Uniform Pseudo-Random Number Generation

Debraj Bose

January, 2018

Contents

1 Introduction

"The world is governed by chance. Randomness stalks us every day of our lives."

- Paul Auster

A random number is a number generated by a process, whose outcome is unpredictable, and which cannot be subsequentially reliably reproduced. This definition works fine provided that one has some kind of a black box - such a black box is usually called a random number generator - that fulfills this task.

Random numbers find lots of applications in gambling, statistical sampling, computer simulation, cryptography, completely randomized design, and other areas where producing an unpredictable result is desirable. Generally, in applications having unpredictability as the paramount, such as in security applications, hardware generators are generally preferred over pseudo-random algorithms, where feasible.

With this basic idea of random numbers, a short note on the history of random numbers may be looked at.

2 A Brief History of Random Numbers

"As an instrument for selecting at random, I have found nothing superior to dice. When they are shaken and tossed in a basket, they hurtle so variously against one another and against the ribs of the basket-work that they tumble wildly about, and their positions at the outset afford no perceptible clue to what they will be even after a single good shake and toss."

- Francis Galton (1890 issue of Nature)

The randomness so beautifully and abundantly generated by nature has not always been easy for us humans to extract and quantify. The oldest known dice were discovered in a 24th century B.C. tomb in the Middle East. More recently, around 1100 B.C. in China, turtle shells were heated with a poker until they cracked at random, and a fortune teller would interpret the cracks. Centuries after that, I Ching hexagrams for fortunetelling were generated with 49 yarrow stalks laid out on a table and divided several times, with results similar to performing coin tosses.

By the mid-1940s, RAND Corporation created a machine that would generate numbers using a random pulse generator and gathered the results into a book titled *A Million Random Digits with 100,000 Normal Deviates.* A similar machine, ERNIE, designed by the now-famous Bletchley Park WWII codebreaking team in the 1940s, was used to generate random numbers for the UK Premium Bond lottery. In 1951, randomness was finally formalized into a real computer, the Ferranti Mark 1, which shipped with a built-in random number instruction that could generate 20 random bits at a time using electrical noise. The feature was designed by Alan Turing.

But Turing's random number instruction created too much uncertainty in an environment that was already so unpredictable. We expect consistency from our software, but programs that used the instruction could never be run in any consistently repeatable way, which made them nearly impossible to test.

The idea of a pseudo-random number generator(PRNG) which is expressed as a deterministic math function was a solution to this problem. A PRNG can be called repeatedly to deliver a sequence of random numbers, but under the same initial conditions (if given the same initial "seed" value) it will always produce the same sequence.

John von Neumann developed a PRNG around 1946. His idea was to start with an initial random seed value, square it, and slice out the middle digits. However, no matter what seed value was used, the sequence would eventually fall into a short repeated cycle of numbers, like 8100, 6100, 4100, 8100, 6100, 4100, ...

Avoiding cycles is impossible when you are working with a deterministic random function whose subsequent value is based on the previous value. But what if the period of the cycle is really, really big, such that it practically didn't matter? The mathematician D. H. Lehmer made good progress toward this idea in 1949 with a linear congruential generator (LCG).

SSL was born around 1995, and its encryption scheme demanded a high quality PRNG. By 1997, a team at SGI created LavaRand, which was a webcam pointed at a couple of lava lamps on a desk. The image data from the camera was an excellent entropy source - a True Random Number Generator (TRNG) like Turing's - and it could generate 165Kb/s of random data. A founder of Autodesk, John Walker, who was intent on spreading randomness around the world created HotBits, a Random Numbers-as-a-Service app backed by a geiger counter that guarantees true quantum randomness. Random.org, created in 1998, provides free Truly Random Numbers. They now offer mobile apps for truly random coin flipping, dice rolling, card shuffling, and so on.

The Mersenne Twister, created in 1997 by Makoto Matsumoto and Takuji Nishimura is based on the idea of a linear feedback shift register (LFSR), which produces a deterministic sequence with very long cycle periods. It has a period of $2^{19937} - 1$, and it's still the default PRNG in many programming languages today. In 1999, Intel added an on-chip random number generator to its i810 server chipset. Finally, new servers had a good local source of randomness from thermal noise - a true random number generator (TRNG), which was great, but still not as fast as software PRNGs. This brought us to the cryptographically secure PRNG (CSPRNG). Open source hardware TRNGs such as REDOUBLER and Infinite Noise TRNG have emerged in recent years.

Today, there are many variants of these algorithms for different speed, space, and security requirements, and security experts are always looking for new ways to attack an implementation. But for most everyday uses, you can comfortably use the `/dev/random` special file on most operating systems, or the `rand()` function in most programming languages, to get a fast and endless stream of randomness.

3 What are truly random numbers?

"So much of life, it seems to me, is determined by pure randomness."

- Sidney Poitier

Random numbers are numbers that occur in a sequence such that two conditions are met:

- The values are uniformly distributed over a defined interval or set, and
- It is impossible to predict future values based on past or present ones.

These are the conditions of *true randomness.*

Truly random numbers are defined as those numbers that exhibit *true randomness* such as the time between "tics" from a Geiger counter exposed to a radioactive element, measuring atmospheric noise, thermal noise and other electromgnetic and quantum phenomena, etc.

4 What are pseudo-random numbers?

"Anyone who considers arithmetical methods of producing random digits is of course in a state of sin."

- John von Neumann

Broadly speaking, pseudo-random numbers are those numbers that have the *appearance* of randomness but nevertheless, exhibit a specific, repeatable pattern.

Pseudo-random numbers are a set of values or elements that is statistically random, but is derived from a known starting point (called a *seed*) and is typically repeated over and over. They provide necessary values for processes that require randomness, such as creating test signals or for synchronizing sending and receiving devices in a spread spectrum transmission. They are called "pseudo" random, because the algorithm can repeat the sequence, and the numbers are thus not entirely random. Numbers calculated by a computer through a deterministic process, cannot, by definition, be random.

5 What are the desirable properties of pseudo-random numbers?

1. **Uncorrelated Sequences** - The sequences of random numbers should be serially uncorrelated.

2. **Long Period** - The generator should be of long period. Ideally, the generator should not repeat; practically, the repetition should occur only after the generation of a very large set of random numbers.

3. **Uniformity** - The sequence of random numbers should be uniform, and unbiased. That is, equal fractions of random numbers should fall into equal "areas" in space. Eg. if random numbers on [0,1) are to be generated, it would be poor practice were more than half to fall into [0, 0.1), presuming the sample size is sufficiently large.

4. **Efficiency** - The generator should be efficient. There should be low overhead for massively parallel computations.

6 How are pseudo-random numbers different from truly random numbers?

"The generation of random numbers is too important to be left to chance."

- Robert R. Coveyou

- Truly random numbers are generated by measuring certain aspects of physical phenomena that is expected to be random such as cosmic background radiation or radioactive decay and then compensates for possible biases whereas pseudo random numbers are generated by computational algorithms that can produce long sequences of apparently random results, which are in fact completely determined by a shorter initial value, known as a seed value or key.

- The speed at which truly random numbers can be generated from natural sources are dependent on the underlying phenomena being measured. Thus they are rate-limited. Pseudo-random number generators are not rate-limited and large chunks of random numbers can be generated easily.

- We cannot restart the truly random number generator and rerun the simulation whereas the entire seemingly random sequence can be reproduced if the seed value is known in a pseudo-random number generator.

While a *pseudo-random* number generator based solely on deterministic logic can never be regarded as a *true* random number source in the purest sense of the word, in practice they are generally sufficient even for demanding security-critical applications.

7 Applications of uniform random numbers: Motivation for why we need random numbers

One of the earliest applications of random numbers was in the computation of integrals. Let $g(x)$ be a function and suppose we wanted to compute θ where

$$\theta = \int_0^1 g(x)dx \tag{1}$$

To compute the value of θ, note that if U is uniformly distributed over (0,1) (i.e., with probability density function (pdf) $f(x) = 1$, $0 < x < 1$), then we can express θ as

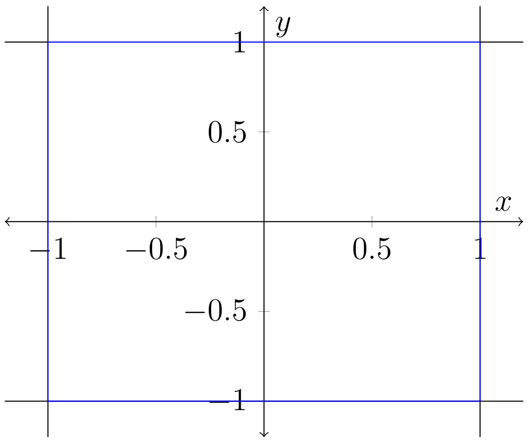
$$\theta = E[g(U)] \tag{2}$$

If $U_1, ..., U_k$ are independent Uniform (0,1) random variables, it thus follows that the random variables $g(U_1), ..., g(U_k)$ are independent and identically distributed random variables having mean θ. Therefore, by the Strong Law of Large Numbers, it follows that, with probability 1,

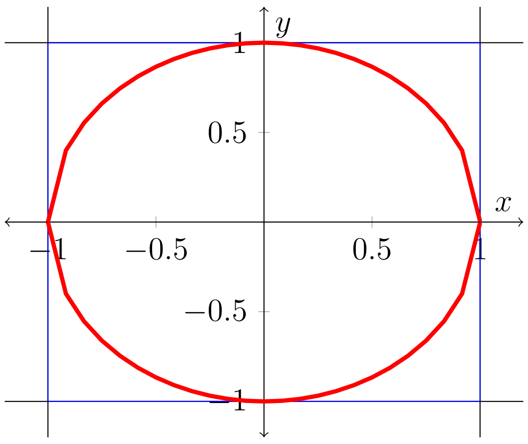
$$\sum_{i=1}^{k} \frac{g(U_i)}{k} \rightarrow E[g(U)] = \theta \quad as \quad k \rightarrow \infty \tag{3}$$

Hence we can approximate θ by generating a large number of random numbers u_i and taking as our approximation the average value of $g(u_i)$. This approach to approximating integrals is called the ***Monte Carlo*** approach.

An application of the above theory is the estimation of π. Suppose that the random vector (X, Y) is uniformly distributed in the square of area 4 centred at the origin, i.e., it is a random point in the following region bordered by the blue lines.



Let us consider the probability that the random point in this square is contained within the inscribed red circle of radius 1.



Since (X, Y) is uniformly distributed in the square, it follows that

$$P[(X,Y)\,is\,in\,the\,circle] = P[X^2 + Y^2 \leq 1] = \frac{Area\,of\,the\,circle}{Area\,of\,the\,square} = \frac{\pi}{4} \tag{4}$$

Hence, if we generate a large number of random points in the square, the proportion of points that fall within the circle will be approximately $\pi/4$. Now, if X and Y were independent and both were uniformly distributed over (-1,1), their joint density would be

$$f(x,y) = f(x)f(y) = \frac{1}{2}.\frac{1}{2} = \frac{1}{4}, \quad -1 \leq x \leq 1, \quad -1 \leq y \leq 1 \tag{5}$$

Now, if U is uniform on (0,1) then $2U$ is uniform on (0,2), and so $2U - 1$ is uniform on (-1,1). Therefore, if we generate random numbers U_1 and U_2, set $X = 2U_1 - 1$ and $Y = 2U_2 - 1$ and define

$$I = \begin{cases} 1 & \text{if } x^2 + y^2 \leq 1 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

then

$$E[I] = P[X^2 + Y^2 \leq 1] = \frac{\pi}{4} \tag{7}$$

Hence we can estimate $\pi/4$ by generating a large number of pairs of random numbers u_1, u_2 and estimating $\pi/4$ by the fraction of pairs for which $(2u_1 - 1)^2 + (2u_2 - 1)^2 \leq 1$.

This has been illustrated by using the Linear Congruential Generator (LCG). We shall explain the intricacies of the LCG later. At present, we shall proceed with the knowledge that the LCG is a PRNG that generates uniform random numbers. The following table gives us estimates of π for various sample sizes of pairs of uniform random numbers.

Sample Size	Estimate of π
10^2	3.04
10^3	3.192
10^4	3.1264
10^5	3.14424
10^6	3.141272
10^7	3.141543

Thus, by generating uniform random numbers, we gradually approach the true value of π as we increase the sample size. Another application of uniform random numbers is generation of values of random variables from arbitrary distributions. With this ability to generate arbitrary random variables we will be able to simulate a probability system, i.e., we will be able to generate according to the specified probability laws of the system, all the random quantities of this system as it evolves over time. For example, pairs of independent standard normally distributed random variables can be generated from pairs of random observations from Uniform (0,1) distribution using the Box-Müller transformation.

8 States, periods, seeds and streams of pseudo-random number generators

The typical random number generator provides a function with a name such as `rand`, that can be invoked via an assignment like `x = rand()`. The result is then a simulated draw from the Uniform (0,1) distribution. Here, we will treat `rand` as a black box and postpone looking inside it until the next section.

The function `rand` maintains a `state` variable. What generally goes on inside `rand` is `state = update(state)` followed by `return(state)`, i.e., after modifying the state, it returns some function of the new state. However, one thing becomes very clear. Because the state variable must have a finite size, the random number generator cannot go on forever without eventually revisiting a state it was in before. At that point it will start repeating values it already delivered.

Suppose that we repeatedly call $x_i =$ `rand()` for $i \geq 1$ and that the state of the generator when x_{i_0} is produced is the same as it was when x_{i_0-P} was produced. Then $x_i = x_{i-P}$ holds for all $i \geq i_0$. From i_0 onwards, the generator is a deterministic cycle with **period** P. We will be simplifyng things and supposing that the random number generator has a fixed period P and that $x_{i+P} = x_i$ holds for all i.

It is pretty clear that a small value of P makes for a very poor simulation of random behaviour. Random number generators with very large periods are preferred. One general guideline is that we will not use more than $\sqrt{P}$ random numbers from a given generator. Hellekalek and L'Ecuyer (1998) describe how an otherwise good linear congruential generator starts to fail tests when about $\sqrt{P}$ numbers are used. Some linear congruential generators have $P = 2^{32} - 1$, which is far too small for Monte Carlo.

A random **seed** (or seed state, or just seed) is a number (or vector) used to initialize a pseudo-random number generator, i.e., it is the initial state of the generator corresponding to which x_0 would be produced. We can make a random number generator repeatable by intervening and setting its seed before using it. When we don't set the seed, random seeds are often generated from the state of the computer system (such as the time), a cryptographically secure pseudorandom number generator or from a hardware random number generator. The choice of a good random seed is crucial in the field of computer security. When a secret encryption key is pseudorandomly generated, having the seed will allow one to obtain the key. High entropy is important for selecting good random seed data.

In moderately complicated situations, we want to have two or more **streams** of random numbers. Each stream should behave like a sequence of independent Uniform (0,1) random variables. In addition the streams need to appear as if they are statistically independent of each other. For example, we might use one stream of random numbers to simulate customer arrivals and another to simulate their service times.

9 Uniform Pseudo-Random Number Generators

"Random numbers should not be generated with a method chosen at random."

- Donald Knuth

There are essentially two categories of algorithms used to produce pseudo-random strings of bits:

- **Heuristic Pseudo Random Number Generators** (*Heuristic PRNGs*) - These are algorithms designed to produce pseudo random strings of bits. The output should be, but often isn't, indistinguishable from random output, because it is completely determined by an initial value, called the PRNG's seed. Heuristic PRNGs are central in applications such as simulations (eg. for the Monte Carlo method), electronic games (eg. for procedural generation), etc.
- **Cryptographically Secure Pseudo Random Number Generators** (*CSPRNGs*) - These are generators designed with the same goal as above, but with the additional property that given some arbitrarily long (but obviously less than the period) sequence of output from the generator, it should be computationally infeasible to determine the next bit with greater certainty than 1/2. A CSPRNG must satisfy all the statistical randomness tests a statistical PRNG does, but it also needs to be unpredictable. It is designed to resist attempts by a human attacker to predict its next output; it should be hard to tell it from a truly random sequence even if the attacker knows the algorithm used to make it. As the name suggests, CSPRNGs have many cryptographical applications such as key generation, nonces, one-time pads (OTP), etc.

10 Some Heuristic PRNGs

10.1 Linear Congruential Generators (LCGs)

The majority of modern random number generators are based on simple recursions using modular arithmetic. A well-known example is the **linear congruential generator** (*LCG*).

The generator is defined by the recurrence relation

$$X_i = a_0 + a_1 X_{i-1} \quad mod \quad m \tag{8}$$

where $\{X_i\}$ is the sequence of pseudo-random values,
m, $m > 0$: *modulus*,
a_1, $0 < a_1 < m$: *multiplier*,
a_0, $0 \leq a_0 < m$: *increment*, and
X_0, $0 \leq X_0 < m$: *seed value*
are integer constants that specify the generator.

This method produces a sequence of integer values $X_i \in \{0, 1, ..., m-1\}$, i.e., integers modulo m. With good choices of the constants, a_j and m, the X_i's can simulate independent random integers modulo m. The LCG takes on atmost m different values and so has period $P \leq m$. Let us look at a few simple examples to see how these pseudo-random number sequences behave and what their periods are like.

Let $a_0 = 3, a_1 = 5, m = 16$ and the seed $X_0 = 9$. The pseudo-random number sequence is as follows:

$$\mathbf{9}, 0, 3, 2, 13, 4, 7, 6, 1, 8, 11, 10, 5, 12, 15, 14, \mathbf{9}, 0, 3, 2, 13, ... \tag{9}$$

This sequence has period $P = 16$, the maximum period possible. Now, let $a_0 = 7, a_1 = 3, m = 16$ and the seed $X_0 = 9$. The pseudo-random number sequence is as follows:

$$\mathbf{9}, 2, 13, 14, 1, 10, 5, 6, \mathbf{9}, 2, 13, 14, 1, 10, 5, 6, \mathbf{9}, 2, 13, 14, ... \tag{10}$$

This sequence has period $P = 8 < m$. Thus, for different choices of a_0, a_1, we get pseudo-random number sequences of different periods for the same m. A commonly used choice of m is $m = 2^k$, where most often, $k = 32$ or 64 is chosen to produce a particularly efficient LCG. If, in addition to such a choice of m, we choose $a_0 \neq 0$, ***Hull-Dobell Theorem*** states that a period equal to m will occur for all seed values if and only if the following three conditions are satisfied:

1. m and a_0 are relatively prime,
2. $a_1 - 1$ is divisible by all prime factors of m, and
3. $a_1 - 1$ is divisible by 4 if m is divisible by 4.

It is to be noted that the LCG in (9) with maximum possible period satisfies Hull-Dobell Theorem while the LCG in (10) does not satisfy the third condition of the theorem.

Now, let us go back to the estimation of π in Section 7. We had to initialise the LCG with choices of a_0, a_1 and m. Also, an initial seed was required. We picked a popular choice of these values: $a_0 = 1013904223, a_1 = 1664525, m = 2^{32}$. The seed is obtained from the system time so we will get a different "naturally occurring" seed every time we run the code to estimate π. To get Uniform(0,1) random numbers, divide the pseudo-random number sequence $\{X_i\}$ by $m = 2^{32}$. This LCG has a very long period ($= 2^{32}$, which is the maximum possible period since it satisfies Hull-Dobell theorem) and passes tests for randomness. For eg., with seed $X_0 = 2576389$, let us generate 1000000 pseudo-random numbers using this LCG and divide all these numbers by $m = 2^{32}$. We obtain mean and variance of this sequence as 0.5002262 and 0.08328703 which are, for all practical purposes, extremely close to the mean and variance of the Uniform(0,1) random variable, i.e., 0.5 and 0.083333 respectively. In general, with these particular choices of a_0, a_1 and m, for any seed, the mean and variance of the resulting sequence are extremely close to their Uniform(0,1) counterparts. This is not a very definitive test for randomness so we have even applied the ***Runs test*** to this sequence to check for autocorrelation. The Runs test had a p-value of 0.7987, which means the null hypothesis that the sequence is random is accepted.

10.2 Multiplicative Congruential Generators (MCGs)

A **Multiplicative Congruential Generator** (*MCG*) is basically an LCG with $a_0 = 0$, i.e., the generator is defined by the recurrence relation

$$X_i = a_1 X_{i-1} \quad mod \quad m \tag{11}$$

where $\{X_i\}$ is the sequence of pseudo-random values,
m, $m > 0$: *modulus*,
a_1, $0 < a_1 < m$: *multiplier*, and
X_0, $0 \leq X_0 < m$: *seed value*
are integer constants that specify the generator.

An LCG is generally slower than an MCG. However, for the MCG, we cannot allow $x_i = 0$ because then the sequence stays at 0. As a result, $P \leq M - 1$ for the MCG.

Choosing m to be a power of 2, most often $m = 2^{32}$ or $m = 2^{64}$, produces a particularly efficient MCG, because this allows the modulus operation to be computed by simply truncating the binary representation. For eg., 1101011 *mod* $2^3 (= 1000) = 011$ since we divide 1101011 into two parts - 1101|**011**, the first part being the quotient on dividing by 8 and the second the remainder, which is what we require. This form has maximal period $m/4$ achieved if $a_1 \equiv 3$ or $a_1 \equiv 5$ *mod* 8. The initial state X_0 must be odd. It can be shown that this form is equivalent to an LCG with $m = m/4$ and $a_0 \neq 0$ with parameters satisfying Hull-Dobell theorem.

A more serious issue with the use of a power-of-two modulus is that the low bits have a shorter period than the high bits. The lowest-order bit of X_i never changes (X_i is always odd), and the next two bits (let us call them bit 1 and bit 2) alternate between two states. If $a_1 \equiv 5$ *mod* 8, bit 1 never changes and bit 2 alternates. The following example with $a_1 = 93 \equiv 5 \; mod \; 8, m = 2^8 = 256, X_0 = 19$ (odd) illustrates this fact.

Output of MCG in binary with $a_1 = 93 \equiv 5 \; mod \; 8, m = 2^8 = 256, X_0 = 19$ (odd)							
00010011	11100111	11101011	01011111	10000011	10010111	11011011	10001111
11110011	01000111	11001011	10111111	01100011	11110111	10111011	11101111
11010011	10100111	10101011	00011111	01000011	01010111	10011011	01001111
10110011	00000111	10001011	01111111	00100011	10110111	01111011	10101111
10010011	01100111	01101011	11011111	00000011	00010111	01011011	00001111
01110011	11000111	01001011	00111111	11100011	01110111	00111011	01101111
01010011	00100111	00101011	10011111	11000011	11010111	00011011	11001111
00110011	10000111	00001011	11111111	10100011	00110111	11111011	00101111
00010011	11100111	11101011	01011111	10000011	10010111	11011011	10001111

If $a_1 \equiv 3 \ mod \ 8$, bit 2 never changes and bit 1 alternates. The following example with $a_1 = 27 \equiv 3 \ mod \ 8, m = 2^8 = 256, X_0 = 201$ (odd) illustrates this fact.

Output of MCG in binary with $a_1 = 27 \equiv 3 \ mod \ 8, m = 2^8 = 256, X_0 = 201$ (odd)

11001001	00110011	01100001	00111011	00111001	00000011	01010001	10001011
10101001	11010011	01000001	11011011	00011001	10100011	00110001	00101011
10001001	01110011	00100001	01111011	11111001	01000011	00010001	11001011
01101001	00010011	00000001	00011011	11011001	11100011	11110001	01101011
01001001	10110011	11100001	10111011	10111001	10000011	11010001	00001011
00101001	01010011	11000001	01011011	10011001	00100011	10110001	10101011
00001001	11110011	10100001	11111011	01111001	11000011	10010001	01001011
11101001	10010011	10000001	10011011	01011001	01100011	01110001	11101011
11001001	00110011	01100001	00111011	00111001	00000011	01010001	10001011

Also, observe that bits 3 and 4 alternate with periods 4 and 8 respectively.

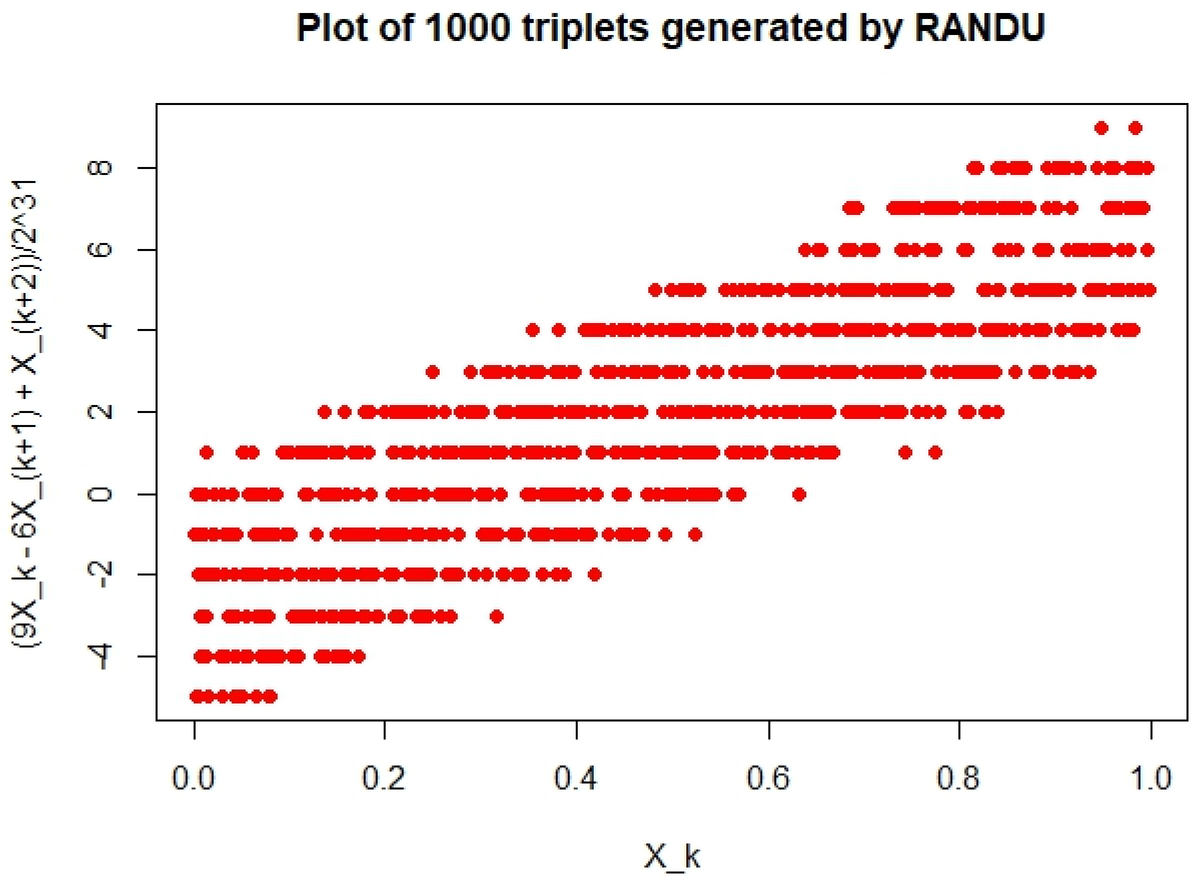
A very famous example of an MCG is ***RANDU***. Developed by IBM, it is widely considered to be one of the most ill-conceived random number generators ever designed. It is defined by taking $a_1 = 65539 (= 2^{16} + 3)$ and $m = 2^{31}$. If one generates a sequence of pseudo-random numbers using RANDU and computes sample moments, the results look good with the k^{th} order sample moment $\approx 1/(k+1)$. However, this is not sufficient. The numbers must be free of correlations with each other and RANDU fails badly in this regard.

Let us generate triplets of numbers X_k, X_{k+1}, X_{k+2}, and plot them in 3-dimensional space. Ideally, they should fill a cube of unit side uniformly. However, it turns out that all the triplets of random numbers generated by RANDU lie on only 15 planes. In fact, the combination $(9X_k - 6X_{k+1} + X_{k+2})/2^{31}$ is an integer. We have

$$X_{k+1} = (2^{16} + 3)X_k, \tag{12}$$

$$\begin{aligned} X_{k+2} &= (2^{16} + 3)X_{k+1} \\ &= (2^{16} + 3)^2 X_k \\ &= (2^{32} + 6 \times 2^{16} + 9)X_k \\ &= (2^{32} + 6 \times (2^{16} + 3) - 9)X_k \\ &= 6X_{k+1} - 9X_k \quad mod \quad 2^{31} \end{aligned} \tag{13}$$

since $2^{32} \ mod \ 2^{31} = 0$. Hence, $9X_k - 6X_{k+1} + X_{k+2}$ is a multiple of 2^{31} and thus, on dividing by 2^{31}, we obtain an integer. In fact, this integer is restricted to values between -5 and 9. Let us plot the $(9X_k - 6X_{k+1} + X_{k+2})/2^{31}$ values of 1000 triplets versus $X_k/2^{31}$ values generated by RANDU with seed $X_0 = 314159$. The plot clearly shows that all triplets of points generated by RANDU lie on one of the 15 planes $9x - 6y + z = m$, where $m = -5, -4, ..., 8, 9$.



10.3 Multiple Recursive Generators (MRGs)

A generalisation of the MCG is the **Multiple Recursive Generator** (*MRG*), which is defined by the recurrence relation

$$X_i = a_1 X_{i-1} + a_2 X_{i-2} + ... + a_k X_{i-k} \quad mod \quad m, \qquad k \geq 1 \tag{14}$$

where $\{X_i\}$ is the sequence of pseudo-random values,
m, $m > 0$: *modulus*,
a_i, $0 < a_i < m$: *sequence of multipliers*, and
$X_0, X_1, ..., X_{k-1}$, $0 \leq X_i < m$, $i = 0, 1, ..., k-1$: *seed values (initial states)*
are integer constants that specify the generator.

The MRG will start to repeat as soon as k consecutive values $x_i, ..., x_{i+k-1}$ duplicate some previously seen consecutive k-tuple. There are m^k of these, and just like with the MCG, the state with k consecutive $0's$ cannot be allowed. Therefore, $P \leq m^k - 1$. To show that the period is greater than m, let us take a simple example: $k = 3, \underline{a} = (a_1, a_2, a_3) = (13, 4, 5), m = 16 = 2^4$ and seed $X_0 = 2, X_1 = 3, X_2 = 11$, we get the following pseudo-random number sequence.

$$\mathbf{2,\ 3,\ 11}, 5, 12, 7, 4, 12, 15, 7, 3, 14, 5, 8, 2, 3, 7, 1, 8, 15, 8, 12, 7, 3, 15, 2, 5, 4, 2, 3, 3, 13, 4, 7, 12, 12, 15, 15, 11, 6, 5, 0, 2, 3, 15, 9, 0, 15, 0, 12, 7, 11, 7, 10, 5, 12, \mathbf{2,\ 3,\ 11}, 5, 12, 7, ... \tag{15}$$

Observe that this sequence has period $P = 56 = 8 \times 7 = 2^{4-1} \times (2^k - 1)$. We have tinkered with various choices of parameters to get the maximum possible period. The results we got are stated here. For the choices of parameters, $k = 3, \underline{a} = (5, 4, 19), m = 32 = 2^5$ and seed $X_0 = 1, X_1 = 17, X_2 = 20$, we get a pseudo-random numer sequence of period $P = 112 = 16 \times 7 = 2^{5-1} \times (2^k - 1)$. For the choices of parameters, $k = 4, \underline{a} = (29, 22, 14, 5), m = 32 = 2^5$ and seed $X_0 = 1, X_1 = 17, X_2 = 20, X_3 = 9$, we obtain a pseudo-random number sequence of period $P = 240 = 16 \times 15 = 2^{5-1} \times (2^k - 1)$. Thus, we observe the following pattern:

For an MRG with modulus $m = 2^p$, the maximum possible period that can be attained is $P = 2^{p-1} \times (2^k - 1)$ for appropriately chosen $\underline{a}$ and seed values.

However, this turns out to be false. Let m be a prime number and $k \geq 1$ be an integer. Let us construct a ***Galois field of order*** m^k, which is denoted by $GF(m^k)$, as follows:

- Take F to be the field of residues modulo m with addition and multiplication modulo m as the binary operations.
- Let $F[x]$ be the set of all polynomials in the variable x with coefficients from the field F, i.e., $F[x] = \{\ c_0 + c_1x + ... + c_kx^k : k$ is a non-negative integer, $c_0, ..., c_k \in F\}$.
- Select an irreducible polynomial $g(x) \in F[x]$ of degree k. (A non-zero polynomial $g(x) \in F[x]$ is said to be *irreducible* over F if for any $f_1(x), f_2(x) \in F[x]$ with $f(x) = f_1(x)f_2(x)$, either $deg(f_1(x)) = 0$ or $deg(f_2(x)) = 0$)
- Then, $F[x]/g(x)$ provides the required Galois field.

For example, let us construct a Galois field of order 2^3. The polynomial $f(x) = x^3 + x + 1$ is seen to be irreducible over $GF(2)$. Thus, the desired Galois field is given by $F[x]/f(x)$. The elements of this field can be represented by $0, 1, x, x+1, x^2, x^2+1, x^2+x, x^2+x+1$. It is interesting to observe that the successive powers of x in this field generate all the non-zero elements of the field as $x^1 = x, x^2 = x^2, x^3 = x+1, x^4 = x^2+x, x^5 = x^2+x+1, x^6 = x^2+1, x^7 = 1$. An element of a field with this property is called a *primitive element* of the field. An irreducible polynomial $f(x)$ over $GF(p)$ of degree k is said to be a *primitive polynomial* if x is a primitive element in the field $GF(m^k) = F[x]/f(x)$. Some primitive polynomials are listed below.

$GF(m^k)$	Primitive Polynomial over $GF(m)$
2^2	$x^2 + x + 1$
2^3	$x^3 + x + 1$
2^4	$x^4 + x + 1$
2^5	$x^5 + x^2 + 1$
3^2	$x^2 + x + 2$
3^3	$x^3 + 2x + 1$
5^2	$x^2 + x + 2$
7^2	$x^2 + x + 3$

Tables of primitive polynomials over various Galois fields are available in *Tables of Finite Fields by Alanen and Knuth (1964).*

Let us define the characteristic polynomial of an MRG as $f(x) = x^k - a_1 x^{k-1} - ... - a_k$. The maximum period of an MRG is $m^k - 1$, which is achieved if and only if its characteristic polynomial $f(x)$ is a primitive polynomial over $GF(m)$. Let us illustrate this with an MRG with $m = 7, k = 2$. The primitive polynomial over $GF(7)$ is $f(x) = x^2 + x + 3$. Now, we choose $\underline{a}$ as follows.

$$\begin{aligned} x^2 - a_1 x^1 - a_2 &= x^2 + x + 3 \\ &= x^2 - (-1)x - (-3) \\ &= x^2 - 6x - 4 \quad mod\ 7 \end{aligned} \tag{16}$$

Thus, $a_1 = 6, a_2 = 4$. The pseudo-random number sequence generated by this MRG is

$$\mathbf{5},\ \mathbf{3}, 3, 2, 3, 5, 0, 6, 1, 2, 2, 6, 2, 1, 0, 4, 3, 6, 6, 4, 6, 3, 0, 5, 2, 4, 4, 5, 4, 2, 0, 1, 6, 5, 5, 1, 5, 6, 0, 3, 4, 1, 1, 3, 1, 4, 0, 2, \mathbf{5},\ \mathbf{3}, 3, 2 \tag{17}$$

This MRG has period $P = 48 = 7^2 - 1$, which is the maximum possible period. Thus, in this way, for a large prime m and any integer k, given the required primitive polynomial, we can construct an MRG of maximum possible period, $m^k - 1$, which will be very large since m is large. Now, on dividing every number in the sequence by m, we get a uniform pseudo-random number sequence.

Lagged Fibonacci Generators which take the form

$$X_i = X_{i-r} + X_{i-s} \quad mod \quad m, \qquad r, s \geq 1, \quad r < s \tag{18}$$

for carefully chosen r, s and m are an important special case, because they are fast. A particular case of a Lagged Fibonacci Generator is $r = 37, s = 100, m = 2^{30}$. The period of this generator is around 2^{129}. This generator has been invented by Knuth (2002) and is generally called "**Knuth-TAOCP-2002**". TAOCP stands for *The Art of Computer Programming*, Knuth's famous book.

10.4 Inversive Congruential Generators (ICGs)

A relatively new and quite different generator type is the **Inversive Congruential Generator** (*ICG*). Let $m \geq 3$ be a prime number. When $x \neq 0$, define x^{-1} as the unique number in $\{0, 1, ..., m-1\}$ with $xx^{-1} = 1$ *mod m.* By convention, $0^{-1} = 0$.

There are several ways to compute the modular multiplicative inverse x^{-1}. Since m is prime, we obtain x^{-1} by the following simple formula.

$$x^{-1} = x^{m-2} \quad mod \quad m \tag{19}$$

The ICG update is

$$X_i = \begin{cases} a_0 + a_1 X_{i-1}^{-1} \quad mod \quad m & \text{if } X_{i-1} \neq 0 \\ a_0 & \text{if } X_{i-1} = 0 \end{cases} \tag{20}$$

The sequence must have $X_i = X_j$ after finitely many steps and since the next element depends only on its predecessor, $X_{i+1} = X_{j+1}$ etc. Thus, the period $P \leq m$. If the polynomial $f(x) = x^2 + a_0 x + a_1$ is primitive over $GF(m)$, then the sequence will have the maximum possible period. Such polynomials are also called *inversive maximal period polynomials*. For eg., the minimal polynomial over $GF(5)$ is $x^2 + x + 2$. So, with $a_0 = 2, a_1 = 1, m = 5$ and seed $X_0 = 1$, the pseudo-random number sequence is

$$\mathbf{1}, 3, 0, 2, 4, \mathbf{1}, 3, 0, 2, 4, \mathbf{1}, ... \tag{21}$$

This sequence has maximum possible period, 5. Again, the minimal polynomial over $GF(7)$ is $x^2 + x + 3$. So with $a_0 = 3, a_1 = 1, m = 7$ and seed $X_0 = 1$, the pseudo-random number sequence is

$$\mathbf{1}, 4, 0, 3, 6, 2, 5, \mathbf{1}, 4, 0, 3, 6, 2, 5, \mathbf{1}, ... \tag{22}$$

This sequence has period 7, again the maximum possible. Dividing the sequence generated by an ICG by m, we get a uniform pseudo-random number sequence. Thus, we can extend the above idea to large prime numbers to get better sequences of longer periods.

There are three main ways to choose m. Sometimes it is advantageous to choose m to be a large prime number. The value $2^{31} - 1$ is popular because it can be exploited in conjunction with 32 bit integers to get great speed. A prime number of the form $2^k - 1$ is called a ***Mersenne prime***. Another choice is to take $m = 2^r$ for some integer $r > 1$. For the MRG, it is reasonable to combine a small m with a large k to get a large value of $m^k - 1$. The third choice, with $m = 2$, is especially convenient because it allows fast bitwise operations to be used.

10.5 Linear Feedback Shift Register (LFSR)

MRGs with $m = 2$ are called **Linear Feedback Shift Register** *LFSR* generators, i.e., they are defined by the recurrence relation

$$X_i = a_1 X_{i-1} + a_2 X_{i-2} + ... + a_k X_{i-k} \quad mod \quad m(= 2), \qquad k \geq 1 \tag{23}$$

where $\{X_i\}$ is the sequence of pseudo-random values,
$m = 2$: *modulus*,
a_i, $0 < a_i < m$: *sequence of multipliers*, and
$X_0, X_1, ..., X_{k-1}$, $0 \leq X_0 < m$, $i = 0, 1, ..., k-1$: *seed values (initial states)*
are integer constants that specify the generator.

They are also called ***Tausworthe generators***.

Since each output is a bit (0 or 1), we could also write the a_i's and initial x_i's as bits as follows.

$$X_i = a_1 X_{i-1} + a_2 X_{i-2} + ... + a_k X_{i-k} \quad mod \quad m(= 2), \qquad k \geq 1 \tag{24}$$

where $\{X_i\}$ is the sequence of pseudo-random values,
$m = 2$: *modulus*,

a_i, $a_i \in \{0,1\}$: *sequence of multipliers*, and
$X_0, X_1, ..., X_{k-1}$, $X_i \in \{0,1\}$, $i = 0, 1, ..., k-1$: *seed values (initial states)*
are integer constants that specify the generator.

A k-bit word is defined as a sequence of k 1's and 0's. For eg., 1011 is a 4-bit word, 01010110 is an 8-bit word, etc. The output of the LFSR is generated as random bits. We take k of these at a time to form a k-bit word, then shift by 1 position, form another k bit word, and so on. For eg., for $k = 4$, we form the first 4-bit word as $X_0X_1X_2X_3$, then shift by 1 position and form the second word as $X_1X_2X_3X_4$, and so on. We could convert these binary numbers into decimal form because we are used to dealing with decimal numbers and then divide by 2^k to get a uniform pseudo-random number sequence. We could also directly convert them into a uniform sequence by using the formula $\sum_{i=1}^{k} 2^{-i}X_i$, where $X_1X_2...X_k$ is the k-bit word which is treated as $0.X_1X_2...X_k$. Let us take the following example: $k = 4, \underline{a} = (a_1, a_2, a_3, a_4) = (1, 1, 0, 1)$ and seed $X_0 = 0, X_1 = 1, X_2 = 0, X_3 = 0$. We get the following pseudo-random sequence.

$$\mathbf{0,\ 1,\ 0,\ 0}, 0, 1, 1, \mathbf{0,\ 1,\ 0,\ 0}, 0, 1, 1, ... \tag{25}$$

The 4-bit words formed are given in the table below.

Time	4-bit word	Fractional form
1	0100	0.2500
2	1000	0.5000
3	0001	0.0625
4	0011	0.1875
5	0110	0.3750
6	1101	0.8125
7	1010	0.6250
8	0100	0.2500

This LFSR has period 7. For any LFSR, there are 2^k possible k-bit words that can be formed but the all zero word cannot be achieved unless one starts with it and if one does actually start with the all zero word, an entire sequence of zeroes will be obtained. So there are $2^k - 1$ possible words, which is the maximum possible period. A sequence produced by a length k LFSR which has period $2^k - 1$ is called a **PN-sequence** (pseudo-noise sequence). It is possible to characterise LFSRs that produce PN-sequences, again using the concept of primitive polynomials. We define the *characteristic polynomial* of an LFSR as the polynomial $f(x) = x^k + a_1x^{k-1} + ... + a_{k-1}x^1 + a_k$. We shall use the following facts about polynomials $f(x)$ with coefficients in $GF(2)$:

- Every polynomial $f(x)$ with coefficients in $GF(2)$ having $f(0) = 1$ divides $x^p + 1$ for some p. The smallest p for which this is true is the period of $f(x)$.
- An irreducible polynomial of degree k has a period which divides $2^k - 1$.
- An irreducible polynomial of degree k whose period is $2^k - 1$ is called a primitive polynomial.

Observe that $f(x) = x^4 + x^3 + 1$ is an irreducible polynomial over $GF(2)$. To find its period, we have to find the smallest p such that $f(x)$ divides $x^p + 1$. Clearly, $p > 4$. Also, the period divides $2^4 - 1 = 15$, thus, it must be either 5 or 15. Now,

$$x^5 + 1 = (x+1)(x^4 + x^3 + 1) + (x^3 + x) \tag{26}$$

so $f(x)$ does not divide $x^5 + 1$. But,

$$x^{15} + 1 = (x^{11} + x^{10} + x^9 + x^8 + x^6 + x^4 + x^3 + 1)(x^4 + x^3 + 1) \tag{27}$$

so $f(x)$ divides $x^{15} + 1$. Thus, $f(x) = x^4 + x^3 + 1$ has period 15 and is a primitive polynomial. As a result, let us look at the LFSR with $k = 4, \underline{a} = (1, 0, 0, 1)$ and seed $X_0 = 0, X_1 = 0, X_2 = 0, X_3 = 1$.

Time	4-bit word	Fractional form
1	0001	0.0625
2	0011	0.1875
3	0111	0.4375
4	1111	0.9375
5	1110	0.8750
6	1101	0.8125
7	1010	0.6250
8	0101	0.3125
9	1011	0.6875
10	0110	0.3750
11	1100	0.7500
12	1001	0.5625
13	0010	0.1250
14	0100	0.2500
15	1000	0.5000
16	0001	0.0625

So, this sequence has period $P = 15 = 2^4 - 1$, which is the maximum possible period. In this manner, for higher powers of 2, we require primitive polynomials of degree k to construct LSFRs of maximum period, $2^k - 1$. For eg., we have listed all the 6^{th} degree primitive polynomials in the following table.

No.	Primitive Polynomial	$\underline{a}$
1	$x^6 + x + 1$	(0, 0, 0, 0, 1, 1)
2	$x^6 + x^5 + 1$	(1, 0, 0, 0, 0, 1)
3	$x^6 + x^5 + x^2 + x + 1$	(1, 0, 0, 1, 1, 1)
4	$x^6 + x^5 + x^4 + x + 1$	(1, 1, 0, 0, 1, 1)
5	$x^6 + x^5 + x^3 + x^2 + 1$	(1, 0, 1, 1, 0, 1)
6	$x^6 + x^4 + x^3 + x + 1$	(0, 1, 1, 0, 1, 1)

Thus, if we choose $\underline{a}$ as any of the 6 given in the table, we will get an LFSR with period $P = 63 = 2^6 - 1$. For every degree k, the following table lists the period, factors of the period and the number of primitive polynomials of that degree.

Degree	Period	Factors of period	No. of primitive polynomials of this degree
3	7	7	2
4	15	3, 5	2
5	31	31	6
6	63	3, 3, 7	6
7	127	127	18
8	255	3, 5, 17	16
9	511	7, 73	48
10	1,023	3, 11, 31	60
11	2,047	23, 89	176
12	4,095	3, 3, 5, 7, 13	144
13	8,191	8191	630
14	16,383	3, 43, 127	756
15	32,767	7, 31, 151	1,800
16	65,535	3, 5, 17, 257	2,048
17	131,071	131071	7,710
18	262,143	3, 3, 3, 7, 19, 73	7,776
19	524,287	524287	27,594
20	1,048,575	3, 5, 5, 11, 31, 41	24,000
21	2,097,151	7, 7, 127, 337	84,672
22	4,194,303	3, 23, 89, 683	120,032
23	8,388,607	47, 178481	356,960
24	16,777,215	3, 3, 5, 7, 13, 17, 241	276,480
25	33,554,431	31, 601, 1801	1,296,000
26	67,108,863	3, 2731, 8191	1,719,900
27	134,217,727	7, 73, 262657	4,202,496
28	268,435,455	3, 5 29, 43, 113, 127	4,741,632
29	536,870,911	233, 1103, 2089	18,407,808
30	1,073,741,823	3, 3, 7, 11, 31, 151, 331	17,820,000
31	2,147,483,647	2147483647	69,273,666
32	4,294,967,295	3, 5, 17, 257, 65537	Not Available

10.6 Generalised Feedback Shift Register (GFSR)

The **Generalised Feedback Shift Register** (*GFSR*) makes use of the ***XOR*** operator in Boolean algebra. We denote the XOR operator by $\oplus$. Let $X_1, X_2, ..., X_k \in \{0, 1\}$. The output of an XOR operator is determined as follows.

$$X_1 \oplus X_2 \oplus ... \oplus X_k = \begin{cases} 1 & \text{if } there\,are\,odd\,number\,of\;1's\,among\,X_i's \\ 0 & \text{if } there\,are\,even\,number\,of\;1's\,among\,X_i's \end{cases} \tag{28}$$

The GFSR generator is given by

$$X_i = a_1X_{i-1} \oplus a_2X_{i-2} \oplus ... \oplus a_kX_{i-k}, \qquad k > 1 \tag{29}$$

where $\{X_i\}$ is the sequence of pseudo-random values, and
a_i, $a_i \in \{0, 1\}$: *sequence of multipliers.*

Again, we are left to wonder for what choice of $\underline{a} = (a_1, a_2, ..., a_k)$ we will get the maximum possible period, which is $2^k - 1$, because k consecutive zeroes cannot be obtained unless one starts with it, and if one starts with k consecutive zeroes, a whole sequence of zeroes will be obtained. Once again, the answer is: if the polynomial $f(x) = x^k + a_1 x^{k-1} + ... + a_{k-1} x^1 + a_k$ is a primitive polynomial over $GF(2)$, then the sequence $\{X_i\}$ will have maximal period $2^k - 1$. For example, the 5th degree primitive polynomial over $GF(2)$ is $x^5 + x^2 + 1$ so let us choose $\underline{a} = (0, 0, 1, 0, 1)$. The GFSR then boils down to $X_k = X_{k-3} \oplus X_{k-5}$. Starting with the seed $X_0 = X_1 = X_2 = X_3 = X_4 = 1$, we get the following pseudo random sequence:

$$\mathbf{1, 1, 1, 1, 1}, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, \mathbf{1, 1, 1, 1, 1}, 0, 0, ... \tag{30}$$

Observe that the sequence has period $P = 31 = 2^5 - 1$, which is the maximal period. In order to produce a better random sequence, let us take the first 31 bits of this sequence and apply *Kendall's algorithm.* Although there are several versions of Kendall's algorithm, what each version essentially does is shift the original sequence

$$1111100011011101010000100|101100 \tag{31}$$

forward by 6 bits, i.e.,

$$1011001111100011011101010|000100 \tag{32}$$

Repeat this process 3 more times to get the following table:

No.	Sequence
0	1 1 1 1 1 0 0 0 1 1 0 1 1 1 0 1 0 1 0 0 0 0 1 0 0 1 0 1 1 0 0
1	1 0 1 1 0 0 1 1 1 1 1 0 0 0 1 1 0 1 1 1 0 1 0 1 0 0 0 0 1 0 0
2	0 0 0 1 0 0 1 0 1 1 0 0 1 1 1 1 1 0 0 0 1 1 0 1 1 1 0 1 0 1 0
3	1 0 1 0 1 0 0 0 0 1 0 0 1 0 1 1 0 0 1 1 1 1 1 0 0 0 1 1 0 1 1
4	0 1 1 0 1 1 1 0 1 0 1 0 0 0 0 1 0 0 1 0 1 1 0 0 1 1 1 1 1 0 0

Finally, we take the columns to form 5-bit words. Thus, we get 31 pseudo random 5-bit words as follows.

No.	Word	No.	Word	No.	Word	No.	Word
W_1	11010	W_2	10001	W_3	11011	W_4	11100
W_5	10011	W_6	00001	W_7	01101	W_8	01000
W_9	11101	W_{10}	11110	W_{11}	01001	W_{12}	10000
W_{13}	10110	W_{14}	10100	W_{15}	01110	W_{16}	11111
W_{17}	00100	W_{18}	11000	W_{19}	01011	W_{20}	01010
W_{21}	00111	W_{22}	01111	W_{23}	10010	W_{24}	01100
W_{25}	00101	W_{26}	10101	W_{27}	00011	W_{28}	10111
W_{29}	11001	W_{30}	00110	W_{31}	00010		

Each W_i is a 5-bit word, which can be converted to a fraction between 0 and 1 so as to generate uniform pseudo-random numbers.

10.7 Twisted GFSR (TGFSR)

Let $X_i = X_{i1}X_{i2}...X_{ik}$ and $X_j = X_{j1}X_{j2}...X_{jk}$ be two k-bit words. $X_i \oplus X_j = (X_{i1} \oplus X_{j1})(X_{i2} \oplus X_{j2})...(X_{ik} \oplus X_{jk})$. For eg., $1001 \oplus 0101 = (1 \oplus 0)(0 \oplus 1)(0 \oplus 0)(1 \oplus 1) = 1100$.

A modification of the GFSR was proposed by "twisting" the bit pattern in X_{i-p-q}. This is done by viewing the X_i's as w-vectors and premultiplying X_{i-p-q} by a $w \times w$ matrix, A. The recurrence then becomes

$$X_i = X_{i-p} \oplus AX_{i-p-q} \tag{33}$$

where $\{X_i\}$ is the sequence of w-bit pseudo-random values. Also, let $k = p + q$.

This is called a **Twisted GFSR** (*TGFSR*). A typical example of the matrix A is

$$A = \begin{bmatrix} 0 & 1 & 0 & ... & 0 \\ 0 & 0 & 1 & ... & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & ... & 1 \\ a_0 & a_1 & a_2 & ... & a_{w-1} \end{bmatrix}$$

We wish to see whether there exists conditions under which maximal period sequences can be generated. Let $\phi_A(t)$ be the characteristic polynomial of the matrix A. The maximum period possible is $2^{kw} - 1$ if and only if $\phi_A(t^p + t^k)$ is a primitive polynomial of degree kw over $GF(2)$, where $k = p + q$. For matrices defined as above, we can use the fact that $\phi_A(t) = t^w + \sum_{i=0}^{w-1} a_i t^i$. Let us look at the simplest case possible, where $w = 2$. Take $a_0 = 1, a_1 = 1$. Thus,

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

The characteristic polynomial of A is given by $\phi_A(t) = t^2 - t - 1 \equiv t^2 + t + 1$. Now, we take $p = 1, q = 1$, so $k = 2$ and $\phi_A(t^p + t^k) = (t^2 + t)^2 + (t^2 + t) + 1 = t^4 + 2t^3 + 2t^2 + t + 1 \equiv t^4 + t + 1$, which is a primitive polynomial of degree 4 over $GF(2)$. Now, take seed $X_0 = 00, X_1 = 01$. The TGFSR is given by $X_i = X_{i-1} \oplus AX_{i-2}$. We get the following sequence.

$$\mathbf{00,\ 01}, 01, 10, 01, 00, 11, 11, 01, 11, 00, 10, 10, 11, 10, \mathbf{00,\ 01}, 01, 10, ... \tag{34}$$

It can be observed that this sequence has period $P = 15 = 2^4 - 1 = 2^{2\times 2} - 1$, which is the maximum period possible. Let us also look at the case of $w = 3$. We choose $a_0 = 1, a_1 = 0, a_2 = 1$. Thus,

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix}$$

The characteristic polynomial of A is given by $\phi_A(t) = t^3 + t^2 + 1$. Now, we take $p = 1, q = 1$, so $k = 2$ and $\phi_A(t^p + t^k) = (t^2 + t)^3 + (t^2 + t)^2 + 1 = t^6 + 3t^5 + 3t^4 + t^3 + t^4 + 2t^3 + t^2 + 1 \equiv t^6 + t^5 + t^3 + t^2 + 1$, which is a primitive polynomial of degree 6 over $GF(2)$. Now, take seed $X_0 = 010, X_1 = 000$. The TGFSR is given by $X_i = X_{i-1} \oplus AX_{i-2}$. We get the following sequence.

$$\mathbf{010,\ 000}, 100, 100, 101, 100, 110, 111, 010, 100, 000, 001, 001, 010, 001, 101, 110, 100, 001, 000, 011, 011, 100, 011, 010, 101, 001, 011, 000, 111, 111, 001, 111, 100, 010, 011, 111, 000, 110, 110, 011, 110, 001, 100, 111, 110, 000, 101, 101, 111, 101, 011, 001, 110, 101, 000, 010, 010, 110, 010, 111, 011, 101, \mathbf{010,\ 000}, 100, ... \tag{35}$$

This sequence has period $P = 63 = 2^6 - 1 = 2^{2\times3} - 1$, which is the maximum period possible.

A slight variation of the TGFSR is the ***Mersenne Twister***. The name comes from the fact that $2^p - 1$ are called Mersenne primes. Mersenne Twister generates a sequence of word vectors considered as uniform pseudo-random integers between 0 and $2^w - 1$, where w is the number of bits in the word. It is based on the following recurrence relation:

$$X_{i+n} = X_{i+m} \oplus (X_i^u | X_{i+l}^l)A, \quad i = 0, 1, 2, ... \tag{36}$$

where, $\{X_i\}$ is the sequence of w-bit pseudo-random values,
n: *degree of recurrence*,
$m, 1 \leq m \leq n$: *integer*,
A: *$w \times w$ matrix of the form given above*,
X_i^u: *upper or leftmost $w - r$ bits of X_i*,
X_{i+l}^l: *lower or rightmost r bits of X_{i+l}*,
$r, 0 \leq r \leq w - 1$: *separation point of one word*,
$(X_i^u | X_{i+l}^l)$: *concatenation vector obtained by concatenating (X_i^u and X_{i+l}^l) in that order*, and
$X_0, X_1, ..., X_n - 1$: *seed values (initial states)*
are constants which specify the Mersenne twister.

The most commonly used version of the Mersenne Twister algorithm is based on the Mersenne prime, $2^{19937} - 1$. The standard implementation of the Mersenne twister, ***MT19937***, uses a 32-bit word length. The coefficients for MT19937 are:
$(w, n, m, r) = (32, 624, 397, 31),\ a = 9908B0DF_{16}$
It has been proved that MT19937 attains its maximum period, which is $2^{nw-r} - 1 = 2^{19937} - 1$.

11 Some Cryptographically Secure PRNGs (CSPRNGs)

11.1 Indirection Shift Accumulation Add Count (ISAAC)

Indirection Shift Accumulation Add Count (*ISAAC*) is a cryptographically secure pseudorandom number generator and a stream cipher designed by Robert J. Jenkins Jr. in 1993. The ISAAC algorithm uses an array of 256 four-octet integers as the internal state, writing the results to another 256 four-octet integer array, from which they are read one at a time until empty, at which point they are recomputed. The computation consists of altering i-element with (i ⊕ 128)-element, two elements of the state array found by indirection, an accumulator, and a counter, for all values of i from 0 to 255. Since it only takes about 19 32-bit operations for each 32-bit output word, it is very fast on 32-bit computers.

Many implementations of ISAAC are so fast that they can compete with other high speed PRNGs, even with those designed primarily for speed not for security. Only a few other generators of such high quality and speed exist in usage. The best attack till date needs $4.67 \times 10^{1240} s$ to recover the inital state. This result has had no practical impact on the security of ISAAC.

11.2 BLUMBLUMSHUB

BLUMBLUMSHUB (*BBS*) is a pseudorandom number generator proposed in 1986 by Lenore Blum, Manuel Blum and Michael Shub. Let p and q be two large primes and $m = pq$. Define $u = (p-1)(q-1)$. Choose an integer $e, 1 < e < u$ such that $gcd(u, e) = 1$. BBS takes the form

$$X_i = X_{i-1}^e \quad mod \quad m \tag{37}$$

At each step of the algorithm, some output is commonly derived from x_i which is most commonly the *parity bit* of x_i. A parity bit is a bit added to a string of binary code to ensure that the total number of 1-bits in the string is even or odd. We shall take even parity bits as output.

The seed x_0 should be an integer that is co-prime to m (i.e. p and q are not factors of x_0) and not 1 or 0. The two primes, p and q should be congruent to 3 (mod 4) and $gcd(\phi(p), \phi(q))$ should be small (this makes the cycle length large), where ϕ is ***Euler's totient function***. In number theory, Euler's totient function counts the positive integers up to a given integer n that are relatively prime to n. It can be calculated using the following formula:

$$\phi(n) = n \prod_{p|n} (1 - \frac{1}{p}), \tag{38}$$

where the product is over the distinct prime numbers dividing n. The first 99 values of ϕ are given below.

$\phi(n)$ for $1 \leq n \leq 99$

+	0	1	2	3	4	5	6	7	8	9
0	N/A	1	1	2	2	4	2	6	4	6
10	4	10	4	12	6	8	8	16	6	18
20	8	12	10	22	8	20	12	18	12	28
30	8	30	16	20	16	24	12	36	18	24
40	16	40	12	42	20	24	22	46	16	42
50	20	32	24	52	18	40	24	36	28	58
60	16	60	30	36	32	48	20	66	32	44
70	24	70	24	72	36	40	36	60	24	78
80	32	54	40	82	24	64	42	56	40	88
90	24	72	44	60	46	72	32	96	42	60

Observe from the table that $\phi(47) = 46$ and $\phi(83) = 82$, which means that $gcd(\phi(47), \phi(83)) = 2$. Let us take $p = 47$ and $q = 83$. Thus, $u = (47-1) \times (83-1)$. Let us choose $e = 3$ since $gcd(u, e) = 1$. This gives us a sequence of period $P = 88$. The corresponding output (based on even parity bits) that we get from the sequence with seed $X_0 = 51$ is:

$$\begin{gathered}\mathbf{0}, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, \\ 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, \\ \mathbf{0}, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, ...\end{gathered} \tag{39}$$

It is to be observed that the period of BLUMBLUMSHUB is significantly less when compared to the previous PRNGs that we have seen so far. However, it is much more secure than the previous ones since the main difficulty of predicting BBS's output lies in the intractability of the "quadratic residuosity" problem, which is:

Given a composite number n, find whether x is a perfect square modulo n.

It has been proven that this is as hard as cracking the RSA public-key cryptosystem which involves the factoring of a large composite.

11.3 devUrandom

In Unix-like operating systems, `/dev/random`, `/dev/urandom` and `/dev/arandom` are special files that serve as pseudo-random number generators. They allow access to environmental noise collected from device drivers and other sources.

12 A proposed PRNG

I have thought of an idea to increase the period of the BLUMBLUMSHUB generator in Section 11.2 in the following way. It will involve two stage randomisation. Take the first 101 bits from the sequence generated by choosing $p = 47, q = 83, e = 3$ with seed $X_0 = 51$:

$$\begin{gathered}\mathbf{0}, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, \\ 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, \\ \mathbf{0}, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, ...\end{gathered} \tag{40}$$

We will form k-bit words from this sequence. Take $k = 6$. Form the first 6-bit word by taking the first 6 bits of the sequence. To get the next 6-bit words, follow the algorithm given below.

- **Step 1:** Set $pos = 1$.

- **Step 2:** Let $r = LCG(a_0 = 17, a_1 = 29, m = 2^6, seed = get.seed(2)\, mod\, 2^6, n = 1)$, where $get.seed(2)$ generates a random integer which is the last two digits of the system time given by R. Thus, r is a random integer between 0 and 63 which has been generated by the LCG algorithm.

- **Step 3:** $pos = pos + r$.

- **Step 4:** If $pos \leq 101 - 6 + 1 = 96$, go to step 5. Else, go to step 6.

- **Step 5:** Form a new 6-bit word by taking 6 consecutive bits starting from X_{pos}. Go to step 7.

- **Step 6:** Form a new 6-bit word by taking $101 - pos + 1$ consecutive bits starting from X_{pos} and $6 - (101 - pos + 1)$ consecutive bits starting from X_1, i.e., the 6-bit word will look like $X_{pos}X_{pos+1}..X_{101}X_1X_2..X_{6-(101-p+1)}$. Go to step 7.

- **Step 7:** To form a new word, go to step 2. Otherwise, exit.

Convert each of the 6-bit words to fractions between 0 and 1 to get a uniform pseudo-random sequence. For eg., let us generate 100 6-bit words in this manner and perform Runs test on them. We get a $p - value$ of 0.4214, which implies we can conclude that the sequence comes from a random process.

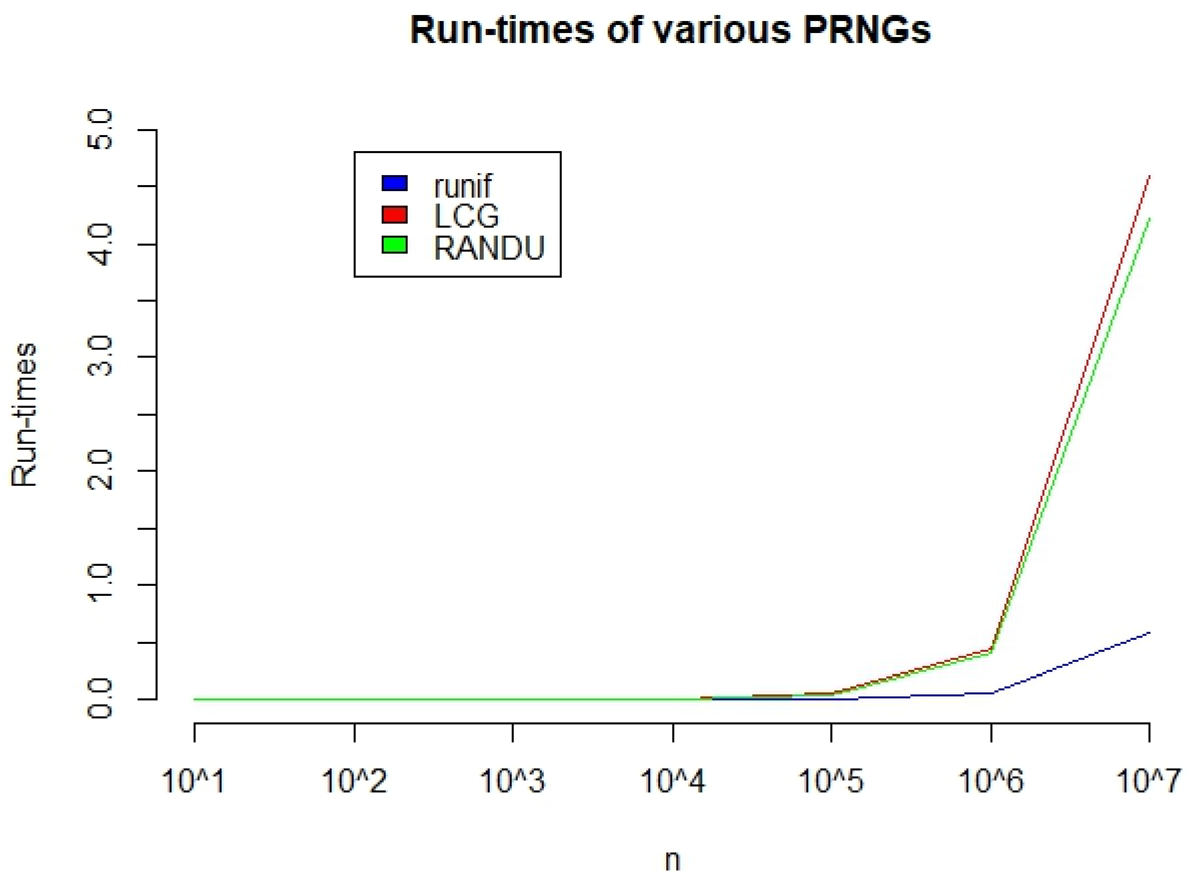
Because of the randomness created in the shift in position while forming words from consecutive bits by the introduction of an LCG, the period of the sequence of k-bit words will be very large. This shows that we need not remain confined to one specific algorithm only while generating pseudo-random numbers. We could think of ways to combine two or more algorithms to obtain pseudo-random sequences having longer periods. Also, they should be harder to crack by a hacker who wants to intercept the information. This combining of various algorithms is an area which requires more research.

13 Run-time Comparison

First, let us compare the run-times of an LCG, RANDU and R's own uniform random number generator, `runif`. We observe the run-times for values of n (sample size) ranging from 10^1 to 10^7. The parameters that we choose for LCG are: $a_0 = 1013904223, a_1 = 1664525, m = 2^{31}$. For RANDU, the parameters are: $a_1 = 65539, m = 2^{31}$. The seed is chosen entirely randomly from the last 7 digits of the system time given by R. The following table contains the run-times.

n	`runif`	LCG	RANDU
10^1	0.00	0.00	0.00
10^2	0.00	0.00	0.00
10^3	0.00	0.00	0.00
10^4	0.00	0.01	0.00
10^5	0.00	0.05	0.03
10^6	0.06	0.44	0.41
10^7	0.58	4.59	4.22

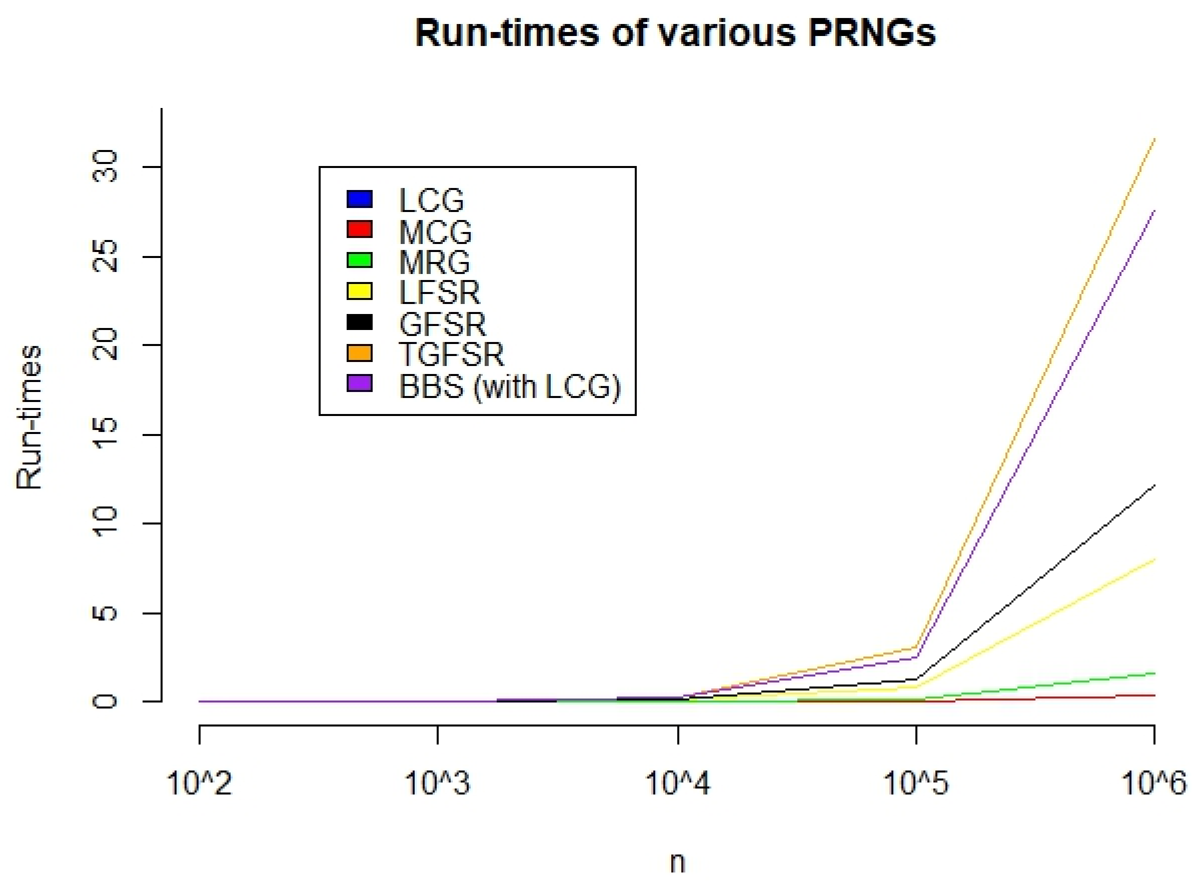
The graph below plots how the run-time increases. It can be observed that runif has a significantly lesser run-time than both RANDU and LCG. Also, for $n \leq 10^4$, LCG and RANDU are almost equally fast. But for $n \geq 10^4$, RANDU is slightly better than LCG in terms of the run-time.



Next, let us compare the run-times of the algorithms covered in this paper. For the comparison to make sense, the parameters of each of the algorithms must be of the same order. So, we consider $m = 2^6$, wherever $m \neq 2$ and consider $k = 6$, wherever $m = 2$. Parameters of the algorithms are chosen randomly (from system time) from 1 to 64 but satisfying the necessary requirements (if any). Seeds are also chosen randomly (from system time) but appropriately. The following table gives us the run-times of the various algorithms to obtain n pseudo-random numbers.

n	LCG	MCG	MRG	LFSR	GFSR	TGFSR	BBS (with LCG)
10^2	0.00	0.00	0.00	0.00	0.02	0.00	0.00
10^3	0.00	0.00	0.00	0.02	0.02	0.02	0.04
10^4	0.00	0.00	0.01	0.08	0.12	0.30	0.26
10^5	0.04	0.05	0.14	0.79	1.22	3.01	2.49
10^6	0.41	0.39	1.55	7.96	12.15	31.57	27.53

The following graph plots the run-times of the various algorithms.



It can be observed that as far as run-time is concerned, LCG and MCG are the best and TGFSR is the worst among all the algorithms. MRG has a slightly higher runtime than that of LCG. The run-times of the others are quite high but the rate of increase of run-time for LFSR and GFSR is not as high as the rate of increase of run-time for TGFSR and our own designed BBS (with LCG). So, the graph makes it clear that a price to pay for higher periods is higher run-times, i.e., slower algorithms.

14 References

- Sheldon Ross. *Simulation.*
- J.D. Alanen, Donald E. Knuth. *Tables of Finite Fields.* Sankhya: The Indian Journal of Statistics, Series A, Vol. 26, December, 1994, pp. 305-328.
- Christophe Dutang, Diethelm Wuertz. *A note on random number generation.* September, 2009.
- Lih-Yuan Deng, Jyh-Jen Horng Shiau, Henry Horng-Shing Lu. *Large-Order Multiple Recursive Generators with Modulus* $2^{31} - 1$. INFORMS Journal on Computing, Articles in Advance, 2011, pp. 1-12.
- Makoto Matsumoto, Yoshiharu Kurita. *Twisted GFSR Generators.* Transactions on Modeling and Computer Simulation (TOMACS), April, 1992.
- *en.wikipedia.org*
- *http://www-math.ucdenver.edu/ wcherowi/courses/m5410/m5410fsr.html*
- *www.maximintegrated.com/en/app-notes/index.mvp/id/4400*
- *math.stackexchange.com*