**CMPUT 331, Fall 2020, Assignment 3**

*Before beginning work on this assignment, carefully read the Assignment Submission Specifications posted on eClass.*

You will submit four files for this assignment: "a3.pdf" or "a3.txt" for problems 1 and 3, "a3p2.py" for problem 2, "a3p4.py" for problem 4, and a README file.

Some problems in this assignment require you to solve for key variables (e.g. $a$ and $b$), or to write code which does so. Consult your textbook and the lecture videos on the details of modular arithmetic and computing a modular inverse. **No credit will be given for "brute force" solutions, i.e. solutions which simply try various possible values until a solution is found. To receive credit, you, and your code, must find the key variables in an efficient, mathematical way.**

## Problem 1

Short answer: Consider a text made up of symbols from a symbol set containing 71 elements, each corresponding to a unique integer from 0 to 70, encrypted with the affine cipher, with keys $a$ and $b$ encrypting each plaintext character $p$ according to the formula $p \cdot a + b \pmod{71}$. Suppose we know that '52' is enciphered as '6', '20' is enciphered as '51', and '4' is enciphered as '38'. Find the keys $a$ and $b$ mod 71. Include your solution, including all relevant work and explanation, in your a3.pdf.

## Problem 2

Random numbers are an integral component of cryptography. However, truly random numbers are difficult to generate efficiently. Pseudorandom number generators (PRNGs) are algorithms which produce sequences of *pseudorandom* numbers, which appear random, but which, with some extra (typically hidden) information, can be predicted. One algorithm which works as such a PRNG is the *linear congruential generator* (LCG), which uses a recurrence relation similar to the affine cipher's encryption function to generate its pseudorandom numbers:

$$R_{i+1} = (aR_i + b) \pmod{m} \qquad i \geq 0$$

where $a$, $b$, $m$, and $R_0$ are chosen in advance. For reference, $a$ and $b$ are called the "keys", $m$ is called the "modulus", and $R_0$ is called the "seed". For example, if we run a LCG with $a = 3$, $b = 5$, $m = 16$ and $R_0 = 6$, then $R_1 = 7$ since $(3 \cdot 6 + 5) = 23 \equiv 7 \pmod{16}$, and $R_2 = 10$ since $(3 \cdot 7 + 5) = 26 \equiv 10 \pmod{16}$.

Modify the file "a3p2.py" so that it contains a function "lcg($a$, $b$, $m$, $r0$)", where $a$, $b$ and $m$ are as above, and $r0$ is the seed value $R_0$, which **returns a list** containing $R_1, R_2, \ldots, R_{10}$.

For example, lcg(22695477, 1, 2**32, 42) – that is, a call to lcg with parameters $a = 22695477$, $b = 1$, $m = 2^{32}$, and a seed $R_0 = 42$ – should return the following list:

```
[953210035, 3724055312, 1961185873, 1409857734, 3384186111,
3302525644, 1389814845, 444192418, 2979187595, 2537979336]
```

## Problem 3

Short answer: An LCG with $m = 475$ generates the numbers $R_1 = 23$, $R_2 = 20$, and $R_3 = 436$. Determine the values of $a$, $b$, $R_0$, and $R_4$. Include the values of these **four** variables, with all relevant work and explanation for how you found them, in your a3.pdf or a3.txt.

## Problem 4

In problem 3, you were able to "hack" an LCG by obtaining the values of $a$ and $b$ and thus predict all of its future output (you predicted $R_4$, but of course, since $a$ and $b$ are fixed, you could predict any quantity of future values). In this problem, you will automate this process.

Modify the file "a3p4.py" so that it contains a function "crack_lcg($m$, $r1$, $r2$, $r3$)", where $m$ is a positive integer, and $r1$, $r2$, and $r3$ are integers between 0 and $m-1$, inclusive. This function **must return a list** $[a, b]$, where $a$ and $b$ are the keys for an LCG, with modulus $m$, which outputs $r1$, $r2$, and $r3$ as its first three random numbers (i.e. $r1 = R_1$, $r2 = R_2$, and $r3 = R_3$).

Note that in some cases, there may not be a solution (this includes the case where multiple different values work for a and b). In that case, your program should output the list $[0, 0]$ (i.e. report $a = b = 0$).

Hint: While not advisable in practice, it is perfectly valid for an LCG to have $b = 0$, and still have $a \neq 0$. Your crack_lcg function *must be able to crack LCGs with* $b = 0$, $a \neq 0$, still returning the two-element list $[a, b]$.