Quiz 4

Problem 1

a. Is $f(x) = x^8 + x^4 + x^3 + x^2 + 1$ a primitive polynomial?

Ans. Yes!

Explanation:

1. $f(x)$ is irreducible over $GF(2)$

   First we could check whether $f(x)$ is irreducible by trying its divisibility by irreducible polynomials of lower degree (1 to 4) over $GF(2)$, such as x, $x+1$, x^2+x+1, x^3+x+1 and x^3+x^2+1. Then we can easily know that $f(x)$ is irreducible.

2. $f(x)$ has order $2^8 - 1$.

   We need to ensure that $x^{2^8-1} \equiv 1 \mod (x^8 + x^4 + x^3 + x^2 + 1)$ for all divisors of $2^8 - 1 = 255$. The divisors of 255 are $1, 3, 5, 15, 17, 51, 85, 255$. For the polynomial to be primitive, it must not satisfy $x^d \equiv 1$ for any d that is a divisor of 255, except for $d = 255$ itself.

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

Ans. $2^8 - 1 = 255$

c. Are all irreducible polynomials primitive polynomials?

Ans. Not all irreducible polynomials primitive polynomials, it should also generate the multiplicative group of the finite field $GF(2^n)$. Consider the polynomial $x^8 + x^4 + x^3 + x + 1$, which is irreducible over $GF(2)$. However, it is not primitive. While it does create a finite field $GF(2^4)$, it does not generate all non-zero elements of $GF(2^8)$ through powers of its root. The order of any root of this polynomial is not $2^8 - 1 = 255$, rather, it is 51, as $x^{51} \equiv 1 \mod (x^8 + x^4 + x^3 + x + 1)$. indicating that the sequence generated by its roots cycles every 5 elements, not 15. Therefore, $x^8 + x^4 + x^3 + x + 1$ over $GF(2)$ is irreducible but not primitive.

Problem 2

a. Code:

```
def lfsr(seed, taps, length):
    """Generate a keystream using LFSR given a seed, tap positions, and desired length."""
    sr, xor = seed, 0
    keystream = []
    for _ in range(length):
        for t in taps:
            xor ^= int(sr[t-1])     # XOR tap positions
        keystream.append(sr[-1])    # Append the output bit
        sr = str(xor) + sr[:-1]     # Shift register
        xor = 0                     # Reset xor
    return ''.join(keystream)

def text_to_bits(text):
    """Convert text to its binary representation."""
    return ''.join(format(ord(i), '08b') for i in text)

def bits_to_text(bits):
    """Convert binary representation back to text."""
    return ''.join(chr(int(bits[i:i+8], 2)) for i in range(0, len(bits), 8))

def xor_strings(s1, s2):
    """XOR two binary strings."""
    return ''.join(str(int(a) ^ int(b)) for a, b in zip(s1, s2))
```

```
# Given parameters
plaintext = """
ATNYCUWEARESTRIVINGTOBEAGREATUNIVERSITYTHATTRAN
SCENDSDISCIPLINARYDIVIDESTOSOLVETHEINCREASINGLYCO
MPLEXPROBLEMSTHATTHEWORLDFACESWEWILLCONTINUET
OBEGUIDEDBYTHEIDEATHATWECANACHIEVESOMETHINGMU
CHGREATERTOGETHERTHANWECANINDIVIDUALLYAFTERALLT
HATWASTHEIDEATHATLEDTOTHECREATIONOFOURUNIVERSI
TYINTHEFIRSTPLACE
"""
initial_key = "00000001"
# Convert plaintext to binary
plaintext_binary = text_to_bits(plaintext)

# Define the characteristic polynomial's tap positions (note: positions are from the end o
f the register)
tap_positions = [8, 4, 3, 2]  # Corresponding to x^8 + x^4 + x^3 + x^2 + 1

# Generate keystream
keystream = lfsr(initial_key, tap_positions, len(plaintext_binary))

# Encrypt the plaintext
ciphertext_binary = xor_strings(plaintext_binary, keystream)

# Decrypt the ciphertext (the operation is identical due to the symmetry of XOR)
decrypted_binary = xor_strings(ciphertext_binary, keystream)

# Convert binary back to text
decrypted_text = bits_to_text(decrypted_binary)

# Check if decryption is correct
if decrypted_text == plaintext:
    print("Decryption successful, plaintext is:", decrypted_text)
else:
    print("Decryption failed.")
```

Result:

```
Decryption successful, plaintext is:
ATNYCUWEARESTRIVINGTOBEAGREATUNIVERSITYTHATTRAN
SCENDSDISCIPLINARYDIVIDESTOSOLVETHEINCREASINGLYCO
MPLEXPROBLEMSTHATTHEWORLDFACESWEWILLCONTINUET
OBEGUIDEDBYTHEIDEATHATWECANACHIEVESOMETHINGMU
CHGREATERTOGETHERTHANWECANINDIVIDUALLYAFTERALLT
HATWASTHEIDEATHATLEDTOTHECREATIONOFOURUNIVERSI
TYINTHEFIRSTPLACE
```

b. Yes, we can find out the characteristic polynomial by the following process:
    a. **Collect Keystream Bits**: We need a sufficient number of output bits from the LFSR. For a LFSR of length n, we need at least $2n$ bits of the output sequence to reliably find the characteristic polynomial.
    b. **Set Up Linear Equations**: Based on the collected bits and the property that the MSB is always zero for ASCII characters 'A' to 'Z', we can set up a system of linear equations. Each equation corresponds to a step in the LFSR's operation, relating current and previous states to the output bit.
    c. **Solve the System**: The system of linear equations can be solved using various methods to find the coefficients of the characteristic polynomial. These coefficients indicate which taps are used for feedback in the LFSR.

d. **Apply Berlekamp-Massey Algorithm**: Instead of manually solving the system, we can apply the Berlekamp-Massey algorithm directly to the collected sequence. This algorithm efficiently finds the shortest LFSR sequence and the corresponding characteristic polynomial.

Problem 3

a. Code:

```
import random
import itertools
random.seed(1)

card = [1, 2, 3, 4]
combinations = list(itertools.permutations(card, 4))

def naive():
    print("Naive algorithm:")
    my_map = {}
    for per in combinations:
        my_map[per] = 0
    for _ in range(1000000):
        card = [1, 2, 3, 4]
        for i in range(4):
            n = random.randint(0, 3)
            card[i], card[n] = card[n], card[i]
        my_map[tuple(card)] += 1
    for key in my_map:
        print(f"{key}: {my_map[key]} times")
def fisher():
    print("Fisher-Yates shuffle:")
    my_map = {}
    for per in combinations:
        my_map[per] = 0
    for _ in range(1000000):
        card = [1, 2, 3, 4]
        for i in range(3, 0, -1):
            n = random.randint(0, i)
            card[i], card[n] = card[n], card[i]
        my_map[tuple(card)] += 1
    for key in my_map:
        print(f"{key}: {my_map[key]} times")

if __name__ == "__main__":
    naive()
    fisher()
```

Result:

```
Naive algorithm:
(1, 2, 3, 4): 39502 times
(1, 2, 4, 3): 39137 times
(1, 3, 2, 4): 38723 times
(1, 3, 4, 2): 54730 times
(1, 4, 2, 3): 43103 times
(1, 4, 3, 2): 35270 times
(2, 1, 3, 4): 38914 times
(2, 1, 4, 3): 58552 times
(2, 3, 1, 4): 54573 times
```

```
(2, 3, 4, 1): 54709 times
(2, 4, 1, 3): 43106 times
(2, 4, 3, 1): 43163 times
(3, 1, 2, 4): 42705 times
(3, 1, 4, 2): 43370 times
(3, 2, 1, 4): 35340 times
(3, 2, 4, 1): 43155 times
(3, 4, 1, 2): 42809 times
(3, 4, 2, 1): 38775 times
(4, 1, 2, 3): 31100 times
(4, 1, 3, 2): 35039 times
(4, 2, 1, 3): 35163 times
(4, 2, 3, 1): 31339 times
(4, 3, 1, 2): 38710 times
(4, 3, 2, 1): 39013 times

Fisher-Yates shuffle:
(1, 2, 3, 4): 41552 times
(1, 2, 4, 3): 41523 times
(1, 3, 2, 4): 41886 times
(1, 3, 4, 2): 41731 times
(1, 4, 2, 3): 41790 times
(1, 4, 3, 2): 41495 times
(2, 1, 3, 4): 41406 times
(2, 1, 4, 3): 41675 times
(2, 3, 1, 4): 41615 times
(2, 3, 4, 1): 41658 times
(2, 4, 1, 3): 41581 times
(2, 4, 3, 1): 41680 times
(3, 1, 2, 4): 41860 times
(3, 1, 4, 2): 41753 times
(3, 2, 1, 4): 41944 times
(3, 2, 4, 1): 41574 times
(3, 4, 1, 2): 41417 times
(3, 4, 2, 1): 41864 times
(4, 1, 2, 3): 41697 times
(4, 1, 3, 2): 41443 times
(4, 2, 1, 3): 42188 times
(4, 2, 3, 1): 41632 times
(4, 3, 1, 2): 41545 times
(4, 3, 2, 1): 41491 times
```

b. Fisher-Yates shuffle is better, because it has been proven to produce a uniformly random permutation of a sequence when implemented correctly. It systematically decreases the range from which random indices are chosen, ensuring that each element has an equal chance of being placed in each position.

c. The **naive algorithm** for shuffling may not produce a uniformly random distribution of permutations, in other words, include the potential for unequal probabilities for different permutations. This can be because the range from which random indices are chosen doesn't shrink as the algorithm progresses, which is crucial for ensuring every permutation is equally likely.