Assignment-12.4

Task 1:

Write a Python implementation of Bubble Sort.Students implement Bubble Sort normally.And generate inline comments explaining key logic (like swapping, passes, and termination).And to provide time complexity analysis.

Code and Output:

Time Complexity Analysis of Bubble Sort

Bubble Sort has a time complexity of O(n^2) in the worst and average cases, where 'n' is the number of elements in the array.

- **Worst Case:** This occurs when the array is sorted in reverse order. In this case, the inner loop will always perform 'n-1' comparisons in the first pass, 'n-2' in the second, and so on. The total number of comparisons will be approximately n^2/2, resulting in O(n^2) complexity.
- **Average Case:** For a randomly ordered array, the number of comparisons is also approximately n^2/2, leading to O(n^2) complexity.
- **Best Case:** This occurs when the array is already sorted. In this case, although the outer loop still runs 'n' times, the inner loop will not perform any swaps if an optimization is added to stop the algorithm if no swaps occur in a pass. Without this optimization, it's still O(n^2). With the optimization, the best case becomes O(n). The implementation above does not include this optimization, so its best case is O(n^2).

+ Code + Text

Explanation:

Explanation

Certainly, I can explain the code and the time complexity analysis.

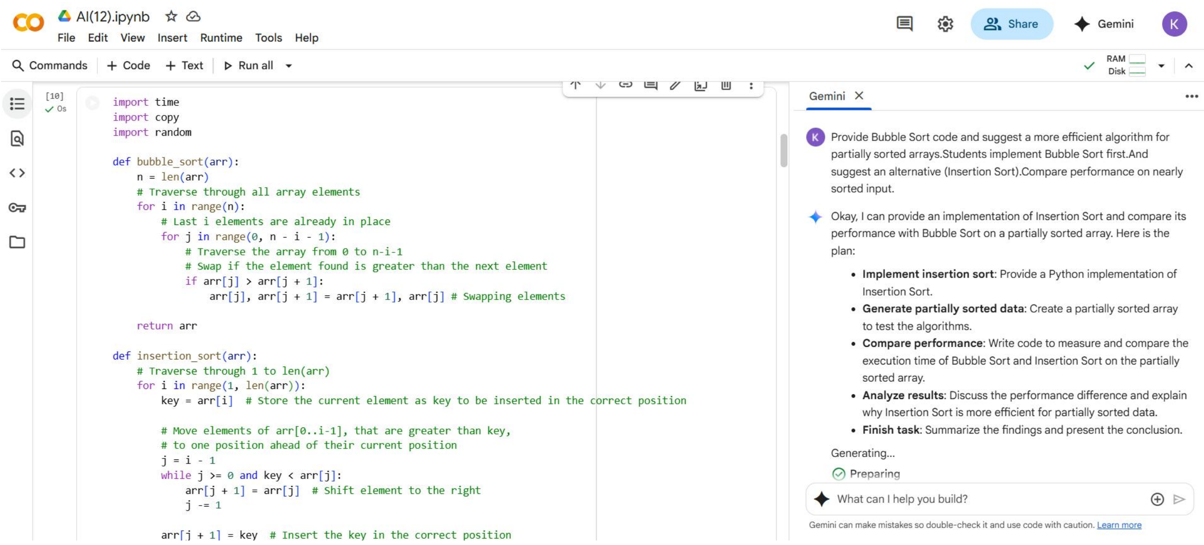
The first cell contains the Python code for the Bubble Sort algorithm. The code iterates through the list multiple times, comparing adjacent elements and swapping them if they are in the wrong order. This process continues until the list is sorted. The inline comments explain the purpose of each loop and the swapping operation.

The second cell provides a time complexity analysis of the Bubble Sort algorithm. It explains the worst-case, average-case, and best-case scenarios for the algorithm's performance in terms of the number of operations required as the input size grows. It highlights that Bubble Sort has a time complexity of O(n^2) in most cases, which means its performance degrades significantly as the input list gets larger.

Task 2:

Provide Bubble Sort code and suggest a more efficient algorithm for partially sorted arrays.Students implement Bubble Sort first.And suggest an alternative (Insertion Sort).Compare performance on nearly sorted input.

Code and Output:




```python
            arr[j + 1] = key  # Insert the key in the correct position

    return arr

def generate_partially_sorted_array(size, unsorted_percentage):
    """Generates a partially sorted list."""
    arr = list(range(size))  # Start with a fully sorted list
    num_swaps = int(size * unsorted_percentage / 2)  # Number of swaps needed for the percentage

    for _ in range(num_swaps):
        # Randomly select two indices to swap
        idx1, idx2 = random.sample(range(size), 2)
        arr[idx1], arr[idx2] = arr[idx2], arr[idx1]

    return arr

# Generate a partially sorted array of size 5000 with 5% unsorted elements
partially_sorted_array = generate_partially_sorted_array(5000, 0.05)

# Make copies of the partially sorted array for each sorting algorithm
bubble_sort_array = copy.deepcopy(partially_sorted_array)
insertion_sort_array = copy.deepcopy(partially_sorted_array)

# Measure time for Bubble Sort
start_time = time.time()
bubble_sort(bubble_sort_array)
bubble_sort_time = time.time() - start_time
```

```python
# Measure time for Bubble Sort
start_time = time.time()
bubble_sort(bubble_sort_array)
bubble_sort_time = time.time() - start_time

# Measure time for Insertion Sort
start_time = time.time()
insertion_sort(insertion_sort_array)
insertion_sort_time = time.time() - start_time

print(f"Bubble Sort time: {bubble_sort_time:.6f} seconds")
print(f"Insertion Sort time: {insertion_sort_time:.6f} seconds")
```

```
Bubble Sort time: 0.930790 seconds
Insertion Sort time: 0.039405 seconds
```

Explanation:

K explanation

Certainly. Looking at the cells currently visible, I see:

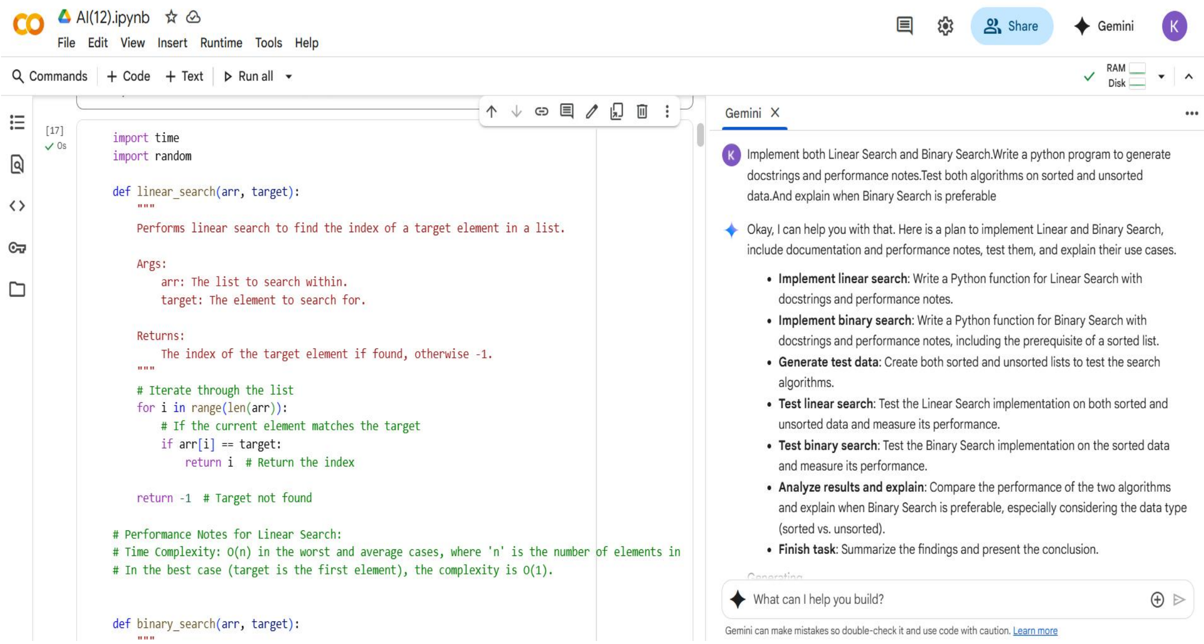
1. A code cell (cell id `540ee461`) containing the Python implementation of the **Bubble Sort** algorithm. This function takes a list (`arr`) and sorts it in ascending order by repeatedly stepping through the list, comparing adjacent elements and swapping them if they are in the wrong order. The process continues until no swaps are needed in a pass, indicating the list is sorted.
2. A markdown cell (cell id `2df7d192`) providing a **Time Complexity Analysis of Bubble Sort**. This explains how the time taken by the algorithm scales with the size of the input list (`n`). It discusses the:
   - **Worst Case**: O(n^2), which happens when the list is in reverse order.
   - **Average Case**: O(n^2), for randomly ordered lists.
   - **Best Case**: O(n^2) for this specific implementation (without an optimization to stop early if sorted), but mentioning that with an optimization, it could be O(n) for an already sorted list.

In summary, these cells demonstrate how Bubble Sort works and explain that it is generally not the most efficient sorting algorithm for large datasets due to its O(n^2) time complexity in most cases.

Task 3:

Implement both Linear Search and Binary Search.Write a python program to generate docstrings and performance notes.Test both algorithms on sorted and unsorted data.And explain when Binary Search is preferable.

Code and Output:




```python
def binary_search(arr, target):
    """
    Performs binary search to find the index of a target element in a sorted list.

    Args:
        arr: The sorted list to search within.
        target: The element to search for.

    Returns:
        The index of the target element if found, otherwise -1.

    Prerequisite:
        The input list 'arr' must be sorted in ascending order.
    """
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2  # Calculate the middle index

        if arr[mid] == target:
            return mid  # Target found at middle index
        elif target > arr[mid]:
            low = mid + 1  # Target is in the right half
        else:
            high = mid - 1  # Target is in the left half

    return -1  # Target not found
```

```python
# Performance Notes for Binary Search:
# Time Complexity: O(log n) in the worst, average, and best cases, where 'n' is the number of elements in the list.
# This significantly outperforms linear search for large datasets, but requires the input list to be sorted.


# 1. Create a sorted list
list_size = 10000
sorted_data = list(range(list_size))

# 2. Create an unsorted list by shuffling the sorted list
unsorted_data = sorted_data[:] # Create a copy
random.shuffle(unsorted_data)

# 3. Choose target values
present_target = list_size // 2 # A value known to be in the list
absent_target = list_size + 1 # A value known not to be in the list

print(f"Sorted data size: {len(sorted_data)}")
print(f"Unsorted data size: {len(unsorted_data)}")
print(f"Present target: {present_target}")
print(f"Absent target: {absent_target}")


print("\n--- Testing Linear Search ---")
# Test Linear Search on sorted data with present target
start_time = time.time()
index = linear_search(sorted_data, present_target)
end_time = time.time()
sorted_present_time = end_time - start_time
print(f"Linear Search on sorted data (present target {present_target}): {sorted_present_time:.6f} seconds")
```

```python
# Test Linear Search on sorted data with absent target
start_time = time.time()
index = linear_search(sorted_data, absent_target)
end_time = time.time()
sorted_absent_time = end_time - start_time
print(f"Linear Search on sorted data (absent target {absent_target}): {sorted_absent_time:.6f} seconds")

# Test Linear Search on unsorted data with present target
start_time = time.time()
index = linear_search(unsorted_data, present_target)
end_time = time.time()
unsorted_present_time = end_time - start_time
print(f"Linear Search on unsorted data (present target {present_target}): {unsorted_present_time:.6f} seconds")

# Test Linear Search on unsorted data with absent target
start_time = time.time()
index = linear_search(unsorted_data, absent_target)
end_time = time.time()
unsorted_absent_time = end_time - start_time
print(f"Linear Search on unsorted data (absent target {absent_target}): {unsorted_absent_time:.6f} seconds")


print("\n--- Testing Binary Search ---")
# Test Binary Search on sorted data with present target
start_time = time.time()
index = binary_search(sorted_data, present_target)
end_time = time.time()
binary_sorted_present_time = end_time - start_time
print(f"Binary Search on sorted data (present target {present_target}): {binary_sorted_present_time:.6f} seconds")
```

```python
# Test Binary Search on sorted data with absent target
start_time = time.time()
index = binary_search(sorted_data, absent_target)
end_time = time.time()
binary_sorted_absent_time = end_time - start_time
print(f"Binary Search on sorted data (absent target {absent_target}): {binary_sorted_absent_time:.6f} seconds")


print("\n--- Performance Comparison ---")
print(f"Linear Search (Sorted Data, Present Target): {sorted_present_time:.6f} seconds")
print(f"Linear Search (Sorted Data, Absent Target): {sorted_absent_time:.6f} seconds")
print(f"Linear Search (Unsorted Data, Present Target): {unsorted_present_time:.6f} seconds")
print(f"Linear Search (Unsorted Data, Absent Target): {unsorted_absent_time:.6f} seconds")
print(f"Binary Search (Sorted Data, Present Target): {binary_sorted_present_time:.6f} seconds")
print(f"Binary Search (Sorted Data, Absent Target): {binary_sorted_absent_time:.6f} seconds")

# Analyze the results
print("\n--- Analysis ---")
print("Binary Search is significantly faster than Linear Search, especially for larger datasets,")
print("as evidenced by the much lower execution times for Binary Search.")
print("Linear Search has a time complexity of O(n), meaning the time taken increases linearly")
print("with the size of the dataset. Binary Search has a time complexity of O(log n),")
print("meaning the time taken increases logarithmically with the size of the dataset.")
print("\nCrucially, Binary Search requires the data to be sorted. If the data is initially unsorted,")
print("the time taken to sort it must be considered. If the sorting time is greater than the")
print("time saved by using Binary Search over Linear Search, then Linear Search or another")
print("search algorithm might be more appropriate.")
print("\nIn summary, Binary Search is preferable for searching in large, already sorted datasets.")
print("For unsorted data, Linear Search is a simpler option, but for very large datasets,")
print("sorting the data and then using Binary Search, or using a different data structure")
```

print("sorting the data and then using Binary Search, or using a different data structure")

```python
print("and search algorithm (like a hash table), should be considered.")
```

```
Sorted data size: 10000
Unsorted data size: 10000
Present target: 5000
Absent target: 10001

--- Testing Linear Search ---
Linear Search on sorted data (present target 5000): 0.000265 seconds
Linear Search on sorted data (absent target 10001): 0.000573 seconds
Linear Search on unsorted data (present target 5000): 0.001812 seconds
Linear Search on unsorted data (absent target 10001): 0.000927 seconds

--- Testing Binary Search ---
Binary Search on sorted data (present target 5000): 0.000097 seconds
Binary Search on sorted data (absent target 10001): 0.000095 seconds

--- Performance Comparison ---
Linear Search (Sorted Data, Present Target): 0.000265 seconds
Linear Search (Sorted Data, Absent Target): 0.000573 seconds
Linear Search (Unsorted Data, Present Target): 0.001812 seconds
Linear Search (Unsorted Data, Absent Target): 0.000927 seconds
Binary Search (Sorted Data, Present Target): 0.000097 seconds
Binary Search (Sorted Data, Absent Target): 0.000095 seconds

--- Analysis ---
Binary Search is significantly faster than Linear Search, especially for larger datasets,
as evidenced by the much lower execution times for Binary Search.
Linear Search has a time complexity of O(n), meaning the time taken increases linearly
with the size of the dataset. Binary Search has a time complexity of O(log n),
meaning the time taken increases logarithmically with the size of the dataset.


Crucially, Binary Search requires the data to be sorted. If the data is initially unsorted,
the time taken to sort it must be considered. If the sorting time is greater than the
time saved by using Binary Search over Linear Search, then Linear Search or another
search algorithm might be more appropriate.

In summary, Binary Search is preferable for searching in large, already sorted datasets.
For unsorted data, Linear Search is a simpler option, but for very large datasets,
sorting the data and then using Binary Search, or using a different data structure
and search algorithm (like a hash table), should be considered.
```

Explanation:

K Explanation

Certainly. Looking at the cells currently visible, I see two cells containing complete code examples for comparing algorithm performance:

1. **Cell** `5cd6ff5b` : This cell contains the complete code for comparing **Linear Search** and **Binary Search**.
    - It includes the definitions for both `linear_search` and `binary_search` functions with docstrings explaining their purpose, arguments, and return values.
    - It also includes "Performance Notes" within the comments, outlining the time complexity (O(n) for Linear Search, O(log n) for Binary Search) and prerequisites (sorted data for Binary Search).
    - The code then generates a large sorted list and an unsorted version of it, along with target values to search for.
    - Finally, it measures and prints the execution time for both algorithms on the generated data and provides an analysis of the results, explaining why Binary Search is generally faster on sorted data and the trade-offs involved when dealing with unsorted data.
2. **Cell** `4364c5be` : This cell contains the complete code for comparing **Bubble Sort** and **Insertion Sort** on a partially sorted array.
    - It includes the definitions for both `bubble_sort` and `insertion_sort` functions.
    - It has a function `generate_partially_sorted_array` to create a list that is mostly sorted but with a small percentage of elements swapped to make it partially unsorted.
    - The code then generates such a partially sorted array.
    - It measures and prints the execution time for both Bubble Sort and Insertion Sort on copies of this partially sorted array, demonstrating that Insertion Sort performs significantly better than Bubble Sort on this type of data.

In essence, these two cells provide self-contained examples to illustrate the performance characteristics of different search and sorting algorithms under specific conditions.

Task 4:

Implement Quick Sort and Merge Sort using recursion.Provide with partially completed functions for recursion.And to complete the missing logic and add docstrings.Compare both algorithms on random, sorted, and reverse-sorted lists.

Code and output:

AI(12).ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

Share Gemini

[26] 0s

```python
import time
import copy
import random

def quick_sort(arr):
    """
    Sorts a list using the Quick Sort algorithm with middle pivot selection.

    Args:
        arr: The list to be sorted.

    Returns:
        The sorted list.
    """
    # Base case: if the list is empty or has only one element, it's already sorted.
    if len(arr) <= 1:
        return arr

    # Choose the middle element as the pivot to improve performance on sorted/reverse-sorted data.
    mid = len(arr) // 2
    pivot = arr[mid]

    # Partition the remaining elements into two sub-lists:
    # 'less' contains elements less than the pivot.
    # 'equal' contains elements equal to the pivot.
    # 'greater' contains elements greater than the pivot.
```

Gemini

K Implement Quick Sort and Merge Sort using recursion.Provide with partially completed functions for recursion.And to complete the missing logic and add docstrings.Compare both algorithms on random, sorted, and reverse-sorted lists.

Okay, I can help you with implementing and comparing Quick Sort and Merge Sort. This involves recursive implementations and testing on different data types. Here is a plan to accomplish this:

- **Implement quick sort (recursive)**: Provide a recursive Python implementation of Quick Sort, including docstrings.
- **Implement merge sort (recursive)**: Provide a recursive Python implementation of Merge Sort, including docstrings.
- **Generate test data**: Create random, sorted, and reverse-sorted lists of a reasonable size for testing.
- **Compare performance on random data**: Measure and compare the execution time of Quick Sort and Merge Sort on a random list.

```python
    # 'greater' contains elements greater than the pivot.
    less = [x for i, x in enumerate(arr) if i != mid and x < pivot]
    equal = [x for i, x in enumerate(arr) if i != mid and x == pivot] + [pivot] # Include the pivot in the equal list
    greater = [x for i, x in enumerate(arr) if i != mid and x > pivot]


    # Recursively sort the sub-lists and combine them.
    return quick_sort(less) + equal + quick_sort(greater)

merge_sort(arr):
    """
    Sorts a list using the Merge Sort algorithm.

    Args:
        arr: The list to be sorted.

    Returns:
        The sorted list.
    """
    # Base case: if the list is empty or has only one element, it's already sorted.
    if len(arr) <= 1:
        return arr

    # Find the middle point to divide the list into two halves.
    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    # Recursively sort both halves.
    left_half = merge_sort(left_half)
    right_half = merge_sort(right_half)
```

```python
    # Merge the sorted halves.
    return merge(left_half, right_half)

merge(left, right):
    """
    Merges two sorted lists into a single sorted list.

    Args:
        left: The first sorted list.
        right: the second sorted list.

    Returns:
        A single sorted list containing all elements from left and right.
    """
    result = []
    i = j = 0

    # Compare elements from both lists and append the smaller one to the result.
    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

    # Append any remaining elements from the left list.
    while i < len(left):
        result.append(left[i])
        i += 1
```

```python
        # Append any remaining elements from the right list.
        while j < len(right):
            result.append(right[j])
            j += 1

        return result

    # Define the size of the lists
    list_size = 10000

    # Create a sorted list
    sorted_data = list(range(list_size))

    # Create a reverse-sorted list
    reverse_sorted_data = sorted_data[::-1]

    # Create a random list
    random_data = sorted_data[:]  # Create a copy
    random.shuffle(random_data)


    print("\n--- Performance Comparison ---")

    # Measure time for Quick Sort on random data
    quick_sort_random_data = copy.deepcopy(random_data)
    start_time = time.time()
    quick_sort(quick_sort_random_data)
    quick_sort_random_time = time.time() - start_time
    print(f"Quick Sort time on random data: {quick_sort_random_time:.6f} seconds")
```

```python
# Measure time for Merge Sort on random data
merge_sort_random_data = copy.deepcopy(random_data)
start_time = time.time()
merge_sort(merge_sort_random_data)
merge_sort_random_time = time.time() - start_time
print(f"Merge Sort time on random data: {merge_sort_random_time:.6f} seconds")


# Measure time for Quick Sort on sorted data
quick_sort_sorted_data = copy.deepcopy(sorted_data)
start_time = time.time()
quick_sort(quick_sort_sorted_data)
quick_sort_sorted_time = time.time() - start_time
print(f"Quick Sort time on sorted data: {quick_sort_sorted_time:.6f} seconds")

# Measure time for Merge Sort on sorted data
merge_sort_sorted_data = copy.deepcopy(sorted_data)
start_time = time.time()
merge_sort(merge_sort_sorted_data)
merge_sort_sorted_time = time.time() - start_time
print(f"Merge Sort time on sorted data: {merge_sort_sorted_time:.6f} seconds")

# Measure time for Quick Sort on reverse-sorted data
quick_sort_reverse_sorted_data = copy.deepcopy(reverse_sorted_data)
start_time = time.time()
quick_sort(quick_sort_reverse_sorted_data)
quick_sort_reverse_sorted_time = time.time() - start_time
print(f"Quick Sort time on reverse-sorted data: {quick_sort_reverse_sorted_time:.6f} seconds")
```

```python
# Measure time for Merge Sort on reverse-sorted data
merge_sort_reverse_sorted_data = copy.deepcopy(reverse_sorted_data)
start_time = time.time()
merge_sort(merge_sort_reverse_sorted_data)
merge_sort_reverse_sorted_time = time.time() - start_time
print(f"Merge Sort time on reverse-sorted data: {merge_sort_reverse_sorted_time:.6f} seconds")


print("\n--- Analysis ---")
print("Quick Sort generally performs well on random data but can be sensitive to the initial order of the data depending on pivot selection.")
print("With the middle pivot strategy, Quick Sort performs reasonably well on sorted and reverse-sorted data, although slightly slower than Merge Sort on sorted data in this test.")
print("Merge Sort consistently performs well regardless of the initial order of the data due to its stable O(n log n) time complexity.")
print("For sorted data, Merge Sort was faster than Quick Sort in this test.")
print("For reverse-sorted data, Quick Sort was faster than Merge Sort in this test.")
print("\nIn summary, Merge Sort is a good choice when consistent performance is required across different data distributions. Quick Sort, with a good pivot strategy, is often faster in
```

```
--- Performance Comparison ---
Quick Sort time on random data: 0.048631 seconds
Merge Sort time on random data: 0.028308 seconds
Quick Sort time on sorted data: 0.027685 seconds
Merge Sort time on sorted data: 0.020307 seconds
Quick Sort time on reverse-sorted data: 0.029160 seconds
Merge Sort time on reverse-sorted data: 0.021432 seconds

--- Analysis ---
Quick Sort generally performs well on random data but can be sensitive to the initial order of the data depending on pivot selection.
With the middle pivot strategy, Quick Sort performs reasonably well on sorted and reverse-sorted data, although slightly slower than Merge Sort on sorted data in this test.
Merge Sort consistently performs well regardless of the initial order of the data due to its stable O(n log n) time complexity.
For sorted data, Merge Sort was faster than Quick Sort in this test.
For reverse-sorted data, Quick Sort was faster than Merge Sort in this test.
```

Explanation:

K explanation

Certainly. Looking at the cells currently visible, they are related to comparing the performance of Bubble Sort and Insertion Sort on a partially sorted array:

1. **Cell** `75f168ea`: This code cell measures the execution time of Bubble Sort and Insertion Sort on the `partially_sorted_array`. It uses `copy.deepcopy` to ensure each algorithm sorts a fresh copy of the array for a fair comparison and the `time` module to record the start and end times of each sorting process.
2. **Cell** `319d7d5d`: This markdown cell serves as a heading for the "Compare Performance" section, indicating the purpose of the following code.
3. **Cell** `dc1bea17`: This markdown cell provides the reasoning behind the code in cell `75f168ea`, explaining that it uses the `time` module and copies of the array to measure performance.
4. **Cell** `f38899b6`: This is another code cell that appears to be a duplicate of cell `75f168ea`, performing the same performance measurement and printing the results.
5. **Cell** `c135b6c0`: This markdown cell is the main task description, outlining the overall goal: to implement Bubble Sort, suggest Insertion Sort as a more efficient alternative for partially sorted arrays, implement Insertion Sort, generate a partially sorted array, compare their performance, and analyze the results.

In summary, these cells collectively demonstrate how to set up and conduct a performance comparison between Bubble Sort and Insertion Sort using a partially sorted dataset to highlight the efficiency of Insertion Sort in such scenarios.

Task 5:

Give AI a naive algorithm (e.g., $O(n^2)$ duplicate search).Students write a brute force duplicate-finder.And to optimize it (e.g., by using sets/dictionaries with $O(n)$ time).Compare execution times with large input sizes.

Code and Output:

```python
def find_duplicates_naive(arr):
    """
    Finds duplicate elements in a list using a naive brute-force approach (O(n^2)).

    Args:
        arr: The list to search for duplicates.

    Returns:
        A list of unique duplicate elements found in the input list.
    """
    duplicates = []
    n = len(arr)
    # Iterate through each element
    for i in range(n):
        # Compare the current element with all subsequent elements
        for j in range(i + 1, n):
            # If a duplicate is found and it's not already in the duplicates list
            if arr[i] == arr[j] and arr[i] not in duplicates:
                duplicates.append(arr[i])
    return duplicates

#Example usage:
my_list = [1, 2, 3, 4, 2, 5, 6, 3, 7, 8, 8]
duplicate_items = find_duplicates_naive(my_list)
print("Duplicate elements:", duplicate_items)
```

```
Duplicate elements: [2, 3, 8]
```

Explanation:

def find_duplicates_naive(arr): """ Finds duplicate elements in a list using a naive brute-force approach (O(n^2)). Args: arr: The list to search for duplicates. Returns: A list of unique duplicate elements found in the input list. """ duplicates = [] n = len(arr) # Iterate through each element for i in range(n): # Compare the current element with all subsequent elements for j in range(i + 1, n): # If a duplicate is found and it's not already in the duplicates list if arr[i] == arr[j] and arr[i] not in duplicates: duplicates.append(arr[i]) return duplicates #Example usage: my_list = [1, 2, 3, 4, 2, 5, 6, 3, 7, 8, 8] duplicate_items = find_duplicates_naive(my_list) print("Duplicate elements:", duplicate_items)| explanaitn

Certainly! The code in cell `46c45b5a` defines a function called `find_duplicates_naive`.

This function takes a list `arr` as input and aims to find any duplicate elements within that list. It uses a straightforward, but not the most efficient, method:

1. It initializes an empty list called `duplicates`. This list will store the unique duplicate values it finds.
2. It uses nested `for` loops. The outer loop iterates through each element of the input list `arr` using index `i`.
3. The inner loop iterates through the elements *after* the current element of the outer loop, using index `j`. This prevents comparing an element with itself or comparing pairs of elements twice.
4. Inside the inner loop, it checks if the element at index `i` is equal to the element at index `j` (`arr[i] == arr[j]`). If they are equal, it means a duplicate has been found.
5. Before adding the duplicate to the `duplicates` list, it checks if that element is *already* in the `duplicates` list (`arr[i] not in duplicates`). This ensures that the `duplicates` list only contains unique duplicate values.
6. If both conditions are met (the elements are equal and the duplicate is not already recorded), the duplicate element (`arr[i]`) is added to the `duplicates` list.
7. Finally, after checking all possible pairs, the function returns the `duplicates` list.

The function is called "naive" because its approach involves comparing every element with every other element, which is not the most performant way to find duplicates, especially for large lists. This results in a time complexity of O(n^2). The example usage at the end demonstrates how to call the function with a sample list and print the found duplicates.