UNIVERSITY INSTITUTE OF ENGINEERING

Department of Computer Science & Engineering

(BE-CSE)

WINTER DOMAIN CAMP

Date : 28/12/2024

Submitted to:

Faculty name: Er.Rajni Devi

Submitted by:

Name: Akshi Datta

UID: 22BCS15369

Section: IOT-620

DAY 7(28/12/24)

PROBLEM 1

Objective: Find Center of Star Graph

Code:

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

int findCenter(vector<vector<int>>& edges) {
    if (edges[0][0] == edges[1][0] || edges[0][0] == edges[1][1])
        return edges[0][0];
    return edges[0][1];
}

int main() {
    vector<vector<int>> edges = {{1, 2}, {2, 3}, {4, 2}};
    cout << findCenter(edges) << endl; // Output: 2
    return 0;
}
```

Output:

PROBLEM 2

Objective: Find the Town Judge

Code:

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

int findJudge(int n, vector<vector<int>>& trust) {
    vector<int> trustCount(n + 1, 0);

    for (auto& t : trust) {
```

```
        trustCount[t[0]]--; // Outgoing trust
        trustCount[t[1]]++; // Incoming trust
    }

    for (int i = 1; i <= n; ++i) {
        if (trustCount[i] == n - 1)
            return i;
    }
    return -1;
}

int main() {
    int n = 3;
    vector<vector<int>> trust = {{1, 3}, {2, 3}, {3, 1}};
    cout << findJudge(n, trust) << endl; // Output: -1
    return 0;
}
```

Output:

PROBLEM 3

Objective: Flood Fill – link

Code:

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

void dfs(vector<vector<int>>& image, int sr, int sc, int newColor, int oldColor) {
    if (sr < 0 || sr >= image.size() || sc < 0 || sc >= image[0].size() || image[sr][sc] != oldColor)
        return;

    image[sr][sc] = newColor;

    dfs(image, sr + 1, sc, newColor, oldColor);
    dfs(image, sr - 1, sc, newColor, oldColor);
    dfs(image, sr, sc + 1, newColor, oldColor);
    dfs(image, sr, sc - 1, newColor, oldColor);
}
```

```
vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int color) {
    int oldColor = image[sr][sc];
    if (oldColor != color) {
        dfs(image, sr, sc, color, oldColor);
    }
    return image;
}

int main() {
    vector<vector<int>> image = {{1, 1, 1}, {1, 1, 0}, {1, 0, 1}};
    int sr = 1, sc = 1, color = 2;
    vector<vector<int>> result = floodFill(image, sr, sc, color);

    for (const auto& row : result) {
        for (int pixel : row) {
            cout << pixel << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Output:

```
Output

2 2 2
2 2 0
2 0 1
```

PROBLEM 4

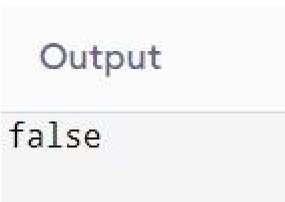
Objective: Find if Path Exists in Graph

Code:

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

```
bool validPath(int n, vector<vector<int>>& edges, int source, int destination) {
    unordered_map<int, vector<int>> graph;
    for (const auto& edge : edges) {
        graph[edge[0]].push_back(edge[1]);
        graph[edge[1]].push_back(edge[0]);
    }

    queue<int> q;
    unordered_set<int> visited;
    q.push(source);

    while (!q.empty()) {
        int current = q.front();
        q.pop();
        if (current == destination) return true;

        if (visited.find(current) == visited.end()) {
            visited.insert(current);
            for (const int& neighbor : graph[current]) {
                if (visited.find(neighbor) == visited.end()) {
                    q.push(neighbor);
                }
            }
        }
    }

    return false;
}

int main() {
    int n = 6;
    vector<vector<int>> edges = {{0, 1}, {0, 2}, {3, 5}, {5, 4}, {4, 3}};
    int source = 0, destination = 5;

    cout << (validPath(n, edges, source, destination) ? "true" : "false") << endl;
    return 0;
}
```

Output:



PROBLEM 5

Objective: BFS of graph link

Code:

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

vector<int> bfsOfGraph(int n, vector<vector<int>>& adj) {
    vector<int> bfsTraversal;
    vector<bool> visited(n, false);

    queue<int> q;
    q.push(0); // Start BFS from vertex 0
    visited[0] = true;

    while (!q.empty()) {
        int node = q.front();
        q.pop();
        bfsTraversal.push_back(node);

        for (const int& neighbor : adj[node]) {
            if (!visited[neighbor]) {
                q.push(neighbor);
                visited[neighbor] = true;
            }
        }
    }

    return bfsTraversal;
}

int main() {
    vector<vector<int>> adj = {{2, 3, 1}, {0}, {0, 4}, {0}, {2}};
    vector<int> bfsResult = bfsOfGraph(5, adj);

    for (int node : bfsResult) {
        cout << node << " ";
    }
    cout << endl;

    return 0;
}
```

Output:

```
Output
0 2 3 1 4
```

PROBLEM 6

Objective: DFS of Graph

Code:

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

void dfsUtil(int node, vector<bool>& visited, vector<vector<int>>& adj, vector<int>& dfsTraversal) {
    visited[node] = true;
    dfsTraversal.push_back(node);

    for (const int& neighbor : adj[node]) {
        if (!visited[neighbor]) {
            dfsUtil(neighbor, visited, adj, dfsTraversal);
        }
    }
}

vector<int> dfsOfGraph(int n, vector<vector<int>>& adj) {
    vector<int> dfsTraversal;
    vector<bool> visited(n, false);

    dfsUtil(0, visited, adj, dfsTraversal); // Start DFS from vertex 0

    return dfsTraversal;
}

int main() {
    vector<vector<int>> adj = {{2, 3, 1}, {0}, {0, 4}, {0}, {2}};
    vector<int> dfsResult = dfsOfGraph(5, adj);

    for (int node : dfsResult) {
        cout << node << " ";
    }
```

```
    cout << endl;

    return 0;
}
```

Output:

```
Output

0 2 4 3 1
```

PROBLEM 7

Objective: 01 Matrix

Code:

```
#include <iostream> // For cout
#include <vector>
#include <queue>
#include <climits> // For INT_MAX
using namespace std;

vector<vector<int>> updateMatrix(vector<vector<int>>& mat) {
    int m = mat.size(), n = mat[0].size();
    vector<vector<int>> dist(m, vector<int>(n, INT_MAX));
    queue<pair<int, int>> q;

    // Add all 0s to the queue and initialize distances
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            if (mat[i][j] == 0) {
                dist[i][j] = 0;
                q.push({i, j});
            }
        }
    }

    // Directions for BFS
    vector<pair<int, int>> directions = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

    while (!q.empty()) {
        auto [x, y] = q.front();
        q.pop();
```

```
        for (auto [dx, dy] : directions) {
            int nx = x + dx, ny = y + dy;
            if (nx >= 0 && ny >= 0 && nx < m && ny < n && dist[nx][ny] > dist[x][y] + 1) {
                dist[nx][ny] = dist[x][y] + 1;
                q.push({nx, ny});
            }
        }
    }

    return dist;
}

int main() {
    vector<vector<int>> mat = {
        {0, 0, 0},
        {0, 1, 0},
        {1, 1, 1}
    };

    vector<vector<int>> result = updateMatrix(mat);

    // Print the result
    for (const auto& row : result) {
        for (int val : row) {
            cout << val << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Output:

```
Output

0 0 0
0 1 0
1 2 1
```

PROBLEM 8

Objective: Course Schedule II

Code:

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

vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {
    if (numCourses == 0) return {}; // Edge case: no courses

    vector<vector<int>> graph(numCourses);
    vector<int> inDegree(numCourses, 0);
    vector<int> order;

    // Build graph and calculate in-degrees
    for (auto& pre : prerequisites) {
        graph[pre[1]].push_back(pre[0]);
        ++inDegree[pre[0]];
    }

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

    // Perform BFS for topological sort
    while (!q.empty()) {
        int course = q.front();
        q.pop();
        order.push_back(course);

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

    // If not all courses are processed, return empty due to cycle
    if (order.size() != numCourses) return {};
    return order;
}

int main() {
    int numCourses = 4;
    vector<vector<int>> prerequisites = {{1, 0}, {2, 0}, {3, 1}, {3, 2}};
```

```
    vector<int> result = findOrder(numCourses, prerequisites);

    if (result.empty()) {
        cout << "Impossible to complete all courses (cycle detected)." << endl;
    } else {
        cout << "Course order: ";
        for (int course : result) {
            cout << course << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Output:

```
Output

Course order: 0 1 2 3
```

PROBLEM 9

Objective: Word Search

Code:

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

bool dfs(vector<vector<char>>& board, string& word, int i, int j, int index) {
    if (index == word.size()) return true;
    if (i < 0 || j < 0 || i >= board.size() || j >= board[0].size() || board[i][j] != word[index])
        return false;

    char temp = board[i][j];
    board[i][j] = '#'; // Mark as visited

    bool found = dfs(board, word, i + 1, j, index + 1) ||
                 dfs(board, word, i - 1, j, index + 1) ||
                 dfs(board, word, i, j + 1, index + 1) ||
                 dfs(board, word, i, j - 1, index + 1);
```

```
    board[i][j] = temp; // Restore cell
    return found;
}

bool exist(vector<vector<char>>& board, string word) {
    if (board.empty() || word.empty()) return false; // Handle edge cases
    for (int i = 0; i < board.size(); ++i) {
        for (int j = 0; j < board[0].size(); ++j) {
            if (dfs(board, word, i, j, 0)) return true;
        }
    }
    return false;
}

int main() {
    vector<vector<char>> board = {
        {'A', 'B', 'C', 'E'},
        {'S', 'F', 'C', 'S'},
        {'A', 'D', 'E', 'E'}
    };
    string word = "ABCCED";

    if (exist(board, word)) {
        cout << "Word exists in the board." << endl;
    } else {
        cout << "Word does not exist in the board." << endl;
    }

    return 0;
}
```

Output:

Output

```
Word exists in the board.
```

PROBLEM 10

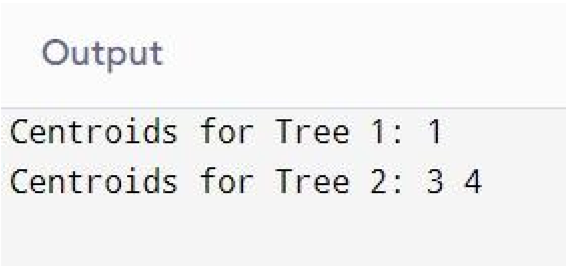
Objective: Minimum Height Trees

Code:

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

vector<int> findMinHeightTrees(int n, vector<vector<int>>& edges) {
    if (n == 0) return {}; // Edge case: No nodes
    if (n == 1) return {0}; // Edge case: Single node

    // Build the graph
    vector<unordered_set<int>> graph(n);
    for (auto& edge : edges) {
        graph[edge[0]].insert(edge[1]);
        graph[edge[1]].insert(edge[0]);
    }

    // Identify initial leaves
    vector<int> leaves;
    for (int i = 0; i < n; ++i) {
        if (graph[i].size() == 1) leaves.push_back(i);
    }

    // Remove leaves layer by layer until <=2 nodes remain
    while (n > 2) {
        n -= leaves.size();
        vector<int> newLeaves;

        for (int leaf : leaves) {
            int neighbor = *graph[leaf].begin(); // Get the only neighbor
            graph[neighbor].erase(leaf); // Remove leaf from neighbor's list
            if (graph[neighbor].size() == 1) newLeaves.push_back(neighbor); // New leaf
        }

        leaves = newLeaves;
    }

    return leaves; // Remaining nodes are the centroids
}

int main() {
    // Test cases
    vector<vector<int>> edges1 = {{1, 0}, {1, 2}, {1, 3}};
    int n1 = 4;
```

```
    vector<int> result1 = findMinHeightTrees(n1, edges1);
    cout << "Centroids for Tree 1: ";
    for (int node : result1) cout << node << " ";
    cout << endl;

    vector<vector<int>> edges2 = {{0, 3}, {1, 3}, {2, 3}, {4, 3}, {5, 4}};
    int n2 = 6;
    vector<int> result2 = findMinHeightTrees(n2, edges2);
    cout << "Centroids for Tree 2: ";
    for (int node : result2) cout << node << " ";
    cout << endl;

    return 0;
}
```

Output:



PROBLEM 11

Objective: Accounts Merge

Code:

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

class Solution {
public:
    vector<vector<string>> accountsMerge(vector<vector<string>>& accounts) {
        unordered_map<string, string> parent;
        unordered_map<string, string> owner;
        unordered_map<string, set<string>> unions;

        // Initialize parent and owner mappings
        for (auto& acc : accounts) {
            for (int i = 1; i < acc.size(); i++) {
```

```
            if (!parent.count(acc[i])) { // Avoid overwriting parent
                parent[acc[i]] = acc[i];
                owner[acc[i]] = acc[0];
            }
        }
    }

    // Find function with path compression
    function<string(string)> find = [&](string email) {
        if (parent[email] != email) {
            parent[email] = find(parent[email]);
        }
        return parent[email];
    };

    // Union emails within the same account
    for (auto& acc : accounts) {
        string p = find(acc[1]);
        for (int i = 2; i < acc.size(); i++) {
            parent[find(acc[i])] = p;
        }
    }

    // Group emails by root parent
    for (auto& acc : accounts) {
        for (int i = 1; i < acc.size(); i++) {
            unions[find(acc[i])].insert(acc[i]);
        }
    }

    // Construct the result
    vector<vector<string>> result;
    for (auto& [email, emails] : unions) {
        vector<string> merged(emails.begin(), emails.end());
        merged.insert(merged.begin(), owner[email]);
        result.push_back(merged);
    }
    return result;
  }
};

int main() {
    Solution sol;
    vector<vector<string>> accounts = {
        {"John", "john@example.com", "john_new@example.com"},
        {"John", "john@example.com", "john@yahoo.com"},
        {"Mary", "mary@example.com"},
        {"John", "john_new@example.com", "john@newdomain.com"}
    };

    vector<vector<string>> result = sol.accountsMerge(accounts);
```

```
    for (const auto& account : result) {
        for (const auto& email : account) {
            cout << email << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Output:

```
Output                                                    Clear
Mary mary@example.com
John john@example.com john@newdomain.com john@yahoo.com john_new@example.com
```

PROBLEM 12

Objective: Rotting Oranges

Code:

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

class Solution {
public:
    int orangesRotting(vector<vector<int>>& grid) {
        int m = grid.size(), n = grid[0].size();
        queue<pair<int, int>> q;
        int fresh = 0, time = 0;

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == 2) q.push({i, j});
                if (grid[i][j] == 1) fresh++;
            }
        }

        vector<int> dirs = {0, 1, 0, -1, 0};
        while (!q.empty() && fresh) {
```

```
            int size = q.size();
            for (int i = 0; i < size; i++) {
                auto [x, y] = q.front(); q.pop();
                for (int d = 0; d < 4; d++) {
                    int nx = x + dirs[d], ny = y + dirs[d + 1];
                    if (nx < 0 || ny < 0 || nx >= m || ny >= n || grid[nx][ny] != 1) continue;
                    grid[nx][ny] = 2;
                    q.push({nx, ny});
                    fresh--;
                }
            }
            time++;
        }
        return fresh == 0 ? time : -1;
    }
};

int main() {
    Solution sol;
    vector<vector<int>> grid = {{2,1,1},{1,1,0},{0,1,1}};  // Example grid
    int result = sol.orangesRotting(grid);
    cout << "Minimum time to rot all oranges: " << result << endl;
    return 0;
}
```

Output:

```
Output
Minimum time to rot all oranges: 4
```

PROBLEM 13

Objective: Pacific Atlantic Water Flow

Code:

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

class Solution {
public:
    vector<vector<int>> pacificAtlantic(vector<vector<int>>& heights) {
```

```
        int m = heights.size(), n = heights[0].size();
        vector<vector<bool>> pacific(m, vector<bool>(n, false));
        vector<vector<bool>> atlantic(m, vector<bool>(n, false));
        vector<vector<int>> result;

        function<void(int, int, vector<vector<bool>>&)> dfs = [&](int i, int j,
vector<vector<bool>>& ocean) {
            ocean[i][j] = true;
            vector<int> dirs = {0, 1, 0, -1, 0};
            for (int d = 0; d < 4; d++) {
                int ni = i + dirs[d], nj = j + dirs[d + 1];
                if (ni < 0 || nj < 0 || ni >= m || nj >= n || ocean[ni][nj] || heights[ni][nj] < heights[i][j])
continue;
                dfs(ni, nj, ocean);
            }
        };

        for (int i = 0; i < m; i++) {
            dfs(i, 0, pacific);
            dfs(i, n - 1, atlantic);
        }
        for (int j = 0; j < n; j++) {
            dfs(0, j, pacific);
            dfs(m - 1, j, atlantic);
        }

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (pacific[i][j] && atlantic[i][j]) result.push_back({i, j});
            }
        }
        return result;
    }
};

int main() {
    Solution sol;
    vector<vector<int>> heights = {
        {1, 2, 2, 3, 5},
        {3, 2, 3, 4, 4},
        {2, 4, 5, 3, 1},
        {6, 7, 1, 4, 2},
        {5, 1, 1, 2, 4}
    };
    vector<vector<int>> result = sol.pacificAtlantic(heights);
    for (const auto& cell : result) {
        cout << "[" << cell[0] << ", " << cell[1] << "] ";
    }
    cout << endl;
    return 0;
}
```

Output:

```
Output
[0, 4] [1, 3] [1, 4] [2, 2] [3, 0] [3, 1] [4, 0]
```

PROBLEM 14

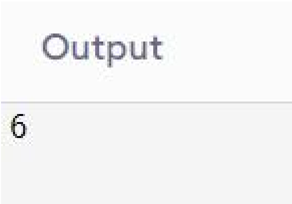
Objective: Max Area of Island

Code:

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

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

        function<int(int, int)> dfs = [&](int i, int j) {
            if (i < 0 || j < 0 || i >= m || j >= n || grid[i][j] == 0) return 0;
            grid[i][j] = 0;
            return 1 + dfs(i + 1, j) + dfs(i - 1, j) + dfs(i, j + 1) + dfs(i, j - 1);
        };

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == 1) maxArea = max(maxArea, dfs(i, j));
            }
        }
        return maxArea;
    }
};

int main() {
    Solution sol;
    vector<vector<int>> grid = {
        {0, 1, 0, 0, 0},
        {1, 1, 1, 0, 0},
        {0, 1, 0, 0, 1},
        {0, 1, 0, 1, 1}
```

```
    };
    cout << sol.maxAreaOfIsland(grid) << endl;
    return 0;
}
```

Output:



PROBLEM 15

Objective: Evaluate Division

Code:

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

class Solution {
public:
    vector<double> calcEquation(vector<vector<string>>& equations, vector<double>& values, vector<vector<string>>& queries) {
        unordered_map<string, unordered_map<string, double>> graph;

        for (int i = 0; i < equations.size(); i++) {
            graph[equations[i][0]][equations[i][1]] = values[i];
            graph[equations[i][1]][equations[i][0]] = 1.0 / values[i];
        }

        function<double(string, string, unordered_set<string>&)> dfs = [&](string start, string end, unordered_set<string>& visited) {
            if (graph.find(start) == graph.end()) return -1.0;
            if (graph[start].find(end) != graph[start].end()) return graph[start][end];
            visited.insert(start);

            for (auto& [neighbor, value] : graph[start]) {
                if (visited.count(neighbor)) continue;
                double res = dfs(neighbor, end, visited);
                if (res != -1.0) return value * res;
            }
            return -1.0;
```

```
        };

        vector<double> result;
        for (auto& query : queries) {
            unordered_set<string> visited;
            result.push_back(dfs(query[0], query[1], visited));
        }
        return result;
    }
};

int main() {
    Solution sol;
    vector<vector<string>> equations = {{"a", "b"}, {"b", "c"}};
    vector<double> values = {2.0, 3.0};
    vector<vector<string>> queries = {{"a", "c"}, {"b", "a"}, {"a", "e"}, {"a", "a"}, {"x", "x"}};

    vector<double> result = sol.calcEquation(equations, values, queries);
    for (double res : result) {
        cout << res << " ";
    }
    cout << endl;

    return 0;
}
```

Output:

Output

```
6 0.5 -1 1 -1
```

PROBLEM 16

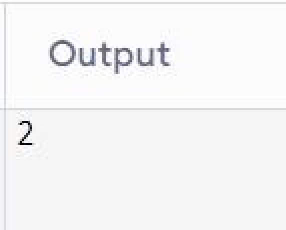
Objective: Network Delay Time

Code:

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

```
using namespace std;

int networkDelayTime(vector<vector<int>>& times, int n, int k) {
    // Create an adjacency list for the graph
    vector<vector<pair<int, int>>> graph(n + 1);
    for (auto& t : times) {
        graph[t[0]].emplace_back(t[1], t[2]); // t[0] -> t[1] with weight t[2]
    }

    // Initialize the distance array with maximum possible values
    vector<int> dist(n + 1, INT_MAX);
    dist[k] = 0;  // Start from node k, so distance to k is 0

    // Min-heap priority queue to always process the node with the smallest distance
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
    pq.emplace(0, k);  // Start from node k with time 0

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

        // If this node has already been processed with a shorter time, skip it
        if (time > dist[node]) continue;

        // Update the distances for each neighbor
        for (auto& [next, weight] : graph[node]) {
            if (dist[next] > time + weight) {
                dist[next] = time + weight;
                pq.emplace(dist[next], next);
            }
        }
    }

    // Find the maximum time to reach all nodes
    int maxTime = *max_element(dist.begin() + 1, dist.end()); // Skip dist[0] as nodes are 1-indexed

    // If any node is unreachable, return -1
    return maxTime == INT_MAX ? -1 : maxTime;
}

int main() {
    vector<vector<int>> times = {{2, 1, 1}, {2, 3, 1}, {3, 4, 1}};
    int n = 4, k = 2;
    cout << networkDelayTime(times, n, k) << endl; // Output: 2
    return 0;
}
```

Output:



PROBLEM 17

Objective: All Paths From Source to Target

Code:

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

void dfs(vector<vector<int>>& graph, int node, vector<int>& path, vector<vector<int>>& result)
{
    path.push_back(node);
    if (node == graph.size() - 1) {
        result.push_back(path);
    } else {
        for (int next : graph[node]) {
            dfs(graph, next, path, result);
        }
    }
    path.pop_back();
}

vector<vector<int>> allPathsSourceTarget(vector<vector<int>>& graph) {
    vector<vector<int>> result;
    vector<int> path;
    dfs(graph, 0, path, result);
    return result;
}

int main() {
    vector<vector<int>> graph = {{1, 2}, {3}, {3}, {}};
    vector<vector<int>> paths = allPathsSourceTarget(graph);

    for (auto& path : paths) {
        for (int node : path) {
```

```
            cout << node << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Output:

```
Output
0 1 3
0 2 3
```

PROBLEM 18

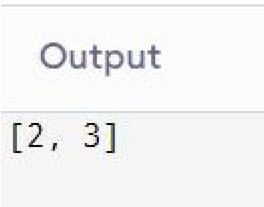
Objective: Redundant Connection

Code:

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

class UnionFind {
    vector<int> parent, rank;

public:
    UnionFind(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; ++i) parent[i] = i;
    }

    int find(int x) {
        if (x != parent[x]) parent[x] = find(parent[x]);
        return parent[x];
    }

    bool unite(int x, int y) {
        int px = find(x), py = find(y);
        if (px == py) return false;
        if (rank[px] < rank[py]) {
            parent[px] = py;
```

```
        } else if (rank[px] > rank[py]) {
            parent[py] = px;
        } else {
            parent[py] = px;
            rank[px]++;
        }
        return true;
    }
};

vector<int> findRedundantConnection(vector<vector<int>>& edges) {
    int n = edges.size();
    UnionFind uf(n + 1);

    for (auto& edge : edges) {
        if (!uf.unite(edge[0], edge[1])) return edge;
    }
    return {};
}

int main() {
    vector<vector<int>> edges = {{1, 2}, {1, 3}, {2, 3}};
    vector<int> result = findRedundantConnection(edges);
    cout << "[" << result[0] << ", " << result[1] << "]" << endl; // Output: [2,3]
    return 0;
}
```

Output:


Output

[2, 3]

PROBLEM 19

Objective: Shortest Path in Binary Matrix

Code:

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

```
int shortestPathBinaryMatrix(vector<vector<int>>& grid) {
    int n = grid.size();
    if (grid[0][0] != 0 || grid[n - 1][n - 1] != 0) return -1;

    vector<vector<int>> directions = {{-1, -1}, {-1, 0}, {-1, 1}, {0, -1}, {0, 1}, {1, -1}, {1, 0}, {1, 1}};
    queue<pair<int, int>> q;
    q.push({0, 0});
    grid[0][0] = 1;

    while (!q.empty()) {
        auto [x, y] = q.front();
        q.pop();

        if (x == n - 1 && y == n - 1) return grid[x][y];

        for (auto& d : directions) {
            int nx = x + d[0], ny = y + d[1];
            if (nx >= 0 && ny >= 0 && nx < n && ny < n && grid[nx][ny] == 0) {
                grid[nx][ny] = grid[x][y] + 1;
                q.push({nx, ny});
            }
        }
    }
    return -1;
}

int main() {
    vector<vector<int>> grid = {{0, 1}, {1, 0}};
    cout << shortestPathBinaryMatrix(grid) << endl; // Output: 2
    return 0;
}
```

Output:

PROBLEM 20

Objective: Remove Max Number of Edges to Keep Graph Fully Traversable

Code:

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

class UnionFind {
    vector<int> parent, rank;

public:
    UnionFind(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; ++i) parent[i] = i;
    }

    int find(int x) {
        if (x != parent[x]) parent[x] = find(parent[x]);
        return parent[x];
    }

    bool unite(int x, int y) {
        int px = find(x), py = find(y);
        if (px == py) return false;
        if (rank[px] < rank[py]) {
            parent[px] = py;
        } else if (rank[px] > rank[py]) {
            parent[py] = px;
        } else {
            parent[py] = px;
            rank[px]++;
        }
        return true;
    }
};

int maxNumEdgesToRemove(int n, vector<vector<int>>& edges) {
    UnionFind alice(n + 1), bob(n + 1);
    int sharedEdges = 0;

    for (auto& edge : edges) {
        if (edge[0] == 3) {
            if (alice.unite(edge[1], edge[2]) | bob.unite(edge[1], edge[2])) {
                sharedEdges++;
            }
        }
```

```
    }

    int aliceEdges = sharedEdges, bobEdges = sharedEdges;
    for (auto& edge : edges) {
        if (edge[0] == 1) aliceEdges += alice.unite(edge[1], edge[2]);
        if (edge[0] == 2) bobEdges += bob.unite(edge[1], edge[2]);
    }

    if (aliceEdges != n - 1 || bobEdges != n - 1) return -1;
    return edges.size() - aliceEdges - bobEdges + sharedEdges;
}

int main() {
    vector<vector<int>> edges = {{3, 1, 2}, {3, 2, 3}, {1, 1, 3}, {1, 2, 4}, {1, 1, 2}, {2, 3, 4}};
    cout << maxNumEdgesToRemove(4, edges) << endl; // Output: 2
    return 0;
}
```

Output: