**ASSIGNMENT\_12.5**

# HT.NO:2303A51909

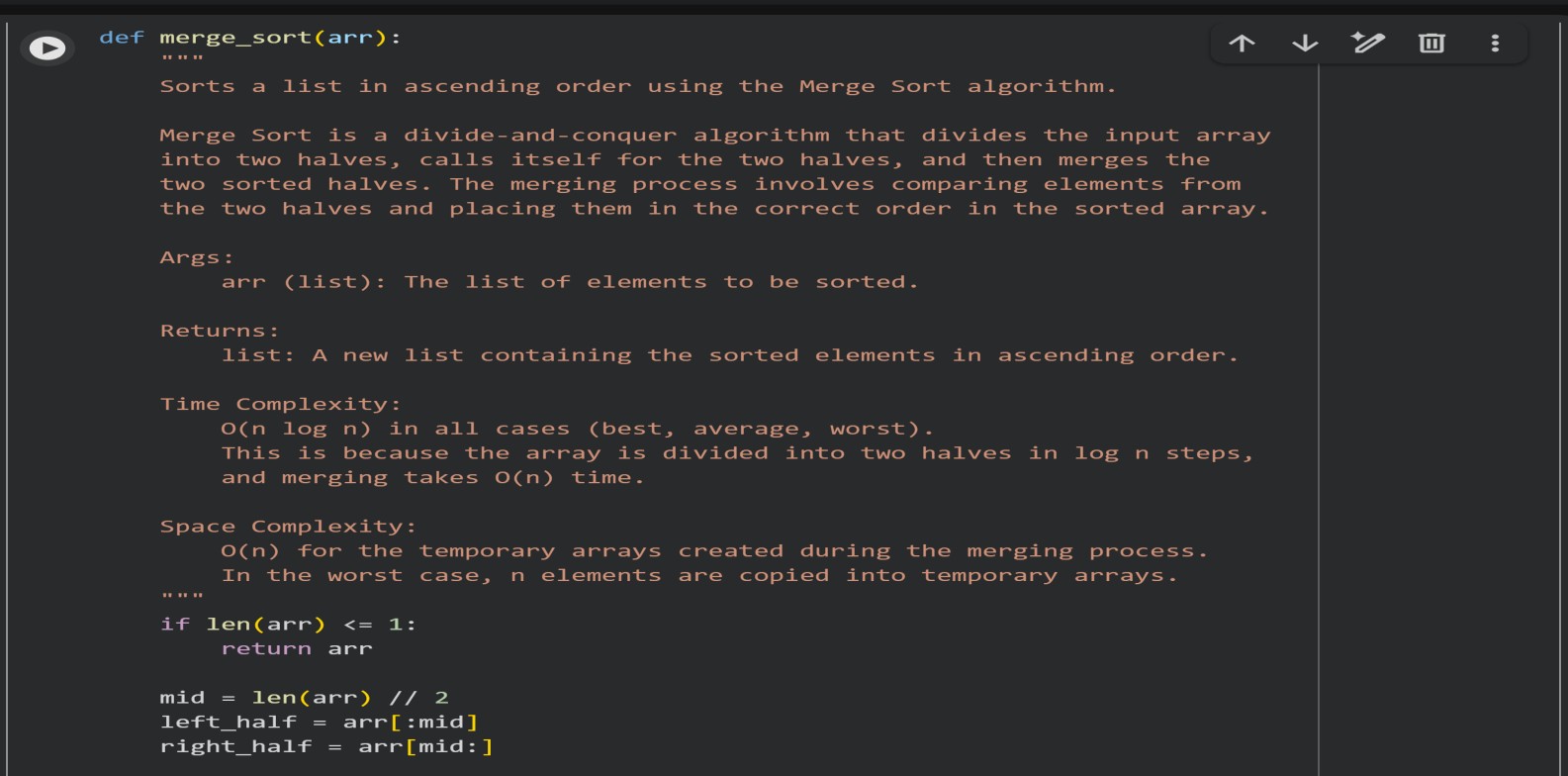
**Task Description #1 (Sorting – Merge Sort Implementation) •** Task: Use AI to generate a Python program that implements the Merge Sort algorithm.

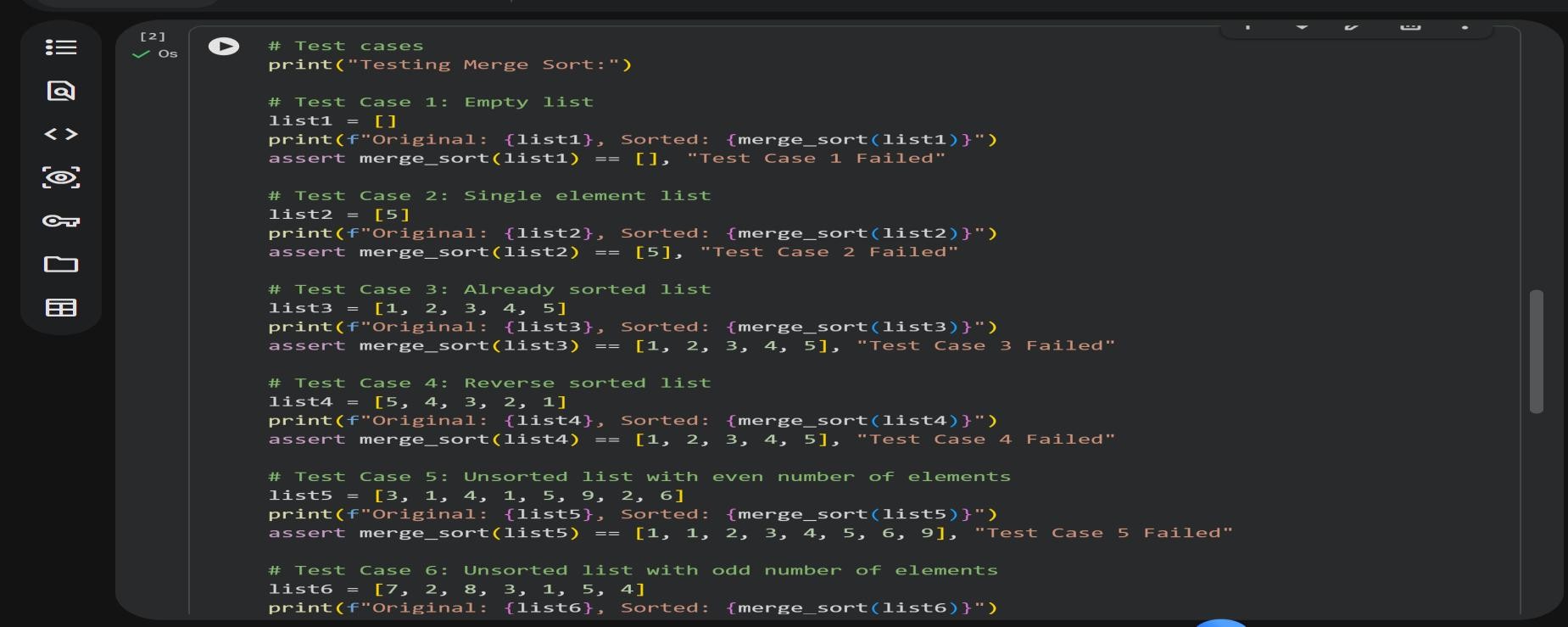
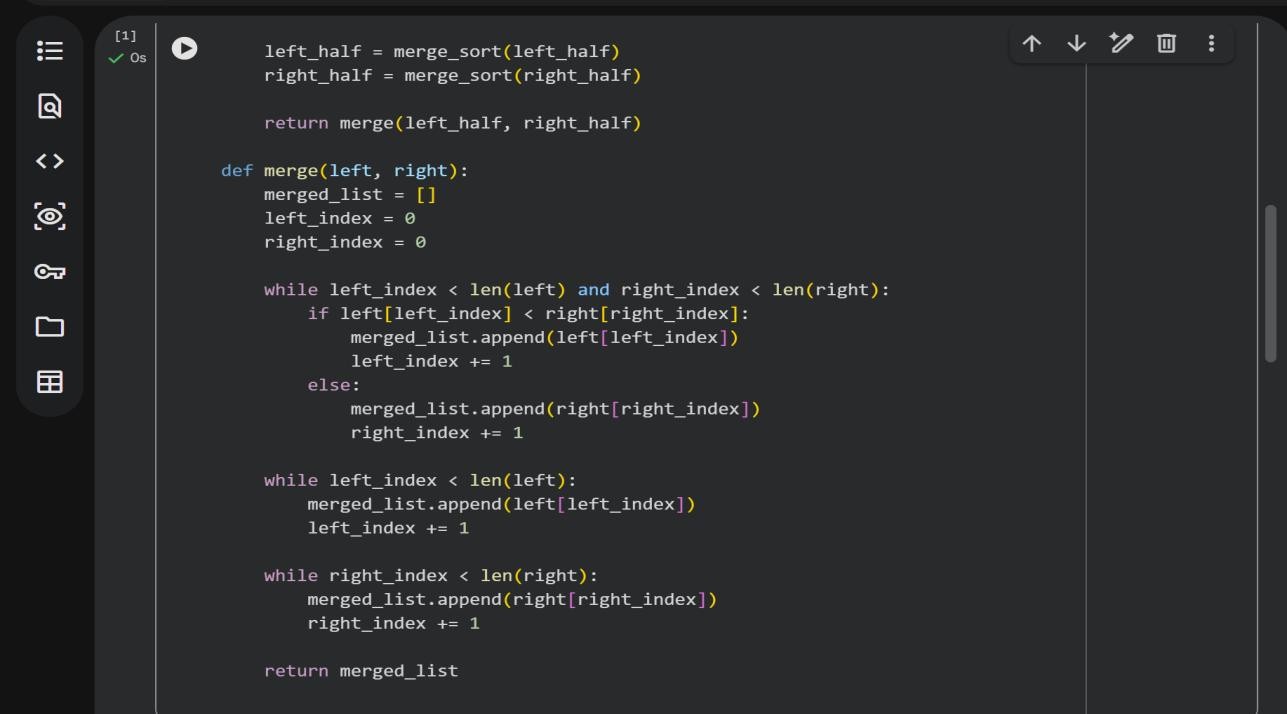
• Instructions: o Prompt AI to create a function merge\_sort(arr) that sorts a list in ascending order.

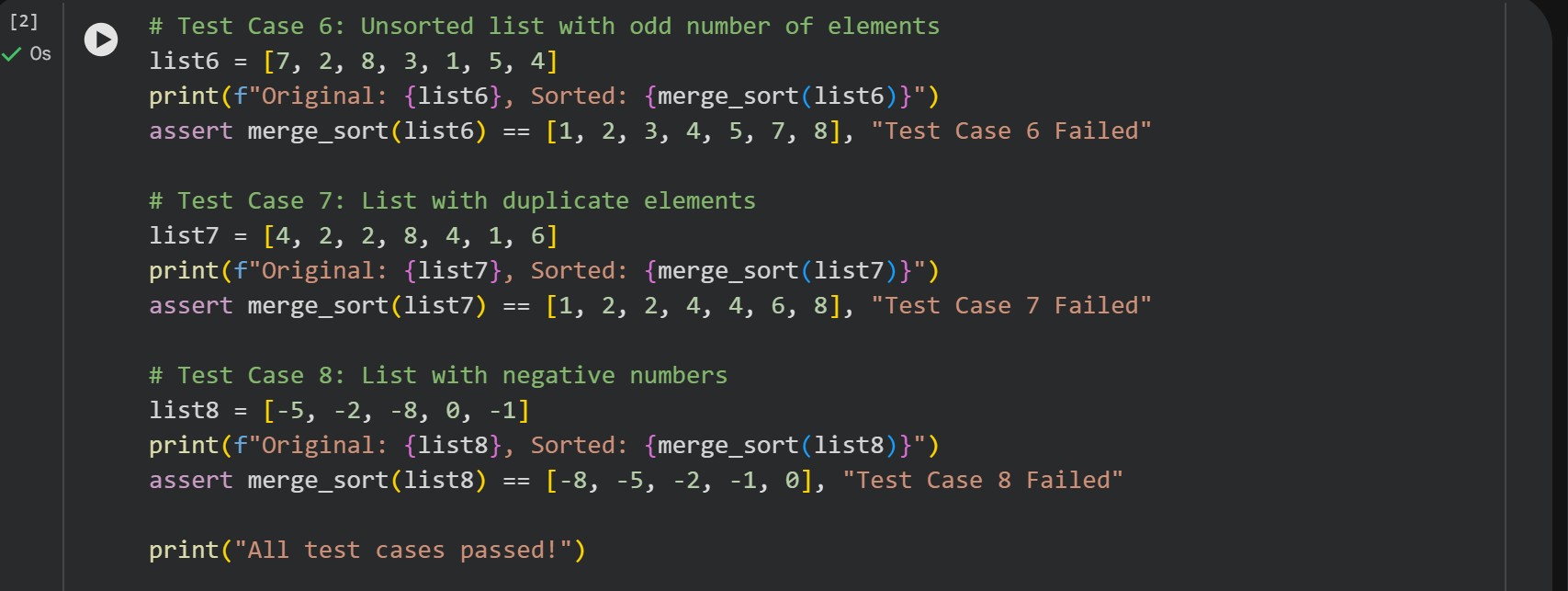
o Ask AI to include time complexity and space complexity in the function docstring. o Verify the generated code with test cases.

* Expected Output: o A functional Python script implementing Merge Sort with proper documentation.

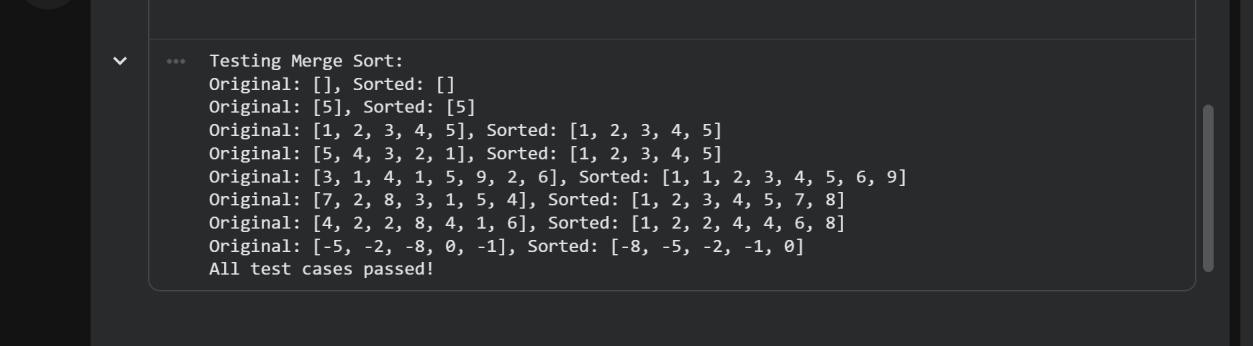
**CODE:**







**OUTPUT:**



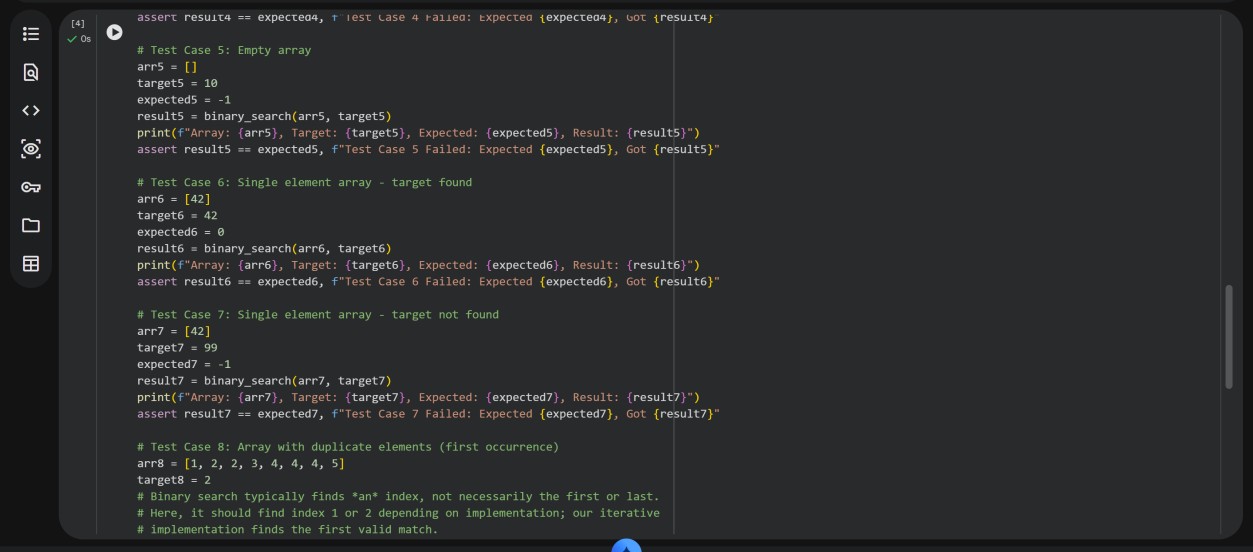
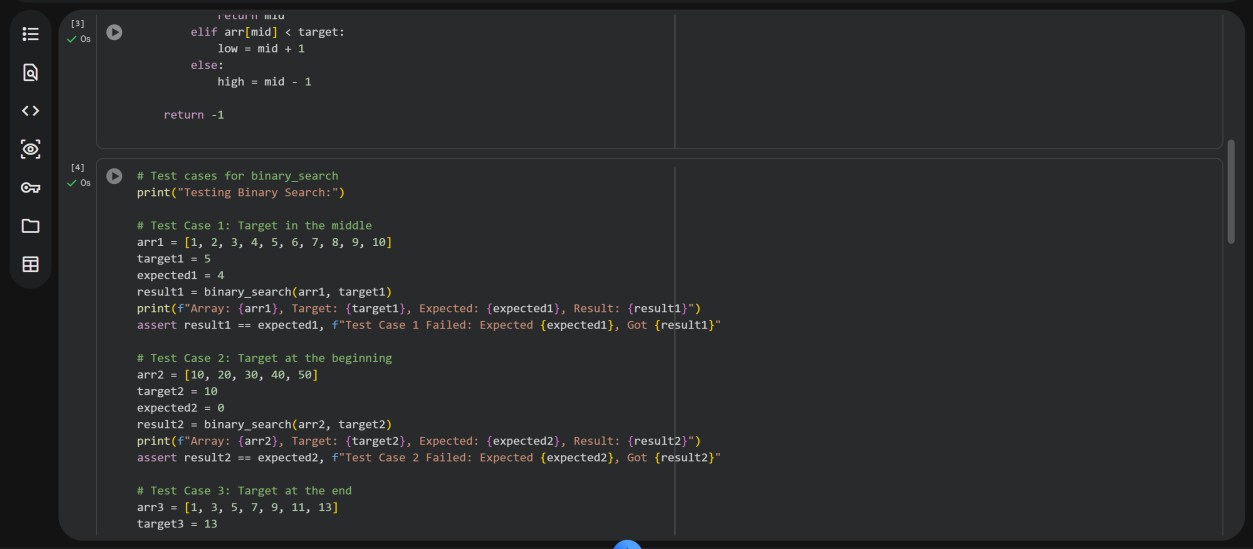
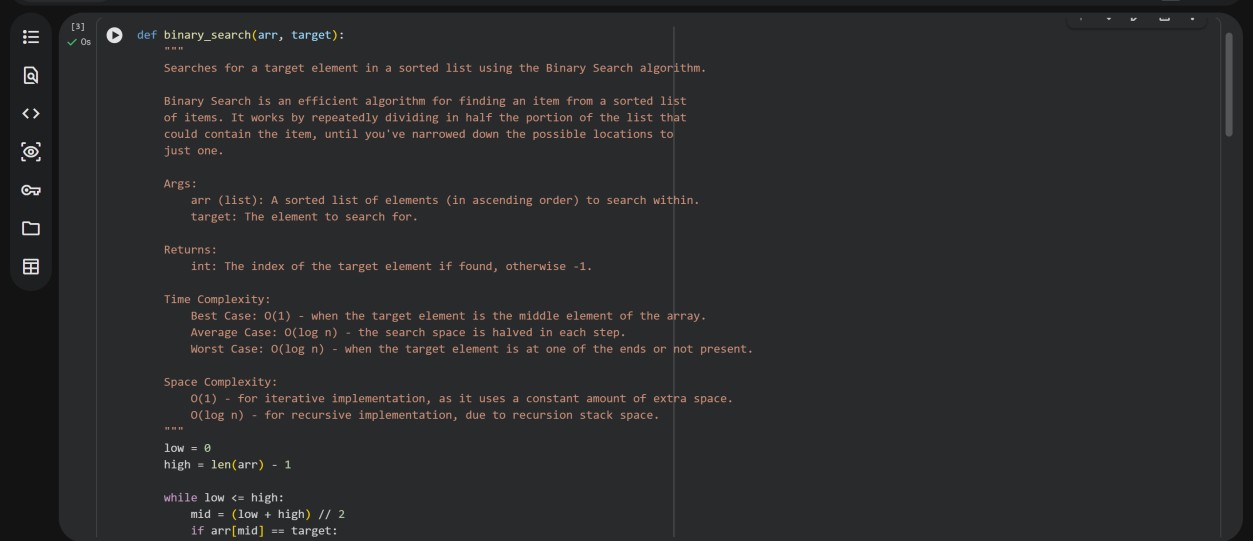
**Task Description #2 (Searching – Binary Search with AI**

Optimization)

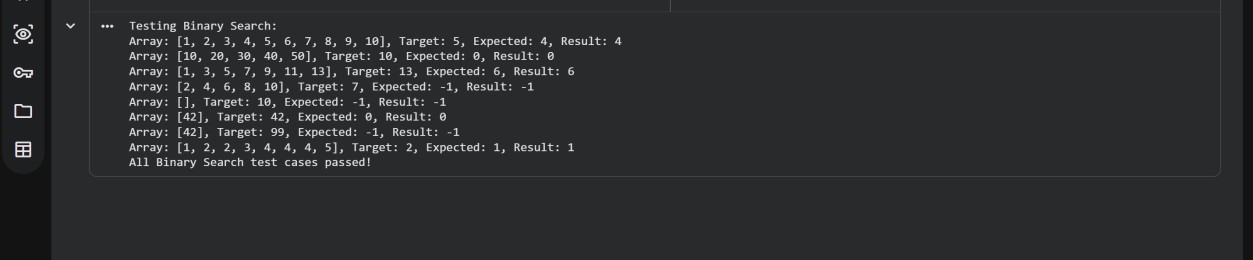
* Task: Use AI to create a binary search function that finds a target element in a sorted list.
* Instructions:
* Prompt AI to create a function binary\_search(arr, target) returning the index of the target or -1 if not found.
* Include docstrings explaining best, average, and worst-case complexities. o Test with various inputs.

• Expected Output: o Python code implementing binary search with AI- generated comments and docstrings.

CODE:



**OUTPUT:**



**Task Description #3: Smart Healthcare Appointment Scheduling**

System

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

1. Search appointments using appointment ID.
2. Sort appointments based on time or consultation fee.

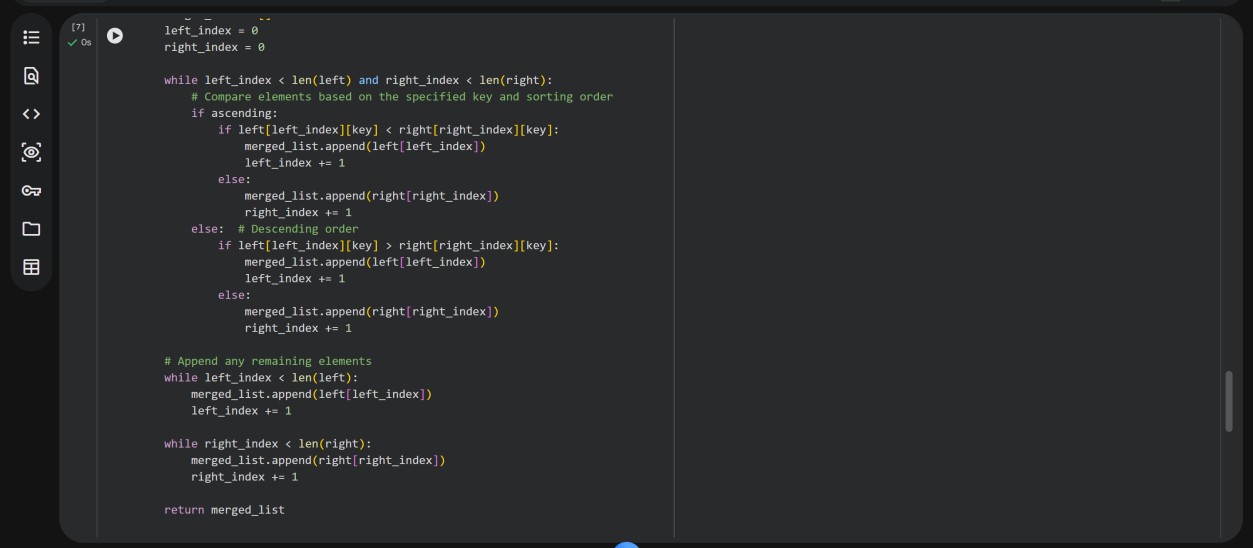
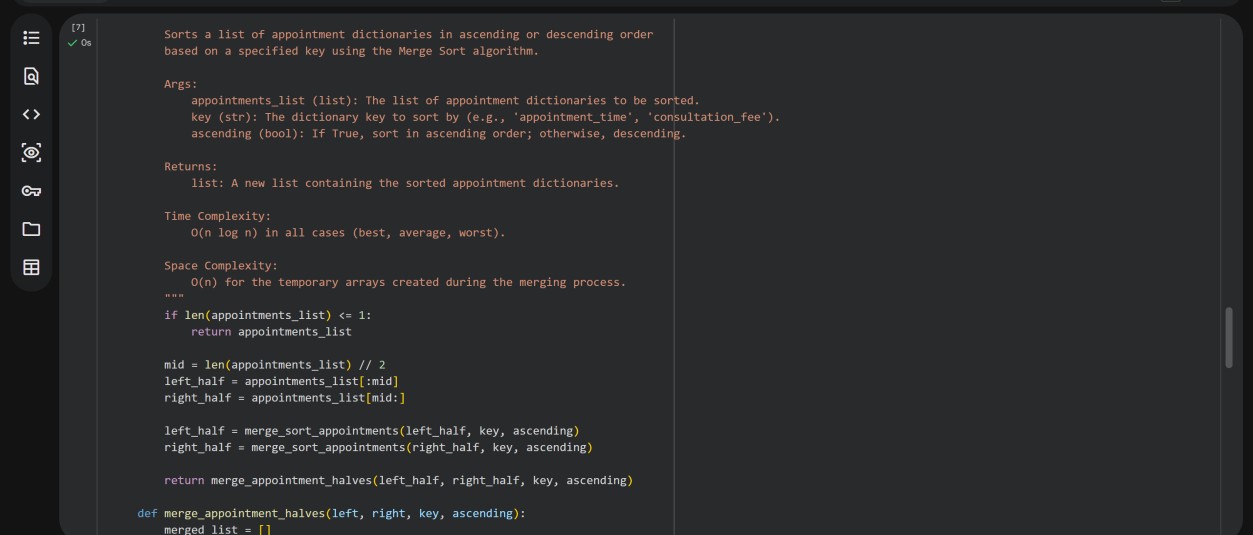
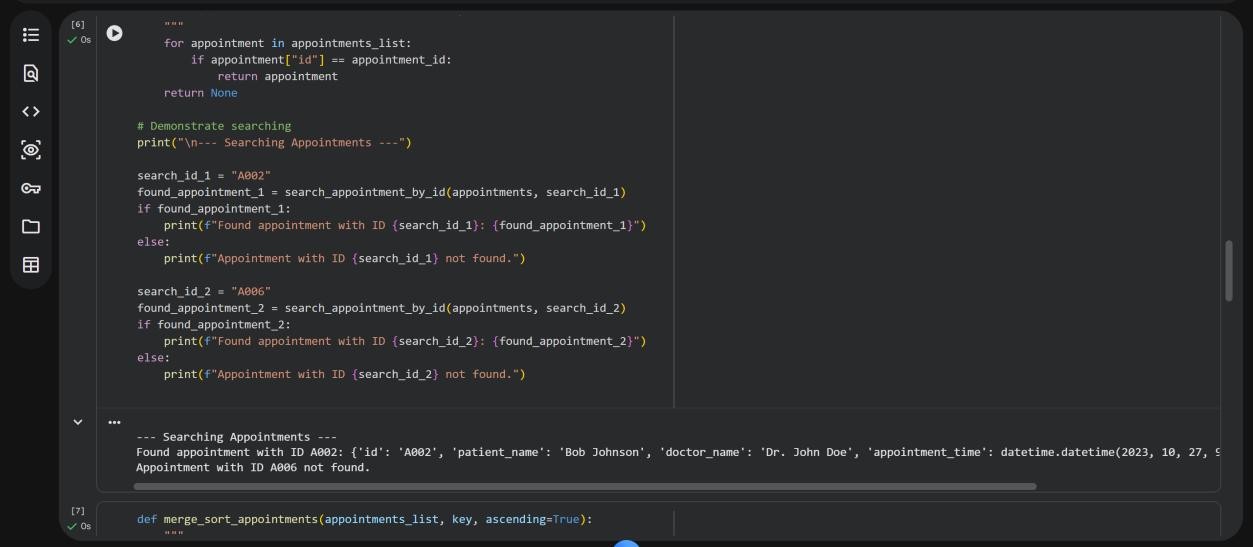
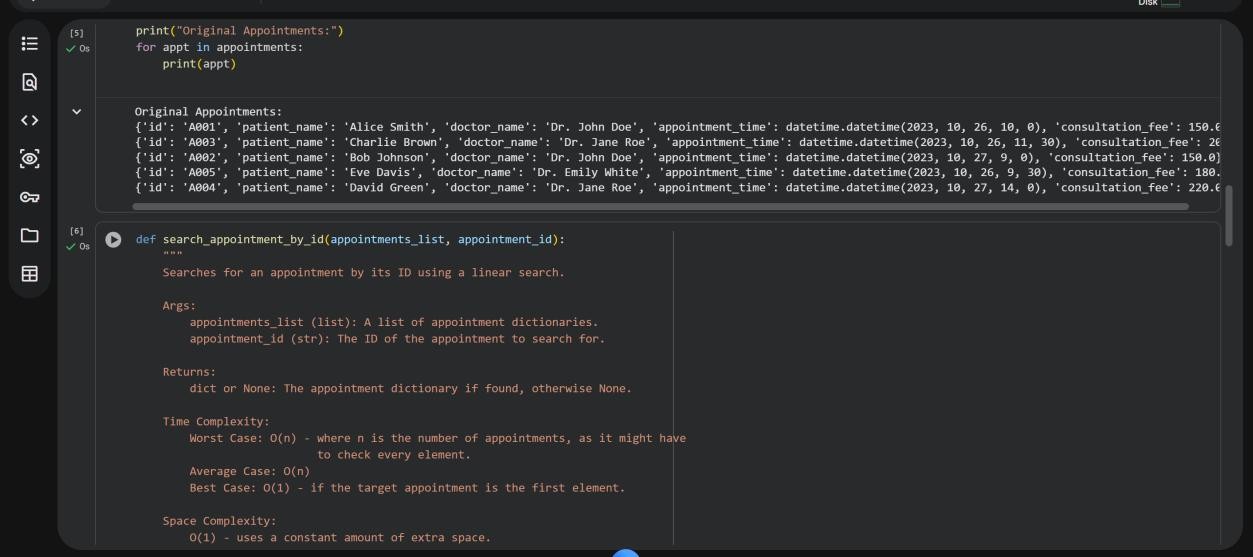
Student Task

* Use AI to recommend suitable searching and sorting algorithms.
* Justify the selected algorithms.
* Implement the algorithms in Python.

CODE:

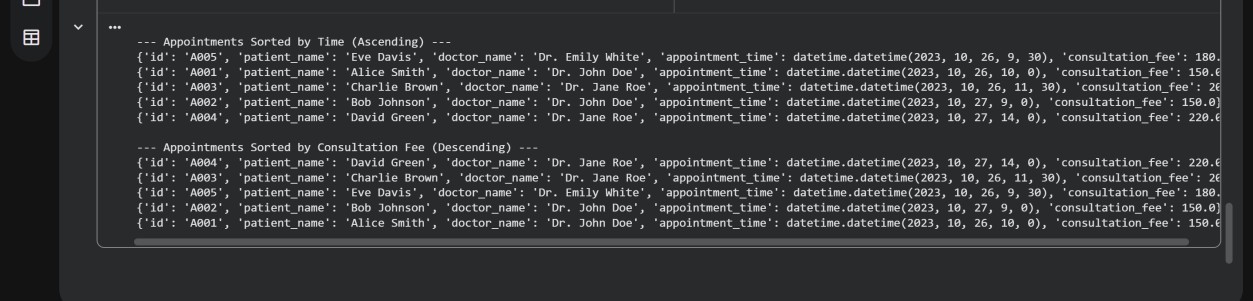
Recommended Algorithms and Justifications

1. Searching Appointments by ID
   * **Algorithm:** Linear Search
   * **Justification:** For a list of appointment records that are not necessarily sorted by appointment\_id, a linear search is the most straightforward approach. It iterates through each appointment until the matching ID is found. While its time complexity is O(n) in the worst case (where 'n' is the number of appointments), it's simple to implement and sufficient for moderately sized lists or when the data structure doesn't guarantee sorting by ID. For a real-world system requiring very fast lookups on large datasets, a hash map (Python dictionary) that maps appointment\_id to the appointment object would provide an average time complexity of O(1). However, since the task implies working with a list, linear search is chosen for direct list iteration.
2. Sorting Appointments by Time or Consultation Fee
   * **Algorithm:** Merge Sort
   * **Justification:** Merge Sort is an excellent choice for sorting appointments based on either time or consultation fee for several reasons:
     + **Time Complexity:** It consistently offers O(n log n) time complexity in all cases (best, average, worst). This makes it highly efficient for larger datasets, unlike Quick Sort which can degrade to O(n^2) in the worst case.
     + **Stability:** Merge Sort is a stable sorting algorithm, meaning that if two appointments have the same time or consultation fee, their relative order in the original list is preserved in the sorted list. This can be important for certain business logic (e.g., first-come, first-served for appointments with the same time).
     + **Predictable Performance:** Its consistent O(n log n) performance makes it reliable for critical system functions where predictable execution time is valued.





OUTPUT:



**Task Description #4: Railway Ticket Reservation System**

Scenario

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

1. Search tickets using ticket ID.
2. Sort bookings based on travel date or seat number.

Student Task

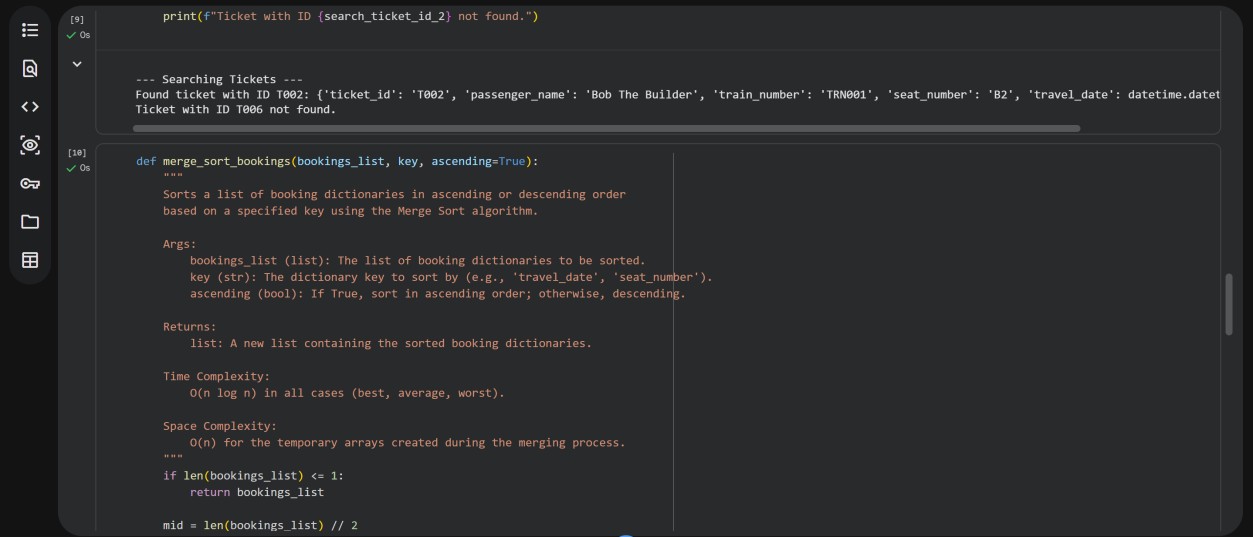
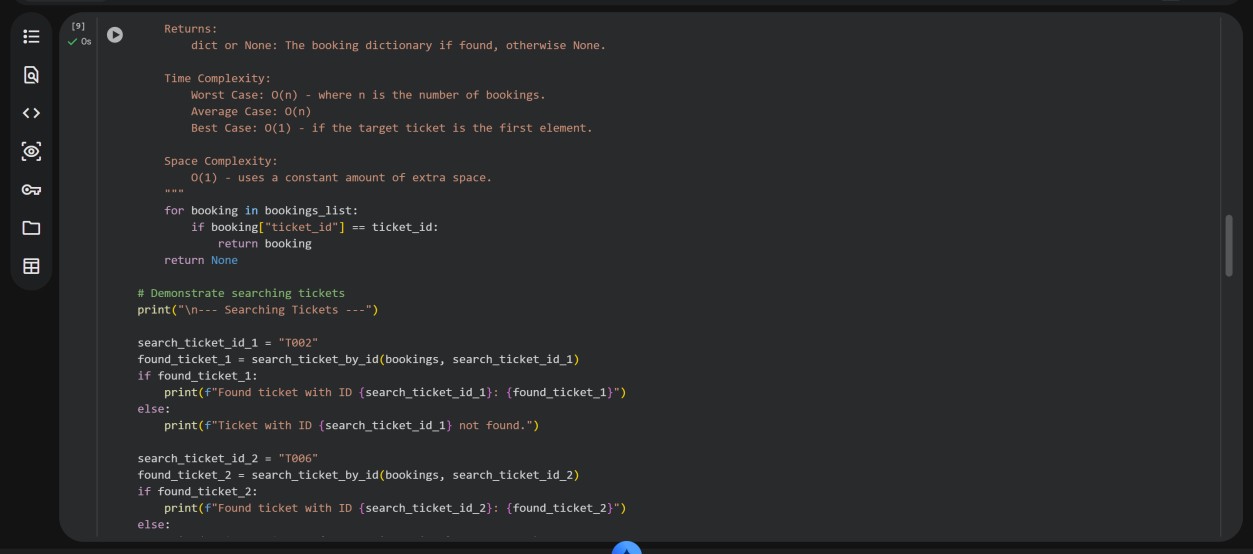
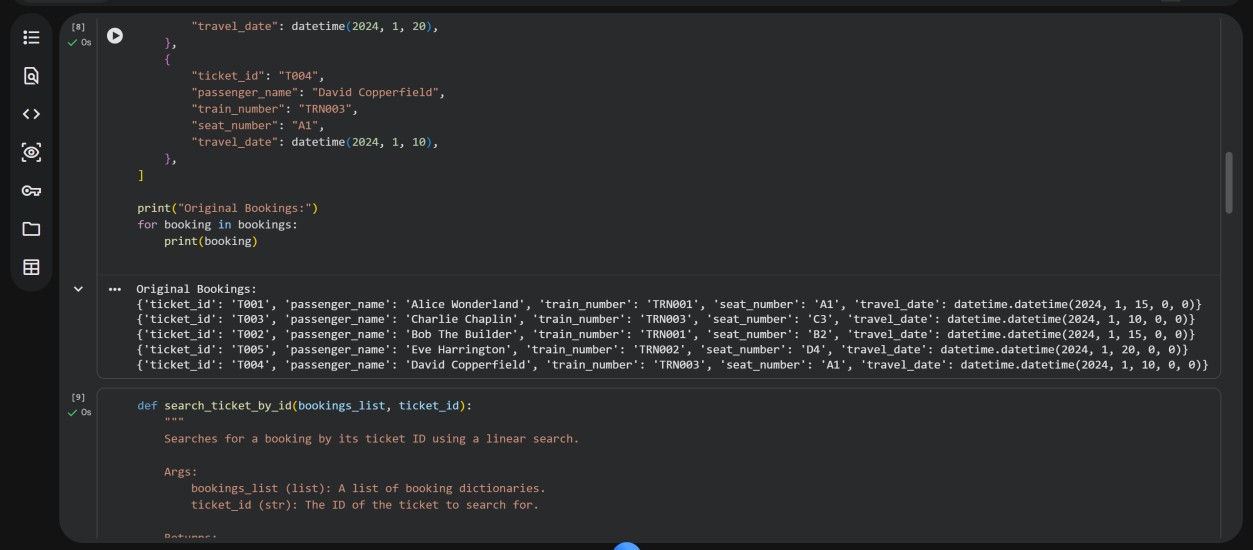
* Identify efficient algorithms using AI assistance.
* Justify the algorithm choices.
* Implement searching and sorting in Python.

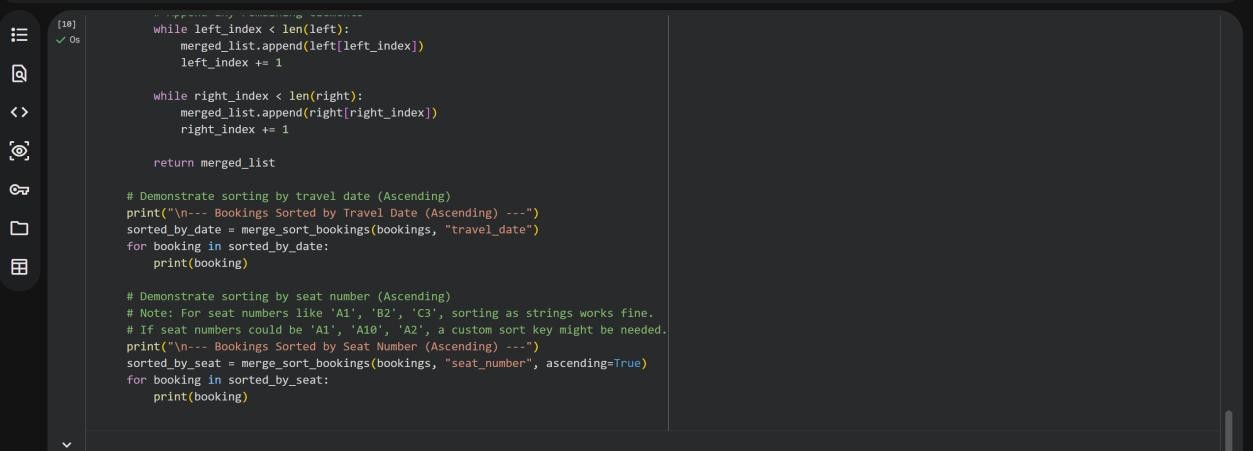
**CODE:**

Recommended Algorithms and Justifications for Railway Ticket Reservation System

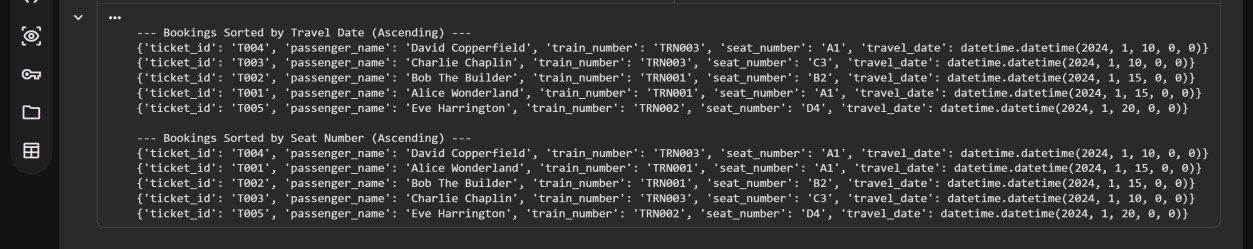
1. Searching Tickets by Ticket ID
   * **Algorithm:** Linear Search
   * **Justification:** When dealing with a list of booking records where ticket\_ids are unique but not necessarily stored in any particular order (e.g., insertion order), a linear search is a practical and easy-to-implement solution. It involves iterating through each booking record until the matching ticket\_id is found. Its time complexity is O(n) in the worst case, which is acceptable for moderately sized lists. For extremely large datasets, a hash map (Python dictionary) could offer O(1) average-case lookup, but for a list-based scenario, linear search is a direct fit.
2. Sorting Bookings by Travel Date or Seat Number
   * **Algorithm:** Merge Sort
   * **Justification:** Merge Sort is an excellent choice for sorting booking records based on travel\_date or seat\_number due to its consistent and efficient performance:
     + **Time Complexity:** It guarantees an O(n log n) time complexity in all scenarios (best, average, worst case). This makes it highly scalable and reliable for various sizes of booking data.
     + **Stability:** Merge Sort is a stable sorting algorithm. This property is beneficial here because if two booking records have the

same travel\_date or seat\_number, their original relative order will be preserved in the sorted output. For example, if multiple passengers are on the same date, their original booking order would be maintained, which can be useful for certain system requirements.





**OUTPUT:**



**Task Description #5: Smart Hostel Room Allocation System** A hostel management system stores student room allocation details including student ID, room number, floor, and allocation date. The system needs to:

1. Search allocation details using student ID.
2. Sort records based on room number or allocation date.

Student Task

* Use AI to suggest optimized algorithms.
* Justify the selections.
* Implement the solution in Python.

**CODE:**

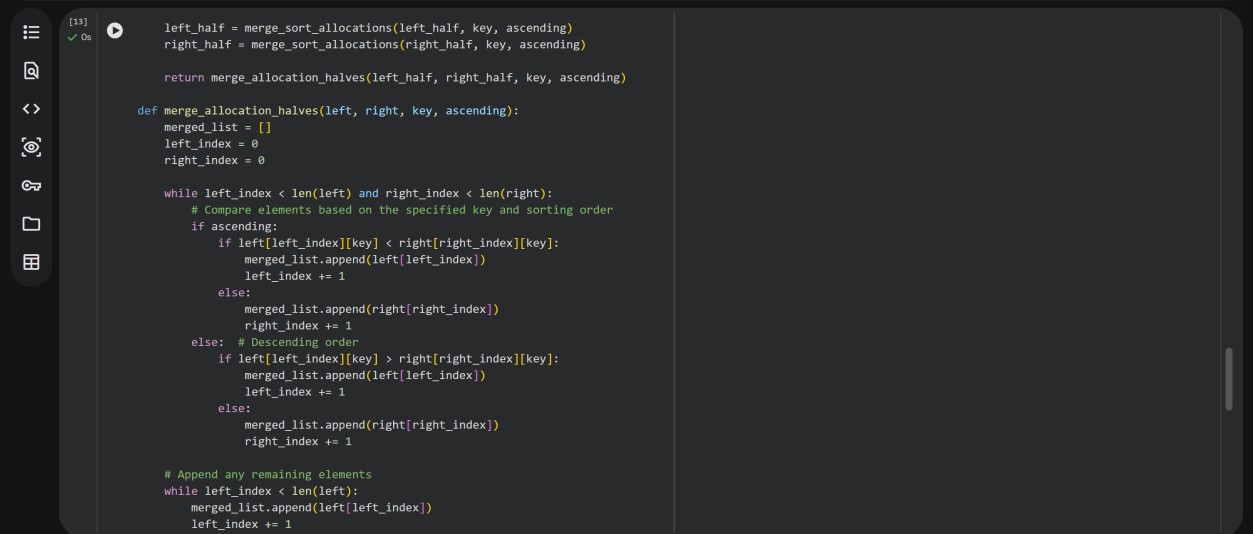
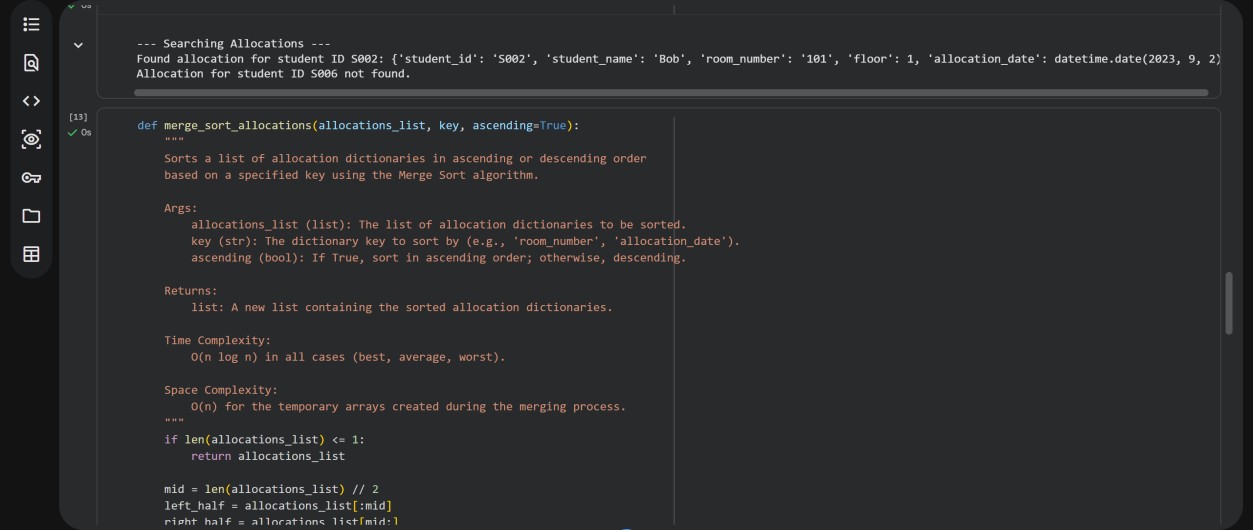
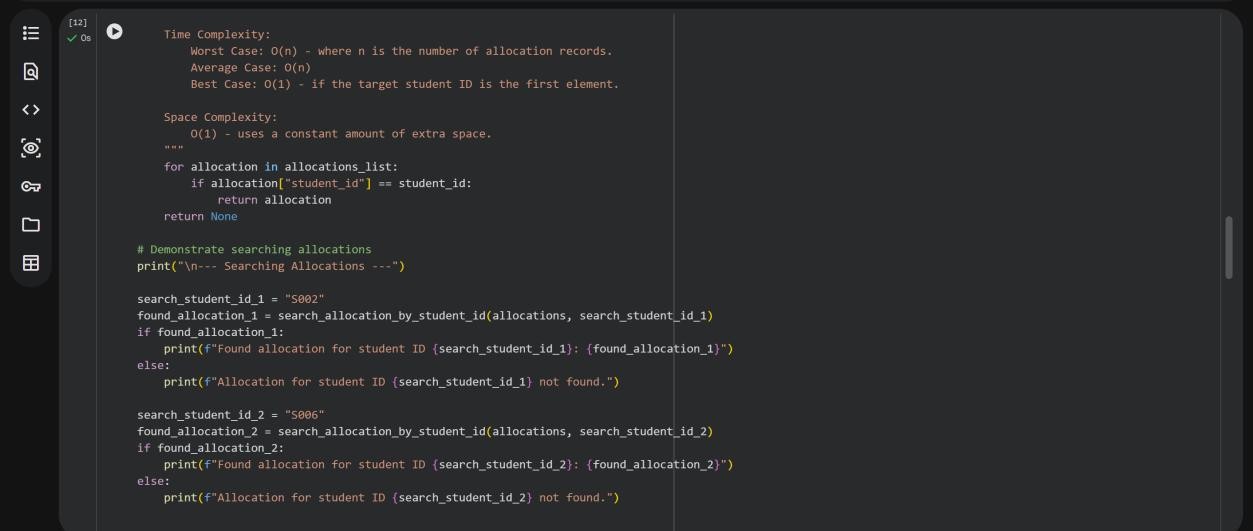
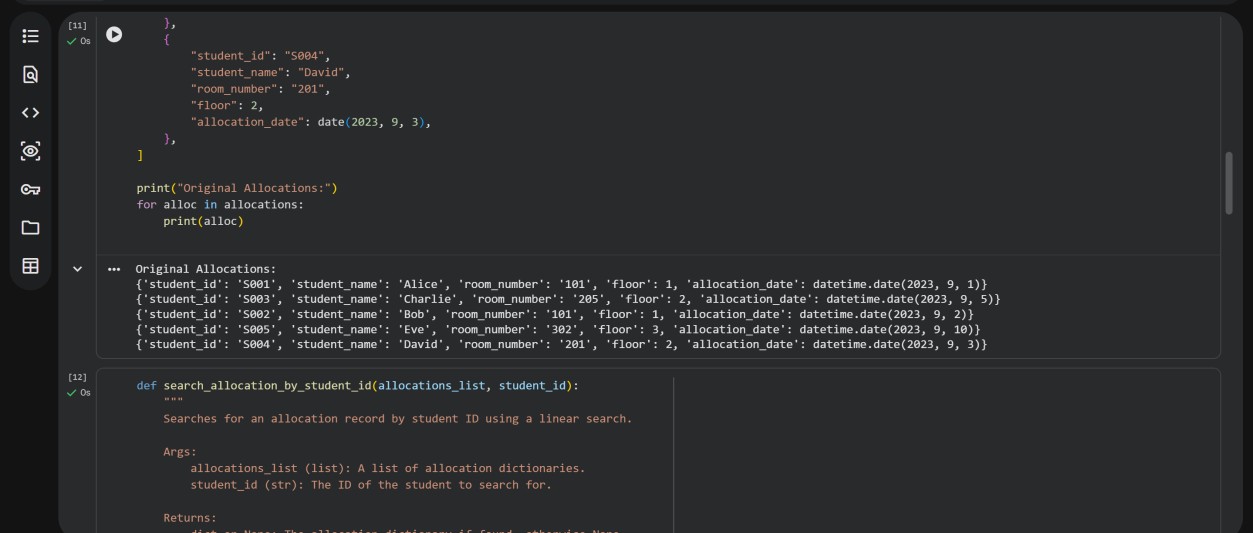
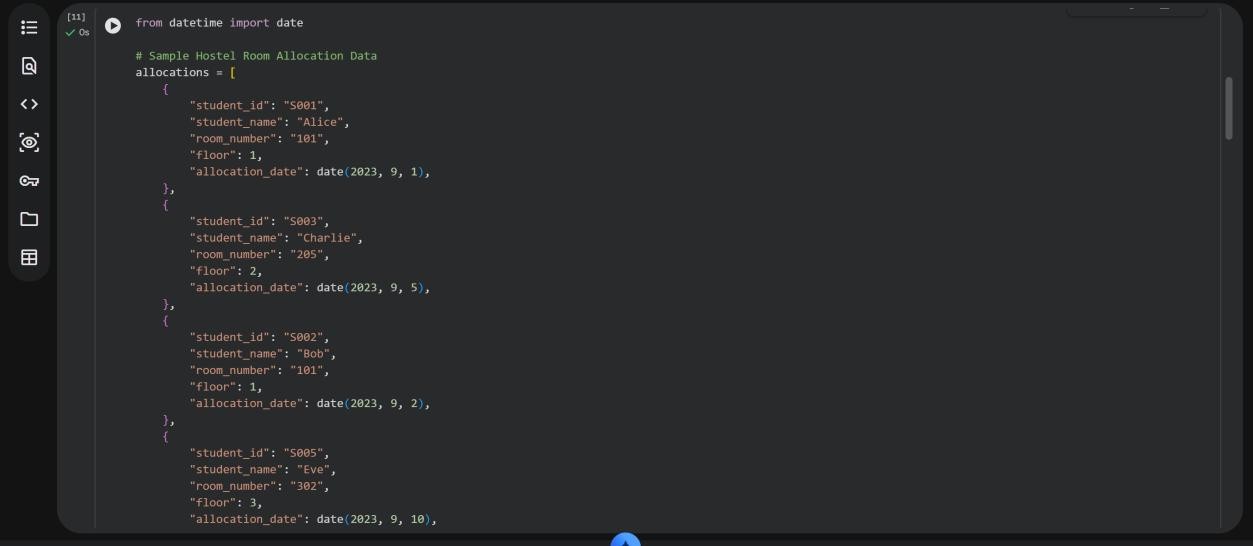
Recommended Algorithms and Justifications for Smart Hostel Room Allocation System

1. Searching Allocation Details by Student ID
   * **Algorithm:** Linear Search
   * **Justification:** For a list of room allocation records that are typically not kept in sorted order by student\_id, a linear search is the most direct and simplest approach. It involves iterating through each allocation record until the student\_id matches. While its worst-case time complexity is O(n), it's efficient enough for lists of moderate size

or when the data structure does not support faster indexed lookups. If real-time, very fast lookups on a massive scale were critical, a hash map (Python dictionary) keyed by student\_id would be preferred for its average O(1) time complexity.

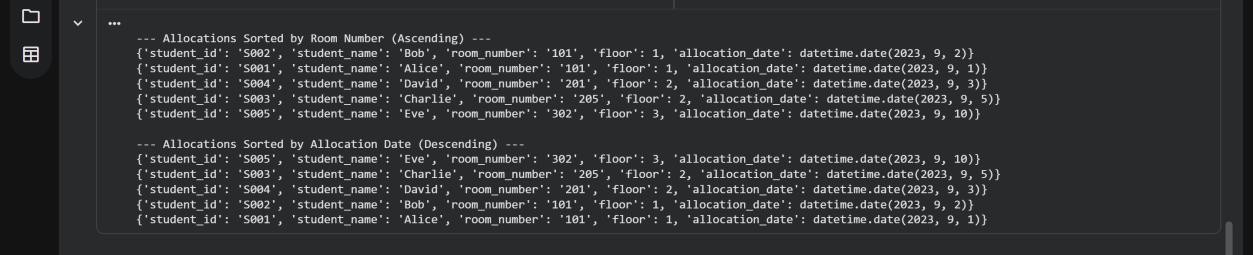
1. Sorting Records by Room Number or Allocation Date
   * **Algorithm:** Merge Sort
   * **Justification:** Merge Sort is an excellent algorithm for sorting room allocation records based on either room\_number or allocation\_date for the following reasons:

o **Time Complexity:** It consistently provides an optimal O(n log n) time complexity across all cases (best, average, and worst). This makes it very efficient for handling larger datasets and ensures predictable performance. o **Stability:** Merge Sort is a stable sorting algorithm. This means that if two allocation records have the same room\_number or allocation\_date, their relative order from the original list will be preserved in the sorted list. This can be important, for instance, if allocations for the same room number need to maintain an original entry order.





**OUTPUT:**



**Task Description #6: Online Movie Streaming Platform**

A streaming service maintains movie records with movie ID, title, genre, rating, and release year. The platform needs to:

1. Search movies by movie ID.
2. Sort movies based on rating or release year.

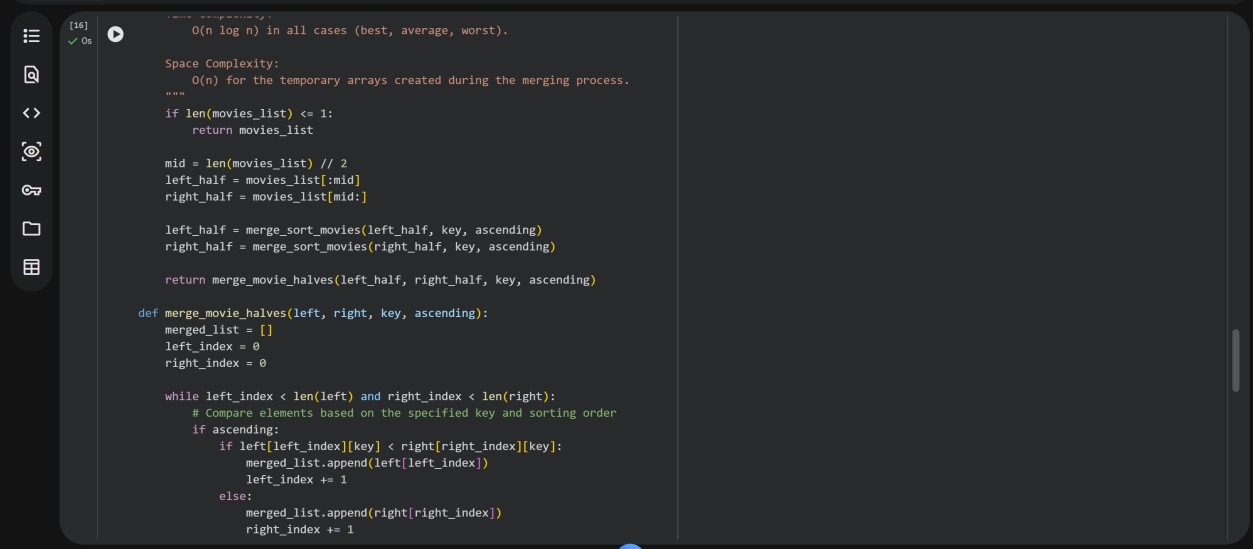
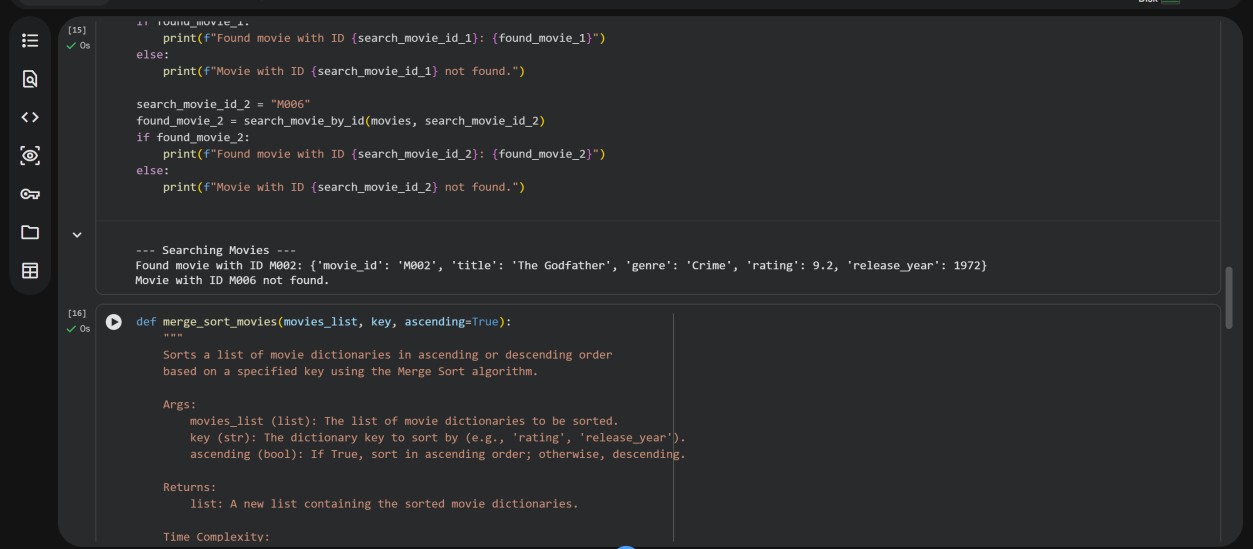
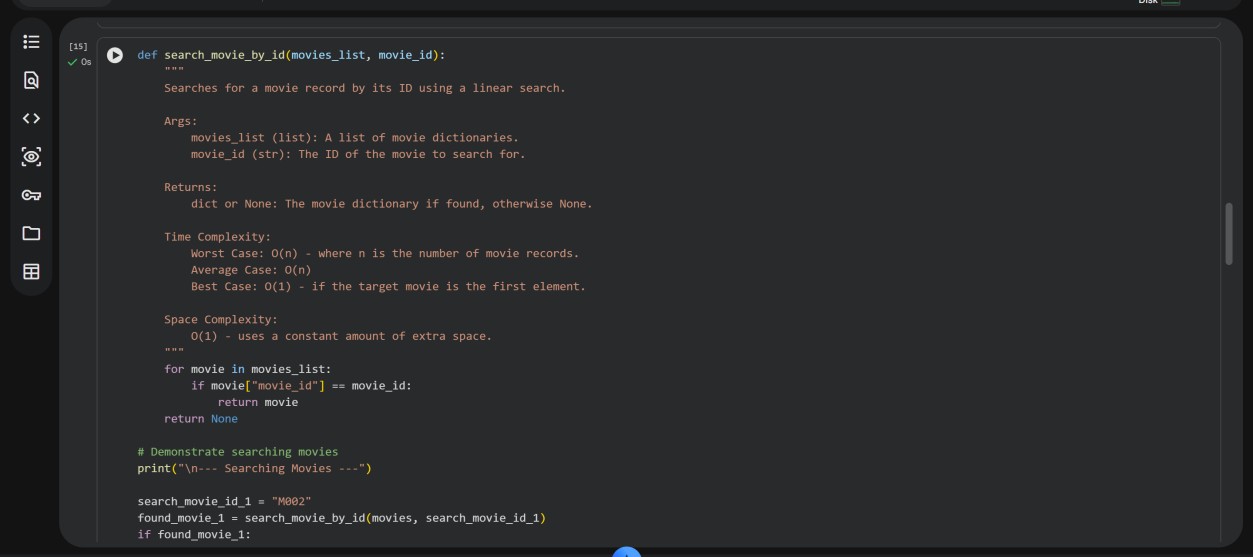
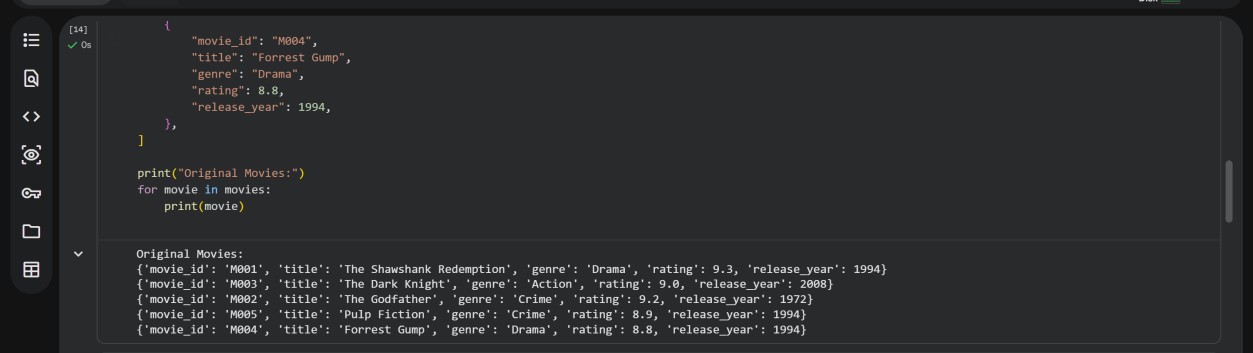
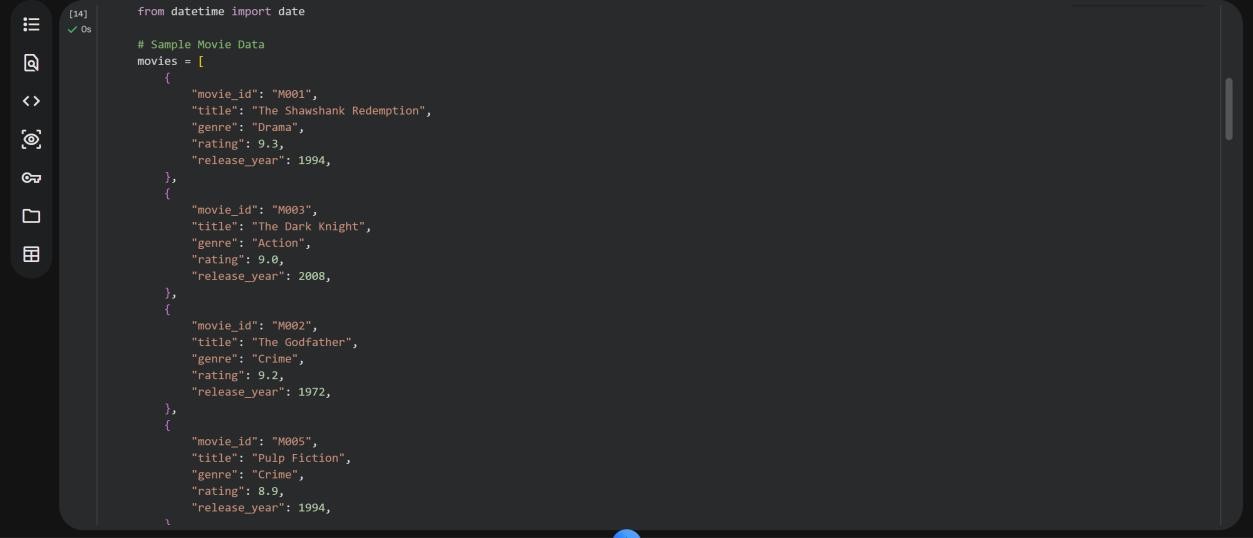
Student Task

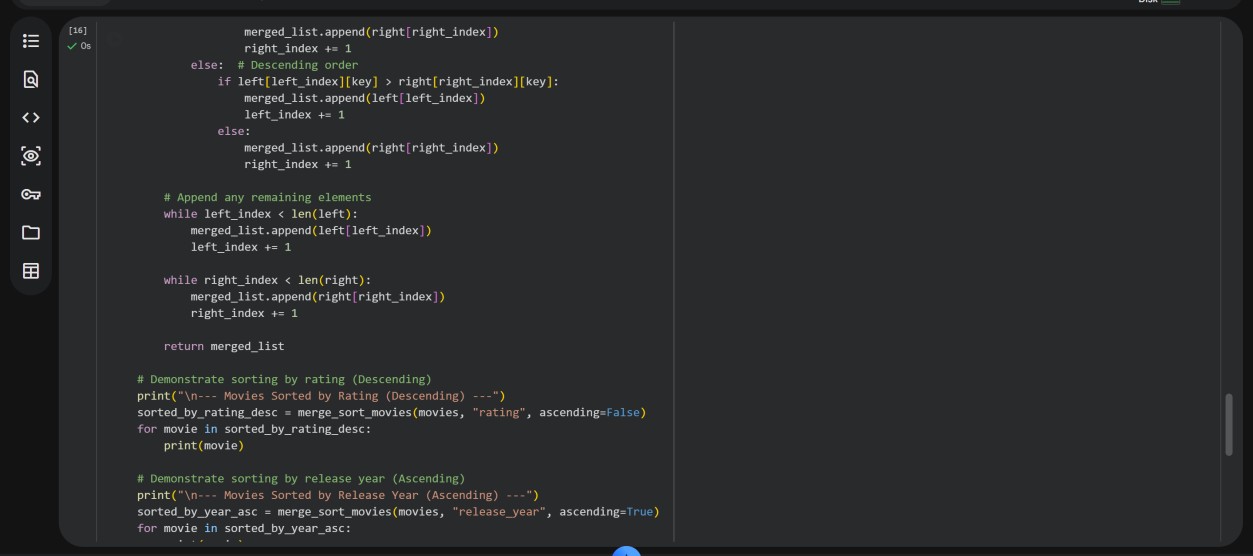
* Recommend searching and sorting algorithms using AI.
* Justify the chosen algorithms.
* Implement Python functions.

**CODE:**

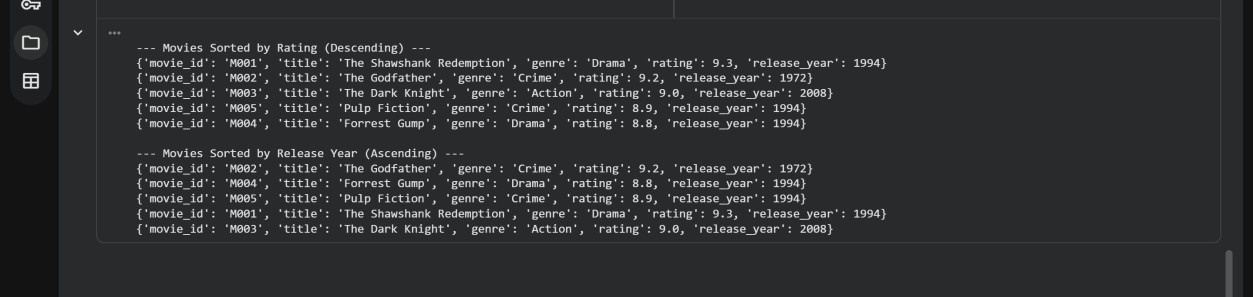
Recommended Algorithms and Justifications for Online Movie Streaming Platform

1. Searching Movies by Movie ID
   * **Algorithm:** Linear Search
   * **Justification:** For a list of movie records where movie\_ids are unique but not necessarily stored in any specific order, a linear search is a practical and easy-toimplement solution. It involves iterating through each movie record until the matching movie\_id is found. Its time complexity is O(n) in the worst case, which is generally acceptable for moderately sized lists. For platforms with extremely large movie catalogs and frequent ID lookups, a hash map (Python dictionary) could provide average O(1) time complexity by mapping movie IDs to their respective movie objects.
2. Sorting Movies by Rating or Release Year
   * **Algorithm:** Merge Sort
   * **Justification:** Merge Sort is an excellent choice for sorting movie records based on rating or release\_year due to its consistent and efficient performance:
     + **Time Complexity:** It guarantees an optimal O(n log n) time complexity across all cases (best, average, and worst). This makes it highly scalable and reliable for various sizes of movie data, ensuring predictable performance.
     + **Stability:** Merge Sort is a stable sorting algorithm. This property is particularly useful here because if multiple movies have the same rating or release year, their original relative order will be preserved in the sorted output. For example, if two movies have the same high rating, their order might be preserved based on their original entry into the system, which can be a desirable feature.





**OUTPUT:**



**Task Description #7: Smart Agriculture Crop Monitoring System** An agriculture monitoring system stores crop data with crop ID, crop name, soil moisture level, temperature, and yield estimate. Farmers need to:

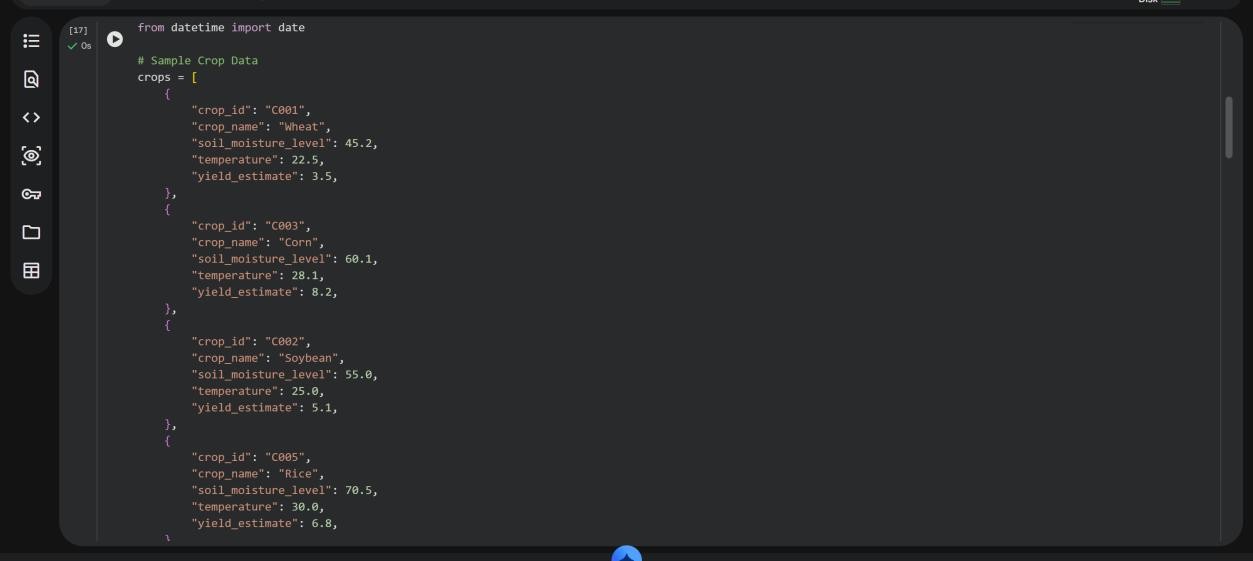
1. Search crop details using crop ID.
2. Sort crops based on moisture level or yield estimate.

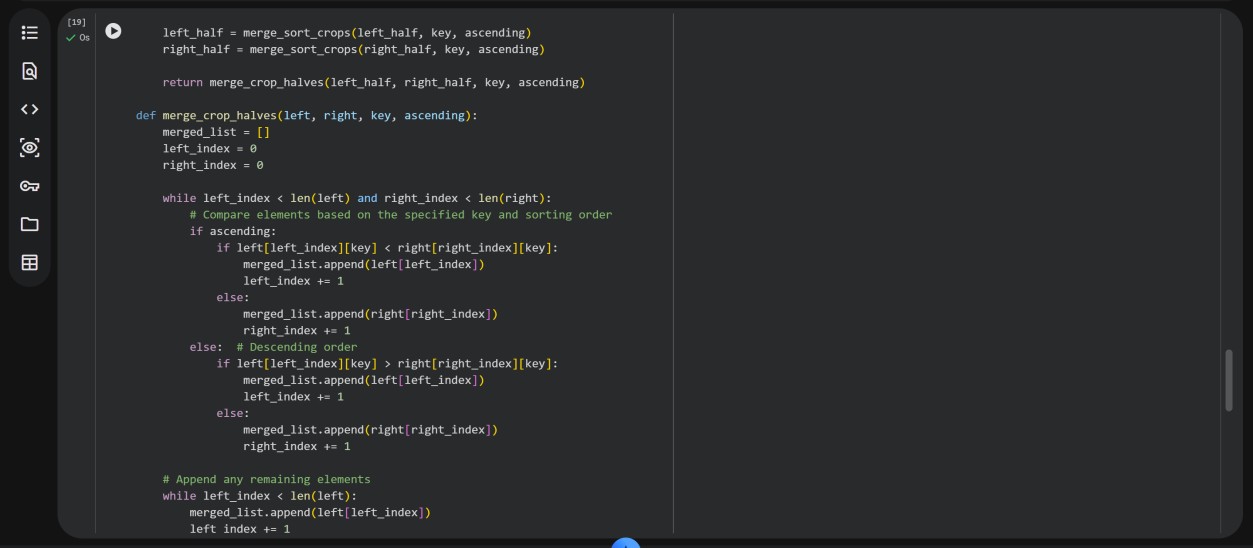
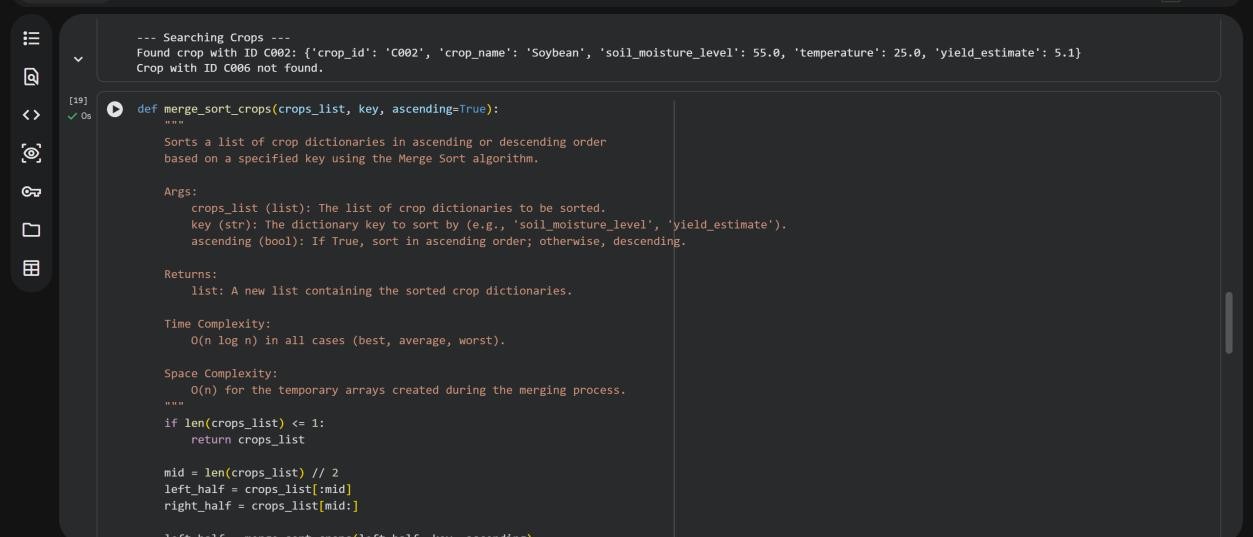
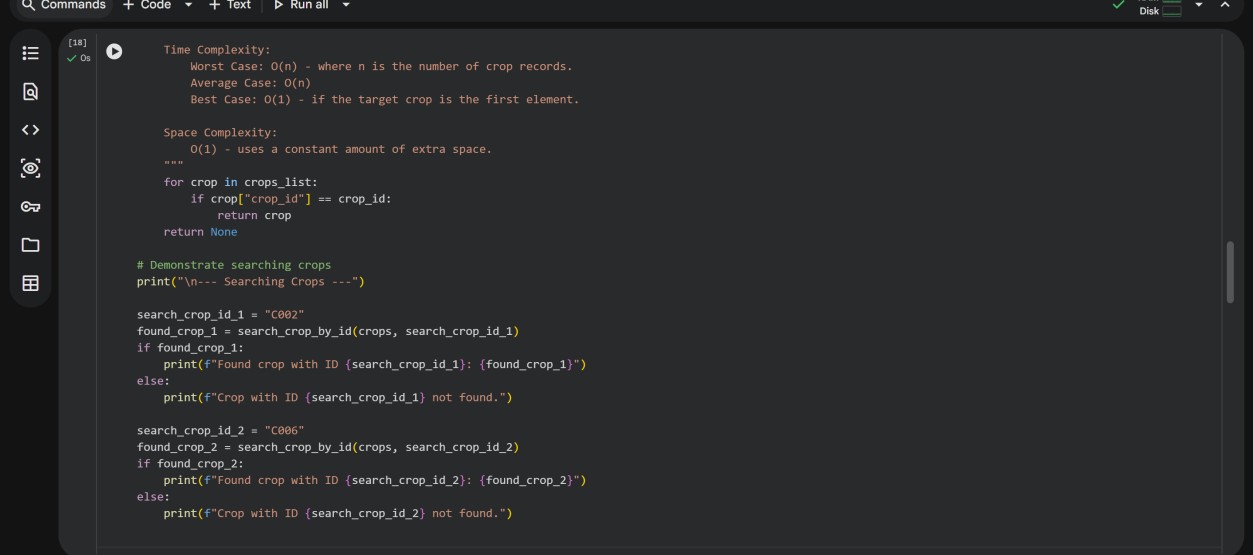
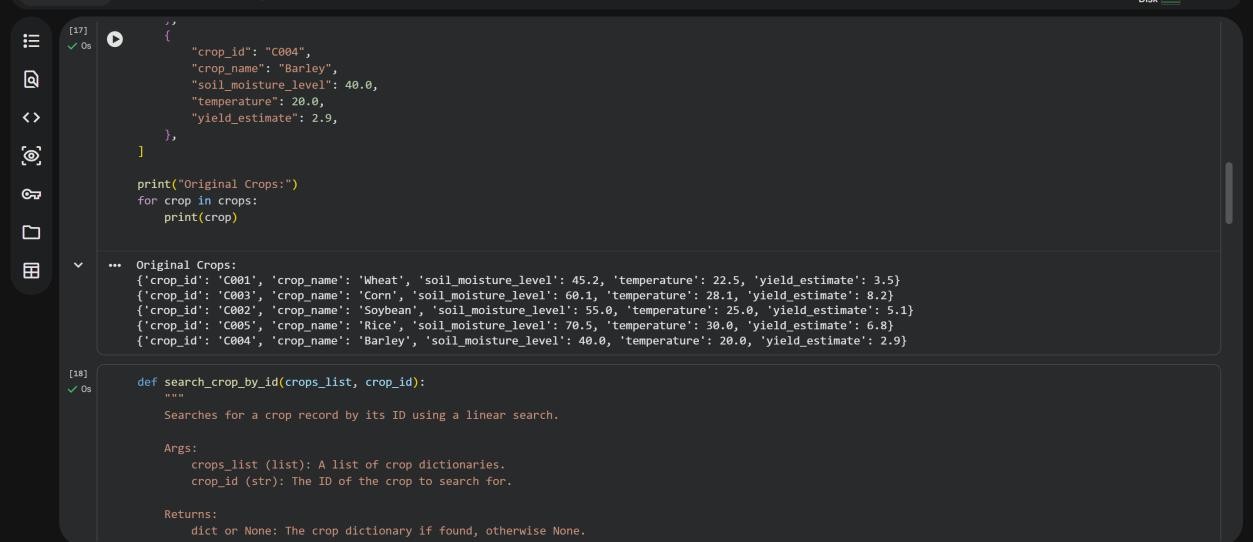
Student Task

* Use AI-assisted reasoning to select algorithms.
* Justify algorithm suitability.
* Implement searching and sorting in Python.

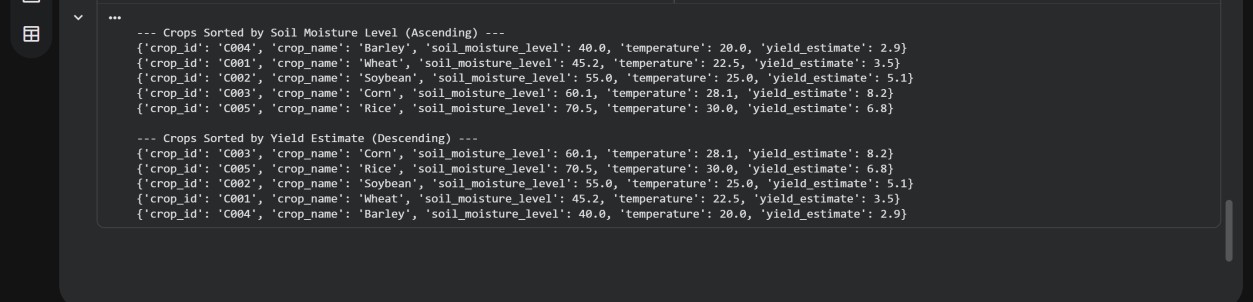
**CODE:**

1. Searching Crop Details by Crop ID
   * **Algorithm:** Linear Search
   * **Justification:** For a list of crop records where crop\_ids are unique but not necessarily stored in a sorted order, a linear search is a straightforward and easy-to-implement approach. It involves iterating through each crop record until the matching crop\_id is found. Its time complexity is O(n) in the worst case, which is generally acceptable for typical list sizes in such a system. For very large datasets requiring extremely fast lookups, a hash map (Python dictionary) that maps crop IDs to their respective crop objects would offer average O(1) time complexity.
2. Sorting Crops by Moisture Level or Yield Estimate
   * **Algorithm:** Merge Sort
   * **Justification:** Merge Sort is an excellent choice for sorting crop records based on soil\_moisture\_level or yield\_estimate due to its consistent and efficient performance:
     + **Time Complexity:** It consistently provides an optimal O(n log n) time complexity across all cases (best, average, and worst). This makes it very efficient for handling larger datasets and ensures predictable performance, which is valuable in monitoring systems.
     + **Stability:** Merge Sort is a stable sorting algorithm. This means that if multiple crops have the same moisture level or yield estimate, their relative order from the original list will be preserved in the sorted output. This can be beneficial for maintaining other inherent orderings within the data (e.g., if you want to see crops with the same moisture level ordered by their planting date).





**OUTPUT:**



**Task Description #8: Airport Flight Management System** An airport system stores flight information including flight ID, airline name, departure time, arrival time, and status. The system must:

1. Search flight details using flight ID.
2. Sort flights based on departure time or arrival time.

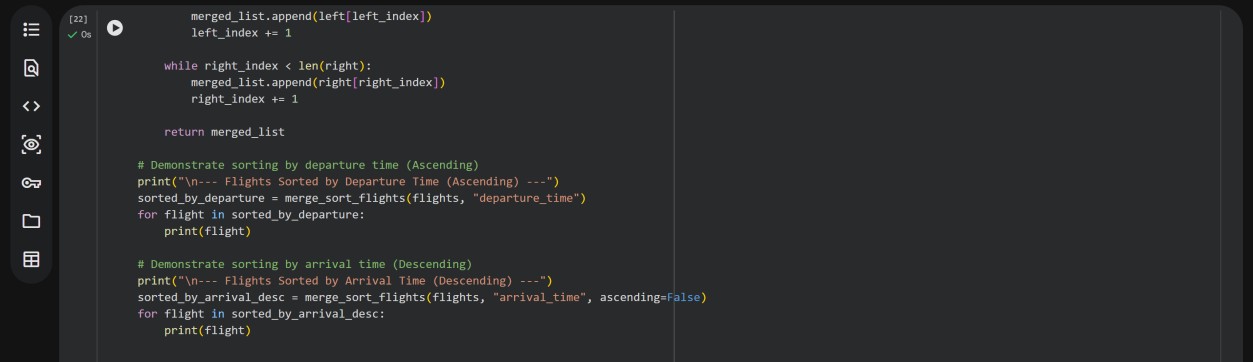
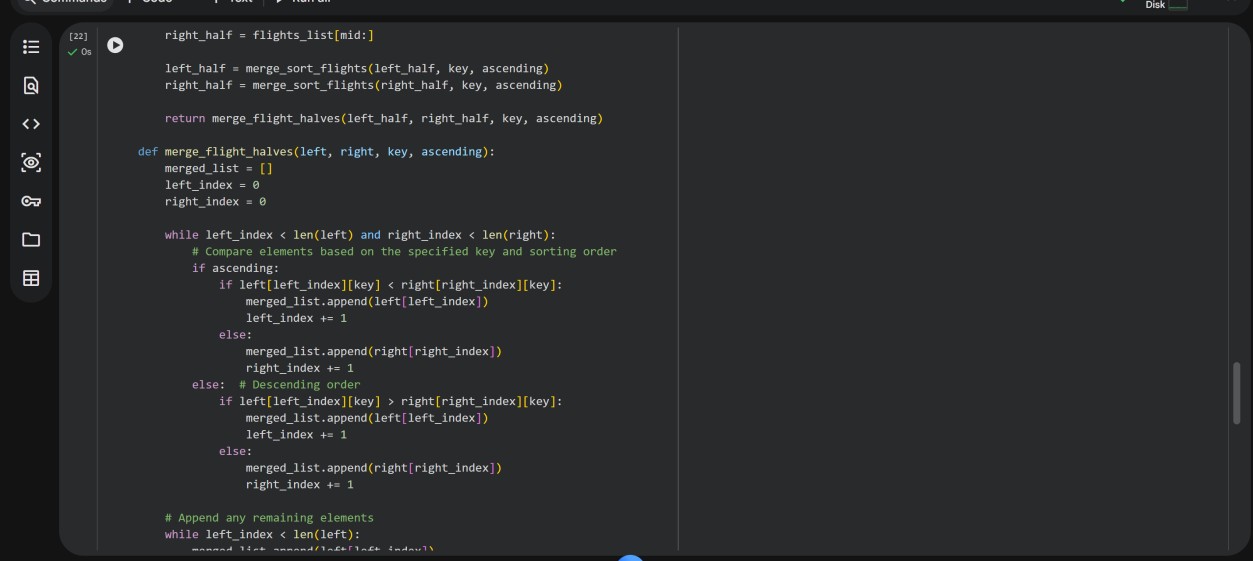
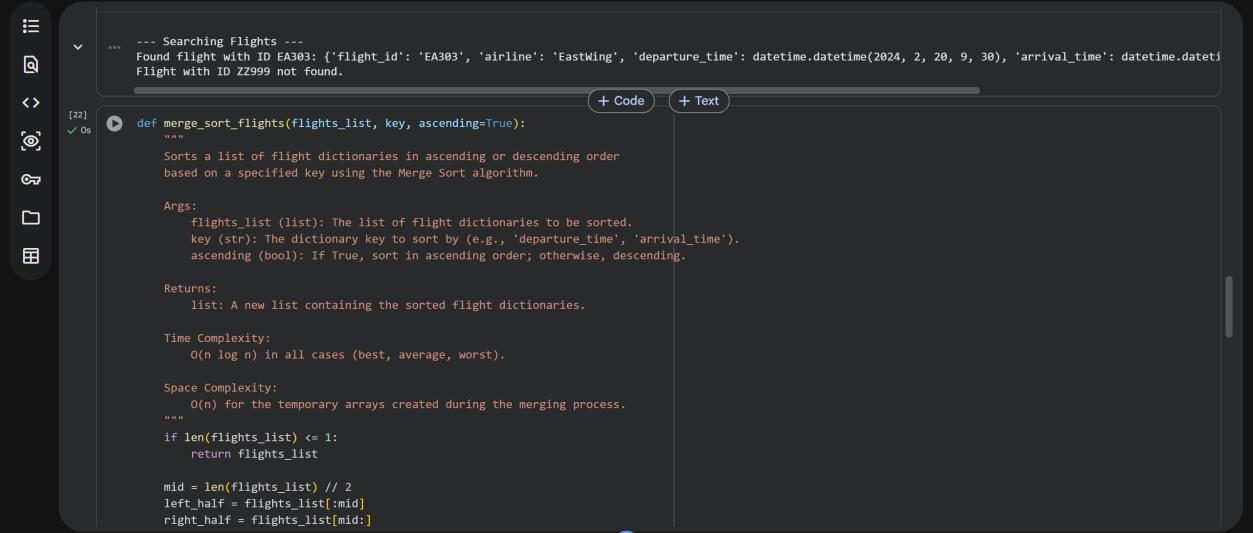
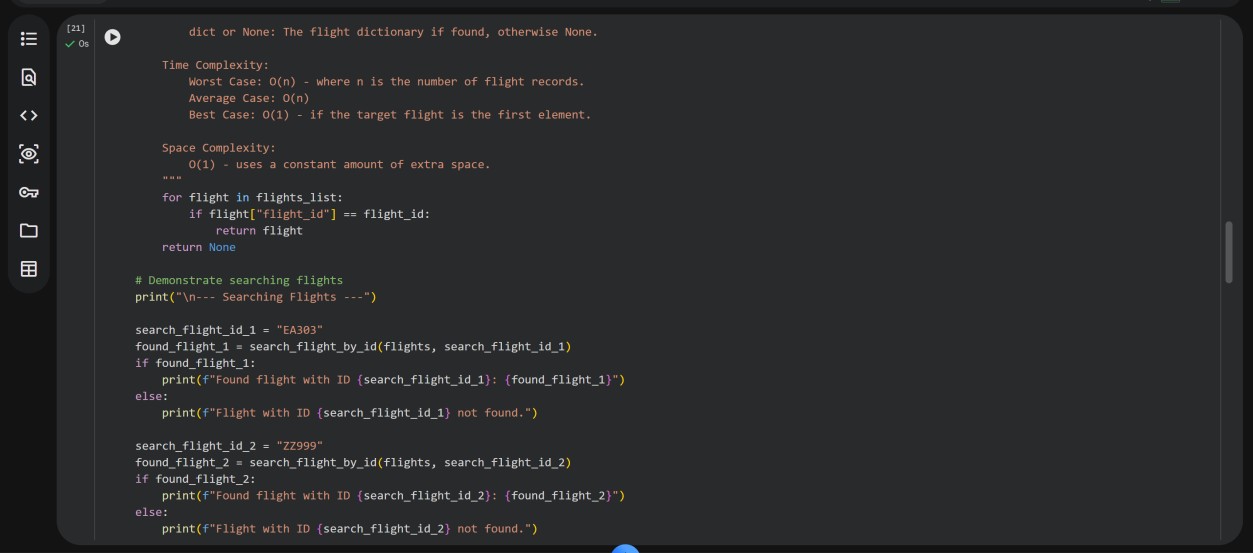
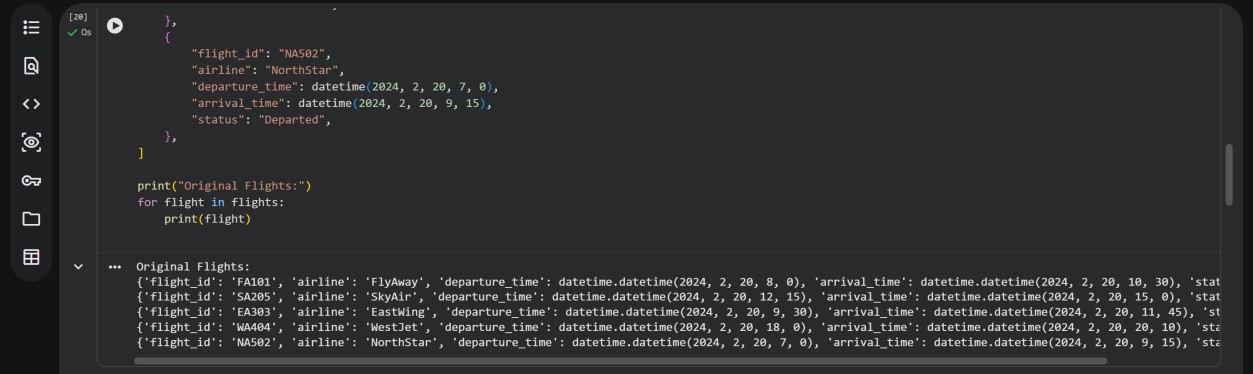
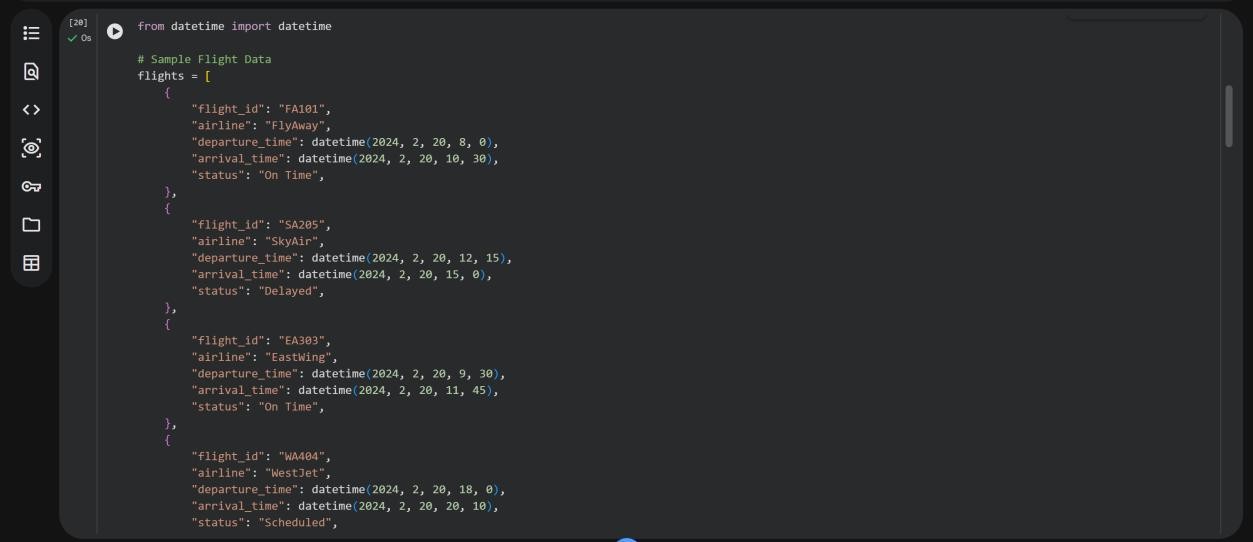
Student Task

* Use AI to recommend algorithms.
* Justify the algorithm selection.
* Implement searching and sorting logic in Python.

**CODE:**

**Recommended Algorithms and Justifications for Airport System**

1. **Searching Flight Details by Flight ID** 
   * Algorithm: Linear Search
   * Justification: For a list of flight records where flight\_ids are unique but not necessarily stored in any specific order, a linear search is a practical and easy-to-implement solution. It involves iterating through each flight record until the matching flight\_id is found. Its time complexity is O(n) in the worst case, which is generally acceptable for typical list sizes in such a system. For very large datasets requiring extremely fast lookups, a hash map (Python dictionary) that maps flight IDs to their respective flight objects would offer average O(1) time complexity.
2. **Sorting Flights by Departure Time or Arrival Time** 
   * Algorithm: Merge Sort
   * Justification: Merge Sort is an excellent choice for sorting flight records based on departure\_time or arrival\_time due to its consistent and efficient performance:
     + Time Complexity: It consistently provides an optimal O(n log n) time complexity across all cases (best, average, and worst). This makes it very efficient for handling larger datasets and ensures predictable performance, which is valuable in real-time systems like airport management.
     + Stability: Merge Sort is a stable sorting algorithm. This means that if multiple flights have the same departure or arrival time, their relative order from the original list will be preserved in the sorted output. This can be beneficial for maintaining other inherent orderings within the data.



**OUTPUT:**

