

## **Lab Assignment-12.5**

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

**Name:** G.DILEEP KUMAR

**Ht.no:** 2303A51722

**Batch:** 11

### **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:

```
> User > chari > OneDrive > Desktop > AI Assistant coding > lab125.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                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
24        if len(arr) <= 1:
25            return arr
26        # Split the array into two halves
27        mid = len(arr) // 2
28        left_half = merge_sort(arr[:mid]) # Recursively sort the left half
29        right_half = merge_sort(arr[mid:]) # Recursively sort the right half
30        # Merge the sorted halves
31        return merge(left_half, right_half)
32    def merge(left, right):
33        """This helper function merges two sorted lists into a single sorted list.
34        Parameters:
35            left (list): The first sorted list.
36            right (list): The second sorted list.
37            Returns:
38                list: A merged sorted list containing all elements from both input lists.
39                """
40        merged = []
41        i = j = 0
42        # Merge the two lists while maintaining sorted order
43        while i < len(left) and j < len(right):
44            # use the __lt__ comparison (or left[i].time < right[j].time)
45            if left[i] < right[j]:
46                merged.append(left[i]) # Append the smaller element from left
47                i += 1
48            else:
49                merged.append(right[j]) # Append the smaller element from right
50                j += 1
51        # If there are remaining elements in left, add them to merged
52        while i < len(left):
53            merged.append(left[i])
54            i += 1
55        # If there are remaining elements in right, add them to merged
56        while j < len(right):
57            merged.append(right[j])
58            j += 1
59        return merged # return the merged sorted list
60 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
61 print("Time complexity: O(n log n)") # Print the time complexity of the merge sort algorithm
62 print("Space Complexity: O(n)") # Print the space complexity of the merge sort algorithm
63
```

## 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:

```
66  #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
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
98  left, right = 0, len(arr) - 1
99
100 while left <= right:
101     mid = left + (right - left) // 2 # calculate the middle index
102
103     if arr[mid] == target:
104         return mid # return the index if target is found
105     elif arr[mid] < target:
106         left = mid + 1 # Move left pointer to mid + 1
107     elif arr[mid] > target:
108         right = mid - 1 # Move right pointer to mid - 1
109
110
111 return -1 # return -1 if target is not found in the list
112 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
113 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
114 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
115 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
116
117
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:

```

15  # 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
16  #
17  # 1. Search appointments using appointment ID.
18  # 2. Sort appointments based on time or consultation fee.
19
20  class Appointment:
21
22      """This class represents an appointment in the Smart Healthcare Appointment Scheduling system.
23
24      Attributes:
25          appointment_id (int): Unique identifier for the appointment.
26          patient_name (str): Name of the patient.
27          doctor_name (str): Name of the doctor.
28          time (str): Time of the appointment in HH:MM format.
29          consultation_fee (float): Consultation fee for the appointment.
30
31      """
32
33      def __init__(self, time, description):
34          self.time = time
35          self.description = description
36          self.time = self.time[:5] + ":" + self.time[5:]
37
38      def __lt__(self, other):
39          return self.time < other.time
40
41  class AppointmentScheduler:
42
43      """This class implements the Smart Healthcare Appointment Scheduling System.
44
45      It provides functionalities to add appointments, search for appointments by ID, and sort appointments based on time or consultation fee.
46      """
47
48      def __init__(self):
49          self.appointments = [] # Initialize an empty list to store appointments
50
51      def add_appointment(self, appointment):
52          """Adds a new appointment to the scheduler.
53
54          Parameters:
55              appointment (Appointment): The appointment object to be added to the scheduler.
56
57          """
58          self.appointments.append(appointment)
59
60      def search_appointment_by_id(self, appointment_id):
61          """Searches for an appointment by its ID using linear search.
62
63          Parameters:
64              appointment_id (int): The unique identifier of the appointment to be searched.
65
66          Returns:
67              Appointment: The appointment object if found, or None if not found.
68
69          """
70
71          for appointment in self.appointments: # Iterate through the list of appointments
72              if appointment.appointment_id == appointment_id: # Check if the current appointment's ID matches the target ID
73                  return appointment # Return the appointment if found
74
75          return None # Return None if no matching appointment is found
76
77
78      def sort_appointments_by_time(self):
79          """Sorts the appointments based on time using merge sort algorithm."""
80          self.appointments = merge_sort(self.appointments) # Sort the appointments using merge sort based on time attribute
81
82      def sort_appointments_by_fee(self):
83          """Sorts the appointments based on consultation fee using merge sort algorithm."""
84          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
85
86      # Example usage of the AppointmentScheduler class
87      scheduler = AppointmentScheduler() # Create an instance of the AppointmentScheduler class
88      scheduler.add_appointment(appointment(1, "Jane Doe", "Dr. Smith", "09:00", 100.0)) # Add an appointment to the scheduler
89      scheduler.add_appointment(appointment(2, "Jane Doe", "Dr. Brown", "11:00", 150.0)) # Add another appointment to the scheduler
90      scheduler.add_appointment(appointment(3, "Alice", "Dr. Smith", "09:30", 120.0)) # Add another appointment to the scheduler
91      appointment = scheduler.search_appointment_by_id(2) # Search for an appointment by ID
92
93      if appointment:
94          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
95      else:
96          print("Appointment not found.") # Print a message if the appointment is not found
97
98      scheduler.sort_appointments_by_time() # Sort the appointments by time
99      for appointment in scheduler.appointments: # Iterate through the sorted appointments
100          print(f"({appointment.patient_name}, {appointment.doctor_name}) at {appointment.time} for ${appointment.consultation_fee}") # Print the details of each appointment sorted by time
101
102      scheduler.sort_appointments_by_fee() # Sort the appointments by consultation fee
103      for appointment in scheduler.appointments: # Iterate through the sorted appointments
104          print(f"({appointment.patient_name}, {appointment.doctor_name}) at {appointment.time} for ${appointment.consultation_fee}") # Print the details of each appointment sorted by consultation fee
105
106      print("Appointments sorted by consultation fee:") # Print a message indicating that the appointments are sorted by consultation fee
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205

```

**OutPut:**

```
Appointment found: Jane Doe with Dr. Brown at 11:00 for a fee of 150.0
```

**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
215 class Ticket:
216
217     """This class represents a ticket in the Railway Ticket Reservation System.
218
219     Attributes:
220         ticket_id (int): Unique identifier for the ticket.
221         passenger_name (str): Name of the passenger.
222         train_number (str): Number of the train.
223         travel_date (str): Date of travel in YYYY-MM-DD format.
224         seat_number (str): Seat number assigned to the passenger.
225     """
226
227     def __init__(self, ticket_id, passenger_name, train_number, travel_date, seat_number):
228         self.ticket_id = ticket_id
229         self.passenger_name = passenger_name
230         self.train_number = train_number
231         self.travel_date = travel_date
232         self.seat_number = seat_number
233
234 class TicketReservationSystem:
235
236     """This class implements the Railway Ticket Reservation System.
237
238     It provides functionalities to add tickets, search for tickets by ID, and sort bookings based on travel date or seat number.
239     """
240
241     def __init__(self):
242         self.tickets = [] # Initialize an empty list to store tickets
243
244     def add_ticket(self, ticket):
245         """Adds a new ticket to the reservation system.
246
247         Parameters:
248             ticket (Ticket): The ticket object to be added to the reservation system.
249             ...
250         self.tickets.append(ticket) # Add the ticket to the list of tickets
251
252     def search_ticket_by_id(self, ticket_id):
253         """Searches for a ticket by its ID using linear search.
254
255         Parameters:
256             ticket_id (int): The unique identifier of the ticket to be searched.
257
258         Returns:
259             Ticket: The ticket object if found, or None if not found.
260             ...
261         for ticket in self.tickets: # Iterate through the list of tickets
262             if ticket.ticket_id == ticket_id: # Check if the current ticket's ID matches the target ID
263                 return ticket # Return the ticket if found
264         return None # Return None if no matching ticket is found
265
266     def sort_tickets_by_travel_date(self):
267         """Sorts the tickets based on travel date using merge sort algorithm."""
268         self.tickets = merge_sort(self.tickets) # Sort the tickets using merge sort based on travel date attribute
269
270     def sort_tickets_by_seat_number(self):
271         """Sorts the tickets based on seat number using merge sort algorithm."""
272         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
273
274 # Example usage of the TicketReservationSystem class
275 reservation_system = TicketReservationSystem() # Create an instance of the TicketReservationSystem class
276 reservation_system.add_ticket(Ticket(1, "John Doe", "12345", "2024-07-01", "A1")) # Add a ticket to the reservation system
277 reservation_system.add_ticket(Ticket(2, "Jane Doe", "12345", "2024-07-01", "A2")) # Add another ticket to the reservation system
278 reservation_system.add_ticket(Ticket(3, "Alice", "54321", "2024-07-02", "B1")) # Add another ticket to the reservation system
279 ticket = reservation_system.search_ticket_by_id(2) # Search for a ticket by ID
280
281 if ticket:
282     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
283 else:
284     print("Ticket not found.") # Print a message if the ticket is not found
285 reservation_system.sort_tickets_by_travel_date() # Sort the tickets by travel date
286 print("Tickets sorted by travel date:") # Print a message indicating that the tickets are sorted by travel date
287 for ticket in reservation_system.tickets: # Iterate through the sorted tickets
288     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
289 reservation_system.sort_tickets_by_seat_number() # Sort the tickets by seat number
290 print("Tickets sorted by seat number:") # Print a message indicating that the tickets are sorted by seat number
291 for ticket in reservation_system.tickets: # Iterate through the sorted tickets
292     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
293
294
295
296
297
298
299
```

## 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:

```

294     """
295     Generates 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
296     # 1. Search allocation details using student ID
297     # 2. Sort records based on room number or allocation date.
298     class RoomAllocation:
299         """
300             This class represents a room allocation in the Smart Hostel Room Allocation System.
301
302             Attributes:
303                 allocation_id (int): Unique identifier for the room allocation.
304                 student_name (str): Name of the student.
305                 student_id (str): ID of the student.
306                 room_number (str): Number of the allocated room.
307                 allocation_date (str): Date of allocation in YYYY-MM-DD format.
308
309             def __init__(self, allocation_id, student_name, student_id, room_number, allocation_date):
310                 self.allocation_id = allocation_id
311                 self.student_name = student_name
312                 self.student_id = student_id
313                 self.room_number = room_number
314                 self.allocation_date = allocation_date
315
316         class HostelRoomAllocationSystem:
317             """
318                 This class implements the Smart Hostel Room Allocation System.
319
320                 It provides functionalities to add room allocations, search for allocation details by student ID, and sort records based on room number or allocation date.
321
322             def __init__(self):
323                 self.allocations = [] # Initialize an empty list to store room allocations
324
325             def add_allocation(self, allocation):
326                 """
327                     Adds a new room allocation to the system.
328
329                     Parameters:
330                         allocation (RoomAllocation): The room allocation object to be added to the system.
331
332                     self.allocations.append(allocation) # Add the room allocation to the list of allocations
333
334             def search_allocation_by_student_id(self, student_id):
335                 """
336                     Searches for room allocation details by student ID using linear search.
337
338                     Parameters:
339                         student_id (str): The ID of the student whose allocation details are to be searched.
340
341                     Returns:
342                         RoomAllocation: The room allocation object if found, or None if not found.
343
344                     for allocation in self.allocations: # Iterate through the list of allocations
345                         if allocation.student_id == student_id: # Check if the current allocation's student ID matches the target ID
346                             return allocation # Return the allocation if found
347
348                     return None # Return None if no matching allocation is found
349
350             def sort_allocations_by_room_number(self):
351                 """
352                     Sorts the room allocations based on room number using merge sort algorithm.
353                     self.allocations = merge_sort(self.allocations) # Sort the allocations using merge sort based on room number attribute
354
355             def sort_allocations_by_date(self):
356                 """
357                     Sorts the room allocations based on allocation date using merge sort algorithm.
358                     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
359
360             # Example usage of the HostelRoomAllocationSystem class
361             allocation_system = HostelRoomAllocationSystem() # Create an instance of the HostelRoomAllocationSystem class
362             allocation_system.add_allocation(RoomAllocation(1, "John Doe", "S123", "101", "2024-07-01")) # Add a room allocation to the system
363             allocation_system.add_allocation(RoomAllocation(2, "Jane Doe", "S124", "102", "2024-07-02")) # Add another room allocation to the system
364             allocation_system.add_allocation(RoomAllocation(3, "Alice", "S125", "101", "2024-07-03")) # Add another room allocation to the system
365             allocation = allocation_system.search_allocation_by_student_id("S124") # Search for room allocation details by student ID
366
367             if allocation:
368                 print("Allocation found: [allocation.student_name] in room [allocation.room_number] allocated on [allocation.allocation_date]") # Print the details of the found allocation
369             else:
370                 print("Allocation not found.") # Print a message if the allocation is not found
371
372             allocation_system.sort_allocations_by_room_number() # Sort the room allocations by room number
373             print("Room allocations sorted by room number:") # Print a message indicating that the room allocations are sorted by room number
374
375             for allocation in allocation_system.allocations: # Iterate through the sorted allocations
376                 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
377
378             allocation_system.sort_allocations_by_date() # Sort the room allocations by allocation date
379             print("Room allocations sorted by allocation date:") # Print a message indicating that the room allocations are sorted by allocation date
380
381             for allocation in allocation_system.allocations: # Iterate through the sorted allocations
382                 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
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
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
789

```

## 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:**

```

381 #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
382 # i. search movies by movie ID.
383 # ii. sort movies based on rating or release year.
384 class Movie:
385     """
386         This class represents a movie in the Online Movie Streaming Platform.
387
388     Attributes:
389         movie_id (int): Unique identifier for the movie.
390         title (str): Title of the movie.
391         director (str): Director of the movie.
392         release_year (int): Year the movie was released.
393         rating (float): Rating of the movie on a scale of 1 to 10.
394
395     def __init__(self, movie_id, title, director, release_year, rating):
396         self.movie_id = movie_id
397         self.title = title
398         self.director = director
399         self.release_year = release_year
400         self.rating = rating
401
402 class MovieStreamingPlatform:
403     """
404         This class implements the Online Movie Streaming Platform.
405
406         It provides functionalities to add movies, search for movies by ID, and sort movies based on rating or release year.
407
408     def __init__(self):
409         self.movies = [] # Initialize an empty list to store movies
410
411     def add_movie(self, movie):
412         """
413             Adds a new movie to the streaming platform.
414
415             Parameters:
416                 movie (Movie): The movie object to be added to the streaming platform.
417
418             self.movies.append(movie) # Add the movie to the list of movies
419
420     def search_movie_by_id(self, movie_id):
421         """
422             Searches for a movie by its ID using linear search.
423
424             Parameters:
425                 movie_id (int): The unique identifier of the movie to be searched.
426
427             Returns:
428                 Movie: The movie object if found, or None if not found.
429
430             for movie in self.movies: # Iterate through the list of movies
431                 if movie.movie_id == movie_id: # Check if the current movie's ID matches the target ID
432                     return movie # Return the movie if found
433
434             return None # Return None if no matching movie is found
435
436     def sort_movies_by_rating(self):
437         """
438             Sorts the movies based on rating using merge sort algorithm.
439             self.movies = merge_sort(self.movies) # Sort the movies using merge sort based on rating attribute
440
441     def sort_movies_by_release_year(self):
442         """
443             Sorts the movies based on release year using merge sort algorithm.
444             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
445
446 # Example usage of the MovieStreamingPlatform class
447 streaming_platform = MovieStreamingPlatform() # Create an instance of the MovieStreamingPlatform class
448 streaming_platform.add_movie(Movie(1, "Inception", "Christopher Nolan", 2010, 8.8)) # Add a movie to the streaming platform
449 streaming_platform.add_movie(Movie(2, "The Matrix", "Lana Wachowski, Lilly Wachowski", 1999, 8.7)) # Add another movie to the streaming platform
450 streaming_platform.add_movie(Movie(3, "Interstellar", "Christopher Nolan", 2014, 8.6)) # Add another movie to the streaming platform
451 movie = streaming_platform.search_movie_by_id(2) # Search for a movie by ID
452
453 if movie:
454     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
455 else:
456     print("Movie not found.") # Print a message if the movie is not found
457
458 streaming_platform.sort_movies_by_rating() # Sort the movies by rating
459 print("Movies sorted by rating:") # Print a message indicating that the movies are sorted by rating
460 for movie in streaming_platform.movies: # Iterate through the sorted movies
461     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
462
463 streaming_platform.sort_movies_by_release_year() # Sort the movies by release year
464 print("Movies sorted by release year:") # Print a message indicating that the movies are sorted by release year
465 for movie in streaming_platform.movies: # Iterate through the sorted movies
466     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
467
468
469
470
471

```

## 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 # Generates 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 #
467 # 1. Search crops details using crop ID.
468 # 2. Sort crops based on moisture level or yield estimate.
469
470 class Crop:
471     """
472         This class represents a crop in the Smart Agriculture Crop Monitoring System.
473
474     Attributes:
475         crop_id (int): Unique identifier for the crop.
476         crop_name (str): Name of the crop.
477         moisture_level (float): Current moisture level of the crop in percentage.
478         yield_estimate (float): Estimated yield of the crop in tons per hectare.
479
480     """
481
482     def __init__(self, crop_id, crop_name, moisture_level, yield_estimate):
483         self.crop_id = crop_id
484         self.crop_name = crop_name
485         self.moisture_level = moisture_level
486         self.yield_estimate = yield_estimate
487
488 class CropMonitoringSystem:
489     """
490         This class implements the Smart Agriculture Crop Monitoring System.
491
492     It provides functionalities to add crops, search for crop details by crop ID, and sort crops based on moisture level or yield estimate.
493     """
494
495     def __init__(self):
496         self.crops = [] # Initialize an empty list to store crops
497
498     def add_crop(self, crop):
499         """Adds a new crop to the monitoring system.
500
501             Parameters:
502                 crop (Crop): The crop object to be added to the monitoring system.
503             """
504         self.crops.append(crop) # Add the crop to the list of crops
505
506     def search_crop_by_id(self, crop_id):
507         """Searches for crop details by crop ID using linear search.
508
509             Parameters:
510                 crop_id (int): The unique identifier of the crop to be searched.
511
512             Returns:
513                 Crop: The crop object if found, or None if not found.
514             """
515
516         for crop in self.crops: # Iterate through the list of crops
517             if crop.crop_id == crop_id: # Check if current crop's ID matches the target ID
518                 return crop # Return the crop if found
519
520         return None # Return None if no matching crop is found
521
522
523     def sort_crops_by_moisture_level(self):
524         """Sorts the crops based on moisture level using merge sort algorithm."""
525
526         self.crops = merge_sort(self.crops) # Sort the crops using merge sort based on moisture level attribute
527
528     def sort_crops_by_yield_estimate(self):
529         """Sorts the crops based on yield estimate using merge sort algorithm."""
530
531         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
532
533 # Example usage of the CropMonitoringSystem class
534
535 monitoring_system = CropMonitoringSystem() # Create an instance of the CropMonitoringSystem class
536 monitoring_system.add_crop(Crop(1, "Wheat", 30.5, 3.2)) # Add a crop to the monitoring system
537 monitoring_system.add_crop(Crop(2, "Corn", 25.0, 4.5)) # Add another crop to the monitoring system
538 monitoring_system.add_crop(Crop(3, "Rice", 40.0, 5.0)) # Add another crop to the monitoring system
539
540 crop = monitoring_system.search_crop_by_id(2) # Search for crop details by crop ID
541
542 if crop:
543     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
544 else:
545     print("Crop not found.") # Print a message if the crop is not found
546
547 monitoring_system.sort_crops_by_moisture_level() # Sort the crops by moisture level
548 print("Crops sorted by moisture level!") # Print a message indicating that the crops are sorted by moisture level
549
550 for crop in monitoring_system.crops: # Iterate through the sorted crops
551     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
552
553 monitoring_system.sort_crops_by_yield_estimate() # Sort the crops by yield estimate
554 print("Crops sorted by yield estimate!") # Print a message indicating that the crops are sorted by yield estimate
555
556 for crop in monitoring_system.crops: # Iterate through the sorted crops
557     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
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
999

```

## **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:

```
649 # 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
650 # 1. Search flight details using Flight ID.
651 # 2. Sort flights based on departure time or destination.
652 class Flight:
653     """
654         This class represents a flight in the Airport Flight Management System.
655     """
656     Attributes:
657     flight_id (int): Unique identifier for the flight.
658     airline (str): Name of the airline operating the flight.
659     departure_time (str): Departure time of the flight in HHMM format.
660     destination (str): Destination of the flight.
661     """
662     def __init__(self, flight_id, airline, departure_time, destination):
663         self.flight_id = flight_id
664         self.airline = airline
665         self.departure_time = departure_time
666         self.destination = destination
667     class FlightManagementSystem:
668         """
669             This class implements the Airport Flight Management System.
670         """
671         It provides functionalities to add flights, search for flight details by flight ID, and sort flights based on departure time or destination.
672         """
673         def __init__(self):
674             self.flights = []
675         def add_flight(self, flight):
676             """
677                 Adds a new flight to the management system.
678             """
679             Parameters:
680             flight (Flight): The flight object to be added to the management system.
681             """
682             self.flights.append(flight) # Add the flight to the list of flights
683         def search_flight_by_id(self, flight_id):
684             """
685                 Searches for flight details by flight ID using linear search.
686             """
687             Parameters:
688             flight_id (int): The unique identifier of the flight to be searched.
689             """
690             Returns:
691             flight: The flight object if found, None if not found.
692             """
693             for flight in self.flights: # Iterate through the list of flights
694                 if flight.flight_id == flight_id: # Check if the current flight's ID matches the target ID
695                     return flight # Return the flight if found
696             return None # Return None if no matching flight is found
697         def sort_flights_by_departure_time(self):
698             """
699                 Sorts the flights based on departure time using merge sort algorithm.
700             """
701             self.flights = merge_sort(self.flights) # Sort the flights using merge sort based on departure time attribute
702         def sort_flights_by_destination(self):
703             """
704                 Sorts the flights based on destination using merge sort algorithm.
705             """
706             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
707         # Example usage of the FlightManagementSystem class
708         management_system = FlightManagementSystem() # Create an instance of the FlightManagementSystem class
709         management_system.add_flight(Flight(1, "Airline A", "10:00", "New York")) # Add a flight to the management system
710         management_system.add_flight(Flight(2, "Airline B", "12:00", "Los Angeles")) # Add another flight to the management system
711         management_system.add_flight(Flight(3, "Airline C", "09:00", "Chicago")) # Add another flight to the management system
712         flight = management_system.search_flight_by_id(2) # Search for flight details by flight ID
713         if flight:
714             print(f"Flight found: {flight.airline} departing at {flight.departure_time} to {flight.destination}") # Print the details of the found flight
715         else:
716             print("Flight not found.") # Print a message if the flight is not found
717         management_system.sort_flights_by_departure_time() # Sort the flights by departure time
718         print("Flights sorted by departure time:") # Print a message indicating that the flights are sorted by departure time
719         for flight in management_system.flights: # Iterate through the sorted flights
720             print(f"{flight.airline} departing at {flight.departure_time} to {flight.destination}") # Print the details of each flight sorted by departure time
721         management_system.sort_flights_by_destination() # Sort the flights by destination
722         print("Flights sorted by destination:") # Print a message indicating that the flights are sorted by destination
723         for flight in management_system.flights: # Iterate through the sorted flights
724             print(f"{flight.airline} departing at {flight.departure_time} to {flight.destination}") # Print the details of each flight sorted by destination
725
```

## OutPut:

```
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.