;
102        }
103
104        int countPfxMatched(string s, int i) {
105            int curr = 0;
106            for (int j = i; j < s.length(); j++) {
107                int base = getBase(s[j]);
108                if (nodes[curr].next[base] == -1) {
109                    return 0;
110                }
111                curr = nodes[curr].next[base];
112            }
113            return nodes[curr].pfxCnt;
114        }
115
116        int countMatched(string s, int i) {
117            int curr = 0;
118            for (int j = i; j < s.length(); j++) {
119                int base = getBase(s[j]);
120                if (nodes[curr].next[base] == -1) {
121                    return 0;
122                }
123                curr = nodes[curr].next[base];
124            }
125            return nodes[curr].wordCnt;
126        }
127
128        int countPfxMatched(string s, int i, int j) {
129            int curr = 0;
130            for (int k = i; k < j; k++) {
131                int base = getBase(s[k]);
132                if (nodes[curr].next[base] == -1) {
133                    return 0;
134                }
135                curr = nodes[curr].next[base];
136            }
137            return nodes[curr].pfxCnt;
138        }
139
140        int countMatched(string s, int i, int j) {
141            int curr = 0;
142            for (int k = i; k < j; k++) {
143                int base = getBase(s[k]);
144                if (nodes[curr].next[base] == -1) {
145                    return 0;
146                }
147                curr = nodes[curr].next[base];
148            }
149            return nodes[curr].wordCnt;
150        }
151
152        int countPfxMatched(string s, int i, int j, int k) {
153            int curr = 0;
154            for (int l = i; l < j; l++) {
155                int base = getBase(s[l]);
156                if (nodes[curr].next[base] == -1) {
157                    return 0;
158                }
159                curr = nodes[curr].next[base];
160            }
161            for (int m = j; m < k; m++) {
162                int base = getBase(s[m]);
163                if (nodes[curr].next[base] == -1) {
164                    return 0;
165                }
166                curr = nodes[curr].next[base];
167            }
168            return nodes[curr].pfxCnt;
169        }
170
171        int countMatched(string s, int i, int j, int k) {
172            int curr = 0;
173            for (int l = i; l < j; l++) {
174                int base = getBase(s[l]);
175                if (nodes[curr].next[base] == -1) {
176                    return 0;
177                }
178                curr = nodes[curr].next[base];
179            }
180            for (int m = j; m < k; m++) {
181                int base = getBase(s[m]);
182                if (nodes[curr].next[base] == -1) {
183                    return 0;
184                }
185                curr = nodes[curr].next[base];
186            }
187            return nodes[curr].wordCnt;
188        }
189
190        int countPfxMatched(string s, int i, int j, int k, int l) {
191            int curr = 0;
192            for (int m = i; m < j; m++) {
193                int base = getBase(s[m]);
194                if (nodes[curr].next[base] == -1) {
195                    return 0;
196                }
197                curr = nodes[curr].next[base];
198            }
199            for (int n = j; n < k; n++) {
200                int base = getBase(s[n]);
201                if (nodes[curr].next[base] == -1) {
202                    return 0;
203                }
204                curr = nodes[curr].next[base];
205            }
206            for (int o = k; o < l; o++) {
207                int base = getBase(s[o]);
208                if (nodes[curr].next[base] == -1) {
209                    return 0;
210                }
211                curr = nodes[curr].next[base];
212            }
213            return nodes[curr].pfxCnt;
214        }
215
216        int countMatched(string s, int i, int j, int k, int l) {
217            int curr = 0;
218            for (int m = i; m < j; m++) {
219                int base = getBase(s[m]);
220                if (nodes[curr].next[base] == -1) {
221                    return 0;
222                }
223                curr = nodes[curr].next[base];
224            }
225            for (int n = j; n < k; n++) {
226                int base = getBase(s[n]);
227                if (nodes[curr].next[base] == -1) {
228                    return 0;
229                }
230                curr = nodes[curr].next[base];
231            }
232            for (int o = k; o < l; o++) {
233                int base = getBase(s[o]);
234                if (nodes[curr].next[base] == -1) {
235                    return 0;
236                }
237                curr = nodes[curr].next[base];
238            }
239            return nodes[curr].wordCnt;
240        }
241
242        int countPfxMatched(string s, int i, int j, int k, int l, int m) {
243            int curr = 0;
244            for (int n = i; n < j; n++) {
245                int base = getBase(s[n]);
246                if (nodes[curr].next[base] == -1) {
247                    return 0;
248                }
249                curr = nodes[curr].next[base];
250            }
251            for (int o = j; o < k; o++) {
252                int base = getBase(s[o]);
253                if (nodes[curr].next[base] == -1) {
254                    return 0;
255                }
256                curr = nodes[curr].next[base];
257            }
258            for (int p = k; p < l; p++) {
259                int base = getBase(s[p]);
260                if (nodes[curr].next[base] == -1) {
261                    return 0;
262                }
263                curr = nodes[curr].next[base];
264            }
265            for (int q = l; q < m; q++) {
266                int base = getBase(s[q]);
267                if (nodes[curr].next[base] == -1) {
268                    return 0;
269                }
270                curr = nodes[curr].next[base];
271            }
272            return nodes[curr].pfxCnt;
273        }
274
275        int countMatched(string s, int i, int j, int k, int l, int m) {
276            int curr = 0;
277            for (int n = i; n < j; n++) {
278                int base = getBase(s[n]);
279                if (nodes[curr].next[base] == -1) {
280                    return 0;
281                }
282                curr = nodes[curr].next[base];
283            }
284            for (int o = j; o < k; o++) {
285                int base = getBase(s[o]);
286                if (nodes[curr].next[base] == -1) {
287                    return 0;
288                }
289                curr = nodes[curr].next[base];
290            }
291            for (int p = k; p < l; p++) {
292                int base = getBase(s[p]);
293                if (nodes[curr].next[base] == -1) {
294                    return 0;
295                }
296                curr = nodes[curr].next[base];
297            }
298            for (int q = l; q < m; q++) {
299                int base = getBase(s[q]);
300                if (nodes[curr].next[base] == -1) {
301                    return 0;
302                }
303                curr = nodes[curr].next[base];
304            }
305            return nodes[curr].wordCnt;
306        }
307
308        int countPfxMatched(string s, int i, int j, int k, int l, int m, int n) {
309            int curr = 0;
310            for (int o = i; o < j; o++) {
311                int base = getBase(s[o]);
312                if (nodes[curr].next[base] == -1) {
313                    return 0;
314                }
315                curr = nodes[curr].next[base];
316            }
317            for (int p = j; p < k; p++) {
318                int base = getBase(s[p]);
319                if (nodes[curr].next[base] == -1) {
320                    return 0;
321                }
322                curr = nodes[curr].next[base];
323            }
324            for (int q = k; q < l; q++) {
325                int base = getBase(s[q]);
326                if (nodes[curr].next[base] == -1) {
327                    return 0;
328                }
329                curr = nodes[curr].next[base];
330            }
331            for (int r = l; r < m; r++) {
332                int base = getBase(s[r]);
333                if (nodes[curr].next[base] == -1) {
334                    return 0;
335                }
336                curr = nodes[curr].next[base];
337            }
338            for (int s = m; s < n; s++) {
339                int base = getBase(s[s]);
340                if (nodes[curr].next[base] == -1) {
341                    return 0;
342                }
343                curr = nodes[curr].next[base];
344            }
345            return nodes[curr].pfxCnt;
346        }
347
348        int countMatched(string s, int i, int j, int k, int l, int m, int n) {
349            int curr = 0;
350            for (int o = i; o < j; o++) {
351                int base = getBase(s[o]);
352                if (nodes[curr].next[base] == -1) {
353                    return 0;
354                }
355                curr = nodes[curr].next[base];
356            }
357            for (int p = j; p < k; p++) {
358                int base = getBase(s[p]);
359                if (nodes[curr].next[base] == -1) {
360                    return 0;
361                }
362                curr = nodes[curr].next[base];
363            }
364            for (int q = k; q < l; q++) {
365                int base = getBase(s[q]);
366                if (nodes[curr].next[base] == -1) {
367                    return 0;
368                }
369                curr = nodes[curr].next[base];
370            }
371            for (int r = l; r < m; r++) {
372                int base = getBase(s[r]);
373                if (nodes[curr].next[base] == -1) {
374                    return 0;
375                }
376                curr = nodes[curr].next[base];
377            }
378            for (int s = m; s < n; s++) {
379                int base = getBase(s[s]);
380                if (nodes[curr].next[base] == -1) {
381                    return 0;
382                }
383                curr = nodes[curr].next[base];
384            }
385            return nodes[curr].wordCnt;
386        }
387
388        int countPfxMatched(string s, int i, int j, int k, int l, int m, int n, int o) {
389            int curr = 0;
390            for (int p = i; p < j; p++) {
391                int base = getBase(s[p]);
392                if (nodes[curr].next[base] == -1) {
393                    return 0;
394                }
395                curr = nodes[curr].next[base];
396            }
397            for (int q = j; q < k; q++) {
398                int base = getBase(s[q]);
399                if (nodes[curr].next[base] == -1) {
400                    return 0;
401                }
402                curr = nodes[curr].next[base];
403            }
404            for (int r = k; r < l; r++) {
405                int base = getBase(s[r]);
406                if (nodes[curr].next[base] == -1) {
407                    return 0;
408                }
409                curr = nodes[curr].next[base];
410            }
411            for (int s = l; s < m; s++) {
412                int base = getBase(s[s]);
413                if (nodes[curr].next[base] == -1) {
414                    return 0;
415                }
416                curr = nodes[curr].next[base];
417            }
418            for (int t = m; t < n; t++) {
419                int base = getBase(s[t]);
420                if (nodes[curr].next[base] == -1) {
421                    return 0;
422                }
423                curr = nodes[curr].next[base];
424            }
425            for (int u = n; u < o; u++) {
426                int base = getBase(s[u]);
427                if (nodes[curr].next[base] == -1) {
428                    return 0;
429                }
430                curr = nodes[curr].next[base];
431            }
432            return nodes[curr].pfxCnt;
433        }
434
435        int countMatched(string s, int i, int j, int k, int l, int m, int n, int o) {
436            int curr = 0;
437            for (int p = i; p < j; p++) {
438                int base = getBase(s[p]);
439                if (nodes[curr].next[base] == -1) {
440                    return 0;
441                }
442                curr = nodes[curr].next[base];
443            }
444            for (int q = j; q < k; q++) {
445                int base = getBase(s[q]);
446                if (nodes[curr].next[base] == -1) {
447                    return 0;
448                }
449                curr = nodes[curr].next[base];
450            }
451            for (int r = k; r < l; r++) {
452                int base = getBase(s[r]);
453                if (nodes[curr].next[base] == -1) {
454                    return 0;
455                }
456                curr = nodes[curr].next[base];
457            }
458            for (int s = l; s < m; s++) {
459                int base = getBase(s[s]);
460                if (nodes[curr].next[base] == -1) {
461                    return 0;
462                }
463                curr = nodes[curr].next[base];
464            }
465            for (int t = m; t < n; t++) {
466                int base = getBase(s[t]);
467                if (nodes[curr].next[base] == -1) {
468                    return 0;
469                }
470                curr = nodes[curr].next[base];
471            }
472            for (int u = n; u < o; u++) {
473                int base = getBase(s[u]);
474                if (nodes[curr].next[base] == -1) {
475                    return 0;
476                }
477                curr = nodes[curr].next[base];
478            }
479            return nodes[curr].wordCnt;
480        }
481
482        int countPfxMatched(string s, int i, int j, int k, int l, int m, int n, int o, int p) {
483            int curr = 0;
484            for (int q = i; q < j; q++) {
485                int base = getBase(s[q]);
486                if (nodes[curr].next[base] == -1) {
487                    return 0;
488                }
489                curr = nodes[curr].next[base];
490            }
491            for (int r = j; r < k; r++) {
492                int base = getBase(s[r]);
493                if (nodes[curr].next[base] == -1) {
494                    return 0;
495                }
496                curr = nodes[curr].next[base];
497            }
498            for (int s = k; s < l; s++) {
499                int base = getBase(s[s]);
500                if (nodes[curr].next[base] == -1) {
501                    return 0;
502                }
503                curr = nodes[curr].next[base];
504            }
505            for (int t = l; t < m; t++) {
506                int base = getBase(s[t]);
507                if (nodes[curr].next[base] == -1) {
508                    return 0;
509                }
510                curr = nodes[curr].next[base];
511            }
512            for (int u = m; u < n; u++) {
513                int base = getBase(s[u]);
514                if (nodes[curr].next[base] == -1) {
515                    return 0;
516                }
517                curr = nodes[curr].next[base];
518            }
519            for (int v = n; v < o; v++) {
520                int base = getBase(s[v]);
521                if (nodes[curr].next[base] == -1) {
522                    return 0;
523                }
524                curr = nodes[curr].next[base];
525            }
526            for (int w = o; w < p; w++) {
527                int base = getBase(s[w]);
528                if (nodes[curr].next[base] == -1) {
529                    return 0;
530                }
531                curr = nodes[curr].next[base];
532            }
533            return nodes[curr].pfxCnt;
534        }
535
536        int countMatched(string s, int i, int j, int k, int l, int m, int n, int o, int p) {
537            int curr = 0;
538            for (int q = i; q < j; q++) {
539                int base = getBase(s[q]);
540                if (nodes[curr].next[base] == -1) {
541                    return 0;
542                }
543                curr = nodes[curr].next[base];
544            }
545            for (int r = j; r < k; r++) {
546                int base = getBase(s[r]);
547                if (nodes[curr].next[base] == -1) {
548                    return 0;
549                }
550                curr = nodes[curr].next[base];
551            }
552            for (int s = k; s < l; s++) {
553                int base = getBase(s[s]);
554                if (nodes[curr].next[base] == -1) {
555                    return 0;
556                }
557                curr = nodes[curr].next[base];
558            }
559            for (int t = l; t < m; t++) {
560                int base = getBase(s[t]);
561                if (nodes[curr].next[base] == -1) {
562                    return 0;
563                }
564                curr = nodes[curr].next[base];
565            }
566            for (int u = m; u < n; u++) {
567                int base = getBase(s[u]);
568                if (nodes[curr].next[base] == -1) {
569                    return 0;
570                }
571                curr = nodes[curr].next[base];
572            }
573            for (int v = n; v < o; v++) {
574                int base = getBase(s[v]);
575                if (nodes[curr].next[base] == -1) {
576                    return 0;
577                }
578                curr = nodes[curr].next[base];
579            }
580            for (int w = o; w < p; w++) {
581                int base = getBase(s[w]);
582                if (nodes[curr].next[base] == -1) {
583                    return 0;
584                }
585                curr = nodes[curr].next[base];
586            }
587            return nodes[curr].wordCnt;
588        }
589
590        int countPfxMatched(string s, int i, int j, int k, int l, int m, int n, int o, int p, int q) {
591            int curr = 0;
592            for (int r = i; r < j; r++) {
593                int base = getBase(s[r]);
594                if (nodes[curr].next[base] == -1) {
595                    return 0;
596                }
597                curr = nodes[curr].next[base];
598            }
599            for (int s = j; s < k; s++) {
600                int base = getBase(s[s]);
601                if (nodes[curr].next[base] == -1) {
602                    return 0;
603                }
604                curr = nodes[curr].next[base];
605            }
606            for (int t = k; t < l; t++) {
607                int base = getBase(s[t]);
608                if (nodes[curr].next[base] == -1) {
609                    return 0;
610                }
611                curr = nodes[curr].next[base];
612            }
613            for (int u = l; u < m; u++) {
614                int base = getBase(s[u]);
615                if (nodes[curr].next[base] == -1) {
616                    return 0;
617                }
618                curr = nodes[curr].next[base];
619            }
620            for (int v = m; v < n; v++) {
621                int base = getBase(s[v]);
622                if (nodes[curr].next[base] == -1) {
623                    return 0;
624                }
625                curr = nodes[curr].next[base];
626            }
627            for (int w = n; w < o; w++) {
628                int base = getBase(s[w]);
629                if (nodes[curr].next[base] == -1) {
630                    return 0;
631                }
632                curr = nodes[curr].next[base];
633            }
634            for (int x = o; x < p; x++) {
635                int base = getBase(s[x]);
636                if (nodes[curr].next[base] == -1) {
637                    return 0;
638                }
639                curr = nodes[curr].next[base];
640            }
641            for (int y = p; y < q; y++) {
642                int base = getBase(s[y]);
643                if (nodes[curr].next[base] == -1) {
644                    return 0;
645                }
646                curr = nodes[curr].next[base];
647            }
648            return nodes[curr].pfxCnt;
649        }
650
651        int countMatched(string s, int i, int j, int k, int l, int m, int n, int o, int p, int q) {
652            int curr = 0;
653            for (int r = i; r < j; r++) {
654                int base = getBase(s[r]);
655                if (nodes[curr].next[base] == -1) {
656                    return 0;
657                }
658                curr = nodes[curr].next[base];
659            }
660            for (int s = j; s < k; s++) {
661                int base = getBase(s[s]);
662                if (nodes[curr].next[base] == -1) {
663                    return 0;
664                }
665                curr = nodes[curr].next[base];
666            }
667            for (int t = k; t < l; t++) {
668                int base = getBase(s[t]);
669                if (nodes[curr].next[base] == -1) {
670                    return 0;
671                }
672                curr = nodes[curr].next[base];
673            }
674            for (int u = l; u < m; u++) {
675                int base = getBase(s[u]);
676                if (nodes[curr].next[base] == -1) {
677                    return 0;
678                }
679                curr = nodes[curr].next[base];
680            }
681            for (int v = m; v < n; v++) {
682                int base = getBase(s[v]);
683                if (nodes[curr].next[base] == -1) {
684                    return 0;
685                }
686                curr = nodes[curr].next[base];
687            }
688            for (int w = n; w < o; w++) {
689                int base = getBase(s[w]);
690                if (nodes[curr].next[base] == -1) {
691                    return 0;
692                }
693                curr = nodes[curr].next[base];
694            }
695            for (int x = o; x < p; x++) {
696                int base = getBase(s[x]);
697                if (nodes[curr].next[base] == -1) {
698                    return 0;
699                }
700                curr = nodes[curr].next[base];
701            }
702            for (int y = p; y < q; y++) {
703                int base = getBase(s[y]);
704                if (nodes[curr].next[base] == -1) {
705                    return 0;
706                }
707                curr = nodes[curr].next[base];
708            }
709            return nodes[curr].wordCnt;
710        }
711
712        int countPfxMatched(string s, int i, int j, int k, int l, int m, int n, int o, int p, int q, int r) {
713            int curr = 0;
714            for (int s = i; s < j; s++) {
715                int base = getBase(s[s]);
716                if (nodes[curr].next[base] == -1) {
717                    return 0;
718                }
719                curr = nodes[curr].next[base];
720            }
721            for (int t = j; t < k; t++) {
722                int base = getBase(s[t]);
723                if (nodes[curr].next[base] == -1) {
724                    return 0;
725                }
726                curr = nodes[curr].next[base];
727            }
728            for (int u = k; u < l; u++) {
729                int base = getBase(s[u]);
730                if (nodes[curr].next[base] == -1) {
731                    return 0;
732                }
733                curr = nodes[curr].next[base];
734            }
735            for (int v = l; v < m; v++) {
736                int base = getBase(s[v]);
737                if (nodes[curr].next[base] == -1) {
738                    return 0;
739                }
740                curr = nodes[curr].next[base];
741            }
742            for (int w = m; w < n; w++) {
743                int base = getBase(s[w]);
744                if (nodes[curr].next[base] == -1) {
745                    return 0;
746                }
747                curr = nodes[curr].next[base];
748            }
749            for (int x = n; x < o; x++) {
750                int base = getBase(s[x]);
751                if (nodes[curr].next[base] == -1) {
752                    return 0;
753                }
754                curr = nodes[curr].next[base];
755            }
756            for (int y = o; y < p; y++) {
757                int base = getBase(s[y]);
758                if (nodes[curr].next[base] == -1) {
759                    return 0;
760                }
761                curr = nodes[curr].next[base];
762            }
763            for (int z = p; z < q; z++) {
764                int base = getBase(s[z]);
765                if (nodes[curr].next[base] == -1) {
766                    return 0;
767                }
768                curr = nodes[curr].next[base];
769            }
770            for (int a = q; a < r; a++) {
771                int base = getBase(s[a]);
772                if (nodes[curr].next[base] == -1) {
773                    return 0;
774                }
775                curr = nodes[curr].next[base];
776            }
777            return nodes[curr].pfxCnt;
778        }
779
780        int countMatched(string s, int i, int j, int k, int l, int m, int n, int o, int p, int q, int r) {
781            int curr = 0;
782            for (int s = i; s < j; s++) {
783                int base = getBase(s[s]);
784                if (nodes[curr].next[base] == -1) {
785                    return 0;
786                }
787                curr = nodes[curr].next[base];
788            }
789            for (int t = j; t < k; t++) {
790                int base = getBase(s[t]);
791                if (nodes[curr].next[base] == -1) {
792                    return 0;
793                }
794                curr = nodes[curr].next[base];
795            }
796            for (int u = k; u < l; u++) {
797                int base = getBase(s[u]);
798                if (nodes[curr].next[base] == -1) {
799                    return 0;
800                }
801                curr = nodes[curr].next[base];
802            }
803            for (int v = l; v < m; v++) {
804                int base = getBase(s[v]);
805                if (nodes[curr].next[base] == -1) {
806                    return 0;
807                }
808                curr = nodes[curr].next[base];
809            }
810            for (int w = m; w < n; w++) {
811                int base = getBase(s[w]);
812                if (nodes[curr].next[base] == -1) {
813                    return 0;
814                }
815                curr = nodes[curr].next[base];
816            }
817            for (int x = n; x < o; x++) {
818                int base = getBase(s[x]);
819                if (nodes[curr].next[base] == -1) {
820                    return 0;
821                }
822                curr = nodes[curr].next[base];
823            }
824            for (int y = o; y < p; y++) {
825                int base = getBase(s[y]);
826                if (nodes[curr].next[base] == -1) {
827                    return 0;
828                }
829                curr = nodes[curr].next[base];
830            }
831            for (int z = p; z < q; z++) {
832                int base = getBase(s[z]);
833                if (nodes[curr].next[base] == -1) {
834                    return 0;
835                }
836                curr = nodes[curr].next[base];
837            }
838            for (int a = q; a < r; a++) {
839                int base = getBase(s[a]);
840                if (nodes[curr].next[base] == -1) {
841                    return 0;
842                }
843                curr = nodes[curr].next[base];
844            }
845            return nodes[curr].wordCnt;
846        }
847
848        int countPfxMatched(string s, int i, int j, int k, int l, int m, int n, int o, int p, int q, int r, int s) {
849            int curr = 0;
850            for (int t = i; t < j; t++) {
851                int base = getBase(s[t]);
852                if (nodes[curr].next[base] == -1) {
853                    return 0;
854                }
855                curr = nodes[curr].next[base];
856            }
857            for (int u = j; u < k; u++) {
858                int base = getBase(s[u]);
859                if (nodes[curr].next[base] == -1) {
860                    return 0;
861                }
862                curr = nodes[curr].next[base];
863            }
864            for (int v = k; v < l; v++) {
865                int base = getBase(s[v]);
866                if (nodes[curr].next[base] == -1) {
867                    return 0;
868                }
869                curr = nodes[curr].next[base];
870            }
871            for (int w = l; w < m; w++) {
872                int base = getBase(s[w]);
873                if (nodes[curr].next[base] == -1) {
874                    return 0;
875                }
876                curr = nodes[curr].next[base];
877            }
878            for (int x = m; x < n; x++) {
879                int base = getBase(s[x]);
880                if (nodes[curr].next[base] == -1) {
881                    return 0;
882                }
883                curr = nodes[curr].next[base];
884            }
885            for (int y = n; y < o; y++) {
886                int base = getBase(s[y]);
887                if (nodes[curr].next[base] == -1) {
888                    return 0;
889                }
890                curr = nodes[curr].next[base];
891            }
892            for (int z = o; z < p; z++) {
893                int base = getBase(s[z]);
894                if (nodes[curr].next[base] == -1) {
895                    return 0;
896                }
897                curr = nodes[curr].next[base];
898            }
899            for (int a = p; a < q; a++) {
900                int base = getBase(s[a]);
901                if (nodes[curr].next[base] == -1) {
902                    return 0;
903                }
904                curr = nodes[curr].next[base];
905            }
906            for (int b = q; b < r; b++) {
907                int base = getBase(s[b]);
908                if (nodes[curr].next[base] == -1) {
909                    return 0;
910                }
911                curr = nodes[curr].next[base];
912            }
913            for (int c = r; c < s; c++) {
914                int base = getBase(s[c]);
915                if (nodes[curr].next[base] == -1) {
916                    return 0;
917                }
918                curr = nodes[curr].next[base];
919            }
920            return nodes[curr].pfxCnt;
921        }
922
923        int countMatched(string s, int i, int j, int k, int l, int m, int n, int o, int p, int q, int r, int s) {
924            int curr = 0;
925            for (int t = i; t < j; t++) {
926                int base = getBase(s[t]);
927                if (nodes[curr].next[base] == -1) {
928                    return 0;
929                }
930                curr = nodes[curr].next[base];
931            }
932            for (int u = j; u < k; u++) {
933                int base = getBase(s[u]);
934                if (nodes[curr].next[base] == -1) {
935                    return 0;
936                }
937                curr = nodes[curr].next[base];
938            }
939            for (int v = k; v < l; v++) {
940                int base = getBase(s[v]);
941                if (nodes[curr].next[base] == -1) {
942                    return 0;
943                }
944                curr = nodes[curr].next[base];
945            }
946            for (int w = l; w < m; w++) {
947                int base = getBase(s[w]);
948                if (nodes[curr].next[base] == -1) {
949                    return 0;
950                }
951                curr = nodes[curr].next[base];
952            }
953            for (int x = m; x < n; x++) {
954                int base = getBase(s[x]);
955                if (nodes[curr].next[base] == -1) {
956                    return 0;
957                }
958                curr = nodes[curr].next[base];
959            }
960            for (int y = n; y < o; y++) {
961                int base = getBase(s[y]);
962                if (nodes[curr].next[base] == -1) {
963                    return 0;
964                }
965                curr = nodes[curr].next[base];
966            }
967            for (int z = o; z < p; z++) {
968                int base = getBase(s[z]);
969                if (nodes[curr].next[base] == -1) {
970                    return 0;
971                }
972                curr = nodes[curr].next[base];
973            }
974            for (int a = p; a < q; a++) {
975                int base = getBase(s[a]);
976                if (nodes[curr].next[base] == -1) {
977                    return 0;
978                }
979                curr = nodes[curr].next[base];
980            }
981            for (int b = q; b < r; b++) {
982                int base = getBase(s[b]);
983                if (nodes[curr].next[base] == -1) {
984                    return 0;
985                }
986                curr = nodes[curr].next[base];
987            }
988            for (int c = r; c < s; c++) {
989                int base = getBase(s[c]);
990                if (nodes[curr].next[base] == -1) {
991                    return 0;
992                }
993                curr = nodes[curr].next[base];
994            }
995            return nodes[curr].wordCnt;
996        }
997    }
998}
```

```

    nodes.emplace_back(maxChars);
41   }
42   curr = nodes[curr].next[base];
43   nodes[curr].pfxCnt++;
44 }
45 if (nodes[curr].wordCnt == 0) {
46   distWords++; // New distinct word found
47 }
48 nodes[curr].wordCnt++;
49 }

// Check if string s exists
50 bool search(string s) {
51   int curr = 0;
52   for (char &ch : s) {
53     int base = getBase(ch);
54     if (nodes[curr].next[base] == -1) return false;
55     curr = nodes[curr].next[base];
56   }
57   return nodes[curr].wordCnt > 0;
58 }
59 // Delete one occurrence of s
60 void erase(string s) {
61   if (!search(s)) return; // Check existence first
62   int curr = 0;
63   nodes[curr].pfxCnt--;
64   for (char &ch : s) {
65
66     int base = getBase(ch);
67     curr = nodes[curr].next[base];
68     nodes[curr].pfxCnt--;
69   }
70
71   if (nodes[curr].wordCnt == 0)
72     distWords--; // Word completely removed
73
74
75   // Count words that have s as a prefix
76   int prefixCount(string s) {
77     int curr = 0;
78     for (char &ch : s) {
79       int base = getBase(ch);
80       if (nodes[curr].next[base] == -1)
81         return 0; // Prefix not found
82       curr = nodes[curr].next[base];
83     }
84     return nodes[curr].pfxCnt;
85   }
86
87 };

```

6.5. Aho-Corasick Automaton

```

1 const int ALPHA = 26, MAXNODES = 500000 + 5;
2 int nxt[MAXNODES][ALPHA];
3 int linkS[MAXNODES];
4 ll cntNode[MAXNODES];

```

```

5 vector<int> adjSL[MAXNODES];
vector<int> patEnd;
7 int nodes = 1;

9 void build_trie(const vector<string> &P) {
    // clear
11 for (int i = 0; i < nodes; i++) {
    memset(nxt[i], 0, sizeof nxt[i]);
    cntNode[i] = 0;
    adjSL[i].clear();
15 }
    nodes = 1;
17 patEnd.clear();
    patEnd.reserve(P.size());
19 // insert
    for (auto &pat : P) {
        int u = 0;
        for (char ch : pat) {
            int c = ch - 'a';
            if (!nxt[u][c]) nxt[u][c] = nodes++;
            u = nxt[u][c];
23     }
        patEnd.pb(u);
27     }
29 }
vector<int> bfsOrder;
31 void build_links() {
33     queue<int> q;
linkS[0] = 0;
// first layer
35 for (int c = 0; c < ALPHA; c++) {
    int v = nxt[0][c];
    if (v) {
        linkS[v] = 0;
        q.push(v);
    }
}
// BFS
39 while (!q.empty()) {
41     int u = q.front();
43     q.pop();
bfsOrder.pb(u);
45     for (int c = 0; c < ALPHA; c++) {
        int v = nxt[u][c];
        if (!v) continue;
        int j = linkS[u];
51         while (j && !nxt[j][c]) j = linkS[j];
        if (nxt[j][c]) j = nxt[j][c];
        linkS[v] = j;
        q.push(v);
53     }
}
55 }
57 for (int u : bfsOrder) { adjSL[linkS[u]].pb(u); }
}

```

```

59 void solve() {
61     string S;
62     ll k;
63     cin >> S >> k;
64     vector<string> P(k);
65     for (int i = 0; i < k; i++) cin >> P[i];
66     build_trie(P);
67     bfsOrder.clear();
68     build_links();
69 }

15     uint32_t val = (s[i - 1] - 'A' + 1);
16     pref1[i] =
17         (val + (uint64_t)BASE1 * pref1[i - 1]) % MOD1;
18     pref2[i] =
19         (val + (uint64_t)BASE2 * pref2[i - 1]) % MOD2;
20     pow1[i] = (uint64_t)pow1[i - 1] * BASE1 % MOD1;
21     pow2[i] = (uint64_t)pow2[i - 1] * BASE2 % MOD2;
22 }
23
24 inline uint64_t getHash(int l, int r) {
25     uint32_t h1 =
26         (pref1[r + 1] -
27          (uint64_t)pref1[l] * pow1[r - l + 1] % MOD1 + MOD1) %
28          MOD1;
29     uint32_t h2 =
30         (pref2[r + 1] -
31          (uint64_t)pref2[l] * pow2[r - l + 1] % MOD2 + MOD2) %
32          MOD2;
33     return (uint64_t)(h2 << 32) | h1;
34 }
35
36 Hasher64 s(a);
37 unordered_map<uint64_t, int> mp;
38 mp.reserve(n);
39 mp.max_load_factor(0.7);
40 uint64_t h = s.getHash(i, i + len - 1);

```

6.6. Hashing

```

1 #define MOD1 999119999
2 #define MOD2 999999733
3 struct Hasher64 {
4     int n;
5     vector<uint32_t> pref1, pref2, pow1, pow2;
6     Hasher64() {}
7     Hasher64(const string &s) {
8         n = s.size();
9         pref1.assign(n + 1, 0);
10        pref2.assign(n + 1, 0);
11        pow1.assign(n + 1, 0);
12        pow2.assign(n + 1, 0);
13        pow1[0] = pow2[0] = 1;
14        for (int i = 1; i <= n; i++) {

```