Experiment-8

Student Name: Karan Goyal **UID:-22BCS15864**

Branch: BE-CSE **Group-Ntpp_IOT_602-A**

Semester: 5TH **Dateof Performance:11-4-25**

Subject Name:-Advance Programming lab-2 **Subject Code:-22CSP-351**

Problem 1

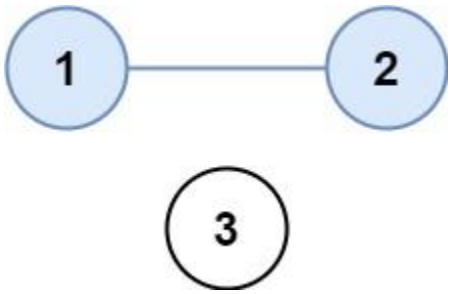
Aim:- There are n cities. Some of them are connected, while some are not. If city a is connected directly with city b, and city b is connected directly with city c, then city a is connected indirectly with city c.

A **province** is a group of directly or indirectly connected cities and no other cities outside of the group.

You are given an n x n matrix isConnected where isConnected[i][j] = 1 if the i^{th} city and the j^{th} city are directly connected, and isConnected[i][j] = 0 otherwise.

Return *the total number of* ***provinces***.

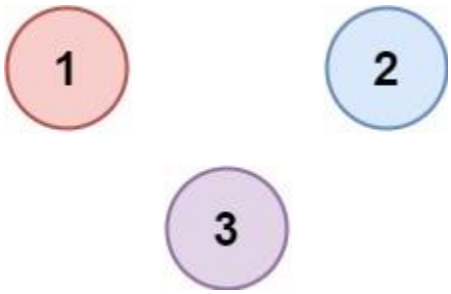
Example 1:



Input: isConnected = [[1,1,0],[1,1,0],[0,0,1]]

Output: 2

Example 2:



Input: isConnected = [[1,0,0],[0,1,0],[0,0,1]]

Output: 3

Objective:- The objective of this problem is to determine the number of provinces among n cities, given their direct connections through an adjacency matrix. A province is defined as a group of cities that are directly or indirectly connected, forming a complete connected component in the graph. The goal is to explore all such components using techniques like Depth First Search (DFS), Breadth First Search (BFS), or Union-Find, and return the count of these unique provinces based on the connectivity matrix.

Apparatus Used:

1. **Software**: -Leetcode
2. **Hardware**: Computer with 4 GB RAM and keyboard.

Algorithm: : Find Number of Provinces (DFS on Adjacency Matrix)

Input:

- isConnected – a 2D matrix of size n x n, where isConnected[i][j] = 1 indicates a direct connection between city i and city j.

Output:

- Return the total number of provinces (i.e., groups of directly or indirectly connected cities).

Steps:

1. **Initialize a visited array** of size n with all values set to 0. This keeps track of which cities have been visited during DFS.
2. **Define a recursive DFS function**:
    - Mark the current city i as visited: visited[i] = 1.
    - For every city j from 0 to n - 1:
        - If isConnected[i][j] == 1 and visited[j] == 0, then:
            - Recursively call DFS on city j.
3. **Iterate over all cities (from 0 to n-1)**:
    - If a city is not visited:
        - Call DFS starting from that city.
        - Increment the province count by 1, as this is a new connected group.
4. **After the loop ends**, return the final count of provinces.

Time Complexity:

- Let n be the number of cities.
- The outer loop runs n times.
- For each DFS call, we may visit every city once.
- The DFS visits each edge of the graph once, so the complexity is:

Time Complexity = $O(n^2)$
(because we iterate over an n x n adjacency matrix)

Space Complexity:

- visited array takes **O(n)** space.
- DFS call stack in the worst case may go up to O(n) deep (if all cities are connected in a single line).

Space Complexity = O(n)

Code:

```
class Solution {
public:
    void dfs(vector<vector<int>>& isConnected, vector<int>& visited, int i) {
        visited[i] = 1;
        for (int j = 0; j < isConnected.size(); ++j) {
            if (isConnected[i][j] == 1 && !visited[j]) {
                dfs(isConnected, visited, j);
            }
        }
```

```
    }

    int findCircleNum(vector<vector<int>>& isConnected) {
        int n = isConnected.size();
        vector<int> visited(n, 0);
        int count = 0;

        for (int i = 0; i < n; ++i) {
            if (!visited[i]) {
                dfs(isConnected, visited, i);
                count++;
            }
        }

        return count;
    }
};
```

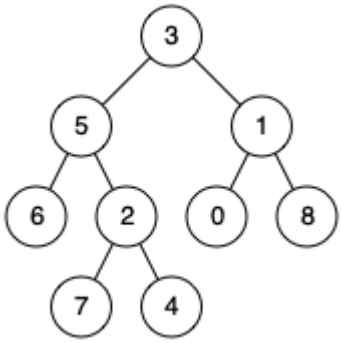
Output- All the test cases passed

Problem-2

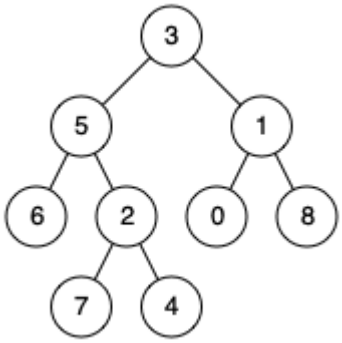
Aim:- Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**)."

Example 1:



Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
Output: 3
Explanation: The LCA of nodes 5 and 1 is 3.
Example 2:



Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
Output: 5
Explanation: The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition.
Example 3:
Input: root = [1,2], p = 1, q = 2
Output: 1

Objective- The objective of this problem is to find the Lowest Common Ancestor (LCA) of two given nodes in a binary tree. The LCA is the lowest node in the tree that has both given nodes as descendants. The task involves traversing the binary tree recursively and efficiently identifying the common ancestor by comparing node relationships. This concept is crucial in hierarchical structures and tree-based applications where node relationships and ancestry paths need to be determined accurately.

Apparatus Used:

1. **Software**: -Leetcode
2. **Hardware**: Computer with 4 GB RAM and keyboard.

Algorithm Lowest Common Ancestor (LCA) in Binary Tree

Input:

- root: The root node of the binary tree.
- p, q: The two nodes whose LCA needs to be found.

Output:

- Return the **lowest common ancestor** of nodes p and q.

Steps:

1. **Base Case**:
   If the current root is nullptr, or equal to p, or equal to q, then return root.
   - This means we've found one of the nodes or reached the end of a branch.
2. **Recursive Search**:
   - Recursively call the function on the **left subtree**:
     left = lowestCommonAncestor(root->left, p, q)
   - Recursively call the function on the **right subtree**:
     right = lowestCommonAncestor(root->right, p, q)
3. **Check Results**:
   - If both left and right are not nullptr, it means p and q are found in different subtrees. So, the current root is the **lowest common ancestor**.
   - If only one of them is not nullptr, return the non-null one.
   - If both are nullptr, return nullptr.

Time Complexity:

- Each node is visited **once**, and we do constant work at each node (checking conditions and returning values).

Time Complexity = O(n)
Where n is the number of nodes in the binary tree.

Space Complexity:

- Space is used in the **recursion call stack**.
- In the **worst case** (for a skewed tree), the recursion depth is O(n).
- In the **average case** (for a balanced tree), the depth is O(log n).

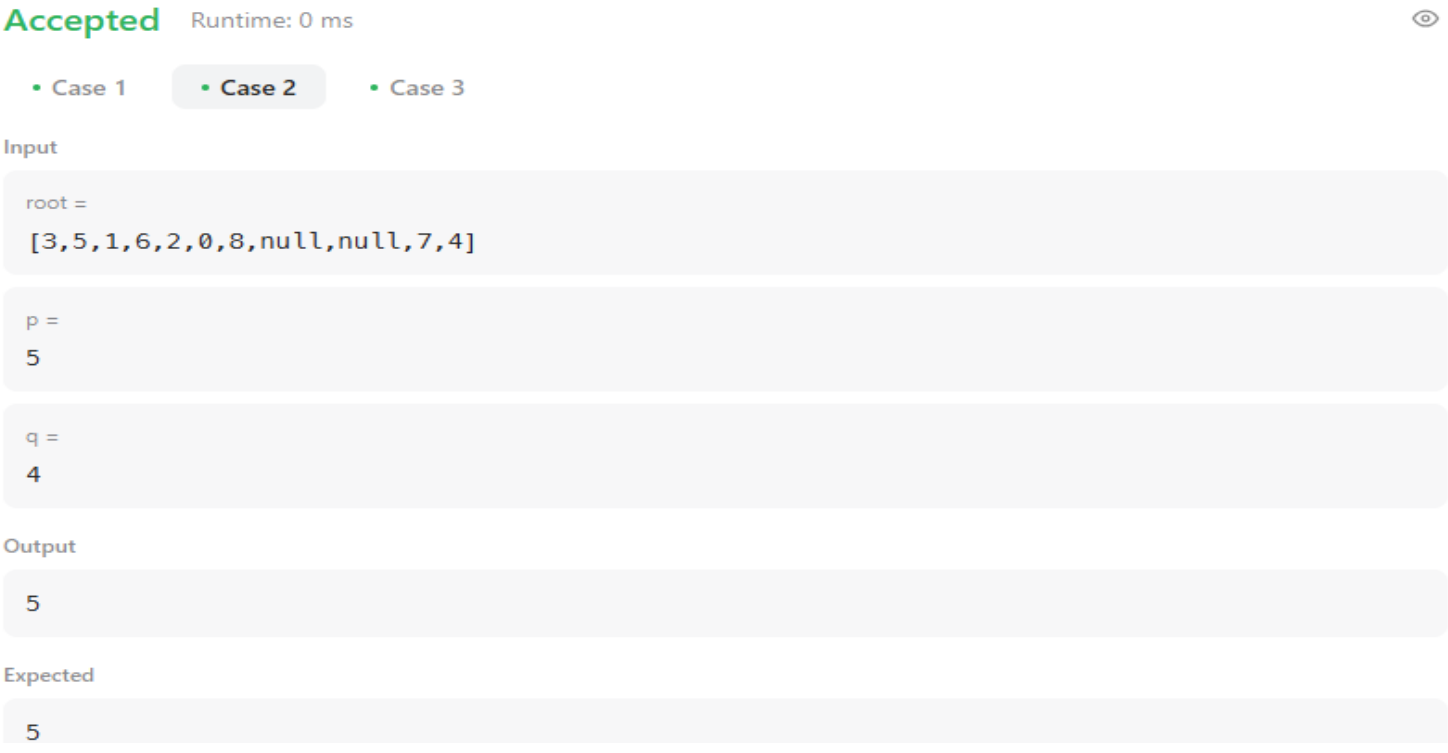
Space Complexity = O(h)
Where h is the height of the tree.

Code-

```
class Solution {
public:
  TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    if (root == nullptr || root == p || root == q) {
      return root;
    }

    TreeNode* left = lowestCommonAncestor(root->left, p, q);
```

```
        TreeNode* right = lowestCommonAncestor(root->right, p, q);

        if (left != nullptr && right != nullptr) {
            return root;
        }

        return left != nullptr ? left : right;
    }
};
```

Result-All test cases passed



Problem-3

Aim- A **transformation sequence** from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s_1 -> s_2 -> ... -> s_k such that:

- Every adjacent pair of words differs by a single letter.
- Every s_i for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.
- s_k == endWord

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

Example 1:

Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]

Output: 5

Explanation: One shortest transformation sequence is "hit" -> "hot" -> "dot" -> "dog" -> cog", which is 5 words long.

Example 2:

Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]

Output: 0

Explanation: The endWord "cog" is not in wordList, therefore there is no valid transformation sequence.

Objective- The objective of this problem is to compute the length of the shortest transformation sequence from a starting word to a target word using a given dictionary. Each transformation must change exactly one letter and result in a valid dictionary word. Using algorithms like Breadth First Search (BFS), the problem is approached as a shortest path problem in a word graph. The goal is to return the total number of words in the transformation sequence or 0 if no valid sequence exists

Apparatus Used:

1. **Software**: -Leetcode
2. **Hardware**: Computer with 4 GB RAM and keyboard.

Algorithm: Word Ladder (Shortest Transformation Sequence)

Input:

- beginWord: The starting word.
- endWord: The target word to transform into.

- wordList: A list of allowed intermediate words.

Output:

- The number of words in the **shortest transformation sequence** from beginWord to endWord.
  Return 0 if no such sequence exists.

Steps:

1. **Create a Set for Fast Lookup**:
    - Convert the wordList into an unordered_set (called dict) to allow **O(1)** lookups and deletions.
2. **Initialize a Queue for BFS**:
    - Start with a queue (todo) and push the beginWord.
    - Initialize ladder = 1 to count transformation levels (starting from the beginWord).
3. **Perform BFS Level-by-Level**:
    - While the queue is not empty:
        - Get the current level size (n) to process all words at the current ladder level.
        - For each word in this level:
            - Pop it from the queue.
            - If it matches the endWord, return the current ladder value.
            - Otherwise, erase this word from dict (to avoid revisiting).
            - For each character position in the word:
                - Try replacing it with all letters from 'a' to 'z'.
                - If the new word exists in the dict, push it to the queue.
4. **Increment ladder**:
    - After processing all words at the current level, increment ladder.
5. **Return 0 if not found**:
    - If the queue becomes empty and we didn't find the endWord, return 0.

Time Complexity:

- L = length of each word,
- N = number of words in the wordList.

For each word in the queue, we generate **26 × L** possible one-letter transformations.
Each transformed word is checked in the unordered_set in **O(1)** time.

- **Worst-case number of transformations**: Each of the N words could be visited once.
- **Total complexity** = O(N × L × 26) = **O(N × L)**

Space Complexity:

- dict stores up to N words → O(N)
- queue stores up to N words → O(N)
- Temporary strings → O(L)

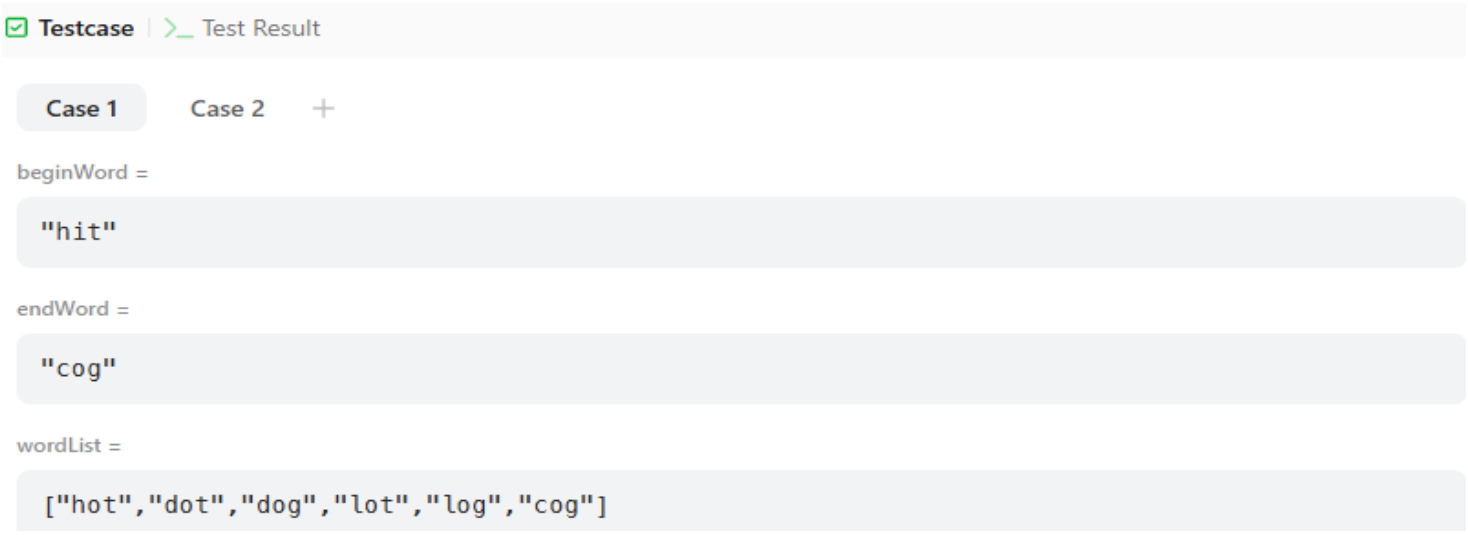
Total space complexity = O(N × L)

Code-

```
class Solution {
public:
    int ladderLength(string beginWord, string endWord, vector<string>& wordList) {
        unordered_set<string> dict(wordList.begin(), wordList.end());
        queue<string> todo;
        todo.push(beginWord);
        int ladder = 1;
        while (!todo.empty()) {
            int n = todo.size();
            for (int i = 0; i < n; i++) {
                string word = todo.front();
                todo.pop();
                if (word == endWord) {
                    return ladder; }
                dict.erase(word);
                for (int j = 0; j < word.size(); j++) {
                    char c = word[j];
                    for (int k = 0; k < 26; k++) {
                        word[j] = 'a' + k;
                        if (dict.find(word) != dict.end()) {
                            todo.push(word);   }}
```

```
            word[j] = c; }}

        ladder++;}

    return 0; }};
```

Result- All test cases passes



Learning Outcomes

1. **Understanding BFS**: Learned how Breadth-First Search (BFS) is used to find the shortest path in an unweighted graph, represented here by word transformations.
2. **Efficient Word Matching**: Gained experience in generating all possible one-letter variations of a word and checking their validity using a hash set.
3. **Optimized Data Structures**: Understood how to use unordered_set for constant-time lookups and queue for level-wise traversal in BFS.
4. **Algorithm Design**: Developed the ability to design algorithms involving string manipulation, condition checks, and dynamic updates within a loop.
5. **Complexity Analysis**: Practiced analyzing time and space complexity for graph-based word problems, helping in efficient code writing for interviews and contests.