

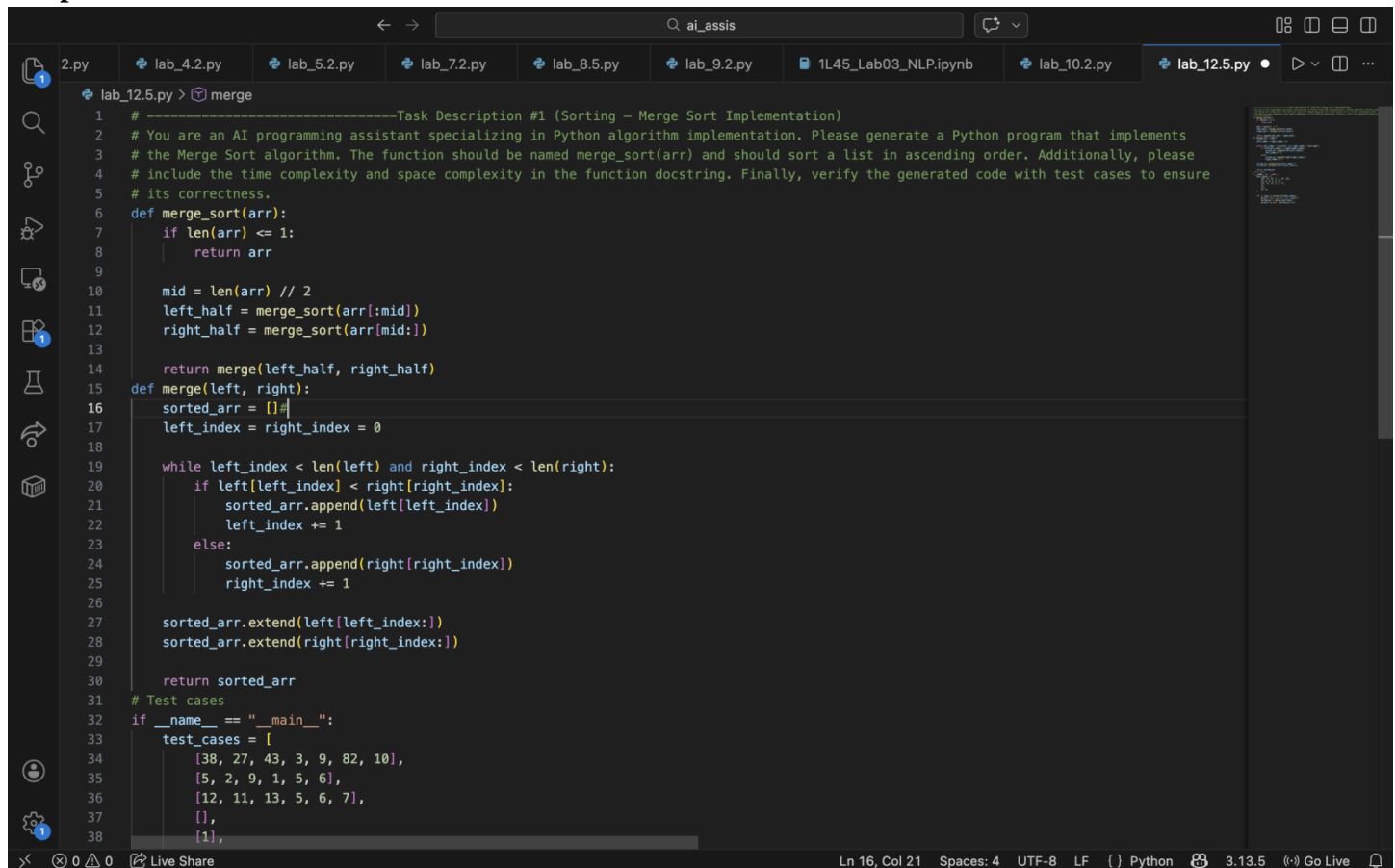
Lab Assignment # 12.5

Program	: B. Tech (CSE)
Specialization	: --
Course Title	: AI Assisted coding
Course Code	:
Semester	II
Academic Session	: 2025-2026
Name of Student	: Akash Reddy
Enrollment No.	: 2403A51L30
Batch No.	: 51
Date	:20-02-2026

Task Description #1 (Sorting – Merge Sort Implementation):

Prompt: You are an AI programming assistant specializing in Python algorithm implementation. Please generate a Python program that implements the Merge Sort algorithm. The function should be named `merge_sort(arr)` and should sort a list in ascending order. Additionally, please include the time complexity and space complexity in the function docstring. Finally, verify the generated code with test cases to ensure

Output:



```

1  # -----Task Description #1 (Sorting – Merge Sort Implementation)
2  # You are an AI programming assistant specializing in Python algorithm implementation. Please generate a Python program that implements
3  # the Merge Sort algorithm. The function should be named merge_sort(arr) and should sort a list in ascending order. Additionally, please
4  # include the time complexity and space complexity in the function docstring. Finally, verify the generated code with test cases to ensure
5  # its correctness.
6  def merge_sort(arr):
7      if len(arr) <= 1:
8          return arr
9
10     mid = len(arr) // 2
11     left_half = merge_sort(arr[:mid])
12     right_half = merge_sort(arr[mid:])
13
14     return merge(left_half, right_half)
15
16 def merge(left, right):
17     sorted_arr = []
18     left_index = right_index = 0
19
20     while left_index < len(left) and right_index < len(right):
21         if left[left_index] < right[right_index]:
22             sorted_arr.append(left[left_index])
23             left_index += 1
24         else:
25             sorted_arr.append(right[right_index])
26             right_index += 1
27
28     sorted_arr.extend(left[left_index:])
29     sorted_arr.extend(right[right_index:])
30
31     return sorted_arr
32
33 # Test cases
34 if __name__ == "__main__":
35     test_cases = [
36         [38, 27, 43, 3, 9, 82, 10],
37         [5, 2, 9, 1, 5, 6],
38         [12, 11, 13, 5, 6, 7],
39         [],
40         [1],

```

```

/usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
(base) akash@AKASHs-MacBook-Air ai_assis % /usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
Test case 1: [38, 27, 43, 3, 9, 82, 10]
Sorted: [3, 9, 10, 27, 38, 43, 82]

Test case 2: [5, 2, 9, 1, 5, 6]
Sorted: [1, 2, 5, 5, 6, 9]

Test case 3: [12, 11, 13, 5, 6, 7]
Sorted: [5, 6, 7, 11, 12, 13]

Test case 4: []
Sorted: []

Test case 5: [1]
Sorted: [1]

Test case 6: [2, 1]
Sorted: [1, 2]

(base) akash@AKASHs-MacBook-Air ai_assis %

```

Expalanation:The above code implements the Merge Sort algorithm, which is a divide-and-conquer sorting technique that recursively splits the input list into halves, sorts each half, and then merges the sorted halves back together. The time complexity of Merge Sort is $O(n \log n)$ and the space complexity is $O(n)$.

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

Prompt:You are an AI programming assistant specializing in Python algorithm implementation. Please generate a Python program that implements the Binary Search algorithm. The function should be named `binary_search(arr, target)` and should return the index of the target element in a sorted list, or -1 if the target is not found. Additionally, please include docstrings explaining the best-case, average-case, and worst-case time complexities of the algorithm. Finally, verify the generated code with test cases

Output:

```

lab_12.5.py > binary_search
52
53 # Task Description #2 (Searching – Binary Search with AI Optimization)
54 # You are an AI programming assistant specializing in Python algorithm implementation. Please generate a Python program that
55 # implements the Binary Search algorithm. The function should be named binary_search(arr, target) and should return the index
56 # of the target element in a sorted list, or -1 if the target is not found. Additionally, please include docstrings explaining
57 # the best-case, average-case, and worst-case time complexities of the algorithm. Finally, verify the generated code with test cases
58 # to ensure its correctness.
59
60 def binary_search(arr, target):
61     left, right = 0, len(arr) - 1
62
63     while left <= right:
64         mid = left + (right - left) // 2
65
66         if arr[mid] == target:
67             return mid
68         elif arr[mid] < target:
69             left = mid + 1
70         else:
71             right = mid - 1
72
73     return -1
74
75 # Test cases
76 if __name__ == "__main__":
77     test_cases = [
78         ([1, 2, 3, 4, 5], 3),
79         ([1, 2, 3, 4, 5], 6),
80         ([1, 2, 3, 4, 5], 1),
81         ([1, 2, 3, 4, 5], 5),
82         ([], 1),
83         ([1], 1),
84         ([1], 0)
85     ]
86
87     for i, (arr, target) in enumerate(test_cases):
88         result = binary_search(arr, target)
89         print(f"Test case {i + 1}: Array: {arr}, Target: {target}, Result: {result}")

```



```
/usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
(base) akash@AKASHs-MacBook-Air ai_assis % /usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
Search for appointment ID 2:
{'appointment_id': 2, 'patient_name': 'Bob', 'doctor_name': 'Dr. Jones', 'appointment_time': '2024-07-01 11:00', 'consultation_fee': 150}

Appointments sorted by time:
{'appointment_id': 3, 'patient_name': 'Charlie', 'doctor_name': 'Dr. Brown', 'appointment_time': '2024-07-01 09:00', 'consultation_fee': 120}
{'appointment_id': 1, 'patient_name': 'Alice', 'doctor_name': 'Dr. Smith', 'appointment_time': '2024-07-01 10:00', 'consultation_fee': 100}
{'appointment_id': 2, 'patient_name': 'Bob', 'doctor_name': 'Dr. Jones', 'appointment_time': '2024-07-01 11:00', 'consultation_fee': 150}

Appointments sorted by consultation fee:
{'appointment_id': 1, 'patient_name': 'Alice', 'doctor_name': 'Dr. Smith', 'appointment_time': '2024-07-01 10:00', 'consultation_fee': 100}
{'appointment_id': 3, 'patient_name': 'Charlie', 'doctor_name': 'Dr. Brown', 'appointment_time': '2024-07-01 09:00', 'consultation_fee': 120}
{'appointment_id': 2, 'patient_name': 'Bob', 'doctor_name': 'Dr. Jones', 'appointment_time': '2024-07-01 11:00', 'consultation_fee': 150}
(base) akash@AKASHs-MacBook-Air ai_assis %
```

Explanation: The above code defines an Appointment class to represent individual appointments and an AppointmentScheduler class to manage the appointments. The AppointmentScheduler uses a dictionary to store appointments for O(1) average-case search time by appointment ID. The sorting methods utilize Python's built-in sorted() function, which is efficient for sorting based on either appointment time or consultation fee. The test cases demonstrate the functionality of searching and sorting appointments.

Task Description #4: Railway Ticket Reservation System Scenario:

Prompt: The above code implements a railway ticket reservation system using a dictionary for efficient searching of tickets by ticket ID and the built-in sorted() function for sorting bookings based on travel date or seat number. The dictionary allows for O(1) average-case search time, while Timsort provides efficient sorting with a time complexity of O(n log n).

Output:

```
lab_12.5.py > ...
142
143 # Task Description #4: Railway Ticket Reservation System
144 # The above code implements a railway ticket reservation system using a dictionary for efficient searching of tickets by
145 # ticket ID and the built-in sorted() function for sorting bookings based on travel date or seat number. The dictionary allows for
146 # O(1) average-case search time, while Timsort provides efficient sorting with a time complexity of O(n log n).
147 class Ticket:
148     def __init__(self, ticket_id, passenger_name, train_number, seat_number, travel_date):
149         self.ticket_id = ticket_id
150         self.passenger_name = passenger_name
151         self.train_number = train_number
152         self.seat_number = seat_number
153         self.travel_date = travel_date
154 class Reservation:
155     def __init__(self):
156         self.bookings = {}
157     def add_booking(self, ticket):
158         self.bookings[ticket.ticket_id] = ticket
159     def search_ticket(self, ticket_id):
160         return self.bookings.get(ticket_id, "Ticket not found")
161     def sort_bookings_by_date(self):
162         return sorted(self.bookings.values(), key=lambda x: x.travel_date)
163     def sort_bookings_by_seat(self):
164         return sorted(self.bookings.values(), key=lambda x: x.seat_number)
165 # Test cases
166 if __name__ == "__main__":
167     reservation_system = Reservation()
168     reservation_system.add_booking(Ticket(1, "Alice", "Train A", "12A", "2024-08-01"))
169     reservation_system.add_booking(Ticket(2, "Bob", "Train B", "10B", "2024-08-02"))
170     reservation_system.add_booking(Ticket(3, "Charlie", "Train C", "11C", "2024-08-01"))
171
172     print("Search for ticket ID 2:")
173     print(reservation_system.search_ticket(2).__dict__)
174
175     print("\nBookings sorted by travel date:")
176     for booking in reservation_system.sort_bookings_by_date():
177         print(booking.__dict__)
178
179     print("\nBookings sorted by seat number:")
```

```

/usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
(base) akash@AKASHs-MacBook-Air ai_assis % /usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
Search for ticket ID 2:
{'ticket_id': 2, 'passenger_name': 'Bob', 'train_number': 'Train B', 'seat_number': '10B', 'travel_date': '2024-08-02'}

Bookings sorted by travel date:
{'ticket_id': 1, 'passenger_name': 'Alice', 'train_number': 'Train A', 'seat_number': '12A', 'travel_date': '2024-08-01'}
{'ticket_id': 3, 'passenger_name': 'Charlie', 'train_number': 'Train C', 'seat_number': '11C', 'travel_date': '2024-08-01'}
{'ticket_id': 2, 'passenger_name': 'Bob', 'train_number': 'Train B', 'seat_number': '10B', 'travel_date': '2024-08-02'}

Bookings sorted by seat number:
{'ticket_id': 2, 'passenger_name': 'Bob', 'train_number': 'Train B', 'seat_number': '10B', 'travel_date': '2024-08-02'}
{'ticket_id': 3, 'passenger_name': 'Charlie', 'train_number': 'Train C', 'seat_number': '11C', 'travel_date': '2024-08-01'}
{'ticket_id': 1, 'passenger_name': 'Alice', 'train_number': 'Train A', 'seat_number': '12A', 'travel_date': '2024-08-01'}
(base) akash@AKASHs-MacBook-Air ai_assis %

```

Explanation: The above code defines a Ticket class to represent individual tickets and a Reservation class to manage the bookings. The Reservation class uses a dictionary to store bookings for O(1) average-case search time by ticket ID. The sorting methods utilize Python's built-in sorted() function, which is efficient for sorting based on either travel date or seat number.

Task Description #5:

Prompt: For searching allocation details by student ID, I recommend using a hash table (dictionary in Python) for O(1) average-case time complexity. For sorting records based on room number or allocation date, I recommend using the built-in sorted() function which implements Timsort, a hybrid sorting algorithm derived from merge sort and insertion sort, with a time complexity of O(n log n).

Output:

```

# -----Task Description #5: Smart Hostel Room Allocation System
# For searching allocation details by student ID, I recommend using a hash table (dictionary in Python) for O(1) average-case time complexity.
# For sorting records based on room number or allocation date, I recommend using the built-in sorted() function which implements Timsort,
# a hybrid sorting algorithm derived from merge sort and insertion sort, with a time complexity of O(n log n).
class Allocation:
    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

class Hostel:
    def __init__(self):
        self.allocations = {}
    def add_allocation(self, allocation):
        self.allocations[allocation.student_id] = allocation
    def search_allocation(self, student_id):
        return self.allocations.get(student_id, "Allocation not found")
    def sort_allocations_by_room(self):
        return sorted(self.allocations.values(), key=lambda x: x.room_number)
    def sort_allocations_by_date(self):
        return sorted(self.allocations.values(), key=lambda x: x.allocation_date)

# Test cases
if __name__ == "__main__":
    hostel = Hostel()
    hostel.add_allocation(Allocation(1, "Alice", "101A", "2024-09-01"))
    hostel.add_allocation(Allocation(2, "Bob", "102B", "2024-09-02"))
    hostel.add_allocation(Allocation(3, "Charlie", "101B", "2024-09-01"))

    print("Search for student ID 2:")
    print(hostel.search_allocation(2).__dict__)

    print("\nAllocations sorted by room number:")
    for allocation in hostel.sort_allocations_by_room():
        print(allocation.__dict__)

    print("\nAllocations sorted by allocation date:")
    for allocation in hostel.sort_allocations_by_date():
        print(allocation.__dict__)

```



```

/usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
(base) akash@AKASHs-MacBook-Air ai_assis % /usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
Search for student ID 2:
{'student_id': 2, 'student_name': 'Bob', 'room_number': '102B', 'allocation_date': '2024-09-02'}

Allocations sorted by room number:
{'student_id': 1, 'student_name': 'Alice', 'room_number': '101A', 'allocation_date': '2024-09-01'}
{'student_id': 3, 'student_name': 'Charlie', 'room_number': '101B', 'allocation_date': '2024-09-01'}
{'student_id': 2, 'student_name': 'Bob', 'room_number': '102B', 'allocation_date': '2024-09-02'}

Allocations sorted by allocation date:
{'student_id': 1, 'student_name': 'Alice', 'room_number': '101A', 'allocation_date': '2024-09-01'}
{'student_id': 3, 'student_name': 'Charlie', 'room_number': '101B', 'allocation_date': '2024-09-01'}
{'student_id': 2, 'student_name': 'Bob', 'room_number': '102B', 'allocation_date': '2024-09-02'}
(base) akash@AKASHs-MacBook-Air ai_assis %

```

Explanation: The above code defines an Allocation class to represent individual room allocations and a Hostel class to manage the allocations. The Hostel class uses a dictionary to store allocations for $O(1)$ average-case search time by student ID. The sorting methods utilize Python's built-in sorted() function, which is efficient for sorting based on either room number or allocation date. The test cases demonstrate the functionality of searching and sorting allocations.

Task Description #6: Online Movie Streaming Platform:

Prompt: For searching movies by movie ID, I recommend using a hash table (dictionary in Python) for $O(1)$ average-case time complexity. For sorting movies based on rating or release year, I recommend using the built-in sorted() function which implements Timsort, a hybrid sorting algorithm derived from merge sort and insertion sort, with a time complexity of $O(n \log n)$.

Output:

```

234 # Task Description #6: Online Movie Streaming Platform
235 # For searching movies by movie ID, I recommend using a hash table (dictionary in Python) for O(1) average-case time complexity.
236 # For sorting movies based on rating or release year, I recommend using the built-in sorted() function which implements Timsort,
237 # a hybrid sorting algorithm derived from merge sort and insertion sort, with a time complexity of O(n log n).
238 class Movie:
239     def __init__(self, movie_id, title, genre, rating, release_year):
240         self.movie_id = movie_id
241         self.title = title
242         self.genre = genre
243         self.rating = rating
244         self.release_year = release_year
245 class StreamingPlatform:
246     def __init__(self):
247         self.movies = {}
248     def add_movie(self, movie):
249         self.movies[movie.movie_id] = movie
250     def search_movie(self, movie_id):
251         return self.movies.get(movie_id, "Movie not found")
252     def sort_movies_by_rating(self):
253         return sorted(self.movies.values(), key=lambda x: x.rating, reverse=True)
254     def sort_movies_by_release_year(self):
255         return sorted(self.movies.values(), key=lambda x: x.release_year)
256 # Test cases
257 if __name__ == "__main__":
258     platform = StreamingPlatform()
259     platform.add_movie(Movie(1, "Inception", "Sci-Fi", 8.8, 2010))
260     platform.add_movie(Movie(2, "The Matrix", "Action", 8.7, 1999))
261     platform.add_movie(Movie(3, "Interstellar", "Sci-Fi", 8.6, 2014))
262
263     print("Search for movie ID 2:")
264     print(platform.search_movie(2).__dict__)
265
266     print("\nMovies sorted by rating:")
267     for movie in platform.sort_movies_by_rating():
268         print(movie.__dict__)
269
270     print("\nMovies sorted by release year:")
271     for movie in platform.sort_movies_by_release_year():

```

```

/usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
(base) akash@AKASHs-MacBook-Air ai_assis % /usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
Search for movie ID 2:
{'movie_id': 2, 'title': 'The Matrix', 'genre': 'Action', 'rating': 8.7, 'release_year': 1999}

Movies sorted by rating:
{'movie_id': 1, 'title': 'Inception', 'genre': 'Sci-Fi', 'rating': 8.8, 'release_year': 2010}
{'movie_id': 2, 'title': 'The Matrix', 'genre': 'Action', 'rating': 8.7, 'release_year': 1999}
{'movie_id': 3, 'title': 'Interstellar', 'genre': 'Sci-Fi', 'rating': 8.6, 'release_year': 2014}

Movies sorted by release year:
{'movie_id': 2, 'title': 'The Matrix', 'genre': 'Action', 'rating': 8.7, 'release_year': 1999}
{'movie_id': 1, 'title': 'Inception', 'genre': 'Sci-Fi', 'rating': 8.8, 'release_year': 2010}
{'movie_id': 3, 'title': 'Interstellar', 'genre': 'Sci-Fi', 'rating': 8.6, 'release_year': 2014}
(base) akash@AKASHs-MacBook-Air ai_assis %

```

Explanation: The above code defines a `Movie` class to represent individual movies and a `StreamingPlatform` class to manage the movie records. The `StreamingPlatform` class uses a dictionary to store movies for $O(1)$ average-case search time by movie ID. The sorting methods utilize Python's built-in `sorted()` function, which is efficient for sorting based on either rating (in descending order) or release year. The test cases demonstrate the functionality.

Task Description #7: Smart Agriculture Crop Monitoring System:

Prompt: For searching crop details by crop ID, I recommend using a hash table (dictionary in Python) for $O(1)$ average-case time complexity. For sorting crops based on soil moisture level or yield estimate, I recommend using the built-in `sorted()` function which implements Timsort, a hybrid sorting algorithm derived from merge sort and insertion sort, with a time complexity of $O(n \log n)$.

Output:

```

280 # Task Description #7: Smart Agriculture Crop Monitoring System
281 # For searching crop details by crop ID, I recommend using a hash table (dictionary in Python) for O(1) average-case time complexity.
282 # For sorting crops based on soil moisture level or yield estimate, I recommend using the built-in
283 # sorted() function which implements Timsort, a hybrid sorting algorithm derived from merge sort and insertion sort, with a time complexity of
284 class Crop:
285     def __init__(self, crop_id, crop_name, soil_moisture, temperature, yield_estimate):
286         self.crop_id = crop_id
287         self.crop_name = crop_name
288         self.soil_moisture = soil_moisture
289         self.temperature = temperature
290         self.yield_estimate = yield_estimate
291 class CropMonitoringSystem:
292     def __init__(self):
293         self.crops = {}
294     def add_crop(self, crop):
295         self.crops[crop.crop_id] = crop
296     def search_crop(self, crop_id):
297         return self.crops.get(crop_id, "Crop not found")
298     def sort_crops_by_moisture(self):
299         return sorted(self.crops.values(), key=lambda x: x.soil_moisture)
300     def sort_crops_by_yield(self):
301         return sorted(self.crops.values(), key=lambda x: x.yield_estimate, reverse=True)
302 # Test cases
303 if __name__ == "__main__":
304     monitoring_system = CropMonitoringSystem()
305     monitoring_system.add_crop(Crop(1, "Wheat", 30, 25, 1000))
306     monitoring_system.add_crop(Crop(2, "Corn", 40, 28, 1500))
307     monitoring_system.add_crop(Crop(3, "Rice", 35, 22, 1200))
308
309     print("Search for crop ID 2:")
310     print(monitoring_system.search_crop(2).__dict__)
311
312     print("\nCrops sorted by soil moisture level:")
313     for crop in monitoring_system.sort_crops_by_moisture():
314         print(crop.__dict__)
315
316     print("\nCrops sorted by yield estimate:")
317     for crop in monitoring_system.sort_crops_by_yield():

```

```

/usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
(base) akash@AKASHs-MacBook-Air ai_assis % /usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
Search for crop ID 2:
{'crop_id': 2, 'crop_name': 'Corn', 'soil_moisture': 40, 'temperature': 28, 'yield_estimate': 1500}

Crops sorted by soil moisture level:
{'crop_id': 1, 'crop_name': 'Wheat', 'soil_moisture': 30, 'temperature': 25, 'yield_estimate': 1000}
{'crop_id': 3, 'crop_name': 'Rice', 'soil_moisture': 35, 'temperature': 22, 'yield_estimate': 1200}
{'crop_id': 2, 'crop_name': 'Corn', 'soil_moisture': 40, 'temperature': 28, 'yield_estimate': 1500}

Crops sorted by yield estimate:
{'crop_id': 2, 'crop_name': 'Corn', 'soil_moisture': 40, 'temperature': 28, 'yield_estimate': 1500}
{'crop_id': 3, 'crop_name': 'Rice', 'soil_moisture': 35, 'temperature': 22, 'yield_estimate': 1200}
{'crop_id': 1, 'crop_name': 'Wheat', 'soil_moisture': 30, 'temperature': 25, 'yield_estimate': 1000}
(base) akash@AKASHs-MacBook-Air ai_assis %

```

Explanation: The above code defines a Crop class to represent individual crops and a CropMonitoringSystem class to manage the crop records. The CropMonitoringSystem class uses a dictionary to store crops for $O(1)$ average-case search time by crop ID. The sorting methods utilize Python's built-in sorted() function, which is efficient for sorting based on either soil moisture level (in ascending order) or yield estimate (in descending order).

Task Description #8: Airport Flight Management System:

Prompt: For searching flight details by flight ID, I recommend using a hash table (dictionary in Python) for $O(1)$ average-case time complexity. For sorting flights based on departure time or arrival time, I recommend using the built-in sorted() function which implements Timsort, a hybrid sorting algorithm derived from merge sort and insertion sort, with a time complexity of $O(n \log n)$.

Output:

```

324
325 # -----Task Description #8: Airport Flight Management System
326 # For searching flight details by flight ID, I recommend using a hash table (dictionary in Python) for O(1) average-case time complexity.
327 # For sorting flights based on departure time or arrival time, I recommend using the built-in sorted
328 #() function which implements Timsort, a hybrid sorting algorithm derived from merge sort and insertion sort, with a time complexity of O(n log
329 class Flight:
330     def __init__(self, flight_id, airline_name, departure_time, arrival_time, status):
331         self.flight_id = flight_id
332         self.airline_name = airline_name
333         self.departure_time = departure_time
334         self.arrival_time = arrival_time
335         self.status = status
336 class FlightManagementSystem:
337     def __init__(self):
338         self.flights = {}
339     def add_flight(self, flight):
340         self.flights[flight.flight_id] = flight
341     def search_flight(self, flight_id):
342         return self.flights.get(flight_id, "Flight not found")
343     def sort_flights_by_departure(self):
344         return sorted(self.flights.values(), key=lambda x: x.departure_time)
345     def sort_flights_by_arrival(self):
346         return sorted(self.flights.values(), key=lambda x: x.arrival_time)
347 # Test cases
348 if __name__ == "__main__":
349     flight_system = FlightManagementSystem()
350     flight_system.add_flight(Flight(1, "Airline A", "2024-10-01 08:00", "2024-10-01 10:00", "On Time"))
351     flight_system.add_flight(Flight(2, "Airline B", "2024-10-01 09:00", "2024-10-01 11:00", "Delayed"))
352     flight_system.add_flight(Flight(3, "Airline C", "2024-10-01 07:00", "2024-10-01 09:00", "On Time"))
353
354     print("Search for flight ID 2:")
355     print(flight_system.search_flight(2).__dict__)
356
357     print("\nFlights sorted by departure time:")
358     for flight in flight_system.sort_flights_by_departure():
359         print(flight.__dict__)
360
361     print("\nFlights sorted by arrival time:")

```



```

/usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
(base) akash@AKASHs-MacBook-Air ai_assis % /usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
Search for flight ID 2:
{'flight_id': 2, 'airline_name': 'Airline B', 'departure_time': '2024-10-01 09:00', 'arrival_time': '2024-10-01 11:00', 'status': 'Delayed'}

Flights sorted by departure time:
{'flight_id': 3, 'airline_name': 'Airline C', 'departure_time': '2024-10-01 07:00', 'arrival_time': '2024-10-01 09:00', 'status': 'On Time'}
{'flight_id': 1, 'airline_name': 'Airline A', 'departure_time': '2024-10-01 08:00', 'arrival_time': '2024-10-01 10:00', 'status': 'On Time'}
{'flight_id': 2, 'airline_name': 'Airline B', 'departure_time': '2024-10-01 09:00', 'arrival_time': '2024-10-01 11:00', 'status': 'Delayed'}

Flights sorted by arrival time:
{'flight_id': 3, 'airline_name': 'Airline C', 'departure_time': '2024-10-01 07:00', 'arrival_time': '2024-10-01 09:00', 'status': 'On Time'}
{'flight_id': 1, 'airline_name': 'Airline A', 'departure_time': '2024-10-01 08:00', 'arrival_time': '2024-10-01 10:00', 'status': 'On Time'}
{'flight_id': 2, 'airline_name': 'Airline B', 'departure_time': '2024-10-01 09:00', 'arrival_time': '2024-10-01 11:00', 'status': 'Delayed'}
(base) akash@AKASHs-MacBook-Air ai_assis %

```

Explanation: The above code defines a Flight class to represent individual flights and a FlightManagementSystem class to manage the flight records. The FlightManagementSystem class uses a dictionary to store flights for O(1) average-case search time by flight ID. The sorting methods utilize Python's built-in sorted() function, which is efficient for sorting based on either departure time or arrival time. The test cases demonstrate the functionality of searching and sorting flights.