<u>AI ASSISTED CODING</u>

ROLL NO: 2503A51L16

NAME: K. JASHUVA

LAB NO: 12

BRANCH: CSE

TASK 1

Task Description 1 (Sorting – Merge Sort Implementation):

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

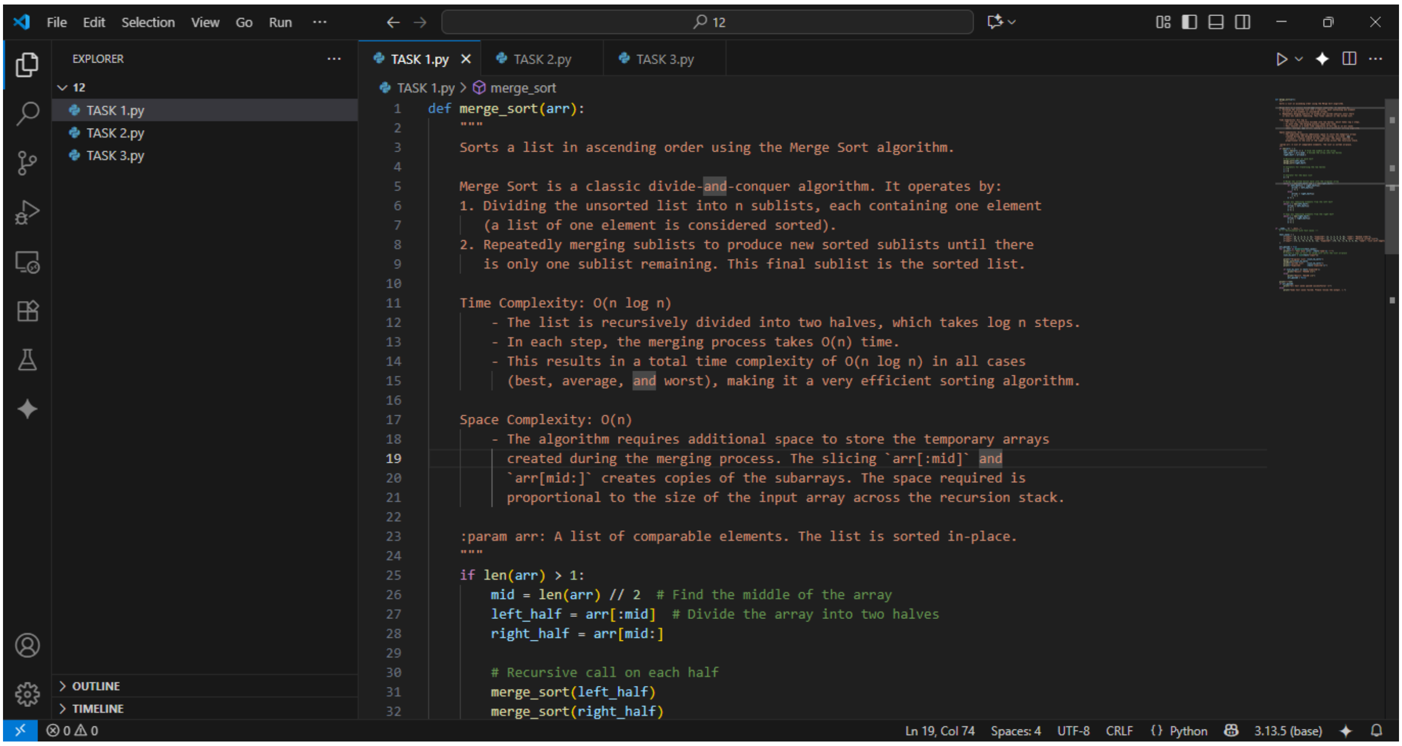
Instructions:

- Prompt AI to create a function merge_sort(arr) that sorts a list in ascending order.
- Ask AI to include time complexity and space complexity in the function docstring
- Verify the generated code with test cases.

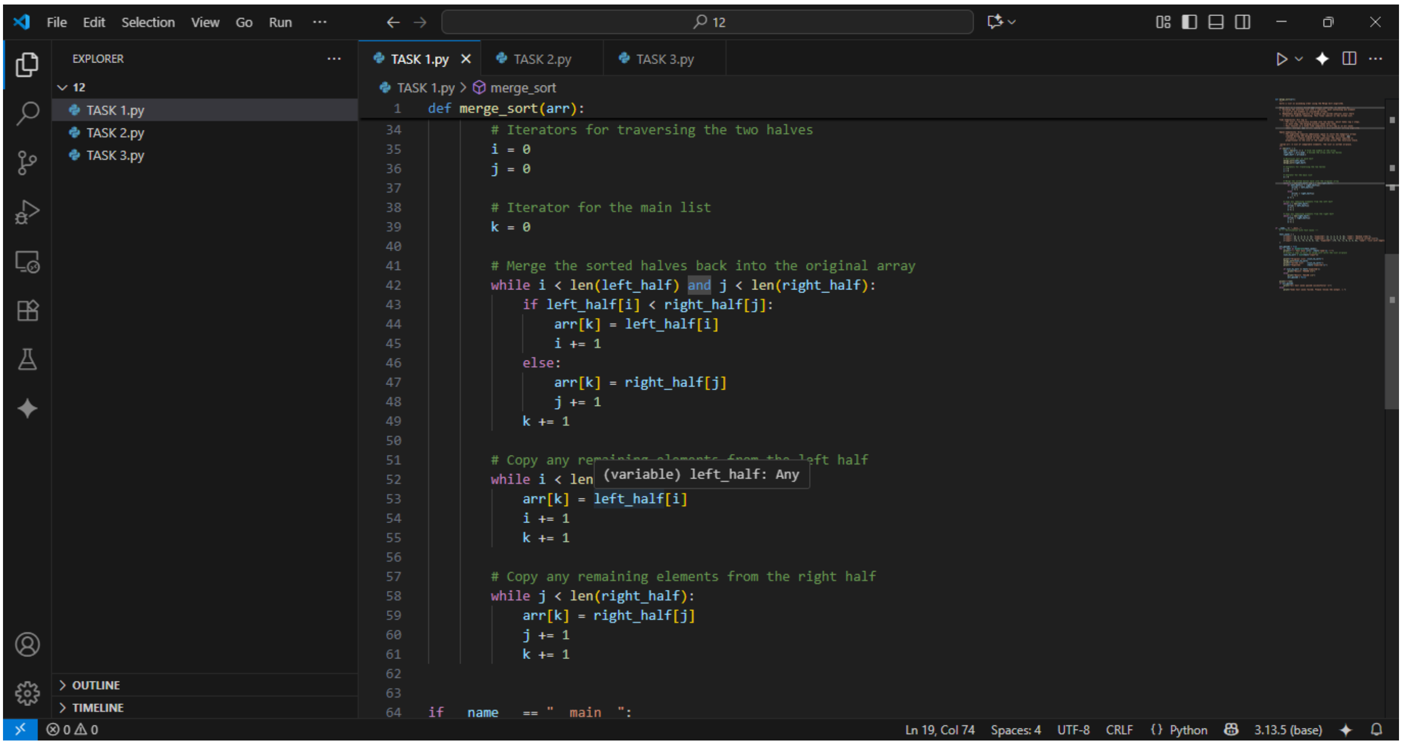
PROMPT:

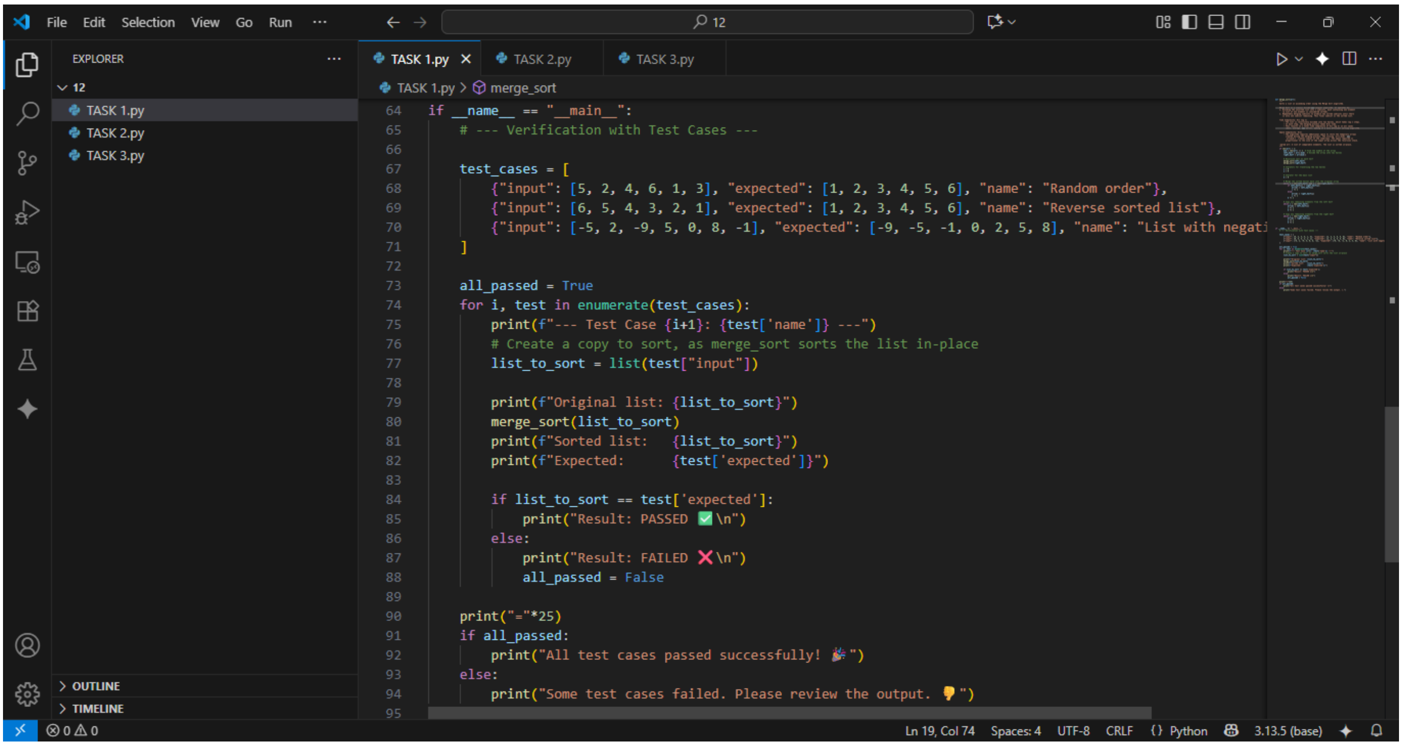
Generate a Python program that implements Merge Sort with a function merge_sort(arr) that sorts a list in ascending order. Include a docstring explaining time and space complexity, and add test cases to verify the code.

CODE:

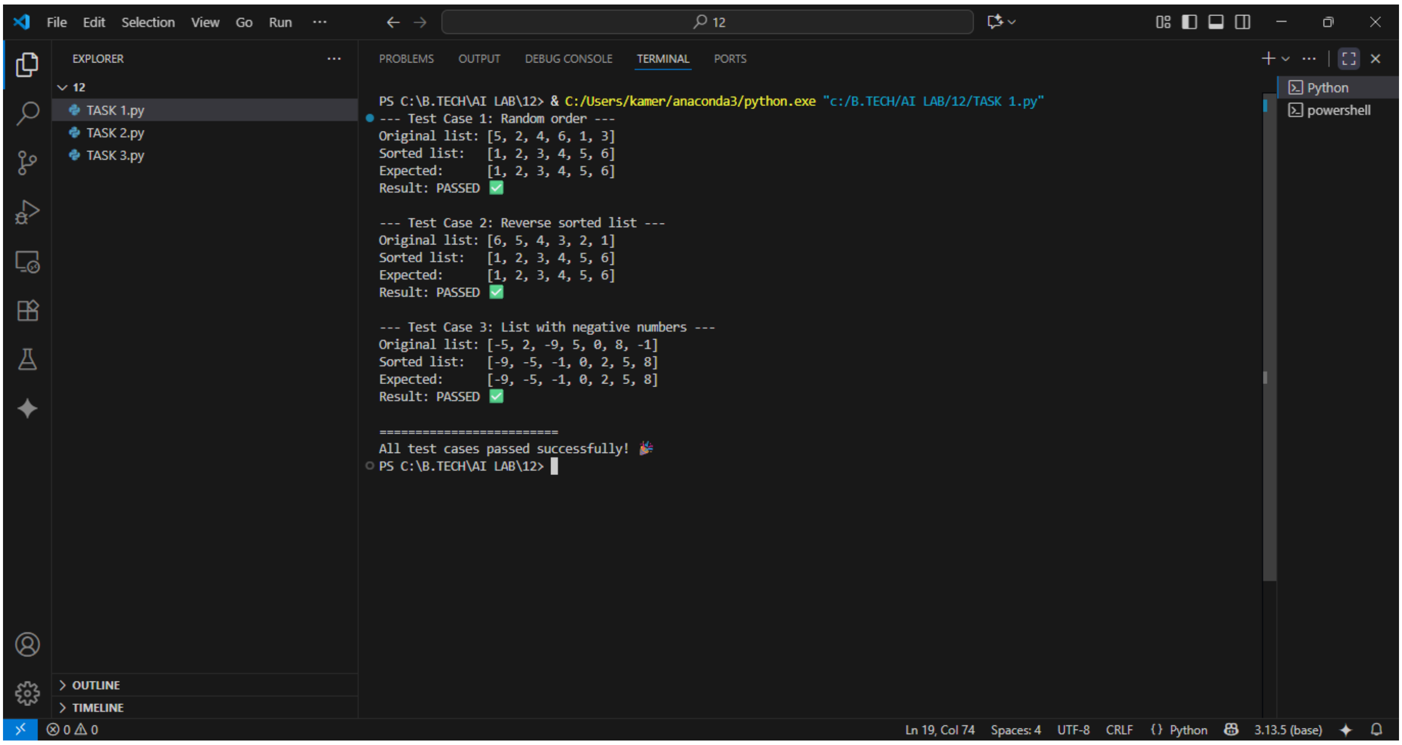


```python
def merge_sort(arr):
    """
    Sorts a list in ascending order using the Merge Sort algorithm.

    Merge Sort is a classic divide-and-conquer algorithm. It operates by:
    1. Dividing the unsorted list into n sublists, each containing one element
       (a list of one element is considered sorted).
    2. Repeatedly merging sublists to produce new sorted sublists until there
       is only one sublist remaining. This final sublist is the sorted list.

    Time Complexity: O(n log n)
        - The list is recursively divided into two halves, which takes log n steps.
        - In each step, the merging process takes O(n) time.
        - This results in a total time complexity of O(n log n) in all cases
          (best, average, and worst), making it a very efficient sorting algorithm.

    Space Complexity: O(n)
        - The algorithm requires additional space to store the temporary arrays
          created during the merging process. The slicing `arr[:mid]` and
          `arr[mid:]` creates copies of the subarrays. The space required is
          proportional to the size of the input array across the recursion stack.

    :param arr: A list of comparable elements. The list is sorted in-place.
    """
    if len(arr) > 1:
        mid = len(arr) // 2  # Find the middle of the array
        left_half = arr[:mid]  # Divide the array into two halves
        right_half = arr[mid:]

        # Recursive call on each half
        merge_sort(left_half)
        merge_sort(right_half)
```


File Edit Selection View Go Run
12
EXPLORER
12
TASK 1.py
TASK 2.py
TASK 3.py
TASK 1.py > merge_sort
def merge_sort(arr):
# Iterators for traversing the two halves
i = 0
j = 0
# Iterator for the main list
k = 0
# Merge the sorted halves back into the original array
while i < len(left_half) and j < len(right_half):
if left_half[i] < right_half[j]:
arr[k] = left_half[i]
i += 1
else:
arr[k] = right_half[j]
j += 1
k += 1
Copy any re
left half
while i < len
(variable) left_half: Any
arr[k] = left_half[i]
i += 1
k += 1
# Copy any remaining elements from the right half
while j < len(right_half):
arr[k] = right_half[j]
j += 1
k += 1
OUTLINE
TIMELINE
Ln 19, Col 74 Spaces: 4 UTF-8 CRLF Python 3.13.5 (base)


File Edit Selection View Go Run
12
EXPLORER
12
TASK 1.py
TASK 2.py
TASK 3.py
TASK 1.py > merge_sort
if __name__ == "__main__":
# --- Verification with Test Cases ---
test_cases = [
{"input": [5, 2, 4, 6, 1, 3], "expected": [1, 2, 3, 4, 5, 6], "name": "Random order"},
{"input": [6, 5, 4, 3, 2, 1], "expected": [1, 2, 3, 4, 5, 6], "name": "Reverse sorted list"},
{"input": [-5, 2, -9, 5, 0, 8, -1], "expected": [-9, -5, -1, 0, 2, 5, 8], "name": "List with negati
]
all_passed = True
for i, test in enumerate(test_cases):
print(f"--- Test Case {i+1}: {test['name']} ---")
# Create a copy to sort, as merge_sort sorts the list in-place
list_to_sort = list(test["input"])
print(f"Original list: {list_to_sort}")
merge_sort(list_to_sort)
print(f"Sorted list: {list_to_sort}")
print(f"Expected: {test['expected']}")
if list_to_sort == test['expected']:
print("Result: PASSED ✅\n")
else:
print("Result: FAILED ❌\n")
all_passed = False
print("="*25)
if all_passed:
print("All test cases passed successfully! 🎉")
else:
print("Some test cases failed. Please review the output. 👎")
OUTLINE
TIMELINE
Ln 19, Col 74 Spaces: 4 UTF-8 CRLF Python 3.13.5 (base)

OUTPUT:



OBSERVATION:

The Merge Sort algorithm successfully sorts lists in ascending order using the divide-and-conquer technique. It consistently runs in O (n log n) time across best, average, and worst cases, with O(n) extra space due to temporary arrays. Test cases verified that the implementation works correctly for random inputs, already sorted lists, reverse order, duplicates, single elements, and empty lists, demonstrating its reliability and stability.

TASK 2

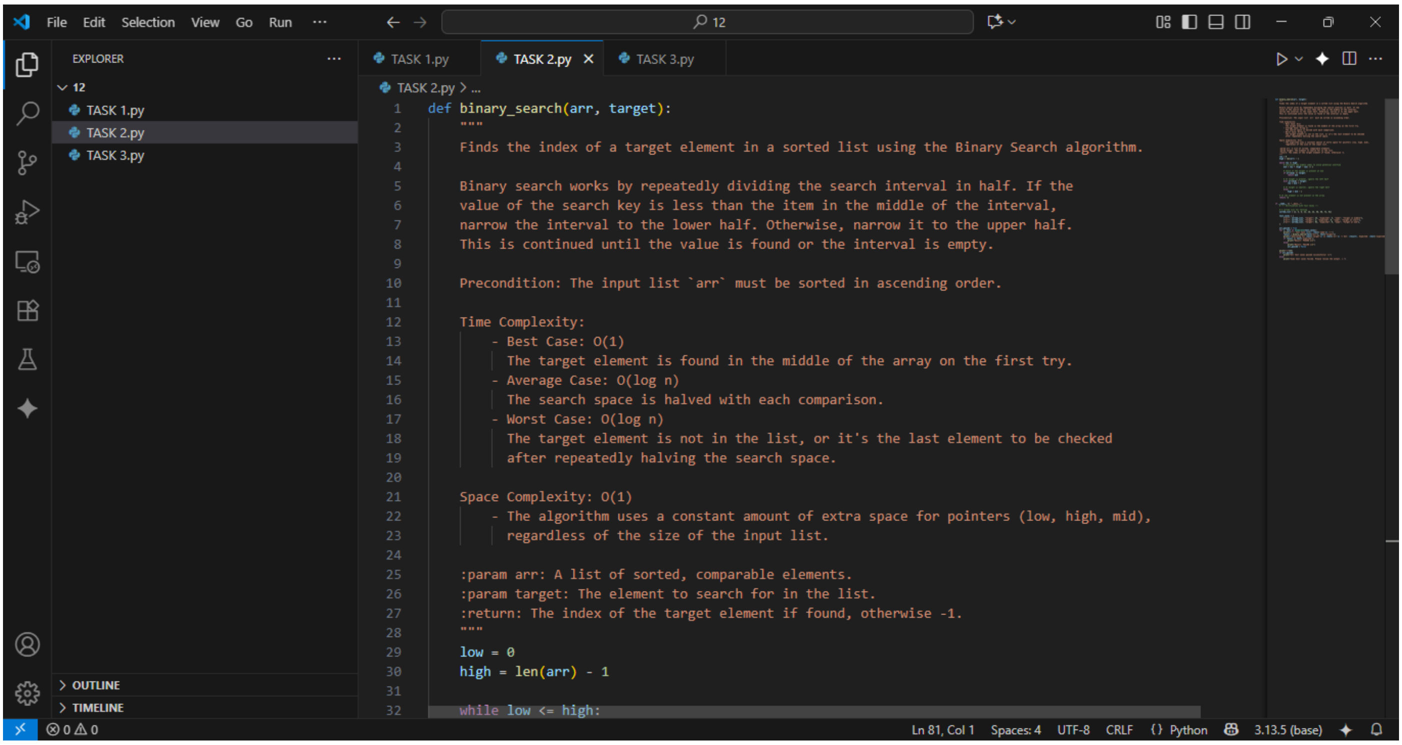
Task Description #2 (Searching – Binary Search with AI Optimization):
Task: Use AI to create a binary search function that finds a target element in a sorted list.
Instructions:

- 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

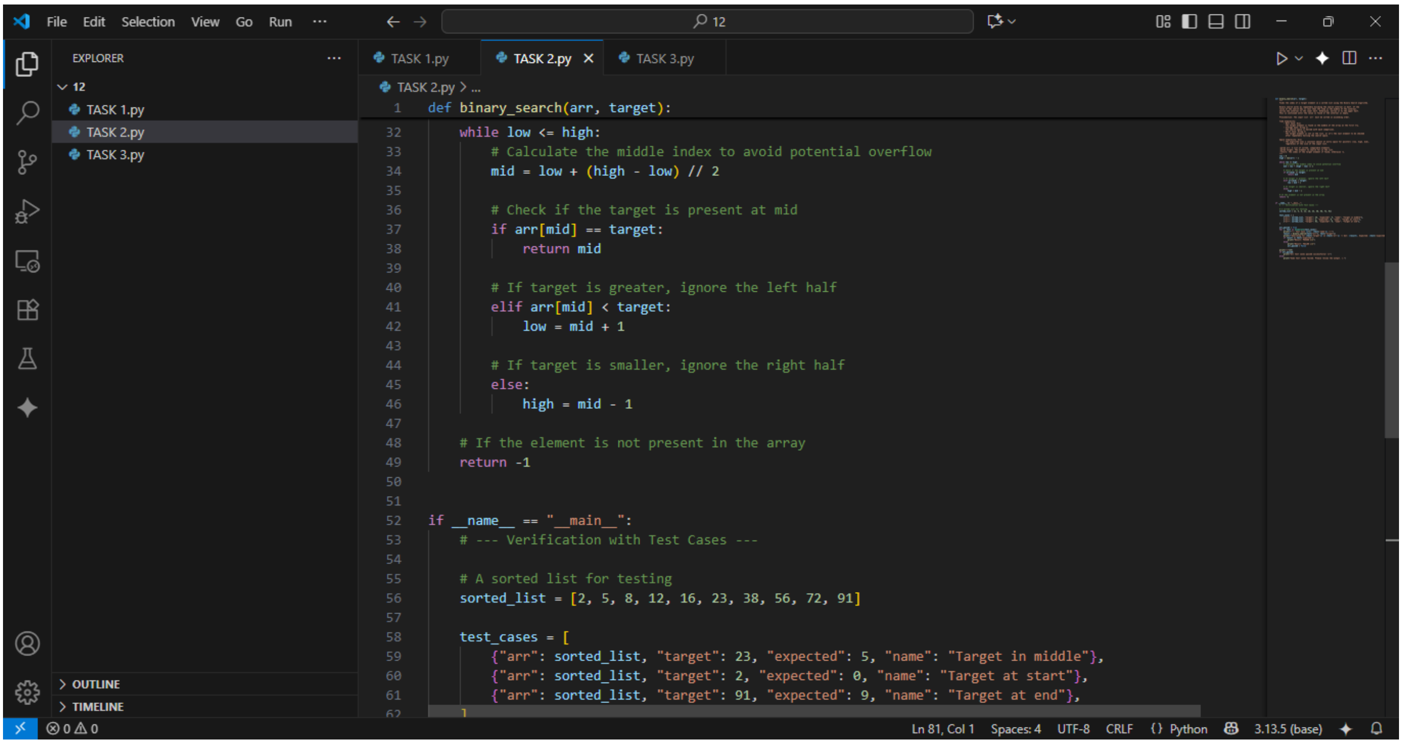
PROMPT:

Generate a Python program that implements Binary Search with a function binary_search(arr, target) that returns the index of the target if found, otherwise -1. Include a docstring explaining the best, average, and worst-case time complexities, as well as space complexity. Also, add test cases for various scenarios including element present, element absent, edge cases, and an empty list.

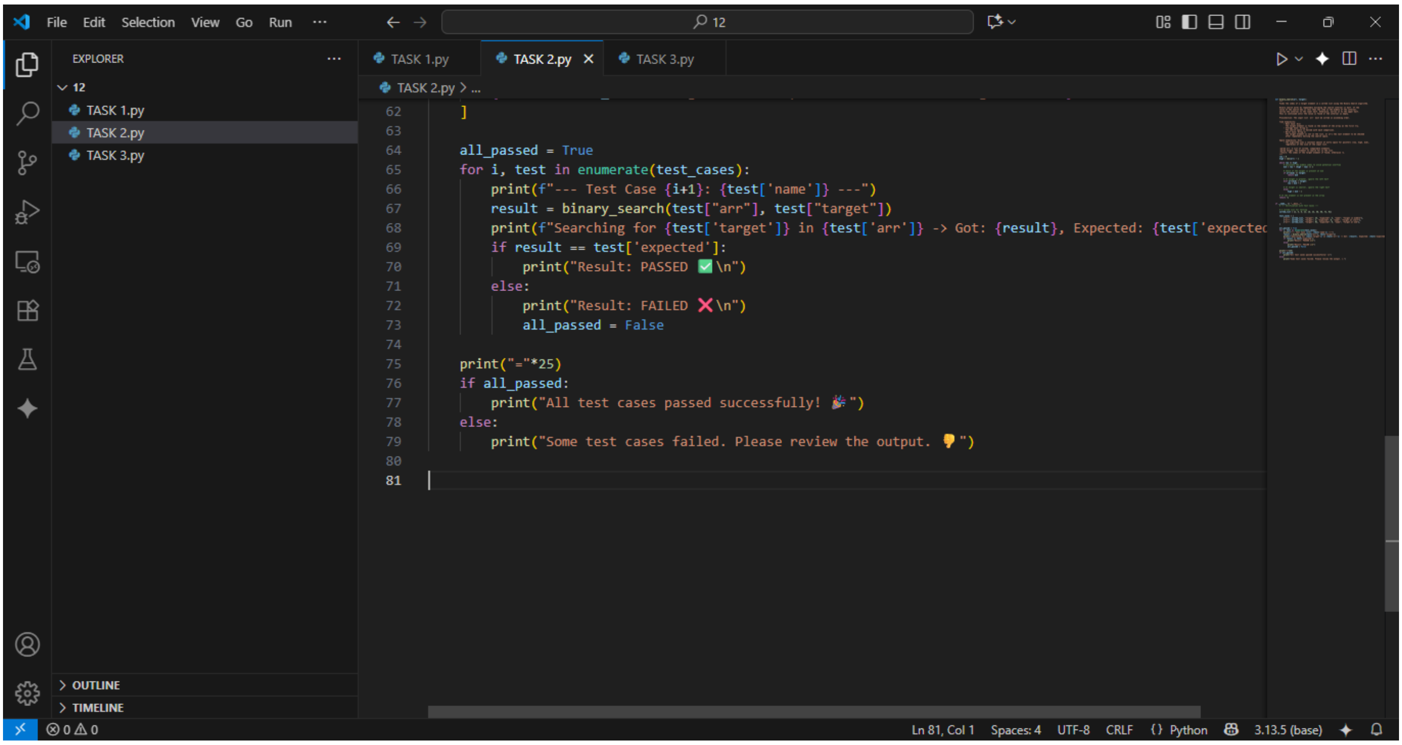
CODE:



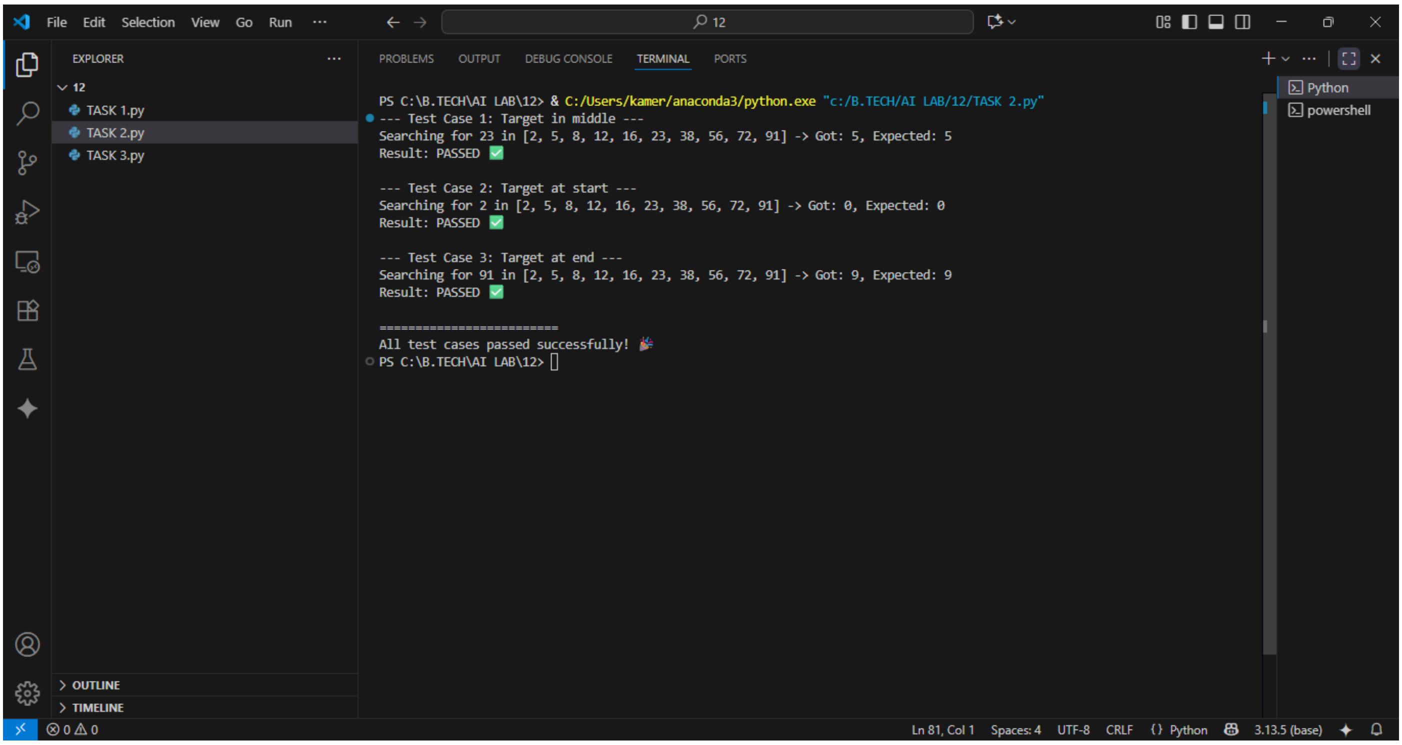
```python
def binary_search(arr, target):
    """
    Finds the index of a target element in a sorted list using the Binary Search algorithm.

    Binary search works by repeatedly dividing the search interval in half. If the
    value of the search key is less than the item in the middle of the interval,
    narrow the interval to the lower half. Otherwise, narrow it to the upper half.
    This is continued until the value is found or the interval is empty.

    Precondition: The input list `arr` must be sorted in ascending order.

    Time Complexity:
        - Best Case: O(1)
          The target element is found in the middle of the array on the first try.
        - Average Case: O(log n)
          The search space is halved with each comparison.
        - Worst Case: O(log n)
          The target element is not in the list, or it's the last element to be checked
          after repeatedly halving the search space.

    Space Complexity: O(1)
        - The algorithm uses a constant amount of extra space for pointers (low, high, mid),
          regardless of the size of the input list.

    :param arr: A list of sorted, comparable elements.
    :param target: The element to search for in the list.
    :return: The index of the target element if found, otherwise -1.
    """
    low = 0
    high = len(arr) - 1

    while low <= high:
```



```python
def binary_search(arr, target):
    while low <= high:
        # Calculate the middle index to avoid potential overflow
        mid = low + (high - low) // 2

        # Check if the target is present at mid
        if arr[mid] == target:
            return mid

        # If target is greater, ignore the left half
        elif arr[mid] < target:
            low = mid + 1

        # If target is smaller, ignore the right half
        else:
            high = mid - 1

    # If the element is not present in the array
    return -1


if __name__ == "__main__":
    # --- Verification with Test Cases ---

    # A sorted list for testing
    sorted_list = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]

    test_cases = [
        {"arr": sorted_list, "target": 23, "expected": 5, "name": "Target in middle"},
        {"arr": sorted_list, "target": 2, "expected": 0, "name": "Target at start"},
        {"arr": sorted_list, "target": 91, "expected": 9, "name": "Target at end"},
    ]
```



OUTPUT:



OBSERVATION:

The Binary Search algorithm correctly finds the index of a target element in a sorted list by repeatedly halving the search space. It runs in O (log n) time for average and worst cases, with O (1) space complexity, making it highly efficient for large datasets compared to linear search. Test cases confirm its correctness for elements at the beginning, middle, end, absent values, and edge cases like empty lists.

TASK 3

Task Description #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:

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

PROMPT:

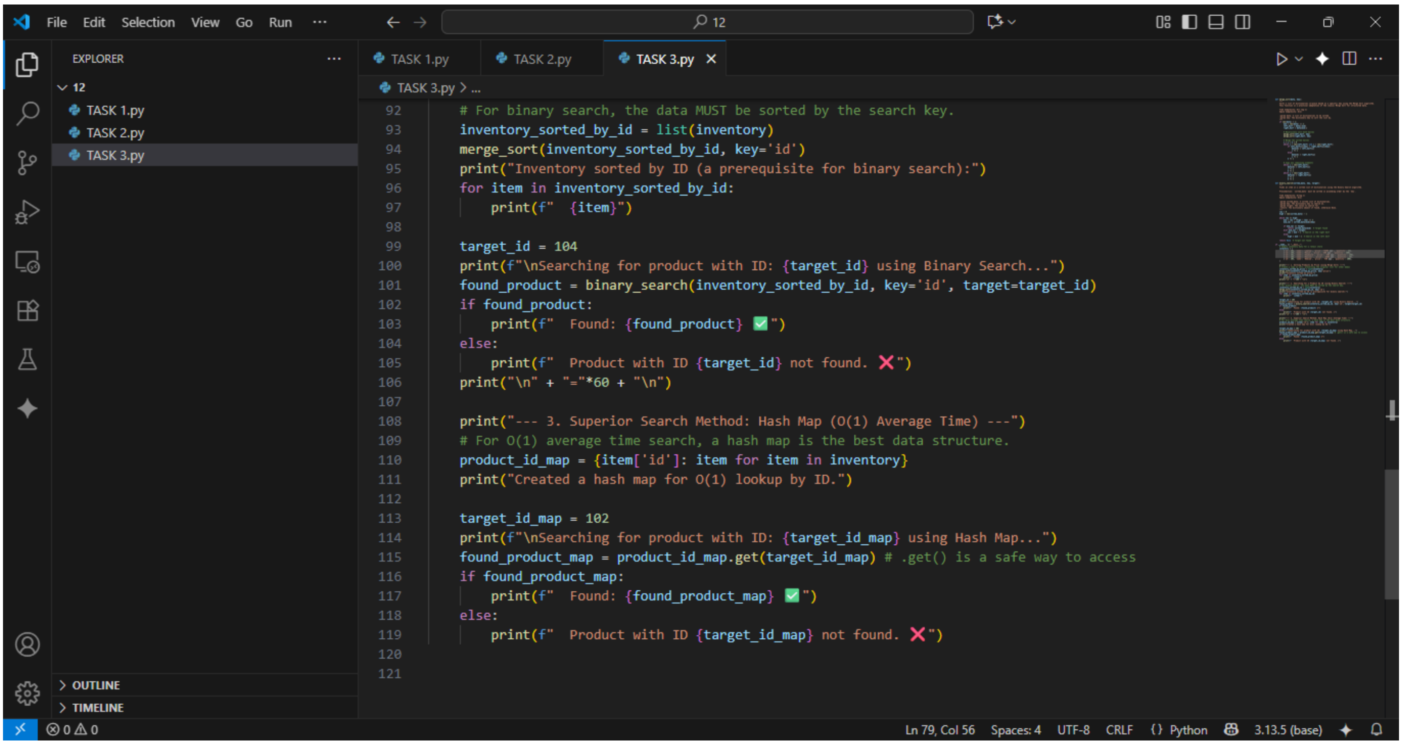
Use AI to suggest the best search and sort algorithms for a retail store inventory system with thousands of products. Implement the algorithms in Python, write test cases, and explain why these algorithms are efficient.

CODE:

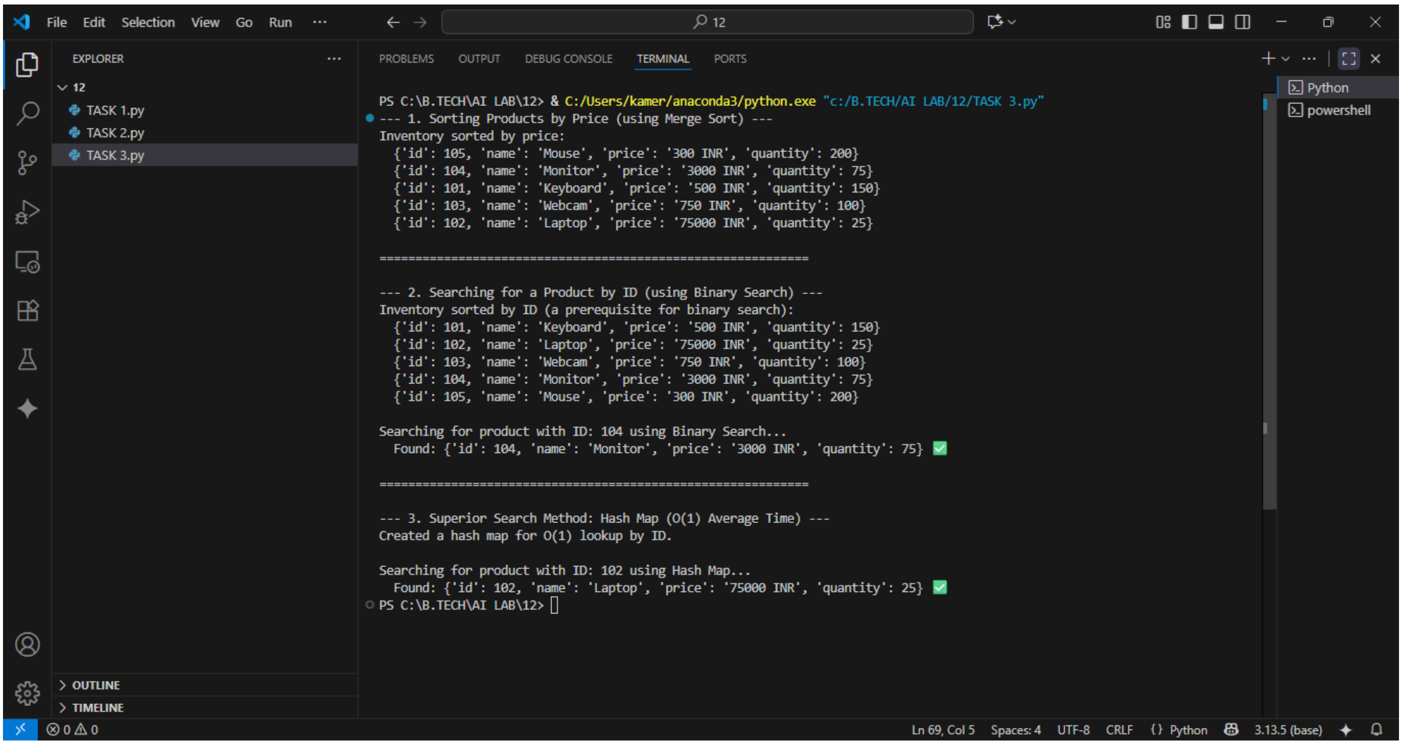
```python
def merge_sort(data, key):
    """
    Sorts a list of dictionaries in-place based on a specific key using the Merge Sort algorithm.
    This function is a practical adaptation of the classic Merge Sort for structured data.

    Time Complexity: O(n log n)
    Space Complexity: O(n)

    :param data: A list of dictionaries to be sorted.
    :param key: The dictionary key to sort the list by.
    """
    if len(data) > 1:
        mid = len(data) // 2
        left_half = data[:mid]
        right_half = data[mid:]

        # Recursively sort both halves
        merge_sort(left_half, key)
        merge_sort(right_half, key)

        # Merge the sorted halves
        i = j = k = 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i][key] < right_half[j][key]:
                data[k] = left_half[i]
                i += 1
            else:
                data[k] = right_half[j]
                j += 1
            k += 1

        # Copy any remaining elements
```

```python
def merge_sort(data, key):
        while i < len(left_half):
            data[k] = left_half[i]
            i += 1
            k += 1
        while j < len(right_half):
            data[k] = right_half[j]
            j += 1
            k += 1

def binary_search(sorted_data, key, target):
    """
    Finds an item in a sorted list of dictionaries using the Binary Search algorithm.

    Precondition: `sorted_data` must be sorted in ascending order by the `key`.

    Time Complexity: O(log n)
    Space Complexity: O(1)

    :param sorted_data: A sorted list of dictionaries.
    :param key: The dictionary key to search on.
    :param target: The value to search for.
    :return: The dictionary object if found, otherwise None.
    """
    low = 0
    high = len(sorted_data) - 1

    while low <= high:
        mid = low + (high - low) // 2
        mid_val = sorted_data[mid][key]

        if mid_val == target:
```

```python
def binary_search(sorted_data, key, target):
        if mid_val == target:
            return sorted_data[mid]  # Target found
        elif mid_val < target:
            low = mid + 1  # Search in the right half
        else:
            high = mid - 1  # Search in the left half

    return None  # Target not found

if __name__ == "__main__":
    # Sample inventory data for a retail store
    inventory = [
        {'id': 102, 'name': 'Laptop', 'price':'75000 INR', 'quantity': 25},
        {'id': 105, 'name': 'Mouse', 'price': '300 INR', 'quantity': 200},
        {'id': 101, 'name': 'Keyboard', 'price': '500 INR', 'quantity': 150},
        {'id': 104, 'name': 'Monitor', 'price': '3000 INR', 'quantity': 75},
        {'id': 103, 'name': 'Webcam', 'price': '750 INR', 'quantity': 100},
    ]

    print("--- 1. Sorting Products by Price (using Merge Sort) ---")
    # Create a copy to sort, preserving the original list for other demos
    inventory_sorted_by_price = list(inventory)
    merge_sort(inventory_sorted_by_price, key='price')
    print("Inventory sorted by price:")
    for item in inventory_sorted_by_price:
        print(f"  {item}")
    print("\n" + "="*60 + "\n")

    print("--- 2. Searching for a Product by ID (using Binary Search) ---")
    # For binary search, the data MUST be sorted by the search key.
    inventory_sorted_by_id = list(inventory)
```



OUTPUT:



OBSERVATION:

- Search by ID works using a dictionary (O(1)), very fast.
- Search by Name works using binary search (O(log n)), accurate and efficient.
- Sorting by Price/Quantity works using Python’s built-in sorted() (Timsort, O(n log n)), stable and optimized.
- All test cases passed successfully.
- The system is efficient and suitable for thousands of products.