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

Name: *Chunchu Siri*

Hall Ticket No.: 2303A51281

Batch No.: 5

Task 1: Bubble Sort for Ranking Exam Scores

Prompt Given to AI:
Implement Bubble Sort in Python with inline comments explaining comparisons, swaps, and passes. Add early termination optimization and explain best, average, and worst-case time complexity.

Implementation:

```
def bubble_sort(scores):
    n = len(scores)
    for i in range(n):
        swapped = False  # Track if any swap happens in this pass

        for j in range(0, n - i - 1):
            # Compare adjacent elements
            if scores[j] > scores[j + 1]:
                # Swap if elements are out of order
                scores[j], scores[j + 1] = scores[j + 1], scores[j]
                swapped = True

        # If no swaps occurred, the list is already sorted
        if not swapped:
            break
    return scores

scores = [78, 45, 89, 32, 67]
print("Sorted Scores:", bubble_sort(scores))
```

Sample Output:
Sorted Scores: [32, 45, 67, 78, 89]

```
PS C:\Users\Siri Chunchu\AIAssisted-1281> python Assignment11.3.py
Sorted Scores: [32, 45, 67, 78, 89]
PS C:\Users\Siri Chunchu\AIAssisted-1281>
```

Time Complexity Analysis:
Best Case: O(n) – When the list is already sorted (early termination works).

Average Case: $O(n^2)$ – Comparisons required for most elements.

Worst Case: $O(n^2)$ – When the list is reverse sorted.

Observation:
Bubble Sort is suitable for small datasets but inefficient for large inputs.

Task 2: Improving Sorting for Nearly Sorted Attendance Records

Prompt Given to AI:
Review the problem of nearly sorted data and suggest a better sorting algorithm than Bubble Sort. Generate Insertion Sort code and explain why it performs better.

Insertion Sort Implementation:

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1

        # Shift elements greater than key
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1

        arr[j + 1] = key
    return arr

attendance = [1, 2, 3, 5, 4, 6, 7]
print("Sorted Attendance:", insertion_sort(attendance))
```

Sample Output:
Sorted Attendance: [1, 2, 3, 4, 5, 6, 7]

```
PS C:\Users\Siri Chunchu\AIAssisted-1281> python Assignment11.3.py
Sorted Attendance: [1, 2, 3, 4, 5, 6, 7]
PS C:\Users\Siri Chunchu\AIAssisted-1281>
```

Comparison Explanation:
Insertion Sort performs better on nearly sorted data because it only shifts a few elements. In best case (already sorted), its time complexity becomes O(n). Bubble Sort still performs multiple unnecessary comparisons.

Observation:
For partially sorted datasets, Insertion Sort is more efficient than Bubble Sort.

Task 3: Searching Student Records

Prompt Given to AI:
Implement Linear Search and Binary Search with proper docstrings. Explain when Binary Search is applicable and compare performance.

Implementation:

```
def linear_search(arr, target):
    """Search for target in unsorted list.
    Parameters: arr (list), target (int)
    Returns: index if found, else -1"""
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1

def binary_search(arr, target):
    """Search for target in sorted list using Binary Search.
    Parameters: arr (sorted list), target (int)
    Returns: index if found, else -1"""
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
```

```
        left = mid + 1
    else:
        right = mid - 1
return -1
```

Time Complexity:
Linear Search: O(n)

Binary Search: O(log n) – Only applicable when data is sorted.

Observation:
Binary Search is significantly faster for large sorted datasets. For unsorted data, Linear Search is required unless sorting is performed first.

Task 4: Quick Sort vs Merge Sort

Prompt Given to AI:
Complete recursive Quick Sort and Merge Sort implementations, add docstrings, and explain recursion and complexity differences.

Implementation:

```
def quick_sort(arr):
    """Recursive Quick Sort implementation"""
    if len(arr) <= 1:
        return arr
    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)

def merge_sort(arr):
    """Recursive Merge Sort implementation"""
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
```

```
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
```

Assignment11.3.py > ...

```python
def linear_search(arr, target):
    """Search for target in unsorted list.
    Parameters: arr (list), target (int)
    Returns: index if found, else -1"""
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1

def binary_search(arr, target):
    """Search for target in sorted list using Binary Search.
    Parameters: arr (sorted list), target (int)
    Returns: index if found, else -1"""
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Complexity Comparison:

Quick Sort: Best/Average O(n log n), Worst $O(n^2)$

Merge Sort: O(n log n) in all cases, but requires extra space O(n)

Observation:
Quick Sort is generally faster in practice but may degrade on already sorted data. Merge Sort guarantees stable O(n log n) performance.

Task 5: Optimizing Duplicate Detection

Prompt Given to AI:
Analyze naive duplicate detection using nested loops. Suggest optimized solution using sets and compare time complexity.

Implementation:

```
# Brute Force O(n^2)
def has_duplicates_bruteforce(arr):
    for i in range(len(arr)):
        for j in range(i + 1, len(arr)):
            if arr[i] == arr[j]:
                return True
    return False

# Optimized O(n)
def has_duplicates_optimized(arr):
    seen = set()
    for item in arr:
        if item in seen:
            return True
        seen.add(item)

 return False
```

```python
# Brute Force O(n^2)
def has_duplicates_bruteforce(arr):
    for i in range(len(arr)):
        for j in range(i + 1, len(arr)):
            if arr[i] == arr[j]:
                return True
    return False

# Optimized O(n)
def has_duplicates_optimized(arr):
    seen = set()
    for item in arr:
        if item in seen:
            return True
        seen.add(item)
    return False

# Test cases
if __name__ == "__main__":
    test_array = [1, 2, 3, 4, 5, 2]
    print("Brute Force:", has_duplicates_bruteforce(test_array))  # Output: True
    print("Optimized:", has_duplicates_optimized(test_array))      # Output: True

    test_array_no_duplicates = [1, 2, 3, 4, 5]
    print("Brute Force:", has_duplicates_bruteforce(test_array_no_duplicates))  # Output: False
    print("Optimized:", has_duplicates_optimized(test_array_no_duplicates))      # Output: False
```

```
PS C:\Users\Siri Chunchu\AIAssisted-1281> python Assignment11.3.py
Brute Force: True
Optimized: True
Brute Force: False
Optimized: False
PS C:\Users\Siri Chunchu\AIAssisted-1281>
```

Complexity Comparison:
Brute Force: $O(n^2)$ – compares every pair.

Optimized: $O(n)$ – uses hash set for constant-time lookup.

Observation:
Using appropriate data structures like sets dramatically improves performance for large datasets. This demonstrates the importance of algorithm optimization.