CAB301 Assignment 2

Empirical Comparison of Median Calculation Algorithms

Kathryn Dorge (n9701761)

and Luke Reynolds (n10083481)

Date submitted:

Table of Contents

[1.0 Summary 2](#_Toc513463309)

[2.0 Description of the Algorithm 2](#_Toc513463310)

[2.1 Brute Force Median 2](#_Toc513463311)

[2.2 Partitioning Median 2](#_Toc513463312)

[3.0 Theoretical Analysis of the Algorithm 3](#_Toc513463313)

[3.1 The algorithm’s basic operation 3](#_Toc513463314)

[3.1.1 Brute Force Median Algorithm 3](#_Toc513463315)

[3.1.2 Partition Median Algorithm 3](#_Toc513463316)

[3.2 Average Case Efficiency 3](#_Toc513463317)

[3.2.1 Brute Force Median Algorithm 3](#_Toc513463318)

[3.2.2 Partition Median Algorithm 3](#_Toc513463319)

[3.3 Best Case Efficiency 6](#_Toc513463320)

[3.4 Worst Case Efficiency 6](#_Toc513463321)

[3.5 Order of Growth 6](#_Toc513463322)

[4.0 Methodology, Tools and Techniques 7](#_Toc513463323)

[5.0 Experimental Results 8](#_Toc513463324)

[5.1 Functional Testing 8](#_Toc513463325)

[5.1.1 Brute Force Median 8](#_Toc513463326)

[5.1.2 Partitioning Median 8](#_Toc513463327)

[5.2 Average-Case Number of Basic Operations 9](#_Toc513463328)

[5.3 Average-Case Execution Time 9](#_Toc513463329)

[6.0 References 10](#_Toc513463330)

[7.0 Appendices 11](#_Toc513463331)

[7.1 Appendix I: Code for Algorithm Implementation 11](#_Toc513463332)

[7.2 Appendix II: Code for Generating Random Test Data 12](#_Toc513463333)

[7.3 Appendix III: Code for Testing Functionality 12](#_Toc513463334)

[7.4 Appendix IV: Code for Basic Operation Count 12](#_Toc513463335)

[7.5 Appendix V: Code for Time Execution Test 13](#_Toc513463336)

# **Summary**

# **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 set remains unsorted. Thus, an efficient algorithm for locating the median value has been investigated, designed and compared.

## **Brute Force Median**

## **Partitioning Median**

Johnsonbaugh and Schaefer have proposed a version of the median algorithm 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 data sets, 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**

## **The algorithm’s basic operation**

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

### **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.

## **Average Case Efficiency**

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

### **3.2.2 Partition Median Algorithm**

Upon execution of the partitioning median algorithm with the input data set being unknown and randomised, the execution of the basic operation is unknown due the divide and conquer nature of the algorithm. Therefore, the average case efficiency analysis is conducted, with assumptions made for an average data set.

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, the expected complexity can be determined to be,

This can be simplified to,

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,

## **Best Case Efficiency**

## **Worst Case Efficiency**

## **Order of Growth**

# **Methodology, Tools and Techniques**

# **Experimental Results**

## **Functional Testing**

### **Brute Force Median**

### **5.1.2 Partitioning Median**

Before commencing the empirical analysis of the partitioning algorithm, the functionality must first be verified. This was accomplished by utilising Java JUnit testing suite to provide the median method with a range of hard coded test cases. The code associated with these test cases are contained within Appendix …. These test cases include what is considered normal data set conditions, and any extreme cases that may arise during a programs execution. It should be noted, that both even and odd data set lengths where tested, as this solution to the selection median problem will select the upper value when two middle values are present, which occur in even sized data sets. In order to verify that the correct value was returned, the ***assertEquals*** assertion, which is a part of the JUnit test frame work, was utilised and given the expected value.

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.

## **Average-Case Number of Basic Operations**

## **Average-Case Execution Time**

# **References**

# **Appendices**

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

## **7.2 Appendix II: Code for Partitioning Median Algorithm Implementation**

## **7.3 Appendix III: Code for Generating Random Test Data**

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

## **7.5** **Appendix V: Code for Testing Partitioning Median Functionality**

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

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

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

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