Great question 👍 You’ve pasted a **prefix-sum based solution** to the *Path Sum III* type problem. Let’s break it step by step so it’s crystal clear.

**🔎 Problem in Simple Words**

We want to count **how many paths** in a binary tree sum to target\_sum.

* A path can start and end anywhere (but must go downward).
* We don’t just check root-to-leaf — it can be *any subpath*.

**🧠 Idea Behind the Code**

Instead of checking all possible paths (which is expensive), the solution uses **prefix sums + a hashmap** to track how many ways a certain sum has appeared.

**📌 Step-by-Step Walkthrough of the Code**

**1. Function structure**

def count\_paths(self, root, target\_sum):

map = {} # stores prefix sums seen so far

return self.count\_paths\_prefix\_sum(root, target\_sum, map, 0)

* map keeps track of **prefix sums** → "how many times have we seen this sum so far?"
* We start recursion with an initial sum of 0.

**2. Recursive helper**

def count\_paths\_prefix\_sum(self, current\_node, target\_sum, map, current\_path\_sum):

if current\_node is None:

return 0

* If node is None, no paths here, return 0.

**3. Update running sum**

current\_path\_sum += current\_node.val

* This is the **prefix sum** (sum of all nodes from root down to current\_node).

**4. Direct root-to-current path match**

if target\_sum == current\_path\_sum:

path\_count += 1

* If the whole path (root → current) equals target, count it.

**5. Subpath check with hashmap**

path\_count += map.get(current\_path\_sum - target\_sum, 0)

* Suppose current\_path\_sum = 18 and target\_sum = 8.
* If at some ancestor we had a prefix sum = 10, then the path between that ancestor and here is 18 - 10 = 8.
* So we just check map for (current\_path\_sum - target\_sum).

**6. Store prefix sum**

map[current\_path\_sum] = map.get(current\_path\_sum, 0) + 1

* Record that we’ve now seen this current\_path\_sum.

**7. DFS recursion**

path\_count += self.count\_paths\_prefix\_sum(current\_node.left, target\_sum, map, current\_path\_sum)

path\_count += self.count\_paths\_prefix\_sum(current\_node.right, target\_sum, map, current\_path\_sum)

* Explore both children with updated sums.

**8. Backtracking**

map[current\_path\_sum] = map.get(current\_path\_sum, 1) - 1

* When recursion returns (backtracking), we remove the current node’s sum from the map.
* Otherwise, it would wrongly affect sibling branches.

**9. Return total count**

return path\_count

**✅ Example**

Tree:

10

/ \

5 -3

/ \ \

3 2 11

/ \ \

3 -2 1

Target = 8

Paths:

* [5, 3]
* [5, 2, 1]
* [-3, 11]
* [8] (if single node existed with value 8)

This code will correctly count **all such paths**, not just root-to-leaf.

**🚀 Key Insight**

* Instead of recomputing sums for every possible subpath (O(N²)),
* We store **prefix sums in a hashmap** → check subpaths in **O(1)**.
* Overall complexity = **O(N)**.

👉 Do you want me to also **draw a visual diagram** (prefix sums on each node and how map updates) for this algorithm? That would make the logic super intuitive.

Perfect 👌 Let’s walk through a **visual diagram** using the classic example tree.

We’ll solve for **target\_sum = 8**.

**🌳 Tree Example**

10

/ \

5 -3

/ \ \

3 2 11

/ \ \

3 -2 1

**📊 Visualizing Prefix Sum Updates**

We’ll keep two things:

1. **Current Path Sum** → sum of values from root to current node
2. **Map** → {prefix\_sum: count} showing how many times we’ve seen each sum

**Step 1: At Node(10)**

* current\_path\_sum = 10
* target = 8 → no direct match
* Map update: {10: 1}

**Step 2: Go Left → Node(5)**

* current\_path\_sum = 10 + 5 = 15
* Check map[15 - 8] = map[7] → not found
* Map update: {10: 1, 15: 1}

**Step 3: Go Left → Node(3)**

* current\_path\_sum = 15 + 3 = 18
* Check map[18 - 8] = map[10] = 1 ✅ → Path found [5, 3]
* Map update: {10: 1, 15: 1, 18: 1}

**Step 4: Go Left → Node(3)**

* current\_path\_sum = 18 + 3 = 21
* Check map[21 - 8] = map[13] → not found
* Map update: {10: 1, 15: 1, 18: 1, 21: 1}

**Step 5: Backtrack (remove 21)**

* Map: {10: 1, 15: 1, 18: 1}

**Step 6: Right child of Node(3) → Node(-2)**

* current\_path\_sum = 18 + (-2) = 16
* Check map[16 - 8] = map[8] → not found
* Map update: {10: 1, 15: 1, 18: 1, 16: 1}

**Step 7: Backtrack**

* Remove 16
* Map: {10: 1, 15: 1, 18: 1}

**Step 8: Right child of Node(5) → Node(2)**

* current\_path\_sum = 15 + 2 = 17
* Check map[17 - 8] = map[9] → not found
* Map update: {10: 1, 15: 1, 18: 1, 17: 1}

**Step 9: Node(1)**

* current\_path\_sum = 17 + 1 = 18
* Check map[18 - 8] = map[10] = 1 ✅ → Path [5, 2, 1]
* Map update: {10: 1, 15: 1, 18: 2, 17: 1}

**Step 10: Backtrack up to root and go Right → Node(-3)**

* current\_path\_sum = 10 + (-3) = 7
* Check map[7 - 8] = map[-1] → not found
* Map update: {10: 1, 7: 1}

**Step 11: Node(11)**

* current\_path\_sum = 7 + 11 = 18
* Check map[18 - 8] = map[10] = 1 ✅ → Path [-3, 11]
* Map update: {10: 1, 7: 1, 18: 1}

**✅ Paths Found**

1. [5, 3]
2. [5, 2, 1]
3. [-3, 11]

📌 **Diagram (Prefix Sum Tracking)**

Map along traversal:

At Node 10 → {10:1}

At Node 5 → {10:1, 15:1}

At Node 3 → {10:1, 15:1, 18:1} ✅ path found

At Node 3 → {10:1, 15:1, 18:1, 21:1}

Backtrack → remove 21

At Node -2 → {10:1, 15:1, 18:1, 16:1}

Backtrack → remove 16

At Node 2 → {10:1, 15:1, 18:1, 17:1}

At Node 1 → {10:1, 15:1, 18:2, 17:1} ✅ path found

Backtrack up

At Node -3 → {10:1, 7:1}

At Node 11 → {10:1, 7:1, 18:1} ✅ path found

👉 Do you want me to **draw this tree visually with arrows showing prefix sums + map lookups** (like a stepwise tree diagram with annotations), so it’s easier than reading the text version?

def count\_paths(self, root, target\_sum):

        # A map that stores the number of times a prefix sum has occurred so far.

        map = {}

        return self.count\_paths\_prefix\_sum(root, target\_sum, map, 0)

def count\_paths\_prefix\_sum(self, current\_node, target\_sum, map, current\_path\_sum):

        if current\_node is None:

            return 0

        # The number of paths that have the required sum.

        path\_count = 0

        # 'current\_path\_sum' is the prefix sum, i.e., sum of all node values from the root

        # to the current node.

        current\_path\_sum += current\_node.val

        # This is the base case. If the current sum is equal to the target sum, we have found

        # a path from root to the current node having the required sum. Hence, we increment

        # the path count by 1.

        if target\_sum == current\_path\_sum:

            path\_count += 1

        # 'current\_path\_sum' is the path sum from root to the current node. If within this path,

        # there is a valid solution, then there must be an 'old\_path\_sum' such that:

        # => current\_path\_sum - old\_path\_sum = target\_sum

        # => current\_path\_sum - target\_sum = old\_path\_sum

        # Hence, we can search such an 'old\_path\_sum' in the map from the key

        # 'current\_path\_sum - target\_sum'.

        path\_count += map.get(current\_path\_sum - target\_sum, 0)

        # This is the key step in the algorithm. We are storing the number of times the prefix sum

        # `current\_path\_sum` has occurred so far.

        map[current\_path\_sum] = map.get(current\_path\_sum, 0) + 1

        # Counting the number of paths from the left and right subtrees.

        path\_count += self.count\_paths\_prefix\_sum(

            current\_node.left, target\_sum, map, current\_path\_sum)

        path\_count += self.count\_paths\_prefix\_sum(

            current\_node.right, target\_sum, map, current\_path\_sum)

        # Removing the current path sum from the map for backtracking.

        # 'current\_path\_sum' is the prefix sum up to the current node. When we go

        # back (i.e., backtrack), then the current node is no more a part of the path, hence, we

        # should remove its prefix sum from the map.

        map[current\_path\_sum] = map.get(current\_path\_sum, 1) - 1

        return path\_count

Perfect 👍 Let’s do a **step-by-step walkthrough with the diagram annotated** so you see how count\_paths\_prefix\_sum works.

We’ll use this tree:

10

/ \

5 -3

/ \ \

3 2 11

/ \ \

3 -2 1

🎯 **Target Sum = 8**  
We want to count all downward paths (parent → child) where the sum = 8.

**🔹 Step 1: Start at root (10)**

* current\_path\_sum = 10
* map = {} (empty initially)
* current\_path\_sum - target\_sum = 10 - 8 = 2, not in map.
* Path count = 0

Now we store prefix sum:  
map = {10: 1}

**🔹 Step 2: Go left → Node (5)**

* current\_path\_sum = 10 + 5 = 15
* Check if 15 == 8 → no
* Check 15 - 8 = 7 in map? → no
* Path count = 0

Update map:  
map = {10: 1, 15: 1}

**🔹 Step 3: Go left → Node (3)**

* current\_path\_sum = 15 + 3 = 18
* Check if 18 == 8 → no
* Check 18 - 8 = 10 in map? → YES ✅ (map[10] = 1)
* Found a path (10 → 5 → 3) with sum = 8.
* Path count = 1

Update map:  
map = {10: 1, 15: 1, 18: 1}

**🔹 Step 4: Go left → Node (3)**

* current\_path\_sum = 18 + 3 = 21
* Check if 21 == 8 → no
* Check 21 - 8 = 13 in map? → no
* Path count = 0

Map now: {10:1, 15:1, 18:1, 21:1}

(No children → backtrack: decrement map[21])

**🔹 Step 5: Go right → Node (-2)**

* Back at parent (18), now right child
* current\_path\_sum = 18 + (-2) = 16
* Check if 16 == 8 → no
* Check 16 - 8 = 8 in map? → no
* Path count = 0

Map: {10:1, 15:1, 18:1, 16:1}

(No children → backtrack, remove 16)

**🔹 Step 6: Backtrack to Node (5), go right → Node (2)**

* current\_path\_sum = 15 + 2 = 17
* Check 17 == 8? → no
* Check 17 - 8 = 9 in map? → no
* Path count = 0

Map: {10:1, 15:1, 18:1, 17:1}

**🔹 Step 7: Node (1)**

* current\_path\_sum = 17 + 1 = 18
* Check if 18 == 8? → no
* Check 18 - 8 = 10 in map? → YES ✅
* Another path found: (5 → 2 → 1) sums to 8.
* Path count = 1

**🔹 Step 8: Backtrack to Node (10), go right → Node (-3)**

* current\_path\_sum = 10 + (-3) = 7
* Check 7 == 8? → no
* Check 7 - 8 = -1 in map? → no
* Path count = 0

Map: {10:1, 15:0, 18:0, 17:0, 7:1}

**🔹 Step 9: Node (11)**

* current\_path\_sum = 7 + 11 = 18
* Check if 18 == 8 → no
* Check 18 - 8 = 10 in map? → YES ✅
* Found path: (-3 → 11) = 8
* Path count = 1

✅ **Final Count = 3 paths**:

1. 10 → -3 → 11
2. 10 → 5 → 3
3. 5 → 2 → 1

📌 **Big Picture**

* The map keeps track of all **prefix sums** seen so far.
* Each time we’re at a node, we check if (current\_path\_sum - target\_sum) was seen before → meaning a subpath sums to target.
* This avoids recomputing all paths → efficient O(N) solution.

Would you like me to **draw a full tree diagram showing prefix sums at each node and map lookups (color-coded paths that matched)**? That’ll give you a crystal-clear visual.