

Assignment 12.4

N SOURISH

2303A52360

Batch: 44

Task 1: Bubble Sort for Ranking Exam Scores

Code:

```
def bubble_sort(scores):
    """
    Sorts a list of scores using Bubble Sort and returns a new sorted list.

    Best case: O(n) when the list is already sorted (early termination).
    Average case: O(n^2)
    Worst case: O(n^2)
    """

    sorted_scores = scores.copy() # Keep original input unchanged
    n = len(sorted_scores)
    # Outer loop controls the number of passes through the list
    for pass_num in range(n - 1):
        swapped = False # Tracks if any swap occurs in this pass
        # Inner loop performs pairwise comparisons for the unsorted portion
        for index in range(0, n - 1 - pass_num):
            # Compare adjacent elements
            if sorted_scores[index] > sorted_scores[index + 1]:
                # Swap when elements are in the wrong order
                sorted_scores[index], sorted_scores[index + 1] = (
                    sorted_scores[index + 1],
                    sorted_scores[index],
                )
                swapped = True
        # Early termination: if no swaps happened, list is already sorted
        if not swapped:
            break
    return sorted_scores

if __name__ == "__main__":
    # Sample input: exam scores after an internal assessment
    student_scores = [78, 92, 65, 88, 70, 95, 81]
    print("Sample Input (Unsorted Scores):", student_scores)
    ranked_scores = bubble_sort(student_scores)
    print("Sample Output (Sorted Scores):", ranked_scores)
    print("\nTime Complexity Analysis:")
    print("- Best Case: O(n) -> already sorted (early termination)")
    print("- Average Case: O(n^2)")
    print("- Worst Case: O(n^2) -> reverse sorted")
```

Input & Output:

```
Sample Input (Unsorted Scores): [78, 92, 65, 88, 70, 95, 81]
Sample Output (Sorted Scores): [65, 70, 78, 81, 88, 92, 95]
```

Time Complexity Analysis:

- Best Case: $O(n)$ -> already sorted (early termination)
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$ -> reverse sorted

Task 2: Improving Sorting for Nearly Sorted Attendance Records.

Code:

```
def bubble_sort(arr):
    """Sorts using Bubble Sort and returns (sorted_list, stats)."""
    data = arr.copy()
    n = len(data)
    comparisons = 0
    swaps = 0
    passes = 0
    # Each pass moves the largest unsorted element toward the end.
    for pass_num in range(n - 1):
        swapped = False
        passes += 1
        # Compare adjacent elements in the unsorted part of the list.
        for index in range(0, n - 1 - pass_num):
            comparisons += 1
            if data[index] > data[index + 1]:
                data[index], data[index + 1] = data[index + 1], data[index]
                swaps += 1
                swapped = True
        # Early termination: stop if the list is already sorted.
        if not swapped:
            break
    stats = {
        "comparisons": comparisons,
        "swaps": swaps,
        "passes": passes,
    }
    return data, stats

def insertion_sort(arr):
    """Sorts using Insertion Sort and returns (sorted_list, stats)."""
    data = arr.copy()

    comparisons = 0
    shifts = 0
    passes = 0

    # Build a sorted portion from left to right.
    for index in range(1, len(data)):
        key = data[index]
        position = index - 1
        passes += 1
```

```

# Move larger elements one step right to make room for key.
while position >= 0:
    comparisons += 1
    if data[position] > key:
        data[position + 1] = data[position]
        shifts += 1
        position -= 1
    else:
        break

# Insert key at its correct position.
data[position + 1] = key

stats = {
    "comparisons": comparisons,
    "shifts": shifts,
    "passes": passes,
}
return data, stats

if __name__ == "__main__":
    # Nearly sorted attendance roll numbers (only a few misplaced entries).
    roll_numbers = [101, 102, 103, 104, 106, 105, 107, 108, 109, 111, 110, 112]

    print("Nearly Sorted Input:", roll_numbers)

    bubble_sorted, bubble_stats = bubble_sort(roll_numbers)
    insertion_sorted, insertion_stats = insertion_sort(roll_numbers)

    print("\nBubble Sort Output:", bubble_sorted)
    print("Bubble Sort Stats:", bubble_stats)

    print("\nInsertion Sort Output:", insertion_sorted)
    print("Insertion Sort Stats:", insertion_stats)

    print("\nAI-Assisted Review:")
    print("- More suitable algorithm for nearly sorted data: Insertion Sort")
    print("- Why: It shifts only the few out-of-place elements, so work stays low.")
    print("- Bubble Sort still scans adjacent pairs repeatedly across passes.")

    print("\nComplexity Notes:")
    print("- Bubble Sort: Best O(n) with early stop, Average/Worst O(n^2)")
    print("- Insertion Sort: Best O(n), Average/Worst O(n^2)")
    print("- On nearly sorted lists, Insertion Sort usually performs fewer operations.")

```

Input & Output:

```

Nearly Sorted Input: [101, 102, 103, 104, 106, 105, 107, 108, 109, 111, 110, 112]

Bubble Sort Output: [101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112]
Bubble Sort Stats: {'comparisons': 21, 'swaps': 2, 'passes': 2}

Insertion Sort Output: [101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112]
Insertion Sort Stats: {'comparisons': 13, 'shifts': 2, 'passes': 11}

AI-Assisted Review:
- More suitable algorithm for nearly sorted data: Insertion Sort
- Why: It shifts only the few out-of-place elements, so work stays low.
- Bubble Sort still scans adjacent pairs repeatedly across passes.

Complexity Notes:
- Bubble Sort: Best O(n) with early stop, Average/Worst O(n^2)
- Insertion Sort: Best O(n), Average/Worst O(n^2)
- On nearly sorted lists, Insertion Sort usually performs fewer operations.

```

Task 3: Searching Student Records in a Database

```
def linear_search_student(records, target_roll):
    for index, record in enumerate(records):
        if record["roll"] == target_roll:
            return index, record
    return -1, None

def binary_search_student(sorted_records, target_roll):
    left = 0
    right = len(sorted_records) - 1

    while left <= right:
        mid = (left + right) // 2
        mid_roll = sorted_records[mid]["roll"]

        if mid_roll == target_roll:
            return mid, sorted_records[mid]
        if mid_roll < target_roll:
            left = mid + 1
        else:
            right = mid - 1

    return -1, None

if __name__ == "__main__":
    # Unsorted student records (typical raw database export order)
    student_records_unsorted = [
        {"roll": 108, "name": "Asha"}, 
        {"roll": 102, "name": "Ravi"}, 
        {"roll": 115, "name": "Neha"}, 
        {"roll": 101, "name": "Arjun"}, 
        {"roll": 110, "name": "Meera"}, 
        {"roll": 105, "name": "Kabir"}, 
    ]
    target = 110
```

```

# Linear search works directly on unsorted data.
linear_index, linear_record = linear_search_student(student_records_unsorted, target)

# Binary search requires data sorted by roll number.
student_records_sorted = sorted(student_records_unsorted, key=lambda rec: rec["roll"])
binary_index, binary_record = binary_search_student(student_records_sorted, target)

print("Unsorted Records:", student_records_unsorted)
print("Sorted Records:", student_records_sorted)

print("\nSearch Target Roll:", target)
print("Linear Search Result (unsorted):", (linear_index, linear_record))
print("Binary Search Result (sorted):", (binary_index, binary_record))

print("\nAI-Generated Explanation:")
print("- Binary Search is applicable only when records are sorted by roll number.")
print("- Linear Search scans one-by-one, so it works for both unsorted and sorted lists.")
print("- Binary Search halves the search space each step, making it much faster on large sorted data.")

print("\nTime Complexity:")
print("- Linear Search: Best O(1), Average O(n), Worst O(n)")
print("- Binary Search: Best O(1), Average O(log n), Worst O(log n)")

print("\nUse Cases:")
print("- Use Linear Search when data is unsorted or very small.")
print("- Use Binary Search when data is already sorted and frequent lookups are needed.")

print("\nStudent Observation:")
print("For unsorted records, Linear Search is straightforward and reliable.")
print("After sorting by roll number, Binary Search finds the same student with fewer checks,")
print("which is more efficient for partially ordered or fully ordered datasets.")

```

Input & Output:

```

Unsorted Records: [{"roll": 108, "name": "Asha"}, {"roll": 102, "name": "Ravi"}, {"roll": 115, "name": "Neha"}, {"roll": 101, "name": "Arjun"}, {"roll": 110, "name": "Meera"}, {"roll": 105, "name": "Kabir"}]
Sorted Records: [{"roll": 101, "name": "Arjun"}, {"roll": 102, "name": "Ravi"}, {"roll": 105, "name": "Kabir"}, {"roll": 108, "name": "Asha"}, {"roll": 110, "name": "Meera"}, {"roll": 115, "name": "Neha"}]

Search Target Roll: 110
Linear Search Result (unsorted): (4, {"roll": 110, "name": "Meera"})
Binary Search Result (sorted): (4, {"roll": 110, "name": "Meera"})

AI-Generated Explanation:
- Binary Search is applicable only when records are sorted by roll number.
- Linear Search scans one-by-one, so it works for both unsorted and sorted lists.
- Binary Search halves the search space each step, making it much faster on large sorted data.

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

Use Cases:
- Use Linear Search when data is unsorted or very small.
- Use Binary Search when data is already sorted and frequent lookups are needed.

Student Observation:
For unsorted records, Linear Search is straightforward and reliable.
After sorting by roll number, Binary Search finds the same student with fewer checks,
which is more efficient for partially ordered or fully ordered datasets.

```

Task 4: Choosing Between Quick Sort and Merge Sort

For Data Processing

Code:

```
from random import sample

def quick_sort(data):
    if len(data) <= 1:
        return data.copy()

    pivot = data[len(data) // 2]
    left = [value for value in data if value < pivot]
    middle = [value for value in data if value == pivot]
    right = [value for value in data if value > pivot]

    return quick_sort(left) + middle + quick_sort(right)

def merge_sort(data):
    if len(data) <= 1:
        return data.copy()
    mid = len(data) // 2
    left_half = data[:mid]
    right_half = data[mid:]

    sorted_left = merge_sort(left_half)
    sorted_right = merge_sort(right_half)
    merged = []
    left_index = 0
    right_index = 0
    while left_index < len(sorted_left) and right_index < len(sorted_right):
        if sorted_left[left_index] <= sorted_right[right_index]:
            merged.append(sorted_left[left_index])
            left_index += 1
        else:
            merged.append(sorted_right[right_index])
            right_index += 1

    merged.extend(sorted_left[left_index:])
    merged.extend(sorted_right[right_index:])
    return merged
```

```

def run_test_case(case_name, values):
    """Run both sorts on one dataset and print comparable results."""
    quick_result = quick_sort(values)
    merge_result = merge_sort(values)

    print(f"\n{case_name}:")
    print("Input:", values)
    print("Quick Sort:", quick_result)
    print("Merge Sort:", merge_result)
    print("Same Output:", quick_result == merge_result)

if __name__ == "__main__":
    random_data = sample(range(1, 60), 12)
    sorted_data = list(range(1, 13))
    reverse_sorted_data = list(range(12, 0, -1))

    run_test_case("Random Data", random_data)
    run_test_case("Already Sorted Data", sorted_data)
    run_test_case("Reverse-Sorted Data", reverse_sorted_data)

    print("\nAI-Generated Comparison:")
    print("Quick Sort Complexity:")
    print("- Best Case: O(n log n)")
    print("- Average Case: O(n log n)")
    print("- Worst Case: O(n^2) (bad pivot splits)")

    print("Merge Sort Complexity:")
    print("- Best Case: O(n log n)")
    print("- Average Case: O(n log n)")
    print("- Worst Case: O(n log n)")

    print("\nPractical Scenario Guidance:")
    print("- Prefer Quick Sort for in-memory data when average performance is the priority.")
    print("- Prefer Merge Sort when stable and predictable O(n log n) behavior is needed.")
    print("- For already sorted or reverse-sorted inputs, Merge Sort remains consistently reliable.")

```

Input & Output:

```

Random Data:
Input: [43, 7, 37, 48, 1, 49, 3, 15, 18, 12, 31, 35]
Quick Sort: [1, 3, 7, 12, 15, 18, 31, 35, 37, 43, 48, 49]
Merge Sort: [1, 3, 7, 12, 15, 18, 31, 35, 37, 43, 48, 49]
Same Output: True

Already Sorted Data:
Input: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
Quick Sort: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
Merge Sort: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
Same Output: True

Reverse-Sorted Data:
Input: [12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
Quick Sort: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
Merge Sort: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
Same Output: True

AI-Generated Comparison:
Quick Sort Complexity:
- Best Case: O(n log n)
- Average Case: O(n log n)
- Worst Case: O(n^2) (bad pivot splits)
Merge Sort Complexity:
- Best Case: O(n log n)
- Average Case: O(n log n)
- Worst Case: O(n log n)

Practical Scenario Guidance:
- Prefer Quick Sort for in-memory data when average performance is the priority.
- Prefer Merge Sort when stable and predictable O(n log n) behavior is needed.
- For already sorted or reverse-sorted inputs, Merge Sort remains consistently reliable.

```

Task 5: Optimizing a Duplicate Detection Algorithm

Code:

```
def find_duplicates_bruteforce(user_ids):
    duplicates = set()
    n = len(user_ids)

    # Compare every element with every later element.
    for i in range(n):
        for j in range(i + 1, n):
            if user_ids[i] == user_ids[j]:
                duplicates.add(user_ids[i])

    return duplicates

def find_duplicates_optimized(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

if __name__ == "__main__":
    # Sample user IDs (contains duplicates)
    sample_user_ids = [
        1021, 1045, 1021, 1099, 1102, 1045, 1200, 1301, 1200, 1407, 1503
    ]

    print("Input User IDs:", sample_user_ids)

    brute_force_duplicates = find_duplicates_bruteforce(sample_user_ids)
    optimized_duplicates = find_duplicates_optimized(sample_user_ids)

    print("\nBrute-force Duplicates:", sorted(brute_force_duplicates))
    print("Optimized Duplicates:", sorted(optimized_duplicates))

    print("\nAI-Assisted Analysis:")
    print("- Brute-force uses nested loops, so checks grow roughly as  $n^2$ .")
    print("- Optimized approach uses a set, reducing repeated comparisons.")
    print("- Both return the same duplicates, but optimized scales much better.")

    print("\nComplexity Comparison:")
    print("- Brute-force:  $O(n^2)$ ")
    print("- Optimized (set-based):  $O(n)$  average")

    print("\nConceptual Behavior on Large Inputs:")
    print("- If  $n$  doubles, brute-force work grows about 4x.")
    print("- If  $n$  doubles, optimized work grows about 2x.")
    print("- For very large datasets, set-based detection is significantly faster.")
```

Input & Output:

```
Input User IDs: [1021, 1045, 1021, 1099, 1102, 1045, 1200, 1301, 1200, 1407, 1503]
```

```
Brute-force Duplicates: [1021, 1045, 1200]
```

```
Optimized Duplicates: [1021, 1045, 1200]
```

AI-Assisted Analysis:

- Brute-force uses nested loops, so checks grow roughly as n^2 .
- Optimized approach uses a set, reducing repeated comparisons.
- Both return the same duplicates, but optimized scales much better.

Complexity Comparison:

- Brute-force: $O(n^2)$
- Optimized (set-based): $O(n)$ average

Conceptual Behavior on Large Inputs:

- If n doubles, brute-force work grows about 4x.
- If n doubles, optimized work grows about 2x.
- For very large datasets, set-based detection is significantly faster.