

Assignment-12.4

Name: Arjun Manoj

H.No:2303A52134

Batch:44

Task 1: Bubble Sort for Ranking Exam Scores

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

Prompt:

Implement Bubble Sort in Python to sort a list of student exam scores. Include inline comments explaining comparisons, swaps, and iteration passes.

Also implement an optimized version with early termination and analyze time complexity.

Code:

```
1 #Implement Bubble Sort in Python to sort a list of student scores.
2 #Insert inline comments explaining key operations such as comparisons, swaps, and iteration passes
3 def bubble_sort(student_scores):
4     n = len(student_scores)
5     # Outer loop for number of passes
6     for i in range(n):
7         # Inner loop for comparisons in each pass
8         for j in range(0, n - i - 1):
9             # Compare adjacent elements
10            if student_scores[j] > student_scores[j + 1]:
11                # Swap if they are in wrong order
12                student_scores[j], student_scores[j + 1] = student_scores[j + 1], student_scores[j]
13        return student_scores
14 # Example usage
15 scores = [85, 92, 78, 90, 88]
16 sorted_scores = bubble_sort(scores)
17 print("Sorted student scores:", sorted_scores)
18
19 #Identify early-termination conditions when the list becomes sorted
20 def bubble_sort_optimized(student_scores):
21     n = len(student_scores)
22     for i in range(n):
23         swapped = False # Flag to track if any swaps occurred in this pass
24         for j in range(0, n - i - 1):
25             if student_scores[j] > student_scores[j + 1]:
26                 student_scores[j], student_scores[j + 1] = student_scores[j + 1], student_scores[j]
27                 swapped = True # Mark that a swap occurred
28         if not swapped: # If no swaps occurred, the list is already sorted
29             break
30     return student_scores
31
32 #Provide a brief time complexity analysis
33 # The time complexity of Bubble Sort is O(n^2) in the worst and average cases,
34 # as it requires two nested loops to compare and swap elements. In the best case,
35 # when the list is already sorted, the time complexity is O(n)
36 # due to the early termination condition that checks for swaps.
```

Sorted student scores: [78, 85, 88, 90, 92]
PS C:\Users\varjun\OneDrive\Desktop\AI - Assist>

Observation:

Bubble Sort repeatedly compares adjacent elements and swaps them if they are in the wrong order.

After each pass, the largest unsorted element moves to its correct position.

In the optimized version, if no swaps occur during a pass, the algorithm stops early (early termination).

- Time Complexity:
- Best Case: $O(n)$ (already sorted list with optimization)
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$

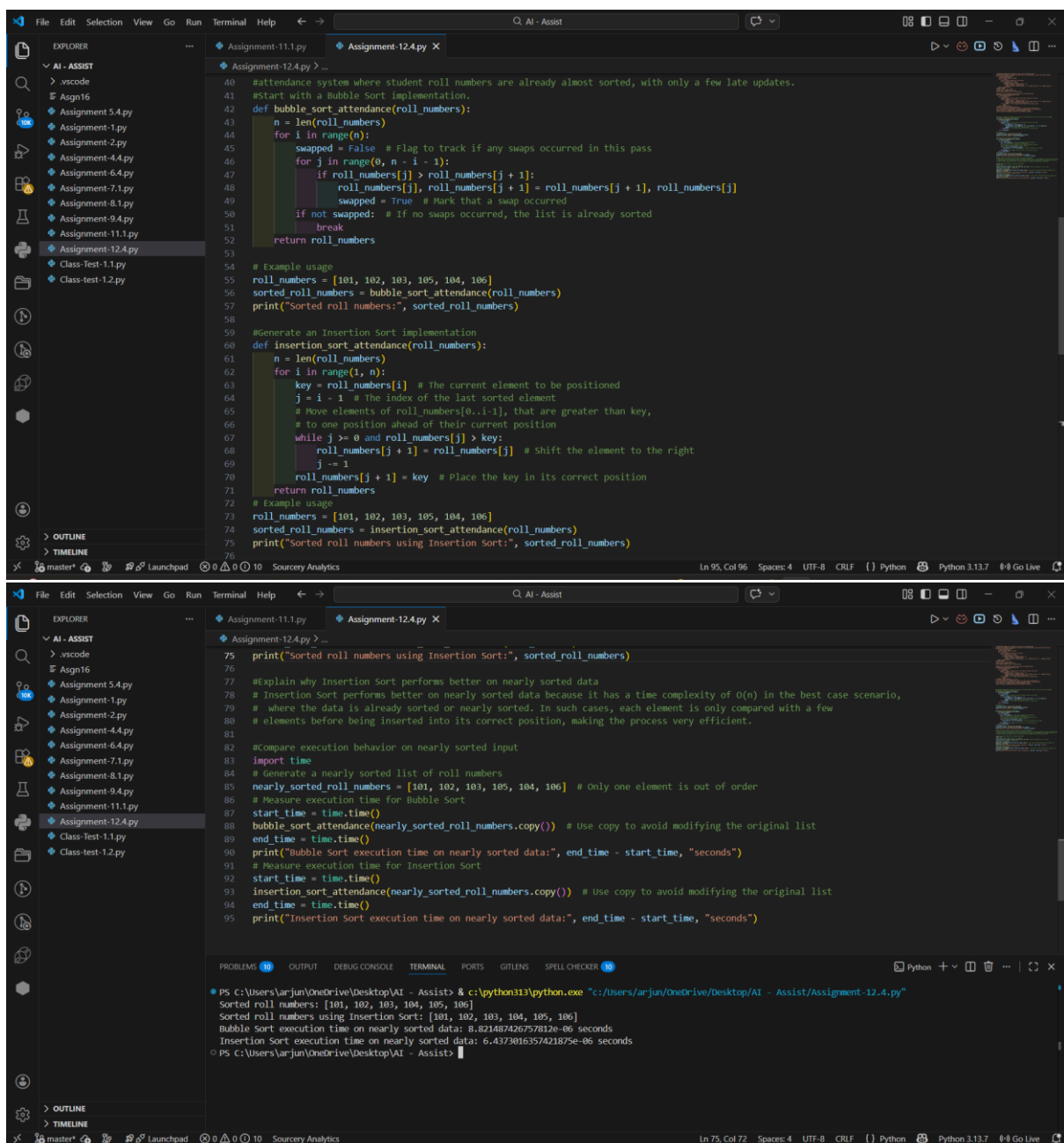
Task 2: Improving Sorting for Nearly Sorted Attendance Records

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

Prompt:

Implement Bubble Sort and Insertion Sort in Python to sort student roll numbers in an attendance system where the data is nearly sorted. Compare their performance on nearly sorted input and explain why one performs better.

Code:



```
40 #attendance system where student roll numbers are already almost sorted, with only a few late updates.
41 #start with a bubble sort implementation.
42 def bubble_sort_attendance(roll_numbers):
43     n = len(roll_numbers)
44     for i in range(n):
45         swapped = False # Flag to track if any swaps occurred in this pass
46         for j in range(0, n - i - 1):
47             if roll_numbers[j] > roll_numbers[j + 1]:
48                 roll_numbers[j], roll_numbers[j + 1] = roll_numbers[j + 1], roll_numbers[j]
49                 swapped = True # Mark that a swap occurred
50             if not swapped: # If no swaps occurred, the list is already sorted
51                 break
52     return roll_numbers
53
54 # Example usage
55 roll_numbers = [101, 102, 103, 105, 104, 106]
56 sorted_roll_numbers = bubble_sort_attendance(roll_numbers)
57 print("Sorted roll numbers:", sorted_roll_numbers)
58
59 #Generate an Insertion Sort implementation
60 def insertion_sort_attendance(roll_numbers):
61     n = len(roll_numbers)
62     for i in range(1, n):
63         key = roll_numbers[i] # The current element to be positioned
64         j = i - 1 # The index of the last sorted element
65         # Move elements of roll_numbers[0..i-1], that are greater than key,
66         # to one position ahead of their current position
67         while j >= 0 and roll_numbers[j] > key:
68             roll_numbers[j + 1] = roll_numbers[j] # Shift the element to the right
69             j -= 1
70         roll_numbers[j + 1] = key # Place the key in its correct position
71     return roll_numbers
72
73 # Example usage
74 roll_numbers = [101, 102, 103, 105, 104, 106]
75 sorted_roll_numbers = insertion_sort_attendance(roll_numbers)
76 print("Sorted roll numbers using Insertion Sort:", sorted_roll_numbers)
77
78 # Explain why Insertion Sort performs better on nearly sorted data
79 # Insertion Sort performs better on nearly sorted data because it has a time complexity of O(n) in the best case scenario,
80 # where the data is already sorted or nearly sorted. In such cases, each element is only compared with a few
81 # elements before being inserted into its correct position, making the process very efficient.
82
83 # Compare execution behavior on nearly sorted input
84 import time
85 # Generate a nearly sorted list of roll numbers
86 nearly_sorted_roll_numbers = [101, 102, 103, 105, 104, 106] # Only one element is out of order
87 # Measure execution time for Bubble Sort
88 start_time = time.time()
89 bubble_sort_attendance(nearly_sorted_roll_numbers.copy()) # Use copy to avoid modifying the original list
90 end_time = time.time()
91 print("Bubble Sort execution time on nearly sorted data:", end_time - start_time, "seconds")
92 # Measure execution time for Insertion Sort
93 start_time = time.time()
94 insertion_sort_attendance(nearly_sorted_roll_numbers.copy()) # Use copy to avoid modifying the original list
95 end_time = time.time()
96 print("Insertion Sort execution time on nearly sorted data:", end_time - start_time, "seconds")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GIT LENS SPELL CHECKER

```
PS C:\Users\arjun\OneDrive\Desktop\AI - Assist> & c:\python313\python.exe "c:/Users/arjun/OneDrive/Desktop/AI - Assist/Assignment-12.4.py"
Sorted roll numbers: [101, 102, 103, 104, 105, 106]
Sorted roll numbers using Insertion Sort: [101, 102, 103, 104, 105, 106]
Bubble sort execution time on nearly sorted data: 8.821487426757812e-06 seconds
Insertion Sort execution time on nearly sorted data: 6.4373016357421875e-06 seconds
PS C:\Users\arjun\OneDrive\Desktop\AI - Assist>
```

Observation:

Bubble Sort compares adjacent elements and swaps them repeatedly. Even with optimization, it may still require multiple passes.

Insertion Sort inserts each element into its correct position within the already sorted portion of the list.

For nearly sorted data, Insertion Sort performs fewer shifts and comparisons.

Time Complexity:

Bubble Sort:

Best Case: $O(n)$ (with early termination)

Average/Worst Case: $O(n^2)$

Insertion Sort:

Best Case: $O(n)$ (already/nearly sorted)

Average/Worst Case: $O(n^2)$

In practice, Insertion Sort executes faster than Bubble Sort for nearly sorted roll numbers because it minimizes unnecessary swaps.

Task 3: Searching Student Records in a Database

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

Prompt:

Implement Linear Search and Binary Search in Python to search student records by roll number.

Add proper docstrings explaining parameters and return values.

Compare both searching techniques and explain when Binary Search is applicable.

Code:

```
File Edit Selection View Go Run Terminal Help Q AI - Assist
EXPLORER
  AI - ASSIST
  > .vscode
  > Asgn16
  > Assignment-5.4.py
  > Assignment-1.py
  > Assignment-2.py
  > Assignment-4.4.py
  > Assignment-6.4.py
  > Assignment-7.1.py
  > Assignment-8.1.py
  > Assignment-9.4.py
  > Assignment-11.1.py
  > Assignment-12.4.py
  > Class-Test-1.1.py
  > Class-Test-1.2.py
  > OUTLINE
  > TIMELINE
  > master
  > Launchpad
  > 0 0 11 Sourcery Analytics
Assignment-11.1.py Assignment-12.4.py X
Assignment-12.4.py > ...
99 #student information portal where users search for student records by roll number.
100 #linear search for unsorted student data
101 #add docstrings explaining parameters and return values
102 def linear_search(student_records, roll_number):
103     """
104     Performs a linear search on a list of student records to find a record by roll number.
105
106     Parameters:
107         student_records (list): A list of dictionaries representing student records.
108         roll_number (int): The roll number to search for.
109
110     Returns:
111         dict or None: The matching student record if found, otherwise None.
112     """
113     for record in student_records:
114         if record['roll_number'] == roll_number: # compare the roll number with the current record
115             return record # return the matching record if found
116     return None # Return None if no matching record is found
117 # Example usage
118 student_records = [
119     {'roll_number': 101, 'name': 'Alice'},
120     {'roll_number': 102, 'name': 'Bob'},
121     {'roll_number': 103, 'name': 'Charlie'}]
122 roll_number_to_search = 102
123 result = linear_search(student_records, roll_number_to_search)
124 if result:
125     print("Student record found:", result)
126 else:
127     print("Student record not found.")
128 #Binary Search for sorted student data
129
130 def binary_search(student_records, roll_number):
131     """
132     Performs a binary search on a sorted list of student records to find a record by roll number.
133
134     Parameters:
135         student_records (list): A sorted list of dictionaries representing student records.
136         roll_number (int): The roll number to search for.
137
138     Returns:
139         dict or None: The matching student record if found, otherwise None.
140     """
141     left, right = 0, len(student_records) - 1
142     while left <= right:
143         mid = left + (right - left) // 2 # Calculate the middle index
144         if student_records[mid]['roll_number'] == roll_number: # Check if the middle record matches the roll number
145             return student_records[mid] # Return the matching record if found
146         elif student_records[mid]['roll_number'] < roll_number: # If the middle record's roll number is less than the target
147             left = mid + 1 # Search in the right half
148         else: # If the middle record's roll number is greater than the target
149             right = mid - 1 # Search in the left half
150     return None # Return None if no matching record is found
151 # Example usage
152 sorted_student_records = sorted(student_records, key=lambda x: x['roll_number']) #
153 roll_number_to_search = 102
154 result = binary_search(sorted_student_records, roll_number_to_search)
155 if result:
156     print("Student record found:", result)
157 else:
158     print("Student record not found.")
159
160 #Explain when Binary Search is applicable
161 # Binary Search is applicable when the data is sorted. It works by repeatedly dividing the search interval in half,
162 # which allows it to find the target value efficiently. If the data is not sorted, Binary Search cannot be used,
163 # and a different search algorithm, such as Linear Search, would be necessary.
164
165 #differences between the two searches
166 # Linear Search:
167 # - Time complexity: O(n) in worst case, O(1) in best case.
168 # - Space complexity: O(1).
169 # - Works on both sorted and unsorted data.
170
Ln 177, Col 70 Spaces: 4 UTF-8 CRLF Python Python 3.13.7 Go Live
```

```
File Edit Selection View Go Run Terminal Help Q AI - Assist
EXPLORER
  AI - ASSIST
  > .vscode
  > Asgn16
  > Assignment-5.4.py
  > Assignment-1.py
  > Assignment-2.py
  > Assignment-4.4.py
  > Assignment-6.4.py
  > Assignment-7.1.py
  > Assignment-8.1.py
  > Assignment-9.4.py
  > Assignment-11.1.py
  > Assignment-12.4.py
  > Class-Test-1.1.py
  > Class-Test-1.2.py
  > OUTLINE
  > TIMELINE
  > master
  > Launchpad
  > 0 0 11 Sourcery Analytics
Assignment-11.1.py Assignment-12.4.py X
Assignment-12.4.py > ...
130 def binary_search(student_records, roll_number):
131     """
132     student_records (list): A sorted list of dictionaries representing student records.
133     roll_number (int): The roll number to search for.
134
135     Returns:
136         dict or None: The matching student record if found, otherwise None.
137     """
138     left, right = 0, len(student_records) - 1
139     while left <= right:
140         mid = left + (right - left) // 2 # Calculate the middle index
141         if student_records[mid]['roll_number'] == roll_number: # Check if the middle record matches the roll number
142             return student_records[mid] # Return the matching record if found
143         elif student_records[mid]['roll_number'] < roll_number: # If the middle record's roll number is less than the target
144             left = mid + 1 # Search in the right half
145         else: # If the middle record's roll number is greater than the target
146             right = mid - 1 # Search in the left half
147     return None # Return None if no matching record is found
148 # Example usage
149 sorted_student_records = sorted(student_records, key=lambda x: x['roll_number']) #
150 roll_number_to_search = 102
151 result = binary_search(sorted_student_records, roll_number_to_search)
152 if result:
153     print("Student record found:", result)
154 else:
155     print("Student record not found.")
156
157 #Explain when Binary Search is applicable
158 # Binary Search is applicable when the data is sorted. It works by repeatedly dividing the search interval in half,
159 # which allows it to find the target value efficiently. If the data is not sorted, Binary Search cannot be used,
160 # and a different search algorithm, such as Linear Search, would be necessary.
161
162 #differences between the two searches
163 # Linear Search:
164 # - Time complexity: O(n) in worst case, O(1) in best case.
165 # - Space complexity: O(1).
166 # - Works on both sorted and unsorted data.
167
Ln 177, Col 70 Spaces: 4 UTF-8 CRLF Python Python 3.13.7 Go Live
```

```
162 # which allows it to find the target value efficiently. If the data is not sorted, Binary Search cannot be used,
163 # and a different search algorithm, such as Linear Search, would be necessary.
164
165
166 #differences between the two searches
167 # Linear Search:
168 # - Time Complexity: O(n) in worst case, O(1) in best case.
169 # - Space Complexity: O(1).
170 # - Works on both sorted and unsorted data.
171 # - Simple to implement and understand.
172
173 # Binary Search:
174 # - Time Complexity: O(log n).
175 # - Space Complexity: O(1).
176 # - Works only on sorted data.
177 # - More efficient for large datasets but requires data to be sorted.
```

```
PS C:\Users\arjun\OneDrive\Desktop\AI - Assist> & c:\python313\python.exe "c:/Users/arjun/OneDrive/Desktop/AI - Assist/Assignment-12.4.py"
Student record found: {'roll_number': 102, 'name': 'Bob'}
Student record found: {'roll_number': 102, 'name': 'Bob'}
```

Observation:

- Linear Search checks each student record one by one until the roll number is found.
- It works on both sorted and unsorted data.
- Time Complexity of Linear Search:
 - Best Case: $O(1)$
 - Worst Case: $O(n)$
- Binary Search repeatedly divides the sorted data into halves to find the roll number.
- It works only when the student records are sorted.
- Time Complexity of Binary Search:
 - Best/Worst Case: $O(\log n)$
- For small or unsorted student data, Linear Search is suitable.
- For large and sorted datasets, Binary Search is much more efficient.

Task 4: Choosing Between Quick Sort and Merge Sort for Data Processing

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).

Prompt:

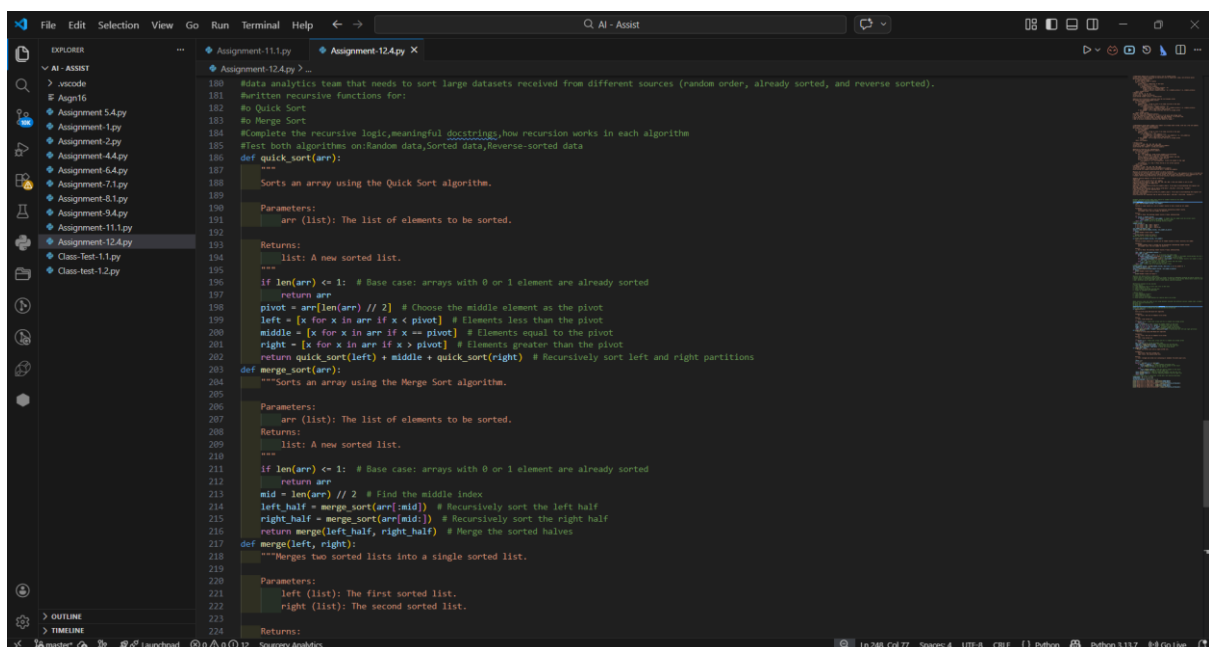
Implement recursive Quick Sort and Merge Sort algorithms in Python to sort large datasets.

Add meaningful docstrings explaining parameters and return values.

Explain how recursion works in each algorithm.

Test both algorithms on random, already sorted, and reverse-sorted data.

Code:



```
180 """Data analytics team that needs to sort large datasets received from different sources (random order, already sorted, and reverse sorted).
181 #Written recursive functions for:
182 #Quick Sort
183 #Merge Sort
184 #Complete the recursive logic, meaningful docstrings, how recursion works in each algorithm
185 #Test both algorithms on Random data, Sorted data, Reverse-sorted data
186 def quick_sort(arr):
187     """
188     Sorts an array using the Quick Sort algorithm.
189
190     Parameters:
191     arr (list): The list of elements to be sorted.
192
193     Returns:
194     list: A new sorted list.
195     """
196     if len(arr) <= 1: # Base case: arrays with 0 or 1 element are already sorted
197         return arr
198     pivot = arr[len(arr) // 2] # Choose the middle element as the pivot
199     left = [x for x in arr if x < pivot] # Elements less than the pivot
200     middle = [x for x in arr if x == pivot] # Elements equal to the pivot
201     right = [x for x in arr if x > pivot] # Elements greater than the pivot
202     return quick_sort(left) + middle + quick_sort(right) # Recursively sort left and right partitions
203 def merge_sort(arr):
204     """Sorts an array using the Merge Sort algorithm.
205
206     Parameters:
207     arr (list): The list of elements to be sorted.
208
209     Returns:
210     list: A new sorted list.
211     """
212     if len(arr) <= 1: # Base case: arrays with 0 or 1 element are already sorted
213         return arr
214     mid = len(arr) // 2 # Find the middle index
215     left_half = merge_sort(arr[:mid]) # Recursively sort the left half
216     right_half = merge_sort(arr[mid:]) # Recursively sort the right half
217     return merge(left_half, right_half) # Merge the sorted halves
218 def merge(left, right):
219     """Merges two sorted lists into a single sorted list.
220
221     Parameters:
222     left (list): The first sorted list.
223     right (list): The second sorted list.
224
225     Returns:
226     list: A new sorted list.
227     """
```



```
217 def merge(left, right):
218     """
219     Returns:
220     list: A merged and sorted list containing all elements from both input lists.
221     """
222     result = []
223     i = j = 0
224     while i < len(left) and j < len(right):
225         if left[i] < right[j]: # Compare elements from both lists
226             result.append(left[i]) # Add the smaller element to the result
227             i += 1 # Move the pointer in the left list
228         else:
229             result.append(right[j]) # Add the smaller element to the result
230             j += 1 # Move the pointer in the right list
231     result.extend(left[i:]) # Add any remaining elements from the left list
232     result.extend(right[j:]) # Add any remaining elements from the right list
233     return result
234
235 # Test both algorithms on random data, sorted data, and reverse-sorted data
236 random_data = [5, 2, 9, 1, 5, 6]
237 sorted_data = [1, 2, 3, 4, 5, 6]
238 reverse_sorted_data = [6, 5, 4, 3, 2, 1]
239 print("Quick Sort on random data:", quick_sort(random_data))
240 print("Quick Sort on sorted data:", quick_sort(sorted_data))
241 print("Quick Sort on reverse-sorted data:", quick_sort(reverse_sorted_data))
242 print("Merge Sort on random data:", merge_sort(random_data))
243 print("Merge Sort on sorted data:", merge_sort(sorted_data))
244 print("Merge Sort on reverse-sorted data:", merge_sort(reverse_sorted_data))
```

```
PS C:\Users\arjun\OneDrive\Desktop\AI - Assist> & c:\python313\python.exe "c:/Users/arjun/OneDrive/Desktop/AI - Assist/Assignment-12.4.py"
Quick Sort on sorted data: [1, 2, 3, 4, 5, 6]
Quick Sort on reverse-sorted data: [1, 2, 3, 4, 5, 6]
Merge Sort on random data: [1, 2, 5, 5, 6, 9]
Merge Sort on sorted data: [1, 2, 3, 4, 5, 6]
Merge Sort on reverse-sorted data: [1, 2, 3, 4, 5, 6]
```

Observation:

- Quick Sort and Merge Sort both use recursion and follow the Divide and Conquer approach.
- Quick Sort divides the list around a pivot and recursively sorts left and right parts.
- Merge Sort divides the list into halves and recursively merges them in sorted order.
- Merge Sort always runs in $O(n \log n)$ time.
- Quick Sort runs in $O(n \log n)$ on average but $O(n^2)$ in the worst case (bad pivot choice).

Task 5: Optimizing a Duplicate Detection Algorithm

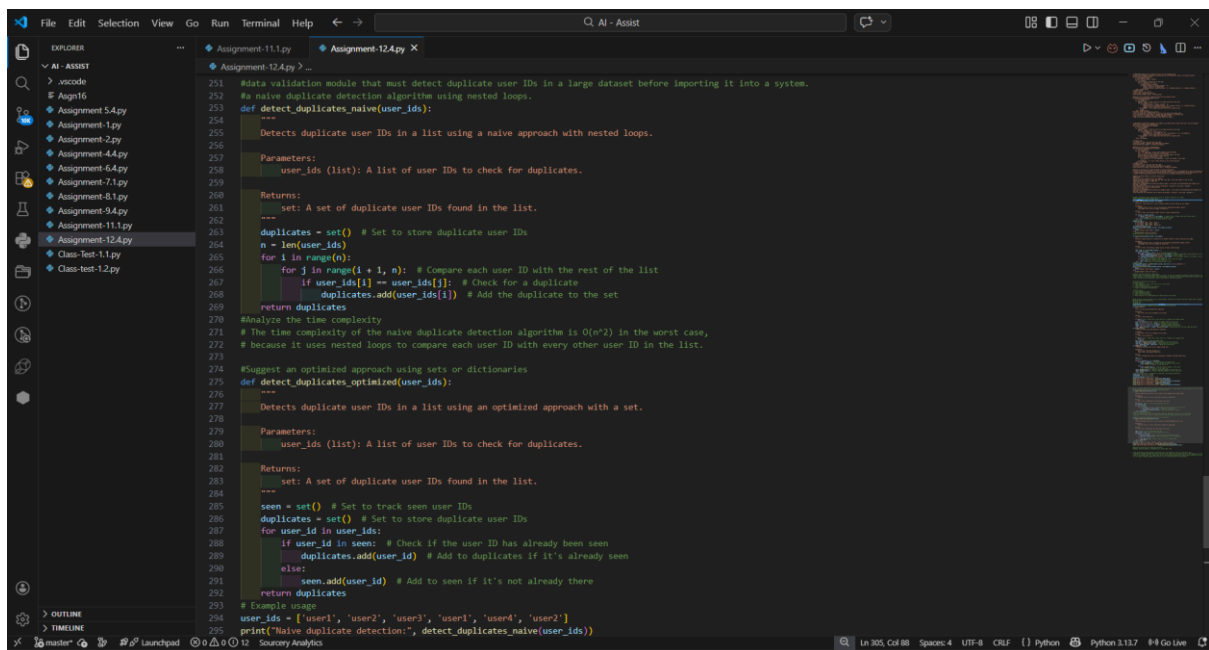
You are building a data validation module that must detect duplicate user IDs in a large dataset before importing it into a system.

Prompt:

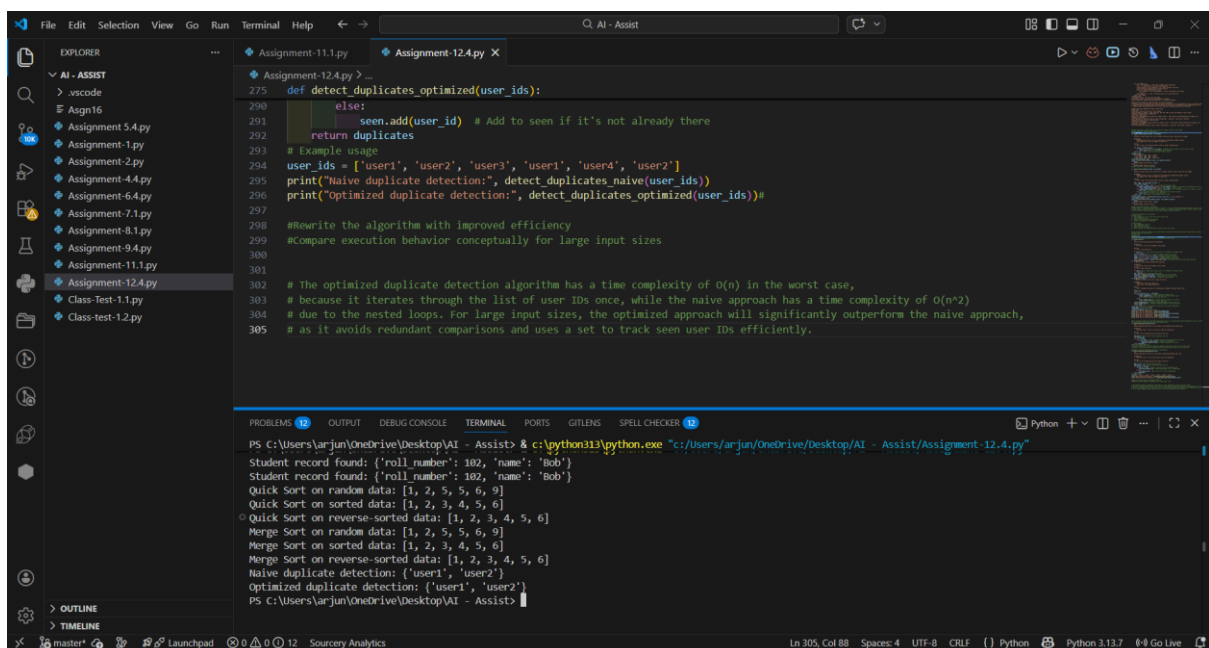
Implement a naive duplicate detection algorithm using nested loops to detect duplicate user IDs.

Analyze its time complexity. Then implement an optimized version using sets and compare their performance conceptually for large datasets.

Code:



```
251 #data validation module that must detect duplicate user IDs in a large dataset before importing it into a system.
252 #naive duplicate detection algorithm using nested loops.
253 def detect_duplicates_naive(user_ids):
254     """
255     Detects duplicate user IDs in a list using a naive approach with nested loops.
256
257     Parameters:
258     user_ids (list): A list of user IDs to check for duplicates.
259
260     Returns:
261     set: A set of duplicate user IDs found in the list.
262     """
263     duplicates = set() # Set to store duplicate user IDs
264     n = len(user_ids)
265     for i in range(n):
266         for j in range(i + 1, n): # Compare each user ID with the rest of the list
267             if user_ids[i] == user_ids[j]: # Check for a duplicate
268                 duplicates.add(user_ids[i]) # Add the duplicate to the set
269     return duplicates
270
271 #Analyze the time complexity
272 # The time complexity of the naive duplicate detection algorithm is O(n^2) in the worst case,
273 # because it uses nested loops to compare each user ID with every other user ID in the list.
274
275 #Suggest an optimized approach using sets or dictionaries
276 def detect_duplicates_optimized(user_ids):
277     """
278     Detects duplicate user IDs in a list using an optimized approach with a set.
279
280     Parameters:
281     user_ids (list): A list of user IDs to check for duplicates.
282
283     Returns:
284     set: A set of duplicate user IDs found in the list.
285     """
286     seen = set() # Set to track seen user IDs
287     duplicates = set() # Set to store duplicate user IDs
288     for user_id in user_ids:
289         if user_id in seen: # Check if the user ID has already been seen
290             duplicates.add(user_id) # Add to duplicates if it's already seen
291         else:
292             seen.add(user_id) # Add to seen if it's not already there
293     return duplicates
294
295 # Example usage
296 user_ids = ['user1', 'user2', 'user3', 'user1', 'user4', 'user2']
297 print("Naive duplicate detection:", detect_duplicates_naive(user_ids))
298 print("Optimized duplicate detection:", detect_duplicates_optimized(user_ids))
```



```
275 def detect_duplicates_optimized(user_ids):
276     """
277     Detects duplicate user IDs in a list using an optimized approach with a set.
278
279     Parameters:
280     user_ids (list): A list of user IDs to check for duplicates.
281
282     Returns:
283     set: A set of duplicate user IDs found in the list.
284     """
285     seen = set() # Set to track seen user IDs
286     duplicates = set() # Set to store duplicate user IDs
287     for user_id in user_ids:
288         if user_id in seen: # Check if the user ID has already been seen
289             duplicates.add(user_id) # Add to duplicates if it's already seen
290         else:
291             seen.add(user_id) # Add to seen if it's not already there
292     return duplicates
293
294 # Example usage
295 user_ids = ['user1', 'user2', 'user3', 'user1', 'user4', 'user2']
296 print("Naive duplicate detection:", detect_duplicates_naive(user_ids))
297 print("Optimized duplicate detection:", detect_duplicates_optimized(user_ids))
298
299 #Rewrite the algorithm with improved efficiency
300 #Compare execution behavior conceptually for large input sizes
301
302 # The optimized duplicate detection algorithm has a time complexity of O(n) in the worst case,
303 # because it iterates through the list of user IDs once, while the naive approach has a time complexity of O(n^2)
304 # due to the nested loops. For large input sizes, the optimized approach will significantly outperform the naive approach,
305 # as it avoids redundant comparisons and uses a set to track seen user IDs efficiently.
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS SPELL CHECKER

```
PS C:\Users\arjun\OneDrive\Desktop\AI - Assist> & c:\python313\python.exe "c:/Users/arjun/OneDrive/Desktop/AI - Assist/Assignment-12.4.py"
Student record found: {'roll_number': 102, 'name': 'Bob'}
Student record found: {'roll_number': 102, 'name': 'Bob'}
Quick Sort on random data: [1, 2, 5, 5, 6, 9]
Quick Sort on sorted data: [1, 2, 3, 4, 5, 6]
Quick Sort on reverse-sorted data: [1, 2, 3, 4, 5, 6]
Merge Sort on random data: [1, 2, 5, 5, 6, 9]
Merge Sort on sorted data: [1, 2, 3, 4, 5, 6]
Merge Sort on reverse-sorted data: [1, 2, 3, 4, 5, 6]
Naive duplicate detection: {'user1', 'user2'}
Optimized duplicate detection: {'user1', 'user2'}
```

Observation:

- The naive approach uses nested loops and compares every pair of user IDs.
- Its time complexity is $O(n^2)$, which is inefficient for large datasets.
- The optimized approach uses a set to track seen user IDs.
- Its time complexity is $O(n)$ since it scans the list only once.
- For large input sizes, the optimized method performs significantly faster and avoids unnecessary comparisons.