CAB301 Assignment 2

Empirical Comparison of Two Algorithms:

*BruteForceMedian* and *Median*

James Flannery – n8326631

Yee Chen Fong – n9602976

Date submitted: 14th May 2016

# **Summary**

This report is a summary of the empirical comparison of the two algorithms which are *BruteForceMedian* and *Median*, that finds the median value of an array. The report includes the description of both algorithms, the choice of basic operation and problem size of both algorithms, and the theoretical time efficiency of both algorithms.

The two algorithms are functionally tested and implemented using a C++ Console Application, with both the number of basic operations performed and the total execution time of the algorithms being measured and recorded. The results are produced by experimenting both algorithms with identical data and the experiment is repeated several times for each size of array. The results from the experiment conducted confirms that the observed numbers of basic operations were consistent with the theoretical predictions for the algorithms. The results show that the *Median* algorithm has an average-case efficiency class of , which **performs better** than the *BruteForceMedian* algorithm that has an average-case efficiency class of when finding the median value of an array.

1. **Description of the Algorithm**

Assume *A[]* is an array. The first algorithm (Figure 1), *BruteForceMedian* is an algorithm that begins by first finding the middle index of the array known as the **k**th index, where   
**k** = ⌈ Size of array / 2 ⌉, with the celling brackets round up to the nearest integer (Figure 1, **condition b**). The algorithm then goes into a nested loop where the outer loop loops from the first element of the array to the last element of the array (Figure 1, **condition c**), setting a counter called *numsmaller* that counts the number of elements that are smaller than the current outer looping element, *A[i]* and another counter called *numequal* that counts the number of elements that are equal to the current outer looping element, *A[i]*, both to 0 (Figure 1, **condition d and e**).

The inner loop then loops from the first element of the array to the last element of the array again (Figure 1, **condition f**), this time to either increment the *numsmaller* counter if the currently inner looping element, *A[j]* is smaller than current looping outer element, *A[i]* (Figure 1, **condition g and h**) or increment the *numequal* counter if the currently inner looping element, *A[j]*is equals to the current looping outer element *A[i]* (Figure 1, **condition j and k**). This is followed by an if else condition which returns the **median** value of the array, *A[i]* if *numsmaller* is lesser than **k** and if **k** is lesser or equals to the total of *numsmaller* and *numequal* (Figure 1, **condition l**).

The second algorithm (Figure 2a), *Median* is one of the 3-part selection problem algorithm procedure that is used to find the median of an array. *Median* begins by first checking if the given array contains only one item and returns the value, *A[0]* as the **median** value of the array (Figure 2a, **condition b and c**). Otherwise, the algorithm will initialize the recursive procedure called *Select* (Figure 2b), while passing the array *A[]*, the first index of the array, the middle index of the array, and the last index of the array as arguments (Figure 2a, **condition d and e**).

The *Select* algorithm procedure (Figure 2b), is the second component of the 3-part selection problem algorithm procedure that is used to find the median of an array. *Select* begins by initiating the *Partition* procedure (Figure 2c) while passing the array A[], lowest index of array slice, highest index of array slice as arguments, which will return the pivot location index of the array (Figure 2b, **condition b**). The pivot location index, *pos* is compared with the middle index of the array, *m*. If *pos* is more than *m*, the *Select* algorithm recurs, this time passing the array *A[]*, the lowest index of array slice *l*, the middle index of the array *m*, and the highest index of array slice excluding the pivot index *pos-1* (Figure 2b, **condition e and f**). If *pos* is less than *m*, the *Select* algorithm recurs, passing the array *A[]*, the lowest index of array slice excluding the pivot index *pos +1*, the middle index of the array *m*, and the highest index of array slice *h* (Figure 2b, **condition g and h**). If *pos* is equals to *m*, the value of *pos*, *A[pos]* is returned as the **median** of the array, ending the recurs (Figure 2b, **condition c and d**). The array slice decreases as each recursive is called.

The *Partition* algorithm procedure (Figure 2c), is the last component of the 3-part selection problem algorithm procedure that is used to find the median of an array. *Partition* firstly takes the first value of the array slice to be set as the pivot value, *pivotval* (Figure 2c, **condition b**). The first index of the array slice is then used to set *pivotloc*, which is the location to insert pivot value (Figure 2c, **condition c**). The algorithm then proceeds into a loop that loops from the second index of the array slice to the last index of the array (Figure 2c, **condition d**). Inside this loop, an if statement checks if the currently looping element, *A[j]* is less than *pivotval*. If the condition is fulfilled, the *pivotloc* is increment by one, and the element of pivotloc, *A[pivotloc]* is swapped with the currently looping element, *A[j]* (Figure 2c, **condition e to g**). Once the looping is done, the first element of the array slice, *A[l]*, swap places with *A[pivotloc]* to put the pivot element in place. This result all the elements before the *pivotloc* to be smaller than the *pivotloc* element, likewise all elements after the *pivotloc* to be bigger than the *pivotloc* element (Figure 2c, **condition h**). Finally, the value of *pivotloc* is returned (Figure 2c, **condition i**).

# **Theoretical Analysis of the Algorithm**

This section describes the algorithm’s predicted (theoretical) average-case efficiency from a theoretical perspective.

## **Choice Basic Operation**

The Basic Operation of the algorithm is identified by observing the part of the algorithm that has the largest effect on the execution time of the program [1, pp. 44].

According to Berman and Paul, comparison of elements is the basic operation for sorting a list with *n* elements [2, Figure 2.1, pp. 30]. In the *BruteForceMedian* algorithm in Figure 1, a basic operation can be identified at the if and else if statements comparisonat **condition g, I and j**, because for any input list of size *n*, the inner loop is executed *n* - 1 times (**condition f**) and the outer loop is executed *n* -1 times (**condition c**).

Any other operations of the *BruteForceMedian* algorithm such as *k* ← ⌈*n/*2⌉ at **condition b**, *numsmaller* ← 0 at **condition d**, *numequal* ← 0 at **condition e,** *numsmaller* ← *numsmaller* + 1 at **condition h**, *numequal* ← *numequal* + 1 at **condition k**, if statement at **condition l and m** can be ignored as they are not comparison of elements, thus are insignificant in the complexity analysis.

In the *Select* procedure of *Median* algorithm in Figure 2c, a basic operation can be identified at the whole *Partition* procedure at **condition b**. The reason *Partition* procedure is identified as a basic operation is because it has array comparisons [2, pp. 247]. At the beginning of the operation, *Partition* will always take *n*-1 times and the size of the array input to further recursive call of *Select* will be less than is *n*-1.

In the real experiment, the basic operation counting increment will be placed in between **condition d** **and e** in the *Partition* procedure which is equivalent to the whole Partition execution times.

Any other operations of the *Partition* procedure such as *pivotval* ← *A*[*l*] at **condition b**, *pivotloc* ←*l* at **condition c**, *pivotloc* ← *pivotloc* + 1 at **condition f**, swap(*A*[*pivotloc*], *A*[*j*]) at **condition g** and swap(*A*[*l*], *A*[*pivotloc*]) at **condition h** can be ignored as they are not comparison of elements, thus are insignificant in the complexity analysis.

## **Choice of Problem Size**

The ‘problem size’ for both algorithms is the number of elements in the given array pass into the *BruteForceMedian* and *Median* algorithm.

## **Average-Case Efficiency**

Both algorithms have different efficiency when finding the median value of an array. In this report, only the average-case efficiency of the algorithms is concerned.

The *BruteForceMedian* algorithm basically starts by setting a variable *k* = size of array/2, which is the position of the median when the array is sorted. The algorithm has an outer loop which iterates through the array of n elements from the element (0) to the last element (n-1) (Figure 1, **condition c**) and an inner loop which iterates through the array from element (0) to the element of (n-1) (Figure 1, **condition f**). Once the inner loop completes, an if statement is used to check if the median value of the array has been found by checking if the number of elements smaller than the element currently looping in the outer loop is less than *k* and if *k* is smaller or equals to the total of number of elements smaller than the element currently looping in the outer loop and the number of elements equal to the element currently looping in the outer loop (Figure 1, **condition l**).

Since *BruteForceMedian* is a kind of algorithm that can ‘exit early’ when the median value of the array has been found. In an average-case situation, the median value of the array is assumed to be placed at the middle index of the array, which is *n*/2. Thus, the algorithms will exit when the outer loop reaches n/2.

The *BruteForceMedian* algorithm has a **theoretical** average-case efficiency of:

*BruteForceMedian* Inner loop:

*BruteForceMedian* Outer loop:

**Theoretical** *BruteForceMedian* Algorithm’s average case efficiency:

Meanwhile, the key to find the *Median* algorithm average-case efficiency, is the number of comparisons of array elements in the loop in *Partition* procedure (Figure 2c, **condition e**). According to Johnsonbaugh and Schaefer, the number of comparisons of the array elements in *Partition*, for an array slice of size n is n – 1 [3, pp. 246]. This can be found at (Figure 2c, **condition d**), where the *Partition* procedure has a for loop which loops from *l* + 1 to *h*. E.g. When an array slice of *A*[3..9] is pass into the *Partition* procedure, it will result *l* = 3 and *h* = 9, size of array slice = 7, and number of comparison of elements= 6.

Once the *Partition* procedure is done, the algorithm returns to the *Select* procedure with the value of *pos*, where *pos* is compared with the middle index of the array, *m*. Now in the average-case scenario, it is assumed that the *Partition* procedure always divides the array to be sorted into two nearly equal array slices, and the *pos* value returned by partition is not equal to *m* (Figure 2b, **condition b**). Thus, we can conclude that size of array to be processed into the *Partition* procedure will be reduce by half in further recurs, when the input is an array of *n* size.

The *Median* algorithm has a **theoretical** average-case efficiency of:

Loopingin *Partition* procedure in *Median* algorithm:

Recursive *Select* procedure formula:

*Median* algorithm recurrence relation, used wolframalpha.com to calculate summation [8]:

Let *h* = *n* - 1, *l* = 0

, since algorithm does basic operation zero times if *n* = 1

Assume *n* = , because

Since and , let *i* equal *k*

Using the property of logs where =, to prove the statement

Let y = ,

Therefore, as the array size gets bigger, the number of comparisons is at most 4 [2, pp. 250] [3, pp.266] [7].

,

**Theoretical** *Median* Algorithm’s average case efficiency:

This can be further confirm by Levitin, who stated that the average-case efficiency of *Median* algorithms would be the same even as the array size reduce by half in size, and the average-case result in linear efficiency of [1, pp. 189].

## **Order of Growth**

Based on the average case efficiency equation in Section 2.3 above, the finding of the median of numbers in an array using *BruteForceMedian* algorithm will result in an efficiency of , a ‘quadratic growth’. With the *Median* algorithm, the finding of the median of numbers in an array will have an efficiency of , a ‘linear growth’.

# **Methodology, Tools and Techniques**

This section is a summary about the choice of computing environment and methods used to produce test data for the experiment.

## **Programming Environment**

1. C++ programming language is used to implement both the algorithm and the test experiment as it is the assignment specification.

2. The experiments were performed on a PC running Windows 10, equipped with an Intel i5-2500K CPU running at 3.3Ghz. The C++ system libraries used the system time to provide a seed for the pseudo-random numbers used for array population. The use of unimportant applications was minimized while taking time measurements, to ensure consistent results.

3. The graphs provided in Figures 3 and 4 were generated in Excel. The average measurements, along with array size, were written to a text file, which was imported using native Excel functionality. These graphs were then inserted into the word file containing the rest of this report, which was then converted to PDF for submission.

## **Implementation of Algorithms**

The median-finding algorithms rely on the existence of a data structure of array. In our C++ implementation, this data structure was implemented using the C++ 11 vector type.

The algorithms were implemented as methods. The two algorithms in Figures 1 to 2 were implemented in C++ as shown in Appendix A and B, respectively.

## **Generating Test Data and Running the Experiments**

Both algorithms are written separately as a method to make sure that the algorithm is written as similar as possible to the one given in the assignment specification. Moreover, it makes the recording of the algorithm’s execution time more accurate.

Tests were taken at regular intervals of size. In this case, from size 10 to size 1000. The number of basic operations and execution time is recorded for each.

1. **Experimental Results**

This section describes the outcomes of the experiments and compares the results with the theoretical predictions from Section 2. The programming language implementation of the algorithm from Figure 1 and Figure 2 are shown in Appendix A.

## **Functional Testing**

To ensure the C++ implementation of the two algorithms was correct, they were run through the functional testing described in Appendix C. This test program runs each algorithm through a number of randomly sized and randomly populated algorithms, and compares the result with the result of another function, which uses the C++ standard library to find the median. There are also several tests run to include edge cases. For example, one function tests a sorted vector, another a vector with each element the same, rather than being randomly populated.

The program uses exceptions to catch any mismatches between the results. If no exceptions are found, the program prints a success message.

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

In order to count the average number of basic operations, the test took measurements for running the algorithms on arrays of increasing sizes. The code iterated from 0 through to a maximum size, with a test run every time a given interval was reached. For each test, a certain number of randomly populated arrays would be created. The number of operations in each would be aggregated and then divided by the number of arrays, to provide the average number of operations at the given size.

For the graphs shown in Figure 3a and 3b, tests were taken every 10 steps, up to 1000 element arrays. Each test created 100 randomly populated arrays for measurement.

## **Average-Case Execution Time**

Similar to the basic operations, the average case execution time takes tests at a given interval of size, up to a given maximum. These are aggregated and averaged in the same manner. The parameters used for these measurements are the same as in section 4.2.

As these are measurements of time, a way to take measurements was necessary. The standard C++ library provides the clock function, which was used for this. Before each individual measurement, the clock time would be recorded. Once complete, the end time would also be recorded. The difference between these times provides the execution time for each individual execution of the algorithm.

# **Reference**

[1] A. Levitin, *Introduction to the design & analysis of algorithms*. Boston: Pearson Addison-Wesley, 2007. ISBN: 0-321-36413-9

[2] K. Berman and J. Paul, *Algorithms*. Boston, Mass.: Thomson/Course Technology, 2005. ISBN: 0-534-42057-5

[3] R. Johnsonbaugh and M. Schaefer, *Algorithms*. Upper Saddle River, NJ: Pearson Education, 2003. ISBN: 0-023-60692-4

[4] "Input/output with files - C++ Tutorials", Cplusplus.com, 2016. [Online]. Available: http://www.cplusplus.com/doc/tutorial/files/. [Accessed: 24- Mar- 2016].

[5] "srand - C++ Reference", *Cplusplus.com*, 2016. [Online]. Available: http://www.cplusplus.com/reference/cstdlib/srand/. [Accessed: 24- Mar- 2016].

[6] "clock - C++ Reference", Cplusplus.com, 2016. [Online]. Available: http://www.cplusplus.com/reference/ctime/clock/. [Accessed: 25- Mar- 2016].

[7] D. Eppstein, "Blum-style analysis of Quickselect", *11011110.livejournal.com*, 2007. [Online]. Available: http://11011110.livejournal.com/119720.html. [Accessed: 16- May- 2016].

[8] "Wolfram|Alpha Widget: Math Help Boards: Sum Calculator", *Wolframalpha.com*, 2016. [Online]. Available: http://www.wolframalpha.com/widget/widgetPopup.jsp?p=v&id=dfaf1b7d15e572ae5a1b2fa172ce8657&title=Math+Help+Boards%3A+Sum+Calculator&theme=blue. [Accessed: 16- May- 2016].

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

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

// the *k*th 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:** *BruteForceMedian*algorithm. Assume the array is sorted from smallest to largest. Let *k*th element be the kth smallest element of the array.

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

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

1. **if** *n* = 1 **then**
2. **return** *A*[0]
3. **else**
4. *Select*(*A*, 0, ⌊*n/*2⌋, *n* − 1) // NB: The third argument is rounded down

## **Figure 2a:** *Median*algorithm. Assume that the array *A* is unsorted. Let 0 be the first index of the array. Let ⌊*n/*2⌋ be the middle index of the array. Let *n −* 1 be the last index of the array.

1. **ALGORITHM** *Select*(*A*[0..*n* − 1], *l*, *m*, *h*)

// Returns the value at index *m* in array slice *A*[*l*..*h*], if the slice

// were sorted into nondecreasing order.

1. *pos* ← *Partition*(*A*, *l*, *h*)
2. **if** *pos* = *m* **then**
3. **return** *A*[*pos*]
4. **if** *pos* > *m* **then**
5. **return** *Select*(*A*, *l*, *m*, *pos* − 1)
6. **if** *pos* < *m* **then**
7. **return** *Select*(*A*, *pos* + 1, *m*, *h*)

## **Figure 2b:** *Select* algorithm, part of *Median*algorithm. Assume that the array *A* is unsorted. Let *pos* be the variable of the pivot location of the array. Let *l* be the first index of the array. Let *m* be the middle index of the array. Let *h* be the last index of the array.

1. **ALGORITHM** *Partition*(*A*[0..*n* − 1], *l*, *h*)

// Partitions array slice *A*[*l*..*h*] by moving element *A*[*l*] to the position

// it would have if the array slice was sorted, and by moving all

// values in the slice smaller than *A*[*l*] to earlier positions, and all values

// larger than or equal to *A*[*l*] to later positions. Returns the index at which

// the ‘pivot’ element formerly at location *A*[*l*] is placed.

1. *pivotval* ← *A*[*l*] // Choose first value in slice as pivot value
2. *pivotloc* ←*l* // Location to insert pivot value
3. **for** *j* **in** *l* + 1 **to** *h* **do**
4. **if** *A*[*j*] < *pivotval* **then**
5. *pivotloc* ← *pivotloc* + 1
6. swap(*A*[*pivotloc*], *A*[*j*]) // Swap elements around pivot
7. swap(*A*[*l*], *A*[*pivotloc*]) // Put pivot element in place
8. **return** *pivotloc*

## **Figure 2c:** *Partition*algorithm, part of *Median*algorithm. Assume that the array *A* is unsorted. Let *l* be the first index of the array. Let *h* be the last index of the array.

## **Figure 3a:** Graph of the average number of basic operations performed by both algorithms. The graph shows 100 points of data, each representing the average measured at that particular size. It is clear that the brute-force algorithm increases exponentially.

## **Figure 3b:** Graph of the average number of basic operations performed by the select median algorithm. Contains the same data as Figure 3a, but in closer detail, so as to make the trend of the select algorithm clearer. With this, it is easy to see that the number of basic operations increase linearly, as is theorized in Section 2.3.

## **Figure 4a:** Graph of the average execution time of both algorithms. As with basic operations, this graph plots 100 points of data. In this case, it measures the average time taken to find the median of arrays with sizes from 10 to 1000.

## **Figure 4b:** Graph of the average execution time measured for the select algorithm. Provided for the same reasons as Figure 3b, and with the same trend observable.

# **Appendix A: Implementation of the Brute Force Median algorithm**

Shown here is the algorithm presented in pseudocode in Figure 1. This C++ implementation was used to carry out the experiments. This function uses the brute force algorithm to find and return the median of a given array, A. For ease of testing, the array is implemented here as a vector. This is functionally identical, for our purposes. This vector is passed by reference to the function.

The code itself begins by initializing int k, int numsmaller, and int numequals. This is just an allocation of memory, as these variables are initialized later in the function. Indeed, following these initializations k is set to half the size of the vector.

Next, for (int i = 0; i <= A.size() - 1; i++) is used as the C++ equivalent to the pseudocode ‘for i ← 1 to n – 1 do’. This is functionally the same, just coded in C++.

unsigned long long int BruteForceMedian(vector<int> &A)

{

int k;

int numsmaller;

int numequals;

k = ceil(A.size() / 2);

//Loop through the array

for (int i = 0; i <= A.size() - 1; i++)

{

numsmaller = 0;

numequals = 0;

//Inner Loop through the array

for (int j = 0; j <= A.size() - 1; j++)

{

if (A[j] < A[i])

{

numsmaller++;

}

else if (A[j] == A[i])

{

numequals++;

}

}

if (numsmaller < k && k <= (numsmaller + numequals))

{

return A[i];

}

}

}

# **Appendix B: Implementation of the Select Median algorithm**

As in Appendix A, the code shown here is the C++ implementation of a pseudocode algorithm provided earlier. In this case, it’s the *Median* algorithm, as shown in Figure 2. The three parts of this algorithm are coded as three separate functions, which call and pass values to each other as dictated by the pseudocode. Just as in the brute force algorithm, the arrays here are implemented as C++ vectors.

The first function, Median, is passed a reference to the array for which the median is to be found. If the array is having more than one element, it calls the Select function.

unsigned long long Median(vector<int> &A)

{

if (A.size() == 1)

return A[0];

else

return Select(A, 0, floor(A.size() / 2), A.size() - 1); // NB: The third argument is rounded down

}

The Select function calls on the Partition function, as well as recursively calling itself until the median value is found. It takes the same reference that is passed to the Median function, and uses integer variables to handle the values l, m, h.

int Select(vector<int> &A, int l, int m, int h)

{

// Returns the value at index m in array slice A[l..h], if the slice were sorted into nondecreasing order.

int pos = Partition(A, l, h);

if (pos == m)return A[pos];

if (pos > m)return Select(A, l, m, pos - 1);

if (pos < m)return Select(A, pos + 1, m, h);

}

Partition is implemented much the same as the other functions, using a reference to the initial vector, and integers to hold the other values needed. Uniquely, though, it makes use of the swap function to exchange the position of to values in the array. Swap is included in the standard C++ libraries. Because this function is used on the vector that was passed as a reference, this algorithm will shift the elements from their original order.

int Partition(vector<int> &A, int l, int h)

{

// Partitions array slice A[l..h] by moving element A[l] to the position

// it would have if the array slice was sorted, and by moving all

// values in the slice smaller than A[l] to earlier positions, and all values

// larger than or equal to A[l] to later positions. Returns the index at which

// the ｡ｮpivot｡ｯ element formerly at location A[l] is placed.

int pivotval = A[l]; // Choose first value in slice as pivot value

int pivotloc = l; // Location to insert pivot value

for (int j = l + 1; j <= h; j++)

{

if (A[j] < pivotval)

{

pivotloc++; //Increment pivot location

swap(A[pivotloc], A[j]); // Swap elements around pivot

}

}

swap(A[l], A[pivotloc]); // Put pivot element in place

return pivotloc;

}

# **Appendix C: Code for functional testing**

This appendix shows the code used to test the functional correctness of the two algorithm’s C++ implementations.

int MAX\_ARRAY\_SIZE = 100;

int VARIANCE = 1000;

int NUM\_TESTS = 100;

Firstly, several global variables are initialized. These control the number and maximum size of the arrays to be tested, as well as the spread of elements that are randomly generated for each array.

int main()

{

srand(time(NULL));

for (int i = 0; i < NUM\_TESTS; i++) {

TestRandom();

TestSorted();

TestReverseSorted();

TestSameNumber();

cout << endl;

}

cout << "ALL TESTS PASSED!" << endl;

system("PAUSE");

return 0;

}

The main() function begins by setting the seed for C++’s pseudo-random number generator. Then it calls each testing method in a for loop, resulting in every test being run the number of times specified by the global variable NUM\_TESTS. Finally, once all tests are complete, it outputs a successful message. Each individual test asserts a successful result, so any failed tests will error and the end of the main() function will not be reached.

The four test methods are very similar in implementation. Each creates an vector and adds elements until a size (determined randomly) is reached. This vector is run through the two algorithms, and these results are compared to results found using the C++’s existing vector methods. The difference between the four tests is just the type of vector created. The first test is presented in full, but for the sake of brevity the other three are cut down to only the differences.

void TestSameNumber()

{

cout << "Testing same number array." << endl;

int resultBruteForceMedian, resultMedian;

vector<int> A;

int x = rand() % 100;

//Insert random numbers into the array

for (int j = 0; j < 2 + rand() % MAX\_ARRAY\_SIZE; j++)

{

A.push\_back(x);

}

resultBruteForceMedian = BruteForceMedian(A);

resultMedian = Median(A);

assert(TestBruteMedian(A, resultBruteForceMedian));

assert(TestSelectMedian(A, resultMedian));

cout << "Same number array test passed." << endl;

}

The push\_back(x) method adds an element to the given vector. In this case, x is randomly determined before the insertion loop, so every element will have the same value.

The assert method takes a boolean and throws an error if it is false. Both TestBruteMedian and TestSelectMedian return boolean values, and are shown further down in this appendix.

void TestReverseSorted()

{

//Insert random numbers into the array

for (int j = 0; j < 2 + rand() % MAX\_ARRAY\_SIZE; j++)

{

A.push\_back(rand() % VARIANCE);

}

sort(A.rbegin(), A.rend());

}

The line sort(A.rbegin(), A.rend()) sorts the vector into reverse order.

void TestSorted()

{

//Insert random numbers into the array

for (int j = 0; j < 2 + rand() % MAX\_ARRAY\_SIZE; j++)

{

A.push\_back(rand() % VARIANCE);

}

sort(A.begin(), A.end());

}

Likewise, sort(A.begin(), A.end()) sorts the vector normally.

void TestRandom()

{

//Insert random numbers into the array

for (int j = 0; j < 2 + rand() % MAX\_ARRAY\_SIZE; j++)

{

A.push\_back(rand() % VARIANCE);

}

}

bool TestBruteMedian(vector<int> &A, int m)

{

sort(A.begin(), A.end());

return (A[(ceil(A.size() / 2)-1)] == m);

}

bool TestSelectMedian(vector<int> &A, int m)

{

sort(A.begin(), A.end());

return (A[(ceil(A.size() / 2))] == m);

}

These two methods are used to verify that the median found by the output is correct. They take a reference to the vector being used, and the median that the respective algorithm found. To find the correct median, the array is first sorted. Because of this sorting, the middle element of the array will be the true median. As such, the element at (A.size() / 2) is compared to the algorithmically found median, and the result returned.

The reason there are two test functions instead of just the one is that the two different algorithms return different result in the case of an evenly-sized array. As there is no true ‘middle’ element, the first returns the value to the left of the midpoint, whilst the second algorithm returns the right. It is for this reason that the ceil function is used. This rounds the division of the vector size up, giving the element to the right. In the case of the brute-force test, this number is reduced by one, giving the element to the left of the midpoint.

# **Appendix D: Code to count the average-case number of operations**

The code presented in the previous appendices needed to be modified in order to measure the average number of basic operations. Each algorithm needs, naturally, to keep a count of its own operations as they are performed. To find the average case, this count needs to be measured over several arrays of the same size, and that result averaged.

New and modified code is underlined, and explained where necessary.

int TEST\_STEP = 100;

int SET\_SIZE = 10;

int VARIANCE = 1000;

int NUM\_TESTS = 1000;

These new variables control the total the total size of the tests, and the interval for tests to be run. Unlike the functional testing, these measurements are not taken every loop, but only at every TEST\_STEP instead.

unsigned long long int basicCount;

Because the select media algorithm is implemented over multiple methods, the tracker for its basic operations is declared here.

int main()

{

srand(time(NULL));

ofstream out\_data;

out\_data.open("../../outAvgBasics.txt");

for (int i = 1; i <= NUM\_TESTS; i++)

{

if (i % TEST\_STEP == 0)

{

vector<vector<int>> tests = CreateArrays(i);

int opCount = 0;

for (int a = 0; a < SET\_SIZE; a++)

{

opCount = BruteForceMedian(tests[a], i) + opCount;

}

cout << i << "\n";

out\_data << i << "\t " << opCount / SET\_SIZE << "\t\t";

opCount = 0;

for (int a = 0; a < SET\_SIZE; a++)

{

opCount = Median(tests[a], i) + opCount;

}

cout << i << "\n";

out\_data << i << "\t " << opCount / SET\_SIZE << "\n";

}

}

return 0;

}

The line ‘if (i % TEST\_STEP == 0)’ controls the tests to only be run when the array size matches the interval set earlier.

When a test is run, the operation counter variables are declared as ints and initialized to zero. These aggregate the number of operations for all tests in a set, and once the set has run, is divided by SET\_SIZE to find the average.

vector<vector<int>> CreateArrays(int size)

{

vector<vector<int>> tests;

for (int i = 0; i < SET\_SIZE; i++)

{

std::vector<int> testSet;

for (int x = 0; x < size; x++)

{

testSet.push\_back(rand() % VARIANCE);

}

tests.push\_back(testSet);

}

return tests;

}

To approximate the average case, we need to make measurements over multiple tests of the same size. To do this we create an array of arrays (in this case, a vector of vectors). The CreateArrays function takes a size and returns a vector tests, which itself contains SET\_SIZE number of vectors, each populated with a number of random elements of the given size.

# **Appendix E: Code to measure the average-case execution times**

The measurement of the average execution times has a very similiar structure to the measurement of average basic operations. As such, this appendix will only discuss the differences between the two. Apart from the removal of the counting variables in the algorithm methods themselves, these differences are all found in the main() function.

clock\_t is part of the C++ standard clock library, and is used to keep track of the system time. In the inner loop, where the tests themselves are called, it is used alongside two doubles, which are initialised themselves. Before each test begins, the start time is recorded. The difference between this and the end time of the test is added to the avgTime double, which, once the whole set of tests is complete, is divided by the set size to find the average.

int main()

{

srand(time(NULL));

clock\_t startTime;

ofstream out\_data;

out\_data.open("../../outAvgTimes.txt");

for (int i = 1; i <= NUM\_TESTS; i++)

{

if (i % TEST\_STEP == 0)

{

vector<vector<int>> tests = CreateArrays(i);

double endTime = 0;

double avgTime = 0;

for (int a = 0; a < SET\_SIZE; a++)

{

startTime = clock();

BruteForceMedian(tests[a], i);

endTime = clock();

avgTime += endTime - startTime;

}

cout << i << "\n";

out\_data << i << "\t " << avgTime / SET\_SIZE << "\t\t";

avgTime = 0;

for (int a = 0; a < SET\_SIZE; a++)

{

startTime = clock();

Median(tests[a], i);

endTime = clock();

avgTime += endTime - startTime;

}

cout << i << "\n";

out\_data << i << "\t " << avgTime / SET\_SIZE << "\n";

}

}

return 0;

}