508. Most Frequent Subtree Sum

Medium Tree Depth-First Search Hash Table Binary Tree

Leetcode Link

Problem Description

The problem is about finding the most frequently occurring sums of all the nodes in each subtree within a binary tree. A *subtree sum* is the sum of the values of all nodes in a subtree which includes the root node of that subtree. A binary tree is a tree data structure where each node has at most two children, referred to as the left child and the right child.

Given the root node of a binary tree, the task is to compute the sum for every possible subtree and then find out which sum(s) occur most frequently. If there is more than one sum with the highest frequency, all such sums should be returned. The return format is a list of integers in any order.

An example of a subtree sum would be taking a node, adding its value to the sum of all node values in its left subtree, and then adding the sum of all node values in its right subtree.

Intuition

To solve this problem, we need to traverse the binary tree and calculate the subtree sum for each node. This is a classic case for a depth-first search (DFS) traversal, in which we go as deep as possible down one path before backing up and trying a different one.

For each node visited, we calculate the subtree sum by summing:

1. The value of the current node.
2. The subtree sum of the left child.
3. The subtree sum of the right child.

After calculating the sum, we update a histogram (counter) that records how many times each possible sum occurs. A Python `Counter` is handy for this purpose as it is designed to maintain a running count of each subtree sum encountered during the DFS.

Once the entire tree has been traversed and all subtree sums have been computed and counted, we determine the maximum frequency among them. This tells us how many times the most frequent sum occurs.

Finally, we iterate over the items in our counter to find all sums that have this maximum frequency, collecting them into a list. This list is what will be returned as the solution. Since the counter is a dictionary with subtree sum as keys and their frequencies as values, we can easily list the keys for which the values match the maximum frequency. This gives us all the most frequent subtree sums.

Solution Approach

The solution involves a combination of a depth-first search (DFS) algorithm and a `Counter` object from Python's `collections` module to keep track of subtree sum frequencies. Here's a step-by-step approach explaining the above-provided solution code:

1. Define a recursive helper function `dfs` which will be used to perform a depth-first search of the binary tree. This function takes a single argument, the `root` of the current subtree (node).
2. In the DFS function, the base case is to return a sum of `0` when a `None` node is encountered, meaning we've reached a leaf node's child.
3. The function then recursively calls itself on both the left and right children of the current node to calculate their subtree sums. These results are stored in variables `left` and `right`, respectively.
4. With the left and right subtree sums, we can now calculate the current node's total subtree sum as `s = root.val + left + right`. This is the sum of the value stored in the current node plus the sum of all node values in both subtrees.
5. We use a `Counter` to keep track of how frequently each subtree sum occurs. This `Counter` (`counter`) object is then updated with the current subtree sum, incrementing the count of `s`. This is done outside the DFS function in the global scope of the `findFrequentTreeSum` method.
6. After the DFS traversal is complete, we can infer which subtree sums are most frequent. The maximum frequency of occurrence (`mx`) is obtained by applying the `max()` function to the values of the `Counter` object.
7. The final step involves iterating over the items in the Counter and selecting only those keys (`k`) whose values (`.v`) match the maximum frequency (`mx`). This is done using a list comprehension that filters and collects all the keys with the highest frequency.
8. These keys represent the most frequent subtree sums, and they are returned as a list.

The solution nicely ties together the traversal of the binary tree to compute subtree sums and the use of a `Counter` for frequency tracking. The combination of recursive DFS and the `Counter` strategy efficiently solves the problem with clear and understandable logic.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Consider the following binary tree:

```
  5
 / \
2  -5
```

Using the provided solution approach, we'll perform these steps:

1. Call the `dfs` function on the root node (value `5`).
2. Since node `5` is not `None`, we recursively call `dfs` on its left child (value `2`). The left child is a leaf node, so it has no children. The `dfs` function returns `2`, as there are no further nodes to sum.
3. Next, we recursively call `dfs` on the right child of node `5` (value `-5`). This is also a leaf node, so the function returns `-5`.
4. We now calculate the subtree sum for the root node (node `5`). The sum `s` is equal to the root's value plus the left and right subtree sums: `s = 5 + 2 + (-5) = 2`.
5. Update the `Counter` with the subtree sum of `2` for the root node. The `Counter` now looks like this: `{2: 1}`.
6. Now we move up and track the subtree sums of the leaf nodes. The `Counter` is updated with the subtree sum `2` from the left child and `-5` from the right child, leading to: `{2: 2, -5: 1}`.
7. After finishing the DFS traversal and recording all subtree sums, we determine the most frequent sums. The maximum frequency `mx` from the `Counter` is `2` (since the subtree sum `2` occurred twice).
8. We now filter the keys in the `Counter` to find those with a frequency matching the maximum frequency `mx`. In this case, only the subtree sum `2` matches, so it will be collected in the resulting list.

The final list of most frequent subtree sums to be returned is `[2]`, as this is the sum that occurred most frequently in our example binary tree.

Python Solution

```python
from collections import Counter
from typing import List

# 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 findFrequentTreeSum(self, root: TreeNode) -> List[int]:
        """
        Find all subtree sums that occur the most frequently in the binary tree.

        :param root: TreeNode, the root of the binary tree
        :return: List[int], a list of most frequent subtree sums
        """

        def dfs(node):
            """
            Perform a depth-first search to calculate the sum of each subtree.

            :param node: TreeNode, the current node in the binary tree
            :return: int, the sum of the current subtree
            """
            if not node:
                return 0
            # Recursively find the sum of left and right subtrees
            left_sum = dfs(node.left)
            right_sum = dfs(node.right)
            # Calculate the sum of the current subtree
            subtree_sum = node.val + left_sum + right_sum
            # Increment the counter for the subtree sum
            counter[subtree_sum] += 1
            return subtree_sum

        # Initialize a Counter to keep track of the frequency of each subtree sum
        counter = Counter()
        # Start the DFS traversal from the root to fill the subtree_sum_counter
        dfs(root)
        # Find the maximum frequency among the subtree sums
        max_frequency = max(subtree_sum_counter.values())
        # Collect all subtree sum keys with the maximum frequency
        return [subtree_sum for subtree_sum, frequency in subtree_sum_counter.items() if frequency == max_frequency]
```

Java Solution

```java
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

// 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 {
    // A map to keep track of the sum occurrences.
    private Map<Integer, Integer> sumFrequencies;

    // A variable to keep track of the maximum frequency of the sum.
    private int maxFrequency;

    // Method to find the tree sums that appear most frequently.
    public int[] findFrequentTreeSum(TreeNode root) {
        sumFrequency = new HashMap<>();
        maxFrequency = Integer.MIN_VALUE;

        // Recursively find all subtree sums.
        calculateSubtreeSum(root);

        // Store the sums that have the highest frequency.
        List<Integer> frequentSums = new ArrayList<>();
        for (Map.Entry<Integer, Integer> entry : sumFrequency.entrySet()) {
            if (entry.getValue() == maxFrequency) {
                frequentSums.add(entry.getKey());
            }
        }

        // Convert the result to an array.
        int[] result = new int[frequentSums.size()];
        for (int i = 0; i < frequentSums.size(); i++) {
            result[i] = frequentSums.get(i);
        }

        return result;
    }

    // Helper method to perform a depth-first search and calculate subtree sums.
    private int calculateSubtreeSum(TreeNode node) {
        // If the node is null, return sum as 0.
        if (node == null) {
            return 0;
        }

        // Calculate the sum including the current node and its left and right subtrees.
        int sum = node.val + calculateSubtreeSum(node.left) + calculateSubtreeSum(node.right);

        // Update the frequency of the sum in the map.
        sumFrequency.put(sum, sumFrequency.getOrDefault(sum, 0) + 1);

        // Update the maximum frequency if necessary.
        maxFrequency = Math.max(maxFrequency, sumFrequency.get(sum));

        // Return the sum of the subtree rooted at the current node.
        return sum;
    }
}
```

C++ Solution

```cpp
#include <unordered_map>
#include <vector>
#include <climits> // for INT_MIN

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    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:
    // Using an unordered map to keep track of the sum frequencies
    std::unordered_map<int, int> sumFrequencies;
    // Variable to keep track of the highest frequency encountered
    int maxFrequency;

    // Constructor initializes the maximum frequency to the minimum integer value
    Solution() : maxFrequency(INT_MIN) {}

    // Function to find the most frequent subtree sums
    vector<int> findFrequentTreeSum(TreeNode *root) {
        // Reset the max frequency for each call
        maxFrequency = INT_MIN;
        // Calculate the subtree sums starting from the root
        depthFirstSearch(root);
        // Prepare the result vector
        // Iterate over the counter map and select the sums with frequency equal to maxFrequency
        for (auto& entry : sumFrequencies) {
            if (entry.second == maxFrequency) {
                result.push_back(entry.first);
            }
        }
        return result;
    }

    // Helper function to perform a depth-first search and calculate subtree sums
    int depthFirstSearch(TreeNode *node) {
        // If the node is null, return 0 as it contributes nothing to the sum
        if (!node) {
            return 0;
        }
        // Calculate the sum of the current subtree
        int sum = node->val + depthFirstSearch(node->left) + depthFirstSearch(node->right);
        // Increment the frequency count for the current sum
        ++sumFrequencies[sum];
        // Update the max frequency if the current sum's frequency is higher
        maxFrequency = std::max(maxFrequency, sumFrequencies[sum]);
        // Return the sum of this subtree to be used by its parent
        return sum;
    }
};
```

Typescript Solution

```typescript
// TypeScript function to find all subtree sums occurring with the highest frequency in a binary tree.

// TreeNode class definition
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);
    }
}

// Function to calculate the subtree sums with the highest frequency.
function findFrequentTreeSum(root: TreeNode | null): number[] {
    // Map to store subtree sums and their frequencies
    const sumFrequencyMap = new Map<number, number>();

    // Variable to keep track of the highest frequency of subtree sum
    let maxFrequency = 0;

    // Depth-first search function to explore the tree and calculate the sums
    const calculateSubtreeSum = (node: TreeNode | null): number => {
        if (node === null) {
            return 0;
        }
        const leftSum = calculateSubtreeSum(node.left);
        const rightSum = calculateSubtreeSum(node.right);
        const subtreeSum = node.val + leftSum + rightSum;

        // Update the frequency map
        const currentFrequency = (sumFrequencyMap.get(subtreeSum) ?? 0) + 1;
        sumFrequencyMap.set(subtreeSum, currentFrequency);

        // Update the max frequency if the current one is greater
        maxFrequency = Math.max(maxFrequency, currentFrequency);

        return subtreeSum;
    };

    // Start the subtree sum calculation from the root
    calculateSubtreeSum(root);

    // Array to store the most frequent subtree sums
    const mostFrequentSums = [];

    // Find all sums that have the max frequency
    for (const [sum, frequency] of sumFrequencyMap) {
        if (frequency === maxFrequency) {
            mostFrequentSums.push(sum);
        }
    }

    // Return the most frequent subtree sums
    return mostFrequentSums;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code is mainly determined by the DFS traversal of the tree and the operations performed on the Counter object that is used as a hash map.

1. **DFS Traversal**: Every node in the given binary tree is visited exactly once. If the tree has `N` nodes, the DFS traversal takes `O(N)` time.
2. **Counter Operations**:
   - **Update**: The update of the `Counter` `counter` for each subtree sum happens `N` times (once for each node). This operation can be considered `O(1)` for each update since Counter uses a hash table for storage.
   - **Max Operation**: The operation `max(counter.values())` is performed once and takes `O(N)` time in the worst case because it scans through all values in the counter.

In total, we have `O(N)` for DFS and `O(N)` for the `max` operation, which is linear. Thus, the overall time complexity is `O(N)`.

Space Complexity

The space complexity of the code is a combination of the space used by the recursion stack during the DFS traversal and the space used by the Counter.

1. **DFS Recursion Stack**: In the worst case, i.e., when the tree is completely unbalanced, the height of the tree could be `N`, leading to a recursion stack depth of `O(N)`. In the best case, i.e., when the tree is perfectly balanced, the height of the tree would be `log(N)`, leading to a recursion stack depth of `O(log(N))`. Thus, the space used by the recursion stack can vary between `O(log(N))` and `O(N)`.
2. **Counter**: The Counter object `counter` can potentially store a distinct sum for each subtree of the binary tree. In the worst case, this could be `O(N)` distinct sums if every subtree has a unique sum.

Therefore, the overall space complexity of the algorithm is `O(N)` in the worst case, which includes the space for the Counter and the recursion stack.