

Contents			
1 Misc	2		
1.1 Contest	2	4.3.1 Mod Struct	12
1.1.1 Macros and debug	2	4.3.2 Miller-Rabin	13
		4.3.3 Linear Sieve	13
		4.3.4 Get Factors	14
		4.3.5 Extended GCD	14
		4.3.6 Chinese Remainder Theorem	14
		4.3.7 Pollard Rho	15
		4.3.8 De Bruijn Sequence	15
		4.3.9 Combinatorics	15
2 Data Structures	3	5 Geometry	17
2.1 GNU PBDS	3	5.1 convex hull	17
2.2 2D Partial Sums	3		
2.3 Sparse Table	4	6 Strings	19
2.4 Fenwick Tree	5	6.1 Knuth-Morris-Pratt Algorithm	19
2.5 Longest Increasing Subsequence	5	6.2 Z Value	19
2.6 Xobriest	6	6.3 Manachers Algorithm	19
		6.4 Trie	20
3 Graph	7	6.5 Aho-Corasick Automaton	21
3.1 Kuhn-Munkres algorithm	7		
3.2 Strongly Connected Components	8		
3.3 Biconnected Components	8		
3.3.1 Articulation Points	8		
4 Math	10		
4.1 SOS DP	10		
4.2 Matrix	10		
4.3 Number Theory	12		

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
3 using namespace __gnu_pbds;
template <typename T>
5 using ordered_set =
    tree<T, null_type, std::less<T>, rb_tree_tag,
7     tree_order_statistics_node_update>;
ordered_set<int> os;
9 os.insert(5);
os.erase(5);
11 // iterator to k-th element (0-indexed)
os.find_by_order(k);
13 // number of elements < x
os.order_of_key(x);

```

2.2. 2D Partial Sums

```

1 struct PrefixSum2D {
    vvl pref;           // 0-based 2-D prefix sum
3     void build(const vvl &v) { // creates a copy
        int n = v.size(), m = v[0].size();
5         pref.assign(n, vll(m, 0));
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                pref[i][j] = v[i][j] + (i ? pref[i - 1][j] : 0) +

```

```

9             (j ? pref[i][j - 1] : 0) -
(i && j ? pref[i - 1][j - 1] : 0);
11         }
13     }
15     ll query(int ulx, int uly, int brx, int bry) const {
        ll ans = pref[brx][bry];
17     if (ulx) ans -= pref[ulx - 1][bry];
19     if (uly) ans -= pref[brx][uly - 1];
21     if (ulx && uly) ans += pref[ulx - 1][uly - 1];
23     return ans;
}
25     ll query(int ulx, int uly, int size) const {
        return query(ulx, uly, ulx + size - 1, uly + size - 1);
}
struct PartialSum2D : PrefixSum2D {
27     vvl diff; // 0 based
29     int n, m;
31     PartialSum2D(int _n, int _m) : n(_n), m(_m) {
        diff.assign(n + 1, vll(m + 1, 0));
}
33     // add c from [ulx,uly] to [brx,bry]
35     void update(int ulx, int uly, int brx, int bry, ll c) {
        diff[ulx][uly] += c;
        diff[ulx][bry + 1] -= c;
        diff[brx + 1][uly] -= c;

```

```

37    diff[brx + 1][bry + 1] += c;
38 }
39 void update(int ulx, int uly, int size, ll c) {
40     int brx = ulx + size - 1, bry = uly + size - 1;
41     update(ulx, uly, brx, bry, c);
42 }
43 // process the grid using prefix sum
44 void process() { this->build(diff); }
45 };
46 // 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
51 PartialSum2D part(n, m); // dimension of grid 0 based
52 part.update(x1, y1, x2, y2, 1); // add 1 in region
53 // must run after all updates
54 part.process(); // prefix sum on diff array
55 // only exists after processing
56 vvl &grid = part.pref; // processed diff array
57 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     // INF for Min
10    long long func(long long a, long long b) {
11        return (a + b); // Example: Sum
12    }
13    void build(const vector<long long> &a,
14               long long identity) {
15        n = a.size();
16        IDENTITY_VAL = identity;
17        max_log = 32 - __builtin_clz(n);
18        sparse.assign(n, vector<long long>(max_log));
19        Log.assign(n + 1, 0);
20        for (int i = 2; i <= n; ++i) Log[i] = Log[i / 2] + 1;
21        for (int i = 0; i < n; ++i) sparse[i][0] = a[i];
22        for (int j = 1; (1 << j) <= n; ++j) {
23            for (int i = 0; i + (1 << j) <= n; ++i) {
24                sparse[i][j] =
25                    func(sparse[i][j - 1],
26                         sparse[i + (1 << (j - 1))][j - 1]);
27            }
28        }
29    }
30 // 3. IDEMPOTENT QUERY O(1)
31 // Use for: Min, Max, GCD, OR, AND

```

```

33 long long query_idempotent(int l, int r) {
34     int k = Log[r - l + 1];
35     return func(sparse[l][k], sparse[r - (1 << k) + 1][k]);
36 }
37 // 4. NON-IDEMPOTENT QUERY O(log N)
38 // Use for: Sum, Product, XOR, Matrix Multiplication
39 // Logic: Decomposes range [l, r] into disjoint power-of-2
40 long long query_non_idempotent(int l, int r) {
41     long long res = IDENTITY_VAL;
42     for (int j = max_log - 1; j >= 0; --j) {
43         if ((1 << j) <= r - l + 1) {
44             // Combine current result with the next block
45             res = func(res, sparse[l][j]);
46             l += (1 << j); // Move L forward by 2^j
47         }
48     }
49     return res;
50 };

```

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     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

```

```

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

```

```

13 } return 1 + splitmix64() % (r - 1 + 1);
15 void solve() {
map<ll, pll> mp;
vector<pll> pref;
17 for (int i = 1; i <= n; i++) {
mp[a[i]] = {randInRange(0LL, (1LL << 64) - 1),
            randInRange(0LL, (1LL << 64) - 1)};
pref[i].ff = pref[i - 1].ff ^ mp[a[i]].ff;
pref[i].ss = pref[i - 1].ss ^ mp[a[i]].ss;
}
23 }

```

2.6. Xobriest

```

1 static uint64_t seed =
    chrono::steady_clock::now().time_since_epoch().count();
3 uint64_t splitmix64() {
    seed += 0x9e3779b97f4a7c15;
5     uint64_t x = seed;
    x = (x ^ (x >> 30)) * 0xbff58476d1ce4e5b9;
7     x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
    return x ^ (x >> 31);
9 }
11 long long randInRange(long long l, long long r) {
    assert(l <= r);
}

```

3. Graph

3.1. Kuhn-Munkres algorithm

```

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

```

```

53     if (vx[j]) lx[j] -= d;
54     if (vy[j]) ly[j] += d;
55     else slack[j] -= d;
56   }
57 }
58 ll res = 0;
59 for (int i = 0; i < n; i++) {
60   res += edge[match[i]][i];
61 }
62 return res;
63 }
64 graph;

```

3.2. 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])

```

```

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 }
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 };

```

3.3. Biconnected Components

3.3.1. Articulation Points

```

1 class Solution {

```

```
int cnt;
int dfs(int u, int p, vvi &adj, vi &vis, vi &low,
       vvi &ans) {
    if (vis[u] != -1) return low[u];
    vis[u] = cnt, low[u] = cnt;
    cnt++;
    for (auto v : adj[u]) {
        if (v == p) continue;
        int temp = dfs(v, u, adj, vis, low, ans);
        low[u] = min(low[u], low[v]);
        if (temp > vis[u]) ans.push_back({u, v});
        else low[u] = min(low[u], vis[v]);
    }
    return low[u];
}
vvi tarjanAlgorithm(int n, vvi &edges) {
    vector<vector<int>> adj(n);
    for (int i = 0; i < edges.size(); i++) {
        int u = edges[i][0], v = edges[i][1];
        adj[u].pb(v), adj[v].pb(u);
    }
    vi vis(n, -1), low(n, -1);
    vector<vector<int>> ans;
    cnt = 1;
    dfs(0, -1, adj, vis, low, ans);
    return ans;
}
```

29};

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 of forward 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 of forward 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];
24 }
```

4.2. Matrix

```

1 /**
2   * Generic Matrix Template
3   * * Purpose: Matrix Exponentiation and Operations
4   * * Usage:
5   * Matrix A(n, n, 1e9+7); // Modular Arithmetic
6   * Matrix B(n, n);      // Standard Arithmetic (mod = 0)
7   * * Complexity: Multiplication O(N^3), Power O(N^3 log Exp)
8   */
9 struct Matrix {
10   using ll = long long;
11   vector<vector<ll>> mat;
12   int rows, cols;
13   ll mod; // mod = 0 implies Standard Arithmetic (No Modulo)
14
15   // Constructor: Default mod is 0 (No Mod)
16   Matrix(int r, int c, ll m = 0)
17     : rows(r), cols(c), mod(m) {
18       mat.assign(rows, vector<ll>(cols, 0));
19     }
20
21   // Input Matrix
22   void input() {
23     for (int i = 0; i < rows; ++i)
24       for (int j = 0; j < cols; ++j) cin >> mat[i][j];
25   }
26 }
```

```

27 // Print Matrix
28 void print() const {
29     for (const auto &row : mat) {
30         for (const auto &val : row) cout << val << " ";
31         cout << endl;
32     }
33 }
34
35 // Addition
36 Matrix operator+(const Matrix &other) const {
37     Matrix result(rows, cols, mod);
38     for (int i = 0; i < rows; ++i) {
39         for (int j = 0; j < cols; ++j) {
40             result.mat[i][j] = mat[i][j] + other.mat[i][j];
41             if (mod) result.mat[i][j] %= mod;
42         }
43     }
44     return result;
45 }
46
47 // Subtraction
48 Matrix operator-(const Matrix &other) const {
49     Matrix result(rows, cols, mod);
50     for (int i = 0; i < rows; ++i) {
51         for (int j = 0; j < cols; ++j) {
52             result.mat[i][j] = mat[i][j] - other.mat[i][j];
53             if (mod)
54                 result.mat[i][j] =
55                     (result.mat[i][j] % mod + mod) % mod;
56         }
57     }
58     return result;
59 }
60
61 // Multiplication O(N^3)
62 Matrix operator*(const Matrix &other) const {
63     // Assert matching dimensions if needed: assert(cols ==
64     // other.rows);
65     Matrix result(rows, other.cols, mod);
66     for (int i = 0; i < rows; ++i) {
67         for (int k = 0; k < cols;
68              ++k) { // Optimized loop order (i-k-j is cache
69                  // friendly)
70                 if (mat[i][k] == 0)
71                     continue; // Optimization for sparse matrices
72                 for (int j = 0; j < other.cols; ++j) {
73                     if (mod) {
74                         result.mat[i][j] =
75                             (result.mat[i][j] +
76                             mat[i][k] * other.mat[k][j]) %
77                             mod;
78                     } else {
79                         result.mat[i][j] += mat[i][k] * other.mat[k][j];
80                     }
81                 }
82             }
83         }
84     }
85     return result;
86 }

```

```

81      }
82  }
83  return result;
84 }
85

// Matrix Exponentiation O(N^3 log Exp)
86 Matrix power(ll exp) const {
87     // Identity Matrix
88     Matrix result(rows, cols, mod);
89     for (int i = 0; i < rows; ++i) result.mat[i][i] = 1;
90
91     Matrix base = *this;
92     while (exp > 0) {
93         if (exp & 1) result = result * base;
94         base = base * base;
95         exp >>= 1;
96     }
97     return result;
98 }
99
100 };
101

```

4.3. Number Theory

4.3.1. Mod Struct

A list of safe primes: 26003, 27767, 28319, 28979, 29243, 29759, 30467
 910927547, 919012223, 947326223, 990669467, 1007939579, 1019126699

929760389146037459, 975500632317046523, 989312547895528379	NTT prime p	$p - 1$	primitive root
65537	$1 \ll 16$	3	
998244353	$119 \ll 23$	3	
2748779069441	$5 \ll 39$	3	
1945555039024054273	$27 \ll 56$	5	

Requires: Extended GCD

```

1
2
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)); }

```

```

19 // change above implementation to this if MOD is not prime
M inv() {
    auto [p, _, g] = extgcd(v, MOD);
    return assert(g == 1), p;
}
21 friend M operator^(M a, ll b) {
    M ans(1);
    for (; b; b >= 1, a *= a)
        if (b & 1) ans *= a;
    return ans;
}
23 friend M &operator+=(M &a, M b) { return a = a + b; }
friend M &operator-=(M &a, M b) { return a = a - b; }
31 friend M &operator*=(M &a, M b) { return a = a * b; }
friend M &operator/=(M &a, M b) { return a = a / b; }
33 };
using Mod = M<int>;
35 template <> int Mod::MOD = 1'000'000'007;
int &MOD = Mod::MOD;

```

4.3.2. Miller-Rabin

Requires: Mod Struct

```

1
3 // checks if Mod::MOD is prime
bool is_prime() {
5 if (MOD < 2 || MOD % 2 == 0) return MOD == 2;

```

```

7 Mod A[] = {2, 7, 61}; // for int values (< 2^31)
// ll: 2, 325, 9375, 28178, 450775, 9780504, 1795265022
9 int s = __builtin_ctzll(MOD - 1), i;
for (Mod a : A) {
    Mod x = a ^ (MOD >> s);
    for (i = 0; i < s && (x + 1).v > 2; i++) x *= x;
    if (i && x != -1) return 0;
}
return 1;
15

```

4.3.3. Linear Sieve

```

1 constexpr ll MAXN = 1000000;
bitset<MAXN> is_prime;
3 vector<ll> primes;
5 ll mpf[MAXN], phi[MAXN], mu[MAXN];
void sieve() {
7 is_prime.set();
is_prime[1] = 0;
9 mu[1] = phi[1] = 1;
for (ll i = 2; i < MAXN; i++) {
11 if (is_prime[i]) {
    mpf[i] = i;
    primes.push_back(i);
    phi[i] = i - 1;
    mu[i] = -1;
13
15

```

```

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 }

```

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) {
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};
12 }

```

4.3.6. Chinese Remainder Theorem

Requires: Extended GCD

```

1
2 // for 0 <= a < m, 0 <= b < n, returns the smallest x >= 0
3 // such that x % m == a and x % n == b
4 ll crt(ll a, ll m, ll b, ll n) {

```

```

5 | if (n > m) swap(a, b), swap(m, n);
6 | auto [x, y, g] = extgcd(m, n);
7 | assert((a - b) % g == 0); // no solution
8 | x = ((b - a) / g * x) % (n / g) * m + a;
9 | return x < 0 ? x + m / g * n : x;
10|

```

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

```

4.3.8. De Bruijn Sequence

```

1 | int res[kN], aux[kN], a[kN], sz;

```

```

3 | void Rec(int t, int p, int n, int k) {
4 |   if (t > n) {
5 |     if (n % p == 0)
6 |       for (int i = 1; i <= p; ++i) res[sz++] = aux[i];
7 |     else {
8 |       aux[t] = aux[t - p];
9 |       Rec(t + 1, p, n, k);
10|       for (aux[t] = aux[t - p] + 1; aux[t] < k; ++aux[t])
11|         Rec(t + 1, t, n, k);
12|     }
13|   }
14|   int DeBruijn(int k, int n) {
15|     // return cyclic string of length k^n such that every
16|     // string of length n using k character appears as a
17|     // substring.
18|     if (k == 1) return res[0] = 0, 1;
19|     fill(aux, aux + k * n, 0);
20|     return sz = 0, Rec(1, 1, n, k), sz;
21|

```

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) {
9     precompute(maxN);
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     return res;
20   }
21 // Precomputing factorials and modular inverses
22   void precompute(int maxN) {
23     fact[0] = 1;
24     for (int i = 1; i <= maxN; i++) {
25       fact[i] = fact[i - 1] * i % MOD;
26     }
27     invFact[maxN] =
28     modpow(fact[maxN], MOD - 2); // Fermat's Little Theorem
29     for (int i = maxN - 1; i >= 0; i--) {
30       invFact[i] = invFact[i + 1] * (i + 1) % MOD;
31     }
32   }
33 }

35 // Function to calculate nCk % MOD
36   long long nCk(int n, int k) const {
37     if (k > n || k < 0) return 0;
38     return fact[n] * invFact[k] % MOD * invFact[n - k] %
39           MOD;
40   }
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   }
46 // Function to calculate n! % MOD
47   long long factorial(int n) const { return fact[n]; }
48 };
49 // Combinatorics comb(maxN,mod)
50
51

```

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 }
```

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    }
```

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 }

```