CMPUT 331, Winter 2025, Assignment 3

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

Introduction

The Affine Cipher is an improvement over some previous ciphers in this course, as its high number of potential keys allows for greater security. They display a number of important cryptographic concepts at work, as some of the same transformations used for Affine Ciphers are used in random number generation, which is important in the broader context of cryptography.

In this assignment, you will study the Affine Cipher and random number generations, including writing code to implement and crack them yourself. You may find this document about solving systems of equations in modular arithmetic useful in cracking key variables. **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. Please do not use print in your code!**

You will produce five files for this assignment:

- **a3p2.py, a3p4.py**: python solutions to problems 2 and 4 respectively
- **a3.txt**: written responses to problems 1 and 3
- **a3.pdf**: Showing your work on problems 1 and 3
- a README file (**README.txt** or **README.pdf**)

Submit your files in a zip file named 331-as-3.zip.

Problem 1 (2 Marks)

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, and your final answer in a3.txt.

Note: Please strictly follow the template provided in a3.txt. Any change in the template might result in a potential mark deduction.

Problem 2 (2 Marks)

Random numbers are an integral component of cryptography. Alice knows random numbers are difficult to generate efficiently. In order to securely communicate with Bob, she is trying to use pseudorandom number generators (PRNGs), algorithms which produce sequences of *pseudorandom* numbers, which appear random, but which, with some extra (typically hidden) information, can be predicted. Alice found an algorithm which uses a recurrence relation similar to the affine cipher's encryption function:

$$R_{i+2} = (aR_{i+1} + bR_i + c) \pmod{m} \qquad i \geq 0$$

where a, b, c, m, R_1, and R_0 are chosen in advance. For reference, a, b, and c are called the "keys", m is called the "modulus", and R_0 and R_1 are called "seeds". We will assume that m is a prime number. For example, if we run this algorithm with $a = 3$, $b = 5$, $c = 9$, $m = 17$, $R_0 = 11$ and $R_1 = 6$, then $R_2 = 14$ since $(3 \cdot 6 + 5 \cdot 11 + 9) = 82 = 14 \pmod{17}$, and $R_3 = 13$ since $(3 \cdot 14 + 5 \cdot 6 + 9) = 81 = 13 \pmod{17}$.

Complete the module "a3p2.py" by implementing the function "random_generator(a, b, c, m, $r0$, $r1$, n)", where a, b, c, and m are as above, and $r0$ and $r1$ are the seeds values of R_0 and R_1, which **returns a list of integers** containing the next n elements of generated numbers (R_2, R_3, ..., R_{n+1}). The following demonstrates invocation sequences that should run without error and produce identical output:

```
 >> random_generator(3, 5, 9, 17, 11, 6, 3)
[14, 13, 16]

>> random_generator(22695477, 77557187, 259336153, 9672485827, 42, 51, 8)
[4674207334, 3722211255, 3589660660, 1628254817,
8758883504, 7165043537, 4950370481, 2261710858]

>> random_generator(2**31-5, 743, 549, 1559861749, 97, 101, 8)
[75137452, 935657016, 1474108152, 1106636826, 405962062, 778970349, 1377654917,
1174493038]

>> random_generator(1128889, 1023, 511, 222334565193649, 65535, 329, 8)
[438447297, 50289200612813, 17962583104439, 47361932650166,
159841610077391, 19587857129781, 111993173627854, 7567964632208]
```

Problem 3 (3 Marks)

Short answer: According to the given algorithm in problem 2, Alice started to generate some random numbers with $m = 467$, generates the numbers $R_2 = 28$, $R_3 = 137$, $R_4 = 41$, $R_5 = 118$, and $R_6 = 105$. Help Eve to predict next random numbers by determining the values of a, b, c, R_0, R_1 and R_7. Include the values of these **six** variables in a3.txt, with all relevant work and explanation for how you found them, in your a3.pdf.

Note: Please strictly follow the template provided in a3.txt. Any change in the template might result in a potential mark deduction.

Problem 4 (3 Marks)

In problem 3, You were able to "hack" Alice's algorithm by obtaining a, b, and c and thus predict all of its future output (you predicted R_7, but of course, since a, b, and c are fixed, you could predict any quantity of future values). In this problem, you will automate this process.

Complete the module "a3p4.py" by implementing the function "crack_rng(m, *sequence*)", where m is a **prime number**, and *sequence* is list of five consecutive numbers which are generated by the generator (i.e. $[R_2, R_3, R_4, R_5, R_6]$) and they're all between 0 and $m - 1$, inclusive. This function **must return a list of integers** $[a, b, c]$, where a, b, and c are the keys for the provided random generator at problem 2, with modulus m, which outputs five consecutive numbers such as input sequence. It is guaranteed that a and b are non-zero.

Hint: While not advisable in practice, it is perfectly valid for our algorithm to have $c = 0$, and still have $a, b \neq 0$. Your crack_rng function *must be able to crack RNGs with* $c = 0$*,* $a, b \neq 0$, still returning the three-element list $[a, b, c]$. We guaranteed that there is a unique solution for a, b, and c.

```
 >> crack_rng(17, [14, 13, 16, 3, 13])
[3, 5, 9]

>> crack_rng(9672485827,[4674207334,3722211255,3589660660,1628254817,8758883504])
[22695477, 77557187, 259336153]

>> crack_rng(101, [0, 91, 84, 16, 7])
[29, 37, 71]

>> crack_rng(222334565193649,
        [438447297,50289200612813,17962583104439,47361932650166,159841610077391])
[1128889, 1023, 511]
```