Sorting Benchmarking

To effectively benchmark each of my four sorting algorithms, heapSort, mergeSort, selectionSort, and insertionSort, I began by randomly generating 100 different sequences of a given length. I took the mean runtime of each algorithm over 20,000 runs. This resulted in one row of data with 5 elements, consisting of the provided length and the four averaged runtimes, each corresponding to a different algorithm. Lastly, I provided thirteen different array lengths, from 2 to 128 integers in length, and plotted the runtimes below.

This means that for every runtime datapoint on the graph, the corresponding algorithm ran 2 million sequences of the associated length.

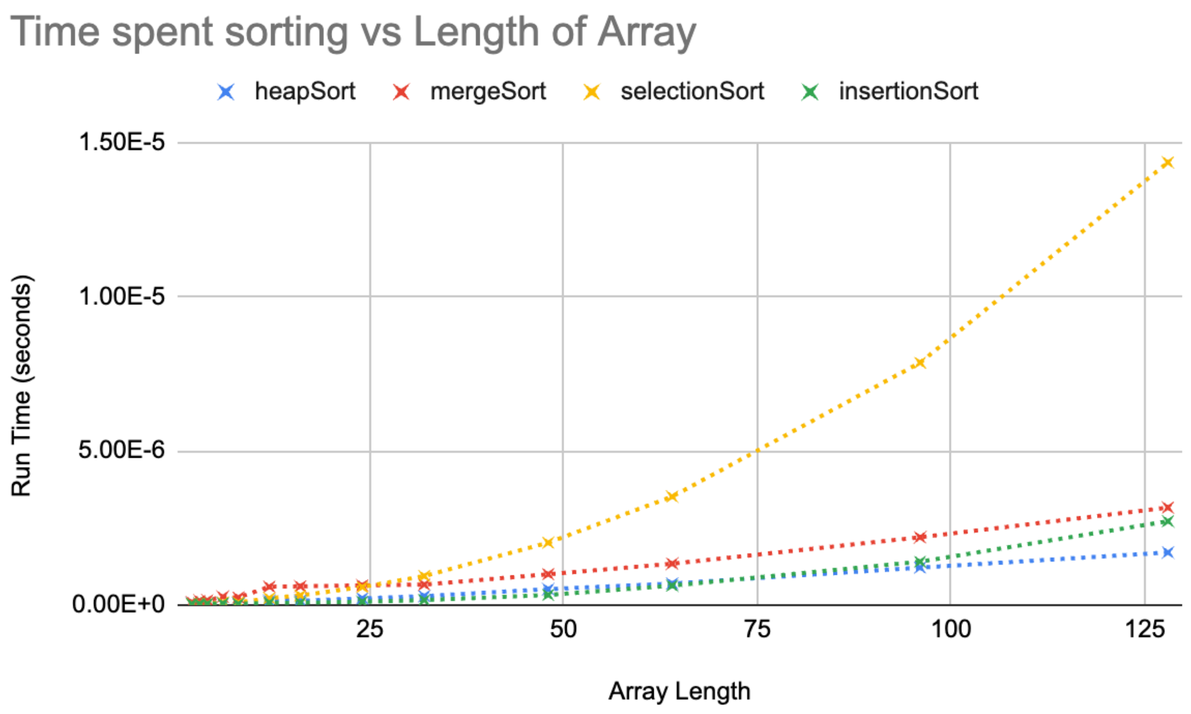

Figure 1: Run Time vs. Array Length for four different sorting algorithms

Selection sort and insertion sort are both O(n^2) algorithms, while merge sort and heap sort are O(nlog(n)) algorithms.

From the graph, it is immediately clear that selection sort is the slowest of the four algorithms once the array length surpasses 30. This makes sense with its time complexity, as we can see that an increase in array length causes a parabolic increase in runtime. Though insertion sort seems to be doing significantly better than selection sort, to the point of being on par with the O(nlog(n)) algorithms, you can also see a parabolic shape emerging from its runtime data points. The significance of this is that although merge sort and heap sort have theoretically more efficient runtimes, their runtime improvement from an $O(n^2)$ algorithm on a small scale sorting task like the ones I tested is not only negligible but possibly negative.

It was interesting to me that merge sort consistently performed worse than insertion sort despite its time complexity. However, since I handle the splitting process within mergeSort by creating new arrays with .sliceArray() rather than through indexing, it makes sense that it is less efficient in its current implementation.

The $O(n^2)$ algorithms will be worse for longer lists than the $O(n\log(n))$ algorithms, but are good for maintaining simple implementations. With an infinitely long list, my best algorithm in the way they are currently implemented is heapSort. With short lists and a desire for simplicity, insertionSort wins out. For reference, insertionSort was 18 lines of code while heapSort was 50 lines of code, but insertionSort is significantly easier conceptually and more readable.

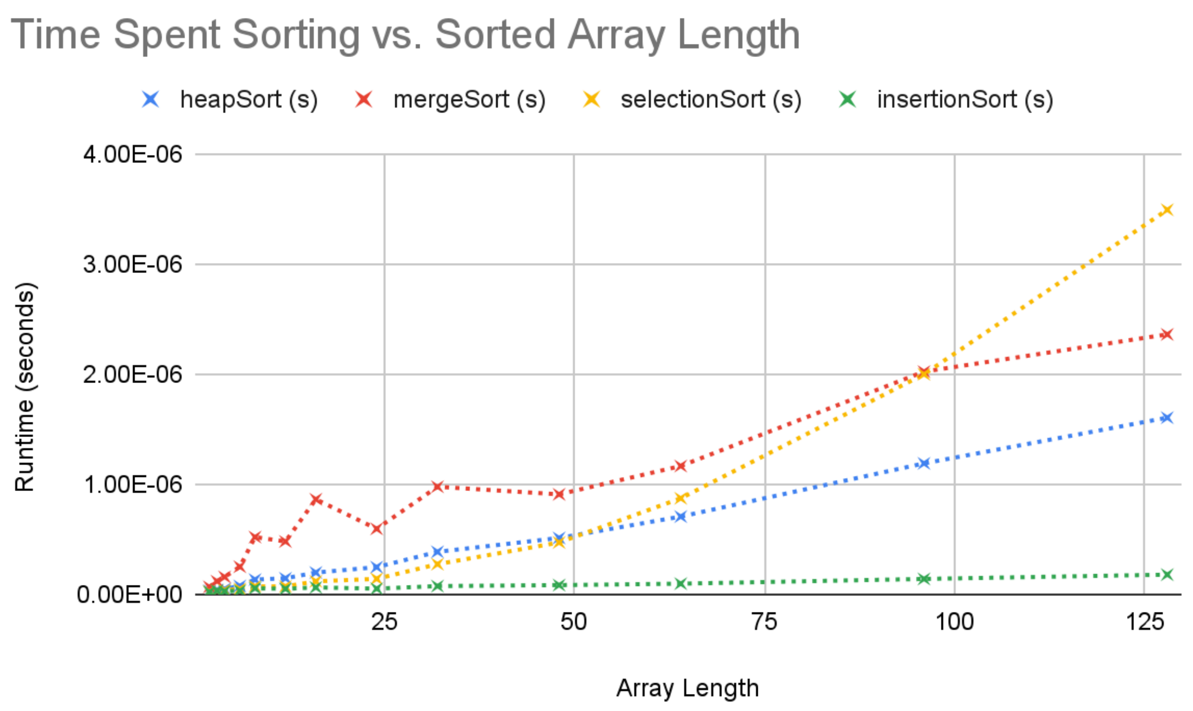

Figure 2: Sorted array benchmarking for four different algorithms

It's important to consider that the runtime efficiency of each algorithm can also be dependent on how many times you run it because of internal computer optimization software, which might explain the strange behavior of the mergeSort function However, from the graph, it is clear that insertion sort consistently performs best when the sequence is already sorted, which makes sense since it should only run about 4 lines of code for each passthrough if the sequence does not require any reassignment of position. It behaves as a linear function with a very small slope. The data in Figure 2 was found using the mean runtime over 1,000,000 runs for each size.

New Frontiers in Sorting

One new problem that the paper "Faster sorting algorithms discovered using deep reinforcement learning" discusses is the application of deep reinforcement learning to the creation of newer, more efficient sorting algorithms. Researchers at Google devised a single-player game dubbed *AssemblyGame* which rewards a player based on correctness and the amount of time it takes to sort a sequence. Their machine player, named *AlphaDev*, receives and optimizes the sorting state through repeatedly testing and generating sorting algorithms. The sorting state is represented by a vector $\mathbf{S_t} = \mathbf{<P_t, Z_t>}$, where $\mathbf{P_t}$ is a representation of the algorithm in assembly code and $\mathbf{Z_t}$ is the state of the memory and registers after executing that algorithm. Their reinforcement learning model consistently created efficient algorithms and introduced a new way of thinking for sorting shorter algorithms. It cut out one line of instruction in both the swap and copy moves, and optimized the sorting pathway for lists of length 4.

AlphaDev was able to surpass human latency benchmarks by 70% for length 5 sequences and ~1.7% for sequences larger than 250,000 elements, which is extremely significant since these benchmarks are considered state-of-the-art for human programmers. Since sorting algorithms are at the base of many computer programs today, these percentage increases could mean a very significant decrease in latency, as well as a more efficient usage of existing hardware architectures through parallelism. On a day-to-day scale, this would lead to more responsive applications, an increase in the amount of big data that an entity could handle, and development in data analysis tools.

Works Cited

Mankowitz, Daniel J., et al. "Faster Sorting Algorithms Discovered Using Deep Reinforcement Learning." *Nature*, vol. 618, no. 7964, 7 June 2023, pp. 257–263, doi:10.1038/s41586-023-06004-9.