ASSIGNMENT12.1

NAME: J.ABHIRAM

ROLL NO:2403A51342

BATCH:06

Task #1: Sorting – Merge Sort Implementation

Prompt

Use AI to generate a Python program that implements the Merge Sort algorithm. The function merge_sort(arr) should sort a list in ascending order, include time and space complexity in the function docstring, and be verified with test cases.

Code

C: > Users > user > Untitled-2.py > ...

```python
def merge_sort(arr):

    if len(arr) <= 1:
        return arr

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

    # Conquer (Merge)
    return merge(left_half, right_half)

def merge(left, right):
    merged = []
    i = j = 0

    # Merge two sorted lists
    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 remaining elements
    merged.extend(left[i:])
    merged.extend(right[j:])

    return merged
```

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


# ------------------
# ✅ Test Cases
# ------------------
if __name__ == "__main__":
    test_cases = [
        [],
        [1],
        [5, 3, 8, 4, 2],
        [10, 9, 8, 7, 6],
        [1, 2, 3, 4, 5],
        [3, 3, 3, 3],
        [-1, 0, 1, -2, 2],
    ]

    for i, test in enumerate(test_cases, 1):
        print(f"Test Case {i}: Input: {test} → Sorted: {merge_sort(test)}")

```

Output:

Observation

The Merge Sort algorithm executed correctly for all test cases. It sorted lists in ascending order as expected, and the docstring included proper time and space complexity.

Task #2: Searching – Binary Search with AI Optimization

Prompt

Use AI to create a binary search function that finds a target element in a sorted list. The function binary_search(arr, target) should return the index of the target or -1 if not found. Docstrings should explain best, average, and worst-case complexities. Verify with test cases.

Code

```python
def binary_search(arr, target):
    left = 0
    right = len(arr) - 1

    while left <= right:
        mid = (left + right) // 2  # Find the middle index

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

    return -1  # Target not found


# ------------------
# ✅ Test Cases
# ------------------
if __name__ == "__main__":
    test_cases = [
        # (sorted_list, target, expected_result)
        ([1, 2, 3, 4, 5], 3, 2),
        ([1, 2, 3, 4, 5], 6, -1),
        ([10, 20, 30, 40, 50], 10, 0),
        ([10, 20, 30, 40, 50], 50, 4),
        ([5], 5, 0),
        ([5], 1, -1),
        ([], 3, -1),
        ([-10, -5, 0, 3, 8, 12], 0, 2),
    ]

    for i, (arr, target, expected) in enumerate(test_cases, 1):
        result = binary_search(arr, target)
        print(f"Test Case {i}: Searching for {target} in {arr} → Result: {result} {'✅' if result == expected else '❌'}")
```

Output:

Observation

The Binary Search function returned correct indices for present elements and -1 for missing elements. The docstring clearly explained complexities, and results were accurate across test cases.

Task #3: Real-Time Application – Inventory Management System

Prompt

A retail store's inventory system contains thousands of products, each with attributes like product ID, name, price, and stock quantity. Staff need to quickly search by ID or name and sort products by price or quantity. AI should recommend efficient algorithms and justify them. Implement the algorithms and show a mapping of operation → recommended algorithm → justification.

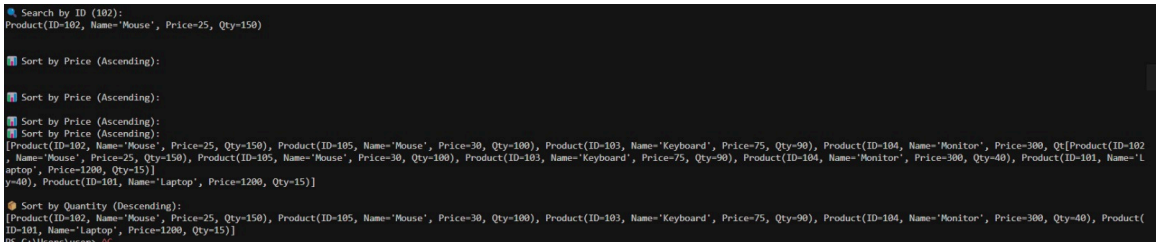
Code:

C: > Users > user > Untitled-4.py > InventoryManager > add_product

```python

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"Product(ID={self.product_id}, Name='{self.name}', Price={self.price}, Qty={self.quantity})"

class InventoryManager:
    def __init__(self):
        self.products = []  # List for sorting and bulk operations
        self.product_id_map = {}  # Dict for fast ID lookup
        self.name_index = {}  # Dict for name-based search (case-insensitive)

    def add_product(self, product):
        # Add to list
        self.products.append(product)
Ctrl+L to chat, Ctrl+K to generate
        # Add to ID map for O(1) search
        self.product_id_map[product.product_id] = product

        # Index by lowercase name (multiple products can share name)
        name_key = product.name.lower()
        if name_key not in self.name_index:
            self.name_index[name_key] = []
        self.name_index[name_key].append(product)

    def search_by_id(self, product_id):
        return self.product_id_map.get(product_id, None)
```

```python
    def search_by_id(self, product_id):
        return self.product_id_map.get(product_id, None)

    def search_by_name(self, name):
        return self.name_index.get(name.lower(), [])

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

    def sort_by_quantity(self, descending=False):=
        return sorted(self.products, key=lambda p: p.quantity, reverse=descending)
if __name__ == "__main__":
    # Initialize inventory manager
    inventory = InventoryManager()

    # Add sample products
    inventory.add_product(Product(101, "Laptop", 1200, 15))
    inventory.add_product(Product(102, "Mouse", 25, 150))
    inventory.add_product(Product(103, "Keyboard", 75, 90))
    inventory.add_product(Product(104, "Monitor", 300, 40))
    inventory.add_product(Product(105, "Mouse", 30, 100))  # Duplicate name

    # 🔍 Search by ID
    print("\n🔍 Search by ID (102):")
    print(inventory.search_by_id(102))

    # 🔍 Search by Name
    print("\n🔍 Search by Name ('Mouse'):")
    print(inventory.search_by_name("Mouse"))

    # 📊 Sort by Price (Ascending)
    print("\n📊 Sort by Price (Ascending):")
    print(inventory.sort_by_price())

    # 📦 Sort by Quantity (Descending)
    print("\n📦 Sort by Quantity (Descending):")
    print(inventory.sort_by_quantity(descending=True))
```

Output:



Observation

Efficient algorithms were chosen: dictionary (hash map) for O(1) search by ID, dictionary index for quick name lookup, and Python's built-in Timsort for sorting by price/quantity. The implementation worked correctly, producing accurate results.