Assignment-12.4

Ht.No: 2303A51584

Batch: 05

Task 1: Bubble Sort for Ranking Exam Scores

Scenario

You are working on a college result processing system where a small

list of student scores needs to be sorted after every internal assessment.

Task Description

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

Prompt:

Implement Bubble Sort to sort a small list of student scores after each internal assessment. The program should take a list of scores and sort them in ascending order using the Bubble Sort algorithm. Insert clear inline comments explaining important steps such as comparisons between elements, swapping values, and iteration passes. Also include an early-termination condition so that the algorithm stops if the list becomes sorted. Provide a brief time complexity analysis explaining the best-case and worst-case scenarios.

Code:

```
"""
Bubble Sort Algorithm for Student Scores
This program implements the Bubble Sort algorithm to sort student scores in ascending order.
It includes an optimization that terminates early if no swaps occur during a pass.
"""

def bubble_sort_scores(scores):
    """
    Sorts a list of student scores in ascending order using the Bubble Sort algorithm.

    Args:
        scores: A list of numeric scores to be sorted

    Returns:
        The sorted list of scores in ascending order
    """

    # Get the length of the scores list
    n = len(scores)

    # Outer loop: represents each pass through the list
    for i in range(n):
        # Flag to track if any swap occurred during this pass
        # Used for early termination optimization
        swapped = False

        # Inner loop: compares adjacent elements
        # We reduce the range each iteration because the largest elements
        # "bubble up" to the end after each pass
        # (n - i - 1) ensures we don't compare already-sorted elements at the end
        for j in range(0, n - i - 1):
            # Compare adjacent elements
            # If current element is greater than next element, they are out of order
            if scores[j] > scores[j + 1]:
                # Swap the two elements
                # Temporary variable stores the first element
                temp = scores[j]
                # Move the second element to the first position
                scores[j] = scores[j + 1]
                # Move the stored first element to the second position
                scores[j + 1] = temp

                # Mark that a swap occurred during this pass
                swapped = True
```

```
        # the list is already sorted, so we can exit early
        if not swapped:
            print(f"✓ List is sorted! Early termination at pass {i + 1}\n")
            break

    return scores


def display_sorting_process(scores):
    """
    Displays the step-by-step sorting process of student scores.

    Args:
        scores: A list of numeric scores
    """
    print("=" * 60)
    print("BUBBLE SORT: STUDENT SCORES INTERNAL ASSESSMENT")
    print("=" * 60)
    print(f"Original scores: {scores}\n")

    # Call bubble sort and display the process
    sorted_scores = bubble_sort_scores(scores)

    print("=" * 60)
    print(f"Final sorted scores: {sorted_scores}")
    print("=" * 60)


# Main program execution
if __name__ == "__main__":
    # Example 1: Unsorted student scores from an internal assessment
    print("\nExample 1: Standard Case - Unsorted Scores\n")
    student_scores_1 = [45, 78, 23, 89, 56, 34, 92, 67]
    display_sorting_process(student_scores_1.copy())

    # Example 2: Already sorted scores (demonstrates early termination)
    print("\n\nExample 2: Already Sorted Scores (Shows Early Termination)\n")
    student_scores_2 = [23, 34, 45, 56, 67, 78, 89, 92]
    display_sorting_process(student_scores_2.copy())

    # Example 3: Reverse sorted scores (worst case scenario)
    print("\n\nExample 3: Reverse Sorted Scores (Worst Case)\n")
    student_scores_3 = [92, 89, 78, 67, 56, 45, 34, 23]
    display_sorting_process(student_scores_3.copy())
```

Output:

```
============================================================
BUBBLE SORT: STUDENT SCORES INTERNAL ASSESSMENT
============================================================
Original scores: [45, 78, 23, 89, 56, 34, 92, 67]

  Swapped: 23 and 78 → List is now: [45, 23, 78, 89, 56, 34, 92, 67]
  Swapped: 56 and 89 → List is now: [45, 23, 78, 56, 89, 34, 92, 67]
  Swapped: 34 and 89 → List is now: [45, 23, 78, 56, 34, 89, 92, 67]
  Swapped: 67 and 92 → List is now: [45, 23, 78, 56, 34, 89, 67, 92]
Pass 1 complete: [45, 23, 78, 56, 34, 89, 67, 92]

  Swapped: 23 and 45 → List is now: [23, 45, 78, 56, 34, 89, 67, 92]
  Swapped: 56 and 78 → List is now: [23, 45, 56, 78, 34, 89, 67, 92]
  Swapped: 34 and 78 → List is now: [23, 45, 56, 34, 78, 89, 67, 92]
  Swapped: 67 and 89 → List is now: [23, 45, 56, 34, 78, 67, 89, 92]
Pass 2 complete: [23, 45, 56, 34, 78, 67, 89, 92]

  Swapped: 34 and 56 → List is now: [23, 45, 34, 56, 78, 67, 89, 92]
  Swapped: 67 and 78 → List is now: [23, 45, 34, 56, 67, 78, 89, 92]
Pass 3 complete: [23, 45, 34, 56, 67, 78, 89, 92]

  Swapped: 34 and 45 → List is now: [23, 34, 45, 56, 67, 78, 89, 92]
Pass 4 complete: [23, 34, 45, 56, 67, 78, 89, 92]

Pass 5 complete: [23, 34, 45, 56, 67, 78, 89, 92]

✓ List is sorted! Early termination at pass 5
```

Explanation:

Time Complexity of Bubble Sort

- **Worst Case – $O(n^2)$:**
If the list is in reverse order, the algorithm has to compare and swap many times. It makes multiple passes through the list, so the total operations become roughly $n \times n$.

- **Average Case – $O(n^2)$:**
For a normal unsorted list, it still performs many comparisons and swaps, so the time complexity remains n^2.

• **Best Case – O(n):**
If the list is already sorted and we use early termination (stop when no swaps happen), the algorithm finishes in just one pass. In this case, it takes linear time.

So, Bubble Sort is fine for small lists but not efficient for large datasets.

Task 2: Improving Sorting for Nearly Sorted

Attendance Records

Scenario

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

Task Description

• Start with a Bubble Sort implementation.

• Ask AI to:

• Review the problem and suggest a more suitable sorting algorithm

• Generate an Insertion Sort implementation

• Explain why Insertion Sort performs better on nearly sorted data

• Compare execution behavior on nearly sorted input

Prompt:

Sort an attendance system where student roll numbers are already almost sorted, with only a few late updates. Modify this bubble sort implementation to insertion sort.

Code:

Bubble sort:

```python
def bubble_sort(scores):
    """Sort student scores in ascending order using Bubble Sort."""
    n = len(scores)
    for i in range(n):
        swapped = False

        # Compare adjacent elements
        for j in range(0, n - i - 1):
            if scores[j] > scores[j + 1]:
                # Swap elements
                scores[j], scores[j + 1] = scores[j + 1], scores[j]
                swapped = True

        # Early termination: exit if no swaps occurred
        if not swapped:
            break

    return scores


if __name__ == "__main__":
    # Student scores to sort
    scores = [45, 78, 89, 92, 100, 20, 46]
    print(f"Original: {scores}")
    print(f"Sorted:   {bubble_sort(scores)}")
```

Output:

```
E:\3-2\AI Assisted Coding>C:/Users/hp/AppData/Local/Programs/Python/Python311/python.exe "e:/3-2/A
Original: [45, 78, 23, 89, 56, 34, 92, 67]
Sorted:   [23, 34, 45, 56, 67, 78, 89, 92]

E:\3-2\AI Assisted Coding>
```

Insertion sort:

```python
def insertion_sort(roll_numbers):
    """Sort student roll numbers using Insertion Sort (optimal for nearly sorted lists)."""
    n = len(roll_numbers)

    # Start from second element (first element is already sorted)
    for i in range(1, n):
        key = roll_numbers[i]  # Current element to be inserted
        j = i - 1

        # Shift elements greater than key one position right
        while j >= 0 and roll_numbers[j] > key:
            roll_numbers[j + 1] = roll_numbers[j]
            j -= 1

        # Insert key at correct position
        roll_numbers[j + 1] = key

    return roll_numbers


if __name__ == "__main__":
    # Student roll numbers (nearly sorted with late updates)
    roll_numbers = [101, 105, 103, 108, 110, 104, 102, 107]
    print(f"Original: {roll_numbers}")
    print(f"Sorted:   {insertion_sort(roll_numbers)}")
```

Output:

```
E:\3-2\AI Assisted Coding>C:/Users/hp/AppData/Local/Programs/Pytho
Original: [101, 105, 103, 108, 110, 104, 102, 107]
Sorted:   [101, 102, 103, 104, 105, 107, 108, 110]
```

Explanation:

Comparison on Nearly Sorted Data

Feature	Insertion Sort	Bubble Sort
Working Method	Places each element in its correct position	Repeatedly compares adjacent elements
Movement of Elements	Moves only misplaced elements	Rechecks many elements repeatedly
Comparisons	Stops early when position is correct	Makes repeated passes over the list
Performance (Nearly Sorted)	Very fast (close to O(n))	Slower compared to Insertion Sort
Efficiency	More efficient	Less efficient

Conclusion

For nearly sorted attendance records, **Insertion Sort performs better** because it only adjusts the few misplaced elements, while Bubble Sort keeps scanning the list multiple times.

Task 3: Searching Student Records in a Database

Scenario

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

Task Description

• Implement:

• Linear Search for unsorted student data

• Binary Search for sorted student data

Use AI to:

• Add docstrings explaining parameters and return values

• Explain when Binary Search is applicable

• Highlight performance differences between the two searches

Prompt:

Develop student record search system where users can search for a student by roll number. First, implement Linear Search to work on unsorted student data. Then, implement Binary Search for sorted student data. Add proper docstrings explaining the parameters and return values for each function.

Code:

```python
"""Student Record Search System - Linear and Binary Search Implementation"""


def linear_search(students, target_roll):
    """
    Search for a student by roll number using Linear Search (works on unsorted data).

    Parameters:
        students (list): List of student dictionaries containing 'roll' and 'name' keys
        target_roll (int): The roll number to search for

    Returns:
        dict: Student record if found, None otherwise
    """
    # Iterate through each student in the list
    for i, student in enumerate(students):
        # Check if current student's roll number matches target
        if student['roll'] == target_roll:
            print(f"  Found at index {i}")
            return student

    # Return None if student not found
    return None


def binary_search(students, target_roll):
    """
    Search for a student by roll number using Binary Search (works on sorted data).

    Parameters:
        students (list): Sorted list of student dictionaries by roll number
        target_roll (int): The roll number to search for

    Returns:
        dict: Student record if found, None otherwise
    """
    left, right = 0, len(students) - 1

    # Keep narrowing the search space
    while left <= right:
        mid = (left + right) // 2
```

```python
    {'roll': 105, 'name': 'Alice Johnson'},
    {'roll': 102, 'name': 'Bob Smith'},
    {'roll': 108, 'name': 'Charlie Brown'},
    {'roll': 101, 'name': 'Diana Prince'},
    {'roll': 107, 'name': 'Eve Wilson'},
    {'roll': 103, 'name': 'Frank Castle'},
    {'roll': 106, 'name': 'Grace Lee'},
    {'roll': 104, 'name': 'Henry Davis'}
]

# Sorted student records
sorted_students = sorted(unsorted_students, key=lambda x: x['roll'])

print("=" * 60)
print("STUDENT RECORD SEARCH SYSTEM")
print("=" * 60)

# Linear Search Examples
print("\n--- LINEAR SEARCH (Unsorted Data) ---\n")
test_rolls = [105, 103, 110]

for roll in test_rolls:
    print(f"Searching for roll number {roll}:")
    result = linear_search(unsorted_students, roll)
    if result:
        print(f"  Result: {result}\n")
    else:
        print(f"  Result: Not found\n")

# Binary Search Examples
print("--- BINARY SEARCH (Sorted Data) ---\n")

for roll in test_rolls:
    print(f"Searching for roll number {roll}:")
    result = binary_search(sorted_students, roll)
    if result:
        print(f"  Result: {result}\n")
    else:
        print(f"  Result: Not found\n")

print("=" * 60)
```

Output:

```
============================================================

E:\3-2\AI Assisted Coding>C:/Users/hp/AppData/Local/Programs/Python/Python311/python.exe
============================================================
STUDENT RECORD SEARCH SYSTEM
============================================================

--- LINEAR SEARCH (Unsorted Data) ---

Searching for roll number 105:
  Found at index 0
  Result: {'roll': 105, 'name': 'Alice Johnson'}

Searching for roll number 103:
  Found at index 5
  Result: {'roll': 103, 'name': 'Frank Castle'}

Searching for roll number 110:
  Result: Not found

--- BINARY SEARCH (Sorted Data) ---

Searching for roll number 105:
  Found at index 4
  Result: {'roll': 105, 'name': 'Alice Johnson'}

Searching for roll number 103:
  Found at index 2
  Result: {'roll': 103, 'name': 'Frank Castle'}

Searching for roll number 110:
  Result: Not found
```

Explanation:

When is Binary Search Applicable?

Binary Search can be used **only when the data is sorted** (in ascending or descending order).

• The list must be arranged in order.
• It works by repeatedly dividing the search space into halves.
• If the data is unsorted, Binary Search will not give correct results.

So, before applying Binary Search, we must ensure the student roll numbers are sorted.

Performance Difference: Linear Search vs Binary Search

Feature	Linear Search	Binary Search
Data Requirement	Works on unsorted data	Requires sorted data
Searching Method	Checks elements one by one	Divides list into halves
Time Complexity	O(n)	O(log n)
Speed (Large Data)	Slower	Much faster

• **Linear Search** is simple and works on any data.
• **Binary Search** is much faster but only works on sorted data.

For large student databases, Binary Search is more efficient if the data is sorted.

Task 4: Choosing Between Quick Sort and

Merge Sort for Data Processing

Scenario

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

Task Description

• Provide AI with partially written recursive functions for:

• Quick Sort

• Merge Sort

• Ask AI to:

• Complete the recursive logic

• Add meaningful docstrings

• Explain how recursion works in each algorithm

• Test both algorithms on:

    • Random data

    • Sorted data

    •Reverse-sorted data

Prompt:

Provide partially written recursive functions for Quick Sort and Merge Sort for sorting large datasets. Then complete the recursive logic for both algorithms and add meaningful docstrings explaining the parameters and return values. Also include a simple explanation of how recursion works in each algorithm. Finally, test both algorithms on three types of input data: random data, already sorted data, and reverse-sorted data, and compare their behavior. Generate everything in a single file with clear comments.

Code:

```
"""
Recursive Sorting Algorithms: Quick Sort and Merge Sort
This module implements two popular divide-and-conquer sorting algorithms with recursion.

Quick Sort: Recursively partitions data around a pivot element
Merge Sort: Recursively divides data in half, then merges sorted halves
"""



# ============================================================================
# QUICK SORT IMPLEMENTATION
# ============================================================================

def quick_sort(data, low=0, high=None):
    """
    Sort data using Quick Sort algorithm (divide-and-conquer with recursion).

    How Recursion Works:
    1. Choose a pivot element (last element in current range)
    2. Partition: move elements smaller than pivot to left, larger to right
    3. Recursively call quick_sort on left partition (low to pivot-1)
    4. Recursively call quick_sort on right partition (pivot+1 to high)
    5. Base case: when low >= high, the subarray is sorted (return)

    Parameters:
        data (list): List of comparable elements to sort
        low (int): Starting index of current partition (default: 0)
        high (int): Ending index of current partition (default: len(data)-1)

    Returns:
        list: The sorted data list (sorted in-place)
    """
    # Initialize high on first call
    if high is None:
        high = len(data) - 1

    # Base case: if partition has 0 or 1 element, it's already sorted
    if low < high:
        # Partition the data and get pivot index
        pivot_index = partition(data, low, high)

        # Recursively sort left partition (elements smaller than pivot)
        quick_sort(data, low, pivot_index - 1)

        # Recursively sort right partition (elements larger than pivot)
```

```python
def quick_sort(data, low=0, high=None):
        # Recursively sort right partition (elements larger than pivot)
        quick_sort(data, pivot_index + 1, high)

    return data


def partition(data, low, high):
    """
    Partition helper function for Quick Sort.

    Parameters:
        data (list): List being partitioned
        low (int): Start index of partition
        high (int): End index of partition

    Returns:
        int: Final pivot index after partitioning
    """
    # Choose last element as pivot
    pivot = data[high]

    # Index for smaller elements
    i = low - 1

    # Compare each element with pivot
    for j in range(low, high):
        if data[j] < pivot:
            # Move smaller element to left side
            i += 1
            data[i], data[j] = data[j], data[i]

    # Place pivot in correct position
    data[i + 1], data[high] = data[high], data[i + 1]
    return i + 1


# ============================================================================
# MERGE SORT IMPLEMENTATION
# ============================================================================

def merge_sort(data):
    """
    Sort data using Merge Sort algorithm (divide-and-conquer with recursion).
```

```python
def merge_sort(data):
    """
    Sort data using Merge Sort algorithm (divide-and-conquer with recursion).

    How Recursion Works:
    1. Divide: Split the list in half recursively until single elements remain
    2. Base case: when list has 0 or 1 element, it's already sorted (return)
    3. Conquer: Recursively call merge_sort on left half
    4. Conquer: Recursively call merge_sort on right half
    5. Combine: Merge the two sorted halves into one sorted list

    Parameters:
        data (list): List of comparable elements to sort

    Returns:
        list: A new sorted list
    """
    # Base case: lists with 0 or 1 element are already sorted
    if len(data) <= 1:
        return data

    # Divide: split list in half
    mid = len(data) // 2
    left = data[:mid]
    right = data[mid:]

    # Conquer: recursively sort both halves
    left_sorted = merge_sort(left)
    right_sorted = merge_sort(right)

    # Combine: merge the sorted halves
    return merge(left_sorted, right_sorted)


def merge(left, right):
    """
    Merge helper function for Merge Sort.
    Combines two sorted lists into one sorted list.

    Parameters:
        left (list): First sorted list
        right (list): Second sorted list

    Returns:
        list: Merged and sorted list
```

```python
def merge(left, right):
    Returns:
        list: Merged and sorted list
    """
    result = []
    i = j = 0

    # Compare elements from left and right, add smaller to result
    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 from left
    result.extend(left[i:])

    # Add remaining elements from right
    result.extend(right[j:])

    return result


# ============================================================================
# TESTING AND COMPARISON
# ============================================================================

def test_sorting_algorithms():
    """Test both sorting algorithms on different types of input data."""

    print("=" * 80)
    print("RECURSIVE SORTING ALGORITHMS COMPARISON")
    print("=" * 80)

    # Test datasets
    test_cases = {
        "Random Data": [64, 34, 25, 12, 22, 11, 90, 88, 45, 50, 32],
        "Already Sorted": [11, 12, 22, 25, 32, 34, 45, 50, 64, 88, 90],
        "Reverse Sorted": [90, 88, 64, 50, 45, 34, 32, 25, 22, 12, 11]
    }

    for test_name, data in test_cases.items():
        print(f"\n{'-' * 80}")
```

```
def test_sorting_algorithms():
        print(f"{'-' * 80}")
        print(f"Original data: {data}")

        # Quick Sort
        quick_data = data.copy()
        quick_result = quick_sort(quick_data)
        print(f"\nQuick Sort result:  {quick_result}")

        # Merge Sort
        merge_data = data.copy()
        merge_result = merge_sort(merge_data)
        print(f"Merge Sort result:  {merge_result}")

        # Verify both produce same result
        if quick_result == merge_result:
            print("\n✓ Both algorithms produced identical sorted results")
        else:
            print("\n✗ Results differ (ERROR)")

    print(f"\n{'=' * 80}")
    print("ALGORITHM CHARACTERISTICS")
    print(f"{'=' * 80}")
    print("""
Quick Sort:
  • Best Case: O(n log n) - when pivot divides data evenly
  • Average Case: O(n log n)
  • Worst Case: O(n²) - when pivot is always smallest/largest
  • Space: O(log n) - recursive call stack
  • In-place: Yes (modifies original list)
  • Preferred for: Most random data, limited memory

Merge Sort:
  • Best Case: O(n log n)
  • Average Case: O(n log n)
  • Worst Case: O(n log n) - guaranteed
  • Space: O(n) - requires temporary lists for merging
  • In-place: No (creates new lists)
  • Preferred for: Guaranteed performance, linked lists, stable sort needed
    """)


if __name__ == "__main__":
    test_sorting_algorithms()
```

Output:

```
--------------------------------------------------------------------------------
Test Case: Random Data
--------------------------------------------------------------------------------
Original data: [64, 34, 25, 12, 22, 11, 90, 88, 45, 50, 32]

Quick Sort result:  [11, 12, 22, 25, 32, 34, 45, 50, 64, 88, 90]
Merge Sort result:  [11, 12, 22, 25, 32, 34, 45, 50, 64, 88, 90]

✓ Both algorithms produced identical sorted results

--------------------------------------------------------------------------------
Test Case: Already Sorted
--------------------------------------------------------------------------------
Original data: [11, 12, 22, 25, 32, 34, 45, 50, 64, 88, 90]

Quick Sort result:  [11, 12, 22, 25, 32, 34, 45, 50, 64, 88, 90]
Merge Sort result:  [11, 12, 22, 25, 32, 34, 45, 50, 64, 88, 90]

✓ Both algorithms produced identical sorted results

--------------------------------------------------------------------------------
Test Case: Reverse Sorted
--------------------------------------------------------------------------------
Original data: [90, 88, 64, 50, 45, 34, 32, 25, 22, 12, 11]

Quick Sort result:  [11, 12, 22, 25, 32, 34, 45, 50, 64, 88, 90]
Merge Sort result:  [11, 12, 22, 25, 32, 34, 45, 50, 64, 88, 90]

✓ Both algorithms produced identical sorted results

================================================================================
ALGORITHM CHARACTERISTICS
================================================================================

Quick Sort:
  • Best Case: O(n log n) - when pivot divides data evenly
  • Average Case: O(n log n)
  • Worst Case: O(n²) - when pivot is always smallest/largest
  • Space: O(log n) - recursive call stack
  • In-place: Yes (modifies original list)
  • Preferred for: Most random data, limited memory

Merge Sort:
  • Best Case: O(n log n)
  • Average Case: O(n log n)
```

Explanation:

Quick Sort (Recursion)

- Select a pivot element from the list.
- Divide the list into elements smaller and greater than the pivot.

• Recursively apply Quick Sort to both parts.
• Stop when the sublist has one or no elements (base case).

Merge Sort (Recursion)

• Divide the list into two halves.
• Recursively keep dividing until each part has one element.
• Merge the smaller sorted parts step by step.
• Continue merging until the full list becomes sorted.

Task 5: Optimizing a Duplicate Detection

Algorithm

Scenario

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

Task Description

• Write a naive duplicate detection algorithm using nested loops.

• Use AI to:

    • Analyze the time complexity

    • Suggest an optimized approach using sets or dictionaries

    • Rewrite the algorithm with improved efficiency

    • Compare execution behavior conceptually for large input sizes

Prompt:

Write a Python program to detect duplicate user IDs using a nested loop (brute-force) method. Analyze the time complexity and then suggest a faster approach using a set or dictionary. Rewrite the program using the optimized method and show both versions of the code for comparison.

Code:

```python
def find_duplicates_brute_force(user_ids):
    """
    Find duplicate user IDs using nested loop (brute-force) method.

    Algorithm:
    1. Iterate through each user ID (outer loop)
    2. Compare it with all remaining user IDs (inner loop)
    3. If any match is found, record it as a duplicate

    Parameters:
        user_ids (list): List of user IDs to check

    Returns:
        set: Set of duplicate user IDs found

    Time Complexity: O(n²) - nested loops iterate through n elements twice
    Space Complexity: O(d) - where d is the number of duplicates

    Performance Issue:
    - For 1,000 IDs: ~1,000,000 comparisons
    - For 10,000 IDs: ~100,000,000 comparisons
    - For 100,000 IDs: ~10,000,000,000 comparisons (very slow!)
    """
    duplicates = set()

    # Outer loop: iterate through each user ID
    for i in range(len(user_ids)):
        # Inner loop: compare with all remaining user IDs
        for j in range(i + 1, len(user_ids)):
            # If IDs match, add to duplicates set
            if user_ids[i] == user_ids[j]:
                duplicates.add(user_ids[i])

    return duplicates
```

```python
def test_duplicate_detection():
    test_cases = {
        "Small Dataset": [101, 102, 103, 102, 104, 105, 103, 106],
        "No Duplicates": [201, 202, 203, 204, 205],
        "Multiple Duplicates": [301, 302, 303, 301, 302, 304, 301, 305, 302],
        "All Duplicates": [401, 401, 401, 402, 402]
    }

    for test_name, user_ids in test_cases.items():
        print(f"\n{'-' * 80}")
        print(f"Test Case: {test_name}")
        print(f"{'-' * 80}")
        print(f"User IDs: {user_ids}")
        print(f"Total IDs: {len(user_ids)}")

        # Brute-force method
        print("\n[METHOD 1: Brute-Force Nested Loops]")
        brute_result = find_duplicates_brute_force(user_ids)
        print(f"Duplicates found: {brute_result if brute_result else 'None'}")
        print(f"Count: {len(brute_result)}")

        # Optimized set method
        print("\n[METHOD 2: Optimized Set-Based]")
        optimized_result = find_duplicates_optimized(user_ids)
        print(f"Duplicates found: {optimized_result if optimized_result else 'None'}")
        print(f"Count: {len(optimized_result)}")

        # Dictionary method with counts
        print("\n[METHOD 3: Dictionary with Counts]")
        dict_result = find_duplicates_with_counts(user_ids)
        if dict_result:
            for uid, count in sorted(dict_result.items()):
                print(f"  ID {uid}: appears {count} times")
        else:
            print("Duplicates found: None")

        # Verify all methods produce same result
        if brute_result == optimized_result == set(dict_result.keys()):
            print("\n✓ All three methods agree on results")
        else:
            print("\n✗ Methods produced different results (ERROR)")

    print(f"\n{'=' * 80}")
    print("COMPLEXITY ANALYSIS AND COMPARISON")
    print(f"{'=' * 80}")
```

Output:

```
[METHOD 3: Dictionary with Counts]
Duplicates found: None

✓ All three methods agree on results

--------------------------------------------------------------------------------
Test Case: Multiple Duplicates
--------------------------------------------------------------------------------
User IDs: [301, 302, 303, 301, 302, 304, 301, 305, 302]
Total IDs: 9

[METHOD 1: Brute-Force Nested Loops]
Duplicates found: {301, 302}
Count: 2

[METHOD 2: Optimized Set-Based]
Duplicates found: {301, 302}
Count: 2

[METHOD 3: Dictionary with Counts]
  ID 301: appears 3 times
  ID 302: appears 3 times

✓ All three methods agree on results

--------------------------------------------------------------------------------
Test Case: All Duplicates
--------------------------------------------------------------------------------
User IDs: [401, 401, 401, 402, 402]
Total IDs: 5

[METHOD 1: Brute-Force Nested Loops]
Duplicates found: {401, 402}
Count: 2

[METHOD 2: Optimized Set-Based]
Duplicates found: {401, 402}
Count: 2

[METHOD 3: Dictionary with Counts]
  ID 401: appears 3 times
  ID 402: appears 2 times

✓ All three methods agree on results
```

Explanation:

- In the brute-force method, the program checks each user ID with every other ID to find duplicates, so it takes more time.

- In the optimized method, the program uses a set to remember IDs and quickly detect if an ID appears again.

- The optimized method is faster because it scans the list only once, so it works better for large data.