Adam Beck

4/23/2016

Testing Report – Project 4 Algorithms Comparison

For the program, I obtained my pivot for quicksort by choosing the middle element of each subarray. For the Reporting1.java class, I wrote every output on the file, but I computed variances and means within the Reporting1.java class using a few helper methods. For the rest of the data, for sorted and reversed, I obtained the data by finding the median of three trials for the four different testing sizes. For the random array, I found the variance and mean of ten trials (each trial having different random values) for each of the four data sizes. It is important to note that my quicksort is slightly slower than what should be expected due to the fact that I did not uncomment the lines that choose a random pivot, thus, they contribute to the runtime. I chose to choose the middle pivot because choosing a random pivot actually would actually have the CPU cost of choosing that random pivot outweigh any benefits of choosing the random pivot in the first place.

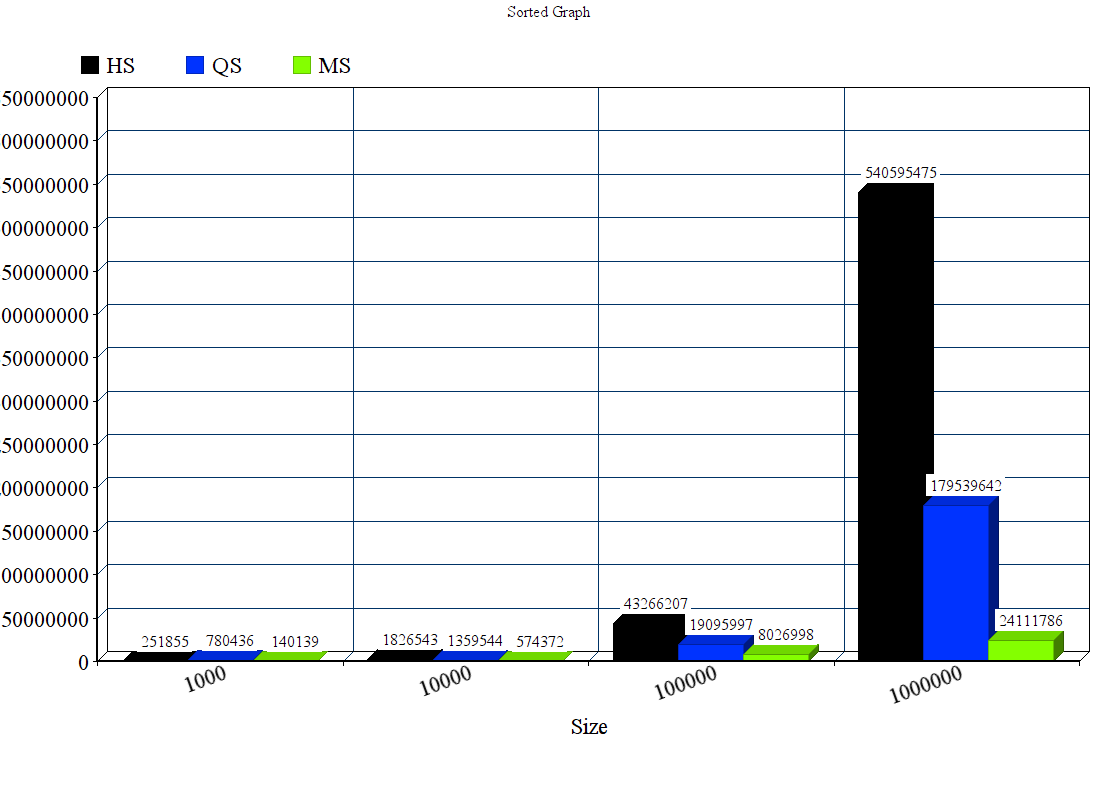
This is an analysis of my data collected from Report1.java. First, here is the median execution times for the sorted and reverse sorted arrays:

|  |  |  |  |
| --- | --- | --- | --- |
| Type of Input | Type of Sort | Size | Median Execution Time (nanoseconds) |
| Sorted | HS | 1000 | 251855 |
| Sorted | HS | 10000 | 1826543 |
| Sorted | HS | 100000 | 43266207 |
| Sorted | HS | 1000000 | 540595475 |
|  |  |  |  |
| Reverse | HS | 1000 | 84478 |
| Reverse | HS | 10000 | 3599398 |
| Reverse | HS | 100000 | 43197914 |
| Reverse | HS | 1000000 | 533771303 |
|  |  |  |  |
| Sorted | QS | 1000 | 780436 |
| Sorted | QS | 10000 | 1359544 |
| Sorted | QS | 100000 | 19095997 |
| Sorted | QS | 1000000 | 179539642 |
|  |  |  |  |
| Reverse | QS | 1000 | 118822 |
| Reverse | QS | 10000 | 1833649 |
| Reverse | QS | 100000 | 18344379 |
| Reverse | QS | 1000000 | 177686650 |
|  |  |  |  |
| Sorted | MS | 1000 | 140139 |
| Sorted | MS | 10000 | 574372 |
| Sorted | MS | 100000 | 8026998 |
| Sorted | MS | 1000000 | 24111786 |
|  |  |  |  |
| Reverse | MS | 1000 | 210011 |
| Reverse | MS | 10000 | 753198 |
| Reverse | MS | 100000 | 7269459 |
| Reverse | MS | 1000000 | 26856929 |

Additionally, here are the means and variances for the randomly sorted arrays.

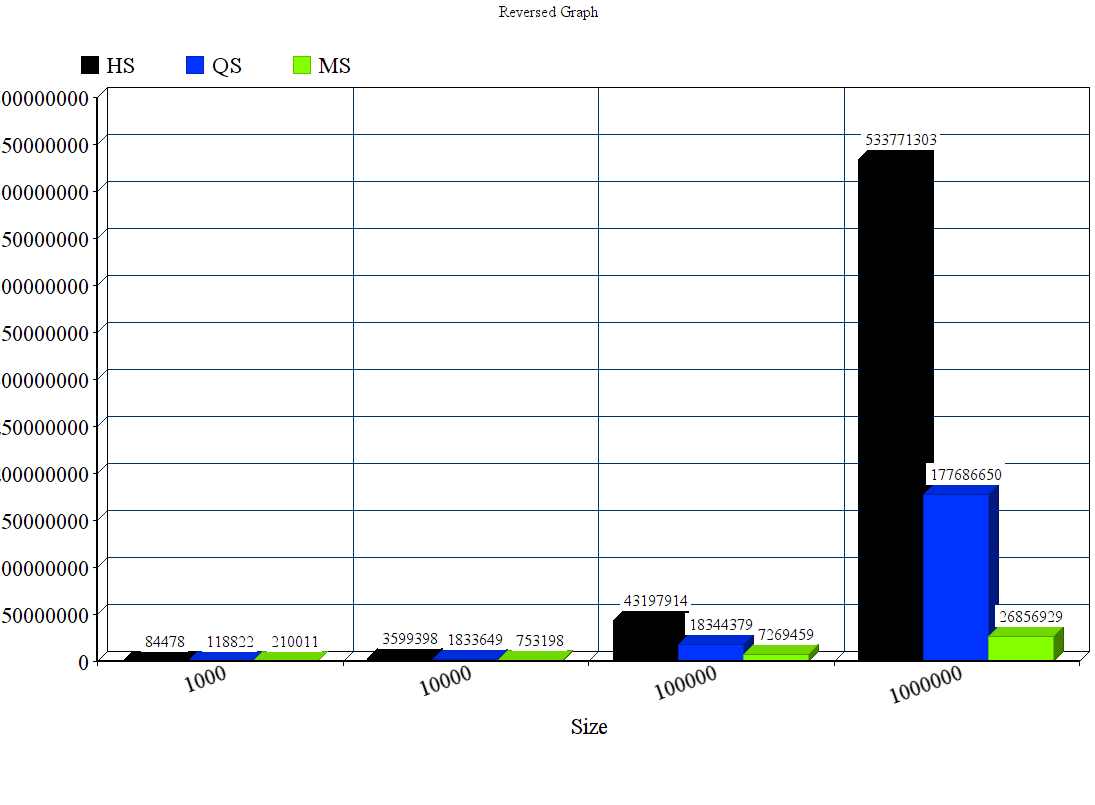
|  |  |  |  |
| --- | --- | --- | --- |
| Type of Sort | Size | Variance (nanoseconds) | Mean (nanoseconds) |
|  |  |  |  |
| HS | 1000 | 3.911101605E7 | 269303.5 |
| HS | 10000 | 3.414075470089999E9 | 3629479.1 |
| HS | 100000 | 4.3967264734129004E11 | 4.71536211E7 |
| HS | 1000000 | 3.9505701197930744E14 | 6.012756193E8 |
|  |  |  |  |
| QS | 1000 | 1560364.04 | 171048.6 |
| QS | 10000 | 6.476514186610001E9 | 1824687.7 |
| QS | 100000 | 3.1270094857921E11 | 1.87363737E7 |
| QS | 1000000 | 2.28331769600229E12 | 1.778100119E8 |
|  |  |  |  |
| MS | 1000 | 70249.84000000001 | 14803.4 |
| MS | 10000 | 1.4340423141000003E8 | 189286.3 |
| MS | 100000 | 1.80662353538E10 | 2277790.0 |
| MS | 1000000 | 1.6890429097516E11 | 2.58902472E7 |

Here are the graphs of size vs runtime for the sorted arrays. The y axis is in intervals of 50 million nanoseconds.



As we can see, the heapsort running time becomes the worst for any size over 1000 in this test, and becomes the worst by far. Quicksort consistently remains in the middle for any size over 1000 in this test, and merge sort is always the fastest. For size 1000, quicksort is slowest, following heap sort. Mergesort grows at an extremely slow rate.

Here is the graphs of size vs runtime for reversed arrays. The y axis is in intervals of 50 million nanoseconds.



This test is very similar to the sorted graph. Heapsort becomes worse by a big margin for sizes greater than 1000. Quicksort remains the middle for any size, and merge sort is always the fastest.

If I divide these results by the sorted results, I get a ratio on how efficient these reversed results were compared to the sorted results Here are the ratios (reversed / sorted) for the four sizes:

HS:

.335

1.97

.998

.987

QS:

.152

1.34

.96

.98

MS:

1.49

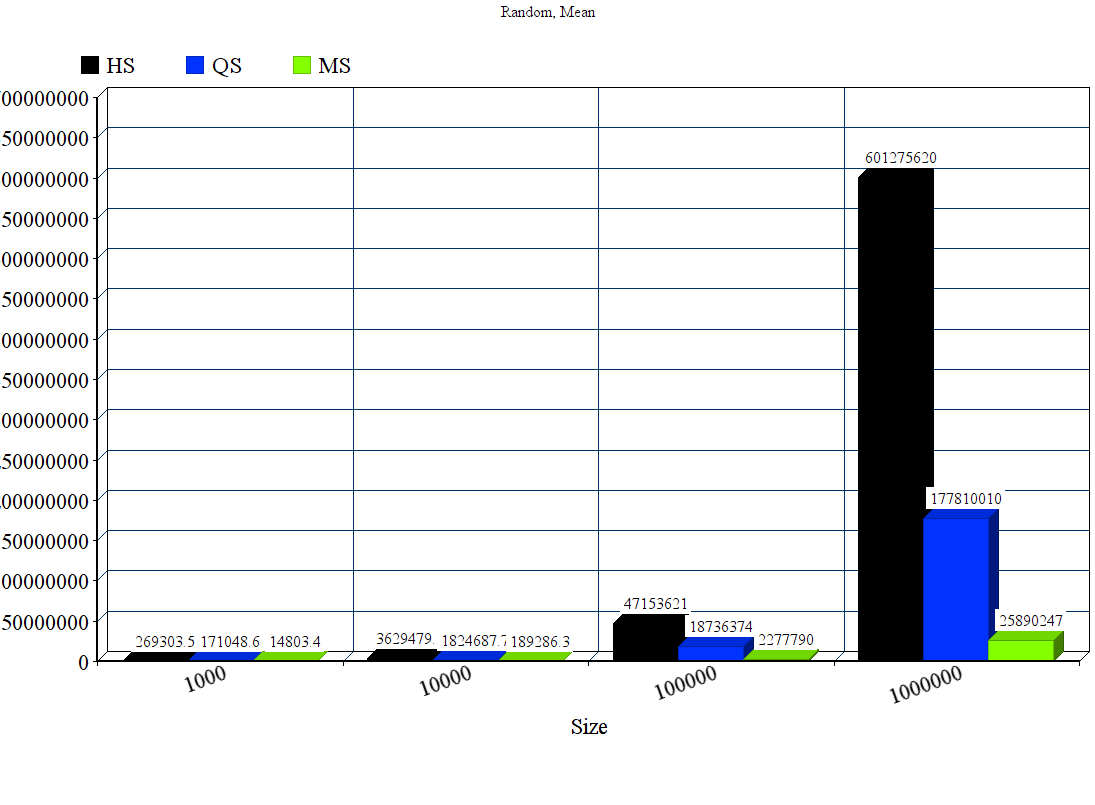
1.31

.90

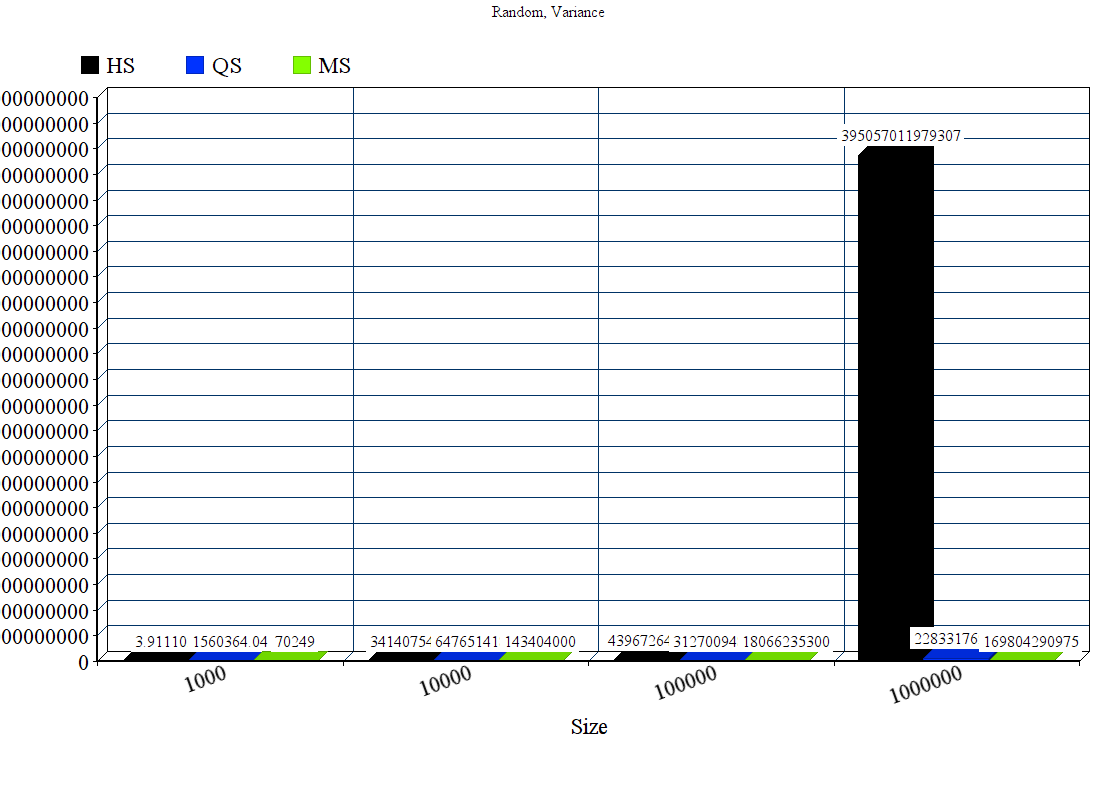
1.11

Overall, sorted vs. unsorted remains relatively the same rate at great sizes because the ratio becomes close to 1. Notable exceptions: Quicksort becomes almost 1/7 the speed reversed vs. sorted. Heapsort becomes 1/3 the speed reversed vs. sorted. In general, one could conclude that the trend is slightly slower for reversed vs. sorted for all data sets, but since there were only four sizes, I would conclude that there needs to be more sampling to conclude accurately.

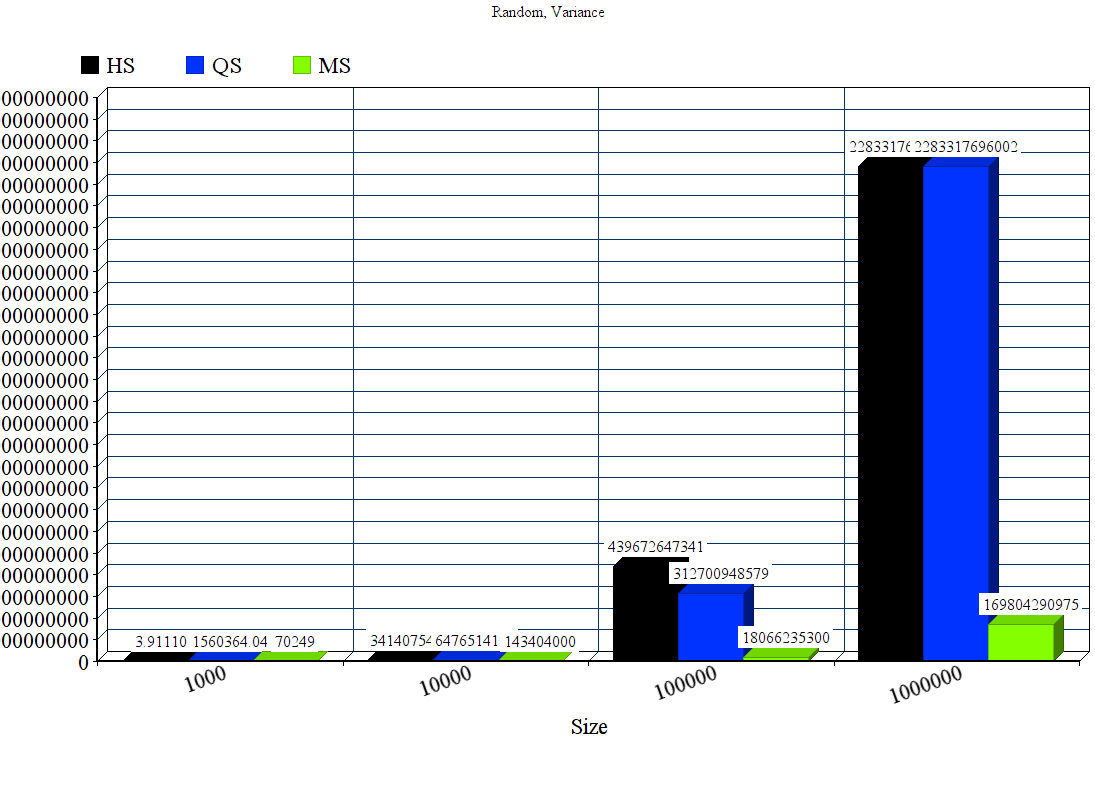
Next, I am going to examine the graphs for the randomly filled arrays. The y axis is in intervals of 50 million nanoseconds.

This is similar to the previous graphs. Heapsort is always the worst for any size in this testing. Merge sort grows the slowest is remains the fastest for any size. Quicksort is in the middle for every case. We can conclude that heapsort has the lowest mean running time for any size.

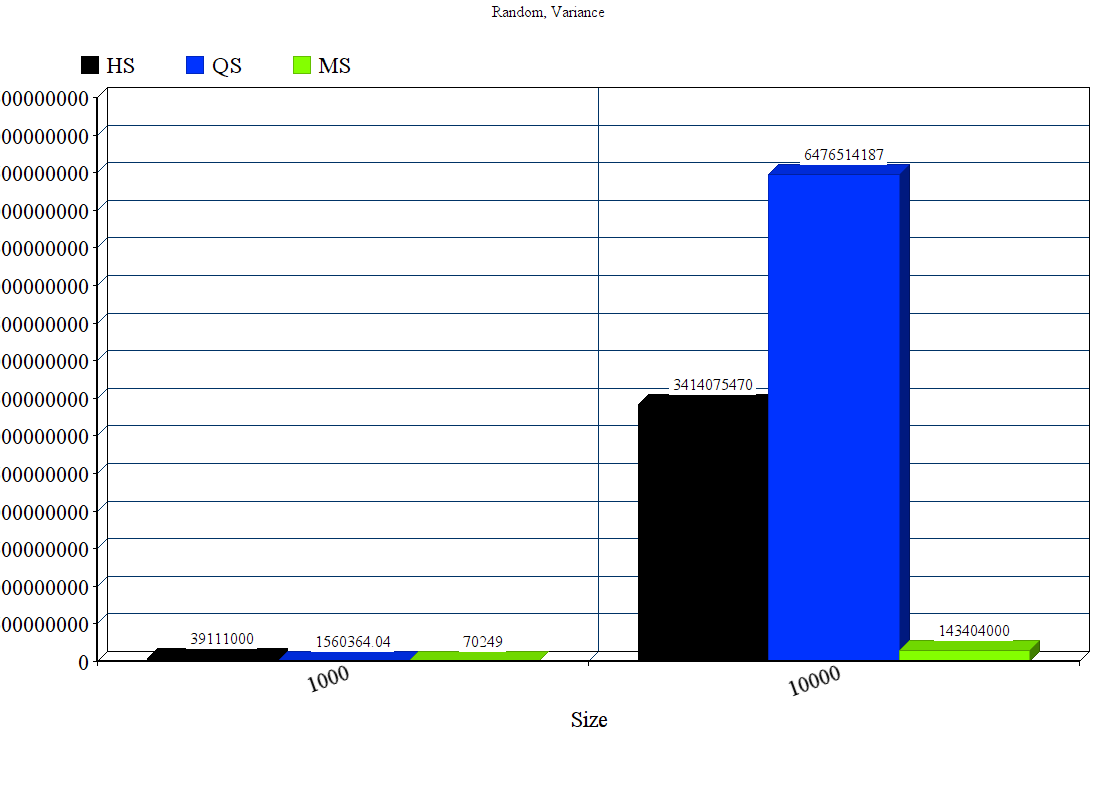
Finally, I am going to analyze variance to determine inconsistency.



As we can see, Heapsort has a variable many orders of magnitude greater than the other sorts. To analyze this graph further, I am going to assume the variance for Heapsort is the variance value for QS at size 1 million to see graph trends. Here is that graph below.

As we can see with this new graph, quicksort has a much higher variance at size 100000 and 1 million than merge sort, so therefore merge sort’s mean is not only a smaller time value, but merge sort is also more consistent with a tighter range of values, AKA more accurate at a large size.

Finally, I am going to analyze just size 1000 and 10000 for variance to see even more graph trends.

Quicksort has a higher variance than the other sorts for size 10000. It’s about twice the variance of heap sort at that size. However, remember, quicksort’s mean was about half the size of heapsorts at this size. So it’s hard to say which sort is better at this size, if we excluse merge sort of course.

At size 1000, heapsort is many order of magnitudes higher than the other sorts, with merge sort having a much less variance than quicksort too.

In conclusion, merge sort demonstrated much faster performance than the other sorts, with a much faster mean and much smaller variance. It reigns supreme in my data tests. For sorted vs. reversed, I am going to conclude that comparisons take a dominant amount of CPU time compared to swaps. This is because reversed vs. sorted were relatively the same at high sizes. Because they were relatively the same, it can’t be the swapping that contributes to runtime, or else the reverse array tests would yield a much slower runtime than the sorted array tests since they do much more swapping.

For variances, heapsort is unreliable at all sizes, quicksort is relatively unreliable compared to Mergesort, yet merge sort is many orders of magnitudes more reliable than the other sorts.

Finally, there could have been a few sources of error. Experimental error could have been caused by inconsistent CPU scheduling. I controlled for as much error as I could by making my methods in Sorting.java synchronized. Another source of error could have been the nature I wrote each method, some sorts were recursive and some sorts weren’t.