CS350 : A comparison of sorting algorithms

Hyunchan Kim

3/12/2015

Hyunchan@pdx.edu

1. Abstract

This project implements the comparisons of seven different sorting algorithms. In the proposal, I planned compare three sorting algorithms. However, I finally decided to have seven different sorting algorithms including the planned ones. They are Selection, Insertion, Shell, Radix, Merge, Quick, and Bubble sort. Everything is implemented in Java language and each algorithm is based on the fairly same environment.

2. Introduction

The first task is to analyze a minimum of three sorting algorithms and use each on a number of test samples of data. These samples of data range from small data sets to large data sets, with an initial 13 tests supplied and later 18 tests. On each test, the elapsed time from the start of the algorithm to the completion time would be recorded and compared against other algorithms in order to reach conclusions. From the results of the implementation, we are able to compare and contrast each of the algorithms to determine the optimized data set sizes for particular circumstances. We are also going to analyze to find the best case average and examine any particularly interesting results and crossover points between the algorithms where one becomes better than another with the different sizes of data sets. The sorting algorithms under test are Insertion Sort, Selection Sort, Shell Sort, Merge Sort, Radix Sort, Quick Sort, and Bubble Sort. For this report, we will basically discuss the specification of what an algorithm does, and how to write it in code.

3. Implementation

In order to perform the sorting algorithms, a testing class has been written to conduct single and multiple tests on single and multiple algorithms. Using this class, the algorithms are tested multiple times on a total of 18 different data sets, each with a differing number of items in the set. By implementing with many different varying sizes of data sets, it's possible to see what the effect of the size of a data set has on the different algorithms, and if on one size. However, it could be better on other larger or smaller sizes. These data set sizes ranged from 10 items to 5,000,000 items, and each number contained no more than 6 digits. Some data sets had repeating data, while others did not. These are shown in a table below.

Name	Num of items
Test1	10
Test1-a	20
Test1-b	50
Test2	100
Test2-a	200
Test2-b	500
Test3	1000
Test3-a	2000
Test3-b	5000
Test4	10000
Test4-a	20000
Test4-b	50000
Test5	100000
Test5-a	200000
Test5-b	500000
Test6	1000000
Test6-a	2000000
Test6-b	5000000

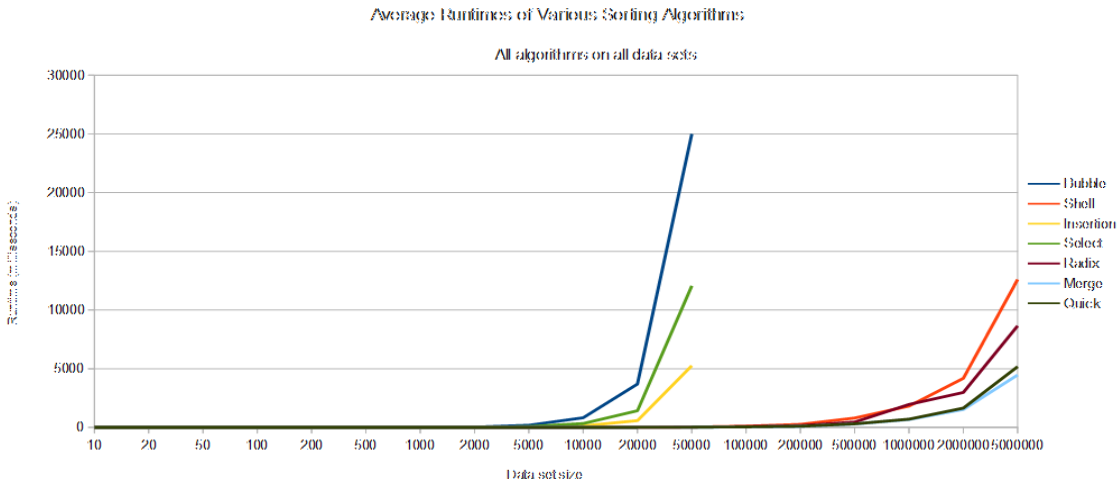
In order to have an accurate outcome, each performance runtime has been recorded in nanoseconds, with the Java command System.nanotime(). This is saved to a variable just before the call to begin the sort, and the nanotime is used again after the completion of the algorithm. This result is then deducted from the start time to result in the elapsed time. This time was then converted into milliseconds for representation in the spreadsheet of results. When implementing each algorithm on a given data set, the runtime could sometimes be skewed by the warm up of the processor as it adjusted to the algorithm, or a result might be abnormal if a sorting process was slowed down due to another process attempting to use the CPU at the same time. To combat these issues, the tests were set to run a minimum of 10 times before any elapsed times were recorded, and treated as a warm up run for the algorithm, Once an algorithm had been 'warmed' up on the system, the actual test iterations were run and the elapsed time between each run was recorded and added to a total time of all runs set for testing. For the purpose of this task, each algorithm was run on each set of data a total of 100 times. The total time was then divided by the amount of times the test was run in order to form an average run time. This average run time for each test is what was considered as the result for each sorting algorithm on each test. For a number of tests, it was necessary to cancel the sorting part way through, or to abort a test entirely. This was based on some tests taking more time than was reasonable to conclude.

4. Result

As each test concluded, below is a chart of these results upon completion, where the numbers shown in the cells are the average runtimes of the algorithm on a particular amount of items in a data set, displayed in Milliseconds. Those cells where 'DNF' is.

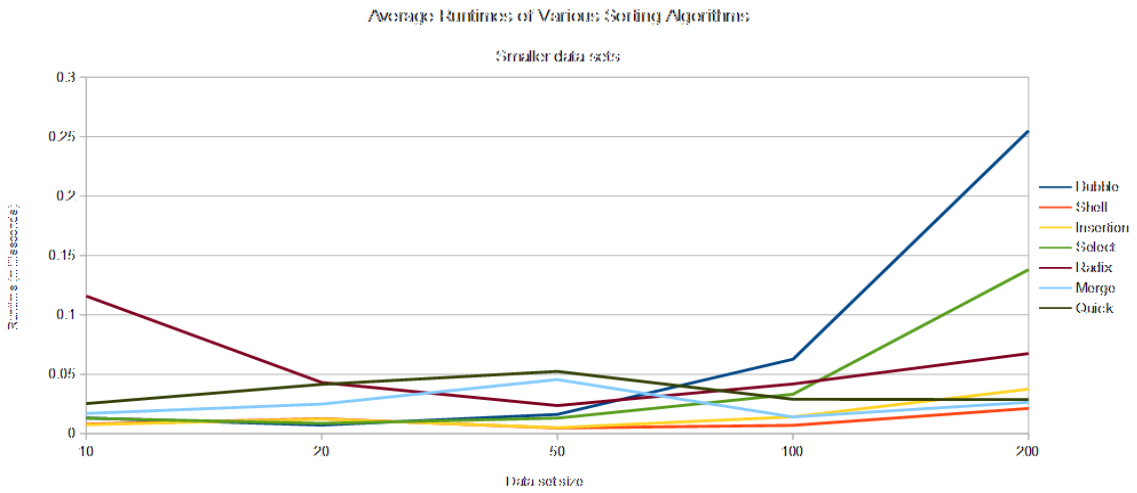
Test Set Size	Bubble	Shell	Insertion	Select	Radix	Merge	Quick
10	0.013219	0.008111	0.007606	0.013073	0.115863	0.017084	0.025373
20	0.007298	0.01269	0.012311	0.008664	0.043089	0.024943	0.041569
50	0.016286	0.004796	0.005135	0.013176	0.023692	0.045627	0.052429
100	0.062724	0.007077	0.014246	0.033244	0.041897	0.014111	0.029009
200	0.255041	0.02128	0.037511	0.138119	0.067472	0.026463	0.028752
500	1.721142	0.118429	0.272361	0.81872	0.18981	0.132431	0.103453
1000	6.954642	0.232347	0.959468	3.161142	0.323236	0.193064	0.19718
2000	28.538557	0.558958	4.1215	11.62406	0.63073	0.395953	0.406859
5000	186.265751	1.729134	26.66457	75.309439	1.568536	1.16239	1.179668
10000	819.756882	3.990999	126.899017	328.00296	2.664475	2.522151	2.613708
20000	3697.893734	9.357743	575.485706	1418.615488	5.278586	5.630202	6.007934
50000	25000	34.923499	5254.582966	12060.08207	21.000163	16.642173	17.970252
100000	*DNF*	104.732371	*DNF*	*DNF*	70.350368	39.906916	44.194461
200000	*DNF*	258.337628	*DNF*	*DNF*	158.38282	93.422975	97.429993
500000	*DNF*	796.996293	*DNF*	*DNF*	454.032771	290.643854	303.178977
1000000	*DNF*	1816.485003	*DNF*	*DNF*	1954.595204	665.837137	700.827472
2000000	*DNF*	4178.30651	*DNF*	*DNF*	*2976.057594*	1523.267786	1643.509526
5000000	*DNF*	12599.89226	*DNF*	*DNF*	*8651.021336*	4468.304037	5177.505199

As is visible from the results, the simple sorts, Bubble, Insertion and Select, did not finish or were aborted for the larger test sizes. This was due to the amount of time it began to take for each sorting algorithm to run the tests. The time it would take became unreasonable and so it seemed best to abort those tests. Even with not running these tests, it is apparent from the resulting graph the way in which these O(n2) time complexity algorithms would continue to run and how their results look.

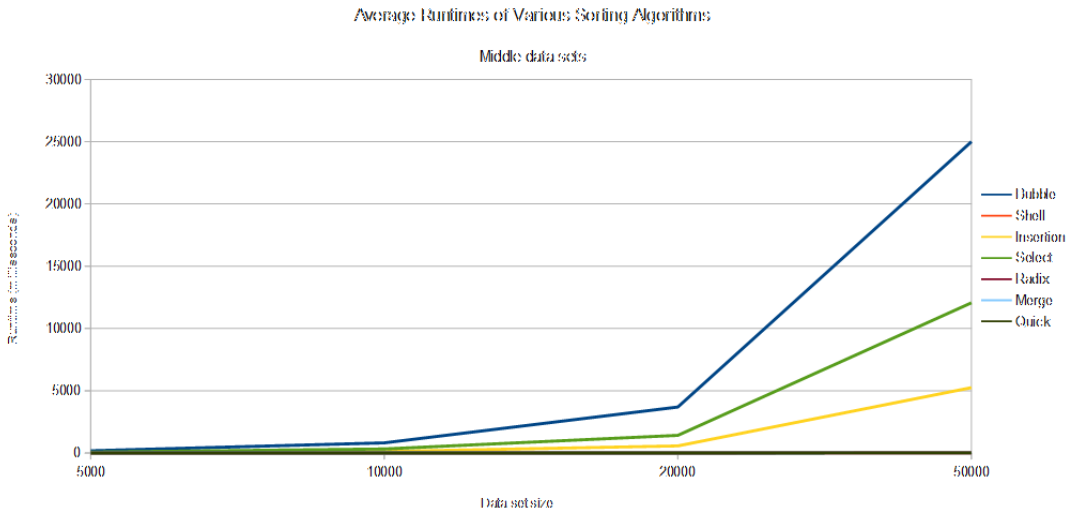


This line graph represents the full results contained in the spreadsheet, and stands as a way of comparing each sorting algorithm against the others. What immediately stands out here, is how the simpler algorithms (Bubble, Select, Insertion) begin to take much longer than those of O(nlog(n)) and O(nk) at around the 10,000 items in a data set mark due to their exponential growth rate. To break down the results and intelligently discuss them, the results have been

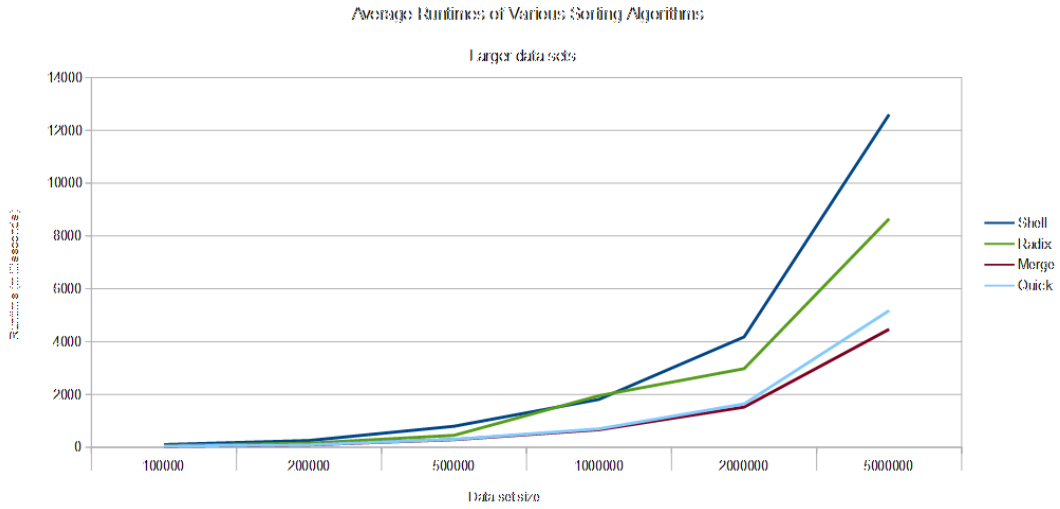
dissected into more appropriate forms of viewing. To begin, the smaller data (10 - 200) sets were looked at and analyzed to see which sort(s) is the most appropriate for smaller data sets.



What is most apparent, is that those O(n2) sorting algorithms that struggle with the larger sets actually outperform the more complicated algorithms for the data sets below 100-200 items. Notably, Insertion and Shell sort share very similar time complexities for these smaller data sets. As can be seen at the 200 items point though, Shell sort continues to perform with a lower runtime, while Insertion sort's time begins to break away and the algorithm takes longer to iterate through the list. This could be expected, as Shell sort is based upon Insertion sort, and could be considered as an improvement on Insertion. At the 200 data set test, it can also be seen that Quick Sort crosses over with the simpler sorts, beginning to perform at a similar rate as them. With any data sets less than this size, Quick Sort does not work efficiently compared to the other algorithms. It is in fact one of the slowest sorting algorithms for smaller data sets, with Radix sort being the absolute worst algorithm until a size of 50 or more items.



Another interesting thing to note is how Bubble sort's exponential growth is very visible by the point there are 100 items of data to process. It is at this stage that the more reliable algorithms cross over with Bubble sort, and shows why Bubble sort is fine for small data sets, but terrible when presented with anything moderately large. The same can be said about Select sort, which also becomes exponentially slower than Bubble sort at 100 items, just at a lower rate. This is backed up by when you look at the middle set of data sets tested for this report. While the other algorithms continue working at a relatively similar pace, Bubble, Select and Insertion sort grow exponentially. This leads them up to the tests they did not process, when they were no longer able to run within a reasonable time frame for the testing. Regardless, it is relatively easy to predict their expected outcomes for future tests based upon the results of the tests that were conducted on these algorithms.



At the other end of the tests, where the data set sizes reach ranges of 100,000 to 5,000,000, each of the remaining sorting algorithms begins to diverge from each other in their time taken to sort the items. Merge sort and Quick sort stay very similar to one another with their results, up until 1,000,000 items. At this point, Quick sort shows itself to be slower at sorting the data set than Merge sort, although not by much. If more tests on even greater data sets were to be made, it is probable that we would see Quick sort progressively slower in its run time than Merge sort. As for Radix and Shell sort, while both have up to that point performed similar to Merge and Quick sort, they both become slower than the others at the 200,000 items in a data set range. Shell sorts 'curve' on the graph becomes a lot deeper, a lot faster than Radix sorts does, and we see the point at which it gets incrementally more difficult for Shell sort to run on larger data sets in comparison to the other algorithms. A note to make about these tests, with particular attention to Radix sort, is that their runtimes and performances will be directly affected by the device they are run upon. If the device begins to have difficulties in processing such large data sets, it is very possible that the runtime would be significantly slower than they would be on a more powerful machine, even if the previous results are practically identical with smaller data sets on a less powerful machine. It is for this reason that it may be the curves become so much steeper so quickly with data sets at this size. However, regardless of the processing power of the machine and the impact it has upon the runtimes, it can still be taken as a valid result, since this report is comparing and contrasting the runtimes of these algorithms against each other.

Therefore, these results can be considered a real life example of a scenario where they are used on a device requiring them, and which algorithm is still the 'best', 'worst' and 'average' case, for ranges of data sets, small to large.

4. Conclusion

The overall conclusion to take from these results are that the size of a data set directly corresponds to the runtime performance of an algorithm, and that no one algorithm is best suited for every single data set, but there is an average best case in QuickSort, should an 'all-round' algorithm be required with little specification. A developer should be aware of the differences in run time between the different types of algorithms and therefore selective of the algorithm they choose to implement when using sorting in an application. They should select based upon their expected data set sizes, their space requirements and be prepared to have conditional statements to choose the best algorithm for a problem presented to the application.

5. Reference

Time complexity - Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Time complexity

Big-O Algorithm Complexity Cheat Sheet
http://bigocheatsheet.com/ Accessed:

Big O notation - Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Big O notation Accessed:

Time, space complexity, and the O-notation
http://www.leda-tutorial.org/en/official/ch02s02s03.html

For more information and code:
https://github.com/ace0625/CS350