CSCI 2110- Data Structures and Algorithms
Laboratory No. 2
Week of September 18th, 2017

Algorithm Complexity Analysis

This is an "experimental" lab in which you will experiment with codes to understand algorithm complexities. The objective is to write programs with code snippets of different time complexities and test how long each code snippet takes to run on your machine for different values of input n.

Although finding the "wall-clock" time or "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, it is an experiment to see how the function grows for different values of the input size 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();

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

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

//display executionTime
```

The above segment will give the time for executing the code segment in milliseconds.

Marking Scheme
Each exercise carries 10 points. Your final score will be scaled down to a value out of 10.
Exercises 1 to 4
Working code, Outputs included, Efficient, Good basic comments included: 10/10
No comments: subtract one point
Unnecessarily inefficient: subtract one point
No outputs: subtract two points
Code not working: subtract up to six points depending upon how many methods are incorrect.

Exercise 5 *carries 10 points (2.5 for each correct O notation).*

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. Submissions are pretty straightforward. Instructions will be also be given in the first lab.*
Deadline for submission: Sunday, September 24th, 2017 at 11.55 p.m. (five minutes before midnight).

What to submit*:*
A zip file containing all source codes, namely,
Prime.java, CollatzSequence.java, MatrixMult.java, Exponential.java, and a text/pdf document containing the sample outputs and graphs.

Exercise 1.

A prime number, as you know, 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. Therefore, if you are asked to find the 6^{th} prime number, the answer is 13.

In this exercise, you are to write a program to find the 11^{th}, the 101^{st}, 1001^{st}, 10001^{st} prime numbers and determine how long your program takes to find each of those primes.

Note your results in a table as shown below.

Table 1: nth Prime Number

n	n^{th} prime number	Execution time (millisecs)
11		
101		
1001		
10001		

Using the above table, draw a graph of the evaluation time (y axis) vs input size n. You can use a simple utility like Microsoft Excel to draw the graph or you can also plot it manually and scan it.

Notes

1. Use long data type instead of int to fit the large prime numbers.
2. If you use the naïve way to determine if a number x is prime (that is, test if every integer less than x is a factor or not), then your program will take a long time to calculate the 10001th prime. There are smarter ways to determine the prime that will reduce the execution time.

Source: This question has been adapted from Project Euler (projecteuler.net and programmingbydoing.com)

Exercise 2:
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← n/2)
c) If the integer is odd, multiply it by 3 and add 1. (that is, n ← 3n + 1)
d) Repeat the process until n becomes 1.

For example, let n = 5. Then its Collatz sequence is:
5 → 16 → 8 → 4 → 2 → 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 → 40 → 20 → 10 → 5 → 16 → 8 → 4 → 2 → 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.

In this exercise, you are to write a program to find which starting number less than or equal to N produces the longest 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 → 1
3: 3 → 10→ 5 → 16→ 8 → 4 → 2 → 1
4: 4 → 2→ 1
5: 5→ 16→8→ 4 → 2 → 1

For each run of the program, change the value of N, find the starting number, the length of the longest seqeuence and the execution time and record your answers in a table as shown below.

Table 2: Collatz Sequences

N	Starting Number with longest Collatz Sequence between 1 and N	Length of the longest sequence	Execution time (millisecs)
5	3	8	0
100			
1000			
10000			
100000			
1000000			
10000000			

You will notice that as N increases, it becomes more and more computationally intensive.
Plot a graph of N vs. execution time.

Source: This question has been adapted from Project Euler (projecteuler.net and programmingbydoing.com)

Exercise 3.
Write a method to multiply two matrices. The header of the method is as follows:

```
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     X    b21  b22  b23     =    c21  c22  c23
a31  a32  a33          b31  b32  b33          c31  c32  c33
```

where `cij = ai1Xb1j + ai2Xb2j + ai3Xb3j`

Since we are only interested in the execution time, you may assume that all the elements of matrices a and b are identical. A sample screen dialog and output is given below:

```
Enter the size of each matrix: 100
Enter the matrix element(all elements of the matrices are assumed to be the
same): 3.5
Execution time: 16 millisecs
```

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.

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

public class MatrixMult {
  /** Main method */
  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
      int n;
```

```
    double num;

    System.out.print("Enter the size of each matrix: ");
    n = keyboard.nextInt();
    System.out.println("Enter the matrix element");
    System.out.print("All elements of the matrices are assumed to be the
same: ");
    num = keyboard.nextDouble();

    double[][] matrix1 = new double[n][n];
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        matrix1[i][j] = num;

    double[][] matrix2 = new double[n][n];
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        matrix2[i][j] = num;


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

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

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

      System.out.println("Execution time: " + executionTime + "
millisecs");

  }

  /** The method for multiplying two matrices */
  public static double[][] multiplyMatrix(double[][] m1, double[][] m2)
  {
      //include your code here
  }


}
```

Test the execution time for different values of n (say n = 100, 200, 300, …. 1000)

Record your values in a table like the one shown below:
Table 3: Matrix Multiplication

Size of matrix (n)	Execution time (millisecs)

Draw a graph of execution time vs n.

Again, use appropriate data sizes in order to generate the curve.

You may notice that if n exceeds 1500 or 2000, the execution time takes longer and longer. You may also get an error such as:

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

Explore why this happens (you can use web resources for this) and write a brief answer.

Exercise 4. The third code has an exponential time complexity (2^n), which means that it grows very fast even for small values of n.

Write a program **Exponential.java** that generates 2 power n binary numbers for a given value of n.
For example, if n is 3, the binary numbers that are generated are 000, 001, 010, 011, 100, 101, 110, and 111 (which are equivalent to 0 to 7 in decimal).

A sample screen dialog and output is given below:

```
Enter the value of n: 17
Execution time to generate 2^17 binary numbers: 32 millisecs
```

Note:
`(int)Math.pow(2,n)` gives 2 power n as an integer.

```
String sb = Integer.toBinaryString(i);
```

converts an integer i into a binary number.

Note: You need not print the binary numbers – you just need to generate them and measure the execution time.

Therefore, all you need to do in the program is to call the above statement
`String sb = Integer.toBinaryString(i);`
2 power n times using a for loop (the loop goes from 0 to 2^n - 1

Test the execution time for different values of n (say n = 10, 11, 15, 17, 20, etc.). Word of caution: The program will take a really long time to complete for values of n even larger than 20 or 30. If the program doesn't end in a few minutes, just record that it "hangs".

Record your values in a table similar to the one given below.

Value of n	Execution Time to generate 2 power n binary numbers (millisecs)

Draw the graph of execution vs. n.

Exercise 5: Analyze your codes using the rules of thumb discussed in the lecture and determine the orders of complexity for each of the programs in exercises 1 to 4. (Write this in a simple text document or handwritten and scanned as pdf and include it to your zip file).