LAB – 12.3

AI-Assisted Coding

Name – Rounak Raj

Hall-Ticket – 2303A54043

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

Optimization Using AI Tools

Lab Objectives

The objectives of this laboratory exercise are to:

• 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.

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.

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.

Note: Report should be submitted as a word document for all tasks in a single

document with prompts, comments & code explanation, and output and if

required, screenshots.

SOLUTION:

Task 1: Sorting Student Records for Placement Drive

```
import time

import random


class Student:

    def __init__(self, name, roll_no, cgpa):

        self.name = name

        self.roll_no = roll_no

        self.cgpa = cgpa


    def __repr__(self):

        return f"{self.name} ({self.roll_no}): {self.cgpa}"


def quick_sort(arr):

    """AI-generated Quick Sort (Descending)."""

    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):
    """AI-generated Merge Sort (Descending)."""
    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].cgpa >= right[0].cgpa:
            result.append(left.pop(0))
        else:
            result.append(right.pop(0))
    result.extend(left or right)
```

```python
    return result


# Simulation
data = [Student(f"Student{i}", i, round(random.uniform(6.0, 10.0), 2)) for i in range(1000)]


# Timing Quick Sort
start = time.time()
qs_result = quick_sort(data)
print(f"Quick Sort Time: {time.time() - start:.5f}s")


# Timing Merge Sort
start = time.time()
ms_result = merge_sort(data)
print(f"Merge Sort Time: {time.time() - start:.5f}s")


print("\nTop 10 Students:")
for s in ms_result[:10]:
    print(s)
```

Task 2: Bubble Sort with AI Comments

```python
def bubble_sort(arr):
    n = len(arr)
    # Traverse through all array elements
    for i in range(n):
        swapped = False # Optimization: track if a swap happened
```

```
        # Last i elements are already in place, so we skip them
        for j in range(0, n - i - 1):
            # Compare the element with the next one
            if arr[j] > arr[j + 1]:
                # Swap if the element found is greater than the next
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True

        # If no two elements were swapped by inner loop, then break
        if not swapped:
            break
    return arr
```

AI Complexity Analysis:

Best Case: O(n) - Occurs when the array is already sorted.

Average/Worst Case: O(n^2) - Occurs when the array is reverse sorted.

Space Complexity: O(1) - It is an in-place sorting algorithm.

Task 3: Quick Sort and Merge Sort Comparison

Performance Logic

- Random Lists: Both perform at $O(n \log n)$, but Quick Sort is often faster due to lower constant factors.
- Sorted Lists: Merge Sort remains $O(n \log n)$. Standard Quick Sort (if picking the first element as pivot) can degrade to $O(n^2)$.

- **Reverse-Sorted: Similar to sorted, Quick Sort may struggle unless a random pivot is used, while Merge Sort remains stable at $O(n \log n)$.**

Task 4: Inventory Management System

Operation	Recommended Algorithm	Justification
Search by ID	Binary Search	If the list is sorted by ID, Binary Search provides $O(\log n)$ speed, which is significantly faster than linear search for thousands of items.
Search by Name	Hash Map (Dict)	Provides $O(1)$ average time complexity for lookups, allowing staff to find products by name instantly.
Sort by Price	Merge Sort	It is a "stable" sort, meaning if two items have the same price, their relative order remains unchanged. Performance is consistent at $O(n \log n)$.

Implementation Snippet

```
def search_product(inventory_dict, product_name):
    """Fast lookup using a dictionary (Hash Map logic)."""
    return inventory_dict.get(product_name, "Product not found")
```

```python
def sort_by_price(products):
    """Uses Python's Timsort (highly optimized Merge/Insertion hybrid)."""
    return sorted(products, key=lambda x: x['price'])
```

Task 5: Real-Time Stock Data Analysis

Implementation with Heap Sort

```python
import heapq


def heap_sort_stocks(stock_list):
    """Ranks stocks by percentage change using a Heap."""
    # stock_list is a list of tuples: (symbol, %_change)
    # We use a min-heap; for descending, we negate values
    heap = [(-change, symbol) for symbol, change in stock_list]
    heapq.heapify(heap)

    return [heapq.heappop(heap) for _ in range(len(heap))]


# Simulated Data
stocks = [("AAPL", 1.2), ("TSLA", -2.5), ("GOOGL", 0.8), ("MSFT", 1.5)]


# Ranking
ranked = heap_sort_stocks(stocks)
print("Stock Rankings (Highest Gain First):")
for gain, symbol in ranked:
    print(f"{symbol}: {abs(gain)}%")
```

```
# Fast Search via Hash Map
stock_map = {symbol: change for symbol, change in stocks}
print(f"\nSearch MSFT: {stock_map.get('MSFT')}%")
```

Trade-offs & Analysis

- **Heap Sort vs. sorted(): Python's sorted() uses Timsort, which is highly optimized in C. While Heap Sort has a guaranteed $O(n \log n)$, sorted() is usually faster in practice for general datasets.**
- **Hash Maps vs. Manual Search: A dictionary lookup is $O(1)$, whereas searching through a list is $O(n)$. For a FinTech app with thousands of tickers, a Hash Map is the only viable choice for "instant" retrieval.**