

main.cpp



Share

Run

Output

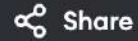
Clear

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int findCenter(vector<vector<int>>& edges) {
6      // Check the first two edges to determine the center
7      if (edges[0][0] == edges[1][0] || edges[0][0] == edges[1][1]
8          ) {
9          return edges[0][0];
10     }
11     return edges[0][1];
12 }
13 int main() {
14     // Example input
15     vector<vector<int>> edges = {{1, 2}, {2, 3}, {4, 2}};
16
17     // Find and print the center of the star graph
18     cout << "Center of the star graph: " << findCenter(edges) <<
19         endl;
20     return 0;
21 }
```

Center of the star graph: 2

=== Code Execution Successful ===

main.cpp



Share

Run

Output

Clear

```
1  #include <iostream>
2  #include <vector>
3  #include <unordered_map>
4  #include <unordered_set>
5  using namespace std;
6
7  bool dfs(int node, int destination, unordered_map<int, vector
    <int>>& graph, unordered_set<int>& visited) {
8      if (node == destination) return true; // Found the path
9      visited.insert(node);
10
11     for (int neighbor : graph[node]) {
12         if (!visited.count(neighbor)) {
13             if (dfs(neighbor, destination, graph, visited)) {
14                 return true;
15             }
16         }
17     }
18     return false;
19 }
20
21 bool validPath(int n, vector<vector<int>>& edges, int source,
    int destination) {
22     if (source == destination) return true;
23
24     // Build the graph as an adjacency list
25     unordered_map<int, vector<int>> graph;
26     for (auto& edge : edges) {
27         graph[edge[0]].push_back(edge[1]);
28         graph[edge[1]].push_back(edge[0]);
```

False

=== Code Execution Successful ===

main.cpp



Share

Run

Output

Clear

```
1  #include <iostream>
2  #include <vector>
3  #include <queue>
4  #include <climits>
5  using namespace std;
6
7  vector<vector<int>> updateMatrix(vector<vector<int>>& mat) {
8      int m = mat.size();
9      int n = mat[0].size();
10
11     vector<vector<int>> result(m, vector<int>(n, INT_MAX));
12     queue<pair<int, int>> q;
13
14     // Initialize the queue with all '0' cells
15     for (int i = 0; i < m; i++) {
16         for (int j = 0; j < n; j++) {
17             if (mat[i][j] == 0) {
18                 result[i][j] = 0;
19                 q.push({i, j});
20             }
21         }
22     }
23
24     // Directions for moving in the grid (up, down, left, right)
25     vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0,
26         -1}, {0, 1}};
27
28     // BFS to update distances
29     while (!q.empty()) {
```

0 0 0

0 1 0

1 2 1

=== Code Execution Successful ===

main.cpp



Share

Run

Output

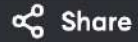
Clear

```
1 #include <iostream>
2 #include <vector>
3 #include <unordered_map>
4 #include <unordered_set>
5 #include <algorithm>
6 using namespace std;
7
8 class UnionFind {
9 public:
10     unordered_map<string, string> parent;
11
12     string find(string email) {
13         if (parent[email] != email) {
14             parent[email] = find(parent[email]);
15         }
16         return parent[email];
17     }
18
19     void unite(string email1, string email2) {
20         string root1 = find(email1);
21         string root2 = find(email2);
22         if (root1 != root2) {
23             parent[root2] = root1;
24         }
25     }
26 };
27
28 vector<vector<string>> accountsMerge(vector<vector<string>>&
    accounts) {
29     UnionFind uf;
```

```
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

Output

Clear

```
1  #include <iostream>
2  #include <vector>
3  #include <queue>
4  #include <climits>
5  using namespace std;
6
7  int networkDelayTime(vector<vector<int>>& times, int n, int k)
8  {
9      // Step 1: Build the graph
10     vector<vector<pair<int, int>>> graph(n + 1); // graph[u] =
11         {(v, time)}
12     for (auto& time : times) {
13         int u = time[0], v = time[1], w = time[2];
14         graph[u].push_back({v, w});
15     }
16
17     // Step 2: Dijkstra's algorithm
18     vector<int> dist(n + 1, INT_MAX); // distance array
19     dist[k] = 0; // distance to the source node is 0
20
21     priority_queue<pair<int, int>, vector<pair<int, int>>,
22         greater<pair<int, int>>> pq;
23     pq.push({0, k}); // {distance, node}
24
25     while (!pq.empty()) {
26         auto [d, node] = pq.top();
27         pq.pop();
28
29         // Skip processing if we found a shorter path already
30         if (d > dist[node]) continue;
```

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

=== Code Execution Successful ===