Count Paths for a Sum (medium)

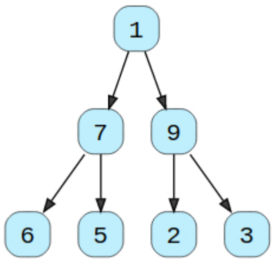
We'll cover the following

Problem Statement

Given a binary tree and a number ‘S’, find all paths in the tree such that the sum of all the node values of each path equals ‘S’. Please note that the paths can start or end at any node but all paths must follow direction from parent to child (top to bottom).

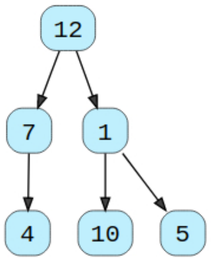
Example 1:

S: 12
Output: 3
Explanation: There are three paths with sum '12':
7 -> 5, 1 -> 9 -> 2, and 9 -> 3



Example 2:

S: 11
Output: 2
Explanation: Here are the two paths with sum '11':
7 -> 4 . and 1 -> 10.



Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

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


def count_paths(root, S):
  # TODO: Write your code here
  return -1


def main():
  root = TreeNode(12)
  root.left = TreeNode(7)
  root.right = TreeNode(1)
  root.left.left = TreeNode(4)
  root.right.left = TreeNode(10)
  root.right.right = TreeNode(5)
  print("Tree has paths: " + str(count_paths(root, 11)))


main()
```

Run Save Reset

Solution

This problem follows the Binary Tree Path Sum pattern. We can follow the same **DFS** approach. But there will be four differences:

1. We will keep track of the current path in a list which will be passed to every recursive call.
2. Whenever we traverse a node we will do two things:
   - Add the current node to the current path.
   - As we added a new node to the current path, we should find the sums of all sub-paths ending at the current node. If the sum of any sub-path is equal to ‘S’ we will increment our path count.
3. We will traverse all paths and will not stop processing after finding the first path.
4. Remove the current node from the current path before returning from the function. This is needed to **Backtrack** while we are going up the recursive call stack to process other paths.

Code

Here is what our algorithm will look like:

Java Python3 C++ JS

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


def count_paths(root, S):
  return count_paths_recursive(root, S, [])


def count_paths_recursive(currentNode, S, currentPath):
  if currentNode is None:
    return 0

  # add the current node to the path
  currentPath.append(currentNode.val)
  pathCount, pathSum = 0, 0
  # find the sums of all sub-paths in the current path list
  for i in range(len(currentPath)-1, -1, -1):
    pathSum += currentPath[i]
    # if the sum of any sub-path is equal to 'S' we increment our path count.
    if pathSum == S:
      pathCount += 1

  # traverse the left sub-tree
  pathCount += count_paths_recursive(currentNode.left, S, currentPath)
  # traverse the right sub-tree
  pathCount += count_paths_recursive(currentNode.right, S, currentPath)

  # remove the current node from the path to backtrack
  # remove the current node from the path to backtrack
  # we need to remove the current node while we are going up the recursive call stack
  del currentPath[-1]

  return pathCount


def main():
  root = TreeNode(12)
  root.left = TreeNode(7)
  root.right = TreeNode(1)
  root.left.left = TreeNode(4)
  root.right.left = TreeNode(10)
  root.right.right = TreeNode(5)
  print("Tree has paths: " + str(count_paths(root, 11)))


main()
```

Run Save Reset

Time complexity

The time complexity of the above algorithm is $O(N^2)$ in the worst case, where ‘N’ is the total number of nodes in the tree. This is due to the fact that we traverse each node once, but for every node, we iterate the current path. The current path, in the worst case, can be $O(N)$ (in the case of a skewed tree). But, if the tree is balanced, then the current path will be equal to the height of the tree, i.e., $O(logN)$. So the best case of our algorithm will be $O(NlogN)$.

Space complexity

The space complexity of the above algorithm will be $O(N)$. This space will be used to store the recursion stack. The worst case will happen when the given tree is a linked list (i.e., every node has only one child). We also need $O(N)$ space for storing the `currentPath` in the worst case.

Overall space complexity of our algorithm is $O(N)$.

← Back
Path With Given Sequence (medium)

Next →
Problem Challenge 1

✓ Completed

Report an Issue Ask a Question