* Cover page = 1 page
* Table of contents = 1 page
* Description of the Algorithm <= 1 page
* Implementation of the Algorithm  <= 2 pages
* Design of experiments <= 2 pages
* Experimental results  <= 4 pages
* Analysis of experimental results <= 4 pages
* References < = 1 page
* Appendices <=  4 pages

# Executive summary (SHOULD BE WRITTEN AFTER THE REPORT)

The Brute Force approach to solving problems in algorithmic design is the simplest of all algorithm design approaches. It seeks to solve a problem by iterating through all possible combinations of the algorithm. Hence, through this process, the Brute Force Approach arrives at the solution. A formal definition is noted below from, *An Introduction to the Design and Analysis of Algorithms, by Anany Levitin:*

*“Brute force is a straight forward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.”*

Whilst this is a straight forward method to apply, in practice there are a few issues with it. One of the major issues is the efficiency of the algorithm, which is hindered depending on the problem. Because this approach iterates through all combinations of the problem, its efficiency is the product of all possible outcomes of the problem.

This report will analyse the implementation of the ***BruteForceMedian*** algorithm through empirical analysis. A detailed description of the algorithm will be discussed in terms of its time complexity, average, worst and best cases efficiencies. The implementation of the algorithm through pseudocode and c++ will be outline, as well as the design of the experiments regarding the functionality, number of operations and time efficiency. The results from each of the tests will be interpreted, analysed and compared to the theoretical complexity of the algorithm.

**WHAT DID THE RESULTS SHOW – WRITE THIS AFTER THE REPORT**

# Description of algorithm:

The ***BruteForceMedian*** algorithm can be described succinctly as an algorithm which takes half the array as a rounded value *(k)* and compares each element in the array to each other. If the elements are equal to each other, it increments a *numequal* variable by 1. If one of the elements is smaller than the other, it increments a *numsmaller* variable by 1. It repeats these steps until the *numsmaller* is less than *k* and *numequal* and *numsmaller* are less than or equal to *k* then it returns the element at that location in the array.

In more detail, the ***BruteForceMedian*** algorithm takes in an array of elements up to *n-1*, where *n* is the length of the array, and returns the median of the array. If the array size is 0 or null, the algorithm returns an error code, or not median at all. For the length of the array, the algorithm computes half the length of the array within a ceiling function and sets this to variable *k*. It enters a loop whereby for each element in the array up to *n-1*, it proceeds with the following steps. It initialises and sets both *numsmaller* and *numequal* variables to 0. The *numsmaller* variable stores the number of elements which are smaller than a compared element. The *numequal* variables stores the number of elements which are equal to the compared element. After initialization and assignment of these variables, the algorithm enters another loop from 0 to  *n-1*. Within this section, the algorithm compares each element to eachother. If the elements are equal to eachother, then it increments *numequal* by 1, if the inner element is less than the outer element, it increments *numsmaller* by 1. The algorithm returns the median at the index of the outer loop when the *numsmaller* variable is less then *k* and when the sum of *numsmaller* and *numequal* is less than or equal to *k.* The reason why this will be the median of the array is that each element of the array will be checked up to half the length of the whole array.

INSERT THE PSUEDOCODE FOR THE ALGORITHM HERE

# Theoretical analysis of algorithm: DOES NOT SAY SPECIFICALLY IN THE CRA TO DO THE THEORETICAL ANALYSIS SO LEAVE THIS FOR THE TIME BEING

Average Case:

In comparison to other methods of finding the median, this algorithm is not efficient due to checking all elements of the array with eachother. The ***BruteForceMedian*** has a theoretical computational efficiency on the order of

Best Case:

* Best case of the algorithm will be if the median is the first element in the array, therefore the algorithm will only have to check all the elements in the array once (n-1)

Worst Case:

* Worst case is if the median is the end element of the array, therefore it will need to go through all the elements in the array n! (n\*n-1\*n-2\*n-3\*…\*(2n-1))

# Methodology, Tools and Techniques

## Programming environment:

The testing, computation, execution and compilation of the program was completed on a 2018 15” Macbook Pro running macOs Mojave (10.14.2). This computer has a 2.2GHz Intel core i7 (6 core) processor, 16 GB 2400 MHz DDR4 RAM and a Radeon Pro 555X GPU. Compilation was performed using the Clang++17 compiler in accordance to the ISO/IEC 14882:2017 current standard and execution was performed on JetBrains CLion Interactive Development Environment (version 2019.1) using cmake (version 3.11). A complete overview of the environment can be found in appendix xx.

## Programming language implementation:

The decision to implement the algorithm in c++ as opposed to c# or Java came down to speed, current development environment and personal reasons. C++ is quite similar to C as in it is very lightweight and has manual memory management. For analysis of algorithms, it is important that compilation time is low, and that memory management is efficient. In addition to the above, the development environment was within the macOS operating system. C# is targeted towards windows developers, and hence has more support. Further to the above, I believed that this was a good opportunity to learn c++, following on from my experience in c.

## Implementation of program:

The initial requirements stated that the algorithm should be implemented in the most faithful way, and a selection of tests would be made to confirm the functionality, operational and time efficiencies of the algorithm. It was then proposed that the user would be able to control the variables for each of the tests from the command line. In addition to this, the program would be able to produce each of the test input arrays automatically from the user inputs, feed this through the ***BruteForceMedian***algorithm, and then store the input and output arrays as a csv for analysis as well as print the tests to the console.

From start to finish, this is the process the user would take to execute the program.

1. The user will navigate to the headerFile.h to input the variables for each of the tests (the variables are stored as pre-processor directives to limit the variables in the global scope of the program).

|  |  |  |
| --- | --- | --- |
| VARIABLE NAME | DEFINITION | RELATED TEST |
| LARGE\_ARRAY\_VALUE | This is the variable which is used for the large array test. | LARGE ARRAY |
| LARGE\_ARRAY\_SIMS | This is the amount of simulations performed for the large array test (DEFAULT 1). | LARGE ARRAY |
| ARRAY\_STEP\_SIZE | Step size for the random array test. | RANDOM ARRAY |
| ARRAY\_NUM\_SIMS | Number of same length arrays for the random array test. | RANDOM ARRAY |
| RANDOM\_RANGE | Range for the random number generation for the random array test. | RANDOM ARRAY |
| SIMULATIONS | Number of simulations performed by the program. | ALL TESTS |

1. The user will compile and run the program. The program will run and display the current variables it is using for the test.
2. The user will be prompted to enter the test they would like to complete (REFER TO APPENDIX FOR THIS).
3. The user will enter the test they would like to perform, and the program will print each of the tests and the results to the console. In addition to this, the program will save all test input and output data to csv files (in the same location as the executable file).

For a more detailed top down diagram of the program as a whole, functions used within each file, linkages between files, and the commands for compiling and execution of the program, see APPENDIX XX.

The experimental data used in the analysis of the algorithm was obtained by running the program with the input variables (as specified in the method above). Post analysis regarding graphs, averaging results and finding the average, worst and best-case efficiencies was done using Microsoft Excel.

## Implementation of algorithm:

The algorithm was implemented in two separate files to separate the timing tests and the operational tests:

* *“algorithmOps.cpp”* 🡪 File which stores the implementation for the number of operations in the algorithm.
* “*algorithmTime.cpp”* 🡪 File which stores the implementation for the computational time of the algorithm.

Between the two files, the only difference was that the operational implementation included a global variable to track the operational count, whereas the timing file did not include this variable (REFER TO APPENDIX XX FOR BOTH OF THE FILES). For the purpose of being succinct, the operational implementation in the *“algorithmOps.cpp”* file will be discussed.

The algorithm inputted a pointer to a vector of ints and returned the median as a double. If the array has a length of 0, the function returns 0. The basic operations of the algorithm is stored in a global variable named ***numOps*** (refer to section XX for a detailed explanation of the operational test). The computation time of the algorithm was measured by using a high\_resolution\_clock in the std namespace and calling the algorithm implementation in the “*algorithmTime.cpp”* file (refer to section XX for a detailed explanation of the operational test). The data types where decided based on the theoretical size required for each of the variables. As the tests are not exceeding anything above the limitation for ints in c++ (4bytes -2147483647 to 2147483647), ints were used for all of the variables (with the exception to the number of operations). Please refer to the source code below for the *“algorithmOps.cpp”* file.

#include "headerFile.h"  
  
unsigned long long numOp = 0;  
  
double BruteForceMedian(vector<int> &A)  
{  
 long int size = A.size(); *// Calculating the size fo the input array* int k = (int) ceil(A.size() / 2.0); *// Calculate the half-way index of the array.* for (int i = 0; i <= (size-1); i++) {  
 int numSmaller = 0; *// Number of elements smaller than A[i];* int numEqual = 0; *// Number of elements equal to A[i];* for (int j = 0; j <= (size-1); j++) {  
 *// Calculate number of array items that are smaller, and equal.* if (A[j] < A[i]) {  
 numSmaller+=1;  
 numOp+=1;  
 } else if (A[j] == A[i]) {  
 numEqual+=1;  
 numOp+=1;  
 }  
 }  
 if ((numSmaller < k) && (k <= (numSmaller + numEqual))) {  
 return A[i];  
 }  
 }  
 return 0;  
}

# Design of experiments:

The design of the experiments is broken down into three different subsections. Below is a succinct summary of each of the three tests.

|  |  |
| --- | --- |
| Test name | Purpose |
| Functional test | To test the functionality of the program and the algorithm. This will confirm that the program and the algorithm are working as intended. |
| Operational test | Testing the operational efficiency of the algorithm. This test relates to the number of operations performed within the algorithm, and the efficiency class of the algorithm. |
| Timing test | To test the time complexity of the algorithm. This test calculates the execution time of the algorithm for a given array size. |

For each of the above tests, there are subtests which are performed. These subtests have been designed to validate the purpose of the test class.

## Functional test:

To ensure the functional correctness off the algorithm and the program, a number of subtests have been created. The subtests which have been created assess the extreme (boundary) conditions of the algorithm, to ensure functionality. The tests are as follows:

* Negative array elements
* Odd length of array
* Even length of array
* Large length of array
* One length array
* No length array
* Random array size and elements
* Random array size and reversed sorted elements
* Random array size and sorted elements

To ensure that the test has passed, the brute force median result is compared another median finding algorithm (comparison algorithm). The comparison algorithm is stored in the *“comparisonAlgorithm.cpp”* (I NEED TO FIND THE SOURCE FOR THIS). This algorithm takes in a vector of ints and returns the median as a double. The general premise of this comparison algorithm is noted below in the pseudocode.

INPUT THE PSEUDOCODE FOR THIS ALGORITHM

If the median between the ***BruteForceMedian*** implementation algorithm and the ***ComparisonAlgorithm*** are the same, then the test will pass. Throughout all of the tests below, the output data is stored as an integer in a 2-dimensional array of vectors (in c++ vectors are a resizable arrays). The pseudocode for all the algorithms which produce the input data can be found in the APPENDIX.

Negative elements

INSERT PSUEDOCODE

In general, the negative arrays are produced as a set of 2 dimensional vectors, whereby the inner vector is an array of negative elements from 0 to the simulation size, and the outer vector is the collection of negative arrays. The size of the negative array is dependent on the number of simulations which the user inputs.

Odd length

The odd length arrays are generated by iterating through 0 to the amount of simulations, as specified by the user, in odd steps. This is stored in an inner vector, which is then stored in an outer vector (the outer vector being the collection of inner vectors).

Even length

The even length arrays are generated in the same way as the odd length, however the step size for this array collection is even instead of odd.

Large array length

The large arrays are generated based on the input from the user (as in the user can specify the length of the large array). In addition to this, the user can specify the number of simulations to be run on that array length that they specified.

One length and No Length

These test arrays are produced by creating an array on length one and no length for the number of simulations as specified by the user. The one length array is always filled with a value of 1, whereas the no length array has no elements.

## Operational testing and timing testing (methodology):

For each of the operational and timing tests, there are a number of the same subtests performed. These subtests are to assess the performance of the algorithm, rather than the functionality. In this sense, the tests are based on the elements of the array, rather than the boundary conditions of the input to the algorithm. The subtests are as follows:

* Random array size and elements
* Random array size and reversed sorted elements
* Random array size and sorted elements

The process for obtaining the input data for each of the tests above is somewhat similar to the functional test. The data is stored as an integer, within a 2-dimensional resizable array (in c++ this is a 2-dimensional vector). The user can specify the step size between each random size array, the number of simulations performed for the array with the same length, the range of the random variable generated and the number of simulations. The random value is generated using a seed based on the current clock time of the computer (found in *“generateData.cpp”*). The range is specified by taking the modulo of the random number generated by the maximum number in the range (defined by RANDOM\_RANGE in the *“headerFile.h”*). The program creates multiple of the same length array (as defined by ARRAY\_NUM\_SIMS) to find the average, best and worst-case efficiencies for a certain array length. In addition to this, the length of the array is stepped based on the ARRAY\_STEP\_SIZE variable. The amount of simulations (defined by SIMULATIONS) is the amount of the same length array. For example, if the user selected the following variables, ARRAY\_NUM\_SIMS = 10, ARRAY\_STEP\_SIZE = 100 and SIMULATIONS = 10, then they would get an input 2-dimensional array of 10 sets of 10 arrays with the same length, with their length stepped by 100 up to a length of 1000 (refer to appendix for a diagram or some math formula).

The best, worst and average case efficiencies for both the operational testing and the timing testing was considered. The methodology to obtain these results consisted of taking the average across the number of the same simulation sized arrays, the maximum value and the minimum value. This was then plotted against the size of the array for the given execution time and operation count.

## Operational testing (numOps location):

The main point of interest for the operational testing is where the number of operations are counted for in the algorithm. As defined in the implementation of the algorithm section, the number of operations is stored in a global variable of type *unsigned long long* in the file *“algorithmOps.cpp”*. This variable is then incremented when the algorithm performs an “operation”. Operation is defined as a comparison between two elements within the input array, or an increment of a variable in the local scope of the function. This occurs when the *numSmaller* and *numEqual* variables are incremented. In addition, this is when a comparison is made between two elements in the input array. Please refer to the code below for the implementation of the number of operations. DO I INCLUDE THE BEST AND WORST CASES FOR THESE OPERATIONS?

case OPERATIONS:  
 for (auto &col : inputVector) {  
 algoMedianVector.push\_back(BruteForceMedian(col));  
 numOpsVector.push\_back(numOp);  
 }  
 *// Comparing the results and printing to the console* cout << "========OPERATION TESTING RESULTS========" << endl;  
 cout << "=====THIS IS THE TEST (" << test << ")=====" << endl;  
 printConsoleOperations(numOpsVector, algoMedianVector, arrayLength);  
 *// Saving the results to a csv file* createOutputCsvOps(arrayLength, algoMedianVector, numOpsVector, test);  
 break;

if (A[j] < A[i]) {  
 numSmaller+=1;  
 numOp+=1;  
} else if (A[j] == A[i]) {  
 numEqual+=1;  
 numOp+=1;  
}

Comparison section of the BruteForceMedian function

## Timing testing (time measurement location):

The time was kept separate from the operational testing implementation. As a result, the time was measured using the BruteForceMedianTime function in the *“algorithmTime.cpp”* file. The time was measured before and after this function was called in the *“runTests.cpp”* file. The c++17 high\_resolution\_clock was used to measure the time, in nanoseconds, points before and after the function call. The difference was taken between these two points, and then added to a vector, execTimeVector which stores the execution time for each test, for later use in the program.

case TIMING:  
 for (auto &col : inputVector)  
 {  
  
 high\_resolution\_clock::time\_point t1 = high\_resolution\_clock::now();  
 int output = BruteForceMedianTime(col);  
 high\_resolution\_clock::time\_point t2 = high\_resolution\_clock::now();  
 auto nanoseconds = std::chrono::duration\_cast<std::chrono::nanoseconds>(t2-t1);  
 int compareOutput = testMedian(col);  
 cout << "Compared output: " << compareOutput << endl;  
 execTimeVector.push\_back(nanoseconds.count());  
 algoMedianVector.push\_back(output);  
 }  
 cout << "========TIMING TESTING RESULTS========" << endl;  
 cout << "=====THIS IS THE TEST (" << test << ")=====" << endl;  
*// printConsoleTiming(execTimeVector,algoMedianVector,arrayLength);  
 // Saving the results to a csv file* createOutputCsvTiming(arrayLength, algoMedianVector, execTimeVector, test);  
 break;

Experimental results (4 pages)

* Results for each of the test below, with some interpretation of the results

Functional testing:

* Negative elements
* Odd length
* Even length
* Large array length
* One length
* No Length

Operational testing:

* Random
* Reversed
* Sorted

Timing testing:

* Random
* Reversed
* Sorted

Experimental analysis (4 pages):

* Analysis for each of the tests, including the following in each section (only really need to focus the bulk of the analysis on the operational testing and the timing testing – focus on average, worst and best cases efficiencies)
  + Theoretical vs experimental results
  + Was this what we were expecting? Why were we expecting this?
  + Max and Min cases for each of the experiments
  + Linking back to the underlying functionality of the algorithm
  + Error analysis/anomalies in the data

Functional testing:

* Negative elements
* Odd length
* Even length
* Large array length
* One length
* No Length

Operational testing:

* Random
* Reversed
* Sorted

Timing testing:

* Random
* Reversed
* Sorted

Operational testing:

* The sorted arrays seem to have less operations then the unsorted array, and even less operations then the random array
* This means that the sorting effects the operational efficiency depending on the array length

Timing testing:



Recommendations and conclusions

* Program recommendations 🡪 Do not use 2D vectors for time efficiency
* Program recommendations 🡪 Complete the variable input using compile flags (indef directives), to build on this, create input flags for the program (inputted to main)
* Median sorting algorithm 🡪 Do not use the bruteforcemedian algorithm for finding the median value of an array, rather use the quicksort method (back this up by some research)