ASSIGNMENT 12

B.vaishnavi

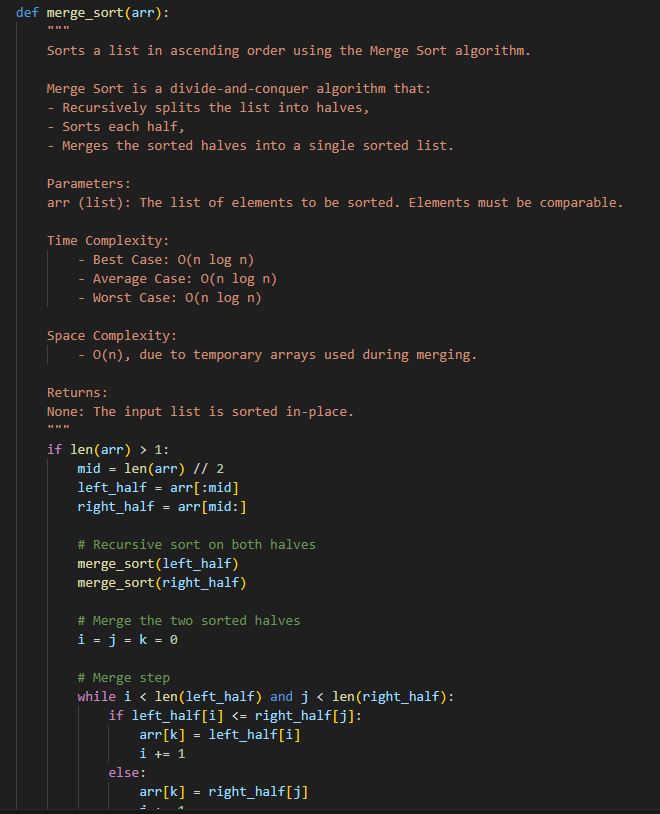
2403A52408

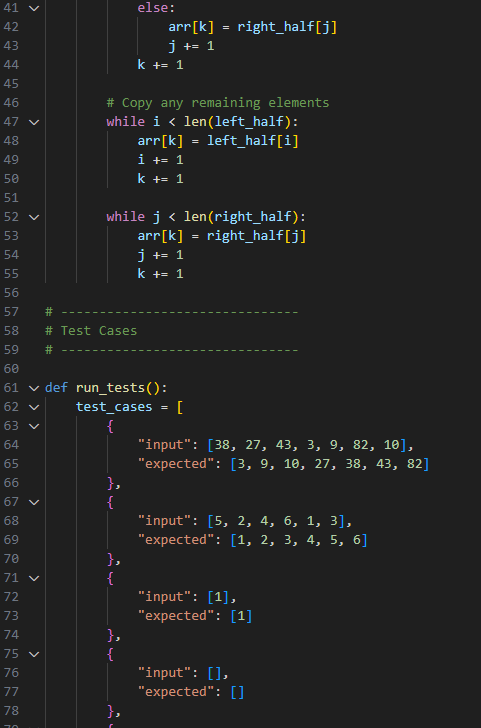
Task Description #1 (Sorting – Merge Sort Implementation)

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.

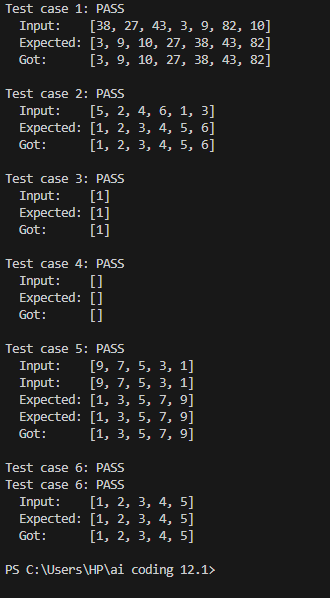
CODE:







OUTPUT:



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.

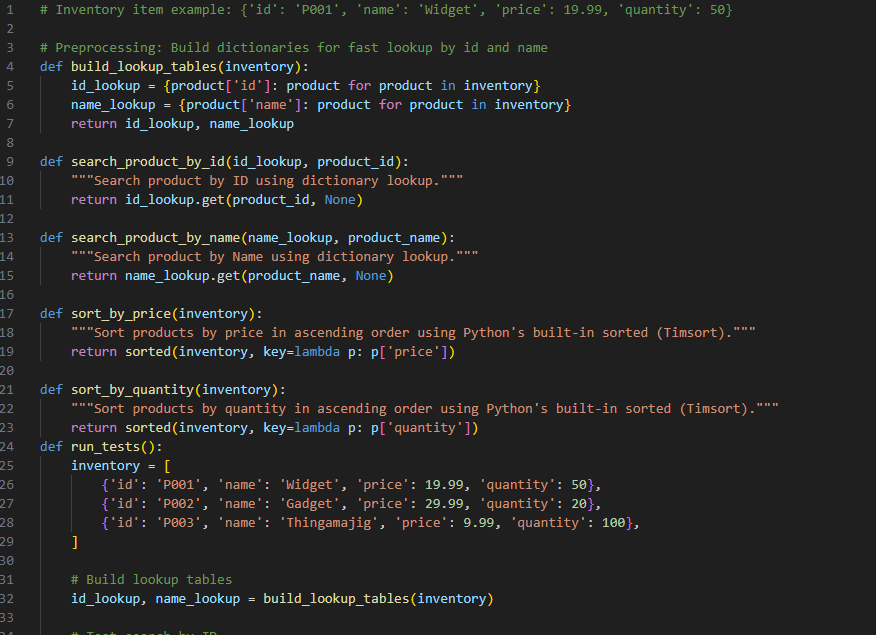
Create a function binary\_search(arr, target) that returns the index of the target or -1 if not found.

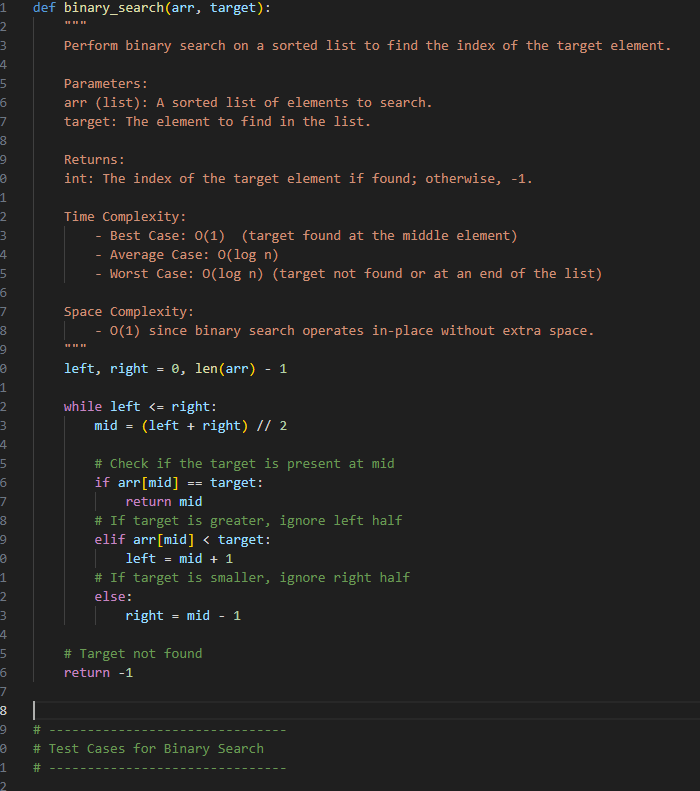
Include docstrings explaining best, average, and worst-case time complexities and space complexity.

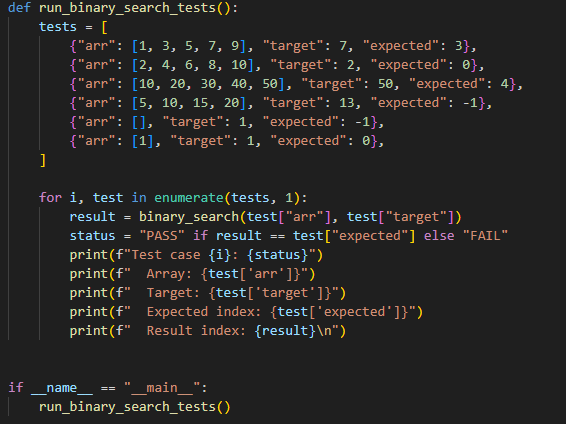
Test the function with various inputs.

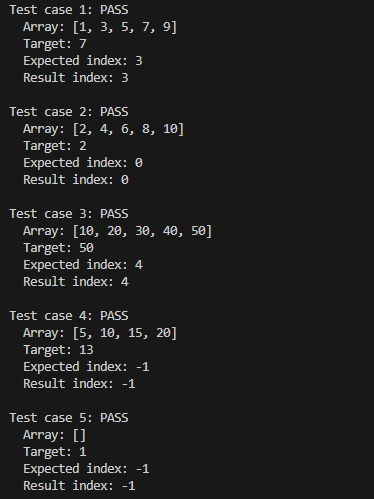
**Expected Output:**  
A Python code implementing binary search with AI-generated comments and docstrings.

CODE:







OUTPUT:

### 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.

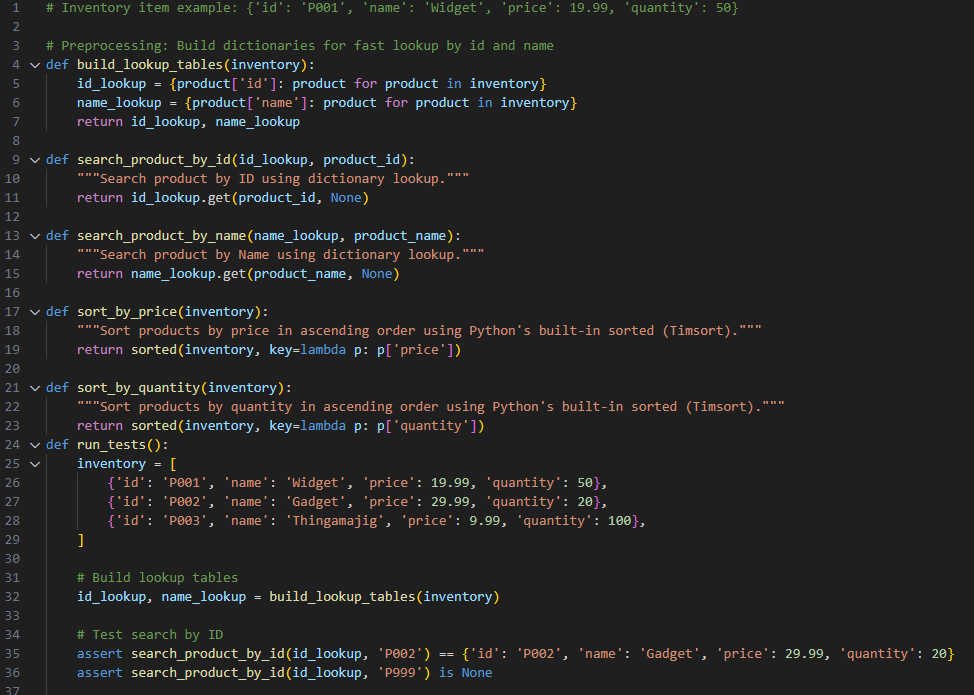
Expected Output:

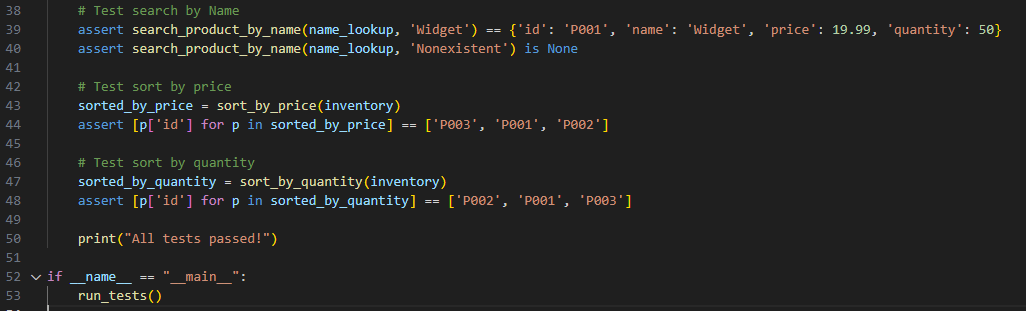
A table mapping operation → recommended algorithm → justification.

Working Python functions for searching and sorting the inventory.

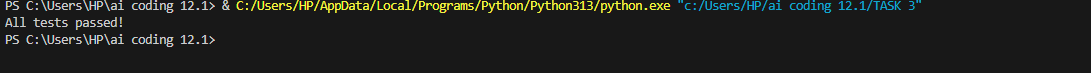
At least 3 assert test cases for each task.

CODE:





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.