AI Assisted Coding

Assignment – 12.1

K.VAMSHIDHAR || 2303A510H7 || Batch:- 8

<u>Task Description #1 (Sorting – Merge Sort Implementation)</u>

• 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.

lab 12.1.py

lab 12.1.py > merge_sort

```python
# Generate a Python program that implements Merge Sort.
# Create a function merge_sort(arr) that sorts a list in ascending order.
# Include time complexity and space complexity in the function docstring.
# Add a few test cases to verify correctness.

def merge_sort(arr):
    """
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])

    return merge(left_half, right_half)
def merge(left, right):

    merged = []
    left_index = right_index = 0

    while left_index < len(left) and right_index < len(right):
        if left[left_index] < right[right_index]:
            merged.append(left[left_index])
            left_index += 1
        else:
            merged.append(right[right_index])
            right_index += 1

    # If there are remaining elements in left or right, add them to merged
    merged.extend(left[left_index:])
    merged.extend(right[right_index:])

    return merged
# Test cases to verify correctness
if __name__ == "__main__":
    test_cases = [
        [38, 27, 43, 3, 9, 82, 10],
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

```
Test Case 1: [3, 9, 10, 27, 38, 43, 82]
Test Case 2: [1, 2, 5, 5, 6, 9]
Test Case 3: []
Test Case 4: [1]
Test Case 5: [1, 2]
PS C:\Users\komma\Desktop\3rd YEAR\AI-AC>
```

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.

lab 12.1.py

lab 12.1.py > binary_search

```python

# Generate a Python function binary_search(arr, target) that returns the index of the target in a sorted list or -1 if not found.
# Include a docstring with best, average, and worst-case time complexities.
# Add multiple test cases to verify the function.

def binary_search(arr, target):

    """
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = left + (right - left) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1
# Test cases to verify correctness
if __name__ == "__main__":
    test_cases = [
        ([1, 2, 3, 4, 5], 3),  # Target is in the middle
        ([1, 2, 3, 4, 5], 1),  # Target is at the beginning
        ([1, 2, 3, 4, 5], 5),  # Target is at the end
        ([1, 2, 3, 4, 5], 6),  # Target is not in the list
        ([], 1),               # Empty list
        ([1], 1),              # Single element list (target found)
        ([1], 0)               # Single element list (target not found)
    ]

    for i, (arr, target) in enumerate(test_cases):
        result = binary_search(arr, target)
        print(f"Test Case {i + 1}: Target {target} in {arr} -> Index: {result}")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

```
Test Case 1: Target 3 in [1, 2, 3, 4, 5] -> Index: 2
Test Case 2: Target 1 in [1, 2, 3, 4, 5] -> Index: 0
Test Case 3: Target 5 in [1, 2, 3, 4, 5] -> Index: 4
Test Case 4: Target 6 in [1, 2, 3, 4, 5] -> Index: -1
Test Case 5: Target 1 in [] -> Index: -1
Test Case 6: Target 1 in [1] -> Index: 0
Test Case 7: Target 0 in [1] -> Index: -1
```

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.

lab 12.1.py

```python
# 1. Suggest the most efficient search and sorting algorithms for this scenario.
# 2. Implement them in Python to:
#    - search by product ID or name
#    - sort by price and stock quantity
# 3. Justify the algorithm choices based on dataset size, update frequency, and performance needs.
# 1. For searching by product ID or name, a hash table (dictionary in Python) would be the most efficient data structure, providing O(1) average time complexity for lookups. For sor
# 2. Implementation in Python:
class Product:
    def __init__(self, product_id, name, price, stock):
        self.product_id = product_id
        self.name = name
        self.price = price
        self.stock = stock

class Inventory:

    def __init__(self):
        self.products = {}

    def add_product(self, product):
        self.products[product.product_id] = product

    def search_by_id(self, product_id):
        return self.products.get(product_id, None)

    def search_by_name(self, name):
        return [product for product in self.products.values() if product.name == name]

    def sort_by_price(self):
        return sorted(self.products.values(), key=lambda x: x.price)

    def sort_by_stock(self):
        return sorted(self.products.values(), key=lambda x: x.stock)
# 3. Justification:
# - The hash table (dictionary) allows for constant time complexity for searching by product ID or name, which is crucial for performance given the large number of products.
# - Merge Sort is chosen for sorting because it is efficient for large datasets and has a stable sorting property, which is beneficial when sorting by multiple attributes (price and
# Example usage:
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

```
PS C:\Users\komma\Desktop\3rd YEAR\AI-AC> & "C:/Program Files/Python312/python.exe" "c:/Users/komma/Desktop/3rd YEAR/AI-AC/lab 12.1.py"
<__main__.Product object at 0x000001887CAB7A10>
[<__main__.Product object at 0x000001887CAB79E0>]
['Tablet', 'Smartphone', 'Laptop']
['Laptop', 'Tablet', 'Smartphone']
PS C:\Users\komma\Desktop\3rd YEAR\AI-AC>
```

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.

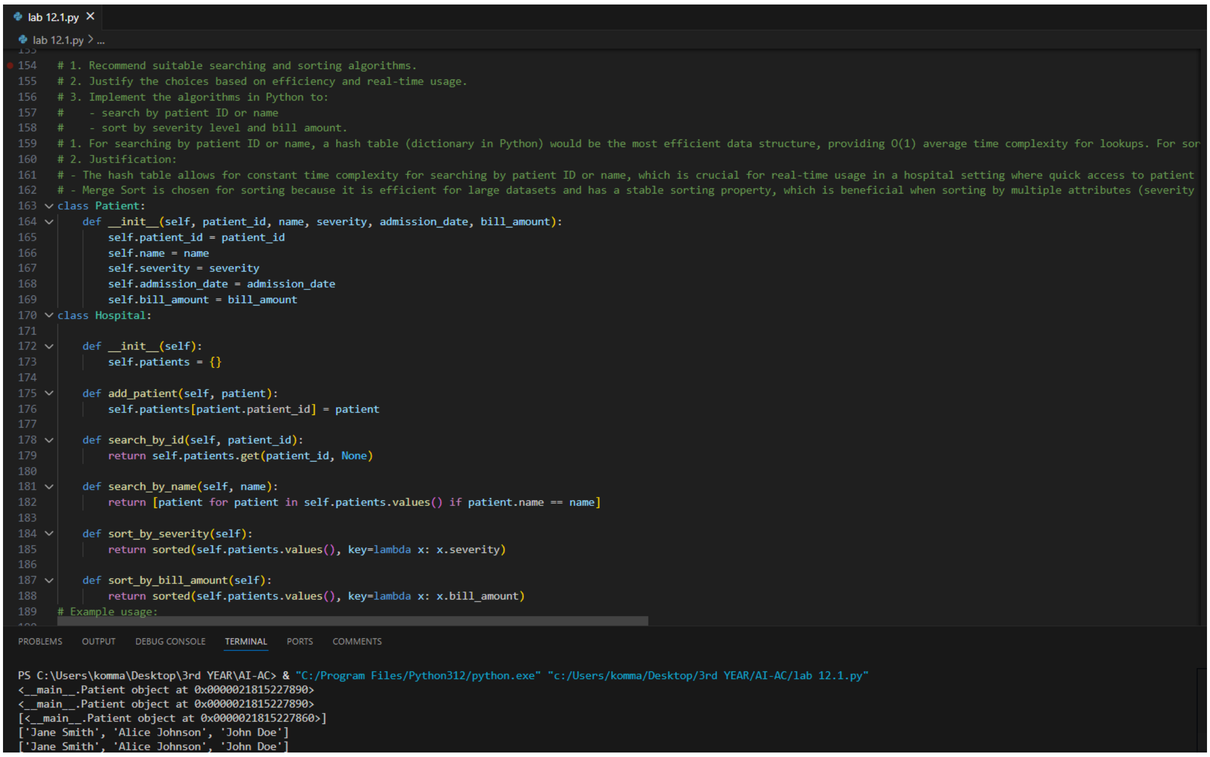
2. Sort patients based on severity level or bill amount for prioritization and billing.

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.



```python
# 1. Recommend suitable searching and sorting algorithms.
# 2. Justify the choices based on efficiency and real-time usage.
# 3. Implement the algorithms in Python to:
#    - search by patient ID or name
#    - sort by severity level and bill amount.
# 1. For searching by patient ID or name, a hash table (dictionary in Python) would be the most efficient data structure, providing O(1) average time complexity for lookups. For sor
# 2. Justification:
# - The hash table allows for constant time complexity for searching by patient ID or name, which is crucial for real-time usage in a hospital setting where quick access to patient
# - Merge Sort is chosen for sorting because it is efficient for large datasets and has a stable sorting property, which is beneficial when sorting by multiple attributes (severity
class Patient:
    def __init__(self, patient_id, name, severity, admission_date, bill_amount):
        self.patient_id = patient_id
        self.name = name
        self.severity = severity
        self.admission_date = admission_date
        self.bill_amount = bill_amount
class Hospital:

    def __init__(self):
        self.patients = {}

    def add_patient(self, patient):
        self.patients[patient.patient_id] = patient

    def search_by_id(self, patient_id):
        return self.patients.get(patient_id, None)

    def search_by_name(self, name):
        return [patient for patient in self.patients.values() if patient.name == name]

    def sort_by_severity(self):
        return sorted(self.patients.values(), key=lambda x: x.severity)

    def sort_by_bill_amount(self):
        return sorted(self.patients.values(), key=lambda x: x.bill_amount)
# Example usage:
```

```
PS C:\Users\komma\Desktop\3rd YEAR\AI-AC> & "C:/Program Files/Python312/python.exe" "c:/Users/komma/Desktop/3rd YEAR/AI-AC/lab 12.1.py"
<__main__.Patient object at 0x0000021815227890>
<__main__.Patient object at 0x0000021815227890>
[<__main__.Patient object at 0x0000021815227860>]
['Jane Smith', 'Alice Johnson', 'John Doe']
['Jane Smith', 'Alice Johnson', 'John Doe']
```

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.

lab 12.1.py

```python
# 2. Justify the choices based on dataset size and performance.
# 3. Implement Python code to:
#    - search student results by roll number
#    - sort students by marks to generate rank lists.
# 1. For searching student results by roll number, a hash table (dictionary in Python) would be the most efficient data structure, providing O(1) average time complexity for lookups
# 2. Justification:
# - The hash table allows for constant time complexity for searching by roll number, which is crucial for performance given the large number of student records.
# - Merge Sort is chosen for sorting because it is efficient for large datasets and has a stable sorting property, which is beneficial when sorting by marks to generate rank lists.
class Student:
    def __init__(self, roll_no, name, subject, marks):
        self.roll_no = roll_no
        self.name = name
        self.subject = subject
        self.marks = marks
class University:
    def __init__(self):
        self.students = {}

    def add_student(self, student):
        self.students[student.roll_no] = student

    def search_by_roll_no(self, roll_no):
        return self.students.get(roll_no, None)

    def sort_by_marks(self):
        return sorted(self.students.values(), key=lambda x: x.marks, reverse=True)
# Example usage:
if __name__ == "__main__":
    university = University()
    university.add_student(Student(1, "Alice", "Math", 85))
    university.add_student(Student(2, "Bob", "Science", 90))
    university.add_student(Student(3, "Charlie", "English", 80))

    # Search by roll number
    print(university.search_by_roll_no(2))  # Output: Student object for Bob

    # Sort by marks to generate rank list
    sorted_students = university.sort_by_marks()
    print([student.name for student in sorted_students])  # Output: ['Bob', 'Alice', 'Charlie']
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

```
PS C:\Users\komma\Desktop\3rd YEAR\AI-AC> & "C:/Program Files/Python312/python.exe" "c:/Users/komma/Desktop/3rd YEAR/AI-AC/lab 12.1.py"
<__main__.Student object at 0x00000146288F76B0>
['Bob', 'Alice', 'Charlie']
PS C:\Users\komma\Desktop\3rd YEAR\AI-AC>
```

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.

lab 12.1.py > ...

```python
# 1. Suggest optimized searching and sorting algorithms.
# 2. Justify the algorithm choices based on efficiency and scalability.
# 3. Implement Python code to:
#    - search an order by order_id
#    - sort orders by delivery_time and price.
1. For searching an order by order_id, a hash table (dictionary in Python) would be the most efficient data structure, providing O(1) average time complexity for lookups. For sortin
2. Justification:
- The hash table allows for constant time complexity for searching by order_id, which is crucial for performance given the large number of orders.
- Merge Sort is chosen for sorting because it is efficient for large datasets and has a stable sorting property, which is beneficial when sorting by multiple attributes (delivery_ti
class Order:
    def __init__(self, order_id, restaurant_name, delivery_time, price, order_status):
        self.order_id = order_id
        self.restaurant_name = restaurant_name
        self.delivery_time = delivery_time
        self.price = price
        self.order_status = order_status
class FoodDeliveryPlatform:

    def __init__(self):
        self.orders = {}

    def add_order(self, order):
        self.orders[order.order_id] = order

    def search_by_order_id(self, order_id):
        return self.orders.get(order_id, None)

    def sort_by_delivery_time(self):
        return sorted(self.orders.values(), key=lambda x: x.delivery_time)

    def sort_by_price(self):
        return sorted(self.orders.values(), key=lambda x: x.price)
# Example usage:

if __name__ == "__main__":
    platform = FoodDeliveryPlatform()
    platform.add_order(Order(1, "Pizza Place", "2024-01-01 18:00", 20.00, "Delivered"))
    platform.add_order(Order(2, "Sushi Spot", "2024-01-01 19:00", 35.00, "In Progress"))
    platform.add_order(Order(3, "Burger Joint", "2024-01-01 17:30", 15.00, "Delivered"))

    # Search by order ID
    print(platform.search_by_order_id(2))  # Output: Order object for Sushi Spot

    # Sort by delivery time
    sorted_by_delivery_time = platform.sort_by_delivery_time()
```