Part 1: Q1
https://github.com/KadiaAshish/DSI_Algorithms_Assignment_1/blob/main/Algorithm_Assignn

Part 2 Partner: Ashish Kadia

Paraphrase the problem in your own words.

In [ ]:
```
#Given tree of intergers, find nodes that are duplicate. If there are no dup
```

Create 1 new example that demonstrates you understand the problem.
Trace/walkthrough 1 example that your partner made and explain it.

In [ ]:
```
[10, 2, 4, 3, 3, 5, 7]
 (10)
(2)(4)
(3) (3) (5) (7)
#returns 3
```

```
  Cell In[3], line 2
    (10)
    ^
IndentationError: unexpected indent
```

In [ ]:
```
[10, 4, 4, 3, 3, 5, 7]
 (10)
(4)(4)
(3) (3) (5) (7)
#returns 4
```

In [ ]:
```
#imbalanced tree
 [10, 4, 4, 3, 3]
 (10)
(4)(4)
(3) (3)
#returns 4
```

Copy the solution your partner wrote.

In [ ]:
```
# importing necessary library
from collections import deque

# creating a template for each node which will have two child (left and righ
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

# defining a new function which can convert a list of input to a binary tree
def list_to_tree(nodes):
```

```
    # creating and storing first node as root
    root = TreeNode(nodes[0])
    # initiates level order traversal
    queue = deque([root])
    i = 1

    # creating a while loop which will keep stroing left and right child
    while queue and i < len(nodes):
        current = queue.popleft()  # Pop the leftmost node from the queue

        # creating left child if the value is not None
        if nodes[i] is not None:
            current.left = TreeNode(nodes[i])
            queue.append(current.left)

        i += 1

        # Creating right child if the value is not None
        if i < len(nodes) and nodes[i] is not None:
            current.right = TreeNode(nodes[i])
            queue.append(current.right)

        i += 1

    return root

# finding duplicates and storing them in dictionary
def find_closest_duplicate(root):

    queue = deque([(root, 0)])
    duplicates = {}

    while queue:
        # Pop the leftmost node from the queue
        node, distance = queue.popleft()
        # check for duplicate
        if node.val in duplicates:
            return node.val

        # update dictionary with current node's value and distance
        duplicates[node.val] = distance

        # add left child to the queue with an increased distance
        if node.left:
            queue.append((node.left, distance + 1))
        # add right child to the queue with an increased distance
        if node.right:
            queue.append((node.right, distance + 1))

    # returning -1 if no duplicates found
    return -1

def is_symmetric(root: TreeNode) -> int:
    return find_closest_duplicate(root)
```

In [ ]:
```
root = [10, 2, 4, 3, 3, 5, 7]
tmp=list_to_tree(root)
print(is_symmetric(tmp))
```

3

In [ ]:
```
root = [10, 4, 4, 3, 3, 5, 7]
tmp=list_to_tree(root)
print(is_symmetric(tmp))
```

4

In [ ]:
```
root =  [10, 4, 4, 3, 3]
tmp=list_to_tree(root)
print(is_symmetric(tmp))
```

4

Explain why their solution works in your own words.

This question is a BFS (breadth-first search) algorithm problem. The difficulty of this is in going from a list to a tree and keeping track of the values. Ashish solved this by first iterating the list one value at a time and contstructing the tree along the way (root -> left -> right) in the list_to_tree() function. Then the second fuction find_closest_duplicate() takes the tree and pop the leftmost element from the deque and then add the neighbors of the popped element to the right end of the deque, if it is not a duplicate value. Duplicate values are cleverly handled by the dictionary duplicates{}. As soon as a value from node (node.val) matches with duplicates{} it returns the value. Since this function traverse from the root, this avoids keeping track of the height of the tree as the first duplicate must be closest to the root.

Explain the problem's time and space complexity in your own words.

Since all nodes need to be traversed in the worst case, the theoretical time complexity is O(n). The space complexity is depends on the height of the tree and therefore the worst case would be the maximum depth of the tree. There are some some steps that were taken to improve performance, such as by using linked list 'deque' as a queue (as opposed to using the inefficient 'list') which has a time complexity of O(1) for append and pop operations.

Critique your partner's solution, including explanation, if there is anything should be adjusted.

Ashish has a good idea of the problem and has solved the problem in a quite optimized way. The code is properly commented as well. From my perspective, it seems difficult to make dramatic improvement to the solution. In terms of area of improvement, I think it is not necessary to keep track of 'distance' in find_closest_duplicate() as distance was not necessary when iterating from root node. Also, I am uncertain why a fourth function

is_symmetric() is necssary and can perhaps be be integrated with find_closest_duplicate() for simplicity and cleaner code.

Part 3: Please write a 200 word reflection documenting your studying process from assignment 1, and your presentation and reviewing experience with your partner at the bottom of the Juypter Notebook under a new heading "Reflection." Again, export this Notebook as pdf.

Reflection

For Assignment 1, it was tricky to find a more optimal solution in my own submission to find missing intergers from [0,n] given an integer list. It seems that it could be possible to implement a recurrsive approach, but given the limited time, I used a more naive approach by first finding min max value from the given range and traverse through the numbers. The biggest time compexity contributor is the sorting step. Since the numbers must be in order, it seems impossible to avoid this step. From my research sorting problem is quite inefficient so I would love to learn if there is a better way to handle it. I have missed a requirement which is the intergers can range from 0 rather than 1. That was a mistake I wish I had avoided.

As for the reveiwing experieince, I find the tree problems quite challenging and understanding the mindset of another person requires some time. I had to repeat working on the problem to fully understands the difficulties and limitations. Thanks to going through this question, I had a better grasp of handling tree questions and learned about using deque.

In [ ]: