Contents

1. Misc

1.1. Contest

1.1.1. Makefile

```
.PRECIOUS: ./p%

%: p%
  ulimit -s unlimited && ./$<
p%: p%.cpp
  g++ -o $@ $< -std=c++17 -Wall -Wextra -Wshadow \
    -fsanitize=address,undefined
```

1.2. How Did We Get Here?

1.2.1. Macros

Use vectorizations and math optimizations at your own peril.
For gcc≥9, there are [[likely]] and [[unlikely]] attributes.
Call gcc with -fopt-info-optimized-missed-optall for optimization info.

```
#define _GLIBCXX_DEBUG          1 // for debug mode
#define _GLIBCXX_SANITIZE_VECTOR 1 // for asan on vectors
#pragma GCC optimize("O3", "unroll-loops")
#pragma GCC optimize("fast-math")
#pragma GCC target("avx,avx2,abm,bmi,bmi2") // tip: `lscpu`
// before a loop
#pragma GCC unroll 16 // 0 or 1 -> no unrolling
#pragma GCC ivdep
```

2. Data Structures

2.1. GNU PBDS

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/priority_queue.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

// most std::map + order_of_key, find_by_order, split, join
template <typename T, typename U = null_type>
using ordered_map = tree<T, U, std::less<>, rb_tree_tag,
                         tree_order_statistics_node_update>;
// useful tags: rb_tree_tag, splay_tree_tag

template <typename T> struct myhash {
  size_t operator()(T x) const; // splitmix, bswap(x*R), ...
};
// most of std::unordered_map, but faster (needs good hash)
template <typename T, typename U = null_type>
using hash_table = gp_hash_table<T, U, myhash<T>>;

// most std::priority_queue + modify, erase, split, join
using heap = priority_queue<int, std::less<>>;
// useful tags: pairing_heap_tag, binary_heap_tag,
//              (rc_)?binomial_heap_tag, thin_heap_tag
```

2.2. 2D Partial Sums

```
using vvi = vector<vector<int>>;
using vvll = vector<vector<ll>>;
using vll = vector<ll>;

struct PrefixSum2D {
  vvll pref;              // 0-based 2-D prefix sum
  void build(const vvll &v) { // creates a copy
    int n = v.size(), m = v[0].size();
    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) +
                     (j ? pref[i][j - 1] : 0) -
                     (i && j ? pref[i - 1][j - 1] : 0);
      }
    }
  }
  ll query(int ulx, int uly, int brx, int bry) const {
    ll ans = pref[brx][bry];
    if (ulx) ans -= pref[ulx - 1][bry];
    if (uly) ans -= pref[brx][uly - 1];
    if (ulx && uly) ans += pref[ulx - 1][uly - 1];
    return ans;
  }
  ll query(int ulx, int uly, int size) const {
    return query(ulx, uly, ulx + size - 1, uly + size - 1);
  }
};
```

```cpp
struct PartialSum2D : PrefixSum2D {
  vvll diff; // 0 based
  int n, m;
  PartialSum2D(int _n, int _m) : n(_n), m(_m) {
    diff.assign(n + 1, vll(m + 1, 0));
  }
  // add c from [ulx,uly] to [brx,bry]
  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;
    diff[brx + 1][bry + 1] += c;
  }
  void update(int ulx, int uly, int size, ll c) {
    int brx = ulx + size - 1;
    int bry = uly + size - 1;
    update(ulx, uly, brx, bry, c);
  }
  // process the grid using prefix sum
  void process() { this->build(diff); }
};
// usage
PrefixSum2D pref;
pref.build(v); // takes 2d 0-based vector as input
pref.query(x1, y1, x2, y2); // sum of region

PartialSum2D part(n, m);       // dimension of grid 0 based
part.update(x1, y1, x2, y2, 1); // add 1 in region
// must run after all updates
part.process(); // prefix sum on diff array
// only exists after processing
vvll &grid = part.pref;     // processed diff array
part.query(x1, y1, x2, y2); // gives sum of region
```

2.3. Sparse Table

```cpp
/**
 * Sparse Table (Generic)
 * * Purpose: Static Range Queries
 * * Query Time:
 * - Idempotent (Min/Max/GCD): O(1)
 * - Non-Idempotent (Sum/Prod/XOR): O(log N)
 */
struct SparseTable {
  vector<vector<long long>> sparse;
  vector<int> Log;
  int n, max_log;
  long long IDENTITY_VAL; // e.g., 0 for Sum, 1 for Product,
                          // INF for Min

  // 1. OPERATION FUNCTION (Change this)
  long long func(long long a, long long b) {
    return (a + b); // Example: Sum
  }

```

```cpp
  // 2. BUILD
  void build(const vector<long long> &a,
             long long identity) {
    n = a.size();
    IDENTITY_VAL = identity;
    max_log = 32 - __builtin_clz(n);
    sparse.assign(n, vector<long long>(max_log));
    Log.assign(n + 1, 0);

    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];

    for (int j = 1; (1 << j) <= n; ++j) {
      for (int i = 0; i + (1 << j) <= n; ++i) {
        sparse[i][j] =
        func(sparse[i][j - 1],
             sparse[i + (1 << (j - 1))][j - 1]);
      }
    }
  }

  // 3. IDEMPOTENT QUERY O(1)
  // Use for: Min, Max, GCD, OR, AND
  long long query_idempotent(int l, int r) {
    int k = Log[r - l + 1];
    return func(sparse[l][k], sparse[r - (1 << k) + 1][k]);
  }

  // 4. NON-IDEMPOTENT QUERY O(log N)
  // Use for: Sum, Product, XOR, Matrix Multiplication
  // Logic: Decomposes range [l, r] into disjoint power-of-2
  // blocks
  long long query_non_idempotent(int l, int r) {
    long long res = IDENTITY_VAL;
    for (int j = max_log - 1; j >= 0; --j) {
      if ((1 << j) <= r - l + 1) {
        // Combine current result with the next block
        res = func(res, sparse[l][j]);
        // Move L forward by 2^j
        l += (1 << j);
      }
    }
    return res;
  }
};
```

2.4. Fenwick Tree

```cpp
/*
 * Interface: 0-based indexing (Internal logic handles
 * 1-based conversion). for point updates and range query
 */
struct FenwickTree {
  // Use long long to prevent overflow during sum operations
```

```cpp
vector<long long> bit;
int n;

// Initialize with size n. All values 0.
FenwickTree(int n) {
  this->n = n + 1;
  bit.assign(n + 1, 0);
}

// Initialize with an existing vector
// Passing by reference (const &) saves memory copy time
FenwickTree(const vector<int> &a)
    : FenwickTree(a.size()) {
  for (size_t i = 0; i < a.size(); i++) add(i, a[i]);
}

// Computes sum of [0...idx]
long long sum(int idx) {
  long long ret = 0;
  for (++idx; idx > 0; idx -= idx & -idx) ret += bit[idx];
  return ret;
}

// Computes sum of range [l...r] (inclusive)
long long sum(int l, int r) {
  if (l > r) return 0; // Guard clause
  return sum(r) - sum(l - 1);
}

// Adds delta to the element at idx
void add(int idx, int delta) {
  for (++idx; idx < n; idx += idx & -idx)
    bit[idx] += delta;
}
};
```

2.5. Longest Increasing Subsequence

```cpp
template <class I> vi lis(const vector<I> &S) {
  if (S.empty()) return {};
  vi prev(sz(S));
  typedef pair<I, int> p;
  vector<p> res;
  rep(i, 0, sz(S)) {
    // change 0 -> i for longest non-decreasing subsequence
    auto it = lower_bound(all(res), p{S[i], 0});
    if (it == res.end())
      res.emplace_back(), it = res.end() - 1;
    *it = {S[i], i};
    prev[i] = it == res.begin() ? 0 : (it - 1)->second;
  }
  int L = sz(res), cur = res.back().second;
  vi ans(L);
  while (L--) ans[L] = cur, cur = prev[cur];
  return ans;
```

```
}
```

2.6. Xobriest

```
long long randInRange(long long l, long long r) {
  static random_device rd;      // seed
  static mt19937_64 gen(rd()); // 64-bit Mersenne Twister
  uniform_int_distribution<long long> dist(l, r);
  return dist(gen);
}
void solve() {
  map<ll, pll> mp;
  vector<pll> pref;
  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;
  }
}
```

3. Graph

3.1. Kuhn-Munkres algorithm

```
// Maximum Weight Perfect Bipartite Matching
// Detect non-perfect-matching:
// 1. set all edge[i][j] as INF
// 2. if solve() >= INF, it is not perfect matching.

typedef long long ll;
struct KM {
  static const int MAXN = 1050;
  static const ll INF = 1LL << 60;
  int n, match[MAXN], vx[MAXN], vy[MAXN];
  ll edge[MAXN][MAXN], lx[MAXN], ly[MAXN], slack[MAXN];
  void init(int _n) {
    n = _n;
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++) edge[i][j] = 0;
  }
  void add_edge(int x, int y, ll w) { edge[x][y] = w; }
  bool DFS(int x) {
    vx[x] = 1;
    for (int y = 0; y < n; y++) {
      if (vy[y]) continue;
      if (lx[x] + ly[y] > edge[x][y]) {
        slack[y] =
        min(slack[y], lx[x] + ly[y] - edge[x][y]);
      } else {
        vy[y] = 1;
        if (match[y] == -1 || DFS(match[y])) {
          match[y] = x;
          return true;
        }
      }
    }
    return false;
  }
  ll solve() {
    fill(match, match + n, -1);
    fill(lx, lx + n, -INF);
    fill(ly, ly + n, 0);
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        lx[i] = max(lx[i], edge[i][j]);
    for (int i = 0; i < n; i++) {
      fill(slack, slack + n, INF);
      while (true) {
        fill(vx, vx + n, 0);
        fill(vy, vy + n, 0);
        if (DFS(i)) break;
        ll d = INF;
        for (int j = 0; j < n; j++)
          if (!vy[j]) d = min(d, slack[j]);
        for (int j = 0; j < n; j++) {
```

```
        if (vx[j]) lx[j] -= d;
        if (vy[j]) ly[j] += d;
        else slack[j] -= d;
      }
    }
  }
  ll res = 0;
  for (int i = 0; i < n; i++) {
    res += edge[match[i]][i];
  }
  return res;
 }
} graph;
```

3.2. Strongly Connected Components

```
struct TarjanScc {
 int n, step;
 vector<int> time, low, instk, stk;
 vector<vector<int>> e, scc;
 TarjanScc(int n_)
    : n(n_), step(0), time(n), low(n), instk(n), e(n) {}
 void add_edge(int u, int v) { e[u].push_back(v); }
 void dfs(int x) {
  time[x] = low[x] = ++step;
  stk.push_back(x);
  instk[x] = 1;
  for (int y : e[x])
   if (!time[y]) {
    dfs(y);
    low[x] = min(low[x], low[y]);
   } else if (instk[y]) {
    low[x] = min(low[x], time[y]);
   }
  if (time[x] == low[x]) {
   scc.emplace_back();
   for (int y = -1; y != x;) {
    y = stk.back();
    stk.pop_back();
    instk[y] = 0;
    scc.back().push_back(y);
   }
  }
 }
 void solve() {
  for (int i = 0; i < n; i++)
   if (!time[i]) dfs(i);
  reverse(scc.begin(), scc.end());
  // scc in topological order
 }
};
```

3.3. Biconnected Components

3.3.1. Articulation Points

```
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;
}
};
```

4. Math

4.1. SOS DP

```
const long long LOG = 20;
const long long sz = (1 << LOG);
void forward1(
vector<long long> &dp) { // subSet contribution to superset
  for (int b = 0; b <= LOG; b++)
    for (int i = 0; i <= sz; i++)
      if (i & (1 << b)) dp[i] += dp[i ^ (1 << b)];
}
void backward1(vector<long long> &dp) { // undo of forward 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)];
}
void forward2(
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];
}
void backward2(vector<long long> &dp) { // undo of forward 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];
}
```

4.2. Matrix

```
/**
 * Generic Matrix Template
 * * Purpose: Matrix Exponentiation and Operations
 * * Usage:
 * Matrix A(n, n, 1e9+7); // Modular Arithmetic
 * Matrix B(n, n);       // Standard Arithmetic (mod = 0)
 * * Complexity: Multiplication O(N^3), Power O(N^3 log Exp)
 */
struct Matrix {
  using ll = long long;
  vector<vector<ll>> mat;
  int rows, cols;
  ll mod; // mod = 0 implies Standard Arithmetic (No Modulo)

  // Constructor: Default mod is 0 (No Mod)
  Matrix(int r, int c, ll m = 0)
      : rows(r), cols(c), mod(m) {
    mat.assign(rows, vector<ll>(cols, 0));
  }

  // Input Matrix
  void input() {
    for (int i = 0; i < rows; ++i)
      for (int j = 0; j < cols; ++j) cin >> mat[i][j];
  }
```

```
// Print Matrix
void print() const {
  for (const auto &row : mat) {
    for (const auto &val : row) cout << val << " ";
    cout << endl;
  }
}

// Addition
Matrix operator+(const Matrix &other) const {
  Matrix result(rows, cols, mod);
  for (int i = 0; i < rows; ++i) {
    for (int j = 0; j < cols; ++j) {
      result.mat[i][j] = mat[i][j] + other.mat[i][j];
      if (mod) result.mat[i][j] %= mod;
    }
  }
  return result;
}

// Subtraction
Matrix operator-(const Matrix &other) const {
  Matrix result(rows, cols, mod);
  for (int i = 0; i < rows; ++i) {
    for (int j = 0; j < cols; ++j) {
      result.mat[i][j] = mat[i][j] - other.mat[i][j];
      if (mod)
        result.mat[i][j] =
        (result.mat[i][j] % mod + mod) % mod;
    }
  }
  return result;
}

// Multiplication O(N^3)
Matrix operator*(const Matrix &other) const {
  // Assert matching dimensions if needed: assert(cols ==
  // other.rows);
  Matrix result(rows, other.cols, mod);
  for (int i = 0; i < rows; ++i) {
    for (int k = 0; k < cols;
         ++k) { // Optimized loop order (i-k-j is cache
                // friendly)
      if (mat[i][k] == 0)
        continue; // Optimization for sparse matrices
      for (int j = 0; j < other.cols; ++j) {
        if (mod) {
          result.mat[i][j] =
          (result.mat[i][j] +
           mat[i][k] * other.mat[k][j]) %
          mod;
        } else {
          result.mat[i][j] += mat[i][k] * other.mat[k][j];
        }
```

```
      }
    }
  }
  return result;
}

// Matrix Exponentiation O(N^3 log Exp)
Matrix power(ll exp) const {
  // Identity Matrix
  Matrix result(rows, cols, mod);
  for (int i = 0; i < rows; ++i) result.mat[i][i] = 1;

  Matrix base = *this;
  while (exp > 0) {
    if (exp & 1) result = result * base;
    base = base * base;
    exp >>= 1;
  }
  return result;
}
};
```

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

```
template <typename T> struct M {
  static T MOD; // change to constexpr if already known
  T v;
  M(T x = 0) {
    v = (-MOD <= x && x < MOD) ? x : x % MOD;
    if (v < 0) v += MOD;
  }
  explicit operator T() const { return v; }
  bool operator==(const M &b) const { return v == b.v; }
  bool operator!=(const M &b) const { return v != b.v; }
  M operator-() { return M(-v); }
  M operator+(M b) { return M(v + b.v); }
  M operator-(M b) { return M(v - b.v); }
  M operator*(M b) { return M((__int128)v * b.v % MOD); }
  M operator/(M b) { return *this * (b ^ (MOD - 2)); }
```

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

4.3.2. Miller-Rabin

Requires: Mod Struct

```


// checks if Mod::MOD is prime
bool is_prime() {
  if (MOD < 2 || MOD % 2 == 0) return MOD == 2;
  Mod A[] = {2, 7, 61}; // for int values (< 2^31)
  // ll: 2, 325, 9375, 28178, 450775, 9780504, 1795265022
  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;
}
```

4.3.3. Linear Sieve

```
constexpr ll MAXN = 1000000;
bitset<MAXN> is_prime;
vector<ll> primes;
ll mpf[MAXN], phi[MAXN], mu[MAXN];

void sieve() {
  is_prime.set();
  is_prime[1] = 0;
  mu[1] = phi[1] = 1;
  for (ll i = 2; i < MAXN; i++) {
    if (is_prime[i]) {
      mpf[i] = i;
      primes.push_back(i);
      phi[i] = i - 1;
      mu[i] = -1;
```

```
    }
    for (ll p : primes) {
      if (p > mpf[i] || i * p >= MAXN) break;
      is_prime[i * p] = 0;
      mpf[i * p] = p;
      mu[i * p] = -mu[i];
      if (i % p == 0)
        phi[i * p] = phi[i] * p, mu[i * p] = 0;
      else phi[i * p] = phi[i] * (p - 1);
    }
  }
}
```

4.3.4. Get Factors

Requires: Linear Sieve

```


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

4.3.5. Extended GCD

```
// 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
tuple<ll, ll, ll> extgcd(ll a, ll b) {
  ll s = 1, t = 0, u = 0, v = 1;
  while (b) {
    ll q = a / b;
    swap(a -= q * b, b);
    swap(s -= q * t, t);
    swap(u -= q * v, v);
  }
  return {s, u, a};
}
```

4.3.6. Chinese Remainder Theorem

Requires: Extended GCD

```

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

```
  if (n > m) swap(a, b), swap(m, n);
  auto [x, y, g] = extgcd(m, n);
  assert((a - b) % g == 0); // no solution
  x = ((b - a) / g * x) % (n / g) * m + a;
  return x < 0 ? x + m / g * n : x;
}
```

4.3.7. Pollard Rho

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

4.3.8. De Brujin Sequence

```
int res[kN], aux[kN], a[kN], sz;
void Rec(int t, int p, int n, int k) {
  if (t > n) {
    if (n % p == 0)
      for (int i = 1; i <= p; ++i) res[sz++] = aux[i];
  } else {
    aux[t] = aux[t - p];
    Rec(t + 1, p, n, k);
    for (aux[t] = aux[t - p] + 1; aux[t] < k; ++aux[t])
      Rec(t + 1, t, n, k);
  }
}
int DeBruijn(int k, int n) {
  // return cyclic string of length k^n such that every
  // string of length n using k character appears as a
  // substring.
  if (k == 1) return res[0] = 0, 1;
  fill(aux, aux + k * n, 0);
  return sz = 0, Rec(1, 1, n, k), sz;
}
```

4.3.9. Combinatorics

```
struct Combinatorics {
  const int MOD;
  vector<long long> fact, invFact;

  // Constructor
  Combinatorics(int maxN, int mod)
```

```
    : MOD(mod), fact(maxN + 1), invFact(maxN + 1) {
  precompute(maxN);
 }

 // Function to perform modular exponentiation: a^b % MOD
 long long modpow(long long a, long long b) const {
  long long res = 1;
  while (b) {
   if (b & 1) res = res * a % MOD;
   a = a * a % MOD;
   b >>= 1;
  }
  return res;
 }

 // Precomputing factorials and modular inverses
 void precompute(int maxN) {
  fact[0] = 1;
  for (int i = 1; i <= maxN; i++) {
   fact[i] = fact[i - 1] * i % MOD;
  }
  invFact[maxN] =
  modpow(fact[maxN], MOD - 2); // Fermat's Little Theorem
  for (int i = maxN - 1; i >= 0; i--) {
   invFact[i] = invFact[i + 1] * (i + 1) % MOD;
  }
 }

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

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

 // Function to calculate n! % MOD
 long long factorial(int n) const { return fact[n]; }
};
// Combinatorics comb(maxN,mod)
```

5. Geometry

5.1. convex hull

```
struct Point {
  ll x, y;
  Point() : x(0), y(0) {}
  Point(ll _x, ll _y) : x(_x), y(_y) {}
  bool operator==(const Point &other) const {
    return x == other.x && y == other.y;
  }
  bool operator<(const Point &other) const {
    if (x != other.x) return x < other.x;
    return y < other.y;
  }
};
ll cross_product(const Point &A, const Point &B,
                 const Point &C) {
  /*
  cross(A, B, C) tells you how the angle turns when you go A
  → B → C. If cross > 0 → left turn (counter-clockwise) If
  cross < 0 → right turn (clockwise) If cross = 0 →
  collinear
  */
  return (B.x - A.x) * (C.y - A.y) -
         (B.y - A.y) * (C.x - A.x);
}
long long dot_product(const Point &A, const Point &B,
                      const Point &C) {
  // computes (B - A) · (C - A)
  return (B.x - A.x) * (C.x - A.x) +
         (B.y - A.y) * (C.y - A.y);
}
vector<Point> ConvexHullAndrowChain(vector<Point> pts) {
  sort(pts);
  pts.erase(unique(pts.begin(), pts.end()), pts.end());
  int n = pts.size();
  if (n <= 1) return pts;
  vector<Point> lower, upper;
  // Build lower hull
  for (int i = 0; i < n; ++i) {
    const Point &p = pts[i];
    while (lower.size() >= 2 &&
           cross_product(lower[lower.size() - 2],
                         lower[lower.size() - 1], p) <= 0) {
      lower.pop_back();
    }
    lower.push_back(p);
  }
  // Build upper hull
  for (int i = n - 1; i >= 0; --i) {
    const Point &p = pts[i];
    while (upper.size() >= 2 &&
           cross_product(upper[upper.size() - 2],
                         upper[upper.size() - 1], p) <= 0) {
```

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

  return hull; // CCW order
}
```

6. Strings

6.1. Knuth-Morris-Pratt Algorithm

```
vector<int> pi(const string &s) {
  vector<int> p(s.size());
  for (int i = 1; i < s.size(); i++) {
    int g = p[i - 1];
    while (g && s[i] != s[g]) g = p[g - 1];
    p[i] = g + (s[i] == s[g]);
  }
  return p;
}
vector<int> match(const string &s, const string &pat) {
  vector<int> p = pi(pat + '\0' + s), res;
  for (int i = p.size() - s.size(); i < p.size(); i++)
    if (p[i] == pat.size())
      res.push_back(i - 2 * pat.size());
  return res;
}
```

6.2. Z Value

```
int z[n];
void zval(string s) {
  // z[i] => longest common prefix of s and s[i:], i > 0
  int n = s.size();
  z[0] = 0;
  for (int b = 0, i = 1; i < n; i++) {
    if (z[b] + b <= i) z[i] = 0;
    else z[i] = min(z[i - b], z[b] + b - i);
    while (s[i + z[i]] == s[z[i]]) z[i]++;
    if (i + z[i] > b + z[b]) b = i;
  }
}
```

6.3. Manachers Algorithm

```
int z[n];
void manacher(string s) {
  // z[i] => longest odd palindrome centered at i is
  //        s[i - z[i] ... i + z[i]]
  // to get all palindromes (including even length),
  // insert a '#' between each s[i] and s[i + 1]
  int n = s.size();
  z[0] = 0;
  for (int b = 0, i = 1; i < n; i++) {
    if (z[b] + b >= i)
      z[i] = min(z[2 * b - i], b + z[b] - i);
    else z[i] = 0;
    while (i + z[i] + 1 < n && i - z[i] - 1 >= 0 &&
           s[i + z[i] + 1] == s[i - z[i] - 1])
      z[i]++;
    if (z[i] + i > z[b] + b) b = i;
  }
}
```

6.4. Trie

```cpp
class Trie {
public:
  struct Node {
    vector<int> next; // Indices of children nodes
    int pfxCnt = 0; // How many words pass through this node
    int wordCnt =
    0; // How many words end exactly at this node

    Node(int maxChars) {
      next.assign(maxChars, -1);
      pfxCnt = 0;
      wordCnt = 0;
    }
  };
  vector<Node> nodes;
  int
  distWords;    // Count of distinct words currently in Trie
  int maxChars; // Alphabet size (usually 26)
  int getBase(char c) {
    return c - 'Change'; // based on problem
  }

  Trie(int maxChars = 26) {
    this->maxChars = maxChars;
    nodes.clear();
    distWords = 0;
    // Create Root Node (Index 0)
    nodes.emplace_back(maxChars);
  }

  // Insert string s into Trie
  void insert(string s) {
    int curr = 0;
    nodes[curr].pfxCnt++;
    for (char &ch : s) {
      int base = getBase(ch);
      // If path doesn't exist, create new node
      if (nodes[curr].next[base] == -1) {
        nodes[curr].next[base] = nodes.size();
        nodes.emplace_back(maxChars);
      }
      curr = nodes[curr].next[base];
      nodes[curr].pfxCnt++;
    }
    if (nodes[curr].wordCnt == 0) {
      distWords++; // New distinct word found
    }
    nodes[curr].wordCnt++;
  }

  // Check if string s exists
  bool search(string s) {
    int curr = 0;
```

```cpp
    for (char &ch : s) {
      int base = getBase(ch);
      if (nodes[curr].next[base] == -1) return false;
      curr = nodes[curr].next[base];
    }
    return nodes[curr].wordCnt > 0;
  }
  // Delete one occurrence of s
  void erase(string s) {
    if (!search(s)) return; // Check existence first
    int curr = 0;
    nodes[curr].pfxCnt--;
    for (char &ch : s) {
      int base = getBase(ch);
      curr = nodes[curr].next[base];
      nodes[curr].pfxCnt--;
    }
    nodes[curr].wordCnt--;
    if (nodes[curr].wordCnt == 0)
      distWords--; // Word completely removed
  }

  // Count words that have s as a prefix
  int prefixCount(string s) {
    int curr = 0;
    for (char &ch : s) {
      int base = getBase(ch);
      if (nodes[curr].next[base] == -1)
        return 0; // Prefix not found
      curr = nodes[curr].next[base];
    }
    return nodes[curr].pfxCnt;
  }
};
```

6.5. Aho-Corasick Automaton

```cpp
const int ALPHA = 26, MAXNODES = 500000 + 5;
int nxt[MAXNODES][ALPHA];
int linkS[MAXNODES];
ll cntNode[MAXNODES];
vector<int> adjSL[MAXNODES];
vector<int> patEnd;
int nodes = 1;

void build_trie(const vector<string> &P) {
  // clear
  for (int i = 0; i < nodes; i++) {
    memset(nxt[i], 0, sizeof nxt[i]);
    cntNode[i] = 0;
    adjSL[i].clear();
  }
  nodes = 1;
  patEnd.clear();
  patEnd.reserve(P.size());
```

```
  // insert
  for (auto &pat : P) {
    int u = 0;
    for (char ch : pat) {
      int c = ch - 'a';
      if (!nxt[u][c]) nxt[u][c] = nodes++;
      u = nxt[u][c];
    }
    patEnd.pb(u);
  }
}
vector<int> bfsOrder;
void build_links() {
  queue<int> q;
  linkS[0] = 0;
  // first layer
  for (int c = 0; c < ALPHA; c++) {
    int v = nxt[0][c];
    if (v) {
      linkS[v] = 0;
      q.push(v);
    }
  }
  // BFS
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    bfsOrder.pb(u);
    for (int c = 0; c < ALPHA; c++) {
      int v = nxt[u][c];
      if (!v) continue;
      int j = linkS[u];
      while (j && !nxt[j][c]) j = linkS[j];
      if (nxt[j][c]) j = nxt[j][c];
      linkS[v] = j;
      q.push(v);
    }
  }
  for (int u : bfsOrder) { adjSL[linkS[u]].pb(u); }
}

void solve() {
  string S;
  ll k;
  cin >> S >> k;
  vector<string> P(k);
  for (int i = 0; i < k; i++) cin >> P[i];
  build_trie(P);
  bfsOrder.clear();
  build_links();
}
```

7. Debug List

```
- Pre-submit:
  - Did you make a typo when copying a template?
  - Test more cases if unsure.
    - Write a naive solution and check small cases.
  - Submit the correct file.

- General Debugging:
  - Read the whole problem again.
  - Have a teammate read the problem.
  - Have a teammate read your code.
    - Explain you solution to them (or a rubber duck).
  - Print the code and its output / debug output.
  - Go to the toilet.

- Wrong Answer:
  - Any possible overflows?
    - > `__int128` ?
    - Try `-ftrapv` or `#pragma GCC optimize("trapv")`
  - Floating point errors?
    - > `long double` ?
    - turn off math optimizations
    - check for `==`, `>=`, `acos(1.000000001)`, etc.
  - Did you forget to sort or unique?
  - Generate large and worst "corner" cases.
  - Check your `m` / `n`, `i` / `j` and `x` / `y`.
  - Are everything initialized or reset properly?
  - Are you sure about the STL thing you are using?
    - Read cppreference (should be available).
  - Print everything and run it on pen and paper.

- Time Limit Exceeded:
  - Calculate your time complexity again.
  - Does the program actually end?
    - Check for `while(q.size())` etc.
  - Test the largest cases locally.
  - Did you do unnecessary stuff?
    - e.g. pass vectors by value
    - e.g. `memset` for every test case
  - Is your constant factor reasonable?

- Runtime Error:
  - Check memory usage.
    - Forget to clear or destroy stuff?
    - > `vector::shrink_to_fit()`
  - Stack overflow?
  - Bad pointer / array access?
    - Try `-fsanitize=address`
  - Division by zero? NaN's?
```