Problem 1

LFSR is simply an arrangement of n stages in a row with the last stage, plus any other stages, modulo-two added together and returned to the first stage. An algebraic expression can symbolize this arrangement of stages and tap points called the characteristic polynomial. One kind of characteristic polynomial called primitive polynomials over GF(2), the field with two elements 0, 1, can be used for pseudorandom bit generation to let linear-feedback shift register (LFSR) with maximum cycle length.

a) Is $x^8 + x^4 + x^3 + x^2 + 1$ a primitive polynomial?

b) What is the maximum cycle length generated by $x^8 + x^4 + x^3 + x^2 + 1$?

c) Are all irreducible polynomials primitive polynomials?

Problem 2

(a) Yes，不能化簡

(b) the maximum cycle length of $2^8-1=255$

This is because the cycle length of an LFSR

with n-bit 2^n-1 A: 255

(c) all primitive polynomial are irreducible pol..

but all irreducible polynomials are not

considered as primitive polynomials

because an irreducible polynomial may not have the

property of generating a maximal-length sequence in an LFSR

$p = x^8 + x^4 + x^3 + x + 1$ is irreducible, but not primitive (there are 255 element modulo; but the subgroup generated by x only contain 51 of them)

a) Please use $x^8 + x^4 + x^3 + x^2 + 1$ as a characteristic polynomial to write a Python program to encrypt the following plaintext message with the initial key 00000001, then decrypt it to see if your encryption is correct.

b) Due to the property of ASCII coding the ASCII A to Z, the MSB of each byte will be zero (left most bit); therefore, every 8 bits will reveal 1 bit of random number (i.e. keystream); if it is possible to find out the characteristic polynomial of a system by solving of linear equations?

c) **Extra credit:** Write a linear equations program solving program to find the characteristic polynomial for this encryption with initial 00000001.

(a) pip install pylfsr

run the code

```python
from pylfsr import LFSR

def encrypt_decrypt_pylfsr(plaintext, init_state, poly_taps):
    # 初始化LFSR
    lfsr = LFSR(initstate=init_state, fpoly=poly_taps)

    # 生成key長度耀明文的8倍(明文變成8倍的二進制)
    keystream = []
    for _ in range(len(plaintext) * 8):
        keystream.append(lfsr.next())  # 將每個文字用到key中

  # 將字串轉為二禁制
    plaintext_binary = ''
    for c in plaintext:
        plaintext_binary += format(ord(c), '08b')

    # 把key跟轉成二進制的明文XOR
    encrypted_binary = ''
    for pb, ks in zip(plaintext_binary, keystream):
        encrypted_binary += str(int(pb) ^ int(ks))

    # 把加密的明文回傳
    encrypted_text = ''
    for i in range(0, len(encrypted_binary), 8):
        encrypted_text += chr(int(encrypted_binary[i:i+8], 2))
    return encrypted_text

def main():
    plaintext = ("ATNYCUWEARESTRIVINGTOBEAGREATUNIVERSITYTHATTRAN"
                 "SCENDSDISCIPLINARYDIVIDESTOSOLVETHEINCREASINGLYCO"
                 "MPLEXPROBLEMSTHATTHEWORLDFACESWEWILLCONTINUET"
                 "OBEGUIDEDBYTHEIDEATHATWECANACHIEVESOMETHINGMU"
                 "CHGREATERTOGETHERTHANWECANINDIVIDUALLYAFTERALLT"
                 "HATWASTHEIDEATHATLEDTOTHECREATIONOFOURUNIVERSI"
                 "TYINTHEFIRSTPLACE")
    init_state = [0,0,0,0,0,0,0,1]  # 00000001
    poly_taps = [8, 4, 3, 2]  # x^8 + x^4 + x^3 + x^2 + 1

    # 加密
    ciphertext = encrypt_decrypt_pylfsr(plaintext, init_state, poly_taps)
    print(f"Ciphertext: {ciphertext}")

    # 解密
    decrypted_text = encrypt_decrypt_pylfsr(ciphertext, init_state, poly_taps)
    print(f"Decrypted Text: {decrypted_text}")

    # 檢查是否加密正確
    if decrypted_text == plaintext:
        print("Encryption and decryption are correct!")

    else:
        print("There was an error in the encryption/decryption process.")

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

```
PS C:\Users\user\Desktop\課程\密碼工\QUIZ4> python .\111550149.py
Ciphertext: Áå¦&Ôò5#áj:·sêw#5XLrçA ÇVê8W&¬h´'
èÚ6}ù÷I(#YoI  °[è¨
ÛÞì6÷ÃmÂÄ¢ÄÊ)9u]õWÖHß¹ÝnùÌWÕ;¾$~lMSMí¹ª%éîf'ÝNtÍ\WËµI2Rb|¨£ºtËU0ø²
tD^|[¥Q¹ïB&³ÿç(q)v£¯áûõÞÚ'&Ëax?M°f²Lä¥ÏñÇ?;
ß®c%ãAÔ´ùÏê.ÒÒÖ:
ÅwÔù <Ââ$+üa(ªwöwª7R
DaìM¢ÇKá7H1«që£
Decrypted Text: ATNYCUWEARESTRIVINGTOBEAGREATUNIVERSITYTHATTRANSCENDSDISCIPLINARYDIVIDESTOSOLVETHEINCREASINGLYCOMPLEXPROBLEMSTHATTHEWORLDFACESWEWILLCONTINUETOBE
GUIDEDBYTHEIDEATHATWECANACHIEVESOMETHINGMUCHGREATERTOGETHERTHANWECANINDIVIDUALLYAFTERALLTHATWASTHEIDEATHATLEDTOTHECREATIONOFOURUNIVERSITYINTHEFIRSTPLACE
Encryption and decryption are correct!
PS C:\Users\user\Desktop\課程\密碼工\QUIZ4>
```

(b) ① ASCII 性質：A-Z 之2進制 MSB 為 0

在 keystream 中每8位有1位是已知的

② 知道明文&密文可以嘗試破解出一組特徵多項式

③ LSFR 由特徵多項式定義，理論上推算出 生成 從 keystream 的 LFSR 的特徵多項式

前提是要有足夠的 keystream 的樣本

a) Please write a Python program to simulate two algorithms with a set of 4 cards, shuffling each **a million times**. Collect the count of all combinations and output, for example:

```
$ python problem3.py
Naive algorithm:
[1 2 3 4]: 41633
[1 2 4 3]: 41234
... and so on
Fisher-Yates shuffle:
[1 2 3 4]: 41234
[1 2 4 3]: 41555
... and so on
```

Hint: you can use ***random*** *library.*

b) Based on your analysis, which one is better, why?

(a)

c) What are the drawbacks of the other one, and what causes these drawbacks?

```python
import random

def naive_shuffle(cards):
    counts = {}
    for _ in range(1000000):  # 洗一百萬次牌
        shuffled = cards.copy()
        # 執行naive洗牌
        for i in range(len(shuffled)):
            n = random.randint(0, len(shuffled) - 1)  # 隨機洗牌
            shuffled[i], shuffled[n] = shuffled[n], shuffled[i]  # 交換
        # 計算count
        counts[tuple(shuffled)] = counts.get(tuple(shuffled), 0) + 1
    return counts

def fisher_yates_shuffle(cards):
    counts = {}
    for _ in range(1000000):  # 洗衣百萬次牌
        shuffled = cards.copy()  #
        # 執行Fisher-Yates
        for i in range(len(shuffled) - 1, 0, -1):
            n = random.randint(0, i)  # 隨機拿牌
            shuffled[i], shuffled[n] = shuffled[n], shuffled[i]  # 交換
        # 計算
        counts[tuple(shuffled)] = counts.get(tuple(shuffled), 0) + 1
    return counts

# 四張牌要洗
cards = [1, 2, 3, 4]


naive_counts = naive_shuffle(cards)
fisher_yates_counts = fisher_yates_shuffle(cards)

# 印出結果
print("Naive algorithm:")
for combination, count in naive_counts.items():
    print(f"{list(combination)}: {count}")

print("\nFisher-Yates shuffle:")
for combination, count in fisher_yates_counts.items():
    print(f"{list(combination)}: {count}")
```

```
Naive algorithm:
[2, 4, 1, 3]: 43128
[3, 2, 1, 4]: 35148
[2, 1, 4, 3]: 58563
[3, 1, 4, 2]: 42840
[1, 4, 3, 2]: 35077
[1, 2, 3, 4]: 38885
[4, 3, 1, 2]: 39401
[1, 3, 2, 4]: 38918
[4, 3, 2, 1]: 39116
[3, 2, 4, 1]: 42719
[2, 3, 4, 1]: 54419
[1, 3, 4, 2]: 54599
[4, 1, 3, 2]: 35100
[2, 4, 3, 1]: 43022
[2, 1, 3, 4]: 39115
[3, 4, 1, 2]: 43025
[4, 1, 2, 3]: 31382
[2, 3, 1, 4]: 54842
[1, 2, 4, 3]: 39301
[3, 4, 2, 1]: 38967
[1, 4, 2, 3]: 43323
[4, 2, 3, 1]: 31463
[3, 1, 2, 4]: 42859
[4, 2, 1, 3]: 34788
```

```
Fisher-Yates shuffle:
[2, 4, 3, 1]: 41469
[4, 1, 3, 2]: 41628
[2, 4, 1, 3]: 41651
[4, 2, 1, 3]: 41673
[1, 3, 2, 4]: 41735
[1, 3, 4, 2]: 41991
[1, 2, 3, 4]: 41525
[3, 4, 2, 1]: 41628
[3, 1, 2, 4]: 41829
[2, 3, 4, 1]: 41767
[3, 2, 1, 4]: 41426
[3, 1, 4, 2]: 41541
[1, 4, 2, 3]: 42003
[3, 2, 4, 1]: 41569
[4, 3, 1, 2]: 41916
[2, 1, 3, 4]: 41884
[4, 2, 3, 1]: 41683
[2, 1, 4, 3]: 41481
[2, 3, 1, 4]: 41603
[4, 3, 2, 1]: 41579
[3, 4, 1, 2]: 41614
[1, 2, 4, 3]: 41611
[4, 1, 2, 3]: 41578
[1, 4, 3, 2]: 41616
```

(b) Fisher_Yate 比较好

naive 的随机性不如 fisher_Yate
因为每次交换者随机选，故
有些牌序大於其他，结果就不够
随机公正

(c) Fisher_Yate drawback
① 对随机器的依赖高，如果随机性
不够高可能一样不公正
② 要很大的内存空间去实现洗牌