ASSIGNMENT 2

1. Paraphrase the problem in your own words. Example 3 Input: lst = [6, 8, 2, 3, 5, 7, 0, 1, 10] Output: [4, 9] Starter Code def missing_num(nums: List) -> int: # TODO

Given a list of integers, the task is to implement a function called missing_num that finds and returns a list containing two missing numbers. The function needs to take a list of integers as input and return a list containing two integers that are missing from the original list. The starter code provides a function signature with a placeholder comment to indicate where the implementation needs to be added.

1. Create 1 new example that demonstrates you understand the problem. Trace/walkthrough 1 example that your partner made and explain it. New Example: Input: lst = [2, 5, 8, 1, 4, 9, 0, 3] Output: [6, 7] In this example, the input list 'lst' is missing the integers 6 and 7. The goal is to create a function, 'missing_num' that takes this list as input and returns a list containing the missing numbers, which, in this case, are 6 and 7.

Trace/Walkthrough of the Partner Example: Input: 'lst = [6, 8, 2, 3, 5, 7, 0, 1, 10] Expected Output: '[4, 9]' The function 'missing_num' should identify and return the two missing numbers from the input list. The missing numbers in this example are 4 and 9. The function should return the list '[4, 9]' as the output. Explanation: • The input list has integers from 0 to 10 with two missing numbers, 4 and 9. • The function needs to identify and return these missing numbers in a list. • The provided starter code sets up the function with the name 'missing_num' and expects a list of integers as input. • The task is to complete the function to find and return the missing numbers. • The output [4, 9] indicates that 4 and 9 are the missing numbers in the input list. This walkthrough demonstrates how the 'missing_num' function is expected to work, identifying and returning the missing numbers in a given list, as provided by your partner.

```
•    Copy the solution your partner wrote.
def find_missing_numbers_corrected(lst):
# Initialize a set from the list to remove duplicates
seen = set(lst)
# Calculate the expected range based on the list's length
n = len(lst)
# Find and return the missing numbers by checking each number in
the expected rang
missing_numbers = [num for num in range(n + 1) if num not in seen]
return missing_numbers if missing_numbers else -1
# Re-test with corrected logic for Custom Examples 4 and 5
# Correcting the logic based on the understanding of the problem
requirements
example4_corrected = [3, 3, 0, 1]
output4_corrected = find_missing_numbers_corrected(example4_corrected)
example5_corrected = [1, 2, 3, 4, 4]
output5_corrected = find_missing_numbers_corrected(example5_corrected)
output4_corrected, output5_corrected
([2, 4], [0, 5])
def find_missing_numbers_refined(lst):
# Calculate the maximum possible number in the list, which is
```

```
len(lst) to account for 0-based indexing
n = len(lst)
# Create a set from the list for O(1) lookups
seen = set(lst)
# Find missing numbers by checking each number in the expected
range against the set
missing_numbers = [num for num in range(n) if num not in seen]
# Return missing numbers if any, otherwise return -1
return missing_numbers if missing_numbers else -1
# Correct the examples to accurately reflect the intended logic
# Example 4 should correctly only identify numbers missing within the
list's actual content range
output4_refined = find_missing_numbers_refined(example4_corrected)
# Example 5 should do the same, accurately identifying only the
missing numbers within the intended range
output5_refined = find_missing_numbers_refined(example5_corrected)
output4_refined, output5_refined
([2], [0])
```

• Explain why their solution works in your own words. The provided code consists of two functions, 'find_missing_numbers_corrected' and 'find_missing_numbers_refined', designed to identify missing numbers in a given list. The corrected version ensures that the output only includes numbers missing within the actual content range of the list, as demonstrated through examples 4 and 5. The refined version further optimizes the approach by eliminating the need for an additional variable in the calculation of the set. These functions exhibit a time complexity of O(n) due to linear iterations over the list, where n is the length of the input list.

• Explain the problem's time and space complexity in your own words. In terms of space complexity, the use of sets contributes O(m) space, where m is the number of unique elements in the list, with a maximum of O(n) The space for the output list is O(n) in the worst case. Overall, the space complexity is O(n), primarily determined by the space needed for the set and the output list. The time complexity of the provided solution is O(n), where n represents the length of the input list. This linear time complexity is primarily influenced by three main operations. Firstly, converting the list to a set involves iterating over each element in the list, contributing to a time complexity of O(n). Secondly, the for-loop that iterates over the range [0, n] checks for each number's presence in the set, and this operation is O(n) as well.

• Critique your partner's solution, including explanation, if there is anything should be adjusted. The provided solution employs efficient techniques such as set operations to achieve the task at hand. However, there's room for minor improvements, such as incorporating the corrected logic into the 'find_missing_numbers_corrected' function for consistency. Additionally, the refined version could be further optimized by directly using the 'set()' constructor with the list. An alternative solution could involve sorting the list and then identifying missing numbers by comparing consecutive elements. While potentially less efficient than using sets, this approach avoids the need for additional data structures and still achieves a linear time complexity, O(n log n), due to the sorting operation. This alternative method involves a systematic traversal of the sorted list to identify missing numbers, providing a different perspective on solving the problem.

• 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.

In completing Assignment 1, I initially focused on understanding the problem statement thoroughly. Breaking down the problem into smaller components allowed me to devise a step-by-step plan for the solution. Utilizing Python's set data structure, I successfully implemented a function to find missing numbers in a given list, considering various edge cases. The process involved debugging, testing with custom examples, and refining the code for better efficiency.

During the presentation and review of my partner’s assignment, I had the opportunity to gain insights into his problem-solving approach. Reviewing his provided solution, I appreciated the clarity in his code comments and the use of sets to efficiently identify missing numbers. The partner's correction and refinement of the logic demonstrated a thoughtful consideration of the problem's requirements.

For Assignment 2, the tree traversal problem was effectively addressed using Depth-First Search (DFS). The recursive approach efficiently captured all paths from the root to the leaf nodes, incorporating leaf node detection and backtracking. The provided alternative solution, considering Breadth-First Search (BFS), showcased a different strategy for traversing the tree level by level using a queue.