250. Count Univalue Subtrees

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

Problem Description

The task is to determine the number of subtrees within a binary tree where all nodes have the same value. These subtrees, where all nodes share a common value, are referred to as **uni-value subtrees**. The input is the `root` of the binary tree, and the expected output is an integer representing the total count of uni-value subtrees within that tree.

Intuition

To solve the problem, we can use recursion. A recursive function can traverse the tree while keeping track of uni-value subtrees. Here's the general idea:

1. If the current node is `None` (meaning we've reached a leaf node's child), we return `True` because a non-existent node doesn't disrupt a subtree's uni-value status.

2. We call the recursive function on the left and right child of the current node.

3. If any child subtree is not uni-value (the recursive call returns `False`), then the current subtree also cannot be uni-value. Hence, we return `False` immediately.

4. Otherwise, we need to check if the current node's value matches the values of its children (if they exist). If the current node is a leaf or if it has children with the same value as itself, then it's a root of a uni-value subtree.

5. We use a nonlocal variable `ans` to maintain the count of uni-value subtrees found during the traversal. This variable is incremented each time we find such a subtree.

6. Finally, we return `True` or `False` from the recursive function to indicate whether the subtree rooted at the current node is a uni-value subtree.

The solution leverages a depth-first search (DFS) approach, processing each node and its subtrees to determine uni-value status. By starting from the leaves and working up to the root, we effectively employ a bottom-up approach. This way, we ensure that a node is counted only when all its descendants form uni-value subtrees.

Solution Approach

The implementation of the solution involves defining a nested helper function `dfs(root)` inside the `countUnivalSubtrees` method of a `Solution` class. This recursive function is the core of the depth-first search (DFS) strategy. Below is a detailed walk-through of the code:

1. **Recursive Depth-First Search (DFS):** The `dfs` function is recursively called on both the left and right children of a node. This traversal goes down to the leaf nodes of the binary tree.

2. **Base Case:** When the `root` is `None`, we've reached beyond the leaf nodes, and in this case, the function returns `True` because a non-existent node doesn't affect the uni-value property of a subtree.

3. **Recursive Calls:** We store the results of the `dfs` calls on the left and right subtrees in the `l` and `r` variables, respectively.

4. **Early Return:** If either `l` or `r` is `False`, it means that at least one of the subtrees (left or right) isn't uni-value, and thus, the current subtree rooted at `root` can't be uni-value either. So the function returns `False` immediately.

5. **Comparison with Children's Values:** The current node's value is compared with its children's values. If there is no left or right child, the value of `root` is used for comparison to ensure that a leaf node is always considered a uni-value subtree. If both comparisons result in equality (`a == b == root.val`), then the current subtree is uni-value.

6. **Counting Uni-value Subtrees:** When a uni-value subtree is identified, we increment the `ans` variable. Here, `nonlocal ans` is used to modify the `ans` variable defined in the enclosing `countUnivalSubtrees` function, effectively keeping track of the total count.

7. **Return Value for Uni-value Subtrees:** After incrementing the `ans` variable for a uni-value subtree, the `dfs` function returns `True`.

8. **Return Value for Non-Uni-value Subtrees:** If the subtree rooted at `root` is not uni-value, then the function returns `False`.

9. **Answer Retrieval:** After calling `dfs(root)`, `countUnivalSubtrees` returns the final count of uni-value subtrees stored in `ans`.

The solution overall uses a bottom-up DFS approach to check for the uni-value property on each subtree starting from the leaves and moving to the root. A `TreeNode` class represents the nodes of the tree, encapsulating the value of the node and pointers to left and right children. This DFS allows us to ensure that every node is counted exactly once and only as part of the largest possible uni-value subtree that it's a root of.

Example Walkthrough

Let's consider a simple binary tree to illustrate the solution approach:

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

In this example binary tree, all nodes have the same value: `1`. We want to count the number of uni-value subtrees in this tree.

Here's how the solution approach would work on this tree:

1. We start the depth-first search (DFS) traversal by calling the `dfs` function on the root node (value `1`).

2. The DFS goes down to the left-most node first, which is another `1`. Since it's a leaf, the base case makes the function return `True`.

3. As the recursion unwinds, we check this node's parent, which is also `1`. Both the left and right children are either `None` or have the same value. So this subtree is also a uni-value subtree. `ans` is incremented.

4. We move up the tree, and now the parent is the root. The same logic applies: since both its children are univalues (`True` from the recursive calls), and their values match the root's value (`1`), we consider the whole tree as a uni-value subtree, and `ans` is incremented again.

5. Now, we move to the right subtree. Since we recursively find that both the right child and the left child (both with value `1`) are uni-value subtrees, they result in two more increments of `ans`.

6. Finally, we finish the traversal. As all the nodes and their subtrees have been evaluated as uni-value, the `ans` reflects the total number of uni-value subtrees. In this case, `ans` would be `5` since there are five nodes and each node itself can be considered a subtree, providing they all have the same value.

7. The `countUnivalSubtrees` function will then return `ans` as the final count of uni-value subtrees, which is `5` for our example.

Thus, by processing each node and its subtrees in a bottom-up DFS manner, we effectively count all subtrees in the binary tree where all nodes have the same value.

Python Solution

```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 countUnivalSubtrees(self, root: Optional[TreeNode]) -> int:
        # Helper function to perform depth-first search
        def is_unival_subtree(node):
            # Base case: An empty tree is a unival tree by default
            if node is None:
                return True

            # Recursively check if left and right subtrees are unival
            is_left_unival = is_unival_subtree(node.left)
            is_right_unival = is_unival_subtree(node.right)

            # If either left or right subtree is not unival, return False
            if not is_left_unival or not is_right_unival:
                return False

            # Get the value of the left child, or use the current node's value if left child is None
            left_val = node.val if node.left is None else node.left.val
            # Get the value of the right child, or use the current node's value if right child is None
            right_val = node.val if node.right is None else node.right.val

            # Check if current node is unival, which means its value equals to
            # both its children's value (or it doesn't have children)
            if left_val == right_val == node.val:
                # Increment the count as this is a unival subtree
                nonlocal total_unival_subtrees
                total_unival_subtrees += 1
                return True

            # If the current node's value does not match one or both of its children's
            # values, this subtree cannot be unival
            return False

        # Start with no unival subtrees counted
        total_unival_subtrees = 0

        # Kick off the depth-first search from the root
        is_unival_subtree(root)

        # Return the total count of unival subtrees
        return total_unival_subtrees
```

Java Solution

```java
class Solution {
    private int univalSubtreeCount;

    public int countUnivalSubtrees(TreeNode root) {
        // Performs DFS traversal to count unival subtrees
        dfs(root);
        return univalSubtreeCount;
    }

    private boolean dfs(TreeNode node) {
        // If current node is null, it is a unival subtree.
        if (node == null) {
            return true;
        }
        // Recursively check if the left subtree is unival
        boolean isLeftUnival = dfs(node.left);
        // Recursively check if the right subtree is unival
        boolean isRightUnival = dfs(node.right);

        // If either left or right subtree is not unival, return false
        if (!isLeftUnival || !isRightUnival) {
            return false;
        }

        // Capture the values of left and right children.
        // Use the current node's value if the child is null.
        int leftVal = node.left == null ? node.val : node.left.val;
        int rightVal = node.right == null ? node.val : node.right.val;

        // If the left value equals right value and also equals the current node value,
        // it's a unival subtree, increment the count.
        if (leftVal == rightVal && rightVal == node.val) {
            univalSubtreeCount++;
            return true;
        }

        // Otherwise, the subtree rooted at the current node isn't unival
        return false;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Helper method to count unival subtrees.
    int countUnivalSubtrees(TreeNode* root) {
        int ans = 0; // Counter for the number of unival subtrees.

        // Inner function to perform a depth-first search.
        std::function<bool(TreeNode*)> dfs = [&](TreeNode* node) -> bool {
            if (!node) {
                return true; // An empty tree is a unival subtree.
            }

            // Recursively check if left and right subtrees are unival.
            bool isLeftUnival = dfs(node->left);
            bool isRightUnival = dfs(node->right);

            // If either of the subtrees is not unival, the current tree can't be unival.
            if (!isLeftUnival || !isRightUnival) {
                return false;
            }

            // Check if the current node is unival with its children.
            int leftVal = node->left ? node->left->val : node->val;
            int rightVal = node->right ? node->right->val : node->val;

            if (leftVal == rightVal && rightVal == node->val) {
                // If the current node and its children have the same value, it is a unival subtree.
                ++ans;
                return true;
            }

            return false; // Current subtree is not unival as the node values differ.
        };

        dfs(root); // Start DFS from the root.
        return ans; // Return the total number of unival subtrees found.
    }
};

/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val; // Value of the node.
 *     TreeNode *left; // Pointer to the left child.
 *     TreeNode *right; // Pointer to the right child.
 *     // Constructor for a node with default value 0 and no children
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     // Constructor for a node with given value x and no children
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     // Constructor for a node with given value x and given children
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
```

Typescript Solution

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

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

// Counts the number of universal value subtrees within the binary tree.
// A subtree is considered universal if all nodes within the subtree have the same value.
function countUnivalSubtrees(root: TreeNode | null): number {
    let count: number = 0;

    // Helper function to perform depth-first search.
    // Returns true if the subtree rooted at the given node is universal.
    const isUnivalSubtree = (node: TreeNode | null): boolean => {
        if (node == null) {
            // A null node is considered a universal subtree.
            return true;
        }

        // Recursively check the left and right subtrees.
        const isLeftUnival: boolean = isUnivalSubtree(node.left);
        const isRightUnival: boolean = isUnivalSubtree(node.right);

        // If either subtree is not universal, then this cannot be a universal subtree.
        if (!isLeftUnival || !isRightUnival) {
            return false;
        }

        // If left child exists and its value is not equal to current node's value, this is not a universal subtree.
        if (node.left != null && node.left.val != node.val) {
            return false;
        }

        // If right child exists and its value is not equal to current node's value, this is not a universal subtree.
        if (node.right != null && node.right.val != node.val) {
            return false;
        }

        // Current subtree is universal; increment count and return true.
        count++;
        return true;
    };

    // Kick-off the depth-first search from the root.
    isUnivalSubtree(root);

    // Return the final count of universal subtrees.
    return count;
}
```

Time and Space Complexity

The given Python function counts the number of "unival" (universal value) subtrees within a binary tree, where a unival subtree is one that has all nodes with the same value.

Time Complexity:

To determine the time complexity, let's consider the action performed by the function and how often it's executed. The solution uses a depth-first search (DFS) strategy, exploring the tree from root to leaves. For each node, it performs a constant number of operations – checking the value of the node, comparing it with its children, and updating the answer variable if it forms a unival subtree.

The DFS traverses each node exactly once since it follows the standard recursion pattern without revisiting any node. Since there are `n` nodes in the tree, the traversal results in `O(n)` operations where `n` is the number of nodes in the binary tree.

Thus, the **Time Complexity** is `O(n)`.

Space Complexity:

The recursive solution also incurs space complexity due to the use of the recursion stack. In the worst case, where the binary tree is skewed (each parent has only one child), the recursion goes as deep as the number of nodes, leading to the maximum depth of recursion stack being `n`. Therefore, the **Space Complexity** in the worst case is `O(n)`.

However, in the best case where the tree is perfectly balanced, the height of the tree would be `log(n)`. Thus, the space complexity would be `O(log(n))`. But since worst-case scenario often dictates our space complexity analysis, we generally consider the former scenario for evaluation.

In summary, the **Space Complexity** is `O(n)` in the worst case, with the best case being `O(log(n))` if the tree is balanced.