ASSIGNMENT 5 TIPS AND TRICKS

- linear-feedback shift registers
- Java implementation
- a simple encryption scheme

http://princeton.edu/~cos126
Goals

- OOP: implement a data type; write a client program to use it.
- LFSR: learn about a simple machine and encryption scheme.
Assignment 5 Tips and Tricks

- linear-feedback shift registers
- Java implementation
- a simple encryption scheme
Linear-feedback shift register

- **Bit**: 0 or 1.
- **Cell**: storage element that holds one bit.
- **Register**: sequence of cells.
- **Seed**: initial sequence of bits.
- **Feedback**: Compute $\text{xor}$ of two bits and put result at right.
- **Tap**: bit positions used for $\text{xor}$ (one is always leftmost bit).
- **Shift register**: when clock ticks, bits propagate one position to left.

An 11-bit LFSR

```
0 1 1 0 1 0 0 0 0 1 0 1
```

11 10 9 8 7 6 5 4 3 2 1
Linear-feedback shift register simulation

```
+-----+-----+-----+-----+-----+-----+-----+
| 0   | 1   | 1   | 0   | 0   | 0   | 0   |
| 1   | 1   | 0   | 1   | 0   | 0   | 0   |
| 0   | 1   | 0   | 0   | 0   | 0   | 1   |
| 1   | 0   | 1   | 0   | 0   | 0   | 0   |
| 0   | 1   | 0   | 0   | 0   | 0   | 1   |
| 1   | 0   | 0   | 0   | 0   | 1   | 0   |
| 0   | 1   | 0   | 0   | 0   | 1   | 0   |
| 1   | 0   | 0   | 0   | 0   | 1   | 0   |
+-----+-----+-----+-----+-----+-----+-----+
```

```
<table>
<thead>
<tr>
<th>history of register contents</th>
<th>step</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 1 0 0 0 0 1 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1 0 1 0 0 0 0 1 0 1</td>
<td>1</td>
</tr>
<tr>
<td>1 0 1 0 0 0 0 1 0 1 1</td>
<td>2</td>
</tr>
<tr>
<td>0 1 0 0 0 0 1 0 1 1 0</td>
<td>3</td>
</tr>
<tr>
<td>0 1 0 0 0 0 1 0 1 1 0 0</td>
<td>4</td>
</tr>
<tr>
<td>0 0 0 0 1 0 1 1 0 0 1</td>
<td>5</td>
</tr>
</tbody>
</table>
```

a pseudo-random bit sequence!
LFSR quiz

Which are the next two bits that the LFSR outputs?

A. 0 0
B. 0 1
C. 1 0
D. 1 1
Assignment 5 Tips and Tricks

- linear-feedback shift registers
- Java implementation
- a simple encryption scheme
Applications programming interface

**API.** Specifies the set of operations.

```java
public class LFSR {
    public LFSR(String seed, int tap) creates an LFSR with specified seed and tap
    public int length() returns the length of the LFSR
    public int bitAt(int i) returns bit i as 0 or 1
    public String toString() returns a string representation of this LFSR
    public int step() simulates one step; return next bit as 0 or 1
    public int generate(int k) simulates k steps; return next k bits as k-bit integer
    public static void main(String[] args) tests every method in this class
}
```
public class LFSR {
    // Define instance variables here.

    // Creates an LFSR with the specified seed and tap.
    public LFSR(String seed, int tap)

    // Returns the length of the LFSR.
    public int length()

    // Returns bit i of this LFSR as 0 or 1.
    public int bitAt(int i)

    // Returns a string representation of this LFSR.
    public String toString()

    // Simulates one step of this LFSR; returns next bit as 0 or 1.
    public int step()

    // Simulates k steps of this LFSR; returns next k bits as a k-bit integer.
    public int generate(int k)

    // Tests every method in this class.
    public static void main(String[] args)
}

Testing

Develop program incrementally, one method at a time.

1. Define instance variables.
2. Implement constructor.
3. Implement length().
4. Implement bitAt().
5. Implement toString().
6. Implement step().
7. Implement generate().

Tip. Testing iteratively is the key to success.
LFSR representation

Q. How to represent the LFSR?
A. Several viable approaches.

Approach 1. Integer array of length $n$ in forward order: $\text{reg}[i] = \text{bit } (i + 1)$.

**Approach 2.** Integer array of length $n+1$ in forward order: $\text{reg}[i] = \text{bit } i$.

Approach 3. Integer array of length $n$ in reverse order: $\text{reg}[i] = \text{bit } n - i$.

Approach 4. Boolean array of length $n$ or $n+1$, in forward or reverse order.

Approach 5. String of length $n$: $\text{reg.charAt}(i) = \text{character corresponding to bit } n - i$.

Key point. The client doesn’t know (or care) how you represent the data type.
Applications programming interface

**API.** Specifies the set of operations.

```java
public class LFSR {
    public LFSR(String seed, int tap)  // creates an LFSR with specified seed and tap
    public int length()  // returns the length of the LFSR
    public int bitAt(int i)  // returns bit i as 0 or 1

    public String toString()  // returns a string representation of this LFSR

    public int step()  // simulates one step; return next bit as 0 or 1

    public int generate(int k)  // simulates k steps; return next k bits as k-bit integer

    public static void main(String[] args)  // tests every method in this class
}
```
String representation

Java's `toString()` method.

- Every Java class has a `toString()` method; by default, it returns the memory address of the object.
- Defining a custom `toString()` method overrides the default one.
- Java calls the `toString()` method automatically with string concatenation and `StdOut.println()`.

```java
LFSR lfsr = new LFSR("01101000010", 9);
StdOut.println(lfsr.toString());  // print string representation
StdOut.println(lfsr);            // better style
```

**Best practice.** Override the `toString()` method to facilitate debugging.

**Tip.** If you use an array representation, concatenate array elements (in proper order).
### Applications programming interface

**API.** Specifies the set of operations.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public class LFSR</code></td>
<td></td>
</tr>
<tr>
<td><code>public LFSR(String seed, int tap)</code></td>
<td>creates an LFSR with specified seed and tap</td>
</tr>
<tr>
<td><code>public int length()</code></td>
<td>returns the length of the LFSR</td>
</tr>
<tr>
<td><code>public int bitAt(int i)</code></td>
<td>returns bit i as 0 or 1</td>
</tr>
<tr>
<td><code>public String toString()</code></td>
<td>returns a string representation of this LFSR</td>
</tr>
<tr>
<td><code>public int step()</code></td>
<td>simulates one step; return next bit as 0 or 1</td>
</tr>
<tr>
<td><code>public int generate(int k)</code></td>
<td>simulates k steps; return next k bits as k-bit integer</td>
</tr>
<tr>
<td><code>public static void main(String[] args)</code></td>
<td>tests every method in this class</td>
</tr>
</tbody>
</table>
Generating a k-bit integer

Q. Suppose next 5 bits are 11001. How to convert into a 5-bit integer?

A1. Binary-to-decimal conversion: \[ 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 25. \]

A2. Horner’s method: \[ 2 \times ( 2 \times ( 2 \times ( 1 ) + 1 ) + 0 ) + 0 ) + 1 = 25. \]

Horner’s method.

- Initialize a variable `sum` to 0.
- For each bit, from left to right
  - double `sum` and add bit

<table>
<thead>
<tr>
<th><code>sum</code></th>
<th>bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( \times )</td>
<td>1</td>
</tr>
<tr>
<td>( \times )</td>
<td>0</td>
</tr>
<tr>
<td>( \times )</td>
<td>0</td>
</tr>
<tr>
<td>( -12 )</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
</tbody>
</table>

Tip. To implement `generate(k)`, don’t simulate LFSR from scratch; instead, make \( k \) calls to `step()`.
Applications programming interface

API. Specifies the set of operations.

```java
public class LFSR {
    public LFSR(String seed, int tap) creates an LFSR with specified seed and tap
    public int length() returns the length of the LFSR
    public int bitAt(int i) returns bit i as 0 or 1
    public String toString() returns a string representation of this LFSR
    public int step() simulates one step; return next bit as 0 or 1
    public int generate(int k) simulates k steps; return next k bits as k-bit integer
    public static void main(String[] args) tests every method in this class
}
```
Java characters

A char in Java is a 16-bit unsigned integer.

Unicode. Characters are encoded using Unicode.

- 'A' is 65.
- '0' is 48.
- '1' is 49.
- 'á' is 225.
- '☺' is 9775.

Best practices. Don’t write code that depends on internal representation.

- Good: if (c == '0').
- Bad: if (c == 48).
ASSIGNMENT 5 TIPS AND TRICKS

- linear-feedback shift registers
- Java implementation
- a simple encryption scheme
A simple encryption scheme

For each pixel in column-major order:

- Read \((\text{red}, \text{green}, \text{blue})\) values of pixel.
- Get 8 bits from LFSR and bitwise \text{\texttt{xor}} those with \text{red}.
- Get 8 bits from LFSR and bitwise \text{\texttt{xor}} those with \text{green}.
- Get 8 bits from LFSR and bitwise \text{\texttt{xor}} those with \text{blue}.
- Write \((\text{red}, \text{green}, \text{blue})\) values of resulting pixel.

original picture

\[\text{\texttt{\textit{Leci n'est pas une pipe.}}}\]

transformed picture

pixel (col, row)
A simple decryption scheme

For each pixel in column-major order:

- Read \((\text{red}, \text{green}, \text{blue})\) values of pixel.
- Get 8 bits from LFSR and bitwise \textit{xor} those with \textit{red}.
- Get 8 bits from LFSR and bitwise \textit{xor} those with \textit{green}.
- Get 8 bits from LFSR and bitwise \textit{xor} those with \textit{blue}.
- Write \((\text{red}, \text{green}, \text{blue})\) values of resulting pixel.
Why does it work?

Key identity. \((x \wedge y) \wedge y = x \wedge (y \wedge y) = x \wedge 0 = x\).

Important requirements.
- Must use the same initial LFSR to encrypt and decrypt.
- Must traverse the pixels in the same order.
public class Photomagic {

    // Returns a transformed copy of the specified picture,
    // using the specified LFSR.
    public static Picture transform(Picture picture, LFSR lfsr)

    // Takes the name of an image file and a description of an LFSR
    // as command-line arguments; displays a copy of the image that
    // is "encrypted" using the LFSR.
    public static void main(String[] args)

}