Introduction to Pseudo-Random Number Generators

Nicola Gigante

March 9, 2016

Why random numbers?

LifeâĂŹs most important questions are, for the most part, nothing but probability problems.

Pierre-Simon de Laplace

Why random numbers?

It often happens to be required to “throw a dice”:

- Randomized algorithms
- Simulation of physical phenomena
- Cryptography

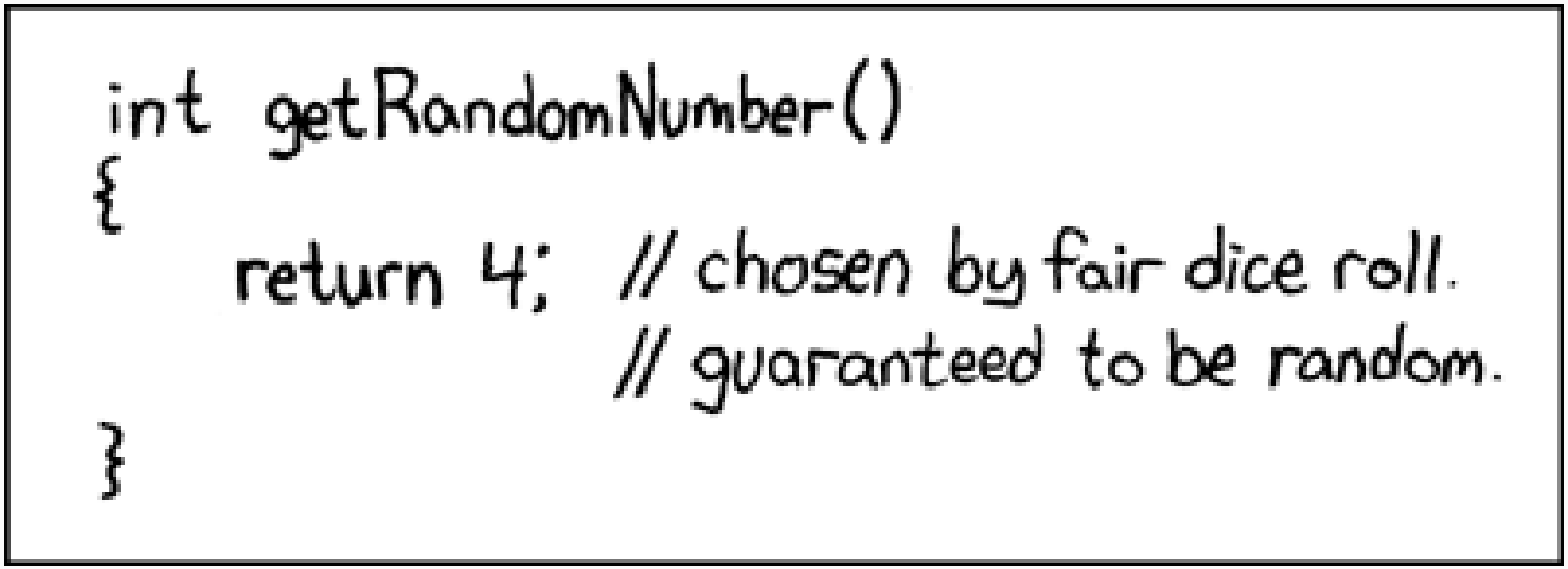
So random numbers are really important in Computer Science.

But what does *random* mean, by the way?

Table of Contents

What is Randomness?

What is Randomness?



RFC 1149.5 specifies 4 as the standard IEEE-vetted random number.

Quiz time

Which of these two sequences is more *random*?

1. `01010101010101010101010101010101010101010101`
2. `00100100001111110110101010001000100000101`

Uniform probability and predictability

Imagine to have a fair coin, e.g. $P(x = 0) = P(x = 1) = \frac{1}{2}$

- Then both sequences have the same probability $\frac{1}{2^{40}}$ of all the other possible 40-bits sequences.
- Nevertheless the second seems more random, why?
  1. Frequency of substrings is more *uniform*
  2. The sequence seems more *unpredictable*
- We will precisely define both properties later

Uniform probability and predictability

Uniformity and predictability seem related, but:

- Uniformity is an objective measure: are all substrings equally frequent?
- Predictability is not an objective property...

Uniform probability and predictability

Predictability is in the eye of the observer:

- Recall the sequence nÂř2?
  - It is the (beginning of the) binary expansion of π.
- So unpredictability and uniform probability are different things.
- We may want both, or only one of them, depending on the application.

Different definitions of randomness

We will look at different definitions of randomness, based on:

- Statistical features of the sequence
- Algorithmic complexity of the sequence
- Predictability of the sequence

Different kinds of randomness will be suitable for different applications.

Randomness as equidistribution

Definition (Equidistribution)

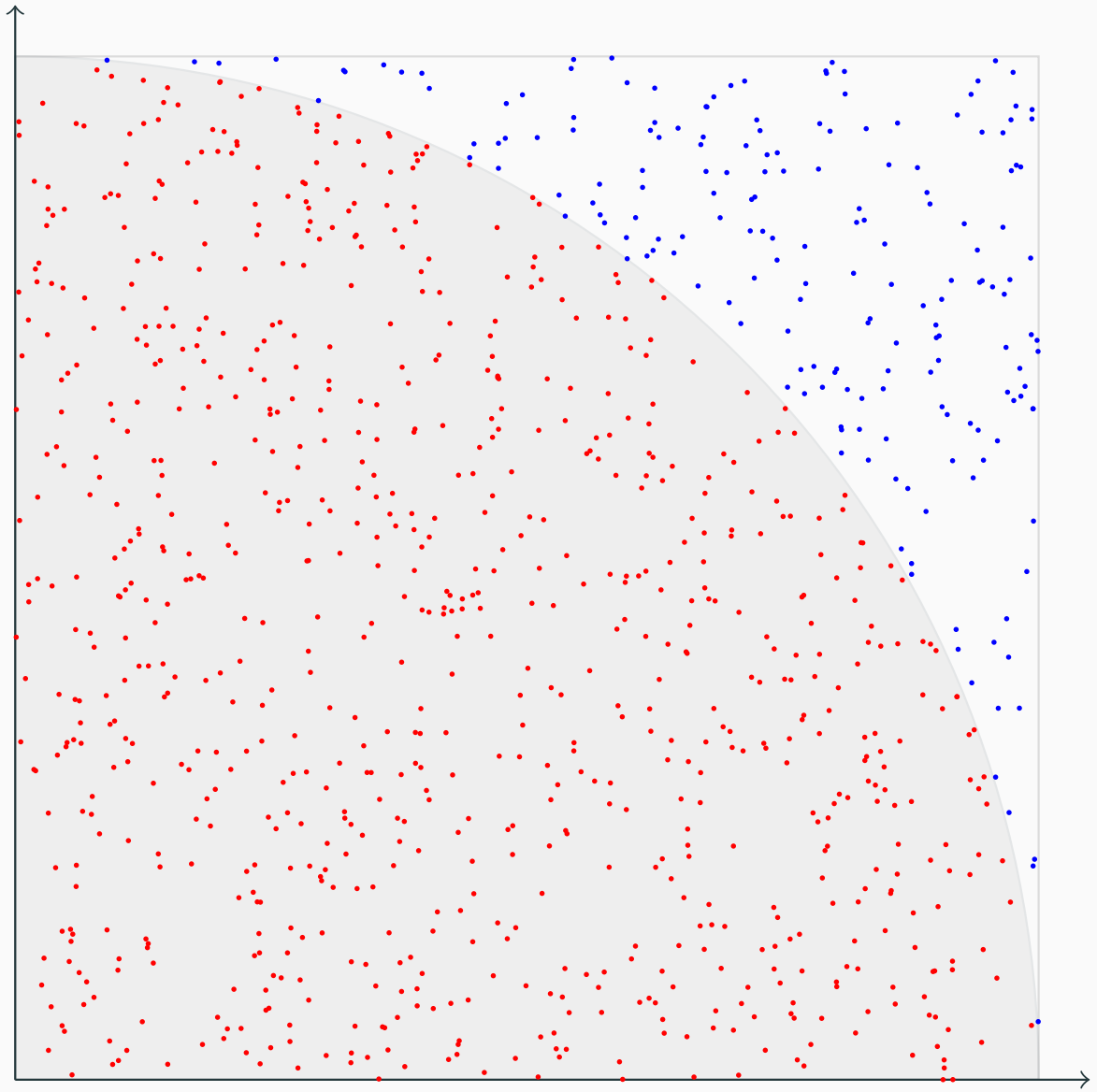
A sequence x_i of values with $x_i \in [0, N]$ is *equidistributed* if every subinterval $[a, b]$ contains a number of different values proportional to the length of the interval.

Informally

There is no region more "dense" than another.

The concept generalizes to k dimensions: k-distribution

Equidistribution (example)



Statistical randomness

Equidistribution is not the only way to define randomness in statistical terms.

Statistical randomness tells us how a given sequence is *likely* to come from a randomness source.

More on that on a statistics book.

Randomness as absence of patterns

The example of π before suggests a characterization.
We may want to exclude strings which exhibit *patterns*.

There are (at least) two different ways to define this concept:

- Shannon's Entropy
- Kolmogorov Complexity

Entropy

Definition

The *empirical Shannon's Entropy* of a sequence s is the following quantity:

$$H(s) = -\sum_{\sigma \in \Sigma} f_\sigma \log_2(f_\sigma)$$

where Σ is the alphabet and f_σ is the frequency of appearance of the character σ in the sequence.

Entropy

Important points

- The entropy function has its *maximum* when the characters are drawn from a *uniform distribution*.
- So a string with higher entropy is *more likely* to come from a source of uniform probability.
- The entropy of a string is the *lower bound* to how much it can be compressed by a *zero-order compression algorithm* (Shannon's Theorem).
- So a string with high entropy is also *less compressible*.

Kolmogorov Complexity

Definition

Let s be a string in some alphabet. The *Kolmogorov Complexity* of s, $K(s)$, is the size of the *shorter* program that can produce s as output.

Kolmogorov Complexity

Important points

- The *computation model* or *programming language* used does not matter.
- The size of the shorter program is *another* way to tell the minimum size to which the string can be *compressed*.
  - To decompress, just execute the program.
- Related to Shannon's Entropy, but different.
  - The π sequence has a *very high* entropy, but a tiny Kolmogorov Complexity.
  - Clearly the converse cannot happen.

Kolmogorov Complexity and Randomness

Definition (Martin-LÃűf)

A string s is called *algorithmically random* if $K(s) > |s| - c$, for some c.

This would be the perfect measure for randomness:

- If $K(s) < |s|$, the string contain some regular patterns that can be exploited to write a shorter program that produce s as output.
- If $K(s) \geq |s|$, it means the only way to produce the string is to show the string itself.

Where's the catch?

Uncomputability of Kolmogorov Complexity

Kolmogorov Complexity is not computable.

Suppose by contraddiction that it is. Fix a $k \in \mathbb{N}$ and consider the following program:

```
foreach string s:
  if K(s) >= k:
    print s
    terminate
```

This program outputs a string s with $K(s) \geq k$, but has length $\mathcal{O}(log(k))$.

So it's shorter than the shorter one that can output s. ↯

Randomness test by compression

So we cannot use $K(s)$ to test randomness, but:

- Asymptotically optimal compression algorithms approximate it
- Approximated Martin-LÃűf test: compress the data; if the size shrinks, data was not random enough.

Unpredictability

Possible definitions of randomness seen so far take into account *statistical features* of the sequences.

- This is the definition we care about in applications like *randomized algorithms* or *physical simulations*.
- The quality of the outcome depends on how much the sequence resemble a really uniform distribution.

However, in other applications, like *cryptography* and *secure communication protocols*, good statistical properties are not enough.

Random numbers in cryptography

Cryptographic algorithms make heavy use of random numbers:

- Key generation of public-key cyphers.
- Key exchange protocols.
- Initialization vectors of encyphered connections.

The security of cryptographic techniques is based on the assumption that an attacker cannot *guess* the random values choosen by the communication parties.

Random numbers in cryptography

Statistical properties of the sequence are *irrelevant* if the attacker can *predict* the next values, or *compute* past values.

Random numbers for cryptographic use must be *unpredictable*. Of course, statistical features follow.

Pseudo-Random Number Generators

How to Produce Random Numbers?

We saw a few different definitions of randomness.

A different question is: how to generate such numbers?

Turing machines — and our physical computers — are deterministic objects.

How can a deterministic machine generate a random sequence?

Spoiler

It can't

Physical randomness

Real randomness exists in the physical world:

- Quantum physics is *intrinsically* random.
- By measuring (for example), the spin of superposed electrons, one may extract a *physically* random sequence of bits.
- Another kind of physical randomness is *thermodynamic noise*.

Physical randomness

Hardware devices that exploit these sources *exist*, but:

- They are too slow.
- They cost too much.

Pseudo-random sequences

Definition

A *pseudo-random* number sequence is a sequence of numbers which *seems* to have been generated randomly.

Pseudo-Random Number Generators

An algorithm to produce a pseudo-random sequence is called a *pseudo-random number generator*.

Some common characteristics of PRNGs:

- Given an initial value, called *seed*, the algorithm produces a pseudo-random sequence of numbers.
- The algorithm is of course deterministic: from the same seed you obtain the same sequence, but the sequence by itself *looks* random.

Pseudo-Random Number Generators

Some common characteristics of PRNGs:

- The sequence evolution depends on an *internal state*
  - In simple PRNGs the internal state is only the current value of the sequence.
- The internal state is finite so the sequence will eventually *repeat*. The number of values before the sequence repeats is called the *period*.

Probability Distribution

PRNGs usually produce *integer* sequences that appear to have been drawn from a *uniform distribution*:

- Other distributions could be needed in an application (e.g. normal, Poisson, etc. . .)
- A sample from a uniform distribution can be transformed into a sample of other common distributions, e.g.:
  - for the central limit theorem, summing any random variable results in a normally-distributed variable.
  - $Y = -\lambda^{-1} \ln(X)$ has *exponential distribution* with rate λ
- Floating point values can be obtained from integers.

Good PRNGs

From a good (non-cryptographic) PRNG, we want:

- A *long* period.
- As much statistical similarity to a uniform distribution as possible.
- Speed.

Linear Congruency Generators

We will now explore one of the simpler kind of PRNG.
Linear Congruency Generators (LCG), aka *Lehmer* generators:

- Simple and very easy to understand.
- Very fast.
- Usually is the implementation of the C `rand()` function.
- Not so good randomness characteristics, but good enough for a lot of cases
- Easy to do it wrong.
  Good example to show an important point:
  Don't design a PRNG yourself.

Linear Congruency Generators

The sequence of a LCG is defined by the following recurrence:

$$x_{n+1} = ax_n + c \pmod{m}$$

Very general by itself:

- Its entire behaviour depends on the three parameters:
  - The *modulus* m
  - The *multiplier* a
  - The *increment* c
- Easy to do wrong (e.g. with $a = 2$ and $c = 0$ the sequence does not seem random at all)

How to choose the parameters? We restrict ourself to the case where $c = 0$.

Choosing the parameters - modulus

Clearly the modulus is an upper bound to the sequence period.

- So suppose to have 32bits integers, could we use a modulus of 2^{32} to cycle through all representable values?
- It's not a good choice.

Choosing the parameters - modulus

Let d be a divisor of m and y_n be the following sequence:

$$y_n = x_n \pmod{d}$$

Consider the x_n sequence.

$$\begin{aligned}
x_{n+1} &= ax_n + c + km && && \text{for some } k \\
y_{n+1} &= ax_n + c + km && \pmod{d} && \text{go } \pmod{m} \text{ on both sides} \\
y_{n+1} &= ax_n + c && \pmod{d} && d \text{ divides } m \\
y_{n+1} &= a(y_n - k'd) + c && \pmod{d} && \text{because } y_n = x_n + k'd \\
y_{n+1} &= ay_n + c && \pmod{d}
\end{aligned}$$

Choosing the parameters - modulus

So the residue modulo d of the sequence is a linear congruence sequence.
Examples of why this is not good:

- The j less significant bits of every number in the sequence form a subsequence that repeat every 2^j steps.
- If d is even, the sequence strictly alternates between even and odd values.

Solution: to choose a prime modulus.

- $2^{31} - 1$ is a common choice for sequences of 32bits integers

Choosing the parameters - multiplier

How to choose the multiplier?

- We want to obtain the maximum period of $m - 1$.
- In other words we need an a such that, for each $x \in [0, m-1]$, there exists an i such that:

$$a^i = x \pmod{m}$$

- $(\mathbb{Z}_m, \cdot)$ is a *cyclic group*. Such an element would be the *generator* of this group.
- If m is prime *any* element is a generator:
  - Any element a' with smaller period would generate a subgroup.
  - $\mathbb{Z}_m$ has a prime number of elements.
  - Lagrange theorem: for any subgroup $G < \mathbb{Z}_m$, $|G|$ divides $|\mathbb{Z}_m|$ (which is m). ↯

Statistical performance

So with a prime modulus, any multiplier reaches maximum period.

- This does not mean any multiplier has good performance.
- Extensively searching for the statistically best multiplier is feasible for 32bits values.
  - Park and Miller suggest this sequence:

$$x_{n+1} = 7^5 x_n \pmod{2^{31} - 1}$$

- This is the minimum standard suggested by Park and Miller, but it has a lot of limitations.

k-distribution of LGCs

Theorem (Marsaglia '68)

All k-tuples of consecutive values of a LGC sequence with modulus m lay on parallel $(k-1)$-planes, and the number of those planes is always less than $\sqrt[k]{k!m}$.

Depending on the application, this can be a bad thing.

- For example, the number of planes of *triples* of consecutive values is 2344, for $m = 2^{31} - 1$.
- This could be or not acceptable in a period of 2 billion elements.
- Increasing the modulus and keeping only the most significant bits can result in a k-distributed sequence.

Linear Congruency Generators - Recap

LCGs could be good when the result of our computation does not depend on the good statistical properties of the sequence, e.g.:

- randomized visual effects
- cheap randomized algorithms
  (used in non-sensible contexts)

There are much better alternatives.

Mersenne Twister

The Mersenne Twister is one of the most used *modern* PRNGs.

- Called this way because its period is always a Mersenne prime number.
- Huge period, e.g. $2^{19937} - 1$ for the `MT19937` variant.
- Great statistical performance: k-distributed up to $k = 623$.
- Very fast on modern architectures (with SIMD instructions).

Mersenne Twister

It still has statistical defects:

- The evolution of the state is not very *chaotic*: a seed with a lot of zeroes can result in a long initial subsequence with bad statistical characteristics.
- Even more recent improvements exist.

Use of Mersenne Twister in Practice

Most programming languages provide a ready implementation of Mersenne Twister in standard or commonly available libraries. Examples:

- `std::mt19937` in C++11.
- `math3.random.MersenneTwister` in Java *Apache Commons Math*.
- `System.Random.Mersenne` in Haskell.

In C, the `rand()` function is deprecated, don't use it. Find a ready MT implementation instead.

Requirements for a cryptographic PRNG

A pseudo-random sequence is *cryptographically strong* if it satisfies these requirements:

Next bit test

Given an initial subsequence, there is no *polynomial* algorithm that can predict the next element with a success probability of more than 50%.

Forward security

Given the knowledge of the internal state of the generator, no polynomial algorithm can compute the *previous* elements of the sequence.

Blum Blum Shub algorithm

Blum Blum Shub is a common cryptographically strong PRNG. It's the sequence of bits z_i produced as:

$$x_{n+1} = x_n^2 \pmod{m}$$
$$z_i = x_i \pmod{2}$$

- z_i is the least significant bit of x_i.
- Similar to LCG, but the recurrence is *quadratic*, and we extract a single bit of the entire state.
- Proved to be secure if *factorization* is hard.

Predictability of the seed

A good PRNG is not enough: what if the attacker could predict the *seed*?

- The predictability of the entire sequence depends on the seed.
- How to choose the seed? We should choose it at *random*.
- Ops...

Collecting physical entropy

The solution is to collect *real* randomness from the physical world:

- Any source of unpredictable events
- Common and easy ones:
  keystrokes, mouse clicks, interrupt from peripheral devices, content of network packets, sequence of syscalls from user processes, time, etc...
- *Real* randomness:
  quantistic phenomena, thermodinamic noise, etc...

Collecting physical entropy

The Operating System usually provides a facility to access physical entropy (e.g. `/dev/urandom` on Linux)

- Common entropy sources are usually sufficient, but can be not enough.
- Strong entropy generators are available. The hardware is not cheap, though.

Collecting physical entropy

Physical entropy is not a *replacement* for PRNGs.

- Physical entropy is a *rare* resource and its extraction is *slow*.
- User code should use it to choose a *seed* and use the seed to feed a cryptographic PRNG.
- Useful only for cryptography. No need for a physical seed for other applications.
- e.g. scientific simulations could even *require* to be able to reproduce the exact result by reusing the same known seed.

Collecting physical entropy

A single source is not enough. How to have enough entropy?

- The Operating System handles a *entropy pool*.
- All the different entropy sources are combined into a high entropy buffer
- e.g. data is compressed and XORed together.

Recap

What we learned:

- Definition of randomness is not easy
- Linear Congruency Generators
- Current state-of-the-art (almost): Mersenne Twister
- Why cryptographic random numbers are different
- Requirements for a cryptographic PRNG
- Collecting physical entropy is required to have an unpredictable seed

Questions?