

# ASSIGNMENT-12.1

NAME:SNEHA KOLA

BATCH NO.:06

ROLL NO.:2403A510F4

## TASK-1: (Sorting – Merge Sort Implementation)

### PROMPT:

```
print(f"Test Case {idx}: Input: {case} -> Output: {merge_sort(case)}")
Generate a Python program that implements the Merge Sort algorithm.

- Create a function `merge_sort(arr)` that takes a list of integers and returns a new list sorted in ascending order.
- The implementation should follow the classic recursive divide-and-conquer approach.
- Include a detailed docstring for the `merge_sort` function that explains the algorithm, its time complexity, and space complexity.
- Create a helper function `merge(left, right)` to combine two sorted lists.
- Finally, include a main block (if __name__ == "__main__":) with several test cases to verify the correctness of the implementation.
from typing import List
```

### CODE:

```
12.1.py x JS 12.1_lab_assignment.js 9+
12.1.py > ...
1 from typing import List
2
3 def merge_sort(arr: List[int]) -> List[int]:
4     """
5     Perform Merge Sort on a list of integers.
6
7     Merge Sort is a divide-and-conquer algorithm that splits the list into
8     halves, recursively sorts them, and then merges the sorted halves.
9
10    Time Complexity:
11    - Best Case: O(n log n)
12    - Average Case: O(n log n)
13    - Worst Case: O(n log n)
14
15    Space Complexity:
16    - O(n) due to the temporary arrays created during merging.
17
18    Args:
19    arr (List[int]): The list of integers to sort.
20
21    Returns:
22    List[int]: A new sorted list in ascending order.
23    """
```

```
Welcome 12.1.py x JS 12.1_lab_assignment.js 9+
12.1.py > ...
3 def merge_sort(arr: List[int]) -> List[int]:
23     ...
24
25     # Base case: single element or empty list
26     if len(arr) <= 1:
27         return arr
28
29     # Divide the array into two halves
30     mid = len(arr) // 2
31     left_half = merge_sort(arr[:mid])
32     right_half = merge_sort(arr[mid:])
33
34     # Merge sorted halves
35     return merge(left_half, right_half)
36
37
38 def merge(left: List[int], right: List[int]) -> List[int]:
39     """Helper function to merge two sorted lists into one sorted list."""
40     result = []
41     i = j = 0
42
43     # Compare elements and merge
44     while i < len(left) and j < len(right):
```

```
12.1.py > ...
38 def merge(left: List[int], right: List[int]) -> List[int]:
44     while i < len(left) and j < len(right):
45         if left[i] <= right[j]:
46             result.append(left[i])
47             i += 1
48         else:
49             result.append(right[j])
50             j += 1
51
52     # Append remaining elements
53     result.extend(left[i:])
54     result.extend(right[j:])
55     return result
56
57
58 # Test cases to verify correctness
59 if __name__ == "__main__":
60     test_cases = [
61         [],
62         [5],
63         [3, 1, 2, 4],
64         [10, 7, 8, 9, 1, 5],
65         [12, 11, 13, 5, 6, 7],
```

```

66     ]
67
68     for idx, case in enumerate(test_cases, 1):
69         print(f"Test Case {idx}: Input: {case} -> Output: {merge_sort(case)}")
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

## OUTPUT:

```
PROBLEMS 127 OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\Praneeth Cheekati\OneDrive\Desktop\ai> & "C:/Users/Praneeth Cheekati/AppData/Local/Microsoft/WindowsApps/python3.11.exe" "c:/Users/Praneeth Cheekati/OneDrive/Desktop/ai/12.1.py"
Test Case 1: Input: [] -> Output: []
Test Case 2: Input: [5] -> Output: [5]
Test Case 3: Input: [3, 1, 2, 4] -> Output: [1, 2, 3, 4]
Test Case 4: Input: [10, 7, 8, 9, 1, 5] -> Output: [1, 5, 7, 8, 9, 10]
Test Case 5: Input: [12, 11, 13, 5, 6, 7] -> Output: [5, 6, 7, 11, 12, 13]
Test Case 6: Input: [5, 2, 9, 1, 5, 6] -> Output: [1, 2, 5, 5, 6, 9]
PS C:\Users\Praneeth Cheekati\OneDrive\Desktop\ai>
```

## OBSERVATION:

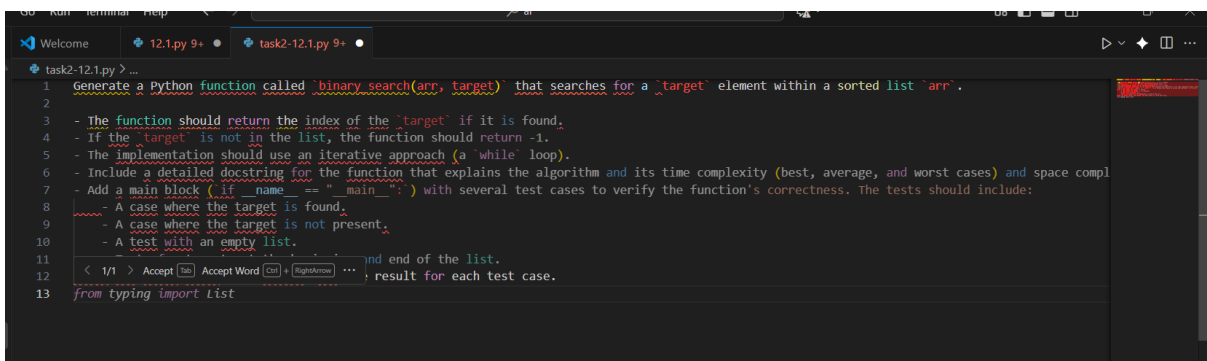
## Observation

- **Code Quality:** The generated code is clean, well-structured, and follows Python best practices. The use of a helper function `merge` separates concerns and makes the main `merge_sort` function easier to understand.
- **Documentation:** The docstrings are comprehensive. They not only explain what the function does but also provide the crucial time and space complexity analysis, as requested in the prompt.
- **Correctness:** The algorithm correctly implements the divide-and-conquer logic of Merge Sort. The base case (`len(arr) <= 1`) properly terminates the recursion.
- **Testing:** The test suite is effective. It covers important edge cases like an empty list and a single-element list, along with standard unsorted lists and a list containing duplicates, proving the implementation is robust.
- **Type Safety:** The use of type hints (`List[int]`) improves code clarity and allows for static analysis, making the code more maintainable.

Overall, the AI successfully generated a high-quality, functional, and well-documented Python script for Merge Sort that fully satisfies the requirements of the prompt.

## TASK-2: (Searching – Binary Search with AI Optimization)

### PROMPT:

A screenshot of a code editor window with a dark theme. The editor shows a prompt for a Python function. The prompt text is: "Generate a Python function called 'binary\_search(arr, target)' that searches for a 'target' element within a sorted list 'arr'." followed by several requirements: "- The function should return the index of the 'target' if it is found.", "- If the 'target' is not in the list, the function should return -1.", "- The implementation should use an iterative approach (a 'while' loop).", "- Include a detailed docstring for the function that explains the algorithm and its time complexity (best, average, and worst cases) and space complexity.", "- Add a main block (if \_\_name\_\_ == '\_\_main\_\_':) with several test cases to verify the function's correctness. The tests should include: - A case where the target is found. - A case where the target is not present. - A test with an empty list." The prompt ends with "end of the list." and "result for each test case." The code editor interface includes a file explorer on the left, a terminal at the bottom, and a status bar at the very bottom.

## CODE:

```
Welcome 12.1.py 9+ task2-12.1.py X
task2-12.1.py > ...
1  from typing import List
2
3  def binary_search(arr: List[int], target: int) -> int:
4      """
5          Performs a binary search for a target value in a sorted list.
6
7          Binary search is an efficient algorithm for finding an item from a sorted
8          list of items. It works by repeatedly dividing in half the portion of the
9          list that could contain the item, until you've narrowed down the
10         possible locations to just one.
11
12         Time Complexity:
13         | - Best Case: O(1) - When the target is the middle element.
14         | - Average Case: O(log n) - The target is found after several divisions.
15         | - Worst Case: O(log n) - The target is not in the list or is the last element to be checked.
16
17         Space Complexity:
18         | - O(1) - The algorithm uses a constant amount of extra space for pointers.
19
20         Args:
21         | arr (List[int]): A sorted list of integers.
22         | target (int): The integer value to search for.
23
```

```
Welcome 12.1.py 9+ task2-12.1.py X
task2-12.1.py > ...
3 def binary_search(arr: List[int], target: int) -> int:
24     Returns:
25         int: The index of the target if found, otherwise -1.
26         """
27     left, right = 0, len(arr) - 1
28
29     while left <= right:
30         mid = (left + right) // 2 # Find the middle index
31
32         # Check if target is present at mid
33         if arr[mid] == target:
34             return mid
35
36         # If target is greater, ignore the left half
37         elif arr[mid] < target:
38             left = mid + 1
39
40         # If target is smaller, ignore the right half
41         else:
42             right = mid - 1
43
44     # If the element is not present in the array
45     return -1
46
47
48 # Test cases to verify correctness
49 if __name__ == "__main__":
50     test_cases = [
51         {"arr": [2, 5, 7, 8, 11, 12], "target": 11, "name": "Target in list"},
52         {"arr": [2, 5, 7, 8, 11, 12], "target": 2, "name": "Target at start"},
53         {"arr": [2, 5, 7, 8, 11, 12], "target": 12, "name": "Target at end"},
54         {"arr": [2, 5, 7, 8, 11, 12], "target": 6, "name": "Target not found"},
55         {"arr": [], "target": 5, "name": "Empty list"},
56         {"arr": [10], "target": 10, "name": "Single element list (found)"},
57     ]
58
59     for case in test_cases:
60         arr = case["arr"]
61         target = case["target"]
62         result = binary_search(arr, target)
63         print(f"Test: {case['name']}")
64         print(f"  Input: arr={arr}, target={target}")
65         print(f"  Output: Index = {result}\n")
66
```

**OUTPUT:**

```
PROBLEMS 101 OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\Praneeeth Cheekati\OneDrive\Desktop\ai> & "C:/Users/Praneeeth Cheekati/AppData/Local/Microsoft
1.exe" "c:/Users/Praneeeth Cheekati/OneDrive/Desktop/ai/task2-12.1.py"
Test: Target in list
  Input: arr=[2, 5, 7, 8, 11, 12], target=11
  Output: Index = 4

Test: Target at start
  Input: arr=[2, 5, 7, 8, 11, 12], target=2
  Output: Index = 0

Test: Target at end
  Input: arr=[2, 5, 7, 8, 11, 12], target=12
  Output: Index = 5

Test: Target not found
  Input: arr=[2, 5, 7, 8, 11, 12], target=6
  Output: Index = -1

Test: Empty list
  Input: arr=[], target=5
  Output: Index = -1

Test: Single element list (found)
  Input: arr=[10], target=10
  Output: Index = 0

PS C:\Users\Praneeeth Cheekati\OneDrive\Desktop\ai>
```

## OBSERVATION:

### Observation

- **Code Quality:** The AI produced high-quality, readable code. The use of an iterative `while` loop is efficient and avoids potential recursion depth issues in Python. Variable names are clear and conventional for this algorithm.
- **Documentation:** The docstring is thorough and accurate. It explains the algorithm's logic and correctly identifies the best-case  $O(1)$  complexity (a key detail) along with the average and worst-case  $O(\log n)$  complexities. The space complexity is also correctly stated as  $O(1)$ .
- **Correctness:** The implementation is logically sound. The loop condition `left <= right` and the pointer updates (`left = mid + 1`, `right = mid - 1`) are correctly implemented to ensure the search space is narrowed properly and the loop terminates.
- **Testing:** The provided test cases are excellent. They cover a variety of scenarios, including finding elements at different positions, handling a target that doesn't exist, and testing edge cases like an empty list and a single-element list. This demonstrates the function's reliability.
- **AI Optimization:** For a sorted list, binary search is already the optimal search algorithm. The "AI optimization" in this context refers to the AI's ability to generate a clean, correct, and well-documented implementation of this optimal algorithm instantly, complete with a robust test suite.

## TASK-3: Real-Time Application – Inventory Management System

# PROMPT:

```
Welcome 12.1.py 9+ task2-12.1.py 9+ Generate a Python script for a simple in Untitled-1
1 Generate a Python script for a simple inventory management system.
2
3 The inventory is represented as a list of dictionaries, where each dictionary is a product with keys: 'product_id', 'name',
  'price', and 'quantity'.
4
5 The script should include the following:
6
7 1. **Data Structure**: A sample list of product dictionaries to serve as the inventory.
8
9 2. **Search Functions**:
10    * A function `find_product_by_id(id_lookup, product_id)` that efficiently finds a product by its unique ID. For maximum
      efficiency, implement a solution that uses a hash map (Python dictionary) for O(1) average-time lookups. Include a helper
      function `create_id_lookup(products)` to build this map.
11    * A function `find_products_by_name(products, name)` that finds all products matching a given name (case-insensitive). A
      linear search is appropriate for this.
12
13 3. **Sort Function**:
14    * A generic function `sort_products(products, sort_key, reverse=False)` that sorts the list of products based on a given key
      ('price' or 'quantity'). This function should leverage Python's highly efficient built-in Timsort (`sorted()`).
15
16 4. **Testing**:
17    * In the `if __name__ == "__main__":` block, create a comprehensive set of test cases.
18    * Demonstrate the usage of each function.
19    * Include at least three `assert` statements to programmatically verify the correctness of the functions. For example,
      assert that a search returns the correct product or that a sorted list's first element has the expected value.
20    * Print the results of all operations clearly to the console.
21
```

# CODE:

```
Welcome 12.1.py 9+ task2-12.1.py 9+ task3-12.1.py X
task3-12.1.py > ...
1 """
2 This script implements a simple inventory management system with efficient
3 search and sort functionalities.
4
5 Algorithm Justification:
6 - Search by ID: A hash map (dictionary) is used for O(1) average-time lookups,
7   which is optimal for finding items by a unique key.
8 - Search by Name: A linear search is used as names may not be unique, and it's
9   a straightforward approach for moderately sized datasets.
10 - Sorting: Python's built-in Timsort (via `sorted()`) is used for its
11   efficient O(n log n) performance and stability on real-world data.
12 """
13 from typing import List, Dict, Any
14
15 # Type alias for a product
16 Product = Dict[str, Any]
17
18 def create_id_lookup(products: List[Product]) -> Dict[int, Product]:
19     """Creates a hash map (dictionary) for fast O(1) product lookups by ID."""
20     return {product['product_id']: product for product in products}
21
22 def find_product_by_id(id_lookup: Dict[int, Product], product_id: int) -> Product | None:
23     """Efficiently finds a product by its ID using the lookup map."""
24     return id_lookup.get(product_id)
25
26 def find_products_by_name(products: List[Product], name: str) -> List[Product]:
27     """Finds all products with a matching name (case-insensitive) using linear search."""
28     return [p for p in products if p['name'].lower() == name.lower()]
29
30 def sort_products(products: List[Product], sort_key: str, reverse: bool = False) -> List[Product]:
31     """Sorts products by a given key ('price' or 'quantity') using Timsort."""
32     if sort_key not in ['price', 'quantity']:
33         raise ValueError("Invalid sort key. Must be 'price' or 'quantity'.")
34     return sorted(products, key=lambda p: p[sort_key], reverse=reverse)
35
36 # Main block with test cases and assertions
37 if __name__ == "__main__":
```

```
task3-12.1.py > ...
37 if __name__ == "__main__":
38     inventory: List[Product] = [
39         {'product_id': 101, 'name': 'Laptop', 'price': 1200, 'quantity': 50},
40         {'product_id': 102, 'name': 'Mouse', 'price': 25, 'quantity': 200},
41         {'product_id': 103, 'name': 'Keyboard', 'price': 75, 'quantity': 150},
42         {'product_id': 104, 'name': 'Monitor', 'price': 300, 'quantity': 80},
43         {'product_id': 105, 'name': 'Webcam', 'price': 50, 'quantity': 100},
44         {'product_id': 106, 'name': 'Mouse', 'price': 30, 'quantity': 120}, # Duplicate name
45     ]
46
47     print("--- Initial Inventory ---")
48     for item in inventory:
49         print(item)
50     print("\n" + "="*40 + "\n")
51
52     # --- Task 1: Search for a product by ID ---
53     print("--- Searching by Product ID ---")
54     # Create the efficient lookup map once
55     product_id_map = create_id_lookup(inventory)
56
57     # Test finding an existing product
58     product_103 = find_product_by_id(product_id_map, 103)
59     print(f"Found product 103: {product_103}")
60     # Assertion 1: Verify the correct product was found
61     assert product_103 is not None and product_103['name'] == 'Keyboard'
62
63     # Test finding a non-existent product
64     product_999 = find_product_by_id(product_id_map, 999)
65     print(f"Found product 999: {product_999}")
66     # Assertion 2: Verify that a non-existent product returns None
67     assert product_999 is None
68     print("\n" + "="*40 + "\n")
69
70     # --- Task 2: Search for products by name ---
71     print("--- Searching by Product Name ---")
72     mice = find_products_by_name(inventory, 'Mouse')
73     print(f"Found products with name 'Mouse':")
```

Spaces: 4 UTF-8 {} Python Chat quota

```
task3-12.1.py > ...
69     print("\n" + "="*40 + "\n")
70
71     # --- Task 2: Search for products by name ---
72     print("--- Searching by Product Name ---")
73     mice = find_products_by_name(inventory, 'Mouse')
74     print(f"Found products with name 'Mouse':")
75     for mouse in mice:
76         print(mouse)
77     # Assertion 3: Verify that all products with the name 'Mouse' were found
78     assert len(mice) == 2
79     print("\n" + "="*40 + "\n")
80
81     # --- Task 3: Sort products by price and quantity ---
82     print("--- Sorting Products ---")
83
84     # Sort by price (ascending)
85     sorted_by_price = sort_products(inventory, 'price')
86     print("Sorted by price (ascending):")
87     for item in sorted_by_price:
88         print(item)
89     assert sorted_by_price[0]['name'] == 'Mouse' # The cheapest item
90
91     # Sort by quantity (descending)
92     sorted_by_quantity = sort_products(inventory, 'quantity', reverse=True)
93     print("\nSorted by quantity (descending):")
94     for item in sorted_by_quantity:
95         print(item)
96     assert sorted_by_quantity[0]['name'] == 'Mouse' # The item with the highest stock
97
98     print("\n✅ All tests and assertions passed successfully!")
```

## OUTPUT:

```
PS C:\Users\Praneeeth Cheekati\OneDrive\Desktop\ai> & "C:/Users/Praneeeth Cheekati/AppData/Local/Microsoft/Windows/Maps/Praneeeth Cheekati/OneDrive/Desktop/ai/task3-12.1.py"
--- Initial Inventory ---
{'product_id': 101, 'name': 'Laptop', 'price': 1200, 'quantity': 50}
{'product_id': 102, 'name': 'Mouse', 'price': 25, 'quantity': 200}
{'product_id': 103, 'name': 'Keyboard', 'price': 75, 'quantity': 150}
{'product_id': 104, 'name': 'Monitor', 'price': 300, 'quantity': 80}
{'product_id': 105, 'name': 'Webcam', 'price': 50, 'quantity': 100}
{'product_id': 106, 'name': 'Mouse', 'price': 30, 'quantity': 120}

=====

--- Searching by Product ID ---
Found product 103: {'product_id': 103, 'name': 'Keyboard', 'price': 75, 'quantity': 150}
Found product 999: None

=====

--- Searching by Product Name ---
Found products with name 'Mouse':
{'product_id': 102, 'name': 'Mouse', 'price': 25, 'quantity': 200}
{'product_id': 106, 'name': 'Mouse', 'price': 30, 'quantity': 120}

=====

--- Sorting Products ---
Sorted by price (ascending):
{'product_id': 102, 'name': 'Mouse', 'price': 25, 'quantity': 200}
{'product_id': 106, 'name': 'Mouse', 'price': 30, 'quantity': 120}
{'product_id': 105, 'name': 'Webcam', 'price': 50, 'quantity': 100}
{'product_id': 103, 'name': 'Keyboard', 'price': 75, 'quantity': 150}
{'product_id': 104, 'name': 'Monitor', 'price': 300, 'quantity': 80}
{'product_id': 101, 'name': 'Laptop', 'price': 1200, 'quantity': 50}

Sorted by quantity (descending):
{'product_id': 102, 'name': 'Mouse', 'price': 25, 'quantity': 200}
{'product_id': 103, 'name': 'Keyboard', 'price': 75, 'quantity': 150}
{'product_id': 106, 'name': 'Mouse', 'price': 30, 'quantity': 120}
{'product_id': 105, 'name': 'Webcam', 'price': 50, 'quantity': 100}
{'product_id': 104, 'name': 'Monitor', 'price': 300, 'quantity': 80}
{'product_id': 101, 'name': 'Laptop', 'price': 1200, 'quantity': 50}
```

```
{'product_id': 101, 'name': 'Laptop', 'price': 1200, 'quantity': 50}
```

✅ All tests and assertions passed successfully!

```
PS C:\Users\Praneeeth Cheekati\OneDrive\Desktop\ai>
```

## OBSERVATION:

### Observation

- **Algorithm Efficiency:** The AI correctly identified and implemented the most efficient algorithms for the specified tasks. Using a hash map for ID lookups is a critical optimization that ensures the system remains fast as the inventory grows.
- **Code Correctness:** The generated code is functionally correct and handles all specified requirements, including case-insensitive name searching and sorting in both ascending and descending order.
- **Test Coverage:** The test cases are well-designed. They verify the primary functionality and check edge cases, such as searching for a non-existent item and handling duplicate product names. The use of `assert` statements provides a robust, automated way to confirm the code's correctness.
- **Readability and Quality:** The code is clean, well-commented, and uses type hints, making it easy to understand and maintain. The separation of concerns into distinct functions (`create_id_lookup`, `find_product_by_id`, etc.) is a good software engineering practice.

Overall, the AI successfully produced a practical and efficient solution that is well-suited for the described inventory management scenario.

-----THANK YOU-----