Lessa M3ana Bokra Templates

Directed By:

Habdasy

SMTYON

Const

📘 Table of Contents

◇ Main

main.cpp

```
#include <bits/stdc++.h>
using namespace std;
#define ll long long
#define ld long double

int main()
{
    // Fast
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    // File input
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);
    // input line int
    void ReadIntLine(vector<int>& numbers) {
    string line;
    getline(cin, line);

    istringstream is(line);

    numbers = vector<int>(istream_iterator<int>(is), istream_iterator<int>());
}

    return 0;
}
```

◇ Data structures

2D BIT tree.cpp

```
// ==================== 2D Fenwick Tree (Binary Indexed Tree)
====================
// ➤ Efficient structure for 2D prefix sums and point updates.
// ➤ Time Complexity:
//     - Point update: O(log N * log M)
//     - Prefix sum query: O(log N * log M)
// ➤ Use Cases:
//     - Dynamic 2D range sum queries
//     - Matrix/grid problems (e.g., subrectangle sums)
// ➤ Note:
//     - Indexing is 0-based
//     - Only supports point updates and prefix sum queries
//
struct FenwickTree2D {
    vector<vector<int>> bit; // 2D BIT array
    int n, m;

    // Constructor to initialize a 2D BIT of size n x m
    void init(int _n, int _m) {
        n = _n;
        m = _m;
        bit.assign(n, vector<int>(m, 0));
    }

    // Add `delta` to element at position (x, y)
    void add(int x, int y, int delta) {
        for (int i = x; i < n; i = i | (i + 1))
            for (int j = y; j < m; j = j | (j + 1))
                bit[i][j] += delta;
    }

    // Query prefix sum in rectangle from (0, 0) to (x, y), inclusive
    int sum(int x, int y) {
        int ret = 0;
        for (int i = x; i >= 0; i = (i & (i + 1)) - 1)
            for (int j = y; j >= 0; j = (j & (j + 1)) - 1)
                ret += bit[i][j];
        return ret;
    }

    // Query sum in subrectangle [(x1,y1), (x2,y2)], inclusive
    int rangeSum(int x1, int y1, int x2, int y2) {
        return sum(x2, y2)
             - sum(x1 - 1, y2)
             - sum(x2, y1 - 1)
             + sum(x1 - 1, y1 - 1);
    }
};
```

2D partial sum.cpp

```
// ========================= 2D Difference Array + Prefix Sum
=========================
// ➤ Purpose: Apply multiple rectangle updates on a 2D grid efficiently, and
query point values.
// ➤ Time Complexity:
//     - update(x1,y1,x2,y2,val): O(1)
//     - calc(): O(N*M) to build final matrix
//     - access result: O(1) for any cell
// ➤ Best for:
//     - Applying many range updates in 2D (e.g., increase submatrices)
//     - Use cases: image processing, 2D simulation, grid-based accumulations
//
using ll = long long;

struct partial_sum_2D {
    int n, m;
    vector<vector<ll>> upd;  // Difference array that holds lazy updates

    // Constructor: initializes an n × m 2D grid
    partial_sum_2D(int _n, int _m) {
        n = _n, m = _m;
        upd.assign(n, vector<ll>(m, 0));
    }

    // Rectangle update in range [x1, y1] to [x2, y2] by +val
    // Achieved using 2D difference array trick
    void update(int x1, int y1, int x2, int y2, ll val) {
        upd[x1][y1] += val;
        if (x2 + 1 < n) upd[x2 + 1][y1] -= val;
        if (y2 + 1 < m) upd[x1][y2 + 1] -= val;
        if (x2 + 1 < n && y2 + 1 < m) upd[x2 + 1][y2 + 1] += val;
    }

    // Finalize all updates by building prefix sums
    // After this, upd[i][j] holds the final value at cell (i, j)
    void calc() {
        // First pass: vertical prefix sum (column-wise)
        for (int i = 1; i < n; i++)
            for (int j = 0; j < m; j++)
                upd[i][j] += upd[i - 1][j];

        // Second pass: horizontal prefix sum (row-wise)
        for (int i = 0; i < n; i++)
            for (int j = 1; j < m; j++)
                upd[i][j] += upd[i][j - 1];
    }

    // Access result at cell (x, y): O(1)
    ll get(int x, int y) {
        return upd[x][y];
    }
};
```

BIT tree.cpp

```
// ==================== Fenwick Tree (Binary Indexed Tree - BIT)
====================
// ➤ Efficient structure for prefix sums and point updates.
// ➤ Time Complexity:
//     - Point update: O(log N)
//     - Prefix sum query: O(log N)
// ➤ Use Cases:
//     - Range sum queries (point update)
//     - Inversion count
//     - Frequency counters, lower_bound on prefix sums
// ➤ Note:
//     - Indexing is 0-based here
//     - Can be adapted for range updates and queries (via two BITs)
//
================================================================================
======

struct FenwickTree { // zero indexed
    vector<int> bit;  // Binary Indexed Tree array
    int n;

    // Constructor: initialize BIT with size n (0-based indexing)
    FenwickTree(int n) {
        this->n = n;
        bit.assign(n, 0); // initialize with zeros
    }

    // Query prefix sum from 0 to r (inclusive)
    int sum(int r) {
        int ret = 0;
        for (; r >= 0; r = (r & (r + 1)) - 1)
            ret += bit[r];
        return ret;
    }

    // Query range sum from l to r (inclusive)
    int sum(int l, int r) {
        return sum(r) - sum(l - 1);
    }

    // Add `delta` to element at index `idx`
    void add(int idx, int delta) {
        for (; idx < n; idx = idx | (idx + 1))
            bit[idx] += delta;
    }
};
```

DSU on Trees.cpp

```
#include <bits/stdc++.h>
using namespace std;

// ==================== DSU on Tree (a.k.a. "Small to Large")
====================
// ➤ Solves subtree queries on trees efficiently.
// ➤ Time Complexity: O(N log N) — each node's data is added/removed at most
logN times.
// ➤ Use Cases:
//      - Most frequent element in subtree
//      - Count of colors/frequencies/values in subtree
//      - Any DSU-like structure per subtree
// ➤ Approach:
//      - Use Euler Tour + subtree sizes
//      - Recursively process light children first (not keeping their data)
//      - Process heavy (big) child and keep its data
//      - Merge light children into big child data
//      - Process queries for current node
//      - If not `keep`, remove data to save memory/time
//
================================================================================
===

const int N = 1e5 + 5;

int sz[N], in[N];
vector<int> adj[N], e; // Euler tour traversal array

// Step 1: Preprocessing DFS — computes subtree sizes and entry times
int dfs(int u, int p) {
    sz[u] = 1;
    in[u] = e.size(); // Entry index in Euler tour
    e.push_back(u);   // Add node to Euler tour
    for (int v : adj[u]) {
        if (v != p) {
            sz[u] += dfs(v, u);
        }
    }
    return sz[u];
}

// Step 2: User-defined update function
// ➤ Modify this according to the problem:
//    - delta = +1 to add node u's data
//    - delta = -1 to remove node u's data
void update(int u, int delta) {
    // Example:
    // cnt[color[u]] += delta;
}
```

```
// Step 3: DSU on Tree logic
// ➤ Processes subtree of `u` with parent `p`
// ➤ If keep = true: keeps subtree data after processing
void dsu(int u, int p, bool keep) {
    int big = -1, max_sz = -1;

    // Identify heavy child (largest subtree)
    for (int v : adj[u]) {
        if (v != p && sz[v] > max_sz) {
            max_sz = sz[v];
            big = v;
        }
    }

    // Process all light children — don't keep their data
    for (int v : adj[u]) {
        if (v != p && v != big) {
            dsu(v, u, false);
        }
    }

    // Process heavy child — keep its data
    if (big != -1) {
        dsu(big, u, true);
    }

    // Add data from light children's subtrees
    for (int v : adj[u]) {
        if (v != p && v != big) {
            for (int i = in[v]; i < in[v] + sz[v]; ++i) {
                update(e[i], +1);
            }
        }
    }

    // Add current node's data
    update(u, +1);

    // ➤ Place subtree query logic here for node u
    // For example:
    // ans[u] = mostFrequentColor();

    // If not keeping, remove data after processing
    if (!keep) {
        for (int i = in[u]; i < in[u] + sz[u]; ++i) {
            update(e[i], -1);
        }
    }
}
```

DSU with rollback.cpp

```
// Disjoint Set Union (Rollback version)
// Supports undoing merges — useful in offline queries, Mo's on trees, D&C on
graphs

class RollbackDSU {
private:
    int n;
    vector<int> par, sz;
    stack<pair<int, int>> st; // Stack to track merge history
public:
    int components;

    RollbackDSU(int _n) : n(_n), par(_n + 5), sz(_n + 5, 1), components(_n) {
        iota(par.begin(), par.end(), 0);
    }

    int find(int u) {
        return par[u] == u ? u : find(par[u]); // No path compression for
rollback support
    }

    bool unite(int u, int v) {
        u = find(u), v = find(v);
        if (u == v) return false;

        // Union by size and record the merge for undo
        if (sz[u] < sz[v]) swap(u, v);
        par[v] = u;
        sz[u] += sz[v];
        st.emplace(u, v); // Save state to undo later
        --components;
        return true;
    }

    int size(int u) {
        return sz[find(u)];
    }

    // Mark a rollback checkpoint (non-merge action)
    void checkpoint() {
        st.emplace(-1, -1);
    }

    // Undo all merges until last checkpoint
    void undo() {
        while (!st.empty() && st.top().first != -1) {
            auto [u, v] = st.top(); st.pop();
            sz[u] -= sz[v];
            par[v] = v;
            ++components;
        }
        if (!st.empty()) st.pop(); // Pop the marker
```

```
    }
};
```

DSU.cpp

```
// Disjoint Set Union (Union-Find) with path compression and union by size
// Used to manage disjoint sets efficiently (e.g., Kruskal's MST, connectivity
queries, cycle detection)
// Time Complexity: Nearly constant amortized per operation, O(α(N)), where α
is inverse Ackermann

struct DSU {
    vector<int> par, sz;
    int groups; // Number of current disjoint sets

    DSU(int n) : par(n + 1), sz(n + 1, 1), groups(n) {
        iota(par.begin(), par.end(), 0);
             // Initialize each node as its own parent
    }

    int find(int u) {
        return par[u] == u ? u : par[u] = find(par[u]); // Path compression
    }

    bool unite(int u, int v) {
        u = find(u), v = find(v);
        if (u == v) return false; // Already in same group
        if (sz[v] > sz[u]) swap(u, v); // Union by size
        par[v] = u;
        sz[u] += sz[v];
        --groups;
        return true;
    }

    int size(int u) {
        return sz[find(u)]; // Size of the set containing u
    }
};
```

MO with update.cpp

```
// ========================================
//   Mo's Algorithm with Updates (Offline)
// ========================================
// ➤ Supports answering range queries (e.g., frequency/count/sum queries)
with intermediate point updates.
// ➤ Time Complexity: O(Q * n^(2/3)) – Efficient when updates and queries are
mixed.
// ➤ Best Use Cases:
//   - Static array with interleaved point updates and queries.
//   - Requires all queries known in advance (offline).
// ========================================

#pragma GCC optimize("O3")
#pragma GCC target("sse4")
#include <bits/stdc++.h>
using namespace std;

using ll = long long;

const int N = 1e5 + 5;
const int M = 2 * N;       // Upper bound on values after compression
const int blk = 2155;      // Block size ~ N^(2/3)
const int mod = 1e9 + 7;

int a[N], b[N];            // a: working array, b: snapshot array
int cnt1[M];               // Frequency of each value
int cnt2[N];               // How many numbers have a given frequency

int L = 0, R = -1, K = -1; // Pointers for Mo’s algorithm

// ========================
//      Query Definition
// ========================
struct Query {
    int l, r, t, idx;
    Query(int a = 0, int b = 0, int c = 0, int d = 0) : l(a), r(b), t(c),
idx(d) {}

    // Mo's comparator with time dimension
    bool operator < (const Query &o) const {
        if (r / blk != o.r / blk) return r < o.r;
        if (l / blk != o.l / blk) return l < o.l;
        return t < o.t;
    }
} Q[N];

map<int, int> id;          // Coordinate compression
int cnt = 0;               // Compressed value count
int ans[N];                // Answer array

int p[N], nxt[N], prv[N];  // Update history: index, new value, old value

// ========================
```

```
//     Frequency Handlers
// ==========================
void add(int x) {
    cnt2[cnt1[x]]--;
    cnt1[x]++;
    cnt2[cnt1[x]]++;
}

void del(int x) {
    cnt2[cnt1[x]]--;
    cnt1[x]--;
    cnt2[cnt1[x]]++;
}

// ==========================
//     Apply/Undo Update
// ==========================
void apply_update(int idx) {
    if (p[idx] >= L && p[idx] <= R)
        del(a[p[idx]]), add(nxt[idx]);
    a[p[idx]] = nxt[idx];
}

void undo_update(int idx) {
    if (p[idx] >= L && p[idx] <= R)
        del(a[p[idx]]), add(prv[idx]);
    a[p[idx]] = prv[idx];
}

int main() {
    int n, q;
    scanf("%d%d", &n, &q);

    // ========== Input & Compression ==========
    for (int i = 0; i < n; i++) {
        scanf("%d", a + i);
        if (!id.count(a[i]))
            id[a[i]] = cnt++;
        a[i] = id[a[i]];
        b[i] = a[i];
    }

    int qIdx = 0; // number of queries
    int ord = 0;  // number of updates

    // ========== Read Queries & Updates ==========
    while (q--) {
        int tp, l, r;
        scanf("%d", &tp);
        if (tp == 1) {
            // Query: range [l, r]
            scanf("%d%d", &l, &r);
            --l, --r;
            Q[qIdx] = Query(l, r, ord - 1, qIdx);
            qIdx++;
```

```
        } else {
            // Update: change p -> nxt
            scanf("%d%d", &p[ord], &nxt[ord]);
            --p[ord];
            if (!id.count(nxt[ord]))
                id[nxt[ord]] = cnt++;
            nxt[ord] = id[nxt[ord]];
            prv[ord] = b[p[ord]];
            b[p[ord]] = nxt[ord];
            ++ord;
        }
    }

    // ========== Process Queries ==========
    sort(Q, Q + qIdx);

    for (int i = 0; i < qIdx; i++) {
        while (L < Q[i].l) del(a[L++]);
        while (L > Q[i].l) add(a[--L]);
        while (R < Q[i].r) add(a[++R]);
        while (R > Q[i].r) del(a[R--]);

        while (K < Q[i].t) apply_update(++K);
        while (K > Q[i].t) undo_update(K--);

        // Example: answer = max frequency
        int res = 0;
        while (cnt2[res + 1] > 0) ++res;
        ans[Q[i].idx] = res;
    }

    // ========== Output Answers ==========
    for (int i = 0; i < qIdx; i++)
        printf("%d\n", ans[i]);

    return 0;
}
```

MO.cpp

```
// ========================== Mo's Algorithm ==========================
// ➤ Purpose: Efficiently answer multiple offline range queries on a static
array.
// ➤ Time Complexity: O((N + Q) * sqrt(N)) ≈ O(N * sqrt(N))
// ➤ Best for: Sum/frequency problems on subarrays (e.g., distinct count,
frequency, XOR).
// ➤ Restrictions:
//     - No updates allowed (only queries).
//     - Queries must be known in advance (offline).
// ====================================================================
const int BLOCK_SIZE = 316; // ≈ sqrt(N), can be tuned

// Comparator for optimal query ordering (Hilbert Curve / Mo Order)
// Z-ordering: minimize pointer movement by adjusting r direction per block
bool cmp(pair<int, int> p, pair<int, int> q) {
    if (p.first / BLOCK_SIZE != q.first / BLOCK_SIZE)
        return p < q;
    return ((p.first / BLOCK_SIZE) & 1) ? (p.second < q.second) : (p.second >
q.second);
}

// Structure for queries
struct Query {
    int l, r, idx;
};

// You must define this according to your problem
void add(int idx);            // Add element at idx to your data structure
void remove(int idx);         // Remove element at idx
int get_answer();             // Return current answer from your structure

vector<int> mos_algorithm(vector<Query>& queries) {
    vector<int> answers(queries.size());

    // Sort queries using improved Mo's comparator
    sort(queries.begin(), queries.end(), [&](const Query& a, const Query& b) {
        return cmp({a.l, a.r}, {b.l, b.r});
    });

    int cur_l = 0, cur_r = -1;
    // Maintain [cur_l, cur_r] as the current active segment
    for (auto q : queries) {
        while (cur_l > q.l) add(--cur_l);
        while (cur_r < q.r) add(++cur_r);
        while (cur_l < q.l) remove(cur_l++);
        while (cur_r > q.r) remove(cur_r--);
        answers[q.idx] = get_answer(); // Answer for current query
    }

    return answers;
}
```

Sparse Table.cpp

```
//The sparse table is optimized for idempotent operations like GCD, min, max
(not sum)
struct Sparse
{
    vector<int> lg;                // Stores precomputed floor(log2(i)) values
    vector<vector<int>> table;        // Sparse table for range GCD queries
    // Constructor to initialize the sparse table for array 'a' of size 'n'
    Sparse(int n, vector<int>& a)
    {
        lg.resize(n + 5);
        lg[1] = 0;

        // Precompute floor(log2(i)) for i in [2, n]
        for (int i = 2; i <= n; i++)
            lg[i] = lg[i / 2] + 1;


        // Initialize the sparse table with dimensions [n][log2(n)]
        table = vector<vector<int>>(n + 5, vector<int>(lg[n] + 5, 0));

        // Base case: the 0-th level stores the original array values
        for (int i = 0; i < n; i++)
            table[i][0] = a[i];


        // Build the sparse table using dynamic programming
        // table[i][j] holds GCD of the subarray starting at i with length 2^j
        for (int j = 1; j <= lg[n]; j++)
        {
            for (int k = 0; k + (1 << j) - 1 < n; k++)
            {
                table[k][j] = __gcd(
                    table[k][j - 1],                        // First half
                    table[k + (1 << (j - 1))][j - 1]       // Second half
                );
            }
        }
    }

    // Query the GCD of the subarray [l, r] (0-based indexing)
    int query(int l, int r)
    {
        int len = r - l + 1;
        int x = lg[len];  // log2 of the segment length
        return __gcd(
            table[l][x],                                // GCD of first
segment
            table[r - (1 << x) + 1][x]                  // GCD of second
segment
        );
    }
};
```

Treap.cpp

```
// Treap (Cartesian Tree) implementation
// Supports fast insertion, cyclic shift, reverse in range, and query by
index.
// Combines binary search tree (by implicit key = position) and heap (by
random priority).
// Time Complexity per operation: O(log N) on average.

// Node structure
struct node {
    int prior, cnt, val; // prior: priority (heap), cnt: subtree size, val:
stored value
    int lazy; // for lazy propagation (reversal flag)
    node *left, *right;

    node(int _val) : val(_val), prior(rand()), cnt(1), lazy(0), left(nullptr),
right(nullptr) {}
    ~node() { delete left; delete right; }
};

using nodePtr = node*;

class Treap {
    nodePtr root;

    int cnt(nodePtr a) {
        return a ? a->cnt : 0;
    }

    void update(nodePtr a) {
        if (a) a->cnt = 1 + cnt(a->left) + cnt(a->right);
    }

    // Propagate the lazy reversal flag
    void pushDown(nodePtr &a) {
        if (a && a->lazy) {
            swap(a->left, a->right);
            if (a->left) a->left->lazy ^= 1;
            if (a->right) a->right->lazy ^= 1;
            a->lazy = 0;
        }
    }

    // Split treap into: [0...key] and [key+1...]
    void split(nodePtr cur, nodePtr &l, nodePtr &r, int key, int curKey = 0) {
        if (!cur) return void(l = r = nullptr);
        pushDown(cur);

        int realKey = curKey + cnt(cur->left) + 1;
        if (key >= realKey) {
            split(cur->right, cur->right, r, key, realKey);
            l = cur;
        } else {
            split(cur->left, l, cur->left, key, curKey);
```

```
            r = cur;
        }
        update(cur);
    }

    // Merge two treaps
    void merge(nodePtr &cur, nodePtr l, nodePtr r) {
        pushDown(l);
        pushDown(r);
        if (!l || !r)
            cur = l ? l : r;
        else if (l->prior > r->prior) {
            merge(l->right, l->right, r);
            cur = l;
        } else {
            merge(r->left, l, r->left);
            cur = r;
        }
        update(cur);
    }

public:
    Treap() : root(nullptr) {}
    ~Treap() { delete root; }

    // Insert value v at position pos (1-based)
    void insert(int pos, int v) {
        nodePtr a, b, c = new node(v);
        split(root, a, b, pos - 1);
        merge(a, a, c);
        merge(root, a, b);
    }

    // Cyclic right shift of subarray [l, r]
    void cyclic_shift(int l, int r) {
        nodePtr a, b, c, d;
        split(root, a, b, r);
        split(a, a, c, r - 1);
        split(a, d, a, l - 1);
        merge(a, c, a);     // bring last element to front
        merge(a, a, b);
        merge(root, d, a);
    }

    // Reverse subarray [l, r]
    void reverse(int l, int r) {
        nodePtr a, b, c;
        split(root, a, b, r);
        split(a, c, a, l - 1);
        a->lazy ^= 1;       // mark for lazy reverse
        merge(a, c, a);
        merge(root, a, b);
    }

    // Return the value at index i (1-based)
```

```
    int at(int i) {
        nodePtr cur = root;
        int curKey = cnt(cur->left) + 1;
        while (curKey != i) {
            pushDown(cur);
            if (i > curKey) {
                cur = cur->right;
                curKey += cnt(cur->left) + 1;
            } else {
                cur = cur->left;
                curKey -= cnt(cur->right) + 1;
            }
        }
        pushDown(cur);
        return cur->val;
    }
};

int main() {
#ifndef ONLINE_JUDGE
    freopen("input.in", "r", stdin);
#endif
    // Input:
    // n: number of elements
    // q: number of operations
    // m: number of queries
    int n, q, m, x;
    scanf("%d %d %d", &n, &q, &m);
    Treap t;

    for (int i = 1; i <= n; ++i) {
        scanf("%d", &x);
        t.insert(i, x); // build the treap
    }

    // q operations: type 1 = cyclic shift, type 2 = reverse
    int type, l, r;
    while (q--) {
        scanf("%d %d %d", &type, &l, &r);
        if (type == 1)
            t.cyclic_shift(l, r);
        else
            t.reverse(l, r);
    }

    // m index queries
    for (int i = 0; i < m; ++i) {
        scanf("%d", &x);
        printf("%d ", t.at(x));
    }
}
```

Wavelet tree.cpp

```
// Purpose:
// Efficiently answers how many numbers in range [l, r] are ≥ k.
//
// Core Use Cases (Catalan style explanation):
// 1. Range frequency queries (e.g., how many ≥ k, ≤ k, = k in [l, r])
// 2. K-th smallest/largest in a range
// 3. Range median queries
// 4. Solving static offline queries in O(log max_element) per query
//
// Time Complexity: O(log(max_element))
// Space Complexity: O(n log(max_element))


// Wavelet Tree: Count numbers ≥ k in range [l, r]

#include <bits/stdc++.h>
using namespace std;

typedef vector<int> vi;
typedef long long ll;

const int N = 3e4 + 5;

struct node {
    vi arr, freq;
    int mn, mx, md;
    node *left = nullptr, *right = nullptr;

    node() {
        arr = {0};  // dummy to make it 1-indexed
    }

    // Builds the wavelet tree recursively
    void build() {
        mn = *min_element(arr.begin() + 1, arr.end());
        mx = *max_element(arr.begin() + 1, arr.end());
        if (mn == mx) return;

        freq = vi(arr.size(), 0);
        left = new node();
        right = new node();
        md = mn + (mx - mn) / 2;

        for (int i = 1; i < arr.size(); ++i) {
            if (arr[i] <= md) {
                left->arr.push_back(arr[i]);
                freq[i] = freq[i - 1] + 1;
            } else {
                right->arr.push_back(arr[i]);
                freq[i] = freq[i - 1];
            }
        }
```

```
        left->build();
        right->build();
    }

    // Returns number of elements in [s, e] that are ≥ k
    int query(int s, int e, int k) {
        if (mn == mx) return mn >= k ? (e - s + 1) : 0;

        int cnt = freq[e] - freq[s - 1];  // # elements sent to left in [s,e]
        int res = 0;

        if (k <= md) {
            // Elements in right child are ≥ k
            res += (e - s + 1) - cnt;
            if (cnt) {
                int new_s = freq[s - 1] + 1;
                int new_e = freq[e];
                res += left->query(new_s, new_e, k);
            }
        } else {
            if ((e - s + 1) - cnt) {
                int new_s = s - freq[s - 1];
                int new_e = e - freq[e];
                res = right->query(new_s, new_e, k);
            }
        }
        return res;
    }
};

int main() {
#ifndef ONLINE_JUDGE
    freopen("input.in", "r", stdin);
#endif

    int n, x;
    scanf("%d", &n);
    node root;

    for (int i = 0; i < n; ++i) {
        scanf("%d", &x);
        root.arr.push_back(x);
    }

    root.build();

    int q, s, e, k;
    scanf("%d", &q);
    while (q--) {
        scanf("%d %d %d", &s, &e, &k);
        printf("%d\n", root.query(s, e, k));
    }
}
```

ordered_Sets.cpp

```
#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, less<T>, rb_tree_tag,
tree_order_statistics_node_update>;

template<typename T>
using ordered_multiset = tree<T, null_type, less_equal<T>, rb_tree_tag,
tree_order_statistics_node_update>;

template <class T>
void erase_one(ordered_multiset<T>& ms, const T& val) {
    auto it = ms.lower_bound(val);
    if (it != ms.end() && *it == val) {
        ms.erase(it);
    }
}
```

segtree.cpp

```
struct Node
{
    // Customize according to the problem (e.g., sum, min, max, etc.)
} NEUTRAL; // Neutral value for merging (e.g., 0 for sum, INF for min)

struct SegTree
{
    int size;
    vector<Node> tree;
    vector<int> lazy;

    SegTree(int n)
    {
        size = 1;
        while (size < n) size *= 2;
        tree.resize(2 * size);
        lazy.assign(2 * size, 0);
    }

    Node merge(Node a, Node b)
    {
        Node res;
        // Combine a and b into res
        return res;
    }

    void build(vector<int>& a, int x, int lx, int rx)
    {
        if (rx - lx == 1)
        {
            if (lx < (int)a.size())
```

```
            {
                // Initialize leaf from array
            }
            return;
        }
        int m = (lx + rx) / 2;
        build(a, 2 * x + 1, lx, m);
        build(a, 2 * x + 2, m, rx);
        tree[x] = merge(tree[2 * x + 1], tree[2 * x + 2]);
    }

    void push(int x, int lx, int rx)
    {
        if (lazy[x])
        {
            // Apply and propagate lazy update
            lazy[x] = 0;
        }
    }

    void update(int l, int r, int x, int lx, int rx)
    {
        push(x, lx, rx);
        if (lx >= r || rx <= l) return;
        if (lx >= l && rx <= r)
        {
            lazy[x] = 1;
            push(x, lx, rx);
            return;
        }
        int m = (lx + rx) / 2;
        update(l, r, 2 * x + 1, lx, m);
        update(l, r, 2 * x + 2, m, rx);
        tree[x] = merge(tree[2 * x + 1], tree[2 * x + 2]);
    }

    Node query(int l, int r, int x, int lx, int rx)
    {
        push(x, lx, rx);
        if (lx >= r || rx <= l) return NEUTRAL;
        if (lx >= l && rx <= r) return tree[x];
        int m = (lx + rx) / 2;
        return merge(query(l, r, 2 * x + 1, lx, m), query(l, r, 2 * x + 2, m,
rx));
    }

    void build(vector<int>& a) { build(a, 0, 0, size); }
    void update(int l, int r) { update(l, r, 0, 0, size); }
    Node query(int l, int r) { return query(l, r, 0, 0, size); }
};
```

Segtree with lazy.cpp

```
struct SegmentTree {
    int n;
    vector<long long> seg, lazy;

    // Constructor: Initialize with size n (array size)
    SegmentTree(int size) {
        n = size;
        seg.assign(4 * n, 0);   // segment tree
        lazy.assign(4 * n, 0);  // lazy propagation array
    }

    // Build the segment tree from input array `arr`
    // node: current node index in seg[], [l, r] is the range it covers
    void build(const vector<long long>& arr, int node, int l, int r) {
        if (l == r) {
            seg[node] = arr[l];
            return;
        }
        int mid = (l + r) / 2;
        build(arr, node * 2, l, mid);
        build(arr, node * 2 + 1, mid + 1, r);
        seg[node] = seg[node * 2] + seg[node * 2 + 1];
    }

    // Pushes lazy value at node to its children (if not a leaf)
    // Updates seg[node] and pushes the pending updates down
    void push(int ind, int low, int high) {
        if (lazy[ind] != 0) {
            seg[ind] += (high - low + 1) * lazy[ind];  // apply update
            if (low != high) {  // not a leaf -> push to children
                lazy[2 * ind + 1] += lazy[ind];
                lazy[2 * ind + 2] += lazy[ind];
            }
            lazy[ind] = 0;  // clear the current node's lazy
        }
    }

    // Range increment update: add `val` to range [l, r]
    void rangeUpdate(int ind, int low, int high, int l, int r, long long val)
{
        push(ind, low, high);  // apply any pending updates

        // No overlap
        if (r < low || l > high || low > high) return;

        // Complete overlap: apply and propagate lazily
        if (l <= low && high <= r) {
            seg[ind] += (high - low + 1) * val;
            if (low != high) {
                lazy[2 * ind + 1] += val;
                lazy[2 * ind + 2] += val;
            }
            return;
```

```
        }

        // Partial overlap: go deeper
        int mid = (low + high) / 2;
        rangeUpdate(2 * ind + 1, low, mid, l, r, val);
        rangeUpdate(2 * ind + 2, mid + 1, high, l, r, val);
        seg[ind] = seg[2 * ind + 1] + seg[2 * ind + 2];
    }

    // Query sum in range [l, r] with lazy propagation
    long long querySumLazy(int ind, int low, int high, int l, int r) {
        push(ind, low, high);  // apply any pending updates

        // No overlap
        if (r < low || l > high || low > high) return 0;

        // Complete overlap
        if (l <= low && high <= r) return seg[ind];

        // Partial overlap
        int mid = (low + high) / 2;
        return querySumLazy(2 * ind + 1, low, mid, l, r) +
            querySumLazy(2 * ind + 2, mid + 1, high, l, r);
    }

    // Point update: set arr[idx] = val
    void update(int node, int l, int r, int idx, long long val) {
        if (l == r) {
            seg[node] = val;
            return;
        }
        int mid = (l + r) / 2;
        if (idx <= mid) update(node * 2, l, mid, idx, val);
        else update(node * 2 + 1, mid + 1, r, idx, val);
        seg[node] = seg[node * 2] + seg[node * 2 + 1];
    }

    // Range query (without lazy) for verification or fallback
    long long query(int node, int l, int r, int ql, int qr) {
        if (ql > r || qr < l || r < l) return 0;
        if (ql <= l && qr >= r) return seg[node];
        int mid = (l + r) / 2;
        long long query1 = query(node * 2, l, mid, ql, qr);
        long long query2 = query(node * 2 + 1, mid + 1, r, ql, qr);
        return (query1 + query2);
    }
};
```

◇ Geometry

Closest Pair.cpp

```
// Closest Pair of Points in 2D Plane – O(n log n)
// Returns the **squared distance** between the closest pair (to avoid
floating point errors)
// Input: Vector of points as pairs (x, y)
// Assumes: All points are unique

long long ClosestPair(vector<pair<int, int>> pts) {
    int n = pts.size();
    sort(pts.begin(), pts.end());  // Sort by x-coordinate

    set<pair<int, int>> s;  // Balanced BST ordered by y-coordinate, stores
(y, x)

    long long best_dist = LLONG_MAX;
    int j = 0;

    for (int i = 0; i < n; ++i) {
        int d = ceil(sqrt(best_dist));  // Max x-distance for current best

        // Remove points too far in x from current point
        while (j < n && pts[i].first - pts[j].first >= d) {
            s.erase({ pts[j].second, pts[j].first });
            j++;
        }

        // Find points in set with y within [y-d, y+d]
        auto it1 = s.lower_bound({ pts[i].second - d, -1e9 });
        auto it2 = s.upper_bound({ pts[i].second + d, +1e9 });

        // Check only relevant candidates (in y-range)
        for (auto it = it1; it != it2; ++it) {
            int dx = pts[i].first - it->second;
            int dy = pts[i].second - it->first;
            best_dist = min(best_dist, 1LL * dx * dx + 1LL * dy * dy);
        }

        // Insert current point into set for future comparisons
        s.insert({ pts[i].second, pts[i].first });
    }

    return best_dist;  // Returns the squared minimum distance
}

Convex Hull Trick.cpp
#define ll long long
struct Line
{
    mutable ll k, m, p;
    bool operator<(const Line &o) const { return k < o.k; }
    bool operator<(ll x) const { return p < x; }
```

```
};


// for a line equation y = k*x+m
// if you have multible lines and you want to query max y for some x
// to turn this into min y for some x insert -k and -m
struct LineContainer : multiset<Line, less<>>
{
    // (for doubles, use inf = 1/.0, div(a,b) = a/b)
    static const ll inf = LLONG_MAX;
    ll div(ll a, ll b)
    { // floored division
        return a / b - ((a ^ b) < 0 && a % b);
    }
    bool isect(iterator x, iterator y)
    {
        if (y == end())
            return x->p = inf, 0;
        if (x->k == y->k)
            x->p = x->m > y->m ? inf : -inf;
        else
            x->p = div(y->m - x->m, x->k - y->k);
        return x->p >= y->p;
    }
    void add(ll k, ll m)
    {
        auto z = insert({k, m, 0}), y = z++, x = y;
        while (isect(y, z))
            z = erase(z);
        if (x != begin() && isect(--x, y))
            isect(x, y = erase(y));
        while ((y = x) != begin() && (--x)->p >= y->p)
            isect(x, erase(y));
    }
    ll query(ll x)
    {
        assert(!empty());
        auto l = *lower_bound(x);
        return l.k * x + l.m;
    }
} ls;
```

Convex Hull.cpp

```
struct Point {
    int x, y;

    bool operator<(const Point& p) const {
        return x < p.x || (x == p.x && y < p.y);
    }
};

int cross(const Point& a, const Point& b, const Point& c) {
    return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
}

bool isRightTurn(const Point& a, const Point& b, const Point& c) {
    return cross(a, b, c) < 0;
}

bool isLeftTurnOrColinear(const Point& a, const Point& b, const Point& c) {
    return cross(a, b, c) >= 0;
}
signed main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    cin >> n;
    vector<Point> pts(n);
    for (auto& p : pts)
        cin >> p.x >> p.y;

    // Sort by x then y to ensure deterministic order
    sort(pts.begin(), pts.end());

    // Hull will be built incrementally
    vector<Point> hull;

    for (int i = 0; i < n; ++i) {
        // Lower hull
        while (hull.size() >= 2 && !isLeftTurnOrColinear(hull[hull.size() -
2], hull.back(), pts[i])) {
            hull.pop_back();
        }
        hull.push_back(pts[i]);
    }

    // Upper hull (add in reverse)
    int lower_size = hull.size();
    for (int i = n - 2; i >= 0; --i) {
        while (hull.size() > lower_size &&
!isLeftTurnOrColinear(hull[hull.size() - 2], hull.back(), pts[i])) {
            hull.pop_back();
        }
        hull.push_back(pts[i]);
    }
```

```
    // Remove the duplicate start/end point
    if (hull.size() > 1 && hull.front().x == hull.back().x && hull.front().y
== hull.back().y)
        hull.pop_back();

    // Output the convex hull
    cout << hull.size() << "\n";
    for (auto& p : hull) {
        cout << p.x << " " << p.y << "\n";
    }

    return 0
```

Geometry Stuff.cpp

```
ftype dot(point2d a, point2d b) {
    return a.x * b.x + a.y * b.y;
}
ftype dot(point3d a, point3d b) {
    return a.x * b.x + a.y * b.y + a.z * b.z;
}

ftype norm(point2d a) {
    return dot(a, a);
}

double abs(point2d a) {
    return sqrt(norm(a));
}

double proj(point2d a, point2d b) {
    return dot(a, b) / abs(b);
}

double angle(point2d a, point2d b) {
    return acos(dot(a, b) / abs(a) / abs(b));
}

ftype cross(point2d a, point2d b) {
    return a.x * b.y - a.y * b.x;
}

point2d intersect(point2d a1, point2d d1, point2d a2, point2d d2) {
    return a1 + cross(a2 - a1, d2) / cross(d1, d2) * d1;
}

point3d intersect(point3d a1, point3d n1, point3d a2, point3d n2, point3d a3,
point3d n3) {
    point3d x(n1.x, n2.x, n3.x);
    point3d y(n1.y, n2.y, n3.y);
    point3d z(n1.z, n2.z, n3.z);
    point3d d(dot(a1, n1), dot(a2, n2), dot(a3, n3));
    return point3d(triple(d, y, z),
                   triple(x, d, z),
                   triple(x, y, d)) / triple(n1, n2, n3);
}

struct pt {
    long long x, y;
    pt() {}
    pt(long long _x, long long _y) : x(_x), y(_y) {}
    pt operator-(const pt& p) const { return pt(x - p.x, y - p.y); }
    long long cross(const pt& p) const { return x * p.y - y * p.x; }
    long long cross(const pt& a, const pt& b) const { return (a -
*this).cross(b - *this); }
};

int sgn(const long long& x) { return x >= 0 ? x ? 1 : 0 : -1; }
```

```
bool inter1(long long a, long long b, long long c, long long d) {
    if (a > b)
        swap(a, b);
    if (c > d)
        swap(c, d);
    return max(a, c) <= min(b, d);
}

bool check_inter(const pt& a, const pt& b, const pt& c, const pt& d) {
    if (c.cross(a, d) == 0 && c.cross(b, d) == 0)
        return inter1(a.x, b.x, c.x, d.x) && inter1(a.y, b.y, c.y, d.y);
    return sgn(a.cross(b, c)) != sgn(a.cross(b, d)) &&
           sgn(c.cross(d, a)) != sgn(c.cross(d, b));
}
```

LineSegement Intersection.cpp

```
struct Point {
    int x, y;
};
// To find the orientation of ordered triplet (p, q, r).
// 0 --> colinear, 1 --> clockwise, 2 --> counterclockwise
int orientation(Point p, Point q, Point r) {
    int val = (q.y - p.y) * (r.x - q.x) -
        (q.x - p.x) * (r.y - q.y);

    if (val == 0) return 0;            // colinear
    return (val > 0) ? 1 : 2;          // clock or counterclock wise
}

// Check if point r lies on segment pq (used for colinear case)
bool onSegment(Point p, Point q, Point r) {
    return r.x <= max(p.x, q.x) && r.x >= min(p.x, q.x) &&
        r.y <= max(p.y, q.y) && r.y >= min(p.y, q.y);
}

bool doIntersect(Point p1, Point p2, Point q1, Point q2) {
    int o1 = orientation(p1, p2, q1);
    int o2 = orientation(p1, p2, q2);
    int o3 = orientation(q1, q2, p1);
    int o4 = orientation(q1, q2, p2);

    // General case
    if (o1 != o2 && o3 != o4)
        return true;

    // Special Cases
    if (o1 == 0 && onSegment(p1, p2, q1)) return true;
    if (o2 == 0 && onSegment(p1, p2, q2)) return true;
    if (o3 == 0 && onSegment(q1, q2, p1)) return true;
    if (o4 == 0 && onSegment(q1, q2, p2)) return true;

    return false;
}
```

Point to polygon.cpp

```
 struct Point {
    int x, y;
    void read() { cin >> x >> y; }
    Point operator +(const Point& b) const { return Point{ x + b.x, y + b.y };
}
    Point operator -(const Point& b) const { return Point{ x - b.x, y - b.y };
}
    ll operator *(const Point& b) const { return (ll)x * b.y - (ll)y * b.x; }
    void operator +=(const Point& b) { x += b.x; y += b.y; }
    void operator -=(const Point& b) { x -= b.x; y -= b.y; }
    void operator *=(const int k) { x *= k; y *= k; }

    ll cross(const Point& b, const Point& c) const {
        return (b - *this) * (c - *this);
    }
};
vector<Point>P(1001);

bool pointlineintersect(Point P1, Point P2, Point P3) {
    if (P2.cross(P1, P3) != 0)   return false;
    return (min(P2.x, P3.x) <= P1.x && P1.x <= max(P2.x, P3.x))
        && (min(P2.y, P3.y) <= P1.y && P1.y <= max(P2.y, P3.y));
}

void pointinpolygon() {
    int cnt = 0;
    bool boundary = false;
    for (int i = 1; i <= N; i++) {
        int j = (i == N ? 1 : i + 1);
        if (pointlineintersect(P[0], P[i], P[j]))
            boundary = true;

        if (P[i].x <= P[0].x && P[0].x < P[j].x && P[0].cross(P[i], P[j]) < 0)
cnt++;
        else if (P[j].x <= P[0].x && P[0].x < P[i].x && P[0].cross(P[j], P[i])
< 0)  cnt++;
    }

    if (boundary)    printf("BOUNDARY\n");
    else if (cnt & 1)  printf("INSIDE\n");
    else            printf("OUTSIDE\n");
}
```

Polygon lattance.cpp

```
// Point Struct + Polygon Area + Boundary Integer Points using GCD
// Interior points = (Area + 2 - Boundary) / 2   [Pick's Theorem]
// Area is 2×actual area (unscaled), so result is still integer

struct Point {
    int x, y;

    void read() { cin >> x >> y; }

    Point operator +(const Point& b) const { return { x + b.x, y + b.y }; }
    Point operator -(const Point& b) const { return { x - b.x, y - b.y }; }

    ll operator *(const Point& b) const {  // Cross product
        return (ll)x * b.y - (ll)y * b.x;
    }

    void operator +=(const Point& b) { x += b.x; y += b.y; }
    void operator -=(const Point& b) { x -= b.x; y -= b.y; }
    void operator *=(const int k) { x *= k; y *= k; }

    // Cross product of (b - a) × (c - a)
    ll cross(const Point& b, const Point& c) const {
        return (b - *this) * (c - *this);
    }
};

// Cross product for signed area
int cross(Point a, Point b) {
    return a.x * b.y - a.y * b.x;
}

// Twice the area of polygon using shoelace formula (always positive)
int polygonArea(const vector<Point>& points) {
    int n = points.size();
    int total = 0;
    for (int i = 0; i < n; ++i) {
        const Point& a = points[i];
        const Point& b = points[(i + 1) % n];
        total += cross(a, b);
    }
    return abs(total);
}

signed main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    cin >> n;
    vector<Point> p(n);
    for (auto& pt : p) pt.read();

    int area = polygonArea(p);  // 2 × actual area
```

```
    int boundary = 0;
    for (int i = 0; i < n; i++) {
        auto a = p[i];
        auto b = p[(i + 1) % n];
        int dx = abs(a.x - b.x);
        int dy = abs(a.y - b.y);
        boundary += gcd(dx, dy);  // # of integer points on edge
    }

    int interior = (area + 2 - boundary) / 2;
    cout << interior << " " << boundary << "\n";
}
```

farthest Manhaten.cpp

```
// Find for D-dimensional

long long ans = 0;
for (int msk = 0; msk < (1 << d); msk++) {
    long long mx = LLONG_MIN, mn = LLONG_MAX;
    for (int i = 0; i < n; i++) {
        long long cur = 0;
        for (int j = 0; j < d; j++) {
            if (msk & (1 << j)) cur += p[i][j];
            else cur -= p[i][j];
        }
        mx = max(mx, cur);
        mn = min(mn, cur);
    }
    ans = max(ans, mx - mn);
}
```

◇ Graphs

BFS escape.cpp

```
#include <bits/stdc++.h>

using namespace std;

const int N = 1e3 + 5, M = 1e3 + 5, OO = 0x3f3f3f3f;

int n, m, mR, mC;
char grid[N][M];

bool isValid(int r, int c) {
  return r < n&& r >= 0 && c < m&& c >= 0;
}

int dR[] = { 1, -1, 0, 0 };
int dC[] = { 0, 0, 1, -1 };
int catDis[N][M];
void BFSCats() {
  memset(catDis, OO, sizeof catDis);

  queue <pair<int, int>> q;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      if (grid[i][j] == 'C') {
        q.push({ i, j });
        catDis[i][j] = 0;
      }
    }
  }

  while (!q.empty()) {
    pair<int, int> u = q.front();
    q.pop();
    for (int k = 0; k < 4; k++) {
      int nR = u.first + dR[k];
      int nC = u.second + dC[k];
      if (isValid(nR, nC) && catDis[nR][nC] == OO && grid[nR][nC] != '#') {
        catDis[nR][nC] = catDis[u.first][u.second] + 1;
        q.push({ nR, nC });
      }
    }
  }
}

pair<int, int> parent[N][M];
pair<int, int> ext;
int mouseDis[N][M];
bool BFSMouse(int mR, int mC) {
  memset(mouseDis, OO, sizeof mouseDis);

  queue <pair<int, int>> q;
```

```
  q.push({ mR, mC });
  mouseDis[mR][mC] = 0;

  while (!q.empty()) {
    pair<int, int> u = q.front();
    q.pop();
    for (int k = 0; k < 4; k++) {
      int nR = u.first + dR[k];
      int nC = u.second + dC[k];
      if (isValid(nR, nC) && mouseDis[nR][nC] == OO && grid[nR][nC] != '#' &&
catDis[nR][nC] > mouseDis[u.first][u.second] + 1) {
        parent[nR][nC] = u;
        if (grid[nR][nC] == 'E') {
          ext = { nR, nC };
          return true;
        }

        mouseDis[nR][nC] = mouseDis[u.first][u.second] + 1;
        q.push({ nR, nC });
      }
    }
  }
  return false;
}

void printPath(int r, int c) {
  if (r == mR && c == mC) {
    printf("%d %d\n", r, c);
    return;
  }
  printPath(parent[r][c].first, parent[r][c].second);
  printf("%d %d\n", r, c);
}

int main() {
  scanf("%d %d", &n, &m);
  for (int i = 0; i < n; i++) {
    scanf("%s", grid[i]);
    for (int j = 0; j < m; j++) {
      if (grid[i][j] == 'M') {
        mR = i;
        mC = j;
      }
    }
  }
  BFSCats();
  bool canEscape = BFSMouse(mR, mC);
  puts(BFSMouse(mR, mC) ? "YES" : "NO");
  if (canEscape) printPath(ext.first, ext.second);
  return 0;
}
```

Graph Coloring

```
#include <bits/stdc++.h>

using namespace std;

const int NOT_COLORED = 0, RED = 1, BLUE = 2;

const int N = 1e5 + 5, M = 2e5 + 5, OO = 0x3f3f3f3f;

int n, m, u, v;

vector<int> adj[N];

int color[N];
bool isBiColorable(int src) {
  memset(color, NOT_COLORED, sizeof color);
  color[src] = RED;
  queue<int> q;
  q.push(src);

  while (!q.empty()) {
    int u = q.front();
    q.pop();
    for (int v : adj[u]) {
      if (color[v] == NOT_COLORED) {
        color[v] = (color[u] == RED ? BLUE : RED);
        q.push(v);
      } else if (color[v] == color[u]) {
        return false;
      }
    }
  }
  return true;
}

int main() {
  scanf("%d %d", &n, &m);
  for (int i = 0; i < m; i++) {
    scanf("%d %d", &u, &v);
    adj[u].push_back(v);
    adj[v].push_back(u);
  }
  puts(isBiColorable(0) ? "YES" : "NO");
  return 0;
}
```

Dijkstra

```
const int OO = 0x3f3f3f3f;

void Dijkstra_nlogn(int src, vector<vector<pair<int, int>>> adjList) {
    int n = adjList.size();

    vector<int> dis(n, OO);
    dis[src] = 0;

    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<int>> pq;
    pq.push({0, src});

    while (!pq.empty()) {
        auto [nodeDis, node] = pq.top();
        pq.pop();

        if (nodeDis != dis[node])
            continue;

        for (auto [child, weight] : adjList[node]) {
            if (dis[child] > dis[node] + weight) {
                dis[child] = dis[node] + weight;
                pq.push({dis[child] , child});
            }
        }
    }
}
```

DSU

```
int leader[N];
int sz[N];
void init()
{
  for (int i = 0; i <= n; i++) {
    leader[i] = i;
    sz[i] = 1;
  }
}

int getLeader(int u) // O(n)
{
  if (u == leader[u])
    return u;
  return leader[u] = getLeader(leader[u]);
}

bool areFriends(int u, int v)
{
  return getLeader(u) == getLeader(v);
}
```

```
void makeFriends(int u, int v)
{
  u = getLeader(u);
  v = getLeader(v);
  if (u == v)
    return;

  leader[u] = v;
  sz[v] += sz[u];
}
```

Floyd.cpp

```
int dist[N][N];
int next[N][N];
void path(int fr, int to)
{
    if (next[fr][to] == -1)
    {
        cout << fr << " "; // beytalla3 kolo ma 3ada a5er node
        return;
    }
    path(fr, next[fr][to]);
    path(next[fr][to], to);
}
void floyd()
{
    // dist[i][j] contains the weight of edge (i, j)
    // or INF (1B) if there is no such edge
    for (int k = 0; k < n; k++)
    {
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n; j++)
            {
                if (dist[i][k] + dist[k][j] < dist[i][j])
                {
                    dist[i][j] = dist[i][k] + dist[k][j];
                    next[i][j] = k;
                }
            }
        }
    }
}
```

LCA.cpp

```
const ll N = 4e5 + 5, INF = 1e9, mod = 1e9 + 7, LOG = 23;
int n, q, timer = 1;
vector<int> adj[N];
int ancestor[N][LOG], lvl[N], in[N], out[N];
ll flat[N], value[N];

struct LCA {

    void init(int root = 1) {
        timer = 1;
        BuildAncestors(root, 0, 0);
    }

    // Precomputes in-time, out-time, level, and binary lifting table
    void BuildAncestors(int node, int par, int depth) {
        ancestor[node][0] = par;
        lvl[node] = depth;
        in[node] = timer++;  // in-time for Euler tour
        flat[in[node]] = value[node];  // store value in Euler array

        // Fill ancestor table for binary lifting
        for (int bit = 1; bit < LOG; bit++) {
            int p = ancestor[node][bit - 1];
            ancestor[node][bit] = ancestor[p][bit - 1];
        }

        for (auto& v : adj[node]) {
            if (v == par) continue;
            BuildAncestors(v, node, depth + 1);
        }
        out[node] = timer - 1; // out-time for subtree range queries
    }

    // Returns the k-th ancestor of a node using binary lifting
    int KthAncestor(int node, int k) {
        for (int bit = LOG - 1; bit >= 0; bit--) {
            if ((k >> bit) & 1) {
                node = ancestor[node][bit];
            }
        }
        return node;
    }

    int get_kth_on_path(int u, int v, int k) { // the k-th node on the path
from u to v
        int lca = get_LCA(u, v);
        int d1 = lvl[u] - lvl[lca];
        if (k <= d1)
            return KthAncestor(u, k);
        int d2 = lvl[v] - lvl[lca];
        return KthAncestor(v, d1 + d2 - k);
    }
```

```
    int get_LCA(int u, int v) {
        if (lvl[u] < lvl[v]) swap(u, v);
        // Bring u to the same level as v
        u = KthAncestor(u, lvl[u] - lvl[v]);
        if (u == v) return u;
        // Binary lift both u and v until their parents match
        for (int bit = LOG - 1; bit >= 0; bit--) {
            if (ancestor[u][bit] != ancestor[v][bit]) {
                u = ancestor[u][bit];
                v = ancestor[v][bit];
            }
        }
        return ancestor[u][0];
    }

    // Returns the distance (number of edges) between u and v
    int dis(int u, int v) {
        int lca = get_LCA(u, v);
        return lvl[u] - lvl[lca] + lvl[v] - lvl[lca];
    }

    // Returns true if u is an ancestor of v (based on Euler tour times)
    bool isAncestor(int u, int v) {
        return in[u] <= in[v] && out[v] <= out[u];
    }

} lcaTool;
```

LCA with Lazy Segtree

```
#include <bits/stdc++.h>
#define ll long long
#define ld long double
#define FIO ios_base::sync_with_stdio(0), cin.tie(0), cout.tie(0);
using namespace std;
const ll N = 4e5 + 5, INF = 1e9, mod = 1e9 + 7, LOG = 23;
int n, q, timer = 1;
vector<int> adj[N];
int ancestor[N][LOG], lvl[N], in[N], out[N];
ll flat[N], seg[4 * N], lazy[4 * N], color[N];

struct LCA {

    void init(int root = 1) {
        timer = 1;
        BuildAncestors(root, 0, 0);
    }

    void BuildAncestors(int node, int par, int depth) {
        ancestor[node][0] = par;
        lvl[node] = depth;
        in[node] = timer++;
        flat[in[node]] = color[node];
        for (int bit = 1; bit < LOG; bit++) {
            int p = ancestor[node][bit - 1];
            ancestor[node][bit] = ancestor[p][bit - 1];
        }

        for (auto& v : adj[node]) {
            if (v == par) continue;
            BuildAncestors(v, node, depth + 1);
        }
        out[node] = timer - 1;
    }

    int KthAncestor(int node, int k) {
        for (int bit = LOG - 1; bit >= 0; bit--) {
            if ((k >> bit) & 1) {
                node = ancestor[node][bit];
            }
        }
        return node;
    }

    int get_LCA(int u, int v) {
        if (lvl[u] < lvl[v]) swap(u, v);
        u = KthAncestor(u, lvl[u] - lvl[v]);
        if (u == v) return u;
        for (int bit = LOG - 1; bit >= 0; bit--) {
            if (ancestor[u][bit] != ancestor[v][bit]) {
                u = ancestor[u][bit];
                v = ancestor[v][bit];
            }
```

```
        }
        return ancestor[u][0];
    }

    int dis(int u, int v) {
        int lca = get_LCA(u, v);
        return lvl[u] - lvl[lca] + lvl[v] - lvl[lca];
    }

    bool isAncestor(int u, int v) {
        return in[u] <= in[v] && out[v] <= out[u];
    }

} lcaTool;

struct SegmentTree {

    void build(int node, int l, int r) {
        if (l == r) {
            seg[node] = flat[l];
            return;
        }
        int mid = (l + r) / 2;
        build(node * 2, l, mid);
        build(node * 2 + 1, mid + 1, r);
        seg[node] = (seg[node * 2] | seg[node * 2 + 1]);
    }

    void update(int node, int l, int r, int idx, int val) {
        if (l == r) {
            seg[node] = val;
            return;
        }
        int mid = (l + r) / 2;
        if (idx <= mid) update(node * 2, l, mid, idx, val);
        else update(node * 2 + 1, mid + 1, r, idx, val);
        seg[node] = (seg[node * 2] | seg[node * 2 + 1]);
    }

    ll query(int node, int l, int r, int ql, int qr) {
        if (ql > r or qr < l or r < l) return 0;
        if (ql <= l and qr >= r) return seg[node];
        int mid = (l + r) / 2;
        ll query1 = query(node * 2, l, mid, ql, qr);
        ll query2 = query(node * 2 + 1, mid + 1, r, ql, qr);
        return (query1 | query2);
    }

    void rangeUpdate(int node, int l, int r, int ql, int qr, ll val) {
        // lazy propagation
        if (lazy[node] != 0) {
            seg[node] = lazy[node];
            if (l != r) {
                lazy[node * 2] = lazy[node];
                lazy[node * 2 + 1] = lazy[node];
```

```
            }
            lazy[node] = 0;
        }

        if (r < l || l > qr || r < ql) return;

        if (l >= ql and r <= qr) {
            seg[node] = val;
            if (l != r) {
                lazy[node * 2] = val;
                lazy[node * 2 + 1] = val;
            }
            return;
        }

        int mid = (l + r) / 2;
        rangeUpdate(node * 2, l, mid, ql, qr, val);
        rangeUpdate(node * 2 + 1, mid + 1, r, ql, qr, val);
        seg[node] = (seg[node * 2] | seg[node * 2 + 1]);
    }


    ll queryLazy(int node, int l, int r, int ql, int qr) {
        // lazy propagation
        if (lazy[node] != 0) {
            seg[node] = lazy[node];
            if (l != r) {
                lazy[node * 2] = lazy[node];
                lazy[node * 2 + 1] = lazy[node];
            }
            lazy[node] = 0;
        }

        if (r < l || l > qr || r < ql) return 0;

        if (l >= ql and r <= qr) return seg[node];

        int mid = (l + r) / 2;
        ll left = queryLazy(node * 2, l, mid, ql, qr);
        ll right = queryLazy(node * 2 + 1, mid + 1, r, ql, qr);
        return (left | right);
    }

    void updateSubtree(int node, ll val) {
        segtreeTool.rangeUpdate(1, 1, n, in[node], out[node], val);
    }

    ll querySubtree(int node) {
        return segtreeTool.queryLazy(1, 1, n, in[node], out[node]);
    }

} segtreeTool;
```

Max Matching(plus).cpp

```
int n, k;
vector<vector<int>> g;
vector<int> mt;
vector<bool> used;

bool try_kuhn(int v) {
    if (used[v])
        return false;
    used[v] = true;
    for (int to : g[v]) {
        if (mt[to] == -1 || try_kuhn(mt[to])) {
            mt[to] = v;
            return true;
        }
    }
    return false;
}

int main() {
    //... reading the graph ...

    mt.assign(k, -1);
    for (int v = 0; v < n; ++v) {
        used.assign(n, false);
        try_kuhn(v);
    }

    for (int i = 0; i < k; ++i)
        if (mt[i] != -1)
            printf("%d %d\n", mt[i] + 1, i + 1);
}

////////////////////////////////////////////////////////////////////////////


struct HopcroftKarp { // one based
    static const int inf = 1e16;
    int n , m;
    vector<int> l, r, d;
    vector<vector<int>> g;

    HopcroftKarp(int _n, int _m) {
        n = _n;
        m = _m;
        int p = _n + _m + 1;
        g.resize(p);
        l.resize(p, 0);
        r.resize(p, 0);
        d.resize(p, 0);
    }


    void add_edge(int u, int v) {
```

```
        g[u].push_back(v + n); // right id is increased by n
    }

    bool bfs() {
        queue<int> q;
        for (int u = 1; u <= n; u++) {
            if (!l[u]) d[u] = 0, q.push(u);
            else d[u] = inf;
        }
        d[0] = inf;
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (auto v : g[u]) {
                if (d[r[v]] == inf) {
                    d[r[v]] = d[u] + 1;
                    q.push(r[v]);
                }
            }
        }
        return d[0] != inf;
    }

    bool dfs(int u) {
        if (!u) return true;
        for (auto v : g[u]) {
            if (d[r[v]] == d[u] + 1 && dfs(r[v])) {
                l[u] = v;
                r[v] = u;
                return true;
            }
        }
        d[u] = inf;
        return false;
    }

    int maximum_matching() {
        int ans = 0;
        while (bfs()) {
            for (int u = 1; u <= n; u++) {
                if (!l[u] && dfs(u)) ans++;
            }
        }
        return ans;
    }

    vector<pair<int, int>> build_matching() {
        vector<pair<int, int>> res;
        for (int u = 1; u <= n; ++u) {
            if (l[u]) {
                res.emplace_back(u, l[u] - n); // map back right node ID
            }
        }
        return res;
    }
```

```
    pair<vector<int>, vector<int>> get_min_vertex_cover() {
        maximum_matching();
        vector<bool> visL(n + 1, false), visR(m + 1, false);

        function<void(int)> dfs_cover = [&](int u) {
            visL[u] = true;
            for (int v : g[u]) {
                int vr = v - n; // Convert back to original column index
                if (!visR[vr] && l[u] != v) { // non-matching edge
                    visR[vr] = true;
                    if (r[v]) dfs_cover(r[v]);
                }
            }
        };

        for (int u = 1; u <= n; u++) {
            if (l[u] == 0) dfs_cover(u);
        }

        vector<int> rows, cols;
        for (int u = 1; u <= n; u++) {
            if (!visL[u]) rows.push_back(u);
        }
        for (int v = 1; v <= m; v++) {
            if (visR[v]) cols.push_back(v);
        }

        return {rows, cols};
    }
};
```

Max matching(small).cpp

```
// Bipartite Maximum Matching using DFS-based Augmenting Path Algorithm
// Time Complexity: O(N * M), where N is number of nodes on left side, M is
total edges
// Use Case: Matchings in bipartite graphs – job assignment, pairing problems,
domino tiling, etc.

const int MX = 405;             // Max number of nodes on one side
(adjustable)
vector<int> adj[MX];           // Adjacency list for left-side nodes (0-
indexed)
int r[MX], l[MX];              // r[u] = right node matched to left node u,
l[v] = left node matched to right node v
int vis[MX], vis_id;           // vis[] for visited check per DFS call, vis_id
is incremented each DFS loop
int numR;                      // Number of left-side nodes (indexed 0 to
numR-1)

// Tries to find an augmenting path starting from node `u` on the left side
bool match(int u) {
    if (vis[u] == vis_id) return false;  // Already visited in this DFS
    vis[u] = vis_id;

    for (int nxt : adj[u]) {
        // If `nxt` is unmatched or we can re-match its pair
        if (l[nxt] == -1 || match(l[nxt])) {
            l[nxt] = u;      // Match right node to left
            r[u] = nxt;      // Match left node to right
            return true;
        }
    }
    return false;  // No augmenting path found
}

// Main driver to find max matching
int runMatching() {
    int cc = 0;  // Count of matched pairs
    memset(r, -1, sizeof r);  // Unmatched initially
    memset(l, -1, sizeof l);

    for (int i = 0; i < numR; i++) {
        vis_id++;            // Fresh DFS round
        if (match(i)) cc++;  // Augmenting path found → increase match count
    }
    return cc;               // Total number of matched pairs
}
```

SCC.cpp

```
// Kosaraju's Algorithm for Strongly Connected Components (SCC)
// Time Complexity: O(N + M)
// Use Case: Works on directed graphs to find SCCs — used in 2-SAT,
condensation DAGs, cycles detection, etc.

struct SCC {
    int n;
    vector<vector<int>> adj[2];  // adj[0]: original graph, adj[1]: reversed
graph
    vector<bool> vis;
    vector<vector<int>> com;     // List of SCCs
    stack<int> stk;

    SCC(int _n) {
        n = _n;
        adj[0].resize(n + 1);
        adj[1].resize(n + 1);
    }

    void addEdge(int u, int v) {
        // Add edge u → v to original graph
        // Also add v → u to reversed graph for second DFS
        adj[0][u].push_back(v);
        adj[1][v].push_back(u);
    }

    void dfs1(int u) {
        // First DFS: used to determine finishing times
        vis[u] = true;
        for (int v : adj[0][u]) {
            if (!vis[v]) dfs1(v);
        }
        stk.push(u); // Push finished node onto stack
    }

    void dfs2(int u, vector<int>& group) {
        // Second DFS: collect all reachable nodes from u in the reversed
graph
        vis[u] = true;
        group.push_back(u);
        for (int v : adj[1][u]) {
            if (!vis[v]) dfs2(v, group);
        }
    }

    void build() {
        // Step 1: Run DFS on original graph to fill stack with finish times
        vis.assign(n + 1, false);
        for (int i = 1; i <= n; i++) {
            if (!vis[i]) dfs1(i);
        }
```

```
        // Step 2: Run DFS on reversed graph in decreasing order of finish
times
        vis.assign(n + 1, false);
        while (!stk.empty()) {
            int u = stk.top(); stk.pop();
            if (!vis[u]) {
                com.emplace_back();
                dfs2(u, com.back());  // Store one SCC
            }
        }
    }
};
```

SPFA.cpp

```
// SPFA (Shortest Path Faster Algorithm)
// Solves single-source shortest paths on graphs with negative edge weights
(but no negative cycles)
// Detects negative cycles and supports path reconstruction

const int INF = 1e18;
int n; // number of nodes
vector<vector<pair<int, int>>> adj; // adj[u] = {v, weight}
vector<int> dist, par, cnt;
vector<bool> inq;

bool spfa(int src) {
    dist.assign(n, INF);
    par.assign(n, -1);
    cnt.assign(n, 0);
    inq.assign(n, false);

    queue<int> q;
    dist[src] = 0;
    q.push(src);
    inq[src] = true;

    while (!q.empty()) {
        int u = q.front(); q.pop();
        inq[u] = false;

        for (auto [v, w] : adj[u]) {
            if (dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                par[v] = u;
                if (!inq[v]) {
                    q.push(v);
                    inq[v] = true;
                    cnt[v]++;
                    if (cnt[v] > n) return false; // negative cycle detected
                }
            }
        }
```

```
    }
    return true; // no negative cycles
}

// Use par[] to reconstruct shortest path from src to any node:
// vector<int> path;
// for (int v = dest; v != -1; v = par[v]) path.push_back(v);
// reverse(path.begin(), path.end());
```

Articilation points.cpp

```
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    int children=0;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[to]) {
            low[v] = min(low[v], tin[to]);
        } else {
            dfs(to, v);
            low[v] = min(low[v], low[to]);
            if (low[to] >= tin[v] && p!=-1)
                IS_CUTPOINT(v);
            ++children;
        }
    }
    if(p == -1 && children > 1)
        IS_CUTPOINT(v);
}

void find_cutpoints() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs (i);
    }
}
```

◇ Mathmatics

2D Kadane.cpp

```
#include <bits/stdc++.h>
using namespace std;

const int ROW = 4;
const int COL = 5;

// Standard Kadane's Algorithm for 1D array
int kadane(int* arr, int* start, int* finish, int n) {
    int sum = 0, maxSum = INT_MIN;
    int local_start = 0;
    *finish = -1;

    for (int i = 0; i < n; ++i) {
        sum += arr[i];

        if (sum < 0) {
            sum = 0;
            local_start = i + 1;
        } else if (sum > maxSum) {
            maxSum = sum;
            *start = local_start;
            *finish = i;
        }
    }

    // All elements are negative — fallback to max element
    if (*finish == -1) {
        maxSum = arr[0];
        *start = *finish = 0;
        for (int i = 1; i < n; ++i) {
            if (arr[i] > maxSum) {
                maxSum = arr[i];
                *start = *finish = i;
            }
        }
    }

    return maxSum;
}

// Function to find maximum sum rectangle in 2D matrix
void findMaxSum(int M[][COL]) {
    int maxSum = INT_MIN;
    int finalLeft, finalRight, finalTop, finalBottom;

    int temp[ROW], start, finish;

    for (int left = 0; left < COL; ++left) {
        memset(temp, 0, sizeof(temp));
```

```
        for (int right = left; right < COL; ++right) {
            for (int i = 0; i < ROW; ++i)
                temp[i] += M[i][right];

            int sum = kadane(temp, &start, &finish, ROW);

            if (sum > maxSum) {
                maxSum = sum;
                finalLeft = left;
                finalRight = right;
                finalTop = start;
                finalBottom = finish;
            }
        }
    }

    cout << "Top-Left:     (" << finalTop << ", " << finalLeft << ")\n";
    cout << "Bottom-Right: (" << finalBottom << ", " << finalRight << ")\n";
    cout << "Max Sum:      " << maxSum << "\n";
}

// Example usage
int main() {
    int M[ROW][COL] = {
        {1,  2,  -1, -4, -20},
        {-8, -3, 4,  2,   1},
        {3,  8, 10, 1,   3},
        {-4, -1, 1,  7,  -6}
    };

    findMaxSum(M);
    return 0;
}
```

Catalan numbers.cpp

```
/// Catalan Numbers
/// The nth Catalan number Cn is the solution for:
/// 1. Balanced Parentheses: The number of correct bracket sequences with `n`
opening and `n` closing brackets.
/// 2. Word Problems: Counting valid sequences of balanced words in formal
language theory.
/// 3. Bracketing Problems: The number of ways to fully parenthesize an
expression with `n` pairs of operands/operators.
/// 4. Binary Search Trees: The number of BSTs that can be formed with `n`
distinct nodes.
/// 5. Sorted Trees: Ways to insert `n` elements into an empty BST to maintain
order.
/// 6. Full Binary Trees: Number of full binary trees with `n+1` leaves.
/// 7. Triangulations of a Polygon: Ways to triangulate a convex polygon with
`n+2` sides.
/// 8. Non-Crossing Handshakes: Ways `n` pairs can shake hands without
crossing.
```

```
/// 9. Dyck Paths: Grid paths that never go above the diagonal.
/// 10. Stack Permutations: Valid permutations of {1...n} using a stack.
/// 11. Valid Sequences: Valid stack operation sequences for given
permutation.
/// 12. Mountain Ranges: Ways to draw mountain ranges with `n` upstrokes and
`n` downstrokes never going below start.
/// 13. Schroeder's Second Problem: Valid insertion of `n` pairs of
parentheses maintaining balance.
/// 14. Planar Graphs: Non-crossing partitions of planar points.
/// 15. Matrix Chain Multiplication: Ways to fully parenthesize multiplication
of `n+1` matrices.
/// 16. Catalan Paths: Grid paths that stay on or below the line y = x.

const int MAXN = 1e5 + 5;
const int MOD = 1e9 + 7;

/// Function to compute modular inverse using Fermat's little theorem (MOD
must be prime)
ll modinv(ll a, ll mod) {
    ll res = 1;
    ll p = mod - 2;
    while (p > 0) {
        if (p & 1) res = res * a % mod;
        a = a * a % mod;
        p >>= 1;
    }
    return res;
}

/// Precompute factorials and inverse factorials for efficient Catalan number
calculation
ll fact[MAXN], invfact[MAXN];

void precompute_factorials(int n) {
    fact[0] = invfact[0] = 1;
    for (int i = 1; i <= n; i++)
        fact[i] = fact[i - 1] * i % MOD;
    invfact[n] = modinv(fact[n], MOD);
    for (int i = n - 1; i >= 1; i--)
        invfact[i] = invfact[i + 1] * (i + 1) % MOD;
}

/// Returns the nth Catalan number modulo MOD
ll catalan(int n) {
    if (n == 0) return 1;
    // Cn = (2n)! / (n! * (n + 1)!)
    return fact[2 * n] * invfact[n] % MOD * invfact[n + 1] % MOD;
}
```

Combinatorics.cpp

```
using ll = long long;

ll MOD = 1e9+7;
std::vector<ll> fac, inv, finv;

// Precompute factorials and their inverses up to n
void init_combinatorics(int n, ll mod) {
    MOD = mod;
    fac.resize(n + 1);
    inv.resize(n + 1);
    finv.resize(n + 1);

    fac[0] = finv[0] = 1;
    inv[1] = finv[1] = 1;

    for (int i = 1; i <= n; ++i) fac[i] = fac[i - 1] * i % MOD;
    for (int i = 2; i <= n; ++i) inv[i] = MOD - MOD / i * inv[MOD % i] % MOD;
    for (int i = 2; i <= n; ++i) finv[i] = finv[i - 1] * inv[i] % MOD;
}


// Modular arithmetic utilities
ll mod_add(ll a, ll b) {return (a + b) % MOD;}
ll mod_sub(ll a, ll b) {return (a - b + MOD) % MOD;}
ll mod_mul(ll a, ll b) {return a * b % MOD;}

// Fast modular exponentiation
ll power(ll base, ll exp) {
    ll result = 1;
    base %= MOD;
    while (exp)
    {
        if (exp & 1) result = mod_mul(result,base);
        base = mod_mul(base,base);
        exp >>= 1;
    }
    return result;
}

// Modular inverse using Fermat's Little Theorem (MOD must be prime)
ll mod_inv(ll x) {return power(x, MOD - 2);}


// Compute C(n, r) = n choose r modulo MOD
ll nCr(ll n, ll r) {
    if (n < 0 || r < 0 || r > n) return 0;
    return fac[n] * finv[r] % MOD * finv[n - r] % MOD;
}
// nCr with no mod
long long c(int n, int k) {
     long long result = 1;
    for (int i = 0; i < k; ++i) {
        result *= (n - i);
```

```
        result /= (i + 1);
    }
    return result;
}

ll p(int n, int k) {
    ll result = 1;
    for (int i = 0; i < k; ++i) {
        result *= (n - i);
    }
    return result;
}

// Compute P(n, r) = n permute r modulo MOD
ll nPr(ll n, ll r) {
    if (n < 0 || r < 0 || r > n) return 0;
    return fac[n] * finv[n - r] % MOD;
}

// Catalan number: Cn = C(2n, n) / (n + 1)
ll catalan(int n) {
    return nCr(2 * n, n) * mod_inv(n + 1) % MOD;
}

// Stars and Bars: number of ways to divide n identical items into k parts
ll stars_and_bars(ll n, ll k) {
    return nCr(n + k - 1, k - 1);
}

// a*a + b*b = n
bool isSumOfTwoSquares(int n) {
    for (int i = 2; i * i <= n; ++i) {
        int count = 0;
        while (n % i == 0) {
            ++count;
            n /= i;
        }
        if (i % 4 == 3 && count % 2 != 0)
            return false;
    }
    return !(n % 4 == 3);
}

// a*a + b*b + c*c = n
bool isSumOfThreeSquares(int n) {
    while (n % 4 == 0)
        n /= 4;
    return n % 8 != 7;
}
```

Divisability rules.txt

```
2 The last digit should be even.
3 The sum of the digits should be divisible by 3.
4 The last two digits should be divisible by 4.
5 The last digit should either be 0 or 5.
6 The number should be divisible by both 2 and 3.
7 The double of the last digit, when subtracted by the rest of the number, the
difference obtained should be divisible by 7.
8 The last three digits should be divisible by 8.
9 The sum of the digits should be divisible by 9.
10 The last digit should be 0.
11 The difference of the alternating sum of digits should be divisible by 12.
12 The number should be divisible by both 3 and 4.
13 The four times of the last digit, when added to the rest of the number, the
result obtained should be divisible by 13.
17 The five times of the last digit, when subtracted by the rest of the
number, the difference obtained should be divisible by 17.
19 The double of the last digit, when added to the rest of the number, the
result obtained should be divisible by 19.
```

FFT with mod.cpp

```
#include<bits/stdc++.h>
using namespace std;
#define ll long long
const int N = 1e5+5,mod=1009;
const long double PI = acos(-1);
typedef complex<double> C;
#define rep(i,a,b)  for(int i = a; i < b; i++)
void fft(vector<C>& a) {
    int n = a.size(), L = 31 - __builtin_clz(n);
    static vector<complex<long double>> R(2, 1);
    static vector<C> rt(2, 1);  // (^ 10% faster if double)
    for (static int k = 2; k < n; k *= 2) {
        R.resize(n); rt.resize(n);
        auto x = polar(1.0L, acos(-1.0L) / k);
        rep(i,k,2*k) rt[i] = R[i] = i&1 ? R[i/2] * x : R[i/2];
    }
    vector<ll> rev(n);
    rep(i,0,n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
    rep(i,0,n) if (i < rev[i]) swap(a[i], a[rev[i]]);
    for (int k = 1; k < n; k *= 2)
        for (int i = 0; i < n; i += 2 * k) rep(j,0,k) {
                // C z = rt[j+k] * a[i+j+k]; // (25% faster if hand-rolled)
/// include-line
                auto x = (double *)&rt[j+k], y = (double *)&a[i+j+k];
/// exclude-line
                C z(x[0]*y[0] - x[1]*y[1], x[0]*y[1] + x[1]*y[0]);
/// exclude-line
                a[i + j + k] = a[i + j] - z;
                a[i + j] += z;
            }
}
```

```
vector<ll> conv(const vector<ll> & a, const vector<ll> & b) {
    if (a.empty() || b.empty()) return {};
    vector<ll> res(a.size() + b.size() - 1);
    int L = 32 - __builtin_clz(res.size()), n = 1 << L;
    vector<C> in(n), out(n);
    copy(a.begin(),a.end(), begin(in));
    rep(i,0,b.size()) in[i].imag(b[i]);
    fft(in);
    for (C& x : in) x *= x;
    rep(i,0,n) out[i] = in[-i & (n - 1)] - conj(in[i]);
    fft(out);
    rep(i,0,res.size()) res[i] = (mod+(ll)round(imag(out[i]) / (4 * n))) %
mod;
    return res;
}
signed main()
{
    ios_base::sync_with_stdio(0);cin.tie(0);cout.tie(0);
    int n,m,k;cin>>n>>m>>k;
    int f[m+1]={};
    for(int i=0;i<n;i++)
    {
        int x;cin>>x;
        f[x]++;
    }
    priority_queue<pair<int,vector<ll>>>pq;
    for(int i=0;i<=m;i++)
    {
        if(f[i])
        {
            f[i]%=mod;
            pq.push({-f[i],vector<ll>(f[i]+1,1)});
        }
    }
    while(pq.size()>1)
    {
        auto [a,b]=pq.top();
        pq.pop();
        auto [x,y]=pq.top();
        pq.pop();
        vector<ll>mul=conv(b,y);
        pq.push({-mul.size(),mul});
    }
    cout<<pq.top().second[k];
    return 0;
}
```

FFT.cpp

```
#include <bits/stdc++.h>
using namespace std;

// Fast Fourier Transform for polynomial multiplication
#define int long long // 64-bit integers for handling large coefficients
using cd = complex<double>; // Complex numbers for FFT
const double PI = acos(-1); // Pi constant for angle calculations

// Performs the FFT or inverse FFT on array `a`
void fft(vector<cd> & a, bool invert) {
    int n = a.size();

    // Bit-reversal permutation to order elements before FFT
    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j & bit; bit >>= 1)
            j ^= bit;
        j ^= bit;
        if (i < j)
            swap(a[i], a[j]);
    }

    // Cooley-Tukey FFT algorithm
    for (int len = 2; len <= n; len <<= 1) {
        double ang = 2 * PI / len * (invert ? -1 : 1);
        cd wlen(cos(ang), sin(ang)); // root of unity
        for (int i = 0; i < n; i += len) {
            cd w(1);
            for (int j = 0; j < len / 2; j++) {
                cd u = a[i + j], v = a[i + j + len / 2] * w;
                a[i + j] = u + v;
                a[i + j + len / 2] = u - v;
                w *= wlen;
            }
        }
    }

    // Divide all values by `n` if doing inverse FFT
    if (invert) {
        for (cd & x : a)
            x /= n;
    }
}


// Multiplies two integer polynomials `a` and `b` using FFT
// Returns a new vector with the result of the multiplication
vector<int> multiply(vector<int> const& a, vector<int> const& b) {
    vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
    int n = 1;
    // Find power of 2 size large enough to hold the result
    while (n < a.size() + b.size())
        n <<= 1;
```

```
    fa.resize(n);
    fb.resize(n);

    // Forward FFT
    fft(fa, false);
    fft(fb, false);

    // Point-wise multiplication of transformed values
    for (int i = 0; i < n; i++)
        fa[i] *= fb[i];

    // Inverse FFT to get the result back in time domain
    fft(fa, true);

    // Round real part to get integer coefficients
    vector<int> result(n);
    for (int i = 0; i < n; i++)
        result[i] = round(fa[i].real());
    return result;
}

Fast_Fib.cpp
int Fib(int n) { // 0, 1, 1, 2, 3, 5 | call it with(n+1) to start with 1
    ll i = 1, h = 1, j = 0, k = 0, t;
    while (n > 0) {
        if (n % 2 == 1) {
            t = (j * h) % MOD;
            j = (i * h + j * k + t) % MOD;
            i = (i * k + t) % MOD;
        }
        t = (h * h) % MOD;
        h = (2 * k * h + t) % MOD;
        k = (k * k + t) % MOD;
        n = n / 2;
    }
    return j;
}
```

Floor Sum.cpp

```
// Computes the sum of floor((a * i + b) / c) for i in [0, n]
// That is: ∑_{i=0}^{n} ⌊(a*i + b)/c⌋
// Valid for all non-negative integers a, b, c, n
// Based on recursive transformation (known as [Hermite's reduction])
// Time Complexity: O(log(a + b + c + n))

long long FloorSumAP(long long a, long long b, long long c, long long n) {
    // Base case: when a = 0 ⇒ expression reduces to ⌊b / c⌋ for each i in [0,
n]
    if (a == 0)
        return (b / c) * (n + 1);
```

```
    // If a or b ≥ c, reduce the problem using integer division
    // Use linearity: floor((a*i + b)/c) = floor(a/c * i + b/c + (a%c * i +
b%c)/c)
    if (a >= c || b >= c)
        return ((n * (n + 1)) / 2) * (a / c)           // ∑ i * floor(a/c)
             + (n + 1) * (b / c)                       // ∑ floor(b/c)
             + FloorSumAP(a % c, b % c, c, n);         // recurse on the
remainder

    // General case: when a < c and b < c, apply transformation
    // We compute m = floor((a*n + b) / c)
    // Then use identity:
    // ∑ floor((a*i + b)/c) = n*m - ∑ floor((c*i + (c - b - 1))/a) for i in
[0, m-1]
    long long m = (a * n + b) / c;
    return m * n - FloorSumAP(c, c - b - 1, a, m - 1);
}
```

Inclusion Exclusion.cpp

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

const int MAX = 1e6 + 5;

int sp[MAX];        // Smallest Prime Factor for each number
int cnt[MAX];       // Count of numbers divisible by a certain product of
primes
int sz[MAX];        // Number of prime factors in each product (bitmask subset
size)

// Precomputes the smallest prime factor for all numbers up to MAX using sieve
void Sieve() {
    for (int i = 2; i < MAX; i += 2)
        sp[i] = 2; // even numbers have smallest prime factor 2
    for (int i = 3; i < MAX; i += 2) {
        if (!sp[i]) {
            sp[i] = i;
            for (int j = i * i; j < MAX; j += 2 * i) {
                if (!sp[j]) sp[j] = i;
            }
        }
    }
}

signed main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    Sieve();
    int n;
    cin >> n;
```

```
    vector<int> a(n);
    for (int &x : a) cin >> x;

    // Total number of unordered pairs is n * (n - 1) / 2
    int total_pairs = n * (n - 1) / 2;

    // Count how many numbers share any common prime factor
    for (int x : a) {
        vector<int> primes;
        // Factor x using smallest prime factor sieve
        while (x > 1) {
            int p = sp[x];
            primes.push_back(p);
            while (x % p == 0) x /= p;
        }

        int m = primes.size();
        int mask_limit = (1 << m);

        // For every non-empty subset of prime factors (inclusion-exclusion)
        for (int mask = 1; mask < mask_limit; ++mask) {
            int prod = 1, bits = 0;
            for (int i = 0; i < m; ++i) {
                if (mask & (1 << i)) {
                    prod *= primes[i];
                    ++bits;
                }
            }
            ++cnt[prod];
            sz[prod] = bits;
        }
    }

    // Apply inclusion-exclusion to subtract pairs with GCD > 1
    for (int i = 2; i < MAX; ++i) {
        if (cnt[i] == 0) continue;

        // cnt[i] choose 2 = number of unordered pairs sharing this common
product
        int add = cnt[i] * (cnt[i] - 1) / 2;

        if (sz[i] % 2 == 1)
            total_pairs -= add; // subtract if odd number of prime factors
        else
            total_pairs += add; // add if even number of prime factors
    }

    cout << total_pairs << '\n';
    return 0;
}
```

Josephus.cpp

```
/// Josephus Problem (Optimized Recursive Solution)
/// f(n, k): returns the position (1-based) of the survivor when every k-th
person is eliminated in a circle of n people
///
/// This specific version works efficiently for the special case when `k = 2`
/// The recurrence relation and pattern of elimination allow for an optimized
O(log n) solution

/// Use Cases:
/// - Game theory problems (last person standing)
/// - Problems involving circular elimination
/// - Classic recursion/dynamic programming problems
/// - System design for scheduling/elimination

/// Time Complexity: O(log n)
/// Space Complexity: O(log n) due to recursion depth

ll josephus(ll n, ll k) {
    if (n == 1) return 1;  // Base case: only one person remains

    // Case 1: If k is in the first half of people to be eliminated
    if (k <= (n + 1) / 2) {
        // If 2*k exceeds n, it wraps around the circle
        if (2 * k > n) return (2 * k) % n;
        else return 2 * k;
    }

    // Case 2: k is in the second half
    // We reduce the problem by half and recurse
    ll temp = josephus(n / 2, k - (n + 1) / 2);

    // Reconstruct position in original problem
    if (n % 2 == 1)
        return 2
```

Matrix expo(short).cpp

```
mat operator*(mat a, mat b)
{
    mat ret = mat(a.size(), row(b[0].size()));
    for (int i = 0; i < ret.size(); i++)
        for (int j = 0; j < ret[i].size(); j++)
            for (int k = 0; k < ret[i].size(); k++)
                ret[i][j] = (ret[i][j] + a[i][k] * b[k][j]) % MOD;
    return ret;
}
mat operator^(mat b, int p)
{
    if (p == 1)
        return b;
    if (p & 1)
        return b * (b ^ (p - 1));
    return (b * b) ^ (p / 2);
}
```

Matrix expo.cpp

```
// Matrix Exponentiation Structure
// Useful for solving linear recurrence relations in O(k^3 * log n),
// where k is the size of the transformation matrix.
// Applications: Fibonacci nth term, DP optimizations, Path counts, etc.

struct Matrix {
    int n, m, mod; // Dimensions of the matrix and modulus for operations
    vector<vector<int>> a; // 2D matrix data

    Matrix() {}

    // Constructor: Initialize n x m matrix with all values = val
    Matrix(int n, int m, int mod, int val) : n(n), m(m), mod(mod), a(n, vector<int>(m,
val)) {}

    // Matrix multiplication (modular)
    Matrix operator*(const Matrix& b) {
        Matrix res(n, b.m, mod, 0);
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < b.m; ++j)
                for (int k = 0; k < m; ++k)
                    res.a[i][j] = (res.a[i][j] + 1LL * a[i][k] * b.a[k][j]) % mod;
        return res;
    }

    // Matrix exponentiation (A^e), assumes square matrix (n == m)
    // Used when applying the same transformation matrix multiple times
    Matrix operator^(int e) {
        Matrix res(n, n, mod, 0), b = *this;
        for (int i = 0; i < n; ++i)
            res.a[i][i] = 1; // Initialize res as identity matrix

        // Exponentiation by squaring
        for (; e > 0; e /= 2, b = b * b)
            if (e % 2) res = res * b;

        return res;
    }

    // Matrix addition (modular)
    // Used less often in CP, but may be useful in some DP transitions
    Matrix operator+(const Matrix& b) {
        Matrix res(n, b.m, mod, 0);
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < b.m; ++j)
                for (int k = 0; k < m; ++k)
                    res.a[i][j] = (res.a[i][j] + 1LL * a[i][k] + b.a[k][j]) % mod;
        return res;
    }

    // Helper to turn matrix into diagonal matrix with 1s from position (i,j)
    // Mainly used to create identity or transition patterns
    void IdentifyD(int i, int j) {
        while (i < n && j < m) {
            a[i][j] = 1;
            i++, j++;
        }
    }
```

```
    // Utility function to print the matrix (debugging)
    void print() {
        for (auto& row : a) {
            for (auto& val : row) {
                cout << val << " ";
            }
            cout << "\n";
        }
    }
};
```

MillerRapin.cpp

```
using u128 = __uint128_t;
using ll = long long;

/// Optional: Safe modular multiplication without overflow (unused here due to
__uint128_t)
ll mult(ll s, ll m, ll mod) {
    if (!m) return 0;
    ll ret = mult(s, m / 2, mod);
    ret = (ret + ret) % mod;
    if (m & 1) ret = (ret + s) % mod;
    return ret;
}

/// Fast modular exponentiation (b^p mod mod)
ll fp(ll b, ll p, ll mod) {
    ll res = 1;
    b %= mod;
    while (p > 0) {
        if (p & 1)
            res = (u128)res * b % mod;
        b = (u128)b * b % mod;
        p >>= 1;
    }
    return res;
}

/// Miller-Rabin helper: checks if 'a' is a witness to 'n' being composite
bool check(ll n, ll a, ll d, ll s) {
    ll x = fp(a, d, n);  // Compute a^d % n
    if (x == 1 || x == n - 1) return false;
    for (int r = 1; r < s; r++) {
        x = (u128)x * x % n;
        if (x == n - 1) return false;
    }
    return true;
}

/// Miller-Rabin primality test (deterministic for n < 3.3e18)
bool MillerRabin(ll n) {
    if (n < 2) return false;

    // Write n-1 as 2^r * d with d odd
```

```
    ll d = n - 1;
    int r = 0;
    while ((d & 1) == 0) {
        d >>= 1;
        r++;
    }

    // Witnesses for deterministic version (up to 2^64 guaranteed)
    for (ll a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {
        if (a >= n) break;       // Only use bases < n
        if (check(n, a, d, r))   // If 'a' is a witness → n is composite
            return false;
    }
    return true;  // Probably prime
}
```

Mobious.cpp

```
#include <bits/stdc++.h>
using namespace std;

const int VALMAX = 1e7;

int mobius[VALMAX];  // Möbius function values

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int test_num;
    cin >> test_num;

    // --- Möbius Function Sieve ---
    // mobius[1] is initialized to -1 due to inversion logic used here (it's
usually 1 by definition)
    mobius[1] = -1;

    // Compute Möbius function using a sieve-like approach
    // This approach uses the inversion principle:
    // If mobius[i] ≠ 0, then flip its sign (alternating pattern of Möbius)
    // and update all multiples j of i by adding mobius[i] to mobius[j]
    for (int i = 1; i < VALMAX; i++) {
        if (mobius[i]) {
            mobius[i] = -mobius[i]; // Flip sign
            for (int j = 2 * i; j < VALMAX; j += i) {
                mobius[j] += mobius[i];
            }
        }
    }

    // --- Answer Queries ---
    // The key formula is:
    // Number of square-free integers ≤ n = ∑ (μ(i) × ⌊n / i²⌋), where i² ≤ n
    // This is based on Möbius inversion applied to the characteristic
```

```
function of square-free numbers
    while (test_num--) {
        long long n;
        cin >> n;

        long long ans = 0;
        for (int i = 1; 1LL * i * i <= n; i++) {
            ans += mobius[i] * (n / (1LL * i * i));
        }

        cout << ans << '\n';
    }

    return 0;
}
```

Nth Root.cpp

```
// ------------------------- Nth Root Finder -----------------------------

// Helper function to detect multiplication overflow
// Returns true if a * b would cause overflow
bool overflow(long long a, long long b) {
    long double res = static_cast<long double>(a) * b;
    if (a == 0 || b == 0 || res / b == a)
        return false;
    return true;
}

// Returns the integer `n`-th root of `x` if it exists (i.e., some integer `r`
such that r^n == x)
// Otherwise returns -1
long long nth_root(long long x, int n) {
    long long l = 1, r = x, theOne = -1;

    // Binary search for the nth root
    while (l <= r) {
        long long mid = (l + r) / 2;
        long long ans = 1;
        bool ovf = false;

        // Try to compute mid^n while checking for overflow
        for (int i = 0; i < n; i++) {
            ovf |= overflow(ans, mid);
            ans *= mid;
        }

        // If exact match, we found the root
        if (ans == x) {
            theOne = mid;
            break;
        }

        // If overflow or overshoot, search left
```

```
        if (ovf || ans > x)
            r = mid - 1;
        else // else search right
            l = mid + 1;
    }

    return theOne;
}

🟢 When to use:
- When you need to find an **exact** nth root of a number (e.g. cube root of
64).
- Common in number theory problems, binary search optimizations, and perfect
power detection.

🔴 Limitations:
- Only finds integer roots. If the root is irrational or not exact, returns -
1.
- Slower for very large x or high n due to repeated multiplication.

✅ Handles overflow safely using a helper function and long double casting.
```

PHI function.cpp

```
/// Computes Euler's Totient Function ϕ(n)
/// ϕ(n) = number of integers ≤ n that are coprime with n
/// Time Complexity: O(√n)
ll phi(ll n) {
    ll res = n;
    for (ll i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            while (n % i == 0)
                n /= i;
            res -= res / i;
        }
    }
    if (n > 1)
        res -= res / n;  // For any remaining prime factor > sqrt(n)
    return res;
}



const int N = 1e5 + 5;
int phi[N];

/// Precomputes ϕ(1...N) using a sieve
/// Time Complexity: O(N log log N)
void computeTotients() {
    for (int i = 0; i < N; i++) phi[i] = i;
    for (int i = 2; i < N; i++) {
        if (phi[i] == i) {  // i is prime
            for (int j = i; j < N; j += i)
                phi[j] -= phi[j] / i;
```

```
        }
    }
}

Quadratic Formula.cpp
// Solves the quadratic equation ax^2 + bx + c = 0
// Returns a pair of real roots (as doubles) if they exist
// If no real roots, returns an empty optional
#include <optional>
#include <cmath>
using namespace std;

optional<pair<double, double>> Quadratic_Formula(long long a, long long b,
long long c) {
    // Not a quadratic equation if a == 0
    if (a == 0) return nullopt;

    // Calculate the discriminant
    long long d = b * b - 4 * a * c;

    // No real roots if discriminant is negative
    if (d < 0) return nullopt;

    // Calculate roots using quadratic formula with real square root
    double sqrt_d = sqrt(d);
    double root1 = (-b + sqrt_d) / (2.0 * a);
    double root2 = (-b - sqrt_d) / (2.0 * a);

    return make_pair(root1, root2);
}


// USE CASE:
auto result = Quadratic_Formula(1, -3, 2); // Solves x^2 - 3x + 2 = 0
if (result) {
    auto [x1, x2] = result.value();
    cout << "Roots: " << x1 << " and " << x2 << '\n';
} else {
    cout << "No real roots.\n";
}
```

XOR Basis.cpp

```
struct XorBasis {
    static const int BITS = 60; // For 64-bit numbers, use 60 to be safe
    ll basis[BITS] = {};
    int size = 0; // Number of vectors in the basis

    // Insert number into basis
    void insert(ll mask) {
        for (int i = BITS - 1; i >= 0; i--) {
            if (!(mask >> i & 1)) continue;
            if (!basis[i]) {
                basis[i] = mask;
                size++;
                return;
            }
            mask ^= basis[i];
        }
    }
    // Check if mask can be represented as XOR of basis elements
    bool canRepresent(ll mask) {
        for (int i = BITS - 1; i >= 0; i--) {
            if ((mask >> i) & 1) {
                if (!basis[i]) return false;
                mask ^= basis[i];
            }
        }
        return true;
    }

    // Get maximum XOR value possible from subset of basis
    ll getMaxXor() {
        ll res = 0;
        for (int i = BITS - 1; i >= 0; i--) {
            if ((res ^ basis[i]) > res) {
                res ^= basis[i];
            }
        }
        return res;
    }

    // Count of distinct XORs from subsets: 2^size
    ll countDistinctXors() {
        return 1LL << size;
    }

    // Rebuild minimal basis (optional, ordered and unique)
    vector<ll> getMinimalBasis() {
        vector<ll> res;
        for (int i = BITS - 1; i >= 0; i--)
            if (basis[i]) res.push_back(basis[i]);
        return res;
    }
};
```

extended gcd.cpp

```
/// Extended Euclidean Algorithm
/// Solves: a * x + b * y = gcd(a, b)
/// Returns gcd(a, b) and computes x, y such that the equation holds
/// Use case: Modular inverse when gcd(a, m) == 1 → x = a⁻¹ mod m
ll extended_euclid(ll a, ll b, ll& x, ll& y) {
    if (b == 0) {
        x = 1, y = 0;
        return a;
    }
    ll x1, y1;
    ll g = extended_euclid(b, a % b, x1, y1);
    x = y1;
    y = x1 - (a / b) * y1;
    return g;
}
```

nCr_nPr.cpp

```
vector<int> F, invF;

void pre(int n) {
    F.resize(n + 1);
    invF.resize(n + 1);

    F[0] = 1;
    for (int i = 1; i <= n; i++)
        F[i] = mul(F[i - 1], i, Mod);

    invF[n] = inv(F[n]);
    invF[0] = 1;
    for (int i = n - 1; i; --i)
        invF[i] = mul(invF[i + 1], i + 1);
}

int nCr(int n, int r) {
    if (n < r) return 0;
    return mul(F[n], mul(invF[r], invF[n - r]));
}

int nPr(int n, int r) {
    if (n < r) return 0;
    return mul(F[n], invF[n - r]);
}
```

sums.cpp

```
/// Returns sum of integers in range [l, r]
/// sum = r*(r+1)/2 - (l-1)*l/2
ll sum(ll l, ll r) {
    assert(l <= r);
    return (r * (r + 1) / 2) - (l * (l - 1) / 2);
}


/// Returns sum of all odd numbers in [l, r]
/// Sum of first k odd numbers = k^2
ll sumOdd(ll l, ll r) {
    assert(l <= r);
    r++;
    ll x = r / 2;  // number of odd numbers ≤ r
    ll y = l / 2;  // number of odd numbers < l
    return x * x - y * y;
}
/// Returns sum of all even numbers in [l, r]
ll sumEven(ll l, ll r) {
    return sum(l, r) - sumOdd(l, r);
}
```

◇ Strings

String Tricks

```
void Z(){
    int l = 0 , r = 0;
    for (int i =1;i<n;i++){
        if(i < r)
            z[i] = min(r-i,z[i-l]);
        while(i + z[i] < n && s[i+z[i]] == s[z[i]])
            z[i]++;
        if (i + z[i] > r)
            l = i , r = i + z[i] ;
    }
}
////////////////////////////////////////////////////////////////////////////
void KMP(){
    for (int i = 1; i < n ; i++){
        int j = pi[i-1];
        while(j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j])
            j++;
        pi[i] = j;
    }
}
////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////

deque<int> manacher_odd(string &s){
    s = "@" + s + "!";
    int n = s.size();
    deque<int>pal(n);
    int l = 0 , r = 1;
    for (int i = 1 ; i < n ; i++ ){
        pal[i] = min(r-i , pal[l + (r - i)]);
        while(s[i - pal[i]] == s[i+pal[i]])
            pal[i]++;
        if (i + pal[i] > r)
            l = i - pal[i] , r = i + pal[i];
    }
    pal.pop_front();
    pal.pop_back();
    return pal;
}

deque<int> manacher_even_odd(string &s){
    string t;
    for(auto c:s)
        t += '#' , t+=c;
    t += '#';
    return manacher_odd(t);
}
```

```
signed main() {
/* ^^^ */    AhmedPlusPlus    /* ^^^ */

//    ->> practice makes perfect

    string s;cin>>s;
    deque<int>pal = manacher_even_odd(s);
    int mx = -1 , tar = -1;
    bool odd;
    for (int i = 1; i < pal.size() -1 ; i++){
        int nw = pal[i] - 1;
        if (nw > mx)
            mx = nw , tar = (i-1) / 2 , odd = i & 1;
    }
    string ans;
    for (int i = tar - mx/2 + (!odd) ; i <= tar + mx / 2 ; i++)
        ans += s[i];
    cout << ans << '\n';

}
////////////////////////////////////////////////////////////////////////////
/////////
Tricks

// Number of distinct substrings in a string
/*
To count distinct substrings of a string s, append characters one by one and
track new substrings.
When adding a character c to s, form t = s + c, then reverse t to turn
suffixes into prefixes.
Compute the Z-function of reversed t to find the longest prefix that repeats
elsewhere (z_max).
The number of new substrings ending in c is length(t) - z_max.
Summing this over all characters gives total distinct substrings.
The total time complexity is O(n²) for a string of length n.
*/
////////////////////////////////////////////////////////////////////////////
// freq of each one
/*
    string s;cin>>s;
    string t ;
    vector<int>fr(n+1) , ans(n+5);
    for(int i = n-1; i >= 0 ; --i){
        t = s[i] + t;
        vector<int>z = Z(t);
        z[0] = n - i;
        vector<int>tmp(n+2);
        for (auto j:z)tmp[0]++ , tmp[j+1]--;
        for (int j = 1 ; j <= n ;++j)
            tmp[j] += tmp[j-1] , fr[tmp[j]]++;
    }
    int prv = 0;
    for (int i = n  ; i >= 1 ; --i)
        fr[i] -= prv , prv += fr[i];
*/
```

Aho Corasick.cpp

```
/*
    Aho-Corasick Automaton
    Builds a trie with failure links for multiple pattern matching in O(n + m
+ z)
    n = sum of pattern lengths, m = text length, z = total matches found
    Used for: searching multiple patterns in text efficiently, spam filters,
virus scanners
    Known problems: multi-pattern string search, counting overlapping matches
*/

const int N = 1005;  // Maximum total nodes in trie (patterns * average
length)
const int M = 26;    // Alphabet size ('a' to 'z')

int trie[N][M];      // Trie transitions
int go[N][M];        // Go function with fallback (automaton)
int mrk[N];          // Mark endpoint of inserted patterns
int f[N];            // Failure function (suffix link)
int ptr = 1;         // Next unused trie node index

void BFS() {
    // Builds failure links and go transitions
    queue<int> q;
    for (int i = 0; i < M; i++) {
        if (trie[0][i]) {
            q.push(trie[0][i]);
            f[trie[0][i]] = 0;
        }
        go[0][i] = trie[0][i]; // go[0][i] initialized
    }

    while (!q.empty()) {
        int x = q.front(); q.pop();
        for (int i = 0; i < M; i++) {
            if (trie[x][i]) {
                int y = trie[x][i];
                f[y] = f[x];
                while (f[y] && !trie[f[y]][i])
                    f[y] = f[f[y]];
                if (trie[f[y]][i])
                    f[y] = trie[f[y]][i];

                mrk[y] += mrk[f[y]]; // Accumulate matches from fail links
                q.push(y);
                go[x][i] = y;
            } else {
                go[x][i] = go[f[x]][i];
            }
        }
    }
}
```

```
void ins(string x) {
    // Inserts pattern string x with value v
    int v; cin >> v;
    int cur = 0;
    for (char ch : x) {
        int c = ch - 'a';
        if (!trie[cur][c])
            trie[cur][c] = ptr++;
        cur = trie[cur][c];
    }
    mrk[cur] += v; // Mark this node as terminal with count
}
```

Hashing(double).cpp

```
const int N = 300300;
const int m1 = 1e9 + 7, b1 = 17;
const int m2 = 1e9 + 9, b2 = 31;

int pw1[N], inv1[N];
int pw2[N], inv2[N];

// Double Hashing
// Computes and compares hash values of substrings with two moduli to reduce
collision probability
// Time complexity: O(1) per hash query after O(n) preprocessing
// Used for: string matching, palindrome checks, substring uniqueness, hashing
2D grids
// Known problems: check equality of substrings, count distinct substrings,
detect repeated substrings

void pre(int n = N) {
    pw1[0] = inv1[0] = 1;
    pw2[0] = inv2[0] = 1;

    int iv1 = inv(b1, m1);
    int iv2 = inv(b2, m2);

    for (int i = 1; i < n; i++) {
        pw1[i] = mul(pw1[i - 1], b1, m1);
        inv1[i] = mul(inv1[i - 1], iv1, m1);
        pw2[i] = mul(pw2[i - 1], b2, m2);
        inv2[i] = mul(inv2[i - 1], iv2, m2);
    }
}

struct Hashing {
    int n;
    string s;
    vector<int> h1, h2;

    Hashing(const string& s_) {
        s = s_;
        n = s.size();
```

```
        h1.resize(n), h2.resize(n);
        h1[0] = h2[0] = s[0] - '0' + 1;

        for (int i = 1; i < n; i++) {
            h1[i] = add(h1[i - 1], mul(s[i] - '0' + 1, pw1[i], m1), m1);
            h2[i] = add(h2[i - 1], mul(s[i] - '0' + 1, pw2[i], m2), m2);
        }
    }

    pair<int, int> get(int l, int r) {
        assert(l >= 0 && r < n && l <= r);
        int x = h1[r], y = h2[r];
        if (l) {
            x = mul(sub(x, h1[l - 1], m1), inv1[l], m1);
            y = mul(sub(y, h2[l - 1], m2), inv2[l], m2);
        }
        return { x, y };
    }

    static pair<int, int> concat(pair<int, int> h1, pair<int, int> h2, int
h1_sz) {
        h1.first = add(h1.first, mul(h2.first, pw1[h1_sz]));
        h1.second = add(h1.second, mul(h2.second, pw2[h1_sz]));
        return h1;
    }
};
```

KMP.cpp

```
vector<int> prefix_function(const string& s) {
    // KMP Prefix Function
    // Computes the length of the longest proper prefix which is also a suffix
for each prefix of the string
    // Time complexity: O(n), where n = s.size()
    // Used for: pattern matching, counting occurrences of a pattern, string
periodicity
    // Known problems: find all occurrences of a pattern in text, border
detection, string compression

    int n = s.size();
    vector<int> pi(n);
    for (int i = 1; i < n; ++i) {
        int j = pi[i - 1];
        while (j > 0 && s[i] != s[j])
            j = pi[j - 1];
        if (s[i] == s[j])
            ++j;
        pi[i] = j;
    }
    return pi;
}
```

Trie.cpp

```
struct Trie {
    // Trie (Prefix Tree)
    // Supports insertion, deletion, and count of prefix/full-word occurrences
    // Time complexity per operation: O(len), where len = length of the string
    // Used for: prefix queries, string frequency counting, autocomplete,
dictionary matching
    // Known problems: implement dictionary with insert/count/erase, longest
prefix matching

    Trie* child[26];
    int prefCnt, endCnt;

    Trie() {
        memset(child, 0, sizeof child);
        prefCnt = endCnt = 0;
    }

    void insert(const string& s, int cnt = 1, int i = 0) {
        prefCnt += cnt;
        if (i == s.size()) {
            endCnt += cnt;
            return;
        }
        int cur = s[i] - 'a';
        if (!child[cur]) {
            child[cur] = new Trie();
        }
        child[cur]->insert(s, cnt, i + 1);
    }

    int erase(const string& s, int cnt = 1, int i = 0) {
        if (i == s.size()) {
            int minC = min(cnt, endCnt);
            endCnt -= minC;
            prefCnt -= minC;
            return minC;
        }
        int cur = s[i] - 'a';
        int minC = child[cur]->erase(s, cnt, i + 1);
        prefCnt -= minC;

        if (child[cur]->prefCnt == 0) {
            delete child[cur];
            child[cur] = nullptr;
        }
        return minC;
    }

    pair<int, int> count(const string& s, int i = 0) {
        if (i == s.size()) return { prefCnt, endCnt };
        int cur = s[i] - 'a';
        if (!child[cur]) return { 0, 0 };
        return child[cur]->count(s, i + 1);
    }
};
```

XOR Hashing.cpp

```
#include <chrono>
#include <random>

#include <bits/stdc++.h>
using namespace std;

/*
    XOR Hash - randomized hashing using XOR and a random base
    Use case: fast and collision-resistant hash for strings or containers
    rng(): returns a random 64-bit integer for use as a base or hash seed
    Commonly used in: unordered_map custom hash, randomized hash functions for
strings
*/

long long rng() {
    static mt19937_64
gen(chrono::steady_clock::now().time_since_epoch().count());
    return uniform_int_distribution<long long>(0, LLONG_MAX)(gen);
}
```

Z function.cpp

```
vector<int> Z_function(const string& s) {
    // Z-Function
    // Computes an array where Z[i] is the length of the longest substring
starting from s[i] that matches the prefix of s
    // Time complexity: O(n), where n = s.size()
    // Used for: pattern matching, number of distinct substrings, string
compression, border detection
    // Known problems: find pattern occurrences in text using Z(p + '#' + t),
count unique substrings

    int n = s.size();
    vector<int> Z(n);
    for (int i = 1, l = 0, r = 0; i < n; i++) {
        if (i <= r)
            Z[i] = min(Z[i - l], r - i + 1);
        while (i + Z[i] < n && s[Z[i]] == s[i + Z[i]])
            Z[i]++;
        if (i + Z[i] - 1 > r)
            l = i, r = i + Z[i] - 1;
    }
    return Z;
}
```

manacher.cpp

```
pair<vector<int>, vector<int>> manacher(const string& s) {
    // Manacher's Algorithm
    // Finds the length of the longest palindromic substring centered at each
position
    // d1[i]: radius of the longest odd-length palindrome centered at i
    // d2[i]: radius of the longest even-length palindrome centered at i
    // Time complexity: O(n), where n = s.size()
    // Used for: finding all palindromic substrings, longest palindromic
substring
    // Known problems: count palindromes, find longest palindromic substring
in linear time

    int n = s.size();
    vector<int> d1(n); // odd-length
    for (int i = 0, l = 0, r = -1; i < n; i++) {
        int k = (i > r) ? 1 : min(d1[r - i + l], r - i + 1);
        while (i - k >= 0 && i + k < n && s[i - k] == s[i + k]) k++;
        d1[i] = k--;
        if (i + k > r) l = i - k, r = i + k;
    }

    vector<int> d2(n); // even-length
    for (int i = 0, l = 0, r = -1; i < n; i++) {
        int k = (i > r) ? 0 : min(d2[r - i + l + 1], r - i + 1);
        while (i - k - 1 >= 0 && i + k < n && s[i - k - 1] == s[i + k]) k++;
        d2[i] = k--;
        if (i + k > r) l = i - k - 1, r = i + k;
    }

    return {d1, d2};
}
```

random Hashing.cpp

```
#include <bits/stdc++.h>
using namespace std;
/*
    Randomized Rabin-Karp
    Uses random base and random prime mod for rolling hash of strings
    Reduces collision probability significantly compared to fixed-base hashing
    Time: O(1) hash query after O(n) preprocessing
    Used for: comparing substrings, detecting duplicates, pattern matching
*/

long long rng() {
    // Generates a 64-bit random number using Mersenne Twister
    static mt19937_64
gen(chrono::steady_clock::now().time_since_epoch().count());
    return uniform_int_distribution<long long>(0, LLONG_MAX)(gen);
}

int binpower(int base, int pw, int mod) {
    // Fast exponentiation modulo mod (O(log pw))
    int ret = 1;
    while (pw) {
        if (pw & 1)
            ret = (1LL * ret * base) % mod;
        base = (1LL * base * base) % mod;
        pw /= 2;
    }
    return ret;
}

bool probablyPrimeFermat(int n, int iter = 5) {
    // Fermat primality test: probabilistic, fast for large numbers
    if (n < 4)
        return n == 2 || n == 3;

    for (int i = 0; i < iter; i++) {
        int a = 2 + rand() % (n - 3); // random base for test
        if (binpower(a, n - 1, n) != 1)
            return false; // definitely composite
    }
    return true; // probably prime
}

vector<long long> generate_primes(long long l, long long r) {
    // Collects probably-prime numbers in range [l, r]
    vector<long long> ret;
    for (int i = l; i <= r; i++) {
        if (probablyPrimeFermat(i))
            ret.push_back(i);
    }
    return ret;
}
```

```
int gen_mod() {
    // Picks a random prime in range [1e9, 1e9 + 5000]
    int l = 1000000000 + rng() % 1000000000;
    vector<long long> primes = generate_primes(l, l + 5000);
    return primes[rng() % primes.size()];
}

int base, mod;
vector<int> pw, h;

void init_hash(const string& s) {
    // Initializes prefix hashes and powers of base
    // base: random in [256, 555], mod: large prime from gen_mod
    int n = s.size();
    base = 256 + rng() % 300;
    mod = gen_mod();

    pw.resize(n + 1);
    h.resize(n + 1);
    pw[0] = 1;
    for (int i = 1; i <= n; i++) {
        pw[i] = 1LL * pw[i - 1] * base % mod;
        h[i] = (1LL * h[i - 1] * base + s[i - 1]) % mod;
    }
}

int get_hash(int l, int r) {
    // Computes hash of substring s[l..r] in O(1)
    // Valid only after calling init_hash
    int res = (h[r + 1] - 1LL * h[l] * pw[r - l + 1]) % mod;
    if (res < 0) res += mod;
    return res;
}
```

◇ DP

Dp techniques.cpp

```
// 1. Memoization (Top-Down)
int dp_memo[N];

int fib(int n) {
    if (n <= 1) return n;
    if (dp_memo[n] != -1) return dp_memo[n];
    return dp_memo[n] = (fib(n - 1) + fib(n - 2)) % MOD;
}

// 2. Tabulation (Bottom-Up)
int dp_tab[N];

void solve_fib(int n) {
    dp_tab[0] = 0;
    dp_tab[1] = 1;
    for (int i = 2; i <= n; i++) {
        dp_tab[i] = (dp_tab[i - 1] + dp_tab[i - 2]) % MOD;
    }
}

// 3. Memory Optimization (Rolling Array)
// Use when dp[i] only depends on dp[i-1], dp[i-2], etc.
int fib_rolling(int n) {
    int a = 0, b = 1;
    for (int i = 2; i <= n; i++) {
        int c = (a + b) % MOD;
        a = b;
        b = c;
    }
    return b;
}

// 4. Knapsack 1D Optimization
// from 2D dp[i][w] → dp[w]
// Use when only previous row is needed
int weight[N], value[N];

int knapsack(int n, int W) {
    vector<int> dp(W + 1, 0);
    for (int i = 1; i <= n; i++) {
        for (int w = W; w >= weight[i]; w--) {
            dp[w] = max(dp[w], value[i] + dp[w - weight[i]]);
        }
    }
    return dp[W];
}


// ============================
```

```
// 5. Bitmask DP
// ============================
int tsp(int mask, int pos, vector<vector<int>>& dist, int n,
vector<vector<int>>& dp) {
    if (mask == (1 << n) - 1) return dist[pos][0];
    if (dp[mask][pos] != -1) return dp[mask][pos];

    int ans = INF;
    for (int city = 0; city < n; city++) {
        if (!(mask & (1 << city))) {
            ans = min(ans, dist[pos][city] + tsp(mask | (1 << city), city,
dist, n, dp));
        }
    }
    return dp[mask][pos] = ans;
}


// ============================
// 6. Tree DP (Post-order DFS)
// ============================
vector<int> tree[N];
int dp_tree[N];

int dfs(int u, int p) {
    dp_tree[u] = 1;
    for (int v : tree[u]) {
        if (v == p) continue;
        dp_tree[u] += dfs(v, u);
    }
    return dp_tree[u];
}


// ============================
// 7. State Compression via Parity
// Use dp[i % 2][...] instead of dp[i][...]
// (only two layers kept in memory)
// ============================
int compress_dp_parity(int n, int m) {
    vector<vector<int>> dp(2, vector<int>(m + 1));
    for (int i = 0; i <= m; i++) dp[0][i] = 1; // base case
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= m; j++) {
            dp[i % 2][j] = j == 0 ? 1 : (dp[i % 2][j - 1] + dp[(i - 1) %
2][j]) % MOD;
        }
    }
    return dp[n % 2][m];
}
```

Double knapsack with memory reduction

```
const int N = 5e2 + 1, INF = 1e9, mod = 1e9 + 7, LOG = 18;
int dp[2][N][N];
int Weight[N], value[N];
int pieces, capacity1, capacity2, n, m, ans;
int depe(int idx, int w1, int w2) {
    if (idx == pieces) return 0;
    int& ret = dp[idx][w1][w2];
    if (ret != -1) return ret;

    // leave
    ret = depe(idx + 1, w1, w2);

    //stock 1
    int we1 = Weight[idx] + w1 + n;
    if (we1 <= capacity1) {
        ret = max(ret, depe(idx + 1, we1, w2) + value[idx]);
    }

    //stock 2
    int we2 = Weight[idx] + w2 + m;
    if (we2 <= capacity2) {
        ret = max(ret, depe(idx + 1, w1, we2) + value[idx]);
    }

    return ret;
}

int iterative_dp() {
    for (int i = pieces - 1; i >= 0; i--) {
        int idx = i % 2;
        for (int w1 = 0; w1 <= capacity1; w1++) {
            for (int w2 = 0; w2 <= capacity2; w2++) {
                dp[idx][w1][w2] = dp[!idx][w1][w2];
                // stock 1
                int we1 = Weight[i] + w1 + n;
                if (we1 <= capacity1) {
                    dp[idx][w1][w2] = max(dp[idx][w1][w2], dp[!idx][we1][w2] +
value[i]);
                }

                // stock 2
                int we2 = Weight[i] + w2 + m;
                if (we2 <= capacity2) {
                    dp[idx][w1][w2] = max(dp[idx][w1][w2], dp[!idx][w1][we2] +
value[i]);
                }
                ans = max(ans, dp[idx][w1][w2]);
            }
        }
    }
    return ans;
}
```