Topic Introduction: Measuring Trees with Depth-First Search

Today, let’s take a stroll through the forest — the binary tree forest! We’re going to explore how to *measure* trees: not with a tape measure, but with code. Our toolkit? Depth-First Search (DFS).

What is DFS?
DFS is a tree (or graph) traversal technique. Starting at the root, you explore as far down one branch as possible before backtracking. In code, DFS is often implemented recursively: call yourself on the left child, then on the right, and combine results.

Why is DFS useful?
With trees, DFS is perfect for problems where you need to process or aggregate information from subtrees — such as counting nodes, finding depths, or locating paths. In interviews, DFS is a go-to tool for any problem that asks about *all* root-to-leaf paths, subtree properties, or traversals.

Quick Example:
Suppose you want to sum all values in a binary tree. DFS lets you visit every node, adding the values together as you return from each recursive call.

```
def sum_tree(node):
    if not node:
        return 0
    return node.val + sum_tree(node.left) + sum_tree(node.right)
```

Today, we’ll use DFS to “measure” trees in three different ways!

Meet the Problems: Depth and Diameter

Let’s look at three classic "tree measurement" problems:

- **Maximum Depth of Binary Tree** (LeetCode 104)
- **Minimum Depth of Binary Tree** (LeetCode 111)
- **Diameter of Binary Tree** (LeetCode 543)

Why are these grouped together?
All three require some kind of measurement on a binary tree, and each can be solved efficiently with DFS.

- *Maximum depth* asks for the longest root-to-leaf path.
- *Minimum depth* asks for the shortest root-to-leaf path.
- *Diameter* asks for the longest path between any two nodes (not necessarily passing through the root).

By mastering these, you’ll gain deep insight into how DFS can be used to aggregate information from subtrees — a core skill for many coding interviews.

Let’s start measuring!

Problem 1: Maximum Depth of Binary Tree

Problem Statement (in plain English):

Given the root of a binary tree, find its maximum depth — that is, the number of nodes along the longest path from the root down to a leaf.

LeetCode 104: Maximum Depth of Binary Tree

Example:

Let’s say your tree looks like this:

```
      3
     / \
    9  20
      /  \
     15   7
```

• The longest root-to-leaf path is: 3 → 20 → 15 (or 3 → 20 → 7).

• The maximum depth is **3**.

How to think about it:

• For each node, the maximum depth is 1 (itself) plus the maximum of the depths of its left and right children.

• If you reach a null node, its depth is 0.

Pen-and-paper tip:

Draw the tree and label each node with the depth you’d return for it. This helps you see the recursive pattern in action.

Try this test case yourself:

```
      1
     /
    2
   /
  3
```

Expected output: **3**

Brute Force Approach:

Traverse every path from root to every leaf, tracking the length of each. Take the maximum.

• Time complexity: O(N) (since you visit each node once)

• Space complexity: O(H) (H = height of tree, due to recursion stack)

Optimal Approach:

Let’s use recursive DFS:

• If the node is null, return 0.

• Compute the depth of the left subtree.

• Compute the depth of the right subtree.

• Return 1 (for the current node) plus the greater of the left/right depths.

Here’s the code, step by step:

```
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def maxDepth(root):
    # Base case: if the node is null, depth is 0
    if not root:
        return 0
    # Recursively find the depth of left and right subtrees
    left_depth = maxDepth(root.left)
    right_depth = maxDepth(root.right)
    # The depth for this node is 1 + max(left, right)
    return 1 + max(left_depth, right_depth)
```

- **Time complexity:** O(N) — every node is visited once.
- **Space complexity:** O(H) — height of the tree, for recursion stack.

Explain the parts:

- The function calls itself for left and right children.
- When it hits a leaf's child (null), it returns 0.
- As recursion unwinds, each node returns 1 + the largest depth from its children.
- The final answer (at the root) is the maximum depth.

Trace with our example:

For the tree:

```
      3
     / \
    9  20
      /  \
     15   7
```

- At node 15 and 7: left and right are null, so depth = 1.
- At node 20: left=1, right=1, so depth = 1 + max(1,1) = 2.
- At node 9: left and right null -> depth = 1.
- At root (3): left=1, right=2, so depth = 1 + max(1,2) = **3**.

Try this test case:

```
   1
    \
     2
      \
       3
```

Expected output: **3**

Take a moment:

Try drawing the recursion tree for the above test case!

Optional Challenge:

Did you know you can also solve this iteratively using a stack or queue (BFS)? Try it after you finish!

Problem 2: Minimum Depth of Binary Tree

How is this different?

Now, we want the *shortest* path from root to any leaf node.

LeetCode 111: Minimum Depth of Binary Tree

Example:

```
      1
     / \
    2   3
```

- Paths: 1 → 2 and 1 → 3, both length 2.
- Minimum depth: **2**

Why is this trickier?

If a node has only one child, you must not count the missing side as depth 0. You need to ignore null children when considering minimums.

Pen-and-paper tip:

Write out all root-to-leaf paths. Minimum depth is the length of the shortest.

Try this test case:

```
    1
   /
  2
   \
    3
```

Expected output: **3** (since the only leaf is 3, and path is 1 → 2 → 3)

Brute Force:

List all paths from root to leaves, track their lengths, pick the smallest.

- Time complexity: O(N)
- Space complexity: O(H)

Optimal Approach:

We'll use DFS, but be careful to handle nodes with only one child.

Step-by-step:

- If node is null, return 0.
- If node has no left and right, it's a leaf: return 1.
- If one child is null, recurse on the *other* child.

- If both children exist, take the minimum.

Pseudocode:

```
function minDepth(node):
    if node is null:
        return 0
    if node.left is null and node.right is null:
        return 1
    if node.left is null:
        return 1 + minDepth(node.right)
    if node.right is null:
        return 1 + minDepth(node.left)
    return 1 + min(minDepth(node.left), minDepth(node.right))
```

Test case walk-through:

For

```
    1
   /
  2
   \
    3
```

- Root 1: left exists, right is null
- Go left to 2: left is null, right exists
- Go right to 3: it's a leaf, return 1
- Node 2: left is null, so return 1 + minDepth(right) = 2
- Node 1: right is null, so return 1 + minDepth(left) = 3

Another test case to try:

```
   1
    \
     2
      \
       3
```

Expected output: **3**

Time and Space Complexity:

- Time: O(N) (visit every node once)
- Space: O(H) (maximum height of recursion stack)

Problem 3: Diameter of Binary Tree

What’s new here?

Instead of root-to-leaf, we want the *longest* path between *any* two nodes. This path may pass through the root, or not.

LeetCode 543: Diameter of Binary Tree

Example:

```
      1
     / \
    2   3
   / \
  4   5
```

The diameter is the length of the path [4,2,1,3] or [5,2,1,3], which is 3 edges (4 nodes).

What's the trick?

At each node, the longest path *through* that node is: left subtree depth + right subtree depth.

Pseudocode:

```
function diameter(root):
    max_diameter = 0

    function depth(node):
        if node is null:
            return 0
        left = depth(node.left)
        right = depth(node.right)
        max_diameter = max(max_diameter, left + right)
        return 1 + max(left, right)

    depth(root)
    return max_diameter
```

- Use a helper function to compute the depth.
- At each node, update the global diameter if left+right is larger.

Walk-through example:

At node 2: left=1 (from 4), right=1 (from 5), so diameter could be 2.

At root: left=2 (from 2), right=1 (from 3), diameter = 3.

Try this test case:

```
    1
   /
  2
 /
3
```

Expected output: **2** (number of edges connecting 3 → 2 → 1).

Time and Space Complexity:

- Time: O(N)
- Space: O(H)

Reflect:

Notice the pattern: by processing the left and right subtrees, you can aggregate key measurements at each node.

Summary and Next Steps

Recap:

Today, you explored three classic DFS tree measurement problems:

- Maximum depth: longest root-to-leaf path.
- Minimum depth: shortest root-to-leaf path.
- Diameter: longest path between any two nodes.

Key Insights:

- DFS is perfect when you need to combine results from subtrees.
- Handling null children correctly is crucial, especially for minimum depth.
- For diameter, you must consider the sum of left and right depths.

Common Mistakes:

- Forgetting to handle nodes with only one child in minimum depth.
- Confusing node count vs. edge count (for diameter).
- Not updating global state (like max diameter) during recursion.

Action List:

- Solve all three problems on your own, even the one with code provided.
- Try to solve problem 2 and 3 using BFS (level-order traversal) as an exercise.
- Explore other tree DFS problems (e.g., path sum, subtree checks).
- Compare your solutions with others — pay attention to edge cases!
- If you get stuck, review the recursion patterns and draw trees by hand.
- Practice, practice, practice — deep understanding comes with repetition!

Happy coding, and may your trees always be well-measured!