Assignment-12.3

Ch .Tejaswi
2303A51944
Batch-27

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 Used:

"Generate a Python program to store student records with Name, Roll Number, CGPA. Implement Quick Sort and Merge Sort to sort students by CGPA in descending order. Display top 10 students and compare runtime."

```
import time
from typing import List, Tuple


class Student:
    def __init__(self, name: str, roll_number: int, cgpa: float):
        self.name = name
        self.roll_number = roll_number
```

```
        self.cgpa = cgpa

    def __repr__(self):
        return f"Student({self.name}, {self.roll_number}, {self.cgpa})"

def quick_sort(students: List[Student], low: int, high: int) ->
List[Student]:
    """Sort students by CGPA in descending order using Quick Sort"""
    if low < high:
        pivot_index = partition(students, low, high)
        quick_sort(students, low, pivot_index - 1)         quick_sort(students, pivot_index + 1, high)

    return students

def partition(students: List[Student], low: int, high: int) -> int:
    pivot = students[high].cgpa
    i = low - 1
    for j in range(low, high):
        if students[j].cgpa > pivot:
            i += 1
            students[i], students[j] = students[j], students[i]
    students[i + 1], students[high] = students[high], students[i + 1]
    return i + 1

def merge_sort(students: List[Student]) -> List[Student]:     """Sort students by CGPA in descending
order using Merge Sort"""
    if len(students) <= 1:
        return students

    mid = len(students) // 2
    left = merge_sort(students[:mid])
    right = merge_sort(students[mid:])     return merge(left, right)
```

```
def merge(left: List[Student], right: List[Student]) -> List[Student]:
    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

def display_top_students(students: List[Student], n: int = 10):
    """Display top N students"""
    print(f"\n{'Rank':<6}{'Name':<20}{'Roll Number':<12}{'CGPA':<8}")      print("-" * 48)

    for i, student in enumerate(students[:n], 1):

print(f"{i:<6}{student.name:<20}{student.roll_number:<12}{student.cgpa:<8.
2f}")

def main():
    students = [
        Student("Alice", 101, 3.85),
        Student("Bob", 102, 3.92),
        Student("Charlie", 103, 3.78),
        Student("Diana", 104, 3.95),
        Student("Eve", 105, 3.88),
        Student("Frank", 106, 3.72),
        Student("Grace", 107, 3.91),
        Student("Henry", 108, 3.82),
        Student("Iris", 109, 3.89),
        Student("Jack", 110, 3.79),
        Student("Karen", 111, 3.94),
        Student("Leo", 112, 3.86),
    ]

    # Quick Sort
    qs_students = [Student(s.name, s.roll_number, s.cgpa) for s in
students]
    start = time.time()
    quick_sort(qs_students, 0, len(qs_students) - 1)
    qs_time = time.time() - start

    # Merge Sort
    ms_students = [Student(s.name, s.roll_number, s.cgpa) for s in
students]
    start = time.time()
    ms_students = merge_sort(ms_students)
    ms_time = time.time() - start

    print("=" * 48)
    print("QUICK SORT RESULTS")
    print("=" * 48)
    display_top_students(qs_students)
    print(f"\nExecution Time: {qs_time:.6f} seconds")

    print("\n" + "=" * 48)
    print("MERGE SORT RESULTS")
    print("=" * 48)
    display_top_students(ms_students)
    print(f"\nExecution Time: {ms_time:.6f} seconds")

    print("\n" + "=" * 48)
    print("PERFORMANCE COMPARISON")
    print("=" * 48)
    print(f"Quick Sort Time: {qs_time:.6f} seconds")
    print(f"Merge Sort Time: {ms_time:.6f} seconds")
    print(f"Difference: {abs(qs_time - ms_time):.6f} seconds")
```

```
if __name__ == "__main__":
    main()
```
OUTPUT:

```
==================================================
QUICK SORT RESULTS
==================================================

Rank  Name           Roll Number CGPA
--------------------------------------------------
1        Diana          104       3.95
2        Karen          111       3.94
3        Bob            102       3.92
4        Grace          107       3.91
5        Iris         109       3.89
6        Eve            105       3.88
7        Leo            112       3.86
8        Alice          101       3.85
9        Henry          108       3.82
10       Jack           110       3.79

Execution Time: 0.000011 seconds

==================================================
MERGE SORT RESULTS
==================================================

Rank  Name           Roll Number CGPA
--------------------------------------------------
1        Diana          104       3.95
2        Karen          111       3.94
3        Bob            102       3.92
4        Grace          107       3.91
5        Iris         109       3.89
6        Eve            105       3.88
7        Leo            112       3.86
8        Alice          101       3.85
9        Henry          108       3.82
10       Jack           110       3.79

Execution Time: 0.000015 seconds
```

==
PERFORMANCE COMPARISON
==

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 Used:
"Write Bubble Sort in Python with inline comments explaining each step and provide time complexity."

CODE:

```
def bubble_sort(arr):
    """
    Bubble Sort Algorithm
    Time Complexity: O(n²) - worst and average case
    Space Complexity: O(1) - sorts in place
    """
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):
        # Flag to optimize: stop if array is already sorted
        swapped = False

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

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

        # If no swaps occurred, array is sorted
        if not swapped:
            break

    return arr



# Example usage if __name__ ==
"__main__":
    numbers = [64, 34, 25, 12, 22, 11, 90]
    print("Original array:", numbers)
    print("Sorted array:", bubble_sort(numbers))
```

OUTPUT:

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

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.

CODE:

Quick Sort:

```
def quick_sort_recursive(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quick_sort_recursive(arr, low, pi-1)
        quick_sort_recursive(arr, pi+1, high)

def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1

    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i+1], arr[high] = arr[high], arr[i+1]
    return i+1
```

Merge Sort:

```

def merge_sort_recursive(arr):
    if len(arr) > 1:

        mid = len(arr)//2          left = arr[:mid]

        right = arr[mid:]

        merge_sort_recursive(left)
        merge_sort_recursive(right)

        i=j=k=0

        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                arr[k]=left[i]
                i+=1
            else:
```

```
            arr[k]=right[j]              j+=1

        k+=1

    while i < len(left):
        arr[k]=left[i]
        i+=1
        k+=1

    while j < len(right):
        arr[k]=right[j]
        j+=1
        k+=1
```

Quick Sort
Best: O(n log n) Average: O(n log n)
Worst: $O(n^2)$

Merge Sort
Best: O(n log n)
Average: O(n log n)
Worst: O(n log n)

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.

Operation: Search by Product ID
Algorithm: Hash Table (Dictionary)
Justification: O(1) lookup time

Operation: Search by Name
Algorithm: Hash Table
Justification: Fast retrieval

Operation: Sort by Price
Algorithm: Merge Sort
Justification: Stable and efficient
Operation: Sort by Quantity
Algorithm: Quick Sort
Justification: Fast average performance CODE:

```
products = {
    101: {"name":"Laptop", "price":50000, "qty":10},
    102: {"name":"Mouse", "price":500, "qty":100},
    103: {"name":"Keyboard", "price":1500, "qty":50}
}

def search_product(pid):
    return products.get(pid, "Not Found")

print(search_product(101))
```

OUTPUT:

```
{'name': 'Laptop', 'price': 50000, 'qty': 10}
```

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.

Prompt Used:
"Simulate stock data and implement Heap Sort and Hash Map search."

CODE:

```
import heapq

stocks = [

    ("AAPL", 150, 160),

    ("GOOG", 2800, 2900),

    ("TSLA", 700, 680),

    ("MSFT", 300, 320)

]

def percent_change(stock):

    return ((stock[2]-stock[1])/stock[1])*100

heap = []

for stock in stocks:

    heapq.heappush(heap, (-percent_change(stock), stock))
```

```
print("Sorted Stocks:")

while heap:

    print(heapq.heappop(heap)[1])

stock_map = {s[0]:s for s in stocks}

def search(symbol):

    return stock_map.get(symbol, "Not Found")

print(search("AAPL"))
```

OUTPUT:

```
Sorted Stocks:

('AAPL', 150, 160)

('MSFT', 300, 320)

('GOOG', 2800, 2900)

('TSLA', 700, 680)

('AAPL', 150, 160)
```