AI ASSISTED CODING

LAB ASSIGNMENT – 12.1

NAME : Neha Shazneen

ROLL NO : 2403A510A1

BATCH : 05

DEPARTMENT : CSE

Task 1 : (Sorting – Merge Sort Implementation)

Use AI to generate a Python program that implements the
Merge Sort algorithm.

Prompt :

Generate a Python function called merge_sort(arr) that sorts a list in ascending order using the Merge Sort algorithm.

- Include a detailed docstring for the function that explains its time complexity and space complexity.
- After the function, provide test cases to verify that the implementation works correctly.

Code Generated :

12.1t1.py > ...

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])

    merged = []
    i = j = 0

    # Merge the sorted halves
    while i < len(left_half) and j < len(right_half):
        if left_half[i] < right_half[j]:
            merged.append(left_half[i])
            i += 1
        else:
            merged.append(right_half[j])
            j += 1

    merged.extend(left_half[i:])
    merged.extend(right_half[j:])

    return merged

if __name__ == "__main__":
    test_cases = [
        [],
        [1],
        [5, 2, 9, 1, 5, 6],
        [3, 3, 3],
        [10, -1, 2, 5, 0],
        [100, 50, 25, 12, 6, 3, 1]
    ]

    for i, test in enumerate(test_cases):
        sorted_list = merge_sort(test)
        print(f"Test case {i+1}: {test} -> {sorted_list}")
```

Output :

```
PS C:\Users\Summya Mehveen\OneDrive\Desktop\AI associated coding>  c:; cd 'c:\Users\Summya Mehveen\OneDrive\Desktop\AI associated coding'; & 'c:\Users\Summya Mehveen\AppData\Local\Programs\Python\Python313\python.exe' 'c:\Users\Summya Mehveen\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '61866' '--' 'c:\Users\Summya Mehveen\OneDrive\Desktop\AI associated coding\12.1t1.py'
Test case 1: [] -> []
Test case 2: [1] -> [1]
Test case 3: [5, 2, 9, 1, 5, 6] -> [1, 2, 5, 5, 6, 9]
Test case 4: [3, 3, 3] -> [3, 3, 3]
Test case 5: [10, -1, 2, 5, 0] -> [-1, 0, 2, 5, 10]
Test case 6: [100, 50, 25, 12, 6, 3, 1] -> [1, 3, 6, 12, 25, 50, 100]
```

Observation :

After running the merge_sort(arr) function with the provided test cases, the following observations can be made:

1. **Correctness:**
   - The function correctly sorts lists in ascending order for all tested cases, including empty lists, single-element lists, lists with duplicates, negative numbers, and already sorted or reverse-sorted lists.
2. **Stability:**
   - Merge Sort is a stable sorting algorithm, so equal elements retain their original order.
3. **Efficiency:**
   - The function performs efficiently for small to moderately sized lists, as expected from its O(n log n) time complexity.
   - The space complexity is O(n) due to the creation of new lists during the merge process.
4. **Edge Cases:**
   - The implementation handles edge cases (empty list, single element, all elements equal) without errors.
5. **Practical Use:**
   - Merge Sort is suitable for large datasets where stability is required and extra memory usage is acceptable.

Summary:
The AI-generated merge_sort function is reliable, efficient, and handles a variety of input cases as expected for the Merge Sort algorithm.

Task 2 : (Searching – Binary Search with AI Optimization)

Use AI to create a binary search function that finds a target element in a sorted list.

Prompt :

Use AI to create a binary search function that finds a target element in a sorted list.

Instructions:

- 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.
- Test the function with various inputs.

Code Generated :

12.1t2.py > ...

```python
def binary_search(arr, target):
    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

# Test cases
if __name__ == "__main__":
    arr1 = [1, 3, 5, 7, 9, 11]
    print(binary_search(arr1, 7))
    print(binary_search(arr1, 1))
    print(binary_search(arr1, 11))
    print(binary_search(arr1, 4))

    arr2 = []
    print(binary_search(arr2, 10))

    arr3 = [2]
    print(binary_search(arr3, 2))
    print(binary_search(arr3, 3))
```

Output :

```
PS C:\Users\Summya Mehveen\OneDrive\Desktop\AI associated coding>  c:; cd 'c:\Users\Summya Mehve
en\OneDrive\Desktop\AI associated coding'; & 'c:\Users\Summya Mehveen\AppData\Local\Programs\Pyt
hon\Python313\python.exe' 'c:\Users\Summya Mehveen\.vscode\extensions\ms-python.debugpy-2025.10.
0-win32-x64\bundled\libs\debugpy\launcher' '60156' '--' 'c:\Users\Summya Mehveen\OneDrive\Deskto
p\AI associated coding\12.1t2.py'
3
0
5
-1
-1
0
-1
```

Observation :

The binary search algorithm successfully locates the target element in a sorted list by repeatedly dividing the search interval in half.

- **Working:**
  - If the target is equal to the middle element, its index is returned.
  - If the target is smaller, the search continues in the left half; if larger, in the right half.
  - This reduces the problem size exponentially with each step.
- **Performance:**
  - **Best case:** *O(1)* → Target is found at the middle on the first comparison.
  - **Average case:** *O(log n)* → Target is found after repeatedly halving the list.
  - **Worst case:** *O(log n)* → Target is not found after all divisions.
- **Correctness:**
  The test cases showed that the function correctly returned the index when the element was present and returned -1 when the element was absent.
- **Observation Outcome:**
  Binary search is efficient for large, sorted datasets compared to linear search. The AI-generated implementation handled all inputs accurately, confirming its reliability and efficiency.

Task 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.

Prompt :

You are an AI assistant. A retail store's inventory system contains thousands of products, each with attributes like product ID, name, price, and stock quantity. The 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:

- Suggest the most efficient **search** and **sort** algorithms suitable for this use case.
- Implement the recommended algorithms in **Python**.
- Justify your choice based on:
  - Dataset size (thousands of records)
  - Update frequency (moderate: stock updates daily, product catalog occasionally changes)
  - Performance requirements (fast searches and sorting operations for staff use).

Output:

- Python code with functions for searching and sorting.
- Explanations of time complexity and why these algorithms were chosen.
- Example test cases showing correct functionality.

Code Generated :

12.1t3.py > ...

```python
from bisect import bisect_left

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.product_id} - {self.name} | Price: {self.price} | Qty: {self.quantity}"


def binary_search(products, target_id):
    products.sort(key=lambda x: x.product_id)
    ids = [p.product_id for p in products]

    index = bisect_left(ids, target_id)
    if index < len(products) and ids[index] == target_id:
        return products[index]
    return None


def search_by_name(product_dict, target_name):
    return product_dict.get(target_name, None)


def sort_by_price(products):
    return sorted(products, key=lambda x: x.price)


def sort_by_quantity(products):
    return sorted(products, key=lambda x: x.quantity)


if __name__ == "__main__":
    inventory = [
        Product(103, "Shampoo", 120, 50),
        Product(101, "Soap", 40, 200),
        Product(105, "Toothpaste", 60, 150),
        Product(102, "Lotion", 250, 30),
        Product(104, "Oil", 180, 80),
    ]

    product_dict = {p.name: p for p in inventory}

    print("🔎 Search by ID (101):", binary_search(inventory, 101))
    print("🔎 Search by Name ('Oil'):", search_by_name(product_dict, "Oil"))
    print("🔎 Search by Name ('Perfume'):", search_by_name(product_dict, "Perfume"))

    print("\n📊 Products sorted by Price:")
    for p in sort_by_price(inventory):
        print(p)

    print("\n📊 Products sorted by Quantity:")
    for p in sort_by_quantity(inventory):
        print(p)
```

Output :

```
PS C:\Users\Summya Mehveen\OneDrive\Desktop\AI associated coding>
ding'; & 'c:\Users\Summya Mehveen\AppData\Local\Programs\Python\Py
-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher'
ted coding\12.1t3.py'
🔎 Search by ID (101): 101 - Soap | Price: 40 | Qty: 200
🔎 Search by Name ('Oil'): 104 - Oil | Price: 180 | Qty: 80
🔎 Search by Name ('Perfume'): None

📊 Products sorted by Price:
101 - Soap | Price: 40 | Qty: 200
105 - Toothpaste | Price: 60 | Qty: 150
103 - Shampoo | Price: 120 | Qty: 50
104 - Oil | Price: 180 | Qty: 80
102 - Lotion | Price: 250 | Qty: 30

📊 Products sorted by Quantity:
102 - Lotion | Price: 250 | Qty: 30
103 - Shampoo | Price: 120 | Qty: 50
104 - Oil | Price: 180 | Qty: 80
105 - Toothpaste | Price: 60 | Qty: 150
101 - Soap | Price: 40 | Qty: 200
```

Observation :

The AI-recommended solution for the retail store inventory system effectively balances speed and efficiency for both **searching** and **sorting** tasks.

- **Searching:**
  - **Product ID** search using **Binary Search** was efficient, with a logarithmic time complexity (*O(log n)*).
  - **Product Name** search using a **Hash Map (dictionary lookup)** provided instant retrieval in constant time (*O(1) average*).
  - Test cases confirmed correct behavior: valid products returned their details, while invalid searches returned None.
- **Sorting:**
  - Sorting by **price** and **quantity** using Python's built-in **Timsort** worked efficiently with *O(n log n)* complexity.
  - The results showed correctly ordered products, verifying the stability and accuracy of the algorithm.

- **Overall Performance:**
    - With thousands of records, the chosen algorithms ensure **fast lookups** and **efficient stock analysis**.
    - Moderate update frequency (daily stock changes) did not affect performance, as both searching and sorting remained reliable.
- **Outcome:**
  The AI-optimized approach successfully met the requirements of quick product searches and stock analysis, making it suitable for real-world retail inventory management.