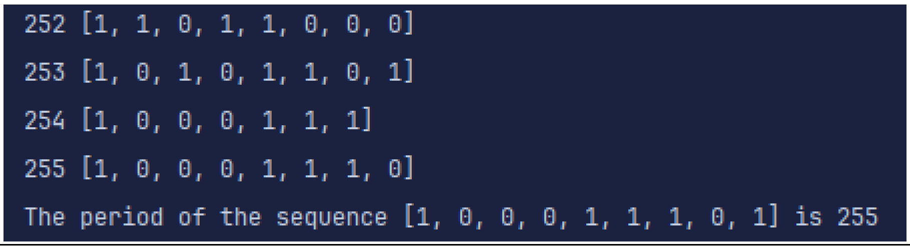
Crypto Engineering Quiz 4

1.

a. Yes.

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

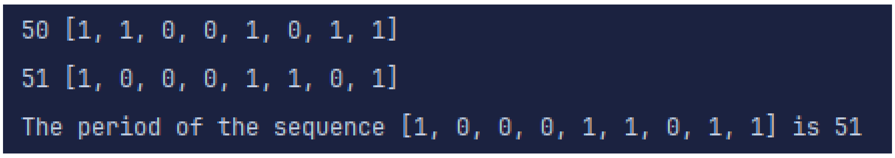


```
252 [1, 1, 0, 1, 1, 0, 0, 0]
253 [1, 0, 1, 0, 1, 1, 0, 1]
254 [1, 0, 0, 0, 1, 1, 1]
255 [1, 0, 0, 0, 1, 1, 1, 0]
The period of the sequence [1, 0, 0, 0, 1, 1, 1, 0, 1] is 255
```

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.



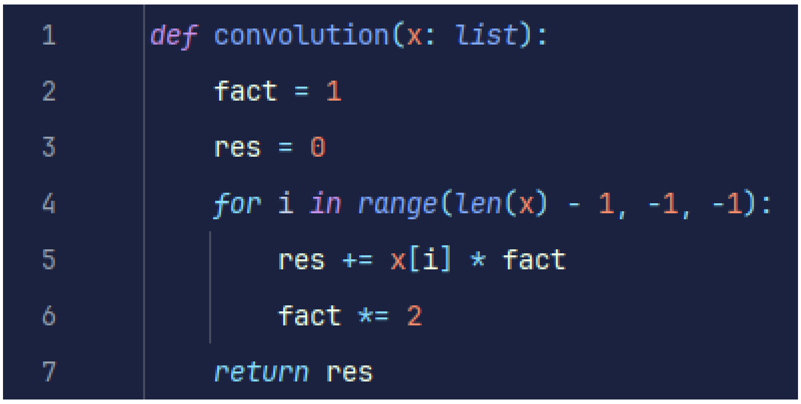
```
50 [1, 1, 0, 0, 1, 0, 1, 1]
51 [1, 0, 0, 0, 1, 1, 0, 1]
The period of the sequence [1, 0, 0, 0, 1, 1, 0, 1, 1] is 51
```

Appendix: Description of problem1.py

To run the code: python problem1.py

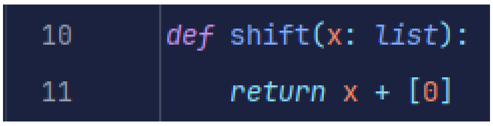
I implemented three functions in the file, including convolution, shift, mod.

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



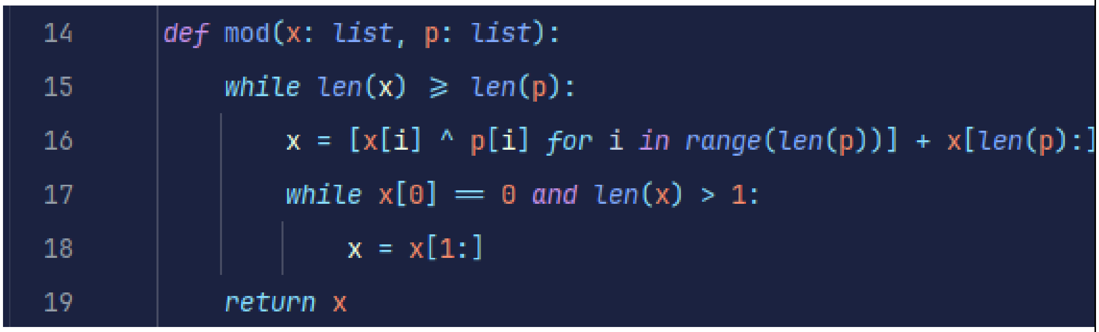
```python
def convolution(x: list):
    fact = 1
    res = 0
    for i in range(len(x) - 1, -1, -1):
        res += x[i] * fact
        fact *= 2
    return res
```

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



```python
def shift(x: list):
    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.



```python
def mod(x: list, p: list):
    while len(x) >= len(p):
        x = [x[i] ^ p[i] for i in range(len(p))] + x[len(p):]
        while x[0] == 0 and len(x) > 1:
            x = x[1:]
    return x
```

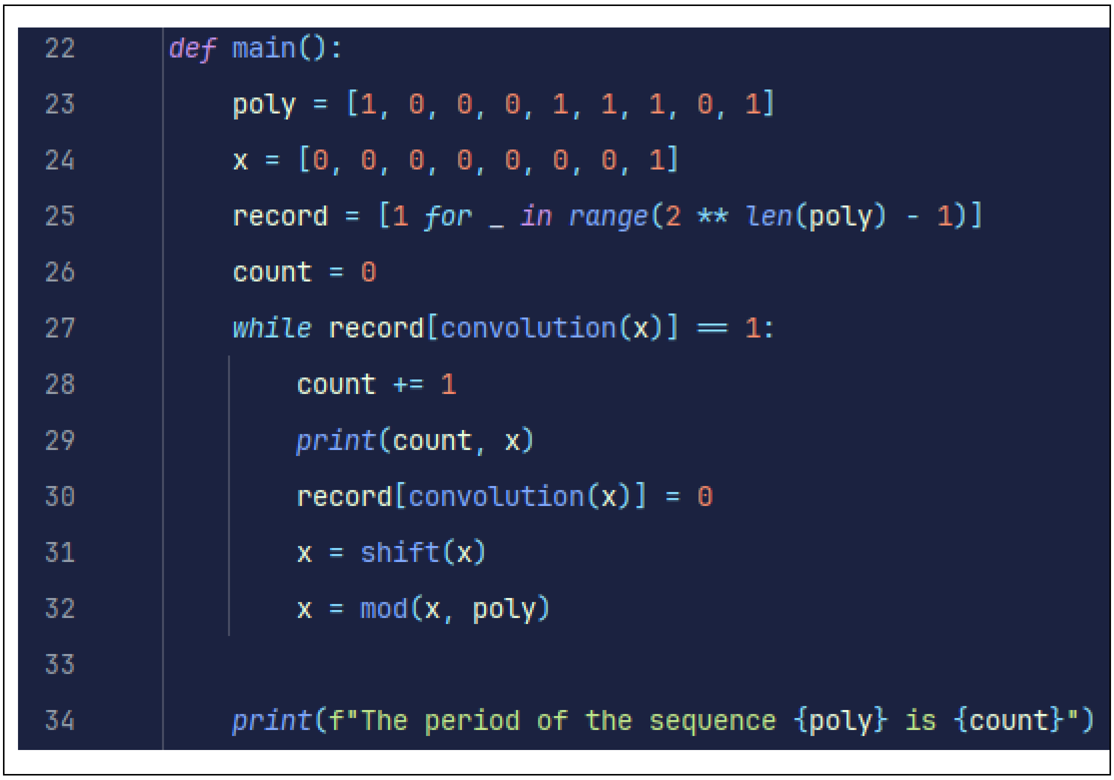
Main function:

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.



```python
def main():
    poly = [1, 0, 0, 0, 1, 1, 1, 0, 1]
    x = [0, 0, 0, 0, 0, 0, 0, 1]
    record = [1 for _ in range(2 ** len(poly) - 1)]
    count = 0
    while record[convolution(x)] == 1:
        count += 1
        print(count, x)
        record[convolution(x)] = 0
        x = shift(x)
        x = mod(x, poly)

    print(f"The period of the sequence {poly} is {count}")
```

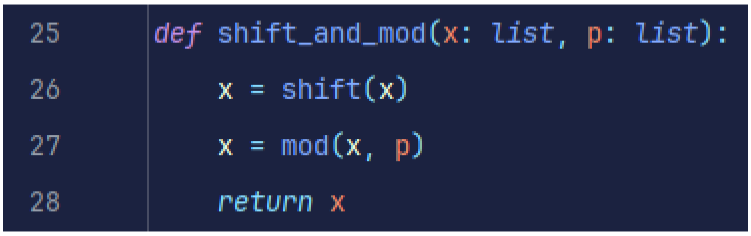
2.

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.

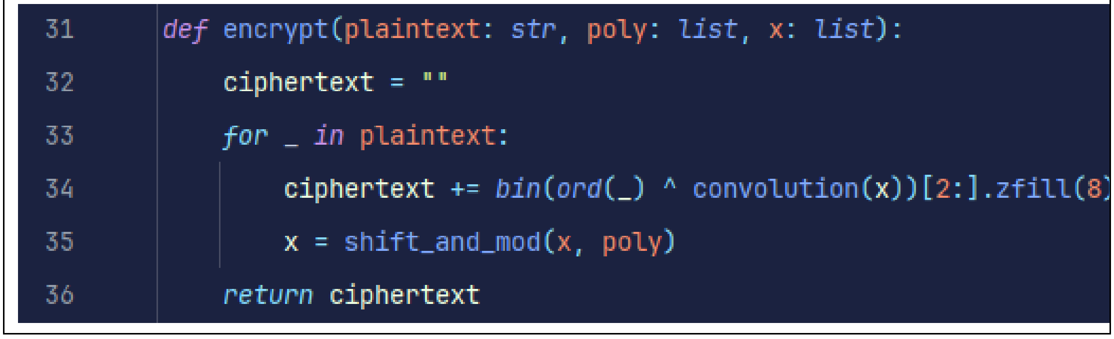
For the first three, they are same as the part in P1.

For shift_and_mod, it just does shift and then mod.



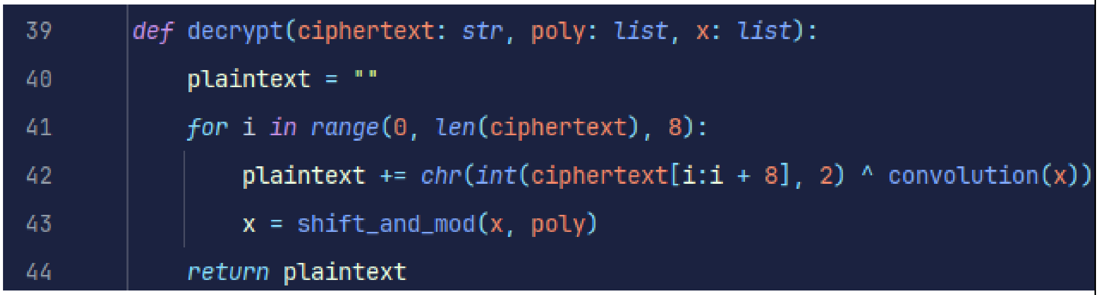
```python
def shift_and_mod(x: list, p: list):
    x = shift(x)
    x = mod(x, p)
    return x
```

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.



```python
def encrypt(plaintext: str, poly: list, x: list):
    ciphertext = ""
    for _ in plaintext:
        ciphertext += bin(ord(_) ^ convolution(x))[2:].zfill(8)
        x = shift_and_mod(x, poly)
    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.

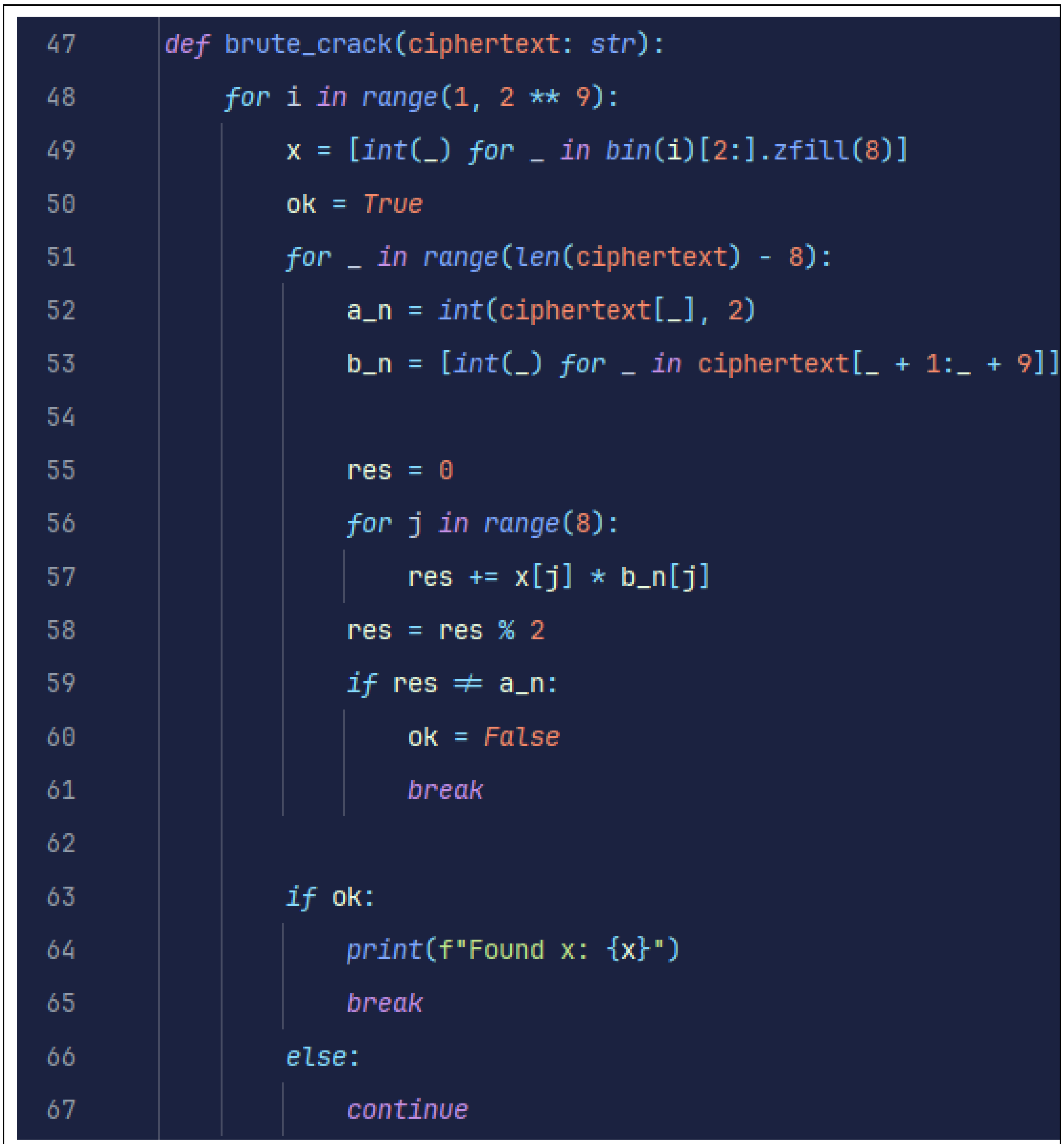


```python
def decrypt(ciphertext: str, poly: list, x: list):
    plaintext = ""
    for i in range(0, len(ciphertext), 8):
        plaintext += chr(int(ciphertext[i:i + 8], 2) ^ convolution(x))
        x = shift_and_mod(x, poly)
    return plaintext
```

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

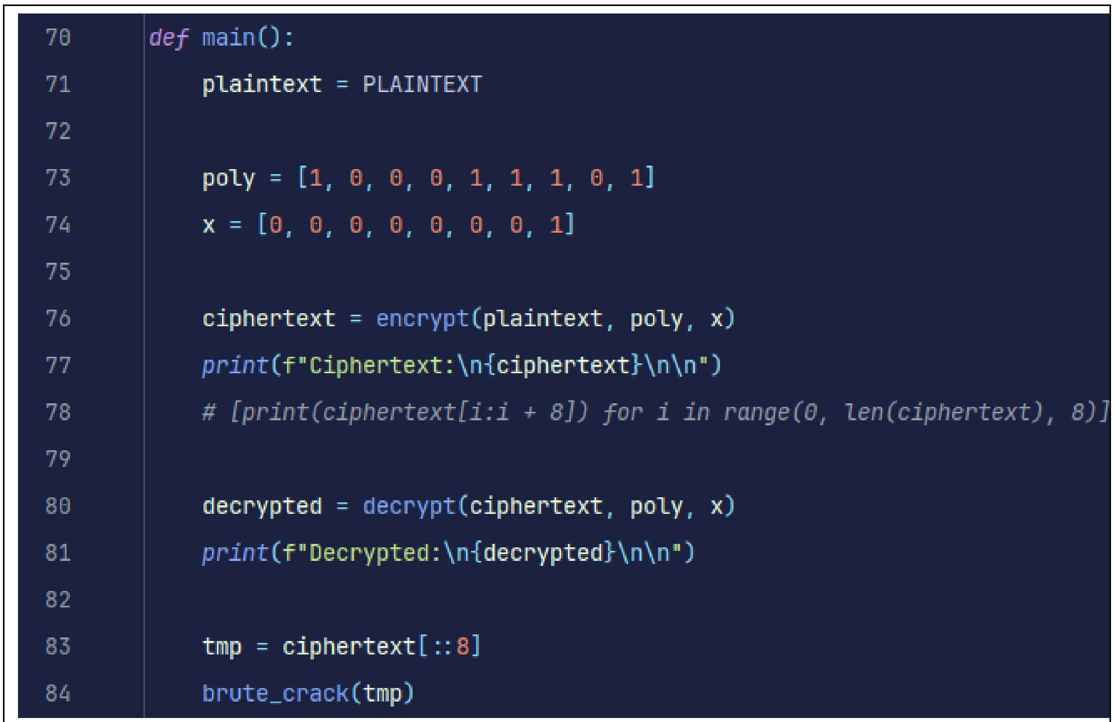
$$a_n = \sum_{i=1}^{8} c_i a_{n+i} \ mod \ 2$$

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.



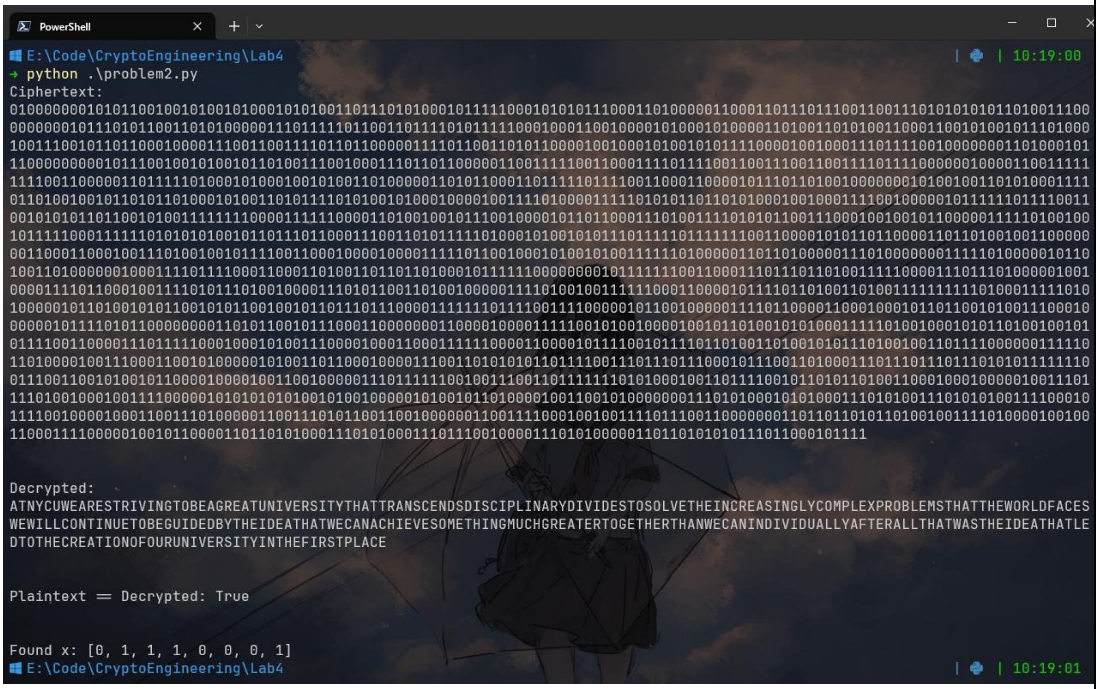
```python
def brute_crack(ciphertext: str):
    for i in range(1, 2 ** 9):
        x = [int(_) for _ in bin(i)[2:].zfill(8)]
        ok = True
        for _ in range(len(ciphertext) - 8):
            a_n = int(ciphertext[_], 2)
            b_n = [int(_) for _ in ciphertext[_ + 1:_ + 9]]

            res = 0
            for j in range(8):
                res += x[j] * b_n[j]
            res = res % 2
            if res != a_n:
                ok = False
                break

        if ok:
            print(f"Found x: {x}")
            break
        else:
            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.



```python
def main():
    plaintext = PLAINTEXT

    poly = [1, 0, 0, 0, 1, 1, 1, 0, 1]
    x = [0, 0, 0, 0, 0, 0, 0, 1]

    ciphertext = encrypt(plaintext, poly, x)
    print(f"Ciphertext:\n{ciphertext}\n\n")
    # [print(ciphertext[i:i + 8]) for i in range(0, len(ciphertext), 8)]

    decrypted = decrypt(ciphertext, poly, x)
    print(f"Decrypted:\n{decrypted}\n\n")

    tmp = ciphertext[::8]
    brute_crack(tmp)
```

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



```
E:\Code\CryptoEngineering\Lab4                                                         | 10:19:00
→ python .\problem2.py
Ciphertext:
010000000101011001001010010100010101001101110101000101111100010101011100011010000011000110111011100110011101010101011010011100
000000010111010110011010100000111011111011001101111010111110001000110010000101000101000011010011010100110001100101001011101000
1001110010110110001000011100110011110110110000011110110011010110000100100010100101011110000100100011101111001000000001101000101
1100000000010111001001010010110100111001000111011011000001100111111001100011110111100110011100110011110111100000010000110011111
1111001100000110111111010001010001001010011010000011010110001110111110111110011000110000101110110100100000000101001001101010001111
0110100100101101011010001010011010111101010010100010000100111101000011111010101101101010001001000111100100000010111111011110011
0010101011011001010011111111000011111100001101001001011100100000101101100011101001111010101100111000100100101100000111110100100
101111110001111110101010100101101110110001110011010111110100010100101011101111110111111110011000010101101100001101101001001100000
00110001100010011101001001011110011000100001000011111011001000101001010011111110100000011011010000011101000000011111010000010110
100110100000010001111011110001100011010011011010100010111111000000000101111111100110001110111011010011110000111011101000001001
00001111011000100111101011101001000011101011001101001000001111010010011111100011000010111101101001101001111111111010001111010
10000001011010010101100101011001001011011101110000111111101111100111110000001011001000000111101100001100010001011011001010011100010
000001011110101100000000011010110010111000110000000011000010000011111001010010000100101101001010100011111010010001010110100100101
0111100110000111011111100010001010011100001000110001111110000110000101111001011110110100110100101011101001001101111000000111110
110100001001110001100101000001010011011000100001110011010101011011110011101110111000101110011010001110111011101110101011111110
01110011001010010110000100001001100100000111011111110010011100110111111101010001001101111001011010110100110001000100000010011101
1110100100010011110000001010101010010100100000101001011010000100110010100000001110101000101010001110101001110101010011100010
1111001000010001100111010000001100111010110011001000000110011110001001001111011100110000000110110110101101001001111010000100100
11000111100000010010110000110110101000111010100011101110010000111010100000110110101010111011000101111


Decrypted:
ATNYCUWEARESTRIVINGTOBEAGREATUNIVERSITYTHATTRANSCENDSDISCIPLINARYDIVIDESTOSOLVETHEINCREASINGLYCOMPLEXPROBLEMSTHATTHEWORLDFACES
WEWILLCONTINUETOBEGUIDEDBYTHEIDEATHATWECANACHIEVESOMETHINGMUCHGREATERTOGETHERTHANWECANINDIVIDUALLYAFTERALLTHATWASTHEIDEATHATLE
DTOTHECREATIONOFOURUNIVERSITYINTHEFIRSTPLACE


Plaintext == Decrypted: True


Found x: [0, 1, 1, 1, 0, 0, 0, 1]
E:\Code\CryptoEngineering\Lab4                                                         | 10:19:01
```

Ciphertext:

010000000101011001001010010100010101001101110101000101111
1100010101010111000110100000110001101110111001100111010101
0101101001110000000000101110101100110101000000111011111011
0011011110101111100010001100100000101000101000001101001101
0100110001100101001011101000100111001011011000100000011100

```
110011110110110000001111011001101011000010010001010010101
```

```
Decrypted:
ATNYCUWEARESTRIVINGTOBEAGREATUNIVERSITYTHATTRANSCENDSDIS
CIPLINARYDIVIDESTOSOLVETHEINCREASINGLYCOMPLEXPROBLEMSTHA
TTHEWORLDFACESWEWILLCONTINUETOBEGUIDEDBYTHEIDEATHATWECAN
ACHIEVESOMETHINGMUCHGREATERTOGETHERTHANWECANINDIVIDUALLY
AFTERALLTHATWASTHEIDEATHATLEDTOTHECREATIONOFOURUNIVERSIT
YINTHEFIRSTPLACE
```

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

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

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

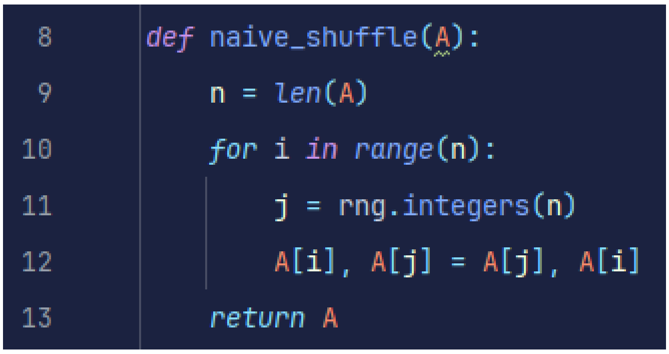
$$x^8 + x^4 + x^3 + x^2 + 1$$

3.

To run the code: python problem3.py

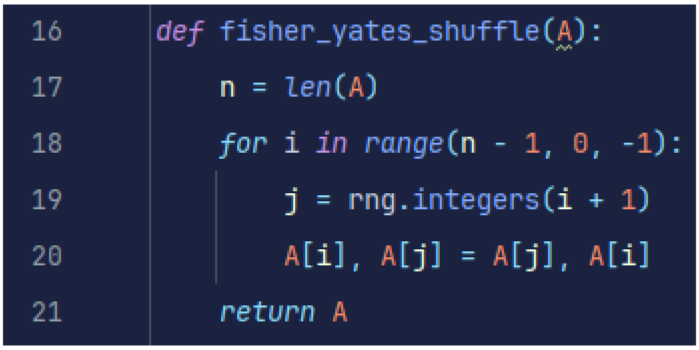
Required libraries: numpy, matplotlib, and itertools
To install: pip install numpy, matplotlib, itertools

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.



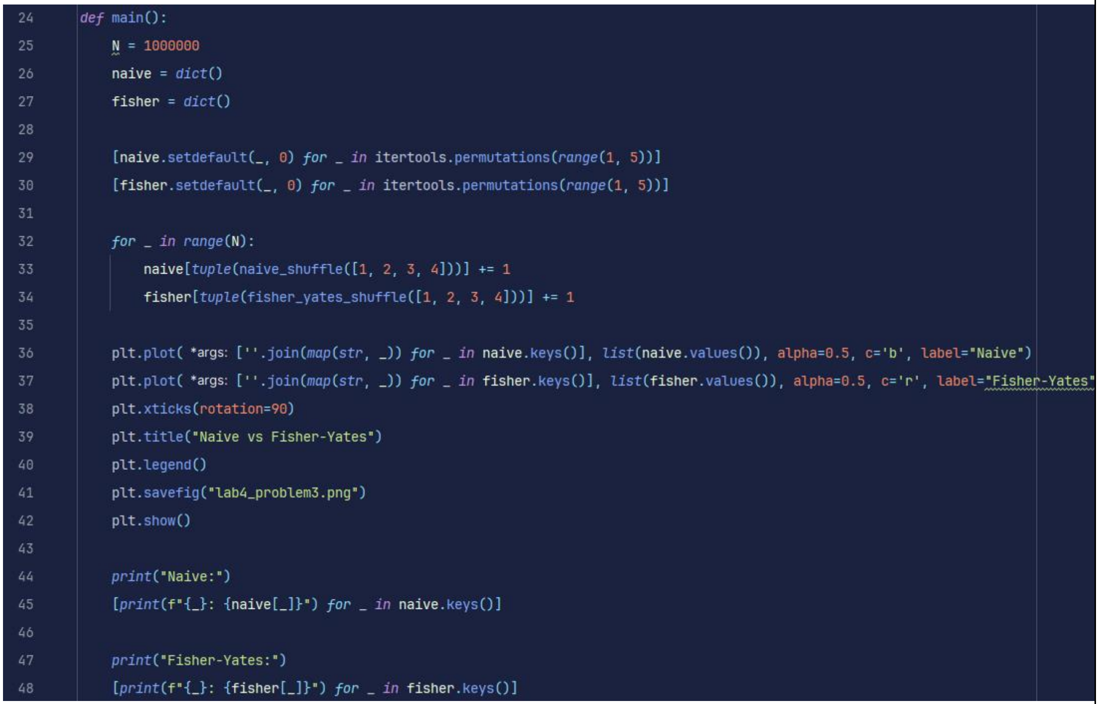
```python
def naive_shuffle(A):
    n = len(A)
    for i in range(n):
        j = rng.integers(n)
        A[i], A[j] = A[j], A[i]
    return A
```

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.



```python
def fisher_yates_shuffle(A):
    n = len(A)
    for i in range(n - 1, 0, -1):
        j = rng.integers(i + 1)
        A[i], A[j] = A[j], A[i]
    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.

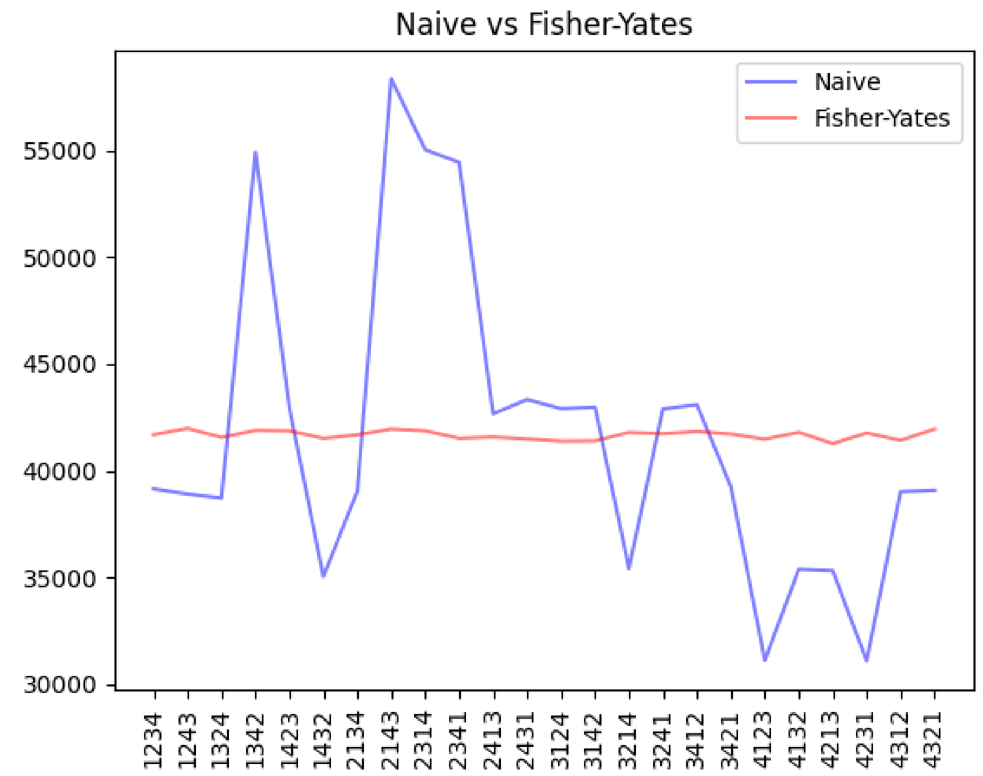


```python
def main():
    N = 1000000
    naive = dict()
    fisher = dict()

    [naive.setdefault(_, 0) for _ in itertools.permutations(range(1, 5))]
    [fisher.setdefault(_, 0) for _ in itertools.permutations(range(1, 5))]

    for _ in range(N):
        naive[tuple(naive_shuffle([1, 2, 3, 4]))] += 1
        fisher[tuple(fisher_yates_shuffle([1, 2, 3, 4]))] += 1

    plt.plot( *args: [''.join(map(str, _)) for _ in naive.keys()], list(naive.values()), alpha=0.5, c='b', label="Naive")
    plt.plot( *args: [''.join(map(str, _)) for _ in fisher.keys()], list(fisher.values()), alpha=0.5, c='r', label="Fisher-Yates"
    plt.xticks(rotation=90)
    plt.title("Naive vs Fisher-Yates")
    plt.legend()
    plt.savefig("lab4_problem3.png")
    plt.show()

    print("Naive:")
    [print(f"{_}: {naive[_]}") for _ in naive.keys()]

    print("Fisher-Yates:")
    [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.