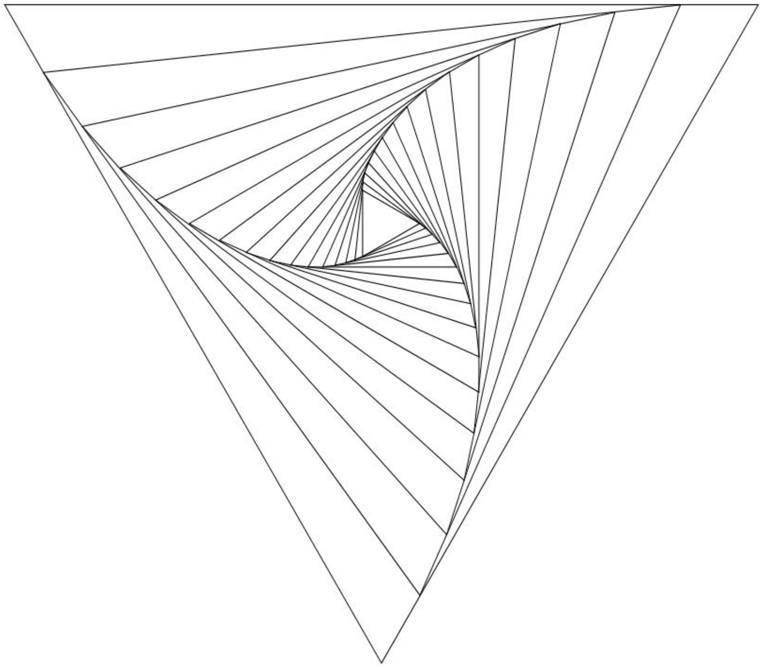
```
 ____                      ______                 ______                  ___
/\  _`\                   /\__  _\__             /\  _  \                /\_ \                   __
\ \ \L\ \   __  __    ___  \/_/\ \/\_\    ___ ___      __      \ \ \L\ \     ___      __  \//\ \    __  __    ____/\_\    ____
 \ \ ,  /  /\ \/\ \ /' _ `\ ______\ \ \\/\ \ /' __` __`\  /'__`\     \ \  __ \  /' _ `\  /'__`\ \ \ \  /\ \/\ \  /',__\/\ \  /',__\
  \ \ \\ \ \ \ \_\ \/\ \/\ \/\______\\ \ \\ \ \/\ \/\ \/\ \/\  __/      \ \ \/\ \/\ \/\ \/\ \L\.\_ \_\ \_ \ \ \_\ \/\__, `\ \ \/\__, `\
   \ \_\ \_\\ \____/\ \_\ \_\/______/ \ \_\\ \_\ \_\ \_\ \_\ \____\      \ \_\ \_\ \_\ \_\ \__/.\_\/\____\\/`____ \/\____/\ \_\/\____/
    \/_/\/ / \/___/  \/_/\/_/          \/_/ \/_/\/_/\/_/\/_/\/____/       \/_/\/_/\/_/\/_/\/__/\/_/\/____/ `/___/> \/___/  \/_/\/___/
                                                                                                             /\___/
                                                                                                             \/__/
```

a(n)



Jason Ivey 2017 - EPCC - Prof. Frank Blando

Explanation for This Experiment

Fundamental to the overarching themes of Programming Fundamentals III is optimization and runtime. This lab assignment is an exercise in student knowledge of these tropes and education as far as presentation of data.

More specific to the task at hand, three functions for the Max Sum of a set of numbers were provided. These functions and their runtime equation are provided in the table below:

Method 1 Brute Force	$f(n) = n^3$
Method 2 Quadratic	$f(n) = n^2$
Kadane's Algorithm	$f(n) = n$

After further investigation I dismissed these general formulas for method runtime for more specific formulas[1] as provided in the improved table below:

Method 1 Brute Force	$f(n) = 5n^3 + 7n^2 + 3n + 2$
Method 2 Quadratic	$f(n) = 7n^2 + 5n + 4$
Kadane's Algorithm	$f(n) = 9n + 3$

For details on how I reached these estimates see Page 6.

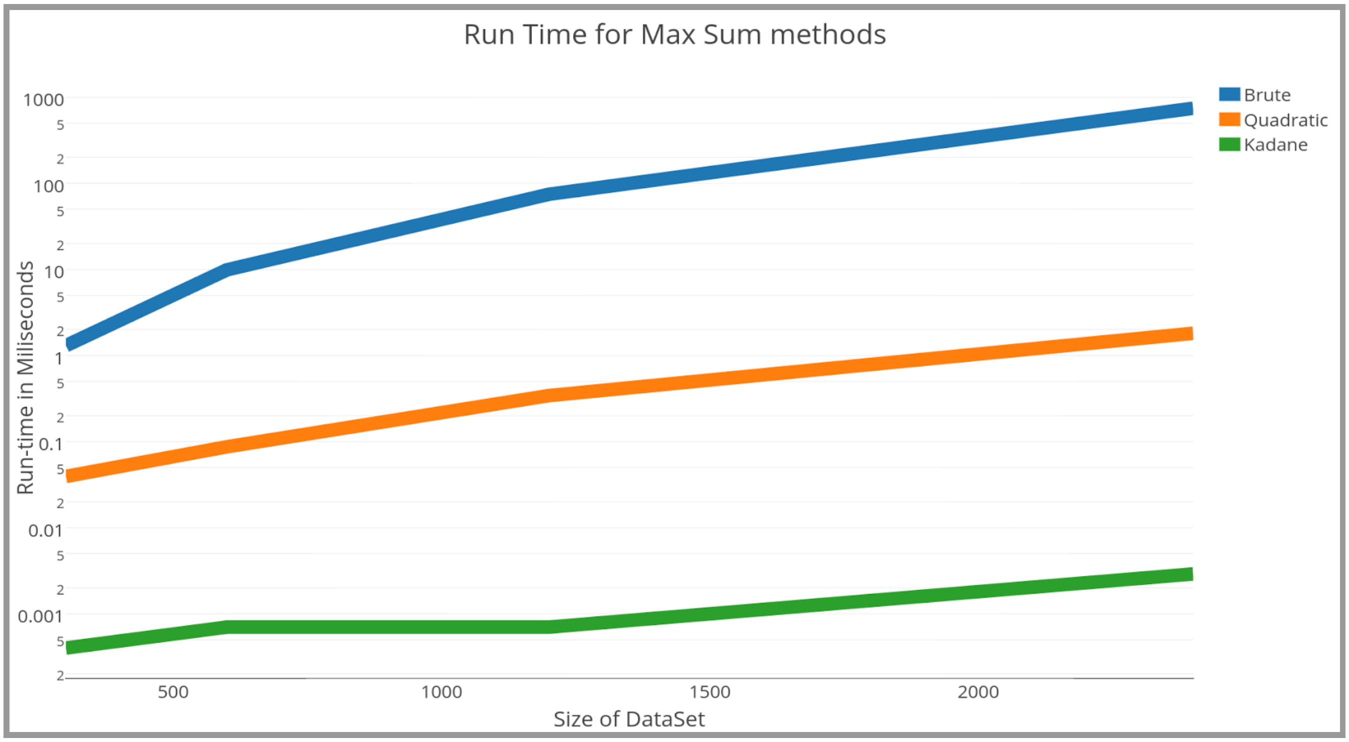
[1] Worst case Big O estimates

Data Visualization

It is important to realize that due to the unpredictability of the JVM I have decided to average the results over 10,000. All values are in milliseconds.

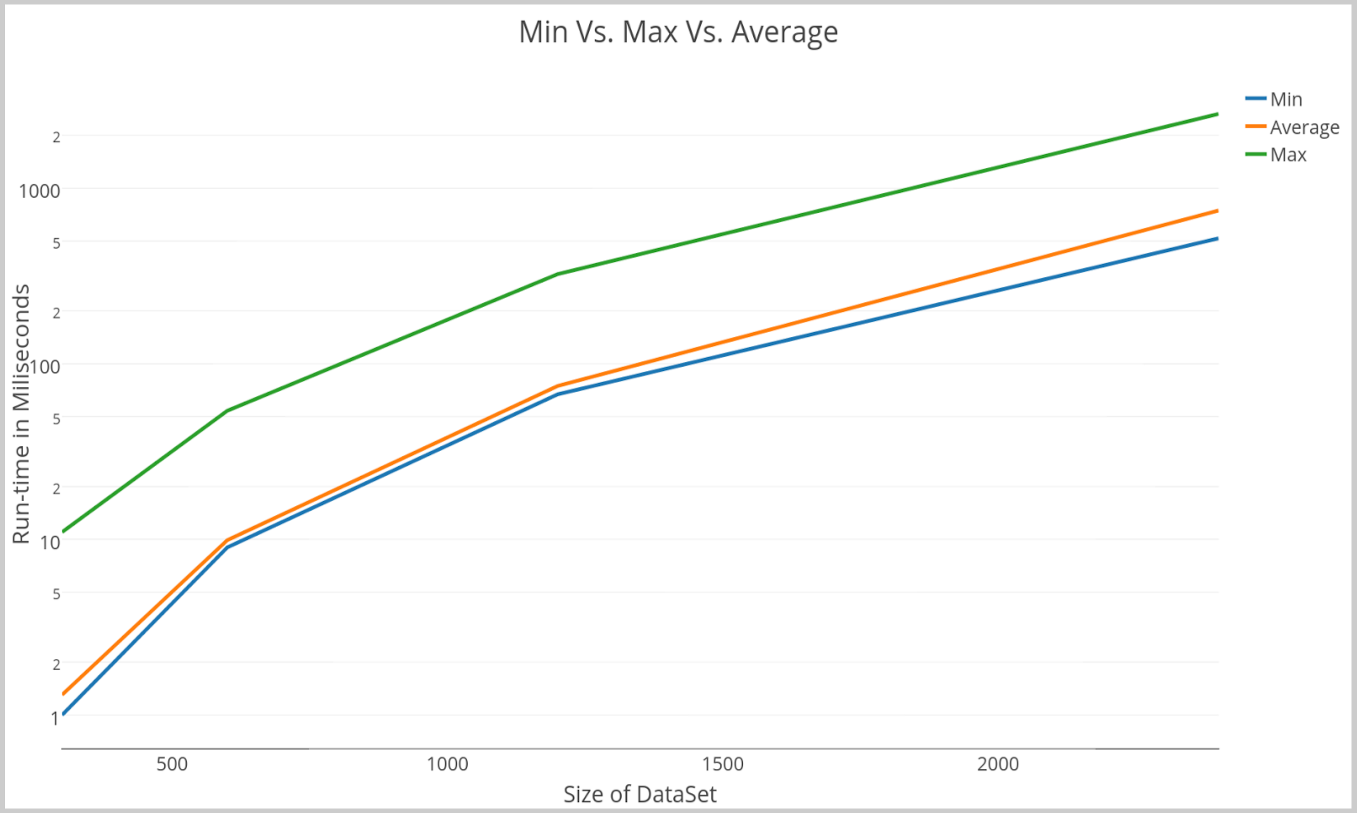
Therefore these values should be extremely stable for hardware similar to my own, to see if your hardware is similar to mine I have included hardware specifics at the end of this document.

Size of Array	Brute Force	Quadratic	Kadane’s
300	1.3022	0.0394	0.0004
600	9.8953	0.0871	0.0007
1200	74.7566	0.3426	0.0007*
2400	745.9027	1.8129	0.0029



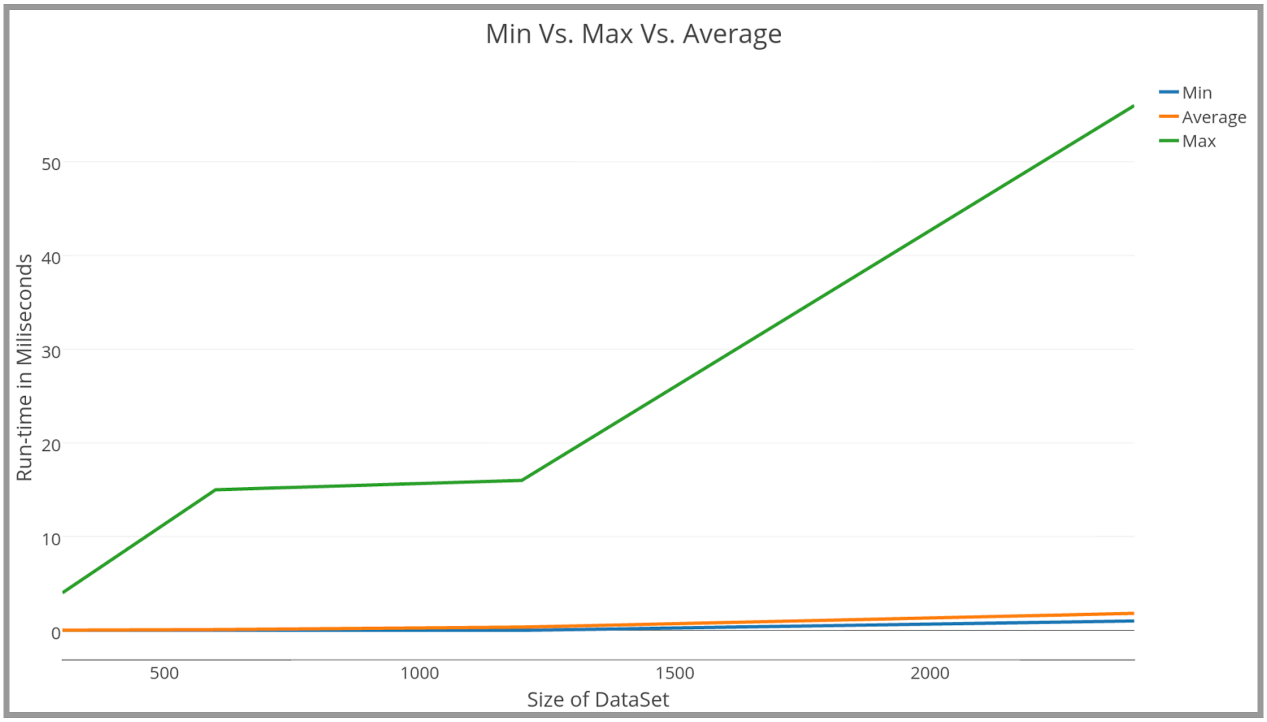
Brute Force Min, Max, & Average:

Size of Array	Min	Average	Max
300	1.0	1.3022	11.0
600	9.0	9.8953	54.0
1200	67	74.7566	324.0
2400	518.0	745.9027	2647.0



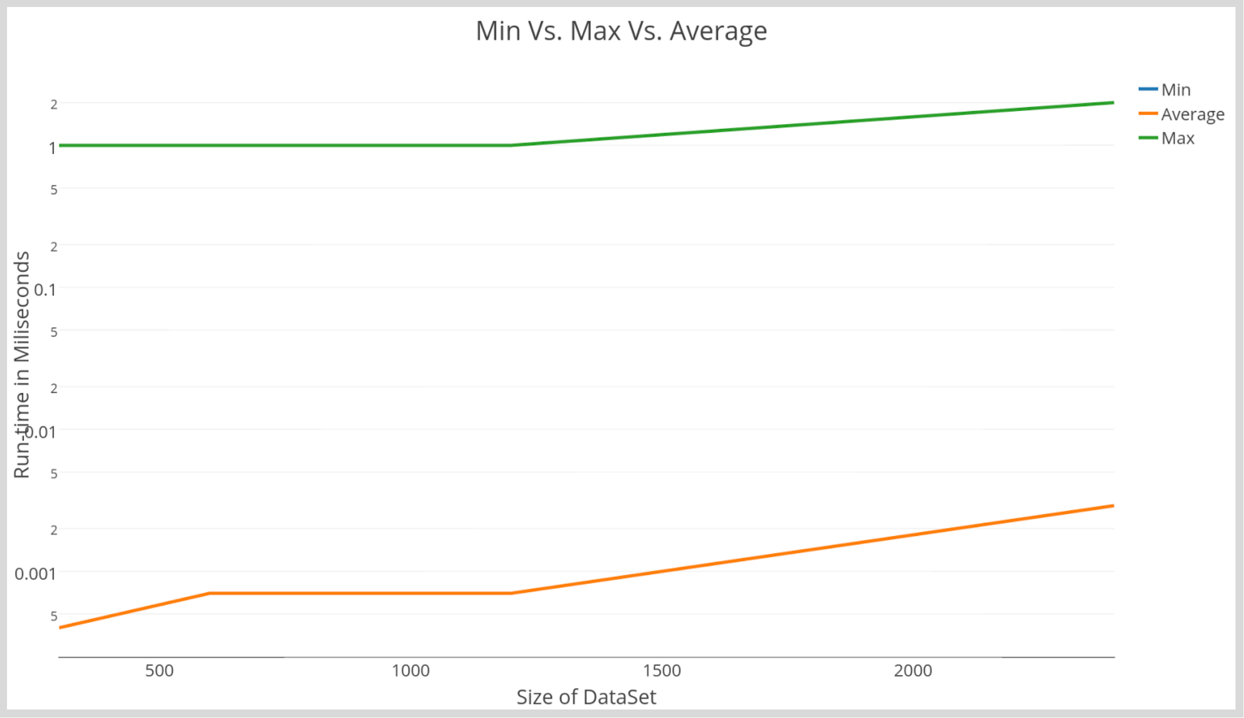
Quadratic Min, Max, & Average:

Size of Array	Min	Average	Max
300	0.0	0.0219	4.0
600	0.0	0.0871	15.0
1200	0.0	0.3426	16.0
2400	1.0	1.8129	56



Kadane’s Min, Max, & Average:

Size of Array	Min	Average	Max
300	0.0	0.0004	1.0
600	0.0	0.0007	1.0
1200	0.0	0.0007*	1.0
2400	0.0	0.0029	2.0



The improved equations for the source code come from number of operations in each respective method. Each operation is repeated by the size of the dataset in each for loop. To show the process for which I arrived at these specific equations here is pseudo representation of the functions, where only the number of operations being done are shown:

```
maxSum(n)
{
  2 operations;
  for(  1 operation; 1 comparison; 2 operations )
  {
    for(  1 operation; 1 comparison; 2 operations )
    {
      1 operation;
      for( 1 operation; 1 comparison; 2 operations )
      {
        2 operations;
      }
      if( 1 comparison )[2]
        1 operation;
    }
   }
   return maxSum;
}
```

We then take the number of operations and multiply it by n to the power of the number of nested loops, doing this we get:

$$f(n) = 5n^3 + 7n^2 + 3n + 2$$

If we repeat this process for the other two functions we will find a more exact number of operations for those methods.

[2] Since we are doing Big-O or worst case, all conditions are true that worsen runtime.

What Numbers I Used & Their Meaning

The Command Line Arguments:

```
java FunctionTester 300 4
```

- The command line arguments passed are 300, & 4.
    - 300 denotes the starting array size.
        - A random array of this size will be generated.
    - 4 denotes the amount of tests to be done
        - Each test will increase the array size by a constant rate.

The constant rate by which the array size is multiplied by is **2.** For better consistency the three being tested are run with one dataset 10,000 times and then averaged. You might think, “Hey that’s over kill.” Let me remind you this is **THE** we are dealing with here.

The dataset is generated by a the Random object from the: java.util.Random, package. This choice in random maker was not random as this is the preferred choice in Random number generators for the Java language. The range of the numbers has no effect on the run-time but is: [1, 500].

Interpretation of Results

The functions that perform the least amount of repetition will be the most optimal for the number of operations necessary. This is an extremely important as processing power continues to stagnate. The exponential growth of yesteryears has ceased and so it is no longer an Electrical Engineer’s duty to build hardware for a Computer Scientist’s software, but rather now the Computer Scientist must make software that can run on stagnate hardware.

The important word during run-time analysis is repetition, while this may bring to mind loop (which does include) this can also be recursive repetition or just hard coded repetition, or even for non-Java code the notorious Goto function. This repetition is almost always necessary, but the amount of repetition can increase runtime so much so that entire timeline of the universe’s existence would elapse before an algorithm would complete. So it is of great importance that runtime be examined and represented mathematically before it is implemented.

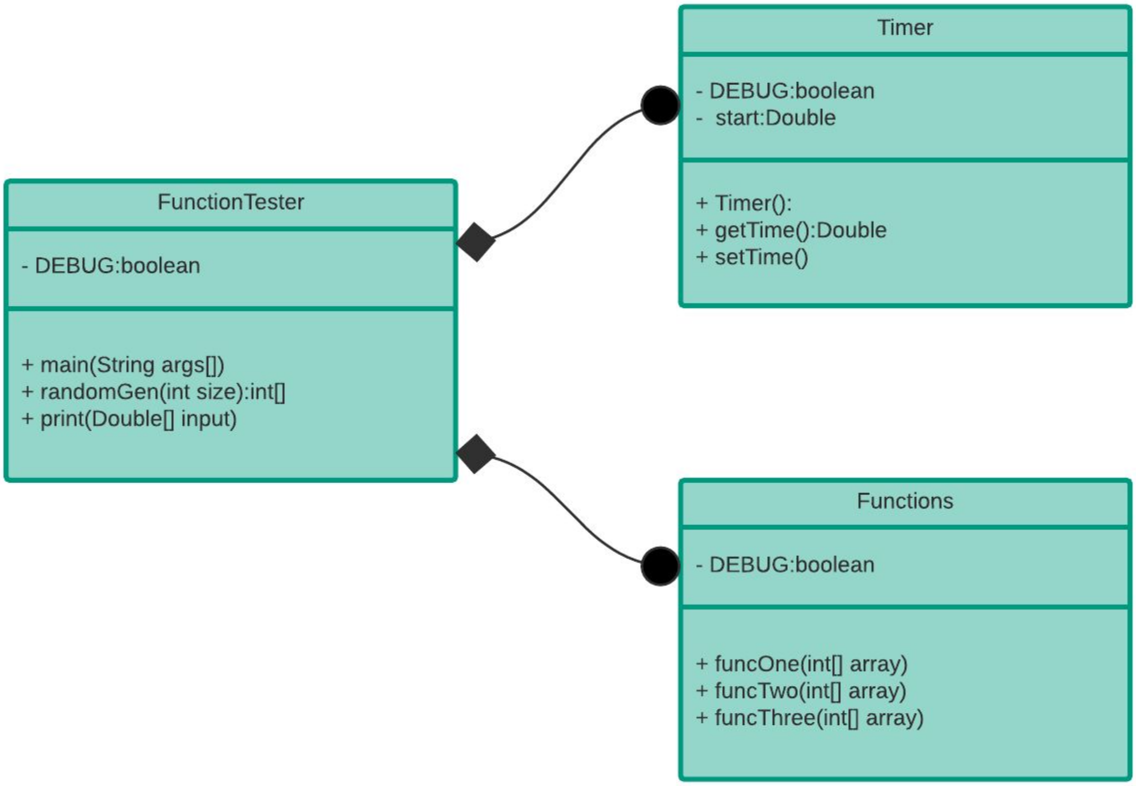
This lab was just that action of examining the runtime of three methods. So to rank these specific functions regarding their runtime and optimization the ranking would be as follows:

1. Kadane’s Function n
2. Quadratic Function n^2
3. Brute Force n^3

The results of this analysis verify that the growth of each function and difference in their execution time could be ranked alongside their mathematical representation. The validity of run-time functions goes without saying but validation is an important task when first exploring optimization.

CODE

This section is dedicated to the source code for this project.



Functions.java

```java
public class Functions{
        public static final boolean DEBUG = false;
        public static void funcOne(int a[])
        {
                int n = a.length;
                int i,j,k;
                int sum,maxSum = 0;
                for(i=0; i<n; i++)
                {
                for(j=i; j<n; j++)
                {
                        sum = 0;
                        for(k=i ; k<j; k++)
                        {
                                sum = sum + a[k];
                        }
                        if(sum>maxSum)
                        maxSum = sum;
                }
                }
        }
        public static void funcTwo(int a[])
        {
                if(DEBUG){System.out.println("funcTwo Called for array length: " + a.length);}
                int n = a.length;
                int i,j,sum,maxSum;
                maxSum = 0;
                for(i = 0;i<n;i++)
                {
                        sum = 0;
                        for(j=i;j<n;j++)
                        {
                                sum = sum + a[j];
                                if(sum>maxSum)
                                        maxSum = sum;
                        }
                }
        }
        public static void funcThree(int a[])
        {
                int n = a.length;
                int maxSum = 0,sum = 0;
                int i;
                for(i = 0;i<n;i++)
                {
                        sum = sum + a[i];
                        if(sum < 0)
                                sum = 0;
                        else if(sum > maxSum)
                                maxSum = sum;
                }
        }
       }
```

Timer.java

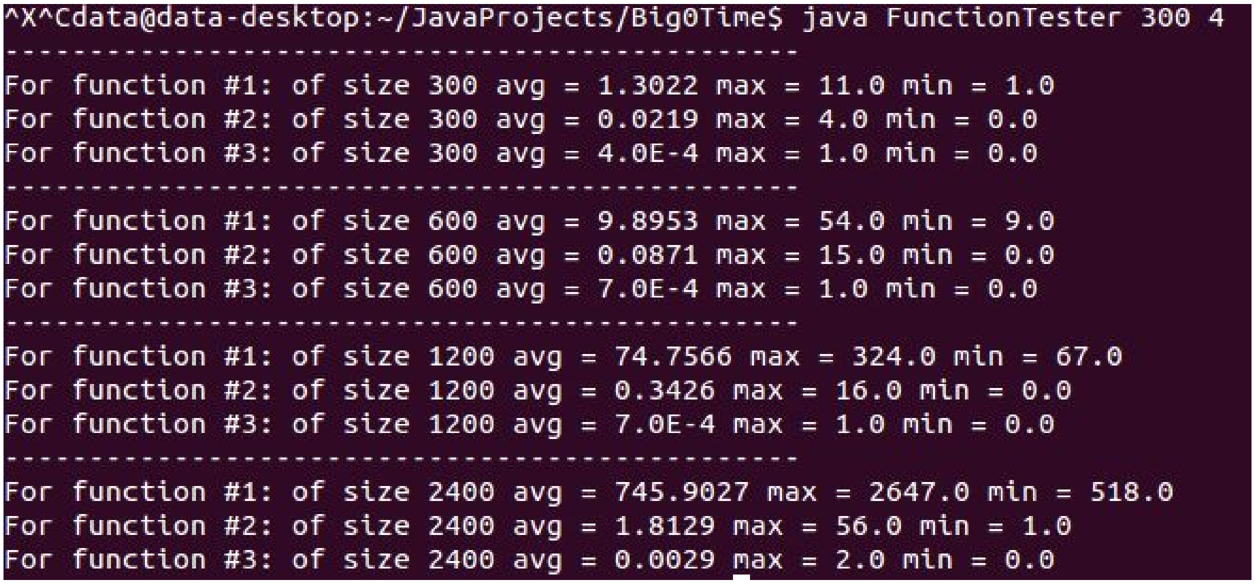
```java
public class Timer{
	public static final boolean DEBUG = false;
	private Double start;//Double of what time start is

	public Timer(){
		if(DEBUG){System.out.println("instance of Timer made");}
		start = new Double(System.currentTimeMillis()); //Inits timer with current start
time
	}

	public void setTime(){
		if(DEBUG){System.out.println("Time|Set");}
		start = new Double(System.currentTimeMillis()); //Sets start time
	}

	public Double getTime(){
		double currentTime = System.currentTimeMillis() - start; //Gets start time
		if(DEBUG){System.out.println("Time returned = " + currentTime);}
		return currentTime;
	}

}
```

FunctionTester.java

```
/**
        @author Jason Ivey

        Function tester tests the function class using
        the Timer object.
*/
import java.util.Random;
import java.util.PriorityQueue;
import java.util.Arrays;

public class FunctionTester{

        public static final boolean DEBUG = false;
        public static void main(String args[]){

                if (DEBUG) System.out.println("Called for:  " + args[0] + " : " + args[1]);


                int size      = Integer.parseInt(args[0]); //Size of array
                int tests     = Integer.parseInt(args[1]); //# of tests to run
                int rate      = 2;

                int[] array   = randomGen(size); //Makes array

                Timer timer   = new Timer(); //Our timer object
                /* Data[0] = function One
                 * Data[0][0] = is average time
                 * Data[0][1] = is max time
                 * Data[0][2] = is min time
                 * Same for other functions just incrementing first pointer
                 */
                Double data[][] = new Double[3][3];


                Double temp = new Double(0.0); //Temp Double holder

                for(int testsCompleted = 0; testsCompleted < tests; testsCompleted++){

                for(int iterations = 0; iterations < 10000; iterations++){

                                if(iterations == 0){ //If this is the first iteration

                                        timer.setTime();//Setting timer's start to now
                                        Functions.funcOne(array);//Running function one, or brute force
                                        temp = timer.getTime(); //Setting temp to current time
                                        data[0][0] = temp; //First run through, no comparison just init
everything to temp
                                        data[0][1] = temp;
                                        data[0][2] = temp;

                                        timer.setTime();
                                        Functions.funcTwo(array);
                                        temp = timer.getTime();
                                        data[1][0] = temp;
                                        data[1][1] = temp;
                                        data[1][2] = temp;

                                        timer.setTime();
                                        Functions.funcThree(array);
                                        temp = timer.getTime();
                                        data[2][0] = temp;
                                        data[2][1] = temp;
                                        data[2][2] = temp;

                                } else {//Other wise this is 1...10th execution

                                        timer.setTime(); //Setting timer's time to now
                                        Functions.funcOne(array);//Running brute force
                                        temp = timer.getTime(); //Setting temp to elapsed time since
start
                                        data[0][0] += temp;//Adding to average. will be divided by
number of additions later to get true average
                                        if( data[0][1] < temp){ //If max is less than temp, give max
temp
                                                data[0][1] = temp;
                                        } else if ( data[0][2] > temp ){ //If min is greater than temp,
give min temp
                                                data[0][2] = temp;
                                        }

                                        timer.setTime();
                                        Functions.funcTwo(array);
                                        temp = timer.getTime();
                                        data[1][0] += temp;
                                        if( data[1][1] < temp){
                                                data[1][1] = temp;
                                        } else if ( data[1][2] > temp ){
                                                data[1][2] = temp;
                                        }

                                        timer.setTime();
                                        Functions.funcThree(array);
                                        temp = timer.getTime();
                                        data[2][0] += temp;
                                        if( data[2][1] < temp){
                                                data[2][1] = temp;
                                        } else if ( data[2][2] > temp ){
                                                data[2][2] = temp;
                                        }

                                }

                        }

                        data[0][0] = data[0][0] / 10000.0; //Dividing averages by iterations, to get
true average
                        data[1][0] = data[1][0] / 10000.0;
                        data[2][0] = data[2][0] / 10000.0;

                        System.out.println("-----------------------------------------------");
                        print(1, data[0], size); //Printing results of this test
                        print(2, data[1], size);
                        print(3, data[2], size);
                        size = size * rate; //Increasing by the rate
                        array = randomGen(size); //New array huray!
                }
        }
        public static int[] randomGen(int size){

        if (DEBUG) System.out.println("Random gen = " + size + " ");
        int[] result = new int[size];
        Random rand = new Random();

        for(int iterations = 0; iterations < size; iterations++){

                result[iterations] = rand.nextInt(500) + 1; // returns randoms based 1 - 500
        }
        return result;
    }

        public static void print(int func, Double[] input, int size){
                //Prints out data based on the meaning of each position in the array
                System.out.println("For function #"+func+": of size "+size+" avg = "+input[0]+" max =
"+input[1]+" min = "+input[2]);
        }
}
```

Sample I/O

This is a screenshot of sample input output:



```
^X^Cdata@data-desktop:~/JavaProjects/BigOTime$ java FunctionTester 300 4
-----------------------------------------------
For function #1: of size 300 avg = 1.3022 max = 11.0 min = 1.0
For function #2: of size 300 avg = 0.0219 max = 4.0 min = 0.0
For function #3: of size 300 avg = 4.0E-4 max = 1.0 min = 0.0
-----------------------------------------------
For function #1: of size 600 avg = 9.8953 max = 54.0 min = 9.0
For function #2: of size 600 avg = 0.0871 max = 15.0 min = 0.0
For function #3: of size 600 avg = 7.0E-4 max = 1.0 min = 0.0
-----------------------------------------------
For function #1: of size 1200 avg = 74.7566 max = 324.0 min = 67.0
For function #2: of size 1200 avg = 0.3426 max = 16.0 min = 0.0
For function #3: of size 1200 avg = 7.0E-4 max = 1.0 min = 0.0
-----------------------------------------------
For function #1: of size 2400 avg = 745.9027 max = 2647.0 min = 518.0
For function #2: of size 2400 avg = 1.8129 max = 56.0 min = 1.0
For function #3: of size 2400 avg = 0.0029 max = 2.0 min = 0.0
```

The Input can be seen to be 300, 4.

The output is:

```
-----------------------------------------------
For function #1: of size 300 avg = 1.3022 max = 11.0 min = 1.0
For function #2: of size 300 avg = 0.0219 max = 4.0  min = 0.0
For function #3: of size 300 avg = 4.0E-4 max = 1.0  min = 0.0
-----------------------------------------------
For function #1: of size 600 avg = 9.8953 max = 54.0 min = 9.0
For function #2: of size 600 avg = 0.0871 max = 15.0 min = 0.0
For function #3: of size 600 avg = 7.0E-4 max = 1.0  min = 0.0
-----------------------------------------------
For function #1: of size 1200 avg = 74.7566 max = 324.0  min = 67.0
For function #2: of size 1200 avg = 0.3426  max = 16.0   min = 0.0
For function #3: of size 1200 avg = 7.0E-4  max = 1.0    min = 0.0
-----------------------------------------------
For function #1: of size 2400 avg = 745.9027 max = 2647.0 min = 518.0
For function #2: of size 2400 avg = 1.8129   max = 56.0   min = 1.0
For function #3: of size 2400 avg = 0.0029   max = 2.0    min = 0.0
```

Hardware Specification:

Hardware	Model	Speed/Data Size
CPU	AMD 9590	4.72 Ghz
GPU	GTX 970	1664 CUDA cores
RAM	Corsair Vengeance	32 GB DDR3
SSD	Samsung Evo	520 GB