Experiment-8

Student Name: Anupreet Kaur **UID: 22BCS50071**
Branch: BE-CSE **Section/Group: _NTPP_IOT-602-A**
Semester: 6th **Date of Performance: 16/03/2025**
Subject Name: AP Lab **Subject Code: 22CSP-351**

1. Aim: Graphs.

- ❖ Problem 1.2.1: Lowest Common Ancestor of a Binary Tree
- ❖ Problem 1.2.2: Word Ladder
- ❖ Problem 1.2.2: Longest Increasing Path in a Matrix

2. Objective:

To find the lowest common ancestor (LCA) of two given nodes in a binary tree, where the LCA is the lowest node that has both nodes as descendants.
To determine the shortest transformation sequence length from a beginWord to an endWord using a dictionary of words, where each transformation changes only one letter at a time.
To find the length of the longest increasing path in a given m x n matrix, where movement is allowed only in four directions (up, down, left, and right).

3. Theory:

The Lowest Common Ancestor (LCA) problem in a binary tree uses a recursive approach to determine the deepest shared ancestor of two nodes, which is useful in hierarchical structures. The Word Ladder problem models word transformations as a graph and applies Breadth-First Search (BFS) to find the shortest path from beginWord to endWord, ensuring the minimum number of transformations. The Longest Increasing Path in a Matrix problem uses Depth-First Search (DFS) with memoization to explore the longest increasing sequence in a grid while avoiding redundant computations. These problems demonstrate efficient pathfinding and relationship identification using recursive search, dynamic programming, and graph traversal techniques.

4. Code:

Lowest Common Ancestor of a Binary Tree:

```
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;
    }
}
```

Word Ladder:

```
import java.util.*;
class Solution {
    public int ladderLength(String beginWord, String endWord, List<String>
wordList) {
        // Convert wordList to a set for fast lookup
        Set<String> wordSet = new HashSet<>(wordList);
        if (!wordSet.contains(endWord)) return 0
        Queue<String> queue = new LinkedList<>();
        queue.add(beginWord);
        int level = 1;
        while (!queue.isEmpty()) {
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                String word = queue.poll();
                char[] wordChars = word.toCharArray();

                // Try all possible one-letter transformations
                for (int j = 0; j < wordChars.length; j++) {
                    char originalChar = wordChars[j];

                    for (char c = 'a'; c <= 'z'; c++) {
                        if (c == originalChar) continue;
                        wordChars[j] = c;
                        String newWord = new String(wordChars);

                        if (newWord.equals(endWord)) return level + 1;
                        if (wordSet.contains(newWord)) {
                            queue.add(newWord);
                            wordSet.remove(newWord);
```

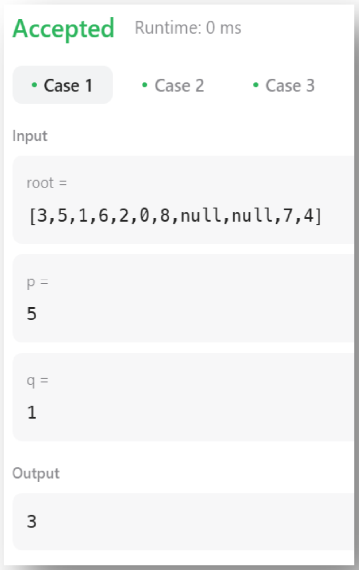
```
                    }
                }

                wordChars[j] = originalChar; // Restore original character
            }
        }
        level++;
    }
    return 0;
  }
}
```

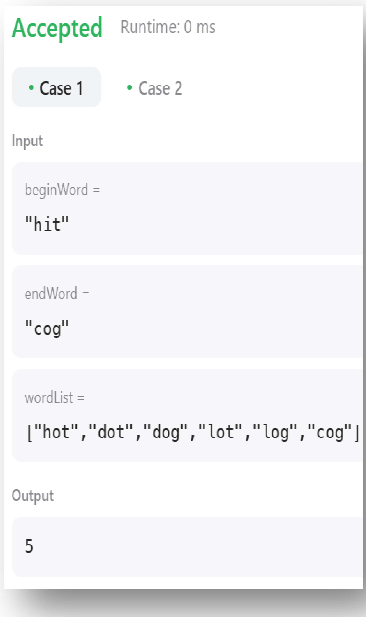
Longest Increasing Path in a Matrix:

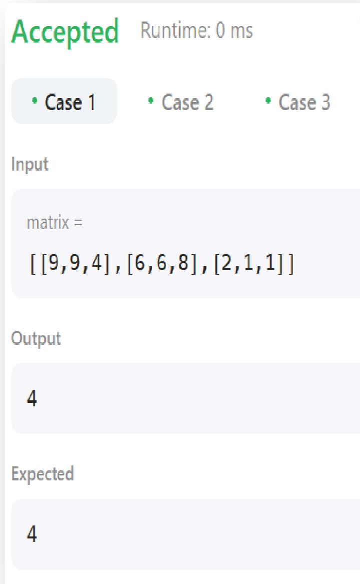
```
class Solution {
  private int[][] directions = {{0,1}, {1,0}, {0,-1}, {-1,0}};    private int m, n;

  public int longestIncreasingPath(int[][] matrix) {
    if (matrix == null || matrix.length == 0) return 0;

    m = matrix.length;
    n = matrix[0].length;
    int[][] dp = new int[m][n]; // Memoization table
    int maxPath = 0;

    // Explore each cell and find the longest path starting from it
    for (int i = 0; i < m; i++) {
      for (int j = 0; j < n; j++) {
        maxPath = Math.max(maxPath, dfs(matrix, i, j, dp));
      }
    }

    return maxPath;
  }

  private int dfs(int[][] matrix, int i, int j, int[][] dp) {
    if (dp[i][j] != 0) return dp[i][j];

    int maxLength = 1;
    for (int[] dir : directions) {
      int x = i + dir[0], y = j + dir[1];
      if (x >= 0 && x < m && y >= 0 && y < n && matrix[x][y] > matrix[i][j])
{
        maxLength = Math.max(maxLength, 1 + dfs(matrix, x, y, dp));
      }
```

```
            }

            dp[i][j] = maxLength;
            return maxLength;
        }
    }
```

6. Output:







7. Learning Outcomes:

- Understand Recursive and Graph Traversal Techniques – Learn how recursion, Depth-First Search (DFS), and Breadth-First Search (BFS) are applied to tree and graph-based problems.
- Optimize Search Using Dynamic Programming and Memoization – Explore how memoization reduces redundant calculations, improving efficiency in problems like the longest increasing path in a matrix.
- Apply Hierarchical and Graph-Based Problem Solving – Gain insights into solving hierarchical problems (LCA in trees) and shortest path problems (Word Ladder) using graph theory.
- Enhance Algorithmic Thinking for Real-World Applications – Develop skills to apply tree and graph traversal techniques to areas like network routing, AI navigation, and computational linguistics.