ASSIGNMENT:12.4

NAME: V.Sai shashank

HTNO:2403A52061

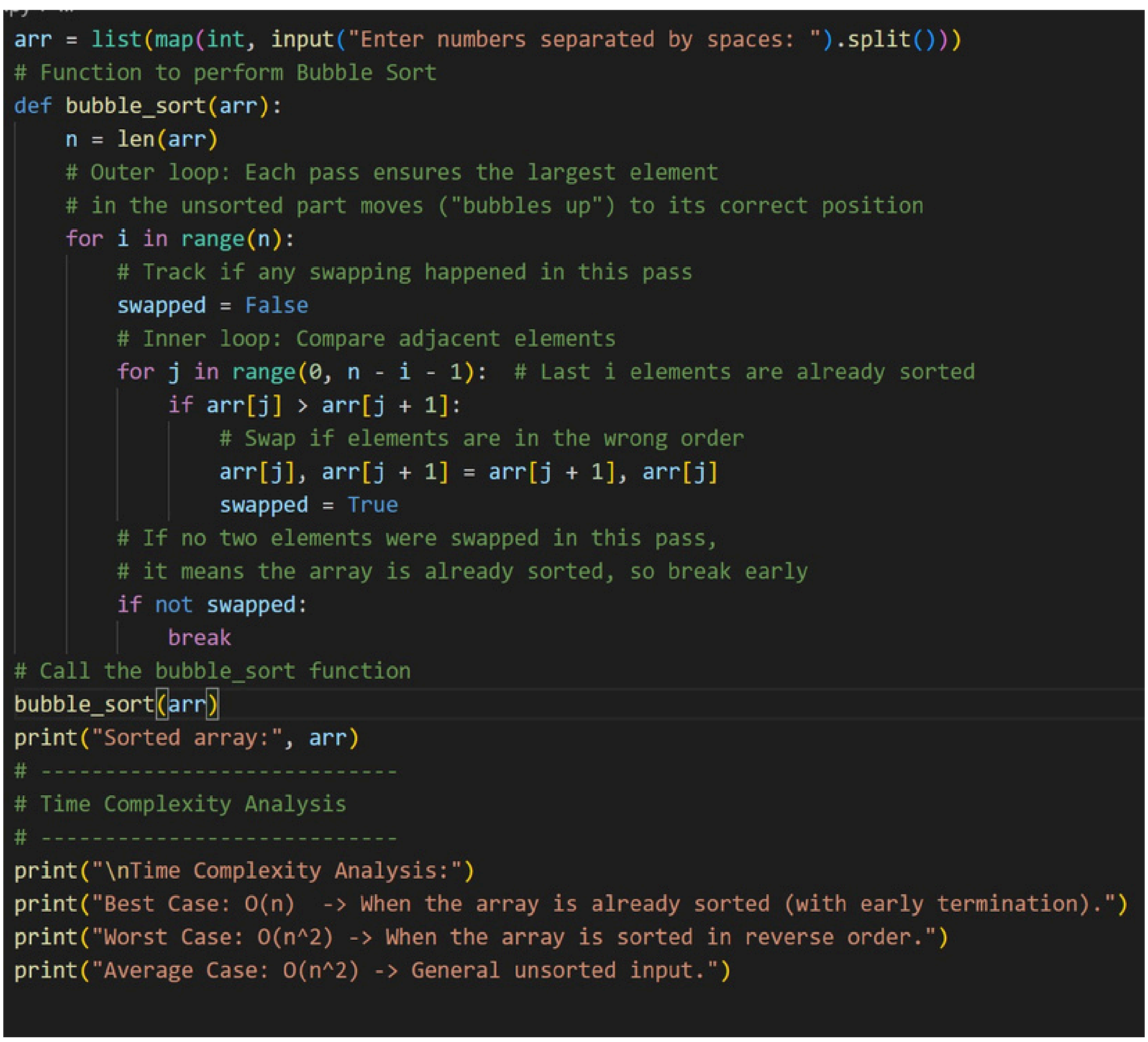
BATCH:03

TASK 1:

PROMPT:

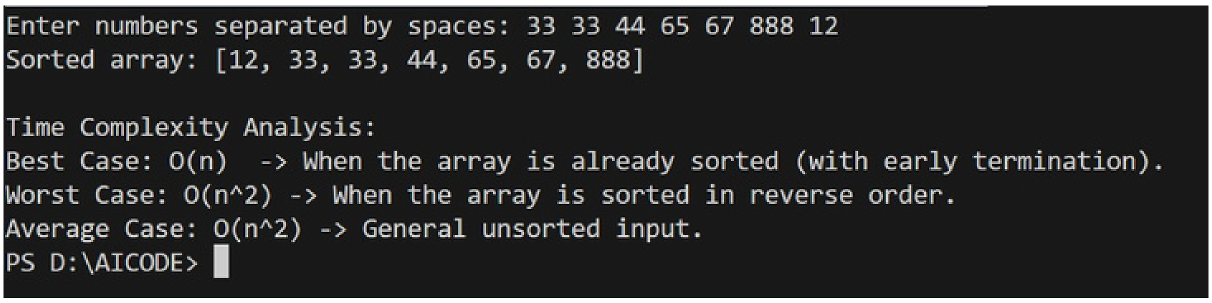
"Write a Python program to implement Bubble Sort. The program should take dynamic input from the user (not hardcoded values). Add detailed inline comments explaining key logic steps (like swapping, outer loop passes, and termination conditions). At the end, also provide a short explanation of the time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

CODE:



```python
arr = list(map(int, input("Enter numbers separated by spaces: ").split()))
# Function to perform Bubble Sort
def bubble_sort(arr):
    n = len(arr)
    # Outer loop: Each pass ensures the largest element
    # in the unsorted part moves ("bubbles up") to its correct position
    for i in range(n):
        # Track if any swapping happened in this pass
        swapped = False
        # Inner loop: Compare adjacent elements
        for j in range(0, n - i - 1):  # Last i elements are already sorted
            if arr[j] > arr[j + 1]:
                # Swap if elements are in the wrong order
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        # If no two elements were swapped in this pass,
        # it means the array is already sorted, so break early
        if not swapped:
            break
# Call the bubble_sort function
bubble_sort(arr)
print("Sorted array:", arr)
# ----------------------------
# Time Complexity Analysis
# ----------------------------
print("\nTime Complexity Analysis:")
print("Best Case: O(n)  -> When the array is already sorted (with early termination).")
print("Worst Case: O(n^2) -> When the array is sorted in reverse order.")
print("Average Case: O(n^2) -> General unsorted input.")
```

OUTPUT:



```
Enter numbers separated by spaces: 33 33 44 65 67 888 12
Sorted array: [12, 33, 33, 44, 65, 67, 888]

Time Complexity Analysis:
Best Case: O(n)  -> When the array is already sorted (with early termination).
Worst Case: O(n^2) -> When the array is sorted in reverse order.
Average Case: O(n^2) -> General unsorted input.
PS D:\AICODE> 
```

OBSERVATION:

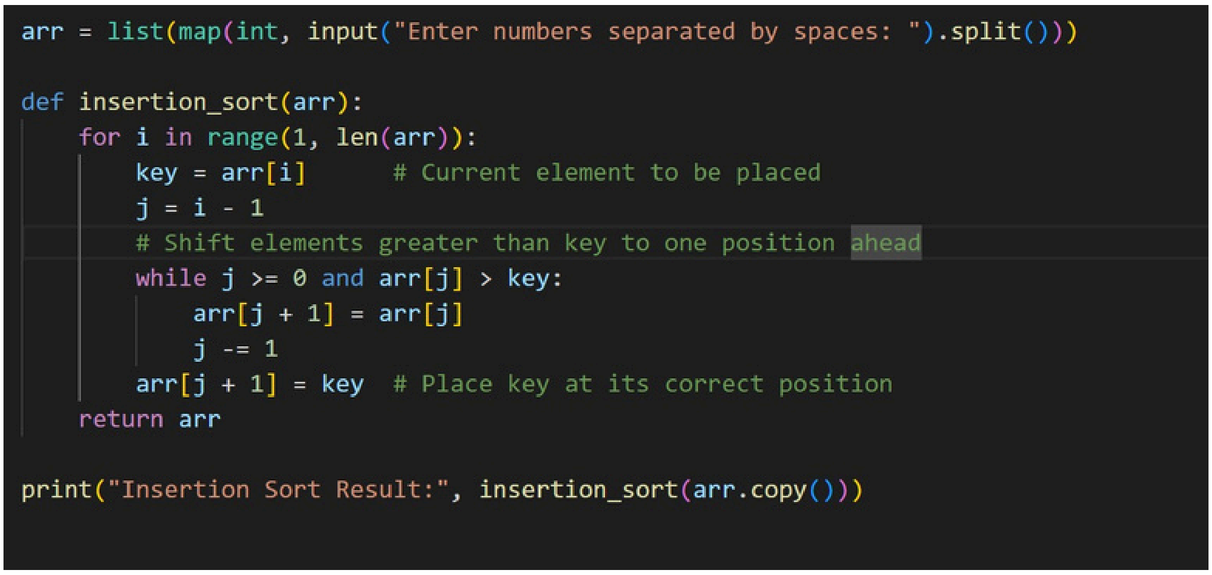
After executing the program, the given numbers were sorted correctly in ascending order. The algorithm compared and swapped adjacent elements in each pass until the list became sorted.

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:

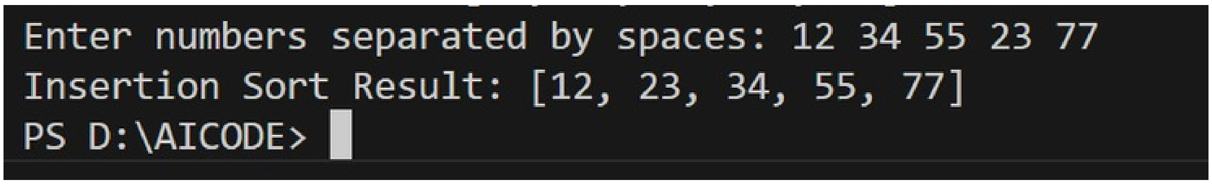


```python
arr = list(map(int, input("Enter numbers separated by spaces: ").split()))

def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]      # Current element to be placed
        j = i - 1
        # Shift elements greater than key to one position ahead
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key  # Place key at its correct position
    return arr

print("Insertion Sort Result:", insertion_sort(arr.copy()))
```

OUTPUT:



```
Enter numbers separated by spaces: 12 34 55 23 77
Insertion Sort Result: [12, 23, 34, 55, 77]
PS D:\AICODE>
```

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 nearlysorted input, Insertion Sort requires far fewer operations, often close to O(n).

Time Complexity Comparison

Bubble Sort:

Best Case: O(n)

Worst & Average Case: $O(n^2)$

Insertion Sort:

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

Worst & Average Case: O(n^2)

OBSERVATION:

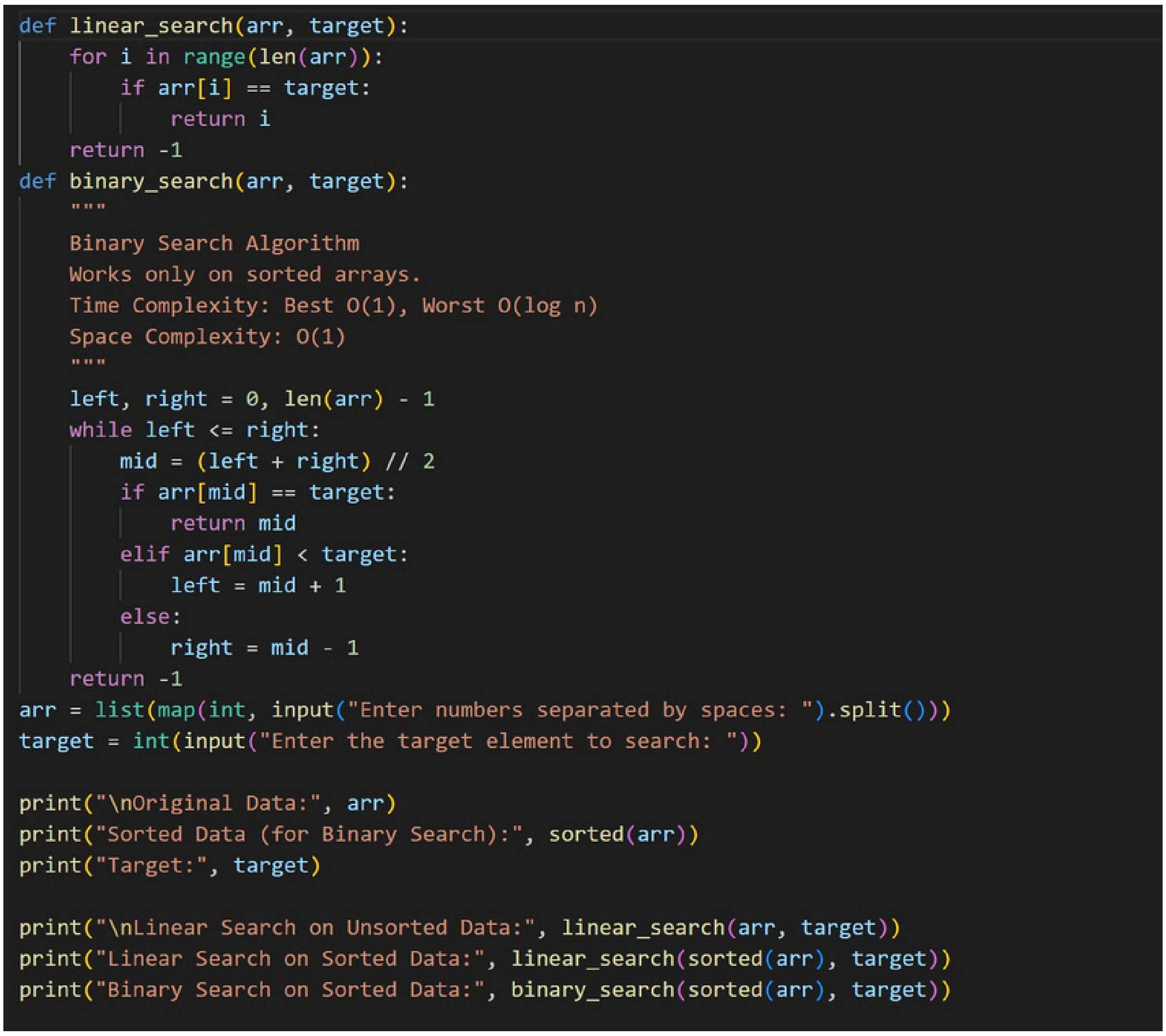
Both Bubble Sort and Insertion Sort gave the correct sorted output. However, Insertion Sort worked faster on the nearly sorted input because it only shifted elements instead of repeatedly swapping in every pass. This makes Insertion Sort more efficient than Bubble Sort for partially sorted arrays.

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:

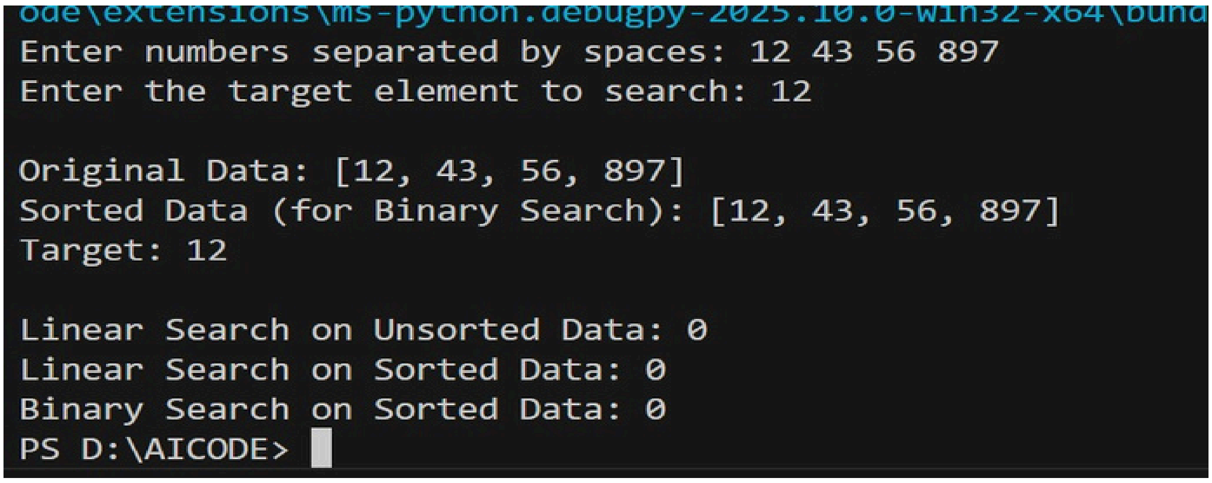


```python
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
def binary_search(arr, target):
    """
    Binary Search Algorithm
    Works only on sorted arrays.
    Time Complexity: Best O(1), Worst O(log n)
    Space Complexity: O(1)
    """
    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
arr = list(map(int, input("Enter numbers separated by spaces: ").split()))
target = int(input("Enter the target element to search: "))

print("\nOriginal Data:", arr)
print("Sorted Data (for Binary Search):", sorted(arr))
print("Target:", target)

print("\nLinear Search on Unsorted Data:", linear_search(arr, target))
print("Linear Search on Sorted Data:", linear_search(sorted(arr), target))
print("Binary Search on Sorted Data:", binary_search(sorted(arr), target))
```

OUTPUT:



```
ode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bund
Enter numbers separated by spaces: 12 43 56 897
Enter the target element to search: 12

Original Data: [12, 43, 56, 897]
Sorted Data (for Binary Search): [12, 43, 56, 897]
Target: 12

Linear Search on Unsorted Data: 0
Linear Search on Sorted Data: 0
Binary Search on Sorted Data: 0
PS D:\AICODE>
```

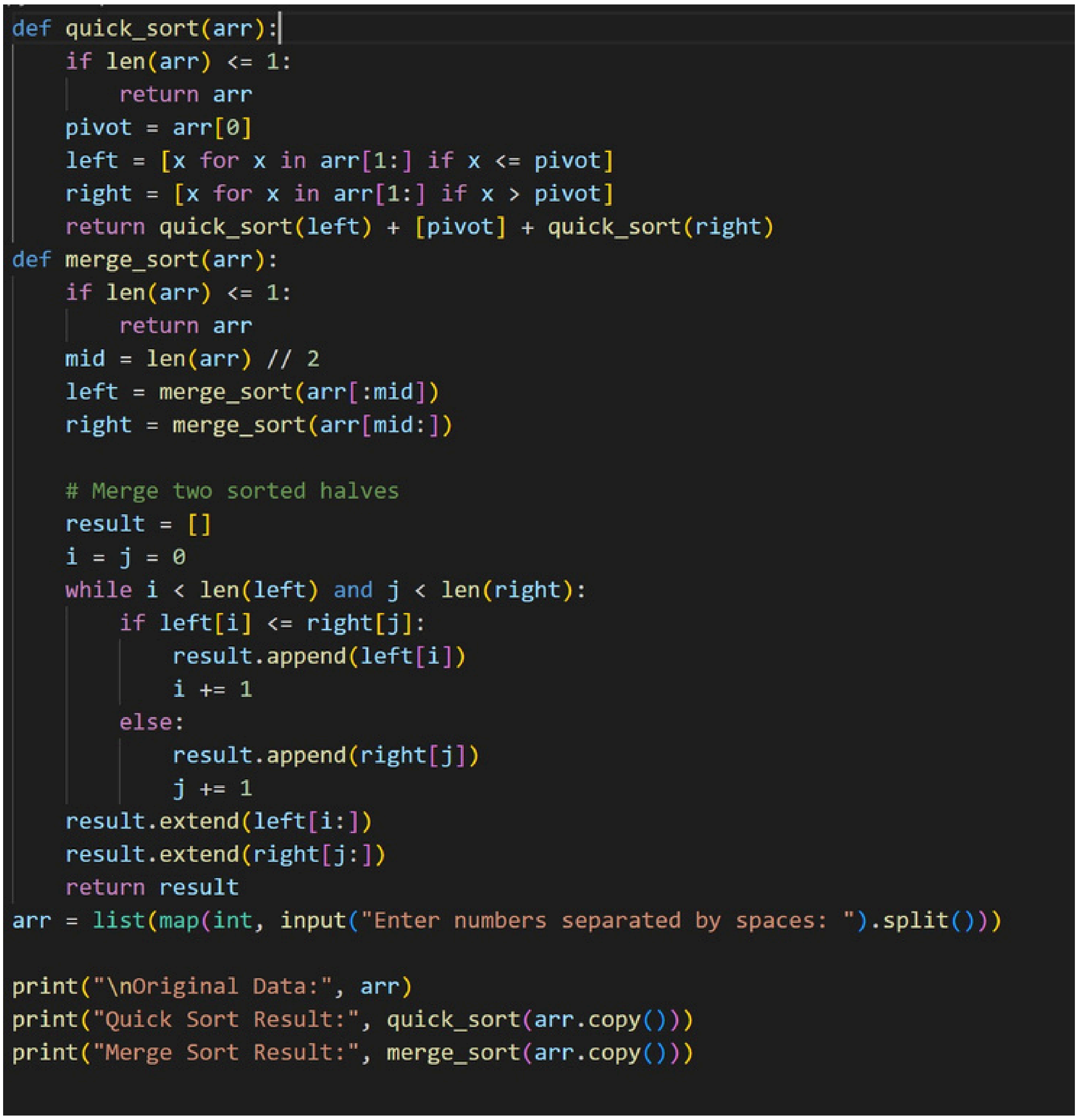
OBSERVATION: Both Linear Search and Binary Search successfully found the target element. Linear Search works on both sorted and unsorted arrays but may take longer for large lists. Binary Search is faster on sorted arrays because it reduces the number of comparisons using a divide-and-conquer approach.

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:

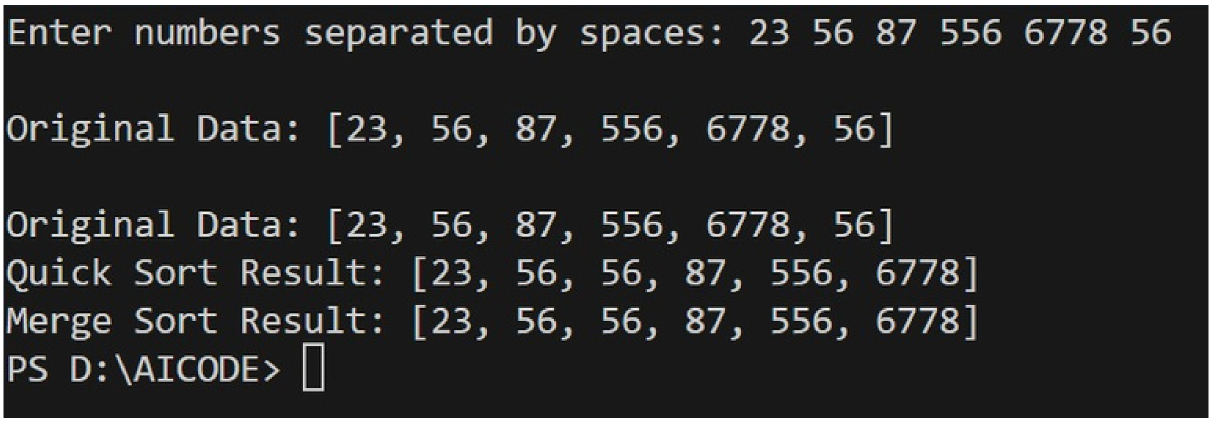


```python
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[0]
    left = [x for x in arr[1:] if x <= pivot]
    right = [x for x in arr[1:] if x > pivot]
    return quick_sort(left) + [pivot] + quick_sort(right)
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    # Merge two sorted halves
    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
arr = list(map(int, input("Enter numbers separated by spaces: ").split()))

print("\nOriginal Data:", arr)
print("Quick Sort Result:", quick_sort(arr.copy()))
print("Merge Sort Result:", merge_sort(arr.copy()))
```

OUTPUT:



```
Enter numbers separated by spaces: 23 56 87 556 6778 56

Original Data: [23, 56, 87, 556, 6778, 56]

Original Data: [23, 56, 87, 556, 6778, 56]
Quick Sort Result: [23, 56, 56, 87, 556, 6778]
Merge Sort Result: [23, 56, 56, 87, 556, 6778]
PS D:\AICODE> 
```

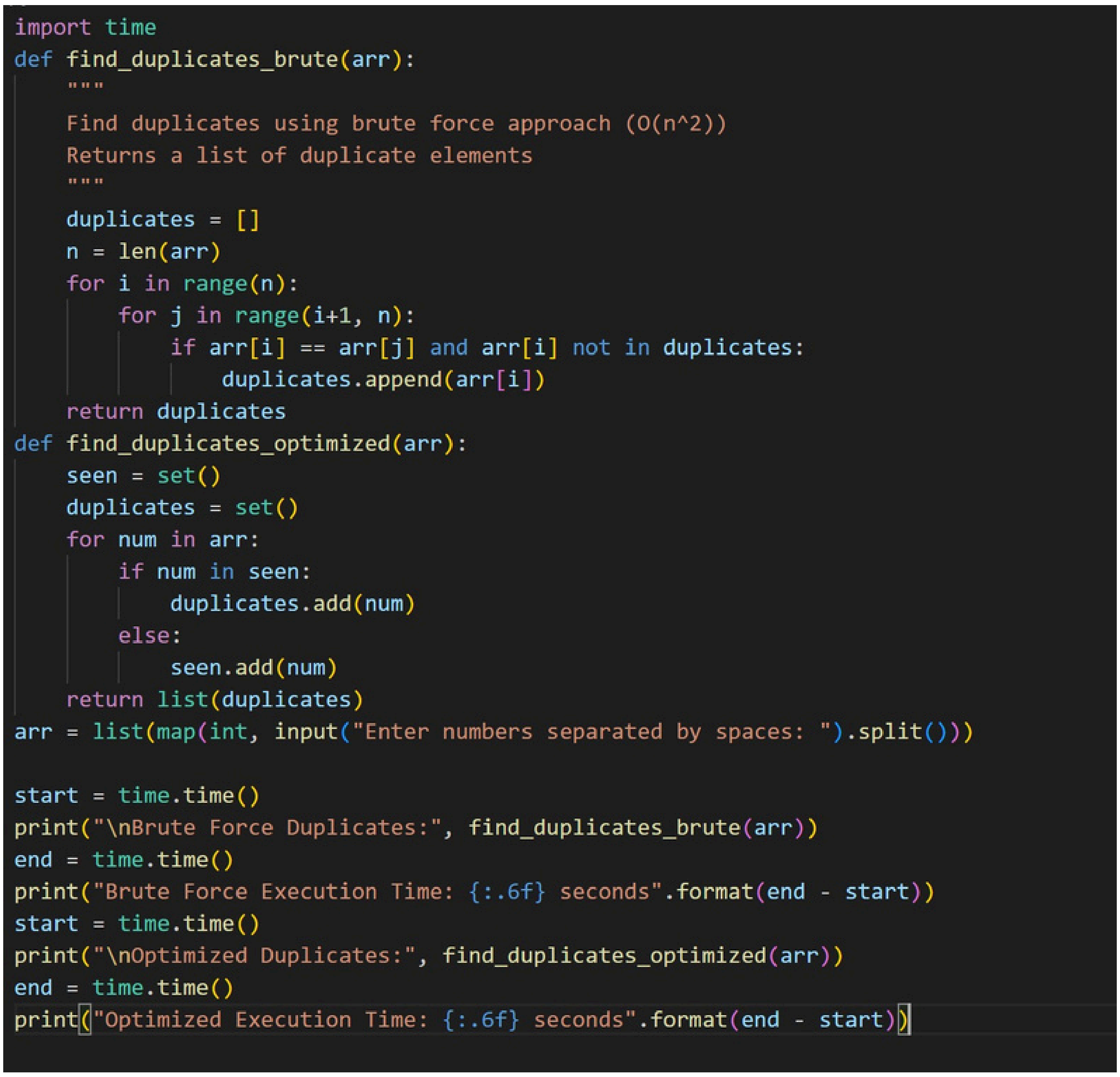
OBSERVATION:

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."

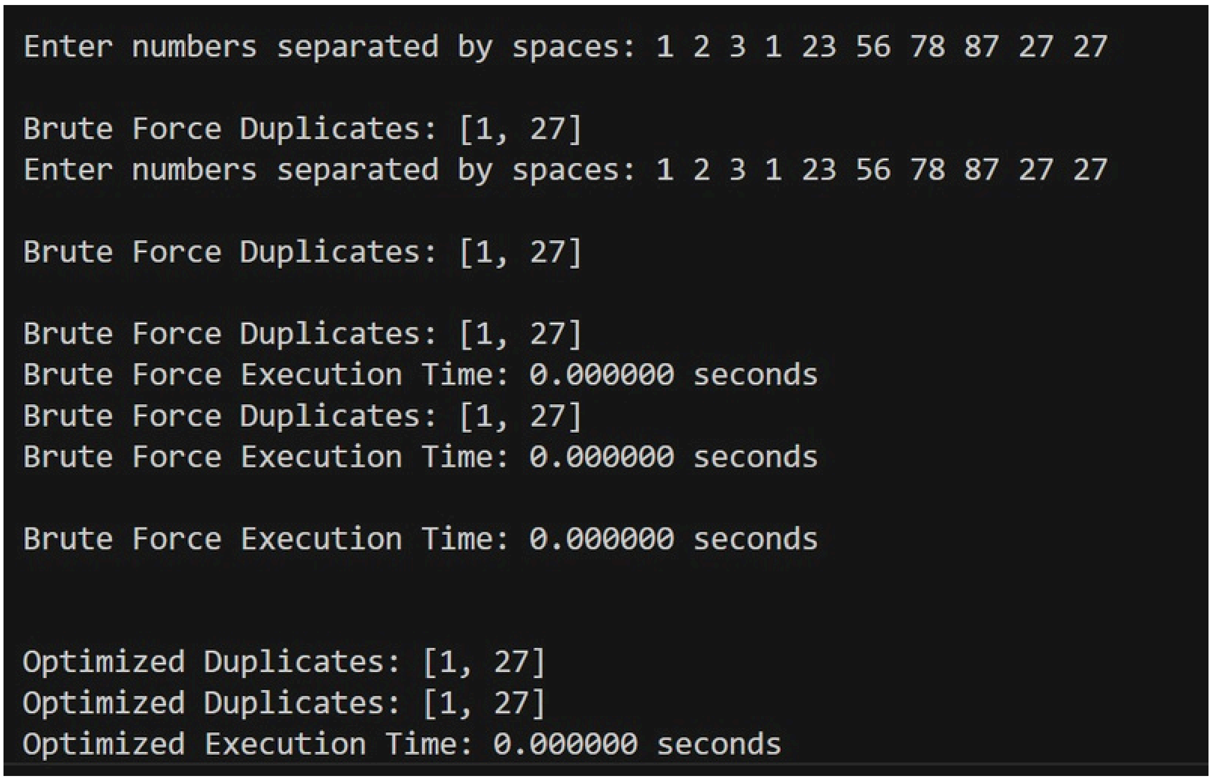
CODE:



```python
import time
def find_duplicates_brute(arr):
    """
    Find duplicates using brute force approach (O(n^2))
    Returns a list of duplicate elements
    """
    duplicates = []
    n = len(arr)
    for i in range(n):
        for j in range(i+1, n):
            if arr[i] == arr[j] and arr[i] not in duplicates:
                duplicates.append(arr[i])
    return duplicates
def find_duplicates_optimized(arr):
    seen = set()
    duplicates = set()
    for num in arr:
        if num in seen:
            duplicates.add(num)
        else:
            seen.add(num)
    return list(duplicates)
arr = list(map(int, input("Enter numbers separated by spaces: ").split()))

start = time.time()
print("\nBrute Force Duplicates:", find_duplicates_brute(arr))
end = time.time()
print("Brute Force Execution Time: {:.6f} seconds".format(end - start))
start = time.time()
print("\nOptimized Duplicates:", find_duplicates_optimized(arr))
end = time.time()
print("Optimized Execution Time: {:.6f} seconds".format(end - start))
```

OUTPUT:



```
Enter numbers separated by spaces: 1 2 3 1 23 56 78 87 27 27

Brute Force Duplicates: [1, 27]
Enter numbers separated by spaces: 1 2 3 1 23 56 78 87 27 27

Brute Force Duplicates: [1, 27]

Brute Force Duplicates: [1, 27]
Brute Force Execution Time: 0.000000 seconds
Brute Force Duplicates: [1, 27]
Brute Force Execution Time: 0.000000 seconds

Brute Force Execution Time: 0.000000 seconds


Optimized Duplicates: [1, 27]
Optimized Duplicates: [1, 27]
Optimized Execution Time: 0.000000 seconds
```

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 $O(n^2)$ to $O(n)$.