DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
Discover. Learn. Empower.

Experiment - 5(A)

Student Name: Ankur
UID: 22BCS16091
Branch: CSE
Section/Group: NTPP_602-A
Semester: 6
Date of Performance: 17-02-25
Subject Name: Advanced Programming Lab-2
Subject Code: 22CSH-359

1. **Title:** Trees (Maximum Depth of Binary Tree)
   https://leetcode.com/problems/maximum-depth-of-binary-tree/description/

2. **Objective:** To find the maximum depth of a binary tree, which is the number of nodes along the longest path from the root node down to the farthest leaf node.

3. **Algorithm:**
   - **Base Case:** If the current node is `None`, return depth `0`.
   - **Recursive Call:**
     - Recursively calculate the maximum depth of the left subtree.
     - Recursively calculate the maximum depth of the right subtree.
   - **Calculate Maximum Depth:**
     - The depth of the current node = `1 + max(depth of left subtree, depth of right subtree)`.
   - **Return the Maximum Depth** obtained from the root node.

4. **Implementation/Code:**

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def maxDepth(self, root: TreeNode) -> int:
        if not root:
            return 0
```

```
        left_depth = self.maxDepth(root.left)
        right_depth = self.maxDepth(root.right)

        return 1 + max(left_depth, right_depth)
```

5. **Output:**

6. **Time Complexity:** O(N)

7. **Space Complexity:** O(H)

8. **Learning Outcomes:**

- **Recursion:** Understanding how to solve tree problems using recursive functions.
- **Tree Traversal:** Gaining knowledge of depth-first search (DFS) traversal.
- **Divide and Conquer:** Applying the divide and conquer strategy to break down the problem.
- **Complexity Analysis:** Learning the time and space complexity of recursive tree algorithms.

Experiment 5(B)

1. **Title:** Symmetric Tree (https://leetcode.com/problems/symmetric-tree/description/

2. **Objective:** To determine if a binary tree is **symmetric** around its center, i.e., whether the left subtree is a mirror reflection of the right subtree.

3. **Algorithm:**

Approach 1: Recursive Solution

1. If the tree is empty (root is None), it is symmetric.
2. Compare the left and right subtrees:
    - For a tree to be symmetric, the left subtree of the left child should be a mirror of the right subtree of the right child, and vice versa.
3. Recursively check the symmetry of the two subtrees:
    - Check if the values of the current nodes are equal.
    - Recursively check if the left child of the left subtree is symmetric with the right child of the right subtree, and the right child of the left subtree is symmetric with the left child of the right subtree.

Approach 2: Iterative Solution (Using a Queue)

1. Initialize a queue and start by adding the left and right children of the root.
2. While the queue is not empty:
    - Dequeue two nodes at a time.
    - Check if their values are equal.
    - Add the children in reverse order (left child of the left node, right child of the right node, and so on) to the queue.
3. Continue until all nodes are checked.

4. Implementation/Code:

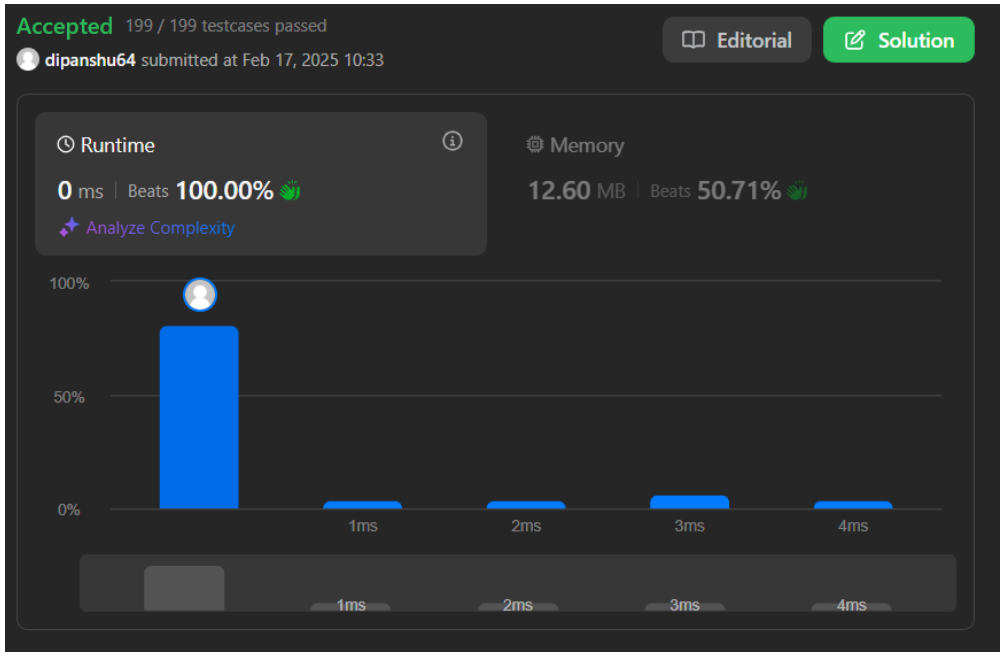
```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def isSymmetric(self, root: TreeNode):
        if not root:
            return True

        def isMirror(t1, t2):
            if not t1 and not t2:
                return True
            if not t1 or not t2:
                return False
            return (t1.val == t2.val) and isMirror(t1.left, t2.right) and
isMirror(t1.right, t2.left)

        return isMirror(root.left, root.right)
```

6. Output:



8. Time Complexity: O(N)

9. Space Complexity: O(N)

Experiment 5(C)

1. **Title:** Binary Tree Level Order Traversal (https://leetcode.com/problems/binary-tree-level-order-traversal/description/)

2. **Objective:** To perform a **Level Order Traversal** of a binary tree, where nodes are visited level by level from left to right.

3. Algorithm:

- **Base Case:** If the root is None, return an empty list.
- **Initialize Queue:**
  - Use a queue (FIFO) to store nodes for level order traversal.
  - Start by enqueuing the root node.
- **Level-wise Traversal:**
  - For each level:
  - Determine the number of nodes at the current level.
  - Dequeue each node, collect its value, and enqueue its children (left and right).
  - Store the values of the current level in a list.
- **Continue until Queue is Empty:**
  - Repeat the process for all levels.
- **Return Result:**
  - Return the list of lists containing node values level by level.

5. Implementation/Code:

```
from collections import deque

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def levelOrder(self, root: TreeNode):
        if not root:
            return []

        result = []
        queue = deque([root])  # Initialize queue with root

        while queue:
            level_size = len(queue)
            current_level = []

            for _ in range(level_size):
                node = queue.popleft()  # Dequeue front node
```

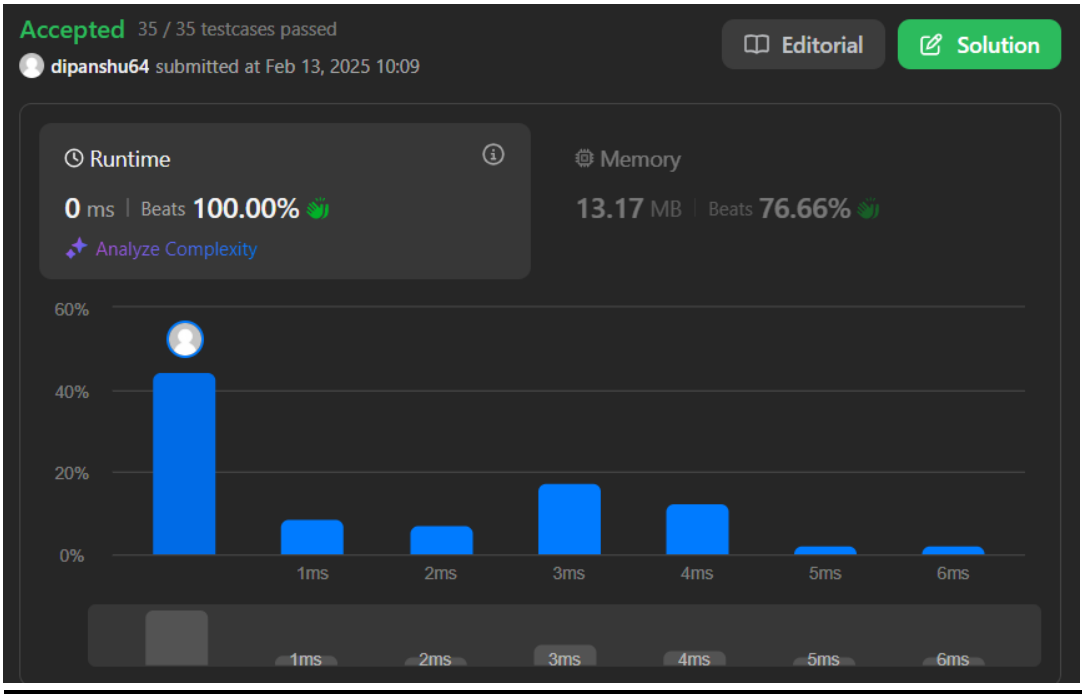
```
                current_level.append(node.val)

                # Enqueue left and right children if they exist
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)

            result.append(current_level)  # Store current level

        return result
```

6. Output:



8. Time Complexity: O(N)

9. Space Complexity: O(N)

10. Learning Outcomes:

- **Breadth-First Search (BFS):** Using a queue for BFS traversal.
- **Queue Data Structure:** Efficiently managing nodes level by level.
- **Edge Case Handling:** Properly handling empty trees.
- **Time and Space Complexity Analysis:** Understanding complexities in tree traversal.