**CSCI 2110 Data Structures and Algorithms**
**Laboratory No. 2**
**Week of 16 September**

**Due: Saturday 21 September**
**23h55 (five minutes to midnight)**

## Algorithm Complexity Analysis

This is an experimental lab in which you will write and run a number of small programs to get a hands-on understanding of algorithm complexity. Each program will implement a simple algorithm of a different time complexity. You will test how long each algorithm takes to execute on your machine for different inputs.

Although raw execution time is not an accurate measure of algorithm complexity, it is acceptable for this experiment, since you are not comparing your execution times with others. This is an experiment to see how a function grows as input size increases on your own machine. You can use the following code template to obtain the execution time of your code.

```
long startTime, endTime, executionTime;
startTime = System.currentTimeMillis();

//code snippet (or call to the method) here

endTime = System.currentTimeMillis();
executionTime = endTime - startTime;
```

The above code will give the time for executing the code snippet in milliseconds. You can display the executionTime using System.out.println and/or save it.

Note: The execution times shown in your output file will differ from those in the sample outputs, as well as from the times captured by your peers during their own experiments. Slight differences related to system architecture, computational resources, load, etc. make if very unlikely that any two students will submit identical outputs.

**Marking Scheme**
Each exercise carries 10 points.
Working code, Outputs included, Efficient, Good basic comments included: 10/10
No comments or poor commenting: subtract one point
Unnecessarily inefficient: subtract one point
No outputs and/or not all test cases covered: subtract up to two points
Code not working: subtract up to six points depending upon how many methods are incorrect.

Your final score will be scaled down to a value out of 10. For example, if there are three exercises and you score 9/10, 10/10 and 8/10 on the three exercises, your total is 27/30 or 9/10.

**Error checking**: Unless otherwise specified, you may assume that the user enters the correct data types and the correct number of input entries, that is, you need not check for errors on input.

**Submission**: All submissions are through Brightspace. Log on dal.ca/brightspace using your Dal NetId. Deadline for submission: Saturday 21 September 23h55 (five minutes before midnight).

**What to submit**:
Submit one ZIP file containing all source code (files with .java suffixes) and a text documents containing sample outputs, PNG or other image files containing your graphs. For each exercise you will minimally have to submit a demo Class with a main method implementing the algorithm to be tested and your test code, sample outputs, and a graph showing execution times. You will additionally submit a small text document with your response to the short answer question in Exercise3.
Your final submission should include the following files: Prime.java, Exercise1.out, Exercise1Plot.png, Collatz.java, Exercise2.out, Exercise2Plot.png, MatrixMult.java, Exercise3.out, Exercise3Plot.png, Exercise3.txt.

You MUST SUBMIT .java files that are readable by your TAs. If you submit files that are unreadable such as .class, you will lose points. Please additionally comment out package specifiers.

Note: In all the following exercises, you will be writing one Class file with static methods to accomplish the tasks given and a main (tester) method. As such, these are not necessarily object-oriented programs in the true sense.

**Exercise1 (Nth Prime)**
A prime number is a positive integer that has no factors other than 1 and itself. For example, the first six prime numbers are 2, 3, 5, 7, 11, and 13. If you are asked to find the $6^{th}$ prime number, the answer is 13.
For this exercise, you will write a program capable of calculating the $11^{th}$, the $101^{st}$, $1001^{st}$, $10001^{st}$, $100001^{st}$, and $1000001^{st}$ prime and determine how long your program takes to find each of those primes.
The skeleton of your Prime Class could look something like the code shared below. You can change and reuse this code if necessary. You can also add other static helper methods if wish.

```
/*
Prime Solution
*/

/**
This class tests the code for Lab2: Exercise1. It calls a method to
calculate the nth prime and prints information about running time.
It expands upon a framework provided by Srini.
*/

import java.util.*;

public class Prime{
  public static void main(String[] args){
    //TODO
   }

  public static long nthPrime(long p){
    //TODO
  }
}
```

Note: Use the long data type instead of int to accommodate large primes. If you use a naïve method to test if a value *x* is prime (testing all values less than *x* to determine if they are factors for example) then your program will take a long time to calculate the 1000001th prime. There are smarter ways to test if a number the prime that will reduce your execution time.

Your program should accept input in the following format:

a) Positive integers separated by whitespace will indicate the primes to be calculated.
b) Your program should exit when passed 0 as an input.

Your program should provide output in the following format:

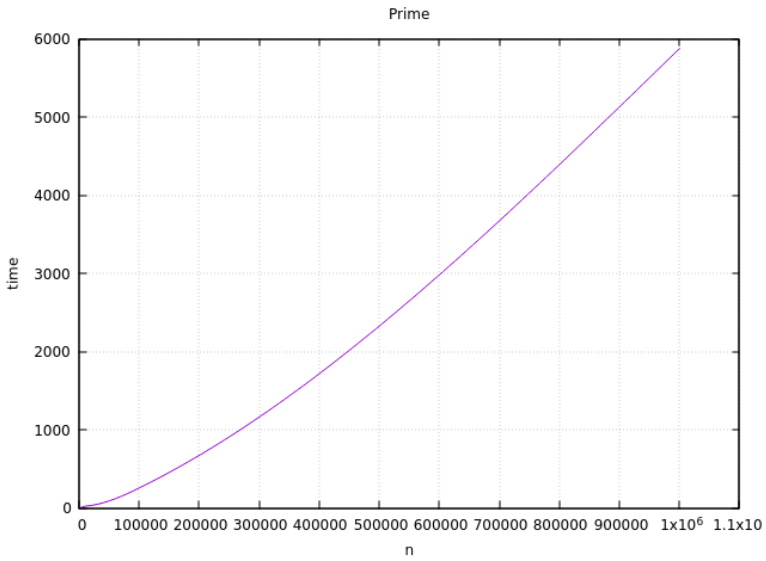
a) There should be one line for each prime calculated.
b) Each line should contain three numerical values separated by whitespace, representing *n*, the $n^{th}$ prime, the execution time in milliseconds.

Test that your program can calculate the $11^{th}$, the $101^{st}$, $1001^{st}$, $10001^{st}$, $100001^{st}$, and $1000001^{st}$ primes. Once you are sure that it is returning the correct values, set it up to time your code's execution in milliseconds. Save data related to execution times in a text file called Exercise1.out, and plot them on a simple graph.

Sample inputs and outputs:

| *Exercise1.in* | *Exercise1.out* |
|---|---|
| 11 101 1001 10001 100001 1000001 0 | 11 31 0<br>101 547 0<br>1001 7927 2<br>10001 104743 18<br>100001 1299721 235<br>1000001 15485867 5761 |

Sample graph:




Note: You may need to test additional cases in order to collect sufficient data to create smooth or representative plot.

You will submit both sample outputs and a line graph showing execution time in milliseconds on the Y-axis and input size on the X-axis. You can use LibreOffice Clac, Microsoft Excel, or a similar program to draw your graph. You have access to a simple but powerful plotting tool called GNUPlot on Unix-like systems like Bluenose, that will automatically create graphs for you. You may optionally choose to plot your points manually and scan your work for submission.

**Exercise2 (Collatz Sequence)**

The Collatz sequence of a positive integer $n$ is defined as follows:

a) Start with the integer $n$ (the starting number).
b) If the integer is even, divide it by 2 (integer division) (that is, $n \leftarrow n/2$)
c) If the integer is odd, multiply it by 3 and add 1. (that is, $n \leftarrow 3n + 1$)
d) Repeat the process until $n$ becomes 1.

For example, let $n = 5$. Then its Collatz sequence is:
$5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$
Since it takes six steps, the length of the Collatz sequence for starting number 5 is 6.

As another example, let $n = 13$. Then its Collatz sequence is:
$13 \rightarrow 40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$
The length of the Collatz sequence for starting number 13 is 10.

Although it is not proven yet, it is thought that all Collatz sequences end in 1.

For this exercise, you will to write a program to calculate which starting number less than or equal to an input value of $n$ produces the longest Collatz sequence. For example, if you were to find which starting number less than or equal to 5 produces the longest sequence, the answer would be 3 and the length of the sequence is 8. See below:

1: 1
2: $2 \rightarrow 1$
3: $3 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$
4: $4 \rightarrow 2 \rightarrow 1$
5: $5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$

As you can see, if $n$=5, the starting number with the longest sequence is 3 and the length of the longest sequence is 8.

Your program should accept input in the following format:

a) Positive integers separated by whitespace will indicate the starting values of $n$.
b) Your program should exit when passed 0 as an input.

Your program should provide output in the following format:

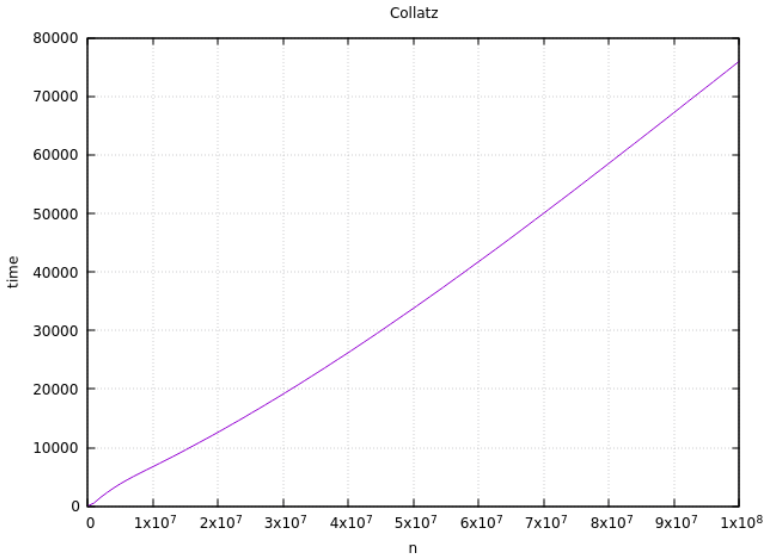
a) There should be one line for each value of $n$.
b) Each line should contain four numerical values separated by whitespace, representing starting value of $n$, the number producing the longest sequence, the length of that sequence, the execution time in milliseconds.

Test that your program works for *n* values of 5, 100, 1000, 10000, 100000, 1000000, and 10000000. Once you are sure that it is correctly identifying the longest sequences, set it up to time your code's execution in milliseconds. Save data related to execution times in a text file called Exercise2.out, and plot them on a simple graph.

Sample inputs and outputs:

| *Exercise2.in* | *Exercise2.out* |
|---|---|
| 5 100 1000 10000 100000 1000000 10000000 0 | 5 3 8 0<br>100 97 119 1<br>1000 871 179 3<br>10000 6171 262 12<br>100000 77031 351 54<br>1000000 837799 525 411<br>10000000 8400511 686 4685 |

Sample graph:




Note: You may need to test additional cases in order to collect sufficient data to create smooth or representative plot.

**Exercise 3 (Matrix Multiplication)**

For this exercise, you will to write a program to multiply two matrices. The header of the method you will write is provided below:

```
public static double[][] multiplyMatrix(double[][] a, double[][] b)
```

Assume that the two input matrices are square (that is, they are *n* X *n* matrices).
To multiply matrix *a* by matrix *b*, where *c* is the result matrix, use the formula:

```
a11  a12  a13        b11  b12  b13        c11  c12  c13
a21  a22  a23   *    b21  b22  b23    =   c21  c22  c23
a31  a32  a33        b31  b32  b33        c31  c32  c33

cij = ai1*b1j + ai2*b2j + ai3*b3j
```

Since we are only interested in the execution time, you may assume that all the elements of matrices *a* and *b* are identical. You can also assume that you are multiplying only square matrices (that is, number of rows == number of columns).

The template of the program with the main method is given below. It includes the code segment to get the execution time. Fill in the method to compute the product of the two matrices.

```
/*
MatrixMult Solution
*/

/**
This class tests the code for Lab2: Exercise3. It calls a method to
multiply two square matrices of size n x n, and prints information about
running time.
It expands upon a framework provided by Srini.
*/

//Multiplication of two square matrices of size n X n each
import java.util.*;

public class MatrixMult{
  public static void main(String[] args){

    Scanner sc = new Scanner(System.in);

    // declare sentinel and element variable
    int n;
    double num;

    while((n = sc.nextInt())>0){
      num = sc.nextDouble();

      double[][] matrix1 = new double[n][n];
      for (int i = 0; i<n; i++)
        Arrays.fill(matrix1[i], num);

      double[][] matrix2 = new double[n][n];
      for (int i = 0; i<n; i++)
        Arrays.fill(matrix2[i], num);

      // This is Srini's template for timing execution
      long startTime, endTime, executionTime;
      startTime = System.currentTimeMillis();
```

```
      double[][] resultMatrix = multiplyMatrix(matrix1, matrix2);

      endTime = System.currentTimeMillis();
      executionTime = endTime - startTime;

      //use for testing/debugging
      System.out.println("Execution time: " + executionTime + " millisecs");
    }
  }

  /** The method for multiplying two matrices */
  public static double[][] multiplyMatrix(double[][] m1, double[][] m2){
    //TODO
  }
}
```

Your program should accept input in the following format:

a) Each line will contain a pair of positive integers separated by whitespace, indicating the dimensions of the matrices to be multiplied (*n*) and the value to fill the matrices with (*num*).
b) Your program should exit when passed 0 as an input for *n*.

Your program should provide output in the following format:

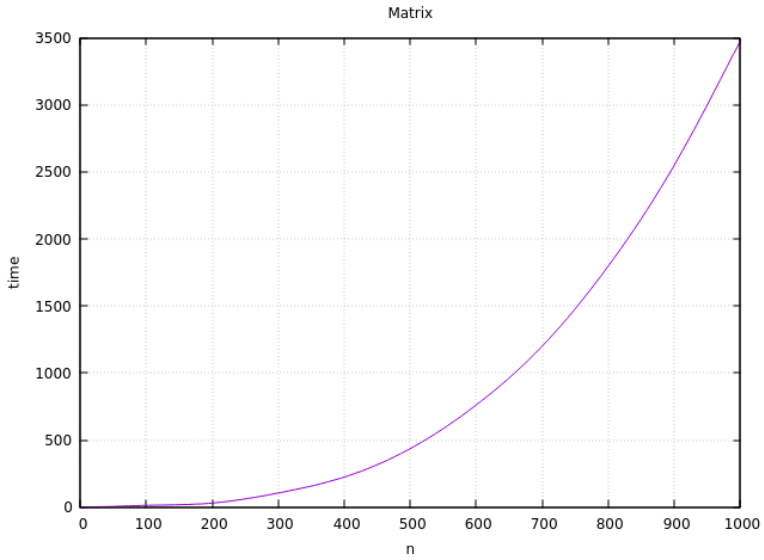
a) There should be one line for each value of *n*.
b) Each line should contain two numerical values separated by whitespace, representing the value of *n*, the execution time in milliseconds.

Test that your program works for *n* values of 3, 20, 100, 200, 300, 400, 500, 600, 700, 800, 900, and 1000. Once you are sure that it is correctly multiplying matrices, set it up to time your code's execution in milliseconds. Save data related to execution times in a text file called Exercise3.out, and plot them on a simple graph.

Sample inputs and outputs:

| *Exercise3.in* | *Exercise3.out* |
|---|---|
| 3 3<br>20 3<br>100 3<br>200 3<br>300 3<br>400 3<br>500 3<br>600 3<br>700 3<br>800 3<br>900 3<br>1000 3<br>0 | 3 0<br>20 1<br>100 13<br>200 30<br>300 105<br>400 224<br>500 436<br>600 762<br>700 1201<br>800 1798<br>900 2551<br>1000 3481 |

Sample graph:




Note: You may need to test additional cases in order to collect sufficient data to create smooth or representative plot.

Short Answer Question:

Finally, test your program with *n*=10000 and *num*=3. You will see an error message, like:

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
  at MatrixMult.multiplyMatrix(MatrixMult.java:45)
  at MatrixMult.main(MatrixMult.java:32)
```

Do a little research, figure out why this happens (use online resources). Prepare and submit a short answer with your solutions called Exercise3.txt. Make sure that you cite your references at the end of your answer.