Liam Murray

Lijamurr

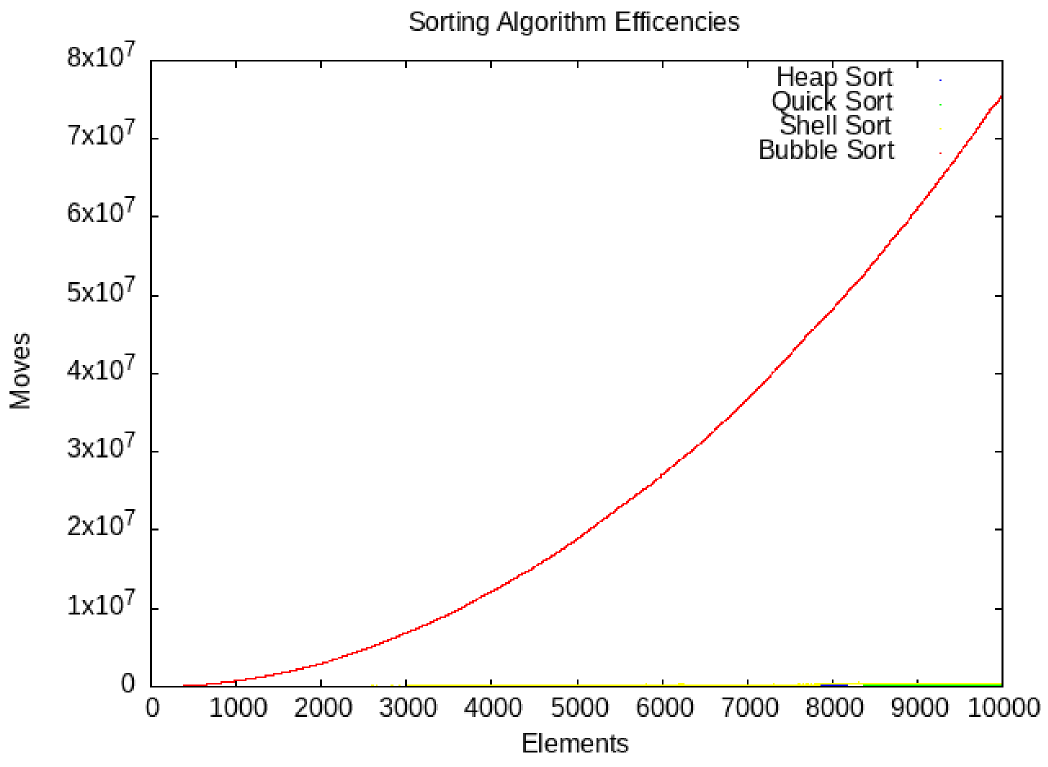
CSE13s Assignment 4

October 23, 2022

All Sorts Of C Code Write-up

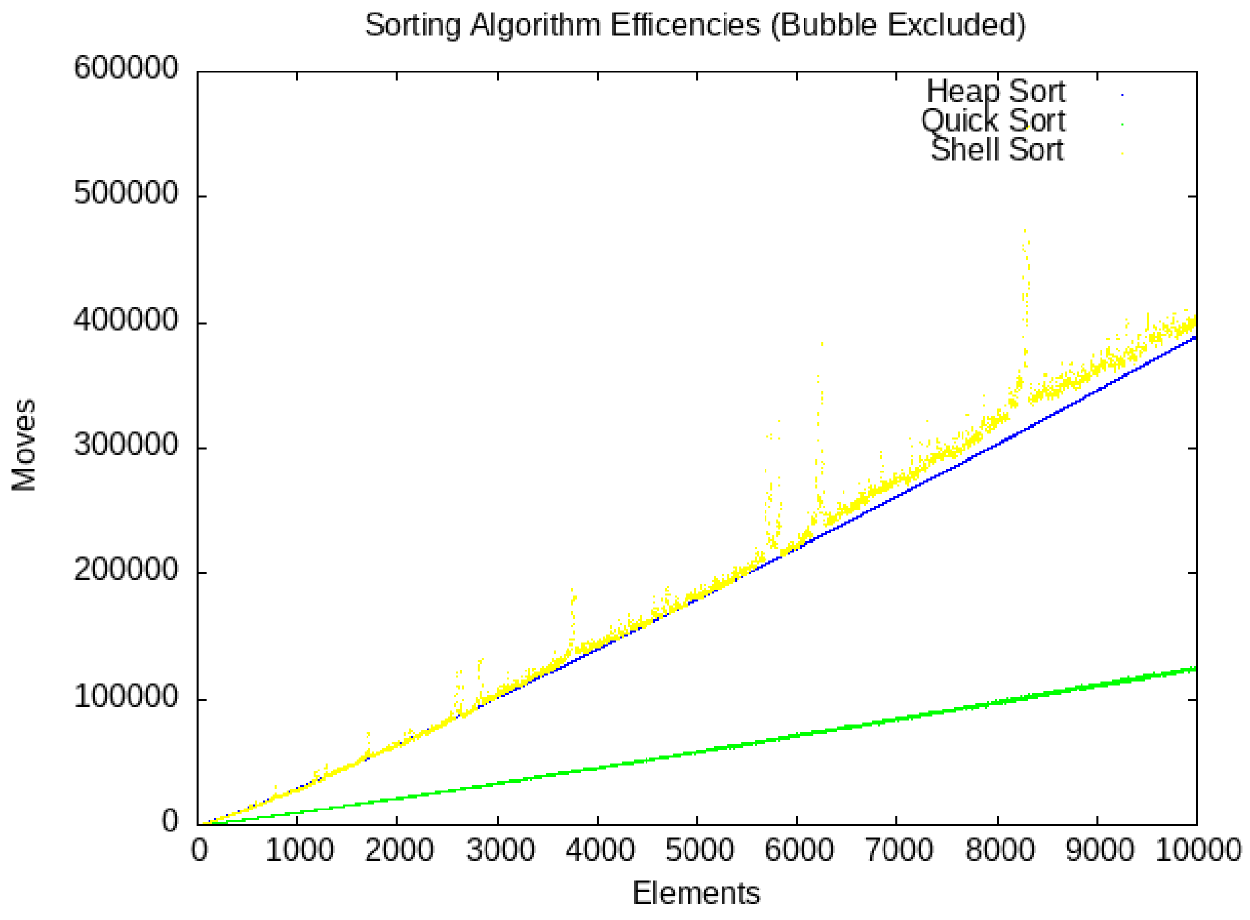
Overall, what this assignment has exemplified is that sorting algorithms are often simple solutions that vary widely in complexity and efficiency. Whereas I once thought that the issue of sorting data was a one-stop fix-all, it has become abundantly clear that certain algorithms outperform others by miles, and that not all data is equal.

For example, the simplest implementation, bubble sort, is tempting to use for all sorting possibilities. However, one soon discovers that simplicity is often repaid in decreased efficiency. Although bubble sort worked well for partially sorted algorithms, anything past a one thousand element array often took far more time. Bubble sort also performed the worst with an array in reverse order. A 50 element reversed array required 3675 moves, whereas a 50 element random array required 1884. A reversed array nearly doubles the computation complexity, rendering bubble sort almost impossible to use.



This graph shows just how drastic the difference between less efficient and more efficient arrays is. Bubble sort (red) dwarfs the other sorting algorithms so badly that they are barely visible. Bubble sorts exponential nature means that move computation gets really big, really fast. Overall, bubble sort works best with smaller, almost sorted arrays.

Bubble sort's closest relative, shell sort, is a much more optimistic story. Shell sort performs much better than bubble sort, and more importantly, does not scale exponentially. Overall, shell sort performs best with partially sorted arrays, and a reverse array is almost no trouble. When tested, shell sort took 512 moves to sort the revered array, where it took 734 to sort a random array. It performed better than when the array is random, likely due to the fact that it swaps elements from opposite sides of the array in its initial computation, resulting in partially sorted arrays later in the process.



Shell Sort (yellow), Heap Sort (blue), Quick Sort (green)

This graph shows that while shell sort is much better than bubble sort, it still loses out to Heap sort and Quick sort, and is much more susceptible to outliers. The wavy line shows how edge cases can decrease its efficiency.

The next most efficient sort is Heap sort. As shown in the graph, it is very close to shell sort, but more consistent. Its graph is linear, which means that it works as well with smaller arrays as it does with larger ones. Overall, it's a very reliable, easy to implement sort that is effective with most data types and array sizes. When tested with a reverse array, it performed 1308 moves, where with a random one it performed 795. A drastic increase, likely due to the repeated use of “up_heap”, however still much better than other counterparts.

And finally, quick sort. Quick sort was by far the most difficult algorithm to implement. Its recursive nature made it very complex, and as such much harder to make functional. However, it is worth it. As shown in the graph, it blows every other sorting algorithm out of the water. It is linear, which makes it a great algorithm for large arrays, and because it uses shell sort when the sub arrays get smaller, it maintains the benefits of being fast with fewer elements. When tested with the reverse array, it took 203 moves, where the random array took 286. Not only is this an improvement in and of itself, it is far better than the other algorithms. Overall, quick sort is the best algorithm in every way except for simplicity.