AI Assistant Coding

Name: CH. Arush Nandhan

HTNO: 2303A51880

Batch-14

Date: 26-02-2026

Assignment - 12.4

solution:

Task #1:

Prompt:

Implement Bubble Sort in Python to sort a list of student exam scores.
Include inline comments explaining comparisons, swaps, iteration passes, and early termination condition. Provide time complexity analysis.

Code:

```
# Bubble Sort for sorting student exam scores


def bubble_sort(scores):

   n = len(scores)


   # Outer loop for passes

   for i in range(n):

       swapped = False  # To check if any swapping occurs
```

```python
        # Inner loop for comparison
        for j in range(0, n - i - 1):

            # Compare adjacent elements
            if scores[j] > scores[j + 1]:

                # Swap if they are in wrong order
                scores[j], scores[j + 1] = scores[j + 1],
scores[j]
                swapped = True

        # Early termination if no swap happens
        if not swapped:
            break

    return scores


# -------- INPUT --------
scores = [78, 45, 90, 62, 84]

# -------- OUTPUT --------
print("Original Scores:", scores)
print("Sorted Scores:", bubble_sort(scores))
```

Output:

```
Original Scores: [78, 45, 90, 62, 84]
Sorted Scores: [45, 62, 78, 84, 90]
```

Explanation:

Bubble Sort compares adjacent elements and swaps them if they are in the wrong order.
After each pass, the largest element moves to the end.

The algorithm stops early if no swaps occur (early termination).

Time Complexity:

- Best Case: $O(n)$ (Already sorted)
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$ (Reverse sorted)

Task #2:

Prompt:

Start with Bubble Sort for nearly sorted roll numbers.

Suggest a better algorithm and implement Insertion Sort.

Explain why it performs better for nearly sorted data.

Code:

Bubble sort

```
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
    return arr



# INPUT
roll_numbers = [101, 102, 103, 105, 104, 106]


print("Bubble Sort Output:", bubble_sort(roll_numbers.copy()))
```

Insertion sort

```
def insertion_sort(arr):

    # Start from second element
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1

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

        arr[j + 1] = key

    return arr


# INPUT
roll_numbers = [101, 102, 103, 105, 104, 106]

print("Insertion Sort Output:",
insertion_sort(roll_numbers.copy()))
```

Output:

```
Bubble Sort Output: [101, 102, 103, 104, 105, 106]
Insertion Sort Output: [101, 102, 103, 104, 105, 106]
```

Explanation:

The list is nearly sorted, with only 104 misplaced.

Insertion Sort is better because:

- It only shifts the misplaced element.
- It performs fewer comparisons.
- It is efficient for nearly sorted data.

Time Complexity:

- Best Case: O(n)
- Worst Case: $O(n^2)$
- Nearly Sorted Case: Close to O(n)

Task #3:

Prompt:

Implement Linear Search for unsorted student roll numbers and Binary Search for sorted data. Add explanation and compare performance.

Code:

Linear Search

```
def linear_search(data, target):
    for i in range(len(data)):
        if data[i] == target:
            return i
    return -1



# INPUT
students = [105, 102, 108, 101, 104]
target = 101

result = linear_search(students, target)

print("Linear Search Result:", result)
```

Binary Search

```
def binary_search(data, target):
    low = 0
    high = len(data) - 1
```

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

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

    return -1


# INPUT
sorted_students = [101, 102, 104, 105, 108]
target = 105

result = binary_search(sorted_students, target)

print("Binary Search Result:", result)
```

Output:

```
Linear Search Result: 3
Binary Search Result: 3
```

Explanation:

Linear Search checks every element one by one.
Binary Search divides the list in half repeatedly.

Time Complexity:

- Linear Search: O(n)
- Binary Search: O(log n)

Binary Search is faster but requires sorted data.

Task #4:

Prompt:

Complete recursive Quick Sort and Merge Sort implementations. Add explanation and compare complexities.

Code:

Quick Sort

```python
def quick_sort(arr):
   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)


# INPUT
data = [50, 23, 9, 18, 61, 32]
print("Quick Sort Output:", quick_sort(data))
```

Merge Sort

```python
def merge_sort(arr):
   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


# INPUT
```

```python
data = [50, 23, 9, 18, 61, 32]

print("Merge Sort Output:", merge_sort(data))
```

Output:

```
Quick Sort Output: [9, 18, 23, 32, 50, 61]
Merge Sort Output: [9, 18, 23, 32, 50, 61]
```

Explanation:

Both use divide-and-conquer strategy.

Quick Sort:

- Selects pivot
- Divides into left and right
- Recursively sorts

Merge Sort:

- Divides into halves
- Recursively sorts
- Merges sorted halves

Time Complexity:

Algorithm	Best	Average	Worst
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$
Merge Sort	O(n log n)	O(n log n)	O(n log n)

Merge Sort guarantees performance.

Task #5:

Prompt:

Write a naive duplicate detection algorithm using nested loops.

Then optimize it using sets. Compare performance.

Code:

Brute Force

```
def find_duplicates_brute(data):
   duplicates = []
   for i in range(len(data)):
       for j in range(i + 1, len(data)):
           if data[i] == data[j] and data[i] not in
duplicates:
               duplicates.append(data[i])
   return duplicates



# INPUT
user_ids = [101, 203, 101, 405, 203, 506]
print("Brute Force Duplicates:",
find_duplicates_brute(user_ids))
```

Optimized Version

```
def find_duplicates_optimized(data):
   seen = set()
```

```python
    duplicates = set()

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

    return list(duplicates)



# INPUT
user_ids = [101, 203, 101, 405, 203, 506]
print("Optimized Duplicates:",
find_duplicates_optimized(user_ids))
```

Output:

```
Brute Force Duplicates: [101, 203]
Optimized Duplicates: [101, 203]
```

Explanation:

Brute Force compares every pair → $O(n^2)$.
Optimized method uses set for constant-time lookup → $O(n)$.

For large datasets, optimized version is much faster.