Documentation in Path Sum III - Complete Documentation

1. **Problem Statement**

   Given the root of a binary tree and an integer targetSum, return the number of paths where the sum of the values along the path equals targetSum.

   - The path does not need to start or end at the root or a leaf.
   - The path must go downwards (from parent to child nodes).

   Example 1

   Input:

   root = [10,5,-3,3,2,null,11,3,-2,null,1]
   targetSum = 8

   Output: 3

   Explanation:
   The paths that sum to 8 are:

   1. 5 → 3
   2. 5 → 2 → 1
   3. -3 → 11

   Example 2

   Input:

   root = [5,4,8,11,null,13,4,7,2,null,null,5,1]
   targetSum = 22

   Output: 3

2. Intuition

The problem can be approached using DFS traversal combined with a prefix sum technique to track subpath sums efficiently.

Instead of checking all possible paths starting from every node (which would be inefficient), we maintain a running sum and use a hash map (dictionary) to store previously seen prefix sums.

By checking if (currentSum - targetSum) exists in the map, we can determine how many valid paths exist up to that node.

3. Key Observations

- The sum of any path can be computed dynamically using a prefix sum.
- The key idea is to check if (current sum - targetSum) exists in a dictionary.
- The prefix sum technique helps in achieving an O(N) time complexity.
- We must use backtracking to remove nodes from the dictionary when returning from recursion.

4. Approach

Step 1: Use DFS Traversal

- Traverse the tree using a recursive DFS.
- Maintain a cumulative sum (currentSum) from the root to the current node.

Step 2: Use a Hash Map (Dictionary)

- Store the number of times a prefix sum has occurred.
- If currentSum - targetSum exists in the dictionary, it means there exists a valid path.

Step 3: Count Paths Efficiently

- If (currentSum - targetSum) exists in the hash map, add its count to the result.
- Add the current sum to the hash map.
- Recursively call DFS on the left and right children.
- Backtrack: Remove currentSum from the hash map when returning from recursion.

5. **Edge Cases**

- Empty Tree (root = None) → Should return 0.
- Tree with Negative Values → The algorithm should correctly count paths when nodes contain negative numbers.
- Multiple Paths with the Same Sum → The hash map should correctly count multiple valid paths.
- Large Trees (Performance Test) → The solution should handle up to 1000 nodes efficiently.
- Single Node Tree → If the single node value equals targetSum, return 1.

6. **Complexity Analysis**

Time Complexity

- O(N), where N is the number of nodes in the tree.
- Each node is visited once, and hash map operations (insert, lookup) take O(1) on average.

Space Complexity

- O(N) in the worst case (for storing the prefix sums and recursive call stack).
- In a balanced tree, recursion depth is O(log N), making space complexity O(log N).
- In a skewed tree (worst case), recursion depth is O(N).

7. **Alternative Approaches**

Brute Force ($O(N^2)$)

1. Start DFS from every node.
2. Traverse all possible paths, summing values.
3. Check if the sum equals targetSum.

Drawback:

- Very slow for large trees (up to $O(N^2)$ in worst case).

Optimized Approach (Prefix Sum - O(N))

- Uses a hash map (dictionary) to store prefix sums.
- Allows constant-time lookup for checking valid paths.
- Uses backtracking to ensure correctness.

8. **Test Cases**

Basic Test Cases

```
# Test case 1
root = TreeNode(10,
    TreeNode(5, TreeNode(3, TreeNode(3), TreeNode(-2)), TreeNode(2, None, TreeNode(1))),
    TreeNode(-3, None, TreeNode(11))
)
targetSum = 8
print(Solution().pathSum(root, targetSum))  # Output: 3

# Test case 2
root = TreeNode(5,
    TreeNode(4, TreeNode(11, TreeNode(7), TreeNode(2)), None),
    TreeNode(8, TreeNode(13), TreeNode(4, TreeNode(5), TreeNode(1)))
)
targetSum = 22
print(Solution().pathSum(root, targetSum))  # Output: 3
```

Edge Cases

```
# Edge Case 1: Empty Tree
print(Solution().pathSum(None, 10))  # Output: 0

# Edge Case 2: Single Node
single_node = TreeNode(5)
print(Solution().pathSum(single_node, 5))  # Output: 1
print(Solution().pathSum(single_node, 10))  # Output: 0
```

```
# Edge Case 3: Large Tree Performance Test
large_tree = TreeNode(1)
current = large_tree
for _ in range(999):
    current.right = TreeNode(1)
    current = current.right
print(Solution().pathSum(large_tree, 10))  # Should execute efficiently
```

9. Final Thoughts

- Optimal Solution: Uses DFS + Prefix Sum Hash Map for an O(N) approach.
- Scalability: It handles large trees efficiently.
- Versatility: Works for trees with negative values and multiple valid paths.

10. Final Thoughts (Conclusion)

The Path Sum III problem is efficiently solved using a Depth-First Search (DFS) approach combined with a Prefix Sum Hash Map, which allows us to count the number of valid paths in O(N) time complexity. This optimization significantly outperforms the brute-force $O(N^2)$ approach, making it highly suitable for trees with thousands of nodes. By maintaining a running sum and leveraging a hash map to track prefix sums, we can quickly determine valid paths without repeatedly recalculating subpaths. Additionally, the use of backtracking ensures that the prefix sum dictionary remains accurate throughout the recursive traversal. This solution is not only efficient but also highly scalable, handling various edge cases such as an empty tree, a single-node tree, negative values, and multiple valid paths. The prefix sum technique applied here is a powerful tool that can be extended to other problems in tree traversal and subarray sum computations. Beyond this specific problem, this method finds applications in numerous scenarios where tracking cumulative sums efficiently is required. Ultimately, this approach strikes a perfect balance between efficiency, correctness, and scalability, making it a highly effective solution for optimally solving tree-based path sum problems. 🚀