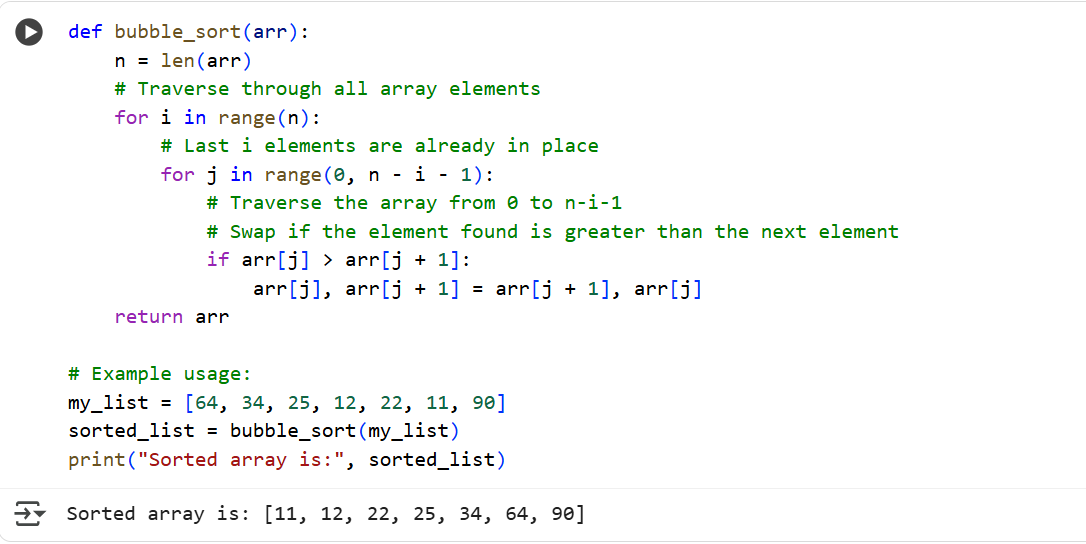
Assignment-12

Task-1

**Implementing Bubble Sort with AI Comments**

* **Task**: Write a Python implementation of **Bubble Sort**.
* **Instructions**:
  + Students implement Bubble Sort normally.
  + Ask AI to generate **inline comments explaining key logic** (like swapping, passes, and termination).
  + Request AI to provide **time complexity analysis**.
* **Expected Output**:

A Bubble Sort implementation with AI-generated explanatory comments and complexity analysis



Explanation:

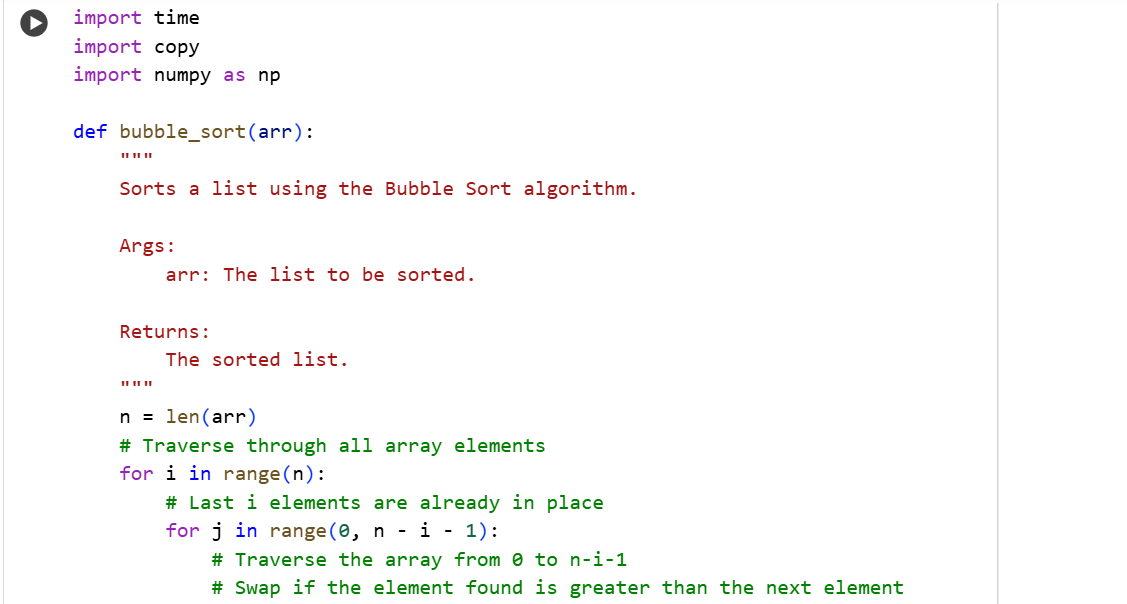
The code defines a function called bubble\_sort that takes a list arr as input.

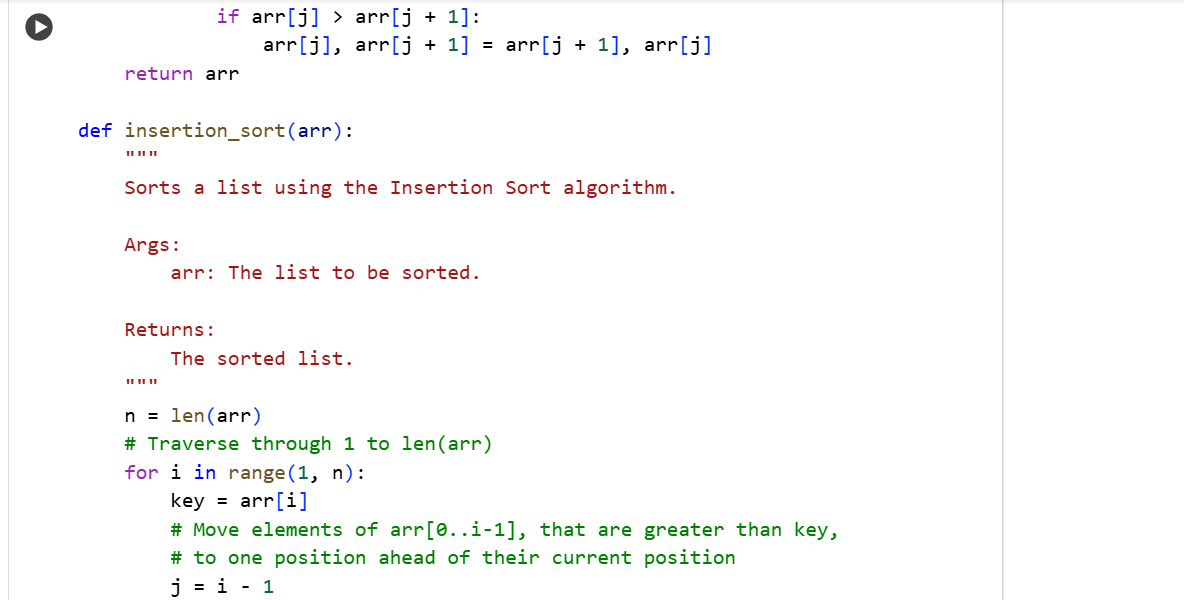
1. n = len(arr): This line gets the number of elements in the list and stores it in the variable n.
2. The outer loop for i in range(n): iterates n times. After each iteration of this outer loop, the largest unsorted element "bubbles up" to its correct position at the end of the unsorted portion of the list.
3. The inner loop for j in range(0, n - i - 1): iterates through the unsorted portion of the list. The range n - i - 1 decreases with each pass of the outer loop because the elements at the end of the list are already sorted.
4. if arr[j] > arr[j + 1]:: This line compares adjacent elements in the list.
5. arr[j], arr[j + 1] = arr[j + 1], arr[j]: If the element at index j is greater than the element at index j + 1, these two elements are swapped. This moves the larger element towards the end of the list.
6. return arr: After the loops complete, the function returns the sorted list.

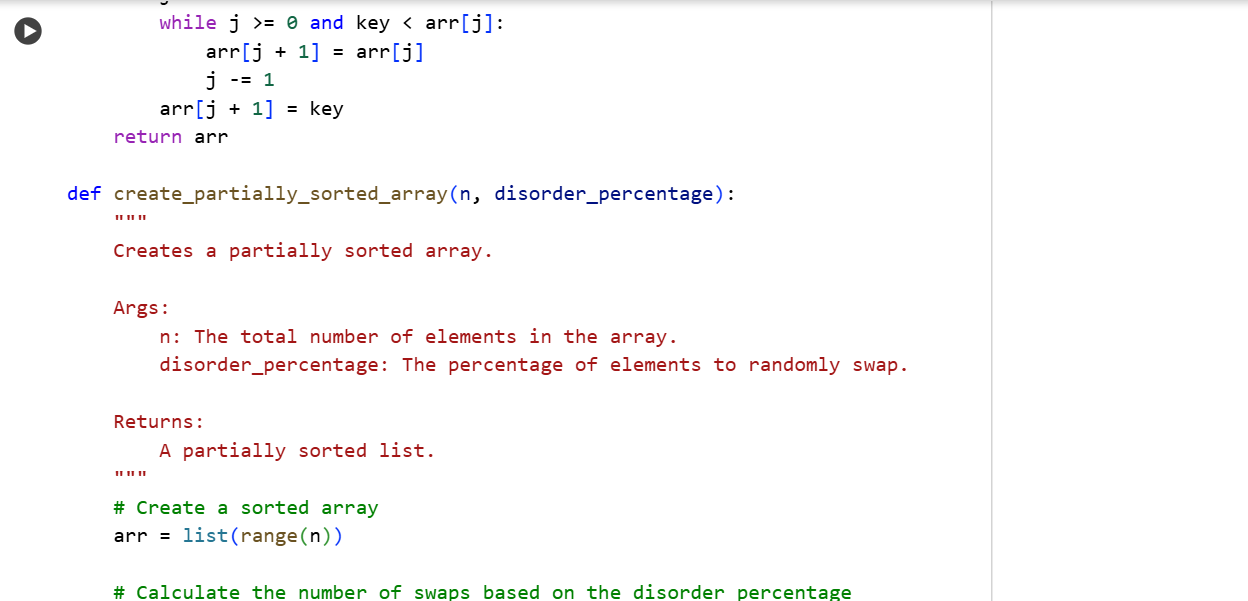
Task 2:

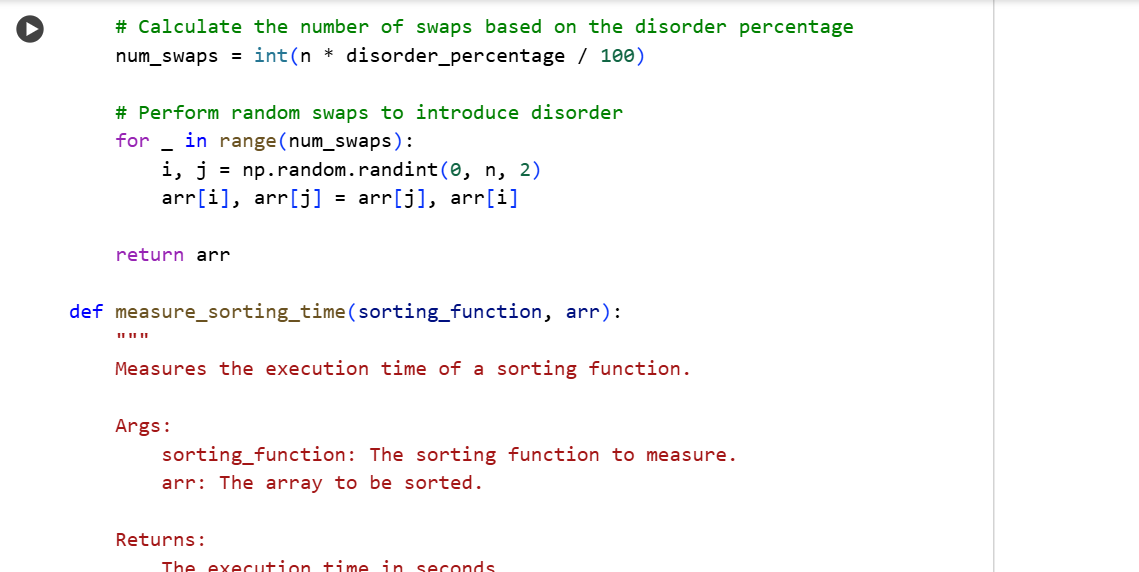
**Optimizing Bubble Sort → Insertion Sort**

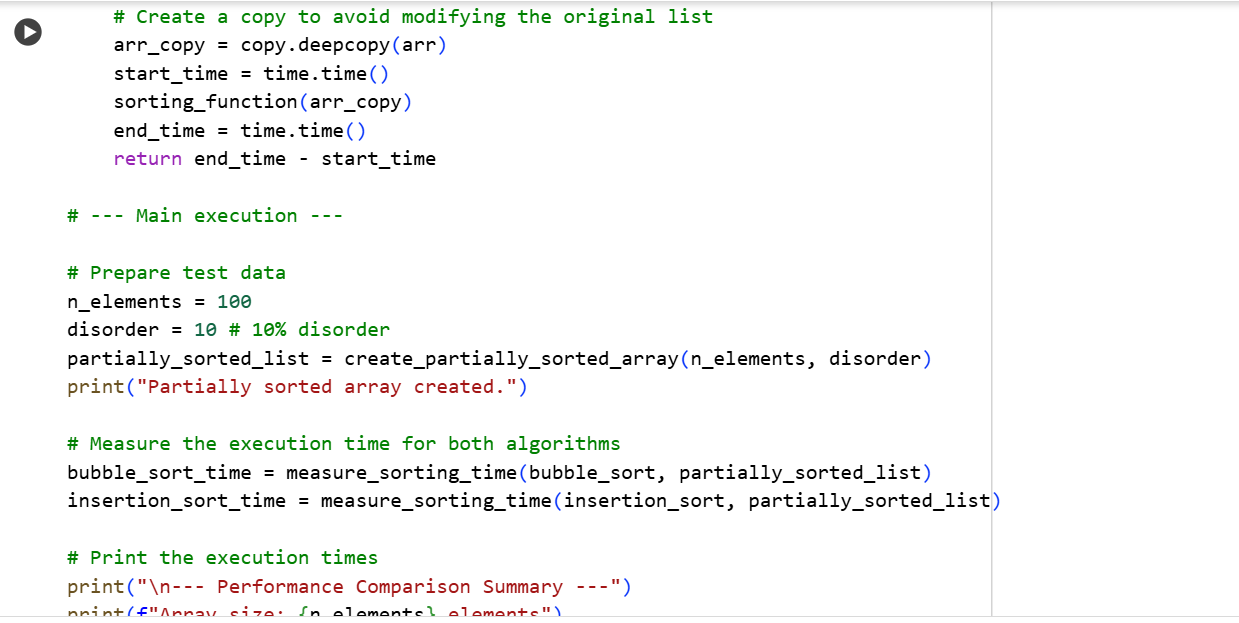
* **Task**: Provide Bubble Sort code to AI and ask it to suggest a **more efficient algorithm** for partially sorted arrays.
* **Instructions**:
  + Students implement Bubble Sort first.
  + Ask AI to suggest an alternative (Insertion Sort).
  + Compare performance on nearly sorted input.
* **Expected Output**:
  + Two codes (Bubble Sort + Insertion Sort).
  + AI explanation of why Insertion Sort is more efficient for partially sorted data.

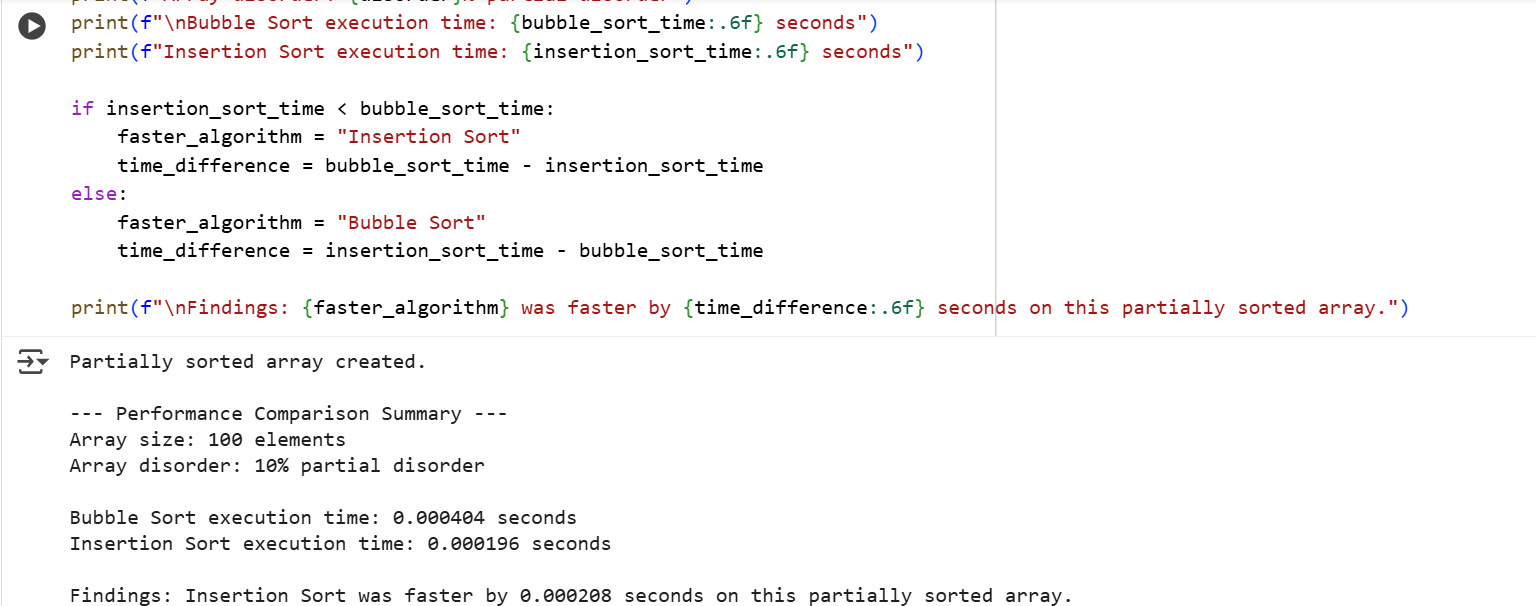












Explanation:

This cell contains the implementations of both Bubble Sort and Insertion Sort, along with helper functions to create partially sorted data and measure the execution time of the sorting algorithms.

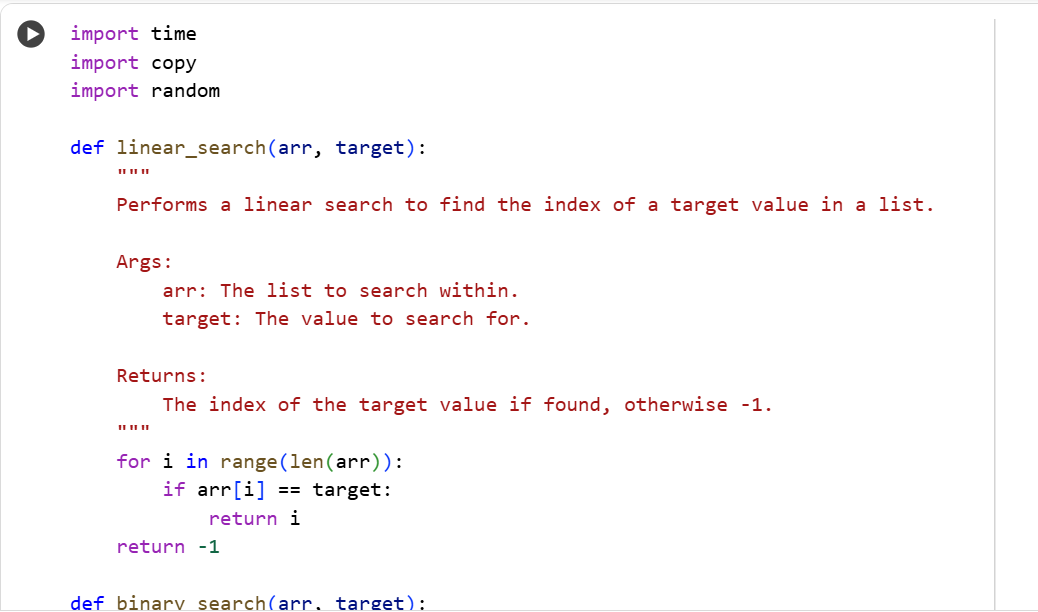
Here's a breakdown:

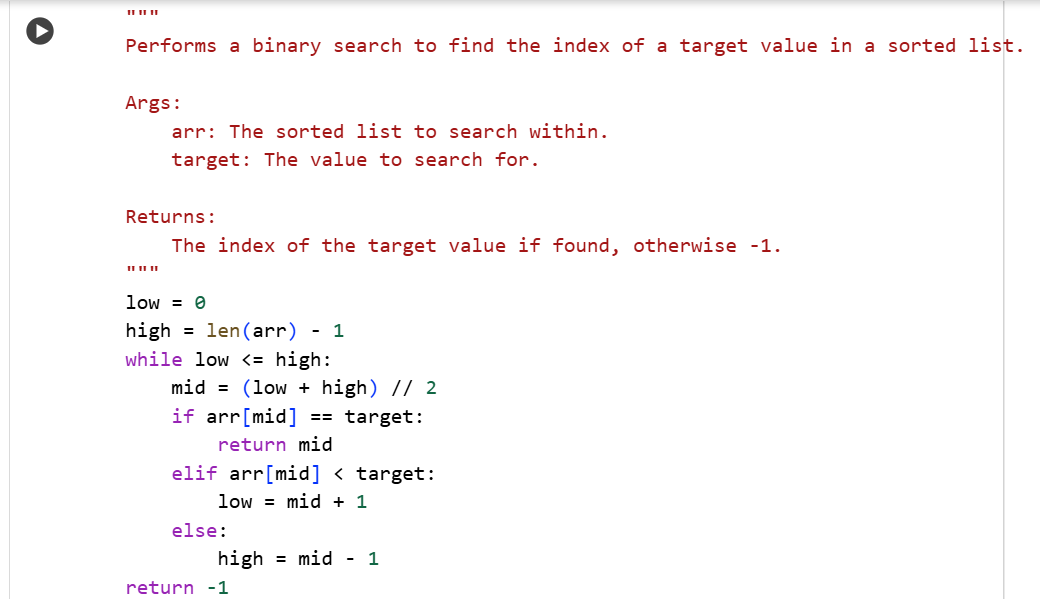
1. **bubble\_sort(arr) function:**
   * This function implements the Bubble Sort algorithm.
   * It iterates through the list multiple times, comparing adjacent elements and swapping them if they are in the wrong order.
   * The largest unsorted element "bubbles up" to its correct position in each pass.
   * It's a simple algorithm but generally not very efficient for larger datasets.
2. **insertion\_sort(arr) function:**
   * This function implements the Insertion Sort algorithm.
   * It builds the final sorted array one item at a time.
   * It iterates through the input list, takes one element at a time, and inserts it into its correct position within the already sorted portion of the list.
   * It is generally more efficient than Bubble Sort, especially for small or partially sorted arrays, because it adapts better to data that is already mostly in order.
3. **create\_partially\_sorted\_array(n, disorder\_percentage) function:**
   * This function generates a list of n elements that is mostly sorted but with a certain disorder\_percentage of elements randomly swapped to make it partially sorted.
   * It starts with a fully sorted list and then performs a calculated number of random swaps based on the desired disorder percentage.
4. **measure\_sorting\_time(sorting\_function, arr) function:**
   * This is a helper function used to measure how long a given sorting\_function takes to sort a specific arr.
   * It creates a copy of the input array (copy.deepcopy(arr)) to ensure the original array is not modified by the sorting process.
   * It records the start time before calling the sorting function and the end time after it finishes, then returns the difference.
5. **Main execution block:**
   * This part of the code sets up the test:
     + n\_elements = 100 and disorder = 10 define the size of the array and the percentage of disorder.
     + partially\_sorted\_list = create\_partially\_sorted\_array(n\_elements, disorder) creates the test data using the helper function.
     + bubble\_sort\_time = measure\_sorting\_time(bubble\_sort, partially\_sorted\_list) and insertion\_sort\_time = measure\_sorting\_time(insertion\_sort, partially\_sorted\_list) measure the time taken by each sorting algorithm on the same partially sorted list.
   * Finally, it prints a summary of the results, including the array characteristics, the execution times for both algorithms, and identifies which algorithm was faster for this specific test case

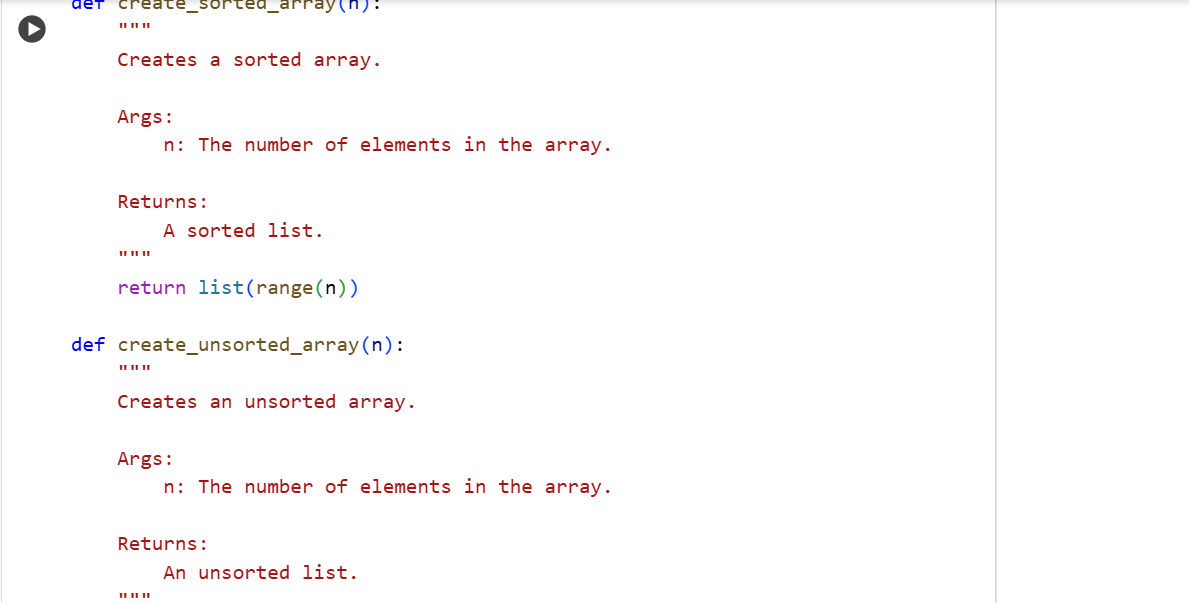
Task 3:

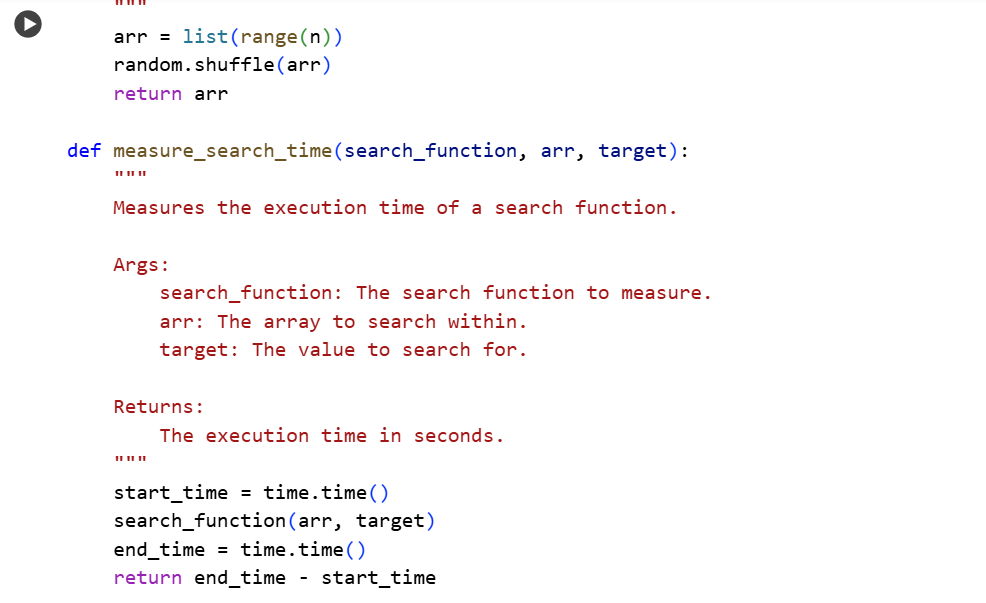
**Binary Search vs Linear Search**

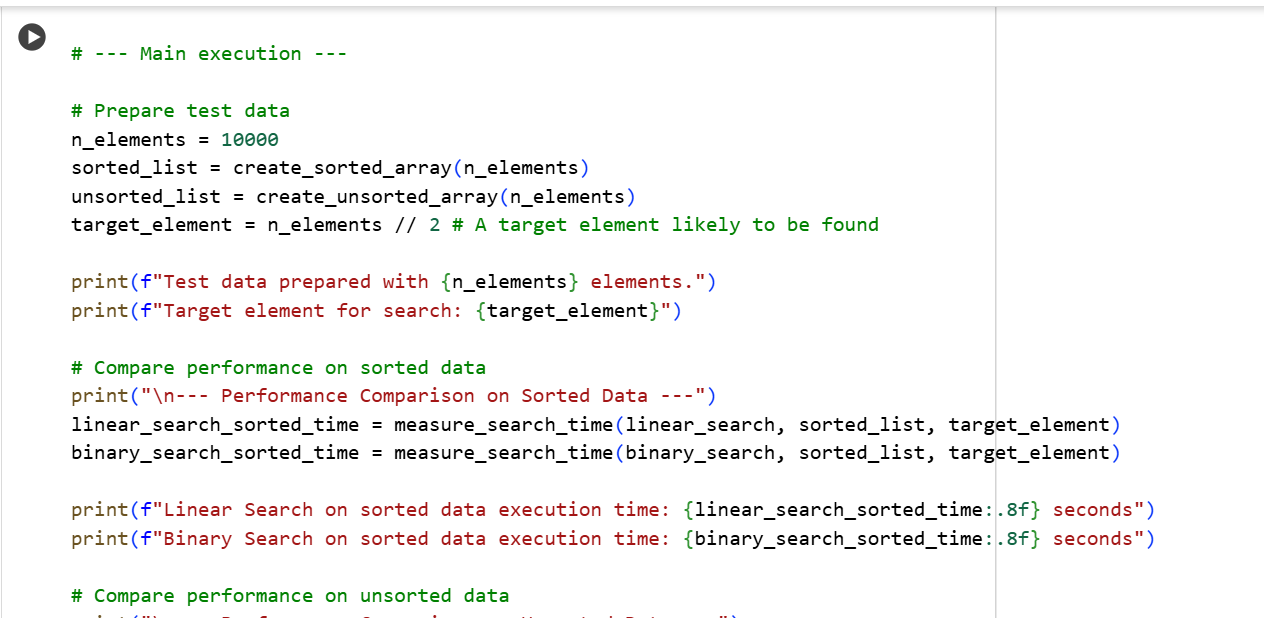
* **Task**: Implement both **Linear Search** and **Binary Search**.
* **Instructions**:
  + Use AI to generate docstrings and performance notes.
  + Test both algorithms on sorted and unsorted data.
  + Ask AI to explain when Binary Search is preferable.
* **Expected Output**:
  + Two implementations with docstrings.
  + A student observation table comparing performance (Linear vs Binary Search).

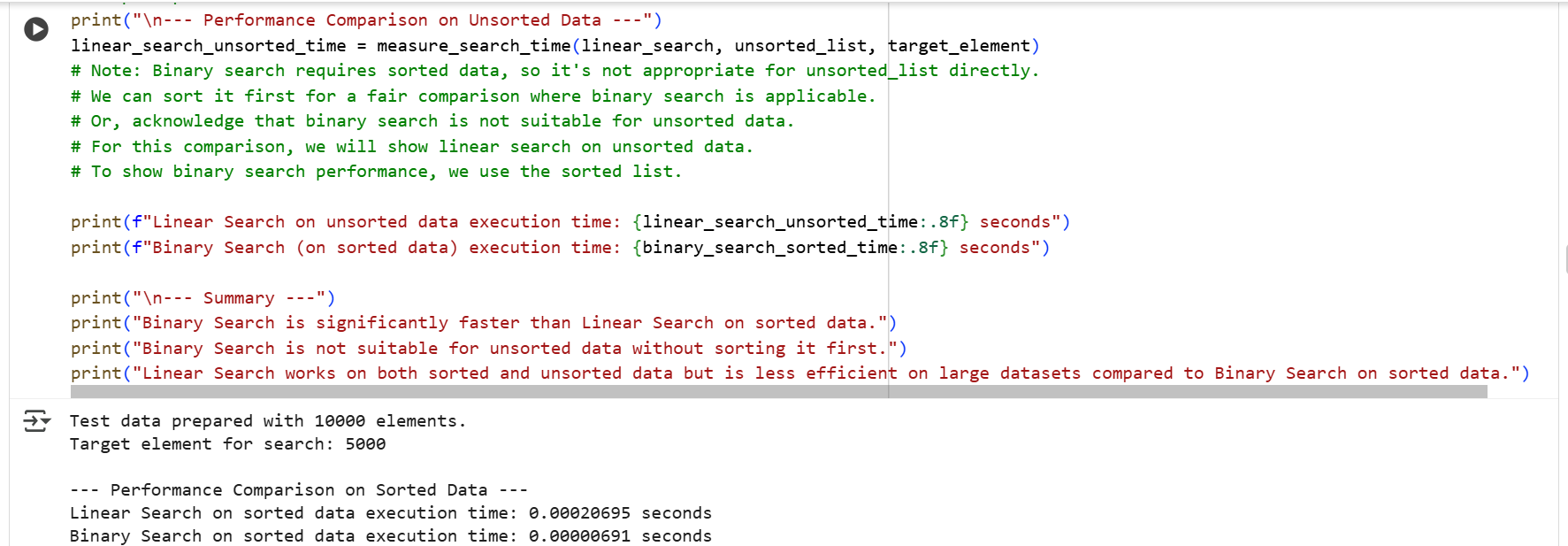


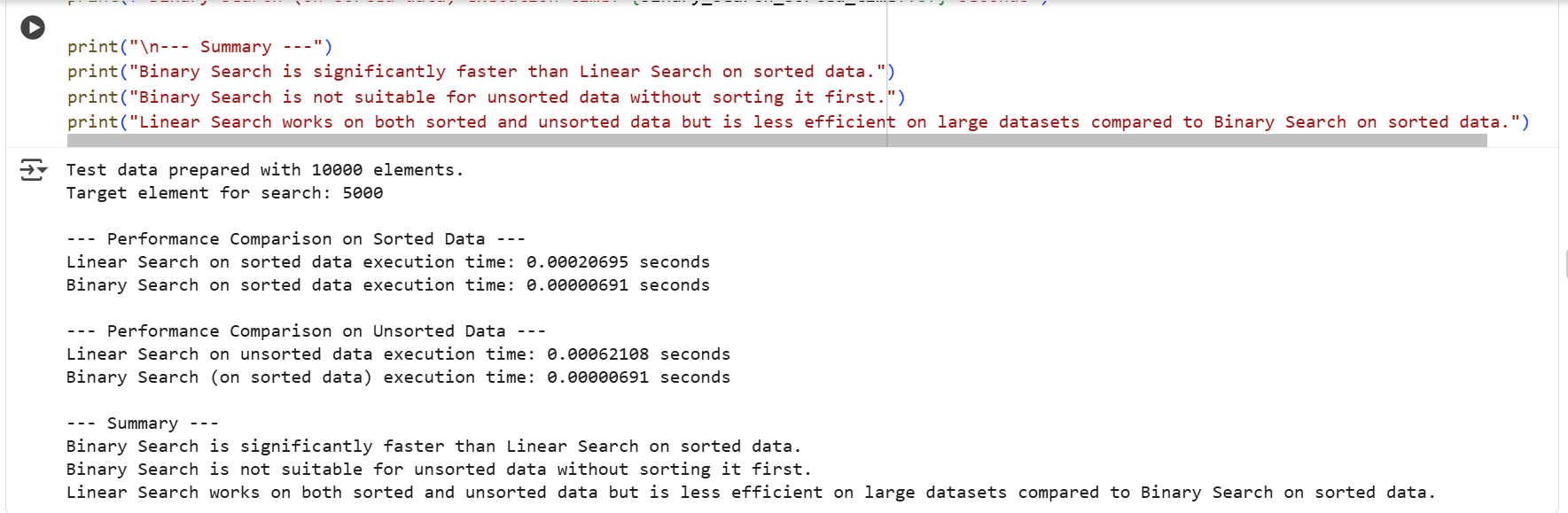












Explanation:

his cell contains the implementations of Linear Search and Binary Search, along with functions to create test data and measure the performance of the search algorithms.

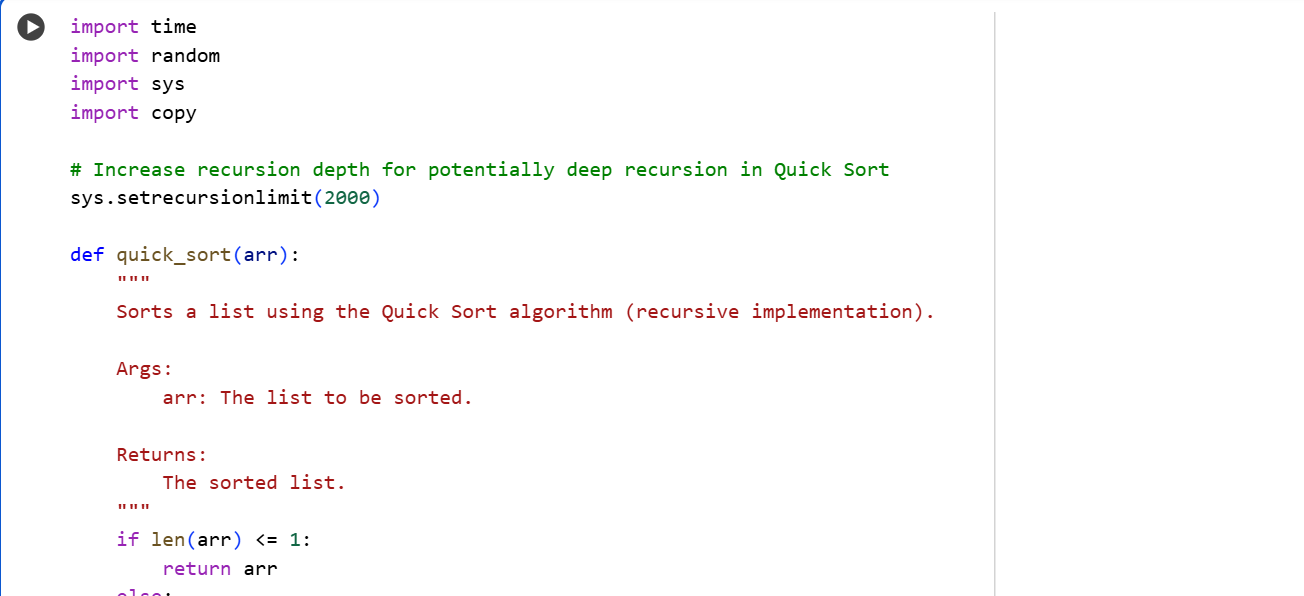
Here's a breakdown of the code:

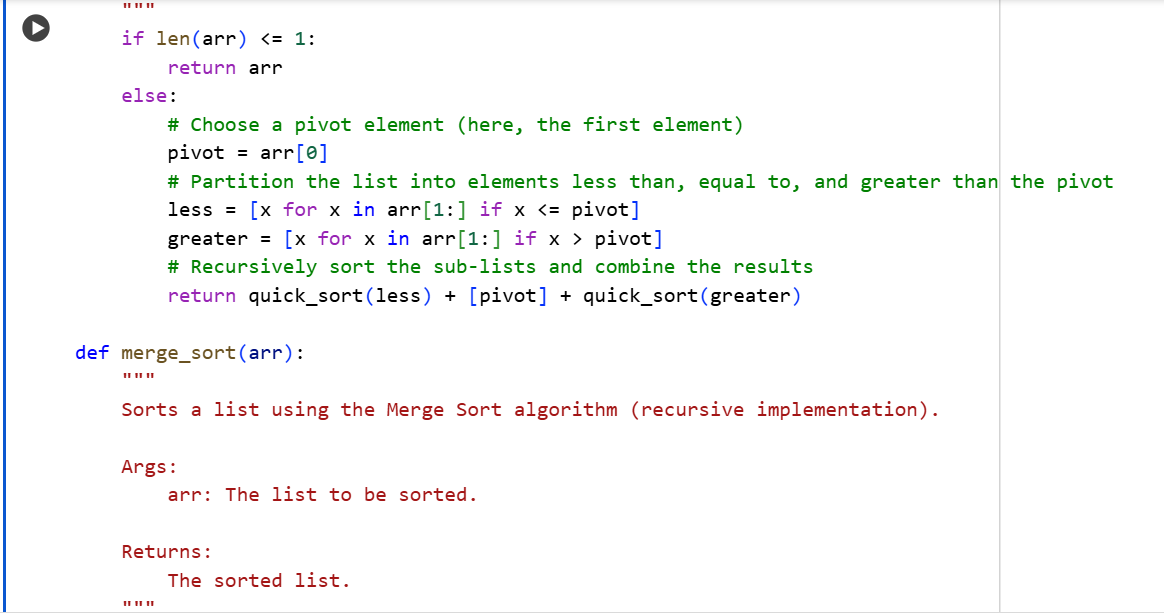
1. **linear\_search(arr, target) function:**
   * This function performs a simple linear search.
   * It iterates through each element of the input list arr one by one.
   * If it finds the target value, it returns the index of that value.
   * If the target is not found after checking all elements, it returns -1.
   * This algorithm works on both sorted and unsorted lists but can be slow for large lists.
2. **binary\_search(arr, target) function:**
   * This function performs a binary search. **Important:** This algorithm requires the input list arr to be sorted.
   * It works by repeatedly dividing the search interval in half.
   * It compares the target value with the middle element of the interval.
   * If the target matches the middle element, the index is returned.
   * If the target is less than the middle element, the search continues in the left half.
   * If the target is greater than the middle element, the search continues in the right half.
   * This process continues until the target is found or the interval is empty.
   * Binary search is significantly faster than linear search for large sorted lists.
3. **create\_sorted\_array(n) function:**
   * This function simply creates a sorted list containing integers from 0 up to n-1.
4. **create\_unsorted\_array(n) function:**
   * This function creates a list of integers from 0 to n-1 and then shuffles them randomly to produce an unsorted list.
5. **measure\_search\_time(search\_function, arr, target) function:**
   * This helper function takes a search function (either linear\_search or binary\_search), a list arr, and a target value.
   * It records the start time, calls the provided search function, records the end time, and returns the difference in seconds, representing the execution time.
6. **Main execution block:**
   * This part sets up and runs the performance comparison:
     + n\_elements = 10000 sets the size of the test lists.
     + sorted\_list = create\_sorted\_array(n\_elements) creates a sorted list.
     + unsorted\_list = create\_unsorted\_array(n\_elements) creates an unsorted list.
     + target\_element = n\_elements // 2 sets a target value to search for, which is likely to be found in the lists.
     + It then measures and prints the execution times for:
       - Linear Search on the sorted list.
       - Binary Search on the sorted list.
       - Linear Search on the unsorted list.
       - Binary Search is specifically noted as being run on the sorted list because it requires sorted data.
     + Finally, it provides a summary of the findings, highlighting the performance differences between the two algorithms on sorted and unsorted data and explaining when Binary Search is preferable (on sorted data).

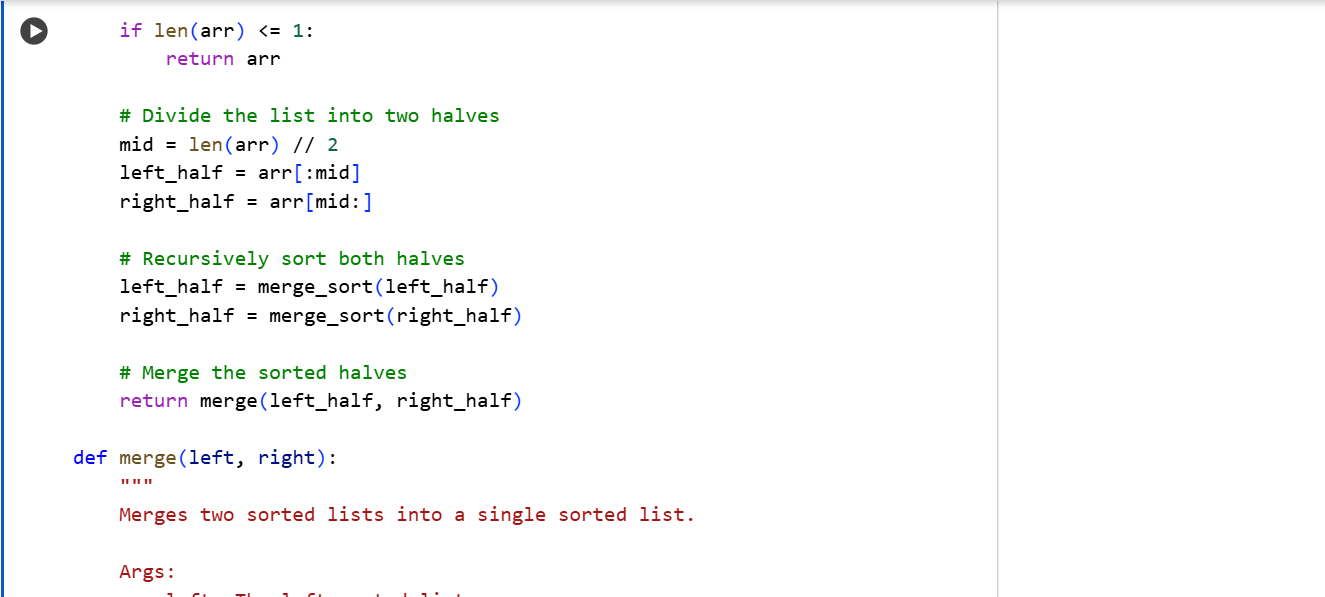
Task 4:

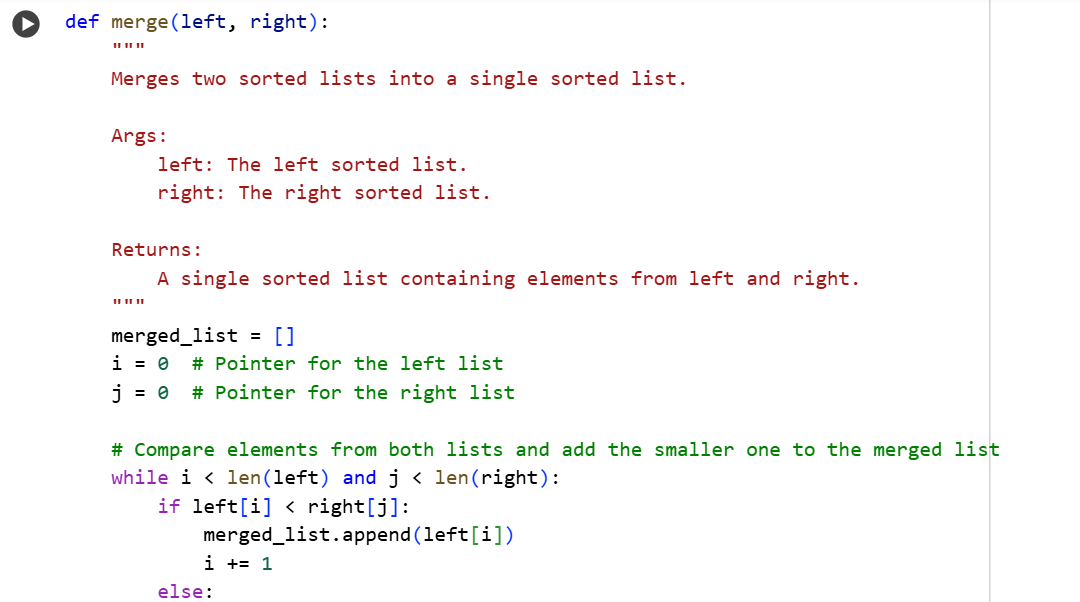
Quick Sort and Merge Sort Comparison

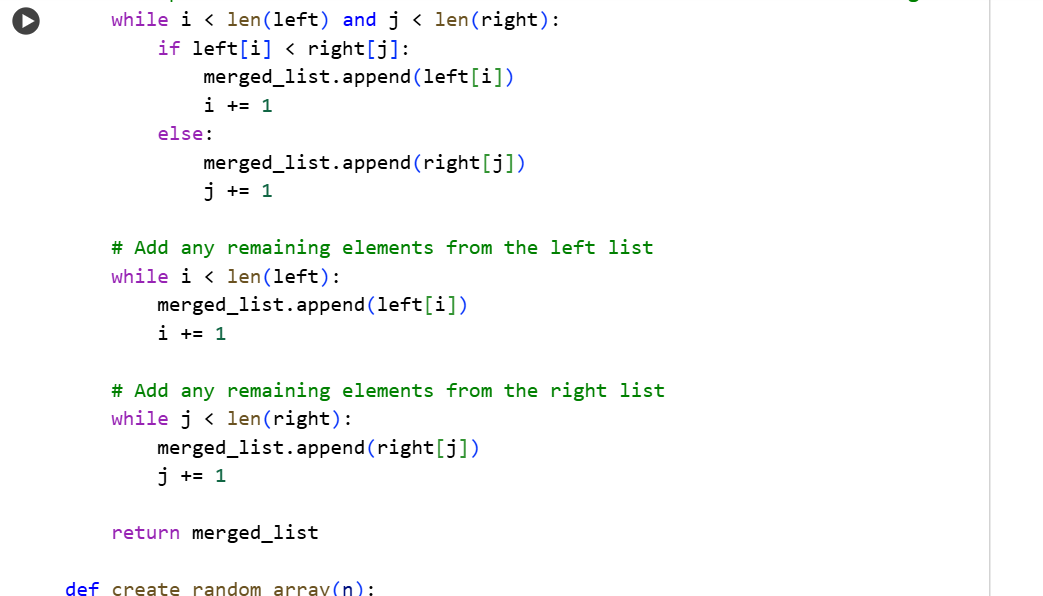
* **Task:** Implement Quick Sort and Merge Sort using recursion**.**
* **Instructions:**
  + Provide AI with partially completed functions for recursion.
  + Ask AI to complete the missing logic and add docstrings.
  + Compare both algorithms on random, sorted, and reverse-sorted lists.
* **Expected Output:**
  + Working Quick Sort and Merge Sort implementations.
  + AI-generated explanation of average, best, and worst-case complexities.

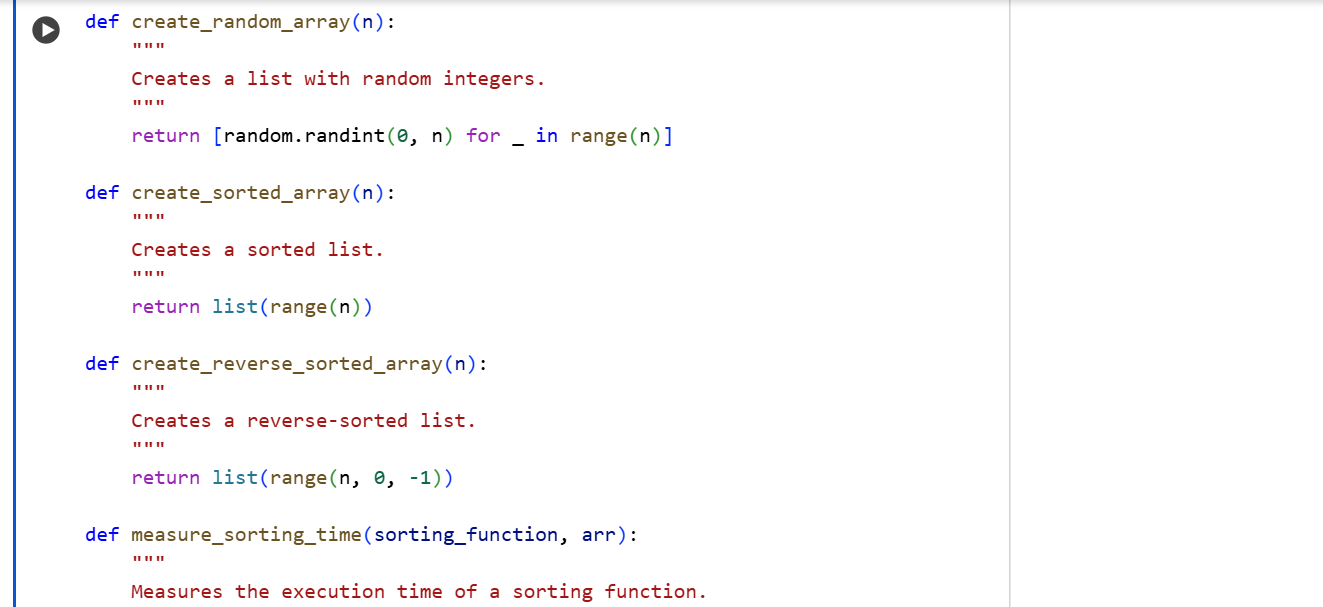


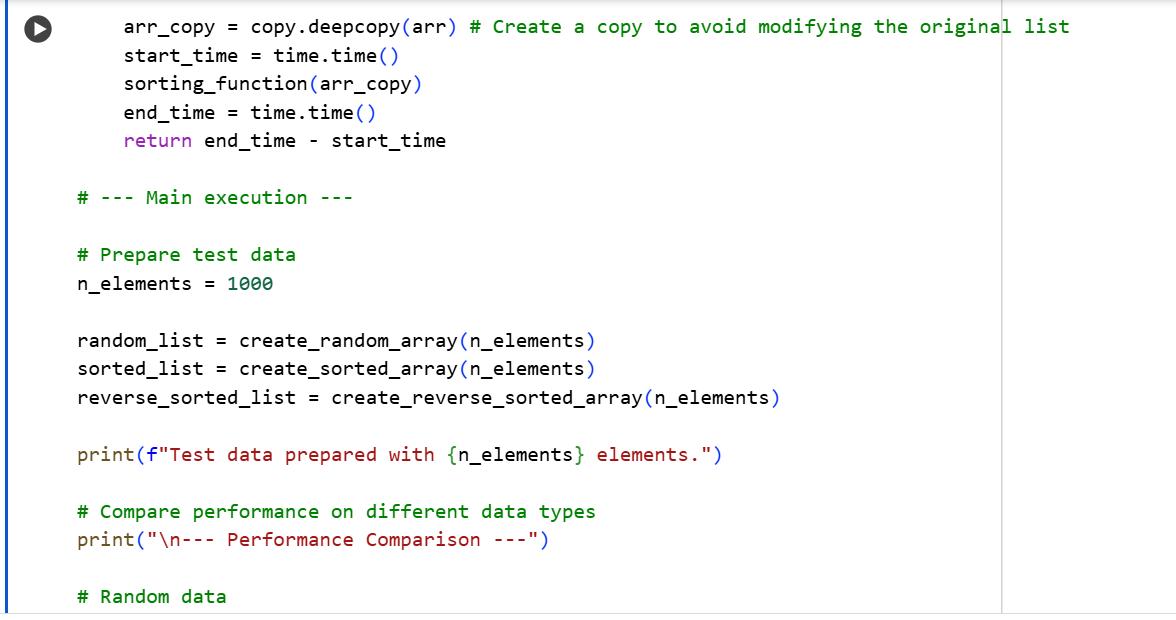


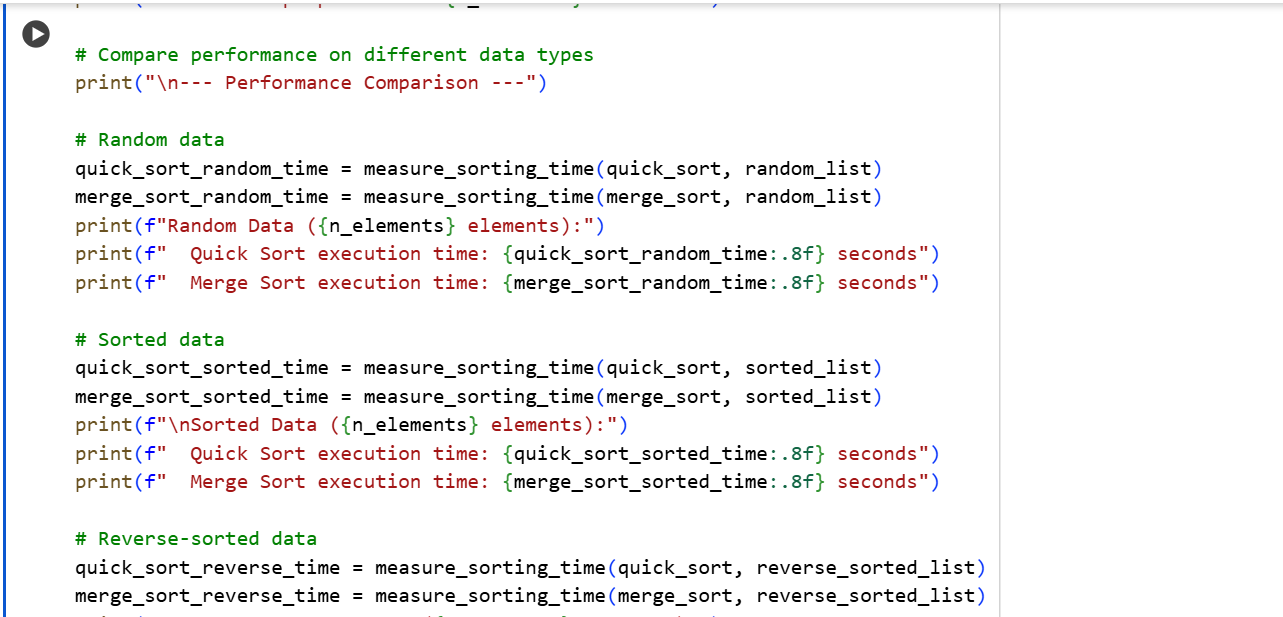


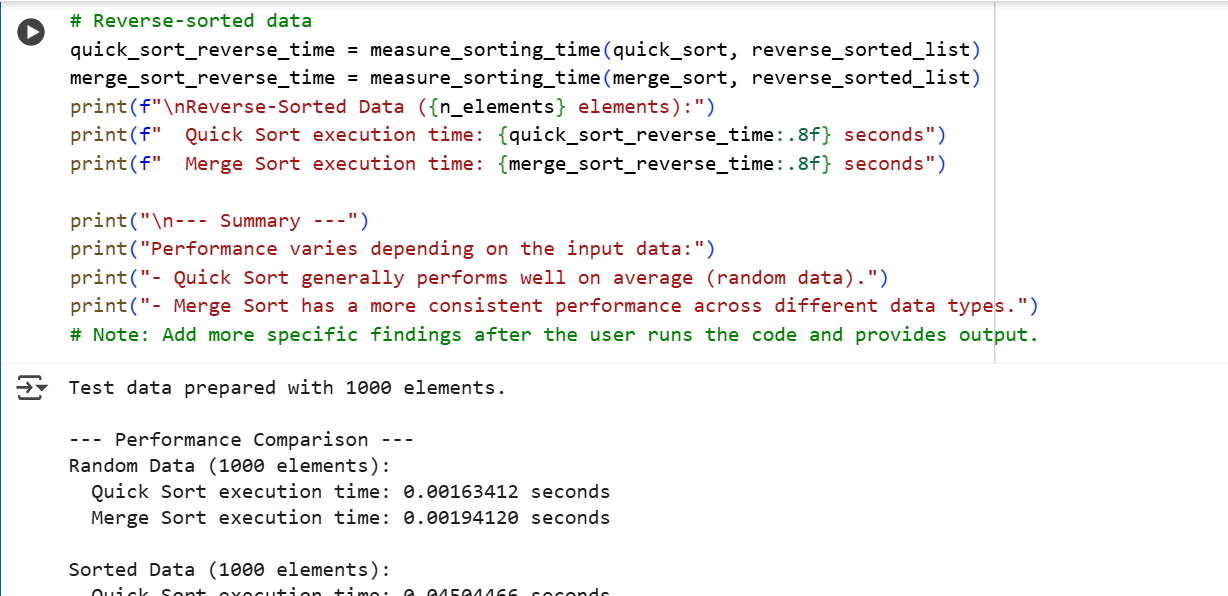


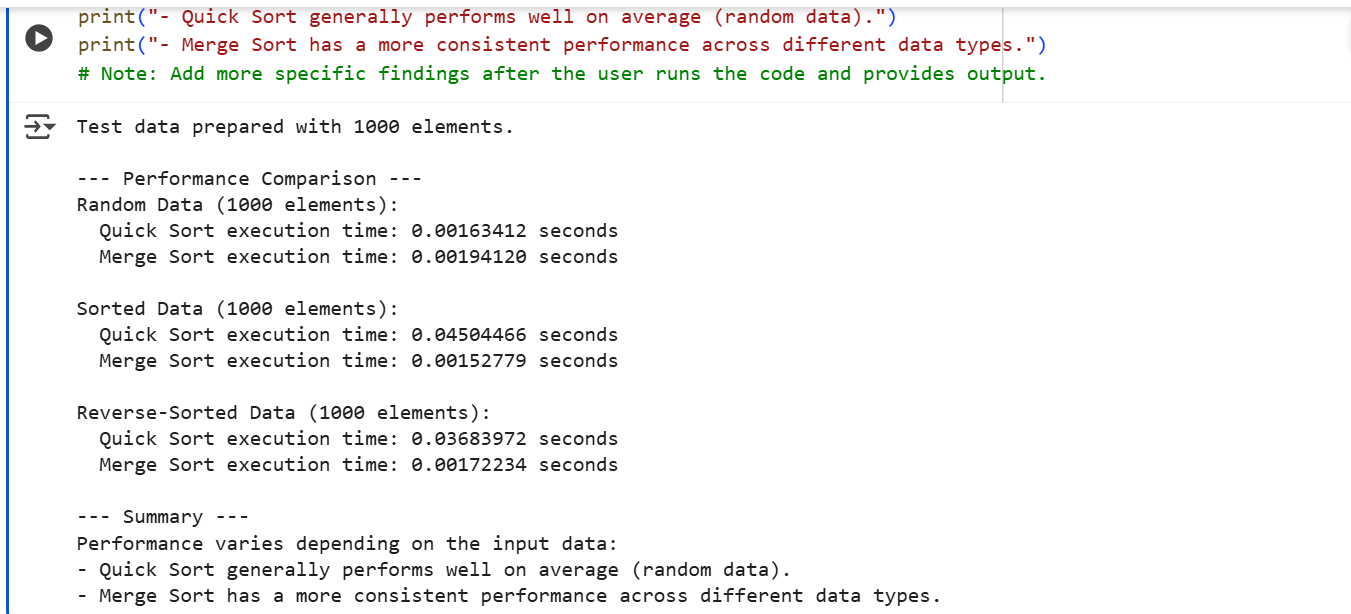












Explanation:

This cell contains the Python implementations of Quick Sort and Merge Sort, both using recursion, along with functions to create different types of test data and measure their performance.

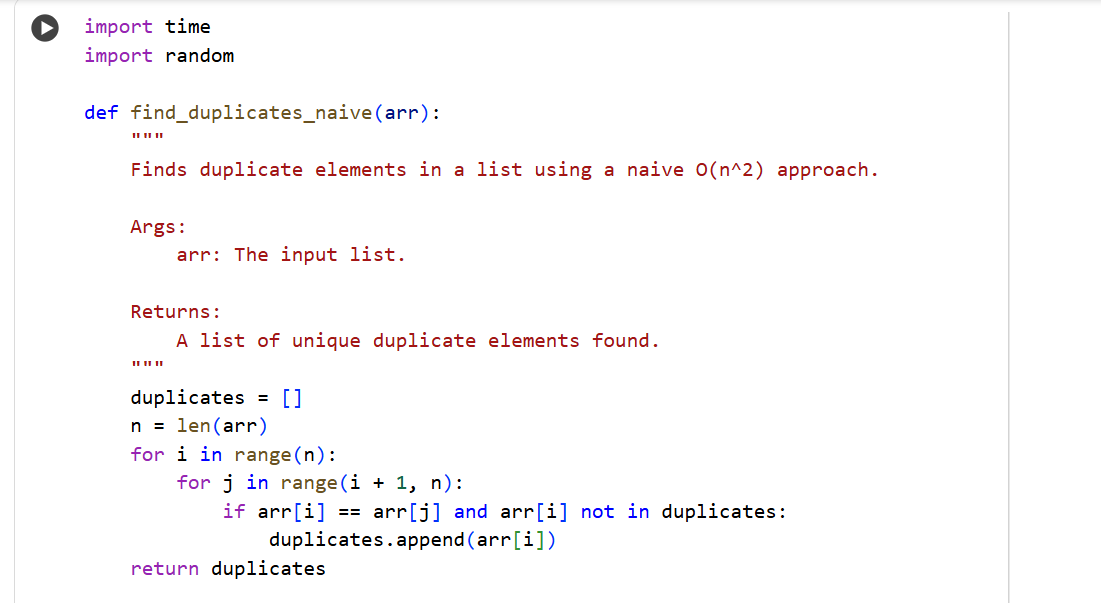
Here's a breakdown:

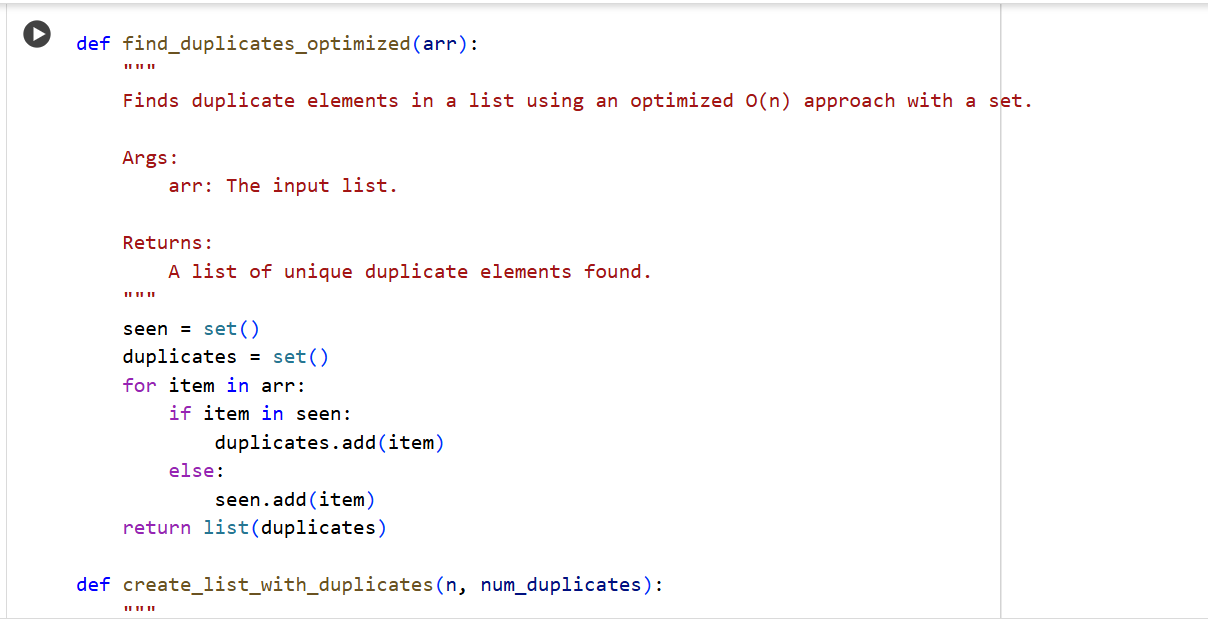
1. **sys.setrecursionlimit(2000):** This line increases the maximum depth of recursion allowed in Python. Recursive algorithms like Quick Sort and Merge Sort can potentially lead to deep recursion, and this helps prevent a RecursionError for larger inputs, although very large inputs might still cause issues depending on the limit set and the specific input data for Quick Sort.
2. **quick\_sort(arr) function:**
   * This is a recursive implementation of Quick Sort.
   * **Base Case:** If the input list arr has 0 or 1 element (len(arr) <= 1), it's already sorted, so the function returns the list as is.
   * **Recursive Step:**
     + It selects the first element as the pivot.
     + It then creates two new lists using list comprehensions: less contains all elements from the rest of the array (arr[1:]) that are less than or equal to the pivot, and greater contains elements greater than the pivot.
     + It recursively calls quick\_sort on the less list and the greater list.
     + Finally, it combines the sorted less list, the pivot element, and the sorted greater list to produce the fully sorted list.
   * Quick Sort's performance is highly dependent on the pivot choice.
3. **merge\_sort(arr) function:**
   * This is a recursive implementation of Merge Sort.
   * **Base Case:** Similar to Quick Sort, if the list has 0 or 1 element, it's returned as is.
   * **Recursive Step:**
     + The list is divided into two halves: left\_half and right\_half.
     + merge\_sort is recursively called on both halves to sort them independently.
     + The merge function is then called to combine the two sorted halves into a single sorted list.
   * Merge Sort consistently divides the list, leading to more predictable performance.
4. **merge(left, right) function:**
   * This is a helper function for merge\_sort.
   * It takes two already sorted lists, left and right, and merges them into a single sorted list.
   * It uses two pointers (i for left and j for right) to compare elements from both lists and append the smaller element to merged\_list until all elements from one list are added.
   * Finally, it appends any remaining elements from the list that still has elements.
5. **create\_random\_array(n), create\_sorted\_array(n), create\_reverse\_sorted\_array(n) functions:**
   * These are utility functions to create different types of lists of size n for testing: a list with random integers, a list sorted in ascending order, and a list sorted in descending order.
6. **measure\_sorting\_time(sorting\_function, arr) function:**
   * This helper function measures the execution time of a given sorting\_function on a provided list arr.
   * It uses copy.deepcopy(arr) to ensure that the original test data is not modified by the sorting process when measuring time.
7. **Main execution block:**
   * This section sets up the test environment:
     + n\_elements = 1000 defines the size of the lists to be used for testing.
     + It creates the random\_list, sorted\_list, and reverse\_sorted\_list using the helper functions.
     + It then calls measure\_sorting\_time for both quick\_sort and merge\_sort on each type of list (random, sorted, reverse-sorted).
     + The execution times are printed for each combination.
     + A summary is printed based on the observed performance on these specific test cases.

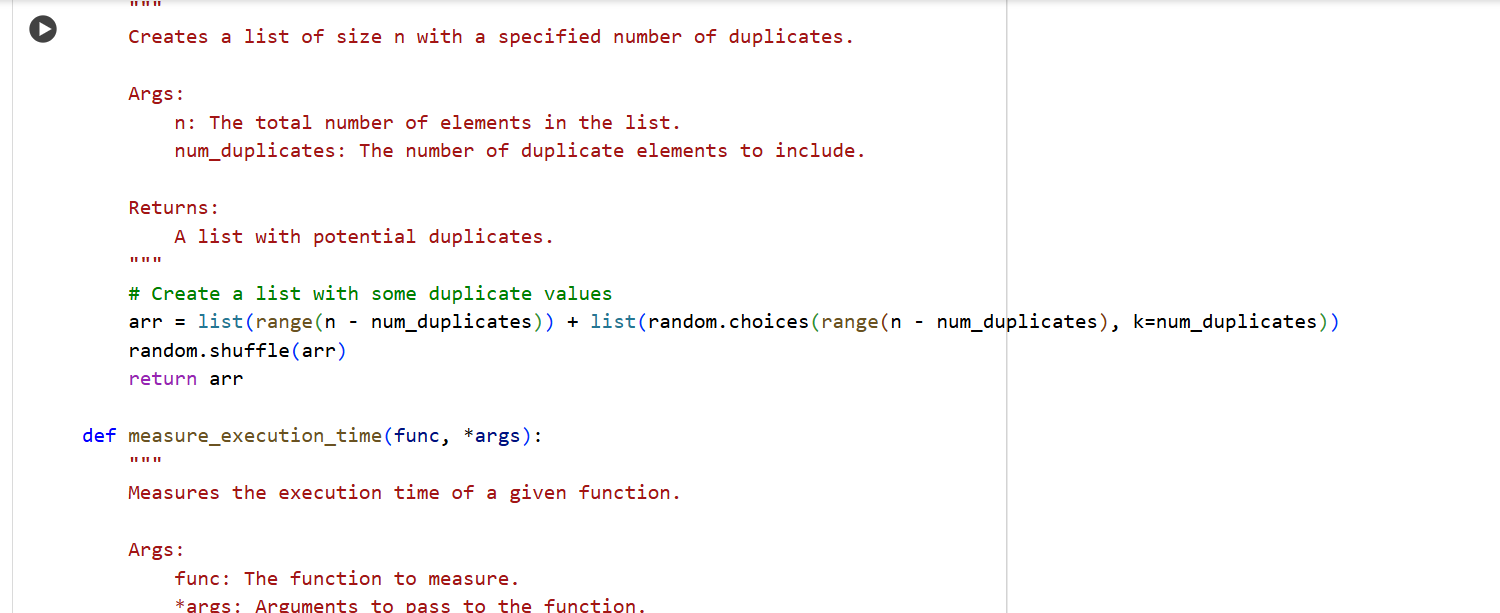
Task 5:

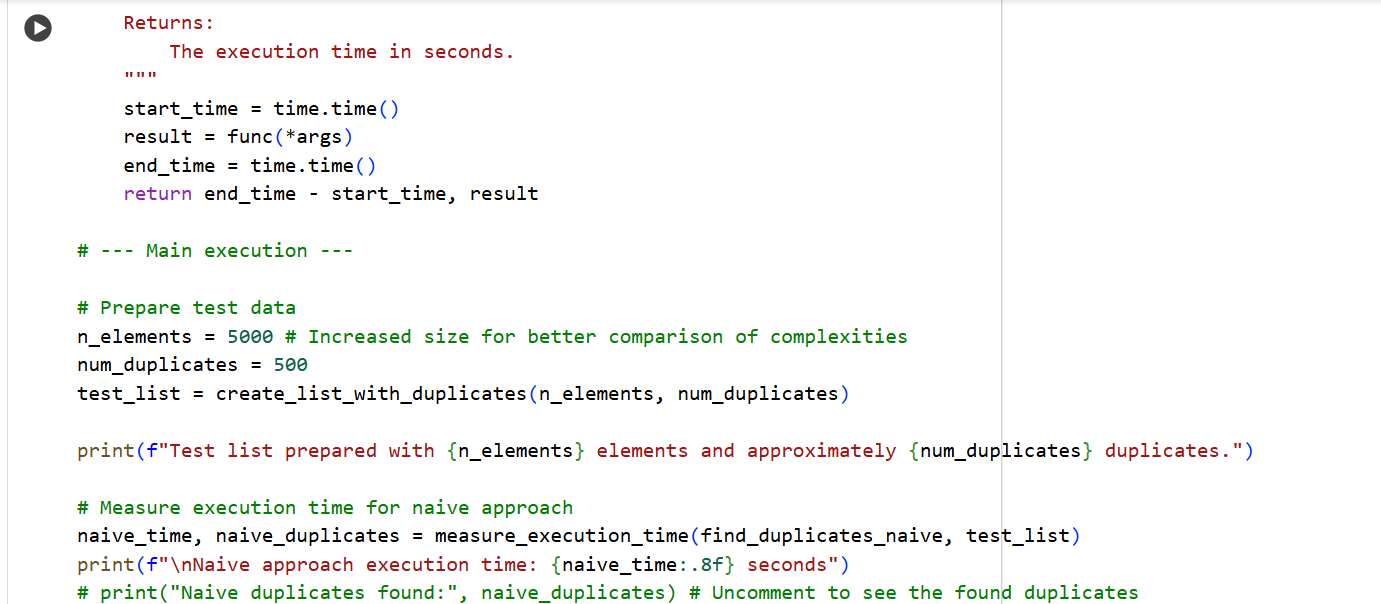
AI-Suggested Algorithm Optimization

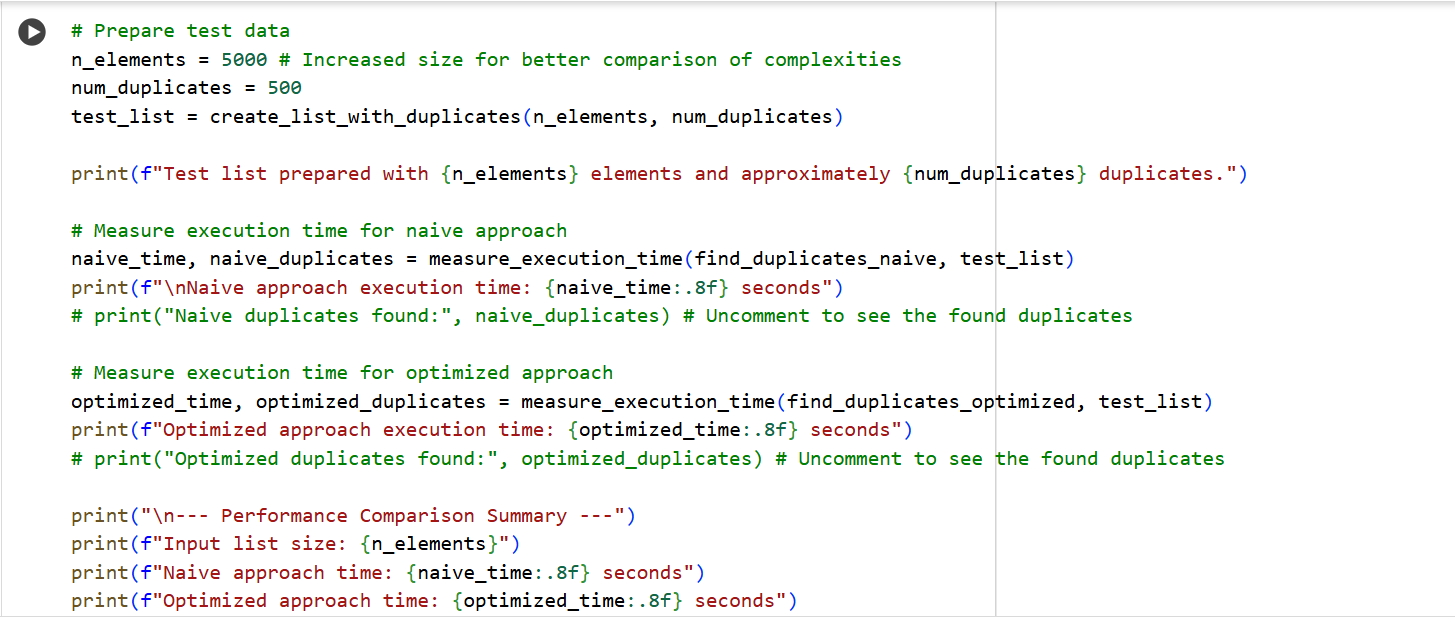
* **Task:** Give AI a naive algorithm (e.g., O(n²) duplicate search).
* **Instructions:**
  + Students write a brute force duplicate-finder.
  + Ask AI to optimize it (e.g., by using sets/dictionaries with O(n) time).
  + Compare execution times with large input sizes.
* **Expected Output:**
  + Two versions of the same algorithm (brute force + optimized).
  + AI explanation of how complexity was improved.

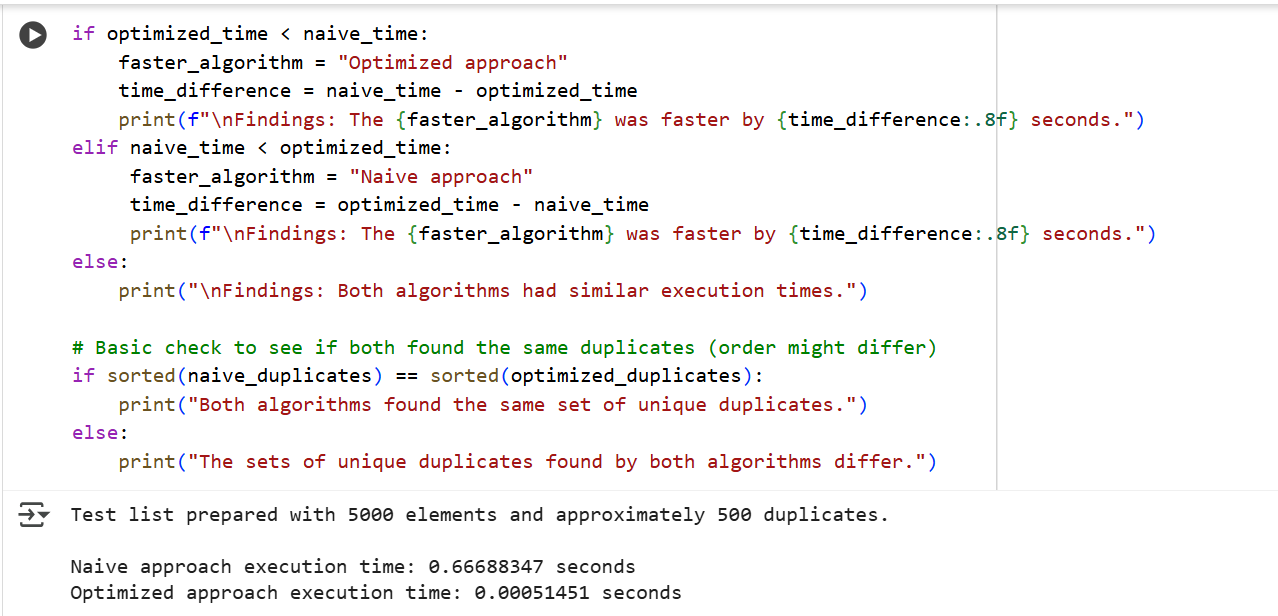


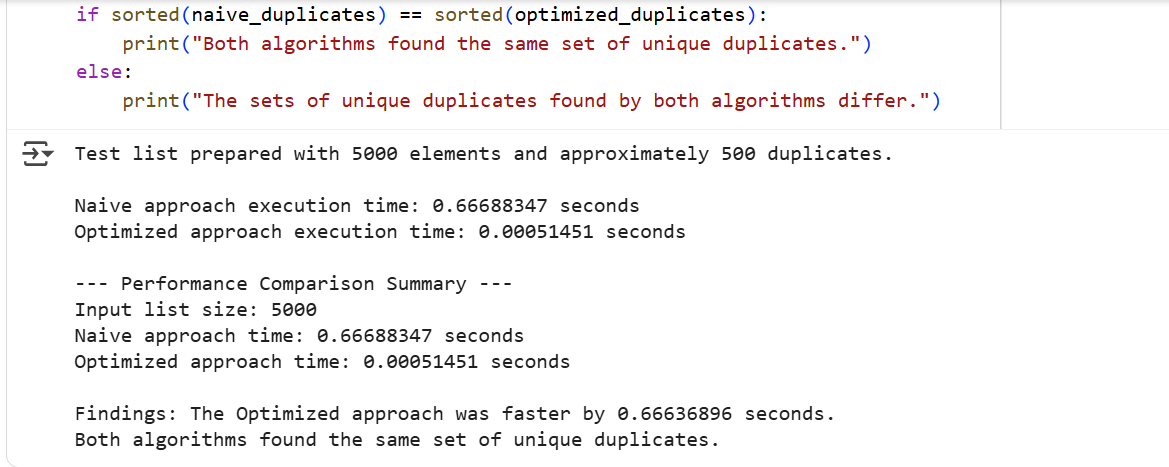












Explanation:

1. **find\_duplicates\_naive(arr) function:**
   * This function implements a naive approach to finding duplicates with a time complexity of O(n^2).
   * It uses nested loops to compare every element with every other element in the list.
   * The outer loop iterates from the first element to the second-to-last element (for i in range(n):).
   * The inner loop iterates from the element after the current outer loop element to the end of the list (for j in range(i + 1, n):).
   * If two elements arr[i] and arr[j] are found to be equal, and arr[i] is not already in the duplicates list, it's added to the duplicates list.
   * This approach is simple to understand but becomes very slow as the size of the input list increases because the number of comparisons grows quadratically with the number of elements.
2. **find\_duplicates\_optimized(arr) function:**
   * This function implements a more optimized approach to finding duplicates with a time complexity of O(n).
   * It uses a set called seen to keep track of elements encountered so far and another set called duplicates to store the unique duplicate elements.
   * It iterates through the input list arr only once (for item in arr:).
   * For each item, it checks if the item is already in the seen set.
     + If it is in seen, it means the element has been encountered before, so it's a duplicate. The item is added to the duplicates set.
     + If it's not in seen, the item is added to the seen set.
   * Using sets for checking membership and adding elements allows for average O(1) time complexity for these operations, resulting in an overall O(n) time complexity for the function.
   * Finally, it converts the duplicates set back to a list before returning it.
3. **create\_list\_with\_duplicates(n, num\_duplicates) function:**
   * This function generates a list of size n that includes a specified number of num\_duplicates.
   * It creates a list of unique elements first (list(range(n - num\_duplicates))) and then adds num\_duplicates by randomly choosing from these unique elements (list(random.choices(range(n - num\_duplicates), k=num\_duplicates))).
   * Finally, it shuffles the combined list to distribute the duplicates randomly.
4. **measure\_execution\_time(func, \*args) function:**
   * This is a helper function that measures the execution time of a given function func with the provided arguments \*args.
5. **Main execution block:**
   * This section sets up and runs the performance comparison:
     + n\_elements = 5000 and num\_duplicates = 500 define the characteristics of the test list.
     + test\_list = create\_list\_with\_duplicates(n\_elements, num\_duplicates) creates the test data.
     + It then measures the execution time for both find\_duplicates\_naive and find\_duplicates\_optimized using the measure\_execution\_time function and prints the times.
     + A performance comparison summary is printed, highlighting which algorithm was faster and by how much.
     + A basic check is performed to verify that both algorithms found the same set of unique duplicates, regardless of the order.

This code effectively demonstrates the significant performance improvement achieved by using a more optimized algorithm (O(n)) compared to a naive approach (O(n^2)) for finding duplicates, especially for larger datasets.