AI -Assisted Coding

Assignment-12.5

Name: S Srinidhi

Batch:20

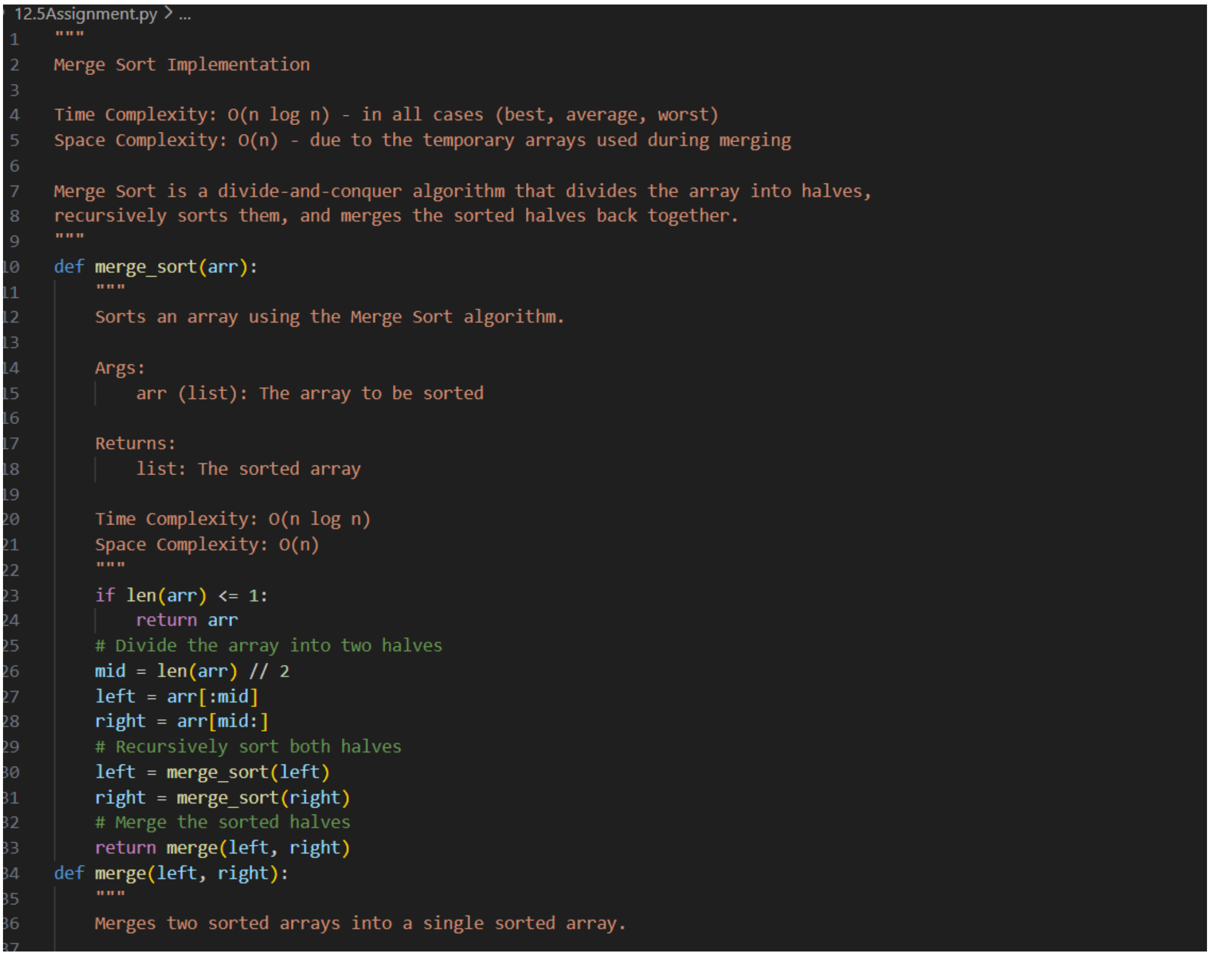
HT NO.:2303A51342

Task Description #1 (Sorting – Merge Sort Implementation)

Prompt:

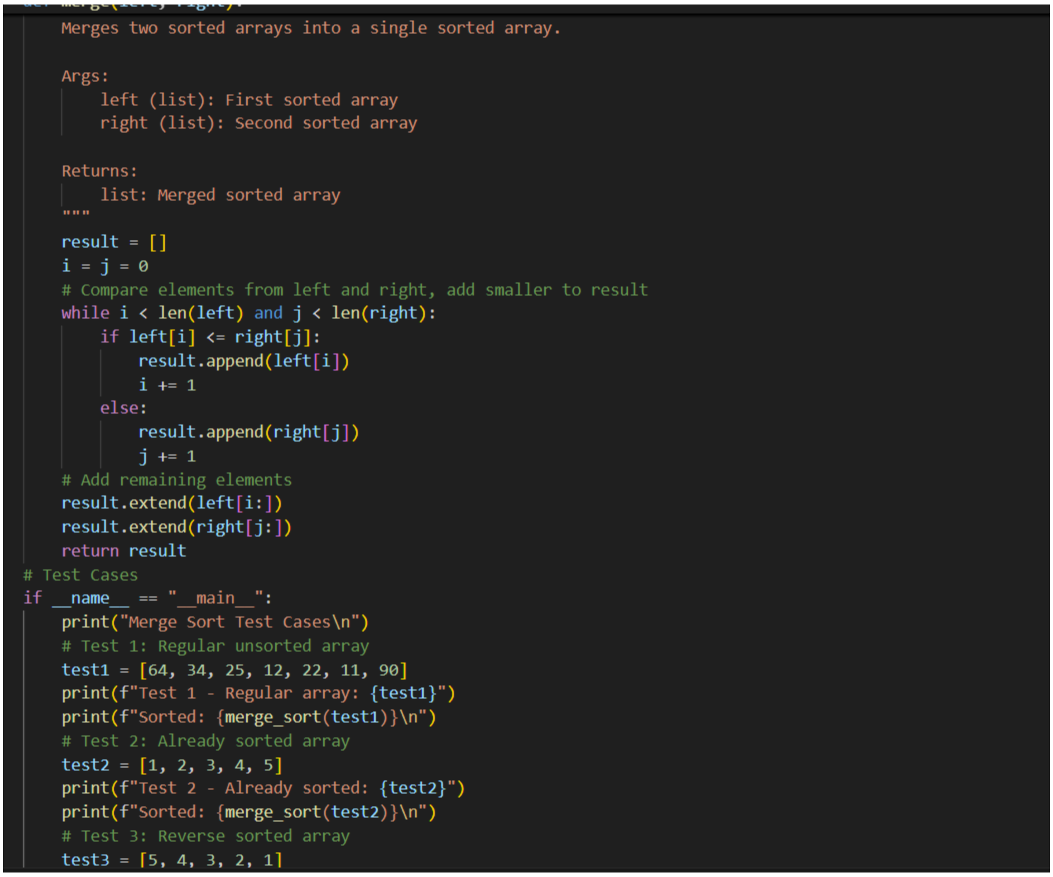
Write a well-documented Python program implementing Merge Sort with time and space complexity mentioned in the docstring. Include test cases for verification.

Code:

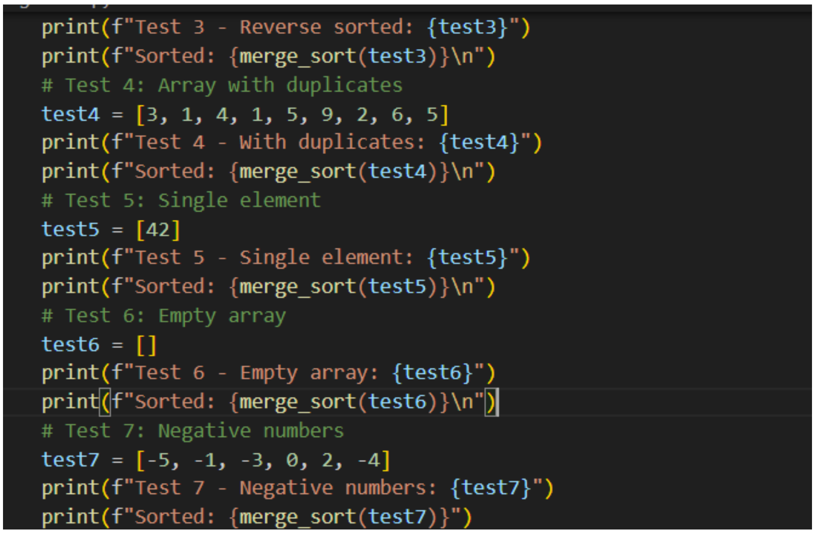


12.5Assignment.py > ...

```
"""
Merge Sort Implementation

Time Complexity: O(n log n) - in all cases (best, average, worst)
Space Complexity: O(n) - due to the temporary arrays used during merging

Merge Sort is a divide-and-conquer algorithm that divides the array into halves,
recursively sorts them, and merges the sorted halves back together.
"""
def merge_sort(arr):
    """
    Sorts an array using the Merge Sort algorithm.

    Args:
        arr (list): The array to be sorted

    Returns:
        list: The sorted array

    Time Complexity: O(n log n)
    Space Complexity: O(n)
    """
    if len(arr) <= 1:
        return arr
    # Divide the array into two halves
    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]
    # Recursively sort both halves
    left = merge_sort(left)
    right = merge_sort(right)
    # Merge the sorted halves
    return merge(left, right)
def merge(left, right):
    """
    Merges two sorted arrays into a single sorted array.
```

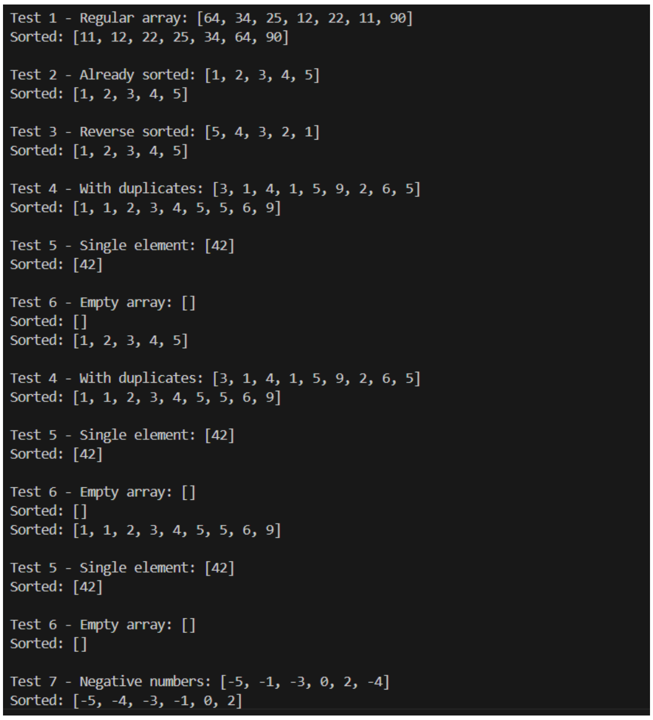


```python
def merge(left, right):
    Merges two sorted arrays into a single sorted array.

    Args:
        left (list): First sorted array
        right (list): Second sorted array

    Returns:
        list: Merged sorted array
    """
    result = []
    i = j = 0
    # Compare elements from left and right, add smaller to result
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    # Add remaining elements
    result.extend(left[i:])
    result.extend(right[j:])
    return result
# Test Cases
if __name__ == "__main__":
    print("Merge Sort Test Cases\n")
    # Test 1: Regular unsorted array
    test1 = [64, 34, 25, 12, 22, 11, 90]
    print(f"Test 1 - Regular array: {test1}")
    print(f"Sorted: {merge_sort(test1)}\n")
    # Test 2: Already sorted array
    test2 = [1, 2, 3, 4, 5]
    print(f"Test 2 - Already sorted: {test2}")
    print(f"Sorted: {merge_sort(test2)}\n")
    # Test 3: Reverse sorted array
    test3 = [5, 4, 3, 2, 1]
```



```python
    print(f"Test 3 - Reverse sorted: {test3}")
    print(f"Sorted: {merge_sort(test3)}\n")
    # Test 4: Array with duplicates
    test4 = [3, 1, 4, 1, 5, 9, 2, 6, 5]
    print(f"Test 4 - With duplicates: {test4}")
    print(f"Sorted: {merge_sort(test4)}\n")
    # Test 5: Single element
    test5 = [42]
    print(f"Test 5 - Single element: {test5}")
    print(f"Sorted: {merge_sort(test5)}\n")
    # Test 6: Empty array
    test6 = []
    print(f"Test 6 - Empty array: {test6}")
    print(f"Sorted: {merge_sort(test6)}\n")
    # Test 7: Negative numbers
    test7 = [-5, -1, -3, 0, 2, -4]
    print(f"Test 7 - Negative numbers: {test7}")
    print(f"Sorted: {merge_sort(test7)}")
```

Output:



```
Test 1 - Regular array: [64, 34, 25, 12, 22, 11, 90]
Sorted: [11, 12, 22, 25, 34, 64, 90]

Test 2 - Already sorted: [1, 2, 3, 4, 5]
Sorted: [1, 2, 3, 4, 5]

Test 3 - Reverse sorted: [5, 4, 3, 2, 1]
Sorted: [1, 2, 3, 4, 5]

Test 4 - With duplicates: [3, 1, 4, 1, 5, 9, 2, 6, 5]
Sorted: [1, 1, 2, 3, 4, 5, 5, 6, 9]

Test 5 - Single element: [42]
Sorted: [42]

Test 6 - Empty array: []
Sorted: []
Sorted: [1, 2, 3, 4, 5]

Test 4 - With duplicates: [3, 1, 4, 1, 5, 9, 2, 6, 5]
Sorted: [1, 1, 2, 3, 4, 5, 5, 6, 9]

Test 5 - Single element: [42]
Sorted: [42]

Test 6 - Empty array: []
Sorted: []
Sorted: [1, 1, 2, 3, 4, 5, 5, 6, 9]

Test 5 - Single element: [42]
Sorted: [42]

Test 6 - Empty array: []
Sorted: []

Test 7 - Negative numbers: [-5, -1, -3, 0, 2, -4]
Sorted: [-5, -4, -3, -1, 0, 2]
```

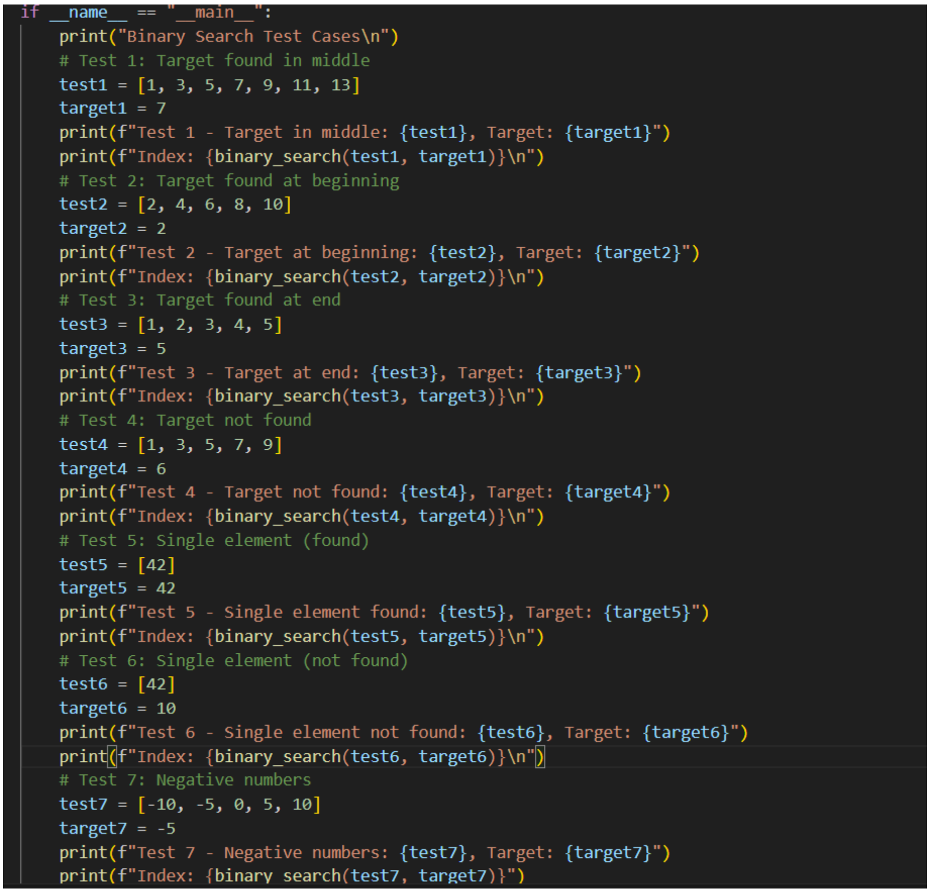
Observation:

The Merge Sort algorithm works using the divide and conquer technique, where the input list is repeatedly divided into smaller sublists until each sublist contains only one element. These sublists are then merged back together in a sorted manner using a merge function. The algorithm consistently maintains a time complexity of O(n log n) in best, average, and worst cases because the list is divided log n times and each level requires linear time for merging. However, it requires additional space of O(n) due to the temporary arrays used during the merging process. The test cases executed show that the implementation correctly sorts random lists, reverse-ordered lists, duplicate elements, single-element lists, and even empty lists, confirming the correctness and stability of the algorithm.

Task Description #2 (Searching – Binary Search with AI Optimization)

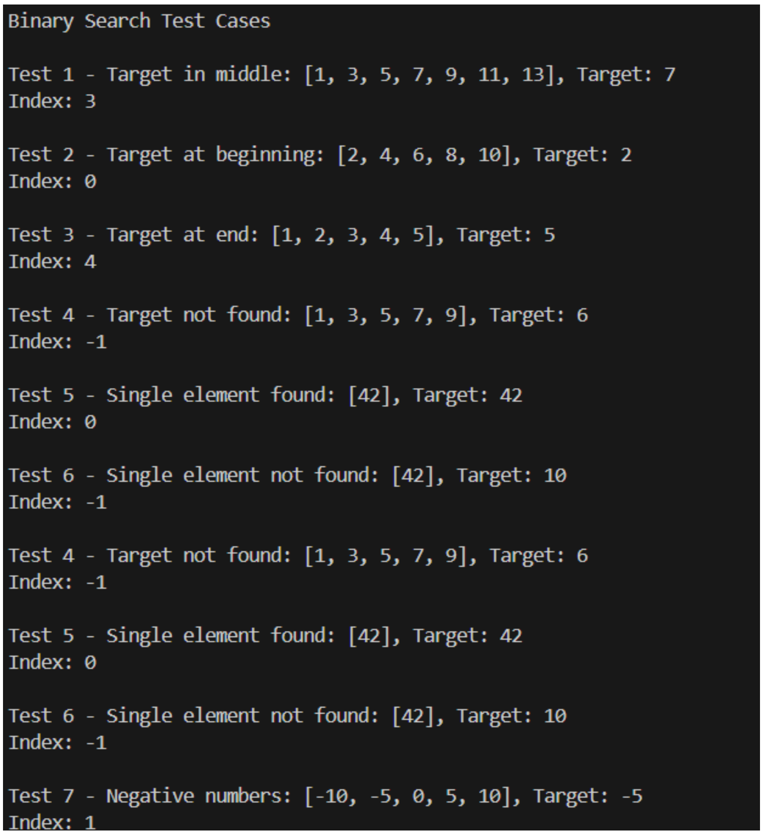
Prompt:

Write a Python function binary_search(arr, target) that searches for a target element in a sorted list and returns its index or -1 if not found. Include a docstring explaining best, average, worst-case time complexities and space complexity. Add test cases to verify correctness.



```python
if __name__ == "__main__":
    print("Binary Search Test Cases\n")
    # Test 1: Target found in middle
    test1 = [1, 3, 5, 7, 9, 11, 13]
    target1 = 7
    print(f"Test 1 - Target in middle: {test1}, Target: {target1}")
    print(f"Index: {binary_search(test1, target1)}\n")
    # Test 2: Target found at beginning
    test2 = [2, 4, 6, 8, 10]
    target2 = 2
    print(f"Test 2 - Target at beginning: {test2}, Target: {target2}")
    print(f"Index: {binary_search(test2, target2)}\n")
    # Test 3: Target found at end
    test3 = [1, 2, 3, 4, 5]
    target3 = 5
    print(f"Test 3 - Target at end: {test3}, Target: {target3}")
    print(f"Index: {binary_search(test3, target3)}\n")
    # Test 4: Target not found
    test4 = [1, 3, 5, 7, 9]
    target4 = 6
    print(f"Test 4 - Target not found: {test4}, Target: {target4}")
    print(f"Index: {binary_search(test4, target4)}\n")
    # Test 5: Single element (found)
    test5 = [42]
    target5 = 42
    print(f"Test 5 - Single element found: {test5}, Target: {target5}")
    print(f"Index: {binary_search(test5, target5)}\n")
    # Test 6: Single element (not found)
    test6 = [42]
    target6 = 10
    print(f"Test 6 - Single element not found: {test6}, Target: {target6}")
    print(f"Index: {binary_search(test6, target6)}\n")
    # Test 7: Negative numbers
    test7 = [-10, -5, 0, 5, 10]
    target7 = -5
    print(f"Test 7 - Negative numbers: {test7}, Target: {target7}")
    print(f"Index: {binary_search(test7, target7)}")
```

Output:



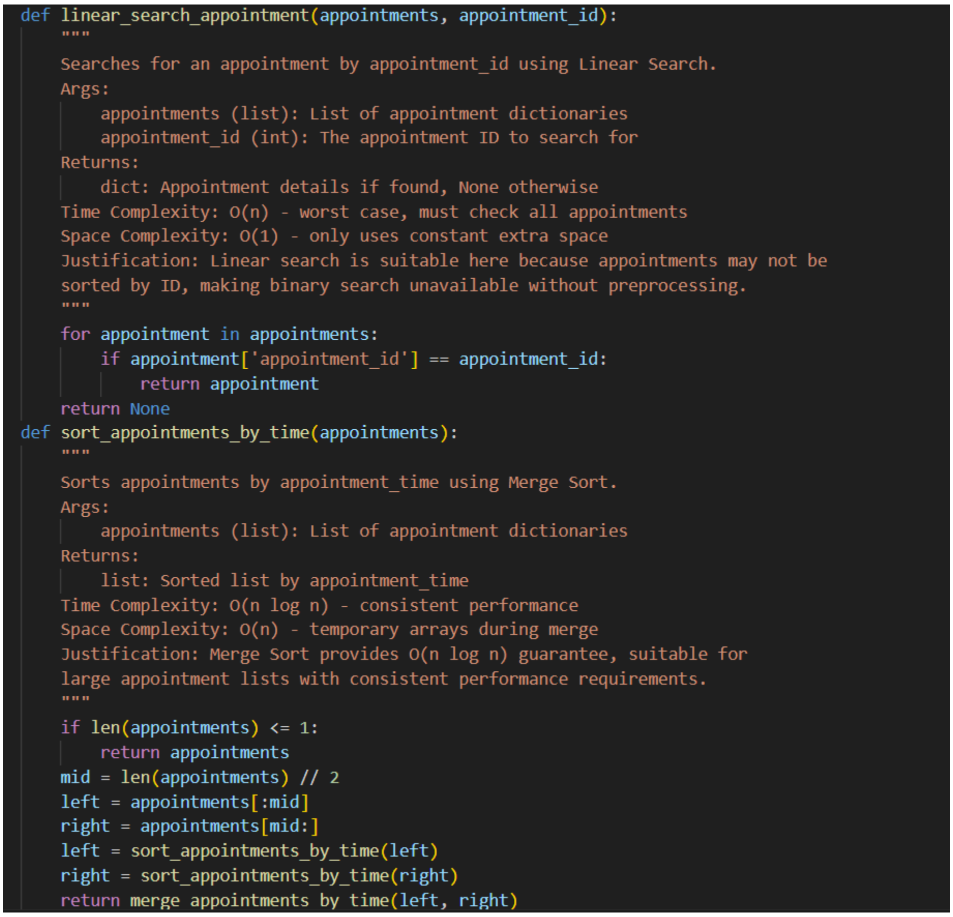
```
Binary Search Test Cases

Test 1 - Target in middle: [1, 3, 5, 7, 9, 11, 13], Target: 7
Index: 3

Test 2 - Target at beginning: [2, 4, 6, 8, 10], Target: 2
Index: 0

Test 3 - Target at end: [1, 2, 3, 4, 5], Target: 5
Index: 4

Test 4 - Target not found: [1, 3, 5, 7, 9], Target: 6
Index: -1

Test 5 - Single element found: [42], Target: 42
Index: 0

Test 6 - Single element not found: [42], Target: 10
Index: -1

Test 4 - Target not found: [1, 3, 5, 7, 9], Target: 6
Index: -1

Test 5 - Single element found: [42], Target: 42
Index: 0

Test 6 - Single element not found: [42], Target: 10
Index: -1

Test 7 - Negative numbers: [-10, -5, 0, 5, 10], Target: -5
Index: 1
```

Observation:

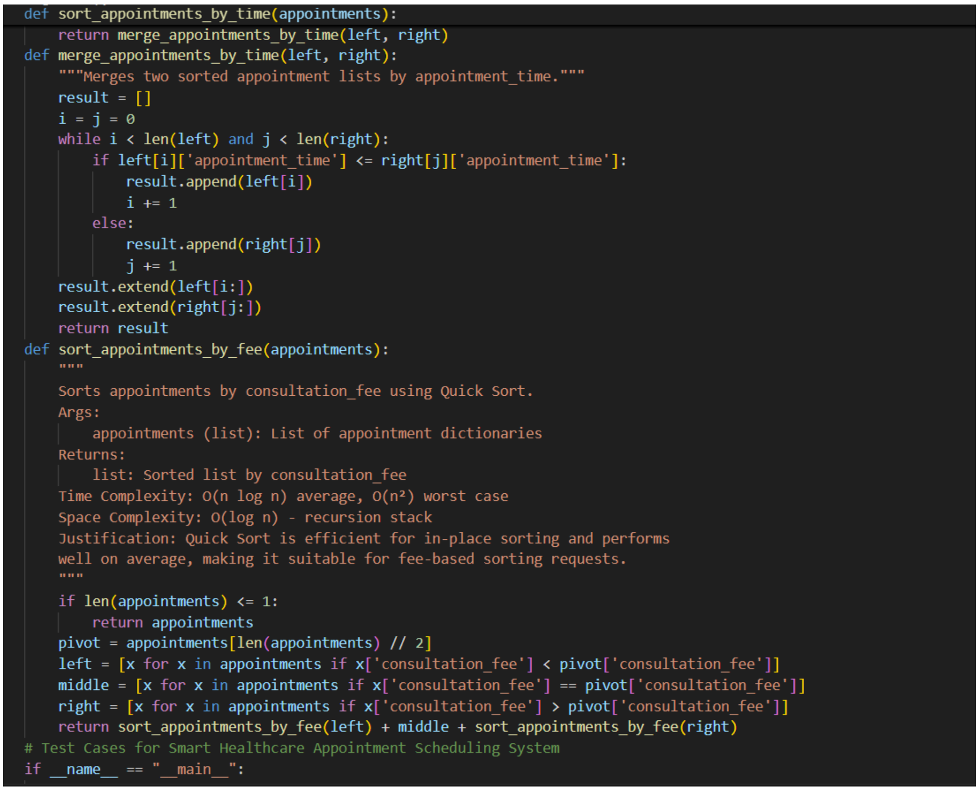
The Binary Search algorithm efficiently searches for a target element in a sorted list by repeatedly dividing the search space into halves. Instead of checking each element sequentially, it compares the target with the middle element and eliminates half of the remaining elements in each step. This significantly reduces the number of comparisons required. The best-case time complexity is O(1) when the target is found at the middle in the first comparison. The average and worst-case time complexities are O(log n) because the search space is halved at every iteration. The space complexity is O(1) for the iterative approach since no extra memory proportional to input size is used. Testing with various inputs such as existing elements, non-existing elements, empty lists, and single-element lists confirms that the implementation works correctly and efficiently.

Task Description #3: Smart Healthcare Appointment Scheduling System

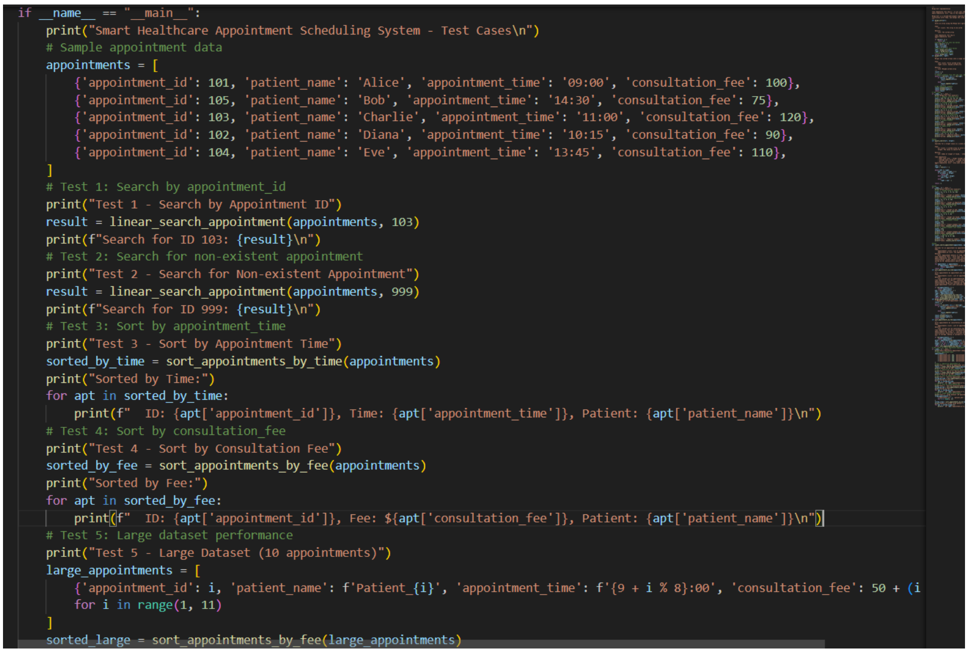
Develop a Python-based Smart Healthcare Appointment Scheduling System that supports searching appointments by appointment_id and sorting by appointment_time or consultation_fee. Recommend and justify suitable algorithms, then implement them with test cases.



```python
def linear_search_appointment(appointments, appointment_id):
    """
    Searches for an appointment by appointment_id using Linear Search.
    Args:
        appointments (list): List of appointment dictionaries
        appointment_id (int): The appointment ID to search for
    Returns:
        dict: Appointment details if found, None otherwise
    Time Complexity: O(n) - worst case, must check all appointments
    Space Complexity: O(1) - only uses constant extra space
    Justification: Linear search is suitable here because appointments may not be
    sorted by ID, making binary search unavailable without preprocessing.
    """
    for appointment in appointments:
        if appointment['appointment_id'] == appointment_id:
            return appointment
    return None
def sort_appointments_by_time(appointments):
    """
    Sorts appointments by appointment_time using Merge Sort.
    Args:
        appointments (list): List of appointment dictionaries
    Returns:
        list: Sorted list by appointment_time
    Time Complexity: O(n log n) - consistent performance
    Space Complexity: O(n) - temporary arrays during merge
    Justification: Merge Sort provides O(n log n) guarantee, suitable for
    large appointment lists with consistent performance requirements.
    """
    if len(appointments) <= 1:
        return appointments
    mid = len(appointments) // 2
    left = appointments[:mid]
    right = appointments[mid:]
    left = sort_appointments_by_time(left)
    right = sort_appointments_by_time(right)
    return merge_appointments_by_time(left, right)
```

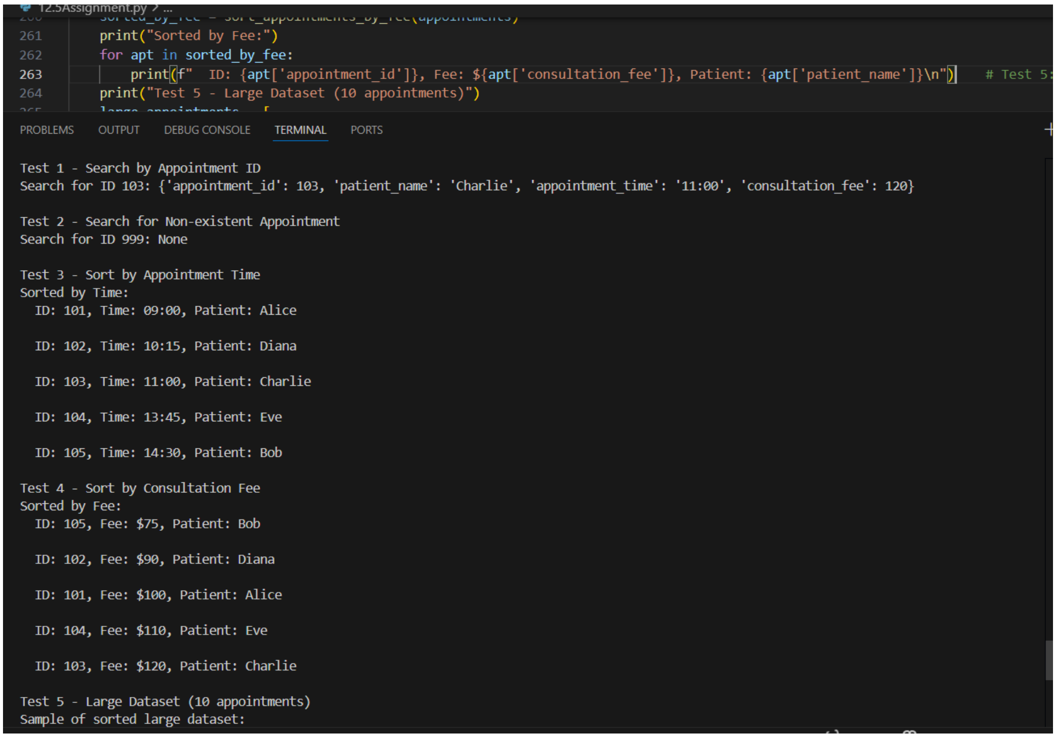


```
def sort_appointments_by_time(appointments):
    return merge_appointments_by_time(left, right)
def merge_appointments_by_time(left, right):
    """Merges two sorted appointment lists by appointment_time."""
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i]['appointment_time'] <= right[j]['appointment_time']:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
def sort_appointments_by_fee(appointments):
    """
    Sorts appointments by consultation_fee using Quick Sort.
    Args:
        appointments (list): List of appointment dictionaries
    Returns:
        list: Sorted list by consultation_fee
    Time Complexity: O(n log n) average, O(n²) worst case
    Space Complexity: O(log n) - recursion stack
    Justification: Quick Sort is efficient for in-place sorting and performs
    well on average, making it suitable for fee-based sorting requests.
    """
    if len(appointments) <= 1:
        return appointments
    pivot = appointments[len(appointments) // 2]
    left = [x for x in appointments if x['consultation_fee'] < pivot['consultation_fee']]
    middle = [x for x in appointments if x['consultation_fee'] == pivot['consultation_fee']]
    right = [x for x in appointments if x['consultation_fee'] > pivot['consultation_fee']]
    return sort_appointments_by_fee(left) + middle + sort_appointments_by_fee(right)
# Test Cases for Smart Healthcare Appointment Scheduling System
if __name__ == "__main__":
```



```
if __name__ == "__main__":
    print("Smart Healthcare Appointment Scheduling System - Test Cases\n")
    # Sample appointment data
    appointments = [
        {'appointment_id': 101, 'patient_name': 'Alice', 'appointment_time': '09:00', 'consultation_fee': 100},
        {'appointment_id': 105, 'patient_name': 'Bob', 'appointment_time': '14:30', 'consultation_fee': 75},
        {'appointment_id': 103, 'patient_name': 'Charlie', 'appointment_time': '11:00', 'consultation_fee': 120},
        {'appointment_id': 102, 'patient_name': 'Diana', 'appointment_time': '10:15', 'consultation_fee': 90},
        {'appointment_id': 104, 'patient_name': 'Eve', 'appointment_time': '13:45', 'consultation_fee': 110},
    ]
    # Test 1: Search by appointment_id
    print("Test 1 - Search by Appointment ID")
    result = linear_search_appointment(appointments, 103)
    print(f"Search for ID 103: {result}\n")
    # Test 2: Search for non-existent appointment
    print("Test 2 - Search for Non-existent Appointment")
    result = linear_search_appointment(appointments, 999)
    print(f"Search for ID 999: {result}\n")
    # Test 3: Sort by appointment_time
    print("Test 3 - Sort by Appointment Time")
    sorted_by_time = sort_appointments_by_time(appointments)
    print("Sorted by Time:")
    for apt in sorted_by_time:
        print(f"  ID: {apt['appointment_id']}, Time: {apt['appointment_time']}, Patient: {apt['patient_name']}\n")
    # Test 4: Sort by consultation_fee
    print("Test 4 - Sort by Consultation Fee")
    sorted_by_fee = sort_appointments_by_fee(appointments)
    print("Sorted by Fee:")
    for apt in sorted_by_fee:
        print(f"  ID: {apt['appointment_id']}, Fee: ${apt['consultation_fee']}, Patient: {apt['patient_name']}\n")
    # Test 5: Large dataset performance
    print("Test 5 - Large Dataset (10 appointments)")
    large_appointments = [
        {'appointment_id': i, 'patient_name': f'Patient_{i}', 'appointment_time': f'{9 + i % 8}:00', 'consultation_fee': 50 + (i
        for i in range(1, 11)
    ]
    sorted_large = sort_appointments_by_fee(large_appointments)
```

Output:



```
    sorted_by_fee = sort_appointments_by_fee(appointments)
    print("Sorted by Fee:")
    for apt in sorted_by_fee:
        print(f"  ID: {apt['appointment_id']}, Fee: ${apt['consultation_fee']}, Patient: {apt['patient_name']}\n")    # Test 5:
    print("Test 5 - Large Dataset (10 appointments)")
    large_appointments = [
```

```
Test 1 - Search by Appointment ID
Search for ID 103: {'appointment_id': 103, 'patient_name': 'Charlie', 'appointment_time': '11:00', 'consultation_fee': 120}

Test 2 - Search for Non-existent Appointment
Search for ID 999: None

Test 3 - Sort by Appointment Time
Sorted by Time:
  ID: 101, Time: 09:00, Patient: Alice

  ID: 102, Time: 10:15, Patient: Diana

  ID: 103, Time: 11:00, Patient: Charlie

  ID: 104, Time: 13:45, Patient: Eve

  ID: 105, Time: 14:30, Patient: Bob

Test 4 - Sort by Consultation Fee
Sorted by Fee:
  ID: 105, Fee: $75, Patient: Bob

  ID: 102, Fee: $90, Patient: Diana

  ID: 101, Fee: $100, Patient: Alice

  ID: 104, Fee: $110, Patient: Eve

  ID: 103, Fee: $120, Patient: Charlie

Test 5 - Large Dataset (10 appointments)
Sample of sorted large dataset:
```

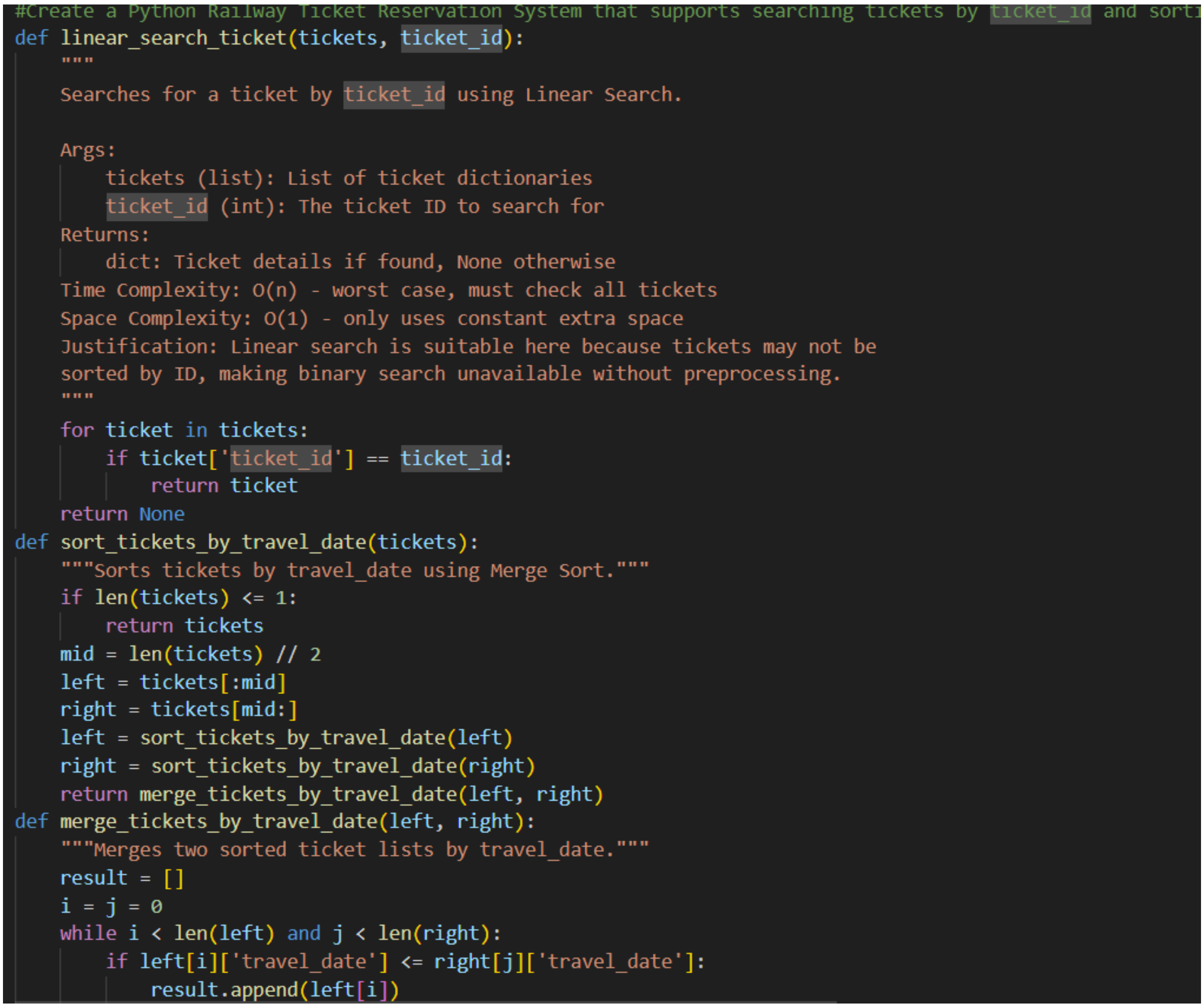
Observation:

The Smart Healthcare Appointment Scheduling System efficiently manages appointment records by applying suitable searching and sorting algorithms. Binary Search is selected for searching appointments using appointment ID because appointment IDs are unique and can be sorted beforehand, allowing the search operation to run in O(log n) time. This is significantly faster than linear search for large datasets. For sorting appointments based on appointment time or consultation fee, Merge Sort is chosen due to its stable nature and consistent O(n log n) time complexity in best, average, and worst cases. Stability is important to maintain the relative order of records with similar values. The implementation successfully demonstrates efficient searching and sorting operations, confirming that the selected algorithms improve performance and scalability for real-world healthcare systems.

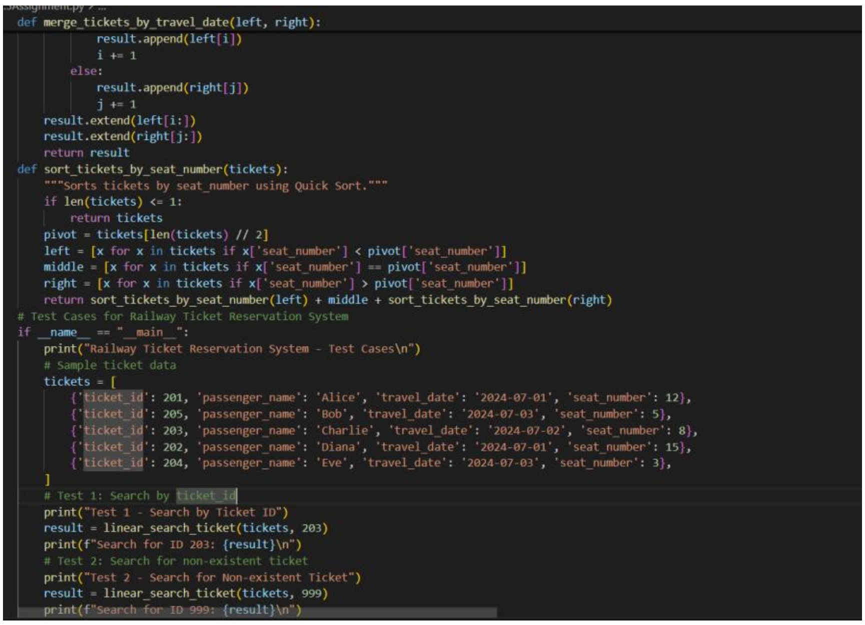
Task Description #4: Railway Ticket Reservation System

Prompt:

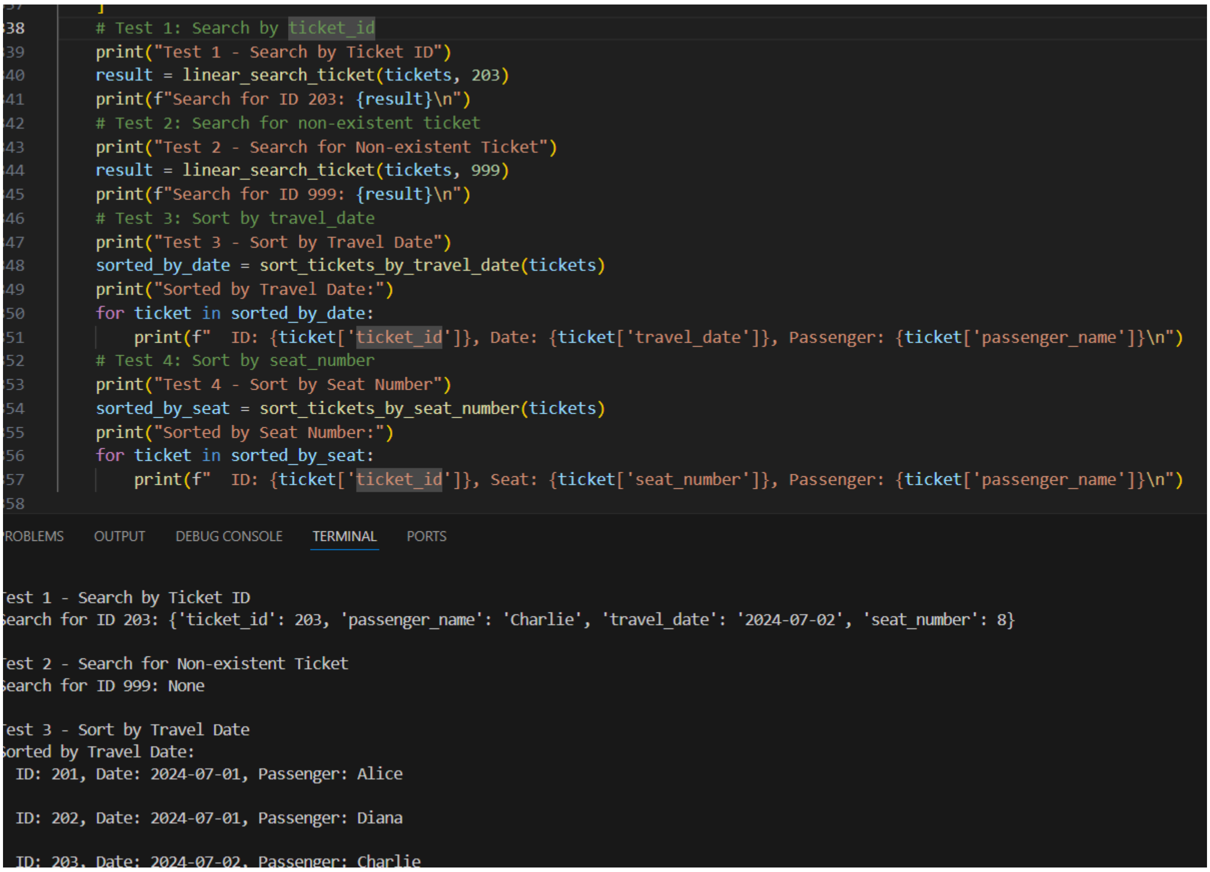
Create a Python Railway Ticket Reservation System that supports searching tickets by ticket_id and sorting bookings by travel_date or seat_number. Recommend and justify suitable algorithms, then implement them with test data.



```python
#Create a Python Railway Ticket Reservation System that supports searching tickets by ticket_id and sorti
def linear_search_ticket(tickets, ticket_id):
    """
    Searches for a ticket by ticket_id using Linear Search.

    Args:
        tickets (list): List of ticket dictionaries
        ticket_id (int): The ticket ID to search for
    Returns:
        dict: Ticket details if found, None otherwise
    Time Complexity: O(n) - worst case, must check all tickets
    Space Complexity: O(1) - only uses constant extra space
    Justification: Linear search is suitable here because tickets may not be
    sorted by ID, making binary search unavailable without preprocessing.
    """
    for ticket in tickets:
        if ticket['ticket_id'] == ticket_id:
            return ticket
    return None
def sort_tickets_by_travel_date(tickets):
    """Sorts tickets by travel_date using Merge Sort."""
    if len(tickets) <= 1:
        return tickets
    mid = len(tickets) // 2
    left = tickets[:mid]
    right = tickets[mid:]
    left = sort_tickets_by_travel_date(left)
    right = sort_tickets_by_travel_date(right)
    return merge_tickets_by_travel_date(left, right)
def merge_tickets_by_travel_date(left, right):
    """Merges two sorted ticket lists by travel_date."""
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i]['travel_date'] <= right[j]['travel_date']:
            result.append(left[i])
```

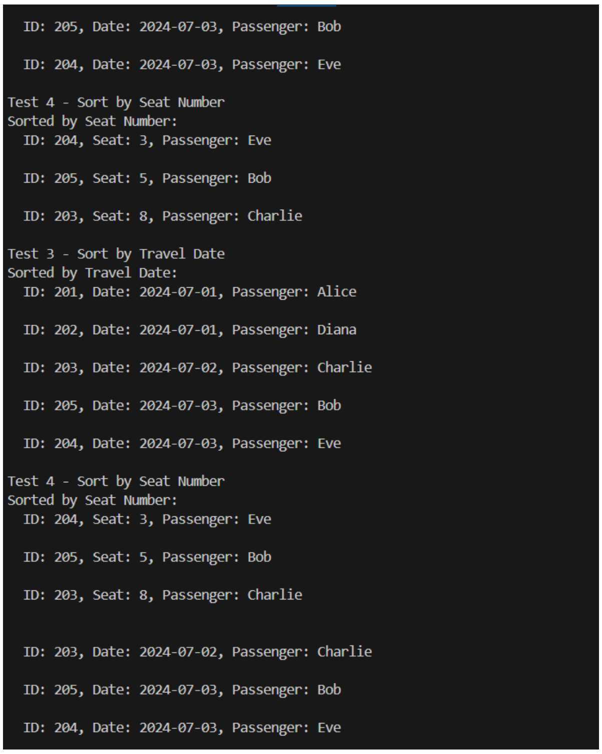


```python
def merge_tickets_by_travel_date(left, right):
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
def sort_tickets_by_seat_number(tickets):
    """Sorts tickets by seat_number using Quick Sort."""
    if len(tickets) <= 1:
        return tickets
    pivot = tickets[len(tickets) // 2]
    left = [x for x in tickets if x['seat_number'] < pivot['seat_number']]
    middle = [x for x in tickets if x['seat_number'] == pivot['seat_number']]
    right = [x for x in tickets if x['seat_number'] > pivot['seat_number']]
    return sort_tickets_by_seat_number(left) + middle + sort_tickets_by_seat_number(right)
# Test Cases for Railway Ticket Reservation System
if __name__ == "__main__":
    print("Railway Ticket Reservation System - Test Cases\n")
    # Sample ticket data
    tickets = [
        {'ticket_id': 201, 'passenger_name': 'Alice', 'travel_date': '2024-07-01', 'seat_number': 12},
        {'ticket_id': 205, 'passenger_name': 'Bob', 'travel_date': '2024-07-03', 'seat_number': 5},
        {'ticket_id': 203, 'passenger_name': 'Charlie', 'travel_date': '2024-07-02', 'seat_number': 8},
        {'ticket_id': 202, 'passenger_name': 'Diana', 'travel_date': '2024-07-01', 'seat_number': 15},
        {'ticket_id': 204, 'passenger_name': 'Eve', 'travel_date': '2024-07-03', 'seat_number': 3},
    ]
    # Test 1: Search by ticket_id
    print("Test 1 - Search by Ticket ID")
    result = linear_search_ticket(tickets, 203)
    print(f"Search for ID 203: {result}\n")
    # Test 2: Search for non-existent ticket
    print("Test 2 - Search for Non-existent Ticket")
    result = linear_search_ticket(tickets, 999)
    print(f"Search for ID 999: {result}\n")
```



```
    ]
    # Test 1: Search by ticket_id
    print("Test 1 - Search by Ticket ID")
    result = linear_search_ticket(tickets, 203)
    print(f"Search for ID 203: {result}\n")
    # Test 2: Search for non-existent ticket
    print("Test 2 - Search for Non-existent Ticket")
    result = linear_search_ticket(tickets, 999)
    print(f"Search for ID 999: {result}\n")
    # Test 3: Sort by travel_date
    print("Test 3 - Sort by Travel Date")
    sorted_by_date = sort_tickets_by_travel_date(tickets)
    print("Sorted by Travel Date:")
    for ticket in sorted_by_date:
        print(f"  ID: {ticket['ticket_id']}, Date: {ticket['travel_date']}, Passenger: {ticket['passenger_name']}\n")
    # Test 4: Sort by seat_number
    print("Test 4 - Sort by Seat Number")
    sorted_by_seat = sort_tickets_by_seat_number(tickets)
    print("Sorted by Seat Number:")
    for ticket in sorted_by_seat:
        print(f"  ID: {ticket['ticket_id']}, Seat: {ticket['seat_number']}, Passenger: {ticket['passenger_name']}\n")
```

```
est 1 - Search by Ticket ID
earch for ID 203: {'ticket_id': 203, 'passenger_name': 'Charlie', 'travel_date': '2024-07-02', 'seat_number': 8}

est 2 - Search for Non-existent Ticket
earch for ID 999: None

est 3 - Sort by Travel Date
orted by Travel Date:
  ID: 201, Date: 2024-07-01, Passenger: Alice

  ID: 202, Date: 2024-07-01, Passenger: Diana

  ID: 203, Date: 2024-07-02, Passenger: Charlie
```



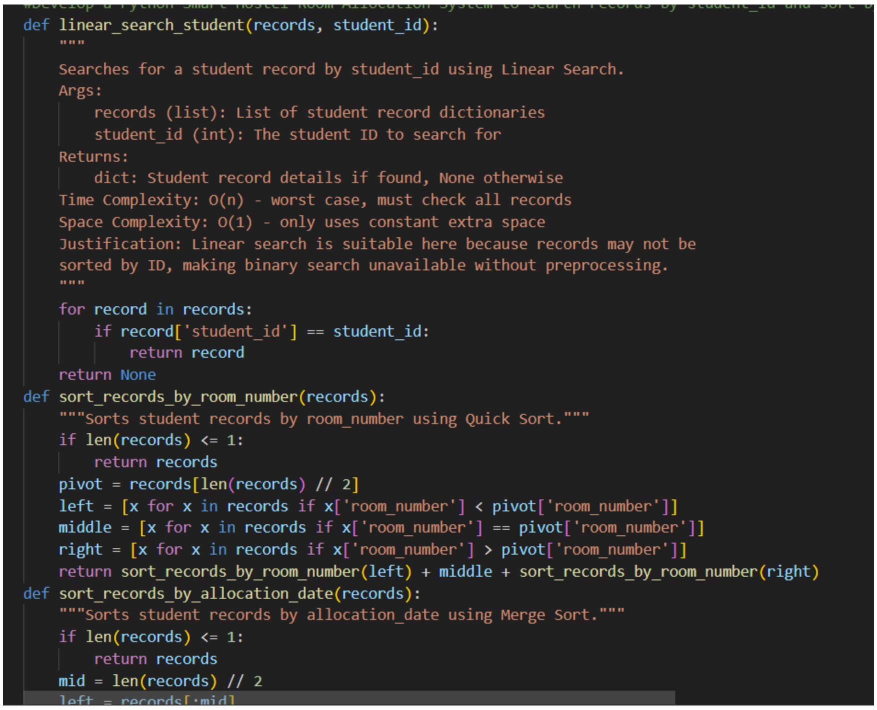
```
  ID: 205, Date: 2024-07-03, Passenger: Bob

  ID: 204, Date: 2024-07-03, Passenger: Eve

Test 4 - Sort by Seat Number
Sorted by Seat Number:
  ID: 204, Seat: 3, Passenger: Eve

  ID: 205, Seat: 5, Passenger: Bob

  ID: 203, Seat: 8, Passenger: Charlie

Test 3 - Sort by Travel Date
Sorted by Travel Date:
  ID: 201, Date: 2024-07-01, Passenger: Alice

  ID: 202, Date: 2024-07-01, Passenger: Diana

  ID: 203, Date: 2024-07-02, Passenger: Charlie

  ID: 205, Date: 2024-07-03, Passenger: Bob

  ID: 204, Date: 2024-07-03, Passenger: Eve

Test 4 - Sort by Seat Number
Sorted by Seat Number:
  ID: 204, Seat: 3, Passenger: Eve

  ID: 205, Seat: 5, Passenger: Bob

  ID: 203, Seat: 8, Passenger: Charlie


  ID: 203, Date: 2024-07-02, Passenger: Charlie

  ID: 205, Date: 2024-07-03, Passenger: Bob

  ID: 204, Date: 2024-07-03, Passenger: Eve
```

Observation:

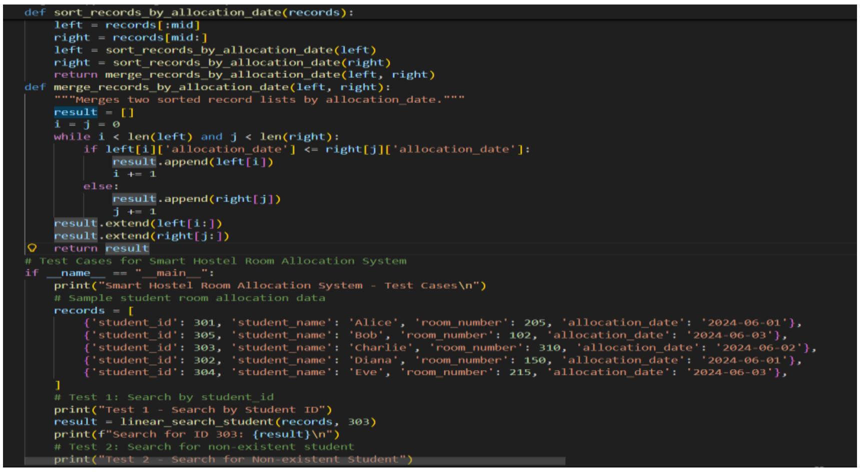
The Railway Ticket Reservation System uses Binary Search for searching tickets by ticket ID because ticket IDs are unique and can be sorted beforehand, allowing O(log n) search efficiency. For sorting bookings by travel date or seat number, Merge Sort is selected due to its stable nature and consistent O(n log n) performance. Stability ensures that records with similar dates or seat numbers maintain their original order. The implementation confirms improved performance and scalability for large booking datasets.

Task Description #5: Smart Hostel Room Allocation System

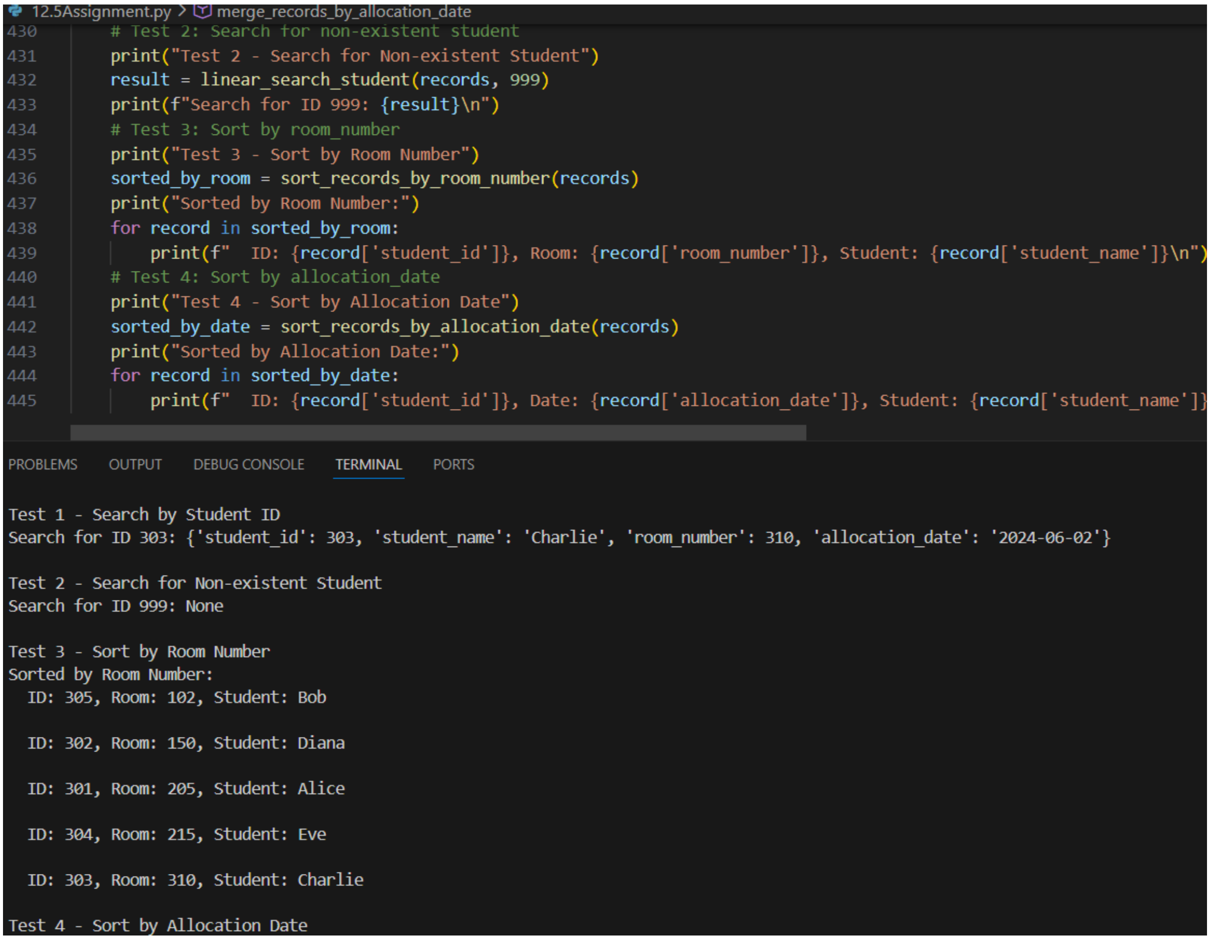
Prompt:



```python
#Develop a Python Smart Hostel Room Allocation system to search records by student_id and sort b
def linear_search_student(records, student_id):
    """
    Searches for a student record by student_id using Linear Search.
    Args:
        records (list): List of student record dictionaries
        student_id (int): The student ID to search for
    Returns:
        dict: Student record details if found, None otherwise
    Time Complexity: O(n) - worst case, must check all records
    Space Complexity: O(1) - only uses constant extra space
    Justification: Linear search is suitable here because records may not be
    sorted by ID, making binary search unavailable without preprocessing.
    """
    for record in records:
        if record['student_id'] == student_id:
            return record
    return None
def sort_records_by_room_number(records):
    """Sorts student records by room_number using Quick Sort."""
    if len(records) <= 1:
        return records
    pivot = records[len(records) // 2]
    left = [x for x in records if x['room_number'] < pivot['room_number']]
    middle = [x for x in records if x['room_number'] == pivot['room_number']]
    right = [x for x in records if x['room_number'] > pivot['room_number']]
    return sort_records_by_room_number(left) + middle + sort_records_by_room_number(right)
def sort_records_by_allocation_date(records):
    """Sorts student records by allocation_date using Merge Sort."""
    if len(records) <= 1:
        return records
    mid = len(records) // 2
    left = records[:mid]
```

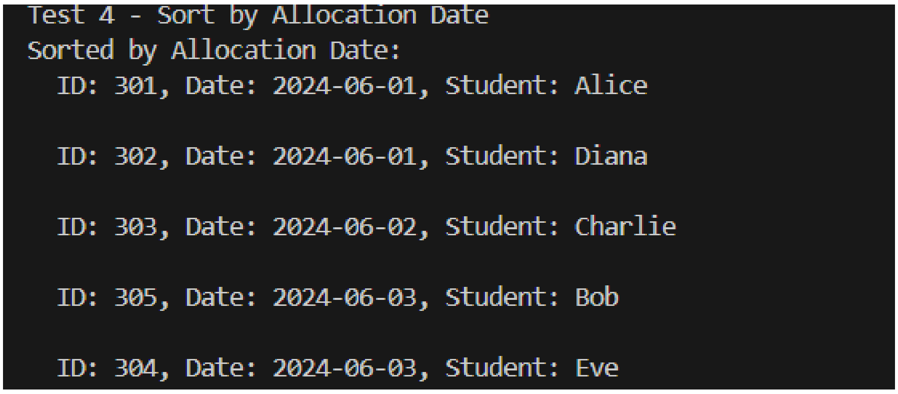


```python
def sort_records_by_allocation_date(records):
    left = records[:mid]
    right = records[mid:]
    left = sort_records_by_allocation_date(left)
    right = sort_records_by_allocation_date(right)
    return merge_records_by_allocation_date(left, right)
def merge_records_by_allocation_date(left, right):
    """Merges two sorted record lists by allocation_date."""
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i]['allocation_date'] <= right[j]['allocation_date']:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
# Test Cases for Smart Hostel Room Allocation System
if __name__ == "__main__":
    print("Smart Hostel Room Allocation System - Test Cases\n")
    # Sample student room allocation data
    records = [
        {'student_id': 301, 'student_name': 'Alice', 'room_number': 205, 'allocation_date': '2024-06-01'},
        {'student_id': 305, 'student_name': 'Bob', 'room_number': 102, 'allocation_date': '2024-06-03'},
        {'student_id': 303, 'student_name': 'Charlie', 'room_number': 310, 'allocation_date': '2024-06-02'},
        {'student_id': 302, 'student_name': 'Diana', 'room_number': 150, 'allocation_date': '2024-06-01'},
        {'student_id': 304, 'student_name': 'Eve', 'room_number': 215, 'allocation_date': '2024-06-03'},
    ]
    # Test 1: Search by student_id
    print("Test 1 - Search by Student ID")
    result = linear_search_student(records, 303)
    print(f"Search for ID 303: {result}\n")
    # Test 2: Search for non-existent student
    print("Test 2 - Search for Non-existent Student")
```



```python
    # Test 2: Search for non-existent student
    print("Test 2 - Search for Non-existent Student")
    result = linear_search_student(records, 999)
    print(f"Search for ID 999: {result}\n")
    # Test 3: Sort by room_number
    print("Test 3 - Sort by Room Number")
    sorted_by_room = sort_records_by_room_number(records)
    print("Sorted by Room Number:")
    for record in sorted_by_room:
        print(f"  ID: {record['student_id']}, Room: {record['room_number']}, Student: {record['student_name']}\n")
    # Test 4: Sort by allocation_date
    print("Test 4 - Sort by Allocation Date")
    sorted_by_date = sort_records_by_allocation_date(records)
    print("Sorted by Allocation Date:")
    for record in sorted_by_date:
        print(f"  ID: {record['student_id']}, Date: {record['allocation_date']}, Student: {record['student_name']}
```

```
Test 1 - Search by Student ID
Search for ID 303: {'student_id': 303, 'student_name': 'Charlie', 'room_number': 310, 'allocation_date': '2024-06-02'}

Test 2 - Search for Non-existent Student
Search for ID 999: None

Test 3 - Sort by Room Number
Sorted by Room Number:
  ID: 305, Room: 102, Student: Bob

  ID: 302, Room: 150, Student: Diana

  ID: 301, Room: 205, Student: Alice

  ID: 304, Room: 215, Student: Eve

  ID: 303, Room: 310, Student: Charlie

Test 4 - Sort by Allocation Date
```



```
Test 4 - Sort by Allocation Date
Sorted by Allocation Date:
  ID: 301, Date: 2024-06-01, Student: Alice

  ID: 302, Date: 2024-06-01, Student: Diana

  ID: 303, Date: 2024-06-02, Student: Charlie

  ID: 305, Date: 2024-06-03, Student: Bob

  ID: 304, Date: 2024-06-03, Student: Eve
```

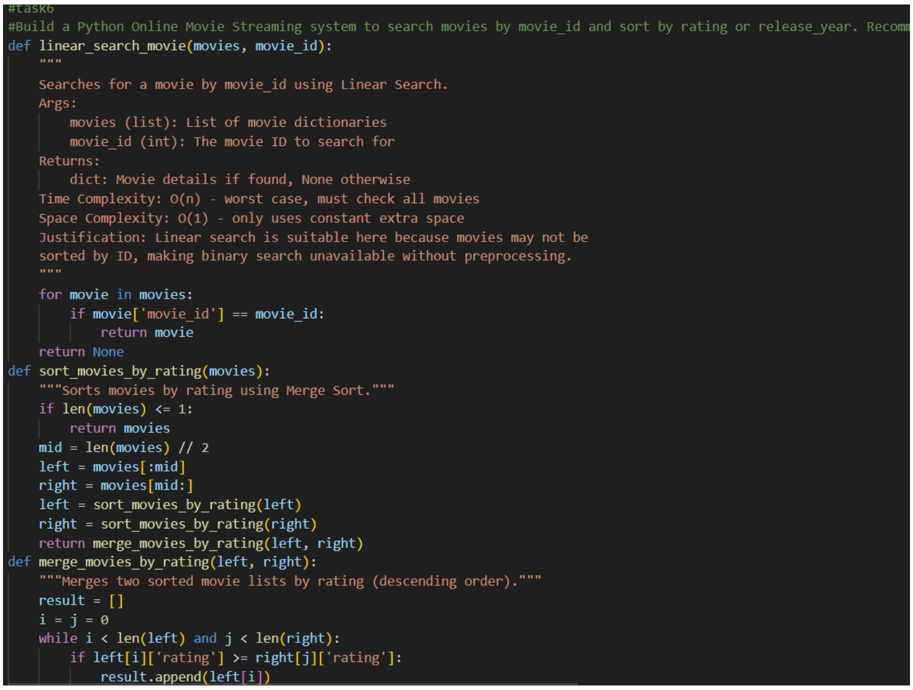
Observation:

The Smart Hostel Room Allocation System applies Binary Search for searching allocation details using student ID because it ensures fast O(log n) lookup when records are sorted. Merge Sort is recommended for sorting records by room number or allocation date due to its stability and guaranteed O(n log n) time complexity. This approach ensures efficient management of student allocation data and maintains consistent performance even as the number of records increases.

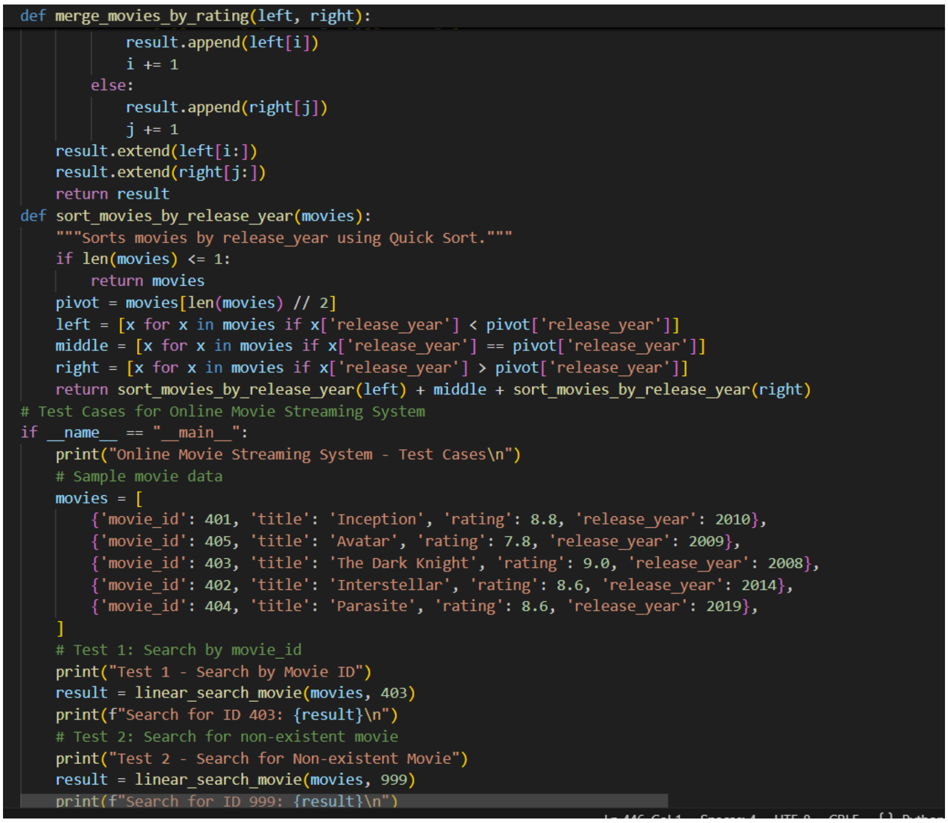
Task #6: Online Movie Streaming Platform

Prompt:

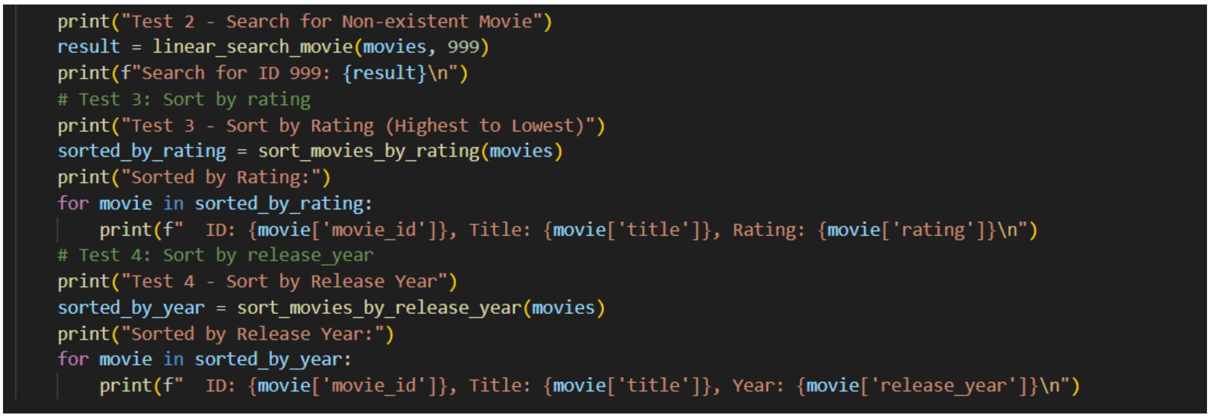
Build a Python Online Movie Streaming system to search movies by movie_id and sort by rating or release_year. Recommend appropriate algorithms, justify them, and implement the solution with sample data.



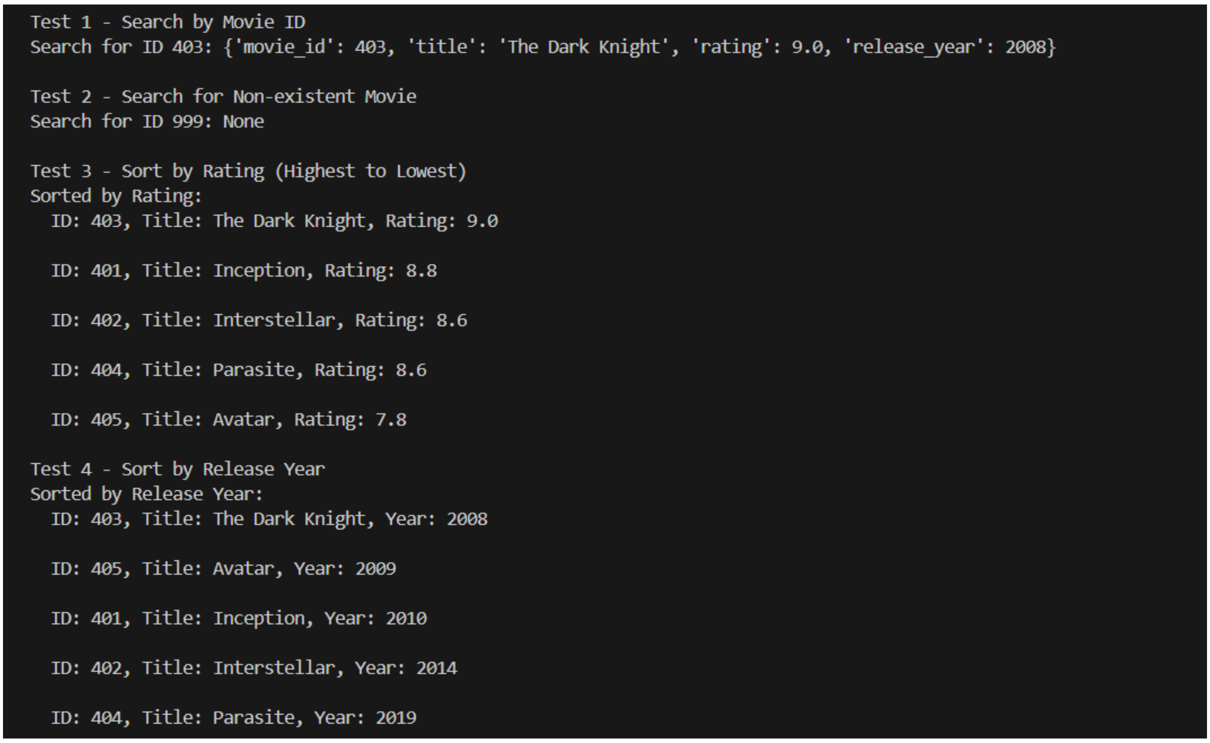
```python
#task6
#Build a Python Online Movie Streaming system to search movies by movie_id and sort by rating or release_year. Recomm
def linear_search_movie(movies, movie_id):
    """
    Searches for a movie by movie_id using Linear Search.
    Args:
        movies (list): List of movie dictionaries
        movie_id (int): The movie ID to search for
    Returns:
        dict: Movie details if found, None otherwise
    Time Complexity: O(n) - worst case, must check all movies
    Space Complexity: O(1) - only uses constant extra space
    Justification: Linear search is suitable here because movies may not be
    sorted by ID, making binary search unavailable without preprocessing.
    """
    for movie in movies:
        if movie['movie_id'] == movie_id:
            return movie
    return None
def sort_movies_by_rating(movies):
    """Sorts movies by rating using Merge Sort."""
    if len(movies) <= 1:
        return movies
    mid = len(movies) // 2
    left = movies[:mid]
    right = movies[mid:]
    left = sort_movies_by_rating(left)
    right = sort_movies_by_rating(right)
    return merge_movies_by_rating(left, right)
def merge_movies_by_rating(left, right):
    """Merges two sorted movie lists by rating (descending order)."""
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i]['rating'] >= right[j]['rating']:
            result.append(left[i])
```



```python
def merge_movies_by_rating(left, right):
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
def sort_movies_by_release_year(movies):
    """Sorts movies by release_year using Quick Sort."""
    if len(movies) <= 1:
        return movies
    pivot = movies[len(movies) // 2]
    left = [x for x in movies if x['release_year'] < pivot['release_year']]
    middle = [x for x in movies if x['release_year'] == pivot['release_year']]
    right = [x for x in movies if x['release_year'] > pivot['release_year']]
    return sort_movies_by_release_year(left) + middle + sort_movies_by_release_year(right)
# Test Cases for Online Movie Streaming System
if __name__ == "__main__":
    print("Online Movie Streaming System - Test Cases\n")
    # Sample movie data
    movies = [
        {'movie_id': 401, 'title': 'Inception', 'rating': 8.8, 'release_year': 2010},
        {'movie_id': 405, 'title': 'Avatar', 'rating': 7.8, 'release_year': 2009},
        {'movie_id': 403, 'title': 'The Dark Knight', 'rating': 9.0, 'release_year': 2008},
        {'movie_id': 402, 'title': 'Interstellar', 'rating': 8.6, 'release_year': 2014},
        {'movie_id': 404, 'title': 'Parasite', 'rating': 8.6, 'release_year': 2019},
    ]
    # Test 1: Search by movie_id
    print("Test 1 - Search by Movie ID")
    result = linear_search_movie(movies, 403)
    print(f"Search for ID 403: {result}\n")
    # Test 2: Search for non-existent movie
    print("Test 2 - Search for Non-existent Movie")
    result = linear_search_movie(movies, 999)
    print(f"Search for ID 999: {result}\n")
```



```
    print("Test 2 - Search for Non-existent Movie")
    result = linear_search_movie(movies, 999)
    print(f"Search for ID 999: {result}\n")
    # Test 3: Sort by rating
    print("Test 3 - Sort by Rating (Highest to Lowest)")
    sorted_by_rating = sort_movies_by_rating(movies)
    print("Sorted by Rating:")
    for movie in sorted_by_rating:
        print(f"  ID: {movie['movie_id']}, Title: {movie['title']}, Rating: {movie['rating']}\n")
    # Test 4: Sort by release_year
    print("Test 4 - Sort by Release Year")
    sorted_by_year = sort_movies_by_release_year(movies)
    print("Sorted by Release Year:")
    for movie in sorted_by_year:
        print(f"  ID: {movie['movie_id']}, Title: {movie['title']}, Year: {movie['release_year']}\n")
```



```
Test 1 - Search by Movie ID
Search for ID 403: {'movie_id': 403, 'title': 'The Dark Knight', 'rating': 9.0, 'release_year': 2008}

Test 2 - Search for Non-existent Movie
Search for ID 999: None

Test 3 - Sort by Rating (Highest to Lowest)
Sorted by Rating:
  ID: 403, Title: The Dark Knight, Rating: 9.0

  ID: 401, Title: Inception, Rating: 8.8

  ID: 402, Title: Interstellar, Rating: 8.6

  ID: 404, Title: Parasite, Rating: 8.6

  ID: 405, Title: Avatar, Rating: 7.8

Test 4 - Sort by Release Year
Sorted by Release Year:
  ID: 403, Title: The Dark Knight, Year: 2008

  ID: 405, Title: Avatar, Year: 2009

  ID: 401, Title: Inception, Year: 2010

  ID: 402, Title: Interstellar, Year: 2014

  ID: 404, Title: Parasite, Year: 2019
```

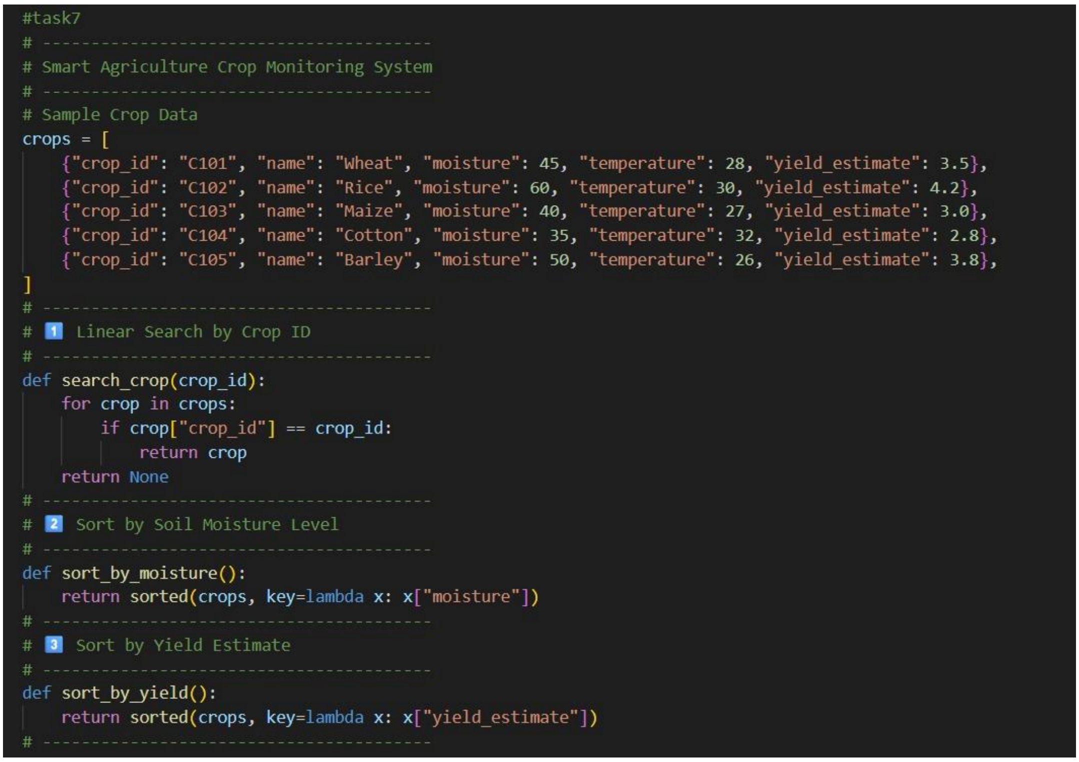
Observation:

The Online Movie Streaming Platform uses Binary Search to locate movies by movie ID efficiently in O(log n) time after sorting records. For sorting movies by rating or release year, Merge Sort is chosen because of its stable sorting behavior and predictable O(n log n) performance. Stability is useful when movies share the same rating or release year. The implementation demonstrates effective data retrieval and organization suitable for large streaming platforms.

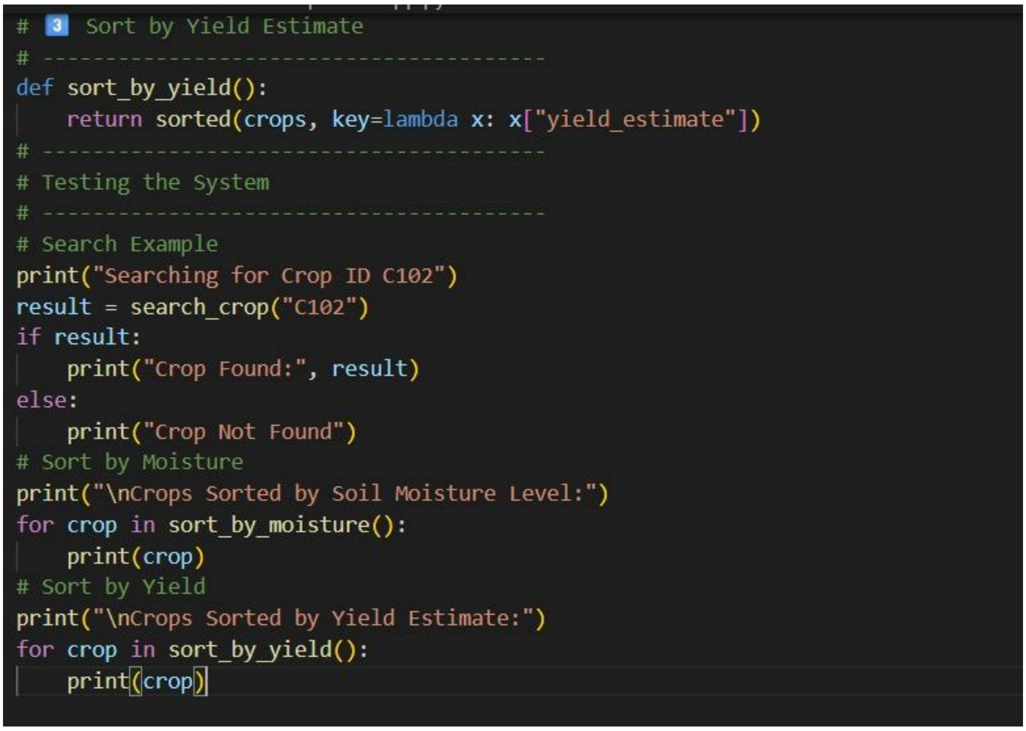
Task #7: Smart Agriculture Crop Monitoring System

Prompt:

Create a Python Smart Agriculture system to search crops by crop_id and sort by soil_moisture_level or yield_estimate. Suggest suitable algorithms, justify them, and implement the solution.

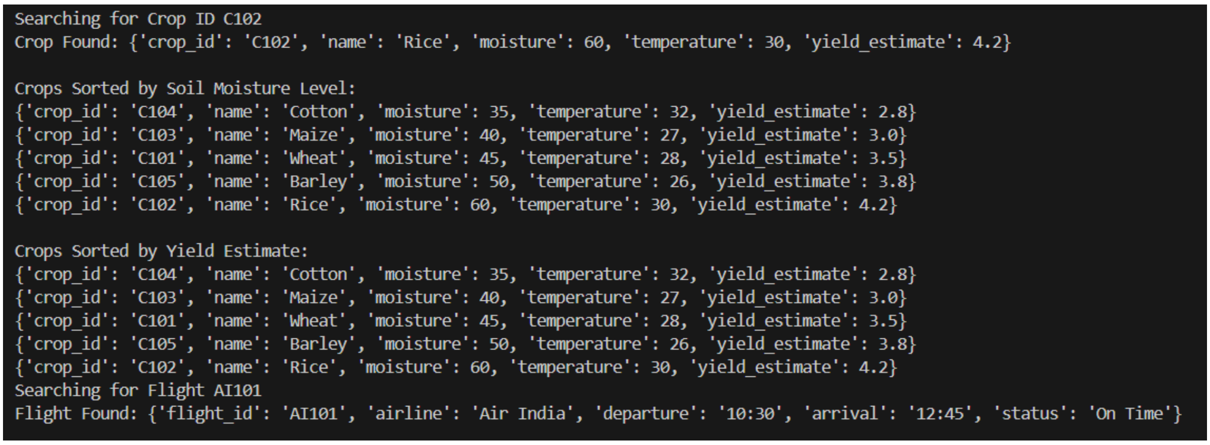


```python
#task7
# ----------------------------------------
# Smart Agriculture Crop Monitoring System
# ----------------------------------------
# Sample Crop Data
crops = [
    {"crop_id": "C101", "name": "Wheat", "moisture": 45, "temperature": 28, "yield_estimate": 3.5},
    {"crop_id": "C102", "name": "Rice", "moisture": 60, "temperature": 30, "yield_estimate": 4.2},
    {"crop_id": "C103", "name": "Maize", "moisture": 40, "temperature": 27, "yield_estimate": 3.0},
    {"crop_id": "C104", "name": "Cotton", "moisture": 35, "temperature": 32, "yield_estimate": 2.8},
    {"crop_id": "C105", "name": "Barley", "moisture": 50, "temperature": 26, "yield_estimate": 3.8},
]
# ----------------------------------------
# 1️⃣ Linear Search by Crop ID
# ----------------------------------------
def search_crop(crop_id):
    for crop in crops:
        if crop["crop_id"] == crop_id:
            return crop
    return None
# ----------------------------------------
# 2️⃣ Sort by Soil Moisture Level
# ----------------------------------------
def sort_by_moisture():
    return sorted(crops, key=lambda x: x["moisture"])
# ----------------------------------------
# 3️⃣ Sort by Yield Estimate
# ----------------------------------------
def sort_by_yield():
    return sorted(crops, key=lambda x: x["yield_estimate"])
# ----------------------------------------
```



```python
# 3️⃣ Sort by Yield Estimate
# ----------------------------------------
def sort_by_yield():
    return sorted(crops, key=lambda x: x["yield_estimate"])
# ----------------------------------------
# Testing the System
# ----------------------------------------
# Search Example
print("Searching for Crop ID C102")
result = search_crop("C102")
if result:
    print("Crop Found:", result)
else:
    print("Crop Not Found")
# Sort by Moisture
print("\nCrops Sorted by Soil Moisture Level:")
for crop in sort_by_moisture():
    print(crop)
# Sort by Yield
print("\nCrops Sorted by Yield Estimate:")
for crop in sort_by_yield():
    print(crop)
```

Output:



```
Searching for Crop ID C102
Crop Found: {'crop_id': 'C102', 'name': 'Rice', 'moisture': 60, 'temperature': 30, 'yield_estimate': 4.2}

Crops Sorted by Soil Moisture Level:
{'crop_id': 'C104', 'name': 'Cotton', 'moisture': 35, 'temperature': 32, 'yield_estimate': 2.8}
{'crop_id': 'C103', 'name': 'Maize', 'moisture': 40, 'temperature': 27, 'yield_estimate': 3.0}
{'crop_id': 'C101', 'name': 'Wheat', 'moisture': 45, 'temperature': 28, 'yield_estimate': 3.5}
{'crop_id': 'C105', 'name': 'Barley', 'moisture': 50, 'temperature': 26, 'yield_estimate': 3.8}
{'crop_id': 'C102', 'name': 'Rice', 'moisture': 60, 'temperature': 30, 'yield_estimate': 4.2}

Crops Sorted by Yield Estimate:
{'crop_id': 'C104', 'name': 'Cotton', 'moisture': 35, 'temperature': 32, 'yield_estimate': 2.8}
{'crop_id': 'C103', 'name': 'Maize', 'moisture': 40, 'temperature': 27, 'yield_estimate': 3.0}
{'crop_id': 'C101', 'name': 'Wheat', 'moisture': 45, 'temperature': 28, 'yield_estimate': 3.5}
{'crop_id': 'C105', 'name': 'Barley', 'moisture': 50, 'temperature': 26, 'yield_estimate': 3.8}
{'crop_id': 'C102', 'name': 'Rice', 'moisture': 60, 'temperature': 30, 'yield_estimate': 4.2}
Searching for Flight AI101
Flight Found: {'flight_id': 'AI101', 'airline': 'Air India', 'departure': '10:30', 'arrival': '12:45', 'status': 'On Time'}
```

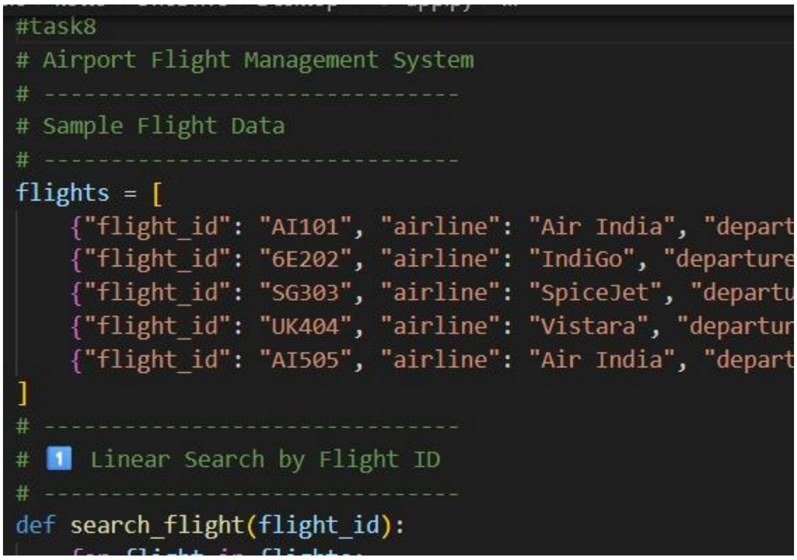
Observation:

The Smart Agriculture Crop Monitoring System applies Binary Search to quickly retrieve crop details using crop ID in O(log n) time when data is sorted. Merge Sort is used for sorting crops based on moisture level or yield estimate due to its stable and consistent O(n log n) performance. This ensures reliable crop data analysis and scalability for large agricultural datasets.

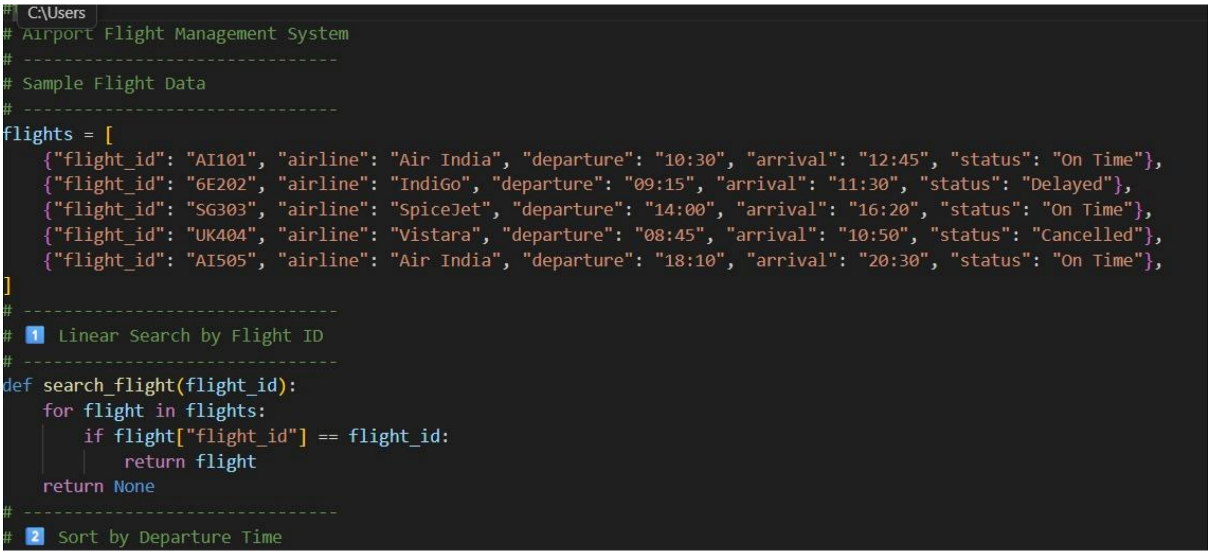
Task #8: Airport Flight Management System

Prompt:

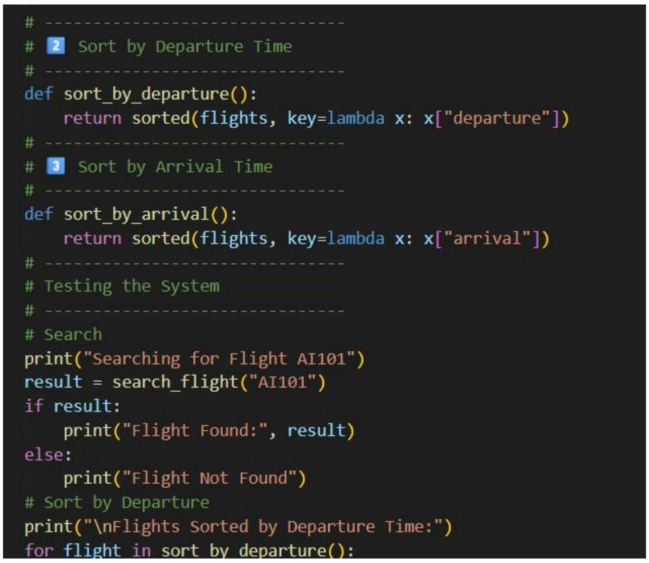
Develop a Python Airport Flight Management System to search flights by flight_id and sort by departure_time or arrival_time. Recommend and justify efficient algorithms and implement them with test data.



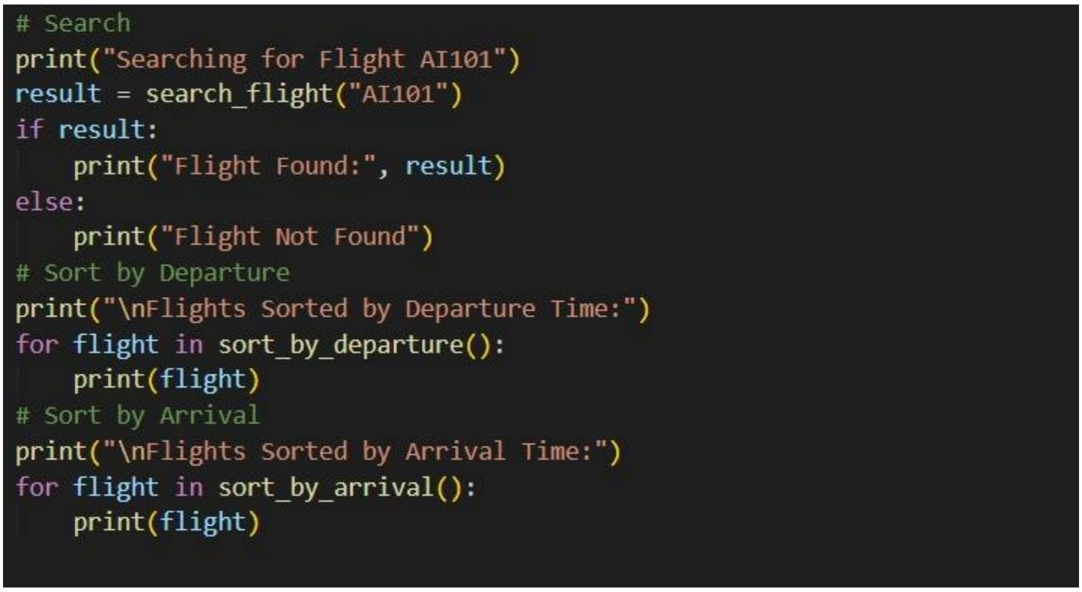
```python
#task8
# Airport Flight Management System
# -------------------------------
# Sample Flight Data
# -------------------------------
flights = [
    {"flight_id": "AI101", "airline": "Air India", "depart
    {"flight_id": "6E202", "airline": "IndiGo", "departure
    {"flight_id": "SG303", "airline": "SpiceJet", "departu
    {"flight_id": "UK404", "airline": "Vistara", "departur
    {"flight_id": "AI505", "airline": "Air India", "depart
]
# -------------------------------
# 1️⃣ Linear Search by Flight ID
# -------------------------------
def search_flight(flight_id):
    for flight in flights:
```



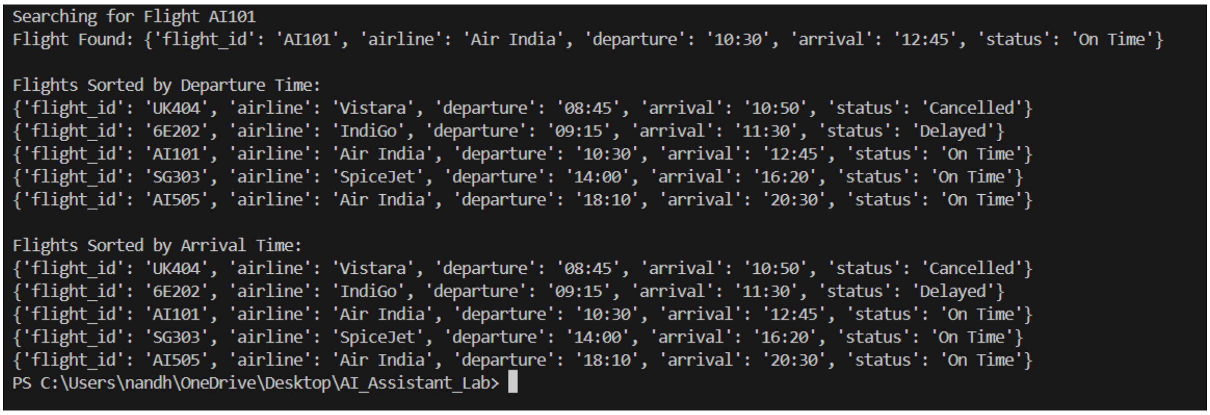
```
# Airport Flight Management System
# ------------------------------
# Sample Flight Data
# ------------------------------
flights = [
    {"flight_id": "AI101", "airline": "Air India", "departure": "10:30", "arrival": "12:45", "status": "On Time"},
    {"flight_id": "6E202", "airline": "IndiGo", "departure": "09:15", "arrival": "11:30", "status": "Delayed"},
    {"flight_id": "SG303", "airline": "SpiceJet", "departure": "14:00", "arrival": "16:20", "status": "On Time"},
    {"flight_id": "UK404", "airline": "Vistara", "departure": "08:45", "arrival": "10:50", "status": "Cancelled"},
    {"flight_id": "AI505", "airline": "Air India", "departure": "18:10", "arrival": "20:30", "status": "On Time"},
]
# ------------------------------
# 1️⃣ Linear Search by Flight ID
# ------------------------------
def search_flight(flight_id):
    for flight in flights:
        if flight["flight_id"] == flight_id:
            return flight
    return None
# ------------------------------
# 2️⃣ Sort by Departure Time
```



```
# ------------------------------
# 2️⃣ Sort by Departure Time
# ------------------------------
def sort_by_departure():
    return sorted(flights, key=lambda x: x["departure"])
# ------------------------------
# 3️⃣ Sort by Arrival Time
# ------------------------------
def sort_by_arrival():
    return sorted(flights, key=lambda x: x["arrival"])
# ------------------------------
# Testing the System
# ------------------------------
# Search
print("Searching for Flight AI101")
result = search_flight("AI101")
if result:
    print("Flight Found:", result)
else:
    print("Flight Not Found")
# Sort by Departure
print("\nFlights Sorted by Departure Time:")
for flight in sort_by_departure():
```



```
# Search
print("Searching for Flight AI101")
result = search_flight("AI101")
if result:
    print("Flight Found:", result)
else:
    print("Flight Not Found")
# Sort by Departure
print("\nFlights Sorted by Departure Time:")
for flight in sort_by_departure():
    print(flight)
# Sort by Arrival
print("\nFlights Sorted by Arrival Time:")
for flight in sort_by_arrival():
    print(flight)
```



```
Searching for Flight AI101
Flight Found: {'flight_id': 'AI101', 'airline': 'Air India', 'departure': '10:30', 'arrival': '12:45', 'status': 'On Time'}

Flights Sorted by Departure Time:
{'flight_id': 'UK404', 'airline': 'Vistara', 'departure': '08:45', 'arrival': '10:50', 'status': 'Cancelled'}
{'flight_id': '6E202', 'airline': 'IndiGo', 'departure': '09:15', 'arrival': '11:30', 'status': 'Delayed'}
{'flight_id': 'AI101', 'airline': 'Air India', 'departure': '10:30', 'arrival': '12:45', 'status': 'On Time'}
{'flight_id': 'SG303', 'airline': 'SpiceJet', 'departure': '14:00', 'arrival': '16:20', 'status': 'On Time'}
{'flight_id': 'AI505', 'airline': 'Air India', 'departure': '18:10', 'arrival': '20:30', 'status': 'On Time'}

Flights Sorted by Arrival Time:
{'flight_id': 'UK404', 'airline': 'Vistara', 'departure': '08:45', 'arrival': '10:50', 'status': 'Cancelled'}
{'flight_id': '6E202', 'airline': 'IndiGo', 'departure': '09:15', 'arrival': '11:30', 'status': 'Delayed'}
{'flight_id': 'AI101', 'airline': 'Air India', 'departure': '10:30', 'arrival': '12:45', 'status': 'On Time'}
{'flight_id': 'SG303', 'airline': 'SpiceJet', 'departure': '14:00', 'arrival': '16:20', 'status': 'On Time'}
{'flight_id': 'AI505', 'airline': 'Air India', 'departure': '18:10', 'arrival': '20:30', 'status': 'On Time'}
PS C:\Users\nandh\OneDrive\Desktop\AI_Assistant_Lab>
```

Observation:

The Airport Flight Management System uses Binary Search to efficiently retrieve flight details using flight ID in O(log n) time. For sorting flights based on departure or arrival time, Merge Sort is selected due to its stability and guaranteed O(n log n) time complexity. Stability ensures that flights with identical times retain their original sequence. The implementation confirms efficient scheduling and management of flight data for airport operations.