# Performance Comparison of Iterative and Recursive Insertion Sort

Thomas Lee

## Introduction

This paper compares the performance of recursive and iterative versions of the insertion sort sorting algorithm using the Java programming language. This algorithm can take an array of integers and sort them in ascending order.

This next section contains high level pseudo code detailing each version of the algorithm followed by a Big O analysis that explains how the algorithm works as well as its projected performance.

The last portion of this paper compares the actual execution time of each version of the algorithm when sorting integer arrays of varying sizes.

## Insertion Sort Pseudo Code Iterative Version

```
Insertion Sort Iterative Function (integer input array){

    For (the first element in the array to the length of input array){

    Get that element's index and its associated value, this is the value to sort

        While (the index is greater than zero(we still have values to compare) AND the value to
    sort < the value to its left within the array){
            Set the value to sort inside the array = value to its left within the array
            Decrement the index
      }//the while loop ends when the value to sort is ready to be inserted (the value to its left is < it
or we reached the end of the array)

     Insert the value to sort into the array at the properly decremented index

    }//end For loop (this process is repeated for every element in the array starting with the first
element)


}//end function
```

## Insertion Sort Pseudo Code Recursive Version

```
Insertion Sort Recursive Function (integer input array, array length){


    If (array length is less than or equal to 1){
        Return;
}//end if

        Insertion Sort Recursive Function (integer input array, array length - 1)

                While (the index is greater than zero (we still have values to compare) AND the value to
        mover's left  > the value to sort){
                        Set the value to sort inside the array = value to its left within the array
                        Decrement the index
        }//end while

                Insert the value to move at its correctly decremented index

}//end method
```

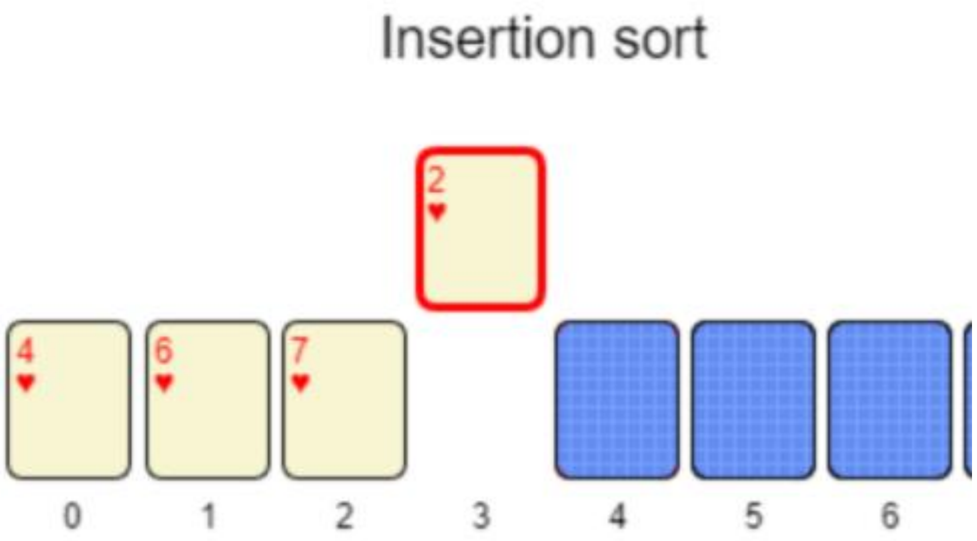
## Iterative Big O analysis

Insertion sort uses two loops to sort a data set.

The outer loop iterates over the data set and the inner loop swaps the values.

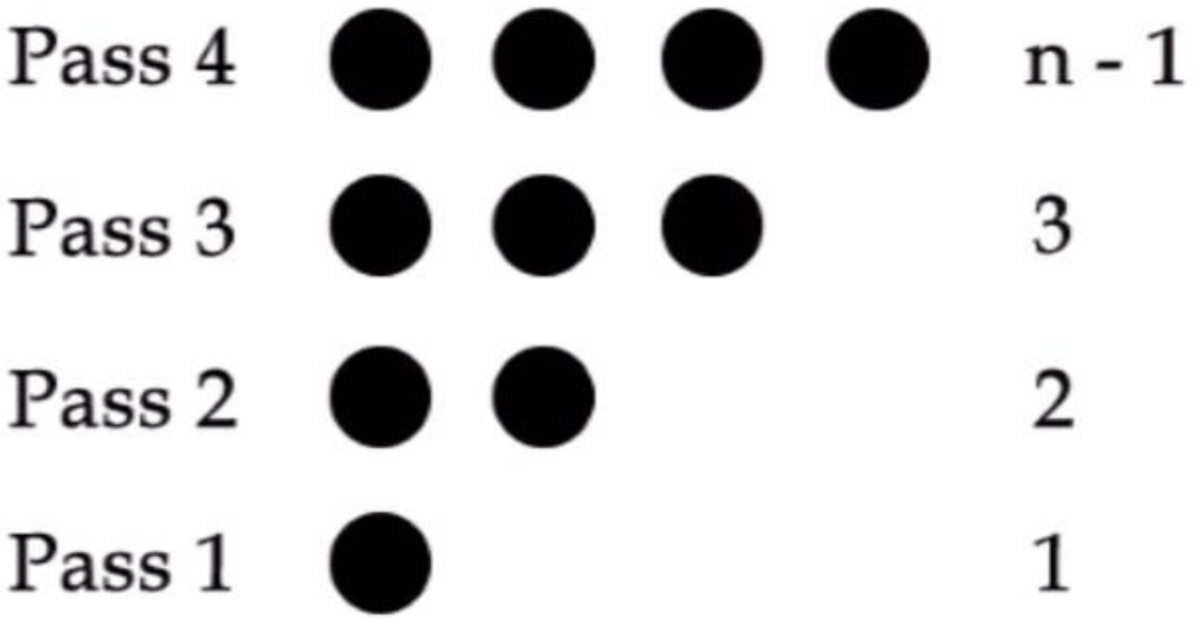
This is achieved by simply doing comparisons on every value to the left of each focus number.

The focus number here is 2. We compare 2 to every number to its left one by one until 2 is in order. In this case we would put the 2 all the way to left since it's the smallest number.



Let us imagine an array of 5 elements, n = 5.
Let each black circle denote a comparison that we have to make to see where we need to put our focus number.

Here we illustrate the maximum amount of comparisons that can occur with each pass.

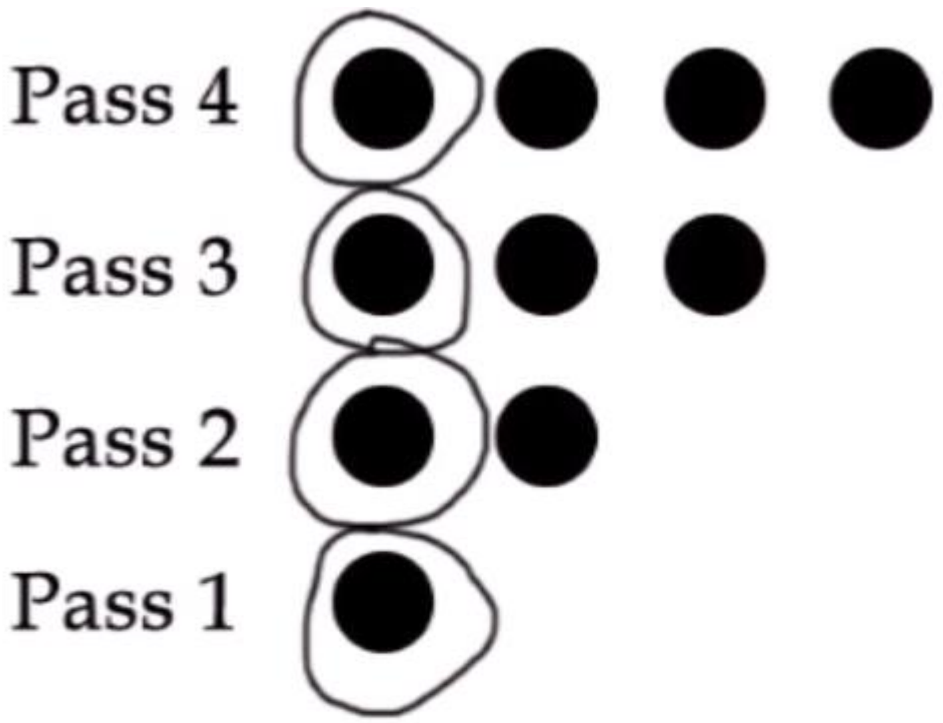


Note that the black circles form a triangle, thus we can get the worst case time by solving for the area of the triangle with formula LW/2

The length is n-1 and the width is n.

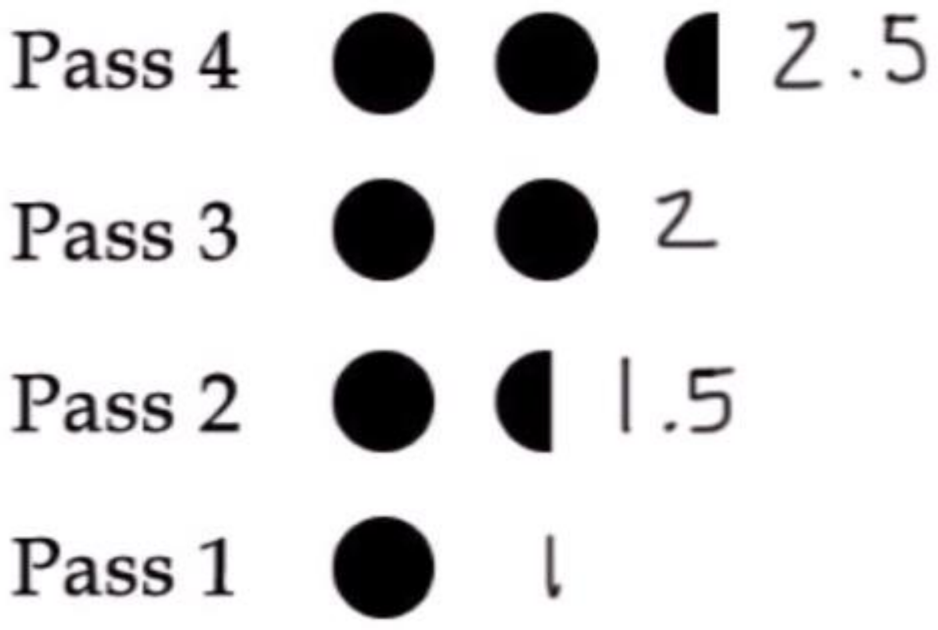
n(n-1)/2 = (n^2-n )/ 2 **thus we have n^2 time for worst case.**

In best case the list is already sorted an only one comparison must be made for each pass



Thus we have n-1 comparisons in total. Dropping everything but the dominant factor, as usual we get **n or big 0 of n for best case.**

For the average case we divide the area of the triangle by 2. Since the first pass always occurs we ignore that in the calculation.



The first column is ignored since that comparison always happens, we divide the remaining columns by two to get the total amount of comparisons for the average case written on the right. 7 comparisons total.

To put this in terms of n let us add the time for best and worst case and divide by 2 (taking the average of both cases to get the middle of the two)

$$n - 1 < \text{avg. case} < \frac{n(n-1)}{2}$$

$$\frac{2(n-1)}{2} + \frac{n(n-1)}{2}$$

$$\frac{\frac{(n-1)(n+2)}{2}}{2} \rightarrow \boxed{\frac{(n-1)(n+2)}{4}}$$

If we plug in n for 5 we should get 7 comparisons as we showed before.

$$\frac{(5-1)(5+2)}{4} \rightarrow \frac{\cancel{4} \times \textcircled{7}}{\cancel{4}}$$

Thus the average time is exactly

$$\boxed{\frac{(n-1)(n+2)}{4}}$$

**This expression contains an n^2 term, thus the average and worst case are the same.**

# Recursive Big O Analysis

**Initial condition**: the time to sort an array of 1 element is constant:

T(1) = O(1) = 1

**Recurrence relation**: `The time to call insertion sort on an array of N elements is equal to the time to sort an array of N-1 elements plus N-1 comparisons. Thus we can express the total time of the function as recurrence relation as follows:

T(N) = T(N-1) + N-1

Next we perform telescoping. *This is a technique used to solve recurrence relations by summing up the left and right hand side of a series of equations. We sum the left hand side and right hand side of the equation separately. The equations we are going to sum are all the same recurrence relation we intend to solve but written in a way to cancel out several terms.*

Re-writing the recurrence relation for N-1, N-2, …, 2

**T(N) = T(N-1) + N-1**

We can rewrite the recurrence relation if we consider the three previous terms in the sequence, namely all terms with N become the previous term N-1.

**T(N-1) = T(N-2) + N-2**

We repeat the process.

**T(N-2) = T(N-3) + N-3**

Now we can write an equation without N, since we know the left hand side is always T(N) and the right hand side is always T(N-1) + N-1. We can replace N with 2.

**T(2) = T(1) + 1**

Next we sum up the left and the right sides of the **bolded** equations above:

T(N) + T(N-1) + T(N-2) + T(N-3) + …. T(3) + T(2) **=** T(N-1) + T(N-2) + T(N-3) + …. T(3) + T(2) + T(1) + (N-1) + (N-2) + (N-3) + …. +3 + 2 + 1

Notice that we can now subtract the blue series from the red series

T(N) **=** T(1)+ (N-1) + (N-2) + (N-3) + …. +3 + 2 + 1

On the right hand side we notice we almost have the sum from 1 to N which is

$$\sum_{q=1}^{p} q = \frac{p(p+1)}{2} .$$

But we are missing the Nth term so we can subtract that from this summation to get

N(N+1)/2 – N

Which simplifies to

N(N-1)/2, plug that back into our equation to get

T(N) = T(1) + N(N-1)/2

Note that T(1)=1

T(N) = 1 + N(N-1)/2

Therefore, the running time of insertion sort is:

T(N) = O(1 + N(N - 1)/2) = O(N^2)

Notice that we have a N^2 on the right hand side after distributing N, thus the running time is O(N^2) .

---

## Warming up the JVM

Classes must be loaded into memory when a JVM process starts after that is completed, certain important classes are pushed into the JVM cache. Other classes are loaded only on request. To avoid the program execution time being affected by class loading times, we can cache all classes beforehand so they are available instantly during run time.

We will create a large amount of Dummy classes to warm up the JVM.

```
public class Dummy {
   public void m() {
   }
}
```

This method, *load*, will be called on during the start of the program, to create 100000 dummy classes.

```
public class ManualClassLoader {

    protected static void load() {
        for (int i = 0; i < 100000; i++) {
            Dummy dummy = new Dummy();
            dummy.m();
        }
    }
}
```

Finally, we make a call to the load method within the main function before the execution of any other code.

```
public static void main(String[] args) {



    ManualClassLoader.load();
```

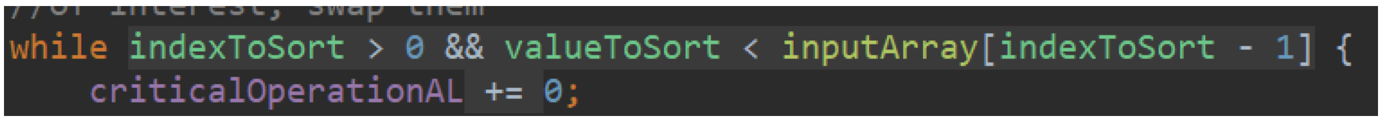
---

## Methods of counting the amount critical operations for each algorithm

**Iterative Critical Operations**

The iterative critical operation count is located within the while loop nested inside the for loop.

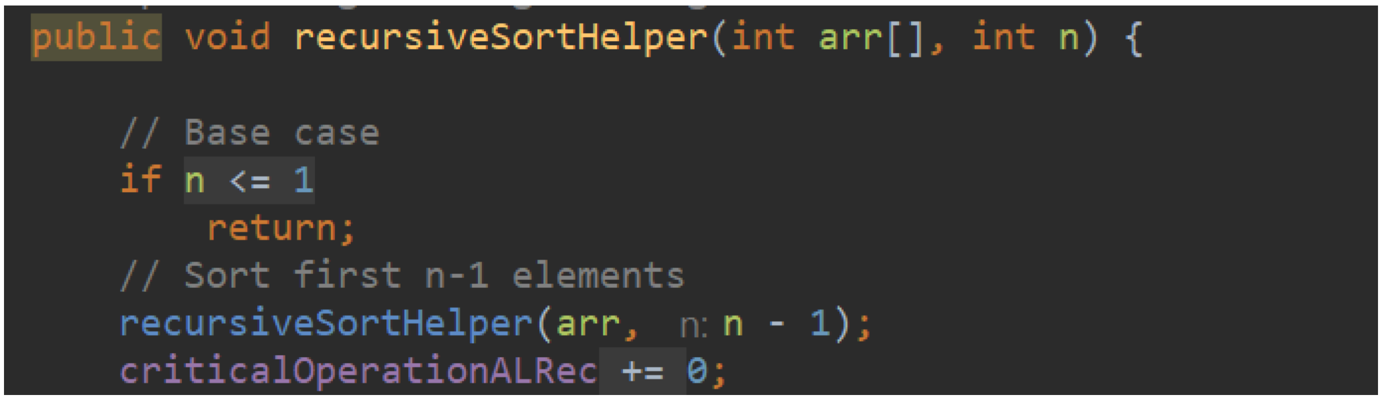
I picked this position within the code because the operations within the while loop contain the part of the algorithm that leads to the shift of the unsorted elements in the array. The while loop also contains code that is executed a certain number of times that changes based on how sorted the incoming array is.



```
//of interest, swap them
while indexToSort > 0 && valueToSort < inputArray[indexToSort - 1] {
    criticalOperationAL += 0;
```

**Recursive Critical Operations**

The recursive critical operation count is located after the recursive call.



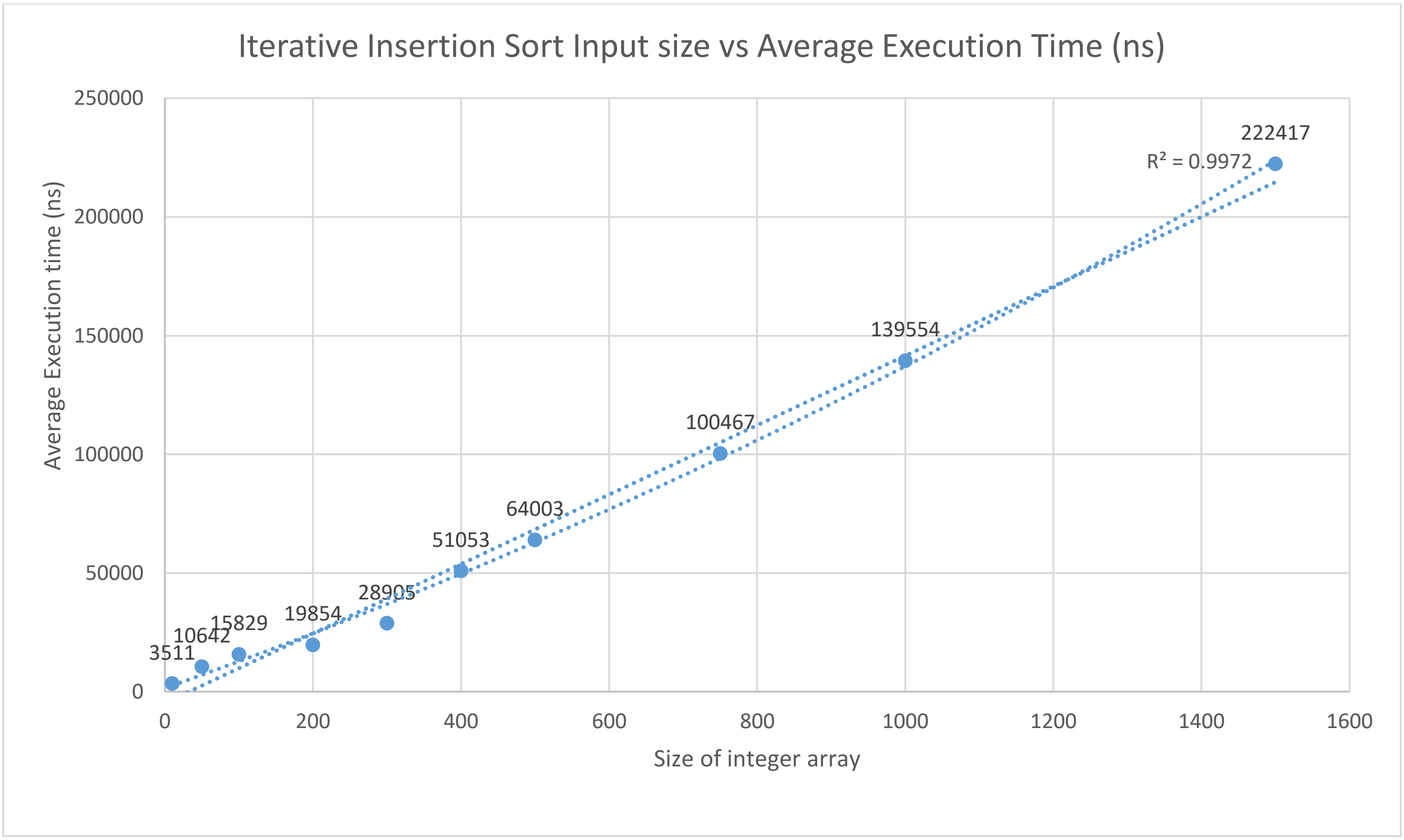
```
public void recursiveSortHelper(int arr[], int n) {

    // Base case
    if n <= 1
        return;
    // Sort first n-1 elements
    recursiveSortHelper(arr,  n: n - 1);
    criticalOperationALRec += 0;
```

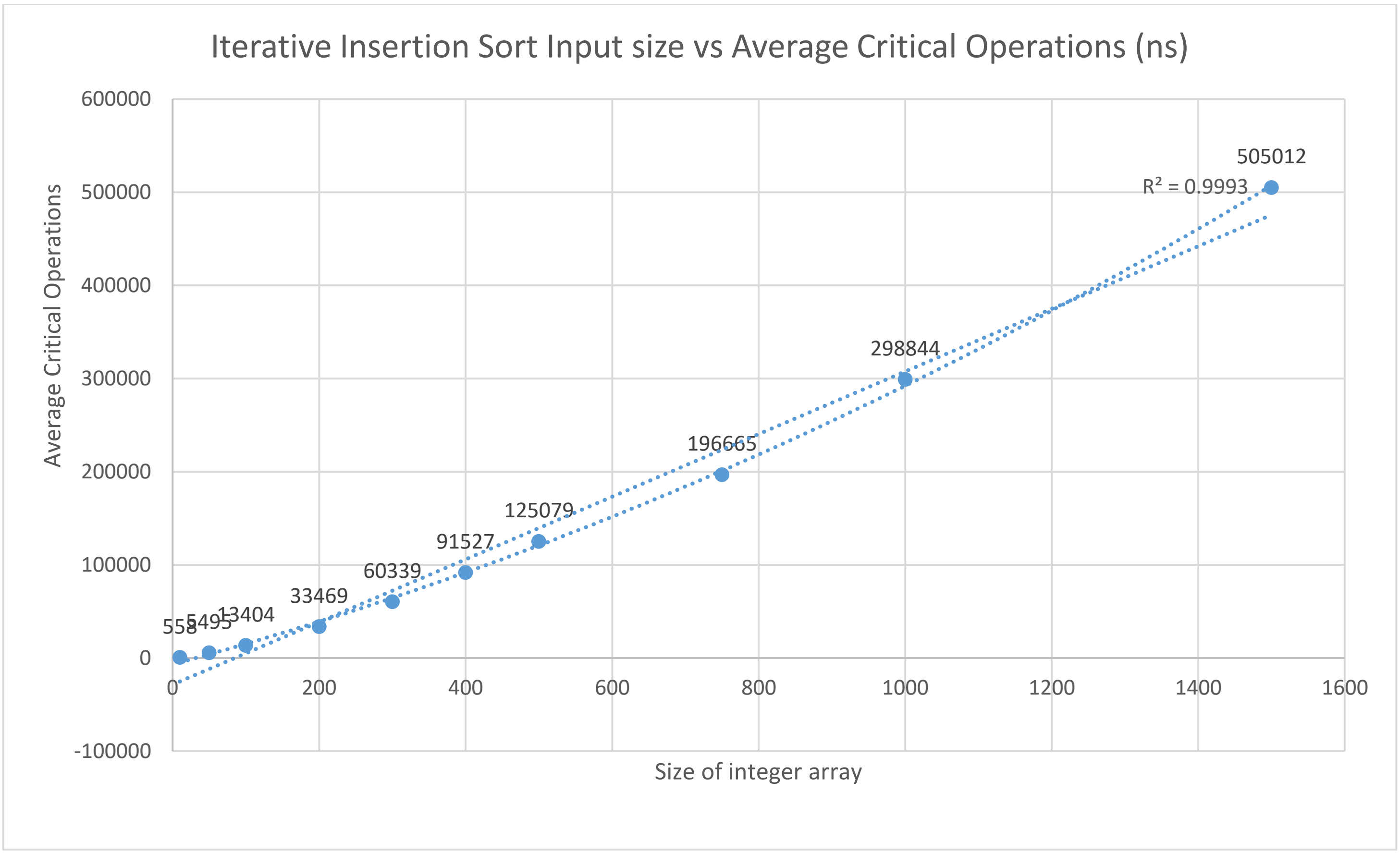
recursiveSortHelper is called within itself and we increment the recursive operations count after it is called.

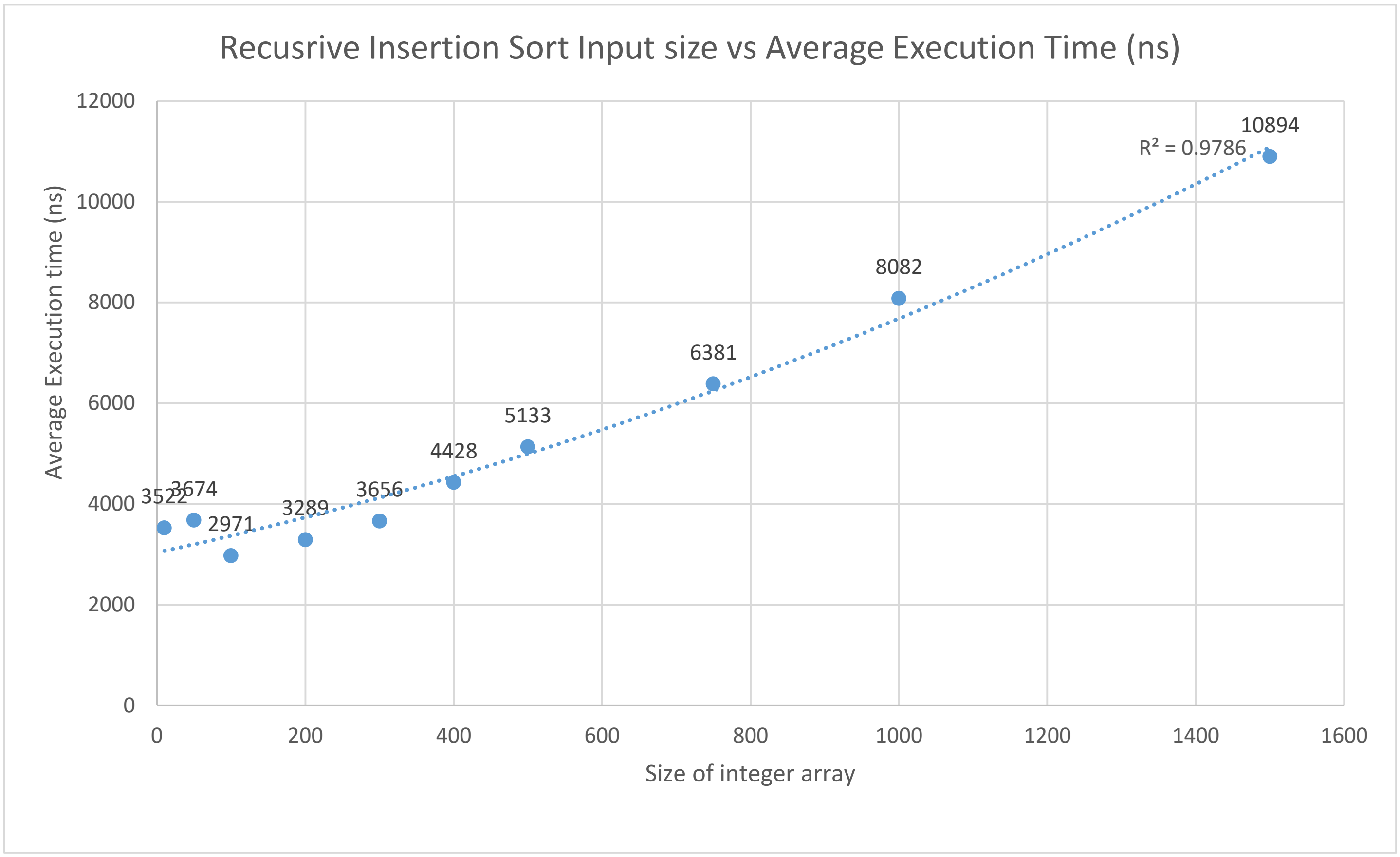
The critical operation within a recursive function is the amount of times the function is called upon; thus we keep track of the number of function calls.

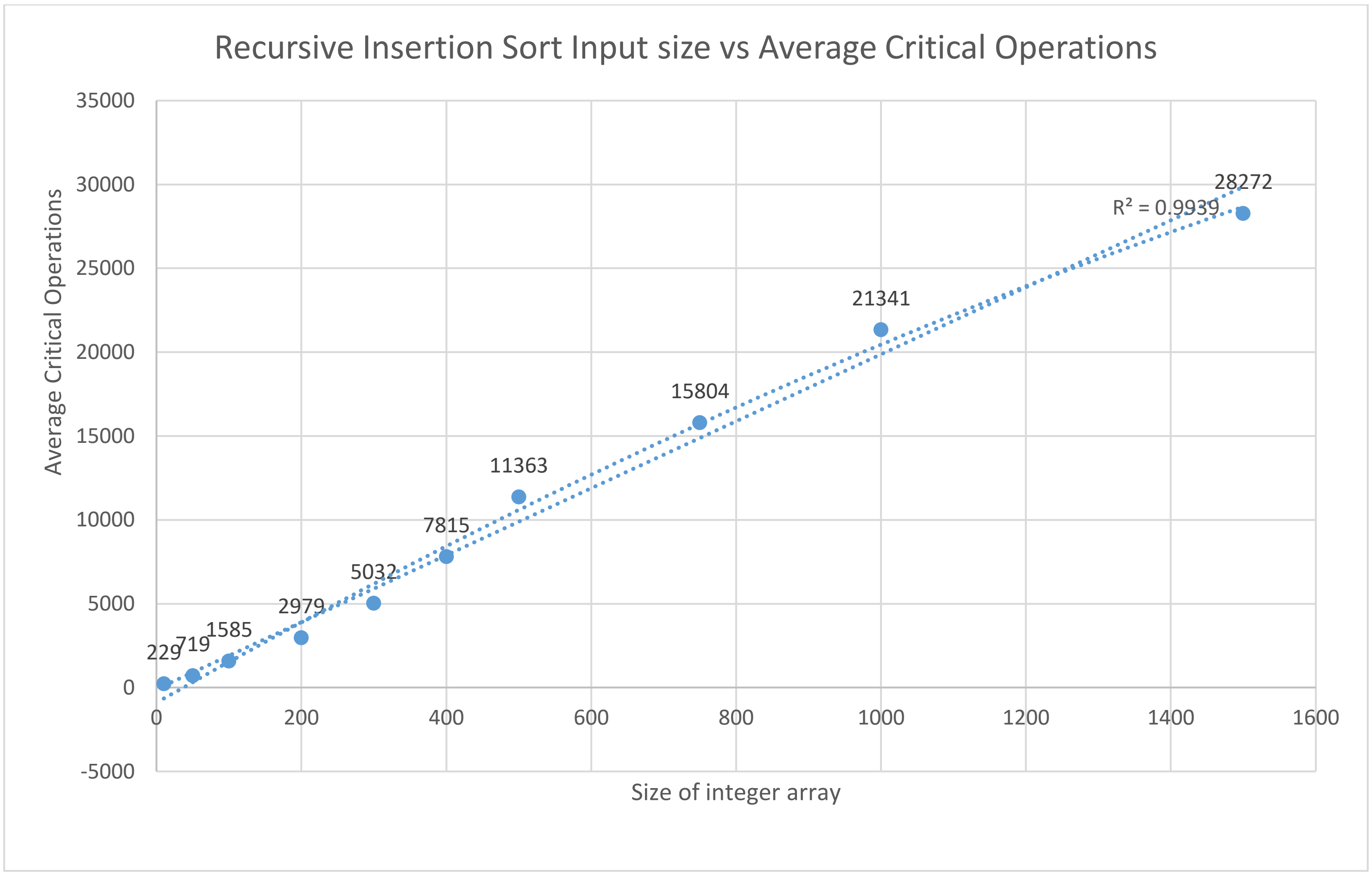
## Comparing input size to average execution time and number of critical operations

The following data comes from running insertion sort on randomly created integers arrays of size 10, 50, 100, 200, 300, 400, 500, 750, 1000 and 1500.




Iterative Insertion Sort Input size vs Average Critical Operations (ns)
Average Critical Operations
Size of integer array
600000
500000
400000
300000
200000
100000
0
-100000
0
200
400
600
800
1000
1200
1400
1600
558
5495
13404
33469
60339
91527
125079
196665
298844
505012
R² = 0.9993


Recusrive Insertion Sort Input size vs Average Execution Time (ns)
Average Execution time (ns)
Size of integer array
0
2000
4000
6000
8000
10000
12000
0
200
400
600
800
1000
1200
1400
1600
3522
3674
2971
3289
3656
4428
5133
6381
8082
10894
R² = 0.9786



In regards to the execution time of each algorithm, the recursive version outperformed the iterative version for all input sizes.

The number of critical operations for the recursive version is also less than the number of critical operations done by the iterative version for every input size n.

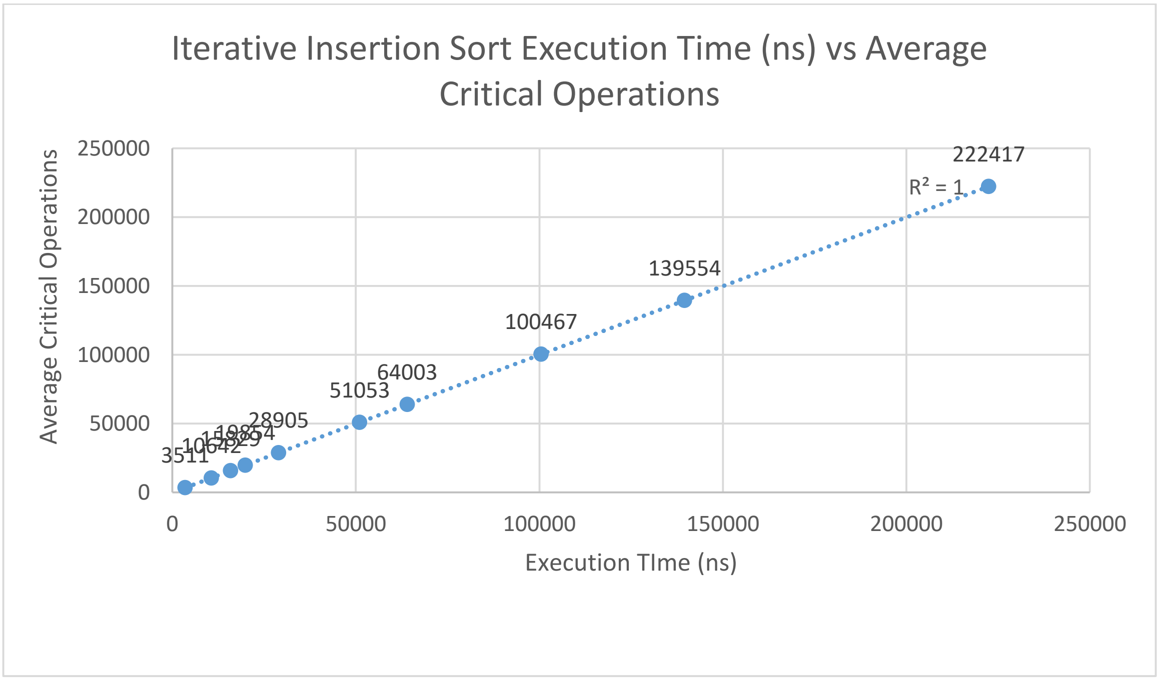
Using a second order polynomial model to fit the data, we see that our R^2 values are all above .99 with one value being .98. As R^2 values tend to .1, it indicates that the model is a great fit thus we can conclude that a second order polynomial describes these data sets very well. This is in line with our big 0 analysis of each algorithm performing in Big O of n^2 time.

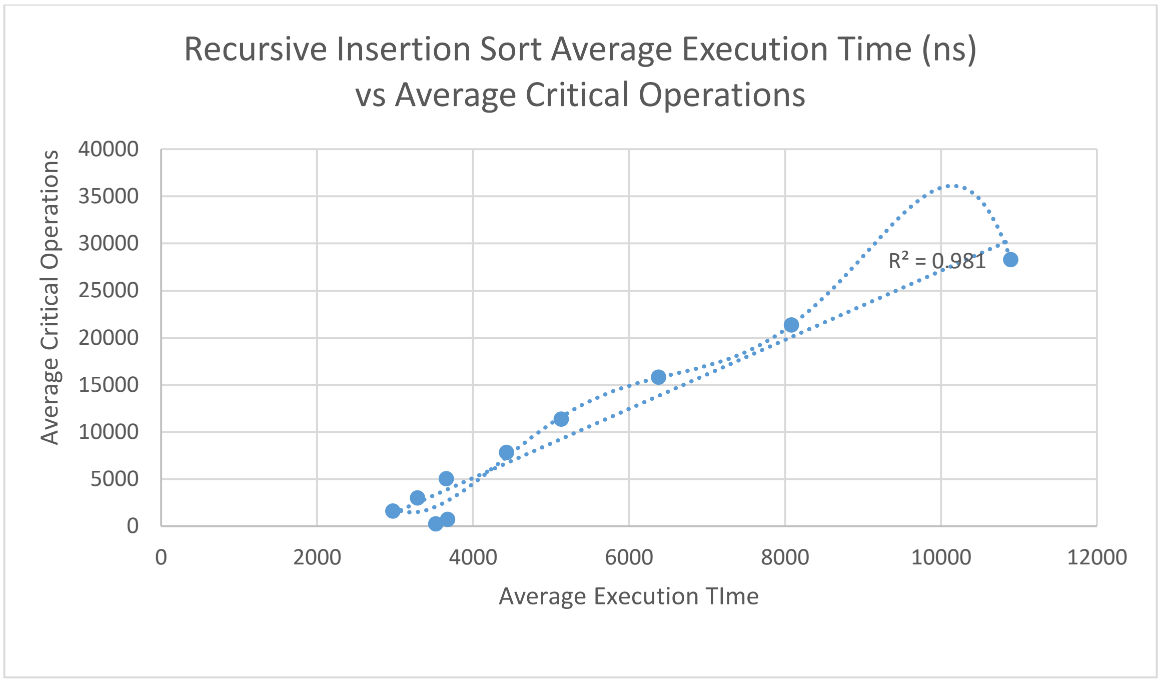
**Comparing Execution Time and Critical Operations**

If we look at each version of the algorithms execution time compared to the number of critical operations, as the number of critical operations increased, the execution time also increased.

A graph of the execution time versus the number of critical operations yields a perfect linear R^2 value for the iterative version while the recursive version has the best R^2 value of a sixth order polynomial at .98 but also had a linear R^2 value of .95.

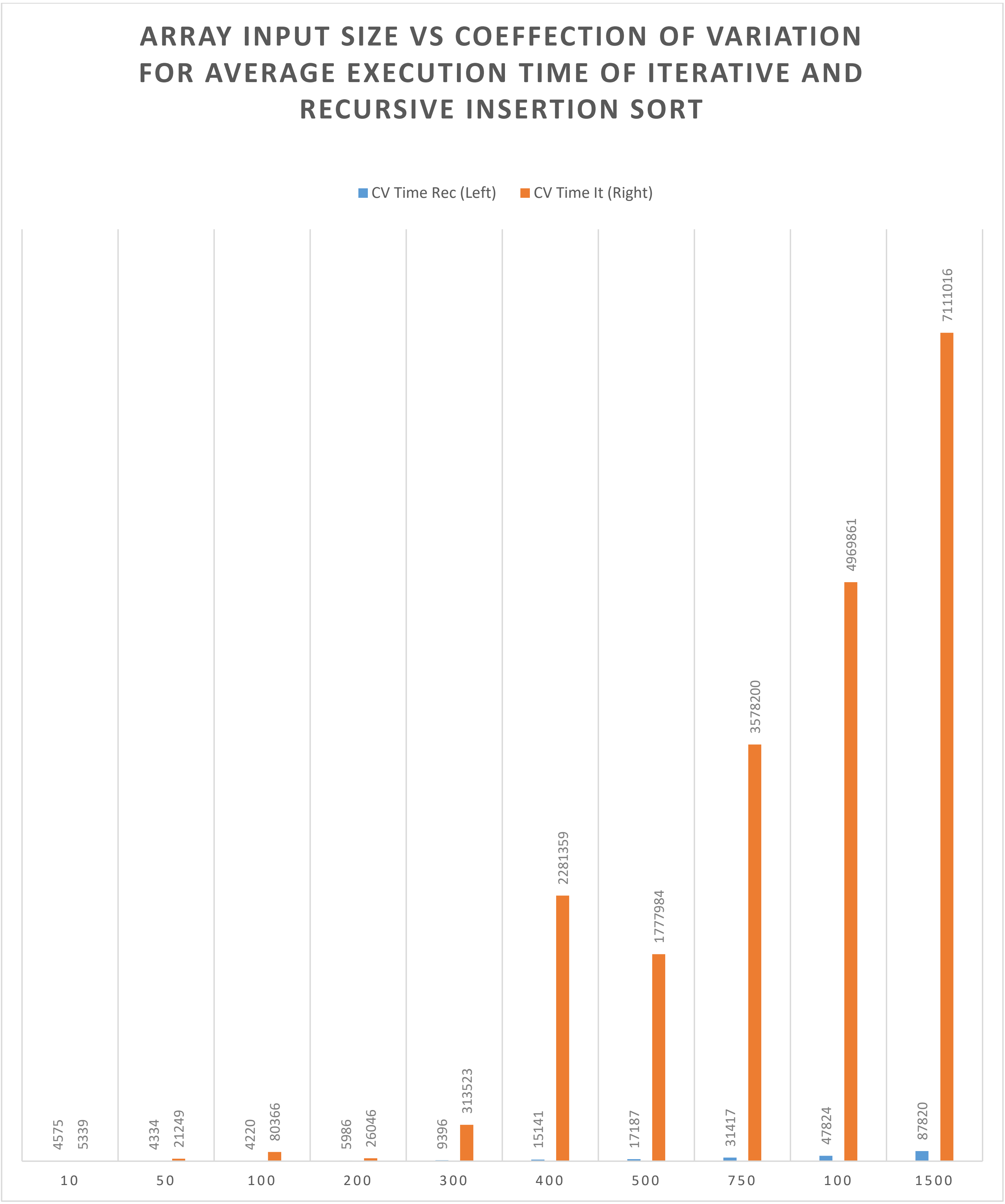
Thus, the iterative version of insertion sort follows that as the execution time becomes longer to sort a data set, the number of critical operations also increases linearly. While the recursive may seem to follow a polynomial relationship, we surmise that more data may also produce a linear relationship akin to the iterative version when comparing these two measurements.

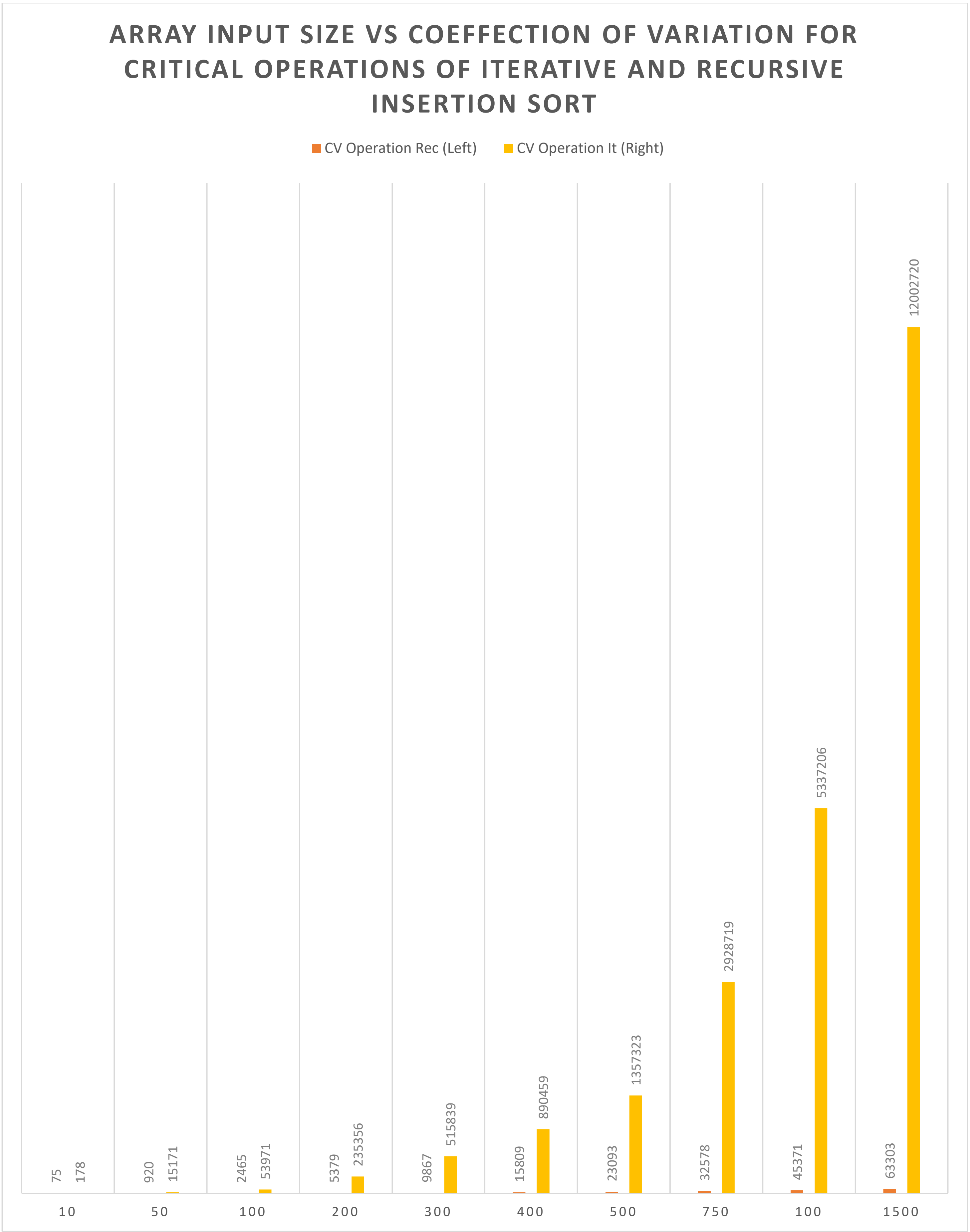




## Coefficient of Variance and Algorithm sensitivity

The coefficient of variance allows us to effectively compare the variation of two different data sets. A larger variation indicates that values are more spread out relative to the mean and unlike standard deviation, the coefficient of variation takes into account the magnitude of the data size. Thus we are effectively able to compare the two algorithms variation at the same input size.


ARRAY INPUT SIZE VS COEFFECTION OF VARIATION FOR AVERAGE EXECUTION TIME OF ITERATIVE AND RECURSIVE INSERTION SORT
CV Time Rec (Left)
CV Time It (Right)
4575
5339
4334
21249
4220
80366
5986
26046
9396
313523
15141
2281359
17187
1777984
31417
3578200
47824
4969861
87820
7111016
10
50
100
200
300
400
500
750
100
1500



ARRAY INPUT SIZE VS COEFFECTION OF VARIATION FOR CRITICAL OPERATIONS OF ITERATIVE AND RECURSIVE INSERTION SORT
CV Operation Rec (Left)
CV Operation It (Right)
75
178
920
15171
2465
53971
5379
235356
9867
515839
15809
890459
23093
1357323
32578
2928719
45371
5337206
63303
12002720
10
50
100
200
300
400
500
750
100
1500

The variation within the iterative algorithm was much greater than its recursive counterpart at every input size for both the average execution time and the average critical operations count.  Thus, comparing the algorithms side by side for a constant input size n, the iterative algorithm displays a higher degree of sensitivity as there is a larger degree of uncertainty in regards to how long it will take to execute and how many critical operations it must perform.

Comparing the change in variation as we increase the input size n, the iterative version of the algorithm experienced higher rates of increasing variation within critical operation count and for average execution time between smaller input sizes, (array sizes 10-100). Thus the iterative version has a higher sensitivity to the shifting of smaller input sizes.

For array sizes 200-1500 both of the algorithms experienced a similar magnitude of variation change between input sizes for critical operation counts, almost doubling each time for input sizes 300-1500 with the exception of the shift from input size 400-500.

There is a notable decrease in execution time variation going from input size 400 to 500 within the iterative version of the algorithm.

## Important Conclusions

Both the recursive and iterative functions have the same theoretical performance times at large input sizes, with a Big O of n^ 2.

If the array is already sorted (best case), each sorting algorithm performs in n time, else it performs in n^2 time (worst and average case).

The JVM requires adequate warming up in order to effectively evaluate algorithm performance times.

According to our findings, the average execution time of the recursive version of insertion sort outperformed the iterative version for all input sizes in addition to having to perform less critical operations at every input size.

Regression analysis on data containing input size versus average execution time or input size versus average number of critical operations yields a very good second order polynomial fit. Which is in-line with our n^2 Big 0 analysis.

Comparing the average execution time to the number of critical operations, we have found that the iterative version yields a perfect linear relationship, while the recursive version has the best R^2 value for a sixth order polynomial. However, it's R^2 value for a linear relationship was also good. More values could be tested to further differentiate these findings.

The variation within the iterative algorithm was much greater than its recursive counterpart at every input size for both the average execution time and the average critical operations count.

Comparing the algorithms side by side for a single input size n, the iterative algorithm displays a higher degree of sensitivity

The iterative version has a higher sensitivity to the shifting of smaller input sizes and changing the input values results in a greater degree of outcome variability between small input sizes.

# Works Cited

Analysis of insertion sort. (n.d.). Retrieved from https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/analysis-of-insertion-sort

CmSc250. (n.d.). Retrieved from http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250_Weiss/L14-RecRel.htm

How to Warm Up the JVM | Baeldung. (2017, June 23). Retrieved from http://www.baeldung.com/java-jvm-warmup

Insertion Sort - Algorithm Walkthrough. (2015, November 30). Retrieved from https://www.youtube.com/watch?v=6K-aC_tjLrg&t=203s

Insertion Sort | Brilliant Math & Science Wiki. (n.d.). Retrieved March 4, 2018, from https://brilliant.org/wiki/insertion/