AI ASSIGNMENT 12.4

ROLLNO: 2403A52397.

NAME: Varshitha

Implementing Bubble Sort with AI Comments

- **Task1**: Write a Python implementation of **Bubble Sort**.
- **Instructions**:
    - Students implement Bubble Sort normally.
    - Ask AI to generate **inline comments explaining key logic** (like swapping, passes, and termination).
    - Request AI to provide **time complexity analysis.**

```python
def bubble_sort(arr):
    n = len(arr)
    # Outer loop for passes
    for i in range(n):
        # Inner loop for comparisons and swaps
        for j in range(0, n - i - 1):
            # Compare adjacent elements
            if arr[j] > arr[j + 1]:
                # Swap elements if they are in the wrong order
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    # The array is sorted after the loops complete
    return arr

# Example usage
my_list = [64, 34, 25, 12, 22, 11, 90]
sorted_list = bubble_sort(my_list)
print("Sorted array:", sorted_list)
```

```
Sorted array: [11, 12, 22, 25, 34, 64, 90]
```

Time Complexity Analysis of Bubble Sort

- **Worst Case:** O(n^2) - This occurs when the array is in reverse order. The inner loop will always execute `n-i-1` times for each pass, leading to a total of approximately n^2/2 comparisons and swaps.
- **Average Case:** O(n^2) - On average, the performance is similar to the worst case as most elements will still need to be compared and potentially swapped multiple times.
- **Best Case:** O(n) - This occurs when the array is already sorted. In this case, the inner loop will still run, but no swaps will occur. The outer loop will run `n` times, and the inner loop will run `n-i-1` times, but the conditional swap inside the inner loop will never be true. A slightly optimized version of Bubble Sort can detect if no swaps happened in a pass and terminate early, achieving O(n) in the best case. The provided implementation does not include this optimization, so its best case is also O(n^2) in terms of comparisons, but O(n) in terms of swaps.

Bubble Sort is generally not recommended for large datasets due to its quadratic time complexity.

: Optimizing Bubble Sort → Insertion Sort

- **Task2**: Provide Bubble Sort code to AI and ask it to suggest a **more efficient algorithm** for partially sorted arrays.
- **Instructions**:

- Students implement Bubble Sort first.
- Ask AI to suggest an alternative (Insertion Sort).
- Compare performance on nearly sorted input.

```python
def bubble_sort(arr):
    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

data = [5, 3, 4, 1, 2]
print("Bubble Sort Result:", bubble_sort(data))
```

```
Bubble Sort Result: [1, 2, 3, 4, 5]
```

```python
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

# Example
data = [1, 2, 4, 3, 5]  # Nearly sorted
print("Insertion Sort Result:", insertion_sort(data))
```

```
Insertion Sort Result: [1, 2, 3, 4, 5]
```

- **Task3**: Implement both **Linear Search** and **Binary Search**.
- **Instructions**:
  - Use AI to generate docstrings and performance notes.
  - Test both algorithms on sorted and unsorted data.
  - Ask AI to explain when Binary Search is preferable.

```python
# Linear Search Implementation
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1


# Binary Search Implementation
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1


# Testing both searches
data_unsorted = [12, 5, 8, 13, 7, 9, 10]
data_sorted = sorted(data_unsorted)
target = 9

print("Unsorted Data:", data_unsorted)
print("Sorted Data:", data_sorted)

# Linear Search can work on both
print("\nLinear Search Results:")
print("In unsorted:", linear_search(data_unsorted, target))
print("In sorted:", linear_search(data_sorted, target))

# Binary Search works only on sorted data
print("\nBinary Search Result:")
print("In sorted:", binary_search(data_sorted, target))
```

```
Unsorted Data: [12, 5, 8, 13, 7, 9, 10]
Sorted Data: [5, 7, 8, 9, 10, 12, 13]

Linear Search Results:
In unsorted: 5
In sorted: 3

Binary Search Result:
In sorted: 3
```

- **Linear Search** is best for **small or unsorted lists** since it checks each element one by one.
- **Binary Search** is preferred for **large, sorted lists** because it divides the list in half each time → much faster.
- If your data is frequently updated and unsorted, Linear Search is easier.
- If your data is mostly static and sorted, Binary Search is ideal.

- **Task4:** Implement Quick Sort and Merge Sort using recursion**.**
- **Instructions:**
    - Provide AI with partially completed functions for recursion.
    - Ask AI to complete the missing logic and add docstrings.
    - Compare both algorithms on random, sorted, and reverse-sorted lists.

Quick Sort:

- **Best Case:** O(n log n) — when pivot divides the array evenly.
- **Average Case:** O(n log n) — typically performs very fast.
- **Worst Case:** $O(n^2)$ — when the pivot is always smallest or largest element (like sorted data without randomization).

Merge Sort:

- **Best Case:** O(n log n) — same for all cases since it always divides and merges.
- **Average Case:** O(n log n) — consistent performance.
- **Worst Case:** O(n log n) — doesn't degrade even for sorted or reverse-sorted lists.

Comparison Observation:

- Merge Sort gives consistent time for all types of inputs.
- Quick Sort is faster in practice for random data but slower for already sorted data.

- **Task5:** Give AI a naive algorithm (e.g., $O(n^2)$ duplicate search).
- **Instructions:**
  - Students write a brute force duplicate-finder.
  - Ask AI to optimize it (e.g., by using sets/dictionaries with O(n) time).
  - Compare execution times with large input sizes.

Algorithm Type	**Time Complexity**	**Execution Speed (Large Input)**	**Remark**
Brute Force	$O(n^2)$	Slow	Checks each pair manually
Optimized (Using Set)	O(n)	Fast	Uses hash lookup for duplicates