Name : Archi bansal **UID: 22BCS15264**

Section: 620 A **Date: 28 dec 2024**

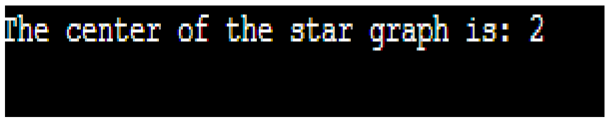
DOMAIN WINTER WINNING CAMP

1. **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.
   Code:

```
#include <iostream>
#include <vector>

using namespace std;

int findCenter(vector<vector<int>>& edges) {
   // In a star graph, the center node will appear in both the first two 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() {
   // Example input: edges of the star graph
   vector<vector<int>> edges = {{1, 2}, {2, 3}, {4, 2}};

   // Find the center of the star graph
   int center = findCenter(edges);
```

```
    // Output the result
    cout << "The center of the star graph is: " << center << endl;

    return 0;
}
```

Output



2. **In a town, there are n people labeled from 1 to n. There is a rumor that one of these people is secretly the town judge.**
   If the town judge exists, then:
   The town judge trusts nobody.
   Everybody (except for the town judge) trusts the town judge.
   There is exactly one person that satisfies properties 1 and 2.
   You are given an array trust where trust[i] = [ai, bi] representing that the person labeled ai trusts the person labeled bi. If a trust relationship does not exist in trust array, then such a trust relationship does not exist.
   Return the label of the town judge if the town judge exists and can be identified, or return -1 otherwise.
   Code:

```
#include <iostream>
#include <vector>

using namespace std;

int findJudge(int n, vector<vector<int>>& trust) {
    vector<int> trustCount(n + 1, 0); // Array to track trust counts

    // Process the trust relationships
    for (const auto& t : trust) {
        trustCount[t[0]]--; // The person who trusts loses a point
        trustCount[t[1]]++; // The person who is trusted gains a point
    }
```

```
    // Check for the town judge
    for (int i = 1; i <= n; ++i) {
        if (trustCount[i] == n - 1) { // The judge is trusted by everyone except themselves
            return i;
        }
    }

    return -1; // No town judge found
}

int main() {
    // Example input: number of people and trust relationships
    int n = 3;
    vector<vector<int>> trust = {{1, 3}, {2, 3}};

    // Find the town judge
    int judge = findJudge(n, trust);

    // Output the result
    if (judge == -1) {
        cout << "There is no town judge." << endl;
    } else {
        cout << "The town judge is: " << judge << endl;
    }

    return 0;
}
```

Output:

```
The town judge is: 3
```

3. **You are given an image represented by an m x n grid of integers image, where image[i][j] represents the pixel value of the image. You are also given three integers sr, sc, and color. Your task is to perform a flood fill on the image starting from the pixel image[sr][sc].**

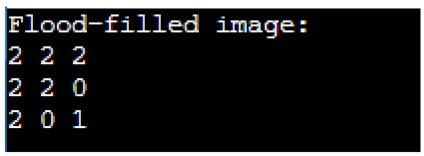
   Code:

```
#include <iostream>
```

```cpp
#include <vector>

using namespace std;

void floodFillUtil(vector<vector<int>>& image, int sr, int sc, int newColor, int originalColor) {
    // Check for boundary conditions and if the current pixel needs to be filled
    if (sr < 0 || sr >= image.size() || sc < 0 || sc >= image[0].size() || image[sr][sc] != originalColor || image[sr][sc] == newColor) {
        return;
    }

    // Fill the pixel with the new color
    image[sr][sc] = newColor;

    // Recursively call for adjacent pixels
    floodFillUtil(image, sr + 1, sc, newColor, originalColor);
    floodFillUtil(image, sr - 1, sc, newColor, originalColor);
    floodFillUtil(image, sr, sc + 1, newColor, originalColor);
    floodFillUtil(image, sr, sc - 1, newColor, originalColor);
}

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

int main() {
    // Example input: image, starting pixel, and new color
    vector<vector<int>> image = {
        {1, 1, 1},
        {1, 1, 0},
        {1, 0, 1}
    };
    int sr = 1, sc = 1, newColor = 2;
```

```
    // Perform the flood fill
    vector<vector<int>> result = floodFill(image, sr, sc, newColor);

    // Output the result
    cout << "Flood-filled image:" << endl;
    for (const auto& row : result) {
        for (int pixel : row) {
            cout << pixel << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Output:



```
Flood-filled image:
2 2 2
2 2 0
2 0 1
```

4. **Given a connected undirected graph represented by an adjacency list adj, which is a vector of vectors where each adj[i] represents the list of vertices connected to vertex i. Perform a Breadth First Traversal (BFS) starting from vertex 0, visiting vertices from left to right according to the adjacency list, and return a list containing the BFS traversal of the graph.**

   Code:

```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

vector<int> bfsTraversal(int V, vector<vector<int>>& adj) {
    vector<int> traversal; // To store the BFS traversal order
    vector<bool> visited(V, false); // To keep track of visited vertices
    queue<int> q; // Queue for BFS
```

```
    // Start BFS from vertex 0
    q.push(0);
    visited[0] = true;

    while (!q.empty()) {
        int current = q.front();
        q.pop();
        traversal.push_back(current);

        // Visit all adjacent vertices of the current vertex
        for (int neighbor : adj[current]) {
            if (!visited[neighbor]) {
                q.push(neighbor);
                visited[neighbor] = true;
            }
        }
    }

    return traversal;
}

int main() {
    // Example input: number of vertices and adjacency list
    int V = 5;
    vector<vector<int>> adj = {
        {1, 2},    // Edges from vertex 0
        {0, 3, 4}, // Edges from vertex 1
        {0},       // Edges from vertex 2
        {1},       // Edges from vertex 3
        {1}        // Edges from vertex 4
    };

    // Perform BFS traversal
    vector<int> result = bfsTraversal(V, adj);

    // Output the result
    cout << "BFS Traversal: ";
    for (int vertex : result) {
```

```
        cout << vertex << " ";
    }
    cout << endl;

    return 0;
}
```

Output:

```
BFS Traversal: 0 1 2 3 4
```

5. **Given a connected undirected graph represented by an adjacency list adj, which is a vector of vectors where each adj[i] represents the list of vertices connected to vertex i. Perform a Depth First Traversal (DFS) starting from vertex 0, visiting vertices from left to right as per the adjacency list, and return a list containing the DFS traversal of the graph.**

   Code:

```
#include <iostream>
#include <vector>

using namespace std;

void dfsUtil(int current, vector<vector<int>>& adj, vector<bool>& visited, vector<int>&
traversal) {
    // Mark the current vertex as visited and add it to the traversal
    visited[current] = true;
    traversal.push_back(current);

    // Recursively visit all unvisited neighbors
    for (int neighbor : adj[current]) {
        if (!visited[neighbor]) {
            dfsUtil(neighbor, adj, visited, traversal);
        }
    }
}

vector<int> dfsTraversal(int V, vector<vector<int>>& adj) {
    vector<int> traversal; // To store the DFS traversal order
```

```
    vector<bool> visited(V, false); // To keep track of visited vertices

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

    return traversal;
}

int main() {
    // Example input: number of vertices and adjacency list
    int V = 5;
    vector<vector<int>> adj = {
        {1, 2},    // Edges from vertex 0
        {0, 3, 4}, // Edges from vertex 1
        {0},       // Edges from vertex 2
        {1},       // Edges from vertex 3
        {1}        // Edges from vertex 4
    };

    // Perform DFS traversal
    vector<int> result = dfsTraversal(V, adj);

    // Output the result
    cout << "DFS Traversal: ";
    for (int vertex : result) {
        cout << vertex << " ";
    }
    cout << endl;

    return 0;
}
```

Output:

```
DFS Traversal: 0 1 3 4 2
```

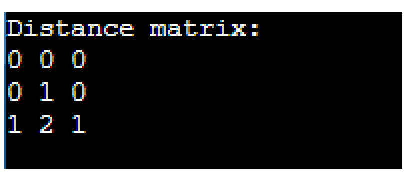
6. **Given an m x n binary matrix mat, return the distance of the nearest 0 for each cell. The distance between two adjacent cells is 1.**

Code:

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits> // For INT_MAX

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)); // Distance matrix
initialized to infinity
    queue<pair<int, int>> q; // Queue for BFS

    // Initialize the queue with all 0 cells and set their distance to 0
    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 moving up, down, left, and right
    vector<pair<int, int>> directions = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

    // Perform BFS
    while (!q.empty()) {
        pair<int, int> cell = q.front();
        q.pop();
        int x = cell.first, y = cell.second;

        for (auto direction : directions) {
            int newX = x + direction.first;
            int newY = y + direction.second;

            // Check bounds and update distance if shorter path is found
            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() {
    // Example input: binary matrix
    vector<vector<int>> mat = {
        {0, 0, 0},
        {0, 1, 0},
        {1, 1, 1}
    };

    // Compute the distance matrix
    vector<vector<int>> result = updateMatrix(mat);

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

    return 0;
}
```

Output:



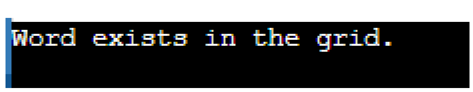
7. **Given an m x n grid of characters board and a string word, return true if word exists in the grid. The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.**

   Code:

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

bool dfs(vector<vector<char>>& board, string& word, int i, int j, int index) {
  // Base case: if the entire word is matched
  if (index == word.size()) {
    return true;
  }

  // Boundary check and character mismatch check
  if (i < 0 || i >= board.size() || j < 0 || j >= board[0].size() || board[i][j] != 
word[index]) {
    return false;
  }

  // Temporarily mark the cell as visited
  char temp = board[i][j];
  board[i][j] = '#';

  // Explore all four possible directions (up, down, left, right)
  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);

  // Restore the cell back to its original value
  board[i][j] = temp;

  return found;
}
```

```cpp
bool exist(vector<vector<char>>& board, string word) {
    int m = board.size();
    int n = board[0].size();

    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            // Start DFS if the first character matches
            if (board[i][j] == word[0] && dfs(board, word, i, j, 0)) {
                return true;
            }
        }
    }

    return false;
}

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

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

    return 0;
}
```

Output:



8. **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.

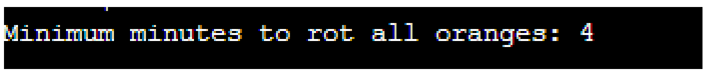
Code:

```
#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; // Queue to perform BFS
   int freshCount = 0;      // Count of fresh oranges

   // Initialize the queue with all rotten oranges and count fresh oranges
   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) {
            freshCount++;
         }
      }
   }

   if (freshCount == 0) return 0; // No fresh oranges to begin with

   int minutes = 0;
   vector<pair<int, int>> directions = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}}; // Directions: right,
down, left, up
```

```
    // Perform BFS
    while (!q.empty()) {
        int size = q.size();
        bool rotten = false; // Track if we rot any fresh oranges in this minute

        for (int i = 0; i < size; ++i) {
            auto [x, y] = q.front();
            q.pop();

            for (auto [dx, dy] : directions) {
                int newX = x + dx;
                int newY = y + dy;

                // Check bounds and if the cell contains a fresh orange
                if (newX >= 0 && newX < m && newY >= 0 && newY < n &&
grid[newX][newY] == 1) {
                    grid[newX][newY] = 2; // Make the orange rotten
                    q.push({newX, newY});
                    freshCount--;         // Decrease fresh orange count
                    rotten = true;
                }
            }
        }

        if (rotten) minutes++; // Increment time only if at least one orange was rotted
    }

    // If there are still fresh oranges left, return -1
    return freshCount == 0 ? minutes : -1;
}

int main() {
    vector<vector<int>> grid = {
        {2, 1, 1},
        {1, 1, 0},
        {0, 1, 1}
    };
```

```
    int result = orangesRotting(grid);
    if (result == -1) {
        cout << "Impossible to rot all oranges." << endl;
    } else {
        cout << "Minimum minutes to rot all oranges: " << result << endl;
    }

    return 0;
}
```

Output:



9. **There is an m x n rectangular island that borders both the Pacific Ocean and Atlantic Ocean. The Pacific Ocean touches the island's left and top edges, and the Atlantic Ocean touches the island's right and bottom edges.**
   The island is partitioned into a grid of square cells. You are given an m x n integer matrix heights where heights[r][c] represents the height above sea level of the cell at coordinate (r, c).
   The island receives a lot of rain, and the rain water can flow to neighboring cells directly north, south, east, and west if the neighboring cell's height is less than or equal to the current cell's height. Water can flow from any cell adjacent to an ocean into the ocean.
   Return a 2D list of grid coordinates result where result[i] = [ri, ci] denotes that rain water can flow from cell (ri, ci) to both the Pacific and Atlantic oceans.
   Code:

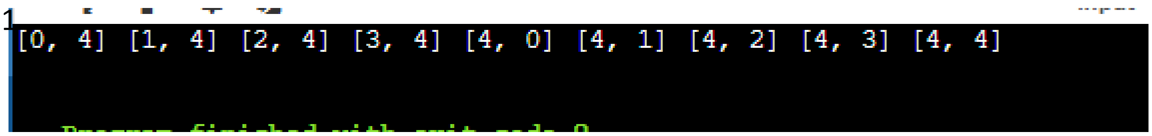
```
#include <vector>
#include <iostream>

using namespace std;

const int MIN_INT = -2147483648;

void dfs(vector<vector<int>>& heights, vector<vector<bool>>& visited, int row, int col, int prevHeight) {
    int m = heights.size();
```

```
    int n = heights[0].size();

    // Base condition: out of bounds or the current cell is not valid for water flow
    if (row < 0 || col < 0 || row >= m || col >= n || visited[row][col] ||
heights[row][col] < prevHeight) {
        return;
    }

    // Mark the cell as visited
    visited[row][col] = true;

    // Explore all 4 directions: up, down, left, right
    dfs(heights, visited, row + 1, col, heights[row][col]);  // Down
    dfs(heights, visited, row - 1, col, heights[row][col]);  // Up
    dfs(heights, visited, row, col + 1, heights[row][col]);  // Right
    dfs(heights, visited, row, col - 1, heights[row][col]);  // Left
}

vector<vector<int>> pacificAtlantic(vector<vector<int>>& heights) {
    if (heights.empty() || heights[0].empty()) {
        return {};
    }

    int m = heights.size();
    int n = heights[0].size();

    // Visited matrices for Pacific and Atlantic oceans
    vector<vector<bool>> pacific(m, vector<bool>(n, false));
    vector<vector<bool>> atlantic(m, vector<bool>(n, false));

    // DFS for Pacific (top and left edges)
    for (int i = 0; i < m; ++i) {
        dfs(heights, pacific, i, 0, MIN_INT);  // Pacific left edge
    }
    for (int j = 0; j < n; ++j) {
        dfs(heights, pacific, 0, j, MIN_INT);  // Pacific top edge
    }
```

```cpp
    // DFS for Atlantic (bottom and right edges)
    for (int i = 0; i < m; ++i) {
        dfs(heights, atlantic, i, n - 1, MIN_INT);  // Atlantic right edge
    }
    for (int j = 0; j < n; ++j) {
        dfs(heights, atlantic, m - 1, j, MIN_INT);  // Atlantic bottom edge
    }

    // Collecting results where water can flow to both oceans
    vector<vector<int>> result;
    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() {
    vector<vector<int>> heights = {
        {1, 2, 3, 4, 5},
        {2, 3, 4, 5, 6},
        {3, 4, 5, 6, 7},
        {4, 5, 6, 7, 8},
        {5, 6, 7, 8, 9}
    };

    vector<vector<int>> result = pacificAtlantic(heights);

    // Output the result
    for (auto& cell : result) {
        cout << "[" << cell[0] << ", " << cell[1] << "] ";
    }
    cout << endl;
```

```
    return 0;
}
```

Output:



10. **You are given an m x n binary matrix grid. An island is a group of 1's (representing land) connected 4-directionally (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.**
    The area of an island is the number of cells with a value 1 in the island.
    Return the maximum area of an island in grid. If there is no island, return 0.
    Code:

```
#include <vector>
#include <iostream>

using namespace std;

// DFS function to explore the island and calculate its area
int dfs(vector<vector<int>>& grid, int row, int col) {
    int m = grid.size();
    int n = grid[0].size();

    // Base case: Out of bounds or the cell is water (0)
    if (row < 0 || col < 0 || row >= m || col >= n || grid[row][col] == 0) {
        return 0;
    }

    // Mark the current cell as visited by changing it to 0 (water)
    grid[row][col] = 0;

    // Initialize the area of the current island
    int area = 1;

    // Explore all 4 directions: down, up, right, left
    area += dfs(grid, row + 1, col);  // Down
```

```
    area += dfs(grid, row - 1, col);  // Up
    area += dfs(grid, row, col + 1);  // Right
    area += dfs(grid, row, col - 1);  // Left

    return area;
}

int maxAreaOfIsland(vector<vector<int>>& grid) {
    int maxArea = 0;

    // Iterate through the grid to find all islands
    for (int i = 0; i < grid.size(); ++i) {
        for (int j = 0; j < grid[0].size(); ++j) {
            // If we find a land cell (1), perform DFS to calculate the area of the island
            if (grid[i][j] == 1) {
                maxArea = max(maxArea, dfs(grid, i, j));
            }
        }
    }

    return maxArea;
}

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

    cout << "Maximum area of an island: " << maxAreaOfIsland(grid) << endl;

    return 0;
}
```

Output: