Timing Big Integer Arithmetic

Due Date:

This assignment is due Wednesday, 2022-09-21 *by the beginning of lecture*.

Before Coding

Plotting

Read "Instructions on Plotting and Timing".[1] To be sure that you understand what you read, see if you can answer the following questions:

- How should the x-values (independent data values) be spaced?
- Approximately what range should the y-(dependent) values cover?
- Suppose that you meansure time values that lie between 0.00218 s and 0.01062 s, what units should you use to report the results?
- Where should these units be reported?

Plotting with `Excel™`

If you use Microsoft Excelto plot your data, know (or learn) how to coerce Excel into producing charts suitable for scientific use. You likely want to

- use X-Y (Scatter) plots,
- add axis titles, and minor grid-lines,
- ensure that the axes cross at (0.0).

`BigIntegers`

Read about the `java.math.BigInteger` class. One important fact about `BigIntegers` is that you *cannot* treat the running time of operations as constants independent of the size of the integers. Learn

- how many constructors this class has,

[1] Can be found at `https://web.unbc.ca/~casper/Semesters/2022-05F/200.php` under "Handouts".

- which constructor is likely most appropriate for constructing random numbers,
- how to add two `BigIntegers`,
- how to multiply two `BigIntegers`.

Random number generation

A couple of the constructors for the `BigInteger` class take random number generators as arguments. Use an object from `java.util.Random` as your random number generator. It is good programming practice to create only *one* random number generator for the whole program.

`System.nanoTime()`

Read the Oractle doucmentation for `System.nanoTime()`. Be sure to understand

- What is the return type of this method?
- (How long before wrap-around?)
- What does the documentation say about precision versus resolution?

Coding

Stopwatches

Create a `StopWatch` class that has the methods shown in Figure 1 on the following page. Note that the action methods have return type `StopWatch` rather than CPSC 101-style return type `void`. This alternate return style allows one to write code like

```
StopWatch sx = new StopWatch().reset().start() ;
```

Use `System.nanoTime()` to provide the timing information. **Your stopwatch must work as described**. In particular it should be possible to repeatedly stop and start the stopwatch without resetting the stopwatch.

Code like

```
1   int nTimes = 0 ;
2   watch.stop().reset() ;
3   while (watch.elapsed() < 0.01 || nTimes<2)
4       {
5       // setup a problem
6       watch.start();
```

```
7       // run a problem
8       watch.stop() ; ++nTimes ;
9       }
10   return watch.elapsed() / nTimes ;
```

should produce an average running time. Note that this code relies on stopping and starting the watch not resetting the total elapsed time. Also note that addtional variables to keep track of start, stop, and elapsed times are not necessary. Finally, note how the code carefully only times the actual running times for problems.

Method	*Meaning*
attributes	
`public double elapsed()`	Returns the elapsed CPU time *in seconds* at the time of the call.
`public boolean isRunning()`	true if and only if start has been called more recently than stop.
actions	
`public StopWatch start()`	Starts the stop watch. Has no effect if the stop watch is already started. Does not reset the time. Returns `this`.
`public StopWatch stop()`	Stops the stop watch. Has no effect if the stop watch is already stopped. Does not reset the time. Returns `this`.
`public StopWatch reset()`	Resets the elapsed time to zero. Neither starts nor stops the stop watch. Returns `this`.
Constructors	
`public StopWatch()`	Creates a new `StopWatch`, which is initially stopped with zero elapsed time.

Figure 1: Properties and Actions of `StopWatches`

Addition and Multiplication

⇒ Write two programs (one is ok) that uses your `StopWatch` clas from above, and the `java.math.BigInteger` class to measure the average time for the following operations:

- Measure the *average* time to add two random n-bit big integers, where n varies over an appropriate range.
- Measure the *average* time to multiply two random n-bit big integers, where n varies over an appropriate range.

Your programs should write comma separated values to `System.out` (or possibly a file). For instance, output might look like

```
Size, Time(ms)
 60000,  4.185
120000,  3.683
180000,  6.379
270000, 13.774   ...
```

If you write to `System.out`, it should be possible to produce output suitable for Excel, by running a command like

```
java TestMultiplication 60000 > multData.csv
```

Comments on Implementation

1. You may need to some preliminary tests to find a good problem-size range.
2. For small problem sizes, the running time that you are measuring may be small compared to the resolution of `System.nanoseconds()`. To accommodate this, you may need to use techniques like starting your stopwatch, running multiple calculations, and then stopping your stop-watch.

   On my laptop multiplication of 30 000 bit numbers takes about 320 μs, but the standard deviation of the values is nearly that large. Using an increment of 100 000 git, and looping for each test for a total of at least 0.5 s (then using the average) seems to give good results.

   On my laptop addition of $10,000\,000$ (10^7) bit numbers takes about 500 μs, and using timing averaged over a half second seems to give good results.
3. Older versions of Java have less efficient algorithms for the `java.math.BigInteger` class. If you have access to an old version of Java, please use it. The data are more interesting.

⇒ Plot your data. Provide separate graphs for the addition data and the multiplication data. Addition is likely to be much faster than multiplication, so you likely need different data ranges.

⇒ Comment on the mathematical nature of the running times. For instance, if you increase the problem size by a factor of 4, does the running time increase by the same factor? What do you suspect the functional relation between problem size and running time is?

Hand In

At the time this lab was written, the mechanics of Moodle submission were not clear. Please submit the following items:

- all JAVA code (*e.g.*, `Stopwatch.java`, `TestAddition.java`, `TestMultiplication.java`, and any other cude that you write).

  These may be submitted as individual `.java` files, or as a `.jar` file *containing* the `.java` source code.

  In your code, add comments that answer the following questions:

  ⋆ why you chose the constructor for `BigIntegers` that you did, and the URL of the Oracle Java documentation for the `BigInteger` class.
  ⋆ Comment on the accuracy of your stopwatch.

- Two clearly identified charts showing your data, stored in a graphical format (`.pdf`, `.png`, ...).

- The data produced by your programs and used to create the charts. These may be uploaded as plain text files, or enclosed in an EXCEL spreadsheet. If you submit an EXCEL spreadsheet, you must still submit the charts separately.

- Your guesses, supported by your data, as to how the run times grow with problem size for big integer addition and multiplication.

Note the `Stopwatch` code will marked for completeness and correctness, regardless of the extent to which you use your `Stopwatch` in your timing code.