Lab 12: Algorithms with AI Assistance – Sorting, Searching, and Optimizing Algorithms

Name: YASHWANTH

Enrollment Number: 2503A51L42

Assignment Number:12.2

Task 1: Bubble Sort with AI Comments

Prompt:
Add inline comments and time complexity analysis to this Bubble Sort implementation.

Code:

assignment 12.2 > task .py > bubble_sort

```python
def bubble_sort(arr):
    """Sort a list in ascending order using Bubble Sort.

    This implementation performs the standard (un-optimized) Bubble Sort:
    - Repeatedly steps through the list
    - Compares adjacent items and swaps them when they are in the wrong order

    Contract:
    - Input: arr (list) — a mutable sequence of comparable items
    - Output: the same list object `arr`, sorted in ascending order (in-place)

        Time complexity:
        - Worst-case: O(n^2) — when the list is in reverse order (every pair must be swapped)
        - Average-case: O(n^2)
        - Best-case: O(n) only if an early-exit optimization is used (array already sorted);
          the implementation below is the standard un-optimized version so its practical
          best-case without the early-exit check is O(n^2) as well.

        Space complexity: O(1) extra space — sorting is done in-place using a few temps.

    Stability: Bubble Sort is a stable sort (equal elements retain their relative order).

    Note: This function returns the same list object for convenience, but it sorts in-place.
    """
    n = len(arr)  # total number of elements in the list

    # Outer loop: we do n passes. After k passes (k starting at 0) the last
    # k elements are guaranteed to be in their final sorted positions because
    # the largest remaining item 'bubbles up' to the end on each pass.
    for i in range(n):
        # Inner loop: walk the list from the start up to the last unsorted
        # element. The upper bound decreases by i because the final i elements
        # are already correctly placed.
        # j goes from 0 to n - i - 2 inclusive so we can safely access arr[j+1].
        for j in range(0, n - i - 1):
            # If the current element is greater than the next element, swap
            # them so the larger value moves one step toward the end.
            # The swap is done in-place and is stable for equal elements
            # (their relative order is preserved because we only swap when
            # the left item is strictly greater than the right one).
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

    # Return the sorted list (same object) to allow usage like: sorted_list = bubble_sort(my_list)
    return arr

# Test cases (simple sanity checks)
# - Should call bubble_sort (the function defined above), not insertion_sort.
print(bubble_sort([1, 5, 2, 4]))  # -> [1, 2, 4, 5]
print(bubble_sort([]))            # -> [] (empty list)
print(bubble_sort([3, 1, 2, 3]))  # -> [1, 2, 3, 3] (stable with duplicates)
```

Output:

```
PS C:\Users\Suhana Rehan\OneDrive\Desktop\AI assistant coding> & "C:/Users/Suhana R
c:/Users/Suhana Rehan/OneDrive/Desktop/AI assistant coding/assignment 12.2/task .py
[1, 2, 4, 5]
[]
[1, 2, 3, 3]
```

Observations:

- Bubble Sort repeatedly swaps adjacent elements until the list is sorted.
- It performs (O(n^2)) comparisons in the worst case, making it inefficient for large datasets.
- AI comments clarified the role of each pass and how early termination can improve performance slightly.

Task 2: Optimizing Bubble Sort → Insertion Sort

Prompt:
Suggest a more efficient sorting algorithm for nearly sorted arrays and explain why.

Code:

```
assignment 12.2 > task 2.py > ...
def insertion_sort(arr):
    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 key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
# Test case
print(insertion_sort([1, 2, 3, 5, 4]))
# Output: [1, 2, 3, 4, 5]
```

Output:

```
PS C:\Users\Suhana Reha
/Users/Suhana Rehan/One
[1, 2, 3, 4, 5]
```

Observations:

- Insertion Sort is faster than Bubble Sort on nearly sorted data due to fewer shifts.
- It has a best-case time complexity of (O(n)) when the array is already sorted.

- AI highlighted that Insertion Sort minimizes unnecessary swaps, making it ideal for incremental sorting.

Task 3: Binary Search vs Linear Search

Prompt:
Generate docstrings and performance notes for Linear and Binary Search, and explain when Binary Search is preferable.

Code:

assignment 12.2 > task 3.py > ...

```python
def linear_search(arr, target):
    """Searches for target in arr using linear scan. Time complexity: O(n)"""
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1

def binary_search(arr, target):
    """Searches for target in sorted arr using binary search. Time complexity: 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
        else:
            high = mid - 1
    return -1
# Test cases
print("Linear Search:")
print(linear_search([3, 5, 1, 9], 5)    )   # Output: 1
print("Binary Search:")
print(binary_search([1, 3, 5, 9], 5)  )     # Output: 2
```

Output:

```
Linear Search:
1
Binary Search:
2
```

Observations:

- Linear Search works on any list but is slower for large datasets ((O(n))).
- Binary Search requires sorted input and performs in (O(\log n)) time.
- AI emphasized Binary Search's efficiency for large, sorted datasets and its limitations on unsorted data.

Task 4: Quick Sort and Merge Sort Comparison

Prompt:
Complete recursive Quick Sort and Merge Sort functions with docstrings and explain their time complexities.

Code:

assignment 12.2 > task 4.py > ...

```python
def quick_sort(arr):
    """Quick Sort using recursion. Avg: O(n log n), Worst: O(n^2)"""
    if len(arr) <= 1:
        return arr
    pivot = arr[0]
    left = [x for x in arr[1:] if x < pivot]
    right = [x for x in arr[1:] if x >= pivot]
    return quick_sort(left) + [pivot] + quick_sort(right)

def merge_sort(arr):
    """Merge Sort using recursion. Always O(n log n)"""
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    # Merge step
    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:])
    result.extend(right[j:])
    return result
#test cases
print("Quick Sort:")
print(quick_sort([4, 2, 7, 1]))   # Output: [1, 2, 4, 7]
print("Merge Sort:")
print(merge_sort([4, 2, 7, 1]))   # Output: [1, 2, 4, 7]
```

Output:

```
/Users/Suhana Re
Quick Sort:
[1, 2, 4, 7]
Merge Sort:
[1, 2, 4, 7]
```

Observations:

- Quick Sort is faster on average but suffers in worst-case scenarios (($O(n^2)$)).
- Merge Sort guarantees ($O(n \log n)$) performance regardless of input order.
- AI explained that Merge Sort is stable and better for linked lists, while Quick Sort is faster in-place.

Task 5: AI-Suggested Algorithm Optimization

Prompt:
Optimize this naive duplicate-finder algorithm and explain how the time complexity improves.

Code:

```
assignment 12.2 > task 5.py > ...
def find_duplicates_optimized(arr):
    seen = set()
    duplicates = set()
    for item in arr:
        if item in seen:
            duplicates.add(item)
        else:
            seen.add(item)
    return list(duplicates)
# Test case
print(find_duplicates_optimized([1, 2, 3, 2, 4, 3, 5]))  # Output: [2, 3]
```

Output:

```
PS C:\Users\Sun
/Users/Suhana R
[2, 3]
```

Observations:

- The brute-force method checks all pairs, resulting in ($O(n^2)$) time.
- AI replaced it with a set-based approach, reducing complexity to ($O(n)$).
- Execution time dropped significantly on large inputs, validating the AI's optimization strategy.