Sorting Report.

Adrian Jonsson Sjödin

Fall 2022

Introduction

The task in this assignment was to explore some different sorting algorithms to gain a better understanding how different implementations can affect the time it takes to sort a large array.

Task

Implement the three following sorting algorithms: Selection sort, Insert sort and Merge sort. For each of the algorithms explain the run time as a function of the size of the array and state their time complexity using Big O notation. Lastly do some benchmark and compare how the different algorithms compares to each other.

Method & Theory

The Selection Sort algorithm was the first to be implemented and is quite straight forward. It takes the first element and then looks through the array to see if there's an element smaller than it. When it has looked through the whole array it swaps the first element with the found minimum element and then goes on to the next element and repeat the process until the end of the array has been reached. This means that it will have a time complexity of $\mathcal{O}(n^2)$ since it will have to take one element at a time (out of n elements) and look through an array of size n to see if there's a smaller element $n-1$ times.

The second algorithm to be implemented was the Insertion Sort algorithm which is slightly more complicated to implement. Instead of looking through the array from start to end after the smallest element, this algorithm compare the element it is looking at with the previous element and its previous element until it finds the spot where it fits in the array. In best case if the array is already sorted so should this algorithm give us a time complexity of $\mathcal{O}(n-1) = \mathcal{O}(n)$, since it then only need to go through the array once doing $n-1$ comparisons checking that every value is larger

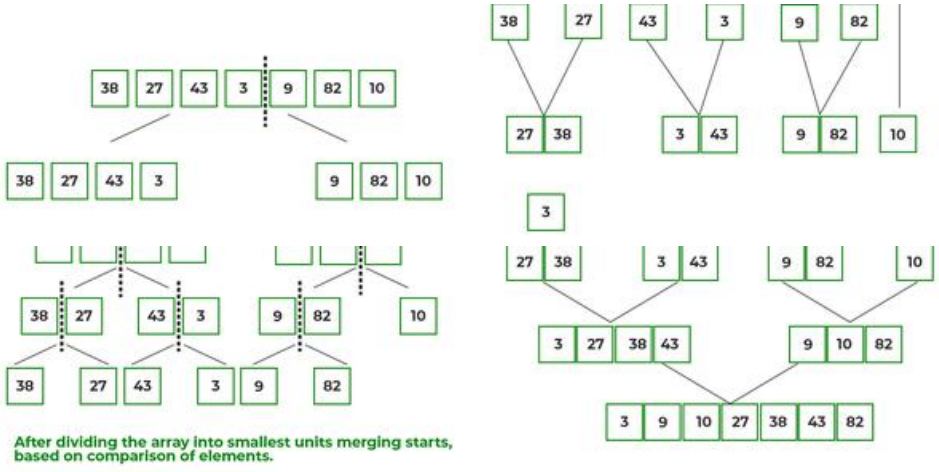

Figure 1: Merge Sort. Read top down, left to right. Picture source

than its predecessor. On average though it should have a time complexity of $\mathcal{O}(n^2)$ since it will have to go through an array of size n and for each element $i \in n$ do up to $i - 1$ comparisons for $i \to n$.

Lastly we implement the Merge Sort algorithm which was by far the hardest to implement because of its recursive nature. This algorithm works by first splitting an array in halves recursively until further division becomes impossible. After this step we start merging the elements back together again based on comparison of size of the elements. We compare the elements for each sublist and then combine them together into another list in a sorted manner. Figure 1 visualizes this in a good way. I haven't included the code fort the recursive call here in the report since the teacher already went through that part during his lecture, and thus would be redundant to show again, but for those interested all the code can be found here: GitHub. The time complexity for the merge sort algorithm should be $\mathcal{O}(n \cdot log(n))$.

Result

Bellow in table 1 and figure 2-3 you can see the data from the benchmarked run for the different algorithms as well as plots over how the time execution increase with the array size. The benchmark is the minimum average time it took to sort an array of size n over 1000 loops.

Array size	Select	Insert	$\frac{select}{insert}$	Merge	$\frac{insert}{merge}$
100	3.80	1.34	2.84	2.76	0.48
200	11.0	4.11	2.68	5.21	0.79
400	43.3	14.3	3.03	9.61	1.49
800	165.1	54.8	3.01	31.7	1.73
1,600	603.3	214.9	2.81	90.6	2.37
3,200	2,439.2	855.9	2.85	243.2	3.52
6,400	9,642.7	3,375.8	2.86	513.3	6.58
12,800	45,481.7	16,778.1	2.71	1247.0	13.45

Table 1: Benchmark for the different sorting algorithm. Time in μs

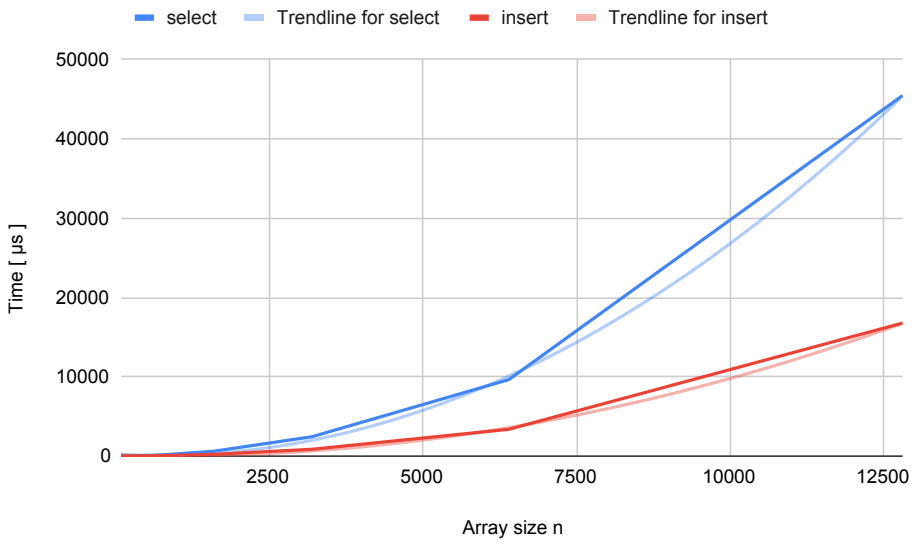

Figure 2: Select vs. Insert Sort

Discussion

As can be seen from the benchmark in table 1 the Selection Sort is the slowest algorithm followed by the Insertion Sort, and then Merge Sort as the fastest out of the three. The Insertion sort seems to be on average 185% faster than Selection Sort, and Merge sort even faster with the percentage by how much raising for larger array sizes. From figure 2 we can also see that the time execution for Selection and Insertion sort takes on the form of a second degree polynomial, and thus confirm our assumption of them being of time complexity $\mathcal{O}(n^2)$. If we had run the benchmark with more

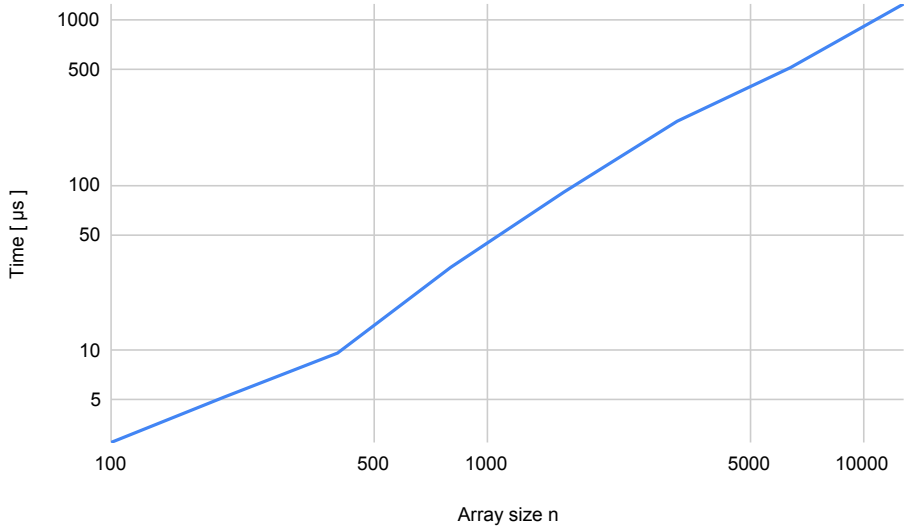

Figure 3: Merge Sort

data points and not just a doubling of n for each measurement the plot would likely have been an even closer match to the trend line seen in figure 2. Either way it is clear that Insertion sort is vastly faster than Selection sort and should be the sorting algorithm to choose between the two of them.

The Merge sort algorithm is the fastest out of the three with it being $3,550\%$ faster than selection sort for the largest array size of $n = 12,800$. It is thus evident that Merge sort is the most effective algorithm. Figure 3 shows us that the graph is nearly linear in a log-log scale which means that the time complexity is the expected $\mathcal{O}(n \cdot log(n))$. This also explains why it is so much faster than the other two algorithms.

Another advantages of merge sort is that it splits the list into smaller and smaller chunks without changing the order of the element first. This means that the general order of the elements is preserved throughout the array, i.e. two identical elements in an unsorted array will have the same relative position to each other after the sorting is done.

Lastly it is possible to improve the Merge sort algorithm by switching to another sorting algorithm for the small subarrays in the merge. Switching to insertion sort for example will improve the running time by up to $10-15\%$[1].

Once again all code for this assignment can be found here: GitHub.

[1] https://algs4.cs.princeton.edu/22mergesort/