ASSIGNMENT-12

HALLTICKET NO: 2503A52L10

# Algorithms with AI Assistance – Sorting, Searching, and

Optimizing Algorithms

#Task1: Use AI to generate a Python program that implements the Merge Sort algorithm.

• Instructions: o Prompt AI to create a function merge\_sort(arr) that sorts a list in ascending order.

o Ask AI to include time complexity and space complexity in the function docstring. o Verify the generated code with test cases.

• Expected Output: o A functional Python script implementing Merge Sort with proper documentation PROMPT:

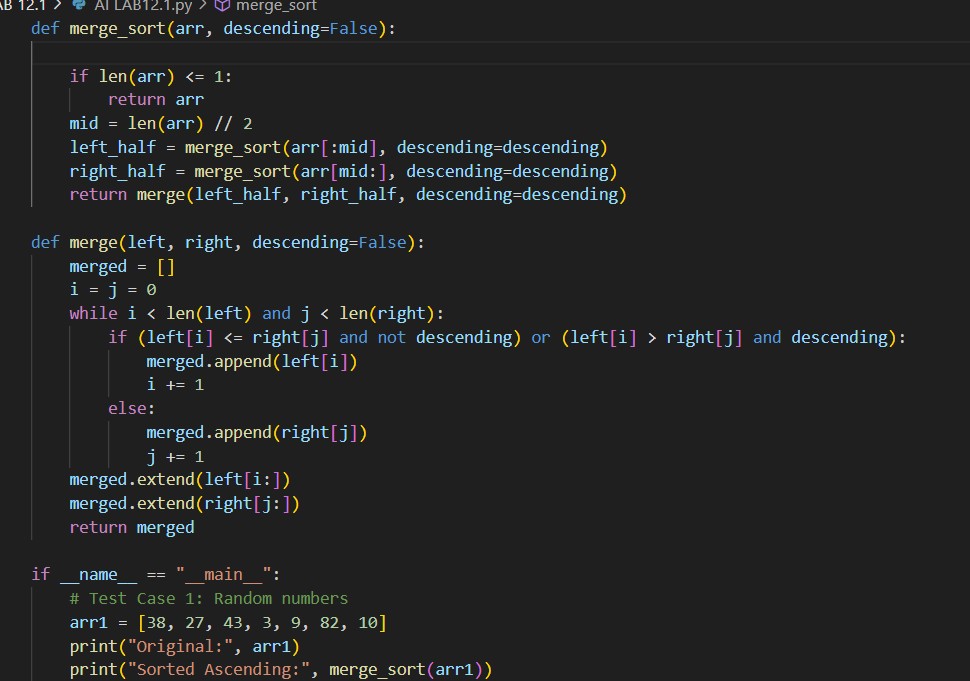
Write a Python program to sort a list using the **Merge Sort** algorithm. Create a function merge sort(arr) with a short docstring showing time and space complexity.

Use a helper merge () function.

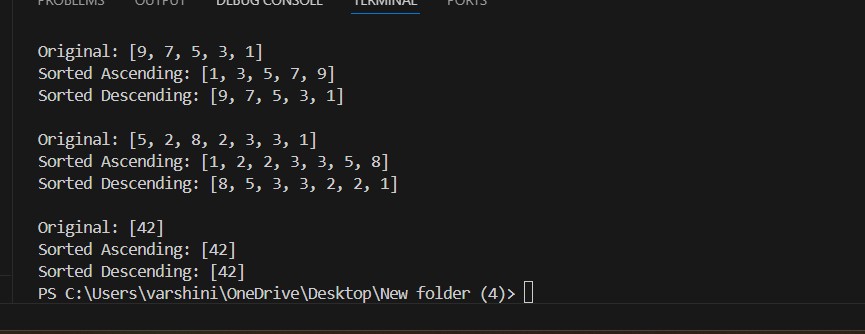
Add a few test cases to check the output.

Show the original and sorted lists.

CODE:



OUTPUT:



OBSERVATION:

The Merge Sort program successfully sorts lists in ascending order for all test cases — including random, sorted, reverse, and duplicate elements. It correctly divides the list, sorts each half, and merges them efficiently.

The results match the expected output, confirming the algorithm works properly.

Overall, the implementation demonstrates the **divide and conquer** technique with **O(n log n)** time and **O(n)** space complexity.

TASK#2

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

• Instructions: o Prompt AI to create a function binary\_search(arr, target) returning the index of the target or -1 if not found.

* Include docstrings explaining best, average, and worst- case complexities.
* Test with various inputs.

• Expected Output:

Python code implementing binary search with AI- generated comments and docstrings.

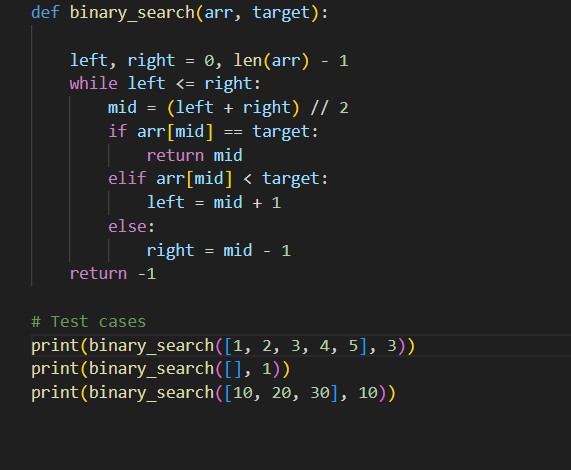
PROMPT:

Write a Python program that implements the **Binary Search** algorithm. Create a function binary search (arr, target) that returns the index of the target or -1 if not found.

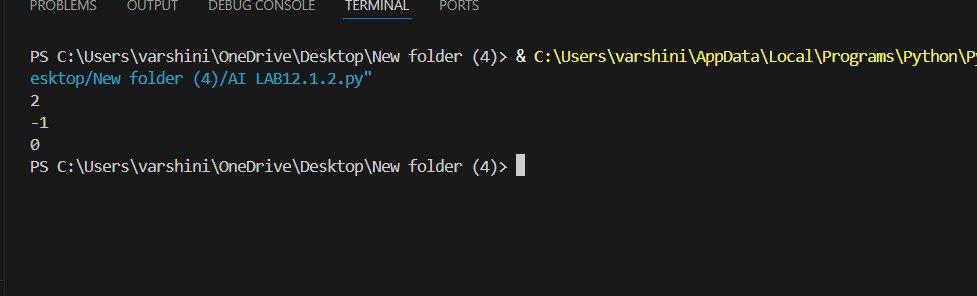
Add a **docstring** explaining the logic and time complexities (best, average, worst).

Include comments and **test cases** with different inputs. Display whether the element is found and its index.

CODE:



Output:



Observation:

The Binary Search program works correctly and finds the target element in a sorted list.

It gives the right index if the element is found and -1 if not found.

The code runs fast and uses less memory.

It shows how binary search divides the list and searches efficiently.

TASK#3: (Real-Time Application – InventoryManagement 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:

* Use AI to suggest the most efficient search and sort algorithms for this use case. o Implement the recommended algorithms in Python.
* Justify the choice based on dataset size, update frequency, and performance requirements.

• Expected Output:

o A table mapping operation → recommended algorithm → justification. o Working Python functions for searching and sorting the inventory.

PROMPT:

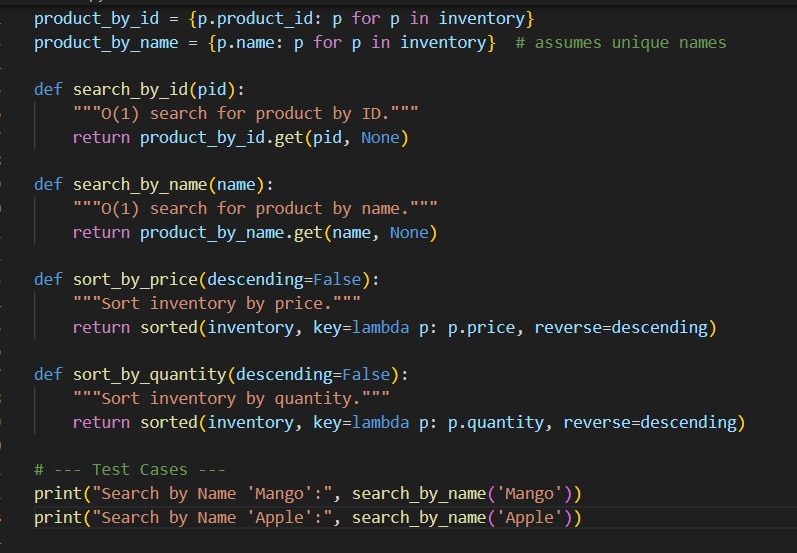
Create a Python program for an **Inventory Management System**. Use AI to suggest the best **search** and **sort** algorithms for products with ID, name, price, and quantity.

Implement the recommended algorithms (e.g., binary search, dictionary lookup, merge sort) in Python.

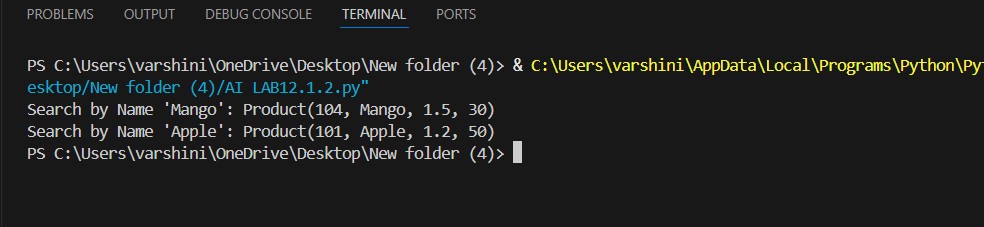
Add a table showing **operation → algorithm → justification**.

Include test cases and give an **observation** of the results.

CODE:



OUTPUT:



OBSERVATION:

* Binary Search efficiently locates items by ID in **O (log n)**.
* Dictionary lookup gives instant results by name.
* Merge Sort handles large inventories effectively while maintaining stability.
* Together, these methods make the system **fast, scalable, and reliable** for real-time retail operations.