Criptografia i Seguretat [104355]

Activity Pseudo-Random Generators

In these set of exercises you are required to implement the specified code described in the questions.

1. We are going to implement now a simple LFSR as a Python class. We will use the notation and structure of the LFSR described in the book (e.g. Figure 3.2). The LFSR is described with two main parameters:

   1. Initial state: the value of the registers $S_n, S_{n-1}, \ldots, S_2, S_1$. These values will be represented as a `numpy` array of ones and zeros. Be aware of the ordering!. For example the LFSR of 4 cells with initial state $S_4 = 1, S_3 = 0, S_2 = 1, S_1 = 0$ can be coded as: `np.array([0, 1, 0, 1])`
   2. Connection polynomial (*polinomi de connexions*): value of the c_i parameters. So the values $c_1 = 0, c_2 = 1, c_3 = 0, c_4 = 1$ can be coded as: `np.array([0, 1, 0, 1])`

   We will make a simple implementation using the following template:

```python
import numpy as np

class LFSR():

    def __init__(self, state: np.ndarray, pol: np.ndarray)
        # CODE GOES HERE

    def next(self) -> int:
        # CODE GOES HERE
```

   - The constructor receives the state and feedback polynomial and should initialize the current state of the LFSR.
   - The `next()` method computes one step of the LFSR returning one bit as output and updating the current state of the LFSR.

   In order to verify the code, your LFSR implementation should generate the following sequences when using the following parameters:

```python
test_s1 = np.array([1,0,1])
test_c1 = np.array([1,1,1])
test_lfsr1 = LFSR(test_s1, test_c1)
test_seq1 = np.array([test_lfsr1.next() for _ in range(20)])
print(f"Test sequence 1:{test_seq1}")
test_s2 = np.array([1,0,0,0,1])
test_c2 = np.array([1,1,0,0,1])
test_lfsr2 = LFSR(test_s2, test_c2)
test_seq2 = np.array([test_lfsr2.next() for _ in range(20)])
print(f"Test sequence 2:{test_seq2}")
```

   Executing this code should output the sequences:

```
Test sequence 1:[1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0]
Test sequence 2:[1 0 0 0 1 0 1 1 0 0 0 1 0 1 1 0 0 0 1 0]
```

It is important to ensure that your code produces the same sequences, since otherwise the responses for the following exercises might be different.

2. With the LFSR we have implemented, generate the sequences of length 1000 bits, with the corresponding initial state s_i and initial values for c_i:

   - `seq_1`:

     ```
     s1 = np.array([1,1,1,1,1,1])
     c1 = np.array([1,0,0,0,0,1])
     ```

   - `seq_2`:

     ```
     s2 = np.array([1,0,0,1,1,1])
     c2 = np.array([1,0,0,1,1,0])
     ```

   - `seq_3`:

     ```
     s3 = np.array([1,0,0,0,0,0])
     c3 = np.array([0,0,0,0,1,0])
     ```

   Please note that these are testing data which might not make sense in a real LFSR implementation (e.g. the connection polynomial should have maximum degree).

3. Implement in Python the 3 tests from the NIST Test Suite for Random Number Generators (RNGs) described in the book.

   - The sequences of bits will be represented as `numpy` arrays of integers. E.g.

     ```
     sequence = np.array([1,0,0,1,1,0,1,0,0,0,0,1,1,1,1,0,1,0,1,1,0,0,1])
     ```

   - Use the following name and parameters for each test:
     - Frequency Test:
       `def rng_test_1(seq: np.ndarray) -> Tuple[bool, float]`
     - Frequency within a block:
       `def rng_test_2(seq: np.array, m: int = 10) -> Tuple[bool, float]`
       Note that we are setting a default value for $m = 10$, the block size. If not explicitly stated we will use this value when running the test.
     - Runs (*ràfegues*):
       `def rng_test_3(seq: np.ndarray) -> Tuple[bool, float]`
   - **Important:** Note that all functions return a tuple with a boolean and a float value. The boolean should be True if the sequence passes the test or False otherwise. The float is the p-value of the test.
   - To compute the final *p-value* of the test you can use the following functions: `math.erfc`, `mpmath.gammainc`
   - A full reference of the test can be found in the original publication: *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*.
   - In general favour correctness and clarity over performance.
   - For Test 1 use the following condition:

     ```
     p_value = math.erfc(s_obs / math.sqrt(2))
     return p_value >= 0.01, p_value
     ```

- For Test 2 use the following condition:

```
b = len(seq) // m
p_value = float(mpmath.gammainc(b / 2, x_obs / 2, regularized=True))
return p_value >= 0.01, p_value
```

- For Test 3 use the following condition:

```
pi = sum(seq) / len(seq)
numerator = abs(v_obs - (2.0 * len(seq) * pi * (1.0 - pi)))
denominator = 2.0 * math.sqrt(2.0 * len(sequence)) * pi * (1.0 - pi)
value = math.erfc(numerator / denominator)
return value >= 0.01, value
```

4. For each sequence in question 2, run the three tests from question 3.