**CAB301 Algorithms and Complexity**

**Assignment 2**

**Empirical Analysis and Comparison for Algorithms**

**Student names:**

**Eliot Wilson**

**Cosmo Gregurek**

Submission Date: 22nd April 2016

**Table of Contents:**

1. Algorithm Description

1.1 Brute Force

1.2 Partition

1.2.1 Median

1.2.2 Selection

1.2.3 Partition

1. Expected Case Efficiency

2.1 Basic Operations and Efficiency

2.2 Problem Size

1. Experimental Methodology
2. Results

4.1 Proof of Correctness

4.2 Algorithm Analysis

4.3 Overheads and Conclusion

1. References
2. Appendix

**1. Algorithm Description**

In this report, the two algorithms used for analysis were the Brute Force algorithm and the Partition algorithm. The algorithms analysed share the same intended purpose, although each differ in terms of approach. Both algorithms calculate the median of any given integer array, however with varying efficiencies and functionalities.

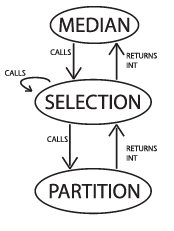
1.1.  **Brute Force**

The Brute Force algorithm is an iterative algorithm which considers all elements when attempting to find the median. Unlike the Partition algorithm, the Brute Force algorithm does not rearrange the array, rather it runs through each element and counts how many other elements are lower and / or equal to it in value. The algorithm analyses two aspects to determine the median; the elements with a smaller value than the comparison value in the array, and the numbers equal to that value.

If the numbers smaller than the comparison value encompass less than half of the array and the combined total of the both numbers smaller and equal to it is also greater than half of the array, then that number is returned as the confirmed median.

Due to the nature of the algorithm, any division performed to half the array is inherently ceiled (rounded up).

1.2. **Partition**

The second algorithm is segmented into three sections and will be known the Partition algorithm. The Partition algorithm analyses one half of an array of integers and incrementally re-orders them in order to determine the middle value. This algorithm is a recursive algorithm which consists of three main components; one which determines the operation based on the size of the array (Median), another component which assesses the found Partition value to determine the median, and a component which reorders the array to allow for the median (Partition).

**1.2.1. Median**

This component checks the length of an array to determine whether the median is either the first element or whether it is required to call the selection component. Unlike the Brute Force algorithm, the nature of this algorithm sees any division performed to half the array as inherently floored (rounded down).

**1.2.2. Selection**

This component immediately calls the Partition component to determine a value for examination. The value is then compared to the positioning in the array. If the value is equivalent to the middle, then it will return the value as the median, otherwise the position value would be incremented or decreased and the Selection algorithm is called again with this new position value. This is where the recursion occurs.

**1.2.3. Partition**

This component incrementally swaps the position of the elements in the array if they are less than the specified pivot value. Through the recursion, this puts the element with the middle element in the median position and then returns that value through the functions.

**2. Expected Case Efficiency**

2.1. **Basic Operations and Efficiency**

Within the two algorithms, the shared purpose was used to determine the basic operation of both algorithms. The comparative statements are used to determine the median value and were therefore used as the basic operations of the algorithm.

**Brute Force -** For code of the algorithm used, see Appendix 1.1

Within the Brute Force algorithm, the segment responsible for determining the median can be seen as the comparison between the selected comparative value and the rest of the array.

**if(theArray[j] < theArray[i]){**

As this operation would be performed for each element in the array, this would indicate a quadratic rate of growth for the number of operations performed.

**Average Efficiency - C Average (n ^2)**

As the average case efficiency is based upon these operations this indicates that the average efficiency for the Brute Force algorithm is bound to **Θ(n2).** This is also confirmed by Levitin (2012, p 70), who states that “Typically, characterizes efficiency of algorithms with two embedded loops”. Which matches the Brute force algorithm criteria, as there are 2 for loops inside the algorithm.

**Partition -** For code of the algorithm functions used, see Appendix 1.2

Within the Partition algorithm, the basic operation of the algorithm falls within the third component which is called upon in each recursion and would be run once for each corresponding element. The operation and efficiency of the algorithm are implied by Levitin who states the aspects of recursive algorithms contain (2012, p 70) *“The basic operation of the algorithm is multiplication, whose number of executions we denote M(n)”.* This can confirm the basic operation as:

**if(testArray[j] < pivotval){**

**Average Efficiency - C Average (n)**

The average efficiency for this algorithm is implied by the qualitative number of operations in relation to the size of the problem. I.e., the efficiency can be implied to be linear in nature and bound to **Θ(n)**. This is in accordance again to Levitin’s proof of N efficiency which sees an (2012, p 59)*"Algorithms that scan a list of size n (e.g., sequential search) belong*(s) *to this class."*

2.2. **Problem Size**

As the found efficiency classes for each algorithm were of different growth rates, the problem sizes were limited based on the size of the higher rate of growth. This was done in order to gain reasonable results that were not too large to draw a comparison. i.e. the sizes of the arrays were large, but limited to 10000, incrementing 500 elements each time from a starting point of 500. This was done as the Brute Force algorithm values would be too large for larger past an array size of 10000, but required demonstrative change for the Partition Algorithm.

These experiments were run ten times per array size and averaged out before graphing to allow for the most accurate results possible.

**3. Experimentation Methodology**

Both algorithms were developed in separate C++ files using the CodeBlocks IDE. Although they were developed simultaneously on separate computers, the actual testing (and compilation of the code) was done on a single machine to allow for accurate data across the algorithms. This machine was a Surface Pro book 3 a I5 cpu and 4 gigabytes of ram.

As it is iterative in nature, the Brute force algorithm was developed as a single function whereas the Partition algorithm was created through three separate functions relating to each component to allow for the recursion of the algorithm.

The codes were altered during the testing and recording stages of the analysis to include methods of recording the operations and execution times, however the section that relate to the algorithm itself remained unchanged. For the code used for measuring basic operations see Appendix 3.1.1 (Brute Force), 3.1.2 (Partition).

The code responsible for measuring the execution time does so for the full functionality of the algorithm rather than each segment. For the code used for measuring executions times see Appendix 3.2.1 (Brute Force), 3.2.2 (Partition).

In both algorithms, the respective code responsible for measuring operations and initial testing was excluded when the execution time was recorded in order to reduce the overheads not present in the algorithm. The times taken for other operations, such as filling shuffling the arrays as well as outputs to text files, were not recorded.

There were two data series recorded for each of the algorithms, one series for the recorded execution times, and one for the basic operations. The recorded data of both algorithms was then placed into spreadsheets in order to allow for the algorithm comparisons.

**4. Results**

4.1. **Proof of Correctness**

Prior to the analysis of the algorithm efficiencies and collection of results, a proof of the algorithm’s functionality and could verify the results. The correctness was tested using small pre-set arrays of integers that was kept constant over each algorithm. These tests recorded the operation time, number of basic operations and the array both pre-sorted and sorted for both algorithms. The code used for testing is consistent with the code used in the later stages of the analysis, however, the testing stage saw the pre-set arrays being used rather than randomly shuffled arrays.

The first set of tests consisted of an array with the elements; 20, 80, 11, 100, 25, 90, 2000,5, 69,1337. The tests for the Partition algorithm sorts the array in order to determine the median, whereas the Brute Force algorithm does not re-order the array, rather just finds the median through increased operations.

Through the testing, it was found that the sorting was accurate for the intended purpose of the Partition algorithm as well the found median was correct for both algorithms.

The found operations for both algorithms can be verified by a comparison to the predicted case values.

**Brute Force expected operations for the given array of 10 elements (n = 10).**

It can be seen in Appendix 4.1, that the operations for this algorithm are run 90 times. This number is close to the predicted operations for an array of that size, however may be different as the median falls before the end of the array, which results in fewer operations, proportionate to the number of elements in the array, e.g.10 elements for an array of size 10.

**Partition expected operations for the given array of 10 elements (n = 10).**

It can be seen in Appendix 4.1, that the found operations for this algorithm is equal to 8 rather than 10. This is due to the first two elements in the array already being sorted and therefore the operation would not have to run. If these two elements were to be included, then the operations would equal the predicted case, indicating that the functionality of the code follow the parameters of the algorithm.

It was also tested if the algorithms were able to handle tests of duplicate, reverse, random and other pre-sorted arrays. These results can also be seen in Appendix 4.1.

As these operations were found to be accurate to the predicted case, the algorithm was ready to prove the efficiency in a practical application.

4.2. **Algorithm Analysis**

Based on the established efficiency cases, the algorithms predicted number of operations can be found for both algorithms.

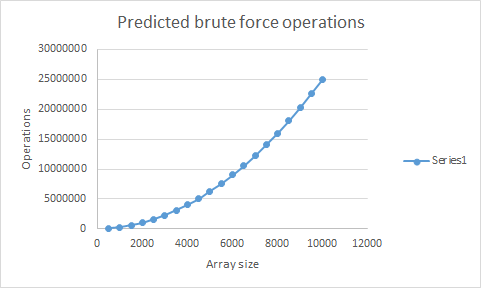
Using these predicted operations, an assessment of the algorithms practical application in terms of both execution time and operations can be compared to the theoretical case.

The Brute Force Algorithm was tested first as the algorithm calculates the median without disrupting the array. After the calculations of that algorithm have been completed, the Partition algorithm is executed using a consistent array as the previous algorithm and the results were recorded. The results below reflect the pattern of completion.

4.2.1. Predicted Operations

**Brute Force Algorithm Efficiency**

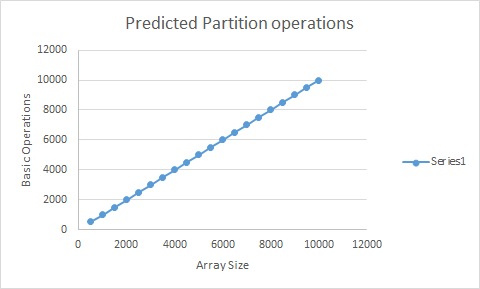
As the Brute Force algorithmwould be **Θ** bound by an efficiency of n^2, the average operations can be predicted as seen below.



The trend implied by this graph can be seen as a quadratic efficiency.

**Partition Algorithm Efficiency**

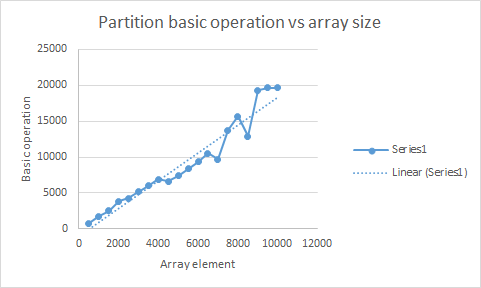
As previously established, the efficiency of the Partition algorithm is bound by **Θ**(n), which indicates a linear rate of growth. From the Levitin's findings of a recursive algorithm the graph below shows the predicted trend of the partition algorithm.



These predicted trends imply that the Partition algorithm is more efficient than the Brute Force algorithm, however, to determine the legitimacy of these predicted trends and draw a comparison between the algorithms efficiencies, a test of the algorithms was required. These tests were completed using the methods in Appendix 3.3.1 Also, as prescribed in Section 2.2, the experiment size was limited to 10,000.

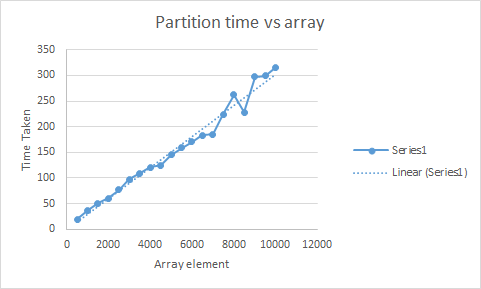
The testing was completed in arrays ranging in size from 1 to 10,000 incrementing in size of 500. The arrays were filled with integers from 1 to the length of the array and then shuffled. (For the code for shuffling these arrays, see Appendix 2.1). Once the testing phase was complete, the results were graphed for comparison.

4.2.2. Partition Algorithm



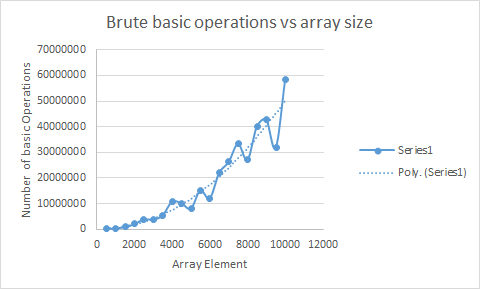
The above graph shows that the Partition Algorithm growth rate linear, based upon the average trend line shown, which confirms the prediction of **Θ**(n). Although the graph does show outliers, this could be due to computational errors by the computer. Another potential explanation for these outliers, may be the randomly shuffled arrays. This shuffling may have caused the elements in the array to be ordered in such a way that causes more operations representing a worst case, rather than the predicted average.

Alongside these operations, the execution time of the algorithm was recorded and appears to follow the same trend as shown in the below graph.



The graph shows the time taken for the Partition Algorithm for each test cases average. The data demonstrates that the execution time increases in a linear fashion, which implies proof of the predicted efficiency of this algorithm. This efficiency allows the algorithm to have a run time closely proportionate to the problem size.

4.2.3. Brute Force Algorithm

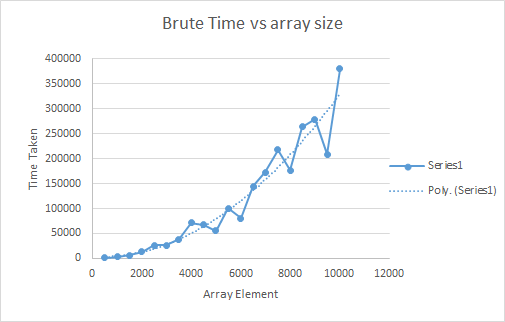


The above graph demonstrates the average operations for the varying array sizes when executed with the Brute Force Algorithm. It can be noted that the trend line shown matches the predictions and confirms the algorithm rate of growth as quadratic, which matches to **Θ**(n^2).

Due to the quadratic nature of the growth, the efficiency of the algorithm does decrease as the array size increases, most notably from the approximate array size of 6000 onwards. The data also shows outliers, which are independent of the Partition operation outliers, indicating that neither of them are due to shared logical coding errors rather computational or functionality differences.

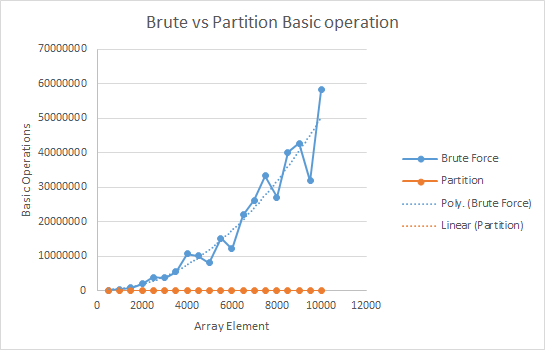
These outliers could be from the randomly shuffled array having the median value closer to the start of the array, which means the algorithm may not have to consider as many the elements.

Like the Partition algorithm, the execution times were also measured for analysis, which have been mapped below. Similarly following the trends of the basic operations, the trend line created again proves that the algorithm is quadratic which matches the efficiency, **Θ**(n^2).

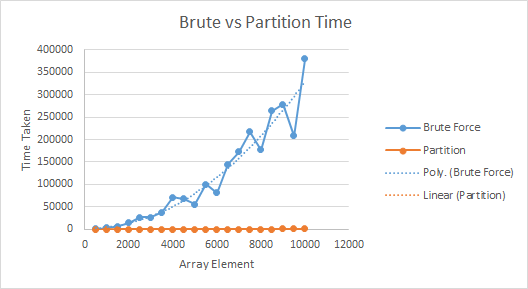


4.2.4. Comparative Analysis

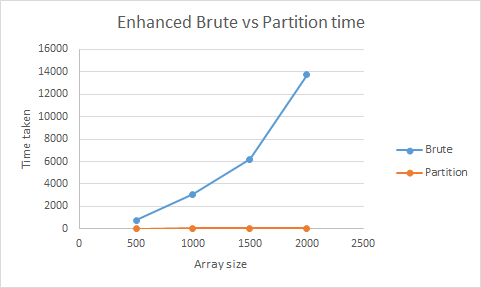
With the data collected and analysed for both algorithms, a comparative analysis case can be made.



The graph above shows a comparison between both algorithm’s basic operations in Microseconds. It can be seen that the linear efficiency of the Partition algorithm is more efficient than the Brute Force for problems of the given scope. Notably, the efficiency is seen to start differing at a greater rate from the array size of 2000. From this graph, it appears that the Partition algorithm is more efficient for large sized problems than the Brute Force, however, in order to confirm this, a comparison of the execution times for both algorithms had to be made.



The graph above allows for a comparison of the executions times of both algorithms. As can be inferred by the efficiency classes and seen on the above graph, the Partition algorithm’s linear efficiency is faster than the Brute Force algorithm for completing the tasks in the given arrays. It can be noted however, that up to the array size of 2000, the execution times are similar in growth and therefore may indicate an efficiency for smaller sized arrays. The graph below demonstrates the array size up to 2000 elements in order to gain a clearer sense of the executional differences.



A comparison between the comparative execution time graphs demonstrates a much clearer indication that the efficiencies differences are notable for arrays of both larger and smaller sizes, although as these execution times are in microseconds, it may be unnecessary in a practical sense to distinguish between algorithms for arrays limited to this size.

4.3 Overheads and Conclusion

Both algorithms had outliers present during the testing and this is likely be due to computational errors of the machine used as the outliers are not consistent across algorithms. The presence of a substantial outlier can skew found average, however multiple tests were used to combat this effect. Another potentially key contributor to the positive outliers (optimisations) is the order in which the arrays were shuffled. If parts of the array were already sorted, the Partition would quickly skip over those values representing a best case scenario. This can potentially be present as represented in the above graphs. If an array was completely unsorted the efficiencies would be closer to the worst case scenario, increasing the execution time. The Brute force algorithm could quickly end if the median number was located randomly at the front of the array, which is one reason why the brute force algorithm can be an alternative to the Partition for small array sizes, but this is very unlikely and subsequently decreases in probability as the array size increases.

In terms of consistent efficiency, it has been confirmed the Partition algorithm can be seen to be more efficient for determining the median than the Brute Force Algorithm in terms of both operations and execution time, although for arrays of smaller size, the differences may practically be negligible for modern computers with the execution times being so low in microseconds.

5. References

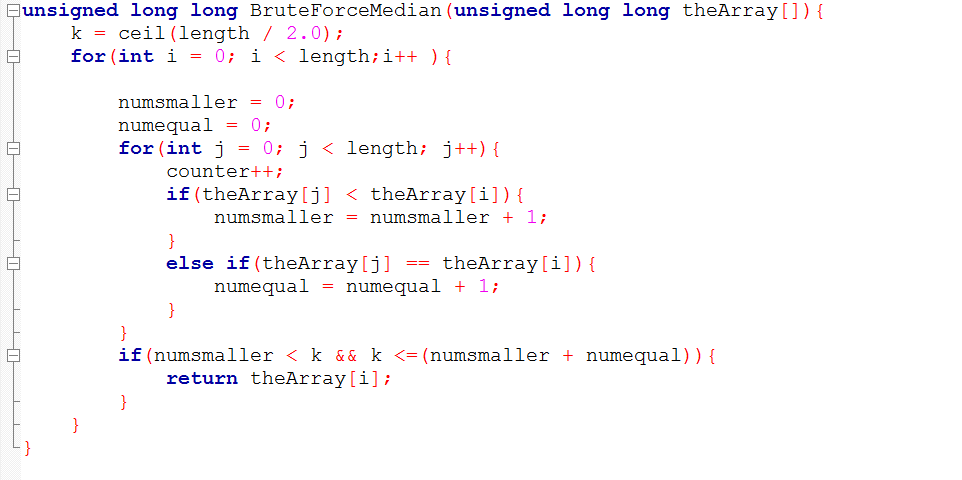
[1] Microsoft. (2016). *QueryPerformanceCounter Function*. Retrieved from<https://msdn.microsoft.com/en-us/library/windows/desktop/ms644904(v=vs.85).aspx>

[2] A. Levitin. *Introductions to the Design and Analysis of Algorithms*. Addison-Wesley, third edition, 2013. ISBN-13 978-0-13-231681-1

6. Appendix

1.1 Brute Force Algorithm Code

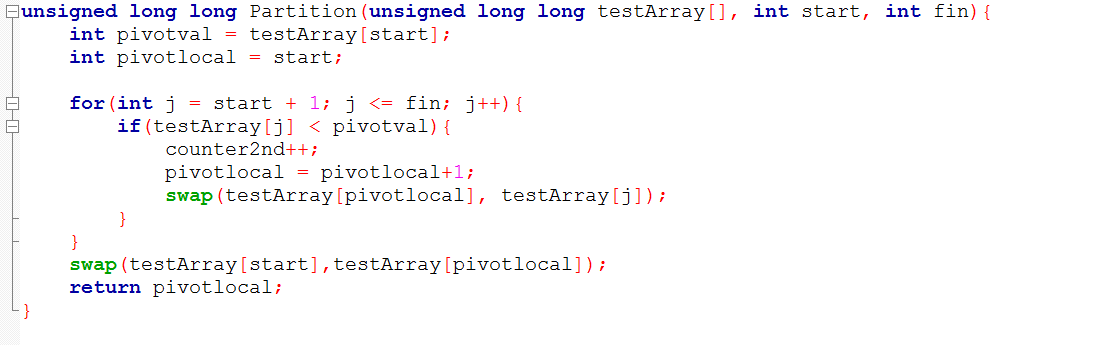
The code below is the implemented algorithm for the Brute Force in C++ code. This code takes an array of integers and uses two for loops to cycle comparisons through each element in order to determine which value in the array is the median.



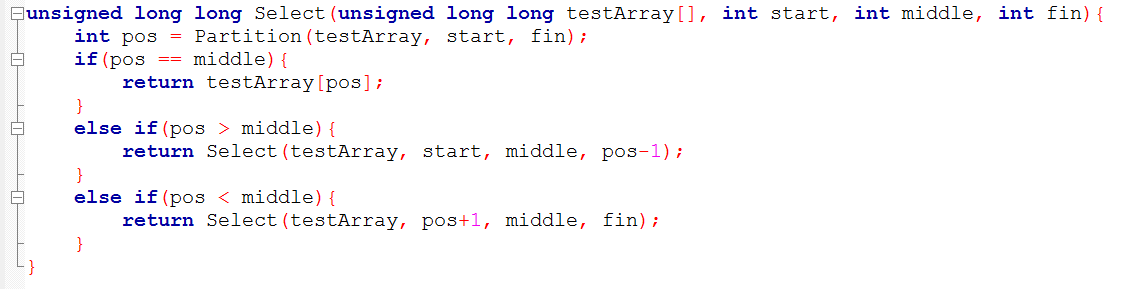
1.2 Partition Algorithm Code

The code for the Partition Algorithm is segmented into three separate functions as outlined below.

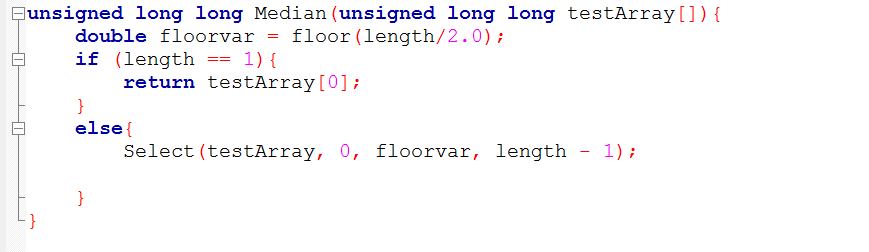
This segment of code compares the start element passed through the other functions and if the values are lesser than the ones compared, then the pivot point is incrementally changed and the found integers are swapped. This is repeated until the comparative integers are of equal size and can be determined as the middle position of the array.

The counter included in the body of the code is separate to the algorithm, however was included for the purpose of calculating efficiency. 

The code below is for the Select function which is called within Median, first is requires an array, and 3 integers; One for the start, then middle and lastly an index integer. An integer pos is created equalling what is returned when partition is called. These integers are compared through a series of if statements to determine if the value is either less than, greater than or equal to the middle value. If the chosen value is greater or less than the middle value, a recursion is called with a new segment of the array.

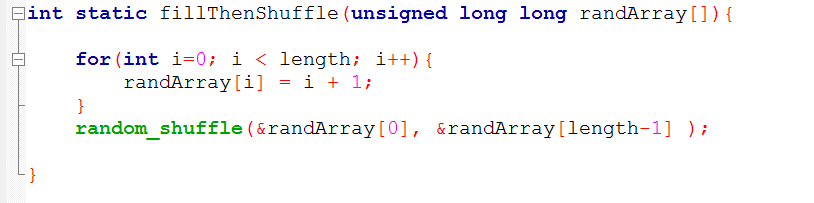


The code below is for the Median method which is responsible for returning the found median number. This segment uses variables passed through the previous functions in order to determine if a value is the median after re-arrangement.



2.1 Code for Shuffling Arrays

Below is the code that created an array based upon the chosen length and fills it with elements corresponding to the position. The array is then shuffled using a C++ native function, random\_shuffle. This shuffles the positions of each element to prepare for the algorithm execution.



This segment is called prior to each algorithm execution.

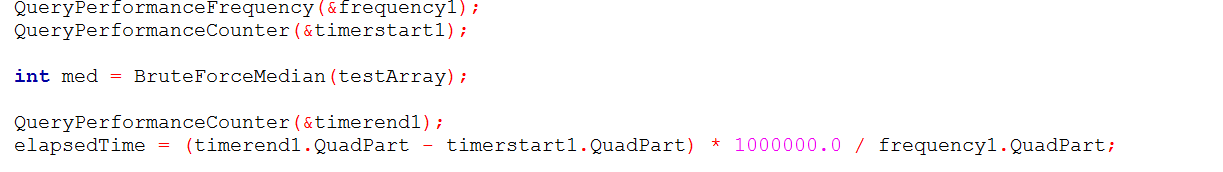
3.1.1 Code for Measuring Basic Operations - Brute Force

For counting the basic operation, a variable counter was placed to proceed the basic operation in each algorithm. The Brute force counter was placed in the second for loop, for every time it must be run again the counter goes up.

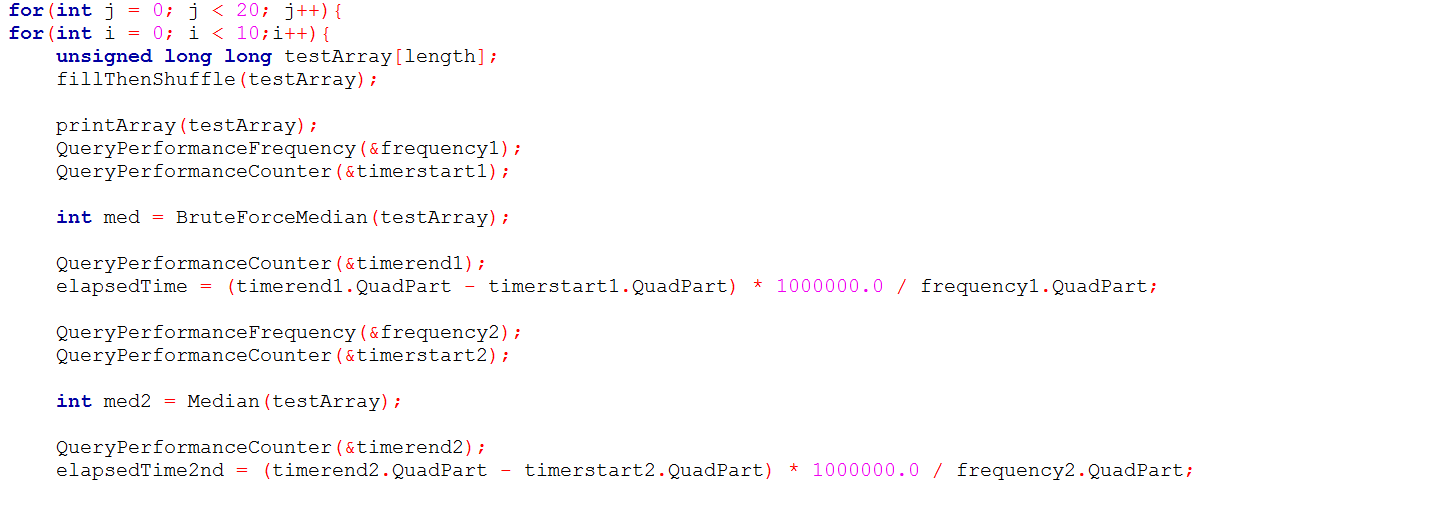
3.1.2 Code for Measuring Basic Operations - Partition

The same was done for the second algorithm, the counter as placed in the partition algorithm, to count how many numbers are swapped till the array is ordered.

3.2.1 Code for Measuring Execution Time - Brute Force and Partition

When measuring the execution times, a Windows native function for measuring execution time called *QueryPerformanceCounter* (Microsoft, 2016)*.* This function takes two points within a program and measures the system ticks in order to determine the total run time of that segment. For this analysis, all run times were calculated in Microseconds. First the Query Performance counter must be opened, then the brute or partition method is run then the timer must be close. The time is converted to milliseconds and placed into the variable elapsedTime to be printed to either the screen (When in the initial testing phase) or the data file.

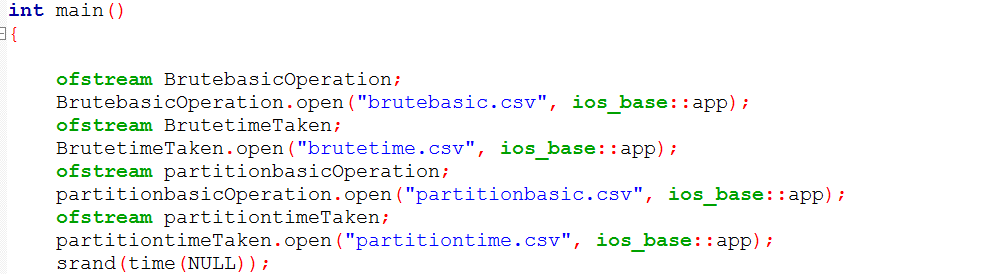
3.3.1 Code for Main

Below will be the code for main, which is the segment responsible for calling each algorithm function.

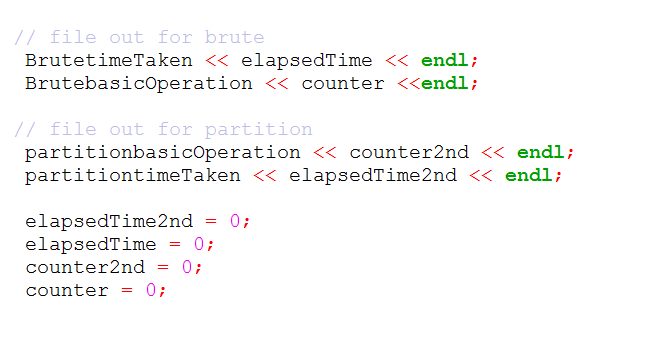
This segment calls all algorithm functions as well initiates and calculates the executions times of each function.

For the execution of the function, the sections responsible for printing to the console were commented out to increase computer efficiency.

This below segment is responsible for outputting all recorded information to different text files.



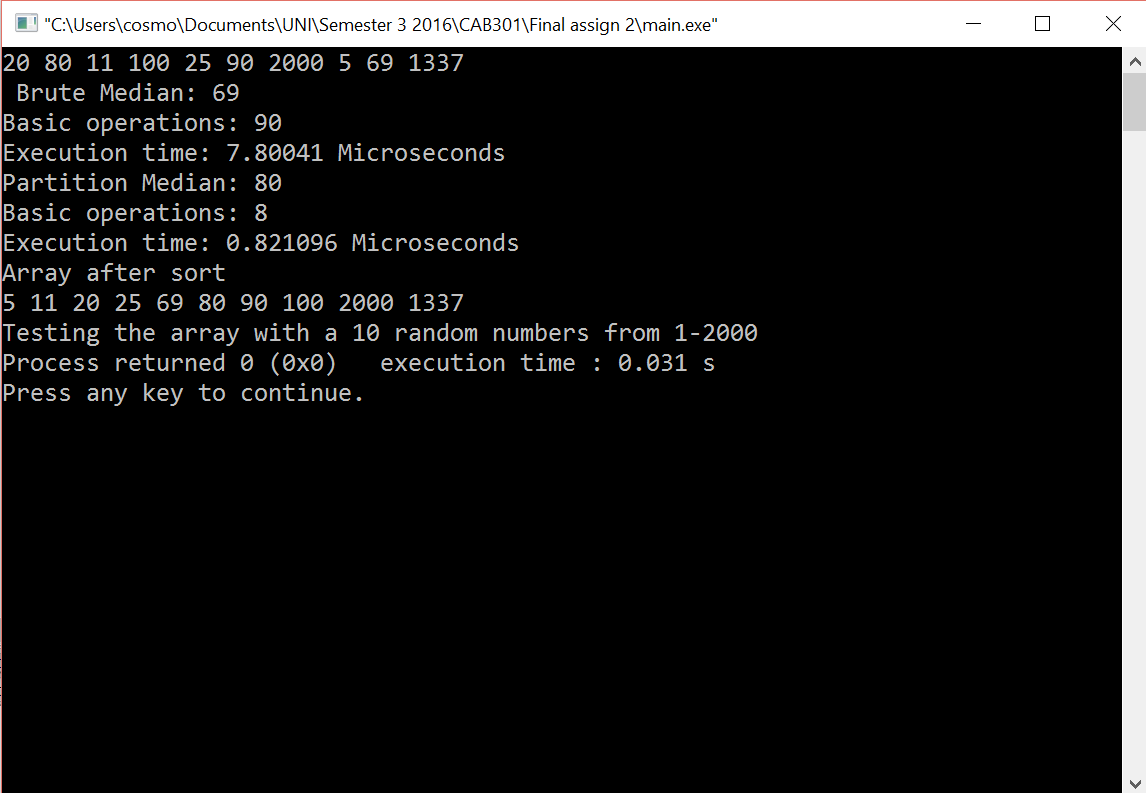
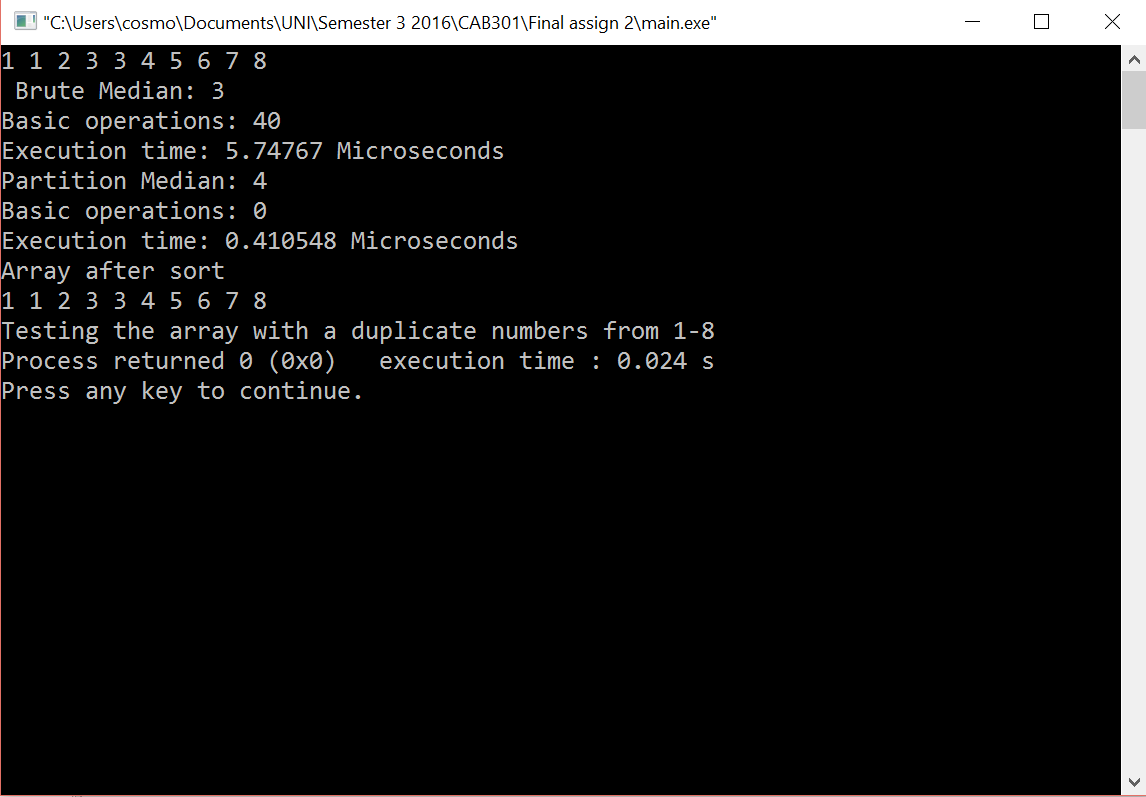
The code block below is for outputting the variables to the excel file, and also resetting the variables. The variable counters were reset to 0 in order to allow for each array counter to not be skewed by previous operations.



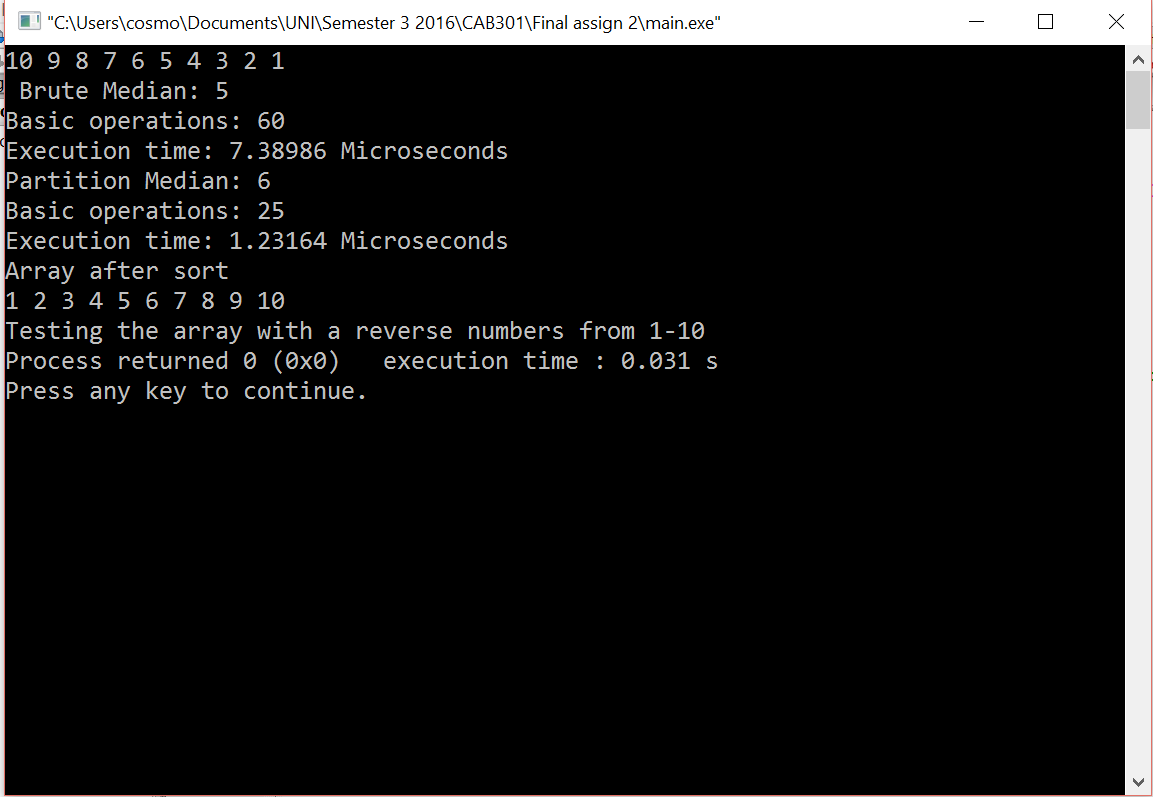
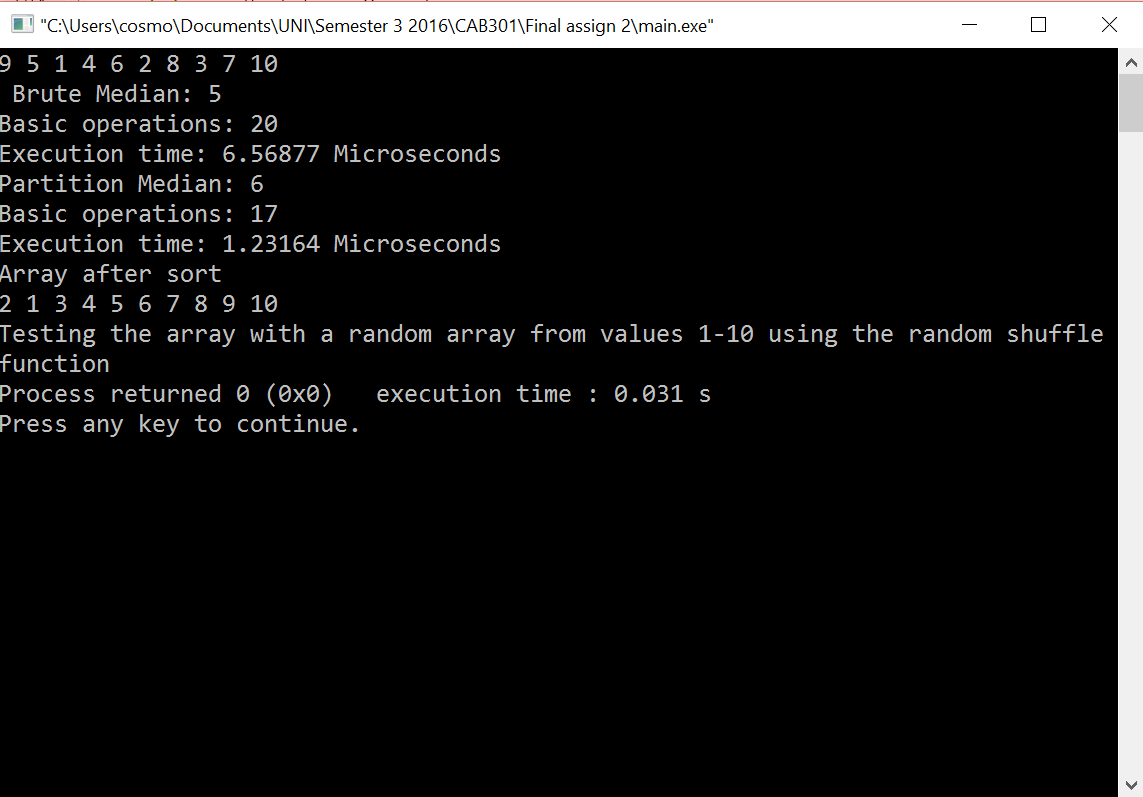
4.1 Testing Array Results

These tests were completed using consistent arrays for each algorithm to ensure unbiased data.

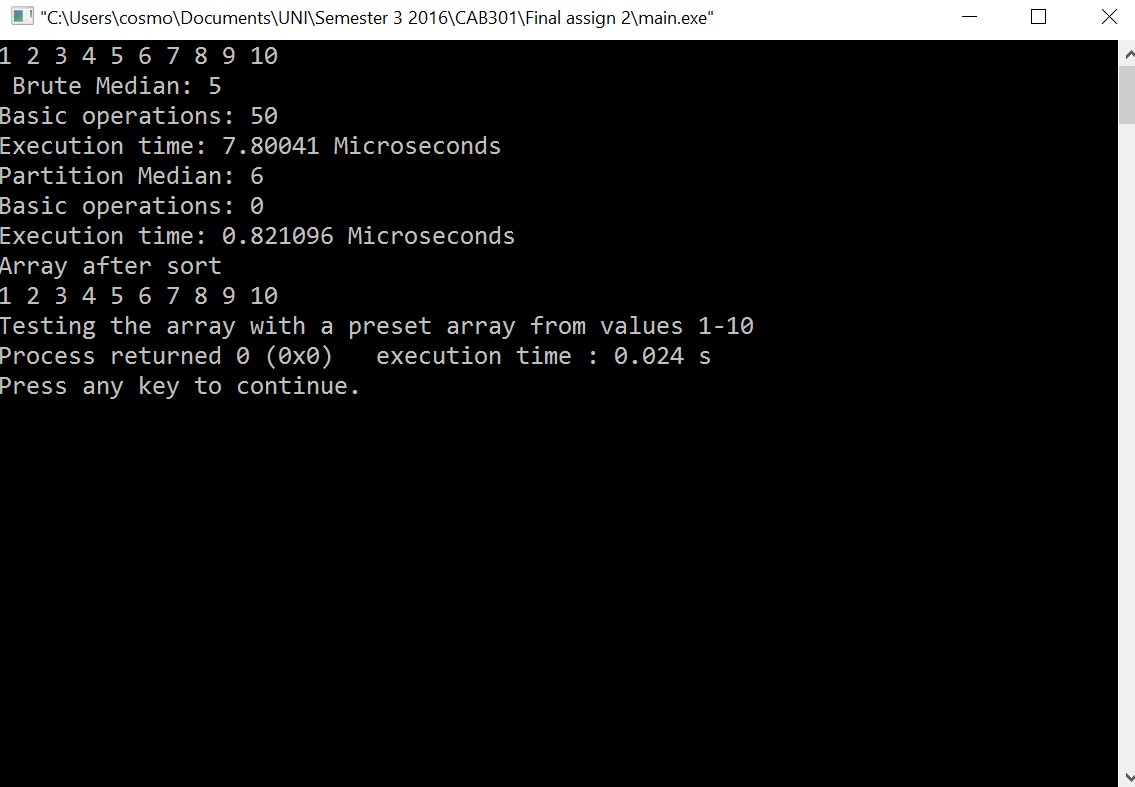
4.1.1 Testing Array 4.1.2 Duplicate Arrays



4.1.3 Random Array 4.1.4 Reverse Array



4.1.5 Pre-Sorted Array



4.2 Normalized Data

Below are normalized versions of the graphs found in the graph, Normalization is to bring the scale of the data from 0 - 1, this is done by x - lowest value / highest value - lowest value.