

Empirical Analysis of Two Algorithms

Cab301 Algorithms and Complexity - Assignment 2

Hugh McIntyre | N9169614

Shane Plater | N1234567

21/05/2018

# Abstract

This report analyses two different algorithms which find the median from a list of numbers, the first algorithm is uses the Brute force method, the second algorithm uses a selection problem algorithm. This Brute Force Median algorithm takes an array of a given length as an input and outputs the median of the given array. The selection problem algorithm involves three separate procedures …. Briefly explain input and output of algorithm …. A theoretical analysis of each algorithm is undertaken to find the average case efficiency level. Each algorithm is implemented and experimental tests are conduction to ensure that the algorithms work effectively and efficiently. The results for the experimental tests are analysed and compared with the theoretical conclusions to ensure that the algorithms efficiency is consistent with the theoretical results.

# 1. Description of the Algorithm

The Brute Force Median algorithm finds the median of a given array of length *n.* The median is the *k*th element of an array of length n, where *k* = |*n*/2|. The algorithm then loops through the array use two for to find the *k*th element and finally the value of the *k*th element is outputted.

# 2. Theoretical Analysis of the Algorithm

## 2.1 Identify thE Alorithms Baisc Operations

The algorithms basic operation is the operation in the code which is the most influential to the total running time. In the case of the Brute Force median algorithm the basic operation is most likely to be the in the inner for loop with the comparison operations (Shown in Figure 1, line 5-10). This operation is performed significantly more than the other operations.

## 2.2 Average case efficiency

The average case efficiency can be found by analysing the algorithm with respect to the length ‘n’. The Brute force algorithm has quadratic efficiency (n2), this is do the two for loop statements in the algorithm. Each for loops statement have an efficiency of *n* with two for loops the calculations are as follows:

With this prediction the output of the experimental results for the BruteForceMedian algorithm is expected to be n2 . In numerical form we can predict that a value 2500 with have approximately 6250000 with the graphical representation expected to show a quadratic shape.

# 3. Methodology, Tools, Techniques

## 3.1 Experimental Environment

This section will describe the computer environment used to implement the algorithms and will briefly explain how the Test cases for the experiment were produced,

- IDE

-Hardware

- where and how pictures were drawn

-anything that would be need to duplicate the experiment.

To test the efficiency and performance of the algorithm, the basic operation and execution time needed to be measured. These were tested with several array between the scale of 1000 to 100,000 with the intervals of 1000. Each increment was tested 100 times with a different input to ensure that the average result was satisfactory.

# 4. Functional Testing

## 4.1 Experimental Design

### 4.1.1 Design of functional tests

The algorithm in shown in figure 1 was implemented using C# in visual studio. The implementation for the BruteForceMedian is show in Appendix A. The functionality of the algorithm was test by using several manual test cases. General and extreme tests were conduction to test the correctness of the algorithm. The test cases are listed below.

1.

2.

3.

4.

5.

4.1.2 Design of performance tests

The performance tests were conducted to measure the number of basic operation for each

# 5.0 Experimental Results

## 5,1 Results of Functional Tests

The m = 10, n = 10 is supposed to give an output as: …..

Its test output was: ……

The test output matches exactly the ther

## 5.2 Average-Case Number of Basic Operations

To calculate the number of basic operations the algorithm needed to be modified slightly to implement a counter variable at relevant points in the code. The counter variable increments every time the basic operation is used. The code in Appendix C shows a modified version of the enhanced bubble sort which is used to calculate the number of basic operations. It takes the average of 10 randomly generated arrays (using the code from Appendix F) which range in size from 1000 to 10,000 (shown in Appendix D). The results are plotted and displayed in Figure 3. The plot is shown to have a quadratic shape which is consistent to the predicted result in section 2,2. The prediction stated that an array size of 1,500 should have approximately 2,250,000 number of basic operation. The experimental results show that an array size of 1500 has 2,183,293 number of basic operations.

## 5.3 Average-Case Execution Time

To measure the execution time, the program needed to be modified to include the System.Diagnostics class which allows the program to use an inbuilt stopwatch function (Shown in Appendix E). The program measures the average execution time of 10 randomly generated arrays. The code in the Appendix E records how long it takes for each randomly generated array to be sorted. The experimental results shown in figure 5 are consistent with the predictions the graphs shows a quadratic shape.

# References

|  |  |
| --- | --- |
| [1] | A. Levitin, Introduction to The Design and Analysis of Algorithms, Addison-Wesley, 2007. |
| [2] | J. Alnihoud and R. Mansi , “An Enhancement of Major Sorting Algorithms,” 27 May 2008. [Online]. Available: http://ccis2k.org/iajit/PDF/vol.7,no.1/9.pdf. [Accessed 15 04 2018]. |
| [3] | Tutorials Point, “DAA- Bubble Sort,” 2018. [Online]. Available: https://www.tutorialspoint.com/design\_and\_analysis\_of\_algorithms/design\_and\_analysis\_of\_algorithms\_bubble\_sort.htm. [Accessed 15 04 2018]. |
| [4] | Interactive Python, “Problem Solving with Algorithms and Data Structures - 5.7 The Bubble Sort,” Runestone, 2015. [Online]. Available: http://interactivepython.org/runestone/static/pythonds/SortSearch/TheBubbleSort.html. [Accessed 13 04 2018]. |
| [5] | QUT CAB301 Algorithms and Complexity, “Workshop 2, Exercise 2/3,” 2018. [Online]. Available: https://blackboard.qut.edu.au/bbcswebdav/pid-7219568-dt-content-rid-10982915\_1/xid-10982915\_1. [Accessed 28 March 2018]. |
| [6] | S. Shastri, . M. Vibhakar and S. Anand , “Sorting Algorithms and their Run-Time Analysis with C#,” 15 October 2015. [Online]. Available: https://ijcsits.org/papers/vol5no52015/6vol5no5.pdf. [Accessed 14 04 2018]. |

# Figures

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

// Returns the median value in a given array A of n numbers. This is

// the kth 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

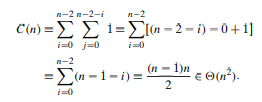
10. *numequal* ← *numequal* + 1

11. if *numsmaller* < *k* and *k* ≤ (*numsmaller* + *numequal*) then

12. return A[i]

## Figure 1:

The BetterBubbleSort algorithm that is analysed. The swap flag allows the algorithm to stop as soon as the array is sorted [1].



## Figure 2:

Levitin’s calculation for the number of key swaps [1].

## Figure 3:

Average number of basic operation. This graph shows the average number of basic operations for a given array size. 100 data points are shown with each point representing the average of 10 arrays. The results confirm that the number of basic operation grows quadratically to the size of the array (n2).

## Figure 4:

Average Execution Time for sorting an Algorithm. This graph shows the average execution time for sorting an array of a given size. 100 data points are shown with each data point representing the average execution time of 10 randomly generated arrays. The result grows quadratically with the size of the set.

# Appendices

## Appendix A – implemented Better BubbleSort Algoritm

The following appendix shows the C# code used to implement the better bubble sort algorithm. The swap flag in line 3, 4, 5 & 11 allows the program to be stopped as soon as the algorithm is sorted. The basic operation is the comparison ‘if’ statement in line 7. The code in lines 8, 9 & 10 are swapping the elements if Array[j + 1] is of greater value than Array[j] [4].

1. static void BetterBubbleSort(int[] Array) {

2. int count = Array.Length - 1;

3. bool sflag = true;

4. while(sflag == true) {

5. sflag = false;

6. for(int j = 0; j < count; j++) {

7. if(Array[j + 1] < Array[j]) {

// swap A[j] and A[j + 1];

8. int temp = Array[j];

9. Array[j] = Array[j + 1];

10. Array[j + 1] = temp;

11. sflag = true;

12. } // end if

13. }// end for

14. } // end while

15. }// end BubbleSort

## Appendix B – Functional Testing

The following code test the functionality of the implemented algorithm. The output of this algorithm is described in Section 4. The program uses 4 manually created arrays to test if the outputs are returned sorted. It uses ascending, descending, random and duplicate arrays to ensure that the result is as expected. The program prints the before and after results to the console screen.

//Test arrays.

1. int[] ascendingArray = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
2. int[] descendingArray = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };
3. int[] randomArray = { 7, 5, 3, 1, 8, 0, 9, 4, 2, 6 };
4. int[] duplicateArray = { 7, 7, 3, 1, 8, 1, 9, 4, 1, 7 };

//Length of Array

1. int arraylengths = 10;

//Print Ascending Array

1. System.Console.Write("\nascendingArray before: ");
2. PrintArray(ascendingArray, arraylengths);
3. BetterBubbleSort(ascendingArray);
4. Console.Write("ascendingArray after: ");
5. PrintArray(ascendingArray, arraylengths);

// Print Descending Array

1. Console.Write("\ndescendingArray before: ");
2. PrintArray(descendingArray, arraylengths);
3. BetterBubbleSort(descendingArray);
4. Console.Write("descendingArray after: ");
5. PrintArray(descendingArray, arraylengths);

//Print Random Array

1. Console.Write("\nrandomArray before: ");
2. PrintArray(randomArray, arraylengths);
3. BetterBubbleSort(randomArray);
4. Console.Write("randomArray after: ");
5. PrintArray(randomArray, arraylengths);

// Array with duplicate values.

1. Console.Write("\nduplicateArray before: ");
2. PrintArray(duplicateArray, arraylengths);
3. BetterBubbleSort(duplicateArray);
4. Console.Write("duplicateArray after: ");
5. PrintArray(duplicateArray, arraylengths);

## Appendix C – BetterBubbleSort modified with a Basic Operation Counter

The following code is a modified version of Appendix A. It implements a basic operation counter. The counter is incremented when the comparison operation is used. In line 11 when the Array[j+1] is smaller than Array[j] the counter is incremented. Since the counter is only incremented when Array[j+1] is smaller another ‘if’ statement needed to be added (line 8 & 9) for when Array[j+1] is larger than Array[j]. This insures that the basic operation counter is always incremented when the comparison is used, even when it is not true.

1. static int BetterBubbleSort(int[] Array) {

2. int basicOperations = 0;

3. int count = Array.Length - 1;

4. bool sflag = true;

5. while(sflag == true) {

6. sflag = false;

7. for(int j = 0; j < count; j++) {

8. if(Array[j + 1] > Array[j]) {

9. basicOperations++;

10. } // end if

11. if(Array[j + 1] < Array[j]) {

12. basicOperations++;

13. // swap A[j] and A[j + 1];

14. int temp = Array[j];

15. Array[j] = Array[j + 1];

16. Array[j + 1] = temp;

17. sflag = true;

18. } // end if

19. }// end for

20. } // end while

21. return basicOperations;

22. }// end BubbleSort

## Appendix D – Counting the Average number of Basic operations

This code creates 10 randomly generated arrays and counts the average number of basic operation with the use of the program in Appendix C. The first ‘for loop’ in line 3 creates 10 arrays of size 100 and increments by 100 to a maximum size of 10,000. The inner ‘for loop’ in line 5 uses the RandomArrayGenerator in Appendix F to generate 10 random arrays it is then sorted using the modified Bubble sort in Appendix C and counts the total number of basic operations. Line 10 calculates the average between the 10 arrays and prints the result to the console [5].

1. int numOfArrays = 10;
2. double averageCount = 0;
3. for(int size = 100; size <= 10000; size += 100) {
4. long totalCounts = 0;
5. for(int i = 1; i <= numOfArrays; i++) {
6. int[] A = RandomArrayGenerator(size);
7. int count = BetterBubbleSort(A);
8. totalCounts = totalCounts + count;
9. }// end inner for loop
10. averageCount = totalCounts \* 1.0 / numOfArrays;
11. Console.WriteLine("\n Size:" + size + " Average Count=" + averageCount);
12. } // end outer for loop
13. Console.ReadKey( )

[5]

## Appendix E – Count the Execution Time

This code uses the System.Diagnostics class Stopwatch to time the sorting algorithm. The program creates 10 randomly generated arrays and calculated the average execution time it takes for algorithms to be sorted. The outer ‘for loop’ creates an array ranging from size 100 to 10,000 in 100 step increments. The inner ‘for loop’ generates 10 random arrays using Appendix F, line 9 initiates the timer and the algorithm is sorted using the program in Appendix C. Line 11 stops the timer once the algorithm as completed sorting. Line 15 then calculates the average milliseconds of the 10 arrrays [5].

1. Stopwatch timer = new Stopwatch( );
2. int numOfArrays = 10;
3. double averageMillisecs = 0;
4. for(int array Size = 100; arraySize <= 1000; arraySize += 100) {
5. long totalMilliSecs = 0;
6. for(int i = 1; i < numOfArrays; i++) {
7. long milliSecs = 0;
8. int[] array = RandomArrayGenerator(arraySize);
9. timer.Start( );
10. BetterBubbleSort(array);
11. timer.Stop( );
12. milliSecs = timer.ElapsedMilliseconds;
13. totalMilliSecs = totalMilliSecs + milliSecs;
14. }//end inner for loop
15. averageMillisecs = totalMilliSecs \* 1.0 / numOfArrays;
16. Console.WriteLine("\nArray Size: " + arraySize + " Execution time (Millisec) = " + averageMillisecs);
17. }//end outer for loop
18. Console.ReadKey( );

[5]

## Appendix F- Random Array Generator

The following program generates a random array of a given size.

static int[] RandomArrayGenerator(int size) {

int[] array;

array = new int[size];

Random random = new Random();

for(int i = 0; i < array.Length; i++) {

array[i] = random.Next(Int32.MaxValue);

}

return array;

}

[5]