Lab 12: Algorithms with AI Assistance

Name: Nikhil N
HT No: 2303A52477

Assignment: 12.3

Theme: Sorting, Searching, and Algorithm Optimization Using AI Tools

Task 1: Sorting Student Records for Placement Drive

Exact Task Description

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:

- Use GitHub Copilot to generate a program that stores student records (Name, Roll Number, CGPA).
- Implement the following sorting algorithms using AI assistance: Quick Sort, Merge Sort.
- Measure and compare runtime performance for large datasets.
- Write a function to display the top 10 students based on CGPA.

Prompt

> "Write a Python program that generates a dataset of student records (Name, Roll, CGPA). Implement Quick Sort and Merge Sort to sort these records by CGPA in descending order. Measure and compare their runtime, and display the top 10 students."

Code

```python
import time
import random

def generate_students(n):
    return [{"name": f"Student_{i}", "roll": i, "cgpa": round(random.uniform(5.0, 10.0), 2)} for i in range(n)]

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, right = merge_sort(arr[:mid]), 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)
    return result
```


```python
import time
import random

def generate_students(n):
    return [{"name": f"Student_{i}", "roll": i, "cgpa": round(random.uniform(5.0, 10.0), 2)} for i in range(n)]

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, right = merge_sort(arr[:mid]), 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)
    return result
```


Output

```
PS C:\Users\nikhi\Projects\AI Assistant Coding> python
Quick Sort Time: 0.0112s | Merge Sort Time: 0.0160s
Top 3 Students (truncated for brevity):
{'name': 'Student_1107', 'roll': 1107, 'cgpa': 10.0}
{'name': 'Student_1483', 'roll': 1483, 'cgpa': 10.0}
{'name': 'Student_1630', 'roll': 1630, 'cgpa': 10.0}
```

Explanation

I used AI to quickly implement Quick Sort and Merge Sort to rank students by CGPA. Quick Sort proved to be slightly faster in practice due to better memory locality, while Merge Sort guarantees stability. Both efficiently allowed us to slice the array to find the top candidates instantly.

Task 2: Implementing Bubble Sort with AI Comments

Exact Task Description

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

> "Here is a standard Python implementation of Bubble Sort. Please add detailed inline comments explaining the swapping, passes, and early termination logic. Also, include a docstring detailing its time and space complexity."

Code

```python
def bubble_sort(arr):
    """
    Sorts a list in ascending order using the Bubble Sort algorithm.

    Time Complexity:
        - Best Case: O(N) (when the array is already sorted)
        - Worst Case: O(N^2) (when the array is reverse sorted)
        - Average Case: O(N^2)
    Space Complexity: O(1) (sorts in-place without extra memory)
    """
    n = len(arr)
    # Outer loop represents the number of passes needed.
    for i in range(n):
        swapped = False # Flag to track if any swaps happen in this pass

        # Inner loop compares adjacent elements.
        # The '- i' optimization prevents checking already sorted elements at the end.
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                # Swap elements if they are in the wrong order
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True

        # Early termination: If no swaps occurred, the array is already sorted
        if not swapped:
            break

    return arr

print(f"Sorted array: {bubble_sort([64, 34, 25, 12, 22, 11, 90])}")
```

Output

```
PS C:\Users\nikhi\Projects\AI Assistant Coding>
Sorted array: [11, 12, 22, 25, 34, 64, 90]
```

Explanation

Bubble Sort works by repeatedly stepping through the list and swapping adjacent elements if they are in the wrong order. The AI's added comments nicely highlight the swapped flag optimization, which stops the algorithm early if the list is already sorted, saving valuable processing time.

Task 3: Quick Sort and Merge Sort Comparison

Exact Task Description

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 these recursive Quick Sort and Merge Sort functions. Add docstrings explaining their complexities. Then, write code to measure and compare their performance on three list types: completely random, already sorted, and reverse-sorted."

Code

```python
import sys, time, random
sys.setrecursionlimit(2000)

def quick_sort_basic(arr):
    """Time Complexity: Best/Avg O(N log N), Worst O(N^2)."""
    if len(arr) <= 1: return arr
    pivot = arr[0] # Basic pivot selection
    left = [x for x in arr[1:] if x <= pivot]
    right = [x for x in arr[1:] if x > pivot]
    return quick_sort_basic(left) + [pivot] + quick_sort_basic(right)

def merge_sort_basic(arr):
    """Time Complexity: Best/Avg/Worst O(N log N)."""
    if len(arr) <= 1: return arr
    mid = len(arr) // 2
    left, right = merge_sort_basic(arr[:mid]), merge_sort_basic(arr[mid:])
    result, i, j = [], 0, 0
    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
    result.extend(left[i:] or right[j:])
    return result

def test_algo(algo, data):
    start = time.time()
    algo(data)
    return time.time() - start
```

```python
# --- Test Execution ---
sizes = 900 # Kept under 1000 to prevent Quick Sort recursion depth errors
random_data = [random.randint(1, 1000) for _ in range(sizes)]
sorted_data = list(range(sizes))
reverse_data = list(range(sizes, 0, -1))

print(f"Quick Sort -> Random: {test_algo(quick_sort_basic, random_data):.4f}s | Sorted: {test_algo(quick_sort_basic, sorted_data):.4f}s | Rev
print(f"Merge Sort -> Random: {test_algo(merge_sort_basic, random_data):.4f}s | Sorted: {test_algo(merge_sort_basic, sorted_data):.4f}s | Rev
```

Output

```
PS C:\Users\nikhi\Projects\AI Assistant Coding> python Assignment-12.py
Quick Sort -> Random: 0.0016s | Sorted: 0.0410s | Reverse: 0.0396s
Merge Sort -> Random: 0.0021s | Sorted: 0.0014s | Reverse: 0.0014s
```

Explanation

By testing on different data states, we clearly see the AI's docstring warnings come to life. Merge Sort maintained consistent speeds across all datasets, while Quick Sort drastically slowed down on sorted and reverse-sorted data due to poor pivot selection leading to its $O(N^2)$ worst-case behavior.

Task 4: Real-Time Application – Inventory Management System

Exact Task Description

Scenario: A retail store’s inventory system contains thousands of products, each with attributes like product ID, name, price, and stock quantity.

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 efficient search and sort algorithms for a retail inventory system needing ID lookups and price sorting. Provide a Markdown table with Algorithm recommendations and justifications, then write Python functions to implement them."

Code

AI Recommended Algorithms:

Operation	Recommended Algorithm	Justification
Search by ID	Hash Map (Dictionary)	Provides O(1) average time complexity. Instant lookups regardless of inventory size.
Sort by Price	TimSort (sorted())	Python's built-in algorithm. Highly optimized O(N log N) that performs exceptionally well on partially sorted real-world data.

```python
class InventorySystem:
    def __init__(self):
        self.inventory_map = {} # Hash Map for O(1) lookups
        self.products = []

    def add_product(self, pid, name, price, stock):
        product = {"id": pid, "name": name, "price": price, "stock": stock}
        self.inventory_map[pid] = product
        self.products.append(product)

    def search_by_id(self, pid):
        # Hash Map Lookup
        return self.inventory_map.get(pid, "Product not found")

    def sort_by_price(self):
        # TimSort implementation
        return sorted(self.products, key=lambda x: x['price'])

# --- Test Execution ---
inv = InventorySystem()
inv.add_product(101, "Laptop", 850.00, 10)
inv.add_product(102, "Mouse", 25.00, 50)
inv.add_product(103, "Monitor", 200.00, 15)

print(f"Search ID 102: {inv.search_by_id(102)['name']}")
print("Cheapest Product:", inv.sort_by_price()[0]['name'])
```

Output

```
PS C:\Users\nikhi\Projects\AI Assistant Coding> python Assignment-12.py
Search ID 102: Mouse
Cheapest Product: Mouse
```

Explanation

For a real-world inventory, the AI correctly recommended a Hash Map (dictionary) because looking up an ID takes O(1) time—instantaneous, even with millions of items. For sorting, relying on Python's heavily optimized internal TimSort is much more efficient and maintainable than writing a custom Quick Sort from scratch.

Task 5: Real-Time Stock Data Sorting & Searching

Exact Task Description

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.

Task:

- Use GitHub Copilot to fetch or simulate stock price data.
- Implement sorting algorithms to rank stocks by percentage change.
- Implement a search function that retrieves stock data instantly.
- Optimize sorting with Heap Sort and searching with Hash Maps.
- Compare performance with standard library functions.

Prompt

> "Simulate a dataset of stock prices. Calculate percentage change. Implement Heap Sort to rank stocks by % change and a Hash Map for symbol searches. Compare their performance against Python's built-in sorted() and dictionary lookups."

Code

```python
import time, random, heapq

# Simulate Data
symbols = [f"STK{i}" for i in range(10000)]
stock_data = {sym: {"open": random.uniform(10, 500)} for sym in symbols}
for sym in stock_data:
    stock_data[sym]["close"] = stock_data[sym]["open"] * random.uniform(0.9, 1.1)
    stock_data[sym]["change"] = ((stock_data[sym]["close"] - stock_data[sym]["open"]) / stock_data[sym]["open"]) * 100

def heap_sort_stocks(stocks_dict):
    # Min-Heap to sort by percentage change
    heap = [(info["change"], sym, info) for sym, info in stocks_dict.items()]
    heapq.heapify(heap)
    return [heapq.heappop(heap) for _ in range(len(heap))]

# --- Performance Comparison ---
# 1. Searching
start = time.time()
_ = stock_data["STK9999"] # Hash Map lookup
hash_time = time.time() - start

# 2. Sorting via Heap Sort
start = time.time()
heap_sorted = heap_sort_stocks(stock_data)
heap_time = time.time() - start

# 3. Sorting via Python Built-in
start = time.time()
built_in_sorted = sorted(stock_data.items(), key=lambda x: x[1]['change'])
builtin_time = time.time() - start

print(f"Hash Map Lookup Time: {hash_time:.8f}s")
print(f"Heap Sort Time: {heap_time:.4f}s | Built-in Sort Time: {builtin_time:.4f}s")
print(f"Top Loser: {heap_sorted[0][1]} with {heap_sorted[0][0]:.2f}%")
```

Output

```
PS C:\Users\nikhi\Projects\AI Assistant Coding> python Assignment-12.py
Hash Map Lookup Time: 0.00000095s
Heap Sort Time: 0.0066s | Built-in Sort Time: 0.0041s
Top Loser: STK5748 with -10.00%
```

Explanation

Heap Sort successfully ranked the stock data without recursive overhead, making it memory efficient. However, the performance comparison clearly shows that Python's built-in sorting is faster because it is implemented in C. The Hash Map search time was practically zero, proving it is the ultimate choice for symbol lookups.