663. Equal Tree Partition

Medium Tree Depth-First Search Binary Tree

Leetcode Link

Problem Description

The problem presents a scenario involving a binary tree, where the goal is to determine whether there is a way to split the tree into two parts by removing exactly one edge, such that the sum of the nodes' values in both parts are equal. A binary tree is defined as a tree data structure where each node has at most two children, referred to as the left child and the right child. The constraint of removing exactly one edge means that we must find a sub-tree whose value equals half the sum of all node values in the entire tree.

Intuition

To solve this problem, we first think about the properties of the tree and its partitions. Since we want to split the tree into two with equal sums, the total sum of the tree's nodes must be even; otherwise, it's mathematically impossible to divide an odd sum into two equal parts.

Knowing this, we can first calculate the sum of the entire tree. To find a partition, we then perform a tree traversal (such as a depth-first search) and calculate the sum of each subtree as we go. Each subtree sum gives us a potential partition point—if the sum of a subtree is exactly half the total sum of the tree, then removing the edge above this subtree would indeed split the tree into two trees with equal sums.

The solution has a helper function `sum` which recursively computes the sum of the node values, while simultaneously constructing a list called `seen` that records the sum of each encountered subtree. With the total sum calculated (stored in variable `s`), the code checks if this sum is odd, in which case it returns `False` as no equal partition is possible.

Finally, before checking for the possibility of partitioning, the last element of the `seen` list (which is the sum of the entire tree) is removed, as the sum of the entire tree should not be considered for partitioning. This makes sure we are only considering subtrees for partitioning. The code then simply checks if half of the total sum `s // 2` is found within the `seen` list. If it is present, it indicates that there is a subtree whose sum is half the total sum, and thus the tree can be partitioned into two trees with equal sums by removing the edge connected to that subtree.

Solution Approach

The solution leverages a depth-first search (DFS) algorithm for recursively traversing the tree. During this traversal, the sum of each subtree is computed and stored in a list. DFS is a typical choice for tree problems, as it allows the exploration of all the nodes and the computation of their aggregate values in a structured manner.

Here's a step-by-step breakdown of the implementation:

1. A helper function named `sum` is defined within the `Solution` class to handle the DFS. This function takes a node of the tree (initially the root) as its argument.
2. If the input node is `None` (indicating a leaf's child), the function returns `0`.
3. Recursively calculate the sum of the left and right children of the current node by calling the `sum` function for `root.left` and `root.right`.
4. The sum of the current subtree is the sum of the left and right subtrees plus the value of the current node (`root.val`). This value is appended to the list `seen`.
5. The helper function returns the last calculated sum (the sum of the current subtree).

When the `sum` function is first called with the root of the tree, it computes the total sum of the tree's elements and populates the `seen` list with the subtree sums, including the total sum as the last element.

The next step is to verify if the sum of the entire tree (`s`) is even. If it's not, we can already conclude that it's impossible to partition the tree into two trees with equal sums.

Then, the total sum, which is the last element in the `seen` list, is removed because we only want to check if any subtree sum equals half of the total sum.

Finally, the solution checks if half of the total sum (`s // 2`) is in the `seen` list. If it is, that means we have found a subtree sum that is exactly half of the total, and the tree can be partitioned by removing the edge that leads to this subtree.

This approach uses a recursive algorithm (DFS) to traverse the tree while using a list data structure to keep track of the sums of all subtrees encountered. The list is later used to determine if a valid partition exists.

Example Walkthrough

Let's use a small binary tree example to illustrate the solution approach. Suppose we have the following binary tree:

```
1      5
2     / \
3    10  10
4       / \
5      2   3
```

Here's the walkthrough of the solution:

1. The sum of all node values is first calculated. For our example tree, the sum is `5 + 10 + 10 + 2 + 3 = 30`. Since the total sum `30` is even, it is possible for the tree to have a valid partition.
2. We initiate the depth-first search (DFS) with the root node (`5`). The helper function `sum` begins the traversal.
3. Starting from the root:
   - The function first calculates the sum of the left subtree which is `10`. It stores this sum in the `seen` list.
   - Next, it calculates the sum of the right subtree. The recursive call to `sum` goes to node (`10`), then to its left child (`2`), and right child (`3`), combining their sums plus the node's own value (`10 + 2 + 3 = 15`). This sum is also recorded in the `seen` list.
4. The `seen` list now contains sums from the subtrees: `[10, 15]`.
5. The function `sum` returns the total sum of the tree which here is `30`, and this is initially appended to the `seen` list as well, making the `seen` list `[10, 15, 30]`.
6. Since we do not consider the sum of the entire tree as a valid partition, the last element (`30`) is removed from the `seen` list.
7. The final step is to check if the half of the total sum, which is `30 // 2 = 15`, is found in the `seen` list. In our case, `15` is indeed in the list, corresponding to the right subtree of the root.

By finding the value `15` in our `seen` list, we have determined that by removing the right edge of the root node, we can split the original tree into two trees with equal sums:

Tree 1:

```
1 5
2 /
3 10
```

Tree 2:

```
1    10
2   / \
3  2   3
```

Both trees have a sum of `15`, and thus the solution approach successfully finds a way to partition the tree into two parts with equal sums.

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 checkEqualTree(self, root: TreeNode) -> bool:
        # Helper function to calculate the sum of subtree node values
        def calculate_subtree_sum(node):
            # Base case: if the node is None, return 0
            if node is None:
                return 0
            # Recursively sum the left and the right subtrees
            left_sum = calculate_subtree_sum(node.left)
            right_sum = calculate_subtree_sum(node.right)
            # Update the cumulative sums we have seen
            subtree_sum = left_sum + right_sum + node.val
            cumulative_sums.append(subtree_sum)
            # Return the cumulative sum of the subtree rooted at the current node
            return subtree_sum

        # Initialize a list to store the cumulative sums of the subtrees
        cumulative_sums = []
        # Calculate the total sum of the tree
        total_sum = calculate_subtree_sum(root)
        # If the total sum is odd, there cannot be two equal subtree sums
        if total_sum % 2 == 1:
            return False
        # The last element in cumulative_sums is the total sum of the tree
        # which must be removed before checking for an equal partition
        cumulative_sums.pop()
        # Check if there is a subtree whose sum is half of the total sum
        half_sum = total_sum // 2
        return half_sum in cumulative_sums
```

Java Solution

```java
class Solution {
    // This list will keep track of the sums of all seen subtrees
    private List<Integer> subtreeSums;

    // Main method to check if the tree can split into two
    // with the equals sum for both parts
    public boolean checkEqualTree(TreeNode root) {
        subtreeSums = new ArrayList<>();
        // Calculate the sum of the whole tree
        int totalSum = calculateSum(root);
        // If total sum of the tree is odd, it cannot be split into equal sum part
        if (totalSum % 2 != 0) {
            return false;
        }
        // We remove the last element as it represents the
        // sum of the whole tree, which we don't want to consider as a split point
        subtreeSums.remove(subtreeSums.size() - 1);
        // Check if any subtree sum is equal to half of the total sum
        return subtreeSums.contains(totalSum / 2);
    }

    // Helper method to calculate the sum of the tree or subtree
    // and store the sums in `subtreeSums` list
    private int calculateSum(TreeNode node) {
        // If node is null, the sum is 0
        if (node == null) {
            return 0;
        }
        // Recursively calculate the sum of left and right subtrees
        int leftSum = calculateSum(node.left);
        int rightSum = calculateSum(node.right);
        // Node sum is its value plus the sum of its left and right subtrees
        int nodeSum = leftSum + rightSum + node.val;
        // Add the node sum to the list of subtree sums
        subtreeSums.add(nodeSum);
        // Return the node sum
        return nodeSum;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // This vector stores the sum of all subtrees encountered.
    vector<int> subtreeSums;

    // Function to check if a binary tree can be split into two trees with equal sum.
    bool checkEqualTree(TreeNode* root) {
        // Calculate the sum of the entire tree.
        int totalSum = sum(root);
        // If the totalSum is odd, we can't split it into two equal parts.
        if (totalSum % 2 != 0) return false;
        // Remove the sum of the entire tree as we can only split from non-root nodes.
        subtreeSums.pop_back();
        // Check if there is a subtree with sum equal to half of the totalSum.
        return count(subtreeSums.begin(), subtreeSums.end(), totalSum / 2);
    }

    // Helper function to calculate the sum of a subtree rooted at 'root'.
    int sum(TreeNode* root) {
        // Base case: if the node is null, return 0.
        if (!root) return 0;
        // Recursive case: calculate the sum of the left and right subtrees.
        int leftSum = sum(root->left);
        int rightSum = sum(root->right);
        // Calculate the sum of the subtree rooted at 'root'.
        int subtreeSum = leftSum + rightSum + root->val;
        // Store the subtree sum in the vector.
        subtreeSums.push_back(subtreeSum);
        // Return the sum of this subtree.
        return subtreeSum;
    }
};
```

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)
    }
}

// This array stores the sum of all subtrees encountered.
let subtreeSums: number[] = [];

// Function to check if a binary tree can be split into two trees with equal sum.
function checkEqualTree(root: TreeNode | null): boolean {
    // Calculate the sum of the entire tree.
    let totalSum = sum(root);
    // If the total sum is odd, it cannot be split into two equal parts.
    if (totalSum % 2 !== 0) return false;
    // Remove the sum of the entire tree as we can only split from non-root nodes.
    subtreeSums.pop();
    // Check if there is a subtree with sum equal to half of the total sum.
    // Check if there is a subtree with sum equal to half of the total sum.
    return subtreeSums.includes(totalSum / 2);
}

// Helper function to calculate the sum of a subtree rooted at 'root'.
function sum(node: TreeNode | null): number {
    // Base case: if the node is null, return 0.
    if (!node) return 0;
    // Recursive case: calculate the sum of the left and right subtrees.
    let leftSum = sum(node.left);
    let rightSum = sum(node.right);
    // Calculate the sum of the subtree rooted at 'root'.
    let subtreeSum = leftSum + rightSum + node.val;
    // Store the subtree sum in the array.
    subtreeSums.push(subtreeSum);
    // Return the sum of this subtree.
    return subtreeSum;
}
```

Time and Space Complexity

Time Complexity

The given Python function entails a depth-first traversal of the binary tree to compute the sum of the tree nodes. The function `sum` is called recursively for each node of the binary tree. Therefore, in the worst-case scenario, which is when the binary tree is either completely unbalanced or is a complete tree, the function `sum` will be called once for each node. Given that there are `n` nodes in the binary tree, the time complexity of the algorithm is $O(n)$, where n is the number of nodes in the tree. This is because each node is processed exactly once to compute its sum and store it in the `seen` list.

Space Complexity

The space complexity is determined by the storage required for the recursive function calls (the call stack) and the additional space used by the list `seen` that keeps track of the sum of each subtree. In the worst-case scenario (in a completely unbalanced tree), the maximum depth of the recursive call stack can be $O(n)$. The list `seen` will also store $n-1$ different sums (since the last total sum isn't included, as observed from `seen.pop()`), contributing $O(n)$ space complexity. Therefore, the overall space complexity of the function is $O(n)$, considering both the recursion call stack and the `seen` list together.