**Batch : T7**

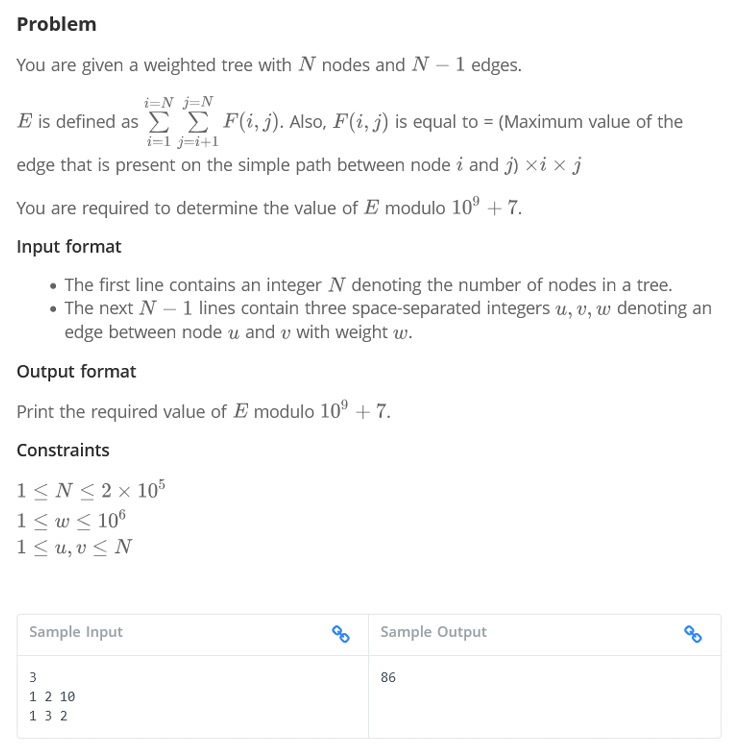
**Practical No . 9**

**Title of Assignment: Graphs, DP**

**Student Name: Om Kulkarni**

**Student PRN: 22510034**

**Problem Statement:**

****

1. Algorithm/Pseudocode

**Input Parsing**:

* Read the integer N, which represents the number of nodes.
* Read N−1 edges of the form (u, v, w), where u and v are connected by an edge with weight w.

**Tree Construction**:

* Construct an adjacency list from the input edges to represent the tree.

**DFS Preprocessing**:

* Perform a **Depth First Search (DFS)** from any arbitrary root node (commonly node 1) to preprocess the tree for Binary Lifting.
* During the DFS, record:
  + The **depth** of each node.
  + The **parent** of each node.
  + The **maximum edge weight** on the path to each ancestor node.

**Binary Lifting Table Setup**:

* Initialize a binary lifting table, where parent[i][j] represents the 2^j-th ancestor of node i.
* For each node i and each power j, update the table by setting parent[i][j] and the maximum edge weight between node i and its 2^j-th ancestor.

**Querying the Maximum Edge Weight using LCA**:

* For each pair of nodes (i,j) where i<j, compute the **Lowest Common Ancestor (LCA)** using binary lifting.
* Once the LCA of iii and j is determined, compute the maximum edge weight on the path between i and j by querying the binary lifting table.

**Compute the Function F(i,j)**

* For each pair (i,j) calculate F(i,j)=max\_weight∗i∗j where max\_weight is the maximum edge weight between i and j.
* Accumulate this value into E.

**Modulo Operation**:

* Since E can be very large, compute E mod (10^9 + 7) at each step to avoid overflow.

**Output**:

* Output the final value of E mod (10^9 + 7)

Program Code

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

const int MAXN = 200005;

const int LOG = 20;

const int MOD = 1000000007;

struct Edge {

    int to, weight;

};

vector<Edge> adj[MAXN];

int depth[MAXN], parent[MAXN][LOG], maxEdge[MAXN][LOG];

int N;

void dfs(int *node*, int *par*, int *w*) {

    parent[*node*][0] = *par*;

    maxEdge[*node*][0] = *w*;

    for (Edge e : adj[*node*]) {

        if (e.to != *par*) {

            depth[e.to] = depth[*node*] + 1;

            dfs(e.to, *node*, e.weight);

        }

    }

}

void preprocess() {

    for (int j = 1; j < LOG; j++) {

        for (int i = 1; i <= N; i++) {

            if (parent[i][j - 1] != -1) {

                parent[i][j] = parent[parent[i][j - 1]][j - 1];

                maxEdge[i][j] = max(maxEdge[i][j - 1], maxEdge[parent[i][j - 1]][j - 1]);

            }

        }

    }

}

int getMaxEdge(int *u*, int *v*) {

    if (depth[*u*] < depth[*v*]) swap(*u*, *v*);

    int maxW = 0;

    int diff = depth[*u*] - depth[*v*];

    for (int i = LOG - 1; i >= 0; i--) {

        if (diff & (1 << i)) {

            maxW = max(maxW, maxEdge[*u*][i]);

*u* = parent[*u*][i];

        }

    }

    if (*u* == *v*) return maxW;

    for (int i = LOG - 1; i >= 0; i--) {

        if (parent[*u*][i] != parent[*v*][i]) {

            maxW = max(maxW, max(maxEdge[*u*][i], maxEdge[*v*][i]));

*u* = parent[*u*][i];

*v* = parent[*v*][i];

        }

    }

    return max(maxW, max(maxEdge[*u*][0], maxEdge[*v*][0]));

}

int main() {

    ios::sync\_with\_stdio(false);

    cin.tie(0);

    cout << "Enter the number of nodes (N): ";

    cout.flush();

    cin >> N;

    cout << "Enter " << N - 1 << " edges in the format (u v w) where:\n";

    cout << "u = node1, v = node2, w = weight of the edge between u and v\n";

    cout.flush();

    for (int i = 1; i < N; i++) {

        int u, v, w;

        cout << "Edge " << i << ": ";

        cout.flush();

        cin >> u >> v >> w;

        adj[u].push\_back({v, w});

        adj[v].push\_back({u, w});

    }

    dfs(1, -1, 0);

    preprocess();

    long long E = 0;

    for (int i = 1; i <= N; i++) {

        for (int j = i + 1; j <= N; j++) {

            int maxW = getMaxEdge(i, j);

            E = (E + 1LL \* maxW \* i \* j) % MOD;

        }

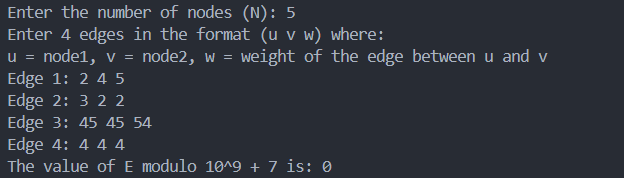
    }

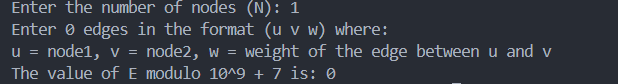
    cout << "The value of E modulo 10^9 + 7 is: " << E << endl;

    return 0;

}

1. Output with verity of test cases





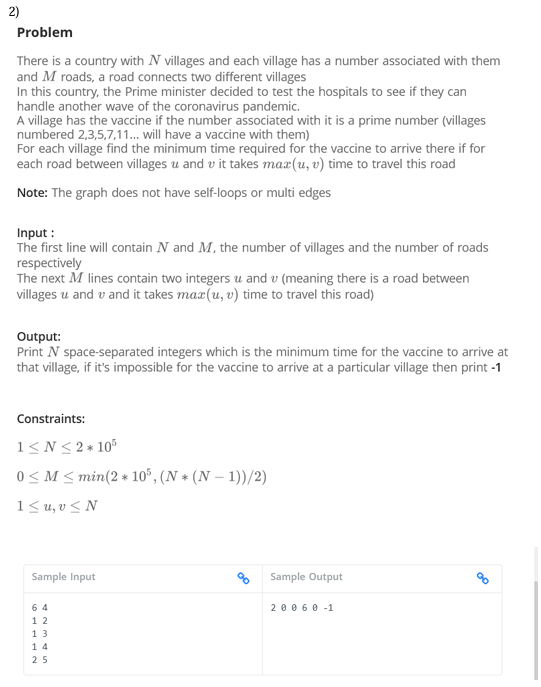
1. Analysis in terms of complexity wherever applicable.

**Time Complexity:**

* **Preprocessing (DFS + Binary Lifting Setup)**: O(NlogN)
* **Querying LCA and Max Weight**: O(logN) for each pair.
* **Total Complexity**: O(N^2 log N)

This is efficient enough for large N, and avoids the O(N^2) time complexity of a brute-force approach.

**Problem Statement:**

****

1. Algorithm/Pseudocode

**1. Input Parsing:**

* Read the number of villages n and the number of roads mmm.
* The villages are numbered from 1 to n.
* The roads between villages are provided as mmm edges, where each edge connects two villages with an undirected road.

**2. Prime Number Detection (Sieve of Eratosthenes):**

* Use the Sieve of Eratosthenes to determine which village numbers are prime.

**Steps for Sieve**:

1. Initialize a boolean array is\_prime[] of size n+1, marking all numbers as prime initially.
2. Mark is\_prime[0] and is\_prime[1] as false because 0 and 1 are not prime.
3. Iterate over numbers from 2 to n\sqrt{n}n​. For each number iii:
   * If i is prime, mark all its multiples i×2,i×3 as not prime.
4. After running the sieve, is\_prime[i] will be true if iii is prime and false otherwise.

**3. Graph Construction:**

* Create an adjacency list to represent the road network between villages.

**Steps**:

1. Initialize an adjacency list adj[] of size n+1
2. For each of the mmm edges, read the pair of connected villages u and v
3. Add village v to the adjacency list of village u, and vice versa, since roads are bidirectional.

**4. Dijkstra's Algorithm:**

* Use **Dijkstra's algorithm** to find the shortest paths from all prime-numbered villages to other villages. The weight of a road between two villages uuu and vvv is the maximum of their indices.

**Steps for Dijkstra**:

1. Initialize a distance array distance[] of size n+1, setting all distances to infinity (INF).
2. Initialize a priority queue pq for processing the villages by their current shortest known distance.
3. For each village iii such that is\_prime[i] is true, set distance[i] = 0 and add (0,i) to the priority queue, where 0 is the initial distance for prime-numbered villages.
4. While the priority queue is not empty:
   1. Extract the village u with the smallest distance from the queue.
   2. If the extracted distance d is greater than the current distance[u], continue (since we are processing an outdated state).
   3. For each neighboring village v of village u from the adjacency list:
      * Calculate the weight of the road as w=max(u,v)
      * If distance[u] + w < distance[v], update the distance for village v to distance[u] + w.
      * Add (distance[v], v) to the priority queue.

**5. Output:**

* For each village iii, print its shortest distance from any prime-numbered village.
* If a village is unreachable, print -1 instead.

Program Code

#include <iostream>

#include <vector>

#include <queue>

#include <climits>

#include <cmath>

using namespace std;

const int MAX\_N = 2e5 + 5;

vector<pair<int, int>> adj[MAX\_N];

int dist[MAX\_N];

bool isPrime[MAX\_N];

void sieve() {

    fill(isPrime, isPrime + MAX\_N, true);

    isPrime[0] = isPrime[1] = false;

    for (int i = 2; i \* i < MAX\_N; i++) {

        if (isPrime[i]) {

            for (int j = i \* i; j < MAX\_N; j += i) {

                isPrime[j] = false;

            }

        }

    }

}

void dijkstra(int *n*) {

    priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

    fill(dist, dist + *n* + 1, INT\_MAX);

    for (int i = 1; i <= *n*; i++) {

        if (isPrime[i]) {

            dist[i] = 0;

            pq.push({0, i});

        }

    }

    while (!pq.empty()) {

        int u = pq.top().second;

        int d = pq.top().first;

        pq.pop();

        if (d > dist[u]) continue;

        for (auto &edge : adj[u]) {

            int v = edge.first;

            int w = edge.second;

            if (dist[u] + w < dist[v]) {

                dist[v] = dist[u] + w;

                pq.push({dist[v], v});

            }

        }

    }

}

int main() {

    int n, m;

    cout << "Enter the number of villages (N) and roads (M): ";

    cin >> n >> m;

    sieve();

    cout << "Enter " << m << " roads (u v):" << endl;

    for (int i = 0; i < m; i++) {

        int u, v;

        cin >> u >> v;

        int w = max(u, v);

        adj[u].push\_back({v, w});

        adj[v].push\_back({u, w});

    }

    dijkstra(n);

    cout << "Minimum time for vaccine to arrive at each village:" << endl;

    for (int i = 1; i <= n; i++) {

        if (dist[i] == INT\_MAX) {

            cout << "-1 ";

        } else {

            cout << dist[i] << " ";

        }

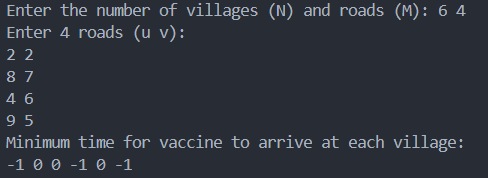
    }

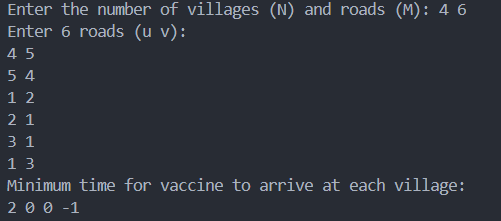
    cout << endl;

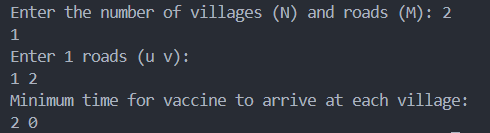
    return 0;

}

1. Output with verity of test cases







Analysis in terms of complexity wherever applicable.

**Time Complexity :**

- Worst case: O((N + M) log N), where N is the number of villages and M is the number of roads. This occurs when all villages are connected and we need to process all edges.

- Average case: O((N + M) log N), same as the worst case.

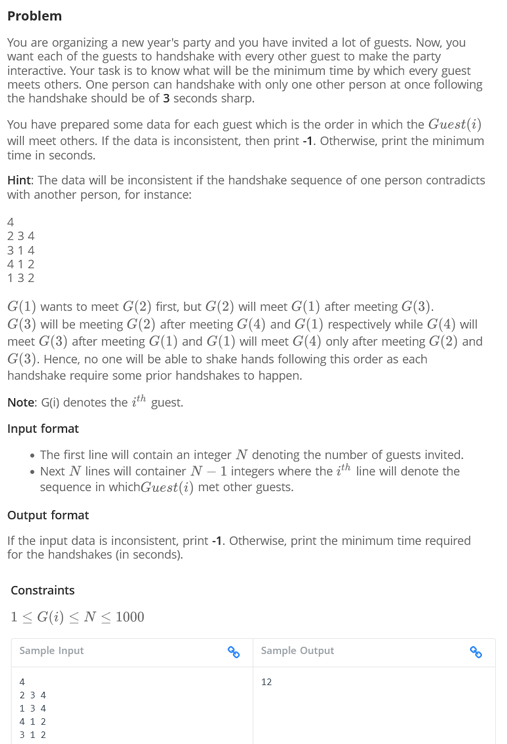
- Best case: O(N), when all villages are prime-numbered and no roads need to be processed.

**Space Complexity**: O(N + M)

- We use O(N) space for the distance array and the prime sieve.

- We use O(M) space for the adjacency list to store the roads.

**Problem Statement:**



1. Algorithm/Pseudocode

**Input Parsing**:

* Read the integer N (number of guests).
* For each guest i, read the sequence of N−1 guests in the order they should handshake.

**Graph Construction**:

* Represent each guest as a node in a graph.
* For each guest i, create directed edges pointing to the guests they should handshake with in the given order. Specifically, if guest iii needs to handshake with guest j, create an edge from i to j.

**Detecting Cycles**:

* Use **Kahn's Algorithm** for topological sorting. This algorithm works by processing nodes with an in-degree of zero.
* Initialize an in-degree array, where in\_degree[i] represents how many guests need to handshake with guest iii before iii can handshake with others.
* If any guest is part of a cycle (i.e., the topological sort cannot process all guests), the sequence is inconsistent, and we print -1.

**Calculate Minimum Time**:

* If the graph is acyclic, perform topological sorting.
* Initialize a time[] array where time[i] stores the minimum time required for guest iii to complete all handshakes.
* For each guest iii, update the time for its neighbors (next handshakes in the sequence) to be time[i] + 3 seconds.

**Output**:

* If the graph is valid (no cycles), print the maximum value in the time[] array, which represents the minimum time required for all guests to complete their handshakes.
* If a cycle is detected, print -1.

1. Program Code

#include <iostream>

#include <vector>

#include <queue>

#include <algorithm>

using namespace std;

// Function to find the minimum handshake time using Kahn's Algorithm

int find\_minimum\_handshake\_time(int *n*, vector<vector<int>>& *handshake\_order*) {

    vector<vector<int>> graph(*n* + 1); // Adjacency list representation of the graph

    vector<int> in\_degree(*n* + 1, 0);  // In-degree array to track dependencies

    vector<int> time(*n* + 1, 0);       // Time array to store the time for each guest

    // Build the graph and calculate in-degrees

    for (int i = 0; i < *n*; ++i) {

        int guest = i + 1;  // Guests are 1-indexed

        for (int j : *handshake\_order*[i]) {

            graph[guest].push\_back(j);  // Create a directed edge from guest to other guests

            in\_degree[j]++;  // Increment the in-degree of the guest j

        }

    }

    // Print graph structure and in-degrees for debugging

    cout << "Graph Structure:\n";

    for (int i = 1; i <= *n*; ++i) {

        cout << "Guest " << i << " -> ";

        for (int j : graph[i]) {

            cout << j << " ";

        }

        cout << endl;

    }

    cout << "In-degrees:\n";

    for (int i = 1; i <= *n*; ++i) {

        cout << "Guest " << i << ": " << in\_degree[i] << endl;

    }

    // Step 1: Initialize the queue with nodes having zero in-degree

    queue<int> q;

    for (int guest = 1; guest <= *n*; ++guest) {

        if (in\_degree[guest] == 0) {

            q.push(guest);  // Add guests with no dependencies to the queue

        }

    }

    // Step 2: Process the graph using Kahn's Algorithm (topological sorting)

    int processed\_count = 0;

    int max\_time = 0;

    while (!q.empty()) {

        int guest = q.front();

        q.pop();

        processed\_count++;

        // Process all the neighbors (other guests the current guest wants to meet)

        for (int neighbor : graph[guest]) {

            in\_degree[neighbor]--;

            time[neighbor] = max(time[neighbor], time[guest] + 3);  // Update the time for neighbor

            if (in\_degree[neighbor] == 0) {

                q.push(neighbor);  // Add to the queue if it has no more dependencies

            }

        }

        max\_time = max(max\_time, time[guest]);

    }

    // Step 3: Check if we processed all guests (if there's no cycle)

    if (processed\_count == *n*) {

        return max\_time;

    } else {

        return -1;  // If not all guests were processed, there is a cycle

    }

}

int main() {

    int n;

    cin >> n;

    vector<vector<int>> handshake\_order(n);

    // Reading input: N-1 sequences of handshakes for each guest

    for (int i = 0; i < n; ++i) {

        handshake\_order[i].resize(n - 1);

        for (int j = 0; j < n - 1; ++j) {

            cin >> handshake\_order[i][j];

        }

    }

    // Debugging print for input

    cout << "Handshake Order Input:\n";

    for (int i = 0; i < n; ++i) {

        cout << "Guest " << i + 1 << ": ";

        for (int j : handshake\_order[i]) {

            cout << j << " ";

        }

        cout << endl;

    }

    // Find and output the minimum time for handshakes or -1 if inconsistent

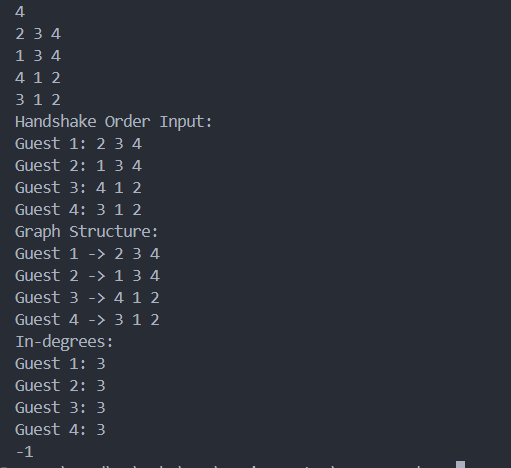
    int result = find\_minimum\_handshake\_time(n, handshake\_order);

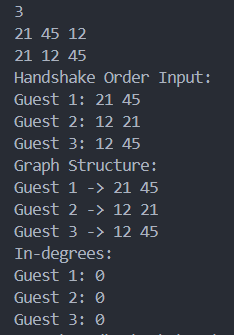
    cout << result << endl;

    return 0;

}

1. Output with verity of test cases





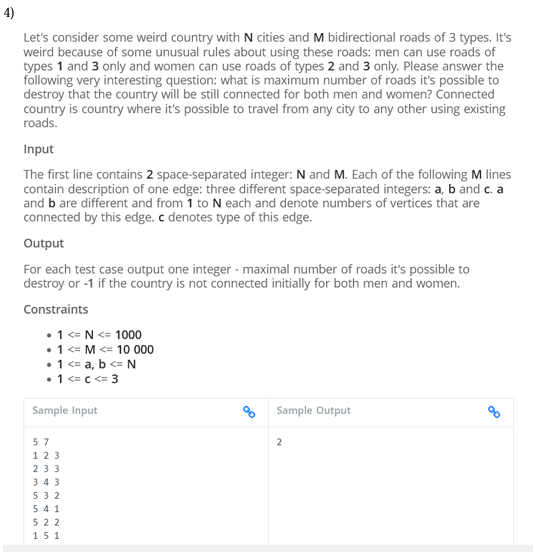
1. Analysis in terms of complexity wherever applicable.

**Time Complexity:**

1. **Graph Construction**: Building the graph takes O(N^2)) since we have N guests and each guest lists N−1 others.
2. **Topological Sort (Kahn’s Algorithm)**: This takes O(N+E) where N is the number of guests and E is the number of edges in the graph (total handshakes).
3. **Overall Complexity**: Since is at most N(N−1), the overall time complexity is O(N^2)

This algorithm works efficiently for N≤1000

**Problem Statement:**

****

1. Algorithm/Pseudocode

**1. Input Parsing:**

* Read the number of cities N and the number of roads M.
* Each road is represented by three integers: u, v, and type, where:
  + u: City u.
  + v: City v.
  + type:
    - 1: Usable by men only.
    - 2: Usable by women only.
    - 3: Usable by both men and women.

**2. Initialize Data Structures:**

* Use the **Union-Find (DSU)** data structure for both men and women.
  + Create parent\_men[] and parent\_women[] to store the connected components for both men and women.
  + Initialize rank\_men[] and rank\_women[] to optimize the union operation.
* Maintain two MSTs:
  + One for men, which uses roads of type 1 and 3.
  + One for women, which uses roads of type 2 and 3.
* Initialize total\_edges\_used to 0 to count the total number of edges used in the MSTs.

**3. Apply Kruskal’s Algorithm:**

**Kruskal’s Algorithm** finds the MST using a greedy approach by sorting edges and adding the smallest edge that does not form a cycle.

**Step-by-Step MST Construction**:

**Men’s Graph**:

* Start with all roads of type 1 and 3.
* For each edge u-v:
  + If the cities u and v are not connected in parent\_men[], add the edge to the men’s MST using the **union operation** in the DSU.
  + Increment total\_edges\_used for every edge added.

**Women’s Graph**:

* Use roads of type 2 and 3 for women’s graph.
* For each edge u-v:
  + If the cities u and v are not connected in parent\_women[], add the edge to the women’s MST using the **union operation** in the DSU.
  + Increment total\_edges\_used for every edge added.

**4. Check for Full Connectivity:**

* After constructing the MSTs for men and women:
  + If there is any city that is not connected in either the men’s or the women’s MST, print -1 and exit (since the country cannot be connected for both men and women).
  + Use the **find operation** of DSU to check if all cities belong to the same connected component in both parent\_men[] and parent\_women[].

**5. Count the Removable Edges:**

* Calculate the maximum number of removable edges by subtracting the number of edges used in the MSTs from the total number of roads (M): Removable Edges=M−total\_edges\_used
* This gives the maximum number of roads that can be destroyed while still keeping the country connected for both men and women.

**6. Output the Result:**

* Print the number of removable edges.

Program Code

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

class DisjointSet {

public:

    vector<int> parent, rank;

    DisjointSet(int *n*) {

        parent.resize(*n* + 1);

        rank.resize(*n* + 1, 0);

        for (int i = 1; i <= *n*; i++) parent[i] = i;

    }

    int find(int *u*) {

        if (parent[*u*] != *u*)

            parent[*u*] = find(parent[*u*]);

        return parent[*u*];

    }

    bool unite(int *u*, int *v*) {

        int rootU = find(*u*), rootV = find(*v*);

        if (rootU == rootV) return false;

        if (rank[rootU] > rank[rootV]) {

            parent[rootV] = rootU;

        } else if (rank[rootU] < rank[rootV]) {

            parent[rootU] = rootV;

        } else {

            parent[rootV] = rootU;

            rank[rootU]++;

        }

        return true;

    }

};

int main() {

    int N, M;

    cout << "Enter the number of cities (N) and roads (M): ";

    cin >> N >> M;

    vector<tuple<int, int, int>> edges;

    cout << "Enter the roads (a, b, type):" << endl;

    for (int i = 0; i < M; i++) {

        int u, v, type;

        cin >> u >> v >> type;

        edges.push\_back(make\_tuple(u, v, type));

    }

    DisjointSet men(N), women(N);

    int totalEdges = 0;

    int usedEdgesMen = 0, usedEdgesWomen = 0;

    for (auto [u, v, type] : edges) {

        if (type == 3) {

            bool connectedMen = men.unite(u, v);

            bool connectedWomen = women.unite(u, v);

            if (connectedMen || connectedWomen) {

                totalEdges++;

                if (connectedMen) usedEdgesMen++;

                if (connectedWomen) usedEdgesWomen++;

            }

        }

    }

    for (auto [u, v, type] : edges) {

        if (type == 1) {

            if (men.unite(u, v)) {

                totalEdges++;

                usedEdgesMen++;

            }

        } else if (type == 2) {

            if (women.unite(u, v)) {

                totalEdges++;

                usedEdgesWomen++;

            }

        }

    }

    bool fullyConnectedMen = true, fullyConnectedWomen = true;

    for (int i = 1; i <= N; i++) {

        if (men.find(i) != men.find(1)) fullyConnectedMen = false;

        if (women.find(i) != women.find(1)) fullyConnectedWomen = false;

    }

    if (fullyConnectedMen && fullyConnectedWomen) {

        cout << "Maximum number of roads that can be destroyed: " << M - totalEdges << endl;

    } else {

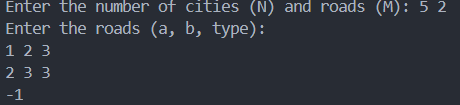
        cout << "-1" << endl;

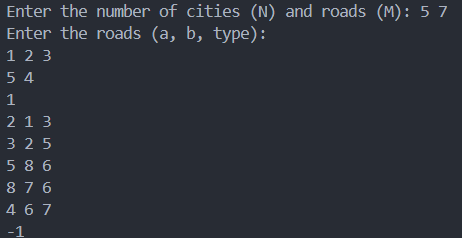
    }

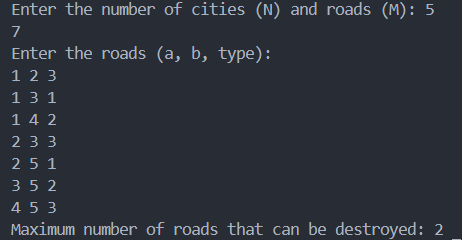
    return 0;

}

1. Output with verity of test cases







1. Analysis in terms of complexity wherever applicable.

**Sorting the Edges**:

* Sorting all M edges in Kruskal’s algorithm takes O(M logM)

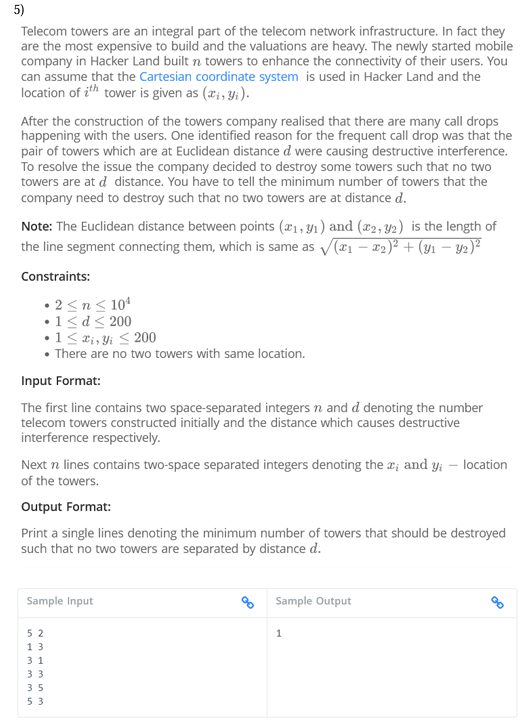
**Union-Find (DSU) Operations**:

* Each Union-Find operation (find and union) has a near-constant time complexity of O(α(N)), where α is the inverse Ackermann function, which grows very slowly. Hence, each edge insertion takes O(α(N))
* Across M edges, the total time for DSU operations is O(Mα(N)).

**Final Time Complexity**:

* The overall time complexity is dominated by the sorting step: O(MlogM+Mα(N))≈O(MlogM)
* This is efficient enough for N,M≤1000

**Problem Statement:**

****

1. Algorithm/Pseudocode

**1. Function canDestroy(towers, d, k):**

* **Input**:
  + towers[]: An array of towers where each tower has coordinates (x, y).
  + d: The required distance between two towers.
  + k: The number of towers to destroy.
* **Output**: Returns true if it is possible to destroy k towers such that no two remaining towers are exactly d units apart.

1.1 Initialize an array destroyed[] of the same size as towers, with all values set to false to mark which towers are destroyed.

1.2 Initialize a variable count to 0, which will track the number of clusters of towers that remain.

1.3 Iterate through the list of towers (i from 0 to n-1):

* If destroyed[i] is true, skip this tower.
* Increment count to indicate a new cluster.

1.4 For every tower i, check all other towers j (j > i):

* If destroyed[j] is true, skip tower j.
* Calculate the distance squared between tower i and tower j using: distance\_squared (x\_i - x\_j)^2 + (y\_i - y\_j)^
* If this distance is equal to d^2, mark destroyed[j] = true.

1.5 After checking all towers, if the total number of clusters (n - count) is less than or equal to k, return true; otherwise, return false.

**2. Function minTowersToDestroy(towers, d):**

* **Input**:
  + towers[]: Array of towers with their coordinates.
  + d: The required distance between two towers.
* **Output**: Minimum number of towers that must be destroyed.

2.1 Set left = 0 (minimum possible towers to destroy) and right = n-1 (maximum towers to destroy).

2.2 Perform binary search to find the minimum number of towers to destroy:

* While left < right:
  + Calculate mid = left + (right - left) / 2.
  + If canDestroy(towers, d, mid) is true, set right = mid, meaning it is possible to destroy mid towers.
  + Otherwise, set left = mid + 1.

2.3 Once the binary search is complete, left will hold the minimum number of towers that must be destroyed. Return left.

**3. Main Function:**

* **Input**:
  + n: The number of towers.
  + d: The required distance between two towers.
  + towers[]: Coordinates of n towers.

3.1 Read n and d.

3.2 Initialize the array towers[] with the coordinates of each tower.

3.3 Call the function minTowersToDestroy(towers, d) and print the result.

Program Code

#include <iostream>

#include <vector>

#include <cmath>

#include <algorithm>

using namespace std;

struct Tower {

    int x, y;

};

bool canDestroy(const vector<Tower>& *towers*, int *d*, int *k*) {

    vector<bool> destroyed(*towers*.size(), false);

    int count = 0;

    for (int i = 0; i < *towers*.size(); i++) {

        if (destroyed[i]) continue;

        count++;

        for (int j = i + 1; j < *towers*.size(); j++) {

            if (destroyed[j]) continue;

            int dx = *towers*[i].x - *towers*[j].x;

            int dy = *towers*[i].y - *towers*[j].y;

            if (dx \* dx + dy \* dy == *d* \* *d*) {

                destroyed[j] = true;

            }

        }

    }

    return *towers*.size() - count <= *k*;

}

int minTowersToDestroy(const vector<Tower>& *towers*, int *d*) {

    int left = 0, right = *towers*.size() - 1;

    while (left < right) {

        int mid = left + (right - left) / 2;

        if (canDestroy(*towers*, *d*, mid)) {

            right = mid;

        } else {

            left = mid + 1;

        }

    }

    return left;

}

int main() {

    int n, d;

    cout << "Enter the number of towers (n) and the interference distance (d): ";

    cin >> n >> d;

    vector<Tower> towers(n);

    cout << "Enter the coordinates of each tower (x y):" << endl;

    for (int i = 0; i < n; i++) {

        cout << "Tower " << i + 1 << ": ";

        cin >> towers[i].x >> towers[i].y;

    }

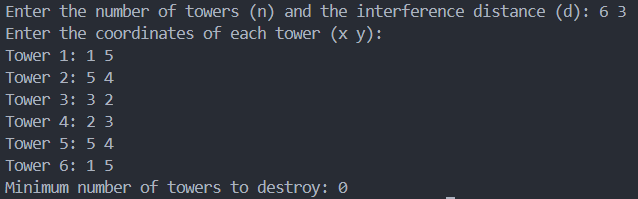
    int result = minTowersToDestroy(towers, d);

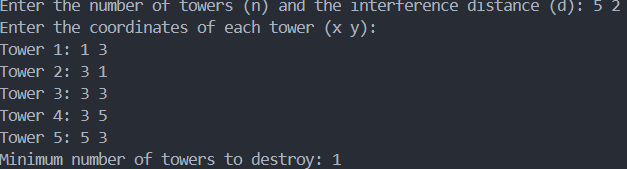
    cout << "Minimum number of towers to destroy: " << result << endl;

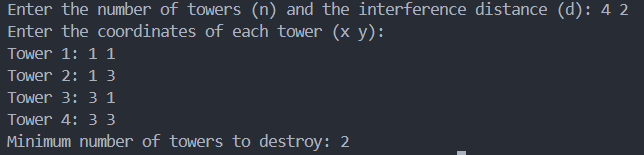
    return 0;

}

1. Output with verity of test cases







1. Analysis in terms of complexity wherever applicable.

**1. Time Complexity:**

* The primary bottleneck in the algorithm is the canDestroy function and the binary search over possible values of k.

**Binary Search (minTowersToDestroy)**:

* We perform binary search on the range [0, n-1]. The binary search takes O(logn) iterations.

**Function canDestroy**:

* For each call to can Destroy, we iterate over all pairs of towers (i, j), which takes O(n^2) in the worst case since we compare distances for each pair of towers.

Therefore, the overall time complexity is:

O(logn) O(n^2) = O(n^2 log n)

**2. Space Complexity:**

* The space complexity comes from storing the array of towers and the destroyed flags, which is O(n). Therefore, the space complexity is O(n).