ASSIGNMENT - 12.4

PROMPT: 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.

CODE:

```
[1]
✓ 0s
def bubble_sort_scores(scores):
    n = len(scores)

    # Outer loop: Iterate through the entire list
    for i in range(n):
        # Track if any swap happened in this pass for early termination
        swapped = False

        # Inner loop: Compare adjacent elements
        # The last i elements are already in place, so we ignore them
        for j in range(0, n - i - 1):

            # Comparison: Check if the current score is greater than the next
            if scores[j] > scores[j + 1]:

                # Swap: If they are in the wrong order, swap them
                scores[j], scores[j + 1] = scores[j + 1], scores[j]

                # Mark that a swap occurred
                swapped = True

        # Early-Termination: If no two elements were swapped by the
        # inner loop, then the list is already sorted.
        if not swapped:
            break

    return scores
```

OUTPUT:

```
[1]
✓ 0s
# Sample Input: Student Exam Scores
student_scores = [88, 72, 95, 61, 88, 50, 99]

print("Original Scores:", student_scores)
sorted_scores = bubble_sort_scores(student_scores)
print("Sorted Scores:  ", sorted_scores)

Original Scores: [88, 72, 95, 61, 88, 50, 99]
Sorted Scores:   [50, 61, 72, 88, 88, 95, 99]
```

EXPLANATION:

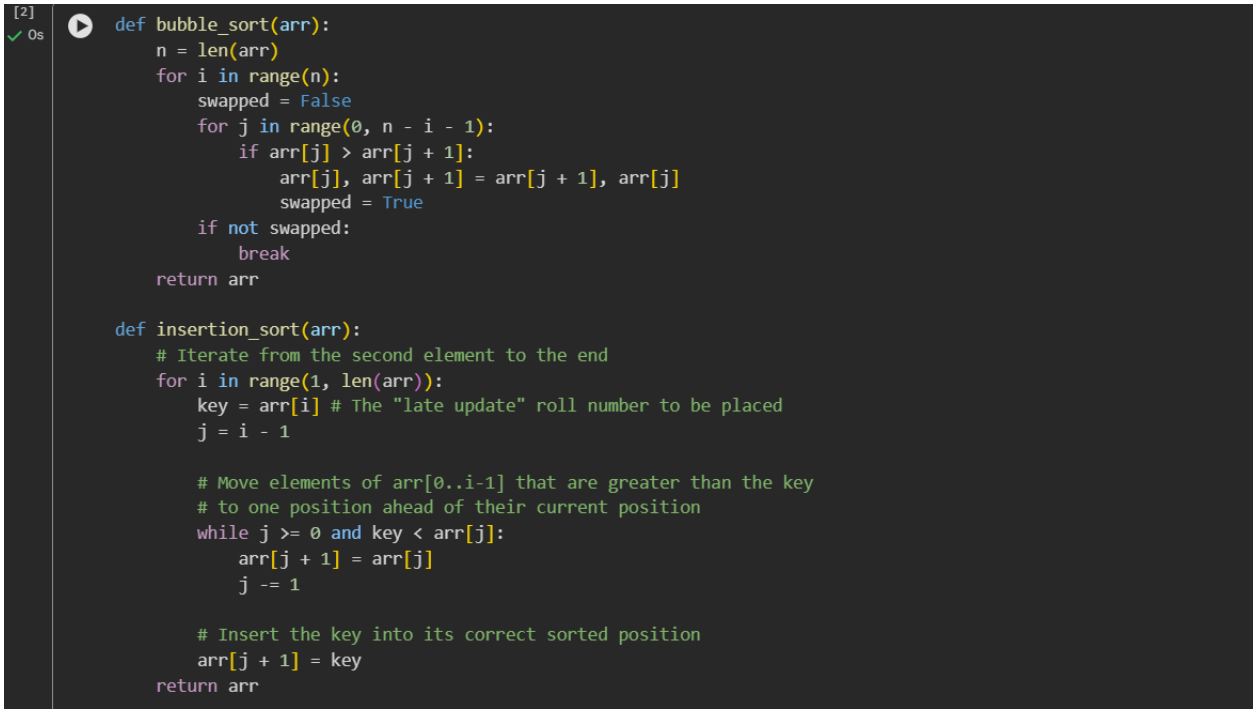
1. **The "Bubble" Effect:** In every pass through the list, the largest unsorted value "bubbles up" to its correct position at the end of the array.

2. **Passes:** For a list of size n, the algorithm requires at most $n-1$ passes.

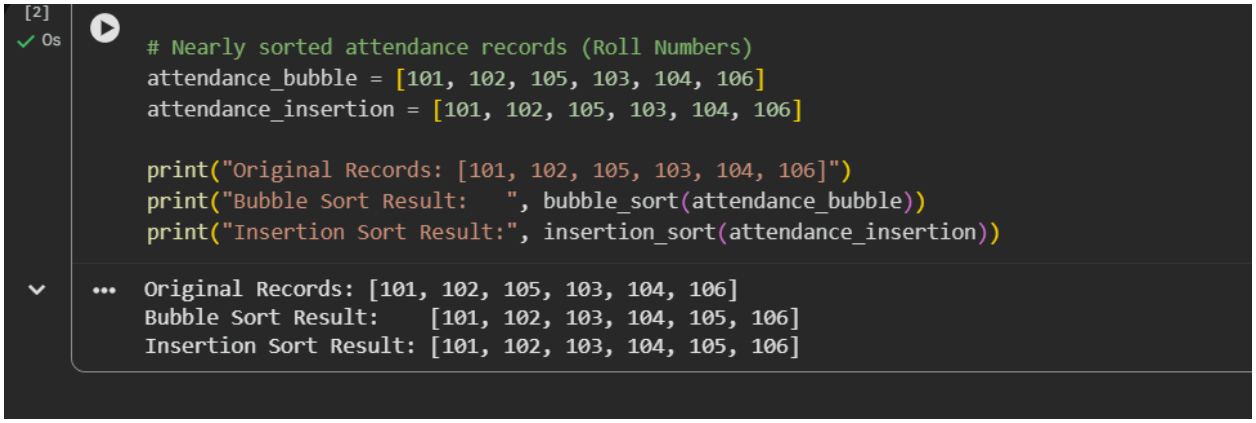
3. **Efficiency:** By including the swapped flag, we ensure that if the scores are already organized (e.g., after a quick manual entry fix), the code doesn't waste cycles re-checking them.

PROMPT:
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.

CODE:



OUTPUT:



```
[2]
✓ 0s

# Nearly sorted attendance records (Roll Numbers)
attendance_bubble = [101, 102, 105, 103, 104, 106]
attendance_insertion = [101, 102, 105, 103, 104, 106]

print("Original Records: [101, 102, 105, 103, 104, 106]")
print("Bubble Sort Result:   ", bubble_sort(attendance_bubble))
print("Insertion Sort Result:", insertion_sort(attendance_insertion))

Original Records: [101, 102, 105, 103, 104, 106]
Bubble Sort Result:    [101, 102, 103, 104, 105, 106]
Insertion Sort Result: [101, 102, 103, 104, 105, 106]
```

EXPLANATION:

- Adaptive Nature: Insertion Sort is "adaptive," meaning it gets faster as the list becomes more sorted. For nearly sorted data, it achieves near $O(n)$ (linear) time, whereas Bubble Sort still struggles with redundant neighbor swaps.
- Less Overhead: While Bubble Sort must constantly swap two elements (requiring 3 steps in memory), Insertion Sort simply shifts elements over to create a gap, which is computationally "cheaper."
- Surgical Precision: Think of it like a librarian. Bubble Sort would swap every book on the shelf one by one to move a single book; Insertion Sort simply picks up the misplaced book and slides it directly into the correct slot.
- Minimal Comparisons: In a nearly sorted list, the "inner loop" of Insertion Sort rarely runs, allowing the algorithm to verify the order in a single quick pass with very few comparisons.

PROMPT:

Searching Student Records in a
Database
Scenario
You are developing a student information portal where users search for student records by roll number.

CODE:

```python
def linear_search(records, target):
    for i in range(len(records)):
        if records[i] == target:
            return i  # Found at index i
    return -1

def binary_search(records, target):
    """
    Performs a logarithmic search by repeatedly halving the search interval.

    :param records: A LIST OF SORTED student roll numbers.
    :param target: The roll number to find.
    :return: The index of the target if found, otherwise -1.
    """
    low = 0
    high = len(records) - 1

    while low <= high:
        mid = (low + high) // 2
        if records[mid] == target:
            return mid
        elif records[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

    return -1
```

CODE/OUTPUT:

```python
# --- Sample Execution ---
unsorted_data = [105, 120, 101, 115, 110]
sorted_data = [101, 105, 110, 115, 120]
target_roll = 115

print(f"Searching for Roll No: {target_roll}")
print(f"Linear Search (Unsorted Data) Index: {linear_search(unsorted_data, target_roll)}")
print(f"Binary Search (Sorted Data) Index:   {binary_search(sorted_data, target_roll)}")
```

```
Searching for Roll No: 115
Linear Search (Unsorted Data) Index: 3
Binary Search (Sorted Data) Index:   3
```

EXPLANATION:

- Linear Search is the only option for unsorted data because it checks every record one by one; it is simple but becomes very slow as the database grows.
- Binary Search is extremely fast because it uses a "divide and conquer" approach, cutting the search area in half each step—however, it requires the data to be sorted beforehand to work.
- Performance Gap: For a database of 1,000 students, Linear Search might take 1,000 comparisons, while Binary Search will find any record in just 10 steps ($log_2 1000 \approx 10$

PROMPT:

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).

CODE:

```python
import sys

# Increase recursion depth for large/sorted datasets in Quick Sort
sys.setrecursionlimit(2000)

def quick_sort(arr):
    if len(arr) <= 1:
        return arr

    # Selecting the middle element as pivot to handle sorted data better
    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]

    # Recursion: Sort the left and right, then combine
    return quick_sort(left) + middle + quick_sort(right)

def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    # Recursion: Split the list into two halves
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])

    # Merge logic: Combine two sorted lists into one
    return merge(left_half, right_half)
```

```
    def merge(left, right):
        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

    # --- Testing Scenarios ---
    data_variants = {
        "Random": [34, 7, 23, 32, 5, 62],
        "Sorted": [5, 7, 23, 32, 34, 62],
        "Reverse": [62, 34, 32, 23, 7, 5]
    }

    for name, data in data_variants.items():
        print(f"\n{name} Data: {data}")
        print(f"Quick Sort: {quick_sort(data.copy())}")
        print(f"Merge Sort: {merge_sort(data.copy())}")
```

OUTPUT:

```
...
    Random Data: [34, 7, 23, 32, 5, 62]
    Quick Sort: [5, 7, 23, 32, 34, 62]
    Merge Sort: [5, 7, 23, 32, 34, 62]

    Sorted Data: [5, 7, 23, 32, 34, 62]
    Quick Sort: [5, 7, 23, 32, 34, 62]
    Merge Sort: [5, 7, 23, 32, 34, 62]

    Reverse Data: [62, 34, 32, 23, 7, 5]
    Quick Sort: [5, 7, 23, 32, 34, 62]
    Merge Sort: [5, 7, 23, 32, 34, 62]
```

EXPLANATION:

- Recursive Logic: Both use "Divide and Conquer"; Quick Sort partitions data around a pivot, while Merge Sort focuses on splitting and merging halves consistently.
- Performance: Quick Sort is usually faster in memory (cache-efficient), but Merge Sort is preferred for linked lists or when you need a guaranteed time regardless of whether the data is already sorted or reversed.
- The Difference: While Quick Sort can degrade to $O(n^2)$ if a poor pivot is chosen, Merge Sort never slows down, making it more predictable for critical data pipelines.

PROMPT:

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.

CODE:

```
[6]
✓ 0s
def find_duplicates_naive(user_ids):
    """
    Brute-force approach using nested loops.
    Compares every ID with every other ID.
    """
    duplicates = []
    for i in range(len(user_ids)):
        for j in range(i + 1, len(user_ids)):
            # If a match is found and it's not already in our duplicate list
            if user_ids[i] == user_ids[j] and user_ids[i] not in duplicates:
                duplicates.append(user_ids[i])
    return duplicates

def find_duplicates_optimized(user_ids):
    """
    Optimized approach using a Set for O(1) average lookup time.
    """
    seen = set()
    duplicates = set()

    for uid in user_ids:
        if uid in seen:
            duplicates.add(uid)
        else:
            seen.add(uid)

    return list(duplicates)
```

```
[6]
✓ 0s
# --- Sample Dataset ---
data = [101, 205, 302, 101, 404, 205, 500]

print(f"Dataset: {data}")
print(f"Naive Result:     {find_duplicates_naive(data)}")
print(f"Optimized Result: {find_duplicates_optimized(data)}")

Dataset: [101, 205, 302, 101, 404, 205, 500]
Naive Result:     [101, 205]
Optimized Result: [205, 101]
```

EXPLANATION:

- **The Brute-force ($O(n^2)$) approach is slow because it uses nested loops to compare every ID against every other ID, which becomes unmanageable as data grows.**
- **The Optimized ($O(n)$) approach is far superior because it uses a Hash Set for near-instant lookups, allowing the algorithm to detect duplicates in a single pass through the dataset**