1022. Sum of Root To Leaf Binary Numbers

Easy Tree Depth-First Search Binary Tree

Problem Description

In this problem, we are presented with the root of a binary tree where each node contains a binary digit, either `0` or `1`. The goal is to calculate the sum of all the numbers formed by the paths from the root to each leaf node. A path represents a binary number with the most significant bit at the root and the least significant bit at a leaf.

To visualize this concept, imagine traversing down the tree from the root to a leaf. Along the way, each time we move from a parent to a child node, we append the value of the child node to the right of the current binary number we have. When we reach a leaf, we have formed a complete binary number that can be converted to its decimal equivalent. We need to perform this operation for all leaf nodes and sum their decimal values to achieve our final answer.

The complexity of the problem arises from having to efficiently traverse the tree, convert binary paths to decimal numbers, and aggregate the sum of these numbers without storing all individual path values simultaneously (since only the sum is needed and we want to manage space efficiently).

Intuition

To solve this problem, we employ Depth-First Search (DFS), a common tree traversal technique particularly useful for situations where we need to explore all possible paths from a root to its leaves.

The essence of the solution lies in maintaining an ongoing sum (denoted as `t` in the code) as we traverse the tree. At each node, we shift the current sum one bit to the left (equivalent to multiplying the current total by 2 in binary) and add the current node's value.

The DFS function recurses with the updated sum `t`, and when a leaf node is encountered, this sum represents the decimal value of the path from the root to this leaf. If the node is a leaf (i.e., it has no children), we return the current sum `t`. Otherwise, we continue DFS on any existing children. The results from these recursive calls (representing the sums of each subtree's paths) are added together to form the cumulative total for larger subtrees.

By initiating this DFS from the root with an initial sum of `0`, we ensure that all root-to-leaf paths are explored, their values are calculated in their binary to decimal form, and their sums are combined to produce the final total sum of all paths, which is returned at the end of the DFS function.

Solution Approach

The solution provided uses a recursive Depth-First Search (DFS) strategy to traverse the tree. In DFS, we visit a node and recursively go deeper into each of the subtrees before coming back to explore other branches or nodes. This is particularly efficient for our purpose here because it allows us to construct the binary number represented by the path from the root to each leaf as we traverse the tree.

The main functions and patterns used in this approach are:

- A Recursive Helper Function: `dfs` is a helper function which takes two parameters: `root`, representing the current node of the tree, and `t`, representing the ongoing path's sum in decimal form.
- Bitwise Operations:
  - Left-Shift Operator `<<`: At each step, `t` is left-shifted by one bit to make space for the next digit. In binary, this is equivalent to `t * 2`.
  - Bitwise OR Operator `|`: After shifting `t`, the current node's value is added using the bitwise OR operator, which in this case simply appends the binary digit (0 or 1) at the least significant bit's position. This is due to `0 | 0 == 0`, `0 | 1 == 1`, `1 | 0 == 1`, and `1 | 1 == 1`.

The recursive process involves:

1. Base Case: If the current node is `None`, the recursion stops and returns `0`, as there's no number to add in an empty tree.
2. Recursive Case: If the node is not empty, we calculate the ongoing sum `t` by left-shifting one bit and adding the node's value.
3. Leaf Check: If the current node is a leaf (has no left or right subtree), the current accumulated sum `t` represents a complete path's number and is returned.
4. Summation: If the current node is not a leaf, we recursively call `dfs` for both the left and right children (if they exist) and add their returned values to get the sum of the current subtree rooted at `root`.

The initial call to `dfs` starts with the root of the tree and an initial sum of `0`. As `dfs` gets called recursively, the sum `t` builds up to represent the number formed by the path from the root to the current node. When it reaches a leaf node, that number gets added to the total sum.

The solution's beauty lies in its elegant adding up of all the root-to-leaf path sums without explicitly storing each path's binary string, significantly optimizing both time and space complexity. This approach ensures that the solution is efficient and the algorithm can handle trees where the resulting sum fits within the bounds of a 32-bit integer.

Example Walkthrough

Let's take a simple example where our binary tree looks like this:

```
    1
   / \
  0   1
 / \
1   0
```

In this binary tree, there are three leaf nodes. To find the sum of all numbers formed by paths from the root to each leaf, we will calculate the binary number for each path, convert it to decimal and sum them up.

Step-by-Step Process:

1. Start at the root node (value = 1). Initialize the ongoing sum `t = 0`.
2. For the root node, calculate `t = t * 2 + node_value`. So initially, `t = 0 * 2 + 1 = 1`.
3. Move to the left child (value = 0).
   - `t = t * 2 + node_value` becomes `t = 2 * 2 + 0 = 2`.
4. Move to the left leaf (value = 1).
   - `t = t * 2 + node_value` becomes `t = 2 * 2 + 1 = 5`. This is the decimal value of the path '101', which is `5`.
   - Since it's a leaf, add `5` to the sum.
5. Move back and go to the right child of node `0` (value = 0, ignoring for now as it's non-leaf).
   - The previous path sum `t` was `2` before reaching left leaf, now `t = 2 * 2 + 0 = 4`. This is for path '100'.
   - Since it's a leaf, add `4` to the sum.
6. The total sum is now `5 + 4 = 9`. Now retreat to the root and go to the right child (value = 1).
   - At the root, `t` was `1`. Now, `t = 1 * 2 + 1 = 3`.
7. Move to the right leaf (value = 1).
   - `t = t * 2 + node_value` becomes `t = 3 * 2 + 1 = 7`. This is the decimal value of the path '11', which is `3`.
   - Since it's a leaf, add `3` to the sum.

The total sum of the paths is now `9 (sum from the left subtree) + 7 = 16`.

This is how the DFS algorithm works to compute the sum of all the numbers formed by the paths from the root to the leaves. The final answer for this example tree is `16`.

Solution Implementation

Python

```python
# 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

class Solution:
    def sumRootToLeaf(self, root: TreeNode) -> int:
        # Helper function to perform depth-first search
        def dfs(node, current_total):
            # If the current node is None, we have reached a leaf or the tree is empty
            if node is None:
                return 0
            # Add the current node's value to the running total by shifting
            # the current total left (binary left shift) and OR with node's value
            current_total = (current_total << 1) | node.val
            # Check if the current node is a leaf node
            if node.left is None and node.right is None:
                # Return the current total for this root-to-leaf path
                return current_total
            # Otherwise, recursively call dfs for left and right subtrees and sum their values
            return dfs(node.left, current_total) + dfs(node.right, current_total)

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

Java

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

    // Default constructor
    TreeNode() {}

    // Constructor with value
    TreeNode(int val) { this.val = val; }

    // Constructor with value, left, and right child
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

class Solution {
    /**
     * Public method to start the process of summing paths from root to leaves in a binary tree.
     * @param root The root node of the binary tree.
     * @return The total sum of all paths from root to leaves.
     */
    public int sumRootToLeaf(TreeNode root) {
        return depthFirstSearch(root, 0);
    }

    /**
     * Helper method to perform a depth-first search to calculate the sum of paths.
     * It accumulates the binary numbers represented by the paths from the root to leaf nodes.
     * @param node The current TreeNode being processed.
     * @param currentSum The sum accumulated from the root to the current node.
     * @return The sum of all leaf paths in the subtree rooted at the given node.
     */
    private int depthFirstSearch(TreeNode node, int currentSum) {
        // Base condition - if the current node is null, return 0.
        if (node == null) {
            return 0;
        }

        // Left-shift the current sum to make space for the current node's value,
        // and then add the current node's value using bitwise OR.
        currentSum = (currentSum << 1) | node.val;

        // If the node is a leaf node, return the current sum.
        if (node.left == null && node.right == null) {
            return currentSum;
        }

        // Recursively calculate the sum of the left and right subtrees,
        // and return the total.
        return depthFirstSearch(node.left, currentSum) + depthFirstSearch(node.right, currentSum);
    }
}
```

C++

```cpp
/**
 * Definition for a binary tree node structure.
 */
struct TreeNode {
    int val;         // Value of the node
    TreeNode *left;  // Pointer to left child
    TreeNode *right; // Pointer to right child
    // Constructor to initialize with default values
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    // Constructor to initialize with a given value and null children
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    // Constructor to initialize with a value and left and right children
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

/**
 * Solution class contains methods to solve the problem.
 */
class Solution {
public:
    // Public method that initiates the DFS traversal of the tree
    int sumRootToLeaf(TreeNode* root) {
        return depthFirstSearch(root, 0);
    }

private:
    // Helper method using Depth-First Search to find the sum of all root-to-leaf numbers
    // @param root The current node in the tree
    // @param currentNumber The sum computed so far from the root to this node
    // @return: the sum of the root-to-leaf numbers
    int depthFirstSearch(TreeNode* node, int currentSum) {
        // base case: if the current node is null, return 0
        if (!node) return 0;

        // Shift current sum left and add current node's value to it
        currentSum = (currentSum << 1) | node->val;

        // If we're at a leaf node, return the current sum
        if (!node->left && !node->right) return currentSum;

        // Recursively compute the sum for left and right children and return their sum
        return depthFirstSearch(node->left, currentSum) + depthFirstSearch(node->right, currentSum);
    }
};
```

TypeScript

```typescript
// Tree definition for a binary tree node.
type TreeNode = {
  val: number;
  left: TreeNode | null;
  right: TreeNode | null;
};

/**
 * Calculates the sum of all root-to-leaf binary numbers in a binary tree.
 * @param {TreeNode | null} root - The root node of the binary tree.
 * @returns {number} - The sum of all root-to-leaf numbers.
 */
function sumRootToLeaf(root: TreeNode | null): number {
    /**
     * Depth-first search helper function to traverse the tree and calculate binary numbers.
     * @param {TreeNode | null} node - Current node in the binary tree.
     * @param {number} currentNumber - The binary number formed from the root to the current node.
     * @returns {number} - The sum of root-to-leaf binary numbers starting from this node.
     */
    function dfs(node: TreeNode | null, currentNumber: number): number {
        if (node === null) {
            return 0;
        }

        // Update the binary number by shifting left and adding the current node's value.
        currentNumber = (currentNumber << 1) | node.val;

        // If we are at a leaf node, return the current binary number as it represents a root-to-leaf path.
        if (node.left === null && node.right === null) {
            return currentNumber;
        }

        // Recursively call the dfs on the left and right children, and return their sum.
        return dfs(node.left, currentNumber) + dfs(node.right, currentNumber);
    }

    // Initialize the sum from the root node with 0 as the initial binary number.
    return dfs(root, 0);
}
```

Python3

```python
# 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

class Solution:
    def sumRootToLeaf(self, root: TreeNode) -> int:
        # Helper function to perform depth-first search
        def dfs(node, current_total):
            # If the current node is None, we have reached a leaf or the tree is empty
            if node is None:
                return 0
            # Add the current node's value to the running total by shifting
            # the current total left (binary left shift) and OR with node's value
            current_total = (current_total << 1) | node.val
            # Check if the current node is a leaf node
            if node.left is None and node.right is None:
                # Return the current total for this root-to-leaf path
                return current_total
            # Otherwise, recursively call dfs for left and right subtrees and sum their values
            return dfs(node.left, current_total) + dfs(node.right, current_total)

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

Time and Space Complexity

The code provided is a depth-first search (DFS) algorithm on a binary tree that calculates the sum of root-to-leaf numbers represented as binary numbers.

Time Complexity

The time complexity of the DFS algorithm 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. During each visit, a constant amount of work is done: updating the current total `t`, and checking if the node is a leaf.

Therefore, the time complexity is `O(N)`.

Space Complexity

The core of the space complexity analysis lies in the stack space taken up by the depth of the recursive calls. In the worst case, for a skewed tree (where each node has only one child), the maximum depth of the recursive call stack would be O(N).

However, in the best case, where the tree is perfectly balanced, the height of the tree would be O(log N), leading to a best-case space complexity of O(log N) due to the call stack.

Thus, the space complexity of the code can be expressed as O(h) where h is the height of the tree, which ranges from O(log N) (best case for a balanced tree) to O(N) (worst case for a skewed tree).

So, the space complexity is `O(h)`.

Note: The space complexity does not consider the output since it is only an integer, which takes constant space.