Assignment-12.4
Ai Assisted Coding

Htno:2303A51286

Batch no-05

Task-01 : 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.

Prompt:

Write a Python program to implement Bubble Sort for sorting student exam scores.

Add inline comments explaining comparisons, swaps, and each pass through the list.

Include an early termination condition to stop when the list becomes sorted. Show sample input and output demonstrating the sorted scores

Code:

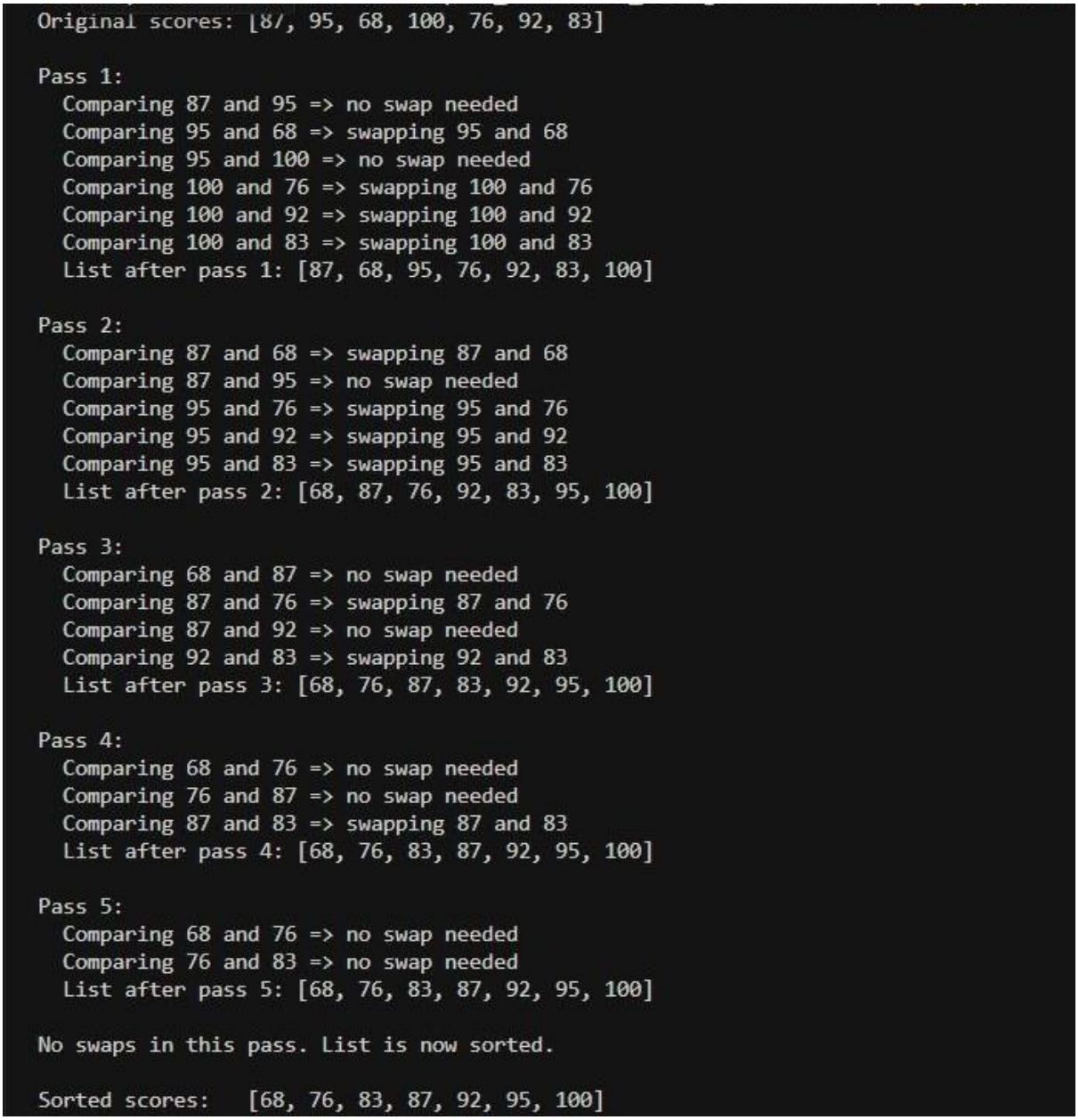
```python
def bubble_sort(scores):
    n = len(scores)
    # Traverse through all elements in the list
    for i in range(n):
        # Flag to check if any swaps happened in this pass
        swapped = False
        print(f"\nPass {i+1}:")
        # Last i elements are already sorted
        for j in range(0, n-i-1):
            # Show the comparison
            print(f"  Comparing {scores[j]} and {scores[j+1]}", end='')
            # Traverse the list from 0 to n-i-1
            # Swap if the element found is greater than the next element
            if scores[j] > scores[j+1]:
                # Explain the swap
                print(f" => swapping {scores[j]} and {scores[j+1]}")
                scores[j], scores[j+1] = scores[j+1], scores[j]  # Perform swap
                swapped = True
            else:
                print(" => no swap needed")
        # Current state of the list after this pass
        print(f"  List after pass {i+1}: {scores}")
        # If no swaps occurred, the list is sorted
        if not swapped:
            print("\nNo swaps in this pass. List is now sorted.")
            break

# Sample input
student_scores = [87, 95, 68, 100, 76, 92, 83]
print("Original scores:", student_scores)

bubble_sort(student_scores)

print("\nSorted scores:  ", student_scores)
```

Output:



Explanation:

The function compares adjacent student scores and swaps them if they are in the wrong order.

Each outer loop pass moves the highest remaining score to its correct position.

A swapped flag tracks whether any exchange happens during a pass.

If no swaps occur, the algorithm stops early because the list is already sorted.

This results in an efficient Bubble Sort with best-case O(n) and worst/average-case O(n^2) time complexity.

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.

Prompt:

Given a nearly sorted list of student roll numbers in an attendance system, start with a Bubble Sort implementation.
Review the problem and suggest a more suitable sorting algorithm for nearly sorted data.
Generate a Python implementation of Bubble sort, Insertion Sort and compare both outputs.

Code:

```python
# Bubble Sort implementation
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr

arr = [64, 34, 25, 12, 22, 11, 90]
print("Original array:", arr)
sorted_arr = bubble_sort(arr)
print("Sorted array by Bubble Sort:", sorted_arr)
#Insertion Sort implementation
def insertion_sort(arr):
    n = len(arr)
    for i in range(1, n):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

arr = [64, 34, 25, 12, 22, 11, 90]
print("Original array:", arr)
sorted_arr = insertion_sort(arr)
print("Sorted array by Insertion Sort:", sorted_arr)
```

Output:

```
sk2.py
Original array: [64, 34, 25, 12, 22, 11, 90]
Sorted array by Bubble Sort: [11, 12, 22, 25, 34, 64, 90]
Original array: [64, 34, 25, 12, 22, 11, 90]
Sorted array by Insertion Sort: [11, 12, 22, 25, 34, 64, 90]
```

Explanation:

Bubble Sort works by repeatedly comparing adjacent elements and swapping them if they are out of order, which causes many unnecessary comparisons even when the list is almost sorted. In your code, every pass still scans most of the array, so its time complexity remains $O(n^2)$ regardless of how sorted the input already is.

Insertion Sort, on the other hand, builds the sorted list gradually by inserting each element into its correct position among the previously sorted elements. When the input is nearly sorted, very few shifts are needed, so the algorithm runs much faster. In such cases, Insertion Sort approaches O(n) time complexity, making it more efficient than Bubble Sort.

Insertion Sort is the better choice

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.

Prompt:

Implement Linear Search in Python to search for a student record by roll number in an unsorted list.

Implement Binary Search in Python to search for a student record by roll number in a sorted list.

Add docstrings to both functions explaining parameters and return values.

Code:

```python
# ------------------ Linear Search Implementation ------------------
def linear_search(records, roll_number):
    """
    Perform a linear search to find a student record by roll number.

    Parameters:
    records (list): A list of student records, where each record is a dictionary with 'roll_number' and 'name'.
    roll_number (int): The roll number to search for.

    Returns:
    dict: The student record if found, otherwise None.
    """
    for record in records:
        if record['roll_number'] == roll_number:
            return record
    return None
# ------------------ Binary Search Implementation ------------------
def binary_search(records, roll_number):
    """
    Perform a binary search to find a student record by roll number.

    Parameters:
    records (list): A sorted list of student records, where each record is a dictionary with 'roll_number' and 'name'.
    roll_number (int): The roll number to search for.

    Returns:
    dict: The student record if found, otherwise None.
    """
    left, right = 0, len(records) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if records[mid]['roll_number'] == roll_number:
            return records[mid]
```

```python
def binary_search(records, roll_number):
    left, right = 0, len(records) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if records[mid]['roll_number'] == roll_number:
            return records[mid]
        elif records[mid]['roll_number'] < roll_number:
            left = mid + 1
        else:
            right = mid - 1
    return None
# ------------------ Main Program ------------------
def main():
    # Sample student records
    student_records = [
        {'roll_number': 101, 'name': 'Alice'},
        {'roll_number': 102, 'name': 'Bob'},
        {'roll_number': 103, 'name': 'Charlie'},
        {'roll_number': 104, 'name': 'David'},
        {'roll_number': 105, 'name': 'Eve'}
    ]

    # Unsorted list for linear search
    print("Linear Search:")
    roll_number_to_search = int(input("Enter roll number to search (linear): "))
    result = linear_search(student_records, roll_number_to_search)
    if result:
        print(f"Record found: {result}")
    else:
        print("Record not found.")

    # Sorted list for binary search
    sorted_records = sorted(student_records, key=lambda x: x['roll_number'])
    print("\nBinary Search:")
    roll_number_to_search = int(input("Enter roll number to search (binary): "))
    result = binary_search(sorted_records, roll_number_to_search)
    if result:
        print(f"Record found: {result}")
    else:
        print("Record not found.")
if __name__ == "__main__":
    main()
```

Output:

```
Linear Search:
Enter roll number to search (linear): 101
Record found: {'roll_number': 101, 'name': 'Alice'}

Binary Search:
Enter roll number to search (binary): 105
Record found: {'roll_number': 105, 'name': 'Eve'}
```

Explanation: The program searches student records by roll number using Linear Search and Binary Search. Linear Search scans each record one by one and works on unsorted data with O(n) time complexity.

Binary Search operates on sorted records and repeatedly halves the search space. Because of this, Binary Search is much faster with O(log n) time complexity.

Binary Search cannot be used on unsorted lists, while Linear Search can. For large, sorted student databases, Binary Search is the more efficient choice

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).

Prompt:

Give partially written recursive Python functions for Quick Sort and Merge Sort to sort large datasets.

Complete the recursive logic for both algorithms and add clear docstrings explaining parameters and return values.

Explain how recursion works in Quick Sort and Merge Sort in simple terms.

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

Code:

```python
# ------------------ Quick Sort Implementation ------------------
def quick_sort(arr):
    """
    Sort an array using the Quick Sort algorithm.

    Parameters:
    arr (list): A list of elements to be sorted.

    Returns:
    list: A new sorted list containing the elements from the input array.
    """
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
# ------------------ Merge Sort Implementation ------------------
def merge_sort(arr):
    """
    Sort an array using the Merge Sort algorithm.

    Parameters:
    arr (list): A list of elements to be sorted.

    Returns:
    list: A new sorted list containing the elements from the input array.
    """
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
```

```
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])
    return merge(left_half, right_half)
def merge(left, right):
    """
    Merge two sorted lists into a single sorted list.
    Parameters:
    left (list): A sorted list of elements.
    right (list): Another sorted list of elements.
    Returns:
    list: A new 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]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
# ----------------- Main Program -----------------
def main():
    import random

    # Test datasets
    random_dataset = [random.randint(1, 100) for _ in range(10)]
    sorted_dataset = sorted(random_dataset)
    reverse_sorted_dataset = sorted(random_dataset, reverse=True)

    print("Random Dataset:", random_dataset)
    print("Sorted Dataset:", sorted_dataset)
    print("Reverse Sorted Dataset:", reverse_sorted_dataset)

    # Testing Quick Sort
    print("\nTesting Quick Sort:")
    print("Sorted Random Dataset:", quick_sort(random_dataset))
    print("Sorted Already Sorted Dataset:", quick_sort(sorted_dataset))
    print("Sorted Reverse Sorted Dataset:", quick_sort(reverse_sorted_dataset))

    # Testing Merge Sort
```

```
    return result
# ------------------ Main Program ------------------
def main():
    import random

    # Test datasets
    random_dataset = [random.randint(1, 100) for _ in range(10)]
    sorted_dataset = sorted(random_dataset)
    reverse_sorted_dataset = sorted(random_dataset, reverse=True)

    print("Random Dataset:", random_dataset)
    print("Sorted Dataset:", sorted_dataset)
    print("Reverse Sorted Dataset:", reverse_sorted_dataset)

    # Testing Quick Sort
    print("\nTesting Quick Sort:")
    print("Sorted Random Dataset:", quick_sort(random_dataset))
    print("Sorted Already Sorted Dataset:", quick_sort(sorted_dataset))
    print("Sorted Reverse Sorted Dataset:", quick_sort(reverse_sorted_dataset))

    # Testing Merge Sort
    print("\nTesting Merge Sort:")
    print("Sorted Random Dataset:", merge_sort(random_dataset))
    print("Sorted Already Sorted Dataset:", merge_sort(sorted_dataset))
    print("Sorted Reverse Sorted Dataset:", merge_sort(reverse_sorted_dataset))
if __name__ == "__main__":
    main()
```

Output:

```
Random Dataset: [9, 9, 46, 23, 96, 60, 1, 25, 76, 52]
Sorted Dataset: [1, 9, 9, 23, 25, 46, 52, 60, 76, 96]
Reverse Sorted Dataset: [96, 76, 60, 52, 46, 25, 23, 9, 9, 1]

Testing Quick Sort:
Sorted Random Dataset: [1, 9, 9, 23, 25, 46, 52, 60, 76, 96]
Sorted Already Sorted Dataset: [1, 9, 9, 23, 25, 46, 52, 60, 76, 96]
Sorted Reverse Sorted Dataset: [1, 9, 9, 23, 25, 46, 52, 60, 76, 96]

Testing Merge Sort:
Sorted Random Dataset: [1, 9, 9, 23, 25, 46, 52, 60, 76, 96]
Sorted Already Sorted Dataset: [1, 9, 9, 23, 25, 46, 52, 60, 76, 96]
Sorted Reverse Sorted Dataset: [1, 9, 9, 23, 25, 46, 52, 60, 76, 96]
```

Explanation:

Quick Sort uses recursion by selecting a pivot, dividing the list into smaller sublists, and recursively sorting them until single-element lists remain. Merge Sort uses recursion by repeatedly splitting the list into halves and then merging the sorted halves back together.

Quick Sort is fast on average but can degrade to $O(n^2)$ on already sorted or reverse-sorted data.

Merge Sort consistently runs in O(n log n) time for all input orders.

Therefore, Merge Sort is preferred for predictable performance on large datasets, while Quick Sort is efficient for random data.

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.

Prompt:

Write a naive duplicate detection algorithm in Python using nested loops to identify duplicate user IDs in a dataset.

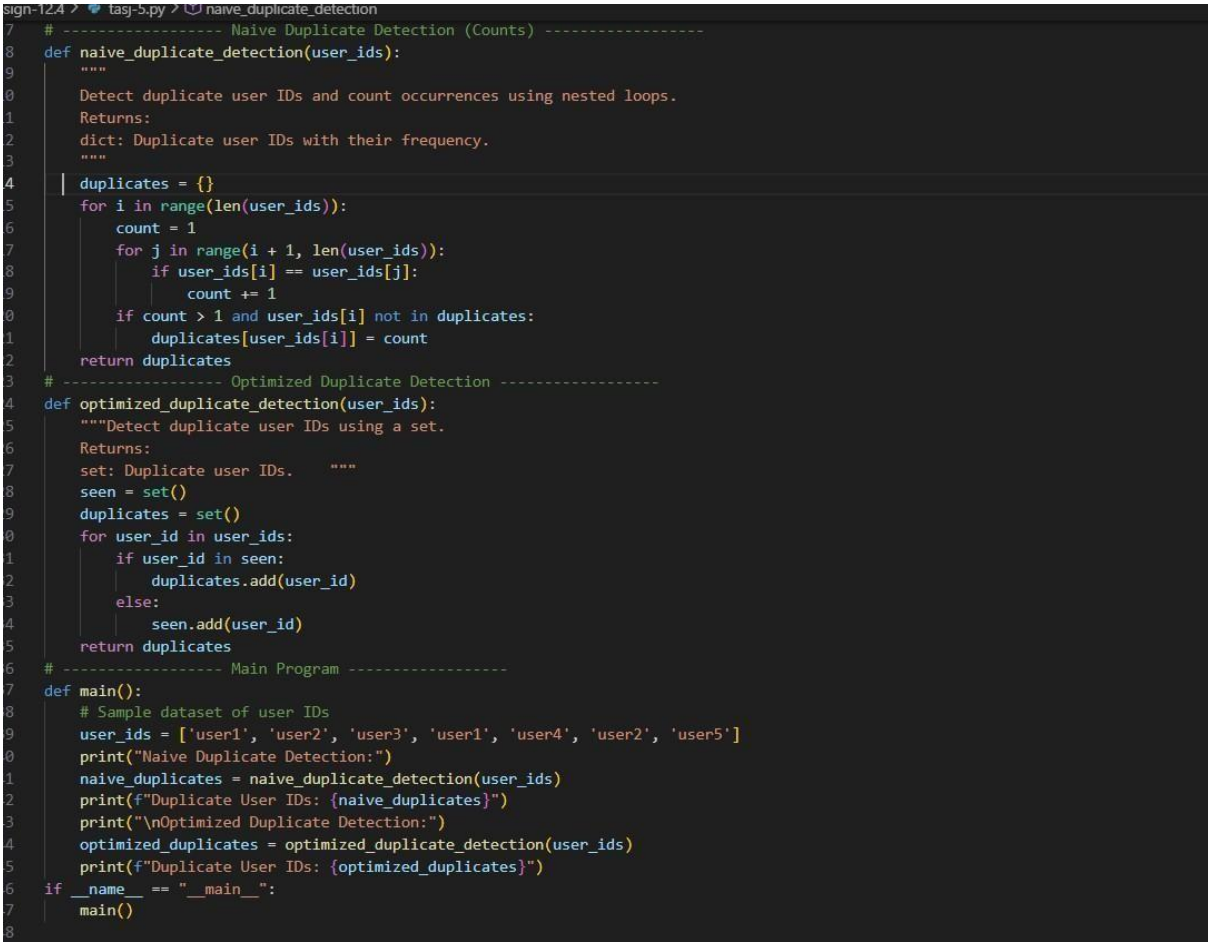
Analyze and explain the time complexity of this brute-force approach.

Suggest a more optimized method using sets or dictionaries for duplicate detection.

Rewrite the algorithm using the optimized approach and explain how it improves efficiency.

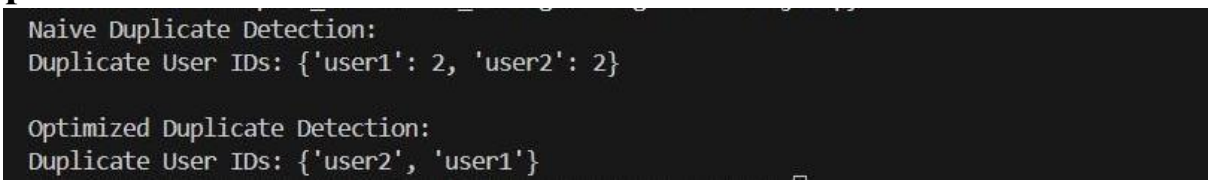
Compare the execution behavior conceptually for large input sizes,

Code:



```python
# ------------------ Naive Duplicate Detection (Counts) ------------------
def naive_duplicate_detection(user_ids):
    """
    Detect duplicate user IDs and count occurrences using nested loops.
    Returns:
    dict: Duplicate user IDs with their frequency.
    """
    duplicates = {}
    for i in range(len(user_ids)):
        count = 1
        for j in range(i + 1, len(user_ids)):
            if user_ids[i] == user_ids[j]:
                count += 1
        if count > 1 and user_ids[i] not in duplicates:
            duplicates[user_ids[i]] = count
    return duplicates
# ------------------ Optimized Duplicate Detection ------------------
def optimized_duplicate_detection(user_ids):
    """Detect duplicate user IDs using a set.
    Returns:
    set: Duplicate user IDs.    """
    seen = set()
    duplicates = set()
    for user_id in user_ids:
        if user_id in seen:
            duplicates.add(user_id)
        else:
            seen.add(user_id)
    return duplicates
# ------------------ Main Program ------------------
def main():
    # Sample dataset of user IDs
    user_ids = ['user1', 'user2', 'user3', 'user1', 'user4', 'user2', 'user5']
    print("Naive Duplicate Detection:")
    naive_duplicates = naive_duplicate_detection(user_ids)
    print(f"Duplicate User IDs: {naive_duplicates}")
    print("\nOptimized Duplicate Detection:")
    optimized_duplicates = optimized_duplicate_detection(user_ids)
    print(f"Duplicate User IDs: {optimized_duplicates}")
if __name__ == "__main__":
    main()
```

Output:



```
Naive Duplicate Detection:
Duplicate User IDs: {'user1': 2, 'user2': 2}

Optimized Duplicate Detection:
Duplicate User IDs: {'user2', 'user1'}
```

Explanation:

The naive algorithm uses nested loops to compare every user ID with others and counts duplicate occurrences, resulting in $O(n^2)$ time complexity. It returns a dictionary showing how many times each duplicate user ID appears.

The optimized algorithm uses a set to track seen IDs and detects duplicates in a single pass. This optimized approach runs in O(n) time due to constant-time set lookups.

The difference in output shows that optimization improves efficiency while also changing how results are reported.