**Diameter of Binary Tree**

Given the root of a binary tree, return *the length of the****diameter****of the tree*.

The **diameter** of a binary tree is the **length** of the longest path between any two nodes in a tree. This path may or may not pass through the root.

The **length** of a path between two nodes is represented by the number of edges between them.

**Example 1:**

Shape

Description automatically generated

**Input:** root = [1,2,3,4,5]

**Output:** 3

**Explanation:** 3 is the length of the path [4,2,1,3] or [5,2,1,3].

**Example 2:**

**Input:** root = [1,2]

**Output:** 1

**Constraints:**

* The number of nodes in the tree is in the range [1, 104].
* -100 <= Node.val <= 100

/\*\*

\* Definition for a binary tree node.

\* function TreeNode(val, left, right) {

\* this.val = (val===undefined ? 0 : val)

\* this.left = (left===undefined ? null : left)

\* this.right = (right===undefined ? null : right)

\* }

\*/

/\*\*

\* @param {TreeNode} root

\* @return {number}

\*/

var diameterOfBinaryTree = function(root) {

};

Solution

Approach 1: Depth-first Search

**Intuition**

Within this article, we will refer to a leaf node as any node that has 1 degree, including the root node if it has 1 degree or less.

The key observation to make is: *the longest path has to be between two leaf nodes*. We can prove this with contradiction. Imagine that we have found the longest path, and it is **not** between two leaf nodes. We can extend that path by 1, by adding the child node of one of the end nodes (as at least one must have a child, given that they aren't both leaves). This contradicts the fact that our path is the longest path. Therefore, the longest path must be between two leaf nodes.

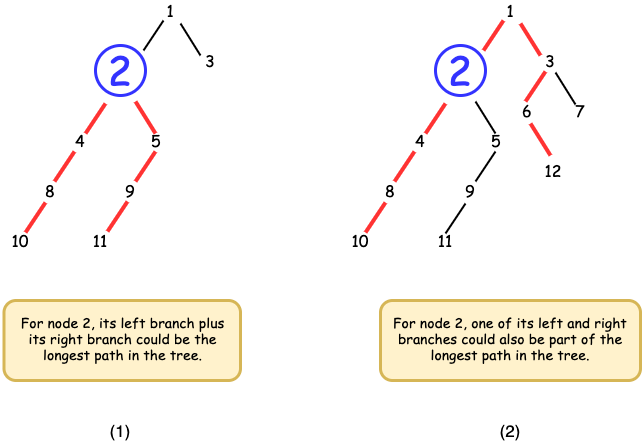
Moreover, we know that in a tree, nodes are only connected with their parent node and 2 children. Therefore we know that the longest path in the tree would consist of a node, its longest left branch, and its longest right branch. So, our algorithm to solve this problem will find the node where the sum of its longest left and right branches is maximized. This would hint at us to apply Depth-first search (DFS) to count each node's branch lengths, because it would allow us to dive deep into the leaves first, and then start counting the edges upwards.

DFS is a widely-used graph traversal algorithm. If you are not familiar with it, feel free to visit our [Explore Cards](https://leetcode.com/explore/learn/card/data-structure-tree/) where you will see different ways to traverse a binary tree with DFS including preorder, inorder, postorder :)

Let's try to be more specific about how to apply DFS to this question. To count the lengths of each node's left and right branches, we can implement a recursion function longestPath which takes a TreeNode as input and returns the longest path from it to the leaf node. It will recursively visit children nodes and retrieve the longest paths from them to the leaf first, and then add 1 to the longer one before returning it as the longest path.

In the midst of DFS, we also need to take the following two cases into account:

1. the current node's both left and right branches might be a part of the longest path;
2. one of the current node's left/right branches might be a part of the longest path.



*Figure 1. Two cases of the longest path.*

You will see we are going to address them by 1) applying DFS to recursively find the longest branches starting with the node's left and right children; 2) initializing a global variable diameter to keep track of the longest path and updating it at each node with the sum of the node's left and right branches; 3) returning the length of the longest branch between a node's left and right branches.

**Algorithm**

* Initalize an integer variable diameter to keep track of the longest path we find from the DFS.
* Implement a recursive function longestPath which takes a TreeNode as input. It should recursively explore the entire tree rooted at the given node. Once it's finished, it should return the longest path out of its left and right branches:
  + if node is None, we have reached the end of the tree, hence we should return 0;
  + we want to recursively explore node's children, so we call longestPath again with node's left and right children. In return, we get the longest path of its left and right children leftPath and rightPath;
  + if leftPath plus rightPath is longer than the current longest diameter found, then we need to update diameter;
  + finally, we return the longer one of leftPath and rightPath. Remember to add 11 as the edge connecting it with its parent.
* Call longestPath with root.

class Solution {

private int diameter;

public int diameterOfBinaryTree(TreeNode root) {

diameter = 0;

longestPath(root);

return diameter;

}

private int longestPath(TreeNode node){

if(node == null) return 0;

// recursively find the longest path in

// both left child and right child

int leftPath = longestPath(node.left);

int rightPath = longestPath(node.right);

// update the diameter if left\_path plus right\_path is larger

diameter = Math.max(diameter, leftPath + rightPath);

// return the longest one between left\_path and right\_path;

// remember to add 1 for the path connecting the node and its parent

return Math.max(leftPath, rightPath) + 1;

}

}

**Complexity Analysis**

Let N*N* be the number of nodes in the tree.

* Time complexity: O(N)*O*(*N*). This is because in our recursion function longestPath, we only enter and exit from each node once. We know this because each node is entered from its parent, and in a tree, nodes only have one parent.
* Space complexity: O(N)*O*(*N*). The space complexity depends on the size of our implicit call stack during our DFS, which relates to the height of the tree. In the worst case, the tree is skewed so the height of the tree is O(N)*O*(*N*). If the tree is balanced, it'd be O(\log N)*O*(log*N*).