Experiment 8

Student Name: Rhythm Tyagi
UID: 22BCS17203
Branch: CSE
Section/Group: IOT_NTPP_602-A
Semester: 6th
Date of Performance: 10-04-25
Subject Name: AP2
Subject Code: 22CSP-351

Aim:

1. Word Ladder
   Given two words, beginWord and endWord, and a dictionary wordList, return *the number of words in the shortest transformation sequence from* begin Word *to* end Word, *or* 0 *if no such sequence exists.*

2. Lowest Common Ancestor of a Binary Tree
   Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree

3. Course Schedule
   There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [a_i, b_i] indicates that you must take course b_i first if you want to take course a_i

Objective: Develop efficient algorithms to solve graph traversal, tree recursion, and topological sorting problems by implementing Word Ladder (BFS), Lowest Common Ancestor (Recursion), and Course Schedule (Topological Sort).

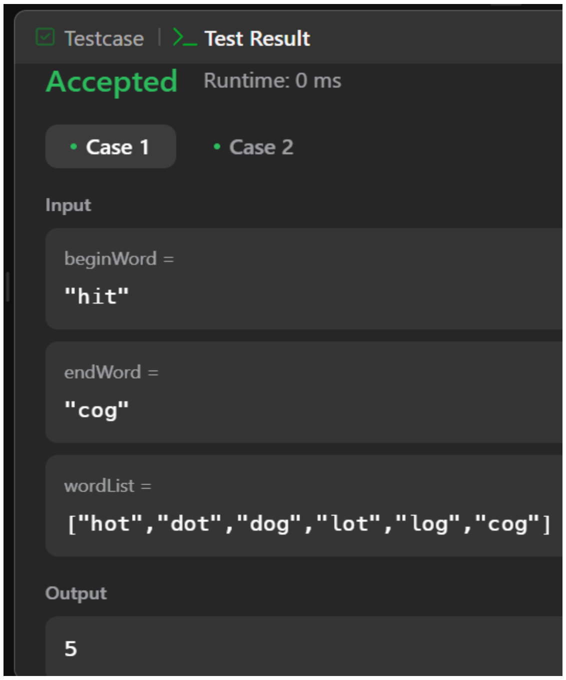
Algorithm 1:

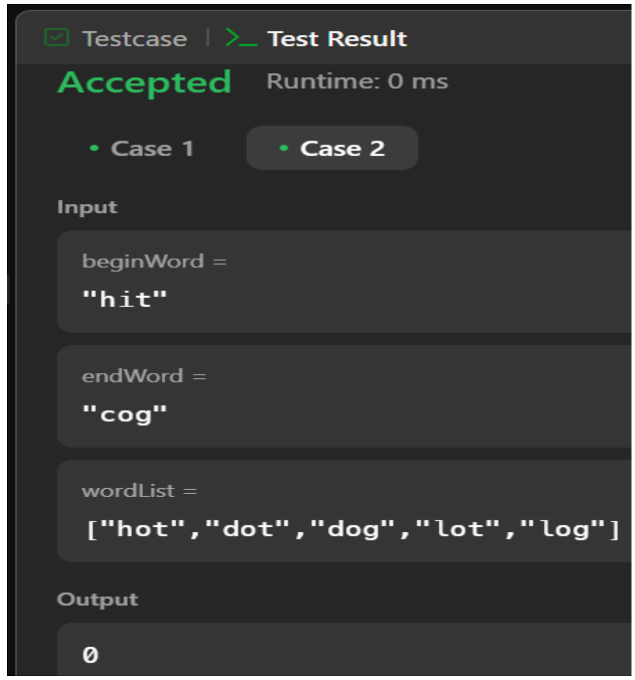
1. Convert wordList to a **HashSet** for quick lookups.
2. Use a **queue** for BFS, starting with beginWord.
3. Set steps = 1 to count transformation levels.
4. For each word in the queue, generate all possible **one-letter transformations**.
5. If endWord is found, return steps + 1.
6. If a valid word exists in wordList, add it to the queue and remove it from wordSet.
7. If endWord is unreachable, return 0

Code 1:

```
import java.util.*;
class Solution {
    public int ladderLength(String beginWord, String endWord, List<String> wordList) {
        Set<String> wordSet = new HashSet<>(wordList);
        if (!wordSet.contains(endWord)) return 0;
        Queue<String> queue = new LinkedList<>();
        queue.add(beginWord);
        int steps = 1;
        while (!queue.isEmpty()) {
            for (int i = queue.size(); i > 0; i--) {
                char[] word = queue.poll().toCharArray();
                for (int j = 0; j < word.length; j++) {
                    char original = word[j];
                    for (char c = 'a'; c <= 'z'; c++) {
                        if (c == original) continue;
                        word[j] = c;
                        String newWord = new String(word);
                        if (newWord.equals(endWord)) return steps + 1;
                        if (wordSet.remove(newWord)) queue.add(newWord);
                    }
                    word[j] = original;
                }
            }
            steps++;
        }
        return 0;
    }
}
```

Output 1:





Algorithm 2:

1. Base Case:

   If root is null, p, or q, return root.

2. Recursive Search:

   Recursively find LCA in the left subtree.

   Recursively find LCA in the right subtree.

3. Check Results:

   If both left and right subtrees return non-null, root is the LCA.

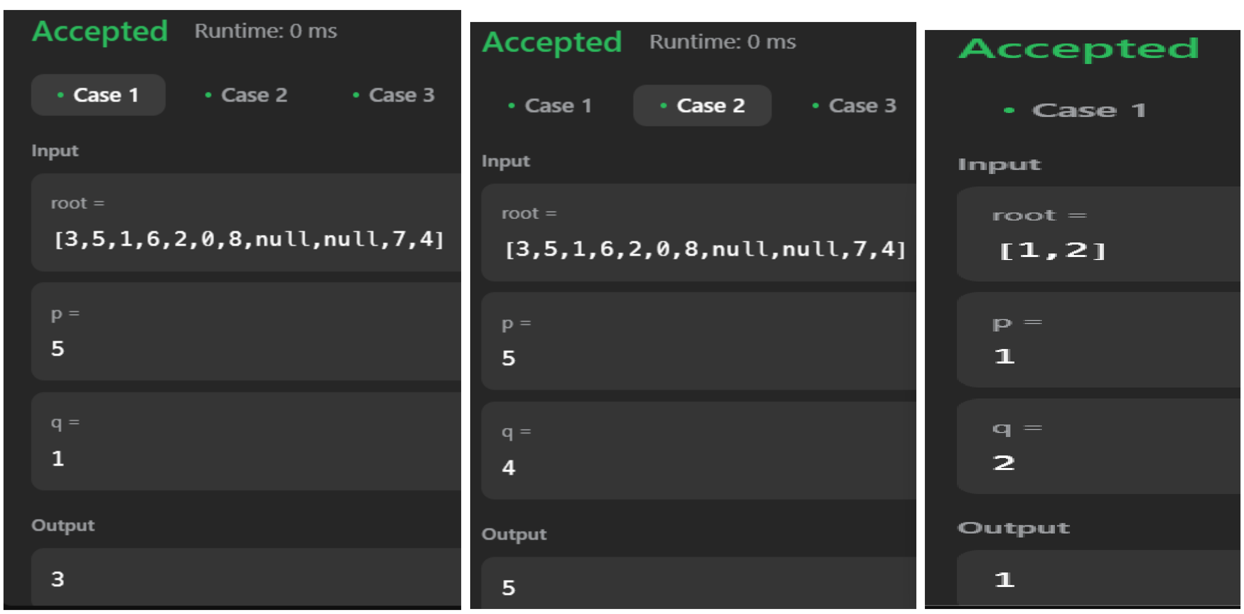
   Else, return the non-null result (either left or right).

Code 2:

```
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;
    }
}
```

Output 2:



Algorithm 3:

1. Initialize variables:
2. maxSum = nums[0] (stores the maximum sum found).
3. currentSum = nums[0] (tracks the current subarray sum).
4. Iterate through the array (starting from index 1):

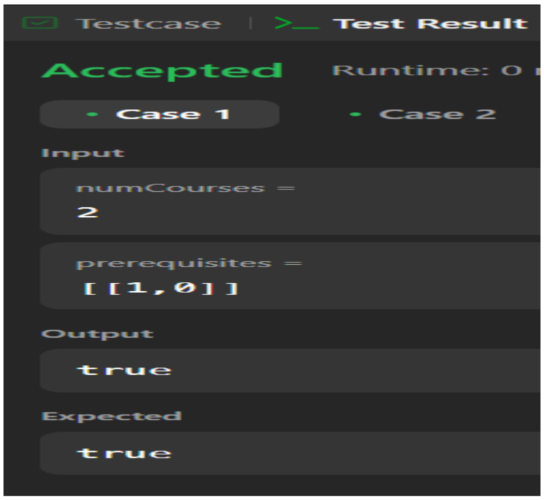
5. Update currentSum as the maximum of the current element or currentSum + current element.
6. Update maxSum if currentSum is greater.
7. Return maxSum as the maximum subarray sum

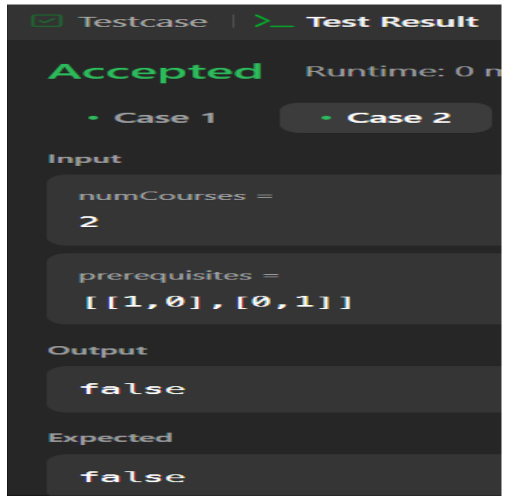
Code 3:

```
import java.util.*;
class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        List<List<Integer>> graph = new ArrayList<>();
        int[] inDegree = new int[numCourses];
        for (int i = 0; i < numCourses; i++) graph.add(new ArrayList<>());
        for (int[] pre : prerequisites) {
            graph.get(pre[1]).add(pre[0]);
            inDegree[pre[0]]++;
        }
        Queue<Integer> queue = new LinkedList<>();
        for (int i = 0; i < numCourses; i++)
            if (inDegree[i] == 0) queue.add(i);

        int count = 0;
        while (!queue.isEmpty()) {
            int course = queue.poll();
            count++;
            for (int next : graph.get(course))
                if (--inDegree[next] == 0) queue.add(next);
        }
        return count == numCourses;
    }
}
```

Output:





Learning Outcomes:

1. Apply Breadth-First Search (BFS) to find the shortest transformation sequence in Word Ladder.
2. Implement recursive traversal to find the Lowest Common Ancestor in a binary tree.
3. Use Kahn's Algorithm (BFS) for detecting cycles and determining course completion in Course Schedule.
4. Construct adjacency lists and manage dependencies in directed graphs.
5. Evaluate and optimize algorithms for efficiency in real-world applications.