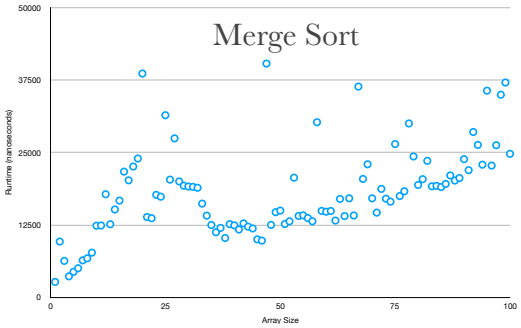
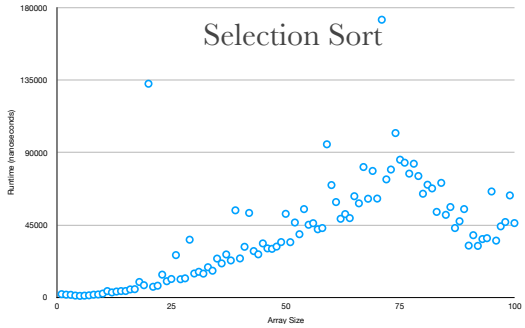
Jordan Rowley
Computer Science
University of St. Andrews

W10 Project Report

Complexity Investigation of Selection Sort and Merge Sort

My investigation into the complexities of selection sort and merge sort exposed the nature of said sorting algorithms when challenged with ordering arrays of increasing size. I collected runtimes of each algorithm by taking a snapshot of the system time in nanoseconds before and after running the program and subtracting the two. I collected data for arrays of size 1 through 100 in two different ways. Initially, I had each array filled with random integers which I knew would present some anomalous results but since I took the average of 100 tests. Here were my findings:



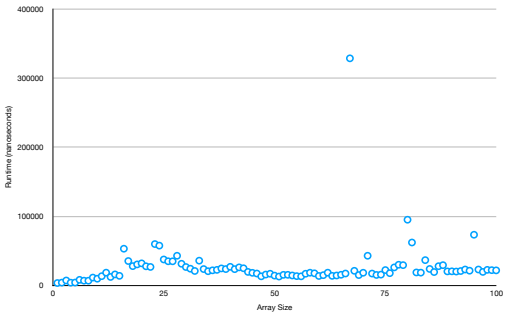


Merge sort seemed to have a weak positive correlation between array size and runtime with a surprising spike at a size of approximately 25. These were curious results since the graph doesn't follow the O(nlogn) time complexity that I expected. On the other side, the selection sort data was more promising. Until a size of approximately 75, the trend is in alignment with the $O(n^2)$. However, runtime dips after this point and I'm not quite sure why.

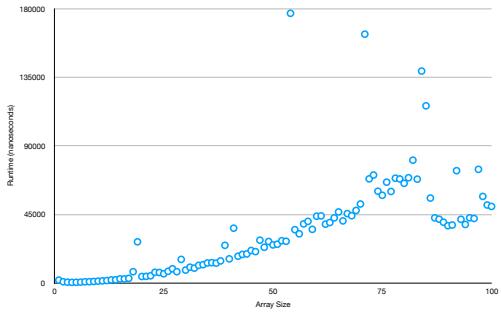
I was unhappy with these results so I devised a new plan to find more reliable experimental results (plotting worst-case complexity this time); I changed the arrays being tested so that instead of being filled with random integers, they would contain an ordered list of descending numbers. This would force the sorting algorithms into a ‘worst case’ scenario

every time and so my experiment had one less random variable. Once again I collected 100 runtimes for each size array from 1 to 100 for each algorithm. My new data showed:

Merge Sort



Selection Sort



Now, my graph for merge sort runtimes was much more linear and had a very strong correlation. There was one anomaly where size was about 70 which I can only explain as inherent randomness caused by background tasks on my computer sharing computational resources. Generally, the runtime increased as the array size increased in a linear fashion which is what is to be expected. In my updated selection sort graph: up until about a size of 80, the curve matches the n^2 complexity almost perfectly with a couple of anomalous points which can be ignored. Unfortunately, once more the runtime values dropped by a notable measure towards the top end of array sizes. Again, I don't understand this behaviour and couldn't find a reasonable excuse for it.

To collect my data I wrote a for loop which would create a temporary array (to be sorted) of increasing size and then run the respective sorting programs on the array while measuring the time between starting the algorithm and completion. At the end of the for loop, the system would print the runtime to the console. I ran this loop 100 times and in total I collected 40,000 data points across the two algorithms and two methods of filling arrays. I built a spreadsheet to house these values and plot them accordingly.

Since merge sort has two parts: divide and conquer, I wrote two methods named similarly. My divide() method would take in an array as a parameter and split it into two halves: sub-arrays called 'left' and 'right'. I called this method recursively so that each half would then be split into two halves and so on. Once each daughter array had size 1, the method would call the conquer() method. This took two arrays as parameters and would merge the two arrays passed in comparing the elements of each so that the unified array would be ordered. In the special cases in which the original set to be ordered was of size 1 or 0, the divide() method would catch this and return the set as is without even calling the conquer() method.

My implementation of selection sort used two methods when one would've been sufficient. I decided to write a separate method called swap() which would swap two elements with given indices in an array. The body of the algorithm is in the selectionSort() method. The list is separated into two partitions: ordered and unordered. It iterates through each element in the unordered partition and compares it to the minimum element found so far in that iteration. If an element is found to be lower than the current minimum, the minimum is updated. At the end of each iteration, the minimum value is swapped with the first element in the non-sorted partition of the array and added to the sorted partition. Once every element belongs to the sorted partition, the algorithm is done.

I wrote 4 JUnit tests for each sorting algorithm which would check if they worked for 4 arrays of varying nature. One used an array of size 6 with predefined elements, one used an array of size 100 filled with random integers and asserted that the algorithm produced the same ordered array as the Arrays.sort() method. The last two tests checked the behaviour of the algorithms in the special cases mentioned earlier; for arrays of size one and zero. All tests passed.

To conclude, selection sort was found to be the faster sorting algorithm until the array size was about 45, after which, merge sort proved the be the quicker sorting algorithm. This cross-over point happens at a runtime of roughly 20,000 nanoseconds. My experimental results contain anomalies which could've been ignored or overwritten by a repeat test. However, I decided to leave them since they are still true results from the test and interference with the values cannot be traced. Since I plotted the average runtime of one hundred runs of each array size my data should be reliable which is backed up by the graphs displaying expected shapes. Additionally, testing the worst-case runtimes further improved the accuracy of my findings since there would then be less randomness in the test and, therefore, less error.