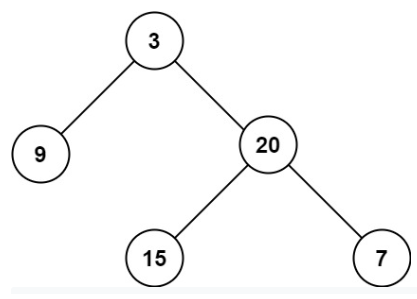
<https://leetcode.com/problems/maximum-depth-of-binary-tree/>

Given the root of a binary tree, return *its maximum depth*.

A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.

**Example 1:**



Input: root = [3,9,20,null,null,15,7]

Output: 3

**Example 2:**

Input: root = [1,null,2]

Output: 2

**Constraints:**

* The number of nodes in the tree is in the range [0, 104].
* -100 <= Node.val <= 100

**Attempt 1: 2022-11-09**

**Solution 1:  Recursive traversal (10 min)**

**Style 1: Divide and Conquer (so called Top Down DFS, 遍历法132: 1.base case -> 3.进行当前层的处理计算 -> 2.递归成为更小的问题), bottom level return 0 with +1 operation in conquer**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

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

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

public int maxDepth(TreeNode root) {

if(root == null) {

return 0; // not return depth

}

int left = maxDepth(root.left);

int right = maxDepth(root.right);

return Math.max(left, right) + 1; // + 1 in conquer

}

}

Time Complexity: O(n), where n is number of nodes in the Binary Tree

Space Complexity: O(n)

**Refer to**

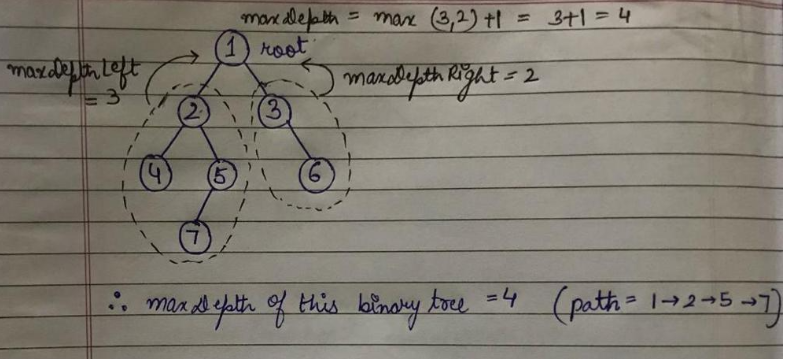
<https://leetcode.com/problems/maximum-depth-of-binary-tree/discuss/1770060/C%2B%2B-oror-Recursive-oror-DFS-oror-Example-Dry-Run-oror-Well-Explained>

**Recursive (DFS):**

Let's **redefine** the problem:So, the question says given the root of a binary tree, return the maximum depth of the tree. Max depth means the number of nodes along the longest path from root to farthest leaf node.

**Recursion:**

Lets have faith in recursion and assume that we are already given the maximum depth of root's left and right subtrees by recursion. So to find the maximum depth of this binary tree, we will have to take out the maximum of the 2 depths given to us by recursion, and add 1 to that to consider the current level i.e. root's level into our depth.



So basically, to find the maximum depth of the binary tree given, we mainly have to have do

int maxDepthLeft = maxDepth(root->left);

int maxDepthRight = maxDepth(root->right);

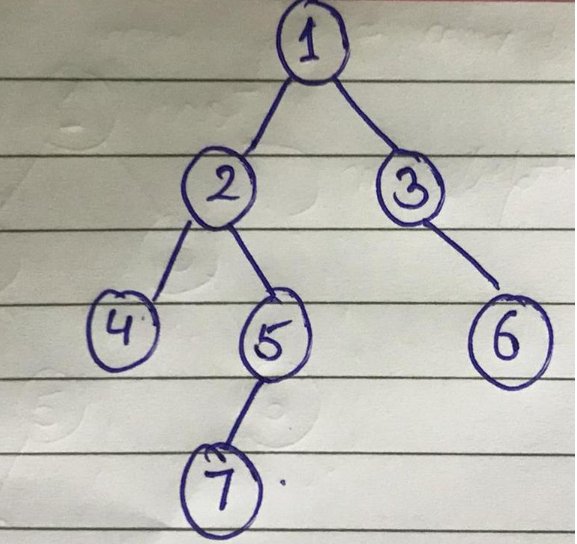
return max(maxDepthLeft, maxDepthRight) + 1;

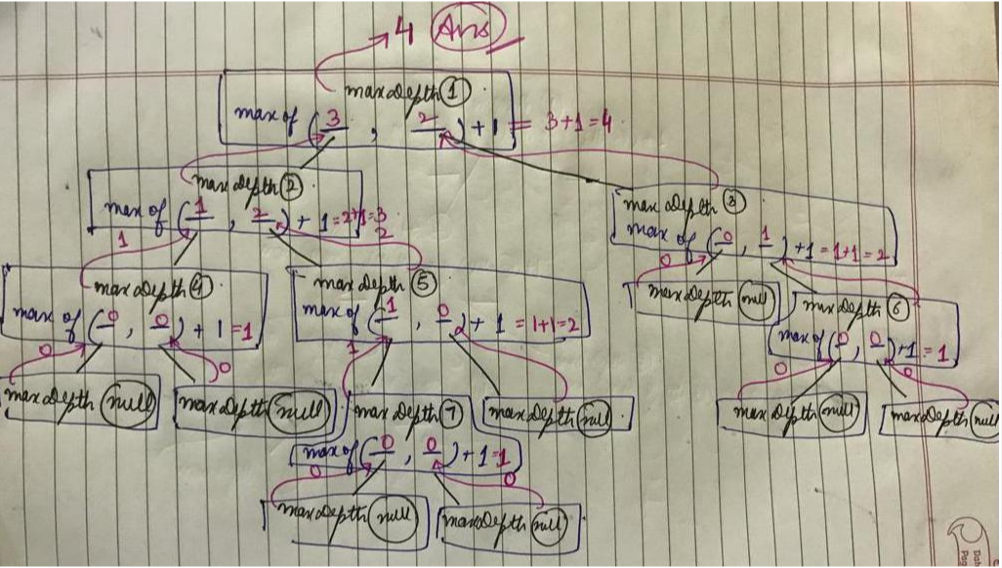
**Base Case:**

We can easily analyse that if we are at a leaf node as root, then its left and right subtrees will have 0 depth, and consecutively, this leaf node will have max depth of 1.

**Example (Dry Run):**

Lets take this example up and try running our approach on it.





**Code:**

int maxDepth(TreeNode\* root) {

if(!root) return 0;

int maxLeft = maxDepth(root->left);

int maxRight = maxDepth(root->right);

return max(maxLeft, maxRight)+1;

}

**Complexity:**

**TC - O(num of nodes)** as we are traversing all the nodes of the tree

**SC - O(height of the tree)** for the recursive stack

**Style 2: Divide and Conquer (so called Top Down DFS, 遍历法132: 1.base case -> 3.进行当前层的处理计算 -> 2.递归成为更小的问题), bottom level return 0 with +1 operation in divide**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

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

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

public int maxDepth(TreeNode root) {

return helper(root);

}

private int helper(TreeNode root) {

if(root == null) {

return 0; // not return depth

}

int left = helper(root.left) + 1; // + 1 in divide

int right = helper(root.right) + 1; // + 1 in divide

return Math.max(left, right);

}

}

**Style 3: Divide and Conquer (so called Top Down DFS, 遍历法132: 1.base case -> 3.进行当前层的处理计算 -> 2.递归成为更小的问题), bottom level return 'depth' since '+1' operation not in DFS three steps (divide -> process -> conquer) but only happen on parameter that passed in recursion function, since no actual operation to update 'depth' during DFS, to reflect change happen on 'depth' in the parameter, requires return 'depth' to pass in next recursion**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

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

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

public int maxDepth(TreeNode root) {

return helper(root, 0);

}

// Additional parameter 'depth'

private int helper(TreeNode root, int depth) {

if(root == null) {

return depth; // not return 0

}

int left = helper(root.left, depth + 1); // + 1 on parameter that passed in recursion function, not in divide -> process -> conquer steps

int right = helper(root.right, depth + 1); // + 1 on parameter that passed in recursion function, not in divide -> process -> conquer steps

return Math.max(left, right);

}

}

**Style 4: Classic recursive traversal (so called Top Down DFS, 遍历法132: 1.base case -> 3.进行当前层的处理计算 -> 2.递归成为更小的问题)**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

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

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

int max = 0;

public int maxDepth(TreeNode root) {

// Consider root as depth 1

helper(root, 1);

return max;

}

private void helper(TreeNode root, int curDepth) {

if(root == null) {

return;

}

max = Math.max(max, curDepth);

helper(root.left, curDepth + 1);

helper(root.right, curDepth + 1);

}

}

**Solution 2:  Iterative level order traversal as BFS (10 min)**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

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

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

public int maxDepth(TreeNode root) {

if(root == null) {

return 0;

}

Queue<TreeNode> q = new LinkedList<TreeNode>();

q.offer(root);

// Differ than L111/P8.5.Minimum Depth of Binary Tree 'int level = 1'

// because in minimum path problem we will directly return if no child

// for a node, it has no chance to touch 'level++', but in maximum path

// problem we will touch 'level++' without condition, so level initial=0

int level = 0;

while(!q.isEmpty()) {

int size = q.size();

for(int i = 0; i < size; i++) {

TreeNode node = q.poll();

if(node.left != null) {

q.offer(node.left);

}

if(node.right != null) {

q.offer(node.right);

}

}

level++;

}

return level;

}

}

Time Complexity: O(n), where n is number of nodes in the Binary Tree

Space Complexity: O(n)

**Refer to**

<https://leetcode.com/problems/maximum-depth-of-binary-tree/discuss/34195/Two-Java-Iterative-solution-DFS-and-BFS>

public int maxDepth(TreeNode root) {

if(root == null) {

return 0;

}

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(root);

int count = 0;

while(!queue.isEmpty()) {

int size = queue.size();

while(size-- > 0) {

TreeNode node = queue.poll();

if(node.left != null) {

queue.offer(node.left);

}

if(node.right != null) {

queue.offer(node.right);

}

}

count++;

}

return count;

}