AIAC-Ass-12.4

Name:T.Shivani

Ht.No:2303A51312

Batch:05

Prompt: You are working on a college result processing system where a small

list of student scores needs to be sorted after every internal assessment.Implement Bubble

Sort in Python to sort a list of student scores.

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

• A Bubble Sort implementation with:

o AI-generated comments explaining the logic

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

complexity

Sample input/output showing sorted scores

Code: scores = [85, 92, 78, 96, 88]

print("Original scores:", scores)

def bubble_sort(arr):

n = len(arr)

Outer loop for number of passes

for i in range(n):

swapped = False # Early termination flag

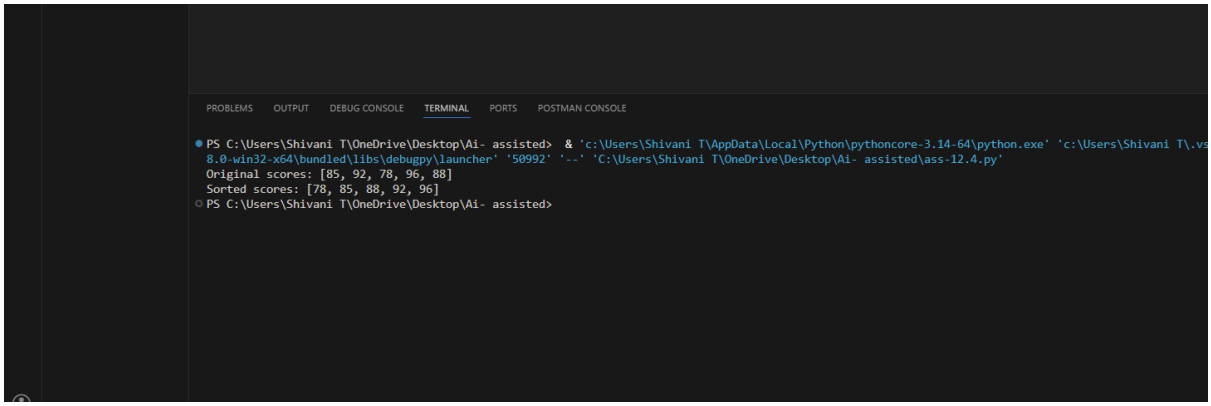
Inner loop for comparisons in each pass

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

Compare adjacent elements

if arr[j] > arr[j + 1]:# Swap if they are in wrong order

```
arr[j], arr[j + 1] = arr[j + 1], arr[j]
swapped = True
# If no swapping occurred, list is sorted
if not swapped:
break
bubble_sort(scores)
print("Sorted scores:", scores)
# Time Complexity Analysis:
# Best Case: O(n) - This occurs when the list is already sorted, and we only make one pass to check that it is sorted.
# Average Case: O(n^2) - This occurs when the list is in random order, requiring multiple passes and comparisons.
# Worst Case: O(n^2) - This occurs when the list is sorted in reverse
# order, requiring the maximum number of comparisons and swaps.
```



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).

Prompt: You are maintaining an attendance system where student roll numbersare already almost sorted, with only a few late updates. Start with a Bubble Sort

implementation. 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. Two sorting implementations:

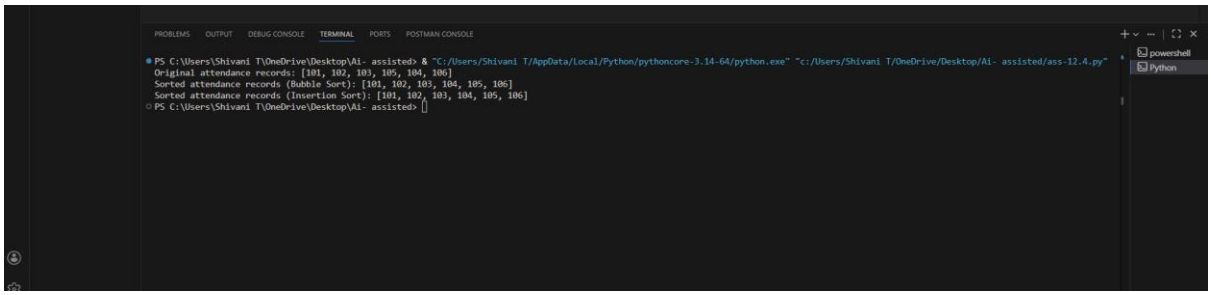
o Bubble Sort

o Insertion Sort

Code: Sample input for nearly sorted attendance records

```
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
```



Explanation: 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.

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

o Linear Search for unsorted student datao Binary Search for sorted student data. Add docstrings explaining parameters and return

values

o Explain when Binary Search is applicable

o Highlight performance differences between the two

searches

• 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

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

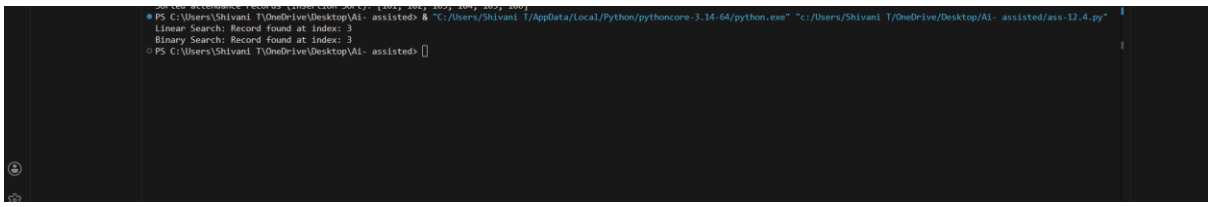
elif arr[mid] < target:

left = mid + 1 # Search in the right half

else:

right = mid - 1 # Search in the left half

return -1 # Target not found



Explanation: 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

Prompt: 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).

• 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

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: # Sample input

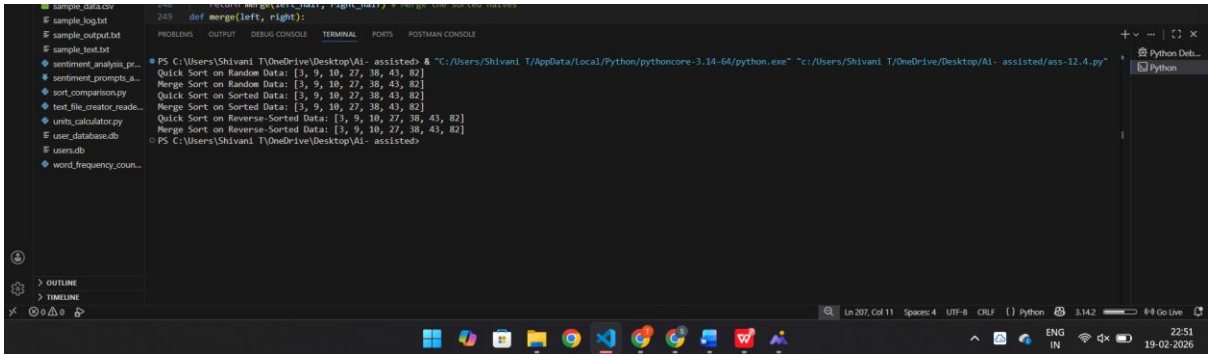
```
data_random = [38, 27, 43, 3, 9, 82, 10]
data_sorted = [3, 9, 10, 27, 38, 43, 82]
data_reverse_sorted = [82, 43, 38, 27, 10, 9, 3]
# Quick Sort Implementation
def quick_sort(arr):
    """
```

```
    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
    else:
        pivot = arr[len(arr) // 2] # Choosing the middle element as pivot
        left = [x for x in arr if x < pivot] # Elements less than pivot
        middle = [x for x in arr if x == pivot] # Elements equal to pivot
        right = [x for x in arr if x > pivot] # Elements greater than pivot
        return quick_sort(left) + middle + quick_sort(right) # Recursively sort 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
```

```
    mid = len(arr) // 2 # Finding the mid of the array
    left_half = merge_sort(arr[:mid]) # Recursively sort the left half
    right_half = merge_sort(arr[mid:]) # Recursively sort the right half
    return merge(left_half, right_half) # Merge the sorted halves
def merge(left, right):
    """Merges two sorted lists into a single sorted list.
    """
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:]) # Append remaining elements of left
    result.extend(right[j:]) # Append remaining elements of right
    return result
# Testing Quick Sort and Merge Sort on different datasets
print("Quick Sort on Random Data:", quick_sort(data_random))
print("Merge Sort on Random Data:", merge_sort(data_random))
print("Quick Sort on Sorted Data:", quick_sort(data_sorted))
print("Merge Sort on Sorted Data:", merge_sort(data_sorted))
print("Quick Sort on Reverse-Sorted Data:", quick_sort(data_reverse_sorted))
print("Merge Sort on Reverse-Sorted Data:", merge_sort(data_reverse_sorted))
```



Explanation: # Complexity Analysis:

Quick Sort:

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

Average Case: O(n log n) - This occurs when the pivot is reasonably well-chosen.

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

Merge Sort: Best Case: O(n log n) - This occurs regardless of the input, as Merge Sort always divides the list and merges it back together.

Average Case: O(n log n) - This occurs regardless of the input, as Merge Sort always divides the list and merges it back together.

Worst Case: O(n log n) - This occurs regardless of the input, as Merge Sort always divides the list and merges it back together.

Prompt: You are building a data validation module that must detect duplicate

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

• 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: Sample input

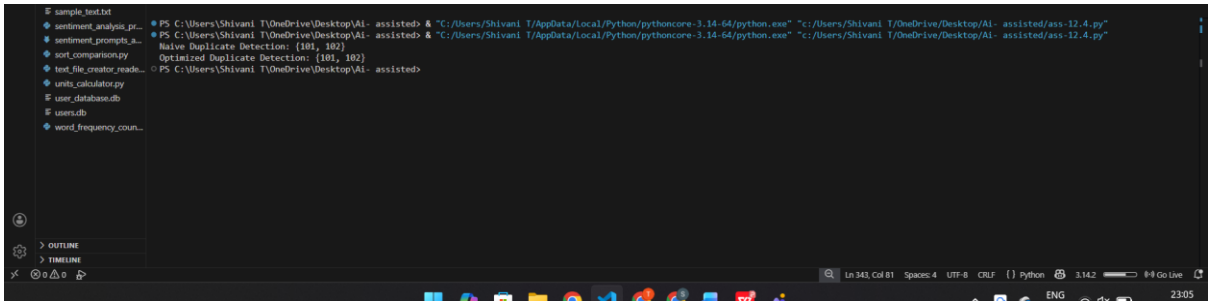
```
user_ids = [101, 102, 103, 104, 105, 101, 106, 107, 102]
# Naive Duplicate Detection Algorithm
def naive_duplicate_detection(ids):
    """
    Detects duplicates in a list of user IDs using a brute-force approach.

    Parameters:
    ids (list): A list of user IDs.

    Returns:
    set: A set of duplicate user IDs.
    """
    duplicates = set()
    for i in range(len(ids)):
        for j in range(i + 1, len(ids)):
            if ids[i] == ids[j]:
                duplicates.add(ids[i])
    return duplicates
# Optimized Duplicate Detection Algorithm
def optimized_duplicate_detection(ids):
    """
    Detects duplicates in a list of user IDs using an optimized approach with a set.

    Parameters:
    ids (list): A list of user IDs.
```

```
    Returns:
    set: A set of duplicate user IDs.
    """
    seen = set()
    duplicates = set()
    for id in ids:
        if id in seen:
            duplicates.add(id)
        else:
            seen.add(id)
    return duplicates
# Testing both algorithms
print("Naive Duplicate Detection:", naive_duplicate_detection(user_ids))
print("Optimized Duplicate Detection:", optimized_duplicate_detection(user_ids))
```



Explanation: The naive approach has a time complexity of $O(n^2)$ because it compares each element with every other element in the list. In contrast, the optimized approach uses a set to track seen elements, achieving a time complexity of O(n). This is because checking membership in a set is on average O(1), making the overall algorithm much more efficient for large datasets.