

ASSIGNMENT-12.4

2303A51893

Batch : 25

Task 1: Bubble Sort for Ranking Exam

Scores

Scenario

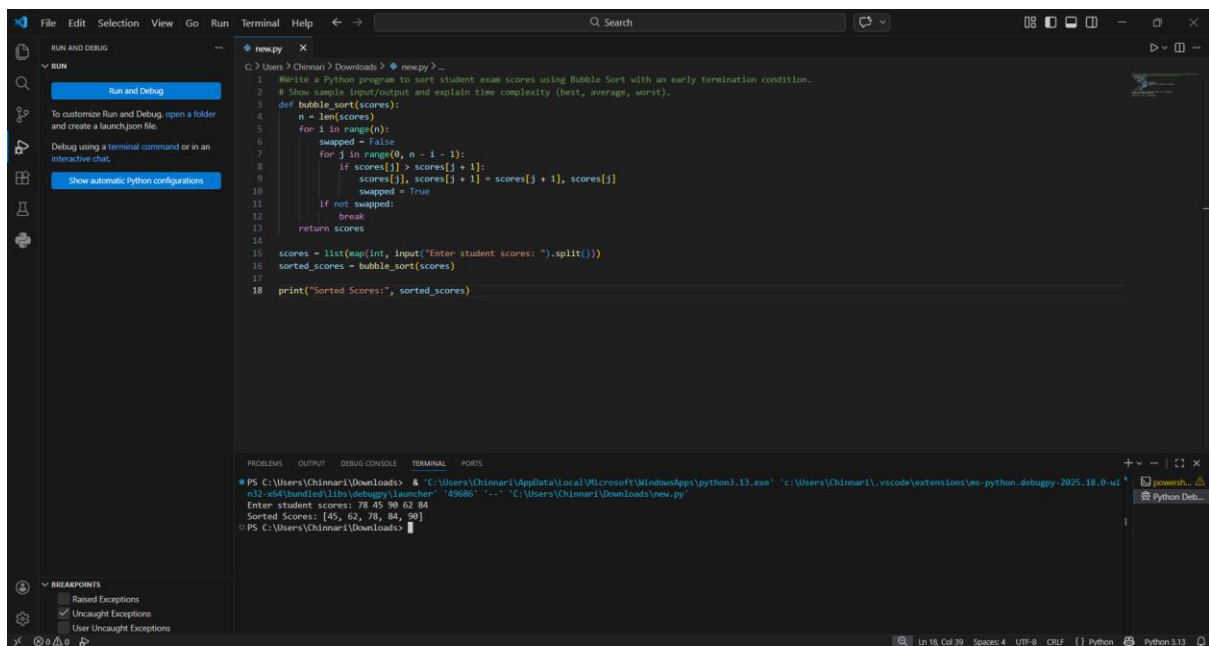
You are working on a college result processing system where a small list of student scores needs to be sorted after every internal assessment.

Task Description

- Implement Bubble Sort in Python to sort a list of student scores.
- Use an AI tool to:
 - o Insert inline comments explaining key operations such as comparisons, swaps, and iteration passes
 - o Identify early-termination conditions when the list becomes sorted
 - o Provide a brief time complexity analysis

Expected Outcome

- A Bubble Sort implementation with:
 - o AI-generated comments explaining the logic
 - o Clear explanation of best, average, and worst-case complexity
 - o Sample input/output showing sorted scores



Task 2: Improving Sorting for Nearly Sorted

Attendance Records

Scenario

You are maintaining an attendance system where student roll numbers are already almost sorted, with only a few late updates.

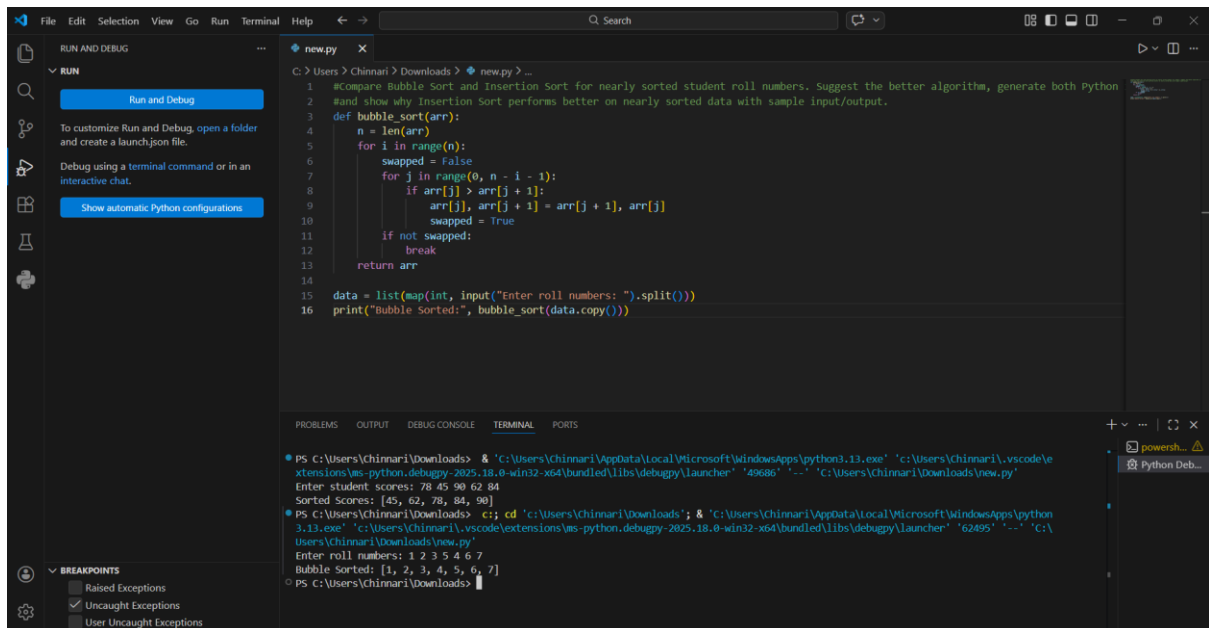
Task Description

- Start with a Bubble Sort implementation.
- Ask AI to:
 - o Review the problem and suggest a more suitable sorting algorithm
 - o Generate an Insertion Sort implementation
 - o Explain why Insertion Sort performs better on nearly sorted data

- Compare execution behavior on nearly sorted input

Expected Outcome

- Two sorting implementations:
 - o Bubble Sort
 - o Insertion Sort
- AI-assisted explanation highlighting efficiency differences for partially sorted datasets give code and prompt for this remove comments



Task 3: Searching Student Records in a Database

Scenario

You are developing a student information portal where users search for student records by roll number.

Task Description

- Implement:
 - o Linear Search for unsorted student data
 - o Binary Search for sorted student data
- Use AI to:
 - o Add docstrings explaining parameters and return values
 - o Explain when Binary Search is applicable
 - o Highlight performance differences between the two searches

Expected Outcome

- Two working search implementations with docstrings
- AI-generated explanation of:

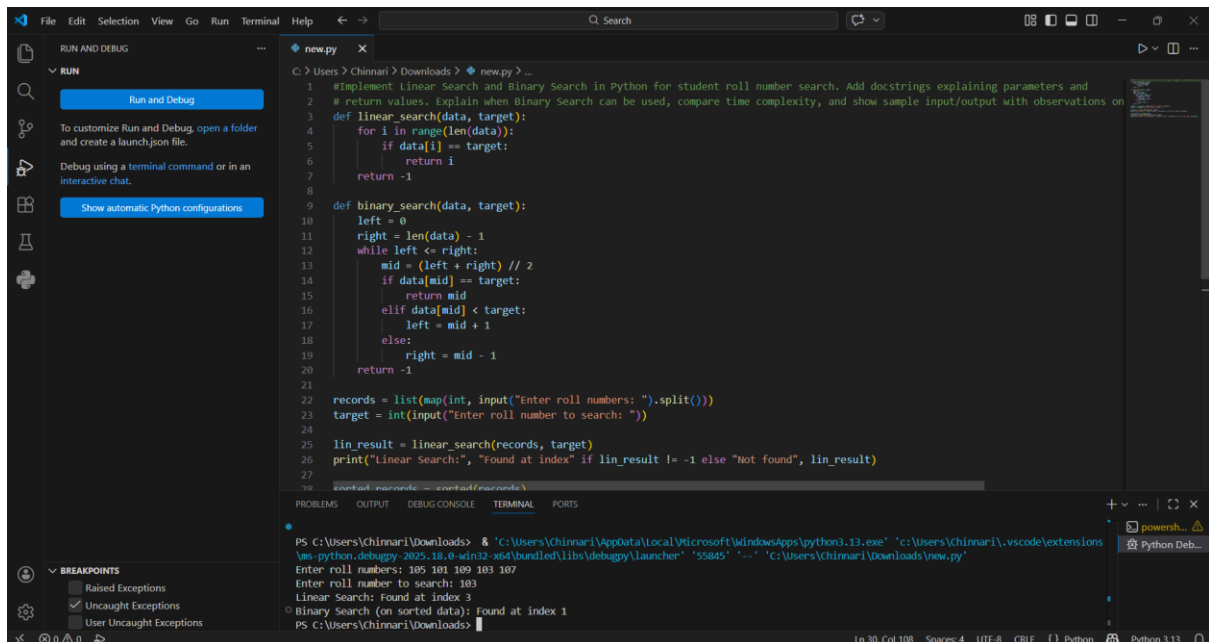
o Time complexity

o Use cases for Linear vs Binary Search

- A short student observation comparing results on sorted vs

unsorted lists

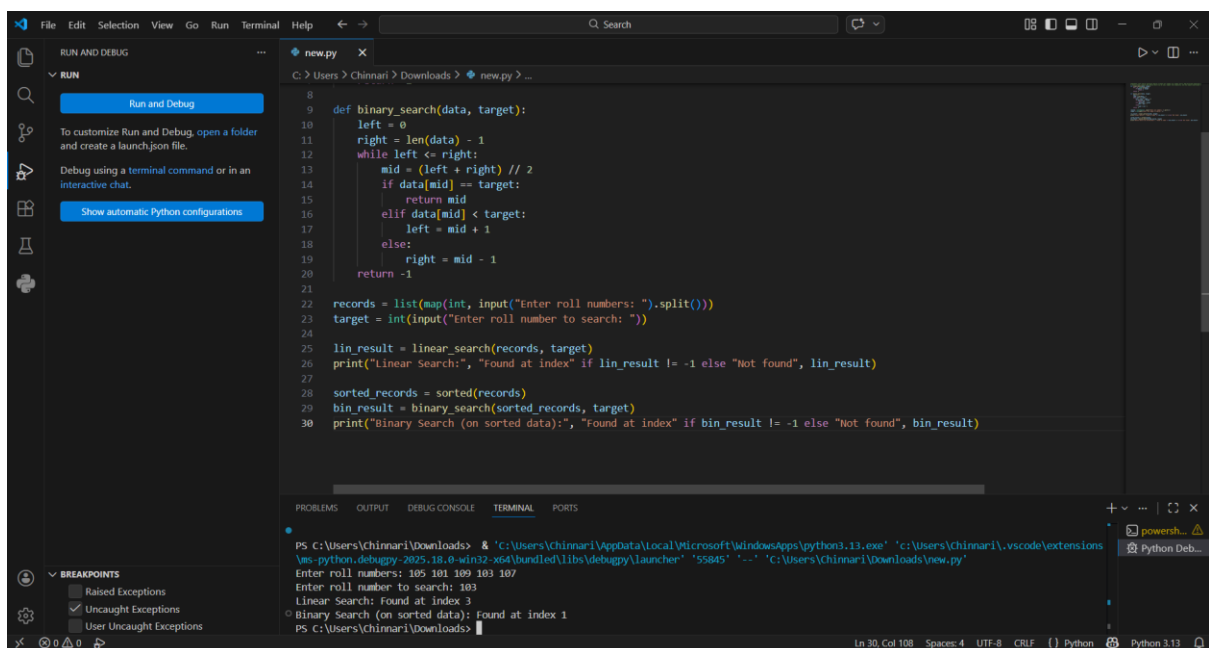
give code and prompt for this and remove comments



```
1 #Implement Linear Search and Binary Search in Python for student roll number search. Add docstrings explaining parameters and
2 # return values. Explain when Binary Search can be used, compare time complexity, and show sample input/output with observations on
3 def linear_search(data, target):
4     for i in range(len(data)):
5         if data[i] == target:
6             return i
7     return -1
8
9 def binary_search(data, target):
10     left = 0
11     right = len(data) - 1
12     while left <= right:
13         mid = (left + right) // 2
14         if data[mid] == target:
15             return mid
16         elif data[mid] < target:
17             left = mid + 1
18         else:
19             right = mid - 1
20     return -1
21
22 records = list(map(int, input("Enter roll numbers: ").split()))
23 target = int(input("Enter roll number to search: "))
24
25 lin_result = linear_search(records, target)
26 print("Linear Search:", "Found at index" if lin_result != -1 else "Not found", lin_result)
27
28 sorted_records = sorted(records)
29 bin_result = binary_search(sorted_records, target)
30 print("Binary Search (on sorted data):", "Found at index" if bin_result != -1 else "Not found", bin_result)
```

PS C:\Users\Chinnari\Downloads> & "C:\Users\Chinnari\AppData\Local\Microsoft\WindowsApps\python3.13.exe" "c:\Users\Chinnari\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundle\libs\debugpy\launcher" "55845" "-." "C:\Users\Chinnari\Downloads\new.py"

Enter roll numbers: 105 101 109 103 107
Enter roll number to search: 103
Linear Search: Found at index 3
Binary Search (on sorted data): Found at index 1
PS C:\Users\Chinnari\Downloads>



```
8
9 def binary_search(data, target):
10     left = 0
11     right = len(data) - 1
12     while left <= right:
13         mid = (left + right) // 2
14         if data[mid] == target:
15             return mid
16         elif data[mid] < target:
17             left = mid + 1
18         else:
19             right = mid - 1
20     return -1
21
22 records = list(map(int, input("Enter roll numbers: ").split()))
23 target = int(input("Enter roll number to search: "))
24
25 lin_result = linear_search(records, target)
26 print("Linear Search:", "Found at index" if lin_result != -1 else "Not found", lin_result)
27
28 sorted_records = sorted(records)
29 bin_result = binary_search(sorted_records, target)
30 print("Binary Search (on sorted data):", "Found at index" if bin_result != -1 else "Not found", bin_result)
```

PS C:\Users\Chinnari\Downloads> & "C:\Users\Chinnari\AppData\Local\Microsoft\WindowsApps\python3.13.exe" "c:\Users\Chinnari\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundle\libs\debugpy\launcher" "55845" "-." "C:\Users\Chinnari\Downloads\new.py"

Enter roll numbers: 105 101 109 103 107
Enter roll number to search: 103
Linear Search: Found at index 3
Binary Search (on sorted data): Found at index 1
PS C:\Users\Chinnari\Downloads>

Task 4: Choosing Between Quick Sort and

Merge Sort for Data Processing

Scenario

You are part of a data analytics team that needs to sort large datasets received from different sources (random order, already sorted, and reverse sorted).

Task Description

- Provide AI with partially written recursive functions for:

- o Quick Sort

- o Merge Sort

- Ask AI to:

- o Complete the recursive logic

- o Add meaningful docstrings

- o Explain how recursion works in each algorithm

- Test both algorithms on:

- o Random data

- o Sorted data

- o Reverse-sorted data

Expected Outcome

- Fully functional Quick Sort and Merge Sort implementations

- AI-generated comparison covering:

- o Best, average, and worst-case complexities

- o Practical scenarios where one algorithm is preferred over

the other give prompt and code for this and remove comments

```

1 #Complete recursive Quick Sort and Merge Sort functions in Python with docstrings. Explain recursion in both algorithms,
2 #test them on random, sorted, and reverse-sorted data, and compare time complexities and practical use cases.
3
4 def quick_sort(arr):
5     if len(arr) <= 1:
6         return arr
7     pivot = arr[len(arr) // 2]
8     left = [x for x in arr if x < pivot]
9     middle = [x for x in arr if x == pivot]
10    right = [x for x in arr if x > pivot]
11    return quick_sort(left) + middle + quick_sort(right)
12
13 def merge_sort(arr):
14     if len(arr) <= 1:
15         return arr
16     mid = len(arr) // 2
17     left = merge_sort(arr[:mid])
18     right = merge_sort(arr[mid:])
19     return merge(left, right)
20
21 def merge(left, right):
22     result = []
23     i = j = 0
24     while i < len(left) and j < len(right):
25         if left[i] < right[j]:
26             result.append(left[i])
27             i += 1
28         else:
29             result.append(right[j])
30             j += 1
31     result.extend(left[i:])
32     result.extend(right[j:])
33     return result
34
35 data = list(map(int, input("Enter numbers: ").split()))
36
37 print("Quick Sort on Random Data:", quick_sort(data))
38 print("Merge Sort on Random Data:", merge_sort(data))
39
40 sorted_data = sorted(data)
41 reverse_data = sorted(data, reverse=True)
42
43 print("Quick Sort on Sorted Data:", quick_sort(sorted_data))
44 print("Merge Sort on Sorted Data:", merge_sort(sorted_data))
45
46 print("Quick Sort on Reverse Data:", quick_sort(reverse_data))
47 print("Merge Sort on Reverse Data:", merge_sort(reverse_data))

```

Enter numbers: 8 3 1 7 0 10 2

Quick Sort on Random Data: [0, 1, 2, 3, 7, 8, 10]

Merge Sort on Random Data: [0, 1, 2, 3, 7, 8, 10]

Quick Sort on Sorted Data: [0, 1, 2, 3, 7, 8, 10]

Merge Sort on Sorted Data: [0, 1, 2, 3, 7, 8, 10]

Quick Sort on Reverse Data: [0, 1, 2, 3, 7, 8, 10]

Merge Sort on Reverse Data: [0, 1, 2, 3, 7, 8, 10]

```

19 def merge(left, right):
20     while i < len(left) and j < len(right):
21         if left[i] < right[j]:
22             result.append(left[i])
23             i += 1
24         else:
25             result.append(right[j])
26             j += 1
27     result.extend(left[i:])
28     result.extend(right[j:])
29     return result
30
31 data = list(map(int, input("Enter numbers: ").split()))
32
33 print("Quick Sort on Random Data:", quick_sort(data))
34 print("Merge Sort on Random Data:", merge_sort(data))
35
36 sorted_data = sorted(data)
37 reverse_data = sorted(data, reverse=True)
38
39 print("Quick Sort on Sorted Data:", quick_sort(sorted_data))
40 print("Merge Sort on Sorted Data:", merge_sort(sorted_data))
41
42 print("Quick Sort on Reverse Data:", quick_sort(reverse_data))
43 print("Merge Sort on Reverse Data:", merge_sort(reverse_data))

```

Enter numbers: 8 3 1 7 0 10 2

Quick Sort on Random Data: [0, 1, 2, 3, 7, 8, 10]

Merge Sort on Random Data: [0, 1, 2, 3, 7, 8, 10]

Quick Sort on Sorted Data: [0, 1, 2, 3, 7, 8, 10]

Merge Sort on Sorted Data: [0, 1, 2, 3, 7, 8, 10]

Quick Sort on Reverse Data: [0, 1, 2, 3, 7, 8, 10]

Merge Sort on Reverse Data: [0, 1, 2, 3, 7, 8, 10]

Task 5: Optimizing a Duplicate Detection

Algorithm

Scenario

You are building a data validation module that must detect duplicate user IDs in a large dataset before importing it into a system.

Task Description

- Write a naive duplicate detection algorithm using nested loops.
- Use AI to:
 - o Analyze the time complexity

- o Suggest an optimized approach using sets or dictionaries
- o Rewrite the algorithm with improved efficiency
- Compare execution behavior conceptually for large input sizes

Expected Outcome

- Two versions of the algorithm:
 - o Brute-force ($O(n^2)$)
 - o Optimized ($O(n)$)
- AI-assisted explanation showing how and why performance improved
- give prompt and code for this and remove comments

```

1 #Implement duplicate detection in Python using a brute-force nested loop and an optimized approach with a set.
2 #Compare time complexity and explain why the optimized version performs better for large datasets with sample input/output.
3 def brute_force_duplicates(data):
4     duplicates = []
5     n = len(data)
6     for i in range(n):
7         for j in range(i + 1, n):
8             if data[i] == data[j] and data[i] not in duplicates:
9                 duplicates.append(data[i])
10    return duplicates
11
12
13 def optimized_duplicates(data):
14     seen = set()
15     duplicates = set()
16     for item in data:
17         if item in seen:
18             duplicates.add(item)
19         else:
20             seen.add(item)
21    return list(duplicates)
22
23
24 data = list(map(int, input("Enter user IDs: ").split()))
25
26 print("Brute Force Duplicates:", brute_force_duplicates(data))
27 print("Optimized Duplicates:", optimized_duplicates(data))
  
```

Terminal Output:

```

Merge Sort on Reverse Data: [9, 1, 2, 3, 7, 8, 10]
PS C:\Users\Chinnari\Downloads> c: cd 'c:\Users\Chinnari\Downloads'; & 'C:\Users\Chinnari\AppData\Local\Microsoft\WindowsApps\python3.13.exe'
'c:\Users\Chinnari\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundle\libs\debugpy\launcher' '60117' '-...' 'C:\Users\Chinnari\Down
loads\new.py'
Enter user IDs: 101 205 101 309 205 410
Brute Force Duplicates: [101, 205]
Optimized Duplicates: [205, 101]
PS C:\Users\Chinnari\Downloads>
  
```