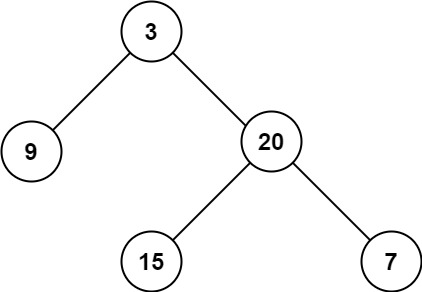
# Question

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

**Example 3:**

**Input:** root = []

**Output:** 0

**Example 4:**

**Input:** root = [0]

**Output:** 1

**Constraints:**

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

# Solution

**Tree definition**

First of all, here is the definition of the TreeNode which we would use.

|  |
| --- |
| /\* Definition for a binary tree node. \*/  public class TreeNode {  int val;  TreeNode left;  TreeNode right;  TreeNode(int x) {  val = x;  }  } |

#### **Approach 1: Recursion**

**Intuition** By definition, the maximum depth of a binary tree is the maximum number of steps to reach a leaf node from the root node.

From the definition, an intuitive idea would be to traverse the tree and record the maximum depth during the traversal.

**Algorithm**

One could traverse the tree either by Depth-First Search (DFS) strategy or Breadth-First Search (BFS) strategy. For this problem, either way would do. Here we demonstrate a solution that is implemented with the **DFS** strategy and **recursion**.

|  |
| --- |
| class Solution {  public int maxDepth(TreeNode root) {  if (root == null) {  return 0;  } else {  int left\_height = maxDepth(root.left);  int right\_height = maxDepth(root.right);  return java.lang.Math.max(left\_height, right\_height) + 1;  }  }  } |

**Complexity analysis**

* Time complexity : we visit each node exactly once, thus the time complexity is \mathcal{O}(N)O(*N*), where N*N* is the number of nodes.
* Space complexity : in the worst case, the tree is completely unbalanced, e.g. each node has only left child node, the recursion call would occur N*N* times (the height of the tree), therefore the storage to keep the call stack would be \mathcal{O}(N)O(*N*). But in the best case (the tree is completely balanced), the height of the tree would be \log(N)log(*N*). Therefore, the space complexity in this case would be \mathcal{O}(\log(N))O(log(*N*)).

#### **Approach 2: Tail Recursion + BFS**

One might have noticed that the above recursion solution is probably not the most optimal one in terms of the space complexity, and in the extreme case the overhead of call stack might even lead to stack overflow.

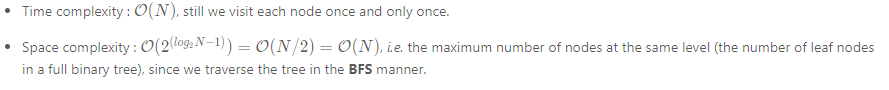
To address the issue, one can tweak the solution a bit to make it **tail recursion**, which is a specific form of recursion where the recursive call is the last action in the function.

The benefit of having tail recursion, is that for certain programming languages (e.g. C++) the compiler could optimize the memory allocation of call stack by reusing the same space for every recursive call, rather than creating the space for each one. As a result, one could obtain the constant space complexity \mathcal{O}(1)O(1) for the overhead of the recursive calls.

Here is a sample solution. Note that the optimization of tail recursion is not supported by Python or Java.

|  |
| --- |
| class Solution {  private:  // The queue that contains the next nodes to visit,  // along with the level/depth that each node is located.  queue<pair<TreeNode\*, int>> next\_items;  int max\_depth = 0;    /\*\*  \* A tail recursion function to calculate the max depth  \* of the binary tree.  \*/  int next\_maxDepth() {    if (next\_items.size() == 0) {  return max\_depth;  }    auto next\_item = next\_items.front();  next\_items.pop();  auto next\_node = next\_item.first;  auto next\_level = next\_item.second + 1;    max\_depth = max(max\_depth, next\_level);  // Add the nodes to visit in the following recursive calls.  if (next\_node->left != NULL) {  next\_items.push(make\_pair(next\_node->left, next\_level));  }  if (next\_node->right != NULL) {  next\_items.push(make\_pair(next\_node->right, next\_level));  }    // The last action should be the ONLY recursive call  // in the tail-recursion function.  return next\_maxDepth();  }    public:  int maxDepth(TreeNode\* root) {  if (root == NULL) return 0;    // clear the previous queue.  std::queue<pair<TreeNode\*, int>> empty;  std::swap(next\_items, empty);  max\_depth = 0;    // push the root node into the queue to kick off the next visit.  next\_items.push(make\_pair(root, 0));    return next\_maxDepth();  }  }; |

**Complexity analysis**



As one can see, this probably is not the best example to apply the tail recursion technique. Because though we did gain the constant space complexity for the recursive calls, we pay the price of \mathcal{O}(N)O(*N*) complexity to maintain the state information for recursive calls. This defeats the purpose of applying tail recursion.

However, we would like to stress on the point that tail recursion is a useful form of recursion that could eliminate the space overhead incurred by the recursive function calls.

Note: a function cannot be tail recursion if there are multiple occurrences of recursive calls in the function, even if the last action is the recursive call. Because the system has to maintain the function call stack for the sub-function calls that occur within the same function.

#### **Approach 3: Iteration**

**Intuition**

We could also convert the above recursion into iteration, with the help of the stack data structure. Similar with the behaviors of the function call stack, the stack data structure follows the pattern of FILO (First-In-Last-Out), i.e. the last element that is added to a stack would come out first.

With the help of the stack data structure, one could mimic the behaviors of function call stack that is involved in the recursion, to convert a recursive function to a function with iteration.

**Algorithm**

The idea is to keep the next nodes to visit in a stack. Due to the FILO behavior of stack, one would get the order of visit same as the one in recursion.

We start from a stack which contains the root node and the corresponding depth which is 1. Then we proceed to the iterations: pop the current node out of the stack and push the child nodes. The depth is updated at each step.

|  |
| --- |
| class Solution {  public int maxDepth(TreeNode root) {  LinkedList<TreeNode> stack = new LinkedList<>();  LinkedList<Integer> depths = new LinkedList<>();  if (root == null) return 0;  stack.add(root);  depths.add(1);  int depth = 0, current\_depth = 0;  while(!stack.isEmpty()) {  root = stack.pollLast();  current\_depth = depths.pollLast();  if (root != null) {  depth = Math.max(depth, current\_depth);  stack.add(root.left);  stack.add(root.right);  depths.add(current\_depth + 1);  depths.add(current\_depth + 1);  }  }  return depth;  }  }; |

**Complexity analysis**

* Time complexity : \mathcal{O}(N)O(*N*).
* Space complexity : in the worst case, the tree is completely unbalanced, e.g. each node has only left child node, the recursion call would occur N*N* times (the height of the tree), therefore the storage to keep the call stack would be \mathcal{O}(N)O(*N*). But in the average case (the tree is balanced), the height of the tree would be \log(N)log(*N*). Therefore, the space complexity in this case would be \mathcal{O}(\log(N))O(log(*N*)).