# Module 8: Linear Feedback Shift Registers

# JHU EP 606.206 - Introduction to Programming Using Python

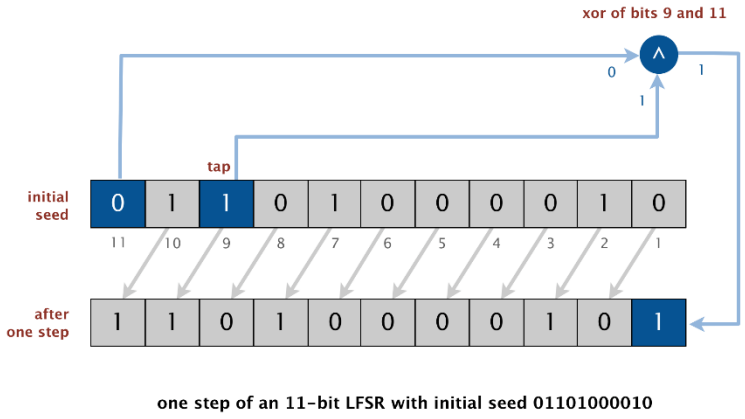
## Introduction

In this assignment you will use a core Python and a library called Pillow which is a Git fork of the now defunct PIL: Python Image Library. Pillow is a Python library built for image processing and you will use it, along with a custom linear feedback shift register class, to encrypt and decrypt an image.

**Skills**: classes, binary operators, functions, modules, external libraries, pair-programming

## What is a Linear Feedback Shift Register?

A Linear Feedback Shift Register (LFSR) is a pseudo-random number generator. It is defined by its length (N), its initial configuration (seed), and a special element within the LFSR known as a "tap".




one step of an 11-bit LFSR with initial seed 01101000010

(Image courtesy of Princeton University CS)

In the example above, the LFSR has N = 11, seed = 01101000010, and tap = 9. Note that the "cells" of an LFSR are numbered from right-to-left. Given the initial configuration, a 1-bit shift left is performed. The leftmost bit (0), which is shifted off, is XOR'ed with the element at the "tap" position (1): 0 XOR 1 = 1. The result of the XOR is then placed in the rightmost position, filling the open position that results from the shift left. The resulting state of the LFSR is now: 11010000101. To generate another pseudo-random number we perform the same process again: shift left, take the leftmost, shifted off bit, 1, and XOR it with the bit in the tap position: 0. 1 XOR 0 = 1 which we now place in the rightmost position resulting in: 10100001011.

LFSR's can be used for encryption. For example, let's say we want to encrypt an image. An image is a 2D array of pixels where each pixel is made up of a red, green, and blue component. Each component has a value from 0-255 which is, by design, exactly 8-bits long. So, if we wanted to encrypt an image, pixel-by-pixel, an 8-bit LFSR (N = 8) would do the trick perfectly.

## Encrypting an Image with a LFSR

We will now use our LFSR to encrypt an image using the following scheme:

Recall an image is defined as a 2D array of pixels. Each pixel has three components (red, green, blue) that range from 0-255. This is convenient because values 0-255 can be represented with exactly 8-bits.

A single white pixel will be of the form: (255, 255, 255)

A tiny white square consisting of 4 pixels will be of the form:

[[(255, 255, 255), (255, 255, 255)], [(255, 255, 255), (255, 255, 255)]]

To encrypt an image using an LFSR start with an 8-bit LFSR with initial values for a seed (this is our "password") and a tap. Calculate the first "random" value using your LFSR by calling the `step()` method.

Next, access the red component of the first pixel in the image. Recall you have a 2D array of pixels, where each element in the array is a list/tuple that contains the red, green, and blue components of the pixel. Calculate the XOR between your random value and the pixel's red component. This result is the "encrypted" value of the first pixel's red component. Repeat this process for the green and blue components making sure to create a new random value by using the `step()` method each time. Replace/update the existing pixel's red, green, and blue value with the newly encrypted value. You've successfully encrypted the first pixel!

- Initial LFSR (password): 10011010 (154), tap = 4
- 1st random LFSR value (LFSR after 1 call to 'step'): 00110100 (52)
- Sample 2D image array (only the red component of first pixel is shown):

  [[(160, G, B), (R, G, B), … (R, G, B)]

  ...

  [(R, G, B), (R, G, B), … (R, G, B)]]

- 10100000 (160) XOR 00110100 (52) = 10010100 (148), this encrypted value becomes the new value for the first pixel's red component. Do the same thing for the green and blue components and repeat this process for all remaining pixels to encrypt the entire image (don't forget to update your LFSR by using `step()` for each R, G, B value of each pixel!):

  [[(148, G', B'), (R', G', B'), … (R', G', B')]

  ...

  [(R', G', B'), (R', G', B'), … (R', G', B')]]

Once you've replaced all the original values with their encrypted values, you can turn your 2D array back into an image. Fun fact: to decrypt this image, all you have to do is read in the encrypted image and perform the EXACT same process you did to encrypt it! That is, if you encrypt an image with an LFSR, you can decrypt it by following the same process as long as your seed and tap are the same, so no extra programming is required to be able to decrypt your image; you'll just run the same code a second time on the encrypted image resulting from the first execution.

## Programming Assignment Part #1: Building an LFSR

```
class LFSR:
    # create an LFSR with initial state 'seed' and tap 'tap'
    __init__(self, seed: string, tap: int):

    # return the bit at index 'i'
    bit(self, i: int):

    # execute one LFSR iteration and return new (rightmost) bit as an int
    # you will find the binary XOR operator useful here
    step(self):

    # return string representation of the LFSR, ex: 01001010
    __str__(self):

# your executable code that invokes LFSR
main():

if __name__ == "__main__":
    main()
```

Your main method should:

- Create 5 LFSR's:
  - seed = 0110100111, tap = 2
  - seed = 0100110010, tap = 8
  - seed = 1001011101, tap = 5
  - seed = 0001001100, tap = 1
  - seed = 1010011101, tap = 7
- Iterate each LFSR once using the `step()` method to produce a new random number
- Print the resulting states of each LFSR and the value of the new bit in the following format
  - [LFSR] [new bit]
    - Ex: 01100001111 1
- For reference, here are the outputs from our solution after calling `step()` one time on each of the 5 LFSR's above:

1101001111 1

1001100100 0

0010111010 0

0010011000 0

0100111011 1

## Programming Assignment Part #2: Encrypting an Image

```
from lfsr import LFSR # import your LFSR class for use in ImageEncrypter

class ImageEncrypter:
    # initialize an ImageEncrypter object with an LFSR and image file name
    __init__(self, lfsr: LFSR, file_name: str)

    # open the image specified by 'file_name' in your constructor
    # you will find the Image.open method useful here
    def open_image(self):

    # calls open_image()
    # converts the image to a 2D array of R, G, B triples
    # you will find the Image.load method useful here
    def pixelate(self):

    # encrypts the 2D pixelated "image" returned from pixelate()
    # returns the encrypted 2D array
    # you will find the binary XOR operator useful here
    def encrypt(self):

    # converts the encrypted 2D pixelated image into an image
    # and names it <file_name>_transform.png
    # you will find the Image.save method useful here
    def save_image(self, file_name: str)

# your executable code that invokes ImageEncrypter and encrypts/decrypts an
# image and saves the result to a file
main():

if __name__ == "__main__":
    main()
```

Your main() method should:

- Import the Image module from the Pillow library
  - You will use Pillow to read in files and save the encrypted/decrypted results
- Create an ImageEncrypter that:
  - imports your `LFSR` class from your `lfsr` module and creates an LFSR object with:
    - seed = 10011010
    - tap = 5
- Opens a file called 'football.png' which you can find on Blackboard (Image.open)
- Converts the input image into a 2D array of pixels (Image.load)
- Encrypts the image using the LFSR/XOR scheme shown above
- Writes the encrypted image back out to a file called 'football_encrypted.png' (Image.save)
- Reads the encrypted image and saves the unencrypted image to a file called 'football_decrypted.png'

You will run your encrypter twice: once to encrypt the image and once to decrypt it. Please verify the decrypted image matches the originally downloaded image and provide screenshots.

For reference, here are the values of the first 4 pixels, the LFSR values for the R, G, and B component of each of the 4 pixels, as well as the resulting encrypted pixel from my solution:

```
Original pixel: (255, 255, 255)
LFSR value 52
LFSR value 105
LFSR value 210
New pixel:[203, 150, 45]
Original pixel: (255, 255, 255)
LFSR value 164
LFSR value 73
LFSR value 146
New pixel:[91, 182, 109]
Original pixel: (255, 255, 255)
LFSR value 36
LFSR value 72
LFSR value 144
New pixel:[219, 183, 111]
Original pixel: (255, 255, 255)
LFSR value 32
LFSR value 64
LFSR value 128
New pixel:[223, 191, 127]
```

Original image (***football.png***):

Encrypted image (***football_transform.png***; we've commonly seen encrypted images that look "fuzzy" but still show the general outline of the football, that is an acceptable output and points will not be deducted):

Decrypted image (***football_transform_transform.png***):

## Group-work Guidelines

From our perspective, **working with a partner this week will be extremely valuable** if you choose to go that route. Please be advised we don't think working in a group makes the Assignment any more/less difficult, it just shifts where the complexity comes from. There are benefits to developing all of the code on your own, but you'll spend a bit more time programming. With a partner, you'll spend a bit less time on the keyboard but a little more time integrating each other's code (something that comes up in a professional software development environment all the time). Our hope is that you can use this Assignment to get a feel for what software development in the workplace actually looks like.

Guidelines:

- You may work with a single partner on this Assignment
- Please include your partner on any email/Teams-based communications with the instructors
- Each member is responsible for completing as close to 50% of the work as possible
- Each partner will submit the same code but should submit their own README
    - README's may also be written jointly with the exception of the **Partner Collaboration** section outlined in (5) of the Deliverables section
- Deductions will be taken in cases where there is a significant mismatch in the portion of work each of the partners has completed

A natural way to split-up the work may be to have one partner build the LFSR and the other build the ImageEncrypter. For example, when we built this Assignment, Alan wrote the LFSR first independently of me. I grabbed his lfsr.py and imported it into image_encrypter.py and, because Alan followed the pseudocode for the LFSR class below precisely, I immediately knew which methods were available for me to use and what they would return to me. I didn't have to look at a single line of his code to know how to use it because we both worked off of the same "blueprint". This is similar to how you may have used the `sqrt()` function from Python's `math` module in the past; you don't need to know how it works internally to be able to use it successfully, and this is the beauty behind the idea of "abstraction".

Alternatively, you may both choose to work jointly on each of the two components of the Assignment, and that's fine too. Either way, **please be sure you understand both parts of the code (LFSR and ImageEncrypter)**.

## Deliverables

### readme.txt

So-called "read me" files are a common way for developers to leave high-level notes about their applications. Here's an example of a README file for the Apache Spark project. They usually contain details about required software versions, installation instructions, contact information, etc. For our purposes, your readme.txt file will be a way for you to describe the approach you took to complete the assignment so that, in the event you may not quite get your solution working correctly, we can still award credit based on what you were trying to do. Think of it as the verbalization of what your code does (or is supposed to do). Your readme.txt file should contain the following:

1. **Name**: Your name and JHED ID
2. **Module Info**: The Module name/number along with the title of the assignment and its due date
3. **Approach**: a detailed description of the approach you implemented to solving the assignment. Be as specific as possible. If you are sorting a list of 2D points in a plane, describe the class you used to represent a point, the data structures you used to store them, and the algorithm you used to sort them, for example. The more descriptive you are, the more credit we can award in the event your solution doesn't fully work.
4. **Known Bugs**: describe the areas, if any, where your code has any known bugs. If you're asked to write a function to do a computation but you know your function returns an incorrect result, this should be noted here. Please also state how you would go about fixing the bug. If your code produces results correctly you do not have to include this section.
5. **Partner Collaboration**: if you worked with a partner on this assignment, please provide their name and section. We'd also like for you to include 1-2 substantive paragraphs detailing which specific parts of the assignment you worked on and any complexities you encountered working on the Assignment with a partner.

Please submit your ***lfsr.py, image_encrypter.py*** source code files along with a PDF file containing screenshots of your outputs in a PDF called JHEDID_mod8.pdf (ex: jkovba1_mod8.pdf). Please do not ZIP your files together.

Recap:

1. readme.txt
2. lfsr.py
3. image_encrypter.py
4. A PDF containing a screenshot of the output from lfsr.py showing the output of the 5 LFSR's after 1 step, the encrypted version of football.png, and the decrypted version of football.png (which should match the original image before it was encrypted).

**Please let us know if you have any questions via Teams or email!**