1973. Count Nodes Equal to Sum of Descendants

Medium Tree Depth-First Search Binary Search Tree Binary Tree Leetcode Link

Problem Description

In this problem, we're given the root of a binary tree. Our goal is to find how many nodes in this tree have values that are equal to the sum of the values of all its descendants. A descendant of a node x is any node that lies in the subtree rooted at x, including all the leaf nodes down the path from node x. If a node has no descendants (i.e., it is a leaf node), the sum of its descendants is considered to be 0. The task is to return the count of such nodes.

To simplify, let's say we have a node with the value 10 and it has two children with values 3 and 7, respectively. The sum of the values of the descendants is 3+7=10, which equals the node's value. This node should be counted as one. We need to do this for every node in the tree.

Intuition

The approach uses a classic tree traversal technique known as Depth-First Search (DFS). The DFS allows us to reach the leaf nodes and then go up the tree, summarizing the values as we go along. This particular solution is a post-order traversal, meaning we process a node after we've processed its children.

Here's the intuition behind the approach:

1. Perform a DFS starting from the root.
2. While traversing, calculate the sum of the values of the left and right subtrees individually.
3. After obtaining the sum of the left and right subtree for a node, we check if the sum is equal to the node's value.
4. If so, we increment a counting variable, `ans`, as we've found a node where the value equals the sum of its descendants.
5. After checking the condition, we return to the parent node the sum of the node's value plus all the values from its descendants.
6. We incrementally build these sums from the child nodes up to the root, performing the checks at each step.
7. The `nonlocal` keyword used inside the `dfs` function is a way to modify the `ans` variable defined outside the nested function's scope.

The traversal ends when all nodes are visited, and the resulting `ans` variable holds the count of nodes that meet the criteria. This is both an efficient and reliable way to determine the sum of descendants for each node in the tree since with post-order traversal, we visit all the children before the parent, ensuring we have all the necessary information when the sums are compared.

Solution Approach

The solution leverages a recursive Depth-First Search (DFS) pattern to traverse the binary tree and calculate the sum of the descendants of each node while also keeping track of the count of nodes that fulfill the given criteria.

Here's a detailed breakdown of the implementation:

- **Recursive DFS Function (`dfs`)**: The `dfs` function is defined within the `equalToDescendants` method. This function is a helper function that is called recursively to perform the post-order traversal.
  - It takes a single parameter, the current `root` of the subtree being processed.
  - Base Case: If we encounter a `None` node, it signifies that we have reached past a leaf node, and we return `0` since there are no descendants to sum up.
  - Recursive Case: If the node is not `None`, we recursively call `dfs` on the left and right children of the current node.
  - We sum up the results of `dfs` calls on the left and right children, `l` and `r` respectively, to get the total sum of the descendants.
  - We compare the sum of the descendant values `l + r` with the current node's value `root.val`. If they are equal, we increment the `ans` counter. This is done using the `nonlocal` keyword to modify the `ans` variable that is defined in the outer function's scope.
  - The `dfs` function returns the sum of the node's value and its descendants' values to its parent node. This is `root.val + l + r`.
- **Recursive Traversal**: The recursion stack unwinds, visiting all nodes from the bottom-up (post-order), ensuring each node is checked precisely once for the condition. This helps to optimize the function as the calculated sums for the subtrees are used directly without recalculating them.

- **Global Counter (`ans`)**: A counter named `ans` is used to keep track of the number of nodes whose values equal the sum of their descendants' values. It is defined in the same scope as the `dfs` function, but outside of it, to be accessible and modifiable from within the nested `dfs` function.

- **Finalizing the Count**: After initiating the traversal with `dfs(root)`, the function `equalToDescendants` ultimately returns the value of `ans`, which is the total number of nodes that satisfy the condition at the end of the traversal.

The time complexity of the solution is `O(N)` where `N` is the number of nodes in the tree, as each node is visited exactly once. The space complexity is also `O(N)`, which is the worst-case space used by the call stack during the DFS (this occurs in the case of a skewed tree).

In summary, by using a post-order DFS and a simple counter, we effectively check the sum of descendants for each node against its value and return the total count of such nodes.

Example Walkthrough

Let's illustrate the solution approach with a small binary tree example. Consider the following binary tree:

```
       10
      /  \
     4    6
    / \
   3   1
```

We want to find out how many nodes have values equal to the sum of the values of their descendants.

Following the steps given in the solution approach:

1. We start the DFS from the root node which is `10`.
2. We traverse to the left child `4` and then to it's left child `3` which is a leaf node.
   - Leaf nodes are base cases, returning `0` since they have no descendants.
3. Back to node `4`, we now traverse its right child which is `1` and since it's a leaf node, it also returns `0`.
4. Now we're back at `4` with the sum of its descendant being `3 + 0 + 1 + 0 = 4`, which equals the node's value.
   - Therefore, we increment our counter `ans` by `1`.
5. We return `4` (node's value) + the sum of its descendants `4`, which is `8` to the parent node `10`.
6. We proceed to node `10`'s right child `6`, which is a leaf node.
   - Thus, it returns `0` as the sum of the descendants.
7. Now at the root node `10`, we take the sum of descendant values we got from the left `8` and right `0` child and compare it with the node `10`'s value.
   - `8 + 0` is not equal to `10`, hence `ans` remains the same.
8. The function `equalToDescendants` returns the final `ans` value which is `1`, so there is `1` node with a value equal to the sum of its descendants.

In this example, the node with the value `4` is the only node meeting the criteria.

By following this post-order DFS approach, we calculate the sum of descendant nodes' values for each node and compare it to each node's own value, incrementing our counter when they match and finally returning the count of such nodes.

Python Solution

```python
class TreeNode:
    # TreeNode class used to represent each node in a binary tree.
    def __init__(self, val=0, left=None, right=None):
        self.val = val    # Value of the node
        self.left = left  # Left child of the node
        self.right = right  # Right child of the node


class Solution:
    def equalToDescendants(self, root: Optional[TreeNode]) -> int:
        """
        This method returns the count of nodes whose value is equal to the sum of values
        of its descendants.

        :param root: The root node of the binary tree.
        :return: The count of nodes for which node's value == sum of descendants' values.
        """

        def sum_of_descendants(node):
            """
            This helper function calculates the sum of values of all descendants of the
            current node, and updates the count if the sum equals the node's value.

            :param node: The current node being processed.
            :return: The sum of values including the current node and its descendants.
            """
            if node is None:
                return 0

            # Recursively compute the sum of values of left and right subtrees
            left_sum = sum_of_descendants(node.left)
            right_sum = sum_of_descendants(node.right)

            # If sum of left and right children's values equals the current node's value, increment the count
            if left_sum + right_sum == node.val:
                nonlocal node_count
                node_count += 1

            # Return the sum of the current node's value and its descendants' values
            return node.val + left_sum + right_sum

        # Counter to hold the number of nodes meeting the specific condition.
        node_count = 0

        # Initiate depth-first search traversal from the root to calculate the sum of descendants and update the count.
        sum_of_descendants(root)

        # Return the final count of nodes whose value is equal to sum of their descendants.
        return node_count


# Additional code such as defining the TreeNode class, constructing a tree, and calling the Solution method
# would be needed here to test the functionality of the code in practice.

```

Java Solution

```java
/**
 * Definition for a binary tree node.
 */
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    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 {
    // Variable to keep track of the number of nodes equal to the sum of their descendants.
    private int numNodesEqualToDescendants;

    /**
     * Public method that initiates the depth-first search traversal of the tree
     * and returns the count of nodes where node value is equal to the sum of its descendants.
     *
     * @param root The root of the binary tree.
     * @return The count of nodes where the node value is equal to the sum of its descendants.
     */
    public int equalToDescendants(TreeNode root) {
        depthFirstSearch(root);
        return numNodesEqualToDescendants;
    }

    /**
     * Recursive helper method to perform the depth-first search and calculate the sum of the descendants.
     *
     * @param node The current node in the depth-first search traversal.
     * @return The sum of this node and its descendant nodes.
     */
    private int depthFirstSearch(TreeNode node) {
        // Base case: if the node is null, return 0 since it does not contribute to the sum.
        if (node == null) {
            return 0;
        }

        // Recursively compute the sum of the left and right subtrees.
        int leftSum = depthFirstSearch(node.left);
        int rightSum = depthFirstSearch(node.right);

        // Check if the current node's value is equal to the sum of its left and right descendants.
        if (leftSum + rightSum == node.val) {
            // Increment the count if the current node's value matches the sum of its descendants.
            numNodesEqualToDescendants++;
        }

        // Return the sum of the current node value and its descendants' values.
        return node.val + leftSum + rightSum;
    }
}

```

C++ Solution

```cpp
// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    // This function counts the number of nodes in a binary tree
    // where the node value is equal to the sum of the values of its descendants.
    int equalToDescendants(TreeNode* root) {
        int count = 0; // Initialize a counter for the nodes that meet the condition.

        // Define a Depth-First Search (DFS) lambda function that calculates the sum of a node's descendants
        // and checks if the node's value equals that sum.
        std::function<long long(TreeNode*)> dfs = [&](TreeNode* node) -> long long {
            if (!node) {
                return 0; // If the current node is null, return 0 as it doesn't contribute to the sum.
            }

            // Recursively call the dfs function on the left and right children and calculate their sums.
            long long leftSum = dfs(node->left);
            long long rightSum = dfs(node->right);

            // If the current node's value is equal to the sum of its descendants, increment the count.
            count += (leftSum + rightSum == node->val);

            // Return the sum of the current node's value and its descendants' values.
            return node->val + leftSum + rightSum;
        };

        // Start the DFS traversal from the root to count the nodes satisfying the condition.
        dfs(root);

        // Return the final count of nodes.
        return count;
    }
};

```

Typescript Solution

```typescript
// Definition for a binary tree node in TypeScript.
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;
    }
}

let count: number = 0; // Initialize a counter for the nodes that meet the condition.

function dfs(node: TreeNode | null): number {
    if (!node) {
        return 0; // If the current node is null, return 0 as it contributes nothing to the sum.
    }

    // Recursively compute the sum of descendants for the left and right subtrees.
    const leftSum = dfs(node.left);
    const rightSum = dfs(node.right);

    // If the node's value equals the sum of its descendants, increment the count.
    if (node.val === leftSum + rightSum) count += 1;

    // Return the sum of the node's value and its descendants' values.
    return node.val + leftSum + rightSum;
}

// This function counts the number of nodes in a binary tree
// where the node value is equal to the sum of the values of its descendants.
function equalToDescendants(root: TreeNode | null): number {
    count = 0; // Reset count to 0 for each call to account for different binary tree inputs.

    // Start the Depth-First Search (DFS) traversal from the root and count the nodes satisfying the condition.
    dfs(root);

    // Return the final count of nodes.
    return count;
}

```

Time and Space Complexity

The given Python code defines a method `equalToDescendants` which checks each node in a binary tree to see if its value is equal to the sum of the values of its descendant nodes. Let's analyze the time and space complexity of this code:

Time Complexity:

The time complexity of the `equalToDescendants` method is `O(n)`, where `n` is the number of nodes in the binary tree. This efficiency results from the fact that the method involves a depth-first search (DFS), visiting each node exactly once. The comparison and addition operations at each node occur in constant time, so the overall time complexity is linear concerning the number of nodes.

Space Complexity:

The space complexity of the code is `O(h)`, where `h` is the height of the tree. This space is used by the call stack during the execution of the DFS. In the worst case, when the binary tree degrades to a linked list (completely unbalanced), the space complexity will be `O(n)`. In the best case, with a completely balanced tree, the space complexity will be `O(log n)` because the height of the tree would be logarithmic relative to the number of nodes.

Therefore, the space complexity ranges from `O(log n)` to `O(n)` depending on the shape of the tree.