
Contents		
1 Misc	2	4 Math 13
1.1 Contest	2	4.1 SOS DP 13
1.1.1 Macros and debug	2	4.2 Matrix 13
2 Data Structures	3	4.3 Number Theory 14
2.1 GNU PBDS	3	4.3.1 Mod Struct 14
2.2 2D Partial Sums	3	4.3.2 Miller-Rabin 15
2.3 Sparse Table	4	4.3.3 Linear Sieve 16
2.4 Fenwick Tree	5	4.3.4 Get Factors 16
2.5 Longest Increasing Subsequence	6	4.3.5 Extended GCD 16
2.6 Xobriest	6	4.3.6 Chinese Remainder Theorem 17
3 Graph	7	4.3.7 Pollard Rho 17
3.1 General	7	4.3.8 De Bruijn Sequence 17
3.1.1 Bellman	7	4.3.9 Combinatorics 18
3.1.2 Floyd	7	5 Geometry 20
3.1.3 DSU	8	5.1 convex hull 20
3.1.4 Binary Lifting	8	6 Strings 22
3.2 Kuhn-Munkres algorithm	9	6.1 Knuth-Morris-Pratt Algorithm 22
3.3 Strongly Connected Components	11	6.2 Z Value 22
3.4 Biconnected Components	11	6.3 Manachers Algorithm 22
3.4.1 Articulation Points	11	6.4 Trie 23
		6.5 Aho-Corasick Automaton 24

1.	Misc		
1.1.	Contest	23	- check for `==`, `>=`, `acos(1.00000001)`, etc.
1.1.1.	Macros and debug	25	- Did you forget to sort or unique?
1	<code>#pragma GCC optimize("O3", "unroll-loops")</code>	27	- Generate large and worst "corner" cases.
1	- Pre-submit:	29	- Check your `m` / `n`, `i` / `j` and `x` / `y`.
3	- Did you make a typo when copying a template?	31	- Are everything initialized or reset properly?
5	- Test more cases if unsure.	33	- Are you sure about the STL thing you are using?
7	- Write a naive solution and check small cases.	35	- Read cppreference (should be available).
9	- Submit the correct file.	37	- Print everything and run it on pen and paper.
11	- General Debugging:	39	- Time Limit Exceeded:
13	- Read the whole problem again.	41	- Calculate your time complexity again.
15	- Have a teammate read the problem.	43	- Does the program actually end?
17	- Have a teammate read your code.	45	- Check for `while(q.size())` etc.
19	- Explain your solution to them (or a rubber duck).		- Test the largest cases locally.
21	- Print the code and its output / debug output.		- Did you do unnecessary stuff?
	- Go to the toilet.		- e.g. pass vectors by value
	- Wrong Answer:		- e.g. `memset` for every test case
	- Any possible overflows?		- Is your constant factor reasonable?
	- > `__int128` ?		- Runtime Error:
	- Try `-ftrapv` or `#pragma GCC optimize("trapv")`		- Check memory usage.
	- Floating point errors?		- Forget to clear or destroy stuff?
	- > `long double` ?		- > `vector::shrink_to_fit()`
	- turn off math optimizations		- Stack overflow?
			- 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.insert(5);
10 os.erase(5);
11 os.find_by_order(k); // iterator to k-th element (0-indexed)
12 os.order_of_key(x); // number of elements < x
13 template <typename T>
14 using ordered_multiset_base =
15     tree<pair<T, int>, null_type, less<pair<T, int>>,
16         rb_tree_tag, tree_order_statistics_node_update>;
17 template <typename T> struct ordered_multiset {
18     ordered_multiset_base<T> os;
19     int timer = 0;
20     void insert(T x) { os.insert({x, timer++}); }
21     void erase_one(T x) {
22         auto it = os.lower_bound({x, -1});
23         if (it != os.end() && it->first == x) os.erase(it);
24     }

```

```

25     int order_of_key(T x) { return os.order_of_key({x, -1}); }
26     T find_by_order(int k) {
27         return os.find_by_order(k)->first;
28     }
29     int size() { return (int)os.size(); }
30 };

```

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         pref.assign(n, vll(m, 0));
7         for (int i = 0; i < n; i++) {
8             for (int j = 0; j < m; j++) {
9                 pref[i][j] = v[i][j] + (i ? pref[i - 1][j] : 0) +
10                     (j ? pref[i][j - 1] : 0) -
11                     (i && j ? pref[i - 1][j - 1] : 0);
12         }
13     }
14     ll query(int ulx, int uly, int brx, int bry) const {
15         ll ans = pref[brx][bry];
16         if (ulx) ans -= pref[ulx - 1][bry];
17         if (uly) ans -= pref[brx][uly - 1];
18         if (ulx && uly) ans += pref[ulx - 1][uly - 1];
19         return ans;
20     }

```

```

21    }
21  ll query(int ulx, int uly, int size) const {
22      return query(ulx, uly, ulx + size - 1, uly + size - 1);
23  }
23};

25 struct PartialSum2D : PrefixSum2D {
26     vvl diff; // 0 based
27     int n, m;
28     PartialSum2D(int _n, int _m) : n(_n), m(_m) {
29         diff.assign(n + 1, vll(m + 1, 0));
30     }
31     // add c from [ulx,uly] to [brx,bry]
32     void update(int ulx, int uly, int brx, int bry, ll c) {
33         diff[ulx][uly] += c;
34         diff[ulx][bry + 1] -= c;
35         diff[brx + 1][uly] -= c;
36         diff[brx + 1][bry + 1] += c;
37     }
38     void update(int ulx, int uly, int size, ll c) {
39         int brx = ulx + size - 1, bry = uly + size - 1;
40         update(ulx, uly, brx, bry, c);
41     }
42     void process() {
43         this->build(diff);
44     } // process grid using prefix sum
45 };
45 // usage

```

```

47 PrefixSum2D pref;
48 pref.build(v); // takes 2d 0-based vector as input
49 pref.query(x1, y1, x2, y2); // sum of region
50 PartialSum2D part(n, m); // dimension of grid 0 based
51 part.update(x1, y1, x2, y2, 1); // add 1 in region
52     // must run after all updates
53 part.process(); // prefix sum on diff array
54     // only exists after processing
55 vvl &grid = part.pref; // processed diff array
56 part.query(x1, y1, x2, y2); // gives sum of region

```

2.3. Sparse Table

```

1 // Purpose: Static Range Queries
2 // Idempotent (Min/Max/GCD): O(1)
3 // Non-Idempotent (Sum/Prod/XOR): O(log N)
4 struct SparseTable {
5     vector<vector<long long>> sparse;
6     vector<int> Log;
7     int n, max_log;
8     long long IDENTITY_VAL; // e.g., 0 for Sum, 1 for Product,
9     long long func(long long a, long long b) {
10         return (a + b); // Example: Sum
11     }
12     void build(const vector<long long> &a,
13               long long identity) {
14         n = a.size();
15         IDENTITY_VAL = identity;

```

```

17 max_log = 32 - __builtin_clz(n);
    sparse.assign(n, vector<long long>(max_log));
    Log.assign(n + 1, 0);
19   for (int i = 2; i <= n; ++i) Log[i] = Log[i / 2] + 1;
20   for (int i = 0; i < n; ++i) sparse[i][0] = a[i];
21   for (int j = 1; (1 << j) <= n; ++j) {
22     for (int i = 0; i + (1 << j) <= n; ++i) {
23       sparse[i][j] =
24         func(sparse[i][j - 1],
25               sparse[i + (1 << (j - 1))][j - 1]);
26     }
27   }
28
29 // 3. IDEMPOTENT QUERY O(1) Min, Max, GCD, OR, AND
30 long long query_idempotent(int l, int r) {
31   int k = Log[r - l + 1];
32   return func(sparse[l][k], sparse[r - (1 << k) + 1][k]);
33 }
34
35 // 4. NON-IDEMPOTENT QUERY O(log N)
36 // Use for: Sum, Product, XOR, Matrix Multiplication
37 long long query_non_idempotent(int l, int r) {
38   long long res = IDENTITY_VAL;
39   for (int j = max_log - 1; j >= 0; --j) {
40     if ((1 << j) <= r - l + 1) {
41       // Combine current result with the next block
42       res = func(res, sparse[l][j]);
43       l += (1 << j); // Move L forward by 2^j
44     }
45   }
46   return res;
47 }

```

```

43   }
44 }
45 return res;
46 }
47

```

2.4. Fenwick Tree

```

1 // Interface: 0-based indexing (Internal logic handles
2 // 1-based conversion). for point updates and range query
3 struct FenwickTree {
4   vector<long long> bit;
5   int n;
6   FenwickTree(int n) {
7     this->n = n + 1;
8     bit.assign(n + 1, 0);
9   }
10  FenwickTree(const vector<int> &a)
11    : FenwickTree(a.size()) {
12      for (size_t i = 0; i < a.size(); i++) add(i, a[i]);
13    }
14  long long sum(int idx) {
15    long long ret = 0;
16    for (++idx; idx > 0; idx -= idx & -idx) ret += bit[idx];
17    return ret;
18  }
19  long long sum(int l, int r) {
20    if (l > r) return 0; // Guard clause
21  }

```

```

21    return sum(r) - sum(l - 1);
22 }
23 void add(int idx, int delta) {
24     for (++idx; idx < n; idx += idx & -idx)
25         bit[idx] += delta;
26 }
27 };

```

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
// of inc subseq of length l, that we have seen
5     d[0] = -INF;
6     // user lower bound for non-decreasing
7     for (int i = 0; i < n; i++) {
8         int l =
9             upper_bound(d.begin(), d.end(), a[i]) - d.begin();
10        if (d[l - 1] < a[i] && a[i] < d[l]) d[l] = a[i];
11    }
12    int ans = 0;
13    for (int l = 0; l <= n; l++) {
14        if (d[l] < INF) ans = l;
15    }
16    return ans;
17 }

```

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 }

```

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;
int main() {
5   int n, m;
  cin >> n >> m;
7   e.resize(m);
  d.assign(
9    n, 0); // use 0 to detect neg-cycle in disconnected graph
  par.assign(n, -1);
11  int x = -1;
  for (int i = 0; i <= n; i++) {
13    x = -1;
    for (auto [u, v, w] : e) {
15      if (d[v] > d[u] + w) {
        d[v] = d[u] + w;
        par[v] = u;
        x = v;
19      }
    }
21  }
  if (x == -1) {
23    // no negative cycle

```

```

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

```

3.1.2. Floyd

```

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

```

```

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:
3 // 1. set all edge[i][j] as INF
```

```

// 2. if solve() >= INF, it is not perfect matching.
5
6  typedef long long ll;
7  struct KM {
8    static const int MAXN = 1050;
9    static const ll INF = 1LL << 60;
10   int n, match[MAXN], vx[MAXN], vy[MAXN];
11   ll edge[MAXN][MAXN], lx[MAXN], ly[MAXN], slack[MAXN];
12   void init(int _n) {
13     n = _n;
14     for (int i = 0; i < n; i++)
15       for (int j = 0; j < n; j++) edge[i][j] = 0;
16   }
17   void add_edge(int x, int y, ll w) { edge[x][y] = w; }
18   bool DFS(int x) {
19     vx[x] = 1;
20     for (int y = 0; y < n; y++) {
21       if (vy[y]) continue;
22       if (lx[x] + ly[y] > edge[x][y]) {
23         slack[y] =
24           min(slack[y], lx[x] + ly[y] - edge[x][y]);
25       } else {
26         vy[y] = 1;
27         if (match[y] == -1 || DFS(match[y])) {
28           match[y] = x;
29           return true;
30         }
31       }
32     }
33     return false;
34   }
35   ll solve() {
36     fill(match, match + n, -1);
37     fill(lx, lx + n, -INF);
38     fill(ly, ly + n, 0);
39     for (int i = 0; i < n; i++)
40       for (int j = 0; j < n; j++)
41         lx[i] = max(lx[i], edge[i][j]);
42     for (int i = 0; i < n; i++) {
43       fill(slack, slack + n, INF);
44       while (true) {
45         fill(vx, vx + n, 0);
46         fill(vy, vy + n, 0);
47         if (DFS(i)) break;
48         ll d = INF;
49         for (int j = 0; j < n; j++)
50           if (!vy[j]) d = min(d, slack[j]);
51         for (int j = 0; j < n; j++) {
52           if (vx[j]) lx[j] -= d;
53           if (vy[j]) ly[j] += d;
54           else slack[j] -= d;
55         }
56       }
57     }
58   }
59 
```

```

59    ll res = 0;
60    for (int i = 0; i < n; i++) {
61        res += edge[match[i]][i];
62    }
63    return res;
} 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_)
6         : n(n_), step(0), time(n), low(n), instk(n), e(n) {}
7     void add_edge(int u, int v) { e[u].push_back(v); }
8     void dfs(int x) {
9         time[x] = low[x] = ++step;
10        stk.push_back(x);
11        instk[x] = 1;
12        for (int y : e[x])
13            if (!time[y]) {
14                dfs(y);
15                low[x] = min(low[x], low[y]);
16            } else if (instk[y]) {
17                low[x] = min(low[x], time[y]);
18            }
}

```

```

19    if (time[x] == low[x]) {
20        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 }
void solve() {
29     for (int i = 0; i < n; i++)
30         if (!time[i]) dfs(i);
31     reverse(scc.begin(), scc.end());
32     // scc in topological order
33 }
34 };

```

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++;
}

```

```
9   for (auto v : adj[u]) {
10     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 }
18 vvi tarjanAlgorithm(int n, vvi &edges) {
19   vector<vector<int>> adj(n);
20   for (int i = 0; i < edges.size(); i++) {
21     int u = edges[i][0], v = edges[i][1];
22     adj[u].pb(v), adj[v].pb(u);
23   }
24   vi vis(n, -1), low(n, -1);
25   vector<vector<int>> ans;
26   cnt = 1;
27   dfs(0, -1, adj, vis, low, ans);
28   return ans;
29 }
```

4. Math

4.1. SOS DP

```

1 const long long LOG = 20;
const long long sz = (1 << LOG);
3 void forward1(
    vector<long long> &dp) { // subSet contribution to superset
5   for (int b = 0; b <= LOG; b++)
    for (int i = 0; i <= sz; i++)
        if (i & (1 << b)) dp[i] += dp[i ^ (1 << b)];
}
9 void backward1(vector<long long> &dp) { // undo offorward 1
    for (int b = LOG; b >= 0; b--)
        for (int i = sz; i >= 0; i--)
            if (i & (1 << b)) dp[i] -= dp[i ^ (1 << b)];
}
13 void forward2(
15   vector<long long> &dp) { // superset contributes to subset
    for (int b = 0; b <= LOG; b++)
        for (int i = 0; i <= sz; i++)
            if (i & (1 << b)) dp[i ^ (1 << b)] += dp[i];
}
19 void backward2(vector<long long> &dp) { // undo offorward 2
    for (int b = LOG; b >= 0; b--)
        for (int i = sz; i >= 0; i--)
            if (i & (1 << b)) dp[i ^ (1 << b)] += dp[i];
}
23

```

4.2. Matrix

```

1 // Matrix A(n, n, 1e9+7); // Modular Arithmetic
// Matrix B(n, n); // Standard Arithmetic (mod = 0)
3 // Complexity: Multiplication O(N^3), Power O(N^3 log Exp)
struct Matrix {
5   vector<vector<ll>> mat;
6   int rows, cols;
7   ll mod; // mod = 0 implies Standard Arithmetic (No Modulo)
// 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;
26   }

```

```

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 };

```

4.3. Number Theory

4.3.1. Mod Struct

Requires: Extended GCD

```

3 template <typename T> struct M {
4     static T MOD; // change to constexpr if already known
5     T v;
6     M(T x = 0) {
7         v = (-MOD <= x && x < MOD) ? x : x % MOD;
8         if (v < 0) v += MOD;
9     }
10    explicit operator T() const { return v; }
11    bool operator==(const M &b) const { return v == b.v; }
12    bool operator!=(const M &b) const { return v != b.v; }
13    M operator-() { return M(-v); }
14    M operator+(M b) { return M(v + b.v); }
15    M operator-(M b) { return M(v - b.v); }
16    M operator*(M b) { return M((__int128)v * b.v % MOD); }
17    M operator/(M b) { return *this * (b ^ (MOD - 2)); }
18    // change above implementation to this if MOD is not prime
19    M inv() {
20        auto [p, _, g] = extgcd(v, MOD);
21        return assert(g == 1), p;
22    }
23    friend M operator^(M a, ll b) {
24        M ans(1);
25        for (; b; b >= 1, a *= a)
26            if (b & 1) ans *= a;
27        return ans;
28    }

```

```

29    friend M &operator+=(M &a, M b) { return a = a + b; }
30    friend M &operator-=(M &a, M b) { return a = a - b; }
31    friend M &operator*=(M &a, M b) { return a = a * b; }
32    friend M &operator/=(M &a, M b) { return a = a / b; }
33 };
34 using Mod = M<int>;
35 template <> int Mod::MOD = 1'000'000'007;
36 int &MOD = Mod::MOD;

```

4.3.2. Miller-Rabin

Requires: Mod Struct

```

1
2
3 // checks if Mod::MOD is prime
4 bool is_prime() {
5     if (MOD < 2 || MOD % 2 == 0) return MOD == 2;
6     Mod A[] = {2, 7, 61}; // for int values (< 2^31)
7     // ll: 2, 325, 9375, 28178, 450775, 9780504, 1795265022
8     int s = __builtin_ctzll(MOD - 1), i;
9     for (Mod a : A) {
10         Mod x = a ^ (MOD >> s);
11         for (i = 0; i < s && (x + 1).v > 2; i++) x *= x;
12         if (i && x != -1) return 0;
13     }
14     return 1;
15 }

```

4.3.3. 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
6
7 void sieve() {
8     is_prime.set();
9     is_prime[1] = 0;
10    mu[1] = phi[1] = 1;
11    for (ll i = 2; i < MAXN; i++) {
12        if (is_prime[i]) {
13            mpf[i] = i;
14            primes.push_back(i);
15            phi[i] = i - 1;
16            mu[i] = -1;
17        }
18        for (ll p : primes) {
19            if (p > mpf[i] || i * p >= MAXN) break;
20            is_prime[i * p] = 0;
21            mpf[i * p] = p;
22            mu[i * p] = -mu[i];
23            if (i % p == 0)
24                phi[i * p] = phi[i] * p, mu[i * p] = 0;
25            else phi[i * p] = phi[i] * (p - 1);
26        }
27    }

```

27 | }

4.3.4. Get Factors

Requires: Linear Sieve

```

1
2
3 vector<ll> all_factors(ll n) {
4     vector<ll> fac = {1};
5     while (n > 1) {
6         const ll p = mpf[n];
7         vector<ll> cur = {1};
8         while (n % p == 0) {
9             n /= p;
10            cur.push_back(cur.back() * p);
11        }
12        vector<ll> tmp;
13        for (auto x : fac)
14            for (auto y : cur) tmp.push_back(x * y);
15        tmp.swap(fac);
16    }
17    return fac;
18}

```

4.3.5. 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) {

```

```

5 ll s = 1, t = 0, u = 0, v = 1;
6 while (b) {
7     ll q = a / b;
8     swap(a -= q * b, b);
9     swap(s -= q * t, t);
10    swap(u -= q * v, v);
11 }
12 return {s, u, a};
13 }
```

4.3.6. 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 * y) % (n / g) * m + a;
8     return x < 0 ? x + m / g * n : x;
9 }
```

4.3.7. Pollard Rho

```

1 ll f(ll x, ll mod) { return (x * x + 1) % mod; }
2 // n should be composite
3 ll pollard_rho(ll n) {
```

```

5 if (!(n & 1)) return 2;
6 while (1) {
7     ll y = 2, x = RNG() % (n - 1) + 1, res = 1;
8     for (int sz = 2; res == 1; sz *= 2) {
9         for (int i = 0; i < sz && res <= 1; i++) {
10            x = f(x, n);
11            res = __gcd(abs(x - y), n);
12        }
13        y = x;
14    }
15    if (res != 0 && res != n) return res;
16 }
17 }
```

4.3.8. De Bruijn Sequence

```

1 int res[kN], aux[kN], a[kN], sz;
2 void Rec(int t, int p, int n, int k) {
3     if (t > n) {
4         if (n % p == 0)
5             for (int i = 1; i <= p; ++i) res[sz++] = aux[i];
6         else {
7             aux[t] = aux[t - p];
8             Rec(t + 1, p, n, k);
9             for (aux[t] = aux[t - p] + 1; aux[t] < k; ++aux[t])
10                Rec(t + 1, t, n, k);
11         }
12     }
```

```

13 int DeBruijn(int k, int n) {
14     // return cyclic string of length  $k^n$  such that every
15     // string of length  $n$  using  $k$  character appears as a
16     // substring.
17     if (k == 1) return res[0] = 0, 1;
18     fill(aux, aux + k * n, 0);
19     return sz = 0, Rec(1, 1, n, k), sz;
20 }
21
22 // 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 }
```

4.3.9. Combinatorics

```

1 struct Combinatorics {
2     const int MOD;
3     vector<long long> fact, invFact;
4
5     // Constructor
6     Combinatorics(int maxN, int mod)
7         : MOD(mod), fact(maxN + 1), invFact(maxN + 1) {
8         precompute(maxN);
9     }
10
11    // Function to perform modular exponentiation: a^b % MOD
12    long long modpow(long long a, long long b) const {
13        long long res = 1;
14        while (b) {
15            if (b & 1) res = res * a % MOD;
16            a = a * a % MOD;
17            b >= 1;
18        }
19    }
20
21    // Function to calculate nPk % MOD
22    long long nPk(int n, int k) const {
23        if (k > n || k < 0) return 0;
24        return fact[n] * invFact[k] % MOD * invFact[n - k] %
25            MOD;
26    }
27
28    // Function to calculate nCk % MOD
29    long long nCk(int n, int k) const {
30        if (k > n || k < 0) return 0;
31        invFact[maxN] =
32            modpow(fact[maxN], MOD - 2); // Fermat's Little Theorem
33        for (int i = maxN - 1; i >= 0; i--) {
34            invFact[i] = invFact[i + 1] * (i + 1) % MOD;
35        }
36    }
37
38    // Function to calculate nPk % MOD
39    long long nPk(int n, int k) const {
40        if (k > n || k < 0) return 0;
41    }
42
43    // Function to calculate nCk % MOD
44    long long nCk(int n, int k) const {
45        if (k > n || k < 0) return 0;
46    }

```

```
45     return fact[n] * invFact[n - k] % MOD;
    }
47
// Function to calculate n! % MOD
49 long long factorial(int n) const { return fact[n]; }
50
51 // Combinatorics comb(maxN,mod)
```

5. Geometry

5.1. convex hull

```

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

```

```

25     const Point &C) {
26     // computes (B - A) · (C - A)
27     return (B.x - A.x) * (C.x - A.x) +
28           (B.y - A.y) * (C.y - A.y);
29 }
30 vector<Point> ConvexHullAndrowChain(vector<Point> pts) {
31     sort(pts);
32     pts.erase(unique(pts.begin(), pts.end()), pts.end());
33     int n = pts.size();
34     if (n <= 1) return pts;
35     vector<Point> lower, upper;
36     // Build lower hull
37     for (int i = 0; i < n; ++i) {
38         const Point &p = pts[i];
39         while (lower.size() >= 2 &&
40                cross_product(lower[lower.size() - 2],
41                             lower[lower.size() - 1], p) <= 0) {
42             lower.pop_back();
43         }
44         lower.push_back(p);
45     }
46     // Build upper hull
47     for (int i = n - 1; i >= 0; --i) {
48         const Point &p = pts[i];
49         while (upper.size() >= 2 &&
50                cross_product(upper[upper.size() - 2],
51                             upper[upper.size() - 1], p) <= 0) {
52

```

```
    upper.pop_back();
53 }
  upper.push_back(p);
55 }
vector<Point> hull = lower;
57 for (int i = 1; i + 1 < (int)upper.size(); ++i) {
  hull.push_back(upper[i]);
59 }

61 return hull; // CCW order
}
```

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 int z[n];
2 void manacher(string s) {
3     // z[i] => longest odd palindrome centered at i is
4     //      s[i - z[i] ... i + z[i]]
5     // to get all palindromes (including even length),
6     // insert a '#' between each s[i] and s[i + 1]
7     int n = s.size();
8     z[0] = 0;
9     for (int b = 0, i = 1; i < n; i++) {
10        if (z[b] + b >= i)
11            z[i] = min(z[2 * b - i], b + z[b] - i);
12        else z[i] = 0;
13        while (i + z[i] + 1 < n && i - z[i] - 1 >= 0 &&
14              s[i + z[i] + 1] == s[i - z[i] - 1])
15            z[i]++;
16        if (z[i] + i > z[b] + b) b = i;
17    }
18 }
```

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         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'; // based on problem
21         }
22
23         Trie(int maxChars = 26) {
24             this->maxChars = maxChars;
25             nodes.clear();
26             distWords = 0;
27         } // Create Root Node (Index 0)
28         void insert(string s) {
29             int curr = 0;
30             nodes[curr].pfxCnt++;
31             for (char &ch : s) {
32                 int base = getBase(ch);
33                 if (nodes[curr].next[base] == -1) {
34                     nodes[curr].next[base] = nodes.size();
35                     nodes.emplace_back(maxChars);
36                 }
37                 curr = nodes[curr].next[base];
38                 nodes[curr].pfxCnt++;
39             }
40             if (nodes[curr].wordCnt == 0) {
41                 distWords++; // New distinct word found
42             }
43             nodes[curr].wordCnt++;
44         }
45         bool search(string s) {
46             int curr = 0;
47             for (char &ch : s) {
48                 int base = getBase(ch);
49                 if (nodes[curr].next[base] == -1) {
50                     return false;
51                 }
52                 curr = nodes[curr].next[base];
53             }
54             return nodes[curr].wordCnt != 0;
55         }
56     }
57 }
```

```

55   for (char &ch : s) {
56     int base = getBase(ch);
57     if (nodes[curr].next[base] == -1) return false;
58     curr = nodes[curr].next[base];
59   }
60   return nodes[curr].wordCnt > 0;
61 }
62 // Delete one occurrence of s
63 void erase(string s) {
64   if (!search(s)) return; // Check existence first
65   int curr = 0;
66   nodes[curr].pfxCnt--;
67   for (char &ch : s) {
68     int base = getBase(ch);
69     curr = nodes[curr].next[base];
70     nodes[curr].pfxCnt--;
71   }
72   nodes[curr].wordCnt--;
73   if (nodes[curr].wordCnt == 0)
74     distWords--; // Word completely removed
75 }
76
77 // Count words that have s as a prefix
78 int prefixCount(string s) {
79   int curr = 0;
80   for (char &ch : s) {
81     int base = getBase(ch);

```

```

81     if (nodes[curr].next[base] == -1)
82       return 0; // Prefix not found
83     curr = nodes[curr].next[base];
84   }
85   return nodes[curr].pfxCnt;
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];
6 vector<int> patEnd;
7 int nodes = 1;

9 void build_trie(const vector<string> &P) {
10   // clear
11   for (int i = 0; i < nodes; i++) {
12     memset(nxt[i], 0, sizeof nxt[i]);
13     cntNode[i] = 0;
14     adjSL[i].clear();
15   }
16   nodes = 1;
17   patEnd.clear();
18   patEnd.reserve(P.size());

```

```

19 // insert
20 for (auto &pat : P) {
21     int u = 0;
22     for (char ch : pat) {
23         int c = ch - 'a';
24         if (!nxt[u][c]) nxt[u][c] = nodes++;
25         u = nxt[u][c];
26     }
27     patEnd.pb(u);
28 }
29 }
30 vector<int> bfsOrder;
31 void build_links() {
32     queue<int> q;
33     linkS[0] = 0;
34     //first layer
35     for (int c = 0; c < ALPHA; c++) {
36         int v = nxt[0][c];
37         if (v) {
38             linkS[v] = 0;
39             q.push(v);
40         }
41     }
42     //BFS
43     while (!q.empty()) {
44         int u = q.front();
45         q.pop();
46     }
47     bfsOrder.pb(u);
48     for (int c = 0; c < ALPHA; c++) {
49         int v = nxt[u][c];
50         if (!v) continue;
51         int j = linkS[u];
52         while (j && !nxt[j][c]) j = linkS[j];
53         if (nxt[j][c]) j = nxt[j][c];
54         linkS[v] = j;
55         q.push(v);
56     }
57     for (int u : bfsOrder) { adjSL[linkS[u]].pb(u); }
58 }
59
60 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 }

```