CAB301 Assignment 2

Empirical Analysis of For Algorithms for Testing Set Membership.

Student Names: John Layson, Nathan Thai

Student No. n9823239 – JL

Student No. n9823361 – NT

Date Submitted: 21/05/2018

Summary

This report analyses 2 algorithms that find the medians in an array, including the determination of their average case efficiency and order of growth. Tests were performed with different size of inputs that counts the basic operation in an iteration and execution time for one algorithm call. These tests were also performed and implemented in the Java programming language.

1 Description of the Algorithms

Brute Force Median

The purpose of the Brute Force Median algorithm is to return the middle (median) element in an array. The algorithm starts off by assigning the length of the array to the *n* variable which is then divided by 2 and assigned in \to the *k* variable which is then rounded up. This mathematic portion of the algorithm determines the position of the median in an array. The array's outer for-loop(i) starts at the first element (index 0) which is then compared to the rest of the array by the inner for-loop(j), this is repeated for each element in the array thus the keyword in its name "Brute Force". There is an if-statement in the outer for-loop placed under the inner for-loop to perform a check after (i) has been compared with (j) in the algorithm, which returns the median if condition is satisfied. There are 2 if-statements within the inner for-loop, if these statements are satisfied then the appropriate variable is incremented. The variable *numsmaller* is incremented when the inner for-loop finds a number smaller than the outer for-loop counting how much numbers in the array have been smaller or equal than the *kth* value in the previous iterations. On the other hand, *numequals* is incremented when the *array[j]* is equals to *array[i].* Both variables are assigned with a value of 0 each time the outer for-loop is ran. The median is returned when the variables are smaller than the *k* and the *k* is less than or equals to *numequals* and *numsmaller* added together.

Median:

The algorithm being analysed verifies where the array contains more than one item. If it contains one item, then the item value is returned. If the array contains more than one item, then the item, a sub-algorithm named *Select* is called. The select algorithm

recurs until the desired index value is reached in which it then returns the value at that specified index location. Select also calls *Partition* which leniently sorts the array by putting the larger values at the end of the array. It then returns the value of the index of the array that indicates the beginning of the sorted partition. The partition algorithm does a linear search through the array by comparing each array value with the array value in the 0^{th} index.

Pos variable is the value that *Partition* returns which is then compared to *m* (predicted position of the median) which value is then returned as the median. *Pivotloc* variable specifies where partition should start or ends the search. It is the boundary of the partition. *Pivotval* variable is the first index of the partition in the search. *Swap* method swaps the value in the pivot location with the last location of the partition.

2 Theoretical Analysis of the Algorithm

Brute Force Median:

Judging from the name of the algorithm it states that it is a Brute Force. Brute Force algorithms usually have an order of growth O(n) which means the basic operation count or time elapsed is increases linearly depending on the size of input. However, this is only partly the case because in this situation, we have 2 for-loops. The algorithm compares each element to the rest of the array, which means 10 elements in an array is compared 10 times to each other in the worst-case scenario. This means that if the median is at the very last element, the algorithm will compare each element to each other giving us an **n*n** or $\mathbf{n^2}$ quadratic order of growth. This type of order of growth also affects the average efficiency which in this case should be **avg = n*(n+1)/2.** The order of growth tells us that the basic operation count and execution time will be dependent on the size of input. Specifically, basic operation is quadrupled each time the size of input is doubled.

Median:

After a brief analysis of the algorithm's control flow, it has a few characteristics that resembles a transform and conquer type algorithm. The algorithm has a *pre-sorting* mechanism to find the median so that it can narrow the selection/finding process after each iteration. Hence that we pre-determined the algorithm to be transforming/simplifying the problem at hand. Therefore we can conclude that its order of growth is **Θ(n log (n))**.

2.1 Identifying the Algorithm's Basic Operation

Brute Force Median:

Basic operation in an algorithm is the line of code that is executed the most in a worst-case scenario. It can be an assignment, conditional, mathematic calculation, or array indexing. In this case, we have chosen the conditional/comparison line in the inner for-loop ***if(array[j] < array[i]) do***. One of the reasons why we chose this is because it is the foundation of the key comparison for the values. This is the first line of code executed when the inner for-loop is entered therefore is always checked.

Another reason why this was chosen over the other if-statement because the else-if could be skipped should this condition be satisfied. This line of code is executed more than the assignments, other comparisons, and return statements.

Median:

The Median calls the helper method *Select* which also calls *Partition*, and after the *Partition* returns a value the control flow goes back to the *Select* again which enters Boolean statements then depending on the result, the median is either returned or the Select function calls itself with a different start point from the previous iteration. This made the *i*dentification of the basic operation in the Median algorithm a little more complex than the Brute Force Median. As we have mentioned in the description section, this algorithm calls a few other helper methods, therefore we really had to consider the control flow to find the basic operation. We concluded that the basic operation should be the line ***if(array[j] < pivotval)*** in Partition. Partition is called many times in each time the algorithm is called. It is the function that occurs the most because it scans through the array elements. It contains a for loop which has an if-statement, this if statement is the basic operation because similar to the other algorithm, this is the first line of code executed in the for loop. Although this is not directly deciding which element the median is, it plays an important role in partially sorting the array in which case this returns the position so that the Select can do the final comparison to return the median.

2.2 Average Case Efficiency

Brute Force Median:

The average case efficiency is calculated by identifying the basic operation, counting the times it is executed with some iterations or recording each execution time per iteration. However, there are factors that could affect the elapsed time, some of which are:

- Computer processor (or load)
- Size of Input
- Operating System
- Hardware
- Programming Language

In our case, the average case efficiency is dependent on both the size of input and the position of the median. If the median is at the start, then the algorithm would only have to iterate once. Each iteration's basic operation count is linearly increased by the size of input. So, in an array of 5, the first element (0 index) is compared 5 times, the second element is compared another 5 times and so on. This means that the further the median is in the array, the longer the algorithm will take. For example:

Array	Number of Basic Operations
{3,2,1,4,5}	5
{1,2,3,4,5}	15(Average case)
{1,2,4,5,3}	25

Notice that with each iteration needed, the algorithm must execute more basic operations. This is because(j) is compared to each (i) in each iteration which ensures that (j) is compared to the next element until the median is found.

If we apply the theoretical average case efficiency **n*(n+1)/2** we get:

$C_{avg} = 5*(5+1)/2$
$C_{avg} = 15$

15 basic operations executed is the average case in these sets of arrays because the median is in the middle, therefore it is not the best (median in first index) nor is it the worst (median in last index).

Median:

Array	Number of Basic Ops
{5,4,3,2,1}	10
{3,2,1,4,5}	4
{1,3,2,4,5}	7
{1,2,3,4,5}	9
{1,2,4,3,5}	9
{1,2,4,5,3}	9

If the index of the median is at the start of the array, the algorithm will perform less basic operations because as it compares itself to the other values in the array, and searches for the median by matching the *predicted* median position which was identified using array length divided by 2 and rounded down. The algorithm partially sorts the array into ascending order, hence that the first array has a larger basic op count.

A single iteration on an array of 5 values returns one less than its size. The algorithm does less search than in the previous iteration because it excludes an end value.

Notice how the second and third array's medians are 2 and 1 positions/index away from the predicted median position. In the second array's case, it only goes through the algorithm once because it only needs to sort the value 3 and 1 and it can be done in one iteration. On the other hand, the third array has more basic ops because it had to iterate twice before finding the *actual* median value in the array which then determines the median and placed into its own index.

Therefore, we can conclude that the average efficiency would be if the median is positioned in the second quartile index of the array.

2.3 Order of Growth

Brute Force Median:

This algorithm has a quadratic order of growth, as mentioned in the specification. Quadratic growth is a type of growth that when the array is doubled in size, the basic operation count/running time is quadrupled hence the order of growth is **$\Theta(n^2)$**. To demonstrate this mathematically:

Arrays	Basic Operation Count
5	5*5 = 25
10	10*10 = 100
20	20*20 = 400
40	40*40 = 1600

Notice that every time the size of input is doubled, the basic operation count is quadrupled. I performed this demonstration (in code) by putting the median to the end of each array so that the $\mathbf{n^2}$ is easily related. See Figure 1.

Median:

Median's Order of Growth is fairly similar to the heapsort's sub-routines *heapify* and *siftdown*, but has a variation of Transform and Conquer *pre-sorting.* The algorithm has a mechanism that pre-sorts the array to a variation by isolating end values in each selection/partition it creates. This then helps it find the median by narrowing down the amount of values to analyse therefore 'transforming' the data hence it is a 'transform' type of algorithm. This effectively giving us a **Θ(n log (n))** order of growth.

3. Methodology Tools and Techniques

Brute Force Median and Median:

We implemented the algorithm in the programming language Java. If the array's elements are specified and are unsorted/sorted, we used Text mechanic. It is a website that allows the users to generate a large amount of numbers and as well as manipulating them by adding prefixes or suffixes characters to each number. Calculating the time for the algorithm was done by using Java's *System.nanoTime().*

The tests were implemented in an Acer Aspire 5750z Notebook. It contains an Intel Pentium B960 clocked at 2.2Ghz Dual Core. 2 GB of DDR3 Memory, 64-bit OS Windows 10.

The tables and graphs used to display results are from Microsoft 365 Pro Plus Excel and Microsoft Office 2010 Excel. I used this program to create tables and generate line graphs with the data.

3.1 Testing methodology

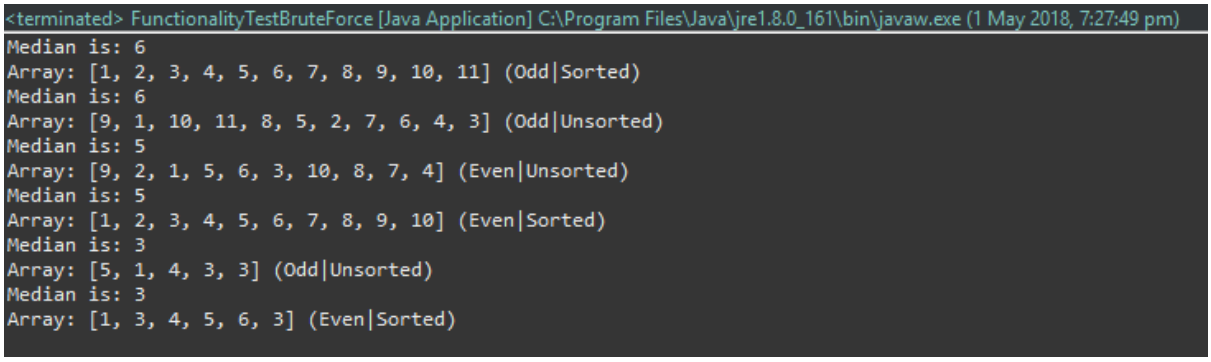
Brute Force Median & Median:

A testing method called Test() is created and changed appropriately for each experiment. It calls the algorithm and returns a string with the results and/or other variables relevant to the test. In basic operations test, we returned the array length and the basic operation count, whereas in time execution test the basic operation count was not needed therefore it wasn't necessary to be in the Test() method. There are also some lines that are added within the algorithm for testing purposes, such as incrementing the basic operation variable.

4 Functional Testing to Prove Correctness

Brute Force Median:

The functional testing was done by having 4 sets of 3 arrays each. We tested the functionality by having number of elements in the array that are even sorted, even unsorted, odd sorted, odd unsorted and even and odd with a duplicate median. The tests were successful in getting the correct median value.

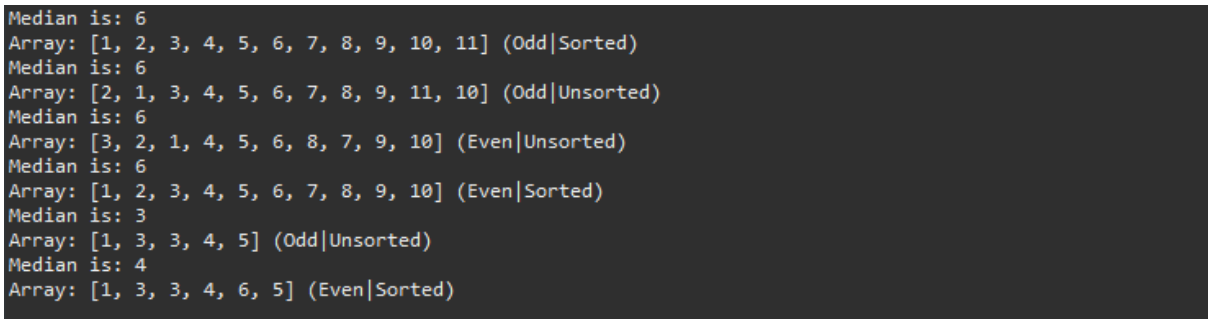


The tests showed that it didn't matter if the array were sorted or unsorted and that the algorithm will still find the right value.

During our testing, we double checked the average efficiency and order of growth analysis by performing different test cases. Our experimental results match the theoretical prediction of the Brute Force Median algorithm. The average efficiency was confirmed by applying the formula **n*(n+1)/2** which indicates that the median is where it's supposed to be in the middle. The order of growth was also confirmed by placing the median value to the last element of the array which indicates the worst-case scenario and better demonstrates the quadrupled basic of operation count **$\Theta(n^2)$**.

Median:

We tested the functionality for median the same way we did Brute Force. Which is by having number of elements in the array that are even sorted, even unsorted, odd sorted, odd unsorted and even and odd with a duplicate median. The tests were also successful in getting the correct median value.



Behaviour Difference between 2 algorithm's functionalities

Notice that when the array is even, the Brute Force median returns the number on the left side of {..5,6..} which is 5. Whilst the Median algorithm, returns the right side which is 6.

5 Counting the Number of Basic Operations

Brute Force Median:

An algorithm's efficiency can be tested by calculating the execution time and/or counting the basic operations. We can test the algorithm for time repeatedly but by counting the basic operation, it should give us a good estimate of what the efficiency is without exhaustive testing.

For testing purposes, we had to add some code in the algorithm to keep count of the basic operation *(See Appendix A).* We used a counter variable and inserted it in the inner for-loop before the if-statement (chosen basic operation), having this counter increment each time the for loop is ran gives us the basic operation count. For this experiment, I created 5 arrays that will show the best, average and worst-case scenario to show the effect of the median's position on the basic operation count.

Arrays	Basic Op Count
{3,1,2,4,5}	5(Best Case)
{1,3,2,4,5}	15
{1,2,3,4,5}	15(Average Case)
{1,2,4,3,5}	20
{1,2,4,5,3}	25(Worst Case)

After performing this test, we concluded that the theoretical analysis about the number of basic operations was correct. For each time the algorithm must iterate, the longer it will take. In the first and last array, the algorithm starts its search from the left and if the median is positioned there, then the algorithm wouldn't have to search any longer. On the other hand, because the median is at the very end, the algorithm had to run more iterations.

I performed another with larger number of arrays and elements for another look at the basic operation count. Similar to the order of growth test, we this test will show that the count will increase by four-fold should the size of input be doubled but this time we will use properly sorted arrays and constant difference in the array. This will prove and show both the average efficiency and order of growth theories. See Figure 2 to see results in graph, which I will explain in **Section 5.1.**

Median:

As previous done with the Brute Force Median algorithm, a counter variable was added to count the number of basic operations performed (see Appendix B). It would seem that the order of the array values plays a significant role in the amount of basic operations performed. If the array is in a descending order, more basic operations are performed to roughly sort the array in ascending order. If the ***actual*** median is in

the first array index, the algorithm iterates once, as it is comparing the first value in the selection, with the rest of the values and in this case, it has already located the median value and must reorder the array.

Arrays	Basic Op Count
{5,4,3,2,1}	10 (Worst Case
{1,2,3,4,5}	9
{3,2,1,4,5}	4 (Best Case)
{1,3,2,4,5}	7
{1,2,4,5,3}	9 (Average Case)

In case 3 where the array being used is {3,2,1,4,5}, this array is relatively sorted in ascending order. Seeing as the ***actual*** median (3) is the first value that is used to compare with the rest of the array values means that the algorithm only iterates once. In this case, there are 5 array values in total, meaning that the first value of the array is compared with the remaining 4, which would then explain a single iteration of the algorithm executing the basic operation 4 times.

In case 4 where the array being used is {1,3,2,4,5}, the array is still relatively sorted in ascending order, but the ***actual*** median is now in the second position rather than the first. If the median is not found in the first iteration of the algorithm, the algorithm would then iterate once more. After each iteration, the value selection is reduced by 1, meaning 1 value is excluded. Therefore, upon the second iteration, the first value in the selection would then be compared with the remaining 3 values. This explains why the basic operation count for this array is 7, because the first iteration of the algorithm returns 4 basic operations, and the second iteration returns 3 basic operations. The sum of the basic operations performed in those 2 iterations is 7.

In other cases, after the first iteration of the algorithm the array is partially sorted into ascending order, in the sense that the quartiles are sorted in ascending order rather than each individual value. In terms individual value comparison with the rest of the selection, only the first half of the array values are individually compared. According to test case 2 and 5, when the array is in a relatively ascending order, the basic operation count will be the same if the ***actual*** median value is located to the right of the predicted median index location.

Aside from the sorting of values, the other important factor that affects the number of basic operations would seem to be the array size as can be seen in figure 3.

5.1 Trends identified in Basic Operations – Prediction Match

Brute Force Median:

The **blue** datapoints in the graph represent the number of basic operations in each array. Each of these blue dot points also represent the average basic count of each array, hence if we apply the average efficiency formula in the first dot point where the size input is 10:

Avg = n*(n+1)/2

Avg = 10*(10+1)/2

Avg = 50 basic ops.

The **x** axis represents the size of input whilst the **y** axis represents the basic op count.

Median:

Similarly to the brute force median, the blue data points in the graph represent the number of basic operations performed for each array. There is a slight trend in basic operations that correlates the amount of basic operations performed to the size of the array used. As the array size increases, as does the number of basic operations performed. It would also appear that order of the array affects the amount of basic operations performed. If the ordering of the array is veering more towards a descending order, the amount of basic operations increases. The opposite can be said if the array is arranged roughly in an ascending order, the amount of basic operations decreases. It would also appear that the amount of basic operations performed increases depending on the location of the median value in the array. If the value is in an index position after the predicted median location, the amount of basic operations performed would be higher than if the value was in an index location before the predicted median location. When the median value is in an index location after the predicted median location, that is the scenario that performs the averaging numbers of basic operations.

6 Execution Times

Note that the execution times will have some ‘spikes’ that might show the CPU processing inconsistency.

Brute Force Median:

The test program in Appendix C was used to measure the average case execution times for the Brute Force Median algorithm. In order to get an average, one-hundred arrays were created with an array size varying from 1-5000. Each individual array is then tested 10 times. Upon the testing of each array, the execution time is stored.

The execution times were measured by assigning and using the Java’s in-built Class System.nanoTime to a variable called *startTime* prior to the algorithm being executed. After the algorithm is executed, System.nanoTime is retrieved once again and assigned to a variable called *endTime*. The time taken is calculated by deducting startTime from endTime and dividing the sum by 1,000,000,000. The result is then stored in a variable called *runTime,* which is then displayed as the elapsed time in seconds. Using a for-loop, this process is repeated for 100 arrays of varying lengths as specified earlier.

The results data indicated that as the array increases the execution times also go up. Just as we’ve seen in the basic operation counting, there is a noticeable pattern that shows it has a quadratic grow.

Median:

The process for calculating the average execution time for Median is the same process stated above for Brute Force Median as can be seen in Appendix D.

6.1 Trends identified in Execution Times – Prediction Match

Brute Force Median:

Referring to figure 4, it is evident that the execution time increase when the array size increases. Each data point represents the average execution time of the algorithm being execution 10 times using a particular array. The execution time results from 100 arrays with an array length ranging from 1-5000 indicate a trend in execution times that would seem to be in the form of quadratic growth. This trend is quite consistent aside from a few minor setbacks in which it could be assumed that they occurred due to unforseen hardware performance inconsistencies.

Median:

As seen with the Brute Force Median execution times, the execution times for the Median algorithm are also seen to increase as the length of the array increases(see figure 5). Although a trend can be seen, it is significantly less noticeable. This is due to the amount of basic operations increasing or decreasing depending on the sorting of values in the array. Seeing as an array sorted in a way that resembles a descending order causes the amount of basic operations to increase, which would then increase the execution time. Similarly to the Brute Force Median execution times, it can be seen that there are some fairly major spikes in execution time which again, we can assume that they occur due to hardware performance inconsistencies.

Figures:

Figure 1 – Order of Growth(BruteForceMedian)

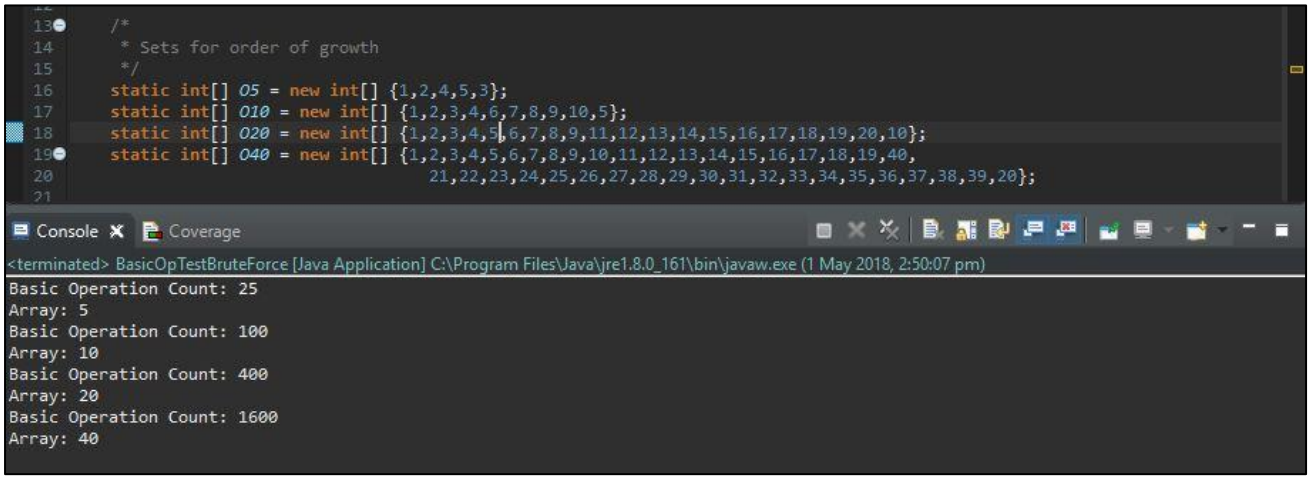


Figure 2 – Counting Basic OP(BruteForceMedian)

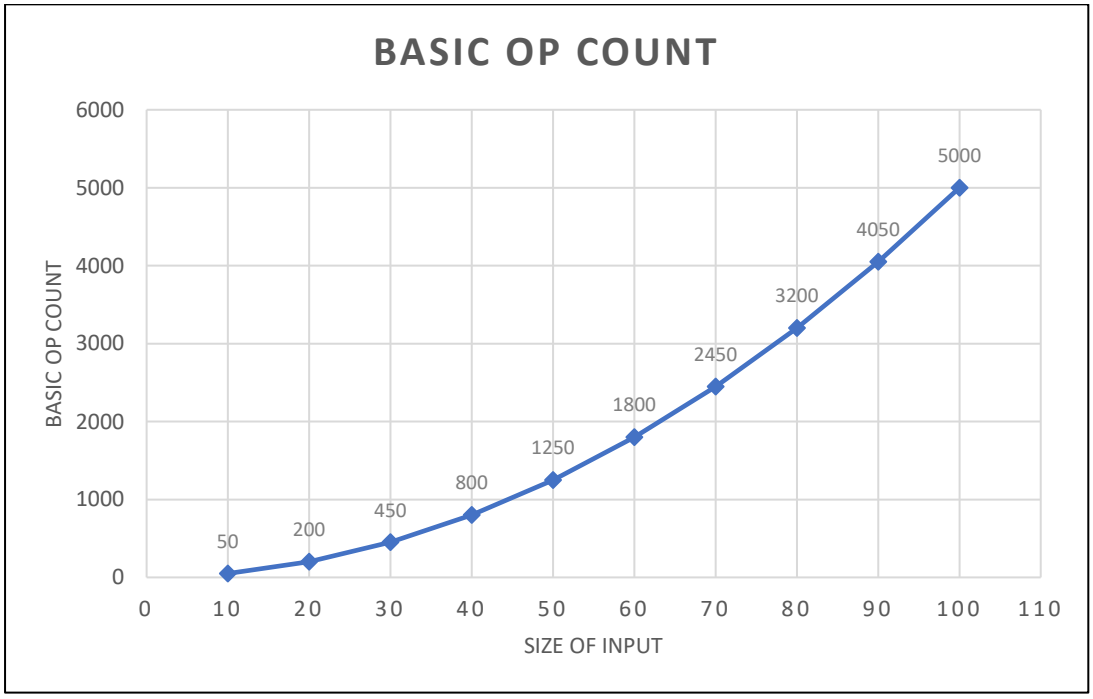


Figure 3 – Counting Basic OP(Median)

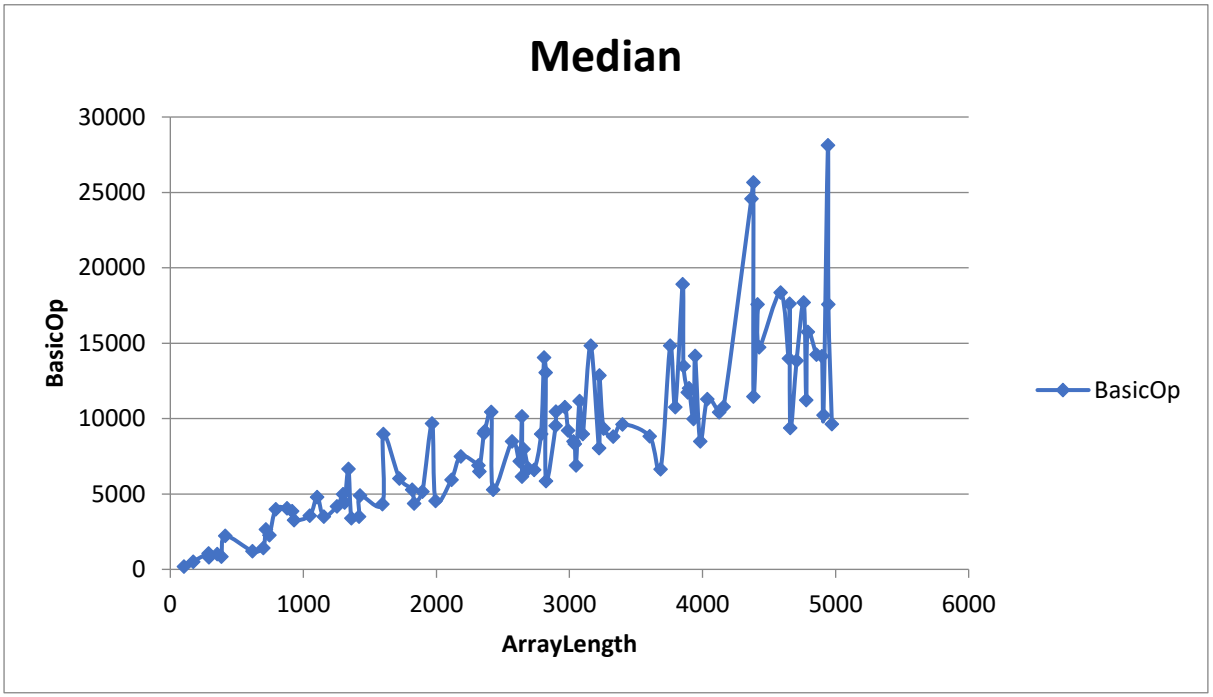


Figure 4 – Execution Time (BruteForceMedian)

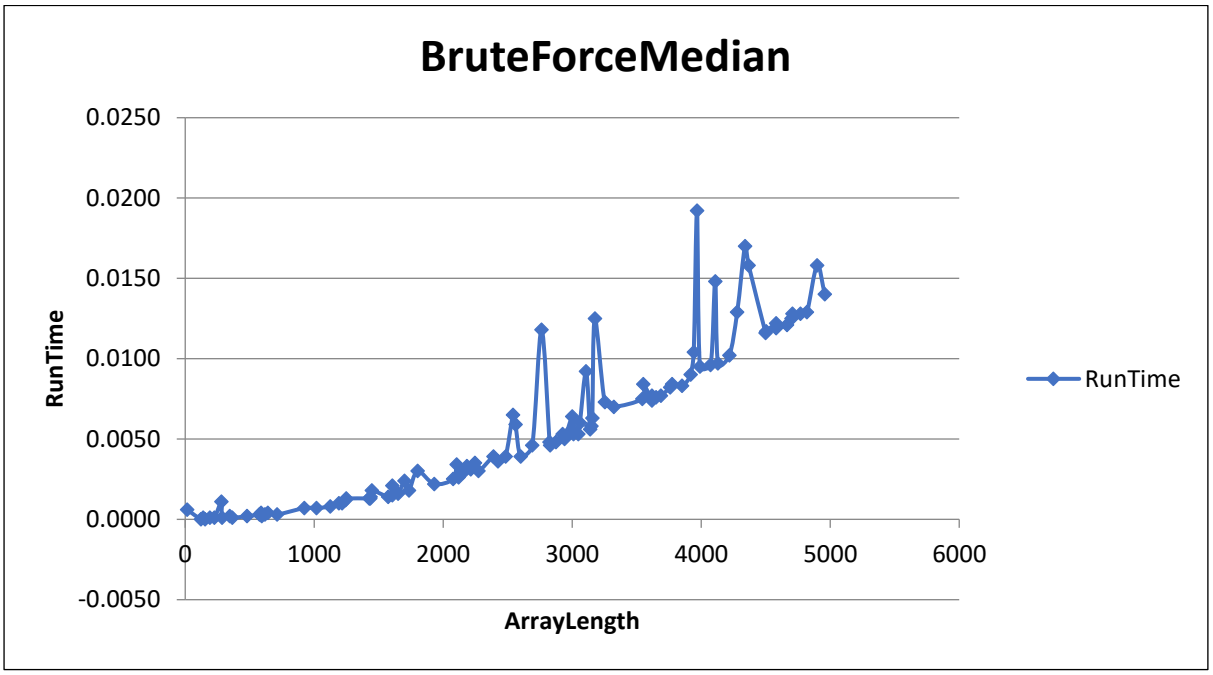
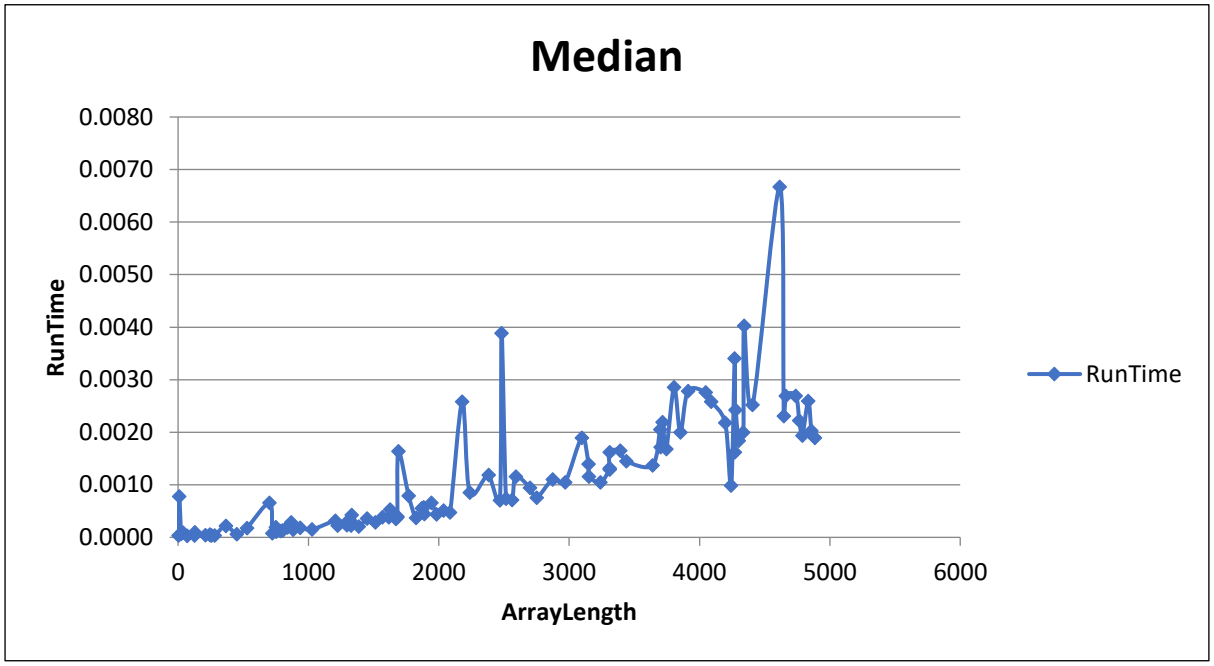


Figure 5 – Execution Time (Median)



Appendices:

Appendix A – Counting Basic OP(BruteForceMedian) Method.

```java
/*
 * BruteForceMedian Algorithm.
 * Used to find median if the array is UNSORTED.
 */
static int basicOp;
//static int basicOp1;
static int bruteForceMedian(int[] array) {
    basicOp = 0;
    int n = array.length;
    double k = Math.ceil((double) n / 2.0);
    for (int i = 0; i <= n - 1; i++) {
        int numSmaller = 0;
        int numEqual = 0;
        for (int j = 0; j <= n - 1; j++) {
            basicOp++;
            if (array[j] < array[i]) {
                numSmaller++;
            } else if (array[j] == array[i]) {
                numEqual++;
            }
        }
        if (numSmaller < k && k <= (numSmaller + numEqual)) {
            return array[i];

        }
    }
    return 0;
}
```

Appendix B – Counting Basic OP(Median) Method.

```
static int basicOp;
static int Median(int[] array) {
    basicOp = 0;
    int n = array.length;
    if(n == 1) {
        return array[0];
    } else {
        return Select(array, 0, Math.floor((n/2.0)), n - 1);
    }
}

/*
 * Select helper method for Median.
 */
static int Select(int[] array, int l, double m, int h) {
    int pos = Partition(array, l, h);
    if(pos == m) {
        return array[pos];
    }
    if(pos > m) {
        return Select(array, l, m, pos - 1);
    }
    if(pos < m) {
        return Select(array, pos + 1, m, h);
    }

    return 0;
}

/*
 * Partition helper method for Select.
 */
static int Partition(int[] array, int l, int h) {
    int pivotval = array[l];
    int pivotloc = l;
    for(int j = l + 1; j <= h; j++) {
        basicOp++;
        if(array[j] < pivotval) {
            pivotloc++;
            swap(array[pivotloc],array[j], pivotloc, j, array);
        }
    }
    swap(array[l], array[pivotloc], l, pivotloc, array);
    return pivotloc;
}

/*
 * Swap helper method for Partition.
 */
static void swap(int value1, int value2, int index1, int index2, int[]array)
{
    int temp = value1;
    array[index1] = value2;
    array[index2] = temp;
}
```

Appendix C – Execution Times(BruteForceMedian) Method.

```java
@Test
  public void testExecutionTimeBruteForceMedian()
  {
      double runTimeAverage = 0.0;
      double runTimeTotal = 0.0;

      for (int i = 0; i < 100; i++)
      {
          createArray("5000");
          runTimeTotal = 0;
          runTimeAverage = 0;
          for (int j = 0; j < 10; j++)
          {
              startTime = System.nanoTime();
              bruteForceMedian(arrayData);
              endTime = System.nanoTime();
              runTime = ((endTime - startTime)/1000000000);
              runTimeTotal += runTime;
              runTimeAverage = runTimeTotal/10.0;
              if (j == 9)
              {
                  try
                  {
                      String str = String.valueOf(runTimeAverage);
                      String str2 = String.valueOf(arrayData.length);
                      BufferedWriter writer = new BufferedWriter(new FileWriter("C:/Users/Nathan/Documents/runTimeBruteForceMedian.csv", true));
                      writer.write(str2 + "," + str);
                      writer.newLine();
                      writer.close();
                  }
                  catch(IOException e)
                  {
                      System.out.println("Oops");
                  }
              }
          }
      }
```

Appendix D – Execution Times (Median) Method.

```java
@Test
public void testExecutionTimeMedian()
{
    double runTimeAverage = 0.0;
    double runTimeTotal = 0.0;

    for (int i = 0; i < 100; i++)
    {
        createArray("5000");
        runTimeTotal = 0;
        runTimeAverage = 0;
        for (int j = 0; j < 10; j++)
        {
            startTime = System.nanoTime();
            Median(arrayData);
            endTime = System.nanoTime();
            runTime = ((endTime - startTime)/1000000000);
            runTimeTotal += runTime;
            runTimeAverage = runTimeTotal/10.0;
            if (j == 9)
            {
                try
                {
                    String str = String.valueOf(runTimeAverage);
                    String str2 = String.valueOf(arrayData.length);
                    BufferedWriter writer = new BufferedWriter(new FileWriter("C:/Users/Nathan/Documents/runTimeMedian.csv", true));
                    writer.write(str2 + "," + str);
                    writer.newLine();
                    writer.close();
                }
                catch(IOException e)
                {
                    System.out.println("Oops");
                }
            }
        }
    }
```