Ayşenur Elif Abanoz

Experiment Setup:

I conducted an experiment to analyze and compare the performance of five sorting algorithms implemented in the "sorter" package. The algorithms tested were QuickSort, MergeSort, Insertion Sort, Bubble Sort, and Selection Sort. I used a Java program to generate arrays of different sizes (5000, 7500, and 10000 elements) with varying orderings (random, ascending, and descending).

Procedure:

Array Generation:

I generated three arrays for each size and ordering combination: random, ascending, and descending.

The arrays were generated using a Java program that utilized the Random class for random order and implemented simple logic for ascending and descending orders.

Sorting Algorithm Testing:

I tested each sorting algorithm (sort1, sort2, sort3, sort4, and sort5) on the generated arrays.

For each array size and ordering, I measured the execution time of each sorting algorithm using System.nanoTime().

The Java reflection API was employed to dynamically call the sorting methods in the "sorter" class.

Data Collection:

I collected the execution times for each sorting algorithm and stored the data in separate arrays for further analysis.

The collected data was organized into three groups based on array ordering: random, ascending, and descending.

Data Visualization:

I created line graphs using Matplotlib in Python to visualize the performance of each sorting algorithm.

The x-axis represents the array size, and the y-axis represents the execution time in nanoseconds.

Separate graphs were created for each sorting algorithm, showing the performance across different array orderings.

Experimental Results and Analysis:

Sort1 (QuickSort):

QuickSort exhibited efficient performance across all array sizes and orderings.

In the random case, the algorithm's execution time increased moderately with larger array sizes.

For ascending and descending arrays, QuickSort demonstrated consistent and fast performance.

Sort2 (MergeSort):

MergeSort showed stable performance but with higher execution times compared to QuickSort.

The algorithm's execution time increased linearly with larger array sizes in the random case.

MergeSort performed well on ascending and descending arrays but had slightly higher execution times.

Sort3 (Insertion Sort):

Insertion Sort exhibited notable differences in performance based on the array ordering.

In the random case, Insertion Sort's execution time increased significantly with larger array sizes.

For ascending arrays, the algorithm showed faster execution, while descending arrays resulted in higher execution times.

Sort4 (Selection Sort):

Selection Sort demonstrated the poorest performance among the tested algorithms.

The execution time increased significantly with larger array sizes for all orderings.

It performed especially poorly on descending arrays.

Sort5 (Bubble Sort):

Bubble Sort's performance was similar to Selection Sort, with slow execution times.

The algorithm's execution time increased substantially with larger array sizes in all orderings.

Bubble Sort showed the least efficiency among the tested algorithms.

Conclusion:

Based on the experimental results and analysis, I conclude that Sort1 (QuickSort) generally outperforms the other sorting algorithms in terms of speed and efficiency. Sort2 (MergeSort) follows QuickSort in performance, while Sort3 (Insertion Sort), Sort4 (Selection Sort), and Sort5 (Bubble Sort) exhibit slower execution times, especially with larger array sizes and specific orderings. The visualizations provide insights into the time complexity of each sorting algorithm under different scenarios, aiding in understanding their strengths and weaknesses.