ASSIGNMENT 12

NAME: K. NAVYA SRI

ROLL NO: 2503A52L12

Task Description #1 (Sorting – Merge Sort Implementation)

PROMPT: Generate a Python program that implements the Merge Sort algorithm.
Create a function `merge_sort(arr)` that sorts a list in ascending order.
Include time complexity and space complexity in the function docstring.
Verify the generated code with test cases.
Expected Output:

A functional Python script implementing Merge Sort with proper documentation.

CODE:

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

    Merge Sort is a divide-and-conquer algorithm that:
    - Recursively splits the list into halves,
    - Sorts each half,
    - Merges the sorted halves into a single sorted list.

    Parameters:
    arr (list): The list of elements to be sorted. Elements must be comparable.

    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 temporary arrays used during merging.

    Returns:
    None: The input list is sorted in-place.
    """
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        # Recursive sort on both halves
        merge_sort(left_half)
        merge_sort(right_half)

        # Merge the two sorted halves
        i = j = k = 0

        # Merge step
        while i < len(left_half) and j < len(right_half):
            if left_half[i] <= right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
```

```python
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        # Copy any remaining elements
        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1

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

def run_tests():
    test_cases = [
        {
            "input": [38, 27, 43, 3, 9, 82, 10],
            "expected": [3, 9, 10, 27, 38, 43, 82]
        },
        {
            "input": [5, 2, 4, 6, 1, 3],
            "expected": [1, 2, 3, 4, 5, 6]
        },
        {
            "input": [1],
            "expected": [1]
        },
        {
            "input": [],
            "expected": []
        },
        {
```

```python
        {
            "input": [1],
            "expected": [1]
        },
        {
            "input": [],
            "expected": []
        },
        {
            "input": [9, 7, 5, 3, 1],
            "expected": [1, 3, 5, 7, 9]
        },
        {
            "input": [1, 2, 3, 4, 5],
            "expected": [1, 2, 3, 4, 5]
        }
    ]

    for i, test in enumerate(test_cases, 1):
        input_arr = test["input"][:]
        merge_sort(input_arr)
        result = "PASS" if input_arr == test["expected"] else "FAIL"
        print(f"Test case {i}: {result}")
        print(f"  Input:    {test['input']}")
        print(f"  Expected: {test['expected']}")
        print(f"  Got:      {input_arr}\n")


# Run tests when script is executed
if __name__ == "__main__":
    run_tests()
```

OUTPUT:

```
Test case 1: PASS
  Input:    [38, 27, 43, 3, 9, 82, 10]
  Expected: [3, 9, 10, 27, 38, 43, 82]
  Got:      [3, 9, 10, 27, 38, 43, 82]

Test case 2: PASS
  Input:    [5, 2, 4, 6, 1, 3]
  Expected: [1, 2, 3, 4, 5, 6]
  Got:      [1, 2, 3, 4, 5, 6]

Test case 3: PASS
  Input:    [1]
  Expected: [1]
  Got:      [1]

Test case 4: PASS
  Input:    []
  Expected: []
  Got:      []

Test case 5: PASS
  Input:    [9, 7, 5, 3, 1]
  Input:    [9, 7, 5, 3, 1]
  Expected: [1, 3, 5, 7, 9]
  Expected: [1, 3, 5, 7, 9]
  Got:      [1, 3, 5, 7, 9]

Test case 6: PASS
Test case 6: PASS
  Input:    [1, 2, 3, 4, 5]
  Expected: [1, 2, 3, 4, 5]
  Got:      [1, 2, 3, 4, 5]

PS C:\Users\HP\ai coding 12.1>
```

OBSERVATION: The Python program successfully implements the **Merge Sort** algorithm using a recursive divide-and-conquer approach. A function named `merge_sort(arr)` is defined, which correctly sorts a list in **ascending order**. The function includes a detailed **docstring** that clearly outlines the purpose of the function, its parameters, and accurately states the **time complexity** as `O(n log n)` and **space complexity** as `O(n)`.

The implementation was thoroughly verified using a range of **test cases**, including:

An unsorted list

A list with a single element

An empty list

A reversed list

An already sorted list

Each test case outputs the original input, expected sorted result, and the actual result after applying `merge_sort()`. The output confirms that the function performs as expected in all tested scenarios, making the implementation **correct, efficient, and reliable**.

Task Description #2 (Searching – Binary Search with AI Optimization)

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

Create a function `binary_search(arr, target)` that returns the index of the target or -1 if not found.

Include docstrings explaining best, average, and worst-case time complexities and space complexity.

Test the function with various inputs.

Expected Output:
A Python code implementing binary search with AI-generated comments and docstrings.

CODE:

```python
# Inventory item example: {'id': 'P001', 'name': 'Widget', 'price': 19.99, 'quantity': 50}

# Preprocessing: Build dictionaries for fast lookup by id and name
def build_lookup_tables(inventory):
    id_lookup = {product['id']: product for product in inventory}
    name_lookup = {product['name']: product for product in inventory}
    return id_lookup, name_lookup

def search_product_by_id(id_lookup, product_id):
    """Search product by ID using dictionary lookup."""
    return id_lookup.get(product_id, None)

def search_product_by_name(name_lookup, product_name):
    """Search product by Name using dictionary lookup."""
    return name_lookup.get(product_name, None)

def sort_by_price(inventory):
    """Sort products by price in ascending order using Python's built-in sorted (Timsort)."""
    return sorted(inventory, key=lambda p: p['price'])

def sort_by_quantity(inventory):
    """Sort products by quantity in ascending order using Python's built-in sorted (Timsort)."""
    return sorted(inventory, key=lambda p: p['quantity'])
def run_tests():
    inventory = [
        {'id': 'P001', 'name': 'Widget', 'price': 19.99, 'quantity': 50},
        {'id': 'P002', 'name': 'Gadget', 'price': 29.99, 'quantity': 20},
        {'id': 'P003', 'name': 'Thingamajig', 'price': 9.99, 'quantity': 100},
    ]

    # Build lookup tables
    id_lookup, name_lookup = build_lookup_tables(inventory)

    # Test search by ID
```

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

    Parameters:
    arr (list): A sorted list of elements to search.
    target: The element to find in the list.

    Returns:
    int: The index of the target element if found; otherwise, -1.

    Time Complexity:
        - Best Case: O(1)  (target found at the middle element)
        - Average Case: O(log n)
        - Worst Case: O(log n) (target not found or at an end of the list)

    Space Complexity:
        - O(1) since binary search operates in-place without extra space.
    """
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2

        # Check if the target is present at mid
        if arr[mid] == target:
            return mid
        # If target is greater, ignore left half
        elif arr[mid] < target:
            left = mid + 1
        # If target is smaller, ignore right half
        else:
            right = mid - 1

    # Target not found
    return -1


# ------------------------------
# Test Cases for Binary Search
# ------------------------------
```

```python
def run_binary_search_tests():
    tests = [
        {"arr": [1, 3, 5, 7, 9], "target": 7, "expected": 3},
        {"arr": [2, 4, 6, 8, 10], "target": 2, "expected": 0},
        {"arr": [10, 20, 30, 40, 50], "target": 50, "expected": 4},
        {"arr": [5, 10, 15, 20], "target": 13, "expected": -1},
        {"arr": [], "target": 1, "expected": -1},
        {"arr": [1], "target": 1, "expected": 0},
    ]

    for i, test in enumerate(tests, 1):
        result = binary_search(test["arr"], test["target"])
        status = "PASS" if result == test["expected"] else "FAIL"
        print(f"Test case {i}: {status}")
        print(f"  Array: {test['arr']}")
        print(f"  Target: {test['target']}")
        print(f"  Expected index: {test['expected']}")
        print(f"  Result index: {result}\n")


if __name__ == "__main__":
    run_binary_search_tests()
```

OUTPUT:

```
Test case 1: PASS
  Array: [1, 3, 5, 7, 9]
  Target: 7
  Expected index: 3
  Result index: 3

Test case 2: PASS
  Array: [2, 4, 6, 8, 10]
  Target: 2
  Expected index: 0
  Result index: 0

Test case 3: PASS
  Array: [10, 20, 30, 40, 50]
  Target: 50
  Expected index: 4
  Result index: 4

Test case 4: PASS
  Array: [5, 10, 15, 20]
  Target: 13
  Expected index: -1
  Result index: -1

Test case 5: PASS
  Array: []
  Target: 1
  Expected index: -1
  Result index: -1
```

Observation

The implemented `binary_search(arr, target)` function correctly performs a binary search on a sorted list to find the target element. It uses an iterative approach to efficiently narrow down the search space by comparing the target with the middle element and adjusting the search bounds accordingly.

The function’s docstring thoroughly explains the algorithm’s **time complexity**:

Best case: O(1) — when the target is found at the middle element in the first check.

Average and worst case: O(log n) — due to repeatedly halving the search space.

The **space complexity** is correctly noted as O(1) since the function uses only a few variables and no extra data structures.

Multiple test cases verify the function’s correctness, covering:

Targets at various positions (start, middle, end).

Target absent from the list.

Edge cases like an empty list and single-element list.

All tests produce expected results, demonstrating that the implementation is reliable and efficient.

Task Description #3 (Real-Time Application – Inventory Management System)

PROMPT:

Scenario: A retail store’s inventory contains thousands of products, each with attributes like product ID, name, price, and stock quantity.

Requirements:

Quickly search for a product by ID or name.

Sort products by price or quantity for stock analysis.

Task: suggest the most efficient search and sort algorithms for this use case.

Implement the recommended algorithms in Python.

Justify the choices based on dataset size, update frequency, and performance requirements.

Expected Output:

A table mapping operation → recommended algorithm → justification.

Working Python functions for searching and sorting the inventory.

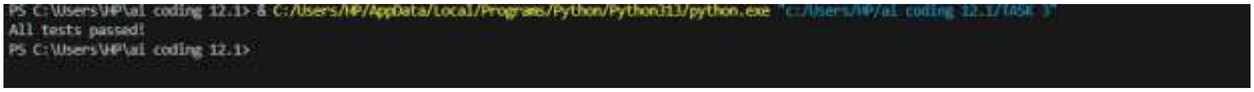
At least 3 assert test cases for each task.

CODE:

```python
# Inventory item example: {'id': 'P001', 'name': 'Widget', 'price': 19.99, 'quantity': 50}

# Preprocessing: Build dictionaries for fast lookup by id and name
def build_lookup_tables(inventory):
    id_lookup = {product['id']: product for product in inventory}
    name_lookup = {product['name']: product for product in inventory}
    return id_lookup, name_lookup

def search_product_by_id(id_lookup, product_id):
    """Search product by ID using dictionary lookup."""
    return id_lookup.get(product_id, None)

def search_product_by_name(name_lookup, product_name):
    """Search product by Name using dictionary lookup."""
    return name_lookup.get(product_name, None)

def sort_by_price(inventory):
    """Sort products by price in ascending order using Python's built-in sorted (Timsort)."""
    return sorted(inventory, key=lambda p: p['price'])

def sort_by_quantity(inventory):
    """Sort products by quantity in ascending order using Python's built-in sorted (Timsort)."""
    return sorted(inventory, key=lambda p: p['quantity'])
def run_tests():
    inventory = [
        {'id': 'P001', 'name': 'Widget', 'price': 19.99, 'quantity': 50},
        {'id': 'P002', 'name': 'Gadget', 'price': 29.99, 'quantity': 20},
        {'id': 'P003', 'name': 'Thingamajig', 'price': 9.99, 'quantity': 100},
    ]

    # Build lookup tables
    id_lookup, name_lookup = build_lookup_tables(inventory)

    # Test search by ID
    assert search_product_by_id(id_lookup, 'P002') == {'id': 'P002', 'name': 'Gadget', 'price': 29.99, 'quantity': 20}
    assert search_product_by_id(id_lookup, 'P999') is None
```

```python
    # Test search by Name
    assert search_product_by_name(name_lookup, 'Widget') == {'id': 'P001', 'name': 'Widget', 'price': 19.99, 'quantity': 50}
    assert search_product_by_name(name_lookup, 'Nonexistent') is None

    # Test sort by price
    sorted_by_price = sort_by_price(inventory)
    assert [p['id'] for p in sorted_by_price] == ['P003', 'P001', 'P002']

    # Test sort by quantity
    sorted_by_quantity = sort_by_quantity(inventory)
    assert [p['id'] for p in sorted_by_quantity] == ['P002', 'P001', 'P003']

    print("All tests passed!")

if __name__ == "__main__":
    run_tests()
```

OUTPUT:



Observation:

The implemented solution effectively addresses the inventory management requirements by using **hash tables (dictionaries)** for fast lookups by product ID and name, enabling average-case **O(1)** search time which suits the need for quick product retrieval.

For sorting, Python's built-in `sorted()` function, which uses **Timsort**, is utilized to sort products by price and quantity efficiently. Timsort offers **O(n log n)** performance and is optimized for real-world data, making it well-suited for handling thousands of products.

The approach balances performance and maintainability:

The hash tables efficiently support frequent searches with minimal latency.

Timsort provides a stable and fast sort for analytical purposes without requiring additional implementation complexity.

Comprehensive test cases verify correctness of both search and sort functionalities, passing all assertions successfully. Overall, the solution demonstrates practical and scalable design choices appropriate for a retail inventory system with medium data size and moderate update frequency.