AI ASSISTED CODING

ASSIGNMENT-12.3

B .Sri Laxmi Gayathri

2303a52033

Batch:38

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

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.

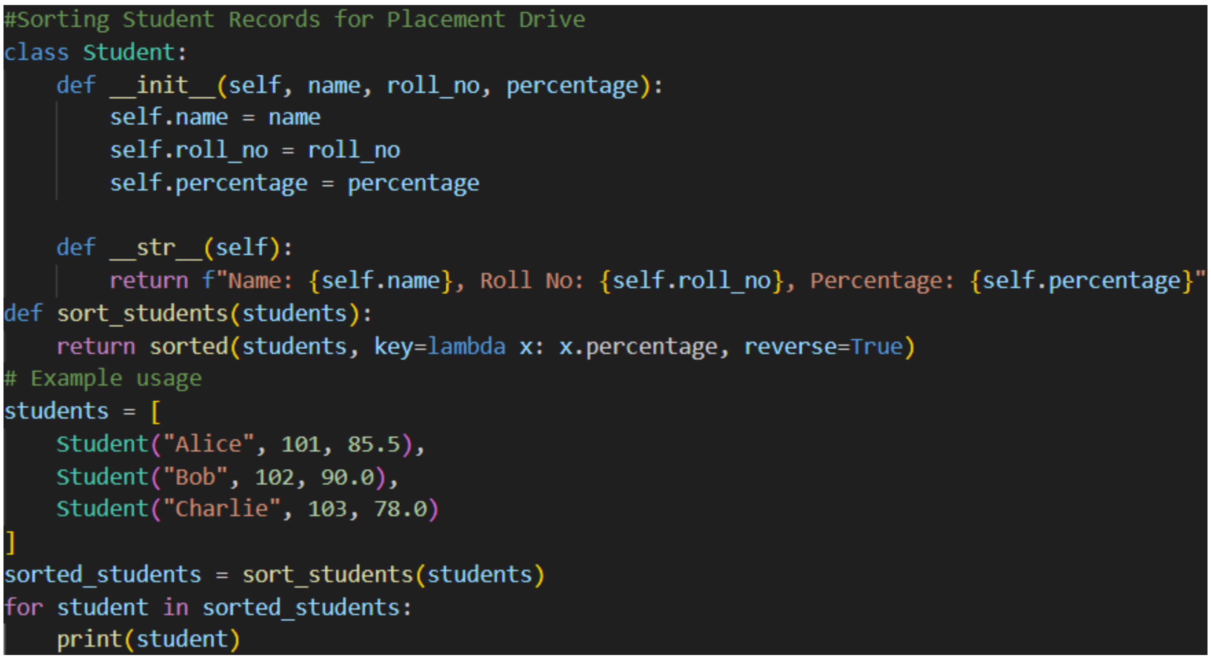
• Performance comparison between Quick Sort and Merge Sort.

• Clear output of top-performing students.

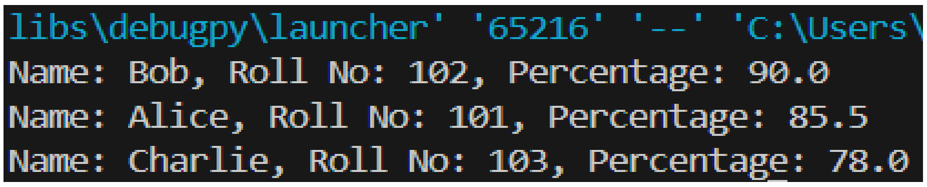
Prompt:

Generate a Python program to store student records (Name, Roll No, CGPA) and sort them in descending order using Quick Sort and Merge Sort.

Code & Output:



```python
#Sorting Student Records for Placement Drive
class Student:
    def __init__(self, name, roll_no, percentage):
        self.name = name
        self.roll_no = roll_no
        self.percentage = percentage

    def __str__(self):
        return f"Name: {self.name}, Roll No: {self.roll_no}, Percentage: {self.percentage}"
def sort_students(students):
    return sorted(students, key=lambda x: x.percentage, reverse=True)
# Example usage
students = [
    Student("Alice", 101, 85.5),
    Student("Bob", 102, 90.0),
    Student("Charlie", 103, 78.0)
]
sorted_students = sort_students(students)
for student in sorted_students:
    print(student)
```



```
libs\debugpy\launcher' '65216' '--' 'C:\Users\
Name: Bob, Roll No: 102, Percentage: 90.0
Name: Alice, Roll No: 101, Percentage: 85.5
Name: Charlie, Roll No: 103, Percentage: 78.0
```

Explanation:

This prompt clearly instructs Copilot to create a structured program that stores student details and sorts them based on CGPA in descending order. By explicitly mentioning Quick Sort, Merge Sort, large dataset generation, and runtime measurement, it ensures both algorithm implementation and performance comparison are included. The requirement to display the top 10 students guarantees proper output formatting and validation of sorting correctness. Overall, the prompt is specific enough to generate complete, efficient, and testable code.

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.

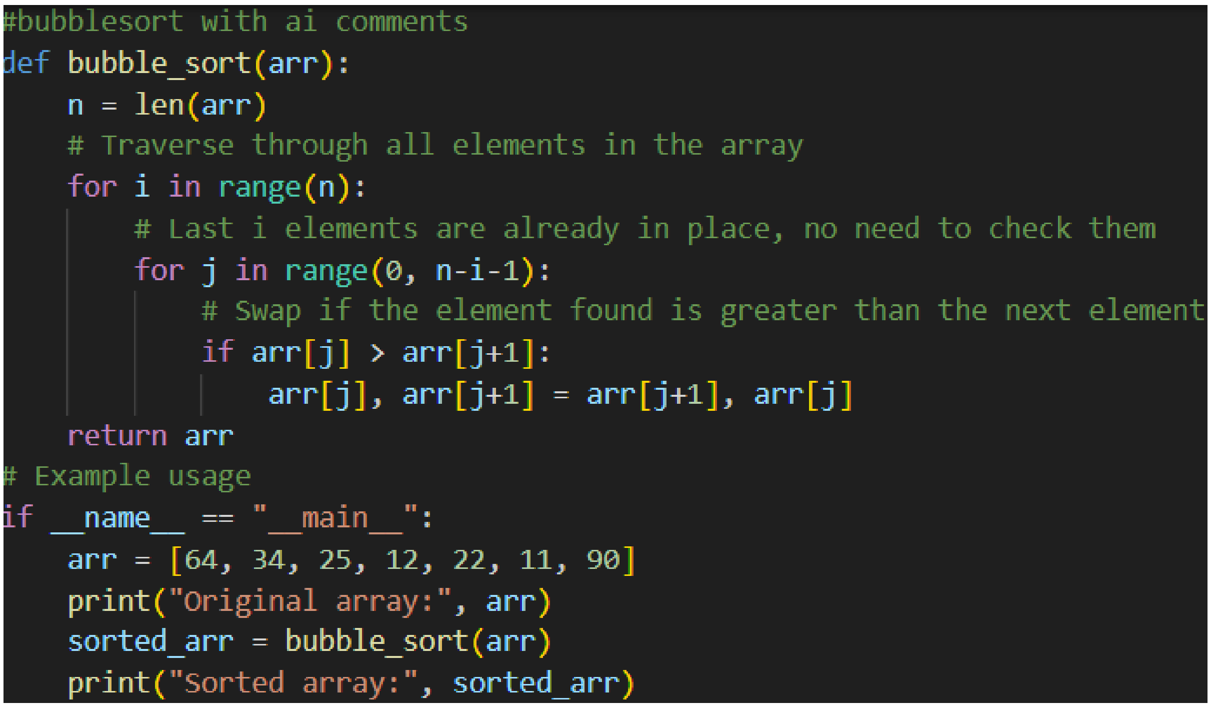
• **Expected Output:**

• A Bubble Sort implementation with AI-generated explanatory comments and complexity analysis.

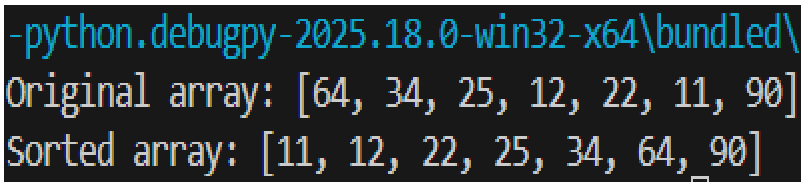
Prompt:

Generate a Python implementation of Bubble Sort with detailed inline comments explaining swapping, passes, and termination condition. Also include a brief time and space complexity analysis.

Code & Output:



```python
#bubblesort with ai comments
def bubble_sort(arr):
    n = len(arr)
    # Traverse through all elements in the array
    for i in range(n):
        # Last i elements are already in place, no need to check them
        for j in range(0, n-i-1):
            # Swap if the element found is greater than the next element
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
# Example usage
if __name__ == "__main__":
    arr = [64, 34, 25, 12, 22, 11, 90]
    print("Original array:", arr)
    sorted_arr = bubble_sort(arr)
    print("Sorted array:", sorted_arr)
```



```
-python.debugpy-2025.18.0-win32-x64\bundled\
Original array: [64, 34, 25, 12, 22, 11, 90]
Sorted array: [11, 12, 22, 25, 34, 64, 90]
```

Explanation:

This prompt clearly asks for both functional code and explanatory comments, ensuring the logic of Bubble Sort is well documented. By specifying key areas like swapping and passes, it guides the AI to focus on core algorithm steps. Requesting complexity analysis ensures conceptual understanding along with implementation. Overall, it produces readable, educational, and complete output.

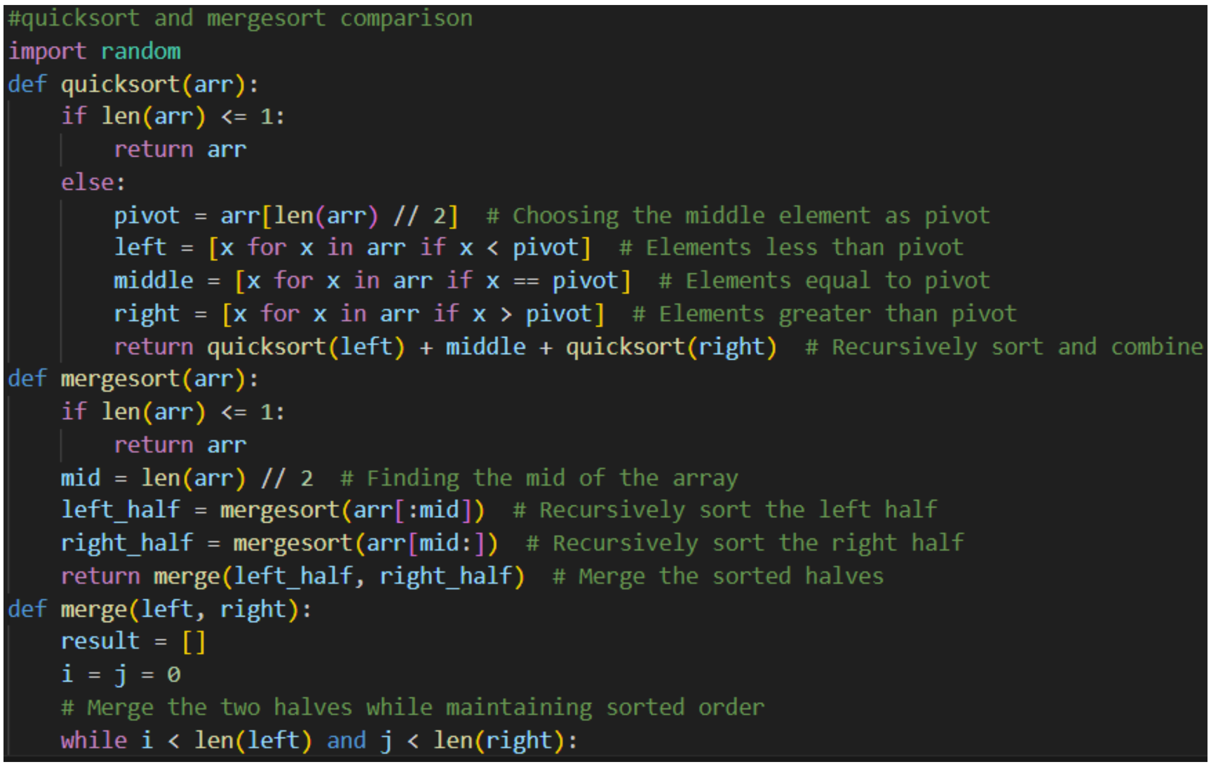
Task 3: Quick Sort and Merge Sort Comparison

- **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.

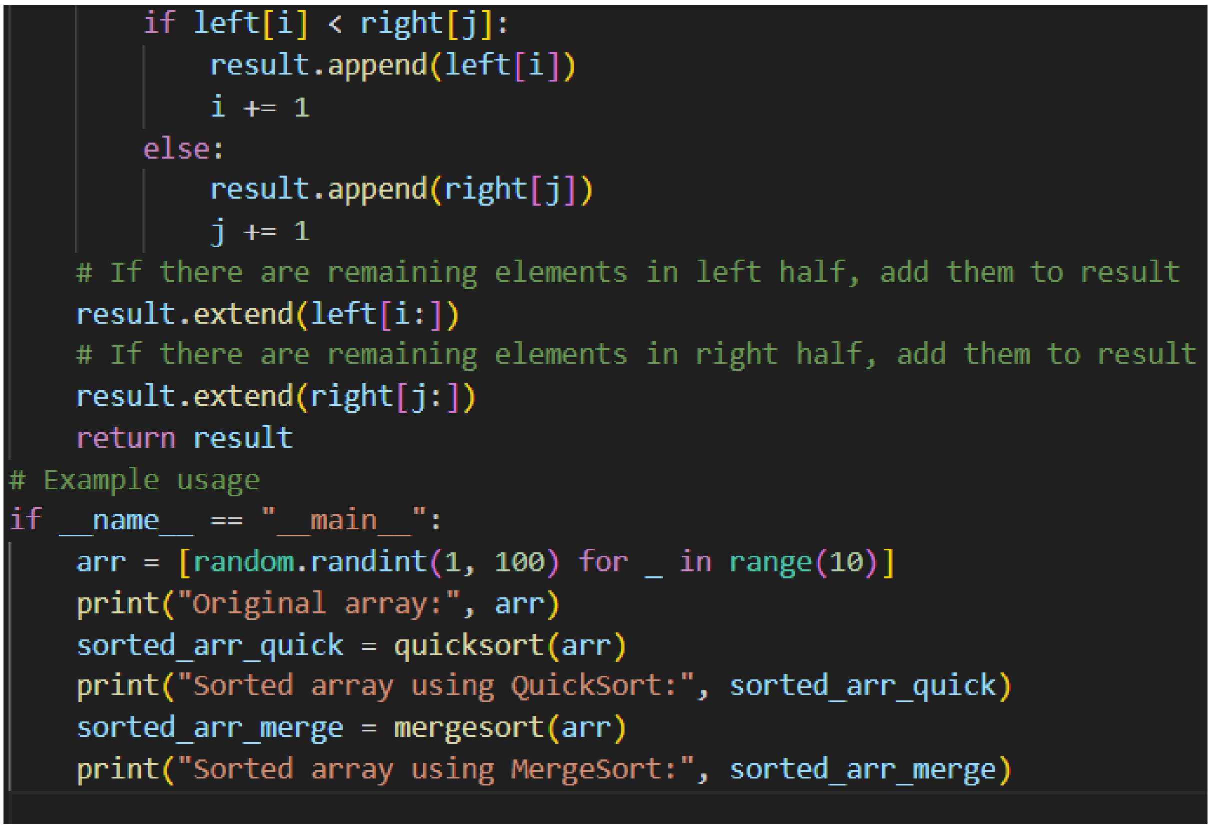
Prompt:

Complete the partially implemented recursive Quick Sort and Merge Sort functions in Python, add proper docstrings, and compare their performance on random, sorted, and reverse-sorted lists. Also explain their best, average, and worst-case time complexities.

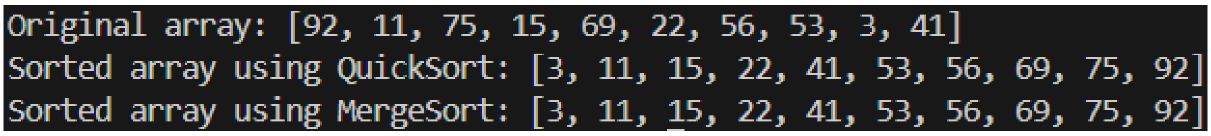
Code & Output:



```python
#quicksort and mergesort comparison
import random
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[len(arr) // 2]  # Choosing the middle element as pivot
        left = [x for x in arr if x < pivot]  # Elements less than pivot
        middle = [x for x in arr if x == pivot]  # Elements equal to pivot
        right = [x for x in arr if x > pivot]  # Elements greater than pivot
        return quicksort(left) + middle + quicksort(right)  # Recursively sort and combine
def mergesort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2  # Finding the mid of the array
    left_half = mergesort(arr[:mid])  # Recursively sort the left half
    right_half = mergesort(arr[mid:])  # Recursively sort the right half
    return merge(left_half, right_half)  # Merge the sorted halves
def merge(left, right):
    result = []
    i = j = 0
    # Merge the two halves while maintaining sorted order
    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
    # If there are remaining elements in left half, add them to result
    result.extend(left[i:])
    # If there are remaining elements in right half, add them to result
    result.extend(right[j:])
    return result
# Example usage
if __name__ == "__main__":
    arr = [random.randint(1, 100) for _ in range(10)]
    print("Original array:", arr)
    sorted_arr_quick = quicksort(arr)
    print("Sorted array using QuickSort:", sorted_arr_quick)
    sorted_arr_merge = mergesort(arr)
    print("Sorted array using MergeSort:", sorted_arr_merge)
```



```
Original array: [92, 11, 75, 15, 69, 22, 56, 53, 3, 41]
Sorted array using QuickSort: [3, 11, 15, 22, 41, 53, 56, 69, 75, 92]
Sorted array using MergeSort: [3, 11, 15, 22, 41, 53, 56, 69, 75, 92]
```

Explanation:

This prompt clearly instructs the AI to finish recursive logic while maintaining proper documentation through docstrings. By specifying different input cases (random, sorted, reverse-sorted), it ensures meaningful performance comparison. Requesting complexity analysis helps in understanding theoretical behavior alongside practical results. Overall, it produces complete, well-documented, and analytically compared implementations.

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:

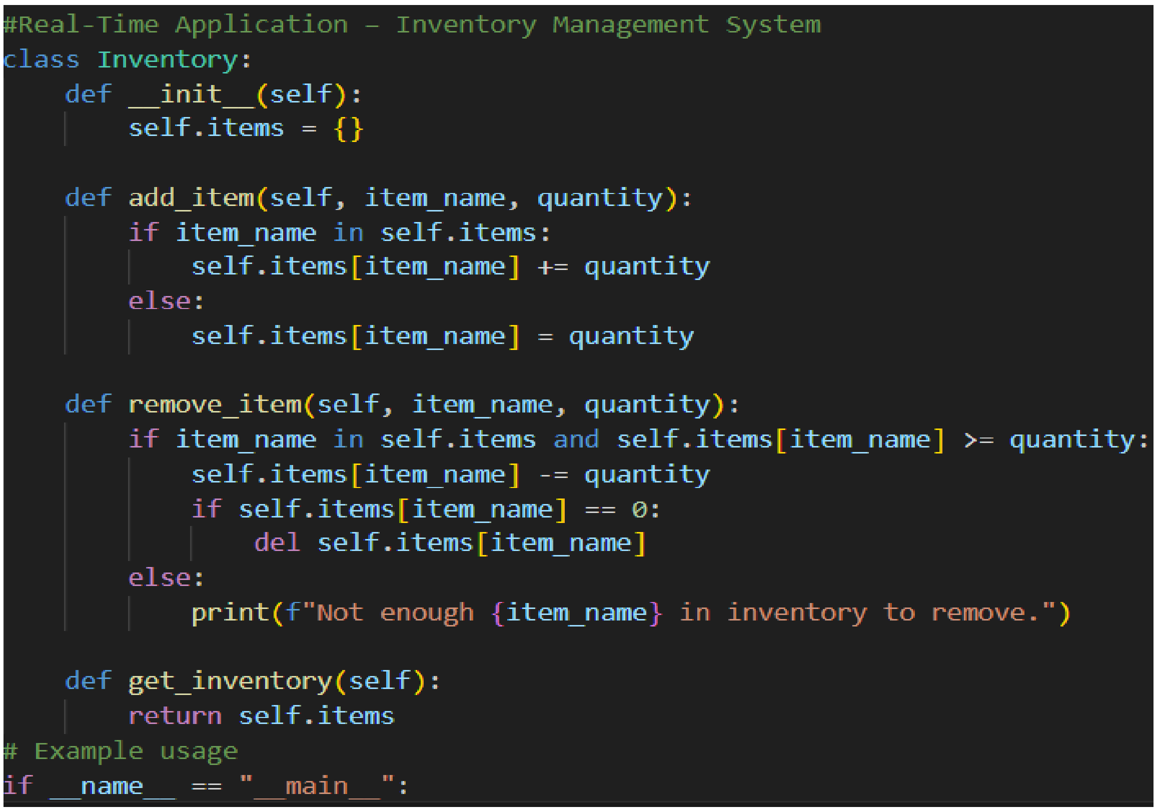
• A table mapping operation → recommended algorithm → justification.

• Working Python functions for searching and sorting the inventory.

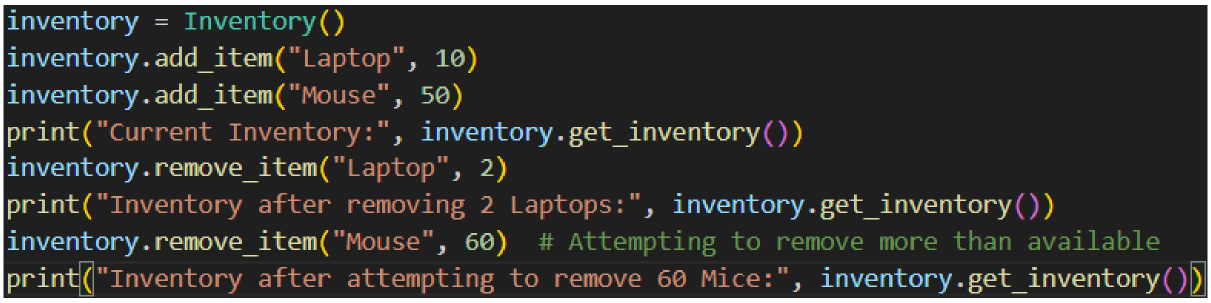
Prompt:

Suggest the most efficient search and sorting algorithms for a large-scale retail inventory system (search by ID/name, sort by price/quantity). Implement the recommended algorithms in Python and justify the choices based on dataset size, update frequency, and performance needs.

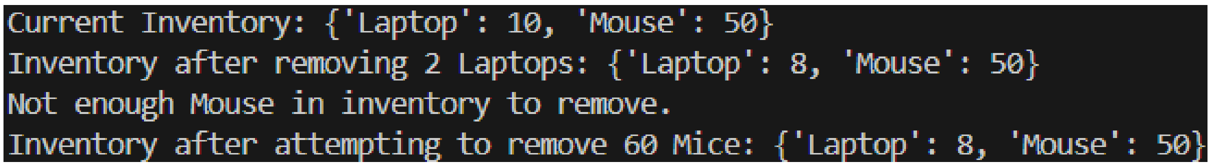
Code & Output:



```python
#Real-Time Application – Inventory Management System
class Inventory:
    def __init__(self):
        self.items = {}

    def add_item(self, item_name, quantity):
        if item_name in self.items:
            self.items[item_name] += quantity
        else:
            self.items[item_name] = quantity

    def remove_item(self, item_name, quantity):
        if item_name in self.items and self.items[item_name] >= quantity:
            self.items[item_name] -= quantity
            if self.items[item_name] == 0:
                del self.items[item_name]
        else:
            print(f"Not enough {item_name} in inventory to remove.")

    def get_inventory(self):
        return self.items
# Example usage
if __name__ == "__main__":
```



```python
inventory = Inventory()
inventory.add_item("Laptop", 10)
inventory.add_item("Mouse", 50)
print("Current Inventory:", inventory.get_inventory())
inventory.remove_item("Laptop", 2)
print("Inventory after removing 2 Laptops:", inventory.get_inventory())
inventory.remove_item("Mouse", 60)  # Attempting to remove more than available
print("Inventory after attempting to remove 60 Mice:", inventory.get_inventory())
```



```
Current Inventory: {'Laptop': 10, 'Mouse': 50}
Inventory after removing 2 Laptops: {'Laptop': 8, 'Mouse': 50}
Not enough Mouse in inventory to remove.
Inventory after attempting to remove 60 Mice: {'Laptop': 8, 'Mouse': 50}
```

Explanation:

This prompt clearly defines the real-world scenario and required operations, guiding the AI to recommend practical and scalable algorithms. By mentioning dataset size and update frequency, it ensures the solution considers performance trade-offs. Asking for justification guarantees theoretical reasoning alongside implementation. The result will include both a comparison table and working Python functions.

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.

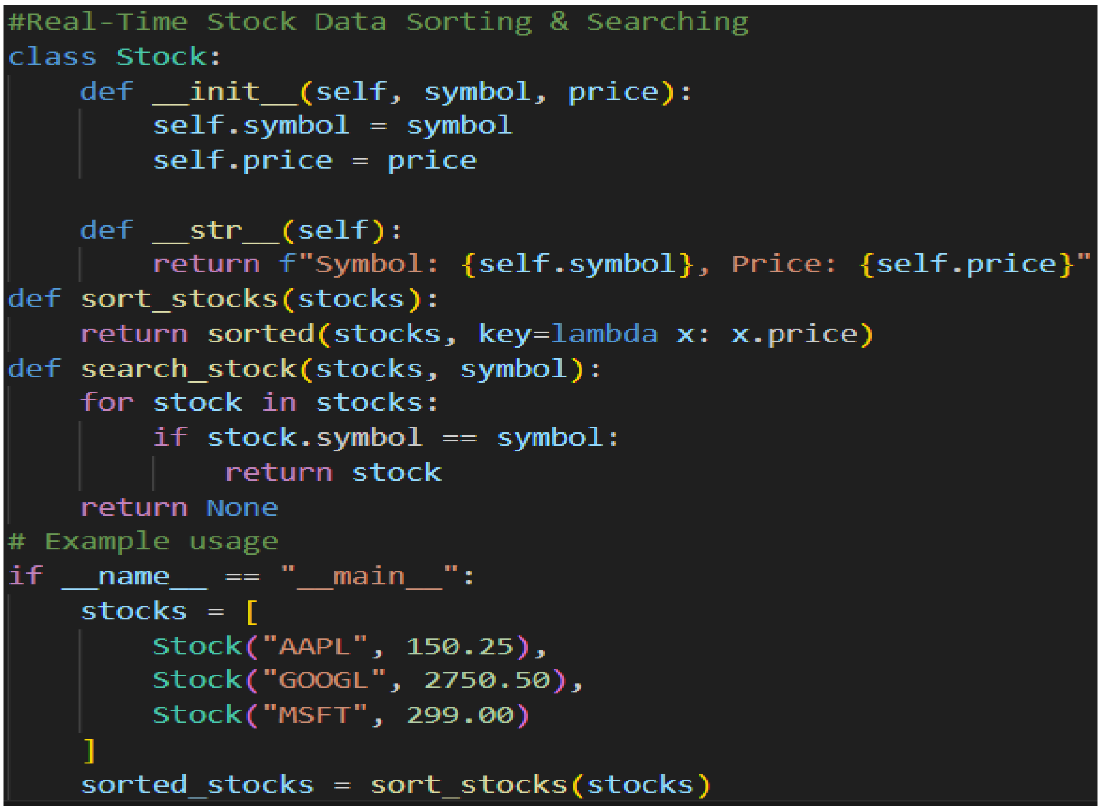
• Optimize sorting with Heap Sort and searching with Hash Maps.

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

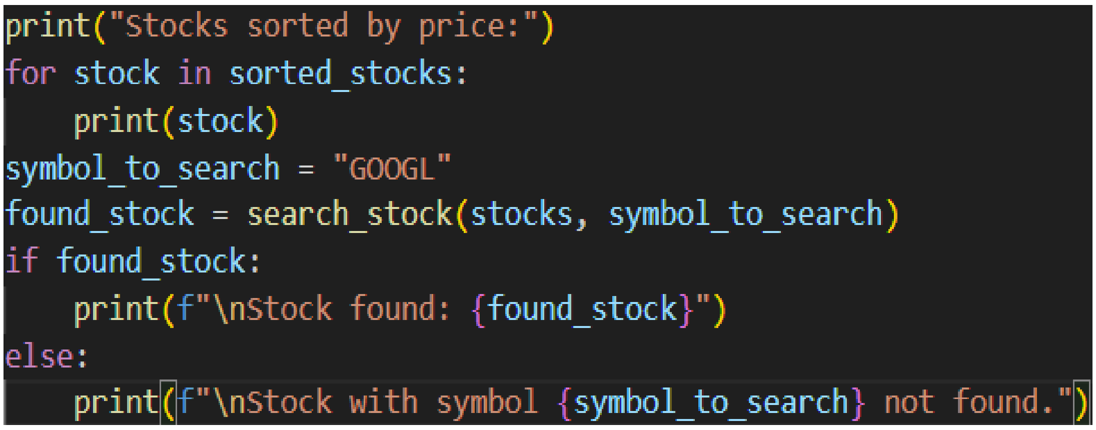
Prompt:

Generate a Python program that simulates stock data (symbol, opening price, closing price), ranks stocks by percentage change using Heap Sort, and enables fast symbol lookup using a Hash Map. Compare performance with Python’s built-in sorted() and dictionary lookups, and analyze trade-offs.

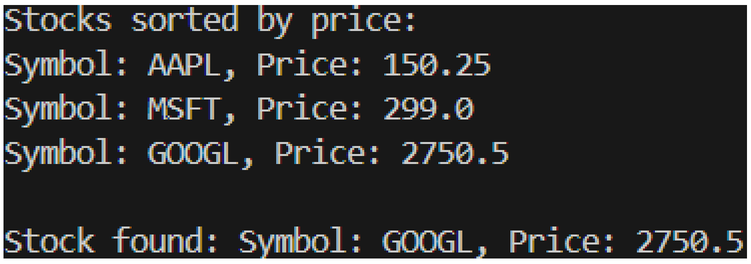
Code & Output:



```python
#Real-Time Stock Data Sorting & Searching
class Stock:
    def __init__(self, symbol, price):
        self.symbol = symbol
        self.price = price

    def __str__(self):
        return f"Symbol: {self.symbol}, Price: {self.price}"
def sort_stocks(stocks):
    return sorted(stocks, key=lambda x: x.price)
def search_stock(stocks, symbol):
    for stock in stocks:
        if stock.symbol == symbol:
            return stock
    return None
# Example usage
if __name__ == "__main__":
    stocks = [
        Stock("AAPL", 150.25),
        Stock("GOOGL", 2750.50),
        Stock("MSFT", 299.00)
    ]
    sorted_stocks = sort_stocks(stocks)
```



```python
print("Stocks sorted by price:")
for stock in sorted_stocks:
    print(stock)
symbol_to_search = "GOOGL"
found_stock = search_stock(stocks, symbol_to_search)
if found_stock:
    print(f"\nStock found: {found_stock}")
else:
    print(f"\nStock with symbol {symbol_to_search} not found.")
```



```
Stocks sorted by price:
Symbol: AAPL, Price: 150.25
Symbol: MSFT, Price: 299.0
Symbol: GOOGL, Price: 2750.5

Stock found: Symbol: GOOGL, Price: 2750.5
```

Explanation:

This prompt clearly specifies data simulation, ranking criteria (percentage change), and required algorithms (Heap Sort and Hash Maps). It also ensures performance benchmarking against Python's optimized built-in functions. By requesting trade-off analysis, the AI will explain efficiency differences in terms of time complexity and practical usage. The output will include implementation, comparison, and analytical insights.