AIAC-12.4

Name: Siri Vempati

Ht.no: 2303A51309

Batch.no:05

Task 1: Bubble Sort for Ranking Exam Scores

Scenario

You are working on a college result processing system where a small list of student scores needs to be sorted after every internal assessment.

Task Description

• Implement Bubble Sort in Python to sort a list of student scores.

• Use an AI tool to:

o Insert inline comments explaining key operations such as comparisons, swaps, and iteration passes

o Identify early-termination conditions when the list becomes sorted

o Provide a brief time complexity analysis

Expected Outcome

• A Bubble Sort implementation with:

o AI-generated comments explaining the logic

o Clear explanation of best, average, and worst-case complexity

o Sample input/output showing sorted scores

Code:

```
def bubble_sort(scores):
    n = len(scores)
    # Traverse through all elements in the list
    for i in range(n):
        # Initialize a flag to check if any swapping occurs
        swapped = False
        # Last i elements are already in place, no need to check them
        for j in range(0, n-i-1):
            # Compare adjacent elements
            if scores[j] > scores[j+1]:
                # Swap if the element found is greater than the next element
                scores[j], scores[j+1] = scores[j+1], scores[j]
                swapped = True  # Set the flag to True if a swap occurred
        # If no swapping occurred, the list is already sorted
        if not swapped:
            break  # Early termination if the list is sorted
    return scores
# Time Complexity Analysis:
# Best Case: O(n) - when the list is already sorted (only one pass needed
# Worst Case: O(n^2) - when the list is sorted in reverse order
# Average Case: O(n^2) - when the list is in random order
# Sample Input/Output
scores = [88, 75, 92, 85, 69]
sorted_scores = bubble_sort(scores)
print("Sorted Scores:", sorted_scores)
```

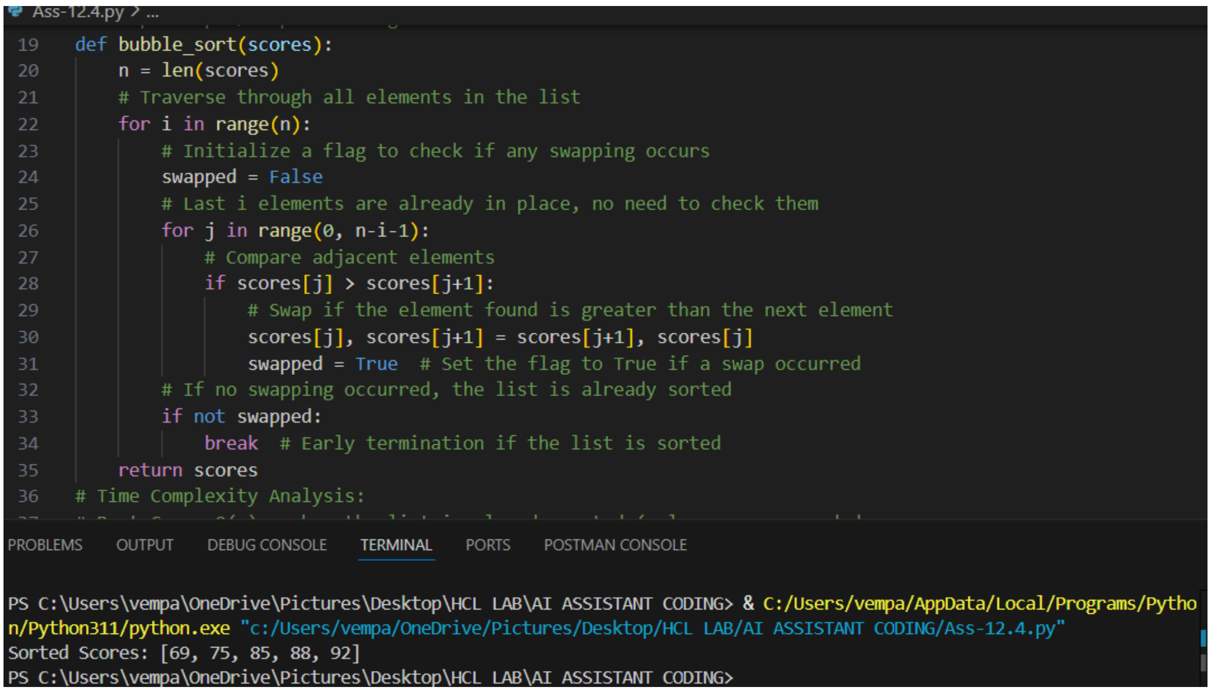
Clear explanation of best, average, and worst-case complexity

# Best Case: O(n) - This occurs when the input list is already sorted. The algorithm makes one pass through the list to check if it is sorted, and since no swaps are needed, it terminates early.

# Worst Case: O(n^2) - This occurs when the input list is sorted in reverse order. The algorithm needs to compare each element with every other element,resulting in n*(n-1)/2 comparisons and swaps.

# Average Case: O(n^2) - This occurs when the input list is in random order. On average, the algorithm will require n/2 comparisons and swaps for each of the n elements, leading to O(n^2) complexity.

Output:



```python
def bubble_sort(scores):
    n = len(scores)
    # Traverse through all elements in the list
    for i in range(n):
        # Initialize a flag to check if any swapping occurs
        swapped = False
        # Last i elements are already in place, no need to check them
        for j in range(0, n-i-1):
            # Compare adjacent elements
            if scores[j] > scores[j+1]:
                # Swap if the element found is greater than the next element
                scores[j], scores[j+1] = scores[j+1], scores[j]
                swapped = True  # Set the flag to True if a swap occurred
        # If no swapping occurred, the list is already sorted
        if not swapped:
            break  # Early termination if the list is sorted
    return scores
# Time Complexity Analysis:
```

```
PS C:\Users\vempa\OneDrive\Pictures\Desktop\HCL LAB\AI ASSISTANT CODING> & C:/Users/vempa/AppData/Local/Programs/Pytho
n/Python311/python.exe "c:/Users/vempa/OneDrive/Pictures/Desktop/HCL LAB/AI ASSISTANT CODING/Ass-12.4.py"
Sorted Scores: [69, 75, 85, 88, 92]
PS C:\Users\vempa\OneDrive\Pictures\Desktop\HCL LAB\AI ASSISTANT CODING>
```

Explanation:

The Bubble Sort algorithm works by repeatedly stepping through the list, comparing adjacent elements and swapping them if they are in the wrong order. This process is repeated until the list is sorted. The algorithm includes an optimization to terminate early if no swaps were made during a pass, indicating that the list is already sorted. The time complexity of Bubble Sort is O(n) in the best case (when the list is already sorted) and O(n^2) in both average and worst cases (when the list is in random order or reverse order).

Task 2: Improving Sorting for Nearly Sorted

Attendance Records

Scenario

You are maintaining an attendance system where student roll numbers are already almost sorted, with only a few late updates.

Task Description

• Start with a Bubble Sort implementation.

• Ask AI to:

o Review the problem and suggest a more suitable sorting algorithm

o Generate an Insertion Sort implementation

o Explain why Insertion Sort performs better on nearly sorted data

• Compare execution behavior on nearly sorted input

Expected Outcome

• Two sorting implementations:

o Bubble Sort

o Insertion Sort

• AI-assisted explanation highlighting efficiency differences for partially sorted datasets

Code:

```
attendance_records = [101, 102, 103, 105, 104, 106]

print("Original attendance records:", attendance_records)

def bubble_sort(arr):

    n = len(arr)

    for i in range(n):

        swapped = False

        for j in range(0, n - i - 1):

            if arr[j] > arr[j + 1]:

                arr[j], arr[j + 1] = arr[j + 1], arr[j]

                swapped = True

        if not swapped:

            break

def insertion_sort(arr):

    # Traverse through 1 to len(arr)

    for i in range(1, len(arr)):

        key = arr[i]

        j = i - 1

        # Move elements of arr[0..i-1], that are greater than key, to one position ahead of their
current position

        while j >= 0 and key < arr[j]:

            arr[j + 1] = arr[j]

            j -= 1

        arr[j + 1] = key

# Sort using Bubble Sort

bubble_sort(attendance_records)
```

print("Sorted attendance records (Bubble Sort):", attendance_records)

# Resetting the list for Insertion Sort

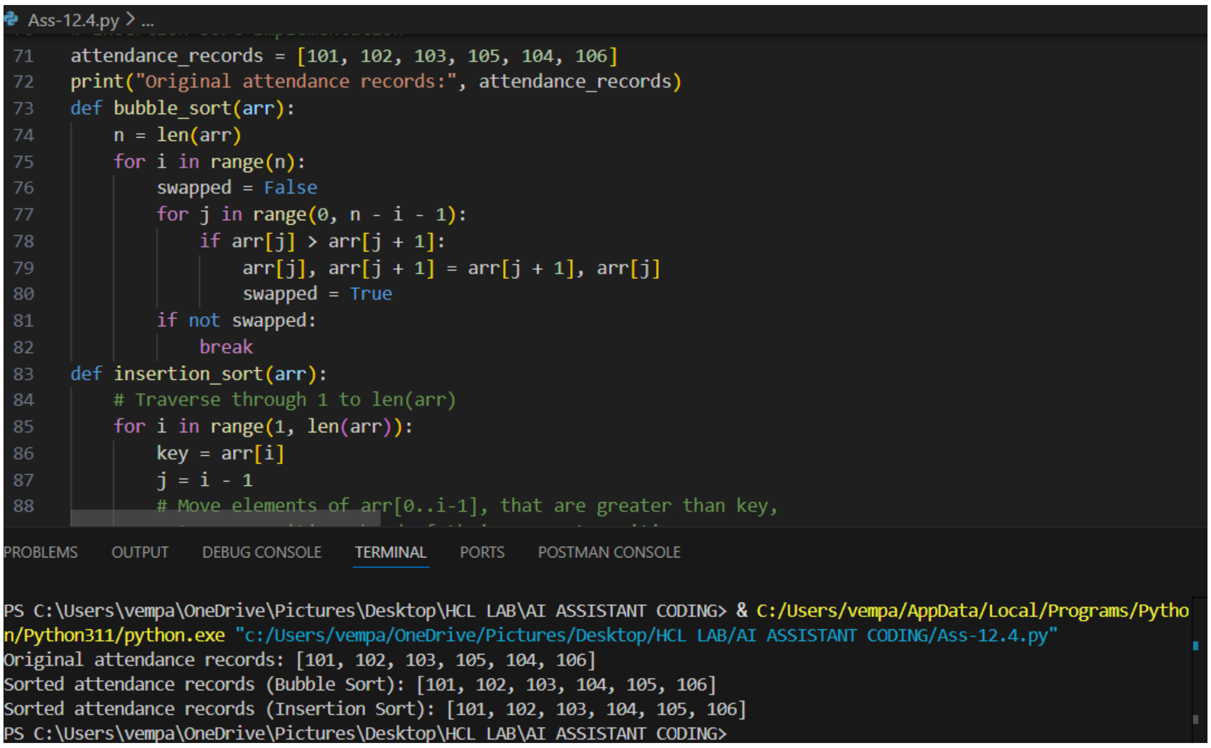
attendance_records = [101, 102, 103, 105, 104, 106]

# Sort using Insertion Sort

insertion_sort(attendance_records)

print("Sorted attendance records (Insertion Sort):", attendance_records)

Output:



```
attendance_records = [101, 102, 103, 105, 104, 106]
print("Original attendance records:", attendance_records)
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped:
            break
def insertion_sort(arr):
    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        # Move elements of arr[0..i-1], that are greater than key,
```

```
PS C:\Users\vempa\OneDrive\Pictures\Desktop\HCL LAB\AI ASSISTANT CODING> & C:/Users/vempa/AppData/Local/Programs/Pytho
n/Python311/python.exe "c:/Users/vempa/OneDrive/Pictures/Desktop/HCL LAB/AI ASSISTANT CODING/Ass-12.4.py"
Original attendance records: [101, 102, 103, 105, 104, 106]
Sorted attendance records (Bubble Sort): [101, 102, 103, 104, 105, 106]
Sorted attendance records (Insertion Sort): [101, 102, 103, 104, 105, 106]
PS C:\Users\vempa\OneDrive\Pictures\Desktop\HCL LAB\AI ASSISTANT CODING>
```

Explanation of Efficiency Differences:

Insertion Sort is more efficient than Bubble Sort for nearly sorted data because it minimizes the number of comparisons and movements. In a nearly sorted list, Insertion Sort can quickly place elements in their correct positions with fewer passes, while Bubble Sort may still require multiple passes to ensure the entire list is sorted. This results in a best-case time complexity of O(n) for Insertion Sort when the list is already sorted, compared to O(n^2) for Bubble Sort in the same scenario.

Task 3: Searching Student Records in a Database

Scenario

You are developing a student information portal where users search for student records by roll number.

Task Description

• Implement:

o Linear Search for unsorted student data

o Binary Search for sorted student data

• Use AI to:

o Add docstrings explaining parameters and return values

o Explain when Binary Search is applicable

o Highlight performance differences between the two searches

Expected Outcome

• Two working search implementations with docstrings

• AI-generated explanation of:

o Time complexity

o Use cases for Linear vs Binary Search

• A short student observation comparing results on sorted vs unsorted lists

Code:

```
def linear_search(arr, target):
    """
    Perform a linear search for the target in the given array.

    Parameters:
    arr (list): The list of student records to search through.
    target: The roll number to search for.

    Returns:
    int: The index of the target if found, otherwise -1.
    """
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Target found at index i
    return -1  # Target not found
```

```python
# Binary Search Implementation
def binary_search(arr, target):
    """
    Perform a binary search for the target in the given sorted array.

    Parameters:
    arr (list): The sorted list of student records to search through.
    target: The roll number to search for.

    Returns:
    int: The index of the target if found, otherwise -1.
    """
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            return mid  # Target found at index mid
        elif arr[mid] < target:
            left = mid + 1  # Search in the right half
        else:
            right = mid - 1  # Search in the left half
    return -1  # Target not found
# Explanation of Binary Search Applicability:
# Binary Search is applicable only when the data is sorted. It works by repeatedly dividing the search space in half, which allows it to find the target in logarithmic time. If the data is unsorted, Binary Search cannot be used, and Linear Search must be employed instead.
# Performance Differences:
# Linear Search has a time complexity of O(n) in the worst case, as it may need to check each element once.
# Binary Search has a time complexity of O(log n) in the worst case, making it significantly faster for large sorted datasets compared to Linear Search.
# Sample Data

student_records = [101, 102, 103, 104, 105]

# Searching for a record using Linear Search

index_linear = linear_search(student_records, 104)

print("Linear Search: Record found at index:", index_linear)  # Output: Linear Search

# Searching for a record using Binary Search

index_binary = binary_search(student_records, 104)

print("Binary Search: Record found at index:", index_binary)  # Output: Binary Search

# Student Observation:

# When searching for a record in an unsorted list, Linear Search is the only option,and it may
take longer as the list grows. In contrast, Binary Search is much more efficient for sorted lists,
providing faster search times as the dataset increases in size.
```

Time Complexity:

Linear Search: O(n) - In the worst case, it checks each element once.

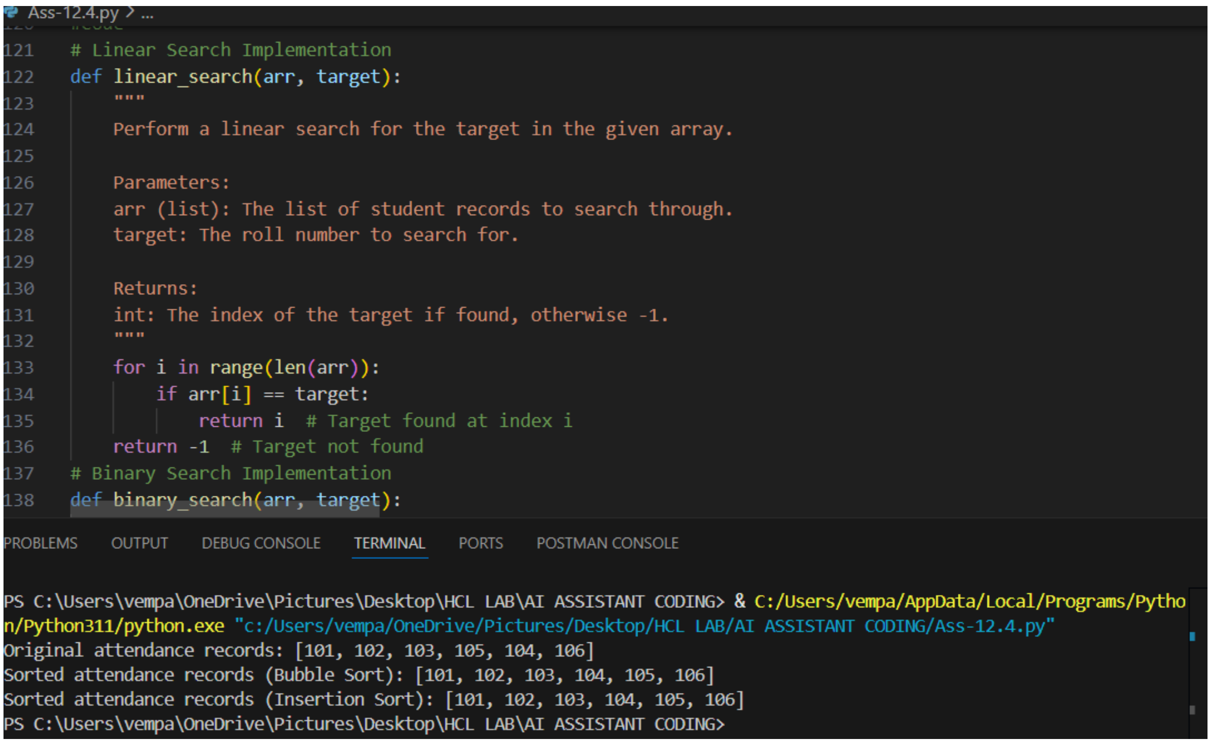
Binary Search: O(log n) - It divides the search space in half with each iteration, leading to logarithmic time complexity.

Use cases for Linear vs Binary Search

Linear Search: Use when the data is unsorted or when the dataset is small, as it does not require any preprocessing.

Binary Search: Use when the data is sorted, as it significantly reduces search time for larger datasets compared to Linear Search.

Output:


Ass-12.4.py > ...

```python
# Linear Search Implementation
def linear_search(arr, target):
    """
    Perform a linear search for the target in the given array.

    Parameters:
    arr (list): The list of student records to search through.
    target: The roll number to search for.

    Returns:
    int: The index of the target if found, otherwise -1.
    """
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Target found at index i
    return -1  # Target not found
# Binary Search Implementation
def binary_search(arr, target):
```

```
PS C:\Users\vempa\OneDrive\Pictures\Desktop\HCL LAB\AI ASSISTANT CODING> & C:/Users/vempa/AppData/Local/Programs/Pytho
n/Python311/python.exe "c:/Users/vempa/OneDrive/Pictures/Desktop/HCL LAB/AI ASSISTANT CODING/Ass-12.4.py"
Original attendance records: [101, 102, 103, 105, 104, 106]
Sorted attendance records (Bubble Sort): [101, 102, 103, 104, 105, 106]
Sorted attendance records (Insertion Sort): [101, 102, 103, 104, 105, 106]
PS C:\Users\vempa\OneDrive\Pictures\Desktop\HCL LAB\AI ASSISTANT CODING>
```

Task 4: Choosing Between Quick Sort and

Merge Sort for Data Processing

Scenario

You are part of a data analytics team that needs to sort large datasets received from different sources (random order, already sorted, and reverse sorted).

Task Description

• Provide AI with partially written recursive functions for:

o Quick Sort

o Merge Sort

• Ask AI to:

o Complete the recursive logic

o Add meaningful docstrings

o Explain how recursion works in each algorithm

• Test both algorithms on:

o Random data

o Sorted data

o Reverse-sorted data

Expected Outcome

• Fully functional Quick Sort and Merge Sort implementations

• AI-generated comparison covering:

o Best, average, and worst-case complexities

o Practical scenarios where one algorithm is preferred over the other

Code:

```
# Quick Sort Implementation
def quick_sort(arr):
    """
    Sorts an array using the Quick Sort algorithm.
    Parameters:
    arr (list): The list of elements to be sorted.
    Returns:
    list: A new sorted list.
    """
    if len(arr) <= 1:
        return arr  # Base case: a list of zero or one element is already sorted
    pivot = arr[len(arr) // 2]  # Choosing the middle element as the pivot
    left = [x for x in arr if x < pivot]  # Elements less than the pivot
    middle = [x for x in arr if x == pivot]  # Elements equal to the pivot
    right = [x for x in arr if x > pivot]  # Elements greater than the pivot
    return quick_sort(left) + middle + quick_sort(right)  # Recursive calls on left and right
partitions
# Merge Sort Implementation
def merge_sort(arr):
    """
    Sorts an array using the Merge Sort algorithm.
      Parameters:
    arr (list): The list of elements to be sorted.
    Returns:
    list: A new sorted list.
    """
    if len(arr) <= 1:
        return arr  # Base case: a list of zero or one element is already sorted
    mid = len(arr) // 2  # Finding the mid of the array
    left_half = merge_sort(arr[:mid])  # Recursively sorting the left half
    right_half = merge_sort(arr[mid:])  # Recursively sorting the right half
    return merge(left_half, right_half)  # Merging the sorted halves
def merge(left, right):
    """
     Merges two sorted lists into a single sorted list.
     Parameters:
     left (list): The first sorted list.
     right (list): The second sorted list.
     Returns:
     list: A merged sorted list.
     """
    merged = []
    i, j = 0, 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1
    # Append any remaining elements from either list
    merged.extend(left[i:])
    merged.extend(right[j:])
    return merged
#Example usage
random_data = [38, 27, 43, 3, 9, 82, 10]
sorted_random_quick = quick_sort(random_data)
sorted_random_merge = merge_sort(random_data)
print("Sorted Random Data (Quick Sort):", sorted_random_quick)
print("Sorted Random Data (Merge Sort):", sorted_random_merge)
```

Best, average, and worst-case complexities

# Quick Sort:

# Best Case: O(n log n) - occurs when the pivot divides the array into two equal halves.

# Average Case: O(n log n) - occurs for random pivots.

# Worst Case: O(n^2) - occurs when the smallest or largest element is always chosen as the pivot (e.g., sorted or reverse-sorted data).

# Merge Sort:

# Best Case: O(n log n) - occurs regardless of the input data, as it always divides the array into halves.

# Average Case: O(n log n) - occurs regardless of the input data.

# Worst Case: O(n log n) - occurs regardless of the input data, as it always divides the array into halves and merges them back together.

Practical scenarios where one algorithm is preferred over the other

# Quick Sort is preferred when:

# - Memory is limited, as it's an in-place sorting algorithm.

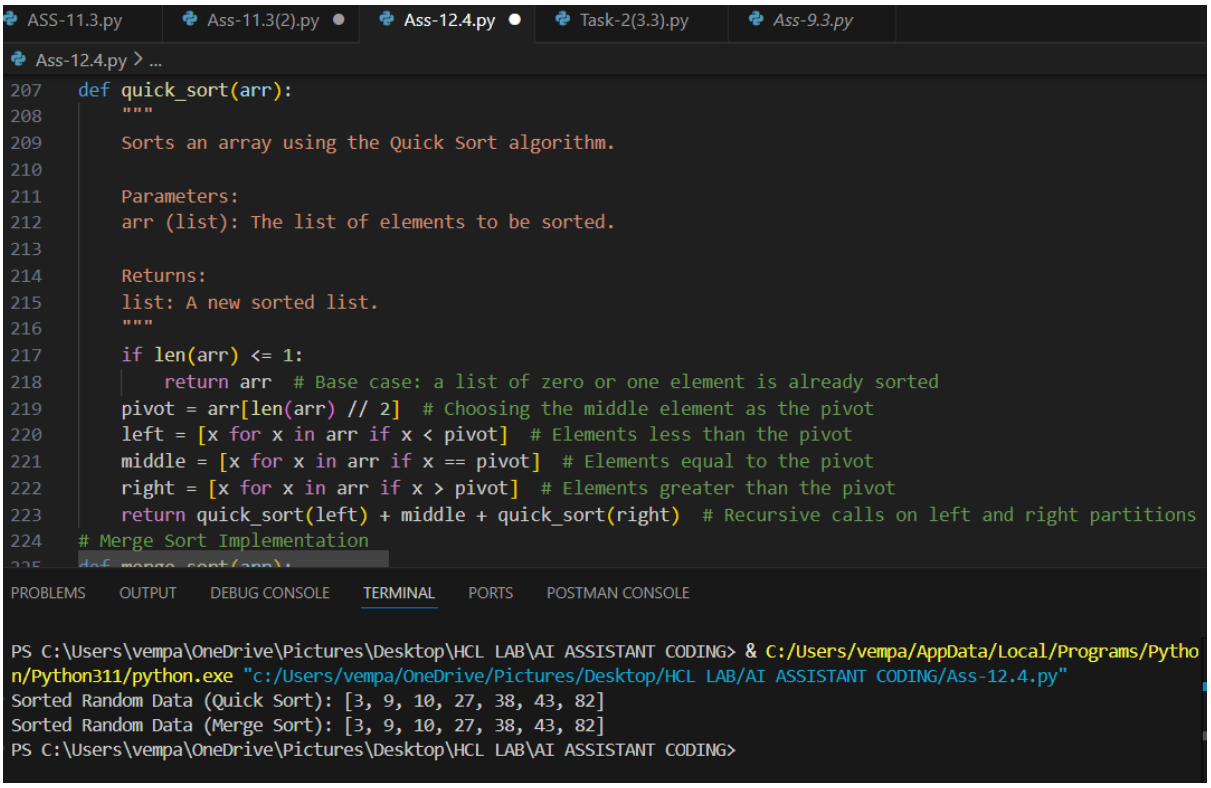
# - The data is random and average-case performance is acceptable.

# Merge Sort is preferred when:

# - Stability is required (i.e., maintaining the relative order of equal elements).

# - Consistent O(n log n) performance is needed regardless of input data.

Output:



```python
def quick_sort(arr):
    """
    Sorts an array using the Quick Sort algorithm.

    Parameters:
    arr (list): The list of elements to be sorted.

    Returns:
    list: A new sorted list.
    """
    if len(arr) <= 1:
        return arr  # Base case: a list of zero or one element is already sorted
    pivot = arr[len(arr) // 2]  # Choosing the middle element as the pivot
    left = [x for x in arr if x < pivot]  # Elements less than the pivot
    middle = [x for x in arr if x == pivot]  # Elements equal to the pivot
    right = [x for x in arr if x > pivot]  # Elements greater than the pivot
    return quick_sort(left) + middle + quick_sort(right)  # Recursive calls on left and right partitions
# Merge Sort Implementation
```

```
PS C:\Users\vempa\OneDrive\Pictures\Desktop\HCL LAB\AI ASSISTANT CODING> & C:/Users/vempa/AppData/Local/Programs/Pytho
n/Python311/python.exe "c:/Users/vempa/OneDrive/Pictures/Desktop/HCL LAB/AI ASSISTANT CODING/Ass-12.4.py"
Sorted Random Data (Quick Sort): [3, 9, 10, 27, 38, 43, 82]
Sorted Random Data (Merge Sort): [3, 9, 10, 27, 38, 43, 82]
PS C:\Users\vempa\OneDrive\Pictures\Desktop\HCL LAB\AI ASSISTANT CODING>
```

Task 5: Optimizing a Duplicate Detection Algorithm

Scenario

You are building a data validation module that must detect duplicate

user IDs in a large dataset before importing it into a system.

Task Description

• Write a naive duplicate detection algorithm using nested loops.

• Use AI to:

o Analyze the time complexity

o Suggest an optimized approach using sets or dictionaries

o Rewrite the algorithm with improved efficiency

• Compare execution behavior conceptually for large input sizes

Expected Outcome

• Two versions of the algorithm:

o Brute-force ($O(n^2)$)

o Optimized (O(n))

• AI-assisted explanation showing how and why performance improved

Code:

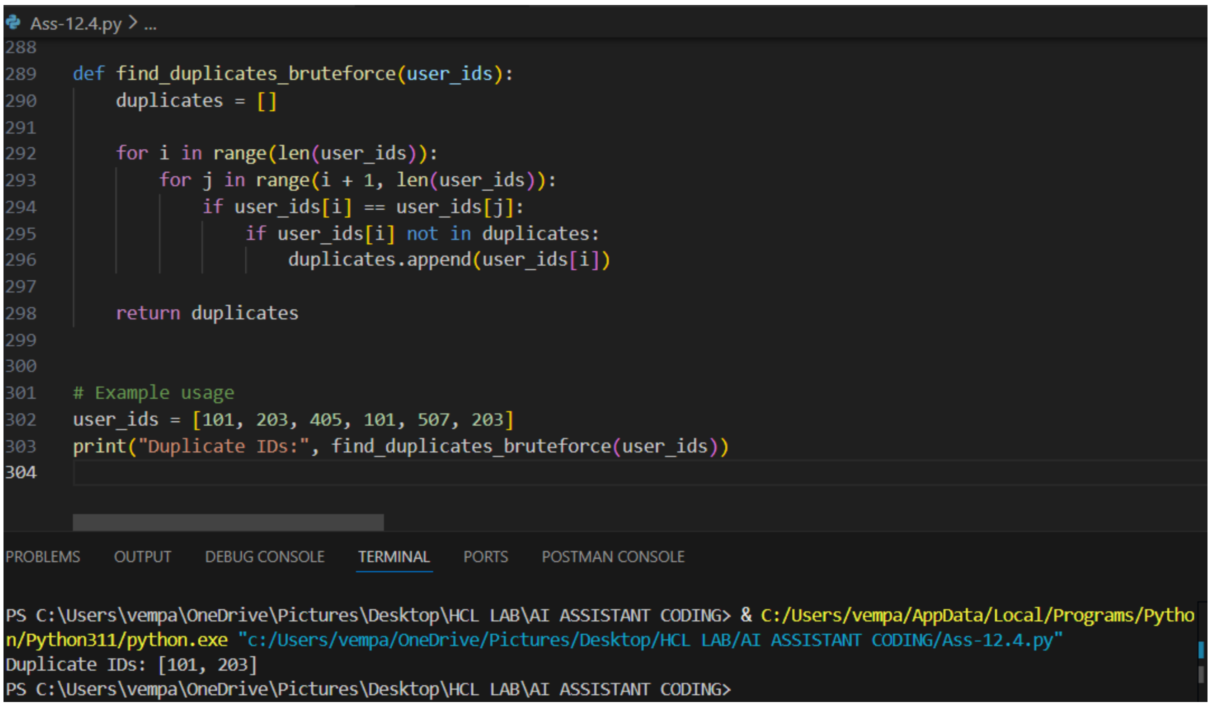
Brute-Force

```
def find_duplicates_bruteforce(user_ids):
    duplicates = []

    for i in range(len(user_ids)):
        for j in range(i + 1, len(user_ids)):
            if user_ids[i] == user_ids[j]:
                if user_ids[i] not in duplicates:
                    duplicates.append(user_ids[i])

    return duplicates
# Example usage
user_ids = [101, 203, 405, 101, 507, 203]
print("Duplicate IDs:", find_duplicates_bruteforce(user_ids))
```

Output:



```
def find_duplicates_bruteforce(user_ids):
    duplicates = []

    for i in range(len(user_ids)):
        for j in range(i + 1, len(user_ids)):
            if user_ids[i] == user_ids[j]:
                if user_ids[i] not in duplicates:
                    duplicates.append(user_ids[i])

    return duplicates


# Example usage
user_ids = [101, 203, 405, 101, 507, 203]
print("Duplicate IDs:", find_duplicates_bruteforce(user_ids))
```

```
PS C:\Users\vempa\OneDrive\Pictures\Desktop\HCL LAB\AI ASSISTANT CODING> & C:/Users/vempa/AppData/Local/Programs/Pytho
n/Python311/python.exe "c:/Users/vempa/OneDrive/Pictures/Desktop/HCL LAB/AI ASSISTANT CODING/Ass-12.4.py"
Duplicate IDs: [101, 203]
PS C:\Users\vempa\OneDrive\Pictures\Desktop\HCL LAB\AI ASSISTANT CODING>
```

Time Complexity Analysis

- Outer loop runs **n times**
- Inner loop runs approximately **n times**

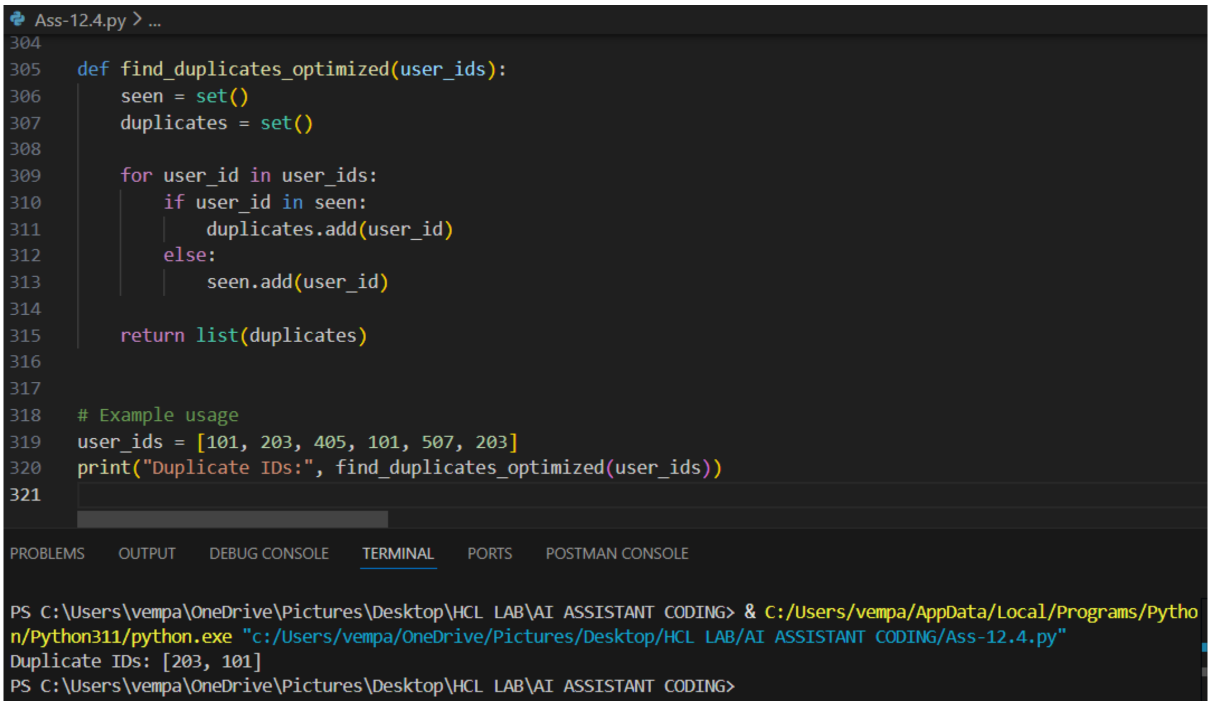
- Total comparisons ≈ n × n
- **Time Complexity: O(n²)**

Optimised Code

```
def find_duplicates_optimized(user_ids):
    seen = set()
    duplicates = set()

    for user_id in user_ids:
        if user_id in seen:
            duplicates.add(user_id)
        else:
            seen.add(user_id)
    return list(duplicates)
# Example usage
user_ids = [101, 203, 405, 101, 507, 203]
print("Duplicate IDs:", find_duplicates_optimized(user_ids))
```

Output



```
def find_duplicates_optimized(user_ids):
    seen = set()
    duplicates = set()

    for user_id in user_ids:
        if user_id in seen:
            duplicates.add(user_id)
        else:
            seen.add(user_id)

    return list(duplicates)


# Example usage
user_ids = [101, 203, 405, 101, 507, 203]
print("Duplicate IDs:", find_duplicates_optimized(user_ids))
```

```
PS C:\Users\vempa\OneDrive\Pictures\Desktop\HCL LAB\AI ASSISTANT CODING> & C:/Users/vempa/AppData/Local/Programs/Pytho
n/Python311/python.exe "c:/Users/vempa/OneDrive/Pictures/Desktop/HCL LAB/AI ASSISTANT CODING/Ass-12.4.py"
Duplicate IDs: [203, 101]
PS C:\Users\vempa\OneDrive\Pictures\Desktop\HCL LAB\AI ASSISTANT CODING>
```

Time Complexity Analysis

- Loop runs once → **O(n)**
- Set lookup → **O(1) average**
- Total complexity → **O(n)**

How Performance Improved:

- The performance improvement occurred because the algorithm shifted from:
- **Comparison-based detection (O(n^2))**
- to
- **Hash-based lookup detection (O(n))**
- By leveraging Python's set data structure, we reduced the number of operations from quadratic growth to linear growth, making the system scalable, efficient, and suitable for large datasets.