Binary Tree

1. Same Tree

Pattern: Tree Traversal (Recursion / DFS)

Problem Statement

> Given the roots of two binary trees `p` and `q`, write a function to check if they are the same or not.
> Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

Sample Input & Output

```
Input: p = [1,2,3], q = [1,2,3]
Output: true
Explanation: Both trees have identical structure and node values.
```

```
Input: p = [1,2], q = [1,null,2]
Output: false
Explanation: Left child of p is 2;
q has no left child but a right child → structural mismatch.
```

```
Input: p = [], q = []
Output: true
Explanation: Both trees are empty - considered identical.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import Optional

# 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:
        # STEP 1: Base cases - both nodes are None → identical
        if not p and not q:
            return True

        # STEP 2: One is None, the other isn't → not identical
        if not p or not q:
            return False

        # STEP 3: Values differ → not identical
        if p.val != q.val:
            return False

        # STEP 4: Recursively check left and right subtrees
        #   - Both must be identical for whole tree to match
        return (self.isSameTree(p.left, q.left) and
                self.isSameTree(p.right, q.right))

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case - identical trees
    p1 = TreeNode(1, TreeNode(2), TreeNode(3))
    q1 = TreeNode(1, TreeNode(2), TreeNode(3))
    assert sol.isSameTree(p1, q1) == True
```

```python
#  Test 2: Edge case - both empty
assert sol.isSameTree(None, None) == True

#  Test 3: Tricky case - same values, different structure
p3 = TreeNode(1, TreeNode(2))
q3 = TreeNode(1, None, TreeNode(2))
assert sol.isSameTree(p3, q3) == False

print(" All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace **Test 3**:
`p3 = [1,2]` (2 is left child), `q3 = [1,null,2]` (2 is right child)

1. **Call `isSameTree(p3, q3)`**
   - `p3` and `q3` both exist → skip first two base cases.
   - `p3.val == 1`, `q3.val == 1` → values match.
   - Now check: `isSameTree(p3.left, q3.left)` and `isSameTree(p3.right, q3.right)`
2. **Left subtree call: `isSameTree(TreeNode(2), None)`**
   - `p = TreeNode(2)`, `q = None`
   - One is `None`, the other isn't → return `False`
3. **Right subtree is never evaluated** due to short-circuiting (`and` stops at first `False`)
4. **Final result**: `False` → correctly detects structural difference.

State snapshots:
- Initial: `p=1(L:2,R:None)`, `q=1(L:None,R:2)`
- After value check: proceed to children
- Left comparison: `(2 vs None)` → mismatch → return `False`

Complexity Analysis

- **Time Complexity**: `O(min(m, n))`

  > We visit each node once until a mismatch or full traversal.
  > In worst case (identical trees), we visit all nodes in the smaller tree.

- **Space Complexity**: `O(min(m, n))`

  > Due to recursion stack depth, which equals the height of the smaller tree.
  > Worst case: skewed tree → height   number of nodes → linear space.

2. Symmetric Tree

Pattern: Tree Traversal (Recursive Mirror Comparison)

Problem Statement

> Given the `root` of a binary tree, check whether it is a mirror of itself (i.e., symmetric around its center).

Sample Input & Output

```
Input: root = [1,2,2,3,4,4,3]
Output: true
Explanation: The left and right subtrees are mirror images.
```

```
Input: root = [1,2,2,null,3,null,3]
Output: false
Explanation: Left subtree has [2,null,3], right has [2,null,3] - but structure isn't mirrore
```

```
Input: root = [1]
Output: true
Explanation: Single node is trivially symmetric.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import Optional

# 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:
        # STEP 1: Handle empty tree
        #   - Empty tree is symmetric by definition
        if not root:
            return True

        # STEP 2: Define helper to compare two subtrees as mirrors
        #   - Recursively checks if left subtree of A mirrors right of B
        def is_mirror(left: Optional[TreeNode],
                      right: Optional[TreeNode]) -> bool:
            # Both nodes are None → symmetric at this branch
            if not left and not right:
                return True
            # One is None, other isn't → asymmetric
            if not left or not right:
                return False
            # Values must match AND:
            #   left's left mirrors right's right
            #   left's right mirrors right's left
            return (left.val == right.val and
                    is_mirror(left.left, right.right) and
                    is_mirror(left.right, right.left))
```

```python
        # STEP 3: Start mirror check on root's children
        #   - Root itself is center; compare its left & right
        return is_mirror(root.left, root.right)

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #   Test 1: Normal symmetric tree
    #      1
    #     / \
    #    2   2
    #   / \ / \
    # 3  4 4  3
    root1 = TreeNode(1)
    root1.left = TreeNode(2)
    root1.right = TreeNode(2)
    root1.left.left = TreeNode(3)
    root1.left.right = TreeNode(4)
    root1.right.left = TreeNode(4)
    root1.right.right = TreeNode(3)
    assert sol.isSymmetric(root1) == True

    #   Test 2: Edge case - single node
    root2 = TreeNode(1)
    assert sol.isSymmetric(root2) == True

    #   Test 3: Tricky asymmetric tree
    #      1
    #     / \
    #    2   2
    #     \   \
    #      3   3
    root3 = TreeNode(1)
    root3.left = TreeNode(2)
    root3.right = TreeNode(2)
    root3.left.right = TreeNode(3)
    root3.right.right = TreeNode(3)
    assert sol.isSymmetric(root3) == False

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 1** (`[1,2,2,3,4,4,3]`):

1. **Call `isSymmetric(root1)`**
   - `root1` exists → skip empty check.
   - Call `is_mirror(root1.left, root1.right)` → `is_mirror(node2a, node2b)`.
2. **First `is_mirror` call**:
   - `left = node2a (val=2)`, `right = node2b (val=2)`
   - Both exist → check `2 == 2` → `True`.
   - Recurse:
     - `is_mirror(node2a.left=node3, node2b.right=node3)`
     - `is_mirror(node2a.right=node4, node2b.left=node4)`
3. **Check left pair (`node3`, `node3`)**:
   - Both exist, `3 == 3` → `True`.
   - Recurse on their children → both `(None, None)` → return `True`.
4. **Check right pair (`node4`, `node4`)**:
   - Same logic → returns `True`.
5. **Combine results**: `True and True and True` → `True`.

Final output: `True`.

Complexity Analysis

- **Time Complexity**: `O(n)`

  We visit every node exactly once in the worst case (fully symmetric or asymmetric at leaves).

- **Space Complexity**: `O(h)`

  Recursion depth equals tree height `h`. In worst case (skewed tree), `h = n`; in balanced tree, `h = log n`.

3. Maximum Depth of Binary Tree

Pattern: Tree Traversal (DFS / Recursion)

Problem Statement

Given the `root` of a binary tree, return *its maximum depth*.

A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.

Sample Input & Output

```
Input: root = [3,9,20,null,null,15,7]
Output: 3
Explanation: Longest path is 3 → 20 → 15 (or 7), 3 nodes deep.
```

```
Input: root = [1,null,2]
Output: 2
Explanation: Only right child exists; depth = 2.
```

```
Input: root = []
Output: 0
Explanation: Empty tree has depth 0.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import Optional

# 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:
        # STEP 1: Base case - empty node contributes 0 depth
        #   - Recursion stops here; prevents infinite calls
        if not root:
            return 0

        # STEP 2: Recursively compute depth of left & right subtrees
        #   - Each call explores one branch fully (DFS)
        #   - We trust recursion to return correct subtree depth
        left_depth = self.maxDepth(root.left)
        right_depth = self.maxDepth(root.right)

        # STEP 3: Current node adds 1 to the deeper subtree
        #   - Ensures we count this node in the path
        #   - Max picks the longer of the two paths
        return 1 + max(left_depth, right_depth)

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case - balanced-ish tree
    # Tree: [3,9,20,null,null,15,7]
    root1 = TreeNode(3)
    root1.left = TreeNode(9)
    root1.right = TreeNode(20)
    root1.right.left = TreeNode(15)
```

```python
    root1.right.right = TreeNode(7)
    assert sol.maxDepth(root1) == 3, "Test 1 failed"

    #   Test 2: Edge case - skewed tree (only right children)
    # Tree: [1,null,2]
    root2 = TreeNode(1)
    root2.right = TreeNode(2)
    assert sol.maxDepth(root2) == 2, "Test 2 failed"

    #   Test 3: Tricky/negative - empty tree
    root3 = None
    assert sol.maxDepth(root3) == 0, "Test 3 failed"

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace **Test 1**: `root = [3,9,20,null,null,15,7]`.

We call `sol.maxDepth(root1)` where `root1.val = 3`.

1. **Call 1**: `maxDepth(3)`
    - `root` exists → skip base case.
    - Compute `left_depth = maxDepth(9)`
    - Compute `right_depth = maxDepth(20)`
    - Will return `1 + max(left, right)`
2. **Call 2**: `maxDepth(9)` (left child of 3)
    - Node `9` exists.
    - `maxDepth(9.left)` → `maxDepth(None)` → returns `0`
    - `maxDepth(9.right)` → `maxDepth(None)` → returns `0`

- Returns `1 + max(0, 0) = 1`

3. **Call 3**: `maxDepth(20)` (right child of 3)
   - Node 20 exists.
   - `left_depth = maxDepth(15)`
   - `right_depth = maxDepth(7)`
4. **Call 4**: `maxDepth(15)`
   - Both children are `None` → returns `1`
5. **Call 5**: `maxDepth(7)`
   - Both children are `None` → returns `1`
6. Back to **Call 3**: `maxDepth(20)`
   - `left_depth = 1`, `right_depth = 1`
   - Returns `1 + max(1,1) = 2`
7. Back to **Call 1**: `maxDepth(3)`
   - `left_depth = 1`, `right_depth = 2`
   - Returns `1 + max(1,2) = 3`

Final output: `3`

Key Insight: Each node waits for its children to report their depths, then adds itself (hence `+1`). The recursion naturally explores all paths and picks the longest.

Complexity Analysis

- **Time Complexity**: `O(n)`

  We visit every node exactly once. In the worst case (skewed tree), recursion depth is `n`, but total work is still proportional to number of nodes.

- **Space Complexity**: `O(h)`

  Where `h` is the height of the tree. This is due to the recursion stack.

– Best case (balanced): `O(log n)`

– Worst case (skewed): `O(n)`

4. Binary Tree Level Order Traversal

Pattern: BFS (Breadth-First Search) / Level-Order Traversal

Problem Statement

> Given the `root` of a binary tree, return *the level order traversal of its nodes' values*. (i.e., from left to right, level by level).

Sample Input & Output

```
Input: root = [3,9,20,null,null,15,7]
Output: [[3],[9,20],[15,7]]
Explanation: Level 0: [3], Level 1: [9,20], Level 2: [15,7]
```

```
Input: root = [1]
Output: [[1]]
Explanation: Single node tree.
```

```
Input: root = []
Output: []
Explanation: Empty tree returns empty list.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List, Optional
from collections import deque

# 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 levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        # STEP 1: Initialize structures
        #   - Use deque for efficient popleft (FIFO queue)
        #   - Result list stores levels as sublists
        if not root:
            return []

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

        # STEP 2: Main loop / recursion
        #   - Process all nodes at current level before moving deeper
        #   - Level size = len(queue) at start of each iteration
        while queue:
            level_size = len(queue)
            current_level = []

            # STEP 3: Update state / bookkeeping
            #   - Dequeue each node in current level
            #   - Append its value and enqueue children
            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)

            result.append(current_level)
```

```python
        # STEP 4: Return result
        #   - Already handles empty tree via early return
        return result

# -------------------- INLINE TESTS --------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    # Tree: [3,9,20,null,null,15,7]
    root1 = TreeNode(3)
    root1.left = TreeNode(9)
    root1.right = TreeNode(20)
    root1.right.left = TreeNode(15)
    root1.right.right = TreeNode(7)
    print(sol.levelOrder(root1))  # [[3], [9, 20], [15, 7]]

    #  Test 2: Edge case - single node
    root2 = TreeNode(1)
    print(sol.levelOrder(root2))  # [[1]]

    #  Test 3: Tricky/negative - empty tree
    print(sol.levelOrder(None))   # []
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 1**: `root = [3,9,20,null,null,15,7]`.

Initial state:
- `queue = deque([3])`
- `result = []`

Level 0 (root level): - `level_size = 1` - Loop runs once: - Pop `3` → `current_level = [3]` - Add left (9) and right (20) to queue → `queue = [9, 20]` - Append `[3]` to `result` → `result = [[3]]`

Level 1: - `level_size = 2` - First iteration: - Pop 9 → `current_level = [9]` - No children → queue becomes `[20]` - Second iteration: - Pop 20 → `current_level = [9, 20]` - Add 15 and 7 → queue = `[15, 7]` - Append `[9,20]` → `result = [[3], [9,20]]`

Level 2: - `level_size = 2` - Pop 15 → `current_level = [15]`; no children - Pop 7 → `current_level = [15,7]`; no children - Append → `result = [[3], [9,20], [15,7]]`

Queue empty → exit loop → return result.

Final output: `[[3], [9, 20], [15, 7]]`

Key insight: **BFS with level-by-level processing** using queue size snapshot.

Complexity Analysis

- **Time Complexity**: `O(n)`

  Each node is visited exactly once. All operations inside the loop (append, popleft) are O(1). Total = O(n).

- **Space Complexity**: `O(n)`

  In worst case (complete binary tree), the queue holds up to ~n/2 nodes (last level). Output list also stores n values. Thus, O(n).

5. Binary Tree Zigzag Level Order Traversal

Pattern: BFS (Level Order Traversal) + Directional Toggle

Problem Statement

> Given the `root` of a binary tree, return *the zigzag level order traversal of its nodes' values.* (i.e., from left to right, then right to left for the next level and alternate between).

Sample Input & Output

```
Input: root = [3,9,20,null,null,15,7]
Output: [[3],[20,9],[15,7]]
Explanation: Level 0: [3] (L→R), Level 1: [9,20] reversed → [20,9], Level 2: [15,7] (L→R)
```

```
Input: root = [1]
Output: [[1]]
Explanation: Single node - only one level.
```

```
Input: root = []
Output: []
Explanation: Empty tree returns empty list.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List, Optional
from collections import deque

# 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 zigzagLevelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        # STEP 1: Initialize structures
        #   - Use deque for efficient BFS
        #   - 'result' stores final levels
        #   - 'left_to_right' toggles direction per level
        if not root:
            return []

        queue = deque([root])
```

```
        result = []
        left_to_right = True

        # STEP 2: Main loop / recursion
        #   - Process level-by-level using queue size
        #   - Maintain invariant: queue holds all nodes of current level
        while queue:
            level_size = len(queue)
            level_nodes = []

            # STEP 3: Update state / bookkeeping
            #   - Collect node values in order
            #   - Reverse if direction is right-to-left
            for _ in range(level_size):
                node = queue.popleft()
                level_nodes.append(node.val)

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

            # Reverse level if needed before appending
            if not left_to_right:
                level_nodes.reverse()
            result.append(level_nodes)

            # Toggle direction for next level
            left_to_right = not left_to_right

        # STEP 4: Return result
        #   - Handles all cases including empty root
        return result

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    # Tree: [3,9,20,null,null,15,7]
    root1 = TreeNode(3)
    root1.left = TreeNode(9)
```

```python
    root1.right = TreeNode(20)
    root1.right.left = TreeNode(15)
    root1.right.right = TreeNode(7)
    assert sol.zigzagLevelOrder(root1) == [[3],[20,9],[15,7]]

    #  Test 2: Edge case - single node
    root2 = TreeNode(1)
    assert sol.zigzagLevelOrder(root2) == [[1]]

    #  Test 3: Tricky/negative - empty tree
    assert sol.zigzagLevelOrder(None) == []

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 1**: `root = [3,9,20,null,null,15,7]`

Initial State:
- `queue = [3]`
- `result = []`
- `left_to_right = True`

Level 0 (left_to_right = True):
- `level_size = 1`
- Process node `3`:
- Append `3` to `level_nodes` → `[3]`
- Enqueue children: `9`, `20` → `queue = [9, 20]`
- Since direction is L→R, **don't reverse** → append `[3]` to `result`
- Toggle direction → `left_to_right = False`
- **State**: `result = [[3]]`, `queue = [9, 20]`

Level 1 (left_to_right = False):
- `level_size = 2`
- Process node `9`:
- `level_nodes = [9]`
- No children → queue becomes `[20]`
- Process node `20`:
- `level_nodes = [9, 20]`
- Enqueue `15`, `7` → `queue = [15, 7]`
- Direction is R→L → **reverse** → `[20, 9]`
- Append to `result` → `[[3], [20,9]]`
- Toggle direction → `left_to_right = True`

Level 2 (left_to_right = True):
- `level_size = 2`
- Process `15`: `level_nodes = [15]`, no children
- Process `7`: `level_nodes = [15, 7]`, no children
- Direction L→R → **no reverse** → append `[15,7]`
- Queue empty → loop ends
- **Final result**: `[[3], [20,9], [15,7]]`

Matches expected output!

Complexity Analysis

- **Time Complexity**: `O(N)`

  We visit each node exactly once. Reversing a level takes `O(k)` for `k` nodes in that level, and sum of all `k` is `N`. So total is still linear.

- **Space Complexity**: `O(N)`

  The queue holds at most the width of the tree (N/2 nodes in worst case, e.g., complete binary tree). The output list also stores `N` values. Thus, `O(N)`.

6. Binary Tree Right Side View

Pattern: Tree BFS (Level-order Traversal)

Problem Statement

Given the `root` of a binary tree, imagine yourself standing on the right side of it. Return the values of the nodes you can see ordered from top to bottom.

Sample Input & Output

```
Input: root = [1,2,3,null,5,null,4]
Output: [1,3,4]
Explanation: From the right, you see node 1 (level 0), node 3 (level 1), and node 4 (level 2)
```

```
Input: root = [1,null,3]
Output: [1,3]
Explanation: Only right children exist beyond root.
```

```
Input: root = []
Output: []
Explanation: Empty tree → nothing to see.
```

LeetCode Editorial Solution + Inline Tests

```
from typing import List, Optional
from collections import deque

# 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 rightSideView(self, root: Optional[TreeNode]) -> List[int]:
```

```
        # STEP 1: Initialize structures
        #   - Use a queue for BFS (level-order traversal)
        #   - Result list stores rightmost node per level
        if not root:
            return []

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

        # STEP 2: Main loop / recursion
        #   - Process level by level using queue size
        #   - The last node in each level is the rightmost
        while queue:
            level_size = len(queue)

            for i in range(level_size):
                node = queue.popleft()

                # STEP 3: Update state / bookkeeping
                #   - Only record the last node in the level
                if i == level_size - 1:
                    right_view.append(node.val)

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

        # STEP 4: Return result
        #   - Already built during traversal
        return right_view

# -------------------- INLINE TESTS --------------------
if __name__ == "__main__":
    sol = Solution()

    #   Test 1: Normal case
    # Tree: [1,2,3,null,5,null,4]
    root1 = TreeNode(1)
    root1.left = TreeNode(2)
    root1.right = TreeNode(3)
```

```python
    root1.left.right = TreeNode(5)
    root1.right.right = TreeNode(4)
    assert sol.rightSideView(root1) == [1, 3, 4], "Test 1 Failed"

    #  Test 2: Edge case - only right children
    root2 = TreeNode(1)
    root2.right = TreeNode(3)
    assert sol.rightSideView(root2) == [1, 3], "Test 2 Failed"

    #  Test 3: Tricky/negative - empty tree
    assert sol.rightSideView(None) == [], "Test 3 Failed"

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 1**: `root = [1,2,3,null,5,null,4]`

Initial state:
- `queue = [1]`
- `right_view = []`

Level 0 (root level):
- `level_size = 1`
- Loop `i = 0` (only node):
- `node = 1`
- Since `i == 0 == level_size - 1`, append 1 → `right_view = [1]`
- Enqueue left (2) and right (3) → `queue = [2, 3]`

Level 1:
- `level_size = 2`
- `i = 0`: `node = 2`
- Not last → skip adding to result
- Enqueue its right child 5 → `queue = [3, 5]`
- `i = 1`: `node = 3`
- Last in level → append 3 → `right_view = [1, 3]`
- Enqueue its right child 4 → `queue = [5, 4]`

Level 2:
- `level_size = 2`
- `i = 0`: `node = 5`
- Not last → skip
- No children → queue becomes `[4]`
- `i = 1`: `node = 4`
- Last → append `4` → `right_view = [1, 3, 4]`
- No children → queue empty

Loop ends → return `[1, 3, 4]`

Final output matches expected.

Complexity Analysis

- **Time Complexity**: `O(n)`

  We visit every node exactly once in BFS. `n` = number of nodes.

- **Space Complexity**: `O(w)`

  `w` = maximum width of the tree (stored in queue at one level).
  In worst case (complete tree), `w` `n/2` → still `O(n)`, but typically much less.