Path With Given Sequence (medium)

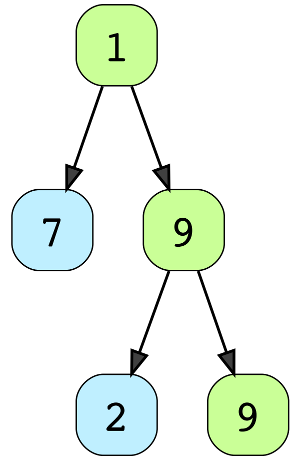
We'll cover the following

Problem Statement#

Given a binary tree and a number sequence, find if the sequence is present as a root-to-leaf path in the given tree.

Example 1:

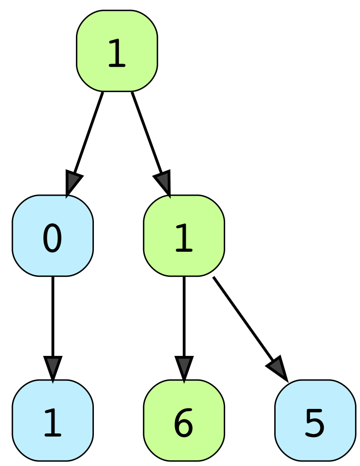
Sequence: [1, 9, 9]
Output: true
Explanation: The tree has a path 1 -> 9 -> 9.



Example 2:

Sequence: [1, 0, 7]
Output: false
Explanation: The tree does not have a path 1 -> 0 -> 7.

Sequence: [1, 1, 6]
Output: true
Explanation: The tree has a path 1 -> 1 -> 6.



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_path = function(root, sequence) {
  // TODO: Write your code here
  return false;
};



var root = new TreeNode(1)
root.left = new TreeNode(0)
root.right = new TreeNode(1)
root.left.left = new TreeNode(1)
root.right.left = new TreeNode(6)
root.right.right = new TreeNode(5)

console.log(`Tree has path sequence: ${find_path(root, [1, 0, 7])}`)
console.log(`Tree has path sequence: ${find_path(root, [1, 1, 6])}`)
```

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 and additionally, track the element of the given sequence that we should match with the current node. Also, we can return `false` as soon as we find a mismatch between the sequence and the node value.

Code#

Here is what our algorithm will look like:

Java Python3 C++ JS

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


function find_path(root, sequence) {
  if (root === null) {
    return sequence.length === 0;
  }

  return find_path_recursive(root, sequence, 0);
}


function find_path_recursive(currentNode, sequence, sequenceIndex) {
  if (currentNode === null) {
    return false;
  }

  const seqLen = sequence.length;
  if (sequenceIndex >= seqLen || currentNode.val !== sequence[sequenceIndex]) {
    return false;
  }

  // if the current node is a leaf, add it is the end of the sequence, we have f
  if (currentNode.left === null && currentNode.right === null && sequenceIndex =
    return true;
  }

  // recursively call to traverse the left and right sub-tree
  // return true if any of the two recursive call return true
  return find_path_recursive(currentNode.left, sequence, sequenceIndex + 1) ||
```

```
    find_path_recursive(currentNode.right, sequence, sequenceIndex + 1);
}


const root = new TreeNode(1);
root.left = new TreeNode(0);
root.right = new TreeNode(1);
root.left.left = new TreeNode(1);
root.right.left = new TreeNode(6);
root.right.right = new TreeNode(5);

console.log(`Tree has path sequence: ${find_path(root, [1, 0, 7])}`);
console.log(`Tree has path sequence: ${find_path(root, [1, 1, 6])}`);
```

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).

← Back

Sum of Path Numbers (medium)

Next →

Count Paths for a Sum (medium)

Mark as Completed

Report an Issue