ASSIGNMENT – 12.3

NAME:CH.Charan Kumar

Roll.no:2303A54006

BATCH:47-A

Lab 12: Algorithms with AI Assistance Sorting, Searching, and Algorithm Optimization Using AI Tools

Lab Objectives

The objectives of this laboratory exercise are to:

Week3 -

Wednesday

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

• Analyze and compare algorithm efficiency using time and space complexity.

• Understand how AI tools can suggest optimizations and alternative algorithmic approaches.

• Strengthen problem-solving skills through real-world, data-driven scenarios.

Learning Outcomes

After completing this lab, students will be able to:

• Implement and optimize classic algorithms using AI-assisted coding tools.

• Compare multiple algorithms for the same problem and justify their selection.

• Measure and analyze runtime performance using experimental data.

• Critically review and refine AI-generated algorithmic solutions.

Task 1: Sorting Student Records for Placement Drive

Scenario

SR University's Training and Placement Cell needs to shortlist candidates efficiently during campus placements. Student records must be sorted by CGPA in descending order.

Tasks

1. Use GitHub Copilot to generate a program that stores student records (Name, Roll Number, CGPA).

2. Implement the following sorting algorithms using AI assistance:

o Quick Sort

o Merge Sort

3. Measure and compare runtime performance for large datasets.

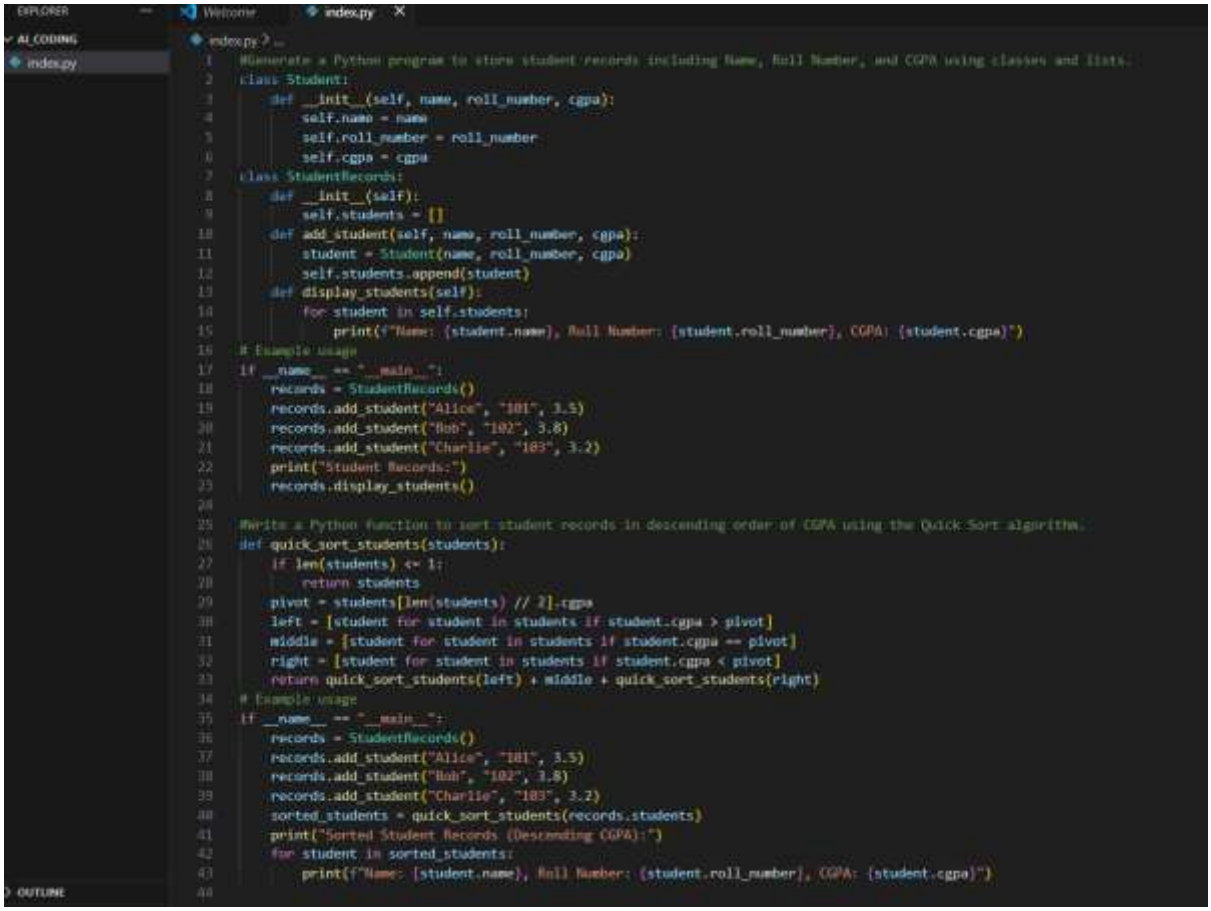
4. Write a function to display the top 10 students based on CGPA.

Expected Outcome

• Correctly sorted student records.

• Performance comparison between Quick Sort and Merge Sort.

• Clear output of top-performing students.



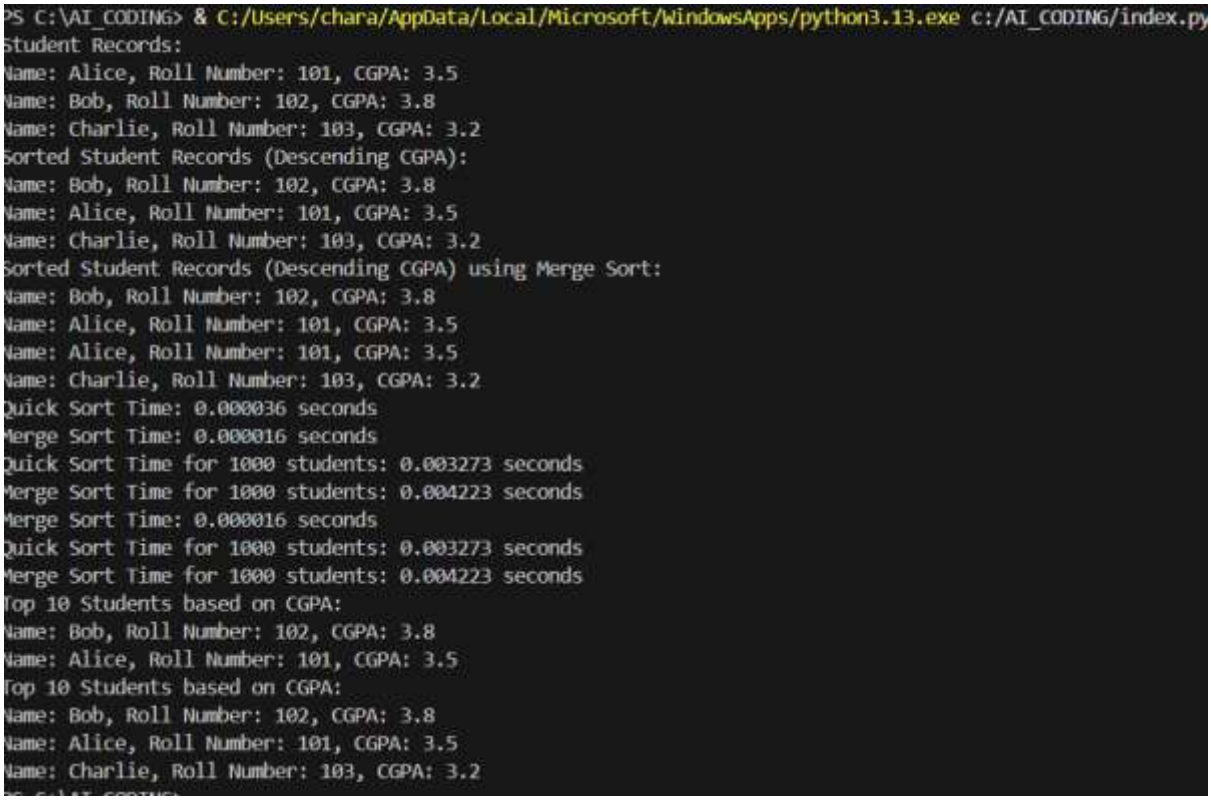

```python
#Generate a Python program to store student records including Name, Roll Number, and CGPA using classes and lists.
class Student:
    def __init__(self, name, roll_number, cgpa):
        self.name = name
        self.roll_number = roll_number
        self.cgpa = cgpa
class StudentRecords:
    def __init__(self):
        self.students = []
    def add_student(self, name, roll_number, cgpa):
        student = Student(name, roll_number, cgpa)
        self.students.append(student)
    def display_students(self):
        for student in self.students:
            print(f"Name: {student.name}, Roll Number: {student.roll_number}, CGPA: {student.cgpa}")
# Example usage
if __name__ == "__main__":
    records = StudentRecords()
    records.add_student("Alice", "101", 3.5)
    records.add_student("Bob", "102", 3.8)
    records.add_student("Charlie", "103", 3.2)
    print("Student Records:")
    records.display_students()

#Write a Python function to sort student records in descending order of CGPA using the Quick Sort algorithm.
def quick_sort_students(students):
    if len(students) <= 1:
        return students
    pivot = students[len(students) // 2].cgpa
    left = [student for student in students if student.cgpa > pivot]
    middle = [student for student in students if student.cgpa == pivot]
    right = [student for student in students if student.cgpa < pivot]
    return quick_sort_students(left) + middle + quick_sort_students(right)
# Example usage
if __name__ == "__main__":
    records = StudentRecords()
    records.add_student("Alice", "101", 3.5)
    records.add_student("Bob", "102", 3.8)
    records.add_student("Charlie", "103", 3.2)
    sorted_students = quick_sort_students(records.students)
    print("Sorted Student Records (Descending CGPA):")
    for student in sorted_students:
        print(f"Name: {student.name}, Roll Number: {student.roll_number}, CGPA: {student.cgpa}")
```

```python
import time
# Performance comparison
if __name__ == "__main__":
    records = StudentRecords()
    records.add_student("Alice", "101", 3.5)
    records.add_student("Bob", "102", 3.8)
    records.add_student("Charlie", "103", 3.2)
    # Timing Quick Sort
    start_time = time.time()
    sorted_students_quick = quick_sort_students(records.students)
    end_time = time.time()
    print(f"Quick Sort Time: {end_time - start_time:.6f} seconds")
    # Timing Merge Sort
    start_time = time.time()
    sorted_students_merge = merge_sort_students(records.students)
    end_time = time.time()
    print(f"Merge Sort Time: {end_time - start_time:.6f} seconds")

#Generate Python code to measure and compare the execution time of Quick Sort and Merge Sort for large datasets.
import random
def generate_random_students(num_students):
    students = []
    for i in range(num_students):
        name = f"Student_{i}"
        roll_number = f"{100 + i}"
        cgpa = round(random.uniform(2.0, 4.0), 2)
        students.append(Student(name, roll_number, cgpa))
    return students
if __name__ == "__main__":
    num_students = 1000
    random_students = generate_random_students(num_students)
    # Timing Quick Sort
    start_time = time.time()
    sorted_students_quick = quick_sort_students(random_students.copy())
    end_time = time.time()
    print(f"Quick Sort Time for {num_students} students: {end_time - start_time:.6f} seconds")
    # Timing Merge Sort
    start_time = time.time()
    sorted_students_merge = merge_sort_students(random_students.copy())
    end_time = time.time()
    print(f"Merge Sort Time for {num_students} students: {end_time - start_time:.6f} seconds")

#Write a Python function to display the top 10 students based on CGPA from a sorted list.
def display_top_students(students, top_n=10):
    print(f"Top {top_n} Students based on CGPA:")
    for student in students[:top_n]:
        print(f"Name: {student.name}, Roll Number: {student.roll_number}, CGPA: {student.cgpa}")
# Example usage
if __name__ == "__main__":
    records = StudentRecords()
    records.add_student("Alice", "101", 3.5)
    records.add_student("Bob", "102", 3.8)
    records.add_student("Charlie", "103", 3.2)
    sorted_students = quick_sort_students(records.students)
    display_top_students(sorted_students)
```



Task 2: Implementing Bubble Sort with AI Comments

• Task: Write a Python implementation of Bubble Sort.

• Instructions:

• Students implement Bubble Sort normally.

• Ask AI to generate inline comments explaining key logic (like

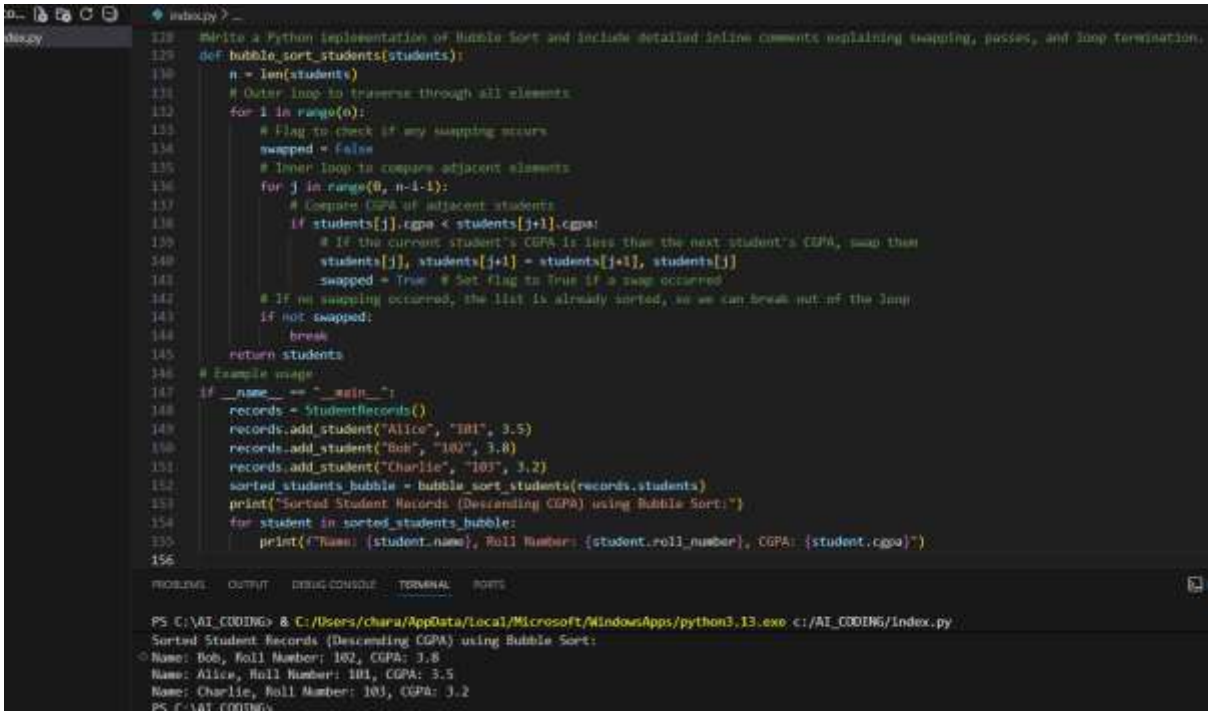
swapping, passes, and termination).

• Request AI to provide time complexity analysis.

• Expected Output:

• A Bubble Sort implementation with AI-generated explanatory

comments and complexity analysis.



Task 3: Quick Sort and Merge Sort Comparison

• Task: Implement Quick Sort and Merge Sort using recursion.

• Instructions:

• Provide AI with partially completed functions for recursion.

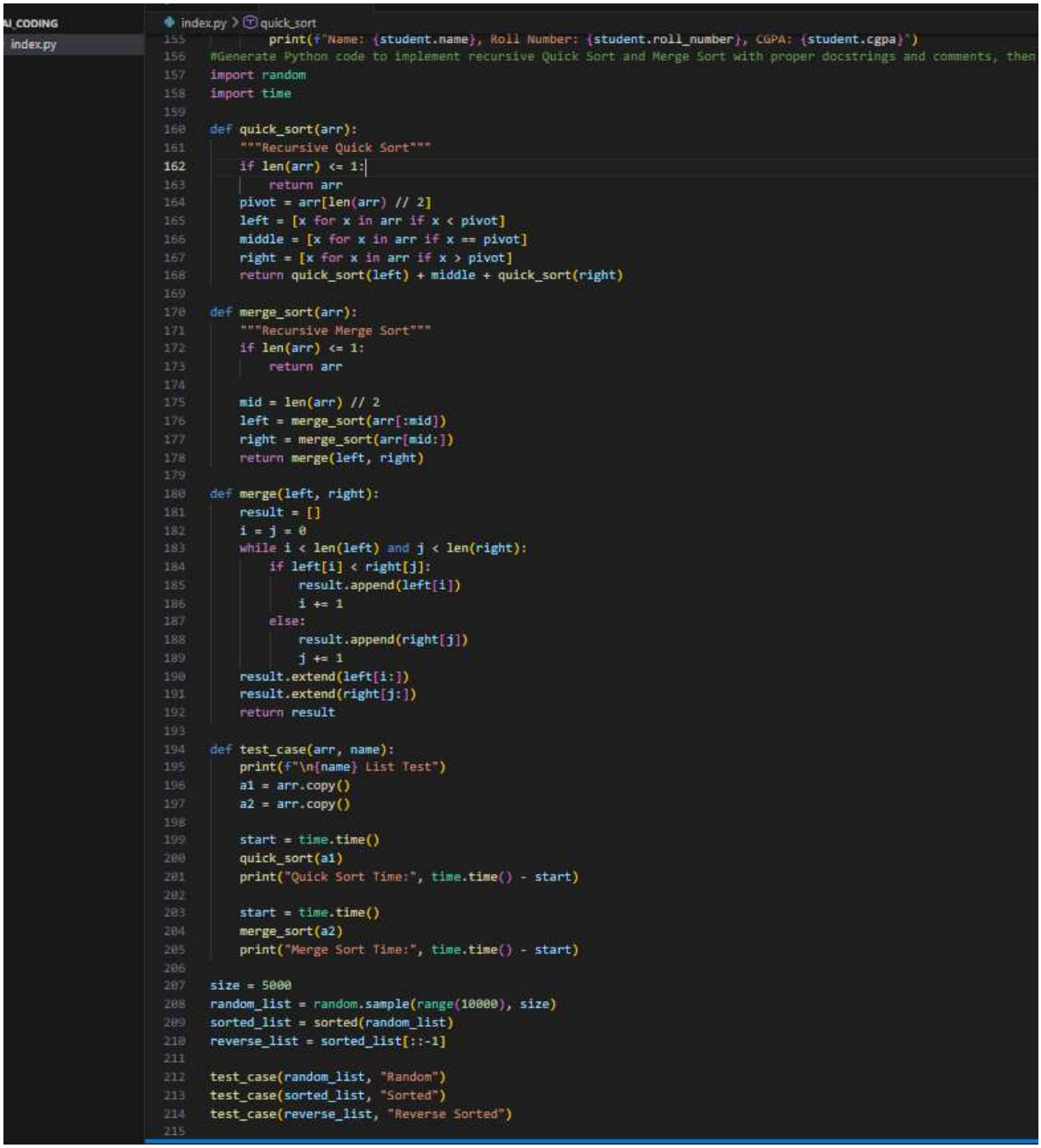
• Ask AI to complete the missing logic and add docstrings.

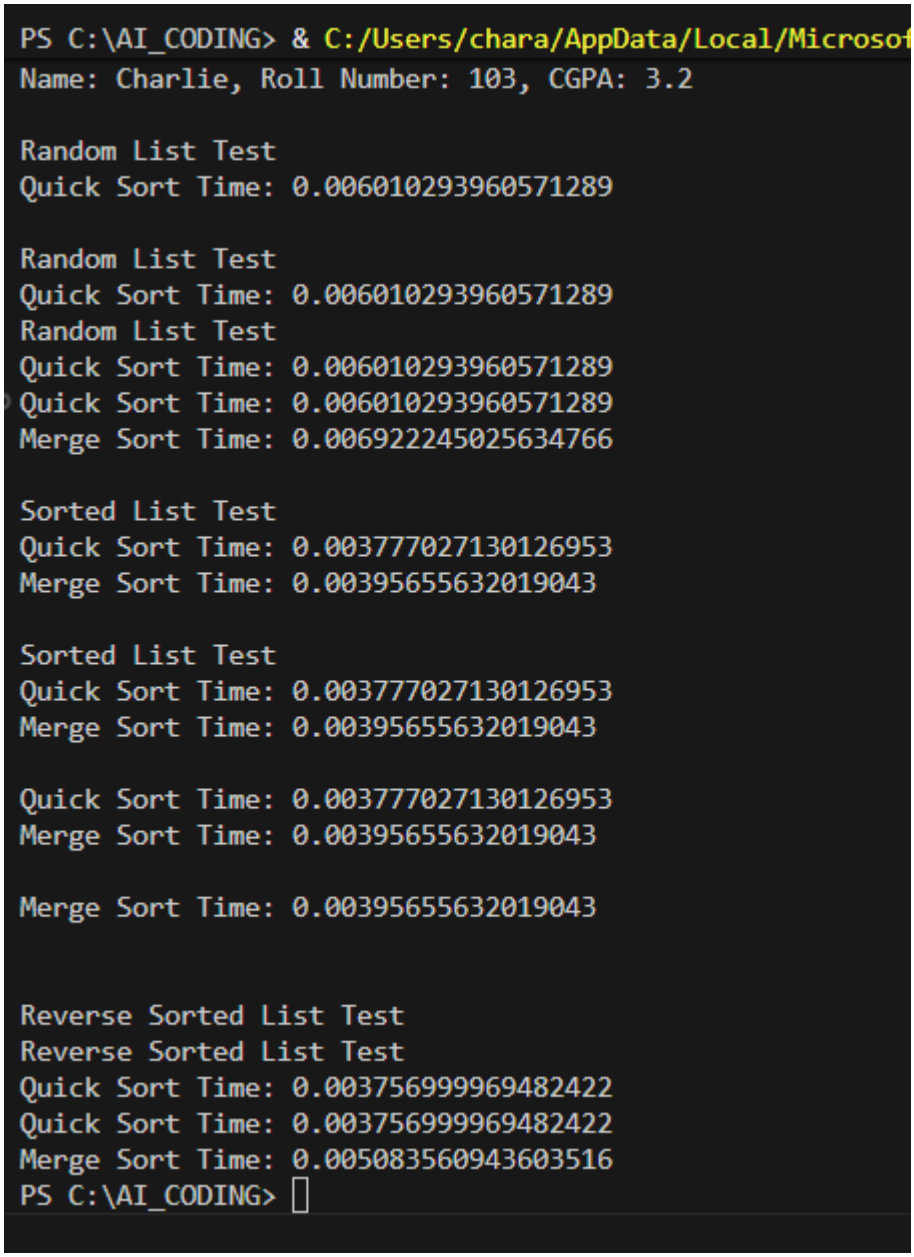
• Compare both algorithms on random, sorted, and reverse-sorted lists.

• Expected Output:

• Working Quick Sort and Merge Sort implementations.

• AI-generated explanation of average, best, and worst-case complexities.


AI_CODING
index.py
index.py > quick_sort
print(f"Name: {student.name}, Roll Number: {student.roll_number}, CGPA: {student.cgpa}")
#Generate Python code to implement recursive Quick Sort and Merge Sort with proper docstrings and comments, then
import random
import time
def quick_sort(arr):
"""Recursive Quick Sort"""
if len(arr) <= 1:
return arr
pivot = arr[len(arr) // 2]
left = [x for x in arr if x < pivot]
middle = [x for x in arr if x == pivot]
right = [x for x in arr if x > pivot]
return quick_sort(left) + middle + quick_sort(right)
def merge_sort(arr):
"""Recursive Merge Sort"""
if len(arr) <= 1:
return arr
mid = len(arr) // 2
left = merge_sort(arr[:mid])
right = merge_sort(arr[mid:])
return merge(left, right)
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
def test_case(arr, name):
print(f"\n{name} List Test")
a1 = arr.copy()
a2 = arr.copy()
start = time.time()
quick_sort(a1)
print("Quick Sort Time:", time.time() - start)
start = time.time()
merge_sort(a2)
print("Merge Sort Time:", time.time() - start)
size = 5000
random_list = random.sample(range(10000), size)
sorted_list = sorted(random_list)
reverse_list = sorted_list[::-1]
test_case(random_list, "Random")
test_case(sorted_list, "Sorted")
test_case(reverse_list, "Reverse Sorted")



Task 4 (Real-Time Application – Inventory Management System)

Scenario: A retail store's inventory system contains thousands of products,

each with attributes like product ID, name, price, and stock quantity. Store staff

need to:

1. Quickly search for a product by ID or name.

2. Sort products by price or quantity for stock analysis.

Task:

• Use AI to suggest the most efficient search and sort algorithms for this

use case.

• Implement the recommended algorithms in Python.

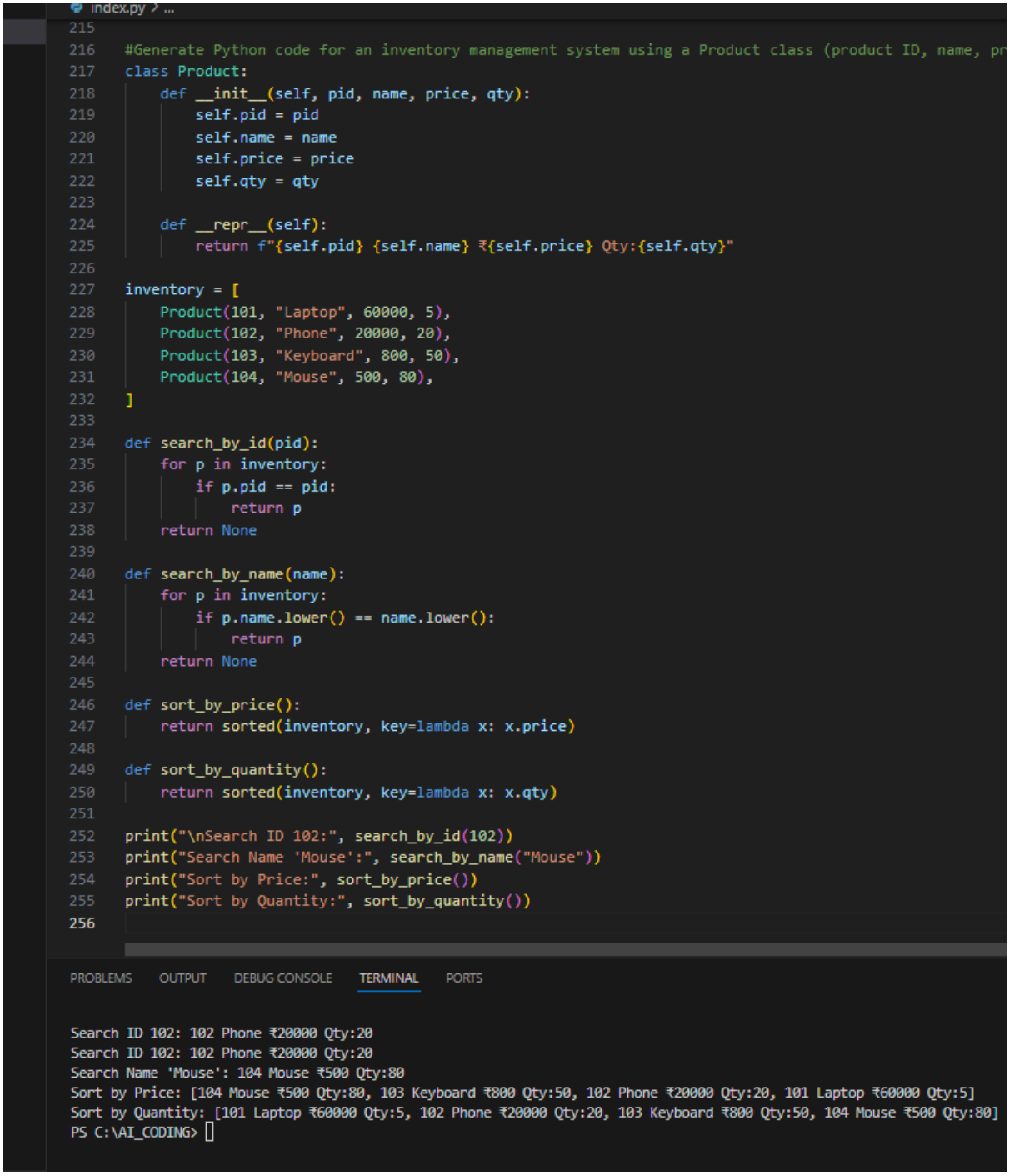
• Justify the choice based on dataset size, update frequency, and

performance requirements.

Expected Output:

• A table mapping operation → recommended algorithm → justification.

• Working Python functions for searching and sorting the inventory.



Task 5: Real-Time Stock Data Sorting & Searching

Scenario:

An AI-powered FinTech Lab at SR University is building a tool for analyzing

stock price movements. The requirement is to quickly sort stocks by daily

gain/loss and search for specific stock symbols efficiently.

• Use GitHub Copilot to fetch or simulate stock price data (Stock

Symbol, Opening Price, Closing Price).

• Implement sorting algorithms to rank stocks by percentage change.

• Implement a search function that retrieves stock data instantly when a stock symbol is entered.

• Optimize sorting with Heap Sort and searching with Hash Maps.

• Compare performance with standard library functions (sorted(), dict lookups) and analyze trade-offs