CSCI 2110 Data Structures and Algorithms
Laboratory No. 3
Week of September 25 – 29, 2023

Due: Sunday, October 1, 11.59 PM

Algorithm Complexity Analysis

This is an experimental lab in which you will write and run three 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. Based on your experiments, you will draw a graph of the execution time vs. input size for each program.

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. Rather, you will be conducting the experiments to see how the run time functions grow as the 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 it very unlikely that two students will submit identical outputs.

Marking Scheme
Each exercise carries 10 points.
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 Requirements:
- No submission other than a single ZIP file will be accepted.
- You MUST SUBMIT .java files that are readable by the TA. If you submit files that are unreadable such as .class file, the lab will be marked 0.
- Please additionally comment out package specifiers.

What to submit:
- One ZIP file containing all source code (files with .java suffixes) and a text file with sample inputs and outputs, and graphs. The complete list of submission items is given at the end of this document.

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.

Late Submission Penalty: The lab is due on Sunday at 11.59 PM. Late submissions up to 5 hours (4.59 AM on Monday) will be accepted without penalty. After that, there will be a 10% late penalty per day on the mark obtained. For example, if you submit the lab on Monday at 12 noon and your score is 8/10, it will be reduced to 7.2/10. Submissions past two days (48 hours) after the grace submission time, that is, submission past 4.59 AM Wednesday will not be accepted.

Note: For each of 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 Exercise1. It calls a method to
calculate the nth prime and prints information about running time.
*/

import java.util.*;

public class Prime{
  public static void main(String[] args){
    //TO-DO
   }

  public static long nthPrime(long p){
    //TO-DO
  }
}
```

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. However, since this is an experimental lab to determine the execution time of an algorithm, any method to determine the prime will be accepted.

Input/output

Your program should accept input in the following format:

- Values denoting which prime to find separated by whitespace
- Entering 0 terminates the list

Your program should provide output in the following format:

- The n-value, the value of the nth prime, followed by the elapsed time.

Six Student Generated Test Cases

Test that your program can calculate the following primes:

- 11^{th},
- 101^{st},
- 1001^{st},
- 10001^{st},
- 100001^{st},
- 1000001^{st}

Do not test all 6 cases at once. Start with the first 2, then 3, etc. If your program hangs on larger values,note this in your test file.

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 **Exercise1out.txt**, and plot them on a simple graph.

Sample inputs and outputs:

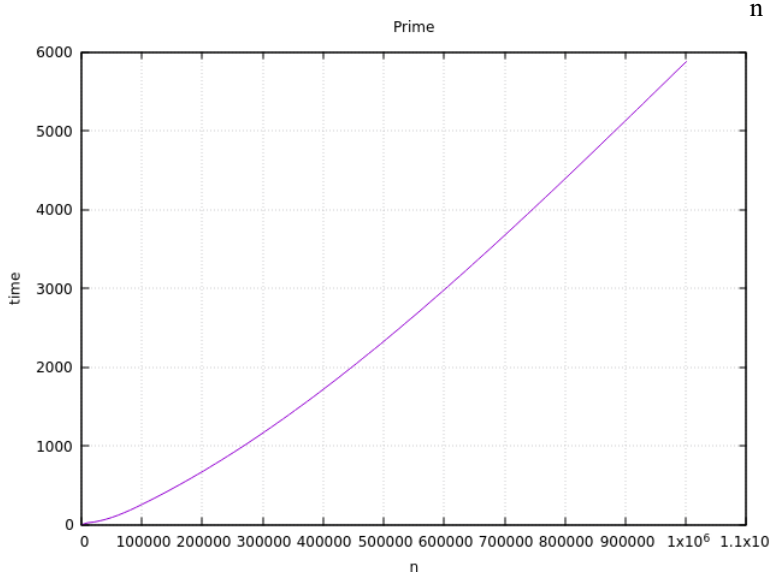
Input	*Output*
11	11 **31** 0
101	101 **547** 0
1001	1001 **7927** 2
10001	10001 **104743** 18
100001	100001 **1299721** 235
1000001	1000001 **15485867** 5761
0	

n

Value of the nth prime

Execution time

Sample Graph



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.

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

Exercise 2 (Matrix Multiplication)

For this exercise, you will write a program to multiply two matrices. See the given java file ***MatrixMult.java.*** You may use this as starter code. It contains commented out print methods to help debug your multiplier.

Assume that the two input matrices are square (that is, they are *n* X *n* matrices, i.e. n rows by ncolumns).

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

For example, the upper left corner of matrix c (row 1 column 1) would be calculated from a11*b11+a12*b21+a13*b31. Remember that in Java, arrays start at index 0.

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, numberof rows == number of columns).

Input/output

Your program should accept input in the following format:

- 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*). Inthe first sample below, we start off with the input 3 3, indicating a 3 x 3 matrix filled with the value 3 in every cell.

Your program should provide output in the following format:

- The size of the two matrices multiplied and the execution time in milliseconds.

The skeleton of the solution should look something like the one given below.

```
/**
This class tests the code for Exercise 2. 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){
    //TO-DO

    }

    /* The method for multiplying two matrices */
```

```
     public static double[][] multiplyMatrix(double[][] m1, double[][] m2){
    //TO-DO
    }
}
```

Twelve Student Generated Test Cases

Test that your program works for the following inputs:

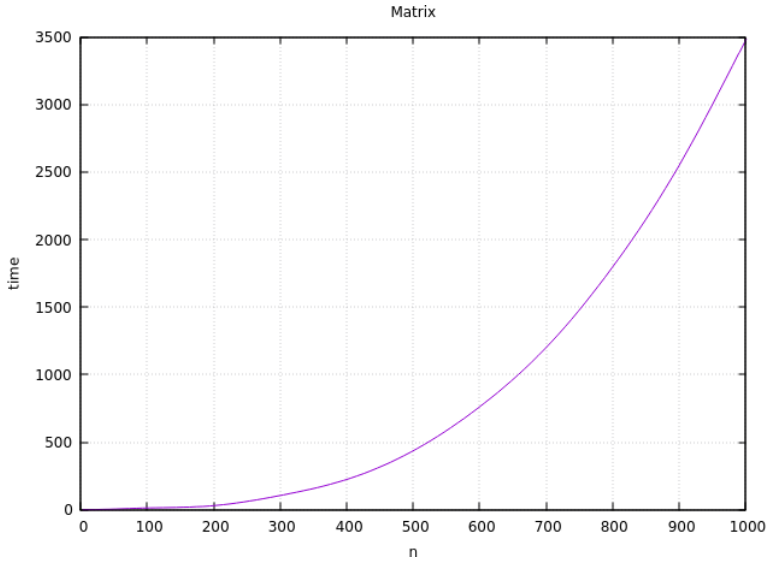
- 3 3
- 20 3
- 100 3
- 200 3
- 300 3
- 400 3
- 500 3
- 600 3
- 700 3
- 800 3
- 900 3
- 1000 3

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 **Exercise2out.txt**, and plot them on a simple graph. You may test your cases one at a time.

Sample inputs and outputs:

Input	*Output*
3 3 20 3 100 3 200 3 300 3 400 3 500 3 600 3 700 3 800 3 900 3 1000 3	Size:3 Time: 0 ms Size: 20 Time: 0 ms Size:100 Time:13 ms Size:200 Time:30 ms Size:300 Time:105 ms Size:400 Time:224 ms Size:500 Time:436 ms Size:600 Time:762 ms Size:700 Time:1201 ms Size:800 Time:1798 ms Size:900 Time:2551 ms Size:1000 Time:3481 ms

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 spaceat 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, call it Exercise2.txt. Make sure that you cite your references at the end of your answer.

Exercise 3: In this exercise you will write a small program for which the execution time grows exponentially. Your program accepts an integer n and generates all the binary numbers for the integers 0 to 2^n-1. For example,

If n = 2, then 2^n is 4, so your program generates binary numbers for the integers 0, 1, 2, and 3, which are 00, 01, 10 and 11, respectively.

Thus, your program generates
00
01
10
11

If n=3, then 2^n is 8, so your program generates binary numbers for the integers 0,1,2,3,4,5,6, and 7.
Your program generates
000
001
010
011
100
101
110
111

```java
/*
Binary Number Generation
*/

/**
This class tests the code for Lab2: Exercise3. It calls a method that accepts a positive
integer n and
generates binary numbers between 0 and 2^n -1. The main method prints information about
running time.
*/

import java.util.*;

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

  public static void generateBinary(int n){
    //TODO
  }
}
```

Note: This is actually a very simple program. The generateBinary method has a for loop to go from 0 to 2^n-1.
You can use the built-in method
`Integer.toBinaryString(x)` to convert an integer x into its binary equivalent. The method will return a String.
Note that you are not printing these binary numbers – just generating them to get the execution times.

`Also (int)(Math.pow(a,b))` gives a^b as an integer.

A sample dialog of your program is:

```
Enter a positive integer: 3
The execution time to generate binary numbers from 0 to 7 is 0.0 ms
```

Another example:

```
Enter a positive integer: 10
The execution time to generate binary numbers from 0 to 1023 is 2 ms
```

Test your program for input values of 10, 12, 14, 16, …., 28, 30, 32, 34, etc.
You will see that the program takes a long time even for n = 28 or 30. Collect enough values and plot a graph of n vs. execution time, similar to the graphs in Exercise 1 and 2.

Submission items

One zip file containing the following files:
1. Prime.java
2. MatrixMult.java
3. Binary.java
4. A word doc or any other document containing input and output values from your experiments and graphs as in the examples given