

AI ASSISTED CODING LAB

ASSIGNMENT 12.2

ENROLLMENT NO :2503A51L25

BATCH NO: 19

NAME: SRIRAMOJU AJAY

TASK1

TASK1 DESCRIPTION:-

- Use AI to generate a Python program that implements the Merge Sort algorithm.
- Instructions:
 - Prompt AI to create a function merge_sort(arr) that sorts a list in ascending order.
 - Ask AI to include time complexity and space complexity in the function docstring.
 - Verify the generated code with test cases.

PROMPT:-

Generate a python program with merge_sort(arr) implementing Merge Sort, include docstring with time/space complexity, do not modify input, and add simple tests/examples under __main__.

CODE:-

```
t1.py > ...
1  def merge_sort(art):
2  >     """ ...
23     if len(art) <= 1:
24         return art[:] # return a shallow copy
25
26     def merge(left, right):
27         i = j = 0
28         merged = []
29         while i < len(left) and j < len(right):
30             if left[i] <= right[j]:
31                 merged.append(left[i])
32                 i += 1
33             else:
34                 merged.append(right[j])
35                 j += 1
36         # append remaining elements
37         if i < len(left):
38             merged.extend(left[i:])
39 >         if j < len(right): ...
41         return merged
42
43     mid = len(art) // 2
44     left_sorted = merge_sort(art[:mid])
45     right_sorted = merge_sort(art[mid:])
46     return merge(left_sorted, right_sorted)
47
48
49 if __name__ == "__main__":
50     # Basic test cases to verify correctness
51 >     tests = [ ...
60
61     for t in tests:
62         result = merge_sort(t)
63         expected = sorted(t)
64         assert result == expected, f"merge_sort({t}) -> {result}, expected {expected}"
65     print("All tests passed. Examples:")
66
67     examples = [
68         [9, 7, 5, 3, 1, 2, 4, 6, 8],
69         [42, 42, 1, 0, -5],
70     ]
71     for ex in examples:
72         print(f"input: {ex} -> output: {merge_sort(ex)}")
```

OUTPUT:-

```
● PS C:\Users\khaja\OneDrive\Pictures\Screenshots\cyc\New folder\12.2> & C:/Users/khaja/anaconda3/python.exe  
Screenshots/cyc/New folder/12.2/t1.py"  
All tests passed. Examples:  
input: [9, 7, 5, 3, 1, 2, 4, 6, 8] -> output: [1, 2, 3, 4, 5, 6, 7, 8, 9]  
input: [42, 42, 1, 0, -5] -> output: [-5, 0, 1, 42, 42]  
○ PS C:\Users\khaja\OneDrive\Pictures\Screenshots\cyc\New folder\12.2>
```

OBSERVATION:-

In this task AI generated `merge_sort(arr)` along with a clear docstring that explains the algorithm and states time ($O(n \log n)$) and space ($O(n)$) complexity. The implementation returns a new sorted list without modifying the input and includes basic tests and example prints to verify correctness (empty list, single element, duplicates, negatives). This demonstrates how AI can quickly produce both working logic and useful documentation, plus ready-made tests so the implementation can be validated immediately.

TASK2

TASK2 DESCRIPTION:-

- 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.
 - Test with various inputs.

PROMPT:-

Create a Python program that implements `binary_search(arr, target)`: the function should assume `arr` is sorted in ascending order and return the index of `target` or -1 if not found, include a docstring explaining the algorithm and best/average/worst-case time complexities and space complexity, use an iterative approach, and include varied tests/examples under `if __name__ == "__main__":` to validate behavior.

CODE:-

```
t2.py > ...
1 def binary_search(arr, target):
2     """
27     left, right = 0, len(arr) - 1
28     while left <= right:
29         mid = left + (right - left) // 2
30         if arr[mid] == target:
31             return mid
32         if arr[mid] < target:
33             left = mid + 1
34         else:
35             right = mid - 1
36     return -1
37
38
39 if __name__ == "__main__":
40     # Test cases
41     tests = [
42         ([], 3, -1),           # empty list
43         ([1], 1, 0),           # single element found
44         ([1], 2, -1),          # single element not found
45         ([1, 2, 3, 4, 5], 3, 2), # middle element
46         ([1, 2, 3, 4, 5], 6, -1), # not present
47         ([-3, -1, 0, 2, 4], -1, 1), # negatives and positives
48         ([1, 2, 2, 2, 3], 2, "any"), # duplicates (any matching index is acceptable)
49     ]
50
51     for arr, tgt, expected in tests:
52         idx = binary_search(arr, tgt)
53         if expected == "any":
54             assert idx != -1 and arr[idx] == tgt, f"binary_search({arr}, {tgt}) -> {idx}, expected any index with value {tgt}"
55         else:
56             assert idx == expected, f"binary_search({arr}, {tgt}) -> {idx}, expected {expected}"
57
58     print("All tests passed. Examples:")
59
60     examples = [
61         ([1, 3, 5, 7, 9], 7),
62         ([0, 2, 4, 6, 8], 1),
63         ([10, 20, 30, 40], 25),
64     ]
65     for arr, tgt in examples:
66         print(f"input: {arr}, target: {tgt} -> index: {binary_search(arr, tgt)}")
```

OUTPUT:-

```
PS C:\Users\khaja\OneDrive\Pictures\Screenshots\cyc\New folder\12.2> & C:/Users/khaja/anaconda3/python.exe
Screenshots/cyc/New folder/12.2/t2.py
All tests passed. Examples:
input: [1, 3, 5, 7, 9], target: 7 -> index: 3
input: [0, 2, 4, 6, 8], target: 1 -> index: -1
input: [10, 20, 30, 40], target: 25 -> index: -1
PS C:\Users\khaja\OneDrive\Pictures\Screenshots\cyc\New folder\12.2>
```

OBSERVATION:-

AI produced an iterative `binary_search(arr, target)` that assumes a sorted input and returns the index or -1; the docstring lists best/average/worst time complexities and space complexity (best $O(1)$, average/worst $O(\log n)$, space $O(1)$). The file includes tests covering empty arrays, single-element cases, not-found cases and duplicates, making it straightforward to confirm correctness. The result shows AI speeds up development by supplying a concise, well-documented, and testable search routine.

TASK3

TASK3 DESCRIPTION:-

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:

- Use AI to suggest the most efficient search and sort algorithms for this use case.
- Implement the recommended algorithms in Python.
- Justify the choice based on dataset size, update frequency, and performance requirements.

PROMPT:-

Create a Python program with a Product dataclass (product_id, name, price, quantity) and an Inventory class implementing add_product, find_by_id, find_by_name(name, exact=True/False), sort_by_price, and sort_by_quantity; document complexity notes ($O(1)$ id lookup via dict, $O(n)$ substring search, $O(n \log n)$ sorting), handle duplicate IDs/names sensibly, and include an example dataset plus assertions and demonstration prints under `if name == "main"`.

CODE:-

```
t3.py > ...
1  from dataclasses import dataclass
2  from typing import List, Optional, Dict
3
4
5  @dataclass(frozen=True)
6  class Product:
7      product_id: str
8      name: str
9      price: float
10     quantity: int
11
12
13     class Inventory:
14         """
15         Inventory supporting:
16         - O(1) lookup by product ID using a dict index.
17         - O(1) exact-name lookup (if unique) using a name->list dict.
18         - substring name search in O(n) time.
19         - sorting by price or quantity in O(n log n) time.
20
21         Space complexity: O(n) for stored products and indexes.
22         """
23
24         def __init__(self, products: Optional[List[Product]] = None):
25             self._products: List[Product] = []
26             self._id_index: Dict[str, Product] = {}
27             self._name_index: Dict[str, List[Product]] = {}
28             if products:
29                 for p in products:
30                     self.add_product(p)
31
32         def add_product(self, product: Product) -> None:
33             """Add product and update indexes. If ID exists, replace product."""
34             # replace in list if exists
35             if product.product_id in self._id_index:
36                 # remove old instance from list and name index
37                 old = self._id_index[product.product_id]
38                 try:
39                     self._products.remove(old)
40                 except ValueError:
41                     pass
42                 lname = old.name.lower()
43                 self._name_index.get(lname, []).remove(old)
44
45             self._products.append(product)
```

```

t3.py > ...
13 class Inventory:
79     return sorted(self._products, key=lambda p: p.quantity, reverse=descending)
80
81
82 if __name__ == "__main__":
83     # Example dataset
84     samples = [
85         Product("P001", "USB Cable", 3.99, 150),
86         Product("P002", "Wireless Mouse", 15.49, 40),
87         Product("P003", "Keyboard", 22.0, 25),
88         Product("P004", "HDMI Cable", 7.5, 80),
89         Product("P005", "USB-C Adapter", 5.25, 60),
90         Product("P006", "Wireless Mouse", 17.99, 10), # duplicate name, different ID
91     ]
92
93     inv = Inventory(samples)
94
95     # Search by ID
96     p = inv.find_by_id("P003")
97     assert p is not None and p.name == "Keyboard"
98
99     # Exact name search (multiple results possible)
100    mice = inv.find_by_name("Wireless Mouse", exact=True)
101    assert len(mice) == 2 and all(m.name == "Wireless Mouse" for m in mice)
102
103    # Partial name search
104    usb_items = inv.find_by_name("usb", exact=False)
105    assert len(usb_items) >= 2 and all("usb" in it.name.lower() for it in usb_items)
106
107    # Sort by price ascending
108    by_price = inv.sort_by_price()
109    prices = [p.price for p in by_price]
110    assert prices == sorted(prices)
111
112    # Sort by quantity descending
113    by_qty_desc = inv.sort_by_quantity(descending=True)
114    qtys = [p.quantity for p in by_qty_desc]
115    assert qtys == sorted(qtys, reverse=True)
116
117    # Demonstration prints
118    print("Find by ID P004 ->", inv.find_by_id("P004"))
119    print("Exact name 'Wireless Mouse' ->", inv.find_by_name("Wireless Mouse"))
120    print("Partial name 'usb' ->", inv.find_by_name("usb", exact=False))
121    print("Sorted by price (asc) ->", [(p.product_id, p.price) for p in by_price])
122    print("Sorted by quantity (desc) ->", [(p.product_id, p.quantity) for p in by_qty_desc])

```

OUTPUT:-

```

PS C:\Users\khaja\OneDrive\Pictures\Screenshots\cyc\New folder\12.2> & C:/Users/khaja/anaconda3/python.exe "c:/Users/khaja/OneDrive/Pictures/Screenshots/cyc/New folder/12.2/t3.py"
Find by ID P004 -> Product(product_id='P004', name='HDMI Cable', price=7.5, quantity=80)
Exact name 'Wireless Mouse' -> [Product(product_id='P002', name='Wireless Mouse', price=15.49, quantity=40), Product(product_id='P006', name='Wireless Mouse', price=17.99, quantity=10)]
Partial name 'usb' -> [Product(product_id='P001', name='USB Cable', price=3.99, quantity=150), Product(product_id='P005', name='USB-C Adapter', price=5.25, quantity=60)]
Sorted by price (asc) -> [('P001', 3.99), ('P005', 5.25), ('P004', 7.5), ('P002', 15.49), ('P006', 17.99), ('P003', 22.0)]
Sorted by quantity (desc) -> [('P001', 150), ('P004', 80), ('P005', 60), ('P002', 40), ('P003', 25), ('P006', 10)]
PS C:\Users\khaja\OneDrive\Pictures\Screenshots\cyc\New folder\12.2>

```

OBSERVATION:-

AI created a Product dataclass and an Inventory class with `add_product`, `find_by_id`, `find_by_name` (exact and substring), `sort_by_price`, and `sort_by_quantity`, and documented the complexity and design decisions (dict for $O(1)$ ID lookup, name->list index, substring search $O(n)$, sorting $O(n \log n)$). An example dataset and assertions verify behavior including duplicate names and ID replacement. This highlights how AI can scaffold a small production-like module with indexing, documentation, and tests, enabling quick validation and iteration.