Sorting Algorithm Performance Analysis

This presentation delves into the world of sorting algorithms, comparing their performance based on key metrics and providing insightful visualizations.

This Report is Made By :

Ayush Kumar (2023CSB114)

Dataset Generation

To analyze the performance of various sorting algorithms, we need a diverse set of datasets to test them against.

We generate large datasets of random integers, ranging from 1 to 10,000, with varying distributions.

We explore various data distributions, such as uniform, skewed, and nearly sorted, to ensure a comprehensive analysis.

These different datasets provide varying input conditions, allowing us to assess how algorithms cope with different data characteristics.

Sorting Algorithms Overview

Bubble Sort

A simple, but inefficient, algorithm that repeatedly compares adjacent elements and swaps them if they are out of order. It's best suited for small datasets.

Insertion Sort

Builds a sorted array one element at a time, efficiently handling small datasets. It inserts each element into its correct position in the already sorted portion of the array.

Selection Sort

Finds the minimum element in the unsorted portion of the array and swaps it with the first element. It repeats this process until the entire array is sorted.

Merge Sort

Utilizes a divide-and-conquer approach, recursively dividing the array into smaller subarrays, sorting them, and then merging them back together.

Quick Sort

A highly efficient algorithm for large datasets, it partitions the array around a pivot element, recursively sorting the subarrays.

Heap Sort

Leverages a binary heap data structure to efficiently sort data, providing both time and space complexity advantages.

Performance Metrics

Number of Comparisons

This metric tracks the number of times elements are compared during the sorting process. It directly reflects the algorithm's computational effort.

Number of Swaps/Assignments

The number of times elements are swapped or assigned to new positions during the sorting process. It indicates the algorithm's data movement efficiency.

Execution Time

This measures the actual time taken for the algorithm to complete the sorting process. It provides a practical assessment of the algorithm's performance.

Experimental Setup

We conduct experiments by running each sorting algorithm multiple times on different datasets.
Each algorithm is run on a variety of datasets of different sizes and distributions, ensuring a comprehensive evaluation.

We carefully control for factors like hardware, software, and environment to ensure fair comparisons between algorithms.
This rigorous setup helps minimize the impact of external factors and isolate the algorithm's intrinsic performance.

CSV Files Used:
Uniform.csv: Contains data for 10 arrays of fixed size 1k with values ranging from 1 to 10k. This dataset was used for static analysis.
Varying.csv: Contains data for varying array sizes from 1k to 52k (with 1k increments in size), providing dynamic analysis.
Box.csv: Contains data for 40 different arrays of size 20k, used for box plot analysis.

Static Analysis (Uniform.csv) : Table Summary, Pie Charts, Bar Charts, Heatmaps, Radar Charts
Dynamic Analysis (Varying.csv) : Line Graphs ,Scatter Plots
Distribution Analysis (Box.csv) : Box Plots

Data Analysis - Table Summary

Algorithm	Mean Comparisons	Min Comparisons	Max Comparisons	Mean Swaps	Min Swaps	Max Swaps	Mean Time (µs)	Min Time (µs)	Max Time (µs)
Bubble Sort	498701.1	497960	499200	249175.3	242268	257308	2872.1	2362	3256
Heap Sort	11692.8	11597	11755	9091.8	9007	9130	193.3	161	221
Insertion Sort	249175.3	242268	257308	249175.3	242268	257308	1876.4	1499	2137
Merge Sort	8702.2	8669	8727	9976	9976	9976	380.7	291	476
Quick Sort	11232.7	10739	12353	2379.4	2315	2415	106	92	132
Selection Sort	499500	499500	499500	992.6	988	997	1479.9	1272	1686

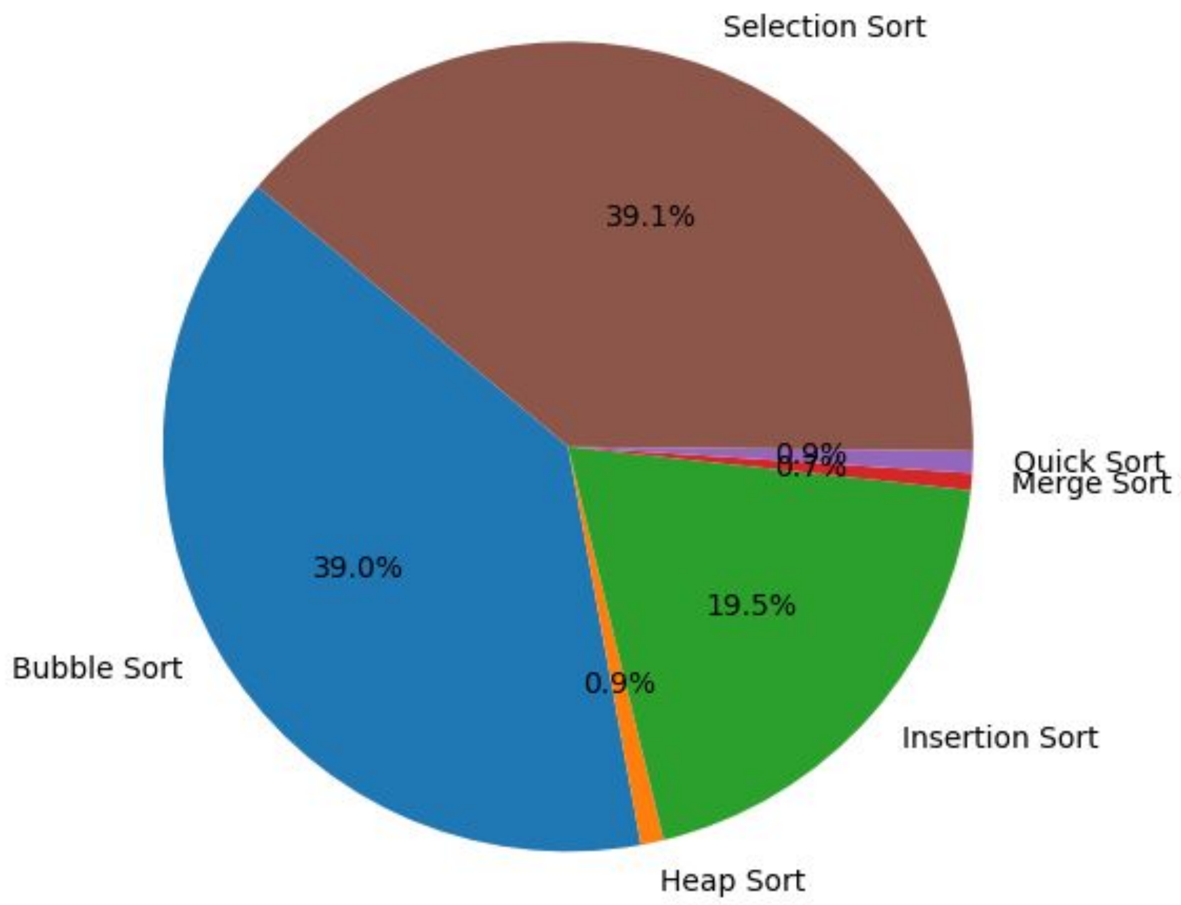
Things to Notice from Table :-

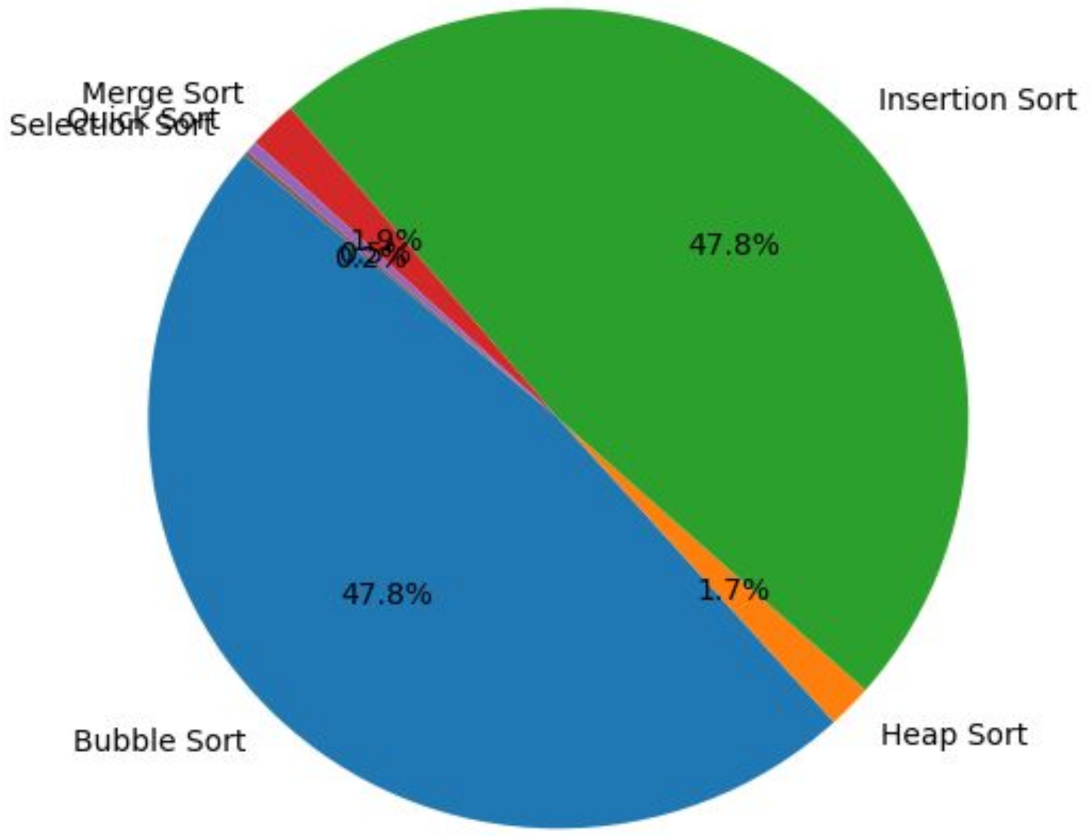
- **Quick Sort:** Excels with the lowest number of comparisons and swaps, and the fastest execution time.
- **Merge Sort:** Consistent performance with a moderate number of operations and execution time.
- **Heap Sort:** Good balance between comparisons and swaps, offering efficient performance.
- **Insertion Sort:** Performs well for smaller datasets (based on the limited data).
- **Bubble Sort & Selection Sort:** High number of operations (comparisons or swaps), leading to slower execution times.

Points

- **Quick Sort** appears to be the most efficient algorithm in this comparison.
- **Bubble Sort and Selection Sort** generally exhibit poorer performance due to their higher operational overhead.

Data Visualization - Pie Charts

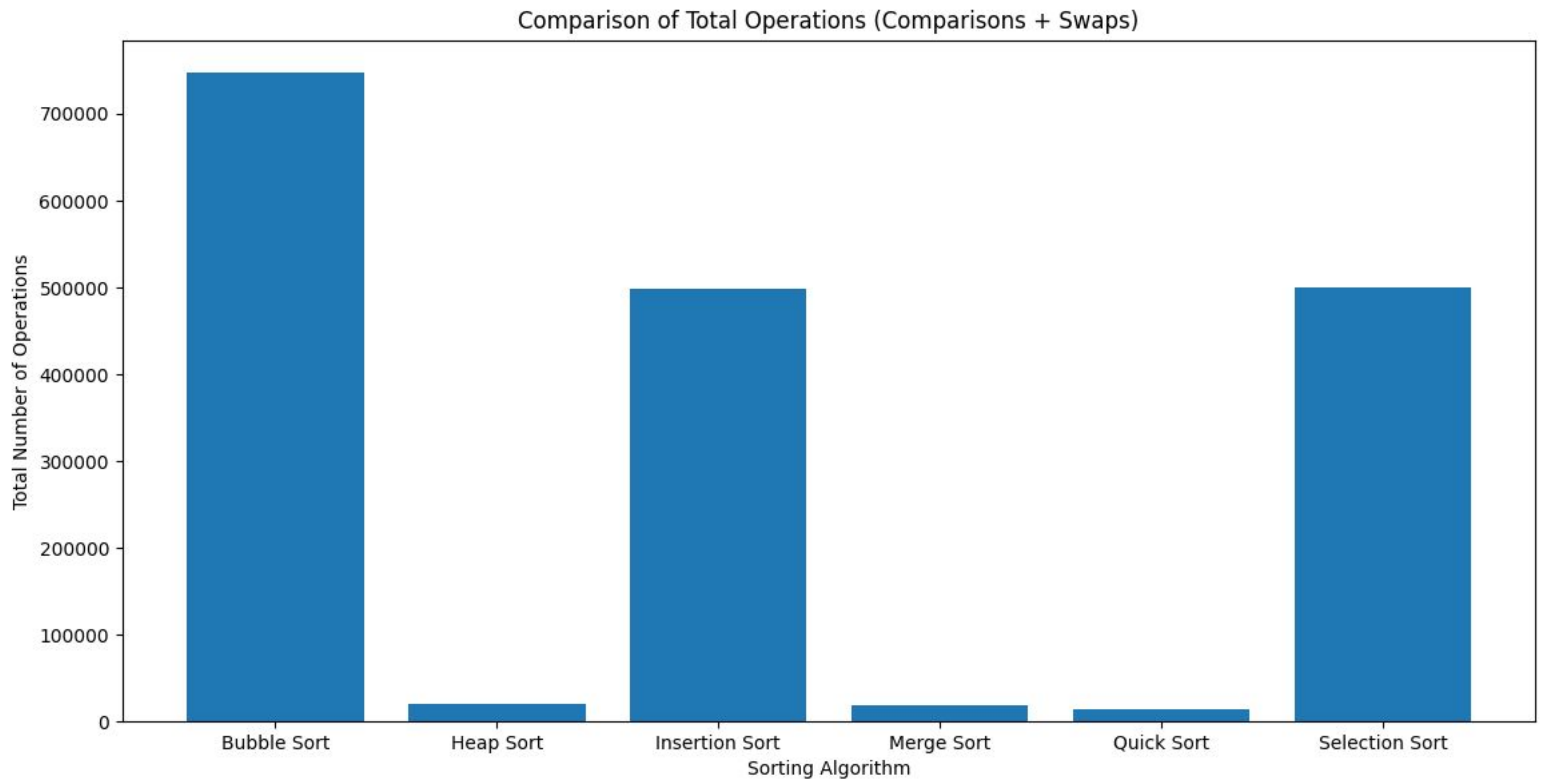




The first pie chart visually represents the proportion of comparisons for each algorithm.
This chart provides an immediate understanding of which algorithms require more comparisons to complete the sorting process.

The second pie chart shows the proportion of swaps/assignments for each algorithm.
This chart reveals the algorithms with higher data movement, highlighting those with less efficient element repositioning.

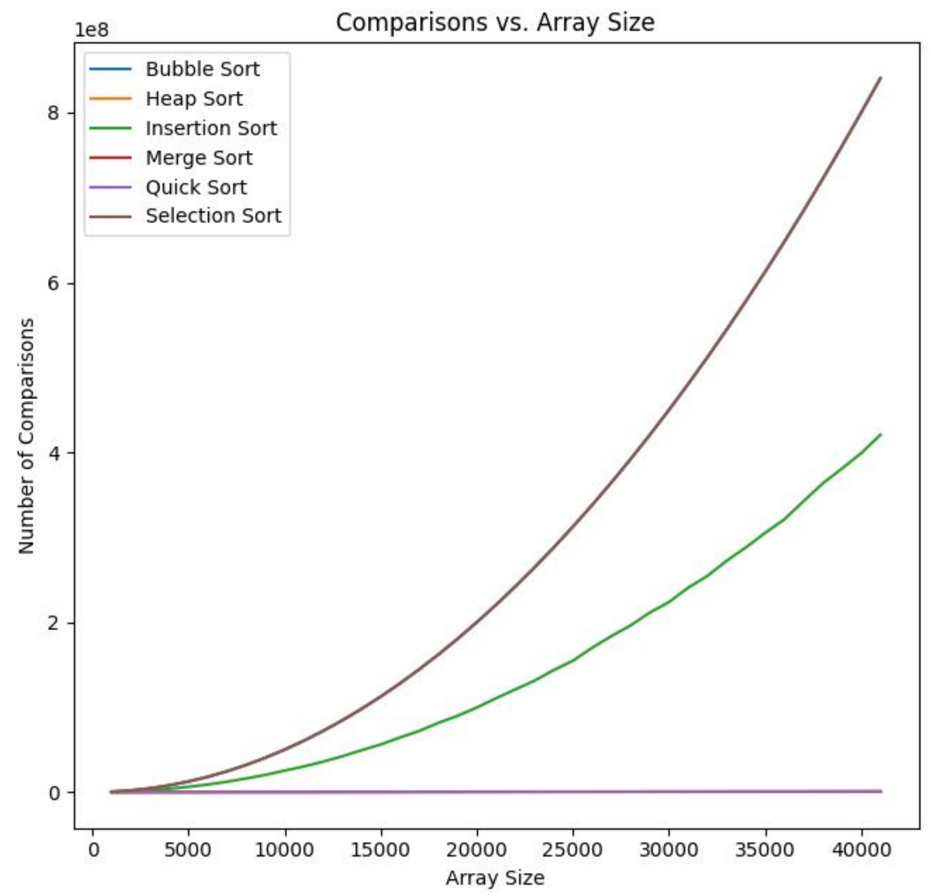
Data Visualization - Bar Charts

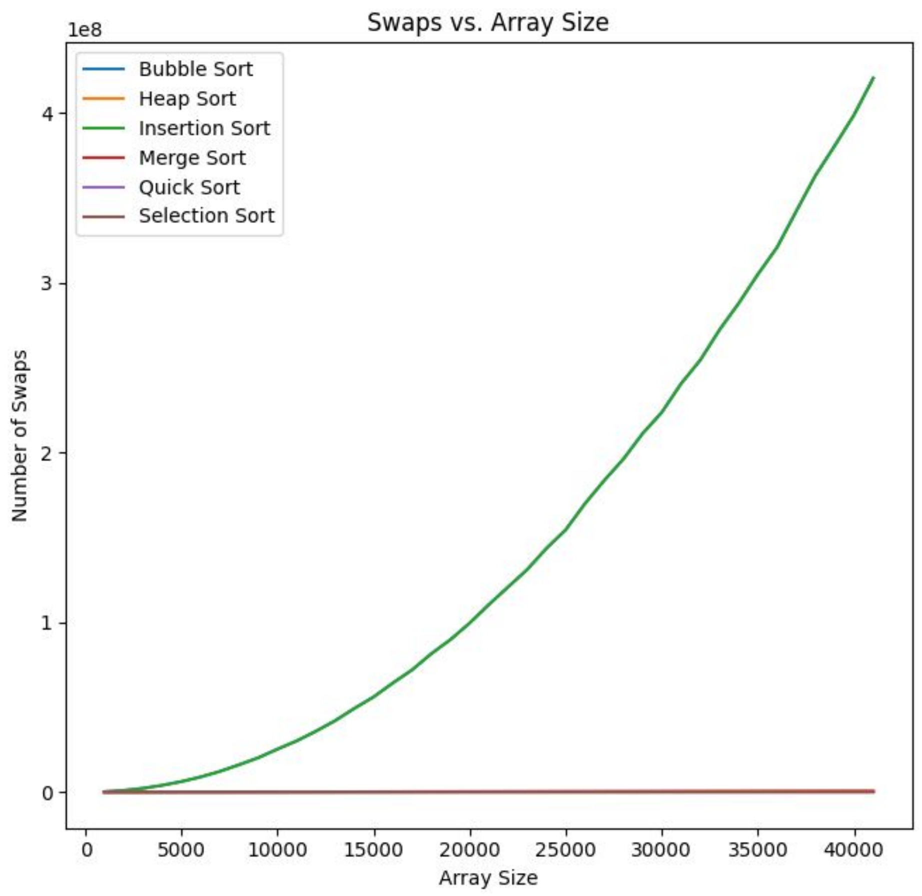


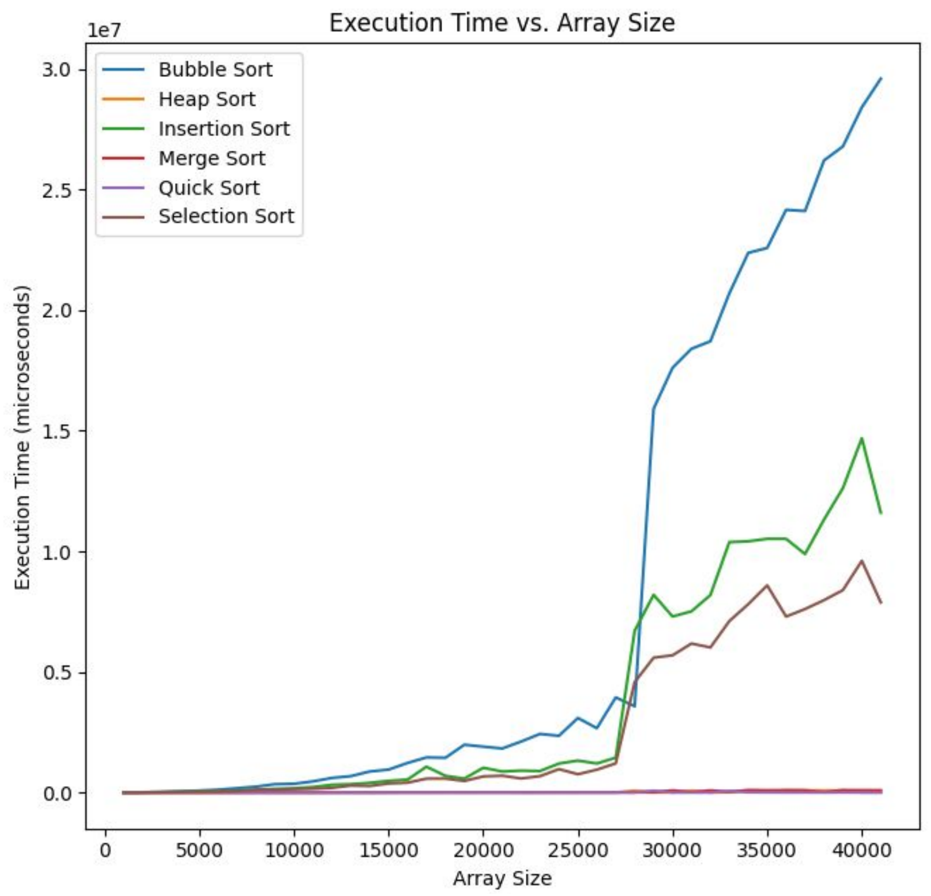
The bar chart depicts the total number of operations (comparisons plus swaps) required by each algorithm.

This allows us to directly compare the overall computational effort involved in sorting using different algorithms.

Data Visualization - Line Graphs





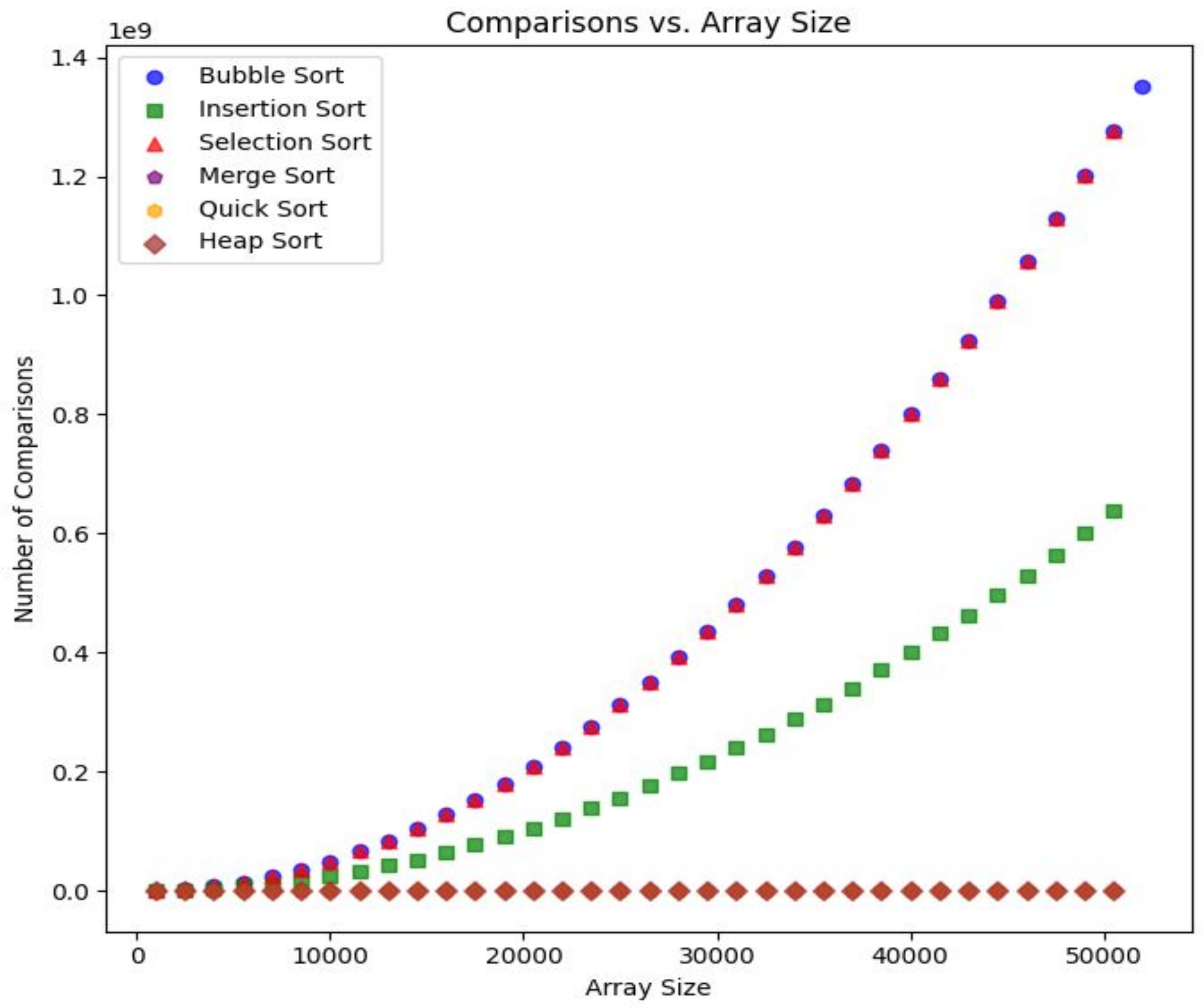


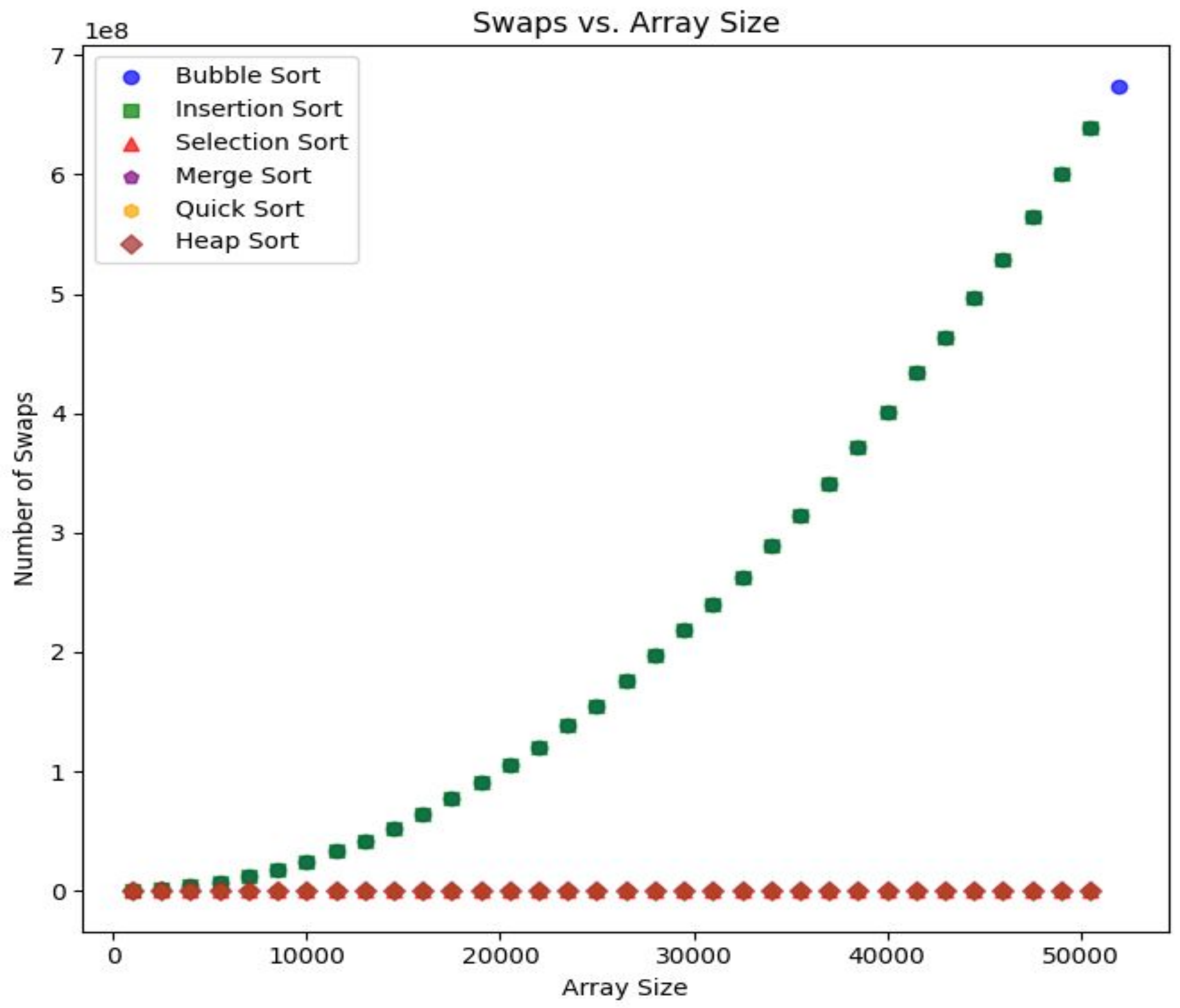
- **Bubble Sort** and **Selection Sort:** The lines would likely exhibit a steep upward slope, indicating a significant increase in comparisons as the array size grows. This is due to their inherent nature of comparing and swapping many elements.
- **Insertion Sort :** The lines might show a moderate upward slope, suggesting that the number of comparisons increases at a slower rate compared to Bubble Sort and Selection Sort.
- **Merge Sort** , **Heap Sort** and **Quick Sort:** The lines would likely have a relatively gentle slope, indicating a more controlled increase in comparisons as the array size increases.

- **Bubble Sort** and **Insertion Sort:** The lines would likely show a steep upward slope, indicating a high number of swaps as the array size increases.
- **Selection Sort** would likely have a relatively flat slope, indicating a low number of swaps even with increasing array size.
- **Heap Sort** and **Merge Sort:** The lines would likely show a moderate upward slope, indicating a controlled increase in swaps as the array size grows.
- **Quick Sort:** The line would likely show a relatively flat slope, indicating a low number of swaps across different array sizes.

- **Bubble Sort** and **Insertion Sort:** The lines would likely show a steep upward slope, indicating a significant increase in execution time with increasing array size. This is due to the high number of comparisons and swaps.
- **Selection Sort:** The line would likely show a moderate upward slope, indicating a moderate increase in execution time.
- **Heap Sort** and **Merge Sort:** The lines would likely show a gradual upward slope, indicating a controlled increase in execution time.
- **Quick Sort:** The line would likely have a relatively flat or slightly increasing slope, indicating a low and consistent execution time across different array sizes.

Data Visualization - Scatter Plots

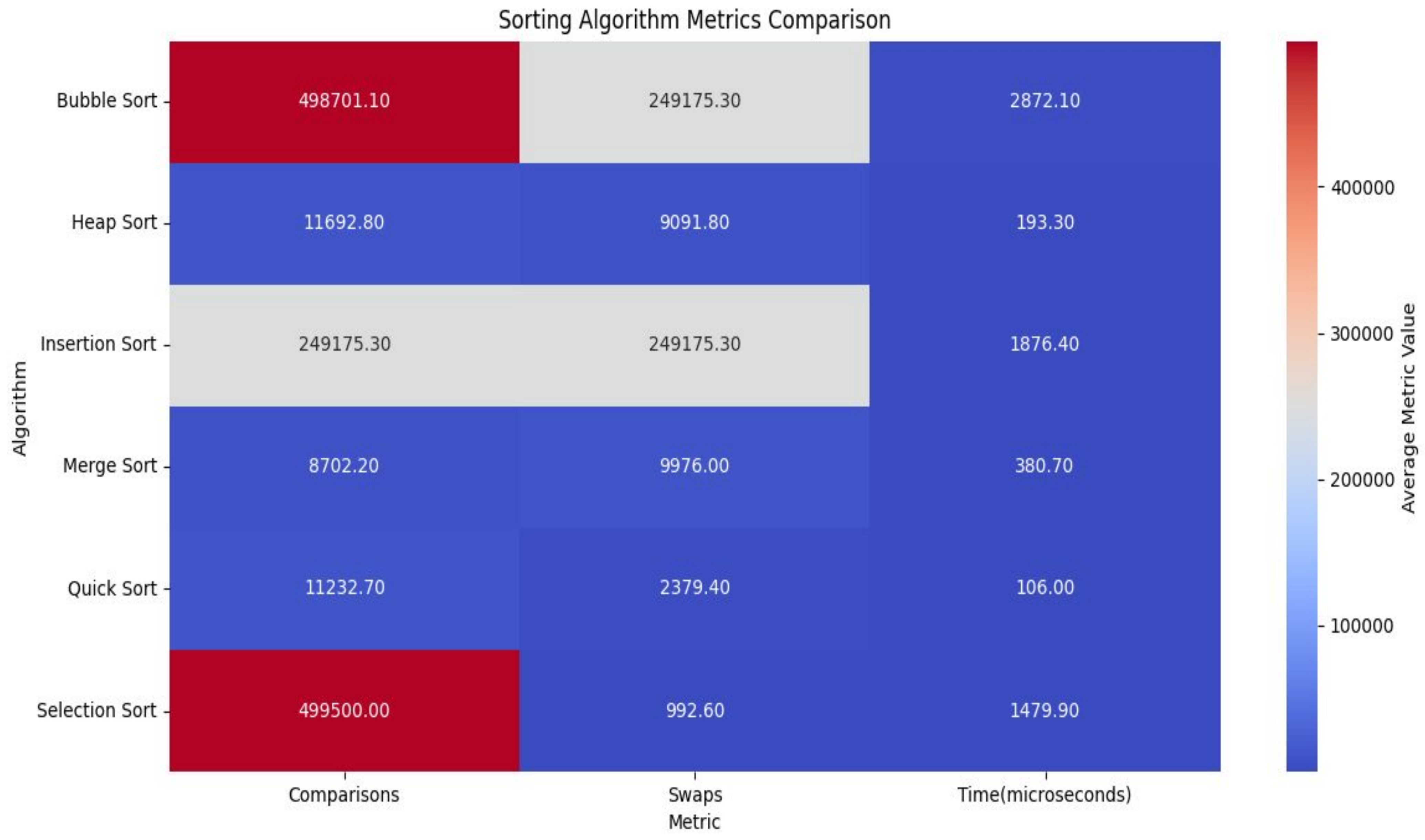




Understanding Relationships

Scatter plots are excellent for visualizing relationships between two variables. Here, we'll plot the number of comparisons or swaps against the input size for each sorting algorithm.

Data Visualization - Heatmaps



Algorithm	Comparisons	Swaps	Time(microseconds)
Bubble Sort	498701.10	249175.30	2872.10
Heap Sort	11692.80	9091.80	193.30
Insertion Sort	249175.30	249175.30	1876.40
Merge Sort	8702.20	9976.00	380.70
Quick Sort	11232.70	2379.40	106.00
Selection Sort	499500.00	992.60	1479.90

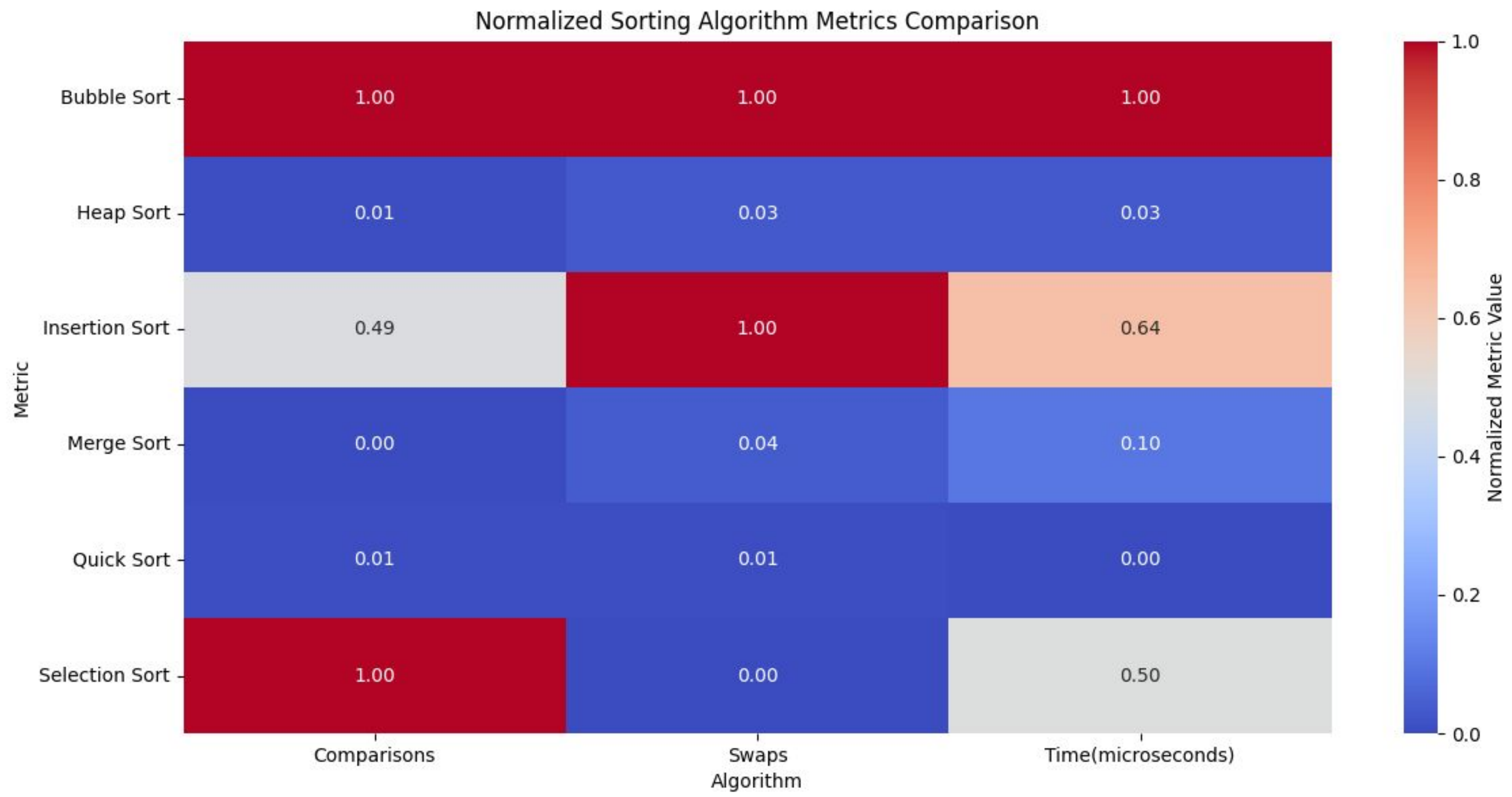
Multidimensional Analysis

Heatmaps allow for multidimensional analysis by representing data values in a grid format.

Visual Comparison

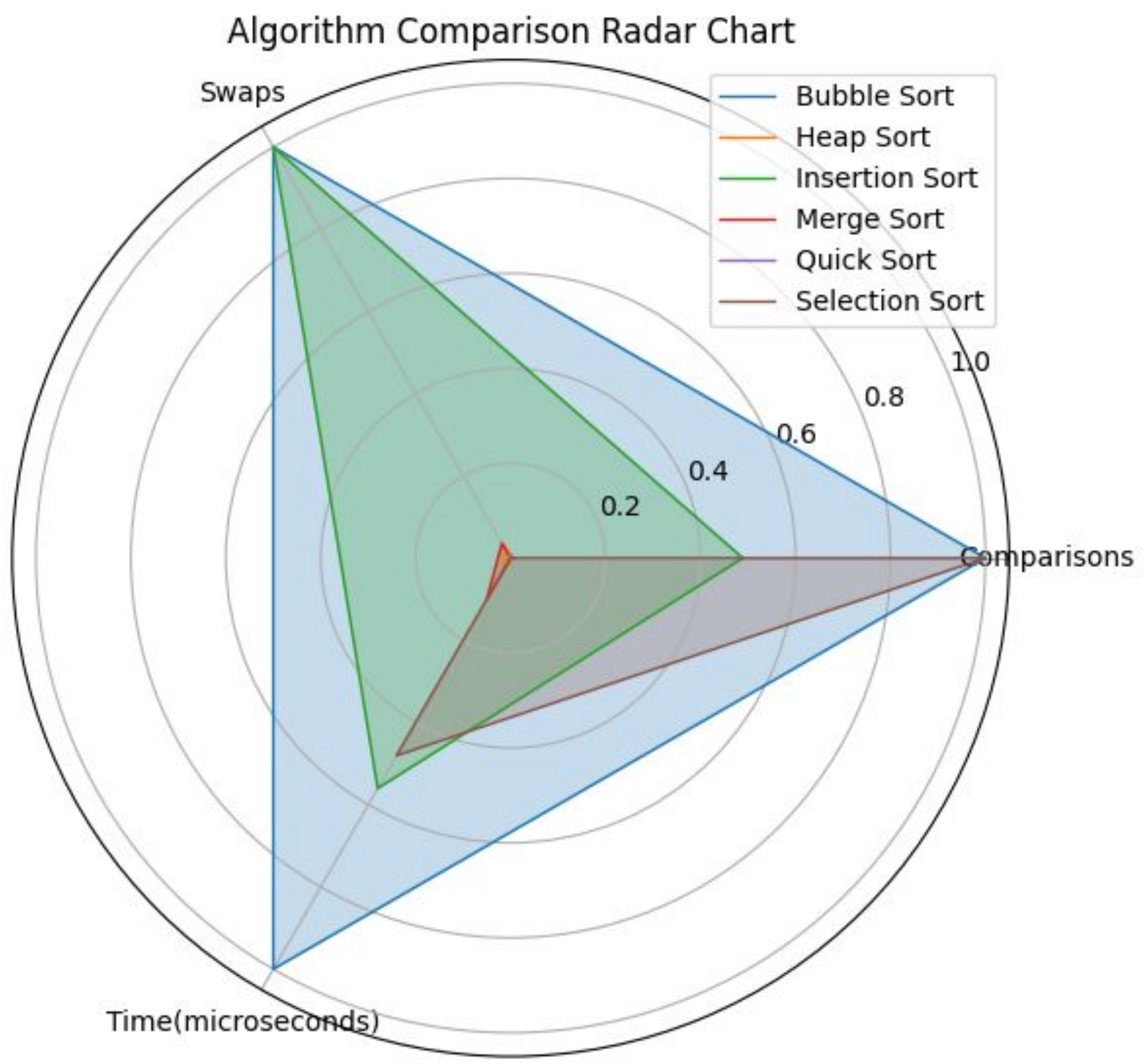
The color-coded cells highlight the relative values of each performance metric for each sorting algorithm, enabling visual comparison.

Data Visualization - Heatmaps



	Comparisons	Swaps	Time(microseconds)
Bubble Sort	1.00	1.00	1.00
Heap Sort	0.01	0.03	0.03
Insertion Sort	0.49	1.00	0.64
Merge Sort	0.00	0.04	0.10
Quick Sort	0.01	0.01	0.00
Selection Sort	1.00	0.00	0.50

Data Visualization - Radar Charts



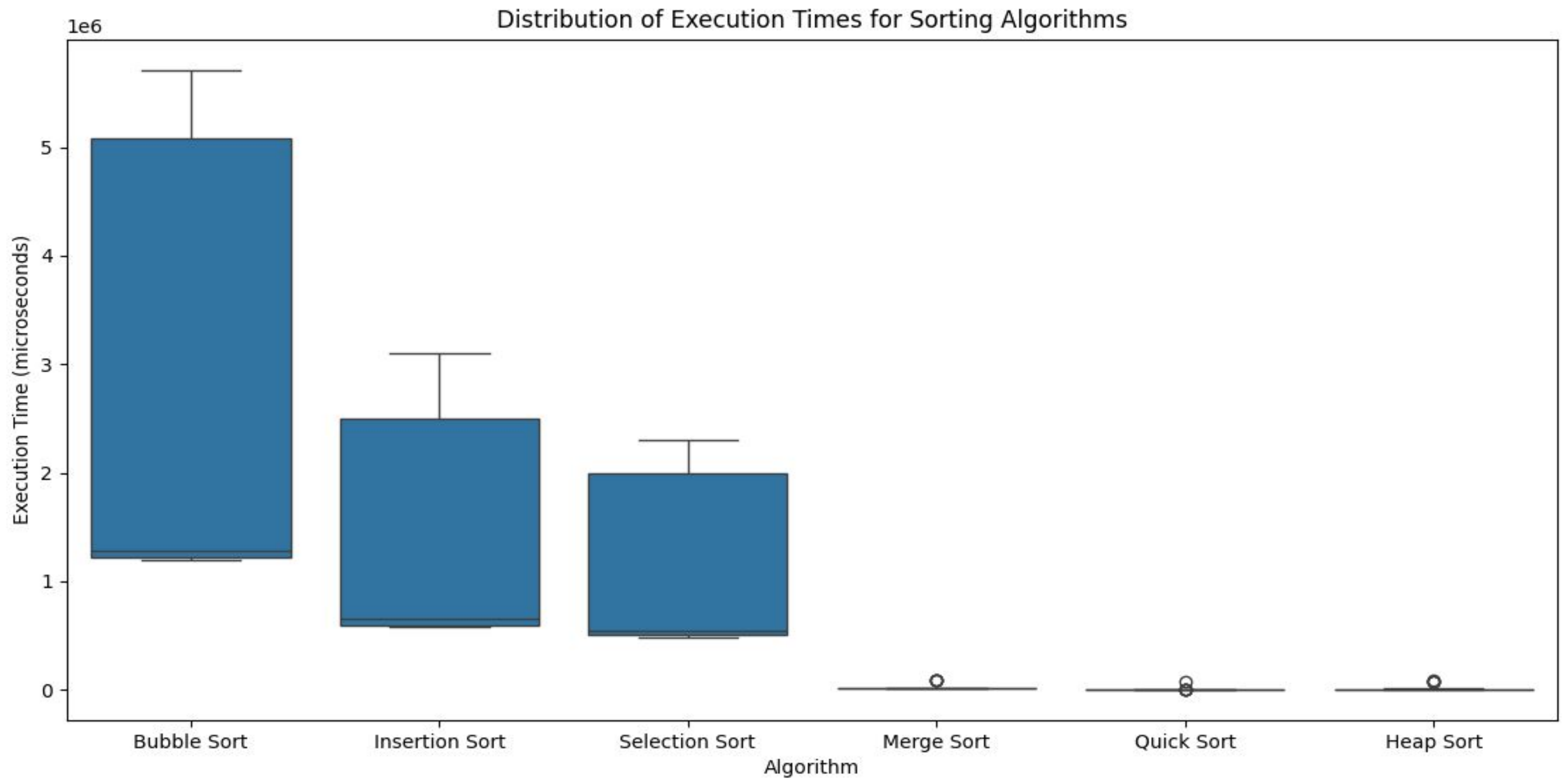
Visual Comparison

Radar charts are great for visually comparing multiple algorithms across multiple dimensions.

Performance Strengths

We can easily identify the strengths and weaknesses of each algorithm by observing the shape and size of the polygons.

Data Visualization - Box Plots



Outlier Identification

They help identify outliers, which are extreme values that may indicate anomalies or edge cases in the algorithm's performance.

Comparison of Performance

Box plots enable easy comparison of the performance of different algorithms based on the central tendency and spread of their execution times.

Distribution Insights

Box plots visualize the distribution of execution times for each algorithm, providing insights into the spread and skewness of the data.

Conclusion

Bubble Sort

High number of comparisons and swaps due to its simple, iterative nature. Inefficient for larger datasets.

Insertion Sort

Efficient for smaller datasets, but performance degrades with increasing array size. High number of swaps can impact execution time.

Selection Sort

Low number of swaps but high number of comparisons. Performance is relatively consistent across array sizes.

Merge Sort

Efficient for larger datasets due to its divide-and-conquer approach. Consistent performance with moderate comparisons and swaps.

Quick Sort

Generally the most efficient algorithm with low comparisons and swaps. High performance across various array sizes. However, performance can degrade in the worst-case scenario (e.g., already sorted data).

Heap Sort

Good balance between comparisons and swaps. Efficient for larger datasets.