**Time analysis for sorting algorithms**

The time analysis for four sorting algorithms was performed. The algorithms were: Quicksort, Randomized Quicksort, Mergesort, and Insertion sort.

The Big-Ohh notation for these sorting algorithms is as follows:

Quicksort – O(n log(n) )

rQuicksort – O(n log(n) )

Mergesort – O(n log(n) )

Insertion – O( n^2 )

**Analysis of the real data**

Tests were conducted to document the real run times of each of the four sorting algorithms. All the tests were made by sorting an array of integers. I used the Random class that is included in standard Java libraries to generate random integers from 0 – 99. “The Random java class uses a 48-bit seed, which is modified using a linear congruential formula… The algorithms implemented by class Random use a protected utility method that on each invocation can supply up to 32 pseudo randomly generated bits.” (Oracle Documentation). Multiple tests were run with the same data and then an average was taken. In my testing, I set this value to five.

In general, the tests followed their respective Big-Ohh representations. Using the graphs to create a visual model of the time results for each algorithm, the two quick sorts performed very well. The insertion sort also performed as expected increasing the time significantly as size of N increased.

The very strange outlier my results showed was the speed at which the merge sort algorithm performed. Merge sort has a Big-Ohh notation of (n log(n) ). This means that is should perform the same as the other two quick sort algorithms. After I noticed this trend, I investigated the values that the mergesort was returning. For most of the runs the times were normal and close to the quick sort times. Around one run out of the five would become an outlier for that testing sequence. The timing for this outlier could upwards of x3 the time of the other runs for the same data. After this was noticed, I made sure to close any programs that was running in the background that could impact the tests. The issue with this thought is that any program impacting the merge sort should also impact the other sorts at some time. This issue was never the seen. None of the other sorts had outliers compared to the merge sort.

I chose to run the tests of these sorting methods on four different length ranges. The four ranges cover a large enough spectrum of lengths to see the benefits and costs of the functions. In the early runs, the length of N was small (10 – 100). This is where the insertion sort can shine with its simplicity. The other three sorts take a longer time with the recursive calls and movement on the stack. Towards the middle of the runs (200 – 1000), the times of the four sorts ran like each other. On the large end of the spectrum (10,000 – 100,000), The n log(n) sorts were able to show their benefits with the log (n) compared to the n ^2 that the insertion sort uses.

The timing function that I used is also included with the standard Java libraries. This function is System.nanoTime(). It returns the elapsed time from some previous start point. This function is called before and after the sort functions and their difference is the total time in used by the sorting algorithm. The documentation says that although time to the nanosecond is returned from the function, it is not guaranteed to be updated every nanosecond. The function is guaranteed to be updated at least as much as millisecond.