nLFSR

一開始的想法就是先拿到 64 個答案（全猜 0 去看 money 差多少），就可以拿掉 64 個方程式，接著用 sage 去解就可以拿掉 initial state，把狀態跟 server 的 LFSR 同步後，就可以拿到 FLAG 了。

解到要找 companion matrix 的時候，發現 server.py 的 LFSR 跟上課教的 LFSR 長不太一樣，因為 state 會 xor poly，這樣會改到不只最後一個 register 的狀態，這邊就有點卡關，於是開始去網路找其他資料來看。

首先我去 wikipedia (https://en.wikipedia.org/wiki/Linear-feedback_shift_register) 看之後才發現，原來上課教的是 Fibonacci LFSRs，是對 hardware 比較友善的實作，作業裡面的是 Galois LFSRs，是對 software 比較友善的作法。兩種 LFSR 在相同的 taps 下都能夠生成相同 output stream，但是輸出會有時間差。除此之外，Galois LFSRs 的 taps 方向要跟 Fibonacci LFSRs 相反，不然順序會是倒過來的。

> ... which is also known as **modular**, **internal XORs**, or **one-to-many LFSR**, is an alternate structure that can generate the same output stream as a conventional LFSR (but offset in time). [3]

因為我想要找的是 Galois LFSRs 的 companion matrix 定義，但是看到這裡我還沒什麼頭緒，決定去翻下面 References [3] 的書來看。

> Press, William; Teukolsky, Saul; Vetterling, William; Flannery, Brian (2007).*Numerical Recipes: The Art of Scientific Computing, Third Edition*. [Cambridge University Press](). p. 386. [ISBN]() [978-0-521-88407-5]()

在書中 Chapter 7.5 中後段有說明到如何驗證給定一個 companion matrix (也就是定義一個 LFSR) 是否為 full-period 的 generator (週期為 $2^n - 1$)，在後方還有提到如果有一個 companion matrix M 是 full-period generator，那有一些矩陣也會是 full-period generator，第一個是 M^{-1} 也就是可以將下一個 state 推回目前這個 state 的 companion matrix，另外一個是 M^T 以及 $(M^T)^{-1}$，然後就會發現 M^T 就是 Galois LFSRs 的 companion matrix 定義！驗證如下：

如果 LFSR 的定義是：

$$LFSR = \begin{cases} a_1^{'} = (\sum_{j=1}^{n-1} c_j a_j) + a_n \\ a_i^{'} = a_{i-1}, i = 2 \dots n \end{cases}$$

因為是線性，可以轉換成矩陣形式，以下是驗證 Fibonacci LFSR:

$$\underbrace{\begin{bmatrix} c_1 & c_2 & \dots & c_{n-2} & c_{n-1} & 1 \\ 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 & 0 \\ . & . & & . & . & . \\ . & . & & . & . & . \\ 0 & 0 & \dots & 1 & 0 & 0 \\ 0 & 0 & \dots & 0 & 1 & 0 \end{bmatrix}}_{M} \underbrace{\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ . \\ . \\ a_{n-1} \\ a_n \end{bmatrix}}_{S_0} = \underbrace{\begin{bmatrix} \sum_{j=1}^{n-1} c_j a_j + a_n \\ a_1 \\ a_2 \\ . \\ . \\ a_{n-2} \\ a_{n-1} \end{bmatrix}}_{S_1}$$

$$
\underbrace{\begin{bmatrix} 0 & 1 & \dots & 0 & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 & 0 \\ . & . & & . & . & . \\ . & . & & . & . & . \\ 0 & 0 & \dots & 0 & 1 & 0 \\ 0 & 0 & \dots & 0 & 0 & 1 \\ 1 & c_1 & c_2 & \dots & c_{n-2} & c_{n-1} \end{bmatrix}}_{M^{-1}}
\underbrace{\begin{bmatrix} \sum_{j=1}^{n-1} c_j a_j + a_n \\ a_1 \\ a_2 \\ . \\ . \\ a_{n-2} \\ a_{n-1} \end{bmatrix}}_{S_1}
=
\underbrace{\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ . \\ . \\ a_{n-1} \\ \cancel{\sum_{j=1}^{n-1} c_j a_j} + a_n + \cancel{\sum_{j=1}^{n-1} c_j a_j} \end{bmatrix}}_{S_0}
$$

把 M 做轉置後，就可以看出 Galios LFSR 的計算行為了：

$$
\underbrace{\begin{bmatrix} c_1 & 1 & \dots & 0 & 0 & 0 \\ c_2 & 0 & \dots & 0 & 0 & 0 \\ c_3 & 0 & \dots & 0 & 0 & 0 \\ . & . & & . & . & . \\ . & . & & . & . & . \\ c_{n-1} & 0 & \dots & 0 & 0 & 1 \\ 1 & 0 & \dots & 0 & 0 & 0 \end{bmatrix}}_{M^T}
\underbrace{\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ . \\ . \\ a_{n-1} \\ a_n \end{bmatrix}}_{S_0}
=
\underbrace{\begin{bmatrix} a_1 c_1 + a_2 \\ a_1 c_2 + a_3 \\ a_1 c_3 + a_4 \\ . \\ . \\ a_1 c_{n-1} + a_n \\ a_1 \end{bmatrix}}_{S_1}
$$

$$
\underbrace{\begin{bmatrix} 0 & 0 & \dots & 0 & 0 & 1 \\ 1 & 0 & \dots & 0 & 0 & c_1 \\ 0 & 1 & \dots & 0 & 0 & c_2 \\ . & . & & . & . & . \\ . & . & & . & . & . \\ 0 & 0 & \dots & 1 & 0 & c_{n-2} \\ 0 & 0 & \dots & 0 & 1 & c_{n-1} \end{bmatrix}}_{(M^T)^{-1}}
\underbrace{\begin{bmatrix} a_1 c_1 + a_2 \\ a_1 c_2 + a_3 \\ a_1 c_3 + a_4 \\ . \\ . \\ a_1 c_{n-1} + a_n \\ a_1 \end{bmatrix}}_{S_1}
=
\underbrace{\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ . \\ . \\ a_{n-1} \\ a_n \end{bmatrix}}_{S_0}
$$

有了這樣的背景知識就可以開始解後續的問題，把 server.py 提供的 `ploy` 轉成 companion matrix (方向在左邊) 後，就可以解出 64 個方程式推回 initial state，後續就是比較瑣碎的事，詳細的可以看附件的 script。

另外，我也有驗證 wiki 說的 offset，在 `lfsr-study.ipynb`。