AI – ASSISTED CODING
ASSIGNMENT – 12.4
2403A52091

Task 1: Implementing Bubble Sort with AI comments.

Code :

```python
# Bubble Sort implementation in Python

def bubble_sort(arr):
    n = len(arr)
    # Outer loop for each pass through the array
    for i in range(n):
        # Track if any swaps happen in this pass
        swapped = False
        # Inner loop for comparing adjacent elements
        for j in range(0, n - i - 1):
            # Compare current element with the next
            if arr[j] > arr[j + 1]:
                # Swap if elements are in wrong order
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True  # Mark that a swap occurred
        # If no swaps happened, array is sorted; terminate early
        if not swapped:
            break

# Example usage:
arr = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(arr)
print("Sorted array:", arr)

# Time Complexity Analysis:
# Best Case: O(n) when the array is already sorted (due to early termination).
# Worst and Average Case: O(n^2) because of nested loops.
# Space Complexity: O(1) since sorting is done in-place.
```

Output :

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS   Filter   Code

[Running] python -u "c:\Users\Laxmiprasanna\OneDrive\Desktop\AI asissted coding\# Bubble Sort implementation in Python.py"
Sorted array: [11, 12, 22, 25, 34, 64, 90]

[Done] exited with code=0 in 0.185 seconds
```

NAME : N.LAXMI PRASANNA
BATCH : 04
ENROLL : 2403A52091

Task 2: Optimizing Bubble Sort → Insertion Sort

Code :

```python
# Bubble Sort implementation
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

# Insertion Sort implementation (more efficient for nearly sorted arrays)
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        # Move elements greater than key one position ahead
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

# Example: Nearly sorted input
arr1 = [1, 2, 3, 5, 4, 6, 7]
arr2 = arr1.copy()

bubble_sort(arr1)
insertion_sort(arr2)

print("Bubble Sort result:", arr1)
print("Insertion Sort result:", arr2)

# AI Explanation:
# Insertion Sort is more efficient for nearly sorted arrays because it only shifts out-of-place elements,
# resulting in fewer comparisons and swaps. Its best-case time complexity is O(n) when the array is almost sorted,
# while Bubble Sort still performs unnecessary passes and comparisons, leading to O(n^2) time even for nearly sorted data.
```

Output :

```
[Running] python -u "c:\Users\Laxmiprasanna\OneDrive\Desktop\AI asissted coding\tempCodeRunnerFile.python"
Bubble Sort result: [1, 2, 3, 4, 5, 6, 7]
Insertion Sort result: [1, 2, 3, 4, 5, 6, 7]

[Done] exited with code=0 in 0.258 seconds
```

NAME : N.LAXMI PRASANNA
BATCH : 04
ENROLL : 2403A52091

Task 3: Binary Search vs Linear Search

Code :

```python
def linear_search(arr, target):
    """
    Performs a linear search for the target in the array.
    Args:
        arr (list): The list to search.
        target: The value to find.
    Returns:
        int: Index of target if found, else -1.
    Performance:
        Time Complexity: O(n)
        Works on both sorted and unsorted arrays.
    """
    for i, value in enumerate(arr):
        if value == target:
            return i
    return -1

def binary_search(arr, target):
    """
    Performs a binary search for the target in a sorted array.
    Args:
        arr (list): The sorted list to search.
        target: The value to find.
    Returns:
        int: Index of target if found, else -1.
    Performance:
        Time Complexity: O(log n)
        Only works on sorted arrays.
    """
    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

# Test data
unsorted_arr = [7, 2, 9, 4, 5]
sorted_arr = sorted(unsorted_arr)
target = 4

# Testing
print("Linear Search (unsorted):", linear_search(unsorted_arr, target))
print("Linear Search (sorted):", linear_search(sorted_arr, target))
print("Binary Search (sorted):", binary_search(sorted_arr, target))
print("Binary Search (unsorted):", binary_search(unsorted_arr, target))  # May not work correctly

# AI Explanation:
# Binary Search is preferable when searching in large, sorted arrays because it is much faster (O(log n)) than Linear Search (O(n)).
# Linear Search can be used on any array, sorted or unsorted, but is slower for large datasets.

# Student Observation Table:
# | Search Type     | Array Type   | Time Complexity | Works?      |
# |-----------------|--------------|-----------------|-------------|
# | Linear Search   | Unsorted     | O(n)            | Yes         |
# | Linear Search   | Sorted       | O(n)            | Yes         |
# | Binary Search   | Unsorted     | O(log n)        | No (incorrect results) |
# | Binary Search   | Sorted       | O(log n)        | Yes         |
```

NAME : N.LAXMI PRASANNA
BATCH : 04
ENROLL : 2403A52091

Output :

```
[Running] python -u "c:\Users\Laxmiprasanna\OneDrive\Desktop\AI asissted coding\tempCodeRunnerFile.python"
Linear Search (unsorted): 3
Linear Search (sorted): 1
Binary Search (sorted): 1
Binary Search (unsorted): -1

[Done] exited with code=0 in 0.159 seconds
```

Task 4: Quick Sort and Merge Sort Comparison

Code :

```python
def quick_sort(arr):
    """
    Recursively sorts an array using the Quick Sort algorithm.
    Args:
        arr (list): The list to sort.
    Returns:
        list: A new sorted list.
    Performance:
        Average/Best Case: O(n log n)
        Worst Case: O(n^2) (when pivot choices are poor, e.g., sorted/reverse-sorted lists)
    """
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]  # Choose middle element as pivot
    left = [x for x in arr if x < pivot]      # Elements less than pivot
    middle = [x for x in arr if x == pivot]   # Elements equal to pivot
    right = [x for x in arr if x > pivot]     # Elements greater than pivot
    # Recursively sort left and right partitions
    return quick_sort(left) + middle + quick_sort(right)

def merge_sort(arr):
    """
    Recursively sorts an array using the Merge Sort algorithm.
    Args:
        arr (list): The list to sort.
    Returns:
        list: A new sorted list.
    Performance:
        Best/Average/Worst Case: O(n log n)
        Merge Sort always divides and merges, so time complexity is stable.
    """
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])   # Recursively sort left half
```

NAME : N.LAXMI PRASANNA
BATCH : 04
ENROLL : 2403A52091

```python
    right = merge_sort(arr[mid:])  # Recursively sort right half
    # Merge the sorted halves
    merged = []
    i = j = 0
    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
    merged.extend(left[i:])
    merged.extend(right[j:])
    return merged

# Test cases
import random
random_list = random.sample(range(1, 100), 10)
sorted_list = sorted(random_list)
reverse_sorted_list = sorted(random_list, reverse=True)

print("Quick Sort (random):", quick_sort(random_list))
print("Merge Sort (random):", merge_sort(random_list))
print("Quick Sort (sorted):", quick_sort(sorted_list))
print("Merge Sort (sorted):", merge_sort(sorted_list))
print("Quick Sort (reverse sorted):", quick_sort(reverse_sorted_list))
print("Merge Sort (reverse sorted):", merge_sort(reverse_sorted_list))

# AI Complexity Explanation:
# Quick Sort:
#   - Average/Best Case: O(n log n) (good pivot splits)
#   - Worst Case: O(n^2) (poor pivots, e.g., already sorted or reverse-sorted lists)
# Merge Sort:
#   - Best/Average/Worst Case: O(n log n) (always divides evenly and merges)
#   - Merge Sort is stable and predictable, while Quick Sort is faster on average but can degrade with bad pivots.
```

Output :

```
[Running] python -u "c:\Users\Laxmiprasanna\OneDrive\Desktop\AI asissted coding\tempCodeRunnerFile.python"
Quick Sort (random): [2, 36, 43, 65, 70, 79, 80, 81, 94, 99]
Merge Sort (random): [2, 36, 43, 65, 70, 79, 80, 81, 94, 99]
Quick Sort (sorted): [2, 36, 43, 65, 70, 79, 80, 81, 94, 99]
Merge Sort (sorted): [2, 36, 43, 65, 70, 79, 80, 81, 94, 99]
Quick Sort (reverse sorted): [2, 36, 43, 65, 70, 79, 80, 81, 94, 99]
Merge Sort (reverse sorted): [2, 36, 43, 65, 70, 79, 80, 81, 94, 99]

[Done] exited with code=0 in 0.16 seconds
```

NAME : N.LAXMI PRASANNA
BATCH : 04
ENROLL : 2403A52091

Task 5: AI-Suggested Algorithm Optimization

Code :

```python
# Brute force duplicate finder (O(n^2))
def find_duplicates_brute_force(arr):
    """
    Finds duplicates in a list using a brute force approach.
    Time Complexity: O(n^2)
    """
    duplicates = set()
    n = len(arr)
    for i in range(n):
        for j in range(i + 1, n):
            if arr[i] == arr[j]:
                duplicates.add(arr[i])
    return list(duplicates)

# Optimized duplicate finder using a set (O(n))
def find_duplicates_optimized(arr):
    """
    Finds duplicates in a list using a set for tracking.
    Time Complexity: O(n)
    """
    seen = set()
    duplicates = set()
    for item in arr:
```

NAME : N.LAXMI PRASANNA
BATCH : 04
ENROLL : 2403A52091

```python
            if item in seen:
                duplicates.add(item)
            else:
                seen.add(item)
        return list(duplicates)

# Example usage and timing comparison
import random
import time

large_list = [random.randint(1, 10000) for _ in range(10000)]

start = time.time()
brute_result = find_duplicates_brute_force(large_list)
print("Brute force time:", time.time() - start)

start = time.time()
opt_result = find_duplicates_optimized(large_list)
print("Optimized time:", time.time() - start)

# AI Explanation:
# The brute force method checks every pair of elements, resulting in O(n^2) time.
# The optimized method uses a set to track seen elements, reducing the time complexity to O(n).
# For large lists, the optimized version is dramatically faster and more scalable.
```

Output :

```
[Running] python -u "c:\Users\Laxmiprasanna\OneDrive\Desktop\AI asissted coding\tempCodeRunnerFile.python"
Brute force time: 2.676950693130493
Optimized time: 0.0026793479919433594

[Done] exited with code=0 in 2.868 seconds
```

NAME : N.LAXMI PRASANNA
BATCH : 04
ENROLL : 2403A52091