

# **AI Assisted Coding**

## **Assignment – 12.1**

### **M.Sathwik || 2303A51483 || Batch:- 08**

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.

```

❸ Lab_12.py > 📁 merge_sort
1  # Generate a Python program that implements Merge Sort.
2  # Create a function merge_sort(arr) that sorts a list in ascending order.
3  # Include time complexity and space complexity in the function docstring.
4  # Add a few test cases to verify correctness.
5  def merge_sort(arr):
6      """
7          Sorts a list in ascending order using the Merge Sort algorithm.
8          Time Complexity: O(n log n) - where n is the number of elements in the list.
9          Space Complexity: O(n) - due to the temporary arrays used for merging.
10         Parameters:
11             arr (list): The list to be sorted.
12         Returns:
13             list: A new sorted list in ascending order.
14         """
15         if len(arr) <= 1:
16             return arr
17         mid = len(arr) // 2
18         left_half = merge_sort(arr[:mid])
19         right_half = merge_sort(arr[mid:])
20         return merge(left_half, right_half)
21     def merge(left, right):
22         """Merges two sorted lists into a single sorted list."""
23         merged = []
24         left_index = right_index = 0
25         while left_index < len(left) and right_index < len(right):
26             if left[left_index] < right[right_index]:
27                 merged.append(left[left_index])
28                 left_index += 1
29             else:
30                 merged.append(right[right_index])
31                 right_index += 1
32         # If there are remaining elements in left or right, add them to merged
33         merged.extend(left[left_index:])
34         merged.extend(right[right_index:])
35         return merged
36     # Test cases
37     print(merge_sort([38, 27, 43, 3, 9, 82, 10])) # Expected output: [3, 9, 10, 27, 38, 43, 82]
38     print(merge_sort([])) # Expected output: []
39     print(merge_sort([5])) # Expected output: [5]
40     print(merge_sort([5, 2, 9, 1, 5, 6])) # Expected output: [1, 2, 5, 5, 6, 9]
41     print(merge_sort([-3, 0, 2, 5, -1, 4, 1])) # Expected output: [-3, 0, 1, 2, 3, 4, 5]
42

```

## 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:

- o Prompt AI to create a function `binary_search(arr, target)`

returning the index of the target or -1 if not found.

- o Include docstrings explaining best, average, and

worst-case complexities.

- o Test with various inputs.

```
43
44  # Generate a Python function binary_search(arr, target) that returns the index of the target in a sorted list or -1 if not found.
45  # Include a docstring with best, average, and worst-case time complexities.
46  # Add multiple test cases to verify the function.
47 v def binary_search(arr, target):
48 v   """
49     Performs binary search on a sorted list to find the index of the target element.
50     Time Complexity:
51     - Best Case: O(1) - when the target is at the middle of the list.
52     - Average Case: O(log n) - where n is the number of elements in the list.
53     - Worst Case: O(log n) - when the target is not found or is at the end of the list.
54
55     Parameters:
56     arr (list): A sorted list in which to search for the target.
57     target: The element to search for in the list.
58
59     Returns:
60     int: The index of the target if found, otherwise -1.
61   """
62   left, right = 0, len(arr) - 1
63   while left <= right:
64     mid = left + (right - left) // 2
65     if arr[mid] == target:
66       return mid
67     elif arr[mid] < target:
68       left = mid + 1
69     else:
70       right = mid - 1
71   return -1
72 # Test cases
73 print(binary_search([1, 2, 3, 4, 5], 3)) # Expected output: 2
74 print(binary_search([1, 2, 3, 4, 5], 6)) # Expected output: -1
75 print(binary_search([], 1)) # Expected output: -1
76 print(binary_search([1], 1)) # Expected output: 0
77 print(binary_search([1, 2, 3, 4, 5], 1)) # Expected output: 0
78 print(binary_search([1, 2, 3, 4, 5], 5)) # Expected output: 4
79 |
```

### Task Description #3 (Real-Time Application – Inventory

#### Management System)

- Scenario: A retail store's inventory system contains thousands of products, each with attributes like product ID, name, price, and stock quantity. Store staff need to:
  1. Quickly search for a product by ID or name.
  2. Sort products by price or quantity for stock analysis.
- Task:
  - o Use AI to suggest the most efficient search and sort algorithms for this use case.
  - o Implement the recommended algorithms in Python.
  - o Justify the choice based on dataset size, update frequency, and performance requirements.

```

75
76 # You are building an inventory management system with thousands of products (product_id, name, price, stock).
77 # 1. Suggest the most efficient search and sorting algorithms for this scenario.
78 # 2. Implement them in Python to:
79 #   - search by product ID or name
80 #   - sort by price and stock quantity
81 # 3. Justify the algorithm choices based on dataset size, update frequency, and performance needs.
82 # For searching by product ID or name, a hash table (dictionary in Python) would be the most efficient data structure,
83 # as it provides average O(1) time complexity for lookups. For sorting by price and stock quantity,
84 # the Timsort algorithm (used by Python's built-in sorted() function) is efficient
85 # For real-world data and has a time complexity of O(n log n) in the worst case.
86 # 2. Implementation:
87
88 class Product:
89     def __init__(self, product_id, name, price, stock):
90         self.product_id = product_id
91         self.name = name
92         self.price = price
93         self.stock = stock
94
95 class Inventory:
96     def __init__(self):
97         self.products_by_id = {}
98         self.products_by_name = {}
99
100    def add_product(self, product):
101        self.products_by_id[product.product_id] = product
102        self.products_by_name[product.name] = product
103
104    def search_by_id(self, product_id):
105        return self.products_by_id.get(product_id, None)
106
107    def search_by_name(self, name):
108        return self.products_by_name.get(name, None)
109
110    def sort_by_price(self):
111        return sorted(self.products_by_id.values(), key=lambda x: x.price)
112
113    def sort_by_stock(self):
114        return sorted(self.products_by_id.values(), key=lambda x: x.stock)
115
116 # Test cases
117 inventory = Inventory()
118 inventory.add_product(Product(1, "Laptop", 999.99, 10))
119 inventory.add_product(Product(2, "Smartphone", 499.99, 50))
120 inventory.add_product(Product(3, "Headphones", 199.99, 100))
121
122 # Search by product ID

```

```

1 Click to add a breakpoint
118 inventory = Inventory()
119 inventory.add_product(Product(1, "Laptop", 999.99, 10))
120 inventory.add_product(Product(2, "Smartphone", 499.99, 50))
121 inventory.add_product(Product(3, "Headphones", 199.99, 100))
122 # Search by product ID
123 print(inventory.search_by_id(1).name) # Expected output: "Laptop"
124 print(inventory.search_by_id(4)) # Expected output: None
125 # Search by product name
126 print(inventory.search_by_name("Smartphone").price) # Expected output: 499.99
127 print(inventory.search_by_name("Tablet")) # Expected output: None
128 # Sort by price
129 sorted_by_price = inventory.sort_by_price()
130 print([product.name for product in sorted_by_price]) # Expected output: ["Headphones", "Smartphone", "Laptop"]
131 # Sort by stock
132 sorted_by_stock = inventory.sort_by_stock()
133 print([product.name for product in sorted_by_stock]) # Expected output: ["Laptop", "Smartphone", "Headphones"]
134 # Justification: The use of a hash table allows for efficient searching by product ID and name,
135 # which is crucial for an inventory management system with thousands of products.
136 # The Timsort algorithm is efficient for sorting large datasets and is optimized for real-world data,
137 # making it suitable for sorting by price and stock quantity. This approach ensures that the system can handle frequent updates
138 # and provide fast search and sort operations, meeting performance needs effectively.
139

```

## Task description #4: Smart Hospital Patient Management

### System

A hospital maintains records of thousands of patients with details such as patient ID, name, severity level, admission date, and bill amount.

Doctors and staff need to:

1. Quickly search patient records using patient ID or name.

2. Sort patients based on severity level or bill amount for prioritization and billing.

Student Task

- Use AI to recommend suitable searching and sorting algorithms.
- Justify the selected algorithms in terms of efficiency and suitability.
- Implement the recommended algorithms in Python.

```
Click to add or break point
141 # You are designing a Smart Hospital Patient Management System with thousands of records (patient_id, name, severity, admission_date, bill_amount).
142 # 1. Recommend suitable searching and sorting algorithms.
143 # 2. Justify the choices based on efficiency and real-time usage.
144 # 3. Implement the algorithms in Python to:
145 #     - search by patient ID or name
146 #     - sort by severity level and bill amount.
147 # 1. For searching by patient ID or name, a hash table (dictionary in Python) is recommended for its average O(1) time complexity for lookups.
148 # For sorting by severity level and bill amount, the Timsort algorithm (used by Python's built-in sorted() function) is efficient
149 # for real-world data and has a time complexity of O(n log n) in the worst case.
150 # 2. The use of a hash table allows for efficient searching by patient ID and name, which is crucial for a patient management system with thousands of records.
151 # The Timsort algorithm is optimized for real-world data and is suitable for sorting by severity level and bill amount,
152 # ensuring that the system can handle frequent updates and provide fast search and sort operations, meeting performance needs effectively.
153 # 3. Implementation:
154 class Patient:
155     def __init__(self, patient_id, name, severity, admission_date, bill_amount):
156         self.patient_id = patient_id
157         self.name = name
158         self.severity = severity
159         self.admission_date = admission_date
160         self.bill_amount = bill_amount
161 class PatientManagementSystem:
162     def __init__(self):
163         self.patients_by_id = {}
164         self.patients_by_name = {}
165
166     def add_patient(self, patient):
167         self.patients_by_id[patient.patient_id] = patient
168         self.patients_by_name[patient.name] = patient
169
170     def search_by_id(self, patient_id):
171         return self.patients_by_id.get(patient_id, None)
172
173     def search_by_name(self, name):
174         return self.patients_by_name.get(name, None)
175
176     def sort_by_severity(self):
177         return sorted(self.patients_by_id.values(), key=lambda x: x.severity)
178
179     def sort_by_bill_amount(self):
180         return sorted(self.patients_by_id.values(), key=lambda x: x.bill_amount)
181
182 # Test cases
183 pms = PatientManagementSystem()
184 pms.add_patient(Patient(1, "John Doe", 5, "2024-01-01", 1000.00))
185
186
187     return sorted(self.patients_by_id.values(), key=lambda x: x.bill_amount)
188
189 # Search by patient ID
190 print(pms.search_by_id(1).name) # Expected output: "John Doe"
191 print(pms.search_by_id(4)) # Expected output: None
192
193 # Search by patient name
194 print(pms.search_by_name("Jane Smith").bill_amount) # Expected output: 500.00
195 print(pms.search_by_name("Bob Brown")) # Expected output: None
196
197 # Sort by severity
198 sorted_by_severity = pms.sort_by_severity()
199 print([patient.name for patient in sorted_by_severity]) # Expected output: ["Jane Smith", "Alice Johnson", "John Doe"]
200
201 # Sort by bill amount
202 sorted_by_bill_amount = pms.sort_by_bill_amount()
203 print([patient.name for patient in sorted_by_bill_amount]) # Expected output: ["Jane Smith", "Alice Johnson", "John Doe"]
```

Task Description #5: University Examination Result Processing

## System

A university processes examination results for thousands of students containing roll number, name, subject, and marks. The system must:

1. Search student results using roll number.
2. Sort students based on marks to generate rank lists.

### Student Task

- Identify efficient searching and sorting algorithms using AI assistance.
- Justify the choice of algorithms.
- Implement the algorithms in Python.

```
200 # You are building a University Examination Result Processing System with thousands of records (roll_no, name, subject, marks).
201 # 1. Recommend efficient searching and sorting algorithms.
202 # 2. Justify the choices based on dataset size and performance.
203 # 3. Implement Python code to:
204 #     - search student results by roll number
205 #     - sort students by marks to generate rank lists.
206 # 1. For searching student results by roll number, a hash table (dictionary in Python) is recommended for its average O(1) time complexity for lookups.
207 # For sorting students by marks to generate rank lists, the Timsort algorithm (used by Python's built-in sorted() function)
208 # is efficient for real-world data and has a time complexity of O(n log n) in the worst case.
209 # 2. The use of a hash table allows for efficient searching by roll number, which is crucial for a result processing system with thousands of records.
210 # The Timsort algorithm is optimized for real-world data and is suitable for sorting by marks,
211 # ensuring that the system can handle frequent updates and provide fast search and sort operations, meeting performance needs effectively.
212 # 3. Implementation:
213 class StudentResult:
214     def __init__(self, roll_no, name, subject, marks):
215         self.roll_no = roll_no
216         self.name = name
217         self.subject = subject
218         self.marks = marks
219 class ResultProcessingSystem:
220     def __init__(self):
221         self.results_by_roll_no = {}
222
223     def add_result(self, result):
224         self.results_by_roll_no[result.roll_no] = result
225
226     def search_by_roll_no(self, roll_no):
227         return self.results_by_roll_no.get(roll_no, None)
228
229     def sort_by_marks(self):
230         return sorted(self.results_by_roll_no.values(), key=lambda x: x.marks, reverse=True)
231
232 # Test cases
233 rps = ResultProcessingSystem()
234 rps.add_result(StudentResult(1, "John Doe", "Math", 85))
235 rps.add_result(StudentResult(2, "Jane Smith", "Math", 90))
236 rps.add_result(StudentResult(3, "Alice Johnson", "Math", 80))
237
238 # Search by roll number
239 print(rps.search_by_roll_no(1).name) # Expected output: "John Doe"
240 print(rps.search_by_roll_no(4)) # Expected output: None
241
242 # Sort by marks to generate rank list
243 sorted_by_marks = rps.sort_by_marks()
244 print([(result.name, result.marks) for result in sorted_by_marks]) # Expected output: [("Jane Smith", 90), ("John Doe", 85), ("Alice Johnson", 80)]
```

## Task Description #6: Online Food Delivery Platform

An online food delivery application stores thousands of orders with order ID, restaurant name, delivery time, price, and order status. The platform needs to:

1. Quickly find an order using order ID.

## 2. Sort orders based on delivery time or price.

### Student Task

- Use AI to suggest optimized algorithms.
- Justify the algorithm selection.
- Implement searching and sorting modules in Python.

```
244 # You are designing an Online Food Delivery Platform with thousands of orders (order_id, restaurant_name, delivery_time, price, order_status).
245 # 1. Suggest optimized searching and sorting algorithms.
246 # 2. Justify the algorithm choices based on efficiency and scalability.
247 # 3. Implement Python code to:
248 #     - search an order by order_id
249 #     - sort orders by delivery_time and price.
250 # 1. For searching an order by order_id, a hash table (dictionary in Python) is recommended for its average O(1) time complexity for lookups.
251 # For sorting orders by delivery_time and price, the Timsort algorithm (used by Python's built-in sorted() function) is
252 # efficient for real-world data and has a time complexity of O(n log n) in the worst case.
253 # 2. The use of a hash table allows for efficient searching by order_id, which is crucial for an online food delivery platform with thousands of orders.
254 # The Timsort algorithm is optimized for real-world data and is suitable for sorting by delivery_time and price,
255 # ensuring that the system can handle frequent updates and provide fast search and sort operations, meeting performance needs effectively.
256 # 3. Implementation:
257 class Order:
258     def __init__(self, order_id, restaurant_name, delivery_time, price, order_status):
259         self.order_id = order_id
260         self.restaurant_name = restaurant_name
261         self.delivery_time = delivery_time
262         self.price = price
263         self.order_status = order_status
264 class FoodDeliveryPlatform:
265     def __init__(self):
266         self.orders_by_id = {}
267     def add_order(self, order):
268         self.orders_by_id[order.order_id] = order
269     def search_by_order_id(self, order_id):
270         return self.orders_by_id.get(order_id, None)
271     def sort_by_delivery_time(self):
272         return sorted(self.orders_by_id.values(), key=lambda x: x.delivery_time)
273     def sort_by_price(self):
274         return sorted(self.orders_by_id.values(), key=lambda x: x.price)
275 # Test cases
276 platform = FoodDeliveryPlatform()
277 platform.add_order(Order(1, "Pizza Place", "2024-01-01 18:00", 20.00, "Delivered"))
278 platform.add_order(Order(2, "Sushi Spot", "2024-01-01 19:00", 30.00, "In Progress"))
279 platform.add_order(Order(3, "Burger Joint", "2024-01-01 17:30", 15.00, "Delivered"))
280 # Search by order ID
281 print(platform.search_by_order_id(1).restaurant_name) # Expected output: "Pizza Place"
282 print(platform.search_by_order_id(4)) # Expected output: None
283 # Sort by delivery time
284 sorted_by_delivery_time = platform.sort_by_delivery_time()
285 print([(order.restaurant_name, order.delivery_time) for order in sorted_by_delivery_time])
286 # Expected output: [("Burger Joint", "2024-01-01 17:30"), ("Pizza Place", "2024-01-01 18:00"), ("Sushi Spot", "2024-01-01 19:00")]
287 # Sort by price
288 sorted_by_price = platform.sort_by_price()
289 print([(order.restaurant_name, order.price) for order in sorted_by_price]) # Expected output: [("Burger Joint", 15.00), ("Pizza Place", 20.00), ("Sushi Spot", 30.00)]
```