Quiz 4

學號：111550129　姓名：林彥亨

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?

(a)

yes.

(b)

The given polynomial is degree 8. Hence, the maximum cycle length is $2^8 - 1$= 255.

(c)

No, not all irreducible polynomials are primitive polynomials.

While every primitive polynomial is irreducible (meaning it cannot be factored into lower-degree polynomials over the same field), not every irreducible polynomial is primitive.

Problem 2

Given the plaintext:

ATNYCUWEARESTRIVINGTOBEAGREATUNIVERSITYTHATTRAN

SCENDSDISCIPLINARYDIVIDESTOSOLVETHEINCREASINGLYCO

MPLEXPROBLEMSTHATTHEWORLDFACESWEWILLCONTINUET

OBEGUIDEDBYTHEIDEATHATWECANACHIEVESOMETHINGMU

CHGREATERTOGETHERTHANWECANINDIVIDUALLYAFTERALLT

HATWASTHEIDEATHATLEDTOTHECREATIONOFOURUNIVERSI

TYINTHEFIRSTPLACE

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)

- The function to do the LFSR

```python
import numpy as np

def initialize_lfsr(seed, size=8):
    """Initializes the LFSR with the given binary seed."""
    state = np.array([int(bit) for bit in seed], dtype=int)
    return np.roll(state, -size)

def step_lfsr(state, polynomial):
    """Performs one step of the LFSR."""
    new_bit = np.bitwise_xor.reduce(state[polynomial])
    state = np.roll(state, -1)
    state[0] = new_bit
    return state

def generate_keystream(initial_state, polynomial, length):
    """Generates a binary keystream."""
    state = initialize_lfsr(initial_state)
    keystream = []
    for _ in range(length):
        state = step_lfsr(state, polynomial)
        keystream.append(state[0])
    return keystream

def encrypt_decrypt(message, keystream):
    """Encrypts or decrypts a binary message."""
    message_bits = np.array([int(bit) for bit in message], dtype=int)
    encrypted_decrypted = np.bitwise_xor(message_bits, keystream)
    result = ''.join(str(bit) for bit in encrypted_decrypted)
    return result
```

- This is the main function to output the answer.

```python

def main():
    plaintext = "ATNYCUWEARESTRIVINGTOBEAGREATUNIVERSITYTHATTRANSCENDSDISCIPLINARYDIVIDESTOSOLVETHEINCREASINGLYCOMPLEXPROBLEMSTHATTHE
    initial_key = "00000001"
    binary_message = ''.join(format(ord(char), '08b') for char in plaintext)
    characteristic_polynomial = [7, 3, 2, 1]
    keystream = generate_keystream(initial_key, characteristic_polynomial, len(binary_message))
    encrypted_message = encrypt_decrypt(binary_message, keystream)
    decrypted_message = encrypt_decrypt(encrypted_message, keystream)
    decrypted_text = ''.join(chr(int(decrypted_message[i:i+8], 2)) for i in range(0, len(decrypted_message), 8))
    print("Encrypted Message (Binary):\n", encrypted_message)
    print("Decrypted Text:\n", decrypted_text)

main()
```

- Here is the result：

```
PS C:\Users\user\Desktop\密碼工q4> & C:/ProgramData/anaconda3/python.exe c:/Users/user/Desktop/密碼工q4/problem2.py
Encrypted Message (Binary):
 111001010011000001111110111001010111100101111000111111011010101000011011010101011010100111111111000
0010110010110000001110010000010110110011111111010100101011100011110101011011010110111000011011001010
1110010010100010000010011011001100100110100010000010101000100000010010000000100101111111100100101010 1
0001000000111101111110010101010011001100110111100101011110010101011011001001010010010011011100100111 10
0011010000010110100011110110001001110000000100101001001111011001100101111011000111001000111111010110 0
1011111101001110100010011011110101110100100111010110101101100111111010100111110010111111001101010111 1
0001100100111000011000101011100011111001010001000000100011100010111001000010101001101010100000000111
1010000000111010100010000011101011101011011001110100110111101111000110100000111000111110101010011110 0
0100110010101011011101100110100010000101111010110000101100010110101111001000100100000101101001010100
1011111110110100100101110101110100010111011110010110010000110100000001010110100000001011010010000101 01
0101000000010010011010010111010100000101100000000110110100010001110011001101111111001000000101101010011
0111111010000101001001000010100100010000101010011111110010111011010100010110000100110110010110110011
0001100011110111010101101110001110011010100001000111010111110001011010011001100101101110010011001010
1111011000110110101011000000110110001101001101111011111110100011110101001000010101101011100100111101 0
0000101000101111001110100100111101010100010001111111100010110111010010000100111101001110001110000000
0001000101011010101110000110110001001001101010011110100100110000011110001101010011001111100010101101
1100100010001110111111000111011100110101001010110010110000001101001001100101010011001010101010011110 1
0110111010111111000010111111101010100111011010010000000110010111111101011101100110000111110011101011 1
0000111000101001110101001111111100001000010001000000000010001000101011011101110000101110100101111101 1
1010010100111101001100000001001001011111010010010001010001011101110011010011000001011001011100010000 1
0100110010011001011101001010101111011100100000001001100000101001110110010001101001001110101101100110 1
0001101001101010101011011010000110110011000000001001100001100011010010111111000101011101000101011110 0
1010010111101100100111001111100101000000101101100010110010011001010100000000111001010101101101101111 1
10110100111100010100011010110010111010011000101111101110100000111000 1
Decrypted Text:
 ATNYCUWEARESTRIVINGTOBEAGREATUNIVERSITYTHATTRANSCENDSDISCIPLINARYDIVIDESTOSOLVETHEINCREASINGLYCOMPL
EXPROBLEMSTHATTHEWORLDFACESWEWILLCONTINUETOBEGUIDEDBYTHEIDEATHATWECANACHIEVESOMETHINGMUCHGREATERTOGE
THERTHANWECANINDIVIDUALLYAFTERALLTHATWASTHEIDEATHATLEDTOTHECREATIONOFOURUNIVERSITYINTHEFIRSTPLACE
```

(b)

Yes, it is theoretically possible to determine the characteristic polynomial of an LFSR by analyzing its output bits and solving a system of linear equations. This process is known as Berlekamp-Massey algorithm. Given a sufficient length of known output bits (which, in this case, can be deduced from the encrypted ASCII text as described), the Berlekamp-Massey algorithm can be used to find the shortest LFSR and its characteristic polynomial that could produce such a sequence.

This ability to reverse-engineer the LFSR from its output underlies some of the weaknesses in using LFSRs for encryption, especially if the plaintext has known or predictable patterns (like ASCII text with MSB = 0). That's why in practical cryptographic applications, pure LFSRs are rarely used on their own. They are typically combined with other operations and structures to form more complex and secure stream ciphers.

Problem 3

RC4's vulnerability mainly arises from its inadequate randomization of inputs, particularly the initialization vector (IV) and key integration, due to its reliance on the initial setup by its Key Scheduling Algorithm (KSA). The cipher operates through two phases: KSA, which shuffles a 256-byte state vector based on the key to ensure dependency and randomization, and the Pseudo-Random Generation Algorithm (PRGA), where it further manipulates this state to produce a seemingly random output stream.

To help you understand the importance of randomization algorithms, here we provide the pseudocode for two slightly different shuffle algorithms.

Naïve algorithm:

```
For i from 0 to length(cards)-1
    Generate a random number n between 0 and length(cards)-1
    Swap the elements at indices i and n
EndFor
```

Fisher–Yates shuffle (Knuth shuffle):

```
For i from length(cards)-1 down to 1
    Generate a random number n between 0 and i
    Swap the elements at indices i and n
EndFor
```

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?

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

(a)

- Here is the code to simulate the two algorithms.

```python
import random
from collections import defaultdict
```

```python
# Define the naive shuffle algorithm
def naive_shuffle(cards):
    for i in range(len(cards)):
        n = random.randint(0, len(cards) - 1)
        cards[i], cards[n] = cards[n], cards[i]
    return cards

# Define the Fisher-Yates shuffle algorithm
def fisher_yates_shuffle(cards):
    for i in range(len(cards) - 1, 0, -1):
        n = random.randint(0, i)
        cards[i], cards[n] = cards[n], cards[i]
    return cards

# Function to simulate shuffling a million times and collect results
def simulate_shuffling():
    naive_results = defaultdict(int)
    fy_results = defaultdict(int)

    for _ in range(1000000):  # Run the simulation a million times
        # Naive shuffle
        deck = [1, 2, 3, 4]
        shuffled_deck = naive_shuffle(deck.copy())
        naive_results[tuple(shuffled_deck)] += 1

        # Fisher-Yates shuffle
        deck = [1, 2, 3, 4]
        shuffled_deck = fisher_yates_shuffle(deck.copy())
        fy_results[tuple(shuffled_deck)] += 1

    return naive_results, fy_results
```

```python

# Output the results
print("Naive algorithm results:")
for outcome, count in sorted(naive_outcomes.items()):
    print(f"{list(outcome)}: {count}")
    naive = 0
    naive = naive + count
naive % 24
print("The average number of suffle:", naive)
print("\nFisher-Yates shuffle results:")
for outcome, count in sorted(fy_outcomes.items()):
    print(f"{list(outcome)}: {count}")
    fy = 0
    fy = fy + count
fy % 24
print("The average number of suffle:", fy)
```

- Here is the results of Naive aligorithm and Fisher-Yates algorithm.

```
PS C:\Users\user\Desktop\密碼工q4> & C:/ProgramData/anaconda3/python.exe c:/Users/user/Desktop/密碼工q4/problem3.py
Naive algorithm results:
[1, 2, 3, 4]: 38945
[1, 2, 4, 3]: 39298
[1, 3, 2, 4]: 39425
[1, 3, 4, 2]: 54617
[1, 4, 2, 3]: 43068
[1, 4, 3, 2]: 35134
[2, 1, 3, 4]: 38955
[2, 1, 4, 3]: 58714
[2, 3, 1, 4]: 54656
[2, 3, 4, 1]: 54950
[2, 4, 1, 3]: 42717
[2, 4, 3, 1]: 42986
[3, 1, 2, 4]: 43204
[3, 1, 4, 2]: 42963
[3, 2, 1, 4]: 35179
[3, 2, 4, 1]: 43075
[3, 4, 1, 2]: 43048
[3, 4, 2, 1]: 38804
[4, 1, 2, 3]: 31247
[4, 1, 3, 2]: 35131
[4, 2, 1, 3]: 34843
[4, 2, 3, 1]: 31158
[4, 3, 1, 2]: 39057
[4, 3, 2, 1]: 38826
The average number of suffle: 38826
```

```
Fisher-Yates shuffle results:
[1, 2, 3, 4]: 41592
[1, 2, 4, 3]: 41724
[1, 3, 2, 4]: 41968
[1, 3, 4, 2]: 41523
[1, 4, 2, 3]: 41668
[1, 4, 3, 2]: 41544
[2, 1, 3, 4]: 41450
[2, 1, 4, 3]: 41661
[2, 3, 1, 4]: 41912
[2, 3, 4, 1]: 41755
[2, 4, 1, 3]: 41818
[2, 4, 3, 1]: 41567
[3, 1, 2, 4]: 41836
[3, 1, 4, 2]: 41725
[3, 2, 1, 4]: 41436
[3, 2, 4, 1]: 41585
[3, 4, 1, 2]: 41441
[3, 4, 2, 1]: 41642
[4, 1, 2, 3]: 41908
[4, 1, 3, 2]: 41612
[4, 2, 1, 3]: 41318
[4, 2, 3, 1]: 41889
[4, 3, 1, 2]: 41582
[4, 3, 2, 1]: 41844
The average number of suffle: 41844
PS C:\Users\user\Desktop\密碼工q4>
```

(b)

According to the average number of the results, for the naive algorithm the average numbe is 38826; for the Fisher-Yates algorithm the average number is 41844, hence, the Fisher-Yates shuffle is better. Because it ensures each permutation of the deck is equally likely, leading to a uniform and unbiased distribution. In contrast, the Naive shuffle can lead to non-uniform distributions and biased results, making it less suitable for applications where fairness and randomness are critical.

(c)

Through the problem 3, the main drawbacks of the Naive shuffle algorithm are non-uniform distribution and biased results. These issues are caused by the algorithm’s approach of allowing each card to swap with any other card, including itself, which does not ensure all permutations have equal probability.