CAB301 Assignment 2

Empirical Comparison of Two Algorithms

Student name: John Santias, Shivam Sachdeva

Student no. n9983244, n9642587

Due Date: 21st May 2018

**Summary**

This report compares two different algorithms that both have the same function. The Median and the Brute Force Median algorithms both work to find the Median value inside a given array, however, both may have different efficiencies during the process. In this report, we analyze each algorithm by measuring the basic operations and execution time then compare the results to determine which algorithm is more efficient. Both programs are analyzed exactly in the same conditions to ensure that the results are comparable.

**1 Description of the Algorithm**

**1.1 Brute Force Median Algorithm**

The brute force median algorithm returns the median value in a given array. 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. It is also known as exhaustive search [2, Section 2]. This algorithm’s pseudocode can be found in **Appendix A**. It starts off by getting the size of the array and divided it by two, this value is assigned to a variable named ‘*k*’. This is indicating the middle index of the array where the median will be. When finding the median inside an uneven array size the program will choose the value right in the middle. With an uneven sized array, it will choose the left of the two middle elements (). A for-loop is then executed to work its way through the array from left to right. Each array value is used as a pivot. This loop firstly assigns the two variables ‘*numsmaller’* and ‘*numequal’* with a value of zero. Afterwards, another for-loop is executed to use each element in the array for comparison with the pivot. Inside this inner loop, there are two comparisons. The first comparison determines if the current element value is less than the pivot’s value. If it is, then the variable ‘numsmaller’ increments by one. Otherwise, the second comparison determines if the values are equal. If they are, then the variable ‘numequal’ increments by one. Once all the elements are compared with the pivot, the algorithm breaks out of the for loop and runs a final comparison which determines if the chosen pivot is the correct Median. This is done by comparing the value of ‘*numsmaller*’ if it is less than the *k* and the value of ‘*numequal’* is greater than or equal to *k*. If this comparison is true, then the algorithm will return the pivot’s value as the median. Otherwise, the algorithm will repeat the whole process until the final comparison is correct.

**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.

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 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 as 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. Each element value in the Array is generated randomly between zero and ten thousand. We are generating random arrays to conduct the experiments for finding the average-case efficiency. As we are comparing two algorithms, we will be giving them the same array for each test.

**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. In analyzing the average-case efficiency, we will need to pass random generated arrays in random order to our algorithms.

**2.3.1 Average-case efficiency for the Brute Force Median algorithm**

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]** and **[5, Summation Formulas]** to compute the average case of the Brute Force Median's basic operation.

Mathematically,

The efficiency will have an trend because of the two nested for-loops. 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 to the graph in **Appendix H**.

**2.3.2 Average-case efficiency for the Median algorithm**

Recalling from section 2.1, we chose the same basic operation for both algorithms. For the median algorithm, we chose the if-condition within the for-loop of the partition procedure. It iterates from according to the pseudocode in **Appendix B**. When the partition procedure is called for the first time, the if-condition is checked times [1, page 161]. After executing n-1 times, it partitions the array in to sub arrays and it can be called recursively until the median is found. The worst-case can have iterations [1, page 161]:

If the partitions are unbalanced, then we will get the worst-case efficiency. The average-case efficiency of the Median is found to be linear [1, page 161]. It will be linear if good partitions occur, which will always depend on the given array. For example, if we provide the array [1, 2, 7, 4, 5, 3, 6] it will produce the worst-case efficiency because the number basic operations using the Median algorithm will be 21 matching the result using worst-case efficiency formula . The worst-case occurs when the median is right in the middle. Whereas if we provide the same array but in different order [4, 1, 2, 3, 5, 6, 7], this will produce the best-case efficiency which is equal to six. This is matching the best-case efficiency formula [1, page 161]:

The best-case occurs when the first element is the median. The average-case iterations can go from (n-1) + (n-3) or (n-1)+(n-2) all depending on the given array. The efficiency will come out to be where is constant. For example, if we have the array [2, 1, 4, 3] the efficiency comes out to be 4 in terms of basic operations because in this case the for-loop is executed (4-1)+(4-3) times equaling 4. If we have an array [1, 3, 2, 4] the efficiency is 5, because the for-loop will execute (4-1)+(4-2) times which equals to 5. Berman and Paul explains that the average efficiency of the Select procedure assumes its inputs are all permutations of l, …, n and each permutation are equally likely. The average-case efficiency for this procedure shows [3, page 249]:

Where A is average-case, n is the array size, and t is any integer where we can always assume it is zero [3, page 249]. Using this formula, we expect that for each array we give to the Median algorithm will produce an average-efficiency of less than or equal to 4n. Thus, producing a linear line as stated by Levitin [1, page 161], as do Johnsonbaugh and Schaefer [4, page 262]. We expect our results to be similar to the graph in **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 stopwatch class 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.

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 **Figure 1 of 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 **Figure 1 of Appendix G.**

As expected, we wanted each test to produce a new array with unique elements and have both algorithms obtaining the correct median:



Keeping in mind, from Section 1.1, an even array would have two median numbers, the Median algorithm would select the number on the right whereas the Brute Force median algorithm would select the left number.

We wanted to see if the algorithms can handle bigger sized array with unique values. This time we increased the size of the array by thirty-five until the size was bigger than one-thousand [**see Figure 2 of Appendix F]**. The results came out to be:

Array size of 36:  
**array**: [5459, 2003, 4531, 4810, 205, 665, 1756, 4603, 1779, 5325, 5113, 5047, 302, 2097, 5080, 2925, 550, 1109, 5628, 6957, 7329, 5902, 7104, 7229, 6720, 7273, 6033, 8240, 9731, 9009, 8260, 9623, 9129, 8931, 8297, 9861, ]

**Median**: 5628 **brute Force Median**: 5459

**Sorted**: [205, 302, 550, 665, 1109, 1756, 1779, 2003, 2097, 2925, 4531, 4603, 4810, 5047, 5080, 5113, 5325, 5459, 5628, 5902, 6033, 6720, 6957, 7104, 7229, 7273, 7329, 8240, 8260, 8297, 8931, 9009, 9129, 9623, 9731, 9861, ]

Array size of 71:  
**array**: [437, 1464, 294, 1048, 978, 636, 383, 1329, 161, 632, 1013, 1367, 1146, 1618, 2735, 1702, 2142, 3108, 2975, 3392, 1870, 2825, 2944, 3427, 3701, 4091, 4650, 3829, 4031, 3802, 4847, 3566, 4008, 3485, 4316, 4922, 5563, 5737, 6084, 5571, 5455, 6031, 6306, 5108, 6216, 6651, 5977, 6055, 6838, 7307, 6967, 7108, 7424, 9680, 8458, 9130, 9571, 8380, 7901, 8275, 8489, 9911, 9310, 9178, 8687, 8153, 8629, 7817, 9250, 9329, 8616, ]

**Median**: 4922 **brute Force Median:** 4922

**Sorted**: [161, 294, 383, 437, 632, 636, 978, 1013, 1048, 1146, 1329, 1367, 1464, 1618, 1702, 1870, 2142, 2735, 2825, 2944, 2975, 3108, 3392, 3427, 3485, 3566, 3701, 3802, 3829, 4008, 4031, 4091, 4316, 4650, 4847, 4922, 5108, 5455, 5563, 5571, 5737, 5977, 6031, 6055, 6084, 6216, 6306, 6651, 6838, 6967, 7108, 7307, 7424, 7817, 7901, 8153, 8275, 8380, 8458, 8489, 8616, 8629, 8687, 9130, 9178, 9250, 9310, 9329, 9571, 9680, 9911, ]

Array size of 106:  
**array**: [85, 107, 4289, 618, 1977, 1962, 4299, 502, 3193, 3584, 2202, 4309, 2143, 1418, 3707, 3348, 2144, 3459, 392, 1246, 2615, 3123, 2861, 3532, 2579, 2146, 516, 1578, 2074, 950, 3915, 2104, 469, 401, 3062, 3806, 2978, 3297, 2080, 2095, 4198, 4007, 523, 583, 4737, 2025, 4661, 4760, 1018, 240, 4781, 4957, 4853, 5130, 5563, 5594, 5611, 5677, 5765, 6233, 5616, 6177, 6110, 6316, 5798, 6318, 6844, 7268, 6625, 7071, 6586, 7004, 7844, 7999, 7821, 7374, 7389, 6348, 7824, 7660, 7572, 6421, 6858, 6526, 7468, 6861, 6938, 8133, 8611, 9493, 8816, 9718, 9869, 9150, 9899, 8269, 8886, 9691, 9172, 9521, 9895, 9097, 8676, 9287, 9337, 9418, ]

**Median**: 5130 **brute Force Median**: 4957

**Sorted**: [85, 107, 240, 392, 401, 469, 502, 516, 523, 583, 618, 950, 1018, 1246, 1418, 1578, 1962, 1977, 2025, 2074, 2080, 2095, 2104, 2143, 2144, 2146, 2202, 2579, 2615, 2861, 2978, 3062, 3123, 3193, 3297, 3348, 3459, 3532, 3584, 3707, 3806, 3915, 4007, 4198, 4289, 4299, 4309, 4661, 4737, 4760, 4781, 4853, 4957, 5130, 5563, 5594, 5611, 5616, 5677, 5765, 5798, 6110, 6177, 6233, 6316, 6318, 6348, 6421, 6526, 6586, 6625, 6844, 6858, 6861, 6938, 7004, 7071, 7268, 7374, 7389, 7468, 7572, 7660, 7821, 7824, 7844, 7999, 8133, 8269, 8611, 8676, 8816, 8886, 9097, 9150, 9172, 9287, 9337, 9418, 9493, 9521, 9691, 9718, 9869, 9895, 9899, ]

Here we sorted the arrays to make it easier for us to find the median. With these tests results, we confirmed that both algorithm did not produce any errors and are functionally correct. The results obtained from this test is shown in **Figure 2 of Appendix G**.

**4 Calculating the 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.

**5 Experimental Results**

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

**5.1.1 Average-case Basic Operations for the Median algorithm**

As mentioned in Section 2.3.2, we expected that the results will produce a linear line and the efficiency would be equal or less than 4n. The results did match our predictions in section 2.3.2, producing a linear line. The results in a graph can be seen in **Appendix K**. Here you can see that the predicted linear line in the graph of **Append H**, almost matches our results. The results did not exactly match the prediction as we are using random generated arrays in each test.

**5.1.2 Average-case Basic Operations for the Brute Force Median algorithm**

The experimental results did match our predictions in Section 2.3.1. As predicted, the Brute Force Median line in the graph of **Appendix K** shows a quadratic line increase as we increase the array size. This brute force line almost matches the graph in **Appendix H**. As the line quadratically increases we can see that the basic operations tend to drop and increase. This fluctuation is because we are giving a random array to with random values in random order in each test.

**5.2 Average-Case Execution Time**

**5.2.1 Average-case Execution Time for the Median algorithm**

The results did match the predictions stated in section 2.3.2. The median line in the graph of Appendix L does show a linear trend line. Almost exactly matching our prediction. However, there are fluctuations in the graph. The reasons for this can be the computer’s processes working in the background affecting the performance and using portions of the RAM and CPU. Even if we had all other programs minimized, there would still be internal processes running such as the clock, ethernet etc.

**5.2.2 Average-case Execution Time for the Brute Force Median algorithm**

The experimental results also matched our predictions in Section 2.3.1. The brute force median line in the graph of **Appendix L** shows a quadratic increase as we increase the array size. Similarly, in section 5.1.2, there is a fluctuation because we are generating random arrays with random values in random order for each test. Other reasons for this fluctuation can be the computer’s processes as stated in 5.2.1.

**6 Conclusion**

1In conclusion, the experimental results matches our predicted results. The graph in Appendix M shows predicted and experimental results of both algorithms in terms of basic operations. We found that the Median algorithm is more efficient than the Brute Force Median algorithm for finding the median.

**References**

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

[2] Daniel Yim. 2009. *Topics on analysis of algorithms: brute force.* [ONLINE] Available at: ftp://cosm.sfasu.edu/cs/rball/public\_html/342/Daniel\_WebProject/index.html. [Accessed 2 May 2018].

[3] A., K., 2004. *Algorithms: Sequential, Parallel, and Distributed.* Cengage Learning.

[4] Johnsonbaugh, R., 2003. Algorithms. TBS.

[5] Rodney Anderson. 2018. *CS Summations*. [ONLINE] Available at: http://everythingcomputerscience.com/discrete\_mathematics/Summations.html. [Accessed 10 May 2018].





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

1. /// <summary>
2. /// Finds the Median value of a given array
3. /// </summary>
4. /// <param name=”A”>Gets the array</param>
5. /// <returns>Median value</returns>
6. static int BruteForceMedian(int[] A)
7. {
8. double k = Math.Ceiling(A.Length / 2.0); //Gets the position of the Median
9. for (int i = 0; i < A.Length; i++) //Uses each element of the array as pivot
10. {
11. int numsmaller = 0; //Sets variables to zero
12. int numequal = 0; //Sets variables to zero
13. for (int j = 0; j < A.Length; j++) //Uses each element of array to compare with the pivot
14. {
15. if (A[j] < A[i]) //if the pivot is greater than current element
16. {
17. numsmaller = numsmaller + 1; //increments by one
18. } else {
19. if (A[j] == A[i]) //if pivot is equal to current element
20. { //incremenet variable by one
21. numequal = numequal + 1;
22. }
23. }
24. }
25. if (numsmaller < k && k <= (numsmaller + numequal)) //Determines if the value of k is greater than the value of numsmaller and is less than or equal to the numsmaller and numequal combineds
26. {
27. return A[i]; //returns pivot value as the median
28. }
29. }
30. return 0;
31. }

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

1. /// <summary>
2. /// Finds the median of a given array
3. /// </summary>
4. /// <param name=”A”>Takes in a given array to solve</param>
5. /// <returns>The median value</returns>
6. static int Median(int[] A) //Takes in array to find the Median
7. {
8. if (A.Length == 1) //Checks if array has one value
9. {
10. return A[0]; //Returns the only value in the array as median
11. } else {
12. return Select(A, 0, A.Length / 2, A.Length – 1); //Runs select procedure passing the array, number zero as a pivot, median index, and length of array
13. }
14. }
15. static int Select(int[] A, int l, int m, int h)
16. {
17. int pos = Partition(A, l, h); //calls partition procedure and passing array, pivot, median index, array size. Obtains the position of the pivot
18. if (pos == m) //Checks if position is equal to the median index
19. {
20. return A[pos]; //Returns array position as final median
21. }
22. if (pos > m) //Checks if position is greater than the median index
23. {
24. return Select(A, l, m, pos – 1); //Re-runs this select procedure
25. }
26. if (pos < m) //Checks if the position is less than the median index
27. {
28. return Select(A, pos + 1, m, h); //Re-runs this select procedure
29. }
30. return 0;
31. }
32. static int Partition(int[] A, int l, int h)
33. {
34. int pivotval = A[l]; //uses the pivot’s value
35. int pivotloc = l; //uses the pivot index
36. for (int j = l + 1; j <= h; j++) //Uses each element in array to compare with the pivot
37. {
38. if (A[j] < pivotval) //if the current element is less than the pivot
39. {
40. pivotloc = pivotloc + 1; //increment the pivot by one
41. swap(A, pivotloc, j); //swap pivot index with the current element
42. }
43. }
44. swap(A, l, pivotloc); //swap pivot index with the pivotloc value
45. return pivotloc; //return pivot index
46. }

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

Figure 1:

1. /// <summary>
2. /// takes array, and the two indexes to swap
3. /// </summary>
4. /// <param name=”A”>Array</param>
5. /// <param name=”first”>index one</param>
6. /// <param name=”second”>index two</param>
7. static void swap(int[] A, int first, int second)
8. {
9. int s = A[second]; //assigns value of index two to variable s
10. A[second] = A[first]; //assigns value of index two with the value of index one
11. A[first] = s; //assigns value of index one with value of variable s
12. }

Figure 2:

1. /// <summary>
2. /// Creates a new array
3. /// </summary>
4. /// <param name=”size”>the size of the array to generate</param>
5. /// <returns>the generated array</returns>
6. static int[] GenerateRandomArray(int size)
7. {
8. int[] A = new int[size]; //creates new array with given size
9. for (int i = 0; i < A.Length; i++) { //iterates the following from zero to one
10. int n; //creates new variable without assigning any value
11. do {
12. n = rand.Next(0, 10000); //generates number between 0 and 10000 //checks if the number is already in the array, if true then repeat the do-loop to generate a new number
13. } while (A.Contains(n));
14. A[i] = n; //adds generated number to the array
15. }
16. return A; //return array
17. }

**Appendix F – The Main functions to test functionality**

Figure 1:

1. /// <summary>
2. /// Main controller of the program. Controls which algorithms, procedures and functions to run/// </summary>
3. /// <param name=”args”></param>
4. static void Main(string[] args) {
5. int numberOfTimes = 3; //number of tests for each array size
6. for (int size = 1; size <= 10; size += 1) {
7. for (int i = 0; i < numberOfTimes; i++) {
8. int[] test = GenerateRandomArray(size); //calls procedure to generate a new array
9. int medianResult = Median(test); //calls median algorithm to find median value
10. int bruteResult = BruteForceMedian(test); //calls brute algorithm to find median value
11. Console.Write(“array: [“);
12. foreach(int x in test) {
13. Console.Write(“{0}, “, x); //prints generated array
14. }
15. Console.WriteLine(“]  Median: {0} brute Force Median: {1} “, medianResult, bruteResult); //prints median values for both algorithms
16. }
17. }
18. Console.ReadKey(); //waits for user input to end program
19. }

Figure 2:

1. /// <summary>
2. /// Main controller of the program. Controls which algorithms, procedures and functions to run/// </summary>
3. /// <param name=”args”></param>
4. static void Main(string[] args) {
5. for (int size = 1; size <= 1000; size += 35) //re-iterates increasing the size of the array by 35
6. {
7. int[] test = GenerateRandomArray(size); //calls procedure to generate a new array
8. int medianResult = Median(test); //calls median algorithm to find median value
9. int bruteResult = BruteForceMedian(test); //calls brute algorithm to find median value
10. Console.Write(“array: [“);
11. foreach(int x in test) {
12. Console.Write(“{0}, “, x); //prints generated array
13. }
14. Console.WriteLine(“]”);
15. Console.WriteLine(“Median: {0} brute Force Median: {1} “, medianResult, bruteResult); //prints median values for both algorithms
16. Console.Write(“Sorted: [“);
17. Array.Sort(test); //sorts array in ascending order
18. foreach(int x in test) {
19. Console.Write(“{0}, “, x); //prints sorted array
20. }
21. Console.WriteLine(“]”);
22. Console.WriteLine();
23. }
24. Console.ReadKey(); //waits for user input to end program
25. }

**Appendix G – Testing functionality results**

Figure 1:

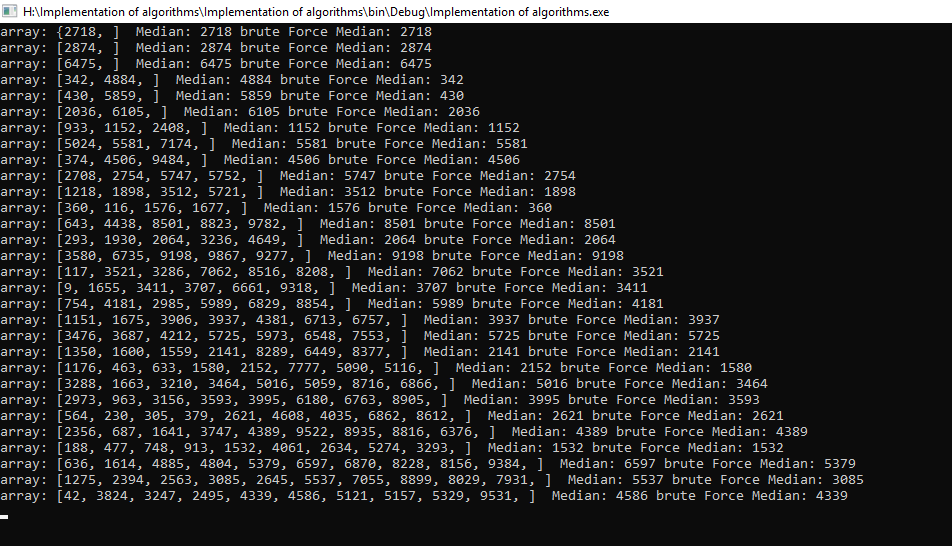
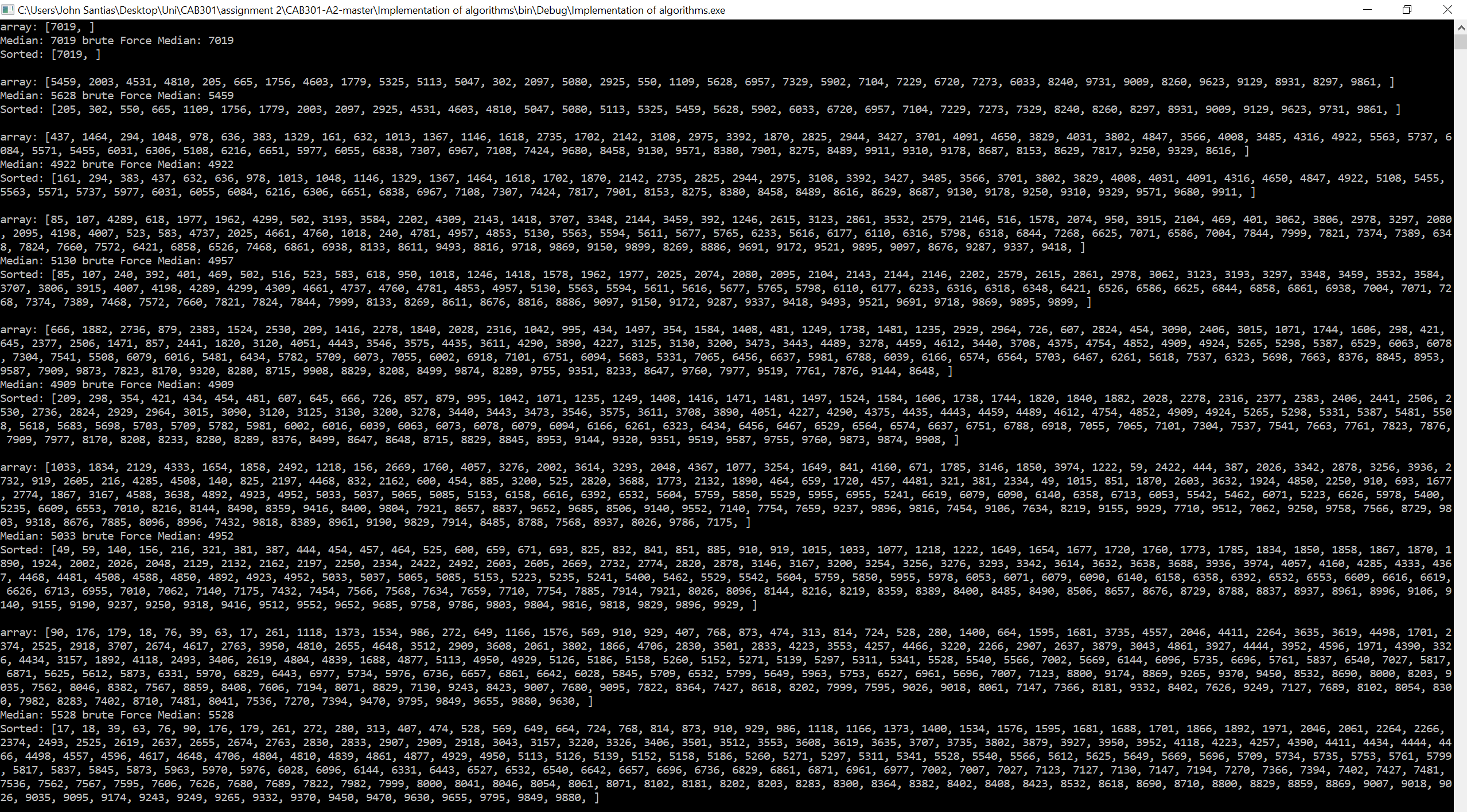


Figure 2

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

**Appendix I – Counting the basic operations**

1. private static int counterForMedian; //global variable to store counter value of median
2. private static int counterForBrute; //global variable to store counter value of Brute
3. private static Random rand = new Random();
4. static int Median(int[] A)
5. {
6. if (A.Length == 1) {
7. return A[0];
8. } else {
9. return Select(A, 0, A.Length / 2, A.Length – 1);
10. }
11. }
12. static int Select(int[] A, int l, int m, int h)
13. {
14. int pos = Partition(A, l, h);
15. if (pos == m) {
16. return A[pos];
17. }
18. if (pos > m) {
19. return Select(A, l, m, pos – 1);
20. }
21. if (pos < m) {
22. return Select(A, pos + 1, m, h);
23. }
24. return 0;
25. }
26. static int Partition(int[] A, int l, int h)
27. {
28. int pivotval = A[l];
29. int pivotloc = l;
30. int count = 0; //new variable count starting from zero
31. for (int j = l + 1; j <= h; j++) {
32. count++; //increment counter by one
33. if (A[j] < pivotval) {
34. pivotloc = pivotloc + 1;
35. swap(A, pivotloc, j);
36. }
37. }
38. swap(A, l, pivotloc);
39. counterForMedian += count; //add value of counter to global variable
40. return pivotloc;
41. }
42. static void swap(int[] A, int first, int second)
43. {
44. int s = A[second];
45. A[second] = A[first];
46. A[first] = s;
47. }
48. static int[] GenerateRandomArray(int size)
49. {
50. int[] A = new int[size];
51. for (int i = 0; i < A.Length; i++) {
52. int n;
53. do {
54. n = rand.Next(0, 10000);
55. } while (A.Contains(n));
56. A[i] = n;
57. }
58. return A;
59. }
60. static int BruteForceMedian(int[] A)
61. {
62. double k = Math.Ceiling(A.Length / 2.0);
63. int counter = 0; //new counter variable starting from zero
64. for (int i = 0; i < A.Length; i++) {
65. int numsmaller = 0;
66. int numequal = 0;
67. for (int j = 0; j < A.Length; j++) {
68. counter++; //increment counter by one
69. if (A[j] < A[i]) {
70. numsmaller = numsmaller + 1;
71. } else {
72. if (A[j] == A[i]) {
73. numequal = numequal + 1;
74. }
75. }
76. }
77. if (numsmaller < k && k <= (numsmaller + numequal)) {
78. counterForBrute += counter; //Add counter value to global variable
79. return A[i];
80. }
81. }
82. return 0;
83. static void Main(string[] args)
84. {
85. int numberOfTimes = 30;
86. for (int size = 1; size <= 100; size += 1) //number of tests for each array size
87. {
88. int averageOneForMedian = 0;  //new variables starting from zero
89. int averageTwoForMedian = 0;
90. int averageOneForBrute = 0;
91. int averagetwoForBrute = 0;
92. for (int i = 0; i < numberOfTimes; i++) {
93. counterForMedian = 0;  //resets global variable to zero
94. counterForBrute = 0;  //resets global variable to zero
95. int[] test = GenerateRandomArray(size);
96. Median(test);
97. BruteForceMedian(test);
98. averageOneForMedian += counterForMedian;  //add global variable
99. averageOneForBrute += counterForBrute;  //add global variable
100. }
101. averageTwoForMedian = averageOneForMedian / numberOfTimes;  //divide averageOneForMedian by number of tests to get average
102. averagetwoForBrute = averageOneForBrute / numberOfTimes;  //divide averageOneForBrute by number of tests to get average
103. Console.WriteLine(“size: {0}  median average: {1} brute average: {2} “, size, averageTwoForMedian, averagetwoForBrute);
104. }
105. Console.ReadKey();
106. }

**Appendix J – Execution Time**

1. private static double medianTimer; //Global variable
2. private static double bruteTimer; //global variable
3. private static Stopwatch sw; //stopwatch
4. private static Random rand = new Random();
5. static int Median(int[] A)
6. {
7. if (A.Length == 1) {
8. return A[0];
9. } else {
10. return Select(A, 0, A.Length / 2, A.Length – 1);
11. }
12. }
13. static int Select(int[] A, int l, int m, int h)
14. {
15. int pos = Partition(A, l, h);
16. if (pos == m) {
17. sw.Stop(); //stops stopwatch
18. medianTimer = sw.Elapsed.TotalMilliseconds; //converts stopwatch into milliseconds and assigns it to the global variable medianTimer
19. sw.Reset(); //Resets stopwatch back to zero
20. return A[pos];
21. }
22. if (pos > m) {
23. return Select(A, l, m, pos – 1);
24. }
25. if (pos < m) {
26. return Select(A, pos + 1, m, h);
27. }
28. return 0;
29. }
30. static int Partition(int[] A, int l, int h)
31. {
32. int pivotval = A[l];
33. int pivotloc = l;
34. sw.Start(); //starts stopwatch
35. for (int j = l + 1; j <= h; j++) {
36. if (A[j] < pivotval) {
37. pivotloc = pivotloc + 1;
38. swap(A, pivotloc, j);
39. }
40. }
41. swap(A, l, pivotloc);
42. return pivotloc;
43. }
44. static void swap(int[] A, int first, int second)
45. {
46. int s = A[second];
47. A[second] = A[first];
48. A[first] = s;
49. }
50. static int[] GenerateRandomArray(int size)
51. {
52. int[] A = new int[size];
53. for (int i = 0; i < A.Length; i++) {
54. int n;
55. do {
56. n = rand.Next(0, 10000);
57. } while (A.Contains(n));
58. A[i] = n;
59. }
60. return A;
61. }
62. static int BruteForceMedian(int[] A)
63. {
64. double k = Math.Ceiling(A.Length / 2.0);
65. var timer = System.Diagnostics.Stopwatch.StartNew(); //starts stopwatch
66. for (int i = 0; i < A.Length; i++) {
67. int numsmaller = 0;
68. int numequal = 0;
69. for (int j = 0; j < A.Length; j++) {
70. if (A[j] < A[i]) {
71. numsmaller = numsmaller + 1;
72. } else {
73. if (A[j] == A[i]) {
74. numequal = numequal + 1;
75. }
76. }
77. }
78. if (numsmaller < k && k <= (numsmaller + numequal)) {
79. timer.Stop(); //stops stopwatch
80. bruteTimer = timer.Elapsed.TotalMilliseconds; //assigns the timer to the global variable
81. return A[i];
82. }
83. }
84. return 0;
85. }
86. static void Main(string[] args)
87. {
88. sw = new Stopwatch(); //Creates new stopwatch
89. int numberOfTimes = 30;
90. for (int size = 1; size <= 100; size += 1) {
91. double averageMedianTimer = 0; //new variable with value of zero
92. double averageBruteForceTimer = 0; //new variable with value of zero
93. for (int i = 0; i < numberOfTimes; i++) {
94. int[] test = GenerateRandomArray(size);
95. Median(test);
96. BruteForceMedian(test);
97. averageMedianTimer += medianTimer; //adds global variable
98. averageBruteForceTimer += bruteTimer; //adds global variable
99. }
100. averageMedianTimer = averageMedianTimer / numberOfTimes; //divides the value of the averageMedianTimer by the number of tests run to get the average
101. averageBruteForceTimer = averageBruteForceTimer / numberOfTimes; //divides the value of the averageBruteForceTimer by the number of tests run to get the average
102. Console.WriteLine(“For size {0}, execution time of Median: {1}, execution time of BruteForceMedian: {2}”, size, Math.Round(averageMedianTimer, 5), Math.Round(averageBruteForceTimer, 5)); //prints results in 5 decimal places
103. }
104. Console.ReadKey();
105. }

**Appendix K – Experimental results for the Basic Operation**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Input size | Median Algorithm | Brute Force Algorithm | Input size | Median Algorithm | Brute Force Algorithm | Input size | Median Algorithm | Brute Force Algorithm |
| 1 | 0 | 1 | **44** | 132 | 913 | **87** | 251 | 3828 |
| 2 | 1 | 2 | **45** | 137 | 1035 | **88** | 271 | 3781 |
| 3 | 2 | 6 | **46** | 132 | 1028 | **89** | 238 | 4005 |
| 4 | 4 | 7 | **47** | 150 | 1128 | **90** | 291 | 3741 |
| 5 | 7 | 15 | **48** | 133 | 1036 | **91** | 267 | 4186 |
| 6 | 9 | 17 | **49** | 136 | 1225 | **92** | 282 | 4121 |
| 7 | 10 | 28 | **50** | 141 | 1213 | **93** | 284 | 4371 |
| 8 | 14 | 28 | **51** | 151 | 1326 | **94** | 293 | 4239 |
| 9 | 16 | 45 | **52** | 157 | 1294 | **95** | 281 | 4560 |
| 10 | 19 | 47 | **53** | 166 | 1431 | **96** | 306 | 4505 |
| 11 | 21 | 66 | **54** | 160 | 1404 | **97** | 311 | 4753 |
| 12 | 21 | 62 | **55** | 171 | 1540 | **98** | 299 | 4498 |
| 13 | 28 | 91 | **56** | 175 | 1554 | **99** | 324 | 4950 |
| 14 | 30 | 88 | **57** | 150 | 1653 | **100** | 315 | 4513 |
| kk15 | 34 | 120 | **58** | 166 | 1566 |  |  |  |
| 16 | 42 | 122 | **59** | 163 | 1770 |  |  |  |
| 17 | 38 | 153 | **60** | 174 | 1700 |  |  |  |
| 18 | 39 | 144 | **61** | 171 | 1891 |  |  |  |
| 19 | 44 | 190 | **62** | 199 | 1886 |  |  |  |
| 20 | 50 | 186 | **63** | 191 | 2016 |  |  |  |
| 21 | 50 | 231 | **64** | 189 | 2001 |  |  |  |
| 22 | 50 | 228 | **65** | 200 | 2145 |  |  |  |
| 23 | 59 | 276 | **66** | 205 | 2063 |  |  |  |
| 24 | 55 | 264 | **67** | 212 | 2278 |  |  |  |
| 25 | 61 | 325 | **68** | 221 | 2194 |  |  |  |
| 26 | 67 | 309 | **69** | 193 | 2415 |  |  |  |
| 27 | 63 | 378 | **70** | 197 | 2440 |  |  |  |
| 28 | 71 | 383 | **71** | 203 | 2556 |  |  |  |
| 29 | 78 | 435 | **72** | 229 | 2395 |  |  |  |
| 30 | 76 | 429 | **73** | 208 | 2701 |  |  |  |
| 31 | 84 | 496 | **74** | 229 | 2567 |  |  |  |
| 32 | 82 | 454 | **75** | 218 | 2850 |  |  |  |
| 33 | 81 | 561 | **76** | 245 | 2852 |  |  |  |
| 34 | 94 | 562 | **77** | 213 | 3003 |  |  |  |
| 35 | 97 | 630 | **78** | 217 | 2862 |  |  |  |
| 36 | 94 | 596 | **79** | 256 | 3160 |  |  |  |
| 37 | 96 | 703 | **80** | 233 | 3157 |  |  |  |
| 38 | 101 | 694 | **81** | 244 | 3321 |  |  |  |
| 39 | 115 | 780 | **82** | 264 | 3293 |  |  |  |
| 40 | 99 | 737 | **83** | 259 | 3486 |  |  |  |
| 41 | 122 | 861 | **84** | 252 | 3455 |  |  |  |
| 42 | 117 | 851 | **85** | 261 | 3655 |  |  |  |
| 43 | 129 | 946 | **86** | 278 | 3531 |  |  |  |

**Appendix L – Experimental results for the Execution Time**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Input size | Median Algorithm (ms) | Brute Force Algorithm (ms) | Input size | Median Algorithm (ms) | Brute Force Algorithm (ms) | Input size | Median Algorithm | Brute Force Algorithm (ms) |
| 1 | 0 | 0.00026 | **42** | 0.00184 | 0.00513 | **83** | 0.00406 | 0.01999 |
| 2 | 0.00398 | 0.00013 | **43** | 0.00224 | 0.00534 | **84** | 0.00386 | 0.01797 |
| 3 | 0.00055 | 0.00019 | **44** | 0.00199 | 0.00579 | **85** | 0.0044 | 0.02462 |
| 4 | 0.0003 | 0.00019 | **45** | 0.002 | 0.006 | **86** | 0.00386 | 0.01994 |
| 5 | 0.00034 | 0.00015 | **46** | 0.00206 | 0.00578 | **87** | 0.00354 | 0.01945 |
| 6 | 0.0004 | 0.00022 | **47** | 0.00296 | 0.00602 | **88** | 0.00413 | 0.01925 |
| 7 | 0.0005 | 0.00027 | **48** | 0.00252 | 0.00627 | **89** | 0.00363 | 0.02126 |
| 8 | 0.00045 | 0.00032 | **49** | 0.00225 | 0.00704 | **90** | 0.00528 | 0.02691 |
| 9 | 0.00057 | 0.00044 | **50** | 0.00238 | 0.00828 | **91** | 0.00475 | 0.02382 |
| 10 | 0.00074 | 0.00051 | **51** | 0.00298 | 0.00759 | **92** | 0.00411 | 0.0243 |
| 11 | 0.00068 | 0.00052 | **52** | 0.00313 | 0.00923 | **93** | 0.00443 | 0.02245 |
| 12 | 0.00063 | 0.00046 | **53** | 0.00265 | 0.00834 | **94** | 0.00372 | 0.02405 |
| 13 | 0.00062 | 0.00069 | **54** | 0.00283 | 0.00767 | **95** | 0.00392 | 0.0224 |
| 14 | 0.00065 | 0.00065 | **55** | 0.00275 | 0.0094 | **96** | 0.00482 | 0.0232 |
| 15 | 0.00078 | 0.00079 | **56** | 0.00254 | 0.00852 | **97** | 0.00464 | 0.02819 |
| 16 | 0.00092 | 0.00078 | **57** | 0.00253 | 0.00851 | **98** | 0.00444 | 0.02436 |
| 17 | 0.00084 | 0.00098 | **58** | 0.0025 | 0.00872 | **99** | 0.00406 | 0.02621 |
| 18 | 0.00093 | 0.00096 | **59** | 0.00248 | 0.00922 | **100** | 0.00442 | 0.02456 |
| 19 | 0.00089 | 0.00114 | **60** | 0.00295 | 0.00947 | **83** | 0.00406 | 0.01999 |
| 20 | 0.00092 | 0.00122 | **61** | 0.00319 | 0.0115 | **84** | 0.00386 | 0.01797 |
| 21 | 0.00094 | 0.00139 | **62** | 0.00311 | 0.01078 | **85** | 0.0044 | 0.02462 |
| 22 | 0.00101 | 0.00163 | **63** | 0.00269 | 0.0111 | **86** | 0.00386 | 0.01994 |
| 23 | 0.00114 | 0.00186 | **64** | 0.00255 | 0.0111 | **87** | 0.00354 | 0.01945 |
| 24 | 0.00117 | 0.00178 | **65** | 0.00253 | 0.01156 | **88** | 0.00413 | 0.01925 |
| 25 | 0.00117 | 0.00199 | **66** | 0.00309 | 0.01189 | **89** | 0.00363 | 0.02126 |
| 26 | 0.00179 | 0.00349 | **67** | 0.00345 | 0.01167 | **90** | 0.00528 | 0.02691 |
| 27 | 0.00157 | 0.00307 | **68** | 0.00329 | 0.01158 | **91** | 0.00475 | 0.02382 |
| 28 | 0.00143 | 0.00246 | **69** | 0.00395 | 0.01302 | **92** | 0.00411 | 0.0243 |
| 29 | 0.00137 | 0.00264 | **70** | 0.00338 | 0.01519 | **93** | 0.00443 | 0.02245 |
| 30 | 0.00117 | 0.0024 | **71** | 0.00553 | 0.01942 | **94** | 0.00372 | 0.02405 |
| 31 | 0.00142 | 0.00296 | **72** | 0.00344 | 0.01514 | **95** | 0.00392 | 0.0224 |
| 32 | 0.00136 | 0.00362 | **73** | 0.00307 | 0.01476 | **96** | 0.00482 | 0.0232 |
| 33 | 0.00161 | 0.00362 | **74** | 0.00327 | 0.01399 | **97** | 0.00464 | 0.02819 |
| 34 | 0.00176 | 0.00391 | **75** | 0.00339 | 0.01485 | **98** | 0.00444 | 0.02436 |
| 35 | 0.00229 | 0.00511 | **76** | 0.00324 | 0.01449 | **99** | 0.00406 | 0.02621 |
| 36 | 0.00194 | 0.00368 | **77** | 0.00348 | 0.01524 | **100** | 0.00442 | 0.02456 |
| 37 | 0.00224 | 0.00391 | **78** | 0.00326 | 0.01446 |  |  |  |
| 38 | 0.0019 | 0.0039 | **79** | 0.00381 | 0.01823 |  |  |  |
| 39 | 0.00168 | 0.00452 | **80** | 0.00373 | 0.0172 |  |  |  |
| 40 | 0.00174 | 0.00446 | **81** | 0.00372 | 0.018 |  |  |  |
| 41 | 0.00216 | 0.00494 | **82** | 0.00347 | 0.01639 |  |  |  |

**Appendix M – Experimental and predicted results for basic operation**