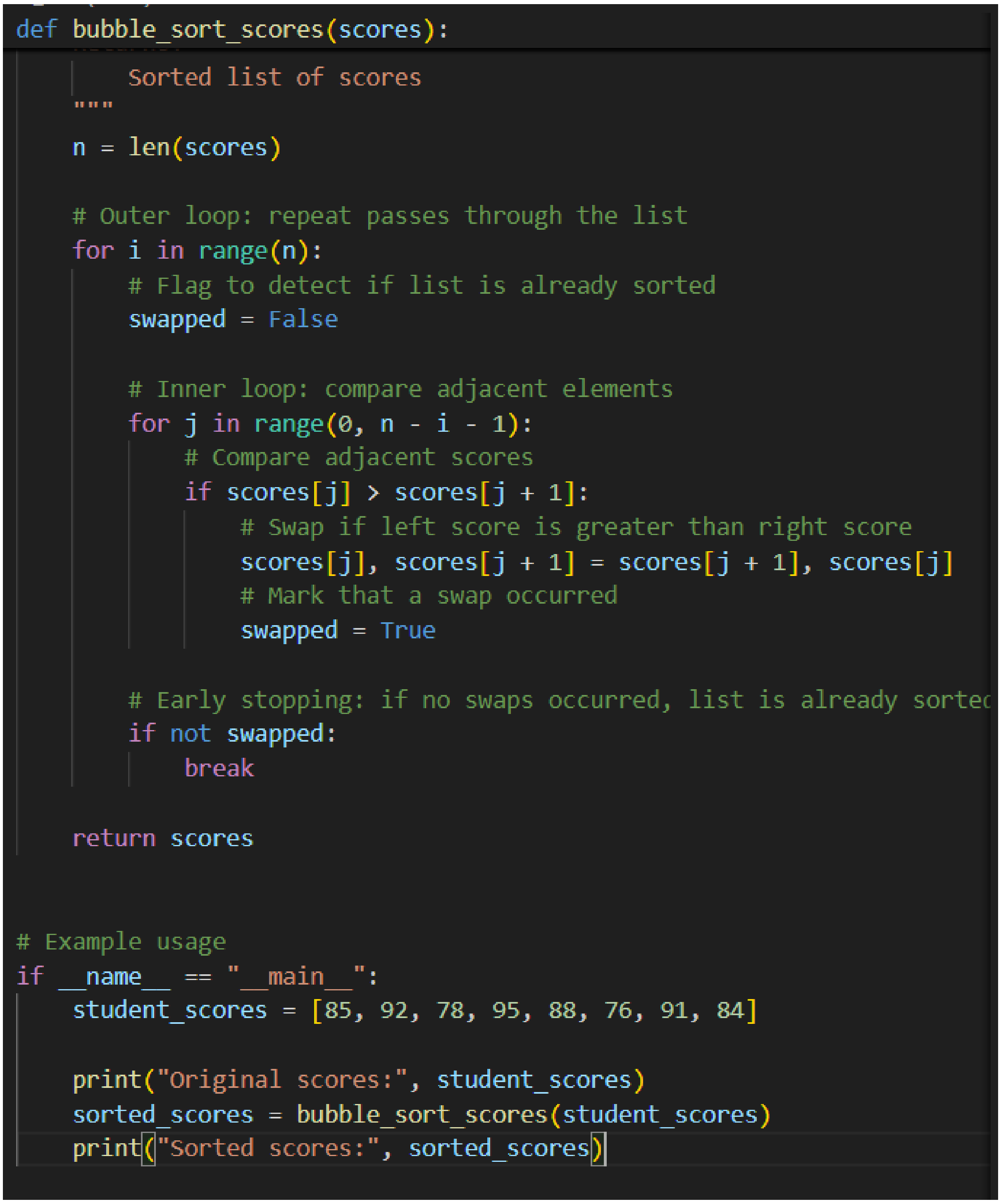
ASSIGNMENT-12.4

Task_1:

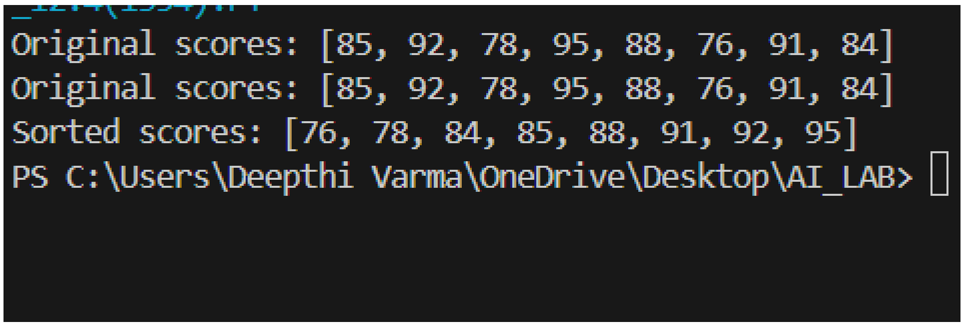
Prompt: Write a simple Python program to sort student scores using Bubble Sort with clear swaps, and early stopping when already sorted.

Code:



```python
def bubble_sort_scores(scores):
        Sorted list of scores
    """
    n = len(scores)

    # Outer loop: repeat passes through the list
    for i in range(n):
        # Flag to detect if list is already sorted
        swapped = False

        # Inner loop: compare adjacent elements
        for j in range(0, n - i - 1):
            # Compare adjacent scores
            if scores[j] > scores[j + 1]:
                # Swap if left score is greater than right score
                scores[j], scores[j + 1] = scores[j + 1], scores[j]
                # Mark that a swap occurred
                swapped = True

        # Early stopping: if no swaps occurred, list is already sorte
        if not swapped:
            break

    return scores


# Example usage
if __name__ == "__main__":
    student_scores = [85, 92, 78, 95, 88, 76, 91, 84]

    print("Original scores:", student_scores)
    sorted_scores = bubble_sort_scores(student_scores)
    print("Sorted scores:", sorted_scores)
```

Output:



```
Original scores: [85, 92, 78, 95, 88, 76, 91, 84]
Original scores: [85, 92, 78, 95, 88, 76, 91, 84]
Sorted scores: [76, 78, 84, 85, 88, 91, 92, 95]
PS C:\Users\Deepthi Varma\OneDrive\Desktop\AI_LAB>
```

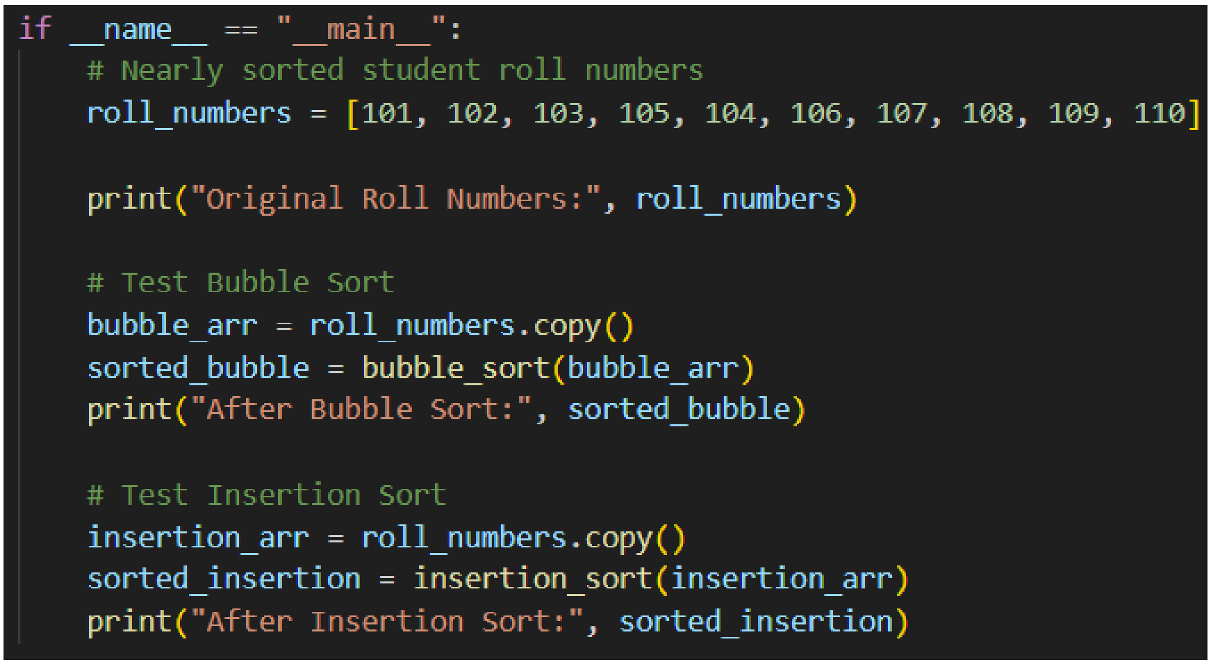
Explanation:

1. The program uses Bubble Sort, where adjacent scores are compared and swapped if they are in the wrong order.
2. After each pass, the largest unsorted score moves to the end of the list.
3. A swapped flag is used to stop early if no swaps happen, meaning the list is already sorted.
4. The time complexity is O(n^2) in worst/average cases and O(n) in the best case (when already sorted).

Task-2:

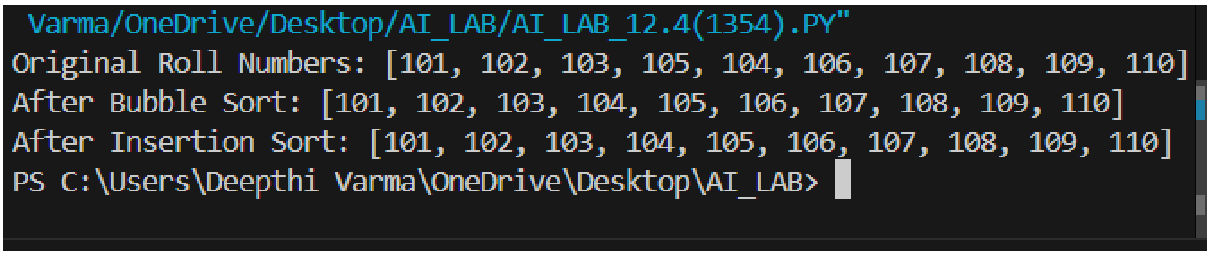
Prompt: Write a Python program implementing both Bubble Sort and Insertion Sort to sort nearly sorted student roll numbers.

Code:



```python
if __name__ == "__main__":
    # Nearly sorted student roll numbers
    roll_numbers = [101, 102, 103, 105, 104, 106, 107, 108, 109, 110]

    print("Original Roll Numbers:", roll_numbers)

    # Test Bubble Sort
    bubble_arr = roll_numbers.copy()
    sorted_bubble = bubble_sort(bubble_arr)
    print("After Bubble Sort:", sorted_bubble)

    # Test Insertion Sort
    insertion_arr = roll_numbers.copy()
    sorted_insertion = insertion_sort(insertion_arr)
    print("After Insertion Sort:", sorted_insertion)
```

Output:



```
 Varma/OneDrive/Desktop/AI_LAB/AI_LAB_12.4(1354).PY"
Original Roll Numbers: [101, 102, 103, 105, 104, 106, 107, 108, 109, 110]
After Bubble Sort: [101, 102, 103, 104, 105, 106, 107, 108, 109, 110]
After Insertion Sort: [101, 102, 103, 104, 105, 106, 107, 108, 109, 110]
PS C:\Users\Deepthi Varma\OneDrive\Desktop\AI_LAB>
```

Explanation: 1. Bubble Sort repeatedly compares and swaps adjacent elements, which causes unnecessary passes even if the list is almost sorted.

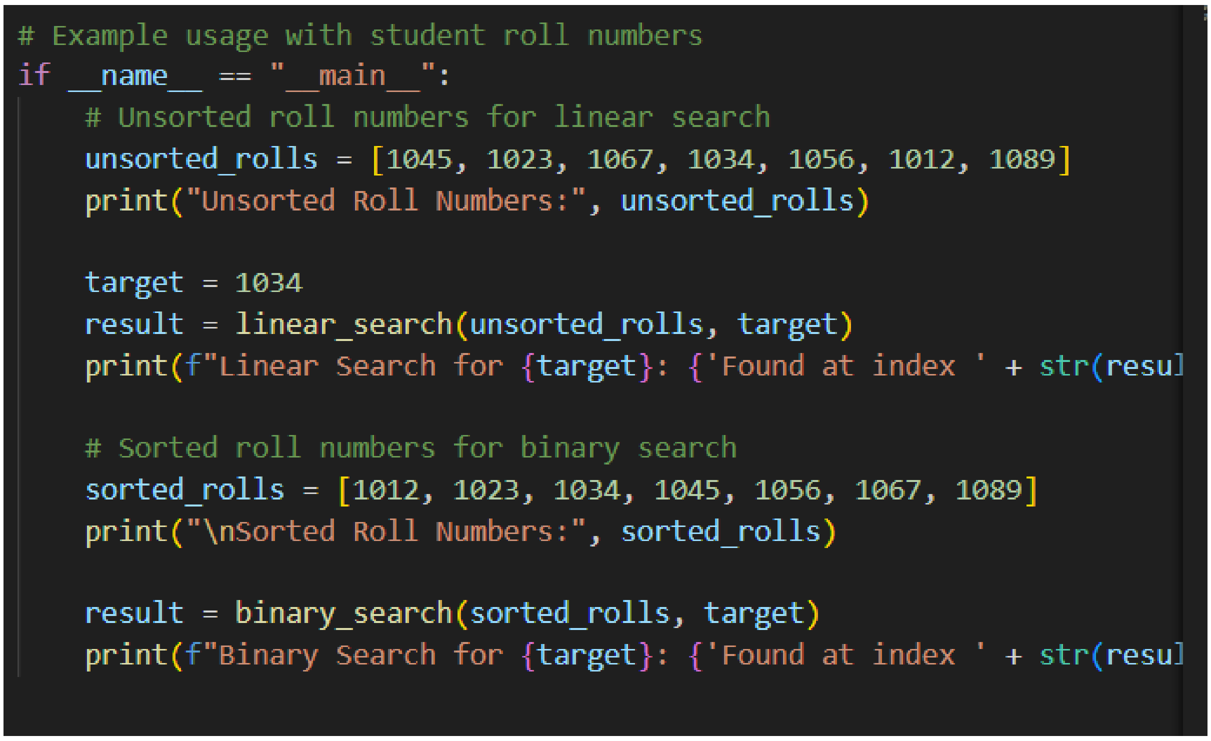
2. Insertion Sort places each element in its correct position by shifting only a few elements, making it efficient for nearly sorted data.

3. Therefore, Insertion Sort runs faster (close to O(n)) on nearly sorted input, while Bubble Sort still takes around $O(n^2)$ time.

Task-3:

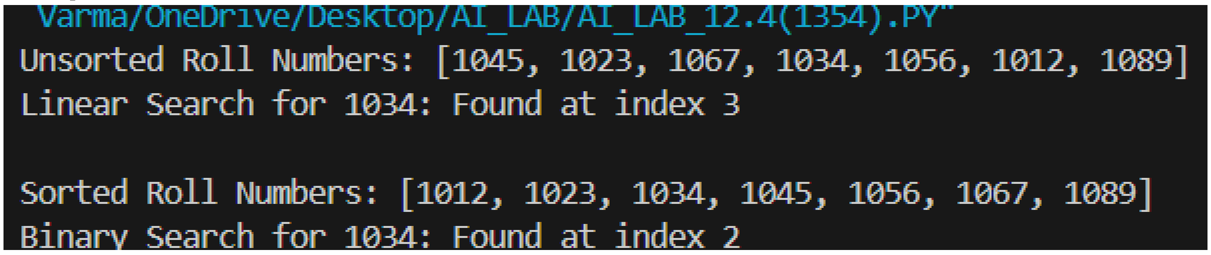
Prompt: Write a Python program implementing Linear Search (for unsorted student roll numbers) and Binary Search (for sorted data).

Code:



```python
# Example usage with student roll numbers
if __name__ == "__main__":
    # Unsorted roll numbers for linear search
    unsorted_rolls = [1045, 1023, 1067, 1034, 1056, 1012, 1089]
    print("Unsorted Roll Numbers:", unsorted_rolls)

    target = 1034
    result = linear_search(unsorted_rolls, target)
    print(f"Linear Search for {target}: {'Found at index ' + str(resul

    # Sorted roll numbers for binary search
    sorted_rolls = [1012, 1023, 1034, 1045, 1056, 1067, 1089]
    print("\nSorted Roll Numbers:", sorted_rolls)

    result = binary_search(sorted_rolls, target)
    print(f"Binary Search for {target}: {'Found at index ' + str(resul
```

Output:



```
 Varma/OneDrive/Desktop/AI_LAB/AI_LAB_12.4(1354).PY"
Unsorted Roll Numbers: [1045, 1023, 1067, 1034, 1056, 1012, 1089]
Linear Search for 1034: Found at index 3

Sorted Roll Numbers: [1012, 1023, 1034, 1045, 1056, 1067, 1089]
Binary Search for 1034: Found at index 2
```

Explanation:

1.Linear Search checks each roll number one by one, so it works for both sorted and unsorted lists but takes O(n) time.

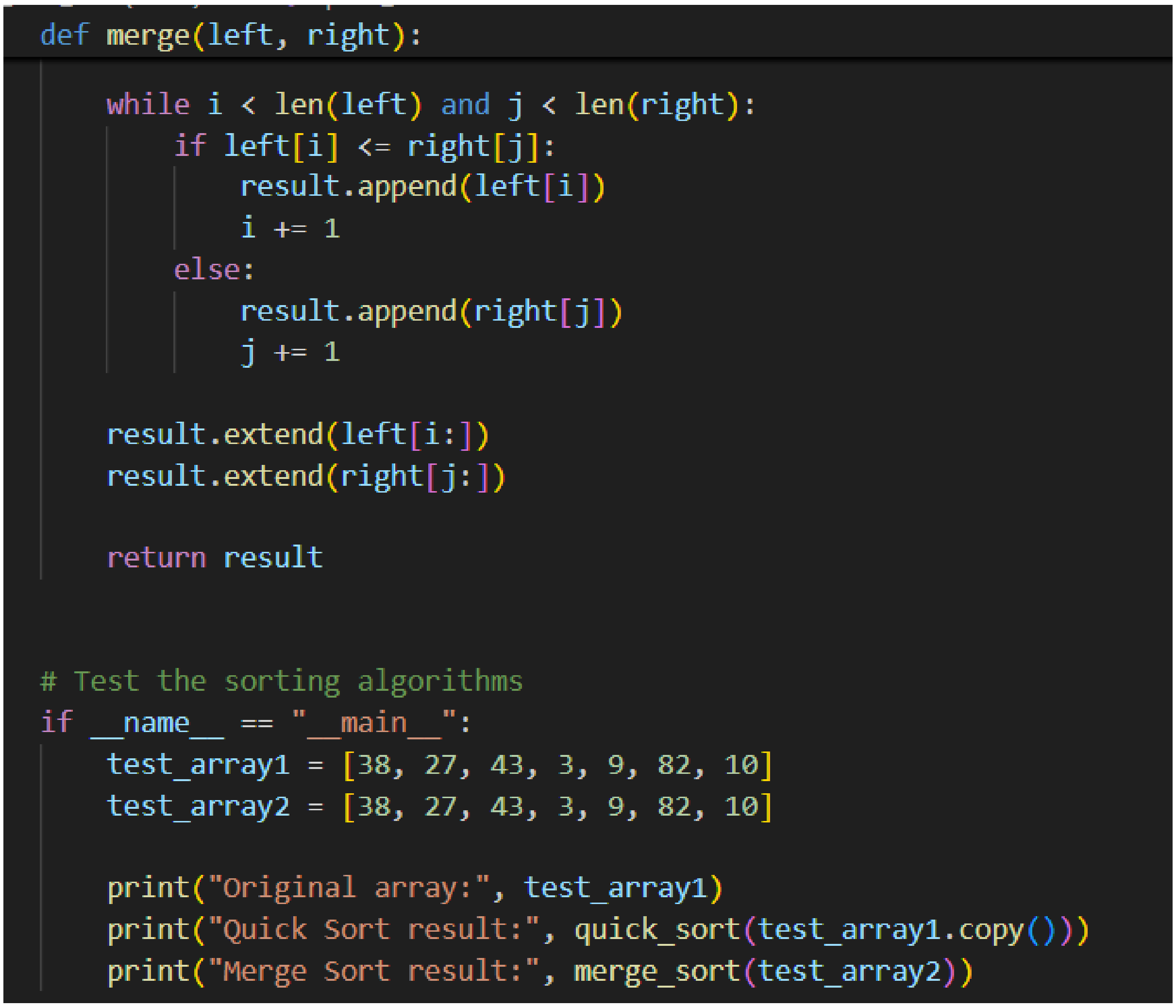
2.Binary Search works only on sorted data, repeatedly dividing the list in half, giving a faster time complexity of O(log n).

3.Therefore, Binary Search is more efficient for large sorted datasets, while Linear Search is suitable for small or unsorted data.

Task-4:

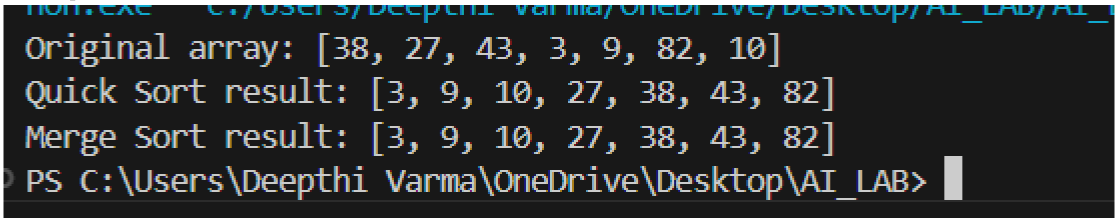
Prompt: Write a Python program completing partially written recursive functions for Quick Sort and Merge Sort.

Code:



```python
def merge(left, right):

    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 the sorting algorithms
if __name__ == "__main__":
    test_array1 = [38, 27, 43, 3, 9, 82, 10]
    test_array2 = [38, 27, 43, 3, 9, 82, 10]

    print("Original array:", test_array1)
    print("Quick Sort result:", quick_sort(test_array1.copy()))
    print("Merge Sort result:", merge_sort(test_array2))
```

Output:



```
Original array: [38, 27, 43, 3, 9, 82, 10]
Quick Sort result: [3, 9, 10, 27, 38, 43, 82]
Merge Sort result: [3, 9, 10, 27, 38, 43, 82]
PS C:\Users\Deepthi Varma\OneDrive\Desktop\AI_LAB>
```

Explanation: 1. Quick Sort uses recursion by selecting a pivot, dividing the list into smaller sublists (less than and greater than pivot), and recursively sorting those sublists until the base case (size ≤ 1).

2. Merge Sort recursively divides the list into two halves until single elements remain, then merges them back in sorted order, combining smaller sorted lists into larger ones.

3. In terms of complexity, Merge Sort guarantees O(n log n) in all cases, while Quick Sort has O(n log n) average/best case but $O(n^2)$ worst case (e.g., already sorted data with poor pivot choice).