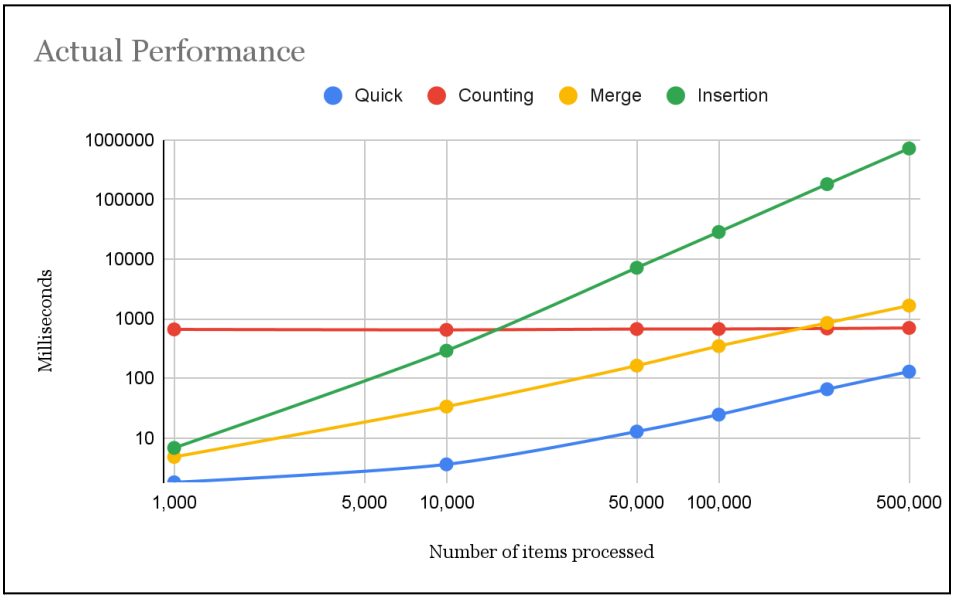
Performance Comparison of Sorting Algorithms

Steven Simmermon and Harrison Downs | May 30, 2024 | CSCI 2320 Final Project

This report compares the execution times of various sorting algorithms to understand their performance characteristics and Big O notation.

We implemented four sorting algorithms: insertion sort, merge sort, quick sort, and counting sort. Additionally, we tested these algorithms in C++ using Microsoft Visual Studio as our IDE. Sorts were organized as their own individual function. In order to keep track of execution time, each function was called in main one after the other. Breakpoints were then placed on each function. Using the Diagnostic Tools, Visual Studio kept track of the duration of execution between each breakpoint. Each algorithm was run five times, so the data shows average execution times accurate to the millisecond.

Expected Performance	
Counting Sort	O(n)
Insertion Sort	O(n^2)
Merge Sort	O($n$log(n))
Quick Sort	O($n$log(n))



Average Elapsed ms	*Items Sorted*					
Algorithm	1,000	10,000	50,000	100,000	250,000	500,000
Counting Sort	662.2	650.2	671.4	672.4	685.2	700
Insertion Sort	6.8	291	7,160.6	28,536	180,991.4	718,051
Merge Sort	4.8	33.6	163.2	347	849.4	1,661
Quick Sort	1.8	3.6	12.8	24.6	65.6	129.75

As anticipated, the insertion sort rapidly became unviable with larger datasets. Its performance was acceptable when $n <= 10{,}000$, as its execution time of 291 ms is comparable to the other algorithms. However, this algorithm took exponentially longer to execute as n increased. The insertion sort's execution times clearly reflect a square relationship between n and required resources, which aligns with its expected performance of $O(n^2)$.

Additionally, the merge and quick sorts' data is consistent with their expected performance of $O(n\log(n))$. Their data exhibits a logarithmic relationship with n, although the quick sort consistently outperformed the merge sort. These algorithms' logarithmic similarity is due to the fact that they both recursively split the unsorted elements in a way that efficiently handles large sets of data.

Contrary to its expected performance of $O(n)$, the counting sort's data reflects an actual performance of $O(1)$. From $n = 1{,}000$ to $n = 500{,}000$, this algorithm consistently required 650-700 ms to fully execute. Although its execution times generally show an increase relative to n, these increases are microscopic, making a weak case for this algorithm having an $O(n)$ performance. In future studies, the counting sort algorithm should be tested with much larger datasets to determine if this algorithm's performance is $O(n)$ or $O(1)$.

The insertion sort algorithm is the simplest to implement, and its reasonable execution times when $n <= 10{,}000$ means that it is only acceptable to use with small amounts of data. The merge and quick sort algorithms both efficiently handle large amounts of data, and they can function when the minimum or maximum data values are unknown. Lastly, the counting sort algorithm sharply outperformed the other algorithms with large data sets. Therefore, this is the best algorithm to implement when vast amounts of data need to be processed and minimum and maximum data values are known.