Solution Review: Problem Challenge 2

We'll cover the following

- Path with Maximum Sum (hard)
- Solution
- Code
  - Time complexity
  - Space complexity

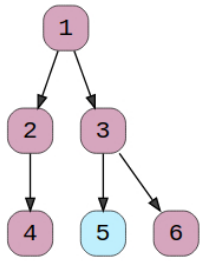
Path with Maximum Sum (hard)

Find the path with the maximum sum in a given binary tree. Write a function that returns the maximum sum.

A path can be defined as a **sequence of nodes between any two nodes** and doesn't necessarily pass through the root. The path must contain at least one node.

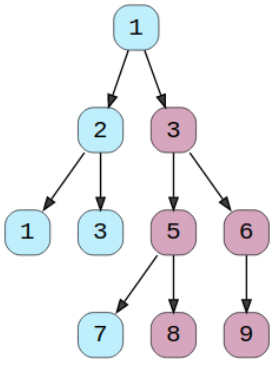
Example 1:

Output: 16
Explanation: The path with maximum sum is: [4, 2, 1, 3, 6]



Example 2:

Output: 31
Explanation: The path with maximum sum is: [8, 5, 3, 6, 9]



Solution

This problem follows the Binary Tree Path Sum pattern and shares the algorithmic logic with Tree Diameter. We can follow the same **DFS** approach. The only difference will be to ignore the paths with negative sums. Since we need to find the overall maximum sum, we should ignore any path which has an overall negative sum.

Code

Here is what our algorithm will look like, the most important changes are in the highlighted lines:

Java | Python3 | C++ | JS

```python
import math


class TreeNode:
  def __init__(self, val, left=None, right=None):
    self.val = val
    self.left = left
    self.right = right


class MaximumPathSum:

  def find_maximum_path_sum(self, root):
    self.globalMaximumSum = -math.inf
    self.find_maximum_path_sum_recursive(root)
    return self.globalMaximumSum

  def find_maximum_path_sum_recursive(self, currentNode):
    if currentNode is None:
      return 0

    maxPathSumFromLeft = self.find_maximum_path_sum_recursive(
      currentNode.left)
    maxPathSumFromRight = self.find_maximum_path_sum_recursive(
      currentNode.right)

    # ignore paths with negative sums, since we need to find the maximum sum we should
    # ignore any path which has an overall negative sum.
    maxPathSumFromLeft = max(maxPathSumFromLeft, 0)
    maxPathSumFromRight = max(maxPathSumFromRight, 0)

    # maximum path sum at the current node will be equal to the sum from the left subtree +
    # the sum from right subtree + val of current node
    localMaximumSum = maxPathSumFromLeft + maxPathSumFromRight + currentNode.val

    # update the global maximum sum
    self.globalMaximumSum = max(self.globalMaximumSum, localMaximumSum)

    # maximum sum of any path from the current node will be equal to the maximum of
    # the sums from left or right subtrees plus the value of the current node
    return max(maxPathSumFromLeft, maxPathSumFromRight) + currentNode.val


def main():
  maximumPathSum = MaximumPathSum()
  root = TreeNode(1)
  root.left = TreeNode(2)
  root.right = TreeNode(3)

  print("Maximum Path Sum: " + str(maximumPathSum.find_maximum_path_sum(root)))
  root.left.left = TreeNode(1)
  root.left.right = TreeNode(3)
  root.right.left = TreeNode(5)
  root.right.right = TreeNode(6)
  root.right.left.left = TreeNode(7)
  root.right.left.right = TreeNode(8)
  root.right.right.left = TreeNode(9)
  print("Maximum Path Sum: " + str(maximumPathSum.find_maximum_path_sum(root)))

  root = TreeNode(-1)
  root.left = TreeNode(-3)
  print("Maximum Path Sum: " + str(maximumPathSum.find_maximum_path_sum(root)))


main()

```

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).

one child).

Back

Problem Challenge 2

Next

Introduction

Completed