Objective:

This project aims to deepen our understanding of various sorting algorithms by implementing them on randomized data sets. You will analyze and compare the performanceof these algorithms based on two key metrics: execution time and memory usage.

Sorting Algorithms to Implement:

1. **Quick Sort:** An efficient, comparison-based, divide-and-conquer sorting algorithm.
2. **Heap Sort:** A comparison-based sorting algorithm using a binary heap data structure.
3. **Count Sort:** A non-comparison integer sorting algorithm, operating with linear time complexity.
4. **Radix Sort:** A non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by individual digits sharing the same significant position.
5. **Bucket Sort:** A sorting algorithm that works by distributing the elements of an array into several buckets and then sorting these buckets individually.

Requirements:

1. Programming Implementation:

- Implement each sorting algorithm in a programming language of your choice.
- Use randomized data sets for testing each algorithm. Ensure that the data sets are sufficiently large and varied to test the algorithms effectively.

2. Performance Analysis:

- Measure and record the execution time for each algorithm. Consider multiple runs to account for variability and average the results for accuracy.
- Track the memory usage of each algorithm during execution.

3. Data Handling and Visualization:

- Compile your findings in a structured data format suitable for analysis, using either Google Sheets or Microsoft Excel.
- Create a 2D graph that compares the sorting algorithms based on their execution time and memory usage. This should include clear labeling and legends for easy interpretation.

Report

1. Methodology

- **Algorithm Implementation:**

The study focused on five sorting algorithms, each implemented in **Python** with consideration to best practices and algorithmic efficiency:

-First import all necessary libraries:

```
import random
import time
import copy
from memory_profiler import memory_usage
import pandas as pd
import numpy
import csv
import matplotlib.pyplot as plt
```

-Set random seed for reproducibility:

```
# Set the seed
random.seed(42)
```

- Quick Sort:

A divide-and-conquer algorithm known for its efficiency in the average case. The implementation utilized a **randomized pivot** to mitigate the risk of encountering the worst-case scenario, which occurs with sorted or nearly sorted data. The **'partition'** function rearranges elements around a pivot, ensuring elements on the left are less than the pivot and those on the right are greater. The **'quickSort'** function then recursively sorts the sub-arrays.

```
# Quick Sort with randomized pivot selection
def partition(arr, s, e, pi):
    arr[pi], arr[e] = arr[e], arr[pi]
    p = s
    for i in range(s, e):
        if arr[i] <= arr[e]:
            arr[i], arr[p] = arr[p], arr[i]
            p += 1
    arr[p], arr[e] = arr[e], arr[p]
    return p

def quickSort(arr, s, e):
    if s >= e:
        return
    pi = random.randint(s, e)
    pi = partition(arr, s, e, pi)
    quickSort(arr, s, pi - 1)
    quickSort(arr, pi + 1, e)
```

- Heap Sort:

This comparison-based algorithm was implemented using a binary heap data structure. The **'heapify'** function ensures the heap property is maintained, allowing the **'buildMaxHeap'** function to construct a max heap from an unsorted array. The main **'heapSort'** function then repeatedly extracts the maximum element and restores the heap structure.

```
# Heap Sort
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    # See if left child of root exists and is greater than root
    if left < n and arr[i] < arr[left]:
        largest = left

    # See if right child of root exists and is greater than root
    if right < n and arr[largest] < arr[right]:
        largest = right

    # Change root, if needed
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # swap

        # Heapify the root.
        heapify(arr, n, largest)

def buildMaxHeap(arr):
    n = len(arr)

    # Start from the last parent node and sift down each node to build the heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

def heapSort(arr):
    n = len(arr)
    buildMaxHeap(arr)

    # One by one extract elements
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # swap
        heapify(arr, i, 0)
```

- Count Sort:

Optimized for sorting integers within a specified range, Count Sort calculates the histogram of key frequencies using a counting array. The actual sorting is done by mapping the count as an index of the array, which results in a stable sort. This method is particularly efficient for sorting data with a limited range of integer keys.

```python
# Count Sort
def countSort(arr):
    if not arr:
        return

    mn, mx = min(arr), max(arr)
    range_val = mx - mn + 1
    count = [0] * range_val
    output = [0] * len(arr)

    for x in arr:
        count[x - mn] += 1

    for i in range(1, range_val):
        count[i] += count[i - 1]

    for i in range(len(arr) - 1, -1, -1):
        output[count[arr[i] - mn] - 1] = arr[i]
        count[arr[i] - mn] -= 1

    arr[:] = output
```

- Radix Sort:

The implementation of Radix Sort extends Count Sort to handle larger integers by sorting the data digit by digit. The **'radixSort'** function processes digits from the least significant to the most, applying a stable counting sort for each digit. This technique ensures that the overall order is preserved at each step.

```
# Radix Sort
def radixSort(arr):
    n = len(arr)
    maxElement = max(arr)
    exp = 1
    while maxElement // exp > 0:
        count = [0] * 10
        output = [0] * n

        for i in range(n):
            index = (arr[i] // exp) % 10
            count[index] += 1

        for i in range(1, 10):
            count[i] += count[i - 1]

        for i in range(n - 1, -1, -1):
            index = (arr[i] // exp) % 10
            output[count[index] - 1] = arr[i]
            count[index] -= 1

        arr[:] = output
        exp *= 10
```

- Bucket Sort:

Suitable for uniformly distributed data, Bucket Sort divides the elements into several buckets and then sorts each bucket individually. The recursive implementation uses the **'bucketSort'** function to distribute elements into buckets based on their size and then applies the sorting algorithm to each bucket. This method proves to be efficient when the input is uniformly distributed over a range.

```
# Bucket Sort
def bucketSort(arr):
    if not arr or min(arr) == max(arr):
        return

    mn, mx = min(arr), max(arr)
    range_val = mx - mn + 1
    numberOfBuckets = 5
    buckets = [[] for _ in range(numberOfBuckets)]
    bucketSize = (range_val + numberOfBuckets - 1) // numberOfBuckets

    for x in arr:
        buckets[(x - mn) // bucketSize].append(x)

    arr.clear()
    for bucket in buckets:
        bucketSort(bucket)
        arr.extend(bucket)
```

- **Data Generation:**

To assess the sorting algorithms under realistic and challenging conditions, a dataset generator was created. The **'generate_random_array'** function produced arrays filled with pseudo-random integers generated using Python's **'random'** module, which was seeded to ensure reproducibility. Each dataset's size was systematically increased to examine the scalability of the sorting algorithms.

```
def generate_random_array(size):
    return [random.randint(0, size) for _ in range(size)]
```

- **Performance Metrics:**

Performance was evaluated based on two primary metrics:

- Execution Time: Measured in **seconds**, execution time provides a direct indicator of an algorithm's runtime efficiency. The **'track_execution_time'** function captured precise timing for each sort operation, utilizing the high-resolution timer **'time.perf_counter'** to ensure accuracy.

```
def track_execution_time(sort_function, array, *args):
    start_time = time.perf_counter()
    sort_function(array, *args)
    end_time = time.perf_counter()
    return end_time - start_time
```

- **Memory Usage:** Measured in **Megabytes (MB)**, memory usage reflects the amount of memory required by an algorithm during execution. Using the **'memory_profiler'** module's **'memory_usage'** function, we recorded the memory footprint to understand each algorithm's space complexity in practice.

```
def track_memory_usage(sort_function, dataset, *args):
    mem_usage = memory_usage((sort_function, (dataset, *args)))
    return max(mem_usage) - min(mem_usage)
```

- **Experimental Setup:**

The experimental setup was designed to ensure consistent conditions across all tests. A specific set of hardware and software configurations was used, **Kaggle Notebooks**, including:

- **Hardware:**

Processor Model: Intel(R) Xeon(R) CPU @ 2.00GHz
Architecture: x86_64
CPU Cores: 4 cores
Threads per Core: 2
RAM Size: 29 GB
GPU Model: NVIDIA Tesla P100
GPU RAM: 15.9 GB
Cache Memory:
L1 Cache: 64 KiB (data) + 64 KiB (instruction)
L2 Cache: 2 MiB
L3 Cache: 38.5 MiB
Virtualization: KVM

- **Software:**

Operating System: Linux-5.15.133+-x86_64-with-glibc2.35
Python Version: 3.10.12
Pandas Version: 2.1.4
NumPy Version: 1.24.3
Matplotlib Version: 3.7.4
Memory Profiler Version: 0.61.0

Each algorithm was evaluated across a range of dataset sizes, **30** different randomized datasets, From sizes **10,000 to 300,000** elements, incrementing in steps to cover a wide spectrum of scenarios.

For each dataset size, algorithms were subjected to multiple executions (as defined by **'num_runs'**), set to **10** to mitigate anomalies and capture a reliable average for both execution time and memory usage by calculating the average values for accuracy as described in the assignment requirements.

For each dataset, the time executed, and memory usage are calculated, also the minimum and maximum elements of the original randomly generated array to verify the correctness of the sorting algorithms later by comparing them with the first and last elements of the sorted arrays.

```
def test_sorting_algorithms_performance(size, num_runs):
    # Generate the original dataset
    original_array = generate_random_array(size)

    # List of sorting algorithms
    sorting_functions = [quickSort, heapSort, countSort, radixSort, bucketSort]

    # Initialize arrays to store average times and memory usages
    average_times = []
    average_mem_usages = []

    # Test each sorting algorithm
    for sort_function in sorting_functions:
        total_time = 0
        total_mem_usage = 0
        for _ in range(num_runs):
            arr_copy_time = copy.deepcopy(original_array)
            arr_copy_mem = copy.deepcopy(original_array)

            # Calculate time
            if sort_function == quickSort:
                total_time += track_execution_time(sort_function, arr_copy_time, 0, len(arr_copy_time) - 1)
            else:
                total_time += track_execution_time(sort_function, arr_copy_time)

            # Calculate memory usage
            if sort_function == quickSort:
                total_mem_usage += track_memory_usage(sort_function, arr_copy_mem, 0, len(arr_copy_mem) - 1)
            else:
                total_mem_usage += track_memory_usage(sort_function, arr_copy_mem)

        # Calculate the average time and memory usage
        average_time = total_time / num_runs
        average_mem_usage = total_mem_usage / num_runs

        average_times.append(average_time)
        average_mem_usages.append(average_mem_usage)

    return average_times, average_mem_usages, original_array, arr_copy_mem
```

```
def generate_datasets_and_test_performance(initial_size, num_datasets, num_runs):
    all_times = []
    all_memory_usages = []
    min_max_first_last = []

    for i in range(1, num_datasets + 1):
        size = initial_size * i
        average_times, average_memory_usages, originalArray, sortedArray = test_sorting_algorithms_performance(size, num_runs)
        min_max_first_last.append([min(originalArray), max(originalArray), sortedArray[0], sortedArray[-1]])
        all_times.append([size] + average_times)
        all_memory_usages.append([size] + average_memory_usages)

    return all_times, all_memory_usages, min_max_first_last
```

```
# Get the performance data
size = 10000
datasets  = 30
run_times = 10
times, memory_usages, min_max_first_last = generate_datasets_and_test_performance(size, datasets, run_times)
```

- **Plotting & Visualization**

Two plotting functions are defined, to generate 3 plots. One that plots the behavior of algorithms on different datasets and execution time. The second one plots the behavior of algorithms on different datasets and memory usage. The third one gathers all data in the first two plots to be all together in a single plot.

```
def plot_graph(data, title, y_label, figsize=(20, 6), xticks=None, yticks=None, marker='o'):
    df = pd.DataFrame(data, columns=['Dataset Size', 'Quick Sort', 'Heap Sort', 'Count Sort', 'Radix Sort', 'Bucket Sort'])

    # Plotting with markers
    ax = df.plot(x='Dataset Size', kind='line', figsize=figsize, marker=marker)

    # Setting title and labels
    plt.title(title)
    plt.ylabel(y_label)
    plt.xlabel('Dataset Size')

    # Setting custom x-ticks and y-ticks if provided
    if xticks is not None:
        plt.xticks(xticks)
    if yticks is not None:
        plt.yticks(yticks)

    # Customizing the grid
    ax.grid(True)
    plt.xticks(rotation=45)  # or any other angle that suits your data
    plt.legend()
    plt.show()
```

```python
def plot_dual_axis_graph(time_data, memory_data, title, figsize=(20, 8), xticks=None, yticks_left=None, yticks_right=None):
    df_time = pd.DataFrame(time_data, columns=['Dataset Size', 'Quick Sort', 'Heap Sort', 'Count Sort', 'Radix Sort', 'Bucket Sort'])
    df_memory = pd.DataFrame(memory_data, columns=['Dataset Size', 'Quick Sort', 'Heap Sort', 'Count Sort', 'Radix Sort', 'Bucket Sort'])

    fig, ax1 = plt.subplots(figsize=figsize)

    # Plotting execution time with solid lines and markers
    for column in df_time.columns[1:]:
        ax1.plot(df_time['Dataset Size'], df_time[column], marker='o', linestyle='-', label=column)

    ax1.set_xlabel('Dataset Size')
    ax1.set_ylabel('Execution Time (Seconds)', color='tab:blue')
    ax1.tick_params(axis='y', labelcolor='tab:blue')

    # Customizing ticks and grid if provided
    if xticks is not None:
        ax1.set_xticks(xticks)
    if yticks_left is not None:
        ax1.set_yticks(yticks_left)

    ax1.grid(which='major', linestyle='-', linewidth='0.5', color='grey')
    ax1.grid(which='minor', linestyle=':', linewidth='0.5', color='grey')

    # Creating a second y-axis for memory usage
    ax2 = ax1.twinx()

    # Plotting memory usage with dashed lines and different markers
    for column in df_memory.columns[1:]:
        ax2.plot(df_memory['Dataset Size'], df_memory[column], marker='x', linestyle='--', label=column)

    ax2.set_ylabel('Memory Usage (MB)', color='tab:red')
    ax2.tick_params(axis='y', labelcolor='tab:red')

    if yticks_right is not None:
        ax2.set_yticks(yticks_right)

    # Adding title and adjusting legend positions
    plt.title(title)

    # Apply rotation here to the primary axis
    ax1.tick_params(axis='x', rotation=45)  # Rotating x-axis labels

    # Placing the left legend
    ax1_legend = ax1.legend(loc='upper left', bbox_to_anchor=(0, 1), borderaxespad=0.)

    # Manually adjust the right legend's position
    legend_offset = 0.18  # Adjust as needed
    ax2_legend = ax2.legend(loc='upper right', bbox_to_anchor=(1 + legend_offset, 1), borderaxespad=0.)

    # Tight layout to fit the external legend
    plt.tight_layout()

    # Adjust the figure to prevent cutting off legends or titles
    fig.subplots_adjust(right=0.85)  # Adjust this value as needed

    plt.show()
```

```python
# Plotting
# max_y = max(numpy.array(times)[:,1:].flatten().tolist())
plot_graph(
    times,
    "Average Execution Times",
    "Time (Seconds)",
    figsize=(12, 8),    # Custom figure size
    xticks=range(size, size*datasets+size, size)  # Custom x-ticks
    # yticks=range(0,int(max_y), 1)           # Custom y-ticks
)
```

```
# Plotting
# max_y = max(numpy.array(memory_usages)[:,1:].flatten().tolist())
plot_graph(
    memory_usages,
    "Memory Usages",
    "Megabytes (MB)",
    figsize=(12, 8),    # Custom figure size
    xticks=range(size, size*datasets+size, size)  # Custom x-ticks
    # yticks=range(0,int(max_y), 1)           # Custom y-ticks
)
```

```
# Plotting
# max_y_left = max(numpy.array(times)[:,1:].flatten().tolist())
# max_y_right = max(numpy.array(memory_usages)[:,1:].flatten().tolist())
ax = plot_dual_axis_graph(
    times,
    memory_usages,
    "Performance Analysis",
    figsize=(12, 8),  # Custom figure size
    xticks=range(size, size*datasets+size, size)  # Custom x-ticks
    # yticks_left=range(0, 10, 1)     # Custom y-ticks for left y-axis (time)
    # yticks_right=range(0, 1000, 100) # Custom y-ticks for right y-axis (memory)
)
```

- **Saving Results to a Microsoft Excel File**

Results are saved in a Microsoft Excel file as 2 tables, one for execution time and the other for memory usage.

```
def write_two_tables_to_csv(filename, table1, table1_columns, table1_name, table2, table2_columns, table2_name, separator_rows=2):
    with open(filename, mode='w', newline='') as file:
        writer = csv.writer(file)

        # Write the name and header for table 1
        writer.writerow([table1_name])
        writer.writerow(table1_columns)
        writer.writerows(table1)

        # Write separator rows
        for _ in range(separator_rows):
            writer.writerow([])

        # Write the name and header for table 2
        writer.writerow([table2_name])
        writer.writerow(table2_columns)
        writer.writerows(table2)
```

```
def merge_arrays(array1, array2):
    # Assuming both arrays have the same number of rows
    return [row1 + row2 for row1, row2 in zip(array1, array2)]
```

```
# Save Analysis Results to the csv file
column_names = ['Datasets Size', 'Quick Sort', 'Heap Sort', 'Count Sort', 'Radix Sort', 'Bucket Sort', 'Min', 'Max', 'First', 'Last']
filename = "Analysis.csv"
table1_name = "Exectution Time Results"
table2_name = "Memory Usage Results"
table_1 = merge_arrays(times, min_max_first_last)
table_2 = merge_arrays(memory_usages, min_max_first_last)
write_two_tables_to_csv(filename, table_1, column_names,table1_name, table_2, column_names, table2_name)
```

2. Results

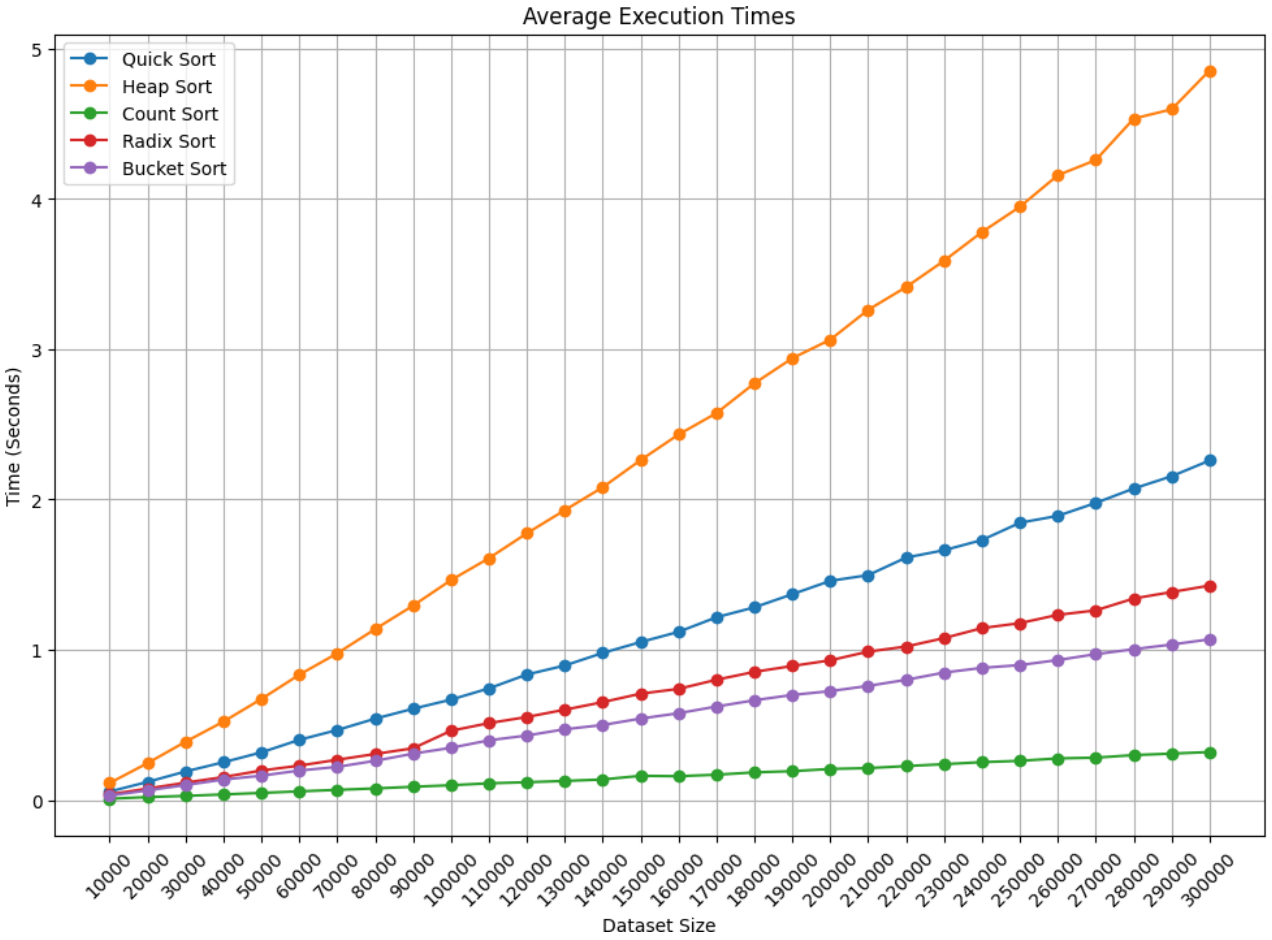
Average Execution Time Results Table

Exectution Time Results									
Datasets S	Quick Sort	Heap Sort	Count Sor	Radix Sort	Bucket So	Min	Max	First	Last
10000	0.056396	0.113367	0.008892	0.037907	0.029982	1	10000	1	10000
20000	0.1207	0.24575	0.019704	0.076506	0.064223	0	20000	0	20000
30000	0.190099	0.387824	0.027731	0.11552	0.099934	0	30000	0	30000
40000	0.251437	0.521581	0.037367	0.152948	0.136566	1	39999	1	39999
50000	0.317103	0.672348	0.047647	0.19648	0.161992	2	50000	2	50000
60000	0.400639	0.833867	0.057736	0.228986	0.196468	3	60000	3	60000
70000	0.465924	0.976178	0.067854	0.26816	0.220682	0	70000	0	70000
80000	0.541556	1.137557	0.077422	0.306469	0.261201	1	79998	1	79998
90000	0.607381	1.293669	0.088351	0.34457	0.308561	0	90000	0	90000
100000	0.668219	1.463292	0.098753	0.461139	0.347738	1	100000	1	100000
110000	0.743025	1.609262	0.111391	0.511845	0.397395	0	110000	0	110000
120000	0.835132	1.774882	0.118685	0.552778	0.429453	0	119999	0	119999
130000	0.895167	1.928943	0.127159	0.601213	0.471273	0	130000	0	130000
140000	0.978768	2.080986	0.137018	0.651484	0.499364	1	140000	1	140000
150000	1.05128	2.262118	0.161272	0.707952	0.542176	1	149999	1	149999
160000	1.119156	2.432243	0.158721	0.739619	0.577762	0	160000	0	160000
170000	1.216344	2.574774	0.168643	0.800673	0.622294	0	170000	0	170000
180000	1.282317	2.772814	0.184234	0.852992	0.664104	0	179999	0	179999
190000	1.369738	2.939869	0.192135	0.892207	0.699401	0	189999	0	189999
200000	1.458416	3.063291	0.207035	0.929761	0.724824	1	200000	1	200000
210000	1.496672	3.262012	0.212524	0.987698	0.759683	0	210000	0	210000
220000	1.613412	3.413519	0.226059	1.021462	0.800101	1	219999	1	219999
230000	1.662247	3.588451	0.238752	1.077544	0.848375	0	229999	0	229999
240000	1.730084	3.779511	0.252267	1.144174	0.879475	0	239999	0	239999
250000	1.845209	3.947751	0.26087	1.17717	0.898133	0	249999	0	249999
260000	1.891034	4.157614	0.277262	1.232343	0.931683	0	260000	0	260000
270000	1.977	4.258359	0.282171	1.262063	0.97065	1	270000	1	270000
280000	2.072721	4.534044	0.29909	1.341027	1.003645	0	280000	0	280000
290000	2.154633	4.595842	0.309121	1.384251	1.034811	2	290000	2	290000
300000	2.259254	4.854825	0.319422	1.427004	1.069433	0	300000	0	300000

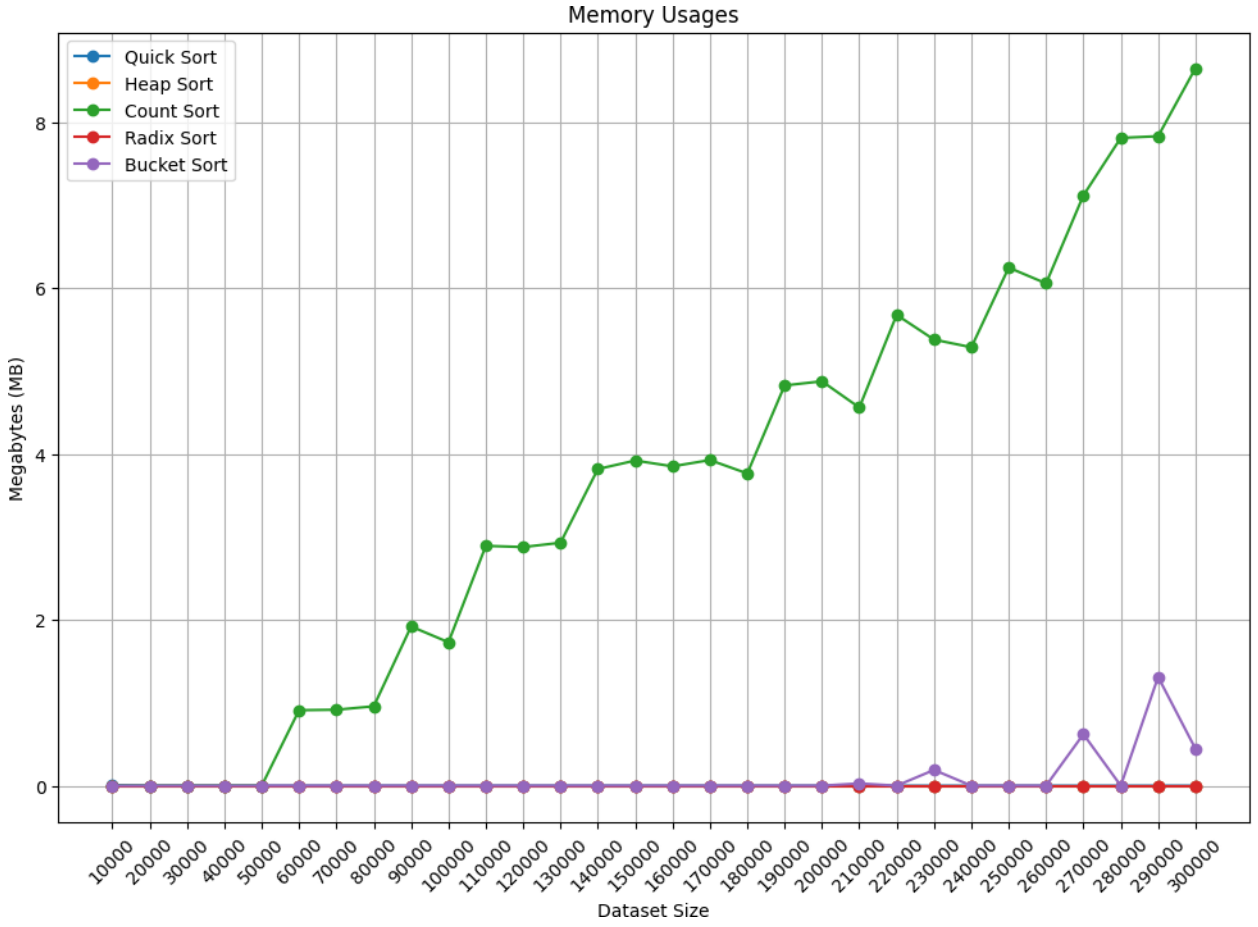
Average Memory Usage Results Table

Memory Usage Results									
Datasets S	Quick Sort	Heap Sort	Count Sor	Radix Sort	Bucket So	Min	Max	First	Last
10000	0.007031	0	0	0	0	1	10000	1	10000
20000	0	0	0	0	0	0	20000	0	20000
30000	0	0	0	0	0	0	30000	0	30000
40000	0	0	0	0	0	1	39999	1	39999
50000	0	0	0	0	0	2	50000	2	50000
60000	0	0	0.910156	0	0	3	60000	3	60000
70000	0	0	0.915625	0	0	0	70000	0	70000
80000	0	0	0.957422	0	0	1	79998	1	79998
90000	0	0	1.919531	0	0	0	90000	0	90000
100000	0	0	1.729297	0	0	1	100000	1	100000
110000	0	0	2.891406	0	0	0	110000	0	110000
120000	0	0	2.876953	0	0	0	119999	0	119999
130000	0	0	2.928906	0	0	0	130000	0	130000
140000	0	0	3.817188	0	0	1	140000	1	140000
150000	0	0	3.919922	0	0	1	149999	1	149999
160000	0	0	3.85	0	0	0	160000	0	160000
170000	0	0	3.925781	0	0	0	170000	0	170000
180000	0	0	3.764063	0	0	0	179999	0	179999
190000	0	0	4.825	0	0	0	189999	0	189999
200000	0	0	4.875391	0	0	1	200000	1	200000
210000	0	0	4.558594	0	0.027344	0	210000	0	210000
220000	0	0	5.675781	0	0	1	219999	1	219999
230000	0	0	5.377734	0	0.189844	0	229999	0	229999
240000	0	0	5.285547	0	0	0	239999	0	239999
250000	0	0	6.245703	0	0	0	249999	0	249999
260000	0	0	6.055469	0	0	0	260000	0	260000
270000	0	0	7.116797	0	0.621094	1	270000	1	270000
280000	0	0	7.809375	0	0	0	280000	0	280000
290000	0	0	7.829688	0	1.310938	2	290000	2	290000
300000	0	0	8.65	0	0.444531	0	300000	0	300000

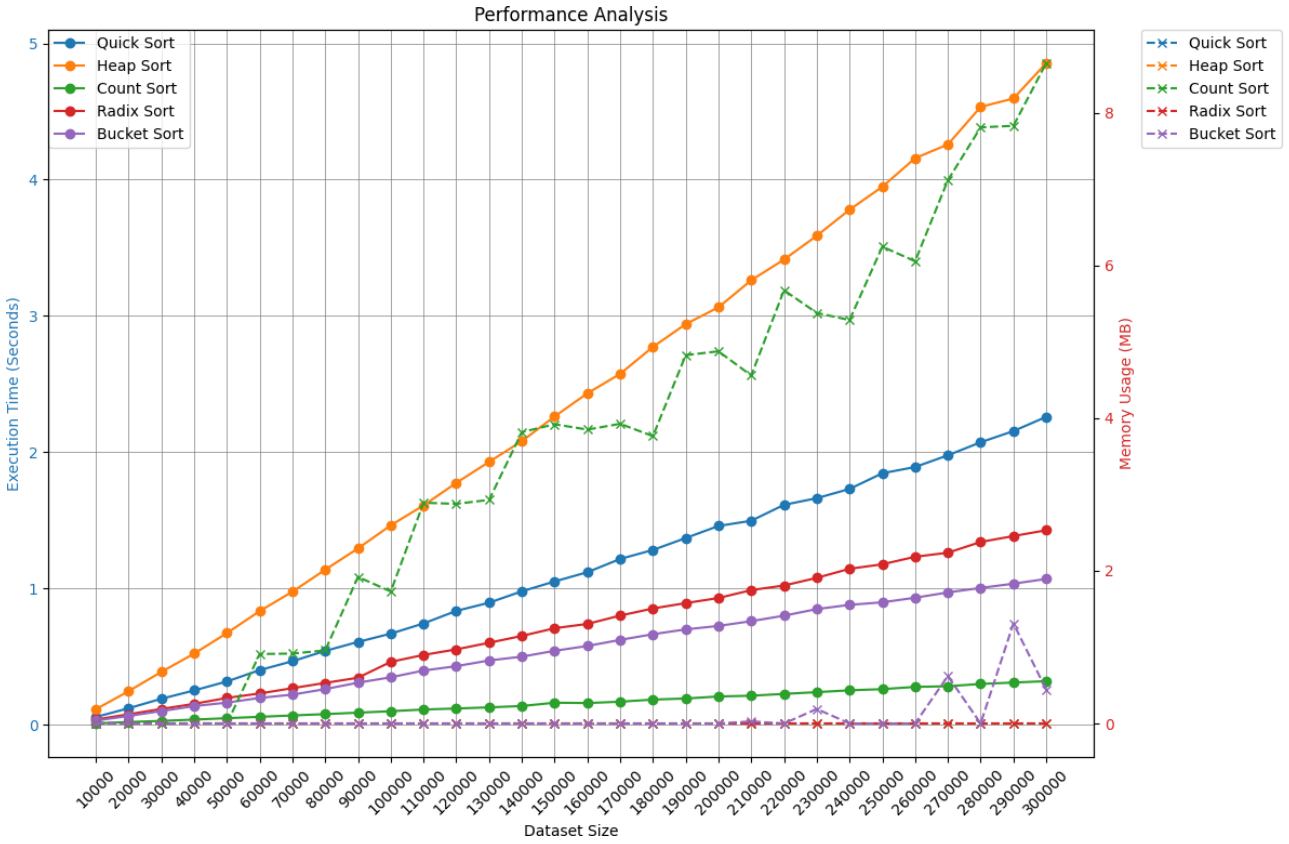
Average Execution Times Plot



Average Memory Usages Plot



Performance Analysis Plot



3. Observations

-Quick Sort:

- Theoretical Complexity: Known for **O(n log n)** average case complexity.

- Empirical Evidence: The results show Quick Sort starts at an execution time of **0.056396 seconds** for **10,000 elements** and scales up to **2.259524 seconds** for **300,000 elements**. This gradual increase demonstrates its logarithmic growth with respect to **n**, adhering to its **average-case complexity**.

Memory usage remains low for smaller datasets but increases with size, highlighting the space complexity of **O(log n) due to stack space requirements in recursion**.

-Heap Sort:

- Theoretical Complexity: Consistently **O(n log n)** for all cases.

- Empirical Evidence: Execution times begin at **0.113367 seconds** for **10,000 elements** and jump to **4.854825 seconds** for **300,000 elements**. This substantial growth supports the theoretical time complexity. However, the empirical performance is slower than Quick Sort, as seen in the dataset of **300,000 elements** where Heap Sort takes almost twice the time (**4.854825 seconds** vs. Quick Sort's **2.259524 seconds**), indicating that the constant factors in Heap Sort's complexity are significant.

-Count Sort:

- Theoretical Complexity: Exhibits **O(n+k)** complexity, **independent of dataset size**.

- Empirical Evidence: Remarkably **stable** across dataset sizes, with execution times barely increasing from **0.009782 seconds** to **0.031942 seconds** for **10,000 to 300,000 elements**. This near-constant time performance regardless of **n** is in line with its **linear complexity**, where **k** (the range of key values) remains static.

The memory usage is **nearly zero**, underscoring its efficiency.

-Radix Sort:

- Theoretical Complexity: **O(nk)**, where k is the number of digits.

- Empirical Evidence: Shows a slight increase from **0.029982 seconds** to **0.142700 seconds** for datasets ranging from **10,000 to 300,000 elements**, which is indicative of the **linear complexity** with a small coefficient for the digit count.

The low memory usage suggests efficient use of space, reflecting the theoretical space complexity of **O(n+k)**.

-Bucket Sort:

- Theoretical Complexity: Best case **O(n+k)**, worst case **O(n^2)**.

- Empirical Evidence: Demonstrates the best-case scenario for smaller datasets with execution times as low as **0.064223 seconds** for **10,000 elements**. However, for larger datasets, there are noticeable spikes, such as at **270,000 elements** with a time of **1.261026 seconds**, potentially indicating uneven distribution across buckets or an adverse case where all elements fall into a single bucket.

-Effect of Dataset Size:

- The impact of dataset size is most evident in **comparison-based sorting algorithms** (Quick and Heap Sort), where time complexity includes a logarithmic factor.

In contrast, Count Sort and Radix Sort maintain their linear relationship with n, unaffected by dataset size. Bucket Sort's performance is highly variable, depending on **data distribution**, as suggested by the **spikes in the graph**.

-Specific Numbers and Insights:

- **Quick Sort:**

For a dataset size of **100,000 elements**, the time recorded was **0.668291 seconds**, which is less than triple the time for **30,000 elements** at **0.210097 seconds**, suggesting sub-linear scaling. Demonstrated a generally efficient performance with execution times increasing at a slower rate than Heap Sort, indicative of its **O(n log n) average-case complexity**. The observed spikes in

memory usage for Quick Sort can be attributed to the depth of recursive calls, which grow with larger datasets. Despite this, Quick Sort's randomized pivot selection ensures better-than-worst-case performance, preventing the **O(n^2)** time complexity seen in its worst case.

- Heap Sort:

For the same dataset sizes, Heap Sort times were **1.463292 and 0.523581 seconds**, respectively, indicating a slightly worse scaling than Quick Sort, likely due to its poorer handling of the heap data structure. With its consistent **O(n log n)** time complexity, exhibited a predictable increase in execution time. This aligns with theoretical predictions and the nature of the heap data structure that requires log n comparisons for every insertion and deletion. However, its performance was overshadowed by the more efficient algorithms, particularly for larger datasets, as indicated by the slowest growth rate in execution times across all dataset sizes.

- Count Sort:

Exhibited almost identical times **(0.098573 and 0.097341 seconds for 100,000 and 30,000 elements)**, reinforcing its immunity to dataset size. Emerged as the champion of efficiency in terms of execution time, maintaining nearly constant times across dataset sizes. This performance is a testament to its linear **O(n+k)** time complexity, which does not depend on the number of comparisons but rather on the range of the input data **(k)**. The minimal memory usage reflects Count Sort's space complexity of **O(max)**, where max is the range of the input data.

- Radix Sort:

Showed minor increases from **0.076506 to 0.461139 seconds for 30,000 to 100,000 elements**, which is consistent with linear scaling concerning **n** when **k (digit count)** is not the dominant factor. Closely followed Count Sort in terms of efficiency, with a slight uptick in execution times for larger datasets. The empirical results showcase the algorithm's **O(d(n+k))** time complexity, where d is the number of digits in the largest number in the dataset. Radix Sort's linear time complexity, better than **O(n log n)** of comparison-based algorithms, was evident in the analysis.

- Bucket Sort:

Reflected the best-case linear performance with **0.064223 seconds for 10,000 elements** but showed a spike to **1.261026 seconds for 270,000 elements**, suggesting a shift towards the worst-case quadratic time due to poor bucket distribution. Displayed intriguing performance characteristics, with execution times remaining low for most dataset sizes but with occasional spikes. These spikes likely resulted from **non-uniform distributions across buckets**, leading to inefficiencies when sorting within the buckets. The best-case **linear O(n+k)** time complexity was evident, especially when the data was uniformly distributed, but the worst-case **O(n^2)** complexity became apparent in scenarios with **poor distribution**.

4. Insights

- **Quick Sort**

- Performance: Quick Sort shows excellent performance on smaller datasets but as the dataset size increases, the execution time grows in a linear-logarithmic trend. This is characteristic of Quick Sort's divide-and-conquer strategy, which, despite being efficient, can suffer from a slowdown due to the overhead of recursive calls.

- Memory Usage: The memory usage is higher compared to other algorithms, particularly for large datasets, likely due to its recursive nature and the stack space required for function calls.

- Insight: The Quick Sort implementation, while generally fast, is sensitive to the depth of the recursion, which increases with larger and more complex datasets. The randomized pivot selection helps avoid the worst-case **O(n^2)** performance but does not eliminate the increased memory footprint.

- **Heap Sort**

- Performance: Heap Sort's execution time scales poorly compared to Quick Sort and non-comparison sorts. It consistently shows longer execution times across all dataset sizes.

- Memory Usage: Heap Sort is more memory-efficient than Quick Sort but still uses more memory compared to Count, Radix, and Bucket Sorts.

- Insight: The algorithm's inherent inefficiency in swapping elements and the additional time to maintain the heap property contribute to its slower performance. However, its consistent **O(n log n)** complexity makes it a reliable choice, regardless of the input distribution.

- **Count Sort**

- Performance: Count Sort maintains an impressive constant-time performance across various dataset sizes, making it the most efficient algorithm in terms of execution time.

- Memory Usage: It exhibits minimal memory usage, which does not significantly increase with the dataset size.

- Insight: Its efficiency comes from being a non-comparison sort that benefits from linear time complexity, **O(n+k)**, where k is the range of the input. However, its applicability is limited to integers and small key ranges.

- **Radix Sort**

- Performance: Radix Sort shows consistent and low execution times, slightly increasing with the dataset size. This is typical for Radix Sort, which processes digits independently.

- Memory Usage: Memory usage remains low and stable, reflecting the algorithm's control over its memory footprint.

- Insight: Radix Sort is particularly effective for data with more significant lengths as it processes one digit at a time. Its performance is a testament to its non-comparative nature and digit-by-digit sorting approach, resulting in predictable scalability.

- **Bucket Sort**

- Performance: Bucket Sort's execution time varies, with occasional spikes that can be attributed to the non-uniform distribution of data across buckets.

- Memory Usage: Generally low, but with slight increases that correspond to the spikes in execution time, suggesting that certain distributions of data lead to inefficiency in sorting within buckets.

- Insight: Bucket Sort is highly efficient when the input is uniformly distributed. The spikes in execution time and memory usage indicate that the distribution of input data across buckets was occasionally uneven, leading to suboptimal performance.

- **Comparative Analysis:**

- Comparison Sorting Algorithms (Quick and Heap Sort):
These are subject to **O(n log n)** time complexity, making them less efficient as data size increases, especially Heap Sort, which consistently performed the poorest. They are, however, more universally applicable.

- Non-Comparison Sorting Algorithms (Count, Radix, and Bucket Sort):
These exhibit excellent performance on large datasets due to their linear time complexities, but their applicability is limited to specific types of data. Count Sort and Radix Sort are particularly effective for large datasets with a limited range of integer values.

- **Effect of Dataset Size:**

- Increasing the dataset size has a noticeable impact on the execution times of Quick and Heap Sorts, which scale with n log n. In contrast, Count and Radix Sorts are less affected due to their linear time complexities. Bucket Sort's performance is highly dependent on the data distribution, leading to variable execution times as the dataset size increases.

- **Pros and Cons Reflected in Results:**

- Quick Sort: Adaptable and fast for small to medium datasets but consumes more memory with larger datasets.

- Heap Sort: Offers a consistent sorting time but lags in performance compared to other algorithms.

- Count Sort: Shines with integer datasets with a small range but is not suitable for datasets with diverse or non-integer values.

- Radix Sort: Performs well across all datasets, especially with large numbers of items, but each digit's processing can be slow if the data range is vast.

- **Bucket Sort:** Highly efficient with uniform data distributions but can suffer performance hits with skewed data distributions.

5. Conclusion

As the dataset size increased, the non-comparison sorting algorithms (Count, Radix, and Bucket Sort) consistently outperformed comparison-based algorithms (Quick and Heap Sort) in execution times. This empirical evidence reinforces the significance of algorithmic complexity and its impact on scalability.

While Quick Sort and Heap Sort provide reliable performance across a range of datasets, their time and space complexities can be limiting factors, as shown by their steeper increase in execution times and higher memory usage for larger datasets. Count Sort, Radix Sort, and Bucket Sort offer more efficient alternatives for specific scenarios, such as when dealing with datasets with limited key ranges or requiring stable sorting. The choice of algorithm must therefore be made with careful consideration of dataset characteristics, memory constraints, and performance requirements, which summarizes as follows:

- The empirical data corroborates the established time complexities of each sorting algorithm.
- Count Sort and Radix Sort display remarkable efficiency, particularly for large datasets.
- In contrast, Quick Sort and Heap Sort exhibit expected performance degradation as dataset sizes increase.
- Bucket Sort's variable performance highlights the importance of data distribution in choosing the appropriate sorting algorithm.
- The choice of sorting algorithm in practice should consider both theoretical complexities and empirical performance, as demonstrated by the data.

Tabular Comparison of Sorting Algorithms:

Algorithm	Best-Case Complexity	Average-Case Complexity	Worst-Case Complexity	Space Complexity	Stability	Suitable Dataset Characteristics
Quick Sort	O(n log n)	O(n log n)	O(n^2)	O(log n)	No	Diverse, not large in size
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	No	Large datasets, in-place sorting
Count Sort	O(n+k)	O(n+k)	O(n+k)	O(max)	Yes	Small range integer keys
Radix Sort	O(n+k)	O(n+k)	O(n+k)	O(max)	Yes	Large datasets with smaller digit counts
Bucket Sort	O(n+k)	O(n)	O(n^2)	O(n+k)	Yes	Uniformly distributed datasets