



What are **random** numbers?

- **True random numbers**

- Truly non-deterministic numbers
- Conceptually, easy to imagine.
- Practically very hard to prove true randomness
- See <http://random.org> and see what they do

- **Pseudo-random numbers**

- A deterministic sequence of apparent random numbers calculated from a seed
- Good pseudo-random numbers should be very hard to guess the next numbers, just based on observations

Usage of random numbers

- **Statistical methods**

- Resampling : Permutation & Bootstrapping
- Simulation of data
- Stochastic processes for estimation : MCMC, Gibbs sampling

- **Cryptography**

- Generating “hard-to-predict” pseudo-random numbers given seed is closely related to encrypting the seed into a sequence of random bits
- Creating a “hard-to-fabricate” digital signature or a certificate is also closely related to generating a good hash function, which are reproducible pseudo-random numbers given seeds

What are **good** random numbers?

```
int getRandomNumber()  
{  
    return 4; // chosen by fair dice roll.  
              // guaranteed to be random.  
}
```

<https://xkcd.com/221/>

[http://ardiri.com/blog/
secure_number_generator_for_the_arduino](http://ardiri.com/blog/secure_number_generator_for_the_arduino)

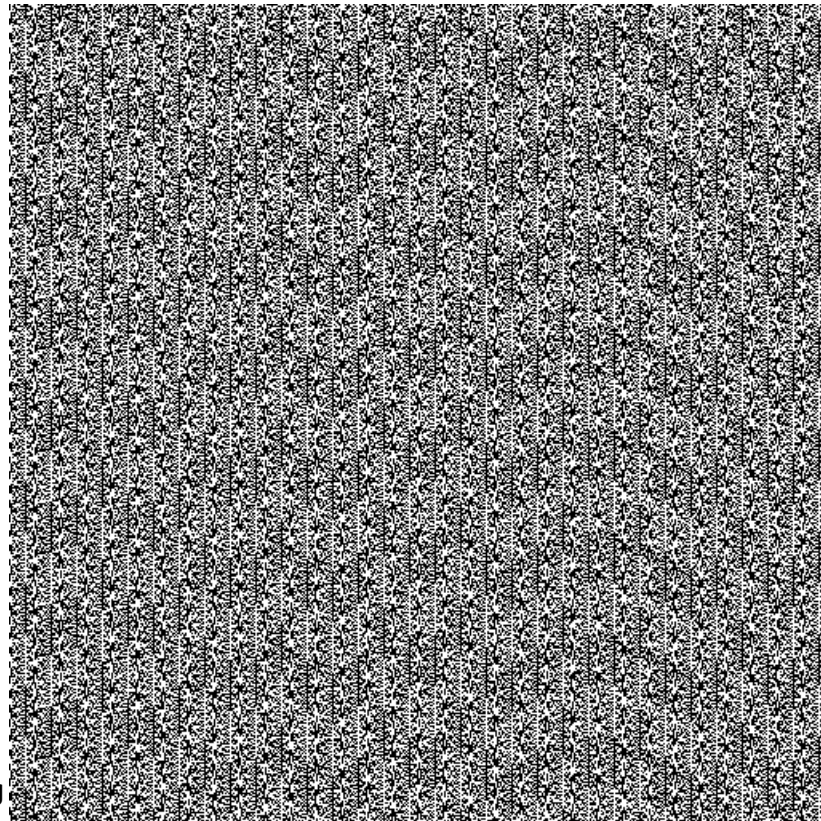
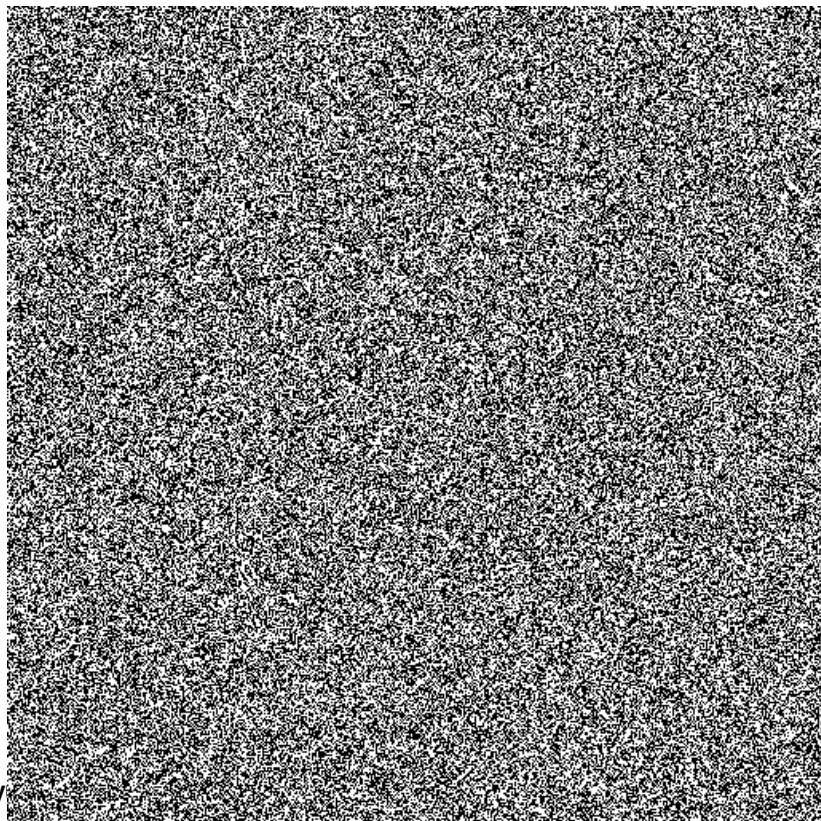


10/25/01 © 2001 United Feature Syndicate, Inc.



Good **vs.** bad pseudo-random numbers

- Which image generated from pseudo-random numbers looks random?



Playing a rock-paper-scissors **game**

- Suppose that you are playing **100** rock-paper-scissors games against a world champion of rock-paper-scissors, betting **\$1** per game.
- Your opponent is known to be **excellent** in predicting what your next move will be.
- What is the best strategy to **minimize** your loss?

Playing a rock-paper-scissors **game**

- Suppose that you are playing **100** rock-paper-scissors games against a world champion of rock-paper-scissors, betting **\$1** per game.
- Your opponent is known to be **excellent** in predicting what your next move will be.
- What is the best strategy to **minimize** your loss?

Answer: Use random.org

C++ `rand()` – An easy but poor choice.

```
#include <cstdlib>
#include <ctime>
#include <cstdio>
int main(int argc, char** argv) {
    srand( argc < 2 ? std::time(NULL) : atoi(argv[1]) );
    for(int i=0; i < 10; ++i)
        printf("%1d ", rand() % 2 );
    printf("\n");
    for(int i=0; i < 10; ++i)
        printf("%.51f ", (rand() + 0.5) / (RAND_MAX+1.0));
    printf("\n");
    return 0;
}
```


In most case, it works fine, but **who** knows?

```
$ ./rand1
```

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

```
0.97964 0.73368 0.99384 0.51616 0.12086 0.36270 0.89395
```

```
0.57954 0.35506 0.53798
```

```
$ ./rand1
```

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

```
0.74664 0.77251 0.65577 0.58531 0.22779 0.42210 0.23625
```

```
0.59494 0.12189 0.67166
```

```
$ ./rand1 12345
```

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

```
0.33315 0.19511 0.26720 0.79270 0.98388 0.14889 0.33915
```

```
0.03411 0.26621 0.16742
```

```
$ ./rand1 12345
```

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

```
0.33315 0.19511 0.26720 0.79270 0.98388 0.14889 0.33915
```

```
0.03411 0.26621 0.16742
```

Using **std::time** for seed could be **dangerous**

```
$ ./rand1;./rand1
```

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

```
0.70750 0.02907 0.60001 0.29918 0.35208 0.43513 0.18139
```

```
0.65265 0.02598 0.58701
```

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

```
0.70750 0.02907 0.60001 0.29918 0.35208 0.43513 0.18139
```

```
0.65265 0.02598 0.58701
```

C++ `random()` – A lower-risk choice.

```
#include <cstdlib>
#include <ctime>
#include <cstdio>
int main(int argc, char** argv) {
    srand( argc < 2 ? std::time(NULL) : atoi(argv[1]) );
    for(int i=0; i < 10; ++i)
        printf("%1d ", random() % 2 );
    printf("\n");
    for(int i=0; i < 10; ++i)
        printf("%.51f ", (random() + 0.5) / (RAND_MAX+1.0));
    printf("\n");
    return 0;
}
```

C++11 `<random>` – A more principled choice

```
#include <random>
#include <cstdio>

int main(int argc, char** argv) {
    std::default_random_engine prg;
    if ( argc > 1 ) prg.seed(atoi(argv[1]));
    else { std::random_device r; prg.seed(r()); }
    std::binomial_distribution<int> bdist(1, 0.5);
    for(int i=0; i < 10; ++i)
        printf("%d ", bdist(prg));
    printf("\n");
    std::uniform_real_distribution<double> udist(0.0, 1.0);
    for(int i=0; i < 10; ++i)
        printf("%.5lf ", udist(prg));
    printf("\n");
    return 0;
}
```

`<random>` is supported in C++11 standard. If it does not compile, you may need to add `--std=c++11` flag

when reproducibility is needed
when want it to be totally random.

Separates these three:

- (1) random_device (to pick a random seed)
- (2) PRG (pseudo-random number generator) engine
- (3) distribution to convert random bits to numbers

Running examples

```
$ ./rand3
```

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

```
0.80351 0.73279 0.53399 0.13210 0.53778 0.53161 0.98968
```

```
0.40718 0.81156 0.32475
```

```
$ ./rand3
```

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

```
0.49421 0.22141 0.97281 0.35042 0.26444 0.74354 0.11054
```

```
0.03712 0.25954 0.45112
```

```
$ ./rand3 12345
```

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

```
0.59747 0.26928 0.12623 0.85370 0.75281 0.78732 0.35292
```

```
0.89005 0.04774 0.86126
```

```
$ ./rand3 12345
```

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

```
0.59747 0.26928 0.12623 0.85370 0.75281 0.78732 0.35292
```

```
0.89005 0.04774 0.86126
```

No collision at the same time

```
$ ./rand3; ./rand3
```

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

```
0.80122 0.83184 0.33650 0.20144 0.35302 0.50180 0.76823  
0.98229 0.02290 0.04000
```

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

```
0.49672 0.36860 0.77740 0.69564 0.14145 0.34864 0.20053  
0.87646 0.32895 0.90694
```

In UNIX or Mac OS X system,

std::random_device looks up **/dev/urandom**
to draw random numbers

(You may try **\$ xxd /dev/urandom | head**)

Pseudo-random numbers in `python`

- Randomly sampling a single value

```
import random
print(random.uniform(0,1))
print(random.randint(0,1))
```

0.963765991622

0

Pseudo-random numbers in `python`

- Randomly sampling multiple values

```
import numpy as np
print(np.random.random(10))
print(np.random.randint(0,2,10))
```

```
[ 0.34386851  0.92403653  0.76340613  0.98810838  0.60215624  0.77208185
 0.3582923   0.28345326  0.41182332  0.20186387]
[1 1 1 1 0 1 1 0 1 1]
```

Pseudo-random numbers in python

- Setting seeds

```
import random
random.seed(12345)
print(random.uniform(0,1))
print(random.randint(0,1))
random.seed(12345)
print(random.uniform(0,1))
print(random.randint(0,1))
```

```
0.416619872545
0
0.416619872545
0
```

```
import numpy as np
np.random.seed(12345)
print(np.random.random(10))
np.random.seed(12345)
print(np.random.random(10))
```

```
[ 0.92961609  0.31637555  0.18391881  0.20456028  0.56772503  0.5955447
 0.96451452  0.6531771   0.74890664  0.65356987]
[ 0.92961609  0.31637555  0.18391881  0.20456028  0.56772503  0.5955447
 0.96451452  0.6531771   0.74890664  0.65356987]
```

Pseudo-random numbers in `python`

- A cryptographically secure way

```
import os
print(int(os.urandom(1).encode('hex'),16))
print(int(os.urandom(4).encode('hex'),16))
```

44

2164963379

Pseudo-random number generator in R

Random {base}

R Documentation

Random Number Generation

Description

`.Random.seed` is an integer vector, containing the random number generator (RNG) **state** for random number generation in **R**. It can be saved and restored, but should not be altered by the user.

`RNGkind` is a more friendly interface to query or set the kind of RNG in use.

`RNGversion` can be used to set the random generators as they were in an earlier **R** version (for reproducibility).

`set.seed` is the recommended way to specify seeds.

Usage

```
.Random.seed <- c(rng.kind, n1, n2, \dots)
```

```
RNGkind(kind = NULL, normal.kind = NULL)
```

```
RNGversion(vstr)
```

```
set.seed(seed, kind = NULL, normal.kind = NULL)
```

Arguments

`kind`

character or `NULL`. If `kind` is a character string, set **R**'s RNG to the kind desired. Use "default" to return to the **R** default

Default is "Mersenne-Twister"

Pseudo-random number generation in R

```
runif(10)
```

```
[1] 0.7950476 0.3546202 0.1492300 0.8132865 0.6745914 0.8969490 0.8494247  
[8] 0.1997723 0.5724314 0.3892715
```

```
rbinom(10,1,0.5)
```

```
[1] 1 1 0 1 0 1 1 0 0 1
```


Using seed in R

```
set.seed(12345)  
runif(10)
```

```
[1] 0.7209039 0.8757732 0.7609823 0.8861246 0.4564810 0.1663718 0.3250954  
[8] 0.5092243 0.7277053 0.9897369
```

```
set.seed(12345)  
runif(10)
```

```
[1] 0.7209039 0.8757732 0.7609823 0.8861246 0.4564810 0.1663718 0.3250954  
[8] 0.5092243 0.7277053 0.9897369
```

Pseudo-random numbers in Rcpp

```
#include <Rcpp.h>
#include <cstdio>
using namespace Rcpp;
// [[Rcpp::export]]
void foo() {
    // create a vector of random numbers and print
    NumericVector x = runif(10);
    for(int i=0; i < 10; ++i) printf("%.5lf ", x[i]);
    printf("\n");
    NumericVector y = rbinom(10, 1, 0.5);
    for(int i=0; i < 10; ++i) printf("%d ", (int)y[i]);
    printf("\n");
    // create a single random number and print
    printf("%.5lf\n", R::runif(0, 1));
    printf("%d\n", (int)R::rbinom(1, 0.5));
}
```

Rcpp::runif(10)

Rcpp::rbinom(10, 1, 0.5)

R::runif(0, 1)

```
foo()
```

```
0.45373 0.32675 0.96542 0.70748 0.64454 0.38983 0.69854 0.54406 0.22647 0.48456  
1 0 0 1 0 0 1 1 1 0  
0.78219  
0
```

```
set.seed(12345)  
foo()
```

```
0.72090 0.87577 0.76098 0.88612 0.45648 0.16637 0.32510 0.50922 0.72771 0.98974  
0 0 1 0 0 0 0 0 0 1  
0.45373  
0
```

```
set.seed(12345)  
foo()
```

```
0.72090 0.87577 0.76098 0.88612 0.45648 0.16637 0.32510 0.50922 0.72771 0.98974  
0 0 1 0 0 0 0 0 0 1  
0.45373  
0
```

Randomly sampling from a 'non-uniform' distribution

- Supposed that you only have a uniformly distributed random number generator `runif()`
- You need to implement your own `rnorm(mu, beta)`
- How?

Inverse transform sampling

- **Assumption**

- You have CDF $F(\cdot)$ and inverse CDF $F^{-1}(\cdot)$ of a known distribution.

- **Method**

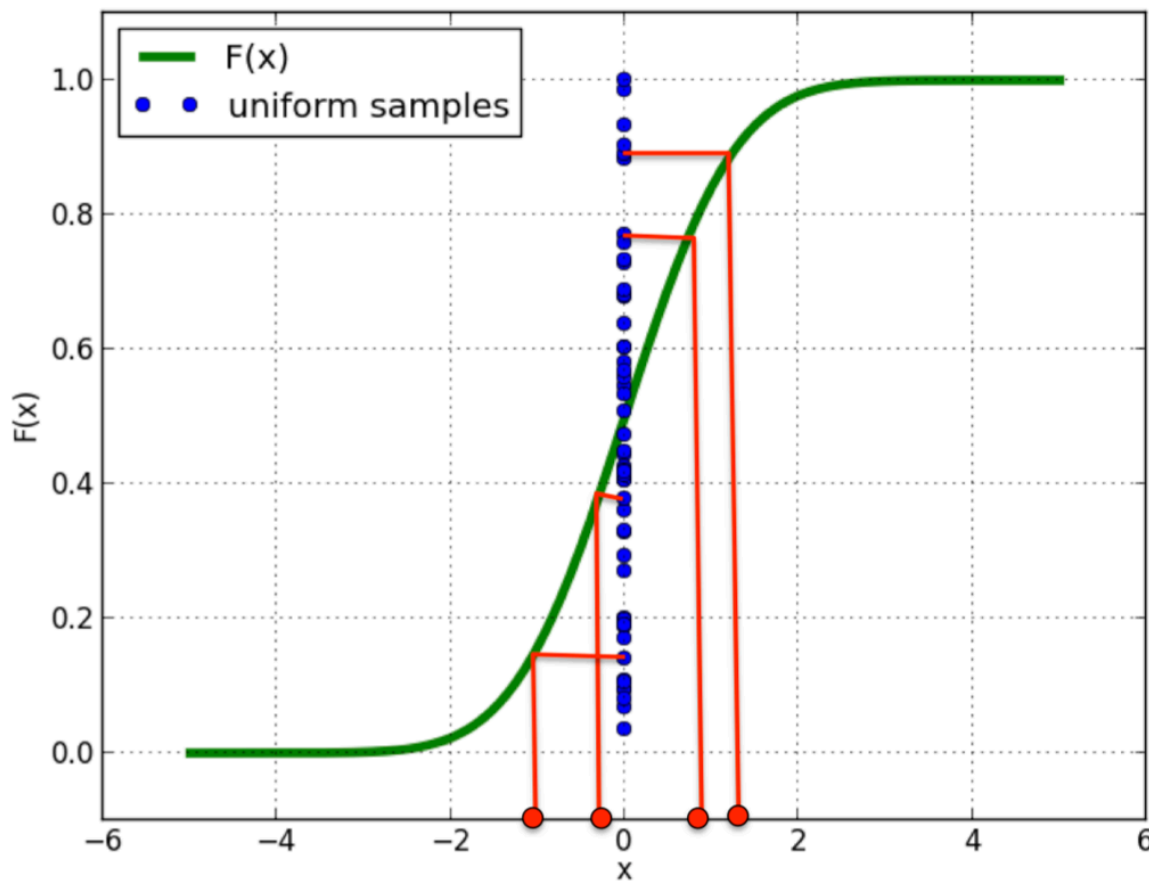
1. Sample a random variable U from Uniform(0,1)
2. Define $X = F^{-1}(U)$
3. Then X follows the distribution that defines $F(\cdot)$

- **Example : exponential distribution**

- Density: $f(x) = \frac{1}{\beta} e^{-\frac{x}{\beta}}$
 - CDF : $F(x) = 1 - \frac{1}{\beta} e^{-\frac{x}{\beta}}$
 - Transformation: $X = -\beta \log(1 - U)$

DIY : Try to prove why

Illustration of inverse transform sampling



Sampling from a **Gaussian** distribution

- **Using inverse transform sampling**
 - Inverse CDF must be known.
 - But normal distribution does not have a close form.
 - Hard to achieve without specialized library to compute inverse CDF
- **Using central limit theorem**
 - In principle, averaging A LOT of uniformly random samples should provide a normally distributed variables
 - But this is slow and imprecise.
- **Using a Box-Muller transformation**

Box-Muller Transformation

1. Sample two random variables U_1, U_2 from $Uniform(0,1)$
2. Define to following values
 - $R = \sqrt{-2\log U_1}$
 - $\Theta = 2\pi U_2$
 - $Z_0 = R \cos \Theta$
 - $Z_1 = R \sin \Theta$
3. Then Z_0, Z_1 are i.i.d. normally distributed random variables following $N(0,1)$

DIY : Can you prove it?

Sampling from a **complex** distribution

- What if CDF is not easy to obtain?
- How can we simulate from a mixture or normal?

$$f(x; \mu_1, \sigma_1^2, \mu_2, \sigma_2^2, \alpha) = \alpha f_{\mathcal{N}}(x; \mu_1, \sigma_1^2) + (1 - \alpha) f_{\mathcal{N}}(x; \mu_2, \sigma_2^2)$$

Sampling from a **complex** distribution

- What if CDF is not easy to obtain?

- How can we simulate from a mixture or normal?

$$f(x; \mu_1, \sigma_1^2, \mu_2, \sigma_2^2, \alpha) = \alpha f_{\mathcal{N}}(x; \mu_1, \sigma_1^2) + (1 - \alpha) f_{\mathcal{N}}(x; \mu_2, \sigma_2^2)$$

```
w <- rbinom(1,1,alpha)
y <- rnorm(1,mu1,sigma1)
z <- rnorm(1,mu2,sigma2)
x <- w*y + (1-w)*z
```

An example:
Not necessarily the best

$w \leftarrow \text{rbinom}(1,1,\alpha)$

$y \leftarrow \text{rnorm}(1,0,1)$

if ($w == 1$)

$y \leftarrow y * \sigma_1 + \mu_1$
else

$w = 0$

$y \leftarrow$

$y * \sigma_2 + \mu_2$

}

$w = 0 \rightarrow z$

$w = 1 \rightarrow y$

Sampling from a **bivariate** normal

$$\begin{pmatrix} x \\ y \end{pmatrix} \sim \mathcal{N} \left(\begin{pmatrix} \mu_x \\ \mu_y \end{pmatrix}, \begin{bmatrix} \sigma_x^2 & \sigma_{xy} \\ \sigma_{xy} & \sigma_y^2 \end{bmatrix} \right)$$

- Is this a correct implementation?

```
x <- rnorm(1,mu.x,sigma.x)
y <- rnorm(1,mu.y,sigma.y)
```

Use **conditional** distribution unless independent

$$\begin{pmatrix} x \\ y \end{pmatrix} \sim \mathcal{N} \left(\begin{pmatrix} \mu_x \\ \mu_y \end{pmatrix}, \begin{bmatrix} \sigma_x^2 & \sigma_{xy} \\ \sigma_{xy} & \sigma_y^2 \end{bmatrix} \right)$$

$$y|x \sim \mathcal{N} \left(\mu_y + \frac{\sigma_{xy}}{\sigma_x^2}(x - \mu_x), \sigma_y^2 \left(1 - \frac{\sigma_{xy}^2}{\sigma_x^2 \sigma_y^2} \right) \right)$$

```
x <- rnorm(1, mu.x, sigma.x)
y <- rnorm(1, mu.y + sigma.xy/sigma.x^2*(x-mu.x),
          sigma.y^2 - sigma.xy^2/sigma.x^2)
```


Sampling from a **multivariate** normal distribution

- **Problem setting**

- Given mean vector and positive-definite covariance matrix $\boldsymbol{\mu}, V$
- The goal is to randomly sample from $N(\boldsymbol{\mu}, V)$

- **One possible solution is..**

1. Sample \mathbf{x}_1 from the marginal distribution $N(\mu_1, V_{11})$.
2. Sample \mathbf{x}_2 from the conditional distribution $N(\mu_2, V_{12}V_{22}^{-1}(x_1 - \mu_1), V_{22} - V_{21}V_{11}^{-1}V_{12})$.
3. ... Sample \mathbf{x}_i from the subsequent conditional distribution

- **Is this computationally efficient? Why or why not?**

Leveraging matrix decomposition $V = UDU^T$

• Cholesky decomposition

- For a positive-definite matrix V , there exists a upper triangular matrix such at

$$V = U^T U$$

- Let $z \sim N(0, I_n)$ a *i.i.d.* random variables, then

$$x = U^T z + m \sim N(m, U^T U) = N(m, V)$$

- An example R code

$O(n^3)$

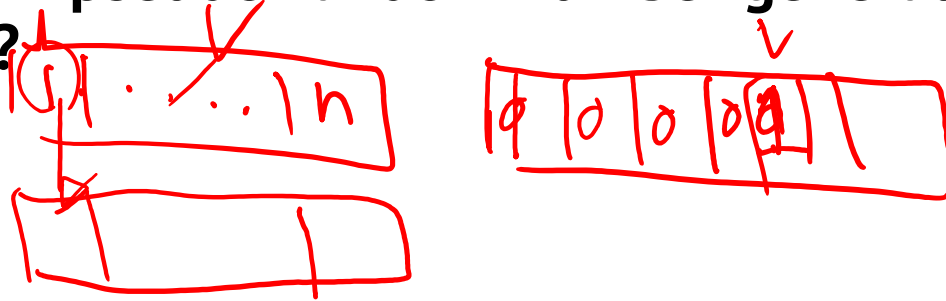
```

z <- rnorm(length(m))
U <- chol(V)
x <- m + t(U) %*% z
    
```

What's the time complexity of this method, compared to the previous one?

Algorithm to generate a random permutation

- Suppose that you have an array with n elements.
- The goal is to randomly shuffle the elements with equal probability for every possible permutation.
→ no extra space → use up to n random variables
- How can we achieve this, assuming that you have a good uniform pseudo-random number generator between 0 and 1?



Summary : Random numbers

- True vs. Pseudo- random numbers
- Different implementations of pseudo-random number generators across different languages
- Sampling a random number from a complex distribution
- Sampling correlating random numbers.