CAB301 – Assignment 2

Zhi Choon Ng (09056572)

Siu Keun Hoh (09026681)

# Summary

This report describes the outcomes of the experiment conducted to test the algorithm. The algorithm will rearrange an unordered list of numbers. During this experiment, the basic operation will be counted, as well as the execution will be measured. The algorithm was initially tested with static data to make sure it is working as intended. After that, random numbers are used to test the algorithm once a bigger size is reached.

# 1 Description of the Algorithm

The first algorithm (Figure 1.0) is implemented to search for the median of an array through Brute Force. Utilizing a nested *for* loop, The algorithm will check if an element in the array is smaller or equal to *A[i]*. If the element is smaller, *numsmaller* will increase by one. Otherwise, it is assumed to be equal or bigger. In the *else* statement which covers the two other options, the element is checked again. If it the element size is equal to *A[i],* then *numequal* will be increased by one. Upon exiting the second *for* statement, if *numsmaller* is smaller than *k* (half of the array size) and *numsmaller* and *numequal* is bigger or equals to k, *A[i]* is then returned. As both for statements will repeat as many times as they are elements in the arrays, this will lead the algorithm to be extremely inefficient as it required to run n2 times.

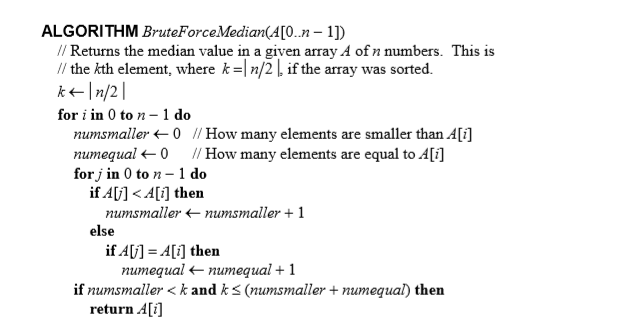


Figure 1.0 – Brute Force Algorithm. Let *n* denote the size of the array and k denote the size of half the array.

The second algorithm (Figure 1.1 to Figure 1.3) is however more efficient, but requires the implementation of three different methods. The three methods in this case are the *Median, Select* and *Partition* algorithm. The Median method calls the Select method, which in turn calls the Partition algorithm. The Partition algorithm (Figure 1.3) will slice the array by moving element *A[l]* to the position it would if the slice was sorted with *l* being the location of the pivot.. It will also move all the values smaller than *A[l]* to the front half of the array, while larger to equal will be moved behind. This is done by swapping the elements around the pivot. This will be done repeatedly until *l + 1 = h.* At the end of the method, it will return the element location in which the middle.

The result of the Partition will then be brought into the Select method. In this method, the result of the Partition method will be denoted as *pos.* If the result is equal to m, the position of the array will be returned as is. If the result is larger than *m,* the Select method will be run. However, the end of the indices is reduced by one. In the final scenario, if the result were to be smaller than *m,* the select method will be run but the start of the indices will be increased by one instead. This shows that the element before the new indices is in the correct location.

Finally, the result will be brought to the Median method (Figure 1.1). The actual median has actually been found in the Select method, but the Median method is used in the scenario where the array has only one element. In that case, the Median will be the only element in it. Else, it will return the result of the Select method.

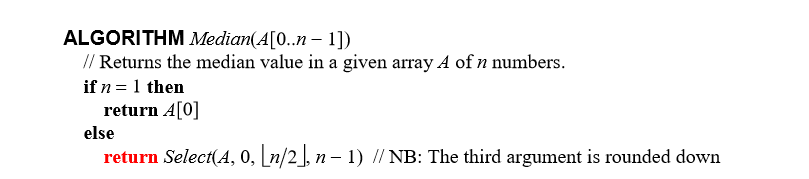


Figure 1.1: Median method. Let the array size be denoted as n. The Select method is covered in Figure 1.2.

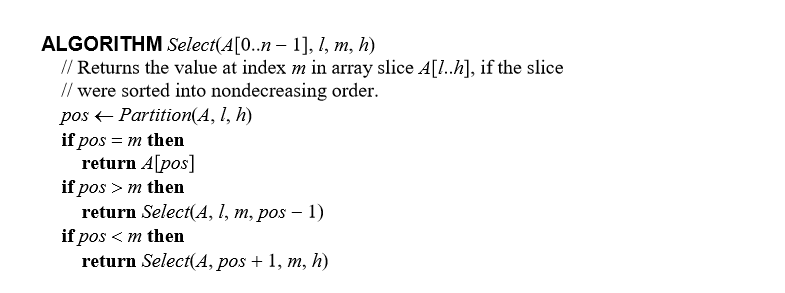


Figure 1.2 Select method. The partition method is covered in Figure 1.3.

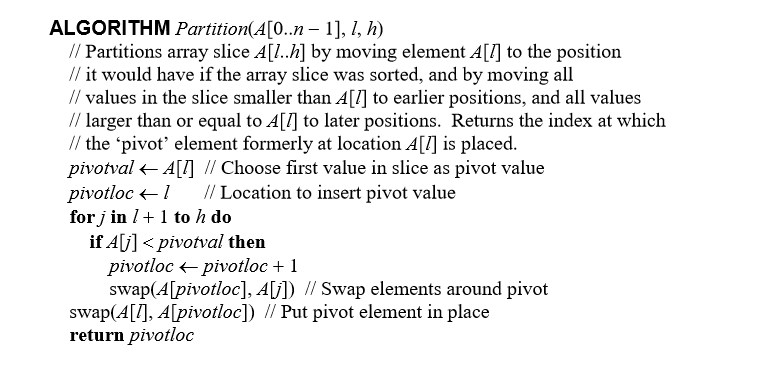


Figure 1.3 – Partition method.

# 2 Theoretical Analysis of the Algorithm

## 2.1 Choice of Basic Operation

The choice of basic operation for the first algorithm is the comparison between the elements in the array, namely the comparison made in the nested loop. We chose it because comparisons impact the performance of a sorting algorithm the most, as stated by Levitin (2012, p. 158).

For the second algorithm, the choice of basic operation is once again, comparisons between elements in the array. That puts our choice of basic operation in the partition function.

## 2.2 Choice of Problem Size

The choice of the problem size is the number of items in the supplied array.

## 2.3 Average-Case Efficiency

### 2.3.1 Brute Force Median Algorithm

The brute force algorithm is not able to exit early, so the best-, worst-, and average-case efficiency should be the same. In regards to the amount of basic operation performed, the best scenario would be having to execute only the first conditional **if** statement without having to execute the second one. However it should not impact the overall efficiency of the whole algorithm.

For every *n*th element in the array, the basic operation is ran *n*th times, resulting in *n \* n*. This would mean that the expected average case efficiency of the algorithm would be quadratic.

### 2.3.2 Partition Median Algorithm

The median algorithm is able to exit early, so the best-, worst-, and average-case will differ from each other. Regardless, we are only interested in the average-case efficiency in this report.

As the chosen basic operation for the first algorithm is a condition if statement that compares between elements in the array, we chose a similar line as the basic operation in the median algorithm.

According to Cormen (2009, p. 177-178), the partition function should have an average-case efficiency of *n log n*. However, the average-case efficiency of the median algorithm overall is linear. The reason seems to be due to the selection function only requiring to recurse on one side of the partition. Workings done by Cormen (2009, p. 215-219) reaches the same conclusion, although their own version of selection and partition is used instead.

# 3 Methodology, Tools, Techniques

## 3.1 Programming Environment

The algorithms and the experiments were implemented using the Java programming language. This is chosen as it is the language we are most familiar with and have sufficient experience in using it. Eclipse is used as the language environment as it is the simplest and most elegant to use.

These algorithms and experiments were run on a laptop with an Intel i7-7500U Processor and 8GB worth of RAM. It uses the Windows 10 operating system. Java’s pseudorandom number generator library, the *java.util.Random* was used to produce test data. Since execution time is needed, the *java.lang.System’s nanoTime* is used to get it. Since the execution time is important, all other software apart from Eclipse has been turned off to ensure a stable and unbiased collection of data.

Graphs of the experiment results were created using Microsoft Excel. Using the *java.io.FileWriter*, the execution time of 50 different tests, starting from 10000, with the increment of 10000 per test, are recorded into the excel document. This allows simple conversion of data to a graph to be used in this report.

## 3.2 Implementation of the Algorithm

Both algorithms implementations respond to their pseudocode examples in terms of correctness. There are no deviations from the original with the exception of basic operation counter is the *BruteFoceMedian* algorithm and the *Partition* method. The usage of a *max* is also used when the random numbers are generated. In the Brute Force Median algorithm, a return A[0] is implemented in the case there is only one element.

The algorithms are called by the *AutoCheck()* function both which relies on the arraySize specified in the main method. With this, the array size and can be easily controlled. This will allow easy testing of both algorithms with a large sample size. Finally, the functionality tests are called by using the *TestCaseCheck()* method.

## 3.3 Generating Test Data and Running the Experiments

To test the correctness of the implementations of the algorithm, a test program is written in the *TestCaseCheck* method. The implementation test involves positive and negatives, even and odd number of elements, arrays that require rearranging and those that did not, duplicates and a single element array. With these tests, it can be seen the algorithms are implemented correctly. However, as per the original document, there are some differences when an even number of elements are provided. The Brute Force Median algorithm will choose the number on the left of the pivot, while the Partition Median algorithm will choose the number on the right of the pivot.

To calculate the number of basic operation of each of the algorithm, the *boCount* counter was implemented in both algorithms where it is necessary. The basic operation counter was then tested on arrays with varying sizes. This is to make sure the basic operation counters were placed in meaningful places. The implementation of the *boCount2* serves the same purpose as *boCount*, but it will allow both algorithms to run using the same array. This will make sure that the comparative results are meaningful.

To calculate the execution time, two variables were created *long startTime* and *long endTime.* *startTime* is ran at the start of the algorithm in the *AutoCheck* function while *endTime* is ran at the end of the algorithm. By subtracting the difference between *endTime* and *startTime,* we’ll be able to to find out the execution time of the algorithm with that size. Similar to the basic operation, *startTime2* and *endTime2* are implemented with the same function, except it is in charge of measuring the time for the Partition median algorithm. This allows us to check the execution time of the algorithms when both utilize the same array.

# 4 Experimental Results

## 4.1 Functional Testing

To test the correctness of the implementations of the 2 algorithms, 10 different test cases were written.

Test Case 1 – The set of {200,400,499,550,600}

Brute Force Median result: 400

Partition Median result: 499

The set contains 5 numbers without requiring any rearrangement. It demonstrates that the algorithm works with an odd number of elements in the array. The results are as expected as 5/2 is 2.5. As it is an double, Java rounds it down to 2.

Test Case 2 – The set of {200,300,100,500,400}

Brute Force Median result: 400

Partition Median result: 499

The set contains 5 numbers but requires rearranging. This demonstrates that the algorithms can rearrange the array correctly.

Test Case 3 – The set of {100,200,300,300,400}

Brute Force Median result: 200

Partition Median result: 300

The set contains 5 numbers and a duplicate. This demonstrates that the algorithm is able to function correctly with duplicates.

Test Case 4 – The set of {100}

Brute Force Median result: 100

Partition Median result: 100

The set contains only one number. This test was done to check if the algorithms work with only a single element.

Test Case 5 – The set of {100,200,300,400,500,600}

Brute Force Median result: 300

Partition Median result: 400

The set contains 6 numbers. This test is done to check if the algorithm works with an even number amounts of arrays.

Test Case 6 – The set of {600,500,400,300,200,100}

Brute Force Median result: 400

Partition Median result: 300

The set contains 6 numbers that require arranging. This test is done to make sure the algorithms are able to rearrange an even-numbered amount of elements array.

Test Case 7 – The set of {550,500,450,400,350,300,250,200,150,100}

Brute Force Median result: 350

Partition Median result: 300

The set contains an extreme number of elements, having 10 numbers. This test is done to ensure the algorithm functions correctly in an extreme case.

Test Case 8 – {-300, -200, 0, 100, 200}

Brute Force Median result: -200

Partition Median result: 0

The set contains 5 numbers and a negative number. This test is done to ensure the algorithm functions correctly when a negative number is present.

## 4.2 Basic Operation Comparison

To measure the algorithm’s basic operation, the test program in Appendix X was used. The test is run with the array size from 10000 with an increment of 10000 per test. These arrays were filled with random numbers from -0 to 100000. The results for the Brute Force Median algorithm is shown In Figure 4.2.1 and the Partition Median algorithm is shown in Figure 4.2.2.

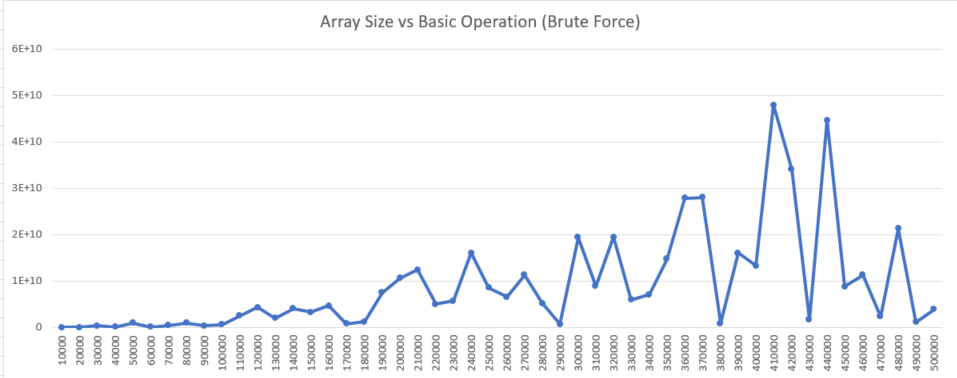


Figure 4.2.1 Array Size vs Basic Operation Count for the Brute Force Median algorithm. There should be 50 data points, with an increasing size of 10000. While the results are rather noisy towards the end, this confirms that the basic operation count grows exponentially.

At the same time, the exact same array is used to test the Partition Median algorithm. The results are shown in Figure 4.2.2

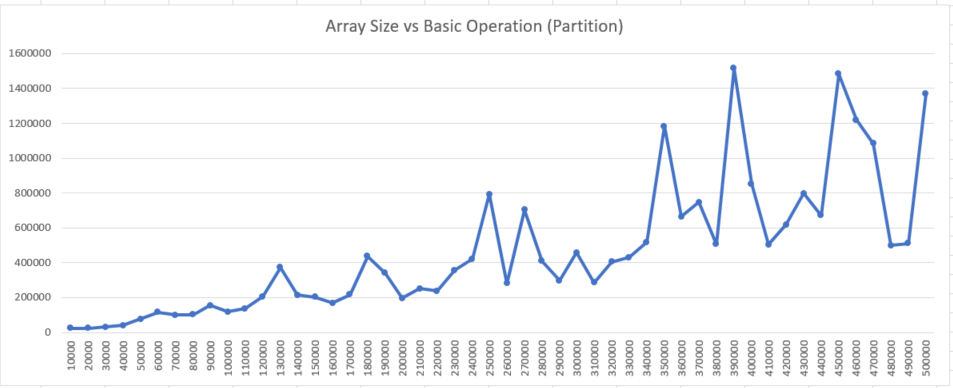


Figure 4.2.2 Array Size vs Basic Operation Count for the Partition algorithm. There is 50 data points, with an increasing size of 10000. The result confirms that the basic operation count of the partition algorithm grows linearly.

From the two graphs, it is obvious that the Brute Force algorithm basic operation count grows exponentially while the Partition algorithm grows linearly. This is consistent with the theoretical predictions.

## 4.3 Execution Time comparison

To measure the algorithm’s basic operation, the test program in Appendix D was used. The test is run with the array size from 10000 with an increment of 10000 per test. These arrays were filled with random numbers from 0 to 100000. The results for the Brute Force Median algorithm is shown In Figure 4.3.1 and the Partition Median algorithm is shown in Figure 4.3.2.

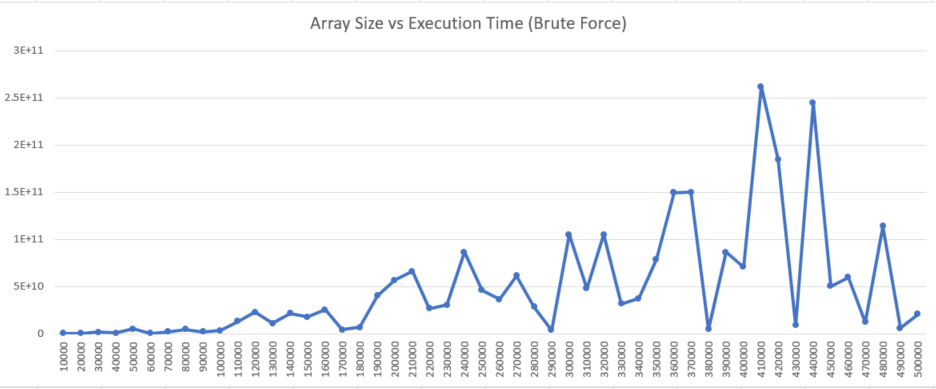


Figure 4.3.1 - Array Size vs Basic Operation Count for the Brute Force Median algorithm. There is 50 data points, with an increasing size of 10000. The result confirms that the basic operation count of the partition algorithm grows exponentially.

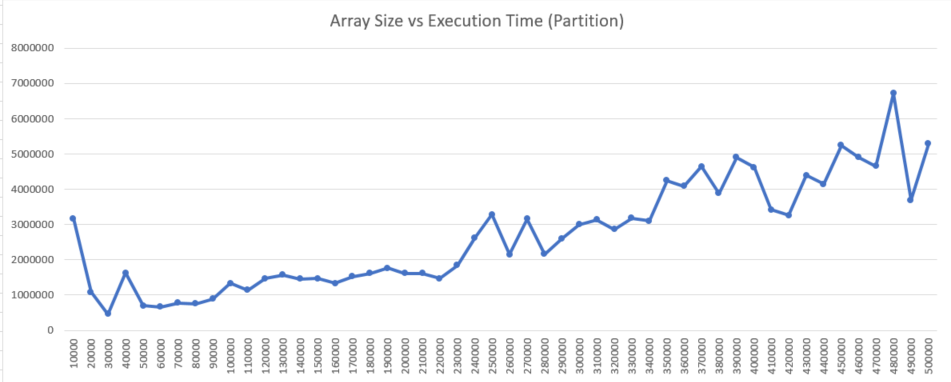


Figure 4.3.2 - Array Size vs Basic Operation Count for the Partition algorithm. There is 50 data points, with an increasing size of 10000. The result confirms that the basic operation count of the partition algorithm grows linearly.

Based on the graphs, the execution time shows that the Partition median algorithm is more efficient compared to the Brute Force Median algorithm. The execution time in the Brute Force Algorithm takes more and more time as the array size grows bigger while the Partition Algorithm time increase increases linearly. Even at the start of the test, the Brute Force Median algorithm already takes a lot more time when compared to the Partition median algorithm.

# References

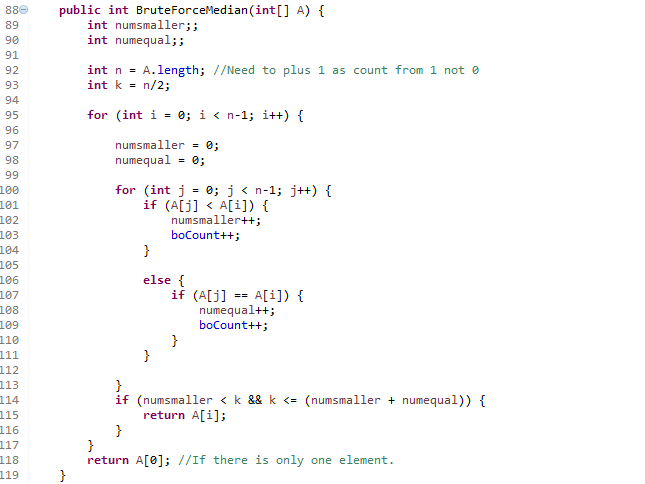
1. Cormen, T. (2009). Introduction to algorithms (Third Edition.). Cambridge, Mass.: MIT Press.
2. Levitin, A. (2012). Introduction to the design & analysis of algorithms (3rd ed., International ed.). Harlow: Pearson Education.

# Appendix A Code for the first algorithm

This appendix presents the Java code written to implement the algorithm in Figure 1.0.1. It includes several different imports that are used in the implementation. *Java.util.Random* is used to implement random numbers during the execution time test. *Java.io.FileWriter* and *java.io.IOException* is used to enter the execution time and basic operation count in the Microsoft Excel document for easy graphing.

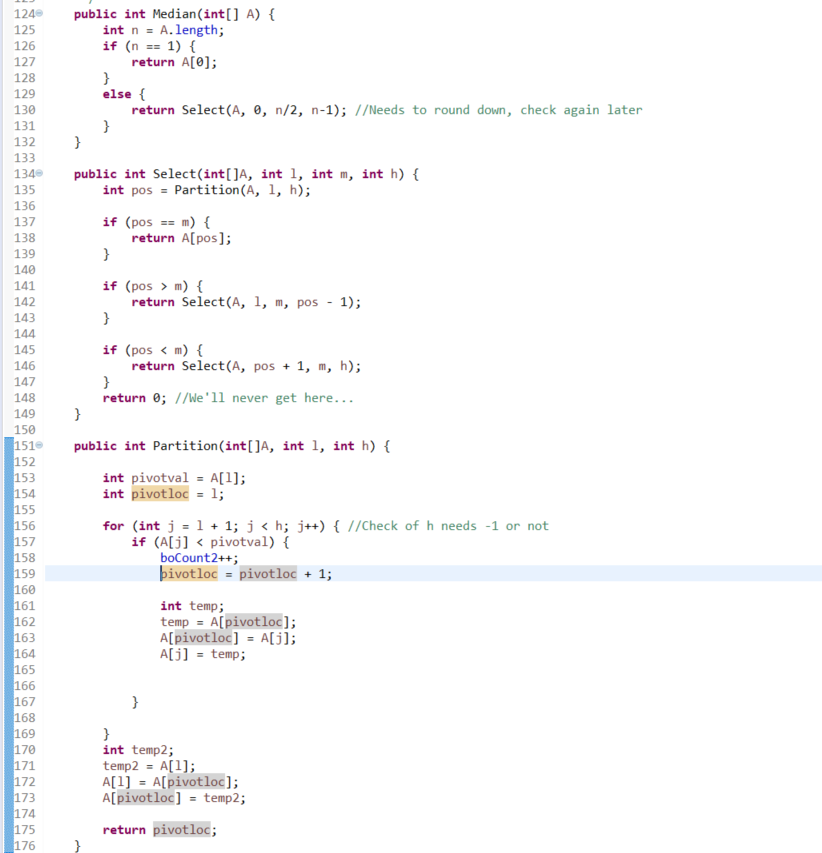


The first algorithm, which is the Brute Force Median Algorithm is contained within the *BruteForceMedian(int[] A)* method.



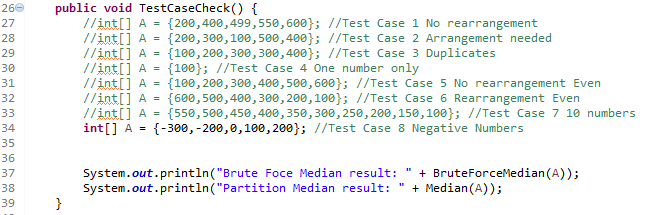
# Appendix B Code for Second Algorithm

The second algorithm consists of 3 different methods, the *Median(int[] A)* method, the *Select(int []A, int l, int m, int h)* method and the *Partition (int[]A, int l, int h)* method.



# Appendix C Code for functional testing.

The code will print out the answer every time the test is run. This allows us to test for the functionality of the algorithms. It is covered in the *TestCaseCheck()* function.

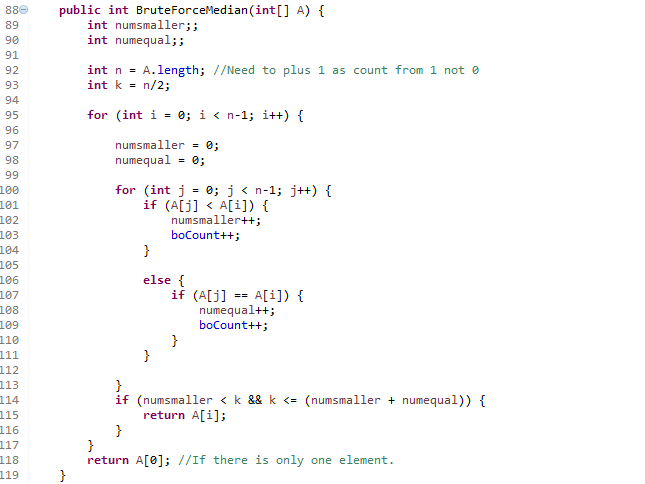


# Appendix D Code for calculating the Basic Operation Count

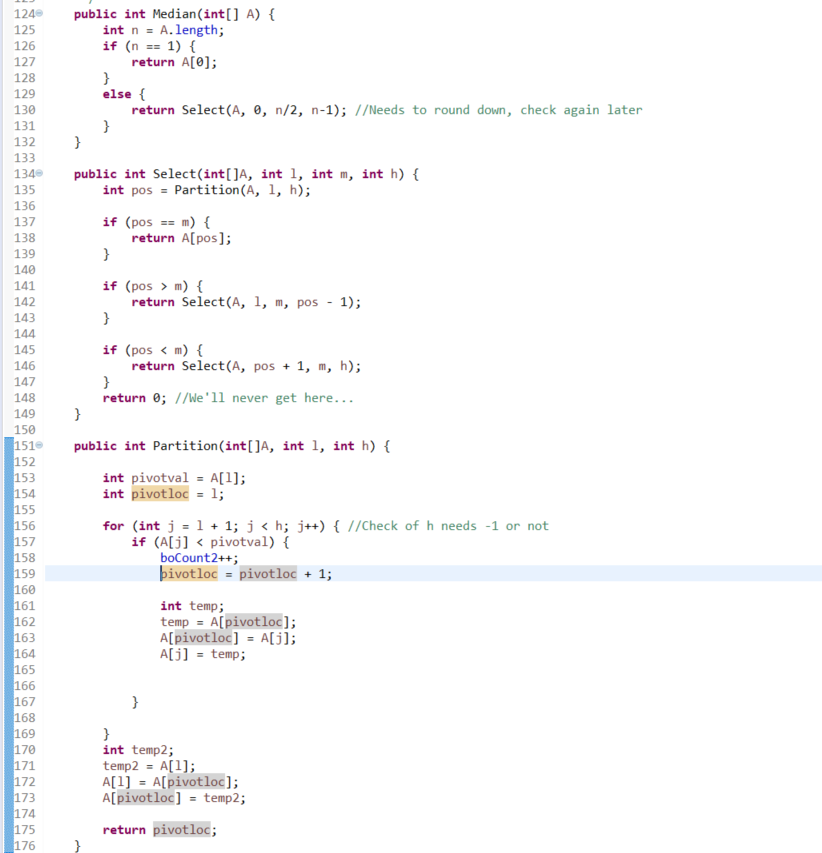
As both algorithms are going to run in unison to reduce the amount of bias, two separate counters were to be implemented:



The boCount and boCount2 are both initialized as 0 at the start of the test.

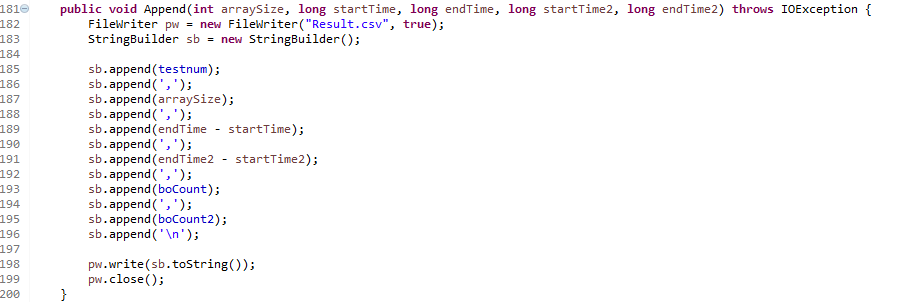


Every time the basic operation occurs in the Brute Force algorithm, the *boCount* (Line 103 and Line 109)will increase by 1.



Every time the basic operation occurs in the Median algorithm, the *boCount2* (Line 165 and Line 173) will increase by 1.

Using the method below, the basic operation as well the execution time file is written in the Microsoft Excel file.

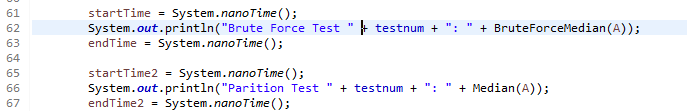


# Appendix E Code for calculating Execution Time

As both algorithms are going to run in unison to reduce the amount of bias, two separate counters were to be implemented:



*startTime* will take the time of the program at the start of the Brute Force Algorithm while *endTime* will take the time once it finished running. *startTime2* will record the time when the Partition Algorithm starts while *endTime2* will record when the algorithm is done. The results will then be subtracted at the end to find the execution time.



The data will then be written to the Microsoft Excel using the same function as the basic operation count.