CAB301 Assignment 1
Average Case Efficiency Comparison: Brute Force Median vs Median

Student name: Thomas Feldman
(Student no. n8306699)
Date submitted: 15 April 2018

Summary

The following report will analyse the average-case efficiency comparison of the Brute Force Median and Median Algorithms. Firstly, I will explain what each algorithm does, and define the basic operations of both. In doing so I will show the theoretical average case efficiency, before moving on to experimental testing to determine whether the implementation introduces additions variables not accounted for. By the end of this paper, you should be able to see the trend that occur within the data, and how closely the data matches theory.

1. Algorithm Definition

Below defines both the Brute Force Median algorithm, and the Median algorithms.

1.1 Brute Force Median Algorithm

The brute force median algorithm is an iterative algorithm used to find a median value within an array of length n. This algorithm works by setting a middle index position of the array to a variable which will be used to determine whether a median value has been found. The algorithm has two for loops which are used to search the array. The first loop sets its index position “i” to 0 for the loop to start at the first position of array A. The second loop does the same for “j”, and proceeds to iterate the loop, comparing each index j to each index i. If the index j is smaller than index i, than it will increment a variable numsmaller, else, it will check if it is equal and increment that variable numequal. It does this for all values of j then breaks the loop and checks if the middle index value is greater than the numsmaller, but less that both numsmaller and numequal. The algorithm increments i, and the inner loop starts again, until i can no longer increment. From this, the median value will be determined. The median value given will always be the smaller value of the two middle number if the array length is even.

1.2 Median Algorithm

Median is a recursive algorithm, which employs the use of 2 other methods to return a median answer. The median algorithm starts by checking if the array size is length 1, returning the value as the median answer. If the array length is greater than one, it will pass four index positions as arguments to the Select method; The array A, the first index position, the middle index position, and the last index position. For each call to Select, these index values remain the same, however each iteration will partition the array, thus the length of array A will halve per iteration.

Select will initially take the array A, the first index position, and the last index position arguments and pass them to Partition to return an int value. Partition stores the initial position index, and the initial index value in separate variables, before looping. The loop increments the initial index position by 1 as a starting point for the loop, running from that value, until the end of the array length. Partition checks if this value is smaller than the value at the initial index position, and if so, swaps the values and increments the pivot location index position value. It does this for all values of j, and after breaking the loop, swaps the values for the value at saved initial index position, with the value for the pivot location index position, before returning that index position as the int value passed back to Select.

With a value now returned to Select, it begins by comparing this value to the value given for the middle index position passed by median, returning the value at this index position if true. If false, it will then check if the value returned from Partition is greater than the middle index value passed from median, and pass the value initially back into Select, with the last index value being the value returned from Partition minus one. If neither of those two options are correct, it will pass the initial values back into Select, with the first index value being the value returned from Partition plus one. Select will continue removing sections of the array until there is a match found. It is worth noting that for even length array's, Median will always return the larger of the two median values.

2. Theoretical Analysis of the Algorithm

2.1 Basic Operation

For the Brute Force Median Algorithm, the choice of basic operation was the comparison of A[j] and A[i]. This comparison is the if statement inside the inner loop, and is the basic operation for the algorithm because this is the calculation performed the most. Every A[j] will be compared to every A[i], making the algorithm perform this comparison for the input size^2.

For the Median algorithm, the basic operation chosen was the comparison between A[j] and pivotval. This comparison sits inside the for loop of Partition; and was chosen because this is the most used calculation. As this calculation is used for every value of A[j], comparing to the length of the partition, it is evident that it will be performed the most. Furthermore, it can be seen that Select must run the Partition every time it is called, thus it will always run this for loop, unless the array length is 1, or the median is already found.

2.2 Problem Size

The problem size used for this experiment was started from 1, incrementing by 1 until the array length reached 100. The array length would then increment by 25, until the length reached 10000. Any further size increase would cause a stackoverflow and as such I terminated the tests here. As each test was conducted 100 times per array length, the test data is accurate enough that we can see trends with this data size.

2.3 Average Case Efficiency

As stated in the specifications, the expected efficiency of the Brute Force Median algorithm appears to be quadratic. This can be proven theoretically by simply looking at the function of the algorithm. As the algorithm has two for loops, both referencing variables used as index position, we can see that for each index position of the array, the array will compare each index position to it. Therefore, we can see that for an array of length 2, the outer loop will use both values as it's index, while the inner loop will increment its index position for the comparisons; such that it will loop 2 * 2 times for it's index values. Therefore, we can assume that for an array length n, it will loop n * n times, or n^2.

For the Median Algorithm, to theoretically predict the average efficiency we can look into the Quicksort algorithm as it is closely related. As stated in the specification, we know that the worst-case efficiency is known to be quadratic. From readings, we can see that if the median algorithm is optimal, meaning O(*n*), then the resulting general selection algorithm is also optimal, again meaning linear. This is because Quickselect is a decrease and conquer algorithm, and using the median at each pivot means that at each step the search set decreases by half in size. If good partitions occur

this can make the average case efficiency linear, as discussed in the specifications. However, upon further reading, other sources claim that average case expectations should be a bit higher. This can be linked to sorting an array of *n* distinct elements. Quicksort takes $O(n \log n)$ time in expectation, averaged over all $n!$ permutations of *n* elements with equal probability.

3. Methodology, Tools, and Techniques

1. Experiments run on a desktop PC, running an AMD 8 core fx-8350 @ GHz, gtx 1050ti @ 1354 MHz and 4GB GDDR5, 16gb DDR3 RAM @ 2133MHz, running Microsoft Windows 10 operating system. All algorithms and experiments were implemented using C# multi-paradigm programming language, and Microsoft Visual Studio Community 2017. Visual Studio Community is a free, fully featured and widely used IDE. One of the biggest advantages of using Visual Studio is its ItelliSense.

2. All data was tested using a randomly generated array with a variable input size. The test size per array length was 100. Array length started at 1, incrementing to 100. From 100 to 10000, I incremented by 25. This was to generate enough results per array length, and to see the effects of a larger array size.

3. The data was output via File.AppendAllText. This allowed data to be output to txt files for future reference.

4. Algorithm Implementation

4.1 Functional Correctness

Through testing we can see that the results are correct due to the answers display in [1]. For the array lengths 1-10, we can see that a correct answer was displayed for the arrays, including the even arrays, which we will see the Brute Force median selecting the smaller of the two middle values, and the median algorithm selecting the greater of the two middle values. As such, we can safely assume that the algorithms work correctly.

4.2 Data Analysis – Basic Operations

As you can see from graphs [4] and [5], the basic operation both produce clear trends which we can analyse. The input size for [4] is 1 through 100, while [5] is from 100 through 10000, testing at increments of 25. For individual number values, please see table [6]. From this we can show how each algorithm behaves in comparison to each other. As expected, the Brute force median algorithm behaves quadratically, with a basic operation time on n^2. This is seen in [6]. As for the Median algorithm, the results are not terribly unsurprising either.

4.3 Data Analysis – Execution Time

As you can see from graphs [2] and [3], the basic operation both produce clear trends which we can analyse. The input size for [2] is 1 through 100, while [3] is from 100 through 10000, testing at increments of 25. All tests at each input size were done 100 times each, and an average time was taken. These values represent microsecond averages for each input size. For individual number values, please see table [6]. From this we can show how each algorithm behaves in comparison to each other. As seen in graphs [2] and [3] the Brute Force Median algorithm behaves exactly as predicted theoretically. As input size increase, the execution time increases exponentially, such that the resulting trend can be seen as almost perfectly quadratic in nature, some variance in this could be cause by operating environment.

The median function however performed somewhat differently than was predicted theoretically. While the trend produced is neither linear or truly quadratic; it fails to meet what could have been expected when compared to an average case expectancy of Quicksort or Quickselect. These results could be explained by the operational environment that the tests were conducted in.

References

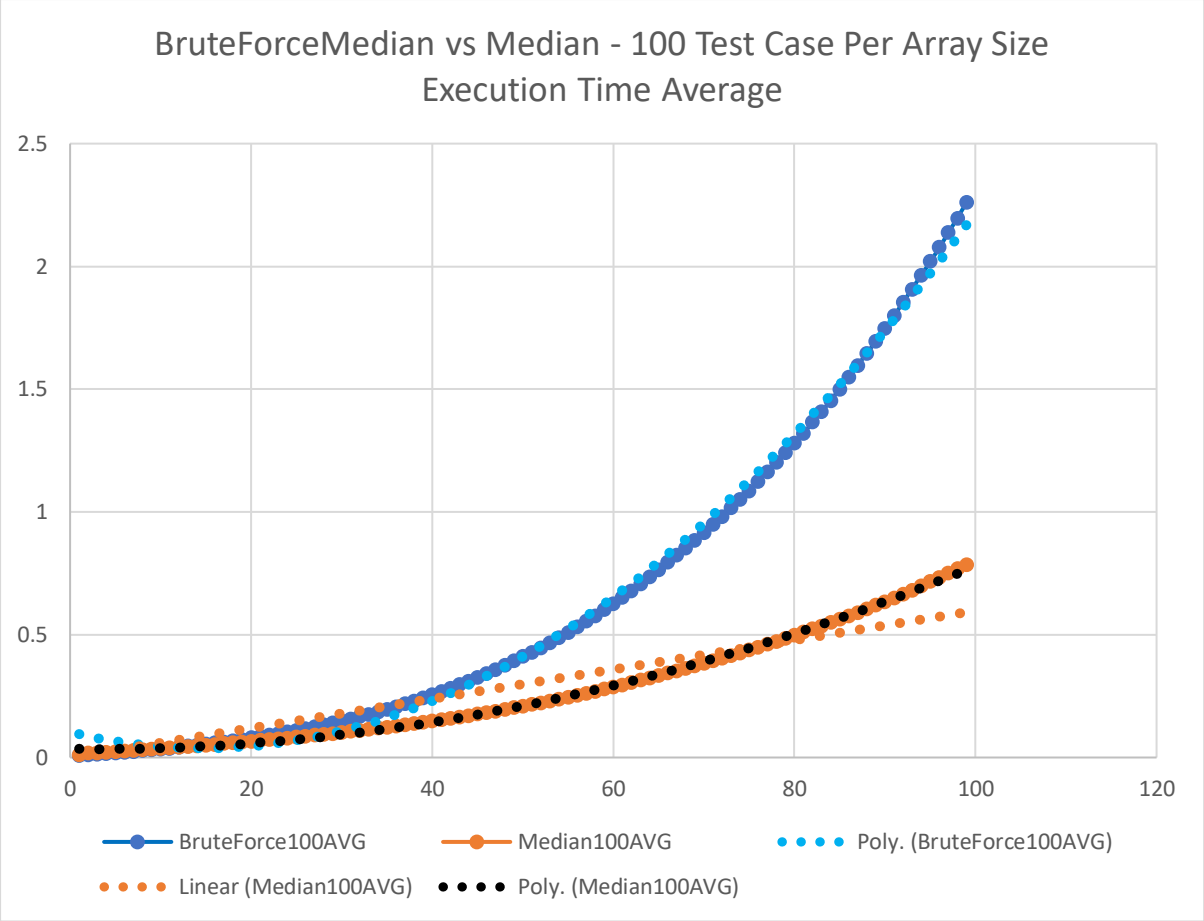
[1] Levitin, Anany. Pearson, 2012. "*Introduction to the design & analysis of algorithms, 3rd ed.*".

[2] Donald Knuth. Addison-Wesley, 1997. "*The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition".

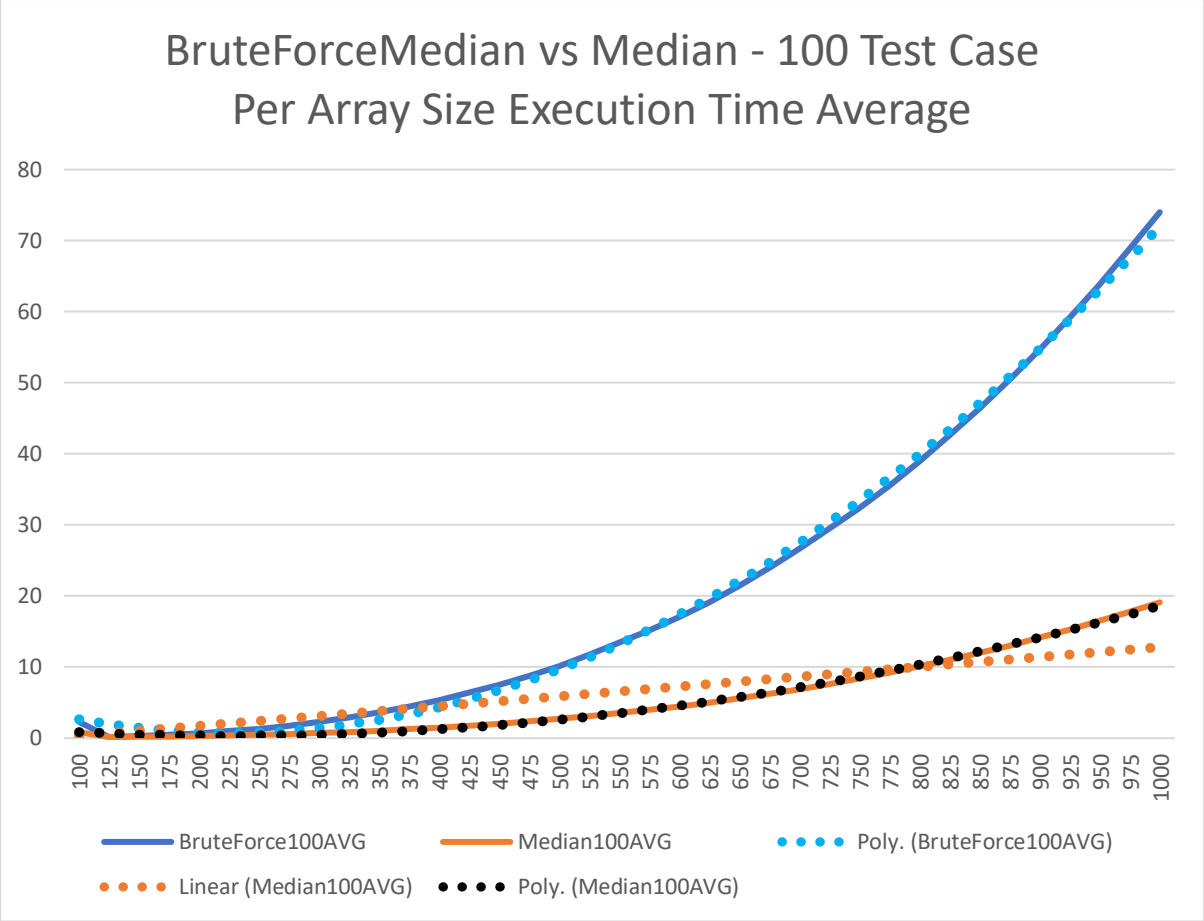
Appendix

	BruteForceMedianAnswers	MedianAnswers
Array Length 1: 1	1	1
Array Length 2: 1, 2	1	2
Array Length 3: 3, 4, 5	4	4
Array Length 4: 4, 1, 1, 6	1	4
Array Length 5: 1, 8, 8, 8, 1	8	8
Array Length 6: 5, 1, 3, 1, 1, 8	1	3
Array Length 7: 5, 8, 4, 7, 7, 11, 3	7	7
Array Length 8: 14, 6, 2, 8, 11, 13, 4, 7	7	8
Array Length 9: 5, 10, 8, 13, 12, 1, 1, 9, 2	8	8
Array Length 10: 16, 7, 6, 15, 18, 3, 2, 2, 18, 18	7	15

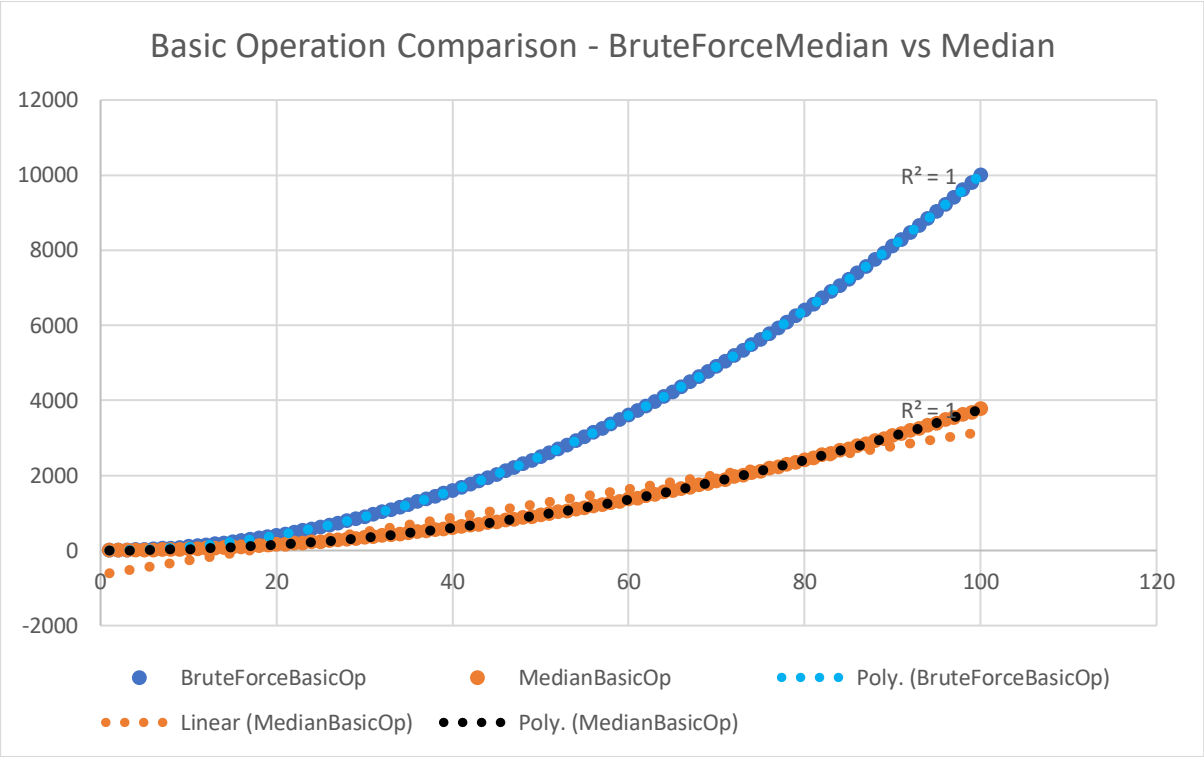
(Appendix 1: Median Answers for both Algorithms per array length 1-10)



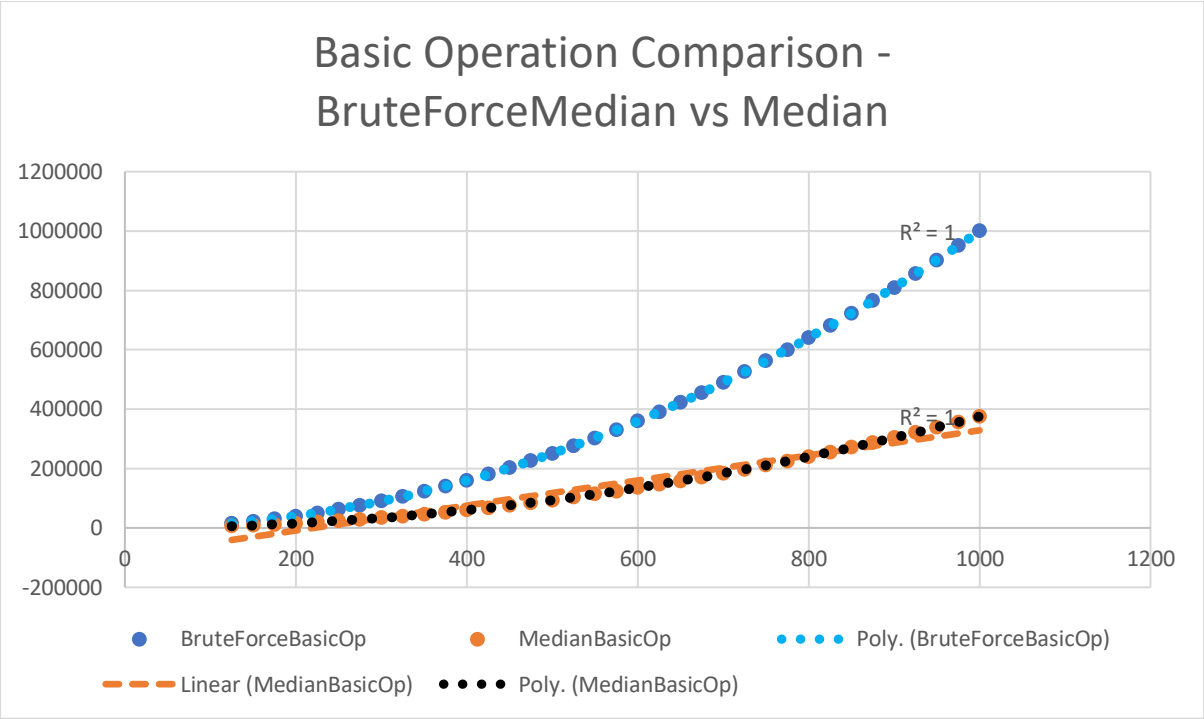
(Appendix 2: Average execution time for each algorithm for 100 tests per array length)



(Appendix 3: Average execution time for each algorithm for 100 tests per array length)



(Appendix 4: Basic operations performed for each algorithm per array length)



(Appendix 5: Basic operations performed for each algorithm per array length)

Appendix 6: Basic Operations

Array length	BruteForce100AVG	Median100AVG	BruteForceBasicOp	MedianBasicOp
1	0.007192	0.00962	1	0
2	0.009784	0.01733	4	1
3	0.012626	0.019685	9	3
4	0.015428	0.021982	16	6
5	0.018292	0.024147	25	9
6	0.02143	0.026337	36	14
7	0.024698	0.028794	49	18
8	0.027885	0.031394	64	25
9	0.031092	0.033729	81	30
10	0.034529	0.036268	100	39
11	0.037973	0.038693	121	45
12	0.041869	0.041643	144	56
13	0.045864	0.044281	169	63
14	0.049897	0.047099	196	76
15	0.054325	0.050078	225	84
16	0.05898	0.053119	256	99
17	0.064087	0.056462	289	108
18	0.069011	0.059559	324	125
19	0.074446	0.062967	361	135
20	0.080161	0.066589	400	154

21	0.085881	0.069799	441	165
22	0.091486	0.073055	484	186
23	0.097548	0.076313	529	198
24	0.103795	0.079573	576	221
25	0.110395	0.082911	625	234
26	0.117326	0.086676	676	259
27	0.124401	0.090287	729	273
28	0.132012	0.093996	784	300
29	0.139789	0.097753	841	315
30	0.148185	0.101899	900	344
31	0.156651	0.105856	961	360
32	0.165852	0.110308	1024	391
33	0.176145	0.115025	1089	408
34	0.185763	0.119583	1156	441
35	0.196113	0.123969	1225	459
36	0.207463	0.128796	1296	494
37	0.219389	0.133688	1369	513
38	0.230824	0.138486	1444	550
39	0.243184	0.143436	1521	570
40	0.255765	0.148586	1600	609
41	0.26868	0.153861	1681	630
42	0.282277	0.159481	1764	671
43	0.296182	0.164982	1849	693
44	0.311062	0.170876	1936	736
45	0.326379	0.176894	2025	759
46	0.341892	0.182763	2116	804
47	0.357538	0.188682	2209	828
48	0.376592	0.196641	2304	875
49	0.394286	0.203016	2401	900
50	0.411315	0.209398	2500	949
51	0.42886	0.215854	2601	975
52	0.447285	0.222707	2704	1026
53	0.466587	0.229641	2809	1053
54	0.487175	0.237269	2916	1106
55	0.508198	0.245124	3025	1134
56	0.532167	0.253714	3136	1189
57	0.554303	0.261924	3249	1218
58	0.577156	0.270045	3364	1275
59	0.601286	0.278141	3481	1305
60	0.62586	0.28629	3600	1364
61	0.651805	0.2959	3721	1395
62	0.678973	0.305807	3844	1456
63	0.707851	0.315511	3969	1488
64	0.736267	0.324566	4096	1551

65	0.765065	0.33422	4225	1584
66	0.794709	0.344042	4356	1649
67	0.824188	0.353345	4489	1683
68	0.853832	0.363376	4624	1750
69	0.883439	0.37336	4761	1785
70	0.915649	0.383999	4900	1854
71	0.948983	0.394231	5041	1890
72	0.98085	0.404568	5184	1961
73	1.016752	0.415632	5329	1998
74	1.050849	0.426464	5476	2071
75	1.085077	0.437347	5625	2109
76	1.125049	0.44961	5776	2184
77	1.163275	0.461311	5929	2223
78	1.201246	0.473236	6084	2300
79	1.24109	0.485302	6241	2340
80	1.280189	0.497392	6400	2419
81	1.319783	0.510181	6561	2460
82	1.365495	0.523118	6724	2541
83	1.407859	0.535911	6889	2583
84	1.45327	0.549326	7056	2666
85	1.500633	0.56238	7225	2709
86	1.548072	0.576184	7396	2794
87	1.597068	0.590056	7569	2838
88	1.645518	0.604811	7744	2925
89	1.695286	0.61983	7921	2970
90	1.746938	0.634356	8100	3059
91	1.798123	0.6497	8281	3105
92	1.853061	0.665908	8464	3196
93	1.90656	0.681749	8649	3243
94	1.963304	0.698953	8836	3336
95	2.020875	0.716351	9025	3384
96	2.078185	0.733085	9216	3479
97	2.138223	0.750441	9409	3528
98	2.196526	0.768465	9604	3625
99	2.259537	0.786213	9801	3675
100	2.321014	0.805551	10000	3774

Array length	BruteForce100AVG	Median100AVG	BruteForceBasicOp	MedianBasicOp
100	2.321014	0.805551	10000	3774
125	0.156336	0.156062	15625	5859
150	0.287066	0.193761	22500	8474
175	0.460579	0.241359	30625	11484
200	0.681048	0.304794	40000	15049
225	0.982373	0.384041	50625	18984

250	1.336298	0.476407	62500	23499
275	1.7758	0.593381	75625	28359
300	2.328414	0.727393	90000	33824
325	2.941832	0.880812	105625	39609
350	3.600923	1.058193	122500	46024
375	4.418273	1.266651	140625	52734
400	5.361264	1.501414	160000	60099
425	6.468537	1.757103	180625	67734
450	7.526646	2.044683	202500	76049
475	8.734855	2.363755	225625	84609
500	10.096358	2.716346	250000	93874
525	11.755606	3.105747	275625	103359
550	13.427248	3.52265	302500	113574
575	15.19845	3.977079	330625	123984
600	17.083771	4.477589	360000	135149
625	19.117119	5.011973	390625	146484
650	21.435409	5.601072	422500	158599
675	23.964441	6.233193	455625	170859
700	26.702499	6.913362	490000	183924
725	29.493316	7.636006	525625	197109
750	32.382397	8.41559	562500	211124
775	35.523327	9.231728	600625	225234
800	38.917816	10.105712	640000	240199
825	42.647055	11.027277	680625	255234
850	46.45009	12.008412	722500	271149
875	50.493562	13.033083	765625	287109
900	54.724177	14.137027	810000	303974
925	59.161868	15.293332	855625	320859
950	63.808721	16.500818	902500	338674
975	68.85488	17.77459	950625	356484
1000	74.011364	19.114468	1000000	375249

Appendix 7: Program Code

Assignment2 | Assignment2.Program | BruteForceMedian(int[] A)

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Assignment2
{
    class Program
    {
        public static int[] arrayMedian;
        public static Stopwatch stopwatch = new Stopwatch();
        public static Stopwatch stopwatch2 = new Stopwatch();
        public static int basicOperationCounter;
        public static int basicOperationCounter2;
        public static double timeAvg1 = 0;
        public static double timeAvg2 = 0;
        public static int medianBruteForce;
        public static int median;


        static void Main(string[] args)
        {
            for (int i = 10; i <= 10; i++) {
                arrayMedian = CreateArray(i, (i*2));

                for (int j =10; j <= 10; j++) {
                    basicOperationCounter = 0;
                    stopwatch.Start();
                    medianBruteForce = BruteForceMedian(arrayMedian);
                    stopwatch.Stop();
                    File.AppendAllText(@"E:\Thomas's Shit\n8306699\CAB301\ASSIGNMENT 2 OUTPUT\BruteForceMedianMedianTimes.txt", stopwatch.Elapsed.TotalMilliseconds + Environment.NewLine);
                    timeAvg1 += stopwatch.Elapsed.TotalMilliseconds;
                    stopwatch.Reset();

                    basicOperationCounter2 = 0;
                    stopwatch2.Start();
                    median = Median(arrayMedian);
                    stopwatch2.Stop();
                    timeAvg2 += stopwatch2.Elapsed.TotalMilliseconds;
                    stopwatch2.Reset();
                    File.AppendAllText(@"E:\Thomas's Shit\n8306699\CAB301\ASSIGNMENT 2 OUTPUT\MedianTimes.txt", stopwatch2.Elapsed.TotalMilliseconds + Environment.NewLine);
                }
               // File.AppendAllText(@"E:\Thomas's Shit\n8306699\CAB301\ASSIGNMENT 2 OUTPUT\BruteForceMedianMedianTimes.txt", Environment.NewLine);
                //File.AppendAllText(@"E:\Thomas's Shit\n8306699\CAB301\ASSIGNMENT 2 OUTPUT\BruteForceMedianMedianAnswer.txt", Environment.NewLine);
                //File.AppendAllText(@"E:\Thomas's Shit\n8306699\CAB301\ASSIGNMENT 2 OUTPUT\BruteForceMedianMedianBasicOperations.txt", Environment.NewLine);
                //File.AppendAllText(@"E:\Thomas's Shit\n8306699\CAB301\ASSIGNMENT 2 OUTPUT\MedianTimes.txt", Environment.NewLine);
                //File.AppendAllText(@"E:\Thomas's Shit\n8306699\CAB301\ASSIGNMENT 2 OUTPUT\MedianAnswer.txt", Environment.NewLine);
                //File.AppendAllText(@"E:\Thomas's Shit\n8306699\CAB301\ASSIGNMENT 2 OUTPUT\MedianBasicOperations.txt", Environment.NewLine);
                //File.AppendAllText(@"E:\Thomas's Shit\n8306699\CAB301\ASSIGNMENT 2 OUTPUT\ArrayDisplay.txt", Environment.NewLine);
                File.AppendAllText(@"E:\Thomas's Shit\n8306699\CAB301\ASSIGNMENT 2 OUTPUT\AvgFrom100BruteForceMedianMedianTimes.txt", (timeAvg1 / 100) + Environment.NewLine);
                File.AppendAllText(@"E:\Thomas's Shit\n8306699\CAB301\ASSIGNMENT 2 OUTPUT\AvgFrom100MedianTimes.txt", (timeAvg2 / 100) + Environment.NewLine);
```

```
64      }
65
66      public static int[] CreateArray(int length, int randLength)
67      {
68          arrayMedian = new int[length];
69          Random randNum = new Random();
70          for (int i = 0; i < arrayMedian.Length; i++)
71          {
72              arrayMedian[i] = randNum.Next(1, randLength);
73          }
74          File.AppendAllText(@"E:\Thomas's Shit\n8306699\CAB301\ASSIGNMENT 2 OUTPUT\ArrayDisplay.txt", "Array Length " + arrayMedian.Length + ": ");
75          for (int i = 0; i < arrayMedian.Length; i++)
76          {
77              File.AppendAllText(@"E:\Thomas's Shit\n8306699\CAB301\ASSIGNMENT 2 OUTPUT\ArrayDisplay.txt", arrayMedian[i].ToString() + ", ");
78          }
79          File.AppendAllText(@"E:\Thomas's Shit\n8306699\CAB301\ASSIGNMENT 2 OUTPUT\ArrayDisplay.txt", Environment.NewLine);
80          return arrayMedian;
81      }
82
83      public static int BruteForceMedian(int[] A)
84      {
85          int k = (int)Math.Round((double)A.Length/2, MidpointRounding.AwayFromZero);
86          int median = A[0];
87          for (int i = 0; i < A.Length; i++)
88          {
89              int numsmaller = 0;
90              int numequal = 0;
91              for (int j = 0; j < A.Length; j++)
92              {
93                  basicOperationCounter++;
94                  if (A[j] < A[i])
95                  {
96                      numsmaller++;
97                  }
98                  else
99                  {
100                     if (A[j] == A[i])
101                     {
102                         numequal++;
103                     }
104                 }
105             }
106             Console.WriteLine("Here");
107             if (numsmaller < k && k <= (numsmaller + numequal))
108             {
109                 median = A[i];
110             }
111         }
112
113         return median;
114     }
115
116     public static int Median(int[] A)
117     {
118         if (A.Length == 1)
```

0 %

```
115
116         public static int Median(int[] A)
117         {
118             if (A.Length == 1)
119             {
120                 return A[0];
121             }
122             else
123             {
124                 return Select(A, 0, A.Length / 2, A.Length - 1);
125             }
126         }
127
128         public static int Select(int[] A, int l, int m, int h)
129         {
130
131             int pos = Partition(A, l, h);
132             if (pos == m)
133             {
134                 return A[pos];
135             }
136             else if (pos > m)
137             {
138                 return Select(A, l, m, pos - 1);
139             }
140             else
141             {
142                 return Select(A, pos + 1, m, h);
143             }
144         }
145
146         public static int Partition(int[] A, int l, int h)
147         {
148             int pivotval = A[l];
149             int pivotloc = l;
150             for (int j = l + 1; j <= h; j++)
151             {
152                 basicOperationCounter2++;
153                 if (A[j] < pivotval)
154                 {
155                     pivotloc++;
156                     swap(ref A[pivotloc], ref A[j]);
157                 }
158             }
159             swap(ref A[l], ref A[pivotloc]);
160             return pivotloc;
161         }
162
163         public static void swap(ref int a, ref int b)
164         {
165             int temp = a;
166             a = b;
167             b = temp;
168         }
169     }
```