AIAC Assignment-12.5
CH UDAY
Hall Ticket:2303A510G2

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.

Prompt:

#Generate a python program create function for merge sort algorithm to sort a list of integers in ascending order. The function should take a list of integers as input and return a new list that is sorted in ascending order. The merge sort algorithm should be implemented using a divide-and-conquer approach, where the list is recursively divided into smaller sublists until each sublist contains only one element. Then, the sublists are merged back together in the correct order to produce the final sorted list. Additionally, the program should include error handling to ensure that the input is valid (i.e., it should be a list of integers) and should provide appropriate messages for invalid inputs.

#include time complexity and space complexity of the merge sort algorithm in the program's docstring.

Code:

```python
#include time complexity and space complexity of the merge sort algorithm in the program's docstring.
def merge_sort(arr):
    """
    Sorts a list of integers in ascending order using the merge sort algorithm.

    Args:
        arr (list): A list of integers to be sorted.
    Returns:
        list: A new list that is sorted in ascending order.
    Time Complexity: O(n log n) - The merge sort algorithm divides the list into halves log n times and merges n elements at each level.
    Space Complexity: O(n) - The merge sort algorithm requires additional space for the temporary arrays used during the merging process.
    """
    if not isinstance(arr, list) or not all(isinstance(x, int) for x in arr):
        raise ValueError("Input must be a list of integers.")

    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.
    Args:
        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:
            merged.append(right[right_index])
            right_index += 1

    merged.extend(left[left_index:])
    merged.extend(right[right_index:])

    return merged
# Example usage
if __name__ == "__main__":
    try:
        unsorted_list = [38, 27, 43, 3, 9, 82, 10]
        sorted_list = merge_sort(unsorted_list)
        print("Sorted list:", sorted_list)  # Output: Sorted list: [3, 9, 10, 27, 38, 43, 82]
    except ValueError as e:
        print(e)
```

Output:

```
Sorted list: [3, 9, 10, 27, 38, 43, 82]
```

Observations:

1. The program defines a merge_sort function that implements the merge sort algorithm to sort a list of integers in ascending order.
2. The merge function is a helper function that merges two sorted lists into a single sorted list.
3. The program includes error handling to ensure that the input is a valid list of integers and provides appropriate messages for invalid inputs.
4. The time complexity of the merge sort algorithm is O(n log n) and the space complexity is O(n).
5. The example usage demonstrates how to use the merge_sort function to sort an unsorted list of integers and prints the sorted result.

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.

Prompt:

#generate a binary search function that takes in a sorted list and a target value and returns the index of the target value in the list, or -1 if the target value is not in the list

The binary search algorithm should be implemented using a divide-and-conquer approach, where the list is repeatedly divided in half until the target value is found or the search space is exhausted. Additionally, the program should include error handling to ensure that the input is valid (i.e., it should be a sorted list of integers and a target value that is an integer) and should provide appropriate messages for invalid inputs.

#Include docstrings explaining best, average, and worst-case complexities.

Code:

```python
def binary_search(arr, target):
    """
    Performs binary search on a sorted list to find the index of the target value.

    Args:
        arr (list): A sorted list of integers.
        target (int): The value to search for in the list.

    Returns:
        int: The index of the target value in the list, or -1 if the target value is not found.
    Time Complexity:
        Best Case: O(1) - The target value is found at the middle index on the first comparison.
        Average Case: O(log n) - The search space is halved with each iteration, leading to logarithmic time complexity.
        Worst Case: O(log n) - The target value is not found after exhausting the search space, which also results in logarithmic time complexity.
    Space Complexity: O(1) - The binary search algorithm uses constant space as it only requires a few variables for indexing and comparison.
    """
    if not isinstance(arr, list) or not all(isinstance(x, int) for x in arr):
        raise ValueError("Input must be a sorted list of integers.")
    if not isinstance(target, int):
        raise ValueError("Target value must be an integer.")

    left, right = 0, len(arr) - 1

    while left <= right:
        mid = left + (right - left) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1
# Example usage
if __name__ == "__main__":
    try:
        sorted_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
        target_value = 5
        index = binary_search(sorted_list, target_value)
        if index != -1:
            print(f"Target value {target_value} found at index: {index}")
        else:
            print(f"Target value {target_value} not found in the list.")
    except ValueError as e:
        print(e)
```

Output:

```
Target value 5 found at index: 4
```

Observations:

1. The program defines a binary_search function that implements the binary search algorithm to find the index of a target value in a sorted list of integers.

2. The function includes error handling to ensure that the input is a valid sorted list of integers and that the target value is an integer, providing appropriate messages for invalid inputs.

3. The time complexity of the binary search algorithm is O(1) in the best case, O(log n) in the average case, and O(log n) in the worst case. The space complexity is O(1) since it uses constant space for indexing and comparison.

4. The example usage demonstrates how to use the binary_search function to search for a target value in a sorted list and prints the result, indicating whether the target value was found and its index or if it was not found in the list.

5. The binary search algorithm is efficient for searching in sorted lists, and it significantly reduces the number of comparisons needed compared to a linear search, especially as the size of the list increases.

Task Description #3: Smart Healthcare Appointment Scheduling System

A healthcare platform maintains appointment records containing appointment ID, patient name, doctor name, appointment time, and consultation fee. The system needs to:

1. Search appointments using appointment ID.

2. Sort appointments based on time or consultation fee.

Student Task

• Use AI to recommend suitable searching and sorting algorithms.

• Justify the selected algorithms.

• Implement the algorithms in Python.

Prompt:

#Generate a Python program for a Smart Healthcare Appointment Scheduling System that supports searching appointments by appointment ID and sorting by appointment time or consultation fee.

#Use efficient algorithms, justify why they are suitable, and include sample appointment data for testing.

Code:

```python
class Appointment:
    def __init__(self, appointment_id, patient_name, doctor_name, appointment_time, consultation_fee):
        self.appointment_id = appointment_id
        self.patient_name = patient_name
        self.doctor_name = doctor_name
        self.appointment_time = appointment_time
        self.consultation_fee = consultation_fee

    def __repr__(self):
        return f"Appointment({self.appointment_id}, {self.patient_name}, {self.doctor_name}, {self.appointment_time}, {self.consultation_fee})"
class AppointmentScheduler:
    def __init__(self):
        self.appointments = []

    def add_appointment(self, appointment):
        self.appointments.append(appointment)

    def search_appointment_by_id(self, appointment_id):
        """Searches for an appointment by its ID using a linear search algorithm.
        Time Complexity: O(n) - In the worst case, we may have to check all appointments.
        Space Complexity: O(1) - We are using constant space for the search.
        """
        for appointment in self.appointments:
            if appointment.appointment_id == appointment_id:
                return appointment
        return None

    def sort_appointments_by_time(self):
        """Sorts appointments by appointment time using the built-in sorted function, which uses Timsort (a hybrid sorting algorithm derived from merge sort and insertion sort).
        Time Complexity: O(n log n) - Timsort has a worst-case time complexity of O(n log n).
        Space Complexity: O(n) - Timsort requires additional space for temporary arrays during sorting.
        """
        self.appointments.sort(key=lambda x: x.appointment_time)

    def sort_appointments_by_fee(self):
        """Sorts appointments by consultation fee using the built-in sorted function.
        Time Complexity: O(n log n) - Timsort has a worst-case time complexity of O(n log n).
        Space Complexity: O(n) - Timsort requires additional space for temporary arrays during sorting.
        """
        self.appointments.sort(key=lambda x: x.consultation_fee)
# Sample appointment data for testing
if __name__ == "__main__":
    scheduler = AppointmentScheduler()
    scheduler.add_appointment(Appointment(1, "Alice", "Dr. Smith", "2024-07-01 10:00", 100))
    scheduler.add_appointment(Appointment(2, "Bob", "Dr. Johnson", "2024-07-01 11:00", 150))
    scheduler.add_appointment(Appointment(3, "Charlie", "Dr. Lee", "2024-07-01 09:30", 120))

    # Search for an appointment by ID
    appointment = scheduler.search_appointment_by_id(2)
    print("Search result for appointment ID 2:", appointment)

    # Sort appointments by time
    scheduler.sort_appointments_by_time()
    print("Appointments sorted by time:", scheduler.appointments)

    # Sort appointments by consultation fee
    scheduler.sort_appointments_by_fee()
    print("Appointments sorted by consultation fee:", scheduler.appointments)
```

Output:

```
Search result for appointment ID 2: Appointment(2, Bob, Dr. Johnson, 2024-07-01 11:00, 150)
Appointments sorted by time: [Appointment(3, Charlie, Dr. Lee, 2024-07-01 09:30, 120), Appointment(1, Alice, Dr. Smith, 2024-07-01 10:00, 100), Appointment(2, Bob, Dr. Johnson, 2024-07-01 11:00, 150)]
Appointments sorted by consultation fee: [Appointment(1, Alice, Dr. Smith, 2024-07-01 10:00, 100), Appointment(3, Charlie, Dr. Lee, 2024-07-01 09:30, 120), Appointment(2, Bob, Dr. Johnson, 2024-07-01 11:00, 150)]
```

Observations:

1. The program defines an Appointment class to represent individual appointments and an AppointmentScheduler class to manage the collection of appointments.

2. The AppointmentScheduler class includes methods for adding appointments, searching appointments by ID, and sorting appointments by time or consultation fee.

3. The search_appointment_by_id method uses a linear search algorithm, which is suitable for small datasets but may not be efficient for larger datasets. For larger datasets, a more efficient search algorithm (like binary search) could be implemented if the appointments are sorted by ID.

4. The sorting methods use the built-in sorted function, which is efficient for general-purpose sorting and is suitable for this application. Timsort, the underlying algorithm, has a good performance on real-world data and is stable, making it a good choice for sorting appointments.

5. The sample appointment data provided allows for testing the search and sorting functionalities effectively, demonstrating how the system can manage and organize appointments based on different criteria.

Task Description #4: Railway Ticket Reservation System Scenario

A railway reservation system stores booking details such as ticket ID, passenger name, train number, seat number, and travel date. The system must:

1. Search tickets using ticket ID.

2. Sort bookings based on travel date or seat number.

Student Task

• Identify efficient algorithms using AI assistance.

• Justify the algorithm choices.

• Implement searching and sorting in Python.

Prompt:

#Generate a Python program for a Railway Ticket Reservation System that supports searching tickets by ticket ID and sorting bookings by travel date or seat number.

#Use efficient searching and sorting algorithms, justify why they are suitable, and include sample booking data for testing.

Code:

```python
class Ticket:
    def __init__(self, ticket_id, passenger_name, travel_date, seat_number):
        self.ticket_id = ticket_id
        self.passenger_name = passenger_name
        self.travel_date = travel_date
        self.seat_number = seat_number

    def __repr__(self):
        return f"Ticket({self.ticket_id}, {self.passenger_name}, {self.travel_date}, {self.seat_number})"
class TicketReservationSystem:
    def __init__(self):
        self.tickets = []

    def add_ticket(self, ticket):
        self.tickets.append(ticket)

    def search_ticket_by_id(self, ticket_id):
        """Searches for a ticket by its ID using a linear search algorithm.
        Time Complexity: O(n) - In the worst case, we may have to check all tickets.
        Space Complexity: O(1) - We are using constant space for the search.
        """
        for ticket in self.tickets:
            if ticket.ticket_id == ticket_id:
                return ticket
        return None

    def sort_tickets_by_travel_date(self):
        """Sorts tickets by travel date using the built-in sorted function, which uses Timsort (a hybrid sorting algorithm derived from merge sort and insertion sort).
        Time Complexity: O(n log n) - Timsort has a worst-case time complexity of O(n log n).
        Space Complexity: O(n) - Timsort requires additional space for temporary arrays during sorting.
        """
        self.tickets.sort(key=lambda x: x.travel_date)

    def sort_tickets_by_seat_number(self):
        """Sorts tickets by seat number using the built-in sorted function.
        Time Complexity: O(n log n) - Timsort has a worst-case time complexity of O(n log n).
        Space Complexity: O(n) - Timsort requires additional space for temporary arrays during sorting.
        """
        self.tickets.sort(key=lambda x: x.seat_number)
# Sample booking data for testing
if __name__ == "__main__":
    reservation_system = TicketReservationSystem()
    reservation_system.add_ticket(Ticket(1, "Alice", "2024-08-01", "A1"))
    reservation_system.add_ticket(Ticket(2, "Bob", "2024-08-02", "B2"))
    reservation_system.add_ticket(Ticket(3, "Charlie", "2024-08-01", "A2"))

    # Search for a ticket by ID
    ticket = reservation_system.search_ticket_by_id(2)
    print("Search result for ticket ID 2:", ticket)

    # Sort tickets by travel date
    reservation_system.sort_tickets_by_travel_date()
    print("Tickets sorted by travel date:", reservation_system.tickets)

    # Sort tickets by seat number
    reservation_system.sort_tickets_by_seat_number()
    print("Tickets sorted by seat number:", reservation_system.tickets)
```

Output:

```
Search result for ticket ID 2: Ticket(2, Bob, 2024-08-02, B2)
Tickets sorted by travel date: [Ticket(1, Alice, 2024-08-01, A1), Ticket(3, Charlie, 2024-08-01, A2), Ticket(2, Bob, 2024-08-02, B2)]
Tickets sorted by seat number: [Ticket(1, Alice, 2024-08-01, A1), Ticket(3, Charlie, 2024-08-01, A2), Ticket(2, Bob, 2024-08-02, B2)]
```

Observations:

1. The program defines a Ticket class to represent individual tickets and a TicketReservationSystem class to manage the collection of tickets.

2. The search_ticket_by_id method uses a linear search algorithm, which is suitable for small datasets but may not be efficient for larger datasets. For larger datasets, a more efficient search algorithm like binary search could be implemented if the tickets are sorted by ticket ID.

3. The sort tickets by travel date and sort tickets by seat number methods use the built-in sorted function, which is efficient for sorting and has a worst-case time complexity of O(n log n). This is suitable for our use case as it can handle larger datasets efficiently.

4. The sample booking data allows us to test the functionality of searching and sorting tickets effectively.

5.Overall, the program provides a basic structure for a Railway Ticket Reservation System with efficient searching and sorting capabilities.

Task Description #5: Smart Hostel Room Allocation System

A hostel management system stores student room allocation details including student ID, room number, floor, and allocation date. The system needs to:

1. Search allocation details using student ID.

2. Sort records based on room number or allocation date.

Student Task

• Use AI to suggest optimized algorithms.

• Justify the selections.

• Implement the solution in Python.

Prompt:

#Generate a Python program for a Smart Hostel Room Allocation System that supports searching records by student ID and sorting by room number or allocation date.

#Use efficient searching and sorting algorithms, justify the choices, and include sample data for testing.

Code:

```python
class RoomAllocation:
    def __init__(self, student_id, student_name, room_number, allocation_date):
        self.student_id = student_id
        self.student_name = student_name
        self.room_number = room_number
        self.allocation_date = allocation_date

    def __repr__(self):
        return f"RoomAllocation({self.student_id}, {self.student_name}, {self.room_number}, {self.allocation_date})"
class HostelRoomAllocationSystem:
    def __init__(self):
        self.allocations = []

    def add_allocation(self, allocation):
        self.allocations.append(allocation)

    def search_allocation_by_student_id(self, student_id):
        """Searches for a room allocation by student ID using a linear search algorithm.
        Time Complexity: O(n) - In the worst case, we may have to check all allocations.
        Space Complexity: O(1) - We are using constant space for the search.
        """
        for allocation in self.allocations:
            if allocation.student_id == student_id:
                return allocation
        return None

    def sort_allocations_by_room_number(self):
        """Sorts allocations by room number using the built-in sorted function, which uses Timsort (a hybrid sorting algorithm derived from merge sort and insertion sort).
        Time Complexity: O(n log n) - Timsort has a worst-case time complexity of O(n log n).
        Space Complexity: O(n) - Timsort requires additional space for temporary arrays during sorting.
        """
        self.allocations.sort(key=lambda x: x.room_number)

    def sort_allocations_by_allocation_date(self):
        """Sorts allocations by allocation date using the built-in sorted function.
        Time Complexity: O(n log n) - Timsort has a worst-case time complexity of O(n log n).
        Space Complexity: O(n) - Timsort requires additional space for temporary arrays during sorting.
        """
        self.allocations.sort(key=lambda x: x.allocation_date)
# Sample data for testing
if __name__ == "__main__":
    allocation_system = HostelRoomAllocationSystem()
    allocation_system.add_allocation(RoomAllocation(1, "Alice", "101A", "2024-09-01"))
    allocation_system.add_allocation(RoomAllocation(2, "Bob", "102B", "2024-09-02"))
    allocation_system.add_allocation(RoomAllocation(3, "Charlie", "101B", "2024-09-01"))

    # Search for an allocation by student ID
    allocation = allocation_system.search_allocation_by_student_id(2)
    print("Search result for student ID 2:", allocation)

    # Sort allocations by room number
    allocation_system.sort_allocations_by_room_number()
    print("Allocations sorted by room number:", allocation_system.allocations)

    # Sort allocations by allocation date
    allocation_system.sort_allocations_by_allocation_date()
    print("Allocations sorted by allocation date:", allocation_system.allocations)
```

Output:

```
Search result for student ID 2: RoomAllocation(2, Bob, 102B, 2024-09-02)
Allocations sorted by room number: [RoomAllocation(1, Alice, 101A, 2024-09-01), RoomAllocation(3, Charlie, 101B, 2024-09-01), RoomAllocation(2, Bob, 102B, 2024-09-02)]
Allocations sorted by allocation date: [RoomAllocation(1, Alice, 101A, 2024-09-01), RoomAllocation(3, Charlie, 101B, 2024-09-01), RoomAllocation(2, Bob, 102B, 2024-09-02)]
```

Observations:

1. The program defines a RoomAllocation class to represent individual room allocations and a HostelRoomAllocationSystem class to manage the allocations.

2. The search_allocation_by_student_id method uses a linear search algorithm, which is suitable for small datasets but may not be efficient for larger datasets. For larger datasets, a more efficient search algorithm like binary search could be implemented if the data is sorted by student ID.

3. The sorting methods use the built-in sorted function, which is efficient for general-purpose sorting and is suitable for this application. Timsort is a hybrid sorting algorithm that performs well on real-world data, making it a good choice for sorting the allocations by room number and allocation date.

4. Sample data is provided for testing the search and sorting functionalities, demonstrating how the system can be used to manage hostel room allocations effectively.

5. The program can be further enhanced by adding features such as updating or deleting allocations, handling edge cases (e.g., duplicate student IDs), and improving the search algorithm for larger datasets.

Task Description #6: Online Movie Streaming Platform A

streaming service maintains movie records with movie ID, title,

genre, rating, and release year. The platform needs to:

1. Search movies by movie ID.

2. Sort movies based on rating or release year.

Student Task

• Recommend searching and sorting algorithms using AI.

• Justify the chosen algorithms.

• Implement Python functions.

Prompt:

#Generate a Python program for an Online Movie Streaming Platform that allows searching movies by movie ID and sorting movies by rating or release year.

#Use efficient searching and sorting algorithms, justify why they are suitable, and include sample movie data for testing.

Code:

```python
class Movie:
    def __init__(self, movie_id, title, rating, release_year):
        self.movie_id = movie_id
        self.title = title
        self.rating = rating
        self.release_year = release_year

    def __repr__(self):
        return f"Movie({self.movie_id}, {self.title}, {self.rating}, {self.release_year})"
class MovieStreamingPlatform:
    def __init__(self):
        self.movies = []

    def add_movie(self, movie):
        self.movies.append(movie)

    def search_movie_by_id(self, movie_id):
        """Searches for a movie by its ID using a linear search algorithm.
        Time Complexity: O(n) - In the worst case, we may have to check all movies.
        Space Complexity: O(1) - We are using constant space for the search.
        """
        for movie in self.movies:
            if movie.movie_id == movie_id:
                return movie
        return None

    def sort_movies_by_rating(self):
        """Sorts movies by rating using the built-in sorted function, which uses Timsort (a hybrid sorting algorithm derived from merge sort and insertion s
        Time Complexity: O(n log n) - Timsort has a worst-case time complexity of O(n log n).
        Space Complexity: O(n) - Timsort requires additional space for temporary arrays during sorting.
        """
        self.movies.sort(key=lambda x: x.rating, reverse=True)

    def sort_movies_by_release_year(self):
        """Sorts movies by release year using the built-in sorted function.
        Time Complexity: O(n log n) - Timsort has a worst-case time complexity of O(n log n).
        Space Complexity: O(n) - Timsort requires additional space for temporary arrays during sorting.
        """
        self.movies.sort(key=lambda x: x.release_year)
# Sample movie data for testing
if __name__ == "__main__":
    platform = MovieStreamingPlatform()
    platform.add_movie(Movie(1, "Inception", 8.8, 2010))
    platform.add_movie(Movie(2, "The Matrix", 8.7, 1999))
    platform.add_movie(Movie(3, "Interstellar", 8.6, 2014))

    # Search for a movie by ID
    movie = platform.search_movie_by_id(2)
    print("Search result for movie ID 2:", movie)

    # Sort movies by rating
    platform.sort_movies_by_rating()
    print("Movies sorted by rating:", platform.movies)

    # Sort movies by release year
    platform.sort_movies_by_release_year()
    print("Movies sorted by release year:", platform.movies)
```

Output:

```
Search result for movie ID 2: Movie(2, The Matrix, 8.7, 1999)
Movies sorted by rating: [Movie(1, Inception, 8.8, 2010), Movie(2, The Matrix, 8.7, 1999), Movie(3, Interstellar, 8.6, 2014)]
Movies sorted by release year: [Movie(2, The Matrix, 8.7, 1999), Movie(1, Inception, 8.8, 2010), Movie(3, Interstellar, 8.6, 2014)]
```

Observations:

1. The program defines a Movie class to represent individual movies and a MovieStreamingPlatform class to manage the collection of movies.

2. The search_movie_by_id method uses a linear search algorithm, which is suitable for small datasets but may not be efficient for larger datasets. For larger datasets, a more efficient search algorithm like binary search could be implemented if the data is sorted by movie ID.

3. The sorting methods use the built-in sorted function, which is efficient for general-purpose sorting and is appropriate for this use case. Timsort is a hybrid sorting algorithm that performs well on real-world data, making it a good choice for sorting movies by rating and release year.

4. The sample movie data provided allows for testing the search and sorting functionalities effectively, demonstrating how the platform can manage and organize movie information based on user preferences.

5. Overall, the program is designed to be simple and efficient for managing a collection of movies, allowing users to easily search for specific movies and sort them based on their preferences.

Task Description #7: Smart Agriculture Crop Monitoring System

An agriculture monitoring system stores crop data with crop ID, crop name, soil moisture level, temperature, and yield estimate.

Farmers need to:

1. Search crop details using crop ID.

2. Sort crops based on moisture level or yield estimate.

Student Task

• Use AI-assisted reasoning to select algorithms.

• Justify algorithm suitability.

• Implement searching and sorting in Python.

Prompt:

#Generate a Python program for a Smart Agriculture Crop Monitoring System that supports searching crops by crop ID and sorting by soil moisture level or yield estimate.

#Use efficient searching and sorting algorithms, justify why they are appropriate, and include sample crop data for demonstration.

Code:

```python
class Crop:
    def __init__(self, crop_id, crop_name, soil_moisture_level, yield_estimate):
        self.crop_id = crop_id
        self.crop_name = crop_name
        self.soil_moisture_level = soil_moisture_level
        self.yield_estimate = yield_estimate

    def __repr__(self):
        return f"Crop({self.crop_id}, {self.crop_name}, {self.soil_moisture_level}, {self.yield_estimate})"
class CropMonitoringSystem:
    def __init__(self):
        self.crops = []

    def add_crop(self, crop):
        self.crops.append(crop)

    def search_crop_by_id(self, crop_id):
        """Searches for a crop by its ID using a linear search algorithm.
        Time Complexity: O(n) - In the worst case, we may have to check all crops.
        Space Complexity: O(1) - We are using constant space for the search.
        """
        for crop in self.crops:
            if crop.crop_id == crop_id:
                return crop
        return None

    def sort_crops_by_soil_moisture(self):
        """Sorts crops by soil moisture level using the built-in sorted function, which uses Timsort (a hybrid sorting algorithm derived from merge sort and insertion sort).
        Time Complexity: O(n log n) - Timsort has a worst-case time complexity of O(n log n).
        Space Complexity: O(n) - Timsort requires additional space for temporary arrays during sorting.
        """
        self.crops.sort(key=lambda x: x.soil_moisture_level)

    def sort_crops_by_yield_estimate(self):
        """Sorts crops by yield estimate using the built-in sorted function.
        Time Complexity: O(n log n) - Timsort has a worst-case time complexity of O(n log n).
        Space Complexity: O(n) - Timsort requires additional space for temporary arrays during sorting.
        """
        self.crops.sort(key=lambda x: x.yield_estimate, reverse=True)
# Sample crop data for demonstration
if __name__ == "__main__":
    monitoring_system = CropMonitoringSystem()
    monitoring_system.add_crop(Crop(1, "Wheat", 30, 500))
    monitoring_system.add_crop(Crop(2, "Corn", 25, 600))
    monitoring_system.add_crop(Crop(3, "Rice", 35, 550))

    # Search for a crop by ID
    crop = monitoring_system.search_crop_by_id(2)
    print("Search result for crop ID 2:", crop)

    # Sort crops by soil moisture level
    monitoring_system.sort_crops_by_soil_moisture()
    print("Crops sorted by soil moisture level:", monitoring_system.crops)

    # Sort crops by yield estimate
    monitoring_system.sort_crops_by_yield_estimate()
    print("Crops sorted by yield estimate:", monitoring_system.crops)
```

Output:

```
Search result for crop ID 2: Crop(2, Corn, 25, 600)
Crops sorted by soil moisture level: [Crop(2, Corn, 25, 600), Crop(1, Wheat, 30, 500), Crop(3, Rice, 35, 550)]
Crops sorted by yield estimate: [Crop(2, Corn, 25, 600), Crop(3, Rice, 35, 550), Crop(1, Wheat, 30, 500)]
```

Observations:

1. The program defines a Crop class to represent individual crops and a CropMonitoringSystem class to manage the collection of crops.

2. The search_crop_by_id method uses a linear search algorithm, which is suitable for small datasets. For larger datasets, a more efficient search algorithm (like binary search) could be implemented if the crops are sorted by ID.

3. The sorting methods use Timsort, which is efficient for real-world data and has a good performance on partially sorted data, making it a suitable choice for sorting crops by soil moisture level and yield estimate.

4. The sample crop data demonstrates the functionality of searching and sorting within the CropMonitoringSystem, allowing for easy testing and validation of the implemented features.

5. The sorting by yield estimate is done in descending order to prioritize crops with higher yield estimates, which is a common requirement in agricultural monitoring systems.

Task Description #8: Airport Flight Management System

An airport system stores flight information including flight ID, airline name, departure time, arrival time, and status.

The system must:

1. Search flight details using flight ID.

2. Sort flights based on departure time or arrival time.

Student Task

• Use AI to recommend algorithms.

• Justify the algorithm selection.

• Implement searching and sorting logic in Python.

Prompt:

#Generate a Python program for an Airport Flight Management System that supports searching flights by flight ID and sorting by departure time or arrival time.

#Use efficient searching and sorting algorithms, justify the choices, and include sample flight data for testing.

Code:

```python
class Flight:
    def __init__(self, flight_id, airline, departure_time, arrival_time):
        self.flight_id = flight_id
        self.airline = airline
        self.departure_time = departure_time
        self.arrival_time = arrival_time

    def __repr__(self):
        return f"Flight({self.flight_id}, {self.airline}, {self.departure_time}, {self.arrival_time})"
class FlightManagementSystem:
    def __init__(self):
        self.flights = []

    def add_flight(self, flight):
        self.flights.append(flight)

    def search_flight_by_id(self, flight_id):
        """Searches for a flight by its ID using a linear search algorithm.
        Time Complexity: O(n) - In the worst case, we may have to check all flights.
        Space Complexity: O(1) - We are using constant space for the search.
        """
        for flight in self.flights:
            if flight.flight_id == flight_id:
                return flight
        return None

    def sort_flights_by_departure_time(self):
        """Sorts flights by departure time using the built-in sorted function, which uses Timsort (a hybrid sorting algorithm derived from merge sort and insertion sort).
        Time Complexity: O(n log n) - Timsort has a worst-case time complexity of O(n log n).
        Space Complexity: O(n) - Timsort requires additional space for temporary arrays during sorting.
        """
        self.flights.sort(key=lambda x: x.departure_time)

    def sort_flights_by_arrival_time(self):
        """Sorts flights by arrival time using the built-in sorted function.
        Time Complexity: O(n log n) - Timsort has a worst-case time complexity of O(n log n).
        Space Complexity: O(n) - Timsort requires additional space for temporary arrays during sorting.
        """
        self.flights.sort(key=lambda x: x.arrival_time)
# Sample flight data for testing
if __name__ == "__main__":
    flight_system = FlightManagementSystem()
    flight_system.add_flight(Flight(1, "Airline A", "2024-08-01 10:00", "2024-08-01 12:00"))
    flight_system.add_flight(Flight(2, "Airline B", "2024-08-01 09:00", "2024-08-01 11:00"))
    flight_system.add_flight(Flight(3, "Airline C", "2024-08-01 11:00", "2024-08-01 13:00"))

    # Search for a flight by ID
    flight = flight_system.search_flight_by_id(2)
    print("Search result for flight ID 2:", flight)

    # Sort flights by departure time
    flight_system.sort_flights_by_departure_time()
    print("Flights sorted by departure time:", flight_system.flights)

    # Sort flights by arrival time
    flight_system.sort_flights_by_arrival_time()
    print("Flights sorted by arrival time:", flight_system.flights)
```

Output:

```
Search result for flight ID 2: Flight(2, Airline B, 2024-08-01 09:00, 2024-08-01 11:00)
Flights sorted by departure time: [Flight(2, Airline B, 2024-08-01 09:00, 2024-08-01 11:00), Flight(1, Airline A, 2024-08-01 10:00, 2024-08-01 12:00), Flight(3, Airline C, 2024-08-01 11:00, 2024-08-01 13:00)]
Flights sorted by arrival time: [Flight(2, Airline B, 2024-08-01 09:00, 2024-08-01 11:00), Flight(1, Airline A, 2024-08-01 10:00, 2024-08-01 12:00), Flight(3, Airline C, 2024-08-01 11:00, 2024-08-01 13:00)]
```

Observation:

1. The program defines a Flight class to represent individual flights and a FlightManagementSystem class to manage the collection of flights.

2. The search_flight_by_id method uses a linear search algorithm, which is suitable for small datasets. For larger datasets, a more efficient search algorithm (like binary search) could be implemented if the flights were stored in a sorted structure.

3. The sorting methods use Timsort, which is efficient for real-world data and has a good performance on partially sorted data, making it a suitable choice for sorting flights by departure and arrival times.

4. The sample flight data provided allows for testing the search and sorting functionalities effectively, demonstrating the expected behavior of the system.

5. Overall, the program provides a basic structure for managing flight information, allowing for efficient searching and sorting based on key attributes of the flights.