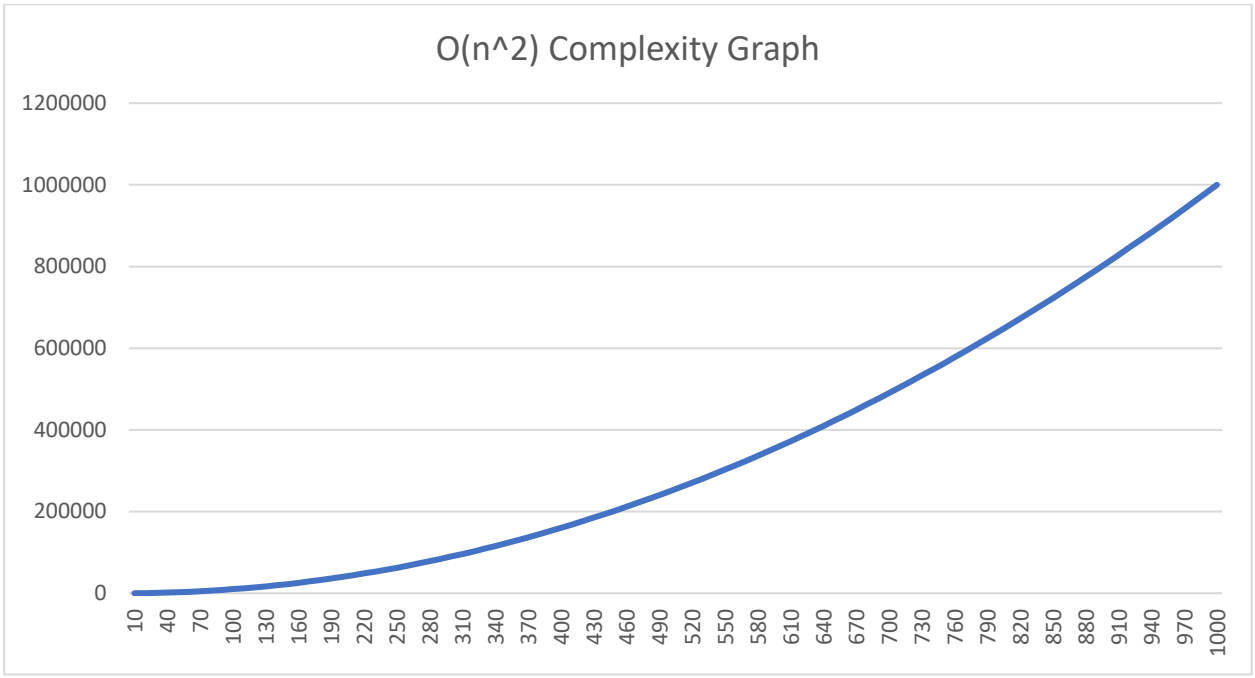
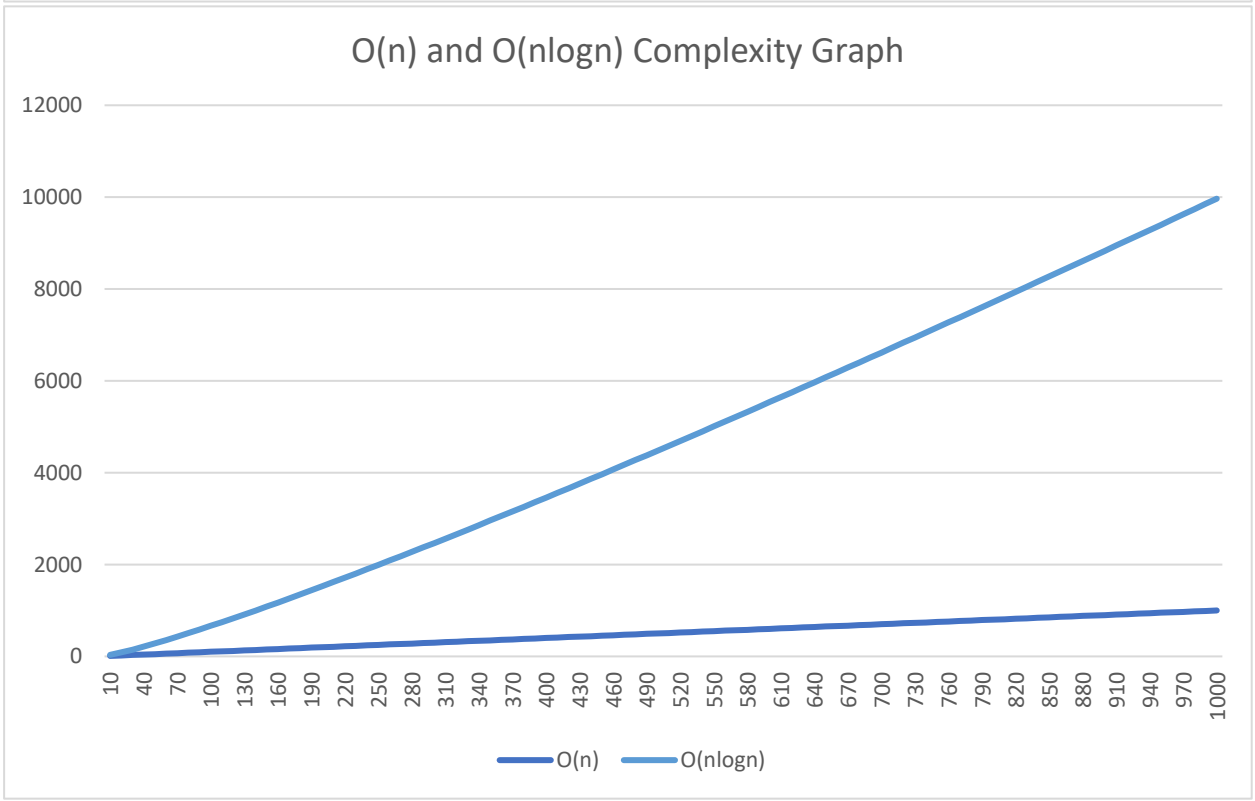
Analysis of Sort Algorithms

Complexity of algorithm depends on how algorithm runs the code for each element of input. In this report, 3 sort algorithms are constructed and complexity is calculated by determining the number of executed statements and the time elapsed in operation. The number of statements is calculated by storing a count parameter and incrementing every time the code block executed. Time is determined by nanoTime() method under java.lang.System class. This method returns the exact time in nanoseconds. In code, the method is called twice. One is before, other is after the operation and these two time values are subtracted to find the elapsed time.

For all algorithms in the report, sorting is executed in ascending order and the size of array is called n where n is a natural number.

Complexities of Algorithms at Best and Worst-Case





Insertion Sort:

This algorithm sorts elements by inserting every single element in its correct position in the array. Initially, the program iterates over the array for all elements except the first one (names them as key or etc.) and makes some operation. In that operation, program looks the elements at the left side of the relevant element and checks how many elements is bigger than key. Then, key element moves to the right in that amount. In order to keep the other elements stored, we have to shift that number of elements to the left. After that takes the other element in the iteration and does the operation again.

For complexity analysis the algorithm does an iteration and makes another iteration in the loop. Operation is executed n^2 times just because of the iterations. There isn't any operation that runs more than that, so the complexity is O(n^2) for average case. In the second iteration, program look how many elements at the very right is bigger than key. If the element at the very right is bigger, program doesn't finish the iteration. So, the best case is an array that has already been sorted. In that case program just finished its first iteration and the complexity becomes O(n). On the However, if the array is reversed ordered, in the second iteration of the element at the right is always bigger than key. So, program always finishes its iteration and complexity is O(n^2) again.

Quick Sort:

This algorithm sorts elements by taking a pivot element and divides other by looking if the element is bigger than pivot or not. After the division running the same algorithm for the partitions recursively. That's why this algorithm is similar to other divide and conquer algorithms that has complexity of O(logn). However, the program includes an iteration in order to select the elements for partitions. So that the program has two different methods in the code part. In partition part program selects a pivot. Pivot can be random, always first or last part, median etc. In this report pivot is always the first element. Then iteration of the elements except pivot begins. If any element is smaller than pivot it is swapped with the elements in right side that has been iterated and bigger than pivot. After that, the pivot is swapped with the last element that is bigger than pivot. Finally, all elements are divided with the pivot. In the recursion part, the right part and left part are operated again.

For complexity, in the best case, if the pivot comes up to the median element. The size partitions will be the half of n, so the pivot keeps coming up to median value, the recursive part will be called logn times. In addition, there is an iteration of n-1 times in the partition part. The complexity in the best-case in O(nlogn). But in the worst-case, pivot can be the smallest or largest element and there will be just one partition of size n-1. That's why recursion is called n times and complexity will be O(n^2). Average is also O(nlogn) because in the average-case there cannot be much bigger degree in terms of n in the growth function.

Selection Sort:

This algorithm sorts elements by selecting the smallest element and making it as the first element and so on. This algorithm is relatively inefficient because program iterates over the array and select a minimum value and swaps it with the first element. In order to find the minimum element, there is an iteration again to check every value if smaller one exists. After that, checking the elements except the first one and determine a minimum value again. Then swaps it with the second value and so on.

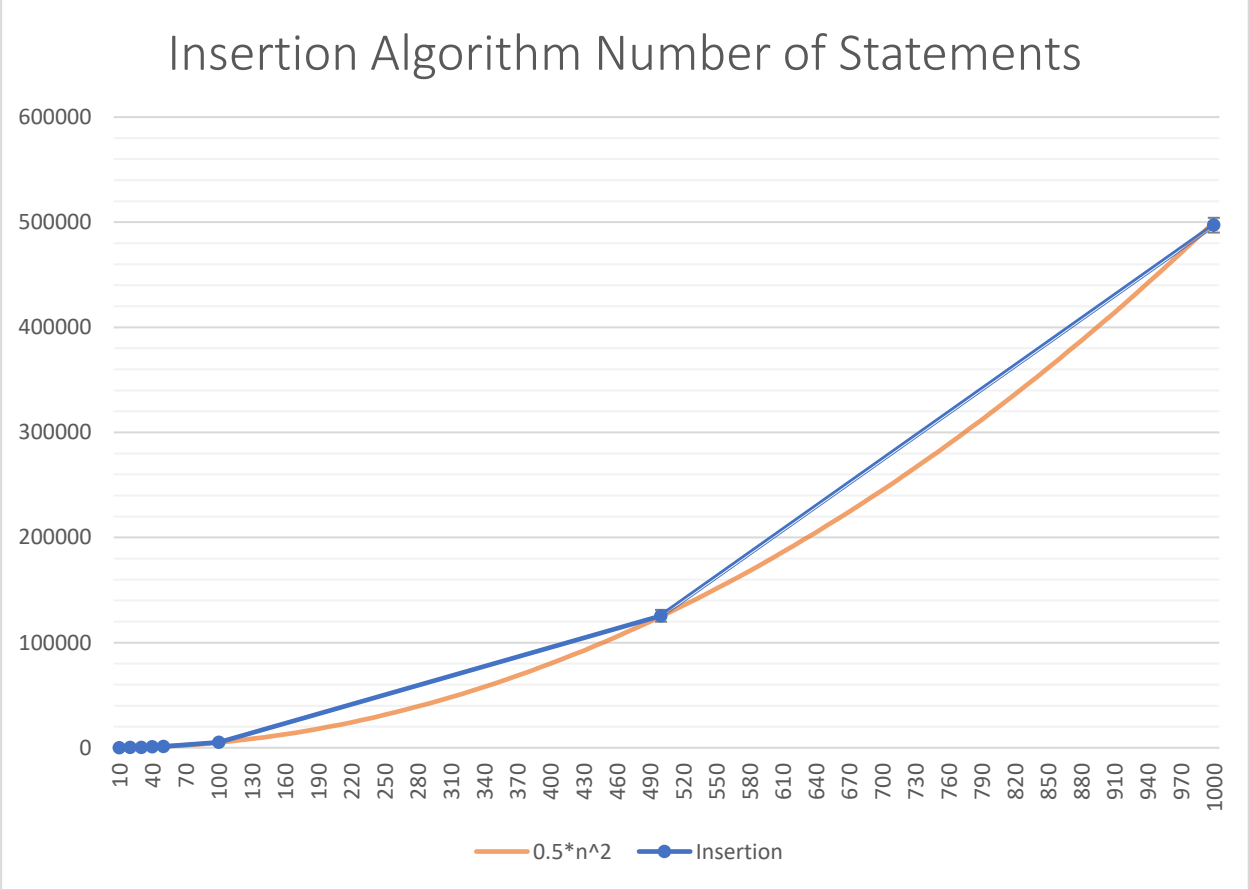
For complexity program has an iteration of size n and then has an iteration of size n-1 and so on. Mathematically total amount of operation is n^2-n and in terms of big-oh the complexity will be O(n^2). In contrast of the other algorithms, selection sort has no worst and best-case complexity. Because whatever the array is input. The program makes n^2-n operations to sort all the elements.

Performance of Algorithms with Random Input

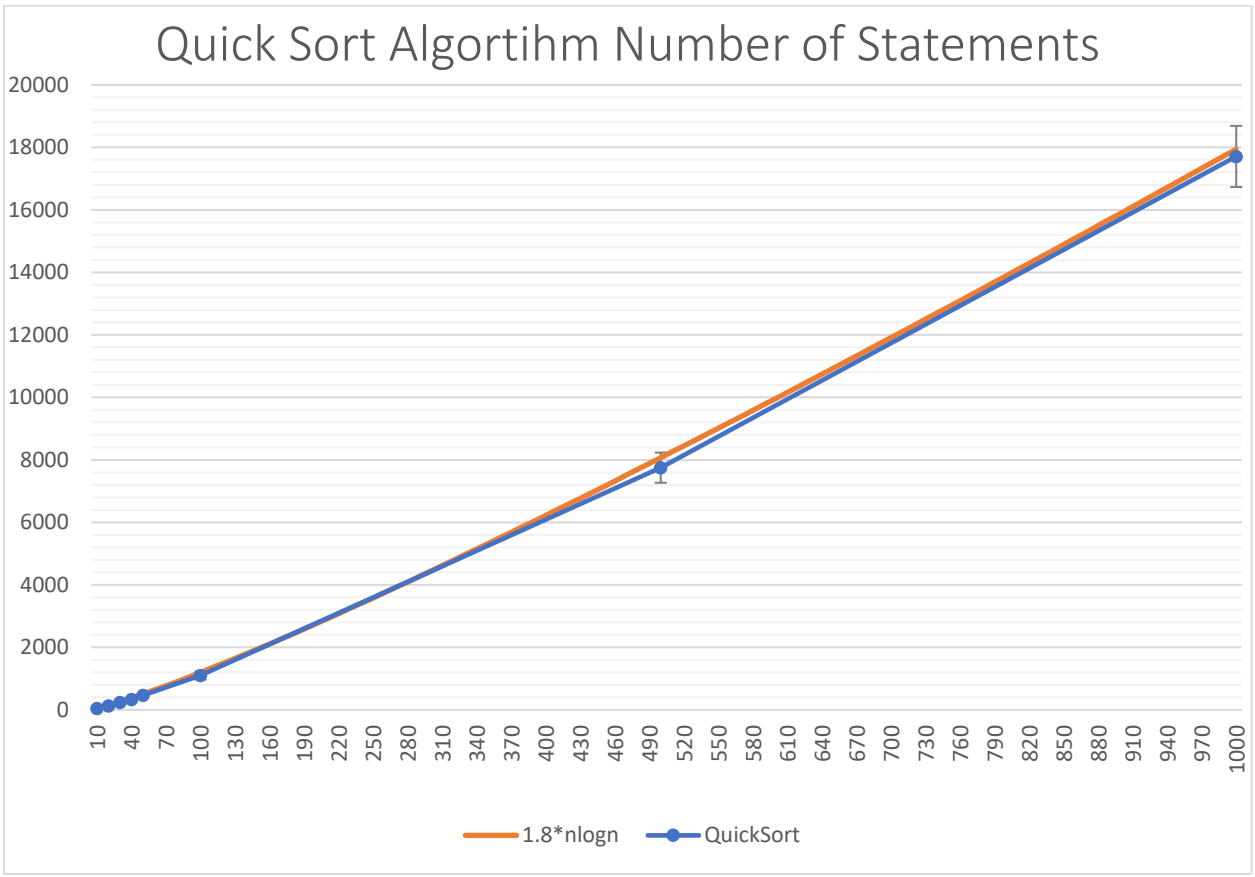
In this part of the report there will be random arrays with size n∈{10,20,30,40,50,100,500,1000}. The content of the inputs will be determined randomly. In this report, java.util.Random class will be used as a random number generator. Source code will run the experiments and stores the relevant data in some txt files. The data of the graphs in this report is based on those files. The orange line segments of the graphs are mathematical function whose content is denoted in the legend. These functions are intended to show that the experimental values fit the big-oh complexity of algorithms. There are some coefficients in mathematical functions. These coefficients are determined experimentally to overlap the graphs. Since big-oh notation is independent of coefficients of functions, we can freely select a coefficient to fit the graphs.

Number of Statements in the Algorithms and big-oh Complexity Graph:

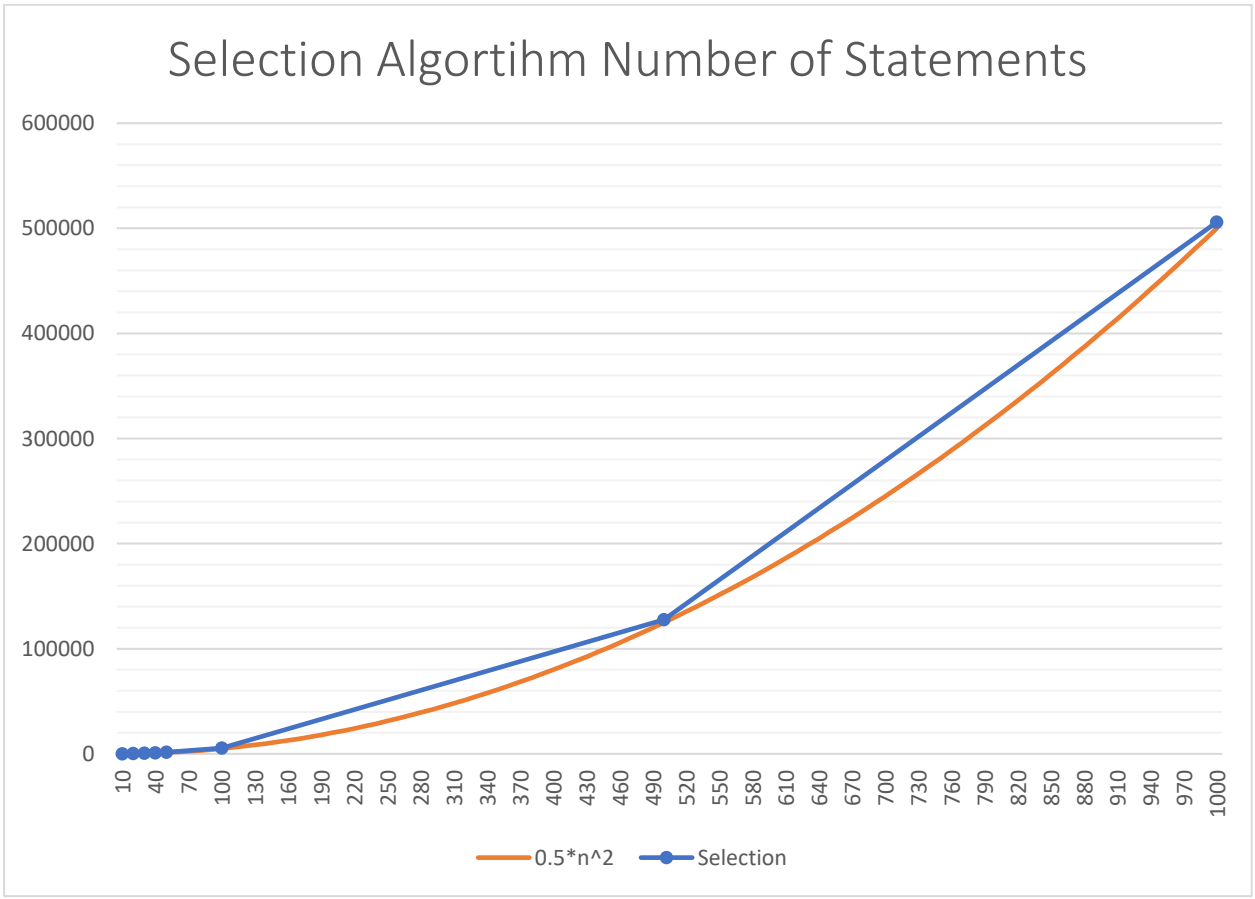
In the graph below, there are some data includes the number of executions of the innermost statement in insertion sort algorithm. We can interpret the mean and standard deviation values like this: Despite the fastest best-case scenario of these three algorithms is in insertion algorithm, the performance with random input is inefficient. Insertion algorithm works well in specific inputs, but most of the cases are bad case for this algorithm. That's why the variance of the algorithm is not small while the general graph fits with the worst-case complexity.



Here are the experimental values of Quick Sort Algorithm. Unlike the other algorithms, Quick Sort has O(nlogn) best-case complexity. In the graph, it is clear that average-case is similar the best-case. In addition, standard deviation is so high. That means there is complete difference between the good and bad cases.



In the graph below, complexity graph of Selection Algorithm is drawn. When viewed, the remarkable point is that standard deviation is so low, even the n goes higher and higher, variance does not change. That means Selection algorithm almost works same in different-mixed input. This gives the algorithm stability which Quick Sort doesn't have and Insertion barely has. This doesn't mean that the algorithm is fast. Selection sort works worse than other two algorithms.



Time Measurement in the Source Code and Complexity Graph:

Sometimes counting the number of executed statements don't infer the performance of algorithm alone. Measuring the execution time is good reinforcement to determine the complexity of algorithms. In this report, there will be some graphs of algorithms with complexity function of n multiplied by a coefficient in order to fit the line bars.

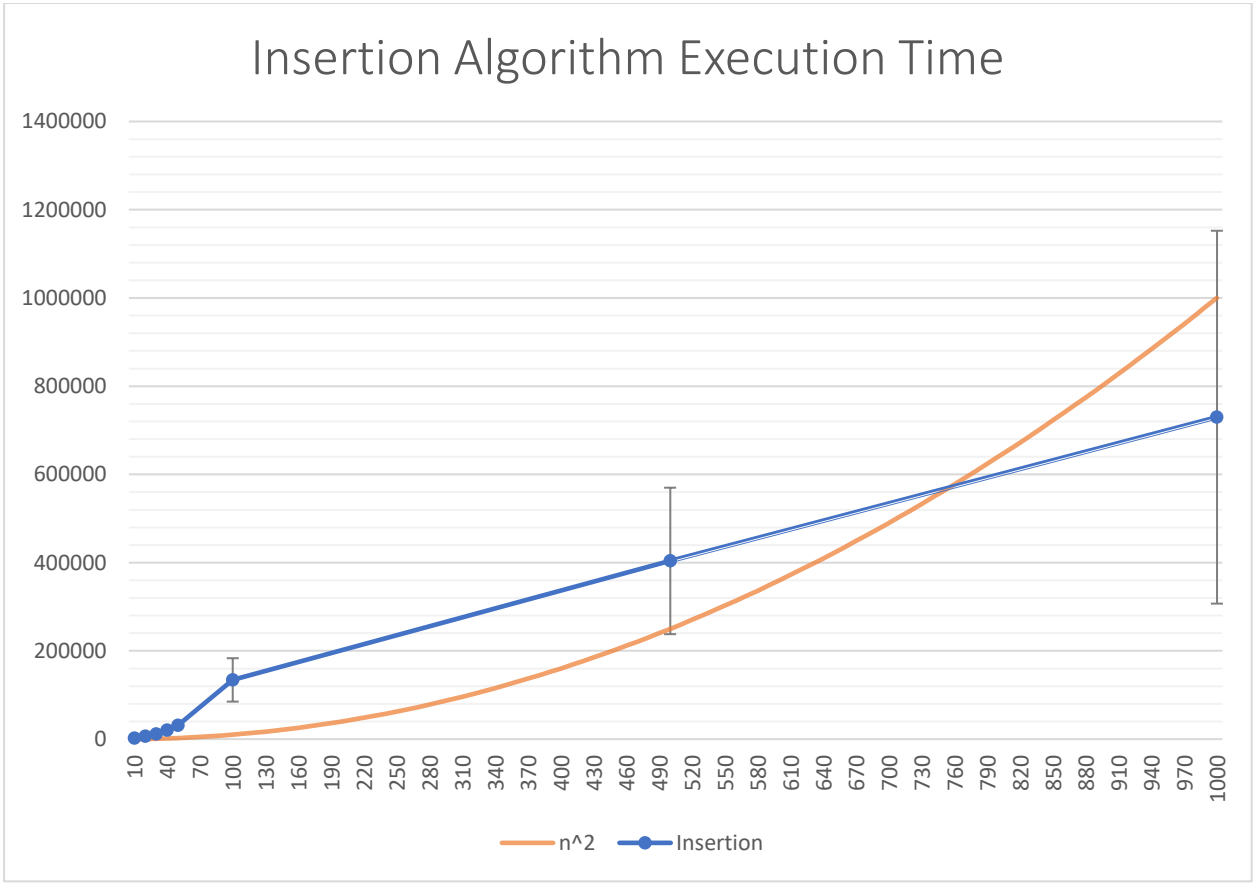
The execution time varies depending on the other processes running at that time in the computer. That's why same experiment is repeated with more browser tabs, running a game backwards etc.

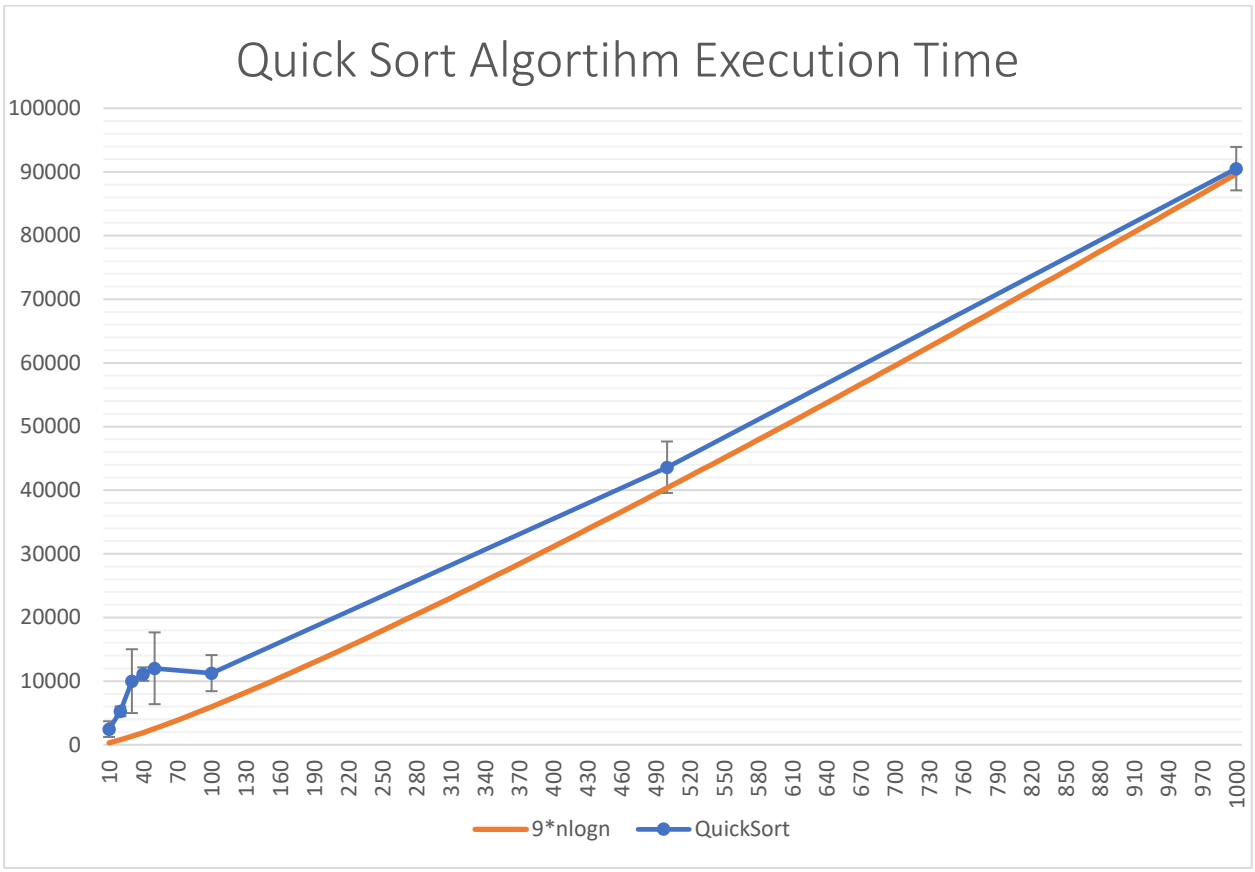
The Graphs with the Same Experiment:

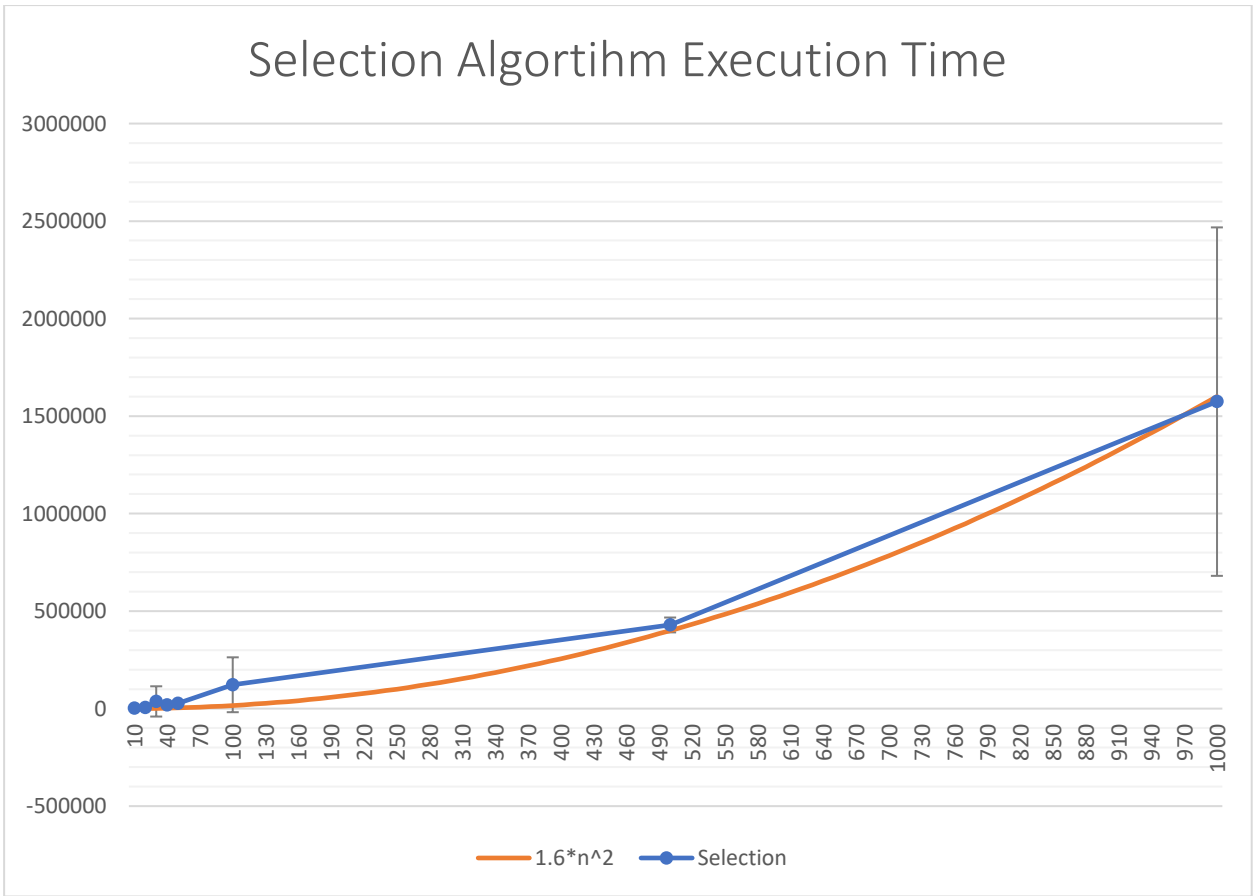
The graphs down below shows execution time of size n. The time values are given in nanoseconds. The orange line bar is again a mathematical function of values fitting the empirical data. This empirical data is recorded at the same time number of statements recorded above so that the random cases are same with the previous part.

As an interpretation of these graph, the standard deviations may seem unexpectedly high for the relevant algorithm. For example, Selection Sort must have the least variance in the charts, especially less than Quick Sort because of the working principle of algorithms. However, it is clear that the variance of execution time doesn't only depend on working principle but the execution time itself. When the computer is in idle, it has other tasks to operate and we cannot determine when the computer is busy with other tasks. So the longer execution time, the higher possibility the computer operate something behind. In addition, Number of experiments is 10 which is scarce for calculate variance.

Apart from all these, there are sudden dislocation in the graph at the small n. It is obvious in Quick Sort graph (because of small intervals in the y-axis) and less obivous in other graphs. Dislocation at small n is normal because the algorithms has bias execution time in terms of lower degrees of n so that they aren't in the big-oh notation. The program does some operation at the beginning of sorting and this operation takes significant time when n is small. When n goes to infinity, this initial operation fades next to big numbers of n and graph converges to what it is supposed to be.

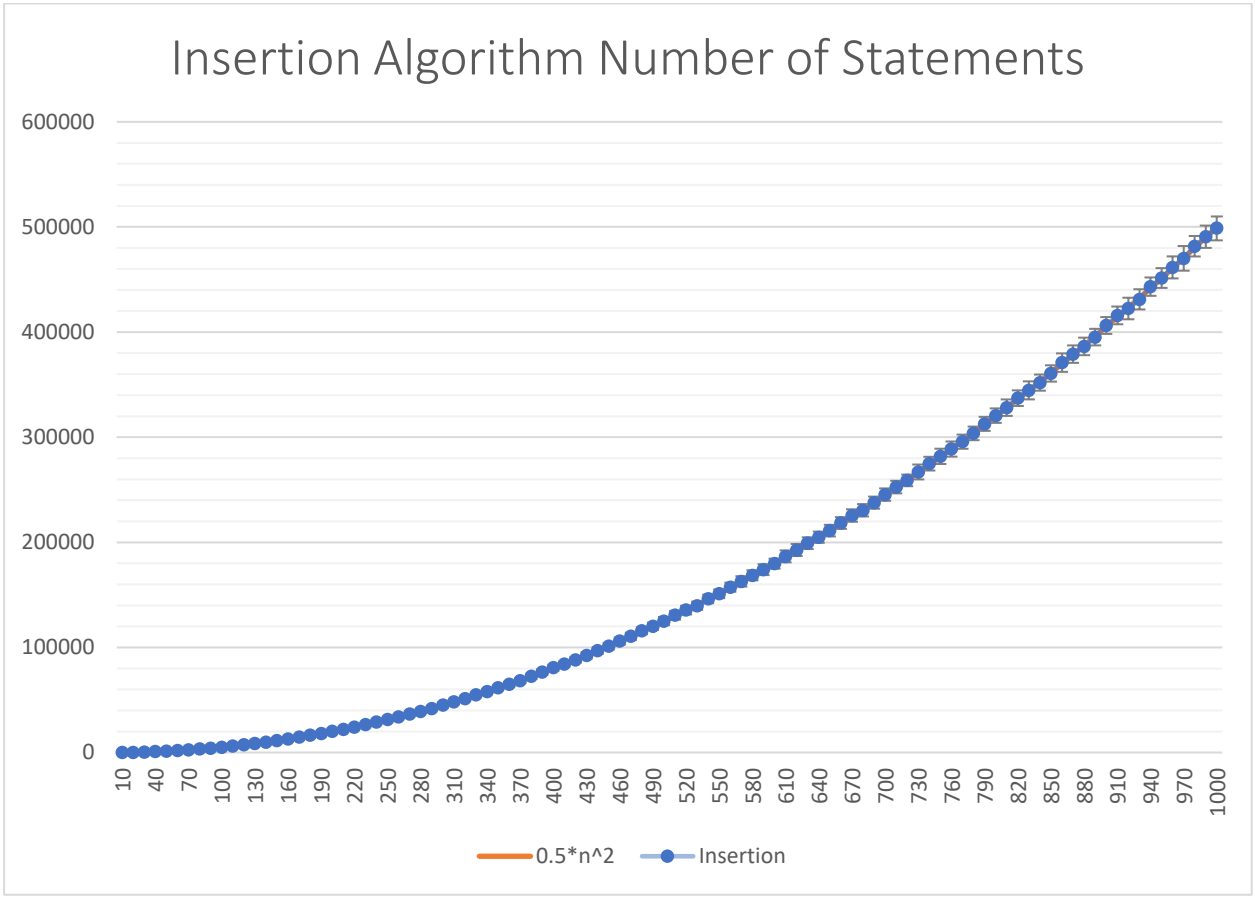


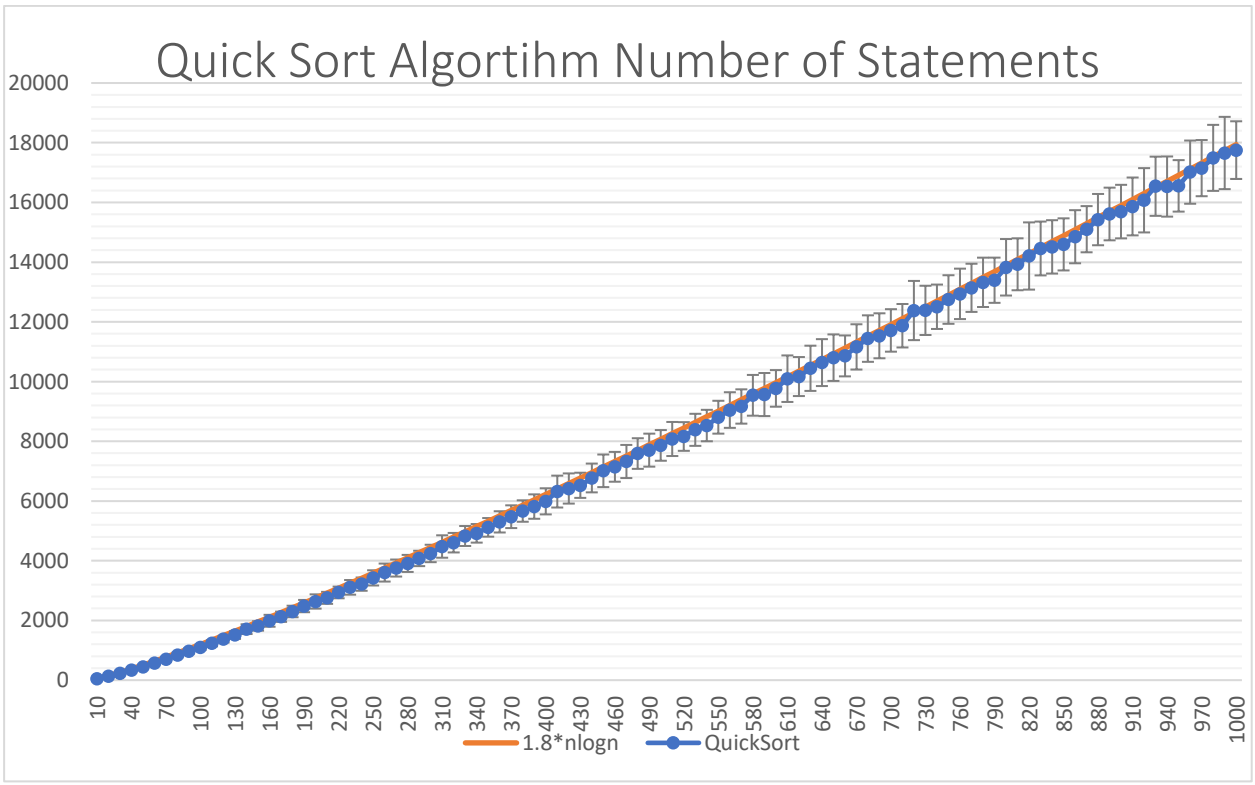

Quick Sort Algortihm Execution Time
100000
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
10 40 70 100 130 160 190 220 250 280 310 340 370 400 430 460 490 520 550 580 610 640 670 700 730 760 790 820 850 880 910 940 970 1000
9*nlogn
QuickSort

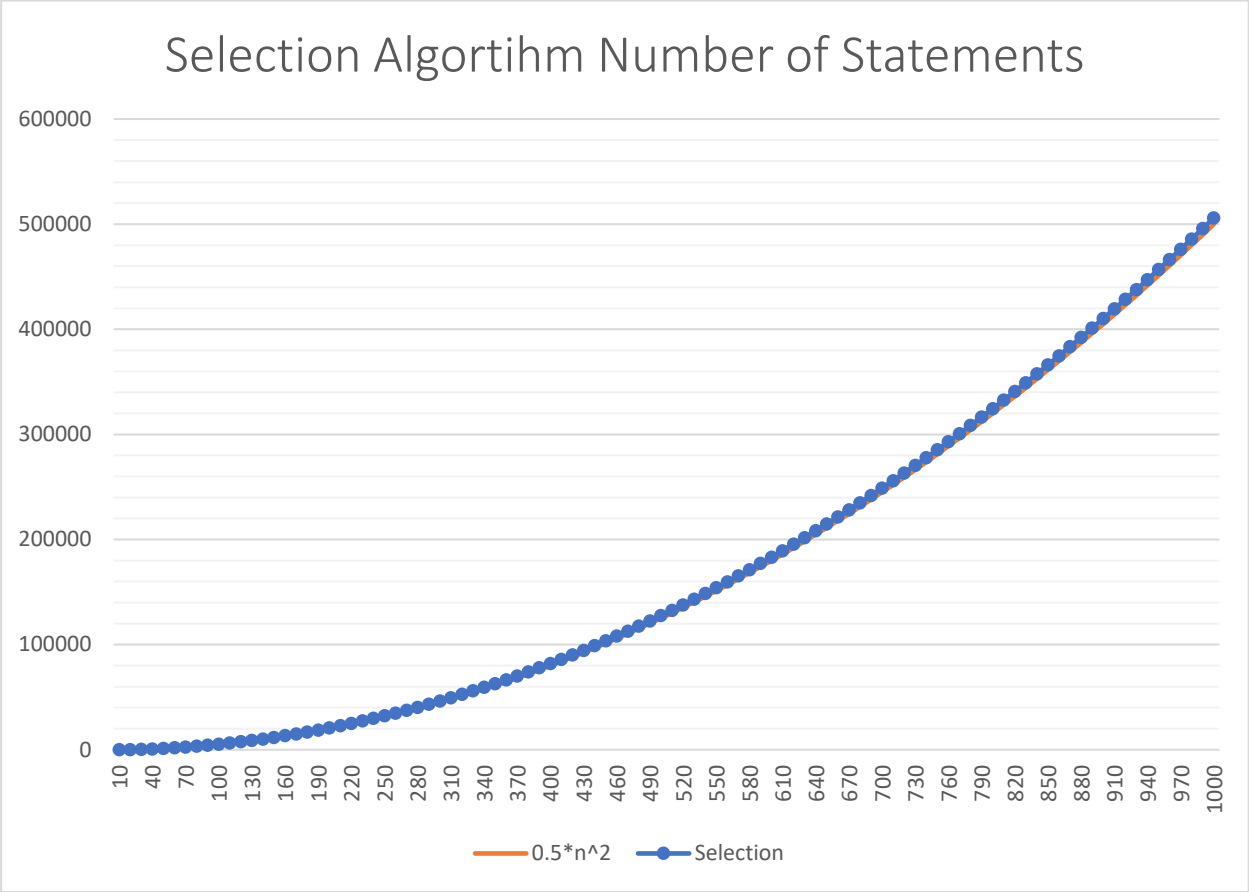

Selection Algortihm Execution Time
3000000
2500000
2000000
1500000
1000000
500000
0
-500000
10 40 70 100 130 160 190 220 250 280 310 340 370 400 430 460 490 520 550 580 610 640 670 700 730 760 790 820 850 880 910 940 970 1000
1.6*n^2
Selection

Experiments on a Bigger Scale:

The experiments are made by innermost statements counting and executing time calculating. These two methods give a lot of information to understand how algorithm works while increasing size of input. Because of the few repeat of same size of input, the empirical data is farther away from the theoretical values. So, same experiment is now running again with a huge scale of input and repeat number. The sizes of inputs are n for in range 0 to 1000 with 10 increment and repeat number is 100. While the code is running, the computer is occupied in snatches to observe fluctuations in data graph.





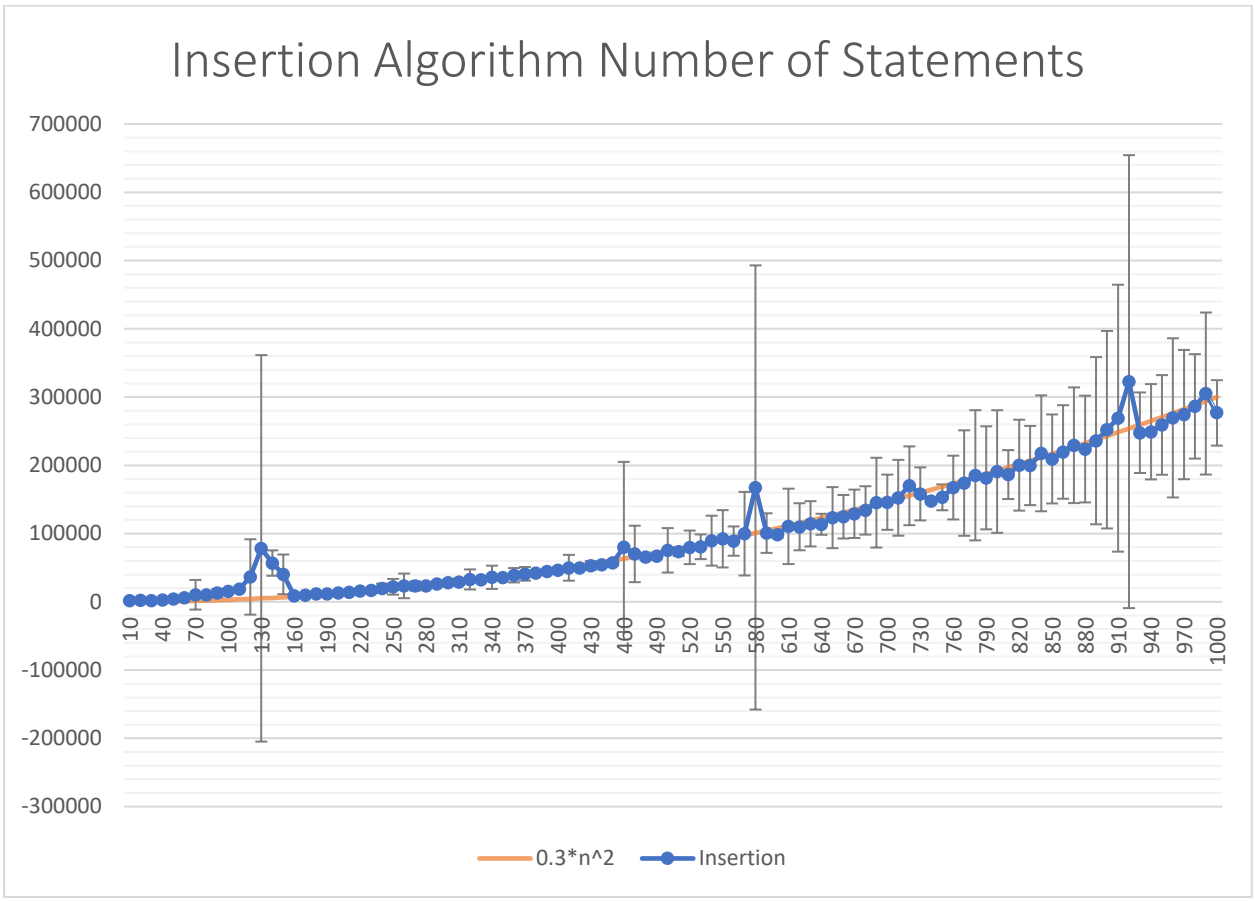


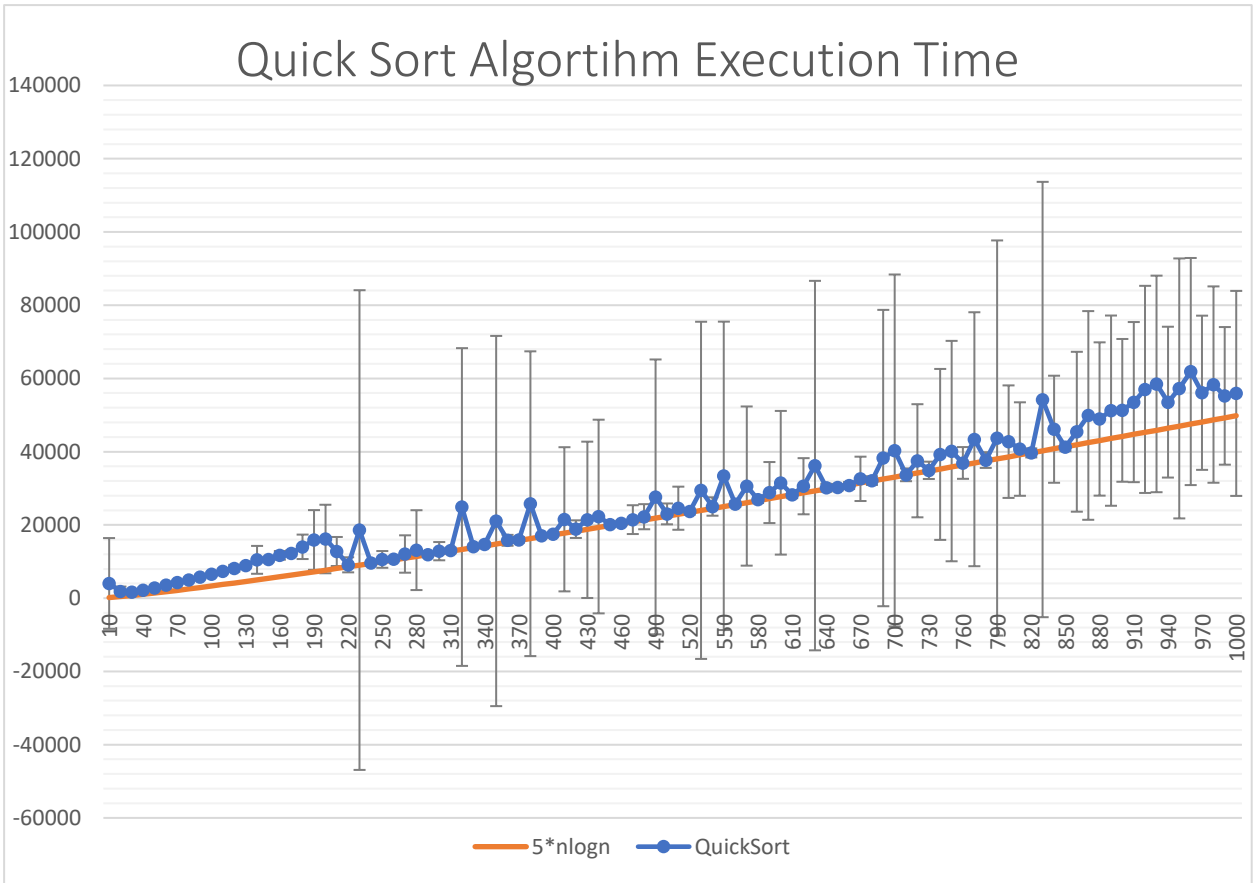
Number of Statements:

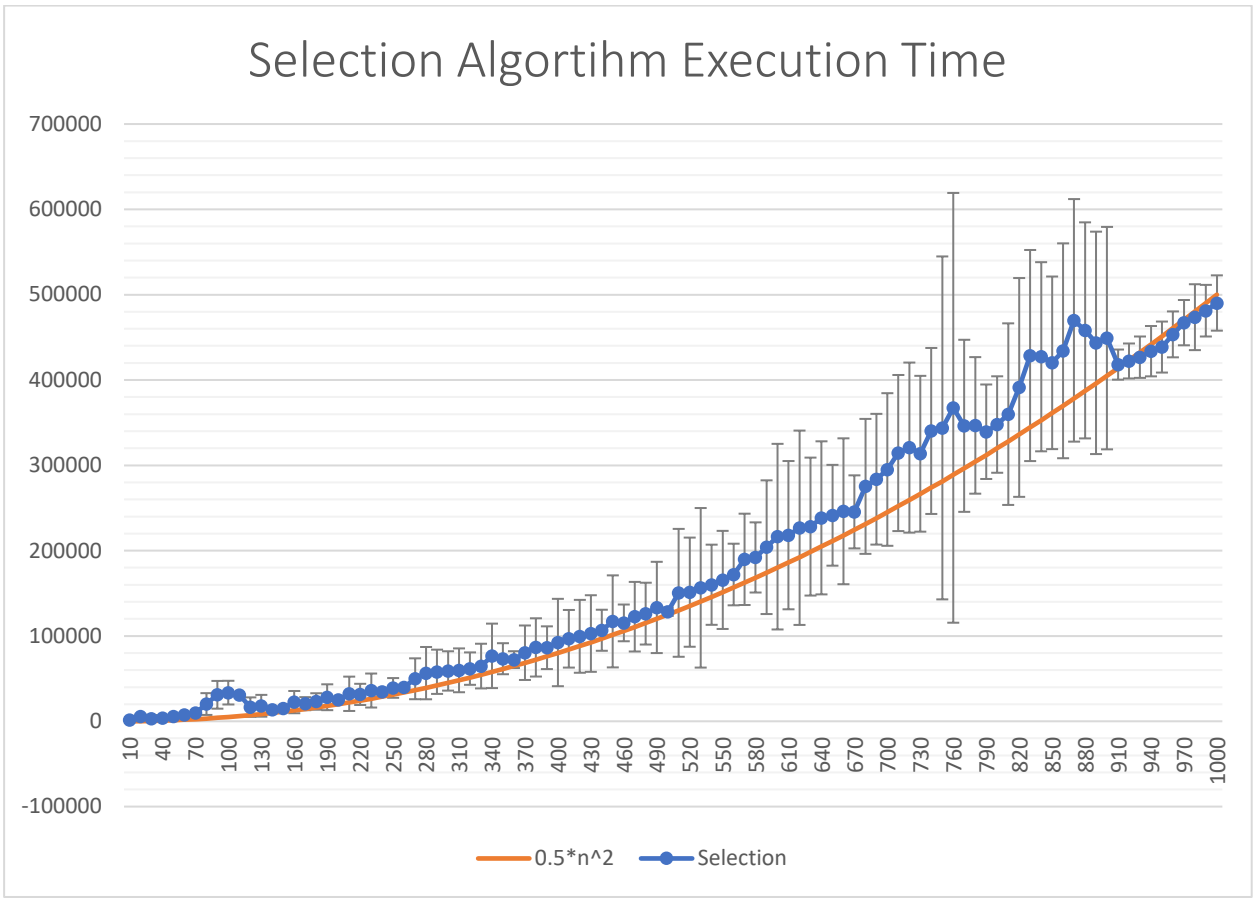
When we look at the number of statements graphs of algorithms, it is more clear that the complexities converge to big-oh functions. The difference between good and bad-cases of algortihms can be compared by looking the error bars. Quick Sort algorithm has the largest error in the graph cause the bad-cases have O(n^2) complexity while average case has O(nlogn). Selection has negligible error because of the lack of bias in inputs of algortihm. Insertion has an O(n) good case complexity while average and bad has O(n^2) but it is clear to see that the probability of good-case coming up is low. That means most of the cases are similar to worst-case and this makes the algorithm has O(n^2) average complexity.

Execution Time:

The execution time complexity growths are shown below. While looking at the graph, at some parts the graph has deviated mean values and high error bars at some points. When the computer is occupied at some other tasks willingly, the average execution time slightly increases and standrad deviation is so high.


Insertion Algorithm Number of Statements
700000
600000
500000
400000
300000
200000
100000
0
-100000
-200000
-300000
10 40 70 100 130 160 190 220 250 280 310 340 370 400 430 460 490 520 550 580 610 640 670 700 730 760 790 820 850 880 910 940 970 1000
0.3*n^2
Insertion


Quick Sort Algortihm Execution Time
140000
120000
100000
80000
60000
40000
20000
0
-20000
-40000
-60000
10 40 70 100 130 160 190 220 250 280 310 340 370 400 430 460 490 520 550 580 610 640 670 700 730 760 790 820 850 880 910 940 970 1000
5*nlogn
QuickSort



Conclusion

When sorting elements by ascending order, there are some different approaches to the cases. In some cases, some algorithms are much more efficient while in some cases they are not very efficient. But in the average, every algorithm has complexity and this complexity can be determined. In order to determine the value, some experiments must be executed and the results must be interpreted by looking the average and variance.

This report is prepared to show that when the size of input and number of repeat is increased, the performance of the algorithm converges to its big-oh complexity. This complexity gives the general idea of how algorithm works with different sizes of input and its general performance.