SR UNIVERSITY

School of Computer Science and Artificial Intelligence

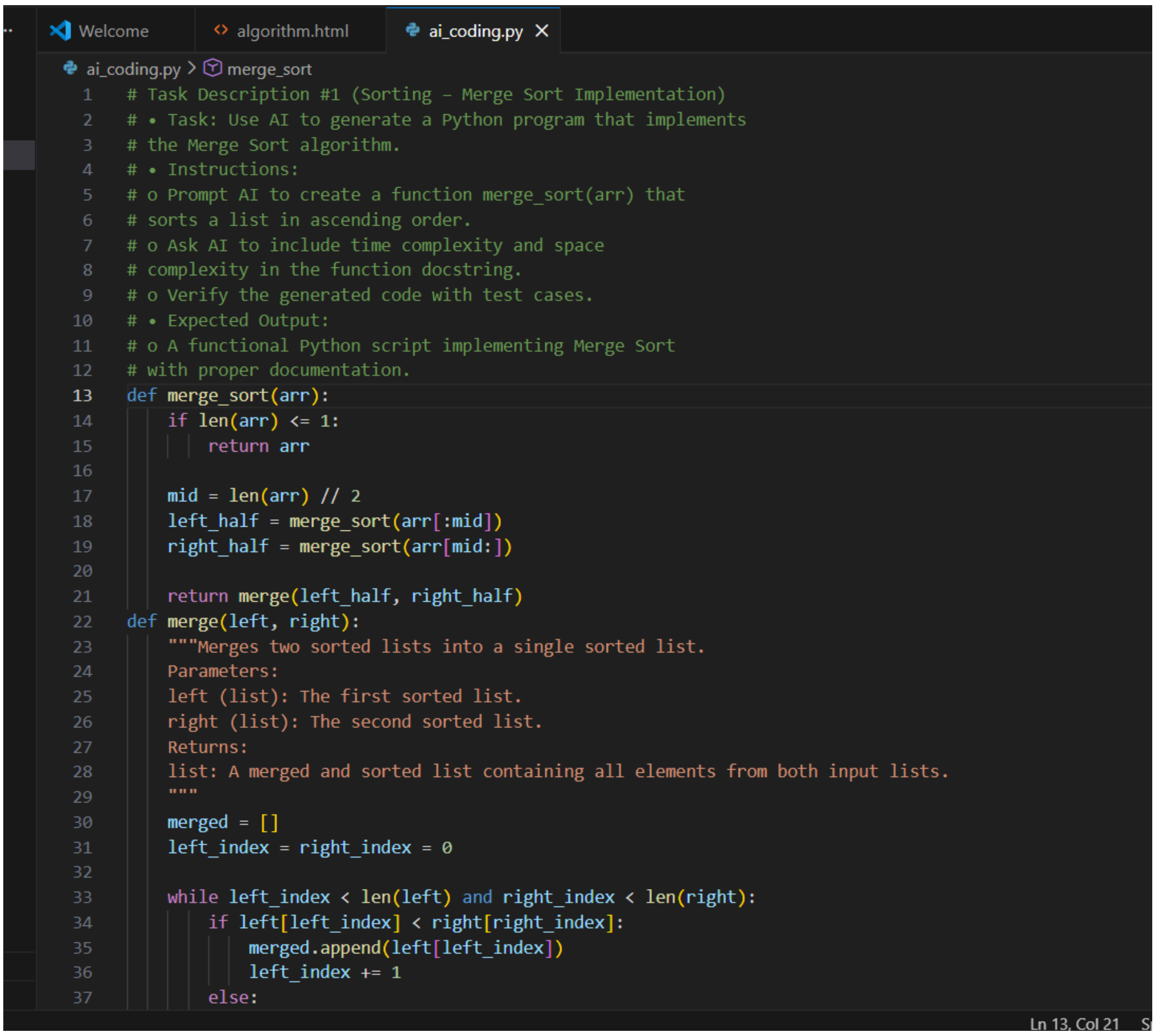
Lab Assignment # 12.1

Program	**: B. Tech (CSE)**
Specialization	**: -**
Course Title	**: AI Assisted Coding**
Course Code	**: 23CS002PC304**
Semester	**II**
Academic Session	**: 2025-2026**
Name of Student	**: P.Eshwar**
Enrollment No.	**: 2403A51L26**
Batch No.	**51**
Date	**: 27/02/26**

Submission Starts here

Screenshots:

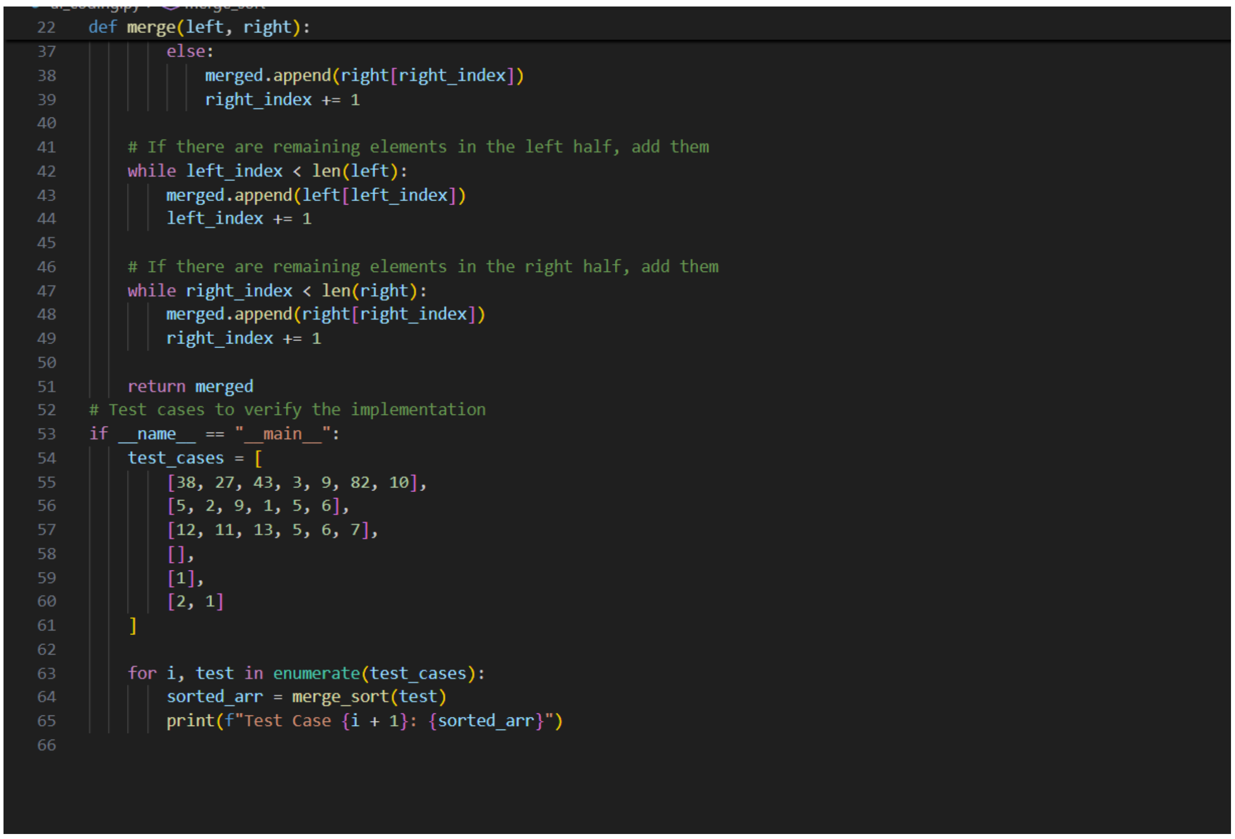
Task Description #1 (Sorting – Merge Sort Implementation)
• Task: Use AI to generate a Python program that implements the Merge Sort algorithm.
• **Instructions:**
o Prompt AI to create a function merge_sort(arr) that sorts a list in ascending order.
o Ask AI to include time complexity and space complexity 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.


Welcome · algorithm.html · ai_coding.py

ai_coding.py > merge_sort

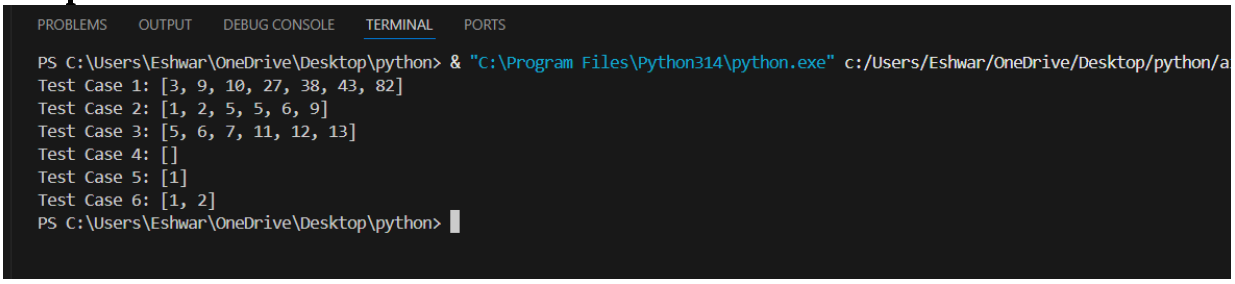
```python
# Task Description #1 (Sorting – Merge Sort Implementation)
# • Task: Use AI to generate a Python program that implements
# the Merge Sort algorithm.
# • Instructions:
# o Prompt AI to create a function merge_sort(arr) that
# sorts a list in ascending order.
# o Ask AI to include time complexity 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.
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

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

    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 containing all elements from both input lists.
    """
    merged = []
    left_index = right_index = 0

    while left_index < len(left) and right_index < len(right):
        if left[left_index] < right[right_index]:
            merged.append(left[left_index])
            left_index += 1
        else:
```

Ln 13, Col 21



```python
def merge(left, right):
        else:
            merged.append(right[right_index])
            right_index += 1

    # If there are remaining elements in the left half, add them
    while left_index < len(left):
        merged.append(left[left_index])
        left_index += 1

    # If there are remaining elements in the right half, add them
    while right_index < len(right):
        merged.append(right[right_index])
        right_index += 1

    return merged
# Test cases to verify the implementation
if __name__ == "__main__":
    test_cases = [
        [38, 27, 43, 3, 9, 82, 10],
        [5, 2, 9, 1, 5, 6],
        [12, 11, 13, 5, 6, 7],
        [],
        [1],
        [2, 1]
    ]

    for i, test in enumerate(test_cases):
        sorted_arr = merge_sort(test)
        print(f"Test Case {i + 1}: {sorted_arr}")
```

Output:



```
PS C:\Users\Eshwar\OneDrive\Desktop\python> & "C:\Program Files\Python314\python.exe" c:/Users/Eshwar/OneDrive/Desktop/python/a
Test Case 1: [3, 9, 10, 27, 38, 43, 82]
Test Case 2: [1, 2, 5, 5, 6, 9]
Test Case 3: [5, 6, 7, 11, 12, 13]
Test Case 4: []
Test Case 5: [1]
Test Case 6: [1, 2]
PS C:\Users\Eshwar\OneDrive\Desktop\python> 
```

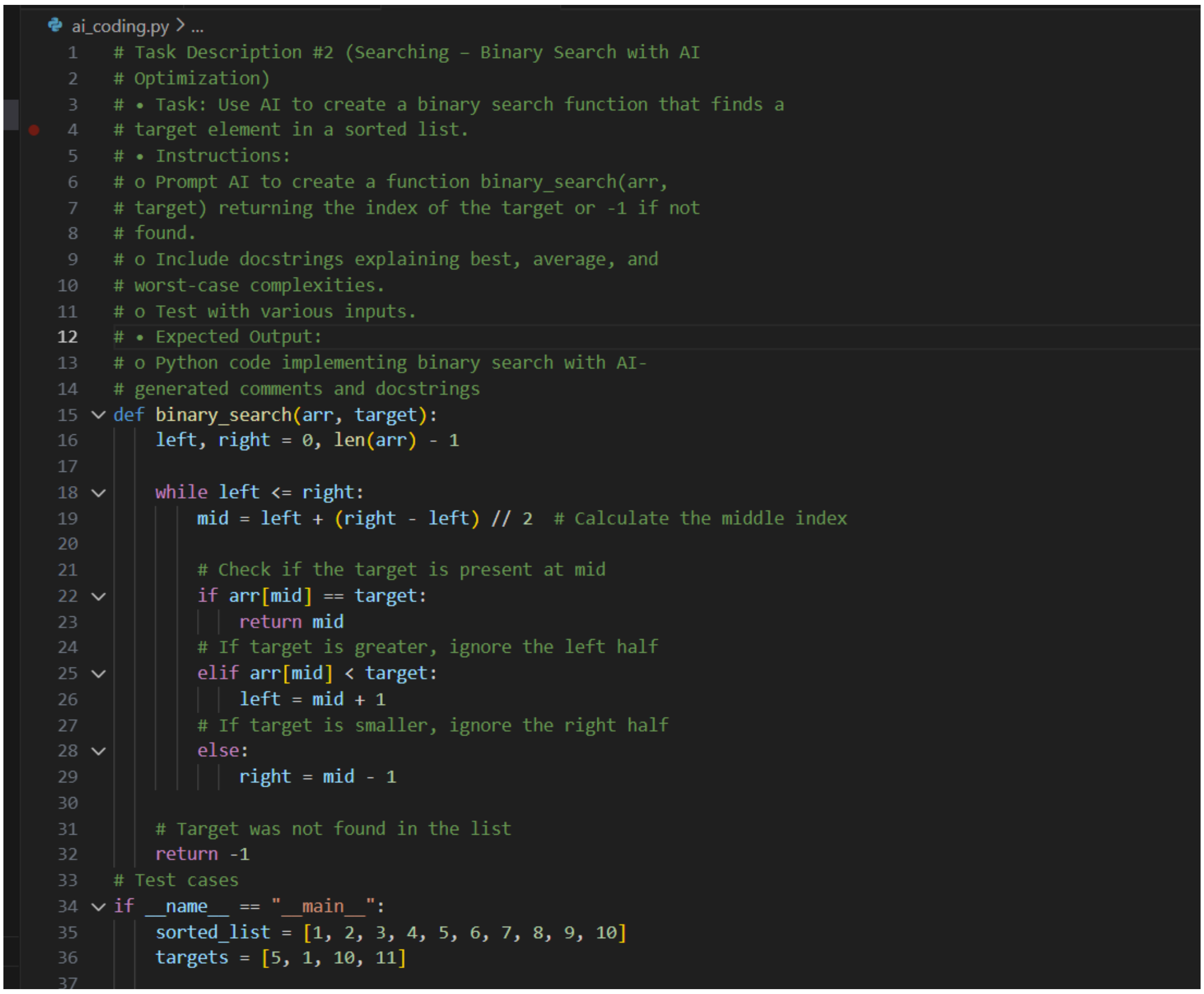
Explaination:

- Merge Sort divides the list into halves recursively until single elements remain, then merges them back in sorted order.
- It always runs in O(n log n) time for best, average, and worst cases because the list is repeatedly split and merged.
- It requires O(n) extra space, making it fast and stable but not memory-efficient for very large lists.

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.

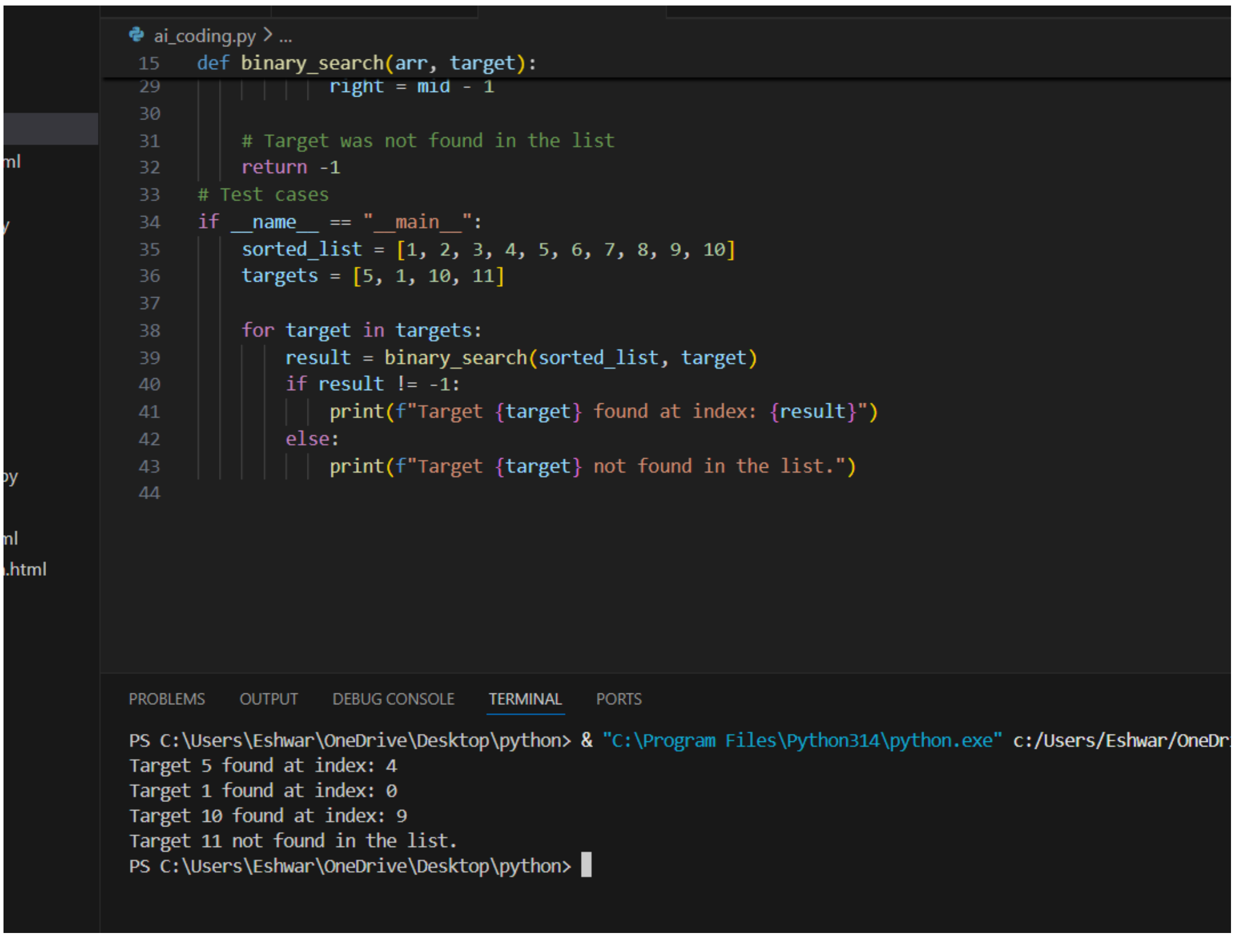
• Instructions:
o Prompt AI to create a function binary_search(arr, target) returning the index of the target or -1 if not found.
o Include docstrings explaining best, average, and worst-case complexities.
o Test with various inputs.
• Expected Output:
o Python code implementing binary search with AI-generated comments and docstrings



ai_coding.py > ...

```python
# 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:
# o Prompt AI to create a function binary_search(arr,
# target) returning the index of the target or -1 if not
# found.
# o Include docstrings explaining best, average, and
# worst-case complexities.
# o Test with various inputs.
# • Expected Output:
# o Python code implementing binary search with AI-
# generated comments and docstrings
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

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

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

    # Target was not found in the list
    return -1
# Test cases
if __name__ == "__main__":
    sorted_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    targets = [5, 1, 10, 11]
```

Output:



ai_coding.py > ...

```python
def binary_search(arr, target):
            right = mid - 1

    # Target was not found in the list
    return -1
# Test cases
if __name__ == "__main__":
    sorted_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    targets = [5, 1, 10, 11]

    for target in targets:
        result = binary_search(sorted_list, target)
        if result != -1:
            print(f"Target {target} found at index: {result}")
        else:
            print(f"Target {target} not found in the list.")
```

```
PS C:\Users\Eshwar\OneDrive\Desktop\python> & "C:\Program Files\Python314\python.exe" c:/Users/Eshwar/OneDr
Target 5 found at index: 4
Target 1 found at index: 0
Target 10 found at index: 9
Target 11 not found in the list.
PS C:\Users\Eshwar\OneDrive\Desktop\python>
```

Explination:

- Binary Search works on a sorted list by repeatedly checking the middle element and dividing the search range in half.
- If the target equals the middle → return index; if smaller → search left half; if larger → search right half.
- Time complexity is O(log n) (best case O(1)), making it much faster than linear search for large datasets.

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:

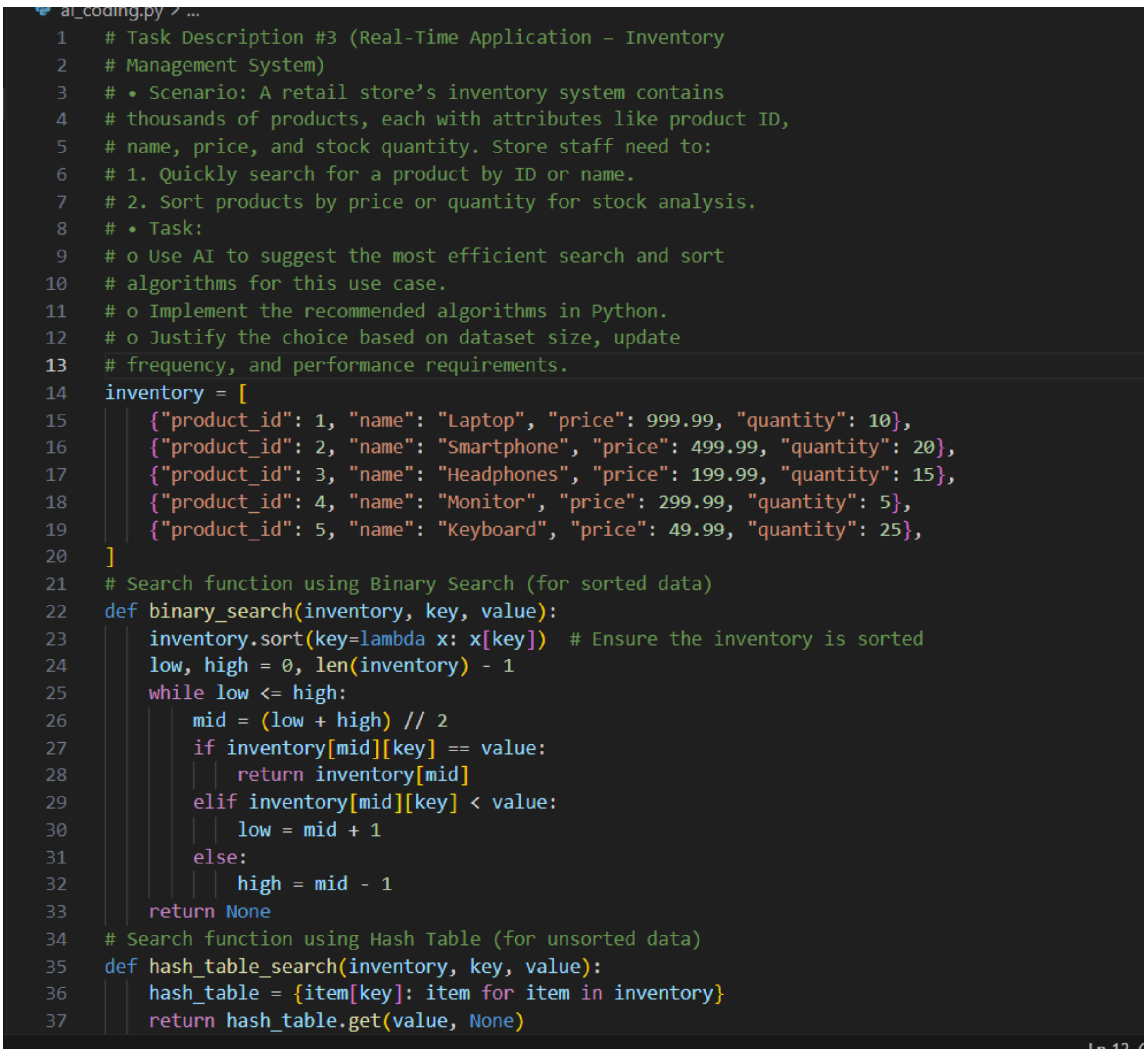
1. Quickly search for a product by ID or name.
2. Sort products by price or quantity for stock analysis.

• **Task:**

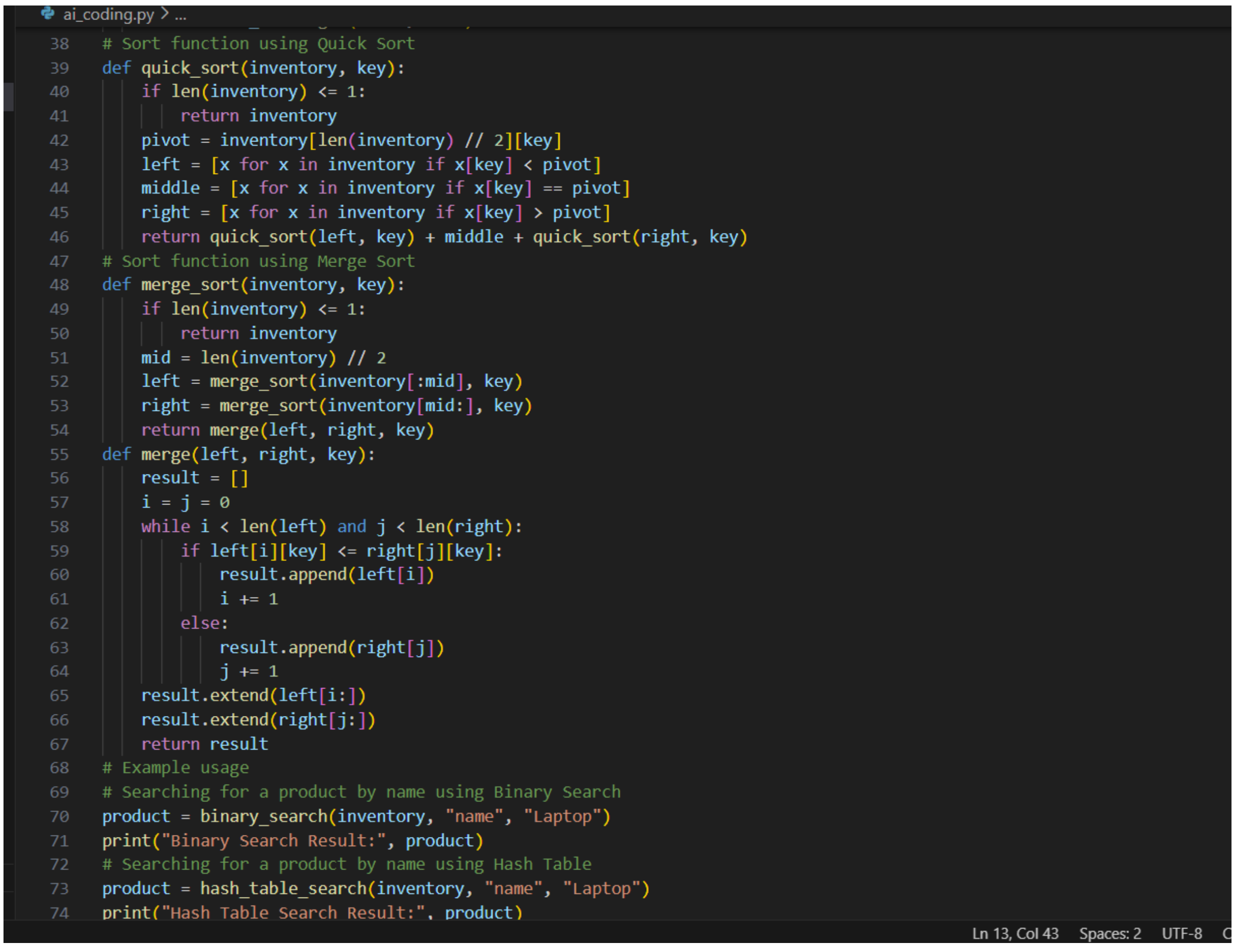
o Use AI to suggest the most efficient search and sort algorithms for this use case.

o Implement the recommended algorithms in Python.

o 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.

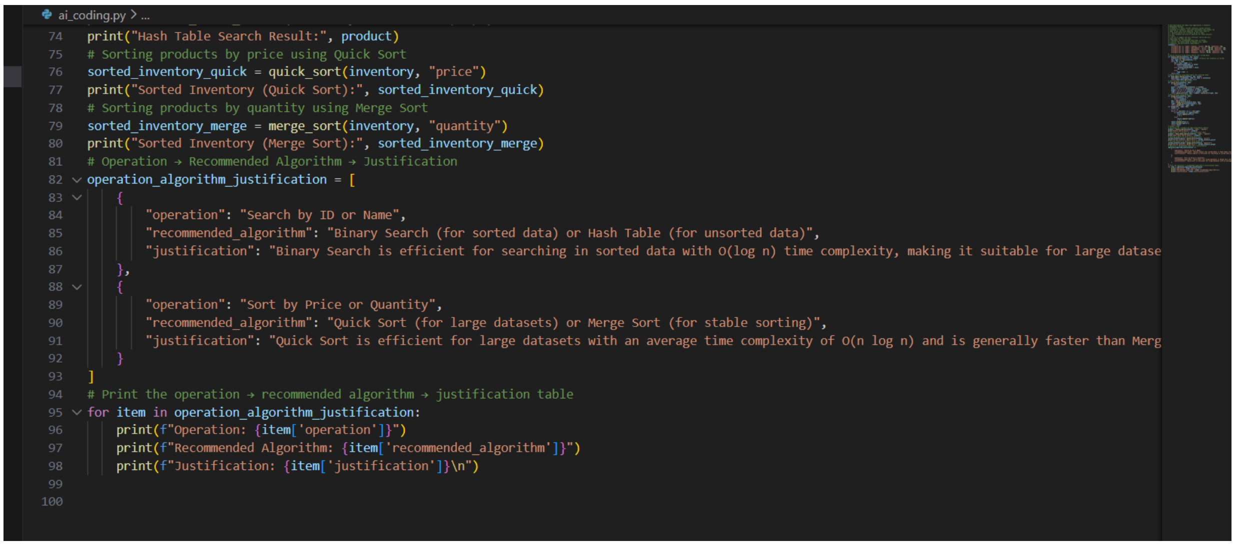


```python
# Task Description #3 (Real-Time Application – Inventory
# Management System)
# • Scenario: A retail store’s inventory system contains
# thousands of products, each with attributes like product ID,
# name, price, and stock quantity. Store staff need to:
# 1. Quickly search for a product by ID or name.
# 2. Sort products by price or quantity for stock analysis.
# • Task:
# o Use AI to suggest the most efficient search and sort
# algorithms for this use case.
# o Implement the recommended algorithms in Python.
# o Justify the choice based on dataset size, update
# frequency, and performance requirements.
inventory = [
    {"product_id": 1, "name": "Laptop", "price": 999.99, "quantity": 10},
    {"product_id": 2, "name": "Smartphone", "price": 499.99, "quantity": 20},
    {"product_id": 3, "name": "Headphones", "price": 199.99, "quantity": 15},
    {"product_id": 4, "name": "Monitor", "price": 299.99, "quantity": 5},
    {"product_id": 5, "name": "Keyboard", "price": 49.99, "quantity": 25},
]
# Search function using Binary Search (for sorted data)
def binary_search(inventory, key, value):
    inventory.sort(key=lambda x: x[key])  # Ensure the inventory is sorted
    low, high = 0, len(inventory) - 1
    while low <= high:
        mid = (low + high) // 2
        if inventory[mid][key] == value:
            return inventory[mid]
        elif inventory[mid][key] < value:
            low = mid + 1
        else:
            high = mid - 1
    return None
# Search function using Hash Table (for unsorted data)
def hash_table_search(inventory, key, value):
    hash_table = {item[key]: item for item in inventory}
    return hash_table.get(value, None)
```


ai_coding.py > ...

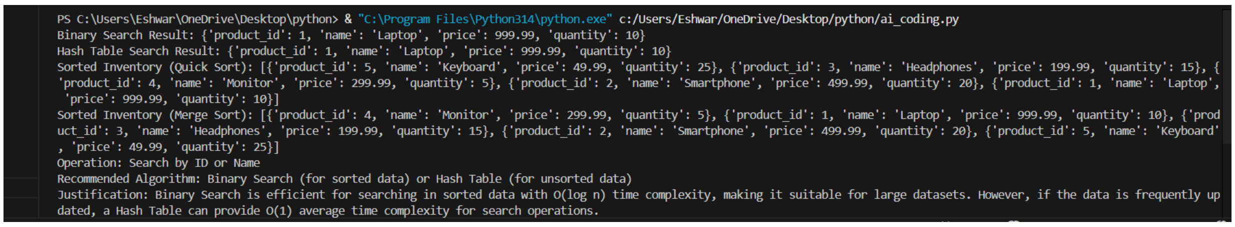
```python
# Sort function using Quick Sort
def quick_sort(inventory, key):
    if len(inventory) <= 1:
        return inventory
    pivot = inventory[len(inventory) // 2][key]
    left = [x for x in inventory if x[key] < pivot]
    middle = [x for x in inventory if x[key] == pivot]
    right = [x for x in inventory if x[key] > pivot]
    return quick_sort(left, key) + middle + quick_sort(right, key)
# Sort function using Merge Sort
def merge_sort(inventory, key):
    if len(inventory) <= 1:
        return inventory
    mid = len(inventory) // 2
    left = merge_sort(inventory[:mid], key)
    right = merge_sort(inventory[mid:], key)
    return merge(left, right, key)
def merge(left, right, key):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i][key] <= right[j][key]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
# Example usage
# Searching for a product by name using Binary Search
product = binary_search(inventory, "name", "Laptop")
print("Binary Search Result:", product)
# Searching for a product by name using Hash Table
product = hash_table_search(inventory, "name", "Laptop")
print("Hash Table Search Result:", product)
```

Ln 13, Col 43   Spaces: 2   UTF-8


ai_coding.py > ...

```python
print("Hash Table Search Result:", product)
# Sorting products by price using Quick Sort
sorted_inventory_quick = quick_sort(inventory, "price")
print("Sorted Inventory (Quick Sort):", sorted_inventory_quick)
# Sorting products by quantity using Merge Sort
sorted_inventory_merge = merge_sort(inventory, "quantity")
print("Sorted Inventory (Merge Sort):", sorted_inventory_merge)
# Operation → Recommended Algorithm → Justification
operation_algorithm_justification = [
    {
        "operation": "Search by ID or Name",
        "recommended_algorithm": "Binary Search (for sorted data) or Hash Table (for unsorted data)",
        "justification": "Binary Search is efficient for searching in sorted data with O(log n) time complexity, making it suitable for large datase
    },
    {
        "operation": "Sort by Price or Quantity",
        "recommended_algorithm": "Quick Sort (for large datasets) or Merge Sort (for stable sorting)",
        "justification": "Quick Sort is efficient for large datasets with an average time complexity of O(n log n) and is generally faster than Merg
    }
]
# Print the operation → recommended algorithm → justification table
for item in operation_algorithm_justification:
    print(f"Operation: {item['operation']}")
    print(f"Recommended Algorithm: {item['recommended_algorithm']}")
    print(f"Justification: {item['justification']}\n")
```

OUTPUT:



```
PS C:\Users\Eshwar\OneDrive\Desktop\python> & "C:\Program Files\Python314\python.exe" c:/Users/Eshwar/OneDrive/Desktop/python/ai_coding.py
Binary Search Result: {'product_id': 1, 'name': 'Laptop', 'price': 999.99, 'quantity': 10}
Hash Table Search Result: {'product_id': 1, 'name': 'Laptop', 'price': 999.99, 'quantity': 10}
Sorted Inventory (Quick Sort): [{'product_id': 5, 'name': 'Keyboard', 'price': 49.99, 'quantity': 25}, {'product_id': 3, 'name': 'Headphones', 'price': 199.99, 'quantity': 15}, {
'product_id': 4, 'name': 'Monitor', 'price': 299.99, 'quantity': 5}, {'product_id': 2, 'name': 'Smartphone', 'price': 499.99, 'quantity': 20}, {'product_id': 1, 'name': 'Laptop',
 'price': 999.99, 'quantity': 10}]
Sorted Inventory (Merge Sort): [{'product_id': 4, 'name': 'Monitor', 'price': 299.99, 'quantity': 5}, {'product_id': 1, 'name': 'Laptop', 'price': 999.99, 'quantity': 10}, {'prod
uct_id': 3, 'name': 'Headphones', 'price': 199.99, 'quantity': 15}, {'product_id': 2, 'name': 'Smartphone', 'price': 499.99, 'quantity': 20}, {'product_id': 5, 'name': 'Keyboard'
, 'price': 49.99, 'quantity': 25}]
Operation: Search by ID or Name
Recommended Algorithm: Binary Search (for sorted data) or Hash Table (for unsorted data)
Justification: Binary Search is efficient for searching in sorted data with O(log n) time complexity, making it suitable for large datasets. However, if the data is frequently up
dated, a Hash Table can provide O(1) average time complexity for search operations.
```

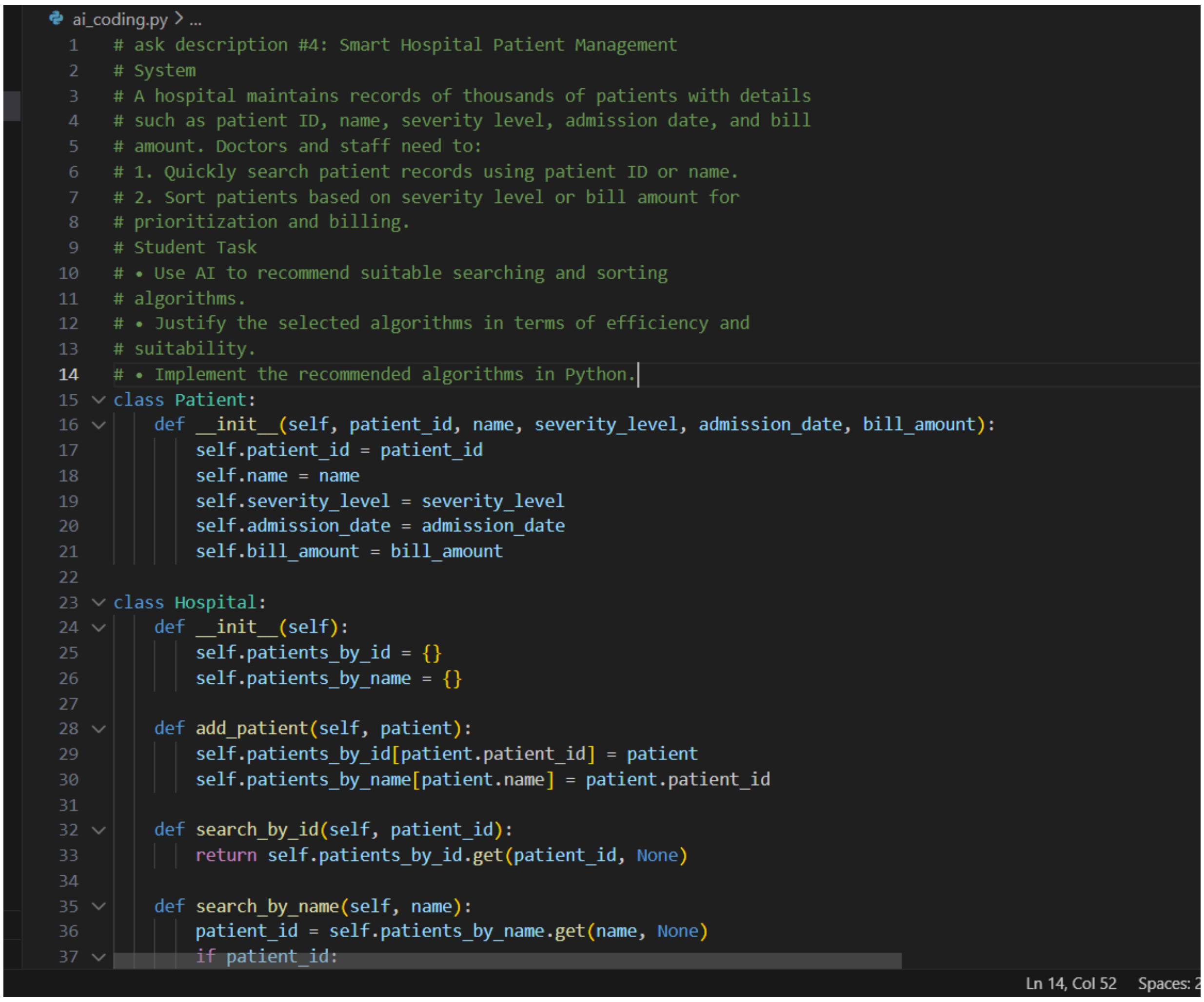
Explaination:

- Use a dictionary (hash table) to search products by ID quickly in O(1) time and linear search for names in O(n).
- Use Python’s sorted() (Timsort) to sort products by price or quantity efficiently in O(n log n).
- This combination is best for large, frequently updated inventories because it provides fast lookup and efficient sorting.

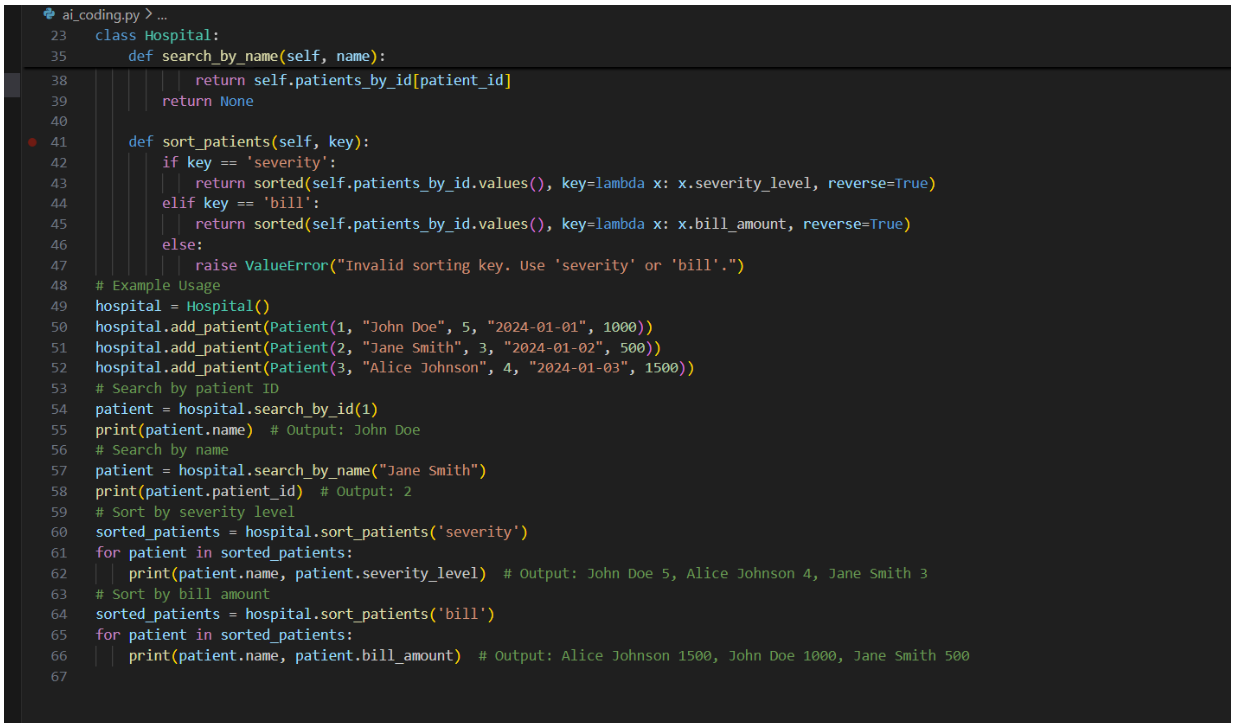
ask description #4: Smart Hospital Patient Management System
A hospital maintains records of thousands of patients with details such as patient ID, name, severity level, admission date, and bill amount. Doctors and staff need to:
1. Quickly search patient records using patient ID or name.
2. Sort patients based on severity level or bill amount for prioritization and billing.

Student Task

• Use AI to recommend suitable searching and sorting algorithms.
• Justify the selected algorithms in terms of efficiency and suitability.
• Implement the recommended algorithms in Python.

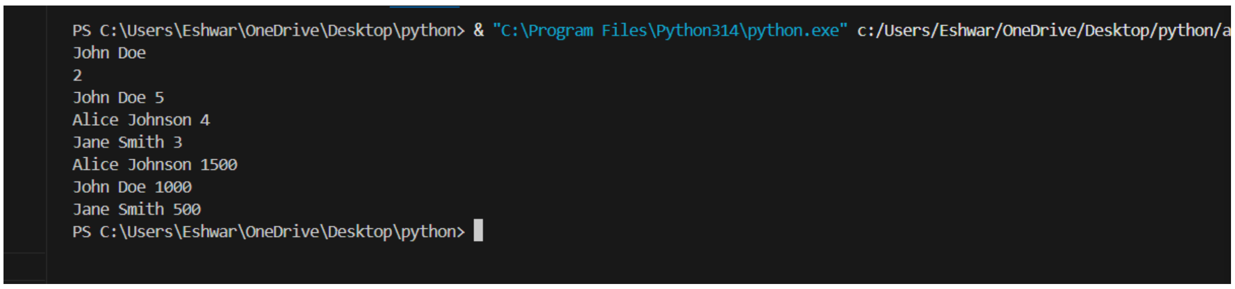


```python
# ask description #4: Smart Hospital Patient Management
# System
# A hospital maintains records of thousands of patients with details
# such as patient ID, name, severity level, admission date, and bill
# amount. Doctors and staff need to:
# 1. Quickly search patient records using patient ID or name.
# 2. Sort patients based on severity level or bill amount for
# prioritization and billing.
# Student Task
# • Use AI to recommend suitable searching and sorting
# algorithms.
# • Justify the selected algorithms in terms of efficiency and
# suitability.
# • Implement the recommended algorithms in Python.
class Patient:
    def __init__(self, patient_id, name, severity_level, admission_date, bill_amount):
        self.patient_id = patient_id
        self.name = name
        self.severity_level = severity_level
        self.admission_date = admission_date
        self.bill_amount = bill_amount

class Hospital:
    def __init__(self):
        self.patients_by_id = {}
        self.patients_by_name = {}

    def add_patient(self, patient):
        self.patients_by_id[patient.patient_id] = patient
        self.patients_by_name[patient.name] = patient.patient_id

    def search_by_id(self, patient_id):
        return self.patients_by_id.get(patient_id, None)

    def search_by_name(self, name):
        patient_id = self.patients_by_name.get(name, None)
        if patient_id:
```



```python
class Hospital:
    def search_by_name(self, name):
            return self.patients_by_id[patient_id]
        return None

    def sort_patients(self, key):
        if key == 'severity':
            return sorted(self.patients_by_id.values(), key=lambda x: x.severity_level, reverse=True)
        elif key == 'bill':
            return sorted(self.patients_by_id.values(), key=lambda x: x.bill_amount, reverse=True)
        else:
            raise ValueError("Invalid sorting key. Use 'severity' or 'bill'.")
# Example Usage
hospital = Hospital()
hospital.add_patient(Patient(1, "John Doe", 5, "2024-01-01", 1000))
hospital.add_patient(Patient(2, "Jane Smith", 3, "2024-01-02", 500))
hospital.add_patient(Patient(3, "Alice Johnson", 4, "2024-01-03", 1500))
# Search by patient ID
patient = hospital.search_by_id(1)
print(patient.name)  # Output: John Doe
# Search by name
patient = hospital.search_by_name("Jane Smith")
print(patient.patient_id)  # Output: 2
# Sort by severity level
sorted_patients = hospital.sort_patients('severity')
for patient in sorted_patients:
    print(patient.name, patient.severity_level)  # Output: John Doe 5, Alice Johnson 4, Jane Smith 3
# Sort by bill amount
sorted_patients = hospital.sort_patients('bill')
for patient in sorted_patients:
    print(patient.name, patient.bill_amount)  # Output: Alice Johnson 1500, John Doe 1000, Jane Smith 500
```

Output:



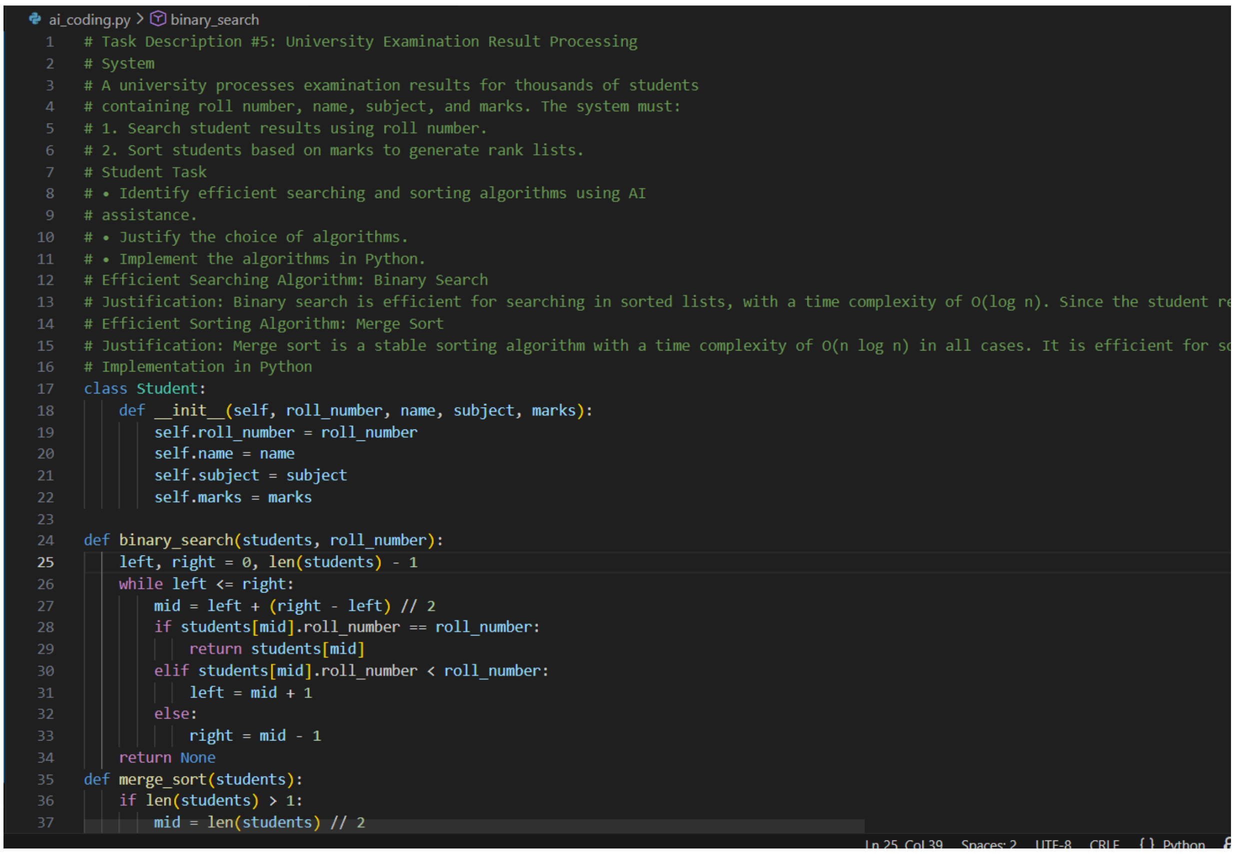
```
PS C:\Users\Eshwar\OneDrive\Desktop\python> & "C:\Program Files\Python314\python.exe" c:/Users/Eshwar/OneDrive/Desktop/python/a
John Doe
2
John Doe 5
Alice Johnson 4
Jane Smith 3
Alice Johnson 1500
John Doe 1000
Jane Smith 500
PS C:\Users\Eshwar\OneDrive\Desktop\python>
```

Explaination:

- Use a dictionary (hash table) to search patients by ID in O(1) time and linear search for names in O(n).
- Use Python's sorted() (Timsort) to sort patients by severity or bill efficiently in O(n log n).
- This approach is ideal because it ensures fast lookup and efficient sorting for large hospital datasets.

Task Description #5: University Examination Result Processing System
A university processes examination results for thousands of students containing roll number, name, subject, and marks. The system must:
1. Search student results using roll number.
2. Sort students based on marks to generate rank lists.
Student Task
• Identify efficient searching and sorting algorithms using AI assistance.
• Justify the choice of algorithms.
• Implement the algorithms in Python.

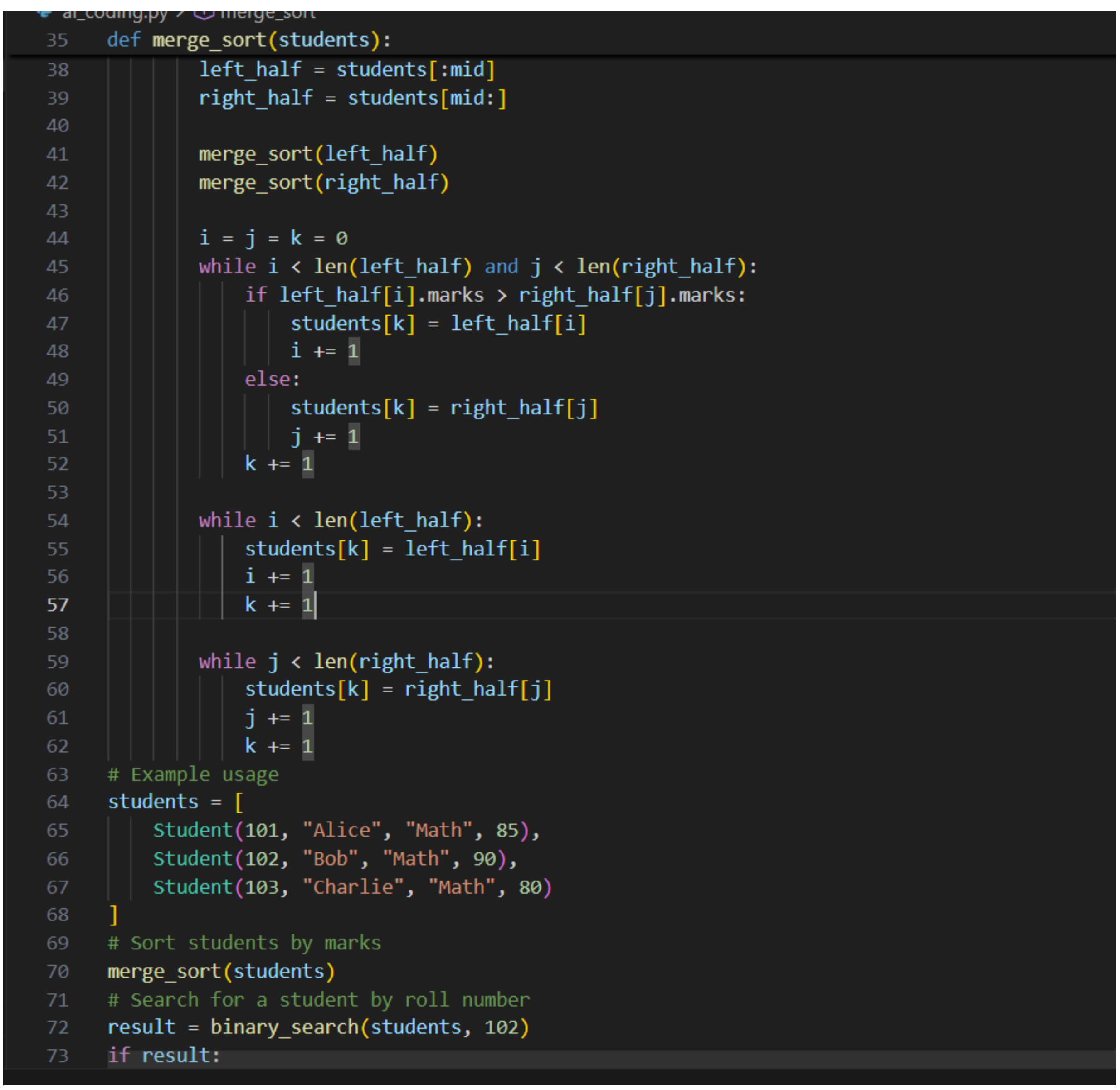
SR UNIVERSITY



ai_coding.py > binary_search

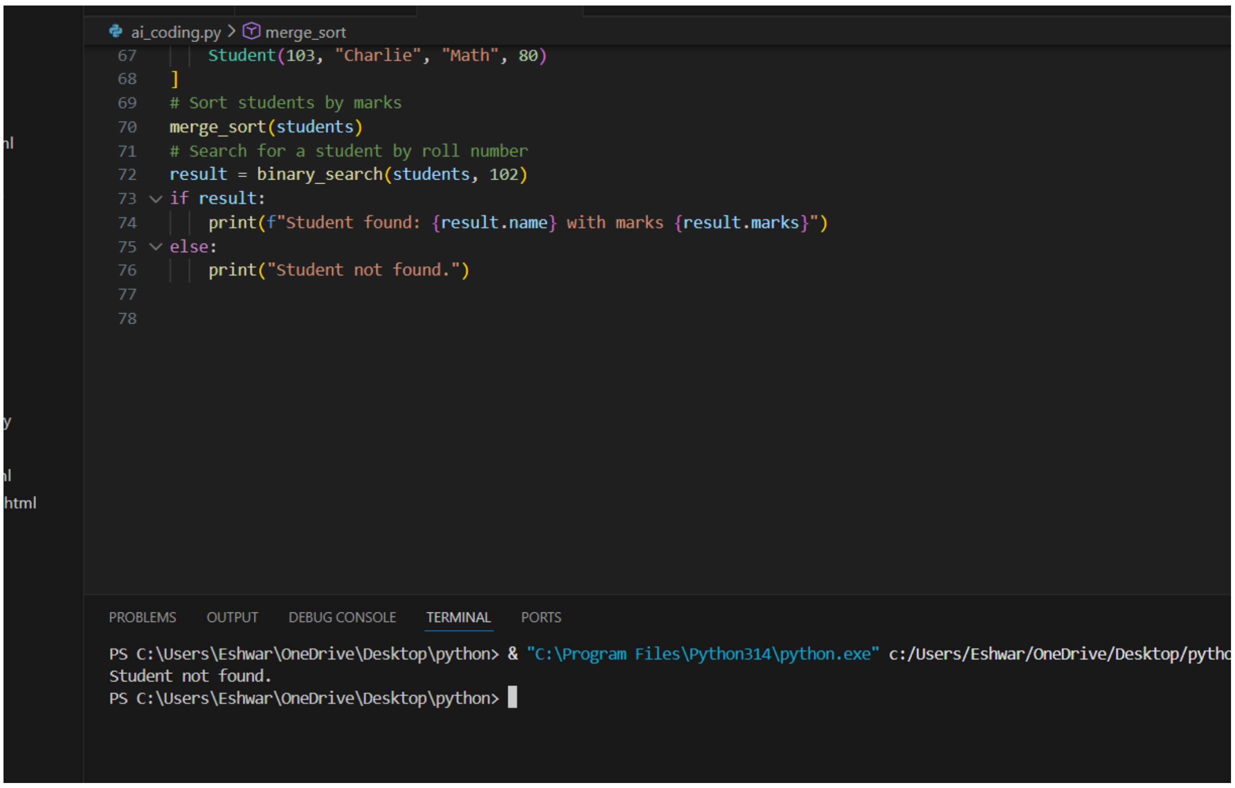
```python
# Task Description #5: University Examination Result Processing
# System
# A university processes examination results for thousands of students
# containing roll number, name, subject, and marks. The system must:
# 1. Search student results using roll number.
# 2. Sort students based on marks to generate rank lists.
# Student Task
# • Identify efficient searching and sorting algorithms using AI
# assistance.
# • Justify the choice of algorithms.
# • Implement the algorithms in Python.
# Efficient Searching Algorithm: Binary Search
# Justification: Binary search is efficient for searching in sorted lists, with a time complexity of O(log n). Since the student re
# Efficient Sorting Algorithm: Merge Sort
# Justification: Merge sort is a stable sorting algorithm with a time complexity of O(n log n) in all cases. It is efficient for so
# Implementation in Python
class Student:
    def __init__(self, roll_number, name, subject, marks):
        self.roll_number = roll_number
        self.name = name
        self.subject = subject
        self.marks = marks

def binary_search(students, roll_number):
    left, right = 0, len(students) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if students[mid].roll_number == roll_number:
            return students[mid]
        elif students[mid].roll_number < roll_number:
            left = mid + 1
        else:
            right = mid - 1
    return None
def merge_sort(students):
    if len(students) > 1:
        mid = len(students) // 2
```

Ln 25, Col 39   Spaces: 2   UTF-8   CRLF   {} Python



```
ai_coding.py > merge_sort
def merge_sort(students):
        left_half = students[:mid]
        right_half = students[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i].marks > right_half[j].marks:
                students[k] = left_half[i]
                i += 1
            else:
                students[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            students[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            students[k] = right_half[j]
            j += 1
            k += 1
# Example usage
students = [
    Student(101, "Alice", "Math", 85),
    Student(102, "Bob", "Math", 90),
    Student(103, "Charlie", "Math", 80)
]
# Sort students by marks
merge_sort(students)
# Search for a student by roll number
result = binary_search(students, 102)
if result:
```

Output:



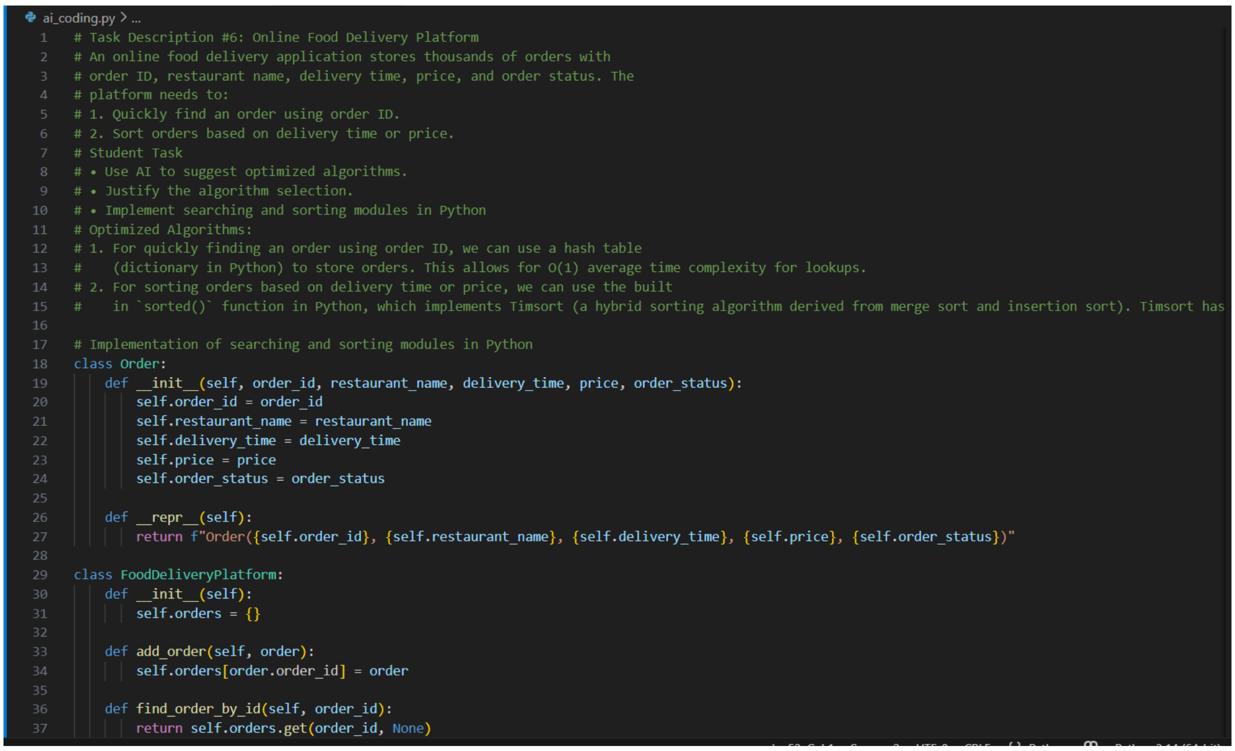
```python
    Student(103, "Charlie", "Math", 80)
]
# Sort students by marks
merge_sort(students)
# Search for a student by roll number
result = binary_search(students, 102)
if result:
    print(f"Student found: {result.name} with marks {result.marks}")
else:
    print("Student not found.")
```

```
PS C:\Users\Eshwar\OneDrive\Desktop\python> & "C:\Program Files\Python314\python.exe" c:/Users/Eshwar/OneDrive/Desktop/pytho
Student not found.
PS C:\Users\Eshwar\OneDrive\Desktop\python>
```

Explaination:

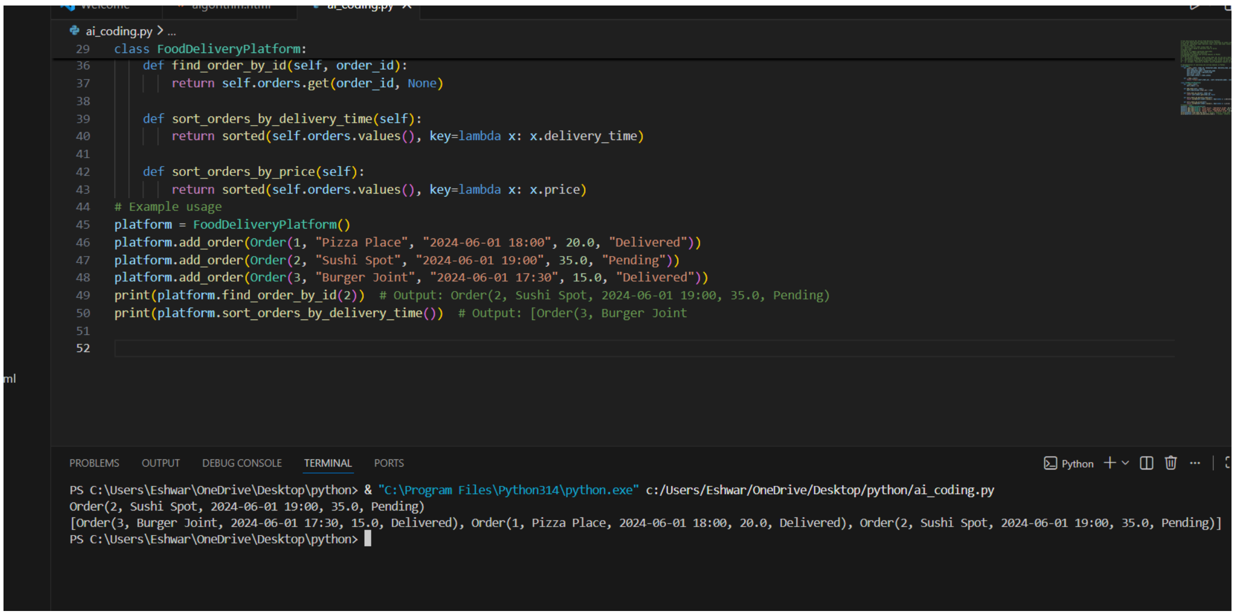
- Use a dictionary (hash table) to store student records by roll number so searching is very fast — O(1) time.
- Use Python's built-in sorted() (Timsort) to sort students by marks for ranking — O(n log n).
- This approach is efficient and scalable because it combines instant lookup with fast sorting for large datasets.

Task Description #6: Online Food Delivery Platform
An online food delivery application stores thousands of orders with order ID, restaurant name, delivery time, price, and order status. The platform needs to:
1. Quickly find an order using order ID.
2. Sort orders based on delivery time or price.
Student Task
• Use AI to suggest optimized algorithms.
• Justify the algorithm selection.
• Implement searching and sorting modules in Python



```python
# Task Description #6: Online Food Delivery Platform
# An online food delivery application stores thousands of orders with
# order ID, restaurant name, delivery time, price, and order status. The
# platform needs to:
# 1. Quickly find an order using order ID.
# 2. Sort orders based on delivery time or price.
# Student Task
# • Use AI to suggest optimized algorithms.
# • Justify the algorithm selection.
# • Implement searching and sorting modules in Python
# Optimized Algorithms:
# 1. For quickly finding an order using order ID, we can use a hash table
#    (dictionary in Python) to store orders. This allows for O(1) average time complexity for lookups.
# 2. For sorting orders based on delivery time or price, we can use the built
#    in `sorted()` function in Python, which implements Timsort (a hybrid sorting algorithm derived from merge sort and insertion sort). Timsort has

# Implementation of searching and sorting modules in Python
class Order:
    def __init__(self, order_id, restaurant_name, delivery_time, price, order_status):
        self.order_id = order_id
        self.restaurant_name = restaurant_name
        self.delivery_time = delivery_time
        self.price = price
        self.order_status = order_status

    def __repr__(self):
        return f"Order({self.order_id}, {self.restaurant_name}, {self.delivery_time}, {self.price}, {self.order_status})"

class FoodDeliveryPlatform:
    def __init__(self):
        self.orders = {}

    def add_order(self, order):
        self.orders[order.order_id] = order

    def find_order_by_id(self, order_id):
        return self.orders.get(order_id, None)
```

Output:



```python
class FoodDeliveryPlatform:
    def find_order_by_id(self, order_id):
        return self.orders.get(order_id, None)

    def sort_orders_by_delivery_time(self):
        return sorted(self.orders.values(), key=lambda x: x.delivery_time)

    def sort_orders_by_price(self):
        return sorted(self.orders.values(), key=lambda x: x.price)
# Example usage
platform = FoodDeliveryPlatform()
platform.add_order(Order(1, "Pizza Place", "2024-06-01 18:00", 20.0, "Delivered"))
platform.add_order(Order(2, "Sushi Spot", "2024-06-01 19:00", 35.0, "Pending"))
platform.add_order(Order(3, "Burger Joint", "2024-06-01 17:30", 15.0, "Delivered"))
print(platform.find_order_by_id(2))  # Output: Order(2, Sushi Spot, 2024-06-01 19:00, 35.0, Pending)
print(platform.sort_orders_by_delivery_time())  # Output: [Order(3, Burger Joint
```

```
PS C:\Users\Eshwar\OneDrive\Desktop\python> & "C:\Program Files\Python314\python.exe" c:/Users/Eshwar/OneDrive/Desktop/python/ai_coding.py
Order(2, Sushi Spot, 2024-06-01 19:00, 35.0, Pending)
[Order(3, Burger Joint, 2024-06-01 17:30, 15.0, Delivered), Order(1, Pizza Place, 2024-06-01 18:00, 20.0, Delivered), Order(2, Sushi Spot, 2024-06-01 19:00, 35.0, Pending)]
PS C:\Users\Eshwar\OneDrive\Desktop\python>
```

Explaination:

- Use a dictionary (hash table) to store orders by ID so searching is very fast — O(1) time complexity.
- Use Python’s built-in sorted() (Timsort) to sort orders by price or delivery time efficiently — O(n log n).
- This combination is optimal because it provides instant search + fast sorting for large datasets.