

Cryptography Lab 1

Done By: Joori Eihab Alhindi

ID: 2310656

Table of Contents:

- 1 Implementing and Analyzing a Linear Feedback Shift Register (LFSR)
- 2 Non-linear Feedback/Feedforward Shift Register (NFSR) Design and Implementation
- 3 Appendix:

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 $p(1) = 1 + 1 + 1 + 1 = 4 \equiv 0 \pmod{2}$, 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 $2^n - 1$

$$2.1) - p(x) = x^7 + x^4 + x^2 + 1 \quad \text{we know that:}$$

Step 1 Look for a root: $x \oplus R1 = 0$

$$\begin{array}{l} \textcircled{1} \boxed{x=0} \quad \text{since} \\ 0 \oplus R0 = 0 \end{array}$$

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

$$= 0 + 0 + 0 + 1$$

$$= 1 \neq 0, \text{ so } x=0 \text{ is not root}$$

$$\textcircled{2} \boxed{x=1} \quad \text{since}$$

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

$$= 1 + 1 + 1 + 1$$

$$= 4 \text{ it's even so}$$

$$p(1) = 0 \checkmark, \text{ so } x=1 \text{ is root}$$

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:

$$p(x) = x^7 + x^4 + x^2 + 1 \quad \text{same as } x^7 + ox^6 + ox^5 + x^4 + ox^3 + x^2 + ox + 1$$

$$\begin{array}{r} \text{multiply } x^6 + x^5 + x^4 + x^3 + x^2 + ox + 1 \\ \times \boxed{x+1} \quad \text{divide } \frac{x^7}{x} = x^6 \end{array}$$

$$\begin{array}{r} \cancel{(x^7 + ox^6 + ox^5 + x^4 + ox^3 + x^2 + ox + 1)} \\ \cancel{(x^6 + x^5 + x^4 + x^3 + x^2 + ox + 1)} - \\ \cancel{(x^6 + ox^5 + x^4 + ox^3 + x^2 + ox + 1)} \quad \text{divide } \frac{x^6}{x} = x^5 \\ \cancel{(x^5 + x^4 + x^3 + x^2 + ox + 1)} - \end{array}$$

$$\begin{array}{r} \cancel{(x^5 + x^4 + x^3 + x^2 + ox + 1)} \quad \text{divide } \frac{x^5}{x} = x^4 \\ \cancel{(x^4 + x^3)} - \end{array}$$

$$\begin{array}{r} \cancel{(x^4 + x^3)} - \quad \text{divide } \frac{x^4}{x} = x^3 \\ \cancel{(x^3 + x^2)} - \end{array}$$

$$\begin{array}{r} \cancel{(x^3 + x^2)} - \quad \text{divide } \frac{x^3}{x} = x^2 \\ \cancel{(x^2 + x)} - \end{array}$$

$$\begin{array}{r} \cancel{(x^2 + x)} - \quad \text{divide } \frac{x^2}{x} = x \\ \cancel{(x + 1)} - \end{array}$$

0

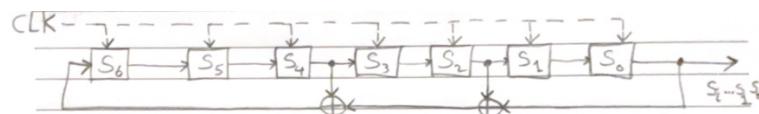
$$\text{So, the result} = x^6 + x^5 + x^4 + x + 1$$

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

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:

- Choose a Programming Language:

I have used Java Language for implementing the LFSR.

- Define the LFSR Parameters:

- Length of the LFSR (n).

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

- Feedback polynomial

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

- 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]

- Write the LFSR Algorithm:

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

Done in code. [1]

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

Done in code. [1]

- 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:

- Run the LFSR and record the generated sequence.

Full output sequence: 0000001001011101111010011010101101000011110010001100111000101

[To see the full output \[2\].](#)

- 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	Clk	S6	S5	S4	S3	S2	S1	S0	Output
0	1	0	0	0	0	0	0		30	0	0	1	1	0	1	1	.
1	0	1	0	0	0	0	0	.	31	0	0	0	1	1	0	1	.
2	0	0	1	0	0	0	0	.	32	0	0	0	0	1	1	0	.
3	1	0	0	1	0	0	0	.	33	1	0	0	0	0	1	1	.
4	0	1	0	0	1	0	0	.	34	1	1	0	0	0	0	1	.
5	1	0	1	0	0	1	0	.	35	1	1	1	0	0	0	0	.
6	1	1	0	1	0	0	1	.	36	1	1	1	1	0	0	0	.
7	1	1	1	0	1	0	0	.	37	1	1	1	1	1	0	0	.
8	0	1	1	1	0	1	0	.	38	0	1	1	1	1	1	0	.
9	1	0	1	1	1	0	1	.	39	0	0	1	1	1	1	1	.
10	1	1	0	1	1	1	0	.	40	1	0	0	1	1	1	1	.
11	1	1	1	0	1	1	1	.	41	0	1	0	0	1	1	1	.
12	1	1	1	1	0	1	1	.	42	0	0	1	0	0	1	1	.
13	0	1	1	1	1	0	1	.	43	0	0	0	1	0	0	1	.
14	1	0	1	1	1	1	0	.	44	1	0	0	0	1	0	0	.
15	0	1	0	1	1	1	1	.	45	1	1	0	0	0	1	0	.
16	0	0	1	0	1	1	1	.	46	0	1	1	0	0	0	1	.
17	1	0	0	1	0	1	1	.	47	0	0	1	1	0	0	1	.
18	1	1	0	0	1	0	1	.	48	0	0	1	1	0	0	0	.
19	0	1	1	0	0	1	0	.	49	1	0	0	1	1	0	0	.
20	1	0	1	1	0	0	1	.	50	1	1	0	0	1	1	0	.
21	0	1	0	1	1	1	1	.	51	0	0	1	1	1	0	1	.
22	0	0	1	0	1	1	1	.	52	0	0	1	1	1	1	0	.
23	1	0	0	0	1	1	1	.	53	1	0	1	0	0	0	1	.
24	1	1	0	0	0	1	1	.	54	0	1	0	1	0	0	0	.
25	1	1	1	0	0	0	1	.	55	0	0	1	0	0	0	1	.
26	1	1	1	1	0	0	0	.	56	1	0	0	0	0	0	0	.
27	1	1	1	1	1	0	0	.	57	0	0	0	0	0	0	0	.
28	0	1	1	1	1	1	0	.	58	0	0	1	0	1	0	0	.
29	0	0	1	1	1	1	1	.	59	0	0	0	1	0	1	0	.
30	0	0	0	1	1	1	1	.	60	0	0	0	0	1	0	1	.
31	1	0	0	0	0	1	1	.	61	0	0	0	0	0	1	0	.
32	1	1	0	0	0	0	1	.	62	0	0	0	0	0	0	1	.
33	1	1	1	0	0	0	0	.	63	1	0	0	0	0	0	0	.

**** 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 achieve 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 achieve 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:

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

    int ones = 0; // to count 1s
    for (char bit : sequence.toCharArray()) {
        if (bit == '1') ones++; // if the bit equals 1 , plus 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) + "%)");

    // 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");
    }
}
```

```

188     /* PATTERN TEST with sliding window of size 2n (14 bits)
189      * Add this method right after assessRandomness */
190  □ private static void performPatternTest(String sequence) {
191      int n = LENGTH; // n = 7
192      int windowHeight = 2 * n; // 2n = 14 bits
193
194      System.out.println("\nPATTERN TEST (sliding window: " + windowHeight + " bits = 2 × " + n + ":)");
195
196      // Check if sequence is long enough
197      if (sequence.length() < windowHeight) {
198          System.out.println("  • Sequence too short for " + windowHeight + "-bit windows");
199          return;
200      }
201
202      Set<String> uniquePatterns = new HashSet<>();
203      int totalWindows = 0;
204
205      // Slide the 14-bit window through the sequence
206      for (int start = 0; start <= sequence.length() - windowHeight; start++) {
207          String window = sequence.substring(beginIndex: start, start + windowHeight);
208          uniquePatterns.add(window);
209          totalWindows++;
210      }
211      System.out.println("  • Total sliding windows: " + totalWindows);
212      System.out.println("  • Unique " + windowHeight + "-bit patterns: " + uniquePatterns.size());
213
214      // Simple assessment
215      double diversity = (uniquePatterns.size() * 100.0) / totalWindows;
216      if (diversity > 50) {
217          System.out.println("  ✓ Good pattern diversity (" + String.format(format: "%.1f", args:diversity) + "%)");
218      } else {
219          System.out.println("  ✗ Limited pattern diversity (" + String.format(format: "%.1f", args:diversity) + "%)");
220      }
221      // Show a few sample patterns
222      System.out.println("  • Sample patterns:");
223      int count = 0;
224      for (String pattern : uniquePatterns) {
225          if (count < 2) { // Show only 2 samples
226              System.out.println("    - " + pattern);
227              count++;
228          } else {
229              break;
230          }
231      }
232  }

```

Here is the output:

```

**** Simple Randomness check for LFSR****
BALANCE TEST:
• 1s: 62 (≈ 1, ≈ %)
• 0s: 66 (≈ 1, ≈ %)
✓ 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 (≈ 1, ≈ %)
• 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}; //initial seed
```

- o The sequence generated by your LFSR.

Full output sequence: 000001001011101110100110101101100011111001000110011100010100000100101110110100110101100011111001000110011100010100

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 (±1, ±1%)
• 0s: 66 (±1, ±1%)
 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 (±1, ±1%)
• 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.

Polynomial: $p(x) = x^6 + x + 1$

① First we must check if it irreducible

make $x=0$

$$= 0^6 + 0 + 1 = 0 + 0 + 1 = 1 \neq 0$$

make $x=1$

$$= 1^6 + 1 + 1 = 1 + 1 + 1 = 3 \neq 0$$

so there's not such root, and
the polynomial can't factored
into smaller pieces.

Max Length = $2^6 - 1 = 63$

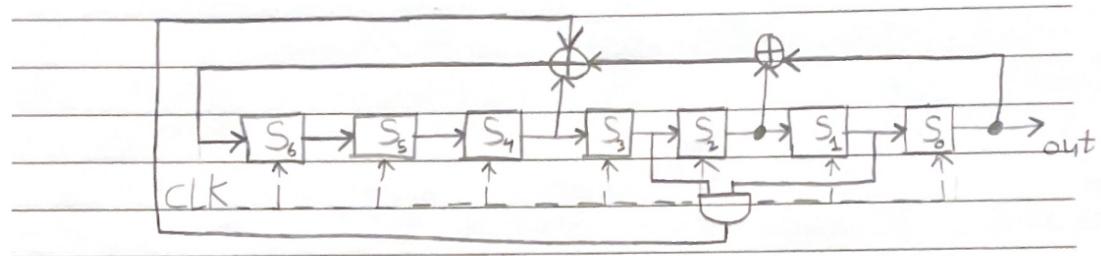
Non-linear Feedback/Feedforward Shift Register (NFSR) Design and Implementation

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

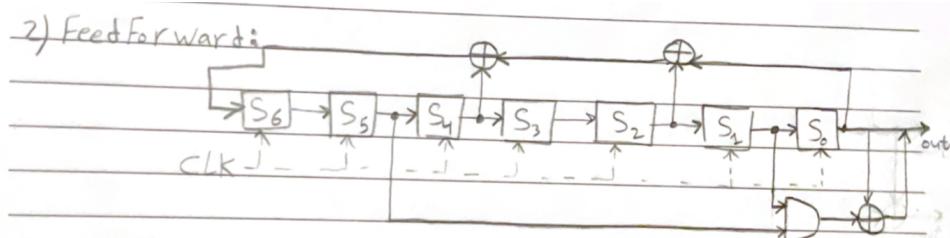
1. feedback loop only
2. feedforward loop only
3. both feedback and feedforward loops

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

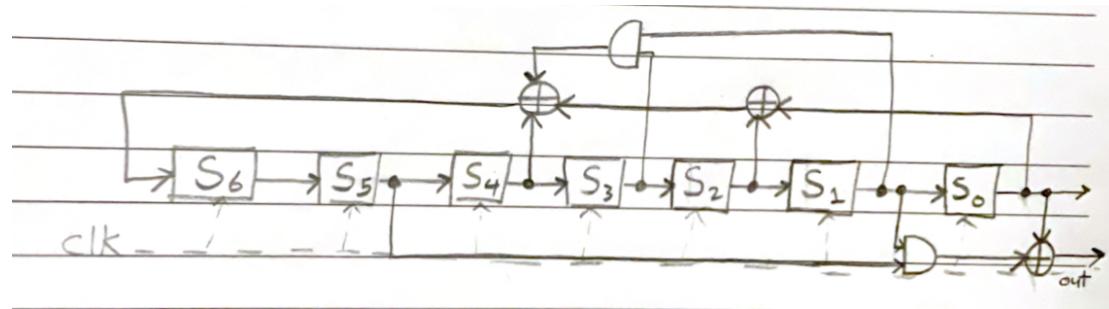
1) Feedback:



2) Feedforward:



3) both feedback and feedforward:



o Give the polynomials for each of the three cases

1) Feedback:

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

$$\text{Output Function: } O = S_0$$

2) Feedforward:

$$\text{Feedback Polynomial: } p(x) = x^7 + x^4 + x^2 + 1$$

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

3) both feedback and feedforward:

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

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

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

1) Feedback:

$$\begin{aligned} S_{6,i+1} &= S_{4,i} + S_{2,i} + S_{0,i} + (S_{1,i} \cdot S_{3,i}) \\ S_{\text{out},i} &= S_{0,i} \end{aligned}$$

2) Feedforward:

$$\begin{aligned} S_{6,i+1} &= S_{4,i} + S_{2,i} + S_{0,i} \\ S_{\text{out},i} &= S_{0,i} + (S_{2,i} \cdot S_{5,i}) \end{aligned}$$

3) both feedback and feedforward:

$$\begin{aligned} \text{Both:} \\ S_{6,i+1} &= S_{4,i} + S_{2,i} + S_{0,i} + (S_{1,i} \cdot S_{3,i}) \\ S_{\text{out},i} &= S_{0,i} + (S_{1,i} \cdot S_{5,i}) \end{aligned}$$

- 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 -
    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(: 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(: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(: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	$S_4 \oplus S_2 \oplus S_0$	S_0	127bits	63bits	31	32	49.2%:50.8%	Excellent
Feedback NLFSR	Nonlinear feedback with AND gate in feedback path	$S_4 \oplus S_2 \oplus S_0(S_1 \wedge S_3)$	S_0	127bits	98bits	49	49	50% : 50%	Perfect
Feedforward NLFSR	Nonlinear output with AND gate in output path	$S_4 \oplus S_2 \oplus S_0$	$S_0 \oplus (S_1 \wedge S_5)$	127 bits	63bits	32	31	50.8%:49.2%	Excellent
Both NLFSR	Combined nonlinear with AND gates in both paths	$S_4 \oplus S_2 \oplus S_0 \oplus (S_1 \wedge S_3)$	$\oplus (S_1 \wedge S_5)S_0$	127 bits	98bits	50	48	51% : 49%	Excellent

Appendix:

[1] Screen shot of the LFSR code:

```
1  /*  
2   Joori Ehab 2310656 section CY2  
3   */  
4   package lfsr;  
5   import java.util.*;  
6  
7   public class LFSR {  
8  
9       // ----- Parameters -----  
10      private static final int LENGTH = 7; //length of the polynomial , LFSR have 7FF(s6-s0)  
11      private static final String POLYNOMIAL = "x^7 + x^4 + x^2 + 1"; //our polynomial  
12      private static final String FeedbackPOLYNOMIAL = "x^7 = x^4 + x^2 + 1"; //our polynomial  
13      private static final int[] TAPS = {2, 4, 6}; // S4, S2, S0 positions of XOR  
14      private static final int[] INITIAL_SEED = {1, 0, 0, 0, 0, 0, 0}; //initial seed  
15  
16      public static void main(String[] args) throws InvalidSeedException, InvalidTapException {  
17  
18          // Step 1: Check if our parameters are valid before running with try-catch  
19          checkValidParameter();  
20  
21          //Separate logic from I/O , Step 2: Run the LFSR algorithm and get results  
22          LFSRResult result = runLFSR();  
23  
24          // Step 3: Show the results to the user  
25          displayResults(result);  
26      }  
27  
28      /**  
29       *           Implements LFSR algorithm  
30       *   This is the core function that runs the Linear Feedback Shift Register:  
31       */  
32      private static LFSRResult runLFSR() {  
33  
34          // create a copy of the initial seed so we don't modify the original one  
35          int[] register = INITIAL_SEED.clone();  
36  
37          //to store the output sequence (bits that come out)  
38          StringBuilder output = new StringBuilder();  
39  
40          int period = 0;// How many steps until repetition  
41          int maxPeriod = (int) Math.pow(a: 2, b: LENGTH) - 1; //2^7 - 1 = 127  
42  
43          // to display header informations  
44          System.out.println("LFSR Analysis Polynomial: " + POLYNOMIAL);  
45          System.out.println("Our feedback Polynomial: " + FeedbackPOLYNOMIAL);
```

```

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        // register[TAPS[0]] = S4, register[TAPS[1]] = S2, register[TAPS[2]] = S0
57        int feedback = register[TAPS[0]] ^ register[TAPS[1]] ^ register[TAPS[2]]; //XOR the tapped bits together
58
59        // The output bit is the rightmost bit (S0) before we shift
60        int outputBit = register[6];
61        output.append(l: outputBit); // Add to our output sequence
62
63        // // SHIFT operation:
64        int[] newRegister = new int[LENGTH]; // new empty register
65
66        newRegister[0] = feedback; // Put feedback bit(result of XOR) in leftmost position (S6)
67
68
69        // Copy all bits from old register, shifting them right by one position
70        // register[0] goes to newRegister[1], register[1] to newRegister[2], and so on..
71        System.arraycopy(src:register, srcPos: 0, dest:newRegister, destPos:1, LENGTH - 1);
72
73        // Update our register to the new state
74        register = newRegister;
75
76        // Display the current state
77        System.out.printf(format: "%-2d      %s      %d%n", args:clock, args:arrayToString(arr:register), args:outputBit);
78
79        // Period detection ,check if we've returned to the initial state
80        if (arrayEquals(a:register, b:INITIAL_SEED)) {
81            period = clock; // Found the period length
82            System.out.println("**** Period: " + period + " ****");
83        }
84    }
85
86
87    // Return all our results packaged together
88    return new LFSRResult(period, output:output.toString(), maxPeriod);
89
90}

```

```

91 /**
92 * This function checks if our configurations are valid before running
93 */
94 private static void checkValidParameter() throws InvalidSeedException, InvalidTapException {
95     try {
96         // Validate seed length must be exactly 7 bits
97         if (INITIAL_SEED.length != LENGTH) {
98             throw new InvalidSeedException("Seed length must be " + LENGTH);
99         }
100
101        // Validate seed is not all zeros , because all-zero state would cause problems
102        boolean allZero = true;
103        for (int bit : INITIAL_SEED) {
104            if (bit != 0) {
105                allZero = false; // Found a 1, so not all zeros
106                break;
107            }
108        }
109        if (allZero) {
110            throw new InvalidSeedException(message:"Seed cannot be all zeros");
111        }
112
113        // Validate tap array not empty , need at least one tap
114        if (TAPS.length == 0) {
115            throw new InvalidTapException(message:"Taps array cannot be empty");
116        }
117
118        System.out.println(x: " All parameters validated successfully ✅");
119
120    }
121    catch (InvalidSeedException | InvalidTapException e) {
122        throw e; // Re-throw specific exceptions to main method
123        catch (Exception e) {
124            throw new InvalidSeedException("Unexpected error during parameter validation: " + e.getMessage());
125        }
126    }
127
128 /**
129 *           Display results with analysis
130 *           Shows the final results to the user
131 */
132 private static void displayResults(LFSRResult result) {
133
134     System.out.println(x: "\n**** ANALYSIS RESULTS ****");
135     System.out.println("Period: " + result.period);
136     System.out.println("Max possible: " + result.maxPeriod);
137
138 }

```

```

135     System.out.println("Max possible: " + result.maxPeriod);
136     System.out.println("Output length: " + result.output.length());
137
138
139     //Check if we achieved maximum period (in our case no)
140     if (result.period == result.maxPeriod) {
141         System.out.println(": Maximum period achieved ✅");
142     }
143     else {
144         System.out.println(": Maximum period not achieved ❌");
145         System.out.println(": Reason: Polynomial is reducible and not primitive");
146     }
147
148     // to show the full sequence of bits
149     System.out.println("Full output sequence: " + result.output);
150
151     assessRandomness(sequence: result.output); // method to asses the randomness
152 }
153
154 /**
155 * Randomness assessment : Check the balance
156 */
157 private static void assessRandomness(String sequence) {
158     System.out.println(": \n**** Simple Randomness check for LFSR****");
159
160     int ones = 0; // to count 1s
161     for (char bit : sequence.toCharArray()) {
162         if (bit == '1') ones++;
163     }
164     int zeros = sequence.length() - ones; // to count 0s
165
166
167     // to calculate percentage :
168     double onesPercent = (ones * 100.0) / sequence.length();
169     double zerosPercent = (zeros * 100.0) / sequence.length();
170
171     // for printing
172     System.out.println(": BALANCE TEST:");
173     System.out.println("  • 1s: " + ones + " (" + String.format(format: "%.1f", args: onesPercent) + "%)");
174     System.out.println("  • 0s: " + zeros + " (" + String.format(format: "%.1f", args: zerosPercent) + "%)");
175
176
177     // if ratio is between 40-60%, it's good
178     if (ones >= sequence.length()*0.4 && ones <= sequence.length()*0.6) { // between 40-60%
179         System.out.println(": ✅ Good balance (close to 50/50)");
180     } else {
181         System.out.println(": ❌ Poor balance");
182     }
183
184     // to call the pattern test:
185     performPatternTest(sequence);
186 }
187 /**
188 * PATTERN TEST with sliding window of size 2n (14 bits)
189 * Add this method right after assessRandomness
190 */
191 private static void performPatternTest(String sequence) {
192     int n = LENGTH; // n = 7
193     int windowSize = 2 * n; // 2n = 14 bits
194
195     System.out.println("\nPATTERN TEST (sliding window: " + windowSize + " bits = 2 x " + n + ")");
196
197     // Check if sequence is long enough
198     if (sequence.length() < windowSize) {
199         System.out.println("  • Sequence too short for " + windowSize + "-bit windows");
200         return;
201     }
202
203     Set<String> uniquePatterns = new HashSet<>();
204     int totalWindows = 0;
205
206     // Slide the 14-bit window through the sequence
207     for (int start = 0; start <= sequence.length() - windowSize; start++) {
208         String window = sequence.substring(beginIndex: start, start + windowSize);
209         uniquePatterns.add(e: window);
210         totalWindows++;
211     }
212
213     System.out.println("  • Total sliding windows: " + totalWindows);
214     System.out.println("  • Unique " + windowSize + "-bit patterns: " + uniquePatterns.size());
215
216     // Simple assessment
217     double diversity = (uniquePatterns.size() * 100.0) / totalWindows;
218     if (diversity > 50) {
219         System.out.println("  ✅ Good pattern diversity (" + String.format(format: "%.1f", args: diversity) + "%)");
220     } else {
221         System.out.println("  ❌ Limited pattern diversity (" + String.format(format: "%.1f", args: diversity) + "%)");
222     }
223 }

```

```

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

266 // ----- CUSTOM EXCEPTION CLASSES -----
267 // These help us to handle different types of errors specifically
268
269 static class InvalidSeedException extends Exception {
270     public InvalidSeedException(String message) { super(message); }
271 }
272
273 static class InvalidTapException extends Exception {
274     public InvalidTapException(String message) { super(message); }
275 }
276
277 /**
278 *                         Result container class
279 *                         A simple class to package all our results together
280 */
281 static class LFSRResult {
282     int period; // How many steps until repetition
283     String output; // sequence of bits generated
284     int maxPeriod; // maximum period from (2^n - 1)
285
286     LFSRResult(int period, String output, int maxPeriod) {
287         this.period = period;
288         this.output = output;
289         this.maxPeriod = maxPeriod;
290     }
291 }
292
293 }
294

```

```
193     System.out.println("All validation tests passed ✅");
194
195 } catch (TestFailureException e) {
196     System.out.println("Validation test failed: " + e.getMessage());
197 } catch (Exception e) {
198     System.out.println("Unexpected error during validation: " + e.getMessage());
199 }
200
201 /**
202 * Test deterministic reproduction
203 */
204 private static void testDeterminism() throws TestFailureException {
205     try {
206         // Create two identical registers
207         int[] reg1 = INITIAL_SEED.clone();
208         int[] reg2 = INITIAL_SEED.clone();
209         StringBuilder out1 = new StringBuilder();
210         StringBuilder out2 = new StringBuilder();
211
212         // Run both for 10 steps
213         for (int i = 0; i < 10; i++) {
214             int fb1 = reg1[TAPS[0]] ^ reg1[TAPS[1]] ^ reg1[TAPS[2]]; // xor
215             out1.append(reg1[6]); // Store output
216             reg1 = shiftRight(register: reg1, feedback: fb1); // Shifting
217
218             // run the second LFSR
219             int fb2 = reg2[TAPS[0]] ^ reg2[TAPS[1]] ^ reg2[TAPS[2]]; // xor
220             out2.append(reg2[6]); // Store output
221             reg2 = shiftRight(register: reg2, feedback: fb2); // Shifting
222         }
223
224         //Compare outputs , they should be identical
225         if (!out1.toString().equals(anObject:out2.toString())) {
226             throw new TestFailureException(message:"LFSR is not deterministic");
227         }
228         System.out.println("Deterministic reproduction test passed ✅");
229
230     } catch (Exception e) {
231         throw new TestFailureException("Determinism test error: " + e.getMessage());
232     }
233 }
234 }
235 */
236
```

```
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}, // Our main seed
246                 {0, 0, 0, 0, 0, 1}, // diff seed
247                 {1, 0, 1, 0, 1, 0} // 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(" ✅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("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             //Check if we achieved maximum period (in our case no)
321             if (result.period == result.maxPeriod) {
322                 r
```

```

322     if (result.period == result.maxPeriod) {
323         System.out.println("Maximum period achieved ✓");
324     }
325     else {
326         System.out.println("Maximum period not achieved ✗");
327         System.out.println("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 used 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(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(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 *          Result container class
413 *          A simple class to package all our results together
414 */
415 static class LFSRResult {
416     int period; // How many steps until repetition
417     String output; // sequence of bits generated
418     int maxPeriod; // maximum period from (2^n - 1)
419
420     LFSRResult(int period, String output, int maxPeriod) {
421         this.period = period;
422         this.output = output;
423         this.maxPeriod = maxPeriod;
424     }
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
.....1.	
.1.	1	
.1..	2	
.	...1..1..	3	
.	..1...1..1.	4	
.	.1...1..1..1	5	
.	1...1..1..1..1	6	
1	...1..1..1..1..1	7	
.	..1..1..1..1..1..	8	
.	.1..1..1..1..1..1	9	
1	..1..1..1..1..1..1	10	
.	1..1..1..1..1..1..1	11	
1	..1..1..1..1..1..1..1	12	
1	1..1..1..1..1..1..1..1	13	
1	..1..1..1..1..1..1..1..1	14	
.	1..1..1..1..1..1..1..1..1	15	
1	..1..1..1..1..1..1..1..1..1	16	
1	1..1..1..1..1..1..1..1..1..1	17	
1	..1..1..1..1..1..1..1..1..1..1	18	
1	1..1..1..1..1..1..1..1..1..1..1	19	
.	..1..1..1..1..1..1..1..1..1..1..1..1	20	

1	...1110110	21
.	..11101101	22
.	111011010	23
1	101101101	24
1	..110110111	25
.	101101110	26
1	..11011101	27
.	101110111	28
1	..11101110	29
.	111011100	30
1	10111000	31
1	..1110000	32
.	11100001	33
1	100000111	34
1	...0001111	35
.	...00011111	36
.	...00111111	37
.	..11111110	38
.	111111100	39
1	11111001	40
1	11110010	41
1	11100100	42
1	10010000	43
1	..0010001	44
.	..10000111	45
.	10000110	46
1	..00011100	47
.	..00111001	48
.	..1100111	49

. 11001111 50
1 10011110 51
1 00111100 52
. 01111000 53
. 1110001 54
1 11000100 55
1 1000101 56
1 0001010 57
. 00101000 58
. 01010000 59
. 1010000 60
1 0100000 61
. 1000000 62
1 000001 63

**** Period: 63 ****

. 0000010 64
. 00000100 65
. 00001001 66
. 0010010 67
. 0100101 68
. 1001011 69
1 0010111 70
. 0101110 71
. 10111101 72
1 01111011 73
. 11101111 74
1 11011111 75
1 10111110 76
1 01111101 77

.	1111010	78
1	1110100	79
1	1101001	80
1	1010011	81
1	0100110	82
.	1001101	83
1	0011010	84
.	0110101	85
.	1101010	86
1	1010101	87
1	0101011	88
.	1010110	89
1	0101101	90
.	1011011	91
1	0110110	92
.	1101100	93
1	1011000	94
1	0110000	95
.	1100001	96
1	1000011	97
1	0000111	98
.	0001111	99
.	0011111	100
.	0111110	101
.	1111110	102
1	1111101	103
1	1110010	104
1	1100100	105
1	1001000	106

1 . . 1 . . 1 107
· . 1 . . . 1 1 108
· 1 . . . 1 1 1 109
1 . . . 1 1 1 110
· . . 1 1 1 . 1 111
· . . 1 1 . . 1 112
· 1 1 . . . 1 1 113
1 1 . . 1 1 1 114
1 . . . 1 1 1 1 115
· . . 1 1 1 . . 116
· 1 1 1 . . . 1 117
1 1 1 1 118
1 1 . . . 1 . 1 119
1 . . . 1 . 1 1 120
· . . 1 . 1 . 1 121
· . . 1 . 1 . . 122
· 1 . 1 123
1 . 1 124
· 1 125
1 1 126

**** Period: 126 ****

· 1 . 127
· 1 . . 128

**** ANALYSIS RESULTS ****

Period: 126

Max possible: 127

Output length: 128

✗ Maximum period not achieved

Reason: Polynomial is reducible and not primitive

Full output sequence:

000000100101101110100110101101100001111001000110011100010100000010
01011101110100110101101100001111001000110011100010100

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

:BALANCE TEST

(%ξ^,ξ) s: 621 •

(%ο¹,¹) s: 66• •

(Good balance (close to 50/50 ✓)

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

Total sliding windows: 115 •

Unique 14-bit patterns: 63 •

(%οξ,ξ) Good pattern diversity ✓

:Sample patterns •

1...11...1...111...1 -

1...1.....1...1...1 -

(BUILD SUCCESSFUL (total time: 0 seconds

[3] Screen shot of the NLFSR Feedback:

```
1  /*
2  * Joori Ehab 2310656 section CY2
3  */
4  package lfsr;
5
6  import java.util.HashSet;
7  import java.util.Set;
8
9  public class NLFSRFeedback {
10
11     // ----- Parameters -----
12     private static final int LENGTH = 7; //length of the polynomial , LFSR have 7FF(s6-s0
13     private static final String POLYNOMIAL = "x^7 + x^4 + x^2 + 1"; //our polynomial
14     private static final int[] TAPS = {2, 4, 6}; // S4, S2, S0 positions of XOR
15     private static final int[] AND_TAPS = {3, 5}; // taps for AND operation ←
16     private static final int[] INITIAL_SEED = {1, 0, 0, 0, 0, 0, 0}; //initial seed
17
18     public static void main(String[] args) throws InvalidSeedException, InvalidTapException {
19
20         // Step 1: Check if our parameters are valid before running with try-catch
21         checkValidParameter();
22
23         //Separate logic from I/O , Step 2: Run the LFSR algorithm and get results
24         NLFSRBack_Result result = runNLFSR_Back();
25
26         // Step 3: Show the results to the user
27         displayResults(result);
28     }
29
30     /**
31      *          Implements LFSR algorithm
32      * This is the core function that runs the Linear Feedback Shift Register:
33      */
34     private static NLFSRBack_Result runNLFSR_Back() {
35
36         // create a copy of the initial seed so we don't modify the original one
37         int[] register = INITIAL_SEED.clone();
38
39         //to store the output sequence (bits that come out)
40         StringBuilder output = new StringBuilder();
41
42         int period = 0;// How many steps until repetition
43         int maxPeriod = (int)Math.pow(2, b: LENGTH) - 1; //2^7 - 1 = 127
44
45         // to display header informations
```

```

45     // to display header informations
46     System.out.println("LFSR Analysis Polynomial: " + POLYNOMIAL);
47     System.out.println("Max possible period: " + maxPeriod);
48     System.out.println(x: "\nClk      S6 S5 S4 S3 S2 S1 S0      Output");
49     System.out.println(x: "-----");
50
51     //// Show initial state (step 0) Clk 0
52     System.out.printf(format: "0      %s      %n", args:arrayToString(arr:register));
53
54     // Main LFSR loop, run to 128 cycles (more than max possible)
55     for (int clock = 1; clock <= 128; clock++) {
56
57         int andResult = register[AND_TAPS[0]] & register[AND_TAPS[1]]; // to calculate the AND =
58
59         // register[TAPS[0]] = S4, register[TAPS[1]] = S2, register[TAPS[2]] = S0
60         int feedback = register[TAPS[0]] ^ register[TAPS[1]] ^ register[TAPS[2]] ^ andResult; // XOR with AND =
61
62         // The output bit is the rightmost bit (S0) before we shift
63         int outputBit = register[6];
64         output.append(l: outputBit); // Add to our output sequence
65
66         // // SHIFT operation:
67         int[] newRegister = new int[LENGTH]; // new empty register
68
69         newRegister[0] = feedback; // Put feedback bit(result of XOR) in leftmost position (S6)
70
71
72         // Copy all bits from old register, shifting them right by one position
73         // register[0] goes to newRegister[1], register[1] to newRegister[2], and so on..
74         System.arraycopy(src:register, srcPos: 0, dest:newRegister, destPos:1, LENGTH - 1);
75
76         // Update our register to the new state
77         register = newRegister;
78
79         // Display the current state
80         System.out.printf(format: "%-2d      %s      %d%n", args:clock, args:arrayToString(arr:register), args:outputBit);
81
82         // Period detection ,check if we've returned to the initial state
83         if (arrayEquals(a: register, b: INITIAL_SEED)) {
84             period = clock; // found the period length
85             System.out.println("**** Period: " + period + " ****");
86         }
87
88     } // Return all our results packaged together
89     return new NLFSRBack_Result(period, output: output.toString(), maxPeriod);
90
91 }
92 /**
93 * This function checks if our configurations are valid before running
94 */
95 private static void checkValidParameter() throws InvalidSeedException, InvalidTapException {
96     try {
97
98         // Validate seed length must be exactly 7 bits
99         if (INITIAL_SEED.length != LENGTH) {
100             throw new InvalidSeedException("Seed length must be " + LENGTH);
101
102         // Validate seed is not all zeros , because all-zero state would cause problems
103         boolean allZero = true;
104         for (int bit : INITIAL_SEED) {
105             if (bit != 0) {
106                 allZero = false; // Found a 1, so not all zeros
107                 break;
108             }
109         }
110         if (allZero) {
111             throw new InvalidSeedException(message:"Seed cannot be all zeros");
112         }
113
114         // Validate tap array not empty , need at least one tap
115         if (TAPS.length == 0) {
116             throw new InvalidTapException(message:"Taps array cannot be empty");
117         }
118
119         System.out.println(x: " All parameters validated successfully ✅");
120
121     } catch (InvalidSeedException | InvalidTapException e) {
122         throw e; // Re-throw specific exceptions to main method
123     } catch (Exception e) {
124         throw new InvalidSeedException("Unexpected error during parameter validation: " + e.getMessage());
125     }
126
127 }
128
129 /**
130 *          *
131 *          * Shows the final results to the user
132 */

```

```

133     private static void displayResults(NLFSRBack_Result result) {
134
135         System.out.println("\n**** ANALYSIS RESULTS ****");
136         System.out.println("Period: " + result.period);
137         System.out.println("Max possible: " + result.maxPeriod);
138         System.out.println("Output length: " + result.output.length());
139
140
141         //Check if we achieved maximum period (in our case no)
142         if (result.period == result.maxPeriod) {
143             System.out.println("Maximum period achieved ✓");
144         }
145         else {
146             System.out.println("Maximum period not achieved ✗");
147             System.out.println("Reason: Polynomial is reducible and not primitive");
148         }
149
150         // to show the full sequence of bits
151         System.out.println("Full output sequence: " + result.output);
152
153         assessRandomness(sequence: result.output); // method to asses the randomness
154
155
156     /*
157      * Randomness assessment : Check the balance
158     */
159     private static void assessRandomness(String sequence) {
160         System.out.println("\n**** Simple Randomness check for feedback****");
161
162         int ones = 0; // to count 1s
163         for (char bit : sequence.toCharArray()) {
164             if (bit == '1') ones++;
165         }
166         int zeros = sequence.length() - ones; // to count 0s
167
168
169         // to calculate percentage :
170         double onesPercent = (ones * 100.0) / sequence.length();
171         double zerosPercent = (zeros * 100.0) / sequence.length();
172
173         //      for printing
174         System.out.println("BALANCE TEST:");
175         System.out.println("  • 1s: " + ones + " (" + String.format(format: "%.1f", args: onesPercent) + "%)");
176         System.out.println("  • 0s: " + zeros + " (" + String.format(format: "%.1f", args: zerosPercent) + "%)");
177
178
179         // if ratio is between 40-60%, it's good
180         if (ones >= sequence.length()*0.4 && ones <= sequence.length()*0.6) { // between 40-60%
181             System.out.println("  ✓ Good balance (close to 50/50)");
182         } else {
183             System.out.println("  ✗ Poor balance");
184         }
185         // to call the pattern test:
186         performPatternTest(sequence);
187     }
188
189     /**
190      * PATTERN TEST with sliding window of size 2n (14 bits)
191      * Add this method right after assessRandomness
192     */
193     private static void performPatternTest(String sequence) {
194
195         int n = LENGTH; // n = 7
196         int windowSize = 2 * n; // 2n = 14 bits
197
198         System.out.println("\nPATTERN TEST (sliding window: " + windowSize + " bits = 2 x " + n + ")");
199
200         // Check if sequence is long enough
201         if (sequence.length() < windowSize) {
202             System.out.println("  • Sequence too short for " + windowSize + "-bit windows");
203             return;
204         }
205
206         Set<String> uniquePatterns = new HashSet<>();
207         int totalWindows = 0;
208
209         // Slide the 14-bit window through the sequence
210         for (int start = 0; start <= sequence.length() - windowSize; start++) {
211             String window = sequence.substring(beginIndex: start, start + windowSize);
212             uniquePatterns.add(e: window);
213             totalWindows++;
214         }
215
216         System.out.println("  • Total sliding windows: " + totalWindows);
217         System.out.println("  • Unique " + windowSize + "-bit patterns: " + uniquePatterns.size());
218
219         // Simple assessment
220         double diversity = (uniquePatterns.size() * 100.0) / totalWindows;
221         if (diversity > 50) {
222             System.out.println("  ✓ Good pattern diversity (" + String.format(format: "%.1f", args: diversity) + "%)");
223         } else {
224             System.out.println("  ✗ Limited pattern diversity (" + String.format(format: "%.1f", args: diversity) + "%)");
225         }
226

```

```

225     // Show a few sample patterns
226     System.out.println("  • 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
// ----- In this section I used methods to help me -----
239
240 /*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" */
241
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 }

```

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

[4] The output of the NLSR 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
.....
.	1
.	0	2
.	1	0	3
.	1	0	1	4
.	.	.	.	1	0	0	1	5
.	.	.	1	0	0	1	0	6
.	1	0	0	1	0	1	1	7
1	0	0	1	0	1	1	1	8
1	0	0	1	1	0	1	0	9
1	0	0	1	1	1	0	1	10
1	0	0	1	1	1	1	0	11
1	0	0	1	1	0	0	1	12
1	0	0	1	0	0	1	1	13
1	0	0	1	0	0	0	1	14
1	0	0	1	0	0	1	1	15
1	0	0	1	0	1	0	1	16
1	0	0	1	1	0	0	1	17
1	0	0	1	1	0	1	1	18
1	0	0	1	1	1	0	1	19
1	0	0	1	1	1	1	0	20

1	1	0	0	1	1	1	0	21
1	1	0	0	1	1	1	0	22
.	1	0	1	1	1	0	0	23
.	1	1	1	0	0	0	0	24
1	1	1	0	0	0	0	0	25
1	1	0	0	0	0	1	0	26
1	0	0	0	0	0	1	1	27
.	0	0	0	0	1	1	0	28
.	0	0	0	1	1	0	1	29
.	0	0	1	1	1	0	1	30
.	0	1	1	0	1	1	1	31
.	1	1	0	1	1	1	0	32
1	1	0	1	1	1	0	1	33
1	0	1	1	1	0	1	1	34
.	1	1	1	0	1	1	0	35
1	1	1	0	1	1	0	1	36
1	1	0	1	1	0	1	1	37
1	0	1	1	0	1	1	0	38
.	1	1	0	1	1	0	0	39
1	1	0	1	1	0	0	1	40
1	0	1	1	0	0	1	0	41
.	1	1	0	0	1	0	1	42
1	1	0	0	1	0	1	0	43
1	0	0	1	0	1	0	1	44
.	0	1	0	1	0	1	0	45
.	1	0	1	0	1	0	1	46
1	0	1	0	1	1	1	0	47
.	1	0	1	0	1	1	1	48

1	0	1	0	1	1	1	1	1	49
0	1	0	1	1	1	1	1	1	50
1	0	1	1	1	1	1	0	1	51
0	1	1	1	1	1	0	1	1	52
1	1	1	1	0	1	1	1	1	53
1	1	1	1	0	1	1	1	1	54
1	1	0	1	1	1	1	1	1	55
1	0	1	1	1	1	1	1	1	56
0	1	1	1	1	1	1	1	1	57
1	1	1	1	1	1	1	0	1	58
1	1	1	1	1	1	1	0	0	59
1	1	1	1	1	1	0	0	0	60
1	1	1	1	0	0	0	1	1	61
1	1	1	0	0	0	1	0	1	62
1	1	0	0	0	1	0	1	1	63
1	0	0	0	1	0	1	0	1	64
0	0	0	1	0	1	0	0	1	65
0	0	1	0	0	0	1	0	0	66
0	1	0	1	0	0	0	1	1	67
1	0	1	0	0	0	1	0	1	68
0	1	0	0	0	1	0	0	1	69
1	0	0	0	1	0	0	0	1	70
0	0	0	1	0	0	0	0	1	71
0	0	1	0	0	0	0	1	1	72
0	1	0	0	0	0	1	0	1	73
1	0	0	0	0	1	0	1	1	74
0	0	0	1	0	1	1	1	1	75
0	0	1	0	1	1	0	1	1	76

. . 1 0 1 1 1 0 0 77
· 1 0 1 1 1 0 0 78
1 · 1 1 1 0 0 0 79
. 1 1 1 0 0 0 1 80
1 1 0 0 0 0 1 1 81
1 0 0 0 0 1 1 1 82
. 0 0 0 1 1 1 1 83
. 0 0 0 1 1 1 1 1 84
. 0 1 1 1 1 1 1 0 85
. 1 1 1 1 1 1 1 1 86
1 1 1 1 1 1 0 1 87
1 1 1 1 1 0 1 1 88
1 1 1 1 0 1 1 1 89
1 1 0 1 0 1 1 1 90
1 0 1 0 1 1 1 1 91
. 1 0 1 1 0 1 1 92
1 0 1 1 0 1 0 93
. 1 1 0 1 0 0 94
1 1 0 1 0 0 0 95
1 0 1 0 0 0 96
. 1 0 0 0 0 97
1 0 0 0 0 0 1 98

**** Period: 98 ****

. 0 0 0 0 0 1 0 99
. 0 0 0 0 0 1 0 0 100
. 0 0 0 1 0 0 1 101
. 0 0 1 0 0 1 0 102
. 0 1 0 0 1 0 1 103

.	1	0	0	1	0	1	1	1	1	0	1	0	4
1	1	0	0	1	0	1	1	1	1	1	1	1	5
.	1	0	1	0	1	1	1	1	0	0	1	0	6
.	1	0	1	1	1	1	1	0	0	0	1	0	7
1	0	1	1	1	1	1	0	0	0	1	0	8	
.	1	1	1	1	0	0	1	0	0	1	0	9	
1	1	1	0	0	1	0	0	0	1	1	0	10	
1	1	0	0	1	0	0	0	0	1	1	1	11	
1	0	0	1	0	0	0	1	0	0	1	1	12	
.	1	0	0	0	0	1	1	1	1	1	1	13	
.	1	0	0	0	1	1	1	1	0	0	1	14	
1	0	0	0	1	1	0	0	1	1	0	1	15	
.	1	0	0	1	1	0	0	1	1	0	1	16	
.	1	0	1	1	0	0	1	1	1	0	1	17	
.	1	1	0	0	1	1	1	1	1	1	1	18	
1	1	0	0	1	1	1	1	1	0	0	1	19	
1	0	0	1	1	1	1	1	0	0	0	1	20	
.	1	0	1	1	1	0	0	0	1	1	1	21	
.	1	1	1	0	0	0	0	0	1	1	2	22	
1	1	1	0	0	0	0	0	0	1	1	2	23	
1	1	0	0	0	0	0	1	1	1	1	1	24	
1	0	0	0	0	0	1	1	1	1	1	1	25	
.	1	0	0	0	0	1	1	1	1	0	0	26	
.	1	0	0	0	1	1	1	1	0	1	1	27	
.	1	0	0	1	1	0	1	1	1	1	1	28	

**** ANALYSIS RESULTS ****

Period: 98

Max possible: 127

Output length: 128

✗ Maximum period not achieved

Reason: Polynomial is reducible and not primitive

Full output sequence:

000000100101110010001100111000001101110110110010101011110111111000
101000100001011000011110101101000000100101110010001100111000

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

:BALANCE TEST

(%॒०,१) s: 60 •

(%॒०३,१) s: 68 •

(Good balance (close to 50/50 ✓)

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

Total sliding windows: 115 •

Unique 14-bit patterns: 98 •

(%॒००,२) Good pattern diversity ✓

:Sample patterns •

..1.1.1.111111 -

1....11111111 -

(BUILD SUCCESSFUL (total time: 0 seconds

[5] Screen shot of the NLFSR Feedforward:

```
1  /*
2  * Joori Ehab 2310656 section CY2
3  */
4  package lfsr;
5
6  import java.util.HashSet;
7  import java.util.Set;
8
9  public class NLFSRFeedForward {
10     // ----- Parameters -----
11     private static final int LENGTH = 7; //length of the polynomial , LFSR have 7FF(s6-s0)
12     private static final String POLYNOMIAL = "x^7 + x^4 + x^2 + 1"; //our polynomial
13     private static final int[] TAPS = {2, 4, 6}; // S4, S2, S0 positions of XOR
14     private static final int[] AND_TAPS = {1, 5}; // taps for AND operation =
15     private static final int[] INITIAL_SEED = {1, 0, 0, 0, 0, 0, 0}; //initial seed
16
17     public static void main(String[] args) throws InvalidSeedException, InvalidTapException {
18
19         // Step 1: Check if our parameters are valid before running with try-catch
20         checkValidParameter();
21
22         //Separate logic from I/O , Step 2: Run the LFSR algorithm and get results
23         NLFSRForward_Result result = runNLFSR_Forward();
24
25         // Step 3: Show the results to the user
26         displayResults(result);
27     }
28
29     /**
30      *          Implements LFSR algorithm
31      *  This is the core function that runs the Linear Feedback Shift Register:
32     */
33     private static NLFSRForward_Result runNLFSR_Forward() {
34
35         // create a copy of the initial seed so we don't modify the original one
36         int[] register = INITIAL_SEED.clone();
37
38         //to store the output sequence (bits that come out)
39         StringBuilder output = new StringBuilder();
40
41         int period = 0;// How many steps until repetition
42         int maxPeriod = (int) Math.pow(2, b: LENGTH) - 1; //2^7 - 1 = 127
43
44         // to display header informations
45         System.out.println("LFSR Analysis Polynomial: " + POLYNOMIAL);
46         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 |
89 |     // Return all our results packaged together
90 |     return new NLFSRForward_Result(period, output: output.toString(), maxPeriod);
91 |

```

```

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 |
125 |     catch (InvalidSeedException | InvalidTapException e) {
126 |         throw e; // Re-throw specific exceptions to main method
127 |     }
128 |     catch (Exception e) {
129 |         throw new InvalidSeedException("Unexpected error during parameter validation: " + e.getMessage());
130 |     }
131 |
132 |     /**
133 |      * Shows the final results to the user
134 |     */

```

```

135     private static void displayResults(NLFSRForward_Result result) {
136
137         System.out.println("\n**** ANALYSIS RESULTS ****");
138         System.out.println("Period: " + result.period);
139         System.out.println("Max possible: " + result.maxPeriod);
140         System.out.println("Output length: " + result.output.length());
141
142
143         //Check if we achieved maximum period (in our case no)
144         if (result.period == result.maxPeriod) {
145             System.out.println("Maximum period achieved ✅");
146         }
147         else {
148             System.out.println("Maximum period not achieved ❌");
149             System.out.println("Reason: Polynomial is reducible and not primitive");
150         }
151
152         // to show the full sequence of bits
153         System.out.println("Full output sequence: " + result.output);
154
155         assessRandomness(sequence: result.output); // method to asses the randomness
156     }
157
158     /*
159      * Randomness assessment : Check the balance
160     */
161     private static void assessRandomness(String sequence) {
162         System.out.println("\n**** Simple Randomness check for feedforward****");
163
164         int ones = 0; // to count 1s
165         for (char bit : sequence.toCharArray()) {
166             if (bit == '1') ones++; // if the bit equals 1 , plass the ones
167         }
168         int zeros = sequence.length() - ones; // to count 0s
169
170
171         // to calculate percentage :
172         double onesPercent = (ones * 100.0) / sequence.length();
173         double zerosPercent = (zeros * 100.0) / sequence.length();
174
175         //      for printing
176         System.out.println("BALANCE TEST:");
177         System.out.println("  • 1s: " + ones + " (" + String.format(format: "%.1f", args:onesPercent) + "%)");
178         System.out.println("  • 0s: " + zeros + " (" + String.format(format: "%.1f", args:zerosPercent) + "%)");
179     }
180
181     // if ratio is between 40-60%, it's good |
182     if (ones >= sequence.length()*0.4 && ones <= sequence.length()*0.6) { // between 40-60%
183         System.out.println("  ✅ Good balance (close to 50/50)");
184     } else {
185         System.out.println("  ❌ Poor balance");
186     }
187
188     // to call the pattern test:
189     performPatternTest(sequence);
190 }
191 /**
192  * PATTERN TEST with sliding window of size 2n (14 bits)
193  * Add this method right after assessRandomness
194 */
195     private static void performPatternTest(String sequence) {
196         int n = LENGTH; // n = 7
197         int windowSize = 2 * n; // 2n = 14 bits
198
199         System.out.println("\nPATTERN TEST (sliding window: " + windowSize + " bits = 2 x " + n + ")");
200
201         // Check if sequence is long enough
202         if (sequence.length() < windowSize) {
203             System.out.println("  • Sequence too short for " + windowSize + "-bit windows");
204             return;
205         }
206
207         Set<String> uniquePatterns = new HashSet<>();
208         int totalWindows = 0;
209
210         // Slide the 14-bit window through the sequence
211         for (int start = 0; start <= sequence.length() - windowSize; start++) {
212             String window = sequence.substring(beginIndex: start, start + windowSize);
213             uniquePatterns.add(e: window);
214             totalWindows++;
215         }
216
217         System.out.println("  • Total sliding windows: " + totalWindows);
218         System.out.println("  • Unique " + windowSize + "-bit patterns: " + uniquePatterns.size());
219
220         // Simple assessment
221         double diversity = (uniquePatterns.size() * 100.0) / totalWindows;
222         if (diversity > 50) {
223             System.out.println("  ✅ Good pattern diversity (" + String.format(format: "%.1f", args:diversity) + "%)");
224         } else {
225             System.out.println("  ❌ Limited pattern diversity (" + String.format(format: "%.1f", args:diversity) + "%)");
226         }
227     }

```

```

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 used 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] = "1 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 // ----- CUSTOM EXCEPTION CLASSES -----  
273 // These help us to handle different types of errors specifically  
274  
275 static class InvalidSeedException extends Exception {  
276     public InvalidSeedException(String message) { super(message); }  
277 }  
278  
279 static class InvalidTapException extends Exception {  
280     public InvalidTapException(String message) { super(message); }  
281 }  
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	
1	0	1	0	0	0	0	0	.
2	0	0	1	0	0	0	0	.
3	1	0	0	1	0	0	0	.
4	0	1	0	0	1	0	0	.
5	1	0	1	0	0	1	0	.
6	1	1	0	1	0	0	1	.
7	1	1	1	0	1	0	0	.
8	0	1	1	1	0	1	0	.
9	1	0	1	1	1	0	1	.
10	1	1	0	1	1	1	0	.
11	1	1	1	0	1	1	1	.
12	1	1	1	1	0	1	1	.
13	0	1	1	1	1	0	1	.
14	1	0	1	1	1	1	0	.
15	0	1	0	1	1	1	1	.
16	0	0	1	0	1	1	1	.

۱۷ ۱۰۰۱۰۱۱ ۱
۱۸ ۱۱۰۰۱۰۱ ۱
۱۹ ۰۱۱۰۰۱۰ ۱
۲۰ ۱۰۱۱۰۰۱ ۱
۲۱ ۰۱۰۱۱۰۰ ۱
۲۲ ۱۰۱۰۱۱۰ .
۲۳ ۰۱۰۱۰۱۱ .
۲۴ ۱۰۱۰۱۰۱ .
۲۵ ۱۱۰۱۰۱۰ ۱
۲۶ ۰۱۱۰۱۰۱ ۱
۲۷ ۱۰۱۱۰۱۰ ۱
۲۸ ۱۱۰۱۱۰۱ .
۲۹ ۰۱۱۰۱۱۰ ۱
۳۰ ۰۰۱۱۰۱۱ ۱
۳۱ ۰۰۰۱۱۰۱ ۱
۳۲ ۰۰۰۰۱۱۰ ۱
۳۳ ۱۰۰۰۰۱۱ .
۳۴ ۱۱۰۰۰۰۱ ۱
۳۵ ۱۱۱۰۰۰۰ ۱
۳۶ ۱۱۱۱۰۰۰ .
۳۷ ۱۱۱۱۱۰۰ .
۳۸ ۰۱۱۱۱۱۰ .
۳۹ ۰۰۱۱۱۱۱ ۱
۴۰ ۱۰۰۱۱۱۱ ۱
۴۱ ۰۱۰۰۱۱۱ ۱

42 0010011 .
43 0001001 1
44 1000100 1
45 1100010 .
46 0110001 1
47 0011000 1
48 1001100 .
49 1100110 .
50 1110011 1
51 0111001 .
52 0011100 1
53 0001110 .
54 1000111 .
55 0100011 1
56 1010001 .
57 0101000 1
58 0010100 .
59 0001010 .
60 0000101 .
61 0000010 1
62 0000001 .
63 1000000 1

**** Period: 63 ****

64 0100000 .
65 0010000 .

۶۶ ۱۰۰۱۰۰۰ .
۶۷ ۰۱۰۰۱۰۰ .
۶۸ ۱۰۱۰۰۱۰ .
۶۹ ۱۱۰۱۰۰۱ .
۷۰ ۱۱۱۰۱۰۰ ۱
۷۱ ۰۱۱۱۰۱۰ .
۷۲ ۱۰۱۱۱۰۱ ۱
۷۳ ۱۱۰۱۱۱۰ ۱
۷۴ ۱۱۱۰۱۱۱ ۱
۷۵ ۱۱۱۱۰۱۱ .
۷۶ ۰۱۱۱۱۰۱ .
۷۷ ۱۰۱۱۱۱۰ ۱
۷۸ ۰۱۰۱۱۱۱ .
۷۹ ۰۰۱۰۱۱۱ .
۸۰ ۱۰۰۱۰۱۱ ۱
۸۱ ۱۱۰۰۱۰۱ ۱
۸۲ ۰۱۱۰۰۱۰ ۱
۸۳ ۱۰۱۱۰۰۱ ۱
۸۴ ۰۱۰۱۱۰۰ ۱
۸۵ ۱۰۱۰۱۱۰ .
۸۶ ۰۱۰۱۰۱۱ .
۸۷ ۱۰۱۰۱۰۱ .
۸۸ ۱۱۰۱۰۱۰ ۱
۸۹ ۰۱۱۰۱۰۱ ۱
۹۰ ۱۰۱۱۰۱۰ ۱

91 1101101 .
92 0110110 1
93 0011011 1
94 0001101 1
95 0000110 1
96 1000011 .
97 1100001 1
98 1110000 1
99 1111000 .
100 1111100 .
101 0111110 .
102 0011111 1
103 1001111 1
104 0100111 1
105 0010011 .
106 0001001 1
107 1000100 1
108 1100010 .
109 0110001 1
110 0011000 1
111 1001100 .
112 1100110 .
113 1110011 1
114 0111001 .
115 0011100 1

116 0001110 ·
117 1000111 ·
118 0100011 ·
119 1010001 ·
120 0101000 ·
121 0010100 ·
122 0001010 ·
123 0000101 ·
124 0000010 ·
125 0000001 ·
126 1000000 ·

**** Period: 126 ****

127 0100000 ·
128 0010000 ·

**** ANALYSIS RESULTS ****

Period: 126

Max possible: 127

Output length: 128

Maximum period not achieved **X**

Reason: Polynomial is reducible and not primitive

Full output sequence:

000000101110010011110001110111011000111011011001010010100010
100000010111001001111000111011101100011101101100101001010001
0100

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

BALANCE TEST:

- 1s: 64 ($\circ\cdot,\cdot\%$)
 - 0s: 64 ($\circ\cdot,\cdot\%$)
- 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 ($\circ\cdot,\cdot\%$)
- Sample patterns:
 - 11111000111011
 - 0100111100011

BUILD SUCCESSFUL (total time: 0 seconds)

[7] Screen shot of the Both NLSFR :

```

1  /*
2  Joori Ehab 2310656 section CY2
3  */
4  package lfsr;
5
6  import java.util.HashSet;
7  import java.util.Set;
8
9  public class NLFSRBoth {
10     // ----- Parameters -----
11     private static final int LENGTH = 7; //length of the polynomial , LFSR have 7FF(s6-s0
12     private static final String POLYNOMIAL = "x^7 + x^4 + x^2 + 1"; //our polynomial
13     private static final int[] TAPS = {2, 4, 6}; // S4, S2, S0 positions of XOR
14     private static final int[] AND_TAPS_FEEDBACK = {3, 5}; // NEW: AND taps for feedback (positions 3 and 5) ~
15     private static final int[] AND_TAPS_OUTPUT = {1, 5}; // AND taps for output (positions 1 and 5) ~
16     private static final int[] INITIAL_SEED = {1, 0, 0, 0, 0, 0, 0}; //initial seed
17
18     public static void main(String[] args) throws InvalidSeedException, InvalidTapException {
19
20         // Step 1: Check if our parameters are valid before running with try-catch
21         checkValidParameter();
22
23         //Separate logic from I/O , Step 2: Run the LFSR algorithm and get results
24         NLFSRBoth_Result result = runNLFSR_Both();
25
26         // Step 3: Show the results to the user
27         displayResults(result);
28     }
29
30     /**
31      *          Implements LFSR algorithm
32      *  This is the core function that runs the Linear Feedback Shift Register:
33      */
34     private static NLFSRBoth_Result runNLFSR_Both() {
35
36         // create a copy of the initial seed so we don't modify the original one
37         int[] register = INITIAL_SEED.clone();
38
39         //to store the output sequence (bits that come out)
40         StringBuilder output = new StringBuilder();
41
42         int period = 0;// How many steps until repetition
43         int maxPeriod = (int)Math.pow(2, b: LENGTH) - 1; //2^7 - 1 = 127
44
45         // to display header informations

```

```

45         // to display header informations
46         System.out.println("LFSR Analysis Polynomial: " + POLYNOMIAL);
47         System.out.println("Max possible period: " + maxPeriod);
48         System.out.println(x: "\nClk    S6 S5 S4 S3 S2 S1 S0      Output");
49         System.out.println(x: "-----");
50
51         // Show initial state (step 0) Clk 0
52         System.out.printf(format: "%d      %s      %n", args:arrayToString(arr:register));
53
54         // Main LFSR loop, run to 128 cycles (more than max possible)
55         for (int clock = 1; clock <= 128; clock++) {
56
57             int andFeedback = register[AND_TAPS_FEEDBACK[0]] & register[AND_TAPS_FEEDBACK[1]]; // AND between positions 3 and 5
58
59             int feedback = register[2] ^ register[4] ^ register[6] ^ andFeedback;
60
61
62             int andOutput = register[AND_TAPS_OUTPUT[0]] & register[AND_TAPS_OUTPUT[1]]; // AND between positions 1 and 5
63
64             int outputBit = register[6] ^ andOutput;
65             output.append(i: outputBit); // Add to our output sequence
66
67             // // SHIFT operation:
68             int[] newRegister = new int[LENGTH]; // new empty register
69
70             newRegister[0] = feedback; // Put feedback bit(result of XOR) in leftmost position (S6)
71
72
73             // Copy all bits from old register, shifting them right by one position
74             // register[0] goes to newRegister[1], register[1] to newRegister[2], and so on..
75             System.arraycopy(src:register, srcPos: 0, dest:newRegister, destPos:1, LENGTH - 1);
76
77             // Update our register to the new state
78             register = newRegister;
79
80             // Display the current state
81             System.out.printf(format: "%-2d      %s      %d%n", args:clock, args:arrayToString(arr:register), args:outputBit);
82
83             // Period detection ,check if we've returned to the initial state
84             if (arrayEquals(a: register, b: INITIAL_SEED)) {
85                 period = clock; // found the period length
86                 System.out.println("**** Period: " + period + " ****");
87                 break; // Exit the loop since we found repetition
88             }
89

```

```

90     }
91     // Return all our results packaged together
92     return new NLFSRBoth_Result(period, output.toString(), maxPeriod);
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 *          Display results with analysis
132 *          Shows the final results to the user
133 */
134 }

135 /**
136 *          Display results with analysis
137 *          Shows the final results to the user
138 */
139 private static void displayResults(NLFSRBoth_Result result) {
140
141     System.out.println(x: "\n**** ANALYSIS RESULTS ****");
142     System.out.println("Period: " + result.period);
143     System.out.println("Max possible: " + result.maxPeriod);
144     System.out.println("Output length: " + result.output.length());
145
146
147     //Check if we achieved maximum period (in our case no)
148     if (result.period == result.maxPeriod) {
149         System.out.println(x: "Maximum period achieved ✅");
150     }
151     else {
152         System.out.println(x: "Maximum period not achieved ❌");
153         System.out.println(x: "Reason: Polynomial is reducible and not primitive");
154     }
155
156     // to show the full sequence of bits
157     System.out.println("Full output sequence: " + result.output);
158
159     assessRandomness(sequence: result.output); // method to asses the randomness
160 }

161 /**
162 *          Randomness assessment : Check the balance
163 */
164 private static void assessRandomness(String sequence) {
165     System.out.println(x: "\n**** Simple Randomness check for both feedback & feedforward****");
166
167     int ones = 0; // to count 1s
168     for (char bit : sequence.toCharArray()) {
169         if (bit == '1') ones++; // if the bit equals 1 , plass the ones
170     }
171     int zeros = sequence.length() - ones; // to count 0s
172
173     // to calculate percentage :
174     double onesPercent = (ones * 100.0) / sequence.length();
175     double zerosPercent = (zeros * 100.0) / sequence.length();
176
177     //      for printing
178     System.out.println(x: "BALANCE TEST:");
179     System.out.println("    • 1s: " + ones + " (" + String.format(format: "%.1f", args:onesPercent) + "%)");
179     System.out.println("    • 0s: " + zeros + " (" + String.format(format: "%.1f", args:zerosPercent) + "%)");

```

```

181 // if ratio is between 40-60%, it's good
182 if (ones >= sequence.length()*0.4 && ones <= sequence.length()*0.6) { // between 40-60%
183     System.out.println(x: " ✅ Good balance (close to 50/50)");
184 } else {
185     System.out.println(x: " ❌ Poor balance");
186 }
187
188 // to call the pattern test:
189 performPatternTest(sequence);
190 }
191 /**
192 * PATTERN TEST with sliding window of size 2n (14 bits)
193 * Add this method right after assessRandomness
194 */
195 private static void performPatternTest(String sequence) {
196     int n = LENGTH; // n = 7
197     int windowSize = 2 * n; // 2n = 14 bits
198
199     System.out.println("\nPATTERN TEST (sliding window: " + windowSize + " bits = 2 x " + n + ")");
200
201     // Check if sequence is long enough
202     if (sequence.length() < windowSize) {
203         System.out.println(" • Sequence too short for " + windowSize + "-bit windows");
204         return;
205     }
206
207     Set<String> uniquePatterns = new HashSet<>();
208     int totalWindows = 0;
209
210     // Slide the 14-bit window through the sequence
211     for (int start = 0; start <= sequence.length() - windowSize; start++) {
212         String window = sequence.substring(beginIndex: start, start + windowSize);
213         uniquePatterns.add(e: window);
214         totalWindows++;
215     }
216
217     System.out.println(" • Total sliding windows: " + totalWindows);
218     System.out.println(" • Unique " + windowSize + "-bit patterns: " + uniquePatterns.size());
219
220     // Simple assessment
221     double diversity = (uniquePatterns.size() * 100.0) / totalWindows;
222     if (diversity > 50) {
223         System.out.println(" ✅ Good pattern diversity (" + String.format(format: "%.1f", args:diversity) + "%)");
224     } else {
225         System.out.println(" ❌ Limited pattern diversity (" + String.format(format: "%.1f", args:diversity) + "%)");
226     }
227
228     // Show a few sample patterns
229     System.out.println(x: " • Sample patterns:");
230     int count = 0;
231     for (String pattern : uniquePatterns) {
232         if (count < 2) { // Show only 2 samples
233             System.out.println(" - " + pattern);
234             count++;
235         } else {
236             break;
237         }
238     }
239
240 // ----- In this section I used methods to help me -----
241
242 /*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" */
243 private static String arrayToString(int[] arr) {
244     try {
245         StringBuilder sb = new StringBuilder();
246         for (int value : arr) {
247             sb.append(i: value).append(str: " ");
248         }
249         return sb.toString().trim(); //trim to remove spaces
250     } catch (Exception e) {
251         return "Error converting array to string";
252     }
253 }
254
255 /*
256 * Compare two arrays for equality
257 * it returns true if both arrays have same length and same values in same index*/
258 private static boolean arrayEquals(int[] a, int[] b) {
259     try {
260         if (a.length != b.length) return false;
261         for (int i = 0; i < a.length; i++) {
262             if (a[i] != b[i]) return false;
263         }
264         return true;
265     } catch (Exception e) {
266         return false;
267     }
268 }
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 NLFSRBoth_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     NLFSRBoth_Result(int period, String output, int maxPeriod) {
295         this.period = period;
296         this.output = output;
297         this.maxPeriod = maxPeriod;
298     }
299 }
300
301 }
302

```

[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	Output

1	0	1	0	0	0	0	0	.
2	0	0	1	0	0	0	0	.
3	1	0	0	1	0	0	0	.
4	0	1	0	0	1	0	0	.

◦ 1010010 ·
¬ 1101001 ·
∨ 1110100 ·
∧ 0111010 ·
⊕ 1011101 ·
⊖ 1101110 ·
⊻ 1110111 ·
⊼ 1111011 ·
⊽ 0111101 ·
⊿ 1011110 ·
⊸ 0101111 ·
⊶ 0010111 ·
⊹ 1001011 ·
⊺ 1100101 ·
⊻ 0110010 ·
⊻ 1011001 ·
⊻ 0101100 ·
⊻ 1010110 ·
⊻ 0101011 ·
⊻ 1010101 ·
⊻ 1101010 ·
⊻ 0110101 ·
⊻ 1011010 ·
⊻ 1101101 ·
⊻ 0110110 ·

٣٠ ٠٠١١٠١١ ٩
٣١ ٠٠٠١١٠١ ٩
٣٢ ٠٠٠٠١١٠ ٩
٣٣ ١٠٠٠٠١١ .
٣٤ ١١٠٠٠٠١ ٩
٣٥ ١١١٠٠٠٠ ٩
٣٦ ١١١١٠٠٠ .
٣٧ ١١١١١٠٠ .
٣٨ ٠١١١١١٠ .
٣٩ ٠٠١١١١١ ٩
٤٠ ١٠٠١١١١ ٩
٤١ ٠١٠٠١١١ ٩
٤٢ ٠٠١٠٠١١ .
٤٣ ٠٠٠١٠٠١ ٩
٤٤ ١٠٠٠١٠٠ ٩
٤٥ ١١٠٠٠١٠ .
٤٦ ٠١١٠٠٠١ ٩
٤٧ ٠٠١١٠٠٠ ٩
٤٨ ١٠٠١١٠٠ .
٤٩ ١١٠٠١١٠ .
٥٠ ١١١٠٠١١ ٩
٥١ ٠١١١٠٠١ .
٥٢ ٠٠١١١٠٠ ٩
٥٣ ٠٠٠١١١٠ .
٥٤ ١٠٠٠١١١ .

55 0100011 1
56 1010001 .
57 0101000 1
58 0010100 .
59 0001010 .
60 0000101 .
61 0000010 1
62 0000001 .
63 1000000 1

**** Period: 63 ****

64 0100000 .
65 0010000 .
66 1001000 .
67 0100100 .
68 1010010 .
69 1101001 .
70 1110100 1
71 0111010 .
72 1011101 1
73 1101110 1
74 1110111 1
75 1111011 .
76 0111101 .
77 1011110 1
78 0101111 .

79 0010111 ·
80 1001011 ·
81 1100101 ·
82 0110010 ·
83 1011001 ·
84 0101100 ·
85 1010110 ·
86 0101011 ·
87 1010101 ·
88 1101010 ·
89 0110101 ·
90 1011010 ·
91 1101101 ·
92 0110110 ·
93 0011011 ·
94 0001101 ·
95 0000110 ·
96 1000011 ·
97 1100001 ·
98 1110000 ·
99 1111000 ·
100 1111100 ·
101 0111110 ·
102 0011111 ·
103 1001111 ·

104 0100111 1
105 0010011 .
106 0001001 1
107 1000100 1
108 1100010 .
109 0110001 1
110 0011000 1
111 1001100 .
112 1100110 .
113 1110011 1
114 0111001 .
115 0011100 1
116 0001110 .
117 1000111 .
118 0100011 1
119 1010001 .
120 0101000 1
121 0010100 .
122 0001010 .
123 0000101 .
124 0000010 1
125 0000001 .
126 1000000 1
**** Period: 126 ****
127 0100000 .

128 0010000 .

**** ANALYSIS RESULTS ****

Period: 126

Max possible: 127

Output length: 128

Maximum period not achieved X

Reason: Polynomial is reducible and not primitive

Full output sequence:

00000010111001001111100011101111011000111011011001010010100010
10000001011100100111110001110111101100011101101100101001010001
0100

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

BALANCE TEST:

- 1s: 64 ($\pm 0\%$)
 - 0s: 64 ($\pm 0\%$)
- 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 ($\pm 0\%$)

- Sample patterns:

- 11111000111011

- 0100111100011

BUILD SUCCESSFUL (total time: 0 seconds)