密碼工程 quiz4
111550035 蔡昀錚

Problem 1

(a)

To be a primitive polynomials over GF(2), it need to satisfy these conditions:
(1) The polynomial cannot be factored into the product of smaller non-trivial polynomials.
(2) The polynomial must generate a sequence with the maximum possible period, which is 2^n - 1, when n is the degree of polynomial. For 8-th degree, maximum cycle length = 2^8 - 1 = 255.
And for $x^8 + x^4 + x^3 + x^2 + 1$, it satisfy the above conditions, so it is a primitive polynomial.

(b)

Since it is primitive polynomial with 8-th degree, it has the maximum length = 2 ^ 8 - 1 = 255.

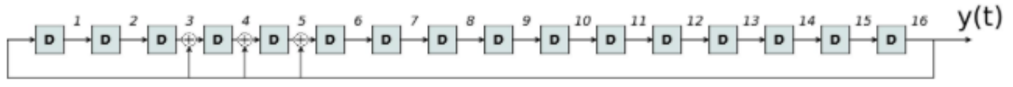
(c)

No, not all irreducible polynomials are primitive polynomials, but all primitive polynomials are irreducible. A primitive polynomial is an irreducible polynomial with an additional property.

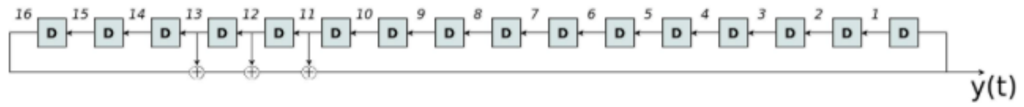
Problem 2

(a)

There are 2 kinds of LFSR, Galois and Fibonacci. They have different structure:
Source: https://www.01signal.com/other/lfsr-galois-fibonacci/
Galois:



Fibonacci:



Two method will generate different key and ciphertext, I output both of them in problem2.py.
Here we set initial state = [0, 0, 0, 0, 0, 0, 0, 1] has position [x8, x7, …, x2, x1].

(b)

Yes.

The assertion about the most significant bit (MSB) being zero in ASCII codes for capital letters A to Z (which range from binary `01000001` to `01011010`) is correct. Because the MSB is always zero for these letters, the MSB of each byte encrypted with an XOR operation using a Linear Feedback Shift Register (LFSR) will reveal the MSB of the corresponding byte in the keystream. This happens because XORing a bit with 0 leaves it unchanged.

Therefore, if you have a sequence of encrypted ASCII capital letters, you can indeed extract part of the keystream. Each 8th bit of the ciphertext can be used to construct a binary sequence that represents the keystream's MSBs.

Regarding finding out the characteristic polynomial of an LFSR system by solving a system of linear equations, in theory, it is possible under certain conditions. If you have enough consecutive bits of the LFSR output (keystream), you could set up a system of equations based on the LFSR's feedback function, which is defined by its characteristic polynomial. This is essentially how the Berlekamp-Massey algorithm works, which can find the shortest LFSR and its characteristic polynomial capable of producing a given binary sequence.

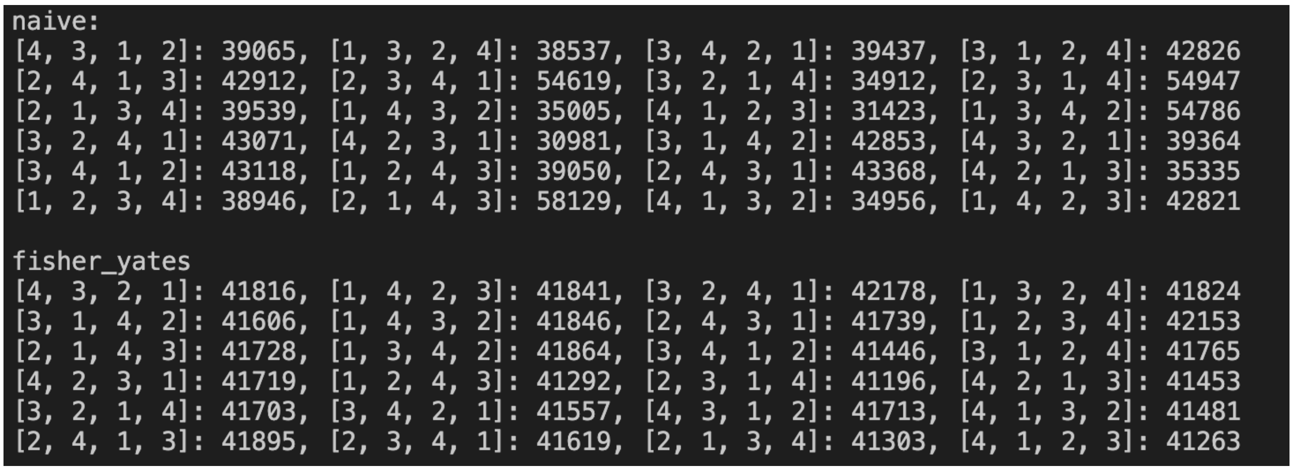
To solve for the characteristic polynomial, you need a sequence of output bits at least twice as long as the degree of the LFSR. This sequence is then used to form a system of linear equations where the tap coefficients are the unknowns. By solving this system, you can potentially determine the feedback taps of the LFSR, and thus its characteristic polynomial. However, in practice, this is not a trivial task and requires careful implementation of the algorithm to successfully recover the LFSR settings from the keystream.

(c)

Code is in problem2.py. For problem 1, it produce a 2368-bits binary sequence, so we get 2360 linear equations (distinct equation # should > 2 *n), here is an example: x1*1 + x2*0 + x3*1 + x4*1 + x5*1 + x6*0 + x7*1 + x8*0 mod 2 = 1. Since it module 2, the coefficient is wether 0 or 1 (other number have same effect as 0 or 1). As a result we can have a matrix multiplication like this:
A_(2380 * 1) = B_(2380 * 8) * C(8 * 1) mod 2. A B is known, so I try every possible combination of C (about 2^7 - 1 combinations, since 0 bit must be 1). Finally I get the characteristic polynomial $= x^8 + x^4 + x^3 + x^2 + 1$.

Problem 3

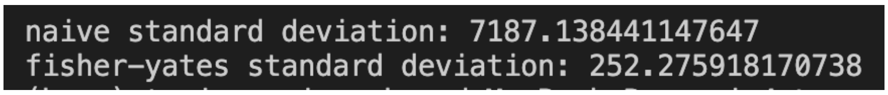
(a)



```
naive:
[4, 3, 1, 2]: 39065, [1, 3, 2, 4]: 38537, [3, 4, 2, 1]: 39437, [3, 1, 2, 4]: 42826
[2, 4, 1, 3]: 42912, [2, 3, 4, 1]: 54619, [3, 2, 1, 4]: 34912, [2, 3, 1, 4]: 54947
[2, 1, 3, 4]: 39539, [1, 4, 3, 2]: 35005, [4, 1, 2, 3]: 31423, [1, 3, 4, 2]: 54786
[3, 2, 4, 1]: 43071, [4, 2, 3, 1]: 30981, [3, 1, 4, 2]: 42853, [4, 3, 2, 1]: 39364
[3, 4, 1, 2]: 43118, [1, 2, 4, 3]: 39050, [2, 4, 3, 1]: 43368, [4, 2, 1, 3]: 35335
[1, 2, 3, 4]: 38946, [2, 1, 4, 3]: 58129, [4, 1, 3, 2]: 34956, [1, 4, 2, 3]: 42821

fisher_yates
[4, 3, 2, 1]: 41816, [1, 4, 2, 3]: 41841, [3, 2, 4, 1]: 42178, [1, 3, 2, 4]: 41824
[3, 1, 4, 2]: 41606, [1, 4, 3, 2]: 41846, [2, 4, 3, 1]: 41739, [1, 2, 3, 4]: 42153
[2, 1, 4, 3]: 41728, [1, 3, 4, 2]: 41864, [3, 4, 1, 2]: 41446, [3, 1, 2, 4]: 41765
[4, 2, 3, 1]: 41719, [1, 2, 4, 3]: 41292, [2, 3, 1, 4]: 41196, [4, 2, 1, 3]: 41453
[3, 2, 1, 4]: 41703, [3, 4, 2, 1]: 41557, [4, 3, 1, 2]: 41713, [4, 1, 3, 2]: 41481
[2, 4, 1, 3]: 41895, [2, 3, 4, 1]: 41619, [2, 1, 3, 4]: 41303, [4, 1, 2, 3]: 41263
```

(b)
Use standard deviation to evaluate:



```
naive standard deviation: 7187.138441147647
fisher-yates standard deviation: 252.275918170738
```

Even the standard deviation is different every time it runs, Fisher-Yates obviously still have much better performance than Naive method (smaller deviation -> each combinations have more equal probability).

(c)

(1) In the Naive algorithm, each card is equally likely to end up in any position, but not every possible final arrangement of cards is equally likely. This is because the range from which random indices are chosen doesn't shrink as the algorithm progresses through the array. It does not account for the fact that some cards have already been shuffled, thus skewing the randomness.
(2) The Naive algorithm allows a card to be swapped multiple times, which can lead to certain positions being more likely to contain certain cards, particularly in small arrays. This is because later swaps can undo the randomness introduced by earlier swaps.
(3) Non-decreasing Range: Unlike the Fisher-Yates shuffle, which decreases the range of possible indices as it progresses (ensuring each position is swapped only once), the Naive algorithm always selects a random index from the full range. This means some positions may never be chosen, while others may be chosen multiple times.

The drawbacks of the Naive shuffle stem from its failure to reduce the range from which random indices are chosen as the algorithm iterates through the array, resulting in a non-uniform probability distribution of the permutations. The Fisher-Yates shuffle does not have these drawbacks because it ensures that each element is swapped exactly once, leading to each permutation being equally likely.