1120. Maximum Average Subtree Medium Tree Depth-First Search Binary Tree

Problem Description The problem provides us with the root node of a binary tree and asks us to calculate the maximum average value of any subtree

within that binary tree. Here, a subtree can be any node and all of its descendants—it could be the entire tree, a leaf node, or any other set of connected nodes within the tree. The average value of a subtree is the sum of all node values in that subtree divided by the number of nodes it contains. Since we are

Leetcode Link

looking for the maximum average, we are concerned with finding the subtree that has the highest average value compared to all other subtrees in the binary tree.

The intuitive approach to solving this problem is to explore each subtree, calculate its sum and node count, and then use these to

compute the average value for that particular subtree. To achieve this, we'll need to perform a traversal of the binary tree.

Intuition

A depth-first search (DFS) traversal is suitable for this task because it allows us to go deep into the tree to examine individual subtrees before moving on to the next subtree. This way, we can use a postorder traversal (left-right-root) to compute the sum and node count of each subtree.

While traversing, for each node, we calculate: 1. The total sum of node values (s) in the current subtree (includes the current node, its left subtree, and its right subtree). 2. The total number of nodes (n) in the current subtree.

With both these values at hand for each subtree, we then calculate the average by dividing s by n. This average is then compared to

The purpose of the nonlocal ans in the solution code is to allow the nested dfs function to modify the ans variable defined in the

- the current maximum average we've found (ans). If the current average is greater, we update ans with this new value.
- enclosing scope, which maintains the current maximum average. As we recursively return the sum and node count for each subtree, we can easily compute the average for each parent subtree,

inherently checking every subtree in the binary tree. The function ultimately returns the maximum average found during the traversal.

Solution Approach The solution uses a recursive function to traverse the tree using depth-first search (DFS), specifically a postorder traversal pattern.

In postorder traversal, we visit the left and right subtrees before processing the current node. This allows us to gather information from the subtrees before deciding on the parent node's information, which is essential for this problem as we need to compute subtrees' sums and node counts.

s is the sum of all nodes within the current subtree.

on is the count of the nodes within the current subtree. 2. When the dfs function is called with None (i.e., the subtree is empty), it returns (0, 0), indicating that the sum and count are both 0 for an empty subtree. 3. The function calculates 1s and 1n by recursively calling dfs on the left child of the current node, and similarly rs and rn for the

right child. 4. Once we have the sums (1s and rs) and counts (1n and rn) for both child subtrees, compute the sum s for the current node,

contain the maximum average.

the number of nodes in the binary tree.

1 if root is None:

1 nonlocal ans

end of the traversal.

Example Walkthrough

return 0, 0

Here's a breakdown of the solution implementation:

which is the value of the current node plus ls plus rs.

that it retains its value across different calls to dfs and isn't reinitialized in each recursive call.

1. Define a recursive function dfs that takes a node as an argument and returns a tuple (s, n) where:

5. Compute the count n for the current node as 1 (to count the current node) plus ln plus rn.

The code for calculating sum and number of nodes in each subtree looks like this:

Let's walk through a simple example to illustrate the solution approach. Consider a binary tree:

Call dfs on the right subtree (None), which returns (0, 0).

Call dfs on the left subtree (Node 7), returning (7, 1).

Call dfs on the right subtree (Node 3), returning (3, 1).

And updating the maximum average found so far is as simple as:

7. Update ans with the maximum value between the current ans and the average for the current subtree (s / n). 8. After processing the root node, the dfs traversal ensures that each node's subtree has been considered. The ans variable will

6. Use a nonlocal variable ans to keep track of the current maximum average. This variable is defined outside the dfs function so

10. Lastly, return the ans variable as the final result, which represents the maximum average value of a subtree of the binary tree. This approach is efficient because it traverses each node of the tree exactly once, resulting in an O(n) time complexity, where n is

9. The main function of the class Solution will initialize ans to 0 and call the dfs function on the binary tree's root.

ls, ln = dfs(root.left) # Postorder traversal - Left rs, rn = dfs(root.right) # Postorder traversal - Right 5 s = root.val + ls + rs # Sum of current subtree 6 n = 1 + ln + rn # Number of nodes in current subtree

2 ans = max(ans, s / n)By carefully updating the ans variable during the traversal, we ensure that we have the maximum average value of any subtree by the

Call dfs on the left subtree (Node 1), which returns (1, 1) - its own value and count since it is a leaf node.

3. Now, the sum at Node 5 is 5 (itself) + 7 (left subtree) + 18 (right subtree) = 30 and the count is 1 (itself) + 2 (left subtree) + 3

This example illustrates the process of traversing the tree using postorder traversal, where we calculate the sum and count of the

Now, the sum at Node 6 is 6 (itself) + 1 (left subtree) = 7 and the count is 1 (itself) + 1 (left subtree) = 2.

Update the maximum average ans if the average here (7 / 2 = 3.5) is greater than the current ans.

We want to find the maximum average value of any subtree. Start by calling the dfs function on the root. We initialize ans = 0. At Node 5 - the root:

At Node 6:

At Node 8:

(right subtree) = 6.

Python Solution

class TreeNode:

10

13

14

15

16

17

18

20

21

22

23

24

25

33

34

35

8

9

10

the subtree rooted with Node 8).

1 # Definition for a binary tree node.

def dfs(node):

if node is None:

return 0, 0

nonlocal max_average

return max_average

private double maxAverage;

return maxAverage;

#include <utility> // for std::pair

// Definition for a binary tree node.

if (!node) {

TreeNode() : val(0), left(nullptr), right(nullptr) {}

double maximumAverageSubtree(TreeNode* root) {

TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}

return {0, 0}; // If the node is null, return 0 sum and 0 count.

// Current sum is the value of the node plus sum of left and right subtrees.

// Current count is 1 (for the current node) plus count of left and right subtrees.

maxAverage = std::max(maxAverage, static_cast<double>(currentSum) / currentCount);

int currentSum = node->val + leftSubtree.first + rightSubtree.first;

int currentCount = 1 + leftSubtree.second + rightSubtree.second;

// Update the maximum average if the current average is greater.

std::function<std::pair<int, int>(TreeNode*)> dfs = [&](TreeNode* node) -> std::pair<int, int> {

double maxAverage = 0; // This will hold the maximum average found.

// Recursive function to perform DFS (Depth-First Search).

std::pair<int, int> leftSubtree = dfs(node->left);

std::pair<int, int> rightSubtree = dfs(node->right);

// Return the current sum and count for further use.

// Compute the sum and count for left subtree.

// Compute the sum and count for right subtree.

struct TreeNode {

int val;

15 class Solution {

public:

8

9

10

11

12

14

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

13 };

TreeNode *left;

TreeNode *right;

dfs(root);

left_sum, left_count = dfs(node.left)

right_sum, right_count = dfs(node.right)

total_sum = node.val + left_sum + right_sum

// A member variable to store the maximum average found so far.

public double maximumAverageSubtree(TreeNode root) {

// This method starts the process by calling the DFS method on the root.

total_count = 1 + left_count + right_count

Sum at Node 8 is 8 + 7 + 3 = 18 and count is 1 + 1 + 1 = 3. Update the maximum average ans if the average here (18 / 3 = 6) is greater than the current ans.

2. Then, call dfs on the right subtree (Node 8).

1. First, call dfs on the left subtree (Node 6).

- 4. Update the maximum average ans if the average here (30 / 6 = 5) is greater than the current ans. After traversing the entire tree, the ans variable holds the maximum average found, which is 6 in this case (the average value of
- nodes for each subtree, updating the maximum average found (ans) along the way. The tree is only traversed once, making the solution efficient with O(n) complexity, where n is the number of nodes in the binary tree.

def maximumAverageSubtree(self, root: Optional[TreeNode]) -> float:

Helper function to perform DFS (Depth-First Search) on the tree.

If the node is null, return sum = 0 and count = 0

max_average = max(max_average, total_sum / total_count)

Return the final maximum average value after traversing all subtrees

Recurse on the left child and get left subtree sum and node count

Recurse on the right child and get right subtree sum and node count

Calculate the total sum and total count of nodes including this node

Update the maximum average as max of current max and this subtree's average

This function returns the sum of subtree values and the count of nodes in the subtree.

def __init__(self, val=0, left=None, right=None): self.val = val self.left = left self.right = right class Solution:

26 # Return the sum and count to parent node's calculation 27 return total_sum, total_count 28 29 # Initialize the maximum average variable 30 max_average = 0 # Call the helper function and start the DFS from the root 32 dfs(root)

```
Java Solution
```

class Solution {

```
// Perform depth-first search to calculate sum and count of nodes for each subtree.
11
       private int[] dfs(TreeNode node) {
12
           // Base case: If the current node is null, return an array containing 0 sum and 0 node count.
13
           if (node == null) {
14
15
               return new int[2]; // Default initialization to {0, 0}.
16
17
           // Recursive case: Calculate the sum of values and count of nodes in the left subtree.
           int[] leftSubtreeResult = dfs(node.left);
19
20
           // Recursive case: Calculate the sum of values and count of nodes in the right subtree.
21
           int[] rightSubtreeResult = dfs(node.right);
22
23
           // Computing the sum and count for the current subtree including the current node's value.
           int sumSubtree = node.val + leftSubtreeResult[0] + rightSubtreeResult[0];
24
           int countSubtree = 1 + leftSubtreeResult[1] + rightSubtreeResult[1];
26
27
           // Update the maximum average if the average of the current subtree is greater.
           maxAverage = Math.max(maxAverage, (double) sumSubtree / countSubtree);
28
29
           // Return an array containing the sum and count for use in parent node calculations.
30
           return new int[] {sumSubtree, countSubtree};
31
32
33 }
34
35
   /**
    * Definition for a binary tree node.
    * public class TreeNode {
          int val; // The value of the node.
38
          TreeNode left; // Reference to the left child node.
39
          TreeNode right; // Reference to the right child node.
40
          TreeNode() {} // Default constructor.
          TreeNode(int val) { this.val = val; } // Constructor with initial value.
          TreeNode(int val, TreeNode left, TreeNode right) { // Constructor with initial value and left/right children.
              this.val = val;
44
              this.left = left;
45
              this.right = right;
46
    *
    *
48
    * }
    */
50
C++ Solution
  1 #include <algorithm> // for std::max
  2 #include <functional> // for std::function
```

return {currentSum, currentCount}; 40 **}**; 41 42 // Start DFS from the root of the tree. 43 dfs(root); 44

```
// After the DFS is complete, maxAverage holds the maximum average value.
 45
 46
             return maxAverage;
 47
 48
    };
 49
Typescript Solution
 1 // Definition for a binary tree node.
   class TreeNode {
       val: number;
       left: TreeNode | null;
       right: TreeNode | null;
       constructor(val: number = 0, left: TreeNode | null = null, right: TreeNode | null = null) {
           this.val = val;
           this.left = left;
           this.right = right;
10
11
12 }
13
   let maximumAverage: number = 0.0; // This will hold the maximum average found.
15
16 // A helper function that takes a TreeNode and returns the sum and number of nodes
  // in the subtree rooted at the given node using depth-first search.
   function dfs(node: TreeNode | null): [number, number] {
       if (!node) {
19
            return [0, 0]; // If the node is null, return 0 sum and 0 count.
21
22
23
       // Compute the sum and count for the left and right subtrees.
       const [leftSum, leftCount] = dfs(node.left);
24
       const [rightSum, rightCount] = dfs(node.right);
26
27
       // Calculate current sum by adding the node's value to the sum of its subtrees.
       const currentSum = node.val + leftSum + rightSum;
28
29
       // Count the number of nodes in this subtree.
       const currentCount = 1 + leftCount + rightCount;
30
31
32
       // Update the maximum average if the current node's average is greater.
33
       maximumAverage = Math.max(maximumAverage, currentSum / currentCount);
34
35
       // Return the current sum and count for the subtree.
36
       return [currentSum, currentCount];
37 }
38
   // The entry function that initiates the depth-first search to find the maximum
   // average of any subtree in the binary tree, starting from the root.
   function maximumAverageSubtree(root: TreeNode | null): number {
       dfs(root); // Start DFS from the root of the tree.
       return maximumAverage;
43
45
```

Time and Space Complexity

The given Python code defines a function maximumAverageSubtree that calculates the maximum average value of all subtrees in a binary tree. The dfs function, a helper function, is used to perform a depth-first search on the tree. Time Complexity:

The time complexity for this code is determined by the need to visit each node of the binary tree exactly once during the DFS

traversal. Since every node is visited exactly once, and at every node we perform a constant amount of work (calculating sums and

the number of nodes, and updating the maximum average), the time complexity is O(N), where N is the number of nodes in the binary

Space Complexity:

tree.

The space complexity is a bit more tricky to analyze due to the recursive nature of the DFS. In the worst case, the space complexity is defined by the height of the tree, which in the case of a skewed tree (where every node has only one child) could be O(N). However, for a balanced tree, the height (thus the maximum depth of the recursive call stack) would be O(log N). Thus, the space complexity of the code is O(H), where H is the height of the tree. In the average case for a reasonably balanced tree, this can be expressed as O(log N), but in the worst-case scenario (a completely unbalanced tree), the space complexity would be O(N).

 Time Complexity: 0(N) Space Complexity: O(H), which would be O(log N) in a balanced tree and O(N) in the worst case.

To summarize: