437. Path Sum III **Depth-First Search** Medium Tree **Binary Tree**

Problem Description

are: A path is a sequence of nodes where each subsequent node is a direct child of the previous node.

The problem involves finding the number of paths in a binary tree that sum up to a given target value. The key aspects to remember

- Unlike some other path problems, the paths in this problem do not need to begin at the root or end at a leaf. This means paths can start and end anywhere in the tree as long as they go downwards.
- The goal is to count all such unique paths that have a sum equal to the targetSum.
- One of the main challenges is to consider all possible paths efficiently, including those that start and end in the middle of the tree.

Intuition

The intuition behind the solution involves a depth-first traversal of the tree while keeping track of the running sum of node values for

each path traversed. Here's how we arrive at the solution:

1. We use a recursive depth-first search (dfs) method to explore all paths. Each time we visit a node, we add it to our current

- path's running sum. 2. To handle paths that start in the middle of the tree, we utilize a prefix sum technique. This involves keeping a count of the
- number of times each sum has occurred so far in the current path using a Counter (dictionary), which is updated as we go deeper into the tree.
- 3. For each node visited, we check if there's a previous sum that is exactly current_sum targetSum. If such a sum exists, it means there's a subpath that sums to targetSum, and we increment our ans counter accordingly.
- reflects sums for the current path. 5. By calling the dfs function starting from the root and an initial sum of 0, we traverse the entire tree, and our answer is aggregated through the ans counter.

4. As we backtrack (after visiting a node's children), we decrement the count of the sum in our Counter to ensure that it only

- The use of the prefix sum and Counter is what allows us to efficiently keep track of the number of valid paths without having to explicitly store every path's values.
- The solution provided implements a <u>Depth-First Search</u> (DFS) algorithm using recursion. It uses a combination of techniques including the traversal method and the prefix sum count, which are integrated into a single pass through the binary tree. Here's a

1. **Definition**: A helper function dfs(node, s) is defined, where node is the current node in the binary tree and s is the running sum of values from the root to this node.

step-by-step explanation of the approach:

dictionary to reflect this sum has been reached.

Solution Approach

2. Base Case: If node is None, meaning we have reached past a leaf, the function returns 0. 3. Running Sum Update: Add the current node's value to the running sum (i.e., s += node.val).

a Counter of prefix sums. If it does, that indicates there are paths that, when removed from the current path, yield a path that sums to targetSum (i.e., ans = cnt[s - targetSum]). 5. Updating the Prefix Sum Counter: Before continuing to child nodes, increment the count for the current sum in the cnt

4. Prefix Sum Calculation: Check if the current running sum (s), minus the targetSum, exists as a key in the cnt dictionary, which is

- 6. Recursive DFS Calls: Perform DFS on the left and right children, if they exist, and add their results to our ans (i.e., ans += dfs(node.left, s) and ans += dfs(node.right, s)). This counts all paths from the child nodes downwards that sum to
- targetSum. 7. Backtracking: Once we have explored the current node's descendants, we decrement the prefix sum count (cnt[s] -= 1). This is to prevent the current path's sum from affecting other paths as we backtrack up the tree.
- from the root with an initial sum of 0. By using this approach, each possible path in the tree is considered exactly once, and we incrementally build up our understanding

of how many times each prefix sum has been seen without needing to store entire paths or revisit nodes. The algorithm scales

reasonably well as it visits each node only once, and prefix sum lookup and update operations are typically O(1) on average.

9. Initialization and Invocation: In the pathSum function, we initialize cnt = Counter({0: 1}). This is important because it accounts

for the path that consists only of the node itself if the node's value equals targetSum. Then, we invoke dfs(root, 0) starting

8. Returning the Result: The dfs function ultimately returns the count of all valid paths found (ans).

Let's use a small binary tree example to illustrate the solution approach. Consider the following binary tree where the target sum is 8:

2. At the root node (5), update running sum to s = 0 + 5 = 5. Since s - targetSum = 5 - 8 = -3 is not in cnt, no paths ending

1. Start the dfs with the root node (value 5) and an initial running sum s of 0. We initialize the counter dictionary cnt with one entry {0: 1} to account for any one-node path that might equal the targetSum.

Example Walkthrough

here sum to 8. Update cnt to {0: 1, 5: 1} and proceed. 3. Move to the left child (4), and s becomes 9. s - targetSum = 9 - 8 = 1 is not in cnt, so no paths ending here sum to 8 either.

not form a path sum of 8. Proceed similarly with the right child (node 1).

Increment the current path's sum with the current node's value

path_count = path_counts[current_sum - target_sum]

Recursively find paths in left and right subtrees

Store the current path's sum in the counter

Initialize a counter to keep track of all path sums

path_count += dfs(node.left, current_sum)

Number of times the (current_sum - target_sum) has occurred so far

which indicates a valid path when subtracted from the current_sum

Let's walk through the dfs method applied to this tree:

Update cnt to {0: 1, 5: 1, 9: 1}.

Definition for a binary tree node.

self.val = val

self.left = left

self.right = right

def __init__(self, val=0, left=None, right=None):

current_sum += node.val

path_counts[current_sum] += 1

Return the number of paths found

return path_count

class TreeNode:

16

17

18

19

20

21

22

23

24

25

26

27

34

35

36

37

27

28

29

30

31

32

34

35

36

37

38

40

43

44

45

46

47

48

49

50

51

52

39 }

class TreeNode {

int val;

TreeNode left;

TreeNode() {}

TreeNode right;

this.val = val;

1, 5: 1, 9: 1, 20: 1}.

5. Now move to the left child of 11, which is 7, and s becomes 27. There's no prefix sum that makes s - targetSum equal to an existing key in cnt. Update cnt to {0: 1, 5: 1, 9: 1, 20: 1, 27: 1}.

4. Move to the left child of 4, which is node 11, and update s to 20. s - targetSum = 20 - 8 = 12 is not in cnt. Update cnt to {0:

27 and 22). 7. Now, we go down the right subtree following a similar process, until we reach the right child of 8, which is the node 4 on the

6. Backtrack to 11 and then to its right child 2, making s equal to 22. Checking for s - targetSum gives us 22 - 8 = 14, which is not

present in cnt. Update cnt and move up the tree as there are no more children, updating cnt at each step (removing counts for

key in our cnt with a value of 1 ({0: 1, 5: 1, 9: 1, ...}). This means we found a path sum (from 5 to 4 on the right) to 8. Increment our answer counter by 1.

8. Explore the left child of this 4 (node 5), and we get a running sum of 17 + 5 = 22. Now, 22 - 8 = 14 is not in cnt; thus, this does

By using this recursive approach, we traverse each path in the tree, update a running sum, check against our cnt to find valid paths,

and backtrack accordingly. Once the whole tree is traversed, we would have counted all unique paths that sum to the given target

right side. When we reach node 4, our running sum s is 5 + 8 + 4 = 17. Checking our cnt, we see that 17 - 8 = 9, and 9 is a

- sum of 8. Python Solution 1 from collections import Counter
- class Solution: 11 def pathSum(self, root: Optional[TreeNode], target_sum: int) -> int: # Helper function to perform depth-first search 12 13 def dfs(node, current_sum): if node is None: 14 15 return 0

```
28
                path_count += dfs(node.right, current_sum)
29
30
               # Once the node is done, remove its sum from the counter
               # to not use it in the parallel subtree calls
31
32
                path_counts[current_sum] -= 1
33
```

38 path_counts = Counter({0: 1}) 39 # Call the dfs function from the root of the tree return dfs(root, 0) 42 Java Solution 1 class Solution { // A map to store the cumulative sum up to all the ancestors of a node and their respective counts. private Map<Long, Integer> cumulativeSumCount = new HashMap<>(); // The target sum to find in the path. private int targetSum; // Public function to call the private dfs function and initialize the cumulativeSumCount. public int pathSum(TreeNode root, int targetSum) { 8 // Initialize the map with zero cumulative sum having a count of one. 9 cumulativeSumCount.put(0L, 1); 10 // Store the target sum in the global variable to use in the dfs function. 11 this.targetSum = targetSum; // Start the DFS traversal. 13 return dfs(root, 0); 14 15 16 // A private function to perform the DFS traversal and find paths with sums equal to targetSum. 17 private int dfs(TreeNode node, long currentSum) { 18 // Base case: If the current node is null, return 0 as there are no paths through this node. 19 20 if (node == null) { 21 return 0; 22 23 // Add the current node's value to the cumulative sum. 24 currentSum += node.val; 25 // Find the number of paths that end at this node with a sum equal to targetSum. 26 int pathCount = cumulativeSumCount.getOrDefault(currentSum - targetSum, 0);

// Update the map with the new cumulative sum, incrementing its count by 1.

// After the children have been processed, decrement the count of the currentSum

cumulativeSumCount.merge(currentSum, 1, Integer::sum);

cumulativeSumCount.merge(currentSum, -1, Integer::sum);

// path count because it should not be counted in other paths.

// Return the total count of valid paths found from this node.

// Recursively call dfs for the left child.

// Recursively call dfs for the right child.

pathCount += dfs(node.left, currentSum);

pathCount += dfs(node.right, currentSum);

return pathCount;

// Definition for a binary tree node.

TreeNode(int val) { this.val = val; }

TreeNode(int val, TreeNode left, TreeNode right) {

```
this.left = left;
53
54
           this.right = right;
55
56 }
57
C++ Solution
  1 #include <unordered map>
  2 #include <functional>
    // Definition for a binary tree node.
  5 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) {}
 11
 12 };
 13
 14 class Solution {
     public:
 16
         int pathSum(TreeNode* root, int targetSum) {
 17
             // Counter to store the prefix sum frequencies
 18
             std::unordered_map<long long, int> prefixSumCounter;
 19
             // Initialize the counter with sum 0 and frequency 1
             prefixSumCounter[0] = 1;
 20
 21
 22
             // Recursive lambda function that performs DFS traversal
 23
             std::function<int(TreeNode*, long long)> dfs = [&](TreeNode* node, long long currentSum) -> int {
 24
                 if (!node) return 0; // Base case: if the node is null, return 0
 25
 26
                 currentSum += node->val; // Update the current sum by adding the node's value
                 // Determine the number of paths with sum (currentSum - targetSum)
 27
 28
                 int numPaths = prefixSumCounter[currentSum - targetSum];
 29
                 // Increment the count of paths equal to the current sum
 30
                 ++prefixSumCounter[currentSum];
 31
                 // Recur on the left and right subtrees to find more paths and add to numPaths
 32
                 numPaths += dfs(node->left, currentSum) + dfs(node->right, currentSum);
 33
                 // Decrement the count back as we backtrack, ensuring the current
 34
                 // path's sum does not affect other paths
 35
                 --prefixSumCounter[currentSum];
 36
 37
                 return numPaths; // Return the total number of valid paths found
             };
 38
 39
             // Kickstart the DFS from the root with an initial sum of `0`
 40
 41
             return dfs(root, 0);
 42
 43
    };
 44
Typescript Solution
```

26 27 // Get the number of times we have seen the currentSum - targetSum. let pathCount = prefixSumCount.get(currentSum - targetSum) || 0; 28 29 30 // Update the count of the current sum in the map. 31 prefixSumCount.set(currentSum, (prefixSumCount.get(currentSum) | | 0) + 1);

1 // Represents a node in the binary tree.

this.val = val === undefined ? 0 : val;

this.left = left === undefined ? null : left;

// Map to keep the cumulative sum and its frequency.

const prefixSumCount: Map<number, number> = new Map();

this.right = right === undefined ? null : right;

// Function to calculate the number of paths that sum to a given value.

// Helper function to perform DFS on the tree and calculate paths.

const dfs = (node: TreeNode | null, currentSum: number): number => {

// After returning from the recursion, decrement the frequency of the current sum.

prefixSumCount.set(currentSum, (prefixSumCount.get(currentSum) | | 0) - 1);

function pathSum(root: TreeNode | null, targetSum: number): number {

// Update the current sum by adding the node's value.

constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {

left: TreeNode | null;

if (!node) {

return 0;

return pathCount;

prefixSumCount.set(0, 1);

dictionary and calculating the ans.

return dfs(root, 0);

currentSum += node.val;

// Explore left and right subtrees.

pathCount += dfs(node.left, currentSum);

pathCount += dfs(node.right, currentSum);

// Return the total count of paths found.

// Initialize the map with base case before the recursion.

// Start DFS from the root node with an initial sum of 0.

work done is proportional to the number of nodes, which is O(N).

right: TreeNode | null;

2 class TreeNode {

6

8

9

10

11

13

16

17

18

19

20

21

22

23

24

25

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

};

12 }

val: number;

```
Time and Space Complexity
Time Complexity
The time complexity of the given DFS (Depth-First Search) algorithm is O(N), where N is the number of nodes in the binary tree.
Here's why:
```

The algorithm visits each node exactly once. At each node, it does a constant amount of work, primarily updating the cnt

Space Complexity

The recursion stack could go as deep as the height of the tree, but since the function visits all N nodes once, the total amount of

- The space complexity of the algorithm consists of two parts: the recursion stack space and the space used by the cnt dictionary. Recursion Stack Space: In the worst-case scenario, the binary tree could be skewed (e.g., a linked-list shaped tree), which would mean that the recursion depth could be N. Therefore, the space complexity for recursion would be O(N).
- cnt Dictionary Space: The cnt dictionary can contain at most one entry for every prefix sum encountered during the DFS traversal. In the worst case, the number of unique prefix sums could be O(N).

Combining both, the overall space complexity of the algorithm is O(N), where N is the number of nodes in the tree. This takes into account the space for the recursion stack and the space for the cnt dictionary.