ASSIGNMENT-12.1

Name: D. Sravika Reddy Hall-Ticket No: 2403a510d0

Batch No: 05 Course: AI Assisted Coding

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.

#Prompt: generate a Python program that implements the Merge Sort algorithm.

Code:

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

    Time Complexity: O(n log n)
    Space Complexity: O(n)

    Args:
        arr (list): The list of elements to sort.

    Returns:
        list: A new sorted list.
    """
    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):
    """
    Merges two sorted lists into one sorted list.
    """
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
```

```python
    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

# Test cases
assert merge_sort([5, 2, 9, 1, 5, 6]) == [1, 2, 5, 5, 6, 9]
assert merge_sort([]) == []
assert merge_sort([3]) == [3]
assert merge_sort([10, -1, 2, 5, 0]) == [-1, 0, 2, 5, 10]

print("All test cases passed.")
```

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS                                    Python

PS C:\Users\sravi\OneDrive\Desktop\AIAC> & C:/ProgramData/anaconda3/python.exe c:/Users/sravi/OneDrive/Desktop/AIAC/12.
1-01.py
All test cases passed.
PS C:\Users\sravi\OneDrive\Desktop\AIAC>
```

Explanation:

- The code implements Merge Sort with a helper merge function.
- The docstring includes time and space complexity.
- Four assert test cases verify correctness for various input scenarios.

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.

#Prompt: create a binary search function that finds a target element in a sorted list.

Code:

12.1-02.py > binary_search

```python
def binary_search(arr, target):
    """
    Performs binary search to find the index of target in a sorted list.

    Best Case Complexity: O(1) (target is at the middle)
    Average Case Complexity: O(log n)
    Worst Case Complexity: O(log n)

    Args:
        arr (list): A sorted list of elements.
        target: The element to search for.

    Returns:
        int: Index of target if found, else -1.
    """
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2  # Find the middle index
        if arr[mid] == target:
            return mid  # Target found
        elif arr[mid] < target:
            left = mid + 1  # Search right half
        else:
            right = mid - 1  # Search left half
    return -1  # Target not found

# Test cases
assert binary_search([1, 2, 3, 4, 5], 3) == 2
assert binary_search([10, 20, 30, 40, 50], 10) == 0
assert binary_search([10, 20, 30, 40, 50], 50) == 4
assert binary_search([1, 2, 3, 4, 5], 6) == -1

print("All test cases passed.")
```

Output:

```
PS C:\Users\sravi\OneDrive\Desktop\AIAC> & C:/ProgramData/anaconda3/python.exe c:/Users/sravi/OneDrive/Desktop/AIAC/12.1-02.py
All test cases passed.
PS C:\Users\sravi\OneDrive\Desktop\AIAC>
```

Explanation:

- The function uses a loop to repeatedly divide the search interval in half.
- Returns the index if the target is found, otherwise returns -1.
- Handles edge cases and works for any sorted list.

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

Code:

```
from tabulate import tabulate
operations = [
    ["Search by Product ID", "Hash Table Lookup", "O(1) average; ideal for large datasets and frequent updates"],
    ["Search by Name", "Binary Search (on sorted list)", "O(log n); efficient if names are indexed or sorted"],
    ["Sort by Price/Quantity", "Timsort (Python's built-in sorted)", "O(n log n); stable and fast for large lists"]
]
print(tabulate(operations, headers=["Operation", "Recommended Algorithm", "Justification"], tablefmt="github"))
class Product:
    def __init__(self, product_id, name, price, quantity):
        self.product_id = product_id
        self.name = name
        self.price = price
        self.quantity = quantity
    def __repr__(self):
        return f"Product({self.product_id}, {self.name}, {self.price}, {self.quantity})"
class Inventory:
    def __init__(self):
        self.products = []
        self.id_map = {}
    def add_product(self, product):
        self.products.append(product)
        self.id_map[product.product_id] = product
    def search_by_id(self, product_id):
        """O(1) average time using hash table."""
        return self.id_map.get(product_id, None)
    def search_by_name(self, name):
        """O(n) linear search; can be optimized with sorted list and binary search."""
        for product in self.products:
            if product.name == name:
                return product
        return None
    def sort_by_price(self):
        """Sort products by price using Python's built-in sorted (Timsort)."""
        return sorted(self.products, key=lambda p: p.price)
    def sort_by_quantity(self):
        """Sort products by quantity using Python's built-in sorted (Timsort)."""
        return sorted(self.products, key=lambda p: p.quantity)
```

```python
# Sample usage and output
inventory = Inventory()
inventory.add_product(Product(101, "Apple", 1.2, 50))
inventory.add_product(Product(102, "Banana", 0.8, 100))
inventory.add_product(Product(103, "Orange", 1.5, 80))
print(inventory.search_by_id(102))
print(inventory.search_by_name("Apple"))
print(inventory.sort_by_price())
print(inventory.sort_by_quantity())
```

Output:

```
PS C:\Users\sravi\OneDrive\Desktop\AIAC> & C:/ProgramData/anaconda3/python.exe c:/Users/sravi/OneDrive/Desktop/AIAC/12.1-03.py
| Operation             | Recommended Algorithm              | Justification                                          |
|-----------------------|------------------------------------|--------------------------------------------------------|
| Search by Product ID  | Hash Table Lookup                  | O(1) average; ideal for large datasets and frequent updates |
| Search by Name        | Binary Search (on sorted list)     | O(log n); efficient if names are indexed or sorted     |
| Sort by Price/Quantity | Timsort (Python's built-in sorted) | O(n log n); stable and fast for large lists           |
Product(102, Banana, 0.8, 100)
Product(101, Apple, 1.2, 50)
[Product(102, Banana, 0.8, 100), Product(101, Apple, 1.2, 50), Product(103, Orange, 1.5, 80)]
[Product(101, Apple, 1.2, 50), Product(103, Orange, 1.5, 80), Product(102, Banana, 0.8, 100)]
PS C:\Users\sravi\OneDrive\Desktop\AIAC>
```

Explanation:

- **Search by Product ID:** Uses a hash table for O(1) lookup, ideal for large, frequently updated datasets.
- **Search by Name:** Linear search shown; binary search possible if names are sorted/indexed.
- **Sort by Price/Quantity:** Uses Python's built-in sorted (Timsort), efficient for large lists.
- The table maps each operation to the recommended algorithm and justification.
- The code demonstrates searching and sorting in a sample inventory.