ASSIGNMENT-12.4

2303A510E3

BATCH-14

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

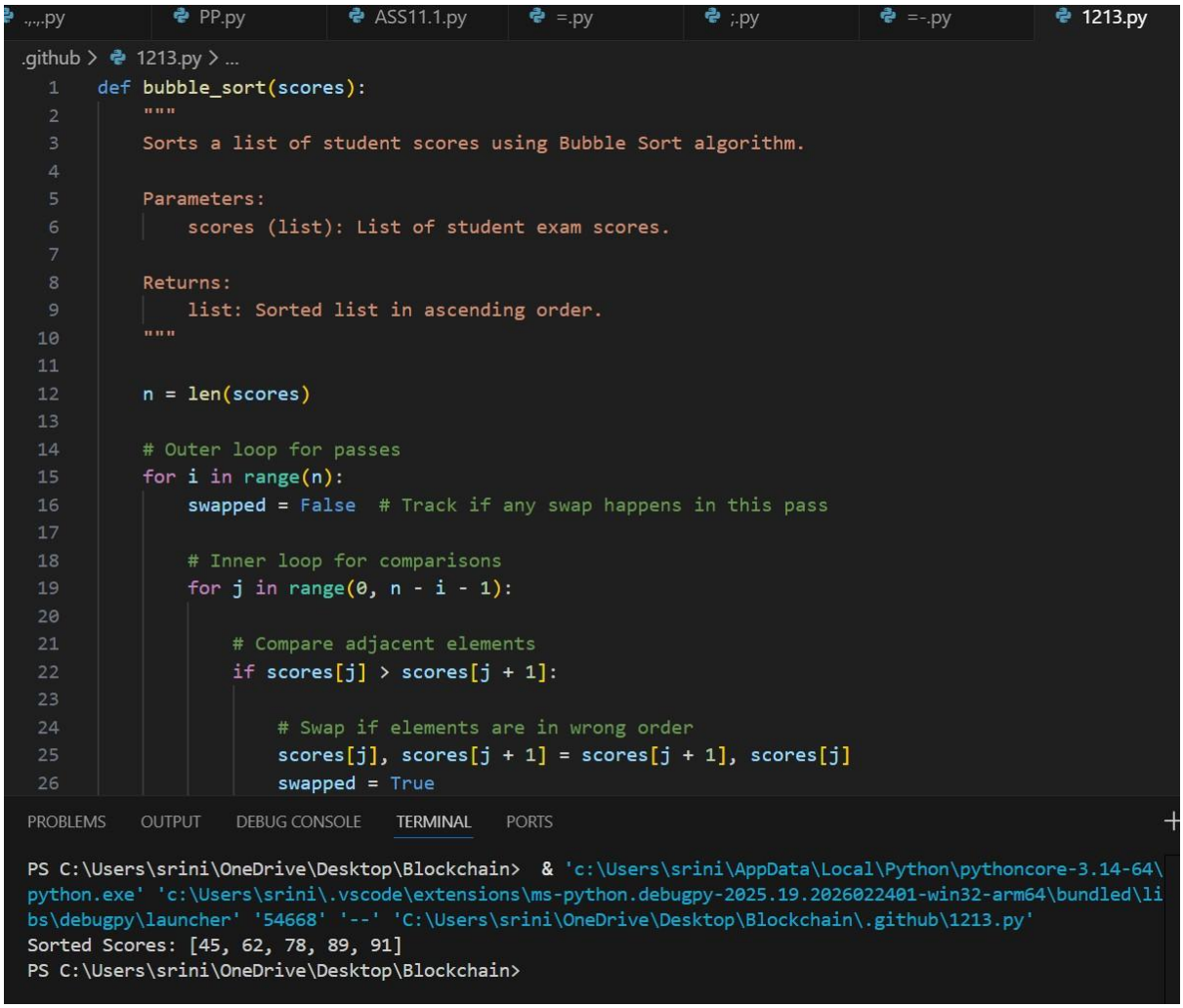
• A Bubble Sort implementation with:

o AI-generated comments explaining the logic

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

complexity

o Sample input/output showing sorted scores



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

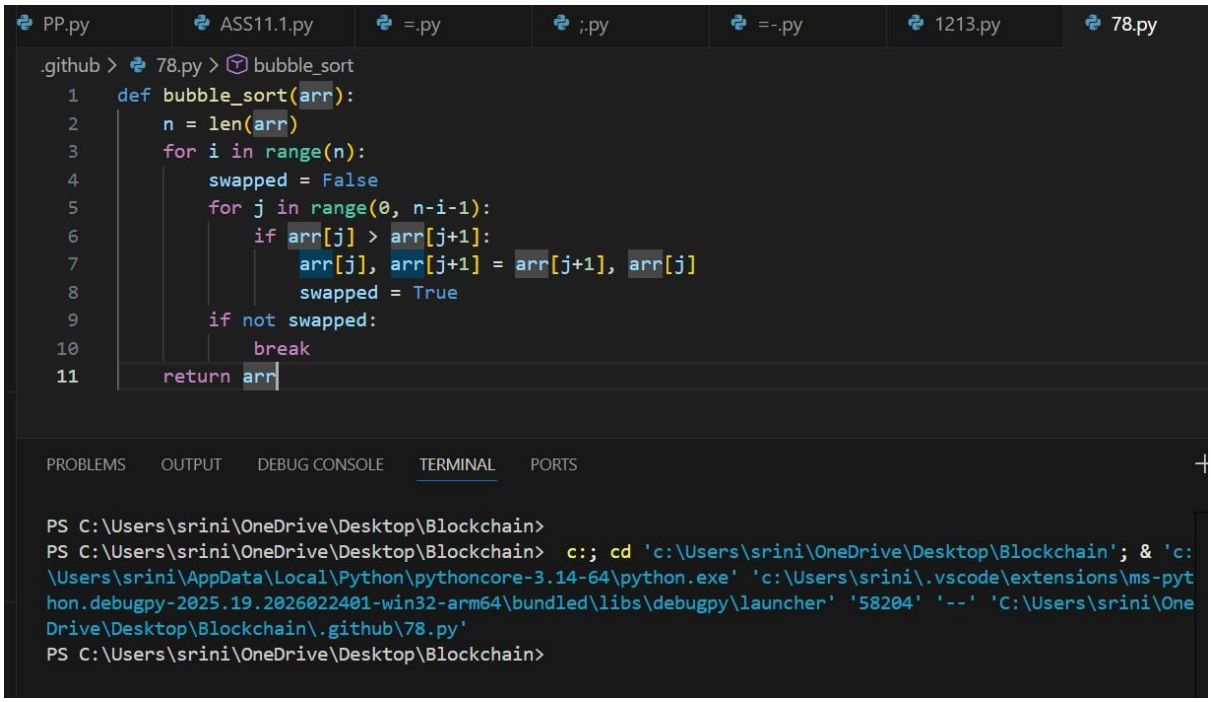
• Two sorting implementations:

o Bubble Sort

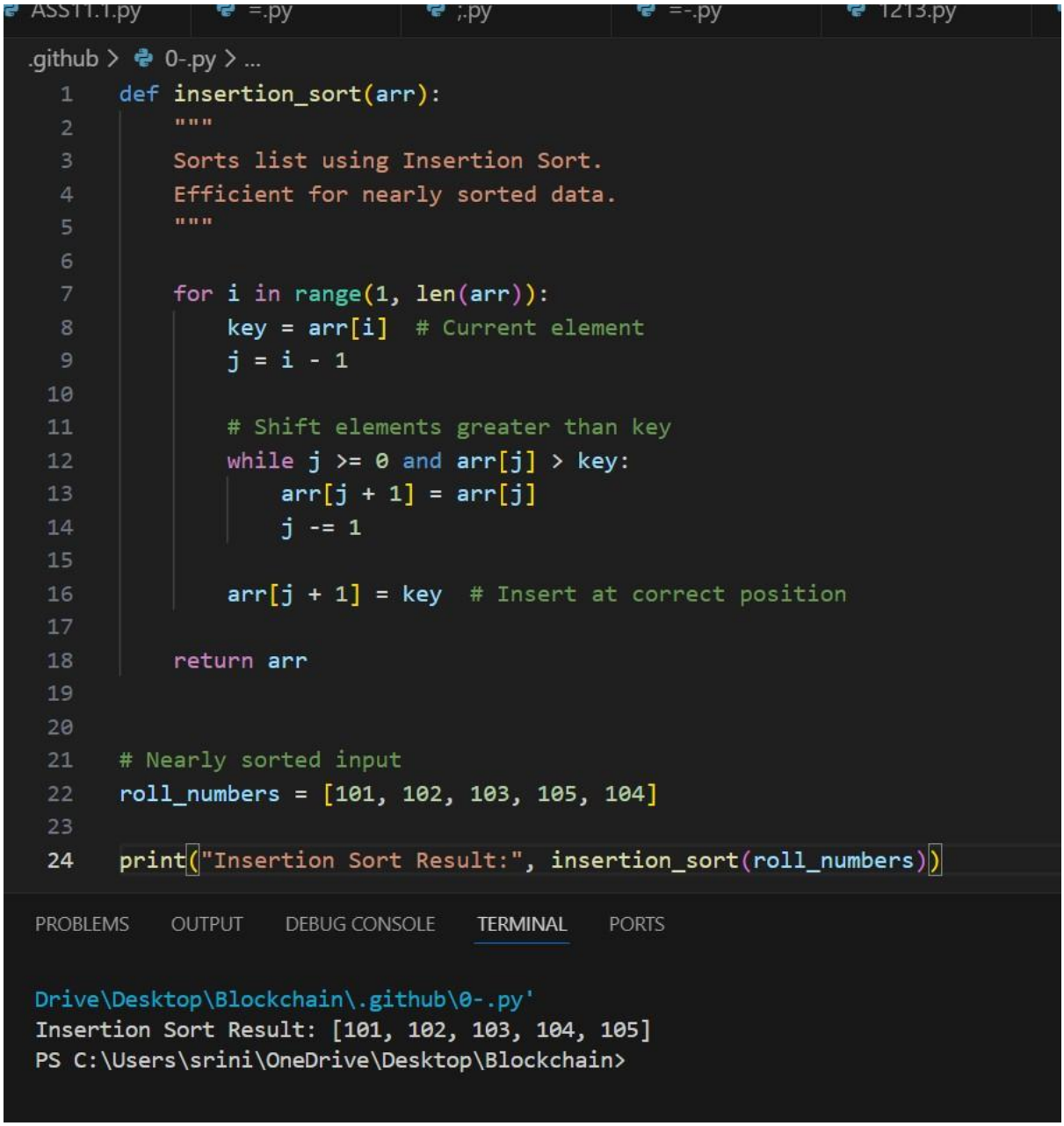
o Insertion Sort

• AI-assisted explanation highlighting efficiency differences for partially sorted datasets

Bubble Sort



Insertion Sort Implementation



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

Linear Search Implementation

.github > YU.py > linear_search

```python
def linear_search(data, target):
    """
    Searches for target roll number in an unsorted list.

    Parameters:
        data (list): List of student roll numbers.
        target (int): Roll number to search.

    Returns:
        int: Index of target if found, otherwise -1.
    """

    for i in range(len(data)):
        if data[i] == target:
            return i
    return -1
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\srini\OneDrive\Desktop\Blockchain> ^C
PS C:\Users\srini\OneDrive\Desktop\Blockchain>
PS C:\Users\srini\OneDrive\Desktop\Blockchain>  c:; cd 'c:\Users\srini\OneDrive\Desktop\Blockchain'; & 'c:
\Users\srini\AppData\Local\Python\pythoncore-3.14-64\python.exe' 'c:\Users\srini\.vscode\extensions\ms-pyt
hon.debugpy-2025.19.2026022401-win32-arm64\bundled\libs\debugpy\launcher' '63196' '--' 'C:\Users\srini\One
Drive\Desktop\Blockchain\.github\YU.py'
PS C:\Users\srini\OneDrive\Desktop\Blockchain>
```

.github > BH.py > binary_search

```python
def binary_search(data, target):
    """
    Searches for target roll number in a sorted list.

    Parameters:
        data (list): Sorted list of roll numbers.
        target (int): Roll number to search.

    Returns:
        int: Index if found, else -1.
    """

    left = 0
    right = len(data) - 1

    while left <= right:
        mid = (left + right) // 2

        if data[mid] == target:
            return mid
        elif data[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\srini\OneDrive\Desktop\Blockchain> ^C
PS C:\Users\srini\OneDrive\Desktop\Blockchain>
PS C:\Users\srini\OneDrive\Desktop\Blockchain>  c:; cd 'c:\Users\srini\OneDrive\Desktop\Blockchain'; & 'c:
\Users\srini\AppData\Local\Python\pythoncore-3.14-64\python.exe' 'c:\Users\srini\.vscode\extensions\ms-pyt
hon.debugpy-2025.19.2026022401-win32-arm64\bundled\libs\debugpy\launcher' '63959' '--' 'C:\Users\srini\One
Drive\Desktop\Blockchain\.github\BH.py'
PS C:\Users\srini\OneDrive\Desktop\Blockchain>
```

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

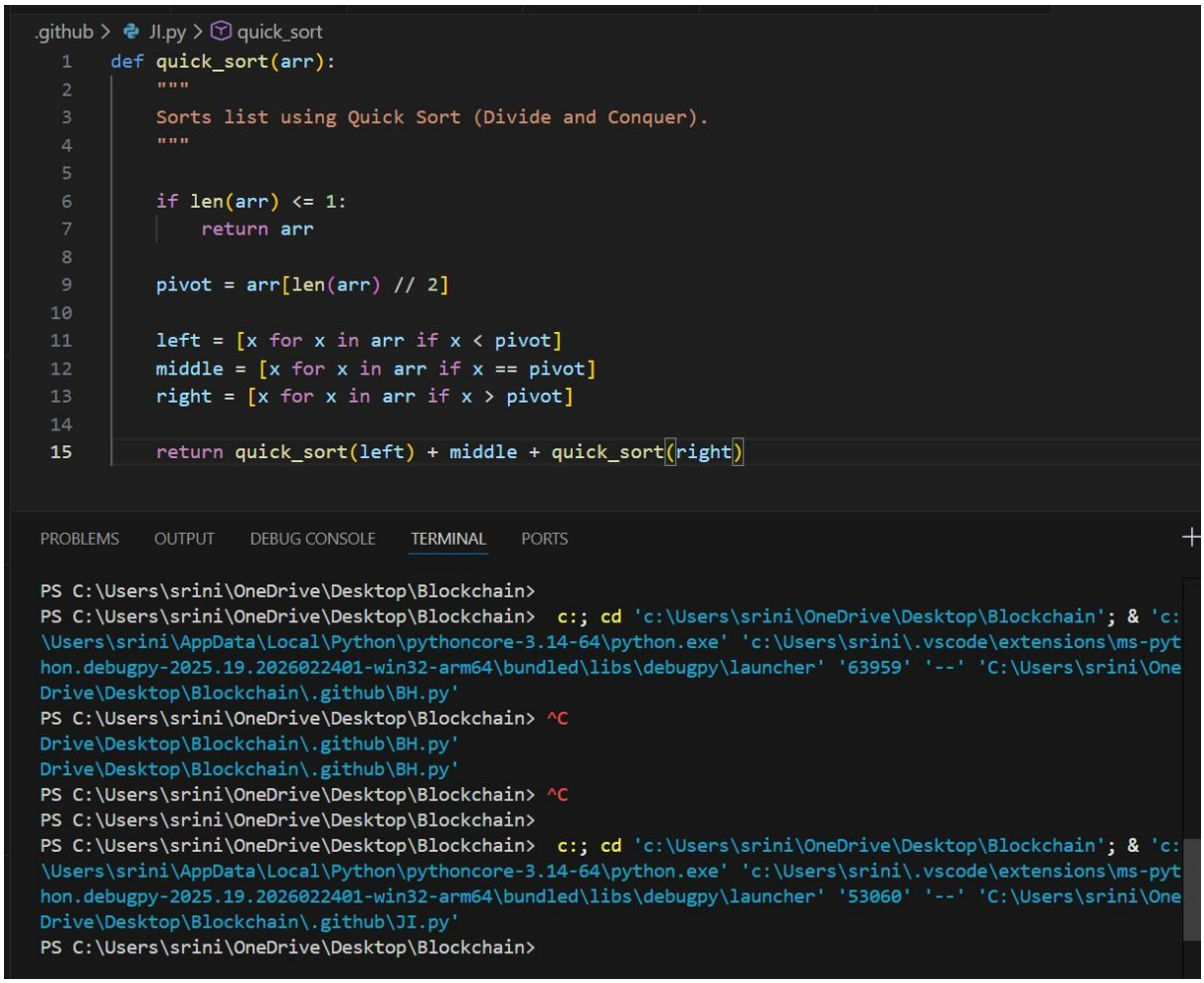
• Fully functional Quick Sort and Merge Sort implementations

• AI-generated comparison covering:

o Best, average, and worst-case complexities

o Practical scenarios where one algorithm is preferred over the other

Quick Sort Implementation

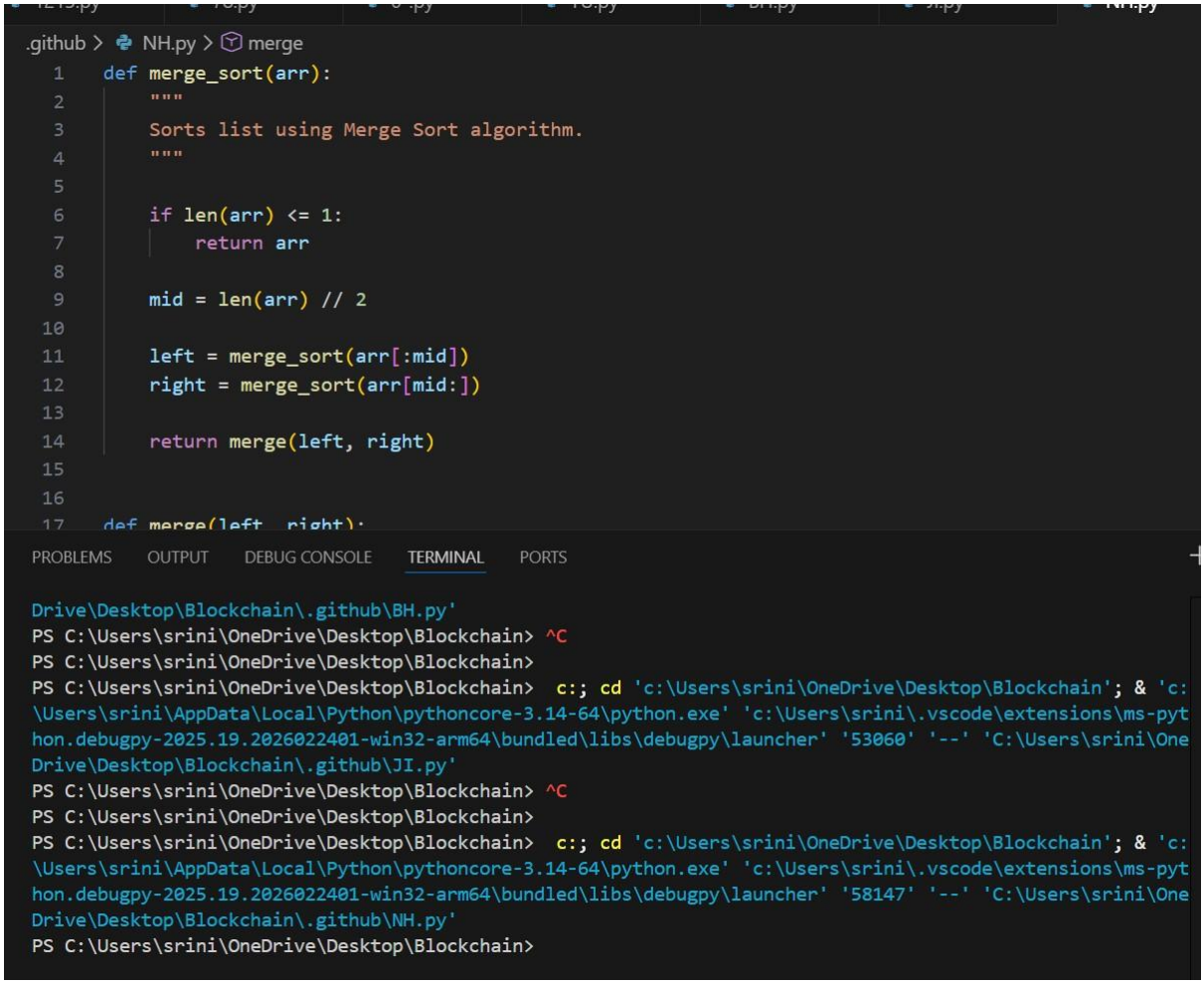


```
.github > JI.py > quick_sort
1   def quick_sort(arr):
2       """
3       Sorts list using Quick Sort (Divide and Conquer).
4       """
5
6       if len(arr) <= 1:
7           return arr
8
9       pivot = arr[len(arr) // 2]
10
11      left = [x for x in arr if x < pivot]
12      middle = [x for x in arr if x == pivot]
13      right = [x for x in arr if x > pivot]
14
15      return quick_sort(left) + middle + quick_sort(right)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\srini\OneDrive\Desktop\Blockchain>
PS C:\Users\srini\OneDrive\Desktop\Blockchain>  c:; cd 'c:\Users\srini\OneDrive\Desktop\Blockchain'; & 'c:
\Users\srini\AppData\Local\Python\pythoncore-3.14-64\python.exe' 'c:\Users\srini\.vscode\extensions\ms-pyt
hon.debugpy-2025.19.2026022401-win32-arm64\bundled\libs\debugpy\launcher' '63959' '--' 'C:\Users\srini\One
Drive\Desktop\Blockchain\.github\BH.py'
PS C:\Users\srini\OneDrive\Desktop\Blockchain> ^C
Drive\Desktop\Blockchain\.github\BH.py'
Drive\Desktop\Blockchain\.github\BH.py'
PS C:\Users\srini\OneDrive\Desktop\Blockchain> ^C
PS C:\Users\srini\OneDrive\Desktop\Blockchain>
PS C:\Users\srini\OneDrive\Desktop\Blockchain>  c:; cd 'c:\Users\srini\OneDrive\Desktop\Blockchain'; & 'c:
\Users\srini\AppData\Local\Python\pythoncore-3.14-64\python.exe' 'c:\Users\srini\.vscode\extensions\ms-pyt
hon.debugpy-2025.19.2026022401-win32-arm64\bundled\libs\debugpy\launcher' '53060' '--' 'C:\Users\srini\One
Drive\Desktop\Blockchain\.github\JI.py'
PS C:\Users\srini\OneDrive\Desktop\Blockchain>
```

Merge Sort Implementation



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

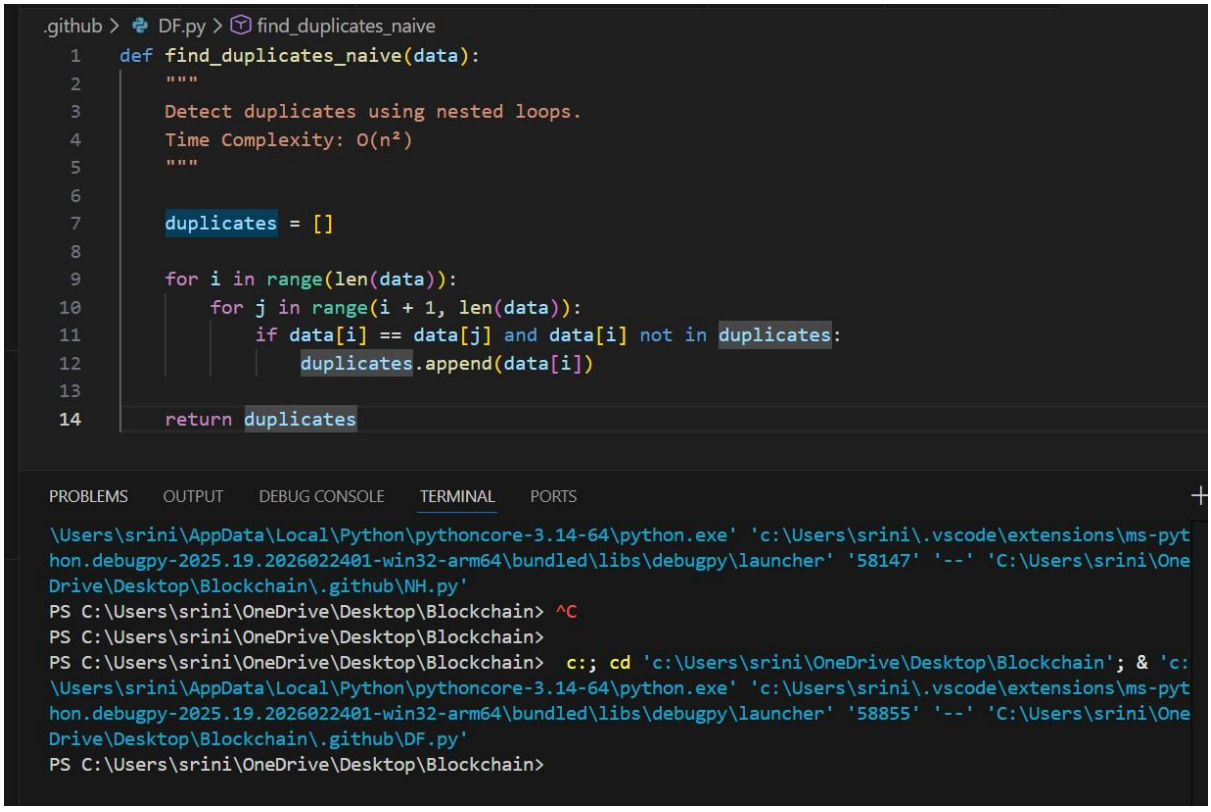
• Two versions of the algorithm:

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

o Optimized ($O(n)$)

• AI-assisted explanation showing how and why performance Improved

Naive Approach

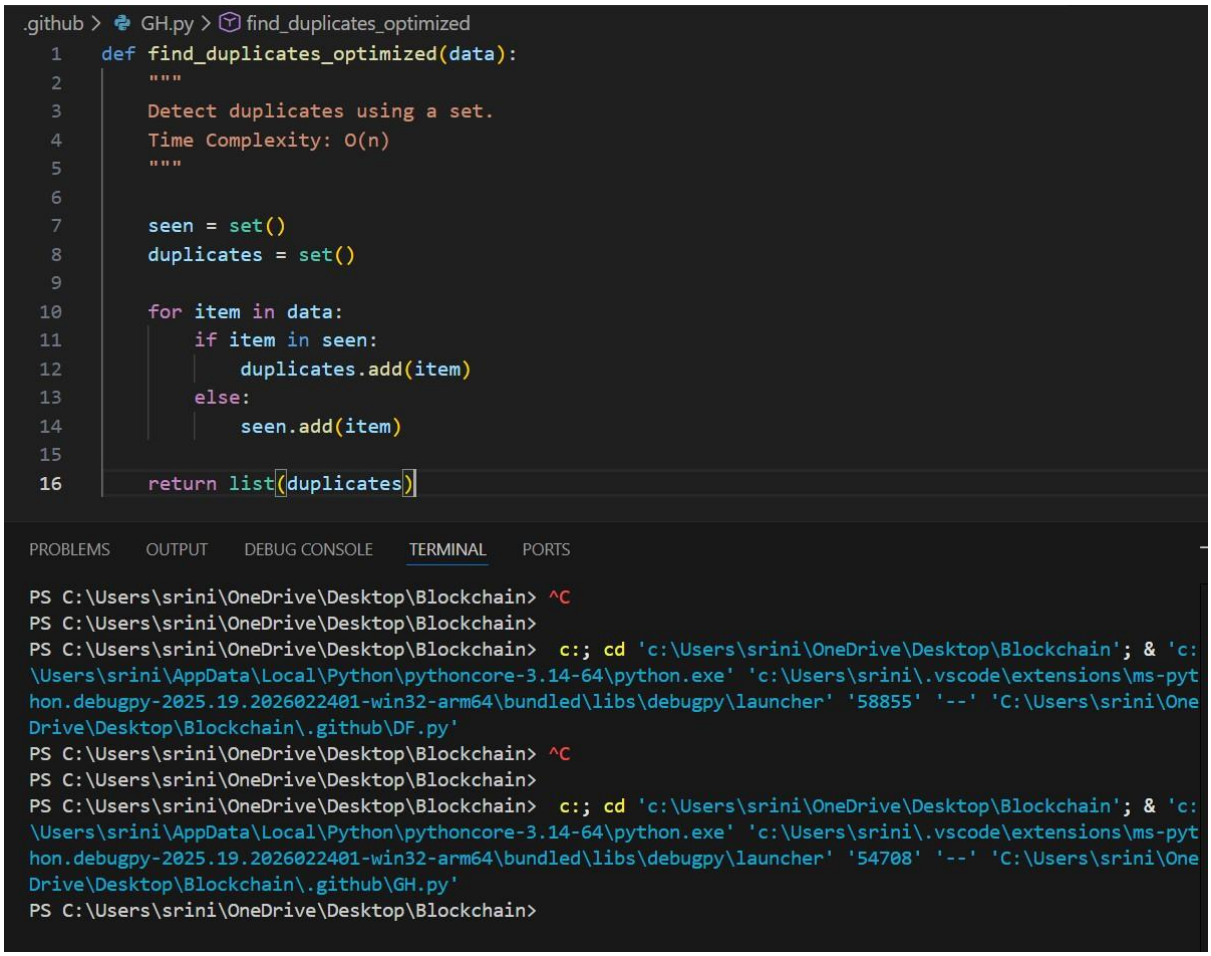


Complexity Analysis

- Two nested loops → $O(n^2)$

- Very slow for large datasets

Optimized Approach Using Set



Why Performance Improved?

- Set lookup is O(1).
- Eliminated nested loops.
- Single traversal of dataset.

- **Critical Evaluation of AI Suggestions**
- ✓ AI correctly suggested Insertion Sort for nearly sorted data
  ✓ AI improved Bubble Sort using early termination
  ✓ AI recommended set-based optimization for duplicate detection
  ✓ AI provided correct complexity analysis