All Paths for a Sum (medium)

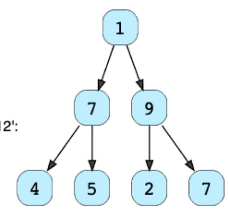
We'll cover the following

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'.

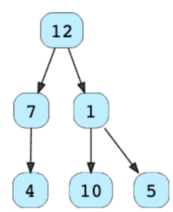
Example 1:

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++

```java
import java.util.*;

class TreeNode {
  int val;
  TreeNode left;
  TreeNode right;

  TreeNode(int x) {
    val = x;
  }
};

class FindAllTreePaths {
  public static List<List<Integer>> findPaths(TreeNode root, int sum) {
    List<List<Integer>> allPaths = new ArrayList<>();
    // TODO: Write your code here
    return allPaths;
  }

  public static void main(String[] args) {
    TreeNode 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);
    int sum = 23;
    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

```java

    // if the current node is a leaf and its value is equal to sum, save the current path
    if (currentNode.val == sum && currentNode.left == null && currentNode.right == null) {
      allPaths.add(new ArrayList<Integer>(currentPath));
    } else {
      // traverse the left sub-tree
      findPathsRecursive(currentNode.left, sum - currentNode.val, currentPath, allPaths);
      // traverse the right sub-tree
      findPathsRecursive(currentNode.right, sum - currentNode.val, currentPath, allPaths);
    }

    // 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.
    currentPath.remove(currentPath.size() - 1);
  }

  public static void main(String[] args) {
    TreeNode 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);
    int sum = 23;
    List<List<Integer>> result = FindAllTreePaths.findPaths(root, sum);
    System.out.println("Tree paths with sum " + sum + ": " + result);
  }
```

RUN SAVE RESET

Time complexity

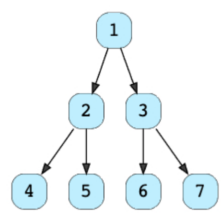
The time complexity of the above algorithm is $O(N^2)$, where 'N' is the total number of nodes in the tree. This is due to the fact that we traverse each node once (which will take $O(N)$), and for every leaf node we might have to store its path which will take $O(N)$.

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 $N/2$ 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 $O(logN)$ 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 the overall space complexity of our algorithm 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

Explore
Tracks
My Courses
Edpresso
Refer a Friend

Create

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.

MARK AS COMPLETED

← Back
Binary Tree Path Sum (easy)

Next →
Sum of Path Numbers (medium)

Report an Issue
Ask a Question