Binary Tree

1. Maximum Depth of Binary Tree

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:

```
Input: root = [3,9,20,null,null,15,7]
```

Sample Output:

```
Output: 3
```

```python
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: TreeNode) -> int:
        # Base case: If the tree is empty, the depth is 0
        if not root:
            return 0

        # Recursively find the maximum depth of the left and right subtrees
        left_depth = self.maxDepth(root.left)
        right_depth = self.maxDepth(root.right)
```

```
        # Return the greater of the two depths +1 (for the current root node)
        return 1 + max(left_depth, right_depth)

# Sample Test Case
root = TreeNode(3)
root.left = TreeNode(9)
root.right = TreeNode(20, TreeNode(15), TreeNode(7))

solution = Solution()
print(solution.maxDepth(root))  # Expected Output: 3
```

3

2. Same Tree

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:

```
Input: p = [1,2,3], q = [1,2,3]
```

Sample Output:

```
Output: true
```

```
class Solution:
    def isSameTree(self, p: TreeNode, q: TreeNode) -> bool:
        """
        This function checks if two binary trees are structurally identical
        and have the same node values.
        """
        # Base case 1: If both trees are None, they are identical.
        if not p and not q:
            return True

        # Base case 2: If one tree is None and the other is not,
        # or if the values of the current nodes are different,
```

```
        # the trees are not the same.
        if not p or not q or p.val != q.val:
            return False

        # Recursive case:
        # Check if the left subtrees of both trees are the same.
        left_same = self.isSameTree(p.left, q.left)

        # Check if the right subtrees of both trees are the same.
        right_same = self.isSameTree(p.right, q.right)

        # Return True only if both left and right subtrees are the same.
        return left_same and right_same


# Sample Test Case
# Tree p:
#       1
#      / \
#     2   3
p = TreeNode(1, TreeNode(2), TreeNode(3))

# Tree q:
#       1
#      / \
#     2   3
q = TreeNode(1, TreeNode(2), TreeNode(3))

# Creating an instance of Solution and testing the method.
solution = Solution()

# The output will be True because both trees are structurally identical
# and have the same node values.
print(solution.isSameTree(p, q))  # Expected Output: True
```

```
True
```

3. Symmetric Tree

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:

```
Input: root = [1,2,2,3,4,4,3]
```

6 **Sample Output:**

```
Output: true
```

```python
class Solution:
    def isSymmetric(self, root: TreeNode) -> bool:
        # Helper function to check if two subtrees are mirror images
        def isMirror(t1: TreeNode, t2: TreeNode) -> bool:
            # Both are None: symmetric
            if not t1 and not t2:
                return True

            # Only one is None or values differ: not symmetric
            if not t1 or not t2 or t1.val != t2.val:
                return False

            # Check if the left subtree of t1 is a mirror of the right subtree of t2 and vice
            return isMirror(t1.left, t2.right) and isMirror(t1.right, t2.left)

        # The tree is symmetric if its left and right subtrees are mirrors
        return isMirror(root, root)

# Sample Test Case
root = TreeNode(1, TreeNode(2, TreeNode(3), TreeNode(4)), TreeNode(2, TreeNode(4), TreeNode(3

solution = Solution()
print(solution.isSymmetric(root))  # Expected Output: True
```

```
True
```

4. Invert Binary Tree

Problem Statement:
Given the `root` of a binary tree, invert the tree, and return its root. Inverting a tree means swapping the left and right child of every node in the tree.

Sample Input:

```
Input: root = [4,2,7,1,3,6,9]
```

Sample Output:

```
Output: [4,7,2,9,6,3,1]
```

```python
class Solution:
    def invertTree(self, root: TreeNode) -> TreeNode:
        # Base case: If the tree is empty, return None
        if not root:
            return None

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

        # Recursively invert the left and right subtrees
        self.invertTree(root.left)
        self.invertTree(root.right)

        return root

# Sample Test Case
root = TreeNode(4, TreeNode(2, TreeNode(1), TreeNode(3)), TreeNode(7, TreeNode(6), TreeNode(9

solution = Solution()
inverted_root = solution.invertTree(root)
# Expected Output: Tree with root -> 4, left -> 7, right -> 2
```

5. Balanced Binary Tree

Problem Statement:
Given a binary tree, determine if it is height-balanced. A binary tree is balanced if the left and right subtrees of every node differ in height by no more than 1.

Sample Input:

```
Input: root = [3,9,20,null,null,15,7]
```

Sample Output:

```
Output: true
```

```python
class Solution:
    def isBalanced(self, root: TreeNode) -> bool:
        # Helper function to calculate height and check balance
        def checkHeight(node: TreeNode) -> int:
            if not node:
                return 0  # Height of an empty tree is 0

            # Recursively calculate heights of left and right subtrees
            left_height = checkHeight(node.left)
            right_height = checkHeight(node.right)

            # If either subtree is unbalanced, propagate -1 upward
            if left_height == -1 or right_height == -1 or abs(left_height - right_height) > 1:
                return -1

            # Return the height of the current node
            return 1 + max(left_height, right_height)

        # The tree is balanced if checkHeight doesn't return -1
        return checkHeight(root) != -1

# Sample Test Case
root = TreeNode(3, TreeNode(9), TreeNode(20, TreeNode(15), TreeNode(7)))

solution = Solution()
print(solution.isBalanced(root))  # Expected Output: True
```

```
True
```

6. Binary Tree 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:

```
root = [3, 9, 20, null, null, 15, 7]
```

Sample Output:

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

```python
from collections import deque

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

def levelOrder(root):
    if not root:
        return []  # Return an empty list if the tree is empty

    result = []  # To store the final level order traversal
    queue = deque([root])  # Initialize a queue with the root node

    while queue:
        level = []  # List to store nodes at the current level
        for _ in range(len(queue)):
            node = queue.popleft()  # Remove the front node from the queue
            level.append(node.val)  # Add the node's value to the current level
            if node.left:
                queue.append(node.left)  # Add the left child to the queue
            if node.right:
                queue.append(node.right)  # Add the right child to the queue
        result.append(level)  # Add the current level to the result

    return result

# Test Case
# Constructing the tree:
#         3
#        / \
#       9   20
#           /  \
#          15   7
root = TreeNode(3)
root.left = TreeNode(9)
root.right = TreeNode(20, TreeNode(15), TreeNode(7))
print(levelOrder(root))  # Output: [[3], [9, 20], [15, 7]]
```

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

7. Binary Tree Zigzag Level Order Traversal

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:

```
root = [3, 9, 20, null, null, 15, 7]
```

Sample Output:

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

```python
from collections import deque

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

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

    while queue:
        level = deque()  # Use deque to support appending on either side
        for _ in range(len(queue)):
            node = queue.popleft()
            if left_to_right:
                level.append(node.val)  # Append to the right
            else:
                level.appendleft(node.val)  # Append to the left

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

        result.append(list(level))
```

```
        left_to_right = not left_to_right  # Toggle the direction

    return result

# Test Case
# Constructing the tree:
#         1
#        / \
#       2   3
#      / \   \
#     4   5   6
root = TreeNode(1)
root.left = TreeNode(2, TreeNode(4), TreeNode(5))
root.right = TreeNode(3, None, TreeNode(6))
print(zigzagLevelOrder(root))  # Output: [[1], [3, 2], [4, 5, 6]]
```

```
[[1], [3, 2], [4, 5, 6]]
```

8. Binary Tree Right Side View

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:

```
root = [1, 2, 3, null, 5, null, 4]
```

Sample Output:

```
[1, 3, 4]
```

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

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

    while queue:
```

```
        for i in range(len(queue)):
            node = queue.popleft()
            if i == 0:  # Capture the rightmost element of this level
                result.append(node.val)
            if node.right:
                queue.append(node.right)
            if node.left:
                queue.append(node.left)

    return result

# Test Case
# Constructing the tree:
#         1
#        / \
#       2   3
#        \   \
#         5   4
root = TreeNode(1)
root.left = TreeNode(2, None, TreeNode(5))
root.right = TreeNode(3, None, TreeNode(4))
print(rightSideView(root))  # Output: [1, 3, 4]
```

```
[1, 3, 4]
```

9. Path Sum II

Problem Statement:
Given the `root` of a binary tree and an integer `targetSum`, return all root-to-leaf paths where each path's sum equals `targetSum`.
A leaf is a node with no children.

Sample Input:

```
root = [5, 4, 8, 11, null, 13, 4, 7, 2, null, null, 5, 1]
targetSum = 22
```

Sample Output:

```
[[5, 4, 11, 2], [5, 8, 4, 5]]
```

```
def pathSum(root, targetSum):
    def dfs(node, target, path, result):
        if not node:
            return

        path.append(node.val)  # Add current node to the path
        target -= node.val  # Subtract current node's value from target

        if not node.left and not node.right and target == 0:
            result.append(list(path))  # Found a valid path

        dfs(node.left, target, path, result)
        dfs(node.right, target, path, result)
        path.pop()  # Backtrack

    result = []
    dfs(root, targetSum, [], result)
    return result

# Test Case
# Constructing the tree:
#         5
#        / \
#       4   8
#      /   / \
#     11  13  4
#    /  \    / \
#   7    2  5   1
root = TreeNode(5)
root.left = TreeNode(4, TreeNode(11, TreeNode(7), TreeNode(2)))
root.right = TreeNode(8, TreeNode(13), TreeNode(4, TreeNode(5), TreeNode(1)))
print(pathSum(root, 22))  # Output: [[5, 4, 11, 2], [5, 8, 4, 5]]
```

```
[[5, 4, 11, 2], [5, 8, 4, 5]]
```

10. Path Sum III

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 (i.e., traveling only from parent nodes to child nodes).

Sample Input:

```
root = [10, 5, -3, 3, 2, null, 11, 3, -2, null, 1]
targetSum = 8
```

Sample Output:

```
3
```

```python
def pathSumIII(root, targetSum):
    def dfs(node, curr_path):
        if not node:
            return 0

        # Add current node to the path
        curr_path.append(node.val)

        # Check for valid paths ending at the current node
        count = 0
        current_sum = 0
        for value in reversed(curr_path):
            current_sum += value
            if current_sum == targetSum:
                count += 1

        # Explore children
        count += dfs(node.left, curr_path)
        count += dfs(node.right, curr_path)

        # Backtrack
        curr_path.pop()
        return count

    return dfs(root, [])

# Test Case
# Constructing the tree:
#         10
#        /  \
```

```
#       5   -3
#      / \     \
#     3   2     11
#    / \   \
#   3  -2   1
root = TreeNode(10)
root.left = TreeNode(5, TreeNode(3, TreeNode(3), TreeNode(-2)), TreeNode(2, None, TreeNode(1)
root.right = TreeNode(-3, None, TreeNode(11))
print(pathSumIII(root, 8))  # Output: 3
```

```
3
```

11. Construct Binary Tree from Preorder and Inorder Traversal

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.

Sample Input:

```
preorder = [3, 9, 20, 15, 7]
inorder = [9, 3, 15, 20, 7]
```

Sample Output:
A binary tree with the structure:

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

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

```python
class Solution:
    def buildTree(self, preorder, inorder):
        if not preorder or not inorder:
            return None

        # The first element in the preorder list is the root.
        root_value = preorder[0]
        root = TreeNode(root_value)

        # Find the index of the root in inorder list.
        mid = inorder.index(root_value)

        # Recursively build the left and right subtrees.
        # For the left subtree, take the elements from the start of the preorder list
        # (after the root element) up to the length of the left subtree defined by inorder[:m
        # and use the elements in inorder[:mid].
        root.left = self.buildTree(preorder[1:mid+1], inorder[:mid])

        # For the right subtree, take the elements from post-left subtree in preorder
        # and use the elements in inorder[mid+1:]
        root.right = self.buildTree(preorder[mid+1:], inorder[mid+1:])

        return root

# Sample test case
preorder = [3,9,20,15,7]
inorder = [9,3,15,20,7]
solution = Solution()
root = solution.buildTree(preorder, inorder)
# The constructed binary tree should have the root with value 3
```

```python
root
```

```
<__main__.TreeNode at 0x25a451f3d90>
```

12. Subtree of Another Tree

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 as `subRoot`, and `false` otherwise.

Sample Input:

root = [3, 4, 5, 1, 2], subRoot = [4, 1, 2]

Sample Output:

true

```
class Solution:
    def isSubtree(self, s, t):
        if not s:
            return False
        if self.isSameTree(s, t):
            return True
        # Recursively check left and right subtree
        return self.isSubtree(s.left, t) or self.isSubtree(s.right, t)

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

# Sample test case
root_s = TreeNode(3)
root_s.left = TreeNode(4)
root_s.right = TreeNode(5)
root_s.left.left = TreeNode(1)
root_s.left.right = TreeNode(2)

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

solution = Solution()
result = solution.isSubtree(root_s, root_t)
# Result should be True as tree t is a subtree of s
```

```
result
```

True

13. Diameter of Binary Tree

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.

Sample Input:

```
root = [1, 2, 3, 4, 5]
```

Sample Output:

```
3
```

```python
class Solution:
    def diameterOfBinaryTree(self, root):
        self.diameter = 0

        def longest_path(node):
            if not node:
                return 0
            left_path = longest_path(node.left)
            right_path = longest_path(node.right)

            # Update the diameter if the path through root is larger
            self.diameter = max(self.diameter, left_path + right_path)

            # Return the longest path from current node
            return max(left_path, right_path) + 1

        longest_path(root)
        return self.diameter

# Sample test case
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

solution = Solution()
```

```
diameter = solution.diameterOfBinaryTree(root)
# The diameter should be 3 (path: 4 -> 2 -> 1 -> 3)
```

14. Maximum Width of Binary Tree

Problem Statement:
Given the root of a binary tree, return the maximum width of the binary tree. The maximum width of a tree is the maximum width among all levels. The width of one level is defined as the number of nodes between the leftmost and rightmost non-null nodes in the level.

Sample Input:

```
root = [1, 3, 2, 5, 3, null, 9]
```

Sample Output:

```
4
```

```
from collections import deque

class Solution:
    def widthOfBinaryTree(self, root):
        if not root:
            return 0

        max_width = 0
        queue = deque([(root, 0)])

        while queue:
            level_length = len(queue)
            _, first_index = queue[0]
            for _ in range(level_length):
                node, index = queue.popleft()
                if node.left:
                    queue.append((node.left, 2 * index))
                if node.right:
                    queue.append((node.right, 2 * index + 1))

            # Current level's width
            current_width = index - first_index + 1
```

```
            max_width = max(max_width, current_width)

        return max_width

# Sample test case
root = TreeNode(1)
root.left = TreeNode(3)
root.right = TreeNode(2)
root.left.left = TreeNode(5)
root.left.right = TreeNode(3)
root.right.right = TreeNode(9)

solution = Solution()
width = solution.widthOfBinaryTree(root)
# The maximum width should be 4 (level 3 containing nodes 5, 3, x, 9)
```

15. Lowest Common Ancestor of a Binary Tree

Problem Statement:
Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

Sample Input:

```
root = [3, 5, 1, 6, 2, 0, 8, null, null, 7, 4], p = 5, q = 1
```

Sample Output:

```
3
```

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

class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeN
        # Base Case: if root is None or root matches one of the nodes (p or q)
        if not root or root == p or root == q:
```

```
            return root

        # Recursively search in the left subtree
        left = self.lowestCommonAncestor(root.left, p, q)

        # Recursively search in the right subtree
        right = self.lowestCommonAncestor(root.right, p, q)

        # If left and right are both non-null, this means p and q are found in separate bran
        if left and right:
            return root  # this root is the LCA

        # Else return the non-null child
        return left if left else right

# Sample Test Case
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)

sol = Solution()
p = root.left  # Node with value 5
q = root.right  # Node with value 1
lca = sol.lowestCommonAncestor(root, p, q)
print(f"LCA of {p.val} and {q.val}: {lca.val}")
```

```
LCA of 5 and 1: 3
```

16. Serialize and Deserialize Binary Tree

Problem Statement:
Design an algorithm to serialize and deserialize a binary tree. Serialization is converting the tree to a string, and deserialization is converting the string back to the original tree structure.

Sample Input:

```
root = [1, 2, 3, null, null, 4, 5]
```

Sample Output:

```
Serialization: "1,2,3,null,null,4,5"
Deserialization: A binary tree with the structure:
    1
   / \
  2   3
     / \
    4   5
```

```python
class Codec:

    def serialize(self, root: 'TreeNode') -> str:
        """Encodes a tree to a single string."""
        def preorder(node):
            if node:
                vals.append(str(node.val))
                preorder(node.left)
                preorder(node.right)
            else:
                vals.append('#')

        vals = []
        preorder(root)
        return ' '.join(vals)

    def deserialize(self, data: str) -> 'TreeNode':
        """Decodes your encoded data to tree."""
        def doDeserialize():
            val = next(vals)
            if val == '#':
                return None
            node = TreeNode(int(val))
            node.left = doDeserialize()
            node.right = doDeserialize()
            return node

        vals = iter(data.split())
        return doDeserialize()
```

```
# Sample Test Case
# Using the same tree structure as before
codec = Codec()
serialized = codec.serialize(root)
print(f"Serialized: {serialized}")
deserialized = codec.deserialize(serialized)
# To verify the deserialization, let's serialize it again and check if it matches
print(f"Re-serialized: {codec.serialize(deserialized)}")
```

```
Serialized: 3 5 6 # # 2 7 # # 4 # # 1 0 # # 8 # #
Re-serialized: 3 5 6 # # 2 7 # # 4 # # 1 0 # # 8 # #
```

17. All Nodes Distance K in Binary Tree

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.

Sample Input:

```
root = [3, 5, 1, 6, 2, 0, 8, null, null, 7, 4], target = 5, k = 2
```

Sample Output:

```
[7, 4, 1]
```

```
class Solution:
    def distanceK(self, root: 'TreeNode', target: 'TreeNode', k: int):
        from collections import defaultdict, deque
        if not root:
            return []

        # Build the graph from the tree using adjacency list
        graph = defaultdict(list)

        def buildGraph(node, parent=None):
            if node:
                if parent:
                    graph[node.val].append(parent.val)
                    graph[parent.val].append(node.val)
```

```
                buildGraph(node.left, node)
                buildGraph(node.right, node)

        buildGraph(root)

        # BFS to find all nodes at distance k from the target
        queue = deque([(target.val, 0)])  # (current node value, current distance)
        visited = set()
        visited.add(target.val)

        res = []
        while queue:
            node, dist = queue.popleft()
            if dist == k:
                res.append(node)
            elif dist < k:
                for neighbor in graph[node]:
                    if neighbor not in visited:
                        visited.add(neighbor)
                        queue.append((neighbor, dist + 1))

        return res

# Sample Test Case
target = root.left  # Node with value 5
distance_k = 2
sol = Solution()
nodes_distance_k = sol.distanceK(root, target, distance_k)
print(f"Nodes distance {distance_k} from target {target.val}: {nodes_distance_k}")
```

```
Nodes distance 2 from target 5: [1, 7, 4]
```

18. Binary Tree Maximum Path Sum

Problem Statement:
Given a non-empty binary tree, return the maximum path sum. A path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The path must contain at least one node and does not need to go through the root.

Sample Input:

```
root = [-10, 9, 20, null, null, 15, 7]
```

Sample Output:

```
42
```

```python
class Solution:
    def maxPathSum(self, root: 'TreeNode') -> int:
        self.max_sum = float('-inf')

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

            # Recursively calculate the maximum path sum from the left subtree
            left_gain = max(max_gain(node.left), 0)

            # Recursively calculate the maximum path sum from the right subtree
            right_gain = max(max_gain(node.right), 0)

            # Calculate the price of the current path
            price_newpath = node.val + left_gain + right_gain

            # Update the global maximum sum
            self.max_sum = max(self.max_sum, price_newpath)

            # Returning the maximum gain if the current node's value is part of the path
            return node.val + max(left_gain, right_gain)

        max_gain(root)
        return self.max_sum

# Sample Test Case
sol = Solution()
max_path_sum = sol.maxPathSum(root)
print(f"Maximum Path Sum: {max_path_sum}")
```

```
Maximum Path Sum: 26
```