AI ASSISTED CODING

NAME: G.SANJANA

ENROLL NUMBER: 2403A52380

BATCH NUMBER :14

Lab assignment-12.4

Task 1: Implementing Bubble Sort with AI Comments

- **Task**: 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
# Bubble Sort Implementation in Python with AI Comments

def bubble_sort(arr):
    n = len(arr)  # Get the total number of elements in the list

    # Outer loop for each pass through the array
    for i in range(n):
        # A flag to detect if any swapping happened during this pass
        swapped = False

        # Inner loop compares adjacent elements
        for j in range(0, n - i - 1):
            # Compare current element with the next one
            if arr[j] > arr[j + 1]:
                # Swap if elements are in the wrong order
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True  # Mark that a swap occurred

        # If no swaps occurred in this pass, the list is already sorted
        if not swapped:
            break  # Exit early to save time (optimization)

    return arr


# Example usage:
data = [64, 34, 25, 12, 22, 11, 90]
print("Original list:", data)
sorted_data = bubble_sort(data)
print("Sorted list:", sorted_data)
```

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

Task 2: Optimizing Bubble Sort → Insertion Sort

- **Task**: 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
# Bubble Sort Implementation (Baseline)

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  # Early termination if no swaps in this pass
    return arr
# Insertion Sort Implementation (Optimized for Nearly Sorted Arrays)

def insertion_sort(arr):
    # Traverse from the 2nd element since the first element is already "sorted"
    for i in range(1, len(arr)):
        key = arr[i]  # Element to be inserted into the sorted portion
        j = i - 1

        # Move elements greater than key one position ahead
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1

        # Place the key at its correct position
        arr[j + 1] = key
    return arr
import time

# Nearly sorted input (only a few elements out of order)
data_nearly_sorted = [10, 20, 30, 25, 40, 50, 55, 60]
```

```
[4]
 0s
# Copy data for both algorithms
arr_bubble = data_nearly_sorted.copy()
arr_insertion = data_nearly_sorted.copy()

# Test Bubble Sort
start_bubble = time.time()
bubble_sort(arr_bubble)
end_bubble = time.time()

# Test Insertion Sort
start_insertion = time.time()
insertion_sort(arr_insertion)
end_insertion = time.time()

# Print results
print("Original nearly sorted data:", data_nearly_sorted)
print("Bubble Sort result:", arr_bubble)
print("Insertion Sort result:", arr_insertion)

print("\nExecution Time:")
print(f"Bubble Sort: {end_bubble - start_bubble:.8f} seconds")
print(f"Insertion Sort: {end_insertion - start_insertion:.8f} seconds")
```

```
Original nearly sorted data: [10, 20, 30, 25, 40, 50, 55, 60]
Bubble Sort result: [10, 20, 25, 30, 40, 50, 55, 60]
Insertion Sort result: [10, 20, 25, 30, 40, 50, 55, 60]

Execution Time:
Bubble Sort: 0.00010562 seconds
Insertion Sort: 0.00008821 seconds
```

Task 3: Binary Search vs Linear Search

- **Task**: 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
def linear_search(arr, target):
    """
    Performs a Linear Search on the given list.

    Args:
        arr (list): The list to search through (can be sorted or unsorted).
        target (any): The element to find.

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

    Performance:
        Time Complexity: O(n) — must check each element.
        Space Complexity: O(1) — no extra storage used.
    """
    for i in range(len(arr)):
        if arr[i] == target:  # Compare each element with the target
            return i  # Return the position if found
    return -1  # Return -1 if target not present
def binary_search(arr, target):
    """
    Performs a Binary Search on a sorted list.

    Args:
        arr (list): The list to search through (must be sorted in ascending order).
        target (any): The element to find.

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

    Performance:
        Time Complexity: O(log n) — divides search range by 2 each step.
        Space Complexity: O(1) — in-place iteration.
    """
```

```python
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2  # Find the middle index

        # Check if the target is at mid
        if arr[mid] == target:
            return mid
        # If target is smaller, ignore the right half
        elif arr[mid] > target:
            high = mid - 1
        # If target is larger, ignore the left half
        else:
            low = mid + 1

    return -1  # Element not found
# Test Data
unsorted_data = [34, 7, 23, 32, 5, 62]
sorted_data = sorted(unsorted_data)

target = 23

print(" 🔸 Testing on UNSORTED Data:")
print("Unsorted list:", unsorted_data)
print("Linear Search Result:", linear_search(unsorted_data, target))
# Binary Search will fail on unsorted data
print("Binary Search Result (Invalid on unsorted data):", binary_search(unsorted_data, target))

print("\n 🔹 Testing on SORTED Data:")
print("Sorted list:", sorted_data)
print("Linear Search Result:", linear_search(sorted_data, target))
print("Binary Search Result:", binary_search(sorted_data, target))
```

```
◆ Testing on UNSORTED Data:
Unsorted list: [34, 7, 23, 32, 5, 62]
Linear Search Result: 2
Binary Search Result (Invalid on unsorted data): 2

◆ Testing on SORTED Data:
Sorted list: [5, 7, 23, 32, 34, 62]
Linear Search Result: 2
Binary Search Result: 2
```

- **Task 4:** 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

[1]
0s

```python
import random
import time

# ------------------- QUICK SORT -------------------
def quick_sort(arr):
    """
    Recursive implementation of Quick Sort algorithm.

    Args:
        arr (list): List of elements to be sorted.
    Returns:
        list: A new sorted list.

    Time Complexity:
        Average: O(n log n)
        Worst: O(n^2) when pivot is poorly chosen.
    """
    # Base case
    if len(arr) <= 1:
        return arr

    # Choose pivot
    pivot = arr[len(arr) // 2]

    # Partition step (AI completes logic)
    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]

    # Recursive step
    return quick_sort(left) + middle + quick_sort(right)
```

```python
# -------------------- MERGE SORT --------------------
def merge_sort(arr):
    """
    Recursive implementation of Merge Sort algorithm.

    Args:
        arr (list): List of elements to be sorted.
    Returns:
        list: A new sorted list.

    Time Complexity:
        Always O(n log n)
    """
    # Base case
    if len(arr) <= 1:
        return arr

    # Divide step
    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    # Recursive sort (AI completes logic)
    left_sorted = merge_sort(left_half)
    right_sorted = merge_sort(right_half)

    # Merge step
    return merge(left_sorted, right_sorted)


def merge(left, right):
    """Helper function to merge two sorted lists."""
    merged = []
```

[1]
0s

```python
    # Merge both halves
    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

    # Add any remaining elements
    merged.extend(left[i:])
    merged.extend(right[j:])

    return merged


# -------------------- COMPARISON --------------------
def compare_algorithms():
    """Compare Quick Sort and Merge Sort on different list conditions."""

    # Test lists
    random_list = [random.randint(1, 1000) for _ in range(1000)]
    sorted_list = sorted(random_list)
    reverse_sorted_list = sorted(random_list, reverse=True)

    tests = {
        "Random List": random_list,
        "Sorted List": sorted_list,
        "Reverse Sorted List": reverse_sorted_list
    }

    for name, test_list in tests.items():
        print(f"\n{name}:")
```

```python
        # Quick Sort timing
        start = time.time()
        quick_sort(test_list.copy())
        quick_time = time.time() - start

        # Merge Sort timing
        start = time.time()
        merge_sort(test_list.copy())
        merge_time = time.time() - start

        print(f"Quick Sort Time: {quick_time:.6f} sec")
        print(f"Merge Sort Time: {merge_time:.6f} sec")
        print(f"Faster: {'Quick Sort' if quick_time < merge_time else 'Merge Sort'}")


# ------------------- MAIN -------------------
if __name__ == "__main__":
    compare_algorithms()
```

```
Random List:
Quick Sort Time: 0.001260 sec
Merge Sort Time: 0.001995 sec
Faster: Quick Sort

Sorted List:
Quick Sort Time: 0.001033 sec
Merge Sort Time: 0.001373 sec
Faster: Quick Sort

Reverse Sorted List:
Quick Sort Time: 0.001117 sec
Merge Sort Time: 0.001390 sec
Faster: Quick Sort
```

Task 5: AI-Suggested Algorithm Optimization

- **Task:** 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

[2]
5s

```python
import random
import time

# ------------------- STEP 1: NAIVE DUPLICATE FINDER (O(n²)) -------------------
def find_duplicates_bruteforce(arr):
    """
    Brute-force algorithm to find duplicates in a list.

    Args:
        arr (list): Input list of elements.
    Returns:
        list: List of duplicate elements (may contain repeats).

    Time Complexity: O(n²)
    Space Complexity: O(1)
    """
    duplicates = []
    for i in range(len(arr)):
        for j in range(i + 1, len(arr)):
            if arr[i] == arr[j] and arr[i] not in duplicates:
                duplicates.append(arr[i])
    return duplicates


# ------------------- STEP 2: AI-OPTIMIZED VERSION (O(n)) -------------------
def find_duplicates_optimized(arr):
    """
    Optimized algorithm using a set for O(n) duplicate detection.

    Args:
        arr (list): Input list of elements.
    Returns:
        list: List of unique duplicate elements.
```

[2] 5s

```python
    Time Complexity: O(n)
    Space Complexity: O(n)
    """
    seen = set()
    duplicates = set()

    for item in arr:
        if item in seen:
            duplicates.add(item)
        else:
            seen.add(item)

    return list(duplicates)


# -------------------- STEP 3: PERFORMANCE COMPARISON --------------------
def compare_algorithms():
    """
    Compare execution times of brute-force and optimized algorithms
    on increasing input sizes.
    """
    sizes = [1000, 3000, 5000, 10000]

    for size in sizes:
        print(f"\nInput Size: {size}")
        data = [random.randint(1, size // 2) for _ in range(size)]  # intentional duplicates

        # Brute force timing
        start = time.time()
        find_duplicates_bruteforce(data)
        brute_time = time.time() - start

        # Optimized timing
        start = time.time()
```

```python
        start = time.time()
        find_duplicates_optimized(data)
        opt_time = time.time() - start

        print(f"Brute Force Time: {brute_time:.6f} sec")
        print(f"Optimized Time:   {opt_time:.6f} sec")
        print(f"Speedup: ~{brute_time / opt_time:.2f}x faster\n")


# -------------------- MAIN --------------------
if __name__ == "__main__":
    compare_algorithms()
```

```
Input Size: 1000
Brute Force Time: 0.031524 sec
Optimized Time:   0.000137 sec
Speedup: ~230.35x faster


Input Size: 3000
Brute Force Time: 0.256500 sec
Optimized Time:   0.001193 sec
Speedup: ~215.08x faster


Input Size: 5000
Brute Force Time: 0.720259 sec
Optimized Time:   0.000682 sec
Speedup: ~1055.55x faster


Input Size: 10000
Brute Force Time: 4.015618 sec
Optimized Time:   0.001084 sec
Speedup: ~3704.95x faster
```