

ASSIGNMENT 12.4

NAME: K.HITHESH

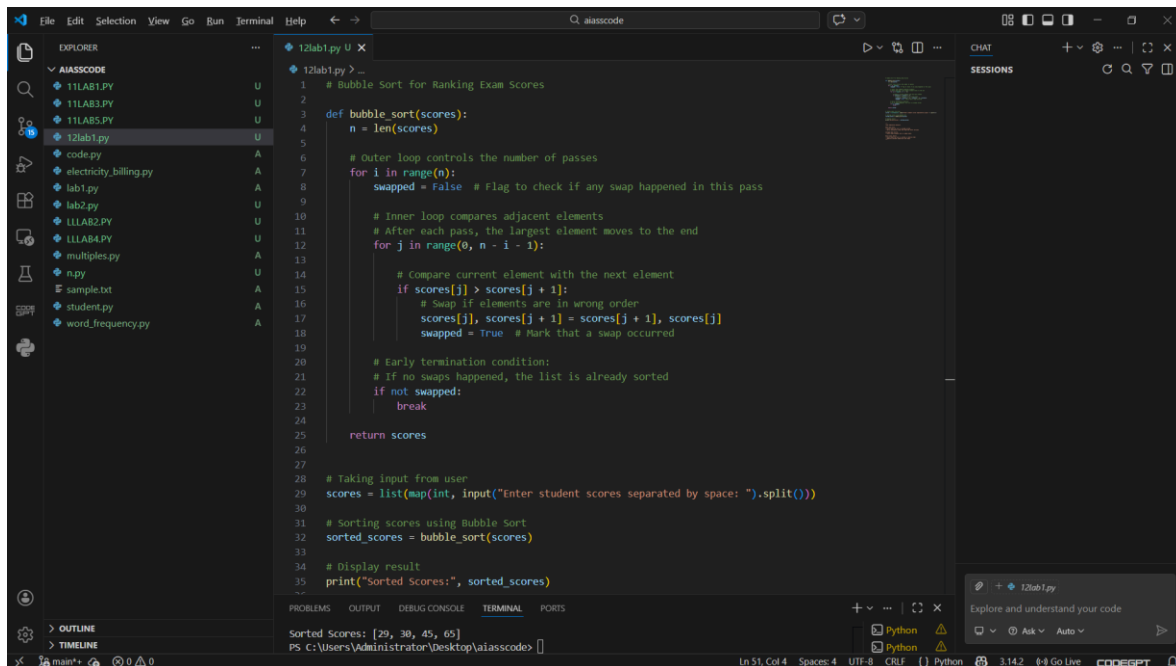
HALLTICKET: 2303A51291

TASK1:

Prompt

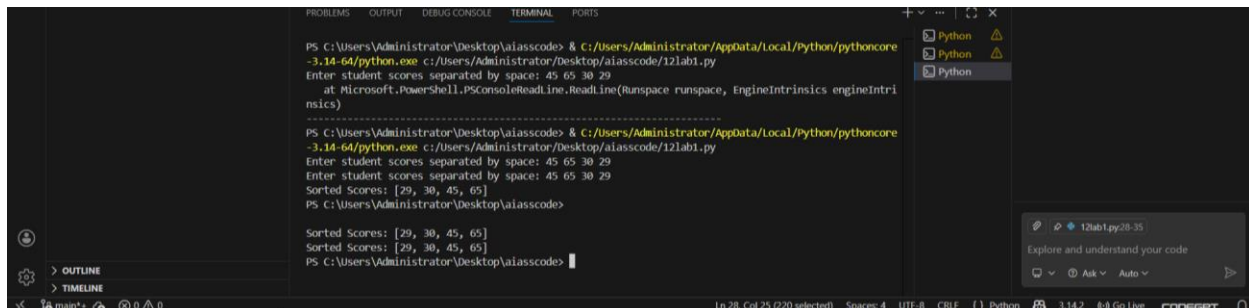
Generate a Python program that implements Bubble Sort to sort a list of student exam scores. Add inline comments explaining comparisons, swaps, and iteration passes. Include an early termination condition if the list becomes sorted before completing all passes. Also include a brief time complexity explanation in comments and show sample input/output.

Code:



```
1 # Bubble Sort for Ranking Exam Scores
2
3 def bubble_sort(scores):
4     n = len(scores)
5
6     # Outer loop controls the number of passes
7     for i in range(n):
8         swapped = False # Flag to check if any swap happened in this pass
9
10        # Inner loop compares adjacent elements
11        # After each pass, the largest element moves to the end
12        for j in range(0, n - i - 1):
13
14            # Compare current element with the next element
15            if scores[j] > scores[j + 1]:
16                # Swap if elements are in wrong order
17                scores[j], scores[j + 1] = scores[j + 1], scores[j]
18                swapped = True # Mark that a swap occurred
19
20        # Early termination condition:
21        # If no swaps happened, the list is already sorted
22        if not swapped:
23            break
24
25    return scores
26
27 # Taking input from user
28 scores = list(map(int, input("Enter student scores separated by space: ").split()))
29
30 # Sorting scores using Bubble Sort
31 sorted_scores = bubble_sort(scores)
32
33 # Display result
34 print("Sorted Scores:", sorted_scores)
```

Output:



```
PS C:\Users\Administrator\Desktop\aiasscode> & C:/Users/Administrator/AppData/Local/Python/pythoncore-3.14-64/python.exe c:/Users/Administrator/Desktop/aiasscode/12lab1.py
Enter student scores separated by space: 45 65 30 29
Sorted Scores: [29, 30, 45, 65]
PS C:\Users\Administrator\Desktop\aiasscode>
```

Explanation:

Bubble Sort repeatedly compares adjacent elements and swaps them if they are in the wrong order. After each pass, the largest element moves to its correct position at the end. Even with early termination, it still performs many comparisons, so it is not very efficient for nearly sorted data.

TASK2:

Prompt

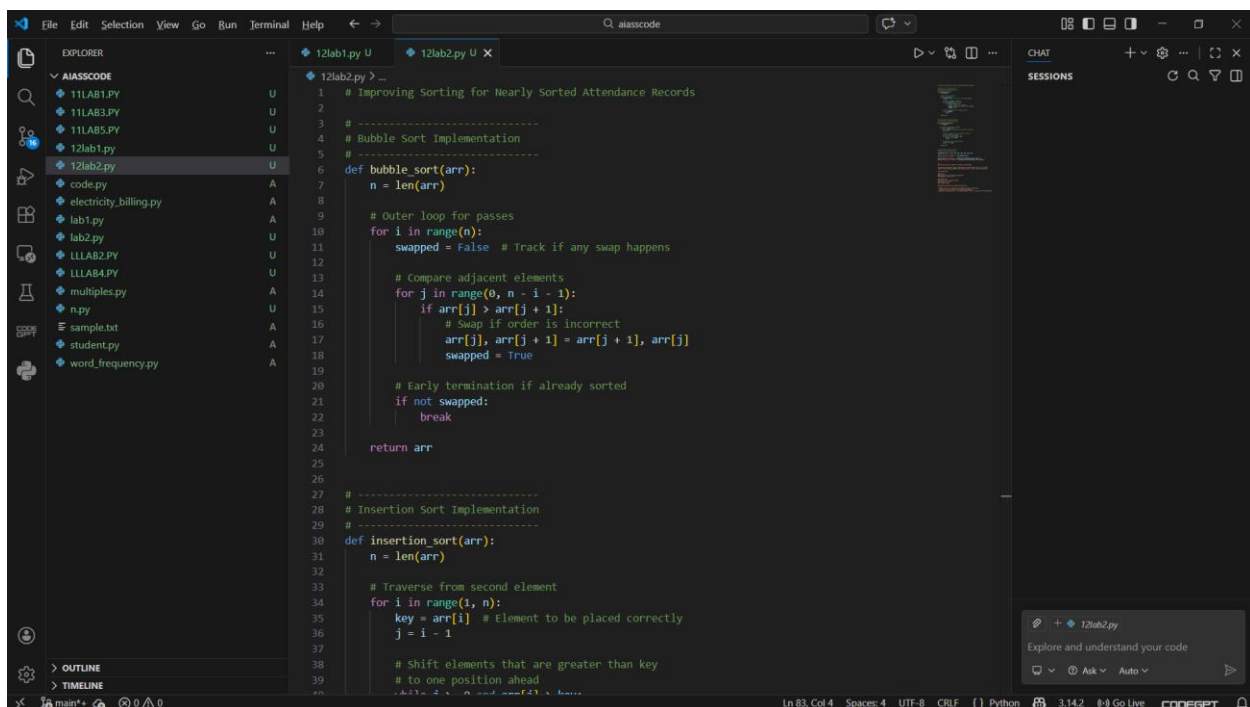
I have an attendance system where student roll numbers are already nearly sorted, with only a few late updates.

Start with a Bubble Sort implementation, then review the scenario and suggest a more suitable sorting algorithm.

Generate Python code for both Bubble Sort and Insertion Sort with inline comments.

Explain why Insertion Sort performs better on nearly sorted data and compare execution behavior on nearly sorted input.

Code:



```
1 # Improving Sorting for Nearly Sorted Attendance Records
2
3 # -----
4 # Bubble Sort Implementation
5 # -----
6 def bubble_sort(arr):
7     n = len(arr)
8
9     # Outer loop for passes
10    for i in range(n):
11        swapped = False # Track if any swap happens
12
13        # Compare adjacent elements
14        for j in range(0, n - i - 1):
15            if arr[j] > arr[j + 1]:
16                # Swap if order is incorrect
17                arr[j], arr[j + 1] = arr[j + 1], arr[j]
18                swapped = True
19
20    # Early termination if already sorted
21    if not swapped:
22        break
23
24    return arr
25
26
27 # -----
28 # Insertion Sort Implementation
29 # -----
30 def insertion_sort(arr):
31     n = len(arr)
32
33     # Traverse from second element
34     for i in range(1, n):
35         key = arr[i] # Element to be placed correctly
36         j = i - 1
37
38         # Shift elements that are greater than key
39         # to one position ahead
40         while j > 0 and arr[j] > key:
```

```
def bubble_sort(arr):
    while True:
        break
    return arr

# -----
# Insertion Sort Implementation
# -----
def insertion_sort(arr):
    n = len(arr)

    # Traverse from second element
    for i in range(1, n):
        key = arr[i] # Element to be placed correctly
        j = i - 1

        # Shift elements that are greater than key
        # to one position ahead
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1

        # Place key at correct position
        arr[j + 1] = key

    return arr

# -----
# Sample Nearly Sorted Input
# -----
attendance_rolls = [101, 102, 103, 105, 104, 106, 107]

print("Original Attendance:", attendance_rolls)
print("Bubble Sort Result:", bubble_sort(attendance_rolls.copy()))
print("Insertion Sort Result:", insertion_sort(attendance_rolls.copy()))
```

Output

```
at Microsoft.PowerShell.PSConsoleReadLine.InputLoop()
at Microsoft.PowerShell.PSConsoleReadLine.ReadLine(Runspace runspace, EngineIntrinsics engineIntrinsics)
PS C:\Users\Administrator\Desktop\aiasscode> & C:/Users/Administrator/AppData/Local/Python/pythoncore-3.14-64/python.exe c:/Users/Administrator/Desktop/aiasscode/12lab2.py
Original Attendance: [101, 102, 103, 105, 104, 106, 107]
PS C:\Users\Administrator\Desktop\aiasscode> & C:/Users/Administrator/AppData/Local/Python/pythoncore-3.14-64/python.exe c:/Users/Administrator/Desktop/aiasscode/12lab2.py
Original Attendance: [101, 102, 103, 105, 104, 106, 107]
Bubble Sort Result: [101, 102, 103, 104, 105, 106, 107]
Insertion Sort Result: [101, 102, 103, 104, 105, 106, 107]
PS C:\Users\Administrator\Desktop\aiasscode>
```

Explanation:

Insertion Sort takes one element at a time and places it in its correct position within the already sorted portion of the list. Since nearly sorted data already has most elements in place, only a few shifts are needed, making it faster and more efficient than Bubble Sort in this scenario

Task3:

Prompt

Create a Python program to search student records by roll number.

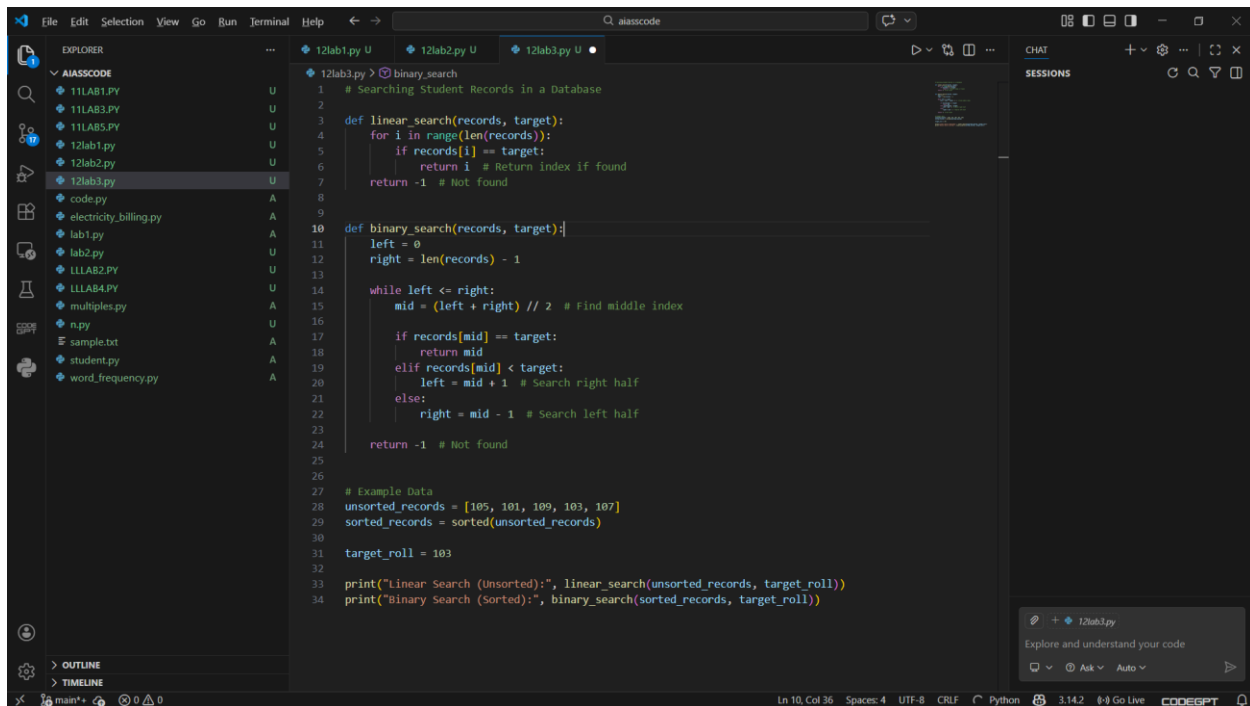
Implement Linear Search for unsorted data and Binary Search for sorted data.

Add docstrings explaining parameters and return values.

Explain when Binary Search is applicable and compare performance differences between Linear and Binary Search.

Include time complexity and a short observation comparing results on sorted vs unsorted lists.

Code:



```
1 # Searching Student Records in a Database
2
3 def linear_search(records, target):
4     for i in range(len(records)):
5         if records[i] == target:
6             return i # Return index if found
7     return -1 # Not found
8
9
10 def binary_search(records, target):
11     left = 0
12     right = len(records) - 1
13
14     while left <= right:
15         mid = (left + right) // 2 # Find middle index
16
17         if records[mid] == target:
18             return mid
19         elif records[mid] < target:
20             left = mid + 1 # Search right half
21         else:
22             right = mid - 1 # Search left half
23
24     return -1 # Not found
25
26
27 # Example Data
28 unsorted_records = [105, 101, 109, 103, 107]
29 sorted_records = sorted(unsorted_records)
30
31 target_roll = 103
32
33 print("Linear Search (Unsorted):", linear_search(unsorted_records, target_roll))
34 print("Binary Search (Sorted):", binary_search(sorted_records, target_roll))
```

Output:

```
at Microsoft.PowerShell.PSConsoleReadLine.ReadLine(runspace runspace, EngineIntrinsics engineIntrinsics)
at Microsoft.PowerShell.PSConsoleReadLine.ProcessOneKey(ConsoleKeyInfo key, Dictionary`2 dispatchTable, Boolean ignoreIfNoAction, Object arg)
at Microsoft.PowerShell.PSConsoleReadLine.InputLoop()
at Microsoft.PowerShell.PSConsoleReadLine.ReadLine(runspace runspace, EngineIntrinsics engineIntrinsics)
at Microsoft.PowerShell.PSConsoleReadLine.ReadLine(runspace runspace, EngineIntrinsics engineIntrinsics)
PS C:\Users\Administrator\Desktop\aiasscode> & C:/Users/Administrator/AppData/Local/Python/pythoncore-3.14-64/python.exe
c:\Users\Administrator\Desktop\aiasscode\12lab3.py
Linear Search (Unsorted): 3
Binary Search (Sorted): 1
PS C:\Users\Administrator\Desktop\aiasscode>
```

Explanation:

Binary Search is used to find a student roll number in a **sorted list** of records. It works by comparing the target value with the middle element and then repeatedly dividing the search range into halves until the element is found or the search space becomes empty. This method is faster than Linear Search for large datasets because it reduces the number of comparisons significantly (time complexity **$O(\log n)$**)

Task4:

Prompt

I have partially written recursive functions for Quick Sort and Merge Sort. Complete the recursive logic, add meaningful docstrings explaining parameters and return values, and explain how recursion works in each algorithm. Then test both algorithms on random data, sorted data, and reverse-sorted data. Also provide a comparison of time complexities and practical scenarios where one algorithm is preferred over the other

Code:

```
1 import random
2
3 # -----
4 # Quick Sort Implementation
5 # -----
6 def quick_sort(arr):
7     # Base case: list with 0 or 1 element is already sorted
8     if len(arr) <= 1:
9         return arr
10
11     pivot = arr[len(arr) // 2] # Choose middle element as pivot
12
13     # Divide elements into three groups
14     left = [x for x in arr if x < pivot]
15     middle = [x for x in arr if x == pivot]
16     right = [x for x in arr if x > pivot]
17
18     # Recursively sort left and right parts
19     return quick_sort(left) + middle + quick_sort(right)
20
21
22 # -----
23 # Merge Sort Implementation
24 # -----
25 def merge_sort(arr):
26     # Base case
27     if len(arr) <= 1:
28         return arr
29
30     mid = len(arr) // 2
31
32     # Divide array into two halves
33     left_half = merge_sort(arr[:mid])
34     right_half = merge_sort(arr[mid:])
35
36     # Merge the sorted halves
37     return merge(left_half, right_half)
38
39
```

The screenshot shows the Visual Studio Code editor with a Python file named `12lab4.py` open. The code implements a merge sort algorithm. The `merge` function takes two sorted lists, `left` and `right`, and merges them into a new sorted list. The `merge_sort` function recursively sorts the input list by repeatedly dividing it into halves and then merging the sorted halves back together. The code includes test data generation and printing the results of the sorting process.

```
def merge(left, right):
    i = j = 0
    # Compare elements and merge in sorted order
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    # Add remaining elements
    result.extend(left[i:])
    result.extend(right[j:])
    return result

# -----
# Testing Data
# -----
random_data = random.sample(range(1, 100), 10)
sorted_data = sorted(random_data)
reverse_data = sorted(random_data, reverse=True)

print("Random Data:", random_data)
print("Quick Sort:", quick_sort(random_data))
print("Merge Sort:", merge_sort(random_data))

print("\nSorted Data:", sorted_data)
print("Quick Sort:", quick_sort(sorted_data))
print("Merge Sort:", merge_sort(sorted_data))

print("\nReverse Sorted Data:", reverse_data)
print("Quick Sort:", quick_sort(reverse_data))
print("Merge Sort:", merge_sort(reverse_data))
```

Output:

The screenshot shows the Visual Studio Code editor with the same Python file `12lab4.py` open. The output of the program is displayed in the terminal window at the bottom. The output shows the results of the sorting process for random, sorted, and reverse sorted data, comparing the performance of Quick Sort and Merge Sort.

```
PS C:\Users\Administrator\Desktop\aiasscode> & C:\Users\Administrator\AppData\Local\Python\pythoncore-3.14-64\python.exe c:\Users\Administrator\Desktop\aiasscode\12lab4.py
Random Data: [20, 89, 73, 40, 34, 92, 65, 15, 60, 29]
Quick Sort: [15, 20, 29, 34, 40, 60, 65, 73, 89, 92]
Merge Sort: [15, 20, 29, 34, 40, 60, 65, 73, 89, 92]

Sorted Data: [15, 20, 29, 34, 40, 60, 65, 73, 89, 92]
Quick Sort: [15, 20, 29, 34, 40, 60, 65, 73, 89, 92]
Merge Sort: [15, 20, 29, 34, 40, 60, 65, 73, 89, 92]

Reverse Sorted Data: [92, 89, 73, 65, 60, 40, 34, 29, 20, 15]
Quick Sort: [15, 20, 29, 34, 40, 60, 65, 73, 89, 92]
Merge Sort: [15, 20, 29, 34, 40, 60, 65, 73, 89, 92]
PS C:\Users\Administrator\Desktop\aiasscode>
```

Explanation:

Quick Sort:

It selects a pivot element, divides the list into smaller parts based on the pivot, and recursively sorts those parts until the list is fully sorted.

Merge Sort:

It divides the list into halves repeatedly until single elements remain, then merges them back together in sorted order using recursion

Task5:

Prompt

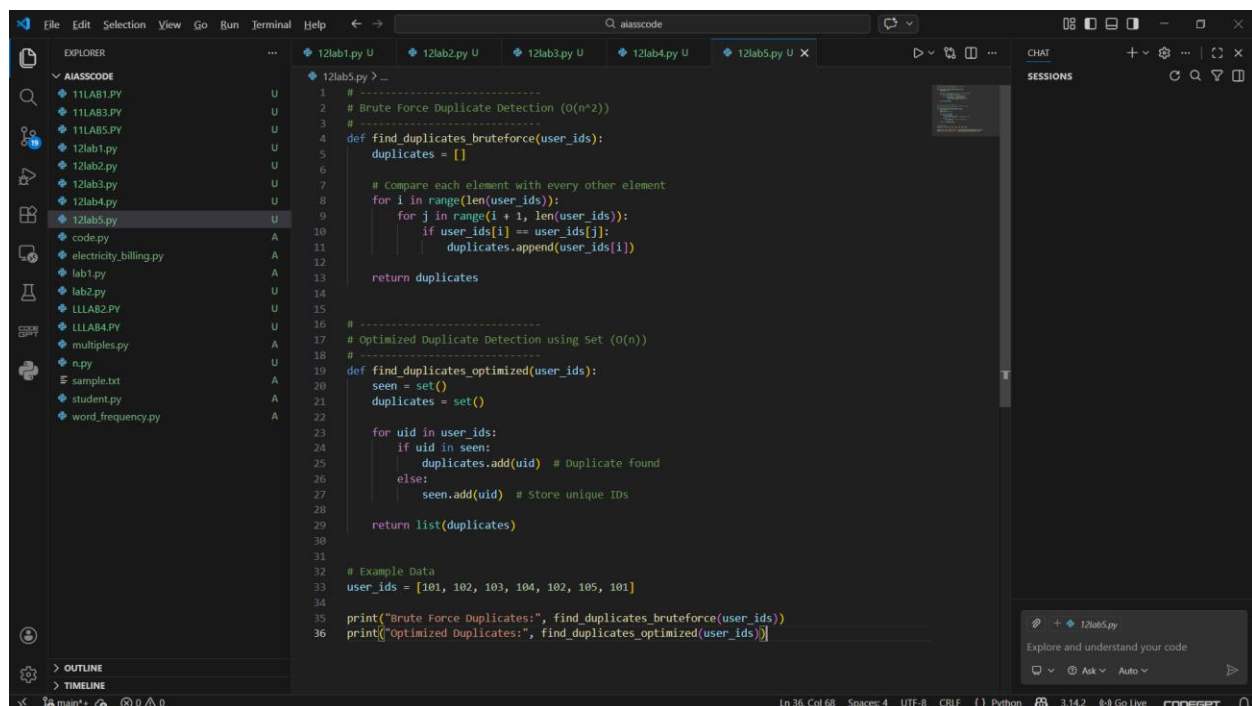
Write a Python program to detect duplicate user IDs in a dataset using a naive nested loop approach.

Analyze the time complexity of this method.

Suggest an optimized approach using sets or dictionaries and rewrite the algorithm with better efficiency.

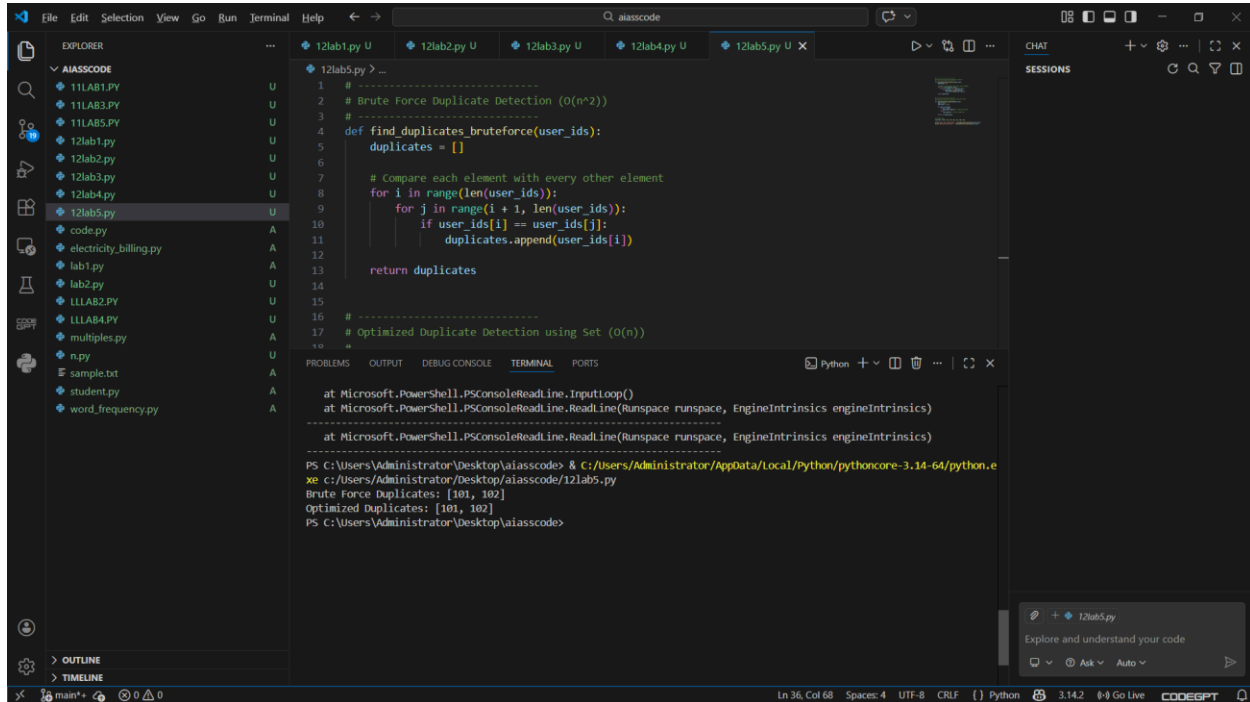
Compare the execution behavior conceptually for large input sizes and explain how performance improves

Code:

The image shows a screenshot of a Visual Studio Code editor window. The Explorer panel on the left shows a file tree with various Python files, including '12lab5.py' which is currently selected. The main editor area displays the code for '12lab5.py'. The code implements two functions: 'find_duplicates_bruteforce' and 'find_duplicates_optimized'. The first function uses a nested loop to compare every element in a list of user IDs with every other element, resulting in O(n^2) time complexity. The second function uses a set to track unique IDs, resulting in O(n) time complexity. The code includes a list of example user IDs: [101, 102, 103, 104, 102, 105, 101]. The status bar at the bottom indicates the file is at line 36, column 68, with 4 spaces, UTF-8 encoding, and CRLF line endings. The editor is running Python 3.14.2 on a Windows machine, as indicated by the status bar and the 'Go Live' button in the bottom right corner.

```
1 # -----  
2 # Brute Force Duplicate Detection (O(n^2))  
3 # -----  
4 def find_duplicates_bruteforce(user_ids):  
5     duplicates = []  
6  
7     # Compare each element with every other element  
8     for i in range(len(user_ids)):  
9         for j in range(i + 1, len(user_ids)):  
10             if user_ids[i] == user_ids[j]:  
11                 duplicates.append(user_ids[i])  
12  
13     return duplicates  
14  
15 # -----  
16 # Optimized Duplicate Detection using Set (O(n))  
17 # -----  
18 def find_duplicates_optimized(user_ids):  
19     seen = set()  
20     duplicates = set()  
21  
22     for uid in user_ids:  
23         if uid in seen:  
24             duplicates.add(uid) # Duplicate found  
25         else:  
26             seen.add(uid) # Store unique IDs  
27  
28     return list(duplicates)  
29  
30  
31  
32 # Example Data  
33 user_ids = [101, 102, 103, 104, 102, 105, 101]  
34  
35 print("Brute Force Duplicates:", find_duplicates_bruteforce(user_ids))  
36 print("Optimized Duplicates:", find_duplicates_optimized(user_ids))
```


Output:



The screenshot shows a VS Code editor with a file explorer on the left containing various Python files. The main editor window displays a Python script named `12lab5.py` with two functions: `find_duplicates_bruteforce` and `find_duplicates_optimized`. The `find_duplicates_bruteforce` function uses nested loops to compare every pair of user IDs, while the `find_duplicates_optimized` function uses a set to check for duplicates in constant time. The terminal at the bottom shows the execution of the script, which outputs the duplicate user IDs for both methods.

```
1 # -----
2 # Brute Force Duplicate Detection (O(n^2))
3 # -----
4 def find_duplicates_bruteforce(user_ids):
5     duplicates = []
6
7     # Compare each element with every other element
8     for i in range(len(user_ids)):
9         for j in range(i + 1, len(user_ids)):
10             if user_ids[i] == user_ids[j]:
11                 duplicates.append(user_ids[i])
12
13     return duplicates
14
15 # -----
16 # Optimized Duplicate Detection using Set (O(n))
17 # -----
18 def find_duplicates_optimized(user_ids):
19     seen = set()
20     duplicates = []
21     for user_id in user_ids:
22         if user_id in seen:
23             duplicates.append(user_id)
24         else:
25             seen.add(user_id)
26     return duplicates
27
28 # Test the functions
29 user_ids = [101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 101, 102]
30
31 brute_duplicates = find_duplicates_bruteforce(user_ids)
32 optimized_duplicates = find_duplicates_optimized(user_ids)
33
34 print("Brute Force Duplicates: ", brute_duplicates)
35 print("Optimized Duplicates: ", optimized_duplicates)
```

```
PS C:\Users\Administrator\Desktop\aiasscode> & C:/Users/Administrator/AppData/Local/Python/pythoncore-3.14-64/python.exe
c:\Users\Administrator\Desktop\aiasscode\12lab5.py
Brute Force Duplicates: [101, 102]
Optimized Duplicates: [101, 102]
PS C:\Users\Administrator\Desktop\aiasscode>
```

Explanation:

The brute-force method uses nested loops, so it compares every pair of user IDs, resulting in $O(n^2)$ time complexity, which becomes very slow for large datasets.

The optimized method uses a `set` to store seen IDs and check duplicates in constant time, reducing the complexity to $O(n)$. This makes it much faster and more efficient for large inputs