Creating Unpredictability (Random numbers)

Problem Statement

Creating unpredictability is a crucial part of many different computer software processes, such as cryptography, simulations, machine learning, gaming, programming, scientific studies, and the list goes on. As we can create the unpredictability using random number generators, its worth mentioning that the random number generators can break down into two different types: true random number generators (TRNGs) and pseudo-random number generators (PRNGs). TRNGs are considered “true” because they utilize entropy from an external, non-computer program source, like the weather, atmospheric noise, or the amount of time you spend pressing down a key on your computer’s keyboard. However, there is not truly random source in existence, only weak random sources: sources that appear random, but for which we do not know the probability distribution of events. Pseudorandom numbers are numbers that are generated from weak random sources such that their distribution is “random enough”. The task is to create an application to produce pseudo-random numbers and study their features, such as their intervals (how many numbers does it produce before start repeating the same 'old' numbers) in different cases.

Students: Maimona Emmanuel Nzinga, Ren Koike, Thanaporn Waiprib, Aliia Bazarkulova, Zhyldyz Zhumabekova

Introductory Statement

Random numbers are an imperative part of many systems, including simulations, cryptography and much more. So the ability to produce values randomly, with no apparent logic and predictability, becomes a prime function. Since computers cannot produce values which are completely random, algorithms, known as **pseudorandom number generators (PRNG)** are used to accomplish this task.

Some of the most popular and highly used PRNGs are:

1. **Mersenne Twister:** Used as the default random number generator in Python, R, Excel, Matlab, Ruby and many more popular software systems.
2. **Linear Congruential Generator:** Used in C++ and Java
3. **Wichmann-Hill Generator:** Used in Excel and was the default in Python 2.2
4. **Park-Miller Generator**
5. **Middle Square Weyl Sequence**

Case of Study: Linear congruential generator (LCG)

A **linear congruential generator** (**LCG**) is an algorithm that yields a sequence of pseudo-randomized numbers calculated with a discontinuous piecewise linear equation. The method is the most common and oldest algorithm for generating pseudo-randomized numbers. The generator is defined by the recurrence relation:

$$X_{n+1} = (aX_n + c) \, mod \, m$$

where

X is the sequence of pseudo-random values, and

- $\boldsymbol{m > 0}$ (The modulus is positive),
- $\boldsymbol{0 < a < m}$ (the multiplier is positive but less than the modulus),
- $\boldsymbol{0 \leq c < m}$ (the increment is non negative but less than the modulus), and
- $\boldsymbol{0 \leq X0 < m}$ (the seed or starting value is non negative but less than the modulus).

are integer constants that specify the generator. If $c = 0$, the generator is often called a multiplicative congruential generator (MCG. If $c \neq 0$, the method is called a mixed congruential generator.

- The selection of the values for ***a**, **c**, **m***, and $\boldsymbol{X_0}$ drastically affects the statistical properties and the cycle length.
- The random integers X_i are being generated in [0, *m*-1]

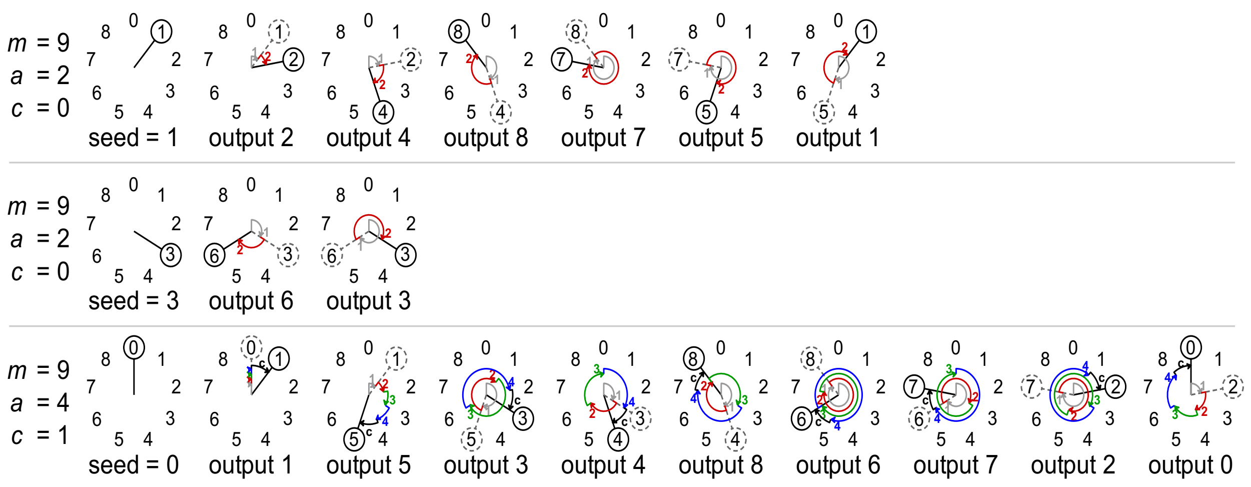


Figure 1 -Two modulo-9 LCGs show how different parameters lead to different cycle lengths.
By Cmglee - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=38637545

Cycle of LCG:

Definition: a sequence generates the same value as a previously generated value, then the sequence cycle.

The length of the cycle is called the **period** of the LCG. The LCG is said to achieve its **full period** if the cycle length is equals to m.

LCG has a long cycle for good choices of parameters a, m, c. Most computers (32-bit) has value for $m = 231 - 1 = 2{,}147{,}483{,}647$ represents the largest integer number.

Characteristics of a good Generator

A benefit of LCGs is that an appropriate choice of parameters results in a period which is both known and long. Although not the only criterion, too short a period is a fatal flaw in a pseudorandom number generator.

While LCGs can produce pseudorandom numbers which can pass formal tests for randomness, the quality of the output is extremely sensitive to the choice of the parameters *m* and *a*.

For example, *a* = 1 and *c* = 1 produces a simple modulo-*m* counter, which has a long period, but is obviously non-random.
Historically, poor choices for *a* have led to ineffective implementations of LCGs. A particularly illustrative example of this is **RANDU**, which was widely used in the early 1970s and led to many results which are currently being questioned because of the use of this poor LCG.
There are three common families of parameter choice:

- *m* prime, *c* = 0
- *m* a power of 2, *c* = 0
- *c* ≠ 0

Note: For the sake of simplicity and effectiveness we have chosen to adopt the last option. Therefore, the followed examples and studies are going to be based on that.

Hull-Dobell Theorem

When c ≠ 0, correctly chosen parameters allow a period equal to m, for all seed values. This will occur if and only if:
- m and c are relatively prime,
- $a - 1$ is divisible by all prime factors of m,
- $a - 1$ is divisible by 4 if m is divisible by 4.

This form may be used with any m, but only works well for m with many repeated prime factors, such as a power of 2; using a computer's word size is the most common choice. If m were a square-free integer, this would only allow $a \equiv 1 \ (mod\ m)$, which makes a very poor PRNG; a selection of possible full-period multipliers is only available when m has repeated prime factors.
Although the Hull–Dobell theorem provides maximum period, it is not sufficient to guarantee a good generator. For example, it is desirable for $a - 1$ to not be any more divisible by prime

factors of m than necessary. Thus, if m is a power of 2, then $a - 1$ should be divisible by 4 but not divisible by 8, i.e., $a \equiv 5 \ (mod\ 8)$.
Indeed, most multipliers produce a sequence which fails one test for non-randomness or another and finding a multiplier which is satisfactory to all applicable criteria is quite challenging. The spectral test is one of the most important tests.
Note that a power-of-2 modulus shares the problem as described above for $c = 0$: the low k bits form a generator with modulus 2^K and thus repeat with a period of 2^K; only the most significant bit achieves the full period. If a pseudorandom number less than r is desired, ⌊rX/m⌋ is a much higher-quality result than X mod r. Unfortunately, most programming languages make the latter much easier to write ($\boldsymbol{X} \,\%\, \boldsymbol{r}$), so it is the more commonly used form.
The generator is not sensitive to the choice of c, if it is relatively prime to the modulus (e.g., if m is a power of 2, then c must be odd), so the value $c = 1$ is commonly chosen.
The series produced by other choices of c can be written as a simple function of the series when $c = 1$. Specifically, if Y is the prototypical series defined by $Y_0 = 0$ and $Y_{n+1} = aY_{n+1}$ mod m, then a general series $\boldsymbol{X_{n+1} = aX_n + c \ mod \ m}$ can be written as an affine function of Y:

Example : (LCG Full Period Conditions)

Note: To apply the theorem, you must check if each of the three conditions holds for the generator.

m = 8 , a = 5 , c= 1

Cond-1.

c and m have no common factors other than 1: factors of $m = 8$ are (1, 2, 4, 8), since c $= 1$ (with factor 1) condition 1 is true.

Cond-2.

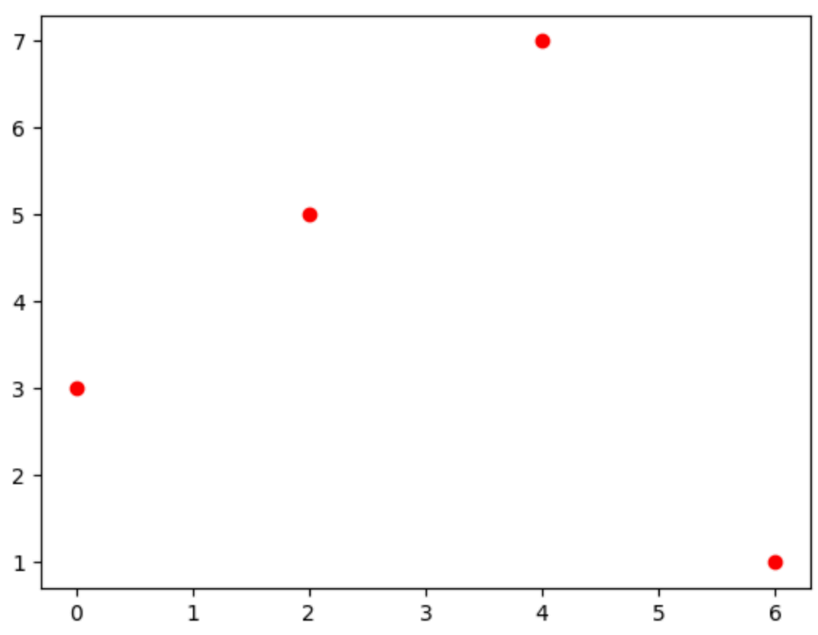
$(a - 1)$ is a multiple of every prime factors of m: The first few prime numbers are (2, 3, 5, 7).

The prime numbers (excluding 1), q, that divide $m = 8$ are (q = 1, 2). Since $a = 5$ and $(a - 1) = 4$, clearly q = 1 divides 4 and $q = 2$ divides 4. Thus, condition 2 is true.

Cond-3:

If 4 divides m, then 4 should divide $(a - 1)$.

Since $m = 8$, clearly 4 divides m. Also, 4 divides $(a - 1) = 4$. Thus, condition 3 holds.

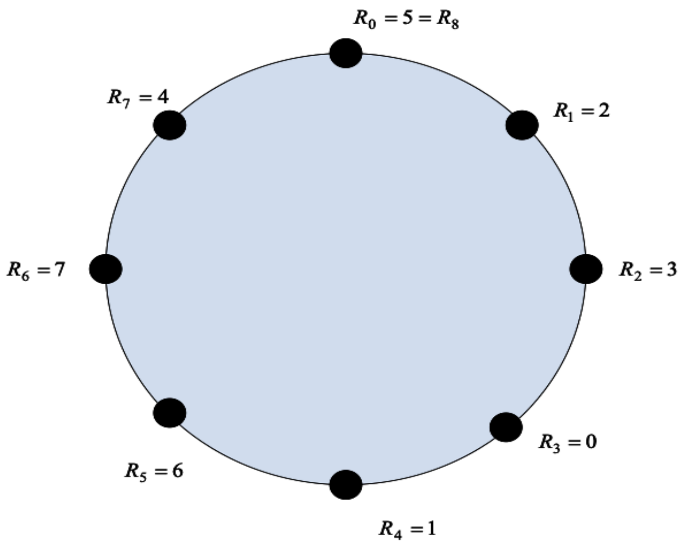


```
Effiency: 100%
Cycle Length:  8
Full period?  Yes
[0, 3, 2, 5, 4, 7, 6, 1]
```

Random Stream

Random Number Stream: is the subsequence of random numbers generated from a given seed.
A seed, e.g. $R_1 = 2$, defines a starting place in the cycle and thus a sequence.

The smaller the **period**, the easier to remember the random number streams, moreover, with large **m** hard to remember the stream.



Proper choice of parameters

For m a power 2, $m = 2^b$, and $c \neq 0$. Longest possible period $P = m = 2^b$ is achieved if c is relative prime to m and $a = 1 + 4k$, where k is an integer

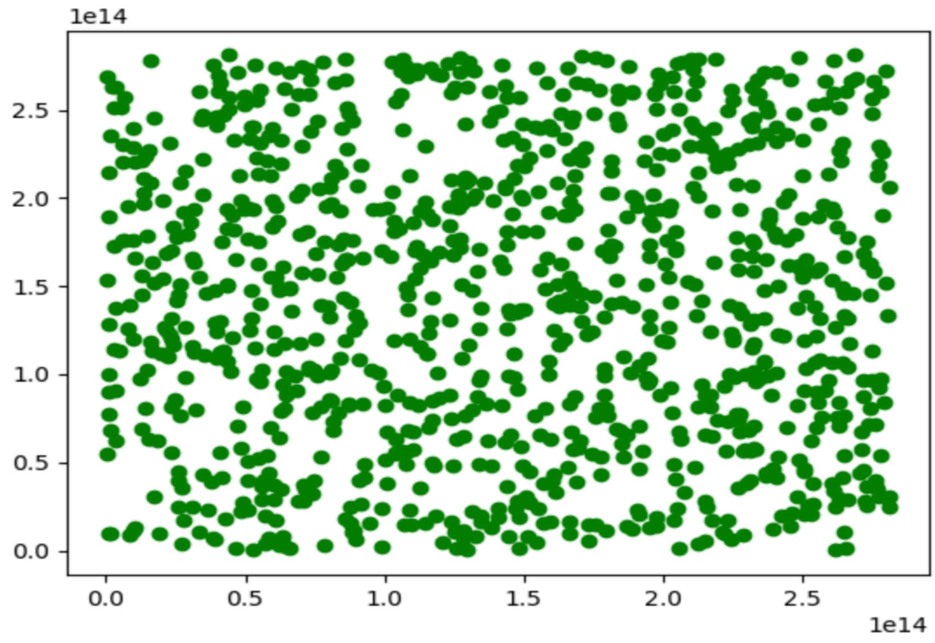
For m a power 2, $m = 2^b$, and $c = 0$. Longest possible period $P = m/4 = 2^{b-2}$ is achieved if the seed X_0 is odd and $a = 3 + 8k$ or $a = 5 + 8k$, for $k = 0,1, \dots$

For m a prime and $c = 0$. Longest possible period $P = m - 1$ is achieved if the multiplier a has property that smallest integer k such that $a^k - 1$ is divisible by m is $k = m - 1$.

Example : Using Parameters in common use

The following table lists the parameters of LCGs in common use, including built-in rand() functions in runtime libraries of various compilers. This table is to show popularity, not examples to emulate; many of these parameters are poor. Tables of good parameters are available

Example : $Java's\ java.util.Random$
$m = 2\text{^}48, a = 25214903917\ , c = 11$
$seed = 3\ (chosen)$
$n = 2000\ (chosen)$



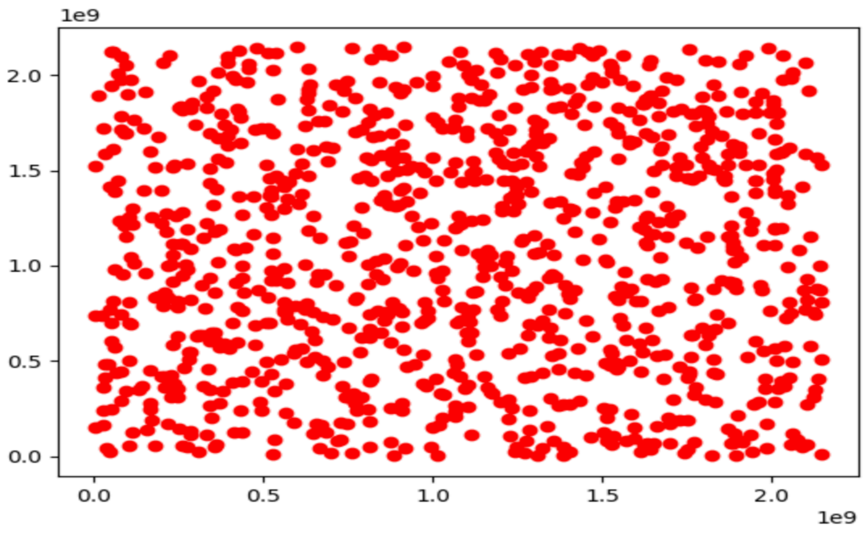
```
Effiency: 100%
Cycle Length:  2000
Full period?  Yes
```

Example : $glibc\ (used\ by\ GCC)$

$m = 2\hat{}31, a = 1103515245\ , c = 12345$

$seed = 3\ (chosen)$

$n = 2000\ (chosen)$



```
Effiency: 100%
Cycle Length:  2000
Full period?  Yes
```

Conclusion

The values produced by PRNGs are not truly random and depend on the initial value provided to the algorithm, known as the seed value. The property of a pseudorandom sequence being reproducible, given it's seed value is essential for its application in simulations, such as the Monte Carlo Simulation, where the system might need to be tested on the same sequence more than once.

To ensure that the values generated by the PRNG are as close to random as possible, several statistical tests including the Diehard tests, TestU01 series, Chi-Square test and the Runs test of Randomness are used. In this work we have chosen two study only the properties mentioned along the work and with the help of some visualization techniques and algorithms written in python. we have achieved our goal, that is, to create algorithms to produce pseudo-random numbers to enable us the study of their features, such as period or cycle in different cases.

Key takeaways:

- Even with generators that have been used for years, some of which still in use, are found to be inadequate.
- Even if generated numbers pass all the tests, some underlying pattern might have gone undetected.

References:

https://en.wikipedia.org/wiki/Linear_congruential_generator

https://www.mathematik.uni-ulm.de/stochastik/lehre/ss06/markov/skript_engl/node26.html
https://www.freecodecamp.org/news/random-number-generator/
https://faculty.ksu.edu.sa/sites/default/files/or441-lec05-random_number_generation_lcg_-sep_2017_0.pdf

Sources for Real random numbers:
http://www.comscire.com
http://www.randomnumbers.info/
http://www.random.org