Stream Ciphers

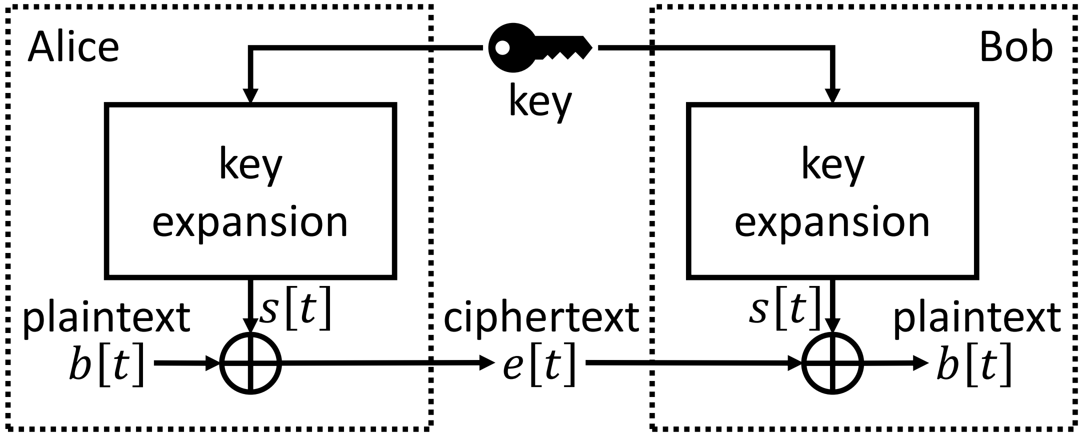
Elements of Applied Data Security M

Alex Marchioni – alex.marchioni@unibo.it

Livia Manovi – livia.manovi@unibo.it

Stream Cipher

A stream cipher is a **symmetric key** cipher where the plaintext is encrypted (and ciphertext is decrypted) one digit at a time. A digit usually is either a bit or a byte.



Encryption (decryption) is achieved by xoring the plaintext (ciphertext) with a stream of pseudorandom digits obtained as an expansion of the key.

Tasks

1. LFSR
2. Berlekamp-Massey Algorithm
3. LFSR-based generator
4. Bonus Task: Statistical Tests

Python Module

- A **module** is a `.py` file in which you can collect several definitions that you may want to use repeatedly.
- You can then **import** such definitions either into your notebook or even in other modules.

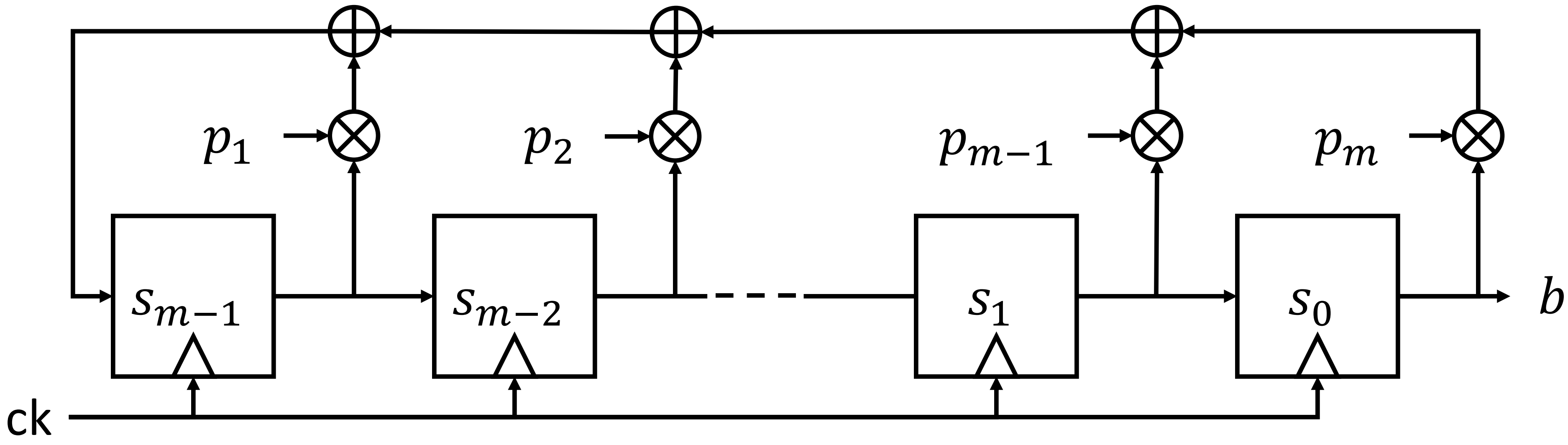
For the current assignment:

- You <u>must</u> collect all the functions and objects employed to implement the requested structures in a python module `streamcipher.py` following the naming conventions indicated in the following slides.

Task 1: LFSR

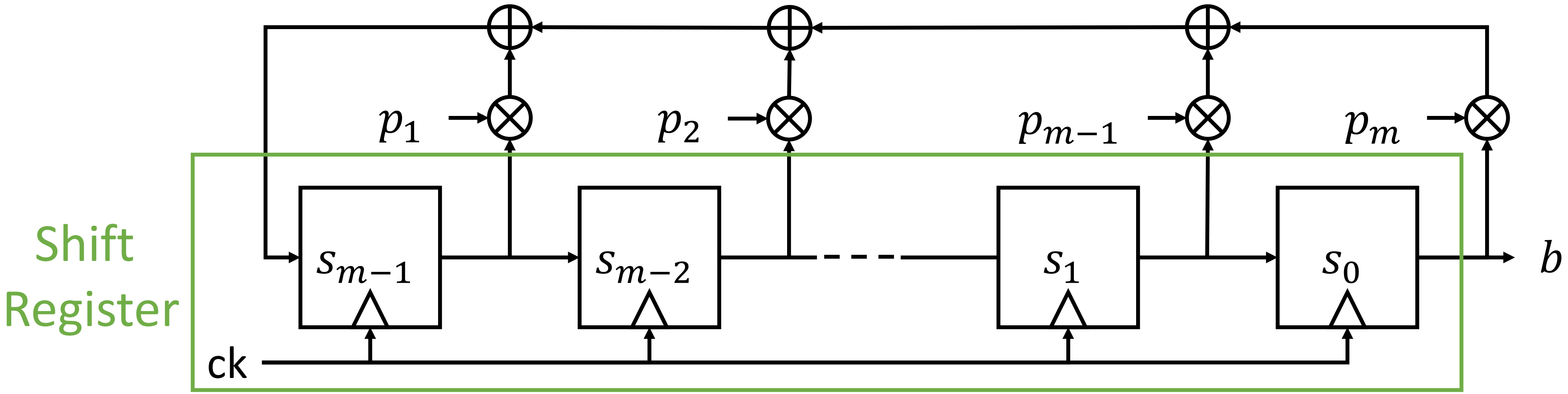
LFSR

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 polynomial coefficients.



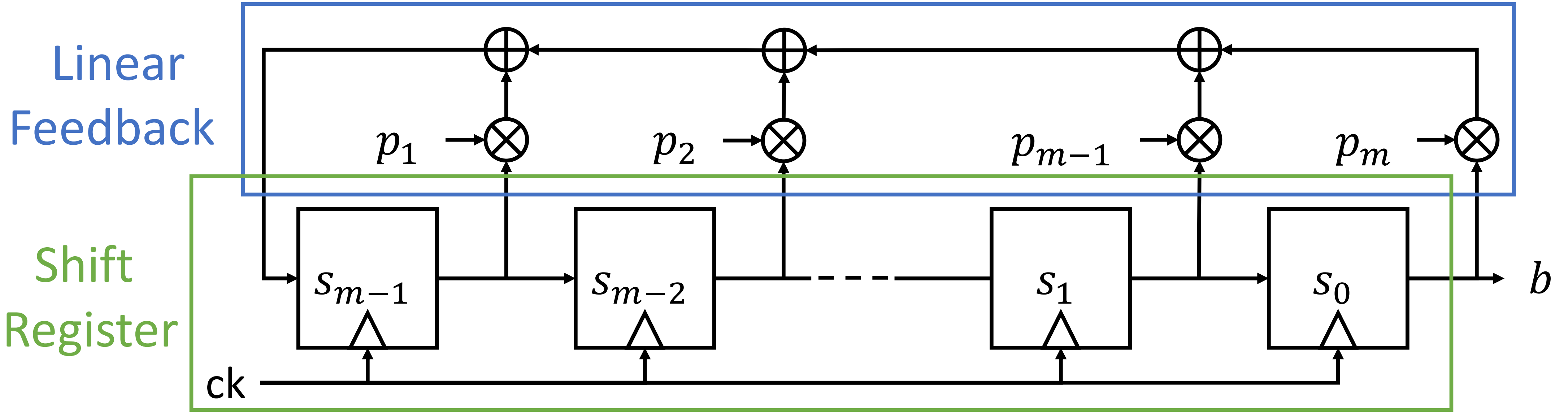
LFSR

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 polynomial coefficients.



LFSR

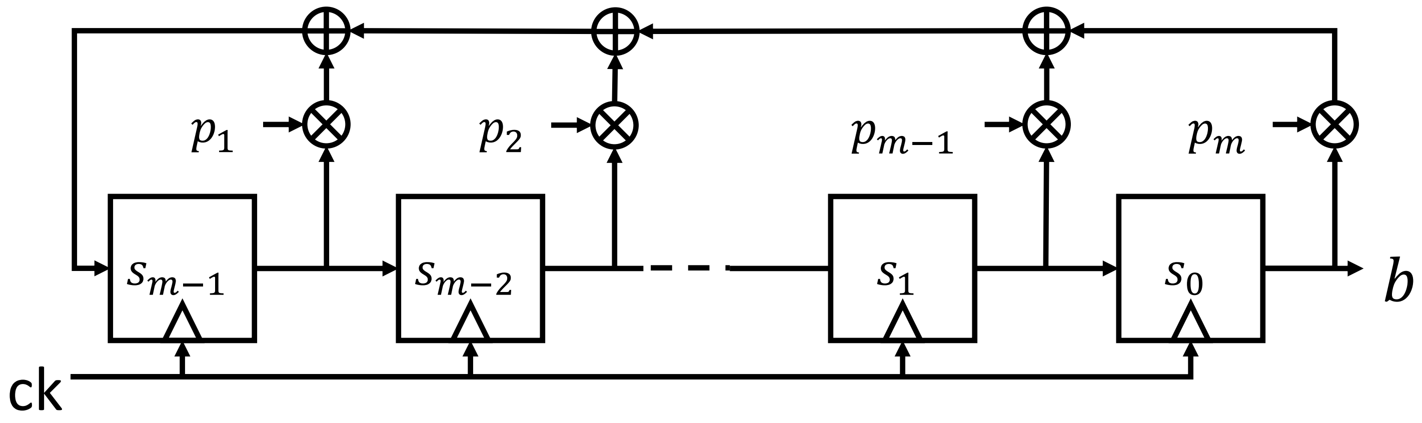
In an LFSR, the output from a standard shift register is fed back into its input causing an endless cycle. The feedback bit is the result of a linear combination of the shift register content and the polynomial coefficients.



LFSR

From the block scheme:

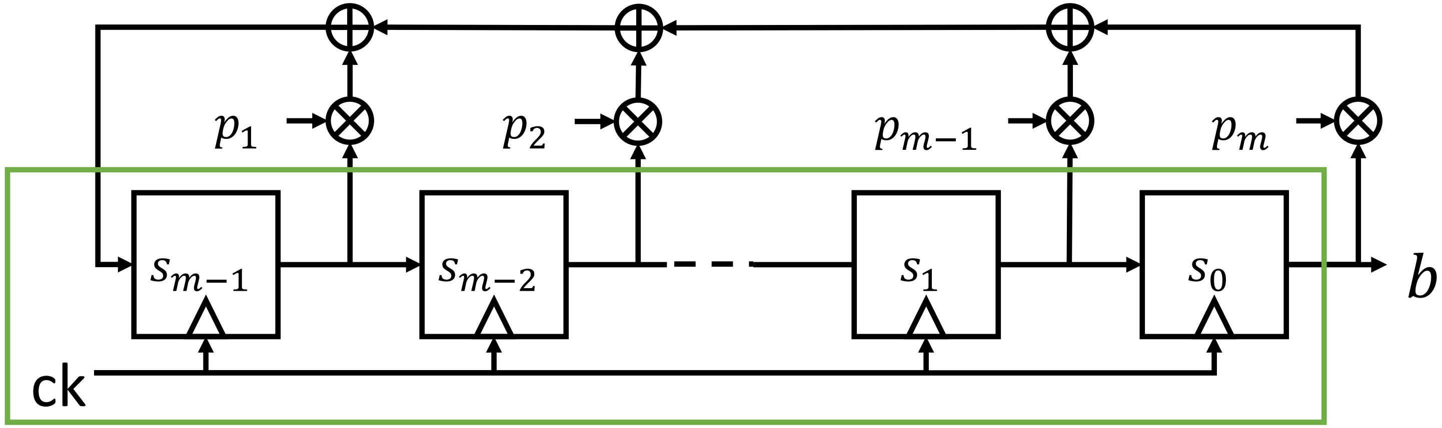
$$
\begin{cases}
s_j[t] = s_{j+1}[t-1], \qquad j = 0,1,\dots,m-2 \\
s_{m-1}[t] = \bigoplus_{j=0}^{m-1} p_{m-j} \otimes s_j[t-1] \\
b[t] = s_0[t]
\end{cases}
$$



LFSR

From the block scheme:

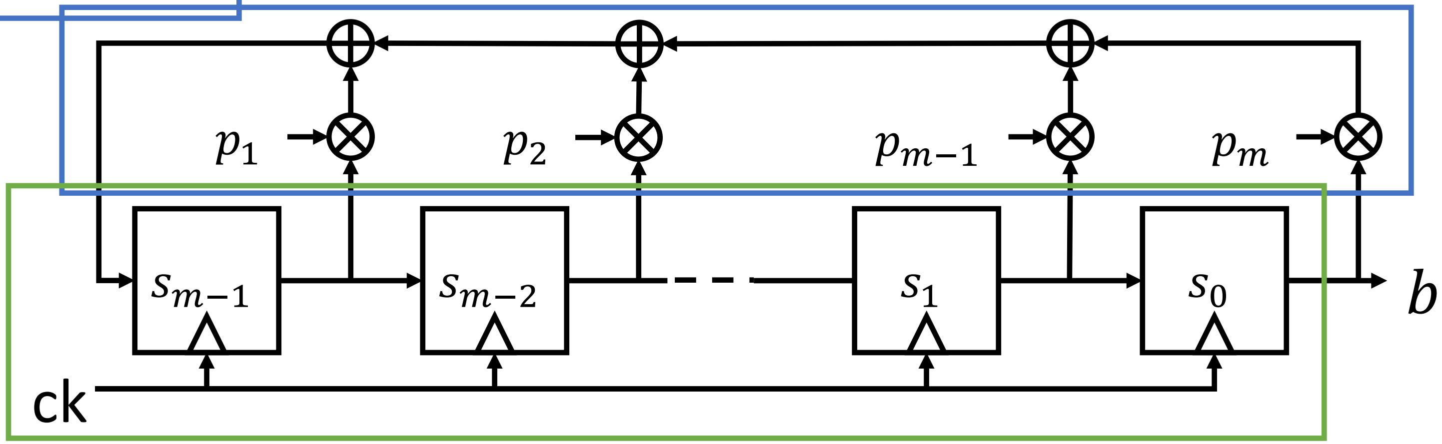
$$\begin{cases} \boxed{s_j[t] = s_{j+1}[t-1], \qquad j = 0,1,\dots,m-2} \\ s_{m-1}[t] = \displaystyle\bigoplus_{j=0}^{m-1} p_{m-j} \otimes s_j[t-1] \\ b[t] = s_0[t] \end{cases}$$



LFSR

From the block scheme:

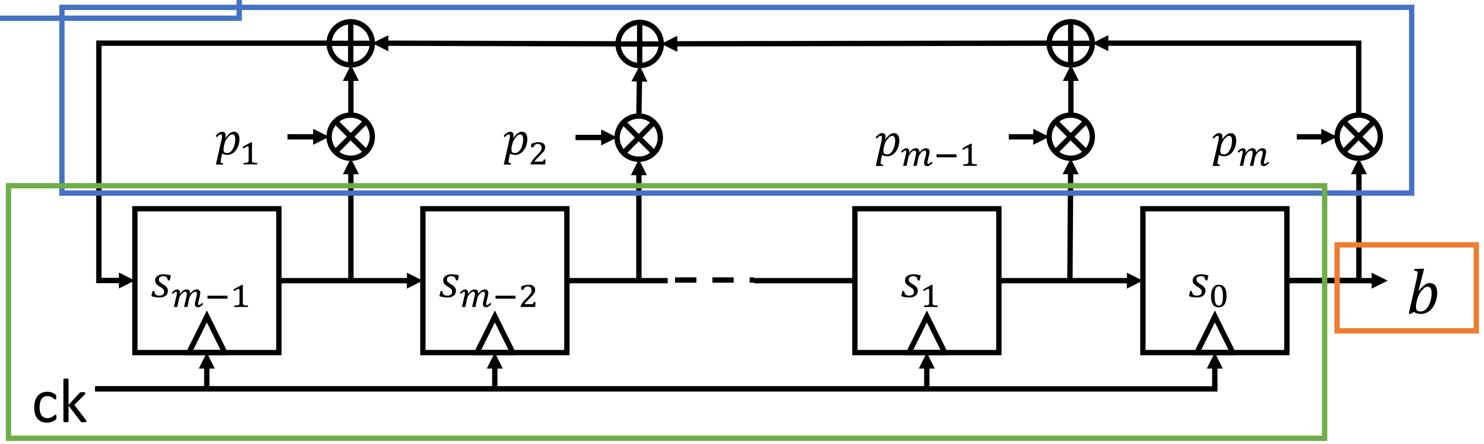
$$\begin{cases} \boxed{s_j[t] = s_{j+1}[t-1], \qquad j = 0,1,\dots,m-2} \\ \boxed{s_{m-1}[t] = \bigoplus_{j=0}^{m-1} p_{m-j} \otimes s_j[t-1]} \\ b[t] = s_0[t] \end{cases}$$



LFSR

From the block scheme:

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

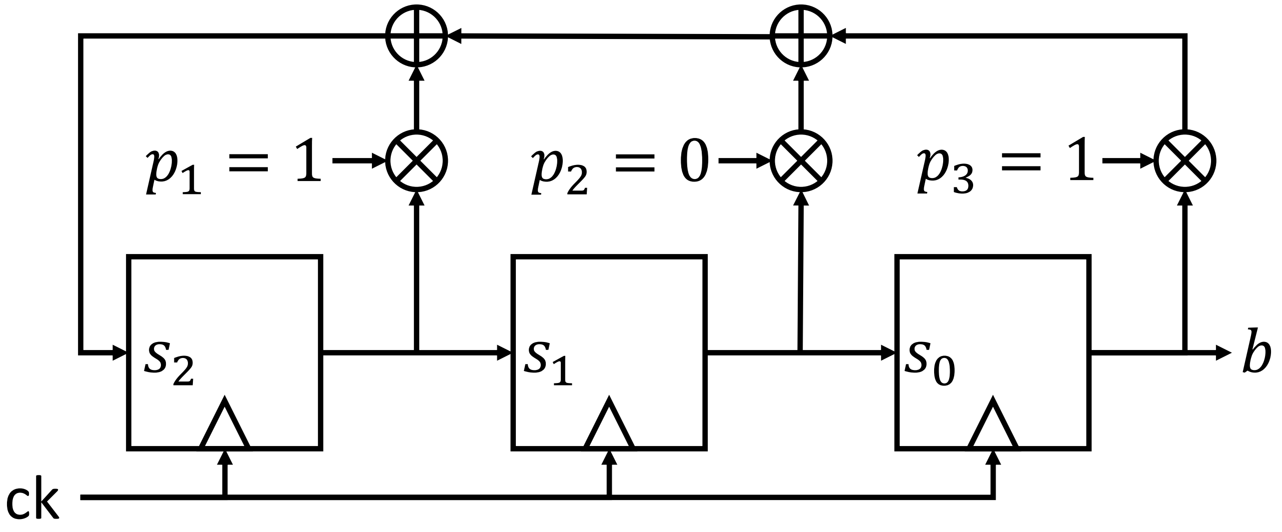


LFSR example

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

s	b	fb

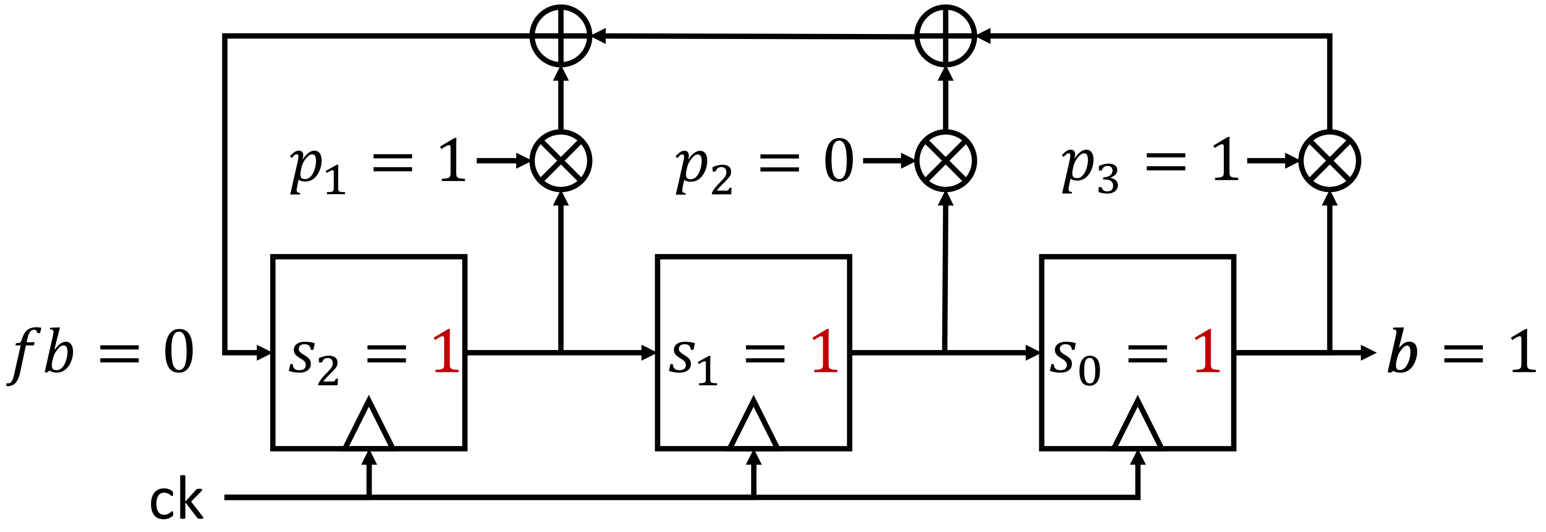
fb



LFSR example

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

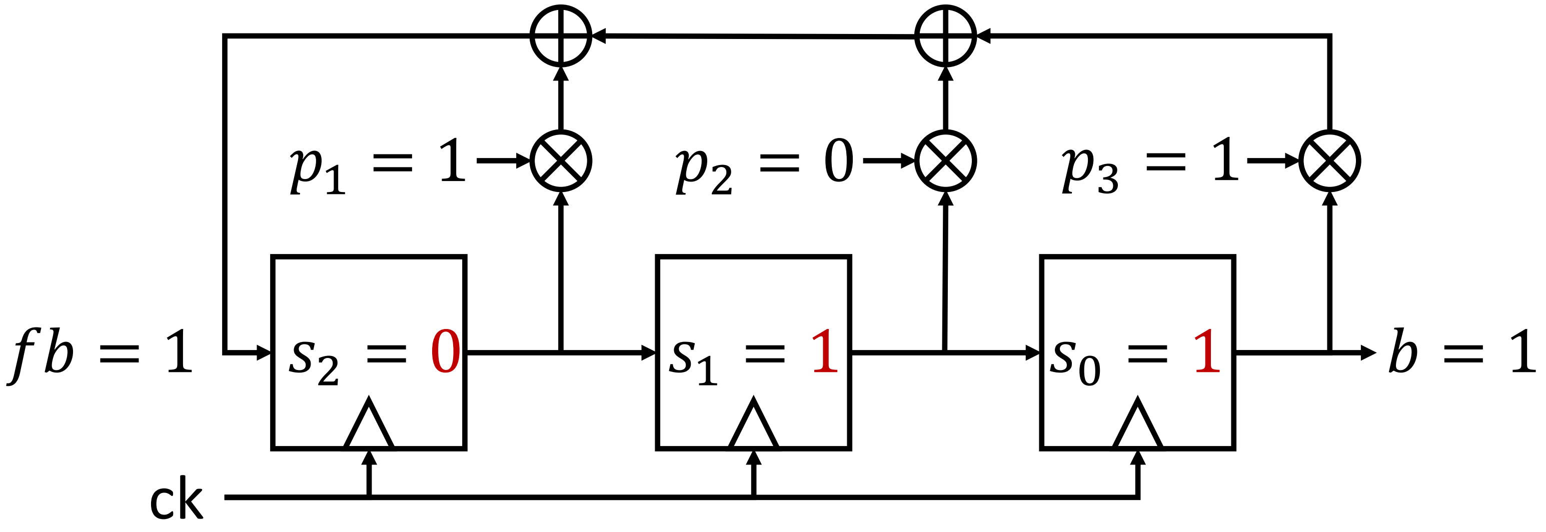
s	b	fb
111 (7)	1	0



LFSR example

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

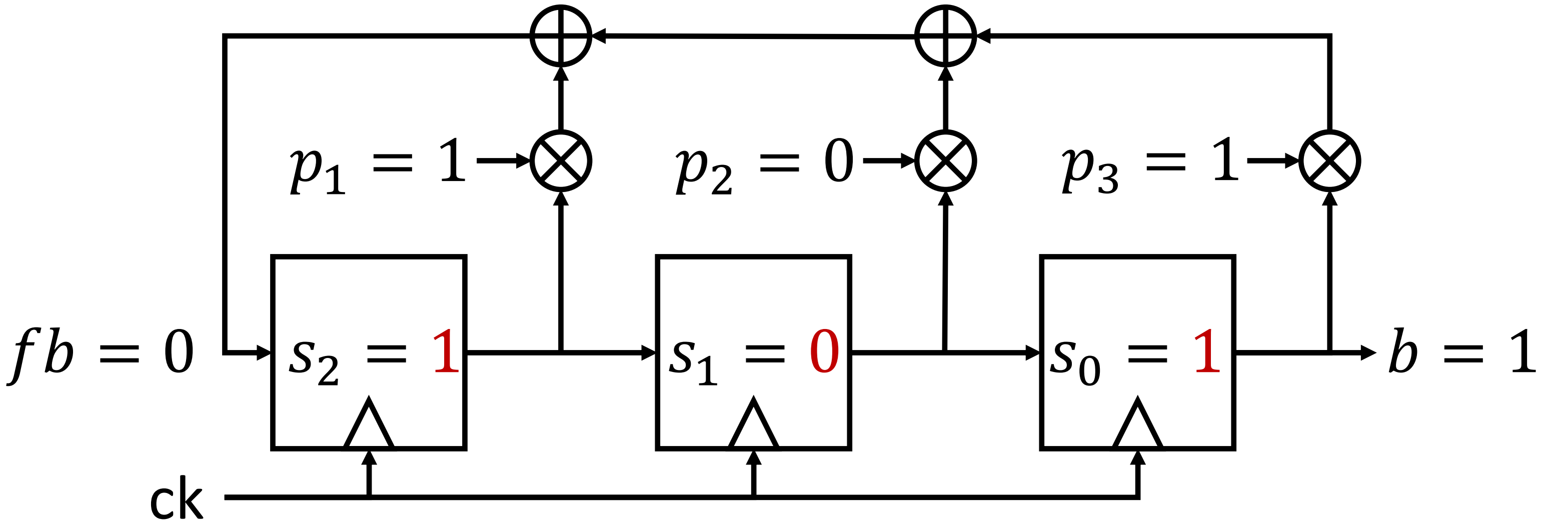
s	b	fb
111 (7)	1	0
011 (3)	1	1



LFSR example

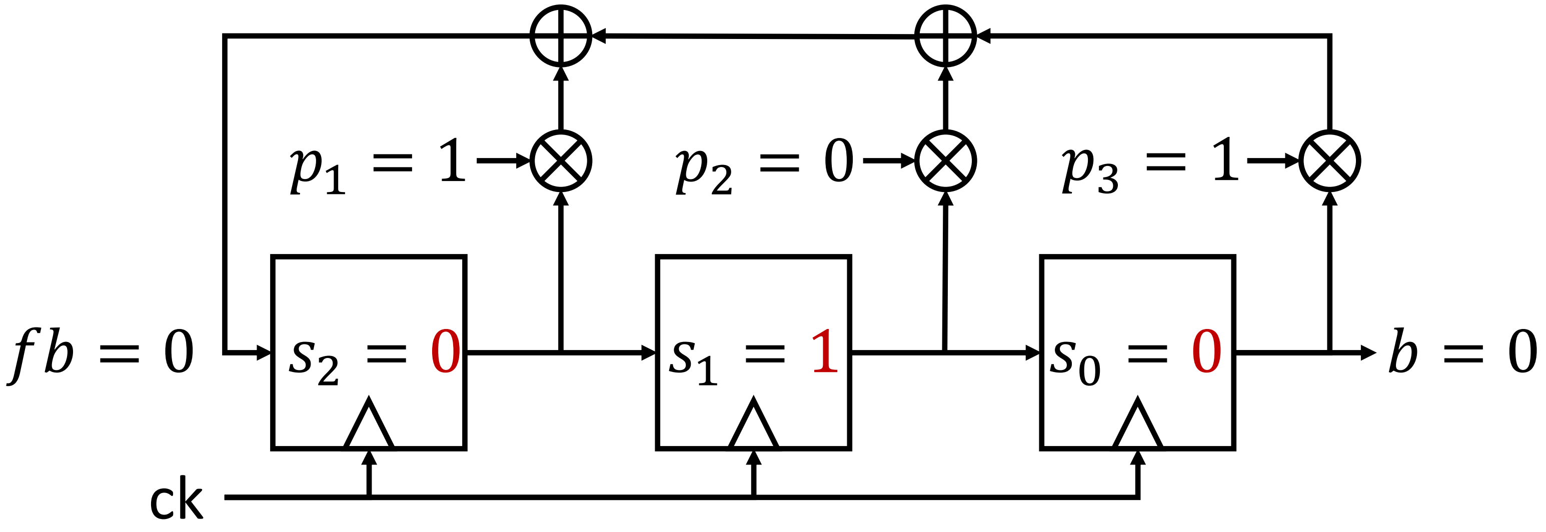
- length = 3
- polynomial = $x^3 + x + 1$ ($p = 0b1011$)
- initial state $0b111$

s	b	fb
111 (7)	1	0
011 (3)	1	1
101 (5)	1	0



LFSR example

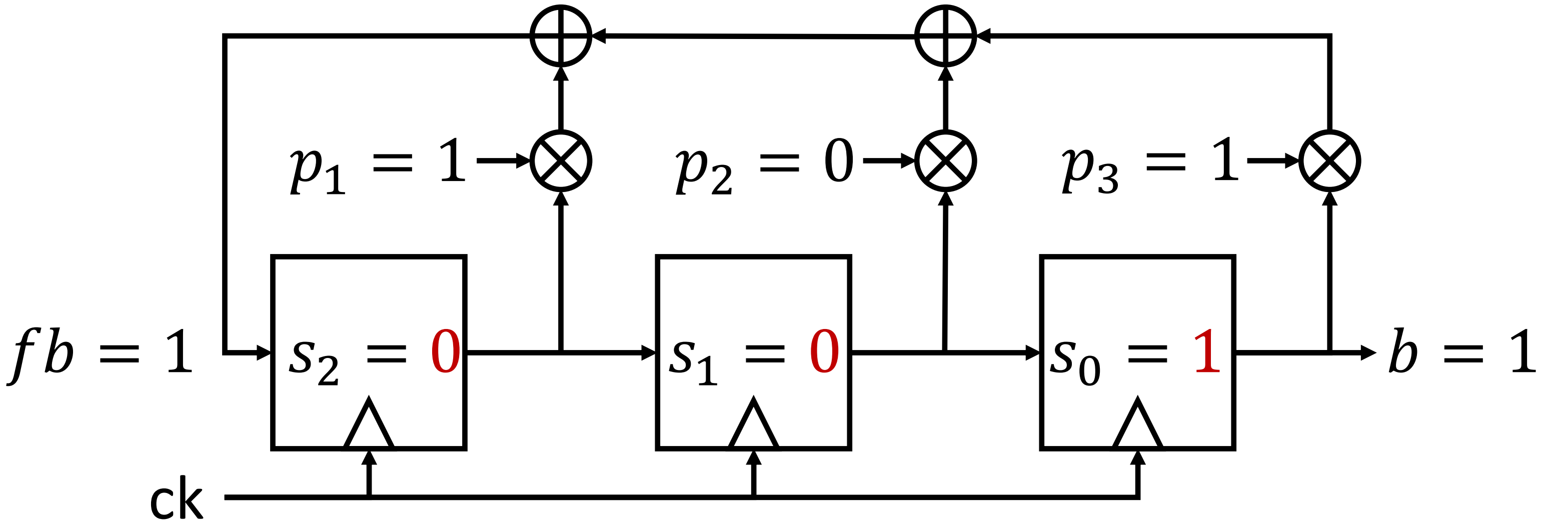
- length = 3
- polynomial = $x^3 + x + 1$ ($p = 0b1011$)
- initial state $0b111$



s	b	fb
111 (7)	1	0
011 (3)	1	1
101 (5)	1	0
010 (2)	0	0

LFSR example

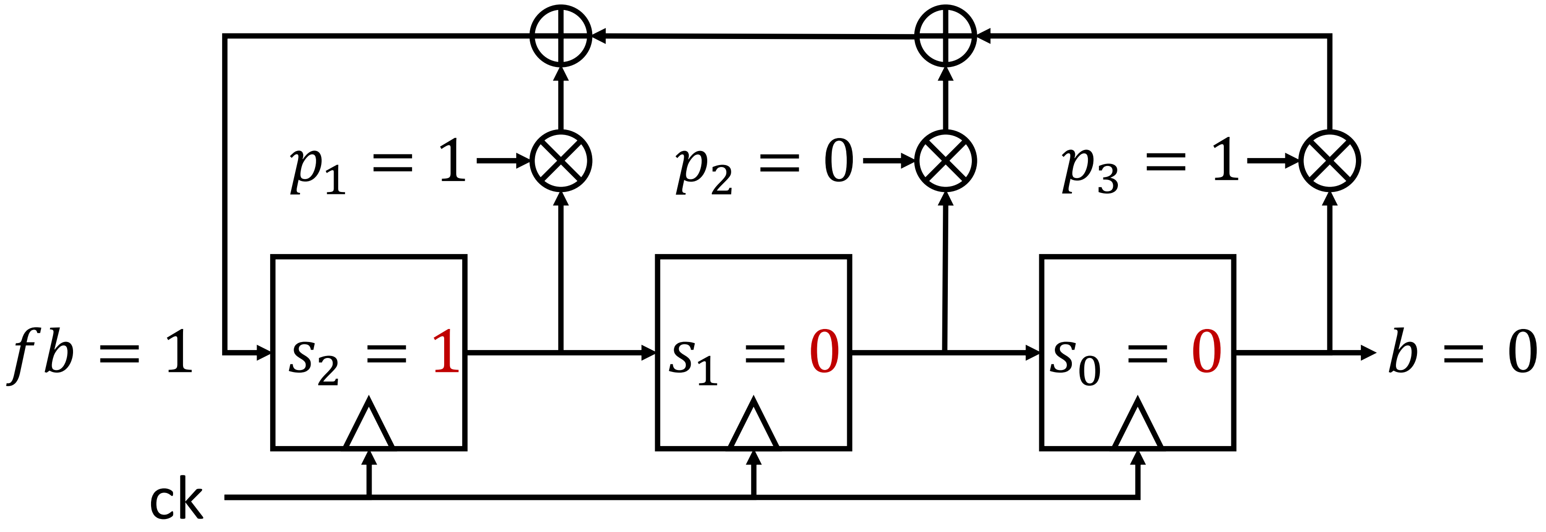
- length = 3
- polynomial = $x^3 + x + 1$ ($p = 0b1011$)
- initial state $0b111$



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

LFSR example

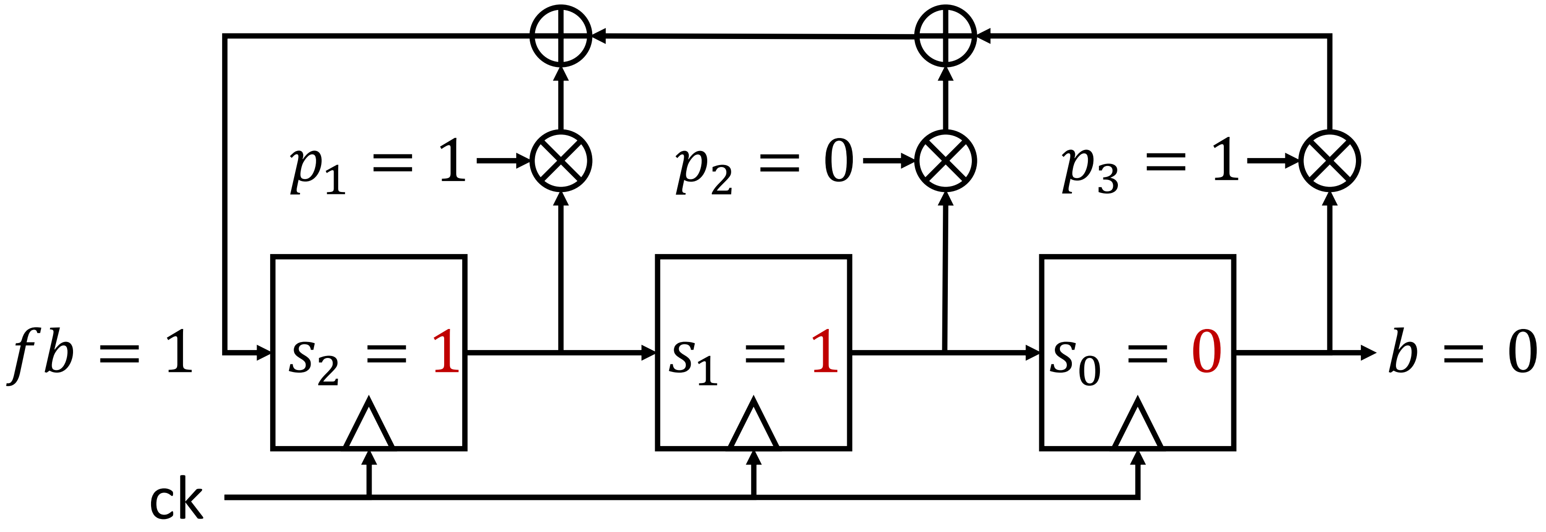
- length = 3
- polynomial = $x^3 + x + 1$ ($p = 0b1011$)
- initial state $0b111$



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

LFSR example

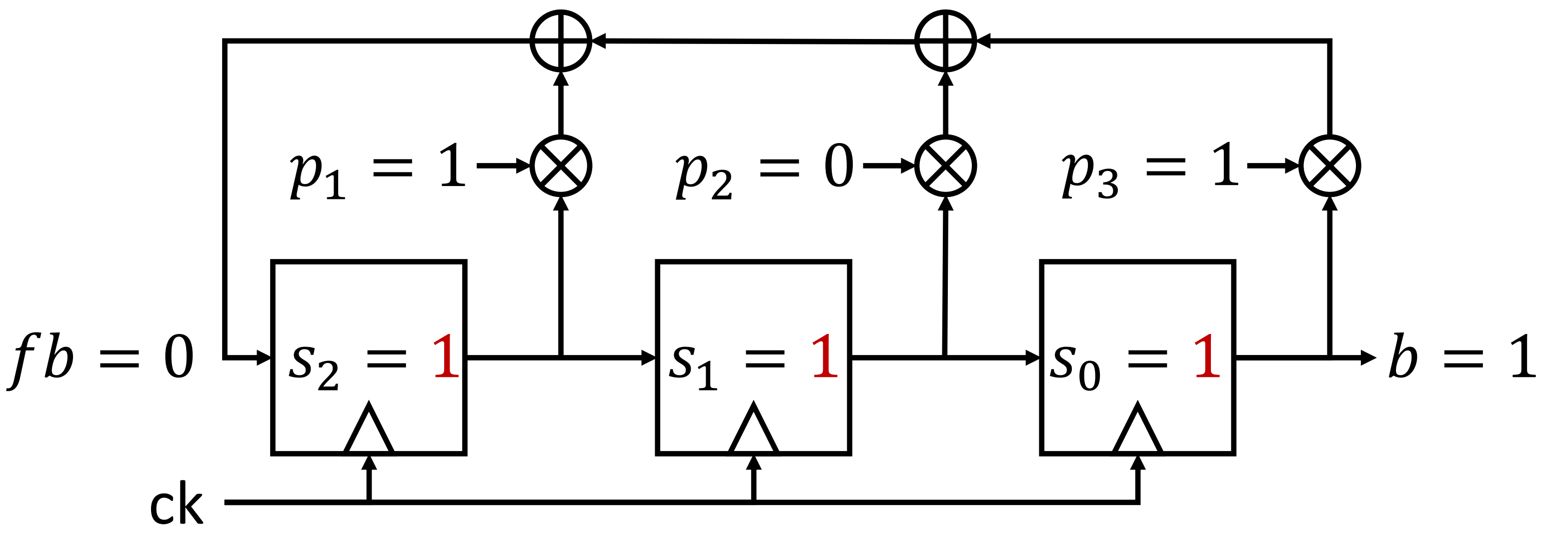
- length = 3
- polynomial = $x^3 + x + 1$ ($p = 0b1011$)
- initial state $0b111$



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

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

LFSR Iterable

Inputs:

- **Feedback Polynomial**:
  list of integers representing the degrees of the non-zero coefficients.
  Example: $[12, 6, 4, 1, 0]$ represents $P(x) = x^{12} + x^6 + x^4 + x^1 + 1$
- **LFSR state** (optional, default all bits to 1)
  bytes object representing the LFSR initial state. Note that bytes are converted into bits as big-endian (most significant bit first).
  Example: `b'e\xa0'` (`[0x65, 0xa0]`) represents the state 0110 0101 1010

LFSR Iterable

Attributes:

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

LFSR Iterable

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 Iterable

```python
class LFSR:
  ''' 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
```

Hints

There are several 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: Berlekamp-Massey Algorithm

Berlekamp-Massey Algorithm

Find the shortest LFSR for a given binary sequence.

- **Inputs**: 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 $2m \leq \tau$ **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 $P(x)$

Berlekamp-Massey Algorithm Task

- Implement the Berlekamp-Massey Algorithm
- Test the algorithm implementation
- Apply the Berlekamp-Massey Algorithm to the bit sequence stored in the file `binary_sequence.bin` to compute:
  - The polynomial of the shortest LFSR that can produce that sequence
  - The linear complexity of the bit sequence

Task 3: LFSR-based generator

Generator Type Assignment

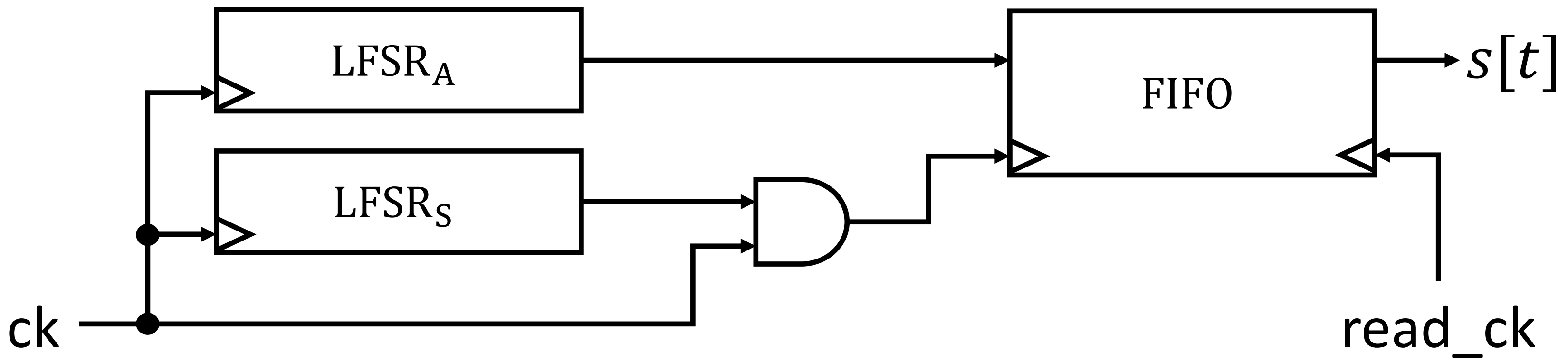
- Select *one of the two* possible generators according to the procedure described below

- Procedure:
  1. Transform the **name of your group** into a bytes object (big-endian)
  2. Transform the bytes object into a sequence of booleans (big-endian)
  3. Compute the **parity bit**

Parity bit	Assigned generator
0	Shrinking Generator
1	Self-Shrinking Generator

Shrinking Generator

The shrinking generator comprises $\mathrm{LFSR_A}$ that produces bits and $\mathrm{LFSR_S}$ that selects the produced bits. Since bit selection is irregular, a FIFO is necessary for a regular output rate.



Coppersmith, D., Krawczyk, H., Mansour, Y. (1994). The Shrinking Generator. In: Stinson, D.R. (eds) Advances in Cryptology — CRYPTO' 93. CRYPTO 1993. Lecture Notes in Computer Science, vol 773. Springer, Berlin, Heidelberg. doi:10.1007/3-540-48329-2_3

Shrinking Generator Class

- **Inputs**
  - Polynomials `polyA` and `polyS` (optional, default are
    - $P_A(x) = x^5 + x^2 + 1$
    - $P_S(x) = x^3 + x + 1$
  - `stateA` and `stateS` (optional, default all bits of the LFSR states at 1)
- **Attributes**
  - `lfsrA`: the LFSR class instance for $\mathrm{LFSR_A}$.
  - `lfsrS`: the LFSR class instance for $\mathrm{LFSR_S}$.
  - `output`: boolean storing the last produced output bit

Shrinking Generator Task

- Implement the Shrinking Generator Class

- Testing the functionality of the implemented class
  - Test that all methods and attributes work as expected

- Decrypt the ciphertext `chipertext_shrinking.bin` given:
  - $P_A(x) = x^{16} + x^{15} + x^{12} + x^{10} + 1$, stateA: `b'\xc5\xd7'`
  - $P_S(x) = x^{24} + x^{11} + x^{5} + x^{2} + 1$, stateS: `b'\x14\x84\xf8'`

Shrinking Generator Template

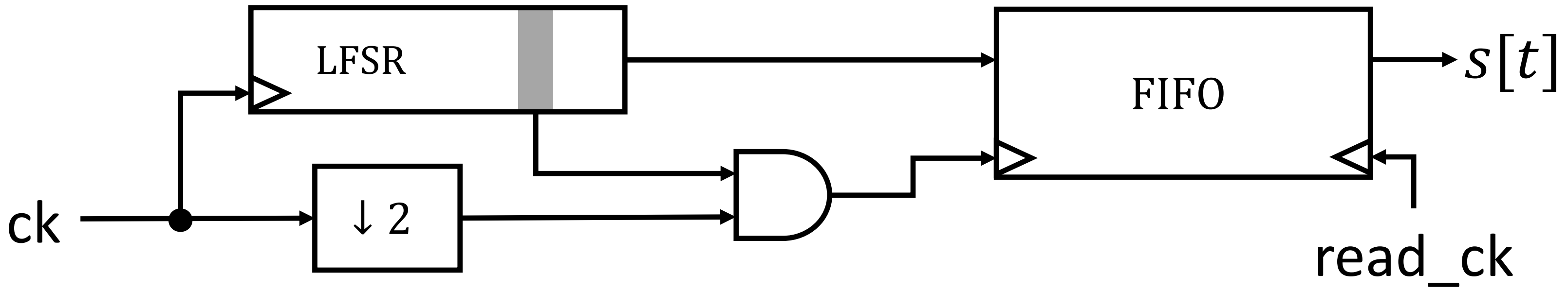
```
class ShrinkingGenerator:
  ''' class docstring '''
  # algorithm implementation
  def __init__(self, polyA=None, polyS=None, stateA=None, stateS=None):
    ''' init docstring '''
    self.lfsrA = LFSR(...)
    self.lfsrS = LFSR(...)
    self.output = ...

def __iter__(self):
    return self

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

Self-Shrinking Generator

The Self-Shrinking Generator is composed by a single LFSR that produce bits and selects them depending on the value of the internal state. A decimator and a FIFO are necessary for a regular output rate.



Meier, W., Staffelbach, O. (1995). The self-shrinking generator. In: De Santis, A. (eds) Advances in Cryptology — EUROCRYPT'94. EUROCRYPT 1994. Lecture Notes in Computer Science, vol 950. Springer, Berlin, Heidelberg. doi:10.1007/BFb0053436

Self-Shrinking Generator Class

- **Inputs**
  - Polynomial `poly` (optional, default is $P(x) = x^5 + x^2 + 1$)
  - Index of the selection bit `selection_bit` (optional, default 3)
  - State `state` (optional, default all bits of the LFSR state at 1)
- **Attributes**
  - `lfsr`: the LFSR class instance for LFSR.
  - `sbit`: integer storing the index of the selection bit.
  - `output`: boolean storing the last produced output bit

Self-Shrinking Generator Task

- Implement the Self-Shrinking Generator Class

- Testing the functionality of the implemented class
  - Test that all methods and attributes work as expected

- Decrypt the ciphertext `chipertext_selfshrinking.bin` given:
  - $P(x) = x^{32} + x^{16} + x^{7} + x^{2} + 1$
  - Selection bit: 4
  - state: `b'mJ\x9by'`

Self-Shrinking Generator Template

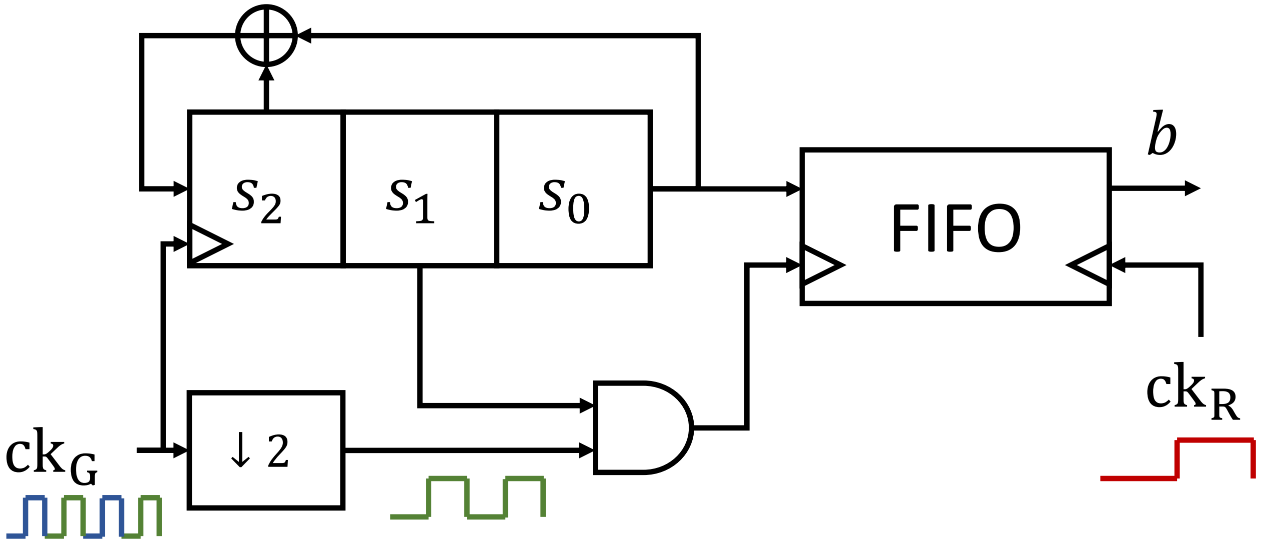
```
class SelfShrinkingGenerator:
  ''' class docstring '''
  # algorithm implementation
  def __init__(self, poly=None, selection_bit=None, state=None):
    ''' init docstring '''
    self.lfsr = LFSR(...)
    self.sbit = ...
    self.output = ...

def __iter__(self):
    return self

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

Example of a Self-Shrinking Generator

- polynomial = $x^3 + x + 1$, Selection bit = 1, initial state = $0b111$



s	s_1	s_0	b	s	s_1	s_0	b
111 (7)	1	1	-	100 (4)	0	0	-
011 (3)	1	1	-	110 (6)	1	0	-
101 (5)	0	1	-	111 (7)	1	1	1
010 (2)	1	0	-	011 (3)	1	1	-
001 (1)	0	1	-	101 (5)	0	1	-
100 (4)	0	0	-	010 (2)	1	0	-
110 (6)	1	0	0	001 (1)	0	1	-
111 (7)	1	1	-	100 (4)	0	0	-
011 (3)	1	1	1	110 (6)	1	0	0
101 (5)	0	1	-	111 (7)	1	1	-
010 (2)	1	0	0	011 (3)	1	1	1
001 (1)	0	1	-	101 (5)	0	1	-

Bonus Task: Statistical Tests

Statistical Tests

Binary sequences must resemble a truly random sequence to be unpredictable.

How to test randomness of a binary sequence?

NIST proposes a **statistical test suite**. Each test assesses the presence of a specific *pattern* which would indicate that the sequence is not random.

National Institute of Standards and Technology (NIST), "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications", Special Publication 800-22 r1a, April 2010

Statistical Test Suite

NIST Test Suite consists of 15 tests to assess the randomness of arbitrarily long binary sequences:

- The Frequency (Monobit) Test
- Frequency Test within a Block
- The Runs Test
- Tests for the Longest-Run-of-Ones in a Block
- The Binary Matrix Rank Test
- The Discrete Fourier Transform (Spectral) Test
- The Non-overlapping Template Matching Test
- The Overlapping Template Matching Test
- Maurer's "Universal Statistical" Test
- The Linear Complexity Test
- The Serial Test
- The Approximate Entropy Test
- The Cumulative Sums (Cusums) Test
- The Random Excursions Test
- The Random Excursions Variant Test.

Statistical Test Suite

NIST Test Suite consists of 15 tests to assess the randomness of arbitrarily long binary sequences:

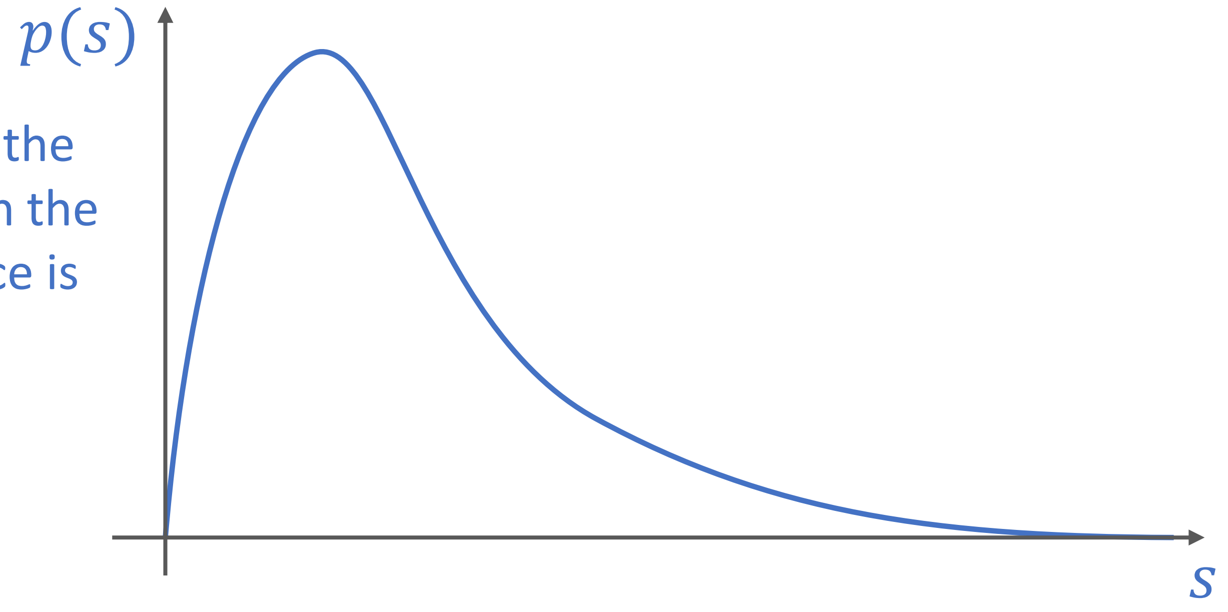
- The Frequency (Monobit) Test
- Frequency Test within a Block
- The Runs Test
- Tests for the Longest-Run-of-Ones in a Block
- The Binary Matrix Rank Test
- The Discrete Fourier Transform (Spectral) Test
- The Non-overlapping Template Matching Test
- The Overlapping Template Matching Test
- Maurer's "Universal Statistical" Test
- The Linear Complexity Test
- The Serial Test
- The Approximate Entropy Test
- The Cumulative Sums (Cusums) Test
- The Random Excursions Test
- The Random Excursions Variant Test.

Statistical Test

A statistical test takes a binary sequence (and optionally other parameters) and returns a binary outcome PASS/FAIL. In general, a test consists of three phases:

- Compute a **statistic**
  - A statistic is any quantity computed from the binary sequence
- Compute the $\boldsymbol{p}$**-value**
  - the probability to compute a statistic at least as extreme as the one actually computed, under the assumption that the binary sequence is random.
- Compare the p-value with the **significance level** of the test (α)
  - the probability of concluding that a truly random sequence is not random.

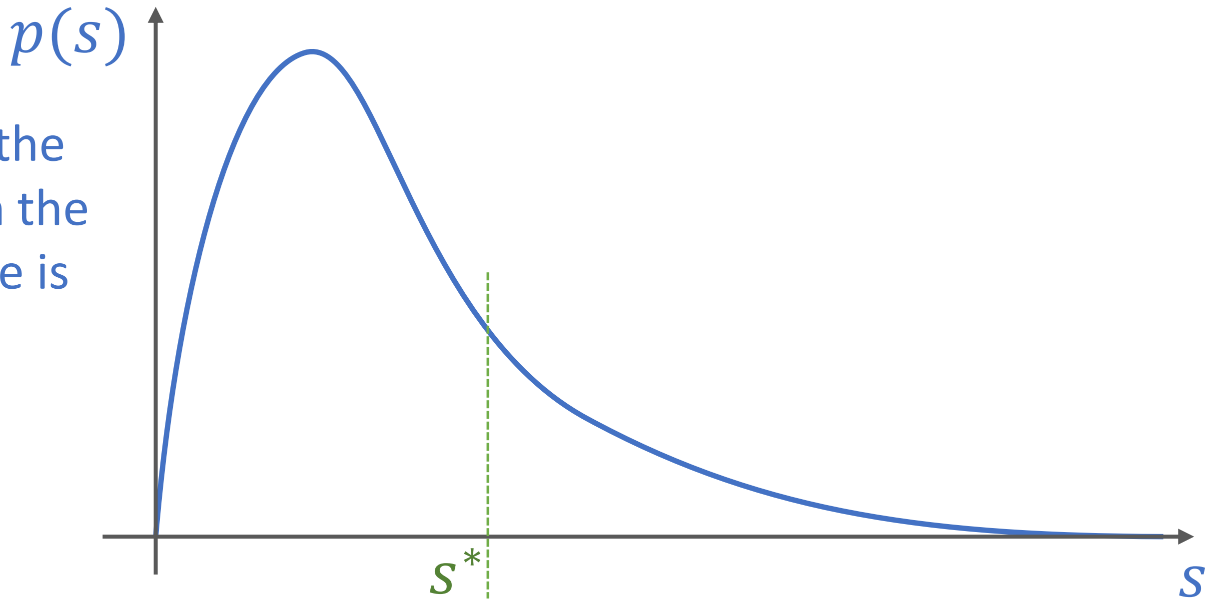
p-value and α



Distribution of the statistic s when the binary sequence is truly random

$\boldsymbol{p}$-value and $\boldsymbol{\alpha}$

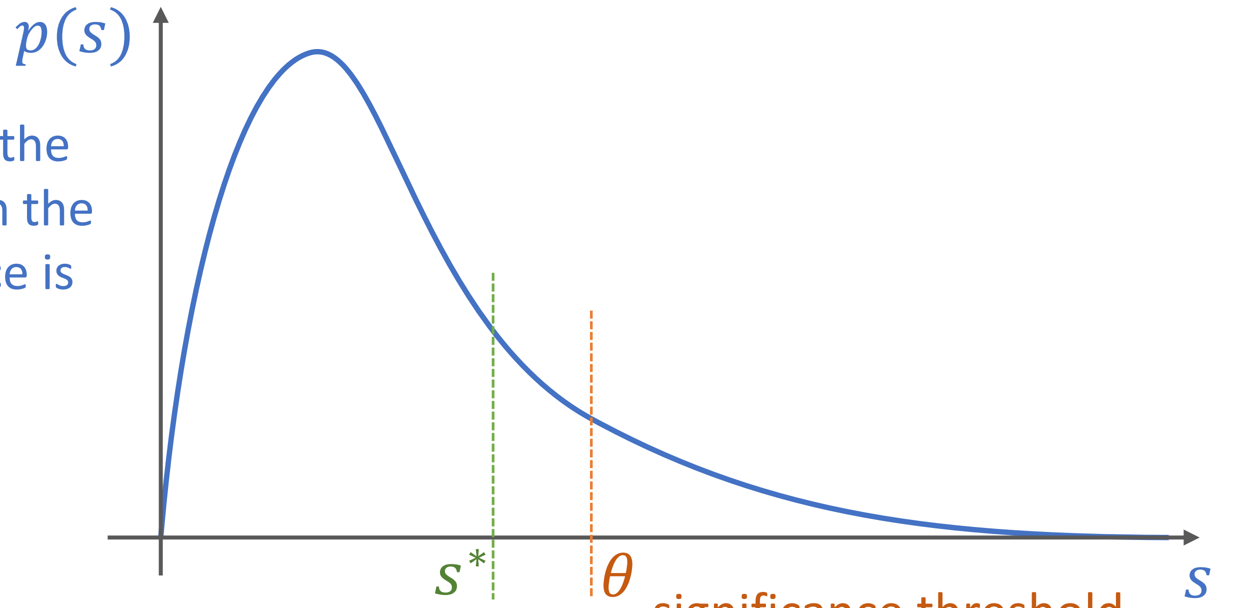
Distribution of the statistic s when the binary sequence is truly random



statistics computed from the binary sequence

p-value and α

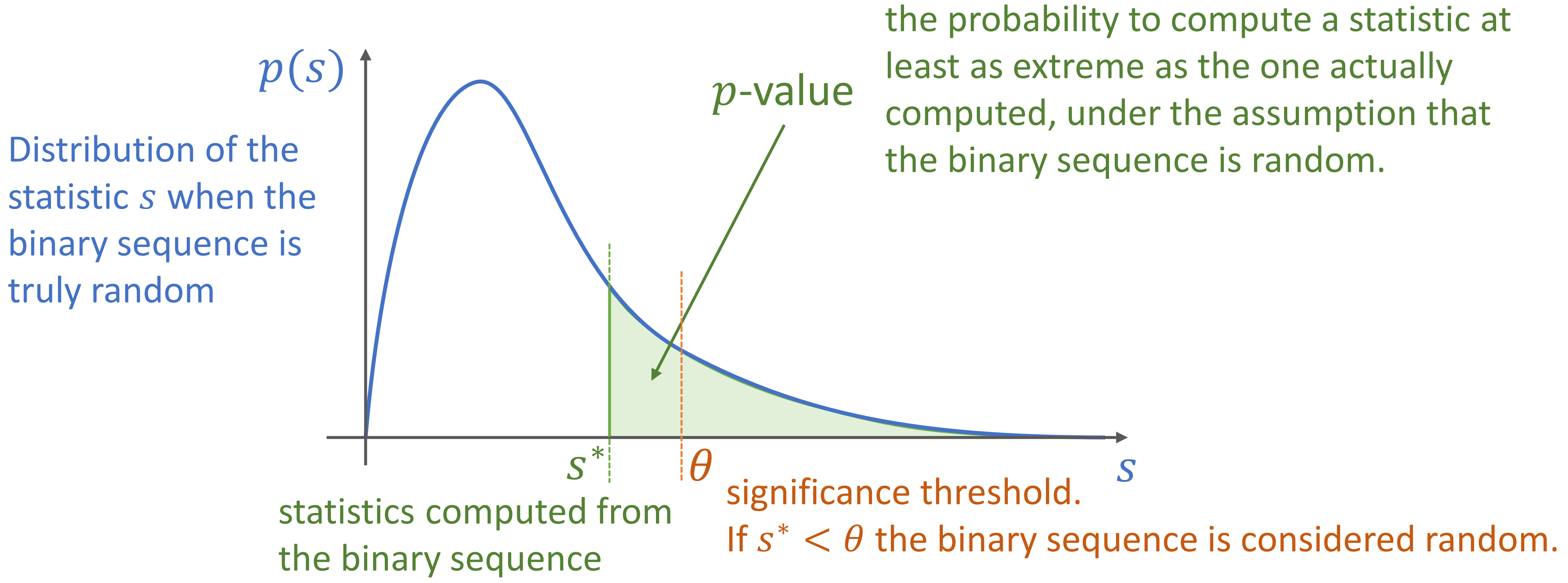
Distribution of the statistic s when the binary sequence is truly random



statistics computed from the binary sequence

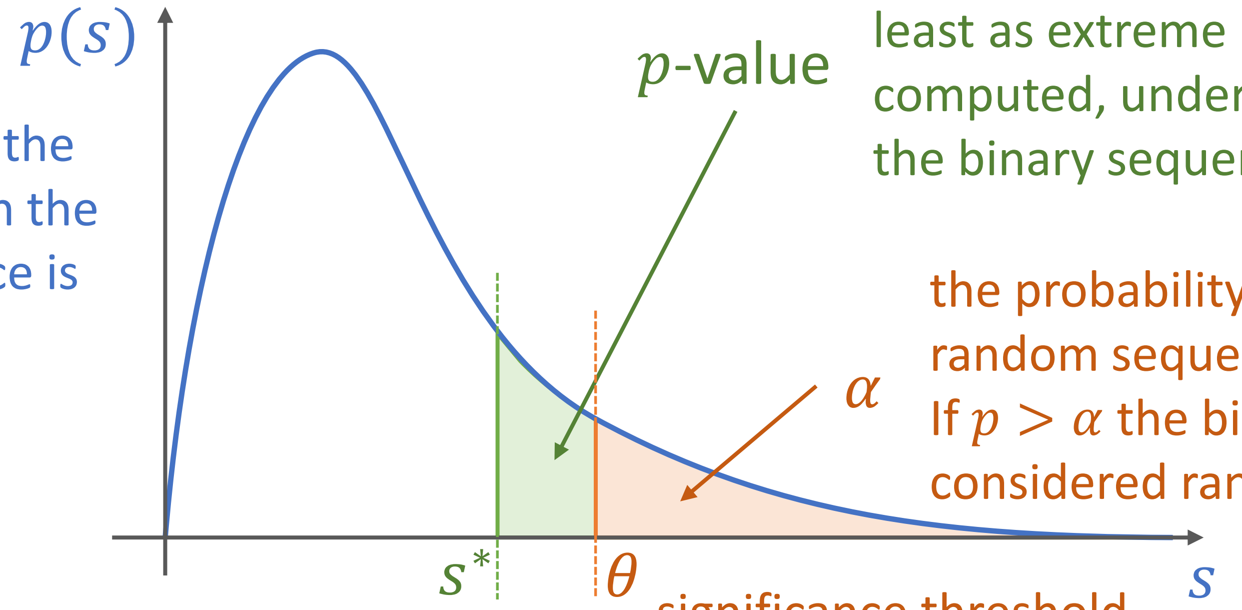
significance threshold.
If $s^* < \theta$ the binary sequence is considered random.

$\boldsymbol{p}$-value and $\boldsymbol{\alpha}$



$\boldsymbol{p}$-value and $\boldsymbol{\alpha}$

Distribution of the statistic s when the binary sequence is truly random



the probability to compute a statistic at least as extreme as the one actually computed, under the assumption that the binary sequence is random.

the probability of concluding that a truly random sequence is not random.
If $p > \alpha$ the binary sequence is considered random.

statistics computed from the binary sequence

significance threshold.
If $s^* < \theta$ the binary sequence is considered random.

The Frequency (Monobit) Test

determine whether the number of ones and zeros in a sequence $\{b_t\}_{t=1}^{n}$ are approximately the same as would be expected for a truly random sequence. ($n > 100$ is recommended)

- Compute test statistic:

$$s = 2\sqrt{n} \cdot \left|\pi - \frac{1}{2}\right|, \qquad \pi = \frac{1}{n}\sum_{t=1}^{n} b_t$$

π indicates the proportion of ones in the i-th block.

- Compute p-value:

$$p = \operatorname{erfc}\left(\frac{s}{\sqrt{2}}\right)$$

***p*-value** is the probability of computing a test statistic $s^* \leq s$ when $\{b_t\}$ is truly random.

$\mathbf{erfc}(x)$ is the complementary error function and indicates the probability for a Normal random variable to have value out of the range $[-x, x]$.

The Frequency (Monobit) Test

- Compare the p-value with the *level of significance* of the test α:

$$\begin{cases} p > \alpha & \Rightarrow & \{b_t\} \text{ is random} \\ p \leq \alpha & \Rightarrow & \{b_t\} \text{ is not random} \end{cases}$$

α is the significance level of the test and represents the probability of concluding that a truly random sequence is not random (Type I error).

Example:

$$\{b_t\} = \{1101001\}, \qquad \alpha = 0.01$$

$$s = \frac{|2 \cdot 4 - 3|}{\sqrt{7}} \sim 0.38 \;\Rightarrow\; p \sim 0.59 > 0.01 \;\Rightarrow\; \{b_t\} \text{ is random}$$

Frequency Test within a Block

determine whether the frequency of ones in an M-bit block of a sequence $\{b_t\}_{t=1}^{n}$ is approximately $M/2$.

- Recommendations: $n > 100$, $M \geq 20$, $M < n/100$, $N < 100$

- Split sequence into $N = \lfloor n/M \rfloor$ non-overlapping blocks.
- Compute test statistic:

$$\chi^2 = 4M \sum_{i=1}^{N} \left(\pi_i - \frac{1}{2}\right)^2, \qquad \pi_i = \frac{1}{M} \sum_{j=1}^{M} b_{(i-1)M+j}$$

χ^2 is related with the sample variance of the π_is

π_i indicates the proportion of ones in the i-th block.

Frequency Test within a Block

- Compute p-value and compare it with the *level of significance* of the test α:

$$p = Q\left(\frac{N}{2}, \frac{\chi^2}{2}\right), \qquad \begin{cases} p > \alpha & \Rightarrow & \{b_t\} \text{ is random} \\ p \leq \alpha & \Rightarrow & \{b_t\} \text{ is not random} \end{cases}$$

$Q(a, x)$ is the regularized upper incomplete gamma function.

Example:

$$\{b_t\} = \{1100100100\ 0011111101\}, \qquad \alpha = 0.01, M = 10, N = 2$$

$$\chi^2 = 2 \Rightarrow p = Q(1,1) \sim 0.38 > 0.01 \Rightarrow \{b_t\} \text{ is random}$$

Runs Test

determine whether the number of runs of ones and zeros in a sequence $\{b_t\}_{t=1}^{n}$ is as expected. A run of length k is a sequence of k identical bits bounded before and after with a bit of the opposite value.

- Recommendations: $n > 100$

- Apply the frequency (monobit) test. If the sequence fails, the runs test fails.
- Compute test statistic:

$$v = \frac{1}{2n}\left(1 + \sum_{t=1}^{n-1} r_t\right), \qquad r_t = \overline{b_t \oplus b_{t+1}} = \begin{cases} 1 & \text{if } b_t = b_{t+1} \\ 0 & \text{if } b_t \neq b_{t+1} \end{cases}$$

Runs Test

- Compute p-value and compare it with the *level of significance* of the test α:

$$p = \operatorname{erfc}\left(\frac{1}{\sqrt{2}} \cdot \frac{|v - \pi(1-\pi)|}{\pi(1-\pi)}\right), \qquad \begin{cases} p > \alpha & \Rightarrow \quad \{b_t\} \text{ is random} \\ p \leq \alpha & \Rightarrow \quad \{b_t\} \text{ is not random} \end{cases}$$

Example:

$$\{b_t\} = \{11001001000011111101\}, \qquad \alpha = 0.01$$

- Frequency (monobit) test passed ($s = 0.44 \Rightarrow p = 0.66 > 0.01$)
- $v = 0.3, \pi = 0.55 \Rightarrow p \sim 0.83 > 0.01 \Rightarrow \{b_t\}$ is random

Bonus Task

- Implement a function for each of the first three tests in the NISTs statistical test suite:
  - `frequency_test(b) -> bool`
  - `block_frequency_test(b, M) -> bool`
  - `runs_test(b) -> bool`

in `scipy.special`, you can find the Python implementation of:

- erfc - complementary error function
- Q - regularized upper incomplete Gamma function

- Test your implementation by applying those functions to:
  - custom sequences that make tests fail
  - binary sequences generated with the previously implemented generators

Deadline

Tuesday, April 23rd at 12PM (noon)