129. Sum Root to Leaf Numbers

Medium Tree Depth-First Search Binary Tree Leetcode Link

Problem Description

In this problem, we have a binary tree where each node contains a digit from `0` to `9`. We need to find the total sum of all the numbers represented by all possible root-to-leaf paths in the tree. A root-to-leaf path is a sequence of nodes from the root of the tree down to a leaf node such that we concatenate the values of all the nodes along the path to form a number. For example, if a path has nodes with values `1`, `2`, and `3` in that order, the number formed is `123`. Finally, we sum all these numbers to get the final result. It is also given that the final sum will be within the range of a 32-bit integer.

To solve the problem, we traverse the tree and explore all possible paths from the root to the leaves. A binary tree does not necessarily have a balanced number of nodes, so some paths may be longer than others, and we need to make sure we include all paths in our calculation. Each path effectively represents a number in base 10, where the depth of the node determines the place value of its digit.

Intuition

The solution is based on Depth-First Search (DFS) traversal because we want to explore each path from the root to the leaves without missing any. The DFS approach allows us to accumulate the value of the number as we traverse the tree by shifting the current accumulated value one place value to the left (multiplying by 10) and then adding the value of the current node.

For implementing DFS, we perform the following steps:

1. Start at the root node (initial state).
2. As we move down to a child node, we update the current number by shifting the current number one place value to the left (by multiplying by 10) and adding the value of the current node to it.
3. If we reach a leaf node (a node without children), we add the current number to the sum since this represents a complete number from root to leaf.
4. We continue the process by exploring all paths; if a node has both left and right children, we explore each branch in separate recursive calls.
5. Finally, the sum of all the root-to-leaf numbers is returned.

This recursive implementation keeps the logic straightforward and uses the call stack to backtrace once we hit a leaf, naturally allowing us to move on to other paths in the tree.

Solution Approach

The solution uses a Depth-First Search (DFS) approach to traverse the binary tree and calculate the sum of all root-to-leaf numbers. DFS is a standard tree traversal algorithm that can be implemented recursively to traverse all the paths in a tree down to its leaves, which fits perfectly with the problem's requirement of exploring each possible root-to-leaf path.

Here's a step-by-step breakdown of the DFS algorithm as implemented in the solution:

1. Begin DFS with the root node of the binary tree and an initial current sum `s` set to `0`. The current sum `s` holds the numeric value represented by the path from the root to the current node.

2. At each node, update `s` to represent the current path's number. This is done by shifting the current accumulated value one decimal place to the left (which is equivalent to multiplying it by `10`) and then adding the node's value. For a node with value `v`, this operation is `s = s * 10 + v`.

3. If the current node is a leaf (both its left and right child are `None`), the current path represents a complete number, so return this current sum `s`.

4. To explore all paths, recursively apply the DFS to the left and right children of the current node (if they exist). Use the updated `s` for these recursive calls.

5. At each node, if the node is not a leaf, the return value is the sum of the results from the left and right recursive calls. This is because the current path's numeric value `s` should be added to the final result only once, at the leaf nodes.

6. The base case for the recursion is when the current node is `None` (a nullptr), which means we've gone past a leaf node. In this case, return `0` because there's no number to add to the sum.

7. The recursion ends when all paths have been traversed. The final return value of `dfs(root, 0)` will be the total sum of all root-to-leaf numbers.

Here is the core code for the DFS function from the solution:

```python
def dfs(root, s):
    if root is None:
        return 0
    s = s * 10 + root.val
    if root.left is None and root.right is None:
        return s
    return dfs(root.left, s) + dfs(root.right, s)
```

This recursive function (`dfs`) clearly illustrates the DFS approach. It's invoked initially with `dfs(root, 0)`, which triggers the DFS traversal from the root of the tree with an initial sum of `0`.

In summary, the DFS pattern, executed through recursive calls, provides a clear and structured way to visit every node in the tree while maintaining the state (`s`) necessary to accumulate the numerical value of the current path. As the problem requires examining all possible paths, DFS ensures that each leaf is reached, and every path's value is computed and added to produce the total sum.

Example Walkthrough

Let's consider a small binary tree as an example:

```
    1
   / \
  2   3
 / \
4   5
```

For this tree, there are three possible root-to-leaf paths:

1. `1 -> 2 -> 4`, which forms the number `124`
2. `1 -> 2 -> 5`, which forms the number `125`
3. `1 -> 3`, which forms the number `13`

Now, let's go through the solution using the Depth-First Search (DFS) approach to calculate the sum of all the numbers that are formed by root-to-leaf paths.

We start with the root node with a value of `1` and an initial sum `s` of `0`.

1. At the root node `(1)`, we update `s` to `s * 10 + 1`, which is `1`.
2. We move to the left child `(2)`. Now `s` is `1 * 10 + 2 = 12`.
   1. At node `(2)` we have two children, so we apply DFS to the left child `(4)`. Now `s` becomes `12 * 10 + 4 = 124`. Since `(4)` is a leaf node, we return `124`.
   2. We also need to consider the right child `(5)`. We apply DFS to `(5)` with `s` as `12 * 10 + 5 = 125`. Since `(5)` is also a leaf node, we return `125`.
3. The sum at node `(2)` will be the sum from its left and right children which is `124 + 125 = 249`.

Now we go back to the root `(1)` and explore the right child `(3)`.

4. At node `(3)`, `s` is updated to `1 * 10 + 3 = 13`. Since `(3)` is a leaf node, we return `13`.
5. Finally, we add the sums from both children of the root node, which are `249 (from the left subtree)` and `13 (from the right subtree)` to get the final sum which is `249 + 13 = 262`.

Thus, the sum of all root-to-leaf numbers in the tree is `262`.

Python Solution

```python
class TreeNode:
    # TreeNode class definition is provided for context.
    def __init__(self, val=0, left=None, right=None):
        self.val = val  # The value of the node.
        self.left = left  # Pointer to the left child.
        self.right = right  # Pointer to the right child.

class Solution:
    def sumNumbers(self, root: TreeNode) -> int:
        # Helper function to perform depth-first search.
        def dfs(node, num_sum):
            # If the current node is None, we've reached a leaf or the tree is empty.
            if not node:
                return 0

            # Update the sum for the current path:
            # previous sum times 10 plus the current node's value.
            num_sum = num_sum * 10 + node.val

            # If we're at a leaf node, return the current sum.
            if not node.left and not node.right:
                return num_sum

            # Continue the depth-first search on left and right subtrees, adding their sums.
            return dfs(node.left, num_sum) + dfs(node.right, num_sum)

        # Start the depth-first search with the root node and an initial sum of 0.
        return dfs(root, 0)

```

Java Solution

```java
// Definition for a binary tree node.
class TreeNode {
    int val; // Value of the node
    TreeNode left; // Left child
    TreeNode right; // Right child

    // Constructors
    TreeNode() {}
    TreeNode(int val) { this.val = val; }
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

class Solution {
    // This method starts the process of summing up all the numbers formed by the root-to-leaf paths.
    public int sumNumbers(TreeNode root) {
        return depthFirstSearch(root, 0);
    }

    // Helper method to perform depth-first search on the tree.
    // It accumulates the values from the root to each leaf, forming the numbers.
    private int depthFirstSearch(TreeNode node, int sum) {
        // If the current node is null, return 0 as there are no numbers to form here.
        if (node == null) {
            return 0;
        }

        // Update the current sum by appending the current node value.
        sum = sum * 10 + node.val;

        // If we reached a leaf, return the sum as we completed the formation of one number.
        if (node.left == null && node.right == null) {
            return sum;
        }

        // Continue the DFS traversal by visiting the left and right subtrees and summing the numbers formed.
        return depthFirstSearch(node.left, sum) + depthFirstSearch(node.right, sum);
    }
}

```

C++ Solution

```cpp
// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    // Constructor initializes current node and its children to default (no children and value 0).
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    // Constructor initializes current node with a value and sets children to default (no children).
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    // Constructor initializes current node with a value and left and right children.
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    int sumNumbers(TreeNode* root) {
        // Recursive function to traverse the tree and calculate the sum.
        // It carries two parameters: the current node and the sum calculated so far.
        std::function<int(TreeNode*, int)> depthFirstSearch = [&](TreeNode* node, int currentSum) -> int {
            // Base case: if the current node is null, return 0 as there's nothing to add.
            if (!node) return 0;
            // Update current sum by appending the current node's value.
            currentSum = currentSum * 10 + node->val;
            // If at a leaf node, return the current sum.
            if (!node->left && !node->right) return currentSum;
            // Recursively call the function for left and right children, and return their sum.
            return depthFirstSearch(node->left, currentSum) + depthFirstSearch(node->right, currentSum);
        };
        // Start the depth first search with the root node and initial sum of 0.
        return depthFirstSearch(root, 0);
    }
};

```

Typescript Solution

```typescript
// Definition for a binary tree node.
class TreeNode {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;

    constructor(val: number = 0, left: TreeNode | null = null, right: TreeNode | null = null) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

/**
 * Computes the total sum of all numbers represented by the paths from the root to the leaf nodes.
 * @param {TreeNode | null} root - The root node of the binary tree.
 * @return {number} The total sum of the numbers.
 */
function sumNumbers(root: TreeNode | null): number {
    /**
     * A helper function to perform a depth-first search on the binary tree.
     * @param {TreeNode | null} node - The current node in the traversal.
     * @param {number} currentSum - The number formed by the path from the root to the current node.
     * @return {number} The sum of all numbers formed by paths from the root to leaf nodes.
     */
    function depthFirstSearch(node: TreeNode | null, currentSum: number): number {
        // Base case: if the current node is null, return 0 as there's no number to add.
        if (node === null) {
            return 0;
        }

        // Update the current sum by adding the current node's value to it.
        // This effectively shifts the previous sum by one decimal place to the left and adds the current node's value.
        currentSum = currentSum * 10 + node.val;

        // Check if the current node is a leaf node (i.e., no left or right children).
        if (node.left === null && node.right === null) {
            // If it's a leaf node, return the current sum which represents a complete number from the root to this leaf.
            return currentSum;
        }

        // Recursively call the function for the left and right children and return the sum of both calls.
        // This effectively adds up the sums of all paths from root to the leaves.
        return depthFirstSearch(node.left, currentSum) + depthFirstSearch(node.right, currentSum);
    }

    // Initialize the depth-first search with the root node and an initial sum of 0.
    return depthFirstSearch(root, 0);
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(N)`, where `N` is the number of nodes in the binary tree. This is because each node in the tree is visited exactly once by the depth-first search (DFS) algorithm.

Space Complexity

The space complexity of the code is `O(H)`, where `H` is the height of the tree. This accounts for the maximum number of function calls on the call stack at any given time during the execution of the DFS, which is essentially the depth of the recursion. In the worst case, when the tree is skewed, the height of the tree would be `N`, making the space complexity `O(N)`. However, in a balanced tree, the space complexity would be `O(log N)`.