

All Paths for a Sum (medium)

We'll cover the following ^

- Problem Statement
- Try it yourself
- Solution
- Code
 - Time complexity
 - Space complexity
- Similar Problems

Problem Statement

Given a binary tree and a number 'S', find all paths from root-to-leaf such that the sum of all the node values of each path equals 'S'.

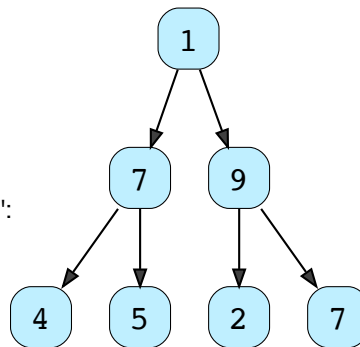
Example 1:

S: 12

Output: 2

Explanation: There are the two paths with sum '12':

1 -> 7 -> 4 and 1 -> 9 -> 2



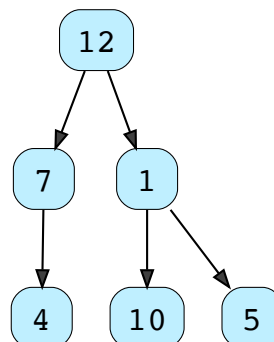
Example 2:

S: 23

Output: 2

Explanation: Here are the two paths with sum '23':

12 -> 7 -> 4 and 12 -> 1 -> 10



Try solving this question here:

Java	Python3	JS JS	C++
<pre> 1 class TreeNode: 2 def __init__(self, val, left=None, right=None): 3 self.val = val 4 self.left = left 5 self.right = right 6 7 8 def find_paths(root, sum): 9 allPaths = [] 10 # TODO: Write your code here 11 return allPaths 12 13 14 def main(): 15 16 root = TreeNode(12) 17 root.left = TreeNode(7) 18 root.right = TreeNode(1) 19 root.left.left = TreeNode(4) 20 root.right.left = TreeNode(10) 21 root.right.right = TreeNode(5) 22 sum = 23 23 print("Tree paths with sum " + str(sum) + 24 ": " + str(find_paths(root, sum))) 25 26 27 main() 28 </pre>			
<div>▶</div> <div>📄 ↩️ ⌂</div>			

Solution

This problem follows the Binary Tree Path Sum

(<https://www.educative.io/collection/page/5668639101419520/5671464854355968/5642684278505472/>) pattern. We can follow the same **DFS** approach. There will be two differences:

1. Every time we find a root-to-leaf path, we will store it in a list.
2. We will traverse all paths and will not stop processing after finding the first path.

Code

Here is what our algorithm will look like:

Java	Python3	C++	JS JS
<pre> 25 # traverse the left sub-tree 26 find_paths_recursive(currentNode.left, required_sum - 27 currentNode.val, currentPath, allPaths) 28 # traverse the right sub-tree 29 find_paths_recursive(currentNode.right, required_sum - </pre>			

```
30         currentNode.val, currentPath, allPaths)
31     }
32     # remove the current node from the path to backtrack,
33     # we need to remove the current node while we are going up the r
34     del currentPath[-1]
35
36
37 def main():
38
39     root = TreeNode(12)
40     root.left = TreeNode(7)
41     root.right = TreeNode(1)
42     root.left.left = TreeNode(4)
43     root.right.left = TreeNode(10)
44     root.right.right = TreeNode(5)
45     required_sum = 23
46     print("Tree paths with required_sum " + str(required_sum) +
47           ": " + str(find_paths(root, required_sum)))
48
49
50 main()
51
```

Time complexity

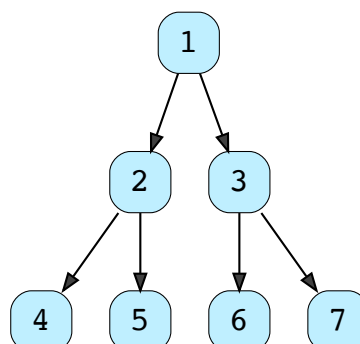
The time complexity of the above algorithm is $O(N^2)$, where 'N' is the total number of nodes in the tree. This is due to the fact that we traverse each node once (which will take $O(N)$), and for every leaf node we might have to store its path which will take $O(N)$.

We can calculate a tighter time complexity of $O(N \log N)$ from the space complexity discussion below.

Space complexity

If we ignore the space required for the `allPaths` list, the space complexity of the above algorithm will be $O(N)$ in the worst case. 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).

How can we estimate the space used for the `allPaths` array? Take the example of the following balanced tree:



Here we have seven nodes (i.e., $N = 7$). Since, for binary trees, there exists only one path to reach any leaf node, we can easily say that total root-to-leaf paths in a binary tree can't be more than the number of leaves. As we know that there can't be more than $N/2$ leaves in a binary tree, therefore the maximum number of elements in `allPaths` will be $O(N/2) = O(N)$. Now, each of these paths can have many nodes in them. For a balanced binary tree (like above), each leaf node will be at maximum depth. As we know that the depth (or height) of a balanced binary tree is $O(\log N)$ we can say that, at the most, each path can have $\log N$ nodes in it. This means that the total size of the `allPaths` list will be $O(N * \log N)$. If the tree is not balanced, we will still have the same worst-case space complexity.

From the above discussion, we can conclude that the overall space complexity of our algorithm is $O(N * \log N)$.

Also from the above discussion, since for each leaf node, in the worst case, we have to copy $\log(N)$ nodes to store its path, therefore the time complexity of our algorithm will also be $O(N * \log N)$.



Similar Problems

Problem 1: Given a binary tree, return all root-to-leaf paths.

Solution: We can follow a similar approach. We just need to remove the “check for the path sum”.

Problem 2: Given a binary tree, find the root-to-leaf path with the maximum sum.

Solution: We need to find the path with the maximum sum. As we traverse all paths, we can keep track of the path with the maximum sum.

[← Back](#)[Next →](#)[Binary Tree Path Sum \(easy\)](#)[Sum of Path Numbers \(medium\)](#)☒ **Mark as Completed** [Report an Issue](#) [Ask a Question](#)
(https://discuss.educative.io/tag/all-paths-for-a-sum-medium__pattern-tree-depth-first-search__grokking-the-coding-interview-patterns-for-coding-questions)