AI ASSISTED CODING

Lab Assignment-12.4

Name : T.Shylasri

H.T.NO: 2303A51876

Batch-14(LAB-12)

Date: 26-02-2026

ASSIGNMENT-12.4

Lab 12 – Algorithms with AI Assistance – Sorting, Searching, and Optimizing Algorithms

Lab Objectives

• Apply AI-assisted programming to implement and optimize sorting and searching algorithms.

• Compare different algorithms in terms of efficiency and use cases.

• Understand how AI tools can suggest optimized code and

complexity improvements.

Learning Outcome

After completing this assignment, students will be able to:

• Implement classical algorithms with AI assistance

• Compare algorithm efficiency using real-world scenarios

• Understand when optimization is necessary

• Critically evaluate AI-generated suggestions instead of blindly accepting them

Task 1: Bubble Sort for Ranking Exam Scores

Scenario

You are working on a college result processing system where a small

list of student scores needs to be sorted after every internal assessment.

Task Description

• Implement Bubble Sort in Python to sort a list of student scores.

Use an AI tool to:

o Insert inline comments explaining key operations such as comparisons, swaps, and iteration passes

o Identify early-termination conditions when the list becomes sorted

o Provide a brief time complexity analysis

Expected Outcome

• **A Bubble Sort implementation with:**

o AI-generated comments explaining the logic

o Clear explanation of best, average, and worst-case complexity

o Sample input/output showing sorted scores

AI Prompt

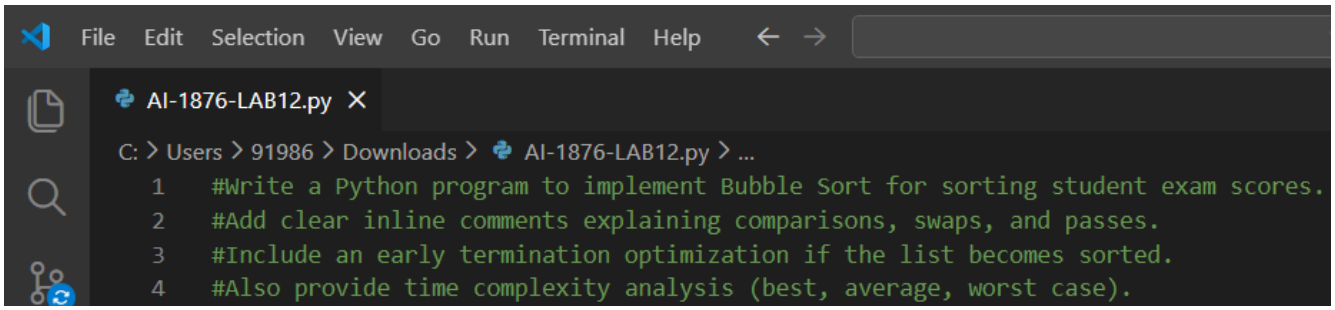
#Write a Python program to implement Bubble Sort for sorting student exam scores.

#Add clear inline comments explaining comparisons, swaps, and passes.

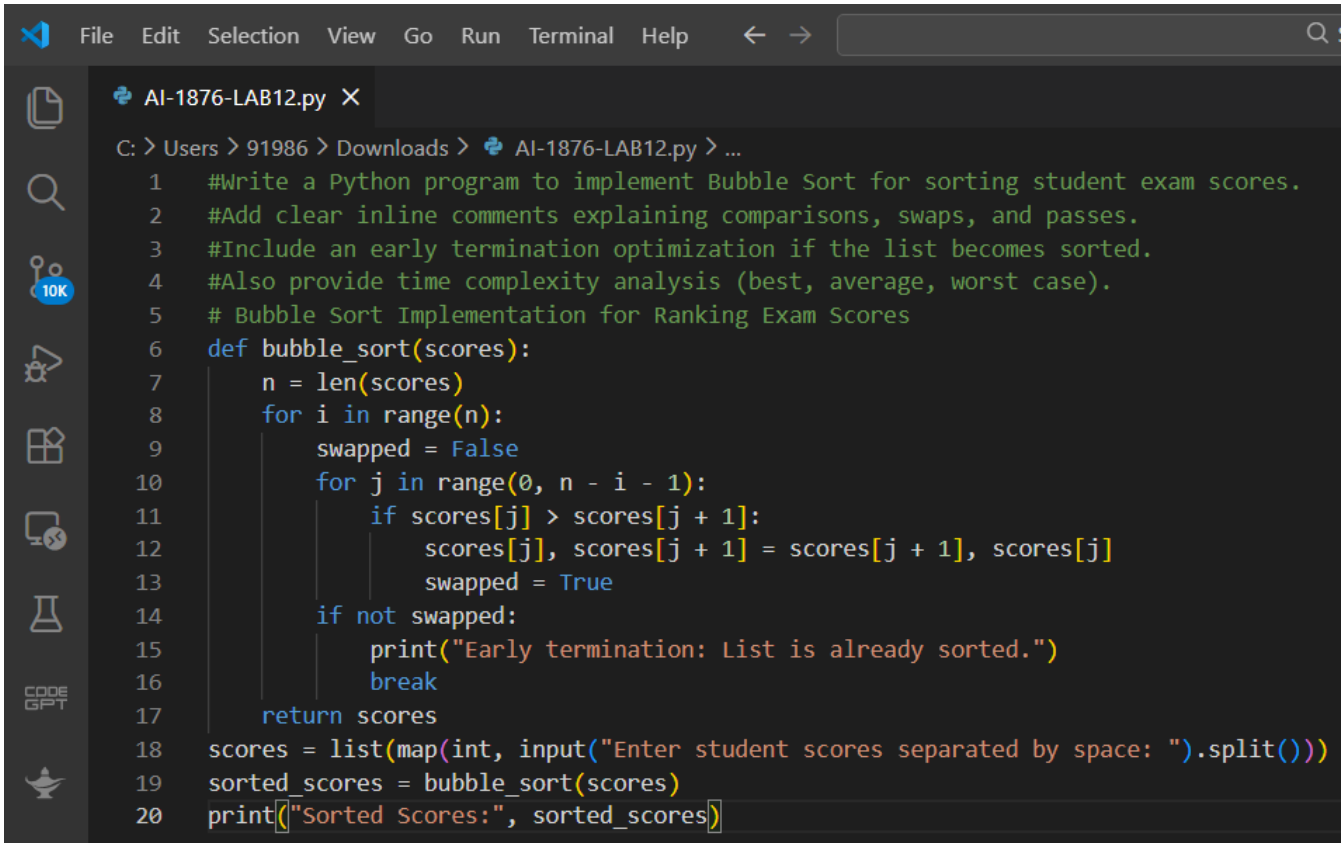
#Include an early termination optimization if the list becomes sorted.

#Also provide time complexity analysis (best, average, worst case).

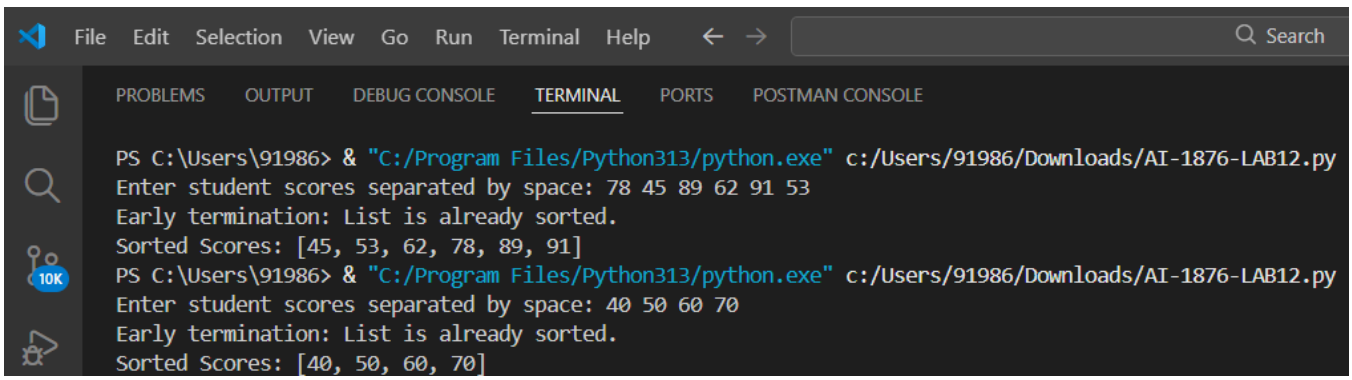
Prompt Screenshot



CODE



OUTPUT:



Time Complexity Analysis

Case	Time Complexity	Explanation
Best Case	**O(n)**	When the list is already sorted. Early termination stops after one pass.
Average Case	**$O(n^2)$**	Random order list requiring multiple comparisons and swaps.
Worst Case	**$O(n^2)$**	Reverse sorted list requiring maximum comparisons and swaps.

Space Complexity:

O(1) (In-place sorting, no extra memory used)

Observation

- Bubble Sort repeatedly compares adjacent elements.
- After each pass, the largest unsorted element moves to its correct position.
- The early termination optimization significantly improves performance when the list is already sorted.
- Suitable for small datasets like internal assessment scores.

Explanation

Bubble Sort works by:

1. Iterating through the list multiple times.
2. Comparing adjacent elements.
3. Swapping them if they are in the wrong order.
4. Repeating until no swaps are needed.

The optimization using a swapped flag ensures:

- If no swap happens during a pass, the algorithm stops early.
- This reduces unnecessary iterations.

Justification

- In a college internal assessment system, the number of students is usually small.
- Bubble Sort is:
    - Easy to implement
    - Easy to debug
    - Good for educational understanding
- Early termination makes it efficient for nearly sorted score lists.
- Although not suitable for very large datasets, it works well in this scenario.

Conclusion

In this task, we:

- Implemented Bubble Sort in Python.
- Used AI assistance to:
    - Add meaningful inline comments
    - Optimize using early termination
    - Analyze time complexity
- Evaluated when the algorithm is appropriate.

Bubble Sort is a simple yet educational algorithm that works well for small datasets like internal exam score ranking, but for larger systems, more efficient algorithms such as Merge Sort or Quick Sort would be preferable.

Task 2: Improving Sorting for Nearly Sorted Attendance Records

Scenario

You are maintaining an attendance system where student roll numbers

are already almost sorted, with only a few late updates.

Task Description

• Start with a Bubble Sort implementation.

Ask AI to:

- o Review the problem and suggest a more suitable sorting algorithm
- o Generate an Insertion Sort implementation
- o Explain why Insertion Sort performs better on nearly sorted data

• Compare execution behavior on nearly sorted input

Expected Outcome

• Two sorting implementations:

- o Bubble Sort
- o Insertion Sort

• AI-assisted explanation highlighting efficiency differences for

partially sorted datasets

AI Prompt

#I have an attendance system where roll numbers are already nearly sorted, with only a few misplaced entries.

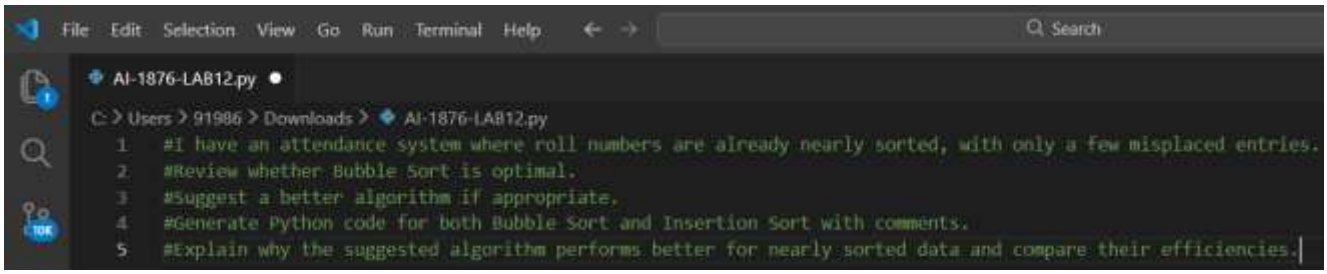
#Review whether Bubble Sort is optimal.

#Suggest a better algorithm if appropriate.

#Generate Python code for both Bubble Sort and Insertion Sort with comments.

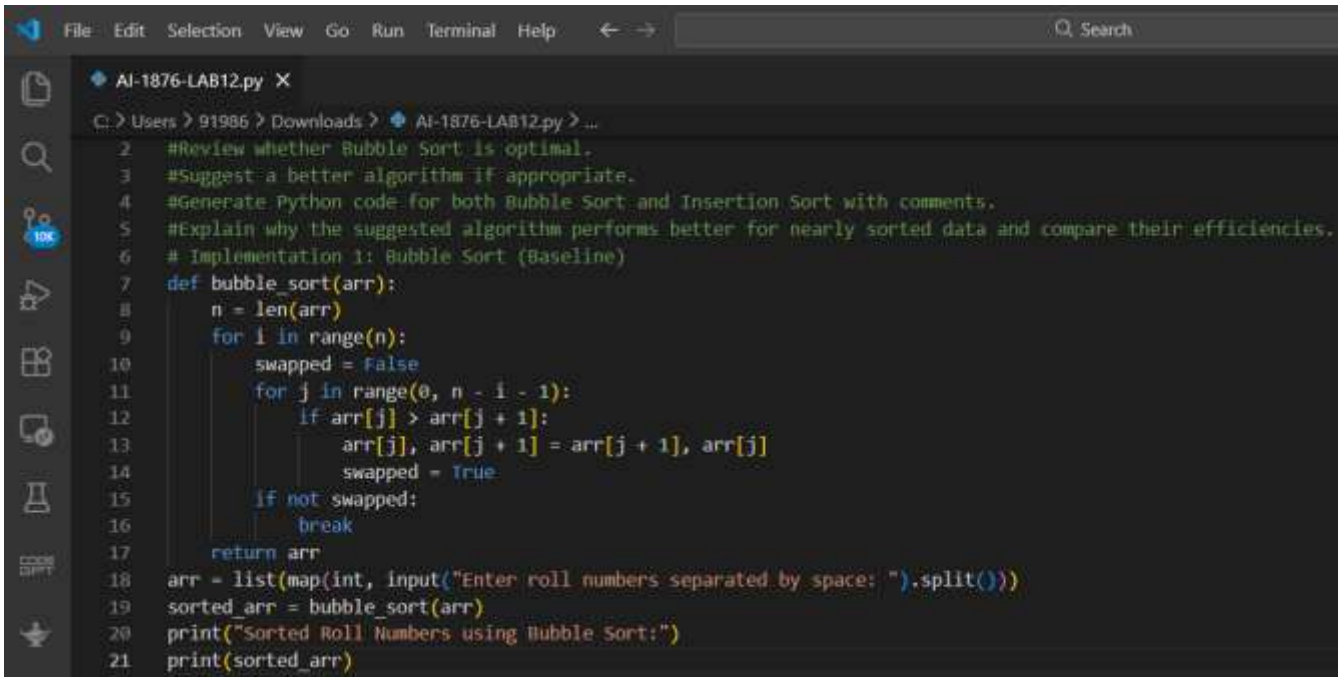
#Explain why the suggested algorithm performs better for nearly sorted data and compare their efficiencies.

Prompt Screenshot



Implementation 1: Bubble Sort (Baseline)

CODE



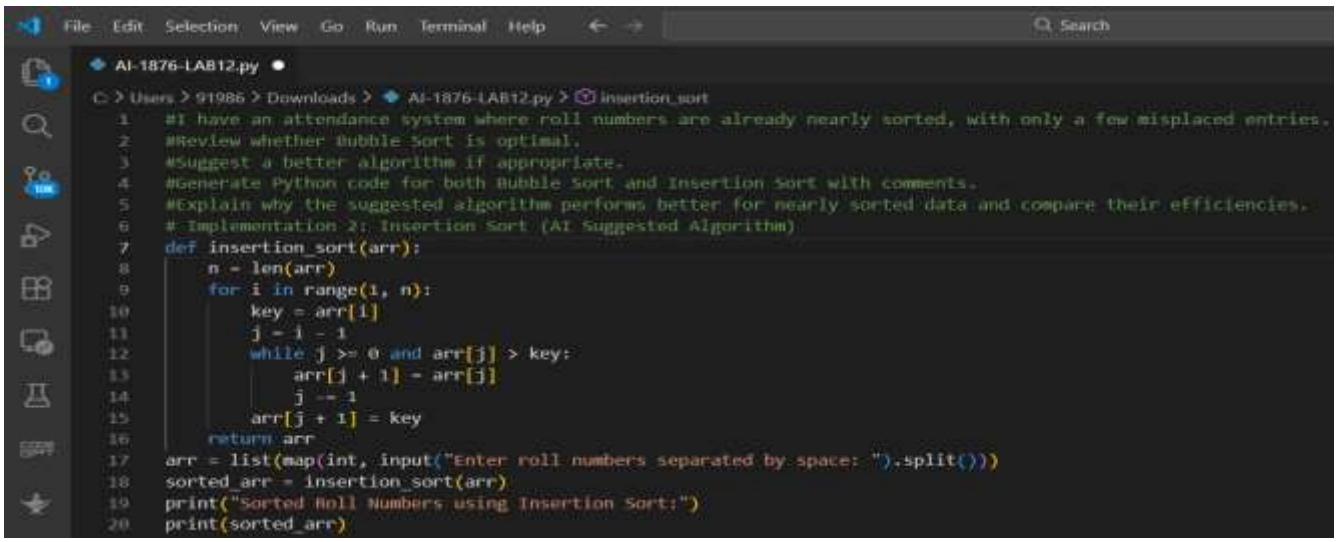
```python
#Review whether Bubble Sort is optimal.
#Suggest a better algorithm if appropriate.
#Generate Python code for both Bubble Sort and Insertion Sort with comments.
#Explain why the suggested algorithm performs better for nearly sorted data and compare their efficiencies.
# Implementation 1: Bubble Sort (Baseline)
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
arr = list(map(int, input("Enter roll numbers separated by space: ").split()))
sorted_arr = bubble_sort(arr)
print("Sorted Roll Numbers using Bubble Sort:")
print(sorted_arr)
```

OUTPUT

```
PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/Downloads/AI-1876-LAB12.py
Enter roll numbers separated by space: 101 102 103 105 104 106
Sorted Roll Numbers using Bubble Sort:
[101, 102, 103, 104, 105, 106]
```

Implementation 2: Insertion Sort (AI Suggested Algorithm)

CODE



```python
#I have an attendance system where roll numbers are already nearly sorted, with only a few misplaced entries.
#Review whether Bubble Sort is optimal.
#Suggest a better algorithm if appropriate.
#Generate Python code for both Bubble Sort and Insertion Sort with comments.
#Explain why the suggested algorithm performs better for nearly sorted data and compare their efficiencies.
# Implementation 2: Insertion Sort (AI Suggested Algorithm)
def insertion_sort(arr):
    n = len(arr)
    for i in range(1, n):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
arr = list(map(int, input("Enter roll numbers separated by space: ").split()))
sorted_arr = insertion_sort(arr)
print("Sorted Roll Numbers using Insertion Sort:")
print(sorted_arr)
```

OUTPUT

Differences

Bubble Sort	Insertion Sort
Repeatedly compares adjacent elements	Inserts element into correct position
Multiple full passes	Only shifts misplaced elements
Less efficient for nearly sorted data	Very efficient for nearly sorted data

Observation

Both Bubble Sort and Insertion Sort successfully sorted the nearly sorted attendance roll numbers. However, Bubble Sort performed more comparisons across multiple passes. In contrast, Insertion Sort made only the necessary shifts to correctly place the misplaced element, resulting in fewer operations.

Explanation

Bubble Sort repeatedly compares adjacent elements and may require multiple passes even if only one element is out of place. Insertion Sort works by inserting each element into its correct position within the already sorted portion. When the data is nearly sorted, very few shifts are required, making Insertion Sort more efficient in practice.

Justification

Attendance records are typically entered in sequential order, with only a few late updates. Since the dataset is almost sorted, Insertion Sort reduces unnecessary comparisons and performs closer to linear time. Therefore, it is more suitable for this scenario than Bubble Sort.

Conclusion

For nearly sorted datasets like attendance records, Insertion Sort performs better than Bubble Sort. The AI suggestion to use Insertion Sort is appropriate because it improves efficiency while maintaining simplicity.

Task 3: Searching Student Records in a Database

Scenario

You are developing a student information portal where users search for student records by roll number.

Task Description

- **Implement:**
    - o Linear Search for unsorted student data
    - o Binary Search for sorted student data
- **Use AI to:**
    - o Add docstrings explaining parameters and return values
    - o Explain when Binary Search is applicable
    - o Highlight performance differences between the two searches

Expected Outcome

- Two working search implementations with docstrings
- **AI-generated explanation of:**
    - o Time complexity
    - o Use cases for Linear vs Binary Search
- A short student observation comparing results on sorted vs

unsorted lists

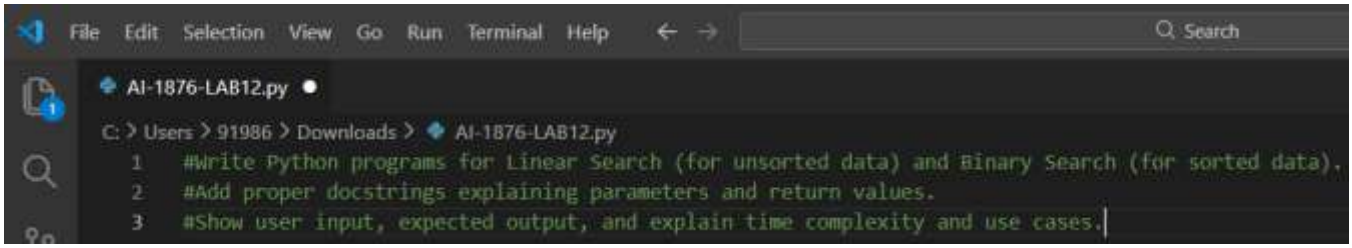
AI Prompt

#Write Python programs for Linear Search (for unsorted data) and Binary Search (for sorted data).

#Add proper docstrings explaining parameters and return values.

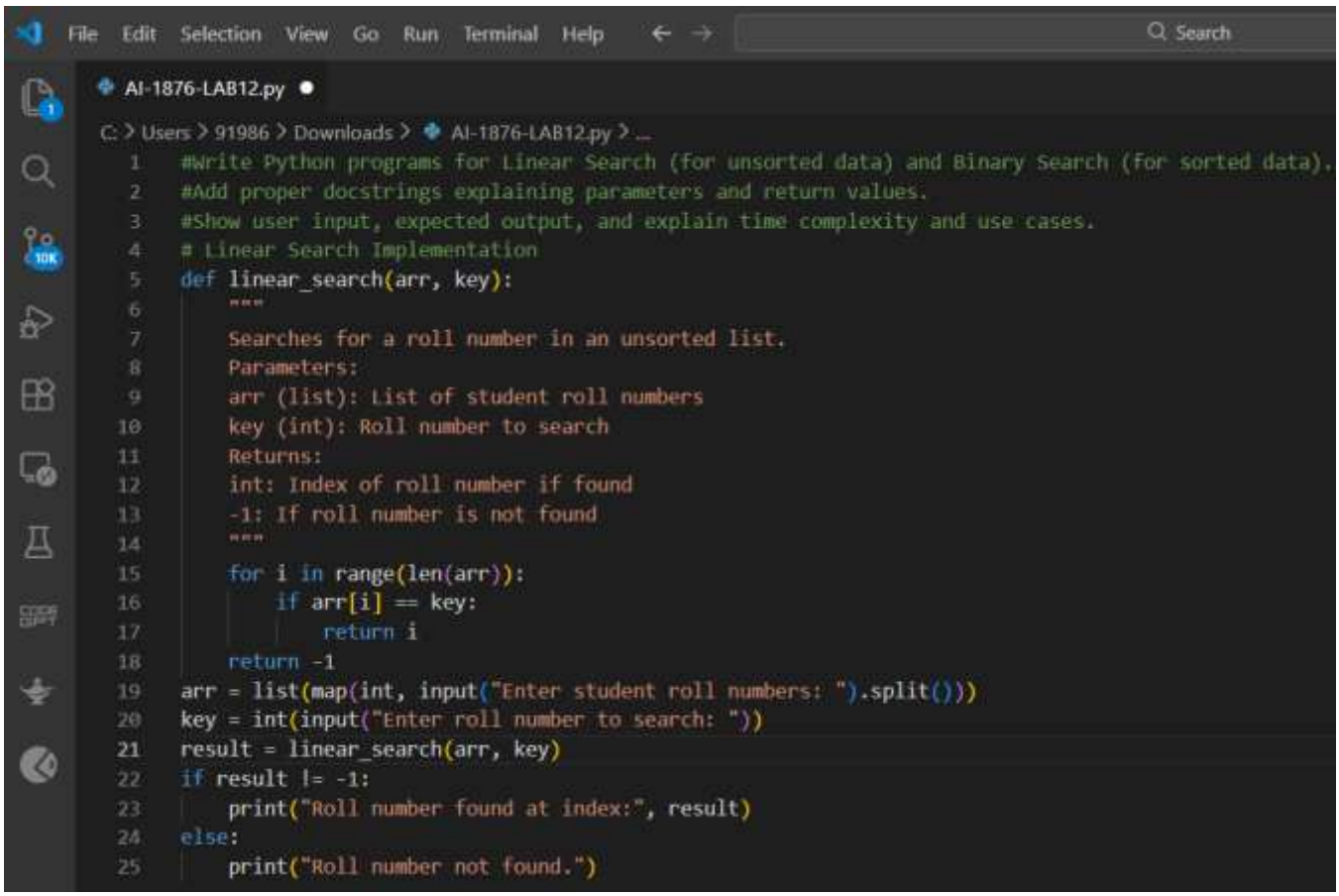
#Show user input, expected output, and explain time complexity and use cases.

Prompt Screenshot



Implementation 1: Linear Search (Unsorted Data)

CODE



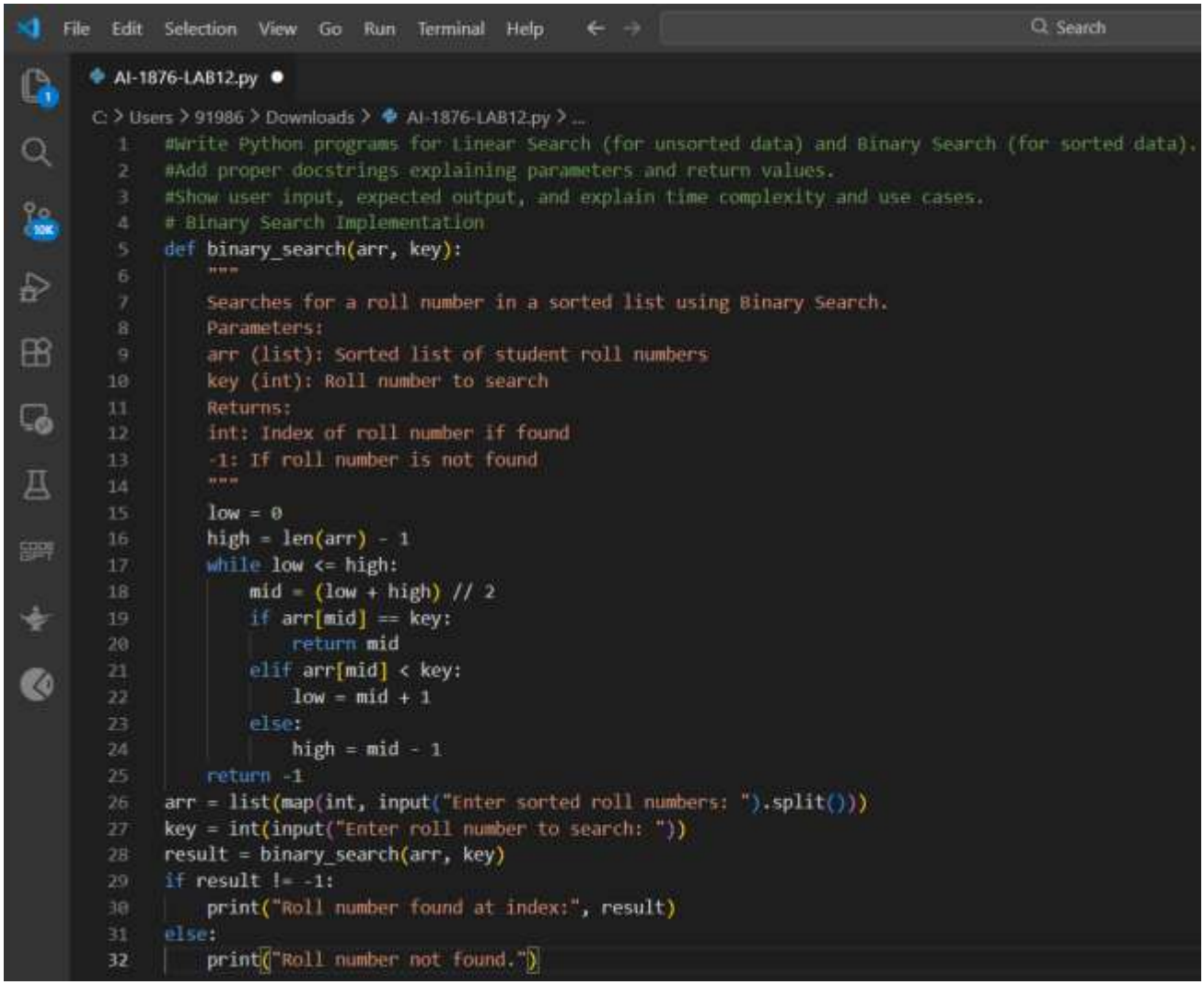
OUTPUT

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  POSTMAN CONSOLE

PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/Downloads/AI-1876-LAB12.py
Enter student roll numbers: 105 101 108 102 110
Enter roll number to search: 102
Roll number found at index: 3
```

Implementation 2: Binary Search (Sorted Data)

CODE



```python
#Write Python programs for Linear Search (for unsorted data) and Binary Search (for sorted data).
#Add proper docstrings explaining parameters and return values.
#Show user input, expected output, and explain time complexity and use cases.
# Binary Search Implementation
def binary_search(arr, key):
    """
    Searches for a roll number in a sorted list using Binary Search.
    Parameters:
    arr (list): Sorted list of student roll numbers
    key (int): Roll number to search
    Returns:
    int: Index of roll number if found
    -1: If roll number is not found
    """
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == key:
            return mid
        elif arr[mid] < key:
            low = mid + 1
        else:
            high = mid - 1
    return -1
arr = list(map(int, input("Enter sorted roll numbers: ").split()))
key = int(input("Enter roll number to search: "))
result = binary_search(arr, key)
if result != -1:
    print("Roll number found at index:", result)
else:
    print("Roll number not found.")
```

OUTPUT

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  POSTMAN CONSOLE

PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/Downloads/AI-1876-LAB12.py
Enter sorted roll numbers: 101 102 105 108 110
Enter roll number to search: 108
Roll number found at index: 3
```

Observation

- Linear Search works for both sorted and unsorted data.
- Binary Search works only when the list is sorted.
- Binary Search finds the element faster in large datasets.
- For small lists, the performance difference is less noticeable.

Explanation

Linear Search checks each element one by one until the target is found.
Binary Search divides the list into halves and eliminates half of the elements in each step.

Time Complexity:

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

Binary Search is more efficient because it reduces the search space quickly.

Justification

In a student portal:

- If records are unsorted → Linear Search is required.
- If records are stored in sorted order → Binary Search is more efficient.
- Large databases benefit significantly from Binary Search.

Thus, algorithm choice depends on data structure and size.

Conclusion

Linear Search is simple and works on any dataset.
Binary Search is faster but requires sorted data.
For large, sorted student databases, Binary Search is the better choice.
AI correctly highlights the importance of choosing the right search method based on data conditions.

Task 4: Choosing Between Quick Sort and Merge Sort for Data Processing

Scenario

You are part of a data analytics team that needs to sort large datasets

received from different sources (random order, already sorted, and reverse sorted).

Task Description

- Provide AI with partially written recursive functions for:
    - o Quick Sort
    - o Merge Sort
- **Ask AI to:**
    - o Complete the recursive logic
    - o Add meaningful docstrings
    - o Explain how recursion works in each algorithm
- **Test both algorithms on:**
    - o Random data
    - o Sorted data
    - o Reverse-sorted data

Expected Outcome

- Fully functional Quick Sort and Merge Sort implementations
- **AI-generated comparison covering:**
    - o Best, average, and worst-case complexities
    - o Practical scenarios where one algorithm is preferred over the other

AI Prompt

#Complete the recursive implementations of Quick Sort and Merge Sort in Python.

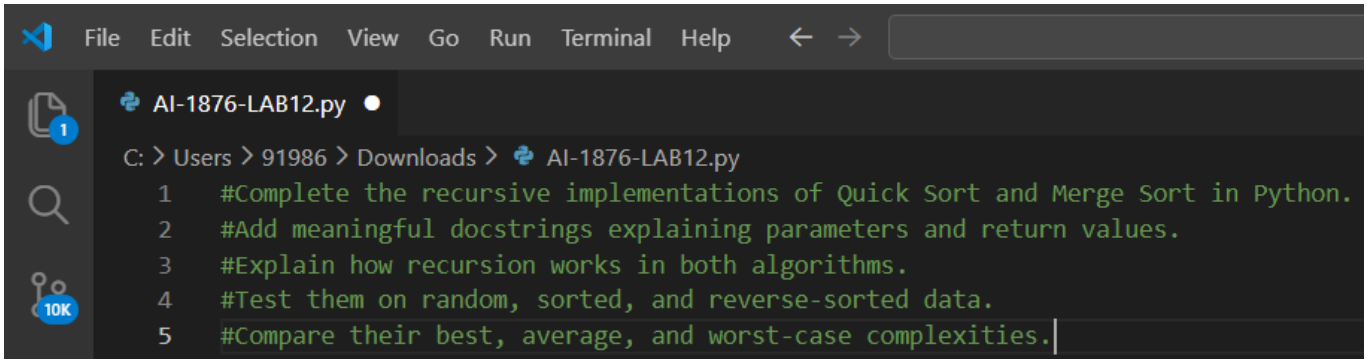
#Add meaningful docstrings explaining parameters and return values.

#Explain how recursion works in both algorithms.

#Test them on random, sorted, and reverse-sorted data.

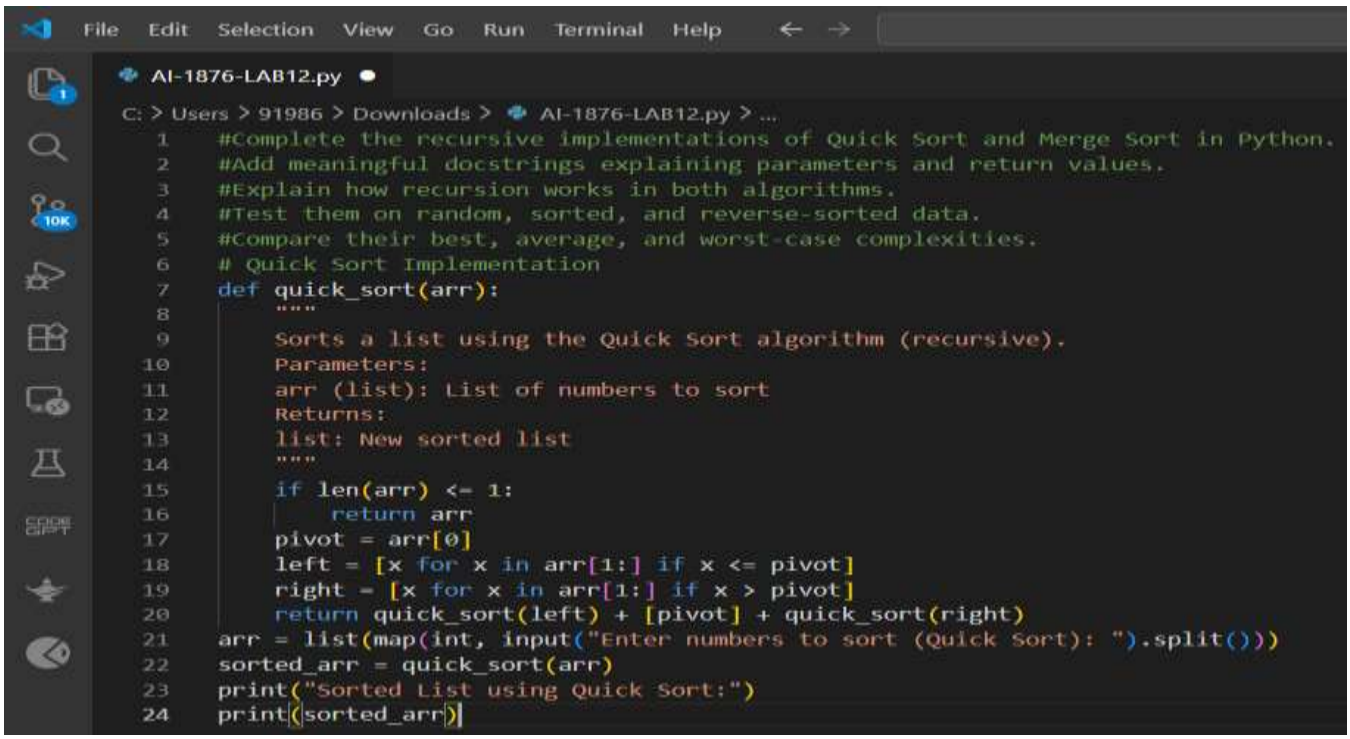
#Compare their best, average, and worst-case complexities.

Prompt Screenshot



Implementation 1: Quick Sort

CODE



OUTPUT

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  POSTMAN CONSOLE

PS C:\Users\91986> & "C:/Program Files/Python313/python.exe" c:/Users/91986/Downloads/AI-1876-LAB12.py
Enter numbers to sort (Quick Sort): 8 3 7 4 9 2 6
Sorted List using Quick Sort:
[2, 3, 4, 6, 7, 8, 9]
```

Implementation 2: Merge Sort

CODE

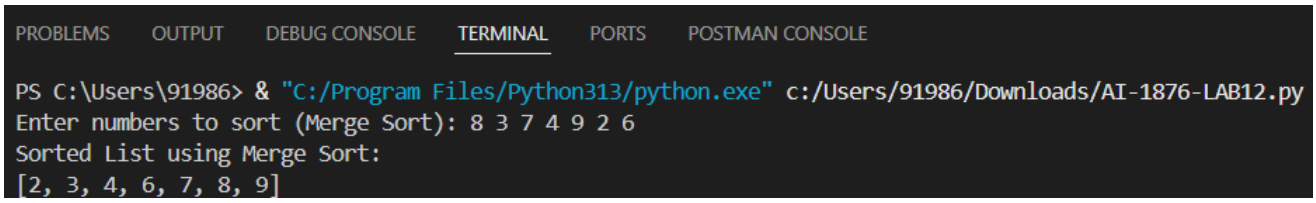
File Edit Selection View Go Run Terminal Help

AI-1876-LAB12.py

C: > Users > 91986 > Downloads > AI-1876-LAB12.py > ...

```python
#Complete the recursive implementations of Quick Sort and Merge Sort in Python.
#Add meaningful docstrings explaining parameters and return values.
#Explain how recursion works in both algorithms.
#Test them on random, sorted, and reverse-sorted data.
#Compare their best, average, and worst-case complexities.
# Merge Sort Implementation
def merge_sort(arr):
    """
    Sorts a list using the Merge Sort algorithm (recursive).
    Parameters:
    arr (list): List of numbers to sort
    Returns:
    list: New sorted list
    """
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])
    return merge(left_half, right_half)
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
arr = list(map(int, input("Enter numbers to sort (Merge Sort): ").split()))
sorted_arr = merge_sort(arr)
print("Sorted List using Merge Sort:")
print(sorted_arr)
```

OUTPUT



How Recursion Works

Quick Sort:

- Selects a pivot.
- Divides list into smaller sublists (left & right).
- Recursively sorts sublists.
- Combines results.

Merge Sort:

- Divides list into halves repeatedly.
- Recursively sorts each half.
- Merges sorted halves back together.

Recursion continues until the base case (list size ≤ 1).

Time Complexity Comparison

Algorithm	Best Case	Average Case	Worst Case
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)

Space Complexity:

- Quick Sort → O(log n) (recursive stack)
- Merge Sort → O(n) (extra array for merging)

Observation

- Both algorithms sort correctly.
- Quick Sort performs very fast on random data.

- Quick Sort performs poorly on already sorted data (if pivot is first element).
- Merge Sort performs consistently in all cases.
- Merge Sort uses extra memory for merging.

Explanation

Quick Sort efficiency depends on pivot choice.
If pivot divides data evenly → O(n log n).
If pivot is smallest/largest repeatedly → $O(n^2)$.

Merge Sort always divides list into equal halves and merges them.
Therefore, it guarantees O(n log n) performance.

Justification

For large datasets:

- If memory usage is limited → Quick Sort preferred.
- If stable and guaranteed performance is required → Merge Sort preferred.
- For already sorted or reverse-sorted data → Merge Sort is safer.

Algorithm choice depends on:

- Data order
- Memory availability
- Performance guarantees required

Conclusion

Quick Sort is generally faster in practice but can degrade in worst-case scenarios.
Merge Sort provides consistent O(n log n) performance regardless of input order.

For large data processing systems:

- Use Quick Sort for average performance efficiency.
- Use Merge Sort when predictable performance is required.

AI correctly highlights that algorithm selection should depend on dataset characteristics and system requirements.

Task 5: Optimizing a Duplicate Detection Algorithm

Scenario

You are building a data validation module that must detect duplicate

user IDs in a large dataset before importing it into a system.

Task Description

• Write a naive duplicate detection algorithm using nested loops.

• **Use AI to:**

  o Analyze the time complexity

  o Suggest an optimized approach using sets or dictionaries

  o Rewrite the algorithm with improved efficiency

• Compare execution behavior conceptually for large input sizes

Expected Outcome

• Two versions of the algorithm:

  o Brute-force ($O(n^2)$)

  o Optimized ($O(n)$)

• AI-assisted explanation showing how and why performance improved

AI Prompt

#Write a Python program to detect duplicate user IDs using a brute-force nested loop approach.

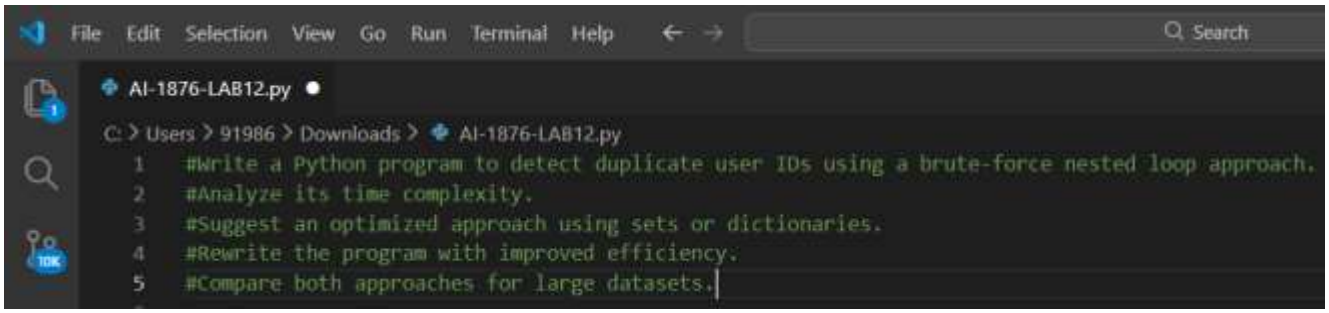
#Analyze its time complexity.

#Suggest an optimized approach using sets or dictionaries.

#Rewrite the program with improved efficiency.

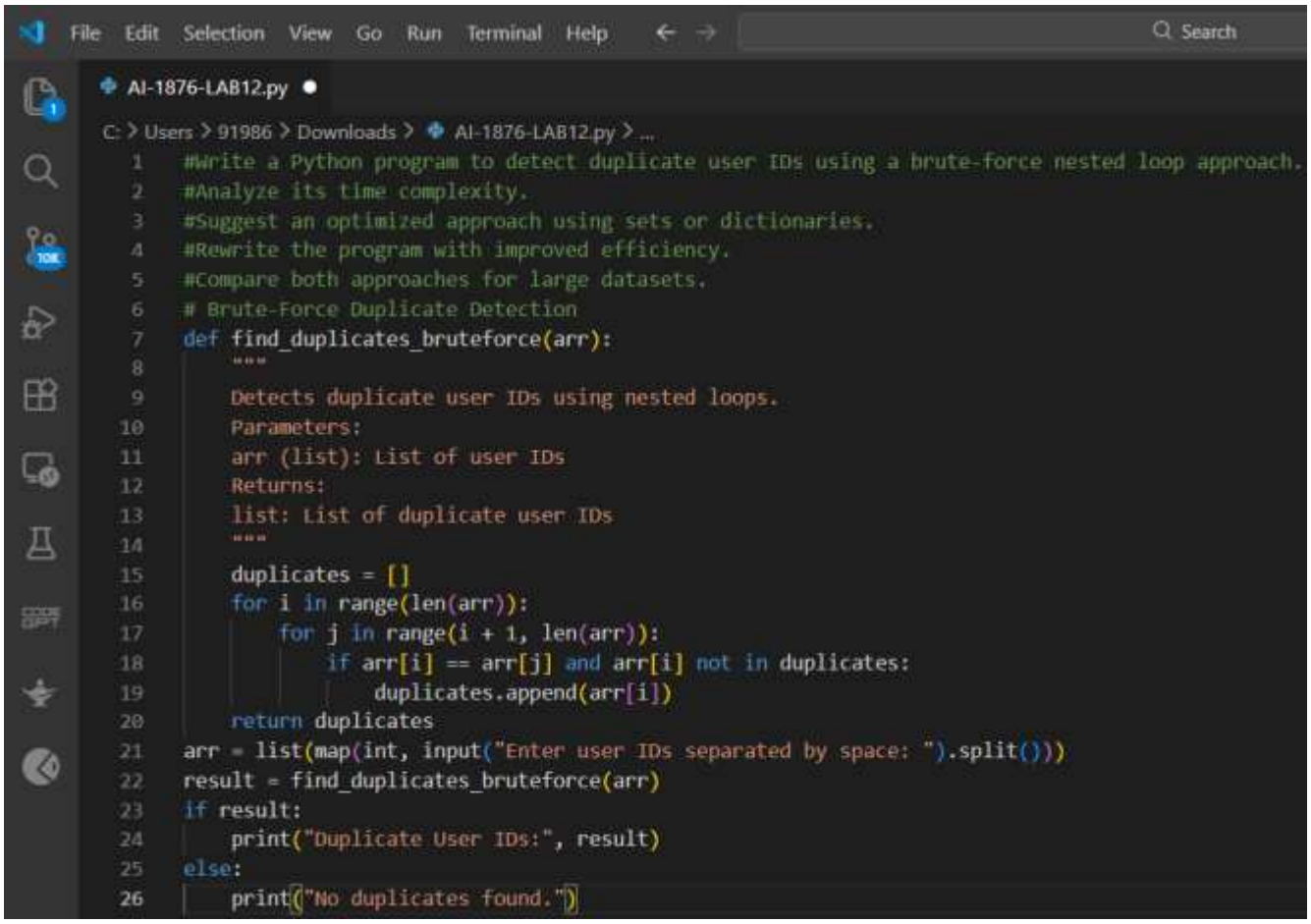
#Compare both approaches for large datasets.

Prompt Screenshot

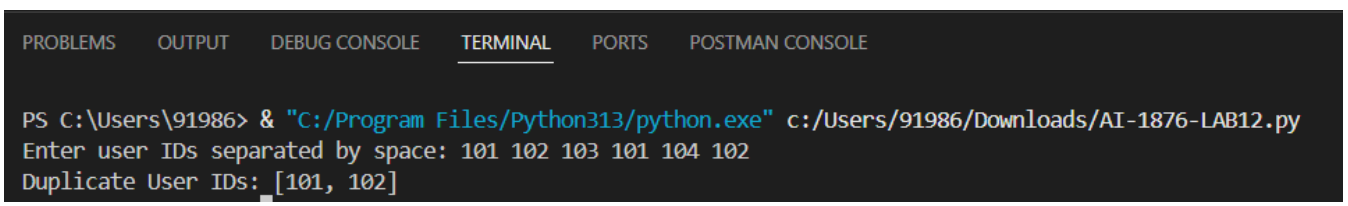


Version 1: Brute-Force Duplicate Detection ($O(n^2)$)

CODE



OUTPUT

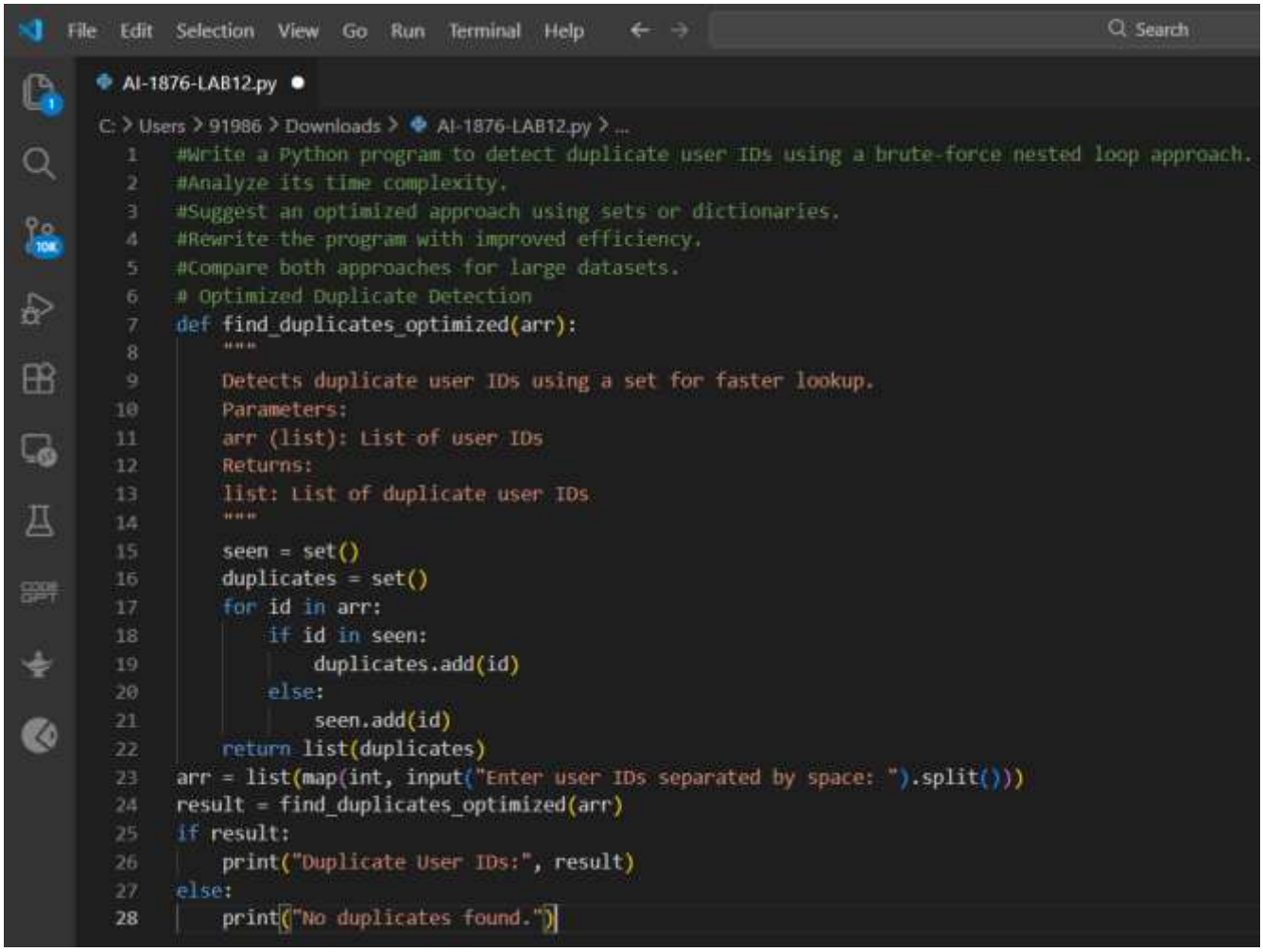


Time Complexity (Brute-Force)

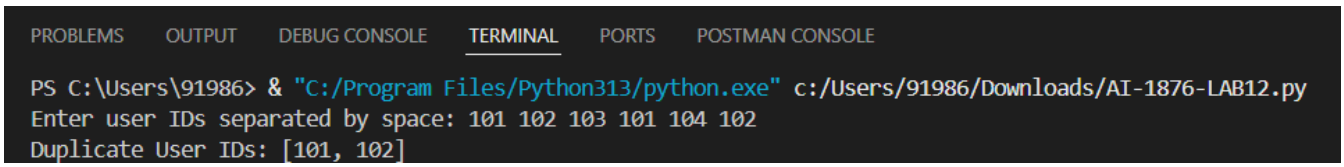
- Two nested loops → $O(n^2)$
- For large datasets (e.g., 100,000 IDs), comparisons become very slow.

Version 2: Optimized Duplicate Detection Using Set (O(n))

CODE



OUTPUT



Conceptual Performance Comparison (Large Input)

Approach	Time Complexity	Behavior on Large Data
Brute-force	$O(n^2)$	Very slow for large datasets
Optimized (Set)	O(n)	Fast and scalable

Example:

- 10,000 IDs
    - Brute-force → ~100 million comparisons
    - Optimized → ~10,000 operations

Huge performance improvement.

Observation

- Both methods correctly detect duplicate IDs.
- Brute-force method becomes slow as dataset size increases.
- Optimized method performs significantly faster for large inputs.
- Set-based approach avoids repeated comparisons.

Explanation

Brute-force compares each element with every other element, leading to quadratic growth in operations.

The optimized method:

- Uses a set for constant-time lookup (O(1)).
- Checks each element only once.
- Reduces overall time complexity to O(n).

This improvement happens because sets use hashing for fast membership checking.

Justification

In real-world systems:

- Datasets may contain thousands or millions of user IDs.
- $O(n^2)$ algorithms are not practical for large-scale systems.
- O(n) solutions are scalable and efficient.

Therefore, the optimized set-based approach is more suitable for data validation modules.

Conclusion

The brute-force duplicate detection algorithm works but is inefficient for large datasets.
The optimized version using sets reduces time complexity from $O(n^2)$ to O(n).

AI-assisted optimization significantly improved performance and scalability.
For large systems, using appropriate data structures like sets or dictionaries is essential.