CAB301 Assignment 2

Empirical Comparison of Median Calculation Algorithms

Kathryn Dorge (n9701761)

and Luke Reynolds (n10083481)

Date submitted:

Table of Contents

[1.0 Summary 3](#_Toc514366149)

[2.0 Description of the Algorithm 3](#_Toc514366150)

[2.1 Brute Force Median 3](#_Toc514366151)

[2.2 Partitioning Median 4](#_Toc514366152)

[3.0 Theoretical Analysis of the Algorithm 5](#_Toc514366153)

[3.1 The algorithm’s basic operation 5](#_Toc514366154)

[3.2 Problem Size 6](#_Toc514366158)

[3.3 Average Case Efficiency 6](#_Toc514366159)

[3.4 Order of Growth 9](#_Toc514366162)

[4.0 Methodology, Tools and Techniques 10](#_Toc514366163)

[4.1 Programming Environment 10](#_Toc514366164)

[4.2 Algorithm Implementation 10](#_Toc514366165)

[4.3 Test Data and Execution of Experiments 10](#_Toc514366166)

[5.0 Experimental Results 12](#_Toc514366167)

[5.1 Functional Testing 12](#_Toc514366168)

[5.2 Average-Case Number of Basic Operations 15](#_Toc514366171)

[5.3 Average-Case Execution Time 20](#_Toc514366178)

[6.0 References 25](#_Toc514366184)

[7.0 Figures 26](#_Toc514366185)

[Figure I: Brute Force Median Algorithm 26](#_Toc514366186)

[Figure II: Partition Median Algorithm (Base Function) 27](#_Toc514366187)

[Figure III: Partition Median Algorithm (Select Function) 28](#_Toc514366188)

[Figure IV: Partition Median Algorithm (Partition Method) 29](#_Toc514366189)

[Figure V: Brute Force Algorithm Basic Operations 15](#_Toc514366172)

[Figure VI: Brute Force Algorithm Basic Operations & Theoretical Results 16](#_Toc514366173)

[Figure VII: Partition Algorithm Basic Operations 17](#_Toc514366174)

[Figure VIII: Partition Algorithm Basic Operations & Theoretical Results 18](#_Toc514366176)

[Figure IX: Brute Force Vs Partition (Basic Operations) 19](#_Toc514366177)

[Figure X: Brute Force Median Execution Time 20](#_Toc514366179)

[Figure XI: Brute Force Median Execution Time & Theoretical Results 21](#_Toc514366180)

[Figure XII: Partition Median Algorithm Execution Time 22](#_Toc514366181)

[Figure XIII: Partition Median Algorithm Execution Time & Theoretical Results 23](#_Toc514366182)

[Figure XIV: Brute Force Vs Partition (Execution Time) 24](#_Toc514366183)

[8.0 Appendices 30](#_Toc514366190)

[Appendix I: Code for Brute Force Median Algorithm Implementation 30](#_Toc514366191)

[Appendix II: Code for Partitioning Median Algorithm Implementation 31](#_Toc514366192)

[Appendix III: Code for Generating Random Test Data 33](#_Toc514366193)

[Appendix IV: Code for Testing Brute Force Median Functionality 34](#_Toc514366194)

[Appendix V: Code for Testing Partitioning Median Functionality 35](#_Toc514366195)

[Appendix VI: Code for Basic Operation Count for Brute Force Median 37](#_Toc514366196)

[Appendix VII: Code for Basic Operation Count for Partitioning Median 39](#_Toc514366197)

[Appendix VIII: Code for Time Execution Test for Brute Force Median 41](#_Toc514366198)

[Appendix IX: Code for Time Execution Test for Partitioning Median 42](#_Toc514366199)

# **Summary**

This report details the theoretical and empirical analysis of two possible median algorithms, with one taking a brute-force approach of locating the median value and another implementing a partitioning method similar to that of the quicksort algorithm. These algorithms where implementing within Java, utilising Oracles JDK, stable release 8. The empirical analysis will be conducted by first creating a simple counter which will count the number of times the basic operation is executed in both algorithms, and then by measuring the execution time of the implemented algorithms. Upon viewing the results ….

# **Description of the Algorithm**

The median in statistics is the value that separates the upper half of a set of data, a population, or a probability distribution, from the lower half or simply the middle value within a data set. This value remains extremely important within statistical analysis and probability theory due to the important of what it can represent for a data set or continuous probability distribution. Therefore, it has become of interest to implement a time efficient algorithm that is capable of locating the median value, even in situations where the data remains unsorted. Thus, an efficient algorithm for locating the median value has been investigated, designed and compared.

## Brute Force Median

The Brute-Force Median algorithm solution, as seen in figure I, operates by systematically checking all possible elements for the solution, as the name implies. After checking each element, the value returned shall be the median value for the input data, which is to be contained within a C-style array or a doubly linked list. This is achieved by first calculating the position of the median, ***k***, if the data set were sorted then by utilising nested ***for*** loops to iterate over each element of the data set and compare it with every other element with ***i*** and ***j*** indexing the outer and inner loop respectively. During this time, two comparisons are conducted, one to check if the current element is larger than what it is being compared to, ***A[j] < A[i]***, if so then the variable ***numsmaller*** is incremented. The other to check if the current element is equal to what it is currently being compared to, ***A[j] = A[i]***, if so then ***numsequal*** is incremented. After the inner loop has finished executing, these two variables indicate the total number of data points smaller than and equal to the current element which can be used to deduce the current elements position index as being in-between the ***numsmaller + 1*** to ***(numsmaller + numequal)***. These values can therefore be compared to ***k*** to determine if the current element is the median, if so it returns a value otherwise it will continue to check the next element in the array with ***numsmaller*** and ***numequal*** are reinitialised to equal zero.

## Partitioning Median

Johnsonbaugh and Schaefer have proposed a version of the median algorithm, as seen in figure II to IV, that is built around the idea that locating the median within an unsorted data set is inherently a sorting problem, or more specifically a selection problem. As the median, as mentioned previously, is the middle value of a discrete data set, we need to only sort the array enough such that we can identify the central value, or the ***kth*** value. This is completed via utilising the underlying principles of the quicksort algorithm, the partitioning of the data set, producing a divide and conquer algorithm.

The implementation of this solution requires three methods, the ***Median*** method which handles the unique case of only one element in the data set existing, where it simply returns the value that element. For other sets of data, which are contained within a C-style array or doubly linked list, calls the recursive method ***Select***, passing the data set, the indexing value for the first element, the middle value of the data set, and the indexing value for the last element in the data set. This Select method recursively calls itself until the partitioning method returns the desired middle value is returned. It should be noted that upon each call, the algorithm is reduced such that the points of interest lay between ***l*** and ***h***. The work of partially sorting the array is completed by the partitioning method, which is the procedure that is heavily utilised in the Quicksort Algorithm. It first selects a pivot as the first element from the portion of the data set in interest, it then swaps all elements, such that values smaller than the pivot lay on the left-hand side and values greater than the pivot lay on the right-hand side of the pivot. The index value of the pivot is then returned to the selection algorithm for it to be determined if the pivot selected is the median.

# **Theoretical Analysis of the Algorithm**

This section of the report details the average case theoretical analysis of each algorithm along with the choice of basic operation and the order of growth.

## The algorithm’s basic operation

Each algorithm’s basic operation will be utilised in the theoretical analysis of the average case efficiency which will be outline below.

### **3.1.1 Brute Force Median Algorithm**

Upon conducting the basic operation of choice was the comparison ***A[j] < A[i]***, and was utilised in the average case efficiency analysis, as seen in figure 1. This operation was selected as the basic operation due to the fact that this comparison has the greatest impact on the execution time of the algorithm, due to the fact that this operation is performed the greatest number of times and being the dominant operation in the algorithm. This comparison is executed within the nested ***for*** loop and is performed upon each iteration of the inner loop.

However, the other comparison present, ***A[j] = A[i]***, ***numsaller < k*** and, ***k ≤ (numsaller + numequal)***, within the algorithm will also affect the execution time by a variable amount. However, for this analysis it was assumed that the ***numsaller < k*** and, ***k ≤ (numsaller + numequal)*** comparison is less dominant and the ***A[j] = A[i]*** comparison being simplified to a two way comparison with ***A[j] < A[i]*** as per Levitin’s simplifying notion … REFERENCE …. when calculating the order of growth. Thus, they were not considered during the average case efficiency analysis, while a more detailed analysis shall take into account these comparisons.

### **3.1.2 Partition Median Algorithm**

When conducting the theoretical analysis of the partitioning median algorithm, the basic operation of interest chosen is the comparison of ***A[j] < pivotval*** contained within the partitioning method, as seen in figure …. This statement was selected due to it being a crucial element in not only the partitioning method, but the median algorithm as well. This is due to the fact that the median methods complexity relies on that of the selection method and in turn the partitioning method, with this comparison being the dominant operation. This operation is executed within a ***for*** loop that iterates over an array between the indexes ***l + 1*** and ***h*** as specified by parameters to the partitioning function.

It should be noted that the swapping of elements completed within the partitioning algorithm will have an impact on the execution time. However, during execution of the algorithm, it is believed that this operation will have negligible impact on its order of growth, as it is assumed that the comparison to provide sufficient estimation of the average case efficiency. Thus, the analysis conducted within this report does not consider the swapping of elements operation.

### **3.1.3 Commonality of Basic Operation**

The basic operation selection for each algorithm was based off what was the dominant operation in each algorithm. While also as these are solutions to a selection problem, the basic operations are chosen to reflect this with the dominant operation being a key comparison. Thus, it can be stated that between the brute-force median algorithm and the partitioning median algorithm, the basic operations of choice reflect their nature of selection problems and represent the dominant operation. Also because of this commonality, the analysis’ derived from these basic operations allows for a valid comparison between the two algorithms.

## Problem Size

The choice of problem size for both algorithms is the input data size as this will affect the number of times the basic operation is executed and in turn the time execution.

## Average Case Efficiency

As both algorithms are capable of exiting early from execution, therefore, there will be a best-, worst- and, average-case efficiency for each algorithm that may be different. With the Brute Force Median being able to exit early from the nested ***for*** loops, and the partitioning median algorithm decreasing the area of inspection by an unknown amount and exiting early from the search. Therefore, a probabilistic distribution needs to be defined such that it represents where the search key is located within the input data.

### **3.2.1 Brute Force Median Algorithm**

First analysing the inner ***for*** loop of the ***BruteForceMedian*** algorithm produces the following complexity:

Now analysing the outer ***for*** loop of this algorithm, it must first be assumed that the median is contained within the input data, ***p = 1***, and that it has equal chance of being with any element, which can be represented as:

Therefore, calculating the expected complexity for the average-case efficiency can be achieved by utilising the expected value formula for discrete probability distributions, where the number of comparisons is between ***1*** × ***n*** and ***n*** × ***n***, due to the face that upon each iteration the inner loop performs the basic operation ***n*** times after each iteration of the outer loop:

With the probability that the median is contained within the input data equal to the average case efficiency of the Brute Force Median algorithm is equal to:

### **3.2.2 Partition Median Algorithm**

First, the ***Partition*** method’s complexity must be analysed which shall be denoted as . The basic operation within this method is executed upon each iteration of the ***for*** loop:

Where indicates the length of the data set being analysed, including the pivot, that the method iterates over. Thus, for simplicity this value shall be represented as , where is the size of the data set within the partitioning method,

This produces an order of growth for the partitioning algorithm of ***n***, for the rest of this analysis, this shall be utilised to represent the complexity of the partitioning method. This is due to the fact that the ***- 1*** component of the complexity is inconsequential to the execution time of the partitioning median algorithm,

Now before analysing the recursive ***Select*** method, of which will be denoted as , some assumptions must first be made, including that upon each iteration the data set size is to decrease by one quarter in size. This was assumed due to the fact that if the data set where to decrease by one half, then the median would be found on the first iteration, or near the first iteration.

To simplify calculations, let

This recursive function can therefore be simplified to,

This function represents the average complexity of the selection method assuming that it does not return early from the function and completes ***k – 1*** recursive calls. Therefore, this will not produce an accurate analysis of the average case efficiency of the selection method. Therefore, it is then assumed that the median can appear in any element and has equal probability to do so,

Where ***p*** is the probability that the median exists within the data set, which will always be equal to one. It should also be noted that after each iteration of the ***for*** loop, the data set that is being iterated over decreases in size by assumedly , therefore, after each iteration it becomes more likely that the median is selected within the array,

Where ***j*** represents the number of recursive calls made by the selection method. Applying the expected value formula, where the basic operation is performed from times to times, the expected complexity can be determined to be,

To simplify this, the summation of geometric progressions is used, as some of these summation elements produce a geometric series,

Now, as , ,

Now investigating the ***Median*** method, it simply calls the select function after handling the unique case of only one element existing within the array. Therefore, it can be stated that the average case efficiency of the Median algorithm is equal to that of the ***Selection*** method,

## Order of Growth

The Brute Force Algorithm initially appears to have an order of growth that is quadratic in nature, of which was proven with formal analysis where the average-case efficiency produced growth. This is due to the nested ***for*** loop seen within figure I along with the fact that there is a need to check every element in the input data. While, Johsonbaugh and Schaefer’s solution to the median problem produced an order of growth of through the formal analysis conducted, and as stated by Levitin through an analogy, and via Berman and Paul whom details an analysis for their version of the algorithm. (REFERENCE)

# **Methodology, Tools and Techniques**

## Programming Environment

The algorithms and experiments were developed using a combination of Java Integrated Development Environments (IDE). IntelliJ IDEA Community (ver. 2017.3.5) was used for the implementation of the brute force median algorithm and Eclipse was used for the implementation of the partition median algorithm. Both IDE’s are completely free and open source development environments.

The experiments were carried out using a custom-built gaming pc running windows 10 x64. The specs for the testing machine are: i3-8100 (3.6Ghz) Processor, 8GB DDR4 RAM, NVidia GeForce GTX1060-6Gb graphics card, 120Gb SSD, 1Tb HDD. The java- integrated random number generator was used to produce the test data for analysis and the built-in Java time module was used for measuring execution times.

Graphing of the experimental results were completed using Matlab. Matlab live scripts are created which reads test data from the .csv files and plots it along with the theoretical growth rate.

## Algorithm Implementation

For the algorithm to function it is required that it is passed an array of integers. A function was built to accommodate this requirement which simply creates an array of set size and fills it with random integer values as seen in Appendix III. The algorithms were developed in separate Java classes to keep the functions separate and organised. The two algorithms seen in Figure I to IV were implemented using the java development kit (ver. 1.8) and can be seen in Appendix I to IV respectively.

## Test Data and Execution of Experiments

To confirm the functionality of the algorithms implemented a test class was built to test the functions against a hard-coded set of integers as seen in Appendix IV & V. The test classes ensure that the functions maintain accurate response to certain cases such as being passed an empty array or an unsorted array.

To count the number of basic operations performed by each algorithm to complete a search, a counter was added to increment on each iteration of the algorithm. At this point another class was developed to test the algorithms and count the average number of basic operations for each array size ranging from 1000 to 100,000.

Measuring the time required to complete an execution of each algorithm was very similar to counting the operations in that a variable had to be declared - prior to executing the algorithm – to store the current time in nanoseconds using the built-in java time module into a long variable. Once the algorithm completes the search the execution time can then be extrapolated by subtracting the current time from the stored start time. This was tested against the same number of arrays and sizes to conclude the average execution times.

# **Experimental Results**

## Functional Testing

Before conducting the empirical analysis of the Brute Force Median and the partitioning Median algorithm, it must first be verified that they are operating as functionally expected. These tests were conducted by utilising Java’s JUnit testing suite, both algorithms where then provided with hard coded values, both representing expected input values, and unusual circumstances. In order to verify that these algorithms where returning the correct value, JUnit’s ***assertEquals*** assertion is utilised.

### **Brute Force Median**

To see the code that performed the functional tests for the Brute Force Algorithm, see Appendix …. It should also be noted that for a given data set that is even, the median value returned to the user is the ‘left’ value of the two middle values.

The first hardcoded array passed into the ***BruteForceMedian*** method is as seen below,

***{ 1, 3, 4, 6, 7 }***

As expected, for this sorted array, the value returned from the method was ***4***, therefore, it can be stated that for sorted data, this algorithm operates as expected. Then to test how the algorithm performs when unsorted data is passed, the same values where passed however, the order of the values have been altered:

***{ 4, 7, 6, 1, 3 }***

As was expected, the value returned from this method was equal to ***4*** and thus, it can be stated that for unsorted expected arrays, the method functionally performs as expected. Another nominal test case was passed during the functional testing was:

***{ -1, -2, 3, 4, 7, 9 }***

As was expected, the ***BruteForceMedian*** method returned the expected value, ***3***, therefore, it can be stated that when the data set contains negative values, and an even number of input elements, the algorithm performs as expected. Finally, the last of the nominal data inputs was passed to the method:

***{ 7, 4, 9, 3, 1, 2 }***

Where the median value returned was as expected and thus it can be stated for nominal inputs to the ***BruteFroceMedian*** method, the functionality of the program is correct. However, there is a need to test the extreme circumstances of the method in order to ensure that if by mistake, unusual sets of data passed to the method, the expected result is still returned. Therefore, two arrays that only contain one element is passed to the method:

***{ 1 } and { 8 }***

And as expected, the median values returned from the method where ***1*** and ***8*** respectively, thus it can be stated that when the ***BruteForceMedian*** method receives data inputs with only one element, then the median value returned as expected. Two other extreme cases where also tested, the first being where the data input elements are all equal and where only three values exist within the data set, with the median being the only unique one:

***{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 } and { 1, 1, 1, 1, 8, 8, 8, 8, 4 }***

As expected, the median values returned were equal to ***1*** and ***4*** respectively, therefore, for input data where all elements are equal and where the median is the only unique element, the method behaves as expected. Finally, a data set containing no elements was passed to the array, under normal circumstances, this behaviour is expected to produce an exception which was as seen. Therefore, it can be stated that the ***BruteForceMedian*** method behaves correctly for the input data utilised above. Also, another test is run to ensure that the basic operation counter is operating as expected. This was accomplished by getting the actual value by executing the algorithm by hand and comparing it to the value returned from the ***BruteForceMedianBasicOperationCounter*** method.

### **5.1.2 Partitioning Median**

For the code that performs the partitioning algorithms functional testing see Appendix …, it should be noted that when an even value of input data is given, this method will return the ‘right’ value of the two middle values.

The first normal data set provided to the Median method is:

***{ 1, 2, 3, 4, 5, 6, 7, 8 }***

Where an expected value of ***5*** was produced thus, it can be stated that for a sorted array, the median value returned was the desired value. It also confirmed that within an even set of numbers, the upper middle value is returned from the median function. Another formal data set implemented is:

***{ 9, 1, 2, 8, 3, 4, 4, 1, 9, 2 }***

The expected value for the median was ***4***, which was returned. Therefore, upon passing this test it can be stated that for unsorted arrays, the median was accurately calculated. Another hardcoded test case was:

***{ -5, 9, 11, -1, -4, 6, 7, 60, 11, 99, 1 }***

This test case checks to determine if the algorithm continues to function as expected when negative values are contained within the data set. As the function produced the expected result of ***7*** then it can be stated that in this circumstance, the algorithm performed as expected. To further investigate the functionality of the algorithm under even and odd sized data sets, one addition value was added to this data set, such that it became:

***{ -5, 9, 11, -1, -4, 6, 7, 60, 11, 99, 1, -99 }***

This formed an even sized data set, where there would be two median values. As this method is to functionally return the upper of the two values, the expected result is ***7***, which was received. Finally, the last hard coded data set is utilised:

***{ 0, 1, 2, 3, 80, 7, 90, 4 }***

This test case was utilised to determine if the median algorithm accurately returned values if there are large variations within the initial data set, of which it performed as expected returning a ***4***. Therefore, it can now be confidently stated that this algorithm is capable of returning the correct median value under normal circumstances.

The next test cases purpose is to determine if the algorithm is to perform as expected under circumstances that may arise due to any errors. This includes passing the median function two data sets that only contained one value:

***{ 1 } and { 5 }***

As expected, these returned the one element that existed within the array, and thus it can be stated that is one element is passed to the median algorithm, it can correctly identify the median. Other cases tested where when only three numbers within the data set existed:

***{ 1, 1, 1, 1, 1, 5, 5, 5, 5, 5, 4 }***

The median function was able to return the expected median value of ***4*** was returned from the function. While also, a data set containing only one value was also tested:

***{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 }***

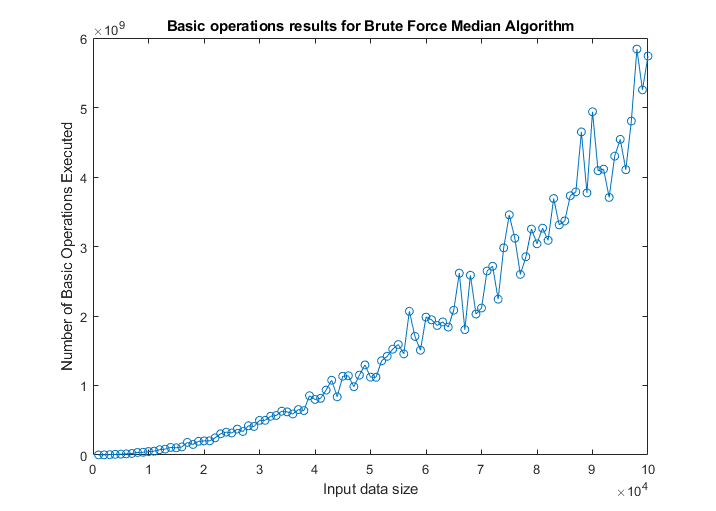
And again, the median value returned was as expected, thus it can be stated that for these circumstances the median function will be capable of producing correct results. It should also be stated that for the purposes of testing, a data set of zero elements was passed to the median, where it was expected that an exception was thrown, as was the case. Also, to ensure that the basic operation counter is operating correctly, a small test is run where it compares the returned basic operation counter with the actual as determined by analysing the algorithm.

## Average-Case Number of Basic Operations

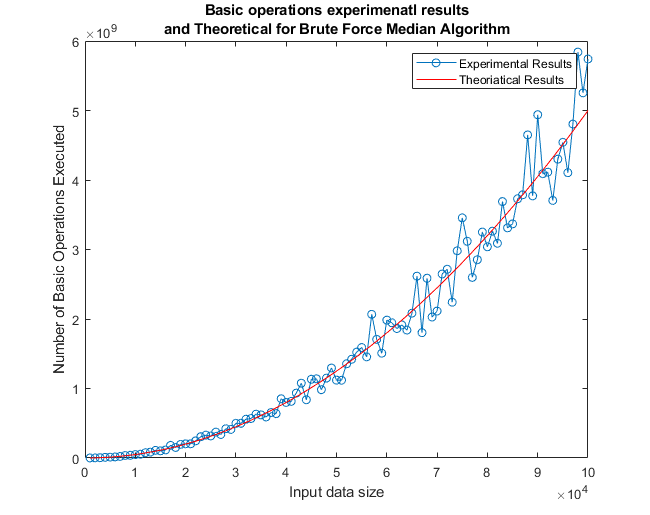
In this section the experimental results will be compared via graph with their theoretical case efficiencies in relation to number of basic operations.

**5.2.1 Brute Force Algorithm**

Figure V displays the experimental results collected from the tests on the brute force algorithm counting the average number of basic operations required to complete a search (Blue) and against its theoretical rate of growth (Red) in figure VI. As seen in the graph the experimental results match that of the rate of growth and time allowing would have had a much closer relation if more tests had been run to give a more defined median value for each set size.



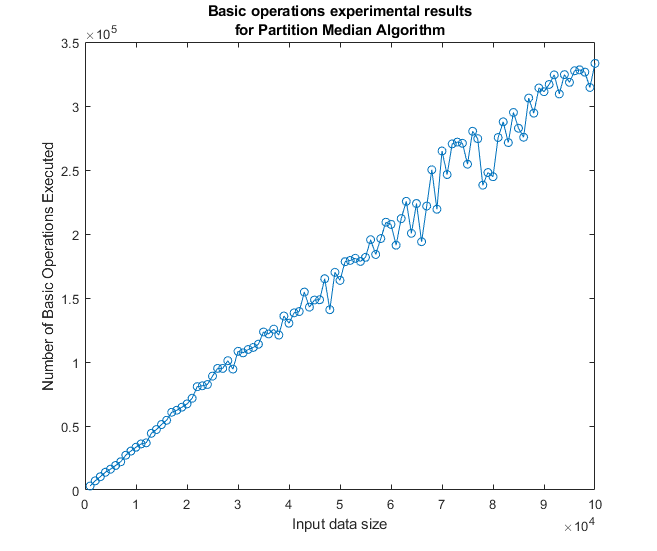
## Figure V: Brute Force Algorithm Basic Operations



## Figure VI: Brute Force Algorithm Basic Operations & Theoretical Results

**5.2.2 Partition Algorithm**

Figure VII depicts the experimental results collected from the tests on the partition algorithm counting the average number of basic operations required to complete a search (Blue) and against its theoretical rate of growth (Red) in figure VIII. The graph shows a similarity between the average basic operations required and the theoretical rate of growth. The variance between the two is derived from assumption in the theoretical growth that the size decreases by exactly half for each iteration when it doesn’t. Aside from this factor the findings are quite accurate and increasing the number of tests would have given a more accurate depiction of the median.



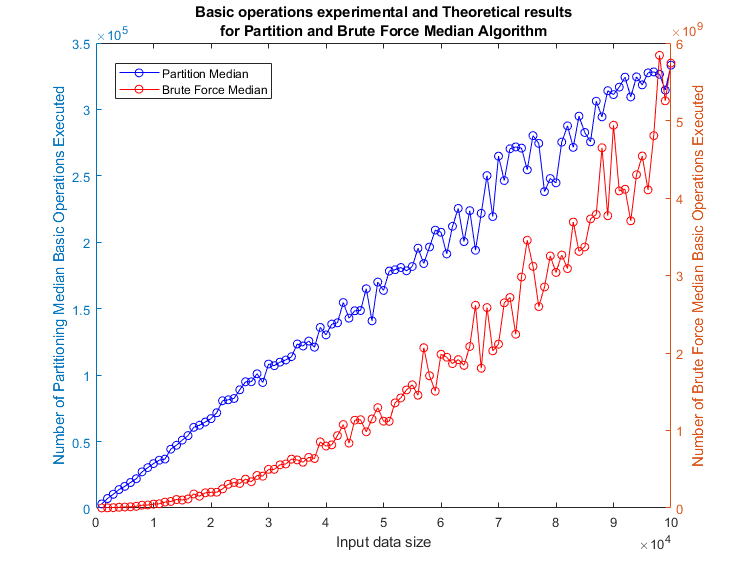
## Figure VII: Partition Algorithm Basic Operations

## 

## Figure VIII: Partition Algorithm Basic Operations & Theoretical Results

**5.2.3 Basic Operation Algorithm Comparison**

Figure IX displays the comparison of experimental data pertaining to the number of operations between the brute force and partition algorithm. The difference in number of basic operations was so large that the graph required 2 separate y-axis values to accurately display the information. Both algorithms are increasing incrementally with the array size however the brute force algorithm increments at a much faster rate and requires up to by the end of the tests compared to the partitions .



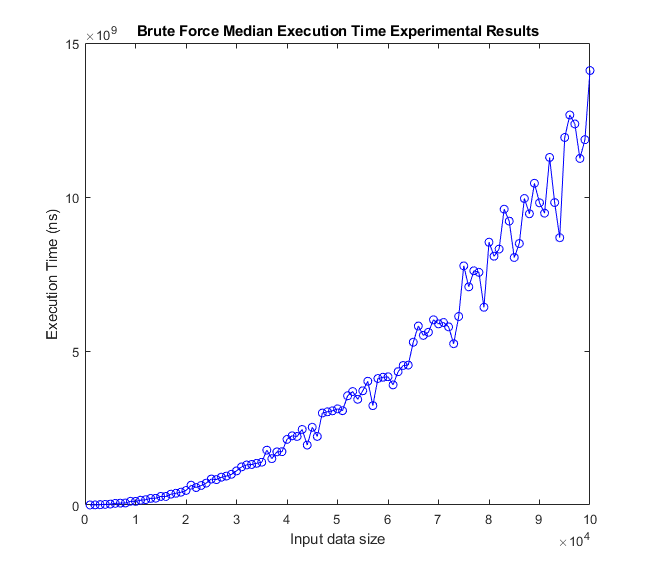
## Figure IX: Brute Force Vs Partition (Basic Operations)

## Average-Case Execution Time

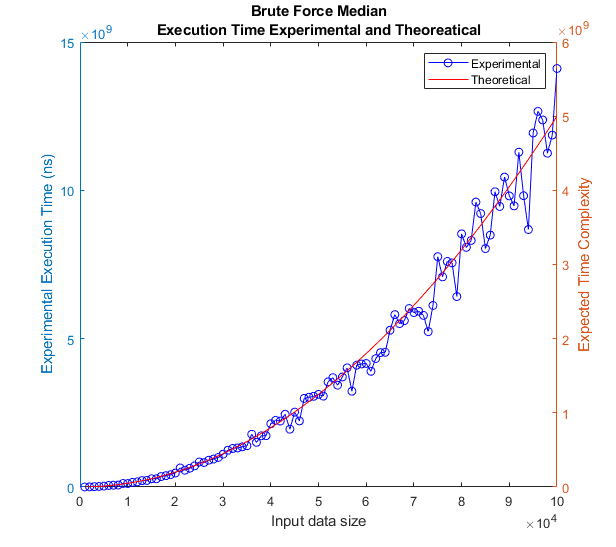
In this section the experimental results will be compared via graph with their theoretical case efficiencies in relation to execution time.

**5.3.1 Brute Force Algorithm**

Figure X depicts the experimental results collected from the tests on the brute force algorithm counting the average execution time required to complete a search (Blue) and its theoretical rate of growth (Red) in figure XI. This graph much like the brute force basic operation and could have benefited by increasing the number of tests to improve the median however it is clear to see that the experimental data very accurately reflects the theoretical execution time growth rate.

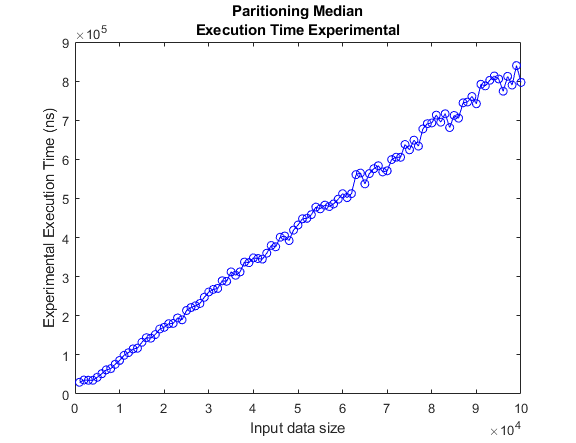
****

## Figure X: Brute Force Median Execution Time

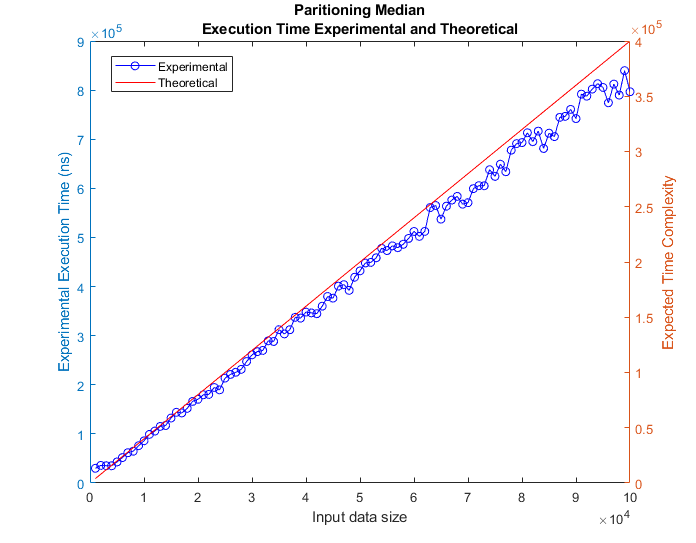
****

## Figure XI: Brute Force Median Execution Time & Theoretical Results

**5.3.2 Partition Algorithm**

Figure XII depicts the experimental results collected from the tests on the partition algorithm to collect the average execution time required to complete a search (Blue) and its theoretical rate of growth (Red) in figure XIII. The execution time shown for the partition algorithm in the graph follows its theoretical results despite a slight drop when reaching much larger array sizes because of the same false assumption stated for the partition basic operation.

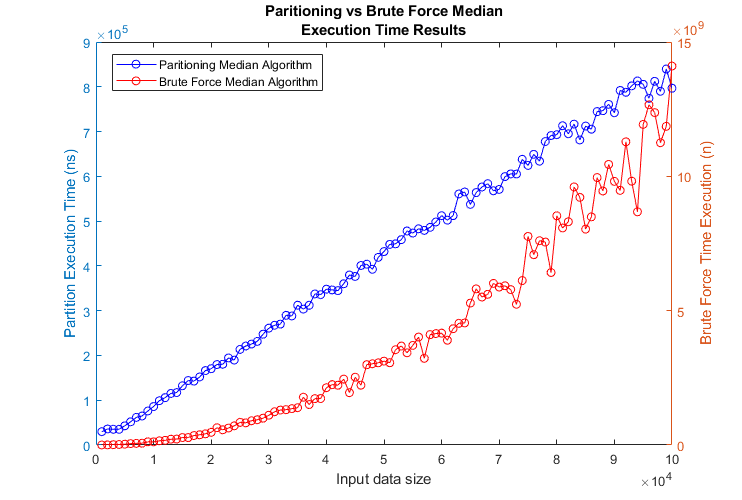
## Figure XII: Partition Median Algorithm Execution Time

****

## Figure XIII: Partition Median Algorithm Execution Time & Theoretical Results

**5.3.3 Execution Time Algorithm Comparison**

Lastly Figure XIV displays the comparison between the brute force and partition algorithm’s pertaining to their execution times. Once again, the difference in performance between the two algorithms is quite clear and the brute force algorithm at an input size of reached up to nanoseconds compared to the partition algorithm with the same input size only reaching up to nanoseconds. In addition to this the curvature of the brute force algorithm indicates a quadratic increase compared to the partition algorithm’s much more linear increase.



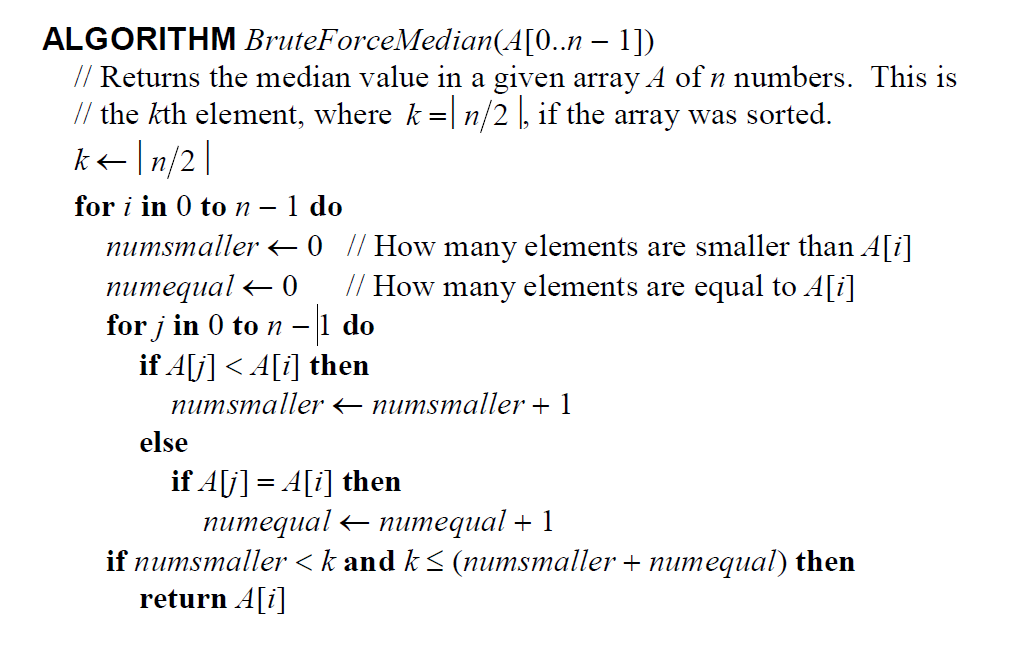
## Figure XIV: Brute Force Vs Partition (Execution Time)

# **References**

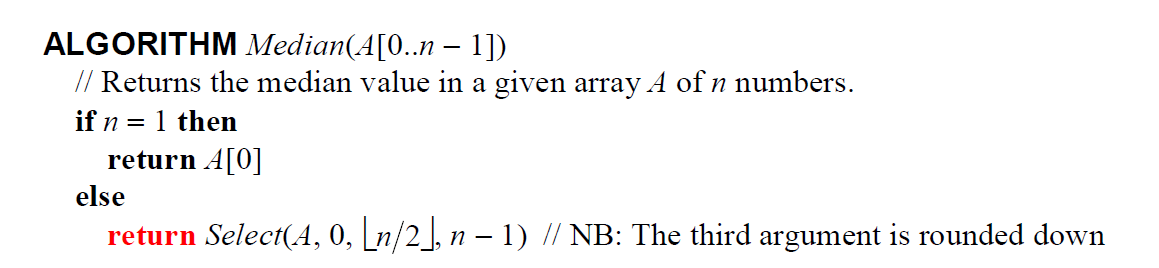
1. <https://en.wikipedia.org/wiki/Brute-force_search>
2. <https://au.mathworks.com/videos/writing-a-matlab-program-69023.html>

# **Figures**

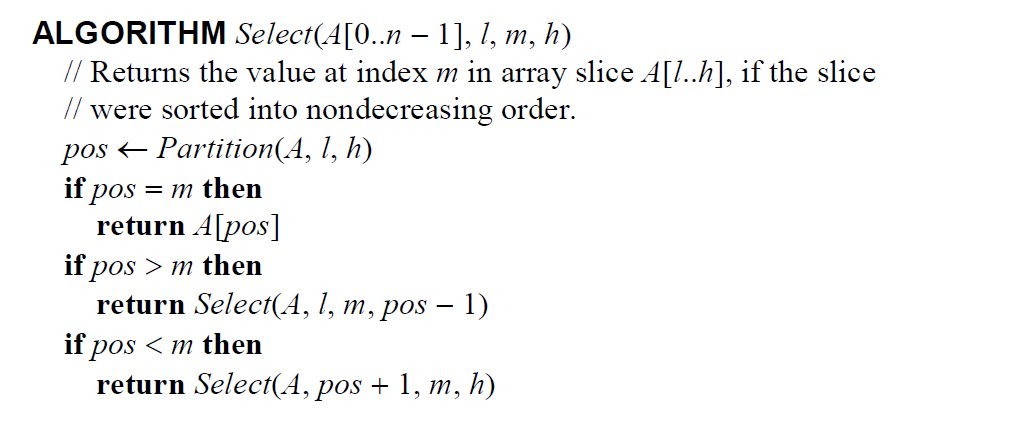
## Figure I: Brute Force Median Algorithm



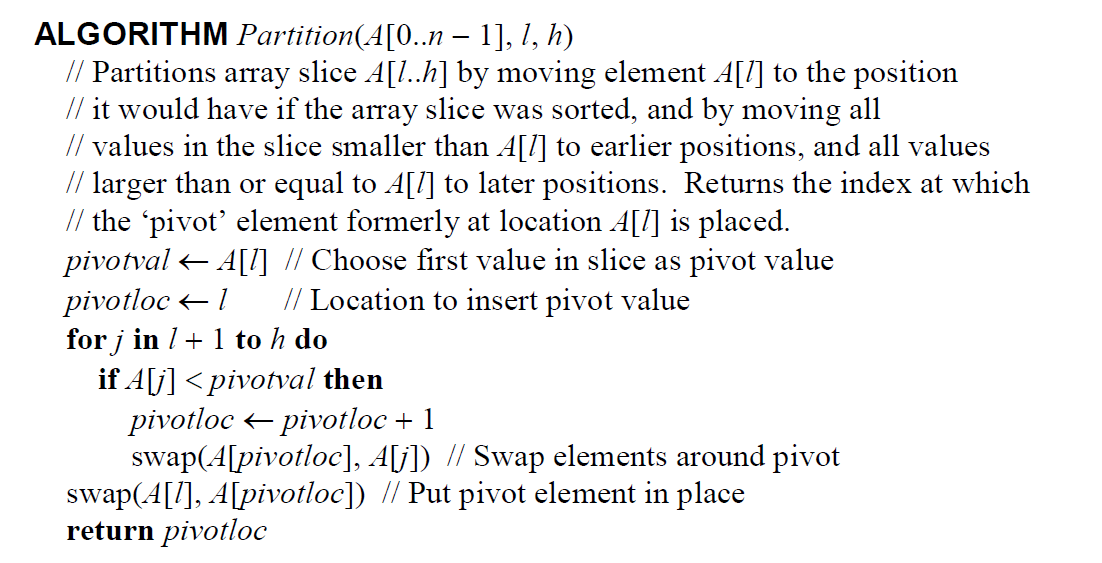
## Figure II: Partition Median Algorithm (Base Function)



## Figure III: Partition Median Algorithm (Select Function)



## Figure IV: Partition Median Algorithm (Partition Method)



# **Appendices**

## Appendix I: Code for Brute Force Median Algorithm Implementation

public static int BruteForceMedian(int[] array) throws Exception  
{  
 int k = (int)Math.*ceil*((array.length+1)/2);  
   
 for (int i = 0; i < array.length; i++){  
   
 int smaller = 0;  
 int equal = 0;  
   
 for (int j = 0; j < array.length; j++)  
 {  
 if (array[j] < array[i])  
 {  
 smaller += 1;  
 }  
 else if (array[j] == array[i])  
 {  
 equal += 1;  
 }  
 }  
 if (smaller < k && k <= (smaller + equal))  
 {  
 return array[i];  
 }  
 }  
 return k;  
}

The algorithm for brute forcing the median is located in the BruteForceAlgorithm class and was developed from the BruteForceMedian Algorithm seen in Figure. 1. The algorithm functions by setting an integer k to the array length / 2 and rounded up to the nearest whole number. The algorithm features a double for loop that checks each value in the array against every other value in the array and increments the smaller, and equal variables dependant on if the value being checked is smaller than or equal to respectively. Once a value has been checked against all other values the algorithm checks to see if the number of smaller values is less than the middle value k and if k is less than or equal to the sum of smaller values the number of equal values. If this is the case, the median has been found and will be returned, else the next value in the array will be checked against all other values and so on. If for some reason no median is found, then k is returned.

## Appendix II: Code for Partitioning Median Algorithm Implementation

The algorithm median whose pseudo code can be viewed in figure …, as denoted by partitioning algorithm in this report to avoid confusion, implements three methods, first the Median method of which the user calls, the Select method, and the Partitioning method. All of which is contained within the PartitionAlgorithm Class.

public static int Median(int[] array)  
{  
 if ( array.length == 1 )  
 return array[0];  
 else  
 return *Select*(array, 0, (int)Math.*floor*(array.length / 2), array.length - 1);  
}

As seen above, the Median algorithm is quite simple and only handles two cases for an input array. The first being if the initial input array contains only one element, of which the element is returned as the result of the algorithm as seen on line …. The second case is then for any other array it then calls the Select method, of which the median of the input data is then located and returned to Median, the parameters passed to this method is the input array data, the location of the start of the data set, the median position if the data were sorted, and the position of the highest data point, as seen on line …..

public static int Select(int[] array, int low, int mid, int high)  
{  
 int pos = *Partition*( array, low, high );  
   
 if ( pos == mid )  
 return array[pos];  
 if ( pos > mid )  
 return *Select*( array, low, mid, pos - 1 );  
 if ( pos < mid )  
 return *Select*( array, pos + 1, mid, high );  
   
 return 0;  
}

The ***Select*** method operates by first partitioning the array over the period being investigated, where the ***low*** parameter represents the lowest index, the ***high*** parameter represents the index in the array and, the ***mid*** parameter represents the median position if the array were sorted, as seen on line ….. After the Partition algorithm returns the index position of the pivot value that was selected of which would have been sorted into the correct position in the array, this index is then compared with ***mid***, as seen on lines … to …. As after the ***Partition*** method, all values less than the pivot shall have a lower index and values greater shall have greater index, if the pivot point selected was not the median value, line …, then the Select method is recursively called on either the left or right sides of the array – depending on if the pivot position is less than or greater than the ***mid*** value respectively, as seen on lines ….

public static int Partition(int[] array, int low, int high) throws Exception  
{  
 int pivotval = array[low];  
 int pivotloc = low;  
   
 for ( int j = low + 1; j <= high; j++ )  
 {   
 if ( array[j] < pivotval )  
 {  
 pivotloc++;  
   
 // swap elements  
 int tempVal = array[pivotloc];  
 array[pivotloc] = array[j];  
 array[j] = tempVal;  
 }  
 }  
   
 int tempVal = array[low];  
 array[low] = array[pivotloc]; // swap elements around pivot  
 array[pivotloc] = tempVal; // put pivot element in place  
   
 return pivotloc;  
}

The ***Partition*** method operates by selecting the first element in the array as the pivot, ***pivotval***, and setting an index to point to where the pivot value will sit in the array after the partitioning method is complete (***pivotloc***), which is initially set to ***low***. as seen on line …. It then enters a ***for*** loop to iterate over the values in the array excluding the pivot value (from ***low + 1*** to ***high***) as seen on line …. Upon each iteration of the for loop, the element that ***j*** is currently pointing to is compared with the pivot value, if the ***pivotval*** is greater than the current element, then ***pivotloc*** is increamented by one to represent the new position of where the pivot will be located and element ***j*** is swapped with the element in the ***pivotloc*** position as seen in lines … to …. After iterating through each value in the array, the pivot is then placed in the correct position of array by swapping it with the element currently in the ***pivotloc*** position with this index being returned as the result, as seen in lines … to ....

## Appendix III: Code for Generating Random Test Data

public static int[] generateRandomArray( int size )  
{  
 Random randomNumberGenerator = new Random();  
  
 int[] randArray = new int[size];  
  
 for ( int i = 0; i < size; i++ )  
 {  
 randArray[i] = randomNumberGenerator.nextInt();  
 }  
  
 return randArray;  
}

Both algorithms are dependent on there being an array to be operated on therefor a function for generating random test data was developed in the Main class. The random array generator must be sent an integer ‘size’ to function. The function created a built-in random number generator and a new integer array of set size. The method then iterates through the new array and fills it with random numbers using the generator. The array is then returned.

## Appendix IV: Code for Testing Brute Force Median Functionality

@Test  
public void testBasicCase() throws Exception{  
 *assertEquals*(4, BruteForceAlgorithm.*BruteForceMedian*(new int[]{1, 3, 4, 6, 7}));  
 *assertEquals*(4, BruteForceAlgorithm.*BruteForceMedian*(new int[]{4, 7, 6, 1, 3}));  
 *assertEquals*(3, BruteForceAlgorithm.*BruteForceMedian*(new int[]{-1, -2, 3, 4, 7, 9}));  
 *assertEquals*(3, BruteForceAlgorithm.*BruteForceMedian*(new int[]{7, 4, 9, 3, 1, 2}));  
}  
  
@Test  
public void testEdgeCases() throws Exception{  
 *assertEquals*(1, BruteForceAlgorithm.*BruteForceMedian*(new int[]{1}));  
 *assertEquals*(8, BruteForceAlgorithm.*BruteForceMedian*(new int[]{8}));  
 *assertEquals*(1, BruteForceAlgorithm.*BruteForceMedian*(new int[]{1, 1, 1, 1, 1, 1, 1, 1, 1, 1}));  
 *assertEquals*(4, BruteForceAlgorithm.*BruteForceMedian*(new int[]{1, 1, 1, 1, 8, 8, 8, 8, 4}));  
}  
  
@Test (expected = Exception.class)  
public void testEmptyCase() throws Exception{  
 BruteForceAlgorithm.*BruteForceMedian*(new int[]{});  
}  
  
@Test  
public void testOperationCount() throws Exception{  
 BruteForceAlgorithm.*operCounter* = 0;  
 BruteForceAlgorithm.*BruteForceMedianBasicCounter*(new int[]{1, 2, 3, 4, 5});  
 *assertEquals*(15, BruteForceAlgorithm.*operCounter*);  
}  
  
@Test  
public void testAverageCount() throws Exception{  
 BruteForceAlgorithm.*operCounter* = 0;  
 int[][] arrays = new int[][]{new int[]{1, 3, 4, 6, 7}, new int[]{1, 3, 4, 6}, new int[]{1, 3, 4, 4, 7, 9}, new int[]{0, 0, 0, 0, 0, 0, 0}};  
 long[] basicOperationCounter = new long[arrays.length];  
 int index = 0;  
 for (int [] arr : arrays){  
 BruteForceAlgorithm.*BruteForceMedianBasicCounter*(arr);  
 basicOperationCounter[index] = BruteForceAlgorithm.*operCounter*;  
 index++;  
 }  
 *assertEquals*(31, Main.*calculateAverage*(basicOperationCounter));  
}

Test cases for the brute force algorithm were developed in the TestBruteForceAlgorithm class using JUnit to test basic cases, edge cases, operation count, average operation count, and exceptions. The basic cases are fairly straight forward and just test that the correct values are returned when sent an array of set values. The edge cases run the same tests as the basic cases except the arrays data is either a single value array or an unordered array. The edge cases ensure that the algorithm accurately functions with odd array values. The exception test simply sends the algorithm an empty set and expects to see an exception thrown from the function. Operation count and average count tests also ensures that the algorithm functions correctly when counting operations and averaging them via the assert equals JUnit function on a fixed set of array values.

## Appendix V: Code for Testing Partitioning Median Functionality

Before executing any empirical analysis tests, the functionality of the implementation must be verified, this is completed through a series of tests via utilising Java’s JUnits testing framework of which is contained with the TestPartitionMedian class.

@Test  
public void testSimpleCases()   
{  
 *assertEquals*( 5, PartitionAlgorithm.*Median*( new int[] { 1, 2, 3, 4, 5, 6, 7, 8 } ) );  
 *assertEquals*( 4, PartitionAlgorithm.*Median*( new int[] { 9, 1, 2, 8, 3, 4, 4, 1, 9, 2 } ) );  
 *assertEquals*( 7, PartitionAlgorithm.*Median*( new int[] { -5, 9, 11, -1, -4, 6, 7, 60, 11, 99, 1 } ) );  
 *assertEquals*( 7, PartitionAlgorithm.*Median*( new int[] { -5, 9, 11, -1, -4, 6, 7, 60, 11, 99, 1, -99 } ) );  
 *assertEquals*( 4, PartitionAlgorithm.*Median*( new int[] { 0, 1, 2, 3, 80, 7, 90, 4 } ) );  
}

The above test, which checks the functionality under nominal conditions, utilises JUnits ***assertEquals*** method to compare the expect values, as seen in argument one, with the actual value returned in argument two. As seen on lines … to … hard coded arrays are passed to the ***assertEquals*** from the ***Median*** method and compared with the known median for the array.

@Test  
public void testEdgeCases()  
{  
 *assertEquals*( 1, PartitionAlgorithm.*Median*( new int[] { 1 } ) );  
 *assertEquals*( 5, PartitionAlgorithm.*Median*( new int[] { 5 } ) );  
   
 *assertEquals*( 4, PartitionAlgorithm.*Median*( new int[] { 1, 1, 1, 1, 1, 5, 5, 5, 5, 5, 4 } ) );  
 *assertEquals*( 1, PartitionAlgorithm.*Median*( new int[] { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 } ) );  
}

This next test method, as seen above, then aims to test more unusual input arrays that may arise during operation. As seen on lines … to … this includes arrays where only one element is passed, and arrays where the elements are not unique or the situation where only the median is unique.

@Test (expected = Exception.class)  
public void testEmptyArray() throws Exception  
{  
 PartitionAlgorithm.*Median*( new int[] {} );  
}

The above test case is to check the functionality of the ***Median*** algorithm in the event that an empty array is passed. It is expected that in this case, an exception will be thrown, as indicated by the ***expected = Exception.class*** annotation attached to ***@Test*** on line …, due to an index out of bounds exception that will be in counted.

@Test  
public void testOperationCount(){  
 PartitionAlgorithm.*MedianBasicOperationCount*(new int[]{1, 2, 3, 4, 5});  
 *assertEquals*(9, PartitionAlgorithm.*basicCounter*);  
}

This final test ensures that when counting the basic operations, the output returned from the basic counter variable is what is expected. This is to ensure that when performing the basic operations counter test, the data returned is not skewed by the incorrect implementation of the basic operations counter.

## Appendix VI: Code for Basic Operation Count for Brute Force Median

public static long *operCounter* = 0;  
public static int BruteForceMedianBasicCounter(int[] array) throws Exception  
{  
 int k = (int)Math.*ceil*((array.length+1)/2);  
   
 for (int i = 0; i < array.length; i++)  
 {  
 int smaller = 0;  
 int equal = 0;  
   
 for (int j = 0; j < array.length; j++)  
 {  
 *operCounter*++;  
   
 if (array[j] < array[i])  
 {  
 smaller += 1;  
 }  
 else if (array[j] == array[i])  
 {  
 equal += 1;  
 }  
 }  
 If (smaller < k && k <= (smaller + equal))  
 {  
 return array[i];  
 }  
 }  
 return k;  
}

The code for counting the basic operations of the brute force median algorithm is split into two separate parts. The first part is located in the BruteForceAlgorithm class and is identical to the brute force algorithm except for a counter of type long and an incrementor located in the nested for loop of the algorithm. See next page for the second part of the code.

public static void bruteForceBasicOperationCounterTests() throws Exception   
 {  
 int numTests = 40;  
 int numArraysTested = 100;  
   
 int increamentSize = 1000;  
 int size = 1000;  
  
 int[] sizeOfArray = new int[numArraysTested];  
 long[] basicOperationCounter = new long[numArraysTested];  
  
 FileWriter fl = new FileWriter( "bruteForceBasicOperationsCounterTest.csv" );  
 // fl.write("Array Size, Basic Operations\n");  
   
 // Initialise execution time counter  
 for ( int i = 0; i < numArraysTested; i++ )  
 basicOperationCounter[i] = 0;  
  
 for (int j = 0; j < numArraysTested; j++)  
 {  
 for ( int i = 0; i < numTests; i++ )  
 {  
 System.*out*.println( "starting array size " + size + " \ttest " + (i + 1) );  
  
 int[] randArray = *generateRandomArray*( size );  
  
 BruteForceAlgorithm.*operCounter* = 0;  
  
 BruteForceAlgorithm.*BruteForceMedianBasicCounter*( randArray );  
  
 // save data  
 sizeOfArray[j] = size;  
 basicOperationCounter[j] += PartitionAlgorithm.*basicCounter*;  
  
 System.*out*.println( "Basic operations performed: " + PartitionAlgorithm.*basicCounter* + "\n");  
 }  
   
 // Calculate average  
 basicOperationCounter[j] /= numTests;   
   
 // Write data to file  
 fl.write( sizeOfArray[j] + "," );  
 fl.write( basicOperationCounter[j] + "\n");  
   
 size += increamentSize;  
 }  
   
 fl.close();  
}

The second part of the code is located in the Main class and simple interacts with the counter located inside the algorithm. The code runs the allotted tests on the algorithm while storing the counter at the end of each execution. These counters are then used to calculate the average, stored in a .csv file and displayed on a graph.

## Appendix VII: Code for Basic Operation Count for Partitioning Median

In order to test the number of basic operations executed during operation, the partitioning median implementation must be slightly altered with a static counter to increment each time the basic operation is executed. As seen on line …, a static long variable named ***basicCounter*** was inserted into the ***Partition*** method, such that each time the comparison is performed, the ***basicCounter*** variable is incremented. It should also be noted that copies of the ***Select*** and ***Median*** methods where created named ***SelectBasicOperationsCount*** and ***MedianBasicOperationsCount*** where created such that this ***PartitionBasicOperationsCount*** method is called during testing of which is contained in ***PartitionAlgorithm*** class.

public static long *basicCounter* = 0;

public static int PartitionBasicOperationsCount(int[] array, int low, int high) throws Exception  
 {  
 int pivotval = array[low];  
 int pivotloc = low;  
   
 for ( int j = low + 1; j <= high; j++ )  
 {  
  
 *basicCounter*++;  
   
 if ( array[j] < pivotval )  
 {  
 pivotloc++;  
   
 // swap elements  
 int tempVal = array[pivotloc];  
 array[pivotloc] = array[j];  
 array[j] = tempVal;  
 }  
 }  
   
int tempVal = array[low];  
array[low] = array[pivotloc]; // swap elements around pivot  
array[pivotloc] = tempVal; // put pivot element in place  
   
 return pivotloc;  
 }

The code on the following page indicates the implementation of details for counting the number of basic operations executed of which is contain within the ***Main*** class. As seen on line … an array containing long integer data points is initialised to equal zero, this array is to contain the basic operations average calculated. The program then enters two nested ***for*** loops, the outer loop controlling the size of the arrays being tested, and the second controlling the total number of tests. Upon entering the inner loop, the basic operation counter is initialised to be equal to zero, and generates an array containing random elements. This is then passed to the ***MedianBasicOperationCount*** method, upon finishing execution, the basic operation is then adding to what is stored in the array ***basicOperationCounter***. This inner loop continuously executes until it has done the required number of tests for the array of that size. The average value for then calculated before continuing on to the next set of tests for a larger array size. Upon executing this test for the entire range of array sizes, the average values executed are then saved into a simple CSV format by utilising Javas FileWriter class, and …. – something about JFree

public static void partitionBasicCounterTest() throws Exception   
{  
 int numTests = 40;  
 int numArraysTested = 100;  
   
 int increamentSize = 1000;  
 int size = 1000;  
  
 int[] sizeOfArray = new int[numArraysTested];  
 long[] basicOperationCounter = new long[numArraysTested];  
  
 FileWriter fl = new FileWriter( "partitionBasicOperationsCounterTest.csv" );  
   
 // Initialise execution time counter  
 for ( int i = 0; i < numArraysTested; i++ )  
 basicOperationCounter[i] = 0;  
  
 for (int j = 0; j < numArraysTested; j++)  
 {  
 for ( int i = 0; i < numTests; i++ )  
 {  
 System.*out*.println( "starting array size " + size + " \ttest " + (i + 1) );  
  
 int[] randArray = *generateRandomArray*( size );  
  
 PartitionAlgorithm.*basicCounter* = 0;  
  
 PartitionAlgorithm.*MedianBasicOperationCount*( randArray );  
  
 // save data  
 sizeOfArray[j] = size;  
 basicOperationCounter[j] += PartitionAlgorithm.*basicCounter*;  
  
 System.*out*.println( "Basic operations performed: " + PartitionAlgorithm.*basicCounter* + "\n");  
 }  
   
 // Calculate average  
 basicOperationCounter[j] /= numTests;   
   
 // Write data to file  
 fl.write( sizeOfArray[j] + "," );  
 fl.write( basicOperationCounter[j] + "\n");  
   
 size += increamentSize;  
 }  
   
 fl.close();  
}

## Appendix VIII: Code for Time Execution Test for Brute Force Median

public static void bruteForceTimeExecutionTests() throws Exception   
 {  
 ...  
  
 int[] sizeOfArray;  
 long[] executionTimeCounter;  
  
 FileWriter fl = new FileWriter( "bruteForceExecutionTimeTest.csv" );  
 // fl.write("Array Size, Execution Time(ms)\n");  
   
 sizeOfArray = new int[numArraysTested];  
 executionTimeCounter = new long[numArraysTested];  
   
 // Initialise execution time counter  
 for ( int i = 0; i < numArraysTested; i++ )  
 executionTimeCounter[i] = 0;  
  
 for (int j = 0; j < numArraysTested; j++ )  
 {  
 for ( int i = 0; i < numTests; i++ )  
 {  
 System.*out*.println("starting array size " + size + " \ttest " + (i + 1));  
  
 int[] randArray = *generateRandomArray*( size );  
  
 startTime = System.*nanoTime*();  
 BruteForceAlgorithm.*BruteForceMedian*( randArray );  
 long duration = System.*nanoTime*() - startTime;  
   
 // save data  
 executionTimeCounter[j] += duration;  
 sizeOfArray[j] = size;  
  
 System.*out*.println( "Execution time: " + duration + "ns\n");  
 }  
   
 // calculate Average  
 executionTimeCounter[j] /= numTests;   
   
 // Write to CSV  
 fl.write( sizeOfArray[j] + "," );  
 fl.write(executionTimeCounter[j] + "\n");  
   
 size += increamentSize;  
 }  
   
 fl.close();  
}

The code used to calculate the execution time for the brute force algorithm functions using the built-in java time module. The allotted tests for the algorithm are run and prior to each execution of the algorithm the time is stored and once the execution is complete the execution time is calculated by subtracting the start time from the current time (in nanoseconds). The execution time is stored for each test of an array size and then added and divided to get an average.

## Appendix IX: Code for Time Execution Test for Partitioning Median

In order to record the execution time of the algorithm, of which is executed in the Main class, Java’s System clock, to nanosecond precision, is utilised to record the execution times. The method of then recording the average execution time is similar to the method used for the basic operations counter, however, on lines … to … an arbitrary value from the system clock is recorded before executing the ***Median*** method, and after, these values are then used to calculate the execution time and is thus recorded.

public static void partitionExecutionTimeTest() throws Exception  
{  
 int numTests = 40;  
 int numArraysTested = 100;  
   
 int increamentSize = 1000;  
 int size = 1000;  
   
 long startTime;  
  
 int[] sizeOfArray;  
 long[] executionTimeCounter;  
  
 FileWriter fl = new FileWriter( "partitionExecutionTimeTest.csv" );  
   
 sizeOfArray = new int[numArraysTested];  
 executionTimeCounter = new long[numArraysTested];  
   
 // Initialise execution time counter  
 for ( int i = 0; i < numArraysTested; i++ )  
 executionTimeCounter[i] = 0;  
  
 for (int j = 0; j < numArraysTested; j++ )  
 {  
 for ( int i = 0; i < numTests; i++ )  
 {  
 System.*out*.println( "starting array size " + size + " \ttest " + (i + 1) );  
  
 int[] randArray = *generateRandomArray*( size );  
  
 startTime = System.*nanoTime*();  
 PartitionAlgorithm.*Median*( randArray );  
 long duration = System.*nanoTime*() - startTime;  
   
 // save data  
 executionTimeCounter[j] += duration;  
 sizeOfArray[j] = size;  
  
 System.*out*.println( "Execution time: " + duration + "ns\n");  
 }  
   
 // calculate Average  
 executionTimeCounter[j] /= numTests;   
   
 // Write to CSV  
 fl.write( sizeOfArray[j] + "," );  
 fl.write(executionTimeCounter[j] + "\n");

size += increamentSize;  
 }  
 fl.close();  
}