In [ ]:

```
# Reviewing my partner assignment 1.

# Paraphrase the problem in your own words.
# Search a list of integers and return the first integer value that is reape
# If there are multiple duplicates, the first duploicated value that is reap
# If there is no duplicated value in the list, code should return -1
```

In [ ]:

```
# Here ia an example : We have a list of [1, 2, 2, 3, 5, 6, 7]
# There is only one value that's repeted more than once and that's 2.
# Based on the defined problem, code should return the duplicated value of 2
```

In [ ]:

```
# Below is my partner code

# 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

# Building the tree(DFS_pre-order)
def build_tree(roots):
    if not roots:
        return None

    root = TreeNode(roots[0])
    built_tree = [root]
    index = 1

    while built_tree and index < len(roots):
        node = built_tree.pop(0)

        if index < len(roots) and roots[index] is not None:
            node.left = TreeNode(roots[index])
            built_tree.append(node.left)
        index += 1

        if index < len(roots) and roots[index] is not None:
            node.right = TreeNode(roots[index])
            built_tree.append(node.right)
        index += 1

    return root

# DFS function to find duplicates and their depths
def dfs(node, depth, visited, closest_duplicate, min_depth):
    if node is None:
        return closest_duplicate, min_depth  # No duplicates found in this p

    if node.val in visited:
        if depth < min_depth:
            closest_duplicate = node.val
            min_depth = depth
    visited.add(node.val)

    closest_duplicate, min_depth = dfs(node.left, depth + 1, visited, closes
    closest_duplicate, min_depth = dfs(node.right, depth + 1, visited, close

    return closest_duplicate, min_depth


# Function to find the duplicate with the smallest depth (closest to the roo
def find_duplicate(root):
    visited = set()
    closest_duplicate, min_depth = dfs(root, 0, visited, None, float('inf'))

    return closest_duplicate if closest_duplicate is not None else -1
```

In [ ]:

```
# This code builds a binary tree from a level-order list (via build_tree).
#   Uses a queue-based level-order construction.
#   Handles None values for missing nodes.

# Then performs a depth-first search (dfs) on the binary tree.
#   Tracks visited node values in a set.
#   Finds the first value that repeats as the tree is traversed depth-first,

# At last it find_duplicate() wraps everything:
    # It returns the first duplicate value it finds closest to the root, not

# But I don't think this code actually does what is asked from it to do, whi
# Given a list of integers, return the **first value that appears more than
# If there are multiple duplicates, return the one that appears **first** in
# If no duplicate exists, return `-1`.
```

In [ ]:

```
# My partner code, build_tree(roots) uses a queue to build the tree in level
# Every element in roots is processed once.
# Therefore Time: O(n)

# dfs(...)
# Visits each node once.
# Set operations (in, add) are average-case O(1).
# Therefore Time: O(n)
```

In [ ]:

```
# Here is my critique of my partner's code

# My partner code takes a list of integers, but instead of checking the list
# builds a binary tree from that list (as if it's a level-order traversal).
# Then performs DFS on the tree, tracking the first duplicate based on depth
# This adds structural assumptions (tree depth, node position) that are irre

# I think what's needed is actually simplere. Something like:
# def first_duplicate(nums):
#     seen = set()
#     for num in nums:
#         if num in seen:
#             return num
#         seen.add(num)
#     return -1
```

In [ ]:

```
# Reflection

# For assignment 1 I answered Question Two and problem was strait forward.
# For reviewing my partner code, I am not sure if I got the problem statemen
# I am not sure why she had treated a simple list as a binary tree.
```