CU
CHANDIGARH UNIVERSITY

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

DOMAIN WINTER WINNING CAMP ASSIGNMENT

Student Name: Gurnoor Oberoi **UID: 22BCS15716**
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.

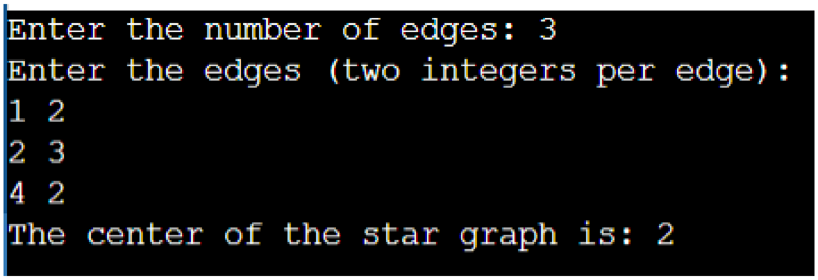
Implementation/Code:

```
#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];
   } else {
      return edges[0][1];
   }
}
int main() {
   int n;
   cout << "Enter the number of edges: ";
   cin >> n;
   vector<vector<int>> edges(n, vector<int>(2));
   cout << "Enter the edges (two integers per edge):" << endl;
   for (int i = 0; i < n; i++) {
      cin >> edges[i][0] >> edges[i][1];
```

```
    }
    int center = findCenter(edges);
    cout << "The center of the star graph is: " << center << endl;

    return 0;
}
```

Output:



```
Enter the number of edges: 3
Enter the edges (two integers per edge):
1 2
2 3
4 2
The center of the star graph is: 2
```

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

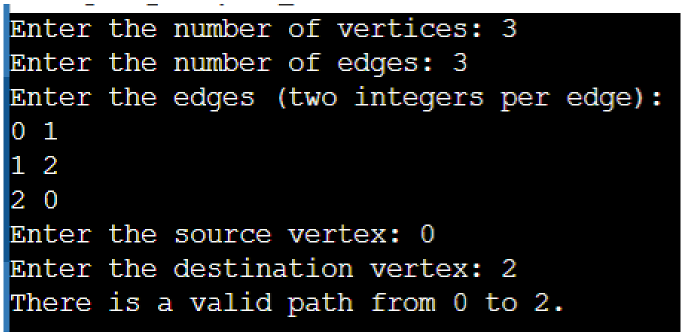
There is a bi-directional graph with n vertices, where each vertex is labeled from 0 to n - 1 (inclusive). The edges in the graph are represented as a 2D integer array edges, where each edges[i] = [ui, vi] denotes a bi-directional edge between vertex ui and vertex vi. Every vertex pair is connected by 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.

Implementation/Code:

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_map>
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]);
    }
    vector<bool> visited(n, false);
    queue<int> q;
    q.push(source);
    visited[source] = true;
    while (!q.empty()) {
        int current = q.front();
        q.pop();
        if (current == destination) {
            return true;
        }
        for (int neighbor : graph[current]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
    return false;
}
int main() {
    int n, m;
    cout << "Enter the number of vertices: ";
    cin >> n;
    cout << "Enter the number of edges: ";
    cin >> m;
    vector<vector<int>> edges(m, vector<int>(2));
    cout << "Enter the edges (two integers per edge):" << endl;
    for (int i = 0; i < m; i++) {
        cin >> edges[i][0] >> edges[i][1];
    }
    int source, destination;
    cout << "Enter the source vertex: ";
    cin >> source;
    cout << "Enter the destination vertex: ";
    cin >> destination;
    if (validPath(n, edges, source, destination)) {
        cout << "There is a valid path from " << source << " to " << destination << "." <<
endl;
    } else {
        cout << "There is no valid path from " << source << " to " << destination << "." <<
endl;
    }
    return 0;
}
```

Output:



```
Enter the number of vertices: 3
Enter the number of edges: 3
Enter the edges (two integers per edge):
0 1
1 2
2 0
Enter the source vertex: 0
Enter the destination vertex: 2
There is a valid path from 0 to 2.
```

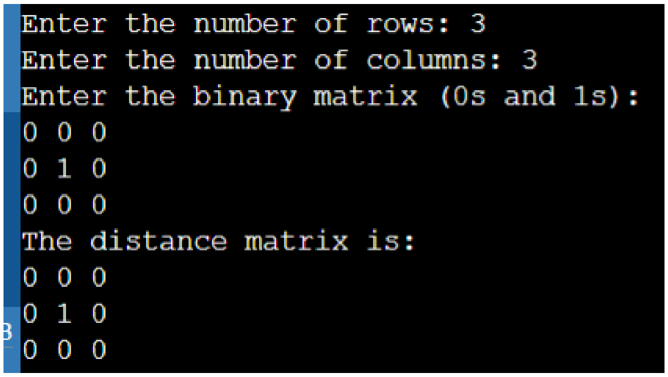
3. 01 Matrix *(Medium)*

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.

Implementation/Code:

```
#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>> dist(m, vector<int>(n, INT_MAX));
    queue<pair<int, int>> q;
    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});
            }
        }
    }
    vector<pair<int, int>> directions = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
    while (!q.empty()) {
        auto [x, y] = q.front();
        q.pop();
        for (const auto& dir : directions) {
            int newX = x + dir.first;
            int newY = y + dir.second;
            if (newX >= 0 && newX < m && newY >= 0 && newY < n) {
                if (dist[newX][newY] > dist[x][y] + 1) {
                    dist[newX][newY] = dist[x][y] + 1;
                    q.push({newX, newY});
                }
            }
        }
    }
    return dist;
}
int main() {
    int m, n;
    cout << "Enter the number of rows: ";
    cin >> m;
    cout << "Enter the number of columns: ";
    cin >> n;
    vector<vector<int>> mat(m, vector<int>(n));
    cout << "Enter the binary matrix (0s and 1s):" << endl;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            cin >> mat[i][j];
        }
    }
    vector<vector<int>> result = updateMatrix(mat);
    cout << "The distance matrix is:" << endl;
    for (const auto& row : result) {
        for (int val : row) {
            cout << val << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Output:



```
Enter the number of rows: 3
Enter the number of columns: 3
Enter the binary matrix (0s and 1s):
0 0 0
0 1 0
0 0 0
The distance matrix is:
0 0 0
0 1 0
0 0 0
```

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.

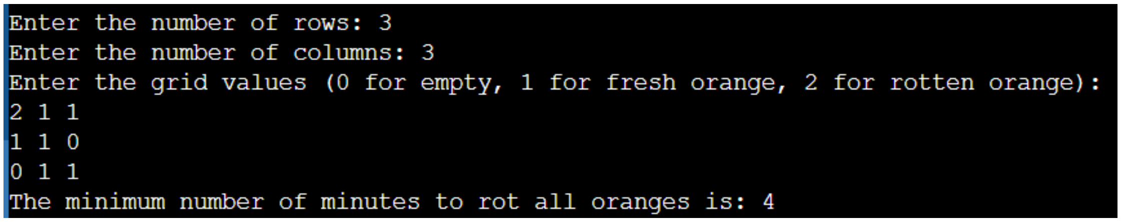
Return the minimum number of minutes that must elapse until no cell has a fresh orange.

If this is impossible, return -1.

Implementation/Code:

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
int orangesRotting(vector<vector<int>>& grid) {
    int m = grid.size();
    int n = grid[0].size();
    queue<pair<int, int>> q;
    int freshOranges = 0;
    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) {
                freshOranges++;
            }
        }
    }
    if (freshOranges == 0) {
        return 0;
    }
    vector<pair<int, int>> directions = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
    int minutes = 0;
    while (!q.empty()) {
        int size = q.size();
        bool rotted = false;
        for (int i = 0; i < size; i++) {
            auto [x, y] = q.front();
            q.pop();
            for (const auto& dir : directions) {
                int newX = x + dir.first;
                int newY = y + dir.second;
                if (newX >= 0 && newX < m && newY >= 0 && newY < n &&
grid[newX][newY] == 1) {
                    grid[newX][newY] = 2;
                    freshOranges--;
                    q.push({newX, newY});
                    rotted = true;
                }
            }
        }
        if (rotted) {
            minutes++;
        }
    }
    return freshOranges == 0 ? minutes : -1;
}
int main() {
    int m, n;
    cout << "Enter the number of rows: ";
    cin >> m;
    cout << "Enter the number of columns: ";
    cin >> n;
    vector<vector<int>> grid(m, vector<int>(n));
    cout << "Enter the grid values (0 for empty, 1 for fresh orange, 2 for rotten orange):"
<< endl;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            cin >> grid[i][j];
        }
    }
    int result = orangesRotting(grid);
    if (result == -1) {
        cout << "It is impossible to rot all fresh oranges." << endl;
    } else {
        cout << "The minimum number of minutes to rot all oranges is: " << result << endl;
    }
    return 0;
}
```

Output:



```
Enter the number of rows: 3
Enter the number of columns: 3
Enter the grid values (0 for empty, 1 for fresh orange, 2 for rotten orange):
2 1 1
1 1 0
0 1 1
The minimum number of minutes to rot all oranges is: 4
```

5. Redundant Connection *(Very Hard)*

In this problem, a tree is an undirected graph that is connected and has no cycles.
You are given a graph that started as a tree with n nodes labeled from 1 to n, with one additional edge added. The added edge has two different vertices chosen from 1 to n, and was not an edge that already existed. The graph is represented as an array edges of length n where edges[i] = [ai, bi] indicates that there is an edge between nodes ai and bi in the graph.
Return an edge that can be removed so that the resulting graph is a tree of n nodes. If there are multiple answers, return the answer that occurs last in the input.

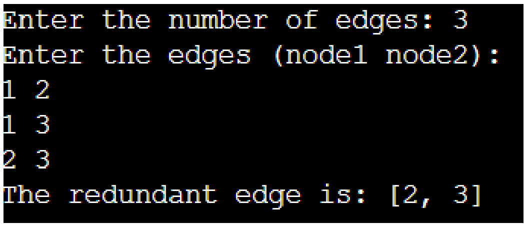
Implementation/Code:

```
#include <iostream>
#include <vector>
using namespace std;
class UnionFind {
private:
    vector<int> parent;
    vector<int> rank;
public:
    UnionFind(int size) {
        parent.resize(size);
        rank.resize(size, 0);
        for (int i = 0; i < size; i++) {
            parent[i] = i;
        }
    }
    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }
    bool unionSets(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX == rootY) {
            return false;
        }
        if (rank[rootX] > rank[rootY]) {
            parent[rootY] = rootX;
        } else if (rank[rootX] < rank[rootY]) {
            parent[rootX] = rootY;
        } else {
            parent[rootY] = rootX;
            rank[rootX]++;
        }
        return true;
    }
};
vector<int> findRedundantConnection(vector<vector<int>>& edges) {
    int n = edges.size();
    UnionFind uf(n + 1);
    for (const auto& edge : edges) {
        if (!uf.unionSets(edge[0], edge[1])) {
            return edge;
        }
    }
    return {};
}

int main() {
    int n;
    cout << "Enter the number of edges: ";
    cin >> n;
    vector<vector<int>> edges(n, vector<int>(2));
    cout << "Enter the edges (node1 node2):" << endl;
    for (int i = 0; i < n; i++) {
        cin >> edges[i][0] >> edges[i][1];
    }
    vector<int> result = findRedundantConnection(edges);
    if (!result.empty()) {
        cout << "The redundant edge is: [" << result[0] << ", " << result[1] << "]" << endl;
    } else {
        cout << "No redundant edge found." << endl;
    }
    return 0;
}
```

Output:



```
Enter the number of edges: 3
Enter the edges (node1 node2):
1 2
1 3
2 3
The redundant edge is: [2, 3]
```