CAB301 Assignment 2

Empirical Analysis of Two Algorithms

Student Name (Student Number):

Lachlan Cesca (n8580413)

Nathaniel Perkins (n7291035)

Due Date:

30th May 2016

# Summary

This report provides an in-depth analysis of two algorithms designed to find the median of a given array but with different approaches to the problem.

Several tests were conducted to experimentally determine the algorithms’ efficiencies, by measuring average-case execution times and average-case number of basic operations for each of the two algorithms. With the results from these tests, the two algorithms are directly compared against each other for their efficiencies based on the size of input arrays and whether or not the arrays are already ordered.

Using C++ to implement both algorithms, the results from testing are also compared against the theoretical to authenticate the validity of the algorithms theoretical predictions.

# - Description of the Algorithms

The two algorithms being analysed in this report are the Brute Force Median algorithm and the Selection Median algorithm. It is important to note that neither algorithms require a sorted array to find the median and depending on this, it will have an effect on the algorithms efficiency.

The Brute Force Median algorithm works by looping through the array from 0 to n-1 while using a second for loop to count the number of elements in the array that are smaller than the current value and also count the number of elements that equal to the current value. Once the second loop is done, the algorithm checks to see if the number of values smaller is less than half the number of elements and also that the sum of both the number of smaller values and the number of equal values are greater than or equal to half the number of elements. If this is the case, then it returns the current value or in this case the median value of the array.

The second algorithm works in 3 stages/functions, the first function (Median) checks that the array isn’t 1 long (if so, that is the median value) and initialises the recursion of the second function. The second function (Select) calls the third function and deals with the output of that. The third and final function (Partition) sorts the array from the lower bound element and puts all values smaller than this element before it and then returns the position of this element after the sort. The Select function then returns this value if it is the Median otherwise it will increase the lower bound or upper bounds of the Partition function according to if it was greater than or less than the median position.

It is also important to note that with an even number of inputs both functions output differently in the fact that the first algorithm will arbitrarily return the value left of the midpoint while the second function returns the value right of the midpoint.

# - Theoretical Analysis of the Algorithm

This section briefly describes both algorithms under theoretical analysis before later comparing them to their experimental results.

## 2.1 - Identifying the Algorithms’ Basic operation

There are 3 main criteria to consider when identifying an algorithms basic operation, these criteria are as followed [1]:

* Is the operation needed to solve the problem.
* Is the operation the most time consuming.
* Is the operation the most frequently used.

For the first algorithm in Figure 1 it can be seen that the most common operation must occur within the second for loop as this section of code will be executed the most. Therefore it is the comparison between A[j] and A[i] that happens inside the nested for loop as the other operations inside deal with incrementing value and are not considered basic operations. This comparison between these two values happens the most frequent and is also the most time consuming operation and it is because of this that this comparison is determined to be the basic operation of the algorithm.

For the second algorithm as shown Figure 2, Figure 3 and Figure 4, it is immediate to see that the basic operation must lie within the Partition function as it is this function that is recursively called the most from within the Select function. By looking at the Partition function it can be seen that the basic operation will fall somewhere inside the for loop and from there, it is obvious that the comparison between A[j] and pivotval is the most common operation. Not only is this operation the most common it is also crucial to the problem and the most time consuming, as the other operations deal with storing and moving variables and it is for these reason that this is the basic operation of the algorithm.

## 2.2 – Choice of Problem Size

For both algorithms the problem size is defined as the number of items or elements “n” in the input array to the function/s. This is because for each algorithm they both require an array of size n to work on. While one operates on the array recursively and the other operates in a loop, both methods maintain an array size of n.

## 2.3 – Average-Case efficiency

The initial assumption for the average-case efficiency provided for the BruteForceMedian algorithm is that it appears to have a quadratic efficiency:

To confirm this the inner loop of the BruteForceMedian function needs to be analysed first. It can be seen that the inner loop runs from 0 to n-1 for every iteration of the outer loop and so therefore has an efficiency of n. Next it can be seen that the outer loop can terminate at any time, but because the input to this function is a uniformly distributed randomized array, the likelihood of the algorithm exiting/returning is equally likely for any iteration. Therefore:

Simplified down to:

But due to the fact that the comparison happens n times for every iteration i:

This reinforces the initial assumption that the algorithm has a quadratic efficiency:

For the Selection-Partition Median Algorithm, it was provided that on an average case, the algorithm should have a linear average-case efficiency:

This was found to be true, through other literature [2] [3].

# 3.0 - Methodology, Tools and Techniques

# 3.1 – Programming Environment

1. The implementation of both algorithms was done in the C++ programming language through the eclipsed IDE. Eclipse is a great cross platform IDE with various debugging tools to facilitate code development.
2. The experiments were performed on Linux machine running Ubuntu 15.10 64bit, with 16GB DDR3 RAM and an Intel core i7-4510u CPU 2GHz quad core. Programs were run in the Eclipse IDE using the G++ compiler native to Linux with G++ 4.8.1 as the current compiler version. The C++ rand function was used for the randomness of the input arrays to both algorithms and the execution time for each algorithm was calculated by the clock function and then divided by the number of clocks per second, which is also provided in the clock module.
3. All data was exported to CSV files, and then graphs Figure 2 and Figure 3 were then generated in Excel. The report was then written up in Microsoft Word.

# 3.2 – Implementation of the Algorithms

The implementation of both algorithms require an array to be passed as an input. Arrays are the simplest structure to implement and as such, are used as the basis for testing. The pseudocode for both algorithms were straight forward to implement in C++ and can be found in Figure 1 for the Brute Force Median algorithm and Figure 2, Figure 3 and Figure 4 for the Select-Partition Median Algorithm.

# 3.3 – Generating test data and Running the Experiments

A series of tests were executed to test the functional validity of both algorithms by creating a series of arrays with known medians and determining if the functions would return the correct results.

A counter variable for both algorithms were placed before their respective basic operation’s as outlined in Section 2.1 to count the number of basic operations for the given input. Once the tests were performed for a particular array size, the result was averaged over the number of tests to provide the average number of basic operations for that given array size. Care was taken to make sure that the arrays were identical for each algorithm so as to not skew any results and keep the experiments fair and even.

The second lot of experiments dealt with measuring execution time and in order to calculate this, the clock function was use to count the number of clock cycles it took to execute the function for the given input array. Once the number of clock cycles was recorded, this value was then divided by the number of clock cycles in a second (which is also a part of the clock module) and was averaged over the number of iterations for that particular sized array. These experiments were done with unmodified versions of the original function (i.e. without additional counting variables) so as to keep the data as accurate as possible. As per the previous experiment, both functions were fed the same input array so as to make the tests fair and even.

For both sets of experiments, data was gathered with linearly increasing sampling rates. Simply, for every three (arbitrary but value should be kept small) iterations of the array size, the increment for the array size increased by one. This is because as the array size gets larger, the difference between concurrent array sizes has very little difference/impact and makes for cleaner and clearer graphs. We also expected theoretically that both functions would take the form of either quadratic or linear growth and as such the trend of the data would still be captured as the sampling rate increased linearly with the array size. However though if the trend happen to take the form of logarithmic/exponential growth, this method would not ideally capture all the relevant data points. (Insert reference to code for this section)

# 4.0 - Experimental Results

This section details the experimental results from testing both the BruteForceMedian algorithm and the Select-Partition Median algorithm and gathering their average-case number of basic operations and their average-case execution times. Both algorithms are then compared against each other for efficiency and relate back to their theoretical efficiencies as well.

## 4.1 - Functional Testing

To test the functionality of both algorithms, five unique tests were performed on each to find the median of various different input arrays. It is important to note that both algorithms would sometimes produce different results for even lengthen arrays as mentioned in Section 1.0, so care was taken to expect the correct output for certain tests:

1. Sorted odd sized array, of 11 elements long.
2. Sorted even sized array, of 10 elements long.
3. Unsorted odd sized array, of 11 elements long.
4. Unsorted even sized array, of 10 elements long.
5. A 10 element long array with all the same value.

The fifth test is an extreme case to see if the function will still operate given an unlikely input. From these three tests alone, the basic functionality of the algorithm can be determined for its validity. As expected, all tests performed correctly for both algorithms as seen in Figure 4.

## 4.2 - Average-Case Number of Basic Operations

For both algorithms the number of basic/operations/comparisons were tested with slightly modified versions of their base functions, this was so that a counter could be added to keep track of the number of basic operations. As mentioned in Section 3.3, the sampling rate for data increased linearly for every couple of iterations, so as to not waste time capturing insignificant data points.

Each function was repeated 100 times for each array size and the results averaged from array sizes 2 to 38,002 (after this point the computation time took too long to continue), because of the linearly increasing sampling rate the total number of data points ended up being 475.

The number of basic operations/comparisons was tested with a slightly modified version of the insertion sort function where the function would keep track of the comparisons and then the output of the function would return the number of comparisons made.

A sampling rate of 10 for the size of the array was used in order to reduce total run time. The overall results as seen in Figure 3 show a very clean smooth graph, which shows that the averaging of the number of comparisons worked out quite nicely. This is also noted by the R2 value of the trend line showing that the trend line models all the data perfectly (100%).

As theorised in Section 2.2, experimental testing of the data obtained verified this relationship as shown in Figure 3, where the equation for the trend line is y = 0.25x2 - 0.9908x + 355.11.

By taking the highest power of the polynomial and ignoring the rest, we are left with y = 0.25x2. It can be seen that the data gained verifies this relationship for the average-case efficiency.

## 4.4 - Average-Case Execution Time

The insertion sort function used in finding the average-case execution time is different than that used in average-case number of basic operations. In order to reduce overhead/extra execution time, the functionality of keep tracking and returning the number of comparisons were stripped into what would be a bare bones insertion sort.

Initial testing showed a fair bit of variance in execution time as the execution time is fairly dependant on the system in which its performed on, as there are other processes taken up by the CPU at the same time. To minimize the effect of other programs on the computation, all non-essential programs were shut down to improve data validity. The initial trend still showed that there was a quadratic increase in execution time respective to the size of the input as it has in Figure 2. In order to produce a cleaner graph, the sampling period was increased as see in the code in Section 6.7.2.

Cross referencing this with the data gathered from testing, as seen in Figure 2 it can be further reinforced that time it takes to execute the insertion sort algorithm is proportional to the square of the size the input array. Thus following a second order polynomial or as theorised in Section 2.3.

# 5.0 – References

|  |  |
| --- | --- |
| [1] | “Algorithm Analysis,” Worchester Polytechnic Institute, [Online]. Available: http://web.cs.wpi.edu/~mebalazs/cs507/slides02/slides.html. [Accessed 13 4 2016]. |
| [2] | “Median Sort,” Safari Books Online, [Online]. Available: https://www.safaribooksonline.com/library/view/algorithms-in-a/9780596516246/ch04s03.html. [Accessed 2016 5 25]. |

# 6.0 – Appendix

## 6.1 – Brute Force Median Pseudocode

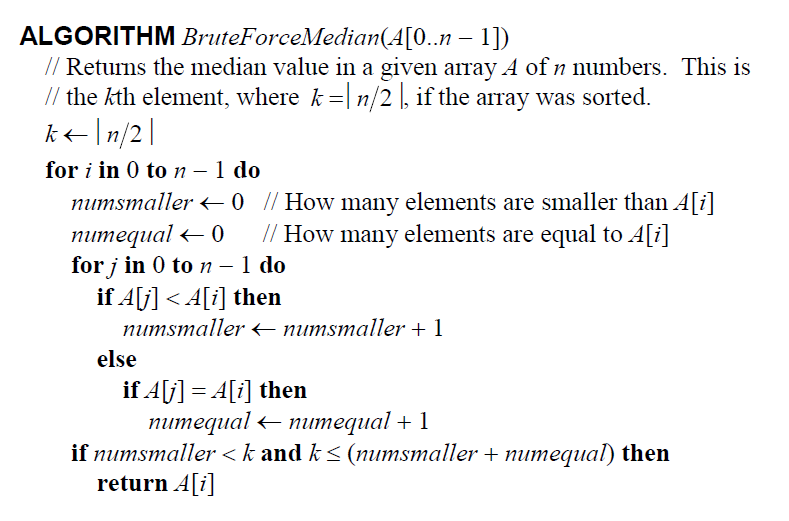


Figure 1: Pseudocode of the BruteForceMedian algorithm.

## 6.2 – Select-Partition Median Pseudocode

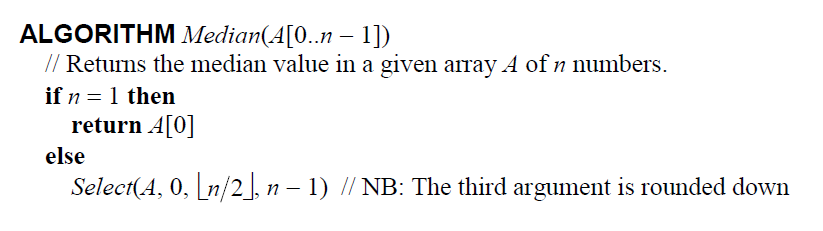


Figure 2: Pseudocode of the Median function for the Select-Partition Median algorithm.

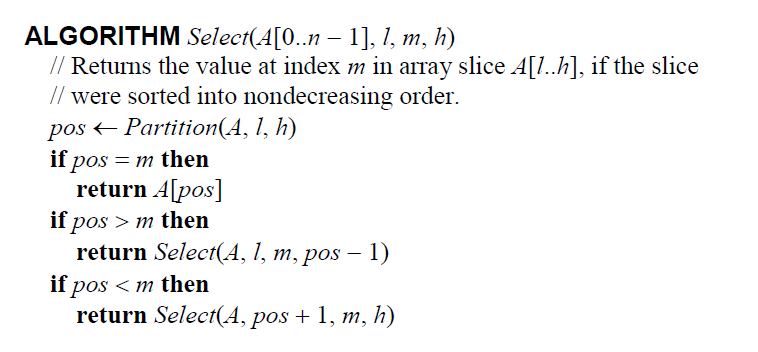


Figure 3: Pseudocode of the Select function for the Select-Partition Median algorithm.

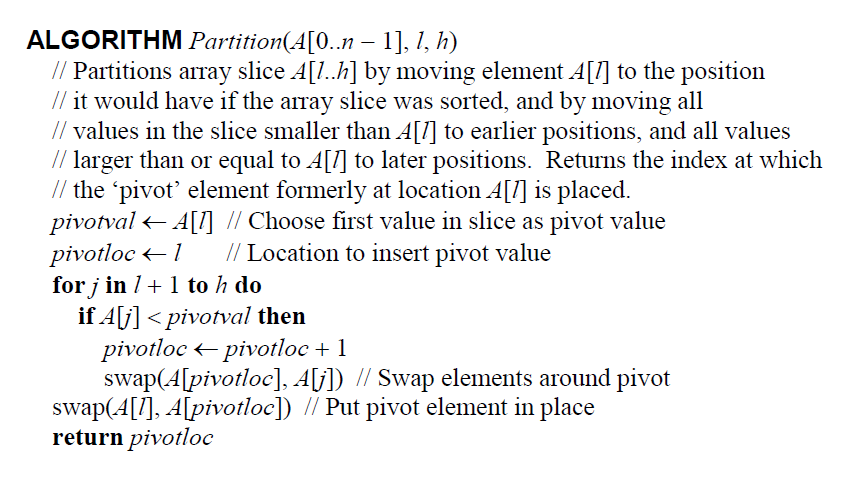


Figure 4: Pseudocode of the Partition function for the Select-Partition Median algorithm.

## 6.3 – Graphs

### 6.3.1 – Average Execution Time Graph for Brute Force Median Algorithm

### 6.3.2 – Average Execution Time Graph for Select-Partition Median Algorithm

### 6.3.3 – Normalised Execution Time Comparison

### 6.3.4 – Average-Case Efficiency Graph for Brute Force Median Algorithm

### 6.3.5 – Average-Case Efficiency Graph for Select-Partition Median Algorithm

### 6.3.6 – Normalised Number of Basic Operations Comparison