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 algorithms to locate the median within a discrete list of items, 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 implemented within Java, utilising Oracles JDK (version 1.8). The empirical analysis will be conducted by creating a simple counter which will record the number of times the basic operation is executed in both algorithms with random data as input, and then by measuring the execution time of the implemented algorithms. Upon viewing the results, it was confirmed that the Brute Force Median algorithm was extremely inefficient with an order of growth of . While, the median algorithm built upon the fundamentals of the quicksort algorithm proves to be significantly more efficient and usable within practical applications experienced an order of growth of .

# **Description of the Algorithm**

The median in statistics is the value that separates the upper half of a set of discrete data, a population, or a probability distribution, from the lower half, or simply the middle value within a sorted set of data. This value remains extremely important within statistical analysis and probability theory due to what it can represent for discrete data or continuous probability distribution [1]. Therefore, there is a need for the median to be calculated on large unsorted sets of discrete data efficiently. Thus, two algorithm’s that implement the functionality of locating the median value in unsorted discrete data differently have been analysed and compared.

## Brute Force Median

The Brute-Force Median algorithm solution, as seen in figure 1, operates by systematically checking all possible elements for the solution, as the name implies [4] [6]. It should be noted that this implementation also assumes that the input data is contained within a C-style array or doubly linked list. This is achieved by first calculating the position of the median, ***k***, if the input data was sorted, then by utilising two nested ***for*** loops to iterate over each element and comparing it with every other element with ***i*** and ***j*** indexing the outer and inner loop respectively. During each iteration of the inner loop, 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 this is true then the variable ***numsmaller*** is incremented. The other is 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 execution, these two variables indicate the total number of data points smaller than and equal to the current element, ***i***, which can be used to deduce its 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 on to the next iteration of the outer loop to perform the same operations on the next element, with ***numsmaller*** and ***numequal*** are reinitialised to equal zero. It should be noted, that when an even data input is passed to this algorithm the median value that will be returned is the right of the midpoint. This algorithm was implanted by utilising a Java Development Kit and implementing it in a Java program of which this code can be viewed in Appendix I.

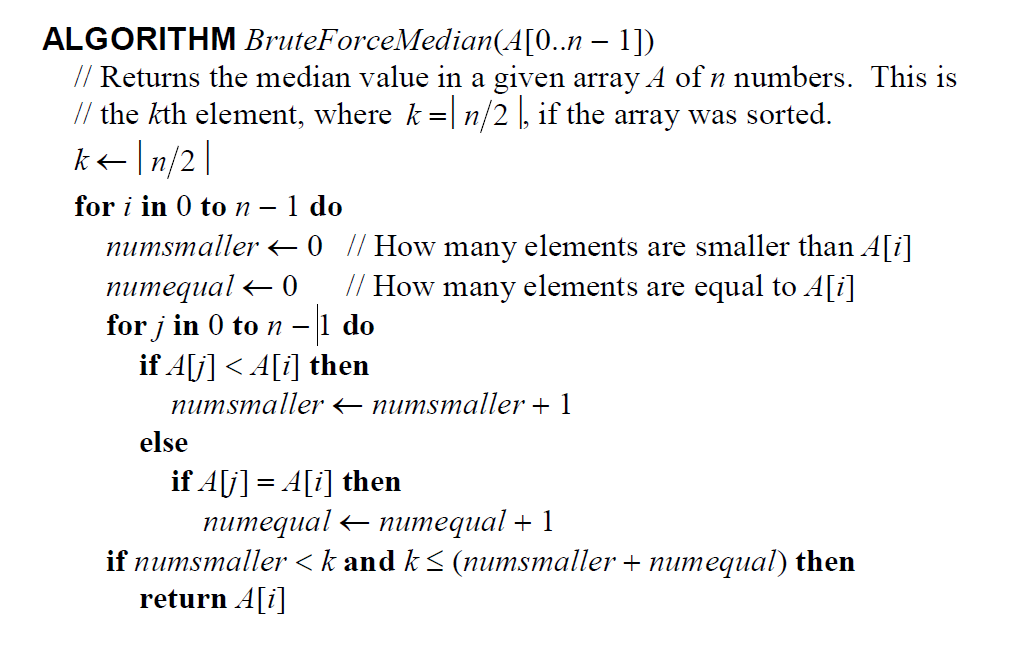


Figure 1: Brute Force Median Pseudocode

## Partitioning Median

Johnsonbaugh and Schaefer have proposed a version of the median algorithm, as seen in figure 2 to 4, 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. Thus, they have proposed a solution that implements a specific version of the quickselect algorithm [2]. As the median, as mentioned previously, is the middle value of a discrete data points, 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 operation, producing a divide and conquer algorithm [2] [3].

The implementation of this solution requires three methods, the ***Median*** method which handles the unique case of only one element in the input data points existing, where it simply returns the value that element. For other sets of data, which are assumed to be contained within a C-style array or doubly linked list, calls the recursive method ***Select***, passing through the input data points, the indexing value for the first element, the middle and, end element. This ***Select*** method recursively calls itself until the ***Partition*** method returns the desired middle value. It should be noted that upon each call, the algorithm is reduced such that the points of interest lay between ***l*** and ***h*** of which is known to contain the median after partially sorting the data. The work of partially sorting the array is completed by the ***Partition*** 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 ***Select*** method for it to be determined if the pivot selected is the median. It should also be noted that for even input data sizes, the algorithm will return the left-hand size of the two middle values of this which will differ from the Brute Force Median however, for large data inputs this functional difference is considered insignificant. This program was also implanted via a Java program, of which the code can be seen in Appendix II.

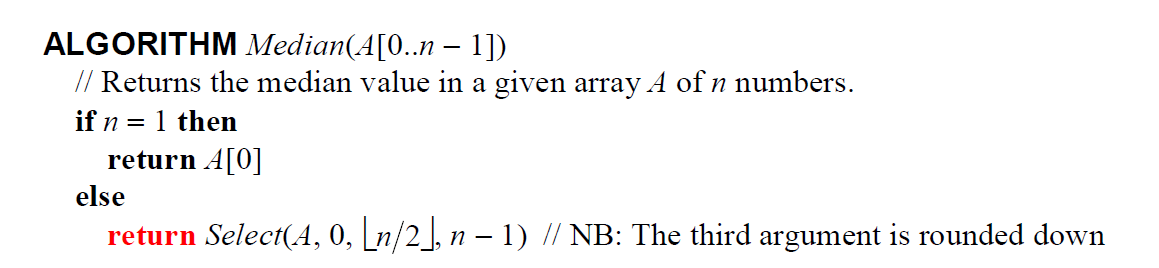


Figure 2: Johnsonbaugh and Schaefer Median method that returns the middle value located in select or handles the unique one element case

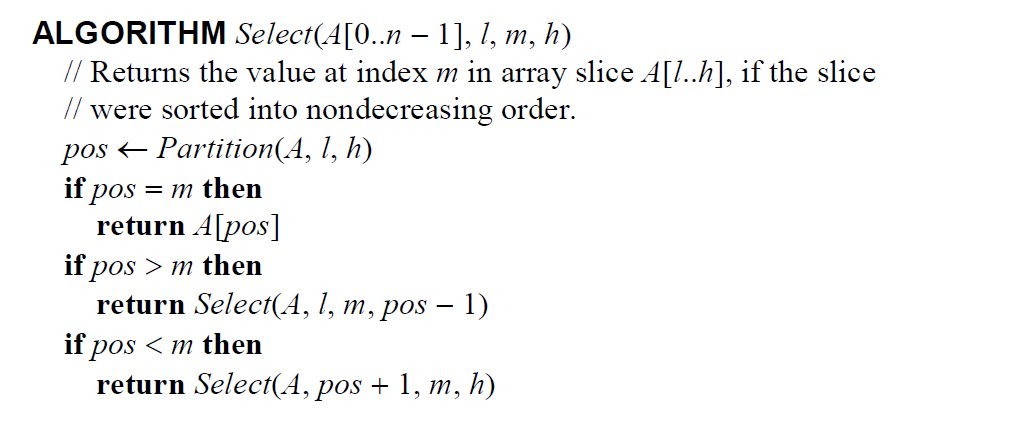


Figure 3: The Select Method that locates the middle value via recursively calling itself with the data set containing the Median

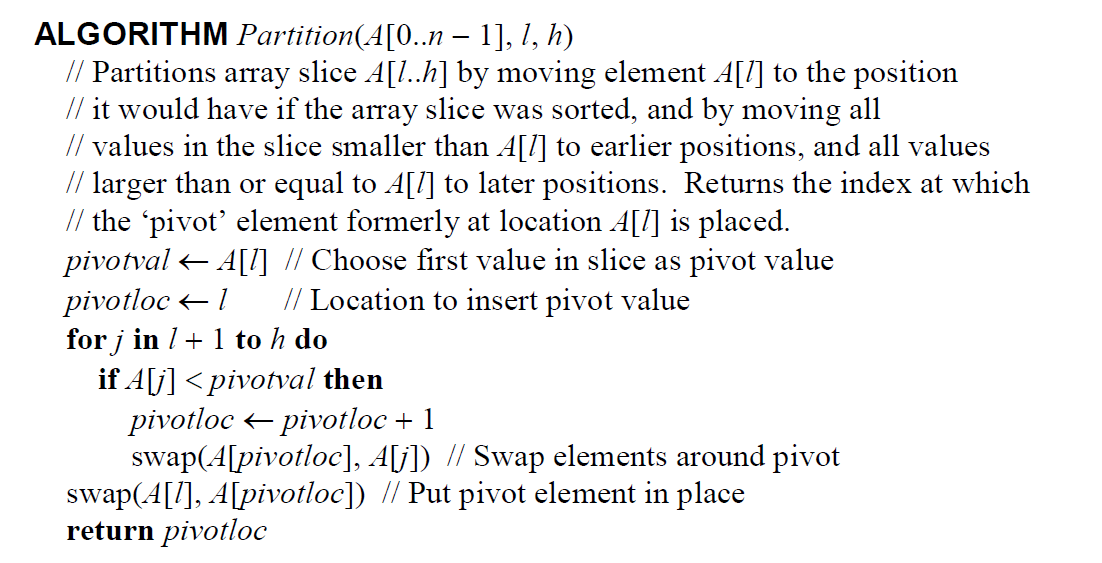


Figure 4: The partition method that sorts the specified data range around a pivot point

# **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 viewing the algorithms, as seen in figure 1, the basic operation of choice was the comparison ***A[j] < A[i]***. 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, as the operation is performed the greatest number of times and this being a ‘sorting’ problem, the where the dominant operations are the main comparisons 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 is simplified as it is a consecutive comparison after ***A[j] < A[i]*** as per Levitin’s simplifying notion, which simplifies this into a single comparison when calculating the order of growth [4]. 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 2. This statement was selected due to it being a crucial element in not only the ***Partition*** method, but the ***Median*** algorithm as well. This is due to the fact that the median methods complexity relies on that of the ***Select*** method and in turn the ***Partition*** method, with this comparison being the dominant operation as this is a sorting problem. 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 ***Partition*** method and comparisons made in the ***Select*** and ***Median*** methods, 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 is the dominant operation and provides a 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 contained within. While also as these are solutions to a selection problem, the basic operations are chosen to reflect this, as they are the key comparisons. 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 as a selection problem [4]. 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, 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 when the median is located. Therefore, a probabilistic distribution needs to be defined such that it represents where the search key, in this case the value that has the middle index, 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*** which will be equal to 1, and that it has equal chance of being with any element, which can be represented as:

Therefore, calculating the average complexity for the average-case efficiency can be achieved by utilising the expected value formula for discrete probability distributions, where the number of times the basic operation is executed 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:

Which can be simplified to,

With the probability that the median is contained within the input data equal to one, ***p = 1***, the average case efficiency of the Brute Force Median algorithm is equal to:

### **3.2.2 Partition Median Algorithm**

In order to analyse the ***Median*** method, 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, excluding the pivot, this 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 for large data inputs 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 what size the array will decrease by during the ***Partition*** method. While in the best-case scenario, the ***Partition*** method will always return the median value however where the array size would decrease by one half therefore, it is assumed that upon each iteration the array size will decrease by approximately one half. It should also be noted that the average decrease in the input size is not exactly one half, however, this provides a suitable assumption for the basis of this analysis.

To simplify calculations, let

Where after each iteration, the basic operation is executed times after the ***Partition*** method is performed. This recursive function can therefore be simplified to,

This function represents the worst-case complexity of the ***Select*** 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. Thus, in order to calculate the average case efficiency, the expected value formula is utilised where it is 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 to find the expected number of times the basic operation is performed. This range is from times to times, the expected complexity can be determined to be,

Simplifying this equation down,

Now as and ,

Note that this is the approximate complexity as variance from this estimation can arise due to the assumptions made. 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 ***Select*** method,

Note that due to the assumptions made, the exact coefficient cannot be calculated however, the order of growth calculated

## 3.3 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 1 along with the fact that there is a need to check every element in the input data with every other element. While, Johsonbaugh and Schaefer’s solution to the median problem produced an order of growth of through the formal analysis conducted, and also as stated by Levitin through an analogy, and via Berman and Paul whom details an analysis for their version of the algorithm [2] [4] [5].

# **Methodology, Tools and Techniques**

## 4.1 Programming Environment

The algorithms and experiments were developed using a combination of two different Java Integrated Development Environments (IDE). The IntelliJ IDEA Community (ver. 2017.3.5) was used for the implementation of the Brute Force Median algorithm and Eclipse Oxygen 2 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 64bit home version. With the hardware specifications of the testing machine being: Intel’s 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 random 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 MATLABs inbuilt plot function. MATLAB scripts are created which reads test data from the CSV formatted files and plots it along with the theoretical growth rate.

## 4.2 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 1 to 4 were implemented using the Java Development Kit (ver. 1.8) and can be seen in Appendix I to IV respectively.

## 4.3 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 arrays containing integers as seen in Appendix IV and V. The test classes ensure the methods and the results are accurate. These methods are also tested against unusual inputs to further ensure the functionality.

To count the number of basic operations performed by each algorithm to complete a search, a counter was added to increment each time they were performed by 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 one thousand to one hundred thousand, this test is then also performed forty times before the results are averaged.

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 system time from the stored start time as recorded from the system clock. 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 IV. 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 0 after inspecting the algorithm, of which was returned however, if implementing this algorithm it is recommended that a check be placed in to ensure that the user knows that the data just passed contained no elements. 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 V, 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

This section details the empirical analysis of number of basic operations executed of the two algorithms investigated in this report and compares them with the theoretical efficiencies. These tests conducted simply count the number of basic operations executed by the implemented algorithms via a simple long integer counter.

**5.2.1 Brute Force Algorithm**

The results of counting the basic operation for the Brute Force Median algorithm can be seen in figures 5 and 6. Where figure 5 displays the averaged results of the tests, of which the code utilised to complete these tests can be seen in Appendix VI, where each point on data can be distinguished on the graph, as seen by a circle around it. While figure 6 displays these results against the theoretical complexity, as analysed in section 3.2.1. As seen in figure 5 the experimental results do form a quadratic rate of growth which matches previous analysis and that conducted in this report. This is further proven by figure 6, where the results when graphed against the theoretical values where there is a correlation between the predicted and theoretical results by how closely the data matches, this therefore demonstrates that the growth of the algorithm is quadratic in nature. It should be noted that the variance seen in the experimental results arise due to the probabilistic nature of the algorithm, thus if the total number of tests completed, before averaging, where to be increased then this variance within the data will decrease.

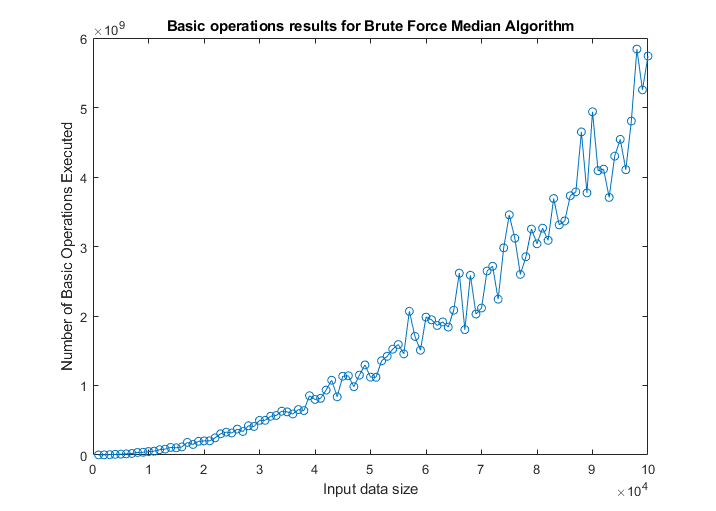


Figure 5: Experimental results of Brute Force Basic Median Operation counter test

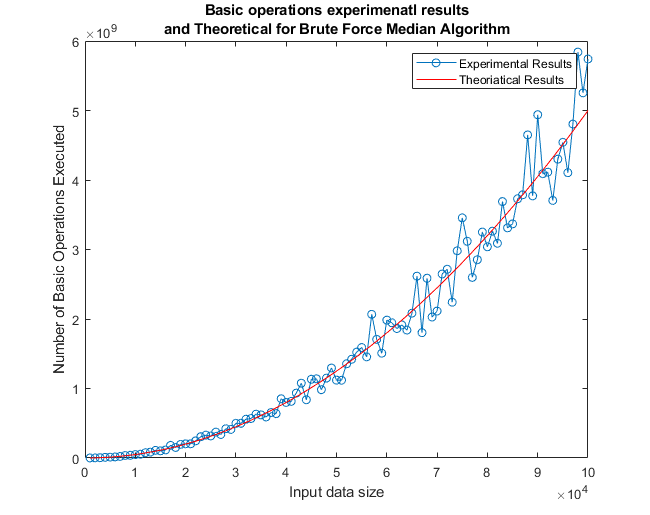


Figure 6: Experimental (blue) and Theoretical (red) Results of the Brute Force Algorithm Basic Operation counter test

**5.2.2 Partition Algorithm**

The basic operation counter test results, which was gathered through implementing a Java program of which the code can be seen in Appendix VII, for the partitioning median algorithm test can be viewed in figures 7 and 8, where each data point can be easily seen. Figure 7 simply displays the experimental results gathered from the test while figure 8 then displays this these results against the theoretical results as outlined in section 3.2.2. Figure 7 clearly displays a linear relationship between the input data size and the number of times the basic operation is executed. This is further emphasised in figure 8 that clearly displays that the order of growth of the experimental results closely match that of theoretical linear growth. It should be noted however, that the difference seen in the theoretical and the experimental results is due to the assumptions made in the theoretical calculations. Particularly, the assumption that upon each partition, the array size decreases by approximately a factor of one half where in practical application this is not the case. While also, within the experimental results, the variance can easily be seen. This is due to the probabilistic nature of the algorithm, and thus increasing the total number of tests conducted will further improve the results gathered from testing.

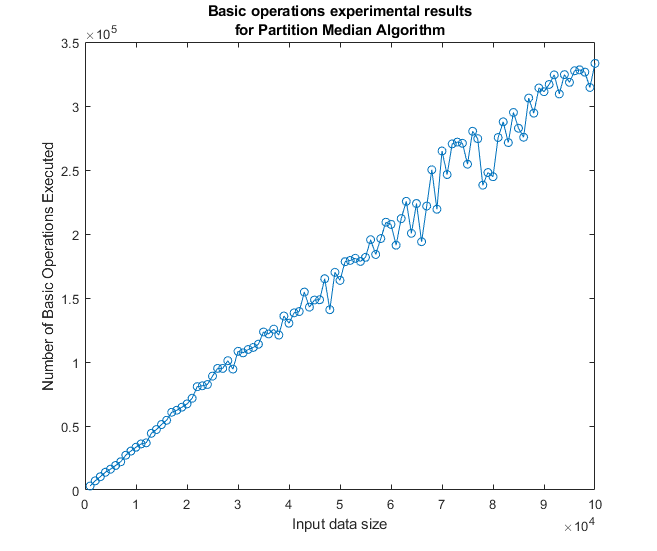


Figure 7: Experimental results of the partition median Basic Operation counter test

## 

Figure 8: Experimental (blue) and Theoretical Results (red) of the partition median Algorithm Basic Operation counter test

**5.2.3 Basic Operation Algorithm Comparison**

Figure 9 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 . This clearly displays that the Brute Force algorithm is significantly inefficient in comparison to the partitioning median algorithm. Thus, it is recommended that for the median implementation, the partitioning median algorithm should be implemented. It should also be noted that while there is a small functional difference that exists that may result in different results, it is believed that this has a negligible impact on all results gathered.

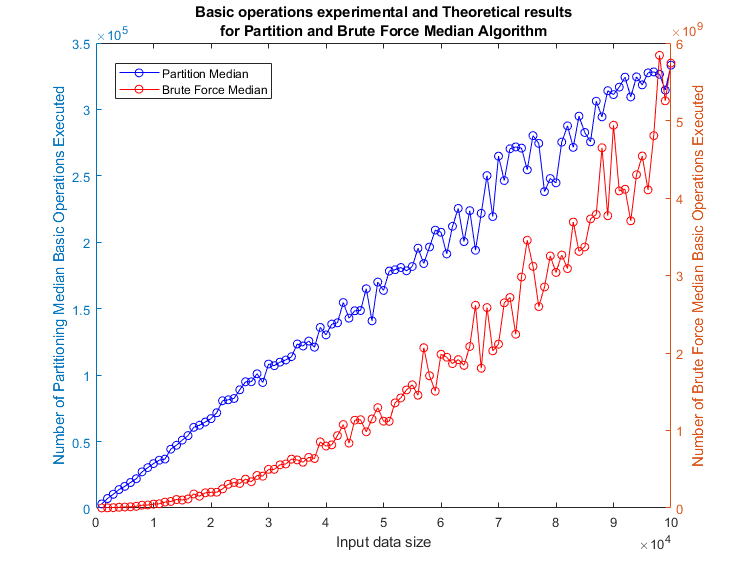


Figure 9: Experimental results of the Brute Force algorithm (blue) and the partitioning median algorithm (red) for the basic operation tests

## Average-Case Execution Time

In this section, the results of the empirical analysis of the execution time of the two algorithms analysed within this report, the results of these experiments are then compared against the theoretical rate of growth of the algorithms. These tests where conducted by utilising Java’s System clock with nanosecond precision with the results stored in a long integer format.

**5.3.1 Brute Force Algorithm**

The results of the Brute Force Median execution time can be seen within two figures, the first, figure 10, depicts the experimental results that were gathered from the Java programs, whose code can be viewed in Appendix VII. While figure 11 shows these experimental graphed against the theoretical results that were analysed in section 3.2.1 of this report. These graphs clearly show that the rate of growth of the algorithm is quadratic in nature. While also, each point that contributes to this growth can clearly be seen on the graph as they are outlined by a circle. Then comparing these data with the theoretical average complexity calculations also show a clear quadratic growth. While the theoretical and experimental results were required to be graphed on different axis, this is simply due to how the theoretical calculations are in terms of the number of times the basic operation is performed. Translating this to an execution time would be inaccurate due to the fact that the execution time is heavily dependent on the computing environment its performed on. It should also be noted that within the test data gathered, there is a variance seen in the results due to the probabilistic nature of the average case efficiency of the algorithm, therefore, if the number of total tests performed where to be increased this variance will reduce in the final results gathered.

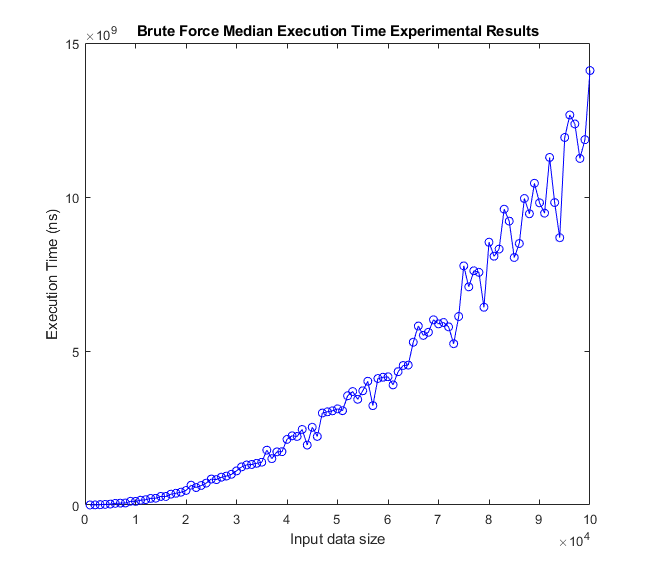
****

Figure 10: Experimental results of Brute Force Basic Median execution time test

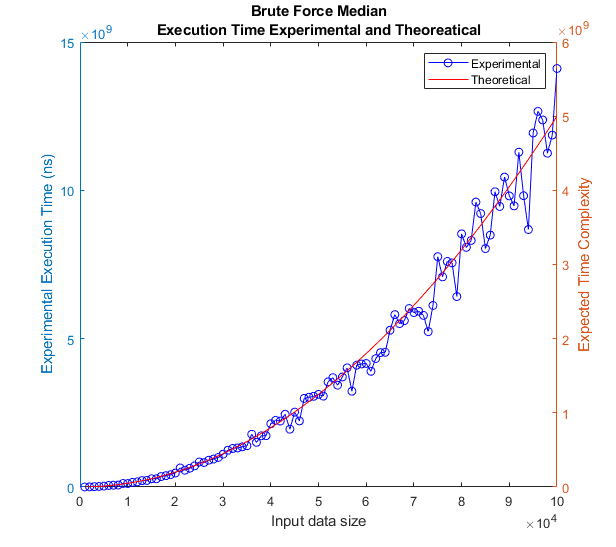
****

Figure 11: Experimental (blue) and Theoretical (red) Results of the Brute Force Algorithm execution time test

**5.3.2 Partition Algorithm**

The results of the execution time tests for the partitioning median algorithm can be viewed in Figures 12 and 13. While the code utilise in the Java program to perform these tests can be viewed in Appendix IX. The results in figure 12 clearly displays a linear growth between the input data and the execution time of the algorithm. Therefore, it can be comfortably stated that the time complexity of the algorithm, as analysed in Section 3.2.2, is linear in growth. To further emphasise this the graph in figure 13 clearly shows a correlation between the experimental and theoretical growth of the algorithm. This is despite the variance that can clearly be seen however, increasing the total number of times the test was performed is expected to reduce this factor. It should be noted that the theoretical and experimental results utilise different y-axis scales as the theoretical results were calculated in terms of the number of basic operations executed and thus, to translate this to the execution time, the results will vary across computers due to the reliance on the computing environment. Also, there is a slight difference between the theoretical and experimental results, this is due to the assumptions made during the analysis as previously mentioned.

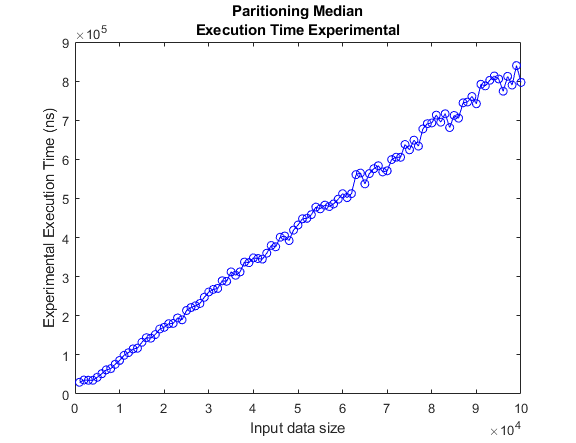


Figure 12: Experimental results of the partition median time execution test

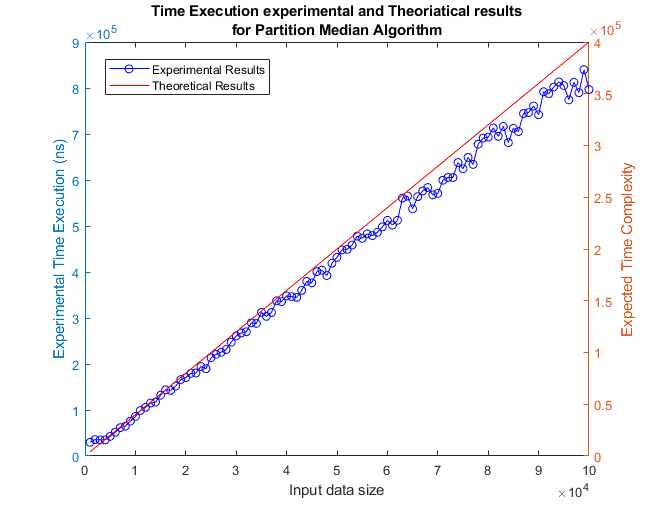


Figure 13: Experimental (blue) and Theoretical Results (red) of the partition median Algorithm Basic Operation counter test

**5.3.3 Execution Time Algorithm Comparison**

Lastly Figure 14 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. Thus, it can be stated that the partitioning algorithm provides a significantly better efficiency when implemented for locating the median within an unsorted data set when compared to the brute force algorithm, with the functional difference between the algorithms having negligible impact.

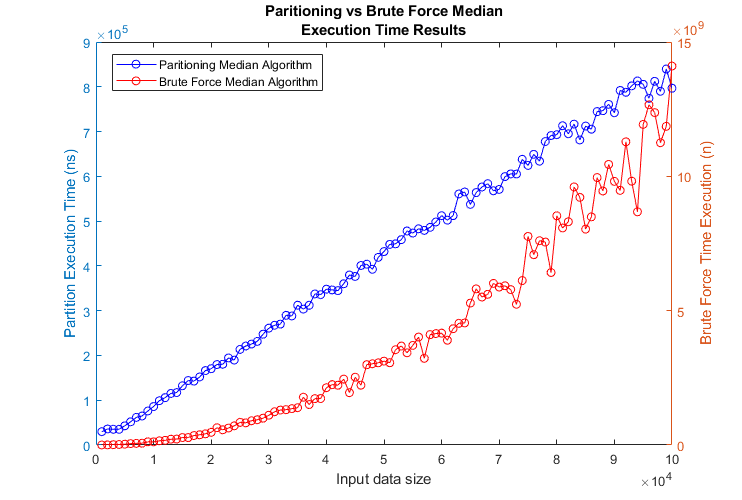


Figure 14: : Experimental results of the Brute Force algorithm (blue) and the partitioning median algorithm (red) for the time execution tests

# **6.0 References**

[1] "Mean, Mode and Median - Measures of Central Tendency - When to use with Different Types of Variable and Skewed Distributions | Laerd Statistics", *Statistics.laerd.com*, 2018. [Online]. Available: https://statistics.laerd.com/statistical-guides/measures-central-tendency-mean-mode-median.php. [Accessed: 16- May- 2018].

[2] R. Johnsonbaugh and M. Schaefer, *Algorithms*. Upper Saddle River, NJ: Pearson Education, 2003, pp. 262-267.

[3] "QuickSelect", *Cs.yale.edu*, 2014. [Online]. Available: http://www.cs.yale.edu/homes/aspnes/pinewiki/QuickSelect.html. [Accessed: 15- May- 2018].

[4] A. Levitin, *Introduction to the design & analysis of algorithms*, 2nd ed. Boston, Mass: Pearson, Addison Wesley, 2008, p. 44, 97-120, 136, 189.

[5] K. Berman and J. Paul, *Algorithms*. Boston: Thomson, 2005, p. 248.

[6] D. Barrington, "INFO 150 Notes on Brute-Force Search Lecture, Fall 2017", *People.cs.umass.edu*, 2017. [Online]. Available: https://people.cs.umass.edu/~barring/info150/lecture/search.html. [Accessed: 10- May- 2018].

# **Appendices**

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

1 public static int BruteForceMedian(int[] array) throws Exception  
2 {  
3 int k = (int)Math.*ceil*((array.length+1)/2);  
4   
5 for (int i = 0; i < array.length; i++){  
6   
7 int smaller = 0;  
8 int equal = 0;  
9   
10 for (int j = 0; j < array.length; j++)  
11 {  
12 if (array[j] < array[i])  
13 {  
14 smaller += 1;  
15 }  
16 else if (array[j] == array[i])  
17 {  
18 equal += 1;  
19 }  
20 }  
21 if (smaller < k && k <= (smaller + equal))  
22 {  
23 return array[i];  
24 }  
25 }  
26 return k;  
27 }

The Brute Force Algorithm, as described in figure 1, is contained the ***BruteForceAlgorithm*** class. The algorithm functions by setting an integer, ***k***, to half the array length, , and if necessary rounding the value up to the next whole integer as seen on line 3. The algorithm then enters a nested ***for*** loop that then compares each value, as indexed by ***i***, by every other element, as indexed by ***j***. This is done to determine the total number of values less than, as completed on lines 12 to 15, which increments a counter, and the total number of elements equal to, as completed on lines 16 to 19, which also increments a separate counter. Upon completion of the inner ***for*** loop, the current element, ***i***, is then checked to see if it is the median and if so returned, as its position in a sorted data set can be extrapolated to be between the total number of smaller elements to the sum of total number smaller and equal elements, as seen on lines 21 to 24. If the current element is not the median, then the algorithm continues on to the next iteration of the outer ***for*** loop to then check the next element. If for some reason the median is not found, then the original value of ***k*** is returned, as seen on line 26.

## Appendix II: Code for Partitioning Median Algorithm Implementation

The algorithm median whose pseudo code can be viewed in figure 2, 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 ***Partition*** method. All of which is contained within the ***PartitionAlgorithm*** Class.

1 public static int Median(int[] array)  
2 {  
3 if ( array.length == 1 )  
4 return array[0];  
5 else  
6 return *Select*(array, 0,(int)Math.*floor*(array.length / 2), array.length - 1);  
7 }

As seen above, the ***Median*** method 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 4. 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 6.

8 public static int Select(int[] array, int low, int mid, int high)  
9 {  
10 int pos = *Partition*( array, low, high );  
11   
**12** if ( pos == mid )  
13 return array[pos];  
14 if ( pos > mid )  
15 return *Select*( array, low, mid, pos - 1 );  
16 if ( pos < mid )  
17 return *Select*( array, pos + 1, mid, high );  
18   
19 return 0;  
20 }

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 10. 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, it is then compared with ***mid*** parameter, as seen on lines 12 to 17. After the ***Partition*** method has been executed 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 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 14 and 16. If the median value was selected then it is simply returned as seen on line 13.

21 public static int Partition(int[] array, int low, int high) throws Exception  
22 {  
23 int pivotval = array[low];  
24 int pivotloc = low;  
25   
26 for ( int j = low + 1; j <= high; j++ )  
27 {   
28 if ( array[j] < pivotval )  
29 {  
30 pivotloc++;  
31   
32 // swap elements  
33 int tempVal = array[pivotloc];  
34 array[pivotloc] = array[j];  
35 array[j] = tempVal;  
36 }  
37 }  
38   
39 int tempVal = array[low];  
40 array[low] = array[pivotloc]; // swap elements around pivot  
41 array[pivotloc] = tempVal; // put pivot element in place  
42   
43 return pivotloc;  
44 }

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 ***Partition*** method is complete (***pivotloc***), which is initially set to equal ***low***, as seen on lines 23 and 24. 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 26. 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 incremented 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 30 to 35. 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 39 to 43.

## Appendix III: Code for Generating Random Test Data

1 public static int[] generateRandomArray( int size )  
2 {  
3 Random randomNumberGenerator = new Random();  
4  
5 int[] randArray = new int[size];  
6  
7 for ( int i = 0; i < size; i++ )  
8 {  
9 randArray[i] = randomNumberGenerator.nextInt();  
10 }  
11  
12 return randArray;  
13 }

In order to provide sufficient test data to both algorithms, a static method has been created to generate array of a set size that contains random test data. This method is contained within the ***Main*** class for easy use during testing. The random array generator must be sent an integer, ***size***, in order to provide the necessary length that the array must be as seen on line 1 and 3. This method also utilises Javas random number generator in order to set each element in the array to a random integer, as seen on line 9, before returning array.

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

1 @Test  
2 public void testBasicCase() throws Exception{  
3 *assertEquals*(4, BruteForceAlgorithm.*BruteForceMedian*(new int[]{1, 3, 4, 6, 7}));  
4 *assertEquals*(4, BruteForceAlgorithm.*BruteForceMedian*(new int[]{4, 7, 6, 1, 3}));  
5 *assertEquals*(3, BruteForceAlgorithm.*BruteForceMedian*(new int[]{-1, -2, 3, 4, 7, 9}));  
6 *assertEquals*(3, BruteForceAlgorithm.*BruteForceMedian*(new int[]{7, 4, 9, 3, 1, 2}));  
7 }  
8  
9 @Test  
10 public void testEdgeCases() throws Exception{  
11 *assertEquals*(1, BruteForceAlgorithm.*BruteForceMedian*(new int[]{1}));  
12 *assertEquals*(8, BruteForceAlgorithm.*BruteForceMedian*(new int[]{8}));  
13 *assertEquals*(1, BruteForceAlgorithm.*BruteForceMedian*(new int[]{1,1,1,1,1,1,1,1,1,1}));  
14 *assertEquals*(4, BruteForceAlgorithm.*BruteForceMedian*(new int[]{1, 1, 1, 1, 8, 8, 8, 8, 4}));  
15 }  
16  
17 @Test

18 public void testEmptyCase() throws Exception{  
19 *assertEquals*(0, BruteForceAlgorithm.*BruteForceMedian*(new int[]{}));  
20 }  
21  
22 @Test  
23 public void testOperationCount() throws Exception{  
24 BruteForceAlgorithm.*operCounter* = 0;  
25 BruteForceAlgorithm.*BruteForceMedianBasicCounter*(new int[]{1, 2, 3, 4, 5});  
26 *assertEquals*(15, BruteForceAlgorithm.*operCounter*);  
27 }

Test cases for the brute force algorithm were developed in the ***TestBruteForceAlgorithm*** class using JUnit to test basic cases, edge cases and, operation count functionality. The basic cases are fairly straight forward and just test that the correct values are returned when sent an array of set values as seen lines 2 to 7. The edge cases run the same tests as the basic cases except the arrays data is either a single value array or similar elements contained within, as these arrays are less likely to arise in normal conditions, as seen on lines 10 to 15. The edge cases ensure that the algorithm accurately functions with odd array values. Also note that on lines 18 to 20, an empty array is passed as a test case, upon inspecting the array, it is expected that this return 0 however, when implementing this algorithm for use it is recommended to insert a check to ensure that the array contains data as this may cause errors if it is not notice by the user. Operation count also ensures that the algorithm functions correctly when counting the total number of operations executed by the algorithm as seen on lines 23 to 27. All these methods utilised the ***assertEquals*** method as provided by JUnit in order to ensure that the expected values where returned.

## 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.

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

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 4 to 8, hard coded arrays are passed to the ***assertEquals*** from the ***Median*** method and compared with the known median for the array.

10 @Test  
11 public void testEdgeCases()  
12 {  
13 *assertEquals*( 1, PartitionAlgorithm.*Median*( new int[] { 1 } ) );  
14 *assertEquals*( 5, PartitionAlgorithm.*Median*( new int[] { 5 } ) );  
15   
16 *assertEquals*( 4, PartitionAlgorithm.*Median*( new int[] { 1, 1, 1, 1, 1, 5, 5, 5, 5, 5, 4 } ) );  
17 *assertEquals*( 1, PartitionAlgorithm.*Median*( new int[] { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 } ) );  
18 }

This next test method, as seen above, then aims to test more unusual input arrays that may arise during operation. As seen on lines 13 to 17 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.

19 @Test (expected = Exception.class)  
20 public void testEmptyArray() throws Exception  
21 {  
22 PartitionAlgorithm.*Median*( new int[] {} );  
23 }

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 19, due to an index out of bounds exception that will be in counted.

24 @Test  
25 public void testOperationCount(){  
26 PartitionAlgorithm.*MedianBasicOperationCount*(new int[]{1, 2, 3, 4, 5});  
27 *assertEquals*(9, PartitionAlgorithm.*basicCounter*);  
28 }

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

1 public static long *operCounter* = 0;  
23 public static int BruteForceMedianBasicCounter(int[] array) throws Exception  
4 {  
5 int k = (int)Math.*ceil*((array.length+1)/2);  
6   
7 for (int i = 0; i < array.length; i++)  
8 {  
9 int smaller = 0;  
10 int equal = 0;  
11   
12 for (int j = 0; j < array.length; j++)  
13 {  
14 *operCounter*++;  
15   
16 if (array[j] < array[i])  
17 {  
18 smaller += 1;  
19 }  
20 else if (array[j] == array[i])  
21 {  
22 equal += 1;  
23 }  
24 }  
25 if (smaller < k && k <= (smaller + equal))  
26 {  
27 return array[i];  
28 }  
29 }  
30 return k;  
31 }

In order to test for the basic operations counter, the algorithm must be modified with a counter in order to keep track of the total number of times the basic operation is executed of which can be seen above and is contained with the ***BruteForceAlgorithm*** class. The counter type implemented of a long integer that increments in the nested ***for*** loop of the algorithm as seen on line 14. The second part of the code details the implementation for executing the basic operations test, which is contained on the next page.

32 public static void bruteForceBasicOperationCounterTests() throws Exception   
33 {  
34 int numTests = 40;  
35 int numArraysTested = 100;  
36   
37 int increamentSize = 1000;  
38 int size = 1000;  
39   
40 int[] sizeOfArray = new int[numArraysTested];  
41 long[] basicOperationCounter = new long[numArraysTested];  
42  
43 FileWriter fl = new FileWriter( "bruteForceBasicOperationsCounterTest.csv" );  
44 // fl.write("Array Size, Basic Operations\n");  
45   
46 // Initialise basic operation counter  
47 for ( int i = 0; i < numArraysTested; i++ )  
48 basicOperationCounter[i] = 0;  
49  
50 for (int j = 0; j < numArraysTested; j++)  
51 {  
52 for ( int i = 0; i < numTests; i++ )  
53 {  
54 System.*out*.println( "starting array size " + size + " \ttest " + (i + 1) );  
55  
56 int[] randArray = *generateRandomArray*( size );  
57  
58 BruteForceAlgorithm.*operCounter* = 0;  
59  
60 BruteForceAlgorithm.*BruteForceMedianBasicCounter*( randArray );  
61  
62 // save data  
63 sizeOfArray[j] = size;  
64 basicOperationCounter[j] += PartitionAlgorithm.*basicCounter*;  
65  
66 System.*out*.println("Basic operations performed: "

67 + PartitionAlgorithm.*basicCounter* + "\n");  
68 }  
69   
70 // Calculate average  
71 basicOperationCounter[j] /= numTests;   
72   
73 // Write data to file  
74 fl.write( sizeOfArray[j] + "," );  
75 fl.write( basicOperationCounter[j] + "\n");  
76   
77 size += increamentSize;  
78 }  
79   
80 fl.close();  
81 }

The second part of the code is located in the ***Main*** class, it operates by first initialising any variables and objects needed as seen on lines 34 to 43. Next it initialises the array that is to contain the results of the number of basic operations executed, as seen on lines 47 and 48. Next the test begins by entering two nested ***for*** loops, the first controlling the size and number of arrays tested and the next controlling the total number of tests performed. Then upon each test, a random array is generated, line 56, the variable containing the total number of operations counted is initialised to zero, line 58, the random array is then passed to the modified Brute Force Median method. After the execution of this, the basic operation counter is then saved into an array that contains the total number of basic operations for an array of that size, along with the size of the array, as seen on lines 60 to 64. After the sum of the basic operations for a specific array size has been saved, then the average is calculated before being written to a file in CSV format as seen on lines 71 to 75. Then in order to continue testing the array size is incremented by a specific amount defined at the start of the test.

## 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 1, 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 as seen on line 11. 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.

1 public static long *basicCounter* = 0;

2

3 public static int PartitionBasicOperationsCount(int[] array, int low, int high) throws Exception  
4 {  
5 int pivotval = array[low];  
6 int pivotloc = low;  
7   
8 for ( int j = low + 1; j <= high; j++ )  
9 {  
10  
11 *basicCounter*++;  
12   
13 if ( array[j] < pivotval )  
14 {  
15 pivotloc++;  
16   
17 // swap elements  
18 int tempVal = array[pivotloc];  
19 array[pivotloc] = array[j];  
20 array[j] = tempVal;  
21 }  
22 }  
23   
24 int tempVal = array[low];  
25 array[low] = array[pivotloc]; // swap elements around pivot  
26 array[pivotloc] = tempVal; // put pivot element in place  
27   
28 return pivotloc;  
29 }

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 39 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 as seen on lines 53 to 55. 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*** as seen on lines 57 to 61. This inner loop continuously executes until it has done the required number of tests for the array of that size. The average value for the number of basic operations for this array size is then calculated before being stored into a txt file of CSV format, line 68 to 72, before continuing on to the next set of tests for a larger array size. After testing is completed, the data contained within the CSV is then graphed by utilising MATLAB.

30 public static void partitionBasicCounterTest() throws Exception   
31 {  
32 int numTests = 40;  
33 int numArraysTested = 100;  
34   
35 int increamentSize = 1000;  
36 int size = 1000;  
37  
38 int[] sizeOfArray = new int[numArraysTested];  
39 long[] basicOperationCounter = new long[numArraysTested];  
40  
41 FileWriter fl = new FileWriter( "partitionBasicOperationsCounterTest.csv" );  
42   
43 // Initialise execution time counter  
44 for ( int i = 0; i < numArraysTested; i++ )  
45 basicOperationCounter[i] = 0;  
46  
47 for (int j = 0; j < numArraysTested; j++)  
48 {  
49 for ( int i = 0; i < numTests; i++ )  
50 {  
51 System.*out*.println( "starting array size " + size + " \ttest " + (i + 1) );  
52  
53 int[] randArray = *generateRandomArray*( size );  
54  
55 PartitionAlgorithm.*basicCounter* = 0;  
56  
57 PartitionAlgorithm.*MedianBasicOperationCount*( randArray );  
58  
59 // save data  
60 sizeOfArray[j] = size;  
61 basicOperationCounter[j] += PartitionAlgorithm.*basicCounter*;  
62   
63 System.*out*.println( "Basic operations performed: "

64 + PartitionAlgorithm.*basicCounter* + "\n");  
65 }  
66   
67 // Calculate average  
68 basicOperationCounter[j] /= numTests;   
69   
70 // Write data to file  
71 fl.write( sizeOfArray[j] + "," );  
72 fl.write( basicOperationCounter[j] + "\n");  
73   
74 size += increamentSize;  
75 }  
76   
77 fl.close();  
78 }

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

1public static void bruteForceTimeExecutionTests() throws Exception   
2 {

3 int numTests = 40;  
4 int numArraysTested = 100;  
5   
6 int increamentSize = 1000;  
7 int size = 1000;  
8  
9 int[] sizeOfArray;  
10 long[] executionTimeCounter;  
11  
12 FileWriter fl = new FileWriter( "bruteForceExecutionTimeTest.csv" );  
13 // fl.write("Array Size, Execution Time(ms)\n");  
14   
15 sizeOfArray = new int[numArraysTested];  
16 executionTimeCounter = new long[numArraysTested];  
17   
18 // Initialise execution time counter  
19 for ( int i = 0; i < numArraysTested; i++ )  
20 executionTimeCounter[i] = 0;  
21  
22 for (int j = 0; j < numArraysTested; j++ )  
23 {  
24 for ( int i = 0; i < numTests; i++ )  
25 {  
26 System.*out*.println("starting array size " + size + " \ttest " + (i + 1));  
27  
28 int[] randArray = *generateRandomArray*( size );  
29  
30 startTime = System.*nanoTime*();  
31 BruteForceAlgorithm.*BruteForceMedian*( randArray );  
32 long duration = System.*nanoTime*() - startTime;  
33   
34 // save data  
35 executionTimeCounter[j] += duration;  
36 sizeOfArray[j] = size;  
37  
38 System.*out*.println( "Execution time: " + duration + "ns\n");  
39 }  
40   
41 // calculate Average  
42 executionTimeCounter[j] /= numTests;   
43   
44 // Write to CSV  
45 fl.write( sizeOfArray[j] + "," );  
46 fl.write(executionTimeCounter[j] + "\n");  
47   
48 size += increamentSize;  
49 }  
50   
51 fl.close();  
52 }

The code, which is contained within the ***Main*** class, used to calculate the execution time for the brute force algorithm functions using the built-in Java system time module while the format of the code is similar to the basic operations counter test seen in Appendix VI. Prior to executing the Brute Force Median, the system time is stored into a long integer as seen on line 30, then after running the algorithm, the execution time is calculated by subtracting the system start time from the current system time (in nanoseconds), as seen on line 32. The execution time is stored within an array that holds the total execution time for all tests, this data is then utilised to get the average result and save it into a CSV formatted file as seen on lines 35 and 45 to 46.

## 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 implementation details of recording the average execution time is similar to the method used for the basic operations counter, however, on lines 31 to 33 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 thus is utilised to calculate and average execution time, as seen on lines 36 and 43.

1 public static void partitionExecutionTimeTest() throws Exception  
2 {  
3 int numTests = 40;  
4 int numArraysTested = 100;  
5   
6 int increamentSize = 1000;  
7 int size = 1000;  
8   
9 long startTime;  
10  
11 int[] sizeOfArray;  
12 long[] executionTimeCounter;  
13  
14 FileWriter fl = new FileWriter( "partitionExecutionTimeTest.csv" );  
15   
16 sizeOfArray = new int[numArraysTested];  
17 executionTimeCounter = new long[numArraysTested];  
18   
19 // Initialise execution time counter  
20 for ( int i = 0; i < numArraysTested; i++ )  
21 executionTimeCounter[i] = 0;  
22   
23 for (int j = 0; j < numArraysTested; j++ )  
24 {  
25 for ( int i = 0; i < numTests; i++ )  
26 {  
27 System.*out*.println( "starting array size " + size + " \ttest " + (i + 1) );  
28  
29 int[] randArray = *generateRandomArray*( size );  
30  
31 startTime = System.*nanoTime*();  
32 PartitionAlgorithm.*Median*( randArray );  
33 long duration = System.*nanoTime*() - startTime;  
34   
35 // save data  
36 executionTimeCounter[j] += duration;  
37 sizeOfArray[j] = size;  
38   
39 System.*out*.println( "Execution time: " + duration + "ns\n");  
40 }  
41   
42 // calculate Average  
43 executionTimeCounter[j] /= numTests;   
44   
45 // Write to CSV  
46 fl.write( sizeOfArray[j] + "," );  
47 fl.write(executionTimeCounter[j] + "\n");  
48

49 size += increamentSize;  
50 }  
51 fl.close();  
52 }