ASSIGNMENT-12.3

2303A51905

BATCH-28

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:
    - Quick Sort
    - Merge Sort
3. Measure and compare runtime performance for large datasets.
4. Write a function to display the top 10 students based on CGPA.

PROMPT :

Generate a Python program that:

-Stores student records (Name, Roll Number, CGPA).

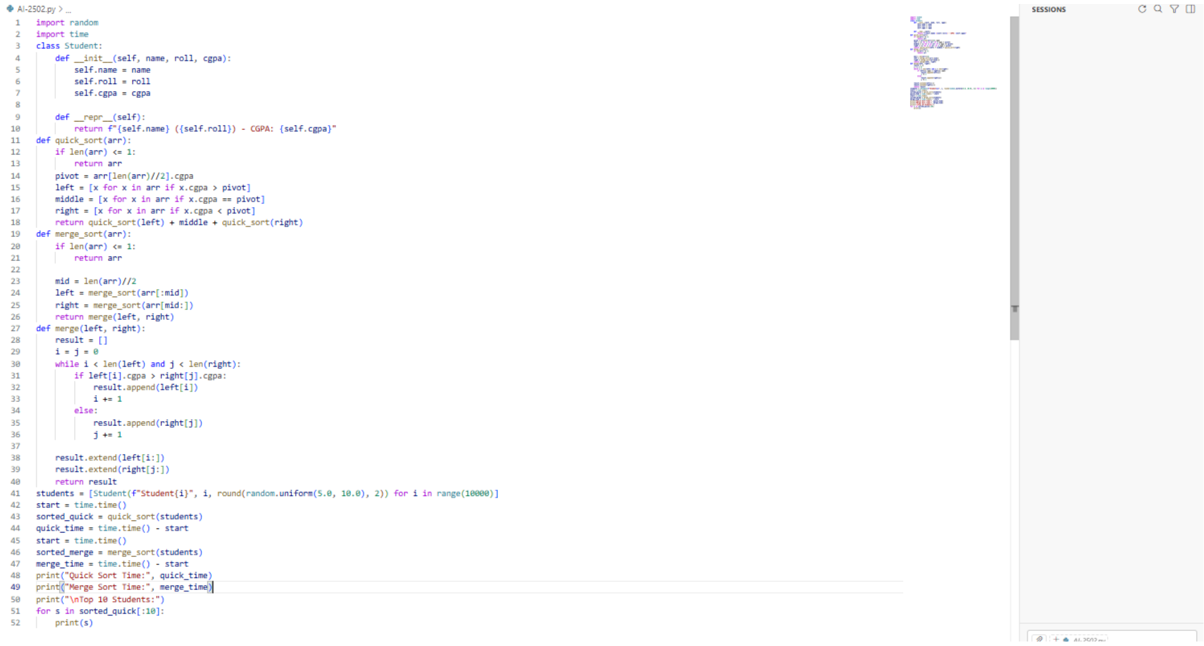
-Implements Quick Sort and Merge Sort to sort students by CGPA in descending order.

-Measures runtime for both algorithms on large datasets.

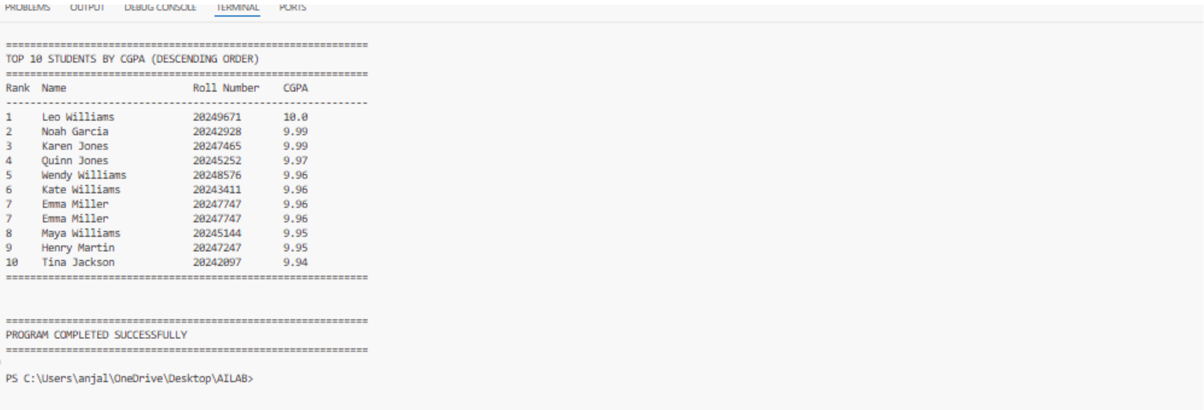
-Displays the top 10 students.

-Include proper function definitions and comments.

CODE AND OUTPUT :



```python
import random
import time
class Student:
    def __init__(self, name, roll, cgpa):
        self.name = name
        self.roll = roll
        self.cgpa = cgpa

    def __repr__(self):
        return f"{self.name} ({self.roll}) - CGPA: {self.cgpa}"
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr)//2].cgpa
    left = [x for x in arr if x.cgpa > pivot]
    middle = [x for x in arr if x.cgpa == pivot]
    right = [x for x in arr if x.cgpa < pivot]
    return quick_sort(left) + middle + 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:])
    return merge(left, right)
def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i].cgpa > right[j].cgpa:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result
students = [Student(f"Student{i}", i, round(random.uniform(5.0, 10.0), 2)) for i in range(10000)]
start = time.time()
sorted_quick = quick_sort(students)
quick_time = time.time() - start
start = time.time()
sorted_merge = merge_sort(students)
merge_time = time.time() - start
print("Quick Sort Time:", quick_time)
print("Merge Sort Time:", merge_time)
print("\nTop 10 Students:")
for s in sorted_quick[:10]:
    print(s)
```



```
============================================================
TOP 10 STUDENTS BY CGPA (DESCENDING ORDER)
============================================================
Rank  Name                    Roll Number    CGPA
------------------------------------------------------------
1     Leo Williams            20249671       10.0
2     Noah Garcia             20242928       9.99
3     Karen Jones             20247465       9.99
4     Quinn Jones             20245252       9.97
5     Wendy Williams          20248576       9.96
6     Kate Williams           20243411       9.96
7     Emma Miller             20247747       9.96
7     Emma Miller             20247747       9.96
8     Maya Williams           20245144       9.95
9     Henry Martin            20247247       9.95
10    Tina Jackson            20242097       9.94
============================================================


============================================================
PROGRAM COMPLETED SUCCESSFULLY
============================================================

PS C:\Users\anjal\OneDrive\Desktop\AILAB>
```

JUSTIFICATION :

1.Quick Sort and Merge Sort both use Divide & Conquer strategy.

2.Merge Sort guarantees **O(n log n)** in all cases.

3.Quick Sort average case is **O(n log n)** but worst case is **O(n^2)**.

4.Merge Sort requires extra space; Quick Sort is more space efficient.

5.For large datasets, Merge Sort gives stable performance while Quick Sort is often faster in practice.

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.

PROMPT :

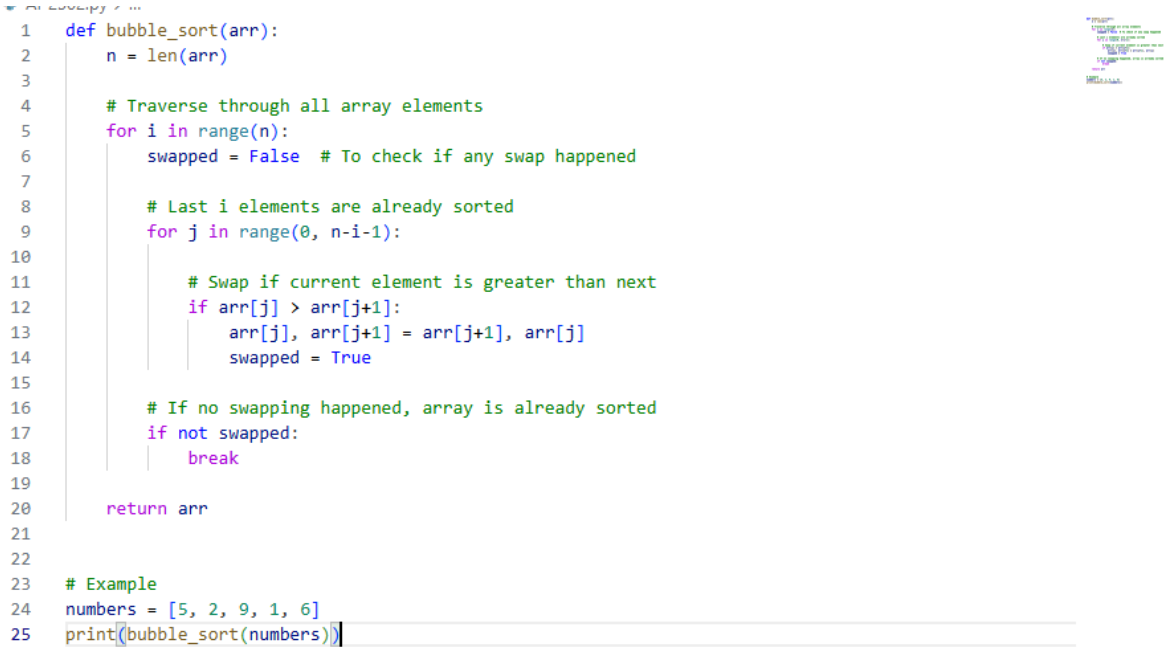
Generate a Python Bubble Sort implementation with:

- Inline comments explaining swapping and passes

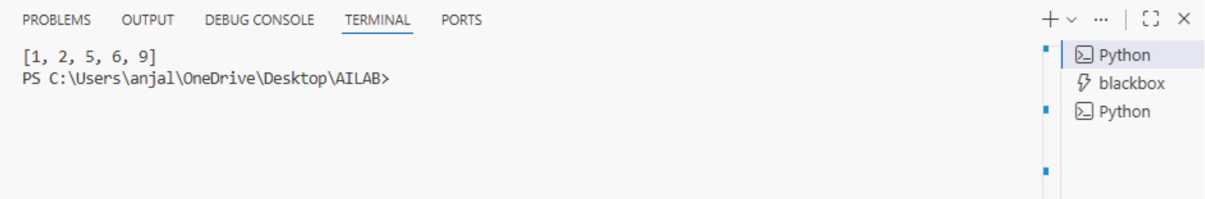
- Early termination optimization

- Time complexity analysis

CODE AND OUTPUT :



```python
def bubble_sort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):
        swapped = False  # To check if any swap happened

        # Last i elements are already sorted
        for j in range(0, n-i-1):

            # Swap if current element is greater than next
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True

        # If no swapping happened, array is already sorted
        if not swapped:
            break

    return arr


# Example
numbers = [5, 2, 9, 1, 6]
print(bubble_sort(numbers))
```



```
[1, 2, 5, 6, 9]
PS C:\Users\anjal\OneDrive\Desktop\AILAB>
```

JUSTIFICATION :

1.Worst Case: **$O(n^2)$** (reverse sorted list).

2.Average Case: **$O(n^2)$**.

3.Best Case: **O(n)** (already sorted with early termination).

4.Space Complexity: **O(1)** (in-place).

5.Not suitable for large datasets.

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

PROMPT :

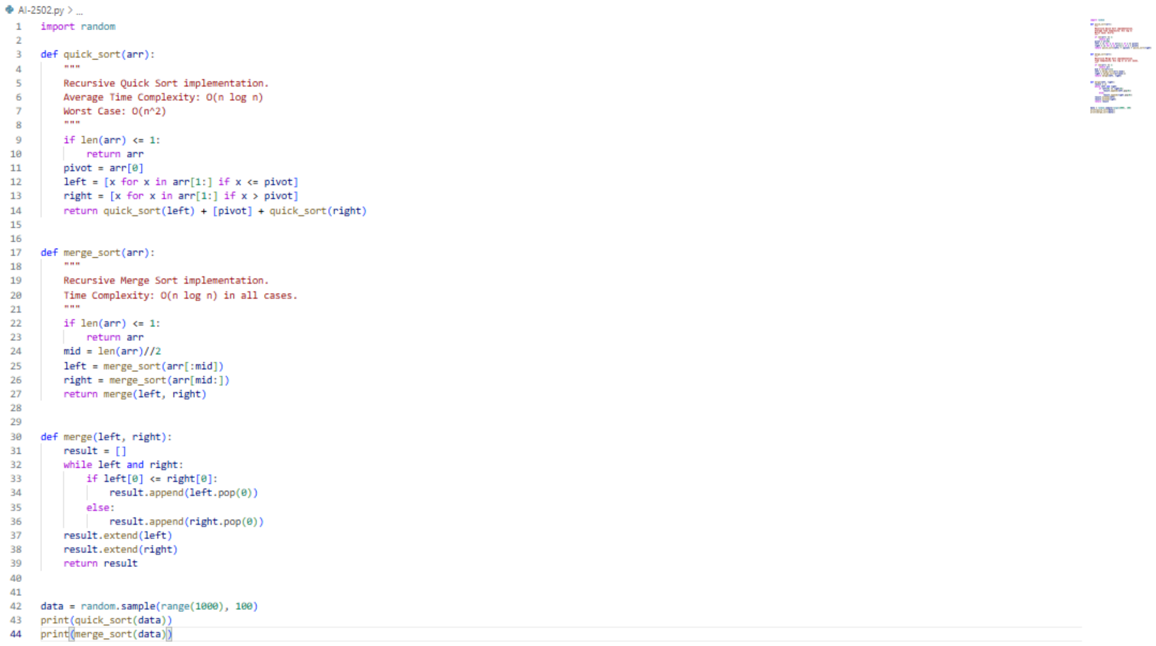
Complete recursive Quick Sort and Merge Sort functions.

-Add proper docstrings.

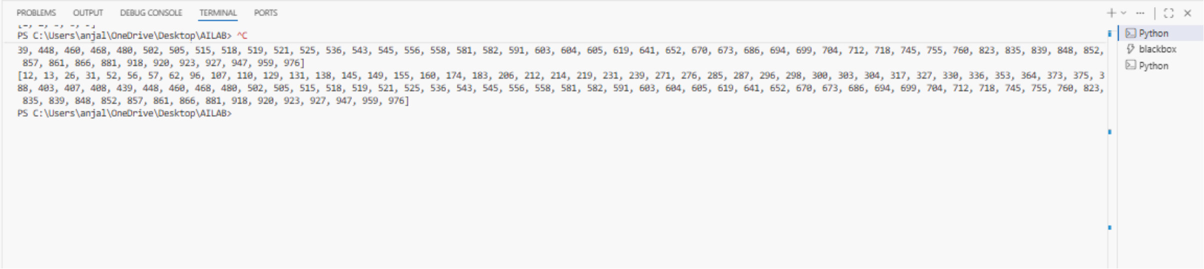
-Compare best, average, and worst case complexities.

-Test on random, sorted, and reverse-sorted lists.

CODE AND OUTPUT :



```python
import random

def quick_sort(arr):
    """
    Recursive Quick Sort implementation.
    Average Time Complexity: O(n log n)
    Worst Case: O(n^2)
    """
    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):
    """
    Recursive Merge Sort implementation.
    Time Complexity: O(n log n) in all cases.
    """
    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 = []
    while left and right:
        if left[0] <= right[0]:
            result.append(left.pop(0))
        else:
            result.append(right.pop(0))
    result.extend(left)
    result.extend(right)
    return result


data = random.sample(range(1000), 100)
print(quick_sort(data))
print(merge_sort(data))
```



```
PS C:\Users\anjal\OneDrive\Desktop\AILAB> ^C
39, 448, 460, 468, 480, 502, 505, 515, 518, 519, 521, 525, 536, 543, 545, 556, 558, 581, 582, 591, 603, 604, 605, 619, 641, 652, 670, 673, 686, 694, 699, 704, 712, 718, 745, 755, 760, 823, 835, 839, 848, 852,
 857, 861, 866, 881, 918, 920, 923, 927, 947, 959, 976]
[12, 13, 26, 31, 52, 56, 57, 62, 96, 107, 110, 129, 131, 138, 145, 149, 155, 160, 174, 183, 206, 212, 214, 219, 231, 239, 271, 276, 285, 287, 296, 298, 300, 303, 304, 317, 327, 330, 336, 353, 364, 373, 375, 3
88, 403, 407, 408, 439, 448, 460, 468, 480, 502, 505, 515, 518, 519, 521, 525, 536, 543, 545, 556, 558, 581, 582, 591, 603, 604, 605, 619, 641, 652, 670, 673, 686, 694, 699, 704, 712, 718, 745, 755, 760, 823,
 835, 839, 848, 852, 857, 861, 866, 881, 918, 920, 923, 927, 947, 959, 976]
PS C:\Users\anjal\OneDrive\Desktop\AILAB>
```

JUSTIFICATION :

1.Quick Sort Best/Average: O(n log n).

2.Quick Sort Worst: O(n^2).

3.Merge Sort always O(n log n).

4.Merge Sort requires extra memory O(n).

5.Quick Sort performs poorly on already sorted data (if bad pivot chosen).

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.

PROMPT :

Suggest best searching and sorting algorithms for a large retail inventory system.

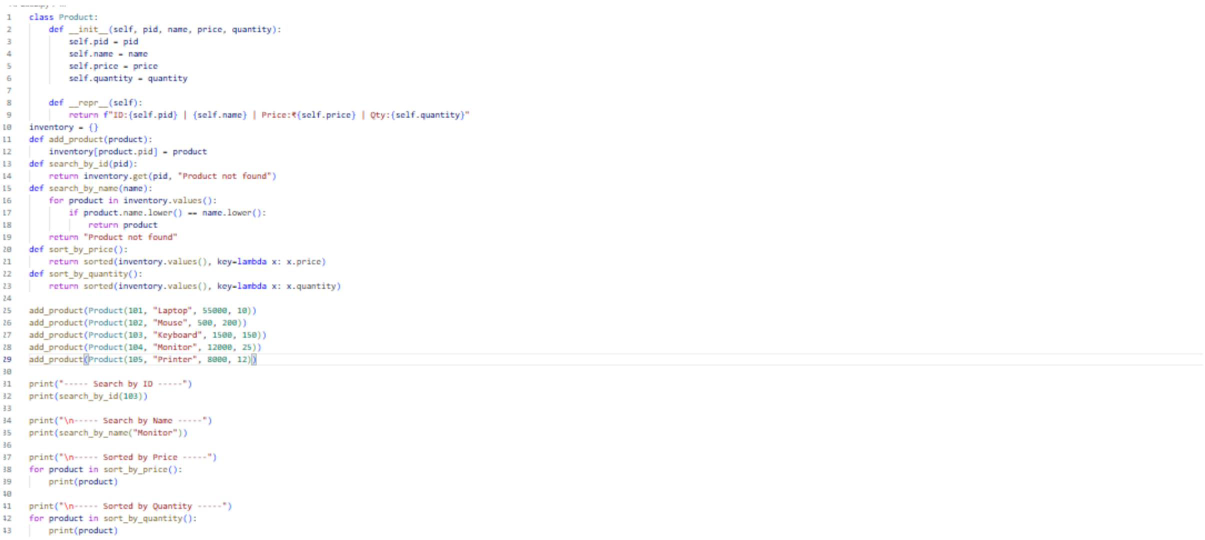
-Dataset size: thousands of products.

-Frequent searches by ID and name.

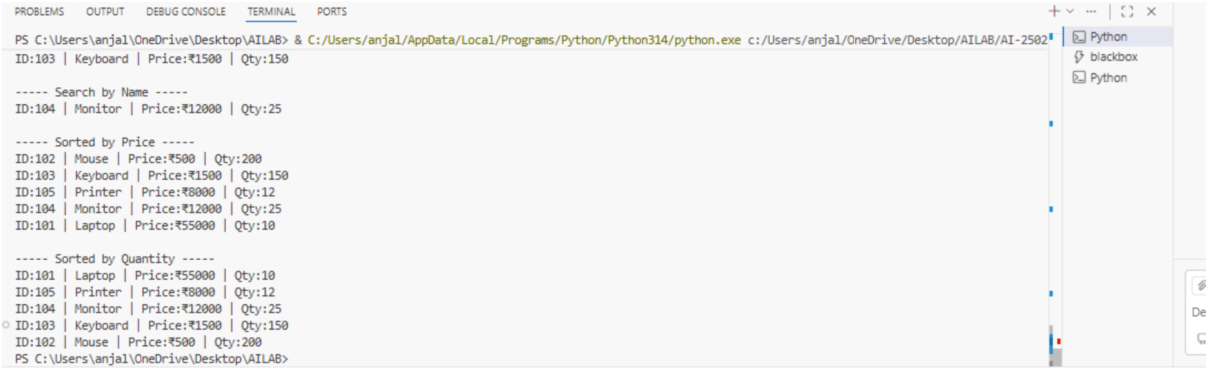
-Occasional sorting by price and stock.

-Provide justification and Python implementation.

CODE AND OUTPUT :



```python
class Product:
    def __init__(self, pid, name, price, quantity):
        self.pid = pid
        self.name = name
        self.price = price
        self.quantity = quantity

    def __repr__(self):
        return f"ID:{self.pid} | {self.name} | Price:₹{self.price} | Qty:{self.quantity}"
inventory = {}
def add_product(product):
    inventory[product.pid] = product
def search_by_id(pid):
    return inventory.get(pid, "Product not found")
def search_by_name(name):
    for product in inventory.values():
        if product.name.lower() == name.lower():
            return product
    return "Product not found"
def sort_by_price():
    return sorted(inventory.values(), key=lambda x: x.price)
def sort_by_quantity():
    return sorted(inventory.values(), key=lambda x: x.quantity)

add_product(Product(101, "Laptop", 55000, 10))
add_product(Product(102, "Mouse", 500, 200))
add_product(Product(103, "Keyboard", 1500, 150))
add_product(Product(104, "Monitor", 12000, 25))
add_product(Product(105, "Printer", 8000, 12))

print("----- Search by ID -----")
print(search_by_id(103))

print("\n----- Search by Name -----")
print(search_by_name("Monitor"))

print("\n----- Sorted by Price -----")
for product in sort_by_price():
    print(product)

print("\n----- Sorted by Quantity -----")
for product in sort_by_quantity():
    print(product)
```



```
PS C:\Users\anjal\OneDrive\Desktop\AILAB> & C:/Users/anjal/AppData/Local/Programs/Python/Python314/python.exe c:/Users/anjal/OneDrive/Desktop/AILAB/AI-2502
ID:103 | Keyboard | Price:₹1500 | Qty:150

----- Search by Name -----
ID:104 | Monitor | Price:₹12000 | Qty:25

----- Sorted by Price -----
ID:102 | Mouse | Price:₹500 | Qty:200
ID:103 | Keyboard | Price:₹1500 | Qty:150
ID:105 | Printer | Price:₹8000 | Qty:12
ID:104 | Monitor | Price:₹12000 | Qty:25
ID:101 | Laptop | Price:₹55000 | Qty:10

----- Sorted by Quantity -----
ID:101 | Laptop | Price:₹55000 | Qty:10
ID:105 | Printer | Price:₹8000 | Qty:12
ID:104 | Monitor | Price:₹12000 | Qty:25
ID:103 | Keyboard | Price:₹1500 | Qty:150
ID:102 | Mouse | Price:₹500 | Qty:200
PS C:\Users\anjal\OneDrive\Desktop\AILAB>
```

JUSTIFICATION :

1.Hash Maps give O(1) search time.

2.Thousands of products require fast lookup.

3.Python's sorted() uses Timsort (O(n log n)).

4.Frequent search → prioritize Hashing.

5.Sorting is occasional → efficient built-in sort is best.

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.

PROMPT :

Simulate stock data (Symbol, Opening Price, Closing Price).

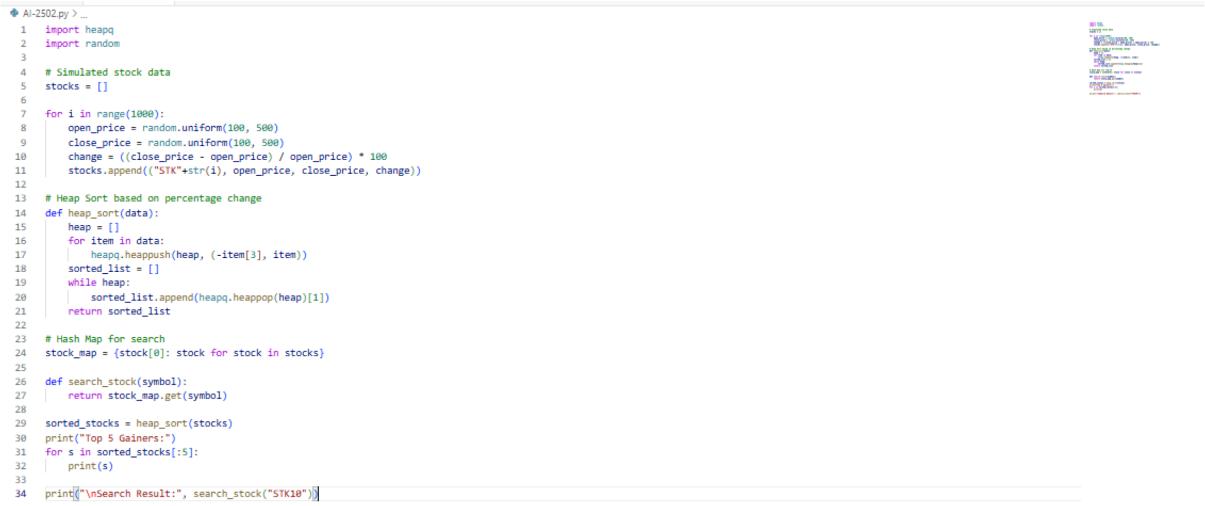
-Calculate percentage change.

-Implement Heap Sort for ranking.

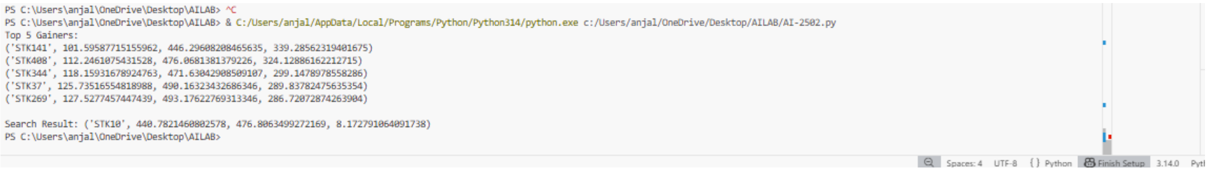
-Use Hash Map for fast symbol search.

-Compare performance with sorted() and dict lookup.

CODE AND OUTPUT :



```python
import heapq
import random

# Simulated stock data
stocks = []

for i in range(1000):
    open_price = random.uniform(100, 500)
    close_price = random.uniform(100, 500)
    change = ((close_price - open_price) / open_price) * 100
    stocks.append(("STK"+str(i), open_price, close_price, change))

# Heap Sort based on percentage change
def heap_sort(data):
    heap = []
    for item in data:
        heapq.heappush(heap, (-item[3], item))
    sorted_list = []
    while heap:
        sorted_list.append(heapq.heappop(heap)[1])
    return sorted_list

# Hash Map for search
stock_map = {stock[0]: stock for stock in stocks}

def search_stock(symbol):
    return stock_map.get(symbol)

sorted_stocks = heap_sort(stocks)
print("Top 5 Gainers:")
for s in sorted_stocks[:5]:
    print(s)

print("\nSearch Result:", search_stock("STK10"))
```



```
PS C:\Users\anjal\OneDrive\Desktop\AILAB> ^C
PS C:\Users\anjal\OneDrive\Desktop\AILAB> & C:/Users/anjal/AppData/Local/Programs/Python/Python314/python.exe c:/Users/anjal/OneDrive/Desktop/AILAB/AI-2502.py
Top 5 Gainers:
('STK141', 101.59587715155962, 446.29608208465635, 339.28562319401675)
('STK408', 112.2461075431528, 476.0681381379226, 324.12886162212715)
('STK344', 118.15931678924763, 471.63042908509107, 299.1478978558286)
('STK37', 125.73516554818988, 490.16323432686346, 289.83782475635354)
('STK269', 127.5277457447439, 493.17622769313346, 286.72072874263904)

Search Result: ('STK10', 440.7821460802578, 476.8063499272169, 8.172791064091738)
PS C:\Users\anjal\OneDrive\Desktop\AILAB>
```

JUSTIFICATION :

1.Heap Sort gives O(n log n) ranking efficiency.

2.Hash Map provides O(1) instant stock lookup.

3.Python sorted() is optimized (Timsort).

4.Heap is useful when repeatedly extracting top gainers.

5.Dict lookup is faster than linear search.