

Crypto Engineering Quiz 4

1.

a. Yes.

Since it can generate all the 255 polynomials, so it is primitive (calculated in problem1.py).

```
252 [1, 1, 0, 1, 1, 0, 0, 0]
253 [1, 0, 1, 0, 1, 1, 0, 1]
254 [1, 0, 0, 0, 1, 1, 1]
255 [1, 0, 0, 0, 1, 1, 1, 0]
The period of the sequence [1, 0, 0, 0, 1, 1, 1, 0, 1] is 255
```

b. As mentioned above, the maximum cycle length is 255.

c. No.

$X^8 + X^4 + X^3 + X^1 + 1$ is also irreducible, but its cycle length starting at 1 is 51, so it is not primitive.

```
50 [1, 1, 0, 0, 1, 0, 1, 1]
51 [1, 0, 0, 0, 1, 1, 0, 1]
The period of the sequence [1, 0, 0, 0, 1, 1, 0, 1, 1] is 51
```

Appendix: Description of problem1.py

To run the code: `python problem1.py`

I implemented three functions in the file, including convolution, shift, mod.

Convolution: it simple evaluate the polynomial for $x=2$ and return the value.

```
1 def convolution(x: list):
2     fact = 1
3     res = 0
4     for i in range(len(x) - 1, -1, -1):
5         res += x[i] * fact
6         fact *= 2
7     return res
```

Shift: concatenate a zero to the end of the coefficient list, performing a left shift.

```

10 def shift(x: list):
11     return x + [0]

```

Mod: perform exclusive-or to the first-n-terms while the length of coefficient list is at least n, where n is the length of the modulo polynomial, and remove the leading zeros of the coefficient list.

```

14 def mod(x: list, p: list):
15     while len(x) ≥ len(p):
16         x = [x[i] ^ p[i] for i in range(len(p))] + x[len(p):]
17         while x[0] == 0 and len(x) > 1:
18             x = x[1:]
19     return x

```

Main function:

Set the modulo polynomial to $X^8 + X^4 + X^3 + X^2 + 1$ (this can change for other problem) and the initial polynomial to 1.

Record the occurrence of every 2^N polynomials (plug in $X=2$ as the index value).

While current polynomial hasn't occurred, then set the record value to zero, then shift and mod the polynomial, and back to the loop.

In the end, output the recorded count of polynomials, this is the cycle length with initial polynomial 1.

```

22     def main():
23         poly = [1, 0, 0, 0, 1, 1, 1, 0, 1]
24         x = [0, 0, 0, 0, 0, 0, 0, 1]
25         record = [1 for _ in range(2 ** len(poly) - 1)]
26         count = 0
27         while record[convolution(x)] == 1:
28             count += 1
29             print(count, x)
30             record[convolution(x)] = 0
31             x = shift(x)
32             x = mod(x, poly)
33
34         print(f"The period of the sequence {poly} is {count}")

```

2.

To run the code: python problem2.py

I implemented seven functions in problem2.py, including convolution, shift, mod, shift_and_mod, encrypt, decrypt, and brute_crack.

For the first three, they are same as the part in P1.

For shift_and_mod, it just does shift and then mod.

```

25     def shift_and_mod(x: list, p: list):
26         x = shift(x)
27         x = mod(x, p)
28         return x

```

Encrypt: I transformed each character of the plain text to ascii and does exclusive-or with the key, then store the binary form and shift and mod the key.

```

31     def encrypt(plaintext: str, poly: list, x: list):
32         ciphertext = ""
33         for _ in plaintext:
34             ciphertext += bin(ord(_) ^ convolution(x))[2:].zfill(8)
35             x = shift_and_mod(x, poly)
36         return ciphertext

```

Decrypt: it does almost same as the encrypt function. It takes eight bits of cipher text a time, does exclusive-or with the key, and store the result as a character, then shift and mod the key.

```
39 def decrypt(ciphertext: str, poly: list, x: list):
40     plaintext = ""
41     for i in range(0, len(ciphertext), 8):
42         plaintext += chr(int(ciphertext[i:i + 8], 2) ^ convolution(x))
43         x = shift_and_mod(x, poly)
44     return plaintext
```

Brute_crack: it enumerates through all $2^9 - 1$ possible characteristic polynomials, and calculate if all input satisfies the equation

$$a_n = \sum_{i=1}^8 c_i a_{n+i} \text{ mod } 2$$

If any a_n in the cypher text does not satisfy, then break the test the next possible polynomial. Otherwise, output the polynomial that pass all the tests.

```

47 def brute_crack(ciphertext: str):
48     for i in range(1, 2 ** 9):
49         x = [int(_) for _ in bin(i)[2:].zfill(8)]
50         ok = True
51         for _ in range(len(ciphertext) - 8):
52             a_n = int(ciphertext[_], 2)
53             b_n = [int(_) for _ in ciphertext[_ + 1:_ + 9]]
54
55             res = 0
56             for j in range(8):
57                 res += x[j] * b_n[j]
58             res = res % 2
59             if res != a_n:
60                 ok = False
61                 break
62
63         if ok:
64             print(f"Found x: {x}")
65             break
66         else:
67             continue

```

Main function: take the plain text, characteristic polynomial, and the initial polynomial, then encrypt, decrypt, test if the plain text is same as the decrypted result, and crack the LFSR.

a. Calling the encrypt function and decrypt to test if it is correct. My code says it is correct.

```

70 def main():
71     plaintext = PLAINTEXT
72
73     poly = [1, 0, 0, 0, 1, 1, 1, 0, 1]
74     x = [0, 0, 0, 0, 0, 0, 0, 1]
75
76     ciphertext = encrypt(plaintext, poly, x)
77     print(f"Ciphertext:\n{ciphertext}\n\n")
78     # [print(ciphertext[i:i + 8]) for i in range(0, len(ciphertext),
79
80     decrypted = decrypt(ciphertext, poly, x)
81     print(f"Decrypted:\n{decrypted}\n\n")
82
83     print(f"Plaintext == Decrypted: {plaintext == decrypted}\n\n")
84
85     tmp = ciphertext[:8]
86     brute_crack(tmp)

```

```

PowerShell
E:\Code\CryptoEngineering\Lab4
python .\problem2.py
Ciphertext:
0100000001010110010010100101000101010011011010100010111000101010110001101000001000110111011001100110101010110011100
0000000101101011001101010000011101111011001101110101111000100010010001010001010001100011001010101101000
10011001011011000100001100110011101101100000111011001101011000010010001010111000010010001101110010000001101000101
1100000000010110010010100101101001110110000011001111001100011101110011001100111011100000010000110011111
11100110000010111110100010100010010110110000010101100011011110111001100011000101101101001000000101001001101010001111
010100100101101011000101001101011010010100010001001110100001111010101101010001001000111001000001011110101110011
001010110100100111111000011111000011010010010110010001011011000110100111010110011100010010010110000011110100100
1011110001111101010100101101101100011001101111010001010010110111011110011000010101101100001010110110000101101001001100000
00110001100010011101001001011100110001000011110110010001010010011111010000011011010000011010000001111010000010110
10011010000010001110111000110001010011011011000101111000000010111100110001101101101001111000110110100001001
00001110110001001110101101001000011010110010000011101001001111000110001011101101001111111010001111010
100000101101001010100101101101100001111101110011100000101001000001110110000110001000101101100101011100010
000001011101011000000001010110010110001100000001000010000111100101001000100101001010100011110100100010101100100101
0111001100001110111100010001000110001100011111000011000010111001011101101001010001011010010011011100000011110
110100010011100010010100000100010110010000011001101001000110011011001101110011001100110011011011011011011110110
0111001100101001010000100001001100111100100111001101111011011010001001101110010110110010100110001000000001101101
110100100010011110000010101010100101001000001010110100010011001101000000011010100010100011101010011101010011100010
1110010000100011001110100000100111010110011001000001100111000100100111011100110000001011011010011010010011101000100100
11000111000001001011000011011010100011101100100001110110010001101101010111011000101111
Decrypted:
ATNYCUWEARESTRIVINGTOBEAGREATUNIVERSITYTHATTRANSCENDSDISCIPLINARYDIVIDESTOSOLVETHEINCREASINGLYCOMPLEXPROBLEMSHATTHEWORLDFACES
WEWILLCONTINUETOBEGUIDEDBYTHEIDEATHATWE CANACHIEVE SOMETHINGMUCH GREATER TOGETHER THAN WE CAN INDIVIDUALLY AFTER ALL THAT WAS THE IDEATHATLE
D TO THE CREATION OF FOUR UNIVERSITY IN THE FIRST PLACE

Plaintext == Decrypted: True

Found x: [0, 1, 1, 1, 0, 0, 0, 1]
E:\Code\CryptoEngineering\Lab4

```

Ciphertext:

```

01000000010101100100101001010001010100110111010100010111
11000101010111000110100000110001101110111001100111010101
01011010011100000000010111010110011010100000111011111011
00110111101011111000100011001000010100010100001101001101
01001100011001010010111010001001110010110110001000011100

```

11001111011011000001111011001101011000010010001010010101
1110000100100011101111001000000011010001011100000000101
11001001010010110100111001000111011011000001100111110011
00011110111100110011100110011110111100000010000110011111
11110011000001101111101000101000100101001101000001101011
00011011111011110011000110000101110110100100000001010010
01101010001111011010010010110101101000101001101011110101
00101000100001001111010000111110101011011010100010010001
11100100000101111110111100110010101011011001010011111111
00001111110000110100100101110010000101101100011101001111
01010110011100010010010110000011111010010010111110001111
11010101010010110111011000111001101011111010001010010101
11011111011111110011000010101101100001101101001001100000
00110001100010011101001001011110011000100001000011111011
00100010100101001111110100000110110100000111010000000111
11010000010110100110100000010001111011110001100011010011
011011010001011111110000000010111111100110001110111011010
01111100001110111010000010010000111101100010011110101110
10010000111010110011010010000011110100100111111000110000
10111101101001101001111111111010001111101010000010110100
1010110010101100100101101110111000011111101111001111000
00101100100000111101100001100010001011011001010011100010
00000101111010110000000011010110010111000110000000110000
10000111110010100100001001011010010101000111110100100010
10110100100101011110011000011101111100010001010011100001
00011000111111000011000010111100101111011010011010010101
11010010011011110000001111101101000010011100011001010000
01010011011000100001110011010101011011110011101110111000
10111001101000111011101110111010101111111001110011001010
01011000010000100110010000011101111110010011100110111111
10101000100110111100101101011010011000100010000010011101
11101001000100111100000101010101010010100100000101001011
01000010011001010000000111010100010101000111010100111010
10100111100010111100100001000110011101000001100111010110
01100100000011001111000100100111101110011000000011011011
01011010010011110100001001001100011110000010010110000110
11010100011101010001110111001000011101010000011011010101
0111011000101111

Decrypted:

ATNYCUWEARESTRIVINGTOBEAGREATUNIVERSITYTHATTRANSCENDSDISCIPLINARYDIVIDESTOSOLVETHEINCREASINGLYCOMPLEXPROBLEMSHATTHEWORLDFACESWEWILLCONTINUEBEGUIDEDBYTHEIDEATHATWE CANACHIEVESOMETHINGMUCHGREATER TOGETHER THAN WE CAN INDIVIDUALLY AFTER ALL THAT WAS THE IDEA THAT LED TO THE CREATION OF FOUR UNIVERSITIES IN THE FIRST PLACE

b. Yes, it is possible. Since the MSB of each 8-bit strings in the cipher text are actually a_n , so we can solve the recurrence equation

$$a_n = \sum_{i=1}^8 c_i a_{n+i} \bmod 2$$

and find out the original characteristic polynomial.

c. As described above, the brute crack function can find the original key polynomial through brute force. In the image above, it finds out that c_1 to c_8 are $[0, 1, 1, 1, 0, 0, 0, 1]$, implies that

$$a_n = a_{n+2} + a_{n+3} + a_{n+4} + a_{n+8}$$

Which equals to

$$x^8 = x^4 + x^3 + x^2 + 1$$

So, the original characteristic polynomial is

$$x^8 + x^4 + x^3 + x^2 + 1$$

3.

To run the code: `python problem3.py`

Required libraries: `numpy`, `matplotlib`, and `itertools`

To install: `pip install numpy, matplotlib, itertools`

a. I take `np.random.default_rng()` as the RNG to use. Naïve shuffle: generate a random index $x_i \in [0, N)$ then swap the i -th and x_i -th element for N iterations.

```
8     def naive_shuffle(A):
9         n = len(A)
10        for i in range(n):
11            j = rng.integers(n)
12            A[i], A[j] = A[j], A[i]
13        return A
```


Fisher-Yates shuffle: generate a random integer $x_i \in [0, i]$ then swap the i -th and x_i -th element for $i = N - 1$ down to 1.

```

16     def fisher_yates_shuffle(A):
17         n = len(A)
18         for i in range(n - 1, 0, -1):
19             j = rng.integers(i + 1)
20             A[i], A[j] = A[j], A[i]
21         return A

```

Main function: initialize two dictionaries for all permutations of [1,2,3,4] with initialize value 0 to record the occurrence of every possible outcome, then shuffle and record for a million times, plot, and output.

```

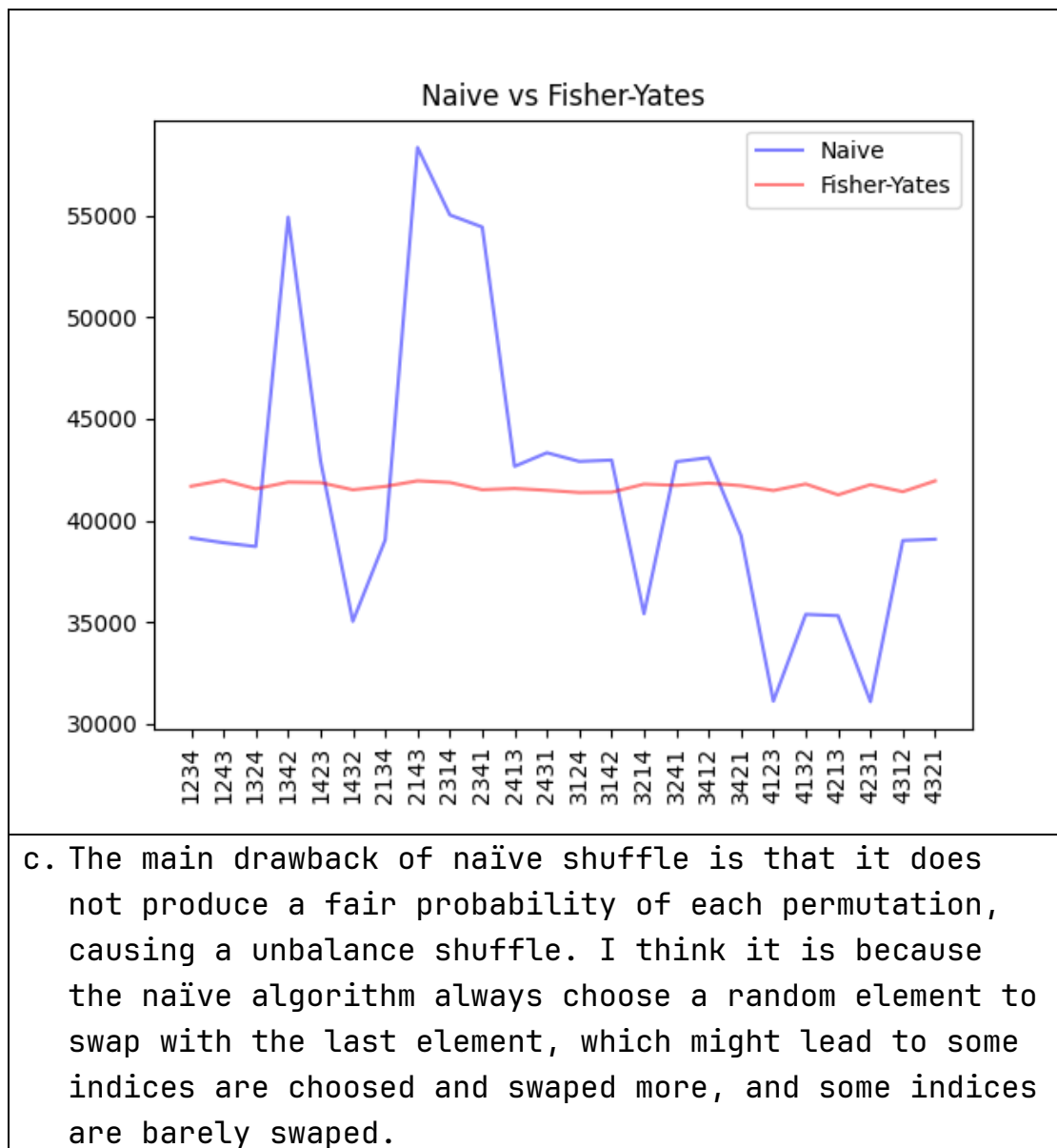
24 def main():
25     N = 1000000
26     naive = dict()
27     fisher = dict()
28
29     [naive.setdefault(_, 0) for _ in itertools.permutations(range(1, 5))]
30     [fisher.setdefault(_, 0) for _ in itertools.permutations(range(1, 5))]
31
32     for _ in range(N):
33         naive[tuple(naive_shuffle([1, 2, 3, 4]))] += 1
34         fisher[tuple(fisher_yates_shuffle([1, 2, 3, 4]))] += 1
35
36     plt.plot(*args: [''.join(map(str, _)) for _ in naive.keys()], list(naive.values()), alpha=0.5, c='b', label="Naive")
37     plt.plot(*args: [''.join(map(str, _)) for _ in fisher.keys()], list(fisher.values()), alpha=0.5, c='r', label="Fisher-Yates")
38     plt.xticks(rotation=90)
39     plt.title("Naive vs Fisher-Yates")
40     plt.legend()
41     plt.savefig("lab4_problem3.png")
42     plt.show()
43
44     print("Naive:")
45     [print(f"{_}: {naive[_]}") for _ in naive.keys()]
46
47     print("Fisher-Yates:")
48     [print(f"{_}: {fisher[_]}") for _ in fisher.keys()]

```

b. From the output result and plot below, we can see that fisher-yates is way better than naïve shuffle as the occurrence of each possible permutation are really close.

Naive:	Fisher-Yates:
(1, 2, 3, 4): 39138	(1, 2, 3, 4): 41684
(1, 2, 4, 3): 38899	(1, 2, 4, 3): 41978

(1, 3, 2, 4): 38714	(1, 3, 2, 4): 41556
(1, 3, 4, 2): 54937	(1, 3, 4, 2): 41884
(1, 4, 2, 3): 42937	(1, 4, 2, 3): 41862
(1, 4, 3, 2): 35030	(1, 4, 3, 2): 41509
(2, 1, 3, 4): 39040	(2, 1, 3, 4): 41675
(2, 1, 4, 3): 58370	(2, 1, 4, 3): 41948
(2, 3, 1, 4): 55046	(2, 3, 1, 4): 41866
(2, 3, 4, 1): 54450	(2, 3, 4, 1): 41511
(2, 4, 1, 3): 42662	(2, 4, 1, 3): 41573
(2, 4, 3, 1): 43332	(2, 4, 3, 1): 41482
(3, 1, 2, 4): 42906	(3, 1, 2, 4): 41374
(3, 1, 4, 2): 42971	(3, 1, 4, 2): 41388
(3, 2, 1, 4): 35403	(3, 2, 1, 4): 41788
(3, 2, 4, 1): 42892	(3, 2, 4, 1): 41726
(3, 4, 1, 2): 43088	(3, 4, 1, 2): 41836
(3, 4, 2, 1): 39252	(3, 4, 2, 1): 41712
(4, 1, 2, 3): 31098	(4, 1, 2, 3): 41472
(4, 1, 3, 2): 35373	(4, 1, 3, 2): 41794
(4, 2, 1, 3): 35307	(4, 2, 1, 3): 41260
(4, 2, 3, 1): 31072	(4, 2, 3, 1): 41761
(4, 3, 1, 2): 39009	(4, 3, 1, 2): 41414
(4, 3, 2, 1): 39074	(4, 3, 2, 1): 41947



c. The main drawback of naïve shuffle is that it does not produce a fair probability of each permutation, causing a unbalance shuffle. I think it is because the naïve algorithm always choose a random element to swap with the last element, which might lead to some indices are choosed and swaped more, and some indices are barely swaped.