CS2001 W09 Practical

140012021

13/11/15

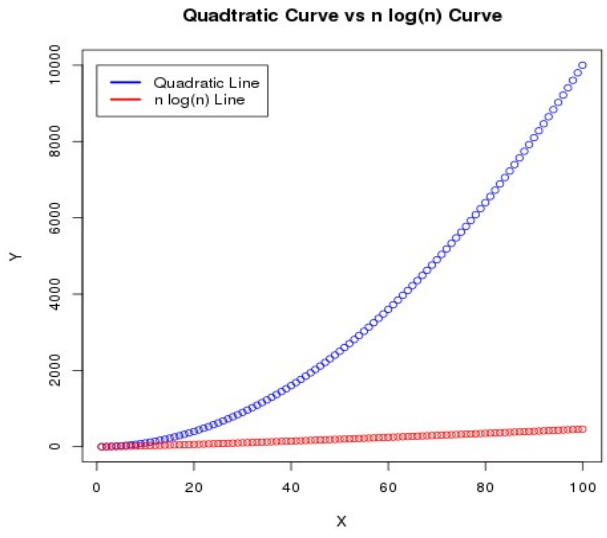
Overview

The overview of the practical was to explore how bubble sort and merge sort's speeds change as the size of the problem changes. This would be done by analysing the two sorting algorithms with different sizes of arrays and data points, producing graphs which would show the differences in the sorts. Additionally for an extension I performed some analysis on the insertion sort comparing it to the bubble sort, because they are both of complexity O(n^2) to show that just because two algorithms have the same complexity they don't necessarily perform the same.

Code Design

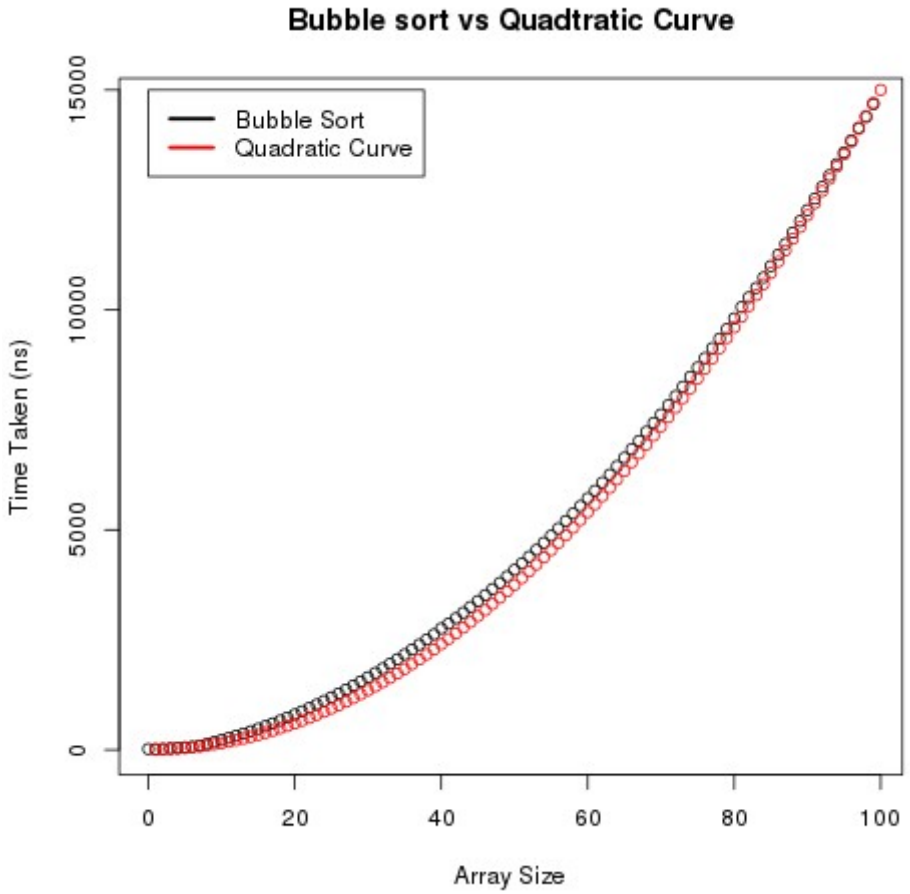
To start this practical it was important for the sorts to be correctly written or the data and results would be incorrect. I started by writing my merge sort. Initially I was going to re use the code from the previous practical, but as it was for a linked list I decided to write a new one, which is more robust. It wasn't too challenging to write as I knew the logic behind how it worked. Next I wrote the bubble sort, which again didn't involve much research as I knew how a bubble sort worked. I added flags to the bubble sort which would allow me to test both a normal bubble sort and a optimised bubble sort.

Next I implemented the timing system for the sorts. I used System.nanoTime() instead of System.currentTimeMillis() as it would yield more accurate results for arrays of smaller sizes. There were two loops over the main part of the code, one which would be the size of the array and the other one would be the number times each size of the array would be run. The total time run was done by simply by doing the System.nanoTime() before and after the method calling the sort and getting the difference. The data was then written to a .csv fie which would be analysed and graphed using R.



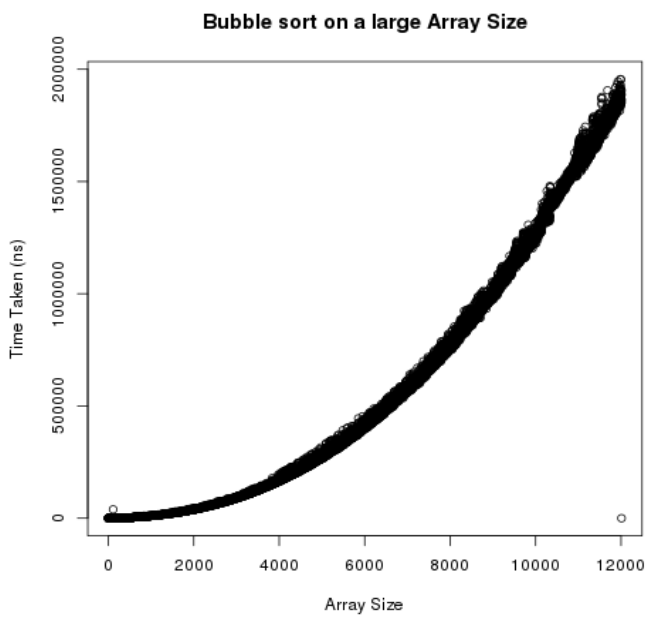
Graph 0: Demonstrating what the complexities look like.

Bubble Sort

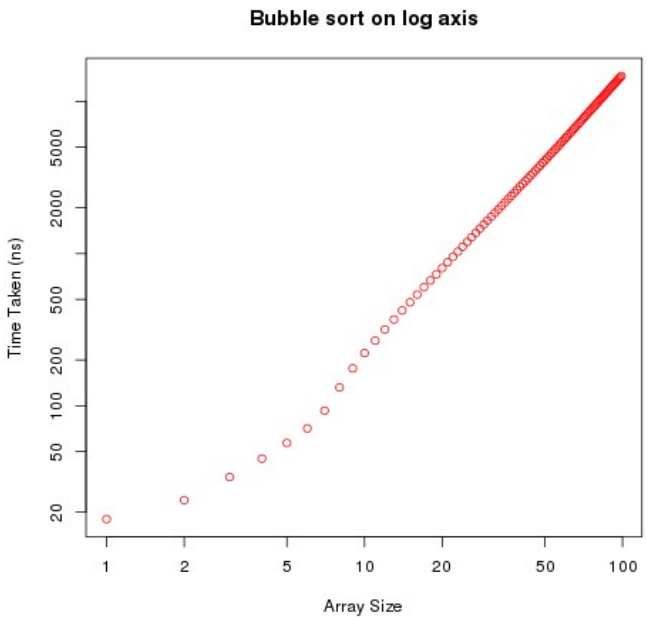


Graph 1: Bubble Sort plotted against a quadratic curve

A bubble sort's complexity is O(n^2). The reason for this is down to how a bubble sort operates. To perform a single swap the algorithm must go through the entire array checking if the selected element needs swapped with the next one. It must do this for each item in the list or in a loop for the length of the array. Therefore it has a complexity of O(n^2). This can be seen in Graph 1. Graph 1 contains a bubble sort which was iterated 10 million times through an array of size 100. This was done to produce a very reliable and smooth curves with no uncertainties. Plotted against this curve is a simple quadratic curve which was multiplied by a coefficient (explained in further detail later). As you can see from the graph the bubble sort follows the same shape as quadratic graph backing up the fact that it is O(n^2).



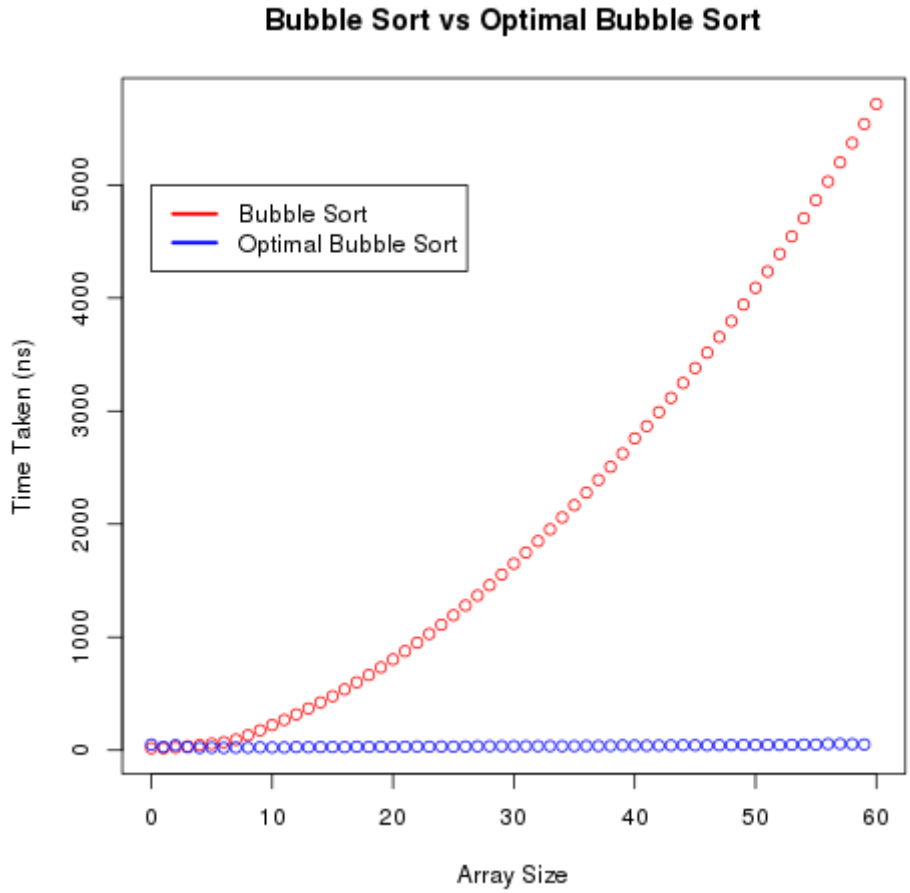
Graph 2: Bubble sort plotted using an array of size 120 000.



Graph 3: Bubble sort plotted on log axis producing a gradient of 2.

I also performed another two test runs using bubble sort to help demonstrate the O(n^2) complexity of bubble sort. I ran a test over an array of size 120 000 to show that the bubble sort would continue its quadratic like curve. Finally I graphed the same data used for Graph 1 again but this time I took the log of each data point before plotting it. This produced a straight line of gradient 2. In mathematics if you plot a polynomial curve on log axis, the gradient of the line produced is equal to the power of the polynomial, again demonstrating the O(n^2) complexity of the bubble sort.

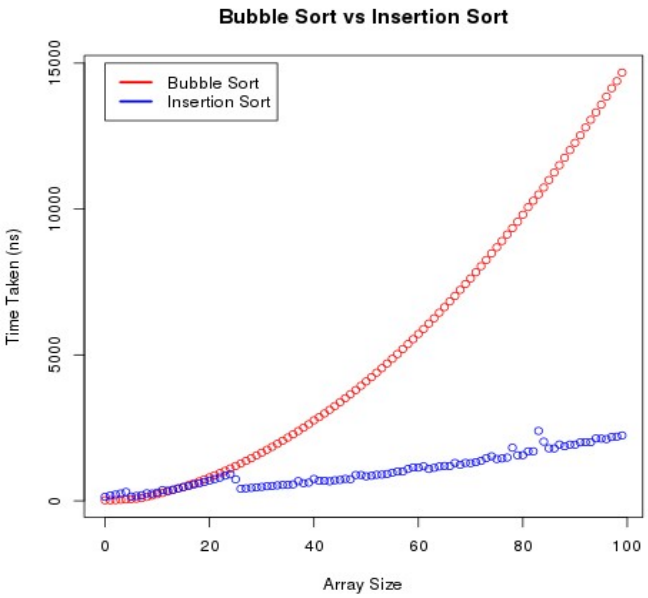
Additionally I also demonstrated, how if you include flags in a bubble sort it can, produce better results. The optimised bubble sort checks to see if the array is sorted every time it finishes going through the list. This could save a lot of time if the array is already sorted or partially sorted. The evidence of this can be seen in Graph 4 below.



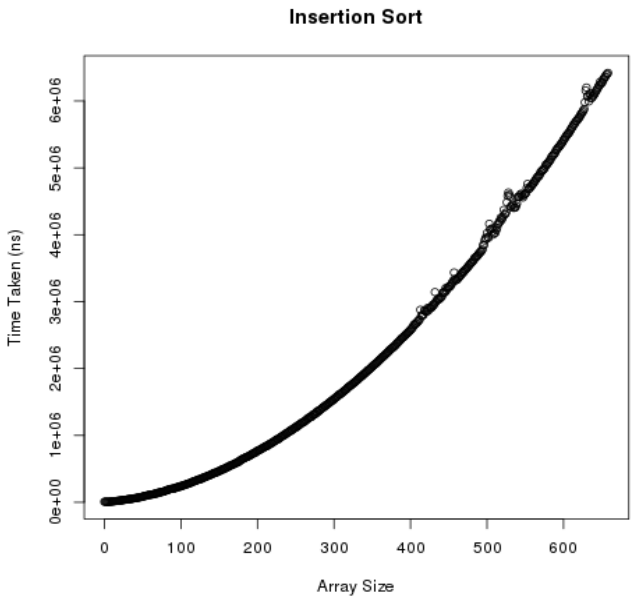
Graph 4: Bubble Sort plotted against an optimised Bubble Sort sorting through a pre sorted array.

Bubble Sort vs Insertion Sort

From the data demonstrated so far you can see that the bubble sort follows O(n^2). But it doesn't follow the perfect n^2 curve and this is due to the coefficients mentioned earlier. A coefficient is the part of the complexity that has been cancelled out. This is due to the fact that big-O notation shows complexity as n tends to infinity. So this explains why there was a coefficient added to the quadratic curve in Graph 1. The effects of coefficients can be seen when comparing two sorts of same complexity. I.e bubble and insertion.



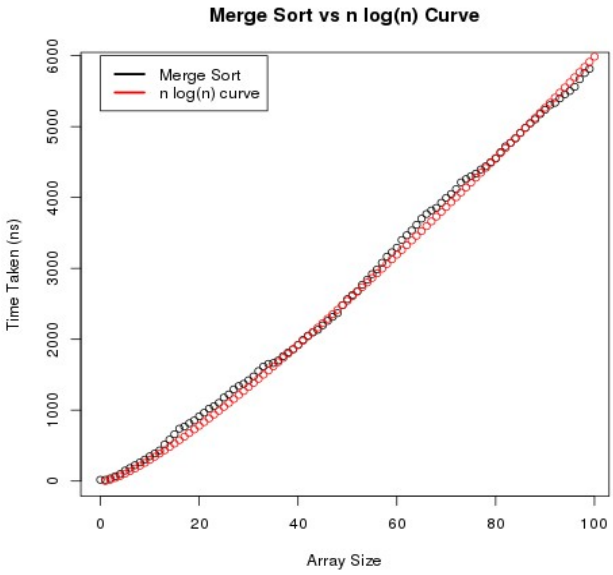
Graph 5: Bubble Sort plotted against Insertion Sort.



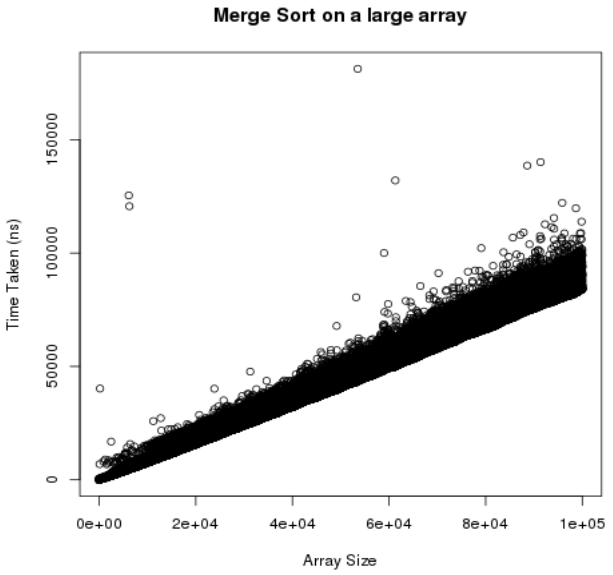
Graph 6: Insertion Sort of a large set of data.

In Graph 5 you can clearly see that an insertion sort does not give the same results as a bubble sort even though they have the same complexity. And from Graph 6 you can see that an insertion sort does eventually show its quadratic nature, it just takes longer than a bubble sort to do this. Also in Graph 5 after array size 20 the insertion sort suddenly becomes more efficient and this is due to java. After a bit of the sort has run java uses Escape analysis to make the sort quicker.

Merge Sort



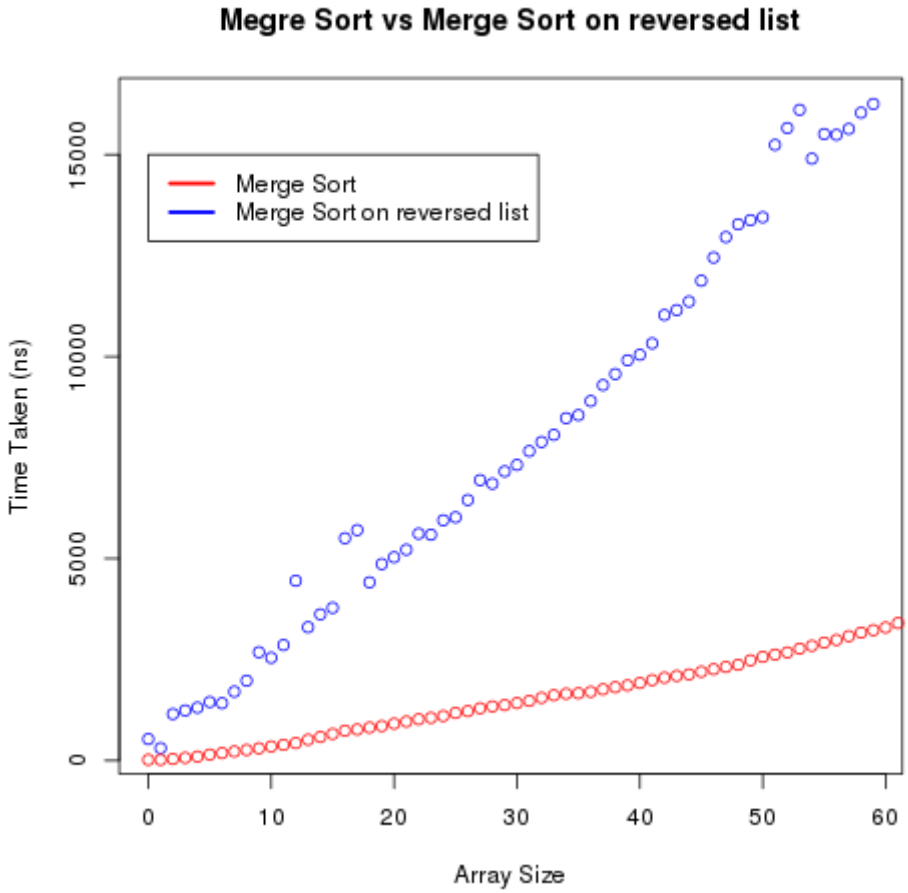
Graph 7: Merge Sort plotted against a logarithmic curve



Graph 8: Merge sort plotted using an array of size 1o0 000.

A merge sort's complexity is O(n(log(n)). The logic behind this is because after each pass through the array the size of the sorted subsections doubles, so after each pass through the array it is growing exponentially. So it will take log(n) doubling of the subsections to reach the required end size. And the n comes from the fact that you are continuously applying the same operation to the elements. This can be seen in Graph 7 when the merge sort is plotted against a n log(n) curve, which again has been multiplied by a coefficient. Finally in Graph 8 you can see even after the sort is run over a massive array size it keeps it n log(n) shape, proving its complexity.

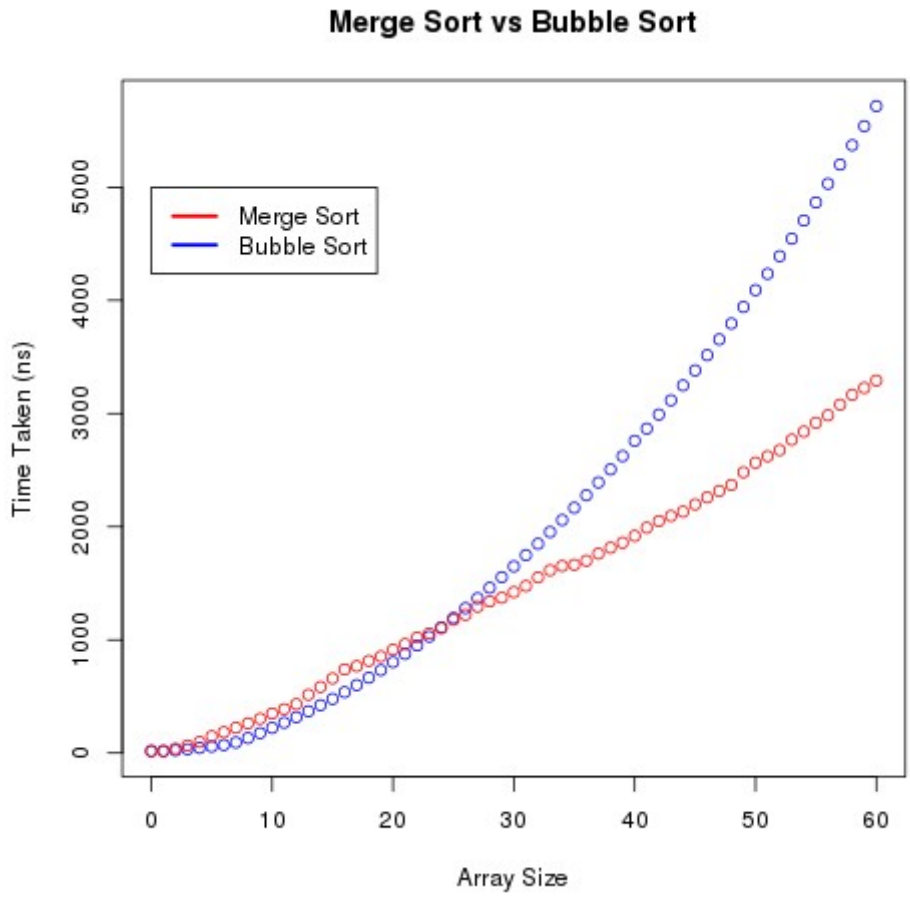
Additionally I created a graph (Graph 9) to show the difference between a merge sort sorting a random set of data and a merge sort sorting a reversed set of data.



Graph 9: Merge Sort on random data set plotted against Merge Sort on a reversed data set.

You can clearly see that the time taken to do a sort on a reversed list is considerably longer.

Bubble Sort vs Merge Sort



Graph 10: Bubble sort plotted with Merge Sort.

As you can see in Graph 10 the bubble sort is initially faster. But after the array size increases to 25 the merge sort becomes faster. This shows that a bubble sort is better for shorter lists and merge better for longer lists. The graph also shows the different behaviours of sorts with different complexities.

Evaluation

I feel my practical was successful in showing how bubble sort and merge sort's speeds change as the size of the problem changes. By taking in data, analysing it, then graphing it I showed how each sort behaves and why they are the complexities they are along with what situations each one benefits. I also included analysis for insertion sort as an extension.

Conclusion

Overall I feel I have achieved the task that was set. The practical allowed me to gain a better understanding on how complexities work and why they can't completely describe a sort. If I had more time I would of also compared a quick sort.