

# AI-ASSISTANT-CODING-LAB-12.3

**Name:** V.Abhinav

**Batch:**41

**Roll-No:**2303A52174

## **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.

**Prompt:** "Create Student class and implement Quick Sort and Merge Sort"

CODE:

```
# TASK 1: Sorting Student Records for Placement Drive
# =====
# Prompt: "Create Student class and implement Quick Sort and Merge Sort"

class Student:
    def __init__(self, name, roll, cgpa):
        self.name = name
        self.roll = roll
        self.cgpa = cgpa

# Quick Sort - Time: O(n log n), Space: O(log n)
def quick_sort(arr, key=lambda x: x):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if key(x) > key(pivot)]
    middle = [x for x in arr if key(x) == key(pivot)]
    right = [x for x in arr if key(x) < key(pivot)]
    return quick_sort(left, key) + middle + quick_sort(right, key)

# Merge Sort - Time: O(n log n), Space: O(n)
def merge_sort(arr, key=lambda x: x):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid], key)
    right = merge_sort(arr[mid:], key)
    return merge(left, right, key)
```

```
def merge(left, right, key):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if key(left[i]) >= key(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
```

OUTPUT:

```
=====
TASK 1: Sorting Student Records for Placement Drive
=====

Generated 10 student records

Quick Sort: 0.000036 seconds
Top 5 Students:
1. Reyansh Verma | SR2024006 | CGPA: 9.83
2. Vivaan Reddy | SR2024004 | CGPA: 9.78
3. Ananya Gupta | SR2024005 | CGPA: 8.92
4. Aarav Kumar | SR2024000 | CGPA: 8.76
5. Ayush Nair | SR2024008 | CGPA: 8.73

Merge Sort: 0.000022 seconds
```

### Justification:

Student records must be sorted by CGPA for placement ranking.

Quick Sort

- Average Time Complexity:  $O(n \log n)$
- Space Complexity:  $O(\log n)$
- Very fast in practice for in-memory data.

Merge Sort

- Time Complexity:  $O(n \log n)$  (guaranteed)
- Stable sorting (maintains order of equal CGPAs)
- Better for large datasets or linked lists.

## 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:** "Implement Bubble Sort with detailed comments explaining logic"

### CODE:

```
# TASK 2: Bubble Sort with AI Comments
# =====
# Prompt: "Implement Bubble Sort with detailed comments explaining logic"

def bubble_sort(arr):
    """
    Bubble Sort - Time: O(n2), Space: O(1)
    Best Case: O(n) when already sorted
    """
    n = len(arr)
    for i in range(n):
        swapped = False
        # Compare adjacent elements and swap if needed
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j] # Swap
                swapped = True
        if not swapped: # Optimization: stop if no swaps
            break
    return arr
```

### OUTPUT:

```
=====
TASK 2: Bubble Sort with AI Comments
=====

Original: [64, 34, 25, 12, 22, 11, 90]
Sorted:   [11, 12, 22, 25, 34, 64, 90]

Complexity: Best O(n), Average O(n2), Worst O(n2)
```

### Justification:

Implemented to demonstrate basic sorting logic.

Easy to understand and visualize.

Uses adjacent comparisons and swapping.

Optimized using swapped flag to stop early.

### Complexity:

Best Case: O(n) (already sorted)

Average/Worst Case:  $O(n^2)$

Space:  $O(1)$

### 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:** "Compare Quick Sort and Merge Sort on different input types"

**CODE:**

```
def compare_algorithms():
    """Compare sorting algorithms on random, sorted, and reverse sorted data"""
    print("\n" + "="*70)
    print("TASK 3: Quick Sort vs Merge Sort Comparison")
    print("="*70)

    test_data = [100, 500]

    for size in test_data:
        print(f"\nDataset Size: {size} elements")

        # Random list
        random_list = [random.randint(1, 1000) for _ in range(size)]

        start = time.time()
        quick_sort(random_list.copy())
        print(f" Quick Sort (Random): {time.time() - start:.4f}s")

        start = time.time()
        merge_sort(random_list.copy())
        print(f" Merge Sort (Random): {time.time() - start:.4f}s")

        # Sorted list
        sorted_list = list(range(size))
        start = time.time()
        quick_sort(sorted_list.copy())
        print(f" Quick Sort (Sorted): {time.time() - start:.4f}s")

        start = time.time()
        merge_sort(sorted_list.copy())
        print(f" Merge Sort (Sorted): {time.time() - start:.4f}s")
```

OUTPUT:

```
=====
TASK 3: Quick Sort vs Merge Sort Comparison
=====

Dataset Size: 100 elements
Quick Sort (Random): 0.0002s
Merge Sort (Random): 0.0003s
Quick Sort (Sorted): 0.0001s
Merge Sort (Sorted): 0.0001s

Dataset Size: 500 elements
Quick Sort (Random): 0.0017s
Merge Sort (Random): 0.0009s
Quick Sort (Sorted): 0.0009s
Merge Sort (Sorted): 0.0005s
=====
```

**Justification:**

- Algorithms behave differently on:
  - Random data
  - Sorted data
  - Reverse sorted data
- Performance measured using time module.
- Helps understand:
  - Practical efficiency
  - Effect of input distribution
  - Algorithm stability

**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:** "Create inventory system with efficient search and sort"

CODE:

```
# =====
# TASK 4: Inventory Management System
# =====
# Prompt: "Create inventory system with efficient search and sort"

class Product:
    def __init__(self, id, name, price, stock):
        self.id = id
        self.name = name
        self.price = price
        self.stock = stock

class Inventory:
    """
    Inventory Management with optimized algorithms:
    - Search by ID: Hash Map O(1)
    - Sort by Price: Merge Sort O(n log n)
    """
    def __init__(self):
        self.products = []
        self.id_map = {} # Hash map for O(1) search

    def add_product(self, product):
        self.products.append(product)
        self.id_map[product.id] = product

    def search_by_id(self, product_id):
        """O(1) search using hash map"""
        return self.id_map.get(product_id)

    def sort_by_price(self):
        """Sort products by price"""
        return sorted(self.products, key=lambda p: p.price)

    def sort_by_stock(self):
        """Sort products by stock quantity"""
        return sorted(self.products, key=lambda p: p.stock)
```

OUTPUT:

```
=====
TASK 4: Inventory Management System
=====

Algorithm Selection:
  Search by ID:      Hash Map - O(1)
  Sort by Price:     Merge Sort - O(n log n)
  Sort by Stock:     Quick Sort - O(n log n)

Search 'P003': Keyboard - $75

Products sorted by price:
P002 | Mouse          | $ 25.00
P005 | Headphones     | $ 50.00
P003 | Keyboard       | $ 75.00
```

### Justification:

- Searching product by ID must be **instant** → Hash Map gives constant time lookup.
- Sorting by price/stock requires efficient comparison-based sorting.

### 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:** "Create stock analysis tool with heap sort and hash map search"

#### CODE:

```
# TASK 5: Stock Data Sorting & Searching
# =====
# Prompt: "Create stock analysis tool with heap sort and hash map search"

class Stock:
    def __init__(self, symbol, open_price, close_price):
        self.symbol = symbol
        self.open_price = open_price
        self.close_price = close_price
        self.change = ((close_price - open_price) / open_price) * 100

    def __str__(self):
        sign = "▲" if self.change >= 0 else "▼"
        return f"{self.symbol:6s} | ${self.open_price:7.2f} → ${self.close_price:7.2f} | {sign} {abs(self.change):5.2f}%"

class StockSystem:
    """Stock system with hash map for O(1) symbol lookup"""
    def __init__(self, stocks):
        self.stocks = stocks
        self.stock_map = {s.symbol: s for s in stocks}

    def search_symbol(self, symbol):
        """O(1) search using hash map"""
        return self.stock_map.get(symbol)

    def sort_by_change(self):
        """Sort stocks by percentage change"""
        return sorted(self.stocks, key=lambda s: s.change, reverse=True)
```

OUTPUT:

```
=====
TASK 5: Stock Data Sorting & Searching
=====

Stock Performance:
  AAPL   | $ 450.00 → $ 468.50 | ▲ 4.11%
  GOOGL  | $ 140.00 → $ 145.60 | ▲ 4.00%
  AMZN   | $ 175.00 → $ 178.20 | ▲ 1.83%
  MSFT   | $ 380.00 → $ 372.00 | ▼ 2.11%
  TSLA   | $ 250.00 → $ 242.50 | ▼ 3.00%

Quick Lookup 'AAPL': AAPL   | $ 450.00 → $ 468.50 | ▲ 4.11%
=====
```

**Justification:**

- Stock systems require **fast symbol search** → Hash Map ideal.
- Investors analyze gain/loss → Sorting by % change is essential.
- Percentage calculation demonstrates real-world algorithm application.