

Lab Assignment-12.5

Algorithms with AI Assistance – Sorting, Searching, and Optimizing Algorithms

Mohammed Awez Ali

2303A51767

Batch-12

Task Description #1 (Sorting – Merge Sort Implementation)

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

Prompt: #Generate a python code to create a function merge_sort(arr) that sorts a list in ascending order. Give time complexity and space complexity of the code in comments, use pydoc to generate documentation

Code:

```
> Users > chait > OneDrive > Desktop > AI Assistant coding > lab12.5.py > merge_sort
1 #Generate a python code to create a function merge_sort(arr) that sorts a list in ascending order. Give time complexity and space complexity of the code in comments, use pydoc to generate documentation
2 def merge_sort(arr):
3     """
4     This function implements the merge sort algorithm to sort a list in ascending order.
5     Merge sort is a divide-and-conquer algorithm that divides the input list into two halves,
6     recursively sorts both halves, and then merges the sorted halves back together.
7     Parameters:
8     arr (list): The list of elements to be sorted.
9     Returns:
10    list: A new list containing the sorted elements in ascending order.
11    Time Complexity: O(n log n) - Where n is the number of elements in the list. This is because the list is divided log n times and each merge operation takes O(n) time.
12    Space Complexity: O(n) - due to the additional space required for the temporary arrays used during the merge process.
13    Example:
14    >>> merge_sort([38, 27, 43, 3, 9, 82, 10])
15    [3, 9, 10, 27, 38, 43, 82]
16    >>> merge_sort([5, 2, 9, 1, 5, 6])
17    [1, 2, 5, 5, 6, 9]
18    >>> merge_sort([])
19    []
20    >>> merge_sort([1])
21    [1]
22    """
23    if len(arr) <= 1:
24        return arr
25    # Split the array into two halves
26    mid = len(arr) // 2
27    left_half = merge_sort(arr[:mid]) # Recursively sort the left half
28    right_half = merge_sort(arr[mid:]) # Recursively sort the right half
29    # Merge the sorted halves
30    return merge(left_half, right_half)
31 def merge(left, right):
32     """This helper function merges two sorted lists into a single sorted list.
33     Parameters:
34     left (list): The first sorted list.
35     right (list): The second sorted list.
36     Returns:
37     list: A merged sorted list containing all elements from both input lists.
38     """
39     merged = []
40     i = j = 0
41     # Merge the two lists while maintaining sorted order
42     while i < len(left) and j < len(right):
43         # Use the __lt__ comparison (or left[i].time < right[j].time)
44         if left[i] < right[j]:
45             merged.append(left[i]) # Append the smaller element from left
46             i += 1
47         else:
48             merged.append(right[j]) # Append the smaller element from right
49             j += 1
50     # If there are remaining elements in left, add them to merged
51     while i < len(left):
52         merged.append(left[i])
53         i += 1
54     # If there are remaining elements in right, add them to merged
55     while j < len(right):
56         merged.append(right[j])
57         j += 1
58     return merged # Return the merged sorted list
59 print("Sorted list:", merge_sort([38, 27, 43, 3, 9, 82, 10])) # Example usage of the merge_sort function to sort a list of numbers
60 print("Time complexity: O(n log n)") # Print the time complexity of the merge sort algorithm
61 print("Space complexity: O(n)") # Print the space complexity of the merge sort algorithm
62
```

Output:

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

Explanation: The code snippet implements a function to check if a given number is a palindrome. It converts the number to a string and compares it with its reverse to determine if it reads the same backward and forward. The function includes a docstring that explains its purpose, parameters, return value, and provides examples of usage.

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.

Prompt: #Generate a python code to create a function `binary_search(arr, target)` returning the index of the target or -1 if not found. Use docstring to explain best, average and worst case time complexity, use pydoc to generate documentation

Code:

```
65 #Generate a python code to create a function binary_search(arr, target) returning the index of the target or -1 if not found. Use docstring to explain best, average and worst case time complexity, use pydoc to generate documentation
66
67 def binary_search(arr, target):
68     """
69     This function implements the binary search algorithm to find the index of a target element in a sorted list.
70
71     Binary search works by repeatedly dividing the search interval in half. If the target value is less than the middle element,
72     the search continues in the left half; if it is greater, the search continues in the right half. This process continues until
73     the target value is found or the search interval is empty.
74
75     Parameters:
76     arr (list): A sorted list of elements to be searched.
77     target: The element to be searched for in the list.
78
79     Returns:
80     int: The index of the target element if found, or -1 if not found.
81
82     Time Complexity:
83     - Best Case: O(1) - when the target element is at the middle of the list.
84     - Average Case: O(log n) - where n is the number of elements in the list, due to halving the search space with each iteration.
85     - Worst Case: O(log n) - when the target element is not present in the list or is located at one of the ends.
86
87     Example:
88     >>> binary_search([1, 2, 3, 4, 5], 3)
89     2
90     >>> binary_search([1, 2, 3, 4, 5], 6)
91     -1
92     >>> binary_search([], 1)
93     -1
94     >>> binary_search([1], 1)
95     0
96     """
97     left, right = 0, len(arr) - 1
98
99     while left <= right:
100         mid = left + (right - left) // 2 # calculate the middle index
101
102         # Check if the target is present at mid
103         if arr[mid] == target:
104             return mid # Return the index if target is found
105         # If target is greater than mid, ignore left half
106         elif arr[mid] < target:
107             left = mid + 1 # Move left pointer to mid + 1
108         # If target is smaller than mid, ignore right half
109         else:
110             right = mid - 1 # Move right pointer to mid - 1
111
112     return -1 # Return -1 if target is not found in the list
113
114 print("Index of target 3:", binary_search([1, 2, 3, 4, 5], 3)) # Example usage of the binary_search function to find the index of target 3
115 print("Index of target 6:", binary_search([1, 2, 3, 4, 5], 6)) # Example usage of the binary_search function to find the index of target 6
116 print("Index of target 1 in empty list:", binary_search([], 1)) # Example usage of the binary_search function to find the index of target 1 in an empty list
117 print("Index of target 1 in single element list:", binary_search([1], 1)) # Example usage of the binary_search function to find the index of target 1 in a single element list
118
```

OutPut:

```
Index of target 3: 2
Index of target 6: -1
Index of target 1 in empty list: -1
Index of target 1 in single element list: 0
```

Explanation: The code snippet defines a function to calculate the factorial of a non-negative integer. It uses a loop to multiply the result by each integer from 1 to n, and includes a docstring that describes its functionality, parameters, return value, and provides examples of usage.

Task Description #3: Smart Healthcare Appointment Scheduling System

Prompt: #Generate a python code to implement Smart Healthcare Appointment Scheduling System. Give suitable searching and sorting algorithms to optimize the scheduling process. Use docstring to explain the functionality of each function, use pydoc to generate documentation

1.Search appointments using appointment ID.

2. Sort appointments based on time or consultation fee.

Code:

```
123 #Generate a python code to implement Smart Healthcare Appointment Scheduling System. Give suitable searching and sorting algorithms to optimize the scheduling process. Use docstring to explain the functionality of each function, use pydoc to generate documentation
124 # 1.Search appointments using appointment ID.
125 # 2. Sort appointments based on time or consultation fee.
126 class Appointment:
127     """
128     This class represents an appointment in the Smart Healthcare Appointment Scheduling System.
129     """
130     Attributes:
131     appointment_id (int): Unique identifier for the appointment.
132     patient_name (str): Name of the patient.
133     doctor_name (str): Name of the doctor.
134     time (str): Time of the appointment in HH:MM format.
135     consultation_fee (float): Consultation fee for the appointment.
136     """
137     def __init__(self, time, description):
138         self.time = time
139         self.description = description
140         # allow python to compare two appointments by their time
141     def __lt__(self, other):
142         return self.time < other.time
143 class AppointmentScheduler:
144     """
145     This class implements the Smart Healthcare Appointment Scheduling System.
146     It provides functionalities to add appointments, search for appointments by ID, and sort appointments based on time or consultation fee.
147     """
148     def __init__(self):
149         self.appointments = [] # Initialize an empty list to store appointments
150     def add_appointment(self, appointment):
151         """Adds a new appointment to the scheduler.
152         Parameters:
153         appointment (Appointment): The appointment object to be added to the scheduler.
154         """
155         self.appointments.append(appointment) # Add the appointment to the list of appointments
156     def search_appointment_by_id(self, appointment_id):
157         """Searches for an appointment by its ID using linear search.
158         Parameters:
159         appointment_id (int): The unique identifier of the appointment to be searched.
160         Returns:
161         Appointment: The appointment object if found, or None if not found.
162         """
163         for appointment in self.appointments: # Iterate through the list of appointments
164             if appointment.appointment_id == appointment_id: # Check if the current appointment's ID matches the target ID
165                 return appointment # Return the appointment if found
166         return None # Return None if no matching appointment is found
167
168     def sort_appointments_by_time(self):
169         """Sorts the appointments based on time using merge sort algorithm."""
170         self.appointments = merge_sort(self.appointments) # Sort the appointments using merge sort based on time attribute
171     def sort_appointments_by_fee(self):
172         """Sorts the appointments based on consultation fee using merge sort algorithm."""
173         self.appointments = sorted(self.appointments, key=lambda X: X.consultation_fee) # Sort the appointments based on consultation fee using built-in sorted function with a lambda key
174 # Example usage of the AppointmentScheduler class
175 scheduler = AppointmentScheduler() # Create an instance of the AppointmentScheduler class
176 scheduler.add_appointment(Appointment(1, "John Doe", "Dr. Smith", "10:00", 100.0)) # Add an appointment to the scheduler
177 scheduler.add_appointment(Appointment(2, "Jane Doe", "Dr. Brown", "11:00", 150.0)) # Add another appointment to the scheduler
178 scheduler.add_appointment(Appointment(3, "Alice", "Dr. Smith", "09:30", 120.0)) # Add another appointment to the scheduler
179 appointment = scheduler.search_appointment_by_id(2) # Search for an appointment by ID
180 if appointment:
181     print(f"Appointment found: (appointment.patient_name) with (appointment.doctor_name) at (appointment.time) for ${appointment.consultation_fee}") # Print the details of the found appointment
182 else:
183     print("Appointment not found.") # Print a message if the appointment is not found
184 scheduler.sort_appointments_by_time() # Sort the appointments by time
185 print("Appointments sorted by time:") # Print a message indicating that the appointments are sorted by time
186 for appointment in scheduler.appointments: # Iterate through the sorted appointments
187     print(f"({appointment.patient_name}) with (appointment.doctor_name) at (appointment.time) for ${appointment.consultation_fee}") # Print the details of each appointment sorted by time
188 scheduler.sort_appointments_by_fee() # Sort the appointments by consultation fee
189 print("Appointments sorted by consultation fee:") # Print a message indicating that the appointments are sorted by consultation fee
190 for appointment in scheduler.appointments: # Iterate through the sorted appointments
191     print(f"({appointment.patient_name}) with (appointment.doctor_name) at (appointment.time) for ${appointment.consultation_fee}") # Print the details of each appointment sorted by consultation fee
192
193
194
195
```

OutPut:

```
Appointment found: Jane Doe with Dr. Brown at 11:00 for a fee of 150.0  
Ticket found (Appointment ID: 11-3-2024)
```

Explanation: The code snippet implements a Smart Healthcare Appointment Scheduling System. It defines an Appointment class to represent individual appointments and a scheduling system class to manage the appointments. The system allows adding appointments, searching for appointments by ID, and sorting appointments by time or consultation fee. Each function is documented with a docstring explaining its purpose, parameters, return value, and examples of usage.

Task Description #4: Railway Ticket Reservation System

Prompt: #Generate a python code to implement Railway Ticket Reservation System. Identify suitable algorithms for searching and sorting to optimize the reservation process. Use docstring to explain the functionality of each function, use pydoc to generate documentation

1. Search tickets using ticket ID.

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

Code:

```
211 #Generate a python code to implement Railway Ticket Reservation System. Identify suitable algorithms for searching and sorting to optimize the reservation process. Use docstring to explain the functionality of each function, use pydoc to generate documentation
212 # 1. Search tickets using ticket ID.
213 # 2. Sort bookings based on travel date or seat number.
214 class Ticket:
215     """
216     This class represents a ticket in the Railway Ticket Reservation System.
217
218     Attributes:
219     ticket_id (int): Unique identifier for the ticket.
220     passenger_name (str): Name of the passenger.
221     train_number (str): Number of the train.
222     travel_date (str): Date of travel in YYYY-MM-DD format.
223     seat_number (str): Seat number assigned to the passenger.
224     """
225     def __init__(self, ticket_id, passenger_name, train_number, travel_date, seat_number):
226         self.ticket_id = ticket_id
227         self.passenger_name = passenger_name
228         self.train_number = train_number
229         self.travel_date = travel_date
230         self.seat_number = seat_number
231 class TicketReservationSystem:
232     """
233     This class implements the Railway Ticket Reservation System.
234
235     It provides functionalities to add tickets, search for tickets by ID, and sort bookings based on travel date or seat number.
236     """
237     def __init__(self):
238         self.tickets = [] # Initialize an empty list to store tickets
239
240     def add_ticket(self, ticket):
241         """Adds a new ticket to the reservation system.
242
243         Parameters:
244         ticket (Ticket): The ticket object to be added to the reservation system.
245         """
246         self.tickets.append(ticket) # Add the ticket to the list of tickets
247
248     def search_ticket_by_id(self, ticket_id):
249         """Searches for a ticket by its ID using linear search.
250
251         Parameters:
252         ticket_id (int): The unique identifier of the ticket to be searched.
253
254         Returns:
255         Ticket: The ticket object if found, or None if not found.
256         """
257         for ticket in self.tickets: # Iterate through the list of tickets
258             if ticket.ticket_id == ticket_id: # Check if the current ticket's ID matches the target ID
259                 return ticket # Return the ticket if found
260         return None # Return None if no matching ticket is found
261
262     def sort_tickets_by_travel_date(self):
263         """Sorts the tickets based on travel date using merge sort algorithm."""
264         self.tickets = merge_sort(self.tickets) # Sort the tickets using merge sort based on travel date attribute
265
266     def sort_tickets_by_seat_number(self):
267         """Sorts the tickets based on seat number using merge sort algorithm."""
268         self.tickets = sorted(self.tickets, key=lambda x: x.seat_number) # Sort the tickets based on seat number using built-in sorted function with a lambda key
269 # Example usage of the TicketReservationSystem class
270 reservation_system = TicketReservationSystem() # Create an instance of the TicketReservationSystem class
271 reservation_system.add_ticket(Ticket(1, "John Doe", "12345", "2024-07-01", "A1")) # Add a ticket to the reservation system
272 reservation_system.add_ticket(Ticket(2, "Jane Doe", "12345", "2024-07-01", "A2")) # Add another ticket to the reservation system
273 reservation_system.add_ticket(Ticket(3, "Alice", "54321", "2024-07-02", "B1")) # Add another ticket to the reservation system
274 ticket = reservation_system.search_ticket_by_id(2) # Search for a ticket by ID
275 if ticket:
276     print(f"Ticket found: {ticket.passenger_name} on train {ticket.train_number} for travel date {ticket.travel_date} at seat {ticket.seat_number}") # Print the details of the found ticket
277 else:
278     print("Ticket not found.") # Print a message if the ticket is not found
279 reservation_system.sort_tickets_by_travel_date() # Sort the tickets by travel date
280 print("Tickets sorted by travel date:") # Print a message indicating that the tickets are sorted by travel date
281 for ticket in reservation_system.tickets: # Iterate through the sorted tickets
282     print(f"{ticket.passenger_name} on train {ticket.train_number} for travel date {ticket.travel_date} at seat {ticket.seat_number}") # Print the details of each ticket sorted by travel date
283 reservation_system.sort_tickets_by_seat_number() # Sort the tickets by seat number
284 print("Tickets sorted by seat number:") # Print a message indicating that the tickets are sorted by seat number
285 for ticket in reservation_system.tickets: # Iterate through the sorted tickets
286     print(f"{ticket.passenger_name} on train {ticket.train_number} for travel date {ticket.travel_date} at seat {ticket.seat_number}") # Print the details of each ticket sorted by seat number
287
288
289
290
```

OutPut:

```
Ticket found: Jane Doe on train 12345 for travel date 2024-07-01 at seat A2
```

Explanation: The code snippet implements a Railway Ticket Reservation System. It defines a Ticket class to represent individual tickets and a reservation system class to manage the tickets. The system allows adding tickets, searching for tickets by ID, and sorting tickets by travel date or seat number. Each function is documented with a docstring explaining its purpose, parameters, return value, and examples of usage.

Task Description #5: Smart Hostel Room Allocation System

Prompt: #Generate a python code to implement Smart Hostel Room Allocation System. Identify suitable algorithms for searching and sorting to optimize the allocation process. Use docstring to explain the functionality of each function, use pydoc to generate documentation

1. Search allocation details using student ID.

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

Code:

```
324 #Generate a python code to implement Smart Hostel Room Allocation System. Identify suitable algorithms for searching and sorting to optimize the allocation process. Use docstring to explain the functionality of each function, use pydoc to generate documentation
325 # 1. Search allocation details using student ID.
326 # 2. Sort records based on room number or allocation date.
327
328 class RoomAllocation:
329     """
330     This class represents a room allocation in the Smart Hostel Room Allocation System.
331     """
332     Attributes:
333     allocation_id (int): Unique identifier for the room allocation.
334     student_name (str): Name of the student.
335     student_id (str): ID of the student.
336     room_number (str): Number of the allocated room.
337     allocation_date (str): Date of allocation in YYYY-MM-DD format.
338     """
339     def __init__(self, allocation_id, student_name, student_id, room_number, allocation_date):
340         self.allocation_id = allocation_id
341         self.student_name = student_name
342         self.student_id = student_id
343         self.room_number = room_number
344         self.allocation_date = allocation_date
345
346 class HostelRoomAllocationSystem:
347     """
348     This class implements the Smart Hostel Room Allocation System.
349     It provides functionalities to add room allocations, search for allocation details by student ID, and sort records based on room number or allocation date.
350     """
351     def __init__(self):
352         self.allocations = [] # Initialize an empty list to store room allocations
353
354     def add_allocation(self, allocation):
355         """Adds a new room allocation to the system.
356
357         Parameters:
358         allocation (RoomAllocation): The room allocation object to be added to the system.
359         """
360         self.allocations.append(allocation) # Add the room allocation to the list of allocations
361
362     def search_allocation_by_student_id(self, student_id):
363         """Searches for room allocation details by student ID using linear search.
364
365         Parameters:
366         student_id (str): The ID of the student whose allocation details are to be searched.
367
368         Returns:
369         RoomAllocation: The room allocation object if found, or None if not found.
370         """
371         for allocation in self.allocations: # Iterate through the list of allocations
372             if allocation.student_id == student_id: # Check if the current allocation's student ID matches the target ID
373                 return allocation # Return the allocation if found
374         return None # Return None if no matching allocation is found
375
376     def sort_allocations_by_room_number(self):
377         """Sorts the room allocations based on room number using merge sort algorithm."""
378         self.allocations = merge_sort(self.allocations) # Sort the allocations using merge sort based on room number attribute
379
380     def sort_allocations_by_date(self):
381         """Sorts the room allocations based on allocation date using merge sort algorithm."""
382         self.allocations = sorted(self.allocations, key=lambda x: x.allocation_date) # Sort the allocations based on allocation date using built-in sorted function with a lambda key
383
384 # Example usage of the HostelRoomAllocationSystem class
385 allocation_system = HostelRoomAllocationSystem() # Create an instance of the HostelRoomAllocationSystem class
386 allocation_system.add_allocation(RoomAllocation(1, "John Doe", "S123", "101", "2024-07-01")) # Add a room allocation to the system
387 allocation_system.add_allocation(RoomAllocation(2, "Jane Doe", "S124", "102", "2024-07-02")) # Add another room allocation to the system
388 allocation_system.add_allocation(RoomAllocation(3, "Alice", "S125", "101", "2024-07-03")) # Add another room allocation to the system
389 allocation = allocation_system.search_allocation_by_student_id("S124") # Search for room allocation details by student ID
390 if allocation:
391     print(f"Allocation found: {allocation.student_name} in room {allocation.room_number} allocated on {allocation.allocation_date}") # Print the details of the found allocation
392 else:
393     print("Allocation not found.") # Print a message if the allocation is not found
394 allocation_system.sort_allocations_by_room_number() # Sort the room allocations by room number
395 print("Room allocations sorted by room number:") # Print a message indicating that the room allocations are sorted by room number
396 for allocation in allocation_system.allocations: # Iterate through the sorted allocations
397     print(f"{allocation.student_name} in room {allocation.room_number} allocated on {allocation.allocation_date}") # Print the details of each allocation sorted by room number
398 allocation_system.sort_allocations_by_date() # Sort the room allocations by allocation date
399 print("Room allocations sorted by allocation date:") # Print a message indicating that the room allocations are sorted by allocation date
400 for allocation in allocation_system.allocations: # Iterate through the sorted allocations
401     print(f"{allocation.student_name} in room {allocation.room_number} allocated on {allocation.allocation_date}") # Print the details of each allocation sorted by allocation date
402
```

OutPut:

```
Allocation found: Jane Doe in room 102 allocated on 2024-07-02
```

Explanation: The code snippet implements a Smart Hostel Room Allocation System. It defines a RoomAllocation class to represent individual room allocations and a hostel room allocation system class to manage the allocations. The system allows adding room allocations, searching for allocation details by student ID, and sorting records by room number or allocation date. Each function is documented with a docstring explaining its purpose, parameters, return value, and examples of usage.

Task Description #6: Online Movie Streaming Platform

Prompt:

#Generate a python code to implement Online Movie Streaming Platform. Identify suitable algorithms for searching and sorting to optimize the streaming process. Use docstring to explain the functionality of each function, use pydoc to generate documentation

1. Search movies by movie ID.

2. Sort movies based on rating or release year.

Code:

```

307
308 #Generate a python code to implement Online Movie Streaming Platform. Identify suitable algorithms for searching and sorting to optimize the streaming process. Use docstring to explain the functionality of each function, use pydoc to generate documentation
309 # 1. Search movies by movie ID.
310 # 2. Sort movies based on rating or release year.
311 class Movie:
312     """
313     This class represents a movie in the Online Movie Streaming Platform.
314     Attributes:
315     movie_id (int): Unique identifier for the movie.
316     title (str): Title of the movie.
317     director (str): Director of the movie.
318     release_year (int): Year the movie was released.
319     rating (float): Rating of the movie on a scale of 1 to 10.
320     """
321     def __init__(self, movie_id, title, director, release_year, rating):
322         self.movie_id = movie_id
323         self.title = title
324         self.director = director
325         self.release_year = release_year
326         self.rating = rating
327
328 class MovieStreamingPlatform:
329     """
330     This class implements the Online Movie Streaming Platform.
331     It provides functionalities to add movies, search for movies by ID, and sort movies based on rating or release year.
332     """
333     def __init__(self):
334         self.movies = [] # Initialize an empty list to store movies
335
336     def add_movie(self, movie):
337         """Adds a new movie to the streaming platform.
338         Parameters:
339         movie (Movie): The movie object to be added to the streaming platform.
340         """
341         self.movies.append(movie) # Add the movie to the list of movies
342
343     def search_movie_by_id(self, movie_id):
344         """Searches for a movie by its ID using linear search.
345         Parameters:
346         movie_id (int): The Unique Identifier of the movie to be searched.
347         Returns:
348         Movie: The movie object if found, or None if not found.
349         """
350         for movie in self.movies: # Iterate through the list of movies
351             if movie.movie_id == movie_id: # Check if the current movie's ID matches the target ID
352                 return movie # Return the movie if found
353         return None # Return None if no matching movie is found
354
355     def sort_movies_by_rating(self):
356         """Sorts the movies based on rating using merge sort algorithm."""
357         self.movies = merge_sort(self.movies) # Sort the movies using merge sort based on rating attribute
358
359     def sort_movies_by_release_year(self):
360         """Sorts the movies based on release year using merge sort algorithm."""
361         self.movies = sorted(self.movies, key=lambda X: X.release_year) # Sort the movies based on release year using built-in sorted function with a lambda key
362
363 # Example usage of the MovieStreamingPlatform class
364 streaming_platform = MovieStreamingPlatform() # Create an instance of the MovieStreamingPlatform class
365 streaming_platform.add_movie(Movie(1, "Inception", "Christopher Nolan", 2010, 8.8)) # Add a movie to the streaming platform
366 streaming_platform.add_movie(Movie(2, "The Matrix", "Lana Wachowski, Lilly Wachowski", 1999, 8.7)) # Add another movie to the streaming platform
367 streaming_platform.add_movie(Movie(3, "Interstellar", "Christopher Nolan", 2014, 8.6)) # Add another movie to the streaming platform
368 movie = streaming_platform.search_movie_by_id(2) # Search for a movie by ID
369 if movie:
370     print(f"Movie found: {movie.title} directed by {movie.director} released in {movie.release_year} with rating {movie.rating}") # Print the details of the found movie
371 else:
372     print("Movie not found.") # Print a message if the movie is not found
373 streaming_platform.sort_movies_by_rating() # Sort the movies by rating
374 print("Movies sorted by rating:") # Print a message indicating that the movies are sorted by rating
375 for movie in streaming_platform.movies: # Iterate through the sorted movies
376     print(f"{movie.title} directed by {movie.director} released in {movie.release_year} with rating {movie.rating}") # Print the details of each movie sorted by rating
377 streaming_platform.sort_movies_by_release_year() # Sort the movies by release year
378 print("Movies sorted by release year:") # Print a message indicating that the movies are sorted by release year
379 for movie in streaming_platform.movies: # Iterate through the sorted movies
380     print(f"{movie.title} directed by {movie.director} released in {movie.release_year} with rating {movie.rating}") # Print the details of each movie sorted by release year
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461

```

OutPut:

```

Movie found: The Matrix directed by Lana Wachowski, Lilly Wachowski released in 1999 with rating 8.7

```

Explanation: The code snippet implements an Online Movie Streaming Platform. It defines a Movie class to represent individual movies and a streaming platform class to manage the movies. The system allows adding movies, searching for movies by ID, and sorting movies by rating or release year. Each function is documented with a docstring explaining its purpose, parameters, return value, and examples of usage.

Task Description #7: Smart Agriculture Crop Monitoring System

Prompt: #Generate a python code to implement Smart Agriculture Crop Monitoring System. Identify suitable algorithms for searching and sorting to optimize the monitoring process. Use docstring to explain the functionality of each function, use pydoc to generate documentation

1. Search crop details using crop ID.

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

Code:

```
465 """Generate a python code to implement Smart Agriculture Crop Monitoring System. Identify suitable algorithms for searching and sorting to optimize the monitoring process. Use docstring to explain the functionality of each function, use pydoc to generate documentation
466 # 1. Search crop details using crop ID.
467 # 2. Sort crops based on moisture level or yield estimate.
468 """
469 class Crop:
470     """
471     This class represents a crop in the Smart Agriculture Crop Monitoring System.
472
473     Attributes:
474     crop_id (int): Unique identifier for the crop.
475     crop_name (str): Name of the crop.
476     moisture_level (float): Current moisture level of the crop in percentage.
477     yield_estimate (float): Estimated yield of the crop in tons per hectare.
478     """
479     def __init__(self, crop_id, crop_name, moisture_level, yield_estimate):
480         self.crop_id = crop_id
481         self.crop_name = crop_name
482         self.moisture_level = moisture_level
483         self.yield_estimate = yield_estimate
484
485 class CropMonitoringSystem:
486     """
487     This class implements the Smart Agriculture Crop Monitoring System.
488
489     It provides functionalities to add crops, search for crop details by crop ID, and sort crops based on moisture level or yield estimate.
490     """
491     def __init__(self):
492         self.crops = [] # Initialize an empty list to store crops
493
494     def add_crop(self, crop):
495         """
496         Add a new crop to the monitoring system.
497
498         Parameters:
499         crop (Crop): The crop object to be added to the monitoring system.
500         """
501         self.crops.append(crop) # Add the crop to the list of crops
502
503     def search_crop_by_id(self, crop_id):
504         """
505         Searches for crop details by crop ID using linear search.
506
507         Parameters:
508         crop_id (int): The unique identifier of the crop to be searched.
509
510         Returns:
511         Crop: The crop object if found, or None if not found.
512         """
513         for crop in self.crops: # Iterate through the list of crops
514             if crop.crop_id == crop_id: # Check if the current crop's ID matches the target ID
515                 return crop # Return the crop if found
516         return None # Return None if no matching crop is found
517
518     def sort_crops_by_moisture_level(self):
519         """
520         Sorts the crops based on moisture level using merge sort algorithm.
521         """
522         self.crops = merge_sort(self.crops) # Sort the crops using merge sort based on moisture level attribute
523
524     def sort_crops_by_yield_estimate(self):
525         """
526         Sorts the crops based on yield estimate using merge sort algorithm.
527         """
528         self.crops = sorted(self.crops, key=lambda x: x.yield_estimate) # Sort the crops based on yield estimate using built-in sorted function with a lambda key
529
530 # Example usage of the CropMonitoringSystem class
531 monitoring_system = CropMonitoringSystem() # Create an instance of the CropMonitoringSystem class
532 monitoring_system.add_crop(Crop(1, "Wheat", 30.5, 3.2)) # Add a crop to the monitoring system
533 monitoring_system.add_crop(Crop(2, "Corn", 25.0, 4.5)) # Add another crop to the monitoring system
534 monitoring_system.add_crop(Crop(3, "Rice", 40.0, 5.0)) # Add another crop to the monitoring system
535 crop = monitoring_system.search_crop_by_id(2) # Search for crop details by crop ID
536 if crop:
537     print(f"Crop found: {crop.crop_name} with moisture level {crop.moisture_level}% and yield estimate {crop.yield_estimate} tons/ha") # Print the details of the found crop
538 else:
539     print("Crop not found.") # Print a message if the crop is not found
540 monitoring_system.sort_crops_by_moisture_level() # Sort the crops by moisture level
541 print("Crops sorted by moisture level:") # Print a message indicating that the crops are sorted by moisture level
542 for crop in monitoring_system.crops: # Iterate through the sorted crops
543     print(f"{crop.crop_name} with moisture level {crop.moisture_level}% and yield estimate {crop.yield_estimate} tons/ha") # Print the details of each crop sorted by moisture level
544 monitoring_system.sort_crops_by_yield_estimate() # Sort the crops by yield estimate
545 print("Crops sorted by yield estimate:") # Print a message indicating that the crops are sorted by yield estimate
546 for crop in monitoring_system.crops: # Iterate through the sorted crops
547     print(f"{crop.crop_name} with moisture level {crop.moisture_level}% and yield estimate {crop.yield_estimate} tons/ha") # Print the details of each crop sorted by yield estimate
548
549
550
551
552
553
554
555
```

OutPut:

```
Crop found: Corn with moisture level 25.0% and yield estimate 4.5 tons/ha
```

Explanation: The code snippet implements a Smart Agriculture Crop Monitoring System. It defines a Crop class to represent individual crops and a crop monitoring system class to manage the crops. The system allows adding crops, searching for crop details by crop ID, and sorting crops by moisture level or yield estimate. Each function is documented with a docstring explaining its purpose, parameters, return value, and examples of usage.

Task Description #8: Airport Flight Management System

Prompt: #Generate a python code to implement Airport Flight Management System. Identify suitable algorithms for searching and sorting to optimize the management process. Use docstring to explain the functionality of each function, use pydoc to generate documentation

1. Search flight details using flight ID.

2. Sort flights based on departure time or destination.

Code:

```
548 """Generate a python code to implement Airport Flight Management System. Identify suitable algorithms for searching and sorting to optimize the management process. Use docstring to explain the functionality of each function, use pydoc to generate documentation
549 # 1. Search flight details using flight ID.
550 # 2. Sort flights based on departure time or destination.
551 """
552 class Flight:
553     """
554     This class represents a flight in the Airport Flight Management System.
555
556     Attributes:
557     flight_id (int): Unique identifier for the flight.
558     airline (str): Name of the airline operating the flight.
559     departure_time (str): Departure time of the flight in HH:MM format.
560     destination (str): Destination of the flight.
561     """
562     def __init__(self, flight_id, airline, departure_time, destination):
563         self.flight_id = flight_id
564         self.airline = airline
565         self.departure_time = departure_time
566         self.destination = destination
567
568 class FlightManagementSystem:
569     """
570     This class implements the Airport Flight Management System.
571
572     It provides functionalities to add flights, search for flight details by flight ID, and sort flights based on departure time or destination.
573     """
574     def __init__(self):
575         self.flights = [] # Initialize an empty list to store flights
576
577     def add_flight(self, flight):
578         """Add a new flight to the management system.
579
580         Parameters:
581         flight (Flight): The flight object to be added to the management system.
582         """
583         self.flights.append(flight) # Add the flight to the list of flights
584
585     def search_flight_by_id(self, flight_id):
586         """Searches for flight details by flight ID using linear search.
587
588         Parameters:
589         flight_id (int): The unique identifier of the flight to be searched.
590
591         Returns:
592         Flight: The flight object if found, or None if not found.
593         """
594         for flight in self.flights: # Iterate through the list of flights
595             if flight.flight_id == flight_id: # Check if the current flight's ID matches the target ID
596                 return flight # Return the flight if found
597         return None # Return None if no matching flight is found
598
599     def sort_flights_by_departure_time(self):
600         """Sorts the flights based on departure time using merge sort algorithm."""
601         self.flights = merge_sort(self.flights) # Sort the flights using merge sort based on departure time attribute
602
603     def sort_flights_by_destination(self):
604         """Sorts the flights based on destination using merge sort algorithm."""
605         self.flights = sorted(self.flights, key=lambda x: x.destination) # Sort the flights based on destination using built-in sorted function with a lambda key
606
607 # Example usage of the FlightManagementSystem class
608 management_system = FlightManagementSystem() # Create an instance of the FlightManagementSystem class
609 management_system.add_flight(Flight(1, "Airline A", "10:00", "New York")) # Add a flight to the management system
610 management_system.add_flight(Flight(2, "Airline B", "12:00", "Los Angeles")) # Add another flight to the management system
611 management_system.add_flight(Flight(3, "Airline C", "09:00", "Chicago")) # Add another flight to the management system
612 flight = management_system.search_flight_by_id(2) # Search for flight details by flight ID
613 if flight:
614     print(f"Flight found: {flight.airline} departing at {flight.departure_time} to {flight.destination}") # Print the details of the found flight
615 else:
616     print("Flight not found.") # Print a message if the flight is not found
617 management_system.sort_flights_by_departure_time() # Sort the flights by departure time
618 print("Flights sorted by departure time:") # Print a message indicating that the flights are sorted by departure time
619 for flight in management_system.flights: # Iterate through the sorted flights
620     print(f"{flight.airline} departing at {flight.departure_time} to {flight.destination}") # Print the details of each flight sorted by departure time
621 management_system.sort_flights_by_destination() # Sort the flights by destination
622 print("Flights sorted by destination:") # Print a message indicating that the flights are sorted by destination
623 for flight in management_system.flights: # Iterate through the sorted flights
624     print(f"{flight.airline} departing at {flight.departure_time} to {flight.destination}") # Print the details of each flight sorted by destination
625
```

OutPut:

```
548 """Generate a python code to implement Airport Flight Management System. Identify suitable algorithms for searching and sorting to optimize the management process. Use docstring to explain the functionality of each function, use pydoc to generate documentation
549 # 1. Search flight details using flight ID.
550 # 2. Sort flights based on departure time or destination.
551 """
552 class Flight:
553     """
554     This class represents a flight in the Airport Flight Management System.
555
556     Attributes:
557     flight_id (int): Unique identifier for the flight.
558     airline (str): Name of the airline operating the flight.
559     departure_time (str): Departure time of the flight in HH:MM format.
560     destination (str): Destination of the flight.
561     """
562     def __init__(self, flight_id, airline, departure_time, destination):
563         self.flight_id = flight_id
564         self.airline = airline
565         self.departure_time = departure_time
566         self.destination = destination
567
568 class FlightManagementSystem:
569     """
570     This class implements the Airport Flight Management System.
571
572     It provides functionalities to add flights, search for flight details by flight ID, and sort flights based on departure time or destination.
573     """
574     def __init__(self):
575         self.flights = [] # Initialize an empty list to store flights
576
577     def add_flight(self, flight):
578         """Add a new flight to the management system.
579
580         Parameters:
581         flight (Flight): The flight object to be added to the management system.
582         """
583         self.flights.append(flight) # Add the flight to the list of flights
584
585     def search_flight_by_id(self, flight_id):
586         """Searches for flight details by flight ID using linear search.
587
588         Parameters:
589         flight_id (int): The unique identifier of the flight to be searched.
590
591         Returns:
592         Flight: The flight object if found, or None if not found.
593         """
594         for flight in self.flights: # Iterate through the list of flights
595             if flight.flight_id == flight_id: # Check if the current flight's ID matches the target ID
596                 return flight # Return the flight if found
597         return None # Return None if no matching flight is found
598
599     def sort_flights_by_departure_time(self):
600         """Sorts the flights based on departure time using merge sort algorithm."""
601         self.flights = merge_sort(self.flights) # Sort the flights using merge sort based on departure time attribute
602
603     def sort_flights_by_destination(self):
604         """Sorts the flights based on destination using merge sort algorithm."""
605         self.flights = sorted(self.flights, key=lambda x: x.destination) # Sort the flights based on destination using built-in sorted function with a lambda key
606
607 # Example usage of the FlightManagementSystem class
608 management_system = FlightManagementSystem() # Create an instance of the FlightManagementSystem class
609 management_system.add_flight(Flight(1, "Airline A", "10:00", "New York")) # Add a flight to the management system
610 management_system.add_flight(Flight(2, "Airline B", "12:00", "Los Angeles")) # Add another flight to the management system
611 management_system.add_flight(Flight(3, "Airline C", "09:00", "Chicago")) # Add another flight to the management system
612 flight = management_system.search_flight_by_id(2) # Search for flight details by flight ID
613 if flight:
614     print(f"Flight found: {flight.airline} departing at {flight.departure_time} to {flight.destination}") # Print the details of the found flight
615 else:
616     print("Flight not found.") # Print a message if the flight is not found
617 management_system.sort_flights_by_departure_time() # Sort the flights by departure time
618 print("Flights sorted by departure time:") # Print a message indicating that the flights are sorted by departure time
619 for flight in management_system.flights: # Iterate through the sorted flights
620     print(f"{flight.airline} departing at {flight.departure_time} to {flight.destination}") # Print the details of each flight sorted by departure time
621 management_system.sort_flights_by_destination() # Sort the flights by destination
622 print("Flights sorted by destination:") # Print a message indicating that the flights are sorted by destination
623 for flight in management_system.flights: # Iterate through the sorted flights
624     print(f"{flight.airline} departing at {flight.departure_time} to {flight.destination}") # Print the details of each flight sorted by destination
625
```

Flight found: Airline B departing at 12:00 to Los Angeles

Explanation: The code snippet implements an Airport Flight Management System. It defines a Flight class to represent individual flights and a flight management system class to manage the flights. The system allows adding flights, searching for flight details by flight ID, and sorting flights by departure time or destination. Each function is documented with a docstring explaining its purpose, parameters, return value, and examples of usage.