Name : Rudraksh Mishra
UID   : 22BCS10607
Class : KPIT - 901 / A

Q1. 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.

Q2. 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 edge, 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.

Q3. Given an m x n binary matrix mat, return the distance of the nearest 0 for each cell. The distance between two adjacent cells is 1.

Q4. Given a list of accounts where each element accounts[i] is a list of strings, where the first element accounts[i][0] is a name, and the rest of the elements are emails representing emails of the account. Now, we would like to merge these accounts. Two accounts definitely belong to the same person if there is some common email to both accounts. Note that even if two accounts have the same name, they may belong to different people as people could have the same name. A person can have any number of accounts initially, but all of their accounts definitely have the same name. After merging the accounts, return the accounts in the following format: the first element of each account is the name, and the rest of the elements are emails in sorted order. The accounts themselves can be returned in any order.

Q5. 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.

Solutions :

A1. Find Center of Star Graph

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

class Solution {
public:
    int find_center(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() {
    Solution solution;
    vector<vector<int>> edges1 = {{1, 2}, {2, 3}, {4, 2}};
    vector<vector<int>> edges2 = {{1, 3}, {3, 2}, {3, 4}};

    int result1 = solution.find_center(edges1);
    int result2 = solution.find_center(edges2);

    cout << "Result 1 : " << result1 << endl;
    cout << "Result 2 : " << result2 << endl;

    return 0;
}
```

Output :

```
Result 1 : 2
Result 2 : 3
```

A2. Find if Path Exists in Graph

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

class Solution {
public:
    bool valid_path(int n, vector<vector<int>>& edges, int source, int destination) {
        vector<vector<int>> adjacency_list(n);
        for (const auto& edge : edges) {
            adjacency_list[edge[0]].push_back(edge[1]);
            adjacency_list[edge[1]].push_back(edge[0]);
        }

        vector<bool> visited(n, false);
        queue<int> to_visit;
        to_visit.push(source);
        visited[source] = true;

        while (!to_visit.empty()) {
            int current = to_visit.front();
            to_visit.pop();

            if (current == destination) { return true; }

            for (int neighbor : adjacency_list[current]) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    to_visit.push(neighbor);
                }
            }
        }

        return false;
    }
};
```

```
int main() {
    Solution solution;
    vector<vector<int>> edges1 = {{0, 1}, {1, 2}, {2, 0}};
    vector<vector<int>> edges2 = {{0, 1}, {0, 2}, {3, 5}, {5, 4}, {4, 3}};

    bool result1 = solution.valid_path(3, edges1, 0, 2);
    bool result2 = solution.valid_path(6, edges2, 0, 5);

    cout << "Result 1 : " << (result1 ? "true" : "false") << endl;
    cout << "Result 2 : " << (result2 ? "true" : "false") << endl;

    return 0;
}
```

Output :

```
Result 1 : true
Result 2 : false
```

A3. 01 Matrix

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

class Solution {
public:
    vector<vector<int>> update_matrix(vector<vector<int>>& mat) {
        int rows = mat.size();
        int cols = mat[0].size();
        vector<vector<int>> distances(rows, vector<int>(cols, INT_MAX));
        queue<pair<int, int>> to_visit;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (mat[i][j] == 0) {
                    distances[i][j] = 0;
                    to_visit.push({i, j});
                }
            }
        }

        vector<pair<int, int>> directions = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

        while (!to_visit.empty()) {
            auto [current_row, current_col] = to_visit.front();
            to_visit.pop();
            for (const auto& [d_row, d_col] : directions) {
                int new_row = current_row + d_row;
                int new_col = current_col + d_col;
                if (new_row >= 0 && new_row < rows && new_col >= 0 && new_col < cols) {
                    if (
                    distances[new_row][new_col] > distances[current_row][current_col] + 1
                    ) {
                    distances[new_row][new_col] = distances[current_row][current_col] + 1;
                    to_visit.push({new_row, new_col});
                    }
```

```
                }
            }
        }
        return distances;
    }
};

int main() {
    Solution solution;
    vector<vector<int>> mat = {{0, 0, 0}, {0, 1, 0}, {1, 1, 1}};
    vector<vector<int>> result = solution.update_matrix(mat);

    for (const auto& row : result) {
        for (int distance : row) {
            cout << distance << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Output :

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

A4. Accounts Merge

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

class Solution {
public:
    vector<vector<string>> accounts_merge(vector<vector<string>>& accounts) {
        unordered_map<string, string> email_to_name;
        unordered_map<string, string> parent;

        for (const auto& account : accounts) {
            string name = account[0];
            for (size_t i = 1; i < account.size(); i++) {
                if (parent.find(account[i]) == parent.end()) {
                    parent[account[i]] = account[i];
                }
                email_to_name[account[i]] = name;
                union_sets(account[1], account[i], parent);
            }
        }

        unordered_map<string, unordered_set<string>> unions;
```

```
        for (const auto& [email, _] : email_to_name) {
            string root = find(email, parent);
            unions[root].insert(email);
        }

        vector<vector<string>> merged_accounts;
        for (const auto& [root, emails] : unions) {
            vector<string> account(emails.begin(), emails.end());
            sort(account.begin(), account.end());
            account.insert(account.begin(), email_to_name[root]);
            merged_accounts.push_back(account);
        }

        return merged_accounts;
    }

private:
    string find(const string& email, unordered_map<string, string>& parent) {
        if (parent[email] != email) {
            parent[email] = find(parent[email], parent);
        }
        return parent[email];
    }

    void union_sets(const string& email1, const string& email2, unordered_map<string,
string>& parent) {
        string root1 = find(email1, parent);
        string root2 = find(email2, parent);
        if (root1 != root2) {
            parent[root1] = root2;
        }
    }
};

int main() {
    Solution solution;
    vector<vector<string>> accounts = {
        {"John", "johnsmith@mail.com", "john00@mail.com"},
        {"John", "johnnybravo@mail.com"},
        {"John", "johnsmith@mail.com", "john_newyork@mail.com"},
        {"Mary", "mary@mail.com"}
    };

    vector<vector<string>> result = solution.accounts_merge(accounts);

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

    return 0;
}
```

Output :

```
John johnnybravo@mail.com
John john00@mail.com john_newyork@mail.com johnsmith@mail.com
Mary mary@mail.com
```

A5. Network Delay Time

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_map>
#include <algorithm>
#include <climits>
using namespace std;

class Solution {
public:
    int network_delay_time(vector<vector<int>>& times, int n, int k) {
        unordered_map<int, vector<pair<int, int>>> graph;
        for (const auto& time : times) { graph[time[0]].emplace_back(time[1], time[2]); }
        vector<int> min_time(n + 1, INT_MAX);
        min_time[k] = 0;
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
        pq.emplace(0, k);

        while (!pq.empty()) {
            auto [current_time, node] = pq.top();
            pq.pop();
            if (current_time > min_time[node]) { continue; }
            for (const auto& [neighbor, weight] : graph[node]) {
                int new_time = current_time + weight;
                if (new_time < min_time[neighbor]) {
                    min_time[neighbor] = new_time;
                    pq.emplace(new_time, neighbor);
                }
            }
        }

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

int main() {
    Solution solution;
    vector<vector<int>> times = {{2, 1, 1}, {2, 3, 1}, {3, 4, 1}};
    int n = 4;
    int k = 2;
    int result = solution.network_delay_time(times, n, k);
    cout  << "Result : " << result << endl;
    return 0;
}
```

Output :

```
Result : 2
```