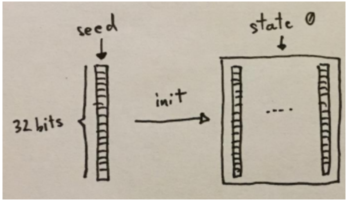
Algorithms and Data Structures

Random Numbers
Mersenne Twister & R

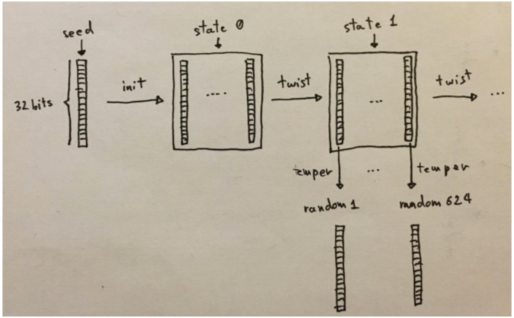


Learning goals

- Mersenne Twister algorithm
- Properties of Mersenne Twister
- Implementation in R

MERSENNE TWISTER

The **Mersenne Twister** is currently the most frequently used random number generator and was developed in 1997 by M. Matsumoto and T. Nishimura.



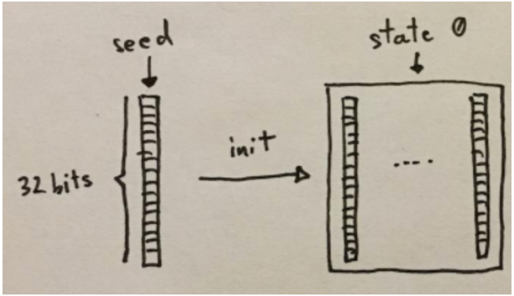
`https://www.cryptologie.net/article/331/how-does-the-mersennes-twister-work/`

MERSENNE TWISTER / 2

Note: Here, random numbers x_i are represented by w-bit vectors (usually $w = 32, 64$). To emphasize this, we write $\mathbf{x}_i$ (in bold).

Description of the algorithm:

1. **Initialization:** A seed $\mathbf{x}_0$ is set, and the first n values are calculated based on it (not described here). These values are not part of the final output.

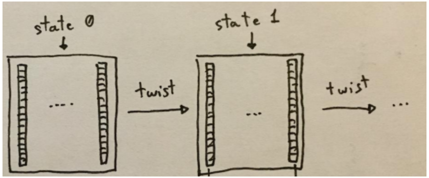


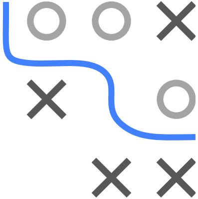
MERSENNE TWISTER / 3

❷ **Recursion:** Formally the following recursion formula is used

$$\mathbf{x}_{k+n} = \mathbf{x}_{k+m} \underbrace{\bigoplus}_{3.} \underbrace{(\mathbf{x}_k^l || \mathbf{x}_{k+1}^r)}_{1.} \underbrace{\mathbf{A}}_{2.}$$

- n: Degree of recurrence, "size" of blocks
- m: Integer $1 \leq m < n$
- $\mathbf{x}_k^l$, $\mathbf{x}_k^r$: Left and right part of the vector x_k
- $0 \leq c \leq w{-}1$ determines where the "left part" ends





MERSENNE TWISTER / 4

For ease of exposition, let $w = 4$, $c = 2$.

1. Concatenation:

$$
\begin{array}{r}
\mathbf{x}_k \\
\mathbf{x}_{k+1} \\
\mathbf{x}_{k+2}
\end{array}
\begin{pmatrix}
0 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 \\
1 & 1 & 1 & 1
\end{pmatrix}
\Rightarrow
\begin{array}{rl}
(x_k^l || x_{k+1}^r) & = (0, 1, 1, 0) \\
(x_{k+1}^l || x_{k+2}^r) & = (1, 0, 1, 1)
\end{array}
$$

MERSENNE TWISTER / 5

2. Multiplication by A:
We multiply with the so-called **Twist Matrix A**

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ a_3 & a_2 & a_1 & a_0 \end{pmatrix}$$

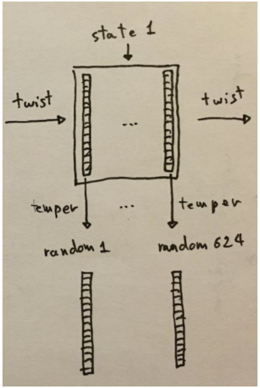
$$\begin{aligned} (0,1,1,0)\mathbf{A} &= (0,0,1,1) \\ (1,0,1,1)\mathbf{A} &= (0 \oplus a_3, 1 \oplus + a_2, 0 \oplus a_1, 1 \oplus a_0) \end{aligned}$$

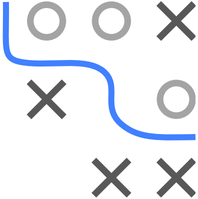
In summary $\quad \mathbf{xA} = \begin{cases} \text{shift}(\mathbf{x}), & \text{if last bit } x_0 = 0 \\ \text{shift}(\mathbf{x}) \oplus \mathbf{a}, & \text{if } x_0 = 1 \end{cases}$

3. XOR:
In the last step a bitwise XOR is calculated, e.g.

$$\mathbf{x}_{k+m} \bigoplus (0,0,1,1)$$

3. **Tempering:** In the last step, tempering is applied to the generated random numbers in order to improve their distribution properties.





MERSENNE TWISTER / 7

The following operations are performed:

$$
\begin{aligned}
\mathbf{x} \quad &\mapsto \quad \mathbf{y} := \mathbf{x} \oplus ((\mathbf{x} >> u) \text{ AND } \mathbf{d}) \\
\mathbf{y} \quad &\mapsto \quad \mathbf{y} := \mathbf{y} \oplus ((\mathbf{y} << s) \text{ AND } \mathbf{b}) \\
\mathbf{y} \quad &\mapsto \quad \mathbf{y} := \mathbf{y} \oplus ((\mathbf{y} << t) \text{ AND } \mathbf{c}) \\
\mathbf{y} \quad &\mapsto \quad \mathbf{z} := \mathbf{y} \oplus (\mathbf{y} >> l)
\end{aligned}
$$

where $\mathbf{x} >> u$ ($\mathbf{x} << u$) describes the bitwise "right"-shift ("left"-shift) by u and "AND" describes the bitwise "and".

This can be summarized as follows:

$$\mathbf{x} \mapsto \mathbf{z} = \mathbf{xT}.$$

Coefficients for MT19937: (standard implementation 32-bit)

- **w** word size (in number of bits): 32
- **n** degree of recurrence: 624
- **m** middle word, an offset used in the recurrence relation defining the series x: 397
- **c** separation point of one word, or the number of bits of the lower bitmask: 31
- **a** coefficients of the rational normal form twist matrix: $9908B0DF_{16}$
- **u, d, l** tempering masks/shifts: $(11, FFFFFFFF_{16}, 18)$
- **s, b** tempering masks/shifts: $(7, 9D2C5680_{16})$
- **t, c** tempering masks/shifts: $(15, EFC60000_{16})$

MERSENNE TWISTER / 9

Properties:

- Extremely long period of $2^{19937} - 1 \approx 4.3 \cdot 10^{6001}$ (so-called "Mersenne prime")
- All bits of the output sequence are uniformly distributed $\rightarrow$ thus the corresponding integer values are also uniformly distributed
- Low correlation of consecutive values
- Fast implementation by calculating *n* ($n = 624$ in MT19937) random numbers in one step
- Highly parallelizable

Further information:

- `www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/mt.pdf` (Original paper Mersenne Twister)
- `http://statweb.stanford.edu/~owen/mc/Ch-unifrng.pdf` (Lecture on PRNGs)

TRANSFORMATION TO THE INTERVAL $(0, 1)$

- In the procedures discussed, integers between 0 and $m - 1$ were generated.
- If real random numbers in the interval $(0, 1)$ need to be simulated, a simple division by m is sufficient.
- **Problem:** The random number can be 0, while for an "actual" $(0, 1)$ - uniformly distributed random variable U the following holds:

$$\mathbb{P}(U = 0) = 0$$

- This is often a problem in practical applications, e.g. with $\log(U)$.
- If a 0 is generated, the random number can be discarded or an error message is issued.
- Since modern congruential generators have a long period, this is unlikely.

PRNGS IN R

The Mersenne Twister is (currently) the default method in R with period $2^{19937}-1 \approx 10^{6001}$ and guaranteed uniform distribution in 623 dimensions. Seeds are 624 32-bit integers on top of the current position in this set. The `set.seed()` function generates a valid seed from a single integer value using the linear congruential generator with

$$m = 2^{32}, \qquad a = 69069, \qquad b = 1$$

There are a number of other generators available. Furthermore, the user can also specify her own generator as default.
`RNGversion('x.y.z')` can be used to set the random generators as they were in an earlier R version (for reproducibility) (Wichman-Hill up to 0.98, Marsaglia-Multicarry up to 0.00). 1.6.1). Initialization of the seed via time.

R: RANDOM NUMBER GENERATION

- `set.seed()` function

```
set.seed(123); u1 <- runif(100)
set.seed(123); u2 <- runif(100)
identical(u1, u2) # the same because of identical RNG status
## [1] TRUE
```

- `.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.

```
# default for "kind", "normal kind" and "sample kind"
RNGkind("default")
RNGkind()
## [1] "Mersenne-Twister" "Inversion"  "Rejection"
.Random.seed[1:3] # the default random seed is 626 integers
## [1] 10403 624 1858651209
```

R: RANDOM NUMBER GENERATION / 2

- Change default of kind ("Mersenne-Twister") to "Wichmann-Hill"

```
RNGkind("Wich")
RNGkind()
## [1] "Wichmann-Hill" "Inversion" "Rejection"
.Random.seed
## [1] 10400  5989  8337  9843
```

- Change methods depending on defaults in a specific R Version

```
RNGversion(getRversion()) # current version
RNGkind()
## [1] "Mersenne-Twister" "Inversion" "Rejection"
RNGversion("1.0.0") # first \texttt{R} version
## Warning in RNGkind("Marsaglia-Multicarry", "Buggy
Kinderman-Ramage", "Rounding"): buggy version of
Kinderman-Ramage generator used
## Warning in RNGkind("Marsaglia-Multicarry", "Buggy
Kinderman-Ramage", "Rounding"): non-uniform 'Rounding'
sampler used
```

PARALLEL COMPUTING

A complex topic is the application of random number generators for parallel computing, where a long calculation is split between several machines and processed in parallel. Usually, a "master" distributes the jobs to several "slaves". Initialization of seed using the two "standard" methods

- Time of day or
- Fixed given number

is not useful. Special algorithms for this purpose are provided e.g. in the `R` packages **rlecuyer** or **rstreams** (both use the same algorithm from L'Ecuyer).