
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 Miller-Rabin	14
2.2 2D Partial Sums	3	4.3.2 Linear Sieve	15
2.3 Sparse Table	4	4.3.3 Get Factors	16
2.4 Fenwick Tree	5	4.3.4 Extended GCD	16
2.5 Longest Increasing Subsequence	6	4.3.5 Chinese Remainder Theorem	16
2.6 Xobriest	6	4.3.6 Pollard Rho	17
3 Graph	7	4.3.7 De Bruijn Sequence	17
3.1 General	7	4.3.8 Combinatorics	17
3.1.1 Bellman	7	5 Geometry	19
3.1.2 Floyd	7	5.1 convex hull	19
3.1.3 DSU	8	6 Strings	21
3.1.4 Binary Lifting	8	6.1 Knuth-Morris-Pratt Algorithm	21
3.2 Kuhn-Munkres algorithm	9	6.2 Z Value	21
3.3 Strongly Connected Components	11	6.3 Manachers Algorithm	21
3.4 Biconnected Components	11	6.4 Trie	22
3.4.1 Articulation Points	11	6.5 Aho-Corasick Automaton	23

1. Misc

1.1. Contest

1.1.1. Macros and debug

```
1 #pragma GCC optimize("O3", "unroll-loops")
```

- 1 - Pre-submit:
 - 3 - Did you make a typo when copying a template?
 - 3 - Test more cases if unsure.
 - 5 - Write a naive solution and check small cases.
 - 5 - Submit the correct file.
- 7 - General Debugging:
 - 9 - Read the whole problem again.
 - 9 - Have a teammate read the problem.
 - 9 - Have a teammate read your code.
 - 11 - Explain your solution to them (or a rubber duck).
 - 11 - Print the code and its output / debug output.
 - 13 - Go to the toilet.
- 15 - Wrong Answer:
 - 17 - Any possible overflows?
 - 17 - > `__int128` ?
 - 17 - Try `-ftrapv` or `#pragma GCC optimize("trapv")`
 - 19 - Floating point errors?
 - 21 - > `long double` ?
 - 21 - turn off math optimizations

- 23 - check for `==`, `>=`, `acos(1.000000001)`, etc.
- 23 - Did you forget to sort or unique?
- 23 - Generate large and worst "corner" cases.
- 25 - Check your `m` / `n`, `i` / `j` and `x` / `y`.
- 25 - Are everything initialized or reset properly?
- 27 - Are you sure about the STL thing you are using?
 - 27 - Read cppreference (should be available).
- 29 - Print everything and run it on pen and paper.
- 31 - Time Limit Exceeded:
 - 31 - Calculate your time complexity again.
 - 33 - Does the program actually end?
 - 33 - Check for `while(q.size())` etc.
 - 35 - Test the largest cases locally.
 - 35 - Did you do unnecessary stuff?
 - 37 - e.g. pass vectors by value
 - 37 - e.g. `memset` for every test case
 - 39 - Is your constant factor reasonable?
- 39 - Runtime Error:
 - 41 - Check memory usage.
 - 41 - Forget to clear or destroy stuff?
 - 43 - > `vector::shrink_to_fit()`
 - 43 - Stack overflow?
 - 45 - Bad pointer / array access?
 - 45 - Division by zero? NaN's?

2. Data Structures

2.1. GNU PBDS

```

1 // #include <ext/pb_ds/assoc_container.hpp>
  // #include <ext/pb_ds/tree_policy.hpp>
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 os.find_by_order(k); // iterator to k-th element (0-indexed)
  os.order_of_key(x); // number of elements < x
13 template <typename T>
  using ordered_multiset_base =
15 tree<pair<T, int>, null_type, less<pair<T, int>>,
   rb_tree_tag, tree_order_statistics_node_update>;
17 template <typename T> struct ordered_multiset {
  ordered_multiset_base<T> os;
19 int timer = 0;
  void insert(T x) { os.insert({x, timer++}); }
21 void erase_one(T x) {
  auto it = os.lower_bound({x, -1});
23 if (it != os.end() && it->first == x) os.erase(it);
  }

```

```

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

```

2.2. 2D Partial Sums

```

1 struct PrefixSum2D {
  vll pref; // 0-based 2-D prefix sum
3 void build(const vll &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++) {
7     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 }
  ll query(int ulx, int uly, int brx, int bry) const {
15   ll ans = pref[brx][bry];
  if (ulx) ans -= pref[ulx - 1][bry];
17   if (uly) ans -= pref[brx][uly - 1];
  if (ulx && uly) ans += pref[ulx - 1][uly - 1];
19   return ans;
  }

```

```

}
21 ll query(int ulx, int uly, int size) const {
    return query(ulx, uly, ulx + size - 1, uly + size - 1);
23 }
};
25 struct PartialSum2D : PrefixSum2D {
    vll diff; // 0 based
27 int n, m;
    PartialSum2D(int _n, int _m) : n(_n), m(_m) {
29     diff.assign(n + 1, vll(m + 1, 0));
    }
31 // add c from [ulx,uly] to [brx,bry]
    void update(int ulx, int uly, int brx, int bry, ll c) {
33     diff[ulx][uly] += c;
        diff[ulx][bry + 1] -= c;
35     diff[brx + 1][uly] -= c;
        diff[brx + 1][bry + 1] += c;
37 }
    void update(int ulx, int uly, int size, ll c) {
39     int brx = ulx + size - 1, bry = uly + size - 1;
        update(ulx, uly, brx, bry, c);
41 }
    void process() {
43     this->build(diff);
    } // process grid using prefix sum
45 };
// usage

```

```

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

```

2.3. Sparse Table

```

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

```

```

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

```

```

43     }
    }
45     return res;
    }
47 };

```

2.4. Fenwick Tree

```

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

```

```

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

```

2.5. Longest Increasing Subsequence

```

1 int lis(vector<int> const &a) {
   int n = a.size();
3   const int INF = 1e9;
   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;
}

```

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)) * 0xbf58476d1ce4e5b9;
7   x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
   return x ^ (x >> 31);
9 }
10 long long randInRange(long long l, long long r) {
11   assert(l <= r);
   return l + splitmix64() % (r - l + 1);
13 }
14 void solve() {
15   map<ll, pll> mp;
   vector<pll> pref;
17   for (int i = 1; i <= n; i++) {
       mp[a[i]] = {randInRange(0LL, (1LL << 64) - 1),
19         randInRange(0LL, (1LL << 64) - 1)};
       pref[i].ff = pref[i - 1].ff ^ mp[a[i]].ff;
21       pref[i].ss = pref[i - 1].ss ^ mp[a[i]].ss;
   }
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;
17                 par[v] = u;
                 x = v;
19             }
           }
21    }
       if (x == -1) {
23        // no negative cycle

```

```

   } else {
25     // negative cycle exists
       for (int i = 0; i < n; i++) x = par[x];
27     vector<int> cyc;
       for (int v = x; v = par[v]) {
29         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++) {
13        d[i][i] = 0;
           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;

```



```

jump[root][0] = parent;
7  g[root][0] = val[root];
  for (int i = 1; i < 20;
9    i++) { // always start from 1, since i depends on i-1
    if (jump[root][i - 1] != -1) {
11      jump[root][i] = jump[jump[root][i - 1]][i - 1];
      g[root][i] =
13      gcd(g[root][i - 1],
          g[jump[root][i - 1]][i - 1]); // combination logic
15    } else {
      jump[root][i] = -1;
17      g[root][i] =
      g[root]
19      [i - 1]; // when the i-1th ancestor doesn't exist,
          // then store same aggregate as i-1th
21    }
  }
23 // do normal DFS from here
}

25 int path_gcd(int u, int v) {
  if (depth[u] < depth[v]) swap(u, v);
27  int GCD = 0; // always use the identity value
  for (int i = 19; i >= 0; i--) {
29    if (((depth[u] - depth[v]) >> i) & 1) {
      {
31      GCD = gcd(g[u][i], GCD);
      u = jump[u][i];

```

```

33    }
    }
35  }
  if (u == v)
37    return gcd(
      GCD,
39      val[u]); // g[u][i], doesn't contain val at jump[u][i]
  for (int i = 19; i >= 0; i--) {
41    if (jump[u][i] != jump[v][i]) {
      GCD = gcd(GCD, g[u][i]);
43      GCD = gcd(GCD, g[v][i]);
      u = jump[u][i];
45      v = jump[v][i];
    }
47  }
  // since both u and v are immediate child of lca
49  // neither u, v nor lca is included in the computation
  // add them explicitly
51  GCD = gcd(GCD, val[u]);
  GCD = gcd(GCD, val[v]);
53  return gcd(GCD, val[jump[u][0]]);
}

```

3.2. Kuhn-Munkres algorithm

```

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

```

```

// 2. if solve() >= INF, it is not perfect matching.
5
typedef long long ll;
7 struct KM {
    static const int MAXN = 1050;
9    static const ll INF = 1LL << 60;
    int n, match[MAXN], vx[MAXN], vy[MAXN];
11    ll edge[MAXN][MAXN], lx[MAXN], ly[MAXN], slack[MAXN];
    void init(int _n) {
13        n = _n;
        for (int i = 0; i < n; i++)
15            for (int j = 0; j < n; j++) edge[i][j] = 0;
    }
17    void add_edge(int x, int y, ll w) { edge[x][y] = w; }
    bool DFS(int x) {
19        vx[x] = 1;
        for (int y = 0; y < n; y++) {
21            if (vy[y]) continue;
            if (lx[x] + ly[y] > edge[x][y]) {
23                slack[y] =
                min(slack[y], lx[x] + ly[y] - edge[x][y]);
25            } else {
                vy[y] = 1;
27                if (match[y] == -1 || DFS(match[y])) {
                    match[y] = x;
29                    return true;
                }
            }
        }
31    }
    }
33    return false;
    }
35    ll solve() {
        fill(match, match + n, -1);
37        fill(lx, lx + n, -INF);
        fill(ly, ly + n, 0);
39        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
41                lx[i] = max(lx[i], edge[i][j]);
        for (int i = 0; i < n; i++) {
43            fill(slack, slack + n, INF);
            while (true) {
45                fill(vx, vx + n, 0);
                fill(vy, vy + n, 0);
47                if (DFS(i)) break;
                ll d = INF;
49                for (int j = 0; j < n; j++)
                    if (!vy[j]) d = min(d, slack[j]);
51                for (int j = 0; j < n; j++) {
                    if (vx[j]) lx[j] -= d;
53                    if (vy[j]) ly[j] += d;
                    else slack[j] -= d;
55                }
            }
57        }
    }

```

```

    ll res = 0;
59  for (int i = 0; i < n; i++) {
    res += edge[match[i]][i];
61  }
    return res;
63 }
} graph;

```

3.3. Strongly Connected Components

```

1 struct TarjanScc {
    int n, step;
3  vector<int> time, low, instk, stk;
    vector<vector<int>>> e, scc;
5  TarjanScc(int n_)
    : 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); }
    void dfs(int x) {
9      time[x] = low[x] = ++step;
        stk.push_back(x);
11     instk[x] = 1;
        for (int y : e[x])
13         if (!time[y]) {
            dfs(y);
15         low[x] = min(low[x], low[y]);
        } else if (instk[y]) {
17         low[x] = min(low[x], time[y]);
        }
    }

```

```

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

```

3.4. Biconnected Components

3.4.1. Articulation Points

```

1 class Solution {
    int cnt;
3  int dfs(int u, int p, vvi &adj, vi &vis, vi &low,
        vvi &ans) {
5      if (vis[u] != -1) return low[u];
        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++)
7       if (i & (1 << b)) dp[i] += dp[i ^ (1 << b)];
   }
9  void backward1(vector<long long> &dp) { // undo of forward 1
   for (int b = LOG; b >= 0; b--)
11    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++)
17    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 of forward 2
21    for (int b = LOG; b >= 0; b--)
        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
   // 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;
       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)
       : rows(r), cols(c), mod(m) {
11    mat.assign(rows, vector<ll>(cols, 0));
   }
13  void input() {
       for (int i = 0; i < rows; ++i)
15         for (int j = 0; j < cols; ++j) cin >> mat[i][j];
   }
17  Matrix operator+(const Matrix &other) const {
       Matrix result(rows, cols, mod);
19     for (int i = 0; i < rows; ++i) {
         for (int j = 0; j < cols; ++j) {
21         result.mat[i][j] = mat[i][j] + other.mat[i][j];
           if (mod) result.mat[i][j] %= mod;
23         }
       }
25     return result;
   }

```

```

27 Matrix operator-(const Matrix &other) const {
    Matrix result(rows, cols, mod);
29   for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
31         result.mat[i][j] = mat[i][j] - other.mat[i][j];
            if (mod)
33             result.mat[i][j] =
                (result.mat[i][j] % mod + mod) % mod;
35         }
    }
37   return result;
}
39 // Multiplication  $O(N^3)$ 
Matrix operator*(const Matrix &other) const {
41   Matrix result(rows, other.cols, mod);
    for (int i = 0; i < rows; ++i) {
43       for (int k = 0; k < cols;
           ++k) { // Optimized loop order (i-k-j is cache
45                 // friendly)
            if (mat[i][k] == 0)
47                 continue; // Optimization for sparse matrices
            for (int j = 0; j < other.cols; ++j) {
49                 if (mod) {
                    result.mat[i][j] =
51                     (result.mat[i][j] +
                     mat[i][k] * other.mat[k][j]) %
53                     mod;

```

```

        } else {
55         result.mat[i][j] += mat[i][k] * other.mat[k][j];
        }
57     }
    }
59   return result;
61 }
    // Matrix Exponentiation  $O(N^3 \log \text{Exp})$ 
63 Matrix power(ll exp) const {
    // Identity Matrix
65   Matrix result(rows, cols, mod);
    for (int i = 0; i < rows; ++i) result.mat[i][i] = 1;
67   Matrix base = *this;
    while (exp > 0) {
69       if (exp & 1) result = result * base;
        base = base * base;
71       exp >>= 1;
    }
73   return result;
    }
75 };

```

4.3. Number Theory

4.3.1. Miller-Rabin

```

1 using u128 = __uint128_t;
    using ull = unsigned long long;

```

```

3 ull mod_mul(ull a, ull b, ull mod) {
    return (u128)a * b % mod;
5 }
ull mod_pow(ull a, ull d, ull mod) {
7     ull res = 1;
    while (d) {
9         if (d & 1) res = mod_mul(res, a, mod);
            a = mod_mul(a, a, mod);
11        d >>= 1;
    }
13    return res;
}
15 bool isPrime(ull n) {
    if (n < 2) return false;
17    for (ull p : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37})
        if (n % p == 0) return n == p;
19    ull d = n - 1;
    int s = __builtin_ctzll(d);
21    d >>= s;
    auto check = [&](ull a) {
23        ull x = mod_pow(a, d, n);
        if (x == 1 || x == n - 1) return true;
25        for (int r = 1; r < s; r++) {
            x = mod_mul(x, x, n);
27            if (x == n - 1) return true;
        }
29    return false;

```

```

    };
31    // Proven deterministic bases for 64-bit
    // {2, 7, 61} for 32 bits
33    for (ull a :
        {2, 325, 9375, 28178, 450775, 9780504, 1795265022})
35        if (a < n && !check(a)) return false;
    return true;
37 }

```

4.3.2. Linear Sieve

```

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

```

```

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

```

4.3.3. Get Factors

```

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

```

4.3.4. Extended GCD

```

1  // returns (p, q, g): p * a + q * b == g == gcd(a, b)
   // g is not guaranteed to be positive when a < 0 or b < 0
3  tuple<ll, ll, ll> extgcd(ll a, ll b) {
    ll s = 1, t = 0, u = 0, v = 1;
5  while (b) {
    ll q = a / b;
7  swap(a -= q * b, b);
    swap(s -= q * t, t);
9  swap(u -= q * v, v);
    }
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
   // such that x % m == a and x % n == b
3  ll crt(ll a, ll m, ll b, ll n) {
5  if (n > m) swap(a, b), swap(m, n);
    auto [x, y, g] = extgcd(m, n);
7  assert((a - b) % g == 0); // no solution
    x = ((b - a) / g * x) % (n / g) * m + a;
9  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;
5  while (1) {
    ll y = 2, x = RNG() % (n - 1) + 1, res = 1;
7    for (int sz = 2; res == 1; sz *= 2) {
      for (int i = 0; i < sz && res <= 1; i++) {
9        x = f(x, n);
        res = __gcd(abs(x - y), n);
11       }
      y = x;
13     }
    if (res != 0 && res != n) return res;
15  }
}

```

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)
5        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 int DeBruijn(int k, int n) {
  // return cyclic string of length k^n such that every
15  // string of length n using k character appears as a
  // substring.
17  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)
7    : MOD(mod), fact(maxN + 1), invFact(maxN + 1) {
    precompute(maxN);
9  }

11  // Function to perform modular exponentiation: a^b % MOD
  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 }
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     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 }
34 // 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 }
46 // Function to calculate n! % MOD
47 long long factorial(int n) const { return fact[n]; }
48 };
49 // 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  $A \rightarrow B \rightarrow C$ . If  $cross > 0 \rightarrow$  left turn (counter-clockwise) If  $cross < 0 \rightarrow$  right turn (clockwise) If  $cross = 0 \rightarrow$  collinear
17     */
18    return (B.x - A.x) * (C.y - A.y) -
19           (B.y - A.y) * (C.x - A.x);
20 }
21 long long dot_product(const Point &A, const Point &B,
```

```

22                 const Point &C) {
23     // computes  $(B - A) \cdot (C - A)$ 
24     return (B.x - A.x) * (C.x - A.x) +
25           (B.y - A.y) * (C.y - A.y);
26 }
27 vector<Point> ConvexHullAndrowChain(vector<Point> pts) {
28     sort(pts);
29     pts.erase(unique(pts.begin(), pts.end()), pts.end());
30     int n = pts.size();
31     if (n <= 1) return pts;
32     vector<Point> lower, upper;
33     // Build lower hull
34     for (int i = 0; i < n; ++i) {
35         const Point &p = pts[i];
36         while (lower.size() >= 2 &&
37              cross_product(lower[lower.size() - 2],
38                          lower[lower.size() - 1], p) <= 0) {
39             lower.pop_back();
40         }
41         lower.push_back(p);
42     }
43     // Build upper hull
44     for (int i = n - 1; i >= 0; --i) {
45         const Point &p = pts[i];
46         while (upper.size() >= 2 &&
47              cross_product(upper[upper.size() - 2],
48                          upper[upper.size() - 1], p) <= 0) {
49             upper.pop_back();
50         }
51         upper.push_back(p);
52     }
53     // Remove the last point of each list (it's the first of the other)
54     lower.pop_back();
55     upper.pop_back();
56     lower.insert(lower.end(), upper.begin(), upper.end());
57     return lower;
58 }
```

```
    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) {
  vector<int> p(s.size());
3  for (int i = 1; i < s.size(); i++) {
    int g = p[i - 1];
5    while (g && s[i] != s[g]) g = p[g - 1];
    p[i] = g + (s[i] == s[g]);
7  }
  return p;
9 }

vector<int> match(const string &s, const string &pat) {
11 vector<int> p = pi(pat + '\0' + s), res;
  for (int i = p.size() - s.size(); i < p.size(); i++)
13   if (p[i] == pat.size())
    res.push_back(i - 2 * pat.size());
15 return res;
}

```

6.2. Z Value

```

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

```

```

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

```

6.3. Manachers Algorithm

```

1 int z[n];
void manacher(string s) {
3  // z[i] => longest odd palindrome centered at i is
  //      s[i - z[i] ... i + z[i]]
5  // to get all palindromes (including even length),
  // insert a '#' between each s[i] and s[i + 1]
7  int n = s.size();
  z[0] = 0;
9  for (int b = 0, i = 1; i < n; i++) {
    if (z[b] + b >= i)
11     z[i] = min(z[2 * b - i], b + z[b] - i);
    else z[i] = 0;
13    while (i + z[i] + 1 < n && i - z[i] - 1 >= 0 &&
      s[i + z[i] + 1] == s[i - z[i] - 1])
15     z[i]++;
    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 - 'Change'; // 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         nodes.emplace_back(maxChars);
29     }
30
31     // Insert string s into Trie
32     void insert(string s) {
33         int curr = 0;
34         nodes[curr].pfxCnt++;
35         for (char &ch : s) {
36             int base = getBase(ch);
37             // If path doesn't exist, create new node
38             if (nodes[curr].next[base] == -1) {
39                 nodes[curr].next[base] = nodes.size();
40                 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     }
50
51     // Check if string s exists
52     bool search(string s) {
53         int curr = 0;

```

```

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

```

```

81  if (nodes[curr].next[base] == -1)
    return 0; // Prefix not found
83  curr = nodes[curr].next[base];
    }
85  return nodes[curr].pfxCnt;
    }
87  };

```

6.5. Aho-Corasick Automaton

```

1  const int ALPHA = 26, MAXNODES = 500000 + 5;
    int nxt[MAXNODES][ALPHA];
3  int linkS[MAXNODES];
    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]);
13  cntNode[i] = 0;
        adjSL[i].clear();
15  }
        nodes = 1;
17  patEnd.clear();
        patEnd.reserve(P.size());

```

```

19 // insert
   for (auto &pat : P) {
21     int u = 0;
     for (char ch : pat) {
23       int c = ch - 'a';
       if (!nxt[u][c]) nxt[u][c] = nodes++;
25       u = nxt[u][c];
     }
27     patEnd.pb(u);
   }
29 }
vector<int> bfsOrder;
31 void build_links() {
   queue<int> q;
33   linkS[0] = 0;
   // first layer
35   for (int c = 0; c < ALPHA; c++) {
     int v = nxt[0][c];
37     if (v) {
       linkS[v] = 0;
39       q.push(v);
     }
41   }
   // BFS
43   while (!q.empty()) {
     int u = q.front();
45     q.pop();

```

```

   bfsOrder.pb(u);
47   for (int c = 0; c < ALPHA; c++) {
     int v = nxt[u][c];
49     if (!v) continue;
     int j = linkS[u];
51     while (j && !nxt[j][c]) j = linkS[j];
     if (nxt[j][c]) j = nxt[j][c];
53     linkS[v] = j;
     q.push(v);
55   }
   }
57   for (int u : bfsOrder) { adjSL[linkS[u]].pb(u); }
   }
59   void solve() {
61     string S;
     ll k;
63     cin >> S >> k;
     vector<string> P(k);
65     for (int i = 0; i < k; i++) cin >> P[i];
     build_trie(P);
67     bfsOrder.clear();
     build_links();
69   }

```