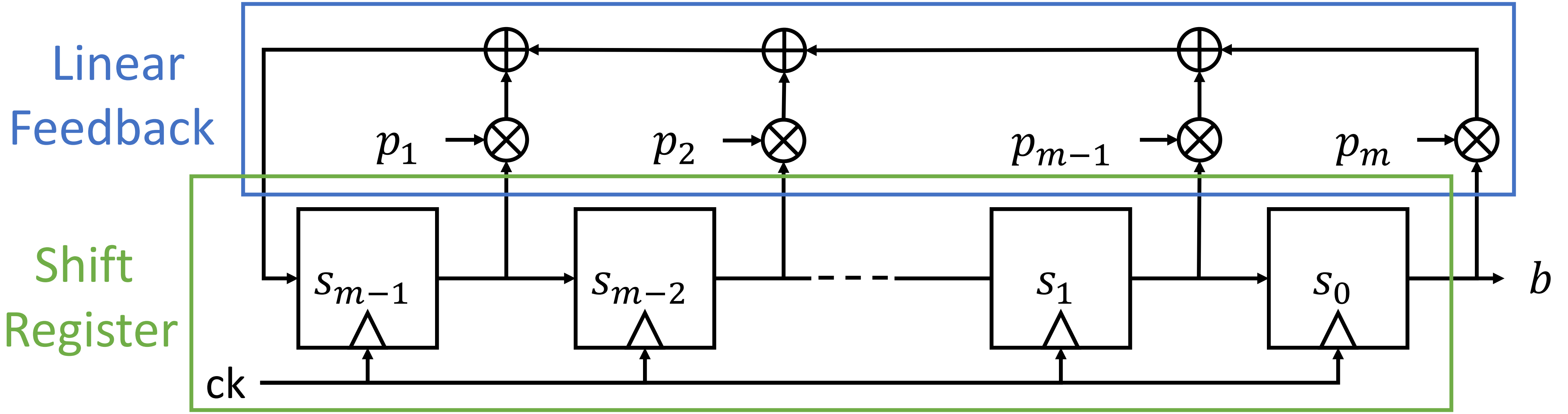
Linear Feedback Shift Register (LFSR)

Elements of Applied Data Security

Alex Marchioni – alex.marchioni@unibo.it

LFSR

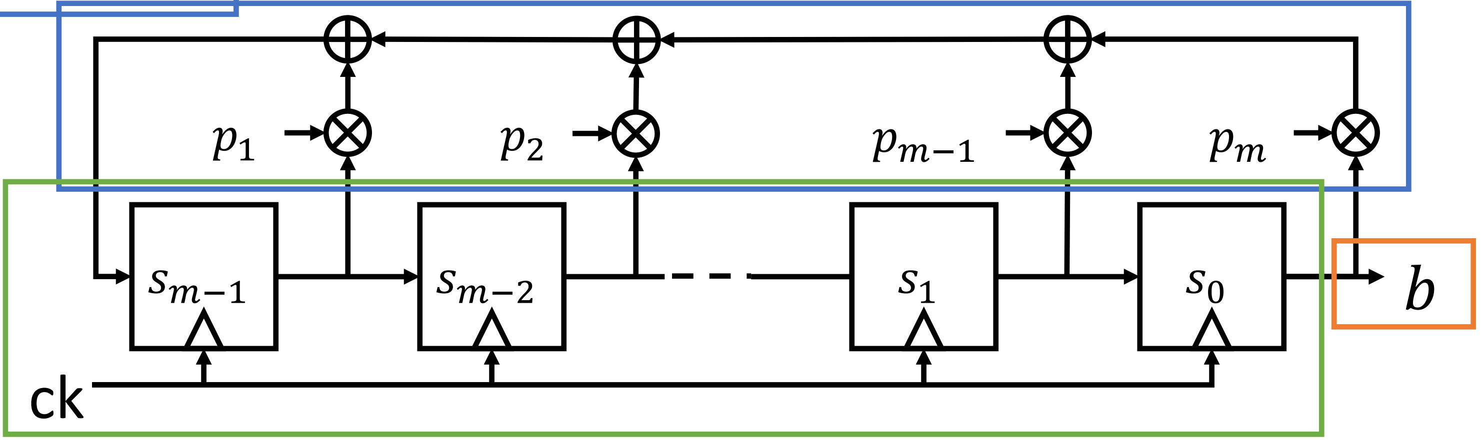
In an LFSR, the output from a standard shift register is fed back into its input causing an endless cycle. The feedback bit is the result of a linear combination of the shift register content and the feedback coefficients.



LFSR

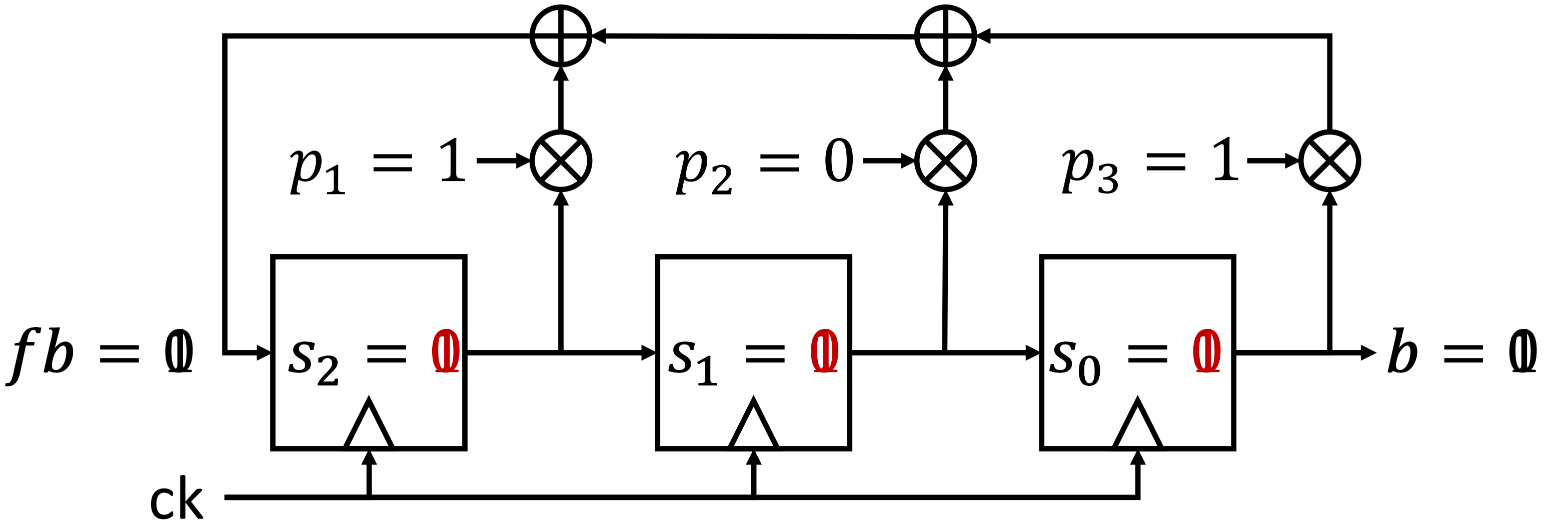
From the block scheme:

$$\begin{cases} \boxed{s_j[t] = s_{j+1}[t-1], \qquad j = 0,1,\ldots,m-2} \\ \boxed{s_{m-1}[t] = \bigoplus_{j=0}^{m-1} p_{m-j} \otimes s_j[t-1]} \\ \boxed{b[t] = s_0[t]} \end{cases}$$



LFSR example

- length = 3
- polynomial = $x^3 + x + 1$ ($p = 0b1011$)
- initial state $0b111$



s	b	fb
111 (7)	1	0
011 (3)	1	1
101 (5)	1	0
010 (2)	0	0
001 (1)	1	1
100 (4)	0	1
110 (6)	0	1
111 (7)	1	0

Task

1. Define a **generator** that implements an LFSR. Given a polynomial and an initial state, it generates an infinite stream of bits.
2. Transform the LFSR generator in an **iterator**, so that it is possible to access to the internal state as an attribute of the class.
3. Define a function that implements the **Berlekamp-Massey algorithm** which finds the shortest LFSR that can generate the input bit stream.
4. Transform the function implementing the Berlekamp-Massey algorithm into a class that can be applied in a **streaming** way.

Task 1: LFSR Generator

LFSR Generator

Inputs:

- **Feedback Polynomial**: list of integers representing the degrees of the non-zero coefficients.
  Example: $[12, 6, 4, 1, 0]$ represents $x^{12} + x^6 + x^4 + x^1 + 1$
- **LFSR state** (optional, default all bits to 1): Integer or list of bits representing the LFSR initial state.
  Example: 0xA65 for [1010 0110 0101]

Yield:

- **Output bit**: bool representing the LFSR output bit

LFSR Generator

Template:

```python
def lfsr_generator(poly, state=None):
  ''' generator docstring '''

  # check inputs
  # define variables storing the internal state

  while True:
    # LFSR iteration:
    # - compute output from poly and state
    # - update state
    yield output
```

Infinite Iterables

To deal with iterables that counts an infinite number of elements, we can use the function `islice` from the built-in package itertools.

This function allows you to take a finite number of elements.

```
from itertools import islice

niter = ... # number of iterations
lfsr = lfsr_generator(...) # define the lfsr generator

for b in islice(lfsr, niter):
  # do stuff
  pass
```

Hints

There are many ways to implement an LFSR in Python.

The first choice to make is how to store the internal state and the polynomial. I suggest two types:

- **list of bool**: it is the most straightforward choice as it directly maps the LFSR block scheme, but bit-wise logical operation may not be as easy.
- **integer**: bit-wise logical operation, as well as bit-shift, are easy to perform on integers, while XOR of multiple bits or reversing the bit order are less straightforward.

Useful functions

- **XOR**: In Python bit-wise xor between two integers is implemented with the `^` mark. It is also implemented as function (`xor`) in the built-in module [`operator`]. Example: `xor(5,4) -> 5^4 -> 0b101^0b100 -> 0b001 -> 1`
- **reduce**: available from the built-in module [`functools`], apply a function of two arguments cumulatively to the items of an iterable so as to reduce the iterable to a single value.
  Example: `reduce(xor, [True, False, True, False]) -> False`
- **compress**: available from the built-in module [`itertools`], make an iterator that filters elements from data returning only those that have a corresponding element in selectors that evaluates to True.
  Example: `compress([3, 7, 5], [True, False, True]) -> [3, 5]`

Task 2: LFSR Iterator

LFSR Iterator

Inputs:

- **Feedback Polynomial**:
  list of integers representing the degrees of the non-zero coefficients.
  Example: $[12, 6, 4, 1, 0]$ represents $x^{12} + x^6 + x^4 + x^1 + 1$
- **LFSR state** (optional, default all bits to 1)
  Integer or bitstream representing the LFSR initial state
  Example: 0xA65 for [1010 0110 0101]

LFSR Iterator

Attributes:

- **poly**: list of the polynomial coefficients (list of int)
- **length**: polynomial degree and length of the shift register (int)
- **state**: LFSR state (int)
- **output**: output bit (bool)
- **feedback**: last feedback bit (bool)

LFSR Iterator

Methods:

- **__init__**: class constructor;
- **__iter__**: necessary to be an iterable;
- **__next__**: update LFSR state and returns output bit;
- **cycle**: returns a list of bool representing the full LFSR cycle ;
- **run_steps**: execute N LFSR steps and returns the corresponding output list of bool (N is a input parameter, default N=1);
- **__str__**: return a string describing the LFSR class instance.

LFSR Iterator

```python
class LFSR(object):
  ''' class docstring '''

  def __init__(self, poly, state=None):
    ''' constructor docstring '''
    ...
    self.poly = ...
    self.length = ...
    self.state = ...
    self.output = ...
    self.feedback = ...

  def __iter__(self):
    return self

  def __next__(self):
    ''' next docstring '''
    ...
    return self.output

  def run_steps(self, N=1):
    ''' run_steps docstring '''
    ...
    return list_of_bool

  def cycle(self, state=None):
    ''' cycle docstring '''
    ...
    return list_of_bool
```

Task 3:
Berlekamp-Massey Algorithm

Berlekamp-Massey Algorithm

Find the shortest LFSR for a given binary sequence.

- **Input**: sequence of bit b of length N
- **Outputs**: feedback polynomial $P(x)$.

```python
def berlekamp_massey(b):
  ''' function docstring '''
  # algorithm implementation
  return poly
```

Input $b = [b_0, b_1, \dots, b_N]$
$P(x) \leftarrow 1, m \leftarrow 0$
$Q(x) \leftarrow 1, r \leftarrow 1$
For $\tau = 0,1, \dots, N-1$
  $d \leftarrow \bigoplus_{j=0}^{m} p_j \otimes b[\tau - j]$
  If $d = 1$ **then**
    If $2m \leq \tau$ **then**
      $R(x) \leftarrow P(x)$
      $P(x) \leftarrow P(x) + Q(x)x^r$
      $Q(x) \leftarrow R(x)$
      $m \leftarrow \tau + 1 - m$
      $r \leftarrow 0$
    else
      $P(x) \leftarrow P(x) + Q(x)x^r$
    endif
  endif
  $r \leftarrow r + 1$
endfor
Output $P(x)$

Berlekamp-Massey Algorithm

τ	b_τ	d		$P(x)$	m	$Q(x)$	r
-	-	-		1	0	1	1
0	1	1	A	$1+x$	1	1	1
1	0	1	B	1	1	1	2
2	1	1	A	$1+x^2$	2	1	1
3	0	0		$1+x^2$	2	1	2
4	0	1	A	1	3	$1+x^2$	1
5	1	1	B	$1+x+x^3$	3	$1+x^2$	2
6	1	0		$1+x+x^3$	3	$1+x^2$	3
7	1	0		$1+x+x^3$	3	$1+x^2$	4

Input $b = [b_0, b_1, \dots, b_N]$
$P(x) \leftarrow 1, m \leftarrow 0$
$Q(x) \leftarrow 1, r \leftarrow 1$
For $\tau = 0,1,\dots,N-1$

$$d \leftarrow \bigoplus_{j=0}^{m} p_j \otimes b[\tau - j]$$

If $d = 1$ **then**
If $2m \le \tau$ **then**

A
$R(x) \leftarrow P(x)$
$P(x) \leftarrow P(x) + Q(x)x^r$
$Q(x) \leftarrow R(x)$
$m \leftarrow \tau + 1 - m$
$r \leftarrow 0$

else

B
$P(x) \leftarrow P(x) + Q(x)x^r$

endif
endif
$r \leftarrow r + 1$
endfor
Output $P(x)$

Task 4: Berlekamp-Massey Streaming Algorithm

Streaming Algorithm

An algorithm is streaming when the **input is a stream of symbols**.

That means the input is a sequence of items and the algorithm is applied to only <u>one item (or a group of successive items) at a time</u>.

Example: implementation of max function in batch and streaming.

batch

```
>>> x = [8, 3, 2, 6, 9, 9, 5, 7, 5, 5]
>>> x_max = max_batch(x)
>>> x_max
9
```

- whole sequence as input.

streaming

```
>>> x = [8, 3, 2, 6, 9, 9, 5, 7, 5, 5]
>>> for xj in x:
        x_max = max_stream(xj)
>>> x_max
9
```

- input is provided one item at a time.
- output is returned at each iteration

Berlekamp-Massey Streaming Algorithm

Since Berlekamp-Massey Algorithm may be applied to very long sequence of bits, therefore, it may be convenient to implement it as a streaming algorithm.

- **Lower memory requirements** as you need to store only the input bits that are significant for the identification of the LFSR polynomial.

```
bit_stream = bit_generator()

for bit in bit_stream:
    poly = berlekamp_massey(bit)
```

- One bit `bit` at a time as input
- At each new bit, it returns the polynomial `poly` generating the bit sequence observed until then

Implementation

Since streaming algorithms needs an **internal state** to be updated at each new incoming symbol of the stream, they cannot be implemented a normal functions. We can implement it as a class.

```python
class BerlekampMassey():

  def __init__(self):
    # do stuff
    self.poly = ...

  def __call__(self, bit):
    # do stuff
    return self.poly
```

- The special method `__call__` make the class **callable**, that means you can call it as a function.

```python
berlekamp_massey = BerlekampMassey()

for bit in bit_stream:
    poly = berlekamp_massey(bit)
```