DOMAIN WINTER WINNING CAMP ASSIGNMENT

Student Name: Jagatjeet Singh   **UID: 22BCS14190**
Branch: BE-CSE::CS201   **Section/Group: 22BCS_FL_IOT-603/B**
Semester: 5th

➢ DAY-7 [26-12-2024]

1. Find Center of Star Graph   *(Very Easy)*

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.

Example 1:
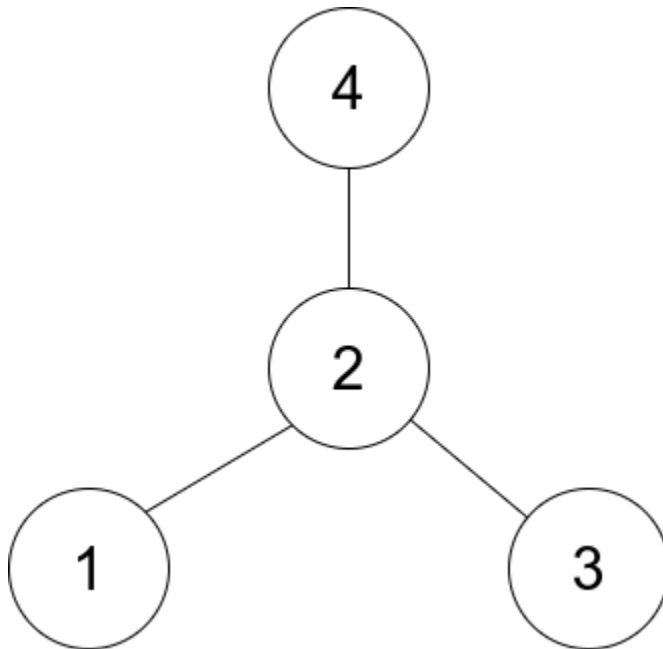
Input: edges = [[1,2],[2,3],[4,2]]

Output: 2

Explanation: As shown in the figure above, node 2 is connected to every other node, so 2 is the center.

Example 2:

Input: edges = [[1,2],[5,1],[1,3],[1,4]]

Output: 1

Constraints:

- $3 <= n <= 1e5$
- edges.length == n - 1
- edges[i].length == 2
- $1 <= ui, vi <= n$
- ui != vi

The given edges represent a valid star graph.

Implementation/Code:

```cpp
#include <iostream>
#include <vector>
#include <unordered_map>

using namespace std;

int findCenter(vector<vector<int>>& edges) {
    // Use a map to count the occurrences of each node in the edges
    unordered_map<int, int> count;

    // Iterate over the edges and count the occurrences of each node
    for (const auto& edge : edges) {
        count[edge[0]]++;
        count[edge[1]]++;
    }
```

```cpp
    // The center node will be the one that appears n-1 times
    for (const auto& entry : count) {
        if (entry.second == edges.size()) {
            return entry.first;
        }
    }

    return -1; // In case of an error (this shouldn't happen in a valid star graph)
}

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

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

    return 0;
}
```

Output:

2. Find if Path Exists in Graph   *(Easy)*

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.

Example 1:
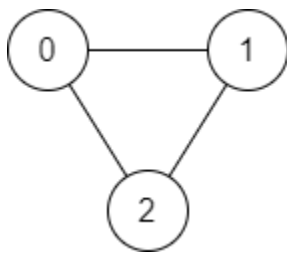
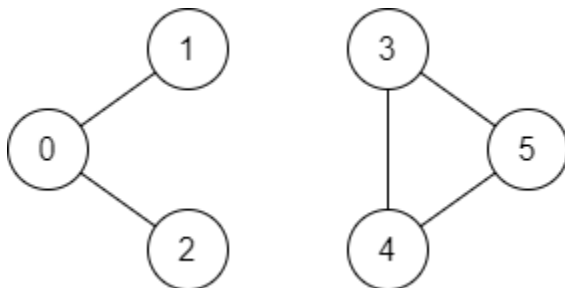**Input**: n = 3, edges = [[0,1],[1,2],[2,0]], source = 0, destination = 2
Output: true
Explanation: There are two paths from vertex 0 to vertex 2:
- 0 → 1 → 2
- 0 → 2

Example 2:

Input: n = 6, edges = [[0,1],[0,2],[3,5],[5,4],[4,3]], source = 0, destination = 5
Output: false
Explanation: There is no path from vertex 0 to vertex 5.

Constraints:

- 1 <= n <= 2 * 1e5
- 0 <= edges.length <= 2 * 1e5

- edges[i].length == 2
- 0 <= ui, vi <= n - 1
- ui != vi
- 0 <= source, destination <= n - 1
- There are no duplicate edges.
- There are no self edges.

Implementation/Code:

```
#include <iostream>
#include <vector>

using namespace std;

void dfs(int node, const vector<vector<int>>& adj, vector<bool>& visited) {
    visited[node] = true;
    for (int neighbor : adj[node]) {
        if (!visited[neighbor]) {
            dfs(neighbor, adj, visited);
        }
    }
}

string validPath(int n, vector<vector<int>>& edges, int source, int destination)
{
    // Create an adjacency list for the graph
    vector<vector<int>> adj(n);
    for (const auto& edge : edges) {
        adj[edge[0]].push_back(edge[1]);
        adj[edge[1]].push_back(edge[0]);
    }

    // Create a visited array to track visited nodes
    vector<bool> visited(n, false);

    // Perform DFS starting from the source node
    dfs(source, adj, visited);
```
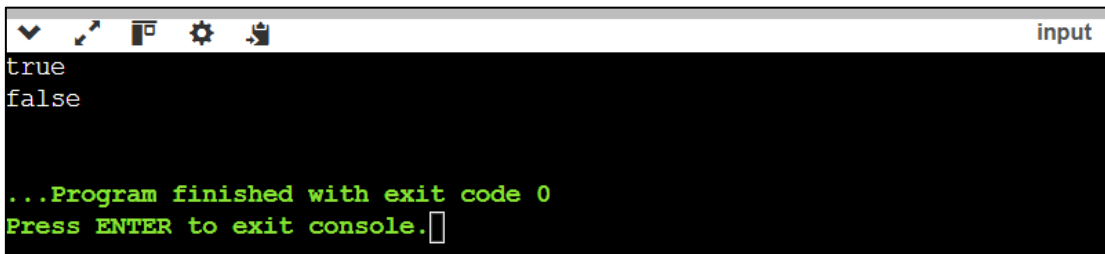
```
        // If the destination node is visited, return "true", otherwise "false"
        return visited[destination] ? "true" : "false";
}

int main()
{
    // Test case 1
    vector<vector<int>> edges1 = {{0,1}, {1,2}, {2,0}};
    cout << validPath(3, edges1, 0, 2) << endl; // Output: true

    // Test case 2
    vector<vector<int>> edges2 = {{0,1}, {0,2}, {3,5}, {5,4}, {4,3}};
    cout << validPath(6, edges2, 0, 5) << endl; // Output: false

    return 0;
}
```

Output:

```
✓  ↗  ▣  ⚙  ⬚                                                    input
true
false


...Program finished with exit code 0
Press ENTER to exit console.▯
```

3. Word Search   *(Medium)*

Given an m x n grid of characters board and a string word, return true if word exists in the grid.

The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

Example 1:

Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"
Output: true

Example 2:

Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "SEE"
Output: true

Example 3:

Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCB"
Output: false

Constraints:

- m == board.length
- n = board[i].length
- 1 <= m, n <= 6
- 1 <= word.length <= 15
- board and word consists of only lowercase and uppercase English letters.

Follow up: Could you use search pruning to make your solution faster with a larger board?

Implementation/Code:

```
#include <iostream>
#include <vector>
#include <functional>

using namespace std;

class Solution {
public:
```

```cpp
bool exist(vector<vector<char>>& board, string word) {
    int m = board.size();
    int n = board[0].size();

    // Function to perform DFS
    function<bool(int, int, int)> dfs = [&](int i, int j, int k) -> bool {
        // If the entire word is found
        if (k == word.size()) return true;

        if (i < 0 || i >= m || j < 0 || j >= n || board[i][j] != word[k]) {
            return false;
        }

        // Temporarily mark the cell as visited
        char temp = board[i][j];
        board[i][j] = '#';

        // Explore all 4 possible directions (up, down, left, right)
        bool found = dfs(i + 1, j, k + 1) || dfs(i - 1, j, k + 1) ||
                     dfs(i, j + 1, k + 1) || dfs(i, j - 1, k + 1);

        // Restore the cell
        board[i][j] = temp;

        return found;
    };

    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            if (board[i][j] == word[0] && dfs(i, j, 0)) {
                return true;
            }
        }
    }

    return false;
}
};
```
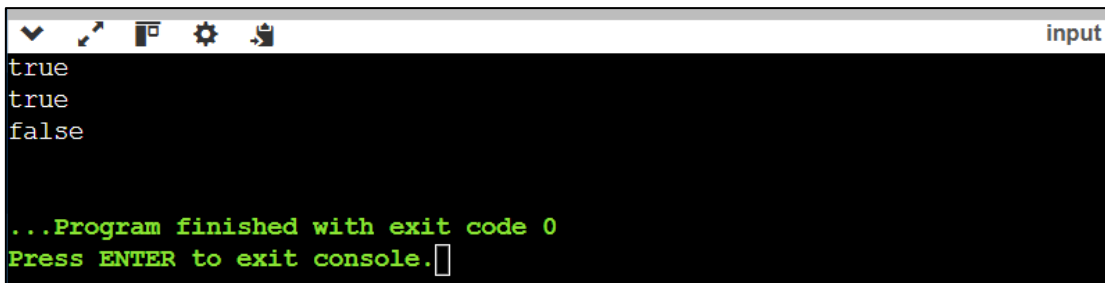
```cpp
int main() {
    Solution solution;

    // Test case 1
    vector<vector<char>> board1 = {
        {'A','B','C','E'},
        {'S','F','C','S'},
        {'A','D','E','E'}
    };
    string word1 = "ABCCED";
    cout << (solution.exist(board1, word1) ? "true" : "false") << endl; // Output: true

    string word2 = "SEE";
    cout << (solution.exist(board1, word2) ? "true" : "false") << endl; // Output: true

    string word3 = "ABCB";
    cout << (solution.exist(board1, word3) ? "true" : "false") << endl; // Output: false

    return 0;
}
```

Output:

4. Rotting Oranges   *(Hard)*

You are given an m x n grid where each cell can have one of three values:

❖ 0 representing an empty cell,

❖ 1 representing a fresh orange, or
❖ 2 representing a rotten orange.

Every minute, any fresh orange that is 4-directionally adjacent to a rotten orange becomes rotten.

Return the minimum number of minutes that must elapse until no cell has a fresh orange. If this is impossible, return -1.

Example 1:
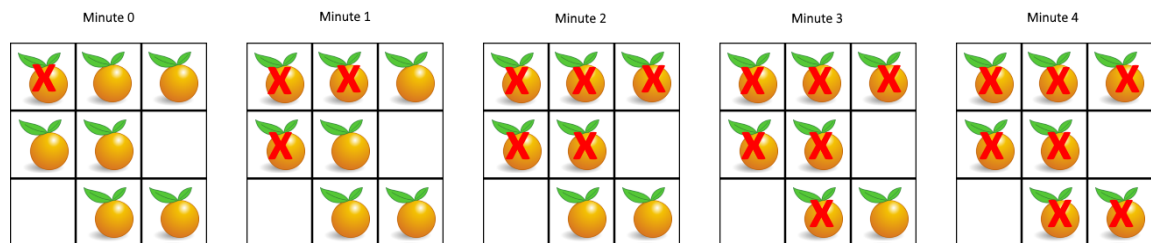
Input: grid = [[2,1,1],[1,1,0],[0,1,1]]
Output: 4

Example 2:

Input: grid = [[2,1,1],[0,1,1],[1,0,1]]
Output: -1
Explanation: The orange in the bottom left corner (row 2, column 0) is never rotten, because rotting only happens 4-directionally.

Example 3:

Input: grid = [[0,2]]
Output: 0
Explanation: Since there are already no fresh oranges at minute 0, the answer is just 0.

Constraints:
● m == grid.length
● n == grid[i].length
● 1 <= m, n <= 10
● grid[i][j] is 0, 1, or 2.

Implementation/Code:

```cpp
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

class Solution {
public:
    int orangesRotting(vector<vector<int>>& grid) {
        int m = grid.size();
        int n = grid[0].size();

        queue<pair<int, int>> q;
        int freshCount = 0;

        // Initialize the queue with all rotten oranges and count fresh oranges
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (grid[i][j] == 2) {
                    q.push({i, j});
                } else if (grid[i][j] == 1) {
                    freshCount++;
                }
            }
        }

        // If there are no fresh oranges, return 0
        if (freshCount == 0) return 0;

        // Directions for 4 possible adjacent cells (up, down, left, right)
        vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
        int minutes = 0;

        // Perform BFS
        while (!q.empty()) {
```

```cpp
            int size = q.size();
            bool rotted = false;

            for (int i = 0; i < size; ++i) {
                auto [x, y] = q.front();
                q.pop();

                for (auto& dir : directions) {
                    int newX = x + dir.first;
                    int newY = y + dir.second;

                    // Check if the new position is within bounds and has a fresh orange
                    if (newX >= 0 && newX < m && newY >= 0 && newY < n &&
grid[newX][newY] == 1) {
                        grid[newX][newY] = 2; // Rotten the fresh orange
                        q.push({newX, newY});
                        freshCount--; // Decrease the fresh orange count
                        rotted = true;
                    }
                }
            }

            // If any fresh orange was rotted in this minute, increase the minute count
            if (rotted) {
                minutes++;
            }
        }

        // If there are still fresh oranges left, return -1
        return freshCount == 0 ? minutes : -1;
    }
};

int main() {
    Solution solution;

    // Test case 1
    vector<vector<int>> grid1 = {{2,1,1},{1,1,0},{0,1,1}};
```
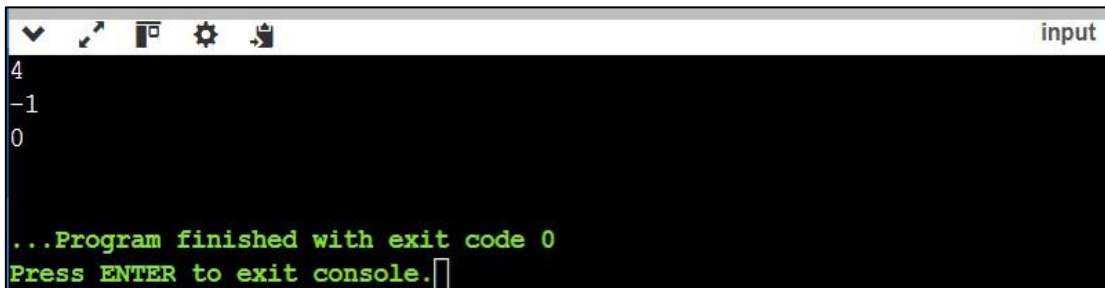
```
cout << solution.orangesRotting(grid1) << endl; // Output: 4

// Test case 2
vector<vector<int>> grid2 = {{2,1,1},{0,1,1},{1,0,1}};
cout << solution.orangesRotting(grid2) << endl; // Output: -1

// Test case 3
vector<vector<int>> grid3 = {{0,2}};
cout << solution.orangesRotting(grid3) << endl; // Output: 0

return 0;
}
```

Output:

```
input
4
-1
0

...Program finished with exit code 0
Press ENTER to exit console.
```

5. Network Delay Time   *(Very Hard)*

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,
- 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.

Example 1:
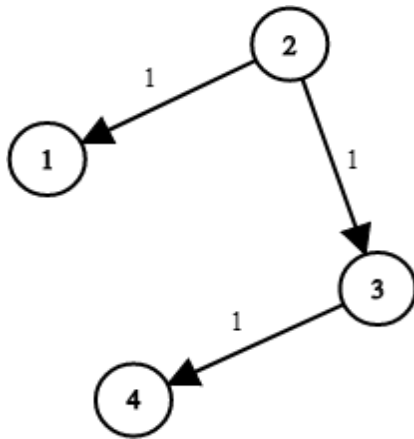
Input: times = [[2,1,1],[2,3,1],[3,4,1]], n = 4, k = 2
Output: 2

Example 2:
Input: times = [[1,2,1]], n = 2, k = 1
Output: 1

Example 3:
Input: times = [[1,2,1]], n = 2, k = 2
Output: -1

Constraints:

- $1 <= k <= n <= 100$
- $1 <= times.length <= 6000$
- times[i].length == 3
- $1 <= ui, vi <= n$
- ui != vi
- $0 <= wi <= 100$
- All the pairs (ui, vi) are unique. (i.e., no multiple edges.)

Implementation/Code:

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
```

```cpp
using namespace std;

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

        // Min-heap to store (time, node) for Dijkstra's algorithm
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
        vector<int> dist(n + 1, INT_MAX); // Distance array initialized to infinity
        dist[k] = 0; // Starting node has distance 0
        pq.push({0, k});

        // Dijkstra's algorithm to find the shortest path from node k
        while (!pq.empty()) {
            auto [currentTime, node] = pq.top();
            pq.pop();

            if (currentTime > dist[node]) continue;

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

        int result = 0;
        for (int i = 1; i <= n; ++i) {
            if (dist[i] == INT_MAX) {
                return -1; // If any node is unreachable, return -1
```

```cpp
            }
            result = max(result, dist[i]);
        }

        return result;
    }
};

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; // Output: 2

    vector<vector<int>> times2 = {{1,2,1}};
    int n2 = 2, k2 = 1;
    cout << solution.networkDelayTime(times2, n2, k2) << endl; // Output: 1

    vector<vector<int>> times3 = {{1,2,1}};
    int n3 = 2, k3 = 2;
    cout << solution.networkDelayTime(times3, n3, k3) << endl; // Output: -1

    return 0;
}
```
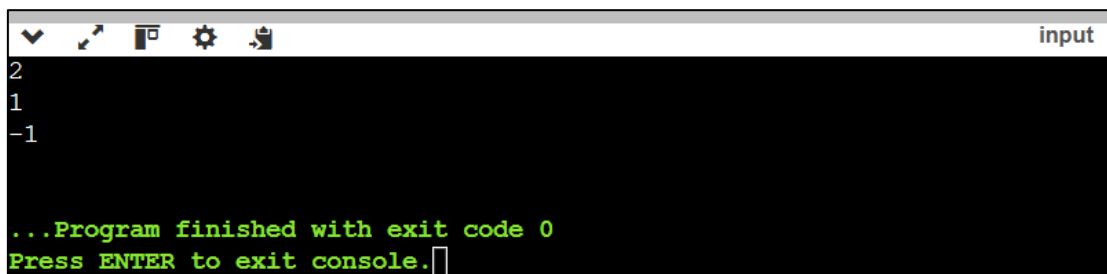
Output: