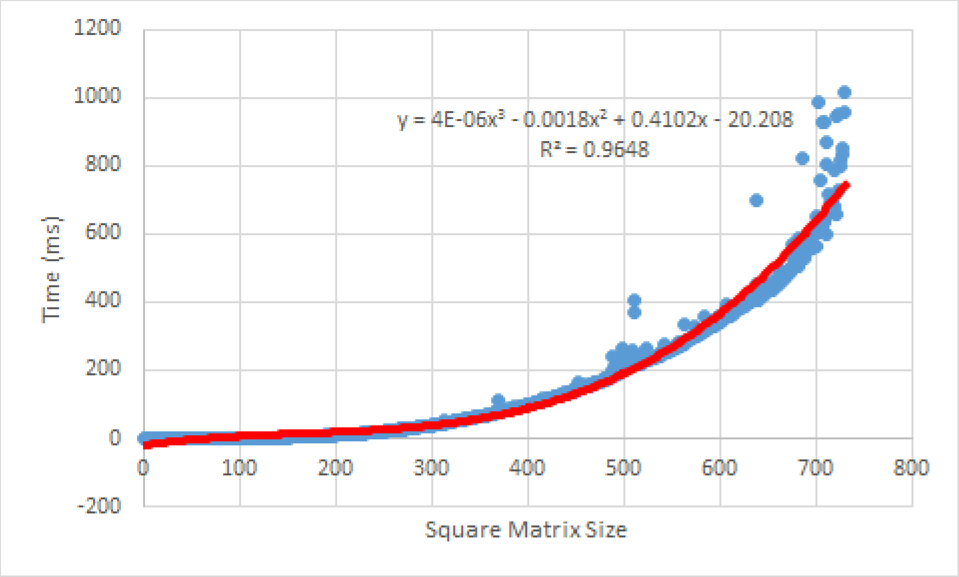
Assignment 3

COMS 311 | Matt McKillip | 2/27/15

Part 1: MatrixOperations



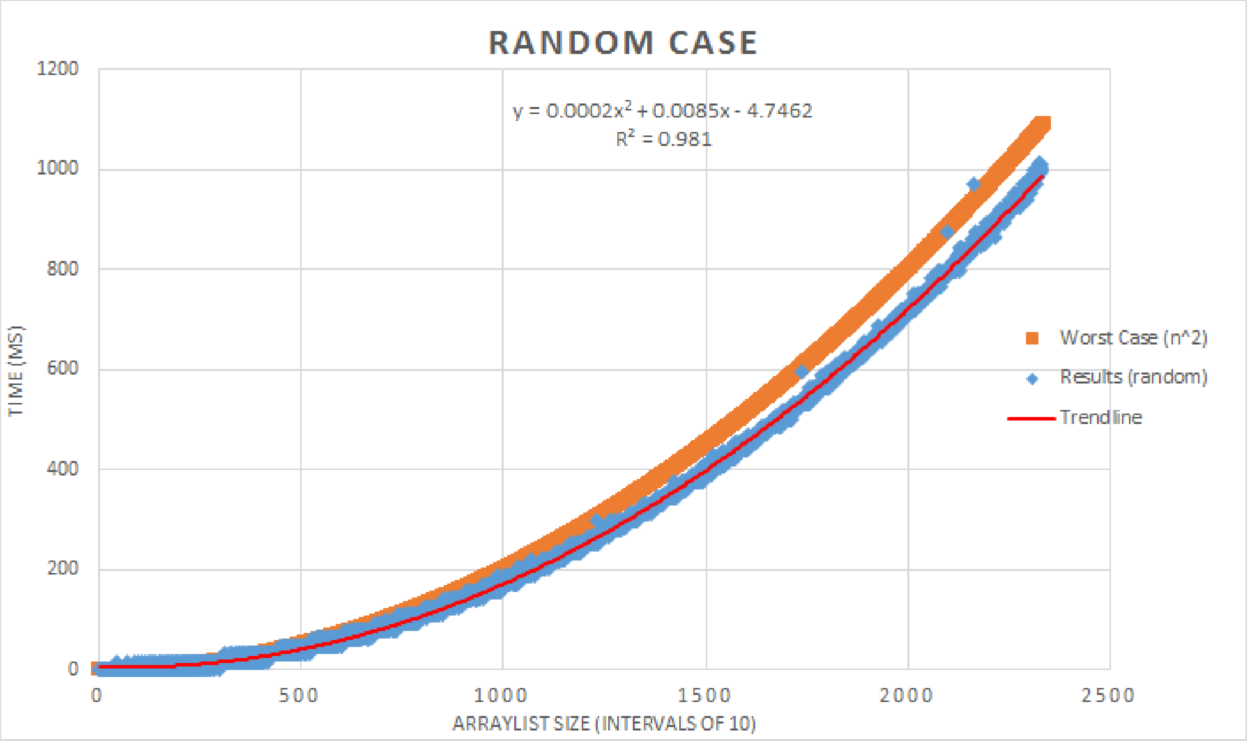
The data from the main method confirms the runtime of O(N^3) when fitted with the equation $y = .000004x^3 - .0018x^s + .4102x - 20.208$. This result is not surprising since the algorithm used to multiply the matrices was the standard algorithm described in school. It should be noted that there are some outliers in this data specifically around size 500, 650, and 700. When I look at the data, the time varies from point to point. This is from the inaccuracies of using System.currentTimeMillis() as pointed out in the problem description.

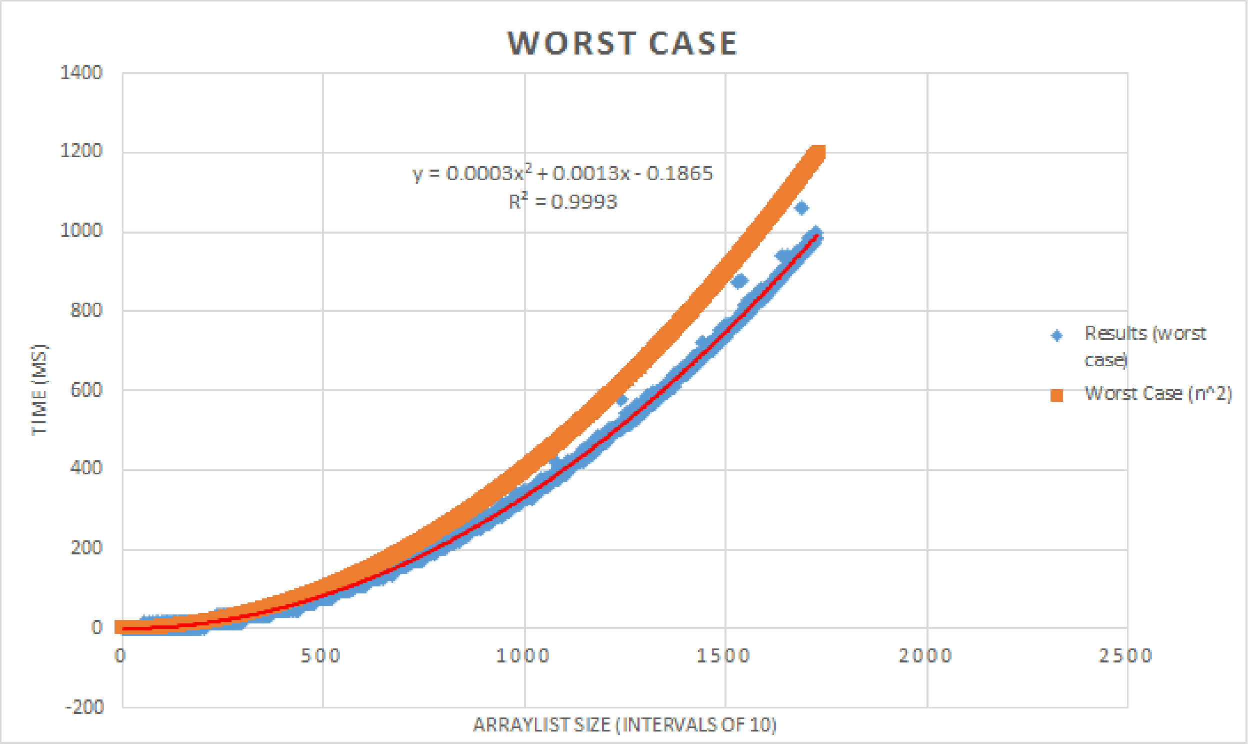
Part 2: InsertionSort

For the insertion sort algorithm, the worst case arrangement of elements is sorting them in descending order. With this arrangement, insertion sort will have to “insert” every element to the beginning of the array.

For the worst case arrangement, an array of size 17,295 elements took 1000 ms to sort. For random data an array of size 23,256 elements took 1000 ms. This large difference in speed is because, every single element in a descending order array needs to be swapped, while in a random array, that is not the case.

Below are the plots and trend lines for worst case, and random case. Due to Excels column limit I chose to only record time for every 10 sizes. I also compared both graphs with the plot of c*n^2 where c = .0002 for random case and c =.0004 for worst case,





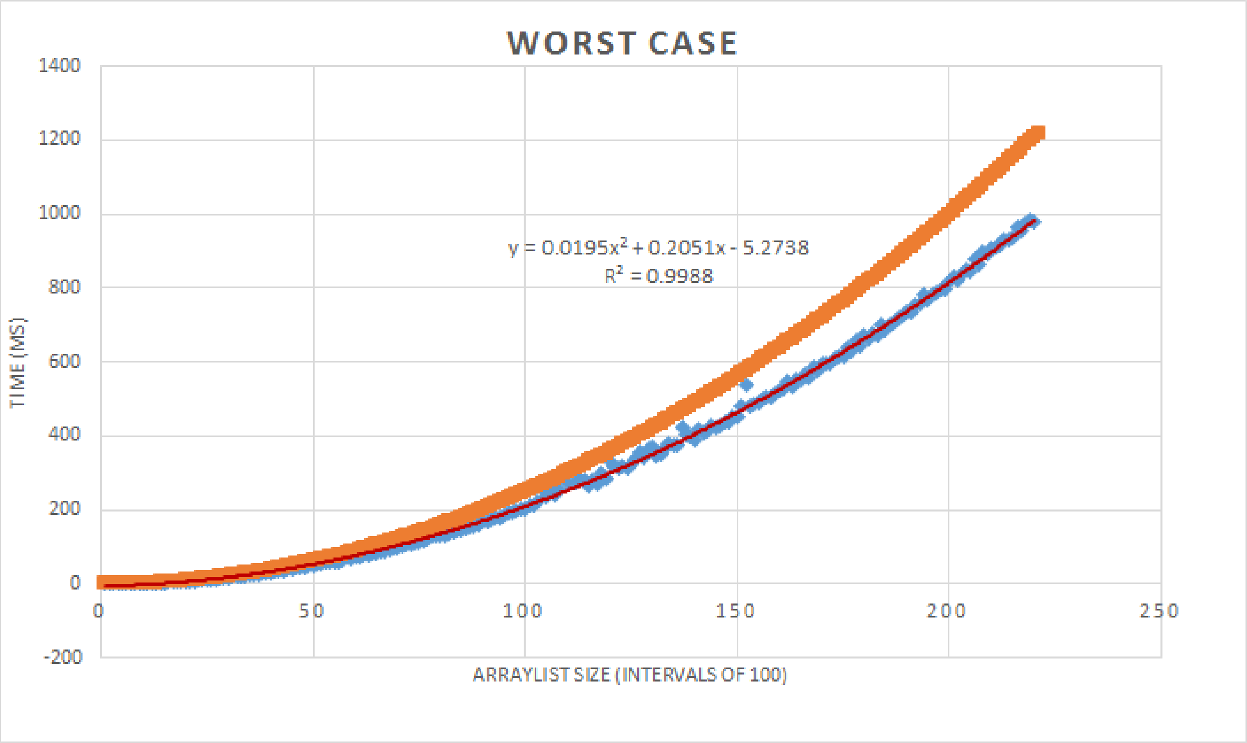
Visually it is clear that the worst case scenario is quite a bit slower than the random data. The coefficient of the x^2 confirms this. Worst case = .0003 and random case = .0002.

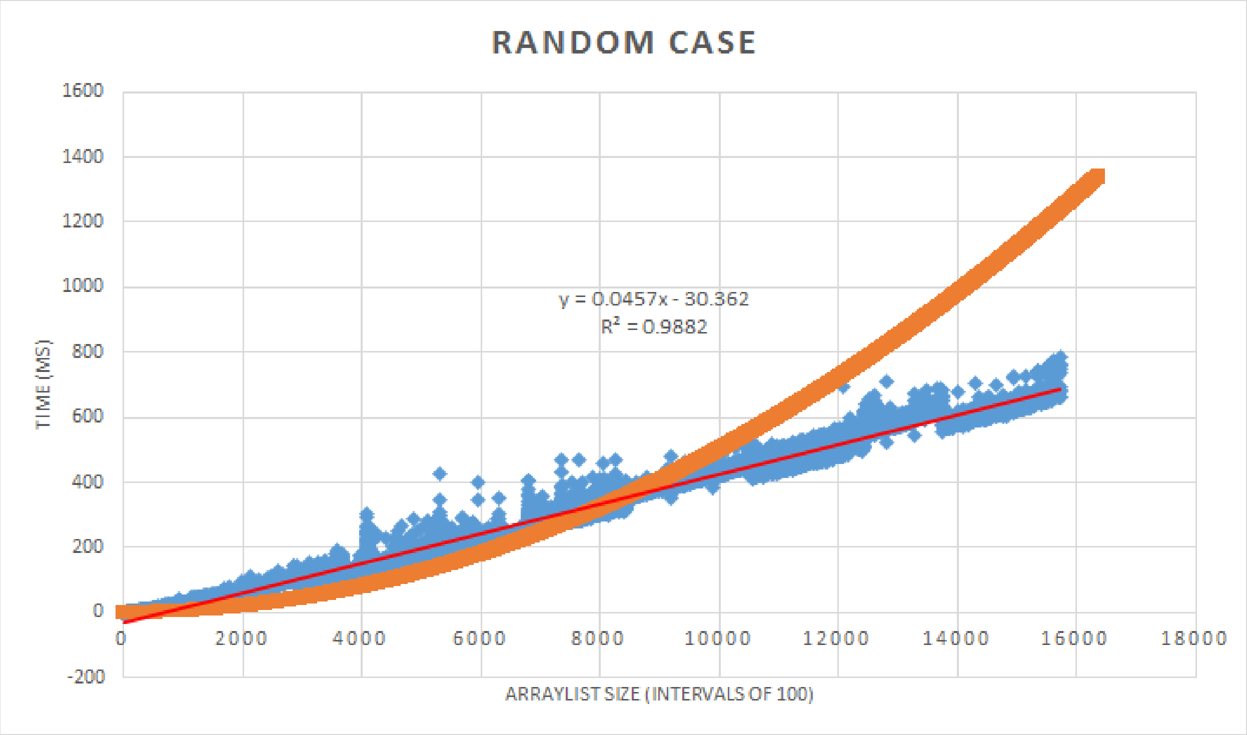
Part 3: QuickSort

For the quick sort algorithm, the worst case arrangement of elements is sorting them in descending order, much like the insertion sort. My implementation uses a left partition, which makes this case worse than if my implementation used a random or median partition. I chose to left partition due to the iterative and in place requirement.

For the worst case arrangement, an array of size 22,011 elements took 1000 ms to sort. For random data an array of size 1,572,862 elements took 1000 ms. This a very large difference in speed caused by choosing a left partition in a descending sorted array.

Below are the plots and trend lines for worst case, and random case. Due to Excels column limit I chose to only record time for every 100 sizes. I also compared both graphs with the plot of c*N^2, with c=.025 for worst case and c =.000005 for random case.



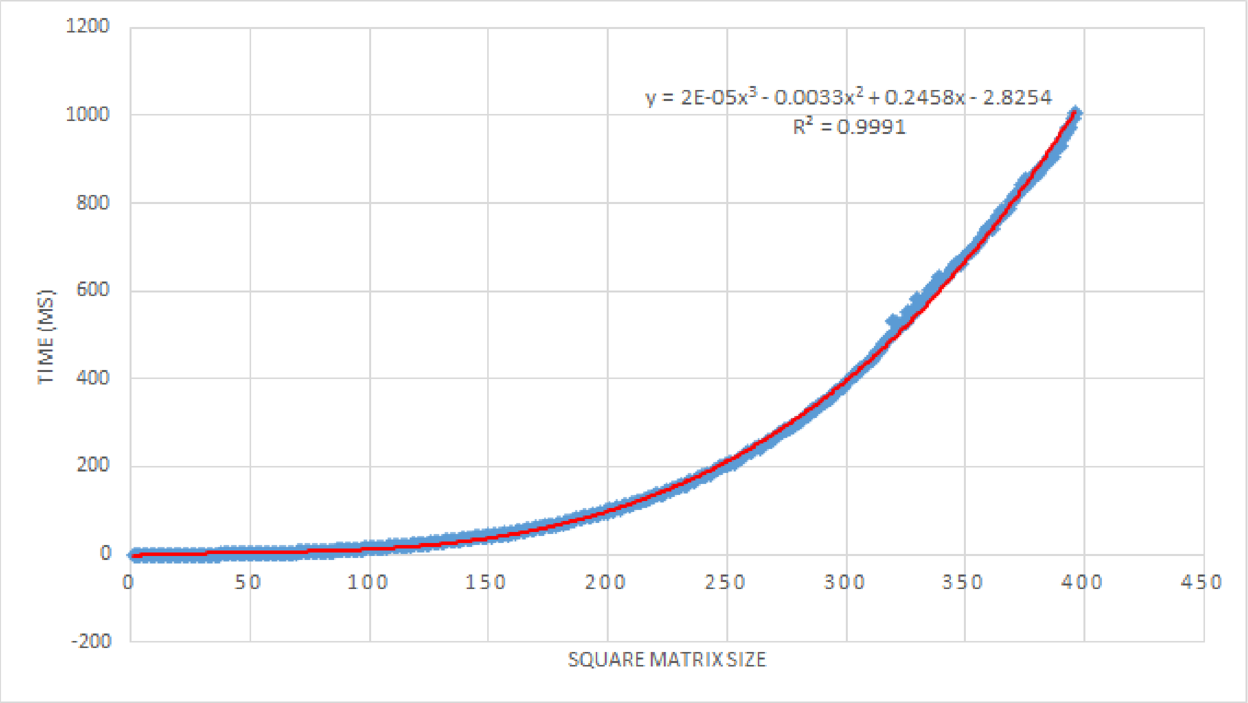


My results were as expected, with the random case being considerably faster. For my worst case I saw O(n^2) behavior and a trendline of the power 2. For the random case I saw what appears to be O(nlogn) behavior, but excel would not allow me to fit that trendline, so I fit linear which still has a good correlation.

It should be noted that the random case graph does not include all of the data points because of excels column limit.

Part 4: analyzeInverse

The implementation for finding inverse of a matrix with the Gauss-Jordan algorithm is O(n^3) runtime. My results and the order 3 trendline are below:



The trendline very strongly correlates with an R^2 value of 0.9991. These results show the N^3 behavior of this algorithm. The data has no outliers and a very correlation.