

## All Paths for a Sum (medium)

### We'll cover the following ^

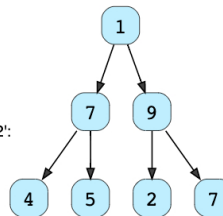
- Problem Statement
- Try it yourself
- Solution
- Code
  - Time complexity
  - Space complexity
- Similar Problems

### 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'.

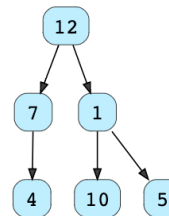
#### Example 1:

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 it yourself #

Try solving this question here:

```
Java Python3 JS C++

1 import java.util.*;
2
3 class TreeNode {
4     int val;
5     TreeNode left;
6     TreeNode right;
7
8     TreeNode(int x) {
9         val = x;
10    }
11 }
12
13 class FindAllTreePaths {
14     public static List<List<Integer>> findPaths(TreeNode root, int sum) {
15         List<List<Integer>> allPaths = new ArrayList<>();
16         // TODO: Write your code here
17         return allPaths;
18     }
19
20     public static void main(String[] args) {
21         TreeNode root = new TreeNode(12);
22         root.left = new TreeNode(7);
23         root.right = new TreeNode(1);
```

```
24 root.left.left = new TreeNode(4);
25 root.right.left = new TreeNode(10);
26 root.right.right = new TreeNode(5);
27 int sum = 23;
28 List<List<Integer>> result = FindAllTreePaths.findPaths(root, sum);
```

Run Save Reset

## Solution #

This problem follows the [Binary Tree Path Sum](#) 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

```
1 import java.util.*;
2
3 class TreeNode {
4     int val;
5     TreeNode left;
6     TreeNode right;
7
8     TreeNode(int x) {
9         val = x;
10    }
11 };
12
13 class FindAllTreePaths {
14     public static List<List<Integer>> findPaths(TreeNode root, int sum) {
15         List<List<Integer>> allPaths = new ArrayList<>();
16         List<Integer> currentPath = new ArrayList<>();
17         findPathsRecursive(root, sum, currentPath, allPaths);
18         return allPaths;
19     }
20
21     private static void findPathsRecursive(TreeNode currentNode, int sum, List<Integer> currentPath,
22         List<List<Integer>> allPaths) {
23         if (currentNode == null)
24             return;
25
26         // add the current node to the path
27         currentPath.add(currentNode.val);
28     }
```

Run Save Reset

## Time complexity #

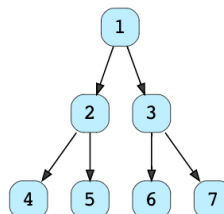
The time complexity of the above algorithm is  $O(N^2)$ , where 'N' is the total number of nodes in the tree. This is due to the fact that we traverse each node once (which will take  $O(N)$ ), and for every leaf node, we might have to store its path (by making a copy of the current path) which will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

We can calculate a tighter time complexity of  $O(N \log N)$  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  $(N + 1)/2$  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  $O(\log N)$  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 our algorithm's overall space complexity 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](#)

Binary Tree Path Sum (easy)

[Next →](#)

Sum of Path Numbers (medium)

☒ Mark as Completed

 Report an Issue  Ask a Question