ASSIGNMENT-12.1

Name: P. Umesh Reddy Roll.no: 2403A510F9
Batch no: 06 Course: AI Assisted Coding

- ✓ #Task Description #1 (Sorting – Merge Sort Implementation)
  - Task: 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.
  - Expected Output:
    - A functional Python script implementing Merge Sort with proper documentation.

- ✓ #Prompt:

Generate a Python program that implements the Merge Sort algorithm.

- ✓ Code:

C: > Users > THIRUPATHI REDDY > Desktop > AIAC_f9 > Untitled-1.py > merge_sort

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return list(arr)  # ensure a new list is returned

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    # merge left and right
    merged = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1
    if i < len(left):
        merged.extend(left[i:])
    if j < len(right):
        merged.extend(right[j:])
    return merged

if __name__ == "__main__":
    # Test prompts and cases:
    # Prompt 1: empty list
    test1 = []
    # Prompt 2: reverse-sorted list
    test2 = [5, 4, 3, 2, 1]
    # Prompt 3: list with duplicates and mixed order
    test3 = [3, 1, 2, 3, 0]

    # Assertions
    assert merge_sort(test1) == [], "Test 1 failed"
    assert merge_sort(test2) == [1, 2, 3, 4, 5], "Test 2 failed"
    assert merge_sort(test3) == [0, 1, 2, 3, 3], "Test 3 failed"

    # Print outputs
    print("merge_sort([]) ->", merge_sort(test1))
    print("merge_sort([5,4,3,2,1]) ->", merge_sort(test2))
    print("merge_sort([3,1,2,3,0]) ->", merge_sort(test3))
    print("All tests passed.")
# ...existing code...
```

✓ Output:

```
> & C:/Python313/python.exe "c:/Users/THIRUPATHI REDDY/Desktop/AIAC_f9/Untitled-1.py"
merge_sort([]) -> []
merge_sort([5,4,3,2,1]) -> [1, 2, 3, 4, 5]
merge_sort([3,1,2,3,0]) -> [0, 1, 2, 3, 3]
All tests passed.
PS C:\Users\THIRUPATHI REDDY\Desktop\apps>
```

✓ Explanation:

- What it does: merge_sort(arr) returns a new list sorted in ascending order using merge sort.

- How it works: recursively split list, sort halves, then merge two sorted lists.
- Time complexity: O(n log n). Space complexity: O(n) (extra lists and recursion).
- Tests: three asserts for [], [5,4,3,2,1], and [3,1,2,3,0]; prints results and "All tests passed."
  - ✓ 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.
    • Expected Output:
    o Python code implementing binary search with AI-generated comments and docstrings
- ✓ #Prompt: create a binary search function that finds a target element in a sorted list.
- ✓ Code:

```python
def binary_search(arr, target):
    """Perform binary search on a sorted list.
    Returns:
        int: index of target in arr, or -1 if not found.
    Time complexity:
        Best-case: O(1)   (target found at middle)
        Average-case: O(log n)
        Worst-case: O(log n)
    Space complexity:
        O(1) (iterative, constant extra space)
    """
    lo = 0
    hi = len(arr) - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            lo = mid + 1
        else:
            hi = mid - 1
    return -1
if __name__ == "__main__":
    bs_test1 = [0, 1, 2, 3, 4]
    bs_test2 = [10, 20, 30, 40, 50]
    bs_test3 = [1, 2, 3, 4, 5]
    assert binary_search(bs_test1, 0) == 0, "BS Test 1 failed"
    assert binary_search(bs_test2, 30) == 2, "BS Test 2 failed"
    assert binary_search(bs_test3, 99) == -1, "BS Test 3 failed"
    print("binary_search(bs_test1, 0) ->", binary_search(bs_test1, 0))
    print("binary_search(bs_test2, 30) ->", binary_search(bs_test2, 30))
    print("binary_search(bs_test3, 99) ->", binary_search(bs_test3, 99))
    print("Binary search tests passed.")
```

✓ Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                Python
PS C:\Users\THIRUPATHI REDDY\Desktop\apps> & C:/Python313/python.exe "c:/Users/THIRUPATHI REDDY/Desktop/AIAC_f9/1
-02.py"
binary_search(bs_test1, 0) -> 0
binary_search(bs_test2, 30) -> 2
binary_search(bs_test3, 99) -> -1
Binary search tests passed.
PS C:\Users\THIRUPATHI REDDY\Desktop\apps>
```

✓ Explanation:

- Purpose: sorts a list in ascending order using merge sort and returns a new list.
- How: recursively split into halves, sort each half, then merge two sorted lists.
- Complexity: time O(n log n), space O(n) (extra lists + recursion).

✓ 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.

• Expected Output:

o A table mapping operation → recommended algorithm → justification.

o Working Python functions for searching and sorting the inventory

✓ Code:

```
from bisect import bisect_left, bisect_right
import heapq
def build_indices(products):
    id_index = {}
    name_index = {}
    name_tuples = []
    for p in products:
        id_index[p['id']] = p
        name_index.setdefault(p['name'], []).append(p)
        name_tuples.append((p['name'], p))
    name_tuples.sort(key=lambda t: t[0])
    sorted_names = [t[0] for t in name_tuples]
    return id_index, name_index, (sorted_names, name_tuples)
def search_by_id(id_index, pid):
    """O(1) average. Return product or None."""
    return id_index.get(pid)
def search_by_name_exact(name_index, name):
    """O(1) average. Return list (may be empty)."""
    return name_index.get(name, [])
def search_by_name_prefix(sorted_names_and_tuples, prefix):
    """
    Prefix search using bisect.
    Time: O(log n + k) where k is number of matches.
    """
    sorted_names, name_tuples = sorted_names_and_tuples
    # high bound using unicode max suffix to include all strings starting with prefix
    lo = bisect_left(sorted_names, prefix)
    hi = bisect_right(sorted_names, prefix + "\uffff")
    return [prod for _, prod in name_tuples[lo:hi]]
def sort_by_price(products, reverse=False):
    """Stable sort by price. Time: O(n log n)."""
    return sorted(products, key=lambda p: p['price'], reverse=reverse)
def sort_by_quantity(products, reverse=False):
```

```python
def sort_by_quantity(products, reverse=False):
    """Stable sort by quantity. Time: O(n log n)."""
    return sorted(products, key=lambda p: p['qty'], reverse=reverse)
def top_k_by_price(products, k):
    """Return top-k products by price. Time: O(n log k)."""
    return heapq.nlargest(k, products, key=lambda p: p['price'])
if __name__ == "__main__":
    # small example and quick checks
    products = [
        {'id': 'P001', 'name': 'Apple',  'price': 1.20, 'qty': 150},
        {'id': 'P002', 'name': 'Banana', 'price': 0.80, 'qty': 200},
        {'id': 'P003', 'name': 'Apple',  'price': 1.10, 'qty': 50},
        {'id': 'P004', 'name': 'Avocado','price': 2.50, 'qty': 30},
        {'id': 'P005', 'name': 'Blueberry','price': 3.00,'qty': 10},
    ]
    id_idx, name_idx, sorted_names_tuple = build_indices(products)
    print("search_by_id P003 ->", search_by_id(id_idx, 'P003'))
    print("search_by_name_exact 'Apple' ->", search_by_name_exact(name_idx, 'Apple'))
    print("search_by_name_prefix 'A' ->", search_by_name_prefix(sorted_names_tuple, 'A'))
    print("sorted by price ->", sort_by_price(products))
    print("top 2 by price ->", top_k_by_price(products, 2))
```

✓ Output:

```
top 2 by price -> [{'id': 'P005', 'name': 'Blueberry', 'price': 3.0, 'qty': 10}, {'id': 'P004', 'name': 'Avocado', 'price': 2.5, 'qty': 30}]
PS C:\Users\THIRUPATHI REDDY\Desktop\apps>
```

✓ Explanation:

- Purpose: utilities for fast inventory lookup and sorting (build indices, exact/prefix name search, id lookup, sort by price/qty, top-k prices).
- build_indices: builds id dict (id -> product), name dict (name -> list), and a name-sorted list for prefix queries. Time O(n log n) due to sorting.
- search_by_id: O(1) average lookup via dict.
- search_by_name_exact: O(1) average to get all products with a given name.
- search_by_name_prefix: uses bisect on the sorted names to find the prefix range; O(log n + k) where k = matches.
- sort_by_price / sort_by_quantity: use Python’s stable Timsort (sorted()), O(n log n).
- top_k_by_price: heapq.nlargest for top-k, O(n log k).
- Example: run the script to see sample index building and queries printed.