main.cpp

Share Run

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

int findCenter(vector<vector<int>>& edges) {
    // Check the first two edges to determine the center
    if (edges[0][0] == edges[1][0] || edges[0][0] == edges[1][1]) {
        return edges[0][0];
    }
    return edges[0][1];
}

int main() {
    // Example input
    vector<vector<int>> edges = {{1, 2}, {2, 3}, {4, 2}};

    // Find and print the center of the star graph
    cout << "Center of the star graph: " << findCenter(edges) << endl;

    return 0;
}
```

Output

Clear

```
Center of the star graph: 2


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

main.cpp

Share Run

Output Clear

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

bool dfs(int node, int destination, unordered_map<int, vector
    <int>>& graph, unordered_set<int>& visited) {
    if (node == destination) return true; // Found the path
    visited.insert(node);

    for (int neighbor : graph[node]) {
        if (!visited.count(neighbor)) {
            if (dfs(neighbor, destination, graph, visited)) {
                return true;
            }
        }
    }
    return false;
}

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

    // Build the graph as an adjacency list
    unordered_map<int, vector<int>> graph;
    for (auto& edge : edges) {
        graph[edge[0]].push_back(edge[1]);
        graph[edge[1]].push_back(edge[0]);
```

```
False


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

main.cpp

Share Run

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

vector<vector<int>> updateMatrix(vector<vector<int>>& mat) {
    int m = mat.size();
    int n = mat[0].size();

    vector<vector<int>> result(m, vector<int>(n, INT_MAX));
    queue<pair<int, int>> q;

    // Initialize the queue with all '0' cells
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (mat[i][j] == 0) {
                result[i][j] = 0;
                q.push({i, j});
            }
        }
    }

    // Directions for moving in the grid (up, down, left, right
        )
    vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0,
        -1}, {0, 1}};

    // BFS to update distances
    while (!q.empty()) {
```

Output Clear

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


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

main.cpp

Share Run

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

class UnionFind {
public:
    unordered_map<string, string> parent;

    string find(string email) {
        if (parent[email] != email) {
            parent[email] = find(parent[email]);
        }
        return parent[email];
    }

    void unite(string email1, string email2) {
        string root1 = find(email1);
        string root2 = find(email2);
        if (root1 != root2) {
            parent[root2] = root1;
        }
    }
};

vector<vector<string>> accountsMerge(vector<vector<string>>&
    accounts) {
    UnionFind uf;
```

Output Clear

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


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

main.cpp

Share Run

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

int networkDelayTime(vector<vector<int>>& times, int n, int k)
    {
    // Step 1: Build the graph
    vector<vector<pair<int, int>>> graph(n + 1); // graph[u] =
        {(v, time)}
    for (auto& time : times) {
        int u = time[0], v = time[1], w = time[2];
        graph[u].push_back({v, w});
    }

    // Step 2: Dijkstra's algorithm
    vector<int> dist(n + 1, INT_MAX);  // distance array
    dist[k] = 0;  // distance to the source node is 0

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

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

        // Skip processing if we found a shorter path already
        if (d > dist[node]) continue;
```

Output Clear

```
The minimum time for all nodes to receive the signal: 2


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