LAB ASSIGNMENT : 12.1

NAME:A.SREEMANTH REDDY

ROLL NO: 2403A510G1

BATCH:06

Task #1 – Merge Sort Implementation

Prompt

Write Python code for Lab 12 – Task 1:
Implement a function merge_sort(arr) that sorts a list in ascending order using the Merge Sort algorithm.

Code

12.py

12.py > merge_sort

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

    Merge Sort is a divide-and-conquer algorithm that:
      1. Splits the list into halves.
      2. Recursively sorts each half.
      3. Merges the sorted halves.

    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 the additional space used for merging.

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

    Returns:
        list: A new list containing the sorted elements in ascending order.
    """
    if len(arr) <= 1:
        return arr

    # Divide the list into two halves
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])

    # Merge the two sorted halves
    return merge(left_half, right_half)


def merge(left, right):
    """
    Merge two sorted lists into one sorted list.
    """
    merged = []
    i = j = 0

    # Compare elements from both lists and add the smaller one
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1

    # Add remaining elements
    merged.extend(left[i:])
    merged.extend(right[j:])
    return merged


# -------------------- Test Cases --------------------

if __name__ == "__main__":
```

```python
    test_cases = [
        [38, 27, 43, 3, 9, 82, 10],
        [5, 4, 3, 2, 1],
        [1, 2, 3, 4, 5],
        [],
        [7],
        [10, -2, 33, 5, 0, -8]
    ]

    for idx, arr in enumerate(test_cases, start=1):
        print(f"Test Case {idx}: Original: {arr}")
        print(f"Sorted: {merge_sort(arr)}\n")

```

Output

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\allas\OneDrive\Documents\web> & C:\Python313\python.exe c:/Users/allas/OneDrive/Documents/web/12.py
Test Case 1: Original: [38, 27, 43, 3, 9, 82, 10]
Sorted: [3, 9, 10, 27, 38, 43, 82]

Test Case 2: Original: [5, 4, 3, 2, 1]
Sorted: [1, 2, 3, 4, 5]

Test Case 3: Original: [1, 2, 3, 4, 5]
Sorted: [1, 2, 3, 4, 5]

Test Case 4: Original: []
Sorted: []

Test Case 5: Original: [7]
Sorted: [7]

Test Case 6: Original: [10, -2, 33, 5, 0, -8]
Sorted: [-8, -2, 0, 5, 10, 33]

PS C:\Users\allas\OneDrive\Documents\web> 
```

Observation

Merge Sort was implemented recursively with helper merge(). The algorithm correctly sorted all test cases and the complexities were documented.

Task #2 – Binary Search

Prompt

Write Python code for Lab 12 – Task 2:
Implement a function binary_search(arr, target) that returns the index of the target in a sorted list or -1 if not found.

Code

12.py   12.2py.py

12.2py.py > binary_search

```python
def binary_search(arr, target):
    """
    Search for a target element in a sorted list using the Binary Search algorithm.

    Binary Search works by repeatedly dividing the search interval in half:
      1. Compare the target with the middle element.
      2. If they are equal, return the index.
      3. If the target is smaller, search the left half; otherwise, search the right half.

    Time Complexity:
        - Best Case:  O(1)   → target is found at the first middle check
        - Average Case: O(log n)
        - Worst Case: O(log n)

    Space Complexity:
        - O(1) (iterative version), as it uses constant extra space.

    Parameters:
        arr (list): A sorted list of comparable elements.
        target (object): The value to search for.

    Returns:
        int: The index of the target in arr if found, otherwise -1.
    """
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = left + (right - left) // 2  # Prevents potential overflow

        # --- AI tip: check middle element first for efficiency ---
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            # Search in the right half
            left = mid + 1
        else:
            # Search in the left half
            right = mid - 1

    # Target not found
```

```python
    return -1


# -------------------- Test Cases --------------------
if __name__ == "__main__":
    sorted_list = [1, 3, 5, 7, 9, 11, 13, 15, 17]

    targets = [1, 7, 17, 4, 13, 20]

    for t in targets:
        result = binary_search(sorted_list, t)
        if result != -1:
            print(f"Target {t} found at index {result}")
        else:
            print(f"Target {t} not found in the list")

    # Edge cases
    print("\nEdge Cases:")
    print("Empty list:", binary_search([], 10))
    print("Single element, present:", binary_search([42], 42))
    print("Single element, absent:", binary_search([42], 7))

```

Output

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\allas\OneDrive\Documents\web> & C:\Python313\python.exe c:/Users/allas/OneDrive/Documents/web/12.2py.py
Target 1 found at index 0
Target 7 found at index 3
Target 17 found at index 8
Target 4 not found in the list
Target 13 found at index 6
Target 20 not found in the list

Edge Cases:
Empty list: -1
Single element, present: 0
Single element, absent: -1
PS C:\Users\allas\OneDrive\Documents\web> 
```

Observation

Binary search was implemented iteratively. It correctly returned indices or -1 for missing elements, with constant space usage.

Task #3 – Inventory Management System

Prompt

Write Python code for Lab 12 – Task 3:
A retail store’s inventory has thousands of products with id, name, price, and quantity.

Code

12.py | 12.2py.py | 12.3.py

12.3.py > ...

```python
from bisect import bisect_left

# ------------------- Data Model -------------------

inventory = [
    {"id": 101, "name": "Laptop", "price": 75000, "quantity": 12},
    {"id": 205, "name": "Smartphone", "price": 25000, "quantity": 35},
    {"id": 150, "name": "Headphones", "price": 3000, "quantity": 120},
    {"id": 302, "name": "Keyboard", "price": 1500, "quantity": 75},
    {"id": 410, "name": "Monitor", "price": 12000, "quantity": 22},
    {"id": 501, "name": "Mouse", "price": 800, "quantity": 150},
]

# Create a dictionary for O(1) search by ID
inventory_by_id = {item["id"]: item for item in inventory}

# Precompute sorted list of names for binary search
sorted_names = sorted(item["name"] for item in inventory)


# ------------------- Search Functions -------------------

def search_by_id(product_id):
    """
    Search for a product by its ID using hash map lookup.

    Time Complexity:
        - Average Case: O(1)
        - Worst Case: O(n) (very rare, due to hash collisions)
    """
    return inventory_by_id.get(product_id, None)


def search_by_name(name):
    """
    Search for a product by name using binary search.

    Time Complexity:
        - Best: O(1) (middle element)
        - Average/Worst: O(log n)
    """
    idx = bisect_left(sorted_names, name)
    if idx < len(sorted_names) and sorted_names[idx] == name:
        # Retrieve full product details
        for item in inventory:
            if item["name"] == name:
                return item
    return None


# ------------------- Sorting Functions -------------------
```

```python

def sort_by_price(descending=False):
    """
    Sort inventory by product price using Timsort (Python's built-in sort).
    Complexity: O(n log n)
    """
    return sorted(inventory, key=lambda x: x["price"], reverse=descending)


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


# ------------------- Demo / Test -------------------

if __name__ == "__main__":
    # Search by ID
    print("Search by ID (205):", search_by_id(205))
    print("Search by ID (999):", search_by_id(999))

    # Search by Name
    print("\nSearch by Name ('Monitor'):", search_by_name("Monitor"))
    print("Search by Name ('Tablet'):", search_by_name("Tablet"))

    # Sort by Price
    print("\nSorted by Price (ascending):")
    for item in sort_by_price():
        print(item)

    print("\nSorted by Quantity (descending):")
    for item in sort_by_quantity(descending=True):
        print(item)

```

Output

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS C:\Users\allas\OneDrive\Documents\web> & C:\Python313\python.exe c:/Users/allas/OneDrive/Documents/web/12.3.py
Search by ID (205): {'id': 205, 'name': 'Smartphone', 'price': 25000, 'quantity': 35}
Search by ID (999): None

Search by Name ('Monitor'): {'id': 410, 'name': 'Monitor', 'price': 12000, 'quantity': 22}
Search by Name ('Tablet'): None

Sorted by Price (ascending):
{'id': 501, 'name': 'Mouse', 'price': 800, 'quantity': 150}
{'id': 302, 'name': 'Keyboard', 'price': 1500, 'quantity': 75}
{'id': 150, 'name': 'Headphones', 'price': 3000, 'quantity': 120}
{'id': 410, 'name': 'Monitor', 'price': 12000, 'quantity': 22}
{'id': 205, 'name': 'Smartphone', 'price': 25000, 'quantity': 35}
{'id': 101, 'name': 'Laptop', 'price': 75000, 'quantity': 12}

Sorted by Quantity (descending):
{'id': 501, 'name': 'Mouse', 'price': 800, 'quantity': 150}
{'id': 150, 'name': 'Headphones', 'price': 3000, 'quantity': 120}
{'id': 302, 'name': 'Keyboard', 'price': 1500, 'quantity': 75}
{'id': 205, 'name': 'Smartphone', 'price': 25000, 'quantity': 35}
{'id': 410, 'name': 'Monitor', 'price': 12000, 'quantity': 22}
{'id': 101, 'name': 'Laptop', 'price': 75000, 'quantity': 12}
PS C:\Users\allas\OneDrive\Documents\web>
```

Observation

Hash map search for IDs, binary search for names, and Timsort for sorting were used. Functions provided fast lookups and correct sorted outputs.