AI ASSISTED CODING – 12.4

2403A52081

- **Task1:** This task involves writing a standard Bubble Sort algorithm and then asking an AI to generate comments and a complexity analysis.
- **Prompt:** I have this Python implementation of Bubble Sort. Can you please add inline comments to the code to explain the key logic, like the purpose of each loop, the swapping mechanism, and the termination condition? Also, could you provide a detailed time and space complexity analysis for it?

Code Before Prompting AI:

```python
def bubble_sort_with_ai_comments(arr):
    """
    Sorts a list of numbers in ascending order using the Bubble Sort algorithm.

    Args:
        arr (list): A list of comparable elements (e.g., numbers).

    Returns:
        list: The sorted list.
    """

    n = len(arr)


    for i in range(n):

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

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

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

# --- Example Usage ---
my_list = [64, 34, 25, 12, 22, 11, 90]
print("--- Task 1: Bubble Sort ---")
print("Original list:", my_list)
sorted_list = bubble_sort_with_ai_comments(my_list.copy())
print("Sorted list:", sorted_list)
print("-" * 20)
```

Name: S.Sai Charan Reddy
Enrol: 2403A52081
Batch: 04

CODE AFTER EXECUTING THE PROMPT:

```python
def bubble_sort_with_ai_comments(arr):
    """
    Sorts a list of numbers in ascending order using the Bubble Sort algorithm.

    Args:
        arr (list): A list of comparable elements (e.g., numbers).

    Returns:
        list: The sorted list.
    """

    n = len(arr)  # Get the length of the list

    # Outer loop: Runs n times, each pass moves the next largest element to its correct position
    for i in range(n):
        # Inner loop: Compares adjacent elements up to the unsorted portion
        for j in range(0, n - i - 1):
            # If the current element is greater than the next, swap them
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]  # Swapping mechanism
    # After all passes, the list is sorted
    return arr

# --- Example Usage ---
my_list = [64, 34, 25, 12, 22, 11, 90]
print("--- Task 1: Bubble Sort ---")
print("Original list:", my_list)
sorted_list = bubble_sort_with_ai_comments(my_list.copy())
print("Sorted list:", sorted_list)
print("-" * 20)

# --- Complexity Analysis ---
# Time Complexity:
#   - Worst-case: O(n^2) (when the list is in reverse order)
#   - Best-case: O(n^2) (unless optimized with a swapped flag)
#   - Each pass compares adjacent elements, and there are n passes.
#
# Space Complexity:
#   - O(1) auxiliary space (sorting is done in-place; only a constant amount of extra memory is
used)
```

OBSERVATION:

When you run the code, the output correctly shows the original unsorted list and the final list sorted in ascending order. The key observation is how the AI-generated comments integrated into the code make the algorithm's logic, which can be confusing for beginners, much easier to understand step-by-step.

Name: S.Sai Charan Reddy
Enrol: 2403A52081
Batch: 04

➢ TASK 2:

Here, we give our Bubble Sort code to an AI and ask for a more efficient alternative for partially sorted arrays.

PROMPT:

I have this Bubble Sort code. I've noticed it performs poorly even on lists that are only slightly out of order. Can you suggest a more efficient sorting algorithm specifically for partially sorted arrays? Please provide the Python code for the new algorithm and an explanation of why it's better in this scenario.

CODE Before Prompt:

```python
def bubble_sort(arr):

    """A simple Bubble Sort implementation."""
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

Code After Prompt:

```python
def insertion_sort(arr):

    """Efficient for partially sorted arrays."""
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        # Move elements greater than key to one position ahead
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

# Example usage:
my_list = [2, 3, 5, 4, 6, 7, 8]  # Nearly sorted
print("Original:", my_list)
print("Sorted:", insertion_sort(my_list.copy()))
```

Observation:

Both functions correctly sort the nearly_sorted_list. While a simple print won't show the performance difference, if you were to time these functions on a very large, nearly-sorted list, you would observe that Insertion Sort finishes dramatically faster than Bubble Sort. This confirms the AI's explanation that Insertion Sort is better suited for this specific use case.

Name: S.Sai Charan Reddy
Enrol: 2403A52081
Batch: 04

➤ Task 3:

This task involves implementing both search algorithms and using an AI to generate docstrings and explain their trade-offs.

➤ Prompt:

I've written two Python functions: one for Linear Search and one for Binary Search. Can you generate professional docstrings for each that explain what they do, their arguments, what they return, and a performance note on their time complexity? Also, please provide a clear explanation of when Binary Search is preferable to Linear Search.

CODE:

```
def linear_search(arr, target):
    """
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1

def binary_search(arr, target):
        else:
            high = mid - 1
    return -1

# --- Example Usage ---
sorted_data = [11, 12, 22, 25, 34, 64, 90]
unsorted_data = [64, 34, 25, 12, 22, 11, 90]
target_val = 22

print("\n--- Task 3: Linear vs Binary Search ---")
print("Searching for", target_val)
print("Linear search on unsorted data:", linear_search(unsorted_data, target_val))
print("Linear search on sorted data:  ", linear_search(sorted_data, target_val))
print("Binary search on sorted data:  ", binary_search(sorted_data, target_val))
print("Binary search on unsorted data:", binary_search(unsorted_data, target_val))
print("-" * 20)
```

Output For First Function:

```
    """
    AI-Generated Docstring:
    Performs a linear search to find the index of a target element in a list.

    This function iterates through each element of the list from the beginning
    and compares it with the target value. It works on both sorted and unsorted lists.

    Args:
        arr (list): The list to be searched.
        target: The value to search for.
```

Name: S.Sai Charan Reddy
Enrol: 2403A52081
Batch: 04

```
    Returns:
        int: The index of the target element if found, otherwise -1.

    Performance Note: Time Complexity is O(n).
```

Output For Second Function:

```
    """
    AI-Generated Docstring:
    Performs a binary search to find the index of a target element in a sorted list.

    This function works by repeatedly dividing the search interval in half.
    IMPORTANT: This algorithm requires the input list `arr` to be sorted.

    Args:
        arr (list): The sorted list to be searched.
        target: The value to search for.

    Returns:
        int: The index of the target element if found, otherwise -1.

    Performance Note: Time Complexity is O(log n).
    """
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
```

Observation:

The output clearly demonstrates the main trade-off. Linear Search works correctly on both sorted and unsorted data. Binary Search works correctly and efficiently on the sorted list but fails on the unsorted list, returning -1 (or an incorrect index) even though the element exists. This is the most crucial observation: Binary Search's speed comes at the cost of requiring sorted input.

Name: S.Sai Charan Reddy
Enrol: 2403A52081
Batch: 04

Task 4:

This task involves using AI to complete partially written recursive sorting algorithms.

Prompt:

I'm trying to implement Quick Sort and Merge Sort recursively in Python but I'm stuck. Here are my partially completed functions. Can you complete the missing logic? Please also add docstrings and an explanation of their average, best, and worst-case complexities.

Code Before Giving Prompt to AI:

```
def quick_sort(arr):
    """
    AI-Generated Docstring:
    Sorts a list using the Quick Sort algorithm via recursion.
    It selects a 'pivot' and partitions other elements into two sub-arrays
    (less than or greater than the pivot), which are then sorted recursively.
    """
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[len(arr) // 2]
        left = [x for x in arr if x < pivot]
        middle = [x for x in arr if x == pivot]
        right = [x for x in arr if x > pivot]
        return quick_sort(left) + middle + quick_sort(right)

# --- AI-Completed Merge Sort ---
def merge_sort(arr):
    """
    AI-Generated Docstring:
    Sorts a list using the Merge Sort algorithm via recursion.
    A 'divide and conquer' algorithm, it divides the array into halves,
    recursively sorts them, and then merges the sorted halves.
    """
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        # --- AI-Completed Merge Logic ---
        i = j = k = 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
```

Name: S.Sai Charan Reddy
Enrol: 2403A52081
Batch: 04

```
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
    return arr

# --- Example Usage ---
random_list = [5, 3, 8, 4, 2, 7, 1, 6]
sorted_list_test = [1, 2, 3, 4, 5, 6, 7, 8]
reverse_sorted_list = [8, 7, 6, 5, 4, 3, 2, 1]

print("\n--- Task 4: Quick Sort vs Merge Sort ---")
print("Quick Sort on random list:    ", quick_sort(random_list.copy()))
print("Merge Sort on random list:    ", merge_sort(random_list.copy()))
print("Quick Sort on reverse list:   ", quick_sort(reverse_sorted_list.copy()))
print("Merge Sort on reverse list:   ", merge_sort(reverse_sorted_list.copy()))
print("-" * 20)
```

Code after Giving Prompt to AI:

```
def quick_sort(arr):
    """
    Sorts a list using the Quick Sort algorithm (recursive, functional version).

    Quick Sort selects a 'pivot' element and partitions the other elements into two sub-arrays,
    according to whether they are less than or greater than the pivot. The sub-arrays are then
    sorted recursively and concatenated with the pivot(s).

    Args:
        arr (list): The list of elements to sort.

    Returns:
        list: A new sorted list.

    Complexity:
        Average Case: O(n log n)
        Best Case:    O(n log n)
        Worst Case:   O(n^2) (when the smallest/largest element is always chosen as pivot)
    """
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[len(arr) // 2]  # Choose 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)
```

Name: S.Sai Charan Reddy
Enrol: 2403A52081
Batch: 04

```python
def merge_sort(arr):
    """
    Sorts a list using the Merge Sort algorithm (recursive, in-place version).

    Merge Sort is a divide-and-conquer algorithm that divides the list into halves,
    recursively sorts each half, and then merges the sorted halves.

    Args:
        arr (list): The list of elements to sort.

    Returns:
        list: The sorted list (sorted in-place).

    Complexity:
        Average Case: O(n log n)
        Best Case:    O(n log n)
        Worst Case:   O(n log n)
    """
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        # Merge the sorted halves
        i = j = k = 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        # Copy any remaining elements of left_half
        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        # Copy any remaining elements of right_half
        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
    return arr

# --- Example Usage ---
random_list = [5, 3, 8, 4, 2, 7, 1, 6]
sorted_list_test = [1, 2, 3, 4, 5, 6, 7, 8]
```

Name: S.Sai Charan Reddy
Enrol: 2403A52081
Batch: 04

```
reverse_sorted_list = [8, 7, 6, 5, 4, 3, 2, 1]

print("\n--- Task 4: Quick Sort vs Merge Sort ---")
print("Quick Sort on random list:    ", quick_sort(random_list.copy()))
print("Merge Sort on random list:    ", merge_sort(random_list.copy()))
print("Quick Sort on reverse list:   ", quick_sort(reverse_sorted_list.copy()))
print("Merge Sort on reverse list:   ", merge_sort(reverse_sorted_list.copy()))
print("-" * 20)

# --- Explanation of Complexities ---
# Quick Sort:
#   - Average Case: O(n log n)
#   - Best Case:    O(n log n)
#   - Worst Case:   O(n^2) (e.g., already sorted or reverse sorted with poor pivot choice)
#
# Merge Sort:
#   - Average Case: O(n log n)
#   - Best Case:    O(n log n)
#   - Worst Case:   O(n log n)
#   - Merge Sort guarantees O(n log n) time but uses extra space for merging.
```

Observation:

Both algorithms correctly sort the lists. The important observation comes from considering the complexity analysis. Although both are fast on a random list, Merge Sort's performance is stable and reliable, whereas Quick Sort's performance can degrade to O(n2) on sorted or reverse-sorted data (its "worst case"). This makes Merge Sort a better choice when you need a guarantee of O(nlogn) performance, regardless of the input.

Task 5:

This task involves giving an AI a naive, inefficient algorithm and asking it to optimize it.

Prompt:

I wrote this Python function to find if a list contains duplicate values. It uses nested loops, and I'm sure it's not efficient. Can you optimize this function to run faster, especially on large lists? Please provide the optimized code and explain how you improved the time complexity.

CODE Before GIVING Prompt:

```
import time


def find_duplicates_brute_force(arr):
    """A naive O(n^2) function to find if duplicates exist."""
    for i in range(len(arr)):
        for j in range(i + 1, len(arr)):
            if arr[i] == arr[j]:
                return True  # Found a duplicate
    return False
```

CODE After Giving Prompt:

```
def find_duplicates_optimized(arr):

    """
    An AI-optimized O(n) function to find if duplicates exist
    by leveraging the O(1) average time complexity of set lookups.
    """
    seen = set()
    for item in arr:
        if item in seen:
            return True  # Found a duplicate
        seen.add(item)
    return False
```

Observation:

```
# --- Comparison with large input ---

large_list_no_duplicates = list(range(10000))
large_list_with_duplicate = list(range(10000)) + [5000]

print("\n--- Task 5: Algorithm Optimization ---")

# --- Brute Force ---
start_time = time.time()
find_duplicates_brute_force(large_list_with_duplicate)
end_time = time.time()
print(f"Brute Force Time: {end_time - start_time:.6f} seconds")
```

Name: S.Sai Charan Reddy
Enrol: 2403A52081
Batch: 04

```python
# --- Optimized ---
start_time = time.time()
find_duplicates_optimized(large_list_with_duplicate)
end_time = time.time()
print(f"Optimized Time:   {end_time - start_time:.6f} seconds")
print("-" * 20)
```

When you run this comparison, the observation is immediate and powerful. The brute-force method takes a noticeable amount of time (potentially several seconds). The optimized method finishes almost instantly. This demonstrates the real-world impact of choosing the right data structure and algorithm, validating the AI's explanation that improving from O(n2) to O(n) results in a massive performance gain for large inputs.

Name: S.Sai Charan Reddy
Enrol: 2403A52081
Batch: 04