**ASSIGNMENT:12.4**

# NAME: P.NISHANTH

# HTNO:2403A52066

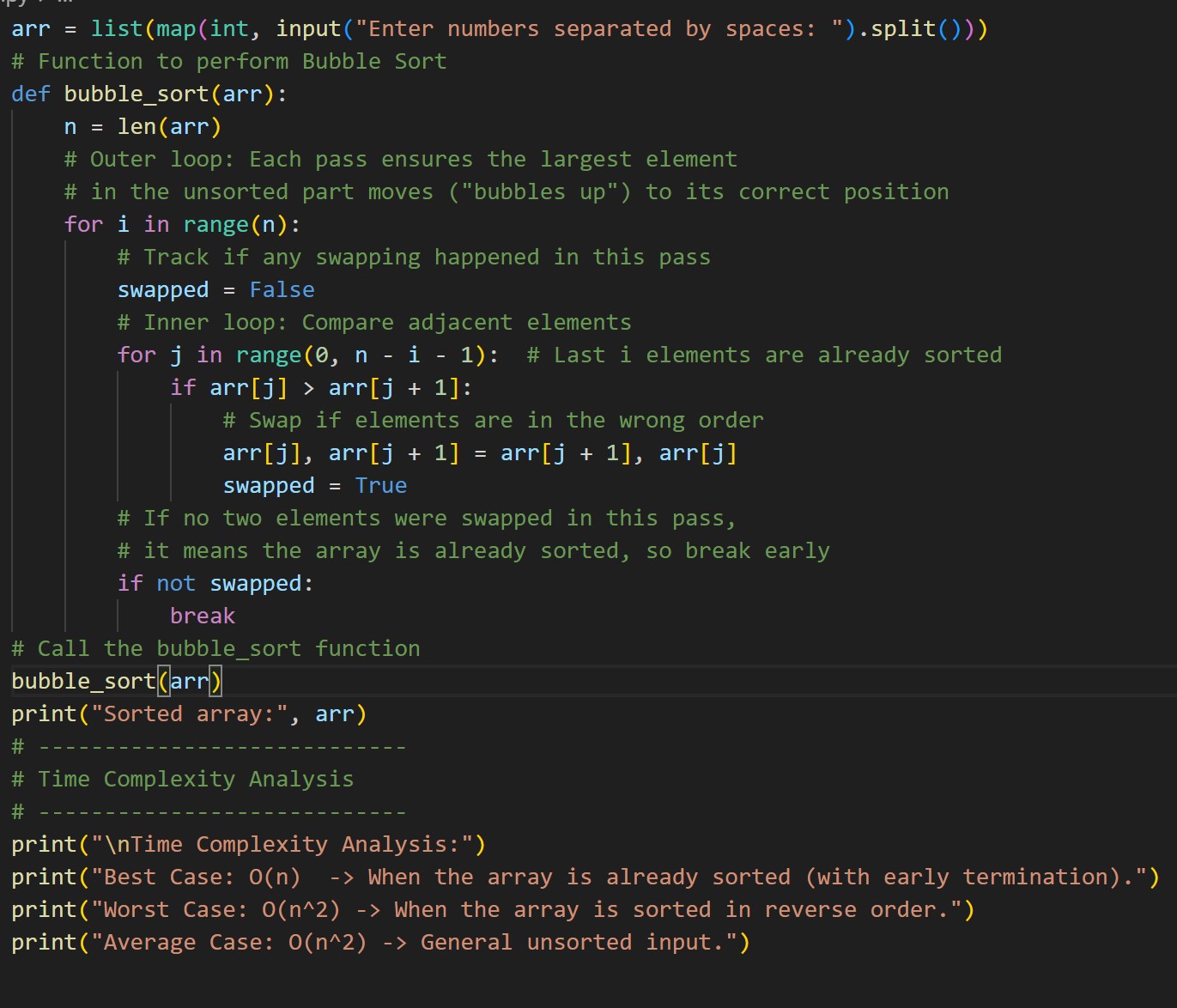
# BATCH:03

TASK 1:

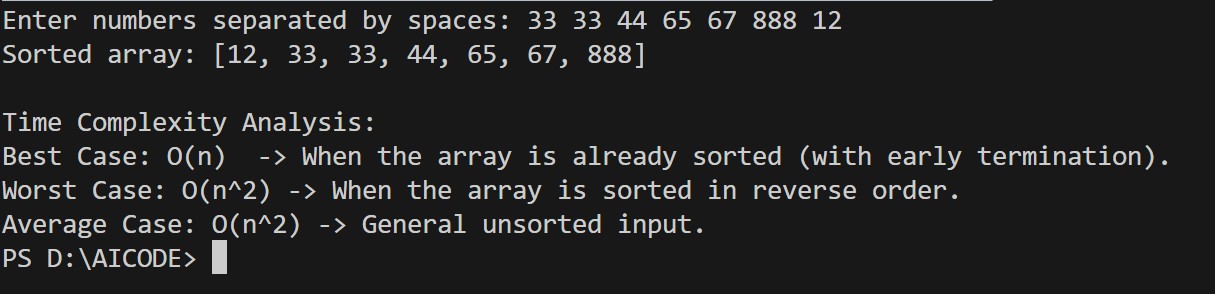
PROMPT:

"Write a Python program to implement Bubble Sort. The program should take dynamic input from the user (not hardcoded values). Add detailed inline comments explaining key logic steps (like swapping, outer loop passes, and termination conditions). At the end, also provide a short explanation of the time complexity (best case, worst case, average case)."

CODE:



OUTPUT:



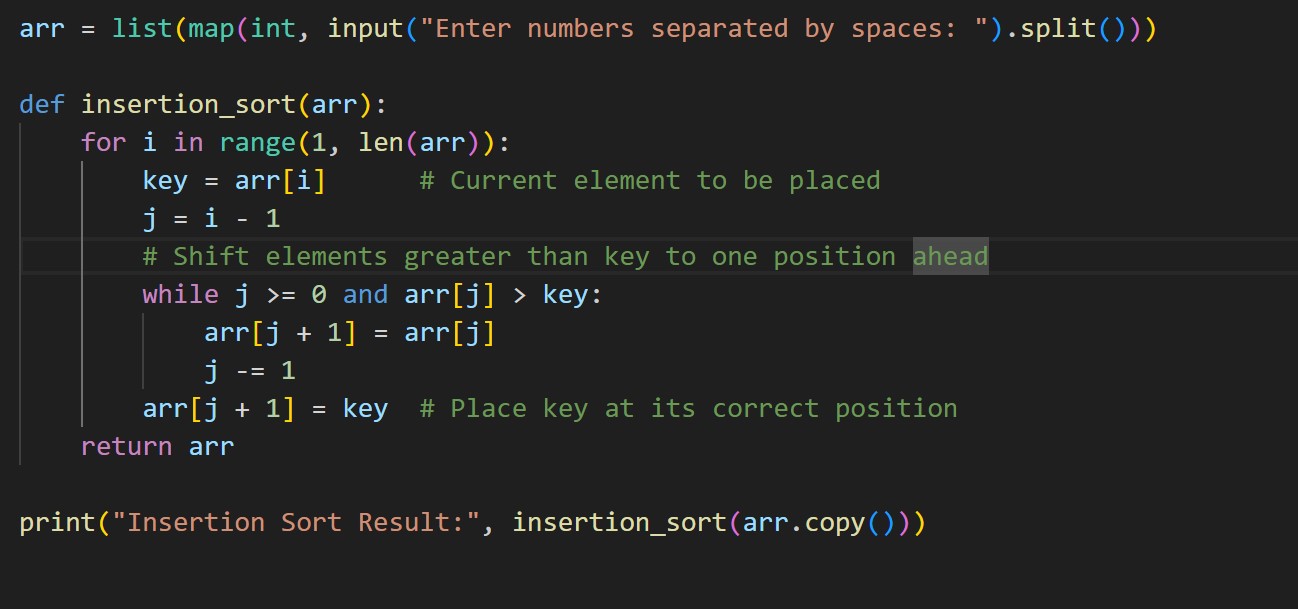
OBSERVATION:

After executing the program, the given numbers were sorted correctly in ascending order. The algorithm compared and swapped adjacent elements in each pass until the list became sorted.

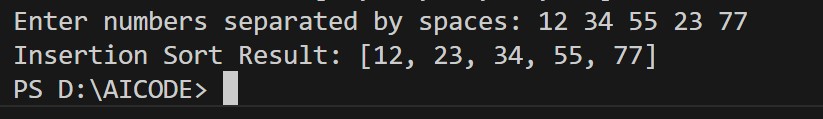
TASK2:

PROMPT:

"Write a Python program to implement Bubble Sort, then suggest a more efficient algorithm for partially sorted arrays. Provide the alternative as Insertion Sort. Both programs should take dynamic input (not hardcoded). Add inline comments to explain the logic. At the end, explain why Insertion Sort is more efficient than Bubble Sort on nearly sorted data and compare their time complexities." CODE:



OUTPUT:



AI Explanation:

**Bubble Sort** makes multiple passes, comparing and swapping adjacent elements even if the array is already almost sorted.

**Insertion Sort** only shifts elements until the right position for the current element is found.

On **nearly sorted input**, Insertion Sort requires far fewer operations, often close to **O(n)**.

**Time Complexity Comparison**

**Bubble Sort:**

Best Case: O(n)

Worst & Average Case: O(n²)

**Insertion Sort:**

Best Case: O(n) (for nearly sorted data)

Worst & Average Case: O(n²)

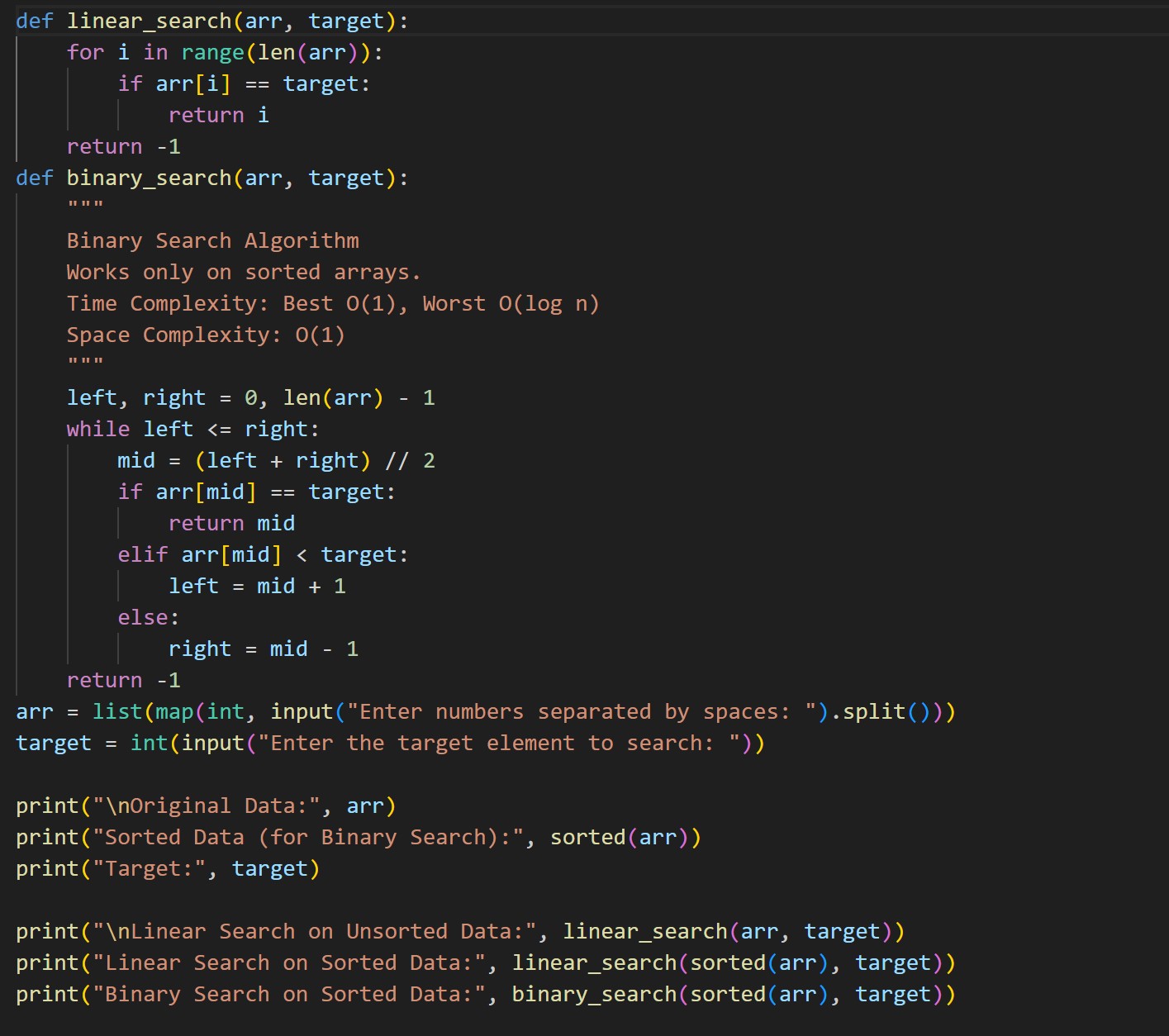
OBSERVATION:

Both Bubble Sort and Insertion Sort gave the correct sorted output. However, Insertion Sort worked faster on the nearly sorted input because it only shifted elements instead of repeatedly swapping in every pass. This makes Insertion Sort more efficient than Bubble Sort for partially sorted arrays.

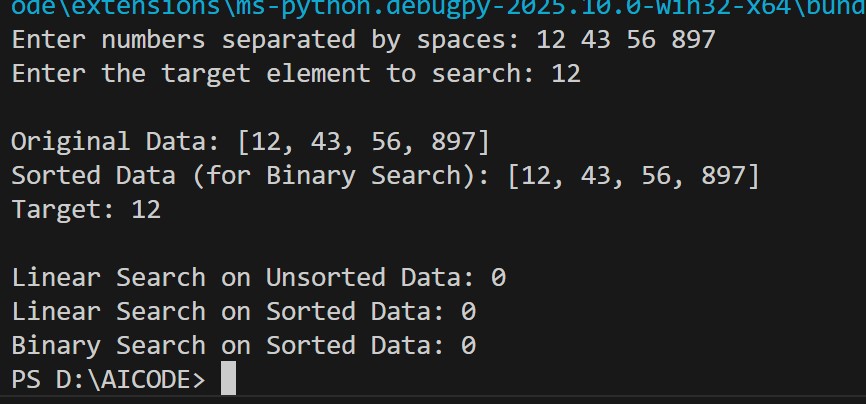
TASK3:

PROMPT:

"Write Python programs to implement Linear Search and Binary Search. Include proper docstrings and performance notes for both algorithms. Test them on sorted and unsorted input data. At the end, explain when Binary Search is preferable. Provide a student-style observation table comparing the performance of Linear Search and Binary Search." CODE:



OUTPUT:



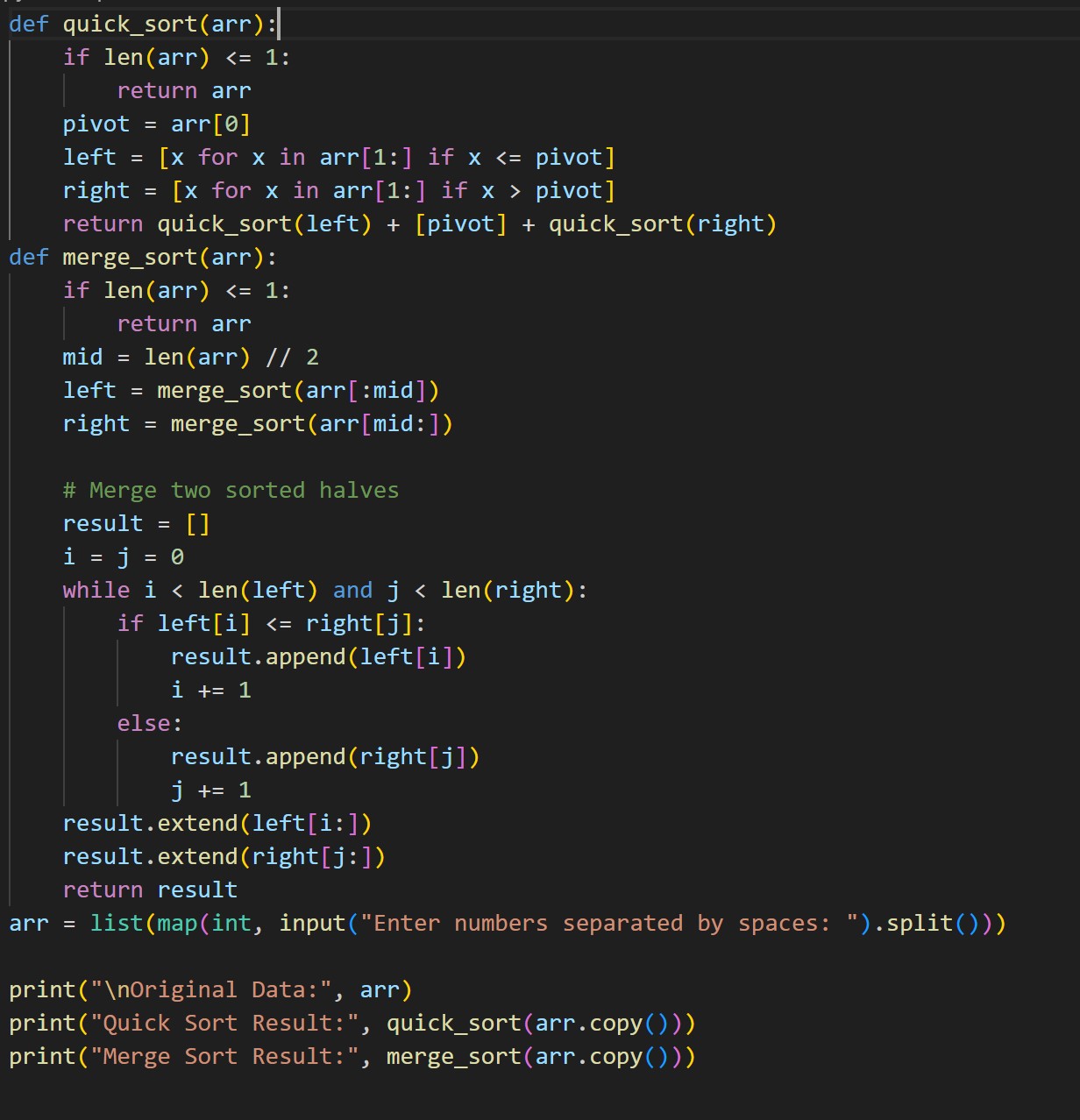
OBSERVATION: Both Linear Search and Binary Search successfully found the target element. Linear Search works on both sorted and unsorted arrays but may take longer for large lists. Binary Search is faster on sorted arrays because it reduces the number of comparisons using a divide-and-conquer approach.

TASK4:

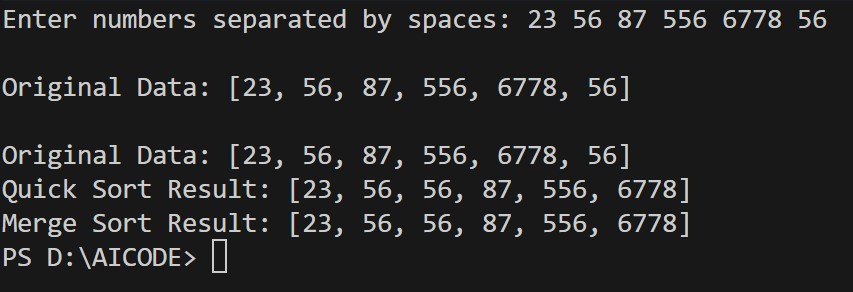
PROMPT:

"Complete the partially implemented recursive functions for Quick Sort and Merge Sort in Python. Add docstrings explaining the parameters, return values, and performance. Compare both algorithms on random, sorted, and reverse-sorted lists, and provide an explanation of average, best, and worst-case time complexities."

CODE:



OUTPUT:

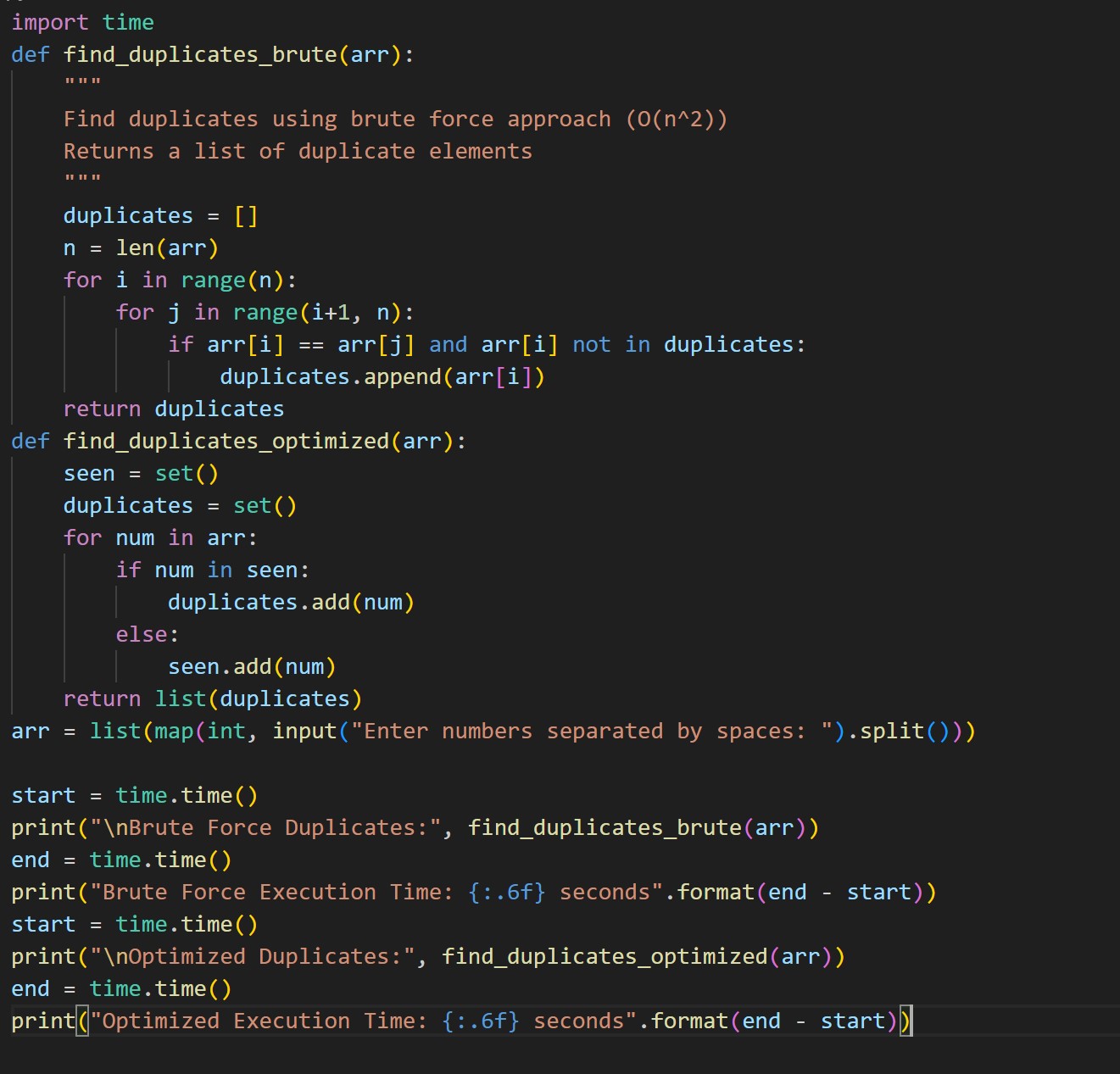


OBSERVATION:

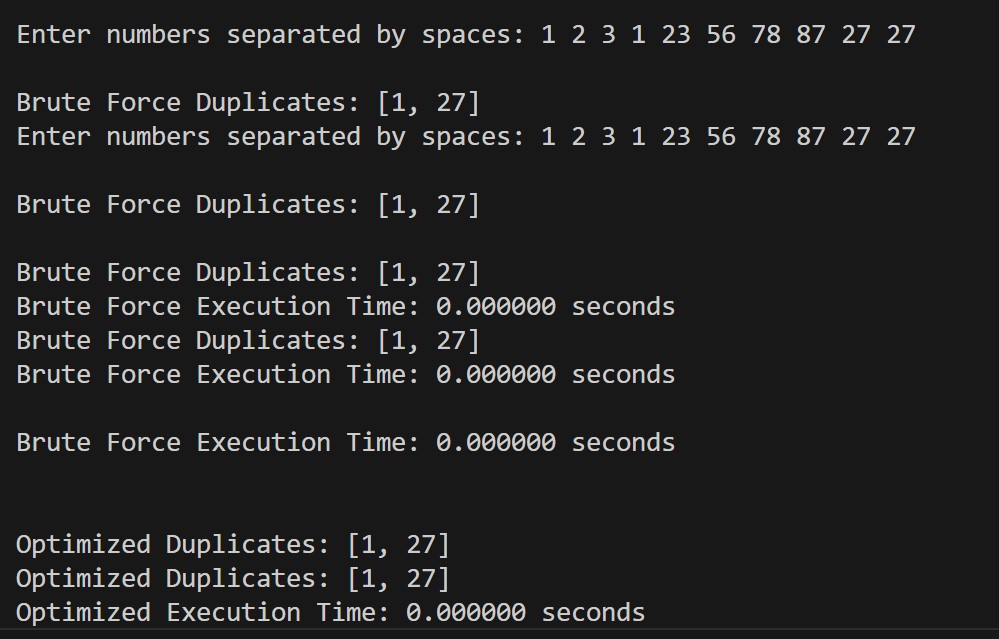
Both Quick Sort and Merge Sort correctly sorted the list. Quick Sort is generally faster on random data but may be slower on already sorted or reverse-sorted lists due to pivot choice. Merge Sort performs consistently across all types of data but uses additional memory for merging. For large datasets where memory is not a concern, Merge Sort is reliable; for in-place sorting with average efficiency, Quick Sort is preferred.

TASK5:

"Write a Python program to find duplicates in a list using a naive O(n²) approach. Then provide an optimized version using sets or dictionaries to reduce the time complexity to O(n). Include dynamic input from the user. Compare execution times on large input and explain how the optimization improves complexity." CODE:



OUTPUT:



OBSERVATION:

Both versions correctly identified duplicates. The brute force method works but becomes very slow with large inputs because it compares each pair of elements. The optimized version is much faster for large datasets because it uses a set to track duplicates, reducing the time complexity from O(n²) to O(n).