# CS 3430: S24: Lecture 21
# Reflections on Randomness: Part 01
# Pseudorandom Number Generation

Vladimir Kulyukin
Department of Computer Science
Utah State University

# Why Study Randomness?

Why do we study randomness? There are many practical reasons such as computer gaming, gambling (e.g., building casinos, designing slot machines, organizing lotteries, etc.), cryptography, etc. I do not make any personal judgement in the previous statement. E.g., I know economists who think that casinos and lotteries are good for local economies. These are just some areas where random number generators are used.

A much more fundamental answer is the nature of the universe. Is the universe random or not? There can be three possible answers to this question with respect to any natural phenomenon – **yes**, **no**, **maybe**.

# Why Study Randomness?

The modern scientific method was outlined in Francis Bacon's book "New Organon" ("Novum Organum"), which he published in 1620 in Latin. There are three fundamental principles of the scientific method: 1) experimentation; 2) repeatability; and 3) indepdent verification (aka replicability).

Broadly speaking, the first principle states that nature can be investigated only through experiments; the second principle states that any experiment must be repeatble (note a subtle assumption – experiments cannot be repetable if the investigated phenomenon is not repetable); 3) the third principle states that the experiment's design, setup, execution, and results can be done by somebody else.

# Why Study Randomness?

Since the current scientific method is based on repeatability, it is not well suited for investigating random phenomena, because they are, by definition, not repetable. E.g., you may have heard that in finance/investment, non-repeatable events are sometimes called "black swans" Why? Because they cannot be predicted with probability-theoretic models.

Thus, if we establish that a natural phenomenon is random, we establish the non-applicability of the scientific method in investigating it. If we estabilish that it is not random, it makes sense to use the scientific method. If we cannot answer the randomness question affirmatively or negatively, we can keep investigating the phenonmenon, e.g., keep collecting more observations.

If we establish that some phenomenon is random and we cannot use the scientific method in investigating it, we can use random number generation to simulate this phenomenon.

# Pseudorandom Number Generation

**Pseudorandom Number Generation** (PRNG) is a collection of methods for generating sequence of real numbers that are hard to predict/repeat.

The term *hard* is ambiguous, because it varies from context to context. In computer science, *hard* typically means that it is impossible or difficult to write a computer program to repeat the same number sequences or, if it is possible, it takes a lot of time to compute or requires access to special/expensive hardware.

# Question

Consider the following three sequences and assume that whatever patterns we identify in each sequence are repeatable into infinity:

1. $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \ldots$;
2. $1, 9, 4, 6, 2, 3, 5, 5, 7, 0, \ldots$;
3. $1, 4, 7, 2, 5, 8, 3, 6, 9, 4, \ldots$.

Let us ask ourselves, Which sequences seems more "random" and why?

# Answer

Recall that in our lecture today (04/01/2024), we voted on which sequence appears random to us. Most people (including me) voted for the second sequence.

1. $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \ldots$;
2. $1, 9, 4, 6, 2, 3, 5, 5, 7, 0, \ldots$;
3. $1, 4, 7, 2, 5, 8, 3, 6, 9, 4, \ldots$.

Perception of randomness is variable. My rationale for voting for sequence 2 was this. In sequence 1, every subsequent number is generated from the previous one with $+1$; sequence 3 is generated as follows: from 1 we generate $1+3 = 4$, and $4+3 = 7$; from 1 we generate $1+1 = 2$; from 2 we generate $2+3 = 5$ and $5+3 = 8$; from 2 we generate $2+1 = 3$ and then from 3 we generate $3+3=6$, $6+3=9$, etc. I do not see any straightforward pattern in sequence 2. Well, my rationale is biased, because I generated sequence 2 with `random.randint()` :-). But, jokes aside, I honestly do not see any pattern in this sequence.

# Random Number Generation Methods

We can identify several groups of methods to generate random numbers.

1. **Gambiling:** We can throw dice, flip coins, spin roulette wheels, etc. to generate random numbers;
2. **Natural Phenomena:** We can pick a natural phenomenon that we expect to be random and measure it over a period of time while compensating, if we can, for possible biases in the measurement process. Examples: atmospheric noise, thermal noise, various electromagnetic and quantum phenomena, solar radiation or radioactive decay measured over short time periods, etc. We can subsequently treat numerical measurements as random numbers.
3. **Mathematics:** We can use the non-repeating mantissas of transcendental numbers such as $\pi$ and $e$ as the source of random numbers.

# Why Are Random Number Generators Called Pseudorandom?

Computational methods can produce long sequences of apparently random numbers. However, they are, in fact, determined by an initial value, known as **seed**/**key**/**start**, and a specific method that uses the seed to generate the next numbers in the sequence.

If the method and the seed are known, the entire pseudorandom sequence can likely be reproduced. I said *likely* in the previous sentence, because if we do not know the hardware or do not have access to the hardware on which given sequences are generated, we may not be able to generate these sequences. This is a good place to recall numerical instability. If we do have access to the same hardware, then we will be able to generate the same sequences. Hence, the term *pseudorandom*.

If a sequence is *purely* random, it cannot be repeated, by definition, under any circumstances.

# Usefulness of Pseudorandom Number Generation

Many pseudorandom number sequences have good "random" properties, because they have long non-repeating periods. Eventually, all pseudorandom sequences start repeating or the memory usage must grow without bound.

Among other useful properties of PRNG, we can mention:

1. PRNG does not depend on any external events;
2. PRNG produces long sequences reasonably fast;
3. PRNG is applicable in many situations that random model natural phenomena.

# Linear Congruential Generator (LCG)

A common PRNG method is the linear congruential generator (LCG), which uses the following recurrence generator formula:

$$X_{n+1} = (aX_n + b) \bmod m,$$

where $a$, $b$, and $m$ are positive integers, $X_{n+1}$ is the next pseudorandom number in the sequence and $X_n$ is the previous number.

The maximum length of a non-repetable sequence this formula can generate is one less than the modulus $m$, i.e., $m - 1$.

# Linear Congruential Generator (LCG)

In the formula,

$$X_{n+1} = (aX_n + b) \bmod m,$$

1. $X$ is the sequence of pseudorandom values;
2. $X_{n+1}$ is the next pseudorandom number:
3. $X_n$ is the previous pseudorandom number;
4. $m : 0 < m$ is the modulus (a positive integer);
5. $a : 0 < a < m$ is the multiplier;
6. $b : 0 \leq b < m$ is the increment;
7. $X_0 : 0 \leq X_0 < m$ is the seed/key/start value.

If $b = 0$, the generator is often called a multiplicative congruential generator (MCG).

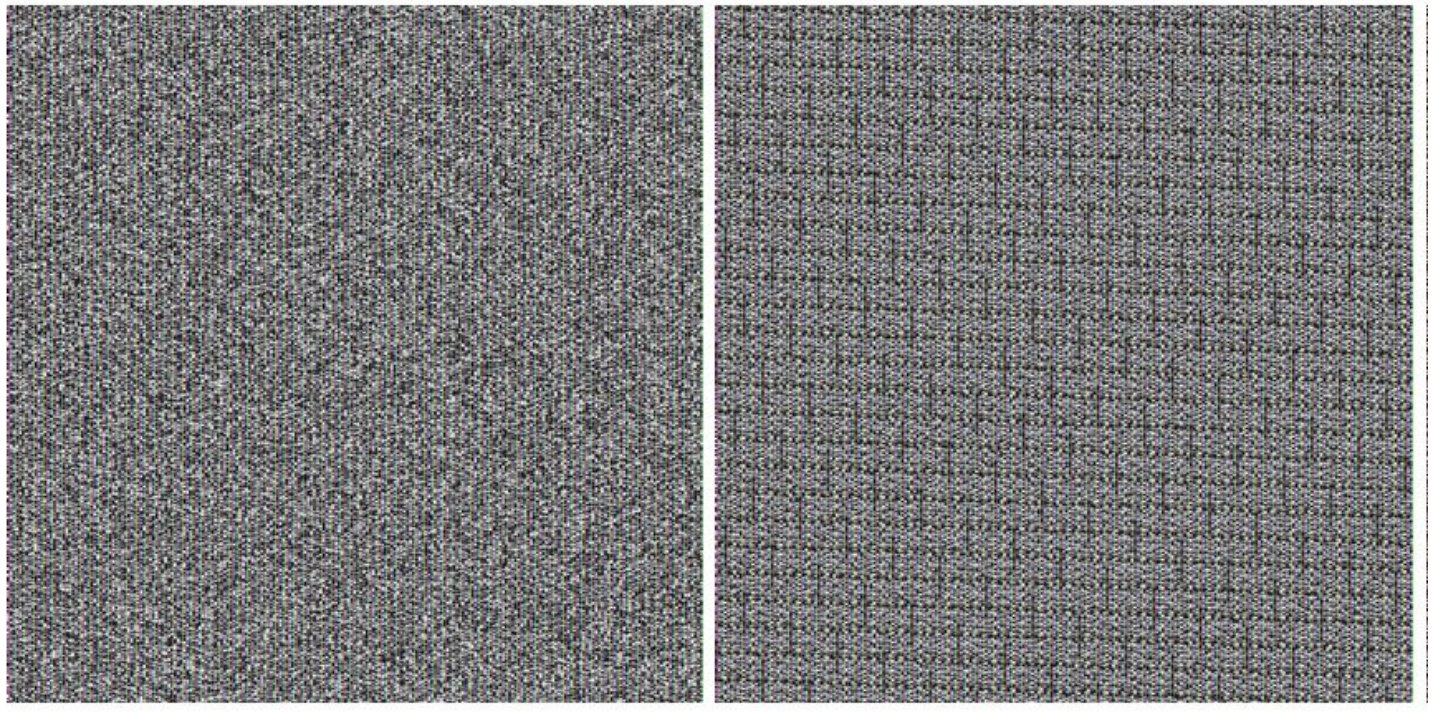
# Visual LCG Evaluation

Let us evaluate the randomness of LCG visually with the following LCG parameters:

1. $a = 214013$;
2. $b = 2531011$;
3. $m = 4294967296$.

Let us generate two sequences of $600^2$ random numbers with LCG: sequence 1 is generated from seed=0 and sequence 2 is generated from seed=$0, 1, ..., 600^2 - 1$. We can put these random numbers into into two 600x600 images.

# Visual LCG Evaluation



**Left Image:** image presentation of LCG sequence generated from $seed = 0$, $a = 214013$, $b = 2531011$, $m = 4294967296$.

**Right Image:** image presentation of LCG sequence generated from $seed = 0, 1, ..., 600^2 - 1$, $a = 214013$, $b = 2531011$, $m = 4294967296$.

# Choosing Triples $(a, b, m)$ for LCG

How to find good triples $(a, b, m)$? We want to choose $(a, b, m)$ so that the sequence $\{X_n\}$ has a period of exactly $m$ (no less!). We can use the following theorem proved by Dr. Donald Knuth:

**Theorem:** The linear congruential sequence defined by $m, a, b$, and $X_0$ has a period of length $m$ if and only if

1. $b$ is relatively prime to $m$;
2. $c = a - 1$ is a multiple of $p$, for every prime $p$ dividing $m$
3. $c$ is a multiple of 4 whenever $m$ is a multiple of 4.

**Source:** Knuth, D. E. The Art of Computer Programming. vol. 2. 2 ed. Reading, MA: Addison-Wesley, 1981, p. 17.

# Choosing Triples $(a, b, m)$ for LCG

For example: $b$ can any odd positive integer and $m$, $a$ can be as follows:

| m | a |
|---|---|
| $2^{30}$ | 438,293,613 |
| $2^{36}$ | 12,132,445 |
| $2^{48}$ | 181,465,474,592,829 |
| $2^{60}$ | 454,339,144,066,433,781 |

For more details, cf. Pierre L'Ecuyer "Tables of linear congruential generators of different sizes and good lattice structure" (Table 4, on p.258).

# XORShift

XORShift is a random number generator, also called *shift-register* generator, that generates the next number in a sequence by repeatedly taking the exclusive or (xor) of a number with a bit-shifted version of itself.

XOR abbreviates *e***X***clusive* **OR**.

# XOR and Left and Right Shifts

Since XORShift uses the bitwise XOR and left and right shifts, let us us briefly review them. Bitwise XOR is a bitwise operation computed by the following table.

| Bit 1 | Bit 2 | XOR |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

In the following slides, we will use $(x)_{10}$ to refer to some number $x$ written in decimal notation and $(x)_2$ to refer to some number $x$ written in binary.

# XOR and Left and Right Shifts

Let $x = 53$, i.e., $(x)_{10} = 53$ and $(x)_2 = 110101$. In Python, we can obtain the binary representation of a integer as follows.

```
>>> bin(53)
'0b110101'
>>> bin(53)[2:]
'110101'
```

Here is how we can compute XOR in Python:

```
>>> 53 ^ 19 == 38
```

The above statement can be interpreted as follows: when 53 and 19 are from decimal to binary and XOR is computed between the two binary strings, the result is the binary string such that when it is converted to decimal is equal to 38.

# XOR and Left and Right Shifts

```
>>> bin(53)[2:]
'110101'
>>> bin(19)[2:]
'10011'
>>> bin(38)[2:]
'100110'
```

Or, in tabular form,

| | | | | | | |
|---|---|---|---|---|---|---|
| $(53)_2$ | 1 | 1 | 0 | 1 | 0 | 1 |
| $(19)_2$ | 0 | 1 | 0 | 0 | 1 | 1 |
| $(53)_2 \wedge (19)_2$ | 1 | 0 | 0 | 1 | 1 | 0 |

# XOR and Left and Right Shifts

In Python, the bitwise right shift, can be expressed as follows.

$$x_{10} >> y \equiv x_{10} // 2 * *y$$

For example,

```
>>> 115 >> 1 == 115 // 2**1
True
>>> 115 >> 2 == 115 // 2**2
True
>>> 115 >> 3 == 115 // 2**3
True
```

# XOR and Left and Right Shifts

In Python, the left shift can be expressed as follows.

$$x_{10} << y \equiv x_{10} * (2 * *y)$$

Example:

```
>>> 115 << 1 == 115 * 2**1
True
>>> 115 << 2 == 115 * 2**2
True
>>> 115 << 3 == 115 * 2**3
True
```

# 32-Bit XORShift

There are different versions of XORShift. We will take a look at 32-bit XORShift
Let $\beta$ be the seed expressed in the form $\beta_0 = (b_1, b_2, \ldots, b_n)$, where $b_i \in \{0, 1\}$ and $n = 32$. Then, $\beta_1$ can be derived from $\beta_0$ as follows:

$$\begin{aligned} \alpha_0 &= \beta_0 \wedge (\beta_0 << a); \\ \alpha_1 &= \alpha_0 \wedge (\alpha_0 >> b); \\ \beta_1 &= \alpha_1 \wedge (\alpha_1 << c), \end{aligned}$$

for some triple $(a, b, c)$ of integers. The $\wedge$ is the XOR operation. The general formula for the $n$-th number, i.e., $\beta_n$ , in the sequence is:

$$\begin{aligned} \alpha_0 &= \beta_{n-1} \wedge (\beta_{n-1} << a); \\ \alpha_1 &= \alpha_0 \wedge (\alpha_0 >> b); \\ \beta_n &= \alpha_1 \wedge (\alpha_1 << c). \end{aligned}$$

This algorithm generates the pseudorandom sequence $\beta_0$, $\beta_1$, $\beta_2$, $\beta_3$, ....

# Choosing Triples $(a, b, c)$ in 32-Bit XORShift

The triple $(a, b, c)$ is chosen so that 32-bit XORShift has a period $2^{32} - 1$ (i.e., they provide the longest possible period for 32-XORShift). Here are several possibilities for $(a, b, c)$:

1. 1, 3, 10;
2. 2, 5, 15;
3. 3, 23, 25;
4. 5, 9, 28;
5. 7, 13, 25;
6. 13, 3, 27.

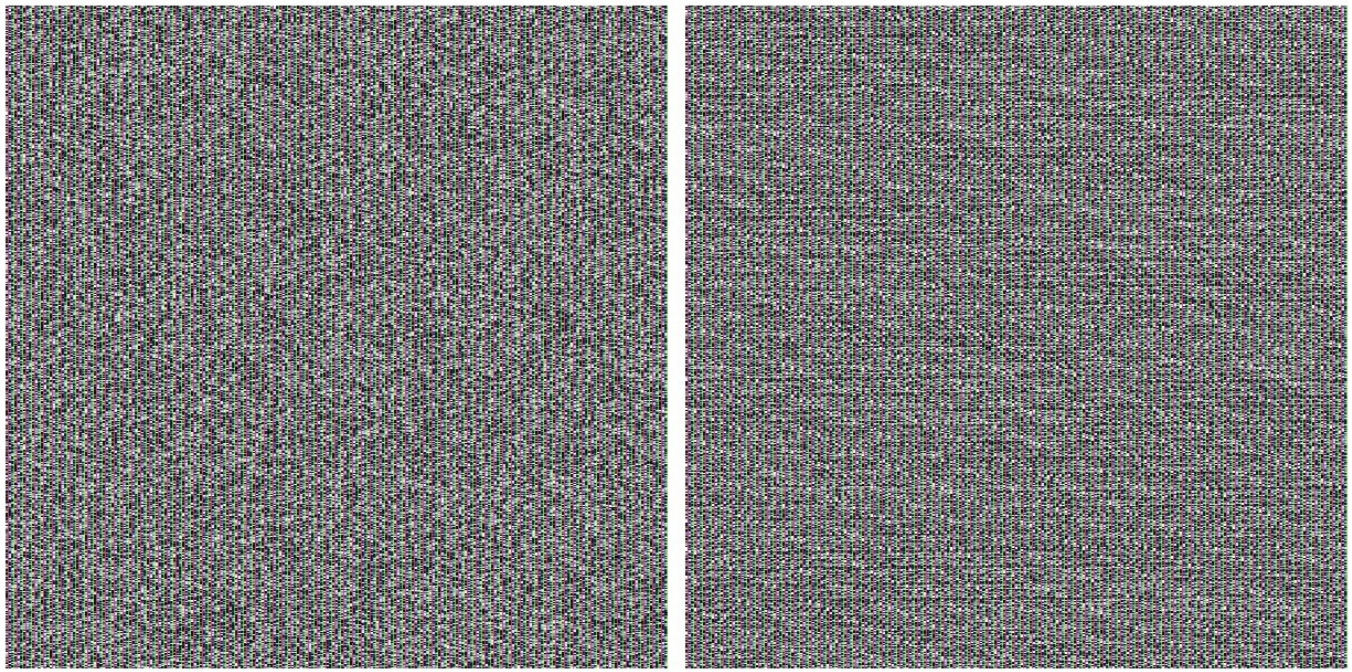
More suitable triples can be found in George Marsaglia's "Xorshift RNGs" (bottom of the page 2).

# Visual Evaluation of 32-bit XORShift

Let us generate two sequences of $600^2$ numbers with 32-bit XORshift: the first sequence is generated from seed 1; the second sequence is generated from seed=1, 2, ..., $600^2$.

Both sequences are saved in 600x600 images.

# Visual Evaluation of 32-bit XORShift



**Left Image:** image presentation of LCG sequence generated from $seed = 1$.

**Right Image:** image presentation of LCG sequence generated from $seed = 0, 1, ..., 600^2$.

# Mersenne Twister

Does there exist a generator whose patterns are really difficult to detect?

One such generator is the Mersenne Twister; this algorithm has a very large period, which is why the appearance of a pattern in large sequences of numbers is physically impossible or very hard; this algorithm is quite complex, and is beyond the scope of this class.

Python's random module is based on this algorithm.

```
>>> import random
>>> random.randint(0, 10)
>>> 4
```

# References

1. Knuth, D. E. The Art of Computer Programming, vol. 2. 2nd Ed. Reading, MA: Addison-Wesley, 1981, p. 17.

2. L'Ecuyer, P. "Tables of linear congruential generators of different sizes and good lattice structure." *Mathematics of Computation*, 68, pp. 249-260, 1999.

3. Marsagalia, G. "Xorshift RNGs." *Journal of Statistical Software*, July 4, 2003.