**Solve Tree Problems Recursively**

* ["Top-down" Solution](https://leetcode.com/explore/learn/card/data-structure-tree/17/solve-problems-recursively/534/#top-down-solution)
* ["Bottom-up" Solution](https://leetcode.com/explore/learn/card/data-structure-tree/17/solve-problems-recursively/534/#bottom-up-solution)
* [Conclusion](https://leetcode.com/explore/learn/card/data-structure-tree/17/solve-problems-recursively/534/#conclusion)

In previous sections, we have introduced how to solve tree traversal problems recursively. Recursion is one of the most powerful and frequently used techniques for solving tree problems.

As we know, a tree can be defined recursively as a node(the root node) that includes a value and a list of references to children nodes. Recursion is one of the natural features of a tree. Therefore, many tree problems can be solved recursively. For each recursive function call, we only focus on the problem for the current node and call the function recursively to solve its children.

Typically, we can solve a tree problem recursively using a top-down approach or using a bottom-up approach.

"Top-down" Solution

"Top-down" means that in each recursive call, we will visit the node first to come up with some values, and pass these values to its children when calling the function recursively. So the "top-down" solution can be considered as a kind of **preorder** traversal. To be specific, the recursive function top\_down(root, params) works like this:

1. return specific value for null node

2. update the answer if needed // answer <-- params

3. left\_ans = top\_down(root.left, left\_params) // left\_params <-- root.val, params

4. right\_ans = top\_down(root.right, right\_params) // right\_params <-- root.val, params

5. return the answer if needed // answer <-- left\_ans, right\_ans

For instance, consider this problem: given a binary tree, find its maximum depth.

We know that the depth of the root node is 1. For each node, if we know its depth, we will know the depth of its children. Therefore, if we pass the depth of the node as a parameter when calling the function recursively, all the nodes will know their depth. And for leaf nodes, we can use the depth to update the final answer. Here is the pseudocode for the recursive function maximum\_depth(root, depth):

1. return if root is null

2. if root is a leaf node:

3. answer = max(answer, depth) // update the answer if needed

4. maximum\_depth(root.left, depth + 1) // call the function recursively for left child

5. maximum\_depth(root.right, depth + 1) // call the function recursively for right child

Here is an example to help you understand how it works:

1 / 12

We also provide C++ and Java codes for reference.

|  |
| --- |
| private int answer; // don't forget to initialize answer before call maximum\_depth  private void maximum\_depth(TreeNode root, int depth) {  if (root == null) {  return;  }  if (root.left == null && root.right == null) {  answer = Math.max(answer, depth);  }  maximum\_depth(root.left, depth + 1);  maximum\_depth(root.right, depth + 1);  } |

"Bottom-up" Solution

"Bottom-up" is another recursive solution. In each recursive call, we will firstly call the function recursively for all the children nodes and then come up with the answer according to the returned values and the value of the current node itself. This process can be regarded as a kind of **postorder** traversal. Typically, a "bottom-up" recursive function bottom\_up(root) will be something like this:

1. return specific value for null node

2. left\_ans = bottom\_up(root.left) // call function recursively for left child

3. right\_ans = bottom\_up(root.right) // call function recursively for right child

4. return answers // answer <-- left\_ans, right\_ans, root.val

Let's go on discussing the question about maximum depth but using a different way of thinking: for a single node of the tree, what will be the maximum depth x of the subtree rooted at itself?

If we know the maximum depth l of the subtree rooted at its **left** child and the maximum depth r of the subtree rooted at its **right** child, can we answer the previous question? Of course yes, we can choose the maximum between them and add 1 to get the maximum depth of the subtree rooted at the current node. That is x = max(l, r) + 1.

It means that for each node, we can get the answer after solving the problem for its children. Therefore, we can solve this problem using a "bottom-up" solution. Here is the pseudocode for the recursive function maximum\_depth(root):

1. return 0 if root is null // return 0 for null node

2. left\_depth = maximum\_depth(root.left)

3. right\_depth = maximum\_depth(root.right)

4. return max(left\_depth, right\_depth) + 1 // return depth of the subtree rooted at root

Here is an example to help you understand how it works:

1 / 25

We also provide C++ and Java codes for reference.

|  |
| --- |
| public int maximum\_depth(TreeNode root) {  if (root == null) {  return 0; // return 0 for null node  }  int left\_depth = maximum\_depth(root.left);  int right\_depth = maximum\_depth(root.right);  return Math.max(left\_depth, right\_depth) + 1; // return depth of the subtree rooted at root  } |

Conclusion

It is not easy to understand recursion and find out a recursive solution for the problem. It needs practice.

When you meet a tree problem, ask yourself two questions: Can you determine some parameters to help the node know its answer? Can you use these parameters and the value of the node itself to determine what should be the parameters passed to its children? If the answers are both yes, try to solve this problem using a "top-down" recursive solution.

Or, you can think of the problem in this way: for a node in a tree, if you know the answer of its children, can you calculate the answer of that node? If the answer is yes, solving the problem recursively using a bottom up approach might be a good idea.

In the following sections, we provide several classic problems for you to help you understand tree structure and recursion better.