|  |
| --- |
| [Type the company name] |
| Analysis of Two Median Searching Algorithms |
| Submitted: 30th May 2016 |
|  |
| **Arjai Reynolds | Connor Jackson** |
| **9378600 | 9378855** |

|  |
| --- |
|  |

Contents

[Summary 2](#_Toc452234916)

[1 Description of the Algorithms 2](#_Toc452234917)

[2 Theoretical Analysis of the Algorithms 2](#_Toc452234918)

[2.1 Choice of Basic Operation 2](#_Toc452234919)

[2.2 Choice of Problem Size 2](#_Toc452234920)

[2.3 Average-Case Efficiency 2](#_Toc452234921)

[3 Methodology, Tools and Techniques 2](#_Toc452234922)

[3.1 Programming Environment 2](#_Toc452234923)

[3.2 Implementation of the Algorithms 2](#_Toc452234924)

[3.3 Generating Test Data and Running the Experiments 2](#_Toc452234925)

[4 Experimental Results 2](#_Toc452234926)

[4.1 Functional Testing 2](#_Toc452234927)

[4.2 Average-Case Number of Basic Operations 2](#_Toc452234928)

[4.3 Average-Case Execution Time 2](#_Toc452234929)

[References 3](#_Toc452234930)

# Summary

# 1 Description of the Algorithms

# 2 Theoretical Analysis of the Algorithms

## 2.1 Choice of Basic Operation

**BruteForceMedian**

The basic operation is chosen based on what influences the algorithm the most. There are several basic operations of this algorithm, however, one particular comparison stands out as it varies depending on the size of input and does the most work to solve the problem.

The comparison A[j] < A[i] has the greatest influence on the algorithm because it is used to compare each element to each other. The comparison A[j] == A[i] is only executed if A[j] < A[i] comparison fails, and is a special case that there are duplicates in the list. The comparison numSmaller <k and k<= (numSmaller + numEqual) is not considered as it is all performed at the end of each iteration of the outside loop and does not have a significant effect on the performance of the algorithm as this comparison uses the results of the A[j] < A[i] process.

## 2.2 Choice of Problem Size

The performance of the algorithms are relatively fast with small input sizes. To produce a trend, so that a meaningful comparison can be made, the chosen problem size was a list of unsorted elements with duplicate values that ranged from 2 elements up to 1000. With this problem size, a trend was able to be clearly seen.

## 2.3 Average-Case Efficiency

# 3 Methodology, Tools and Techniques

## 3.1 Programming Environment

1. The algorithms and test data to produce graphs were both implemented in C++
2. The experiments were performed on an Apple MacBook Pro, running a UNIX based OS. For the time to be accurately measured, all programs on the test computer were closed to avoid interruptions to the run time of the algorithm. To make a meaningful comparison, the same instance of the datasets were used to measure results for both algorithms. The data was random, to create a scenario where the median could be anywhere in the list. The data had natural numbers ranging from 0 to 250. Duplicate values were allowed.
3. To graph results, Microsoft Excel was used, in conjunction with the graphing tools. The program produced text files that were later imported into Excel using CSV. The graphs were then copy and pasted into the report.

## 3.2 Implementation of the Algorithms

The algorithms were implemented staying true to the pseudo-code to ensure accurate test results could be attained. Both algorithms rely on an array to find a median on. The algorithms were implemented in two separate classes. They were combined in the function (*testNumberOfOperations*) and (*testTime*), using the same instance of the array populated with data. Results were then written to a text file, then imported into excel.

## 3.3 Generating Test Data and Running the Experiments

To test the correctness of the algorithms, a test program was developed (*functionalTesting)* (Appendix ). This program simply outputted to the console the results of variables being assigned in main. To count the basic operations, a counter was declared in main, and a reference was passed into *testNumberOfOperations* (Appendix ). This counter was then incremented everytime a basic operation was performed. This was then averaged over 100 tests for each interval.

For time execution, the time taken for the algorithm to find the median was outputted to the textfile straight away after averaging over 100 tests.

# 4 Experimental Results

## 4.1 Functional Testing

To test the programs, datasets were manually created with the knowledge of what the expected answer is in mind. They were then fed into the program, to see what the algorithm would return as the median. This was then compared with what we expected.

**BruteForceMedian and Median**

int test1[9]= {4, 1, 10, 9, 7, 12, 8, 2, 15}

This dataset is a normal case with unique numbers.

int test2[9] = {1,2,3,4,5,6,7,8,9}

This dataset is numbers that are in ascending order.

int test3[9] = {9,8,7,6,5,4,3,2,1}

Numbers are in descending order.

int test4[8] = {4,1,10,9,7,12,8,2}

Numbers are out of order.

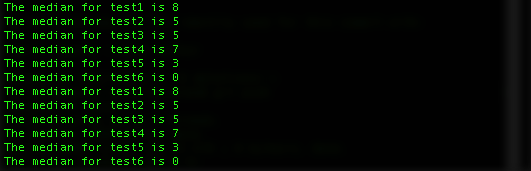
int test5[9] = {1,1,1,1,4,3,3,3,9};

Set contains duplicates .

int test6[9] = {-1,-2,-3,3,4,5,6,3,1};

Set contains negative numbers

The results are in the console screenshot below.



## 4.2 Average-Case Number of Basic Operations

Figure

To analyse the shape of the efficiency, the data was normalised (Figure 1). As it can be clearly seen, Median has a linear growth and BruteForceMedian has an exponential growth, just as expected. With this in mind, this is further evidence that Median is more efficient overall due to it’s linear growth versus BruteForceMedian’s exponential growth.

BruteForceMedians results are more spread than Medians because of the random nature of the algorithm. BruteForceMedian searches each and every single element. But, due to the ability to exit early when the median is found, the results become pseudo-random. The efficiency is largely dependent on where the median is located in the list. This makes the algorithm “unstable”.

Median however has a very stable scatter plot. It demonstrates that it tightly bounds a linear growth, with a stable way of finding the median using quicksort methodology.

Figure

To compare actual results, a raw data graph was made (Figure 2).

Median is simply a much better algorithm. At the upper end of the tests, Median reaches a maximum around 340. BruteForceMedian hits 38857 basic operations at the same interval. At the lower end at only an input size of 10, the basic operation count of the algorithms start to grow a part.

## 

## 4.3 Average-Case Execution Time

Figure

Looking at the shape of each of the lines, from an eye glance, they seem to match what the efficiency was theorised (Figure 3). The BruteForceMedian algorithm follows a quadratic growth on average. This confirms with the theoretical efficiency of the algorithm.The median algorithm follows a linear growth, which also confirms with what was theorised.

Due to the nature of a computer, the results were a little spread due to the allocation of resources and the CPU fluctuating.

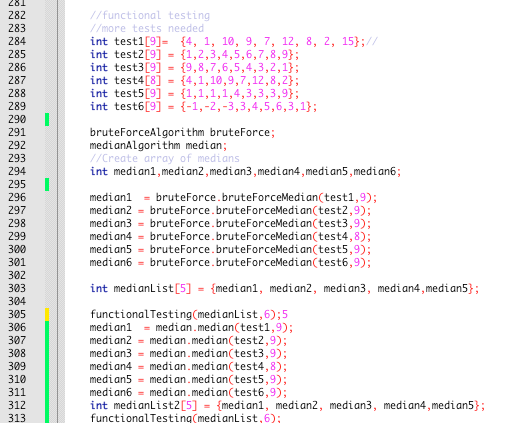
Figure

In this graph, the data was not normalised so actual values can be compared. It is obvious that Median is far more efficient overall than BruteForceMedian. At the top of the spectrum, the excection time for an input size of 985, the time was 1.97394ms. This compared to Medians 0.02156, it is clear that at this particular point, Median is more efficient. For very small inputs, the execution time is around the same, but when the input size is only 20, the two algorithms start to diverge. BruteForceMedian executes in 0.02156ms and Median at 0.00119 ms.

# References

Functional Testing

Appendix A



To test the algorithms, median variables were assigned the result of what each algorithm picked as the median. The algorithms were fed test arrays, with the knowledge of what the median of each array was. They were then printed out using the function *functionalTesting*. This function simply printed to the console the elements of the medianList array. The results in the console were compared to what was expected to occur, and it was concluded that the algorithms were accurate in finding the median of these arrays.

Apendix B

**testNumberOfOperations**



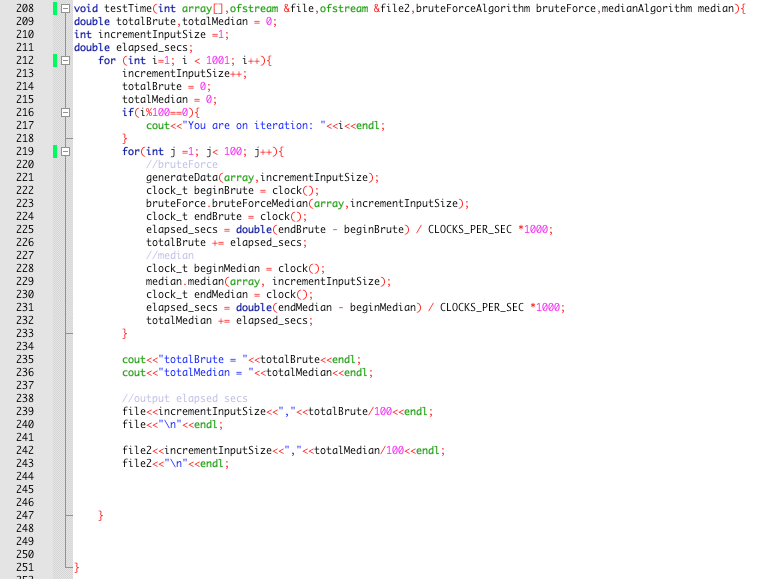
This function tested how many basic operations in both Median and BruteForceMedian. Two counters were declared in main.



These counters were then passed into *testNumberOfOperations*. In the implementation of the algorithm, it increased the counters. On line 263, this is where multiple tests were ran per interval. 100 iterations of generating the data, and incrementing the basic operation counters happened within this loop. At the end of the outside loop, the values of these counters were divided by 100 to attain the average, then they were written to a file. The size of input, then followed by a comma the average of the basic operation for that interval were written to this file. This was so Excel could understand which value was X and Y for use in graphing the results.

Appendix C

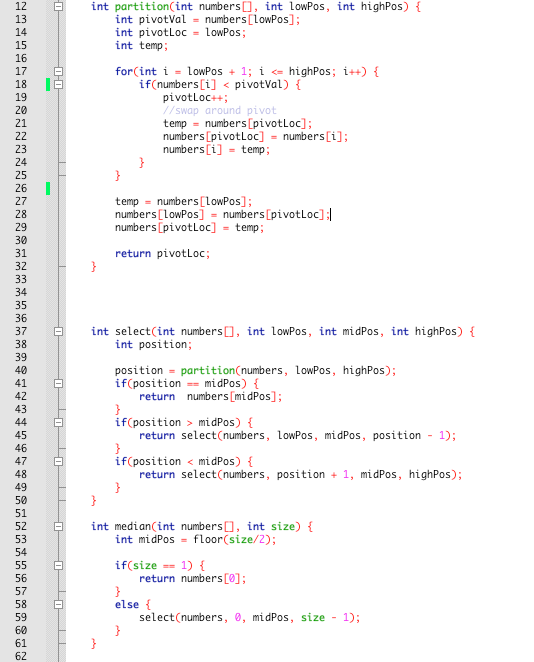
testTime



This function measured the execution time of the algorithms then wrote these results to a file. On line 222, the clock was started. Followed by this statement the algorithm was called. The next line then stopped the clock. This interval of time was then added to a total called totalBrute. The same process was done for the Median algorithm. The results of this were outputted on line 239, where they were averaged then written to the file.

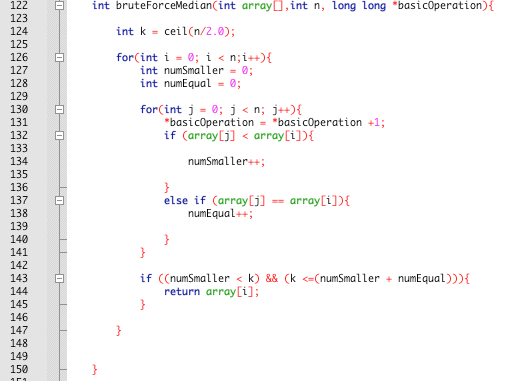
**Implementation**

**Median**

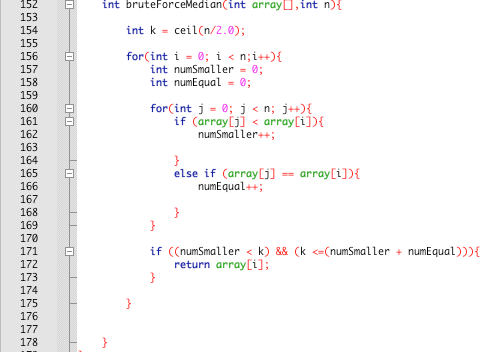


Median with basic operation implemented





This implementation of the algorithm has the counter embedded. It follows faithfully the pseudo-code given.



This implementation of the algorithm does not have the basic operation counter. This is so that the increase of the count would not affect execution times.