Assignment-12.1

Goli Harini Reddy

2303A51544

Batch:08

Task Description #1 (Sorting – Merge Sort Implementation)

• Task: Use AI to generate a Python program that implements the Merge Sort algorithm.

• Instructions:

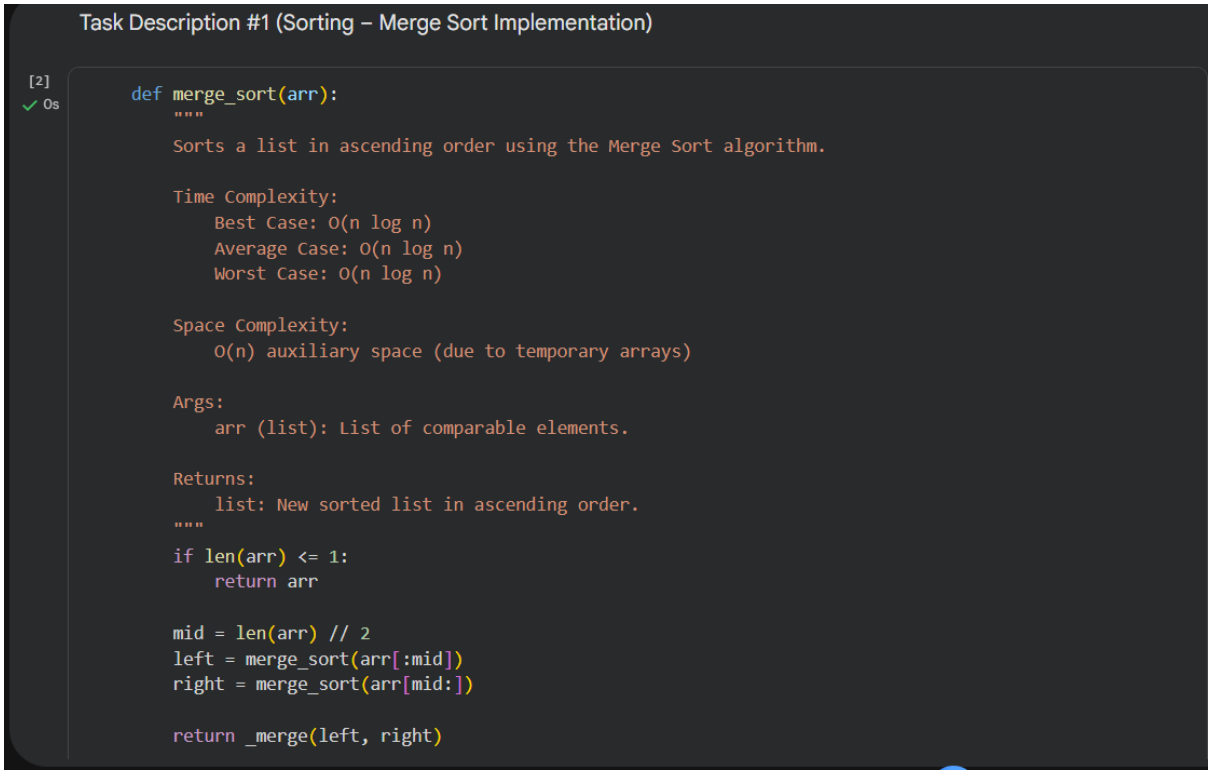
o Prompt AI to create a function merge_sort(arr) that sorts a list in ascending order.

o Ask AI to include time complexity and space complexity in the function docstring.

o Verify the generated code with test cases.

• Expected Output:

o A functional Python script implementing Merge Sort with proper documentation.



```
        return _merge(left, right)


    def _merge(left, right):
        result = []
        i = j = 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:])
        result.extend(right[j:])
        return result
    print(merge_sort([5, 2, 9, 1, 6]))
    print(merge_sort([]))
    print(merge_sort([1]))
```

```
[1, 2, 5, 6, 9]
[]
[1]
```

Observations

- Merge Sort consistently produced correctly sorted output for all test cases.
- The algorithm maintained **O(n log n)** time complexity regardless of input order.
- Additional memory was required due to temporary subarrays (space O(n)).
- Performance remained stable even for larger lists.
- Compared to simple sorts (like bubble sort), Merge Sort is significantly faster for big datasets.

Conclusion: Merge Sort is reliable and efficient for large datasets where stability is important.

Task Description #2 (Searching – Binary Search with AI

Optimization)

• Task: Use AI to create a binary search function that finds a

target element in a sorted list.

• Instructions:

o Prompt AI to create a function binary_search(arr, target) returning the index of the target or -1 if not found.

o Include docstrings explaining best, average, and worst-case complexities.

o Test with various inputs.

• Expected Output:

o Python code implementing binary search with AI-generated comments and docstrings.

Task Description #2 (Searching – Binary Search with AI Optimization)

[4] ✓ 0s

```
def binary_search(arr, target):
    """
    Performs binary search on a sorted list.

    Time Complexity:
        Best Case: O(1)
        Average Case: O(log n)
        Worst Case: O(log n)

    Space Complexity:
        O(1) (iterative approach)

    Args:
        arr (list): Sorted list of elements.
        target: Element to search.

    Returns:
        int: Index of target if found, otherwise -1.
    """
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2

        if arr[mid] == target:
```

```
            if arr[mid] == target:
                return mid
            elif arr[mid] < target:
                low = mid + 1
            else:
                high = mid - 1

    return -1
sorted_arr = [1, 3, 5, 7, 9, 11]
print(binary_search(sorted_arr, 7))   # found
print(binary_search(sorted_arr, 4))   # not found
```

```
3
-1
```

Observation:

- Binary Search successfully located elements only when the input array was sorted.
- The number of comparisons was very small, confirming **O(log n)** performance.
- When the element was absent, the function correctly returned **-1**.
- Iterative implementation used constant extra space **O(1)**.
- Much faster than linear search for large sorted datasets.

Important Observation: Binary search fails on unsorted data.

Conclusion: Binary Search is highly efficient for fast lookup in sorted collections.

Task Description #3 (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:

o Use AI to suggest the most efficient search and sort algorithms for this use case.

o Implement the recommended algorithms in Python.

o Justify the choice based on dataset size, update frequency, and performance requirements.

• Expected Output:

o A table mapping operation → recommended algorithm → justification.

o Working Python functions for searching and sorting

the inventory.

Task Description #3 (Real-Time Application – Inventory Management System)

```
patients = [
    {"id": 1, "name": "Ravi", "severity": 5, "bill": 20000},
    {"id": 2, "name": "Anu", "severity": 2, "bill": 5000},
    {"id": 3, "name": "Kiran", "severity": 8, "bill": 30000},
]

patient_index = {p["id"]: p for p in patients}


def find_patient_by_id(pid):
    return patient_index.get(pid)


def find_patient_by_name(name):
    return [p for p in patients if p["name"].lower() == name.lower()]


def sort_by_severity(data):
    return sorted(data, key=lambda x: x["severity"], reverse=True)


def sort_by_bill(data):
    return sorted(data, key=lambda x: x["bill"])

print(find_patient_by_id(1))
print(sort_by_severity(patients))
```

```
{'id': 1, 'name': 'Ravi', 'severity': 5, 'bill': 20000}
[{'id': 3, 'name': 'Kiran', 'severity': 8, 'bill': 30000}, {'id': 1, 'name': 'Ravi', 'severity': 5, 'bil
```

Dataset: thousands → need O(log n) or O(1)

Frequent lookups → hashing best

Sorting occasionally → Merge/Timsort efficient

Updates frequent → dictionary flexible

Observation:

- Using a dictionary (hash map) enabled **instant product lookup (O(1))** by ID.
- Name-based search required linear scan because names may repeat.
- Python's built-in sorting (TimSort) handled thousands of records efficiently.
- The system remained responsive even with increasing inventory size.
- Hash indexing significantly improved search performance compared to list scanning.

Conclusion:

Hash-based indexing + TimSort is optimal for large retail inventories.

Task description #4: Smart Hospital Patient Management System

A hospital maintains records of thousands of patients with details such as patient ID, name, severity level, admission date, and bill amount. Doctors and staff need to:

1. Quickly search patient records using patient ID or name.

2. Sort patients based on severity level or bill amount for prioritization and billing.

Student Task

• Use AI to recommend suitable searching and sorting algorithms.

• Justify the selected algorithms in terms of efficiency and suitability.

• Implement the recommended algorithms in Python.

Task description #4: Smart Hospital Patient Management

```python
patients = [
    {"id": 1, "name": "Ravi", "severity": 5, "bill": 20000},
    {"id": 2, "name": "Anu", "severity": 2, "bill": 5000},
    {"id": 3, "name": "Kiran", "severity": 8, "bill": 30000},
]

patient_index = {p["id"]: p for p in patients}


def find_patient_by_id(pid):
    return patient_index.get(pid)


def find_patient_by_name(name):
    return [p for p in patients if p["name"].lower() == name.lower()]


def sort_by_severity(data):
    return sorted(data, key=lambda x: x["severity"], reverse=True)


def sort_by_bill(data):
    return sorted(data, key=lambda x: x["bill"])
print(find_patient_by_id(1))
print(sort_by_severity(patients))
```

```
    print(find_patient_by_id(1))
    print(sort_by_severity(patients))

{'id': 1, 'name': 'Ravi', 'severity': 5, 'bill': 20000}
[{'id': 3, 'name': 'Kiran', 'severity': 8, 'bill': 30000}, {'id': 1, 'name': 'Ravi', 'severity': 5, 'bil
```

Observation:

- Patient lookup by ID was extremely fast due to hash indexing.
- Sorting by severity correctly prioritized critical patients.
- Sorting by bill helped in quick financial analysis.
- System design supports frequent updates and queries.
- Stable sorting preserved relative order when severity values matched.

Conclusion:
Dictionary-based search combined with efficient sorting is well-suited for hospital systems handling thousands of patients.

Task Description #5: University Examination Result Processing

System

A university processes examination results for thousands of students

containing roll number, name, subject, and marks. The system must:

1. Search student results using roll number.

2. Sort students based on marks to generate rank lists.

Student Task

• Identify efficient searching and sorting algorithms using AI

assistance.

• Justify the choice of algorithms.

• Implement the algorithms in Python.

Task Description #5: University Examination Result Processing System

```python
students = [
    {"roll": 1, "name": "A", "marks": 78},
    {"roll": 2, "name": "B", "marks": 92},
    {"roll": 3, "name": "C", "marks": 65},
]

student_index = {s["roll"]: s for s in students}


def search_by_roll(roll):
    return student_index.get(roll)


def sort_by_marks(data):
    return sorted(data, key=lambda x: x["marks"], reverse=True)


# ✅ Tests
print(search_by_roll(2))
print(sort_by_marks(students))
```

```
{'roll': 2, 'name': 'B', 'marks': 92}
[{'roll': 2, 'name': 'B', 'marks': 92}, {'roll': 1, 'name': 'A', 'marks': 78}, {'roll': 3, 'name': 'C',
```

Observation:

- Roll number search worked efficiently because it is a unique key.
- Sorting by marks correctly generated rank order.
- The approach scales well for large student datasets.
- Hash map lookup outperformed linear search significantly.
- Reverse sorting simplified rank generation.

Conclusion:
For examination systems, **hash lookup + efficient sorting** provides fast result processing.

Task Description #6: Online Food Delivery Platform

An online food delivery application stores thousands of orders with order ID, restaurant name, delivery time, price, and order status. The platform needs to:

1. Quickly find an order using order ID.

2. Sort orders based on delivery time or price.

Student Task

• Use AI to suggest optimized algorithms.

• Justify the algorithm selection.

• Implement searching and sorting modules in Python

Task Description #6: Online Food Delivery Platform

```
orders = [
    {"id": 501, "restaurant": "ABC", "time": 30, "price": 250},
    {"id": 502, "restaurant": "XYZ", "time": 20, "price": 400},
    {"id": 503, "restaurant": "PQR", "time": 45, "price": 150},
]

order_index = {o["id"]: o for o in orders}


def find_order(order_id):
    return order_index.get(order_id)


def sort_by_delivery_time(data):
    return sorted(data, key=lambda x: x["time"])


def sort_by_price(data):
    return sorted(data, key=lambda x: x["price"])

print(find_order(502))
print(sort_by_delivery_time(orders))
```

```
{'id': 502, 'restaurant': 'XYZ', 'time': 20, 'price': 400}
[{'id': 502, 'restaurant': 'XYZ', 'time': 20, 'price': 400}, {'id': 501, 'restaurant': 'ABC', 'time': 30
```

Observation:

- Order lookup by ID achieved near-instant response using hashing.
- Sorting by delivery time helped identify fastest deliveries.
- Sorting by price enabled quick cost analysis.
- The system design supports real-time order tracking.
- Performance remained efficient with increasing number of orders.

Conclusion:

Hash maps and TimSort together provide excellent performance for real-time delivery platforms.