AI-ASSIST CODING

LAB-12.4

K.SAI RAHUL

2303A52392

BATCH-43

Task 1: Bubble Sort for Ranking Exam Scores

PROMPT:

Write a Python program to implement Bubble Sort to sort student scores. Add inline comments explaining comparisons, swaps, and iteration passes. Include early termination when the list is already sorted.Also provide time complexity and sample input/output.

CODE:

```python
scores = [78, 45, 90, 62, 88]
# Length of the list
n = len(scores)
# Outer loop for passes
for i in range(n):
    swapped = False  # Flag to check if any swap happened in this pass
    # Inner loop for comparing adjacent elements
    for j in range(0, n - i - 1):
        # Comparison of adjacent elements
        if scores[j] > scores[j + 1]:
            # Swap if elements are in wrong order
            scores[j], scores[j + 1] = scores[j + 1], scores[j]
            swapped = True  # Mark that swap happened
    # Early termination: if no swaps, list is already sorted
    if not swapped:
        print("Early termination: List is already sorted")
        break
# Output sorted scores
print("Sorted Scores:", scores)
```

OUTPUT:

```
PS C:\Users\rahul\OneDrive\Desktop\AI ASSIST> & C:/Users/rahul/AppData/Local/Microsoft/WindowsApps/python3.1
3.exe "c:/Users/rahul/OneDrive/Desktop/AI ASSIST/lab-9.4.py"
Early termination: List is already sorted
Sorted Scores: [45, 62, 78, 88, 90]
```

OBSERVATION:

Bubble Sort works by repeatedly comparing adjacent elements in a list and swapping them whenever they are in the wrong order. With each pass of the

algorithm, the largest element among the unsorted portion moves to its correct position at the end of the list. The use of early termination makes the algorithm faster when the list becomes sorted before completing all passes, as it stops further unnecessary comparisons. This method is simple and efficient for sorting small datasets, such as classroom student scores.

Task 2: Improving Sorting for Nearly Sorted Attendance Records

PROMPT:

Given that attendance roll numbers are nearly sorted, suggest a suitable sorting algorithm. Generate Bubble Sort and Insertion Sort implementations in Python. Explain why Insertion Sort performs better for nearly sorted data.Compare execution behaviour.

CODE:

```python
# Bubble Sort for attendance roll numbers
roll = [101, 102, 105, 104, 106]  # Nearly sorted
n = len(roll)
for i in range(n):
    swapped = False
    for j in range(0, n - i - 1):
        if roll[j] > roll[j + 1]:
            roll[j], roll[j + 1] = roll[j + 1], roll[j]
            swapped = True
    if not swapped:
        break
print("Bubble Sorted Roll Numbers:", roll)

# Insertion Sort for attendance roll numbers
roll = [101, 102, 105, 104, 106]  # Nearly sorted
for i in range(1, len(roll)):
    key = roll[i]      # Element to be inserted
    j = i - 1
    while j >= 0 and roll[j] > key:
        roll[j + 1] = roll[j]
        j -= 1
    roll[j + 1] = key  # Insert at correct position
print("Insertion Sorted Roll Numbers:", roll)
```

OUTPUT:

```
3.exe "c:/Users/rahul/OneDrive/Desktop/AI ASSIST/lab-9.4.py"
Bubble Sorted Roll Numbers: [101, 102, 104, 105, 106]
Insertion Sorted Roll Numbers: [101, 102, 104, 105, 106]
```

OBSERVATION:

Insertion Sort works efficiently for nearly sorted data because it inserts each element into its correct position in the already sorted portion of the list. When only a few elements are misplaced, it performs fewer comparisons and shifts. In contrast, Bubble Sort still compares many adjacent elements even if the list is almost sorted. Therefore, Insertion Sort reduces unnecessary operations and improves performance for partially sorted datasets.

Task 3: Searching Student Records in a Database

PROMPT:

Write Python programs for Linear Search and Binary Search to find student roll numbers. Add docstrings explaining parameters and return values. Explain when Binary Search is applicable and compare performance.

CODE:

```python
#Linear search
def linear_search(roll_list, target):
    for i in range(len(roll_list)):
        if roll_list[i] == target:
            return i
    return -1
# Example
roll = [105, 101, 109, 102, 108]
result = linear_search(roll, 102)
if result != -1:
    print("Student found at index:", result)
else:
    print("Student not found")

#Binary search
def binary_search(roll_list, target):
    low = 0
    high = len(roll_list) - 1
    while low <= high:
        mid = (low + high) // 2
        if roll_list[mid] == target:
            return mid
        elif roll_list[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
# Example
sorted_roll = [101, 102, 105, 108, 109]
result = binary_search(sorted_roll, 102)

if result != -1:
    print("Student found at index:", result)
else:
    print("Student not found")
```

OUTPUT:

```
PS C:\Users\rahul\OneDrive\Desktop\AI ASSIST> & C:/Users/rahul/AppData/Local/Microsoft/WindowsApps/python3.1
3.exe "c:/Users/rahul/OneDrive/Desktop/AI ASSIST/lab-9.4.py"
Student found at index: 3
Student found at index: 1
```

OBSERVATION:

Linear Search works for both sorted and unsorted lists but becomes slow when the dataset grows. Binary Search is much faster and efficient but requires the list to be sorted beforehand. Therefore, Binary Search is preferred for large student databases, while Linear Search is suitable for small or unsorted attendance or temporary records. Advanced searching techniques like **Hashing** can further improve performance in large systems.

Task 4: Choosing Between Quick Sort and Merge Sort for Data Processing

PROMPT:

Complete recursive implementations of Quick Sort and Merge Sort in Python. Add meaningful docstrings. Explain how recursion works in both algorithms. Compare performance for random, sorted, and reverse-sorted data.

CODE:

```python
#Quick Sort Algorithm (Recursive)
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[0]
    # Partitioning step
    left = [x for x in arr[1:] if x <= pivot]
    right = [x for x in arr[1:] if x > pivot]
    # Recursive calls on left and right partitions
    return quick_sort(left) + [pivot] + quick_sort(right)
data = [50, 20, 60, 10, 40, 30]
print("Quick Sort Result:", quick_sort(data))

#Merge Sort Algorithm (Recursive)
def merge_sort(arr):
    # Base case
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])
    return merge(left_half, right_half)
def merge(left, right):
    result = []
    i = j = 0
    # Compare elements from both halves
    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
    # Append remaining elements
    result.extend(left[i:])
    result.extend(right[j:])
    return result
# Testing
data = [50, 20, 60, 10, 40, 30]
print("Merge Sort Result:", merge_sort(data))
```

OUTPUT:

```
PS C:\Users\rahul\OneDrive\Desktop\AI ASSIST> & C:/Users/rahul/AppData/Local/Microsoft/WindowsApps/python3.13.exe "c:/Users/rahul/OneDrive/Desktop/AI ASSIST/lab-9.4.py"
Quick Sort Result: [10, 20, 30, 40, 50, 60]
Merge Sort Result: [10, 20, 30, 40, 50, 60]
```

OBSERVATION:

Quick Sort performs very fast on random datasets due to its efficient divide-and-conquer approach. However, it may degrade to **O(n^2)** time complexity for already sorted or reverse-sorted data if the pivot selection is poor. On the other hand, **Merge Sort** provides consistent performance for all types of input because its time complexity remains **O(n log n)** in the best, average, and worst cases. It is also more stable and predictable for handling large datasets. In comparison, **Quick Sort** is more memory-efficient since it requires less additional space than Merge Sort, making it suitable when memory usage is a concern.

Task 5: Optimizing a Duplicate Detection Algorithm

PROMPT:

Write a brute-force algorithm to detect duplicate user IDs using nested loops. Analyse its time complexity. Suggest and implement an optimized approach using sets or dictionaries. Compare performance for large datasets.

CODE:

```python
# Naive approach using nested loops
def find_duplicates_bruteforce(user_ids):
    duplicates = []
    for i in range(len(user_ids)):
        for j in range(i + 1, len(user_ids)):
            if user_ids[i] == user_ids[j]:
                duplicates.append(user_ids[i])
    return duplicates
# Example
data = [101, 205, 309, 101, 450, 205]
print("Duplicates (Brute-force):", find_duplicates_bruteforce(data))

# Optimized approach using a set
def find_duplicates_optimized(user_ids):
    seen = set()
    duplicates = set()
    for user in user_ids:
        if user in seen:
            duplicates.add(user)
        else:
            seen.add(user)
    return list(duplicates)
# Example
data = [101, 205, 309, 101, 450, 205]
print("Duplicates (Optimized):", find_duplicates_optimized(data))
```

OUTPUT:

```
PS C:\Users\rahul\OneDrive\Desktop\AI ASSIST> & C:/Users/rahul/AppData/Local/Microsoft/WindowsApps/python3.1
3.exe "c:/Users/rahul/OneDrive/Desktop/AI ASSIST/lab-9.4.py"
Duplicates (Brute-force): [101, 205]
Duplicates (Optimized): [205, 101]
```

OBSERVATION:

The brute-force algorithm is simple but inefficient for large datasets due to its quadratic time complexity. The optimized approach using hashing improves performance by reducing the time complexity to linear time. This makes it suitable for detecting duplicates in large-scale databases and real-world applications where speed and efficiency are important.