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
```

```
#  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 mirrored.
```

```
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))
        # 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)
    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)
        # 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)
    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

```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 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)
    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.

7. Path Sum II

Pattern: Tree DFS (Backtracking)

Problem Statement

> Given the `root` of a binary tree and an integer `targetSum`, return all root-to-leaf paths where the sum of the node values in the path equals `targetSum`. Each path should be returned as a list of the node values, not node references.
> A **root-to-leaf path** starts at the root and ends at a leaf node (a node with no children).

Sample Input & Output

```
Input: root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22
Output: [[5,4,11,2],[5,8,4,5]]
Explanation: Two paths sum to 22: 5→4→11→2 and 5→8→4→5.
```

```
Input: root = [1,2,3], targetSum = 5
Output: []
Explanation: No root-to-leaf path sums to 5.
```

```
Input: root = [1,2], targetSum = 1
Output: []
Explanation: The only leaf is 2; path [1,2] sums to 3   1.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List, 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 pathSum(self, root: Optional[TreeNode], targetSum: int):
        # STEP 1: Initialize result list and current path
        #   - `result` stores all valid paths
        #   - `path` tracks current DFS traversal (mutable list)
        result = []
        path = []

        def dfs(node, curr_sum):
            # Base case: empty node → stop recursion
            if not node:
                return

            # STEP 2: Include current node in path & sum
            path.append(node.val)
            curr_sum += node.val

            # STEP 3: Check if leaf and sum matches target
            #   - Leaf: no left/right children
            #   - If match, add *copy* of path to result
            if not node.left and not node.right:
                if curr_sum == targetSum:
                    result.append(list(path))  # shallow copy

            # STEP 4: Recurse on children (backtrack after)
            dfs(node.left, curr_sum)
            dfs(node.right, curr_sum)

            # STEP 5: Backtrack - remove current node before
            #         returning to parent (restore path state)
            path.pop()

        dfs(root, 0)
        return result

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

    #  Test 1: Normal case
    # Tree: [5,4,8,11,null,13,4,7,2,null,null,5,1]
    root1 = TreeNode(5)
    root1.left = TreeNode(4)
    root1.right = TreeNode(8)
    root1.left.left = TreeNode(11)
    root1.left.left.left = TreeNode(7)
    root1.left.left.right = TreeNode(2)
    root1.right.left = TreeNode(13)
    root1.right.right = TreeNode(4)
    root1.right.right.left = TreeNode(5)
    root1.right.right.right = TreeNode(1)
    print(sol.pathSum(root1, 22))  # [[5,4,11,2],[5,8,4,5]]
```

```
#   Test 2: Edge case - no valid path
root2 = TreeNode(1)
root2.left = TreeNode(2)
root2.right = TreeNode(3)
print(sol.pathSum(root2, 5))   # []

#   Test 3: Tricky/negative - target too small
root3 = TreeNode(1)
root3.left = TreeNode(2)
print(sol.pathSum(root3, 1))   # []
```

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

Example Walkthrough

We'll trace **Test 1** with `targetSum = 22`.

1. **Start**: `result = []`, `path = []`, call `dfs(root=5, curr_sum=0)`.
2. **Node 5**:
   - Append 5 → `path = [5]`
   - `curr_sum = 0 + 5 = 5`
   - Not a leaf → recurse left (4)
3. **Node 4**:
   - Append 4 → `path = [5,4]`
   - `curr_sum = 5 + 4 = 9`
   - Not a leaf → recurse left (11)
4. **Node 11**:
   - Append 11 → `path = [5,4,11]`
   - `curr_sum = 9 + 11 = 20`
   - Not a leaf → recurse left (7)

5. **Node 7**:
    - Append 7 → `path = [5,4,11,7]`
    - `curr_sum = 20 + 7 = 27`
    - Leaf? Yes. 27   22 → skip.
    - **Backtrack**: `path.pop()` → `path = [5,4,11]`
6. **Node 2** (right of 11):
    - Append 2 → `path = [5,4,11,2]`
    - `curr_sum = 20 + 2 = 22`
    - Leaf? Yes. Match! → `result.append([5,4,11,2])`
    - **Backtrack**: pop → `path = [5,4,11]`
7. Back to **Node 11**: both children done → pop → `path = [5,4]`
8. Back to **Node 4**: no right child → pop → `path = [5]`
9. Now recurse **right** from root: **Node 8**
    - Append 8 → `path = [5,8]`, sum = 13
    - Recurse left (13) → leaf, sum=26 22 → backtrack
    - Recurse right (4) → `path=[5,8,4]`, sum=17
        - Left child 5: leaf, sum=22 → add `[5,8,4,5]`
        - Right child 1: leaf, sum=18 → skip
    - Backtrack fully
10. Final `result = [[5,4,11,2], [5,8,4,5]]`

Key insight: **backtracking** ensures `path` always reflects the current root-to-node route.

Complexity Analysis

- **Time Complexity**: $O(N^2)$ in worst case

We visit every node once (`O(N)`), but copying the path (of length `O(H)`, up to `O(N)` in skewed tree) for each leaf leads to `O(N²)` total. In balanced trees, it's closer to `O(N log N)`.

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

  The `path` list uses `O(H)` space (height of tree). Recursion stack also uses `O(H)`. In worst case (skewed tree), `H = N`. Output storage is extra and not counted in auxiliary space.

8. Path Sum III

Pattern: Prefix Sum + Hash Map (Cumulative Sum Tracking)

Problem Statement

Given the `root` of a binary tree and an integer `targetSum`, return the number of paths where the sum of the values along the path equals `targetSum`.

The path does **not** need to start or end at the root or a leaf, but it must go **downwards** (traveling only from parent nodes to child nodes).

Sample Input & Output

```
Input: root = [10,5,-3,3,2,null,11,3,-2,null,1], targetSum = 8
Output: 3
Explanation: The paths that sum to 8 are:
  1. 5 -> 3
  2. 5 -> 2 -> 1
  3. -3 -> 11
```

```
Input: root = [1,null,2,null,3], targetSum = 3
Output: 2
Explanation:
  1. 1 -> 2
  2. 3 (standalone node)
```

```
Input: root = [], targetSum = 0
Output: 0
Explanation: Empty tree → no paths.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import Optional
from collections import defaultdict

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

class Solution:
    def pathSum(self, root: Optional[TreeNode], targetSum: int) -> int:
        # STEP 1: Initialize structures
        #   - prefix_sum_count tracks how many times a cumulative sum
        #     has occurred in the current DFS path.
        #   - We seed it with {0: 1} to handle paths starting at root.
        prefix_sum_count = defaultdict(int)
        prefix_sum_count[0] = 1

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

            # STEP 2: Main loop / recursion
            #   - Update current cumulative sum with node value.
            curr_sum += node.val

            #   - Check if (curr_sum - targetSum) exists in map.
            #     If yes, there's a subpath ending here that sums to target.
            count = prefix_sum_count[curr_sum - targetSum]

            # STEP 3: Update state / bookkeeping
            #   - Record current sum before recursing to children.
            prefix_sum_count[curr_sum] += 1

            # Recurse left and right
            count += dfs(node.left, curr_sum)
            count += dfs(node.right, curr_sum)

            # Backtrack: remove current sum from map after returning
            #   - Critical! Prevents cross-branch contamination.
            prefix_sum_count[curr_sum] -= 1

            # STEP 4: Return result
            return count

        return dfs(root, 0)

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

    #  Test 1: Normal case
    # Tree: [10,5,-3,3,2,null,11,3,-2,null,1]
    root1 = TreeNode(10)
    root1.left = TreeNode(5)
    root1.right = TreeNode(-3)
    root1.left.left = TreeNode(3)
    root1.left.right = TreeNode(2)
    root1.right.right = TreeNode(11)
    root1.left.left.left = TreeNode(3)
    root1.left.left.right = TreeNode(-2)
    root1.left.right.right = TreeNode(1)
    assert sol.pathSum(root1, 8) == 3

    #  Test 2: Edge case - single path
    root2 = TreeNode(1)
    root2.right = TreeNode(2)
    root2.right.right = TreeNode(3)
    assert sol.pathSum(root2, 3) == 2

    #  Test 3: Tricky/negative - empty tree
    assert sol.pathSum(None, 0) == 0

    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** with `targetSum = 8` on the tree:

```
      10
     /  \
    5   -3
   / \    \
  3   2    11
 / \   \
3  -2   1
```

Goal: Count all downward paths summing to 8.

Step 1: Initialize `prefix_sum_count = {0: 1}` and call `dfs(root=10, curr_sum=0)`.

- `curr_sum = 0 + 10 = 10`
- Check: `10 - 8 = 2` → Is 2 in map? No → `count = 0`
- Update map: `{0:1, 10:1}`
- Recurse left → node `5`

Step 2: At node `5` (`curr_sum = 10` from parent)

- `curr_sum = 10 + 5 = 15`
- Check: `15 - 8 = 7` → not in map → `count = 0`
- Map: `{0:1, 10:1, 15:1}`
- Recurse left → node `3`

Step 3: At node `3`

- `curr_sum = 15 + 3 = 18`
- Check: `18 - 8 = 10` → **YES!** `prefix_sum_count[10] = 1` → `count = 1`
  - This means: path from after sum=10 (i.e., node 10) to current node sums to 8 → `5 → 3`

- Map: `{..., 18:1}`
- Recurse to node `3` (leaf)

Step 4: At leaf `3` (leftmost)

- `curr_sum = 18 + 3 = 21`
- `21 - 8 = 13` → not in map → count = 0
- Map updated → then backtrack: remove `21`
- Return 0

Back to node `3` → now recurse to `-2`:

- `curr_sum = 18 + (-2) = 16`
- `16 - 8 = 8` → not in map → count = 0
- Backtrack → remove `16`

Back to node `3` → done. Backtrack: remove `18`

Now back at node `5` → recurse right to node `2`

- `curr_sum = 15 + 2 = 17`
- `17 - 8 = 9` → not in map → count = 0
- Map: `{..., 17:1}`
- Recurse to node `1`:
  - `curr_sum = 17 + 1 = 18`
  - `18 - 8 = 10` → **YES!** → count += 1 → total = 1 (from here)
    - Path: `5 → 2 → 1` (since sum=10 occurred at root, subtract → 18–10=8)
- Backtrack `18`, then `17`

Now back at root `10` → go right to `-3`

- `curr_sum = 10 + (-3) = 7`
- `7 - 8 = -1` → not in map
- Map: `{..., 7:1}`
- Recurse to `11`:
  - `curr_sum = 7 + 11 = 18`
  - `18 - 8 = 10` → **YES!** → count += 1
    - Path: `-3 → 11` (18 – 10 = 8)

Total count = 1 (5→3) + 1 (5→2→1) + 1 (-3→11) = **3**

Backtracking ensures sums from left subtree don't pollute right subtree.

Complexity Analysis

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

  We visit each node exactly once during DFS. Hash map operations (`get`, `set`) are `O(1)` average.

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

  In worst case (skewed tree), recursion depth is `O(N)`. The hash map also stores up to `O(N)` prefix sums.

9. Diameter of Binary Tree

Pattern: Tree DFS (Postorder Traversal)

Problem Statement

Given the `root` of a binary tree, return *the length of the diameter of the tree.*
The **diameter** of a binary tree is the *length of the longest path between any two nodes* in a tree. This path may or may not pass through the root.
The **length** of a path between two nodes is represented by the number of **edges** between them.

Sample Input & Output

```
Input: root = [1,2,3,4,5]
Output: 3
Explanation: The longest path is [4,2,1,3] or [5,2,1,3] → 3 edges.
```

```
Input: root = [1,2]
Output: 1
Explanation: Only one edge between 1 and 2.
```

```
Input: root = []
Output: 0
Explanation: Empty tree has no edges.
```

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 diameterOfBinaryTree(self, root: Optional[TreeNode]) -> int:
        # STEP 1: Initialize max diameter as instance variable
        #   - We track the global max across recursive calls
        self.max_diameter = 0

        # STEP 2: Define helper to compute height + update diameter
        #   - Postorder: process children before current node
        def height(node: Optional[TreeNode]) -> int:
            if not node:
                return 0  # Base case: null node has height 0

            # Recursively get heights of left and right subtrees
            left_height = height(node.left)
            right_height = height(node.right)

            # STEP 3: Update max diameter using current node as "top"
            #   - Path through node = left_height + right_height (edges)
            current_diameter = left_height + right_height
            self.max_diameter = max(self.max_diameter, current_diameter)

            # Return height of current subtree (for parent's use)
            return 1 + max(left_height, right_height)
        # STEP 4: Trigger recursion and return result
        height(root)
        return self.max_diameter

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

    #   Test 1: Normal case - [1,2,3,4,5]
    #        1
    #       / \
    #      2   3
    #     / \
    #    4   5
    root1 = TreeNode(1)
    root1.left = TreeNode(2)
    root1.right = TreeNode(3)
    root1.left.left = TreeNode(4)
    root1.left.right = TreeNode(5)
    print(sol.diameterOfBinaryTree(root1))  # Expected: 3

    #   Test 2: Edge case - [1,2]
    root2 = TreeNode(1)
    root2.left = TreeNode(2)
    print(sol.diameterOfBinaryTree(root2))  # Expected: 1

    #   Test 3: Tricky/negative - empty tree
    print(sol.diameterOfBinaryTree(None))   # Expected: 0
```

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

Example Walkthrough

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

1. **Initialize**: `self.max_diameter = 0`.
2. Call `height(root=1)`:
    - `node=1` is not null → proceed.

- Call `height(node=2)` (left child).
    - `node=2` not null.
    - Call `height(node=4)`:
        - `node=4` has no children → `left_height = 0`, `right_height = 0`.
        - `current_diameter = 0 + 0 = 0` → `max_diameter` stays 0.
        - Return `1 + max(0,0) = 1`.
    - Call `height(node=5)`:
        - Same as above → returns 1.
    - Now at `node=2`: `left_height=1`, `right_height=1`.
    - `current_diameter = 1 + 1 = 2` → `max_diameter = max(0,2) = 2`.
    - Return `1 + max(1,1) = 2`.
- Call `height(node=3)` (right child):
    - No children → returns 1.
- Now at `node=1`: `left_height=2`, `right_height=1`.
- `current_diameter = 2 + 1 = 3` → `max_diameter = max(2,3) = 3`.
- Return `1 + max(2,1) = 3`.

3. Final return: `self.max_diameter = 3`.

Key Insight: The diameter isn't just "height of left + height of right at root" — it could be deeper inside a subtree (e.g., if root had only one child with a long internal path). That's why we **track max globally** during postorder traversal.

Complexity Analysis

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

    We visit every node exactly once during the DFS. Each node does O(1) work (comparisons, additions).

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

    Due to recursion stack depth, where `h` is the height of the tree. In worst case (skewed tree), `h = n`; in balanced tree, `h = log n`.

10. Binary Tree Maximum Path Sum

Pattern: Tree + Recursion + Postorder Traversal (with Global Tracking)

Problem Statement

A **path** in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence **at most once**. Note that the path does not need to pass through the root.

The **path sum** of a path is the sum of the node's values in the path.

Given the `root` of a binary tree, return *the maximum path sum of any non-empty path*.

Sample Input & Output

```
Input: root = [1,2,3]
Output: 6
Explanation: The optimal path is 2 → 1 → 3 with sum 6.
```

```
Input: root = [-10,9,20,null,null,15,7]
Output: 42
Explanation: The optimal path is 15 → 20 → 7 with sum 42.
```

```
Input: root = [-3]
Output: -3
Explanation: Only one node; path must be non-empty.
```

LeetCode Editorial Solution + Inline Tests

```
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
```

```python
        self.right = right

class Solution:
    def maxPathSum(self, root: Optional[TreeNode]) -> int:
        # STEP 1: Initialize global max with -inf
        #   - We track best path seen anywhere in tree
        self.max_sum = float('-inf')

        def dfs(node):
            # STEP 2: Base case - null node contributes 0
            if not node:
                return 0

            # STEP 3: Recurse on children
            #   - Get max path sum from left/right subtrees
            #   - Clamp negative contributions to 0 (ignore them)
            left_gain = max(dfs(node.left), 0)
            right_gain = max(dfs(node.right), 0)

            # STEP 4: Compute path sum through current node as root
            #   - This path cannot be extended upward
            current_path_sum = node.val + left_gain + right_gain

            # STEP 5: Update global max if this path is better
            self.max_sum = max(self.max_sum, current_path_sum)

            # STEP 6: Return max path sum that can be extended upward
            #   - Only one branch (left or right) can be chosen
            return node.val + max(left_gain, right_gain)

        dfs(root)
        return self.max_sum

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

    #  Test 1: Normal case - [1,2,3]
    root1 = TreeNode(1, TreeNode(2), TreeNode(3))
    print(sol.maxPathSum(root1))  # Expected: 6

    #  Test 2: Tricky case - [-10,9,20,null,null,15,7]
    root2 = TreeNode(-10)
    root2.left = TreeNode(9)
    root2.right = TreeNode(20)
    root2.right.left = TreeNode(15)
    root2.right.right = TreeNode(7)
    print(sol.maxPathSum(root2))  # Expected: 42

    #  Test 3: Edge case - single negative node
    root3 = TreeNode(-3)
    print(sol.maxPathSum(root3))  # Expected: -3
```

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

Example Walkthrough

We'll walk through **Test 2**: `[-10,9,20,null,null,15,7]`.

Tree structure:

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

Goal: Find max path sum (anywhere).

Step 1: Call `maxPathSum(root2)`
- `self.max_sum = -inf`
- Call `dfs(-10)`

Step 2: Inside `dfs(-10)`
- Node exists → proceed
- Call `dfs(9)` (left child)

Step 3: Inside `dfs(9)`
- Left = None → returns 0

- Right = None → returns 0
- `current_path_sum = 9 + 0 + 0 = 9`
- Update `self.max_sum = max(-inf, 9) = 9`
- Return `9 + max(0,0) = 9`

Step 4: Back in `dfs(-10)`, now call `dfs(20)` (right child)

Step 5: Inside `dfs(20)`
- Call `dfs(15)` → returns 15 (leaf)
- Call `dfs(7)` → returns 7 (leaf)
- `left_gain = max(15, 0) = 15`
- `right_gain = max(7, 0) = 7`
- `current_path_sum = 20 + 15 + 7 = 42`
- Update `self.max_sum = max(9, 42) = 42`
- Return `20 + max(15,7) = 35`

Step 6: Back in `dfs(-10)`
- `left_gain = max(9, 0) = 9`
- `right_gain = max(35, 0) = 35`
- `current_path_sum = -10 + 9 + 35 = 34`
- Compare: `self.max_sum = max(42, 34) = 42` (unchanged)
- Return `-10 + max(9,35) = 25` (but this return value isn't used for answer)

Step 7: Return `self.max_sum = 42`

Final output: **42**

Key Insight:
The best path (15→20→7) is **not extendable** to parent (-10), so we only consider it during the `current_path_sum` step. The return value (35) represents the best **extendable** path from 20 upward.

Complexity Analysis

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

  We visit each node exactly once in the DFS traversal. All operations per node are O(1).

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

  Due to recursion stack depth, where H is height of tree.
  Worst case: O(N) for skewed tree; O(log N) for balanced tree.
  No additional data structures scale with input size.

11. Construct Binary Tree from Preorder and Inorder Traversal

Pattern: Divide and Conquer + Hashing

Problem Statement

Given two integer arrays `preorder` and `inorder` where `preorder` is the preorder traversal of a binary tree and `inorder` is the inorder traversal of the same tree, construct and return the binary tree.

You may assume that duplicates do not exist in the tree.

Sample Input & Output

```
Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
Output: [3,9,20,null,null,15,7]
Explanation: Root is 3 (first in preorder).
In inorder, left of 3 is [9] → left subtree;
right is [15,20,7] → right subtree.
```

```
Input: preorder = [-1], inorder = [-1]
Output: [-1]
Explanation: Single-node tree.
```

```
Input: preorder = [1,2], inorder = [2,1]
Output: [1,2]
Explanation: 1 is root; 2 is left child
(since it appears before 1 in inorder).
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List, 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 buildTree(self, preorder: List[int], inorder: List[int]):
        # STEP 1: Initialize structures
        #   - Build a hash map for O(1) index lookup in inorder
        #   - Use a mutable index (list) to track current root in preorder
        inorder_index_map = {val: idx for idx, val in enumerate(inorder)}
        preorder_index = [0]  # mutable reference to track position

        # STEP 2: Main loop / recursion
        #   - Recursively build subtree given inorder bounds [left, right]
        #   - Invariant: preorder[preorder_index[0]] is current root
        def array_to_tree(left: int, right: int) -> Optional[TreeNode]:
            if left > right:
                return None

            # Current root value from preorder
            root_val = preorder[preorder_index[0]]
            root = TreeNode(root_val)
            preorder_index[0] += 1  # move to next root candidate

            # STEP 3: Update state / bookkeeping
            #   - Split inorder at root index → left/right subtrees
            #   - Why here? Because inorder tells us subtree boundaries
            root_idx = inorder_index_map[root_val]

            # Recurse on left then right (preorder: root → left → right)
            root.left = array_to_tree(left, root_idx - 1)
            root.right = array_to_tree(root_idx + 1, right)

            return root

        # STEP 4: Return result
        #   - Entire tree built from full inorder range
        return array_to_tree(0, len(inorder) - 1)

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

    # Helper to serialize tree to list (level-order)
    def serialize(root: Optional[TreeNode]) -> List:
        if not root:
            return []
        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)
        # Trim trailing Nones
        while result and result[-1] is None:
            result.pop()
        return result

    #   Test 1: Normal case
    t1 = sol.buildTree([3,9,20,15,7], [9,3,15,20,7])
    assert serialize(t1) == [3,9,20,None,None,15,7], "Test 1 failed"

    #   Test 2: Edge case (single node)
    t2 = sol.buildTree([-1], [-1])
    assert serialize(t2) == [-1], "Test 2 failed"

    #   Test 3: Tricky/negative (left-skewed)
    t3 = sol.buildTree([1,2], [2,1])
    assert serialize(t3) == [1,2], "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 `preorder = [3,9,20,15,7]`, `inorder = [9,3,15,20,7]`.

1. **Build `inorder_index_map`**:
   `{9:0, 3:1, 15:2, 20:3, 7:4}` → lets us find root position in O(1).
2. **Start recursion**: `array_to_tree(0, 4)` (entire inorder range).
   - `preorder_index = [0]` → root_val = `3`
   - Create `TreeNode(3)`
   - Increment `preorder_index` → now `[1]`
   - `root_idx = 1` (from map)
3. **Build left subtree**: `array_to_tree(0, 0)`
   - `preorder_index = [1]` → root_val = `9`
   - Create `TreeNode(9)`
   - Increment → `[2]`
   - `root_idx = 0`
   - Left: `array_to_tree(0, -1)` → returns `None`
   - Right: `array_to_tree(1, 0)` → returns `None`
   - Return node `9` → becomes left child of `3`
4. **Build right subtree**: `array_to_tree(2, 4)`
   - `preorder_index = [2]` → root_val = `20`
   - Create `TreeNode(20)`
   - Increment → `[3]`

- `root_idx = 3`
- Left: `array_to_tree(2, 2)` → uses `preorder[3]=15` → returns node `15`
- Right: `array_to_tree(4, 4)` → uses `preorder[4]=7` → returns node `7`
- Attach `15` and `7` as children of `20`

5. **Final tree**:

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

6. **Serialize**: Level-order → `[3,9,20,None,None,15,7]`
   (Trailing `None`s trimmed)

Complexity Analysis

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

  We visit each node exactly once. Hash map gives O(1) root index lookup. Total work is linear in number of nodes.

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

  Hash map stores `n` entries. Recursion depth is `O(h)`, where `h` is height. Worst case (skewed tree) → `O(n)`. So total space is `O(n)`.

12. Serialize and Deserialize Binary Tree

Pattern: Tree Traversal + DFS (Preorder) + String Encoding

Problem Statement

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

Clarification:
- You may serialize the `null` nodes as `"null"` (or any consistent sentinel).
- The serialization format must be self-delimiting (e.g., comma-separated).
- The tree may contain duplicate values.

Sample Input & Output

```
Input: root = [1,2,3,null,null,4,5]
Output: [1,2,3,null,null,4,5]
Explanation: The serialized string should allow perfect reconstruction.
```

```
Input: root = []
Output: []
Explanation: Empty tree serializes to empty string or "null".
```

```
Input: root = [1,null,2,null,3]
Output: [1,null,2,null,3]
Explanation: Right-skewed tree must preserve structure.
```

LeetCode Editorial Solution + Inline Tests

```
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 serialize(self, root: Optional[TreeNode]) -> str:
        # STEP 1: Initialize list to hold node values
        #   - Use list for efficient appending; join later
        vals = []

        # STEP 2: Preorder DFS traversal
        #   - Visit root, then left, then right
        #   - "null" marks missing children
        def dfs(node):
            if not node:
                vals.append("null")
                return
            vals.append(str(node.val))
            dfs(node.left)
            dfs(node.right)

        dfs(root)
        return ",".join(vals)

    def deserialize(self, data: str) -> Optional[TreeNode]:
        # STEP 1: Split data into tokens
        #   - Each token is either number or "null"
        tokens = data.split(",")
        self.index = 0  # Tracks current token

        # STEP 2: Reconstruct via preorder DFS
        #   - Same order as serialization
        def dfs():
            if tokens[self.index] == "null":
                self.index += 1
                return None
            node = TreeNode(int(tokens[self.index]))
            self.index += 1
            node.left = dfs()
            node.right = dfs()
            return node

        # STEP 3: Handle empty input
        if not data or data == "null":
            return None

        return dfs()

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

    #  Test 1: Normal case
    root1 = TreeNode(1)
    root1.left = TreeNode(2)
    root1.right = TreeNode(3)
    root1.right.left = TreeNode(4)
    root1.right.right = TreeNode(5)
    ser1 = sol.serialize(root1)
    deser1 = sol.deserialize(ser1)
    # Re-serialize to verify structure
    assert sol.serialize(deser1) == ser1
    print("  Test 1 passed")

    #  Test 2: Edge case - empty tree
    empty_root = None
    ser_empty = sol.serialize(empty_root)
    deser_empty = sol.deserialize(ser_empty)
    assert sol.serialize(deser_empty) == ser_empty
    print("  Test 2 passed")

    #  Test 3: Tricky case - right-skewed tree
    root3 = TreeNode(1)
    root3.right = TreeNode(2)
    root3.right.right = TreeNode(3)
    ser3 = sol.serialize(root3)
    deser3 = sol.deserialize(ser3)
    assert sol.serialize(deser3) == ser3
    print("  Test 3 passed")
```

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

Example Walkthrough

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

Step 1: Build the tree

- `root1 = TreeNode(1)`
- `root1.left = TreeNode(2)`
- `root1.right = TreeNode(3)`
- `root1.right.left = TreeNode(4)`
- `root1.right.right = TreeNode(5)`

Tree structure:

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

Step 2: Serialize (`serialize(root1)`)

- Call `dfs(root1)`
  - `node = 1` → append `"1"` → `vals = ["1"]`
  - Recurse left → `dfs(2)`
    * `node = 2` → append `"2"` → `vals = ["1","2"]`

        - Recurse left → `dfs(None)` → append `"null"` → `vals = [...,"null"]`
        - Recurse right → `dfs(None)` → append `"null"`
    - Back to root, recurse right → `dfs(3)`
        - Append `"3"`
        - `dfs(4)` → append `"4"`, then two `"null"`s
        - `dfs(5)` → append `"5"`, then two `"null"`s

Final `vals`:
`["1","2","null","null","3","4","null","null","5","null","null"]`

Return: `"1,2,null,null,3,4,null,null,5,null,null"`

Step 3: Deserialize that string

- `tokens = ["1","2","null","null","3","4","null","null","5","null","null"]`
- `index = 0`

Call `dfs()`: - `tokens[0] = "1"` → create `TreeNode(1)`, `index=1`
- Build left subtree: - `tokens[1] = "2"` → `TreeNode(2)`, `index=2`
- Left: `tokens[2]="null"` → return `None`, `index=3`
- Right: `tokens[3]="null"` → return `None`, `index=4`
- Back to root, build right: - `tokens[4]="3"` → `TreeNode(3)`, `index=5`
- Left: `tokens[5]="4"` → build node, then two nulls → `index=8`
- Right: `tokens[8]="5"` → build node, then two nulls → `index=11`

Reconstructed tree matches original.

Step 4: Re-serialize and compare

- `serialize(deser1)` produces same string → assertion passes.

All tests pass with correct structure preservation.

Complexity Analysis

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

  Each node is visited exactly once during serialization and once during deserialization. Splitting the string is also O(N).

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

  The `vals` list and `tokens` list each store N+1 entries (including nulls). The recursion stack depth is O(H), where H is tree height, but in worst case (skewed tree) H = N, so overall space is O(N).

13. Subtree of Another Tree

Pattern: Tree Traversal + Recursion (Tree Matching)

Problem Statement

Given the roots of two binary trees `root` and `subRoot`, return `true` if there is a subtree of `root` with the same structure and node values of `subRoot`. A subtree of a binary tree `tree` is a tree that consists of a node in `tree` and all of this node's descendants. The tree `tree` could also be considered as a subtree of itself.

Sample Input & Output

```
Input: root = [3,4,5,1,2], subRoot = [4,1,2]
Output: true
Explanation: The subtree rooted at node 4 in root matches subRoot exactly.
```

```
Input: root = [3,4,5,1,2,null,null,null,null,0], subRoot = [4,1,2]
Output: false
Explanation: The candidate subtree has an extra child (0),
so structures differ.
```

```
Input: root = [1], subRoot = [1]
Output: true
Explanation: Identical single-node trees - a tree is a subtree of itself.
```

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 isSubtree(
        self, root: Optional[TreeNode], subRoot: Optional[TreeNode]
    ) -> bool:
        # STEP 1: Base cases for outer recursion
        #   - If subRoot is empty, it's always a subtree (by definition)
        #   - If root is empty but subRoot isn't, impossible match
        if not subRoot:
            return True
        if not root:
            return False

        # STEP 2: Check if current trees match exactly
        #   - Use helper to compare structure + values
        if self._is_same_tree(root, subRoot):
            return True

        # STEP 3: Recurse on left and right subtrees
        #   - If either side contains subRoot, return True
        return (self.isSubtree(root.left, subRoot) or
                self.isSubtree(root.right, subRoot))

    def _is_same_tree(
        self, p: Optional[TreeNode], q: Optional[TreeNode]
    ) -> bool:
        # STEP 1: Both nodes null → identical
        if not p and not q:
            return True
        # STEP 2: One null, other not → 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: Recurse on children
        return (self._is_same_tree(p.left, q.left) and
                self._is_same_tree(p.right, q.right))

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

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

    sub1 = TreeNode(4)
    sub1.left = TreeNode(1)
    sub1.right = TreeNode(2)

    assert sol.isSubtree(root1, sub1) == True
    print(" Test 1 passed")

    #  Test 2: Edge case - extra node breaks match
    # root = [3,4,5,1,2,null,null,null,null,0], subRoot = [4,1,2]
    root2 = TreeNode(3)
    root2.left = TreeNode(4)
    root2.right = TreeNode(5)
    root2.left.left = TreeNode(1)
    root2.left.right = TreeNode(2)
    root2.left.right.left = TreeNode(0)  # extra node
```

```python
sub2 = TreeNode(4)
sub2.left = TreeNode(1)
sub2.right = TreeNode(2)

assert sol.isSubtree(root2, sub2) == False
print("  Test 2 passed")

#   Test 3: Tricky/negative - single node match
# root = [1], subRoot = [1]
root3 = TreeNode(1)
sub3 = TreeNode(1)

assert sol.isSubtree(root3, sub3) == True
print("  Test 3 passed")
```

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

Example Walkthrough

We'll trace **Test 1** step by step:

1. **Initial Call**: `isSubtree(root1, sub1)`
   - `root1.val = 3`, `sub1.val = 4` → not equal
   - So, skip `_is_same_tree` (returns `False`)
   - Recurse left: `isSubtree(root1.left, sub1)` → now `root = node(4)`
2. **Second Call**: `isSubtree(node(4), sub1)`
   - Now both roots have value 4 → call `_is_same_tree(node(4), sub1)`
3. **Inside `_is_same_tree`**:
   - Compare `4 == 4` → OK
   - Recurse left: `_is_same_tree(node(1), node(1))` → both exist, values equal, children null → returns `True`

- Recurse right: `_is_same_tree(node(2), node(2))` → same logic → `True`
- So `_is_same_tree` returns `True`

4. **Back to `isSubtree`**: returns `True` immediately → propagate up

Final Output: `True`
Match found at left child of root.

Complexity Analysis

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

  In worst case, we compare `subRoot` (size `n`) against every node in `root` (size `m`). Each comparison takes `O(n)`, leading to `O(m * n)`. Common when trees are skewed or many partial matches exist.

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

  Due to recursion stack depth. In worst case (skewed trees), depth is `O(m)` for outer recursion and `O(n)` for `_is_same_tree`, totaling `O(m + n)`.

14. Balanced Binary Tree

Pattern: Tree DFS (Postorder Traversal + Early Termination)

Problem Statement

Given a binary tree, determine if it is height-balanced.
A height-balanced binary tree is defined as a binary tree in which the left and right subtrees of every node differ in height by no more than one.

Sample Input & Output

```
Input: root = [3,9,20,null,null,15,7]
Output: true
Explanation: Heights of left (1) and right (2) subtrees of root differ by  1.
```

```
Input: root = [1,2,2,3,3,null,null,4,4]
Output: false
Explanation: Left subtree of root has height 3, right has height 1 → diff = 2.
```

```
Input: root = []
Output: true
Explanation: Empty tree is trivially balanced.
```

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 isBalanced(self, root: Optional[TreeNode]) -> bool:
        # STEP 1: Define helper that returns (is_balanced, height)
        #   - Returns (-1, _) if unbalanced to signal early exit
        def check_height(node):
            if not node:
                return 0  # Base case: height = 0

            # STEP 2: Recurse left and right (postorder)
            #   - Process children before current node
            left_height = check_height(node.left)
            if left_height == -1:
                return -1  # Propagate imbalance upward
            right_height = check_height(node.right)
            if right_height == -1:
                return -1  # Propagate imbalance upward

            # STEP 3: Check balance condition at current node
            #   - If diff > 1, mark as unbalanced (-1)
            if abs(left_height - right_height) > 1:
                return -1

            # STEP 4: Return actual height if balanced
            #   - Height = 1 + max(child heights)
            return 1 + max(left_height, right_height)

        # Final check: if helper returns -1 → unbalanced
        return check_height(root) != -1

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

    #   Test 1: Normal case - balanced
    #     3
    #    / \
    #   9  20
    #     /  \
    #    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.isBalanced(root1))  # Expected: True

    #   Test 2: Edge case - empty tree
    print(sol.isBalanced(None))  # Expected: True

    #   Test 3: Tricky/negative - unbalanced deep left
    #         1
    #        / \
    #       2   2
    #      / \
    #     3   3
    #    / \
    #   4   4
    root3 = TreeNode(1)
    root3.left = TreeNode(2)
    root3.right = TreeNode(2)
    root3.left.left = TreeNode(3)
    root3.left.right = TreeNode(3)
    root3.left.left.left = TreeNode(4)
    root3.left.left.right = TreeNode(4)
    print(sol.isBalanced(root3))  # Expected: False
```

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]`):

1. **Call `isBalanced(root1)`**
   → Enters helper `check_height(root1)` where `root1.val = 3`.
2. **Recurse left: `check_height(node=9)`**
   - Node 9 has no children → calls `check_height(None)` twice → both return 0.
   - `abs(0 - 0) = 0  1` → returns `1 + max(0,0) = 1`.
     → `left_height = 1`
3. **Recurse right: `check_height(node=20)`**
   - Node 20 has left=15, right=7.
     - `check_height(15)` → leaf → returns 1.
     - `check_height(7)` → leaf → returns 1.
   - `abs(1 - 1) = 0  1` → returns `1 + 1 = 2`.
     → `right_height = 2`
4. **At root (3):**

    - `abs(1 - 2) = 1   1` → balanced!
    - Returns `1 + max(1,2) = 3` (but we only care it's -1).
5. **Final return:** `3 != -1` → `True`.

State Summary:
- All subtrees checked bottom-up (postorder).
- No subtree violated balance → result is `True`.

Complexity Analysis

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

  Each node is visited exactly once in postorder traversal. Early termination avoids unnecessary work but worst-case still visits all nodes.

- **Space Complexity**: `O(h)` where `h = height of tree`

  Due to recursion stack depth. In worst case (skewed tree), `h = n`; in balanced tree, `h = log n`.

15. Lowest Common Ancestor of a Binary Tree

Pattern: Tree Traversal (Postorder DFS)

Problem Statement

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.
According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes `p` and `q` as the lowest node in `T` that has both `p` and `q` as descendants (where we allow a node to be a descendant of itself)."

Sample Input & Output

```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
Output: 3
Explanation: Nodes 5 and 1 are in left and right subtrees of 3 → LCA is 3.
```

```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
Output: 5
Explanation: Node 4 is in subtree of 5 → LCA is 5
(a node is ancestor of itself).
```

```
Input: root = [1,2], p = 1, q = 2
Output: 1
Explanation: Edge case - one node is root, other is its child.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import Optional

# 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'
    ) -> 'TreeNode':
        # STEP 1: Base case - if root is None or matches p/q,
        #         we've found one of the targets or hit leaf.
        if not root or root == p or root == q:
            return root
        # STEP 2: Recurse on left and right subtrees.
        #   - Postorder: explore children before deciding at root.
        left = self.lowestCommonAncestor(root.left, p, q)
        right = self.lowestCommonAncestor(root.right, p, q)

        # STEP 3: Decide based on what left/right returned.
        #   - If both non-null: current root is LCA.
        #   - If only one side non-null: that side has both nodes.
        if left and right:
            return root
        return left or right  # return whichever is not None

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

    # Build tree: [3,5,1,6,2,0,8,null,null,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

    #  Test 1: Normal case - LCA is root (3)
    result1 = sol.lowestCommonAncestor(root, p, q)
    assert result1.val == 3, f"Expected 3, got {result1.val}"

    #  Test 2: Tricky case - q is descendant of p (LCA = p = 5)
    q2 = root.left.right.right  # node 4
    result2 = sol.lowestCommonAncestor(root, p, q2)
    assert result2.val == 5, f"Expected 5, got {result2.val}"

    #  Test 3: Edge case - one node is root
    p3 = root          # node 3
    q3 = root.left     # node 5
    result3 = sol.lowestCommonAncestor(root, p3, q3)
    assert result3.val == 3, f"Expected 3, got {result3.val}"

    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 2**: `p = node 5`, `q = node 4` in the tree:

```
        3
       / \
      5   1
     / \
    6   2
       / \
      7   4   ← q
```

Goal: Find LCA of 5 and 4.

Step 1: Call `lowestCommonAncestor(root=3, p=5, q=4)`
- Root (3)   p and   q → recurse left and right.

Step 2: Left call → `lowestCommonAncestor(5, 5, 4)`
- Root (5) **equals p** → **return 5 immediately** (base case).

Step 3: Right call → `lowestCommonAncestor(1, 5, 4)`
- Root (1)   p/q → recurse its left (0) and right (8).
- Both return `None` (no p/q in subtree).
- So this call returns `None`.

Step 4: Back at root=3:
- `left = node 5`, `right = None`
- Since only one side non-null → return `left` → **node 5**

Final Output: `5` — correct! Because 4 is in subtree of 5, so 5 is LCA.

Key insight: The recursion **"bubbles up"** the first node that sees both targets in different subtrees — or the target itself if the other is below it.

Complexity Analysis

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

  In worst case, visit every node once (e.g., when p/q are leaves). DFS explores entire tree.

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

  Due to recursion stack depth, where `H` = height of tree.
  Worst case: `O(N)` for skewed tree; best case: `O(log N)` for balanced tree.

16. All Nodes Distance K in Binary Tree

Pattern: Tree Traversal + Graph Conversion (BFS/DFS)

Problem Statement

Given the `root` of a binary tree, a `target` node, and an integer `k`, return *an array of the values of all nodes that have a distance* `k` *from the* `target` *node.*

You can return the answer in **any order**.

The distance between two nodes is the number of edges on the path between them.

Sample Input & Output

```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, k = 2
Output: [7,4,1]
Explanation: Nodes at distance 2 from node 5 are 7, 4
(children of its child 2) and 1 (sibling via root).
```

```
Input: root = [1], target = 1, k = 3
Output: []
Explanation: Only one node exists; no node is 3 edges away.
```

```
Input: root = [0,1,null,3,2], target = 2, k = 1
Output: [1]
Explanation: Node 2's only neighbor at distance 1 is its parent, node 1.
```

LeetCode Editorial Solution + Inline Tests

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

class Solution:
    def distanceK(
        self, root: 'TreeNode', target: 'TreeNode', k: int
    ) -> List[int]:
        # STEP 1: Build adjacency list (graph) via DFS
        #   - Treat tree as undirected graph so we can move
        #     upward (to parent) and downward (to children).
        graph = defaultdict(list)

        def build_graph(node, parent):
            if not node:
                return
            if parent:
                graph[node.val].append(parent.val)
                graph[parent.val].append(node.val)
            build_graph(node.left, node)
            build_graph(node.right, node)

        build_graph(root, None)

        # STEP 2: BFS from target node up to distance k
        #   - Maintain queue of (node_value, distance)
        #   - Stop when distance exceeds k
        queue = deque([(target.val, 0)])
        visited = {target.val}
        result = []

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

            # STEP 3: Collect nodes exactly at distance k
            if dist == k:
                result.append(node_val)
                continue  # No need to explore further

            # Explore neighbors (parent + children)
            for neighbor in graph[node_val]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append((neighbor, dist + 1))

        # STEP 4: Return result (empty if k too large)
        return result

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    # Mock TreeNode for testing
    class TreeNode:
        def __init__(self, x, left=None, right=None):
            self.val = x
            self.left = left
            self.right = right

    sol = Solution()

    #  Test 1: Normal case
    # Tree: [3,5,1,6,2,0,8,null,null,7,4], target=5, k=2
    node4 = TreeNode(4)
    node7 = TreeNode(7)
    node2 = TreeNode(2, node7, node4)
    node6 = TreeNode(6)
    node5 = TreeNode(5, node6, node2)  # target
    node0 = TreeNode(0)
    node8 = TreeNode(8)
    node1 = TreeNode(1, node0, node8)
    root1 = TreeNode(3, node5, node1)
    assert set(sol.distanceK(root1, node5, 2)) == {7, 4, 1}

    #   Test 2: Edge case - k too large
    root2 = TreeNode(1)
    assert sol.distanceK(root2, root2, 3) == []

    #   Test 3: Tricky - target is leaf, k=1
    # Tree: [0,1,null,3,2], target=2, k=1
    node2 = TreeNode(2)  # target
    node3 = TreeNode(3)
    node1 = TreeNode(1, node3, node2)
    root3 = TreeNode(0, node1)
    assert sol.distanceK(root3, node2, 1) == [1]

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

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

Example Walkthrough

We'll walk through **Test 1** step by step:

Tree structure:

```
    3
   / \
  5   1
 / \ / \
6  2 0 8
  / \
 7   4
```

Target = node `5`, k = 2.

Step 1: Build graph via `build_graph` (DFS)
- Start at root (3), parent = None → no edge added yet.
- Recurse to 5 (parent=3): add edge 3 5.
- Recurse to 6 (parent=5): add 5 6.
- Recurse to 2 (parent=5): add 5 2.
- Recurse to 7 (parent=2): add 2 7.
- Recurse to 4 (parent=2): add 2 4.
- Back to root → recurse to 1 (parent=3): add 3 1.
- Then 0 and 8: add 1 0, 1 8.

Resulting graph (adjacency list):

```
3: [5, 1]
5: [3, 6, 2]
1: [3, 0, 8]
6: [5]
2: [5, 7, 4]
0: [1]
8: [1]
7: [2]
4: [2]
```

Step 2: BFS from target (5)
- Queue: `[(5, 0)]`, visited = `{5}`
- Pop (5,0): dist 2 → enqueue neighbors 3,6,2 → queue = `[(3,1),(6,1),(2,1)]`, visited = `{5,3,6,2}`

- Pop (3,1): dist 2 → neighbors: 5 (visited), 1 → enqueue (1,2)
- Pop (6,1): neighbors: 5 (visited) → nothing
- Pop (2,1): neighbors: 5 (visited), 7, 4 → enqueue (7,2), (4,2)

Now queue = `[(1,2), (7,2), (4,2)]`

- Pop (1,2): dist == 2 → add `1` to result
- Pop (7,2): dist == 2 → add `7`

- Pop (4,2): dist == 2 → add `4`

Result = `[1,7,4]` → order may vary, but set = `{1,7,4}`

Final output matches expected.

Complexity Analysis

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

  We visit each node **twice**: once during graph construction (DFS) and once during BFS. All operations per node are O(1). N = number of nodes.

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

  The adjacency list stores up to `2*(N-1)` edges (tree has N-1 edges, undirected → 2×). BFS queue and visited set also store up to O(N) entries.

17. Maximum Width of Binary Tree

Pattern: BFS (Level-Order Traversal) + Indexing Trick

Problem Statement

Given the `root` of a binary tree, return *the maximum width of the tree.*
The **width** of one level is defined as the length between the leftmost and rightmost non-null nodes in that level (including any null nodes in between).
The maximum width is the maximum width among all levels.
The answer will be in the range of a **32-bit signed integer**.

Clarification: We assign an index to each node as if the tree were a complete binary tree:
- Root has index `1`
- Left child of node with index `i` → `2 * i`
- Right child → `2 * i + 1`
Width of a level = `last_index - first_index + 1`

Sample Input & Output

```
Input: root = [1,3,2,5,3,null,9]
Output: 4
Explanation: Level 3 has nodes at indices 4,5,null,7 → width = 7 - 4 + 1 = 4
```

```
Input: root = [1,3,null,5,3]
Output: 2
Explanation: Level 2: [3, null] → indices 2 and (none for right),
but level 3: [5,3] at indices 4 and 5 → width = 5 - 4 + 1 = 2
```

```
Input: root = [1,3,2,5]
Output: 2
Explanation: Level 2: [3,2] → indices 2,3 → width = 2;
Level 3: [5] → only one node → width = 1 → max = 2
```

LeetCode Editorial Solution + Inline Tests

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

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

class Solution:
    def widthOfBinaryTree(self, root: Optional[TreeNode]) -> int:
        if not root:
            return 0

        # STEP 1: Initialize queue with (node, index)
        #   - Indexing mimics complete binary tree (root=1)
        queue = deque([(root, 1)])
        max_width = 0
        # STEP 2: BFS level-by-level
        #   - For each level, record first and last index
        #   - Width = last - first + 1
        while queue:
            level_size = len(queue)
            first_index = queue[0][1]
            last_index = first_index

            # Process all nodes at current level
            for _ in range(level_size):
                node, idx = queue.popleft()
                last_index = idx  # update to current (rightmost so far)

                # STEP 3: Enqueue children with correct indices
                # - Prevents overflow by using relative indexing
                # - But we keep absolute for clarity (Python handles big int)
                if node.left:
                    queue.append((node.left, 2 * idx))
                if node.right:
                    queue.append((node.right, 2 * idx + 1))

            # STEP 4: Update max_width after processing level
            current_width = last_index - first_index + 1
            max_width = max(max_width, current_width)

        return max_width

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

    #   Test 1: Normal case - [1,3,2,5,3,null,9]
    root1 = TreeNode(1)
    root1.left = TreeNode(3)
    root1.right = TreeNode(2)
    root1.left.left = TreeNode(5)
    root1.left.right = TreeNode(3)
    root1.right.right = TreeNode(9)
    assert sol.widthOfBinaryTree(root1) == 4, "Test 1 Failed"
    print("  Test 1 Passed")

    #   Test 2: Edge case - skewed left [1,3,null,5,3]
    root2 = TreeNode(1)
    root2.left = TreeNode(3)
    root2.left.left = TreeNode(5)
    root2.left.right = TreeNode(3)
    assert sol.widthOfBinaryTree(root2) == 2, "Test 2 Failed"
    print("  Test 2 Passed")

    #  Test 3: Tricky - only root and left child [1,3,2,5]
    root3 = TreeNode(1)
    root3.left = TreeNode(3)
    root3.right = TreeNode(2)
    root3.left.left = TreeNode(5)
    assert sol.widthOfBinaryTree(root3) == 2, "Test 3 Failed"
    print("  Test 3 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 = [1,3,2,5,3,null,9]`

Initial state:
- Queue = `[(node1, 1)]`
- `max_width = 0`

Level 0 (root):
- `level_size = 1`
- `first_index = 1`, `last_index = 1`
- Pop `(1,1)` → enqueue left `(3,2)`, right `(2,3)`
- Width = `1 - 1 + 1 = 1` → `max_width = 1`
- Queue now: `[(3,2), (2,3)]`

Level 1:
- `level_size = 2`
- `first_index = 2`
- Pop `(3,2)` → enqueue `(5,4)`, `(3,5)`
- Pop `(2,3)` → enqueue nothing for left, `(9,7)` for right
- `last_index = 7`
- Width = `7 - 2 + 1 = 6`? Wait—no!
  Mistake! Actually, we only process nodes **in this level**.

But note: the level has two nodes: indices 2 and 3 →
- After popping both: `last_index = 3`
- Width = `3 - 2 + 1 = 2` → `max_width = max(1,2) = 2`
- Queue now: `[(5,4), (3,5), (9,7)]`

Level 2:
- `level_size = 3`
- `first_index = 4`
- Pop `(5,4)` → no children
- Pop `(3,5)` → no children
- Pop `(9,7)` → no children
- `last_index = 7`
- Width = `7 - 4 + 1 = 4` → `max_width = max(2,4) = 4`

Final result: `4`

Key insight: We **only consider nodes actually present** in the level, but their **indices reflect their position in a complete tree**, so gaps (nulls) are implicitly counted via index difference.

Complexity Analysis

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

    We visit each node exactly once in BFS. Each enqueue/dequeue is O(1). Total nodes = N.

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

    Where W is the maximum width of the tree (i.e., max number of nodes in a level).
    In worst case (complete tree), W   N/2 → O(N).
    Queue stores at most one level's nodes.

18. Invert Binary Tree

Pattern: Tree Traversal (DFS / Recursion)

Problem Statement

Given the root of a binary tree, invert the tree, and return its root.
Inverting a binary tree means swapping the left and right children of all nodes.

Sample Input & Output

```
Input: root = [4,2,7,1,3,6,9]
Output: [4,7,2,9,6,3,1]
Explanation: Every node's left and right subtrees are swapped.
```

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

```
Input: root = [1]
Output: [1]
Explanation: Single node - no children to swap.
```

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 invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        # STEP 1: Base case - empty node
        #   - If node is None, nothing to invert; return None.
        if not root:
            return None

        # STEP 2: Swap left and right subtrees
        #   - This is the core inversion step.
        root.left, root.right = root.right, root.left

        # STEP 3: Recursively invert the new left and right subtrees
        #   - After swap, left is original right, and vice versa.
        #   - Recursion ensures all descendants are inverted.
        self.invertTree(root.left)
        self.invertTree(root.right)

        # STEP 4: Return the root (now inverted)
        #   - Root structure is modified in-place; return it.
        return root

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

    # Helper to build tree from list (level-order)
    def build_tree(vals):
        if not vals:
            return None
        nodes = [TreeNode(v) if v is not None else None for v in vals]
        kids = nodes[::-1]
        root = kids.pop()
        for node in nodes:
            if node:
                if kids:
                    node.left = kids.pop()
                if kids:
                    node.right = kids.pop()
        return root

    # Helper to serialize tree to list (level-order)
    from collections import deque
    def serialize(root):
        if not root:
            return []
```

```python
    result = []
    q = deque([root])
    while q:
        node = q.popleft()
        if node:
            result.append(node.val)
            q.append(node.left)
            q.append(node.right)
        else:
            result.append(None)
    # Trim trailing Nones
    while result and result[-1] is None:
        result.pop()
    return result

#   Test 1: Normal case
t1 = build_tree([4,2,7,1,3,6,9])
inverted = sol.invertTree(t1)
assert serialize(inverted) == [4,7,2,9,6,3,1]

#   Test 2: Edge case - empty tree
t2 = build_tree([])
inverted = sol.invertTree(t2)
assert serialize(inverted) == []

#   Test 3: Tricky/negative - single node
t3 = build_tree([1])
inverted = sol.invertTree(t3)
assert serialize(inverted) == [1]

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 `invertTree` on input `[4,2,7,1,3,6,9]`.

1. **Initial Call**: `invertTree(root=4)`

   - `root` exists → proceed.
   - Swap `left=2` and `right=7` → now `left=7`, `right=2`.
   - Recurse on new left (`7`) and new right (`2`).

2. **Recurse Left**: `invertTree(root=7)`
   - Swap `left=6` and `right=9` → now `left=9`, `right=6`.
   - Recurse on `9` and `6`.
     - Both are leaves → swap (no effect), then return.
3. **Recurse Right**: `invertTree(root=2)`
   - Swap `left=1` and `right=3` → now `left=3`, `right=1`.
   - Recurse on `3` and `1` → both leaves, return.
4. **Unwind**: All recursive calls return. Original root (`4`) is returned.

Final Tree Structure:

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

Serialized as `[4,7,2,9,6,3,1]`.

Complexity Analysis

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

  We visit every node exactly once. Each swap is O(1). Total = O(n).

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

  Recursion stack depth = height of tree (`h`).
  Worst case (skewed tree): `O(n)`.
  Best case (balanced): `O(log n)`.