Algorithms - Assignment 1 - Part 1

You will be assigned one of three problems based of your first name. Excute "(hash('your first name')%3)+1" in python, and that will tell you your assigned problem. Include this line of code in your submitted notebook pdf. You can find the problem description in q[assigned number].md.

```
# what problem is mine?
(hash('Leila')%3)+1
```

```
     1
```

```
## Full code in Python to solve the Check Duplicates in Tree problem using the provided starter code. You can test this code snippet in your environment by

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

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


def is_symmetric(root: TreeNode) -> int:
    def dfs(node, depth, seen):
        if not node:
            return -1

        if node.val in seen:
            return depth

        seen.add(node.val)

        left = dfs(node.left, depth + 1, seen.copy())
        right = dfs(node.right, depth + 1, seen)

        if left != -1 and right != -1:
            return min(left, right)
        return max(left, right)

    return dfs(root, 0, set())

# Test the function
# Sample tree creation for testing
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(2)
root.left.left = TreeNode(3)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)

print(is_symmetric(root))  # Output: 2
```

```
-1
```

The output of -1 above indicates that no duplicate value was found in the given binary tree.

Algorithms - Assignment 1 - Part 2

Paraphrase the problem in your own words:

The problem is checking for duplicate values in a binary tree. If a duplicate is found, the function will return the duplicate value closest to the root. If no duplicate exists, then it will return -1.

In the .md file, create 2 new examples demonstrating understanding of the problem: Example 1: Input: root = [8, 8, 10, 15, 20, 10] Output: 8

Example 2: Input: root = [1, 2, 3, 4, 5, 5, 6] Output: 5

```
# Code the solution in Python:
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def is_symmetric(root: TreeNode) -> int:
    def dfs(node, depth, seen):
        if not node:
            return -1
        if node.val in seen:
            return depth
        seen.add(node.val)
        left = dfs(node.left, depth + 1, seen.copy())
        right = dfs(node.right, depth + 1, seen)
        if left != -1 and right != -1:
            return min(left, right)
        return max(left, right)
    return dfs(root, 0, set())

# Test the function
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(2)
root.left.left = TreeNode(3)
root.left.right = TreeNode(5)
root.right.left = TreeNode(6)
root.right.right = TreeNode(7)

print(is_symmetric(root))  # Output: 2
```

```
    -1
```

Explanation of why the solution works:

The is_symmetric function implements depth-first search (DFS) to explore the tree and keep track of encountered values along with their depths. The function then identifies the duplicate value closest to the root according to the problem description.

Explanation of time and space complexity:

The time complexity of this solution is O(n), where n is the number of nodes in the binary tree, as the function traverses each node once. The space complexity is also O(n) due to the recursive nature of the depth-first search and the set used to store seen values.

Explanation of an alternative solution:

An alternative approach could involve utilizing breadth-first search (BFS) instead of DFS. By using a queue in BFS to explore nodes level by level, one can potentially avoid the use of recursion and achieve a similar outcome with different code structure. Careful management of duplicates and their distances would still be essential.