Problem 1

110550143 洪巧芸 1/2

(a) Yes (查表, Lecture 3 p.118)

自符合定義, 可以得到 $1\sim2^8-1$ 的所有數值

$x^8+x^4+x^3+x^2+1 \rightarrow 100011101$

$\alpha^7 = 0100\ 00000$ $\quad\alpha^{21} = 0111\ 10101$

$\alpha^8 = 0000\ 11101$ $\quad\alpha^{22} = 0111\ 10111$

$\alpha^9 = 0001\ 11010$ $\quad\alpha^{23} = 0111\ 10011$

$\alpha^{10} = 0011\ 10100$ $\quad\alpha^{24} = 0111\ 11010$

$\alpha^{11} = 0111\ 01000$ $\quad\alpha^{25} = 0111\ 01001$

$\alpha^{12} = 0110\ 01101$ $\quad\vdots$

$\alpha^{13} = 0100\ 00111$

$\alpha^{14} = 0000\ 10010$

$\alpha^{18} = 0001\ 11101$

$$\begin{array}{r} 1111\ 10100 \\ 1000\ 11101 \\ \hline 0111\ 01001 \end{array}$$

(b) 256# (∵是 primitive polynomial ∴找最高次項取 2 的次方 $2^8=256$)#

(c) × e.g. x^2+1 是 irreducible 但不是 primitive#

$$\left.\begin{array}{l} \alpha^0 = 001 \\ \alpha^1 = 010 \\ \alpha^2 = 001 \end{array}\right\} \text{沒有出現所有可能的係數組合}$$

Problem 2

(a) 引用 numpy (import numpy as np)

分成 lfsr, a_encrypt, a_decrypt 三個 function 分別給出它們的 pseudocode

①
```
function lfsr(seed, characteristic_polynomial, length):
    state := seed                      # 初始化LFSR狀態為種子值
    keystream := []                    # 初始化密鑰流列表為空
    for i from 1 to length:            # 重複直到生成指定長度的密鑰流
        keystream[i] := state & 1      # 將當前狀態的最低位添加到密鑰流
        feedback := 0                  # 初始化反饋值為0
        # 對特徵多項式的每一項進行迭代
        for j from 0 to length of characteristic_polynomial:
            feedback := feedback XOR ((state >> j) AND characteristic_polynomial[j])
        # 根據反饋值更新狀態
        state := ((state >> 1) OR (feedback << (length of characteristic_polynomial - 1)))
    return keystream                   # 返回生成的密鑰流
```

②
```
function a_encrypt(plaintext, key):
    # 使用 LFSR 生成密鑰流
    keystream := lfsr(key, [1, 0, 0, 0, 1, 1, 1, 0, 1], plaintext的長度)
    ciphertext := []  # 初始化密文列表為空
    # 對每個明文字符和密鑰流進行異或運算，並轉換為字符形式
    for i 從 0 到 plaintext的長度 - 1:
        ciphertext[i] := chr(ord(plaintext[i]) XOR keystream[i])
    返回將ciphertext的所有元素連接成一個字符串  # 將所有密文字符連接成一個字符串並返回
```

執行 python3 problem2.py

③
```
function a_decrypt(ciphertext, key):
    keystream := lfsr(key, [1, 0, 0, 0, 1, 1, 1, 0, 1], ciphertext的長度)
    plaintext := []
    for i 從 0 到 ciphertext的長度 - 1:
    # 對每個密文字符和密鑰流進行異或運算，並轉換為字符形式
        plaintext[i] := chr(ord(ciphertext[i]) XOR keystream[i])
    返回將plaintext的所有元素連接成一個字符串  # 將所有明文字符連接成一個字符串並返回
```

```
cassidy@cassidydeMacBook-Air Quiz04 % python3 problem2.py
Plaintext: ATNYCUWEARESTRIVINGTOBEAGREATUNIVERSITYTHATTRANSCENDSDISCIPLINARYDIVIDESTOSOLVETHEINCREASINGLYCOMPLEXPROBLEMSTHATTHEWORLDFACESWEWILLCONTINUETOBEG
UIDEDBYTHEIDEATHATWECANACHIEVESOMETHINGMUCHGREATERTOGETHERTHANWECANINDIVIDUALLYAFTERALLTHATWASTHEIDEATHATLEDTOTHECREATIONOFOURUNIVERSITYINTHEFIRSTPLACE
Decrypted text: ATNYCUWEARESTRIVINGTOBEAGREATUNIVERSITYTHATTRANSCENDSDISCIPLINARYDIVIDESTOSOLVETHEINCREASINGLYCOMPLEXPROBLEMSTHATTHEWORLDFACESWEWILLCONTINUE
TOBEGUIDEDBYTHEIDEATHATWECANACHIEVESOMETHINGMUCHGREATERTOGETHERTHANWECANINDIVIDUALLYAFTERALLTHATWASTHEIDEATHATLEDTOTHECREATIONOFOURUNIVERSITYINTHEFIRSTPLACE
```

(b) Yes, 但要同時有明文和密文,

根據已知的明文和密文設一個線性方程式, 未知數為 primitive polynomial,

再代入每位明文, 密文求解。

(primitive polynomial 越高次, 則需要的已知明文, 密文對便越多)#

Problem3

(a) 引用 random, itertools 的 permutation
直接跟著 pseudo code 去做　　執行 python3 problem3.py
多設一個 function 去計算不同排組出現的次數

```
function count_combinations(shuffle_function, iterations):
    # 初始化一個空字典來存儲不同牌組的組合數量
    counts = {}
    # 進行指定次數的迭代
    for _ in range(iterations):
        # 使用給定的洗牌函數對牌組進行洗牌
        shuffled_cards = shuffle_function([1, 2, 3, 4])
        # 如果已經有相同的牌組出現過，則將其組合數加1；否則，初始化為1
        if shuffled_cards 已存在於 counts:
            counts[shuffled_cards] = counts[shuffled_cards] + 1
        else:
            counts[shuffled_cards] = 1
    # 返回存儲不同牌組的組合數量的字典
    return counts
```

(b) Fisher-Yates 較好, ∵每種牌出現的次數較平均(有隨機性), 效率較好(O(n))

```
cassidy@cassidydeMacBook-Air Quiz04 % python3 problem3.py
Naive algorithm:
(2, 3, 4, 1): 78150
(1, 3, 2, 4): 78800
(3, 2, 1, 4): 62450
(2, 1, 3, 4): 78306
(2, 4, 1, 3): 30996
(3, 1, 4, 2): 46880
(3, 4, 2, 1): 46910
(3, 4, 1, 2): 30806
(4, 3, 1, 2): 31407
(1, 4, 3, 2): 31418
(1, 2, 3, 4): 62133
(3, 1, 2, 4): 62564
(1, 2, 4, 3): 31217
(4, 2, 1, 3): 15664
(4, 2, 3, 1): 31116
(1, 4, 2, 3): 15668
(2, 3, 1, 4): 78005
(2, 4, 3, 1): 31532
(3, 2, 4, 1): 31136
(2, 1, 4, 3): 31092
(1, 3, 4, 2): 31142
(4, 3, 2, 1): 31458
(4, 1, 2, 3): 15445
(4, 1, 3, 2): 15705
```

```
Fisher-Yates shuffle:
(1, 4, 3, 2): 41810
(3, 2, 4, 1): 41477
(4, 3, 2, 1): 41795
(3, 1, 2, 4): 41474
(3, 1, 4, 2): 41690
(1, 2, 3, 4): 41729
(3, 2, 1, 4): 41806
(4, 2, 1, 3): 41597
(2, 4, 1, 3): 41683
(4, 2, 3, 1): 41462
(1, 4, 2, 3): 42037
(1, 3, 4, 2): 41826
(2, 3, 1, 4): 41553
(1, 2, 4, 3): 41463
(2, 1, 3, 4): 41499
(4, 1, 2, 3): 41782
(2, 3, 4, 1): 41591
(2, 4, 3, 1): 41641
(2, 1, 4, 3): 41917
(3, 4, 2, 1): 41684
(4, 3, 1, 2): 41356
(1, 3, 2, 4): 41746
(4, 1, 3, 2): 41668
(3, 4, 1, 2): 41714
```

(c) Naive 則較分散, 且標準差大, 某些牌組可能出現 70000 多次但有些才 10000 多 (猜牌較容易)
① 缺少 random → 因為每張牌每個位置都可排
(對它而言有 4^4 種可能, 但實際只有 4! ∴不會每種可能平均分布)
② 效率 → ∵每次都是考慮"每個位置" ∴當牌組數量大花費時間和 Fisher-Yates 相比較多
(Fisher-Yates: 只考慮 0 到自己當前所在位置)。