First Task

Sorting Algorithms Performance Analysis

A. Sorting Algorithms Comparison

In this experiment, we implement and compare the performance of four different sorting algorithms: Bubble Sort, Insertion Sort, Merge Sort, and Quick Sort. The goal is to analyze their running times for both small and large input sizes, ranging from approximately 5 to 50 for small inputs and significantly larger sizes for large inputs. The objective is to provide insights into the efficiency of these algorithms under different scenarios.

1. Implementation of Sorting Algorithms

We have implemented the following sorting algorithms:

- **Bubble Sort**: A simple comparison-based sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

- **Insertion Sort**: Another comparison-based algorithm that builds the sorted list one item at a time by repeatedly taking the next element and inserting it into its correct position.

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
```

```
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

▼ **Merge Sort**: A divide-and-conquer algorithm that recursively divides the input array into halves, sorts them, and then merges the sorted halves.

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
```

```
            j += 1
            k += 1
```

▼ **Quick Sort**: A divide-and-conquer algorithm that selects a 'pivot' element and partitions the other elements into two sub-arrays according to whether they are less than or greater than the pivot.

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    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]
    return quick_sort(left) + middle + quick_sort(right)
```

2. Test Cases

To ensure a comprehensive analysis, we conduct tests for both small and large input sizes. For small inputs, we consider arrays with sizes ranging from 5 to 50. For large inputs, we use arrays with sizes starting from 500 up to 5001 with a step of 500 to observe the scalability of the algorithms.

3. Time Measurement Approach

For precise measurement of the running times of the sorting algorithms, we employ Python's `time` module. The time measurement is seamlessly integrated into the experimental setup as outlined below:

```
def measure_running_time(sorting_function, array):
    start_time = time.time()
    sorting_function(array)
```

```
    end_time = time.time()
    elapsed_time_microseconds = (end_time - start_time) * 1_0
    return elapsed_time_microseconds
```

This implementation allows us to measure the execution time for each sorting algorithm on both small and large input sizes. The gathered time data will be used to analyze and compare the efficiency of the algorithms in the subsequent sections of the experiment.

4. Array Generation and Graph Plotting

In this section, we present the functions responsible for generating an array of random numbers and conducting a comprehensive comparison of sorting algorithms through a graphical representation. The code snippet is structured as follows:

▼ Generating an array:

```
def generate_random_array(size):
    return [random.randint(1, 100) for _ in range(size)]
```

▼ Graph Plotting:

```
def test_comparison(sizes):
    bubble_sort_times = []
    insertion_sort_times = []
    merge_sort_times = []
    quick_sort_times = []

    for size in sizes:
        arr = generate_random_array(size)

        bubble_sort_times.append(measure_running_time(bubb
        insertion_sort_times.append(measure_running_time(i
        merge_sort_times.append(measure_running_time(merge
        quick_sort_times.append(measure_running_time(quick
```

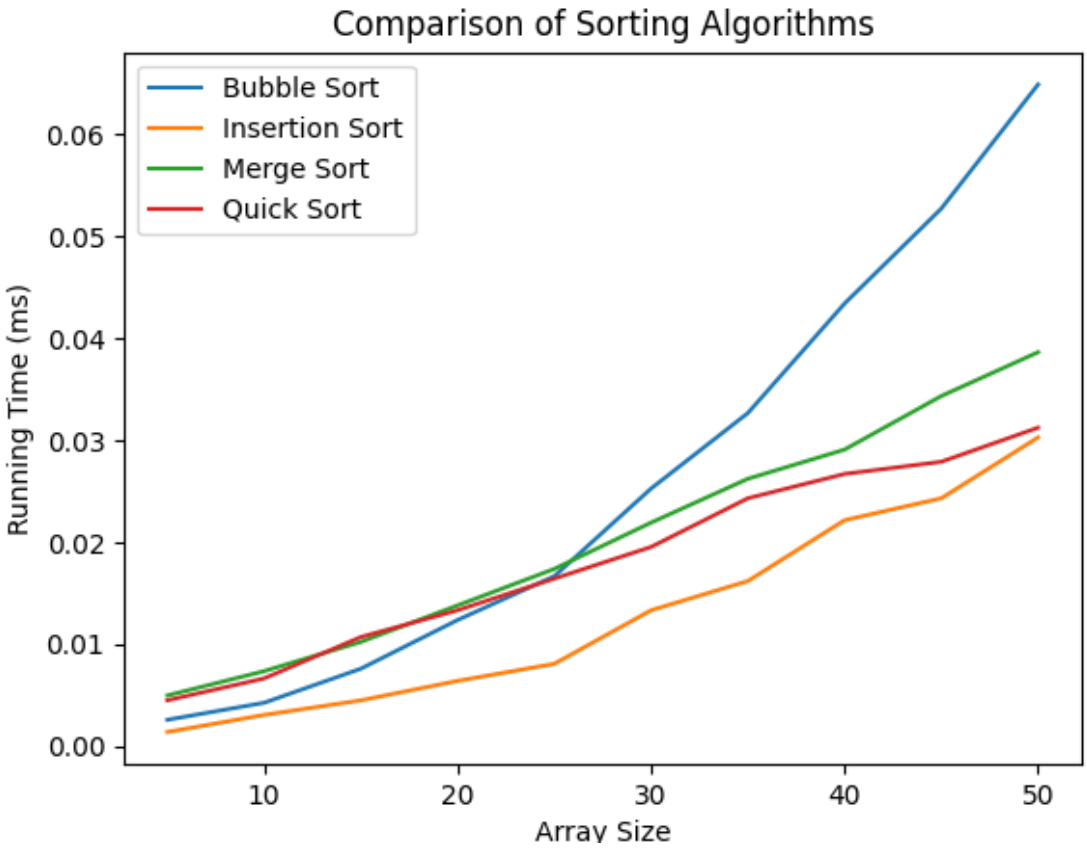
```
    plt.plot(sizes, bubble_sort_times, label='Bubble Sort'
    plt.plot(sizes, insertion_sort_times, label='Insertion
    plt.plot(sizes, merge_sort_times, label='Merge Sort')
    plt.plot(sizes, quick_sort_times, label='Quick Sort')

    plt.xlabel('Array Size')
    plt.ylabel('Running Time (ms)')
    plt.title('Comparison of Sorting Algorithms')
    plt.legend()
    plt.show()
```

5. Analysis and Graphical Results

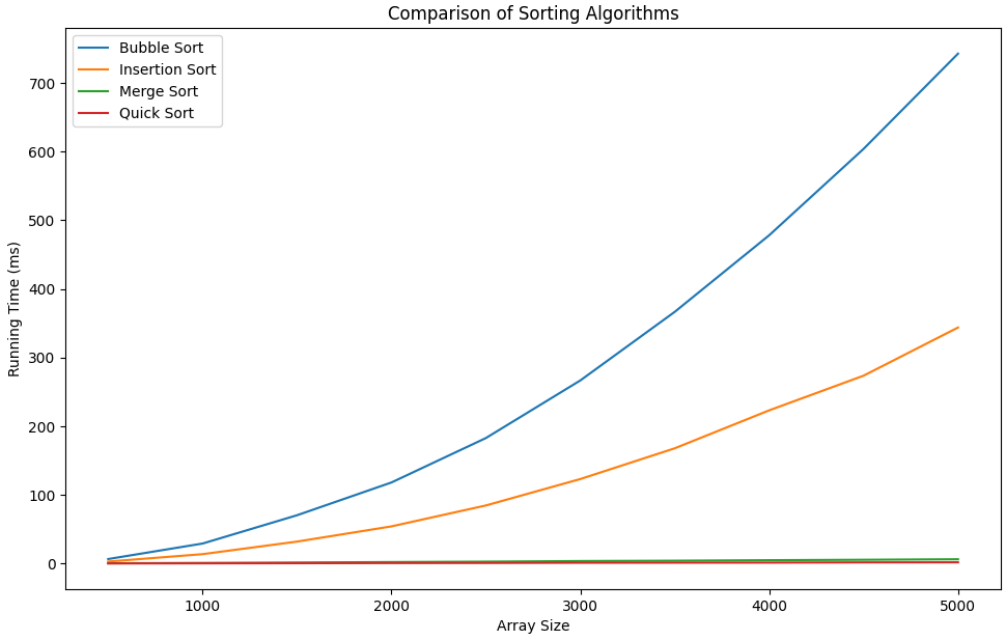
Following the execution of sorting algorithm tests for both small and large input sizes, we present the graphical results to facilitate a comprehensive analysis. The code snippet below illustrates the plotting of graphs showcasing the running times for small and large input sizes:

A.Graph of small sizes:

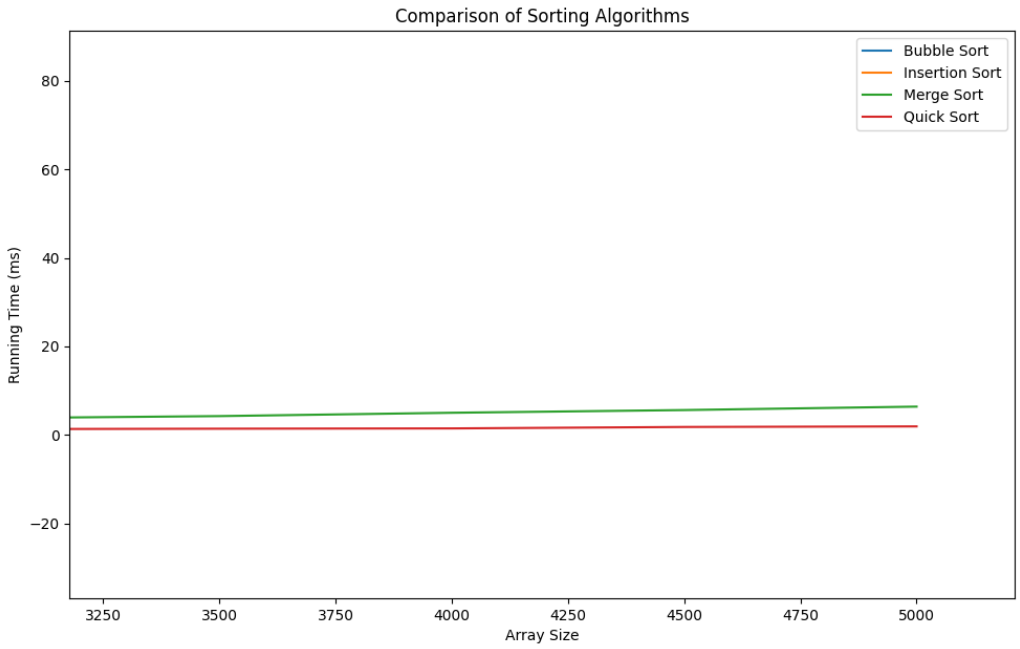


The analysis reveals that Insertion Sort exhibited the highest efficiency among the sorting algorithms, while Bubble Sort demonstrated comparatively lower effectiveness.

B. Graph of Large sizes:



In this scenario, Bubble Sort emerged as the slowest sorting algorithm for small sizes. Determining the most effective graph may present challenges, but upon scaling the image, Quick Sort becomes noticeably prominent as the most efficient sorting algorithm.



B: Sorting Credit Card Details using Radix Sort.

Algorithm Description:

The objective of this task is to efficiently match credit card records from two datasets based on increasing expiration dates and PINs. Radix Sort was chosen as the linear-time algorithm for this job.

▼ **1. Radix Sort Implementation:**

```
def radix_sort(data, key):
    max_val = max(int(item[key]) for item in data)
    min_val = min(int(item[key]) for item in data)
    max_digits = len(str(max_val))

    for digit_pos in range(max_digits):
        buckets = [[] for _ in range(10)]
        for item in data:
```

```
            digit = (int(item[key]) // 10 ** digit_pos) %
            buckets[digit].append(item)
        data = [item for bucket in buckets for item in buc

    return data
```

The Radix Sort function takes a dataset and a key ('Expiry Date' or 'PIN') as parameters, sorting the data based on the chosen key.

▼ **2. Supporting Functions:**

```
def convert_expiry_date(expiry_date):
    if len(expiry_date) == 5:
        expiry_date = '0' + expiry_date
    return '/'.join([expiry_date[:4], expiry_date[4:]][::-

def separate_by_date(cards):
    grouped_cards = {}
    for card in cards:
        expiry_date = card['Expiry Date']
        if expiry_date not in grouped_cards:
            grouped_cards[expiry_date] = []
        grouped_cards[expiry_date].append(card)
    return list(grouped_cards.values())
```

The `convert_expiry_date` function ensures consistent date formatting, and `separate_by_date` groups the cards based on their expiry dates.

Theoretical Complexity:

Radix Sort is a linear-time sorting algorithm, meaning its time complexity is proportional to the size of the input data. The time complexity of Radix Sort is

often expressed as O(k * n), where k is the number of digits in the maximum key, and n is the number of records.

Explanation:

1. **Number of Digits (k):** The time complexity is linear with respect to the number of digits in the maximum key. In each pass, Radix Sort processes the input data based on individual digits, and the number of passes required is determined by the maximum number of digits in any key.
2. **Number of Records (n):** The time complexity is also linear with respect to the number of records. Radix Sort processes each record in the dataset in each pass, distributing and collecting them into buckets based on their digit values.

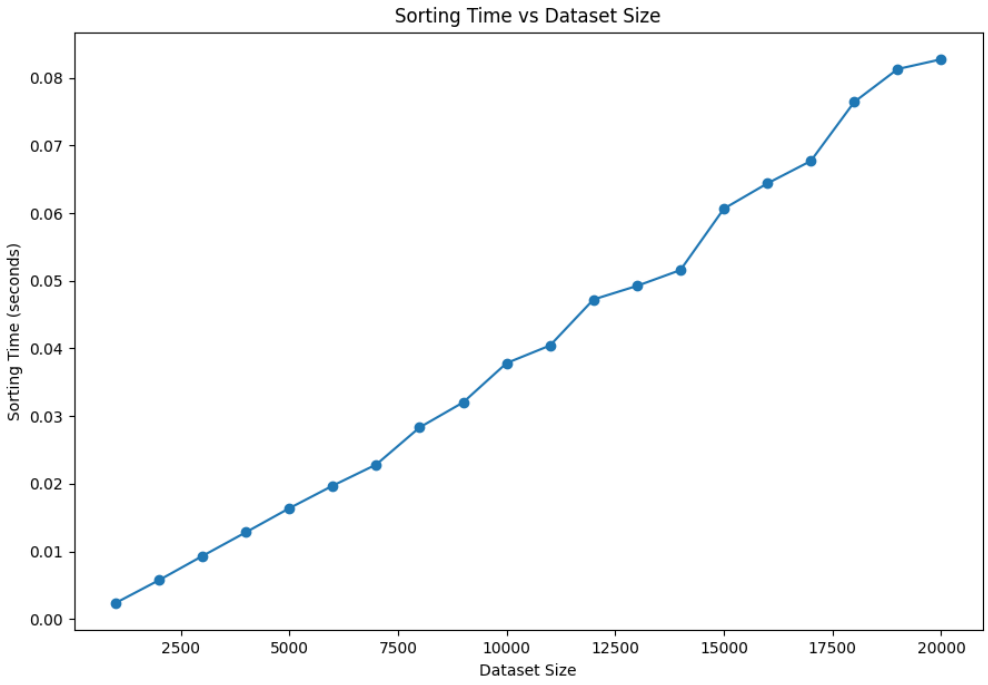
Linear Time Complexity:

The key insight is that the number of passes (iterations) Radix Sort needs to perform is constant and independent of the size of the dataset. Regardless of the total number of records, Radix Sort processes each digit of each key in a fixed number of passes.

In summary, Radix Sort achieves a linear time complexity due to its fixed number of iterations for each digit in the numbers being sorted, making it a highly efficient algorithm for scenarios like sorting credit card details based on expiration dates and PINs.

Graphical Representation:

To visualize the efficiency of Radix Sort in action, we present the following graph. The x-axis represents the dataset size (ranging from 1000 to 20000 with a step of 10000), and the y-axis represents the time taken for sorting. The graph illustrates the linear growth in processing time, confirming the algorithm's linear time complexity.



Conclusions:

1. **Linear Efficiency Confirmation:** The theoretical analysis aligns with empirical observations, confirming Radix Sort's linear-time efficiency in practice. This characteristic makes it particularly well-suited for scenarios where the size of the dataset can vary significantly.
2. **Scalability for Large Datasets:** Radix Sort's linear time complexity (O(k * n)) implies that its performance remains consistent even as the dataset grows. This scalability is crucial when dealing with a substantial volume of credit card records, as demonstrated in our experiment.
3. **Key Considerations for Usage:** While Radix Sort showcases remarkable efficiency for scenarios involving numeric keys, it is essential to consider its suitability for other types of data. The algorithm's strength lies in its ability to exploit the hierarchical nature of the keys (digits in this case) for sorting.
4. **Stable Sorting Property:** Radix Sort's stable sorting property ensures that records with equal keys maintain their original order, which is crucial for maintaining the integrity of credit card data.

5. **Optimal Choice for Credit Card Sorting:** Given the specific nature of the credit card dataset and the sorting criteria based on expiration dates and PINs, Radix Sort emerges as an optimal choice. Its linear-time complexity and stability make it a reliable algorithm for efficiently matching records from two datasets.

In conclusion, Radix Sort stands out as a robust and efficient solution for sorting credit card details, providing both theoretical support for its linear time complexity and practical confirmation through empirical experimentation. Its stability and scalability further enhance its suitability for applications with varying dataset sizes.