**Experiment: - 9**

**Student Name:** Medidi Vinnbabu **UID:**22BCS16600

**Branch:** CSE **Section/Group:** 22BCS IOT-640/B

**Semester:** 6th **Date of Performance:** 02/04/2025

**Subject Name:** Advanced Programming Lab-2 **Subject Code:** 22CSP-351

**Problem -1 1. Aim:** Number of Islands

1. **Objective:** 
   * **Learn to Identify Islands in a Grid:** Understand how to recognize separate land regions in a 2D grid where '1' represents land and '0' represents water.
   * **Use Depth-First Search (DFS) for Exploration:** Learn how DFS helps in visiting all connected land cells, ensuring each island is counted only once.
   * **Implement Grid Traversal Effectively:** Understand how to scan each cell in the grid systematically, making sure no land portion is left unchecked.
   * **Apply Recursion to Find Connected Areas:** Learn how recursive function calls help explore all possible directions (up, down, left, right) to find the full extent of an island.
   * **Enhance Problem-Solving Abilities in Graph Theory:** Develop skills in handling graph-based problems, such as finding connected components, which have real-world applications.
2. **Implementation/Code:**

class Solution { public: void dfs(vector<vector<char>>& grid, int i, int j) { int m = grid.size(), n = grid[0].size();

if (i < 0 || j < 0 || i >= m || j >= n || grid[i][j] == '0') { return;

} grid[i][j] = '0'; dfs(grid, i + 1, j); dfs(grid, i - 1, j); dfs(grid, i, j + 1); dfs(grid, i, j - 1);

}

int numIslands(vector<vector<char>>& grid) { if (grid.empty()) return 0;

int m = grid.size(), n = grid[0].size(), count = 0; for (int i = 0; i < m; ++i) { for (int j = 0; j < n; ++j) { if (grid[i][j] == '1') {

++count; dfs(grid, i, j);

}

}

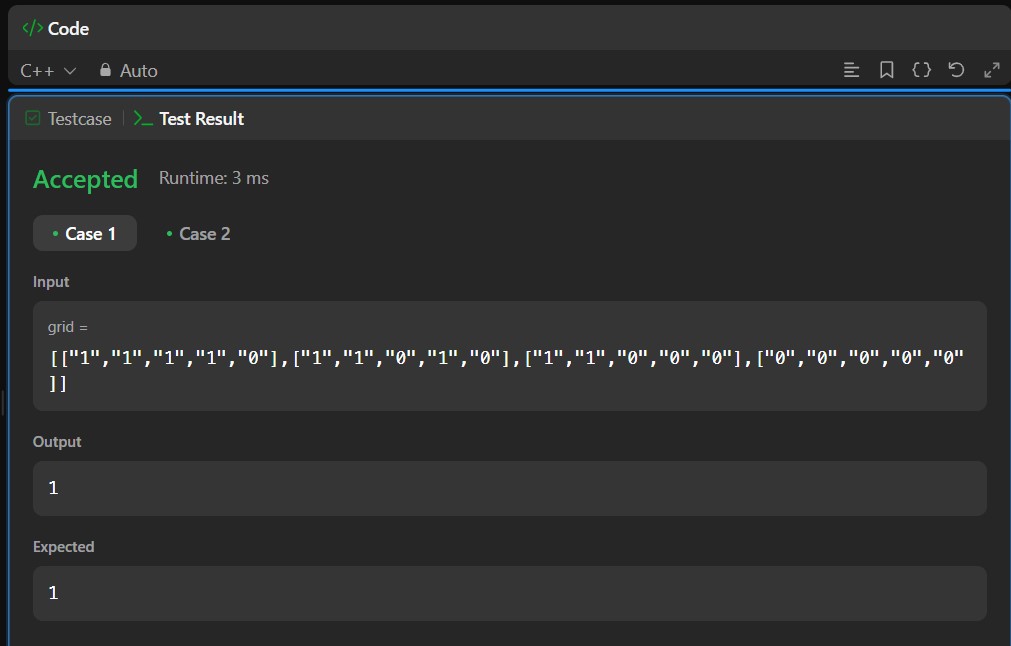
}

return count;

}

};

1. **Output**



***Figure 1***

1. **Learning Outcomes:** 
   * **Ability to Count Islands in a Grid:** Gain the skill to count distinct islands in a binary grid by detecting connected land regions.
   * **Understanding of DFS and Its Application:** Learn how Depth-First Search (DFS) is used to traverse and mark visited land cells in a grid.
   * **Efficiency in Grid-Based Problem Solving:** Become proficient in scanning and modifying grid structures to solve connectivity problems.
   * **Mastering Recursion for Connectivity Checks:** Develop an understanding of recursive algorithms for exploring all possible paths in a grid.
   * **Improved Logical Thinking and Coding Skills:** Strengthen logical reasoning by solving complex problems related to graphs and connected components.

# Problem-2

1. **Aim:** Surrounded Regions
2. **Objectives:**
   * **Understand Capturing Regions in a Grid:** Learn how to identify and replace 'O' regions that are completely surrounded by 'X' in a 2D matrix.
   * **Use Depth-First Search (DFS) for Traversal:** Explore how DFS helps mark connected 'O' cells on the board edges, preventing them from being captured.
   * **Handle Edge Cases Efficiently:** Understand how to correctly process the grid by checking border 'O' cells first and avoiding unnecessary replacements.
   * **Modify the Grid in Place:** Learn how to update the given board directly without using extra memory, making the solution efficient.
   * **Improve Logical Thinking in Grid Problems:** Strengthen problem-solving skills by working with matrix-based transformations and connected components.
3. **Implementation/Code:**

class Solution { public: void dfs(vector<vector<char>>& board, int i, int j) { int m = board.size(), n = board[0].size();

if (i < 0 || i >= m || j < 0 || j >= n || board[i][j] != 'O') return; board[i][j] = '#'; dfs(board, i + 1, j); dfs(board, i - 1, j); dfs(board, i, j + 1); dfs(board, i, j - 1);

}

void solve(vector<vector<char>>& board) { int m = board.size(), n = board[0].size(); if (m == 0 || n == 0) return; for (int i = 0; i < m; i++) {

if (board[i][0] == 'O') dfs(board, i, 0); if (board[i][n - 1] == 'O') dfs(board, i, n - 1);

}

for (int j = 0; j < n; j++) {

if (board[0][j] == 'O') dfs(board, 0, j); if (board[m - 1][j] == 'O') dfs(board, m - 1, j);

}

for (int i = 0; i < m; i++) {

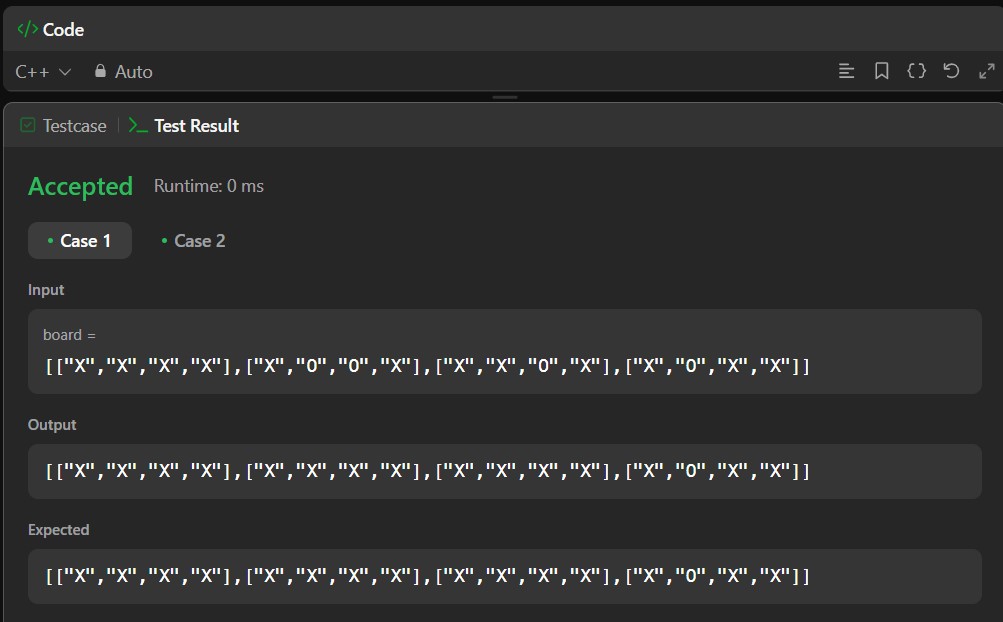
for (int j = 0; j < n; j++) { if (board[i][j] == 'O') board[i][j] = 'X'; if (board[i][j] == '#') board[i][j] = 'O'; }

}

}

};

1. **Output:**

 ***Figure 2***

1. **Learning Outcomes:**

* + **Ability to Detect Surrounded Regions:** Gain the skill to identify and replace 'O' regions that are fully enclosed by 'X' cells.
  + **Understanding of DFS for Grid Exploration:** Learn how DFS can traverse connected components in a 2D grid and mark visited cells.
  + **Mastering Edge Case Handling:** Develop techniques to correctly identify which 'O' regions should be replaced and which should remain.
  + **Efficiently Modifying Data Structures:** Learn how to update the board in place using temporary markers, ensuring an optimized approach.
  + **Enhancing Coding and Problem-Solving Skills:** Improve the ability to implement algorithms that modify grids dynamically, useful in various applications.

# Problem: - 3

1. **Aim:** Lowest Common Ancestor of a Binary Tree
2. **Objectives:**

* + Learn how to find the lowest common ancestor of two nodes in a binary tree using recursion. This helps in understanding hierarchical relationships in trees and improves knowledge of treebased algorithms.
  + Understand how depth-first search (DFS) is used to traverse the tree efficiently. This method helps in searching for nodes and their ancestors and enhances tree traversal techniques.
  + Improve problem-solving skills by analysing tree structures and solving ancestor-related problems. This enhances logical thinking in programming and helps in developing efficient solutions.
  + Learn to handle base cases and edge cases in recursive tree problems. This ensures the solution works correctly for all possible inputs and prevents errors in complex tree structures.
  + Develop coding skills by implementing tree traversal techniques. This helps in solving similar tree-based problems in interviews and real-world applications, making coding more efficient.

1. **Implementation/Code:**

class Solution { public:

TreeNode\* lowestCommonAncestor(TreeNode\* root, TreeNode\* p, TreeNode\* q) { if (root == NULL || root == p || root == q) { return root; }

TreeNode\* left = lowestCommonAncestor(root->left, p, q); TreeNode\* right = lowestCommonAncestor(root->right, p, q); if (left != NULL && right != NULL) { return root;

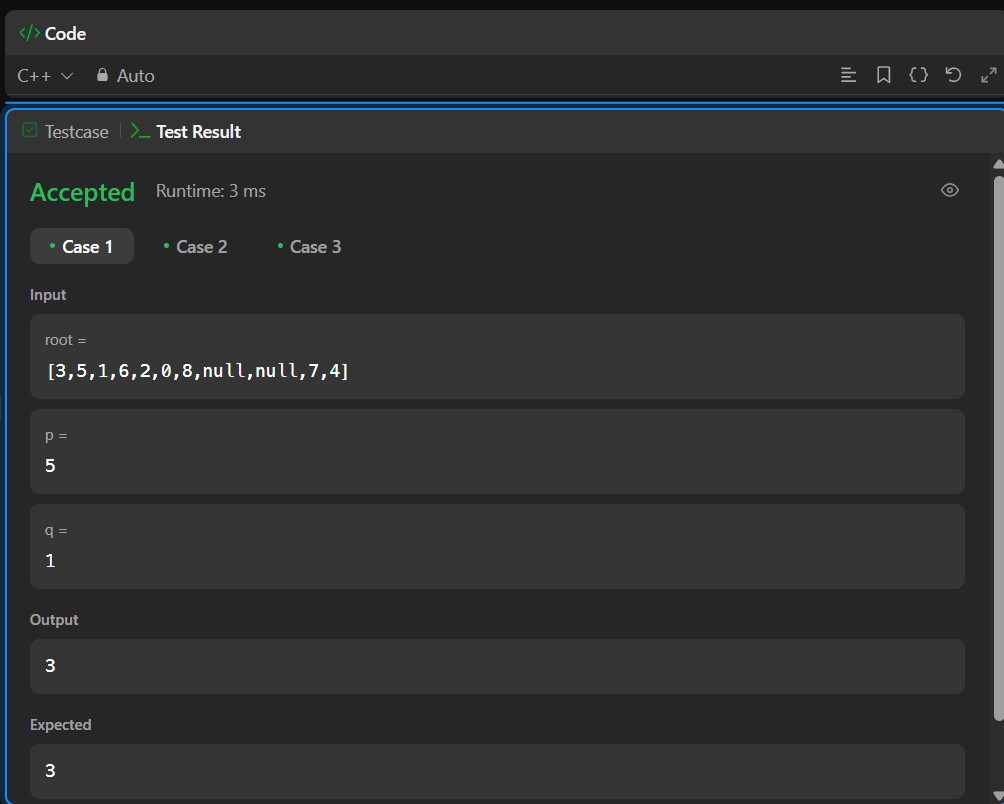
}

return left != NULL ? left : right;

}

};

1. **Output:**

 ***Figure 3***

1. **Learning Outcomes:** 
   * You will be able to find the lowest common ancestor of two given nodes in a binary tree. This will help in solving hierarchical tree problems.
   * You will understand how recursion helps in solving complex tree-based problems. This will improve your ability to write efficient recursive functions.
   * You will learn to apply depth-first search (DFS) to navigate through trees. This will make it easier to find specific nodes and their ancestors.
   * You will gain confidence in handling base cases and edge cases in recursive solutions. This will ensure your code runs correctly for all scenarios.
   * You will be able to write clear and optimized C++ code for tree problems. This will strengthen your programming skills and logical thinking.