DOMAIN WINTER CAMP

DAY - 7

Student Name: Riya Kungotra **UID:** 22BCS14298

Branch: BE-CSE **Section:** 22BCS_FL_IOT-603

Graph

Very Easy:

Find Center of Star Graph

There is an undirected star graph consisting of n nodes labeled from 1 to n. A star graph is a graph where there is one center node and exactly n - 1 edges that connect the center node with every other node.

You are given a 2D integer array edges where each edges[i] = [ui, vi] indicates that there is an edge between the nodes ui and vi. Return the center of the given star graph.

Solution:

```
#include <iostream>
#include <vector>
using namespace std;

class Solution {
public:
    int findCenter(vector<vector<int>>& edges) {
        // Compare the nodes in the first two edges
        if (edges[0][0] == edges[1][0] || edges[0][0] == edges[1][1]) {
            return edges[0][0];
        }
        return edges[0][1];
    }
};

int main() {
    Solution solution;

    vector<vector<int>> edges1 = {{1, 2}, {2, 3}, {4, 2}};
    cout << solution.findCenter(edges1) << endl;  // Output: 2

    vector<vector<int>> edges2 = {{1, 2}, {5, 1}, {1, 3}, {1, 4}};
    cout << solution.findCenter(edges2) << endl;  // Output: 1

    return 0;
```

}

Output:

```
2
1


=== Code Execution Successful ===
```

Easy

Find if Path Exists in Graph

There is a bi-directional graph with n vertices, where each vertex is labeled from 0 to n - 1 (inclusive). The edges in the graph are represented as a 2D integer array edges, where each edges[i] = [ui, vi] denotes a bi-directional edge between vertex ui and vertex vi. Every vertex pair is connected by at most one edge, and no vertex has an edge to itself.

You want to determine if there is a valid path that exists from vertex source to vertex destination.

Given edges and the integers n, source, and destination, return true if there is a valid path from source to destination, or false otherwise.

Input:

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_set>
using namespace std;

class Solution {
public:
    bool validPath(int n, vector<vector<int>>& edges, int source, int destination) {
        if (source == destination) return true;

        // Build the adjacency list
        vector<vector<int>> adj(n);
        for (auto& edge : edges) {
            adj[edge[0]].push_back(edge[1]);
            adj[edge[1]].push_back(edge[0]);
        }

        // BFS to find if a path exists
        queue<int> q;
        vector<bool> visited(n, false);
```

```
        q.push(source);
        visited[source] = true;

        while (!q.empty()) {
            int node = q.front();
            q.pop();
            if (node == destination) return true;
            for (int neighbor : adj[node]) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    q.push(neighbor);
                }
            }
        }
        return false;
    }
};

int main() {
    Solution solution;
    int n = 6;
    vector<vector<int>> edges = {{0, 1}, {0, 2}, {3, 5}, {5, 4}, {4, 3}};
    int source = 0, destination = 5;
    cout << (solution.validPath(n, edges, source, destination) ? "true" : "false") << endl;

    return 0;
}
```

Output:

```
false


=== Code Execution Successful ===
```

Medium

Course Schedule II

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you must take course bi first if you want to take course ai.

For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.
Return the ordering of courses you should take to finish all courses. If there are many valid answers, return any of them. If it is impossible to finish all courses, return an empty array.

Solution:

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

class Solution {
public:
    vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {
        vector<vector<int>> adj(numCourses); // Adjacency list
        vector<int> inDegree(numCourses, 0); // In-degree of each course
        vector<int> order; // Stores the topological order

        // Build the graph
        for (auto& prereq : prerequisites) {
            adj[prereq[1]].push_back(prereq[0]);
            inDegree[prereq[0]]++;
        }

        // Initialize the queue with courses having in-degree 0
        queue<int> q;
        for (int i = 0; i < numCourses; i++) {
            if (inDegree[i] == 0) {
                q.push(i);
            }
        }

        // Process the courses
        while (!q.empty()) {
            int course = q.front();
            q.pop();
            order.push_back(course);

            for (int neighbor : adj[course]) {
                inDegree[neighbor]--;
                if (inDegree[neighbor] == 0) {
                    q.push(neighbor);
                }
            }
        }

        // Check if all courses are processed
        if (order.size() == numCourses) {
            return order;
```

```
        } else {
            return {}; // Impossible to finish all courses
        }
    }
};

int main() {
    Solution solution;

    int numCourses = 4;
    vector<vector<int>> prerequisites = {{1, 0}, {2, 0}, {3, 1}, {3, 2}};
    vector<int> result = solution.findOrder(numCourses, prerequisites);
    for (int course : result) cout << course << " ";
    cout << endl;

    return 0;
}
```

Output:

```
0 1 2 3


=== Code Execution Successful ===
```

Hard

Max Area of Island

You are given an m x n binary matrix grid. An island is a group of 1's (representing land) connected 4-directionally (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

The area of an island is the number of cells with a value 1 in the island.

Return the maximum area of an island in grid. If there is no island, return 0.

Solution:

```
#include <iostream>
#include <vector>
using namespace std;

class Solution {
public:
    int maxAreaOfIsland(vector<vector<int>>& grid) {
        int maxArea = 0;
```

```
        int rows = grid.size(), cols = grid[0].size();

        // Lambda function for DFS
        auto dfs = [&](int r, int c, auto& dfs) -> int {
            if (r < 0 || c < 0 || r >= rows || c >= cols || grid[r][c] == 0)
                return 0;

            grid[r][c] = 0; // Mark as visited
            int area = 1;

            // Explore neighbors
            area += dfs(r + 1, c, dfs);
            area += dfs(r - 1, c, dfs);
            area += dfs(r, c + 1, dfs);
            area += dfs(r, c - 1, dfs);

            return area;
        };

        // Traverse the grid
        for (int r = 0; r < rows; ++r) {
            for (int c = 0; c < cols; ++c) {
                if (grid[r][c] == 1) {
                    maxArea = max(maxArea, dfs(r, c, dfs));
                }
            }
        }

        return maxArea;
    }
};

int main() {
    Solution solution;

    vector<vector<int>> grid1 = {
        {0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0},
        {0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0},
        {0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0}
    };

    cout << solution.maxAreaOfIsland(grid1) << endl;
```

```
    return 0;
}
```

Output:

```
6


=== Code Execution Successful ===
```

Very Hard

Network Delay Time

You are given a network of n nodes, labeled from 1 to n. You are also given times, a list of travel times as directed edges times[i] = (ui, vi, wi), where ui is the source node, vi is the target node, and wi is the time it takes for a signal to travel from source to target.

We will send a signal from a given node k. Return the minimum time it takes for all the n nodes to receive the signal. If it is impossible for all the n nodes to receive the signal, return -1.

Input:

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_map>
#include <climits>
#include <algorithm> // Include this for max_element

using namespace std;

class Solution {
public:
    int networkDelayTime(vector<vector<int>>& times, int n, int k) {
        vector<vector<pair<int, int>>> graph(n + 1); // Adjacency list
        for (const auto& time : times) {
            graph[time[0]].push_back({time[1], time[2]});
        }

        vector<int> dist(n + 1, INT_MAX); // Distance array
        dist[k] = 0;
```

```
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq; // Min-heap
        pq.push({0, k}); // {distance, node}

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

            if (time > dist[node]) continue;

            for (auto [neighbor, weight] : graph[node]) {
                if (dist[node] + weight < dist[neighbor]) {
                    dist[neighbor] = dist[node] + weight;
                    pq.push({dist[neighbor], neighbor});
                }
            }
        }

        int maxTime = *max_element(dist.begin() + 1, dist.end());
        return maxTime == INT_MAX ? -1 : maxTime;
    }
};

int main() {
    Solution solution;

    vector<vector<int>> times1 = {{2, 1, 1}, {2, 3, 1}, {3, 4, 1}};
    int n1 = 4, k1 = 2;
    cout << solution.networkDelayTime(times1, n1, k1) << endl;

    return 0;
}
```

Output:

```
2


=== Code Execution Successful ===
```