Chunk 1: Binary Tree Problems

Core Pattern: Recursive DFS (Post-order / Pre-order)

Used in: Invert, Balanced, Max Depth, LCA, Diameter, Max Path Sum, Path Sum II, Symmetric Tree, Subtree of Another Tree

How to Recognize:

- Problems involving tree traversal where you need to process children before parent (post-order) or after (pre-order).
- Common in: balancing checks, path sums, subtree comparisons, symmetry.
- Often requires returning values (height, sum, boolean) from recursive calls.

Step-by-Step Thinking Process (Recipe):

1. Define base case (`if not root: return ...`)
2. Recursively solve for left and right subtrees.
3. Combine results based on problem logic (e.g., `max(left_height, right_height) + 1`).
4. Use a global variable if needed (e.g., max diameter/path sum).
5. Return appropriate value for parent node.

Pitfalls & Edge Cases:

- Forgetting to handle empty trees (`root is None`).
- Returning wrong value types (e.g., returning height instead of boolean).
- Not updating global variables correctly (e.g., `diameter = max(diameter, left + right)`).
- Misunderstanding post-order vs pre-order traversal order.

Core Pattern: Breadth-First Search (BFS) by Levels

Used in: Level Order, Right Side View, Zigzag, Maximum Width

How to Recognize:

- Need level-by-level processing.
- Output depends on order within each level (first/last/alternating).
- Requires tracking nodes per level using queue.

Step-by-Step Thinking Process (Recipe):

1. Use a queue (collections.deque) and initialize with root.
2. While queue not empty:
    - Get current level size (`len(queue)`).
    - Process all nodes at this level in a loop.
    - Add their children to queue.
    - Record required info (first, last, or all values).
3. For zigzag: alternate direction every level using `reverse()` or `deque`.

Pitfalls & Edge Cases:

- Using `queue.pop(0)` (list) → O(n), use `deque` instead.
- For width: indices can grow large; rebase to avoid overflow.
- Handling empty root (return 0 width).

Core Pattern: Tree Reconstruction from Traversals

Used in: Construct from Preorder & Inorder

How to Recognize:

- Given two traversals (e.g., preorder + inorder), reconstruct tree.
- Preorder gives root order; inorder splits left/right subtrees.

Step-by-Step Thinking Process (Recipe):

1. Use a hashmap to store `inorder[i] -> index` for O(1) lookup.
2. Use recursion with bounds: `in_start`, `in_end`, `pre_start`, `pre_end`.
3. Root is `preorder[pre_start]`.
4. Find root index in inorder → split into left/right subtrees.
5. Recursively build left and right children.

Pitfalls & Edge Cases:

- Off-by-one errors in indices.
- Not handling empty input properly.
- Reconstructing without hash map → $O(n^2)$ time.

Core Pattern: Prefix Sum on Trees + Backtracking

Used in: Path Sum III

How to Recognize:

- Any path (not just root-to-leaf) summing to target.
- Use prefix sum technique: `current_sum - prev_sum == target`.

Step-by-Step Thinking Process (Recipe):

1. Use a `Counter` to track frequency of prefix sums encountered.
2. At each node: update `curr_sum += node.val`.
3. Check if `curr_sum - target` exists → valid path ending here.
4. Recurse left/right.
5. Backtrack: decrement count of `curr_sum` when leaving.

Pitfalls & Edge Cases:

- Forgetting to backtrack (count not removed).
- Missing edge case: `target = 0`, single node.

Core Pattern: Parent Map + BFS (Implicit Graph)

Used in: All Nodes Distance K in Binary Tree

How to Recognize:

- Need to traverse up and down from a node.
- Can't go upward in binary tree → convert to undirected graph via parent mapping.

Step-by-Step Thinking Process (Recipe):

1. Build parent map via DFS/BFS.
2. Start BFS from target node (k=0).
3. Traverse neighbors: left, right, parent.
4. Track visited nodes to avoid cycles.
5. Stop when distance > k.

Pitfalls & Edge Cases:

- Not storing parent relationships.
- Revisiting same node → infinite loop.
- k=0 → only return the target node.

Problem 1: Invert Binary Tree

Summary

Given a binary tree, invert it so that left and right children are swapped at every node.

Pattern

- **Recursive DFS (Pre-order)**

Solution with Comments

```
# 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 invertTree(root):
    # Base case: if node is null, return None
    if not root:
        return None

    # Swap left and right subtrees recursively
    # This is pre-order: process root first, then recurse
    root.left, root.right = invertTree(root.right), invertTree(root.left)

    # Return the modified (inverted) root
    return root


# ---- Official LeetCode Example ----
if __name__ == "__main__":
```

```python
# Example Input: root = [4,2,7,1,3,6,9]
# Tree structure:
#       4
#      / \
#     2   7
#    / \ / \
#   1  3 6  9
root = TreeNode(4)
root.left = TreeNode(2)
root.right = TreeNode(7)
root.left.left = TreeNode(1)
root.left.right = TreeNode(3)
root.right.left = TreeNode(6)
root.right.right = TreeNode(9)

# Call function
inverted_root = invertTree(root)

# Output should be [4,7,2,9,6,3,1]
# Level order: [4,7,2,9,6,3,1]
result = []
queue = [inverted_root]
while queue:
    node = queue.pop(0)
    if node:
        result.append(node.val)
        queue.append(node.left)
        queue.append(node.right)
    else:
        result.append(None)

# Remove trailing Nones for clean output
while result and result[-1] is None:
    result.pop()

print("Output:", result)  # Output: [4, 7, 2, 9, 6, 3, 1]
```

Walkthrough (Example)

- Start at root (4): swap its left and right → now left=7, right=2
- Go to left child (7): swap its children → left=9, right=6

- Go to right child (2): swap its children → left=3, right=1
- Final tree:

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

- Level order: `[4,7,2,9,6,3,1]`

Complexity

- **Time**: O(n) — visit every node once
- **Space**: O(h) — recursion stack depth, h = height (O(log n) avg, O(n) worst)

Problem 2: Balanced Binary Tree

Summary

Check if a binary tree is height-balanced (for every node, height difference between left and right 1).

Pattern

- **Recursive DFS (Post-order with height + flag)**

Solution with Comments

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

def isBalanced(root):
    # Helper returns (is_balanced, height)
```

```python
    def dfs(node):
        # Base case: empty node is balanced with height 0
        if not node:
            return True, 0

        # Recursively check left and right subtrees
        left_balanced, left_height = dfs(node.left)
        right_balanced, right_height = dfs(node.right)

        # Check if current node is balanced
        is_current_balanced = left_balanced and right_balanced and abs(
            left_height - right_height) <= 1

        # Compute current height
        current_height = max(left_height, right_height) + 1

        return is_current_balanced, current_height

    # Return whether tree is balanced
    balanced, _ = dfs(root)
    return balanced


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: root = [3,9,20,null,null,15,7]
    # Tree:
    #     3
    #    / \
    #   9   20
    #      /  \
    #     15   7
    root = TreeNode(3)
    root.left = TreeNode(9)
    root.right = TreeNode(20)
    root.right.left = TreeNode(15)
    root.right.right = TreeNode(7)

    # Call function
    result = isBalanced(root)

    print("Output:", result)  # Output: true
```

Walkthrough (Example)

- Node 9: height=1, balanced → True
- Node 15: height=1, balanced → True
- Node 7: height=1, balanced → True
- Node 20: |1–1|=0 → balanced, height=2
- Node 3: |1–2|=1 → balanced, height=3
- All levels balanced → return True

Complexity

- **Time**: O(n) — visit each node once
- **Space**: O(h) — recursion stack

Problem 3: Binary Tree Level Order Traversal

Summary

Return the level order traversal of a binary tree (list of lists, each inner list is a level).

Pattern

- **BFS by Levels**

Solution with Comments

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

from collections import deque

def levelOrder(root):
    # Handle empty tree
```

```python
    if not root:
        return []

    result = []
    queue = deque([root])

    while queue:
        level_size = len(queue)  # Number of nodes at current level
        current_level = []

        # Process all nodes at current level
        for _ in range(level_size):
            node = queue.popleft()
            current_level.append(node.val)

            # Add children to queue for next level
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        result.append(current_level)

    return result


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: root = [3,9,20,null,null,15,7]
    root = TreeNode(3)
    root.left = TreeNode(9)
    root.right = TreeNode(20)
    root.right.left = TreeNode(15)
    root.right.right = TreeNode(7)

    # Call function
    result = levelOrder(root)

    print("Output:", result)  # Output: [[3],[9,20],[15,7]]
```

Walkthrough (Example)

- Level 0: [3]
- Level 1: [9,20] (children of 3)
- Level 2: [15,7] (children of 9 and 20)
- Final: `[[3],[9,20],[15,7]]`

Complexity

- **Time**: O(n) — each node processed once
- **Space**: O(w) — max width of tree (w n)

Problem 4: Lowest Common Ancestor of a Binary Tree

Summary

Find the lowest common ancestor (LCA) of two nodes p and q in a binary tree.

Pattern

- **Recursive DFS (Post-order)**

Solution with Comments

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

def lowestCommonAncestor(root, p, q):
    # Base case: if root is None or matches p/q, return root
    if not root or root == p or root == q:
        return root

    # Recursively search in left and right subtrees
```

```python
    left_lca = lowestCommonAncestor(root.left, p, q)
    right_lca = lowestCommonAncestor(root.right, p, q)

    # If both sides return non-null, current node is LCA
    if left_lca and right_lca:
        return root

    # Otherwise, return the non-null result (either left or right)
    return left_lca or right_lca


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
    # Tree:
    #       3
    #      / \
    #     5   1
    #    / \ / \
    #   6  2 0  8
    #     / \
    #    7   4
    root = TreeNode(3)
    root.left = TreeNode(5)
    root.right = TreeNode(1)
    root.left.left = TreeNode(6)
    root.left.right = TreeNode(2)
    root.right.left = TreeNode(0)
    root.right.right = TreeNode(8)
    root.left.right.left = TreeNode(7)
    root.left.right.right = TreeNode(4)

    p = root.left      # node 5
    q = root.right     # node 1

    # Call function
    lca = lowestCommonAncestor(root, p, q)

    print("Output:", lca.val)  # Output: 3
```

Walkthrough (Example)

- Search for 5 and 1:
  - Left subtree finds 5 (at node 5), but not 1 → returns 5
  - Right subtree finds 1 → returns 1
  - At root: both left and right return non-null → return root (3)
- LCA is 3

Complexity

- **Time**: O(n) — visit each node once
- **Space**: O(h) — recursion stack

Problem 5: Serialize and Deserialize Binary Tree

Summary

Convert a binary tree to a string (serialize), and reconstruct it from the string (deserialize).

Pattern

- **DFS Pre-order Serialization**

Solution with Comments

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

class Codec:
    def serialize(self, root):
        """Encodes a tree to a single string using pre-order DFS."""
        if not root:
```

```python
            return "null,"

        # Serialize root, then left, then right
        return (
            str(root.val) + "," +
            self.serialize(root.left) +
            self.serialize(root.right)
            )

    def deserialize(self, data):
        """Decodes a string to a binary tree."""
        # Split by comma and use iterator for consumption
        vals = iter(data.split(","))

        def build():
            val = next(vals)
            if val == "null":
                return None

            node = TreeNode(int(val))
            node.left = build()
            node.right = build()
            return node

        return build()


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: root = [1,2,3,null,null,4,5]
    root = TreeNode(1)
    root.left = TreeNode(2)
    root.right = TreeNode(3)
    root.right.left = TreeNode(4)
    root.right.right = TreeNode(5)

    codec = Codec()

    # Serialize
    serialized = codec.serialize(root)
    print("Serialized:", serialized)
    # Output: "1,2,null,null,3,4,null,null,5,null,null,"
```

```python
# Deserialize
deserialized = codec.deserialize(serialized)

# Verify structure via level order
result = []
queue = [deserialized]
while queue:
    node = queue.pop(0)
    if node:
        result.append(node.val)
        queue.append(node.left)
        queue.append(node.right)
    else:
        result.append(None)

while result and result[-1] is None:
    result.pop()

print("Deserialized Level Order:", result)  # Output: [1,2,3,4,5]
```

Walkthrough (Example)

- Pre-order: 1 → 2 → null → null → 3 → 4 → null → null → 5 → null → null
- String: `"1,2,null,null,3,4,null,null,5,null,null,"`
- Rebuild: start from 1, then left=2, right=3, etc. → correct tree

Complexity

- **Time**: O(n) — each node processed once
- **Space**: O(n) — string and recursion stack

Problem 6: Diameter of Binary Tree

Summary

Find the length of the longest path between any two nodes (path can pass through root).

Pattern

- **Recursive DFS (Height + Global Max Path)**

Solution with Comments

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

def diameterOfBinaryTree(root):
    # Global variable to track maximum diameter
    max_diameter = 0

    def dfs(node):
        nonlocal max_diameter

        # Base case: empty node has height 0
        if not node:
            return 0

        # Get heights of left and right subtrees
        left_height = dfs(node.left)
        right_height = dfs(node.right)

        # Update max diameter: path through this node
        current_diameter = left_height + right_height
        max_diameter = max(max_diameter, current_diameter)

        # Return height of this subtree
        return max(left_height, right_height) + 1

    dfs(root)
    return max_diameter


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: root = [1,2,3,4,5]
```

```python
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

# Call function
result = diameterOfBinaryTree(root)

print("Output:", result)  # Output: 3
```

Walkthrough (Example)

- Path: $4 \to 2 \to 1 \to 3 \to$ length = 3 edges
- Or: $4 \to 2 \to 5 \to$ length = 2
- Max = 3

Complexity

- **Time**: O(n) — visit each node once
- **Space**: O(h) — recursion stack

Problem 7: Binary Tree Right Side View

Summary

Return the values visible from the right side of the tree (rightmost node at each level).

Pattern

- **BFS by Levels (take last element)**

Solution with Comments

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

from collections import deque

def rightSideView(root):
    if not root:
        return []

    result = []
    queue = deque([root])

    while queue:
        level_size = len(queue)
        # The last node in this level is the rightmost
        rightmost = None

        for _ in range(level_size):
            node = queue.popleft()
            rightmost = node.val  # Update to latest (rightmost)

            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        result.append(rightmost)

    return result


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: root = [1,2,3,null,5,null,4]
    root = TreeNode(1)
    root.left = TreeNode(2)
    root.right = TreeNode(3)
    root.left.right = TreeNode(5)
    root.right.right = TreeNode(4)
```

```python
# Call function
result = rightSideView(root)

print("Output:", result)  # Output: [1,3,4]
```

Walkthrough (Example)

- Level 0: [1] → rightmost = 1
- Level 1: [2,3] → rightmost = 3
- Level 2: [5,4] → rightmost = 4
- Result: `[1,3,4]`

Complexity

- **Time**: O(n)
- **Space**: O(w) — max width

Chunk 2: Binary Tree Problems

Core Pattern: Recursive DFS (Height Tracking)

Used in: Maximum Depth of Binary Tree

How to Recognize:

- Need height/depth of tree.
- Often recursive with `max(left_height, right_height) + 1`.
- Base case: empty node → height 0.

Step-by-Step Thinking Process (Recipe):

1. Base case: `if not root: return 0`
2. Recursively compute left and right heights.
3. Return `max(left, right) + 1`

Pitfalls & Edge Cases:

- Returning `1` for empty tree → should be `0`.
- Not handling `None` properly in recursion.

Core Pattern: Tree Reconstruction from Traversals

Used in: Construct Binary Tree from Preorder and Inorder Traversal

(Already covered in Chunk 1 — we'll apply it here again.)

Core Pattern: Path Tracking via DFS + Backtracking

Used in: Path Sum II, Path Sum III

(Already discussed — now applied to specific cases.)

Core Pattern: BFS with Index Rebalancing

Used in: Maximum Width of Binary Tree

How to Recognize:

- Need width per level; indices can grow large.
- Use index-based BFS: left child = 2*i+1, right* = 2i+2.
- Rebase indices per level to avoid overflow.

Step-by-Step Thinking Process (Recipe):

1. Start with root at index 0.
2. For each level, track min/max index.
3. Width = max - min + 1.
4. Rebase: subtract min from all indices before next level.

Pitfalls & Edge Cases:

- Not rebasing → integer overflow.
- Empty tree → width = 0.

Problem 8: Maximum Depth of Binary Tree

Summary

Return the depth of the deepest leaf node.

Pattern

- **Recursive DFS (Height)**

Solution with Comments

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

def maxDepth(root):
    # Base case: empty tree has depth 0
    if not root:
        return 0

    # Recursively find max depth of left and right subtrees
    left_depth = maxDepth(root.left)
    right_depth = maxDepth(root.right)

    # Return the greater depth + 1 for current node
    return max(left_depth, right_depth) + 1


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: root = [3,9,20,null,null,15,7]
    root = TreeNode(3)
    root.left = TreeNode(9)
    root.right = TreeNode(20)
    root.right.left = TreeNode(15)
    root.right.right = TreeNode(7)

    # Call function
    result = maxDepth(root)

    print("Output:", result)  # Output: 3
```

Walkthrough (Example)

- Node 9: depth = 1
- Node 15: depth = 2
- Node 7: depth = 2
- Node 20: max(2,2)+1 = 3
- Node 3: max(1,3)+1 = 4? Wait — no!

Wait: Actually, **node 3 is level 1**, so depth = 3.

Let's trace: - Leaf nodes (9,15,7): depth = 1 (from their parents) - Node 20: max(2,2)+1 = 3 - Node 3: max(1,3)+1 = 4?

No — correction: **the depth is number of nodes along path from root to deepest leaf**.

So: - 3 → 20 → 15 → depth = 3 - 3 → 20 → 7 → depth = 3

Thus, output is **3**

> Corrected: `maxDepth` returns 3.

Complexity

- **Time**: O(n)
- **Space**: O(h) — recursion stack

Problem 9: Construct Binary Tree from Preorder and Inorder Traversal

Summary

Given preorder and inorder traversals, reconstruct the original binary tree.

Pattern

- **Tree Reconstruction from Traversals (HashMap + Indices)**

Solution with Comments

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

def buildTree(preorder, inorder):
    # Create a map for O(1) lookup of inorder indices
    inorder_map = {val: i for i, val in enumerate(inorder)}

    # Helper function using indices
    def build(pre_start, pre_end, in_start, in_end):
        # Base case: invalid range
        if pre_start > pre_end or in_start > in_end:
            return None

        # Root is first element in preorder
        root_val = preorder[pre_start]
        root = TreeNode(root_val)

        # Find root position in inorder
        root_idx = inorder_map[root_val]

        # Number of elements in left subtree
        left_size = root_idx - in_start

        # Recursively build left and right subtrees
        root.left = build(
            pre_start + 1,           # Left starts after root
            pre_start + left_size,   # Left ends at left_size from start
            in_start,                # Left starts at same as inorder
            root_idx - 1             # Left ends just before root
        )

        root.right = build(
            pre_start + left_size + 1,  # Right starts after left part
            pre_end,
            root_idx + 1,               # Right starts after root
            in_end
        )

        return root
```

```python
    return build(0, len(preorder) - 1, 0, len(inorder) - 1)


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
    preorder = [3,9,20,15,7]
    inorder = [9,3,15,20,7]

    # Call function
    root = buildTree(preorder, inorder)

    # Verify via level order
    result = []
    queue = [root]
    while queue:
        node = queue.pop(0)
        if node:
            result.append(node.val)
            queue.append(node.left)
            queue.append(node.right)
        else:
            result.append(None)

    while result and result[-1] is None:
        result.pop()

    print("Output:", result)  # Output: [3,9,20,15,7]
```

Walkthrough (Example)

- Preorder: [3,9,20,15,7] → root = 3
- Inorder: [9,3,15,20,7] → left: [9], right: [15,20,7]
- Build left: preorder=[9], inorder=[9] → node 9
- Build right: preorder=[20,15,7], inorder=[15,20,7]
  - Root = 20
  - Left: [15], Right: [7]
- Final tree:

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

- Level order: `[3,9,20,15,7]`

Complexity

- **Time**: O(n) — each node processed once, hashmap lookup O(1)
- **Space**: O(n) — hashmap + recursion stack

Problem 10: Binary Tree Maximum Path Sum

Summary

Find the maximum sum of any path (can start/end anywhere).

Pattern

- **Recursive DFS (Max Path Through Node + Global Max)**

Solution with Comments

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

def maxPathSum(root):
    # Global variable to track maximum path sum
    max_sum = float('-inf')

    def dfs(node):
        nonlocal max_sum
```

```python
        # Base case: empty node contributes 0
        if not node:
            return 0

        # Get max path sum from left and right (only positive)
        left_gain = max(dfs(node.left), 0)
        right_gain = max(dfs(node.right), 0)

        # Current node creates a path: left + node + right
        # This is a candidate for global max
        current_path_sum = node.val + left_gain + right_gain
        max_sum = max(max_sum, current_path_sum)

        # Return max gain going up from this node (only one branch)
        return node.val + max(left_gain, right_gain)

    dfs(root)
    return max_sum


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: root = [-10,9,20,null,null,15,7]
    root = TreeNode(-10)
    root.left = TreeNode(9)
    root.right = TreeNode(20)
    root.right.left = TreeNode(15)
    root.right.right = TreeNode(7)

    # Call function
    result = maxPathSum(root)

    print("Output:", result)  # Output: 42
```

Walkthrough (Example)

- Path: 15 → 20 → 7 → sum = 42
- Or: 9 → -10 → 20 → 15 → 7 → but that's negative
- Only paths through 20 are valid
- Max = 15+20+7 = 42

Complexity

- **Time**: O(n)
- **Space**: O(h) — recursion stack

Problem 11: Path Sum II

Summary

Return all root-to-leaf paths where sum equals target.

Pattern

- **Recursive DFS with Backtracking**

Solution with Comments

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

def pathSum(root, targetSum):
    result = []

    def dfs(node, current_path, current_sum):
        # Base case: null node
        if not node:
            return

        # Add current node to path and update sum
        current_path.append(node.val)
        current_sum += node.val

        # Check if leaf and sum matches
        if not node.left and not node.right and current_sum == targetSum:
```

```python
            result.append(current_path[:])  # Deep copy

        # Recurse on children
        dfs(node.left, current_path, current_sum)
        dfs(node.right, current_path, current_sum)

        # Backtrack: remove current node
        current_path.pop()

    dfs(root, [], 0)
    return result


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Ex Input: root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22
    root = TreeNode(5)
    root.left = TreeNode(4)
    root.right = TreeNode(8)
    root.left.left = TreeNode(11)
    root.right.left = TreeNode(13)
    root.right.right = TreeNode(4)
    root.left.left.left = TreeNode(7)
    root.left.left.right = TreeNode(2)
    root.right.right.left = TreeNode(5)
    root.right.right.right = TreeNode(1)

    # Call function
    result = pathSum(root, 22)

    print("Output:", result)  # Output: [[5,4,11,2],[5,8,4,5]]
```

Walkthrough (Example)

- Path 1: 5→4→11→2 = 22
- Path 2: 5→8→4→5 = 22
- No others match.

Complexity

- **Time**: O(n^2) — up to n paths, each path length O(n)

- **Space**: O(n) — recursion depth + path storage

Problem 12: Maximum Width of Binary Tree

Summary

Find the maximum width of any level (number of nodes between leftmost and rightmost, inclusive).

Pattern

- **BFS with Index Rebalancing**

Solution with Comments

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

from collections import deque

def widthOfBinaryTree(root):
    if not root:
        return 0

    # Queue stores (node, index)
    queue = deque([(root, 0)])
    max_width = 0

    while queue:
        level_size = len(queue)
        # First and last index in current level
        first_idx = queue[0][1]
        last_idx = queue[-1][1]
```

```python
        # Update max width
        max_width = max(max_width, last_idx - first_idx + 1)

        # Process all nodes in current level
        for _ in range(level_size):
            node, idx = queue.popleft()

            # Left child: 2*idx + 1
            if node.left:
                queue.append((node.left, 2 * idx + 1))

            # Right child: 2*idx + 2
            if node.right:
                queue.append((node.right, 2 * idx + 2))

    return max_width


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: root = [1,3,2,5,3,null,9]
    root = TreeNode(1)
    root.left = TreeNode(3)
    root.right = TreeNode(2)
    root.left.left = TreeNode(5)
    root.left.right = TreeNode(3)
    root.right.right = TreeNode(9)

    # Call function
    result = widthOfBinaryTree(root)

    print("Output:", result)  # Output: 4
```

Walkthrough (Example)

- Level 0: [1] → index 0 → width = 1
- Level 1: [3,2] → indices 1,2 → width = 2
- Level 2: [5,3,9] → indices 3,4,6 → width = 6–3+1 = 4
- Max = 4

  Note: Without rebasing, indices grow fast — but we don't need to rebase because we only care about difference.

Complexity

- **Time**: O(n)
- **Space**: O(w) — max width of tree

Problem 13: Same Tree

Summary

Check if two binary trees are structurally identical and have same values.

Pattern

- **Recursive DFS (Structural Equality)**

Solution with Comments

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

def isSameTree(p, q):
    # Both nodes are None → equal
    if not p and not q:
        return True

    # One is None, other isn't → not equal
    if not p or not q:
        return False

    # Values must match, and subtrees must match
    return (p.val == q.val and
            isSameTree(p.left, q.left) and
            isSameTree(p.right, q.right))
```

```python
# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: p = [1,2,3], q = [1,2,3]
    p = TreeNode(1)
    p.left = TreeNode(2)
    p.right = TreeNode(3)

    q = TreeNode(1)
    q.left = TreeNode(2)
    q.right = TreeNode(3)

    # Call function
    result = isSameTree(p, q)

    print("Output:", result)  # Output: true
```

Walkthrough (Example)

- Compare roots: 1==1 → yes
- Left: 2==2 → yes
- Right: 3==3 → yes
- All match → return True

Complexity

- **Time**: O(n)
- **Space**: O(h)

Problem 14: Binary Tree Zigzag Level Order Traversal

Summary

Return level order traversal with alternating directions per level.

Pattern

- **BFS by Levels (Alternate Direction)**

Solution with Comments

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

from collections import deque

def zigzagLevelOrder(root):
    if not root:
        return []

    result = []
    queue = deque([root])
    left_to_right = True  # Direction flag

    while queue:
        level_size = len(queue)
        current_level = []

        for _ in range(level_size):
            node = queue.popleft()
            current_level.append(node.val)

            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        # Reverse if direction is right-to-left
        if not left_to_right:
            current_level.reverse()

        result.append(current_level)
        left_to_right = not left_to_right  # Toggle direction

    return result


# ---- Official LeetCode Example ----
```

```python
if __name__ == "__main__":
    # Example Input: root = [3,9,20,null,null,15,7]
    root = TreeNode(3)
    root.left = TreeNode(9)
    root.right = TreeNode(20)
    root.right.left = TreeNode(15)
    root.right.right = TreeNode(7)

    # Call function
    result = zigzagLevelOrder(root)

    print("Output:", result)  # Output: [[3],[20,9],[15,7]]
```

Walkthrough (Example)

- Level 0: [3] → left_to_right → [3]
- Level 1: [9,20] → reverse → [20,9]
- Level 2: [15,7] → reverse → [7,15]? No — wait: we reverse **after** collecting.

Actually: - After collecting: [15,7] → then reverse → [7,15]? But expected: [15,7]

Wait — no: the example says `[[3],[20,9],[15,7]]`

So: - Level 0: [3] - Level 1: collect [9,20] → reverse → [20,9] - Level 2: collect [15,7] → **don't reverse** → [15,7]

Direction: T → F → T → so level 2 is forward → correct.

Output: `[[3],[20,9],[15,7]]`

Complexity

- **Time**: O(n)
- **Space**: O(w)

Chunk 3: Binary Tree Problems

Core Pattern: Prefix Sum on Trees + Backtracking

Used in: Path Sum III

(Already covered — now applied to this specific case.)

Core Pattern: Tree Symmetry / Mirror Check

Used in: Symmetric Tree

How to Recognize:

- Need to check if a tree is symmetric (mirror image).
- Compare left subtree of root with right subtree (inverted).

Step-by-Step Thinking Process (Recipe):

1. Define helper: `isMirror(left, right)`
2. Base cases:
    - Both null → True
    - One null → False
3. Check:
    - Values equal
    - Left's left == Right's right
    - Left's right == Right's left

Pitfalls & Edge Cases:

- Forgetting to handle both nulls.
- Swapping left/right comparison.

Core Pattern: Parent Map + BFS (Implicit Graph)

Used in: All Nodes Distance K in Binary Tree

(Already covered — applied again.)

Core Pattern: Subtree Matching via DFS or Serialization

Used in: Subtree of Another Tree

How to Recognize:

- Check if one tree is a subtree of another.
- Can do via:
  - DFS: at each node, check if trees are identical (`isSameTree`)
  - Or serialize both and use substring search

Step-by-Step Thinking Process (Recipe):

1. Use `isSameTree` function recursively.
2. At each node, check if current subtree matches target.
3. If not, recurse on left and right.

   Alternative: serialize both trees as strings and check if one string contains the other.

Pitfalls & Edge Cases:

- Not handling empty trees correctly.
- String serialization must include nulls.

Problem 15: Path Sum III

Summary

Count the number of paths that sum to `targetSum`, where path can start anywhere and end anywhere.

Pattern

- **Prefix Sum on Trees + Backtracking**

Solution with Comments

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

from collections import defaultdict

def pathSum(root, targetSum):
    # Counter to store frequency of prefix sums
    prefix_count = defaultdict(int)
    prefix_count[0] = 1  # Empty path has sum 0

    def dfs(node, current_sum):
        if not node:
            return 0

        # Update current prefix sum
        current_sum += node.val

        # Number of valid paths ending at this node
        # i.e., how many times we've seen (current_sum - target)
        count = prefix_count[current_sum - targetSum]

        # Add this prefix sum to count
        prefix_count[current_sum] += 1

        # Recurse into children
        left_count = dfs(node.left, current_sum)
        right_count = dfs(node.right, current_sum)

        # Backtrack: remove current prefix sum
        prefix_count[current_sum] -= 1

        # Total paths = valid at this node + paths in subtrees
        return count + left_count + right_count

    return dfs(root, 0)


# ---- Official LeetCode Example ----
if __name__ == "__main__":
```

```python
# Example Input: root = [10,5,-3,3,2,null,11,3,-2,null,1], targetSum = 8
root = TreeNode(10)
root.left = TreeNode(5)
root.right = TreeNode(-3)
root.left.left = TreeNode(3)
root.left.right = TreeNode(2)
root.right.right = TreeNode(11)
root.left.left.left = TreeNode(3)
root.left.left.right = TreeNode(-2)
root.left.right.right = TreeNode(1)

# Call function
result = pathSum(root, 8)

print("Output:", result)  # Output: 3
```

Walkthrough (Example)

- Paths:
  1. 5 → 3 → 0? No
  2. 5 → 2 → 1 → 8?
  3. 10 → 5 → 3 → 8? Yes
  4. 10 → -3 → 11 → 8? Yes
  5. 5 → 3 → -2 → 1 → 8? Yes
- Actually: three paths match → output 3

Complexity

- **Time**: O(n)
- **Space**: O(h) — recursion + hash map

Problem 16: Symmetric Tree

Summary

Check if a binary tree is symmetric around its center (mirror image).

Pattern

- **Recursive DFS (Mirror Check)**

Solution with Comments

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

def isSymmetric(root):
    def isMirror(left, right):
        # Both null → symmetric
        if not left and not right:
            return True

        # One null, other not → not symmetric
        if not left or not right:
            return False

        # Values must match, and subtrees must be mirrors
        return (left.val == right.val and
                isMirror(left.left, right.right) and
                isMirror(left.right, right.left))

    return isMirror(root, root)


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: root = [1,2,2,3,4,4,3]
    root = TreeNode(1)
    root.left = TreeNode(2)
    root.right = TreeNode(2)
    root.left.left = TreeNode(3)
    root.left.right = TreeNode(4)
    root.right.left = TreeNode(4)
    root.right.right = TreeNode(3)
```

```
# Call function
result = isSymmetric(root)

print("Output:", result)  # Output: true
```

Walkthrough (Example)

- Root: compare left and right
- Left: 2 → left=3, right=4
- Right: 2 → left=4, right=3
- Compare: 3==3? No — wait: actually:
  - `isMirror(left.left, right.right)` → 3 vs 3 → True
  - `isMirror(left.right, right.left)` → 4 vs 4 → True
- So yes → symmetric

Complexity

- **Time**: O(n)
- **Space**: O(h)

Problem 17: All Nodes Distance K in Binary Tree

Summary

Given a binary tree, a target node, and distance k, return all nodes at distance k.

Pattern

- **Parent Map + BFS (Implicit Graph)**

Solution with Comments

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

from collections import deque

def distanceK(root, target, k):
    # Build parent map via DFS
    parent_map = {}

    def build_parent(node, parent):
        if not node:
            return
        parent_map[node] = parent
        build_parent(node.left, node)
        build_parent(node.right, node)

    build_parent(root, None)

    # BFS from target node
    queue = deque([(target, 0)])
    visited = {target}
    result = []

    while queue:
        node, dist = queue.popleft()

        if dist == k:
            result.append(node.val)
            continue  # Don't go further

        # Explore neighbors: left, right, 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
```

```python
# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, k = 2
    root = TreeNode(3)
    root.left = TreeNode(5)
    root.right = TreeNode(1)
    root.left.left = TreeNode(6)
    root.left.right = TreeNode(2)
    root.right.left = TreeNode(0)
    root.right.right = TreeNode(8)
    root.left.right.left = TreeNode(7)
    root.left.right.right = TreeNode(4)

    target = root.left  # node 5
    k = 2

    # Call function
    result = distanceK(root, target, k)

    print("Output:", result)  # Output: [7,4,1]
```

Walkthrough (Example)

- From node 5:
  - Level 0: [5]
  - Level 1: [3, 2, 6] (parent, children)
  - Level 2: [7,4,1] (children of 2 and parent of 3)
- So nodes at distance 2: 7,4,1

Complexity

- **Time**: O(n)
- **Space**: O(n)

Problem 18: Subtree of Another Tree

Summary

Check if `t` is a subtree of `s`.

Pattern

- **DFS + isSameTree** or **Serialization**

Solution with Comments (DFS Approach)

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

def isSubtree(s, t):
    def isSameTree(p, q):
        if not p and not q:
            return True
        if not p or not q:
            return False
        return (p.val == q.val and
                isSameTree(p.left, q.left) and
                isSameTree(p.right, q.right))

    if not s:
        return False

    # Check if t is subtree rooted at s
    if isSameTree(s, t):
        return True

    # Else check left and right subtrees
    return isSubtree(s.left, t) or isSubtree(s.right, t)


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: s = [3,4,5,1,2], t = [4,1,2]
    s = TreeNode(3)
    s.left = TreeNode(4)
    s.right = TreeNode(5)
    s.left.left = TreeNode(1)
    s.left.right = TreeNode(2)
```

```python
t = TreeNode(4)
t.left = TreeNode(1)
t.right = TreeNode(2)

# Call function
result = isSubtree(s, t)

print("Output:", result)  # Output: true
```

Walkthrough (Example)

- At root 3: `isSameTree(3,4)` → no
- Go to left child (4): `isSameTree(4,4)` → yes
- Then check children: 1==1, 2==2 → yes
- Return True

Complexity

- **Time**: O(m × n) — m = size of s, n = size of t
- **Space**: O(h) — recursion stack