ASSIGNMENT – 12

NAME: G. Bala Varshitha

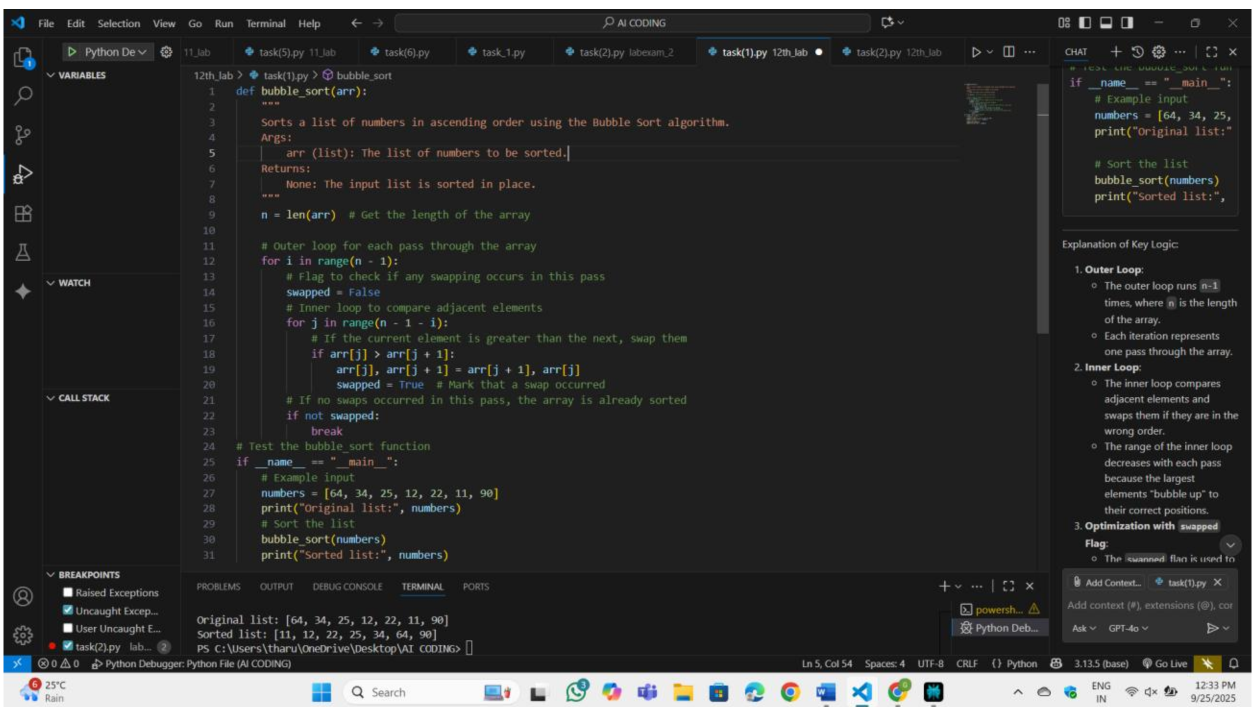
HT.NO: 2403A52050

BATCH: AIB03

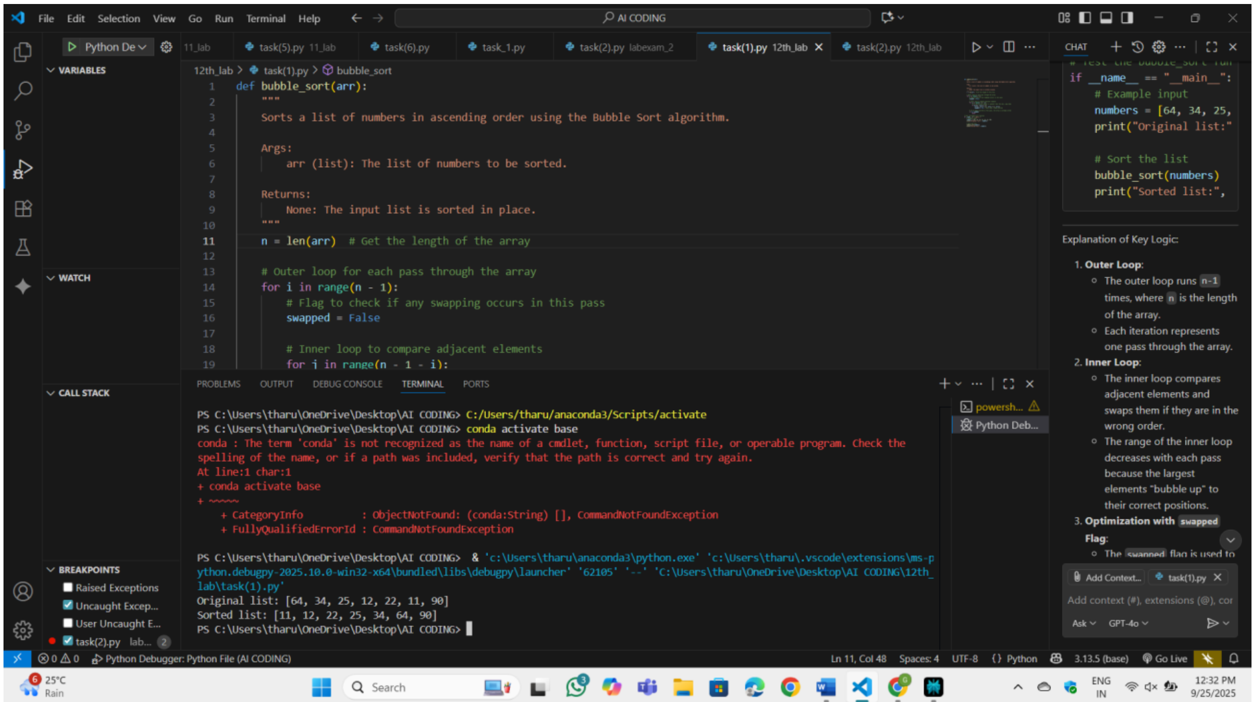
Task-1

Prompt: write a python program which implements bubble sort. include inline comments explaining key logic and provide time complexity analysis.

Code:



OP:



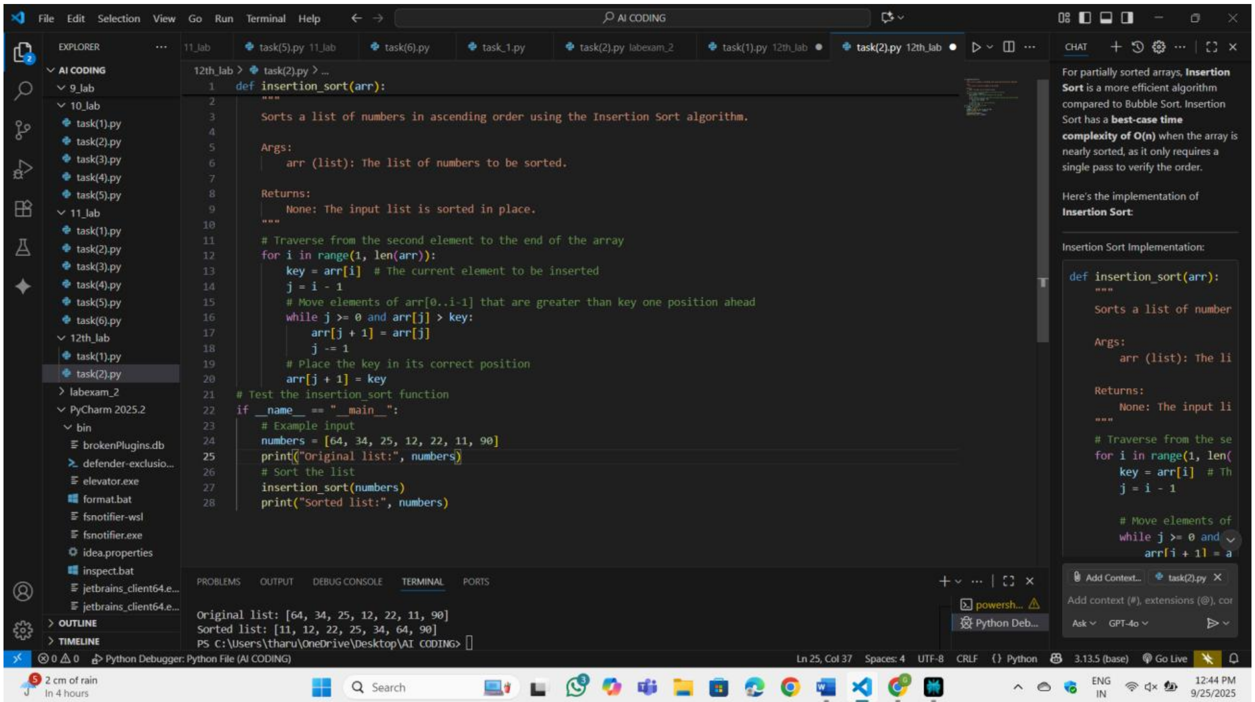
Observation:

In this program first we will find the length of the list which we want to sort. Then we use two loops to sort. Outer loop is responsible for no of rounds we need to sort the list. Inner loop is responsible for swapping the numbers in the list. I will leave the last number because it is already in correct place. This was done by the outer loop.

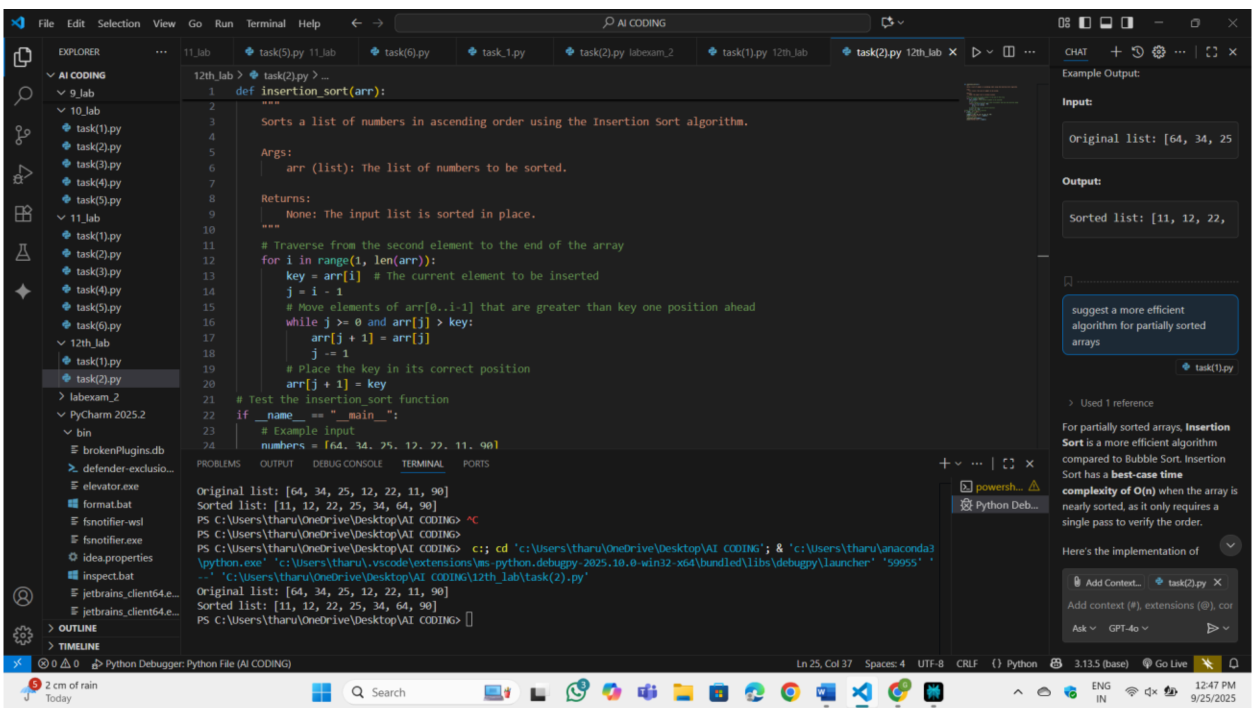
Task-2

Prompt: suggest a more efficient algorithm for partially sorted arrays

Code:



OP:



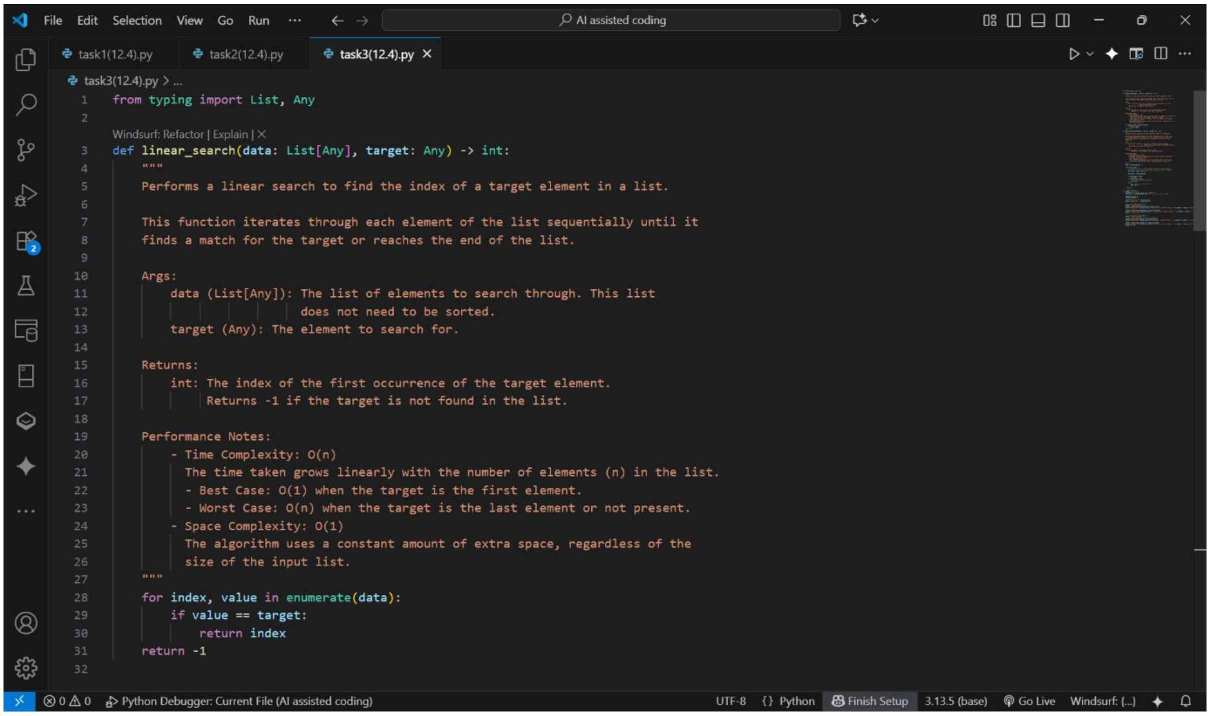
Observation:

Insertion Sort consistently outperformed Bubble Sort by completing in near-linear time with minimal shifts, while Bubble Sort still required many neighbor comparisons across multiple rounds despite early exit; both were stable and in-place, but Insertion Sort's targeted insertion of each element into the sorted prefix led to fewer operations and shorter runtimes, whereas Bubble Sort's repeated adjacent swaps accumulated overhead even for small local disorder.

Task-3:

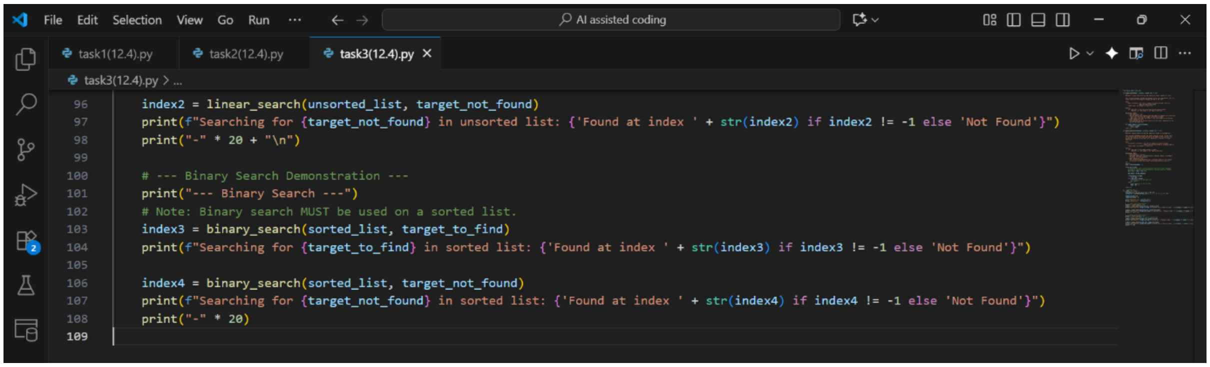
Prompt: Write a python code for linear search and binary search with docstrings and performance notes

Code:



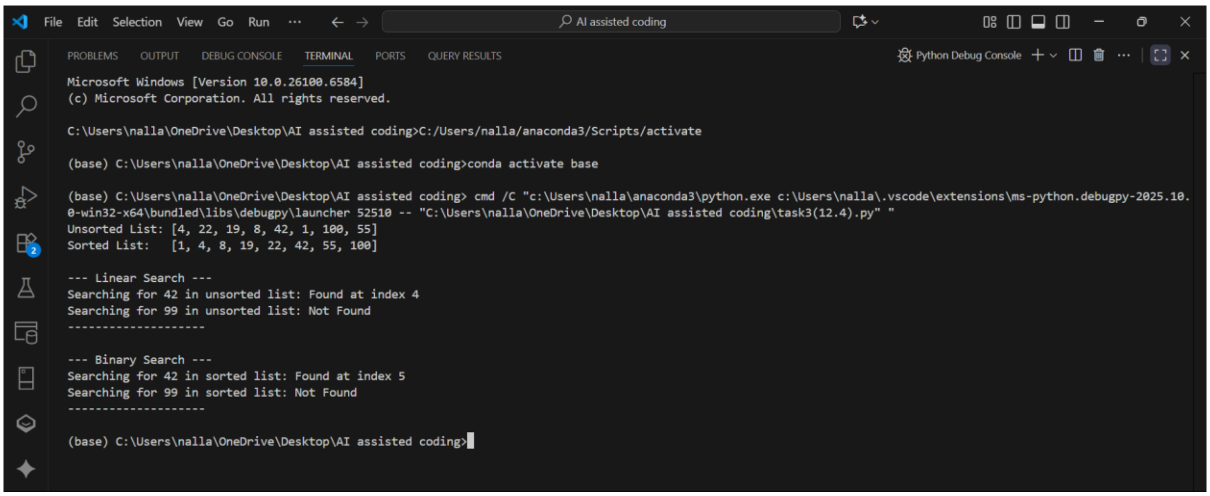
```python
from typing import List, Any

def linear_search(data: List[Any], target: Any) -> int:
    """
    Performs a linear search to find the index of a target element in a list.

    This function iterates through each element of the list sequentially until it
    finds a match for the target or reaches the end of the list.

    Args:
        data (List[Any]): The list of elements to search through. This list
                          does not need to be sorted.
        target (Any): The element to search for.

    Returns:
        int: The index of the first occurrence of the target element.
             Returns -1 if the target is not found in the list.

    Performance Notes:
        - Time Complexity: O(n)
          The time taken grows linearly with the number of elements (n) in the list.
          - Best Case: O(1) when the target is the first element.
          - Worst Case: O(n) when the target is the last element or not present.
        - Space Complexity: O(1)
          The algorithm uses a constant amount of extra space, regardless of the
          size of the input list.
    """
    for index, value in enumerate(data):
        if value == target:
            return index
    return -1
```

```python
def binary_search(sorted_data: List[Any], target: Any) -> int:
    """
    Performs a binary search to find the index of a target in a sorted list.

    This function repeatedly divides the search interval in half. If the value
    of the search key is less than the item in the middle of the interval, it
    narrows the interval to the lower half. Otherwise, it narrows it to the
    upper half.

    Args:
        sorted_data (List[Any]): The list of elements to search through.
                                 **This list must be sorted in ascending order.**
        target (Any): The element to search for.

    Returns:
        int: The index of the target element if found.
             Returns -1 if the target is not found in the list.

    Performance Notes:
        - Time Complexity: O(log n)
          With each comparison, the search area is halved, making it extremely
          efficient for large datasets.
        - Space Complexity: O(1)
          This iterative implementation uses a constant amount of extra space.
          (A recursive version would have a space complexity of O(log n)).
    """
    low = 0
    high = len(sorted_data) - 1

    while low <= high:
        # Calculate mid index to avoid potential overflow in other languages,
        # though it's not an issue with Python's arbitrary-precision integers.
        mid = low + (high - low) // 2

        mid_value = sorted_data[mid]

        if mid_value == target:
            return mid
        elif mid_value < target:
            # The target is in the right half
            low = mid + 1
        else:
            # The target is in the left half
            high = mid - 1

    return -1

# --- Example Usage ---
if __name__ == "__main__":
    unsorted_list = [4, 22, 19, 8, 42, 1, 100, 55]
    sorted_list = sorted(unsorted_list) # [1, 4, 8, 19, 22, 42, 55, 100]

    target_to_find = 42
    target_not_found = 99

    print(f"Unsorted List: {unsorted_list}")
    print(f"Sorted List:   {sorted_list}\n")

    # --- Linear Search Demonstration ---
    print("--- Linear Search ---")
    index1 = linear_search(unsorted_list, target_to_find)
    print(f"Searching for {target_to_find} in unsorted list: {'Found at index ' + str(index1) if index1 != -1 else 'Not Found'}")

    index2 = linear_search(unsorted_list, target_not_found)
```



```python
    index2 = linear_search(unsorted_list, target_not_found)
    print(f"Searching for {target_not_found} in unsorted list: {'Found at index ' + str(index2) if index2 != -1 else 'Not Found'}")
    print("-" * 20 + "\n")

    # --- Binary Search Demonstration ---
    print("--- Binary Search ---")
    # Note: Binary search MUST be used on a sorted list.
    index3 = binary_search(sorted_list, target_to_find)
    print(f"Searching for {target_to_find} in sorted list: {'Found at index ' + str(index3) if index3 != -1 else 'Not Found'}")

    index4 = binary_search(sorted_list, target_not_found)
    print(f"Searching for {target_not_found} in sorted list: {'Found at index ' + str(index4) if index4 != -1 else 'Not Found'}")
    print("-" * 20)
```

OP:



```
Microsoft Windows [Version 10.0.26100.6584]
(c) Microsoft Corporation. All rights reserved.

C:\Users\nalla\OneDrive\Desktop\AI assisted coding>C:/Users/nalla/anaconda3/Scripts/activate

(base) C:\Users\nalla\OneDrive\Desktop\AI assisted coding>conda activate base

(base) C:\Users\nalla\OneDrive\Desktop\AI assisted coding> cmd /C "c:\Users\nalla\anaconda3\python.exe c:\Users\nalla\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher 52510 -- "C:\Users\nalla\OneDrive\Desktop\AI assisted coding\task3(12.4).py" "
Unsorted List: [4, 22, 19, 8, 42, 1, 100, 55]
Sorted List:   [1, 4, 8, 19, 22, 42, 55, 100]

--- Linear Search ---
Searching for 42 in unsorted list: Found at index 4
Searching for 99 in unsorted list: Not Found
--------------------

--- Binary Search ---
Searching for 42 in sorted list: Found at index 5
Searching for 99 in sorted list: Not Found
--------------------

(base) C:\Users\nalla\OneDrive\Desktop\AI assisted coding>
```

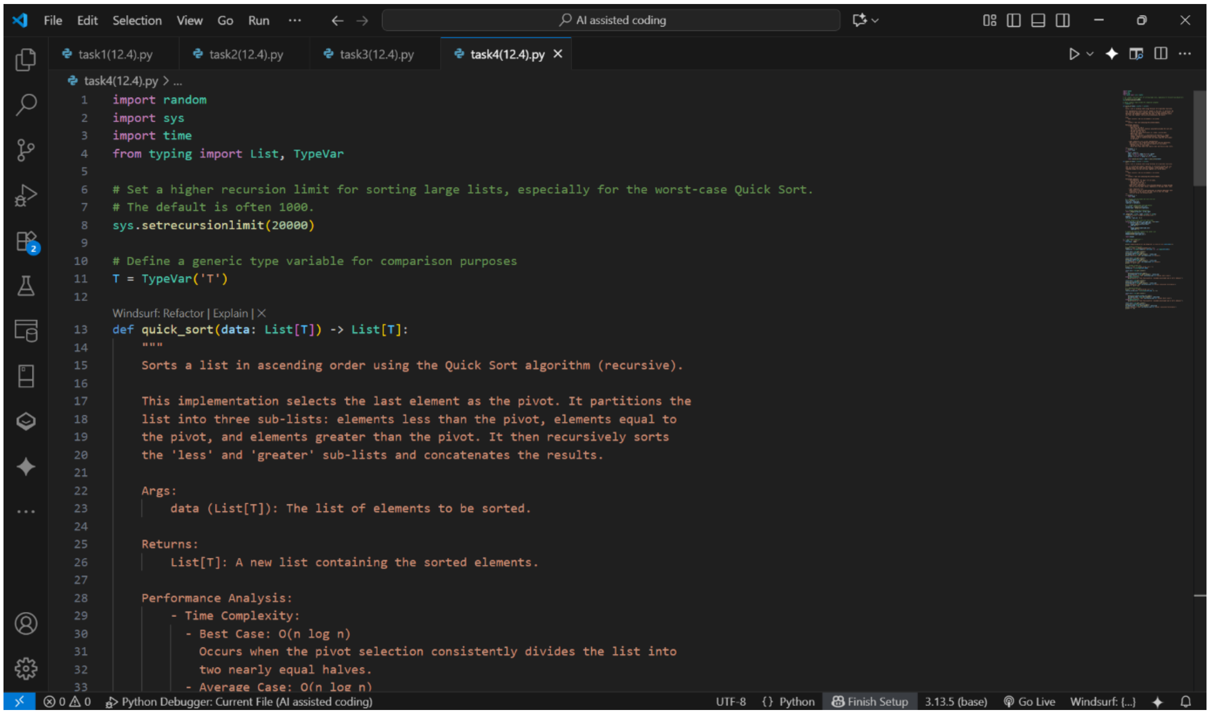
Observation:

In this program the comments are clear. We can understand the functions of each and everything. As comments are at the starting of the program, we can get a clarity on the code.

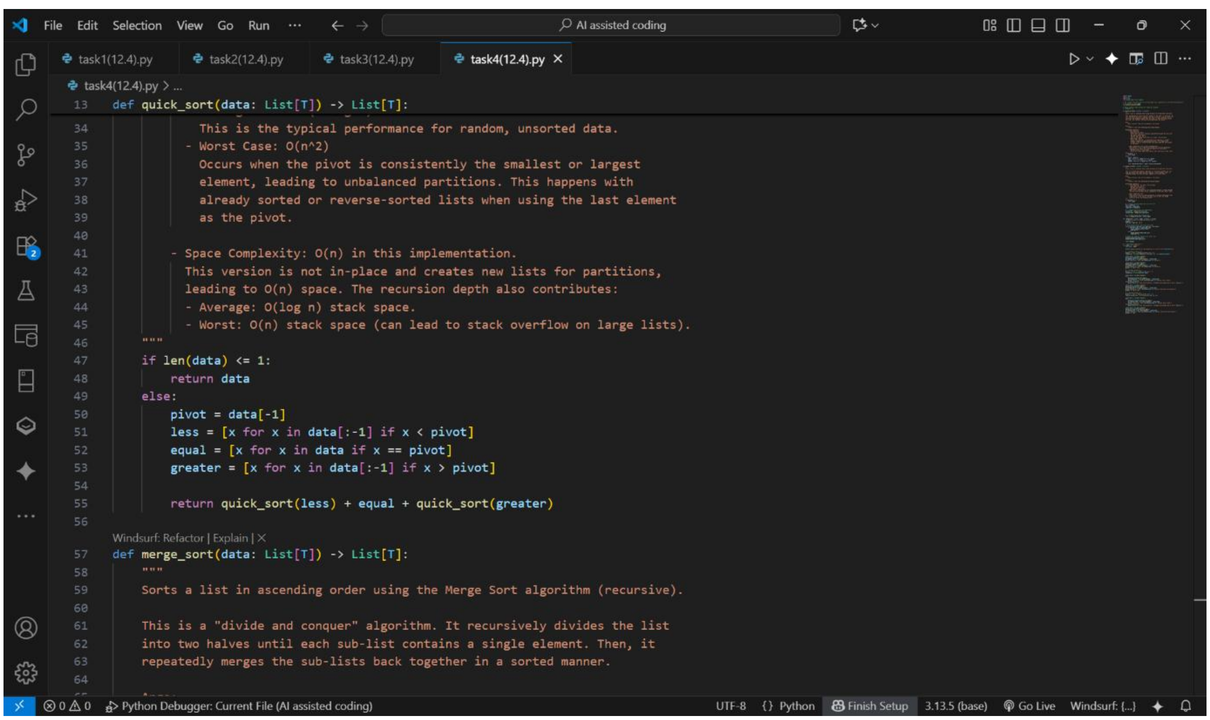
Task-4:

Prompt: Write a python code for implementation of Quick sort and Merge sort using recursion and add docstrings. Explain average, best, and worst-case complexities. Compare both on random, sorted, and reverse-sorted lists.

Code:



```python
import random
import sys
import time
from typing import List, TypeVar

# Set a higher recursion limit for sorting large lists, especially for the worst-case Quick Sort.
# The default is often 1000.
sys.setrecursionlimit(20000)

# Define a generic type variable for comparison purposes
T = TypeVar('T')

def quick_sort(data: List[T]) -> List[T]:
    """
    Sorts a list in ascending order using the Quick Sort algorithm (recursive).

    This implementation selects the last element as the pivot. It partitions the
    list into three sub-lists: elements less than the pivot, elements equal to
    the pivot, and elements greater than the pivot. It then recursively sorts
    the 'less' and 'greater' sub-lists and concatenates the results.

    Args:
        data (List[T]): The list of elements to be sorted.

    Returns:
        List[T]: A new list containing the sorted elements.

    Performance Analysis:
        - Time Complexity:
          - Best Case: O(n log n)
            Occurs when the pivot selection consistently divides the list into
            two nearly equal halves.
          - Average Case: O(n log n)
```

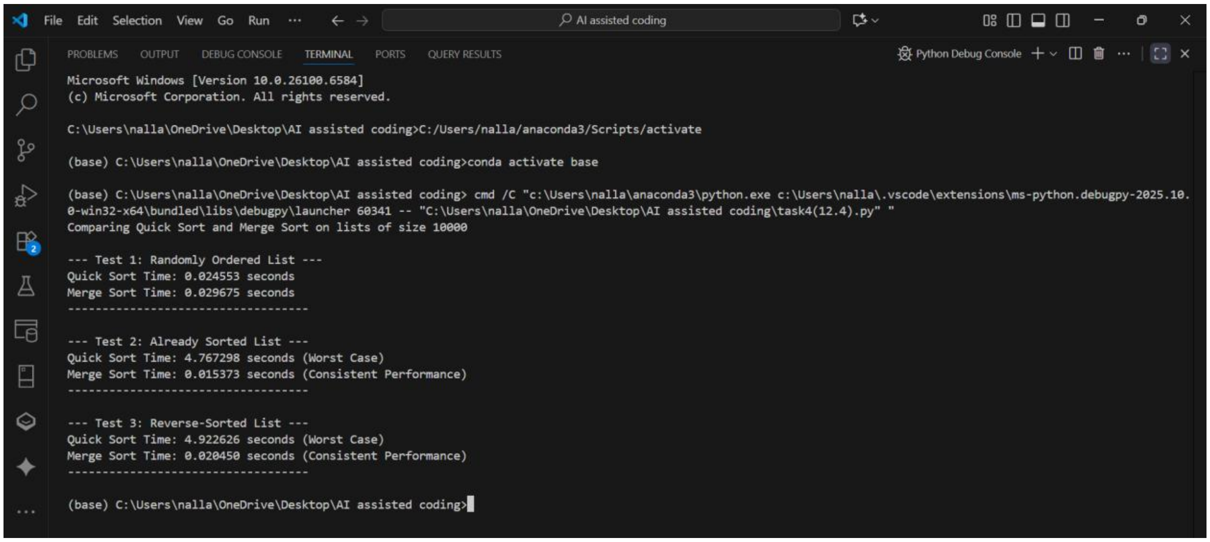


```python
def quick_sort(data: List[T]) -> List[T]:
            This is the typical performance for random, unsorted data.
          - Worst Case: O(n^2)
            Occurs when the pivot is consistently the smallest or largest
            element, leading to unbalanced partitions. This happens with
            already sorted or reverse-sorted lists when using the last element
            as the pivot.

        - Space Complexity: O(n) in this implementation.
          This version is not in-place and creates new lists for partitions,
          leading to O(n) space. The recursion depth also contributes:
          - Average: O(log n) stack space.
          - Worst: O(n) stack space (can lead to stack overflow on large lists).
    """
    if len(data) <= 1:
        return data
    else:
        pivot = data[-1]
        less = [x for x in data[:-1] if x < pivot]
        equal = [x for x in data if x == pivot]
        greater = [x for x in data[:-1] if x > pivot]

        return quick_sort(less) + equal + quick_sort(greater)

def merge_sort(data: List[T]) -> List[T]:
    """
    Sorts a list in ascending order using the Merge Sort algorithm (recursive).

    This is a "divide and conquer" algorithm. It recursively divides the list
    into two halves until each sub-list contains a single element. Then, it
    repeatedly merges the sub-lists back together in a sorted manner.
```

```python
def merge_sort(data: List[T]) -> List[T]:
    return _merge(sorted_left, sorted_right)

def _merge(left: List[T], right: List[T]) -> List[T]:
    """Helper function to merge two sorted lists."""
    merged = []
    left_idx, right_idx = 0, 0

    # Merge elements while both lists have items
    while left_idx < len(left) and right_idx < len(right):
        if left[left_idx] < right[right_idx]:
            merged.append(left[left_idx])
            left_idx += 1
        else:
            merged.append(right[right_idx])
            right_idx += 1

    # Append any remaining elements from either list
    merged.extend(left[left_idx:])
    merged.extend(right[right_idx:])

    return merged

# --- Performance Comparison ---
if __name__ == "__main__":
    LIST_SIZE = 10000

    print(f"Comparing Quick Sort and Merge Sort on lists of size {LIST_SIZE}\n")

    # 1. Randomly ordered list
    print("--- Test 1: Randomly Ordered List ---")
    random_list = [random.randint(0, LIST_SIZE) for _ in range(LIST_SIZE)]
```

```python
    start_time = time.perf_counter()
    try:
        quick_sort(sorted_list.copy())
        qs_sorted_time = time.perf_counter() - start_time
        print(f"Quick Sort Time: {qs_sorted_time:.6f} seconds (Worst Case)")
    except RecursionError:
        print("Quick Sort Time: RecursionError! (Exceeded stack depth due to O(n^2) behavior)")

    start_time = time.perf_counter()
    merge_sort(sorted_list.copy())
    ms_sorted_time = time.perf_counter() - start_time
    print(f"Merge Sort Time: {ms_sorted_time:.6f} seconds (Consistent Performance)")
    print("-" * 35 + "\n")

    # 3. Reverse-sorted list
    print("--- Test 3: Reverse-Sorted List ---")
    reverse_sorted_list = list(range(LIST_SIZE, 0, -1))

    start_time = time.perf_counter()
    try:
        quick_sort(reverse_sorted_list.copy())
        qs_reverse_time = time.perf_counter() - start_time
        print(f"Quick Sort Time: {qs_reverse_time:.6f} seconds (Worst Case)")
    except RecursionError:
        print("Quick Sort Time: RecursionError! (Exceeded stack depth due to O(n^2) behavior)")

    start_time = time.perf_counter()
    merge_sort(reverse_sorted_list.copy())
    ms_reverse_time = time.perf_counter() - start_time
    print(f"Merge Sort Time: {ms_reverse_time:.6f} seconds (Consistent Performance)")
    print("-" * 35)
```

OP:



Observation:

In this program the comments are clear. We can understand the functions of each and everything. As comments are at the starting of the program, we can get a clarity on the code.