1214. Two Sum BSTs

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 have 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 != -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(n²) 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` meet 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
1 class TreeNode:
2     def __init__(self, val=0, left=None, right=None):
3         self.val = val
4         self.left = left
5         self.right = right
6
7 class Solution:
8     def twoSumBSTs(self, root1: Optional[TreeNode], root2: Optional[TreeNode], target: int) -> bool:
9         # Helper function to perform in-order traversal and store values
10        def in_order_traversal(node: Optional[TreeNode], index: int):
11            if not root:
12                return
13            in_order_traversal(root.left, index)  # Traverse left subtree
14            values[index].append(root.val)  # Store the node value
15            in_order_traversal(root.right, index)  # Traverse right subtree
16
17        # Initialize list to hold values from both trees
18        values = [[], []]
19        # Fill the values list with values from both trees using in-order traversal
20        in_order_traversal(root1, 0)
21        in_order_traversal(root2, 1)
22
23        # Initialize pointers
24        left_index, right_index = 0, len(values[1]) - 1
25
26        # Use a two-pointer approach to find two elements that sum up to target
27        while left_index < len(values[0]) and right_index >= 0:
28            current_sum = values[0][left_index] + values[1][right_index]
29            if current_sum == target:
30                return True  # Found the elements that sum to target
31            if current_sum < target:
32                left_index += 1  # Move the left pointer rightward
33            else:
34                right_index -= 1  # Move the right pointer leftward
35
36        # Return False if no pair is found that adds up to target
37        return False
38
39 # Example usage:
40 # root1 = TreeNode(2, TreeNode(1), TreeNode(3))
41 # root2 = TreeNode(2, TreeNode(1), TreeNode(4))
42 # solution = Solution()
43 # result = solution.twoSumBSTs(root1, root2, 4)
44 # print(result)  # Output should be True if there are two elements from each tree that add up to 4
```

Java Solution

```java
1 // Definition for a binary tree node.
2 class TreeNode {
3     int val;
4     TreeNode left;
5     TreeNode right;
6     TreeNode() {}
7     TreeNode(int val) { this.val = val; }
8     TreeNode(int val, TreeNode left, TreeNode right) {
9         this.val = val;
10        this.left = left;
11        this.right = right;
12    }
13 }
14
15 class Solution {
16     private List<Integer>[] listValues = new List[2]; // An array of lists to store the values from both BSTs
17
18     // Function to check if there exists two elements from both BSTs that add up to the target
19     public boolean twoSumBSTs(TreeNode root1, TreeNode root2, int target) {
20         // Initialize the lists in the array
21         Arrays.setAll(listValues, k -> new ArrayList<>());
22         // Perform in-order traversal for both trees and store values in lists
23         inOrderTraversal(root1, 0);
24         inOrderTraversal(root2, 1);
25         // Two-pointer approach to find two numbers adding up to target
26         int i = 0, j = listValues[1].size() - 1;
27         while (i < listValues[0].size() && j >= 0) {
28             int sum = listValues[0].get(i) + listValues[1].get(j);
29             if (sum == target) {
30                 return true; // Found the two numbers
31             } else if (sum < target) {
32                 ++i; // Increase the lower end
33             } else {
34                 --j; // Decrease the upper end
35             }
36         }
37         return false; // No two numbers found that add up to the target
38     }
39
40     // Helper function to perform in-order traversal of a BST and store the values in a list
41     private void inOrderTraversal(TreeNode root, int index) {
42         if (root == null) {
43             return; // Base case when node is null
44         }
45         inOrderTraversal(root.left, index); // Traverse to the left child
46         listValues[index].add(root.val); // Add current node's value
47         inOrderTraversal(root.right, index); // Traverse to the right child
48     }
49 }
```

C++ Solution

```cpp
1 // Definition for a binary tree node.
2 struct TreeNode {
3     int val;
4     TreeNode *left;
5     TreeNode *right;
6     TreeNode() : val(0), left(nullptr), right(nullptr) {}
7     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
8     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
9 };
10
11 class Solution {
12 public:
13     bool twoSumBSTs(TreeNode* root1, TreeNode* root2, int target) {
14         // Vectors to store the elements in each tree
15         vector<int> elements[2];
16         // A lambda function to perform in-order DFS traversal of a BST
17         // and store the elements in the vector
18         function<void(TreeNode*, int)> inOrderTraverse = [&](TreeNode* root, int index) {
19             if (!root) {
20                 return;
21             }
22             inOrderTraverse(root->left, index);
23             elements[index].push_back(root->val);
24             inOrderTraverse(root->right, index);
25         };
26         // Perform the traversal for both trees
27         inOrderTraverse(root1, 0);
28         inOrderTraverse(root2, 1);
29         // Use two pointers to find two numbers that add up to the target
30         int leftIndex = 0, rightIndex = elements[1].size() - 1;
31         while (leftIndex < elements[0].size() && rightIndex >= 0) {
32             int sum = elements[0][leftIndex] + elements[1][rightIndex];
33             if (sum == target) {
34                 // If the sum is equal to the target, we've found the numbers
35                 return true;
36             } else {
37                 if (sum is less than the target, move the left pointer to the right
38                     ++leftIndex;
39                 } else {
40                 // If the sum is greater than the target, move the right pointer to the left
41                     --rightIndex;
42                 }
43             }
44         }
45         // If we exit the loop, no such pair exists that adds up to the target
46         return false;
47     }
48 };
```

Typescript Solution

```typescript
1 // Definition for a binary tree node.
2 class TreeNode {
3     val: number
4     left: TreeNode | null
5     right: TreeNode | null
6     constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
7         this.val = (val === undefined ? 0 : val);
8         this.left = (left === undefined ? null : left);
9         this.right = (right === undefined ? null : right);
10    }
11 }
12
13 /**
14  * Determine if there exist two elements from BSTs root1 and root2 such that their sum is equal to the target.
15  * @param {TreeNode | null} root1 - The root of the first binary search tree.
16  * @param {TreeNode | null} root2 - The root of the second binary search tree.
17  * @param {number} target - The target sum to find.
18  * @return {boolean} - Returns true if such a pair is found, otherwise false.
19  */
20 function twoSumBSTs(root1: TreeNode | null, root2: TreeNode | null, target: number): boolean {
21     // Initialize the array to hold the values from each tree.
22     const nodeValues: Array<Array<number>> = [[], []];
23
24     /**
25      * A depth-first search that traverses the tree and stores its values.
26      * @param {TreeNode | null} node - The current node in the traversal.
27      * @param {number} treeIndex - The index representing which tree (0 or 1) is being traversed.
28      */
29     const depthFirstSearch = (node: TreeNode | null, treeIndex: number) => {
30         if (node === null) {
31             return;
32         }
33         depthFirstSearch(node.left, treeIndex); // Traverse left subtree
34         treeValues[treeIndex].push(node.val);   // Store current node's value
35         depthFirstSearch(node.right, treeIndex); // Traverse right subtree
36     };
37
38     // Perform DFS on both trees and store their values
39     depthFirstSearch(root1, 0);
40     depthFirstSearch(root2, 1);
41
42     // Initialize pointers for each list
43     let i = 0;
44     let j = treeValues[1].length - 1;
45
46     // Process both lists to search for two values that add up to target
47     while (i < treeValues[0].length && j >= 0) {
48         const currentSum = treeValues[0][i] + treeValues[1][j];
49         if (currentSum === target) {
50             // If sum is found return true
51             return true; // Pair found
52         }
53
54         // Move the pointer based on the comparison of currentSum and target
55         if (currentSum < target) {
56             i++; // Increase sum by moving to the next larger value in the first tree
57         } else {
58             j--; // Decrease sum by moving to the next smaller value in the second tree
59         }
60     }
61     return false; // No pair found that adds up to target
62 }
```

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 square n, 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(k) in the worst case if k 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(k) for the two-pointer approach, which results in O(m + n) since we do not know 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).