Assignment 12.1

Name:N.Prudhvi

HT.NO: 2403A510G7

Task1: (Sorting – Merge Sort Implementation)

Prompt:

Write a Python program that demonstrates the Merge Sort algorithm. Define a function `merge_sort(arr)` that takes a list of numbers and returns the sorted list in ascending order. Make sure the function includes a clear docstring describing how Merge Sort works, along with its **time complexity (O(n log n))** and **space complexity (O(n))**. Finally, provide at least three test cases to verify that the sorting works correctly for different types of input (e.g., random list, already sorted list, reverse sorted list)

Code:

```
Users > HP > t1.py > merge
    def merge_sort(arr):
        right_half = merge_sort(arr[mid:])

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

    def merge(left, right):
        """
        Merges two sorted lists into a single sorted list.

        Parameters:
        left (list): The first sorted list.
        right (list): The second sorted list.

        Returns:
        list: A merged and sorted list.
        """
        merged = []
        i = j = 0

        # Merge the two lists until one is exhausted
        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

        # Append any remaining elements from both lists
        merged.extend(left[i:])
        merged.extend(right[j:])

        return merged
```

Output:

```
> & C:/Users/HP/AppData/Local/Microsoft/WindowsApps/python3.11.exe c:/Users/HP/t1.py
Sorted test_case_1: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
Sorted test_case_2: [5, 6, 7, 11, 12, 13]
Sorted test_case_3: [1, 2]
Sorted test_case_4: [10]
PS C:\Users\HP>
```

Observation:

1)Merge Sort might look a bit complex at first, but once you see it break the list into smaller chunks, it feels very natural — like organizing scattered papers before filing them."

2)What I found interesting is how Merge Sort always guarantees efficiency, no matter if the list is random, sorted, or completely reversed."

3)Unlike some other algorithms, Merge Sort's process is very systematic and almost calming — it's like solving a puzzle piece by piece."

Task 2: Searching – Binary Search with AI Optimization

Prompt:

Design a Python program that demonstrates the **Binary Search** algorithm. Implement a function `binary_search(arr, target)` that takes a sorted list and a target value, and returns the index of the target if it exists, otherwise returns `-1`. The function should include a detailed docstring explaining how binary search works, along with its **best-case (O(1))**, **average-case (O(log n))**, and **worst-case (O(log n))** time complexities. Finally, provide several test cases with different types of inputs (e.g., element present at beginning, middle, end, and not present) to verify correctness.

Code:

```
: > Users > HP > t2.py > binary_search
  def binary_search(arr, target):
      while low <= high:
          mid = (low + high) // 2

          # Check if the target is at the middle
          if arr[mid] == target:
              return mid
          # If target is greater, ignore left half
          elif arr[mid] < target:
              low = mid + 1
          # If target is smaller, ignore right half
          else:
              high = mid - 1
      | Ctrl+L to chat, Ctrl+K to generate
      # Target is not present
      return -1

  # Test cases
  if __name__ == "__main__":
      test_case_1 = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
      test_case_2 = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
      test_case_3 = [10, 20, 30, 40, 50]
      test_case_4 = [1, 2, 3, 4, 5]

      # Searching for target 9
      print(f"Index of 9 in test_case_1: {binary_search(test_case_1, 9)}")  # Should return 4
      # Searching for target 12
      print(f"Index of 12 in test_case_2: {binary_search(test_case_2, 12)}")  # Should return 5
      # Searching for target 30
      print(f"Index of 30 in test_case_3: {binary_search(test_case_3, 30)}")  # Should return 2
      # Searching for target 5
      print(f"Index of 5 in test_case_4: {binary_search(test_case_4, 5)}")  # Should return 4
      # Searching for target 100 (not in list)
      print(f"Index of 100 in test_case_1: {binary_search(test_case_1, 100)}")  # Should return -1
```

Output:

```
PS C:\Users\HP> & C:/Users/HP/AppData/Local/Microsoft/WindowsApps/python3.11.exe c:/Users/HP/t2.py
Index of 9 in test_case_1: 4
Index of 12 in test_case_2: 5
Index of 30 in test_case_3: 2
Index of 5 in test_case_4: 4
Index of 100 in test_case_1: -1
PS C:\Users\HP>
```

Observation:

1)Binary Search feels efficient because it doesn't waste time checking every element — it just keeps cutting the problem in half."

2)The algorithm works best when the data is already sorted, which is both its strength and limitation."

3)I noticed how quickly it narrows down possibilities; even in a large list, the target can be found in just a few steps."

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

Prompt:

Imagine a retail store that manages a large inventory with thousands of products. Each product has attributes such as product ID, name, price, and stock quantity. Store employees need an efficient system to:

1. Search for products quickly using either product ID or name.
2. Organize products by price or stock quantity for better inventory analysis.

Your task is to:

1)Recommend suitable **search** and **sorting algorithms** that balance efficiency and practicality for this scenario.

2)Implement these algorithms in Python.

3)Provide a clear justification for your choices, considering factors like dataset size, frequency of inventory updates, and performance needs.

Code:

C: > Users > HP > t3.py > Inventory

```python
from bisect import bisect_left

# --- Product class ---
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} | ${self.price} | Qty: {self.quantity}"


# --- Inventory System ---
class Inventory:
    def __init__(self):
        self.products = []                  # List of all products
        self.product_by_id = {}             # Dict for O(1) ID lookup
        self.products_sorted_by_name = []   # Sorted names for binary search

    def add_product(self, product: Product):
        self.products.append(product)
        self.product_by_id[product.product_id] = product
        self.products_sorted_by_name.append(product.name)
        self.products_sorted_by_name.sort()  # Maintain sorted names for binary search

    # --- Search ---
    def search_by_id(self, product_id):
        return self.product_by_id.get(product_id, "Product not found")

    def search_by_name(self, name):
        idx = bisect_left(self.products_sorted_by_name, name)
        if idx < len(self.products_sorted_by_name) and self.products_sorted_by_name[idx] == name:
            return [p for p in self.products if p.name == name]
        return "Product not found"

    # --- Sorting ---
    def sort_by_price(self, reverse=False):
        return sorted(self.products, key=lambda p: p.price, reverse=reverse)

    def sort_by_quantity(self, reverse=False):
        return sorted(self.products, key=lambda p: p.quantity, reverse=reverse)


# --- Example usage ---
if __name__ == "__main__":
    inventory = Inventory()
```

```python
    # --- Sorting ---
    def sort_by_price(self, reverse=False):
        return sorted(self.products, key=lambda p: p.price, reverse=reverse)

    def sort_by_quantity(self, reverse=False):
        return sorted(self.products, key=lambda p: p.quantity, reverse=reverse)

    Ctrl+L to chat, Ctrl+K to generate
# --- Example usage ---
if __name__ == "__main__":
    inventory = Inventory()

    # Add products
    inventory.add_product(Product(101, "Apples", 2.5, 150))
    inventory.add_product(Product(102, "Bananas", 1.2, 200))
    inventory.add_product(Product(103, "Oranges", 3.0, 100))
    inventory.add_product(Product(104, "Grapes", 4.5, 75))

    # Search examples
    print("Search by ID (102):", inventory.search_by_id(102))
    print("Search by Name (Bananas):", inventory.search_by_name("Bananas"))
    print("Search by Name (Mangoes):", inventory.search_by_name("Mangoes"))

    # Sorting examples
    print("\nSort by Price (Low → High):", inventory.sort_by_price())
    print("Sort by Price (High → Low):", inventory.sort_by_price(reverse=True))
    print("Sort by Quantity (Low → High):", inventory.sort_by_quantity())
    print("Sort by Quantity (High → Low):", inventory.sort_by_quantity(reverse=True))
```

Output:

```
PS C:\Users\HP> & C:/Users/HP/AppData/Local/Microsoft/WindowsApps/python3.11.exe c:/Users/HP/t3.py
Search by ID (102): 102 | Bananas | $1.2 | Qty: 200
Search by Name (Bananas): [102 | Bananas | $1.2 | Qty: 200]
Search by Name (Mangoes): Product not found

Sort by Price (Low → High): [102 | Bananas | $1.2 | Qty: 200, 101 | Apples | $2.5 | Qty: 150, 103 | Oranges | $3.0 | Qty: 100, 104 | Grapes | $4.5 | Qty: 75]
Sort by Price (High → Low): [104 | Grapes | $4.5 | Qty: 75, 103 | Oranges | $3.0 | Qty: 100, 101 | Apples | $2.5 | Qty: 150, 102 | Bananas | $1.2 | Qty: 200]
Sort by Quantity (Low → High): [104 | Grapes | $4.5 | Qty: 75, 103 | Oranges | $3.0 | Qty: 100, 101 | Apples | $2.5 | Qty: 150, 102 | Bananas | $1.2 | Qty: 200]
Sort by Quantity (High → Low): [102 | Bananas | $1.2 | Qty: 200, 101 | Apples | $2.5 | Qty: 150, 103 | Oranges | $3.0 | Qty: 100, 104 | Grapes | $4.5 | Qty: 75]
PS C:\Users\HP>
```

Observation:

1)Efficient searching is crucial because staff cannot afford to scan through thousands of items manually.”

2)Binary search works well for product ID lookups when the data is sorted, while hash-based search offers faster lookups by name.”

3)Merge Sort or Quick Sort is suitable for sorting large datasets like inventory because they balance speed and reliability.”