

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:

```

File Edit Selection View Go Run Terminal Help < > Q AI - Assist
EXPLORER ... Assignment-11.1.py Assignment-12.4.py ...
Assignment-12.4.py > ...
# Implement Bubble Sort in Python to sort a list of student scores.
# Insert inline comments explaining key operations such as comparisons, swaps, and iteration passes
def bubble_sort(student_scores):
    n = len(student_scores)
    # Outer loop for number of passes
    for i in range(n):
        # Inner loop for comparisons in each pass
        for j in range(0, n - i - 1):
            # Compare adjacent elements
            if student_scores[j] > student_scores[j + 1]:
                # Swap if they are in wrong order
                student_scores[j], student_scores[j + 1] = student_scores[j + 1], student_scores[j]
    return student_scores

# Example usage
scores = [85, 92, 78, 90, 88]
sorted_scores = bubble_sort(scores)
print("Sorted student scores:", sorted_scores)

# Identify early-termination conditions when the list becomes sorted
def bubble_sort_optimized(student_scores):
    n = len(student_scores)
    for i in range(n):
        swapped = False # Flag to track if any swaps occurred in this pass
        for j in range(0, n - i - 1):
            if student_scores[j] > student_scores[j + 1]:
                student_scores[j], student_scores[j + 1] = student_scores[j + 1], student_scores[j]
                swapped = True # Mark that a swap occurred
        if not swapped: # If no swaps occurred, the list is already sorted
            break
    return student_scores

# Provide a brief time complexity analysis
# The time complexity of Bubble Sort is O(n^2) in the worst and average cases,
# as it requires two nested loops to compare and swap elements. In the best case,
# when the list is already sorted, the time complexity is O(n)
# due to the early termination condition that checks for swaps.

PROBLEMS 19 OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS SPELL CHECKER 19
Sorted student scores: [78, 85, 88, 90, 92]
PS C:\Users\arjun\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:

The screenshot shows two versions of the assignment code in VS Code. The top window displays `Assignment-12.4.py`, which contains the following 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)
```

The bottom window displays `Assignment-11.1.py`, which contains the following code:

```
75     print("Sorted roll numbers using Insertion Sort:", sorted_roll_numbers)
76
77     #Explain why Insertion Sort performs better on nearly sorted data
78     # Insertion Sort performs better on nearly sorted data because it has a time complexity of O(n) in the best case scenario,
79     # where the data is already sorted or nearly sorted. In such cases, each element is only compared with a few
80     # elements before being inserted into its correct position, making the process very efficient.
81
82     #Compare execution behavior on nearly sorted input
83     import time
84     # Generate a nearly sorted list of roll numbers
85     nearly_sorted_roll_numbers = [101, 102, 103, 105, 104, 106] # Only one element is out of order
86     # Measure execution time for Bubble Sort
87     start_time = time.time()
88     bubble_sort_attendance(nearly_sorted_roll_numbers.copy()) # Use copy to avoid modifying the original list
89     end_time = time.time()
90     print("Bubble Sort execution time on nearly sorted data:", end_time - start_time, "seconds")
91     # Measure execution time for Insertion Sort
92     start_time = time.time()
93     insertion_sort_attendance(nearly_sorted_roll_numbers.copy()) # Use copy to avoid modifying the original list
94     end_time = time.time()
95     print("Insertion Sort execution time on nearly sorted data:", end_time - start_time, "seconds")
```

The terminal output shows the execution times for both sorts on nearly 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"
Sorted roll numbers: [101, 102, 103, 105, 104, 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:

The screenshot shows the VS Code interface with the AI-Assist extension active. The Explorer sidebar on the left lists files in the 'AI - ASSIST' folder, including Assignment-11.1.py, Assignment-12.4.py, Assignment-1.1.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, Class-Test-1.1.py, and Class-test-1.2.py. The main editor tab is open to Assignment-12.4.py, which contains Python code for a linear search function. A tooltip from the AI-Assist extension provides detailed documentation for the function, including its purpose, parameters, returns, and example usage.

```
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
117            return None # Return None if no matching record is found
118
119            # Example usage
120            student_records = [
121                {'roll_number': 101, 'name': 'Alice'},
122                {'roll_number': 102, 'name': 'Bob'},
123                {'roll_number': 103, 'name': 'Charlie'}
124            ]
125            roll_number_to_search = 102
126            result = linear_search(student_records, roll_number_to_search)
127            if result:
128                print("Student record found:", result)
129            else:
130                print("Student record not found.")
131
132            #Binary Search for sorted student data
133
134            def binary_search(student_records, roll_number):
135                """
136                    Performs a binary search on a sorted list of student records to find a record by roll number.
137
138                    Parameters:
139                        student_records (list): A sorted list of dictionaries representing student records.
140
141                    Returns:
142                        dict or None: The matching student record if found, otherwise None.
143
144                    left, right = 0, len(student_records) - 1
145                    while left <= right:
146                        mid = left + (right - left) // 2 # Calculate the middle index
147                        if student_records[mid]['roll_number'] == roll_number: # Check if the middle record matches the roll number
148                            return student_records[mid] # Return the matching record if found
149                        elif student_records[mid]['roll_number'] < roll_number: # If the middle record's roll number is less than the target
150                            left = mid + 1 # Search in the right half
151                        else: # If the middle record's roll number is greater than the target
152                            right = mid - 1 # Search in the left half
153
154                    return None # Return None if no matching record is found
155
156            # Example usage
157            sorted_student_records = sorted(student_records, key=lambda x: x['roll_number']) #
158            roll_number_to_search = 102
159            result = binary_search(sorted_student_records, roll_number_to_search)
160            if result:
161                print("Student record found:", result)
162            else:
163                print("Student record not found.")
164
165            #Explain when Binary Search is applicable
166            # Binary Search is applicable when the data is sorted. It works by repeatedly dividing the search interval in half,
167            # which allows it to find the target value efficiently. If the data is not sorted, Binary Search cannot be used,
168            # and a different search algorithm, such as Linear Search, would be necessary.
169
170            #differences between the two searches
171            # - Linear Search:
172            #   - Time Complexity: O(n) in worst case, O(1) in best case.
173            #   - Space Complexity: O(1).
174            # - Binary Search:
175            #   - Works on both sorted and unsorted data.
```

This screenshot shows the same VS Code environment with the AI-Assist extension active. The main editor tab is now open to Assignment-12.4.py, displaying the binary search function. A tooltip from the AI-Assist extension provides detailed documentation for the function, highlighting its applicability to sorted data and comparing it to linear search.

```
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
148                    return None # Return None if no matching record is found
149
150            # Example usage
151            sorted_student_records = sorted(student_records, key=lambda x: x['roll_number']) #
152            roll_number_to_search = 102
153            result = binary_search(sorted_student_records, roll_number_to_search)
154            if result:
155                print("Student record found:", result)
156            else:
157                print("Student record not found.")
158
159            #Explain when Binary Search is applicable
160            # Binary Search is applicable when the data is sorted. It works by repeatedly dividing the search interval in half,
161            # which allows it to find the target value efficiently. If the data is not sorted, Binary Search cannot be used,
162            # and a different search algorithm, such as Linear Search, would be necessary.
163
164            #differences between the two searches
165            # - Linear Search:
166            #   - Time Complexity: O(n) in worst case, O(1) in best case.
167            #   - Space Complexity: O(1).
168            # - Binary Search:
169            #   - Works on both sorted and unsorted data.
```

The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows files in the "AI - ASSIST" folder, including Assignment-12.4.py.
- Code Editor:** Displays the content of Assignment-12.4.py, which includes comments about the differences between Linear and Binary Search.
- Terminal:** Shows the command PS C:\Users\arjun\OneDrive\Desktop\AI - Assist> & c:\python313\python.exe "c:/Users/arjun/OneDrive/Desktop/AI - Assist/Assignment-12.4.py". The output indicates a student record was found: {"roll number": 102, "name": "Bob"}.
- Status Bar:** Shows the file is Line 177, Column 70, with 11 spaces, using UTF-8 encoding, and is a Python 3.13.7 file.

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:

The screenshot shows the VS Code interface with the AI-Assist extension active. The left sidebar has sections for EXPLORER, OUTLINE, and TIMELINE. The main editor area displays two files: Assignment-11.py and Assignment-12.py. The code is for data analytics, involving recursive functions for Quick Sort and Merge Sort. A tooltip from the AI-Assist extension provides detailed documentation for the quick sort function, including its parameters (arr: list), returns (list: A new sorted list), and implementation. The right side of the screen shows a preview of the generated documentation or further code snippets.

```
File Edit Selection View Go Run Terminal Help < > Q AI - Assist
EXPLORER ... Assignment-11.py Assignment-12.py x
AI - ASSIST > .vscode
E Asgn16
Assignment-5.py
Assignment-1.py
Assignment-2.py
Assignment-4.py
Assignment-6.py
Assignment-7.py
Assignment-8.py
Assignment-9.py
Assignment-11.py
Assignment-12.py
Class-Test-1.py
Class-test-1.py
Assignment-11.py
Assignment-12.py
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 #o Quick Sort
183 #o 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
187 def quick_sort(arr):
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
197     if len(arr) <= 1: # Base case: arrays with 0 or 1 element are already sorted
198         return arr
199
200     pivot = arr[len(arr) // 2] # Choose the middle element as the pivot
201     left = [x for x in arr if x < pivot] # Elements less than the pivot
202     middle = [x for x in arr if x == pivot] # Elements equal to the pivot
203     right = [x for x in arr if x > pivot] # Elements greater than the pivot
204     return quick_sort(left) + middle + quick_sort(right) # Recursively sort left and right partitions
205
206 def merge_sort(arr):
207     """Sorts an array using the Merge Sort algorithm.
208
209     Parameters:
210         arr (list): The list of elements to be sorted.
211
212     Returns:
213         list: A new sorted list.
214     """
215
216     if len(arr) <= 1: # Base case: arrays with 0 or 1 element are already sorted
217         return arr
218
219     mid = len(arr) // 2 # Find the middle index
220     left_half = merge_sort(arr[:mid]) # Recursively sort the left half
221     right_half = merge_sort(arr[mid:]) # Recursively sort the right half
222     return merge(left_half, right_half) # Merge the sorted halves
223
224 def merge(left, right):
225     """Merges two sorted lists into a single sorted list.
226
227     Parameters:
228         left (list): The first sorted list.
229         right (list): The second sorted list.
230
231     Returns:
232         list: The merged sorted list.
233     """
234
235     i = j = 0
236     result = []
237
238     while i < len(left) and j < len(right):
239         if left[i] < right[j]:
240             result.append(left[i])
241             i += 1
242         else:
243             result.append(right[j])
244             j += 1
245
246     result += left[i:]
247     result += right[j:]
248
249     return result
```

```

File Edit Selection View Go Run Terminal Help ← → Q: AI - Assist
EXPLORER ... Assignment-11.1.py Assignment-12.4.py
AI - ASSIST > .vscode
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
Assignment-12.4.py
Returns:
list: A merged and sorted list containing all elements from both input lists.
...
result = []
i = j = 0
while i < len(left) and j < len(right):
    if left[i] < right[j]: # Compare elements from both lists
        result.append(left[i]) # Add the smaller element to the result
        i += 1 # Move the pointer in the left list
    else:
        result.append(right[j]) # Add the smaller element to the result
        j += 1 # Move the pointer in the right list
result.extend(left[i:]) # Add any remaining elements from the left list
result.extend(right[j:]) # Add any remaining elements from the right list
return result
# Test both algorithms on random data, sorted data, and reverse-sorted data
random_data = [5, 2, 9, 1, 5, 6]
sorted_data = [1, 2, 3, 4, 5, 6]
reverse_sorted_data = [6, 5, 4, 3, 2, 1]
print("Quick Sort on random data:", quick_sort(random_data))
print("Quick Sort on sorted data:", quick_sort(sorted_data))
print("Quick Sort on reverse-sorted data:", quick_sort(reverse_sorted_data))
print("Merge Sort on random data:", merge_sort(random_data))
print("Merge Sort on sorted data:", merge_sort(sorted_data))
print("Merge Sort on reverse-sorted data:", merge_sort(reverse_sorted_data))

```

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"

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, 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]

PS C:\Users\Arjun\OneDrive\Desktop\AI - Assist>

Ln 248, Col 77 Spaces: 4 UFT-8 CRLF () Python Python 3.13.7 ⓘ Go Live

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

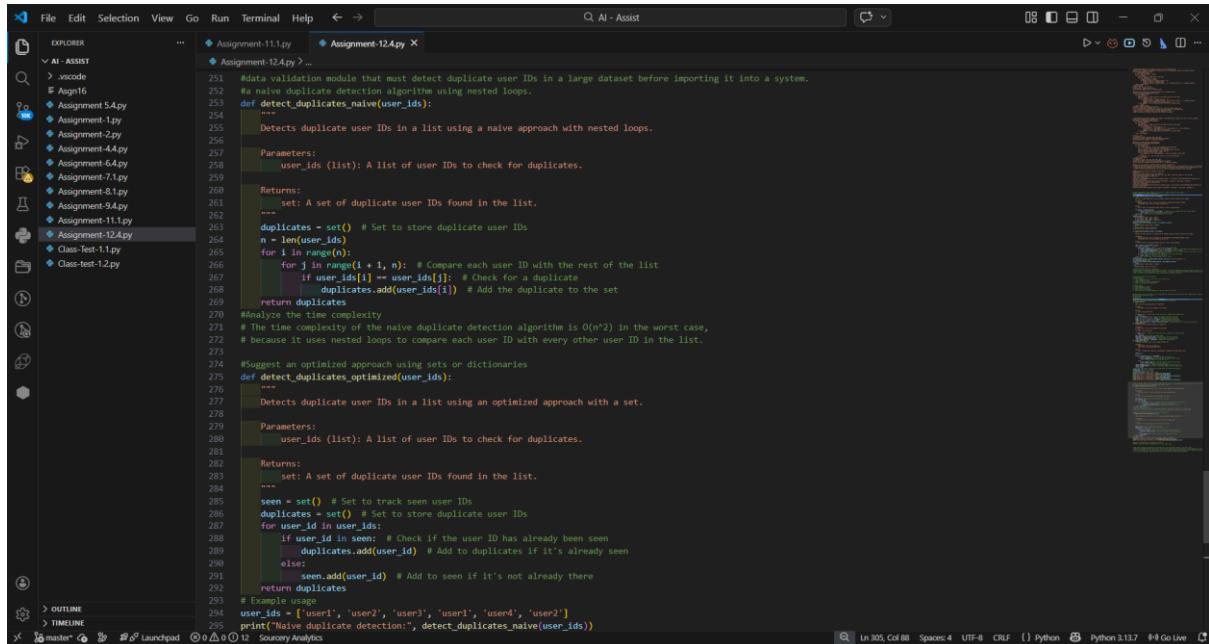
You are building a data validation module that must detect duplicate user IDs in a large dataset before importing it into a system.

Prompt:

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

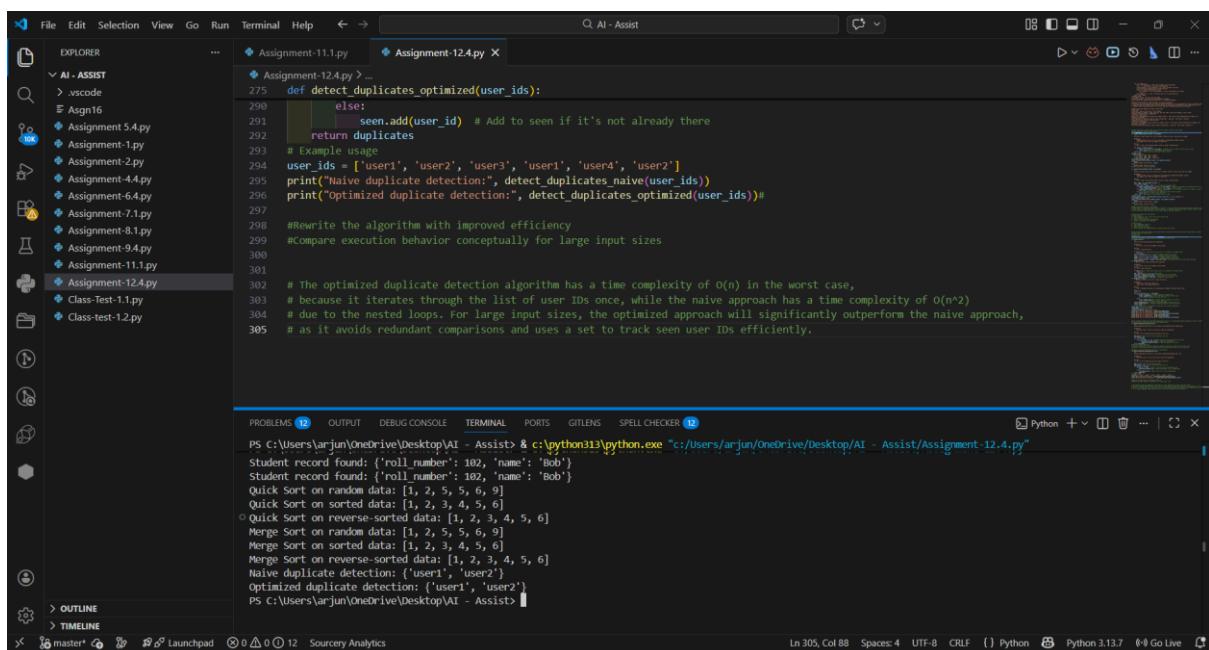
Analyze its time complexity. Then implement an optimized version using sets and compare their performance conceptually for large datasets.

Code:



The screenshot shows the VS Code interface with the Explorer sidebar open, displaying files like Assignment-11.py, Assignment-12.4.py, and Assignment-12.4.py (the active file). The code editor contains the following Python script:

```
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
270     return duplicates
271
272 # Analyze the time complexity
273 # The time complexity of the naive duplicate detection algorithm is O(n^2) in the worst case,
274 # because it uses nested loops to compare each user ID with every other user ID in the list.
275
276 # Suggest an optimized approach using sets or dictionaries
277 def detect_duplicates_optimized(user_ids):
278     """
279     Detects duplicate user IDs in a list using an optimized approach with a set.
280     """
281     Parameters:
282         user_ids (list): A list of user IDs to check for duplicates.
283
284     Returns:
285         set: A set of duplicate user IDs found in the list.
286
287     seen = set() # Set to track seen user IDs
288     duplicates = set() # Set to store duplicate user IDs
289     for user_id in user_ids:
290         if user_id in seen: # Check if the user ID has already been seen
291             duplicates.add(user_id) # Add to duplicates if it's already seen
292         else:
293             seen.add(user_id) # Add to seen if it's not already there
294
295     return duplicates
296
297 # Example usage
298 user_ids = ['user1', 'user2', 'user3', 'user1', 'user4', 'user2']
299 print("Naive duplicate detection:", detect_duplicates_naive(user_ids))
300 print("Optimized duplicate detection:", detect_duplicates_optimized(user_ids))
```



The screenshot shows the VS Code interface with the Explorer sidebar open, displaying files like Assignment-11.1.py, Assignment-12.4.py, and Assignment-12.4.py (the active file). The code editor contains the following Python script:

```
275 def detect_duplicates_optimized(user_ids):
290     """
291         else:
292             seen.add(user_id) # Add to seen if it's not already there
293
294     return duplicates
295
296 # Example usage
297 user_ids = ['user1', 'user2', 'user3', 'user1', 'user4', 'user2']
298 print("Naive duplicate detection:", detect_duplicates_naive(user_ids))
299 print("Optimized duplicate detection:", detect_duplicates_optimized(user_ids))# 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.
```

The terminal below shows the execution of the script:

```
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, 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, 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.