http://princeton.edu/~cos126

Assignment 5 Tips and Tricks

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

Goals

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

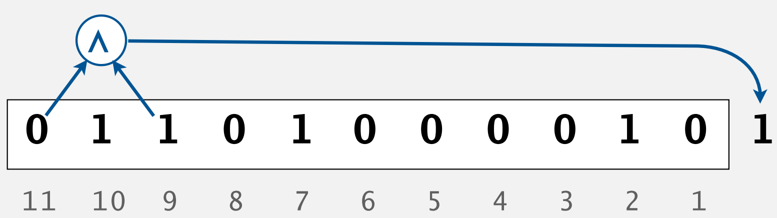
Assignment 5 Tips and Tricks

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

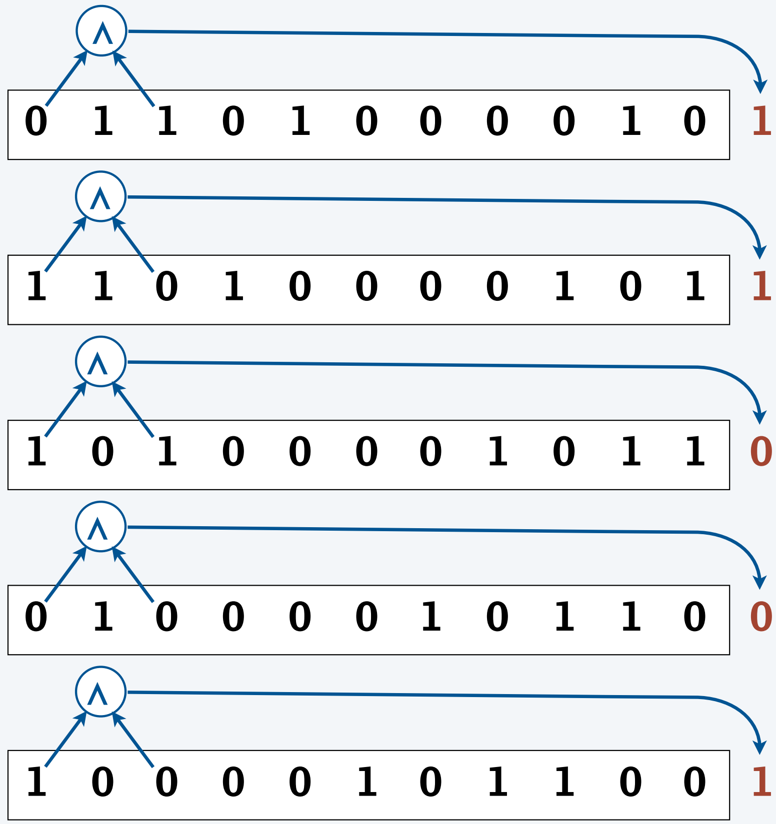
Linear-feedback shift register

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

an 11-bit LFSR



Linear-feedback shift register simulation



history of register contents	step
0 1 1 0 1 0 0 0 0 1 0	0
1 1 0 1 0 0 0 0 1 0 1	1
1 0 1 0 0 0 0 1 0 1 1	2
0 1 0 0 0 0 1 0 1 1 0	3
1 0 0 0 0 1 0 1 1 0 0	4
0 0 0 0 1 0 1 1 0 0 1	5

a pseudo-random bit sequence!

LFSR quiz

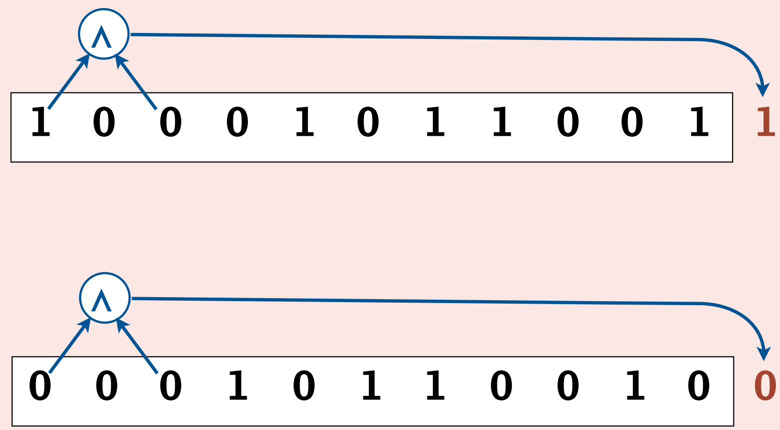
Which are the next two bits that the LFSR outputs?

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.

`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*

LFSR.java template

```java
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 construtor.
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.

0	1	1	0	1	0	0	0	0	1	0
11	10	9	8	7	6	5	4	3	2	1

LFSR indices go from right to left, starting at 1

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

Approach 2. Integer array of length $n+1$ in forward order: `reg[i]` = bit i.

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

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

Approach 5. String of length n: `reg.charAt(i)` = 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.

`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()`.

```
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.

`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*

Generating a k-bit integer

Q. Suppose next 5 bits are `1 1 0 0 1`. 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: $2 \times (2 \times (2 \times (2 \times (1) + 1) + 0) + 0) + 1 = 25$.

Horner's method.

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

sum	bits
~~0~~	1
~~1~~	1
~~3~~	0
~~6~~	0
~~12~~	1
25	

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.

```
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.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	“	#	$	%	&	‘	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

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

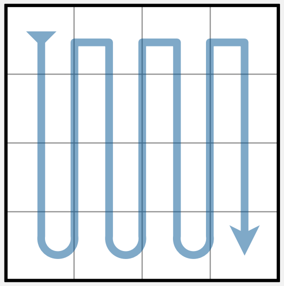
- *linear-feedback shift registers*
- *Java implementation*
- *a simple encryption scheme*

A simple encryption scheme

For each pixel in column-major order:

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

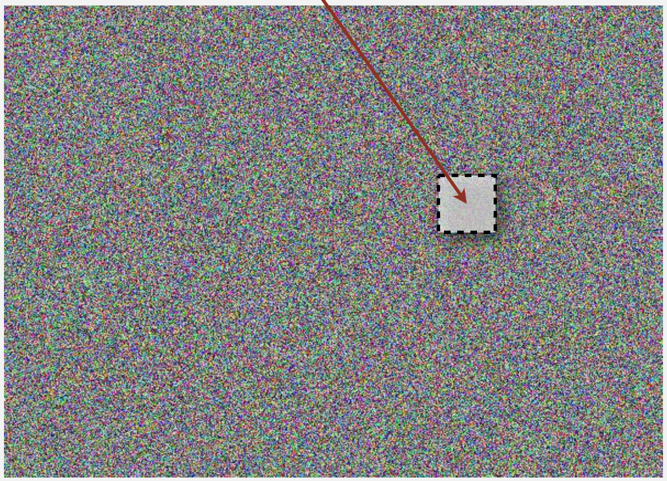
column-major order



pixel (col, row)

original picture

transformed picture

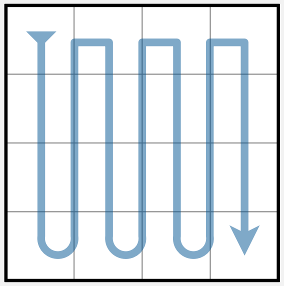


A simple decryption scheme

For each pixel in column-major order:

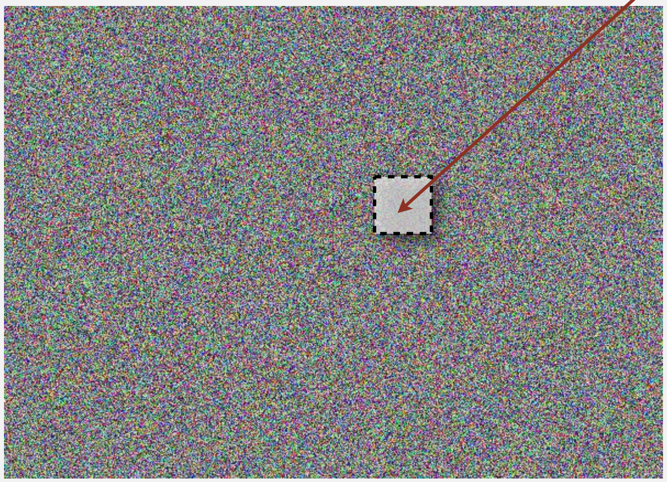
- Read (*red*, *green*, *blue*) values of pixel.
- Get 8 bits from LFSR and bitwise *xor* those with *red*.
- Get 8 bits from LFSR and bitwise *xor* those with *green*.
- Get 8 bits from LFSR and bitwise *xor* those with *blue*.
- Write (*red*, *green*, *blue*) values of resulting pixel.

column-major order



pixel (col, row)

transformed picture



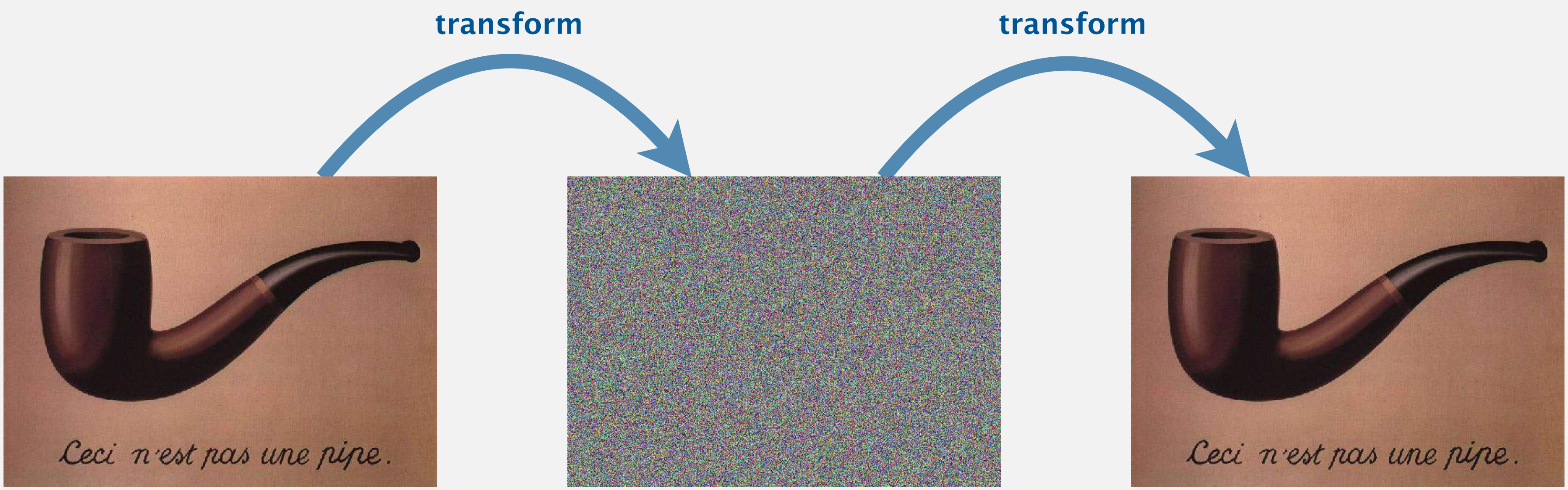
transformed transformed picture

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.



Photomagic.java template

Tip. See `ColorSeparation.java` from precept.

must create and return a copy of picture,
(do not mutate argument picture)

```
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)
}
```

```
% java-introcs PhotoMagic pipe.png 01101000010100010000 16
```

name of image

description of LFSR
(seed and tap)