600.107 Introductory Programming in Java, Summer 2015
Homework 5
Due: Monday, July 13th at 11:59pm

Please review the **Homework Guidelines** handout for this course before you begin.

Restriction: For all tasks in this assignment, you are NOT permitted to use arrays or ArrayLists.

Repetition Statements: "while", "for", or "do-while"- type statements MAY be used in this exercise, but you are NOT permitted to use `break;` or `continue;` statements except that you may use `break;` inside a case within a `switch` statement.

Background for Task 1: A *linear congruential generator* is the name for one method for generating a sequence of integers $x_0, x_1, x_2, \ldots$ that appear random. For this assignment, assume a generator is specified by 4 values:

- the multiplicative constant a
- the additive constant b
- the initial value (or seed) x_0
- the modulus value N

Starting from any value in the sequence x_i where $i \geq 0$, the next value in the sequence (called x_{i+1}) is computed from the formula

$$x_{i+1} = (ax_i + b) \bmod N.$$

Note that the notation *expr* $\bmod N$ indicates calculating the value of the expression *expr*, then dividing it by N and keeping the remainder. In Java, we represent this operation using the $\%$ symbol.

For example, using a linear congruential generator with $a = 17$, $b = 4$, $x_0 = 1$, and $N = 32$, the sequence of values generated is 1, 21, 9, 29, 17, 5, and so on. These values represent x_0, x_1, x_2, up through x_5, but the sequence continues indefinitely.

Task 1. Write a Java class called `LinConGen` which asks the user to input from the keyboard the four integer values a, b, x_0, and N, and then computes the first N values (including the value x_0) generated by the specified linear congruential generator, and writes those values to an output file called `lcg.txt`. Format your output so that exactly 16 values are printed on each line in the file except perhaps the final line, where 16 or fewer will appear. In addition, use printf so that each number is set to take up exactly 4 places, with no spaces in between. For Task 1 only, you can assume that N is no greater than 1000, so that the output will always line up nicely. **You are required to write and use in your code a helper method named `nextValue` which has arguments representing a value x_i, as well as a, b, and N, and which calculates and returns the next value in the sequence. You may not use any "global" variables declared outside a method in your code; this method must accept any needed information via its formal parameters. You are free to define additional helper methods if you wish.** See the three separate sample executions below, with user input shown in bold, and the contents of the output file shown immediately below the screen dialog with the user.

```
This program calculates values generated by a linear congruential generator.
Separating values by spaces, please enter the multiplicative constant a,
additive constant b, seed value x_0, and modulus N.
```
89 17 2 101

```
Writing the first 101 values of the sequence to file lcg.txt.
```

After the execution shown above, the file lcg.txt will contain the following 7 lines:

```
  2  94   0  17  15  39  54  76  14  51  11  87  84  19  92  24
 32  37  78  91  36  90  48  47  59  16  27  97  65  45  83  31
 49  35   1   5  58  28  85   7  34  13  63  69  98  53  88  72
 62  81  55  64  57  40  42  18   3  82  43   6  46  71  74  38
 66  33  25  20  80  67  21  68   9  10  99  41  30  61  93  12
 75  26   8  22  56  52 100  29  73  50  23  44  95  89  60   4
 70  86  96  77   2
```

```
This program calculates values generated by a linear congruential generator.
Separating values by spaces, please enter the multiplicative constant a,
additive constant b, seed value x_0, and modulus N.
15 7 1 38
Writing the first 38 values of the sequence to file lcg.txt.
```

After the execution shown above, the file lcg.txt will contain the following 3 lines:

```
  1  22  33   8  13  12  35   0   7  36  15   4  29  24  25   2
 37  30   1  22  33   8  13  12  35   0   7  36  15   4  29  24
 25   2  37  30   1  22
```

```
This program calculates values generated by a linear congruential generator.
Separating values by spaces, please enter the multiplicative constant a,
additive constant b, seed value x_0, and modulus N.
2 1 0 10
Writing the first 10 values of the sequence to file lcg.txt.
```

After the execution shown above, the file lcg.txt will contain the following 1 line:

```
  0   1   3   7   5   1   3   7   5   1
```

Additional background for Task 2: The set of the first 100 values shown in the example above where $N = 101$ contains no duplicate values. However, the final value (x_{100}), the 2, is a repeat - it was actually the seed value x_0. This means that the values of x_{101}, x_{102}, and x_{103} (and so on) are completely predictable based solely on earlier values of the sequence; they'll match the values of x_1, x_2, and x_3 (and so on) exactly. This means we don't even need to know the parameters; once we find a duplicate value in the sequence, later values will not appear to be very random at all!

In fact, note that none of these linear congruential generator sequences are actually random, but (for proper choices of the parameters) they exhibit many properties of random sequences. For small values of the parameters, the non-randomness is particularly evident: whenever we generate a value that we have seen before, we enter into a *cycle*, where we continually generate the same subsequence over and over again.

Since there are only N different possibilities for sequence values, we must always enter such a cycle with a linear congruential generator if we generate more than N values of the sequence. But some parameter choices yield repeats even more quickly. In the final sample above where $N = 10$, for example, the generator's cycle is $1 - 3 - 7 - 5 - 1$. We say this is a cycle of length 4. The generator in the middle example above has a cycle of length 18 (the value 1 is the first one repeated), and in the first sample, as we saw above, the generator has a cycle length of 100. Note that not all cycles begin with the first item in the sequence.

In order to obtain a sequence whose values appear to be more random, we hope to avoid short cycles. There are facts from number theory that tell us which choices for parameters yield maximally-long cycles, but another way to check the length of a cycle given certain parameters is using an algorithm due to Robert W. Floyd.

Floyd's algorithm for cycle length detection consists of two phases. Phase I involves finding some value on the cycle, and Phase 2 computes the cycle length. Each phase is described in more detail below.

Phase I. To find a value on the cycle, Floyd suggested using the following strategy: run two versions of the generator concurrently from the beginning, one advancing twice as fast as the other. If you are familiar with the Aesop fable called *The Tortoise and The Hare*, you'll understand why Floyd's algorithm is sometimes called the tortoise-and-hare approach. The clever observation needed here is that eventually, both versions of the generator are guaranteed to have the same value at the same time. When this occurs, we can be sure we have found a value on the cycle.

A helpful illustration of the Phase I process applied to a sequence with the cycle $6-3-1-6$ appears at `http://en.wikipedia.org/wiki/Cycle_detection#mediaviewer/File:Tortoise_and_hare_algorithm.svg`. Within the illustration, notice how the slow-moving tortoise advances forward only one square between pictures, while the speedier hare always advances two steps. And the process terminates when they reach squares containing equal values.

Phase II. Once we determine some sequence value v on the cycle during Phase I, we move to the second phase, where we simply continue advancing from value v onward in the sequence until we detect v again. During this stage, counting our steps allows us to determine the length of the cycle, and the algorithm is complete.

Task 2. Write a Java class called `CycleLength` that implements Floyd's algorithm for cycle length detection. Specifically, after asking the user to input from the keyboard the four values a, b, x_0, and N as in Task 1, the program should simply output to the screen the length of the cycle of the linear congruential generator that those parameters specify.

There are no input files or output files used in this task, and you may not assume anything about N other than that it is positive. In fact, your program should allow very large values of N and the other parameters. For this part, you should use the Java *long* type rather than the *int* type for your calculations. **You are required to copy over your helper method from Task 1 and modify it here to accept and return long types, and then use it in your code.** Three separate sample executions are shown below. You should minimally test your code on these values, but check others as well.

```
This program calculates the cycle length of a linear congruential generator.
Separating values by spaces, please enter the multiplicative constant a,
additive constant b, seed value x_0, and modulus N.
89 17 2 101
The cycle length is 100.

This program calculates the cycle length of a linear congruential generator.
Separating values by spaces, please enter the multiplicative constant a,
additive constant b, seed value x_0, and modulus N.
123 456 789 1000000
The cycle length is 50000.

This program calculates the cycle length of a linear congruential generator.
Separating values by spaces, please enter the multiplicative constant a,
additive constant b, seed value x_0, and modulus N.
78 60 89 129024
The cycle length is 7.
```

Task 3. An n-digit integer number is said to be *polydivisible* if the leftmost digit is divisible by 1, the leftmost two digits are divisible by 2, the leftmost three digits are divisible by 3, and so on, and the leftmost n digits (which are the number itself) are divisible by n. For example, the number 42325 is a 5-digit polydivisible number because 4 is divisible by 1 (of course!), 42 is divisible by 2, 423 is divisible by 3, 4232 is divisible by 4 and 42325 is divisible by 5.

Write a Java class called `Polydivisible` that first asks the user to enter the name of an input file. The input file will contain an unknown number of integers separated by whitespace (some combination of spaces, tabs, and newline characters). The program should then read all values listed in the file, and create an output file containing only the values from the input file which are polydivisible, each one on a separate line in the output file. The output file's name should be the same as the input file's name, but with `.out` appended to it. If no input file values are polydivisible, the output file should be completely empty. Be sure to explicitly close both input and output files when you finish using them. Finally, the program should output to the screen a summary message, indicating the total number of values processed, and the number which were polydivisible. Nothing else should be output to the screen.

Suppose the file `poly1.txt` contains the values shown to the right in the top box. Below is a sample execution, with user input shown in bold. The contents of the output file `poly1.txt.out`, which is generated by this execution, are shown to the right in the bottom box.

```
Please enter the name of the input file:  poly1.txt
Read 10 values from input file poly1.txt.
Wrote 7 of them to output file poly1.txt.out.
```

```
12345
4232
8765
42325
102
1
65
10
360852885
12
```

`poly1.txt`

```
4232
42325
102
1
10
360852885
12
```

`poly1.txt.out`

In your solution, you are required to write and make use of the following helper method named `isPolydivisible` which has a single `long` integer as its only parameter, and returns a boolean indicating a decision about whether or not that single integer is polydivisible. Test that thoroughly before moving on.

```
/**
    Check if the specified long integer is
    polydivisible or not.

    @param num   the number to check
    @return      true exactly when argument
                 is polydivisble
*/
public static boolean isPolydivisible(long num)
```

Turn-in. Before the due date listed at the top of this file, please submit via the course Blackboard site one complete `.zip` file named `HW5-jhed.zip` (where you replace 'jhed' with your own personal JHED) containing all `.java` files required for the above tasks. Be sure that all submitted code compiles! If you were unable to complete a task, please include as part of your `.zip` submission a text file named `README` explaining anything you'd like the graders to know.

Need to fix something that you already submitted? At any point up until the due date for this assignment, you may re-submit a complete zip file. The submission most recently received is the one that will be graded. Any earlier submissions will be ignored, so please include *all* files in each submission.