# Question

Given a n-ary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

*Nary-Tree input serialization is represented in their level order traversal, each group of children is separated by the null value (See examples).*

**Example 1:**



**Input:** root = [1,null,3,2,4,null,5,6]

**Output:** 3

**Example 2:**



**Input:** root = [1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14]

**Output:** 5

**Constraints:**

* The depth of the n-ary tree is less than or equal to 1000.
* The total number of nodes is between [0, 104].

# Solution

**Tree definition**

First of all, please refer to [this article](https://leetcode.com/articles/maximum-depth-of-binary-tree/) for the solution in case of binary tree. This article offers the same ideas with a bit of generalisation.

Here is the definition of the TreeNode which we would use.

|  |
| --- |
| // Definition for a Node.  class Node {  public int val;  public List<Node> children;  public Node() {}  public Node(int \_val,List<Node> \_children) {  val = \_val;  children = \_children;  }  }; |

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

**Algorithm**

The intuitive approach is to solve the problem by recursion. Here we demonstrate an example with the DFS (Depth First Search) strategy.

|  |
| --- |
| class Solution {  public int maxDepth(Node root) {  if (root == null) {  return 0;  } else if (root.children.isEmpty()) {  return 1;  } else {  List<Integer> heights = new LinkedList<>();  for (Node item : root.children) {  heights.add(maxDepth(item));  }  return Collections.max(heights) + 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 one 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: Iteration**

We could also convert the above recursion into iteration, with the help of stack.

The idea is to visit each node with the DFS strategy, while updating the max depth at each visit.

So 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(Node root) {  Queue<Pair<Node, Integer>> stack = new LinkedList<>();  if (root != null) {  stack.add(new Pair(root, 1));  }  int depth = 0;  while (!stack.isEmpty()) {  Pair<Node, Integer> current = stack.poll();  root = current.getKey();  int current\_depth = current.getValue();  if (root != null) {  depth = Math.max(depth, current\_depth);  for (Node c : root.children) {  stack.add(new Pair(c, current\_depth + 1));  }  }  }  return depth;  }  } |

**Complexity analysis**

* Time complexity : \mathcal{O}(N)O(*N*).
* Space complexity : \mathcal{O}(N)O(*N*).