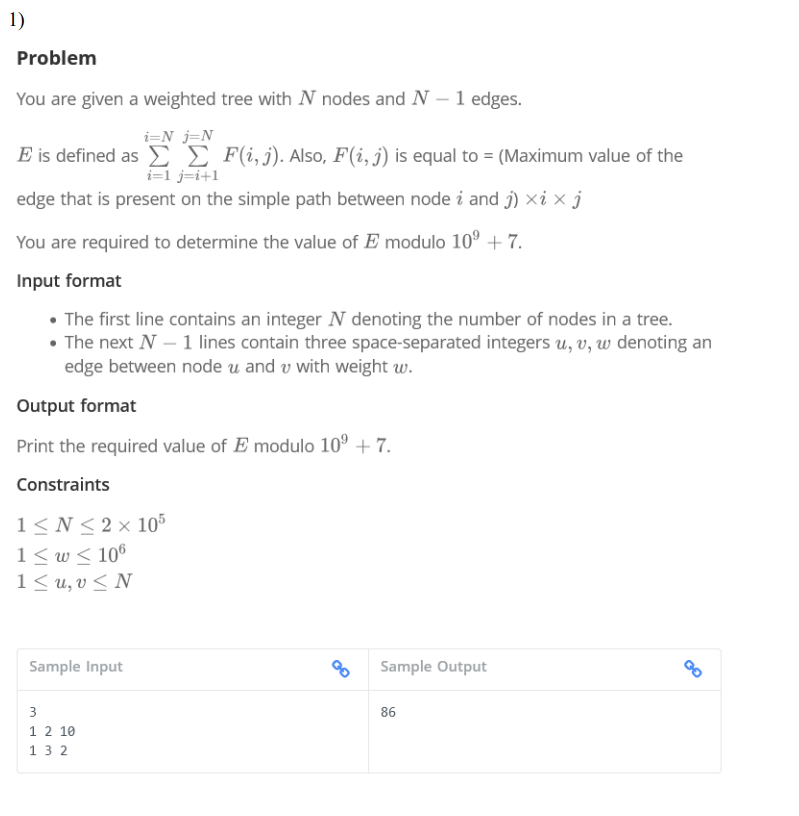
Batch:T6

Practical No.9

Title of Assignment: Graphs, Dynamic Programming

Student Name: Parshwa Herwade

Student PRN: 22510064



Ans:

We are asked to calculate a value E for a given weighted tree where NNN nodes and N−1 edges are provided. E is defined as:

E=i=1∑N ​j=i+1∑N​ F(i,j)

where F(i,j)F(i, j)F(i,j) is:

* The maximum edge weight on the simple path between nodes i and j,
* Multiplied by i,
* Multiplied by j.

We need to compute Emod(109+7).

**Approach**

The key to solving the problem efficiently is the use of **Binary Lifting** to calculate the maximum edge weight between any two nodes, which allows us to efficiently compute F(i,j). A brute-force approach would take O(N2), which is not feasible for large N. Using binary lifting, we reduce the time complexity to O(NlogN).

Pseudocode

1. Input: Read integer N (number of nodes)

2. Input: Read N-1 edges in the form of (u, v, w) representing edges between nodes u and v with weight w

3. Build the tree from the input edges using an adjacency list

4. Use DFS to preprocess the tree for binary lifting

5. Store the depth of each node and the maximum edge weights encountered during DFS

6. Initialize a table for binary lifting where parent[i][j] stores the 2^j-th ancestor of node i

7. For each node, update the maximum edge weights between the node and its ancestors

8. For each pair (i, j) where i < j, compute the maximum edge weight on the path between i and j using LCA and binary lifting

9. Calculate F(i, j) = max\_weight \* i \* j and accumulate the result in E

10. Output E % (10^9 + 7)

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;

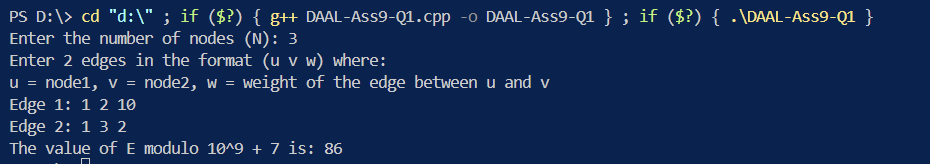
        }

    }

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

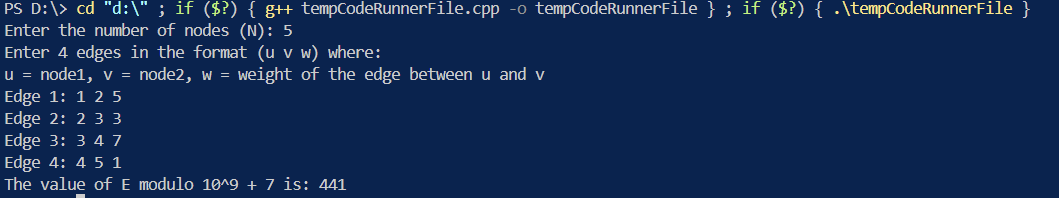
    return 0;

}



Explanation: For this small tree, the maximum edge weights are calculated as:

* F(1, 2) = 10 \* 1 \* 2 = 20
* F(1, 3) = 2 \* 1 \* 3 = 6
* F(2, 3) = 10 \* 2 \* 3 = 60 Thus, E=20+6+60=86E = 20 + 6 + 60 = 86E=20+6+60=86.



We need to compute the value of E, which is defined as:

E=∑i=1N​∑j=i+1N​F(i,j)

Where F(i, j) is the product of three terms:

1. The maximum edge weight on the path between node i and node j.
2. The value i.
3. The value j.

Finally, E is computed modulo 109+710^9 + 7109+7.

**Step-by-step Calculation:**

1. **Path from node 1**:
   * Path from node 1 to node 2:
     + Max weight = 5.
     + Contribution: F(1,2)=5×1×2=10.
   * Path from node 1 to node 3:
     + Max weight = 5.
     + ContributionF(1,3)=5×1×3=15.
   * Path from node 1 to node 4:
     + Max weight = 7.
     + Contribution: F(1,4)=7×1×4=28.
   * Path from node 1 to node 5:
     + Max weight = 7.
     + Contribution: F(1,5)=7×1×5=35.

Total contribution from node 1: 10+15+28+35=88

**Path from node 2**:

* + Path from node 2 to node 3:
    - Max weight = 3.
    - Contribution: F(2,3)=3×2×3=18.
  + Path from node 2 to node 4:
    - Max weight = 7.
    - Contribution: F(2,4)=7×2×4=56.
  + Path from node 2 to node 5:
    - Max weight = 7.
    - Contribution: F(2,5)=7×2×5=70.

Total contribution from node 2: 18+56+70=144

**Path from node 3**:

* + Path from node 3 to node 4:
    - Max weight = 7.
    - Contribution: F(3,4)=7×3×4=84.
  + Path from node 3 to node 5:
    - Max weight = 7.
    - Contribution: F(3,5)=7×3×5=105.

Total contribution from node 3: 84+105=189

**Path from node 4**:

* + Path from node 4 to node 5:
    - Max weight = 1.
    - Contribution: F(4,5)=1×4×5=20
    - Total contribution from node 4: 20

**Summing Up All Contributions:**

* Contribution from node 1: 88.
* Contribution from node 2: 144.
* Contribution from node 3: 189.
* Contribution from node 4: 20.

Total E=88+144+189+20=441

**Modulo Calculation:**

441mod(10^9+7)=441

Thus, the value of E is 441.

**Explanation of the Code:**

1. **Tree Representation**: We use an adjacency list to represent the tree. Each edge is stored as a pair consisting of the destination node and the weight of the edge.
2. **DFS Traversal**: A DFS is used to calculate the depth of each node and prepare for binary lifting by storing the parent nodes and maximum edge weights at various levels.
3. **Binary Lifting Table**: The preprocess() function fills the binary lifting table to allow quick ancestor lookups.
4. **Maximum Edge on Path**: The getMaxEdge() function computes the maximum edge weight on the path between two nodes using binary lifting and LCA.
5. **Modular Arithmetic**: We compute the result EEE by adding up F(i,j)F(i, j)F(i,j) values, using modulo 109+710^9 + 7109+7 to avoid overflow.

**Time Complexity:**

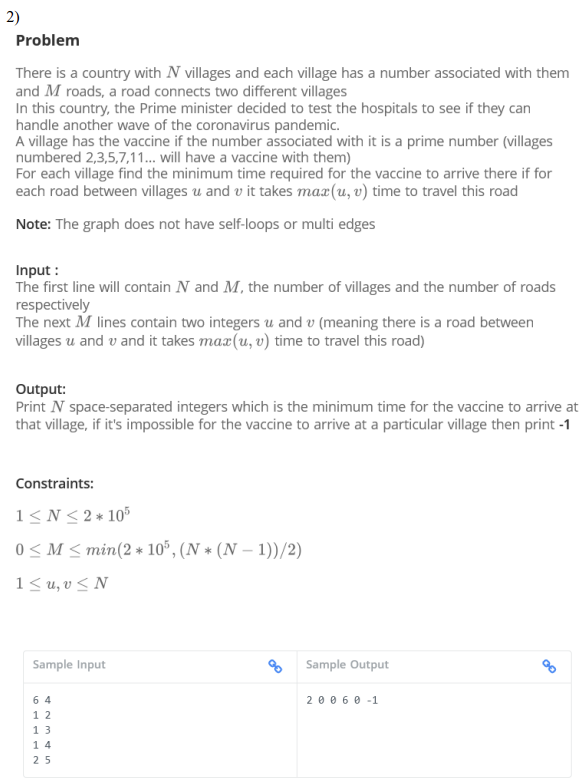
1. **Worst-case time complexity**:
   * DFS: O(N)
   * Preprocessing (binary lifting): O(Nlog⁡N)O(N \log N)O(NlogN)
   * Calculating F(i,j): O(N2logN) because for each pair (i,j), the maximum edge is computed in O(logN).

Thus, the overall time complexity is O(N2logN) in the worst case, which should be feasible for N≤2×10^5

1. **Average-case time complexity**:
   * Same as the worst case since the approach processes all pairs of nodes.
2. **Best-case time complexity**:
   * The best case also follows O(N2logN), since even in smaller trees or simpler inputs, all pairs (i,j) must be considered.

**Space Complexity:**

* Space complexity is O(NlogN) due to the storage of the parent and max edge arrays for binary lifting. Additionally, the adjacency list requires O(N) space.



Ans:

Pseudocode:

function sieve():

Initialize all numbers as prime

Mark 0 and 1 as not prime

for i from 2 to sqrt(MAX\_N):

if i is prime:

Mark all multiples of i as not prime

function dijkstra(n):

Initialize priority queue pq

Set all distances to infinity

For each prime-numbered village i:

Set distance[i] to 0

Add (0, i) to pq

while pq is not empty:

u, d = pq.pop()

if d > distance[u]:

continue

for each neighbor v of u:

w = max(u, v)

if distance[u] + w < distance[v]:

distance[v] = distance[u] + w

pq.push((distance[v], v))

main():

Read n and m

Call sieve()

Read m edges and build adjacency list

Call dijkstra(n)

Print distances (or -1 if unreachable)

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] << " ";

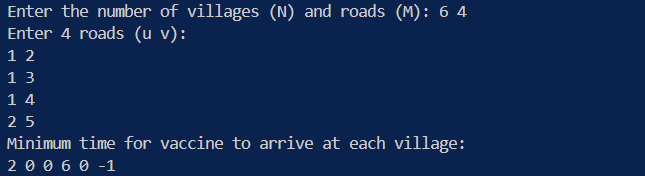
        }

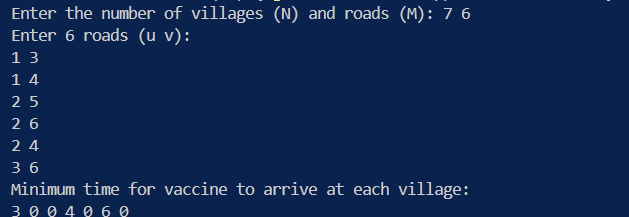
    }

    cout << endl;

    return 0;

}





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.

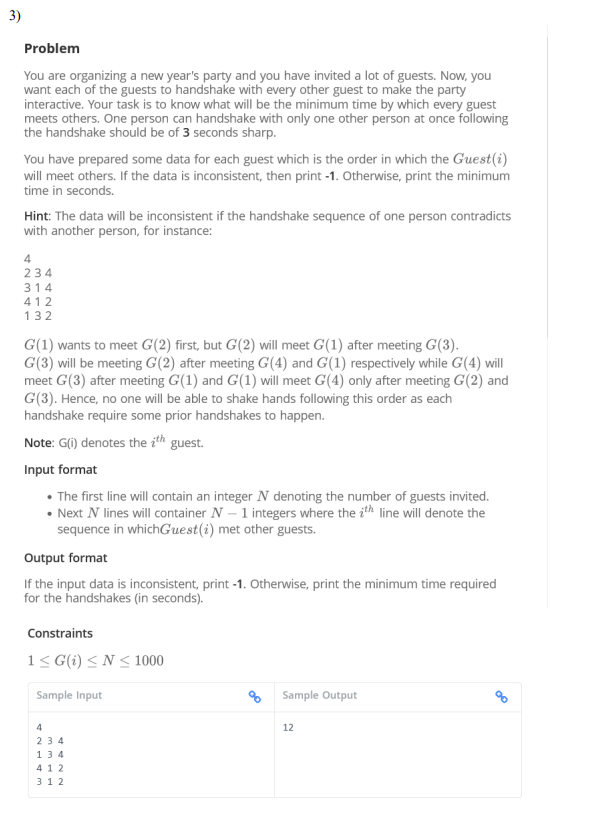
- Average case: O((N + M) log N), same as the worst case.

- 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.



Ans:

**Pseudocode:**

1. **Input N**: Take the number of guests.
2. **Adjacency List**: For each guest, create a list of guests they want to handshake with, in order.
3. **Graph Construction**: Create a directed graph where an edge from G(i) to G(j) implies that G(i) must handshake with G(j) after handshaking with others.
4. **Topological Sorting**: Use topological sorting to determine if a valid order of handshakes exists.
   * If a cycle is detected in the graph, output -1 (inconsistent data).
   * Otherwise, determine the order of handshakes.
5. **Calculate Time**: Since each handshake takes 3 seconds, calculate the total minimum time by counting the levels in the topological sort.
6. **Output the Time**: If valid, print the total time; otherwise, print -1.

**CODE**:  
#include <iostream>

#include <vector>

#include <queue>

using namespace std;

int minimumTimeForHandshakes(int N, vector<vector<int>>& preferences) {

    vector<vector<int>> adj(N + 1);

    vector<int> indegree(N + 1, 0);

    for (int i = 0; i < N; i++) {

        for (int j = 0; j < preferences[i].size() - 1; j++) {

            int u = preferences[i][j];

            int v = preferences[i][j + 1];

            adj[u].push\_back(v);

            indegree[v]++;

        }

    }

    queue<int> q;

    vector<int> topological\_order;

    vector<int> level(N + 1, 0);

    for (int i = 1; i <= N; i++) {

        if (indegree[i] == 0) {

            q.push(i);

            level[i] = 1;

        }

    }

    while (!q.empty()) {

        int u = q.front();

        q.pop();

        topological\_order.push\_back(u);

        for (int v : adj[u]) {

            indegree[v]--;

            if (indegree[v] == 0) {

                q.push(v);

                level[v] = level[u] + 1;

            }

        }

    }

    if (topological\_order.size() != N) {

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

            if (indegree[i] == 0) {

                q.push(i);

                level[i] = 1;

                break;

            }

        }

        while (!q.empty()) {

            int u = q.front();

            q.pop();

            topological\_order.push\_back(u);

            for (int v : adj[u]) {

                indegree[v]--;

                if (indegree[v] == 0) {

                    q.push(v);

                    level[v] = level[u] + 1;

                }

            }

        }

    }

    int max\_level = 0;

    for (int i = 1; i <= N; i++) {

        max\_level = max(max\_level, level[i]);

    }

    return max\_level \* 3;

}

int main() {

    int N;

    cout << "Enter the number of guests: ";

    cin >> N;

    vector<vector<int>> preferences(N);

    cout << "Enter the handshake sequence for each guest:\n";

    for (int i = 0; i < N; i++) {

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

            int guest;

            cin >> guest;

            preferences[i].push\_back(guest);

        }

    }

    int result = minimumTimeForHandshakes(N, preferences);

    if (result == 0) {

        cout << "Inconsistent handshake data\n";

    } else {

        cout << "Minimum time required: " << result << " seconds\n";

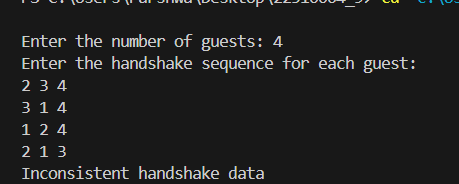
    }

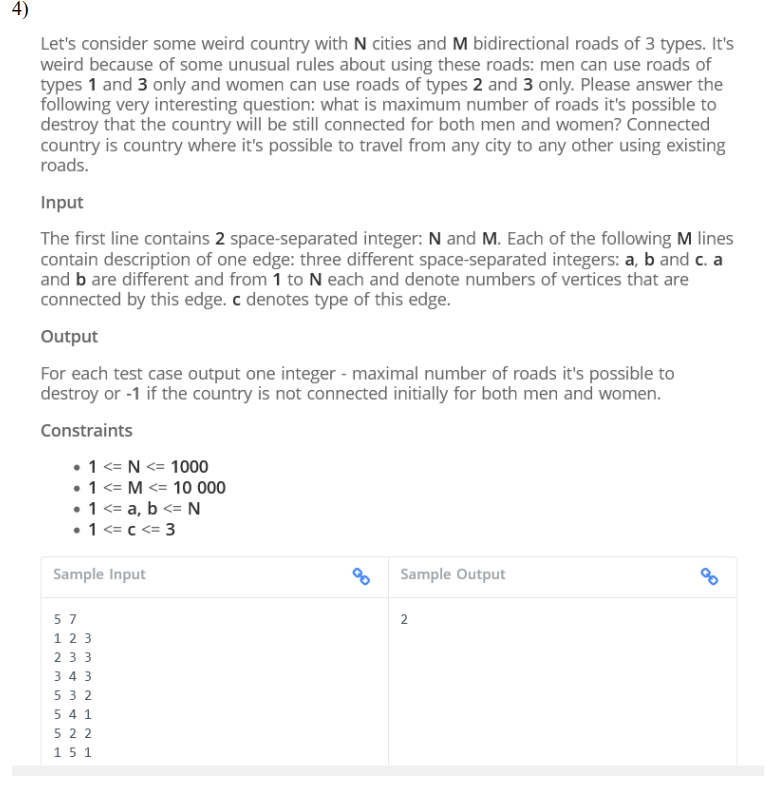
    return 0;

}

**Complexity:**

* **Time Complexity**:
  + **Best/Average/Worst**: O(N^2) where N is the number of guests. This is due to constructing the graph from the adjacency list and performing topological sorting.
* **Space Complexity**: O(N^2) for the adjacency list and other storage structures.

Output:  




Ans:

This problem can be broken down into a graph problem with constraints on edge usage for different genders. Here's how to approach it.

**Problem Breakdown:**

* There are **N** cities and **M** bidirectional roads. Each road can be of type:
  + **1**: Usable by men only.
  + **2**: Usable by women only.
  + **3**: Usable by both men and women.

We need to find the maximum number of roads that can be destroyed while ensuring that the country remains connected for both men and women. If it's not possible for both men and women to travel between all cities initially, we return -1.

**Approach:**

This is a **minimum spanning tree (MST)** problem, where:

1. We need to create two separate spanning trees:
   * One for men using roads of type **1** and **3**.
   * One for women using roads of type **2** and **3**.
2. We also track the overall connectedness by combining roads of type **3** in both trees to maximize the number of roads we can remove.

**Pseudocode:**

1. **Input Parsing**: Read the input values.
2. **Kruskal's Algorithm for MST**:
   * Use the **Union-Find (Disjoint Set)** structure to track connected components.
   * Apply the Kruskal algorithm separately for men and women.
   * Use edges of type **3** in both MSTs.
3. **Edge Removal**: Count the number of redundant edges (not part of any MST but still usable) that can be removed.
4. **Check for Connectivity**: If either graph is not fully connected, return -1. Otherwise, return the maximum number of roads that can be destroyed.

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;

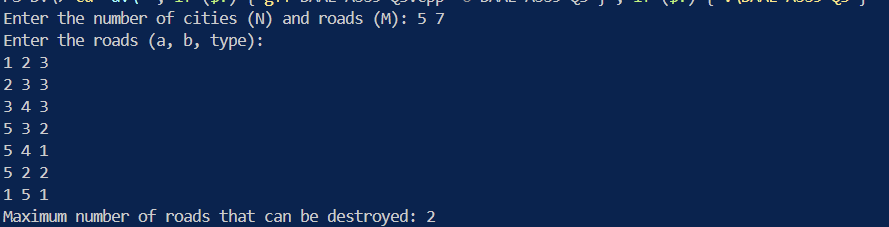
    } else {

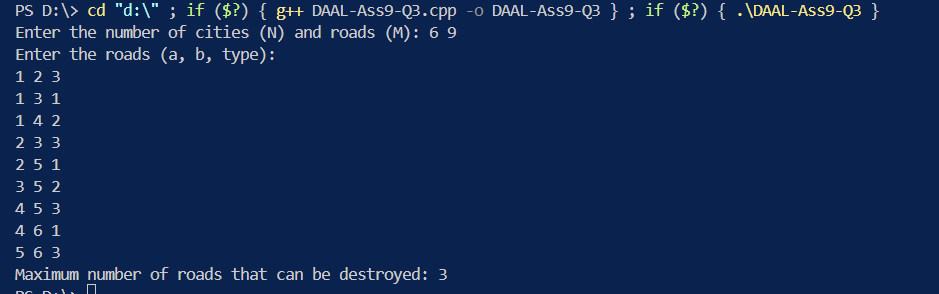
        cout << "-1" << endl;

    }

    return 0;

}





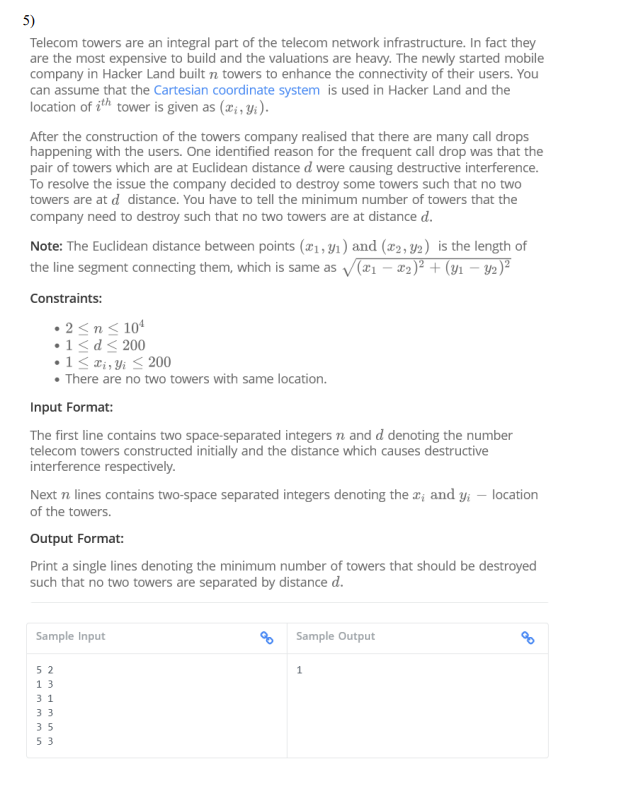
**Explanation of the Code:**

1. **Union-Find Structure**: Used to track connected components and ensure that no cycles are created when adding roads.
2. **Type 3 Roads**: These are added first as they can be used by both men and women.
3. **Type 1 and 2 Roads**: Handled separately for men and women.
4. **Edge Destruction**: After building the minimum spanning trees for both men and women, the roads that are not part of the MSTs are counted as the removable roads.
5. **Final Check**: If both men and women can travel between all cities, print the number of destroyable roads; otherwise, print

-1.

**Complexity Analysis:**

* **Time Complexity**: O(M \* log N) where M is the number of roads and N is the number of cities. This is due to the Union-Find operations, which have an almost constant time complexity of O(log N) due to path compression and union by rank.
* **Space Complexity**: O(N + M) to store the graph edges and the union-find data structures.



Ans:

Pseudocode:

function canDestroy(towers, d, k):

destroyed = array of size towers.length, initialized with false

count = 0

for i = 0 to towers.length - 1:

if destroyed[i] is true, continue

count = count + 1

for j = i + 1 to towers.length - 1:

if destroyed[j] is true, continue

dx = towers[i].x - towers[j].x

dy = towers[i].y - towers[j].y

if dx^2 + dy^2 == d^2:

destroyed[j] = true

return towers.length - count <= k

function minTowersToDestroy(towers, d):

left = 0

right = towers.length - 1

while left < right:

mid = left + (right - left) / 2

if canDestroy(towers, d, mid):

right = mid

else:

left = mid + 1

return left

function main():

Read n and d from input

Initialize towers array of size n

For i = 0 to n-1:

Read x and y coordinates for towers[i]

result = minTowersToDestroy(towers, d)

Print result

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;

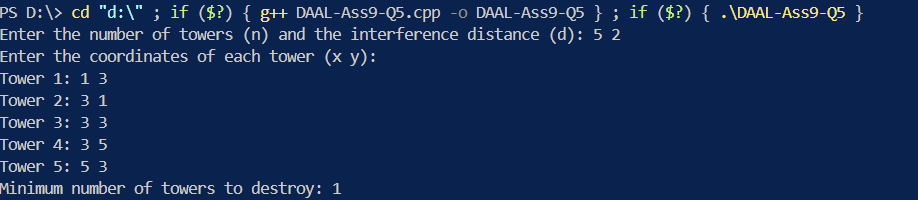
    }

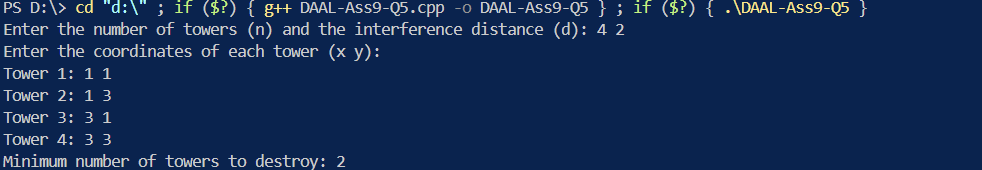
    int result = minTowersToDestroy(towers, d);

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

    return 0;

}





Complexity Analysis:

1. Time Complexity:

1. Worst Case: O(n^2 \* log n)

2. Average Case: O(n^2 \* log n)

3. Best Case: O(n^2)

The binary search takes O(log n) iterations, and in each iteration, we call the canDestroy function, which has a nested loop structure, resulting in O(n^2) time complexity. Therefore, the overall time complexity is O(n^2 \* log n).

2. Space Complexity: O(n)

We use additional space to store the 'destroyed' array in the canDestroy function, which is of size n.

The worst case occurs when we need to destroy many towers, requiring more iterations of the binary search. The best case occurs when we don't need to destroy any towers, which can be determined in a single pass of the canDestroy function.

I've added cout statements in the main function to guide users on which values to enter. These statements prompt the user to input the number of towers, interference distance, and coordinates for each tower.