Binary Tree Path Sum (easy)

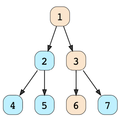
We'll cover the following

Problem Statement

Given a binary tree and a number ‘S’, find if the tree has a path from root-to-leaf such that the sum of all the node values of that path equals ‘S’.

Example 1:

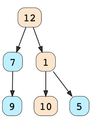
S: 10
Output: true
Explanation: The path with sum ‘10’ is highlighted



Example 2:

S: 23
Output: true
Explanation: The path with sum ‘23’ is highlighted

S: 16
Output: false
Explanation: There is no root-to-leaf path with sum ‘16’.



Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

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

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

class TreePathSum {
  public static boolean hasPath(TreeNode root, int sum) {
    // TODO: Write your code here
    return false;
  }

  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(9);
    root.right.left = new TreeNode(10);
    root.right.right = new TreeNode(5);
    System.out.println("Tree has path: " + TreePathSum.hasPath(root, 23));
    System.out.println("Tree has path: " + TreePathSum.hasPath(root, 16));
  }
}
```

Run | Save | Reset

Solution

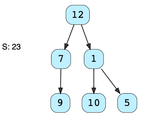
As we are trying to search for a root-to-leaf path, we can use the **Depth First Search (DFS)** technique to solve this problem.

To recursively traverse a binary tree in a DFS fashion, we can start from the root and at every step, make two recursive calls one for the left and one for the right child.

Here are the steps for our Binary Tree Path Sum problem:

1. Start DFS with the root of the tree.
2. If the current node is not a leaf node, do two things:
   - Subtract the value of the current node from the given number to get a new sum => `S = S - node.value`
   - Make two recursive calls for both the children of the current node with the new number calculated in the previous step.
3. At every step, see if the current node being visited is a leaf node and if its value is equal to the given number ‘S’. If both these conditions are true, we have found the required root-to-leaf path, therefore return `true`.
4. If the current node is a leaf but its value is not equal to the given number ‘S’, return false.

Let’s take the example-2 mentioned above to visually see our algorithm:



Let’s start with the root node

1 of 10

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

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

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

class TreePathSum {
  public static boolean hasPath(TreeNode root, int sum) {
    if (root == null)
      return false;

    // if the current node is a leaf and its value is equal to the sum, we've found a path
    if (root.val == sum && root.left == null && root.right == null)
      return true;

    // recursively call to traverse the left and right sub-tree
    // return true if any of the two recursive call return true
    return hasPath(root.left, sum - root.val) || hasPath(root.right, sum - root.val);
  }

  public static void main(String[] args) {
    TreeNode root = new TreeNode(12);
    root.left = new TreeNode(7);
    root.right = new TreeNode(1);
```

Run | Save | Reset

Time complexity

The time complexity of the above algorithm is $O(N)$, where ‘N’ is the total number of nodes in the tree. This is due to the fact that we traverse each node once.

Space complexity

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).