All Paths for a Sum (medium)

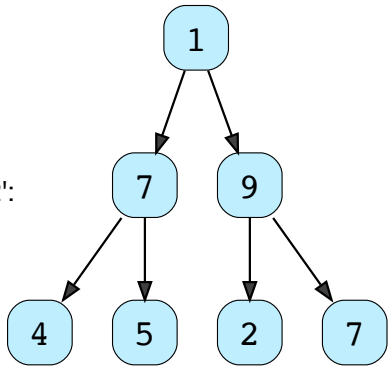
We'll cover the following

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’.

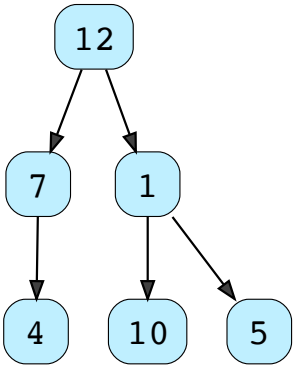
Example 1:

S: 12
Output: [[1, 7, 4], [1, 9, 2]]
Explanation: Here are the two paths with sum '12':
1 -> 7 -> 4 and 1 -> 9 -> 2



Example 2:

S: 23
Output: [[12, 7, 4], [12, 1, 10]]
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++

```
class TreeNode {
  constructor(value) {
    this.value = value;
    this.left = null;
    this.right = null;
```

```
  }
};


const find_paths = function(root, sum) {
  allPaths = [];
  // TODO: Write your code here
  return allPaths;
};


var root = new TreeNode(12)
root.left = new TreeNode(7)
root.right = new TreeNode(1)
root.left.left = new TreeNode(4)
root.right.left = new TreeNode(10)
root.right.right = new TreeNode(5)
sum = 23
console.log(`Tree paths with sum: ${sum}: ${find_paths(root, sum)}`)

```

Run Save Reset

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

```
const Deque = require('./collections/deque'); //http://www.collectionsjs.com

class TreeNode {
  constructor(val, left = null, right = null) {
    this.val = val;
    this.left = left;
    this.right = right;
  }
}


function find_paths(root, sum) {
  allPaths = [];
  find_paths_recursive(root, sum, new Deque(), allPaths);
  return allPaths;
}


function find_paths_recursive(currentNode, sum, currentPath, allPaths) {
  if (currentNode === null) {
    return;
  }

  // add the current node to the path
  currentPath.push(currentNode.val);

  // if the current node is a leaf and its value is equal to sum, save the curre
  if (currentNode.val === sum && currentNode.left === null && currentNode.right =
    allPaths.push(currentPath.toArray());
  } else {
    // traverse the left sub-tree
    find_paths_recursive(currentNode.left, sum - currentNode.val, currentPath, a
    // traverse the right sub-tree
```

```
    find_paths_recursive(currentNode.right, sum - currentNode.val, currentPath, a
  }
  // remove the current node from the path to backtrack,
  // we need to remove the current node while we are going up the recursive call
  currentPath.pop();
}


const root = new TreeNode(12);
root.left = new TreeNode(7);
root.right = new TreeNode(1);
root.left.left = new TreeNode(4);
root.right.left = new TreeNode(10);
root.right.right = new TreeNode(5);
let sum = 23,
  result = find_paths(root, sum);

process.stdout.write(`Tree paths with sum ${sum}: `);
for (i = 0; i < result.length; i++) {
  process.stdout.write(`[${result[i]}] `);
}
```

Run Save Reset

Time complexity#

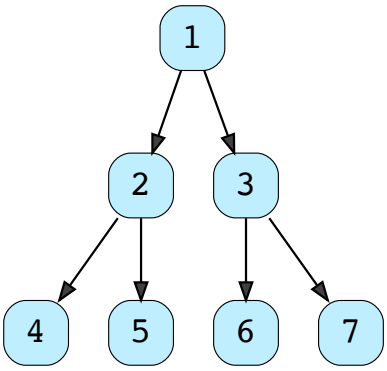
The time complexity of the above algorithm is $O(N^2)$, where 'N' is the total number of nodes in the tree. This is due to the fact that we traverse each node once (which will take $O(N)$), and for every leaf node, we might have to store its path (by making a copy of the current path) which will take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

We can calculate a tighter time complexity of $O(NlogN)$ 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

Next

Binary Tree Path Sum (easy)

Sum of Path Numbers (medium)

Mark as Completed

Report an Issue