

AI Assisted Coding

Assignment 12.5

Name: ch .koushik

Hall ticket no: 2303A51938

Batch no: 19

Task 1: Sorting – Merge Sort Implementation

Prompt:

Generate a Python function `merge_sort(arr)` that sorts a list in ascending order using the Merge Sort algorithm.

Include time complexity and space complexity in the function docstring.

Code & Output:

The screenshot shows a Microsoft Visual Studio Code interface with the following details:

- Title Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Left Sidebar:** Welcome, Assignment_12.5.py (active tab).
- Code Editor:** The main pane displays Python code for a merge sort algorithm. The code includes docstrings for both the `merge_sort` and `merge` functions, detailing their parameters, returns, and descriptions.
- Right Sidebar:** Shows the file path: E:\3rd Year (2nd Sem) AI Assisted Coding\Assignment_12.5.py, and the status bar indicates Line 48, Col 17, Spaces: 4, UTF-8, CRLF.

```
Assignment_12.5.py > ...
1 # Task 1: Sorting - Merge Sort Implementation"
2 # Generate a Python function merge_sort(arr) that sorts a list in ascending order
3 # using the Merge Sort algorithm.
4 # Include time complexity and space complexity in the function docstring.
5
6 def merge_sort(arr):
7     """
8         Sorts a List in ascending order using the Merge Sort algorithm.
9         Time Complexity: O(n Log n) - where n is the number of elements in the List.
10        Space Complexity: O(n) - due to the temporary arrays used for merging.
11        Parameters:
12            arr (list): The list to be sorted.
13        Returns:
14            list: A new sorted list in ascending order.
15        """
16        if len(arr) <= 1:
17            return arr
18
19        mid = len(arr) // 2
20        left_half = merge_sort(arr[:mid])
21        right_half = merge_sort(arr[mid:])
22
23        return merge(left_half, right_half)
24
25 def merge(left, right):
26     """
27         Merges two sorted lists into a single sorted list.
28         Parameters:
29             left (list): The first sorted list.
30             right (list): The second sorted list.
31         Returns:
32             list: A merged and sorted list containing all elements from left and right.
33         """
34
35         merged = []
36         left_index = right_index = 0
37
38         while left_index < len(left) and right_index < len(right):
39             if left[left_index] < right[right_index]:
```

The screenshot shows a code editor with two tabs: 'Assignment_12.5.py' and 'Code'. The 'Assignment_12.5.py' tab contains Python code for a merge sort algorithm. The 'Code' tab shows a terminal window with the output of running the code.

```
Assignment_12.5.py > ...
def merge(left, right):
    """
    Merges two sorted lists into a single sorted list.
    Parameters:
    left (list): The first sorted list.
    right (list): The second sorted list.
    Returns:
    list: A merged and sorted list containing all elements from left and right.
    """
    merged = []
    left_index = right_index = 0

    while left_index < len(left) and right_index < len(right):
        if left[left_index] < right[right_index]:
            merged.append(left[left_index])
            left_index += 1
        else:
            merged.append(right[right_index])
            right_index += 1

    # If there are remaining elements in left or right, add them to merged
    merged.extend(left[left_index:])
    merged.extend(right[right_index:])

    return merged

if __name__ == "__main__":
    unsorted_list = [38, 27, 43, 3, 9, 82, 10]
    sorted_list = merge_sort(unsorted_list)
    print("Sorted List:", sorted_list)
```

Code

```
Code > ...
SemAT Assisted Coding\Assignment_12.5\Assignment_12.5.py
Sorted List: [3, 9, 10, 27, 38, 43, 82]
PS E:\3rd Year\2nd Sem\SemAT Assisted Coding\Assignment_12.5>
```

Explanation:

Merge Sort divides the list into smaller halves recursively and then merges them in sorted order. It guarantees $O(n \log n)$ time complexity in all cases, making it efficient for large datasets. The algorithm requires additional memory for merging, resulting in $O(n)$ space complexity. AI assistance helped generate structured recursive logic with proper documentation.

Task 2: Searching – Binary Search

Prompt:

Generate a Python function `binary_search(arr, target)` that returns the index of the target element in a sorted list or -1 if not found. Include complexity explanation in the docstring.

Code & Output:

The screenshot shows a code editor interface with two panes. The left pane displays the Python file `Assignment_12.5.py`. The code implements a binary search algorithm with detailed docstrings and complexity analysis. The right pane shows the terminal output of running the script, which includes the path to the file, the search results for element 5, and the current working directory.

```
Assignment_12.5.py > ...
54  # Task 2: Searching - Binary Search"
55  # Generate a Python function binary_search(arr, target) that returns the index of the
56  # target element in a sorted list or -1 if not found. Include complexity explanation in the
57  # docstring.
58  def binary_search(arr, target):
59      """
59       Returns the index of the target element in a sorted list or -1 if not found.
60       Time Complexity: O(log n) - where n is the number of elements in the list.
61       Space Complexity: O(1) - as it uses constant extra space.
62   Parameters:
63     arr (list): The sorted list to search through.
64     target: The element to search for.
65   Returns:
66     int: The index of the target element if found, otherwise -1.
67   """
68   left, right = 0, len(arr) - 1
69
70   while left <= right:
71       mid = left + (right - left) // 2
72
73       if arr[mid] == target:
74           return mid
75       elif arr[mid] < target:
76           left = mid + 1
77       else:
78           right = mid - 1
79
80   return -1
81
82 # Example usage:
83 if __name__ == "__main__":
84     sorted_list = [1, 3, 5, 7, 9, 11]
85     target = 5
86     index = binary_search(sorted_list, target)
87     if index != -1:
88         print(f"Element {target} found at index: {index}")
89     else:
90         print(f"Element {target} not found in the list.")

Code X
Code Coding\Assignment_12.5\Assignment_12.5.py
\2nd Sem\AI Assisted Coding\Assignment_12.5\Assignment_12.5.py
Element 5 found at index: 2
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5>
```

Explanation:

Binary Search works by repeatedly dividing the sorted list into halves. It significantly reduces the search space, resulting in logarithmic time complexity. It is efficient compared to linear search for large sorted datasets. AI assistance helped structure the iterative logic and complexity documentation clearly.

Task 3: Smart Healthcare Appointment Scheduling

Prompt:

Recommend suitable searching and sorting algorithms for appointment management and implement them in Python.

Code & Output:

```

Assignment_12.5.py > ...
85
86 "Task 3: Smart Healthcare Appointment Scheduling"
87 # Recommend suitable searching and sorting algorithms for appointment management and implement them in Python.
88 "Algorithm Recommendation:"
89 "For appointment management in a smart healthcare system, I recommend using the following algorithms:"
90 "1. Sorting Algorithm: Merge Sort - This algorithm is efficient for sorting large lists of appointments based on their scheduled time. It has a time complexity of O(n log n) and is stable, which means it maintains the relative order of records with equal keys."
91 "2. Searching Algorithm: Binary Search - Once the appointments are sorted, binary search can be used to quickly find specific appointments based on their scheduled time or patient ID. It has a time complexity of O(log n) and is efficient for searching in sorted lists."
92
93
94 def merge_sort(arr):
95     if len(arr) <= 1:
96         return arr
97
98     mid = len(arr) // 2
99     left_half = merge_sort(arr[:mid])
100    right_half = merge_sort(arr[mid:])
101
102    return merge(left_half, right_half)
103
104 def merge(left, right):
105     merged = []
106     left_index = right_index = 0
107
108     while left_index < len(left) and right_index < len(right):
109         if left[left_index]['time'] < right[right_index]['time']:
110             merged.append(left[left_index])
111             left_index += 1
112         else:
113             merged.append(right[right_index])
114             right_index += 1
115
116     merged.extend(left[left_index:])
117     merged.extend(right[right_index:])
118
119
Assignment_12.5.py > ...
100
101 def merge(left, right):
102     return merged
103
104 def binary_search(arr, target_time):
105     left, right = 0, len(arr) - 1
106
107     while left <= right:
108         mid = left + (right - left) // 2
109
110         if arr[mid]['time'] == target_time:
111             return mid
112         elif arr[mid]['time'] < target_time:
113             left = mid + 1
114         else:
115             right = mid - 1
116
117     return -1
118
119 # Example usage:
120 if __name__ == "__main__":
121     appointments = [
122         {"patient_id": 1, "time": "2024-07-01 10:00"},  # Patient ID 1 at 10:00
123         {"patient_id": 2, "time": "2024-07-01 09:00"},  # Patient ID 2 at 09:00
124         {"patient_id": 3, "time": "2024-07-01 11:00"}   # Patient ID 3 at 11:00
125     ]
126
127     sorted_appointments = merge_sort(appointments)
128     print("Sorted Appointments:", sorted_appointments)
129
130     target_time = "2024-07-01 10:00"
131     index = binary_search(sorted_appointments, target_time)
132     if index != -1:
133         print(f"Appointment found at index: {index}, Patient ID: {sorted_appointments[index]}")
134         print(f"[patient_id'{index}')")
135     else:
136         print(f"Appointment at time {target_time} not found.")
137
138

```

Explanation:

Binary Search is efficient for searching appointment IDs when records are sorted. Merge Sort ensures stable and efficient sorting by time or fee. These algorithms provide optimal performance for medium to large datasets. AI-assisted reasoning supported algorithm selection based on efficiency requirements.

Task 4: Railway Ticket Reservation System

Prompt:

Generate Python code for a Railway Ticket Reservation System that stores booking details including 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.
3. Sort bookings based on seat number.

First, recommend suitable searching and sorting algorithms with justification.

Then implement the selected algorithms in Python with proper function definitions and brief docstrings explaining time complexity.

Code & Output:

```
Assignment_12.5.py
153     "Task 4: Railway Ticket Reservation System"
154     # Generate python code for a Railway Ticket Reservation System that stores booking details
155     # including ticket ID, passenger name, train number, seat number, and travel date.
156     # The system must:
157     # Search tickets using ticket ID.
158     # Sort bookings based on travel date.
159     # Sort bookings based on seat number.
160     # First, recommend suitable searching and sorting algorithms with justification.
161     # Then implement the selected algorithms in Python with proper function definitions and
162     # brief docstrings explaining time complexity
163
164     "For the Railway Ticket Reservation System, I recommend the following algorithms:"
165     "1. Searching Algorithm: Hash Table - This allows for O(1) average time complexity for
166     searching tickets by ticket ID, making it efficient for quick lookups."
167     "2. Sorting Algorithm: Merge Sort - This algorithm is efficient for sorting large lists of
168     bookings based on travel date or seat number. It has a time complexity of O(n log n) and is
169     stable, which is beneficial for maintaining the order of records with equal keys."
170
171     class TicketReservationSystem:
172         def __init__(self):
173             self.bookings = []
174             self.ticket_id_map = {}
175
176         def add_booking(self, ticket_id, passenger_name, train_number, seat_number, travel_date):
177             booking = {
178                 'ticket_id': ticket_id,
179                 'passenger_name': passenger_name,
180                 'train_number': train_number,
181                 'seat_number': seat_number,
182                 'travel_date': travel_date
183             }
184             self.bookings.append(booking)
185             self.ticket_id_map[ticket_id] = booking
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
```

```
Assignment_12.5.py
167     class TicketReservationSystem:
168
169         def search_ticket(self, ticket_id):
170             ...
171             Searches for a ticket using the ticket ID.
172             Time Complexity: O(1) - due to the use of a hash table for quick lookups.
173             Parameters:
174                 ticket_id: The ID of the ticket to search for.
175             Returns:
176                 dict: The booking details if found, otherwise None.
177             ...
178             return self.ticket_id_map.get(ticket_id)
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
206
207
208
209
210
211
212
213
214
215
216
217
```

```
Assignment_12.5.py
Assignment_12.5.py > ...
207 def merge_sort(arr, key):
208     if len(arr) <= 1:
209         return arr
210
211     mid = len(arr) // 2
212     left_half = merge_sort(arr[:mid], key)
213     right_half = merge_sort(arr[mid:], key)
214
215     return merge(left_half, right_half, key)
216
217 def merge(left, right, key):
218     merged = []
219     left_index = right_index = 0
220
221     while left_index < len(left) and right_index < len(right):
222         if key(left[left_index]) < key(right[right_index]):
223             merged.append(left[left_index])
224             left_index += 1
225         else:
226             merged.append(right[right_index])
227             right_index += 1
228
229     merged.extend(left[left_index:])
230     merged.extend(right[right_index:])
231
232     return merged
233
234 # Example usage:
235 if __name__ == "__main__":
236     system = TicketReservationSystem()
237     system.add_booking(1, "Alice", "Train A", "1A", "2024-07-01")
238     system.add_booking(2, "Bob", "Train B", "2B", "2024-07-02")
239     system.add_booking(3, "Charlie", "Train A", "1B", "2024-07-01")
240
241     print("Search for ticket ID 2:", system.search_ticket(2))
242     system.sort_bookings_by_date()
243     print("Bookings sorted by travel date:", system.bookings)
244     system.sort_bookings_by_seat_number()
245     print("Bookings sorted by seat number:", system.bookings)

Code
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5> python -u e:
ted CodingAssignment_12.5\Assignment_12.5.py
search for ticket ID 2: {'ticket_id': 2, 'passenger_name': 'Bob', 'train_number':
': 'Train B', 'seat_number': '2B', 'travel_date': '2024-07-02'}
Bookings sorted by travel date: [{"ticket_id": 3, "passenger_name": "Charlie", "train_number": "Train A", "seat_number": "1B", "travel_date": "2024-07-01"}, {"ticket_id": 1, "passenger_name": "Alice", "train_number": "Train A", "seat_number": "1A", "travel_date": "2024-07-01"}, {"ticket_id": 2, "passenger_name": "Bob", "train_number": "Train B", "seat_number": "2B", "travel_date": "2024-07-02"}]
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5>
```

Explanation:

Binary Search is chosen for efficient ticket lookup when records are sorted by ticket ID, reducing search time to $O(\log n)$. Sorting is done using a stable sorting method (similar to Merge Sort Behaviour) to organize bookings by date or seat number. This ensures structured record management. AI assistance helped select efficient algorithms and implement clean functions.

Task 5: Smart Hostel Room Allocation System

Prompt:

Generate Python code for a Smart Hostel Room Allocation System storing student_id, room_number, floor, and allocation_date.

The system must:

- Search allocation details using student_id
 - Sort records based on room_number
 - Sort records based on allocation_date

Recommend suitable algorithms and implement them with brief complexity documentation.

Code & Output:

Assignment_12.5.py

```
Assignment_12.5.py > @merge
Assignment_12.5.py

245     "Task 5: Smart Hostel Room Allocation System"
246
247     # generate Python code for a Smart Hostel Room Allocation system storing student_id, room_number,
248     # floor, and allocation_date.
249     # The system must:
250     # Search allocation details using student_id
251     # Sort records based on room_number
252     # Sort records based on allocation_date
253     # Recommend suitable algorithms and implement them with brief complexity documentation.
254
255     "Algorithm Recommendation:"
256     "For the Smart Hostel Room Allocation System, I recommend the following algorithms:"
257     "1. Searching Algorithm: Hash Table - This allows for O(1) average time complexity for searching
258     allocation details by student_id, making it efficient for quick lookups."
259     "2. Sorting Algorithm: Merge Sort - This algorithm is efficient for sorting large lists of
260     allocations based on room number or allocation date. It has a time complexity of O(n log n) and is
261     stable, which is beneficial for maintaining the order of records with equal keys."
262
263     class HostelRoomAllocationSystem:
264         def __init__(self):
265             self.allocations = []
266             self.student_id_map = {}
267
268         def add_allocation(self, student_id, room_number, floor, allocation_date):
269             allocation = {
270                 'student_id': student_id,
271                 'room_number': room_number,
272                 'floor': floor,
273                 'allocation_date': allocation_date
274             }
275             self.allocations.append(allocation)
276             self.student_id_map[student_id] = allocation
277
278         def search_allocation(self, student_id):
279             ...
280
281             Searches for allocation details using the student ID.
282             Time complexity: O(1) - due to the use of a hash table for quick lookups.
283             Documentation:
```

Assignment_12.5

```
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5> python -u "e:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5\Assignment_12.5.py"
y
Search for student ID ?> 1
[[{"student_id": 1, "room_number": "102", "floor": "1st Floor", "allocation_date": "2024-07-01"}, {"student_id": 2, "room_number": "102", "floor": "1st Floor", "allocation_date": "2024-07-02"}, {"student_id": 3, "room_number": "103", "floor": "1st Floor", "allocation_date": "2024-07-01"}, {"student_id": 4, "room_number": "103", "floor": "2nd Floor", "allocation_date": "2024-07-01"}], [{"student_id": 1, "room_number": "101", "floor": "1st Floor", "allocation_date": "2024-07-01"}, {"student_id": 2, "room_number": "102", "floor": "1st Floor", "allocation_date": "2024-07-02"}]
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5>
```

```
File Edit Selection View Go Run Terminal Help ← → Assignment_12.5
```

Welcome Assignment_12.5.py

```
Assignment_12.5.py > @merge
258 class HostelRoomAllocationSystem:
259     def search_allocation(self, student_id):
260         Parameters:
261             student_id: The ID of the student to search for.
262         Returns:
263             dict: The allocation details if found, otherwise None.
264
265         return self.student_id_map.get(student_id)
266
267     def sort_allocations_by_room_number(self):
268         ...
269
270         Sorts records based on room number using Merge Sort.
271
272         Time Complexity: O(n log n) - where n is the number of allocations.
273
274         ...
275
276         self.allocations = merge_sort(self.allocations, key=lambda x: x['room_number'])
277
278     def sort_allocations_by_date(self):
279         ...
280
281         Sorts records based on allocation date using Merge Sort.
282
283         Time complexity: O(n log n) - where n is the number of allocations.
284
285         ...
286
287         self.allocations = merge_sort(self.allocations, key=lambda x: x['allocation_date'])
288
289     def merge_sort(arr, key):
290         if len(arr) <= 1:
291             return arr
292
293         mid = len(arr) // 2
294         left_half = merge_sort(arr[:mid], key)
295         right_half = merge_sort(arr[mid:], key)
296
297         return merge(left_half, right_half, key)
298     def merge(left, right, key):
299         merged = []
300         left_index = right_index = 0
301
302         while left_index < len(left) and right_index < len(right):
303             if key(left[left_index]) < key(right[right_index]):
```

Assignment_12.5

Code

```
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5> python -u "e:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5\Assignment_12.5.py"
y
Search for student ID 2: {'student_id': 2, 'room_number': '102', 'floor': '1st Floor', 'allocation_date': '2024-07-02'}
Allocations sorted by room number: [{'student_id': 1, 'room_number': '101', 'floor': '1st Floor', 'allocation_date': '2024-07-01'}, {'student_id': 2, 'room_number': '102', 'floor': '1st Floor', 'allocation_date': '2024-07-02'}, {'student_id': 3, 'room_number': '103', 'floor': '2nd Floor', 'allocation_date': '2024-07-01'}]
Allocations sorted by allocation date: [('student_id': 3, 'room_number': '103', 'floor': '2nd Floor', 'allocation_date': '2024-07-01'), ('student_id': 1, 'room_number': '101', 'floor': '1st Floor', 'allocation_date': '2024-07-01'), ('student_id': 2, 'room_number': '102', 'floor': '1st Floor', 'allocation_date': '2024-07-02')]
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5>
```

```

Assignment_12.py > ⚡ merge
305     return merge(left_half, right_half, key)
306 def merge(left, right, key):
307     merged = []
308     left_index = right_index = 0
309
310     while left_index < len(left) and right_index < len(right):
311         if key(left[left_index]) < key(right[right_index]):
312             merged.append(left[left_index])
313             left_index += 1
314         else:
315             merged.append(right[right_index])
316             right_index += 1
317
318     merged.extend(left[left_index:])
319     merged.extend(right[right_index:])
320
321     return merged
322
# Example usage:
323 if __name__ == "__main__":
324     system = HostelRoomAllocationSystem()
325     system.add_allocation(1, '101', '1st Floor', '2024-07-01')
326     system.add_allocation(2, '102', '1st Floor', '2024-07-02')
327     system.add_allocation(3, '103', '2nd Floor', '2024-07-01')
328
329     print("Search for student ID 2:", system.search_allocation(2))
330     system.sort_allocations_by_room_number()
331     print("Allocations sorted by room number:", system.allocations)
332     system.sort_allocations_by_date()
333     print("Allocations sorted by allocation date:", system.allocations)
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355

```

PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12> python -u "e:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12\Assignment_12.py"

Search for student ID 2: {"student_id": 2, "room_number": "102", "floor": "1st Floor", "allocation_date": "2024-07-02"}
Allocations sorted by room number: [{"student_id": 1, "room_number": "101", "floor": "1st Floor", "allocation_date": "2024-07-01"}, {"student_id": 2, "room_number": "102", "floor": "1st Floor", "allocation_date": "2024-07-02"}, {"student_id": 3, "room_number": "103", "floor": "2nd Floor", "allocation_date": "2024-07-01"}]
Allocations sorted by allocation date: [{"student_id": 3, "room_number": "103", "floor": "2nd Floor", "allocation_date": "2024-07-01"}, {"student_id": 1, "room_number": "101", "floor": "1st Floor", "allocation_date": "2024-07-01"}, {"student_id": 2, "room_number": "102", "floor": "1st Floor", "allocation_date": "2024-07-02"}]

Explanation:

Binary Search allows fast lookup of student allocation records when sorted by student ID. Sorting by room number or allocation date organizes hostel data efficiently. These algorithms ensure scalability for large datasets. AI guidance supported correct structure and optimized implementation.

Task 6: Online Movie Streaming Platform

Prompt:

Generate Python code for a Movie Streaming Platform storing movie_id, title, genre, rating, and release_year.

The system must:

- Search movies by movie_id
- Sort movies based on rating
- Sort movies based on release_year

Recommend suitable algorithms and implement them.

Code & Output:

```
File Edit Selection View Go Run Terminal Help ← → Assignment_12.5
```

```
Assignment_12.5.py ...
```

```
334
335     """Task 6: Online Movie Streaming Platform"""
336     # Generate Python code for a Movie Streaming Platform storing movie_id, title, genre, rating,
337     # and release_year.
338     # The system must:
339     # Search movies by movie_id
340     # Sort movies based on rating
341     # Sort movies based on release_year
342     # Recommend suitable algorithms and implement them.
```

```
343
344     """Algorithm Recommendation:"""
345     """For the Online Movie Streaming Platform, I recommend the following algorithms:"""
346     """1. Searching Algorithm: Hash Table - This allows for O(1) average time complexity for
347     searching movies by movie_id, making it efficient for quick lookups."""
348     """2. Sorting Algorithm: Merge Sort - This algorithm is efficient for sorting large lists of
349     movies based on rating or release_year. It has a time complexity of O(n log n) and is stable,
350     which is beneficial for maintaining the order of records with equal keys."""
351
352     class MovieStreamingPlatform:
353         def __init__(self):
354             self.movies = []
355             self.movie_id_map = {}
356
357         def add_movie(self, movie_id, title, genre, rating, release_year):
358             movie = {
359                 'movie_id': movie_id,
360                 'title': title,
361                 'genre': genre,
362                 'rating': rating,
363                 'release_year': release_year
364             }
365             self.movies.append(movie)
366             self.movie_id_map[movie_id] = movie
367
368         def search_movie(self, movie_id):
369             ...
370             """Searches for a movie using the movie ID.
```

```
File Edit Selection View Go Run Terminal Help ← → Assignment_12.5
```

```
Assignment_12.5.py ...
```

```
348     class MovieStreamingPlatform:
349         def search_movie(self, movie_id):
350             ...
351             """Searches for a movie using the movie ID.
352             Time complexity: O(1) - due to the use of a hash table for quick lookups.
353             Parameters:
354                 movie_id: The ID of the movie to search for.
355             Returns:
356                 dict: The movie details if found, otherwise None.
357             """
358             return self.movie_id_map.get(movie_id)
359
360         def sort_movies_by_rating(self):
361             ...
362             """Sorts movies based on rating using Merge Sort.
363             Time complexity: O(n log n) - where n is the number of movies.
364             """
365             self.movies = merge_sort(self.movies, key=lambda x: x['rating'])
366
367         def sort_movies_by_release_year(self):
368             ...
369             """Sorts movies based on release year using Merge Sort.
370             Time complexity: O(n log n) - where n is the number of movies.
371             """
372             self.movies = merge_sort(self.movies, key=lambda x: x['release_year'])
373
374         def merge_sort(arr, key):
375             if len(arr) <= 1:
376                 return arr
377
378             mid = len(arr) // 2
379             left_half = merge_sort(arr[:mid], key)
380             right_half = merge_sort(arr[mid:], key)
381
382             return merge(left_half, right_half, key)
383
384         def merge(left, right, key):
385             merged = []
386             left_index = right_index = 0
387
388             while left_index < len(left) and right_index < len(right):
389                 if left[left_index][key] < right[right_index][key]:
390                     merged.append(left[left_index])
391                     left_index += 1
392                 else:
393                     merged.append(right[right_index])
394                     right_index += 1
395
396             merged.extend(left[left_index:])
397             merged.extend(right[right_index:])
398
399             return merged
```

The screenshot shows a code editor with a Python file named `Assignment_12.5.py`. The code implements a merge sort algorithm for a list of movies based on their rating. It includes a main function that adds movies to a platform and then sorts them by rating and release year. The terminal window to the right shows the command `python -u e:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5\Assignment_12.5.py` being run, and it displays the sorted movie data.

```

File Edit Selection View Go Run Terminal Help ← →
Assignment_12.5.py ×
Assignment_12.5
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5> python -u e:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5\Assignment_12.5.py
Search for movie ID 2: {'movie_id': 2, 'title': 'The Matrix', 'genre': 'Action', 'rating': 8.7, 'release_year': 1999}
Movies sorted by rating: [{"movie_id": 3, "title": "Interstellar", "genre": "Sci-Fi", "rating": 8.6, "release_year": 2014}, {"movie_id": 2, "title": "The Matrix", "genre": "Action", "rating": 8.7, "release_year": 1999}, {"movie_id": 1, "title": "Inception", "genre": "Sci-Fi", "rating": 8.8, "release_year": 2010}]
Movies sorted by release year: [{"movie_id": 2, "title": "The Matrix", "genre": "Action", "rating": 8.7, "release_year": 1999}, {"movie_id": 1, "title": "Inception", "genre": "Sci-Fi", "rating": 8.8, "release_year": 2010}, {"movie_id": 3, "title": "Interstellar", "genre": "Sci-Fi", "rating": 8.6, "release_year": 2014}]
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5>

```

Explanation:

Binary Search is suitable for quick movie lookup when sorted by movie ID. Sorting by rating helps rank movies, while sorting by release year organizes content chronologically. These algorithms ensure efficient data management for streaming platforms. AI assistance improved modularity and clarity.

Task 7: Smart Agriculture Crop Monitoring System

Prompt:

Generate Python code for a Crop Monitoring System storing `crop_id`, `crop_name`, `soil_moisture_level`, `temperature`, and `yield_estimate`.

The system must:

- Search crop details using `crop_id`
- Sort crops based on `soil_moisture_level`
- Sort crops based on `yield_estimate`

Recommend suitable algorithms and implement them.

Code & Output:

File Edit Selection View Go Run Terminal Help ← →

Assignment_12.5.py

```
426 # Task 2: Smart Agriculture Crop Monitoring System"
427 # Generate Python code for a Crop Monitoring System storing crop_id, crop_name,
428 # soil_moisture_level, temperature, and yield_estimate.
429 # The system must:
430 # Search crop details using crop_id
431 # Sort crops based on soil_moisture_level
432 # Sort crops based on yield_estimate
433 # Recommend suitable algorithms and implement them.
434 #Algorithm Recommendation:
435 "For the Smart Agriculture Crop Monitoring System, I recommend the following algorithms:"
436 "1. Searching Algorithm: Hash Table - This allows for O(1) average time complexity for
437 # searching crop details by crop_id, making it efficient for quick lookups."
438 "2. Sorting Algorithm: Merge Sort - This algorithm is efficient for sorting large lists of
439 # crops based on soil moisture level or yield_estimate. It has a time complexity of O(n log n)
440 # and is stable, which is beneficial for maintaining the order of records with equal keys."
441 class CropMonitoringSystem:
442     def __init__(self):
443         self.crops = []
444         self.crop_id_map = {}
445
446     def add_crop(self, crop_id, crop_name, soil_moisture_level, temperature, yield_estimate):
447         crop = {
448             'crop_id': crop_id,
449             'crop_name': crop_name,
450             'soil_moisture_level': soil_moisture_level,
451             'temperature': temperature,
452             'yield_estimate': yield_estimate
453         }
454         self.crops.append(crop)
455         self.crop_id_map[crop_id] = crop
456
457     def search_crop(self, crop_id):
458         ...
459
460         Searches for crop details using the crop ID.
461         Time complexity: O(1) - due to the use of a hash table for quick lookups.
462         Parameters:
463             crop_id: The ID of the crop to search for.
464
465         Returns:
466             dict: The crop details if found, otherwise None.
467             ...
468
469         return self.crop_id_map.get(crop_id)
```

File Edit Selection View Go Run Terminal Help ← →

Assignment_12.5

```
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5> python -u "e:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5\Assignment_12.5.py"
Search for crop ID 2: {'crop_id': 2, 'crop_name': 'Corn', 'soil_moisture_level': 40, 'temperature': 28, 'yield_estimate': 1500}
Crops sorted by soil moisture level: [{'crop_id': 1, 'crop_name': 'Wheat', 'soil_moisture_level': 30, 'temperature': 25, 'yield_estimate': 1000}, {'crop_id': 3, 'crop_name': 'Rice', 'soil_moisture_level': 35, 'temperature': 27, 'yield_estimate': 1200}, {'crop_id': 2, 'crop_name': 'Corn', 'soil_moisture_level': 40, 'temperature': 28, 'yield_estimate': 1500}]
Crops sorted by yield estimate: [{'crop_id': 1, 'crop_name': 'Wheat', 'soil_moisture_level': 30, 'temperature': 25, 'yield_estimate': 1000}, {'crop_id': 3, 'crop_name': 'Rice', 'soil_moisture_level': 35, 'temperature': 27, 'yield_estimate': 1200}, {'crop_id': 2, 'crop_name': 'Corn', 'soil_moisture_level': 40, 'temperature': 28, 'yield_estimate': 1500}]
```

File Edit Selection View Go Run Terminal Help ← →

Assignment_12.5.py

```
437 class CropMonitoringSystem:
438     def search_crop(self, crop_id):
439         ...
440
441         Searches for crop details using the crop ID.
442         Time complexity: O(1) - due to the use of a hash table for quick lookups.
443         Parameters:
444             crop_id: The ID of the crop to search for.
445             ...
446
447         Returns:
448             dict: The crop details if found, otherwise None.
449             ...
450
451         return self.crop_id_map.get(crop_id)
```

```
452
453     def sort_crops_by_soil_moisture(self):
454         ...
455
456         Sorts crops based on soil moisture level using Merge Sort.
457         Time Complexity: O(n log n) - where n is the number of crops.
458         ...
459
460         self.crops = merge_sort(self.crops, key=lambda x: x['soil_moisture_level'])
```

```
461
462     def sort_crops_by_yield_estimate(self):
463         ...
464
465         Sorts crops based on yield estimate using Merge Sort.
466         Time Complexity: O(n log n) - where n is the number of crops.
467         ...
468
469         self.crops = merge_sort(self.crops, key=lambda x: x['yield_estimate'])
```

```
470
471     def merge_sort(arr, key):
472         if len(arr) <= 1:
473             return arr
474
475         mid = len(arr) // 2
476         left_half = merge_sort(arr[:mid], key)
477         right_half = merge_sort(arr[mid:], key)
478
479         return merge(left_half, right_half, key)
```

```
480     def merge(left, right, key):
481         merged = []
482         left_index = right_index = 0
```

```

File Edit Selection View Go Run Terminal Help ← →
Assignment_12.5.py
Assignment_12.5.py > ...
466     def merge(left, right, key):
467         merged = []
468         left_index = right_index = 0
469
470         while left_index < len(left) and right_index < len(right):
471             if key(left[left_index]) < key(right[right_index]):
472                 merged.append(left[left_index])
473                 left_index += 1
474             else:
475                 merged.append(right[right_index])
476                 right_index += 1
477
478         merged.extend(left[left_index:])
479         merged.extend(right[right_index:])
480
481     return merged
482
483 # Example usage:
484 if __name__ == "__main__":
485     system = CropMonitoringSystem()
486     system.add_crop(1, 'wheat', 30, 25, 1000)
487     system.add_crop(2, 'corn', 40, 28, 1500)
488     system.add_crop(3, 'rice', 35, 27, 1200)
489
490     print("Search for crop ID 2:", system.search_crop(2))
491     system.sort_crops_by_soil_moisture()
492     print("Crops sorted by soil moisture level:", system.crops)
493     system.sort_crops_by_yield_estimate()
494     print("Crops sorted by yield estimate:", system.crops)
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516

```

PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5> python -u "e:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5\Assignment_12.5.py"

Search for crop ID 2: {crop_id: 2, 'crop_name': 'Corn', 'soil_moisture_level': 40, 'temperature': 28, 'yield_estimate': 1500}

Crops sorted by soil moisture level: [{crop_id: 1, 'crop_name': 'Wheat', 'soil_moisture_level': 30, 'temperature': 25, 'yield_estimate': 1000}, {crop_id: 3, 'crop_name': 'Rice', 'soil_moisture_level': 35, 'temperature': 27, 'yield_estimate': 1200}, {crop_id: 2, 'crop_name': 'Corn', 'soil_moisture_level': 40, 'temperature': 28, 'yield_estimate': 1500}]

Crops sorted by yield estimate: [{crop_id: 1, 'crop_name': 'Wheat', 'soil_moisture_level': 30, 'temperature': 25, 'yield_estimate': 1000}, {crop_id: 3, 'crop_name': 'Rice', 'soil_moisture_level': 35, 'temperature': 27, 'yield_estimate': 1200}, {crop_id: 2, 'crop_name': 'Corn', 'soil_moisture_level': 40, 'temperature': 28, 'yield_estimate': 1500}]

Explanation:

Binary Search enables efficient crop lookup after sorting by crop ID. Sorting by moisture or yield estimate helps farmers analyze data effectively. These algorithms ensure optimized performance for agricultural monitoring systems. AI-supported reasoning guided appropriate algorithm selection.

Task 8: Airport Flight Management System

Prompt:

Generate Python code for an Airport Flight Management System storing flight_id, airline_name, departure_time, arrival_time, and status.

The system must:

- Search flight details using flight_id
- Sort flights based on departure_time
- Sort flights based on arrival_time

Recommend suitable algorithms and implement them.

Code & Output:

The screenshot shows a Jupyter Notebook interface with two code cells and their outputs.

Code Cell 1:

```
Assignment_12.5.py
Assignment_12.5.py > ...
# Task B: Airport Flight Management System"
# Generate Python code for an Airport Flight Management System storing
# flight_id, airline_name, departure_time, arrival_time, and status.
# The system must:
# Search flight details using flight_id
# sort flights based on departure_time
# sort flights based on arrival_time
# Recommend suitable algorithms and implement them.
#Algorithm Recommendation:
#For the Airport Flight Management System, I recommend the following
algorithms:
#1. Searching Algorithm: Hash Table - This allows for O(1) average time
complexity for searching flight details by flight_id, making it efficient
for quick lookups.
#2. Sorting Algorithm: Merge Sort - This algorithm is efficient for sorting
large lists of flights based on departure_time or arrival_time. It has a
time complexity of O(n log n) and is stable, which is beneficial for
maintaining the order of records with equal keys.
class FlightManagementSystem:
    def __init__(self):
        self.flights = []
        self.flight_id_map = {}

    def add_flight(self, flight_id, airline_name, departure_time,
arrival_time, status):
        flight = {
            'flight_id': flight_id,
            'airline_name': airline_name,
            'departure_time': departure_time,
            'arrival_time': arrival_time,
            'status': status
        }
        self.flights.append(flight)
        self.flight_id_map[flight_id] = flight

    def search_flight(self, flight_id):
        ...

PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5> python -u "e:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5\Assignment_12.5.py"
Search for flight ID: 2: {'flight_id': 2, 'airline_name': 'Airline A', 'departure_time': '2024-07-01 11:00', 'status': 'Delayed'}
Flights sorted by departure time: [('flight_id': 2, 'airline_name': 'Airline B', 'departure_time': '2024-07-01 09:00', 'arrival_time': '2024-07-01 11:00', 'status': 'Delayed'), ('flight_id': 1, 'airline_name': 'Airline A', 'departure_time': '2024-07-01 10:00', 'arrival_time': '2024-07-01 12:00', 'status': 'On Time'), ('flight_id': 3, 'airline_name': 'Airline C', 'departure_time': '2024-07-01 11:00', 'arrival_time': '2024-07-01 13:00', 'status': 'On Time')]
Flights sorted by arrival time: [('flight_id': 2, 'airline_name': 'Airline B', 'departure_time': '2024-07-01 09:00', 'arrival_time': '2024-07-01 11:00', 'status': 'Delayed'), ('flight_id': 1, 'airline_name': 'Airline A', 'departure_time': '2024-07-01 10:00', 'arrival_time': '2024-07-01 12:00', 'status': 'On Time'), ('flight_id': 3, 'airline_name': 'Airline C', 'departure_time': '2024-07-01 11:00', 'arrival_time': '2024-07-01 13:00', 'status': 'On Time')]
```

Code Cell 2:

```
Assignment_12.5.py
Assignment_12.5.py > ...
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5>
```

The screenshot shows a Jupyter Notebook interface with two code cells and their outputs.

Code Cell 1:

```
File Edit Selection View Go Run Terminal Help ← →
Assignment_12.5.py ×
Assignment_12.5.py > ...
class FlightManagementSystem:
    def search_flight(self, flight_id):
        ...
        Searches for flight details using the flight ID.
        Time complexity: O(1) - due to the use of a hash table for quick lookups.
        Parameters:
            flight_id: The ID of the flight to search for.
        Returns:
            dict: The flight details if found, otherwise None.
        ...
        return self.flight_id_map.get(flight_id)

    def sort_flights_by_departure_time(self):
        ...
        Sorts flights based on departure time using Merge Sort.
        Time complexity: O(n log n) - where n is the number of flights.
        ...
        self.flights = merge_sort(self.flights, key=lambda x: x['departure_time'])

    def sort_flights_by_arrival_time(self):
        ...
        Sorts flights based on arrival time using Merge Sort.
        Time Complexity: O(n log n) - where n is the number of flights.
        ...
        self.flights = merge_sort(self.flights, key=lambda x: x['arrival_time'])

    def merge_sort(arr, key):
        if len(arr) <= 1:
            return arr

        mid = len(arr) // 2
        left_half = merge_sort(arr[:mid], key)
        right_half = merge_sort(arr[mid:], key)

        return merge(left_half, right_half, key)
```

Code Cell 2:

```
Assignment_12.5.py ×
Assignment_12.5.py > ...
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5> python -u "e:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5\Assignment_12.5.py"
Search for flight ID 2: {'flight_id': 2, 'airline_name': 'Airline B', 'departure_time': '2024-07-01 10:00', 'arrival_time': '2024-07-01 11:00', 'status': 'Delayed'}
Flights sorted by departure time: [{"flight_id": 1, "airline_name": "Airline B", "departure_time": "2024-07-01 09:00", "arrival_time": "2024-07-01 11:00", "status": "Delayed"}, {"flight_id": 1, "airline_name": "Airline A", "departure_time": "2024-07-01 10:00", "arrival_time": "2024-07-01 12:00", "status": "On Time"}, {"flight_id": 3, "airline_name": "Airline C", "departure_time": "2024-07-01 11:00", "arrival_time": "2024-07-01 13:00", "status": "On Time"}]
Flights sorted by arrival time: [{"flight_id": 2, "airline_name": "Airline B", "departure_time": "2024-07-01 09:00", "arrival_time": "2024-07-01 11:00", "status": "Delayed"}, {"flight_id": 1, "airline_name": "Airline A", "departure_time": "2024-07-01 10:00", "arrival_time": "2024-07-01 12:00", "status": "On Time"}, {"flight_id": 3, "airline_name": "Airline C", "departure_time": "2024-07-01 11:00", "arrival_time": "2024-07-01 13:00", "status": "On Time"}]
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5>
```

The screenshot shows a code editor with two tabs: 'Assignment_12.5.py' and 'Assignment_12.5'. The code editor displays Python code for a flight management system, including merge sort and search functions. The terminal window shows the execution of the script and its output, which includes a search for flight ID 2 and sorted lists of flights by departure and arrival time.

```
Assignment_12.5.py
Assignment_12.5

575     def merge(left, right, key):
576         merged = []
577         left_index = right_index = 0
578
579         while left_index < len(left) and right_index < len(right):
580             if key(left[left_index]) < key(right[right_index]):
581                 merged.append(left[left_index])
582                 left_index += 1
583             else:
584                 merged.append(right[right_index])
585                 right_index += 1
586
587         merged.extend(left[left_index:])
588         merged.extend(right[right_index:])
589
590         return merged
591
592 if __name__ == "__main__":
593     system = FlightManagementSystem()
594     system.add_flight(1, "Airline A", "2024-07-01 10:00", "2024-07-01 12:00", "On Time")
595     system.add_flight(2, "Airline B", "2024-07-01 09:00", "2024-07-01 11:00", "Delayed")
596     system.add_flight(3, "Airline C", "2024-07-01 11:00", "2024-07-01 13:00", "On Time")
597
598     print("Search for flight ID 2:", system.search_flight(2))
599     system.sort_flights_by_departure_time()
600     print("Flights sorted by departure time:", system.flights)
601     system.sort_flights_by_arrival_time()
602     print("Flights sorted by arrival time:", system.flights)

Assignment_12.5
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5> python -u "E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5\Assignment_12.5.py"
Search for flight ID 2: {'flight_id': 2, 'airline_name': 'Airline B', 'departure_time': '2024-07-01 09:00', 'arrival_time': '2024-07-01 11:00', 'status': 'Delayed'}
Flights sorted by departure time: [{"flight_id": 2, "airline_name": "Airline B", "departure_time": "2024-07-01 09:00", "arrival_time": "2024-07-01 11:00", "status": "Delayed"}, {"flight_id": 1, "airline_name": "Airline A", "departure_time": "2024-07-01 10:00", "arrival_time": "2024-07-01 12:00", "status": "On Time"}, {"flight_id": 3, "airline_name": "Airline C", "departure_time": "2024-07-01 11:00", "arrival_time": "2024-07-01 13:00", "status": "On Time"}]
Flights sorted by arrival time: [{"flight_id": 2, "airline_name": "Airline B", "departure_time": "2024-07-01 09:00", "arrival_time": "2024-07-01 11:00", "status": "Delayed"}, {"flight_id": 1, "airline_name": "Airline A", "departure_time": "2024-07-01 10:00", "arrival_time": "2024-07-01 12:00", "status": "On Time"}, {"flight_id": 3, "airline_name": "Airline C", "departure_time": "2024-07-01 11:00", "arrival_time": "2024-07-01 13:00", "status": "On Time"}]
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5>
```

Explanation:

Binary Search ensures fast flight lookup when sorted by flight ID. Sorting by departure or arrival time organizes airport schedules effectively. These algorithms provide efficient record management. AI assistance supported correct algorithm implementation and modular function design.

Final Conclusion:

This lab demonstrated the implementation of sorting and searching algorithms using AI assistance across multiple real-world scenarios. Binary Search was selected for efficient searching due to its logarithmic time complexity, while Merge Sort was chosen for stable and reliable sorting performance. AI tools supported algorithm recommendation, code generation, and optimization. However, understanding algorithm efficiency and selecting appropriate techniques remain the developer's responsibility to ensure correct and scalable solutions.