CAB301 Assignment 2

Empirical Comparison of Two Algorithms

Student name: John Santias, Shivam Sachdeva

Student no. n9983244, n9642587

Due Date: 21st May 2018

**Summary**

…

**1 Description of the Algorithm**

**1.1 Brute Force Median Algorithm**

The brute force median algorithm returns the median value in a given array. Typically, in a sorted list the median would always be in the middle, but by having an unordered list it can be harder to find the value. “Brute force” means the algorithm will try to find many possible ways to find the median value of a given array. This algorithm’s pseudocode can be found in **Appendix A**. Referring to this pseudocode, firstly, the algorithm assigns the value in the middle of the list to a variable named ‘*k*’. This is assuming that we do have the correct median value seeing as the list is ordered or not. When selecting a median inside an uneven array size (decimal rounded up), the program will choose the value in the middle, but with an uneven sized array, it will choose the left of the two middle elements (). No matter the selected median is correct or not, the following for-loop runs to double check it. This for-loop works its way through the array from left to right. Each array value is used as a pivot to be compared with the other values. This loop starts off with assigning the two variables, ‘numsmaller’ and ‘numequal’, with a value of zero. Afterwards, another for-loop is executed to use each array value for comparison with the pivot. Inside this inner loop, there are two comparisons. The first comparison determines whether there is a value in the array less than the current pivot. If it is, then the variable ‘numsmaller’ increments by one. Otherwise, the second comparison determines if the values are equal then it increments the variable ‘numequal’ by one. Breaking back out to the original for-loop, a final comparison determines if it has chosen the correct median value by seeing that the value of ‘k’ is greater than the value of ’numsmaller’ and it is less than and equal to the total value of ‘numsmaller’ and ‘numequal’. In other words, it determines if the value of ‘k’ is the median value of all values in the array, seeing that there are even numbers on the left and the right side of the chosen median or there is one more value on the right compared to the left side of the chosen median. If this comparison is not met, the for-loop increases by one and executes another iteration of the whole process.

**1.2 Median Algorithm**

The Median algorithm is used to find the median of an array of elements. There are three separate procedures in this algorithm. This algorithm is special case of Johnsonbaugh and Schaefer’s version of the selection problem algorithm. The pseudocode of the algorithm is shown in **Appendix B**. The main procedure called median takes the array as a parameter. If the array contains only one element, the algorithm returns that element as the median whereas if the array contains more than one element, the procedure calls the second procedure called select. The select procedure takes four parameters: array, first index value of array, index of median element and the index of last element of array. This procedure calls the partition procedure which takes the first element of array as a pivot and compare it with all the other values of array. Every time the pivot value is greater than the other element, the pivotloc variable increases by one and swaps the element around the pivot. After the for-loop is executed, the pivot is placed on position where it should be if the array was sorted. This procedure returns the pivotloc variable. The value returned from partition procedure is stored in pos variable of Select procedure and compared with the index of median element which is equal to n/2 where n is number of elements in array. If the value matches, the select procedure returns the median of array and in case value does not matches, the select procedure is called again but not with the subarray. Now again partition procedure is called to do partitioning on subarray. The returned value of partition procedure is compared again and this process goes on until the value matches with the index of median element. Basically, Select algorithm is a recursive method which is called many times until the median is found. Every recursive call involves an array slice whose size is reduced every time.

**2 Theoretical Analysis of the Algorithm**

**2.1 Choice of Basic Operation**

The basic operation of Brute force median algorithm is the if and else-if conditions within nested for loops. Every array element is compared to itself and to the other elements. The block of if and else if condition checks whether the element is greater than or equal to other array elements. If the array element is greater than the other element, the ‘numsmaller’ variable increments where if two elements are equal, the ‘numequal’ variable increments. This block of if and else if condition is considered as the one basic operation which determines the efficiency of brute force median algorithm. The basic operation of the Median algorithm is the if condition within the for-loop of the partition procedure which checks whether the pivot is greater than the other elements. Basically, how many times pivot is checked against other elements or in other words how many times for-loop executes for an array determines the efficiency of the Median algorithm. The partition procedure can be called recursively many times until the median is found. The basic operation chosen for both the algorithms performs same function which is comparing element with the other array elements.

**2.2 Choice of Problem Size**

The problem size given to our algorithms ranges from one to a hundred. Incrementing the problem size by one. We implemented a for-loop re-iterating 100 times to generate arrays with sizes from one to a hundred, then passing the array to the two median algorithms to solve. Each element value in the Array has a random generated number between zero and ten thousand. This is because we are dealing with average-case efficiency of our algorithms, we don’t want our array elements to be in order or in descending order. Both are best and worst-case efficiencies. As we are comparing two algorithms, we will be giving the algorithms the same input sizes and same element values.

Both algorithms have different behaviors when finding the median of an even array size. The Brute Force algorithm returns the left of the two middle values while the Median algorithm returns the right value.

**2.3 Average-case efficiency**

Each algorithm has a different behavior to find the median of a given array. For each array that is passed on to the algorithms, we want to analyze which algorithm is more efficient. In this report we are only analyzing the average-case efficiency, not the best- and worst-case efficiency. If an array passed on to our algorithm has elements in ascending order, it will produce the best time complexity because there will be less iterations and the algorithms can almost immediately find the median. This is the best-case efficiency. The opposite of this is the worst-case efficiency, having an array in descending order making the algorithms produce the most iterations possible and longer time complexity. In analyzing the average-case, we will need to pass unsorted arrays to our algorithms. As explained in section 2.1, this will involve having a random number generator to append to our arrays.

For analyzing the Brute Force algorithm, the for-loop iterates from to and the inner for-loop iterates from to . Recalling from section 2.1, we chose the if and else comparisons as our basic operation. As these comparisons are executed inside a for-loop, we use the summation formulae described by Levitin **[levitin, page 62-63]** to compute the average case of the Brute Force Median's basic operation.

Mathematically,

Using this formula, where n = size of the array, we will find that by increasing the problem size the efficiency will increase quadratic as shown on the graph in **Appendix H.** We used an excel spreadsheet and the formula to calculate the number of comparisons for each problem size, we expect our tests results to match this graph [**Appendix H**].

**3 Methodology, Tools, and Techniques**

**3.1 Programming Environment**

1. We decided to implement the algorithms and perform the experiments in the C# programming language. As having used it before, we found it easy to run experiments and retrieve results.
2. The experiments were performed on a Windows 10 PC. It has an intel i7 Core processor running at 3.20GHz, 16GB of RAM and 64-bit operating system. We used the software called visual studio community to run our C# experiments and made sure other programs and windows were minimized to prevent interruptions. We used C#’s random number class to create random values inside our array and its system timer to record the program’s execution time.
3. A Microsoft Excel spreadsheet was used for recording our results and producing graphs. This software will help organise our experiment results in separate columns and rows. It will also help us calculate the average result of basic operations and time efficiency for each size input automatically. Using the results, we generated a graph using Excel’s graph function.

**3.2 Implementation of the Algorithms**

Algorithms were implemented in C# following the pseudocodes in **Appendix A and B**. The algorithms run on a single file to find the median value of the given arrays. As mentioned in section 2.2, we want to be using the same array for both algorithms in each test so that we can compare it correctly.

~~we want the algorithms to use the same array with the same elements so we don’t produce any errors in the results~~

Whilst implementing the Brute Force algorithm, we found that making the variable ‘k’ an integer would produce the wrong result. Reason for this is when the algorithm is given an uneven array, it returns the left of the median value. For example, an array {1, 2, 3, 4, 5} would assign the value ‘2’ to the variable ‘k’ because (or ) which is wrong. We expect the value to be ‘3’ in this case. As having found that using an integer would produce the wrong result, we defined ‘k’ as a double to use decimal numbers and the ‘*Math.Ceiling’* method to round up the value to 3.

The method ‘*swap’* [**see Figure 1 of Appendix E]** was implemented in our program to swap two elements in an array. This method is used in the Partition procedure of the Median algorithm to sort the elements. The algorithms implemented in C# can be seen in **Appendix C and D**.

**3.3 Generating Test Data and Running the Experiments**

To test the correctness of the implementation of our median algorithms, a function called ‘*GenerateRandomArray’* [**see Figure 2 Appendix E**] was included. It takes an array size as a parameter and generates an array of that size with random unique values ranging from zero to ten-thousand. The produced array is then used in the two algorithms to find the median.

**3.4 Functional Testing**

To test the correctness of the program, a test method, shown in **Appendix F,** was used. This test method runs three tests for each array size ranging from one to ten. Each test generates a new unique random array and for that array, the median is calculated using both algorithms. The results obtained from this test is shown in **Appendix G.**

**\*\*\*Maybe include console print in here just in case?**

**4 Experimental Results**

…

**4.1 Average-efficiency case comparison of both algorithm in terms of number of basic operation**

To calculate the basic operations, we inserted two global variable counters called ‘*counterForMedian’* and ‘*counterForBrute’* [**see Appendix I**]. This was to store the number of basic operations for each test in which we can then calculate the average out thirty tests in our ‘Main’ function **[See Appendix I].**

**4.1.1 Basic operation counter for Median Algorithm**

We inserted a variable called ‘*count’* in the Median algorithm **[See Appendix I**] to calculate the basic operation. Referring to section 2.1, we chose the if statement within the for-loop of the partition procedure as the basic operation. For each iteration of the for-loop we increment the counter by one. After the for-loop, we add the value of the ‘*count’* variable to the global variable ‘*counterForMedian’.*

**4.1.2 Basic operation counter for Brute Force Median Algorithm**

To calculate the basic operation of the Brute Force Median algorithm, we inserted a variable called ‘*counter’* [**See appendix I**]. Starting from zero, we continuously increment the variable by one in each iteration of the inner for-loop. The ‘*counter’* variable is then added to the global variable ‘*counterForBrute’* [**See appendix I]***.*

**4.1.3 Calculating the counter’s average**

For an array size we executed thirty tests, and in each test, we produced a random array of its size. For each test we calculated the number of basic operations performed by both algorithms which we stored in our global variables named ‘*counterForMedian’* and ‘*counterForBrute’* [**See appendix I**]*.* These values are further added to the variables *‘averageOneForMedian’* and *‘averageOneForBrute’* [**See appendix I**]*.* After executing all thirty tests, we calculated the average by dividing the *‘averageOneForMedian’* and *‘averageOneForBrute’* variables with the number of tests run which is thirty in this case. The final average basic operation is stored in the variables *‘averageTwoForMedian’* and *‘averageTwoForBrute’* [**See appendix I**]to be displayed on the console.

**4.2 Average-efficiency case comparison of both algorithm in terms of execution time**

To calculate the execution time, we inserted two global variables called *medianTimer* and *bruteTimer* to retrieve the execution time for both algorithms. The ‘sw’ [See appendix J] is a stopwatch of a global scope.

**4.2.1 Execution timer for Median Algorithm**

To calculate the execution time of the Median algorithm, we start the stopwatch in the partition procedure before the for-loop [See appendix J] and stops the timer in the Select procedure when the Median is found. The execution time is then stored in the global variable called ‘*medianTimer*’. Afterwards, the stopwatch is reset back to zero for another test.

**4.2.2 Execution timer for Brute Force Median Algorithm**

To calculate the execution time of the Brute Force Median algorithm, a variable named ‘*timer’* starts a new stopwatch before the for-loop. This timer stops when the median is found similarly to the Median algorithm. The execution time is then stored in the global variable called ‘*bruteTimer’*.

**4.2.3 Calculating the average execution time**

Similarly, on section 4.1.3, for an array size we executed thirty tests and with each test we produced a random array for its size. In each test we calculated the execution time for both algorithms which are then stored in our global variables, ‘*medianTimer’* and ‘*bruteTimer’.* The value of the ‘*medianTimer’* is added to the ‘*averageMedianTimer’* and the ‘*bruteTimer’* is added to the ‘*averageBruteTimer’*. Afterwards, when all the tests are executed, the ‘*averageMedianTimer’* and the ‘*averageBruteTimer’* is divided by the number of tests run (thirty in this case) [see Appendix J] which gives the average execution time for both algorithms.

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

…

**4.4 Average-Case Execution Time**

…

**References**

[1] Levitin, A., 2011. *Introduction to the Design and Analysis of Algorithms*. Addison-Wesley.

**Appendix A – The Brute Force Algorithm**



**Appendix B – The Median Algorithm**



**Appendix C – Implementing the Brute Force Median algorithm in C# programming language**



**Appendix D – Implementing the Median algorithm in C# programming algorithm.**



**Appendix E – Functions to swap values inside an array and generating arrays with random values**

Figure 1:



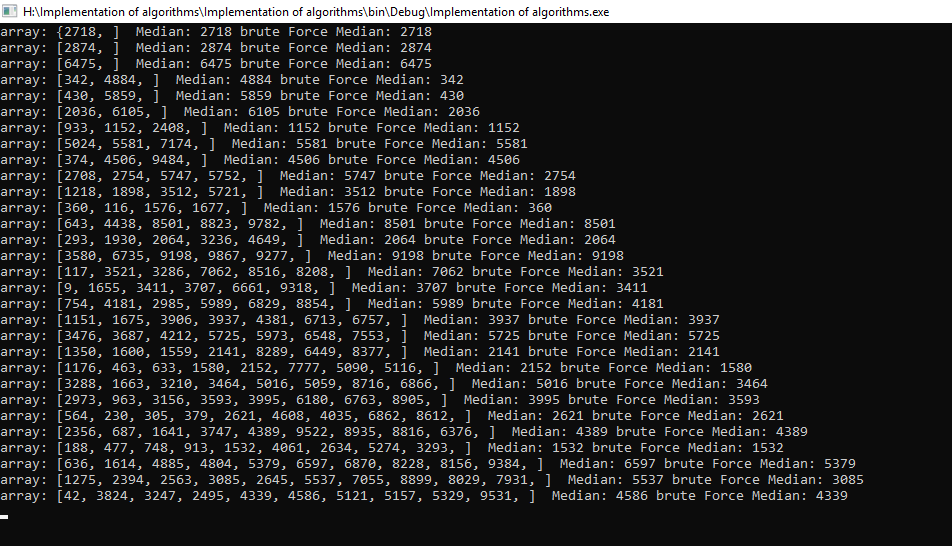
Figure 2:



**Appendix F**



**Appendix G – Testing functionality results**



**Appendix H – Brute Force average-case prediction**











