1214. Two Sum BSTs

Medium | Stack | Tree | Depth-First Search | Binary Search Tree | Two Pointers | Binary Search | Binary Tree | Leetcode Link

Problem Description

In this problem, we are provided with two roots of binary search trees (`root1` and `root2`) and an integer value `target`. Our goal is to determine if there is a pair of nodes, one from each tree, whose values add up to exactly the `target` value. If such a pair exists, we should return `true`; otherwise, we return `false`.

Intuition

To solve this problem, we could try every possible pair of nodes between the two trees, which would give us a solution but with a high time complexity. However, we can tackle this more efficiently by utilizing the properties of a Binary Search Tree (BST) – particularly, the fact that it is ordered.

The strategy we use here is to perform an in-order traversal on both trees, which gives us two sorted lists of values from both trees. With these sorted lists, we can use a two-pointer technique to look for a pair that adds up to the target. This method works similarly to the 'Two Sum' problem where the list is already sorted.

We initialize two indices, `i` and `j`, to the start and end of the two lists, respectively. Then, in a while loop, we check the sum of the values at these indices:

- If the sum equals the target, it means we found a valid pair and we can return `true`.
- If the sum is less than the target, we increment `i` to get a larger sum because the lists are sorted in ascending order.
- If the sum is greater than the target, we decrement `j` to get a smaller sum.

We continue this process until `i` and `j` meet the stopping criteria (either finding a pair or exhausting the search possibilities). If no valid pair is found, we return `false`.

This solution is efficient since both the in-order traversal and the two-pointer search are linear in time, giving us an O(n) complexity overall, where n is the total number of nodes in both trees.

Solution Approach

The solution approach can be broken down into several steps:

1. **In-order Traversal**: An in-order traversal of a BST visits the nodes in ascending order. The `dfs` (Depth-First Search) function recursively traverses the given tree in this manner. It starts by traversing left, then processes the current node, and finally traverses right.

2. **Accumulate Tree Values**: During the in-order traversal, each node's value is appended to a list corresponding to its tree. We maintain two lists within `nums`, where `nums[0]` is for `root1` and `nums[1]` is for `root2`.

3. **Two-pointers Technique**: After both trees have been traversed and their values are stored in two sorted lists, we use two pointers `i` and `j` to search for two numbers that add up to the target. Pointer `i` starts at the beginning of `nums[0]` (the smallest value from `root1`), and `j` starts at the end of `nums[1]` (the largest value from `root2`).

4. **Searching for the Pair**: We loop until `i` is less than the length of `nums[0]` and `j` is non-negative (`~j` is shorthand for `j != -1`). For each iteration, we calculate the sum of the elements pointed to by `i` and `j`.

   - If the sum is equal to `target`, we have found the solution and return `true`.
   - If the sum is less than `target`, we need to increase the sum. Since `nums[0]` is sorted in ascending order, we can move `i` to the right (increment `i`) to increase the sum.
   - If the sum is greater than `target`, we need to decrease the sum. We can move `j` to the left (decrement `j`) to reduce the sum since `nums[1]` is sorted in descending order by using the pointer from the end.
5. **Return Result**: If we exit the loop without finding a pair that adds up to `target`, we return `false` indicating that no such pair exists.

By using the sorted properties of the BSTs and the two-pointer method, we avoid the O(n^2) complexity of comparing every node in `root1` with every node in `root2` and instead have a much more efficient linear time solution.

Example Walkthrough

Let's assume we have two binary search trees `root1` and `root2`, and we are given a `target` value of 5. The trees are as follows:

For `root1`, imagine the tree structure:

```
    2
   / \
  1   3
```

For `root2`, the tree structure is:

```
    2
   / \
  1   4
```

According to the problem, we must find a pair of nodes, one from each tree, that adds up to the `target` value. Let's walk through the solution step by step using these example trees:

1. **In-order Traversal**: We perform in-order traversals of both trees.

   - For `root1`, the in-order traversal would give us [1, 2, 3].
   - For `root2`, it would result in [1, 2, 4].
2. **Accumulate Tree Values**: We accumulate these traversed values in two separate lists. So, we get `nums[0] = [1, 2, 3]` from `root1` and `nums[1] = [1, 2, 4]` from `root2`.

3. **Two-pointers Technique**: We place pointer `i` at the start of the first list (`nums[0]`) and `j` at the end of the second list (`nums[1]`). This means `i` points to the value `1` in the first list, and `j` points to the value `4` in the second list.

4. **Searching for the Pair**:

   - Sum the current values pointed by `i` and `j`. `nums[0][i] + nums[1][j]` gives us `1 + 4 = 5`, which is equal to the `target` value of 5. Thus, we have found a valid pair (1 from `root1` and 4 from `root2`), and we return `true`.
   - In a different scenario where the sum did not initially meet the target, we'd adjust `i` or `j` accordingly, following the rules laid out in the solution approach. In this example, however, the first pair we check gives us the required sum.
5. **Return Result**: Since we found a valid pair that adds up to the `target`, the function returns `true`. If no such pair had been found, we would continue the while loop until `i` and `j` met the stopping criteria, and if no pair was found by that point, we would return `false`.

By using the in-order traversal to get sorted lists and the two-pointer technique, we efficiently find a valid pair that sums up to the `target` without having to compare every possible pair from the two trees.

Python Solution

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def twoSumBSTs(self, root1: Optional[TreeNode], root2: Optional[TreeNode], target: int) -> bool:
        # Helper function to perform in-order traversal and store the values
        def in_order_traversal(root: Optional[TreeNode], index: int):
            if not root:
                return
            in_order_traversal(root.left, index)  # Traverse left subtree
            values[index].append(root.val)  # Store the node value
            in_order_traversal(root.right, index)  # Traverse right subtree

        # Initialize list to hold values from both trees
        values = [[], []]
        # Fill the values list with values from both trees using in-order traversal
        in_order_traversal(root1, 0)
        in_order_traversal(root2, 1)

        # Initialize pointers
        left_index, right_index = 0, len(values[1]) - 1

        # Use a two-pointer approach to find two elements that sum up to target
        while left_index < len(values[0]) and right_index >= 0:
            current_sum = values[0][left_index] + values[1][right_index]
            if current_sum == target:
                return True  # Found the elements that sum to target
            if current_sum < target:
                left_index += 1  # Move the left pointer rightward
            else:
                right_index -= 1  # Move the right pointer leftward

        # Return False if no pair is found that adds up to target
        return False

# Example usage:
# root1 = TreeNode(2, TreeNode(1), TreeNode(3))
# root2 = TreeNode(2, TreeNode(1), TreeNode(3))
# solution = Solution()
# result = solution.twoSumBSTs(root1, root2, 4)
# print(result)  # Output should be True if there are two elements from each tree that add up to 4

```

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 {
    private List<Integer>[] listValues = new List[2]; // An array of lists to store the values from both BSTs

    // Function to check if there exists two elements from both BSTs that add up to the target
    public boolean twoSumBSTs(TreeNode root1, TreeNode root2, int target) {
        // Initialize the lists in the array
        Arrays.setAll(listValues, x -> new ArrayList<>());
        // Perform in-order traversal for both trees and store values in lists
        inOrderTraversal(root1, 0);
        inOrderTraversal(root2, 1);
        // Two-pointer approach to find two numbers adding up to target
        int i = 0, j = listValues[1].size() - 1;
        while (i < listValues[0].size() && j >= 0) {
            int sum = listValues[0].get(i) + listValues[1].get(j);
            if (sum == target) {
                return true; // Found the two numbers
            } else if (sum < target) {
                ++i; // Increase the lower end
            } else {
                --j; // Decrease the upper end
            }
        }
        return false; // No two numbers found that add up to the target
    }

    // Helper function to perform in-order traversal of a BST and store the values in a list
    private void inOrderTraversal(TreeNode root, int index) {
        if (root == null) {
            return; // Base case when node is null
        }
        inOrderTraversal(root.left, index); // Traverse to the left child
        listValues[index].add(root.val); // Add current node's value
        inOrderTraversal(root.right, index); // Traverse to the right child
    }
}

```

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:
    bool twoSumBSTs(TreeNode* root1, TreeNode* root2, int target) {
        // Vectors to store the elements in each tree
        vector<int> elements[2];
        // A lambda function to perform in-order DFS traversal of a BST
        // and store the elements in the vector
        function<void(TreeNode*, int)> inOrderTraversal = [&](TreeNode* root, int index) {
            if (!root) {
                return;
            }
            inOrderTraversal(root->left, index);
            elements[index].push_back(root->val);
            inOrderTraversal(root->right, index);
        };
        // Perform the traversal for both trees
        inOrderTraversal(root1, 0);
        inOrderTraversal(root2, 1);
        // Use two pointers to find two numbers that add up to the target
        int leftIndex = 0, rightIndex = elements[1].size() - 1;
        while (leftIndex < elements[0].size() && rightIndex >= 0) {
            int sum = elements[0][leftIndex] + elements[1][rightIndex];
            if (sum == target) {
                // If the sum is equal to the target, we've found the numbers
                return true;
            }
            if (sum < target) {
                // If the sum is less than the target, move the left pointer to the right
                ++leftIndex;
            } else {
                // If the sum is greater than the target, move the right pointer to the left
                --rightIndex;
            }
        }
        // If we exit the loop, no such pair exists that adds up to the target
        return false;
    }
};

```

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;
    }
}

/**
 * Determine if there exist two elements from BSTs root1 and root2 such that their sum is equal to the target.
 * @param {TreeNode | null} root1 - The root of the first binary search tree.
 * @param {TreeNode | null} root2 - The root of the second binary search tree.
 * @param {number} target - The target sum to find.
 * @return {boolean} - Returns true if such a pair is found, otherwise false.
 */
function twoSumBSTs(root1: TreeNode | null, root2: TreeNode | null, target: number): boolean {
    // Initialize the array to hold the values from each tree
    const treeValues: number[][] = Array(2).fill(0).map(() => []);

    /**
     * Depth-first search that traverses the tree and stores its values.
     * @param {TreeNode | null} node - The current node in the traversal.
     * @param {number} treeIndex - The index representing which tree (0 or 1) is being traversed.
     */
    const depthFirstSearch = (node: TreeNode | null, treeIndex: number) => {
        if (node === null) {
            return;
        }
        depthFirstSearch(node.left, treeIndex); // Traverse left subtree
        treeValues[treeIndex].push(node.val);   // Store current node's value
        depthFirstSearch(node.right, treeIndex); // Traverse right subtree
    };

    // Perform DFS on both trees and store their values
    depthFirstSearch(root1, 0);
    depthFirstSearch(root2, 1);

    // Initialize pointers for each list
    let i = 0;
    let j = treeValues[1].length - 1;

    // Process both lists to search for two values that add up to target
    while (i < treeValues[0].length && j >= 0) {
        const currentSum = treeValues[0][i] + treeValues[1][j];
        if (currentSum === target) {
            return true; // Pair found
        }

        // Move the pointer based on the comparison of currentSum and target
        if (currentSum < target) {
            i++; // Increase sum by moving to the next larger value in the first tree
        } else {
            j--; // Decrease sum by moving to the next smaller value in the second tree
        }
    }
    return false; // No pair found that adds up to target
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is governed by the depth-first search (DFS) traversal of two binary search trees (BSTs) and the subsequent two-pointer approach used to find the two elements that sum up to the given target.

- `dfs`: The DFS function is called two times (once for each tree). As it traverses all nodes exactly once, its time complexity is `O(n)` for each tree, where `n` is the number of nodes in each tree. If we assume `m` is the number of nodes in `tree1` and `n` is the number of nodes in `tree2`, then the combined time complexity of both DFS calls is `O(m + n)`.

- Two-pointer approach: After the DFS calls, we have two sorted arrays. The while loop with the two-pointer approach runs in `O(m)` in the worst case if `m` is the size of the smaller array, because the two pointers can iterate over the entire array in a linear fashion.

The combined time complexity thus is `O(m + n)` from the DFS calls, plus `O(m)` for the two-pointer approach, which results in `O(m + n)` since we do not knwo which one is smaller, either `m` or `n` could be the size of the smaller array.

Space Complexity

The space complexity is determined by the storage of the node values in `nums` lists and the recursion stack used in DFS.

- `nums`: Two lists are used, each containing all the values from each BST. The space complexity for these lists is `O(m + n)`, where `m` and `n` are the sizes of the two trees.

- DFS recursion stack: The maximum depth of the recursion stack is bounded by the height of the trees. In the worst case with a skewed tree, the space complexity due to recursion can be `O(m)` or `O(n)` depending on which tree is taller.

If we consider that the height of the trees could be proportional to the number of nodes in the worst case (i.e., a skewed tree), the total space complexity is `O(m + n)` for the DFS recursion stack and the space needed to store the values from both trees in lists.

Combining these factors, the overall space complexity of the algorithm is `O(m + n)`.