AI ASSISTED CODING

NAME:-SANIYA

ROLL NO:-2403A510E7

BATCH :-05

LAB:12

- **TASK:1**

- PROMPT:

"Generate Python code for a function merge_sort(arr) that implements the Merge Sort algorithm. The function should return a sorted list in ascending order. Include a detailed docstring explaining the algorithm's time complexity (best, average, worst) and space complexity. Provide at least three test cases using assert statements to verify correctness."

- GENERATED CODE:

lab12.py > ...

```python
def merge_sort(arr):
    """
    Merge Sort Algorithm

    Time Complexity:
        - Best Case: O(n log n)
        - Average Case: O(n log n)
        - Worst Case: O(n log n)
    Space Complexity:
        - O(n) due to auxiliary arrays used during merging.

    Args:
        arr (list): List of comparable elements.

    Returns:
        list: Sorted list in ascending order.
    """
    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):
    """Helper function to merge two sorted lists."""
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
```

```python
def merge(left, right):

    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([3, 1, 4, 1, 5]) == [1, 1, 3, 4, 5]
assert merge_sort([10, -2, 0, 8]) == [-2, 0, 8, 10]
assert merge_sort([1]) == [1]
```

❖ OUTPUT:

```
PS C:\Users\mdyou\ai.ass> & C:/Users/mdyou/anaconda3/python.exe c:/Users/mdyou/ai.ass/lab12.py
All test cases passed successfully ✅
[1, 1, 2, 3, 4, 5, 6, 9]
[]
[-1, 0, 2, 8, 10]
```

❖ OBSERVATION:

The merge_sort function sorts a list correctly using the divide-and-conquer approach.
It has a time complexity of **O(n log n)** in best, average, and worst cases.
The space complexity is **O(n)** due to extra lists created during merging.
It is a **stable sorting algorithm**, preserving the order of equal elements.
All test cases passed, proving the code is correct and efficient.

❖ TASK:2

❖ PROMPT:

"Generate Python code for a function binary_search(arr, target) that performs binary search on a sorted list. The function should return the index of the target if found, otherwise -1. Add a docstring explaining best, average, and worst case complexities. Provide at least three assert test cases."

❖ GENERATED CODE:

task2lab12.py > ...

```python
def binary_search(arr, target):
    """
    Binary Search Algorithm

    Description:
        Binary Search works on a sorted list by repeatedly dividing
        the search interval in half until the target is found or the
        search space is empty.

    Time Complexity:
        - Best Case: O(1) → Target found at the middle index immediately
        - Average Case: O(log n) → Each step halves the search space
        - Worst Case: O(log n) → Target not present or found after all halvings

    Space Complexity:
        - O(1) for iterative implementation (no extra data structures)
        - O(log n) for recursive implementation (due to call stack)

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

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

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

        # Debugging / Explanation line
        # print(f"Searching between {low} and {high}, mid = {mid}")
```

```python
        if arr[mid] == target:
            return mid  # Target found
        elif arr[mid] < target:
            low = mid + 1  # Search right half
        else:
            high = mid - 1  # Search left half

    return -1  # Target not found


# ✅ Test Cases
assert binary_search([1, 2, 3, 4, 5], 3) == 2      # Target present in middle
assert binary_search([10, 20, 30, 40], 25) == -1  # Target not present
assert binary_search([5], 5) == 0                  # Single element, target found
assert binary_search([2, 4, 6, 8, 10], 10) == 4   # Target is last element
assert binary_search([2, 4, 6, 8, 10], 2) == 0    # Target is first element

print("All binary search test cases passed successfully ✅")
```

❖ OUTPUT:

```
PS C:\Users\mdyou\ai.ass> & C:/Users/mdyou/anaconda3/python.exe c:/Users/mdyou/ai.ass/task2lab12.py
All binary search test cases passed successfully ✅
PS C:\Users\mdyou\ai.ass> 
```

❖ OBSERVATION:

The binary_search function correctly implements the iterative binary search algorithm.
It has a **time complexity of O(log n)** on average and in the worst case, with a best case of **O(1)**.
The space complexity is **O(1)** since no extra memory is used.
The function works efficiently on sorted lists and passed all given test cases.
Thus, the implementation is **correct, optimized, and reliable** for searching in large datasets.

❖ TASK:3

❖ PROMPT:

"Suggest the most efficient search and sorting algorithms for an inventory management system with thousands of products. Implement searching by product ID and product name, and sorting by price or quantity. Provide a justification table mapping operations → recommended algorithm → justification.

❖ GENERATED CODE:

task3.12.py > ...

```python
# Binary Search function
def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1


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"{self.name} (ID: {self.product_id}, Price: {self.price}, Qty: {self.quantity})


# Search by ID using Binary Search
def search_by_id(products, target_id):
    products.sort(key=lambda x: x.product_id)
    ids = [p.product_id for p in products]
    index = binary_search(ids, target_id)
    return products[index] if index != -1 else None


# Search by Name using Dictionary (Hash Map)
def search_by_name(products, target_name):
    product_dict = {p.name: p for p in products}
```

```python
    return product_dict.get(target_name, None)


# Sort Products by Price
def sort_by_price(products):
    return merge_sort(products, key=lambda x: x.price)


# Sort Products by Quantity
def sort_by_quantity(products):
    return merge_sort(products, key=lambda x: x.quantity)


# Modified Merge Sort to handle key
def merge_sort(arr, key=lambda x: x):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid], key)
    right_half = merge_sort(arr[mid:], key)
    return merge(left_half, right_half, key)


def merge(left, right, key):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if key(left[i]) <= key(right[j]):
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
```

```python
    result.extend(left[i:])
    result.extend(right[j:])
    return result


# ✅ Test Cases
products = [
    Product(102, "Shampoo", 120, 50),
    Product(101, "Soap", 40, 200),
    Product(103, "Toothpaste", 80, 100)
]

assert search_by_id(products, 101).name == "Soap"
assert search_by_name(products, "Shampoo").product_id == 102
assert [p.name for p in sort_by_price(products)] == ["Soap", "Toothpaste", "Shampoo"]
assert [p.name for p in sort_by_quantity(products)] == ["Shampoo", "Toothpaste", "Soap"]

print("All inventory management test cases passed successfully ✅")
```

❖ OUTPUT:

```
PS C:\Users\mdyou\ai.ass> & C:/Users/mdyou/anaconda3/python.exe c:/Users/mdyou/ai.ass/task3.12.py
All inventory management test cases passed successfully ✅
PS C:\Users\mdyou\ai.ass> 
```

❖ OBSERVATION:

1. The search_by_id function efficiently finds products using **binary search** after sorting by product_id (O(log n) time).
2. The search_by_name function uses a **hash map**, allowing O(1) average-time lookup for product names.
3. Sorting functions sort_by_price and sort_by_quantity use **merge sort with a key**, giving O(n log n) time and stable sorting.
4. All test cases passed, verifying correct search and sort functionality for different attributes.
5. The code is **modular, scalable, and optimized** for handling large inventory datasets.