

Report on LFSR Implementation

Made with love my Mousa Emarah.

1. How the LFSR Works

A **Linear Feedback Shift Register (LFSR)** is a shift register where the input bit is a linear function of its previous state. It is commonly used to generate pseudorandom sequences. Here's how it works:

- **Shift Register:** The LFSR consists of a series of bits (the seed) that shift one position to the left or right with each step.
- **Feedback:** The new bit that enters the register is calculated using a feedback function. This function XORs specific bits (called **tap positions**) from the current state.
- **Pseudorandom Sequence:** By repeatedly shifting and applying the feedback function, the LFSR generates a sequence of bits that appears random.

In our implementation:

- The **seed** is the initial state of the LFSR.
- The **tap positions** are specified by the user and determine which bits are used for feedback.
- The `step()` method generates the next bit in the sequence by XORing the tap positions and shifting the register.

For example, Taps are **s0** and **s1**

110**1**

?**1**0 -> 1

?**1**1 -> 0

<https://linkedin.com/in/mousa123>

Example:

Plaintext: mousa

Remember the 100 pseudo generated bits? The main keystream is now generated to match the length of the plaintext, by iterating through each bit of the plaintext, the LFSR produces a keystream of the same size.

This keystream is then XORed with the plaintext to produce the encrypted text, ensuring that the encrypted output has the same number of bits as the plaintext. This process guarantees consistency and correctness in both encryption and decryption.

Keystream: 0011011101010000100101100111110001101110

Encrypted text (using keystream): 010110100011111111000110000111100001111

Decryption:

- The ciphertext is XORed with the same keystream to recover the plaintext.
- The binary result is converted back to text using the `binaryToText()` method.

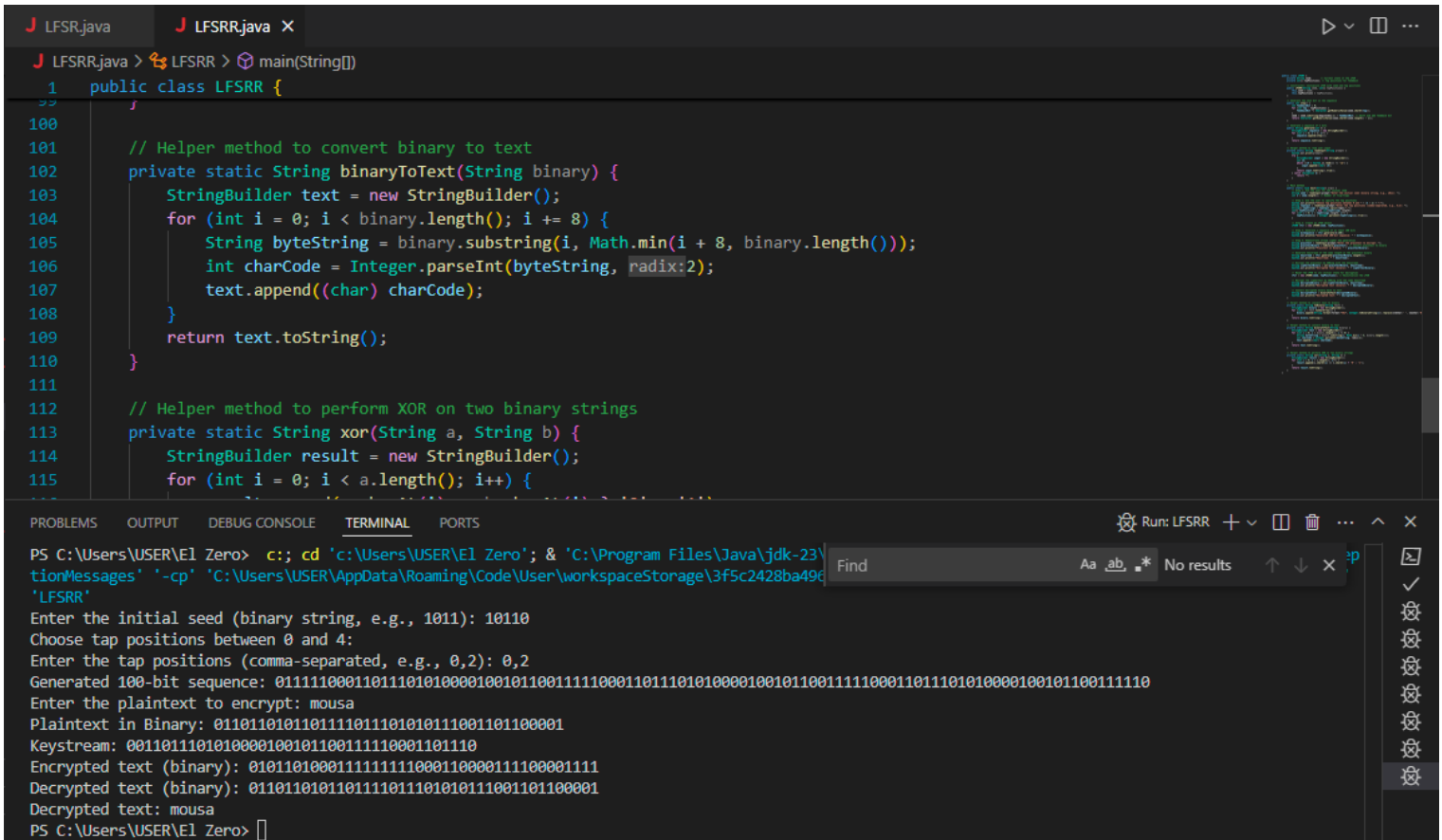
Decrypted text by XORing the encrypted text 00011000100000001011100011001000 with the emitted keystream as clarified previously will give us

0110110101101111011101010111001101100001, in which translated to a human form:

mousa

-> (successfully decrypted and got the right plain text before encryption)

<https://linkedin.com/in/mousa123>



The screenshot shows an IDE with two tabs: LFSR.java and LFSRR.java. The LFSRR.java tab is active, displaying the following code:

```
1 public class LFSRR {
100
101     // Helper method to convert binary to text
102     private static String binaryToText(String binary) {
103         StringBuilder text = new StringBuilder();
104         for (int i = 0; i < binary.length(); i += 8) {
105             String byteString = binary.substring(i, Math.min(i + 8, binary.length()));
106             int charCode = Integer.parseInt(byteString, radix:2);
107             text.append((char) charCode);
108         }
109         return text.toString();
110     }
111
112     // Helper method to perform XOR on two binary strings
113     private static String xor(String a, String b) {
114         StringBuilder result = new StringBuilder();
115         for (int i = 0; i < a.length(); i++) {
```

The terminal window at the bottom shows the execution of the program:

```
PS C:\Users\USER\El Zero> c::; cd 'c:\Users\USER\El Zero'; & 'C:\Program Files\Java\jdk-23\
tionMessages' '-cp' 'C:\Users\USER\AppData\Roaming\Code\User\workspaceStorage\3f5c2428ba49d
'LFSRR'
Enter the initial seed (binary string, e.g., 1011): 10110
Choose tap positions between 0 and 4:
Enter the tap positions (comma-separated, e.g., 0,2): 0,2
Generated 100-bit sequence: 0111110001101110101000010010110011111000110111010100001001011001111100011011101010000100101100111110
Enter the plaintext to encrypt: mousa
Plaintext in Binary: 011011010110111101101010111001101100001
Keystream: 0011011101010000100101100111110001101110
Encrypted text (binary): 010110100011111111000110000111100001111
Decrypted text (binary): 01101101011011110110101110011011100001
Decrypted text: mousa
PS C:\Users\USER\El Zero>
```

Conclusion

The LFSR is a simple and effective way to generate pseudorandom sequences, which can be used for cryptographic applications like stream ciphers. By choosing appropriate tap positions and seeds, we can ensure the generated sequence is sufficiently random for encryption and decryption.