CS2001: Foundations of Computing

Practical 4: Sort Complexity

Matriculation Number: 220029376

Tutor: Steven Linton

School of Computer Science

University of St Andrews

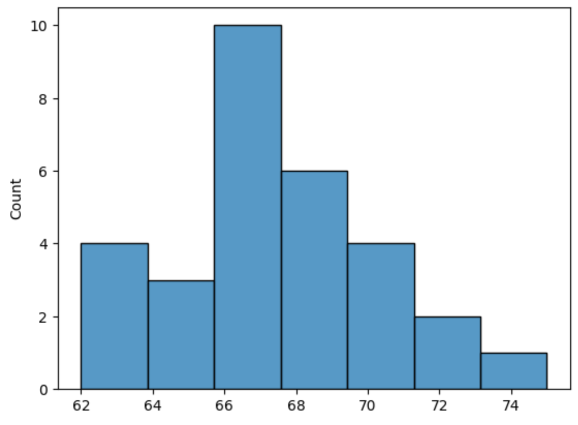
Analysis

Collecting Data

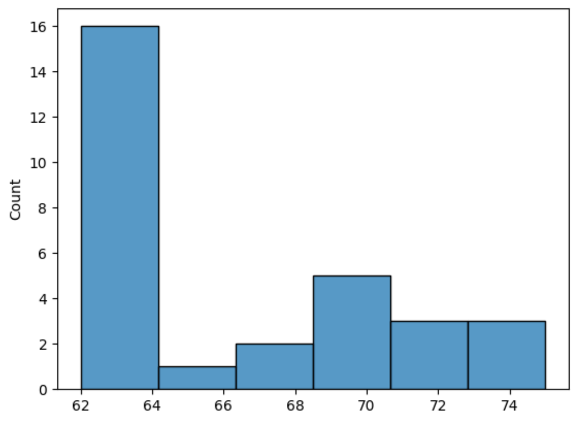
To analyse the speeds of the two algorithms I decided to use Python: I used Pandas to analyse the results of timing the algorithms and I found a library that allows Python to run a JVM and call Java methods using this JVM. I used this to call methods which I wrote in Java that return the time each sort method took for one run.

My initial approach to collecting data was to record the elapsed times of each algorithm at each array size every interval exponentially, running each size n number of times (I settled on 10,000 being a suitable n value) and take the min, max, mean and median for each size. For example, I would test array length 1, 2, 4, 8, and so on up to 1024. Then I would "zoom" into the interval where the cross-over occurred, for instance between 32 and 64, and repeat the process, recording times for 33, 34, 36, 40 and so on until 64. I would keep doing this until I found the exact cross-over point. I realised this was a bad approach because often times, when zooming into an interval, there would no longer be a cross-over point, since it had moved to a different interval upon re-running the algorithms. Instead, I decided to simply test every size from 0 to 200, and find the mean cross-over points from this data set.

Although slower, this approach was better because I got more consistent results for cross-over points. I looked at the cross-over points using both mean and median times for each size. The median and mean times were not so different, indicating little skewness and outliers, and while the median times were smoother, their sample of cross-over points was quite skewed, different from the mean times, whose sample of cross-over points looked much more normally distributed (see below).



Distribution of cross-over points using mean times.



Distribution of cross-over points using median times.

Fig. 1: Cross-over point distributions (sample size: 30).

Results

Given the distribution of the cross-over points using mean, I decided to build a confidence interval using the cross-over points collected with mean times:

$n = 30$

$\alpha = 0.01$ (99% confidence level)

$\bar{x} = 67.43$

$E = 1.40$

Interval: $(66.04, 68.83)$

Conclusions

Comparing Implementations

A classmate and I compared algorithms by running my analysis on both of our implementations. The results can be found below.

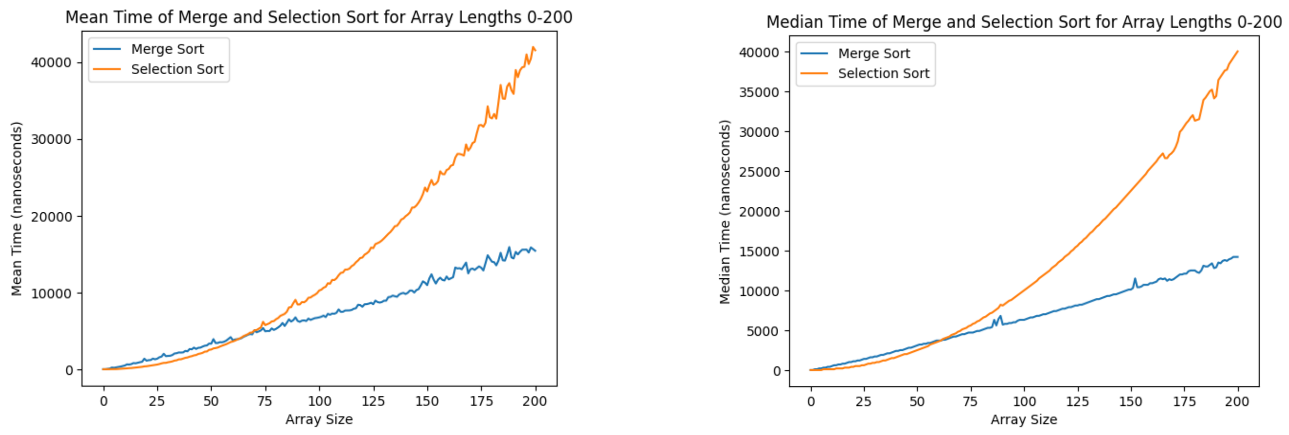


Fig. 2: My implementation

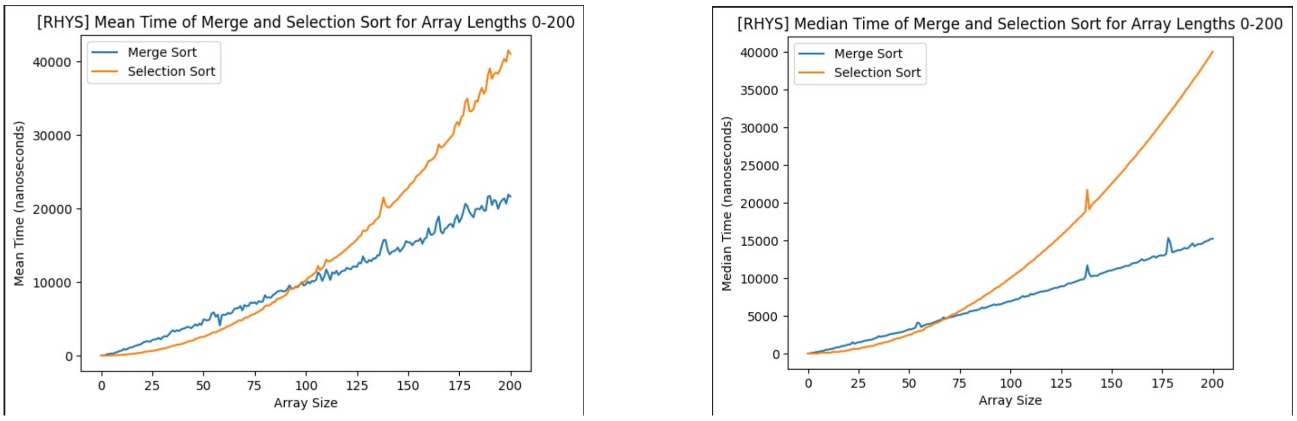


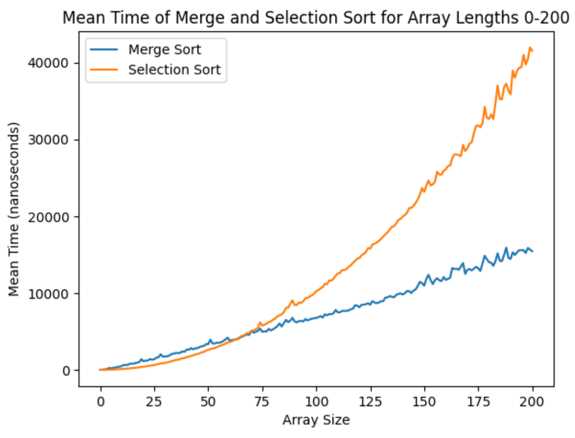
Fig. 3: Classmate's implementation

As evidenced by the figures, my classmate's implementation had a higher cross-over point when using means, but a similar cross-over point when using medians. His graph using means was also more noisy than his median graph; these two observations point to the fact that his algorithm had many outliers, which in turn indicates that his implementation is less consistent than mine.

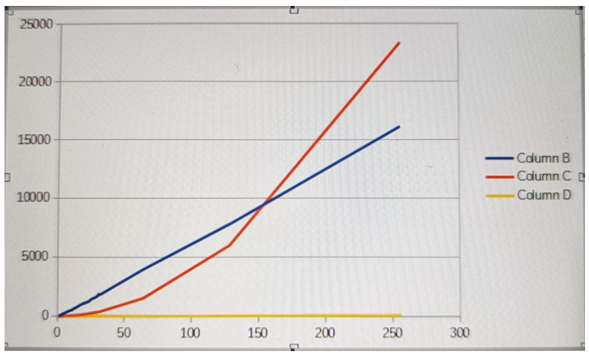
Comparing Analysis Methods

My classmate also ran his analysis method on my algorithms. His analysis method, however, was drastically different from mine: it was more pointed towards a lower cross-over point since it used the exponential method (as described before), it used a "warm-up" method to get the JVM used to the code it was testing, and it ran the algorithm far less times at each size.

The results were also quite different (see figure below). When comparing the mean graph from my analysis and that of my classmate's analysis, it looks as if the latter is scaled with a constant. This is likely where the difference lies, since the values we look at are quite small, we don't quite see the asymptotic behaviour of each algorithm's complexity so because of this the difference in analysis results can be affected by constants. Since the same algorithm was run on two different machines there is probably a difference overall in computation time due to a number of factors: JVM warm-up, system performance, background tasks, cooling of the system etc.



My analysis method



Classmate's analysis method: Blue: merge sort; Orange: selection sort; Yellow: Java's built-in sort (control).

Fig. 4

Review

Since I was more focused on the analysis, I didn't spend too much time on my implementation so I believe that some improvements could've been made. First, my algorithm can only deal with arrays of integers which makes it very limited. Second, I could improve efficiency by sorting the array in-place instead of creating and returning a new array in the merge sort implementation. Finally, I could've implemented merge sort to run iteratively rather than recursively as this would make it slightly more efficient as well.

After reviewing my analysis method I realised that there were also a few improvements that could be made. Instead of testing random arrays, I could test the sorting methods against levels of sorted-ness; from completely sorted to inversely sorted. In theory, when testing using just randomly-generated arrays, all types of arrays will be covered (according to law of large numbers) but the amount of times each size is ran would have to be very big. Instead, I could specifically target different levels of sorted-ness within the tests of each size.