#Practice Interview based on Assignemnt No.1 and working on Problem 2

Question Two: Path to Leaves Given the root of a binary tree, return all root to leaf paths in any order.

Examples Example 1

Input: root = [1, 2, 2, 3, 5, 6, 7] What traversal method is this?

Output: [[1, 2, 3], [1, 2, 5], [1, 2, 6], [1, 2, 7]]

Example 2

Input: root = [10, 9, 8, 7]

Output: [[10, 7], [10, 9, 8]]

Starter Code

Definition for a binary tree node.

class TreeNode(object):

def **init**(self, val = 0, left = None, right = None):

self.val = val

self.left = left

self.right = right

def bt_path(root: TreeNode) -> List[List[int]]: # TODO

1. Paraphrase the problem in your own words. Imagine we have a family tree, but instead of tracking ancestors and descendants, this tree tracks a journey from a starting point (we'll call this the "root" because it's where everything begins) down different paths to various destinations (these are the "leaves"). Our task is to list all the possible journeys from the starting point to each destination. In programming, this family tree is called a binary tree. Each point in the tree where a path splits into two is like a person in the family tree who has two children. However, in our case, these points or "nodes" have up to two paths:

one to the left and one to the right. Some paths might end quickly (meaning they have no further splits), while others might split several times before reaching the end. The question provides a binary tree in a compact form, like a list of numbers, and asks us to write a program that finds every possible path from the root to the leaves. A leaf is a node that doesn't have any children (or, in our analogy, a destination with no further paths) For example, if we are given a simple tree like a path that splits into two, and then each of those paths ends, we are supposed to list those paths. The input is given as a list representing a binary tree in a specific order. This order is a common way to represent a binary tree as a list, especially when the tree is complete or nearly complete. It's like a shorthand that tells you which number goes where in the tree, starting from the root and moving left to right, level by level. For the output, we return a list of lists, with each inner list representing one path from the start to a destination. So, each number in the inner list is a step along the path, starting with where we are and ending with a leaf.

In the context of this problem, DFS is applied to traverse the binary tree from the root to each leaf. As it progresses, it keeps track of the path taken. When it reaches a leaf (a node with no children), it records the path that led to that leaf. Then, it backtracks to explore other paths that lead to other leaves. This method ensures that all possible paths from the root to the leaves are discovered and recorded.

1. Create 1 new example that demonstrates you understand the problem. Trace/walkthrough 1 example that your partner made and explain it.

```
  5
/   \
```

3 8 / / 1 4 7 9 In this tree, the root node is 5, which has two children: 3 (left child) and 8 (right child). The node 3 has two children, 1 and 4, making them the left and right children, respectively. Similarly, the node 8 has two children: 7 (left) and 9 (right). This tree has four leaves: 1, 4, 7, and 9. Therefore, the expected paths from the root to each leaf are:

Path from root to 1: [5, 3, 1] Path from root to 4: [5, 3, 4] Path from root to 7: [5, 8, 7] Path from root to 9: [5, 8, 9]

```python
from typing import List

# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

# Function to solve the problem
def bt_path(root: TreeNode) -> List[List[int]]:
    paths = []  # To store the final paths

    def dfs(node, current_path):
        if not node:
            return

        current_path.append(node.val)  # Add the current node to the
path

        # Check if it's a leaf node
        if not node.left and not node.right:
            paths.append(list(current_path))  # Add a copy of the
current path to the final paths
        else:
            # Recursively traverse the left and right subtrees
            dfs(node.left, current_path)
            dfs(node.right, current_path)

        current_path.pop()  # Backtrack - remove the current node
while going back up

    # Start the DFS traversal
    dfs(root, [])

    return paths

# Create the new binary tree
root_new = TreeNode(5)
root_new.left = TreeNode(3)
root_new.right = TreeNode(8)
root_new.left.left = TreeNode(1)
root_new.left.right = TreeNode(4)
root_new.right.left = TreeNode(7)
root_new.right.right = TreeNode(9)

# Call the function and print the result
result_new = bt_path(root_new)
print(result_new)
```

```
[[5, 3, 1], [5, 3, 4], [5, 8, 7], [5, 8, 9]]
```

```
1
```

/ 2 2 /| | 3 5 6 7

Trace back/walkthrough example

The dfs function starts at the root (1), and since it's not a leaf, it continues to both children (2 and 2). For each child (both valued 2), it again checks for children. Starting with the left 2: It goes to 3 (left child), finds it's a leaf, records the path [1, 2, 3], and backtracks. Next, it goes to 5 (right child), finds it's a leaf, records the path [1, 2, 5], and backtracks. After finishing with the left side, the algorithm repeats the process for the right child (also valued 2): It visits 6 (left child), records the path [1, 2, 6], and backtracks. Finally, it visits 7 (right child), records the path [1, 2, 7], and backtracks. Once all paths have been recorded, the function returns the list of paths.

1. Copy the solution your partner wrote.

```python
from typing import List

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

# Function to solve the problem
def bt_path(root: TreeNode) -> List[List[int]]:
    paths = []  # To store the final paths

    def dfs(node, current_path):
        if not node:
            return

        # Add the current node to the path
        current_path.append(node.val)

        # Check if it's a leaf node
        if not node.left and not node.right:
            # Add a copy of the current path to the final paths
            paths.append(list(current_path))
        else:
            # Recursively traverse the left and right subtrees
            dfs(node.left, current_path)
            dfs(node.right, current_path)

        # Backtrack - remove the current node while going back up
        current_path.pop()

    # Start the DFS traversal
    dfs(root, [])

    return paths

# Example usage:
# Create a binary tree (you can modify this based on your actual tree
structure)
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(2)
root.left.left = TreeNode(3)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)

# Call the function and print the result
```

```python
result = bt_path(root)
print(result)
```

```
[[1, 2, 3], [1, 2, 5], [1, 2, 6], [1, 2, 7]]
```

4.Explain why their solution works in your own words.

Imagine we are in a maze, and your goal is to find all possible paths from the entrance to the exits. We have a map that shows many branching paths. Our strategy is to explore as far down one path as we can go, marking our way with a piece of chalk. When we reach a dead end (an exit in this case), we note down the path we took, then backtrack to the last junction to explore a path we haven't tried yet. This process repeats until we explored all paths and found all exits.

This is essentially what the Depth-First Search (DFS) algorithm does in binary tree problem, and here's a breakdown of why it works so well:

1. DFS Traversal: Exploring Deeply: Just like choosing a path in the maze and following it as far as possible, DFS goes as deep as it can down each branch of the tree before it hits a leaf (a node with no children, similar to a dead end or an exit in the maze).
2. Leaf Node Detection: Finding an Exit: In the maze analogy, a dead end or exit is equivalent to a leaf in the binary tree (a node without children). When DFS reaches a leaf, it means a complete path from the entrance (root of the tree) to an exit (leaf) has been found.
3. Backtracking: Returning to the Last Junction: After reaching an exit in the maze, we backtrack to the last decision point to explore untried paths. Similarly, the algorithm removes the last node it visited (backtracks) to return to the previous junction (node) and tries the other branch (the other child of the node).

Why It's Effective: The combination of deep exploration, marking paths, finding exits, and backtracking makes DFS ideal for this task. It ensures every single path is explored and recorded, just like ensuring every possible route in a maze is checked for exits

5.Explain the problem’s time and space complexity in your own words.

We consider time and space complexity as a road trip. (O(N)): In this algorithm, think of N as the total number of stops (or nodes) in trip through the tree. The "O(N)" part means that if we have 100 stops, we make 100 visits, and if we have 1000 stops, we make 1000 visits.

Space complexity is like considering how much space you need in your car for the trip, depending on how many passengers (data) we are taking and how many supplies (like food, maps, or in our case, the recursion stack and paths list) we need. In this Case (O(H)): Here, H represents the tallest point we reach on our trip, like climbing to the top of a hill or mountain. The taller the hill (or deeper the tree), the more supplies (space) we might need.

Recursion Stack: This is like a stack of maps for each part of journey. The deeper into the countryside we go (or the deeper the recursion), the more maps we need.

6.Critique your partner's solution, including explanation, if there is anything should be adjusted.

Perhaps explaining in detailed method for beginners will be helpful.

Space Complexity Explanation: While the explanation of space complexity as O(H) is accurate, it might be helpful to elaborate on how the space used by the recursion stack relates to the tree's height. Additionally, mentioning that the space complexity also depends on the number of concurrent recursive calls (which correlates with the tree's height) could provide a clearer picture.

Handling of the Paths List: The critique could highlight that the paths list's space requirement depends not just on the height of the tree but also on the total number of paths and their lengths. In the worst case (a fully balanced tree), the number of paths (and thus the space required to store them) grows exponentially with the height of the tree, which could be an important consideration for very large trees.

Reflection

Reviewing and critiquing algorithms and data structure assignments has deepened my understanding, particularly through analyzing solutions like the one employing Depth-First Search (DFS) for traversing a binary tree to uncover all root-to-leaf paths. This method exemplifies the strategic use of algorithms to navigate complex problems, with DFS ensuring comprehensive exploration and backtracking to cover all possible paths efficiently. The explanation of the algorithm's linear time complexity (O(N)) and its space complexity (O(H)) related to the tree's height, highlights the resourceful execution of the solution. Additionally, the consideration of Breadth-First Search (BFS) as an alternative approach introduces a contrasting, yet effective, level-by-level traversal method, emphasizing adaptability in solving tree structure problems. Similarly, the task of finding missing numbers from a list leverages DFS for a meticulous examination of sequences, illustrating an efficient solution that simplifies understanding of operational complexities. The solution's conversion of the list to a set to eliminate duplicates and its calculated iteration underscore a harmonious blend of accuracy and optimization in addressing the problem, showcasing the value of thoughtful algorithm selection and the significance of clear, analytical problem-solving strategies.

This approach, complemented by a straightforward iteration over a calculated range, showcases an elegant balance between thoroughness and efficiency.