A comparison of sorting algorithms based on the time complexity.

Ryan Gouldsmith[*]
Department of Computer Science, Aberystwyth
Llandinam Building, Aberystwyth University
Aberystwyth, Ceredigion, SY23 3DB
ryg1@aber.ac.uk

ABSTRACT

This report contains experimental data which compares a selection of sorting algorithms based on the the Big O notation, and clearly states the time it takes to perform these sorts on variable sized data sets along with basic pseudocode for each sorting method. The report will compare the different sorting algorithms in the Java Programming language. There have been five different sorting algorithms compared: BubbleSort, QuickSort, RadixSort, InsertionSort and ShellSort. There will is graphical representations which re-enforce the authors statements, along with a comparison of the 5 different sorting Algorithms; there is discussions on the which is the quickest, do datasets matter and if the simpler algorithms are quicker.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*; E.1 [**Data**]: Data Structures—*Arrays,Lists, Stacks and Queues*; F.2.2 [**Nonnumerical Algorithms and Problems**]: [Sequencing and scheduling, Sorting and searching]

General Terms

Experimentation, Performance, Algorithms

Keywords

Sorting,BubbleSort,QuickSort,ShellSort,InsertionSort,Time Complexity.

1. INTRODUCTION

Sorting algorithms are the essential building blocks for anything in Computer Science; without a proper understanding of the logistics behind a variety of sorting techniques then you could find that the application's time taken for execution could grow exponentially. As a result, having a key understanding of the time complexity of an algorithm will be vital in assuring you have the confidence that your application will optimised to the best possible time complexity.

When considering the time complexity of a sorting algorithm we have to consider: *Best, Average* and *Worst* scenarios. For the majority of the sorting algorithms the best, average and worst all have the different time complexities and as a result each sorting algorithm will depend on each data set.

1.1 Methododology

For this report each algorithm will be tested on a variety of datasets, which will contain the same values for each different algorithm. The data will be collected from the experiments by running the algorithms under the same conditions and testing on identical datasets; this will produce a fair trial as the results will be consistent across the platform. Each algorithm will produce a time in milli seconds on how long it took to sort the data, which uses a pre-built timer. Each algorithm will have a median time taken for each sorting implementation and these values will be used in the visual representation of my results. Finally there have been 3 different sorts implemented alongside the existing sorts implemented[1] therefore choosing to do an *O(n log n)*, $O(n^2)$ and a *O(1.5)* time complexity algorithm.

The experiment will be running on an Intel Core i3-2350M CPU @ 2.30GHz, with 4GB of RAM machine.

2. THE EXPERIMENT

The experiment that is being conducted is to see which sorting algorithm is better under the size of datasets generated randomly. The dataset sizes, which the algorithms will be tested under is: 1000, 2000, 5000, 10000, 20000, 50000, 100000, 200000, 500000, 1000000, 2000000 and 5000000 data items. The following equation, will be used to ensure consistency within my conduction of the tests:

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} \qquad (1)$$

As the experiment is checking for the elapsed time of the algorithms then the tests will be ran n times, when n = 3 and then take the mean average of the results. The resultant of the mean will then be used as the accurate data plot for the the graphical representation and the comparison.

[*]Only Author

[1]See acknowledgements

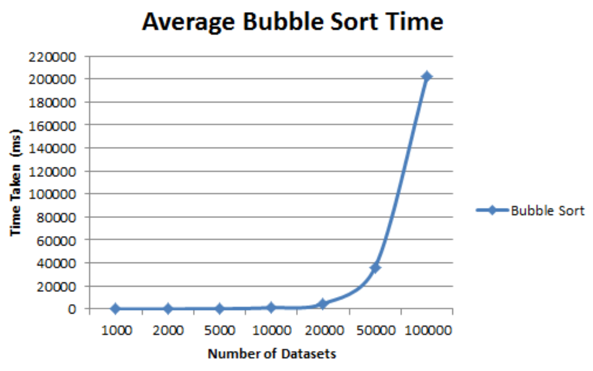

Figure 1: Overall view of Bubble Sort

2.1 Bubble Sort

The bubble sort algorithm is generally considered to be an inefficient algorithm due to its time complexity, which is: *Worst:*$\mathbf{O(n^2)}$, *Average:*$\mathbf{O(n^2)}$, *Best:*$\mathbf{O(n)}$[10] as a result of this complexity then it's often considered that there are other algorithms which are better for larger datasets; as n increases with the bubble sort then it begins to tend towards the worst case scenario. However, if a list is sorted then the resultant complexity tends towards the Best time complexity.

2.1.1 Pseudocode

Below is the psuedocode for the Bubble sort, the implementation of this psuedocode could be implemented in any programming language.

```
function BubbleSort(array){
  tempArray
  for i =0 to arraylength do
    for j = 0 to arraylength − i do
      if array[j] > array[j+1] then
        tempArray ← array[j+1]
        array[j+1] ← array[j]
        array[j] ← tempArray
}
```

From the psuedocode above, it is evident that because of the two for loops which loop around the length of the array then it will be on average an $O(n^2)$ algorithm. On each execution of the loop then move the highest value further up the array; this is done in the inner loop. Once the values have reached the end of the array, it will increment the outer loop and repeat. [3] This solution could be optimised even further by adding a flag checking whether any items have been swapped, this would reduce the amount of comparison, but it wouldn't reduce the time complexity to anything lower than a normal bubble sort, but it will improve the speed. [1]

2.1.2 Results

Figure 1 shows the results of the bubble sort.

2.1.3 Explanation Of The Results

Figure 1 shows the graphical representation of the bubble sort over the tested values, 1000 to 100000. There wasn't tests on > 100000 tests as soon the algorithm becomes to inefficient and shows it's worst case scenario. Therefore, it becomes evident that the higher the dataset then the slower this algorithm becomes. The shape of the graph is evidently a $O(n^2)$ especially around 50,000 data items, where the shape of the graph rises in a steep curve, whilst up to around 10,000 data items the results rise at a constant rate up to 1103ms. Where as when it reaches 1,000,000 data sets the time taken is 2012183.3ms, thus showing a distinctive increase in the time taken. See *Table 1.*

2.2 Radix Sort

The Radix sort is considered to be a good algorithm for when the data sets are very small as the time complexity of this is: *Worst:* $\mathbf{O(nk)}$ *Average:* $\mathbf{O(nk)}$ and *Best:* $\mathbf{O(nk)}$ [10] [7]

2.2.1 Pseudocode

The pseudocode below is for a radix sort of integers

```
function RadixSort(array){

      digits ← maxlength
  for i to RadixValue do
      create a Queuearray

  for j← digits to j > 0 AND value ← 1 to
       factor * RadixValue do
      decrement j
  for k to array length do
      stringval ← array[k]
      if j < stringval length
         index ← stringval char at d
      else
         index ← 0
      Add items[j] to the Queuearray
  for n to RadixValue do
      items[n] ← Queuearray[n]

}
```

Above is the pseudocode for the radix sort. The concept behind this is to split each of the array items based on the last digit in the array item; these are then grouped together, but there are no swaps in the sub arrays and they remain in the same order as prior. This procedure is recursed up to the first index item. Once we reach this index, we collaborate all the sub sorted items together into one dataset. The sorting method itself can be any stable sort[2] such as bucket sort. This implementation used a queue

2.2.2 Results

Figure 2 shows the overall results for the Radix sort

2.2.3 Explanation of the Results

With Radix sorts time complexity being a constant $\mathbf{O(nk)}$ then it's expected the algorithm would be able to be able to sort the entire 5 million data items. To do this however the size of the heap in the JVM had to be changed, else the application would run out of memory.To do this the following command was added as an argument: `-Xmx4g` to request

[2]Stable Sorts are if the keys remain in the same entry, before sorting aswell as after sorting.[8]

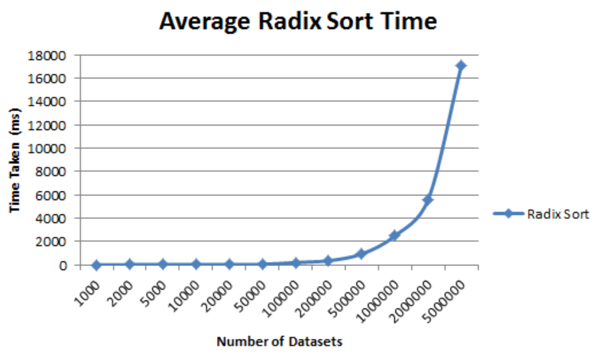

Figure 2: Overall view of Radix Sort

that the maximum memory usage was 4gb. However, the sort managed to solve the smaller dataset times very quickly, solving the 1000 dataset in 10.3ms. In comparison to that of the quick sort below, the radix sort had a similar shape on the graph to the Quick sort, but at around 2 million data sets, the growth is arguably more rapid and completed the sorting in 5594.3ms. Finally, on average the quick sort was still quicker than the Radix sort, but considering that Quick sort is an *O(n log n)* algorithm the Radix sort was very quick, and managed to sort 5,000,000 data items in 17,053.6 ms which results in a 42% slower algorithm execution time than the Quick Sort *See Table 1*

2.3 QuickSort

The Quicksort algorithms is a good overall algorithm, which has a very good time complexity. As a result, this will be algorithm which I will be running as my **O(n log(n))** algorithm. The time complexity for this sort are very efficient, especially on the larger data sets. They are: *Best:* **O(n log (n))**, *Average:* **O(n log (n))** and *Worst:* **O(n^2)** This an example of a divide and conquer sort which uses a pivot as the center point of sorting, thus making it a comparison based sort. [2]

2.3.1 Pseudocode

Below is the pseudocode for creating a quick sort implementation.

```
function Quicksort(array, right, left){
        pivotValue ← choosepivot(array,
            left, right)
        Quicksort(array, right, pivotValue)
        QuickSort(array, pivotValue, left)

}
```

Above is the algorithm which will recursively call itself until all the values have been sorted. It uses a choose Pivot function which returns a pivot value.

2.3.2 Results

Figure 3 shows the results of the average QuickSort time

2.3.3 Explanation of Results

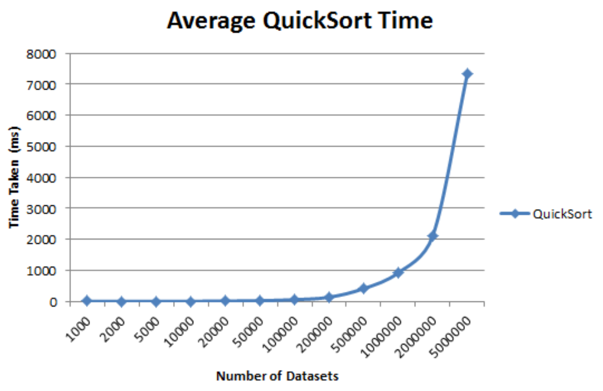

Figure 3: Overall view of Quick Sort

The quicksort algorithm really did yield 40% increase over the $O(n^2)$ time complexity. Mainly, as the quick sort managed to sort all the dataset values, all the way up to 5 million, this took 7,342.3ms to perform. As you can see it was performing around a *O(n log n)*. However when it reached around 2 million data items, the values began to rise exponentially and then the algorithm began to show it's worse case scenario of $O(n^2)$, as at 2,000,000 data items the time took 2,120.6ms whereas for the 1,000,000 dataset values it took 925ms, having a 100% increase. So, overall the algorithm performs on the results collected to the average case scenario up until the 2,000,000 data item, which it then spikes and looks like an $O(n^2)$ complexity. There however was a fluctuation between the 2,000 and 10,000 dataset item: where 2,000 took 4.3ms, 5,000 took 2.6 ms, and 10,000 took 4.3ms. This happened to be a trend throughout the faster algorithms and this can conclude that the conditions on finishing an algorithm on these datasets, which may have been sorted more than others, more efficient.

2.4 ShellSort

Shellsort is considered to be an improvement another sort called Insertion Sort. As a result, the time complexity of this algorithm is considered to be a lot better than others: *Best*: **O(n)**, Average: **O($n^{1.25}$)** and *Worst:* **O($n^{1.5}$)** time complexities. So, from a quick glance that it should yield faster results than a bubble sort will produce results between O(n) and $O(n^2)$ [9]

2.4.1 Psuedocode

Below is the Psuedocode for a Shell Sort. [11]

```
function shellSort(array){
  for i to array length do
        for j to array length do
          tempvalue ← array[j]
        for k = j to k > j and tempvalue <
            array[k−i] do
                  array[k] ← array[k−i]
                  array[k] ← tempvalue
```

The shell sort algorithm, is considered to be a complex algorithm in terms of calculating the complexity. However, it is one of the oldest sorting algorithms. The shell short is very

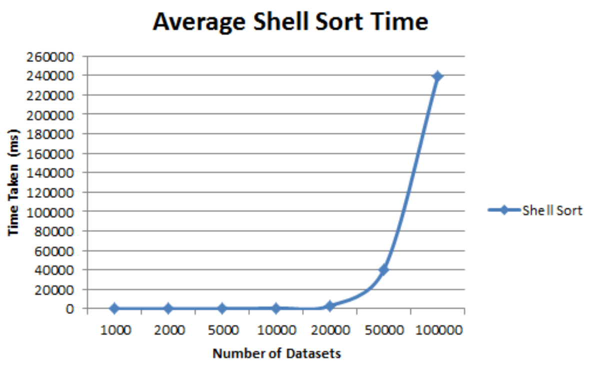

Figure 4: Overall view of Shell Sort

similar to the implementation of an Insertion Sort, where we insert the values into the correct position depending on the value of the item. However, by comparing the temporary value against the array item inside the loop execution, it should mean that the Shell sort should yield a faster execution of the larger datasets. [4] However, in general shell sort analysis is considered to be a difficult, as there tends to be a lot of mathematical analysis involved. [4]

2.4.2 *Results*

Figure 4 shows the results for the average ShellSort Time

2.4.3 *Explanation of Results*

The shellsorts performance could only go up to 100000 data items, as then the sort began to get too slow to perform. In the case of our randomised data files then it was the same number of data items that a bubble sort could execute without the algorithm taking too long to execute. (see fig 1.) However, the shell sort was generally quicker than the bubble sort, but not a considerable amount quicker, which indicated that it was tending towards the worst case scenario rather than the average and best. However when it reached the 100,000 dataset the time taken was, 239235 ms, which showing on *figure 4* is the exponential increase. However, the shell sorts time to complete the algorithms increased at around 50,000 data items where the time taken was 40,655ms and prior to that the 20,000 took 3111.6ms. This shows the sharp increase, tending to the worst case scenario for the time complexity *See Table 1*

2.5 Insertion Sort

Insertion Sort has a time complexity of: *Best:* $\mathbf{O(n)}$, *Average:* $\mathbf{O(n^2)}$ and *Worst:* $\mathbf{O(n^2)}$ [10]

2.5.1 *PseudoCode*

Below is the pseudocode for an Insertion Sort [6]

```
function insertionSort(array){
 for i to arraylength do
     for k =i to k >1 AND array[k] <array[k
         -1] do
         temp ← array[k]
         array[k] ← array[k-1]
         array[k-1] ← temp
}
```

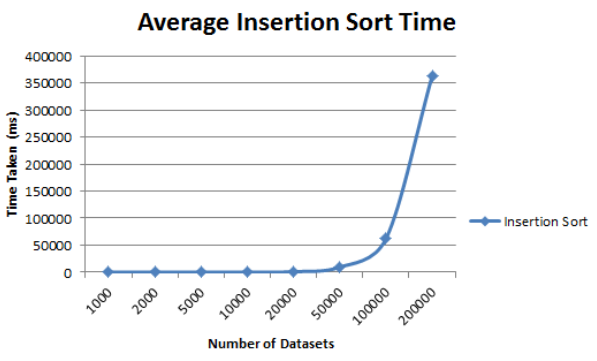

Figure 5: Overall view of Insertion Sort

The Insertion sort itself is very self explanatory: you store the value which will be replaced in a temporary variable, and then perform a simple swap. When you choose an item, the sort will insert it into the correct position in the array [5]. As a result because it needs to complete these comparisons for each item on the whole array, it results in the algorithm having two for loops - which is why it has a time complexity of $O(n^2)$. But, because it has a comparison inside the for loop then it makes the time taken quicker than other $O(n^2)$ sorting algorithms, such as the bubble sort. [5]

2.5.2 *Results*

Figure 5 shows the results for the average time Insertion Sort

2.5.3 *Explanation of the results*

The Insertion sort on my datasets was rather slow. It did manage to get up to 200,000 data items however, the time taken to get to reach 5,000 data items was quicker than the Shell Sort - for the Insertion sort 5,000 data items took 52.3ms.*Figure 5* shows a beginning of a steep curve after the 100,000 dataset items to which the time taken was 62,160.6ms. Initially the datapoints were fairly consistent with majority of the other sorts, and that it only got to where n = 100,000 that the time was increasing exponentially. Prior to that the 1,000 data items took 15.6ms to sort fully; the radix sort only yielded a 50% better execution time. However, when we reached the 200,000 dataset items the time taken was 363,300ms which is roughly 1010% increase over that of the Radix Sort at the same data set values; this shows where the algorithm becomes to get inefficient.

3. COMPARISON OF THE SORTS

3.1 Differences in time

The experiment concluded that the quickest Sort out of the sorts that were tested was the quick sort, this was to be expected as since it had an average time complexity of *O(n log n)*. By comparing the two sorts which did the entire data set, Radix and Quick, it can concluded that: to sort the 1000 data set items the quick sort took 6 ms to complete, whereas the Radix sort took 10.3ms. For the top end of the data values, for 5,000,000 data items: Radix sort took 17,053.6 ms whereas the Quicksort took 7342.3 ms. So even comparing the the two algorithms which managed to sort all the test data there was still a large yield in speed from

Table 1: **Algorithms time taken in ms against the Number of data items.**

Sort	1000	2000	5000	10000	20000	50000	100000	200000	500000	1000000	2000000	5000000
Quick Sort	6	4.3	2.6	4.3	9.6	23.6	55	136	412.3	925	2120.6	7342.3
Radix Sort	10.3	16.3	15	25.6	17	48	200.6	359	938.3	2491	5594.3	17053.6
Insertion Sort	15.6	8.3	52.3	249.6	1070	9476.3	62160.6	363300	n/a	n/a	n/a	n/a
Bubble Sort	19.6	42.6	252.6	1103	4723.6	36418	202183.3	n/a	n/a	n/a	n/a	n/a
Shell Sort	18	24	155.3	702.3	3111.6	40655	239235	n/a	n/a	n/a	n/a	n/a

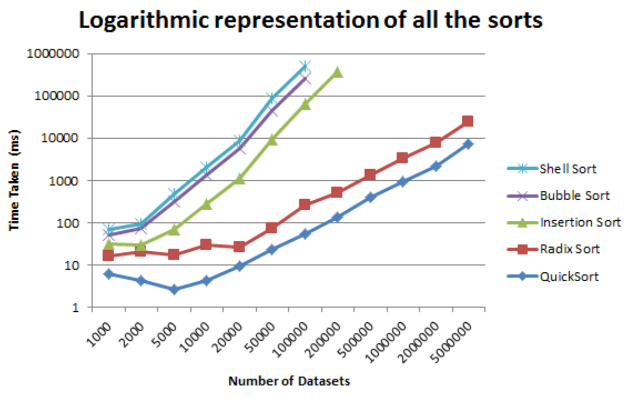

Figure 6: Overall logarithmic representation of all the sorting algorithms

the Quick sort. Next, the bubble sort and insertion sort have very similar time complexities, even though Insertion is considered to be quicker. For 1000 data sets: Insertion took; 15.6 ms, and Bubble Sort took 19.6ms. and for the top end of the data sets of 100000, since that is what the bubble sort went to:Insertion took: 9476.3ms, whereas the Bubble sort took 36418ms. So overall, there was a difference in the times taken, and they lived up to expectation especially for the slower sorts, compared to more optimised sorts of the same time complexity, *table1* shows the average time complexity.

3.2 Change over points

From *Figure 6* we can conclude that there are no cross over points in the data. As a result on the data in which we tested against then there is no proof that the data will cross over at any given time. From looking at the graph, we can assume that they're all increasing at their own exponential rate, and thus don't seem to be crossing over any time soon.

3.3 Faster Simple sorts

As all our values are randomised, then simple sorts will never be faster than other complicated implementations. By looking at the experimental results it shows that the other algorithms such as quicksort and radix sort would always have a greater yield over sorts such as bubble and shell, and produce quicker times than all the others. The only time in which the simpler sorts maybe quicker is when the values are already sorted and algorithms such as the Insertion Sort will perform around at $O(n)$ time complexity - which is quicker than an $O(nlog(n))$ algorithm.

3.4 Dataset size

The size of the data set does make a difference, especially in terms of time taken to execute the algorithm. Because the time is n value, then the values will increase the bigger n grows. As a result, and seeing from the results that the bigger the result set grows the longer the algorithm will take, especially the algorithms which have a poorer complexity and are more wasteful. However, there are a few anomalies: On bubble sort for 50,000 data items the average time was 36418ms but for Shell sort, which is considered to be a quicker algorithm, then it took 40655ms. So, this is a case where a better algorithm, at around the same complexity, is actually the slower of the two.

3.5 General purpose

Probably the Quick sort is the best general purpose sort, it is very quick and sorts the most amount of data the quickest. Additionally, the implementation of said algorithm isn't too strenuous and therefore a genuine good algorithm, with a good time complexity. Therefore, the QuickSort is the best general purpose algorithm out of those implemented.

4. CONCLUSIONS

There was 5 different sort algorithms presented this report, with each having differing implementations and varying time complexities. After conducting the experiments and working out an average for the datasets, it can concluded that there is a relation between the time complexity of the algorithm and the dataset size. This was particularly evident in the slower algorithms, as they struggled to complete the top end of the data items. In terms of memory, all the algorithms were quite resource intensive, but especially that of the radix sort which had to request more memory. However, the experiment has shown that choosing the right sorting algorithm is vital to ensuring the application is as quick as possible, and for each application there are different algorithms which adhere to different needs.

5. ACKNOWLEDGMENTS

The Author would like to thank Richard Shipman for providing the code for the BubbleSort and RadixSort as well as the timer method and an implemented Queue Class.

6. REFERENCES

[1] M. S. G. C. Canaan and M. Daya. Popular sorting algorithms. *World Applied Programming*, 1:63, 2011.

[2] M. S. G. C. Canaan and M. Daya. Popular sorting algorithms. *World Applied Programming*, 1:69, 2011.

[3] B. Kinariwala and T. Dobry. Bubble sort. `http://ee.hawaii.edu/~tep/EE160/Book/chap10/subsection2.1.2.2.html`, March 2014.

[4] R. L. Kruse. *Data Structures and Program Design third edition.* Wiley, 2014.

[5] R. L. Kruse. *Data Structures and Program Design third edition.* Wiley, 2014.

[6] D. R. Martin. Insertion sort. `http://www.sorting-algorithms.com/insertion-sort`, March 2014.

[7] R. T. Michael T. Goodrich. *Data Structures & Algorithms in Java.* Wiley, 2014.

[8] R. T. Michael T. Goodrich. *Data Structures & Algorithms in Java.* Wiley, 2014.

[9] T. Niemann. Sorting and searching algorithms:a cookbook. `https://www.cs.auckland.ac.nz//~jmor159/PLDS210/niemann/s_man.pdf`, March 2014.

[10] E. Rowell. Know thy complexities! `http://bigocheatsheet.com/`, March 2014.

[11] L. Sinapova. Sorting algorithms shellsort. `https://www.cs.auckland.ac.nz//~jmor159/PLDS210/niemann/s_man.pdf`, March 2014.