Trees

104. Maximum Depth of Binary Tree

Here is how a Senior Staff Engineer (L6) would break down this problem. We prioritize clarity, scalability of understanding, and maintainable code over "clever" one-liners.

1. Problem Explanation

The problem asks us to find the **maximum depth** of a binary tree.

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

Think of it as calculating the "height" of a building, where every level of the tree adds `1` to the height.

Visual Examples:

Example A: A Balanced Tree

```
      3          <-- Level 1 (Root)
     / \
    9  20        <-- Level 2
      /  \
     15   7      <-- Level 3 (Leaf nodes)

Path 1: 3 -> 9      (Length: 2)
Path 2: 3 -> 20 -> 15 (Length: 3)
Path 3: 3 -> 20 -> 7  (Length: 3)

Max Depth: 3
```

Example B: A Skewed Tree (Linked List like)

```
    1            <-- Level 1
     \
      2          <-- Level 2
       \
        3        <-- Level 3
         \
          4      <-- Level 4

Path: 1 -> 2 -> 3 -> 4
Max Depth: 4
```

Example C: Empty Tree

```
(null)

Max Depth: 0
```

---

2. Solution Explanation

We will use **Depth First Search (DFS)** with **Recursion**. This is the most "elegant" solution and preferred in interviews for its readability.

The Core Logic (The Recursive Formula):

To find the depth of any node, we ask two questions:

1. What is the depth of my left child?
2. What is the depth of my right child?

The depth of the *current* node is: `1 + the larger of those two numbers.`

The Base Case (The Stopping Point): If we reach a node that doesn't exist (null), its depth is `0`.

Visual Walkthrough (Step-by-Step) Let's trace the execution flow on this tree:

```
Target Tree:
      3
     / \
    9  20
```

Step 1: Start at Root (3) The function `maxDepth(3)` is called. It needs to calculate: `1 + max( maxDepth(9), maxDepth(20) )`

```
Stack Frame 1: Node 3
Waiting for:
  Left: maxDepth(9)
  Right: maxDepth(20)
```

Step 2: Traverse Left to (9) `maxDepth(9)` is called. It checks its children. `1 + max( maxDepth(null), maxDepth(null) )`

```
Stack Frame 2: Node 9
  Left Child: null -> Returns 0
  Right Child: null -> Returns 0

  Calculation: 1 + max(0, 0) = 1
  return 1 -> to Stack Frame 1
```

Step 3: Back to Root (3) Node 3 now knows the left side depth is 1. Now it checks the right side.

```
Stack Frame 1: Node 3
  Left Result: 1
  Right: Calling maxDepth(20)...
```

Step 4: Traverse Right to (20) `maxDepth(20)` is called. `1 + max( maxDepth(null), maxDepth(null) )`

```
Stack Frame 3: Node 20
  Left Child: null -> Returns 0
  Right Child: null -> Returns 0

  Calculation: 1 + max(0, 0) = 1
  return 1 -> to Stack Frame 1
```

Step 5: Final Calculation at Root (3) Node 3 now has both results.

```
Stack Frame 1: Node 3
  Left Result: 1
  Right Result: 1

  Final Calculation: 1 + max(1, 1) = 2
  return 2
```

Answer: 2

---

3. Time and Space Complexity Analysis

We need to analyze how "expensive" this solution is.

Time Complexity: O(N) **Explanation:** We must visit every single node in the tree exactly once to ensure we haven't missed a longer path. If the tree has N nodes, we do N units of work.

Visual Derivation:

```
Tree:
    A
   / \
  B   C

Execution Flow:
1. Visit A (Cost: 1)
   -> Call B
2. Visit B (Cost: 1)
   -> B returns
   -> Call C
3. Visit C (Cost: 1)
   -> C returns
```

```
4. A returns

Total Steps = Total Nodes = N
Time Complexity = O(N)
```

Space Complexity: O(H) (or O(N) worst case) **Explanation:** This depends on the "Recursion Stack." The computer needs memory to keep track of where it is in the tree. The maximum memory used is equal to the height of the tree (H).

Scenario A: Best Case (Balanced Tree) In a perfectly balanced tree, the height (H) is roughly `log(N)`.

```
      A        <-- Stack Depth 1
     / \
    B   C      <-- Stack Depth 2
   / \
  D   E        <-- Stack Depth 3

Max Stack Height = 3
Space = O(log N)
```

Scenario B: Worst Case (Skewed Tree) This is the dangerous case. The tree is a straight line.

```
A              <-- Stack Depth 1
 \
  B            <-- Stack Depth 2
   \
    C          <-- Stack Depth 3
     \
      D        <-- Stack Depth 4

Max Stack Height = N (Total number of nodes)
Space = O(N)
```

---

4. Solution Code

Python Solution

```python
# 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
```

```
class Solution:
    def maxDepth(self, root: Optional[TreeNode]) -> int:
        """
        Calculates the maximum depth of a binary tree.

        Strategy: Recursive DFS
        1. Base Case: If node is None, depth is 0.
        2. Recursive Step: Depth = 1 + max(left_depth, right_depth)
        """

        # Base Case: We have reached past a leaf node
        if not root:
            return 0

        # Recursive Step: Get depth of left and right subtrees
        left_depth = self.maxDepth(root.left)
        right_depth = self.maxDepth(root.right)

        # The depth of the current node is itself (1) + the deeper of its two children
        return 1 + max(left_depth, right_depth)
```

JavaScript Solution

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 * this.val = (val===undefined ? 0 : val)
 * this.left = (left===undefined ? null : left)
 * this.right = (right===undefined ? null : right)
 * }
 */

/**
 * @param {TreeNode} root
 * @return {number}
 */
var maxDepth = function(root) {
    // Base Case: If the tree is empty or we reached a null child
    if (root === null) {
        return 0;
    }

    // Step 1: Calculate depth of left subtree
    const leftDepth = maxDepth(root.left);

    // Step 2: Calculate depth of right subtree
    const rightDepth = maxDepth(root.right);

    // Step 3: Return 1 (current node) + the maximum of the children
    return 1 + Math.max(leftDepth, rightDepth);
};
```

Note 1: Terms & Techniques

Recursion: This is a method where a function calls itself to solve smaller instances of the same problem.

- *Why it helps:* Trees are recursive structures (a tree is made of smaller subtrees). Recursion allows us to write code that naturally mirrors the data structure.

DFS (Depth First Search): A traversal strategy where we go as deep as possible down one path before backing up and trying a different path.

- *Why it helps:* To find the maximum depth, we *must* reach the very bottom of the tree. DFS dives straight to the bottom leaf nodes immediately, making it intuitive for "depth" calculations.

Note 2: Real-World Interview Variations

Big tech companies (Google, Meta, Bloomberg) rarely ask "Find the Max Depth" directly anymore because it is too well-known. However, they use this **exact logic** in disguised practical problems:

1. **Organization Chart Hierarchy (Google/Bloomberg):**

- *Question:* "Given a JSON object representing a CEO and their direct reports (who have their own reports), find the number of layers in the management hierarchy."
- *Mapping:* Root = CEO, Depth = Management Layers.

2. **DOM Tree Analysis (Meta/Front-end roles):**

- *Question:* "Given a raw HTML string or a DOM element, find the maximum nesting depth of the `<div>` tags."
- *Mapping:* Tree = DOM, Depth = Nesting level.

3. **File System Structure:**

- *Question:* "Given a path string, determine the deepest nested folder level."
- *Mapping:* Tree = Folders, Depth = Subfolder level.

100. Same Tree

Here is how a Senior Software Engineer (L5/L6) would approach the "Same Tree" problem.

At this level, we look beyond just "getting the right answer." We care about **readability**, **maintainability**, handling **edge cases** gracefully, and understanding **why** a specific traversal works best.

---

1. Problem Explanation

The goal is to determine if two binary trees are identical. "Identical" means two conditions must be met simultaneously:

1. **Structural Identity:** The arrangement of nodes (parents, left children, right children) is exactly the same.
2. **Value Identity:** Every corresponding node has the exact same value.

If tree `p` has a node with value `5` at the left child, tree `q` must also have a node with value `5` at the left child. If `p` has `null` somewhere, `q` must also have `null` in that exact spot.

Visualizing the "Same" concept:

Scenario A: Identical Trees (Return True) Both structure and values match perfectly.

```
    Tree P            Tree Q

      (1)               (1)
     /   \             /   \
   (2)   (3)         (2)   (3)
```

Scenario B: Structural Mismatch (Return False) Values are the same, but the shape is different.

```
    Tree P            Tree Q

      (1)               (1)
     /                     \
   (2)                     (2)
```

```
(P has 2 on the left)  (Q has 2 on the right)
```

Scenario C: Value Mismatch (Return False) Structure is the same, but values differ.

```
    Tree P            Tree Q
    (1)              (1)
   /   \            /   \
 (2)   (3)        (2)   (4)  <-- Mismatch!
```

---

2. Solution Explanation

An L5/L6 engineer chooses **Depth First Search (DFS)** recursion here because it is intuitive, cleaner to write than an iterative approach, and mirrors the inductive definition of a tree.

The Algorithm: Synchronized Traversal

We traverse both trees at the exact same time. At any given node pair `(nodeP, nodeQ)`, we perform three checks:

1. **The "Both Empty" Check:** If both `nodeP` and `nodeQ` are `null` (None), we have reached the end of a branch successfully in both trees. This is a match.
2. **The "One Empty" Check:** If one is `null` but the other is not, the structure is different. This is a mismatch.
3. **The "Value" Check:** If the values inside `nodeP` and `nodeQ` are different, the content is different. This is a mismatch.

If the current nodes match, we recursively check:

- `nodeP.left` vs `nodeQ.left`
- `nodeP.right` vs `nodeQ.right`

Visualizing the Execution Flow

Let's trace the algorithm on these two trees:

```
   Tree P            Tree Q
     (1)               (1)
    /   \             /   \
  (2)   (3)         (2)   (3)
```

Step 1: Compare Roots Current: P=(1), Q=(1)

```
[P-Node: 1]  vs  [Q-Node: 1]
    |                |
1. Both null? No.
2. One null? No.
3. Values diff? No (1 == 1).

Result: MATCH. Proceed to check subtrees.
Action: Splits into two parallel recursive calls.
        -> Call A: Check P-Left (2) vs Q-Left (2)
        -> Call B: Check P-Right (3) vs Q-Right (3)
```

Step 2: Recursive Call A (The Left Branch) Current: P=(2), Q=(2)

```
      (1)              (1)
     /                /
-> (2)              (2)

   [P-Node: 2]  vs  [Q-Node: 2]
       |                |
   Result: MATCH.
   Action: Split again.
           -> Call A1: Check Left children of 2 (Both Null)
           -> Call A2: Check Right children of 2 (Both Null)
```

Step 3: Hitting the leaves (Base Case) Inside Call A1 (Left children of the nodes 2):

```
      (2)              (2)
     /                /
   null             null

   [P-Node: null] vs [Q-Node: null]
         |                 |
   1. Both null? YES.

   Result: Return TRUE. (Stop going deeper)
```

Because the Left and Right children of node (2) return TRUE, node (2) returns TRUE to the root. The same happens for the right branch (nodes 3). Since both branches are TRUE, the root returns TRUE.

---

3. Time and Space Complexity Analysis

Time Complexity: O(N)

We visit every node exactly once. If the trees are different, we stop early. If they are identical, we visit all N nodes.

```
Iteration Visual:

Node 1 visited:  [x] Cost: 1
Node 2 visited:  [x] Cost: 1
Node 3 visited:  [x] Cost: 1
...
Total Cost = Sum of all nodes = N
```

Notation: O(N), where N is the number of nodes in the smaller tree.

Space Complexity: O(H)

This is determined by the "Recursion Stack." In the worst case, the stack grows as deep as the tree height (H).

Best Case (Balanced Tree): Height is roughly log(N).

```
Stack Frame Depth:
| [Check (1) vs (1)] |
| [Check (2) vs (2)] |
| [Check (4) vs (4)] | <--- Max depth is small
----------------------
Space: O(log N)
```

Worst Case (Skewed Tree/Linked List): Height is N.

```
      (1)
        \
        (2)
          \
          (3)
            \
            (4)

Stack Frame Depth:
| [Check (1) vs (1)] |
| [Check (2) vs (2)] |
| [Check (3) vs (3)] |
| [Check (4) vs (4)] | <--- Stack grows linearly with N
----------------------
Space: O(N)
```

---

4. Solution Code

An L6 engineer ensures code is idiomatic. In Python, we use explicit `is None` checks. In JS, we handle strict equality.

Python Solution

```python
# 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

class Solution:
    def isSameTree(self, p: Optional[TreeNode], q: Optional[TreeNode]) -> bool:
        """
        Determines if two binary trees are structurally and value identical.
        Uses Depth First Search (DFS).
        """

        # 1. Base Case: Both nodes are None (End of branch)
        # If both are None, they are identical up to this point.
        if not p and not q:
            return True

        # 2. Structural Mismatch Check
        # If one is None but the other isn't, trees aren't same.
        if not p or not q:
            return False

        # 3. Value Mismatch Check
        # If values differ, trees aren't same.
        if p.val != q.val:
            return False

        # 4. Recursive Step
        # The trees are same ONLY if:
        # - The left subtrees match AND
        # - The right subtrees match
        return self.isSameTree(p.left, q.left) and self.isSameTree(p.right, q.right)
```

Javascript Solution

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 * this.val = (val===undefined ? 0 : val)
 * this.left = (left===undefined ? null : left)
 * this.right = (right===undefined ? null : right)
 * }
 */

/**
 * @param {TreeNode} p
 * @param {TreeNode} q
 * @return {boolean}
 */
var isSameTree = function(p, q) {
    // 1. Base Case: Both are null
    if (!p && !q) {
        return true;
    }
    // 2. Structural Check: One is null, the other is not
    if (!p || !q) {
        return false;
    }

    // 3. Value Check: Data mismatch
    if (p.val !== q.val) {
        return false;
    }

    // 4. Recursive Step: Check Left AND Right
    return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
};
```

Note 1: Terminology Breakdown

DFS (Depth First Search): This is the strategy used above. Think of it like a maze explorer who goes as far as possible down one path before backtracking. In this problem, we go all the way down to the leaf nodes (bottom) of the left side before we even look at the right side. This is helpful here because if we find a mismatch deep in the left branch, we can return `False` immediately without wasting time checking the right branch.

Note 2: Real World Interview Variations

Top-tier companies (Google, Meta, Bloomberg) rarely ask "Same Tree" directly anymore because it is too simple. However, they ask the **exact same logic** disguised as practical system design or data manipulation problems:

1. **DOM Tree Comparison (Meta/Google Frontend):**

- *Question:* "Given two HTML DOM trees, write a function to detect if they render the exact same structure."
- *Mapping:* The DOM is just a tree. You use the exact code above, but instead of checking `.val`, you check `tagName` (div, span) and `classList`.

2. **JSON Diffing Tool (Bloomberg/FinTech):**

- *Question:* "Write a function that compares two complex nested JSON objects and returns True if they are identical."
- *Mapping:* A JSON object is a tree. Keys are nodes. Values are children. You must recursively traverse both objects to ensure keys and values match.

3. **File System Replica Check (Google/Dropbox):**

- *Question:* “Verify that a backup folder has the exact same directory structure and files as the source folder.”
- *Mapping:* Folders are tree nodes. Files are leaves. You traverse the directory structure (Tree P) and the backup structure (Tree Q) simultaneously to verify integrity.

226. Invert Binary Tree

Here is how a Senior Staff Engineer (L5/L6) would break down **LeetCode 226: Invert Binary Tree**, focusing on clarity, visualization, and production-quality reasoning.

---

1. Problem Explanation

The Core Concept: To “invert” a binary tree means to create its mirror image. Imagine standing a mirror right next to the tree; the reflection is the inverted tree.

In computer science terms, for **every single node** in the tree, you must swap its left child with its right child.

Visualizing the Goal:

Input Tree:

```
      Root
        |
        4
      /   \
     2     7
    / \   / \
   1   3 6   9
```

The Transformation (Mirroring): We want to flip it horizontally.

- The `2` (left child) should move to the right.
- The `7` (right child) should move to the left.
- But it doesn’t stop there! The children of `2` (`1` and `3`) must also swap places.
- The children of `7` (`6` and `9`) must also swap places.

Output Tree (Inverted):

```
      Root
        |
        4
      /   \
     7     2    <-- Note: 7 and 2 swapped
```

```
  / \   / \
 9   6 3   1  <-- Note: Children also swapped (9&6, 3&1)
```

---

2. Solution Explanation

An L5/L6 engineer looks for the **sub-problem**. To invert the whole tree (Root 4), we don't need to panic about the bottom leaves immediately. We just need to follow a simple rule:

The Algorithm (The "Recursive Leap of Faith"):

1. Look at the current node.
2. **Swap** its left and right children.
3. Tell the left child (which was previously the right child) to go invert itself.
4. Tell the right child (which was previously the left child) to go invert itself.

Let's trace this with **ASCII Diagrams**.

Initial State:

```
    4
  /   \
 2     7
```

Step 1: Process Node 4 (The Root) We temporarily store the Left child (`2`) and Right child (`7`). We overwrite `4.left` with `7`. We overwrite `4.right` with `2`.

Result after Step 1:

```
    4
  /   \
 7     2   <-- Children swapped!
```

Wait, are we done? No. The sub-trees attached to `7` and `2` are still in their original orientation. We must go deeper.

Step 2: Recurse on the new Left Child (Node 7) Now we focus *only* on the subtree rooted at `7`.

```
    (Focusing on 7)
       7
     /   \
    6     9
```

Perform the swap on `7`'s children. `6` goes right, `9` goes left.

```
       7
     /   \
    9     6
```

Step 3: Recurse on the new Right Child (Node 2) Now we focus *only* on the subtree rooted at 2.

```
    (Focusing on 2)
        2
      /   \
     1     3
```

Perform the swap on 2's children. 1 goes right, 3 goes left.

```
        2
      /   \
     3     1
```

Final Assembly: When the recursion finishes, the changes "bubble up" to the main view.

```
        4
      /   \
     7     2
    / \   / \
   9   6 3   1
```

The tree is fully inverted.

---

3. Time and Space Complexity Analysis

An L6 engineer cares deeply about "at scale" performance.

Time Complexity: O(N) **Why?** Because we visit every node exactly once.

Visualization of Work Done: Imagine N is the number of dots (nodes). We perform a constant amount of work (a swap) at each dot.

```
Nodes:  (o)    (o)    (o)    (o)    (o)
         |      |      |      |      |
Work:   [Swap] [Swap] [Swap] [Swap] [Swap]

Total Time = Constant * Number of Nodes
           = c * N
           = O(N)
```

Space Complexity: O(h) or O(N) **Why?** This depends on the "shape" of the tree. The space is consumed by the **Call Stack** (the computer's memory keeping track of "where am I in the recursion?").

Scenario A: The Balanced Tree (Best Case) If the tree is perfectly triangular (balanced), the height h is roughly log N.

```
      (Root)       <-- Stack Frame 1
      /    \
    (A)    (B)     <-- Stack Frame 2
    / \    / \
  (C) (D) (E) (F)  <-- Stack Frame 3
```

In a balanced tree, we only go as deep as the height. **Space = O(log N)**

Scenario B: The Skewed Tree (Worst Case) This is what interviewers look for. What if the tree is just a long line (essentially a linked list)?

```
(Root)             <-- Stack Frame 1 (Waiting for child...)
  \
  (A)              <-- Stack Frame 2 (Waiting for child...)
    \
    (B)            <-- Stack Frame 3 (Waiting for child...)
      \
      (C)          <-- Stack Frame 4 (Active)
```

Here, the recursion goes all the way down before returning. The stack grows to the size of the tree. **Space = O(N)**

---

4. Solution Code

Here is the clean, production-ready implementation.

Python Solution We use a standard definition for a tree node.

```python
# 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

class Solution:
    # Function to invert the binary tree
    # Input: root (TreeNode)
    # Output: TreeNode (The root of the inverted tree)
    def invertTree(self, root):
        # Base Case:
        # If the current node is None (we reached past a leaf),
        # there is nothing to invert. Just return None.
        if root is None:
            return None

        # The Swap Logic:
```

```
        # We store the current children in temporary variables
        # (or use Python's tuple unpacking) to swap them.

        # 1. Swap the children
        root.left, root.right = root.right, root.left

        # 2. Recurse Left (which was the old right)
        self.invertTree(root.left)

        # 3. Recurse Right (which was the old left)
        self.invertTree(root.right)

        # Return the current node so the parent can link to it
        return root
```

JavaScript Solution

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 * this.val = (val===undefined ? 0 : val)
 * this.left = (left===undefined ? null : left)
 * this.right = (right===undefined ? null : right)
 * }
 */

/**
 * @param {TreeNode} root
 * @return {TreeNode}
 */
var invertTree = function(root) {
    // Base Case: Safety check.
    // If tree is empty or we hit the bottom, stop.
    if (root === null) {
        return null;
    }

    // Step 1: Swap the pointers
    // We hold a reference to the left child so we don't lose it
    // when we overwrite root.left.
    const temp = root.left;
    root.left = root.right;
    root.right = temp;

    // Step 2: Recursive calls
    // It doesn't matter if root.left is now null or a valid node,
    // the base case at the start of the function will handle it.
    invertTree(root.left);
    invertTree(root.right);

    // Step 3: Return the root to the caller
    return root;
};
```

Terminology & Concepts

Recursion (DFS - Depth First Search): This is the technique where a function calls itself. In this problem, we use recursion to dive deep into the tree. We process the current node, then dive into the left child, then the right child. It simplifies the code because we solve the problem for *one* node and assume the recursive calls will handle the rest.

Real-World Interview Variations (The "Indirect" Ask)

Top companies (Google, Meta, Bloomberg) rarely ask "Invert a binary tree" directly anymore because it is too famous. Instead, they wrap it in practical scenarios:

1. **The "UI Mirror" (Front-end/Mobile Focus):**

- *Question:* "You are building an app that needs to support Right-to-Left (RTL) languages like Arabic or Hebrew. You have a tree of UI components. How do you programmatically flip the entire layout?"
- *Connection:* The UI is a tree (DOM tree). Flipping the layout is literally inverting the binary tree of components.

2. **The "Dependency Swap" (Systems/Build Tools):**

- *Question:* "We have a hierarchical configuration format (JSON-like). We need to write a migration script that swaps the priority of 'Primary' and 'Secondary' servers defined at every level of the nested config."
- *Connection:* The nested config is a tree. Swapping 'Primary' and 'Secondary' keys recursively is the Invert Tree problem.

3. **The "Syntax Tree Transformation" (Compilers/Tools):**

- *Question:* "You have an expression tree for a mathematical formula (e.g., `A - B`). We want to negate the logic such that every subtraction becomes a reverse subtraction (`B - A`)."
- *Connection:* This requires traversing the Abstract Syntax Tree (AST) and swapping the operands (children) of every operator node.

124. Binary Tree Maximum Path Sum

Here is a breakdown of "Binary Tree Maximum Path Sum" from the perspective of a Senior Staff Software Engineer (L6). We focus on not just getting the right answer, but understanding the *contract* of our functions and handling edge cases cleanly.

1. Problem Explanation

The core difficulty in this problem is changing our definition of a "path." In many basic tree problems, a path goes from the Root down to a Leaf.

However, for this problem:

1. A path can start and end at **any** node.
2. It must follow connections (parent-child).
3. It cannot branch (you can't go down both left and right children and call that one single line).
4. It must contain at least one node.

The Goal: Find the sequence of nodes where the sum of their values is the highest possible.

Simple Example:

```
  1
 / \
2   3
```

- Possible paths: `2`, `3`, `1`, `2->1`, `1->3`, `2->1->3`.
- Sums: `2`, `3`, `1`, `3`, `4`, `6`.
- **Result:** 6 (The path `2 -> 1 -> 3`).

Tricky Example (Negatives):

```
 -10
 /  \
9    20
    /  \
   15   7
```

- If we include the root `-10`, we lower our total.
- However, the subtree on the right has a very high value.
- The best path is just inside the right subtree: `15 -> 20 -> 7`.
- Sum: `15 + 20 + 7 = 42`.

---

2. Solution Explanation

As an L6 engineer, I look for a **single pass** solution. We cannot afford to calculate every path separately. We need to traverse the tree once (Depth First Search) and gather all necessary info.

The "Split" vs. "Straight" Decision

At any specific Node (let's call it `N`), we have two distinct responsibilities:

1. **The Internal Calculation (The "Split"):** Could the highest sum path in the entire universe be the curve that goes `Left Child -> N -> Right Child`? We calculate this, compare it to our global maximum record, and update if it's better.
2. **The External Contribution (The "Straight"):** `N`'s parent is asking, "What is the best path ending at you that I can extend?" We cannot return the "Split" path because a path cannot branch three ways. We must return `N.val + max(Left, Right)`.

Visualizing the Logic

Imagine we are processing Node `20` from the example above.

```
       (Parent -10 waiting...)
             |
            20
          /    \
 (Result 15)  (Result 7)
       |          |
      15          7
```

Step A: Receive Reports Node 15 reports: "My best straight path is 15." Node 7 reports: "My best straight path is 7."

Step B: The Internal "Split" Check (Update Global Max) We check if creating a bridge through `20` is the best path ever found. Formula: `Left_Report + Node.val + Right_Report` Calculation: `15 + 20 + 7 = 42`. *Action: Update Global Max to 42.*

Step C: The External "Straight" Return We report back to Parent `-10`. We can only pick ONE leg (Left or Right) to extend upward. Formula: `Node.val + max(Left_Report, Right_Report)` Calculation: `20 + max(15, 7) = 20 + 15 = 35`. *Action: Return 35 to parent.*

Crucial Optimization: The "Zero Floor" What if a child returns a negative value (e.g., `-50`)? Including that child would only hurt our sum. **Rule:** If a child's contribution is negative, we pretend it is `0` (we cut that branch off).

---

3. Time and Space Complexity Analysis

Time Complexity: O(N) We touch every node exactly once using a Post-Order Traversal (process children, then process self).

Space Complexity: O(H) H is the height of the tree. This is the cost of the recursion stack.

- Best Case (Balanced): `O(log N)`
- Worst Case (Skewed Line): `O(N)`

Visual Derivation of Complexity

```
SCENARIO: A Skewed Tree (Worst Case Space)

    1  <-- Stack Frame 1 Open
     \
      2  <-- Stack Frame 2 Open
       \
        3  <-- Stack Frame 3 Open
         \
          4  <-- Stack Frame 4 Open (Hit leaf)

[Space Analysis]
Since 4 waits for children, 3 waits for 4, 2 waits for 3...
We hold 4 stack frames in memory simultaneously.
Space = Height of Tree.

[Time Analysis]
    1
   / \
  2   3
We visit 1.
  -> Call 2 (Visit 2, Return)
  -> Call 3 (Visit 3, Return)
Merge logic at 1.
Total operations is proportional to number of nodes (N).
Time = Linear O(N).
```

---

4. Solution Code

Here is the clean, production-quality code.

Python Solution

```python
# 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

class Solution:
    def maxPathSum(self, root: Optional[TreeNode]) -> int:
        # We need a placeholder for the global maximum.
        # We initialize it to negative infinity so any real node value will beat it.
        self.max_path = float('-inf')

        def get_max_gain(node):
            if not node:
                return 0

            # RECURSIVE STEP:
            # Calculate max sum coming from left and right subtrees.
            # CRITICAL: If a subtree gives negative gain, we treat it as 0 (don't take that
            left_gain = max(get_max_gain(node.left), 0)
            right_gain = max(get_max_gain(node.right), 0)

            # PHASE 1: The "Split" Path
            # Calculate path sum passing THROUGH this node (Left + Node + Right).
            # This path assumes this node is the highest point (root of the path).
            current_path_sum = node.val + left_gain + right_gain

            # Update global maximum if this split path is the best we've seen.
            self.max_path = max(self.max_path, current_path_sum)

            # PHASE 2: The "Straight" Path
            # Return the max gain this node contributes to its PARENT.
            # We can only bring ONE child's path up with us.
            return node.val + max(left_gain, right_gain)

        get_max_gain(root)
        return self.max_path
```

Javascript Solution

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 * this.val = (val===undefined ? 0 : val)
 * this.left = (left===undefined ? null : left)
 * this.right = (right===undefined ? null : right)
 * }
 */
/**
```

```
 * @param {TreeNode} root
 * @return {number}
 */
var maxPathSum = function(root) {
    // Initialize with smallest possible integer to handle trees with all negative numbers
    let maxPath = Number.MIN_SAFE_INTEGER;

    // Helper function for DFS
    // Returns: The maximum contribution this branch can provide to its parent
    function getMaxGain(node) {
        if (node === null) {
            return 0;
        }

        // Post-order traversal: Solve children first
        // If a child returns a negative path, simple math: ignore it (use 0).
        // This effectively "cuts" the path before reaching the negative section.
        const leftGain = Math.max(getMaxGain(node.left), 0);
        const rightGain = Math.max(getMaxGain(node.right), 0);

        // Check "Bridge" case: Left -> Node -> Right
        // This is a candidate for the final answer, but cannot be passed up.
        const pathThroughNode = node.val + leftGain + rightGain;

        // Update global record
        maxPath = Math.max(maxPath, pathThroughNode);

        // Return "Extension" case: Node + (Best of Left or Right)
        // This value allows the parent to possibly extend the path further up.
        return node.val + Math.max(leftGain, rightGain);
    }

    getMaxGain(root);
    return maxPath;
};
```

New Terms / Techniques

"Post-Order Traversal with State Accumulation" This is the specific flavor of Depth First Search used here. Instead of just visiting nodes, we are bubbling up "state" (the partial sum) from the bottom (leaves) to the top. This technique is ubiquitous in L5+ interviews for Tree problems (e.g., finding tree diameter, checking balanced status).

"Kadane's Algorithm on Trees" You might recognize the logic `max(current, 0)`. This is very similar to **Kadane's Algorithm** for finding the maximum subarray sum in a 1D array. Here, we apply that same "reset if negative" logic to tree branches.

Real World / Interview Variations

Google, Meta, and Bloomberg rarely ask this exact vanilla question to senior candidates anymore because it is well-known. They ask these variations:

1. **The "Expensive Network" (Google/Bloomberg)**

- **Context:** Nodes are servers, edges have latency (negative cost) or bandwidth (positive capacity).
- **Question:** Find the server chain with the highest throughput-to-latency ratio.
- **Twist:** Edges have weights, not just nodes.

2. **Organization Chart "Influence" (Meta)**

- **Context:** You have an org chart (Tree). Each employee has an "influence score" (positive or negative).
- **Question:** Select a team that is connected (a valid subtree or path) that maximizes total influence, but you cannot select an employee without selecting their direct manager unless you break the chain.

3. **Longest Path with Same Value (Google)**

- **Question:** Instead of Sum, find the length of the longest path where every node has the *same* value.
- **Relation:** Uses the exact same "Split vs Straight" logic, but updates lengths instead of sums.

102. Binary Tree Level Order Traversal

Here is a comprehensive breakdown of **Leetcode 102: Binary Tree Level Order Traversal**, approached with the clarity, depth, and structural rigor expected of a Senior (L5) or Staff (L6) Engineer at Google.

---

1. Problem Explanation

The core of this problem is to traverse a **Binary Tree** layer by layer, starting from the top (root) and moving downwards. Within each layer (level), we must visit nodes from **left to right**.

The output must be a list of lists, where each inner list represents all the values found at a specific depth level.

Visualizing the Goal:

Imagine the tree as a corporate hierarchy or a pyramid. We want to take a "census" of everyone at Rank 1, then everyone at Rank 2, and so on.

Input Tree:

```
        3          <-- Level 0 (Root)
       / \
      9  20        <-- Level 1
        /  \
       15   7      <-- Level 2
```

Expected Output:

```
[
  [3],       // All nodes at Level 0
  [9, 20],   // All nodes at Level 1
  [15, 7]    // All nodes at Level 2
]
```

Key Constraints & Edge Cases:

- **Empty Tree:** If the root is `null` (or `None`), the result should be an empty list `[]`.
- **Single Node:** If the tree only has a root, the result is `[[3]]`.
- **Unbalanced:** One side might be much deeper than the other; the algorithm must handle this naturally.

---

2. Solution Explanation

To solve this, we use an algorithm called **Breadth-First Search (BFS)**.

The "Why" behind the logic: A standard recursion (Depth-First Search) dives deep into one branch before coming back up. That doesn't work well here because we need to group nodes by their horizontal level, not their vertical lineage.

Instead, we use a **Queue** data structure. A Queue works on the principle of **FIFO (First In, First Out)**—like a line of people waiting for coffee.

- We join the line (enqueue) at the back.
- We get served (dequeue) from the front.

The Algorithm Strategy:

1. Initialize a **Queue** and put the `root` node in it.
2. While the Queue is not empty:

- Measure the current length of the Queue. Let's call this `levelSize`. **This is the crucial L5/L6 insight**: The number of elements currently in the queue represents exactly *all* the nodes at the current level.
- Create a temporary list `currentLevelNodes`.
- Loop `levelSize` times:
- Remove (dequeue) a node from the front.
- Add its value to `currentLevelNodes`.
- Check if it has a Left Child. If yes, add it to the back of the Queue.
- Check if it has a Right Child. If yes, add it to the back of the Queue.
- Add `currentLevelNodes` to our final result.

Step-by-Step ASCII Walkthrough:

Initial State:

- `Result = []`
- `Queue = [3]`

Iteration 1 (Level 0):

1. Measure `levelSize`: Queue has 1 item (`3`). So we loop 1 time.
2. **Process Node 3**:

- Pop `3`.
- Add `3` to `currentLevelNodes`.
- Has Left Child (`9`)? Yes -> Push `9` to Queue.
- Has Right Child (`20`)? Yes -> Push `20` to Queue.

```
Visual Check:
       3  <-- Just processed
      / \
     9  20 <-- Now waiting in Queue

Queue State:  [9, 20]
Result State: [[3]]
```

Iteration 2 (Level 1):

1. Measure `levelSize`: Queue has 2 items (`9` and `20`). So we loop 2 times.

- *Note: Even if we add children during this loop, we only process the original 2 items for this level.*

2. **Process Node 9**:

- Pop `9`.
- Add to list.
- Children? None.

3. **Process Node 20**:

- Pop 20.
- Add to list.
- Has Left Child (15)? Yes -> Push 15.
- Has Right Child (7)? Yes -> Push 7.

```
Visual Check:
       3
      / \
     9  20 <-- Just processed
         / \
       15  7 <-- Now waiting in Queue

Queue State:  [15, 7]
Result State: [[3], [9, 20]]
```

Iteration 3 (Level 2):

1. Measure `levelSize`: Queue has 2 items (15, 7). Loop 2 times.
2. **Process Node 15**:

- Pop 15.
- Add to list.
- Children? None.

3. **Process Node 7**:

- Pop 7.
- Add to list.
- Children? None.

```
Visual Check:
All nodes processed.

Queue State:  [] (Empty)
Result State: [[3], [9, 20], [15, 7]]
```

Termination: Queue is empty. Return Result.

---

3. Time and Space Complexity Analysis

An L6 engineer communicates complexity not just with a label ("It's O(N)"), but by proving it structurally.

Time Complexity: O(N)

- **N** is the total number of nodes in the tree.
- We visit every single node exactly once to add it to the queue.
- We visit every single node exactly once to pop it from the queue.

- Operations inside the loop (checking children, adding to list) are constant time `O(1)`.

```
Visual Derivation (Time):

Node Count (N) = 5

Processing:
[Node 3]  -> 1 visit
[Node 9]  -> 1 visit
[Node 20] -> 1 visit
[Node 15] -> 1 visit
[Node 7]  -> 1 visit

Total Steps ~ 1 * N
Therefore: Time = O(N)
```

Space Complexity: O(N)

- We need space for the `Result` list, which holds N integers.
- We need space for the `Queue`.
- **Worst Case for Queue:** The queue is biggest when storing the "widest" level of the tree. In a perfect binary tree, the bottom-most level contains roughly `N / 2` nodes.
- Since `N / 2` scales linearly with `N`, the space is `O(N)`.

```
Visual Derivation (Space - Worst Case):

       O        Level 0 (1 node)
     /   \
    O     O     Level 1 (2 nodes)
   / \   / \
  O   O O   O   Level 2 (4 nodes) <- Queue holds these simultaneously!

If Total Nodes (N) = 7, the last level holds 4.
4 is roughly N/2.
In Big-O notation, we drop constants like 1/2.
Therefore: Space = O(N)
```

---

4. Solution Code

Python Solution *Using `collections.deque` for an efficient O(1) pop from the left. A standard list `pop(0)` is O(N) and would be considered inefficient in an interview.*

```python
from collections import deque
```

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

def levelOrder(root):
    # L5 Check: Handle the edge case immediately
    if not root:
        return []

    result = []
    # Initialize Queue with the root
    queue = deque([root])

    # Continue until there are no nodes left to process
    while queue:
        level_size = len(queue)  # Snapshot current queue length
        current_level_nodes = []

        # Process strictly the number of nodes in this level
        for _ in range(level_size):
            node = queue.popleft() # Efficient O(1) removal
            current_level_nodes.append(node.val)

            # Add children to the back of the line for the NEXT level
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        # Level complete, add to result
        result.append(current_level_nodes)

    return result
```

JavaScript Solution *Note on JS Arrays: Using `shift()` on a JS array is technically O(N), making the total time O(N^2) in the worst case (a skewed tree). In a rigorous Google L6 interview, you should mention this. To be perfectly optimal, you would use a Queue implementation or pointers. However, for most interviews, `shift()` is accepted if you acknowledge the trade-off. Below is the clean `shift()` version.*

```javascript
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 * this.val = (val===undefined ? 0 : val)
 * this.left = (left===undefined ? null : left)
 * this.right = (right===undefined ? null : right)
 * }
 */

/**
 * @param {TreeNode} root
 * @return {number[][]}
 */
var levelOrder = function(root) {
    // L5 Check: Handle edge case
    if (!root) return [];

    const result = [];
    const queue = [root];

    while (queue.length > 0) {
        // Capture the size of the *current* level
        const levelSize = queue.length;
        const currentLevelNodes = [];

        // Loop through only the nodes present at the start of this level
        for (let i = 0; i < levelSize; i++) {
            // shift() removes the first element
            const node = queue.shift();
            currentLevelNodes.push(node.val);

            // Enqueue children for the next level
            if (node.left) queue.push(node.left);
            if (node.right) queue.push(node.right);
        }

        result.push(currentLevelNodes);
    }

    return result;
};
```

Note 1: New Terms / Techniques

BFS (Breadth-First Search): This is the name of the algorithm used. It describes a “radiating” movement. Imagine dropping a stone in a pond; the ripples move outward in perfect circles. BFS visits nodes in those expanding circles (levels). It applies here because the problem explicitly asks for "Level Order," which is synonymous with Breadth-First.

Queue (FIFO): The data structure used to manage the nodes. "First In, First Out." It is essential for BFS because it ensures we process older discoveries (nodes higher up) before newer discoveries (nodes lower down).

Note 2: Real World & Interview Variations

Top-tier companies (Google, Meta, Bloomberg) rarely ask "Implement Level Order Traversal" directly anymore. They mask it in practical scenarios:

1. **The "Company Org Chart" (Bloomberg/Meta):**

- *Question:* "Given a CEO and their direct reports, print the company structure level by level so we can see who is at the VP level, Director level, etc."
- *Mapping:* The CEO is the Root. Direct reports are children.

2. **DOM Tree Serialization (Google - Frontend):**

- *Question:* "How would you save the structure of an HTML page (DOM) to a JSON file so that when we load it back, the hierarchy is preserved layer by layer?"
- *Mapping:* HTML tags are nodes. Nested tags are children.

3. **Network Broadcasting / Infection Spread (Google/Uber):**

- *Question:* "A computer gets a virus. It spreads to connected computers in 1 second. How many computers are infected after K seconds?"
- *Mapping:* This is a Level Order Traversal where "K seconds" equals "K Levels" deep.

4. **Right Side View (Meta):**

- *Question:* "If you stand on the right side of the tree, what numbers do you see?"
- *Mapping:* Do a Level Order Traversal, but only save the **last** element of every level list.

297. Serialize and Deserialize Binary Tree

Here is how a Senior Staff Engineer (L6) at Google would break down, solve, and explain **LeetCode 297: Serialize and Deserialize Binary Tree**.

At this level, we care about more than just getting the code to run. We care about **maintainability**, **readability**, and handling **edge cases** (like empty trees or negative values) gracefully. We also choose the right tool for the job—in this case, choosing between Depth First Search (DFS) and Breadth First Search (BFS) based on the tree's structure and implementation simplicity.

---

1. Problem Explanation

The Goal: Imagine you have a Binary Tree object in memory. It consists of nodes connected by pointers (references). You need to save this tree to a text file or send it over a network to another computer.

You cannot send "memory pointers" over the network because the other computer has its own separate memory.

1. **Serialization:** Convert the Tree structure into a single string (sequence of characters).
2. **Deserialization:** Take that string and reconstruct the exact same Tree structure in memory, with the same values and connections.

The Challenge: A simple list of values isn't enough. If I give you the values `[1, 2, 3]`, how do you know if `2` is the left child of `1`, or if `2` is the right child? Or if `3` is a child of `2`?

We must encode the **structure** (the nulls/emptiness) alongside the values.

---

2. Solution Explanation: Preorder Traversal (DFS)

We will use **Depth First Search (DFS)**, specifically **Preorder Traversal** (Root Left Right).

Why DFS? It is often easier to implement recursively. When we read the data back to rebuild the tree, we process the root, then immediately try to build the left subtree, then the right. This maps naturally to the order in which we read the string.

Part A: Serialization (Tree String) Strategy:

1. Visit the Root. Add its value to our list.
2. Recursively serialize the Left Child.
3. Recursively serialize the Right Child.
4. **Crucial Step:** If we encounter a `null` (empty) node, we must record it (e.g., using a symbol like `#` or `N`). This tells us where a branch ends.

Visual Walkthrough:

Consider this Binary Tree:

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

Step-by-Step Traversal:

1. **Start at Root (1):**

- Write `1`.
- Move Left.
- *Current String:* `1`

2. **At Node (2):**

- Write `2`.
- Move Left.
- *Current String:* `1,2`

3. **At Node (2)'s Left Child:**

- It is `null`. Write `#`.
- Return to (2).
- *Current String:* `1,2,#`

4. **At Node (2)'s Right Child:**

- It is `null`. Write `#`.
- Return to (2), then return to (1).
- *Current String:* `1,2,#,#`

(Visual Check: We have finished the entire left side of Root 1) 5. **Move to Root (1)'s Right Child -> Node (3):** * Write `3`. * Move Left. * *Current String:* `1,2,#,#,3`

6. **At Node (3)'s Left Child -> Node (4):**

- Write `4`.
- Move Left (`null`) -> Write `#`.
- Move Right (`null`) -> Write `#`.
- *Current String:* `1,2,#,#,3,4,#,#`

7. **At Node (3)'s Right Child -> Node (5):**

- Write `5`.
- Move Left (`null`) -> Write `#`.
- Move Right (`null`) -> Write `#`.
- *Current String:* `1,2,#,#,3,4,#,#,5,#,#`

Final Serialized String: `"1,2,#,#,3,4,#,#,5,#,#"`

Part B: Deserialization (String Tree) **Strategy:** We treat the string as a queue of values. We pop the first value and create a node. Because we used **Preorder**, we know the next values in the queue belong to the **Left** subtree until we hit the null markers, then we move to the **Right**.

Input Queue: `[1, 2, #, #, 3, 4, #, #, 5, #, #]`

Visual Walkthrough:

Step 1: Pop `1`.

- This is not `#`. Create a new node `1`.
- We need to find `1.left`. **Recurse.**

```
   1
  / ?
 ?
```

Step 2: Pop `2`.

- This is not `#`. Create node `2`. Set it as `1.left`.
- We need to find `2.left`. **Recurse.**

```
   1
  /
 2
/ ?
?
```

Step 3: Pop `#`.

- It is null. Return `null`.
- `2.left` becomes `null`.
- Now we need `2.right`. **Recurse.**

Step 4: Pop `#`.

- It is null. Return `null`.
- `2.right` becomes `null`.
- *Node 2 is complete.* Return `2` back to the caller (Node 1).

Step 5: Back at Node `1`.

- We finished `1.left`. Now we need `1.right`. **Recurse.**
- (Remaining Queue: `[3, 4, #, #, 5, #, #]`)

Step 6: Pop `3`.

- Create node `3`. Set as `1.right`.
- Recurse for `3.left`.

```
    1
   / \
  2   3
```

```
     / ?
    ?
```

Step 7: Pop 4.

- Create node 4. Set as `3.left`.
- Next two pops are `#` and `#`.
- `4.left` = null, `4.right` = null.
- Return 4 to Node 3.

Step 8: Back at Node 3.

- Recurse for `3.right`.
- Pop 5. Create node 5.
- Next two pops are `#` and `#`.
- `5.left` = null, `5.right` = null.
- Return 5 to Node 3.

Final Result: The tree is fully reconstructed.

---

3. Time and Space Complexity Analysis

Time Complexity: O(N) We visit every node exactly once.

Visual Derivation: Imagine the timeline of the algorithm as a straight line. Every node (N) adds a constant amount of work (writing the value or creating the node).

```
Timeline: [Node 1] -> [Node 2] -> [Node 3] ... -> [Node N]
Work:     [ Constant ] + [ Constant ] + [ Constant ] ...
Total:    N * Constant = O(N)
```

Space Complexity: O(N) We need space for the recursion stack (in DFS) and space to store the final string.

Visual Derivation (Recursion Stack): In the worst case (a skewed tree, looking like a linked list), the recursion goes deep before it returns.

```
Tree:
  1
   \
    2
     \
      3
       \
        4

Stack Growth:
| Frame 4 (Node 4) |  <-- Max depth = N
| Frame 3 (Node 3) |
| Frame 2 (Node 2) |
| Frame 1 (Node 1) |
--------------------
Space Used = O(N)
```

In a balanced tree, the height is log(N), so stack space is O(log N). However, we also store the output string, which contains N values and roughly N+1 nulls. Therefore, total space is dominated by the storage of the output: **O(N)**.

---

4. Solution Code

Python Solution Using an iterator is clean and avoids manually passing an index variable around.

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

class Codec:

    def serialize(self, root):
        """Encodes a tree to a single string.

        Args:
            root: The root TreeNode of the binary tree.
        Returns:
            A string like "1,2,#,#,3,#"
        """
        vals = []

        def dfs(node):
            if not node:
                vals.append("#")
                return

            # Preorder: Root -> Left -> Right
            vals.append(str(node.val))
            dfs(node.left)
            dfs(node.right)

        dfs(root)
        # Join with a delimiter to handle multi-digit numbers (e.g. "10, 20")
        return ",".join(vals)

    def deserialize(self, data):
        """Decodes your encoded data to tree.

        Args:
            data: The string generated by serialize.
        Returns:
            The root TreeNode.
        """
        if not data:
            return None

        # Split the string back into a list of values
        vals_list = data.split(",")

        # Turn the list into an iterator.
        # This allows us to "consume" the next value just by calling next()
        # without worrying about array indices.
        vals_iter = iter(vals_list)

        def build_tree():
            try:
                val = next(vals_iter)
            except StopIteration:
                return None

            if val == "#":
                return None

            # Create the node
            node = TreeNode(int(val))

            # Recursively build left and right children
            # The iterator state is shared, so the 'left' call consumes
            # exactly as many tokens as it needs, leaving the rest for 'right'.
            node.left = build_tree()
            node.right = build_tree()

            return node

        return build_tree()

# Your Codec object will be instantiated and called as such:
# ser = Codec()
# deser = Codec()
# ans = deser.deserialize(ser.serialize(root))
```

Javascript Solution In JavaScript, we don't have Python's convenient `iter()`, so we simply shift items off the front of the array (treating it like a queue).

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 * this.val = val;
 * this.left = this.right = null;
 * }
 */

/**
 * Encodes a tree to a single string.
 *
 * @param {TreeNode} root
 * @return {string}
 */
var serialize = function(root) {
    const output = [];

    function dfs(node) {
        if (!node) {
            output.push("#");
            return;
        }

        // Preorder processing
        output.push(node.val);
        dfs(node.left);
        dfs(node.right);
    }

    dfs(root);
    return output.join(",");
};

/**
 * Decodes your encoded data to tree.
 *
 * @param {string} data
 * @return {TreeNode}
 */
```

```
var deserialize = function(data) {
    if (!data) return null;

    // Split into an array
    const tokens = data.split(",");

    function build() {
        // Shift removes the first element from the array and returns it
        // This acts like our "pop from queue"
        if (tokens.length === 0) return null;

        const val = tokens.shift();

        if (val === "#") {
            return null;
        }

        const node = new TreeNode(parseInt(val));

        // Recursion naturally flows to left first, then right
        node.left = build();
        node.right = build();

        return node;
    }

    return build();
};
```

Note 1: New Terms / Techniques

Preorder Traversal (DFS): A method of visiting nodes in a tree. You visit the current node first ("Pre"), then visit the left branch, then the right branch.

- *Why it helps here:* It places the Root value at the very start of the data stream. This is crucial because when we start reading the stream to rebuild the tree, the first thing we need is the Root to hang everything else off of.

Serialization: The process of translating a data structure or object state into a format that can be stored (for example, in a file or memory buffer) or transmitted (for example, across a network connection) and reconstructed later.

Note 2: Real World & Interview Variations

If you are interviewing at Google, Meta, or Bloomberg, they might not ask this exact LeetCode question directly. Instead, they will disguise it as a practical system design or data manipulation problem:

1. **DOM Tree Transmission (Meta/Google):**

- *Question:* "How would you send the structure of a simplified HTML DOM tree from a server to a client if you couldn't use HTML syntax?"
- *Connection:* The DOM is just a tree. You are essentially serializing a tree of generic nodes (divs, spans) into a JSON or binary stream.

2. **File System Snapshots (Google):**

- *Question:* "Design a backup system for a directory structure. How do you store the hierarchy in a single flat file so it can be restored later?"
- *Connection:* Directories are tree nodes. Files are leaves. This is exactly Tree Serialization.

3. **Expression Trees (Bloomberg):**

- *Question:* "Save a mathematical expression like `((3 + 5) * 2)` to a database so it can be evaluated later."
- *Connection:* This expression is a binary tree where `*` is the root, `+` and `2` are children. You must serialize it to store it.

572. Subtree of Another Tree

Here is how a Senior Staff Engineer (L6) would break down, solve, and analyze **Leetcode 572: Subtree of Another Tree**.

1. Problem Explanation

The Core Question: Imagine you have two binary trees:

1. `root` (The Big Tree): The main structure.
2. `subRoot` (The Small Tree): The pattern we are looking for.

We need to answer **True** or **False**: Does the `subRoot` exist exactly inside the `root`?

"Exactly" means:

- The structure of the nodes must be identical.
- The values inside the nodes must be identical.
- If we find the `subRoot` inside `root`, the `subRoot` must not have any extra descendants hanging off the bottom that the match in `root` doesn't have. They must end at the same place (leaves match leaves).

Visual Examples **Example 1: A Perfect Match**

```
[Big Tree: root]                  [Small Tree: subRoot]

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

```
Result: TRUE
Reason: Look at the left side of the Big Tree (node 4). The tree rooted at 4
is identical to the subRoot.
```

Example 2: Value Mismatch

```
[Big Tree: root]                  [Small Tree: subRoot]

      3                                    4
     / \                                  / \
    4   5                                1   9
   / \
  1   2
```

```
Result: FALSE
Reason: The structure matches, but the node value '2' in the Big Tree
does not match the value '9' in the Small Tree.
```

Example 3: Structure Mismatch (The "Extra Leg" Problem)

```
[Big Tree: root]                  [Small Tree: subRoot]

      3                                    4
     / \                                  / \
    4   5                                1   2
   / \
  1   2
     /
    0
```

```
Result: FALSE
Reason: Even though the values 4, 1, and 2 exist in that arrangement in the
Big Tree, the Big Tree has an extra child (0) attached to 2. The subRoot
stops at 2. Therefore, they are not identical.
```

---

2. Solution Explanation

An L5/L6 engineer values **readability** and **maintainability** first. While there are complex ways to solve this (like hashing), the most robust solution relies on a clear algorithmic strategy: **"Traverse and Check"**.

The Strategy:

1. We visit every single node in the Big Tree (`root`).
2. Treat the current node as a potential "anchor" point.
3. Ask a helper function: "If I start at this anchor, is the tree identical to `subRoot`?"
4. If **Yes** -> We are done! Return True.
5. If **No** -> Move to the left child and right child and repeat step 2.

To do this, we need two functions:

1. **`isSameTree(p, q)`**: A helper that strictly checks if two specific trees are clones.
2. **`isSubtree(root, subRoot)`**: The main explorer that moves through the Big Tree.

Step-by-Step Walkthrough with Diagrams Let's trace **Example 1**.

Phase 1: Start at the Top

```
Current Anchor: Node 3 (Root of Big Tree)

     Big Tree Scope            Small Tree
          3                        4
         / \                      / \
        4...                     1   2
```

We call `isSameTree(3, 4)`.

- Compare values: 3 vs 4.
- **Mismatch!**
- Move to children of 3: Try Anchor 4 and Anchor 5.

Phase 2: Try Left Child

```
Current Anchor: Node 4 (Left child of 3)

     Big Tree Scope            Small Tree
          4                        4
         / \                      / \
        1   2                    1   2
```

We call `isSameTree(4, 4)`.

1. Compare Roots: 4 vs 4. **Match.**
2. Now we must check *both* left and right sub-branches simultaneously.

Phase 2a: Check Left Sub-branch

```
      Big Tree (Left)            Small Tree (Left)
           1                          1
```

- Roots: 1 vs 1. **Match.**
- Children: Both have no children (None/Null). **Match.**
- Left side is good.

Phase 2b: Check Right Sub-branch

```
      Big Tree (Right)           Small Tree (Right)
           2                          2
```

- Roots: 2 vs 2. **Match.**
- Children: Both have no children. **Match.**
- Right side is good.

Conclusion: Since Anchor 4 matches the root, and the left/right families match perfectly, we return **TRUE**.

---

3. Time and Space Complexity Analysis

An L6 engineer must visualize the "cost" of the algorithm to ensure it scales.

Time Complexity: O(N * M)

- Let **N** = Number of nodes in the Big Tree (`root`).
- Let **M** = Number of nodes in the Small Tree (`subRoot`).

Why O(N * M)? In the worst-case scenario, the trees look very similar, forcing us to check deeply at every node.

Visualizing the Worst Case:

Imagine `subRoot` is a long line of A's ending in B. Imagine `root` is just a massive line of A's.

```
Big Tree (N nodes):   A -> A -> A -> A -> A -> ... -> A
Small Tree (M nodes): A -> A -> B
```

At **every single node** of the Big Tree, we initiate a check that runs almost to the end of the Small Tree before realizing it's a mismatch.

```
Iteration 1:
   Big:   A -> A -> A...
   Small: A -> A -> B
          ^    ^    ^
          OK   OK   Fail! (Cost: M operations)

Iteration 2:
```

```
Big:        A -> A -> A...
Small:      A -> A -> B
            ^    ^    ^
            OK   OK   Fail! (Cost: M operations)
```

We repeat this `M` cost for `N` nodes. **Total Time = N x M**

Space Complexity: O(N + M)

We are using recursion (Depth First Search). This uses the "Call Stack" (memory used by the computer to track functions calling other functions).

1. We recurse through the Big Tree (Depth N in worst case, e.g., a straight line).
2. Inside the check, we recurse through the Small Tree (Depth M).

However, since `isSameTree` returns before we continue traversing `root`, the max stack depth is determined by the height of the trees. **Total Space = O(Height of Big Tree + Height of Small Tree)**. In the worst case (skewed trees), this is **O(N + M)**.

---

4. Solution Code

Python Solution

```python
# 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

class Solution:
    def isSubtree(self, root: Optional[TreeNode], subRoot: Optional[TreeNode]) -> bool:
        """
        Main function: Traverses 'root' to find a starting point
        that matches 'subRoot'.
        """

        # Base Case 1: If the Big Tree is empty, we can't find anything in it.
        # (Unless subRoot is also None, but problem constraints usually say non-empty)
        if not root:
            return False

        # Check 1: Is the tree starting RIGHT HERE identical?
        if self.isSameTree(root, subRoot):
            return True
        # Check 2: If not, maybe the match is in the left child OR the right child?
        return self.isSubtree(root.left, subRoot) or self.isSubtree(root.right, subRoot)

    def isSameTree(self, p: Optional[TreeNode], q: Optional[TreeNode]) -> bool:
        """
        Helper function: Checks if two trees are identical clones
        (Same structure, same values).
        """

        # Case A: Both nodes are None (we reached the bottom successfully).
        # This is a Match.
        if not p and not q:
            return True

        # Case B: One is None and the other is not.
        # This is a Structure Mismatch.
        if not p or not q:
            return False

        # Case C: Both nodes exist, but values are different.
        # This is a Value Mismatch.
        if p.val != q.val:
            return False

        # Case D: Current nodes match. Now check BOTH children recursively.
        return self.isSameTree(p.left, q.left) and self.isSameTree(p.right, q.right)
```

JavaScript Solution

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 * this.val = (val===undefined ? 0 : val)
 * this.left = (left===undefined ? null : left)
 * this.right = (right===undefined ? null : right)
 * }
 */

/**
 * @param {TreeNode} root
 * @param {TreeNode} subRoot
 * @return {boolean}
 */
var isSubtree = function(root, subRoot) {
    // If we have exhausted the Big Tree without finding a match, return false.
    if (!root) {
        return false;
    }

    // Check if the tree rooted at the current node matches the subRoot completely.
    if (isSameTree(root, subRoot)) {
        return true;
    }

    // If not, try to find the match in the left subtree OR the right subtree.
    return isSubtree(root.left, subRoot) || isSubtree(root.right, subRoot);
};

/**
 * Helper function to strict compare two trees
 * @param {TreeNode} p
 * @param {TreeNode} q
 * @return {boolean}
 */
var isSameTree = function(p, q) {
    // 1. Both are null -> Identical end of branch.
    if (!p && !q) return true;

    // 2. One is null, the other is not -> Structure mismatch.
    if (!p || !q) return false;

    // 3. Values do not match -> Value mismatch.
    if (p.val !== q.val) return false;

    // 4. Current nodes match, recurse deeper into both left and right sides.
    return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
};
```

Note 1: Terminology & "L6" Optimization

DFS (Depth First Search): The technique used above is DFS. It means we dive deep into a branch before moving to the next one, rather than checking level-by-level.

The "L6" Optimization (Merkle Hashing / Serialization): While the O(N * M) solution above is standard, an L6 engineer might mention an optimization to make this **O(N + M)**.

- **Technique:** Serialization (turning the tree into a unique string).
- **How:** You convert the `root` tree into a string (e.g., `...4,1,#,#,2,#,#...`)

and the `subRoot` into a string.

- **The Trick:** Then, you just check if the `subRoot` string is a substring of the `root` string using a string searching algorithm (like KMP).
- **Why strict L5/L6s prefer the recursive solution first:** String serialization has tricky edge cases. For example, node value "12" might look like "1" and "2" next to each other. You need careful delimiters. The recursive solution is less prone to bugs in a 45-minute interview.

Note 2: Real World & Indirect Interview Questions

Companies like Google, Meta, and Bloomberg rarely ask "Check if subtree exists" directly anymore. They wrap it in real-world scenarios:

1. **Google (The DOM Tree):**

- *Question:* "Given a large HTML Document Object Model (DOM) and a small HTML snippet, find if the snippet exists exactly within the page."
- *Context:* This is used in web scrapers or test automation frameworks to locate elements.

2. **Meta (Code Plagiarism):**

- *Question:* "We have a massive codebase (represented as an Abstract Syntax Tree). We have a known function that contains a vulnerability or is copied code. Find all occurrences of this function structure in the codebase."
- *Context:* Detecting malicious code patterns or copy-paste refactoring.

3. **Bloomberg (Financial Tickers):**

- *Question:* "You have a hierarchy of financial instruments (Portfolio -> Sector -> Stock). Find if a specific 'sector configuration' exists within a larger portfolio."

105. Construct Binary Tree from Preorder and Inorder Traversal

Here is how a Senior Software Engineer (L5/L6) at Google approaches this problem. We move beyond "getting it right" to "getting it efficient and readable," focusing on **Divide and Conquer** and **Hash Map Optimization**.

1. Problem Explanation

We are given two arrays representing the same Binary Tree:

1. **Preorder Traversal:** Process Root, then Left Subtree, then Right Subtree.

- *Key Trait:* The **first** element is always the **Root**.

2. **Inorder Traversal:** Process Left Subtree, then Root, then Right Subtree.

- *Key Trait:* Once we find the Root, everything to its **left** is the Left Subtree, and everything to its **right** is the Right Subtree.

The Goal: Reconstruct the exact unique binary tree structure.

Visualizing the Inputs:

Let's say our tree looks like this:

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

The inputs provided to us would be:

```
Index:     0  1   2   3  4
--------------------------
Preorder: [3, 9, 20, 15, 7]  (Root -> Left -> Right)
Inorder:  [9, 3, 15, 20, 7]  (Left -> Root -> Right)
```

---

2. Solution Explanation (The "Divide and Conquer" Strategy)

An L5 engineer immediately spots the recursive pattern.

The Algorithm:

1. **Identify Root:** Look at `Preorder[0]`. This is the current root.
2. **Locate Root in Inorder:** Find the index of this value in the `Inorder` array.
3. **Calculate Sizes:** The number of elements to the left of the root in `Inorder` tells us exactly how large the Left Subtree is.
4. **Divide:**

- Slice the `Preorder` and `Inorder` arrays into Left and Right sets based on that size.
- Recursively build the Left Child.
- Recursively build the Right Child.

The Bottleneck (Non-Trivial Part): A naive solution iterates through the `Inorder` array to find the root's index every single time. This results in O(N*N) time complexity (slow). **The L5 Optimization:** We create a **Hash Map** (Dictionary) first. This maps `Value -> Index` for the Inorder array. This allows us to find the root's position in O(1) time.

Detailed ASCII Walkthrough Let's trace the reconstruction of the example tree.

Initial State: Preorder: [3, 9, 20, 15, 7] Inorder: [9, 3, 15, 20, 7] Map: {9:0, 3:1, 15:2, 20:3, 7:4}

Step 1: The Root Call

- Current Root Value = Preorder[0] = **3**.
- Look up **3** in Map -> Index **1**.
- **Left Subtree Size**: Index 1 - Start Index 0 = **1 element**. Visualization of the Split:

```
   [Root: 3]
       |
 /-----+-----\
/             \
```

Left Logic Right Logic (1 element) (3 elements) Pre: [9] Pre: [20, 15, 7] In: [9] In: [15, 20, 7]

Step 2: Recurse Left (Building node 9)

- Input Pre: [9], In: [9]
- Current Root = **9**.
- Look up **9** in Map -> Index **0**.
- Left Subtree Size = 0.

```
   [Node: 9]
   /       \
None        None
```

(Base Case) (Base Case) *Result: Node 3's left child is 9.*

Step 3: Recurse Right (Building node 20)

- Input Pre: [20, 15, 7], In: [15, 20, 7]
- Current Root = **20**.
- Look up **20** in Map -> Index **3**.
- **Context:** In current range (indices 2 to 4), 20 is at index 3.
- Left Subtree Size = Index 3 - Index 2 = **1 element** (the number 15).

```
   [Node: 20]
       |
 /-----+-----\
/             \
```

Left Logic Right Logic (1 element) (1 element) Pre: [15] Pre: [7] In: [15] In: [7]

Step 4: Recurse Right's Children

- **Left Child of 20:** Pre=[15], In=[15]. Root is 15. No children. Return Node **15**.

- **Right Child of 20:** Pre=[7], In=[7]. Root is 7. No children. Return Node **7**.

Final Tree Assembly:

```
  3  <-- (Step 1 assembled this)
 / \
9   20 <-- (Step 3 assembled this)
   /  \
 15    7 <-- (Step 4 assembled these)
```

---

3. Time and Space Complexity Analysis

We avoid complex notation and look at the "Physical" cost of the operations.

Time Complexity: O(N) Where N is the number of nodes in the tree.

Derivation:

```
1. Building the Hash Map:
   [Read Node 1] -> [Read Node 2] -> ... -> [Read Node N]
   Cost: N operations.

2. Tree Construction (Recursion):
   We visit every node exactly once to create it.
   Inside each visit, we do constant time math (lookup index, calculate size).

   Visualization of Time Spent:
   [Create Node 3] (Const time)
   [Create Node 9] (Const time)
   [Create Node 20] (Const time)
   ...
   [Create Node 7] (Const time)
   -----------------------------
   Total = N * Constant = O(N)
```

Space Complexity: O(N) **Derivation:**

```
1. The Hash Map storage:
   Stores N entries (Value -> Index).
   Space: O(N)

2. The Recursion Stack (Memory used by function calls):
   In the worst case (a skewed tree like a linked list), the recursion goes N deep.
   In the average case (balanced tree), it goes log(N) deep.

   Stack Visualization (Worst Case):
```

```
   | Call for Node N    |  ^
   | ...                |  |
   | Call for Node 2    |  | Stack grows to Depth N
   | Call for Node 1    |  |
   +--------------------+

Total Space = Map(N) + Stack(N) = O(N).
```

---

4. Solution Code

We use index pointers (`left`, `right`) rather than slicing arrays. Slicing arrays (e.g., `arr[0:5]`) copies data and increases complexity. Passing indices is the "L6" way to keep memory usage low.

Python Solution

```python
# 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

class Solution:
    def buildTree(self, preorder: list[int], inorder: list[int]) -> TreeNode:
        # NON-TRIVIAL:
        # We assume duplicates do not exist in the tree.
        # We build a hash map for O(1) access to inorder indices.
        # This reduces the overall time complexity from O(N^2) to O(N).
        inorder_map = {val: idx for idx, val in enumerate(inorder)}

        # pre_idx is a mutable reference (list) so it updates across recursive calls.
        # It tracks which node in 'preorder' we are currently processing.
        pre_idx = [0]

        def array_to_tree(left_bound, right_bound):
            # Base Case: If the bounds cross, there are no elements to construct a subtree.
            if left_bound > right_bound:
                return None

            # 1. Pick the current root value using the global preorder index
            root_val = preorder[pre_idx[0]]
            root = TreeNode(root_val)

            # 2. Advance the preorder index for the next recursive call
            pre_idx[0] += 1

            # 3. Find where the root splits the inorder array
            # This logic determines the boundaries of left and right subtrees
            mid = inorder_map[root_val]

            # 4. Recursion
            # IMPORTANT: We must build the left subtree first because
            # Preorder traversal is Root -> Left -> Right.
            # The next element in preorder is guaranteed to be the left child (if it exists
            root.left = array_to_tree(left_bound, mid - 1)
            root.right = array_to_tree(mid + 1, right_bound)

            return root

        return array_to_tree(0, len(inorder) - 1)
```

JavaScript Solution

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 * this.val = (val===undefined ? 0 : val)
 * this.left = (left===undefined ? null : left)
 * this.right = (right===undefined ? null : right)
 * }
 */

/**
 * @param {number[]} preorder
 * @param {number[]} inorder
 * @return {TreeNode}
 */
var buildTree = function(preorder, inorder) {
    // 1. Optimization: Map for O(1) lookup
    const inorderMap = new Map();
    inorder.forEach((val, index) => inorderMap.set(val, index));

    // Global pointer for preorder array
    let preIndex = 0;

    // Helper function
    // left and right represent the range in the INORDER array
    function arrayToTree(left, right) {
        // Base case: If no elements remain in this range
        if (left > right) {
            return null;
        }

        // The current root is always the current element in preorder
        const rootVal = preorder[preIndex];
        const root = new TreeNode(rootVal);

        // Advance global pointer
        preIndex++;

        // Split point in inorder array
        const mid = inorderMap.get(rootVal);

        // Recursively build left and right subtrees
        // We use (mid - 1) because mid is the root itself
        root.left = arrayToTree(left, mid - 1);
        root.right = arrayToTree(mid + 1, right);

        return root;
    }

    return arrayToTree(0, inorder.length - 1);
};
```

Note 1: Terminology & Techniques

1. Divide and Conquer: This is the core paradigm used here. We break a large problem (construct a big tree) into two smaller, identical problems (construct the left subtree, construct the right subtree) and combine the results.

2. Hash Map Optimization (Space-Time Tradeoff): We sacrificed `O(N)` space (to store the map) to gain a significant speed increase (removing the search loop). In interviews, explicitly mentioning "I am trading space for time here" is a strong signal of seniority.

Note 2: Indirect & Real-World Interview Variations

Big tech companies (Google, Meta, Bloomberg) rarely ask this verbatim anymore. They wrap it in practical scenarios:

1. **Serialization/Deserialization (System Design/Parsing):**

- *Question:* "Design a protocol to send a DOM tree from a server to a client efficiently."
- *Context:* You might send the tree as a flat string (Preorder) to save bandwidth. The client needs to reconstruct it. If the DOM has unique IDs

(like the values in our tree), you use this exact logic to rebuild the DOM.

2. **Log Reconstruction:**

- *Question:* "We have a server log of function calls (Enter Function A, Enter Function B…). We lost the 'Exit' logs but we have a separate list of which functions were called in alphabetical order. Reconstruct the Call Stack."
- *Context:* 'Enter' logs are Preorder. Alphabetical lists act like Inorder (if the tree was a Binary Search Tree).

3. **The "Ambiguous" Variation:**

- *Question:* "Can you construct a tree from Preorder and Postorder?"
- *Answer:* No, not uniquely, unless it is a "Full" binary tree (every node has 0 or 2 children). Discussing *why* this works for Pre+In but not Pre+Post is a common follow-up to test depth of understanding.

98. Validate Binary Search Tree

Here is how a Senior Software Engineer (L5/L6) would break down and solve **Leetcode 98: Validate Binary Search Tree**.

At this level, we look beyond just "getting it to work." We focus on **correctness, code maintainability, handling edge cases (like integer limits), and clarity.**

---

1. Problem Explanation

The core task is to determine if a given Binary Tree is a valid **Binary Search Tree (BST)**.

The Golden Rules of a BST: For *every* node in the tree:

1. **Left Condition:** All nodes in its **left** subtree must be strictly **smaller** than the node's value.
2. **Right Condition:** All nodes in its **right** subtree must be strictly **larger** than the node's value.
3. **Recursive Condition:** Both the left and right subtrees must also be valid BSTs.

Common Pitfall (The "Local" Trap): Junior engineers often just check if `left.val < current.val < right.val`. This is incorrect because it only checks immediate children, not the entire subtree.

Visualizing the Pitfall:

```
      (5)   <-- Root
     /   \
   (1)   (7)  <-- Right child (7) > Root (5). Good.
```

```
     / \
   (4) (8)
    ^
    |
  Error!
```

- Look at node (4).
- Local check: 4 is less than its parent 7. (Seems okay locally).
- Global check: 4 is in the **Right Subtree** of 5. Therefore, it **MUST** be greater than 5.
- Since 4 < 5, this tree is **INVALID**.

---

2. Solution Explanation: The "Valid Range" Approach

The most robust way to solve this is to pass down a **constraint range** (min, max) to every node as we traverse.

The Algorithm:

1. Start at the `Root`. The valid range is effectively negative infinity to positive infinity `(-inf, +inf)`.
2. When moving to a **Left Child**, the node must be smaller than its parent.

- Update the `Max` limit to the parent's value.
- Keep the `Min` limit the same.

3. When moving to a **Right Child**, the node must be larger than its parent.

- Update the `Min` limit to the parent's value.
- Keep the `Max` limit the same.

4. If a node's value falls outside its inherited range, the tree is invalid.

Visual Walkthrough Let's validate the tree from the "Pitfall" example above using Ranges.

Step 1: Process Root (5)

- Allowed Range: `(-inf, +inf)`
- Is `5` in range? Yes.
- Action: Split into children.

```
   (5) [-inf, +inf]
  /   \
Left  Right
```

Step 2: Process Left Child (1)

- Going Left, we tighten the **Max**. New Max = Parent(5).
- Inherited Range: `(-inf, 5)`

- Is `1` in range `(-inf, 5)`? Yes.

Step 3: Process Right Child (7)

- Going Right, we tighten the **Min**. New Min = Parent(5).
- Inherited Range: `(5, +inf)`
- Is `7` in range `(5, +inf)`? Yes.

```
      (5)
     /   \
   (1)   (7) [5, +inf]  <-- Current Node
         / \
       (4) (8)
```

Step 4: Process Right->Left Child (4)

- We are at Node `(7)`. Going **Left** to Node `(4)`.
- We tighten the **Max** to the parent `(7)`.
- We keep the **Min** inherited from `(7)`, which was `5`.
- **New Range for Node (4): `(5, 7)**`

```
Check Node (4)
Range: (Min: 5, Max: 7)

Is 4 >= 5 AND 4 < 7 ?
NO! 4 is NOT >= 5.

FAIL -> Return False.
```

Step 5: Process Right->Right Child (8)

- We are at Node `(7)`. Going **Right** to Node `(8)`.
- We tighten the **Min** to parent `(7)`.
- We keep the **Max** inherited from `(7)`, which was `+inf`.
- **New Range for Node (8): `(7, +inf)**`
- Is `8` in `(7, +inf)`? Yes.

Logic Flow (ASCII)

```
FUNC validate(Node, min_val, max_val):

    1. Base Case: If Node is Null?
       -> RETURN True (Empty trees are valid BSTs)

           (null)
             ^
       Is Valid? YES.

    2. Check Constraints:
       If (Node.val <= min_val) OR (Node.val >= max_val)?
```

```
        -> RETURN False

           Range(5, 10)
              |
             (4)  <-- 4 <= 5? FAIL!

     3. Recursive Step (The "Divide and Conquer"):
        RETURN
           validate(Node.left,  min_val,  Node.val)  <-- Tighter Max
              AND
           validate(Node.right, Node.val, max_val)   <-- Tighter Min
```

---

3. Time and Space Complexity Analysis

Time Complexity: O(N) We visit every node exactly once.

Derivation: Imagine the function calls as a flow of water touching every node.

```
      (5)      <-- Visit #1
     /   \
   (1)   (7)   <-- Visit #2, #3
        / \
      (6) (8) <-- Visit #4, #5

Total Nodes (N) = 5
Total Visits    = 5
Operations per visit = Constant (Comparison checks)

Total Time = c * N  -> O(N)
```

Space Complexity: O(N) [Worst Case] / O(H) [Average] This is determined by the **Recursion Stack** (how deep the function calls go).

Scenario A: Skewed Tree (Worst Case) The tree looks like a linked list. We recurse all the way down before returning.

```
Stack Frame 1: Root(10)
  |
  +-- Stack Frame 2: Left(9)
        |
        +-- Stack Frame 3: Left(8)
              |
              ...
              +-- Stack Frame N: Left(1)
```

```
Stack Height = N
Space = O(N)
```

Scenario B: Balanced Tree (Best/Average Case) The tree splits evenly.

```
          (10)       <-- Frame 1
         /    \
       (5)    (15)   <-- Frame 2 (max depth at any time)
      /   \
    (2)   (7)        <-- Frame 3

Stack Height = Height of Tree = log(N)
Space = O(log N) or O(H)
```

---

4. Solution Code

Python Solution We use `float('-inf')` and `float('inf')` to represent the initial open boundaries.

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def isValidBST(self, root: TreeNode) -> bool:
        # We define a helper function to carry the Min/Max state
        # distinct from the main function signature.
        def validate(node, lower_limit, upper_limit):

            # Base Case: An empty node (None) is technically a valid BST.
            # We reached the bottom of a branch successfully.
            if not node:
                return True

            # The Main Check:
            # The node's value must be strictly strictly between lower and upper limits.
            if not (lower_limit < node.val < upper_limit):
                return False

            # Recursive Step:
            # 1. Check Left: Max limit becomes current node's value.
            # 2. Check Right: Min limit becomes current node's value.
            return (validate(node.left, lower_limit, node.val) and
                    validate(node.right, node.val, upper_limit))
        # Initial call with infinite boundaries
        return validate(root, float('-inf'), float('inf'))
```

JavaScript Solution In JavaScript, we use simple `null` checks or `-Infinity/Infinity`.

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 * this.val = (val===undefined ? 0 : val)
 * this.left = (left===undefined ? null : left)
 * this.right = (right===undefined ? null : right)
 * }
 */

/**
 * @param {TreeNode} root
 * @return {boolean}
 */
var isValidBST = function(root) {

    // Helper function to maintain state (min/max constraints)
    // traversing down the tree.
    const validate = (node, min, max) => {
        // Base Case: We hit the bottom (null), so this path is valid.
        if (!node) {
            return true;
        }

        // Check 1: Does the current node violate the Min constraint?
        // (Must be strictly greater than min)
        if (min !== null && node.val <= min) {
            return false;
        }

        // Check 2: Does the current node violate the Max constraint?
        // (Must be strictly less than max)
        if (max !== null && node.val >= max) {
            return false;
        }

        // Recurse:
        // For the Left child, the current node becomes the new Ceiling (Max).
        // For the Right child, the current node becomes the new Floor (Min).
        return validate(node.left, min, node.val) &&
            validate(node.right, node.val, max);
    };

    // Start with no constraints (null, null)
    return validate(root, null, null);
};
```

Note 1: Terminology

Depth First Search (DFS) / Pre-order Traversal: The technique used above is a variation of DFS. Specifically, it is a Pre-order traversal because we validate the "Current Node" first (checking the range) before we visit the children (Left and Right).

- **Why it helps:** It allows us to "fail fast." If the root is invalid, we return `False` immediately without wasting time checking the thousands of nodes beneath it.

Note 2: Real World & Interview Context

How is this asked indirectly?

1. **Bloomberg / Financial Systems:**

- *Question:* "Verify the integrity of a timestamp-based transaction log organized as a tree."
- *Context:* Transactions must be strictly ordered. If a transaction at 10:00 AM appears in the "after 11:00 AM" branch, the database index is corrupted.

2. **Google / File Systems:**

- *Question:* "You have a hierarchy of file permissions where child folders inherit restrictions. Validate that no child folder grants more permission than its parent allows."
- *Context:* This is the same logic. `Child Permission <= Parent Permission`. You pass the "Max Permission" down the tree, just like passing the `Max Value`.

3. **Meta / UI Rendering:**

- *Question:* "Validate a rendering tree where z-indexes (layer depth) must increase as you go deeper into certain component types."
- *Context:* Ensuring strictly increasing order in a hierarchy.

230. Kth Smallest Element in a BST

Here is a breakdown of how a Senior Staff (L6) Engineer at Google would approach, analyze, and solve the "Kth Smallest Element in a BST" problem, keeping readability and system limitations in mind.

1. Problem Explanation

The Core Concept We are given a **Binary Search Tree (BST)** and an integer **k**. Our goal is to find the **k-th** smallest value in that tree.

- If k = 1, we want the minimum value.
- If k = 2, we want the second smallest value.
- And so on.

The "Why This Matters" Check Before jumping into code, an L6 engineer verifies the properties of the data structure to exploit them.

- **BST Property:** For every node, all nodes in its **left** subtree are smaller, and all nodes in its **right** subtree are larger.
- **The Consequence:** If you read a BST by going `Left -> Node -> Right` (this is called an **In-order Traversal**), you get the values in a perfectly sorted ascending order.

Visualizing the Problem

Imagine this BST structure. We want to find the **3rd smallest** element (`k = 3`).

```
      5
     / \
    3   6
   / \
  2   4
 /
1
```

If we flatten this tree using the BST property (Left, then Node, then Right), it looks like a sorted array:

```
Sorted View: [1, 2, 3, 4, 5, 6]
              ^  ^  ^
              |  |  |
k=1 (1st) ----+  |  |
k=2 (2nd) -------+  |
k=3 (3rd) ----------+ -> The answer is 3.
```

---

2. Solution Explanation

The Approach: Iterative In-order Traversal

While a recursive solution is simple to write, an L5/L6 engineer often prefers an **Iterative** approach using a **Stack**.

Why?

1. **Control:** We can stop the moment we find the `k-th` element. Recursion sometimes requires more complex flags to stop the unwinding process immediately.
2. **Stack Safety:** In languages with limited recursion depth, an iterative approach using an explicit heap-allocated stack is more robust for very deep, skewed trees.

The Algorithm

1. Initialize an empty `Stack`.
2. Start with the `Current` node at the `Root`.
3. **Dive Left:** Push `Current` and all its left children onto the stack until you hit `None` (null). (We are looking for the smallest numbers first).
4. **Pop and Process:**

- Pop the top node from the stack.
- This is the next smallest number in the sequence.
- Decrement `k`.
- If `k == 0`, we found our target! Return this node's value.

5. **Go Right:** Set `Current` to the popped node's right child and repeat step 3.

Visual Walkthrough (The "Debug" View)

Let's trace `k = 3` on the tree below.

```
      5
     / \
    3   6
   / \
  2   4
 /
1
```

Step 1: Dive Left We start at 5 and keep going left, pushing everything onto the stack.

```
Processing: Pushing 5, then 3, then 2, then 1.

Stack Visualization (Top is right):
[ 5, 3, 2, 1 ]
```

```
              ^
              Top
```

Step 2: Pop (1st Smallest) We hit a dead end (1 has no left child). We pop from the stack.

```
Pop: 1
Stack: [ 5, 3, 2 ]
k was 3.
Decrement k -> k is now 2.
Is k == 0? No.
Current moves to Right of 1 -> None.
```

Step 3: Dive Left (Skipped) Current is None, so we don't push anything.

Step 4: Pop (2nd Smallest) We pop the next available node from the stack.

```
Pop: 2
Stack: [ 5, 3 ]
k was 2.
Decrement k -> k is now 1.
Is k == 0? No.
Current moves to Right of 2 -> None.
```

Step 5: Pop (3rd Smallest) Pop again.

```
Pop: 3
Stack: [ 5 ]
k was 1.
Decrement k -> k is now 0.
Is k == 0? YES!

Answer: 3
```

---

3. Time and Space Complexity Analysis

An L6 engineer communicates complexity not just with "Big O" but by explaining the "Worst Case" topology.

Time Complexity: O(H + k)

- **H** is the height of the tree.
- **k** is the target index.

Why? We have to go down to the bottom-left leaf first (O(H) work). Then, we traverse `k` nodes upwards/sideways to find our target. Since `k` can be close to `N` (total nodes) in the worst case, this is often simplified to **O(N)**.

```
Timeline of Operations:
```

```
Start
  |
  | (Drill down left: Height of tree) -> Cost: H
  |
  v
Smallest Node
  |
  | (Pop, verify, go right: Repeat k times) -> Cost: k
  |
  v
Target Found
```

Space Complexity: O(H)

- We need a stack to remember our path so we can move back up.
- In the worst case (a generic skewed tree), the stack holds all nodes from the root to the deepest leaf.

Space Visualization (Worst Case vs Best Case)

Case A: Skewed Tree (Worst Case) The tree is just a line. We push everything.

```
  5
 /
4       Stack needs to hold: [5, 4, 3, 2, 1]
/       Space = O(N) which is equal to Height.
3
/
...
```

Case B: Balanced Tree (Best Case) The tree is full. Height is `log N` (base 2).

```
      4
    /   \
   2     6    Stack holds max: [4, 2, 1]
  / \   / \   Space = O(log N) which is equal to Height.
 1   3 5   7
```

---

4. Solution Code

Here is the implementation. Note the defensive checks and clear variable naming.

Python Solution

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def kthSmallest(root, k):
    """
    Finds the kth smallest element in a BST using iterative in-order traversal.
    """
    # The stack simulates the recursive call stack.
    stack = []

    # 'current' acts as our pointer traversing the tree.
    current = root

    # We continue as long as we have a node to visit OR the stack is not empty.
    while current or stack:

        # 1. Reach the left-most node of the current subtree.
        while current:
            stack.append(current)
            current = current.left

        # 2. Pop the node. This is the next smallest element in order.
        current = stack.pop()

        # 3. Process the node.
        k -= 1
        if k == 0:
            return current.val

        # 4. Move to the right subtree.
        # If right is None, the loop continues and pops the parent from stack.
        current = current.right

    return -1 # Should not happen if k is valid
```

JavaScript Solution

```javascript
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 * this.val = (val===undefined ? 0 : val)
 * this.left = (left===undefined ? null : left)
 * this.right = (right===undefined ? null : right)
 * }
 */

/**
 * @param {TreeNode} root
 * @param {number} k
 * @return {number}
 */
var kthSmallest = function(root, k) {
    // Stack to keep track of the path
    const stack = [];
    let current = root;

    while (current !== null || stack.length > 0) {

        // Go as deep left as possible
        while (current !== null) {
            stack.push(current);
            current = current.left;
        }

        // Pop the element (Backtracking)
        current = stack.pop();

        // Decrement k. If k hits 0, we found our node.
        k--;
        if (k === 0) {
            return current.val;
        }

        // Move to the right child
        current = current.right;
    }

    return -1; // Fallback
};
```

Note 1: Terminology Clarification

In-order Traversal: A method of visiting nodes in a tree where you visit the Left child, then the Node itself, then the Right child. In BSTs, this guarantees you visit nodes from smallest to largest value.

Augmented Tree (The "L6" optimization): If this problem were "dynamic" (meaning we insert/delete nodes often and *then* ask for the k-th smallest many times), the O(N) solution above is too slow. An L6 engineer would suggest modifying the `TreeNode` class to store a `count` or `subtree_size` variable.

- `node.count` = 1 + size of left subtree + size of right subtree.
- By looking at `root.left.count`, we can decide instantly whether the k-th element is in the left, is the root, or is in the right, bringing the time complexity down to **O(Height)** or **O(log N)**.

Note 2: Real World / Interview Context

At companies like Google, Meta, and Bloomberg, you won't always get this exact problem. They mask it in "real world" scenarios:

1. **The "Median" Problem:** "Given a continuous stream of numbers, how do you efficiently find the median?" (This is essentially finding the `k`-th element where `k = Total_Elements / 2`).
2. **Database Query Engine:** "Design a function for a database that supports `SELECT * FROM table ORDER BY value LIMIT 1 OFFSET k`."
3. **Ranking Systems:** "We have a leaderboard of game scores updating in real-time. How do you find the score of the player at rank 10,000?"
4. **Weighted Random Selection:** "Pick a random item from a tree, but items with higher 'weights' should be picked more often." (This uses the same logic as the "Augmented Tree" approach mentioned above).

235. Lowest Common Ancestor of a Binary Search Tree

Here is how a Senior (L5) or Staff (L6) Engineer at Google would approach, explain, and solve the "Lowest Common Ancestor (LCA) of a Binary Search Tree" problem.

We prioritize code that is not just "correct," but also computationally efficient, readable, and robust. For this specific problem, leveraging the unique properties of the data structure (the BST) is the key to an optimal solution.

---

1. Problem Explanation

The core of this problem is finding a specific node in a hierarchy.

The Setup: You are given a **Binary Search Tree (BST)**. In a BST, for every single node:

- All values in its **Left** subtree are **smaller** than the node.
- All values in its **Right** subtree are **larger** than the node.

The Goal: Given two nodes, `p` and `q`, find their **Lowest Common Ancestor (LCA)**.

What is an LCA? Imagine a family tree. The LCA of two people is the closest relative (going up the tree) that they both share.

- Technically: The LCA of `p` and `q` is the lowest node `T` such that both `p` and `q` are descendants of `T`.
- **Important Edge Case:** A node is considered a descendant of itself. So, if `p` is a direct child of `q`, then `q` is the LCA.

Visualizing the Goal Consider this BST:

```
      [6]  <-- Root
     /   \
   [2]    [8]
   / \    / \
 [0] [4] [7] [9]
     / \
   [3] [5]
```

- **Example A:** Find LCA of `2` and `8`.
- They meet at the top. The LCA is **6**.
- **Example B:** Find LCA of `2` and `4`.
- Node `4` is actually inside the subtree of `2`. The LCA is **2**.
- **Example C:** Find LCA of `3` and `5`.
- Go up from 3: [3] -> [4] -> [2] -> [6]
- Go up from 5: [5] -> [4] -> [2] -> [6]
- The first (lowest) node they share is **4**.

---

2. Solution Explanation

An L3 (Junior) engineer might solve this by treating the BST like a regular Binary Tree (searching all paths). An L5 (Senior) engineer solves this by **utilizing the BST property** to eliminate half the tree at every step. This turns a generic search into a targeted binary search.

The Algorithm: "The Split Point" We start at the `Root` and traverse down. We are looking for the **Split Point**.

1. **Go Right:** If *both* `p` and `q` are greater than the `Current Node`, the LCA must be to the right. The current node is too small to be the ancestor of both.

2. **Go Left:** If *both* `p` and `q` are smaller than the `Current Node`, the LCA must be to the left. The current node is too big.
3. **Found It (The Split):** If `p` and `q` are on different sides of the `Current Node` (one is smaller, one is larger), OR if the `Current Node` equals `p` or `q`, then **we have found the LCA**.

Why? Because this is the first point where the paths to `p` and `q` diverge. Any node below this will only cover one of them, not both.

ASCII Walkthroughs **Scenario 1: Both targets are to the Left**

- Tree Root: 6
- Targets: p=0, q=4

```
STEP 1: Start at Root [6]
-------------------------
       [6]   <-- Current. Is (0 < 6) AND (4 < 6)? YES.
      /   \        Both are smaller. Go LEFT.
    [2]   [8]
    / \
  [0] [4]


STEP 2: Move to [2]
-------------------
       [6]
      /
    [2]      <-- Current.
    / \          We check: Is p(0) on one side and q(4) on the other?
  [0] [4]        YES. 0 is < 2. 4 is > 2.

RESULT:
The paths "split" here. [2] is the LCA.
```

Scenario 2: The "Current Node is the Target" Case

- Tree Root: 6
- Targets: p=2, q=4

```
STEP 1: Start at Root [6]
-------------------------
       [6]   <-- Current.
      /   \      Is (2 < 6) AND (4 < 6)? YES.
    [2]   [8]    Both are smaller. Go LEFT.
    / \
  [0] [4]
```

```
STEP 2: Move to [2]
-------------------
       [6]
      /
    [2]      <-- Current.
    / \          Wait! The Current Node is equal to p(2).
  [0] [4]

LOGIC:
We cannot go lower. If we go left to [0], we lose [4].
If we go right to [4], we lose [2] (the current node).
Therefore, [2] is the LCA.
```

Scenario 3: Deep Traversal

- Tree Root: 6
- Targets: p=3, q=5

```
       [6]
      /   \
    [2]    [8]
    / \
  [0] [4]
      / \
    [3] [5]

Iteration 1 (at 6):
p(3) < 6 AND q(5) < 6.
Both on Left. Move Current to [2].

Iteration 2 (at 2):
p(3) > 2 AND q(5) > 2.
Both on Right. Move Current to [4].

Iteration 3 (at 4):
p(3) < 4 (Left)
q(5) > 4 (Right)
SPLIT DETECTED!
Return [4].
```

---

3. Time and Space Complexity Analysis

This is how we derive complexity without complex formulas, visualizing the "shape" of the operations.

Time Complexity: O(H) where H is the Height of the tree.

Best Case (Balanced Tree): O(log N) We discard half the nodes at every step. It's just like Binary Search.

```
       [X]      <-- Step 1 (Discard Right half)
      /   \
    [X]   ...   <-- Step 2 (Discard Left half)
    / \
  ... [X]       <-- Step 3 (Found)

Depth is small. Operations = Height of tree.
```

Worst Case (Skewed Tree): O(N) If the tree is just a straight line (essentially a linked list), and our targets are at the bottom, we visit every node.

```
[X]             <-- Step 1
  \
  [X]           <-- Step 2
    \
    [X]         <-- Step 3
      \
      ...       <-- Visits N nodes
```

Space Complexity: O(1) (Iterative Solution) We are using the **Iterative** approach (using a `while` loop) rather than Recursion. This is preferred in production systems at Google because it does not use the Call Stack.

```
Memory Usage Visualization:

[ Heap Memory (Tree Nodes) ]   [ Stack Memory (Our Function) ]
        (Exists already)               (What we use)
       [6]
      /   \                    Variable 'current': [Holds 1 pointer]
    [2]   [8]                  Variable 'p_val':   [Holds Integer]
                               Variable 'q_val':   [Holds Integer]

No matter how deep the tree is (10 nodes or 10 million nodes),
we only ever store these 3 variables in our function's memory.
Thus, Space is Constant: O(1).
```

(Note: If we used Recursion, the space would be O(H) because every step down the tree adds a "frame" to the system stack, which consumes memory.)

---

4. Solution Code

We will use the **Iterative Approach**. It is cleaner, safer (no Stack Overflow on massive trees), and uses less memory.

Python Solution

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

class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeN
        """
        Finds the LCA in a BST using an iterative approach (O(1) Space).
        """

        # We start searching from the top
        current = root

        # Loop until we find the split point or the node itself
        while current:

            # Case 1: Both targets are greater than current.
            # This means the LCA is in the Right subtree.
            if p.val > current.val and q.val > current.val:
                current = current.right

            # Case 2: Both targets are smaller than current.
            # This means the LCA is in the Left subtree.
            elif p.val < current.val and q.val < current.val:
                current = current.left

            # Case 3: The Split Point.
            # One is smaller, one is larger (OR current matches p or q).
            # We have found the LCA.
            else:
                return current

        return None # Should not happen given problem constraints
```

JavaScript Solution

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 * this.val = val;
 * this.left = this.right = null;
 * }
 */

/**
 * @param {TreeNode} root
 * @param {TreeNode} p
 * @param {TreeNode} q
 * @return {TreeNode}
 */
var lowestCommonAncestor = function(root, p, q) {
    // Start traversal at the root
    let current = root;

    while (current !== null) {

        // Case 1: Both p and q are in the right subtree
        if (p.val > current.val && q.val > current.val) {
            current = current.right;
        }

        // Case 2: Both p and q are in the left subtree
        else if (p.val < current.val && q.val < current.val) {
            current = current.left;
        }

        // Case 3: The Split.
        // We found the divergence point. This is the LCA.
        else {
            return current;
        }
    }

    return null;
};
```

Note 1: Terminology & Techniques

- **BST Property:** This is the invariant that `Left < Node < Right`. Recognizing this property allowed us to skip entire sections of the tree, optimizing the search from O(N) to O(H). Without this property (in a standard Binary Tree), we would have to search both left and right subtrees, making the code much slower and more complex.
- **Iterative Traversal:** This refers to using a loop (`while` or `for`) to move through a data structure instead of a function calling itself (Recursion).

In interview settings for tree problems, offering an iterative solution often demonstrates seniority because it shows awareness of memory constraints (Stack Overflow).

Note 2: Real World & Interview Variations

How do top-tier tech companies (Google, Meta, Bloomberg) actually ask this? They rarely ask "Solve Leetcode 235." They wrap it in a practical scenario:

1. **The "Org Chart" Problem (Meta/Bloomberg):**

- *Question:* "Given two employees in a massive corporate hierarchy (CEO at top), find their lowest common manager."
- *Context:* If the org chart is sorted by ID or Department Code (BST structure), use this solution. If not, use the general Tree LCA approach.

2. **The "File System" Problem (Google):**

- *Question:* "You have two file paths: `/usr/bin/java` and `/usr/lib/config`. Write a function to find the deepest common folder."
- *Connection:* The folder structure is a tree. `/usr` is the common ancestor.

3. **The "CSS/DOM Selector" Problem (Frontend Interviews):**

- *Question:* "Given two HTML elements on a page, find the closest container `<div>` that wraps both of them."
- *Connection:* The DOM (Document Object Model) is a tree. This is an LCA problem on the DOM tree.

208. Implement Trie (Prefix Tree)

Here is how a Senior Staff Software Engineer (L6) would break down, visualize, and solve "Implement Trie (Prefix Tree)".

1. Problem Explanation

The goal is to build a data structure called a **Trie** (pronounced "try"). It is a special type of tree used to store strings, but unlike a standard list or hash map, it is optimized for prefix-based operations.

We need to implement three specific functions:

1. **`insert(word)`**: Add a word (e.g., "apple") into the trie.
2. **`search(word)`**: Return `true` if the exact word exists in the trie, `false` otherwise.
3. **`startsWith(prefix)`**: Return `true` if *any* word in the trie starts with this prefix.

The "Top-of-the-Band" Insight: Why not just use a Hash Set?

- A Hash Set can check if "apple" exists in O(1) (technically O(L) where L is string length).
- **However**, a Hash Set cannot efficiently answer `startsWith("app")`. You would have to scan every single word in the set.
- A Trie solves both efficiently.

---

2. Solution Explanation (with Visualizations)

Think of a Trie as a directory system on your computer.

- The **Root** is the main folder `/`.
- Each folder contains subfolders named after characters.
- To find "apple", you open folder `a` -> then `p` -> then `p` -> then `l` -> then `e`.
- Inside folder `e`, there is a special "flag" or "file" that says: **"A word ends here."**

The Node Structure Every node in our tree needs two things:

1. **Children**: A way to point to the next characters. (We often use a Hash Map or an Array of size 26).
2. **IsEndOfWord**: A boolean flag (True/False). If `True`, it means the path from root to this node forms a complete word.

Scenario 1: Inserting "apple" We start at the Root. We check if a path exists for each character. If not, we create it.

```
Step 1: Insert 'a'
ROOT
  |
  +---(a)

Step 2: Insert 'p'
ROOT
  |
  +---(a)
        |
        +---(p)

... Fast forward to 'e' ...

Final State after inserting "apple":
ROOT
  |
  +---(a)
        |
        +---(p)
              |
              +---(p)
                    |
                    +---(l)
                          |
                          +---(e) [isEndOfWord = True]
```

Scenario 2: Inserting "app" Now, let's insert "app". We follow the existing path for `a` -> `p` -> `p`. Since `p` already exists, we don't create new nodes. We just stop at the second `p` and mark it as an end of a word.

```
Visualizing "apple" AND "app" stored together:

ROOT
  |
  +---(a)
        |
        +---(p)
              |
              +---(p) [isEndOfWord = True]  <-- "app" ends here
                    |
                    +---(l)
                          |
                          +---(e) [isEndOfWord = True] <-- "apple" ends here
```

Scenario 3: Searching for "app" vs "apple" vs "apricot" A. Search("app")

1. Start at Root.
2. Go to `a`. Exists? Yes.
3. Go to `p`. Exists? Yes.
4. Go to `p`. Exists? Yes.
5. Finished string. Is the current node marked `isEndOfWord`? **Yes**. -> **Return True**.

B. Search("app") IF we only had inserted "apple" (Scenario 1)

1. Follow `a` -> `p` -> `p`.
2. Finished string. Is current node `isEndOfWord`? **No** (It was False in Scenario 1). -> **Return False**. (It is a prefix, but not a whole word).

C. StartsWith("app")

1. Follow `a` -> `p` -> `p`.
2. Did we make it to the end of the prefix without getting stuck? **Yes**.
3. Do we care about `isEndOfWord`? **No**. -> **Return True**.

---

3. Time and Space Complexity Analysis

Time Complexity Derivation

Let L be the length of the word we are inserting or searching.

```
Visualizing the Path Traversal:

Word:  "c  a  t"
        |  |  |
        v  v  v
Level: 1  2  3  ... up to L

Operation cost at each level is O(1) (Hash map lookup or Array index access).

Total Cost = 1 + 1 + 1 ... (L times)
           = O(L)
```

- **Insert:** O(L)
- **Search:** O(L)
- **StartsWith:** O(L)

Space Complexity Derivation

In the worst case, we have no common prefixes among words (e.g., "apple", "zoo", "ball").

```
Visualizing Worst Case Space:

Word 1: "a-b-c" (Length L) -> Creates L nodes
Word 2: "x-y-z" (Length L) -> Creates L nodes

Total Nodes = Sum of all characters in all words inserted.

If N = total number of words
If L = average length of a word

Total Space = O(N * L)
```

Top-of-the-Band Note on Space: If we used a Hash Map approach (like in the code below), the space is `O(N * L * Alphabet_Size)` roughly, because each node stores a map references. If we use an Array[26], the nodes are larger but the overhead is constant.

---

4. Solution Code

I will provide the solution using a **Hash Map (Dictionary)** for the children. This is generally preferred in production-level code over the fixed-size array (`size 26`) approach because it handles any character (Unicode, emojis, punctuation) without code changes, which is a trait of robust engineering.

Python Solution

```
class TrieNode:
    def __init__(self):
        # We use a dictionary for children.
        # Key: Character (e.g., 'a'), Value: TrieNode
        self.children = {}
        # Flag to mark if a word ends at this node
        self.is_end_of_word = False

class Trie:
    def __init__(self):
        """
        Initialize the Trie with a generic root node.
        """
        self.root = TrieNode()

    def insert(self, word: str) -> None:
        """
        Inserts a word into the trie.
        Iterate through char by char. If char path doesn't exist, create it.
        Mark the final node as end_of_word.
        """
        current_node = self.root

        for char in word:
            # If the character is not in the current node's children, add it
            if char not in current_node.children:
                current_node.children[char] = TrieNode()
            # Move the pointer to the child node
            current_node = current_node.children[char]

        # Mark the end of the word
        current_node.is_end_of_word = True

    def search(self, word: str) -> bool:
        """
        Returns True if the word is in the trie.
        """
        current_node = self.root

        for char in word:
            # If at any point the path breaks, the word doesn't exist
            if char not in current_node.children:
                return False
            current_node = current_node.children[char]

        # We found the path, but is it a marked word?
        # e.g., We have "apple", user searches "app". Path exists, but 'p' is not end.
        return current_node.is_end_of_word

    def startsWith(self, prefix: str) -> bool:
        """
        Returns True if there is any word in the trie that starts with the given prefix.
        """
        current_node = self.root

        for char in prefix:
            if char not in current_node.children:
                return False
            current_node = current_node.children[char]

        # If we successfully traversed the prefix, return True.
        # We don't care if it's an end of a word or not.
        return True
```

JavaScript Solution

```
class TrieNode {
    constructor() {
        // Map allows for flexible keys (unicode compliant)
        this.children = new Map();
        this.isEndOfWord = false;
    }
}

class Trie {
    constructor() {
        this.root = new TrieNode();
    }

    /** * @param {string} word
     * @return {void}
     */
    insert(word) {
        let currentNode = this.root;

        for (const char of word) {
            if (!currentNode.children.has(char)) {
                currentNode.children.set(char, new TrieNode());
            }
            currentNode = currentNode.children.get(char);
        }

        currentNode.isEndOfWord = true;
    }

    /** * @param {string} word
     * @return {boolean}
     */
    search(word) {
        let currentNode = this.root;

        for (const char of word) {
            if (!currentNode.children.has(char)) {
                return false;
            }
            currentNode = currentNode.children.get(char);
        }

        return currentNode.isEndOfWord;
    }

    /** * @param {string} prefix
     * @return {boolean}
     */
    startsWith(prefix) {
        let currentNode = this.root;

        for (const char of prefix) {
            if (!currentNode.children.has(char)) {
                return false;
            }
            currentNode = currentNode.children.get(char);
        }

        return true;
    }
}
```

Note 1: Terminology Clarification

Radix Tree (or Compact Prefix Tree) While the solution above is a standard Trie, in a "top-of-the-band" discussion, you might mention the **Radix Tree**.

- **Why it helps:** In a standard Trie, storing "Facebook" requires 8 nodes. If "Facebook" is the *only* word starting with 'F', we are wasting memory on nodes `a`->`c`->`e`…
- **How it works:** A Radix tree compresses the path. Instead of `F -> a -> c -> e`, a single node can store the string chunk `Face`. This optimizes space significantly for sparse datasets.

Note 2: Real World & Interview Variations

Companies like Google, Meta, and Bloomberg rarely ask "Implement a Trie" in isolation anymore. They wrap it in real-world scenarios:

1. **Typeahead / Autocomplete System (Google/Meta):**

- *Question:* "Design a search bar that suggests words as you type."
- *Application:* You implement the Trie `startsWith` logic, but each node also caches the "Top 3 most searched terms" in that branch to return results quickly.

2. **IP Routing Table (Networking roles):**

- *Question:* "How does a router decide where to send a packet based on an IP address?"
- *Application:* This is a "Longest Prefix Match" problem using Tries (specifically, a variant where the alphabet is 0 and 1).

3. **Spell Checker / Boggle Game Solver (Bloomberg):**

- *Question:* "Given a grid of characters, find all valid English words."
- *Application:* You load the dictionary into a Trie. As you traverse the grid (DFS), you check the Trie to prune invalid paths early (e.g., if you have path "xz", and the Trie says no words start with "xz", you stop searching that path immediately).

211. Design Add and Search Words Data Structure

Here is a breakdown of how a Senior (L5/L6) Engineer at Google would approach "Design Add and Search Words Data Structure," focusing on clarity, scalability, and robust handling of edge cases.

1. Problem Explanation

We need to build a data structure that acts like a specialized dictionary. It has two main operations:

1. **Add a Word:** Store a word (like "bad", "dad", "mad") efficiently.
2. **Search for a Word:** Check if a word exists in the dictionary.

The Twist (The Non-Trivial Part): The search function supports a wildcard character `.` (dot).

- A letter (e.g., 'a') must match the exact letter 'a'.
- A dot `.` can match **any** single letter.

Example: If we added "bad", "dad", and "mad":

- Search "pad" -> **False** (Not in our list)
- Search "bad" -> **True** (Exact match)
- Search ".ad" -> **True** (The `.` matches 'b', 'd', or 'm')
- Search "b.." -> **True** (Matches "bad" because `.` matches 'a' and `.` matches 'd')

---

2. Solution Explanation

An L5/L6 engineer would immediately reject a simple List or Hash Map (Object/Dict) for this problem.

- **Why not a Hash Map?** Hash maps are great for exact matches (), but they are terrible for pattern matching. To find ".ad", you would have to iterate through *every single key* in the map to see if it matches the pattern. That is too slow () for large dictionaries.

The L5 Choice: The Trie (Prefix Tree)

We will use a **Trie**. A Trie organizes words by their characters. Words with the same prefix share the same "path" in the tree. This saves space and makes prefix-based searching very fast.

Visualizing the Data Structure Imagine we add these words: **"BAD", "DAD", "MAD"**.

Step 1: The Trie Structure (ASCII)

```
       [ROOT NODE]
      /     |     \
     /      |      \
   'b'     'd'     'm'
    |       |       |
   'a'     'a'     'a'
    |       |       |
   'd'     'd'     'd'
    |       |       |
    * * * (end)   (end)   (end)
```

(Note: The `*` indicates a flag `isEnd = true`, marking that "bad" is a complete word, not just a prefix for something like "badge".)

The Algorithm **1. Adding a Word (`addWord`)** This is standard. We start at the root and traverse down, creating new nodes if the path doesn't exist yet.

2. Searching (`search`) - The Logic

- **Case A: Regular Letter (e.g., 'b')**
- Check if the current node has a child labeled 'b'.
- If yes, move to that child.
- If no, the word doesn't exist. Return False.
- **Case B: The Wildcard (e.g., '.')**
- This is the non-trivial part. Since `.` can be *any* letter, we cannot just pick one path.
- We must explore **ALL** existing children of the current node.
- If **any** of those paths leads to a successful match for the rest of the word, we return True.

Visualizing the Search with Wildcards **Scenario: Search for ".ad"**

```
QUERY: ".ad"

Step 1: Process character '.' at Root
   Root has 3 children: 'b', 'd', 'm'.
   Because it is a dot, we split into 3 parallel searches!

       [ROOT] -- (Current Node)
      /   |   \
     /    |    \
   (1)   (2)   (3)  <-- Try all 3 paths
   'b'   'd'   'm'
    ^     ^     ^
    |     |     |
   Try   Try   Try
  match match match
  '.ad' '.ad' '.ad'

Step 2: Let's follow Path (1) -> The 'b' branch.
   We consumed '.' (matched with 'b').
   Remaining query: "ad"
   Current Node: 'b' node.

       'b' -- (Current Node)
        |
       'a'  <-- Next char in query is 'a'. Does 'b' have child 'a'? YES.
        |

Step 3: Move down.
   We consumed 'a'.
   Remaining query: "d"
   Current Node: 'a' node.

       'a' -- (Current Node)
        |
       'd'  <-- Next char is 'd'. Does 'a' have child 'd'? YES.
        |

Step 4: Move down.
   We consumed 'd'.
   Remaining query: "" (Empty)
   Current Node: 'd' node.

   Check: Is this node marked as "End of Word"?
   YES (See the * in the first diagram).

   RESULT: TRUE.
   (Since one path returned True, we don't even need to finish checking paths 2 and 3).
```

---

3. Time and Space Complexity Analysis

An L5 engineer communicates complexity not just with "Big O" but by explaining the factors that drive it (like branching factor).

Time Complexity **1. `addWord(word)`**

- **Analysis:** We process each character of the word exactly once to insert it.
- **Complexity: O(M)**
- (where M is the length of the word being added)

2. `search(word)`

- **Best Case (No dots):** O(M). We just follow one path down.
- **Worst Case (Lots of dots):** This is where it gets heavy. If the search word is "...." (all dots), we have to visit every single node in the tree.

- **Derivation (Visual):**

```
Let M = Length of word
Let 26 = The alphabet size (Branching Factor)

Depth 0 (Root) -> 1 node
Depth 1 (.)    -> Checks up to 26 nodes
Depth 2 (..)   -> Checks up to 26 * 26 nodes
...
Depth M        -> Checks 26 * ... * 26 nodes

Total operations = 26^0 + 26^1 + ... + 26^M
```

- **Complexity: O(26^M)** (Exponential in the worst case with all dots)
- *Note:* In reality, it is O(N) where N is the total number of characters in the Trie, because we can't visit more nodes than we actually stored.

Space Complexity

- **Analysis:** We store every character of every word. In the worst case (no common prefixes), we store every character individually.
- **Complexity: O(Total Characters)** or **O(N * M)**
- (Where N is the number of words, M is average length).

---

4. Solution Code

Here is how we implement this cleanly.

Python Implementation

```python
class TrieNode:
    def __init__(self):
        # Using a dictionary for children mapping: char -> TrieNode
        self.children = {}
        # Flag to mark the end of a valid word
        self.is_end = False

class WordDictionary:

    def __init__(self):
        self.root = TrieNode()

    def addWord(self, word: str) -> None:
        """
        Standard Trie insertion. O(M) time.
        """
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end = True

    def search(self, word: str) -> bool:
        """
        Starts the recursive search from the root.
        """
        return self._dfs(0, self.root, word)

    # NON-TRIVIAL FUNCTION: DFS Helper
    # We use a helper because we need to pass the current 'node' state
    # and the current 'index' in the word during recursion.
    def _dfs(self, index: int, node: TrieNode, word: str) -> bool:
        # Base Case: If we have reached the end of the search string
        if index == len(word):
            # Return True only if this node marks the end of a stored word
            return node.is_end

        char = word[index]

        # Case 1: The Wildcard '.'
        if char == '.':
            # This is the branching part. We must check ALL children.
            for child_node in node.children.values():
                # If ANY child path returns True, the whole search is True
                if self._dfs(index + 1, child_node, word):
                    return True
            # If we tried all children and none worked
            return False

        # Case 2: A regular letter
        else:
            if char in node.children:
                # Continue searching down this specific path
                return self._dfs(index + 1, node.children[char], word)
            else:
                # Character mismatch
                return False

# Example Usage:
# obj = WordDictionary()
# obj.addWord("bad")
# obj.addWord("dad")
# param_2 = obj.search(".ad") # Returns True
```

JavaScript Implementation

```
class TrieNode {
    constructor() {
        this.children = {}; // Map char -> TrieNode
        this.isEnd = false;
    }
}

class WordDictionary {
    constructor() {
        this.root = new TrieNode();
    }

    /** * @param {string} word
     * @return {void}
     */
    addWord(word) {
        let node = this.root;
        for (const char of word) {
            if (!node.children[char]) {
                node.children[char] = new TrieNode();
            }
            node = node.children[char];
        }
        node.isEnd = true;
    }

    /** * @param {string} word
     * @return {boolean}
     */
    search(word) {
        return this.dfs(0, this.root, word);
    }

    // NON-TRIVIAL: Recursive Depth First Search helper
    // index: current position in the word we are searching
    // node: current node in the Trie we are standing on
    dfs(index, node, word) {
        // Base case: We processed the whole search word
        if (index === word.length) {
            return node.isEnd;
        }
        const char = word[index];

        if (char === '.') {
            // WILDCARD CASE:
            // Iterate over all keys (letters) in the current node's children
            for (const key in node.children) {
                const childNode = node.children[key];
                // Recursively call for the next character (index + 1)
                // at the child node.
                if (this.dfs(index + 1, childNode, word)) {
                    return true;
                }
            }
            // If checking all children failed
            return false;
        } else {
            // REGULAR LETTER CASE:
            // Check if the specific letter exists
            if (node.children[char]) {
                return this.dfs(index + 1, node.children[char], word);
            } else {
                return false;
            }
        }
    }
}
```

Note 1: Terminology & Techniques

1. Trie (Prefix Tree): A tree-like data structure used to store strings. Each node represents a character. It is incredibly efficient for "Autocomplete" or "Spell Checker" problems because you can quickly reject huge chunks of the dictionary if the prefix doesn't match.

2. DFS (Depth First Search) / Backtracking: In the solution above, when we hit a `.`, we used DFS. This technique "explores" one possible path (e.g., assuming `.` is 'a') as deep as possible. If that hits a dead end, it "backtracks" (returns) and tries the next possibility (e.g., assuming `.` is 'b').

Note 2: Real World & Interview Variations

How do Big Tech companies actually ask this? They rarely ask the "raw" Leetcode question. They wrap it in a scenario:

1. **Google (Typeahead / Autocomplete):**

- *Question:* "Design a search bar that can handle typos."
- *Connection:* The `.` acts like a "fuzzy" match. If the user types "Goggle", and you want to match "Google", you might search `Go.gle` or allow 1 character deviation using a similar DFS logic.

2. **Meta (Spell Checker):**

- *Question:* "Design a spell checker that supports single-character wildcards for crossword puzzle solvers."
- *Connection:* This is the exact implementation of Leetcode 211.

3. **Bloomberg (Ticker Symbol Matching):**

- *Question:* "We have a stream of partial stock tickers (e.g., 'AAP.'). Find all valid stocks that match."
- *Connection:* You store all valid tickers (AAPL, AMZN, etc.) in a Trie and use the wildcard logic to return all tickers fitting the pattern.

212. Word Search II

Here is a breakdown of "Word Search II" (Leetcode 212) from the perspective of a Senior Staff Software Engineer (L6).

An L6 engineer doesn't just look for *a* solution; they look for the *scalable* solution that handles bottlenecks efficiently. In this problem, the bottleneck is repetitive searching. We will move from a naive approach to an optimized Trie-based approach with pruning.

---

1. Problem Explanation

The Goal: You are given two things:

1. A 2D grid of characters (the "Board").
2. A list of valid words (the "Dictionary").

You need to find **all** the words from the dictionary that exist in the grid.

The Rules:

- You can start at any cell.
- You can move to adjacent cells (Up, Down, Left, Right).
- You cannot reuse the same cell more than once within a single word.

Visual Example:

Imagine this input:

Board:

```
+---+---+---+---+
| o | a | a | n |
+---+---+---+---+
| e | t | a | e |
+---+---+---+---+
| i | h | k | r |
+---+---+---+---+
| i | f | l | v |
+---+---+---+---+
```

Dictionary: `["oath", "pea", "eat", "rain"]`

The Task:

1. Look for "oath".

- Start at (0,0) 'o' -> (0,1) 'a' -> (1,1) 't' -> (1,2) 'h'. **FOUND**.

2. Look for "pea".

- Start at 'p'? No 'p' in grid. **NOT FOUND**.

3. Look for "eat".

- Start at (1,0) 'e' -> (0,1) 'a' -> (1,1) 't'. **FOUND**.

Output: `["eat", "oath"]`

---

2. Solution Explanation

The Naive Approach (Why it fails L5/L6 expectations) A junior engineer might iterate through every word in the list and run a grid search (DFS) for each one.

- Check "APPLE": Scan whole grid…
- Check "APPLY": Scan whole grid…
- Check "APRON": Scan whole grid…

The Problem: If you have 1,000 words starting with "A", you re-scan the "A"s on the board 1,000 times. This is redundant work.

The L5/L6 Approach: Inverted Thinking with a Trie Instead of holding a word and searching the grid, we **hold the grid** and explore all possible paths, checking if they form *any* valid word in our dictionary simultaneously.

To do this efficienty, we stuff all our dictionary words into a **Trie (Prefix Tree)**.

Step 1: Build the Trie We insert "oath", "pea", "eat", "rain" into a tree structure.

```
       [ROOT]
       /  |  \
     o    p    e    r
    /     |    |    |
   a      e    a    a
  /       |    |    |
 t        a* t* i
 |                  |
 h* n*

(* denotes the end of a word)
```

Step 2: Backtracking (DFS) on the Grid We iterate through every cell in the grid. We treat every cell as a potential starting point. We traverse the grid and the Trie **in sync**.

Walkthrough Visualization:

Let's start at grid position (0,0) which is `'o'`.

State 1: Start at (0,0)

- **Grid Cell:** 'o'
- **Trie Node:** Does Root have a child 'o'? Yes.
- **Action:** Move to Trie node `o`. Mark (0,0) as visited (#).

```
  Grid:              Trie Pointer:
  +---+---+          [ROOT]
  | # | a | ...         |
  +---+---+             o  <-- You are here
  | e | t |            /
  ...                 a
```

State 2: Move Right to (0,1)

- **Grid Cell:** 'a'
- **Trie Node:** Does current Trie node `o` have child 'a'? Yes.
- **Action:** Move to Trie node `a`. Mark (0,1) as visited.

```
  Grid:              Trie Pointer:
  +---+---+          [ROOT]
  | # | # | ...         |
  +---+---+             o
  | e | t |            /
  ...                 a  <-- You are here
                     /
                    t
```

State 3: Move Down to (1,1)

- **Grid Cell:** 't'

- **Trie Node:** Does current Trie node `a` have child '`t`'? Yes.
- **Action:** Move to Trie node `t`. Mark (1,1) as visited.

```
Grid:              Trie Pointer:
+---+---+             o
| # | # | ...        /
+---+---+           a
| e | # |          /
...               t  <-- You are here
                  |
                  h*
```

State 4: Move Right to (1,2)

- **Grid Cell:** 'h' (Assume grid has 'h' at 1,2)
- **Trie Node:** Does `t` have child `h`? Yes.
- **Check:** Is this a word end? **YES**. Found "oath".
- **CRITICAL OPTIMIZATION (Pruning):** Once "oath" is found, we remove "oath" from the output set to avoid duplicates.
- **Advanced Pruning:** An L6 engineer would remove the leaf node from the Trie so we never search for "oath" again.

---

3. Time and Space Complexity Analysis

This is how you justify your solution design in an interview without using complex math notation.

Time Complexity 1. Trie Construction:

- Cost: `O(Total characters in all words)`
- Why: We process every letter of every word once.

2. Grid Search (Backtracking): Let:

- `M` = Number of rows
- `N` = Number of columns
- `L` = Maximum length of a word

For every cell in the grid (`M * N` cells), we start a DFS.

Visualization of the DFS Branching:

```
      Start (Cell i,j)
           |
  +------+------+------+
  |      |      |      |
 Up    Down   Left   Right  (4 directions)
  |      |      |      |
 ...    ...    ...    ...   (3 directions each*)
```

```
*Note: We don't go back to the parent cell, so after the
 first step, the branching factor is at most 3.
```

In the worst case, we might traverse deep until the maximum word length `L`. So, the complexity is roughly: `O(M * N * 3^L)`

Reality Check (Pruning): Because we use a Trie, we stop immediately if the letter sequence doesn't exist in the dictionary. We don't actually run `3^L` blindly. The Trie bounds our search to only valid prefixes.

Space Complexity 1. Trie Storage: `O(Total characters in dictionary)` In the worst case, no words share prefixes.

2. Recursion Stack: `O(L)` The recursion goes as deep as the longest word in the dictionary.

Space Visualization:

```
Memory Usage
+---------------------------+
| Trie Structure            | <-- Major component
| (Nodes for 'a', 'b'...)   |
+---------------------------+
| Recursion Stack           | <-- Minor component (Max length L)
| [Call: search('h')]       |
| [Call: search('t')]       |
| [Call: search('a')]       |
| [Call: search('o')]       |
+---------------------------+
```

---

4. Solution Code

Here is the clean, robust implementation.

Python Solution

- **Note:** We use a specialized "TrieNode" class.
- **Optimization:** The `prune` step removes nodes from the Trie after a word is found. This progressively speeds up the algorithm as words are found.

```python
class TrieNode:
    def __init__(self):
        self.children = {}
        self.word = None  # Stores the word if this node is an end-of-word

class Solution:
    def findWords(self, board: list[list[str]], words: list[str]) -> list[str]:
```

```
# 1. Build the Trie
root = TrieNode()
for w in words:
    node = root
    for char in w:
        if char not in node.children:
            node.children[char] = TrieNode()
        node = node.children[char]
    node.word = w  # Mark end of word

ROWS, COLS = len(board), len(board[0])
res = []

# 2. Backtracking function
def backtrack(r, c, parent_node):
    char = board[r][c]
    curr_node = parent_node.children[char]

    # Check if we found a word
    if curr_node.word:
        res.append(curr_node.word)
        curr_node.word = None  # Avoid finding same word twice

    # Mark current cell as visited
    board[r][c] = '#'

    # Explore neighbors (Up, Down, Left, Right)
    for dr, dc in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
        nr, nc = r + dr, c + dc
        if (0 <= nr < ROWS and
            0 <= nc < COLS and
            board[nr][nc] in curr_node.children):
            backtrack(nr, nc, curr_node)

    # Restore cell (Backtracking step)
    board[r][c] = char

    # Optimization: Pruning the Trie
    # If the current node has no children, it's a dead leaf.
    # Remove it from the parent to speed up future searches.
    if not curr_node.children:
        del parent_node.children[char]

# 3. Iterate every cell on the board
for r in range(ROWS):
    for c in range(COLS):
```

```
                if board[r][c] in root.children:
                    backtrack(r, c, root)

        return res
```

JavaScript Solution

```
/**
 * @param {character[][]} board
 * @param {string[]} words
 * @return {string[]}
 */
var findWords = function(board, words) {
    // Trie Node Construction
    const root = {};

    // 1. Build Trie
    for (const w of words) {
        let node = root;
        for (const char of w) {
            if (!node[char]) node[char] = {};
            node = node[char];
        }
        node.word = w; // Store the actual word at the end node
    }

    const ROWS = board.length;
    const COLS = board[0].length;
    const res = [];

    // 2. DFS / Backtracking Helper
    const backtrack = (r, c, parentNode) => {
        const char = board[r][c];
        const currNode = parentNode[char];

        // Check if word match found
        if (currNode.word) {
            res.push(currNode.word);
            currNode.word = null; // Mark as found to prevent duplicates
        }

        // Mark visited
        board[r][c] = '#';

        // Directions: Right, Left, Down, Up
        const dirs = [[0, 1], [0, -1], [1, 0], [-1, 0]];
        for (const [dr, dc] of dirs) {
            const nr = r + dr;
            const nc = c + dc;

            // Check bounds and if the next char exists in Trie children
            if (nr >= 0 && nr < ROWS &&
                nc >= 0 && nc < COLS &&
                currNode[board[nr][nc]]) {
                backtrack(nr, nc, currNode);
            }
        }

        // Backtrack: Restore the cell
        board[r][c] = char;

        // Optimization: Leaf Pruning
        // If currNode has no children, remove it from parent
        // (This part is slightly complex in JS due to object keys,
        //  but in interviews, mentioning the concept is often enough)
    };

    // 3. Scan the grid
    for (let r = 0; r < ROWS; r++) {
        for (let c = 0; c < COLS; c++) {
            if (root[board[r][c]]) {
                backtrack(r, c, root);
            }
        }
    }

    return res;
};
```

Note 1: Terminology Breakdown

- **Trie (Prefix Tree):** A special tree used to store strings. Each edge represents a character. It is incredibly fast for prefix-based lookups (like autocomplete). In this problem, it allows us to check if the path we are currently walking on the grid is "promising" (i.e., actually leads to a word).
- **Backtracking:** A method of solving problems by trying to build a solution incrementally. If we hit a dead end (a letter not in the Trie), we "backtrack" (step back) and try a different path.
- **Pruning:** The act of cutting off parts of the search tree (or Trie) that we know won't lead to a solution. In our code, removing a word from the Trie

after we find it is “pruning” because we stop searching for it elsewhere.

Note 2: Real World / Interview Context

How do big tech companies ask this indirectly? They rarely ask “Solve Word Search II” verbatim anymore. They wrap it in a real-world scenario:

1. **Bloomberg (The Terminal Context):**

- “We have a grid of stock tickers and a stream of valid ticker symbols. Find all valid tickers hidden in the grid.”
- *Why:* Tests ability to handle financial dictionary data efficiently.

2. **Google (The Android Keyboard Context):**

- “Design the algorithm for ‘Swipe Typing’ (Gesture Typing) on a phone. Given a path traced by a finger on a keyboard grid, return the most likely valid words.”
- *Why:* This is literally Word Search II but with continuous paths and probability weights.

3. **Meta (The Game Context):**

- “You are building a Boggle solver backend. The user uploads a photo of a board. We have parsed it into a grid. Quickly return all valid words to show the user their score.”
- *Why:* Focuses on speed (latency) and the ability to prune searches early.

2458. Height of Binary Tree After Subtree Removal Queries

Here is how a Senior Staff Software Engineer (L6) at Google would break down and solve Leetcode 2458: “Height of Binary Tree After Subtree Removal Queries”.

---

1. Problem Explanation

Imagine you are maintaining a complex organizational hierarchy (a tree). You want to run simulations: **“If I temporarily fire this manager and their entire team (subtree), how deep does the reporting line go for the rest of the company?”**

You have to do this thousands of times. Since the queries are independent, the company structure “resets” after every simulation.

The Challenge:

- You have a **Binary Tree**.
- You have a list of **Queries**. Each query gives you a `node_id`.
- For every query, pretend that `node` and all its descendants are removed.
- Calculate the **Height** of the remaining tree (the length of the longest path from the root to any remaining leaf).

Why the "Naive" Solution Fails: If you actually copy the tree and delete nodes for every query, it takes too long.

- Deleting and measuring height takes **O(N)** time (where N is the number of nodes).
- If you have **M** queries, the total time is **O(N * M)**.
- With 100,000 nodes and 100,000 queries, that's 10 billion operations. This will crash or time out.

The L6 Engineer's Goal: We need to answer each query in **O(1)** (instant) time after some clever preparation.

---

2. Solution Explanation: The "Linearization" Technique

The secret to solving hard tree problems efficiently is often **turning the tree into an array**.

We can use a technique called **DFS Linearization** (or Euler Tour). When we traverse a tree using Depth First Search (DFS), we enter a node, visit all its children, and then exit.

- We can record the **Entry Time** (when we first see the node).
- We can record the **Exit Time** (when we finish the entire subtree).

Crucial Insight: In this "Linearized" array view, a **subtree is always a solid, contiguous block**. Removing a subtree is the same as cutting a slice out of an array.

Step-by-Step Visualization **1. The Tree** Let's look at this simple tree. The numbers are the Node Values.

```
        1  (Root, Depth 0)
       / \
      /   \
     3     2
    /     / \
   4     5   9
```

2. The DFS Traversal (The "Snake" Path) We walk through the tree: `1 -> 3 -> 4 -> (back) -> (back) -> 2 -> 5 -> (back) -> 9`.

Let's record the **Depth** of the node at each step of this walk.

Time	0	1	2	3	4	5
Node	1	3	4	2	5	9
Depth	0	1	2	1	2	2

Note: In a full Euler tour, we might record nodes twice, but for this specific problem, recording them once (Pre-order) and tracking the subtree size/exit time is sufficient.

Let's formalize the **Entry** and **Exit** indices for each node based on the table above:

- **Node 1**: Starts at index 0, Ends at index 5 (covers everything)
- **Node 3**: Starts at index 1, Ends at index 2 (covers 3, 4)
- **Node 2**: Starts at index 3, Ends at index 5 (covers 2, 5, 9)

3. The "Cut" Visualization Suppose the query is: **Remove Node 2**.

We know Node 2 corresponds to the range `[3, 5]` in our array. To find the max height of the *remaining* tree, we just need the maximum value in the array **excluding** indices 3, 4, and 5.

```
Indices:      [0,  1,  2,  3,  4,  5]
Depths:       [0,  1,  2,  1,  2,  2]
               ^   ^   ^   x   x   x
               |   |   |   |   |   |
               KEEP     REMOVE (Node 2 & kids)

We look at the LEFT part: [0, 1, 2] -> Max is 2
We look at the RIGHT part: []      -> Empty
Result: Max height is 2.
```

4. Precomputing for Speed To answer "What is the max value to the left/right of this range?" instantly, we build two helper arrays:

1. **PrefixMax:** `PrefixMax[i]` = Max depth from index `0` to `i`.
2. **SuffixMax:** `SuffixMax[i]` = Max depth from index `i` to the end.

Visualizing Prefix and Suffix:

```
Original Depths: [ 0, 1, 2, 1, 2, 2 ]

Prefix Max:      [ 0, 1, 2, 2, 2, 2 ]
(Scaning Left -> Right, carrying the max)

Suffix Max:      [ 2, 2, 2, 2, 2, 2 ]
(Scanning Right -> Left, carrying the max)
```

5. The Final Formula If we remove a node that spans indices `[L, R]`: The answer is `Max( PrefixMax[L - 1], SuffixMax[R + 1] )`.

3. Time and Space Complexity Analysis

Time Complexity Derivation

```
Phase 1: DFS Traversal
   (Visit every node once to build arrays)
   Nodes: [o] [o] [o] ... [o]
   Count:  N nodes
   Cost:   O(N)

Phase 2: Build Prefix/Suffix Arrays
   (Scan array left-to-right, then right-to-left)
   Array: [x] [x] [x] ... [x]
   Len:    N
   Cost:   O(N)

Phase 3: Answer Queries
   (Look up array indices)
   Query 1: Read Prefix[L-1], Read Suffix[R+1] -> O(1)
   ...
   Query M: Read Prefix[L-1], Read Suffix[R+1] -> O(1)
   Cost:   O(M)

TOTAL TIME: O(N + M)
This is Linear, which is optimal.
```

Space Complexity Derivation

```
We store:
1. Entry Times map:  [NodeID -> Index]  : Size N
2. Exit Times map:   [NodeID -> Index]  : Size N
3. Depths Array:     [Int, Int, ...]    : Size N
4. PrefixMax Array:  [Int, Int, ...]    : Size N
5. SuffixMax Array:  [Int, Int, ...]    : Size N

TOTAL SPACE: O(N)
```

---

4. Solution Code

Javascript Solution

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 * this.val = (val===undefined ? 0 : val)
 * this.left = (left===undefined ? null : left)
 * this.right = (right===undefined ? null : right)
 * }
 */

/**
 * @param {TreeNode} root
 * @param {number[]} queries
 * @return {number[]}
 */
var treeQueries = function(root, queries) {
    // Maps to store the range (start, end) for each node in the linearized array
    const nodeStart = new Map();
    const nodeEnd = new Map();
    const nodeDepth = new Map(); // Actually stores depth at the specific index

    // The "Linearized" tree (depths in DFS order)
    const depthsArr = [];

    let timer = 0;

    // ---------------------------------------------------------
    // Helper: DFS to flatten the tree and record depths
    // ---------------------------------------------------------
    function dfs(node, depth) {
        if (!node) return;

        // Record entry time (start of range)
        const start = timer;
        nodeStart.set(node.val, start);

        // Add current depth to our array
        depthsArr[timer] = depth;
        timer++;

        dfs(node.left, depth + 1);
        dfs(node.right, depth + 1);

        // Record exit time (end of range)
        // The 'end' is the last index used by this subtree
        nodeEnd.set(node.val, timer - 1);
    }

    // 1. Build the arrays
    dfs(root, 0);
    const n = depthsArr.length;

    // 2. Precompute Prefix Max (Left to Right)
    const prefixMax = new Array(n).fill(0);
    prefixMax[0] = depthsArr[0];
    for (let i = 1; i < n; i++) {
        prefixMax[i] = Math.max(prefixMax[i-1], depthsArr[i]);
    }

    // 3. Precompute Suffix Max (Right to Left)
    const suffixMax = new Array(n).fill(0);
    suffixMax[n-1] = depthsArr[n-1];
    for (let i = n - 2; i >= 0; i--) {
        suffixMax[i] = Math.max(suffixMax[i+1], depthsArr[i]);
    }

    // 4. Answer Queries
    const results = [];
    for (const q of queries) {
        const start = nodeStart.get(q);
        const end = nodeEnd.get(q);

        // Max height is max of everything to the LEFT of start
        // AND everything to the RIGHT of end.
        let maxH = 0;

        if (start > 0) {
            maxH = Math.max(maxH, prefixMax[start - 1]);
        }
        if (end < n - 1) {
            maxH = Math.max(maxH, suffixMax[end + 1]);
        }

        results.push(maxH);
    }

    return results;
};
```

Python Solution

```
# 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

class Solution:
    def treeQueries(self, root: Optional[TreeNode], queries: List[int]) -> List[int]:
        # Maps to store the linear range [start, end] for each node
        node_start = {}
        node_end = {}

        # The flattened array containing depths
        depths_arr = []

        # ----------------------------------------------------------
        # Helper: DFS to linearize the tree
        # ----------------------------------------------------------
        def dfs(node, depth):
            if not node:
                return

            # Record entry index
            start_idx = len(depths_arr)
            node_start[node.val] = start_idx
            depths_arr.append(depth)

            dfs(node.left, depth + 1)
            dfs(node.right, depth + 1)

            # Record exit index (current length - 1)
            node_end[node.val] = len(depths_arr) - 1

        # 1. Linearize tree
        dfs(root, 0)

        n = len(depths_arr)

        # 2. Build Prefix Max
        prefix_max = [0] * n
        prefix_max[0] = depths_arr[0]
        for i in range(1, n):
            prefix_max[i] = max(prefix_max[i-1], depths_arr[i])

        # 3. Build Suffix Max
        suffix_max = [0] * n
        suffix_max[n-1] = depths_arr[n-1]
        for i in range(n - 2, -1, -1):
            suffix_max[i] = max(suffix_max[i+1], depths_arr[i])
        # 4. Process Queries
        result = []
        for q in queries:
            l = node_start[q]
            r = node_end[q]

            current_max = 0

            # Check left of the removed range
            if l > 0:
                current_max = max(current_max, prefix_max[l-1])

            # Check right of the removed range
            if r < n - 1:
                current_max = max(current_max, suffix_max[r+1])

            result.append(current_max)

        return result
```

Note 1: Terminology & Techniques

Technique: Euler Tour / DFS Linearization This is a technique where you flatten a tree into an array based on the time you visit nodes. It is incredibly powerful because it turns "Subtree" problems (which are hard) into "Subarray" problems (which are easy).

- **Why it helps:** It groups a node and all its children into one contiguous block of indices.
- **Application:** Used heavily in "Lowest Common Ancestor" algorithms and advanced data structures like Segment Trees.

Note 2: Real World & Interview Variations

Google / Meta / Bloomberg Context: They rarely ask this exact Leetcode problem verbatim. Instead, they wrap it in a practical scenario:

1. **The "Simulated Failure" (Infrastructure):**

- *Question:* "You have a dependency graph of microservices (a tree). If Service X fails (and brings down all dependent services), what is the longest latency path remaining in the system?"
- *Connection:* Tree Height = Latency/Cost. Removing a subtree = Service Failure.

2. **The "DOM Rendering" (Frontend/Browser Engine):**

- *Question:* "We have a DOM tree. If we set a specific `<div>` to `display: none` (hiding it and its children), calculate the new maximum nesting depth of the visible page to determine if we need to enable a horizontal scrollbar."

3. **Organization Chart (Management):**

- *Question:* "If a VP leaves and takes their whole department, who is the now the lowest-ranking employee (furthest from CEO) in the remaining org?"

101. Symmetric Tree

Here is how a Senior Staff Engineer (L6) would break down **LeetCode 101: Symmetric Tree**. We focus on readability, edge cases, and distinct mental models.

1. Problem Explanation

At its core, this problem asks us to determine if a Binary Tree is a mirror image of itself. A common mistake is to think this just means "the left side looks like the right side." That is too vague.

The Strict Definition: A tree is symmetric if, and only if, for every node, the left subtree is a **mirror reflection** of the right subtree.

Imagine placing a vertical mirror right down the center of the root node.

Visualizing the Mirror Axis:

```
       [Mirror Line]
            |
      (Root Node)
            |
      /     |     \
   Left     |    Right
  Subtree   |   Subtree
            |
```

If you fold the tree along this dotted line, the nodes on the left must land perfectly on top of the nodes on the right with matching values.

Scenario A: The Perfect Symmetry (Returns True)

```
        1
      /   \
     2     2   <-- Layer 1: 2 == 2 (Match)
    / \   / \
  3   4 4   3 <-- Layer 2: Outer nodes (3,3) match.
                   Inner nodes (4,4) match.
```

Scenario B: The Value Mismatch (Returns False)

```
        1
      /   \
     2     2
    /       \
  3          4  <-- Error: 3 is not equal to 4.
```

Even though the structure is valid (both have children), the values do not mirror each other.

Scenario C: The Structural Mismatch (Returns False)

```
        1
      /   \
     2     2
      \     \
       3     3
```

Wait, why is this False? Let's look closer:

```
      Left Side (2)        Right Side (2)
      /   \                /   \
  (None)  3              (None)  3
```

- Left Side has a right child (3).
- Right Side has a right child (3).
- **But a mirror reflects!** The Right Side's *Right* child should match the Left Side's *Left* child.
- Here: Left.Left is `None`, but Right.Right is `3`. **Mismatch.**

---

2. Solution Explanation

The "Two-Pointer" Mental Model

An L6 engineer stops thinking about "one tree" and starts thinking about **"two trees traversing in opposite directions."**

We cannot solve this by looking at a single node. We need a helper function (let's call it `isMirror`) that takes **two** nodes (let's call them `t1` and `t2`) and compares them.

The Algorithm (Recursive Approach):

1. **Base Case 1 (Both Empty):** If `t1` is `None` AND `t2` is `None`, they are symmetric. Return `True`.

2. **Base Case 2 (One Empty):** If only one is `None` (structure mismatch), return `False`.
3. **Base Case 3 (Value Mismatch):** If `t1.val` does not equal `t2.val`, return `False`.
4. **The Recursive Step (The Mirror Move):**

- Compare `t1.left` with `t2.right` (The Outer Pair).
- Compare `t1.right` with `t2.left` (The Inner Pair).

Visualization of the "Opposite Direction" Traversal:

Let's trace the logic with ASCII diagrams.

Step 1: Compare Roots We start by calling the helper on the root's children.

```
      1
    /   \
  (t1)  (t2)
   2     2
```

- Result: Values match (2 == 2). Proceed deeper.

Step 2: Split into "Outer" and "Inner" Comparisons

This is the critical step. We launch two simultaneous checks.

Check A: The Outer Hug (t1.left vs t2.right)

```
        1
      /   \
     2     2
    /       \
  (t1)      (t2)
   3         3
```

- We compare the far-left node (3) with the far-right node (3).
- Result: Match.

Check B: The Inner Hug (t1.right vs t2.left)

```
        1
      /   \
     2     2
      \   /
     (t1)(t2)
       4   4
```

- We compare the inner-left node (4) with the inner-right node (4).
- Result: Match.

Step 3: What happens if we have a Structural Flaw?

Consider this broken tree:

```
      1
    /   \
   2     2
  /     /
 3     3
```

1. **Roots (2, 2)** match.
2. **Outer Check:** Compare `Left(2).left` (which is 3) vs `Right(2).right` (which is `None`).
3. **Mismatch!** One is 3, one is None. Return False immediately.

---

3. Time and Space Complexity Analysis

Time Complexity: O(N)

We must visit every node in the tree exactly once (or strictly speaking, every pair of nodes once) to verify symmetry.

- `N` is the number of nodes.
- We perform constant time `O(1)` work (comparisons) at each node.
- Total time = `O(N)`.

Visualization of Time Cost:

```
  Tree Size (N)   |   Operations
------------------|----------------------
      1           |   1 check
      3           |   3 checks
      7           |   7 checks
      ...         |   ...
      N           |   N checks (Linear)
```

Space Complexity: O(H) (Where H is Height of Tree)

This is where the recursion stack lives. In the worst case, the recursion goes as deep as the tree height.

Scenario 1: Balanced Tree (Best Case for Space)

```
      1          <-- Stack Frame 1
    /   \
   2     2       <-- Stack Frame 2
  / \   / \
 3   4 4   3     <-- Stack Frame 3
```

- Height `H` is roughly `log2(N)`.
- Space = `O(log N)`.

Scenario 2: Linear / Skewed Tree (Worst Case for Space) *Note: A skewed tree isn't symmetric unless it's empty or just a root, but strictly speaking,* *if we passed a non-symmetric linear tree, the algorithm would fail fast. However, if we consider the* Stack Capacity* required to traverse down to the failure point:*

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

- To check the bottom nodes (4 vs 4), we must hold 2, 3, and 4 in the stack.
- Height `H` is `N/2`.
- Space = `O(N)` in the worst case scenarios (like a long line of matching nodes).

---

4. Solution Code

We use the recursive approach because it directly maps to the "Definition" of symmetry (Top-Down Mirroring).

Python Solution

```python
# 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

class Solution:
    def isSymmetric(self, root: Optional[TreeNode]) -> bool:
        # Edge Case: An empty tree is technically symmetric
        if not root:
            return True

        # Start the recursive helper comparing left and right children
        return self.check_mirror(root.left, root.right)

    # Helper function to compare two sub-trees
    # This is the "isMirror" logic described in the solution
    def check_mirror(self, t1: Optional[TreeNode], t2: Optional[TreeNode]) -> bool:
        # Case 1: Both nodes are None (End of branch) -> Match!
        if not t1 and not t2:
            return True
        # Case 2: One is None, but not the other -> Mismatch!
        if not t1 or not t2:
            return False

        # Case 3: Values differ -> Mismatch!
        if t1.val != t2.val:
            return False

        # Case 4: Values match, so we must dig deeper.
        # We need BOTH strict conditions to be true:
        # 1. Outer Hug: t1.left matches t2.right
        # 2. Inner Hug: t1.right matches t2.left
        return (self.check_mirror(t1.left, t2.right) and
                self.check_mirror(t1.right, t2.left))
```

JavaScript Solution

```javascript
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 * this.val = (val===undefined ? 0 : val)
 * this.left = (left===undefined ? null : left)
 * this.right = (right===undefined ? null : right)
 * }
 */

/**
 * @param {TreeNode} root
 * @return {boolean}
 */
var isSymmetric = function(root) {
    if (!root) return true;

    return checkMirror(root.left, root.right);
};

/**
 * Helper function to validate if two trees are mirror images
 * @param {TreeNode} t1 - The left node being compared
 * @param {TreeNode} t2 - The right node being compared
 */
function checkMirror(t1, t2) {
    // 1. Both empty -> Valid Symmetry
    if (t1 === null && t2 === null) {
        return true;
    }

    // 2. One empty, one not -> Broken Symmetry
    if (t1 === null || t2 === null) {
        return false;
    }

    // 3. Values do not match -> Broken Symmetry
    if (t1.val !== t2.val) {
        return false;
    }

    // 4. Recursive Step:
    // Compare Outer Pair (Left vs Right) AND Inner Pair (Right vs Left)
    return checkMirror(t1.left, t2.right) &&
           checkMirror(t1.right, t2.left);
}
```

Note 1: Terms and Techniques

"Short-Circuit Evaluation" In the return statement `checkMirror(Outer) && checkMirror(Inner)`, we utilize short-circuiting. If the "Outer" check returns `False`, the code immediately stops and returns `False` without even running the "Inner" check. This saves processing time in non-symmetric trees.

"Depth-First Search (DFS) on Pairs" The algorithm we used is a variation of DFS. Standard DFS traverses one tree. Here, we traverse **two trees in lockstep**. This is a common pattern in "Tree Comparison" problems (like checking if two trees are identical or subtrees).

Note 2: Real World & Interview Context

How Google/Meta/Bloomberg ask this indirectly:

1. **The "DOM Tree Diff" (Meta/Front-end):**

- *Question:* "Given two DOM trees (HTML structures), write a function to determine if the second one is a reversed/mirrored layout of the first."
- *Application:* This tests your ability to manipulate UI heirarchies.

2. **The "Palindrome Graph" (Google):**

- *Question:* "Given a graph structure (or N-ary tree), determine if the connections form a palindrome relative to the center."
- *Application:* This is the same logic as Symmetric Tree, but scaled up to `N` children instead of just Left/Right.

3. **Financial Modeling (Bloomberg):**

- *Question:* "We have a decision tree for a trading algorithm. We want to ensure the 'Buy' logic on one side is the exact inverse of the 'Sell' logic on the other side."
- *Application:* This is a semantic symmetry check. Instead of checking `val == val`, you might check `val == -val` or `Action == Inverse(Action)`.

222. Count Complete Tree Nodes

An L5/L6 (Senior/Staff) engineer approaches this problem not just by writing code that works, but by exploiting the **specific structural properties** of the input to beat the standard linear time complexity.

Here is the breakdown of the "Complete Binary Tree" problem.

1. Problem Explanation

The Goal: Count the total number of nodes in a "Complete Binary Tree".

What is a Complete Binary Tree? In a Complete Binary Tree, every level, except possibly the last, is completely filled. All nodes in the last level are as far left as possible.

Visualizing the Difference:

Case A: A Complete Binary Tree (Valid) All levels are full. The last level has nodes packed to the left.

```
      1           <-- Level 0 (Full)
    /   \
   2     3        <-- Level 1 (Full)
  / \   /
 4   5 6          <-- Level 2 (Filled left-to-right)
```

Total Nodes: 6

Case B: NOT a Complete Binary Tree Notice node 5 is missing, but node 6 exists. This creates a "gap" in the left-to-right filling order.

```
      1
    /   \
   2     3
  /     /
 4     6          <-- Missing child for 2 (Right), but 3 has a child.
```

Case C: A Full Binary Tree (A specific type of Complete Tree) Every single level is 100% full.

```
      1
    /   \
      2     3
     / \   / \
    4   5 6   7
```

Total Nodes: 7

The Mathematical Shortcut: If a tree is a **Full Binary Tree** with height `H` (where root is height 1), the total nodes are exactly: `Nodes = 2^H - 1`

- Example Case C (Height 3): `2^3 - 1` = `8 - 1` = `7`.

---

2. Solution Explanation: The "Divide and Conquer" Optimization

The Junior (L3) Approach: Traverse every node (DFS or BFS) and count them one by one.

- **Time Complexity:** O(N) - Linear.
- **Why an L6 rejects this:** If the tree has 1 billion nodes, you visit 1 billion nodes. We can do much faster by skipping huge chunks of the tree using the "Complete" property.

The Senior (L5/L6) Approach: We combine the structure of the tree with the **Full Tree Shortcut**.

Core Logic:

1. Calculate the height of the entire tree. Let's call this `TotalHeight`.
2. Check the height of the **Right Subtree**.

- **Scenario A:** If the Right Subtree's height is exactly `TotalHeight - 1`, it means the **Left Subtree is a Full Binary Tree**.
- **Scenario B:** If the Right Subtree's height is less than that, it means the **Right Subtree is a Full Binary Tree** (but one level shorter).

Let's visualize this decision process.

Step-by-Step Execution with ASCII Definitions:

- `H`: Height of the current tree (counting levels).
- `Left`: Left child.
- `Right`: Right child.

Scenario A: Left Subtree is Full

Imagine this tree. `H = 4`.

```
             (Root)
            /      \
      (Left)        (Right)
      /    \        /     \
    (..)  (..)    (..)   (..)   <-- Right subtree reaches this level
    /  \  /  \    /
   x    x x   x  x             <-- The last level extends into the Right side
```

Analysis:

1. We check the height of `Right`. It reaches the bottom level.
2. This guarantees that `Left` is fully packed.
3. **Action:**

- Use math for `Left`: `2^(H-1) - 1` nodes.
- Add 1 for `Root`.
- **Total so far:** `2^(H-1)`.
- **Recurse:** We still need to count the specific nodes in `Right`.
- **New Problem:** Count nodes in `Right`.

Scenario B: Right Subtree is Full (but smaller)

Imagine this tree. `H = 4`.

```
             (Root)
            /      \
      (Left)        (Right)
      /    \        /     \
    (..)  (..)    (..)   (..)   <-- Right subtree stops here
    /  \  /
   x    x x                     <-- The last level ends inside Left side
```

Analysis:

1. We check the height of `Right`. It does **not** reach the bottom level.
2. This means the "gap" is somewhere in the `Left` subtree.
3. However, this guarantees that `Right` is a **Full Binary Tree** of height `H-2`.
4. **Action:**

- Use math for `Right`: `2^(H-2) - 1` nodes.
- Add 1 for `Root`.
- **Total so far:** `2^(H-2)`.
- **Recurse:** We still need to count the nodes in `Left`.

---

3. Time and Space Complexity Analysis

This is the most critical part for a Google interview.

Time Complexity Derivation

We traverse down the height of the tree (`d` or `log N`). At each step, we calculate the height of a subtree.

```
Visualizing the Work:
```

```
Step 1: Root
   | Calculate Height of Right Child (Cost: O(log N))
   | Decide to go Left or Right
   V
Step 2: Next Level Node
   | Calculate Height of Child (Cost: O(log N - 1))
   | Decide...
   V
...
   V
Step d: Leaf Node
```

Calculation:

- The tree depth `d` is roughly `log N`.
- At every single level, we run a "height check".
- The "height check" takes `O(d)` time (just zooming down the left edge).
- We do this `d` times.

Total Time: `O(d * d)` which equals **O((log N)^2)**.

Comparison:

- L3 Approach (Count all): `O(N)`
- L6 Approach (Divide & Conquer): `O((log N)^2)`

Example: If N = 1,000,000 (1 Million)

- `O(N)` = 1,000,000 operations.
- `O((log N)^2)` = `(20)^2` = 400 operations.
- **The L6 solution is 2,500x faster.**

Space Complexity:

- We use recursion. The recursion stack goes as deep as the height of the tree.
- **Space:** `O(d)` or `O(log N)`.

---

4. Solution Code

Here is the clean, production-ready code.

Python Solution

```python
# 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

class Solution:
    def countNodes(self, root: TreeNode) -> int:
        """
        Calculates the number of nodes in a complete binary tree.
        Time Complexity: O((log N)^2)
        Space Complexity: O(log N)
        """
        if not root:
            return 0

        # 1. Calculate the depth (height) of the leftmost path
        # This tells us the maximum height of the tree.
        # Note: We can cache this, but calculating it is O(log N)
        left_depth = self.get_depth(root.left)
        right_depth = self.get_depth(root.right)

        if left_depth == right_depth:
            # SCENARIO A: The Left Subtree is a Perfect/Full Binary Tree.
            #
            #       Root
            #      /    \
            #   (Full)  (Recurse)
            #
            # 1. Nodes in Left (Full) = 2^left_depth - 1
            # 2. Add Root (+1)
            #    Total known = (2^left_depth - 1) + 1 = 2^left_depth
            # 3. Add result of recursion on Right

            # Note: 1 << left_depth is the bitwise way to write 2^left_depth
            return (1 << left_depth) + self.countNodes(root.right)

        else:
            # SCENARIO B: The Right Subtree is a Perfect/Full Binary Tree
            # (but one level shallower).
            #
            #       Root
            #      /    \
            #  (Recurse) (Full, but smaller)
            #
            # 1. Nodes in Right (Full) = 2^right_depth - 1
            # 2. Add Root (+1)
            #    Total known = 2^right_depth
            # 3. Add result of recursion on Left
            return (1 << right_depth) + self.countNodes(root.left)

    def get_depth(self, node: TreeNode) -> int:
        """
        Helper to find depth of the leftmost path.
        Since it's a complete tree, leftmost path = height.
        """
        depth = 0
        while node:
            node = node.left
            depth += 1
        return depth
```

Javascript Solution

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 * this.val = (val===undefined ? 0 : val)
 * this.left = (left===undefined ? null : left)
 * this.right = (right===undefined ? null : right)
 * }
 */

/**
 * @param {TreeNode} root
 * @return {number}
 */
var countNodes = function(root) {
    if (!root) return 0;

    const leftHeight = getDepth(root.left);
    const rightHeight = getDepth(root.right);

    if (leftHeight === rightHeight) {
        // Left subtree is a full binary tree
        // The formula for nodes in full tree is 2^h - 1.
        // We add the root (+1).
        // Total = (2^h - 1) + 1 + countNodes(right)
        //       = 2^h + countNodes(right)

        // Math.pow(2, leftHeight) is the same as (1 << leftHeight)
        return Math.pow(2, leftHeight) + countNodes(root.right);
    } else {
        // Right subtree is full, but 1 level shorter
        // Total = (2^rightHeight) + countNodes(left)
        return Math.pow(2, rightHeight) + countNodes(root.left);
    }
};

// Helper function to get height of the tree (down the left side)
function getDepth(node) {
    let depth = 0;
    while (node) {
        node = node.left;
        depth++;
    }
    return depth;
}
```

Note 1: New Terms & Techniques

Bitwise Shift (`1 << h`): In the code, you saw `1 << left_depth`.

- In binary, `1` is `0001`.
- `1 << 3` shifts the bit 3 spots: `1000`, which is 8.
- This is a CPU-efficient way to calculate `2^h`.
- Google engineers prefer this over `Math.pow()` as it signals an understanding of low-level binary operations.

Binary Search on Tree Structure: Standard Binary Search works on arrays (indices). Here, we are performing a Binary Search logic on the **structure** of the tree. We check the middle (by comparing left vs right height) and discard half of the problem (by calculating it mathematically instead of traversing it).

Note 2: Real World & Interview Context

This question is a favorite at **Google, Meta (Facebook), and Bloomberg**. They rarely ask it as "Here is a tree, count it." They disguise it.

1. The Distributed Database ID Problem (Google)

> "You have a distributed system where IDs are generated in a specific tree-like hierarchy to ensure uniqueness. The IDs fill up 'shards' from left to right. Given the root shard, how do you quickly calculate the total number of active IDs without querying every single shard?" *(This is exactly Count Complete Tree Nodes).*

2. The Heap Storage Question (Bloomberg)

> "We are implementing a custom binary heap for a high-frequency trading engine. We store the heap in a pointer-based tree (not an

array) for dynamic memory reasons. We need a function `getSize()` that runs faster than O(N) to ensure our metrics dashboard doesn't slow down the trading loop."

3. **The Implicit Indexing Problem (Meta)**

   "Given a node in a Complete Binary Tree, how do you find its index in an array representation? Conversely, if you know the max index, how do you find the specific node?" *(Understanding the height/splitting logic of this solution is the prerequisite to solving this).*

543. Diameter of Binary Tree

Here is a breakdown of "Diameter of Binary Tree" from the perspective of a Senior (L5/L6) Engineer. At this level, we care deeply not just about getting the right answer, but about code readability, maintainability, and understanding *why* the algorithm works the way it does.

---

1. Problem Explanation

The Core Concept The "diameter" of a tree is the length of the longest path between any two nodes.

- **Crucial detail:** The length is measured by the number of **edges** (the lines connecting nodes), not the number of nodes.
- **The Trap:** The longest path does **not** always have to pass through the root node. It could be entirely contained within the left subtree or the right subtree.

Let's visualize this.

Example 1: The Standard Case (Passes through Root) The longest path goes from the bottom left, up to the root, and down to the bottom right.

```
      1  (Root)
     / \
    2   3
   /     \
  4       5
```

- Path: 4 -> 2 -> 1 -> 3 -> 5
- Count the edges (lines): 4 edges.
- Diameter: 4

Example 2: The "Hidden" Case (Does NOT pass through Root) The root is just a spectator here. The "action" is happening deep in the left subtree.

```
        1 (Root)
       /
      2
     / \
    3   4
   /     \
  5       6
 /         \
7           8
```

- Look at Node 2.
- Left arm goes down: 2 -> 3 -> 5 -> 7 (Length 3)
- Right arm goes down: 2 -> 4 -> 6 -> 8 (Length 3)
- Path connects 7 to 8: 7->5->3->2->4->6->8.
- Total edges: 6.
- Note: The path from Root (1) to Node 7 is only length 4. The diameter (6) is larger!

---

2. Solution Explanation: The "Bottom-Up" Approach

As an L6 engineer, I look at this and think: "I cannot know the diameter passing through a specific node unless I know the height of its left and right children."

This dependency implies a **Post-Order Traversal** (Bottom-Up).

1. Go deep to the leaves.
2. As we return from the recursion, pass information **up** to the parent.

The "Information Flow" Strategy Every node in the tree has two responsibilities:

1. **Report to Parent:** "Here is my height (longest chain down)."
2. **Update Global Record:** "Here is the longest path that pivots *around me* (Left Height + Right Height)."

Visualizing the Algorithm Flow

Let's trace this small tree:

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

Step A: Hit the bottom (Nodes 4, 5, 3) Nodes 4, 5, and 3 are leaves. They have no children.

- Left child: 0 height.

- Right child: 0 height.
- **Calculate Diameter:** $0 + 0 = 0$. (Update Max Diameter if 0 is > current max).
- **Report to Parent:** max(0, 0) + 1 = **1**.

```
Visual State after leaves process:

      1
     / \
    2   3 (Returns 1)
   / \
  4   5
 (Returns 1)
 (Returns 1)
```

Step B: Process Node 2 Node 2 receives reports from its children (4 and 5).

- From Left (4): 1
- From Right (5): 1
- **Calculate Diameter through Node 2:** Left(1) + Right(1) = **2**.
- *System Check:* Is 2 bigger than our current max? Yes. Set Max Diameter = 2.
- **Report to Parent (1):** max(Left, Right) + 1 -> max(1, 1) + 1 = **2**.

```
Visual State after Node 2 processes:

      1
     / \
    2   3 (Returns 1)
   (Returns 2)
```

Step C: Process Root (Node 1) Node 1 receives reports.

- From Left (Node 2): 2
- From Right (Node 3): 1
- **Calculate Diameter through Root:** Left(2) + Right(1) = **3**.
- *System Check:* Is 3 bigger than current max (2)? Yes. Set Max Diameter = 3.
- **Report to Parent:** (End of function, but would return 3).

Final Result: 3.

---

3. Time and Space Complexity Analysis

Time Complexity: Linear Time

We are visiting every node exactly once.

```
      (Root)  <-- 1 visit
      /    \
     O      O <-- 1 visit each
    / \    / \
   O   O  O   O <-- 1 visit each

Total Operations = Constant work * Number of Nodes (N)
Time Complexity = O(N)
```

Space Complexity: Stack Height

We use recursion. Recursion uses the "Call Stack" (memory). The space used depends on how deep the tree goes.

Scenario A: Balanced Tree The tree is nice and full. The depth is roughly log(N).

```
      O
     / \
    O   O
   / \ / \  <-- Stack depth is shallow
  O  O O  O
```

Space = O(log N)

Scenario B: Skewed Tree (Worst Case) The tree is essentially a linked list. The recursion goes all the way to the bottom before returning.

```
    O
     \
      O
       \
        O
         \
          O <-- Stack depth = N
```

Space = O(N)

---

4. Solution Code

Here is how we write this in a production environment. Note the separation of concerns: the helper function calculates height, while the main function manages the state (the max diameter).

Python Solution

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def diameterOfBinaryTree(self, root: TreeNode) -> int:
        # We use a non-local variable (or self variable) to track the
        # global maximum diameter found so far.
        self.max_diameter = 0

        # Start the Depth First Search
        self.calculate_height(root)

        return self.max_diameter

    # Helper function: The "Worker"
    # Responsibility: Return height of the current subtree
    # Side Effect: Update self.max_diameter
    def calculate_height(self, node):
        # Base Case: If node is None, it contributes 0 to the path length
        if not node:
            return 0

        # Recursive Step: Go deep first (Post-Order Traversal)
        left_height = self.calculate_height(node.left)
        right_height = self.calculate_height(node.right)

        # CRITICAL STEP:
        # The longest path PASSING through this specific node is
        # the sum of the heights of its left and right children.
        path_through_node = left_height + right_height

        # Update the global maximum if this path is the new winner
        self.max_diameter = max(self.max_diameter, path_through_node)

        # Return Step:
        # Return the height of this node to its parent.
        # Height = 1 (for itself) + the longer of the two children paths.
        return 1 + max(left_height, right_height)
```

JavaScript Solution

```javascript
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 * this.val = (val===undefined ? 0 : val)
 * this.left = (left===undefined ? null : left)
 * this.right = (right===undefined ? null : right)
 * }
 */

/**
 * @param {TreeNode} root
 * @return {number}
 */
var diameterOfBinaryTree = function(root) {
    // Variable to store the maximum diameter found.
    // This acts as our "Global State" for the recursion.
    let maxDiameter = 0;

    // Helper function to perform DFS
    function calculateHeight(node) {
        // Base case: If we hit null, height is 0
        if (node === null) {
            return 0;
        }

        // Recursively calculate height of left and right subtrees
        const leftHeight = calculateHeight(node.left);
        const rightHeight = calculateHeight(node.right);

        // LOGIC: The diameter passing through *this* node is L + R.
        // We update the global max if this local diameter is larger.
        maxDiameter = Math.max(maxDiameter, leftHeight + rightHeight);

        // RETURN: We must return the vertical height of this node to its parent.
        // Height = 1 (this node) + max(left, right)
        return 1 + Math.max(leftHeight, rightHeight);
    }

    // Kick off the recursion
    calculateHeight(root);

    return maxDiameter;
};
```

Note 1: Terminology & Techniques

- **DFS (Depth First Search):** An algorithm that explores as far as possible along each branch before backtracking. We used this here.
- **Post-Order Traversal:** A specific type of DFS where we visit the children first (Left, Right) and process the current node last. This is crucial here because a parent cannot calculate its diameter until it knows the height of its children.
- **Global State (Closure):** In the solution, `maxDiameter` is outside the recursive function. This allows different recursive calls to communicate with each other by updating a shared variable.

Note 2: Real World & Interview Variations

Big tech companies rarely ask "Find the diameter" directly. They wrap it in a practical scenario:

1. **Google (File Systems):**

- *Question:* "Given a directory structure, find the longest path between any two files to determine the maximum network hop count needed for a peer-to-peer transfer."
- *Mapping:* Directories are nodes. The problem is identical to Diameter of Binary Tree (or N-ary tree).

2. **Meta (Social Graphs):**

- *Question:* "In a strictly hierarchical corporate structure (a tree), what is the longest chain of command between two employees?"
- *Mapping:* Employees are nodes. The "chain of command" distance is the diameter.

3. **Bloomberg (Network Latency):**

- *Question:* "You have a network of servers arranged in a tree topology. Signals travel from server to server. Find the two servers that have the highest latency (longest delay) between them."
- *Mapping:* This is a weighted version of the diameter problem (where edges might have weights representing milliseconds), but the logic of `Left + Right` remains the core principle.

951. Flip Equivalent Binary Trees

Here is how a Senior Staff Engineer (L6) at Google would break down, analyze, and solve LeetCode 951: **Flip Equivalent Binary Trees**.

---

1. Problem Explanation

The Core Question: We are given two binary trees, let's call them `root1` and `root2`. The problem asks: Can we make `root1` look *exactly* like `root2` by flipping the left and right children of any number of nodes?

What is a "Flip"? A flip operation on a node means swapping its left child with its right child. We can do this for any node, or no nodes, or all nodes. We choose node by node.

Visualizing the Goal: Imagine a mobile (the hanging art structure) where each joint is a node. If you twist a joint, the left and right parts swap places, but the structure remains the same. The question is: Are these two mobiles actually the same structure, just twisted differently?

---

2. Solution Explanation

The Approach: Recursive Structure Matching

As an L6 engineer, I look for recursive patterns immediately. Trees are recursive data structures. If the whole trees are equivalent, their sub-parts must also be equivalent.

The Decision Logic (The "Algorithm"):

For any two nodes `node1` from Tree A and `node2` from Tree B to be "flip equivalent", three checks must pass:

1. **Null Check:** If both are `null` (empty), they are equivalent. If one is `null` but the other isn't, they are NOT equivalent.
2. **Value Check:** The value held at `node1` must equal the value at `node2`. If the roots are different, the trees can never be the same.
3. **Structure Check (The Recursion):**

- **Case A (No Flip):** The left child of `node1` matches the left child of `node2` **AND** the right child of `node1` matches the right child of `node2`.
- **OR**
- **Case B (Flip):** The left child of `node1` matches the *right* child of `node2` **AND** the right child of `node1` matches the *left* child of `node2`.

If either Case A or Case B is true, the trees are equivalent.

ASCII Walkthrough

Let's trace this with a simple example.

Scenario: We want to check if Tree 1 and Tree 2 are equivalent.

```
     Tree 1                  Tree 2
       (1)                     (1)
      /   \                   /   \
    (2)   (3)              (3)   (2)
    /                              /
  (4)                            (4)
```

Step 1: Compare Roots

- `node1` is (1). `node2` is (1).
- Values match. Proceed.

Step 2: Check Children (Recursive Step) We check two possibilities:

- **Possibility A: Direct Match (No Flip)**
- Compare `Left(1)` with `Left(1)` -> Compare (2) with (3).
- Values differ (2 != 3). **Possibility A Fails immediately.**
- **Possibility B: Crossed Match (Flip)**
- Compare `Left(1)` with `Right(1)` -> Compare (2) with (2). **Match!**
- Compare `Right(1)` with `Left(1)` -> Compare (3) with (3). **Match!**

Now we dive deeper into the successful branches.

Step 3: Analyzing the Sub-trees (from the Flip)

Sub-comparison 1: Tree 1's (2) vs Tree 2's (2).

```
Tree 1 Node (2)      Tree 2 Node (2)
    (2)                  (2)
   /                    /
 (4)                  (4)
```

- Values match.
- **Direct Match Check:**
- Left vs Left: (4) vs (4). Match!
- Right vs Right: `null` vs `null`. Match!
- Since Direct Match worked, these subtrees are equivalent.

Sub-comparison 2: Tree 1's (3) vs Tree 2's (3).

```
Tree 1 Node (3)      Tree 2 Node (3)
    (3)                  (3)
```

- Both are leaf nodes.
- Left vs Left (`null` vs `null`) -> Match.
- Right vs Right (`null` vs `null`) -> Match.
- Equivalent.

Conclusion: Since the root values matched, and the children matched via a "Flip", the result is **TRUE**.

3. Time and Space Complexity Analysis

Time Complexity: O(N)

We process each node in the trees at most a constant number of times.

Optimization Note: A naive solution might check *both* flip and no-flip recursively, leading to 4 calls. However, because node values are unique (a standard constraint for this specific problem context), we only proceed down a branch if the values match. This effectively prunes the search immediately if the root values don't align.

```
Visualizing Complexity:

      (Node)  <-- 1. Check Value O(1)
      /    \
   (L)      (R)
                  2. If values match, we recurse.
                     We visit every node exactly once
                     assuming values are unique constraints.

    Total Steps = Number of Nodes (N) * Constant Work
                = O(N)
```

Space Complexity: O(H) to O(N)

The space is determined by the recursion stack (how deep the tree goes).

- **H** is the height of the tree.
- In a balanced tree, `H = log N`.
- In the worst case (a skewed line), `H = N`.

```
Stack Visual (Worst Case):

|  Call: Node 4  |
|  Call: Node 3  |
|  Call: Node 2  |
|  Call: Node 1  | <--- Stack grows to depth of tree
|________________|
```

---

4. Solution Code

Here is the clean, production-ready code.

Python Solution

```
# 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

class Solution:
    # Key Logic: Recursive Equivalence
    # We define a helper that takes two nodes and returns
    # true if the subtrees rooted at them are flip-equivalent.
    def flipEquiv(self, root1: Optional[TreeNode], root2: Optional[TreeNode]) -> bool:

        # Base Case 1: Both are empty
        # If both are None, they are identical.
        if root1 is None and root2 is None:
            return True

        # Base Case 2: One is empty, or values differ
        # If structure or values mismatch, return False immediately.
        if root1 is None or root2 is None or root1.val != root2.val:
            return False

        # Recursive Step:
        # Check if they are equivalent WITHOUT flipping
        # (Left matches Left AND Right matches Right)
        no_flip = (self.flipEquiv(root1.left, root2.left) and
                   self.flipEquiv(root1.right, root2.right))

        # Optimization: If no_flip is True, we don't need to check flip.
        # But for clarity, we usually write it as an OR condition.
        if no_flip:
            return True

        # Check if they are equivalent WITH flipping
        # (Left matches Right AND Right matches Left)
        flip = (self.flipEquiv(root1.left, root2.right) and
                self.flipEquiv(root1.right, root2.left))

        return flip
```

JavaScript Solution

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
```

```
 * this.val = (val===undefined ? 0 : val)
 * this.left = (left===undefined ? null : left)
 * this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root1
 * @param {TreeNode} root2
 * @return {boolean}
 */
var flipEquiv = function(root1, root2) {
    // Base Case 1: Both null -> Equal
    if (root1 === null && root2 === null) {
        return true;
    }

    // Base Case 2: One null (but not both) or Values different -> Not Equal
    if (root1 === null || root2 === null || root1.val !== root2.val) {
        return false;
    }

    // Recursive Step:
    // Option A: Regular match (Left-Left && Right-Right)
    const noFlip = flipEquiv(root1.left, root2.left) &&
                   flipEquiv(root1.right, root2.right);

    // Short-circuit: If noFlip worked, return true immediately
    if (noFlip) return true;

    // Option B: Flipped match (Left-Right && Right-Left)
    const flip = flipEquiv(root1.left, root2.right) &&
                 flipEquiv(root1.right, root2.left);

    return flip;
};
```

Note 1: The "Canonical Representation" Technique

While the recursive solution above is standard, an **L6 Engineer** might also discuss "Canonical Representation."

The Concept: Instead of comparing two dynamic trees, we force both trees into a strict, static format (a "Canonical Form") and then just check if they are identical.

How it applies here: For every node, if `left.val > right.val`, we swap them. We do this for the entire tree. This forces the tree into a sorted structure.

1. Take `root1`, sort its children recursively so smaller values are always on the left.
2. Take `root2`, do the same.
3. Compare `root1` and `root2` for strict equality.

Why it helps: It changes the problem from "Is there *any* way to flip these?" to "Are these the same tree if we normalize them?" This is often easier to reason about in distributed systems (like comparing JSON objects where key order doesn't matter).

Note 2: Real World & Interview Variations

Google, Meta, and Bloomberg don't always ask this with "Trees". They mask the problem to test your ability to recognize the pattern.

1. **The "File System Diff" (Google):**

- **Question:** You have two directory structures. A user might have dragged and dropped folders around (swapping their order visually), but the contents are the same. Write a function to determine if Folder A and Folder B contain the exact same data hierarchy, ignoring the display order of subfolders.
- **Mapping:** Folders are Nodes. Subfolders are Children. "Ignoring display order" is the "Flip" equivalence (or rather, set equality).

2. **The "JSON Equivalence" (Meta/Bloomberg):**

- **Question:** You have two complex nested JSON objects. In one JSON, the keys are `{ "a": 1, "b": 2 }`. In the other, they are `{ "b": 2, "a": 1 }`. Are these objects semantically equal?
- **Mapping:** This is the N-ary tree version of this problem. You must check if the set of children in Object A matches the set of children in Object B, regardless of order.

3. **The "Chemical Isomer" (Biotech/HealthTech):**

- **Question:** Given two chemical compound structures (represented as graphs/trees), are they the same molecule just viewed from a different rotation?
- **Mapping:** This requires checking graph isomorphism, which is the "hard mode" version of flip equivalent binary trees.

1448. Count Good Nodes in Binary Tree

Here is a breakdown of "Count Good Nodes in Binary Tree" from the perspective of a Senior Staff Engineer. We value clarity, scalability, and understanding the "why" behind the code.

1. Problem Explanation

The Core Concept: We have a binary tree (a root node with at most two children, left and right). We need to count how many nodes are considered "Good."

Definition of a Good Node: A node `X` is "Good" if, starting from the `Root` and walking down to `X`, there are **no nodes** with a value strictly greater than `X`.

In simpler terms: Is the current node the "champion" (or tied for champion) of the path traversed so far? If yes, it's Good. If we saw a bigger number earlier in the path, it's not Good.

Visualizing the Rule:

Let's look at a path from Root to a Leaf.

```
Path:  [Root: 3] -> [Node A: 1] -> [Node B: 4] -> [Node C: 3]

1. Check Root (3):
   - Max value seen so far: 3
   - Is 3 >= 3? YES.
   - Status: GOOD NODE.

2. Check Node A (1):
   - Path so far: 3 -> 1
   - Max value seen so far: 3 (from Root)
   - Is 1 >= 3? NO.
   - Status: NOT Good.

3. Check Node B (4):
   - Path so far: 3 -> 1 -> 4
   - Max value seen so far: 3
   - Is 4 >= 3? YES.
   - New Max found: 4.
   - Status: GOOD NODE.

4. Check Node C (3):
   - Path so far: 3 -> 1 -> 4 -> 3
   - Max value seen so far: 4 (from Node B)
   - Is 3 >= 4? NO.
   - Status: NOT Good.
```

---

2. Solution Explanation

The Strategy: Depth First Search (DFS)

An L5/L6 engineer chooses DFS here because the definition of a "Good Node" depends entirely on the **path** from the root. DFS naturally preserves path context as we dive deep into the tree.

The Algorithm:

1. Start at the root.
2. Maintain a variable, let's call it `max_so_far`. Initially, this is the root's value.
3. Traverse to children (left and right).
4. For every node we visit:

- Compare `node.val` with `max_so_far`.
- If `node.val >= max_so_far`, increment our count. Update `max_so_far` to this new higher value.
- If `node.val < max_so_far`, do not increment. Pass the current `max_so_far` down to the children (do not update it).

Step-by-Step Execution with ASCII Diagrams:

Consider this tree: `[3, 1, 4, 3, null, 1, 5]`

```
      (3)  <-- Root
     /   \
   (1)   (4)
   /     / \
 (3)   (1) (5)
```

Step 1: Start at Root (3)

- Current Max: -Infinity (or just take root value)
- Is 3 >= -Inf? YES. **Count = 1**
- New Max to pass down: 3

```
      [3] (Good! Max: 3)
     /   \
    ?     ?
```

Step 2: Go Left to Node (1)

- Inherited Max: 3
- Is 1 >= 3? NO.
- **Count remains 1**.
- Max to pass down: Still 3 (because 1 didn't beat it).

```
       [3]
      /
    (1) (Bad. Val 1 < Max 3)
    /
   ?
```

Step 3: Go Left again to Node (3)

- Inherited Max: 3
- Is 3 >= 3? YES (Equal is okay!).
- **Count = 2**.
- Max to pass down: 3.

```
       [3]
      /
    (1)
    /
  [3] (Good! Val 3 >= Max 3)
```

(Left side done. Backtrack to Root, go Right)

Step 4: Go Right to Node (4)

- Inherited Max: 3 (From Root)
- Is 4 >= 3? YES.
- **Count = 3**.
- Max to pass down: 4 (Update because 4 is bigger).

```
       [3]
          \
          [4] (Good! Val 4 >= Max 3)
          / \
         ?   ?
```

Step 5: Go Right -> Left to Node (1)

- Inherited Max: 4 (From parent Node 4)
- Is 1 >= 4? NO.
- **Count remains 3**.

Step 6: Go Right -> Right to Node (5)

- Inherited Max: 4
- Is 5 >= 4? YES.
- **Count = 4**.

Final Tally: The Good Nodes are: Root(3), Leaf(3), Node(4), Leaf(5). Total = 4.

---

3. Time and Space Complexity Analysis

In a senior engineering interview, simply stating “O(N)” isn’t enough. We need to explain the bounds based on the structural constraints of the data.

Time Complexity: O(N)

- **Derivation:** We must visit every single node exactly once to determine if it is “Good.” We cannot skip nodes because a node deep in the tree might be huge (e.g., 1,000,000) and count as “Good” regardless of the path.
- **N** = Total number of nodes.

```
  Visualization of Work Done:

     (O)  <-- Visit 1
    /   \
  (O)   (O) <-- Visit 2, 3
  / \   / \
(O) (O)(O) (O) <-- Visit 4, 5, 6, 7

Total Operations is proportional to Total Nodes.
Time = O(N)
```

Space Complexity: O(H)

- **Derivation:** This depends on the “Recursion Stack.” When we go deep into the tree, the computer’s memory holds the state of the function for every level we descend.
- **H** = Height of the tree.

Scenario A: Balanced Tree (Best Case) The height is small (logarithmic). `Space = O(log N)`

```
      (Root)       Level 1
     /      \
   (L)      (R)    Level 2
  /   \    /   \
(LL) (LR)(RL) (RR) Level 3

Stack Depth = 3
```

Scenario B: Skewed Tree (Worst Case) The tree looks like a linked list. The recursion goes all the way to the bottom before returning. `Space = O(N)`

```
(Root)
  \
  (R)
    \
    (RR)
      \
        (RRR) ...

Stack Depth = N (Total number of nodes)
```

Senior Engineer Note: In a non-recursive (iterative) solution using a Queue (BFS) or Stack, we still store nodes proportional to the width or height, maintaining similar complexity profiles.

---

4. Solution Code

Here is how we write this in production-grade style.

Python Solution We use a simple DFS helper function. Note the use of default arguments to make the initial call clean.

```python
class Solution:
    def goodNodes(self, root: TreeNode) -> int:
        """
        Counts nodes that are >= max value on the path from root.
        """

        # Helper function for Depth First Search
        # node: current node being visited
        # max_val: maximum value seen on the path from root to current node
        def dfs(node, max_val):
            # Base Case: If the node is empty, it doesn't count.
            if not node:
                return 0

            # Check if current node is 'Good'
            is_good = 1 if node.val >= max_val else 0

            # Update the max_val for the children.
            # We only pass down the larger of the two:
            # the current max or the current node's value.
            new_max = max(max_val, node.val)

            # Recursively count good nodes in left and right subtrees
            left_count = dfs(node.left, new_max)
            right_count = dfs(node.right, new_max)

            # Sum up current result + left subtree + right subtree
            return is_good + left_count + right_count

        # Initial call: Start at root.
        # Use a very small number for initial max, or root.val
        return dfs(root, float('-inf'))
```

JavaScript Solution In JS, we often use a slightly different pattern or closures to maintain the count.

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 * this.val = (val===undefined ? 0 : val)
 * this.left = (left===undefined ? null : left)
 * this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number}
 */
var goodNodes = function(root) {
    // DFS function to traverse the tree
    // node: current node
    // maxSoFar: the highest value encountered on the path
    const traverse = (node, maxSoFar) => {
        if (!node) return 0;

        let count = 0;

        // Step 1: Check if current node is 'Good'
        if (node.val >= maxSoFar) {
            count = 1;
        }

        // Step 2: Determine new max to pass down
        // Math.max returns the larger of the two numbers
        const newMax = Math.max(maxSoFar, node.val);

        // Step 3: Recurse left and right
        count += traverse(node.left, newMax);
        count += traverse(node.right, newMax);

        return count;
    };

    // Start recursion with the root and a sufficiently low number
    // (or simply root.val)
    return traverse(root, -Infinity);
};
```

New Terms & Techniques

Pre-order Traversal (DFS variant): The technique used here is a form of Pre-order Traversal.

- **Order:** Process Node -> Process Left Child -> Process Right Child.
- **Why it helps:** We need to process the "parent" before the "child" because the child needs to know the parent's `max_val` to make a decision. Information flows strictly from top to bottom.

Real World / Interview Variations

When companies like Google, Meta, or Bloomberg ask this, they rarely use "Binary Trees" explicitly in the problem description. They mask it as a hierarchy problem:

1. **Bloomberg (Organizational Hierarchy):**

- *Problem:* "Given an organizational chart (tree) where every employee has a 'rank', count how many employees have a rank higher than or equal to all their bosses up to the CEO."
- *Mapping:* CEO is Root. "Bosses" are the path. "Rank" is Node Value.

2. **Meta (Visibility/Line of Sight):**

- *Problem:* "You have sensors placed on hills (nodes). A sensor can only transmit data to the base (root) if it is higher than every other hill on the path to the base."
- *Mapping:* Base is Root. Elevation is Value.

3. **Google (Network Signal):**

- *Problem:* "In a mesh network routed as a tree, a packet degrades as it travels. Identify nodes where the signal strength (value) is actually boosted or maintained relative to the input signal received from the gateway (root)."

236. Lowest Common Ancestor of a Binary Tree

Here is how a Senior Staff Engineer (L6) would break down the "Lowest Common Ancestor" (LCA) problem. We focus on clarity, visualizing the flow of data, and understanding the "contract" of our recursive function.

---

1. Problem Explanation

The Lowest Common Ancestor (LCA) of two nodes, `p` and `q`, in a binary tree is the lowest node (deepest level) that has both `p` and `q` as descendants.

Important Rule: A node can be a descendant of itself. This means if `p` is a direct parent of `q`, then `p` is the LCA.

Let's look at a few ASCII scenarios to visualize this.

Scenario A: The "Split" Case Here, `p` and `q` are on different sides of a parent. That parent is the LCA. We are looking for LCA of **Node 6** and **Node 4**.

```
       3  <-- LCA is 3. Why?
      / \     Because 3 is the first point where
     5   1    the paths to 6 and 4 diverge.
    / \
   6   2
      / \
     7   4
```

Scenario B: The "Direct Lineage" Case Here, one node is inside the subtree of the other. We are looking for LCA of **Node 5** and **Node 4**.

```
       3
      / \
     5   1  <-- LCA is 5.
    / \     Why? Because 5 is an ancestor of 4,
   6   2    and 5 is an ancestor of itself.
      / \   It is the "lowest" one they share.
     7   4
```

---

2. Solution Explanation

To solve this efficiently, we use a **Recursive Depth-First Search (DFS)**.

The "Contract" of our Recursive Function To think like an L6 engineer, don't just "write code." Define exactly what the function returns to its caller.

Our function `findLCA(node, p, q)` will return:

1. **The Target Node:** If the current `node` is `p` or `q`, we return the `node` itself.
2. **The LCA:** If the current subtree contains *both* `p` and `q`, we return the LCA found deep below.
3. **Null:** If the current subtree contains *neither* `p` nor `q`.

The Algorithm Logic (The "Bubbling Up" Strategy) We traverse to the bottom of the tree and pass information back up (Post-order traversal).

1. **Base Case:** If current node is `null`, return `null`.
2. **Found Case:** If current node is `p` or `q`, return current node.
3. **Recursive Step:** Ask left child (`left_result`) and right child (`right_result`) what they found.
4. **Decision Time (The Core Logic):**

- **Case 1: Conflict (The LCA found!)** If `left_result` is NOT null AND `right_result` is NOT null, it means `p` is on one side and `q` is on the other. **Current node is the LCA.**
- **Case 2: One Side Match** If only one side returns a node (e.g., `left_result` is not null, but `right` is), return `left_result`. We are passing the found node up the chain.
- **Case 3: Ghost Town** If both are null, return null.

Visualizing the "Bubble Up"

Let's trace finding **LCA(6, 4)**.

```
      [Root: 3]
      /       \
    [5]         [1]
   / \         /
 [6] [2]      ... (1 returns null, no 6 or 4 here)
      / \
    [7] [4]
```

Step 1: The Recursion hits bottom. We go deep.

- Node 6 matches `p`. It returns **6**.
- Node 7 matches nothing. It returns **null**.
- Node 4 matches `q`. It returns **4**.

Step 2: Processing Node 2 Node 2 receives:

- From Left (7): `null`
- From Right (4): `4`
- Logic: One side is not null. Return that side (**4**).

```
      [2] returns 4 upwards
      / \
    (7) (4)
```

Step 3: Processing Node 5 Node 5 receives:

- From Left (6): `6`
- From Right (2): `4` (bubbled up from below)
- **CRITICAL MOMENT:** Both Left and Right returned non-null values.
- **Result:** Node 5 declares "I am the LCA" and returns **5**.

```
      [5] returns 5 (The LCA)
      / \
    (6) (4 <-- bubbled from 2)
```

Step 4: Processing Root 3 Root 3 receives:

- From Left (5): `5` (The LCA)
- From Right (1): `null`
- Logic: One side is not null. Return that side (**5**).

Final Answer: **Node 5**.

---

3. Time and Space Complexity Analysis

Time Complexity: O(N)

- **Reasoning:** In the worst case, we might visit every single node in the tree exactly once to find `p` and `q`.
- N = Number of nodes.

```
Time Complexity Visualization:
We visit nodes 1, 2, 3, 4... once.

      (1)  <-- Visit 1
      / \
    (2) (3) <-- Visit 2, 3
    / \
  (4) (5) ... and so on.

Total steps = Total Nodes (N)
```

Space Complexity: O(H) (Height of the tree)

- **Reasoning:** This depends on the recursion stack (how deep the function calls go).
- **Best Case (Balanced Tree):** O(log N).
- **Worst Case (Skewed Tree):** O(N).

Skewed Tree Visualization (Worst Case SC) If the tree is a straight line, the recursion stack grows to the size of N.

```
Stack Frame 1: Root
 |
 +-> Stack Frame 2: Child
      |
      +-> Stack Frame 3: Grandchild
           |
           ...
           |
           +-> Stack Frame N: Leaf
```

Balanced Tree Visualization (Best Case SC) The stack only grows as deep as the height.

```
      Root
     /    \     <-- Stack depth is roughly log(N)
   Left  Right
```

---

4. Solution Code

Here is the implementation. I have added comments to explain the “L5/L6” level decision making inside the code.

Python Solution

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

class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeN
        # --- Base Cases ---
        # 1. If we hit the bottom (None), return None.
        if not root:
            return None

        # 2. If we find either p or q, return the node itself.
        # This stops the recursion for this specific path because:
        # - If the other node is in the subtree of this node, THIS node is the LCA.
        # - If the other node is elsewhere, this node is a candidate to be passed up.
        if root == p or root == q:
            return root

        # --- Recursive Steps (Dive Deep) ---
        left_result = self.lowestCommonAncestor(root.left, p, q)
        right_result = self.lowestCommonAncestor(root.right, p, q)

        # --- Decision Logic (Post-order processing) ---

        # Case 1: We found p on one side and q on the other.
        # This means the current 'root' is the split point (LCA).
        if left_result and right_result:
            return root

        # Case 2 & 3: We found something on one side, but nothing on the other.
        # We bubble up the non-null result (which could be p, q, or an already found LCA).
        # If both are None, this returns None.
        return left_result if left_result else right_result
```

JavaScript Solution

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 * this.val = val;
 * this.left = this.right = null;
 * }
 */

/**
 * @param {TreeNode} root
 * @param {TreeNode} p
 * @param {TreeNode} q
 * @return {TreeNode}
 */
var lowestCommonAncestor = function(root, p, q) {
    // --- Base Cases ---
    // If leaf node reached or root is null
    if (!root) return null;

    // If current node is either p or q, we found one of them.
    // Return it immediately.
    if (root === p || root === q) return root;

    // --- Recursive Search ---
    // Look in left and right subtrees
    const leftResult = lowestCommonAncestor(root.left, p, q);
    const rightResult = lowestCommonAncestor(root.right, p, q);

    // --- Resolution ---

    // If both children returned a non-null value, it means p is in one
    // subtree and q is in the other. This root is the intersection.
    if (leftResult && rightResult) {
        return root;
    }

    // Otherwise, return the non-null child (bubbling up the result).
    // If both are null, this returns null.
    return leftResult !== null ? leftResult : rightResult;
};
```

Note 1: Terminology & Technique

The primary technique used here is **Post-order Traversal**.

- **What it is:** We process the children *first* before making a decision at the parent (Root).
- **Why it helps:** We cannot decide if the current node is the LCA until we know if `p` and `q` exist in its subtrees. We must "ask" the children first, gather the data, and then process the logic at the node level.

Note 2: Real World & Interview Variants

When companies like Google, Meta, or Bloomberg ask this, they rarely ask just the "vanilla" version essentially because it's too common. They ask these variations to test if you understand the *concept* vs just memorizing code:

1. **LCA of Deepest Leaves (Meta/Facebook):**

- Instead of giving you `p` and `q`, they ask you to find the LCA of all the deepest leaves in the tree. You first have to calculate height, identifying the deepest nodes, and then run LCA logic on them.

2. **LCA with Parent Pointers (Google/Bloomberg):**

- **Scenario:** The tree nodes have a `parent` pointer.
- **Twist:** You are given `p` and `q` but NOT the root.
- **Real-world mapping:** This is exactly like finding the merge point of two Git branches or the intersection of two Linked Lists. You don't need recursion; you trace back parent pointers.

3. **LCA in a File System (Google):**

- **Scenario:** You are given a list of file paths (e.g., `/home/user/docs/a.txt` and `/home/user/pictures/b.jpg`). Find the deepest common folder.
- **Connection:** This is LCA on a generic tree (N-ary tree), not just a binary tree. The logic is the same: the LCA is the folder where the path strings diverge.

4. **LCA of K Nodes (Google):**

- **Scenario:** Find the LCA of a list of `K` nodes, not just two.
- **Solution:** You can run LCA on `node1` and `node2` to get `result1`. Then run LCA on `result1` and `node3`, and so on. Or modify the recursion to return a count of targets found.

5. **DOM Tree Manipulation (Frontend Interviews):**

- **Scenario:** Given two HTML Elements in a DOM, find the closest common container `div` or `section` that wraps both.
- **Connection:** The DOM is a tree. `element.parentNode` allows you to solve this using the "Parent Pointer" approach.

106. Construct Binary Tree from Inorder and Postorder Traversal

Here is a breakdown of how a Senior (L5/L6) Engineer at Google would approach, explain, and solve **LeetCode 106: Construct Binary Tree from Inorder and Postorder Traversal**.

---

1. Problem Explanation

The Goal: We have two arrays representing the same binary tree but traversed in different orders:

1. **Inorder:** Left Node -> Root Node -> Right Node
2. **Postorder:** Left Node -> Right Node -> Root Node

We need to reconstruct the exact structure of the original binary tree from these two lists.

The Constraints:

- You may assume that duplicates do not exist in the tree. (This is crucial because it allows us to uniquely identify nodes).
- We are guaranteed that a valid tree exists for the given inputs.

The "Senior Engineer" Insight: An L5/L6 engineer looks for the **defining characteristics** of the data structures:

- **Postorder** gives us the **ROOT** immediately. Because Postorder is (Left, Right, Root), the *last* element in the Postorder list is always the root of the current subtree.
- **Inorder** gives us the **STRUCTURE**. Once we know the root (from Postorder), we can find that root value in the Inorder list. Everything to the left of it belongs to the left subtree, and everything to the right belongs to the right subtree.

---

2. Solution Explanation (Visualized)

We will use a **Recursive** strategy. We will peel off the root from the Postorder list, find it in the Inorder list to determine the boundaries of the left and right subtrees, and then recursively repeat the process for the children.

Optimization (The L5 Move): A naive approach scans the Inorder array linearly to find the root's index every time. This results in **O(N^2)** time complexity. To solve this efficiently (**O(N)**), we first build a **Hash Map** (Dictionary in Python, Object/Map in JS) that stores `{ value: index }` for the Inorder array. This lets us find the root's position in **O(1)** time.

The Algorithm Step-by-Step Let's trace this with an example:

- **Inorder:** `[9, 3, 15, 20, 7]`
- **Postorder:** `[9, 15, 7, 20, 3]`

Step 1: Identify the Global Root Look at the **end** of the Postorder list. The value is `3`. This is our Root.

```
Postorder: [9, 15, 7, 20, (3)]  <-- Last element is Root
Tree so far:
      3
    /   \
   ?     ?
```

Step 2: Slice the Inorder List Find `3` in the Inorder list.

- Left of `3`: `[9]` -> These form the **Left Subtree**.
- Right of `3`: `[15, 20, 7]` -> These form the **Right Subtree**.

```
Inorder Index Map: {9:0, 3:1, 15:2, 20:3, 7:4}
Root Value: 3
Root Index in Inorder: 1

Inorder Array Splitting:
Indices:  [0]     [1]     [2,  3,  4]
Values:   [9]     [3]     [15, 20, 7]
           ^       ^           ^
       Left Child  Root    Right Children
```

Step 3: Recurse on the Right Subtree *Critical Note:* In Postorder, the nodes are arranged `[Left Part, Right Part, Root]`. So, if we pop the Root from the end, the *next* elements at the end of the list belong to the **Right Subtree**. We **must** process the Right child before the Left child when processing the Postorder array from back to front.

- Current Postorder: `[9, 15, 7, 20]` (3 is removed)
- New Root for Right Subtree: `20` (Last element)

```
Tree so far:
      3
    /   \
   ?    20
       /  \
      ?    ?
```

Now we look at the Inorder scope for this Right Subtree (Indices 2 to 4: `[15, 20, 7]`).

- Find `20` in Inorder. Index is 3.
- Left of `20` (inside this scope): `[15]`
- Right of `20` (inside this scope): `[7]`

Step 4: Recurse on Right's Right Child

- Current Postorder: `[9, 15, 7]` (20 is removed)
- New Root: `7`

```
Tree so far:
      3
    /   \
   ?     20
        /  \
      ?     7
```

In the Inorder scope for `[7]`, there are no items left or right of it. It's a leaf.

Step 5: Recurse on Right's Left Child

- Current Postorder: `[9, 15]` (7 is removed)
- New Root: `15`

```
Tree so far:
      3
    /   \
   ?     20
        /  \
      15    7
```

In the Inorder scope for `[15]`, it is also a leaf.

Step 6: Recurse on Left Subtree We finished the whole Right side. Now we return to the Left side of the main root (3).

- Current Postorder: `[9]`
- New Root: `9`

```
Final Tree:
      3
    /   \
   9     20
        /  \
      15    7
```

---

3. Time and Space Complexity Analysis

Time Complexity: O(N) We visit every node exactly once to construct it. The “lookup” to find where to split the tree is done using a Hash Map, which takes constant time, O(1).

```
Time Complexity Visualization:

1. Map Construction:    [====================]  O(N)
                        Iterate Inorder once

2. Tree Construction:   (Node 1) -> O(1) work
                        (Node 2) -> O(1) work
                        ...
                        (Node N) -> O(1) work
                        ---------------------
                        Total: N * O(1) = O(N)

Total Time = O(N) + O(N) = O(N)
```

Space Complexity: O(N) We need space for the Hash Map and the Recursion Stack.

```
Space Complexity Visualization:

1. Hash Map Storage:
   +-----------+
   | Key | Val |
   |  9  |  0  |
   |  3  |  1  |  ... Stores N entries
   | ... | ... |
   +-----------+
   Space: O(N)

2. Recursion Stack (Memory used by function calls):

   Case A: Balanced Tree      Case B: Skewed Line Tree
        O                     O
       / \                     \
      O   O                     O
     / \ / \                     \
    O  O O  O                     O
                                   \
   Stack Depth:                 Stack Depth:
   approx log(N)                approx N (Worst Case)

   Space: O(N) in worst case.
```

```
Total Space = O(N) + O(N) = O(N)
```

---

4. Solution Code

Python Solution

```python
# 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

class Solution:
    def buildTree(self, inorder: list[int], postorder: list[int]) -> TreeNode:
        """
        Builds a binary tree from inorder and postorder traversal arrays.
        Strategy:
        1. Use Postorder to find the root (last element).
        2. Use Inorder to split data into Left and Right subtrees.
        3. Recurse.
        """

        # Optimization: Build a hash map for O(1) index lookups of inorder values.
        # This prevents us from having to search the array in every recursive step.
        inorder_map = {val: index for index, val in enumerate(inorder)}

        def helper(in_left, in_right):
            # Base Case: If the left pointer exceeds the right,
            # there are no elements left to construct the subtree.
            if in_left > in_right:
                return None

            # 1. Pick the last element from postorder as the current root.
            # We pop it so the next recursive call gets the next correct root.
            root_val = postorder.pop()
            root = TreeNode(root_val)

            # 2. Find the split point in the inorder array
            index = inorder_map[root_val]

            # 3. Recurse.
            # CRITICAL: We MUST build the Right subtree before the Left subtree.
            # Why? Because postorder is [Left, Right, Root].
            # Once we pop Root, the end of the list contains the Right subtree nodes.
            root.right = helper(index + 1, in_right)
            root.left = helper(in_left, index - 1)

            return root

        # Initialize recursion with the full range of the inorder array
        return helper(0, len(inorder) - 1)
```

JavaScript Solution

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 * this.val = (val===undefined ? 0 : val)
 * this.left = (left===undefined ? null : left)
 * this.right = (right===undefined ? null : right)
 * }
 */

/**
 * @param {number[]} inorder
 * @param {number[]} postorder
 * @return {TreeNode}
 */
var buildTree = function(inorder, postorder) {
    // Optimization: Map value -> index for O(1) access
    const inorderMap = new Map();
    inorder.forEach((val, index) => inorderMap.set(val, index));

    // We use a mutable pointer for postorder index to track our position
    // starting from the end of the array (the root).
    let postIndex = postorder.length - 1;

    /**
     * Recursive helper function
     * @param {number} inLeft - Left boundary of current inorder scope
     * @param {number} inRight - Right boundary of current inorder scope
     */
    function helper(inLeft, inRight) {
        // Base Case: No elements left in this subtree range
        if (inLeft > inRight) {
            return null;
        }

        // 1. Get the root value from the current position in postorder
        const rootVal = postorder[postIndex];
        const root = new TreeNode(rootVal);

        // Move the postorder pointer backwards for the next call
        postIndex--;

        // 2. Find the split point
        const index = inorderMap.get(rootVal);

        // 3. Recurse
        // IMPORTANT: Right child first!
        // Postorder traverses Left -> Right -> Root.
        // Traversing backward (Root first), we encounter Right, then Left.
        root.right = helper(index + 1, inRight);
        root.left = helper(inLeft, index - 1);

        return root;
    }

    return helper(0, inorder.length - 1);
};
```

Note 1: Terminology Breakdown

- **Hash Map (or Hash Table):** A data structure that maps keys to values. In this problem, it allows us to ask "Where is the number 3 in the inorder array?" and get the answer "Index 1" instantly, without scanning the list.
- **Recursion (DFS - Depth First Search):** A method where the solution depends on solutions to smaller instances of the same problem. Here, building the *whole* tree is the same logic as building the *left* subtree, which is the same logic as building the *left-left* subtree.
- **Postorder Traversal:** A way of walking the tree: visit Left child, then Right child, then the Node itself.
- **Inorder Traversal:** A way of walking the tree: visit Left child, then the Node itself, then Right child.

Note 2: Real World & Indirect Interview Variations

Top-tier companies (Google, Meta, Bloomberg) rarely ask this verbatim anymore. Instead, they ask practical variations:

1. **Bloomberg (Financial Data):** "You have a stream of organizational hierarchy data. One stream lists employees by 'rank' (Inorder-ish) and another by 'exit interview time' (Postorder-ish). Reconstruct the Org Chart."

- *Real World Application:* Reconstructing dependency graphs or organizational charts from logs.

2. **Meta (Serialization):** "Design a serialization format for a binary tree that is as compact as possible. Then, write the deserializer."

- *Real World Application:* Sending DOM trees (React Virtual DOM) over the network. If you send the tree structure as two simple arrays (Inorder + Postorder) instead of a heavy JSON object with nested pointers, you save bandwidth.

3. **Google (Expression Parsing):** "Given a mathematical expression in Reverse Polish Notation (which is essentially Postorder), convert it into an Expression Tree to evaluate it."

- *Real World Application:* Compilers and Calculators use this to understand order of operations (like `3 + 4 * 5`).

4. **System Design Context:** Restoring a distributed system's state. If you have a log of operations completed (Postorder) and a snapshot of the system's sorted state (Inorder), how do you rebuild the dependency tree of tasks?

129. Sum Root to Leaf Numbers

Here is a breakdown of "Sum Root to Leaf Numbers" from the perspective of a Senior Staff Engineer. We value clarity, maintainability, and understanding the "why" behind the code over just getting the green checkmark.

1. Problem Explanation

The core of this problem is interpreting a path in a binary tree not as a sequence of independent numbers, but as digits in a single number.

Imagine you are walking down a path. Every time you step on a new node, you append that node's digit to the end of the number you are currently holding.

The Rule:

- Start at the Root.
- Every step down shifts your current number to the left (multiplies by 10) and adds the new node's value.
- When you hit a "Leaf" (a node with no children), that number is complete.
- Sum up all complete numbers found at the leaves.

Example A:

```
    1
   / \
  2   3
```

- **Path 1 (Left):** Start at `1`. Move to `2`. Number becomes `12`. It's a leaf. **Save 12**.
- **Path 2 (Right):** Start at `1`. Move to `3`. Number becomes `13`. It's a leaf. **Save 13**.
- **Total:** 12 + 13 = **25**.

Example B (Deeper Tree):

```
      4
     / \
    9   0
   / \
  5   1
```

Here we have three paths to leaves:

1. **4 -> 9 -> 5**:

- Start: `4`
- Next: `49`
- Leaf: `495`

2. **4 -> 9 -> 1**:

- Start: `4`
- Next: `49`
- Leaf: `491`

3. **4 -> 0**:

- Start: `4`
- Leaf: `40`

Total Sum: 495 + 491 + 40 = **1026**.

---

2. Solution Explanation (DFS Approach)

As an L5/L6 engineer, I recommend **Depth First Search (DFS)** here. Why? Because the problem is inherently about "paths." We want to dive deep from the root to the leaf to construct the full number before moving to the next path.

The Algorithm:

1. We define a function `dfs(node, current_sum)`.
2. **State Transition:** When entering a node, we update the `current_sum`: `new_sum = (current_sum * 10) + node.val`
3. **Base Case (Leaf):** If the node has no children, we have found a complete number. Return `new_sum`.

4. **Recursive Step:** If the node has children, pass the `new_sum` down to them. The result for the current node is the sum of results from its left and right children. `return dfs(left, new_sum) + dfs(right, new_sum)`

Visual Walkthrough (Step-by-Step) Let's trace the execution on this tree:

```
    4
   / \
  9   0
 /
5
```

Step 1: Start at Root (4)

- Initial sum coming in: `0`
- Calculation: `0 * 10 + 4 = 4`
- Action: Not a leaf. Call Left and Right.

```
  [4] (Current Sum: 4)
  / \
[?] [?]
```

Step 2: Go Left to (9)

- Sum coming in: `4`
- Calculation: `4 * 10 + 9 = 49`
- Action: Not a leaf. Call Left (5) and Right (None).

```
    [4]
    /
  [9] (Current Sum: 49)
  /
[?]
```

Step 3: Go Left to (5)

- Sum coming in: `49`
- Calculation: `49 * 10 + 5 = 495`
- Action: **It is a leaf!** (No left, no right children).
- **RETURN 495** up the chain.

```
    [4]
    /
  [9]
  /
[5] -> Leaf! Found number: 495
```

Step 4: Back at (9)

- Left child returned: `495`.
- Right child is `null` (treat as 0).

- Total for Node 9: `495 + 0 = 495`.
- **RETURN 495** up to Node 4.

Step 5: Back at Root (4) - Process Right Side

- We finished the left side. Now call Right child (0).
- Sum coming in: `4` (passed from root).

Step 6: Go Right to (0)

- Sum coming in: `4`
- Calculation: `4 * 10 + 0 = 40`
- Action: **It is a leaf!**
- **RETURN 40** up the chain.

```
   [4]
     \
      [0] -> Leaf! Found number: 40
```

Step 7: Final Sum at Root (4)

- Left side returned: `495`
- Right side returned: `40`
- Total: `495 + 40 = 535`.

---

3. Time and Space Complexity Analysis

Time Complexity: O(N) Where N is the number of nodes.

Derivation: We simply visit every node exactly once to do the multiplication and addition.

```
Graph Traversal:

    (Start)
       |
       v
    [Node 1]  <-- 1 visit (Constant time math)
       |
    [Node 2]  <-- 1 visit (Constant time math)
       |
      ...
    [Node N]  <-- 1 visit

Total Work = N * k (constant) = O(N)
```

Space Complexity: O(H) Where H is the height of the tree.

Derivation: We use recursion. The computer stores "stack frames" (memory for each function call) as we dive deep. We only store as many frames as the deepest path in the tree.

Scenario A: Balanced Tree (Best Case) Height is approx log(N).

```
      O      <-- Stack frame 1
     / \
    O   O    <-- Stack frame 2
   / \ / \
  O  O O  O  <-- Stack frame 3

Space = O(log N)
```

Scenario B: Skewed Tree (Worst Case) The tree looks like a linked list. Height is N.

```
    O        <-- Stack frame 1
     \
      O      <-- Stack frame 2
       \
        O    <-- Stack frame 3
         \
          O  <-- Stack frame 4

Space = O(N)
```

---

4. Solution Code

Python Solution I used a helper function `dfs` here. In a production environment, this isolates the recursion logic from the main class interface.

```python
# 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

class Solution:
    def sumNumbers(self, root: Optional[TreeNode]) -> int:
        # L5 Note: Always handle the edge case of an empty tree immediately.
        # It reduces nesting and makes the code "fail fast".
        if not root:
            return 0

        def dfs(node, current_sum):
            # 1. Update the sum for the current node
            # We shift the previous sum by 10 (decimal shift) and add current val
            new_sum = current_sum * 10 + node.val

            # 2. Check if Leaf
            # A leaf has no left child AND no right child
            if not node.left and not node.right:
                return new_sum

            # 3. Recursive Step (Depth First Search)
            total_sum = 0

            # If left child exists, explore it
            if node.left:
                total_sum += dfs(node.left, new_sum)

            # If right child exists, explore it
            if node.right:
                total_sum += dfs(node.right, new_sum)

            return total_sum

        # Start the recursion with the root and an initial sum of 0
        return dfs(root, 0)
```

JavaScript Solution In JavaScript, we must be careful with types, though LeetCode handles the tree structure. The logic remains identical.

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 * this.val = (val===undefined ? 0 : val)
 * this.left = (left===undefined ? null : left)
 * this.right = (right===undefined ? null : right)
 * }
 */

/**
 * @param {TreeNode} root
 * @return {number}
 */
var sumNumbers = function(root) {
    if (!root) return 0;

    // Helper function to traverse the tree
    // currentSum holds the value accumulated from root down to parent
    function dfs(node, currentSum) {
        if (!node) return 0;

        // Shift left and add current node's value
        const newSum = currentSum * 10 + node.val;

        // Base Case: If it's a leaf node, return the fully formed number
        if (!node.left && !node.right) {
            return newSum;
        }

        // Recursively sum the left and right subtrees
        // We do not need to check if left/right exist before calling
        // because the first line 'if (!node) return 0' handles nulls gracefully.
        return dfs(node.left, newSum) + dfs(node.right, newSum);
    }

    return dfs(root, 0);
};
```

Note 1: Terminology & Techniques

Pre-order Traversal: The technique used here is a variation of Pre-order Traversal (Root -> Left -> Right).

- **Why it helps:** We process the "Root" (current node) to update the math *before* we dive into the "Left" or "Right" children. This ensures that when we reach the children, they have the correct accumulated value from their ancestors.

State Passing: Notice we passed `current_sum` as an argument.

- **Why it helps:** This avoids using a global variable. In multi-threaded environments (common at Google/Meta), global variables are dangerous. Passing state locally ensures the function is "pure" and thread-safe.

Note 2: Real World & Interview Variations

Companies like Google, Bloomberg, and Meta often tweak this question to test if you memorized the code or if you understand the tree traversal.

1. **Binary Representation (Meta/Facebook):**

- *Question:* "Same problem, but the tree contains only 0s and 1s. Treat the path as a binary string (e.g., 1->0->1 is 5, not 101)."
- *Adjustment:* Change `current_sum * 10` to `current_sum * 2`.

2. **String Concatenation (Bloomberg):**

- *Question:* "The nodes contain characters 'A', 'B', 'C'. Return the concatenated strings of all paths."
- *Adjustment:* Change math addition to string concatenation.

3. **File System Paths (Google):**

- *Question:* "Given a tree where nodes are directory names, print all full file paths."
- *Adjustment:* Join node values with a `/` slash. This is the exact same logic: accumulating state from root to leaf.

199. Binary Tree Right Side View

Here is how a Senior (L5/L6) Engineer would deconstruct and solve the "Binary Tree Right Side View" problem, focusing on clarity, edge cases, and maintainable code.

1. Problem Explanation

The "Right Side View" of a binary tree is the set of nodes visible when you look at the tree from the right side.

The Common Misconception: Many candidates initially think, "Oh, I just need to traverse the right child pointers: `root -> right -> right...`".

The Reality: If a right branch is shorter than a left branch, you will see the nodes from the left branch "peeking" out from behind.

Visualizing the "View" Imagine you are standing to the right of the tree. You can only see the **last node** present at each horizontal level (depth).

Example A: The "Obvious" Case (Right side is fully populated)

```
      1           <--- Level 0: You see 1
    /   \
   2     3        <--- Level 1: You see 3
    \     \
     5     4      <--- Level 2: You see 4
```

```
Right Side View: [1, 3, 4]
```

Example B: The "Tricky" Case (The Left-Heavy Tree) (The right side ends early, revealing the left side)

```
      1           <--- Level 0: You see 1
    /   \
   2     3        <--- Level 1: You see 3 (blocks 2)
  /
  4                  <--- Level 2: You see 4 (because there is no node to its right!)
 /
5                    <--- Level 3: You see 5

Right Side View: [1, 3, 4, 5]
```

Note: Even though 4 and 5 are "left children", they are the rightmost nodes at their respective levels.

---

2. Solution Explanation

To an L5 engineer, this problem maps directly to a **Level Order Traversal** (or Breadth-First Search - BFS).

Why BFS? Since the problem asks us to pick one node per *level* (depth), processing the tree level-by-level is the most natural approach. If we traverse each level from left to right, the **last node** we touch at that level is the one visible from the right.

The Algorithm:

1. Initialize a Queue and add the Root.
2. Loop while the Queue is not empty (this means we have levels to process).
3. Identify the number of nodes currently in the queue (`level_length`). This represents all nodes at the *current* level.
4. Iterate `level_length` times to process just this level:

- Pop a node.
- If it has a left child, add it to the queue.
- If it has a right child, add it to the queue.
- **Crucial Step:** If this is the *last* node in our iteration (the loop index equals `level_length - 1`), this is the **Rightmost Node**. Add it to our result list.

ASCII Visualization of the Algorithm Let's trace **Example B** (The Tricky Case).

Tree Structure:

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

Step 1: Initialization

- `Result` = []
- `Queue` = [Node 1]

Step 2: Processing Level 0

- `level_length` = 1 (Only Node 1 is in queue)
- **Iteration 1 of 1:**
- Pop Node 1.
- Add left child (2) to Queue.
- Add right child (3) to Queue.
- Is this the last node of the level? **Yes.** Add 1 to Result.
- **End Level 0**
- `Result` = [1]
- `Queue` = [2, 3]

Step 3: Processing Level 1

- `level_length` = 2 (Nodes 2 and 3 are in queue)
- **Iteration 1 of 2:**
- Pop Node 2.
- Add left child (4) to Queue.
- Is this the last node? **No.**
- **Iteration 2 of 2:**
- Pop Node 3.
- No children to add.
- Is this the last node? **Yes.** Add 3 to Result.
- **End Level 1**
- `Result` = [1, 3]
- `Queue` = [4]

Step 4: Processing Level 2

- `level_length` = 1 (Node 4)
- **Iteration 1 of 1:**
- Pop Node 4.
- No children.
- Is this the last node? **Yes.** Add 4 to Result.
- **End Level 2**
- `Result` = [1, 3, 4]

- `Queue` = [] (Empty)

Step 5: Done

- Return `[1, 3, 4]`

---

3. Time and Space Complexity Analysis

Time Complexity: O(N) We visit every node exactly once.

- N = Total number of nodes in the tree.
- We perform constant time operations (enqueue, dequeue, check value) for each node.

```
Visualization of Time:
[Node 1] -> [Node 2] -> [Node 3] -> [Node 4] ... -> [Node N]
   ^           ^           ^           ^               ^

  1 ops       1 ops       1 ops       1 ops           1 ops

Total = N * 1 = O(N)
```

Space Complexity: O(D) (Where D is the maximum diameter or width of the tree).

- In the worst case (a perfect binary tree), the bottom-most level contains roughly N/2 nodes.
- Our queue holds all nodes of a single level at once.
- Therefore, the space required is proportional to the maximum width of the tree.

```
Space Visualization (Queue Size):

Level 0: [1]                 (Size 1)
Level 1: [2, 3]              (Size 2)
Level 2: [4, 5, 6, 7]        (Size 4)  <-- Max Space Used
...
Level k: [ ... N/2 nodes ... ]

Max Width = O(N) in a full tree.
Max Width = O(1) in a skewed tree (like a linked list).
```

---

4. Solution Code

Python Implementation

```python
from collections import deque
```

```python
# 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 rightSideView(root):
    """
    Returns the values of the nodes visible from the right side.
    Strategy: Breadth-First Search (Level Order Traversal).
    """
    # 1. Edge Case: If tree is empty, return empty list immediately.
    if not root:
        return []

    result = []
    # Initialize queue with the root node
    queue = deque([root])

    # 2. Loop while there are nodes to process
    while queue:
        # Capture the number of nodes at the CURRENT level.
        # This acts as a snapshot. We must iterate exactly this many times.
        level_length = len(queue)

        for i in range(level_length):
            node = queue.popleft()

            # 3. If this is the last node in the current level's snapshot,
            # it is the "Rightmost" node.
            if i == level_length - 1:
                result.append(node.val)

            # 4. Add children to queue for the NEXT level
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

    return result
```

Javascript Implementation

```javascript
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 * this.val = (val===undefined ? 0 : val)
 * this.left = (left===undefined ? null : left)
 * this.right = (right===undefined ? null : right)
 * }
 */

/**
 * @param {TreeNode} root
 * @return {number[]}
 */
var rightSideView = function(root) {
    // 1. Edge Case: Empty tree
    if (!root) return [];

    const result = [];
    const queue = [root]; // Using array as queue

    // 2. Loop while queue has elements
    while (queue.length > 0) {
        // Snapshot the size of the current level
        const levelLength = queue.length;

        for (let i = 0; i < levelLength; i++) {
            // Shift removes the first element (De-queue)
            const node = queue.shift();

            // 3. Logic: If it's the last index of this level, it's visible
            if (i === levelLength - 1) {
                result.push(node.val);
            }

            // 4. Enqueue children (Left first, then Right)
            if (node.left) queue.push(node.left);
            if (node.right) queue.push(node.right);
        }
    }

    return result;
};
```

Note 1: Terms & Techniques

Breadth-First Search (BFS): This is the core "Level Order" technique used here. In tree problems, whenever you see requirements related to "levels," "depths," or "views" (top view, bottom view, right view), BFS is usually the superior tool because it naturally organizes data by horizontal layers.

DFS Alternative: You *can* solve this with Depth First Search (Reverse Preorder: Root -> Right -> Left). You would pass a `depth` variable. The first time you visit a specific depth, you save that node. Since we visit Right children before Left children in this variation, the first node we see at `depth X` is guaranteed to be the rightmost one.

Note 2: Indirect/Real-World Interview Contexts

Big Tech companies (Google, Meta, Bloomberg) rarely ask "Solve Leetcode 199" directly in 2024. They wrap it in a practical scenario:

1. **UI/DOM Rendering (Meta/Google):**

- *Question:* "Given a tree of DOM elements where each element has coordinates, calculate which elements are visible to a user if an 'overlay' or 'sidebar' covers the left 80% of the screen?"
- *Connection:* This is essentially a "Right Side View" problem. You are determining the right-most elements that aren't obscured.

2. **Organization/reporting hierarchies (Bloomberg):**

- *Question:* "We have an employee hierarchy tree. We need to generate a report showing only the 'Point of Contact' for every tier of the organization. The Point of Contact is defined as the most recently hired person at that specific rank (assuming the tree is ordered by hire date from left to right)."
- *Connection:* This requires finding the right-most node (last processed) at every level of the hierarchy.

3. **Network Broadcasting:**

- *Question:* "In a mesh network arranged like a tree, a signal is propagated level-by-level. If we place a sensor array on the East side of the network field, which nodes' signals will be detected first without interference?"

103. Binary Tree Zigzag Level Order Traversal

Here is a comprehensive breakdown of **LeetCode 103: Binary Tree Zigzag Level Order Traversal**, structured as if a Senior Google Engineer (L5/L6) were guiding you through a system design or algorithmic whiteboard session.

---

1. Problem Explanation

The Goal: We need to traverse a binary tree level by level. However, unlike a standard "Level Order Traversal" where we always read from left to right, we need to **alternate directions**.

- **Level 0 (Root):** Read Left -> Right
- **Level 1:** Read Right -> Left
- **Level 2:** Read Left -> Right
- **Level 3:** Read Right -> Left
- …and so on.

Input: The root node of a binary tree. **Output:** A list of lists, where each inner list contains the node values of that specific level in the requested order.

Visualizing the "Zigzag" Pattern:

Imagine a snake crawling down the tree:

```
    3    --------> (Level 0: Left to Right)
   / \
  9  20  <-------- (Level 1: Right to Left)
    /  \
   15   7  --------> (Level 2: Left to Right)
```

Expected Output: `[[3], [20, 9], [15, 7]]`

---

2. Solution Explanation (Deep Dive)

As a senior engineer, my first instinct is to identify the underlying pattern. This is clearly a **Breadth-First Search (BFS)** problem because we are processing the tree row by row.

The Strategy: Two-Ended Construction

1. **Standard BFS:** We use a **Queue** to store nodes. We process nodes level by level.
2. **The Twist:** For every level, we need a list of values.

- If the current level is **even** (0, 2, 4…), we append values normally (Left -> Right).
- If the current level is **odd** (1, 3, 5…), we insert values in reverse (Right -> Left).

There are two main ways to handle the "reverse" part:

- **Method A (Post-process):** Collect normally, then `reverse()` the list if it's an odd level. (Easier to code, slightly inefficient due to double pass).

- **Method B (Deque Construction - Optimal):** Use a double-ended queue (deque) for the *current level's list.* If standard order, add to the **tail**. If reverse order, add to the **head**.

We will use **Method B** or efficient list indexing to avoid the O(K) reversal cost, ensuring maximum performance.

Detailed ASCII Walkthrough Let's trace this tree:

```
      1
     / \
    2   3
   / \   \
  4   5   6
```

Variables:

- `queue`: Stores nodes to visit next.
- `result`: Final list of lists.
- `left_to_right`: Boolean flag (starts `True`).

Step 1: Initialize

- `queue`: `[Node(1)]`
- `left_to_right`: `True`

Step 2: Process Level 0 (Size = 1)

- We pop `1`.
- Direction is `True` (Left -> Right), so add `1` to `current_level` list normally.
- Add children of `1` (which are `2` and `3`) to the queue.

```
Processing Node: 1
Current Level List: [1]

   Queue State (for next level):
   [ Front | 2 , 3 | Rear ]
```

- Finish Level 0. Add `[1]` to result.
- Flip direction: `left_to_right` = `False`.

Step 3: Process Level 1 (Size = 2)

- Queue contains `[2, 3]`.
- **Important:** Even though direction is "Right to Left", we **always** process queue nodes Left to Right (First In, First Out) to find children correctly. The "Zigzag" happens only in how we *store* the values.
- **Pop 2:** Direction is `False`.
- Add children (`4`, `5`) to queue.

- *Storage:* Since we want Right->Left, we can just collect them normally `[2, 3]` and reverse at the end, OR insert `2` into a new list. Let's imagine we collect `[2, 3]`.
- **Pop 3:** Direction is `False`.
- Add child (`6`) to queue.
- *Storage:* List is `[2, 3]`.
- Finish Level 1.
- Direction was `False`, so we reverse the values: `[3, 2]`.
- Add to result.
- Flip direction: `left_to_right` = `True`.

```
Processing Level 1
Raw Values Found: 2, 3
Direction: Reverse (<---)
Stored as: [3, 2]

   Queue State (for next level):
   [ Front | 4 , 5 , 6 | Rear ]
```

Step 4: Process Level 2 (Size = 3)

- Queue contains `[4, 5, 6]`.
- **Pop 4:** Direction `True`. Store `4`. (No children).
- **Pop 5:** Direction `True`. Store `5`. (No children).
- **Pop 6:** Direction `True`. Store `6`. (No children).

```
Processing Level 2
Raw Values Found: 4, 5, 6
Direction: Normal (--->)
Stored as: [4, 5, 6]

   Queue State (for next level):
   [ Empty ]
```

Final Result: `[[1], [3, 2], [4, 5, 6]]`

---

3. Time and Space Complexity Analysis

In a Google interview, you shouldn't just state the complexity; you should derive it.

Time Complexity: O(N)

Why? Let's look at the operations per node using a block diagram.

```
For every node in the tree (N nodes total):

   [ Enqueue ] ---> Happens exactly 1 time
       |
   [ Dequeue ] ---> Happens exactly 1 time
       |
   [ Value Processing ] ---> Constant time O(1) insertion
```

Total Work = N * (1 + 1 + 1) operations = 3N. In Big-O notation, constants drop out. **Total Time = O(N)**.

(Note: If using `list.insert(0, val)` in Python, inside the loop, that is O(K) where K is level width. Doing this repeatedly makes it O(N^2) in worst case (skewed tree). That is why we usually append and then reverse, or use a pre-allocated array/deque for O(1) insertion. The solution below uses the optimal approach.)

Space Complexity: O(N)

We need to look at the maximum memory consumed at any specific point.

1. **Result Storage:** We must store N integers. O(N).
2. **Queue Storage:** The queue stores the nodes of the "current level".

```
Worst Case Scenario: A Perfect Binary Tree
(The bottom level contains roughly N/2 nodes)

        O
      /   \
     O     O    <-- Queue holds 2
    / \   / \
   O   O O   O  <-- Queue holds 4
  / \ / \ / \ / \
 O  O O O O O O O <-- Queue holds 8 (Max Width)

Max Width approx = N / 2
```

The queue grows proportional to the maximum width of the tree. **Total Space = O(N)**.

---

4. Solution Code

We will write clean, production-grade code.

Python Solution *Using `collections.deque` for efficient BFS.*

```python
from collections import deque
```

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def zigzagLevelOrder(root):
    """
    Performs a zigzag level order traversal on a binary tree.
    """
    # Base Case: Handle empty tree input
    if not root:
        return []

    results = []

    # Initialize Queue with the root
    queue = deque([root])

    # Direction Flag: True means Left -> Right, False means Right -> Left
    left_to_right = True

    while queue:
        level_size = len(queue)
        current_level_values = deque() # Use deque for O(1) head/tail insertions

        for _ in range(level_size):
            node = queue.popleft()

            # LOGIC FOR ZIGZAG:
            if left_to_right:
                # Normal order: Add to the end (tail)
                current_level_values.append(node.val)
            else:
                # Reverse order: Add to the front (head)
                current_level_values.appendleft(node.val)

            # Standard BFS: Add children to queue
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        # Convert deque to list and add to results
        results.append(list(current_level_values))
        # Toggle direction for the next level
        left_to_right = not left_to_right

    return results
```

JavaScript Solution *Using standard Arrays. Since JS arrays* **`unshift`** *(add to front) is technically O(N), for strict interviews we might prefer reversing a standard array or pre-allocating, but* **`unshift`** *is acceptable for readability in most JS interviews unless the interviewer drills into optimization.*

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 * this.val = (val===undefined ? 0 : val)
 * this.left = (left===undefined ? null : left)
 * this.right = (right===undefined ? null : right)
 * }
 */

/**
 * @param {TreeNode} root
 * @return {number[][]}
 */
var zigzagLevelOrder = function(root) {
    if (!root) return [];

    const results = [];
    const queue = [root]; // Using array as queue
    let leftToRight = true;

    while (queue.length > 0) {
        const levelSize = queue.length;
        const currentLevelValues = [];

        for (let i = 0; i < levelSize; i++) {
            // Shift removes the first element (FIFO)
            const node = queue.shift();

            // LOGIC FOR ZIGZAG:
            if (leftToRight) {
                // Normal order: push to end
                currentLevelValues.push(node.val);
            } else {
                // Reverse order: unshift to front
                // (Or push and reverse the array later to optimize speed)
                currentLevelValues.unshift(node.val);
            }

            if (node.left) queue.push(node.left);
            if (node.right) queue.push(node.right);
        }

        results.push(currentLevelValues);
        // Flip direction
        leftToRight = !leftToRight;
    }

    return results;
};
```

New Terms & Techniques

1. BFS (Breadth-First Search): An algorithm for traversing tree or graph data structures. It starts at the tree root and explores all neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

- **Why it helps here:** It naturally processes the tree row-by-row, which is exactly how the output needs to be structured.

2. Deque (Double-Ended Queue): A data structure that allows adding and removing elements from both the front and the back in O(1) time.

- **Why it helps here:** In Python, `list.insert(0, x)` shifts all elements, costing O(N). `deque.appendleft(x)` costs O(1). This keeps our algorithm efficient.

Real World / Interview Variations

Why do companies like Google, Meta, and Bloomberg ask this? It's rarely about actual binary trees. It's about **manipulating data traversal orders**.

1. **UI/UX Rendering (The "Feed" Problem):**

- *Question:* "Design a news feed where posts alternate between 'friend updates' (left aligned) and 'sponsored content' (right aligned) based on relevance scores."
- *Connection:* This requires processing a stream of data and rendering it in alternating layouts.

2. **Network Broadcasting:**

- *Question:* "Simulate a network packet broadcast where the signal bounces back and forth between server racks arranged in a grid."

- *Connection:* This is a physical mapping of the zigzag traversal.

3. **File System Visualization:**

- *Question:* "Print a file directory tree. However, sort the files in ascending order for even depths, and descending order for odd depths to optimize visual scanning."

4. **Physical Warehouse Picking (Amazon/Logistics):**

- *Question:* "A robot needs to pick items from aisles. To save travel time, it goes down Aisle 1 (North to South), then Aisle 2 (South to North), then Aisle 3 (North to South)."
- *Connection:* This is exactly Zigzag traversal (Snake-like sorting) to minimize robot movement distance.

1110. Delete Nodes And Return Forest

A Google L5 or L6 engineer approaches a problem like this by focusing on **clean state management** and **identifying the recursive invariant**. They don't just "solve" it; they look for a solution that handles edge cases (like the root being deleted) naturally without messy `if-else` blocks.

---

1. Problem Explanation

You are given the root of a binary tree and a list of values to delete. After deleting the nodes specified in the list, you are left with several disjoint trees (a "forest"). Your goal is to return a list of the roots of every tree in this forest.

The Core Challenges

- **Disconnection:** When you delete a node, its children become potential new roots.
- **Parent-Child Linkage:** A parent needs to know if its child was deleted so it can set its `left` or `right` pointer to `null`.
- **Root Management:** The original root of the tree might also be deleted.

---

2. Solution Explanation

The most elegant way to solve this is using **Post-Order Traversal (Bottom-Up)**.

Why Bottom-Up?

If we process a node *after* its children, the children can tell the parent, "Hey, I've been deleted, set your pointer to null." This prevents "dangling pointers."

The Logic (The "L5" Way)

1. Convert the `to_delete` list into a **Set** for constant time lookups.
2. Define a recursive function `dfs(node, is_root)`.
3. If a node `is_root` and it is **not** being deleted, add it to our result list.
4. A node's children become "roots" if the current node is deleted.

Step-by-Step Visualization

Imagine this tree: `to_delete = [3, 5]`

```
      1
     / \
    2   3
   / \   \
  4   5   6
```

Step 1: Traverse down to the leaves (4, 5, 6).

- **Node 4:** Not in `to_delete`. Returns itself to parent (2).
- **Node 5:** In `to_delete`! We return `null` to parent (2). Since it's deleted, its children (none) would have been potential roots.
- **Node 6:** Not in `to_delete`. However, its parent (3) is getting deleted. So 6 becomes a new root.

Step 2: Process intermediate nodes.

- **Node 2:** Parent of 4 and 5. It receives `4` from the left and `null` from the right. It updates its pointers.
- **Node 3:** In `to_delete`! It returns `null` to parent (1). Its child (6) was flagged to be added to the result.

The Final Forest:

```
Forest Result: [Root 1, Root 6]

Tree 1:       Tree 2:
   1            6
  /
 2
/
4
```

The “Is Root” Logic Diagram

We pass a boolean flag `is_root` down the tree. A node is a “potential root” if its parent was just deleted OR if it is the original root of the entire structure.

```
Recursion Flow:
---------------------------------------------------------
Node (Value) | to_delete? | is_root? | Action
---------------------------------------------------------
1            | No         | True     | Add to result!
3            | Yes        | False    | Don't add. Tell children they are roots.
6            | No         | True     | Add to result!
---------------------------------------------------------
```

---

3. Time and Space Complexity Analysis

An L6 engineer would derive complexity by looking at the work done per node and the storage requirements for the output.

Time Complexity (TC)

We visit every node exactly once. Inside each visit, we do a lookup in the Hash Set.

```
TC Derivation:
Nodes visited (N) * Work per node (O(1) lookup + pointer swap)
= N * 1
= O(N)
```

Space Complexity (SC)

We store the `to_delete` set, the recursion stack, and the result list.

```
SC Derivation:
- Set storage: O(D) where D is number of nodes to delete.
- Recursion Stack: O(H) where H is tree height (Worst case O(N) for skewed tree).
- Result List: O(N) in worst case (e.g., deleting every other node).
Total = O(N + D)
Since D <= N, total simplified = O(N)
```

---

4. Solution Code

Python Snippet

```python
class Solution:
    def delNodes(self, root, to_delete):
        # O(1) average lookup time is crucial
        delete_set = set(to_delete)
        forest = []

        # The helper returns the node itself if it stays,
        # or None if it is deleted.
        def dfs(node, is_root):
            if not node:
                return None

            # If this node is a root and not slated for deletion,
            # it's a root of a tree in our final forest.
            is_deleted = node.val in delete_set
            if is_root and not is_deleted:
                forest.append(node)

            # Post-order style: process children first,
            # then update the current node's links.
            # If the current node is deleted, its children
            # will be 'is_root = True' for the next level.
            node.left = dfs(node.left, is_deleted)
            node.right = dfs(node.right, is_deleted)

            return None if is_deleted else node

        dfs(root, True)
        return forest
```

JavaScript Snippet

```javascript
var delNodes = function(root, to_delete) {
    const deleteSet = new Set(to_delete);
    const forest = [];

    /**
     * @param {TreeNode} node - Current node being processed
     * @param {boolean} isRoot - Whether this node is a potential new root
     * @returns {TreeNode|null} - The node if it should remain attached to parent
     */
    function dfs(node, isRoot) {
        if (!node) return null;

        const isDeleted = deleteSet.has(node.val);

        // If it's a root and staying, add to our list
        if (isRoot && !isDeleted) {
            forest.push(node);
        }

        // Reassign children: if current node is deleted,
        // children effectively become potential roots.
        node.left = dfs(node.left, isDeleted);
        node.right = dfs(node.right, isDeleted);

        return isDeleted ? null : node;
    }

    dfs(root, true);
    return forest;
};
```

Key Terms & Real-World Context

Algorithm Technique: Post-Order Traversal

- **What is it?** A way of visiting a tree where you visit the Left child, then the Right child, then the Node itself.
- **Why it helps here:** It allows a "bottom-up" cleanup. By the time we decide if a node is deleted, its children have already been processed and potentially added to the forest. It makes managing the parent's `left` and `right` pointers very simple.

Real-World / Interview Variations

Companies like Google and Meta rarely ask this exact Leetcode problem anymore; they use "flavored" versions:

1. **Google (Access Control):** "You have a directory tree. If a user loses permission to a folder, they might still have permission to sub-folders. Return all 'entry points' the user still has."
2. **Meta (Social Graph):** "A user deactivates their account. You need to remove them from a hierarchy but keep the sub-groups they managed as independent units. How do you re-map the organizational structure?"
3. **Bloomberg (Data Streaming):** "Given a stream of financial instrument dependencies, if a 'source' node fails, identify all downstream 'islands' that are now isolated but still functional."

2096. Step-By-Step Directions From a Binary Tree Node to Another

A senior-level (L5/L6) approach to a problem like this isn't just about getting the code to pass; it's about identifying the most efficient pattern, handling edge cases gracefully, and ensuring the logic is readable.

1. Problem Explanation

You are given a binary tree. You need to find the shortest path from a **start node** to a **destination node**.

- Going **Up** to a parent is denoted by **'U'**.
- Going **Down** to a left child is denoted by **'L'**.
- Going **Down** to a right child is denoted by **'R'**.

The Core Challenge

In a typical binary tree, nodes only have pointers to their children, not their parents. This makes moving "Up" tricky. A naive approach might involve converting the tree into a graph (adding parent pointers), but a top-tier engineer looks for a solution that avoids extra memory overhead and stays within the tree structure.

---

2. Solution Explanation

The most elegant solution relies on the concept of the **Lowest Common Ancestor (LCA)**. The LCA is the deepest node in the tree that is an ancestor to both the start and destination nodes.

The Algorithm Strategy

1. **Find the paths from the Root:** Find the path from the Root to the `startValue` and the path from the Root to the `destValue`.
2. **Find the LCA:** Compare these two paths. The point where they diverge is the LCA.
3. **Construct the Path:**

- Any steps from the `startValue` back up to the LCA must be **'U'**.
- Any steps from the LCA down to the `destValue` are taken directly from the remaining portion of the destination path.

Visualizing the Logic

Imagine this tree:

```
        1 (Root)
       /  \
      3    4
     / \    \
    5   2    6
```

Task: Find path from **5** to **6**.

Step 1: Get paths from Root

- Path to 5: Root -> 3 -> 5 (Directions: **L, L**)
- Path to 6: Root -> 4 -> 6 (Directions: **R, R**)

Step 2: Find Divergence (LCA)

- Path 1: [L, L]
- Path 2: [R, R]
- They diverge immediately at the Root. LCA = 1.

Step 3: Build Result

- Convert all steps in Path 1 to 'U': **U, U**
- Keep Path 2 as is: **R, R**
- Result: **UURR**

Example 2: When LCA is one of the nodes

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

Task: Path from **2** to **1**.

- Path to 2: [L, R]
- Path to 1: [] (It's the root)
- Divergence: Immediately.
- Convert Path to 2 into 'U's: **U, U**
- Result: **UU**

---

3. Time and Space Complexity Analysis

A Google L6 would derive complexity by looking at the number of times we touch each node and the memory required to store the paths.

Time Complexity (TC)

We traverse the tree to find the two paths. In the worst case, we visit every node once.

```
TC Derivation:
O(N) - Find path to Start
O(N) - Find path to Destination
O(N) - String manipulation/Comparison
--------------------------------------
Total: O(N) where N is the number of nodes.
```

Space Complexity (SC)

We store the paths from the root to the nodes. In a skewed (unbalanced) tree, the path length can be N.

```
SC Derivation:
O(H) - Recursion stack (H = height of tree)
O(H) - Memory for Start path
O(H) - Memory for Destination path
--------------------------------------
Total: O(H). In worst case (skewed tree), O(N).
```

---

4. Solution Code

Python Snippet

```python
class Solution:
    def getDirections(self, root, startValue, destValue):
        # Helper to find path from root to a specific value
        def find_path(node, target, path):
            if node.val == target:
                return True

            # Try Left
            if node.left:
                path.append('L')
                if find_path(node.left, target, path):
                    return True
                path.pop() # Backtrack

            # Try Right
            if node.right:
                path.append('R')
                if find_path(node.right, target, path):
```

```
                return True
            path.pop() # Backtrack

        return False

    start_path = []
    dest_path = []

    find_path(root, startValue, start_path)
    find_path(root, destValue, dest_path)

    # Remove common prefix (finding the LCA point)
    i = 0
    while i < len(start_path) and i < len(dest_path) and start_path[i] == dest_path[i]:
        i += 1

    # Steps from start up to LCA are all 'U'
    up_steps = "U" * (len(start_path) - i)
    # Steps from LCA down to dest are the remaining dest_path
    down_steps = "".join(dest_path[i:])

    return up_steps + down_steps
```

Javascript Snippet

```
var getDirections = function(root, startValue, destValue) {
    const findPath = (node, target, path) => {
        if (!node) return false;
        if (node.val === target) return true;

        // DFS with Backtracking to record path
        path.push('L');
        if (findPath(node.left, target, path)) return true;
        path.pop();

        path.push('R');
        if (findPath(node.right, target, path)) return true;
        path.pop();

        return false;
    };

    let startPath = [];
    let destPath = [];

    findPath(root, startValue, startPath);
    findPath(root, destValue, destPath);

    let i = 0;
    // Find where the paths diverge
    while (i < startPath.length && i < destPath.length && startPath[i] === destPath[i]) {
        i++;
    }

    // Convert remaining start path segments to 'U'
    let upSteps = 'U'.repeat(startPath.length - i);
    // Join the remaining destination path segments
    let downSteps = destPath.slice(i).join('');

    return upSteps + downSteps;
};
```

Terminology & Techniques

- **Lowest Common Ancestor (LCA):** The shared ancestor farthest from the root. In this problem, it acts as the "pivot point" where you stop going up and start going down.
- **Backtracking:** A technique where you explore a path (add 'L' to an array) and, if it doesn't lead to the target, you remove it (`path.pop()`) before trying the next branch. It keeps memory usage low compared to copying strings at every node.

Real-World / Interview Variations

Companies like **Google** and **Meta** rarely ask the "Leetcoded" version directly. Instead, they frame it as:

1. **Corporate Hierarchy (Google):** "Given an org chart, find the shortest sequence of emails/introductions needed for Employee A to reach Employee B." (Up to managers, then down to reports).
2. **Network Routing (Bloomberg):** "In a hierarchical network of routers, find the path a packet must take from a legacy node to a new server node."
3. **File Systems (Meta):** "Find the relative path between two files in a directory tree (e.g., `../../folder/file.txt`)." This is logically identical to the 'U', 'L', 'R' problem.

987. Vertical Order Traversal of a Binary Tree

A Google L5 or L6 engineer doesn't just look for *a* solution; they look for the most robust, maintainable, and efficient one. They prioritize handling edge cases (like nodes at the same coordinates) and ensure the data structures used are appropriate for the scale.

---

1. Problem Explanation

The goal is to report the "Vertical Order" of a binary tree. Imagine the tree on a 2D grid:

- The **Root** is at row 0, column 0.
- A **Left Child** is at (row + 1, col - 1).
- A **Right Child** is at (row + 1, col + 1).

We need to group nodes by their **column index** from left to right. If two nodes are in the same column, we sort them by their **row index** (top to bottom). If they are in the same row *and* same column, we sort them by their **value** (ascending).

---

2. Solution Explanation

To solve this, we need a way to "tag" every node with its coordinates and then sort those tags.

Step-by-Step Logic

1. **Traverse the Tree:** Use Breadth-First Search (BFS) or Depth-First Search (DFS) to visit every node. BFS is often preferred here because it naturally processes rows in order.
2. **Assign Coordinates:** As we visit, we store a tuple: `(column, row, value)`.
3. **Sort the Data:** This is the "Google-tier" part. We sort primarily by `column`, then `row`, then `value`.
4. **Group by Column:** Iterate through the sorted list and group values that share the same column into sub-lists.

Visualization with ASCII

Example Tree:

```
    3
   / \
  9   20
```

```
       /  \
      15   7
```

Mapping Coordinates:

```
Node(3):  row 0, col 0  -> (0, 0, 3)
Node(9):  row 1, col -1 -> (-1, 1, 9)
Node(20): row 1, col 1  -> (1, 1, 20)
Node(15): row 2, col 0  -> (0, 2, 15)
Node(7):  row 2, col 2  -> (2, 2, 7)
```

Visual Grid Representation:

```
Col:    -1      0      1      2
---------------------------------
Row 0:  .       3      .      .
Row 1:  9       .      20     .
Row 2:  .      15      .      7
```

Sorting the coordinates: List before sorting: `[(0,0,3), (-1,1,9), (1,1,20), (0,2,15), (2,2,7)]` List after sorting: `[(-1,1,9), (0,0,3), (0,2,15), (1,1,20), (2,2,7)]`

Final Grouping (Result): `[[9], [3, 15], [20], [7]]`

---

3. Time and Space Complexity Analysis

An L6 engineer would explain the complexity by breaking down the operations:

Time Complexity Derivation:

Operation	Complexity	Reason
Traversal (BFS/DFS)	O(N)	Visit every node once.
Sorting	O(N log N)	Sorting N coordinate tuples.
Grouping	O(N)	One linear pass through sorted list.
Total Time	O(N log N)	Dominated by the sorting step.

Space Complexity Derivation:

Data Structure	Complexity	Reason
Coordinate List	O(N)	Stores (col, row, val) for N nodes.
Recursion/Queue	O(N)	For tree traversal (worst case).
Final Result List	O(N)	Storing all node values.
Total Space	O(N)	Everything scales linearly with N.

---

4. Solution Code

Python Snippet

```python
from collections import deque

def verticalTraversal(root):
    # node_data will store tuples of (column, row, value)
    node_data = []

    # BFS traversal to assign coordinates
    # Queue stores (node, row, column)
    queue = deque([(root, 0, 0)])

    while queue:
        node, row, col = queue.popleft()
        if node:
            node_data.append((col, row, node.val))
            queue.append((node.left, row + 1, col - 1))
            queue.append((node.right, row + 1, col + 1))

    # Sort by column, then row, then value
    # This ensures the specific ordering requirements are met
    node_data.sort()

    result = []
    if not node_data:
        return result

    # Grouping logic: Collect values that share the same column
    current_col_index = node_data[0][0]
    current_col_nodes = []

    for col, row, val in node_data:
        if col == current_col_index:
            current_col_nodes.append(val)
        else:
            result.append(current_col_nodes)
            current_col_index = col
            current_col_nodes = [val]

    result.append(current_col_nodes)
    return result
```

Javascript Snippet

```javascript
var verticalTraversal = function(root) {
    const nodeData = [];
    const queue = [[root, 0, 0]]; // [node, row, col]

    // Standard BFS to collect coordinates
    while (queue.length > 0) {
        const [node, row, col] = queue.shift();
        if (node) {
            nodeData.push({ col, row, val: node.val });
            queue.push([node.left, row + 1, col - 1]);
            queue.push([node.right, row + 1, col + 1]);
        }
    }

    // Sort: Primary: col, Secondary: row, Tertiary: value
    nodeData.sort((a, b) => {
        if (a.col !== b.col) return a.col - b.col;
        if (a.row !== b.row) return a.row - b.row;
        return a.val - b.val;
    });

    const result = [];
    let lastCol = null;

    // Build the final array of arrays
    for (const { col, row, val } of nodeData) {
        if (col !== lastCol) {
            result.push([]);
            lastCol = col;
        }
        result[result.length - 1].push(val);
    }

    return result;
};
```

Terms and Indirect Real-World Problems

Key Term: Coordinate Transformation

In this problem, we mapped a hierarchical structure (Tree) into a 2D Cartesian plane (x, y). This is a common technique in computer graphics and UI layout engines (like the Chrome rendering engine) to determine where elements should overlap or align.

Real-World Interview Variations

- **Google:** "Design a spreadsheet rendering engine." You need to manage which cells appear in which columns and handle overlapping content—essentially vertical traversal.
- **Meta:** "Find the 'Shadow' of a 3D object." This is often framed as projecting nodes onto a 1D line, which is exactly what grouping by column index does.
- **Bloomberg:** "Top view of a binary tree." This is a subset of vertical order traversal where you only take the *first* node (minimum row) for every unique column.

863. All Nodes Distance K in Binary Tree

A Google L5/L6 engineer doesn't just look for a solution that "passes"; they look for a solution that is **robust, readable, and handles edge cases gracefully.** For this problem, they would recognize that while a tree is a specific structure, the "Distance K" requirement treats it like a generic undirected graph.

---

1. Problem Explanation

Imagine you are standing at a specific house (the `target` node) in a neighborhood (the Binary Tree). You want to find all houses that are exactly `K` blocks away from you.

The catch? In a standard Binary Tree, you can only see your "children" (the houses below you). You can't naturally see your "parent" (the house you came from). To find houses `K` distance away, you might need to go **up** to your parent, then **down** into a different branch.

The Goal: Return a list of all node values that are exactly distance `K` from the target.

---

2. Solution Explanation

To solve this like a Senior Engineer, we break it into two logical phases:

Phase A: Adding "Backlinks" (Building the Graph)

Since we need to move upward, we first traverse the tree and give every node a pointer to its parent. This effectively turns our directed tree into an **undirected graph**.

```
Original Tree:                 Graph (with Parent Pointers):
    1                               1 <---> null
   / \                             / \
  2   3        ----->             2   3
 /                               /     \
4                               4 <---> 2
```

Phase B: Breadth-First Search (BFS)

Starting from the `target`, we move outward layer by layer (like a ripple in a pond).

- **Layer 0:** The target itself.
- **Layer 1:** Neighbors of target (Left child, Right child, and Parent).
- **Layer 2:** Neighbors of those neighbors… and so on.

Once we reach the `K-th` layer, those nodes are our answer.

Step-by-Step Visualization

Let's use an example: `Tree = [3,5,1,6,2,0,8,null,null,7,4]`, `Target = 5`, `K = 2`.

1. The Structure

```
        (3)
       /   \
    [5]     (1)
   /   \    / \
(6)   (2) (0) (8)
      /   \
   (7)   (4)
```

2. BFS Traversal from Target (5)

- **Distance 0:** Starting at `[5]`.
- *Visited:* `{5}`
- **Distance 1:** Move to neighbors of `5`.
- Neighbors are: `6` (left), `2` (right), and `3` (parent).
- *Visited:* `{5, 6, 2, 3}`
- **Distance 2:** Move to neighbors of `6`, `2`, and `3`.
- From `6`: No new neighbors.
- From `2`: Neighbors are `7` and `4`.
- From `3`: Neighbor is `1` (its other child).
- *Result:* `[7, 4, 1]`

3. Visualizing the "Ripple"

```
Distance 0:          (5)
                      |
Distance 1:      (6)--(5)--(2)
                      |
                     (3)
                      |
Distance 2:  (7)--(2)--(4)  (3)--(1)
```

---

3. Time and Space Complexity Analysis

An L5+ engineer must demonstrate why the algorithm is efficient.

Time Complexity (TC)

We visit every node twice: once to map parents, and once (at most) during the BFS.

```
TC Derivation:
O(N) : To traverse N nodes and build the parent map.
+
O(N) : To perform BFS (visiting each node/edge once).
------------------------------------------------------
Total: O(N) where N is the number of nodes in the tree.
```

Space Complexity (SC)

We store the parent map and the queue for BFS.

```
SC Derivation:
O(N) : To store parent pointers for N nodes in a Map.
O(N) : To store visited nodes in a Set to avoid cycles.
O(N) : The Queue for BFS in the worst case (a flat tree).
------------------------------------------------------
Total: O(N)
```

---

4. Solution Code

Python Implementation

```python
from collections import deque

class Solution:
    def distanceK(self, root, target, k):
        # We need to traverse upwards, so we map each node to its parent
        parent_map = {}

        # Simple DFS to populate the parent_map
        def build_parents(node, p=None):
            if node:
                parent_map[node] = p
                build_parents(node.left, node)
                build_parents(node.right, node)

        build_parents(root)

        # BFS starting from the target
        # queue stores (current_node, current_distance)
        queue = deque([(target, 0)])
        visited = {target}
        result = []

        while queue:
            node, dist = queue.popleft()

            if dist == k:
                result.append(node.val)
            elif dist < k:
                # Check all 3 possible directions: Left, Right, and Up (Parent)
                for neighbor in [node.left, node.right, parent_map[node]]:
                    if neighbor and neighbor not in visited:
                        visited.add(neighbor)
                        queue.append((neighbor, dist + 1))

        return result
```

JavaScript Implementation

```
/**
 * Main function to find all nodes at distance K
 */
var distanceK = function(root, target, k) {
    const parentMap = new Map();

    /**
     * Helper: Depth First Search to map children to their parents
     * This turns the tree into a searchable graph.
     */
    const buildParents = (node, parent) => {
        if (!node) return;
        parentMap.set(node, parent);
        buildParents(node.left, node);
        buildParents(node.right, node);
    };

    buildParents(root, null);

    // BFS setup
    const queue = [[target, 0]];
    const visited = new Set([target]);
    const result = [];

    while (queue.length > 0) {
        const [node, dist] = queue.shift();

        if (dist === k) {
            result.push(node.val);
            continue; // No need to check neighbors if we are at distance K
        }

        // Potential neighbors: Left child, Right child, Parent
        const neighbors = [node.left, node.right, parentMap.get(node)];

        for (const neighbor of neighbors) {
            if (neighbor && !visited.has(neighbor)) {
                visited.add(neighbor);
                queue.push([neighbor, dist + 1]);
            }
        }
    }

    return result;
};
```

Technical Terms & Context

- **BFS (Breadth-First Search):** An algorithm for searching a graph layer by layer. It is perfect here because "Distance K" is a shortest-path problem in an unweighted graph.
- **Parent Mapping:** The process of adding a reference from a child node back to its parent. It bridges the gap between a **Tree** (one-way) and a **Graph** (two-way).
- **Visited Set:** Crucial when converting a tree to a graph. Without it, the BFS would move back and forth between a child and parent forever

(infinite loop).

Real-World / Interview Variations

Companies like Google and Meta rarely ask the “raw” Leetcode version. They often wrap it in a scenario:

1. **Social Networks (Meta):** “Given a user (target) in a friend-graph (tree-like structure), find all friends of friends who are exactly 3 degrees away.”
2. **Infrastructure/Routing (Google):** “You have a network of data centers connected in a hierarchy. If one center fails, find all centers within a latency radius of K to reroute traffic.”
3. **Bloomberg (Terminal Data):** “In a corporate ownership tree, find all ‘cousin’ companies that are exactly K steps away from a specific subsidiary.”
4. **Infection Spread:** “Given a person who just tested positive in a contact-tracing tree, find all people exactly K contacts away to notify them.”

99. Recover Binary Search Tree

A top-tier engineer at Google or Meta doesn’t just look for a “working” solution; they look for the most efficient use of resources. For this problem, while a standard DFS approach is common, an L6 (Staff Engineer) would immediately identify that we can solve this in **O(1) space** using **Morris Traversal**.

Here is how a top-of-the-band engineer breaks this down.

---

1. Problem Explanation

In a **Binary Search Tree (BST)**, the values must follow a strict order: for any node, all values in the left subtree are smaller, and all values in the right subtree are larger.

If you perform an **In-order Traversal** (Left -> Root -> Right) on a valid BST, the resulting sequence of numbers must be **sorted in ascending order**.

The Twist

Exactly two nodes in the BST have been swapped by mistake. Our job is to find those two nodes and swap their values back to recover the tree structure, without changing its physical shape.

Example: Input: `[3, 1, 4, null, null, 2]`

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

In-order traversal: `1, 3, 2, 4` Notice `3` and `2` are out of place. If we swap them, the traversal becomes `1, 2, 3, 4` (sorted!).

---

2. Solution Explanation

To solve this like a Staff Engineer, we use the **Morris Traversal** algorithm to achieve constant space.

The Core Logic: Identifying the Swapped Nodes

As we traverse the tree in-order, we keep track of the `previous` node we visited. We compare `previous.val` with `current.val`.

1. **First Violation:** The first time `prev.val > current.val`, the **first** swapped node is `prev`.
2. **Second Violation:** The second time `prev.val > current.val`, the **second** swapped node is `current`.

Wait, what if the swapped nodes are adjacent? If the traversal is `1, 3, 2, 4`, we only see one violation (`3 > 2`). In this case, `first` is `3` and `second` is `2`.

Visualization: Finding the Violations

Let's look at a swapped BST where nodes `10` and `4` are swapped. Expected In-order: `2, 4, 6, 8, 10, 12` Actual In-order: `2, 10, 6, 8, 4, 12`

```
Step 1: Compare 2 and 10 -> (2 < 10) OK.
Step 2: Compare 10 and 6  -> (10 > 6) VIOLATION 1!
        First = 10, Second = 6.
Step 3: Compare 6 and 8   -> (6 < 8) OK.
Step 4: Compare 8 and 4   -> (8 > 4) VIOLATION 2!
        Second = 4.
Final: Swap 10 and 4.
```

The Secret Weapon: Morris Traversal

Normally, in-order traversal uses recursion or a stack, which takes **O(Height)** space. Morris Traversal uses "threads" (temporary pointers) to link a node's predecessor back to itself, allowing us to walk back up the tree without a stack.

How it works (The ASCII View):

Suppose we are at Node `8`. Its left child is `4`. To visit `8` after its left subtree, we find the **rightmost** node in the left subtree (the predecessor) and point its `right` pointer to `8`.

ASCII Step-by-Step:

```
Initial:
      8
     /
    4
     \
      6

1. At 8: Find predecessor of 8 (which is 6).
2. Set 6.right = 8 (The Thread).
3. Move to 4 (left).

Modified Tree (Temporary):
      8 <----
     /       |
    4        |
     \       |
      6 -----|

4. At 4: Predecessor is null. Visit 4. Move to 6 (right).
5. At 6: Predecessor's right is already 8.
   - This means we finished the left side!
   - Visit 8.
   - Remove thread (6.right = null).
   - Move to 8.right.
```

---

3. Time and Space Complexity Analysis

The complexity is derived by observing how many times we visit each node and what extra memory we allocate.

Time Complexity (TC)

```
+-----------------------------------------------------------+
```

Operation	Complexity
Finding Predecessors	Each edge visited at most 3 times (find, link, break)
Total Node Visits	O(N)
Final Swap	O(1)
OVERALL TC	O(N)

Space Complexity (SC)

Component	Memory Usage
Recursion Stack	None (0)
Iterative Stack	None (0)
Pointers (first, second, etc)	Constant (O(1))
OVERALL SC	O(1)

---

4. Solution Code

Python Snippet

```python
class Solution:
    def recoverTree(self, root):
        # Tracking the two misplaced nodes and the previous node in traversal
        first = second = prev = None
        curr = root

        while curr:
            if curr.left is None:
                # Standard In-order Visit logic
                if prev and prev.val > curr.val:
                    if not first: first = prev
                    second = curr
                prev = curr
                curr = curr.right
            else:
                # Find the inorder predecessor (rightmost node in left subtree)
                predecessor = curr.left
                while predecessor.right and predecessor.right != curr:
                    predecessor = predecessor.right

                if predecessor.right is None:
                    # Create a temporary thread to the current node
                    predecessor.right = curr
                    curr = curr.left
                else:
                    # Thread already exists, meaning we finished the left subtree
                    predecessor.right = None
                    if prev and prev.val > curr.val:
                        if not first: first = prev
                        second = curr
                    prev = curr
                    curr = curr.right

        # Swap the values of the two nodes found
        if first and second:
            first.val, second.val = second.val, first.val
```

Javascript Snippet

```
var recoverTree = function(root) {
    let first = null, second = null, prev = null;
    let curr = root;

    while (curr !== null) {
        if (curr.left === null) {
            // Check for order violation
            if (prev && prev.val > curr.val) {
                if (!first) first = prev;
                second = curr;
            }
            prev = curr;
            curr = curr.right;
        } else {
            // Find predecessor to create/break Morris threads
            let pred = curr.left;
            while (pred.right !== null && pred.right !== curr) {
                pred = pred.right;
            }

            if (pred.right === null) {
                pred.right = curr; // Establish thread
                curr = curr.left;
            } else {
                pred.right = null; // Break thread
                if (prev && prev.val > curr.val) {
                    if (!first) first = prev;
                    second = curr;
                }
                prev = curr;
                curr = curr.right;
            }
        }
    }

    // Final swap of values
    let temp = first.val;
    first.val = second.val;
    second.val = temp;
};
```

Note 1: Terms & Techniques

- **Morris Traversal:** An algorithm for in-order tree traversal that achieves O(1) space by temporarily modifying the tree (adding "threads"). It's helpful here because the "Follow-up" for this Leetcode problem specifically asks for O(1) space, which is the "bar" for senior engineering roles.
- **In-order Predecessor:** The node that comes immediately before the current node in an in-order sequence. In Morris Traversal, we use this to know where to "return" to after finishing a left subtree.

Note 2: Real-World / Indirect Interview Variants

Companies like Google and Bloomberg often wrap this "detect order violation" logic in different contexts:

1. **Data Validation:** "You have a stream of timestamps from different servers that should be sorted. Two servers had clock drift and swapped their logs. Identify the logs."
2. **UI/DOM Trees:** "Two elements in a sorted sidebar list were dragged and dropped into the wrong positions. Write a script to identify them using the least amount of memory possible."
3. **Broken Index:** "A database B-Tree index has been corrupted such that two pointers are swapped. How do you detect and fix the keys without rebuilding the entire index?"

173. Binary Search Tree Iterator

A top-tier engineer at Google or Meta doesn't just solve this problem to pass the test cases; they solve it for **production-grade efficiency**. The core challenge of the BST Iterator is balancing the trade-off between "doing all the work upfront" versus "doing it on demand."

An L5/L6 engineer will immediately identify that the goal is to achieve **O(1) average time** for `next()` and **O(h)** space complexity (where h is the height of the tree), rather than O(n) space.

---

1. Problem Explanation

The goal is to implement an iterator over a Binary Search Tree (BST). A BST is a tree where for every node:

- Everything in the **left** subtree is smaller.
- Everything in the **right** subtree is larger.

An "iterator" needs two main functions:

1. **next()**: Returns the next smallest number in the BST.
2. **hasNext()**: Returns true if there are still numbers left to visit.

The "catch" is that you must return these numbers in **ascending order** (In-order traversal), and you should try to be as efficient as possible with memory. You shouldn't just flatten the whole tree into an array at the start, because if the tree has 1 billion nodes and you only call `next()` twice, you've wasted massive amounts of memory and time.

---

2. Solution Explanation

To solve this efficiently, we use a **Controlled Stack**. Instead of a full recursion, we use a stack to simulate the recursion "one step at a time."

The Logic: "Go Left as Far as Possible"

In a BST, the smallest element is always the leftmost node. To get the next smallest, we:

1. Push all nodes from the root down to the leftmost leaf onto a stack.
2. When `next()` is called, we pop from the stack (this is our current smallest).
3. If that popped node has a **right child**, we must visit that child and its entire "leftmost descent" because those values are smaller than the parent of the node we just popped.

ASCII Visualization

Imagine this BST:

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

Step 1: Initialization We push the root (7) and all its left children. Stack: `[7, 3]` (Top is on the right)

Step 2: Calling `next()`

- Pop `3`.
- Does `3` have a right child? No.
- **Result: 3**.
- Stack: `[7]`

Step 3: Calling `next()`

- Pop `7`.
- Does `7` have a right child? Yes (15).
- We must process `15` by pushing it and all its left children.
- Push `15`, then push `9`.
- **Result: 7**.
- Stack: `[15, 9]`

Step 4: Calling `next()`

- Pop `9`.
- Does `9` have a right child? No.
- **Result: 9**.
- Stack: `[15]`

Step 5: Calling `next()`

- Pop `15`.
- Does `15` have a right child? Yes (20).
- Push `20`.
- **Result: 15**.
- Stack: `[20]`

Step 6: Calling `next()`

- Pop `20`.
- **Result: 20**.
- Stack: `[]` (Empty)

---

3. Time and Space Complexity Analysis

```
TIME COMPLEXITY (TC)
----------------------------------------------------------------
Function   | Complexity | Logic
----------------------------------------------------------------
hasNext()  | O(1)       | Just checks if the stack is empty.
next()     | O(1) avg   | While one call might trigger a "push"
           |            | of 'h' nodes, each node is pushed and
           |            | popped exactly ONCE across the entire
           |            | life of the iterator.
           |            | Total work: O(N) for N nodes.
           |            | Average work: O(N)/N = O(1).
----------------------------------------------------------------

SPACE COMPLEXITY (SC)
----------------------------------------------------------------
Complexity | Logic
----------------------------------------------------------------
O(h)       | We only store nodes in the stack up to the
           | height of the tree (h). In a balanced tree,
           | this is O(log n). In the worst case (a line),
           | it is O(n). This is much better than O(n)
           | for all cases.
----------------------------------------------------------------
```

---

4. Solution Code

Python

```python
class BSTIterator:
    def __init__(self, root):
        # The stack will hold the path of nodes we are currently exploring
        self.stack = []
        # We initialize by pushing all left-side nodes of the root
        self._push_left(root)

    # This helper function ensures we always find the next smallest
    # by diving as deep left as possible from a given node.
    def _push_left(self, node):
        while node:
            self.stack.append(node)
            node = node.left

    def next(self) -> int:
        # The top of the stack is the next smallest element
        top_node = self.stack.pop()

        # If the node has a right child, we must process that subtree
        if top_node.right:
            self._push_left(top_node.right)

        return top_node.val

    def hasNext(self) -> bool:
        # If the stack is not empty, there are more nodes to visit
        return len(self.stack) > 0
```

JavaScript

```javascript
class BSTIterator {
    constructor(root) {
        this.stack = [];
        this._pushLeft(root);
    }

    /**
     * Helper to push all left children of a node onto the stack.
     * This mimics the first part of an in-order traversal.
     */
    _pushLeft(node) {
        while (node !== null) {
            this.stack.push(node);
            node = node.left;
        }
    }

    /**
     * @return {number}
     */
    next() {
        const topNode = this.stack.pop();

        // If there's a right subtree, the smallest value in that
        // subtree is its leftmost node.
        if (topNode.right !== null) {
            this._pushLeft(topNode.right);
        }

        return topNode.val;
    }
    /**
     * @return {boolean}
     */
    hasNext() {
        return this.stack.length > 0;
    }
}
```

Terminology & Real-World Context

New Terms

- **Controlled Recursion (or Manual Stack):** Instead of letting the computer handle the "backtracking" automatically via function calls, we use a `List` or `Array` as a stack. This allows us to "pause" and "resume" the traversal, which is exactly what an iterator needs to do.
- **Amortized Analysis:** This explains why `next()` is O(1). Some calls take longer (if they have to push many right-child-left-descendants), but most calls are very fast. When you average them out over the whole tree, it's constant time.

Real-World Interview Variations

Big Tech companies often disguise this problem to see if you understand the underlying "stateful traversal" concept:

1. **Google (Infrastructure):** "Design a log-file iterator that merges sorted logs from different shards without loading all logs into memory." (This uses the same 'lazy loading' logic).
2. **Meta (Social Graph):** "Find the K-th smallest element in a BST." (An L5 would use this Iterator logic to stop exactly at K, rather than sorting the whole tree).
3. **Bloomberg (Financial Data):** "Implement a 'Prev()' method in addition to 'Next()'." (This requires using two stacks or a more complex parent-pointer approach to move backward through the stock price tree).

450. Delete Node in a BST

A senior (L5) or staff (L6) engineer doesn't just look for a way to delete a node; they look for the most **robust, maintainable, and edge-case-proof** way to handle tree mutations. In a BST (Binary Search Tree), deleting a node is non-trivial because you must maintain the **BST Property**: for every node, all nodes in the left subtree must be smaller, and all nodes in the right subtree must be larger.

---

1. Problem Explanation

You are given the root of a Binary Search Tree and a `key`. You need to:

1. **Find** the node with that key.
2. **Remove** it.
3. **Rearrange** the remaining nodes so it remains a valid BST.

The "trick" isn't finding the node (that's just a standard search); the trick is what happens after you "cut" a node out of the middle of a connected structure.

---

2. Solution Explanation

To solve this like a top-tier engineer, we break the problem into three distinct scenarios based on the "family status" of the node we want to delete.

Scenario A: The Leaf Node (No Children)

This is the easiest. We just set the parent's pointer to null.

```
      5
     / \
    3   6
   /
  2  <-- Delete this

Result:
      5
     / \
    3   6
```

Scenario B: One Child

If the node has only one child, we "skip" the deleted node and connect the parent directly to that child (like a grandparent taking over for a parent).

```
      5
     / \
    3   6
     \
      4 <-- Delete 3. Since 3 has only one child (4),
            5 will now point directly to 4.

Result:
      5
     / \
    4   6
```

Scenario C: Two Children (The Non-Trivial Part)

This is where candidates fail. If a node has two children, you can't just "delete" it without leaving a massive hole.

The Strategy: We don't actually delete the node structure. Instead, we **replace its value** with the next best candidate and then delete that candidate from its original position.

Who is the "next best candidate"?

- **The Inorder Successor:** The smallest value in the right subtree.
- **Why?** Because the smallest value in the right subtree is guaranteed to be larger than everything in the left subtree and smaller than everything else in the right subtree. It fits perfectly.

Visualizing Scenario C (Delete Node 5)

```
Step 1: Original Tree
          5 (Target)
        /   \
       3     8
      / \   / \
     2   4 7   9
          /
         6

Step 2: Find Inorder Successor (Smallest in Right Subtree)
Go Right once (8), then Left as far as possible -> Node 6.

Step 3: Copy Successor value (6) to Target (5)
          6 (Updated)
        /   \
       3     8
      / \   / \
     2   4 7   9
          /
         6 (Now a duplicate)

Step 4: Delete the original Successor node (the leaf 6)
          6
        /   \
       3     8
      / \   / \
     2   4 7   9
```

---

3. Time and Space Complexity Analysis

An L5/L6 engineer will always clarify that complexity depends on the **height (h)** of the tree, not just the number of nodes (n).

Time Complexity (TC)

We search from root to leaf in the worst case.

```
Operation          | Complexity | Reasoning
------------------------------------------------------------
Search for Key     | O(h)       | Traversal follows one path down.
Find Successor     | O(h)       | Only happens once in Case C.
Delete Successor   | O(h)       | A second traversal or recursion.
------------------------------------------------------------
Total TC           | O(h)       | Where h is the height of the tree.
```

Note: In a balanced tree, $h = log(n)$. In a skewed tree (like a linked list), $h = n$.

Space Complexity (SC)

The space is consumed by the **Recursion Stack**.

```
Structure          | Complexity | Reasoning
------------------------------------------------------------
Recursion Stack    | O(h)       | Max depth of the function calls
                   |            | equals the height of the tree.
------------------------------------------------------------
Total SC           | O(h)       |
```

---

4. Solution Code

Python Implementation

```python
class Solution:
    def deleteNode(self, root, key):
        if not root:
            return None

        # Standard BST Search Logic
        if key < root.val:
            root.left = self.deleteNode(root.left, key)
        elif key > root.val:
            root.right = self.deleteNode(root.right, key)
        else:
            # WE FOUND THE NODE!

            # Case 1 & 2: Leaf or One Child
            if not root.left:
                return root.right
            if not root.right:
                return root.left

            # Case 3: Two Children
            # Strategy: Find successor, swap value, delete successor
            successor = self._get_min(root.right)
            root.val = successor.val
            # Now delete the duplicate successor in the right subtree
            root.right = self.deleteNode(root.right, successor.val)

        return root

    def _get_min(self, node):
        """
        Helper to find the leftmost node in a subtree.
        In a BST, this is always the minimum value.
        """
        current = node
        while current.left:
            current = current.left
        return current
```

Javascript Implementation

```
var deleteNode = function(root, key) {
    if (!root) return null;

    if (key < root.val) {
        root.left = deleteNode(root.left, key);
    } else if (key > root.val) {
        root.right = deleteNode(root.right, key);
    } else {
        // Node found!

        // Handle Case: No Left Child (covers leaf and one-child cases)
        if (!root.left) return root.right;

        // Handle Case: No Right Child
        if (!root.right) return root.left;

        // Handle Case: Two children
        // 1. Find the inorder successor (min in right subtree)
        let successor = root.right;
        while (successor.left) {
            successor = successor.left;
        }

        // 2. Overwrite current node's value with successor's value
        root.val = successor.val;

        // 3. Recursively delete the successor node
        root.right = deleteNode(root.right, successor.val);
    }
    return root;
};
```

Note 1: Terms & Techniques

- **Inorder Successor:** In a BST, this is the node with the smallest value greater than the current node. It's found by going right once and then as far left as possible. It helps because it's the "least disruptive" replacement for a deleted node.
- **Recursion Stack:** This is the memory used by the computer to keep track of active functions. In tree problems, the depth of this stack is proportional to the tree's height.

Note 2: Real-World Interview Variations

FAANG companies rarely ask "Delete a node" directly anymore. They wrap it in system design or complex scenarios:

1. **Google (Infrastructure):** "Implement a dynamic IP routing table where routes (ranges) are added and deleted frequently. How do you maintain the lookup tree?" (Essentially BST deletion).
2. **Meta (Social Graph):** "Given a hierarchical comment system (nested threads), how would you handle 'deleting' a parent comment while keeping the sub-thread structure intact and logically ordered?"
3. **Bloomberg (Trading Systems):** "You have a Limit Order Book stored in a tree. When a trade is canceled, you must remove that price point. How do you rebalance or restructure the tree efficiently?"

669. Trim a Binary Search Tree

A senior-level approach (L5/L6) to this problem focuses on **recursive elegance** and **immutability principles**. While a junior engineer might get bogged down in complex pointer manipulation or iterative loops, a senior engineer recognizes that a Binary Search Tree (BST) has a recursive structure that naturally lends itself to a recursive solution.

---

1. Problem Explanation

The goal is to "trim" a Binary Search Tree so that all its elements fall within a closed interval `[low, high]`.

The Rules of Trimming

1. If a node's value is **less than `low**`, that node and its entire left subtree are invalid (because everything in the left subtree is even smaller).
2. If a node's value is **greater than `high**`, that node and its entire right subtree are invalid (because everything in the right subtree is even larger).
3. If a node's value is **within the range**, we keep it but must recursively check and trim its left and right children.

The "trick" is that when you remove a node, you don't just delete it; you must stitch the tree back together so it remains a valid BST.

---

2. Solution Explanation

A top-tier engineer uses the **Post-order Traversal** logic. We process the nodes and return the "new" version of the subtree to the parent.

The Core Logic

- **Case 1: Node < low** The current node is too small. Since it's a BST, everything to its left is also too small. We discard the node and its left side, and simply return whatever the result of trimming the **right** side is.
- **Case 2: Node > high** The current node is too big. Everything to its right is also too big. We discard the node and its right side, returning the result of trimming the **left** side.
- **Case 3: low <= Node <= high** The node is a keeper! We "trim" its left child and "trim" its right child, then return the node itself.

Visualization 1: Trimming a "Too Small" Node

Imagine `low = 5, high = 10`. We encounter node `3`.

```
  3 (Current Node)  <-- Too small!
 / \
1   4               <-- These are all < 5. Discard!
     \
      7             <-- This might be valid.
```

Action: The function at `3` tells its parent: "Don't use me. Use the result of trimming my right child (the 7) instead."

Visualization 2: Full Example walkthrough

Range: `[3, 7]`

Original Tree:

```
      5
    /   \
   2     8
  / \   / \
 1   3 6   9
```

Step-by-Step Trimming:

1. **At 5:** In range `[3, 7]`. Keep it. Now check children.
2. **At 2:** `2 < 3` (Too small). Discard `2` and its left child (`1`). Return the result of trimming its right child (`3`).
3. **At 3:** In range. Keep it.
4. **At 8:** `8 > 7` (Too big). Discard `8` and its right child (`9`). Return the result of trimming its left child (`6`).
5. **At 6:** In range. Keep it.

Final Trimmed Tree:

```
      5
    /   \
   3     6
```

---

3. Time and Space Complexity Analysis

A senior engineer explains complexity by tracing the "visit" path of the algorithm.

Time Complexity (TC)

We visit every node in the tree exactly once to check if it's within the range.

```
TC Derivation:
Nodes in tree = N
Work per node = Constant O(1) checks
```

```
Total Work    = N * O(1)
Result        = O(N)
```

Space Complexity (SC)

The space is determined by the recursion stack. In the worst case (a "skewed" tree that looks like a linked list), the stack depth equals the number of nodes.

```
SC Derivation:
Best Case (Balanced Tree) = O(log N) stack depth
Worst Case (Skewed Tree)   = O(N) stack depth
Result                    = O(H) where H is height of tree
```

---

4. Solution Code

Python Implementation

```python
class Solution:
    def trimBST(self, root, low, high):
        if not root:
            return None

        # If the current node is too big, the valid part must be on the left
        if root.val > high:
            return self.trimBST(root.left, low, high)

        # If the current node is too small, the valid part must be on the right
        if root.val < low:
            return self.trimBST(root.right, low, high)

        # If node is in range, recursively trim its children and link them
        # This 're-linking' is what reconstructs the trimmed tree
        root.left = self.trimBST(root.left, low, high)
        root.right = self.trimBST(root.right, low, high)

        return root
```

Javascript Implementation

```javascript
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 * this.val = (val===undefined ? 0 : val)
 * this.left = (left===undefined ? null : left)
 * this.right = (right===undefined ? null : right)
 * }
 */
var trimBST = function(root, low, high) {
    // Base Case: We've hit a leaf's child
    if (!root) return null;

    // Logic: If root.val is out of bounds, we 'skip' it by
    // returning the result of the call on the opposite side.
    if (root.val < low) {
        return trimBST(root.right, low, high);
    }
    if (root.val > high) {
        return trimBST(root.left, low, high);
    }

    // If root.val is within [low, high], we must keep this node
    // and process its children.
    root.left = trimBST(root.left, low, high);
    root.right = trimBST(root.right, low, high);

    return root;
};
```

Note 1: Key Technique - Divide and Conquer (Recursive BST Invariant)

This solution relies on the **BST Invariant**: `Left < Root < Right`.

- **Why it helps:** It allows us to prune entire branches. If a node is too small, we don't even look at its left child because we *know* it's also too small.
- **Application:** This transforms a potentially complex "deletion" problem into a simple "selection" problem. We aren't deleting; we are just choosing which valid subtrees to return up the chain.

Note 2: Real-World Interview Variations

Big Tech companies often disguise this "BST Trimming" logic in broader system design or algorithmic rounds:

1. **Google (Data Privacy/GDPR):** "You have a tree representing a directory of files with timestamps. Remove all files/folders created outside of a specific date range while maintaining the structure of the remaining directory."
2. **Meta (Feed Filtering):** "Given a BST of post IDs (sorted by 'relevance score'), filter the tree to only show posts within a certain 'quality score' range to a user."

3. **Bloomberg (Financial Range Query):** "You have a BST of stock prices. A user wants to see a sub-view of the market where prices are only between Price A and Price B. Return the restructured tree representing this view."

1305. All Elements in Two Binary Search Trees

A Google L5 or L6 engineer doesn't just look for *a* solution; they look for the most efficient, readable, and scalable one. For this problem, they wouldn't just dump everything into an array and sort it—that's the "L3" approach. They would recognize that Binary Search Trees (BSTs) are already "sorted" structures and leverage that property.

---

1. Problem Explanation

You are given two Binary Search Trees. Your goal is to return a single list containing all integers from both trees, sorted in ascending order.

The "hint" hidden in the data structure is that **an in-order traversal of a BST yields a sorted list. The Challenge:** If we just get two sorted lists and join them, we have to sort them again. If we use a standard sorting algorithm, we aren't using the fact that they were *already* partially sorted. An L5+ engineer sees this as a **Merge Sort** problem.

---

2. Solution Explanation

The optimal approach involves two main steps:

1. **Flattening:** Convert both BSTs into sorted arrays using In-order Traversal (Left -> Root -> Right).
2. **Merging:** Use the "Merge" step of the Merge Sort algorithm to combine the two sorted arrays into one in linear time.

Visualizing Step 1: In-order Traversal

Imagine Tree 1 looks like this:

```
    2
   / \
  1   4
```

An in-order traversal visits: **1 -> 2 -> 4**.

Imagine Tree 2 looks like this:

```
      1
     / \
    0   3
```

An in-order traversal visits: **0 -> 1 -> 3**.

Visualizing Step 2: The Two-Pointer Merge

Now we have two sorted lists: List A: `[1, 2, 4]` List B: `[0, 1, 3]`

We use two pointers (i and j) to compare the elements one by one:

Round 1:

```
List A: [1, 2, 4]
         ^ (i)
List B: [0, 1, 3]
         ^ (j)
Comparison: 1 vs 0.
0 is smaller. Add 0 to Result. Move j.
Result: [0]
```

Round 2:

```
List A: [1, 2, 4]
         ^ (i)
List B: [0, 1, 3]
            ^ (j)
Comparison: 1 vs 1.
They are equal. Add either (let's say A[i]). Move i.
Result: [0, 1]
```

Round 3:

```
List A: [1, 2, 4]
            ^ (i)
List B: [0, 1, 3]
            ^ (j)
Comparison: 2 vs 1.
1 is smaller. Add 1 to Result. Move j.
Result: [0, 1, 1]
```

Round 4:

```
List A: [1, 2, 4]
            ^ (i)
List B: [0, 1, 3]
               ^ (j=out)
Comparison: 2 vs 3.
2 is smaller. Add 2 to Result. Move i.
Result: [0, 1, 1, 2]
```

…and so on until both lists are exhausted.

---

3. Time and Space Complexity Analysis

An L5 engineer focuses on "Big O" to ensure the code handles millions of nodes if necessary.

Time Complexity (TC)

We do two in-order traversals and one linear merge.

```
Step 1: In-order Traversal of Tree 1 (N nodes) -> O(N)
Step 2: In-order Traversal of Tree 2 (M nodes) -> O(M)
Step 3: Merging two lists of size N and M      -> O(N + M)
-----------------------------------------------------------
Total Time Complexity: O(N + M)
```

Space Complexity (SC)

We need to store the elements in lists and account for the recursion stack.

```
List storage: N + M elements                   -> O(N + M)
Recursion Stack (Tree height):
  - Tree 1 height (H1)                         -> O(H1)
  - Tree 2 height (H2)                         -> O(H2)
-----------------------------------------------------------
Total Space Complexity: O(N + M)
(Note: In the worst case of a skewed tree, H = N)
```

---

4. Solution Code

Python Snippet

```python
class Solution:
    def getAllElements(self, root1, root2):
        # Helper function to perform in-order traversal
        # This visits nodes in ascending order: Left, then Root, then Right
        def inorder(node, lst):
            if not node:
                return
            inorder(node.left, lst)
            lst.append(node.val)
            inorder(node.right, lst)

        list1, list2 = [], []
        inorder(root1, list1)
        inorder(root2, list2)

        # Merge two sorted lists using the two-pointer technique
        # This is more efficient than calling .sort() on a combined list
        res = []
        i, j = 0, 0
        while i < len(list1) and j < len(list2):
            if list1[i] < list2[j]:
                res.append(list1[i])
                i += 1
            else:
                res.append(list2[j])
                j += 1

        # Append any remaining elements from either list
        return res + list1[i:] + list2[j:]
```

JavaScript Snippet

```
var getAllElements = function(root1, root2) {
    const list1 = [];
    const list2 = [];

    /**
     * Standard recursive in-order traversal.
     * Extracts values from BST in sorted order.
     */
    const inorder = (node, arr) => {
        if (!node) return;
        inorder(node.left, arr);
        arr.push(node.val);
        inorder(node.right, arr);
    };

    inorder(root1, list1);
    inorder(root2, list2);

    const result = [];
    let i = 0, j = 0;

    /**
     * Linear merge process.
     * Compares the heads of two sorted arrays to build the final sorted result.
     */
    while (i < list1.length && j < list2.length) {
        if (list1[i] < list2[j]) {
            result.push(list1[i++]);
        } else {
            result.push(list2[j++]);
        }
    }

    // Concatenate remaining elements
    return [...result, ...list1.slice(i), ...list2.slice(j)];
};
```

Note 1: Terms and Techniques

- **In-order Traversal:** A way of visiting all nodes in a BST such that you visit them in non-decreasing order. It is the "bread and butter" of BST problems.
- **Two-Pointer Merge:** The technique used in Merge Sort. It allows you to combine two sorted sequences into one sorted sequence in a single pass. It's highly efficient because it avoids the O(K log K) cost of a full sort.

Note 2: Real-World Interview Variations

Google, Meta, and Bloomberg often ask variations of this to test if you can handle **Scale** or **Streaming Data**:

1. **Iterative Version (The "Live" Stream):** "What if the trees are too large to fit in memory?" An L6 would propose using an **Iterator** (using a stack) to stream elements from both trees and merge them on the fly without ever creating the full intermediate `list1` and `list2`.
2. **K-Way Merge:** "How would you solve this if there were 1,000 BSTs instead of 2?" This tests your knowledge of **Min-Heaps / Priority Queues**.
3. **Sorted Iterator:** Meta often asks to design a "BST Iterator" class first, then asks you to use two of those iterators to solve this problem.

307. Range Sum Query - Mutable

A Google L5 or L6 engineer doesn't just look for *a* solution; they look for the *optimal* trade-off between update speed and query speed. For a mutable range sum problem, they immediately recognize that a naive approach (like a simple array) will fail on performance.

Here is how a senior engineer breaks this down.

---

1. Problem Explanation

We need to design a data structure that supports two operations on an array of numbers:

1. **update(index, val):** Change the value at a specific position.
2. **sumRange(left, right):** Calculate the sum of elements between two indices.

The Conflict

- **Approach A (Simple Array):** `update` is O(1), but `sumRange` is O(n) because you have to loop through the elements every time.
- **Approach B (Prefix Sums):** `sumRange` is O(1), but `update` is O(n) because changing one element requires recalculating all subsequent sums.

An L5+ engineer looks for the "middle ground" where both operations are **O(log n)**. The most robust tool for this is a **Segment Tree**.

---

2. Solution Explanation: The Segment Tree

A Segment Tree is a binary tree where each node represents an interval (or "segment") of the array. The root represents the whole array, and its children represent the left and right halves.

Building the Tree (Visualization)

Imagine an array: `[1, 3, 5, 7]`

]

ASCII Representation of the Tree:

```
Level 0 (Root):          [0-3] Sum: 16
                        /             \
Level 1:           [0-1] Sum: 4      [2-3] Sum: 12
                  /        \        /        \
Level 2 (Leaves): [0]S:1   [1]S:3  [2]S:5    [3]S:7
```

How `sumRange(1, 3)` works:

We want the sum of indices 1, 2, and 3 (3 + 5 + 7 = 15).

1. Start at Root [0-3]. The range [1-3] overlaps both children.
2. Go Left to [0-1]. Only index [1] is needed. Value = 3.
3. Go Right to [2-3]. This entire range is inside our target [1-3]. Value = 12.
4. Result: 3 + 12 = 15.

How `update(1, 10)` works:

We change index 1 from 3 to 10.

1. Find leaf node [1]. Update 3 -> 10.
2. Update parent [0-1]: New sum = 1 + 10 = 11.
3. Update root [0-3]: New sum = 11 + 12 = 23.

Updated ASCII Tree:

```
Level 0:                 [0-3] Sum: 23*
                        /             \
Level 1:           [0-1] Sum: 11* [2-3] Sum: 12
                  /        \        /        \
Level 2:        [0]S:1    [1]S:10* [2]S:5    [3]S:7
```

---

3. Time and Space Complexity Analysis

The complexity is derived from the height of the tree. Since the tree is binary and splits the array in half at each level, the height is `log(n)`.

Time Complexity Derivation:

```
Operation | Logic                                    | Complexity
------------------------------------------------------------------
Build     | Visits every node once (2n nodes)        | O(n)
Update    | Traverses from leaf to root (height)     | O(log n)
Query     | Traverses at most 4 nodes per level      | O(log n)

Visualizing the "log n" path:
[*] Root
 |
[*] (1/2 size)
 |
[*] (1/4 size) -> Only 'log n' steps to reach the bottom.
```

Space Complexity Derivation:

```
Storage   | Logic                                    | Complexity
------------------------------------------------------------------
Tree      | An array-based segment tree needs        | O(n)
            roughly 4 * n space to be safe.          |
```

---

4. Solution Code

Python Implementation

```python
class NumArray:
    def __init__(self, nums: list[int]):
        self.n = len(nums)
        # We use a 1-indexed array for the tree for easier math
        # Size 4*n is the standard safe bound for a segment tree
        self.tree = [0] * (4 * self.n)
        if self.n > 0:
            self._build(nums, 1, 0, self.n - 1)

    # Recursive function to initialize the tree nodes
    def _build(self, nums, node, start, end):
        if start == end:
            self.tree[node] = nums[start]
            return
        mid = (start + end) // 2
        self._build(nums, 2 * node, start, mid)
        self._build(nums, 2 * node + 1, mid + 1, end)
        self.tree[node] = self.tree[2 * node] + self.tree[2 * node + 1]

    # Updates a value and ripples the change up to the root
    def update(self, index: int, val: int) -> None:
        def _update(node, start, end, idx, new_val):
            if start == end:
                self.tree[node] = new_val
                return
            mid = (start + end) // 2
            if idx <= mid:
                _update(2 * node, start, mid, idx, new_val)
            else:
                _update(2 * node + 1, mid + 1, end, idx, new_val)
            self.tree[node] = self.tree[2 * node] + self.tree[2 * node + 1]

        _update(1, 0, self.n - 1, index, val)

    # Fetches the sum of a specific range by combining node values
    def sumRange(self, left: int, right: int) -> int:
        def _query(node, start, end, l, r):
            if r < start or end < l:
                return 0
            if l <= start and end <= r:
                return self.tree[node]
            mid = (start + end) // 2
            return _query(2 * node, start, mid, l, r) + \
                   _query(2 * node + 1, mid + 1, end, l, r)

        return _query(1, 0, self.n - 1, left, right)
```

JavaScript Implementation

```
class NumArray {
    constructor(nums) {
        this.n = nums.length;
        this.tree = new Array(4 * this.n).fill(0);
        this.build(nums, 1, 0, this.n - 1);
    }

    /* Standard recursive build: O(n) */
    build(nums, node, start, end) {
        if (start === end) {
            this.tree[node] = nums[start];
            return;
        }
        let mid = Math.floor((start + end) / 2);
        this.build(nums, 2 * node, start, mid);
        this.build(nums, 2 * node + 1, mid + 1, end);
        this.tree[node] = this.tree[2 * node] + this.tree[2 * node + 1];
    }

    /* Updates the target leaf and all affected ancestors */
    update(index, val, node = 1, start = 0, end = this.n - 1) {
        if (start === end) {
            this.tree[node] = val;
            return;
        }
        let mid = Math.floor((start + end) / 2);
        if (index <= mid) {
            this.update(index, val, 2 * node, start, mid);
        } else {
            this.update(index, val, 2 * node + 1, mid + 1, end);
        }
        this.tree[node] = this.tree[2 * node] + this.tree[2 * node + 1];
    }

    /* Sums only the segments that fall within [left, right] */
    sumRange(left, right, node = 1, start = 0, end = this.n - 1) {
        if (right < start || end < left) return 0;
        if (left <= start && end <= right) return this.tree[node];
        let mid = Math.floor((start + end) / 2);
        return this.sumRange(left, right, 2 * node, start, mid) +
               this.sumRange(left, right, 2 * node + 1, mid + 1, end);
    }
}
```

Notes on Techniques

Segment Tree

A Segment Tree is a tree data structure used for storing information about intervals. It allows querying which of the stored segments contains a given point or which segments overlap with a given interval.

- **Why it helps:** It balances the "cost" of updates and queries. Instead of one being super fast and the other super slow, it makes both very efficient (logarithmic).

Binary Indexed Tree (Fenwick Tree)

Often mentioned as an alternative. It is more space-efficient (O(n)) and easier to code but slightly less intuitive than Segment Trees. A senior dev might choose this if they want to impress with brevity.

Real-World / Interview Variations

Google, Meta, and Bloomberg rarely ask this "raw." They wrap it in scenarios:

1. **Google (Analytics):** "Given a stream of ad clicks, find the total clicks in a specific timestamp range, allowing for corrections to old data."
2. **Meta (Ranking):** "Count the number of friends with 'activity scores' between X and Y, where scores change frequently." (Often called 'Range Frequency Queries').
3. **Bloomberg (Finance):** "Calculate the total volume of stock trades within a price window, where trades are being cancelled or modified in real-time."
4. **2D Variation:** "Range Sum Query 2D - Mutable." This requires a 2D Segment Tree or 2D BIT, which is a common follow-up for L6 candidates.

652. Find Duplicate Subtrees

A top-tier engineer at Google or Meta doesn't just look for *a* solution; they look for the most **performant** and **scalable** solution. For "Find Duplicate Subtrees," the challenge isn't just finding duplicates—it's doing so without falling into a performance trap called "string serialization bloat."

1. Problem Explanation

We are given the root of a binary tree. We need to find all **duplicate subtrees**. For each kind of duplicate subtree, we only need to return the root node of **one** of them.

Two trees are duplicates if they have the **same structure** with the **same node values**.

The Core Challenge

In a tree, a "subtree" is defined by a node and all of its descendants. To know if two subtrees are identical, we need a way to "fingerprint" or "serialize" them into a unique representation (like a string or an ID) so we can compare them efficiently.

2. Solution Explanation

A senior engineer would approach this using **Post-order Traversal** combined with **Unique Identifier Mapping**.

Step-by-Step Logic

1. **Traverse:** We visit the tree from the bottom up (Left, Right, then Root).
2. **Serialize:** At each node, we create a unique "signature" representing that specific subtree.
3. **Record:** We store these signatures in a Hash Map (Frequency Counter).
4. **Identify:** If a signature's count reaches exactly **2**, we've found a duplicate! We add that node to our result list.

Why ASCII Visualizations Matter

Let's look at an example tree:

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

Visualizing Subtrees: In the tree above, the subtree `(2 -> 4)` appears twice. The leaf node `(4)` also appears three times.

The "Naïve" Serialization Approach: Many candidates use strings: `"4,null,null"`. At node `2` (the left one), the string becomes `"2,4,null,null,null"`. *Danger:* In a very deep, unbalanced tree, these strings get massive, leading to O(N^2) time complexity because copying long strings is expensive.

The "L6" Optimized Approach: Merkle-ish IDs Instead of long strings, we map every unique structure to a **short Integer ID**.

ASCII Execution Trace: Let's trace the "ID" method on this tree:

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

1. **Leaf Node 4:**

- Left child ID: 0 (null)
- Right child ID: 0 (null)
- Signature: `(value:4, left:0, right:0)`
- Map this signature to **ID: 1**.
- Count for ID 1 = 1.

2. **Node 2 (Bottom):**

- Left child (Node 4) ID: 1
- Right child ID: 0
- Signature: `(value:2, left:1, right:0)`
- Map this signature to **ID: 2**.
- Count for ID 2 = 1.

3. **Node 2 (Top Left):**

- Left child (Node 4) ID: 1
- Right child ID: 0
- Signature: `(value:2, left:1, right:0)`
- **Wait!** We've seen this signature before. It already has **ID: 2**.
- Count for ID 2 = 2. **ADD TO RESULT.**

---

3. Complexity Analysis

We use an ASCII representation to derive the complexity of the **ID-mapping** version.

Time Complexity (TC)

```
Total Nodes: N
-----------------------------------------------------------
Operation               | Complexity | Reason
------------------------|------------|----------------------
Tree Traversal          | O(N)       | Visit each node once.
Signature Creation      | O(1)       | Using 3 integers (val, L-id, R-id).
Map Lookup/Insert       | O(1)       | Hash map average case.
------------------------|------------|----------------------
Total Time Complexity   | O(N)       |
-----------------------------------------------------------
```

Note: The string serialization method is O(N^2) because string concatenation and hashing take O(N) at each of the N nodes.

Space Complexity (SC)

```
Component               | Complexity | Reason
------------------------|------------|----------------------
Recursion Stack         | O(H)       | H = height of tree (worst case N).
Hash Map (Signatures)   | O(N)       | Stores up to N unique IDs.
Result List             | O(N)       | Worst case, many duplicates.
------------------------|------------|----------------------
Total Space Complexity  | O(N)       |
-----------------------------------------------------------
```

---

4. Solution Code

Python (The Optimized ID Approach)

```python
class Solution:
    def findDuplicateSubtrees(self, root):
        # map triple (node.val, left_id, right_id) to a unique ID
        trees = {}
        # count occurrences of each unique ID
        count = {}
        res = []

        # We use a counter to assign the next available unique ID
        self.id_counter = 1

        # This helper function performs a post-order traversal.
        # It assigns and returns a unique integer ID for the subtree
        # rooted at the current node.
        def getId(node):
            if not node:
                return 0

            # Post-order: get IDs of children first
            left_id = getId(node.left)
            right_id = getId(node.right)

            # Create a unique "triple" signature for the current subtree
            triple = (node.val, left_id, right_id)

            if triple not in trees:
                trees[triple] = self.id_counter
                self.id_counter += 1

            uid = trees[triple]
            count[uid] = count.get(uid, 0) + 1

            # If we see the ID for the second time, it's a duplicate.
            # We don't add it a third time to avoid duplicates in result.
            if count[uid] == 2:
                res.append(node)

            return uid

        getId(root)
        return res
```

JavaScript (The Serialization Approach)

Note: For many interviews, the string serialization is acceptable, but the ID approach above is the "Top-of-Band" answer.

```
var findDuplicateSubtrees = function(root) {
    const map = new Map();
    const res = [];

    /**
     * postOrder traverses the tree and returns a string
     * representation of the subtree.
     */
    function postOrder(node) {
        if (!node) return "#"; // Use # to represent null

        // Construct the signature: LeftChild, RightChild, currentVal
        const left = postOrder(node.left);
        const right = postOrder(node.right);
        const signature = left + "," + right + "," + node.val;

        const count = map.get(signature) || 0;

        // If count is 1, this is the second time we've seen it
        if (count === 1) {
            res.push(node);
        }

        map.set(signature, count + 1);
        return signature;
    }

    postOrder(root);
    return res;
};
```

New Terms & Real-World Context

Key Terms

- **Post-order Traversal:** Visiting children before the parent. This is essential here because a parent's "identity" depends entirely on what its children are.
- **Merkle Tree Logic:** The ID-mapping technique is similar to how Blockchains or Git works. Instead of comparing whole files (or subtrees), you compare "hashes" (IDs). If the hashes match, the content is identical.

How Big Tech Asks This (Indirectly)

1. **Google (Infrastructure):** "We have millions of files in a file system. How do we find identical folder structures to save space (De-duplication)?"
2. **Meta (Social Graph):** "Given two users' friend networks, find if they have an identical sub-community structure."
3. **Bloomberg (Data):** "You are receiving financial XML trees. Identify repeating data patterns to compress the transmission."

Boundary of Binary Tree

A Google L5 (Senior) or L6 (Staff) engineer doesn't just look for a way to pass the test cases; they look for **symmetry, modularity, and edge-case resilience**. In an interview, they would treat this problem as a composition of three distinct sub-problems rather than one giant, messy traversal.

---

1. Problem Explanation

The "Boundary of Binary Tree" problem asks you to return the values of the nodes that form the "outline" of the tree, starting from the root and moving **counter-clockwise**.

The boundary consists of:

1. **The Root:** (Unless it's already part of another group, but we usually start here).
2. **Left Boundary:** All nodes from the root's left child down to the leftmost leaf (excluding the leaf itself).
3. **Leaves:** All leaf nodes from left to right.
4. **Right Boundary:** All nodes from the root's right child down to the rightmost leaf (excluding the leaf itself), listed in **reverse** order (bottom to top).

A Critical Non-Trivial Detail: The Definition of "Boundary"

A common mistake is thinking the Left Boundary is just the "left side" of the tree. It's actually a specific path:

- If a node has a left child, follow it.
- If it *only* has a right child, follow the right child.
- Stop before you hit a leaf.

---

2. Solution Explanation

An L5+ engineer would solve this by breaking the tree into segments. We use a **Pre-order traversal** logic but specialized for each segment.

The Mental Model: The Four-Part Harmony

```
          [ 1 ] <--- Root
         /     \
      [ 2 ]    [ 3 ] <--- Left/Right Boundary paths
     /   \      /  \
   [4]   [5] [6]  [7] <--- Leaves
```

Visualizing the Walkthrough

Consider this complex tree:

```
        ____ 1 ____
       /           \
      2             3
     / \           /
    4   5         6
```

```
        / \        / \
       7   8      9   10
```

Step 1: The Root

- Add `1`.

Step 2: Left Boundary (Top-Down)

- Start at `2`. It has a left child (`4`), so we take it.
- Wait! `4` is a leaf. We skip leaves in this step to avoid duplicates.
- Result: `[1, 2]`

Step 3: All Leaves (Left-to-Right)

- We do a standard DFS. We find `4`, `7`, `8`, `9`, `10`.
- Result: `[1, 2, 4, 7, 8, 9, 10]`

Step 4: Right Boundary (Bottom-Up)

- Start at `3`. It has a left child `6`, but we prioritize the right child if it exists.
- It doesn't have a right child, so we go to `6`.
- But `6` is not a leaf? Correct, but its children are leaves.
- We track `3` and `6`. Since it's counter-clockwise, we reverse them (or add to a stack).
- Result: `[1, 2, 4, 7, 8, 9, 10, 6, 3]`

---

3. Time and Space Complexity Analysis

Engineers at this level avoid complex notation. Here is the derivation:

```
TIME COMPLEXITY DERIVATION
--------------------------
Left Boundary Scan   : O(H) where H is height
Leaf DFS Scan        : O(N) visits every node once
Right Boundary Scan  : O(H) where H is height
--------------------------
Total Time           : O(N)
(Since N is always greater than or equal to H)

SPACE COMPLEXITY DERIVATION
---------------------------
Result Array         : O(N) to store the boundary
Recursion Stack      : O(H) for the depth of the tree
---------------------------
Total Space          : O(N)
```

---

4. Solution Code

Python Implementation

```python
class Solution:
    def boundaryOfBinaryTree(self, root):
        if not root: return []

        # A node is a leaf if it has no children
        def is_leaf(node):
            return not node.left and not node.right

        # 1. Add root (if not a leaf)
        res = []
        if not is_leaf(root):
            res.append(root.val)

        # 2. Add Left Boundary
        # We start from root.left and move down.
        # If a left child exists, go left. Otherwise, go right.
        curr = root.left
        while curr:
            if not is_leaf(curr):
                res.append(curr.val)
            if curr.left:
                curr = curr.left
            else:
                curr = curr.right

        # 3. Add All Leaves
        # Standard DFS to find leaves from left to right.
        def add_leaves(node):
            if not node: return
            if is_leaf(node):
                res.append(node.val)
                return
            add_leaves(node.left)
            add_leaves(node.right)

        add_leaves(root)

        # 4. Add Right Boundary (Bottom-Up)
        # We use a temporary list to reverse the order easily.
        right_boundary = []
        curr = root.right
        while curr:
            if not is_leaf(curr):
                right_boundary.append(curr.val)
            if curr.right:
                curr = curr.right
            else:
                curr = curr.left

        res.extend(reversed(right_boundary))
        return res
```

Javascript Implementation

```javascript
var boundaryOfBinaryTree = function(root) {
    if (!root) return [];

    const res = [];
    const isLeaf = (node) => !node.left && !node.right;

    // 1. Root handling
    if (!isLeaf(root)) res.push(root.val);

    // 2. Left Boundary: Top-down approach
    let curr = root.left;
    while (curr) {
        if (!isLeaf(curr)) res.push(curr.val);
        curr = curr.left ? curr.left : curr.right;
    }

    // 3. Leaves: Depth First Search to maintain order
    const addLeaves = (node) => {
        if (!node) return;
        if (isLeaf(node)) {
            res.push(node.val);
            return;
        }
        addLeaves(node.left);
        addLeaves(node.right);
    };
    addLeaves(root);

    // 4. Right Boundary: Bottom-up using a stack/reverse
    const rightSide = [];
    curr = root.right;
    while (curr) {
        if (!isLeaf(curr)) rightSide.push(curr.val);
        curr = curr.right ? curr.right : curr.left;
    }
    while (rightSide.length > 0) {
        res.push(rightSide.pop());
    }

    return res;
};
```

Note 1: Terms and Techniques

- **Pre-order vs. Post-order:** For the left boundary, we use pre-order (process node, then move down). For the right boundary, we effectively use post-order (collect nodes as we go down, then reverse them) to ensure the "bottom-up" requirement.
- **Modularization:** This is the "L5 way." Instead of one recursive function with 5 different `if` flags (e.g., `isLeftBoundary`, `isRightBoundary`), we write four clean, readable blocks. This reduces bugs and makes the code easier for teammates to review.

Note 2: Real-World Interview Variants

Google, Meta, and Bloomberg rarely ask the "exact" Leetcode problem anymore. Instead, they frame it like this:

1. **The "Drone" Problem (Google):** "You have a drone flying around a forest represented as a tree. It needs to circle the perimeter to map the area. What is its path?"
2. **The "Shadow" Problem (Meta):** "Given a 3D structure (represented as a tree), return the nodes that would be visible if you looked at it from the side (the silhouette)."
3. **The "UI Rendering" Problem (Bloomberg):** "In a nested UI component tree, identify all 'outer' components that need a specific border-shadow applied."