

# **ASSIGNMENT-12.4**

Name: E.Ramya

Ht.no: 2303A51282

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:

Write a Python program to sort a list of student exam scores using Bubble Sort. Add simple inline comments explaining comparisons, swaps, and each pass, and include an early stop if the list becomes sorted.

Write a Python program to sort a list of student exam scores using Bubble Sort. Add simple inline comments explaining comparisons, swaps, and each pass, and include an early stop if the list becomes sorted.

bubblesort.py

## Code:

```
bubblesort.py > bubble_sort
def bubble_sort(scores):
    """Sort a list of exam scores using Bubble Sort with early stopping.

    Args:
        scores: List of integers representing exam scores

    Returns:
        Sorted list in ascending order
    """
    n = len(scores)

    # Iterate through each pass
    for i in range(n):
        # Flag to detect if any swaps occur in this pass
        swapped = False

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

        # Early stop: if no swaps occurred, list is already sorted
        if not swapped:
            break

    return scores

# Example usage
if __name__ == "__main__":
    exam_scores = [85, 92, 78, 95, 82, 88, 76, 90]
    print("Original scores:", exam_scores)
    sorted_scores = bubble_sort(exam_scores)
    print("Sorted scores:", sorted_scores)
```

## Output:

```
PS C:\AI Assistant Coding> & C:/Users/edula/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/AI Assistant Coding/bubblesort.py"
Original scores: [85, 92, 78, 95, 82, 88, 76, 90]
Sorted scores: [76, 78, 82, 85, 88, 90, 92, 95]
PS C:\AI Assistant Coding>
```

## Explanation:

- Bubble Sort compares two adjacent scores in the list.
- If the left score is greater than the right score, they are swapped.
- This process repeats for all elements in one pass.
- After each pass, the largest score moves to the end of the list.
- Multiple passes are done until the list becomes sorted.

- If no swaps happen in a pass, the algorithm stops early because the list is already sorted.

Time Complexity:

- Best Case:  $O(n)$  – when the list is already sorted.
- Average Case:  $O(n^2)$
- Worst Case:  $O(n^2)$  – when the list is in reverse order.

## 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

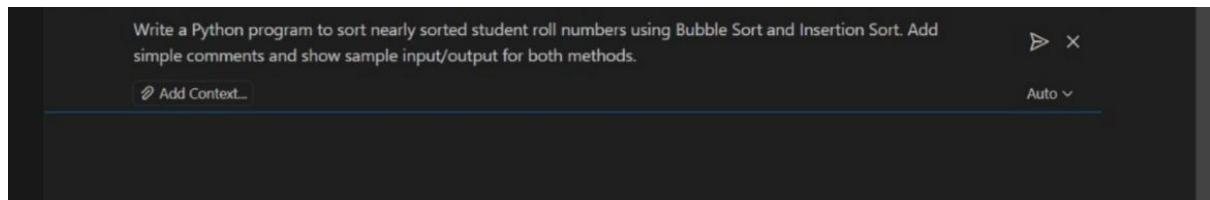
- Start with a Bubble Sort implementation.
- Ask AI to:

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:

Write a Python program to sort nearly sorted student roll numbers using Bubble Sort and Insertion Sort. Add simple comments and show sample input/output for both methods.



## Code:

The screenshot shows the Microsoft Visual Studio Code (VS Code) interface. The left sidebar (EXPLORER) lists various files including Java and Python scripts, PDF assignments, and a LaTeX file. The main editor area contains the following Python code:

```

# Insertion Sort for nearly sorted data
def insertion_sort(arr):
    """Sort array using insertion sort - efficient for nearly sorted data"""
    n = len(arr)
    for i in range(1, n):
        key = arr[i]
        j = i - 1
        # If element greater than key one position ahead
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

# Bubble Sort for nearly sorted data
def bubble_sort(arr):
    """Sort array using bubble sort"""
    n = len(arr)
    swapped
    for i in range(n-1):
        if not swapped:
            swapped = True
            for j in range(0, n-i-1):
                if arr[j] > arr[j+1]:
                    arr[j], arr[j+1] = arr[j+1], arr[j]
                    swapped = False
    # Optimization: stop if no swaps occurred
    if not swapped:
        break
    return arr

# Sample usage
if __name__ == "__main__":
    # Nearly sorted student roll numbers
    roll_numbers = [101, 102, 104, 105, 106, 108, 107, 109, 110]
    print("Original roll numbers:", roll_numbers)
    print()

    # Test Insertion Sort
    arr1 = roll_numbers.copy()
    insertion_sort(arr1)
    print("Insertion Sort Result:", arr1)
    print()

    # Test Bubble Sort
    arr2 = roll_numbers.copy()
    result = bubble_sort(arr2)
    print("Bubble Sort Result:", result)

```

The bottom status bar shows the file path as "C:\AI Assistant Coding\insertionsort.py" and the Python version as "3.11.3 (Microsoft Store)". The terminal tab shows the output of the code execution.

## Output:

```
PS C:\VAI Assistant Coding & C:/Users/edula/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/AI Assistant Coding/bubblesort.py"
PS C:\VAI Assistant Coding & C:/Users/edula/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/AI Assistant Coding/insertionsort.py"
Original roll numbers: [101, 103, 102, 104, 105, 106, 108, 107, 109, 110]
Insertion Sort Result: [101, 102, 103, 104, 105, 106, 107, 108, 109, 110]
Bubble Sort Result: [101, 102, 103, 104, 105, 106, 107, 108, 109, 110]
PS C:\VAI Assistant Coding
```

## Explanation:

The program sorts nearly sorted student roll numbers using two methods: Bubble Sort and Insertion Sort. Bubble Sort repeatedly compares and swaps adjacent elements until the list is sorted, while Insertion Sort places each element in its correct position by shifting larger elements to the right. In nearly sorted data, Insertion Sort is faster because only a few elements need shifting, so it completes in fewer operations compared to Bubble Sort.

## 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

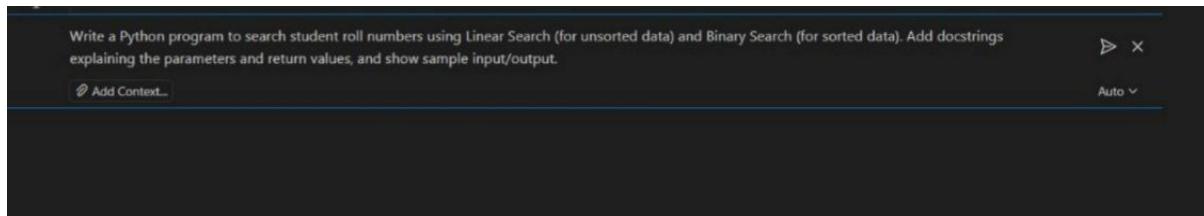
## Binary Search for sorted student data

- Use AI to:

Add docstrings explaining parameters and return values ,Explain when Binary Search is applicable  
Highlight performance differences between the two searches

### **Prompt:**

Write a Python program to search student roll numbers using Linear Search (for unsorted data) and Binary Search (for sorted data). Add docstrings explaining the parameters and return values, and show sample input/output.



### **Code:**

## Output:

```
PS C:\AI Assistant Coding> & C:/Users/edula/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/AI Assistant Coding/searching.py"
---  
== Linear Search (Unsorted Data) ==  
Roll numbers: [105, 103, 101, 104, 102]  
Searching for 104: Found at index 3  
  
== Binary Search (Sorted Data) ==  
Roll numbers: [101, 102, 103, 104, 105]  
Searching for 104: Found at index 3  
Searching for 110: Found at index -1  
PS C:\AI Assistant Coding>
```

## Explanation:

- The program uses Linear Search to find a roll number in an unsorted list by checking each element one by one.
  - It uses Binary Search for a sorted list by repeatedly dividing the list into halves until the roll number is found or not found.

- Binary Search is faster because it reduces the search space each step, while Linear Search checks all elements in the worst case.

Time Complexity:

- Linear Search: Best O(1), Worst O(n)
- Binary Search: Best O(1), Worst O(log n)

## **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).

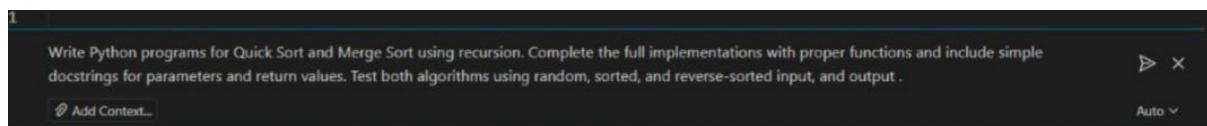
Task Description

- 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:

Write Python programs for Quick Sort and Merge Sort using recursion. Complete the full implementations with proper functions and include simple docstrings for parameters and return values. Test both algorithms using random, sorted, and reverse-sorted input, and output .



## Code:

The screenshot shows the Microsoft Visual Studio Code (VS Code) interface with the 'AI Assistant Coding' extension open. The left sidebar shows an 'EXPLORER' view with various files and documents. The main area is a code editor displaying two Python files: 'quicksort.py' and 'mergesort.py'. The 'quicksort.py' file contains the implementation of the quick sort algorithm, and the 'mergesort.py' file contains the implementation of the merge sort algorithm. Both files have detailed docstrings explaining their parameters and return values.

```
1
def quick_sort(arr):
    """
    Sort array using Quick Sort algorithm.

    Parameters:
    | arr (list): List of elements to sort

    Returns:
    | list: sorted list
    """
    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)

def merge_sort(arr):
    """
    Sort array using Merge Sort algorithm.

    Parameters:
    | arr (list): List of elements to sort

    Returns:
    | list: sorted list
    """
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
```

File Edit Selection View ... ← → Q AI Assistant Coding

EXPLORER OPEN EDITORS

- student.py
- function.py
- temperATURE.PY
- vote.py
- campus\_resource\_management.py
- bubblesort.py
- insertionsort.py
- searching.py
- quicksort.py**
- AI ASSISTANT CODING
- Z303A51282\_ASSIGNMENT-1.3.pdf
- Z303A51282\_ASSIGNMENT-2.3.pdf
- Z303A51282\_ASSIGNMENT-3.3.pdf
- Z303A51282\_ASSIGNMENT-4.4.pdf
- Z303A51282\_ASSIGNMENT-5.4.pdf
- Z303A51282\_ASSIGNMENT-6.3.pdf
- Z303A51282\_ASSIGNMENT-7.3.pdf
- Z303A51282\_ASSIGNMENT-8.4.docx
- Z303A51282\_ASSIGNMENT-8.4.pdf
- Z303A51282\_ASSIGNMENT-9.3.docx
- Z303A51282\_ASSIGNMENT-9.3.pdf
- Z303A51282\_ASSIGNMENT-10.3.pdf
- Z303A51282\_ASSIGNMENT-10.4.pdf
- Z303A51282\_ASSIGNMENT-11.3.docx
- Z303A51282\_ASSIGNMENT-11.3.pdf
- Z303A51317\_ASSIGNMENT-11.3.pdf
- another\_file.txt
- ASSIGNMENT-3-3.docx
- ASSIGNMENT-1.3.docx
- ASSIGNMENT-2.3.docx
- ASSIGNMENT-4.4.docx
- ASSIGNMENT-5.4.docx
- ASSIGNMENT-6.3.docx
- Assignment-7.3.docx
- asss.py

```

quicksort.py > ...
25 def merge_sort(arr):
42     return merge(left, right)
43
44     def merge(left, right):
45         """
46             Merge two sorted arrays.
47
48             Parameters:
49                 left (list): First sorted list
50                 right (list): Second sorted list
51
52             Returns:
53                 list: Merged sorted list
54
55             """
56
57             result = []
58             i = j = 0
59
60             while i < len(left) and j < len(right):
61                 if left[i] <= right[j]:
62                     result.append(left[i])
63                     i += 1
64                 else:
65                     result.append(right[j])
66                     j += 1
67
68             result.extend(left[i:])
69             result.extend(right[j:])
70
71             return result
72
73     def test_sorting_algorithms():
74         """Test Quick Sort and Merge Sort with different input patterns."""
75
76         # Create test datasets
77         size = 1000
78         random_arr = [random.randint(1, 1000) for _ in range(size)]
79         sorted_arr = list(range(1, size + 1))
80         reverse_sorted_arr = list(range(size, 0, -1))
81
82         test_cases = [
83             ("Random", random_arr.copy()),
84             ("Sorted", sorted_arr.copy()),
85             ("Reverse Sorted", reverse_sorted_arr.copy())
86         ]
87
88         print("=" * 70)
89         print(f"{'Test Case':<20} {'Algorithm':<15} {'Time (ms)':<15} {'Correct':<10}")
90         print("=" * 70)
91
92         for test_name, arr in test_cases:
93             # Test Quick Sort
94             arr_copy = arr.copy()
95             start = time.perf_counter()
96             result_qs = quick_sort(arr_copy)
97             time_qs = (time.perf_counter() - start) * 1000
98             is_correct_qs = result_qs == sorted(arr)
99             print(f"{'test_name':<20} {'Quick Sort':<15} {time_qs:<15.4f} {str(is_correct_qs):<10}")
100
101             # Test Merge Sort
102             arr_copy = arr.copy()
103             start = time.perf_counter()
104             result_ms = merge_sort(arr_copy)
105             time_ms = (time.perf_counter() - start) * 1000
106             is_correct_ms = result_ms == sorted(arr)
107             print(f"{'test_name':<20} {'Merge Sort':<15} {time_ms:<15.4f} {str(is_correct_ms):<10}")
108             print("=" * 70)
109
110         if __name__ == "__main__":
111             test_sorting_algorithms()

```

Ln 111, Col 30 Spaces: 4 UTF-8 {} Python 3.11.9 (Microsoft Store) Go Live Prettier

File Edit Selection View ... ← → Q AI Assistant Coding

EXPLORER OPEN EDITORS

- student.py
- function.py
- temperATURE.PY
- vote.py
- campus\_resource\_management.py
- bubblesort.py
- insertionsort.py
- searching.py
- quicksort.py**
- AI ASSISTANT CODING
- Z303A51282\_ASSIGNMENT-1.3.pdf
- Z303A51282\_ASSIGNMENT-2.3.pdf
- Z303A51282\_ASSIGNMENT-3.3.pdf
- Z303A51282\_ASSIGNMENT-4.4.pdf
- Z303A51282\_ASSIGNMENT-5.4.pdf
- Z303A51282\_ASSIGNMENT-6.3.pdf
- Z303A51282\_ASSIGNMENT-7.3.pdf
- Z303A51282\_ASSIGNMENT-8.4.docx
- Z303A51282\_ASSIGNMENT-8.4.pdf
- Z303A51282\_ASSIGNMENT-9.3.docx
- Z303A51282\_ASSIGNMENT-9.3.pdf
- Z303A51282\_ASSIGNMENT-10.3.pdf
- Z303A51282\_ASSIGNMENT-10.4.pdf
- Z303A51282\_ASSIGNMENT-11.3.docx
- Z303A51282\_ASSIGNMENT-11.3.pdf
- Z303A51317\_ASSIGNMENT-11.3.pdf
- another\_file.txt
- ASSIGNMENT-3-3.docx
- ASSIGNMENT-1.3.docx
- ASSIGNMENT-2.3.docx
- ASSIGNMENT-4.4.docx
- ASSIGNMENT-5.4.docx
- ASSIGNMENT-6.3.docx
- Assignment-7.3.docx
- asss.py

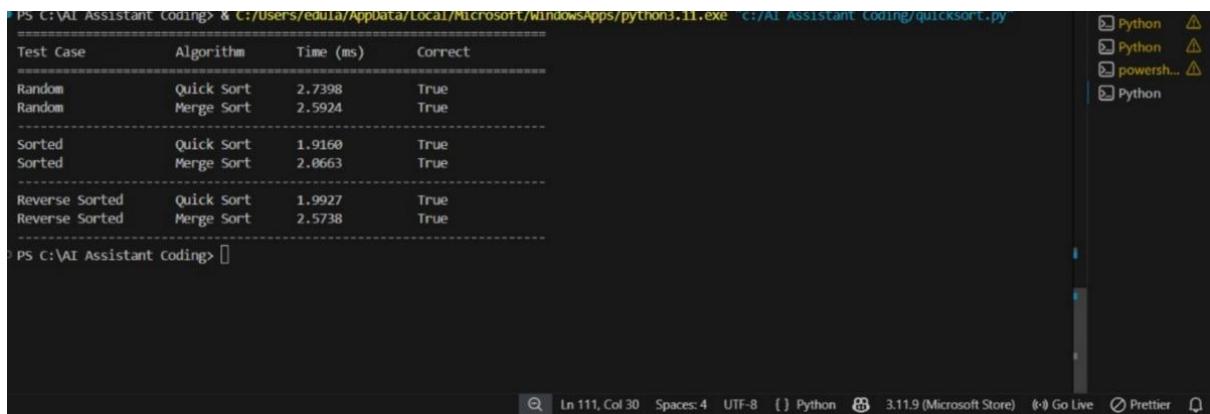
```

quicksort.py > ...
72     def test_sorting_algorithms():
73         """Test Quick Sort and Merge Sort with different input patterns."""
74
75         # Create test datasets
76         size = 1000
77         random_arr = [random.randint(1, 1000) for _ in range(size)]
78         sorted_arr = list(range(1, size + 1))
79         reverse_sorted_arr = list(range(size, 0, -1))
80
81         test_cases = [
82             ("Random", random_arr.copy()),
83             ("Sorted", sorted_arr.copy()),
84             ("Reverse Sorted", reverse_sorted_arr.copy())
85         ]
86
87         print("=" * 70)
88         print(f"{'Test Case':<20} {'Algorithm':<15} {'Time (ms)':<15} {'Correct':<10}")
89         print("=" * 70)
90
91         for test_name, arr in test_cases:
92             # Test Quick Sort
93             arr_copy = arr.copy()
94             start = time.perf_counter()
95             result_qs = quick_sort(arr_copy)
96             time_qs = (time.perf_counter() - start) * 1000
97             is_correct_qs = result_qs == sorted(arr)
98             print(f"{'test_name':<20} {'Quick Sort':<15} {time_qs:<15.4f} {str(is_correct_qs):<10}")
99             print("=" * 70)
100
101             # Test Merge Sort
102             arr_copy = arr.copy()
103             start = time.perf_counter()
104             result_ms = merge_sort(arr_copy)
105             time_ms = (time.perf_counter() - start) * 1000
106             is_correct_ms = result_ms == sorted(arr)
107             print(f"{'test_name':<20} {'Merge Sort':<15} {time_ms:<15.4f} {str(is_correct_ms):<10}")
108             print("=" * 70)
109
110         if __name__ == "__main__":
111             test_sorting_algorithms()

```

Ln 111, Col 30 Spaces: 4 UTF-8 {} Python 3.11.9 (Microsoft Store) Go Live Prettier

# Output:



```
PS C:\AI Assistant Coding> & C:/Users/edua/AppData/Local/Microsoft/WindowsApps/python3.11.exe C:/AI Assistant Coding/quicksort.py
=====
Test Case      Algorithm      Time (ms)      Correct
=====
Random         Quick Sort    2.7398        True
Random         Merge Sort    2.5924        True
Sorted          Quick Sort   1.9160        True
Sorted          Merge Sort   2.0663        True
Reverse Sorted Quick Sort   1.9927        True
Reverse Sorted Merge Sort   2.5738        True
=====

PS C:\AI Assistant Coding> [REDACTED]
```

Ln 111, Col 30 Spaces: 4 UTF-8 { } Python 3.11.9 (Microsoft Store) Go Live Prettier

## Explanation:

- Both programs use recursion, meaning the function calls itself to sort smaller parts of the list.
- Quick Sort selects a pivot element and places smaller elements on the left and larger elements on the right, then recursively sorts both sides.
- Merge Sort divides the list into two halves, recursively sorts each half, and then merges the sorted halves together.

## Time Complexity:

- Quick Sort: Best/Average  $O(n \log n)$ , Worst  $O(n^2)$
- Merge Sort: Best/Average/Worst  $O(n \log n)$

## **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.

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.

Add Context... Auto ▾

## Code:

```

OPEN EDITORS
  temperATURE.PY | vote.py | campus_resource_management.py | bubblesort.py | insertionsort.py | searching.py | quicksort.py | detect.py | bug.py
  evenoddsnum.py
  student.py
  functions.py
  temperATURE.PY
  vote.py
  campus_resource_management.py
  bubblesort.py
  insertionsort.py
  searching.py
  quicksort.py
  detect.py
  another_file.txt
  ASSIGNMENT-3-3.docx
  ASSIGNMENT-13.docx
  ASSIGNMENT-23.docx
  ASSIGNMENT-4-4.docx
  ASSIGNMENT-5-4.docx
  ASSIGNMENT-6-3.docx
  Assignment-7-3.doc
  ass.py
  bubblesort.py
  bug.py
  campus_resource_management.py
  contact_manager.py
  detect.py
  evenoddsnum.py
  fib.java
  fib.cpp
  files.py
  function.py
  insertionsort.py
  library_book_request.py
  quicksort.py

AI ASSISTANT CODING
  2303A5128_ASSIGNMENT-8-4.docx
  2303A5128_ASSIGNMENT-8-4.pdf
  2303A5128_ASSIGNMENT-9-3.docx
  2303A5128_ASSIGNMENT-9-3.pdf
  2303A5128_ASSIGNMENT-10-3.pdf
  2303A5128_ASSIGNMENT-10-4.pdf
  2303A5128_ASSIGNMENT-11-3.docx
  2303A5128_ASSIGNMENT-11-3.pdf
  2303A5117_ASSIGNMENT-11-3.pdf

another_file.txt
ASSIGNMENT-3-3.docx
ASSIGNMENT-13.docx
ASSIGNMENT-23.docx
ASSIGNMENT-4-4.docx
ASSIGNMENT-5-4.docx
ASSIGNMENT-6-3.docx
Assignment-7-3.doc
ass.py
bubblesort.py
bug.py
campus_resource_management.py
contact_manager.py
detect.py
evenoddsnum.py
fib.java
fib.cpp
files.py
function.py
insertionsort.py
library_book_request.py
quicksort.py

detect.py > find_duplicates_brute_force
  # Time Complexity: O(n^2)
  # Space Complexity: O(1)

  def find_duplicates_brute_force(user_ids):
    """
    Detect duplicates using nested loops.
    For each element, compare with all other elements.
    """
    duplicates = []
    for i in range(len(user_ids)):
        for j in range(i + 1, len(user_ids)):
            if user_ids[i] == user_ids[j] and user_ids[i] not in duplicates:
                duplicates.append(user_ids[i])
    return duplicates

# ----- OPTIMIZED APPROACH -----
# Time Complexity: O(n)
# Space Complexity: O(n)

def find_duplicates_optimized(user_ids):
    """
    Detect duplicates using a set for O(1) lookups.
    Single pass through the list.
    """
    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 list(duplicates)

# ----- COMPARISON & TESTING -----
if __name__ == "__main__":
    # Test data
    test_users = [101, 102, 103, 102, 104, 101, 105, 103]
    print("User IDs:", test_users)
    print("Brute Force Result:", find_duplicates_brute_force(test_users))
    print("Optimized Result:", find_duplicates_optimized(test_users))
    print("\n--- Time Complexity Analysis ---")
    print("Brute Force: O(n^2) - nested loops compare each pair")
    print("Optimized: O(n) - single pass with set lookup O(1)")

detect.py

```

## Output:

```

PS C:\AI Assistant Coding> & C:/Users/edula/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/AI Assistant Coding/detect.py"
User IDs: [101, 102, 103, 102, 104, 101, 105, 103]

Brute Force Result: [101, 102, 103]
Optimized Result: [101, 102, 103]

--- Time Complexity Analysis ---
Brute Force: O(n^2) - nested loops compare each pair
Optimized: O(n) - single pass with set lookup O(1)

```

## 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.