# All Paths for a Sum (medium)

**We'll cover the following**

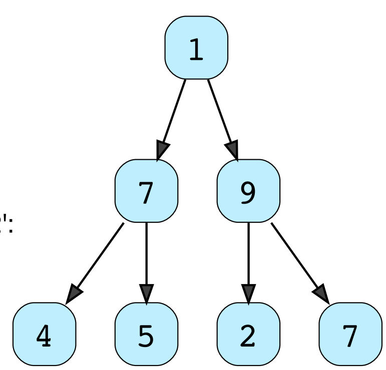


## Problem Statement #

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’.

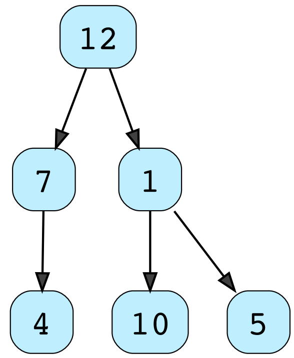
**Example 1:**

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 it yourself #

Try solving this question here:

Java | Python3 | JS | C++

```
# output: path
def find_paths_recursive(node,current_path,all_p
  if not node:
    return
  current_path.append(node.val)
  if node.val == target_sum and not node.left ar
····#·print(all_path)
    all_path.append(list(current_path))
  else:
    find_paths_recursive(node.left,current_path,
    find_paths_recursive(node.right,current_path
  del current_path[-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)
  sum = 23
  print("Tree paths with sum " + str(sum) +
        ": " + str(find_paths(root, sum)))

```

```

main()

```

Output 0.14s

```
Tree paths with sum 23: [[12, 7, 4], [12, 1, 10]]
```

# 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

```
    self.val = val
    self.left = left
    self.right = right


def find_paths(root, required_sum):
  allPaths = []
  find_paths_recursive(root, required_sum, [], a
  return allPaths
```

```
  return allPaths


def find_paths_recursive(currentNode, required_s
  if currentNode is None:
    return

  # add the current node to the path
  currentPath.append(currentNode.val)

  # if the current node is a leaf and its value
  if currentNode.val == required_sum and current
    allPaths.append(list(currentPath))
  else:
    # traverse the left sub-tree
    find_paths_recursive(currentNode.left, requi
                         currentNode.val, currer
    # traverse the right sub-tree
    find_paths_recursive(currentNode.right, requ
                         currentNode.val, currer
```

Output 0.15s

```
Tree paths with required_sum 23: [[12, 7, 4], [12, 1, 10]]
```

## Time complexity #

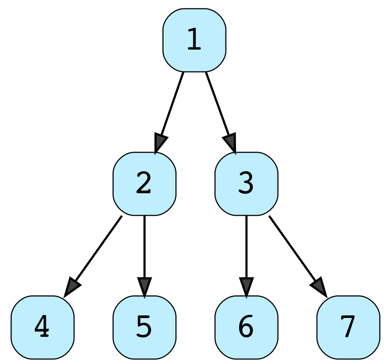
The time complexity of the above algorithm is $O(N^2)$, where 'N' is the total number of nodes in the tree. This is due to the fact that we traverse each node once (which will take $O(N)$), and for every leaf node, we might have to store its path (by making a copy of the current path) which will take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

We can calculate a tighter time complexity of $O(NlogN)$ 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 $(N+1)/2$ leaves in a binary tree, therefore the maximum number of elements in `allPaths` will be $O((N+1)/2) = O(N)$. 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 $O(logN)$ we can say that, at the most, each path can have $logN$ nodes in it. This means that the total size of the allPaths list will be $O(N * logN)$. If the tree is not balanced, we will still have the same worst-case space complexity.

From the above discussion, we can conclude that our algorithm's overall space complexity is $O(N * logN)$.

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 $O(N * logN)$.

## 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
Binary Tree Path Sum (easy)

Next
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)