

ASSIGNMENT – 12

NAME: G. Bala Varshitha

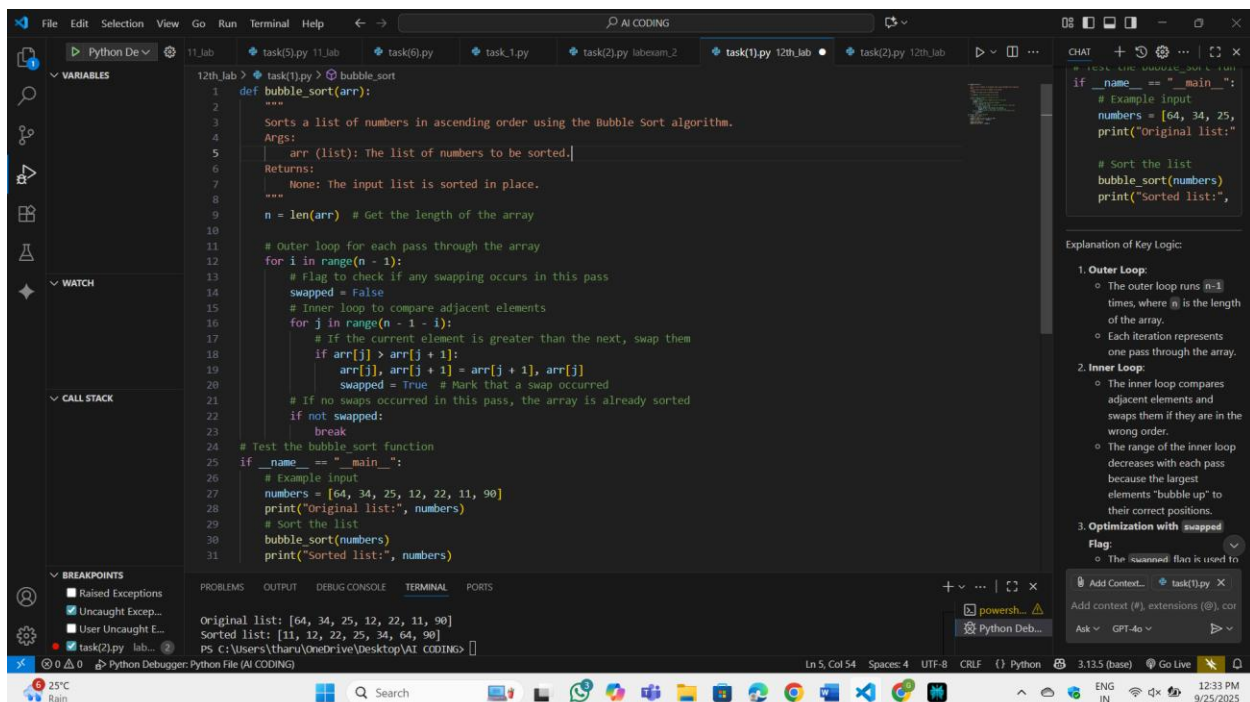
HT.NO: 2403A52050

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:

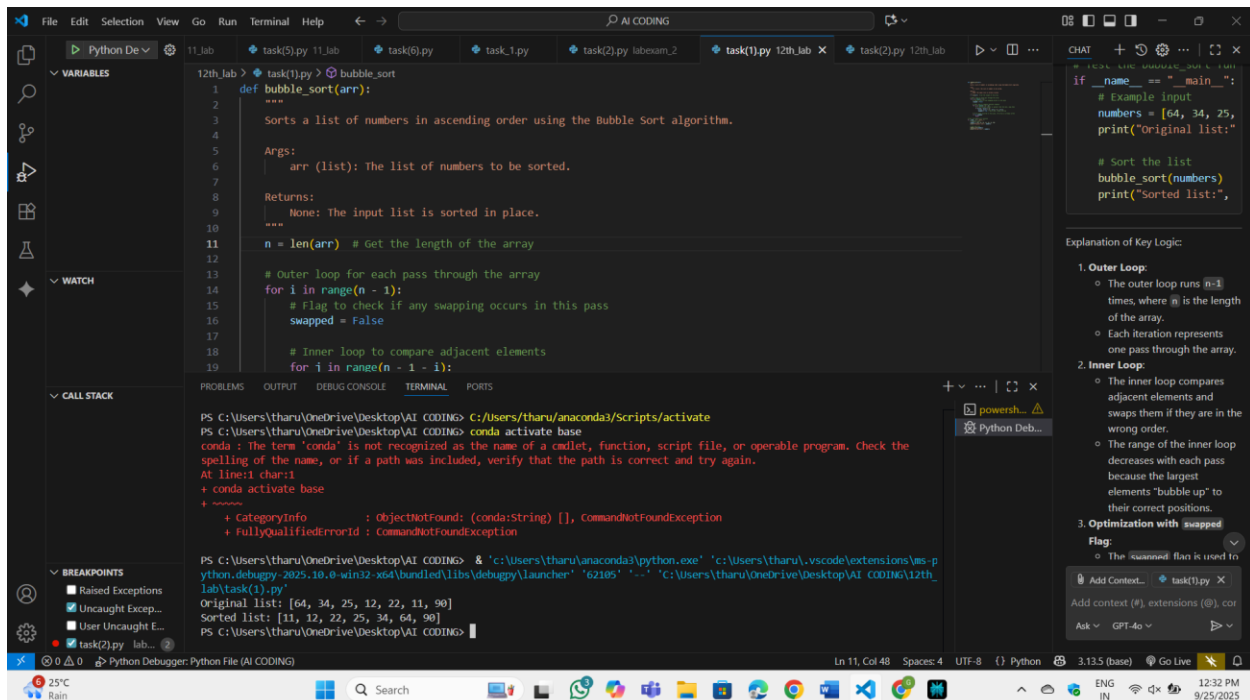


```
1  def bubble_sort(arr):
2      """
3      Sorts a list of numbers in ascending order using the Bubble Sort algorithm.
4      Args:
5      | arr (list): The list of numbers to be sorted.
6      Returns:
7      | None: The input list is sorted in place.
8      """
9      n = len(arr) # Get the length of the array
10
11     # Outer loop for each pass through the array
12     for i in range(n - 1):
13         # Flag to check if any swapping occurs in this pass
14         swapped = False
15         # Inner loop to compare adjacent elements
16         for j in range(n - 1 - i):
17             # If the current element is greater than the next, swap them
18             if arr[j] > arr[j + 1]:
19                 arr[j], arr[j + 1] = arr[j + 1], arr[j]
20                 swapped = True # Mark that a swap occurred
21             # If no swaps occurred in this pass, the array is already sorted
22             if not swapped:
23                 break
24     # Test the bubble_sort function
25     if __name__ == "__main__":
26         # Example input
27         numbers = [64, 34, 25, 12, 22, 11, 90]
28         print("Original list:", numbers)
29         # Sort the list
30         bubble_sort(numbers)
31         print("Sorted list:", numbers)
```

Explanation of Key Logic:

- Outer Loop:**
 - The outer loop runs $n-1$ times, where n is the length of the array.
 - Each iteration represents one pass through the array.
- Inner Loop:**
 - The inner loop compares adjacent elements and swaps them if they are in the wrong order.
 - The range of the inner loop decreases with each pass because the largest elements "bubble up" to their correct positions.
- Optimization with swapped**
 - Flag: The `swapped` flag is used to

OP:



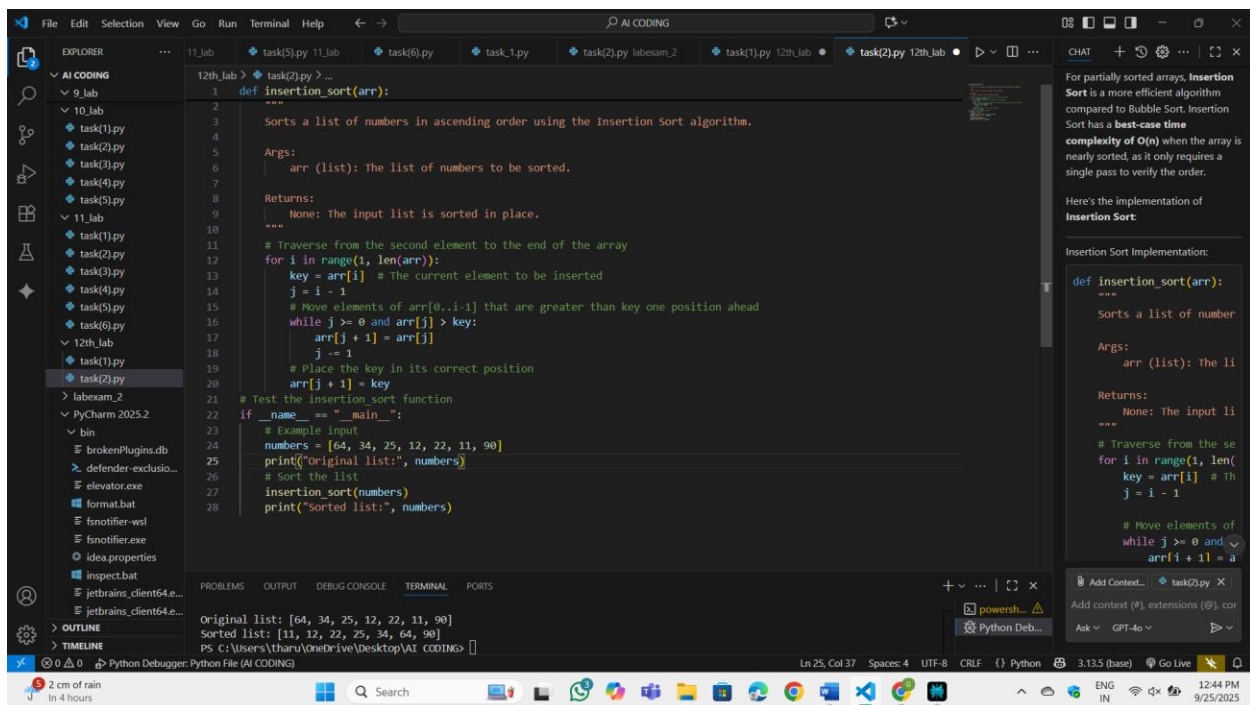
Observation:

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.

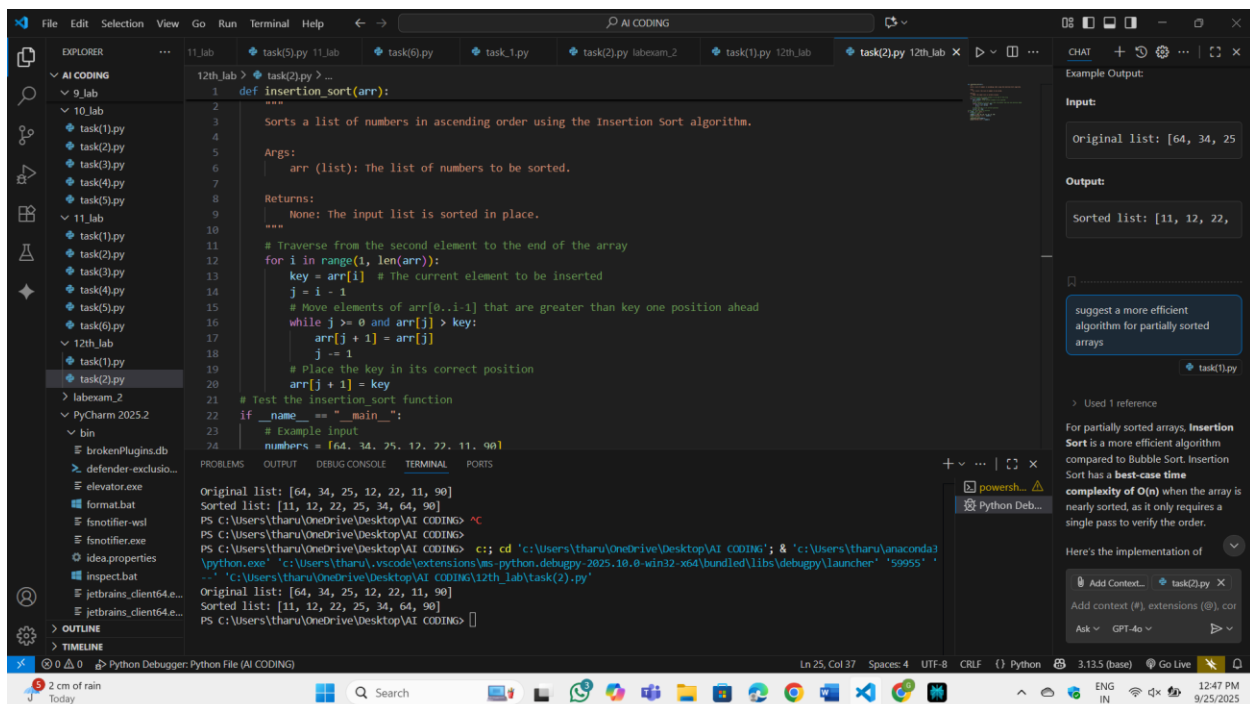
Task-2

Prompt: suggest a more efficient algorithm for partially sorted arrays

Code:



OP:



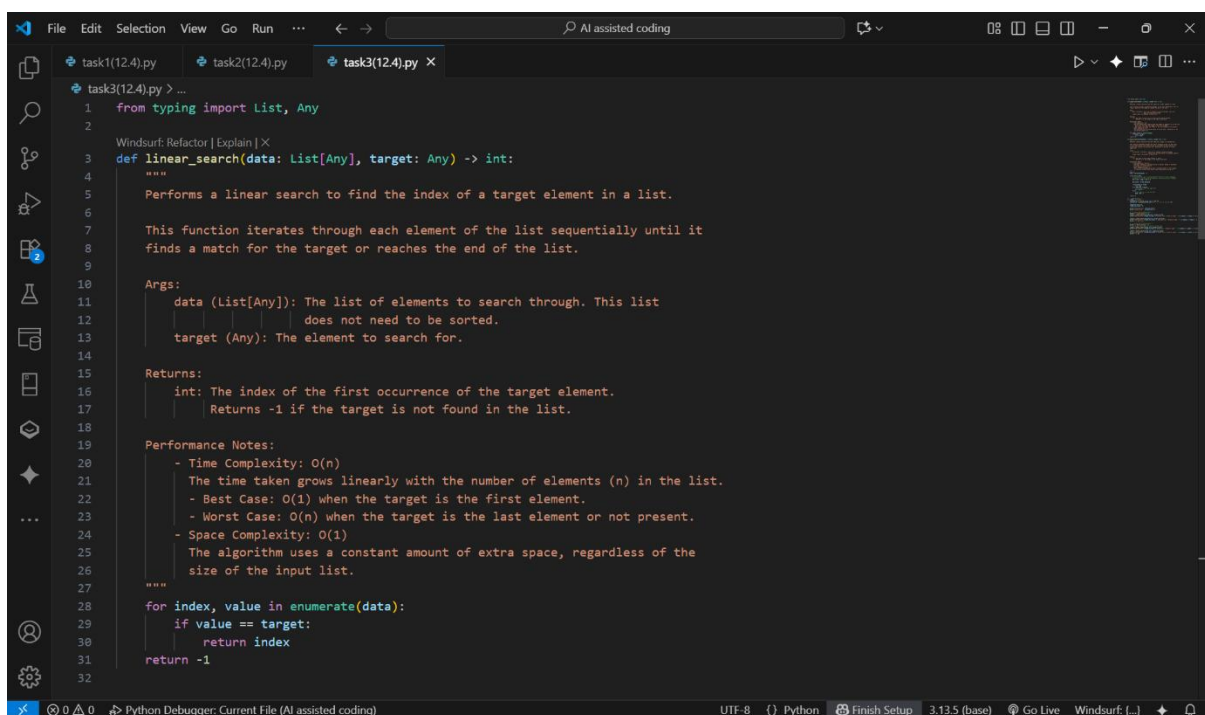
Observation:

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

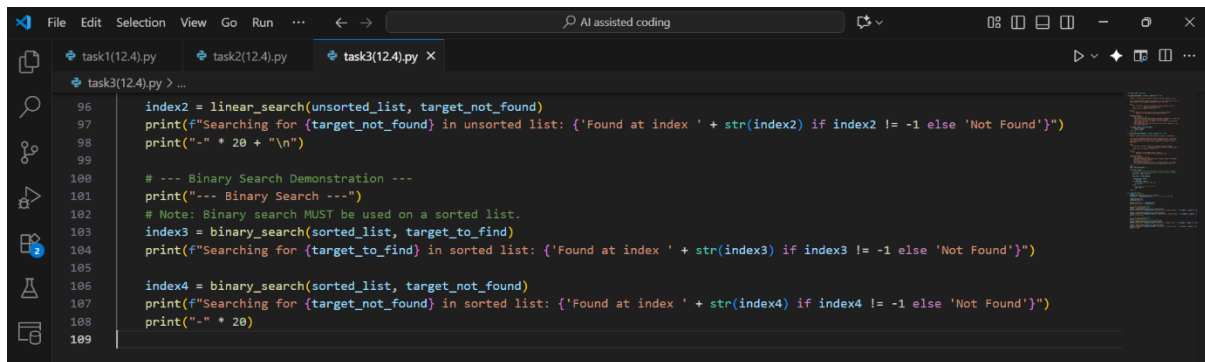
Code:



```
1 from typing import List, Any
2
3 def linear_search(data: List[Any], target: Any) -> int:
4     """
5     Performs a linear search to find the index of a target element in a list.
6
7     This function iterates through each element of the list sequentially until it
8     finds a match for the target or reaches the end of the list.
9
10    Args:
11        data (List[Any]): The list of elements to search through. This list
12                           does not need to be sorted.
13        target (Any): The element to search for.
14
15    Returns:
16        int: The index of the first occurrence of the target element.
17             Returns -1 if the target is not found in the list.
18
19    Performance Notes:
20        - Time Complexity: O(n)
21          The time taken grows linearly with the number of elements (n) in the list.
22        - Best Case: O(1) when the target is the first element.
23        - Worst Case: O(n) when the target is the last element or not present.
24        - Space Complexity: O(1)
25          The algorithm uses a constant amount of extra space, regardless of the
26          size of the input list.
27    """
28    for index, value in enumerate(data):
29        if value == target:
30            return index
31    return -1
32
```

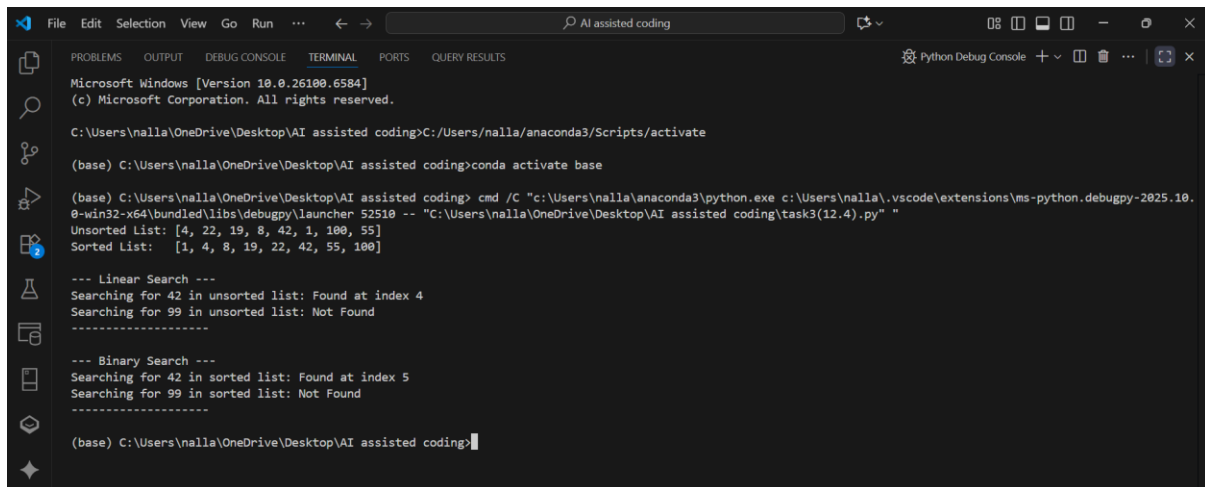
```
File Edit Selection View Go Run ... AI assisted coding
task1(12.4).py task2(12.4).py task3(12.4).py X
task3(12.4).py > ...
32 Windsurf: Refactor | Explain | X
33 def binary_search(sorted_data: List[Any], target: Any) -> int:
34     """
35     Performs a binary search to find the index of a target in a sorted list.
36
37     This function repeatedly divides the search interval in half. If the value
38     of the search key is less than the item in the middle of the interval, it
39     narrows the interval to the lower half. Otherwise, it narrows it to the
40     upper half.
41
42     Args:
43         sorted_data (List[Any]): The list of elements to search through.
44         target (Any): The element to search for.
45
46     Returns:
47         int: The index of the target element if found.
48         Returns -1 if the target is not found in the list.
49
50     Performance Notes:
51         - Time Complexity: O(log n)
52         - With each comparison, the search area is halved, making it extremely
53           efficient for large datasets.
54         - Space Complexity: O(1)
55         - This iterative implementation uses a constant amount of extra space.
56           (A recursive version would have a space complexity of O(log n)).
57     """
58
59     low = 0
60     high = len(sorted_data) - 1
61
62     while low <= high:
63         # Calculate mid index to avoid potential overflow in other languages,
64         # though it's not an issue with Python's arbitrary-precision integers.
```

```
File Edit Selection View Go Run ... AI assisted coding
task1(12.4).py task2(12.4).py task3(12.4).py X
task3(12.4).py > ...
33 def binary_search(sorted_data: List[Any], target: Any) -> int:
34     # though it's not an issue with Python's arbitrary-precision integers.
35     mid = low + (high - low) // 2
36
37     mid_value = sorted_data[mid]
38
39     if mid_value == target:
40         return mid
41     elif mid_value < target:
42         # The target is in the right half
43         low = mid + 1
44     else:
45         # The target is in the left half
46         high = mid - 1
47
48     return -1
49
50 # --- Example Usage ---
51 if __name__ == "__main__":
52     unsorted_list = [4, 22, 19, 8, 42, 1, 100, 55]
53     sorted_list = sorted(unsorted_list) # [1, 4, 8, 19, 22, 42, 55, 100]
54
55     target_to_find = 42
56     target_not_found = 99
57
58     print(f"Unsorted List: {unsorted_list}")
59     print(f"Sorted List: {sorted_list}\n")
60
61     # --- Linear Search Demonstration ---
62     print("--- Linear Search ---")
63     index1 = linear_search(unsorted_list, target_to_find)
64     print(f"Searching for {target_to_find} in unsorted list: {'Found at index ' + str(index1) if index1 != -1 else 'Not Found'}")
65
66     index2 = linear_search(sorted_list, target_not_found)
```



```
96 index2 = linear_search(unsorted_list, target_not_found)
97 print(f"Searching for {target_not_found} in unsorted list: {'Found at index ' + str(index2) if index2 != -1 else 'Not Found'}")
98 print("-" * 20 + "\n")
99
100 # --- Binary Search Demonstration ---
101 print("--- Binary Search ---")
102 # Note: Binary search MUST be used on a sorted list.
103 index3 = binary_search(sorted_list, target_to_find)
104 print(f"Searching for {target_to_find} in sorted list: {'Found at index ' + str(index3) if index3 != -1 else 'Not Found'}")
105
106 index4 = binary_search(sorted_list, target_not_found)
107 print(f"Searching for {target_not_found} in sorted list: {'Found at index ' + str(index4) if index4 != -1 else 'Not Found'}")
108 print("-" * 20)
109
```

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\bundle\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>
```

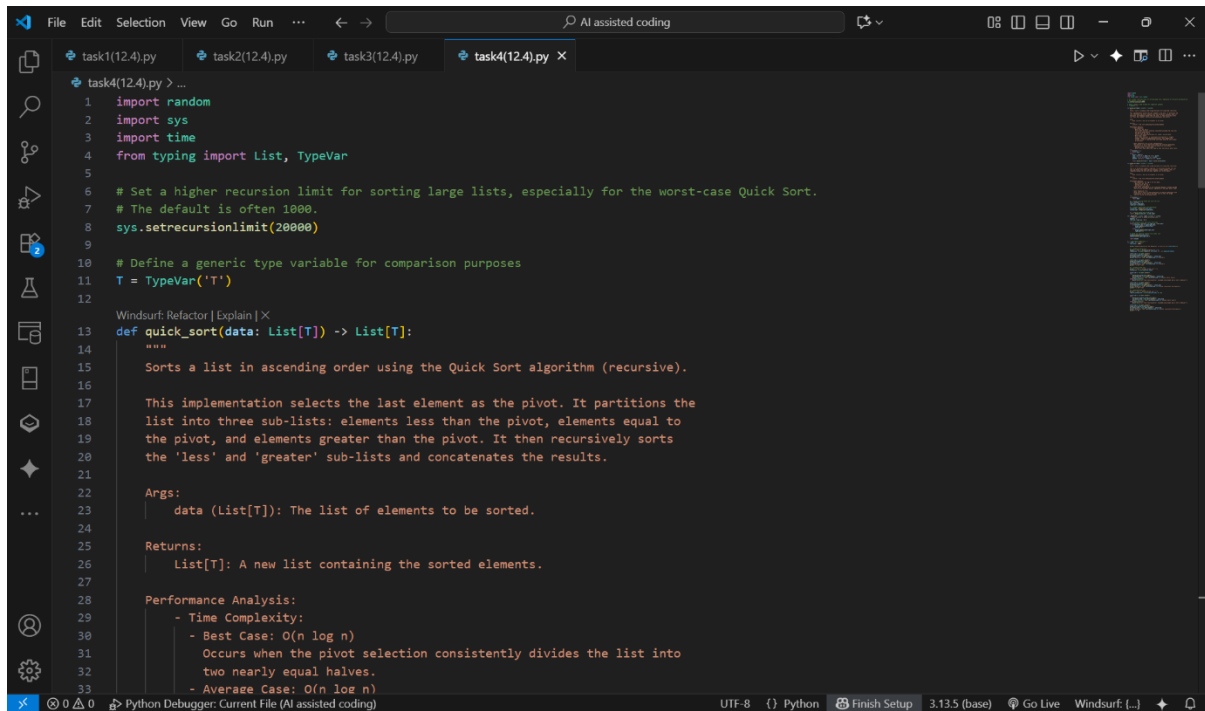
Observation:

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:

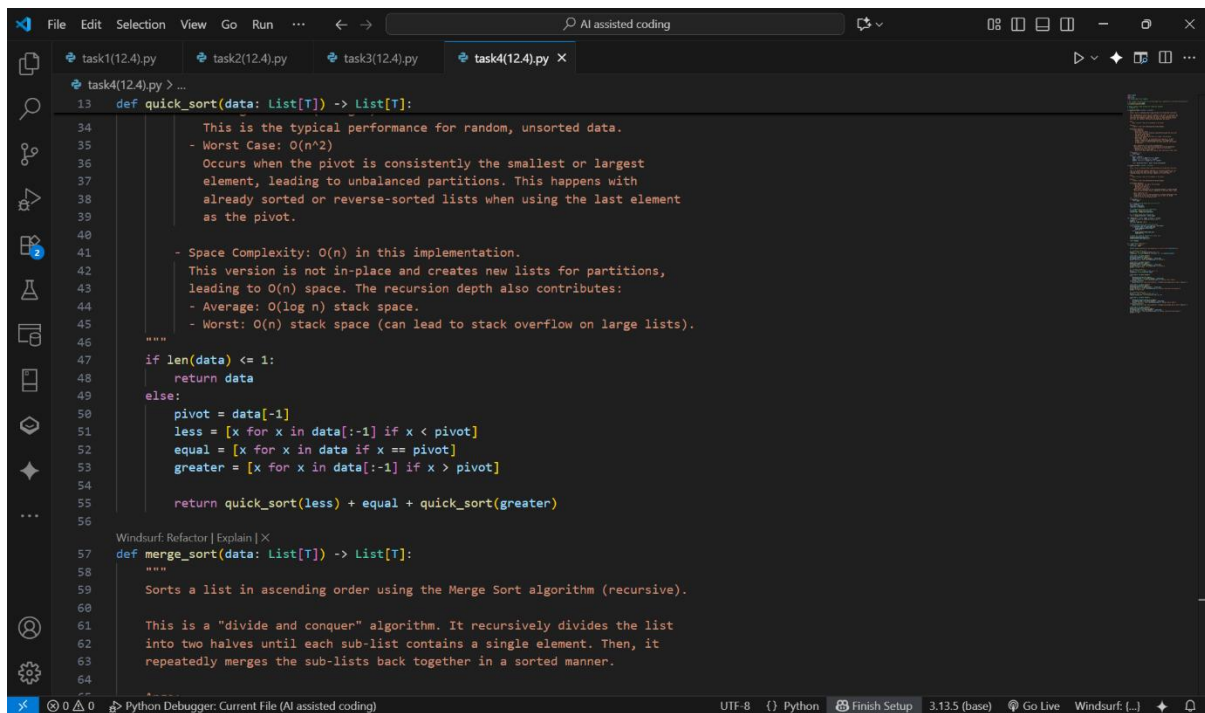
Prompt: Write a python code for implementation of Quick sort and Merge sort using recursion and add docstrings. Explain average, best, and worst-case complexities. Compare both on random, sorted, and reverse-sorted lists.

Code:



```
File Edit Selection View Go Run ... AI assisted coding task1(12.4).py task2(12.4).py task3(12.4).py task4(12.4).py X
```

```
task4(12.4).py > ...
1 import random
2 import sys
3 import time
4 from typing import List, TypeVar
5
6 # Set a higher recursion limit for sorting large lists, especially for the worst-case Quick Sort.
7 # The default is often 1000.
8 sys.setrecursionlimit(20000)
9
10 # Define a generic type variable for comparison purposes
11 T = TypeVar('T')
12
13 Windsurf: Refactor | Explain | X
14 def quick_sort(data: List[T]) -> List[T]:
15     """
16     Sorts a list in ascending order using the Quick Sort algorithm (recursive).
17
18     This implementation selects the last element as the pivot. It partitions the
19     list into three sub-lists: elements less than the pivot, elements equal to
20     the pivot, and elements greater than the pivot. It then recursively sorts
21     the 'less' and 'greater' sub-lists and concatenates the results.
22
23     Args:
24         data (List[T]): The list of elements to be sorted.
25
26     Returns:
27         List[T]: A new list containing the sorted elements.
28
29     Performance Analysis:
30         - Time Complexity:
31             - Best Case:  $O(n \log n)$ 
32               Occurs when the pivot selection consistently divides the list into
33               two nearly equal halves.
34             - Average Case:  $O(n \log n)$ 
```



```
File Edit Selection View Go Run ... AI assisted coding task1(12.4).py task2(12.4).py task3(12.4).py task4(12.4).py X
```

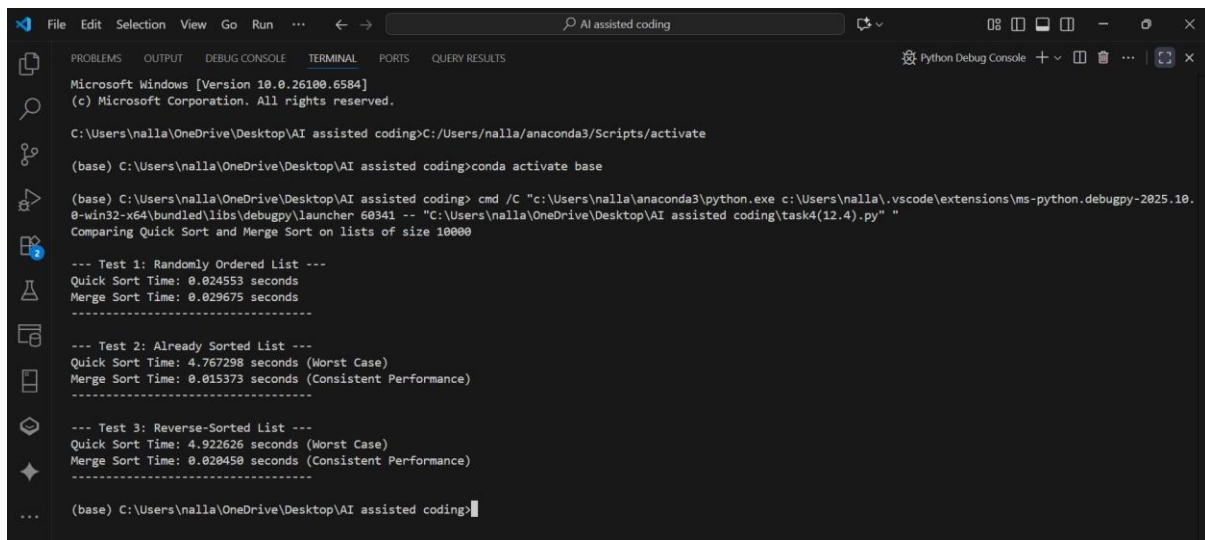
```
task4(12.4).py > ...
13 def quick_sort(data: List[T]) -> List[T]:
14     """
15     This is the typical performance for random, unsorted data.
16     - Worst Case:  $O(n^2)$ 
17       Occurs when the pivot is consistently the smallest or largest
18       element, leading to unbalanced partitions. This happens with
19       already sorted or reverse-sorted lists when using the last element
20       as the pivot.
21
22     - Space Complexity:  $O(n)$  in this implementation.
23       This version is not in-place and creates new lists for partitions,
24       leading to  $O(n)$  space. The recursion depth also contributes:
25       - Average:  $O(\log n)$  stack space.
26       - Worst:  $O(n)$  stack space (can lead to stack overflow on large lists).
27
28     """
29     if len(data) <= 1:
30         return data
31     else:
32         pivot = data[-1]
33         less = [x for x in data[:-1] if x < pivot]
34         equal = [x for x in data if x == pivot]
35         greater = [x for x in data[:-1] if x > pivot]
36
37         return quick_sort(less) + equal + quick_sort(greater)
38
39 Windsurf: Refactor | Explain | X
40 def merge_sort(data: List[T]) -> List[T]:
41     """
42     Sorts a list in ascending order using the Merge Sort algorithm (recursive).
43
44     This is a "divide and conquer" algorithm. It recursively divides the list
45     into two halves until each sub-list contains a single element. Then, it
46     repeatedly merges the sub-lists back together in a sorted manner.
47
48     """
```



```
File Edit Selection View Go Run ... AI assisted coding
task1(12.4).py task2(12.4).py task3(12.4).py task4(12.4).py X
task4(12.4).py > ...
57 def merge_sort(data: List[T]) -> List[T]:
97     return _merge(sorted_left, sorted_right)
98
Windsurf: Refactor | Explain | X
99 def _merge(left: List[T], right: List[T]) -> List[T]:
100     """Helper function to merge two sorted lists."""
101     merged = []
102     left_idx, right_idx = 0, 0
103
104     # Merge elements while both lists have items
105     while left_idx < len(left) and right_idx < len(right):
106         if left[left_idx] < right[right_idx]:
107             merged.append(left[left_idx])
108             left_idx += 1
109         else:
110             merged.append(right[right_idx])
111             right_idx += 1
112
113     # Append any remaining elements from either list
114     merged.extend(left[left_idx:])
115     merged.extend(right[right_idx:])
116
117     return merged
118
119 # --- Performance Comparison ---
120 if __name__ == "__main__":
121     LIST_SIZE = 10000
122
123     print(f"Comparing Quick Sort and Merge Sort on lists of size {LIST_SIZE}\n")
124
125     # 1. Randomly ordered list
126     print("--- Test 1: Randomly Ordered List ---")
127     random_list = [random.randint(0, LIST_SIZE) for _ in range(LIST_SIZE)]
128
Python Debugger: Current File (AI assisted coding) UTF-8 Python Finish Setup 3.13.5 (base) Go Live Windsurf: [..]
```

```
File Edit Selection View Go Run ... AI assisted coding
task1(12.4).py task2(12.4).py task3(12.4).py task4(12.4).py X
task4(12.4).py > ...
144 start_time = time.perf_counter()
145 try:
146     quick_sort(sorted_list.copy())
147     qs_sorted_time = time.perf_counter() - start_time
148     print(f"Quick Sort Time: {qs_sorted_time:.6f} seconds (Worst Case)")
149 except RecursionError:
150     print("Quick Sort Time: RecursionError! (Exceeded stack depth due to O(n^2) behavior)")
151
152 start_time = time.perf_counter()
153 merge_sort(sorted_list.copy())
154 ms_sorted_time = time.perf_counter() - start_time
155 print(f"Merge Sort Time: {ms_sorted_time:.6f} seconds (Consistent Performance)")
156 print("-" * 35 + "\n")
157
158 # 3. Reverse-sorted list
159 print("--- Test 3: Reverse-Sorted List ---")
160 reverse_sorted_list = list(range(LIST_SIZE, 0, -1))
161
162 start_time = time.perf_counter()
163 try:
164     quick_sort(reverse_sorted_list.copy())
165     qs_reverse_time = time.perf_counter() - start_time
166     print(f"Quick Sort Time: {qs_reverse_time:.6f} seconds (Worst Case)")
167 except RecursionError:
168     print("Quick Sort Time: RecursionError! (Exceeded stack depth due to O(n^2) behavior)")
169
170 start_time = time.perf_counter()
171 merge_sort(reverse_sorted_list.copy())
172 ms_reverse_time = time.perf_counter() - start_time
173 print(f"Merge Sort Time: {ms_reverse_time:.6f} seconds (Consistent Performance)")
174 print("-" * 35)
175
```

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\bundle\libs\debugpy\launcher 60341 -- "C:\Users\nalla\OneDrive\Desktop\AI assisted coding\task4(12.4).py" "
Comparing Quick Sort and Merge Sort on lists of size 10000

--- Test 1: Randomly Ordered List ---
Quick Sort Time: 0.024553 seconds
Merge Sort Time: 0.029675 seconds
-----

--- Test 2: Already Sorted List ---
Quick Sort Time: 4.767298 seconds (Worst Case)
Merge Sort Time: 0.015373 seconds (Consistent Performance)
-----

--- Test 3: Reverse-Sorted List ---
Quick Sort Time: 4.922626 seconds (Worst Case)
Merge Sort Time: 0.020450 seconds (Consistent Performance)
-----

... (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.