PSEUDORANDOM NUMBER GENERATION MODULO m

ANAY AGGARWAL

1. Introduction

Pseudorandom number generation involves the creation of sequences of numbers that exhibit properties similar to those of truly random numbers. However, these sequences are generated deterministically by algorithms. These algorithms use a starting point called a seed and employ mathematical formulas to produce sequences of numbers that, while not truly random, appear random in their distribution and properties. The generated sequences must pass various statistical tests to ensure their randomness qualities.

Pseudorandom number generation holds major practical importance. For example, pseudorandom sequences are instrumental in Monte Carlo simulations, aiding in approximating complex integrals and solving various problems in physics, finance, and engineering. Additionally, in cryptography, pseudorandom number generators are crucial for generating cryptographic keys and in the creation of unpredictable sequences for encryption algorithms, ensuring the security and confidentiality of sensitive data and communication.

In this paper, I will begin in section 2 by introducing preliminary results that we will need for the following sections. In sections 3 and 4 of this paper, I shall present two PRNGs hinging on computation modulo m: The Linear Congruence Generator, and the Blum-Blum-Shub Generator. In section 5, I shall analyze the generators based on statistical tests from the German Federal Office for Information Security and the National Institute of Standards and Technology, and compare the results.

2. Preliminaries

First, we need to mathematically define a Pseudorandom Number Generator (PRNG). We will use the definition from [1].

Definition 2.1. (Pseudorandom Number Generator) Let S be a finite set, called the *state space*. Let $f : S \to S$ be a function. Let $g : S \to [0, 1]$ be a function, called the *output function*. With an initial value $S_0 \in S$, called the *seed*, we will generate a sequence of random numbers $U_0, U_1, \cdots$ by defining

$$S_n = f(S_{n-1})$$
$$U_n = g(S_n)$$

We call $\langle f, g, S\rangle$ a *Pseudorandom Number Generator*.

Essentially, a Pseudorandom Number Sequence is a sequence generated by iteratively applying some function to an initial value, and then mapping each element to some real number in the interval $[0, 1]$.

Date: December 8, 2023.

To analyze the LCG and BBS generators, we will need some number-theoretic definitions. For the BBS generator, we will need to pick two primes that are roughly the same size (this will be used as a modulus). To do this, we need to strictly define what "roughly the same size" means. We can do this with the following:

Definition 2.2. Define the *length* of a natural number N to be $\lfloor 1 + \log_2 N \rfloor$, or the number of digits in the binary representation of N.

It turns out that in order to calculate the period of these generators, we will need to calculate the period of sequences of the form $a, a^2, a^3, \cdots$ modulo some number m. This is because, by construction, the LCG and BBS generators produce terms that can be explicitly calculated in terms of powers of a modulo m. The Carmichael function is a useful tool for this.

Definition 2.3. Let the Carmichael function $\lambda : \mathbb{N} \to \mathbb{N}$ be defined by setting $\lambda(n)$ as the smallest positive integer m such that $a^m \equiv 1 \pmod{n}$ holds for every integer a coprime to n.

It turns out that we may explicitly calculate this function! By [2], it is true that

$$\lambda(n) = \begin{cases} \varphi(n) & \text{if } n \text{ is 1, 2, 4, or an odd prime power,} \\ \frac{1}{2}\varphi(n) & \text{if } n = 2^r,\ r \geq 3 \end{cases},$$

and that $\lambda(n) = \operatorname{lcm}(\lambda(n_1), \lambda(n_2), \cdots, \lambda(n_k))$ if $n = n_1 n_2 \cdots n_k$, where the n_i are powers of distinct primes. A key corollary of this fact is that $\lambda(a) \mid \lambda(b)$ if $a \mid b$. This follows by setting $a_1, a_2, \cdots, a_k$ as the prime powers dividing a, and $a_1, a_2, \cdots, a_k, b_1, b_2, \cdots, b_\ell$ as the prime powers dividing b. Then

$$\lambda(a) = \operatorname{lcm}(\lambda(a_1), \lambda(a_2), \cdots \lambda(a_n)) \mid \operatorname{lcm}(\lambda(a_1), \lambda(a_2), \cdots, \lambda(a_n), \lambda(b_1), \lambda(b_2), \cdots, \lambda(b_\ell)) = \lambda(b),$$

as proposed. This fact will come in handy in our analysis.

Equipped with the Carmichael function, we may prove a central lemma in our upcoming period analysis.

Lemma 2.4. *For any positive integers a and m, the sequence $a, a^2, a^3, \cdots$ is periodic modulo m. Furthermore, this period is a divisor of $\lambda(m)$.*

Proof. Let $d = \gcd(a, m)$, so that $a = da'$ and $m = dm'$ with a', m' relatively prime. We also have that $\gcd(a, m') = 1$. Clearly, all terms in the sequence are zero modulo d, so it suffices to show that the sequence is periodic modulo m'. This follows because $a^{\lambda(m')+1} \equiv a \pmod{m'}$. This period is thus a divisor of $\lambda(m')$. Because $m' \mid m$, $\lambda(m') \mid \lambda(m)$, and our lemma is proven. ∎

3. The LCG Generator

The Linear Congruence Generator (LCG), is a generator that iteratively applies a linear function (taken modulo m). This is one of the first natural generators to be thought of after Definition 2.1. In particular, we would like to take the generator modulo m so that the terms don't become too large to store in memory, and linear functions are good examples of basic functions modulo m. To mathematically define the LCG, we will use the definition from [3].

Definition 3.1. (LCG Generator) Let X_m be the set of residues modulo m. Define our state space S to be

$$S = \bigcup_{m \in \mathbb{N}} X_m.$$

We let $f : S \to S$ be defined by $f(x) = (ax + c) \pmod m$ for some integer constants $0 \leq a, c < m$, when $x \in X_m$. We let $g : S \to [0, 1]$ be defined by $g(x) = \frac{x}{m}$, when $x \in X_m$. Then $\langle f, g, S \rangle$ is our LCG generator.

Essentially, the LCG takes input m, a, c and a seed x_0, and outputs the sequence $u_0, u_1, u_2, \cdots$ obtained by setting $x_{i+1} = (ax_i + c) \pmod m$ and $u_i = \frac{x_i}{m}$.

Example. When $m = 10$ and $x_0 = a = c = 7$, the sequence obtained is

$$u_0, u_1, u_2, \cdots = 0.7, 0.6, 0.9, 0, 0.7, 0.6, 0.9, 0, \cdots$$

As the example shows, the sequence isn't always "random". It turns out the sequence is always periodic.

Theorem 3.2. *No matter the choice of (m, a, c, x_0), the sequence $(u_n)_{n \geq 1}$ is always periodic.*

Proof. We may quickly reject the case $a = 1$. For other a, we claim that

$$x_n = \left(a^n x_0 + \frac{a^n - 1}{a - 1} c \right) \pmod m$$

for all $n > 0$. This claim may be verified by a simple induction. The base case $n = 1$ is trivial. For the inductive step, note that

$$a \left(a^n x_0 + \frac{a^n - 1}{a - 1} c \right) + c = a^{n+1} x_0 + \frac{a^{n+1} - a}{a - 1} c + c = a^{n+1} x_0 + \frac{a^{n+1} - 1}{a - 1} c.$$

Now, it is enough to prove that $(a^n)_{n \geq 1}$ is periodic modulo m, which is true by lemma 2.6. ■

To generate an unpredictable sequence, we would like to choose the four-tuple (m, a, c, x_0) so that this period is as large as possible. The periodic length is at most m, as there are only m possible values that the sequence can cycle through. The following theorem from [3] addresses when this maximal length is possible:

Theorem 3.3. *The linear congruential sequence defined by (m, a, c, x_0) has period length m if and only if:*

(i) $\gcd(c, m) = 1$
(ii) $a \equiv 1 \pmod p$ *for all primes p dividing m*
(iii) If m is a multiple of 4, then $a - 1$ is a multiple of 4.

To run tests, we may have to convert the LCG sequence to a binary sequence. To do this, we may map u_i to a bit by setting $b_i = mu_i \pmod 2$.

4. The BBS Generator

The Blum-Blum-Shub (BBS) PRNG is quite similar to the LCG. It still does all calculations modulo some m, but instead of a linear function, it is a quadratic function. In particular, it is the function $f(x) = x^2$. We will use the mathematical definition from the original paper by Blum, Blum, and Shub: [4].

Definition 4.1. (BBS Generator) Let $\mathcal{N}$ be the set of $N \in \mathbb{N}$ such that there exist two primes $p, q \equiv 3 \pmod 4$ of equal length with $N = pq$. For $N \in \mathcal{N}$, we denote X_N as the set of non-zero quadratic residues modulo N. Define

$$S = \bigcup_{N \in \mathcal{N}} X_N.$$

This will be our state space. Let $f : S \to S$ be defined by $f(x) = x^2 \pmod N$ for $x \in X_N$. Let $g : S \to [0, 1]$ be such that $g(x)$ is the parity of x. The BBS generator is then $\langle f, g, S \rangle$.

In other words, the BBS takes input N and a seed x_0, and outputs the sequence of bits $b_0, b_1, b_2, \cdots$ obtained by setting $x_{i+1} = x_i^2 \pmod N$ and extracting the bit b_i as the parity of x_i.

Example. Take $N = 7 \cdot 19 = 133$ and $x_0 = 4$. Then $(x_n)_{n \geq 0}$ is periodic with period 6, and $x_0, x_1, x_2, x_3, x_4, x_5 = 4, 16, 123, 100, 25, 93$, so that $b_0, b_1, b_2, b_3, b_4, b_5 = 0, 0, 1, 0, 1, 1$.

Analogous to the LCG generator, the BBS also generates a periodic sequence.

Theorem 4.2. *No matter the choice of (N, x_0), the sequence $(x_n)_{n \geq 1}$ is always periodic (and hence $(b_n)_{n \geq 1}$ is also always periodic). Furthermore, this period is a divisor of $\lambda(\lambda(N))$.*

Proof. Notice that $x_n = x_0^{2^n} \pmod N$, for all $n \geq 0$. By Lemma 2.6, $(2^n)_{n \geq 1}$ is periodic modulo $\lambda(N)$, with period $d \mid \lambda(N)$. Because $x_0^a \equiv x_0^b \pmod N$ when $a \equiv b \pmod{\lambda(N)}$, the sequence is periodic. The period is then a divisor of $\lambda(d)$. Since $d \mid \lambda(N)$, $\lambda(d) \mid \lambda(\lambda(N))$, and the result follows. ∎

Like the LCG, it is imperative that we choose the pair (N, x_0) so that the generator has a large period so that the generator is as unpredictable as possible. However, it is significantly more difficult to find such pairs, and there is no (as of yet) nice analog to Theorem 3.3. On the other hand, the BBS happens to be extremely unpredictable cryptographically. In [4], it is shown that predicting the BBS generator is equivalent to factoring N, which is a well-known computationally difficult problem for large N.

5. Comparison and Analysis

The guiding question for this section is as follows.

Question 5.1. *How do we tell if a PRNG is "good"?*

The German Federal Office for Information Security has answered this question in the article [5]. In summary, there are four main criteria:

(1) The probability that generated sequences of random numbers are different from each other should be high.
(2) The randomly generated sequence is indistinguishable from "truly random" numbers. In particular, there are specific statistical tests to tell if a sequence experiences true randomness.
(3) An attacker cannot practically guess the future terms in the sequence given any subsequence.
(4) An attacker cannot practically guess previous terms in the sequence given a particular term.

For all cryptographic applications, only the third and fourth criteria are important.

The first and second criteria are quite similar, and the third and fourth criteria are quite similar. In this paper, we shall put particular emphasis on the second criterion.

The main statistical tests given in [5] are the following:

- The Monobit Test
- The Runs Test
- The Autocorrelation Test

These tests essentially test if a bit sequence has a roughly equal number of zeroes and ones, and if any subsequence can be used to predict the rest of the sequence. Note that these tests only work for binary sequences, so we must take the binary version of the LCG sequence. I will be using the definitions of the tests given by the National Institute of Standards and Technology (NIST) at [6].

The Monobit Test: For a bit string $b_1b_2b_3\cdots b_n$, we let $X_i = 2b_i - 1$. We let

$$s = \frac{|X_1 + X_2 + \cdots + X_n|}{\sqrt{n}}.$$

With the complementary error function

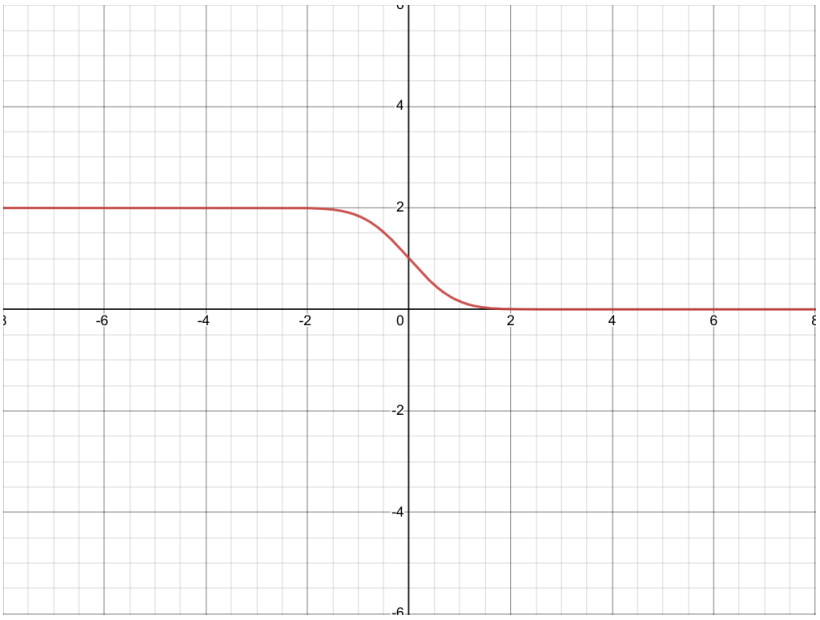
$$\operatorname{erfc}(z) = \frac{2}{\sqrt{\pi}} \int_z^\infty e^{-t^2} \mathrm{d}t,$$

we finally compute the P-value

$$P = \operatorname{erfc}\left(\frac{s}{\sqrt{2}}\right).$$

If $P < 0.01$, the bit sequence is considered non-random. Otherwise, the sequence is acceptable as random. This test is considered valid if $n \geq 100$.

The point is that $X_i = \pm 1$, depending on if $b_i = 0$ or 1. Then $|X_1 + X_2 + \cdots + X_n|$ is really measuring the proportion of the number of zeroes to the number of ones. A large difference between the number of zeroes and ones will produce a large s value, and a small difference will produce a small s value. As for the erfc function, see the below plot of $\operatorname{erfc}(x)$:



Since $s > 0$, this value quickly decreases to 0, so we require s to be small if we want $P < 0.01$.

The Runs Test: For a bit string $b_1 b_2 b_3 \cdots b_n$, we compute

$$\pi = \frac{\sum_i b_i}{n}.$$

If π is reasonable, i.e.

$$|\pi - 1/2| < \frac{2}{\sqrt{n}},$$

we may apply this test. Let r_k be the indicator variable that is 1 if $b_k = b_{k+1}$ and 0 otherwise. We set

$$V = 1 + \sum_{k=1}^{n-1} r_k.$$

Finally, we compute the P-value

$$P = \text{erfc}\left(\frac{|V - 2n\pi(1-\pi)|}{2\pi(1-\pi)\sqrt{2n}}\right).$$

Again, $P < 0.01$ concludes that the bit sequence is non-random, and we run this test if $n \geq 100$. The point of this test is to make sure there are no lengthy runs of zeroes or ones. First, we check if the proportion of zeroes and ones is reasonable with π. Then, V denotes the total length of all the runs. The P-value is a function of V that is low if V is (relatively) high.

The Autocorrelation Test: For a binary sequence $b_1, b_2, \cdots, b_n$, we convert it to a sequence $a_1, a_2, \cdots, a_n$ with $a_i = 2b_i - 1$. The autocorrelations of the sequence are then

$$c_k = \sum_{j=1}^{n-k} a_j a_{j+k}$$

for $0 \leq k \leq n-1$. Essentially, c_k measures how strongly the bit sequence resembles a version of itself that has been acyclically shifted by k positions. We would like for $s = \frac{\sum_{k=0}^{n-1} c_k}{n^3}$ to be

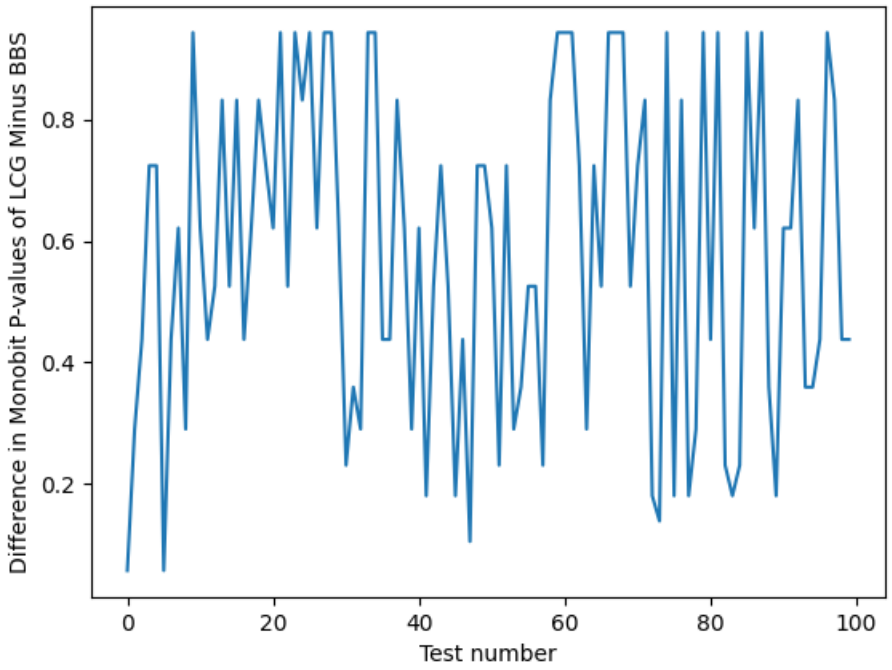
(relatively) small.

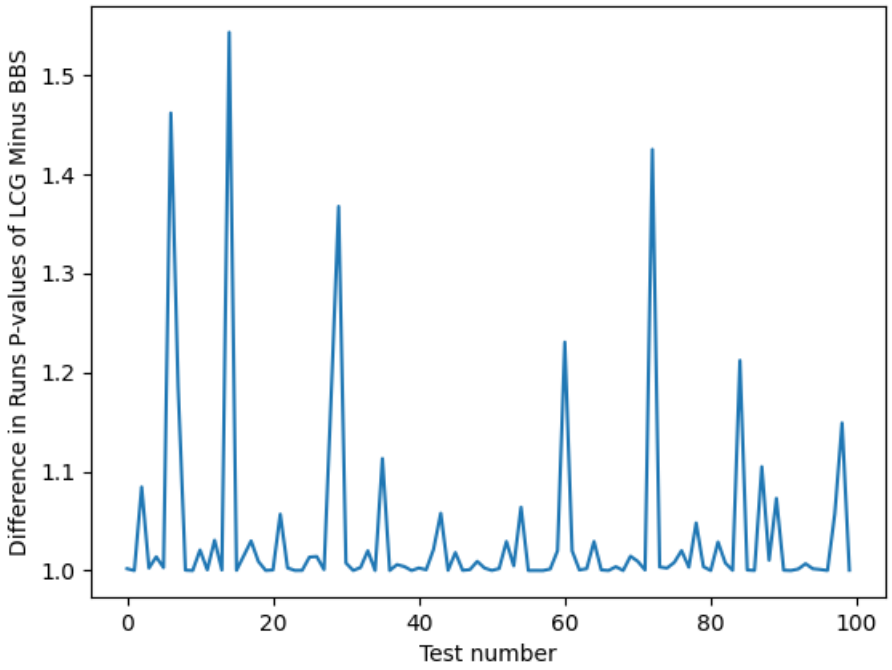
Now, we may test the LCG and BBS programmatically with these statistical tests. First, we shall implement the LCG and BBS generators (with the variable n representing the number of iterations of the generator that we would like to run), as well as our three tests:

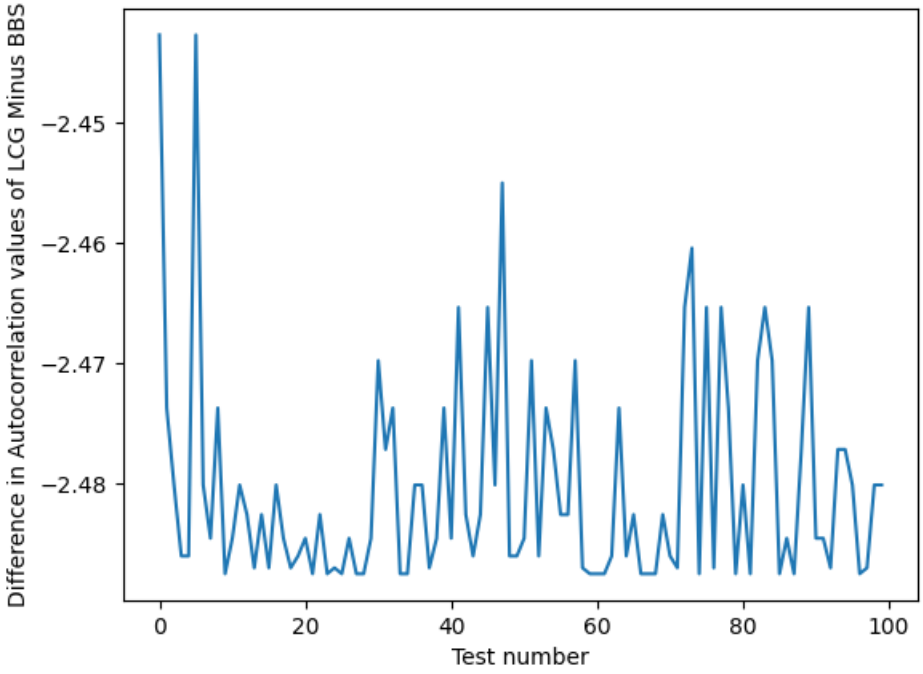
```
import numpy as np
from scipy import special

def LCG(m,a,c,x,n):
  arr = [x]
  for i in range(n):
    arr.append(((a*x+c) % m) % 2)
    x = (a*x+c) % m
  return arr

def BBS(N,x,n):
  arr = [x]
  for i in range(n):
    arr.append(((x*x) % N) % 2)
    x = (x*x) % N
  return arr

def Monobit(arr):
  s = 0
  for i in arr:
    s += 2*i-1
  s = abs(s)/np.sqrt(len(arr))
  return special.erfc(s/np.sqrt(2))

def Runs(arr):
  n = len(arr)
  pi = 0
  for i in arr:
    pi += i
  pi = pi/n
  if(abs(pi-0.5)>=2/np.sqrt(n)):
    return -1
  V = 1
  for k in range(1,n):
    if(arr[k-1] == arr[k]):
      V += 1
  numerator = abs(V-2*n*pi*(1-pi))
  denominator = 2*pi*(1-pi)*np.sqrt(2*n)
  return special.erfc(numerator/denominator)

def Autocorrelation(arr):
  s = 0
```

```
  arr2 = []
  n = len(arr)
  for i in arr:
    arr2.append(2*i-1)
  for k in range(n):
    ck = 0
    for j in range(1, n-k+1):
      ck += arr2[j-1] * arr2[j+k-1]
    s += ck
  return s/(n*n*n)
```

Now, we must figure out how to accurately compare the LCG and the BBS. To do this, we will choose $m = N = pq$ so that the calculations for each algorithm are roughly of the same difficulty. We will choose the original state to be 1 in both cases. For the LCG, we will satisfy theorem 3.3 by taking c as some number such that $\gcd(c, m) = 1$. We cannot quite satisfy theorem 3.3 in terms of a, as m is squarefree so we would have to choose $a = 1$, which is quite predictable. Thus we will choose a arbitrarily. Finally, we will run 200 iterations for each generator. We will generate random $m = N$ by taking two random primes p_k for $10 \leq k \leq 100$ and multiplying them together. For each test, we will create 100 values of N, and plot the difference in the LCG and BBS results. The results are as follows (with the Autocorrelation results scaled appropriately).







As seen by the data, the BBS generally had a better performance on the Monobit and Runs test, whereas the LCG had a better performance on the Autocorrelation test. This was because the LCG's P-value was generally larger than the BBS's P-value for the first two tests, but not the third. The reader may also notice that random spikes seem to appear in the

above graphs. This may be because the tests produce extremely large values for sequences that don't perform well: the functions used to calculate the P-values blow underperforming sequences way out of proportion.

As a final note, the third and fourth criteria have to do with the realm of *predictability*. This is out of the scope of this paper, but the reader should note that the BBS generator is considered *unpredictable* (as mentioned at the end of section 4), and the LCG generator is considered *predictable*, for some definition of predictability. In summary, the BBS generator is considered a "better" PRNG than the LCG generator by most metrics.

References

[1] Art B. Owen. *Monte Carlo theory, methods and examples.* `https://artowen.su.domains/mc/`, 2013.
[2] Robert D. Carmichael. *Theory of numbers.* J.Wiley & Sons, Inc., 1914.
[3] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Addison-Wesley, Boston, third edition, 1997.
[4] L. Blum, M. Blum, and M. Shub. A simple unpredictable pseudo-random number generator. *SIAM Journal on Computing*, 15(2):364–383, 1986.
[5] Werner Schindler. A proposal for: Functionality classes for random number generators - version 2.35 - draft. *Federal Office for Information Security*, Jun 2023.
[6] Andrew Rukhin, Juan Sota, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, and et al. *A statistical test suite for random and pseudorandom number generators for cryptographic applications*, 2000.

Email address: `anay.aggarwal.2007@gmail.com`