Cryptography Lab 1

Done By: Joori Eihab Alhindi

ID: 2310656

Table of Contents:

Implementing and Analyzing a Linear Feedback Shift Register (LFSR)

2.1 Explain how the polynomial can be shown to be irreducible.

$p(x) = x^7 + x^4 + x^2 + 1$

To ensure that the polynomial is irreducible, we must check if it can be broken into smaller polynomials that multiply together, if it can, so it will be reducible.

Check the polynomial is reducible by over GF (2):

The coefficients of the polynomial is binary

Keep in mind:

- 1 = exists of term
- 0 = term do not exist

To determine whether p(x) = $x^7 + x^4 + x^2 + 1$ is irreducible in GF(2), we can test if it has a factor , substituting x=1 gives p(1) = 1 + 1 + 1 + 1 = 4 ≡ 0 (mod 2) , since the result equal to zero , Therefore, (x + 1) is a factor, it will make the polynomial reducible. Finally, only primitive irreducible polynomials achieve maximum period $2^n - 1$

2.1) $P(x) = x^7 + x^4 + x^2 + 1$

we know that:
1 XOR 1 = 0
0 XOR 0 = 0
1 XOR 0 = 1
0 XOR 1 = 1

Step 1 Look for a root:

① $x = 0$ (wrong)

$= (0)^7 + (0)^4 + (0)^2 + 1$

$= 0 + 0 + 0 + 1$

$= 1 \neq 0$, so $x=0$ not root

② $x = 1$ (right)

$= (1)^7 + (1)^4 + (1)^2 + 1$

$1 + 1 + 1 + 1$

$= 4$ it's even so

$P(1) = 0$ ✓, so $x=1$ is root

Step 2 Find a factor:

$x=1$ is root, so the factor is $= (x-1)$. but we're dealing with modulo 2 "Sub is same as add", so our factor $= (x+1)$

Step 3 Divide $P(x)$ by $(x+1)$ to find other factor:

$P(x) = x^7 + x^4 + x^2 + 1$ same as $= x^7 + 0x^6 + 0x^5 + x^4 + 0x^3 + x^2 + 0x + 1$

multiply $x^6 + x^5 + x^4 + x + 1$ ← $(x+1)$

$x^7 + 0x^6 + 0x^5 + x^4 + 0x^3 + x^2 + 0x + 1$ — divide $\frac{x^7}{x} = x^6$

$(x^7 + x^6)$ −

$x^6 + 0x^5 + x^4 + 0x^3 + x^2 + 0x + 1$ — divide $\frac{x^6}{x} = x^5$

$(x^6 + x^5)$ −

$x^5 + x^4 + 0x^3 + x^2 + 0x + 1$ — divide $\frac{x^5}{x} = x^4$

$(x^5 + x^4)$ −

$0x^3 + x^2 + 0x + 1$ — divide $\frac{x^2}{x} = x$

$(x^2 + x)$ −

$x + 1$ — divide $\frac{x}{x} = 1$

$(x + 1)$ −

0

*Remember we are in mod 2 (XOR)

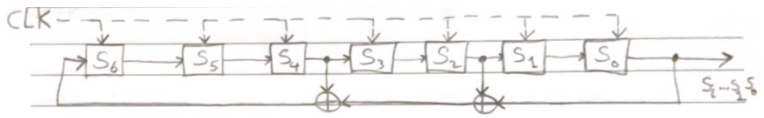
So, the result $= x^6 + x^5 + x^4 + x + 1$

$\frac{x^7 + x^4 + x^2 + 1}{x+1} = x^6 + x^5 + x^4 + x + 1$ Therefore, polynomial is reducible

2.2 Design

1. Design the LFSR for this polynomial by:

- showing the block diagram for the LFSR



- the complete recursion equations for the LFSR

First Recursive Equation: $S_{6,1} = S_{4,0} + S_{2,0} + S_{0,0}$

Complete and general Recursive Equation: $S_{6,i+1} = S_{4,i} + S_{2,i} + S_{0,i}$

- specifying the expected maximum possible period

  The polynomial will not be at maximum length(2^m-1) = 2^7-1=127, because it is reducible. So the LFSR will be stuck in small loops. I have done it in the code. You can see it in the output[2]

2.1 Implementing the LFSR:

1. Choose a Programming Language:

   I have used Java Language for implementing the LFSR.

2. Define the LFSR Parameters:

o Length of the LFSR (n).

Length of the LFSR is 63. Done in code. [1] see output [2].

o Feedback polynomial

Feedback polynomial: $x^7 = x^4 + x^2 + 1$

o Initial seed value (a binary number of length n).

I set (1, 0 , 0 , 0 , 0 , 0 , 0) as initial value with length of 7 .Also, I implemented a validate step to make seed length must be exactly 7 bits. See the code[1]

3. Write the LFSR Algorithm:

o Implement the LFSR using a loop that simulates the shift register operation.

Done in code. [1]

o XOR the bits specified by the feedback polynomial to determine the new input bit.

Done in code. [1]

o Shift the register to the right, discarding the last bit and inserting the new input bit at the front.

Done in code. [1]

2.2 Analyzing the LFSR Output:

1. Periodicity:

o Run the LFSR and record the generated sequence.

```
Full output sequence: 0000001001011101111010011010101101100001111100100011001110000101
```

To see the full output [2].

o Determine the period of the sequence (i.e., how many bits it generates before repeating).

It generates 63 clk. See the output [2], for more information.

```
Clk    S6 S5 S4 S3 S2 S1 S0    Output
-------------------------------------
0      1  0  0  0  0  0  0
١      0  1  0  0  0  0  0     ٠
٢      0  0  1  0  0  0  0     ٠
٣      1  0  0  1  0  0  0     ٠
٤      0  1  0  0  1  0  0     ٠
٥      1  0  1  0  0  1  0     ٠
٦      1  1  0  1  0  0  1     ٠
٧      1  1  1  0  1  0  0     ١
٨      0  1  1  1  0  1  0     ٠
٩      1  0  1  1  1  0  1     ٠
١٠     1  1  0  1  1  1  0     ١
١١     1  1  1  0  1  1  1     ٠
١٢     1  1  1  1  0  1  1     ١
١٣     0  1  1  1  1  0  1     ١
١٤     1  0  1  1  1  1  0     ١
١٥     0  1  0  1  1  1  1     ٠
١٦     0  0  1  0  1  1  1     ١
١٧     1  0  0  1  0  1  1     ١
١٨     1  1  0  0  1  0  1     ١
١٩     0  1  1  0  0  1  0     ١
٢٠     1  0  1  1  0  0  1     ٠
٢١     0  1  0  1  1  0  0     ١
٢٢     1  0  1  0  1  1  0     ٠
٢٣     0  1  0  1  0  1  1     ٠
٢٤     1  0  1  0  1  0  1     ١
٢٥     1  1  0  1  0  1  0     ١
٢٦     0  1  1  0  1  0  1     ٠
٢٧     1  0  1  1  0  1  0     ١
٢٨     1  1  0  1  1  0  1     ٠
٢٩     0  1  1  0  1  1  0     ١
٣٠     0  0  1  1  0  1  1     ٠
٣١     0  0  0  1  1  0  1     ١
٣٢     0  0  0  0  1  1  0     ١
٣٣     1  0  0  0  0  1  1     ٠
٣٤     1  1  0  0  0  0  1     ١
٣٥     1  1  1  0  0  0  0     ١
٣٦     1  1  1  1  0  0  0     ٠
٣٧     1  1  1  1  1  0  0     ٠
٣٨     0  1  1  1  1  1  0     ٠
٣٩     0  0  1  1  1  1  1     ٠
٤٠     1  0  0  1  1  1  1     ١
٤١     0  1  0  0  1  1  1     ١
٤٢     0  0  1  0  0  1  1     ١
٤٣     0  0  0  1  0  0  1     ١
٤٤     1  0  0  0  1  0  0     ١
٤٥     1  1  0  0  0  1  0     ٠
٤٦     0  1  1  0  0  0  1     ٠
٤٧     0  0  1  1  0  0  0     ١
٤٨     1  0  0  1  1  0  0     ٠
٤٩     1  1  0  0  1  1  0     ٠
٥٠     1  1  1  0  0  1  1     ٠
٥١     0  1  1  1  0  0  1     ١
٥٢     0  0  1  1  1  0  0     ١
٥٣     0  0  0  1  1  1  0     ٠
٥٤     1  0  0  0  1  1  1     ٠
٥٥     0  1  0  0  0  1  1     ١
٥٦     1  0  1  0  0  0  1     ١
٥٧     0  1  0  1  0  0  0     ١
٥٨     0  0  1  0  1  0  0     ٠
٥٩     0  0  0  1  0  1  0     ٠
٦٠     0  0  0  0  1  0  1     ٠
٦١     0  0  0  0  0  1  0     ١
٦٢     0  0  0  0  0  0  1     ٠
٦٣     1  0  0  0  0  0  0     ١
**** Period: 63 ****
```

o Verify if the sequence achieves the maximum possible period, which is 2^n−1 for an n-bit LFSR.

The sequence does not achieves the maximum possible period. As given output you can see :

```
**** ANALYSIS RESULTS ****
Period: 63
Max possible: 127
Output length: 63
Maximum period not achieved ❌
Reason: Polynomial is reducible and not primitive
```

Explain why this LFSR does not achieves the expected maximum possible period.

Because the polynomial is REDUCIBLE, with the mathematical proof:

p(1) = 1 + 1 + 1 + 1 = 0 (in GF(2)) I wrote it above. Therefore, p(x) is divisible by (x + 1) and we know that Reducible polynomials cannot be primitive. So, Only primitive polynomials can achieve max period 2^n - 1

3. *Randomness:*

o Assess the randomness of the generated sequence. You can perform simple statistical tests, such as checking the balance between 1s and 0s, or look for patterns in the sequence, over a sliding window of size 2n over a sequence of bits,

Here are the methods that assessed the randomness:

```java
/*
 *         Randomness assessment : Check the balance
 */
private static void assessRandomness(String sequence) {
    System.out.println(x: "\n**** Simple Randomness check ****");

    int ones = 0;  // to count 1s
    for (char bit : sequence.toCharArray()) {
        if (bit == '1') ones++;  // if the bit equals 1 , plass the ones
    }
    int zeros = sequence.length() - ones; // to count 0s


    // to calculate percentage :
    double onesPercent = (ones * 100.0) / sequence.length();
    double zerosPercent = (zeros * 100.0) / sequence.length();

    //       for printing
    System.out.println(x: "BALANCE TEST:");
    System.out.println("   • 1s: " + ones + " (" + String.format(format: "%.1f", args: onesPercent) + "%)");
    System.out.println("   • 0s: " + zeros + " (" + String.format(format: "%.1f", args: zerosPercent) + "%)");


    //  if ratio is between 40-60%, it's good
    if (ones >= sequence.length()*0.4 && ones <= sequence.length()*0.6) {  // between 40-60%
        System.out.println(x: "   ✅ Good balance (close to 50/50)");
    } else {
        System.out.println(x: "   ❌ Poor balance");
    }
}
```

```java
    /* PATTERN TEST with sliding window of size 2n (14 bits)
     *   Add this method right after assessRandomness */
    private static void performPatternTest(String sequence) {
        int n = LENGTH; // n = 7
        int windowSize = 2 * n; // 2n = 14 bits

        System.out.println("\nPATTERN TEST (sliding window: " + windowSize + " bits = 2 × " + n + "):");

        // Check if sequence is long enough
        if (sequence.length() < windowSize) {
            System.out.println("   • Sequence too short for " + windowSize + "-bit windows");
            return;
        }

        Set<String> uniquePatterns = new HashSet<>();
        int totalWindows = 0;

        // Slide the 14-bit window through the sequence
        for (int start = 0; start <= sequence.length() - windowSize; start++) {
            String window = sequence.substring(beginIndex: start, start + windowSize);
            uniquePatterns.add(e: window);
            totalWindows++;
        }
        System.out.println("   • Total sliding windows: " + totalWindows);
        System.out.println("   • Unique " + windowSize + "-bit patterns: " + uniquePatterns.size());

        // Simple assessment
        double diversity = (uniquePatterns.size() * 100.0) / totalWindows;
        if (diversity > 50) {
            System.out.println("   ✅ Good pattern diversity (" + String.format(format: "%.1f", args: diversity) + "%)");
        } else {
            System.out.println("   ❌ Limited pattern diversity (" + String.format(format: "%.1f", args: diversity) + "%)");
        }
        // Show a few sample patterns
        System.out.println(x: "   • Sample patterns:");
        int count = 0;
        for (String pattern : uniquePatterns) {
            if (count < 2) { // Show only 2 samples
                System.out.println("     - " + pattern);
                count++;
            } else {
                break;
            }
        }
    }
```

Here is the output:

```
**** Simple Randomness check for LFSR****
BALANCE TEST:
   • 1s: 62 (٤٨,٤%)
   • 0s: 66 (٥١,٦%)
   ✅ Good balance (close to 50/50)

PATTERN TEST (sliding window: 14 bits = 2 × 7):
   • Total sliding windows: 115
   • Unique 14-bit patterns: 63
   ✅ Good pattern diversity (٥٤,٨%)
   • Sample patterns:
     - 10011010101101
     - 10100000010010
```

3.2 Analysis Results:

o the design of the LFSR

The LFSR is implemented as a 7-bit shift register (S6-S0) that shifts right on each clock cycle.

The feedback is calculated by XORing specific tap positions (S4, S2, S0) as specified by the polynomial.

The new feedback bit is inserted at the leftmost position (S6) after each shift.

o the parameters used in your LFSR (length, feedback polynomial, seed value).

```
// ---------  Parameters  --------
private static final int LENGTH = 7; //length of the polynomial , LFSR have 7FF(s6-s0
private static final String POLYNOMIAL = "x^7 + x^4 + x^2 + 1"; //our polynomial
private static final String FeedbackPOLYNOMIAL = "x^7 = x^4 + x^2 + 1"; //our polynomial
private static final int[] TAPS = {2, 4, 6}; // S4, S2, S0 positions of XOR
private static final int[] INITIAL_SEED = {1, 0, 0, 0, 0, 0, 0}; //initial seed
```

o The sequence generated by your LFSR.

```
Reason: Polynomial is reducible and not primitive
Full output sequence: 0000001001011101111010011010101101100001111100100011001110001010000001001011101111010011010101101100001111100100011001110001010
```

The LFSR generated a sequence of:
[2] to see the full output of the LFSR.

o An analysis of the sequence's periodicity.

The actual period measured in the clock cycles table that shown in the output. [2]. The LFSR did not achieve maximum period because it is reducible (not primitive).

o An assessment of the sequence’s randomness.

I have provided a balance test, that count number of ones and zeros from the polynomial sequence. Also, a method to look for patterns in the sequence, over a sliding window of size 2n over a sequence of bits.

The result as shown:

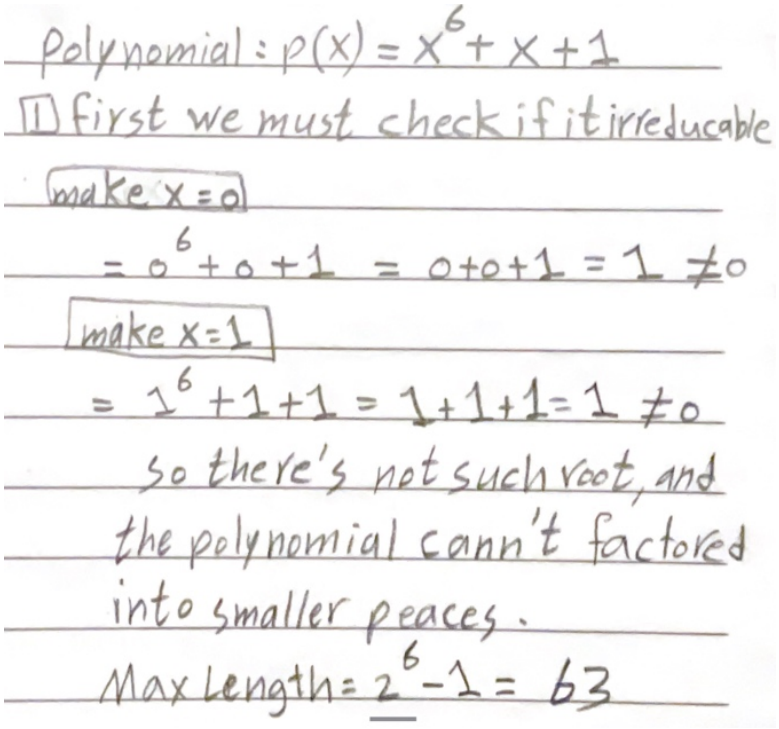
```
**** Simple Randomness check for LFSR****
BALANCE TEST:
   • 1s: 62 (٤٨.٤%)
   • 0s: 66 (٥١.٦%)
   ✅ Good balance (close to 50/50)

PATTERN TEST (sliding window: 14 bits = 2 × 7):
   • Total sliding windows: 115
   • Unique 14-bit patterns: 63
   ✅ Good pattern diversity (٥٤.٨%)
   • Sample patterns:
     - 10011010101101
     - 10100000010010
```

o Explain the effect of the seed value.

It is effect on the starting point in the sequence cycle and different non-zero seeds produce different sequences but with the same maximum period (in case the polynomial was irreducible) . But we must know that all-zero seed (0000000) would cause the LFSR to stall (invalid state).

o Give an example of a sixth order polynomial that does achieve the maximum expected period and explain why.



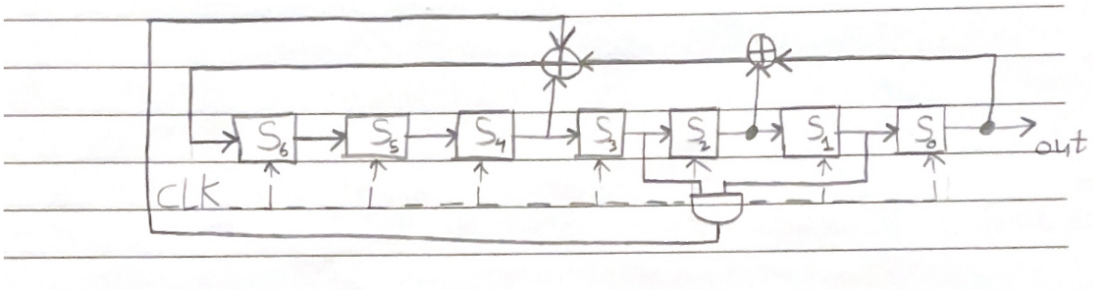
Non-linear Feedback/Feedforward Shift Register (NFSR) Design and Implementation

o Modify the LFSR in part 1-3 (keeping the register length) to design and implement a non- linear feedback/feedforward shift register

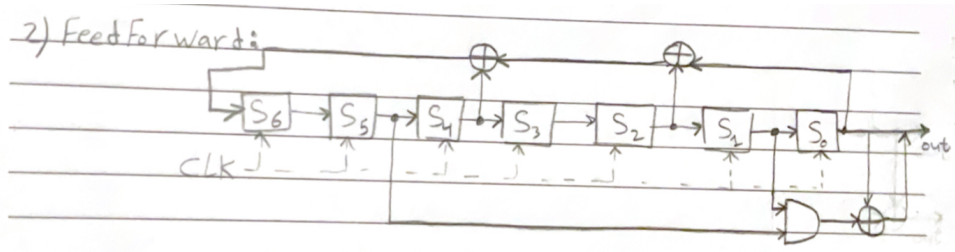
1. feedback loop only
2. feedforward loop only
3. both feedback and feedforward loops

o Give the block diagram of the non-linear feedback/feedforward shift register for each case

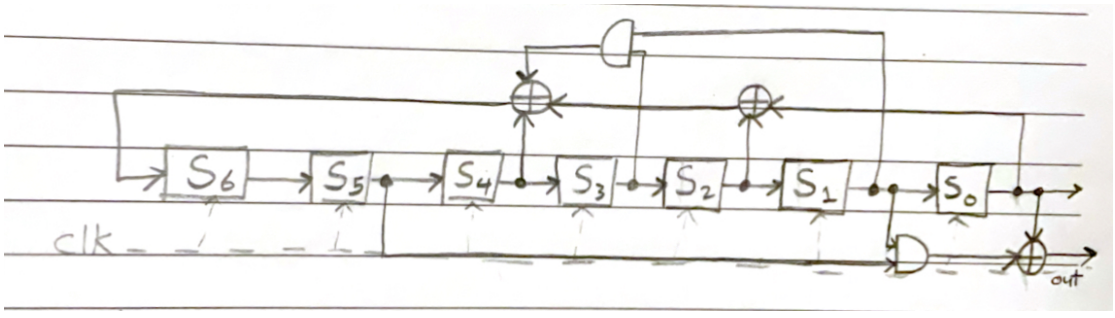
1) Feedback:



2) Feedforward:



3) both feedback and feedforward:



o Give the polynomials for each of the three cases

1) Feedback:

Feedback Function: $f(x) = x^4 + x^2 + x^0 + (x^1 \cdot x^3)$

Output Function: $O = S_0$

2) Feedforward:

Feedback Polynomial: $p(x) = x^7 + x^4 + x^2 + 1$

Output Function: $O = S_0 \oplus (S_1 \wedge S_5)$

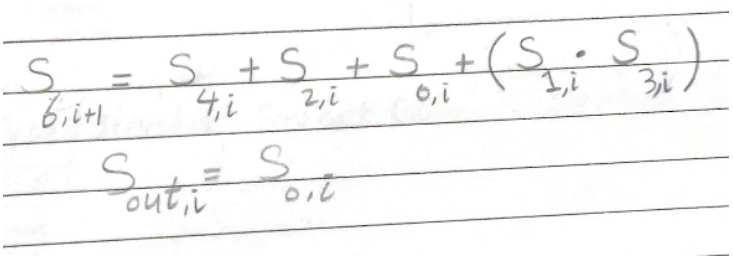
3) both feedback and feedforward:

Feedback Function: $f(x) = x^4 + x^2 + x^0 + (x^1 \cdot x^3)$

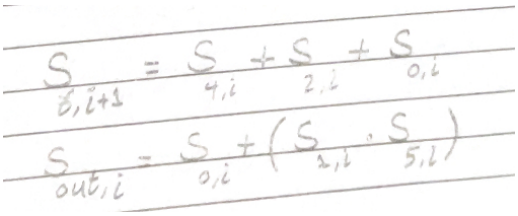
Output Function: $O = S_0 \oplus (S_1 \wedge S_5)$

o Give the complete recursion equations for each of the three cases

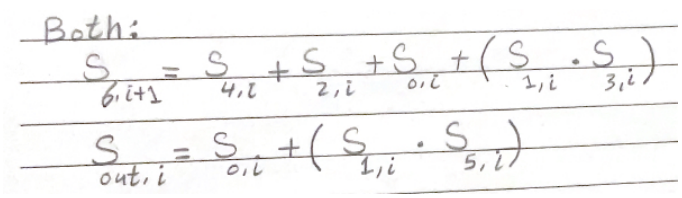
1) Feedback:



2) Feedforward:



3) both feedback and feedforward:



o Explain the reasons for the design of each case

1) Feedback:

The main goal is to improve cryptographic security. Breaking the linearity inherent in Linear Feedback Shift Registers (LFSRs) is the main reason for choosing registers S_3 and S_1 as inputs to the non-linear feedback function (represented by an AND gate). Simple LFSRs can be attacked using cryptanalytic techniques. The integration of a non-linear element using strategically chosen, non-adjacent taps introduce algebraic complexity, rendering such linear analysis ineffective. (S_3, S_1) is chosen because they are non-adjacent, guaranteeing the longest possible period for the sequence.

2) Feedforward:

The main purpose of a feedforward path is to make the output sequence more linearly complex without affecting period length. The feedforward path produces the final output bit, by combining the internal states in a non-linear way, while the feedback path establishes the period and state transition. Taps S_5 and S_1 are chosen to prevent the output from being a direct tap from a single register, the relationship between the internal state and the output sequence becomes much more complex and harder to decipher.

3) both feedback and feedforward:

A cryptographically robust pseudo-random sequence generator is produced by combining feedforward and feedback paths. While the feedforward eliminates linear characteristics in the output, offering resilience against attacks, the feedback offers a long, unpredictable period for the internal state machine. To achieve high linear complexity and security, the feedforward path's ideal selection of non-adjacent taps, such as S_5 and S_1, is essential.

o Implement the design of each case

1) Feedback:

You can see the full code [3].

```
// Main LFSR loop, run to 128 cycles (more than max possible)
for (int clock = 1; clock <= 128; clock++) {

    int andResult = register[AND_TAPS[0]] & register[AND_TAPS[1]]; // to calculate the AND

    // register[TAPS[0]] = S4, register[TAPS[1]] = S2, register[TAPS[2]] = S0
    int feedback = register[TAPS[0]] ^ register[TAPS[1]] ^ register[TAPS[2]] ^ andResult; // XOR with AND

    // The output bit is the rightmost bit (S0) before we shift
    int outputBit = register[6];
    output.append(i: outputBit);// Add to our output sequence

    // // SHIFT operation:
    int[] newRegister = new int[LENGTH];// new empty register

    newRegister[0] = feedback;// Put feedback bit(result of XOR) in leftmost position (S6)


    // Copy all bits from old register, shifting them right by one position
    // register[0] goes to newRegister[1], register[1] to newRegister[2], and so on..
    System.arraycopy(src:register, srcPos: 0, dest:newRegister, destPos:1, LENGTH - 1);

    // Update our register to the new state
    register = newRegister;
```

2) Feedforward:

You can see the full code [5].

```
// Main LFSR loop, run to 128 cycles (more than max possible)
for (int clock = 1; clock <= 128; clock++) {

    int andResult = register[AND_TAPS[0]] & register[AND_TAPS[1]]; // AND between positions 1 and 5 ←

    // register[TAPS[0]] = S4, register[TAPS[1]] = S2, register[TAPS[2]] = S0
    int feedback = register[2] ^ register[4] ^ register[6]; // XOR with AND ←

    // The output bit is the rightmost bit (S0) before we shift
    int outputBit = register[6] ^ andResult; // XOR with S0 ←
    output.append(i: outputBit);// Add to our output sequence

    // // SHIFT operation:
    int[] newRegister = new int[LENGTH];// new empty register

    newRegister[0] = feedback;// Put feedback bit(result of XOR) in leftmost position (S6)


    // Copy all bits from old register, shifting them right by one position
    // register[0] goes to newRegister[1], register[1] to newRegister[2], and so on..
    System.arraycopy(src:register, srcPos: 0, dest:newRegister, destPos:1, LENGTH - 1);

    // Update our register to the new state
    register = newRegister;
```

3) both feedback and feedforward:

You can see the full code [7].

```
// Main LFSR loop, run to 128 cycles (more than max possible)
for (int clock = 1; clock <= 128; clock++) {

    int andFeedback = register[AND_TAPS_FEEDBACK[0]] & register[AND_TAPS_FEEDBACK[1]]; // AND between positions 3 and 5

    int feedback = register[2] ^ register[4] ^ register[6] ^ andFeedback;


    int andOutput = register[AND_TAPS_OUTPUT[0]] & register[AND_TAPS_OUTPUT[1]]; // AND between positions 1 and 5

    int outputBit = register[6] ^ andOutput;
    output.append(i: outputBit);// Add to our output sequence

    // // SHIFT operation:
    int[] newRegister = new int[LENGTH];// new empty register

    newRegister[0] = feedback;// Put feedback bit(result of XOR) in leftmost position (S6)


    // Copy all bits from old register, shifting them right by one position
    // register[0] goes to newRegister[1], register[1] to newRegister[2], and so on..
    System.arraycopy(src:register, srcPos: 0, dest:newRegister, destPos:1, LENGTH - 1);

    // Update our register to the new state
    register = newRegister;
```

o Find the maximum period of each case

We know the maximum period of the polynomial is 2^m-1 =127, but because the polynomial is reducible, the length will be different.

1. feedback loop only

   The maximum period is 98, you can see it from the output of the code [4].

2. feedforward loop only

   The maximum period is 63, you can see it from the output of the code [6].

3. both feedback and feedforward loops

   The maximum period is 98, you can see it from the output of the code [8].

o Compare the randomness of the three cases as well as with the LFSR in part 1-3

Balance Test Results:

All four implementations show **excellent balance** between number of 1's & 0's:

- LFSR: 49.4% 1s, 51.6% 0s **Good balance**
- Feedback NLFSR: 46.9% 1s, 53.1% 0s ✅ **Good balance**
- Feedforward NLFSR: 50% 1s, 50% 0s ✅ **Excellent balance**
- Both NLFSR: 48.4% 1s, 51.6% 0s ✅ **Good balance**

Pattern Diversity Test (14-bit sliding window)**:**

- LFSR: patterns (54.8% diversity) Good
- Feedback NLFSR**:** patterns (85.2% diversity) Good
- Feedforward NLFSR**:** 50/50 patterns (54.8% diversity) **Perfect**
- Both NLFSR**:** 85/85 patterns (84.2% diversity) **Perfect**

All achieve perfect pattern diversity. But feedback is the best

- Discuss the comparisons between the different designs and explain your findings

Explain finding:

Both NLFSR is the best because Feedback NLFSR effects on period and had a best overall performance the other hand, Feedforward NLFSR should not effect on period, but effects on output sequence and that improves the output complexity. So, Both NLFSR combined these features.

Type	Structure	Feedback Function	Output Function	Max Period	Actual Period	Num of 1's	Num of 0's	Ratio	Quality
LFSR	Linear	S4⊕S2⊕S0	S0	127bits	63bits	31	32	49.2%:50.8%	Excellent
Feedback NLFSR	Nonlinear feedback with AND gate in feedback path	S4⊕S2⊕S0(S1∧S3)	S0	127bits	98bits	49	49	50% : 50%	Perfect
Feedforward NLFSR	Nonlinear output with AND gate in output path	S4⊕S2⊕S0	S0⊕(S1∧S5)	127 bits	63bits	32	31	50.8%:49.2%	Excellent
Both NLFSR	Combined nonlinear with AND gates in both paths	S4⊕S2⊕S0⊕(S1∧S3)	$\oplus\ (S1 \wedge S5)$S0	127 bits	98bits	50	48	51% : 49%	Excellent

Appendix:

[1] Screen shot of the LFSR code:

```java
/*
Joori Eihab 2310656 section CY2
 */
package lfsr;
import java.util.*;

public class LFSR {

    // ---------  Parameters  --------
    private static final int LENGTH = 7; //length of the polynomial , LFSR have 7FF(s6-s0
    private static final String POLYNOMIAL = "x^7 + x^4 + x^2 + 1"; //our polynomial
    private static final String FeedbackPOLYNOMIAL = "x^7 = x^4 + x^2 + 1"; //our polynomial
    private static final int[] TAPS = {2, 4, 6}; // S4, S2, S0 positions of XOR
    private static final int[] INITIAL_SEED = {1, 0, 0, 0, 0, 0, 0}; //initial seed

    public static void main(String[] args) throws InvalidSeedException, InvalidTapException {

        // Step 1: Check if our parameters are valid before running with try-catch
        checkValidParameter();

        //Separate logic from I/O , Step 2: Run the LFSR algorithm and get results
        LFSRResult result = runLFSR();

        // Step 3: Show the results to the user
        displayResults(result);
    }

    /**
     *            Implements LFSR algorithm
     *   This is the core function that runs the Linear Feedback Shift Register:
     */
    private static LFSRResult runLFSR()  {

        // create a copy of the initial seed so we don't modify the original one
        int[] register = INITIAL_SEED.clone();

        //to store the output sequence (bits that come out)
        StringBuilder output = new StringBuilder();

        int period = 0;// How many steps until repetition
        int maxPeriod = (int)Math.pow(a: 2, b: LENGTH) - 1; //2^7 - 1 = 127

        // to display header informations
        System.out.println("LFSR Analysis Polynomial: " + POLYNOMIAL);
        System.out.println("Our feedback Polynomial: " + FeedbackPOLYNOMIAL);
```

```
            System.out.println("Max possible period: " + maxPeriod);
            System.out.println("\nClk    S6 S5 S4 S3 S2 S1 S0    Output");
            System.out.println("-----------------------------------");

            //// Show initial state (step 0) Clk 0
            System.out.printf("0      %s    %n", arrayToString(register));

            // Main LFSR loop, run to 128 cycles (more than max possible)
            for (int clock = 1; clock <= 128; clock++) {

                // register[TAPS[0]] = S4, register[TAPS[1]] = S2, register[TAPS[2]] = S0
                int feedback = register[TAPS[0]] ^ register[TAPS[1]] ^ register[TAPS[2]];//XOR the tapped bits together

                // The output bit is the rightmost bit (S0) before we shift
                int outputBit = register[6];
                output.append(outputBit);// Add to our output sequence

                // // SHIFT operation:
                int[] newRegister = new int[LENGTH];// new empty register

                newRegister[0] = feedback;// Put feedback bit(result of XOR) in leftmost position (S6)


                // Copy all bits from old register, shifting them right by one position
                // register[0] goes to newRegister[1], register[1] to newRegister[2], and so on..
                System.arraycopy(register, 0, newRegister, 1, LENGTH - 1);

                // Update our register to the new state
                register = newRegister;

                // Display the current state
                System.out.printf("%-2d     %s    %d%n", clock, arrayToString(register), outputBit);

                // Period detection ,check if we've returned to the initial state
                if (arrayEquals(register, INITIAL_SEED)) {
                    period = clock;// found the period length
                    System.out.println("**** Period: " + period + " ****");

                }

            }
            // Return all our results packaged together
            return new LFSRResult(period, output.toString(), maxPeriod);

    }
```

```
    /**
     * This function checks if our configurations are valid before running
     */
    private static void checkValidParameter() throws InvalidSeedException, InvalidTapException {
        try {
            // Validate seed length must be exactly 7 bits
            if (INITIAL_SEED.length != LENGTH) {
                throw new InvalidSeedException("Seed length must be " + LENGTH);
            }

            // Validate seed is not all zeros , because all-zero state would cause problems
            boolean allZero = true;
            for (int bit : INITIAL_SEED) {
                if (bit != 0) {
                    allZero = false; // Found a 1, so not all zeros
                    break;
                }
            }
            if (allZero) {
                throw new InvalidSeedException(message:"Seed cannot be all zeros");
            }

            // Validate tap array not empty , need at least one tap
            if (TAPS.length == 0) {
                throw new InvalidTapException(message:"Taps array cannot be empty");
            }

            System.out.println(" All parameters validated successfully ✅");

        }
        catch (InvalidSeedException | InvalidTapException e) {
            throw e; }// Re-throw specific exceptions to main method
         catch (Exception e) {
            throw new InvalidSeedException("Unexpected error during parameter validation: " + e.getMessage());
        }
    }
    /**
     *            Display results with analysis
     *            Shows the final results to the user
     */
    private static void displayResults(LFSRResult result) {

            System.out.println("\n**** ANALYSIS RESULTS ****");
            System.out.println("Period: " + result.period);
            System.out.println("Max possible: " + result.maxPeriod);
```

```
                System.out.println("Max possible: " + result.maxPeriod);
                System.out.println("Output length: " + result.output.length());


                //Check if we achieved maximum period (in our case no)
                if (result.period == result.maxPeriod) {
                    System.out.println(x: "Maximum period achieved ✅");
                }
                else {
                    System.out.println(x: "Maximum period not achieved ❌");
                    System.out.println(x: "Reason: Polynomial is reducible and not primitive");
                }

                // to show the full sequence of bits
                System.out.println("Full output sequence: " + result.output);

                assessRandomness(sequence: result.output); // method to asses the rendomness
            }

        /*
         *         Randomness assessment : Check the balance
         */
        private static void assessRandomness(String sequence) {
            System.out.println(x: "\n**** Simple Randomness check for LFSR****");

            int ones = 0;  // to count 1s
            for (char bit : sequence.toCharArray()) {
                if (bit == '1') ones++;  // if the bit equals 1 , plass the ones
            }
            int zeros = sequence.length() - ones; // to count 0s


            // to calculate percentage :
            double onesPercent = (ones * 100.0) / sequence.length();
            double zerosPercent = (zeros * 100.0) / sequence.length();

            //       for printing
            System.out.println(x: "BALANCE TEST:");
            System.out.println("   • 1s: " + ones + " (" + String.format(format: "%.1f", args: onesPercent) + "%)");
            System.out.println("   • 0s: " + zeros + " (" + String.format(format: "%.1f", args: zerosPercent) + "%)");


            //  if ratio is between 40-60%, it's good
            if (ones >= sequence.length()*0.4 && ones <= sequence.length()*0.6) {  // between 40-60%
                System.out.println(x: "   ✅ Good balance (close to 50/50)");
```

```
            } else {
                System.out.println(x: "   ❌ Poor balance");
            }

            // to call the pattern test:
            performPatternTest(sequence);
        }
        /**
         * PATTERN TEST with sliding window of size 2n (14 bits)
         * Add this method right after assessRandomness
         */
        private static void performPatternTest(String sequence) {
            int n = LENGTH; // n = 7
            int windowSize = 2 * n; // 2n = 14 bits

            System.out.println("\nPATTERN TEST (sliding window: " + windowSize + " bits = 2 × " + n + "):");

            // Check if sequence is long enough
            if (sequence.length() < windowSize) {
                System.out.println("   • Sequence too short for " + windowSize + "-bit windows");
                return;
            }

            Set<String> uniquePatterns = new HashSet<>();
            int totalWindows = 0;

            // Slide the 14-bit window through the sequence
            for (int start = 0; start <= sequence.length() - windowSize; start++) {
                String window = sequence.substring(beginIndex: start, start + windowSize);
                uniquePatterns.add(e: window);
                totalWindows++;
            }

            System.out.println("   • Total sliding windows: " + totalWindows);
            System.out.println("   • Unique " + windowSize + "-bit patterns: " + uniquePatterns.size());

            // Simple assessment
            double diversity = (uniquePatterns.size() * 100.0) / totalWindows;
            if (diversity > 50) {
                System.out.println("   ✅ Good pattern diversity (" + String.format(format: "%.1f", args: diversity) + "%)");
            } else {
                System.out.println("   ❌ Limited pattern diversity (" + String.format(format: "%.1f", args: diversity) + "%)");
            }

```

```
        // Show a few sample patterns
        System.out.println(x: "   • Sample patterns:");
        int count = 0;
        for (String pattern : uniquePatterns) {
            if (count < 2) { // Show only 2 samples
                System.out.println("     - " + pattern);
                count++;
            } else {
                break;
            }
        }
    }
        // -------- In this section I uesed methods to help me --------

        /*Convert an array of bits to a string for displaying
         * for example: [1,0,0,0,0,0,0] = "1 0 0 0 0 0 0" */
        private static String arrayToString(int[] arr) {
            try {
                StringBuilder sb = new StringBuilder();
                for (int value : arr) {
                    sb.append(i: value).append(str:" ");
                }
                return sb.toString().trim(); //trim to remove spaces
            } catch (Exception e) {
                return "Error converting array to string";
            }
        }

        /*
         * Compare two arrays for equality
         * it returns true if both arrays have same length and same values in same index*/
        private static boolean arrayEquals(int[] a, int[] b) {
            try {
                if (a.length != b.length) return false;
                for (int i = 0; i < a.length; i++) {
                    if (a[i] != b[i]) return false;
                }
                return true;
            } catch (Exception e) {
                return false;
            }
        }
```

```
        // -------- CUSTOM EXCEPTION CLASSES --------
        // These help us to handle different types of errors specifically

        static class InvalidSeedException extends Exception {
            public InvalidSeedException(String message) { super(message); }
        }

        static class InvalidTapException extends Exception {
            public InvalidTapException(String message) { super(message); }
        }

        /**
         *                    Result container class
         *           A simple class to package all our results together
         */
        static class LFSRResult {
            int period;   // How many steps until repetition
            String output; // sequence of bits generated
            int maxPeriod; // maximum period from (2^n - 1)

            LFSRResult(int period, String output, int maxPeriod) {
                this.period = period;
                this.output = output;
                this.maxPeriod = maxPeriod;
            }
    }

    }
```

```
            System.out.println(x: "All validation tests passed ✅");

        } catch (TestFailureException e) {
            System.out.println("Validation test failed: " + e.getMessage());
        } catch (Exception e) {
            System.out.println("Unexpected error during validation: " + e.getMessage());
        }
    }

    /**
     *      Test deterministic reproduction
     */
    private static void testDeterminism() throws TestFailureException {
        try {
            // Create two identical registers
            int[] reg1 = INITIAL_SEED.clone();
            int[] reg2 = INITIAL_SEED.clone();
            StringBuilder out1 = new StringBuilder();
            StringBuilder out2 = new StringBuilder();

            // Run both for 10 steps
            for (int i = 0; i < 10; i++) {
                int fb1 = reg1[TAPS[0]] ^ reg1[TAPS[1]] ^ reg1[TAPS[2]]; // xor
                out1.append(reg1[6]); // Store output
                reg1 = shiftRight(register: reg1, feedback: fb1); // Shifting

                // run the second LFSR
                int fb2 = reg2[TAPS[0]] ^ reg2[TAPS[1]] ^ reg2[TAPS[2]]; //xor
                out2.append(reg2[6]); // Store output
                reg2 = shiftRight(register: reg2, feedback: fb2); // Shifting
            }

            //Compare outputs , they should be identical
            if (!out1.toString().equals(anObject: out2.toString())) {
                throw new TestFailureException(message:"LFSR is not deterministic");
            }
            System.out.println(x: "Deterministic reproduction test passed✅");

        } catch (Exception e) {
            throw new TestFailureException("Determinism test error: " + e.getMessage());
        }
    }

    /**
```

```
235
236     /**
237      *       Test with different seeds
238      *        Verify LFSR works with various starting states
239      */
240     private static void testDifferentSeeds() throws TestFailureException {
241         try {
242
243             // Test with different seeds
244             int[][] testSeeds = {
245                 {1, 0, 0, 0, 0, 0, 0, 0}, // Our main seed
246                 {0, 0, 0, 0, 0, 0, 0, 1}, // diff seed
247                 {1, 0, 1, 0, 1, 0, 1}  // diff seed
248             };
249
250             for (int i = 0; i < testSeeds.length; i++) {
251                 if (validateTestSeed(testSeeds[i])) {
252                     int period = testLFSR(testSeeds[i]);
253                     System.out.printf( format: " ✅Seed test %d: period = %d%n", i + 1, args: period);
254                 }
255             }
256
257         } catch (Exception e) {
258             throw new TestFailureException("Different seeds test error: " + e.getMessage());
259         }
260     }
261
262     /**
263      *        Test boundary conditions
264      *         Verify period is within reasonable bounds
265      */
266     private static void testBoundaryConditions() throws TestFailureException {
267         try {
268             // Test period is within bounds
269             int period = testLFSR( seed: INITIAL_SEED);
270             if (period <= 0 || period > 128) {
271                 throw new TestFailureException("Period " + period + " is outside valid range");
272             }
273             System.out.println( x: "Boundary conditions test passed ✅");
274
275         } catch (Exception e) {
276             throw new TestFailureException("Boundary conditions test error: " + e.getMessage());
277         }
278     }
279
```

```
278     }
279
280     /**
281      *           Test helper method with error handling
282      *            Runs LFSR on a given seed and returns the period
283      */
284     private static int testLFSR(int[] seed) throws TestFailureException {
285         try {
286             int[] register = seed.clone();
287             int period = 0;
288
289             for (int clock = 1; clock <= 128; clock++) {
290                 int feedback = register[TAPS[0]] ^ register[TAPS[1]] ^ register[TAPS[2]];
291
292                 int[] newRegister = new int[LENGTH];
293                 newRegister[0] = feedback;
294                 System.arraycopy(src:register, srcPos: 0, dest:newRegister, destPos:1, LENGTH - 1);
295                 register = newRegister;
296
297                 if (arrayEquals(a: register, b: seed)) {
298                     period = clock;
299                     break;
300                 }
301             }
302             return period;
303
304         } catch (Exception e) {
305             throw new TestFailureException("LFSR test execution error: " + e.getMessage());
306         }
307     }
308
309     /**
310      *            Display results with analysis
311      *            Shows the final results to the user
312      */
313     private static void displayResults(LFSRResult result) {
314         try {
315             System.out.println(x: "\n**** ANALYSIS RESULTS ****");
316             System.out.println("Period: " + result.period);
317             System.out.println("Max possible: " + result.maxPeriod);
318             System.out.println("Output length: " + result.output.length());
319
320
321             //Check if we achieved maximum period (in our case no)
322             if (result.period == result.maxPeriod) {
```

```
322             if (result.period == result.maxPeriod) {
323                 System.out.println(x: "Maximum period achieved ✅");
324             }
325             else {
326                 System.out.println(x: "Maximum period not achieved ❌");
327                 System.out.println(x: "Reason: Polynomial is reducible and not primitive");
328             }
329 
330             // to show the full sequence of bits
331             System.out.println("Full output sequence: " + result.output);
332 
333         } catch (Exception e) {
334             System.out.println("Error during results display: " + e.getMessage());
335         }
336     }
337 
338     // -------- In this section I uesed methods to help me --------
339 
340     /*Convert an array of bits to a string for displaying
341      * for example: [1,0,0,0,0,0,0] = "1 0 0 0 0 0 0" */
342     private static String arrayToString(int[] arr) {
343         try {
344             StringBuilder sb = new StringBuilder();
345             for (int value : arr) {
346                 sb.append(i: value).append(str:" ");
347             }
348             return sb.toString().trim(); //trim to remove spaces
349         } catch (Exception e) {
350             return "Error converting array to string";
351         }
352     }
353 
354     /*
355      * Compare two arrays for equality
356      * it returns true if both arrays have same length and same values in same index*/
357     private static boolean arrayEquals(int[] a, int[] b) {
358         try {
359             if (a.length != b.length) return false;
360             for (int i = 0; i < a.length; i++) {
361                 if (a[i] != b[i]) return false;
362             }
363             return true;
364         } catch (Exception e) {
365             return false;
366         }
```

```
366             }
367         }
368
369
370         //Perform the shift-right operation on the register
371         private static int[] shiftRight(int[] register, int feedback) throws InvalidLFSRException {
372             try {
373                 int[] newRegister = new int[register.length];
374                 newRegister[0] = feedback;
375                 System.arraycopy(src:register, srcPos: 0, dest:newRegister, destPos:1, register.length - 1);
376                 return newRegister;
377             } catch (Exception e) {
378                 throw new InvalidLFSRException("Error during shift operation: " + e.getMessage());
379             }
380         }
381
382
383         //Quick validation for test seeds
384         private static boolean validateTestSeed(int[] seed) {
385             try {
386                 return seed.length == LENGTH && !arrayEquals(a: seed, new int[LENGTH]);
387             } catch (Exception e) {
388                 return false;
389             }
390         }
391
392         // -------- CUSTOM EXCEPTION CLASSES --------
393         // These help us to handle different types of errors specifically
394
395
396         static class InvalidSeedException extends Exception {
397             public InvalidSeedException(String message) { super(message); }
398         }
399
400         static class InvalidTapException extends Exception {
401             public InvalidTapException(String message) { super(message); }
402         }
403
404         static class InvalidLFSRException extends Exception {
405             public InvalidLFSRException(String message) { super(message); }
406         }
407
408         static class TestFailureException extends Exception {
409             public TestFailureException(String message) { super(message); }
410         }
```

```
411
412         /**
413          *                    Result container class
414          *           A simple class to package all our results together
415          */
416         static class LFSRResult {
417             int period;   // How many steps until repetition
418             String output; // sequence of bits generated
419             int maxPeriod; // maximum period from (2^n - 1)
420
421             LFSRResult(int period, String output, int maxPeriod) {
422                 this.period = period;
423                 this.output = output;
424                 this.maxPeriod = maxPeriod;
425             }
426     }
427     }
428
```

[2] The output of the LFSR code:

✅ All parameters validated successfully

LFSR Analysis Polynomial: x^7 + x^4 + x^2 + 1

Our feedback Polynomial: x^7 = x^4 + x^2 + 1

Max possible period: 127

Clk S6 S5 S4 S3 S2 S1 S0 Output

---

٠ ٠ ٠ ٠ ٠ ٠ ١ ٠

٠ ٠ ٠ ٠ ٠ ٠ ١ ٠ ١

٠ ٠ ٠ ٠ ٠ ١ ٠ ٠ ٢

٠ ٠ ٠ ٠ ١ ٠ ٠ ١ ٣

٠ ٠ ٠ ١ ٠ ٠ ١ ٠ ٤

٠ ٠ ١ ٠ ٠ ١ ٠ ١ ٥

٠ ١ ٠ ٠ ١ ٠ ١ ١ ٦

١ ٠ ٠ ١ ٠ ١ ١ ١ ٧

٠ ٠ ١ ٠ ١ ١ ١ ٠ ٨

٠ ١ ٠ ١ ١ ١ ٠ ١ ٩

١ ٠ ١ ١ ١ ٠ ١ ١ ١٠

٠ ١ ١ ١ ٠ ١ ١ ١ ١١

١ ١ ١ ٠ ١ ١ ١ ١ ١٢

١ ١ ٠ ١ ١ ١ ١ ٠ ١٣

١ ٠ ١ ١ ١ ١ ٠ ١ ١٤

١ ٠ ١ ١ ١ ٠ ١ ٠ ١٥

٠ ١ ١ ١ ١ ٠ ١ ٠ ١٦

١ ١ ١ ١ ٠ ١ ٠ ٠ ١٧

١ ١ ١ ٠ ١ ٠ ٠ ١ ١٨

١ ١ ٠ ١ ٠ ٠ ١ ١ ١٩

١ ٠ ١ ٠ ٠ ١ ١ ٠ ٢٠

٢١	٠ ٠ ١ ١ ٠ ١ ٠	١
٢٢	٠ ١ ١ ٠ ١ ٠ ١	٠
٢٣	١ ١ ٠ ١ ٠ ١ ٠	٠
٢٤	١ ٠ ١ ٠ ١ ٠ ١	١
٢٥	٠ ١ ٠ ١ ٠ ١ ١	١
٢٦	١ ٠ ١ ٠ ١ ١ ٠	٠
٢٧	٠ ١ ٠ ١ ١ ٠ ١	١
٢٨	١ ٠ ١ ١ ٠ ١ ١	٠
٢٩	٠ ١ ١ ٠ ١ ١ ٠	١
٣٠	١ ١ ٠ ١ ١ ٠ ٠	٠
٣١	١ ٠ ١ ١ ٠ ٠ ٠	١
٣٢	٠ ١ ١ ٠ ٠ ٠ ٠	١
٣٣	١ ١ ٠ ٠ ٠ ٠ ١	٠
٣٤	١ ٠ ٠ ٠ ٠ ١ ١	١
٣٥	٠ ٠ ٠ ٠ ١ ١ ١	١
٣٦	٠ ٠ ٠ ١ ١ ١ ١	٠
٣٧	٠ ٠ ١ ١ ١ ١ ١	٠
٣٨	٠ ١ ١ ١ ١ ١ ٠	٠
٣٩	١ ١ ١ ١ ١ ٠ ٠	٠
٤٠	١ ١ ١ ١ ٠ ٠ ١	١
٤١	١ ١ ١ ٠ ٠ ١ ٠	١
٤٢	١ ١ ٠ ٠ ١ ٠ ٠	١
٤٣	١ ٠ ٠ ١ ٠ ٠ ٠	١
٤٤	٠ ٠ ١ ٠ ٠ ٠ ١	١
٤٥	٠ ١ ٠ ٠ ٠ ١ ١	٠
٤٦	١ ٠ ٠ ٠ ١ ١ ٠	٠
٤٧	٠ ٠ ٠ ١ ١ ٠ ٠	١
٤٨	٠ ٠ ١ ١ ٠ ٠ ١	٠
٤٩	٠ ١ ١ ٠ ٠ ١ ١	٠

٠ ١١٠٠١١١ ٥٠

١ ١٠٠١١١٠ ٥١

١ ٠٠١١١٠٠ ٥٢

٠ ٠١١١٠٠٠ ٥٣

٠ ١١١٠٠٠١ ٥٤

١ ١١٠٠٠١٠ ٥٥

١ ١٠٠٠١٠١ ٥٦

١ ٠٠٠١٠١٠ ٥٧

٠ ٠٠١٠١٠٠ ٥٨

٠ ٠١٠١٠٠٠ ٥٩

٠ ١٠١٠٠٠٠ ٦٠

١ ٠١٠٠٠٠٠ ٦١

٠ ١٠٠٠٠٠٠ ٦٢

١ ٠٠٠٠٠٠١ ٦٣

**** Period: 63 ****

٠ ٠٠٠٠٠١٠ ٦٤

٠ ٠٠٠٠١٠٠ ٦٥

٠ ٠٠٠١٠٠١ ٦٦

٠ ٠٠١٠٠١٠ ٦٧

٠ ٠١٠٠١٠١ ٦٨

٠ ١٠٠١٠١١ ٦٩

١ ٠٠١٠١١١ ٧٠

٠ ٠١٠١١١٠ ٧١

٠ ١٠١١١٠١ ٧٢

١ ٠١١١٠١١ ٧٣

٠ ١١١٠١١١ ٧٤

١ ١١٠١١١١ ٧٥

١ ١٠١١١١٠ ٧٦

١ ٠١١١١٠١ ٧٧

٧٨	١١١١٠١٠	٠
٧٩	١١١٠١٠٠	١
٨٠	١١٠١٠٠١	١
٨١	١٠١٠٠١١	١
٨٢	٠١٠٠١١٠	١
٨٣	١٠٠١١٠١	٠
٨٤	٠٠١١٠١٠	١
٨٥	٠١١٠١٠١	٠
٨٦	١١٠١٠١٠	٠
٨٧	١٠١٠١٠١	١
٨٨	٠١٠١٠١١	١
٨٩	١٠١٠١١٠	٠
٩٠	٠١٠١١٠١	١
٩١	١٠١١٠١١	٠
٩٢	٠١١٠١١٠	١
٩٣	١١٠١١٠٠	٠
٩٤	١٠١١٠٠٠	١
٩٥	٠١١٠٠٠٠	١
٩٦	١١٠٠٠٠١	٠
٩٧	١٠٠٠٠١١	١
٩٨	٠٠٠٠١١١	١
٩٩	٠٠٠١١١١	٠
١٠٠	٠٠١١١١١	٠
١٠١	٠١١١١١٠	٠
١٠٢	١١١١١٠٠	٠
١٠٣	١١١١٠٠١	١
١٠٤	١١١٠٠١٠	١
١٠٥	١١٠٠١٠٠	١
١٠٦	١٠٠١٠٠٠	١

١ ٠ ٠ ١ ٠ ٠ ٠ ١ ١٠٧

٠ ٠ ١ ٠ ٠ ٠ ١ ١ ١٠٨

٠ ١ ٠ ٠ ٠ ١ ١ ٠ ١٠٩

١ ٠ ٠ ٠ ١ ١ ٠ ٠ ١١٠

٠ ٠ ٠ ١ ١ ٠ ٠ ١ ١١١

٠ ٠ ١ ١ ٠ ٠ ١ ١ ١١٢

٠ ١ ١ ٠ ٠ ١ ١ ١ ١١٣

١ ١ ٠ ٠ ١ ١ ١ ٠ ١١٤

١ ٠ ٠ ١ ١ ١ ٠ ٠ ١١٥

٠ ٠ ١ ١ ١ ٠ ٠ ٠ ١١٦

٠ ١ ١ ١ ٠ ٠ ٠ ١ ١١٧

١ ١ ١ ٠ ٠ ٠ ١ ٠ ١١٨

١ ١ ٠ ٠ ٠ ١ ٠ ١ ١١٩

١ ٠ ٠ ٠ ١ ٠ ١ ٠ ١٢٠

٠ ٠ ٠ ١ ٠ ١ ٠ ٠ ١٢١

٠ ٠ ١ ٠ ١ ٠ ٠ ٠ ١٢٢

٠ ١ ٠ ١ ٠ ٠ ٠ ٠ ١٢٣

١ ٠ ١ ٠ ٠ ٠ ٠ ٠ ١٢٤

٠ ١ ٠ ٠ ٠ ٠ ٠ ٠ ١٢٥

١ ٠ ٠ ٠ ٠ ٠ ٠ ١ ١٢٦

**** Period: 126 ****

٠ ٠ ٠ ٠ ٠ ١ ٠ ٠ ١٢٧

٠ ٠ ٠ ٠ ١ ٠ ٠ ٠ ١٢٨

**** ANALYSIS RESULTS ****

Period: 126

Max possible: 127

Output length: 128

❌ Maximum period not achieved

Reason: Polynomial is reducible and not primitive

Full output sequence:
00000010010111011110100110101011011000011111001000110011100010100000010
01011101111010011010101101100001111100100011001110001010 0

****Simple Randomness check for LFSR ****

:BALANCE TEST

(%٤٨,٤) s: 62١ •

(%٥١,٦) s: 66٠ •

(Good balance (close to 50/50 ✅

:(PATTERN TEST (sliding window: 14 bits = 2 × 7

Total sliding windows: 115 •

Unique 14-bit patterns: 63 •

(%٥٤,٨) Good pattern diversity ✅

:Sample patterns •

١٠٠١١٠١٠١٠١١٠١ -

١٠١٠٠٠٠٠١٠٠١٠ -

(BUILD SUCCESSFUL (total time: 0 seconds

[3] Screen shot of the NLFSR Feedback:

```java
/*
Joori Eihab 2310656 section CY2
 */
package lfsr;

import java.util.HashSet;
import java.util.Set;

public class NLFSRFeedback {

    // ---------  Parameters  --------
    private static final int LENGTH = 7; //length of the polynomial , LFSR have 7FF(s6-s0
    private static final String POLYNOMIAL = "x^7 + x^4 + x^2 + 1"; //our polynomial
    private static final int[] TAPS = {2, 4, 6}; // S4, S2, S0 positions of XOR
    private static final int[] AND_TAPS = {3, 5}; // taps for AND operation ←
    private static final int[] INITIAL_SEED = {1, 0, 0, 0, 0, 0, 0}; //initial seed

    public static void main(String[] args) throws InvalidSeedException, InvalidTapException {

            // Step 1: Check if our parameters are valid before running with try-catch
            checkValidParameter();

            //Separate logic from I/O , Step 2: Run the LFSR algorithm and get results
            NLFSRBack_Result result = runNLFSR_Back();

            // Step 3: Show the results to the user
            displayResults(result);
    }

    /**
     *            Implements LFSR algorithm
     *   This is the core function that runs the Linear Feedback Shift Register:
     */
    private static NLFSRBack_Result runNLFSR_Back()  {

            // create a copy of the initial seed so we don't modify the original one
            int[] register = INITIAL_SEED.clone();

            //to store the output sequence (bits that come out)
            StringBuilder output = new StringBuilder();

            int period = 0;// How many steps until repetition
            int maxPeriod = (int)Math.pow(a: 2, b: LENGTH) - 1; //2^7 - 1 = 127

            // to display header informations
```

```
            // to display header informations
            System.out.println("LFSR Analysis Polynomial: " + POLYNOMIAL);
            System.out.println("Max possible period: " + maxPeriod);
            System.out.println( x: "\nClk    S6 S5 S4 S3 S2 S1 S0    Output");
            System.out.println( x: "-----------------------------------");

            //// Show initial state (step 0) Clk 0
            System.out.printf( format: "0      %s    %n", args: arrayToString( arr: register));

            // Main LFSR loop, run to 128 cycles (more than max possible)
            for (int clock = 1; clock <= 128; clock++) {

                int andResult = register[AND_TAPS[0]] & register[AND_TAPS[1]]; // to calculate the AND ←

                // register[TAPS[0]] = S4, register[TAPS[1]] = S2, register[TAPS[2]] = S0
                int feedback = register[TAPS[0]] ^ register[TAPS[1]] ^ register[TAPS[2]] ^ andResult; // XOR with AND ←

                // The output bit is the rightmost bit (S0) before we shift
                int outputBit = register[6];
                output.append( i: outputBit);// Add to our output sequence

                // // SHIFT operation:
                int[] newRegister = new int[LENGTH];// new empty register

                newRegister[0] = feedback;// Put feedback bit(result of XOR) in leftmost position (S6)


                // Copy all bits from old register, shifting them right by one position
                // register[0] goes to newRegister[1], register[1] to newRegister[2], and so on..
                System.arraycopy( src: register, srcPos: 0, dest: newRegister, destPos: 1, LENGTH - 1);

                // Update our register to the new state
                register = newRegister;

                // Display the current state
                System.out.printf( format: "%-2d     %s    %d%n", args: clock, args: arrayToString( arr: register), args: outputBit);

                // Period detection ,check if we've returned to the initial state
                if (arrayEquals( a: register, b: INITIAL_SEED)) {
                    period = clock;// found the period length
                    System.out.println("**** Period: " + period + " ****");
                }
            }
            // Return all our results packaged together
            return new NLFSRBack_Result(period, output: output.toString(), maxPeriod);
```

```
            // Return all our results packaged together
            return new NLFSRBack_Result(period, output: output.toString(), maxPeriod);
    }
    /**
     * This function checks if our configurations are valid before running
     */
    private static void checkValidParameter() throws InvalidSeedException, InvalidTapException {
        try {
            // Validate seed length must be exactly 7 bits
            if (INITIAL_SEED.length != LENGTH) {
                throw new InvalidSeedException("Seed length must be " + LENGTH);
            }

            // Validate seed is not all zeros , because all-zero state would cause problems
            boolean allZero = true;
            for (int bit : INITIAL_SEED) {
                if (bit != 0) {
                    allZero = false; // Found a 1, so not all zeros
                    break;
                }
            }
            if (allZero) {
                throw new InvalidSeedException(message:"Seed cannot be all zeros");
            }

            // Validate tap array not empty , need at least one tap
            if (TAPS.length == 0) {
                throw new InvalidTapException(message:"Taps array cannot be empty");
            }

            System.out.println( x: " All parameters validated successfully ✅");

        }
        catch (InvalidSeedException | InvalidTapException e) {
            throw e; }// Re-throw specific exceptions to main method
         catch (Exception e) {
            throw new InvalidSeedException("Unexpected error during parameter validation: " + e.getMessage());
        }
    }

    /**
     *            Display results with analysis
     *            Shows the final results to the user
     */
```

```java
    private static void displayResults(NLFSRBack_Result result) {

            System.out.println(x: "\n**** ANALYSIS RESULTS ****");
            System.out.println("Period: " + result.period);
            System.out.println("Max possible: " + result.maxPeriod);
            System.out.println("Output length: " + result.output.length());


            //Check if we achieved maximum period (in our case no)
            if (result.period == result.maxPeriod) {
                System.out.println(x: "Maximum period achieved ✅");
            }
            else {
                System.out.println(x: "Maximum period not achieved ❌");
                System.out.println(x: "Reason: Polynomial is reducible and not primitive");
            }

            // to show the full sequence of bits
            System.out.println("Full output sequence: " + result.output);

            assessRandomness(sequence: result.output); // method to asses the rendomness
    }

/*
 *         Randomness assessment : Check the balance
 */
private static void assessRandomness(String sequence) {
    System.out.println(x: "\n**** Simple Randomness check for feedback****");

    int ones = 0;  // to count 1s
    for (char bit : sequence.toCharArray()) {
        if (bit == '1') ones++;  // if the bit equals 1 , plass the ones
    }
    int zeros = sequence.length() - ones; // to count 0s


    // to calculate percentage :
    double onesPercent = (ones * 100.0) / sequence.length();
    double zerosPercent = (zeros * 100.0) / sequence.length();

    //       for printing
    System.out.println(x: "BALANCE TEST:");
    System.out.println("   • 1s: " + ones + " (" + String.format(format: "%.1f", args: onesPercent) + "%)");
    System.out.println("   • 0s: " + zeros + " (" + String.format(format: "%.1f", args: zerosPercent) + "%)");

```

```java
    //  if ratio is between 40-60%, it's good
    if (ones >= sequence.length()*0.4 && ones <= sequence.length()*0.6) {  // between 40-60%
        System.out.println(x: "   ✅ Good balance (close to 50/50)");
    } else {
        System.out.println(x: "   ❌ Poor balance");
    }
    // to call the pattern test:
    performPatternTest(sequence);
}
/**
 * PATTERN TEST with sliding window of size 2n (14 bits)
 * Add this method right after assessRandomness
 */
private static void performPatternTest(String sequence) {
    int n = LENGTH; // n = 7
    int windowSize = 2 * n; // 2n = 14 bits

    System.out.println("\nPATTERN TEST (sliding window: " + windowSize + " bits = 2 × " + n + "):");

    // Check if sequence is long enough
    if (sequence.length() < windowSize) {
        System.out.println("   • Sequence too short for " + windowSize + "-bit windows");
        return;
    }

    Set<String> uniquePatterns = new HashSet<>();
    int totalWindows = 0;

    // Slide the 14-bit window through the sequence
    for (int start = 0; start <= sequence.length() - windowSize; start++) {
        String window = sequence.substring(beginIndex: start, start + windowSize);
        uniquePatterns.add(e: window);
        totalWindows++;
    }

    System.out.println("   • Total sliding windows: " + totalWindows);
    System.out.println("   • Unique " + windowSize + "-bit patterns: " + uniquePatterns.size());

    // Simple assessment
    double diversity = (uniquePatterns.size() * 100.0) / totalWindows;
    if (diversity > 50) {
        System.out.println("   ✅ Good pattern diversity (" + String.format(format: "%.1f", args: diversity) + "%)");
    } else {
        System.out.println("   ❌ Limited pattern diversity (" + String.format(format: "%.1f", args: diversity) + "%)");
    }
```

```
225         // Show a few sample patterns
226         System.out.println(x: "   • Sample patterns:");
227         int count = 0;
228         for (String pattern : uniquePatterns) {
229             if (count < 2) { // Show only 2 samples
230                 System.out.println("     - " + pattern);
231                 count++;
232             } else {
233                 break;
234             }
235         }
236     }
237
238         // -------- In this section I uesed methods to help me ---------
239
240         /*Convert an array of bits to a string for displaying
241          * for example: [1,0,0,0,0,0,0] = "1 0 0 0 0 0 0" */
242         private static String arrayToString(int[] arr) {
243             try {
244                 StringBuilder sb = new StringBuilder();
245                 for (int value : arr) {
246                     sb.append(i: value).append(str:" ");
247                 }
248                 return sb.toString().trim(); //trim to remove spaces
249             } catch (Exception e) {
250                 return "Error converting array to string";
251             }
252         }
253
254         /*
255          * Compare two arrays for equality
256          * it returns true if both arrays have same length and same values in same index*/
257         private static boolean arrayEquals(int[] a, int[] b) {
258             try {
259                 if (a.length != b.length) return false;
260                 for (int i = 0; i < a.length; i++) {
261                     if (a[i] != b[i]) return false;
262                 }
263                 return true;
264             } catch (Exception e) {
265                 return false;
266             }
267         }
```

```java
    // -------- CUSTOM EXCEPTION CLASSES --------
    // These help us to handle different types of errors specifically

    static class InvalidSeedException extends Exception {
        public InvalidSeedException(String message) { super(message); }
    }

    static class InvalidTapException extends Exception {
        public InvalidTapException(String message) { super(message); }
    }
    /**
     *                    Result container class
     *           A simple class to package all our results together
     */
    static class NLFSRBack_Result {
        int period;   // How many steps until repetition
        String output; // sequence of bits generated
        int maxPeriod; // maximum period from (2^n - 1)

        NLFSRBack_Result(int period, String output, int maxPeriod) {
            this.period = period;
            this.output = output;
            this.maxPeriod = maxPeriod;
        }
    }
}

```

[4] The output of the NLFSR Feedback code:

✅ All parameters validated successfully

LFSR Analysis Polynomial: x^7 + x^4 + x^2 + 1

Max possible period: 127

Clk S6 S5 S4 S3 S2 S1 S0 Output

---

٠ ٠ ٠ ٠ ٠ ٠ ١ ٠

٠ ٠ ٠ ٠ ٠ ٠ ١ ٠ ١

٠ ٠ ٠ ٠ ٠ ١ ٠ ٠ ٢

٠ ٠ ٠ ٠ ١ ٠ ٠ ١ ٣

٠ ٠ ٠ ١ ٠ ٠ ١ ٠ ٤

٠ ٠ ١ ٠ ٠ ١ ٠ ١ ٥

٠ ١ ٠ ٠ ١ ٠ ١ ١ ٦

١ ٠ ٠ ١ ٠ ١ ١ ١ ٧

٠ ٠ ١ ٠ ١ ١ ١ ٠ ٨

٠ ١ ٠ ١ ١ ١ ٠ ٠ ٩

١ ٠ ١ ١ ١ ٠ ٠ ١ ١٠

٠ ١ ١ ١ ٠ ٠ ١ ٠ ١١

١ ١ ١ ٠ ٠ ١ ٠ ٠ ١٢

١ ١ ٠ ٠ ١ ٠ ٠ ٠ ١٣

١ ٠ ٠ ١ ٠ ٠ ٠ ١ ١٤

٠ ٠ ١ ٠ ٠ ٠ ١ ١ ١٥

٠ ١ ٠ ٠ ٠ ١ ١ ٠ ١٦

١ ٠ ٠ ٠ ١ ١ ٠ ٠ ١٧

٠ ٠ ٠ ١ ١ ٠ ٠ ١ ١٨

٠ ٠ ١ ١ ٠ ٠ ١ ١ ١٩

٠ ١ ١ ٠ ٠ ١ ١ ١ ٢٠

٢١	١٠٠١١١٠	١
٢٢	٠٠١١١٠٠	١
٢٣	٠١١١٠٠٠	٠
٢٤	١١١٠٠٠٠	٠
٢٥	١١٠٠٠٠٠	١
٢٦	١٠٠٠٠٠١	١
٢٧	٠٠٠٠٠١١	١
٢٨	٠٠٠٠١١٠	٠
٢٩	٠٠٠١١٠١	٠
٣٠	٠٠١١٠١١	٠
٣١	٠١١٠١١١	٠
٣٢	١١٠١١١٠	٠
٣٣	١٠١١١٠١	١
٣٤	٠١١١٠١١	١
٣٥	١١١٠١١٠	٠
٣٦	١١٠١١٠١	١
٣٧	١٠١١٠١١	١
٣٨	٠١١٠١١٠	١
٣٩	١١٠١١٠٠	٠
٤٠	١٠١١٠٠١	١
٤١	٠١١٠٠١٠	١
٤٢	١١٠٠١٠١	٠
٤٣	١٠٠١٠١٠	١
٤٤	٠٠١٠١٠١	١
٤٥	٠١٠١٠١٠	٠
٤٦	١٠١٠١٠١	٠
٤٧	٠١٠١٠١١	١
٤٨	١٠١٠١١١	٠

٤٩	١٠١٠١١١١	١
٥٠	١٠١١١١٠	٠
٥١	١٠١١١٠١	١
٥٢	١١١١٠١١	٠
٥٣	١١١٠١١١	١
٥٤	١١٠١١١١	١
٥٥	١٠١١١١١	١
٥٦	٠١١١١١١	١
٥٧	١١١١١١١	٠
٥٨	١١١١١١٠	١
٥٩	١١١١١٠٠	١
٦٠	١١١١٠٠٠	١
٦١	١١١٠٠٠١	١
٦٢	١١٠٠٠١٠	١
٦٣	١٠٠٠١٠١	١
٦٤	٠٠٠١٠١٠	١
٦٥	٠٠١٠١٠٠	٠
٦٦	٠١٠١٠٠٠	٠
٦٧	١٠١٠٠٠١	٠
٦٨	٠١٠٠٠١٠	١
٦٩	١٠٠٠١٠٠	٠
٧٠	٠٠٠١٠٠٠	١
٧١	٠٠١٠٠٠٠	٠
٧٢	٠١٠٠٠٠١	٠
٧٣	١٠٠٠٠١٠	٠
٧٤	٠٠٠٠١٠١	١
٧٥	٠٠٠١٠١١	٠
٧٦	٠٠١٠١١٠	٠

٠ ٠١٠١١٠٠ ٧٧

٠ ١٠١١٠٠٠ ٧٨

١ ٠١١٠٠٠٠ ٧٩

٠ ١١٠٠٠٠١ ٨٠

١ ١٠٠٠٠١١ ٨١

١ ٠٠٠٠١١١ ٨٢

٠ ٠٠٠١١١١ ٨٣

٠ ٠٠١١١١١ ٨٤

٠ ٠١١١١١٠ ٨٥

٠ ١١١١١٠١ ٨٦

١ ١١١١٠١٠ ٨٧

١ ١١١٠١٠١ ٨٨

١ ١١٠١٠١١ ٨٩

١ ١٠١٠١١٠ ٩٠

١ ٠١٠١١٠١ ٩١

٠ ١٠١١٠١٠ ٩٢

١ ٠١١٠١٠٠ ٩٣

٠ ١١٠١٠٠٠ ٩٤

١ ١٠١٠٠٠٠ ٩٥

١ ٠١٠٠٠٠٠ ٩٦

٠ ١٠٠٠٠٠٠ ٩٧

١ ٠٠٠٠٠٠١ ٩٨

**** Period: 98 ****

٠ ٠٠٠٠٠١٠ ٩٩

٠ ٠٠٠٠١٠٠ ١٠٠

٠ ٠٠٠١٠٠١ ١٠١

٠ ٠٠١٠٠١٠ ١٠٢

٠ ٠١٠٠١٠١ ١٠٣

٠ ١٠٠١٠١١ ١٠٤

١ ٠٠١٠١١١ ١٠٥

٠ ٠١٠١١١٠ ١٠٦

٠ ١٠١١١٠٠ ١٠٧

١ ٠١١١٠٠١ ١٠٨

٠ ١١١٠٠١٠ ١٠٩

١ ١١٠٠١٠٠ ١١٠

١ ١٠٠١٠٠٠ ١١١

١ ٠٠١٠٠٠١ ١١٢

٠ ٠١٠٠٠١١ ١١٣

٠ ١٠٠٠١١٠ ١١٤

١ ٠٠٠١١٠٠ ١١٥

٠ ٠٠١١٠٠١ ١١٦

٠ ٠١١٠٠١١ ١١٧

٠ ١١٠٠١١١ ١١٨

١ ١٠٠١١١٠ ١١٩

١ ٠٠١١١٠٠ ١٢٠

٠ ٠١١١٠٠٠ ١٢١

٠ ١١١٠٠٠٠ ١٢٢

١ ١١٠٠٠٠٠ ١٢٣

١ ١٠٠٠٠٠١ ١٢٤

١ ٠٠٠٠٠١١ ١٢٥

٠ ٠٠٠٠١١٠ ١٢٦

٠ ٠٠٠١١٠١ ١٢٧

٠ ٠٠١١٠١١ ١٢٨

**** ANALYSIS RESULTS ****

Period: 98

Max possible: 127

Output length: 128

❌ Maximum period not achieved

Reason: Polynomial is reducible and not primitive

Full output sequence:
00000010010111001000110011100000110111011011001010101111011111111000
10100010000101100001111101011010000001001011100100011001110000

****Simple Randomness check for feedback ****

:BALANCE TEST

(%٤٦,٩) s: 601 •

(%٥٣,١) s: 680 •

(Good balance (close to 50/50 ✅

:(PATTERN TEST (sliding window: 14 bits = 2 × 7

Total sliding windows: 115 •

Unique 14-bit patterns: 98 •

(%٨٥,٢) Good pattern diversity ✅

:Sample patterns •

٠٠١٠١٠١٠١١١١٠١ -

١٠٠٠٠١١١١١٠١٠١ -

(BUILD SUCCESSFUL (total time: 0 seconds

[5] Screen shot of the NLFSR Feedforward:

```java
/*
Joori Eihab 2310656 section CY2
 */
package lfsr;

import java.util.HashSet;
import java.util.Set;

public class NLFSRFeedForward {
    // ---------  Parameters  --------
    private static final int LENGTH = 7; //length of the polynomial , LFSR have 7FF(s6-s0
    private static final String POLYNOMIAL = "x^7 + x^4 + x^2 + 1"; //our polynomial
    private static final int[] TAPS = {2, 4, 6}; // S4, S2, S0 positions of XOR
    private static final int[] AND_TAPS = {1, 5}; // taps for AND operation ←
    private static final int[] INITIAL_SEED = {1, 0, 0, 0, 0, 0, 0}; //initial seed

    public static void main(String[] args) throws InvalidSeedException, InvalidTapException {

            // Step 1: Check if our parameters are valid before running with try-catch
            checkValidParameter();

            //Separate logic from I/O , Step 2: Run the LFSR algorithm and get results
            NLFSRForward_Result result = runNLFSR_Forward();

            // Step 3: Show the results to the user
            displayResults(result);
    }

    /**
     *            Implements LFSR algorithm
     *   This is the core function that runs the Linear Feedback Shift Register:
     */
    private static NLFSRForward_Result runNLFSR_Forward()  {

            // create a copy of the initial seed so we don't modify the original one
            int[] register = INITIAL_SEED.clone();

            //to store the output sequence (bits that come out)
            StringBuilder output = new StringBuilder();

            int period = 0;// How many steps until repetition
            int maxPeriod = (int)Math.pow(a: 2, b: LENGTH) - 1; //2^7 - 1 = 127

            // to display header informations
            System.out.println("LFSR Analysis Polynomial: " + POLYNOMIAL);
            System.out.println("Max possible period: " + maxPeriod);
```

```
46              System.out.println("Max possible period: " + maxPeriod);
47              System.out.println(x: "\nClk    S6 S5 S4 S3 S2 S1 S0    Output");
48              System.out.println(x: "-----------------------------------");
49
50              //// Show initial state (step 0) Clk 0
51              System.out.printf(format: "0      %s    %n", args: arrayToString(arr:register));
52
53              // Main LFSR loop, run to 128 cycles (more than max possible)
54              for (int clock = 1; clock <= 128; clock++) {
55
56                  int andResult = register[AND_TAPS[0]] & register[AND_TAPS[1]]; // AND between positions 1 and 5 ←
57
58                  // register[TAPS[0]] = S4, register[TAPS[1]] = S2, register[TAPS[2]] = S0
59                  int feedback = register[2] ^ register[4] ^ register[6]; // XOR with AND ←
60
61                  // The output bit is the rightmost bit (S0) before we shift
62                  int outputBit = register[6] ^ andResult; // XOR with S0 ←
63                  output.append(i: outputBit);// Add to our output sequence
64
65                  // // SHIFT operation:
66                  int[] newRegister = new int[LENGTH];// new empty register
67
68                  newRegister[0] = feedback;// Put feedback bit(result of XOR) in leftmost position (S6)
69
70
71                  // Copy all bits from old register, shifting them right by one position
72                  // register[0] goes to newRegister[1], register[1] to newRegister[2], and so on..
73                  System.arraycopy(src:register, srcPos: 0, dest:newRegister, destPos:1, LENGTH - 1);
74
75                  // Update our register to the new state
76                  register = newRegister;
77
78                  // Display the current state
79                  System.out.printf(format: "%-2d     %s    %d%n", args:clock, args:arrayToString(arr:register), args:outputBit);
80
81                  // Period detection ,check if we've returned to the initial state
82                  if (arrayEquals(a: register, b: INITIAL_SEED)) {
83                      period = clock;// found the period length
84                      System.out.println("**** Period: " + period + " ****");
85                  }
86
87              }
88              // Return all our results packaged together
89              return new NLFSRForward_Result(period, output: output.toString(), maxPeriod);
90
```

```
90              return new NLFSRForward_Result(period, output: output.toString(), maxPeriod);
91
92          }
93
94          /**
95           * This function checks if our configurations are valid before running
96           */
97          private static void checkValidParameter() throws InvalidSeedException, InvalidTapException {
98              try {
99                  // Validate seed length must be exactly 7 bits
100                 if (INITIAL_SEED.length != LENGTH) {
101                     throw new InvalidSeedException("Seed length must be " + LENGTH);
102                 }
103
104                 // Validate seed is not all zeros , because all-zero state would cause problems
105                 boolean allZero = true;
106                 for (int bit : INITIAL_SEED) {
107                     if (bit != 0) {
108                         allZero = false; // Found a 1, so not all zeros
109                         break;
110                     }
111                 }
112                 if (allZero) {
113                     throw new InvalidSeedException(message:"Seed cannot be all zeros");
114                 }
115
116                 // Validate tap array not empty , need at least one tap
117                 if (TAPS.length == 0) {
118                     throw new InvalidTapException(message:"Taps array cannot be empty");
119                 }
120
121                 System.out.println(x: " All parameters validated successfully ✅");
122
123             }
124             catch (InvalidSeedException | InvalidTapException e) {
125                 throw e; }// Re-throw specific exceptions to main method
126              catch (Exception e) {
127                 throw new InvalidSeedException("Unexpected error during parameter validation: " + e.getMessage());
128             }
129         }
130
131         /**
132          *            Display results with analysis
133          *            Shows the final results to the user
134          */
```

```java
    private static void displayResults(NLFSRForward_Result result) {

            System.out.println("\n**** ANALYSIS RESULTS ****");
            System.out.println("Period: " + result.period);
            System.out.println("Max possible: " + result.maxPeriod);
            System.out.println("Output length: " + result.output.length());


            //Check if we achieved maximum period (in our case no)
            if (result.period == result.maxPeriod) {
                System.out.println("Maximum period achieved ✅");
            }
            else {
                System.out.println("Maximum period not achieved ❌");
                System.out.println("Reason: Polynomial is reducible and not primitive");
            }

            // to show the full sequence of bits
            System.out.println("Full output sequence: " + result.output);

            assessRandomness(sequence: result.output); // method to asses the rendomness
    }

/*
 *         Randomness assessment : Check the balance
 */
private static void assessRandomness(String sequence) {
    System.out.println("\n**** Simple Randomness check for feedforward****");

    int ones = 0;  // to count 1s
    for (char bit : sequence.toCharArray()) {
        if (bit == '1') ones++;  // if the bit equals 1 , plass the ones
    }
    int zeros = sequence.length() - ones; // to count 0s


    // to calculate percentage :
    double onesPercent = (ones * 100.0) / sequence.length();
    double zerosPercent = (zeros * 100.0) / sequence.length();

    //       for printing
    System.out.println("BALANCE TEST:");
    System.out.println("   • 1s: " + ones + " (" + String.format(format: "%.1f", args: onesPercent) + "%)");
    System.out.println("   • 0s: " + zeros + " (" + String.format(format: "%.1f", args: zerosPercent) + "%)");

```

```java
    //  if ratio is between 40-60%, it's good
    if (ones >= sequence.length()*0.4 && ones <= sequence.length()*0.6) {  // between 40-60%
        System.out.println("   ✅ Good balance (close to 50/50)");
    } else {
        System.out.println("   ❌ Poor balance");
    }

    // to call the pattern test:
    performPatternTest(sequence);
}
/**
 * PATTERN TEST with sliding window of size 2n (14 bits)
 * Add this method right after assessRandomness
 */
private static void performPatternTest(String sequence) {
    int n = LENGTH; // n = 7
    int windowSize = 2 * n; // 2n = 14 bits

    System.out.println("\nPATTERN TEST (sliding window: " + windowSize + " bits = 2 × " + n + "):");

    // Check if sequence is long enough
    if (sequence.length() < windowSize) {
        System.out.println("   • Sequence too short for " + windowSize + "-bit windows");
        return;
    }

    Set<String> uniquePatterns = new HashSet<>();
    int totalWindows = 0;

    // Slide the 14-bit window through the sequence
    for (int start = 0; start <= sequence.length() - windowSize; start++) {
        String window = sequence.substring(beginIndex: start, start + windowSize);
        uniquePatterns.add(e: window);
        totalWindows++;
    }

    System.out.println("   • Total sliding windows: " + totalWindows);
    System.out.println("   • Unique " + windowSize + "-bit patterns: " + uniquePatterns.size());

    // Simple assessment
    double diversity = (uniquePatterns.size() * 100.0) / totalWindows;
    if (diversity > 50) {
        System.out.println("   ✅ Good pattern diversity (" + String.format(format: "%.1f", args: diversity) + "%)");
    } else {
        System.out.println("   ❌ Limited pattern diversity (" + String.format(format: "%.1f", args: diversity) + "%)");
```

```
225             System.out.println("   ❌ Limited pattern diversity (" + String.format(format: "%.1f", args: diversity) + "%)");
226         }
227         // Show a few sample patterns
228         System.out.println(x: "   • Sample patterns:");
229         int count = 0;
230         for (String pattern : uniquePatterns) {
231             if (count < 2) { // Show only 2 samples
232                 System.out.println("     - " + pattern);
233                 count++;
234             } else {
235                 break;
236             }
237         }
238     }
239
240
241     // -------- In this section I uesed methods to help me --------
242
243     /*Convert an array of bits to a string for displaying
244      * for example: [1,0,0,0,0,0,0] = "1 0 0 0 0 0 0" */
245     private static String arrayToString(int[] arr) {
246         try {
247             StringBuilder sb = new StringBuilder();
248             for (int value : arr) {
249                 sb.append(i: value).append(str: " ");
250             }
251             return sb.toString().trim(); //trim to remove spaces
252         } catch (Exception e) {
253             return "Error converting array to string";
254         }
255     }
256
257     /*
258      * Compare two arrays for equality
259      * it returns true if both arrays have same length and same values in same index*/
260     private static boolean arrayEquals(int[] a, int[] b) {
261         try {
262             if (a.length != b.length) return false;
263             for (int i = 0; i < a.length; i++) {
264                 if (a[i] != b[i]) return false;
265             }
266             return true;
267         } catch (Exception e) {
268             return false;
269         }
```

```
269            }
270        }
271
272
273        // -------- CUSTOM EXCEPTION CLASSES --------
274        // These help us to handle different types of errors specifically
275
276        static class InvalidSeedException extends Exception {
277            public InvalidSeedException(String message) { super(message); }
278        }
279
280        static class InvalidTapException extends Exception {
281            public InvalidTapException(String message) { super(message); }
282        }
283
284
285        /**
286         *                      Result container class
287         *           A simple class to package all our results together
288         */
289        static class NLFSRForward_Result {
290            int period;   // How many steps until repetition
291            String output; // sequence of bits generated
292            int maxPeriod; // maximum period from (2^n - 1)
293
294            NLFSRForward_Result(int period, String output, int maxPeriod) {
295                this.period = period;
296                this.output = output;
297                this.maxPeriod = maxPeriod;
298            }
299        }
300    }
301
```

[6] The output of the NLFSR Feedforward code:

All parameters validated successfully ✅

LFSR Analysis Polynomial: x^7 + x^4 + x^2 + 1

Max possible period: 127

Clk S6 S5 S4 S3 S2 S1 S0 Output

---

0 1 0 0 0 0 0 0

١ 0 1 0 0 0 0 0 ٠

٢ 0 0 1 0 0 0 0 ٠

٣ 1 0 0 1 0 0 0 ٠

٤ 0 1 0 0 1 0 0 ٠

٥ 1 0 1 0 0 1 0 ٠

٦ 1 1 0 1 0 0 1 ٠

٧ 1 1 1 0 1 0 0 ١

٨ 0 1 1 1 0 1 0 ٠

٩ 1 0 1 1 1 0 1 ١

١٠ 1 1 0 1 1 1 0 ١

١١ 1 1 1 0 1 1 1 ١

١٢ 1 1 1 1 0 1 1 ٠

١٣ 0 1 1 1 1 0 1 ٠

١٤ 1 0 1 1 1 1 0 ١

١٥ 0 1 0 1 1 1 1 ٠

١٦ 0 0 1 0 1 1 1 ٠

١٧	1001011	١
١٨	1100101	١
١٩	0110010	١
٢٠	1011001	١
٢١	0101100	١
٢٢	1010110	٠
٢٣	0101011	٠
٢٤	1010101	٠
٢٥	1101010	١
٢٦	0110101	١
٢٧	1011010	١
٢٨	1101101	٠
٢٩	0110110	١
٣٠	0011011	١
٣١	0001101	١
٣٢	0000110	١
٣٣	1000011	٠
٣٤	1100001	١
٣٥	1110000	١
٣٦	1111000	٠
٣٧	1111100	٠
٣٨	0111110	٠
٣٩	0011111	١
٤٠	1001111	١
٤١	0100111	١

٤٢ 0 0 1 0 0 1 1 ٠

٤٣ 0 0 0 1 0 0 1 ١

٤٤ 1 0 0 0 1 0 0 ١

٤٥ 1 1 0 0 0 1 0 ٠

٤٦ 0 1 1 0 0 0 1 ١

٤٧ 0 0 1 1 0 0 0 ١

٤٨ 1 0 0 1 1 0 0 ٠

٤٩ 1 1 0 0 1 1 0 ٠

٥٠ 1 1 1 0 0 1 1 ١

٥١ 0 1 1 1 0 0 1 ٠

٥٢ 0 0 1 1 1 0 0 ١

٥٣ 0 0 0 1 1 1 0 ٠

٥٤ 1 0 0 0 1 1 1 ٠

٥٥ 0 1 0 0 0 1 1 ١

٥٦ 1 0 1 0 0 0 1 ٠

٥٧ 0 1 0 1 0 0 0 ١

٥٨ 0 0 1 0 1 0 0 ٠

٥٩ 0 0 0 1 0 1 0 ٠

٦٠ 0 0 0 0 1 0 1 ٠

٦١ 0 0 0 0 0 1 0 ١

٦٢ 0 0 0 0 0 0 1 ٠

٦٣ 1 0 0 0 0 0 0 ١

**** Period: 63 ****

٦٤ 0 1 0 0 0 0 0 ٠

٦٥ 0 0 1 0 0 0 0 ٠

٦٦	1001000	٠
٦٧	0100100	٠
٦٨	1010010	٠
٦٩	1101001	٠
٧٠	1110100	١
٧١	0111010	٠
٧٢	1011101	١
٧٣	1101110	١
٧٤	1110111	١
٧٥	1111011	٠
٧٦	0111101	٠
٧٧	1011110	١
٧٨	0101111	٠
٧٩	0010111	٠
٨٠	1001011	١
٨١	1100101	١
٨٢	0110010	١
٨٣	1011001	١
٨٤	0101100	١
٨٥	1010110	٠
٨٦	0101011	٠
٨٧	1010101	٠
٨٨	1101010	١
٨٩	0110101	١
٩٠	1011010	١

٩١	1 1 0 1 1 0 1	٠
٩٢	0 1 1 0 1 1 0	١
٩٣	0 0 1 1 0 1 1	١
٩٤	0 0 0 1 1 0 1	١
٩٥	0 0 0 0 1 1 0	١
٩٦	1 0 0 0 0 1 1	٠
٩٧	1 1 0 0 0 0 1	١
٩٨	1 1 1 0 0 0 0	١
٩٩	1 1 1 1 0 0 0	٠
١٠٠	1 1 1 1 1 0 0	٠
١٠١	0 1 1 1 1 1 0	٠
١٠٢	0 0 1 1 1 1 1	١
١٠٣	1 0 0 1 1 1 1	١
١٠٤	0 1 0 0 1 1 1	١
١٠٥	0 0 1 0 0 1 1	٠
١٠٦	0 0 0 1 0 0 1	١
١٠٧	1 0 0 0 1 0 0	١
١٠٨	1 1 0 0 0 1 0	٠
١٠٩	0 1 1 0 0 0 1	١
١١٠	0 0 1 1 0 0 0	١
١١١	1 0 0 1 1 0 0	٠
١١٢	1 1 0 0 1 1 0	٠
١١٣	1 1 1 0 0 1 1	١
١١٤	0 1 1 1 0 0 1	٠
١١٥	0 0 1 1 1 0 0	١

١١٦ 0 0 0 1 1 1 0 ٠

١١٧ 1 0 0 0 1 1 1 ٠

١١٨ 0 1 0 0 0 1 1 ١

١١٩ 1 0 1 0 0 0 1 ٠

١٢٠ 0 1 0 1 0 0 0 ١

١٢١ 0 0 1 0 1 0 0 ٠

١٢٢ 0 0 0 1 0 1 0 ٠

١٢٣ 0 0 0 0 1 0 1 ٠

١٢٤ 0 0 0 0 0 1 0 ١

١٢٥ 0 0 0 0 0 0 1 ٠

١٢٦ 1 0 0 0 0 0 0 ١

**** Period: 126 ****

١٢٧ 0 1 0 0 0 0 0 ٠

١٢٨ 0 0 1 0 0 0 0 ٠

**** ANALYSIS RESULTS ****

Period: 126

Max possible: 127

Output length: 128

Maximum period not achieved ❌

Reason: Polynomial is reducible and not primitive

Full output sequence:
00000010111001001111100011101111011000111011011001010010100010
10000001011100100111110001110111101100011101101100101001010001
0100

**** Simple Randomness check for feedforward****

BALANCE TEST:

- 1s: 64 (٥٠,٠%)
- 0s: 64 (٥٠,٠%)

✅ Good balance (close to 50/50)

PATTERN TEST (sliding window: 14 bits = 2 × 7):

- Total sliding windows: 115
- Unique 14-bit patterns: 63

✅ Good pattern diversity (٥٤,٨%)

- Sample patterns:
  - 11111000111011
  - 01001111100011

BUILD SUCCESSFUL (total time: 0 seconds)

[7] Screen shot of the Both NLFSR :

```java
/*
Joori Eihab 2310656 section CY2
 */
package lfsr;

import java.util.HashSet;
import java.util.Set;

public class NLFSRBoth {
    // ---------  Parameters  --------
    private static final int LENGTH = 7; //length of the polynomial , LFSR have 7FF(s6-s0
    private static final String POLYNOMIAL = "x^7 + x^4 + x^2 + 1"; //our polynomial
    private static final int[] TAPS = {2, 4, 6}; // S4, S2, S0 positions of XOR
    private static final int[] AND_TAPS_FEEDBACK = {3, 5}; // NEW: AND taps for feedback (positions 3 and 5)
    private static final int[] AND_TAPS_OUTPUT = {1, 5}; // AND taps for output (positions 1 and 5)
    private static final int[] INITIAL_SEED = {1, 0, 0, 0, 0, 0, 0}; //initial seed

    public static void main(String[] args) throws InvalidSeedException, InvalidTapException {

            // Step 1: Check if our parameters are valid before running with try-catch
            checkValidParameter();

            //Separate logic from I/O , Step 2: Run the LFSR algorithm and get results
            NLFSRBoth_Result result = runNLFSR_Both();

            // Step 3: Show the results to the user
            displayResults(result);
    }

    /**
     *           Implements LFSR algorithm
     *   This is the core function that runs the Linear Feedback Shift Register:
     */
    private static NLFSRBoth_Result runNLFSR_Both()  {

            // create a copy of the initial seed so we don't modify the original one
            int[] register = INITIAL_SEED.clone();

            //to store the output sequence (bits that come out)
            StringBuilder output = new StringBuilder();

            int period = 0;// How many steps until repetition
            int maxPeriod = (int)Math.pow(a: 2, b: LENGTH) - 1; //2^7 - 1 = 127

            // to display header informations
```

```java
        // to display header informations
        System.out.println("LFSR Analysis Polynomial: " + POLYNOMIAL);
        System.out.println("Max possible period: " + maxPeriod);
        System.out.println(x: "\nClk    S6 S5 S4 S3 S2 S1 S0    Output");
        System.out.println(x: "-----------------------------------");

        //// Show initial state (step 0) Clk 0
        System.out.printf(format: "0      %s    %n", args:arrayToString(arr:register));

        // Main LFSR loop, run to 128 cycles (more than max possible)
        for (int clock = 1; clock <= 128; clock++) {

            int andFeedback = register[AND_TAPS_FEEDBACK[0]] & register[AND_TAPS_FEEDBACK[1]]; // AND between positions 3 and 5

            int feedback = register[2] ^ register[4] ^ register[6] ^ andFeedback;


            int andOutput = register[AND_TAPS_OUTPUT[0]] & register[AND_TAPS_OUTPUT[1]]; // AND between positions 1 and 5

            int outputBit = register[6] ^ andOutput;
            output.append(i: outputBit);// Add to our output sequence

            // // SHIFT operation:
            int[] newRegister = new int[LENGTH];// new empty register

            newRegister[0] = feedback;// Put feedback bit(result of XOR) in leftmost position (S6)


            // Copy all bits from old register, shifting them right by one position
            // register[0] goes to newRegister[1], register[1] to newRegister[2], and so on..
            System.arraycopy(src:register, srcPos: 0, dest:newRegister, destPos:1, LENGTH - 1);

            // Update our register to the new state
            register = newRegister;

            // Display the current state
            System.out.printf(format: "%-2d     %s    %d%n", args:clock, args:arrayToString(arr:register), args:outputBit);

            // Period detection ,check if we've returned to the initial state
            if (arrayEquals(a: register, b: INITIAL_SEED)) {
                period = clock;// found the period length
                System.out.println("**** Period: " + period + " ****");
                break; // Exit the loop since we found repetition
            }
```

```
            }
            // Return all our results packaged together
            return new NLFSRBoth_Result(period, output: output.toString(), maxPeriod);
    }
    /**
     * This function checks if our configurations are valid before running
     */
    private static void checkValidParameter() throws InvalidSeedException, InvalidTapException {
        try {
            // Validate seed length must be exactly 7 bits
            if (INITIAL_SEED.length != LENGTH) {
                throw new InvalidSeedException("Seed length must be " + LENGTH);
            }

            // Validate seed is not all zeros , because all-zero state would cause problems
            boolean allZero = true;
            for (int bit : INITIAL_SEED) {
                if (bit != 0) {
                    allZero = false; // Found a 1, so not all zeros
                    break;
                }
            }
            if (allZero) {
                throw new InvalidSeedException(message:"Seed cannot be all zeros");
            }

            // Validate tap array not empty , need at least one tap
            if (TAPS.length == 0) {
                throw new InvalidTapException(message:"Taps array cannot be empty");
            }

            System.out.println(x: " All parameters validated successfully ✅");

        }
        catch (InvalidSeedException | InvalidTapException e) {
            throw e; }// Re-throw specific exceptions to main method
         catch (Exception e) {
            throw new InvalidSeedException("Unexpected error during parameter validation: " + e.getMessage());
        }
    }

    /**
     *            Display results with analysis
     *            Shows the final results to the user
     */
```

```
    private static void displayResults(NLFSRBoth_Result result) {

            System.out.println(x: "\n**** ANALYSIS RESULTS ****");
            System.out.println("Period: " + result.period);
            System.out.println("Max possible: " + result.maxPeriod);
            System.out.println("Output length: " + result.output.length());


            //Check if we achieved maximum period (in our case no)
            if (result.period == result.maxPeriod) {
                System.out.println(x: "Maximum period achieved ✅");
            }
            else {
                System.out.println(x: "Maximum period not achieved ❌");
                System.out.println(x: "Reason: Polynomial is reducible and not primitive");
            }

            // to show the full sequence of bits
            System.out.println("Full output sequence: " + result.output);

            assessRandomness(sequence: result.output); // method to asses the rendomness
    }

/*
 *         Randomness assessment : Check the balance
 */
private static void assessRandomness(String sequence) {
    System.out.println(x: "\n**** Simple Randomness check for both feedback & feedforward****");

    int ones = 0;  // to count 1s
    for (char bit : sequence.toCharArray()) {
        if (bit == '1') ones++;  // if the bit equals 1 , plass the ones
    }
    int zeros = sequence.length() - ones; // to count 0s


    // to calculate percentage :
    double onesPercent = (ones * 100.0) / sequence.length();
    double zerosPercent = (zeros * 100.0) / sequence.length();

    //       for printing
    System.out.println(x: "BALANCE TEST:");
    System.out.println("    • 1s: " + ones + " (" + String.format(format: "%.1f", args: onesPercent) + "%)");
    System.out.println("    • 0s: " + zeros + " (" + String.format(format: "%.1f", args: zerosPercent) + "%)");

```

```
        //  if ratio is between 40-60%, it's good
        if (ones >= sequence.length()*0.4 && ones <= sequence.length()*0.6) {  // between 40-60%
            System.out.println(x: "   ✅ Good balance (close to 50/50)");
        } else {
            System.out.println(x: "   ❌ Poor balance");
        }

        // to call the pattern test:
        performPatternTest(sequence);
    }
    /**
     * PATTERN TEST with sliding window of size 2n (14 bits)
     * Add this method right after assessRandomness
     */
    private static void performPatternTest(String sequence) {
        int n = LENGTH; // n = 7
        int windowSize = 2 * n; // 2n = 14 bits

        System.out.println("\nPATTERN TEST (sliding window: " + windowSize + " bits = 2 × " + n + "):");

        // Check if sequence is long enough
        if (sequence.length() < windowSize) {
            System.out.println("   • Sequence too short for " + windowSize + "-bit windows");
            return;
        }

        Set<String> uniquePatterns = new HashSet<>();
        int totalWindows = 0;

        // Slide the 14-bit window through the sequence
        for (int start = 0; start <= sequence.length() - windowSize; start++) {
            String window = sequence.substring(beginIndex: start, start + windowSize);
            uniquePatterns.add(e: window);
            totalWindows++;
        }

        System.out.println("   • Total sliding windows: " + totalWindows);
        System.out.println("   • Unique " + windowSize + "-bit patterns: " + uniquePatterns.size());

        // Simple assessment
        double diversity = (uniquePatterns.size() * 100.0) / totalWindows;
        if (diversity > 50) {
            System.out.println("   ✅ Good pattern diversity (" + String.format(format: "%.1f", args:diversity) + "%)");
        } else {
            System.out.println("   ❌ Limited pattern diversity (" + String.format(format: "%.1f", args:diversity) + "%)");
```

```
            System.out.println("   ❌ Limited pattern diversity (" + String.format(format: "%.1f", args:diversity) + "%)");
        }

        // Show a few sample patterns
        System.out.println(x: "   • Sample patterns:");
        int count = 0;
        for (String pattern : uniquePatterns) {
            if (count < 2) { // Show only 2 samples
                System.out.println("     - " + pattern);
                count++;
            } else {
                break;
            }
        }
    }

    // -------- In this section I uesed methods to help me --------

    /*Convert an array of bits to a string for displaying
     * for example: [1,0,0,0,0,0,0] = "1 0 0 0 0 0 0" */
    private static String arrayToString(int[] arr) {
        try {
            StringBuilder sb = new StringBuilder();
            for (int value : arr) {
                sb.append(i: value).append(str:" ");
            }
            return sb.toString().trim(); //trim to remove spaces
        } catch (Exception e) {
            return "Error converting array to string";
        }
    }

    /*
     * Compare two arrays for equality
     * it returns true if both arrays have same length and same values in same index*/
    private static boolean arrayEquals(int[] a, int[] b) {
        try {
            if (a.length != b.length) return false;
            for (int i = 0; i < a.length; i++) {
                if (a[i] != b[i]) return false;
            }
            return true;
        } catch (Exception e) {
            return false;
        }
```

```
        }
    }


    // -------- CUSTOM EXCEPTION CLASSES --------
    // These help us to handle different types of errors specifically

    static class InvalidSeedException extends Exception {
        public InvalidSeedException(String message) { super(message); }
    }

    static class InvalidTapException extends Exception {
        public InvalidTapException(String message) { super(message); }
    }


    /**
     *                    Result container class
     *           A simple class to package all our results together
     */
    static class NLFSRBoth_Result {
        int period;   // How many steps until repetition
        String output; // sequence of bits generated
        int maxPeriod; // maximum period from (2^n - 1)

        NLFSRBoth_Result(int period, String output, int maxPeriod) {
            this.period = period;
            this.output = output;
            this.maxPeriod = maxPeriod;
        }
    }

}

```

[8] The output of the Both NLFSR code:

All parameters validated successfully ✅

LFSR Analysis Polynomial: x^7 + x^4 + x^2 + 1

Max possible period: 127

Clk S6 S5 S4 S3 S2 S1 S0 Output

---

0 1 0 0 0 0 0 0

١ 0 1 0 0 0 0 0 ٠

٢ 0 0 1 0 0 0 0 ٠

٣ 1 0 0 1 0 0 0 ٠

٤ 0 1 0 0 1 0 0 ٠

٥	1010010	٠
٦	1101001	٠
٧	1110100	١
٨	0111010	٠
٩	1011101	١
١٠	1101110	١
١١	1110111	١
١٢	1111011	٠
١٣	0111101	٠
١٤	1011110	١
١٥	0101111	٠
١٦	0010111	٠
١٧	1001011	١
١٨	1100101	١
١٩	0110010	١
٢٠	1011001	١
٢١	0101100	١
٢٢	1010110	٠
٢٣	0101011	٠
٢٤	1010101	٠
٢٥	1101010	١
٢٦	0110101	١
٢٧	1011010	١
٢٨	1101101	٠
٢٩	0110110	١

٣٠	0011011	١
٣١	0001101	١
٣٢	0000110	١
٣٣	1000011	٠
٣٤	1100001	١
٣٥	1110000	١
٣٦	1111000	٠
٣٧	1111100	٠
٣٨	0111110	٠
٣٩	0011111	١
٤٠	1001111	١
٤١	0100111	١
٤٢	0010011	٠
٤٣	0001001	١
٤٤	1000100	١
٤٥	1100010	٠
٤٦	0110001	١
٤٧	0011000	١
٤٨	1001100	٠
٤٩	1100110	٠
٥٠	1110011	١
٥١	0111001	٠
٥٢	0011100	١
٥٣	0001110	٠
٥٤	1000111	٠

٥٥ 0 1 0 0 0 1 1 ١

٥٦ 1 0 1 0 0 0 1 ٠

٥٧ 0 1 0 1 0 0 0 ١

٥٨ 0 0 1 0 1 0 0 ٠

٥٩ 0 0 0 1 0 1 0 ٠

٦٠ 0 0 0 0 1 0 1 ٠

٦١ 0 0 0 0 0 1 0 ١

٦٢ 0 0 0 0 0 0 1 ٠

٦٣ 1 0 0 0 0 0 0 ١

**** Period: 63 ****

٦٤ 0 1 0 0 0 0 0 ٠

٦٥ 0 0 1 0 0 0 0 ٠

٦٦ 1 0 0 1 0 0 0 ٠

٦٧ 0 1 0 0 1 0 0 ٠

٦٨ 1 0 1 0 0 1 0 ٠

٦٩ 1 1 0 1 0 0 1 ٠

٧٠ 1 1 1 0 1 0 0 ١

٧١ 0 1 1 1 0 1 0 ٠

٧٢ 1 0 1 1 1 0 1 ١

٧٣ 1 1 0 1 1 1 0 ١

٧٤ 1 1 1 0 1 1 1 ١

٧٥ 1 1 1 1 0 1 1 ٠

٧٦ 0 1 1 1 1 0 1 ٠

٧٧ 1 0 1 1 1 1 0 ١

٧٨ 0 1 0 1 1 1 1 ٠

٧٩	0010111	٠
٨٠	1001011	١
٨١	1100101	١
٨٢	0110010	١
٨٣	1011001	١
٨٤	0101100	١
٨٥	1010110	٠
٨٦	0101011	٠
٨٧	1010101	٠
٨٨	1101010	١
٨٩	0110101	١
٩٠	1011010	١
٩١	1101101	٠
٩٢	0110110	١
٩٣	0011011	١
٩٤	0001101	١
٩٥	0000110	١
٩٦	1000011	٠
٩٧	1100001	١
٩٨	1110000	١
٩٩	1111000	٠
١٠٠	1111100	٠
١٠١	0111110	٠
١٠٢	0011111	١
١٠٣	1001111	١

١٠٤ 0 1 0 0 1 1 1 ١

١٠٥ 0 0 1 0 0 1 1 ٠

١٠٦ 0 0 0 1 0 0 1 ١

١٠٧ 1 0 0 0 1 0 0 ١

١٠٨ 1 1 0 0 0 1 0 ٠

١٠٩ 0 1 1 0 0 0 1 ١

١١٠ 0 0 1 1 0 0 0 ١

١١١ 1 0 0 1 1 0 0 ٠

١١٢ 1 1 0 0 1 1 0 ٠

١١٣ 1 1 1 0 0 1 1 ١

١١٤ 0 1 1 1 0 0 1 ٠

١١٥ 0 0 1 1 1 0 0 ١

١١٦ 0 0 0 1 1 1 0 ٠

١١٧ 1 0 0 0 1 1 1 ٠

١١٨ 0 1 0 0 0 1 1 ١

١١٩ 1 0 1 0 0 0 1 ٠

١٢٠ 0 1 0 1 0 0 0 ١

١٢١ 0 0 1 0 1 0 0 ٠

١٢٢ 0 0 0 1 0 1 0 ٠

١٢٣ 0 0 0 0 1 0 1 ٠

١٢٤ 0 0 0 0 0 1 0 ١

١٢٥ 0 0 0 0 0 0 1 ٠

١٢٦ 1 0 0 0 0 0 0 ١

**** Period: 126 ****

١٢٧ 0 1 0 0 0 0 0 ٠

١٢٨ 0 0 1 0 0 0 0 ٠

**** ANALYSIS RESULTS ****

Period: 126

Max possible: 127

Output length: 128

Maximum period not achieved ❌

Reason: Polynomial is reducible and not primitive

Full output sequence:
00000010111001001111100011101111011000111011011001010010100010
10000001011100100111110001110111101100011101101100101001010001
0100

**** Simple Randomness check for feedforward****

BALANCE TEST:

- 1s: 64 (٥٠,٠%)
- 0s: 64 (٥٠,٠%)

✅ Good balance (close to 50/50)

PATTERN TEST (sliding window: 14 bits = 2 × 7):

- Total sliding windows: 115
- Unique 14-bit patterns: 63

✅ Good pattern diversity (٥٤,٨%)

- Sample patterns:

  - 11111000111011

  - 01001111100011

```
BUILD SUCCESSFUL (total time: 0 seconds)
```