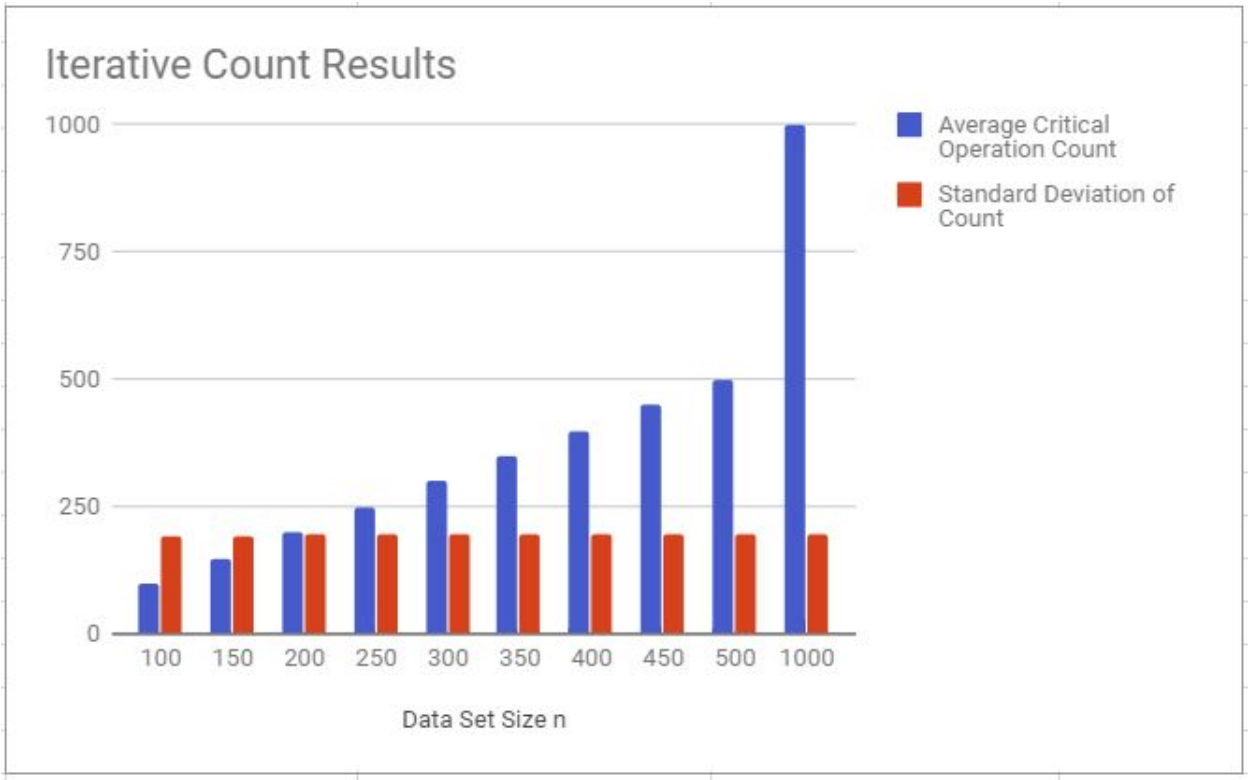
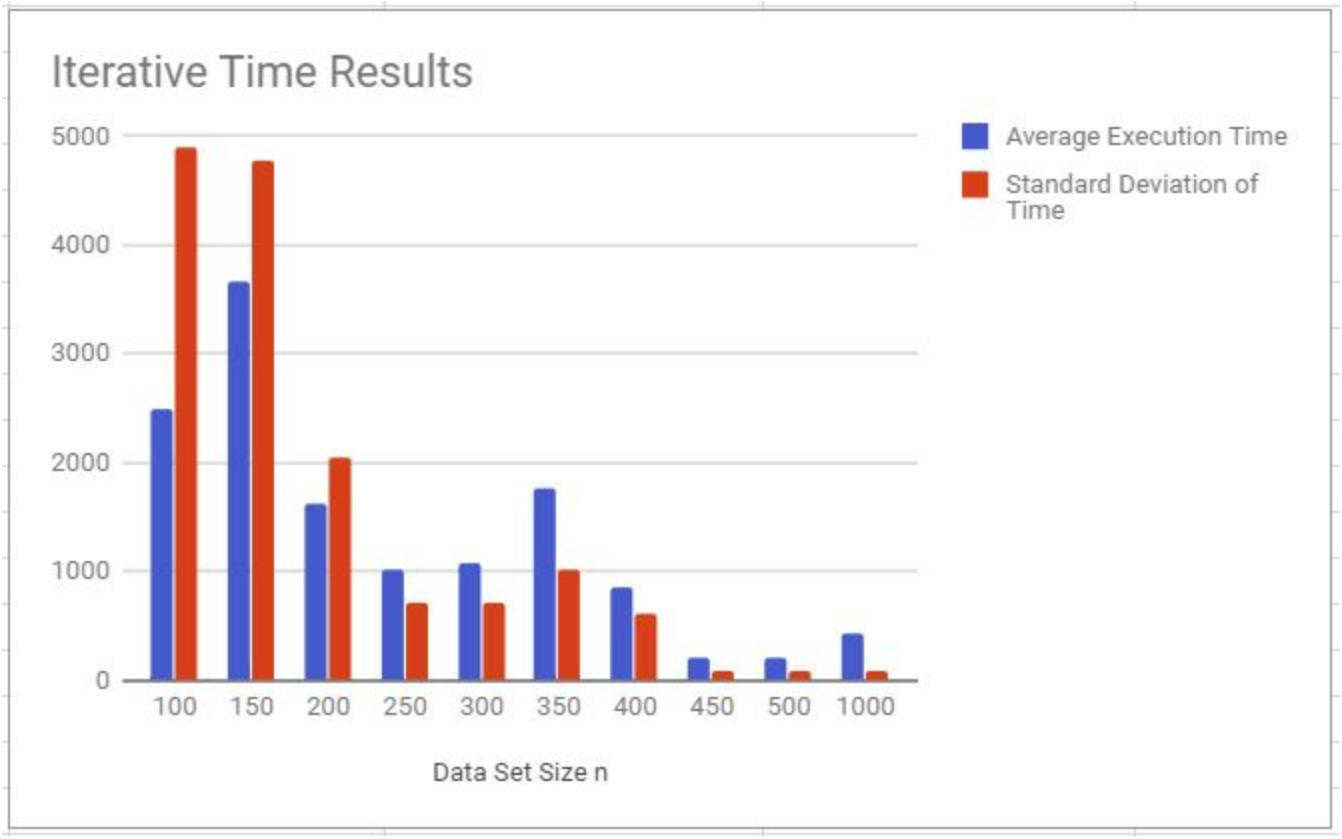
Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. The insertion sort however provides several advantages including it is simple to implement, is very efficient on small data sets, and it is in most cases more efficient than other simple quadratic sorts like selection and bubble sorts. It is very efficient on nearly sorted lists as it is O(n) on an already sorted list, and the less sorted it is the closer it gets to the O(n^2) of normal quadratic sorts. This sort is very stable as well as it does not change the relative order of elements with equal keys plus it can sort a list as it receives it so is a great choice for a list that is constantly changing, but still needing to be sorted consistently. The recursive method of sorting in most cases with the insertion sort is not as effective as the iterative version although it does look more elegant in the writing of the code. This is because each recursive call reserves additional space on the stack, meaning that the recursive solution takes O(n) space, whereas the iterative one O(1).

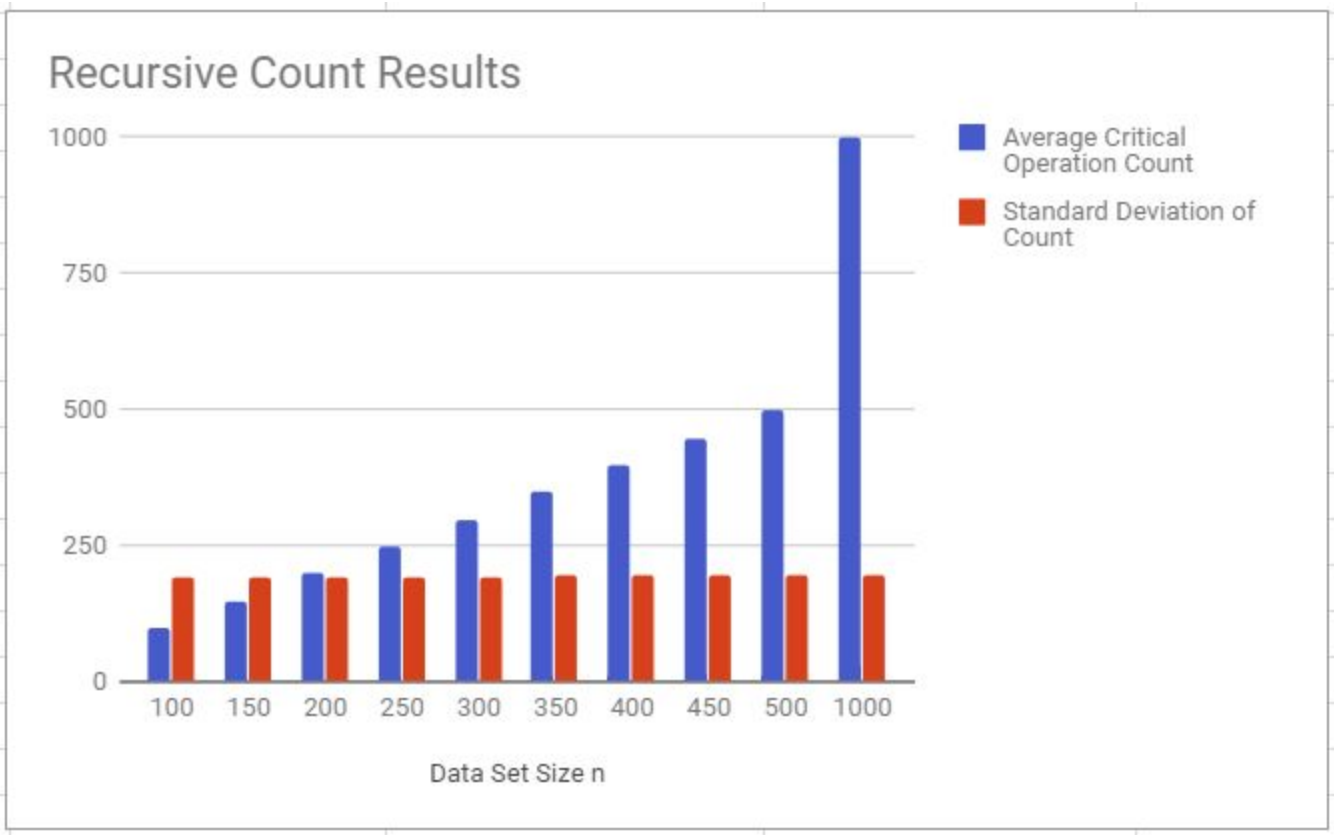
In project 1, in the recursive method for the insertion sort I placed the counter in the main while loop which ended up causing problems I did not recognize until after the project was graded. These problems included very large counts that did not make a lot of sense. Moving the counter to underneath where the recursive method calls itself, in the if statement on line 39 of Sort.java caused a more 'normal' looking count when reflected in the Average Critical Operation Count. This causes the count to be 1 less than that of its iterative counterpart where the counter was placed under the main while loop within its method.

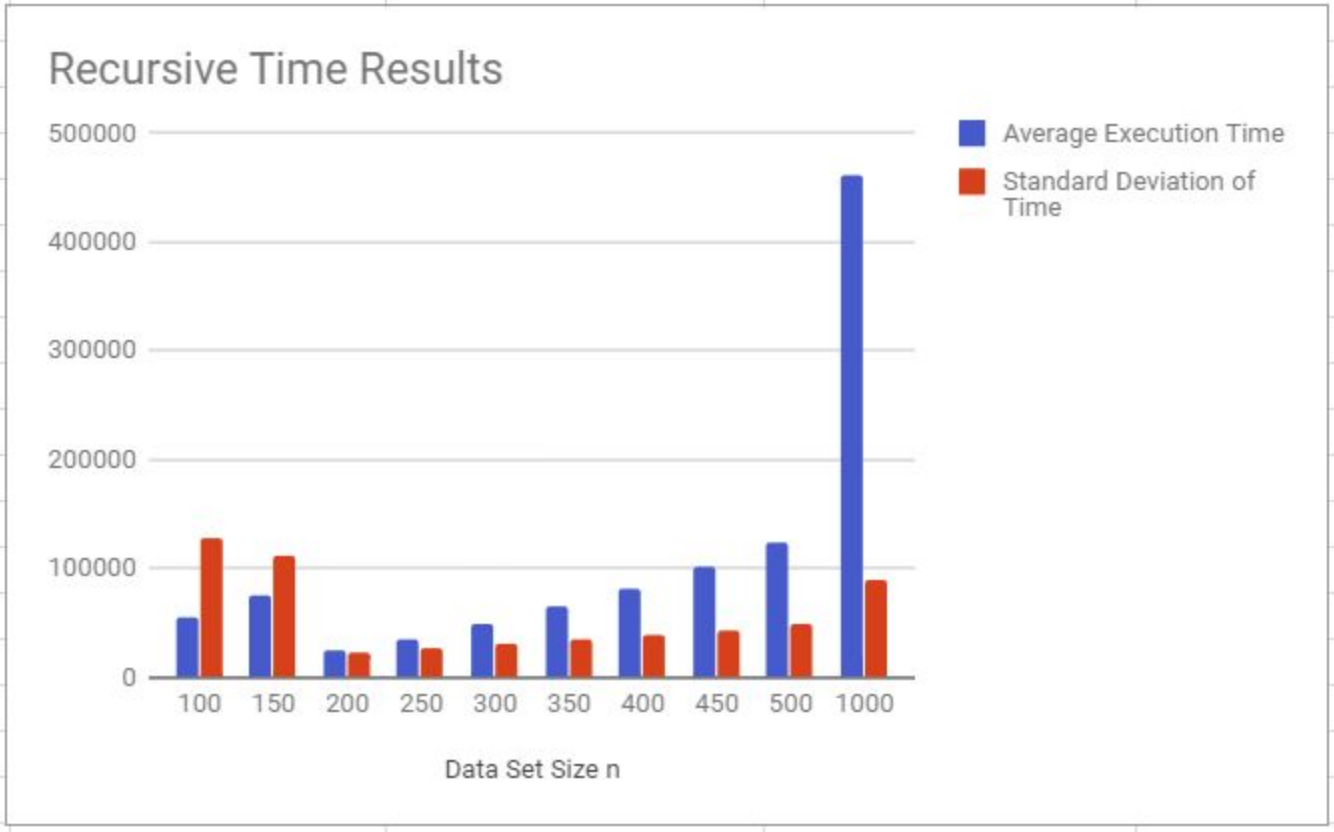
The Big Θ for both the iterative and recursive methods of the insertion sort are in most cases O(n^2) which is standard for quadratic sorts like this. The insertion sort though is at its best when a list is close to already being sorted or already sorted as if it is already sorted it is O(n) and if it is nearly sorted, then it will be closer to O(n) than O(n^2). Keeping in mind that the insertion sort is effective for lists that are constantly taking on new values and needing to be resorted, it is the perfect choice when implementing a sort that requires this kind of need.

Graphs of Results:






Recursive Count Results
1000
750
500
250
0
100
150
200
250
300
350
400
450
500
1000
Data Set Size n
Average Critical Operation Count
Standard Deviation of Count


Recursive Time Results
500000
400000
300000
200000
100000
0
100
150
200
250
300
350
400
450
500
1000
Data Set Size n
Average Execution Time
Standard Deviation of Time

Looking at the iterative results I notice that the standard deviation of count is consistent even as the average critical operation count increases due to the size of the data set. Interestingly the time results of the iterative method of this sort are a bit all over the place, but do trend downwards as the data set size increases. Since the data was inserted randomly and the insertion sort performs better on lists that are nearly sorted already, this makes me think that the random values created in this program to fill the lists that were being sorted tended to already be nearly sorted as the data sets increased in size. The standard deviation of count was consistent in the recursive call as well, which makes sense, since we used the same inputs for the data set sizes. The average critical operation count also increased as the data set increased, which is a natural expectation as the data set size consistently increases. The time results in the recursive method were most interesting as it shows clear proof that the iterative method is more effective when using the insertion sort on this particular data set. The average execution time reflects the iterative calls in the smaller data sets, but then trends upwards as the data set grows. This is due to the need of space required when calling this sort recursively. The standard deviation of time reflects the upward trend as well. The standard deviation of time for these data sets seem to reflect in a similar way in both the iterative and recursive calls. This is understandable as in both situations the Big O is O(n^2) unless the list was already sorted completely in which case it would be O(1). As previously stated, the Big Θ for both the iterative and recursive calls is O(n^2) and in both cases the graphs reflect similar trends upwards or downwards indicating the previous analysis seems to be correct.

When looking at a set of data and deciding what kind of sort to use, I learned that the sort you use is very dependant on the nature of the data. This particular project (project 1) had data inserted randomly, which makes the insertion sort not the best sort to use. The insertion sort is at its best with lists that are nearly sorted already, or already sorted, but constantly changing by only a few values. A good example would be any situation where a list had to change by a value or two at a time before being resorted. The insertion sort is at its best with non random data sets would be my main conclusion.