1. Write a python script to get the binary values from the user and perform XOR operation.

```python
def is_binary(s):
    """Check if a string consists only of '0's and '1's."""
    return set(s) <= {'0', '1'}

def binary_xor(a, b):
    """Perform XOR operation on two binary strings of equal length."""
    return ''.join('1' if a[i] != b[i] else '0' for i in range(len(a)))

def get_binary_input(prompt):
    """Get a valid binary input from the user."""
    while True:
        value = input(prompt)
        if is_binary(value):
            return value
        else:
            print("Invalid input. Please enter a binary number (consisting of only 0s and 1s).")

# Get binary inputs from the user
binary1 = get_binary_input("Enter the first binary number: ")
binary2 = get_binary_input("Enter the second binary number: ")

# Ensure both binary strings are of equal length
max_length = max(len(binary1), len(binary2))
binary1 = binary1.zfill(max_length)
binary2 = binary2.zfill(max_length)

# Perform XOR operation
result = binary_xor(binary1, binary2)

# Print the result
print(f"\nFirst binary number:  {binary1}")
print(f"Second binary number: {binary2}")
print(f"XOR result:           {result}")

# Optional: Convert to decimal for additional context
print(f"\nDecimal equivalents:")
print(f"First number:  {int(binary1, 2)}")
print(f"Second number: {int(binary2, 2)}")
print(f"XOR result:    {int(result, 2)}")
```

Output:

```
┌──(kali㉿kali)-[~/CYS/HYM]
└─$ vi lab9(1).py

┌──(kali㉿kali)-[~/CYS/HYM]
└─$ python3 lab9(1).py
Enter the first binary number: 0101100
Enter the second binary number: 100111

First binary number:  0101100
Second binary number: 0100111
XOR result:           0001011

Decimal equivalents:
First number:  44
Second number: 39
XOR result:    11
```

2. Write a Python script that implements a simple 4-bit LFSR. The initial state of the register and the tap positions should be user inputs.

Simulate 10 steps of the LFSR, displaying the state of the register at each step.

```python
def get_binary_input(prompt, length):
    """Get a valid binary input of specified length from the user."""
    while True:
        value = input(prompt)
        if set(value) <= {'0', '1'} and len(value) == length:
            return [int(bit) for bit in value]
        else:
            print(f"Invalid input. Please enter a {length}-bit binary number.")
def get_tap_positions():
    """Get tap positions from the user."""
    while True:
        taps = input("Enter tap positions (comma-separated, e.g., 1,3): ")
        try:
            tap_list = [int(tap) for tap in taps.split(',')]
            if all(0 <= tap < 4 for tap in tap_list):
                return tap_list
            else:
                print("Invalid tap positions. Please enter numbers between 0 and 3.")
        except ValueError:
            print("Invalid input. Please enter comma-separated numbers.")

def lfsr_step(state, taps):
    """Perform one step of the LFSR."""
    feedback = sum(state[tap] for tap in taps) % 2
    return [feedback] + state[:-1]

# Get initial state from the user
initial_state = get_binary_input("Enter the initial 4-bit state: ", 4)

# Get tap positions from the user
tap_positions = get_tap_positions()

# Simulate 10 steps of the LFSR
current_state = initial_state
print("\nLFSR Simulation:")
print(f"Initial state: {current_state}")

for step in range(1, 11):
    current_state = lfsr_step(current_state, tap_positions)
    print(f"Step {step}: {current_state}")

# Calculate the output sequence
output_sequence = [initial_state[-1]] + [state[-1] for state in [lfsr_step(current_state, tap_positions) for _ in range(9)]]
print(f"\nOutput sequence: {output_sequence}")
```

Output:

```
┌──(kali㉿kali)-[~/CYS/HYM]
└─$ vi lab9(2).py

┌──(kali㉿kali)-[~/CYS/HYM]
└─$ python3 lab9(2).py
Enter the initial 4-bit state: 1001
Enter tap positions (comma-separated, e.g., 1,3): 1,3

LFSR Simulation:
Initial state: [1, 0, 0, 1]
Step 1: [1, 1, 0, 0]
Step 2: [1, 1, 1, 0]
Step 3: [1, 1, 1, 1]
Step 4: [0, 1, 1, 1]
Step 5: [0, 0, 1, 1]
Step 6: [1, 0, 0, 1]
Step 7: [1, 1, 0, 0]
Step 8: [1, 1, 1, 0]
Step 9: [1, 1, 1, 1]
Step 10: [0, 1, 1, 1]

Output sequence: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

3. Write a report on attacks on LFSR. Explain any one attack in detail.

Attacks on Linear Feedback Shift Registers (LFSRs)

Introduction

Linear Feedback Shift Registers (LFSRs) are widely used in cryptography, particularly in stream ciphers and pseudo-random number generators. However, their linear nature makes them vulnerable to several types of attacks. This report outlines the major attacks on LFSRs and provides a detailed explanation of one particularly effective attack.

Common Attacks on LFSRs

1. Berlekamp-Massey Algorithm Attack: Used to determine the shortest LFSR that can produce a given output sequence.

2. Algebraic Attack: Exploits the algebraic structure of the LFSR to set up and solve a system of linear equations.

3. Correlation Attack: Targets stream ciphers that use multiple LFSRs, exploiting statistical weaknesses.

4. Fast Correlation Attack: An improved version of the correlation attack, using error-correcting codes.

5. Known Plaintext Attack: Uses known plaintext-ciphertext pairs to deduce the LFSR state.

6. Divide-and-Conquer Attack: Applies to stream ciphers using multiple LFSRs, attacking each LFSR separately.

Detailed Analysis: The Berlekamp-Massey Algorithm Attack

The Berlekamp-Massey algorithm is a powerful method for cryptanalyzing LFSR-based systems. It can determine the shortest LFSR capable of generating a given output sequence, effectively reverse engineering the LFSR structure.

How the Attack Works

1. Observation: The attacker observes a portion of the LFSR's output sequence.

2. Algorithm Application: The Berlekamp-Massey algorithm is applied to this sequence.

3. LFSR Reconstruction: The algorithm determines:

   - The length of the shortest LFSR that could produce the sequence

   - The feedback polynomial of this LFSR

4. State Recovery: Once the LFSR structure is known, the initial state can often be determined by solving a system of linear equations.

Mathematical Basis

The Berlekamp-Massey algorithm works by iteratively constructing a Linear Recurrence Relation (LRR) that generates the observed sequence. At each step, it either:

- Increases the length of the LFSR if the current one can't generate the next bit, or

- Adjusts the feedback taps without increasing length if possible

The algorithm's complexity is O(n^2), where n is the length of the observed sequence.

Example: Consider a 4-bit LFSR with the output sequence: 1011010110...

1. The algorithm starts with a 1-bit LFSR and progressively increases its size.

2. It might determine that a 4-bit LFSR with feedback polynomial x^4 + x^3 + 1 generates this sequence.

3. The attacker can then deduce that the original LFSR likely had this structure.

Implications for Cryptography

1. Vulnerability of Simple LFSRs: This attack demonstrates why single, short LFSRs are inadequate for secure cryptographic systems.

2. Required Sequence Length: To break an n-bit LFSR, approximately 2n bits of output are needed.

3. Countermeasures

- Use longer LFSRs (e.g., 128 bits or more)
- Combine multiple LFSRs in non-linear ways
- Introduce irregular clocking or output functions

Conclusion

The Berlekamp-Massey algorithm attack, along with other attacks on LFSRs, highlights the importance of using more complex structures in cryptographic systems. While LFSRs remain useful components, they must be employed carefully and in combination with other techniques to ensure cryptographic security.

Understanding these attacks is crucial for cryptographers to design more robust systems and for security analysts to assess the strength of existing cryptographic implementations.

BONUS POINT:

4. write a python script to break hill cipher (2X2) using known plain text attack.

Known Plaintext: "MEET"

Corresponding Ciphertext: "URRG"

```python
import numpy as np

def letter_to_number(letter):
    return ord(letter) - ord('A')

def number_to_letter(number):
    return chr(number % 26 + ord('A'))

def matrix_mod_inv(matrix, modulus):
    """Calculate the modular inverse of a 2×2 matrix"""
    det = int(np.round(np.linalg.det(matrix)))
    det_inv = pow(det % modulus, -1, modulus)
    adjugate = np.array([[matrix[1, 1], -matrix[0, 1]],
                         [-matrix[1, 0], matrix[0, 0]]])
    return (adjugate * det_inv) % modulus

# Known plaintext and ciphertext
plaintext = "MEET"
ciphertext = "URRG"

# Convert plaintext and ciphertext to number matrices
P = np.array([[letter_to_number(plaintext[0]), letter_to_number(plaintext[2])],
              [letter_to_number(plaintext[1]), letter_to_number(plaintext[3])]])

C = np.array([[letter_to_number(ciphertext[0]), letter_to_number(ciphertext[2])],
              [letter_to_number(ciphertext[1]), letter_to_number(ciphertext[3])]])

# Calculate the key matrix
P_inv = matrix_mod_inv(P, 26)
K = (C @ P_inv) % 26

print("Recovered Key Matrix:")
print(K)

# Verify the key by encrypting the plaintext
def encrypt(plaintext, key):
    result = ""
    for i in range(0, len(plaintext), 2):
        chunk = np.array([letter_to_number(plaintext[i]),
                          letter_to_number(plaintext[i+1])])
        encrypted = (key @ chunk) % 26
        result += number_to_letter(encrypted[0]) + number_to_letter(encrypted[1])
    return result

encrypted = encrypt(plaintext, K)
print(f"\nVerification:")
print(f"Original Plaintext: {plaintext}")
print(f"Known Ciphertext:   {ciphertext}")
print(f"Encrypted with recovered key: {encrypted}")
```

Output: