CAB301 Algorithms and Complexity

Assignment 2 Empirical Analysis of Two Algorithms

Alexander Milne & Matthew Price

N9687157 & N9184058

Summary

The following report explains the results of tests conducted on two algorithms which find the median number in an array of any given length. Each test conducted on the algorithms will determine which is the most efficient regarding average time taken and average number of basic operations used to find the median. To test most efficiently and create reputable results each test case was identical with computer environment and array used. This allows for direct comparisons to be drawn between the results from both algorithms and determine which of the two is faster.

1 Description of Algorithms

The first algorithm to be introduced is the Brute Force Median algorithm. This algorithm takes an integer array of any length to be named array A as shown in statement a. The algorithm returns a single integer which is the median of the array. The algorithm begins the process by taking the length of the array and halving it, this value it has calculated is then set to integer k as shown in figure 4 (statement b). It should be noted that for arrays with an equal number of values the median would usually be sum of the two middle numbers in a sorted list and then divided by two. Because the algorithm finds the median from the list of numbers it does not account for this so out of the possible two middle numbers this algorithm chooses the first one of the two as shown in figure 8. The algorithm then begins a for loop which starts at i = 0 and ends at the length of the array (n-1), this is because the steps to follow are completed for every element in array. Two integers are created as numSmaller and numEqual as shown in figure 4 statements d and e. A nested for loop then begins, starting at j = 0 and ending at the length of the array (n-1).  The value in A[i] is compared to A[j], if the result of this comparison shows A[i] is greater than A[j] then the counter numSmaller is increased by one, if the result displays A[j] is greater than A[i], the algorithm moves to the next value to compare A[i] to and finally if the values in A[i] and A[j] are equal the counter NumEqual is increased by one as shown in figure 4 statements g to k. This comparison process is repeated, comparing every number in the array A[j] with whichever current value A[i] is at. Once the algorithm has exhausted its list of numbers, the counters are compared to the integer previously stored known as k (statement i, figure 4). If the algorithm finds that the value of numSmaller is less than k and the value of numSmaller + numEqual is greater than k then it returns the value in A[i] as the median. If the comparisons of numSmaller and numSmaller + numEqual to k do not evaluate to true, the algorithm moves to the next value in the array A[i + 1] and then repeats the whole process. This method is repeated until the algorithm finds the median.

Secondly the Median algorithm is to be explained and detailed to show how it finds the median in an integer array A of any length n. This algorithm uses three separate methods to find the median in any given array. The first method called Median takes an integer array of length n as shown in figure 1 statement a. The method then checks to see if the array has more than one element in it and if it does not, the algorithm returns the only value in the array. If the array has more than one value the method moves to the next course of action and calls the second method in the algorithm called Select as shown in figure 1 statement e. Select takes 4 arguments, the first being the array in question which is followed by the integers l, m and h to represent the low point it will partition from, the middle point in the array and the high point it will partition from. When first called from Median, the Select method takes the array, 0 for the low point, the array length divided by two (n/2) and the array length (n-1) as shown in figure 1 statement e. The Select method then creates an integer and sets its value by taking the array, low point and high point to call partition (figure 2, statement b). Partition takes 3 arguments, the array and then the low and high points to be partitioned between. The Partition method creates two integer variables, the first is called pivotval and is equal to A[l] (figure 3, statement b) which in the first call of partition is A[0]. The second variable is called pivotloc and is given the value of l (figure 3 statement c). The method then begins a for loop beginning at l + 1 and ending at h. In the first run of the algorithm h would equate to the final value of the array and because l is 0 it is moving from A[1] to A[n-1]. The algorithm compares the value of pivotval to A[j]. If the value of A[j] is less than that of pivotval, the two values are swapped and pivotloc is incremented (figure 3 statements e, f, g). if pivotval is greater than A[j], the algorithm moves onto the next value in the array A[j+1]. This process is repeated until j is greater than h. Once j is greater than h, the array swaps A[l] with A[pivotloc]. This is important because it places the value of A[l] in the position where it should be if the array was sorted. The array then returns pivotloc. That value then becomes pos in the Select method and is compared to m. If pos is equal to m, the value is returned as the median of array A, if it is larger than m, the array is sliced by recalling Select and modifying the value of h and as a result the first half of the array partitioned again. If the value of pos is less than m, the array is sliced by calling Select again and modifying the value of l so when the Select calls Partition it only Partitions the second half of the array. This whole process is repeated until the median is found.

2 Theoretical Analysis

This section discusses the choice of basic operation, problem size, and average-case efficiency for each algorithm.

2.1 Basic Operation

The basic operation for the BruteForceMedian algorithm is the comparison on line 175 in appendix B. This has been chosen as the basic operation due to it being the primary comparison used for determining whether a given element is the median of the array. This comparison is used to compare each item to each other item, and is part of the algorithm’s innermost loop [1, p. 44].

The basic operation for the Median algorithm is the comparison on line 122 in appendix A. This has been chosen as the basic operation due to it being the primary comparison in this sorting function. This comparison is used to sort within each partition, and is part of the algorithm’s innermost loop [1, p. 44].

2.2 Problem Size

The problem size for each of the algorithms is N, where N is the number of items in the input array. These algorithms fall under the category of search algorithms, for which this problem size is typical [1, p. 43].

2.3 Average-Case Efficiency

Average-case efficiency for algorithms can be found by averaging the best and worst-case efficiencies of that algorithm. In the BruteForceMedian algorithm’s worst-case efficiency scenario the algorithm will run its basic operation a number of times equal to the length of the array for each number in that array, leading to a worst-case efficiency of N**2**, where N is the length of the array.  The best case efficiency for this algorithm is if the median value is the first item in the array, but the algorithm will still compare this item to each other item in the array. This leads to a best-case efficiency of N. The average of these leads to an average-case efficiency of (N**2** + N) / 2, which is a quadratic efficiency.

In the worst-case efficiency scenario for Median, the algorithm will run it’s basic operation on each element of the input array with each recursion running this operation on half as many values, resulting in a worst-case efficiency of N**2**. In the best-case efficiency scenario for Median, it will perform it’s basic operation on only the first partition of length N, resulting in a best-case efficiency of N. The average of these leads to an average-case efficiency of Ө(N**2** - N + N) / 2.

3 Methodology, Tools and Techniques

3.1 Computing Environment

For the tests to be conducted a single computing environment was used. The computer used is a Dell XPS 15 laptop using windows 10. The system utilises 16GB of ram with a quad core i7 6700HQ at 2.6GHz processor for computing. The Program used is Microsoft Visual Studio 2015, the program is widely used and is free to the public.

3.2 Implementation

Using the environment outlined above the pseudocode for both algorithms was written in C# because it is so familiar to the team. It is taught thoroughly at the institution QUT and as a result should be familiar to the research team. The code for the Basic Operations and time tests were created in an environment with the random class constructor to generate multiple arrays with various random numbers for the tests. The stopwatch class was then used as well for the time testing scenarios as it tracks the time taken between using the start method and stop method from the class.

3.3 Generating Test Data

As mentioned to record time taken for each algorithm to compute arrays the constructor class Stopwatch was used. For each array either algorithm found the median in, the time taken was printed alone into the console window, this allowed for the data to be collected efficiently once it had been generated. This is shown in Appendix E (lines 314-319 and 324-329). A similar process was used for the basic operations, once the an algorithm had completed its task, the number of basic operations was printed as displayed in Appendix D lines 352-354 and 363-365. In order to actually calculate the basic operations, a method to use identical but separate methods to the original with a global counter called basicOps. The counter is placed in the duplicate algorithms and set to increase by one when the correct aspect of code is reached in the respective algorithms BruteForceMedian2 or Median2. This counter is initialised as 0 but because it is global it needs to be reset before the algorithm is used to prevent previous tests from compromising the integrity of the results. The algorithm achieves this in appendix D lines 352 and 362.  The data collected from any test conducted like this was then copied into Microsoft Excel where a spreadsheet was drawn up to record the data. The average from each test was calculated and recorded to be used in graphs also created in Excel. The graphs and the report from the tests was then created collectively between Microsoft Word and Google Docs. Microsoft Word was used to begin with as it is a premium document creating program and is the best around, as it is a team exercise however, the use of Google Docs allows for multiple users to edit documents at a time. This makes version control very simple and elegant and allows for quicker development of the report.

4 Experimental Results

4.1 Functional Testing

The algorithms were implemented into C# as similarly to the pseudo code as possible to ensure the results from the time tests and the basic operation tests are valid and useful. In order to justify the implementation, six tests were created to judge how well the algorithms chose the median of a small list of varying numbers and orders. The tests were used to test under different conditions, a sorted array, an array of length one, an array not sorted, an unsorted array with positive and negative numbers, an array with positive and negative numbers with a duplicate number in the list and finally an array with even length containing unsorted values. To make the tests as fast as possible the algorithms were run one after the other. The BruteForceMedian had to be run first because it does not change the positions of any values in the array whereas the Median algorithm does. This means it would change the test conditions for the BruteForceMedian if they used the same array if Median was run first.

As shown in figure 5 the array begins at 1 and finishes at 7, the brute force median collects the median to be 4 as does the Median algorithm which was the correct value.

Secondly the array of length one uses value 65 and both algorithms correctly select and print the median as 65(figure 6).

Thirdly the array is unsorted and random to test the algorithm works with arrays not in order. The BruteForceMedian and Median algorithms collect value as shown in figure 7. The array is then sorted to prove that the median is indeed 4 which it is.

The fourth test is used to test an unsorted array with negative and positive numbers. The result as seen in figure 8 is a median of 4 and is correctly chosen by both algorithms.

The fifth test is used to test if duplicate numbers has an affect on which number the algorithms choose to be the median. As seen in figure 9, the algorithms both choose -2 as the median which is correct as shown in the sorted version of the array.

Finally the sixth test runs the arrays on a randomly generated, unsorted array with an even number of values as it is of length 8(figure 10). As mentioned this is a curious test because the algorithms are meant to choose different values. Because arrays with even lengths have two numbers as the median technically, the algorithms choose one of the two, the  BruteForceMedian chooses the value that is less. In this case it is the value in the 4th position if the array were sorted. Whereas the Median method chose the value in the 5th position if the array were sorted.

4.2 Basic Operations Results

As described before in part 3.3 Generating Test Data, the algorithms were duplicated and then modified slightly to append a counter in the code for the basic operations tests only. This allowed for the basic operations to be tested quickly and easily without any issues.

In order to test the number of basic operations used by the algorithms an environment was created which constructs 100 arrays with a length which could be modified through the integer variable LengthOfArray. The tests which have been conducted began at length 1000 and ran through to 3000, incrementing by 200 values at a time, this created 11 tests for each algorithm(Appendix D). Each value for every array was generated randomly for the separate tests. In order to make sure the algorithms were tested under exactly the same conditions, each array had to be saved in order to be tested again by the second algorithm. This was not difficult as the array was saved just after being constructed so that no algorithm could impact it (Appendix D). To test the second algorithm it was then called in a separate for loop. When an array finished being tested and the median had been found, the number of basic operations was then printed on a new line on the screen, this made collection of data simple and elegant. It was this method which then place the data in spreadsheets. The graphs were then created from the average of each test where a point on the graph is the average number of basic operations used for an array of that length. This can be seen in figure 11 where the basic operations for the Median algorithm is displayed and figure 13 which outlines the average basic operations taken for the BruteForceMedian algorithm to find the median.

The results from these tests are displayed in figures 11 and 13. The Median algorithm displays a clear linear trend in figure 11 where as the length of array increases so too does the number of basic operations by an amount which is constant for the most part excusing the anomalies which occur simply because some of the test cases were lucky and multiple tests found the median very quickly which used less basic operations. The BruteForceMedian algorithm shows an irregular trend in figure 13 as many points do not follow the quadratic trendline directly. The beginning of the tests show compliance with the trendline but lose sense of direction in tests with 2200 and 2400 recording very strange results. The only explanation is that those tests were outliers as the rest of the tests show a general trend with a quadratic nature. It can be observed that the tests for 2200 and 2400 in the Median algorithm were also slightly irregular but not to the point where they are outliers. It is very clear that the more efficient algorithm for basic operations is the Median algorithm, finding the median with an average of 10,000 basic operations for an array of 3000 whereas the BruteForceMedian finds the median on average with around 2,750,000 basic operations(figures 11, 13).

4.3 Time Efficiency Results

Once again to create reputable tests as described in 3.3 Generating Test Data the algorithms for the basic operation tests were duplicates of the normal ones which left the unchanged algorithms for the time testing purposes. Before an algorithm was run on an array, the stopwatch was reset and then started for the test, after an algorithm had found the median it stopped the stopwatch and printed the time it took on a new line to allow fast and effective collection of data.

The results from the time tests are displayed in figures 12 and 14. The median algorithm once again shows a linear trend as opposed to the quadratic pattern displayed by the BruteForceMedian. The graphs displayed show that the time it takes for the Median algorithm to find the median is parts of a millisecond whereas the BruteForceMedian algorithm is taking 18-22 milliseconds to find the median in an array of only 3000 values which displays an average efficiency of n^2. The time taken for both algorithms to collect the median for the final test dropped in time by what seems like a very similar amount. This shows that the median was early on in the arrays for many of the tests and it is an outlier among the data set. The trend line used to indicate time efficiency matches the graph in figure 12 with the Median algorithm as it is of efficiency class n. Whereas the trendline for the BruteForceMedian algorithm is quadratic to match its efficiency.

5 References

[1]A. Levitin, Introduction to the design & analysis of algorithms. Boston, Mass: Pearson, Addison Wesley, 2008.

1. ALGORITHM *Median*(A[0-n])

    //Returns the median value in a given array A of n values

1. **if** n = 1 **then**
2. **return** A[0]
3. **else**
4. **return** *Select*(A, 0,n/2, n-1)   //NB: The third argument is rounded down

Figure 1: This is one of the algorithms in question, it takes an array of length n and returns the median. If n is equal to 1, it return A[0] as that is the median of the list.

1. ALGORITHM *Select*(A[0…n-1], l, m, h)

//Returns the value at index *m* from array slice *A[l…h]*, if the slice were sorted into

//non-decreasing order

1. pos ← *Partition(A, l, h)*
2. **if** pos = m **then**
3. **return** A[pos]
4. **if** pos > m **then**
5. **return** *Select(A, l, m, pos -1)*
6. **if** pos < m **then**
7. **return** *Select(A, pos + 1, m, h)*

Figure 2: This method is called originally by the Median method and returns the index m. Depending on the result from partition and the value of variable pos, the select method will call itself until the correct value for m is found.

1. ALGORITHM *Partition* (A[0…n-1], l, h)

//Partitions array slice *A*[*l…h*] by moving element *A*[*l*] to the position it would have if the array

//slice was sorted, and by moving all values in the slice smaller than *A*[*l*] to earlier positions, an

//all values larger than or equal to *A*[*l*] to later positions. Returns the index at which the ‘pivot’

//element formerly at location *A*[*l*] is placed.

1. *pivotval* ← *A*[*l*]   //Choose first value in slice as pivot value
2. *pivotloc* ← *l*    //Location to insert pivot value
3. **for** *j* **in** *l* +1 **to** *h* **do**
4. **if** *A*[*j*] < *pivotval* + 1
5. *pivotloc* ← *pivotloc* + 1
6. swap(*A*[*pivotloc*], *A*[*j*])   // Swap elements around pivot
7. Swap(*A*[*l*], *A*[*pivotloc*])        // Put pivot element in place
8. **return** *pivotloc*

Figure 3: This method is called at the start of the Select method in figure 2. Partition takes the value in the first position of the array and moves it to the position it would have if it were sorted by moving all the values smaller than it before itself in the array and all the values larger than it after itself in the array.

1. ALGORITHM *BruteForceMedian* (A[0…n-1])

    //Returns the median value in an array A of n values. This is

//the *k*th element, where k = n/2, if the array was sorted.

1. *k* = n/2
2. **for** I **in** 0 **to** n-1 do
3. *numsmaller* ←0     //How many elements are smaller than *A*[*i*]
4. *numequal* ←0        //How many elements are equal to *A*[*i*]
5. **for** *j* **in** 0 **to** *n-1* **do**
6. **if** *A*[*j*] < *A*[*i*] **then**
7. *numsmaller* ← *numsmaller* + 1
8. **else**
9. **If** *A*[*j*] = *A*[*i*] **then**
10. *numequal* ← *numequal* + 1
11. **If** *numsmaller* < *k* **and** *k* ≤ (*numsmaller* + *numequal*) **then**
12. **return** *A*[*i*]

Figure 4: This is the pseudocode for the BruteForceMedian. It takes an array of length n and sets an integer k to be half of n. The algorithm begins a for loop and creates two further integers. These are used later to determine if the number being analysed is the median. A second for loop is then begun to check all values in the array against the value in *A*[*i*]. if a number is smaller than *A*[*i*] the *numsmaller* increases by one value alternatively if the value is the same as *A*[*i*], *numequal* increases its value by one. Finally if the number of values that are less than *A*[*i*] is also less than half the array (*k*) and the number of values smaller than and the number of values equal to *A*[*i*] is less than half the array (*k*) then the value in *A*[*i*] is returned.

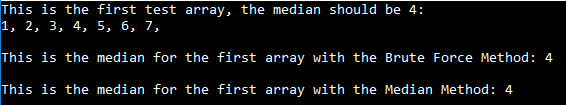
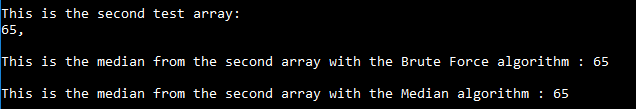


Figure 5: This image represents the results from the first test to justify implementation. The array created for the first test is displayed to show which number is the median and then the algorithms are run and a sentence is printed followed by the result of the test.

Figure 6: The image above displays the result from the second test to prove the algorithms have been implemented properly.  The array is first printed as it is of length one, there is only one possible result. A sentence is printed followed by the result of the algorithm being run.

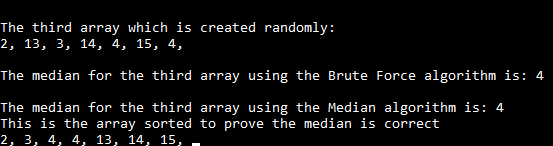


Figure 7: This figure presents the third test to justify the algorithms integrity. An array was generated randomly and is printed onto the screen. The algorithms are then run and a sentence describing the outcome is printed with the result of the algorithm. The array is then sorted to prove that the algorithms have chosen the correct value.

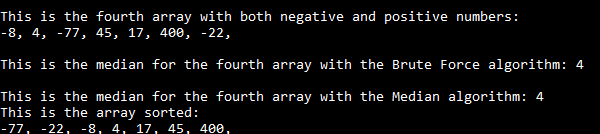


Figure 8: The image above displays the results from the fourth test for each algorithms integrity. A prebuilt array with both positive and negative numbers is used and printed out. The algorithms are then run and printed with a sentence to highlight their relevance. The array is then sorted to prove that the algorithms have retrieved the correct values.

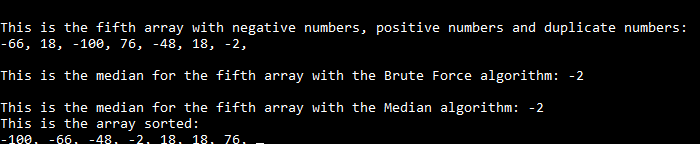


Figure 9: This figure displays the fifth test for the integrity of the algorithms. The array in the test is printed and has both positive and negative numbers and a duplicate number. The algorithms are run with this array and the result is printed with a sentence. The array is then sorted and the algorithms have collected the correct value from the array despite the duplicate numbers and the range from positive to negative.

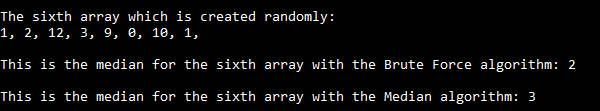
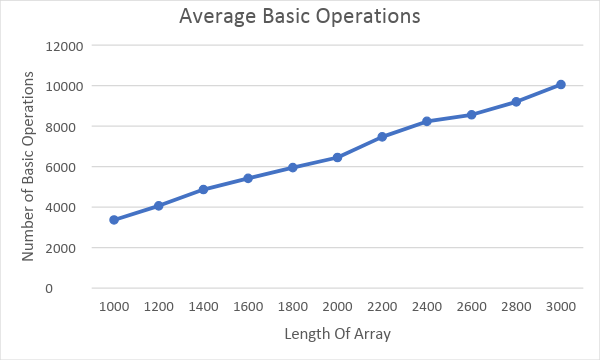
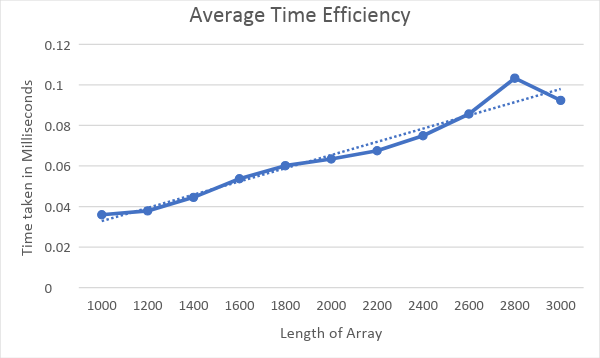


Figure 10: This figure displays the final test to justify implementation of the algorithms. it displays a randomly generated array of length 8. This is interesting because it shows the algorithms choosing different values for the median. This is because when the array is even there are two middle numbers rather than one. The brute force median algorithm chooses the lower of the two whereas the Median algorithm chooses the higher value of the two.



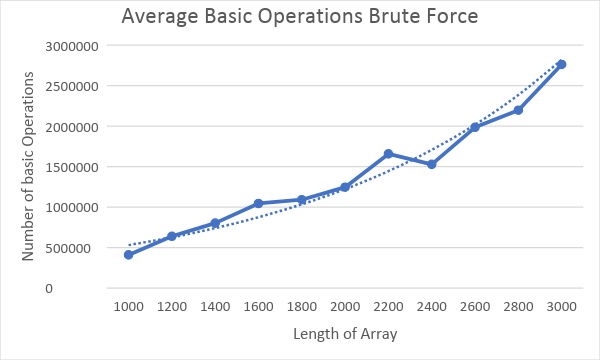
|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Length of array | 1,000 | 1,200 | 1,400 | 1,600 | 1,800 | 2,000 | 2,200 | 2,400 | 2,600 | 2,800 | 3,000 |
| No. of Basic Operations | 3363.91 | 4059.9 | 4863.92 | 5419.03 | 5950.26 | 6449.15 | 7467.27 | 8232.2 | 8557.97 | 9197.82 | 10053.77 |

Figure 11: The graph and average results for the basic operations test using the Median algorithm. It shows a clear relationship between length of array and average number of basic operations. That is as the length of array increases, so too does the number of basic operations.



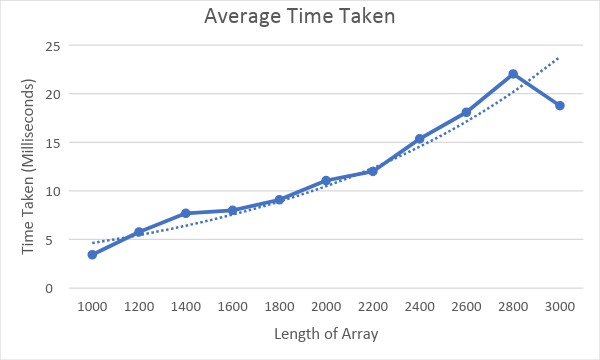
|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Length of Array | 1,000 | 1,200 | 1,400 | 1,600 | 1,800 | 2,000 | 2,200 | 2,400 | 2,600 | 2,800 | 3000 |
| Time Taken: | 0.035937 | 0.037851 | 0.044528 | 0.053707 | 0.060139 | 0.063429 | 0.067485 | 0.074923 | 0.085636 | 0.103299 | 0.092353 |

Figure 12: This graph represents the average time taken for the Median algorithm to find the median with arrays of lengths 1,000 to 3000. It shows a clear trend whereas the length of array increases so too does the average time taken to find the median.



|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Length of array | 1,000 | 1,200 | 1,400 | 1,600 | 1,800 | 2,000 | 2,200 | 2,400 | 2,600 | 2,800 | 3,000 |
| No of Basic Operations | | 408920.7 | 638935.1 | 801249.3 | 1044067 | 1090775 | 1245677 | 1656595 | 1527659 | 1985012 | 2196263 | 2761299 |

Figure 13: The graph and results for average number of basic operations used to find the median with the Brute Force algorithm. The results displayed were gathered with identical arrays as to those used in basic operation tests on the median algorithm. It shows an exponential increase in basic operations as the length of array increases.



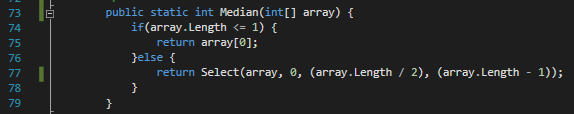
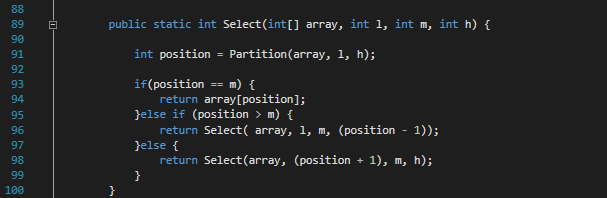
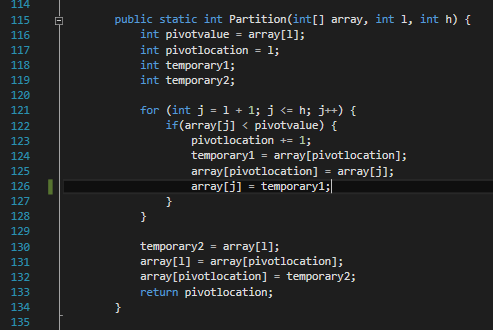
|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Length of Array | 1,000 | 1,200 | 1,400 | 1,600 | 1,800 | 2,000 | 2,200 | 2,400 | 2,600 | 2,800 | 3,000 |
| Time Taken(Milliseconds) | 3.423007 | 5.770494 | 7.690927 | 7.997106 | 9.084135 | 11.06472 | 11.9998 | 15.35067 | 18.09263 | 22.03309 | 18.76544 |

Figure 14: The graph and results for the average time taken for the Brute Force algorithm to find the median. The results displayed used identical arrays for each test as those used in the time tests for median algorithm. The results display a trend with respect to time and array length. As the array length increases so too does the time taken to find the median.

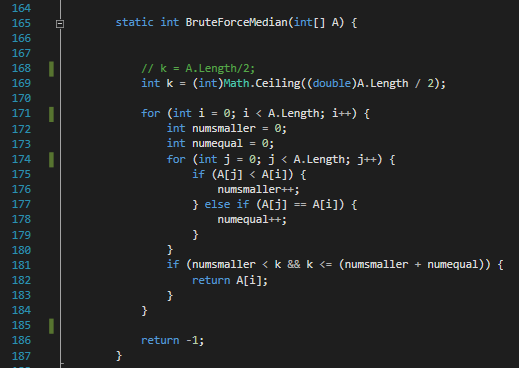
6 Appendices

Appendix A     Algorithm Code for Median Implementation

This appendix shows the code written for the algorithm from Figure 1, 2 and 3, it displays all three methods used in the process. Firstly, the Median method is called to begin the algorithm. It checks the length of the array. If the length is not larger than 1, the array returns its only element. If the array length is greater than 1 then it calls the select method. When it calls select it uses zero for the integer l, n/2 for the integer m and n-1 for the integer h. These are the positions of the first number in the array, the middle most number in the array and the last number in the array before it has begun being sorted. The Select method then creates integer position and calls the method Partition. Partition uses integers l and h for the first run, i.e. it sorts the array from position 0 to the last position. Partition sorts the method and then returns the pivot location. This value is then checked to see if it is equal to the middle number since the array has been sorted. If they are the same that number is returned, if it is not then the select method calls itself appropriately depending on whether the value of position is greater than m or less than m so that it can find the median.

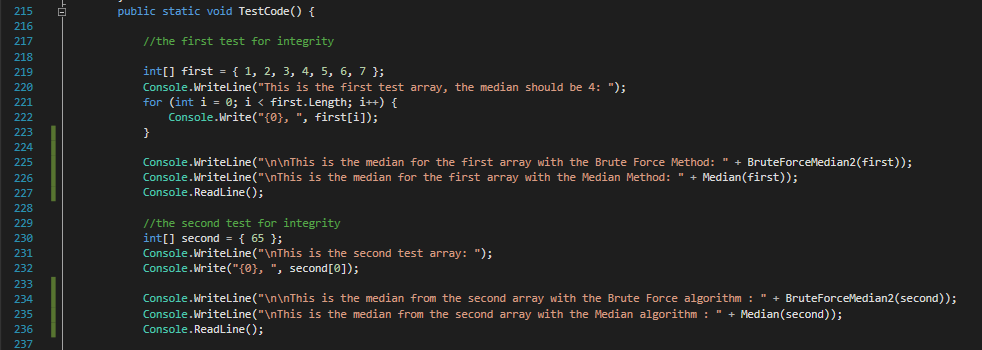
  
  
  
  
  
  
  
  


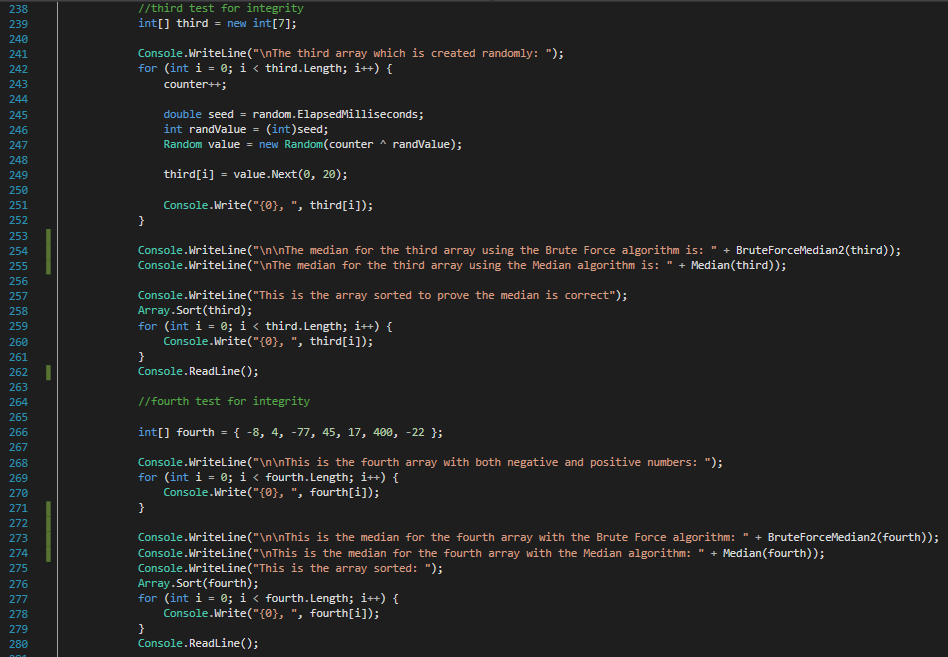
Appendix B Algorithm Code for Brute Force Median Implementation

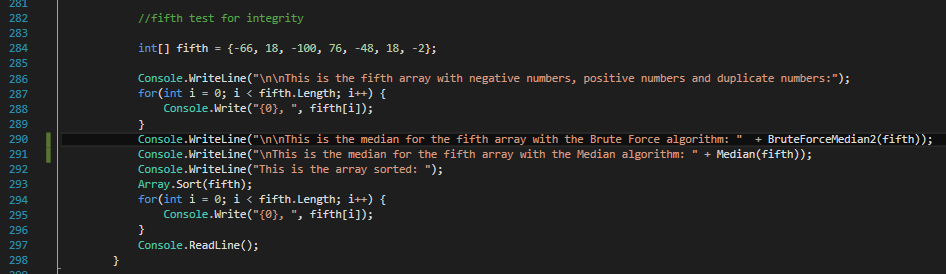
The appendix displays the implementation of the pseudocode for the Brute Force algorithm shown in figure 4. When called the algorithm takes an array of numbers. After being called the algorithm creates an int k. This is equal to half the total numbers in the array, this is important because this will be what determines which number is the median. It then takes the first number in the array, *A*[*i*] and compares it to every other number in the array A[j – n-1]. If A[j] is greater than *A*[*i*] the algorithm moves to the next number. If the value is equal to *A*[*i*], the counter numequal is increased by one, if the number is smaller than *A*[*i*], the counter numsmaller is appended by one. The algorithm compares every value in the array to *A*[*i*] and reacts with the counters accordingly. Once every number has been compared, it checks to see if the value of numsmaller is less than k. If it is not, it means *A*[*i*] is not the median value. If it does evaluate to true it, then also checks to see if the value of numsmaller + numequal to is greater than or equal to k. If this value is equal to or greater than k, the value of *A*[*i*] is returned as it is the median of the arrays. If numsmaller + numequal is less than k, the algorithm moves onto the next value in the array *A*[*i + 1*] and begins the process again. This is repeated until the median is found. 

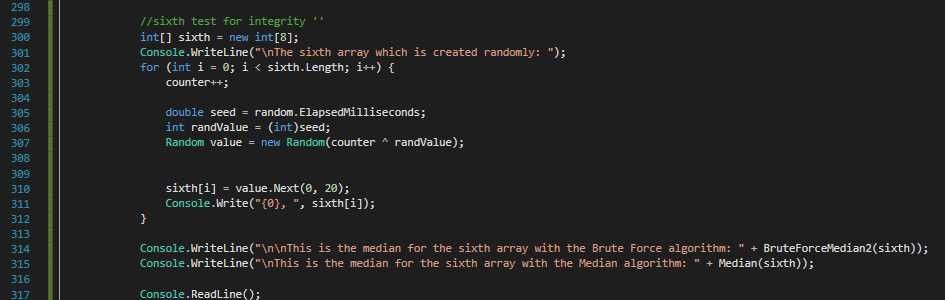
Appendix C    Code to Validate Implementation

The code displayed below is used to test fidelity and justify the implementation of the algorithm into the chosen language of C#. The tests will prove that the code correctly chooses the median number from an array that both has and has not been sorted. The first section tests that the algorithm correctly chooses the median in a sorted array of length 7. The second test finds how the algorithm performs with an array of length one. The third test is an unsorted randomly generated array of length 7, it shows the array before the algorithm has run and then the median it has chosen followed by the array after being sorted to prove that the chosen number is correct. The fourth test uses an array of length 7 with both positive and negative numbers that is unsorted. It shows the array before the algorithm is used, then the median chosen by the algorithm followed by the array after having been sorted. The final test runs the algorithm through an array with duplicate numbers. The test is given an array of length 7 containing numbers both positive and negative. The array is given a duplicate number of 18 and still finds the correct median. Finally the results from this code have been captured and as discussed the algorithm collects the correct median for each array which is proven after it has been sorted. The Final test code creates an even length array which forces the algorithms to choose different median values. The array is randomly generated and the results are printed at the end to prove that they choose separate values.





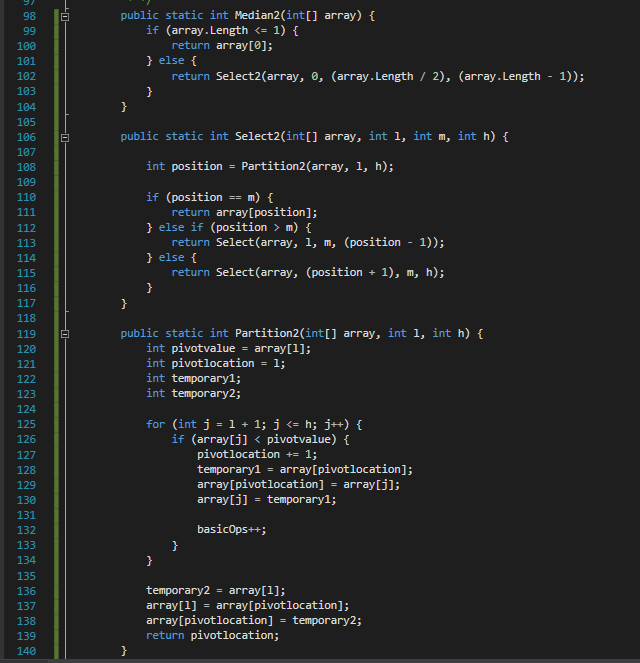


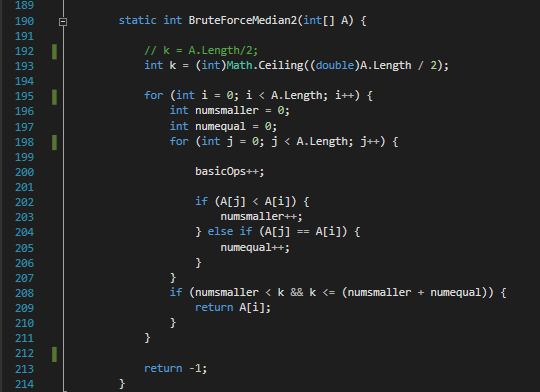


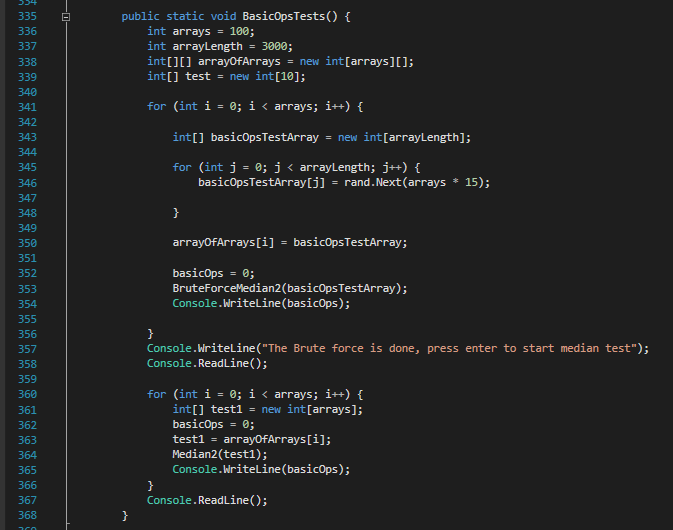
Appendix D    Code to Test Basic Operations

The code written to test for the number of basic operations used by the algorithm for arrays of varying lengths. The code creates 100 arrays and randomly generates every number in them for the length given by the variable lengthOfArray. The tests ranged from arrays with 1,000 values all the way to 3,000 values, incrementing by 200 numbers each test. To count the basic operations a global int variable was created called basicOps and was given a value of 0. To implement the counter, a copy of the pseudocode was made to prevent compromising the time tests for both algorithms. For the Brute Force algorithm, the basicOps counter increments for every comparison that is made by the algorithm. In the case of Median, the basicOps counter increments when the if statement evaluates to true inside the Partition Method.

Within the tester code for the basic operations after array *i* has been created it is filled with an amount of numbers and then the array is appended to the array variable arrayOfArrays to be used by the second algorithms test. After copying the array to be tested, the counter is reset. The method BruteForceMedian2 is then called and with it the counter increments. Once the algorithm is complete the basicOps variable is printed. This process is repeated for every array. Once the Brute Force algorithm test is complete, the method waits for input from the enter key before then running a loop through the arrayOfArrays and testing each array in the list with the median algorithm. This method of testing successfully tests both algorithm under exactly the same circumstances with exactly the same arrays.







Appendix E Code to Test for Time Efficiency

To test for time efficiency a new method was created. Three variables were created, the first called arrays, monitored the number of arrays being created. This was constant and set to 100 arrays. The second variable called arrayLength controlled the length of each array. This array changed for each run of the test. The first test began with 1,000 values, the second had 1,200 and this continued until the final test had 3,000 values. The final variable was a stopwatch object which was used to track the time taken for the algorithms to find the median in each array. Each array had every number randomly generated and once that was complete the timer was reset to make sure the time was 0 before being used. The timer was then started, and the algorithms were tested. After the algorithms had completed it task the timer was immediately stopped. The time was then printed, and the process restarted until 100 arrays were tested.

