ASSIGNMENT-12.1

Name : P.Reshmitha Reddy

Rollno : 2403A510A7

Batch : 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.

• 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)
    """
    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):
    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

# Test cases
if __name__ == "__main__":
    test_cases = [
        [],
        [1],
        [5, 2, 9, 1, 5, 6],
        [3, 2, 1],
        [10, -1, 2, 5, 0],
        [1, 2, 3, 4, 5]
    ]
    for arr in test_cases:
        print(f"Original: {arr} -> Sorted: {merge_sort(arr)}")
```

Output :

```
PS C:\Users\Reshm\Desktop\AIAC> & C:/Users/Reshm/anaconda3/python.exe c:/Users/Reshm/Desktop
/AIAC/12.1.py
Original: [] -> Sorted: []
Original: [1] -> Sorted: [1]
Original: [5, 2, 9, 1, 5, 6] -> Sorted: [1, 2, 5, 5, 6, 9]
Original: [3, 2, 1] -> Sorted: [1, 2, 3]
Original: [10, -1, 2, 5, 0] -> Sorted: [-1, 0, 2, 5, 10]
Original: [1, 2, 3, 4, 5] -> Sorted: [1, 2, 3, 4, 5]
PS C:\Users\Reshm\Desktop\AIAC>
```

Observation :

The code correctly implements the Merge Sort algorithm using recursion and a helper merge function.The function sorts lists in ascending order and works for empty lists, single-element lists, and lists with duplicate or negative values.The docstring clearly states the time complexity (O(n log n)) and space complexity (O(n)).Test cases demonstrate the correctness of the implementation for various input scenarios.The code is readable, modular, and follows good Python practices.

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 :

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

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

    Returns:
        int: Index of target if found, else -1.
    """
    left, right = 0, len(arr) - 1

    # Binary search loop
    while left <= right:
        mid = left + (right - left) // 2  # Prevents overflow
        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
```

```python
if __name__ == "__main__":
    test_cases = [
        ([1, 2, 3, 4, 5], 3),
        ([10, 20, 30, 40, 50], 25),
        ([], 1),
        ([5], 5),
        ([2, 4, 6, 8, 10], 10),
        ([1, 3, 5, 7, 9], 1)
    ]
    for arr, target in test_cases:
        print(f"Array: {arr}, Target: {target} -> Index: {binary_search(arr, target)}")
```

Output :

```
PS C:\Users\Reshm\Desktop\AIAC> & C:/Users/Reshm/anaconda3/python.exe c:/Users/Reshm/Desktop/
AIAC/12.1.py
Array: [1, 2, 3, 4, 5], Target: 3 -> Index: 2
Array: [10, 20, 30, 40, 50], Target: 25 -> Index: -1
Array: [], Target: 1 -> Index: -1
Array: [5], Target: 5 -> Index: 0
Array: [2, 4, 6, 8, 10], Target: 10 -> Index: 4
Array: [1, 3, 5, 7, 9], Target: 1 -> Index: 0
PS C:\Users\Reshm\Desktop\AIAC> 
```

Observation :

The code implements binary search efficiently for sorted lists. The docstring clearly explains best (O(1)), average, and worst-case (O(log n)) time complexities.The function returns the index of the target if found, or -1 if not found.Test cases cover various scenarios: empty list, single-element list, target present/absent, and edge values.The code is well-commented, readable, and demonstrates correct binary search logic.

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.

Prompt :# suggest the most efficient search and sort

algorithms for this use case.

Code :

```python
from typing import List, Dict

class Product:
    def __init__(self, product_id: str, name: str, price: float, quantity: int):
        self.product_id = product_id
        self.name = name
        self.price = price
        self.quantity = quantity

    def __repr__(self):
        return f"Product(ID={self.product_id}, Name={self.name}, Price={self.price}, Qty={self.quantity})"

class Inventory:
    def __init__(self, products: List[Product]):
        # Hash maps for fast lookup
        self.id_map: Dict[str, Product] = {p.product_id: p for p in products}
        self.name_map: Dict[str, Product] = {p.name: p for p in products}
        self.products = products

    def search_by_id(self, product_id: str):
        """O(1) average time using hash map."""
        return self.id_map.get(product_id, None)
    def search_by_name(self, name: str):
        """O(1) average time if names are unique."""
        return self.name_map.get(name, None)

    def sort_by_price(self, reverse=False):
        """O(n log n) using Timsort."""
        return sorted(self.products, key=lambda p: p.price, reverse=reverse)

    def sort_by_quantity(self, reverse=False):
        """O(n log n) using Timsort."""
        return sorted(self.products, key=lambda p: p.quantity, reverse=reverse)

# Example usage
if __name__ == "__main__":
    inventory = Inventory([
        Product("101", "Apple", 0.5, 100),
        Product("102", "Banana", 0.3, 150),
        Product("103", "Orange", 0.7, 80),
        Product("104", "Mango", 1.2, 50)
    ])

    print("Search by ID '102':", inventory.search_by_id("102"))
    print("Search by Name 'Mango':", inventory.search_by_name("Mango"))
    print("Sort by Price:", inventory.sort_by_price())
    print("Sort by Quantity (descending):", inventory.sort_by_quantity(reverse=True))

```

Output :

```
PS C:\Users\Reshm\Desktop\AIAC> & C:/Users/Reshm/anaconda3/python.exe c:/Users/Reshm/Desktop/AIAC/12.1.py
Search by ID '102': Product(ID=102, Name=Banana, Price=0.3, Qty=150)
Search by Name 'Mango': Product(ID=104, Name=Mango, Price=1.2, Qty=50)
Sort by Price: [Product(ID=102, Name=Banana, Price=0.3, Qty=150), Product(ID=101, Name=Apple, Price=0.5, Qty=100), Product
(ID=103, Name=Orange, Price=0.7, Qty=80), Product(ID=104, Name=Mango, Price=1.2, Qty=50)]
Sort by Quantity (descending): [Product(ID=102, Name=Banana, Price=0.3, Qty=150), Product(ID=101, Name=Apple, Price=0.5, Q
ty=100), Product(ID=103, Name=Orange, Price=0.7, Qty=80), Product(ID=104, Name=Mango, Price=1.2, Qty=50)]
PS C:\Users\Reshm\Desktop\AIAC>
```

Observation :

The code models products with a Product class and manages them in an Inventory class.by product ID or name uses maps (dictionaries), providing fast O(1) average lookup—ideal for large datasets.Sorting by price or quantity uses Python's built-in sorted() (Timsort), which is efficient (O(n log n)) and stable.The code is modular, readable, and demonstrates the recommended algorithms for each operation.Test cases show correct search and sort results, validating the implementation for real-world inventory management.