Contents

1 Misc **2**
- 1.1 Contest . . . 2
  - 1.1.1 Macros and debug . . . 2

2 Data Structures **3**
- 2.1 GNU PBDS . . . 3
- 2.2 2D Partial Sums . . . 3
- 2.3 Sparse Table . . . 4
- 2.4 Fenwick Tree . . . 5
- 2.5 Longest Increasing Subsequence . . . 6
- 2.6 Xobriest . . . 6

3 Graph **7**
- 3.1 General . . . 7
  - 3.1.1 Bellman . . . 7
  - 3.1.2 Floyd . . . 7
  - 3.1.3 DSU . . . 8
  - 3.1.4 Binary Lifting . . . 8
- 3.2 Kuhn-Munkres algorithm . . . 9
- 3.3 Strongly Connected Components . . . 11
- 3.4 Biconnected Components . . . 11
  - 3.4.1 Articulation Points . . . 11

4 Math **13**
- 4.1 SOS DP . . . 13
- 4.2 Matrix . . . 13
- 4.3 Number Theory . . . 14
  - 4.3.1 Miller-Rabin . . . 14
  - 4.3.2 Linear Sieve . . . 15
  - 4.3.3 Get Factors . . . 16
  - 4.3.4 Extended GCD . . . 16
  - 4.3.5 Chinese Remainder Theorem . . . 16
  - 4.3.6 Pollard Rho . . . 17
  - 4.3.7 De Brujin Sequence . . . 17
  - 4.3.8 Combinatorics . . . 17

5 Geometry **19**
- 5.1 convex hull . . . 19

6 Strings **21**
- 6.1 Knuth-Morris-Pratt Algorithm . . . 21
- 6.2 Z Value . . . 21
- 6.3 Manachers Algorithm . . . 21
- 6.4 Trie . . . 22
- 6.5 Aho-Corasick Automaton . . . 23
- 6.6 Hashing . . . 24

1. Misc

1.1. Contest

1.1.1. Macros and debug

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

```
- 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?
  - Division by zero? NaN's?
```

2. Data Structures

2.1. GNU PBDS

```
// #include <ext/pb_ds/assoc_container.hpp>
// #include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template <typename T>
using ordered_set =
tree<T, null_type, std::less<T>, rb_tree_tag,
     tree_order_statistics_node_update>;
ordered_set<int> os;
os.insert(5);
os.erase(5);
os.find_by_order(k); // iterator to k-th element (0-indexed)
os.order_of_key(x);  // number of elements < x
template <typename T>
using ordered_multiset_base =
tree<pair<T, int>, null_type, less<pair<T, int>>,
     rb_tree_tag, tree_order_statistics_node_update>;
template <typename T> struct ordered_multiset {
  ordered_multiset_base<T> os;
  int timer = 0;
  void insert(T x) { os.insert({x, timer++}); }
  void erase_one(T x) {
    auto it = os.lower_bound({x, -1});
    if (it != os.end() && it->first == x) os.erase(it);
  }
  int order_of_key(T x) { return os.order_of_key({x, -1}); }
  T find_by_order(int k) {
    return os.find_by_order(k)->first;
  }
  int size() { return (int)os.size(); }
};
```

2.2. 2D Partial Sums

```
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);
  }
};
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, bry = uly + size - 1;
    update(ulx, uly, brx, bry, c);
  }
  void process() {
    this->build(diff);
  } // process grid using prefix sum
};
// 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

```
// Purpose: Static Range Queries
// 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,
  long long func(long long a, long long b) {
    return (a + b); // Example: Sum
  }
  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) 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
  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]);
        l += (1 << j); // Move L forward by 2^j
      }
    }
    return res;
  }
};
```

2.4. Fenwick Tree

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

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

2.5. Longest Increasing Subsequence

```
int lis(vector<int> const &a) {
  int n = a.size();
  const int INF = 1e9;
  vector<int> d(n + 1, INF); // min possible ending value
  // of inc subseq of length l, that we have seen
  d[0] = -INF;
  // user lower bound for non-decreasing
  for (int i = 0; i < n; i++) {
    int l =
    upper_bound(d.begin(), d.end(), a[i]) - d.begin();
    if (d[l - 1] < a[i] && a[i] < d[l]) d[l] = a[i];
  }
  int ans = 0;
  for (int l = 0; l <= n; l++) {
    if (d[l] < INF) ans = l;
  }
  return ans;
}
```

2.6. Xobriest

```
static uint64_t seed =
chrono::steady_clock::now().time_since_epoch().count();
uint64_t splitmix64() {
  seed += 0x9e3779b97f4a7c15;
  uint64_t x = seed;
  x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
  x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
  return x ^ (x >> 31);
}
long long randInRange(long long l, long long r) {
  assert(l <= r);
  return l + splitmix64() % (r - l + 1);
}
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. General

3.1.1. Bellman

```
const long long INF = 1e18;
vector<array<int, 3>> e; // {u, v, w}
vector<long long> d, par;
int main() {
 int n, m;
 cin >> n >> m;
 e.resize(m);
 d.assign(
 n, 0); // use 0 to detect neg-cycle in disconnected graph
 par.assign(n, -1);
 int x = -1;
 for (int i = 0; i <= n; i++) {
  x = -1;
  for (auto [u, v, w] : e) {
   if (d[v] > d[u] + w) {
    d[v] = d[u] + w;
    par[v] = u;
    x = v;
   }
  }
 }
 if (x == -1) {
  // no negative cycle
 } else {
  // negative cycle exists
  for (int i = 0; i < n; i++) x = par[x];
  vector<int> cyc;
  for (int v = x;; v = par[v]) {
   cyc.push_back(v);
   if (v == x && cyc.size() > 1) break;
  }
  reverse(cyc.begin(), cyc.end());
 }
}
```

3.1.2. Floyd

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

```
  }
  while (m--) {
    int u, v, w;
    cin >> u >> v >> w;
    --u;
    --v;
    d[u][v] = min(d[u][v], w);
  }
  for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        if (d[i][k] < INF && d[k][j] < INF &&
            d[i][j] > d[i][k] + d[k][j]) {
          d[i][j] = d[i][k] + d[k][j];
          p[i][j] = p[k][j];
        }
  for (int i = 0; i < n; i++)
    if (d[i][i] < 0) {
      // negative cycle exists
    }
}
```

3.1.3. DSU

```
struct DSU {
  vector<int> p, sz;
  DSU(int n = 0) { init(n); }
  void init(int n) {
    p.resize(n);
    sz.assign(n, 1);
    iota(p.begin(), p.end(), 0);
  }
  int find(int x) {
    if (p[x] == x) return x;
    return p[x] = find(p[x]); // path compression
  }
  bool unite(int a, int b) {
    a = find(a);
    b = find(b);
    if (a == b) return false;
    if (sz[a] < sz[b]) swap(a, b);
    p[b] = a;
    sz[a] += sz[b];
    return true;
  }
  bool same(int a, int b) { return find(a) == find(b); }
  int size(int x) { return sz[find(x)]; }
};
```

3.1.4. Binary Lifting

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

```
  jump[root][0] = parent;
  g[root][0] = val[root];
  for (int i = 1; i < 20;
       i++) { // always start from 1, since i depends on i-1
    if (jump[root][i - 1] != -1) {
      jump[root][i] = jump[jump[root][i - 1]][i - 1];
      g[root][i] =
      gcd(g[root][i - 1],
          g[jump[root][i - 1]][i - 1]); // combination logic
    } else {
      jump[root][i] = -1;
      g[root][i] =
      g[root]
       [i - 1]; // when the i-1th ancestor doesn't exist,
              // then store same aggregate as i-1th
    }
  }
  // do normal DFS from here
}
int path_gcd(int u, int v) {
  if (depth[u] < depth[v]) swap(u, v);
  int GCD = 0; // always use the identity value
  for (int i = 19; i >= 0; i--) {
    if (((depth[u] - depth[v]) >> i) & 1) {
      {
       GCD = gcd(g[u][i], GCD);
       u = jump[u][i];
      }
    }
  }
  if (u == v)
    return gcd(
    GCD,
    val[u]); // g[u][i], doesn't contain val at jump[u][i]
  for (int i = 19; i >= 0; i--) {
    if (jump[u][i] != jump[v][i]) {
      GCD = gcd(GCD, g[u][i]);
      GCD = gcd(GCD, g[v][i]);
      u = jump[u][i];
      v = jump[v][i];
    }
  }
  // since both u and v are immediate child of lca
  // neither u, v nor lca is included in the computation
  // add them explicitly
  GCD = gcd(GCD, val[u]);
  GCD = gcd(GCD, val[v]);
  return gcd(GCD, val[jump[u][0]]);
}
```

3.2. Kuhn-Munkres algorithm

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

```cpp
// 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;
```

3.3. 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
 }
};
```

3.4. Biconnected Components

3.4.1. 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

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

```cpp
  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 {
    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. Miller-Rabin

```cpp
using u128 = __uint128_t;
using ull = unsigned long long;
```

```
ull mod_mul(ull a, ull b, ull mod) {
  return (u128)a * b % mod;
}
ull mod_pow(ull a, ull d, ull mod) {
  ull res = 1;
  while (d) {
    if (d & 1) res = mod_mul(res, a, mod);
    a = mod_mul(a, a, mod);
    d >>= 1;
  }
  return res;
}
bool isPrime(ull n) {
  if (n < 2) return false;
  for (ull p : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37})
    if (n % p == 0) return n == p;
  ull d = n - 1;
  int s = __builtin_ctzll(d);
  d >>= s;
  auto check = [&](ull a) {
    ull x = mod_pow(a, d, n);
    if (x == 1 || x == n - 1) return true;
    for (int r = 1; r < s; r++) {
      x = mod_mul(x, x, n);
      if (x == n - 1) return true;
    }
    return false;
  };
  // Proven deterministic bases for 64-bit
  // {2, 7, 61} for 32 bits
  for (ull a :
      {2, 325, 9375, 28178, 450775, 9780504, 1795265022})
    if (a < n && !check(a)) return false;
  return true;
}
```

4.3.2. 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);
    }
  }
}
```

4.3.3. Get Factors

```
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;
}
```

4.3.4. 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};
}
```

4.3.5. 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;
}
```

4.3.6. 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;
  }
}
```

4.3.7. 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;
}
```

4.3.8. 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;
```

```cpp
    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

```
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;
```

```
  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

```
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();
}
```

6.6. Hashing

```
struct Hasher64 {
```

```
int n;
vector<uint32_t> pref1, pref2, pow1, pow2;
Hasher64() {}
Hasher64(const string &s) {
  n = s.size();
  pref1.assign(n + 1, 0);
  pref2.assign(n + 1, 0);
  pow1.assign(n + 1, 0);
  pow2.assign(n + 1, 0);
  pow1[0] = pow2[0] = 1;
  for (int i = 1; i <= n; i++) {
    uint32_t val = (s[i - 1] - 'A' + 1);
    pref1[i] =
    (val + (uint64_t)BASE1 * pref1[i - 1]) % MOD1;
    pref2[i] =
    (val + (uint64_t)BASE2 * pref2[i - 1]) % MOD2;
    pow1[i] = (uint64_t)pow1[i - 1] * BASE1 % MOD1;
    pow2[i] = (uint64_t)pow2[i - 1] * BASE2 % MOD2;
  }
}
inline uint64_t getHash(int l, int r) {
  uint32_t h1 =
  (pref1[r + 1] -
   (uint64_t)pref1[l] * pow1[r - l + 1] % MOD1 + MOD1) %
  MOD1;
  uint32_t h2 =
  (pref2[r + 1] -
   (uint64_t)pref2[l] * pow2[r - l + 1] % MOD2 + MOD2) %
  MOD2;
  return (uint64_t(h2) << 32) | h1;
 }
};
Hasher64 s(a);
unordered_map<uint64_t, int> mp;
mp.reserve(n);
mp.max_load_factor(0.7);
uint64_t h = s.getHash(i, i + len - 1);
```