101. Symmetric Tree

Depth-First Search Breadth-First Search

Problem Description

Easy

The problem provided is about checking symmetry in a binary tree. Specifically, you are given a binary tree's root node, and your task is to determine if the tree is a mirror image of itself when divided down the middle. Essentially, this is asking whether the left and right sub-trees of the tree are mirror images of each other.

Binary Tree

Intuition

left sub-tree with the right sub-tree to ensure they are mirrors of each other. This is done by checking that: 1. The value of the current node in the left sub-tree is equal to the value of the current node in the right sub-tree.

To solve this problem, the idea is to use a <u>Depth-First Search</u> (DFS) approach. The solution involves recursively comparing the

- 2. The left child of the left sub-tree is a mirror of the right child of the right sub-tree.
- 3. The right child of the left sub-tree is a mirror of the left child of the right sub-tree.

branch before backtracking. Here, DFS is applied recursively through a helper function named dfs.

We can start this process by comparing the root node with itself, initiating symmetry checks between its left and right child

the two nodes are not equal, then the tree is not symmetric.

The recursion continues this process of mirroring the checks to progressively lower levels of the tree until all mirrored nodes pass the comparisons, or a pair of nodes fails, which indicates the tree is not symmetric.

nodes. If both nodes are null, it means we are comparing leaves, and thus they are symmetric. If only one is null or the values of

Solution Approach

The solution to the given problem relies on a <u>Depth-First Search</u> (DFS) algorithm, which explores as far as possible along each

The dfs function is designed to take two nodes as arguments—root1 and root2. Initially, these arguments are both set to the

root of the whole tree since we start by comparing the tree to itself. Here's how the dfs function works: Base case for null nodes: If both root1 and root2 are None, this means that both branches being compared have reached the

end simultaneously, indicating a mirrored structure at this branch level. So, it returns True. Case for asymmetry: If one of the nodes is None (while the other isn't), or if the values of the two nodes are not equal, the function identifies a break in symmetry and returns False.

- Recursive calls: If neither of the above cases is true, the dfs function calls itself twice more: once comparing the left child of root1 with the right child of root2, and then comparing the right child of root1 with the left child of root2. This is the crux of the mirroring check, making sure that each "mirror" position across the two sub-trees holds an equivalent value.
- return True, ensuring that all parts of the tree adhere to the symmetry condition. Finally, the isSymmetric function of the Solution class makes the initial call to dfs using the root as both arguments. If the entire tree is symmetric, the function will eventually return True; if any asymmetry is found at any level, it will return False.

The dfs function uses a logical AND && to combine the results of its recursive calls. Both calls must return True for the function to

The overall time complexity of the algorithm is O(n), where n is the number of nodes in the tree, because each node is visited once. The space complexity is also 0(n) for the call stack due to recursion, which in the worst case, could be the height of the

Let's consider a simple, symmetric binary tree to illustrate the solution approach. Here is the tree structure:

Now let's walk through the dfs function with this tree to see how it validates symmetry: 1. The isSymmetric function starts and calls dfs, passing the root node as both root1 and root2, since we're comparing the tree

with itself.

True.

Python

class TreeNode:

are consistently equal in value.

self.left = left

self.right = right

Example Walkthrough

Two dfs calls are made: one for root1's left (Node 2) and root2's right (also Node 2), the other for root1's right (Node 2) and root2's left (also Node 2). Both pairs are identical, so we proceed.

Now, from the first recursive call of dfs, two more recursive calls are made:

and the equality check condition. Hence, every recursive call returns True.

The isSymmetric function concludes that the tree is symmetric.

Compare root1's left (Node 4) and root2's right (Node 4).

Compare root1's right (Node 3) and root2's left (Node 3).

tree. For a balanced tree, this would result in a space complexity of O(log n) due to its height.

 Compare root1's left (Node 3) and root2's right (Node 3). Compare root1's right (Node 4) and root2's left (Node 4). Simultaneously, from the second recursive call of dfs, another two recursive calls are made:

As none of the root1 and root2 are null and their values are equal (value 1), the dfs function continues to the recursive calls.

Since the && operator is used to combine the results, and all recursive calls returned True, the initial call to dfs also returns

This tree has passed all the checks outlined in the approach; each node on the left has a corresponding node with equal value on

the right, and vice versa. The recursion accurately captures this mirror image property, ensuring that a node and its "mirror" node

All subsequent recursive calls find that the nodes are either simultaneously null or with equal values, satisfying the base case

- Solution Implementation
- # Definition for a binary tree node. def __init__(self, val=0, left=None, right=None): self.val = val

:return: bool, true if the tree is symmetric, false otherwise

:param node1: TreeNode, root of the first tree or subtree

:param node2: TreeNode, root of the second tree or subtree

if node1 is None or node2 is None or node1.val != node2.val:

Helper function that checks if two trees are mirror images of each other.

:return: bool, true if both trees are mirror images, false otherwise

Both nodes are None, meaning both subtrees are empty, thus symmetric

If only one of the nodes is None or if the values don't match, the subtrees aren't mirrors

return is_mirror(node1.left, node2.right) and is_mirror(node1.right, node2.left)

Start the recursion with root as both parameters, as the check is for the tree with itself

def is_mirror(node1: TreeNode, node2: TreeNode) -> bool:

Check the outer and inner pairs of subtrees

// The value of the node

TreeNode(int value, TreeNode leftChild, TreeNode rightChild) {

* @return true if the tree is symmetric, false otherwise.

* Determines if a binary tree is symmetric around its center (mirrored).

TreeNode left; // Pointer to the left child

// Constructors for creating a tree node

TreeNode(int value) { this.val = value; }

TreeNode right; // Pointer to the right child

if node1 is None and node2 is None:

return True

return False

return is_mirror(root, root)

class Solution: def isSymmetric(self, root: TreeNode) -> bool: Check if a binary tree is symmetric around its center. A binary tree is symmetric if the left subtree is a mirror reflection of the right subtree. :param root: TreeNode

```
Java
/**
* Definition for a binary tree node.
```

class TreeNode {

int val;

class Solution {

/**

TreeNode() {}

this.val = value;

this.left = leftChild;

this.right = rightChild;

* @param root The root of the tree.

```
public boolean isSymmetric(TreeNode root) {
       // Start DFS from the root for both subtrees for comparison.
       return isMirror(root, root);
    /**
    * Helper method to perform a DFS to check for symmetry by comparing nodes.
    * @param node1 The current node from the first subtree.
    * @param node2 The current node from the second subtree.
    * @return true if the two subtrees are mirrors of each other, false otherwise.
    private boolean isMirror(TreeNode node1, TreeNode node2) {
       // Both nodes are null, meaning this branch is symmetric.
       if (node1 == null && node2 == null) {
            return true;
       // If only one of the nodes is null, or their values differ,
       // the tree cannot be symmetric.
       if (node1 == null || node2 == null || node1.val != node2.val) {
            return false;
       // Continue to compare the left subtree of node1 with the right subtree of node2
       // and the right subtree of nodel with the left subtree of node2. Both comparisons
       // must be true for the subtree to be symmetric.
       return isMirror(node1.left, node2.right) && isMirror(node1.right, node2.left);
C++
// Definition for a binary tree node.
struct TreeNode {
    int val;
   TreeNode *left;
   TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
class Solution {
public:
   // Function to check whether a binary tree is symmetric around its center
    bool isSymmetric(TreeNode* root) {
       // Define a lambda function to recursively check the symmetry
        function<bool(TreeNode*, TreeNode*)> checkSymmetry = [&](TreeNode* leftSubtree, TreeNode* rightSubtree) -> bool {
           // If both subtrees are null, they are symmetric
            if (!leftSubtree && !rightSubtree) return true;
           // If one subtree is null or the values are different, they are not symmetric
           if (!leftSubtree || !rightSubtree || leftSubtree->val != rightSubtree->val) return false;
           // Recursively check the symmetry of subtrees
           // The left subtree of the left node and the right subtree of the right node
           // The right subtree of the left node and the left subtree of the right node
            return checkSymmetry(leftSubtree->left, rightSubtree->right) &&
                   checkSymmetry(leftSubtree->right, rightSubtree->left);
        };
       // Initialize the recursive function with the root of the tree
       return checkSymmetry(root, root);
};
TypeScript
// TreeNode definition for a binary tree node.
class TreeNode {
  val: number;
  left: TreeNode | null;
```

/** * Given the root of a binary tree, determine if it is a mirror of itself

return true;

return false;

/**

*/

right: TreeNode | null;

this.val = val === undefined ? 0 : val;

* subtrees are mirrors of each other.

this.left = left === undefined ? null : left;

this.right = right === undefined ? null : right;

* @param subtreeOne The root node of the first subtree.

if (subtreeOne == null && subtreeTwo == null) {

* @param subtreeTwo The root node of the second subtree.

* A function that performs a depth-first search to check if two

// If both subtrees are null, they are symmetric (base case).

* @returns A boolean indicating whether the subtrees are symmetric.

constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {

const depthFirstSearch = (subtreeOne: TreeNode | null, subtreeTwo: TreeNode | null): boolean => {

// If one is null and the other is not, or if the values are different, they are not symmetric.

// and the right subtree of the first subtree with the left subtree of the second subtree.

// Recursively compare the left subtree of the first subtree with the right subtree of the second subtree

return depthFirstSearch(subtreeOne.left, subtreeTwo.right) && depthFirstSearch(subtreeOne.right, subtreeTwo.left);

if (subtreeOne == null || subtreeTwo == null || subtreeOne.val != subtreeTwo.val) {

```
* (i.e., symmetric around its center).
   * @param root The root node of the binary tree.
   * @returns A boolean indicating whether the binary tree is symmetric.
  function isSymmetric(root: TreeNode | null): boolean {
    // Handle the edge case where the tree is empty.
    if (root == null) {
      return true;
    // Use helper function to compare the left and right subtree of the root.
    return depthFirstSearch(root.left, root.right);
class TreeNode:
   # Definition for a binary tree node.
   def ___init___(self, val=0, left=None, right=None):
       self.val = val
       self.left = left
       self.right = right
class Solution:
   def isSymmetric(self, root: TreeNode) -> bool:
       Check if a binary tree is symmetric around its center.
        A binary tree is symmetric if the left subtree is a mirror reflection of the right subtree.
        :param root: TreeNode
        :return: bool, true if the tree is symmetric, false otherwise
       def is_mirror(node1: TreeNode, node2: TreeNode) -> bool:
            Helper function that checks if two trees are mirror images of each other.
            :param node1: TreeNode, root of the first tree or subtree
            :param node2: TreeNode, root of the second tree or subtree
            :return: bool, true if both trees are mirror images, false otherwise
           # Both nodes are None, meaning both subtrees are empty, thus symmetric
            if node1 is None and node2 is None:
               return True
           # If only one of the nodes is None or if the values don't match, the subtrees aren't mirrors
            if node1 is None or node2 is None or node1.val != node2.val:
               return False
            # Check the outer and inner pairs of subtrees
            return is_mirror(node1.left, node2.right) and is_mirror(node1.right, node2.left)
       # Start the recursion with root as both parameters, as the check is for the tree with itself
        return is_mirror(root, root)
```

Time Complexity The time complexity of the provided code is O(n), where n is the number of nodes in the binary tree. This is because the

Time and Space Complexity

and the right subtree. **Space Complexity**

The space complexity of the code is primarily determined by the recursion stack used in the dfs function. In the worst-case

recursive function dfs visits every node exactly once in the case of a perfectly symmetrical tree, checking symmetry for the left

scenario (a completely balanced tree), the height of the tree will be log(n) (since every level of the tree is fully filled), resulting in a space complexity of $O(\log(n))$. However, in the worst-case scenario of an unbalanced tree (such as a degenerate tree where every node only has one child), the space complexity would be O(n) because the call stack would grow linearly with the number of nodes, as the tree would

effectively become a linked list.