Problem Description

same value. A path in a binary tree is a sequence of nodes where there is an edge between any two consecutive nodes in the sequence, and the length of the path is represented by the number of edges in that sequence. This specific path can start and end at any node, not necessarily including the root.

The given problem requires us to find the length of the longest path in a binary tree, where all the nodes along the path have the

Intuition

current node.

To solve this problem, we employ a depth-first search (DFS) strategy to traverse the tree. The key intuition here is to use recursion to explore each subtree to find the longest path with equal node values that start from the current node and extend downwards. Once a node is reached by the recursion:

2. We recursively obtain the lengths of the paths extending from its left and right child nodes that have the same value as the

o If the left child's value equals the current node's value, we increment the left path length by one to include the current node;

1. If the node is null, we cannot extend the path, so we return zero.

3. To build the answer for the current node, we consider these cases:

both as we are considering paths, not splits; so we return the larger of the two.

- otherwise, the longest path starting from the current node towards the left is zero, since the values don't match. o Similarly, if the right child's value equals the current node's value, we increment the right path length by one; if not, the right
- path's length is zero.

exploring the tree depth-first, starting from the root.

path we have found during the recursion.

Similarly, we do the same for root.right.

- 4. The longest path that goes through the current node (but not necessarily includes it) is the sum of the left and right path lengths we just calculated. We update our global answer, ans, if this sum is larger than the current answer. 5. Each call needs to return the maximum length from either the left or the right that includes the current node. We can't return
- In the provided solution, dfs is our recursive function, and ans is the variable that keeps track of the longest path found so far. After calling dfs on our root node, ans will contain the maximum length of the path we are looking for, and we return it as the solution.
- **Solution Approach** The solution implements a recursive depth-first search approach, similar to finding the diameter of a binary tree (as suggested in the reference solution approach). Here's a more detailed breakdown of the implementation:

1. Recursive Depth-First Search (DFS): • We define a nested recursive helper function dfs inside our longestUnivaluePath method. This function is responsible for

The purpose of dfs is to traverse the tree and calculate the longest univalue path for each node recursively.

2. Global Variable for the Answer: • We use a nonlocal variable ans which is declared before the nested dfs function and is used to keep track of the longest

o Inside dfs, we initially check if the current root node is None. If it is, the function returns 0, as there is no path extending from a nonexistent node.

3. Tracking the Current Node's Path:

- We then recursively call dfs for both the left and right child nodes of the current root. 4. Path Extension Logic:
- We extend the path only if the child node has the same value as the current node. If root. left exists and its value is the same as root.val, we add 1 to the value returned by dfs(root.left); otherwise, we reset it to 0.
- 5. Updating the Longest Path (ans): At each step, after computing the value of left and right univalue paths, we update our global variable ans with the sum of

7. End of Recursion and Result:

between function calls.

Example Walkthrough

(provided the values match).

• This step corresponds to potentially joining two paths together at the current root node to form a longer path. 6. Returning the Best Unidirectional Path:

left and right paths if this sum represents a new maximum. It's crucial to recognize that the longest univalue path that

passes through the current node is the sum of the longest univalue paths going down through its left and right children

Since we cannot include both the left and right paths in our final answer (as that would constitute a split, not a single path),

Finally, the longestUnivaluePath method returns the value of ans.

able to build the overall solution globally by gathering and combining the local answers.

2. In our first call to the recursive dfs function, we examine the root having the value 4.

child nodes. The left child (value 4) and right child (value 5) calls are made.

Let us consider a small binary tree to illustrate the solution approach:

we return the maximum of the two, again incremented by 1 for the current node if applicable.

The key algorithmic pattern here is recursion in tandem with a global variable to keep track of the optimal solution. The code uses a straightforward recursive DFS strategy tailored to track a specific condition—the nodes along a path have the same value. By breaking down the problem into a series of recursive steps, each handling a local part of the tree around a single node, we are

The recursion bottoms out when we hit None nodes, and it builds the solution as it unwinds the recursive calls.

Note that by defining the dfs function within the longestUnivaluePath method, we are able to use the nonlocal keyword which

allows us to modify the ans variable inside the nested function. This is an example of using closure in Python to maintain the state

Now, let's walk through the algorithm on this example: 1. We initiate the longestUnivaluePath with the root of the tree (value 4).

3. We recursively call dfs on the left and right children, which will return the longest univalue paths with the same value as these

For the left child (value 4): 4. The dfs function is called with the left child, which is a node with value 4. 5. Similarly, dfs is called on its left and right children. Both have the value 4, so each child will contribute a value of 1 to the path

6. No further calls to dfs are necessary down the leftmost branch as the children are None. The call for the left child of our current

7. The call for the right child (also value 4) will also return 1 because its children are None. 8. Both children had the same value as the current node, so the total path length for this node is the sum of both children plus the current node, which is 1 + 1 = 2. This value 2 will be considered further up the tree.

For the right child (value 5):

length (after the dfs call).

9. The dfs function is called with the right child, which is a node with value 5. 10. Its left child is None, and its right child has the same value, 5, so it will contribute 1.

path provided a length of 1 (obtained from the right child call).

node (value 4) returns 1 as the maximum from either side.

11. No further dfs calls down this path are necessary because the right child of the right node is None. The returned path length is 1. Back to the root:

between the left and right path lengths, which is 2 from the left, and returns 2.

answer ans if 3 is greater than the current value of ans. 14. The dfs call at the root then needs to return the longest univalue path that includes the root node. So it chooses the maximum

After the root node's dfs function is processed, our answer (ans) holds the longest univalue path, which is 3. This is the sum of the

lengths from the left path and the right path passing through the root. Since we start counting from one node and end at another,

considering the number of edges between the nodes, we have two edges, which represent the longest path, where all nodes share

13. We take the sum of both lengths since both children's values match with the root's value, so we get 2 + 1. We update the global

12. Now we return to the root node with value 4. The left path provided a length of 2 (obtained from the left child calls), and the right

The visualization of the longest univalue path is as follows:

the same value.

- So, the function longestUnivaluePath finally returns the value 2 which is the length of the longest path with equal node values in the tree.
- **Python Solution** 1 class TreeNode:
- 22 left_path = left_path_length + 1 if node.left and node.left.val == node.val else 0 23 24 # Similarly for the right child. 25 right_path = right_path_length + 1 if node.right and node.right.val == node.val else 0 26 # We use a nonlocal variable 'longest_path' to keep track of the 27

return max(left_path, right_path)

nonlocal longest_path

longest_path = 0

return longest_path

dfs(root)

Java Solution

left_path_length = dfs(node.left)

right_path_length = dfs(node.right)

Initializer for the TreeNode class

self.val = val

def dfs(node):

if node is None:

return 0

class Solution:

9

10

11

12

13

14

15

16

17

18

19

20

21

28

29

30

31

32

33

34

35

36

37

38

39

42

43

44

45

46

47

48

49

50

51

52

53

55

54 }

5

6

8

9

10

11

12

13

14

16

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

6

15 };

18 public:

private:

C++ Solution

2 struct TreeNode {

17 class Solution {

int val;

TreeNode *left;

TreeNode *right;

1 // Definition for a binary tree node.

self.left = left

self.right = right

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

def longestUnivaluePath(self, root: TreeNode) -> int:

to find the longest univalue path from the root node.

longest univalue path found during the DFS.

longest_path = max(longest_path, left_path + right_path)

This helper function performs depth-first search on the binary tree

Recursively find the longest path in the left and right subtree

we can extend the univalue path by 1; otherwise, we reset to 0.

Return the longest path from this node to its children that continues

// If the left child exists and has the same value, extend the left path

longestPath = Math.max(longestPath, leftPathLength + rightPathLength);

// The value of the node.

// Constructor to initialize a node with a value and no children.

int longestPath; // Stores the length of the longest univalue path.

// Pointer to the left child node.

// Constructor to initialize a node with a value and left and right children.

// Public method to initiate the process and return the longest univalue path.

// Helper DFS method to compute the univalue path length for each subtree.

if (!node) return 0; // Base case: if the node is null, return 0.

// Recursively find the longest univalue path in the left and right subtrees.

// If the left child exists and has the same value, increment leftPathLength.

// If the right child exists and has the same value, increment rightPathLength.

// Return the longer path of the two: either leftPathLength or rightPathLength.

longestPath = std::max(longestPath, leftPathLength + rightPathLength);

leftPathLength = (node->left && node->left->val == node->val) ? leftPathLength + 1 : 0;

rightPathLength = (node->right && node->right->val == node->val) ? rightPathLength + 1 : 0;

// Update the longestPath with the maximum value of itself and the sum of left and right paths.

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

// Pointer to the right child node.

return Math.max(leftPathLength, rightPathLength);

// Constructor to initialize a node with no children.

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

int longestUnivaluePath(TreeNode* root) {

int leftPathLength = dfs(node->left);

int rightPathLength = dfs(node->right);

return std::max(leftPathLength, rightPathLength);

// Define the structure for a binary tree node with numeric values

longestPath = 0;

return longestPath;

int dfs(TreeNode* node) {

dfs(root);

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

// If the right child exists and has the same value, extend the right path

// Update the longest path found so far considering both leftPath and rightPath

leftPathLength = (node.left != null && node.left.value == node.value) ? leftPathLength + 1 : 0;

// The returned value should be the longest single side path (not combined) from this node

rightPathLength = (node.right != null && node.right.value == node.value) ? rightPathLength + 1 : 0;

the univalue path (only the longer one of left or right can be part of the path).

If the current node's value matches that of its left child,

```
/**
    * Definition for a binary tree node.
    */
   class TreeNode {
       int value; // Node value
       TreeNode left; // Left child
       TreeNode right; // Right child
 8
       // Default constructor
 9
       TreeNode() {}
10
11
12
       // Constructor with value
13
       TreeNode(int value) { this.value = value; }
14
15
       // Constructor with value, left child, and right child
       TreeNode(int value, TreeNode left, TreeNode right) {
16
            this.value = value;
17
           this.left = left;
18
            this.right = right;
19
20
21 }
22
   class Solution {
24
       private int longestPath; // To store the result of the longest univalue path
25
26
       public int longestUnivaluePath(TreeNode root) {
27
            longestPath = 0; // Initialize the longest path length to 0
28
            depthFirstSearch(root); // Begin DFS from the root
29
            return longestPath; // Return the maximum length found
30
31
32
       // Helper method to perform DFS on each node
       private int depthFirstSearch(TreeNode node) {
33
           if (node == null) {
34
35
               // If the node is null, there is no path
36
               return 0;
37
38
           // Recursively find the longest path in the left subtree
39
            int leftPathLength = depthFirstSearch(node.left);
           // Recursively find the longest path in the right subtree
40
            int rightPathLength = depthFirstSearch(node.right);
41
```

50 Typescript Solution

interface TreeNode {

left: TreeNode | null;

right: TreeNode | null;

val: number;

};

```
/**
    * Computes the length of the longest path where each node in the path has the same value.
    * This path may or may not pass through the root.
11
    * @param {TreeNode | null} root - The root node of the binary tree.
    * @return {number} The length of the longest univalue path.
14
    */
   function longestUnivaluePath(root: TreeNode | null): number {
     // If the tree is empty, return zero as there are no paths.
     if (root === null) {
       return 0;
19
20
     // Initialize the result variable to store the maximum path length found so far.
21
22
     let result = 0;
23
24
     /**
25
      * Performs depth-first search on the binary tree to find the longest univalue path.
26
27
      * @param {TreeNode | null} node - The current node being explored.
      * @param {number} targetValue - The target value nodes in the path should have.
28
      * @return {number} The length of the longest univalue path through the current node.
30
      */
31
     function dfs(node: TreeNode | null, targetValue: number): number {
32
       // Base case: if the node is null, return zero as there is no path.
33
       if (node === null) {
34
         return 0;
35
36
37
       // Recursively find the longest univalue path in the left subtree.
38
       let leftPathLength = dfs(node.left, node.val);
       // Recursively find the longest univalue path in the right subtree.
39
       let rightPathLength = dfs(node.right, node.val);
40
42
       // Update the result with the maximum path length by summing left and right paths if they are univalue.
       result = Math.max(result, leftPathLength + rightPathLength);
43
44
       // If the current node's value matches the target value,
45
       // return the longest path length including the current node;
46
       // otherwise, return zero as the path breaks here.
       if (node.val === targetValue) {
         return 1 + Math.max(leftPathLength, rightPathLength);
49
50
51
       return 0;
52
53
     // Start the depth-first search from the root.
54
     dfs(root, root.val);
55
56
57
     // The result contains the longest univalue path length.
     return result;
58
59
60
Time and Space Complexity
```

The time complexity of the longestUnivaluePath function is O(n) where n is the number of nodes in the binary tree. This is because the dfs function is called once for every node in the tree, and each call to the dfs function processes the current node independently of the other nodes. Since the algorithm must visit all nodes to determine the longest univalue path, it results in a linear time

complexity relative to the number of nodes.

Time Complexity

Space Complexity The space complexity of the longestUnivaluePath function is O(h) where h is the height of the tree. This is due to the recursive nature of the dfs function, which results in a call stack proportional to the height of the tree in the worst case (e.g., the tree is skewed). For a balanced tree, the height h would be log(n), leading to a space complexity of O(log(n)). However, in the worst case

(a skewed tree), the space complexity would be O(n), where n is the total number of nodes.