LAB ASSIGNMENT-12.1

Name : G.Sudeeksha

Roll No : 2403A510E0

Course : AI Assisted Coding

Batch : 05

Branch : CSE

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 :

To generate a python program that implements the Merge Sort Algorithm

Code :

first.py > merge_sort

```python
def merge_sort(arr):
    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

if __name__ == "__main__":
    arr = [38, 27, 43, 3, 9, 82, 10]
    print("Original array:", arr)
    sorted_arr = merge_sort(arr)
    print("Sorted array:", sorted_arr)
```

Output :

```
PS C:\Users\sgoll\OneDrive\Documents\New folder> & C:/ProgramData/anaconda3/python.exe "c:/Users/sgoll/
OneDrive/Documents/New folder/first.py"
Original array: [38, 27, 43, 3, 9, 82, 10]
Sorted array: [3, 9, 10, 27, 38, 43, 82]
```

Observation :

The provided Python program implements the Merge Sort algorithm efficiently and clearly. It uses a recursive approach to divide the

input list into smaller sublists until each contains only one element, which are inherently sorted. The merge function then combines these sorted sublists into a single sorted list by comparing elements from each sublist and appending the smaller one to the result. This process continues until all elements are merged in order. The program demonstrates its functionality with a sample array, displaying both the original and sorted arrays. Overall, the code is well-structured, easy to understand, and effectively showcases the divide-and-conquer strategy of Merge Sort.

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

Prompt :

To create a binary search function that finds a target element in a sorted list.

Code :

first.py > ...

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

    Time complexity: O(n log n) for all cases.
    Space complexity: O(n) due to auxiliary arrays.
    """
    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):
        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
```

first.py > ...

```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 position on the first check.
    Average-case complexity: O(log n) - Each comparison halves the search space.
    Worst-case complexity: O(log n)   - Target is not present or at the end of the search space

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

    Returns:
        int: Index of the target if found, else -1.
    """
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
if __name__ == "__main__":
    arr = [38, 27, 43, 3, 9, 82, 10]
    print("Original array:", arr)
    sorted_arr = merge_sort(arr)
    print("Sorted array:", sorted_arr)
    targets = [43, 3, 82, 100, 9]
    for t in targets:
        idx = binary_search(sorted_arr, t)
        if idx != -1:
            print(f"Target {t} found at index {idx}.")
        else:
            print(f"Target {t} not found in the array.")
```

Output :

```
OneDrive/Documents/New folder/first.py"
Original array: [38, 27, 43, 3, 9, 82, 10]
Sorted array: [3, 9, 10, 27, 38, 43, 82]
Target 43 found at index 5.
Target 3 found at index 0.
Target 82 found at index 6.
Target 100 not found in the array.
Target 9 found at index 1.
```

Observation :

The code provided efficiently demonstrates two fundamental algorithms: Merge Sort and Binary Search. Merge Sort is used to sort an unsorted list, leveraging its divide-and-conquer approach for consistent O(n log n) time complexity. Once sorted, Binary Search is applied to quickly locate target values within the list, offering O(log n) search performance. The program includes clear docstrings explaining the complexities and provides practical test cases to validate both sorting and searching functionalities. Overall, the code is well-structured, readable, and serves as a solid example of combining sorting and searching algorithms for effective data handling.

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.

• Output:

o A table mapping operation → recommended algorithm → justification.

o Working Python functions for searching and sorting the inventory.

Prompt :

To suggest the most efficient search and sort algorithms for this use case

Code :

first.py > search_by_id

```python
inventory = [
    {"id": "P001", "name": "Widget", "price": 19.99, "quantity": 120},
    {"id": "P002", "name": "Gadget", "price": 29.99, "quantity": 80},
    {"id": "P003", "name": "Thingamajig", "price": 9.99, "quantity": 200},

]

id_index = {item["id"]: item for item in inventory}
name_index = {item["name"]: item for item in inventory}

def search_by_id(product_id):
    """Search for a product by its ID using a hash table (O(1) average)."""
    return id_index.get(product_id)

def search_by_name(product_name):
    """Search for a product by its name using a hash table (O(1) average)."""
    return name_index.get(product_name)

def sort_by_price(descending=False):
    """Sort inventory by price using Timsort (O(n log n))."""
    return sorted(inventory, key=lambda x: x["price"], reverse=descending)

def sort_by_quantity(descending=False):
    """Sort inventory by quantity using Timsort (O(n log n))."""
    return sorted(inventory, key=lambda x: x["quantity"], reverse=descending)

# --- Example usage ---
if __name__ == "__main__":
    print("Search by ID 'P002':", search_by_id("P002"))
    print("Search by Name 'Widget':", search_by_name("Widget"))
    print("Sort by Price (ascending):", sort_by_price())
    print("Sort by Quantity (descending):", sort_by_quantity(descending=True))
```

Output :

```
s/New folder/first.py"
Search by ID 'P002': {'id': 'P002', 'name': 'Gadget', 'price': 29.99, 'quantity': 80}
Search by Name 'Widget': {'id': 'P001', 'name': 'Widget', 'price': 19.99, 'quantity': 120}
Sort by Price (ascending): [{'id': 'P003', 'name': 'Thingamajig', 'price': 9.99, 'quantity': 200}, {'id': 'P001', 'name'
: 'Widget', 'price': 19.99, 'quantity': 120}, {'id': 'P002', 'name': 'Gadget', 'price': 29.99, 'quantity': 80}]
Sort by Quantity (descending): [{'id': 'P003', 'name': 'Thingamajig', 'price': 9.99, 'quantity': 200}, {'id': 'P001', 'n
ame': 'Widget', 'price': 19.99, 'quantity': 120}, {'id': 'P002', 'name': 'Gadget', 'price': 29.99, 'quantity': 80}]
PS C:\Users\sgoll\OneDrive\Documents\New folder>
```

Observation :

The code efficiently models a retail store's inventory system, enabling rapid product searches and flexible sorting. By using Python dictionaries as hash tables, it achieves constant-time average lookups for product ID and name, which is ideal for large datasets and frequent updates. Sorting functions utilize Python's built-in Timsort algorithm, providing stable and fast sorting by price or quantity. The example usage demonstrates how staff can quickly retrieve product details and analyze stock, making the solution practical and scalable for thousands of items. Overall, the design balances speed, simplicity, and maintainability for real-world inventory management.