543. Diameter of Binary Tree

<https://leetcode.com/problems/diameter-of-binary-tree/>

1. **Listen**

**Problem Statement**

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

**Input**

root of a binary tree

**Goal**

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.

find the node in the tree that has the maximum sum of its left and right subtrees

**Return**

return the **length** of the***diameter***of the tree.

1. **Example**

**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 range from [1, 104].
* -100 <= Node.val <= 100

**Test Cases:**

* Single node
* Longest path does not pass-through root
* Unbalanced

1. **Brute Force**

**Solution 1: Top-Down Calculations**

**Goal**

* Our Goal is to find the diameter of a binary tree.

**Definitions**

* Let’s establish some definitions:
  + **Diameter:**

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**.

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

**Algorithm**

* For any node in a tree, let’s call it *n*, we can treat *n* as the meeting point between a left path (left subtree) and a right path (right subtree).
* Therefore, if we find the **maximum depth** of the left and right subtrees, we can calculate the **path length for *n***.
* We can define this as Current Diameter = leftSubtreeHeight + rightSubtreeHeight
* We repeat this calculation for every node in the tree, *n*, while keeping track of the Maximum Diameter of all nodes.
* We would need to repeat this routine for every node in the tree, because there are no guarantees that any node n will have deep left and right subtrees.

**Example:**

Text, letter

Description automatically generated

* This is a perfect example of why the longest diameter (longest path between left and right subtrees) **may or may not pass through the root**.
* If we were to calculate the maximum depth for the root’s subtrees, we would find that
  + the left subtree has a depth of 1

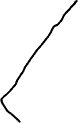
and

* + the right subtree has a depth of 6
* Adding these together gives us the total path length, which is 7.
* If we call total path length L, then we see it matches our definition for the **length** of a path, L-1.
* However, this is not maximum possible diameter of the tree.
* The actual *n* meeting point node would be -9.
* Why?
* Because although -9 is not the root, it has the longest left and right subtrees compared to every other node in the tree.

-root’s Diameter

Text, letter

Description automatically generated

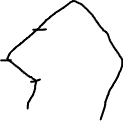
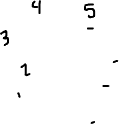


There are actually 3 leaf nodes with the same height here, so there are three possible paths.

-9’s Diameter

Text, letter

Description automatically generated



* This demonstrates that the longest path doesn't have to go through the root node.
* It can pass through the root node of some subtree, because the only constraint is that the path must be from one leaf node to another leaf node.
* The longest path that passes a given node as the ROOT node is D = left\_height + right\_height. So, you just calculate D for all nodes and output the max D.

**Runtime**

// find the diameter of current node

int dia = depth(root.left) + depth(root.right);

// find the diameter of the left child (recursively)

int ldia = diameterOfBinaryTree(root.left);

// find the diameter of the right child (recursively)

int rdia = diameterOfBinaryTree(root.right);

// return the maximum diameter out of current node, left child, and right child

return Math.max(dia,Math.max(ldia,rdia));

While this solution is intuitive, the performance is not good because of the overlapping subproblems when calculating depth.

**diameterOfBinaryTree** is called on every node.

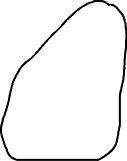
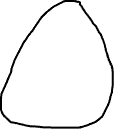
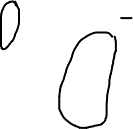
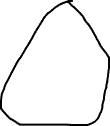
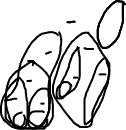
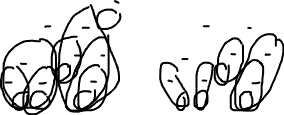
In each call, it traverses all descendants of that node to get the depth.

Text, letter

Description automatically generated Text, letter

Description automatically generated Text, letter

Description automatically generated



Take a look at how much repeated work is don’t just on the first three nodes.

* for root node, it is n => n + 1 - 2^0
* for all nodes on 2nd level, it is 2 \* (n - 1) / 2 => n - 1 => n + 1 - 2^1
* for all nodes on 3rd level, it is 4 \* (n - 3) / 4 => n - 3 => n + 1 - 2^2
* for all nodes on 4th level, it is 8 \* (n - 7) / 8 => n - 7 => n + 1 - 2^3  
  ...

If each call to maxDepth takes O(n) time, and we do this for n nodes, then the time complexity will take O(n^2) time.

1. **Optimize**

**Solution 2: Recursive Post Order**

**Overview**

We can make a recursive based algorithm that involves Post Order traversal.

Reason behind using Post Order comes from our intuition that if we can calculate a result only after knowing the maximum depth of the left and right subtrees.

This way, we work from the bottom up, instead of working from top to bottom. This reduces our runtime to O(n).

**Algorithm**

We keep track of two calculations for every function call.

1. **Diameter** of current node

Math.max(globalMax, leftMaxDepth + rightMaxDepth)

1. **Max** **height** of current node

max(leftMaxDepth, rightMaxDepth) + 1

Returning the height of the current node allows earlier nodes to use this in their calculation of their diameter.

return max(leftMaxDepth, rightMaxDepth) + 1

Therefore, we need a retainer to hold these return values.

int left = maxDepth(root.left);

int right = maxDepth(root.right);

**Runtime**

This will take O(N) time and O(H) space where H is the height.

If the tree is balanced, this will take O(logn) space.

**Example**

**Input:** [12, 2, 5, null, 10, 3, 4, 8, 11]

**Output:**

**One More Optimizations**

Where would we keep the global Maximum Depth variable?

One of Java’s quirks is that it does not allow you to pass primitives by reference.

So, we *could* simply keep a global variable that keeps track of this.

See the following example:

public class Solution

{

in max = 0;

public int diameterOfBinaryTree(TreeNode root) {

maxDepth(root, max);

return max;

}

private int maxDepth(TreeNode root) {

// code

}

}

However, this is messy, and often results in unclear/unsafe code.

A workaround we could do is simply create and pass an array that has a size of 1.

This way we can take advantage of the pass of reference natures of arrays, and act like we are passing a single primitive variable by reference.

We can pass this to the function as so:

public class Solution

{

public int diameterOfBinaryTree(TreeNode root) {

int[] max = new int[1];

maxDepth(root, max);

return max[0];

}

private int maxDepth(TreeNode root, int[] max) {

// code

}

}

1. **Walkthrough**

I already walked through this enough lol.

1. **Implement**

class Solution {

public int diameterOfBinaryTree(TreeNode root) {

int[] max = new int[] {Integer.MIN\_VALUE};

maxDepth(root, max);

return max[0];

}

public int maxDepth(TreeNode root, int[] max)

{

if(root == null) return 0;

int leftDepth = maxDepth(root.left, max);

int rightDepth = maxDepth(root.right, max);

// keep track of diameter

max[0] = Math.max(max[0], rightDepth + leftDepth);

// return height for previous function call's diameter calc

return Math.max(leftDepth, rightDepth) + 1;

}

}

