Experiment:-9

Student Name: Himanshu **UID: 22BCS13141**

Branch: CSE **Section/Group:**22BCSIOT-640/B

Semester:6th **DateofPerformance:**02/04/2025

SubjectName: AdvancedProgrammingLab-2 **SubjectCode:**22CSP-351

Problem-1

1. **Aim:** Number of Islands

2. **Objective:**

- **LearntoIdentifyIslandsinaGrid:**Understandhowtorecognizeseparatelandregionsina 2D grid where '1' represents land and '0' represents water.
- **UseDepth-FirstSearch(DFS)forExploration:**LearnhowDFShelpsinvisitingallconnected land cells, ensuring each island is counted only once.
- **ImplementGridTraversalEffectively:**Understandhowtoscaneachcellinthegrid systematically, making sure no land portion is left unchecked.
- **ApplyRecursiontoFindConnectedAreas:**Learnhowrecursivefunctioncallshelpexplore all possible directions (up, down, left, right) to find the full extent of an island.
- **EnhanceProblem-SolvingAbilitiesinGraphTheory:**Developskillsinhandlinggraph-based problems, such as finding connected components, which have real-world applications.

3. **Implementation/Code:**

```
classSolution{ public:
   voiddfs(vector<vector<char>>&grid,inti,intj){ 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;
      intm=grid.size(),n=grid[0].size(),count=0; for
      (int i = 0; i < m; ++i) {
         for(intj=0;j<n;++j){ if
            (grid[i][j] == '1') {
```

```
                    ++count;
                    dfs(grid,i,j);
                }
            }
        }
        returncount;
    }
};
```

4. Output

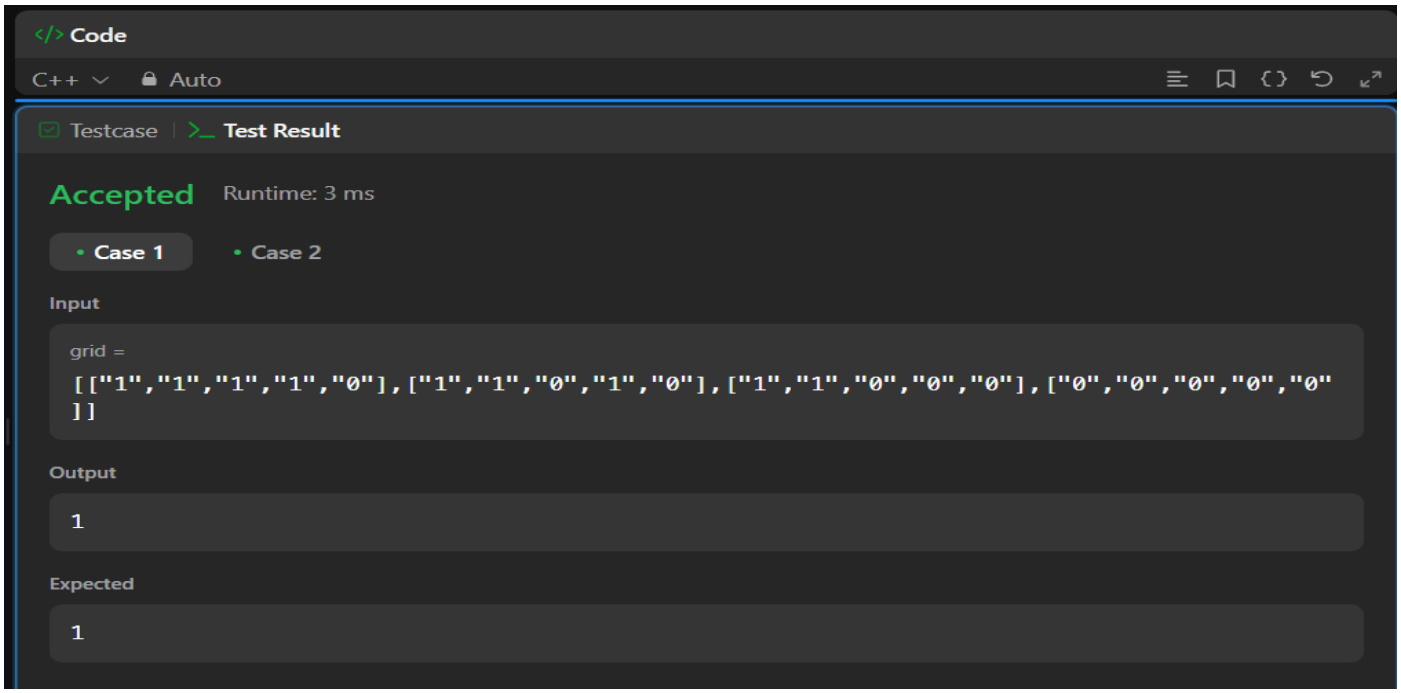

Figure1

5. LearningOutcomes:

- **AbilitytoCountIslandsinaGrid:**Gaintheskilltocountdistinctislandsinabinarygridby detecting connected land regions.
- **UnderstandingofDFSandItsApplication:**LearnhowDepth-FirstSearch(DFS)isusedto traverse and mark visited land cells in a grid.
- **EfficiencyinGrid-BasedProblemSolving:**Becomeproficientinscanningandmodifyinggrid structures to solve connectivity problems.
- **MasteringRecursionforConnectivityChecks:**Developanunderstandingofrecursive algorithms for exploring all possible paths in a grid.
- **ImprovedLogicalThinkingandCodingSkills:**Strengthenlogicalreasoningbysolving complex problems related to graphs and connected components.

Problem-2

1. **Aim:** SurroundedRegions
2. **Objectives:**

- **UnderstandCapturingRegionsinaGrid:** Learnhowtoidentifyandreplace'O'regionsthat are completely surrounded by 'X' in a 2D matrix.
- **UseDepth-FirstSearch(DFS)forTraversal:** ExplorehowDFShelpsmarkconnected'O'cells on the board edges, preventing them from being captured.
- **HandleEdge CasesEfficiently:** Understandhowtocorrectlyprocessthe gridbychecking border 'O' cells first and avoiding unnecessary replacements.
- **ModifytheGridinPlace:** Learnhowtoupdatethegivenboarddirectlywithoutusingextra memory, making the solution efficient.
- **ImproveLogicalThinkinginGridProblems:** Strengthenproblem-solvingskillsbyworking with matrix-based transformations and connected components.

3. **Implementation/Code:**

```
classSolution{ public:
   voiddfs(vector<vector<char>>&board,inti,intj){ 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);
   }
   voidsolve(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';
            }
        }
    }
};
```

4. Output:

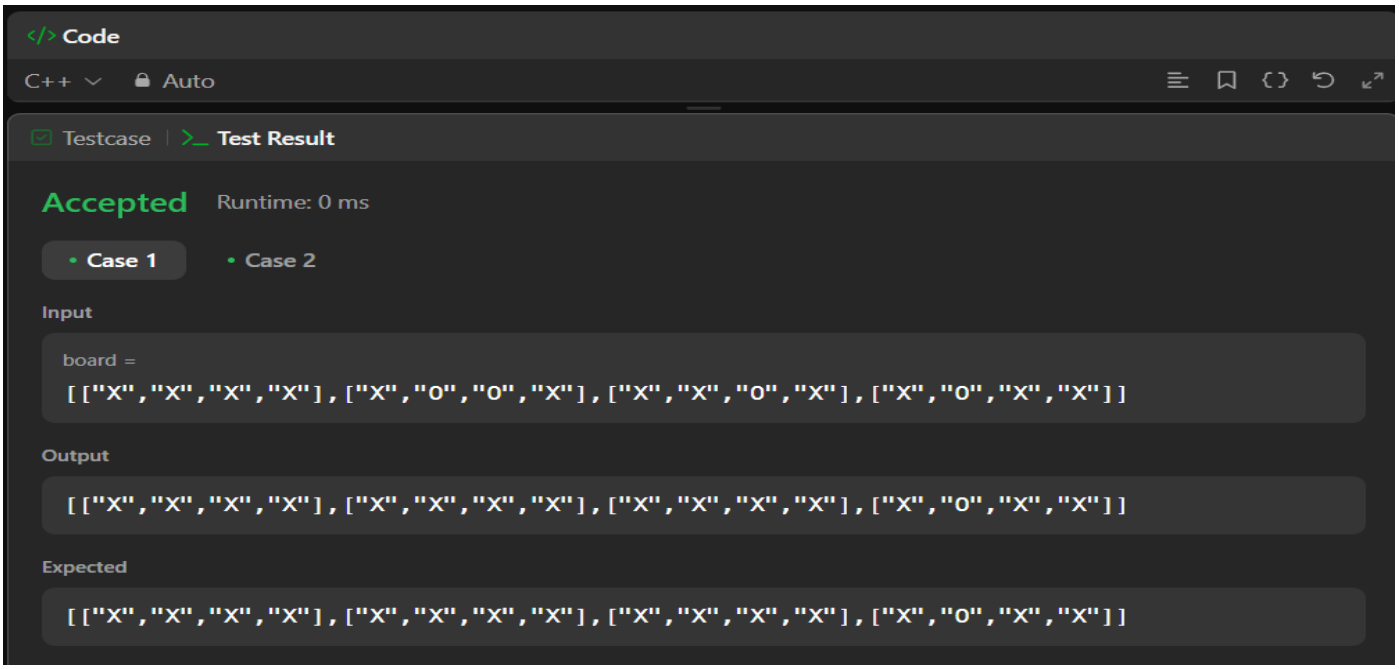

Figure2

5. LearningOutcomes:

- **AbilitytoDetectSurroundedRegions:**Gaintheskilltoidentifyandreplace'O'regionsthat are fully enclosed by 'X' cells.
- **UnderstandingofDFSforGridExploration:**LearnhowDFScantraverseconnected components in a 2D grid and mark visited cells.
- **MasteringEdgeCaseHandling:**Developtechniquestocorrectlyidentifywhich'O'regions should be replaced and which should remain.
- **EfficientlyModifyingDataStructures:**Learnhowtoupdatetheboardinplaceusingtemporary markers, ensuring an optimized approach.
- **EnhancingCodingandProblem-SolvingSkills:**Improvetheabilitytoimplementalgorithms 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. Thishelpsinunderstandinghierarchicalrelationshipsintreesandimprovesknowledgeoftree- based 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.
- Learntohandlebasecasesandedgecasesinrecursivetreeproblems.This ensuresthesolution 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.

3. **Implementation/Code:**

```
classSolution{ public:
   TreeNode*lowestCommonAncestor(TreeNode*root,TreeNode*p,TreeNode*q){ if
      (root == NULL || root == p || root == q) {
         returnroot;
      }
      TreeNode* left = lowestCommonAncestor(root->left, p, q);
      TreeNode* right =lowestCommonAncestor(root->right, p, q);
      if (left != NULL && right != NULL) {
         returnroot;
      }
      returnleft !=NULL?left : right;
   }
};
```

4. Output:

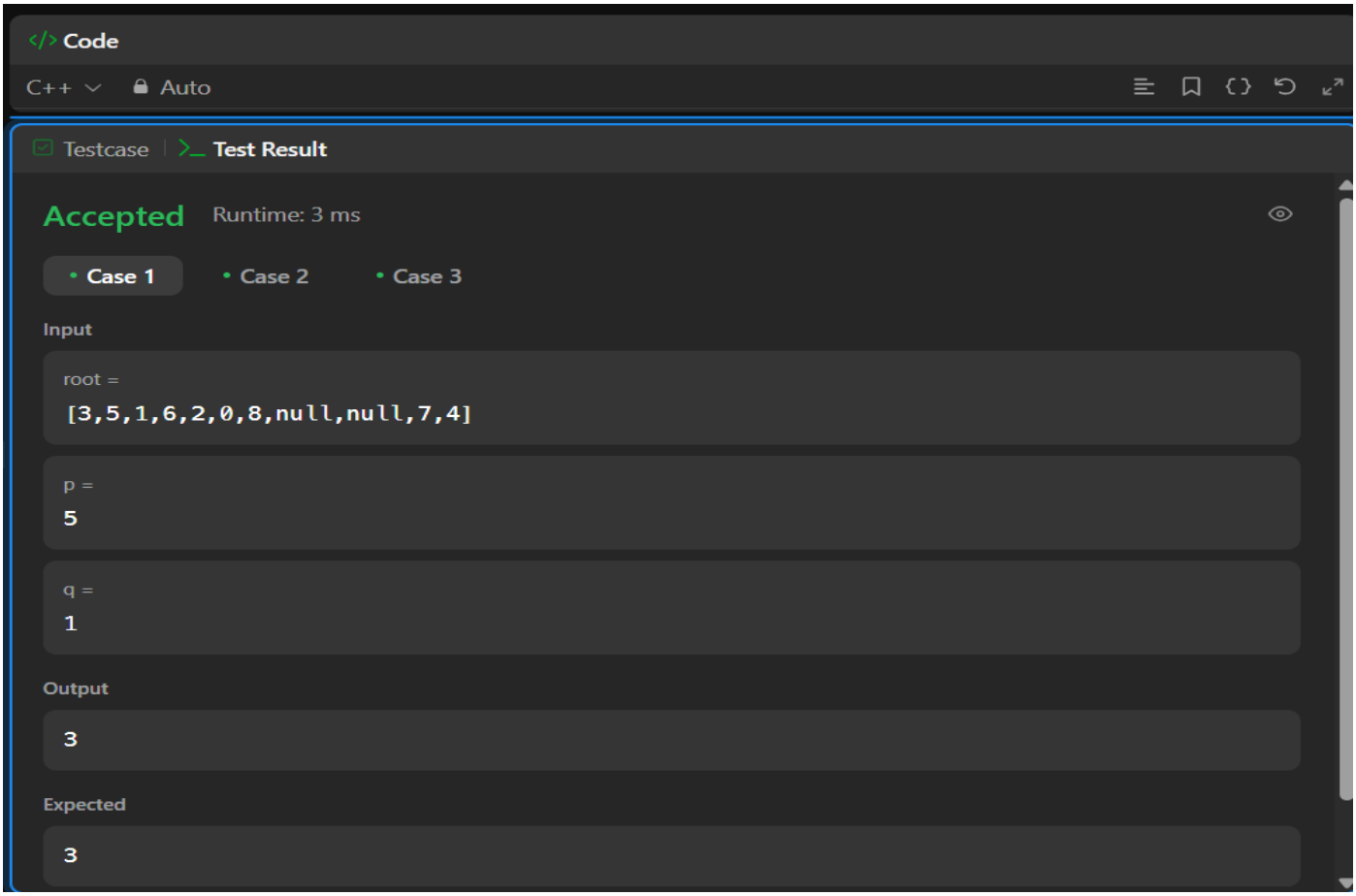

Figure3

5. LearningOutcomes:

- Youwillbeabletofindthelowestcommonancestoroftwogivennodesinabinarytree.This will help in solving hierarchical tree problems.
- You willunderstand how recursion helps in solving complex tree-basedproblems.This willimprove your ability to write efficient recursive functions.
- Youwilllearntoapplydepth-firstsearch(DFS)tonavigatethroughtrees.Thiswillmakeiteasier to find specific nodes and their ancestors.
- Youwillgainconfidenceinhandlingbasecasesandedgecasesinrecursivesolutions.Thiswill ensure your code runs correctly for all scenarios.
- YouwillbeabletowriteclearandoptimizedC++codefortreeproblems.Thiswillstrengthen your programming skills and logical thinking.