ASSIGNMENT_12.4

Name : Chirra Jyothika

Ht.no:2303A51280

Batch:05

Task 1: Bubble Sort for Ranking Exam Scores

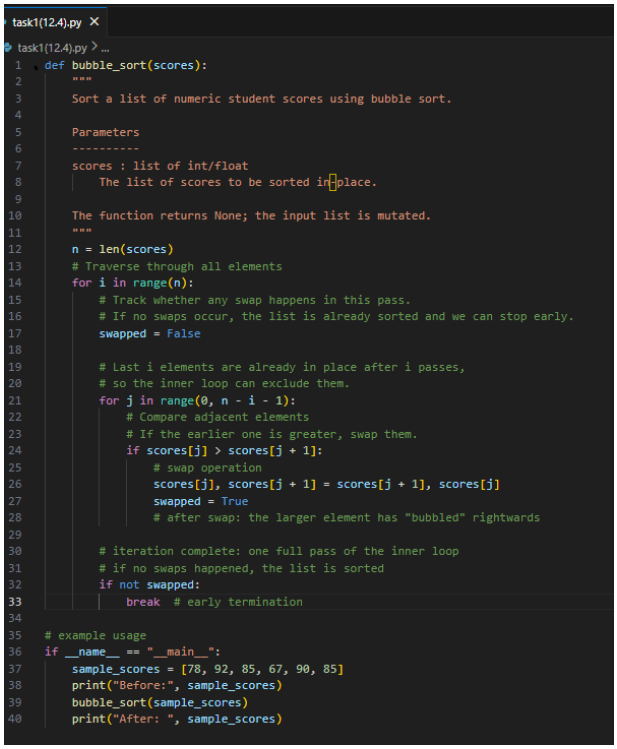
Scenario:

You are working on a college result processing system where a small list of student scores needs to be sorted after every internal assessment.

PROMPT:

Imagine you are working on a college result processing system where a small lift of student scores needs to be sorted after very internal assessment so ,now implement Bubble Sort in python to sort a list of student score and also explain in comments about comparisons, swaps and iteration process and identify early-termination conditions when the list becomes sorted and provide a brief time complexity analysis.

CODE:

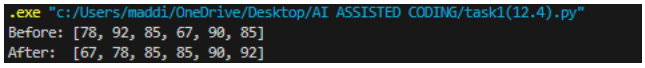


task1(12.4).py

task1(12.4).py > ...

```python
def bubble_sort(scores):
    """
    Sort a list of numeric student scores using bubble sort.

    Parameters
    ----------
    scores : list of int/float
        The list of scores to be sorted in place.

    The function returns None; the input list is mutated.
    """
    n = len(scores)
    # Traverse through all elements
    for i in range(n):
        # Track whether any swap happens in this pass.
        # If no swaps occur, the list is already sorted and we can stop early.
        swapped = False

        # Last i elements are already in place after i passes,
        # so the inner loop can exclude them.
        for j in range(0, n - i - 1):
            # Compare adjacent elements
            # If the earlier one is greater, swap them.
            if scores[j] > scores[j + 1]:
                # swap operation
                scores[j], scores[j + 1] = scores[j + 1], scores[j]
                swapped = True
                # after swap: the larger element has "bubbled" rightwards

        # iteration complete: one full pass of the inner loop
        # if no swaps happened, the list is sorted
        if not swapped:
            break  # early termination

# example usage
if __name__ == "__main__":
    sample_scores = [78, 92, 85, 67, 90, 85]
    print("Before:", sample_scores)
    bubble_sort(sample_scores)
    print("After: ", sample_scores)
```

OUTPUT:



EXPLANATION:

Bubble sort is simple and effective for very small lists (like a handful of scores), but becomes inefficient on larger datasets.
For a college result processor doing only a “small lift” of scores, it’s a clear and easy choice.

Case	Complexity
Best (already sorted)	O(n)
Average	$O(n^2)$
Worst	$O(n^2)$

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:

Imagine you are maintaining an attendance system where student roll numbers are already almost sorted with only a few late updates. Now implement and start with a bubble sort in python and suggest a more suitable sorting algorithm and generate an insertion sort implementation and also explain why insertion sort performs better on nearly sorted data now compare execution behavior on nearly sorted input.

CODE:

```python
def bubble_sort(rolls):
    """
    Classic bubble sort. Good for teaching, but even with early termination
    it still does a full pass if only the last element is misplaced.
    """
    n = len(rolls)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):
            # compare adjacent entries
            if rolls[j] > rolls[j + 1]:
                # swap them - the larger element "bubbles" rightward
                rolls[j], rolls[j + 1] = rolls[j + 1], rolls[j]
                swapped = True
        if not swapped:
            # nothing moved; list is sorted; stop early
            break


def insertion_sort(rolls):
    """
    Insertion sort: build the sorted prefix one element at a time.
    Each new element is compared backwards and shifted into its
    correct position.

    This is O(n^2) in the worst case, but on nearly-sorted input
    it runs in about O(n + k) where k is the number of misplaced
    elements (or inversions).
    """
    for i in range(1, len(rolls)):
        key = rolls[i]
        j = i - 1
        # shift elements of the sorted prefix that are greater than key
        while j >= 0 and rolls[j] > key:
            rolls[j + 1] = rolls[j]   # move larger element right
            j -= 1
        rolls[j + 1] = key            # insert key into correct slot
```

```python
if __name__ == "__main__":
    import random, time

    # nearly sorted list: only a few elements randomly swapped
    base = list(range(1, 101))
    arr1 = base.copy()
    arr2 = base.copy()
    # introduce a handful of displacements
    for _ in range(5):
        i, j = random.sample(range(100), 2)
        arr1[i], arr1[j] = arr1[j], arr1[i]
        arr2[i], arr2[j] = arr2[j], arr2[i]

    print("Initial (first 20):", arr1[:20])

    t0 = time.perf_counter()
    bubble_sort(arr1)
    t1 = time.perf_counter()
    print("Bubble sorted in {:.6f}s".format(t1 - t0))

    t0 = time.perf_counter()
    insertion_sort(arr2)
    t1 = time.perf_counter()
    print("Insertion sorted in {:.6f}s".format(t1 - t0))

    # verify both produce the same result
    assert arr1 == arr2 == base
```

OUTPUT:

```
Desktop/AI ASSISTED CODING/task2(12.4).py"
Initial (first 20): [1, 89, 3, 4, 5, 6, 7, 73, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
Bubble sorted in 0.000357s
Insertion sorted in 0.000044s
```

EXPLANATION:

- Bubble sort repeatedly scans adjacent pairs across the whole list; even with early termination, a single misplaced element near the start forces a full pass to discover it.
- Insertion sort takes each new element and places it into the already-sorted prefix by shifting only as far as needed. On a list that's almost sorted most of its inner loop comparisons immediately fail (because the element is already ≥ its predecessor), so work is essentially linear.
- Early-termination for bubble sort stops when a pass has no

swaps; however, that still requires comparing every adjacent pair up to that point.

Insertion sort's implicit early exit happens per element.

Algorithm	Work per element	Behaviour on our sample
Bubble Sort	**~n comparisons per pass; may do several passes**	**did one full pass discovering no swaps, then stopped**
Insertion Sort	**~1 comparison per element when sorted**	**inspected each element once and inserted immediately**

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:

Imagine you are developing a student information portal where user search for student records by roll number now requirements are implement Linear search for unsorted student data and Binary Search for sorted student data using docstrings for explaining parameters and return values and explain when binary search is applicable and also Highlight performance differences between the two searches

CODE:

```python
def linear_search(records, target):
    """
    Scan an unsorted sequence of student records for a given roll number.

    Parameters
    ----------
    records : iterable of (roll, info)
        Any sequence (list, tuple, etc.) where each element is a pair
        whose first component is the roll number.
    target : comparable
        The roll number we want to find.

    Returns
    -------
    int
        The index of the first matching record, or -1 if not found.
    """
    for idx, rec in enumerate(records):
        if rec[0] == target:
            return idx
    return -1


def binary_search(sorted_records, target):
    """
    Perform binary search on a list of records sorted by roll number.

    Parameters
    ----------
    sorted_records : list of (roll, info)
        Must be sorted in ascending order by the roll field.
    target : comparable
        The roll number to locate.

    Returns
    -------
    int
        The index of a matching record, or -1 if the roll number is absent.

    Notes
    -----
    This algorithm assumes the input list is already sorted. If the list
    is not sorted, the result is meaningless.
    """
    lo, hi = 0, len(sorted_records) - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        mid_roll = sorted_records[mid][0]
        if mid_roll == target:
            return mid
        elif mid_roll < target:
            lo = mid + 1
        else:
            hi = mid - 1
    return -1


if __name__ == "__main__":
    students = [
        (102, "Alice"),
        (215, "Bob"),
        (187, "Charlie"),
        (199, "Dana"),
    ]
    print("unsorted search:", linear_search(students, 199))

    sorted_students = sorted(students, key=lambda r: r[0])
    print("binary search:", binary_search(sorted_students, 199))
```

OUTPUT:

```
 ASSISTED CODING/task3(12.4).py"
unsorted search: 3
binary search: 2
```

EXPLANATION:

- Only on sorted datasets. If you try binary search on unsorted data, you cannot guarantee that skipping half the list won't skip the target.
- Sorting itself costs (O(n\log n)), so binary search is most worthwhile when you perform many lookups on a list that's maintained in sorted order (e.g. via insertion sort or by keeping it sorted as records are added).

Performance Comparison

Algorithm	Precondition	Worst-case Time	Typical Use-case
Linear Search	None (unsorted OK)	O(n)	New or unsorted datasets; one-off search
Binary Search	List must be sorted	O(log n)	Repeated lookups on sorted data

linear search is simple and works everywhere but scans through the entire list, while binary search exploits ordering to cut the search space in half at each step – dramatically faster on large, sorted arrays.

Task 4: Choosing Between Quick Sort and Merge Sort for Data Processing

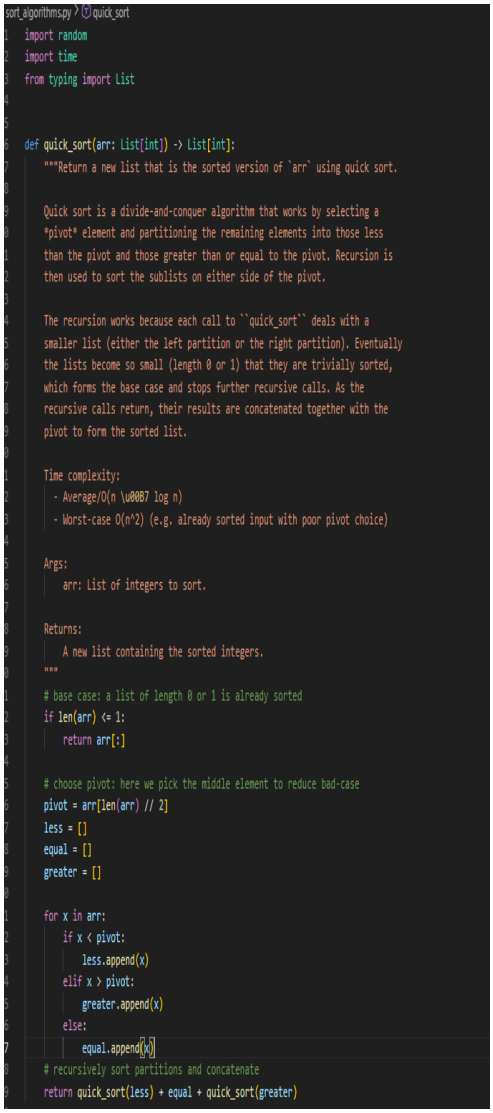
Scenario:

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).

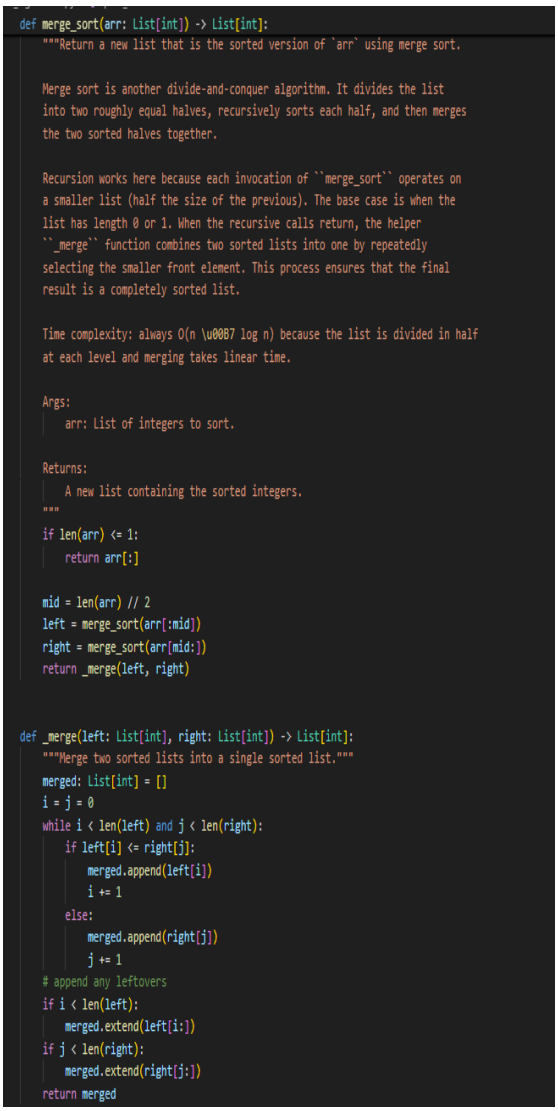
PROMPT:

Imagine 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). Now write partially written recursive functions for Quick sort and Merge Sort now complete the recursive logic and add meaningful docstrings and also explain how recursion works in each algorithm and test both algorithms on Random data ,Sorted data, Reverse -sorted data and also compare time complexities.

CODE:



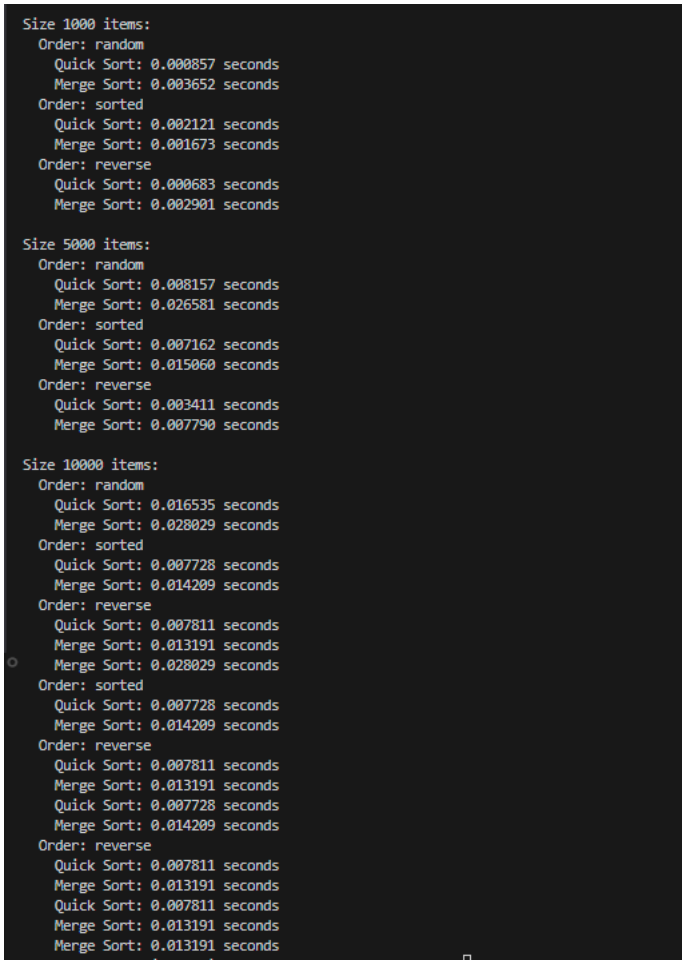
```python
import random
import time
from typing import List


def quick_sort(arr: List[int]) -> List[int]:
    """Return a new list that is the sorted version of `arr` using quick sort.

    Quick sort is a divide-and-conquer algorithm that works by selecting a
    *pivot* element and partitioning the remaining elements into those less
    than the pivot and those greater than or equal to the pivot. Recursion is
    then used to sort the sublists on either side of the pivot.

    The recursion works because each call to ``quick_sort`` deals with a
    smaller list (either the left partition or the right partition). Eventually
    the lists become so small (length 0 or 1) that they are trivially sorted,
    which forms the base case and stops further recursive calls. As the
    recursive calls return, their results are concatenated together with the
    pivot to form the sorted list.

    Time complexity:
      - Average/O(n \u00B7 log n)
      - Worst-case O(n^2) (e.g. already sorted input with poor pivot choice)

    Args:
        arr: List of integers to sort.

    Returns:
        A new list containing the sorted integers.
    """
    # base case: a list of length 0 or 1 is already sorted
    if len(arr) <= 1:
        return arr[:]

    # choose pivot: here we pick the middle element to reduce bad-case
    pivot = arr[len(arr) // 2]
    less = []
    equal = []
    greater = []

    for x in arr:
        if x < pivot:
            less.append(x)
        elif x > pivot:
            greater.append(x)
        else:
            equal.append(x)
    # recursively sort partitions and concatenate
    return quick_sort(less) + equal + quick_sort(greater)
```



```python
def merge_sort(arr: List[int]) -> List[int]:
    """Return a new list that is the sorted version of `arr` using merge sort.

    Merge sort is another divide-and-conquer algorithm. It divides the list
    into two roughly equal halves, recursively sorts each half, and then merges
    the two sorted halves together.

    Recursion works here because each invocation of ``merge_sort`` operates on
    a smaller list (half the size of the previous). The base case is when the
    list has length 0 or 1. When the recursive calls return, the helper
    ``_merge`` function combines two sorted lists into one by repeatedly
    selecting the smaller front element. This process ensures that the final
    result is a completely sorted list.

    Time complexity: always O(n \u00B7 log n) because the list is divided in half
    at each level and merging takes linear time.

    Args:
        arr: List of integers to sort.

    Returns:
        A new list containing the sorted integers.
    """
    if len(arr) <= 1:
        return arr[:]

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return _merge(left, right)


def _merge(left: List[int], right: List[int]) -> List[int]:
    """Merge two sorted lists into a single sorted list."""
    merged: List[int] = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1
    # append any leftovers
    if i < len(left):
        merged.extend(left[i:])
    if j < len(right):
        merged.extend(right[j:])
    return merged
```

```python
# testing harness

def _generate_data(size: int, order: str) -> List[int]:
    """Return a list of the given size.

    order may be "random", "sorted", or "reverse".
    """
    if order == "random":
        return [random.randint(0, size * 10) for _ in range(size)]
    elif order == "sorted":
        return list(range(size))
    elif order == "reverse":
        return list(range(size, 0, -1))
    else:
        raise ValueError("order must be random/sorted/reverse")


def _time_algorithm(alg, data: List[int]) -> float:
    start = time.perf_counter()
    alg(data)
    end = time.perf_counter()
    return end - start


def run_tests():
    sizes = [1000, 5000, 10000]
    orders = ["random", "sorted", "reverse"]
    algorithms = [(quick_sort, "Quick Sort"), (merge_sort, "Merge Sort")]

    for size in sizes:
        print(f"\nSize {size} items:")
        for order in orders:
            data = _generate_data(size, order)
            print(f"  Order: {order}")
            for alg, name in algorithms:
                # copy so the sort doesn't modify the original
                data_copy = data[:]
                t = _time_algorithm(alg, data_copy)
                print(f"    {name}: {t:.6f} seconds")


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

OUTPUT:



EXPLANATION:

Quick sort uses recursion by sorting partitions around a pivot,
and the depth of recursion is proportional to the list length (O(log n) on average).
Merge sort recurses on halves; each level of recursion halves the input size,
and merging the results produces the sorted list.

Algorithm	Average Time	Worst Case	Stable	Space
Quick Sort	$O(n \cdot \log n)$	$O(n^2)$	No	$O(\log n)$
Merge Sort	$O(n \cdot \log n)$	$O(n \cdot \log n)$	Yes	$O(n)$

Task 5: Optimizing a Duplicate Detection Algorithm

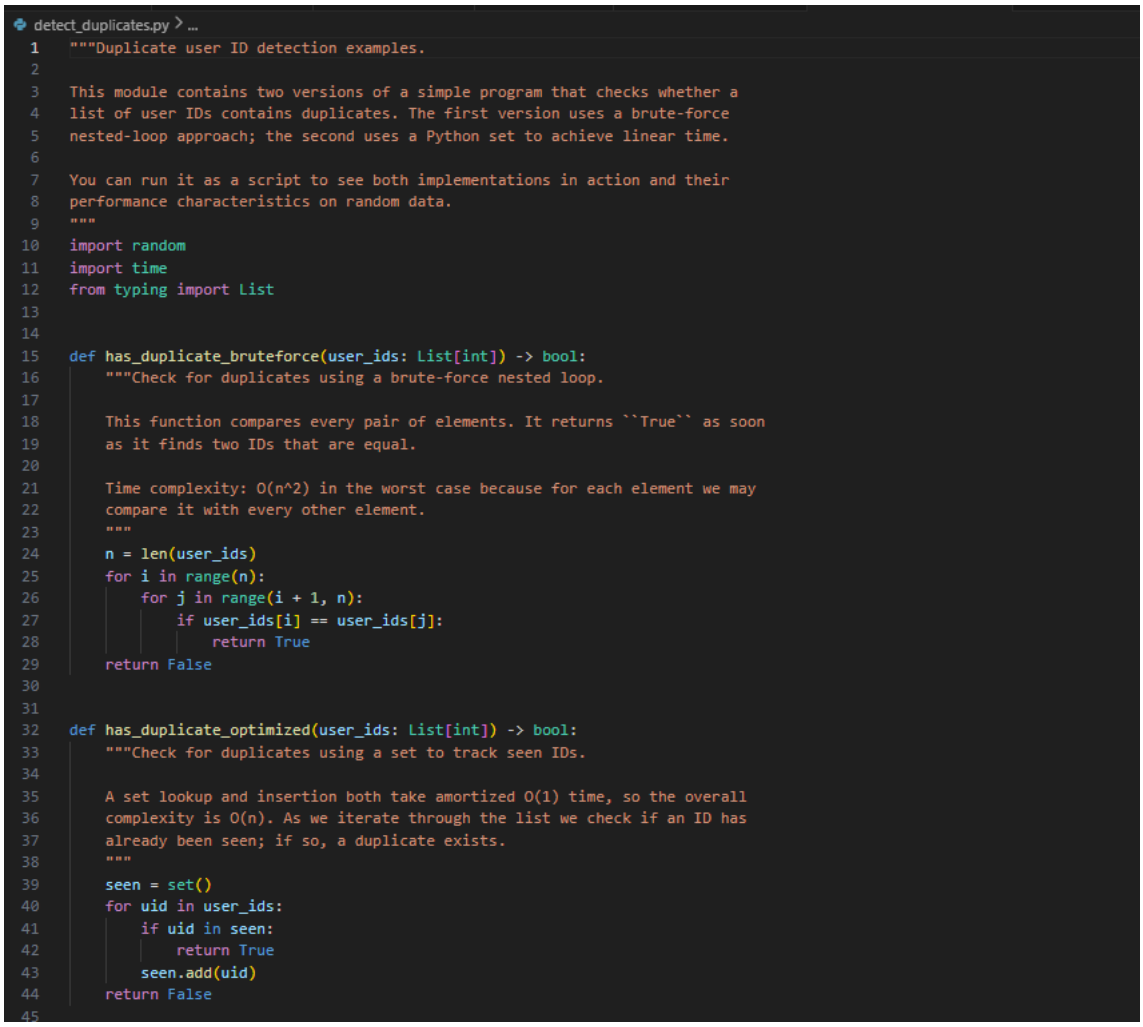
Scenario:

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

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
"""Duplicate user ID detection examples.

This module contains two versions of a simple program that checks whether a
list of user IDs contains duplicates. The first version uses a brute-force
nested-loop approach; the second uses a Python set to achieve linear time.

You can run it as a script to see both implementations in action and their
performance characteristics on random data.
"""
import random
import time
from typing import List


def has_duplicate_bruteforce(user_ids: List[int]) -> bool:
    """Check for duplicates using a brute-force nested loop.

    This function compares every pair of elements. It returns ``True`` as soon
    as it finds two IDs that are equal.

    Time complexity: O(n^2) in the worst case because for each element we may
    compare it with every other element.
    """
    n = len(user_ids)
    for i in range(n):
        for j in range(i + 1, n):
            if user_ids[i] == user_ids[j]:
                return True
    return False


def has_duplicate_optimized(user_ids: List[int]) -> bool:
    """Check for duplicates using a set to track seen IDs.

    A set lookup and insertion both take amortized O(1) time, so the overall
    complexity is O(n). As we iterate through the list we check if an ID has
    already been seen; if so, a duplicate exists.
    """
    seen = set()
    for uid in user_ids:
        if uid in seen:
            return True
        seen.add(uid)
    return False
```

```python
def _time_function(func, data: List[int]) -> float:
    start = time.perf_counter()
    result = func(data)
    end = time.perf_counter()
    return result, end - start


def main():
    size = 10000
    # create sample data with one duplicate at the end
    user_ids = list(range(size)) + [0]
    random.shuffle(user_ids)

    print("Brute-force check:")
    dup, t = _time_function(has_duplicate_bruteforce, user_ids)
    print(f"  duplicate? {dup} (took {t:.6f}s)")

    print("Optimized check:")
    dup, t = _time_function(has_duplicate_optimized, user_ids)
    print(f"  duplicate? {dup} (took {t:.6f}s)")

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

OUTPUT:

```
Brute-force check:
  duplicate? True (took 1.568182s)
Optimized check:
  duplicate? True (took 0.000783s)
```

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.