A Field Based Pseudorandom Number Generator

Aaryaman Sagar
rmn100@gmail.com

A Field Based Pseudorandom Number Generator

Aaryaman Sagar
rmn100@gmail.com

Abstract. In this paper the concept of prime numbers and their finite fields has been used to present a Pseudorandom Number Generator, which can be used in public-key cryptography.

1 Introduction

Random numbers are critical in every aspect of cryptography and for simulating real world phenomenon such as a variety of simulations. Ideally, we would require a device that generates a completely random string of 0s and 1s that appear in every way as the sequence of heads and tails to have been generated by the successive flips of an unbiased coin. According to Quantum theory such devices do exist. They are based on measuring the radioactive decay of atoms. According to quantum theory, given an atom of some radioactive substance, there is a number T such that the atom has a 50% chance of decaying in the next T seconds but there is no way of predicting of predicting in advance whether the atom will decay. The device can then return a 1 if the atom decays and a 0 if it does not. Therefore, in theory it can be effectively used to produce a long string of random bits by repeating the same process with a number of such atoms.

Unfortunately true randomness is very hard to achieve mathematically as almost all mathematical algorithms designed to imitate randomness proceed to result in a sequence, which is determined by mathematical factors and therefore are predictable.

Many modern algorithmic Random Number Generators use a particular value or 'seed' to produce a bit string which looks like a random sequence to someone who does not know the seed. A function of this sort is called a Pseudorandom Number Generator (PRNG). A Pseudorandom Number Generator is a function so the output it produces is not random at all but is determined by the input.

A simple model of a PRNG can be developed by assuming a binary function $f(k,i)$, which returns the following pseudorandom bits for different values of k, which in this case can be taken to be the seed value 's'

$$b_0 = f(s,0), \qquad b_1 = f(s,1), \qquad b_2 = f(s,2), \quad \ldots\ldots..$$

Such that the general sequence or the pseudorandom bit string becomes $b_0 \| b_1 \| b_2 \| b_{3\ldots\ldots}$

2 Some Number-Theoretic Preliminaries

DEFINITION The ring of a number is defined as the set of all the whole numbers less than it.

DEFINITION The set (Z/mZ) is defined as all the integers in the ring of 'm'

$(Z/mZ)^* = \{ a \in (Z/mZ) : \gcd(a,m) = 1 \}$;

where gcd(a,b) is the greatest common divisor of a and b

The set $(Z/mZ)^*$ is called the group of units modulo m. F_m^* can also be written instead of $(Z/mZ)^*$.

DEFINITION Euler's phi function (also sometimes known as Euler's totient function) is the function

$\Phi(m) = \#(Z/mZ)^* = \#\{ 0 \le a \le m : \gcd(a,m) = 1 \}$

In words it can be defined as the cardinality of the set of whole numbers less than 'm' such that each element is co-prime to 'm'.

Note : The totient function is multiplicative, i.e. $\Phi(mn) = \Phi(m) \cdot \Phi(n)$

DEFINITION The order of an element of (Z/mZ) is defined as the number after which the powers of the element modulo 'm' start repeating their values.

PROPOSITION 1. Let $m \ge 1$ be any integer.

Let 'a' be an integer. Then

$ab \equiv 1 \pmod{m}$ for some integer 'b' if and only if $\gcd(a,m) = 1$.

THEOREM (Fermat's Little Theorem). Let 'p' be any prime number and let a be any integer. Then

$$a^{p-1} \equiv 1 (\mathrm{mod}\, p) \text{ if 'a' is not divisible by 'p'}$$

OR

$$a^{p-1} \equiv 0 (\mathrm{mod}\, p) \text{ if 'a' is divisible by 'p'}$$

DEFINITION (Carmichael function). The Carmichael function of a positive integer 'm' denoted $\lambda(m)$ is defined as the smallest integer n such that

$a^n \equiv 1 (\mathrm{mod}\, p)$

for every 'a' that is co-prime to 'm', or that belongs to (Z/mZ)*

THEOREM (Euler's Formula for pq). Let 'p' and 'q' be distinct primes and let

$$g = \gcd(p-1, q-1)$$

Then

$a^{(p-1)(q-1)/g} \equiv 1 \pmod{pq}$ for all a satisfying $\gcd(a,pq) = 1$

DEFINITION (Safe Primes and Sophie Germain Prime). A safe prime is defined as a prime of the form $2p+1$, where p is also a prime, here p is called a Sophie Germain Prime.

THEOREM (Hardy-Littlewood estimate) The number of Sophie Germain Primes and thus the number of safe primes less than 'n', i.e. N(n) is given by

$$N(n) = \frac{C_2 n}{(\ln n)^2}$$

PROPOSITION 2. Let 'p' and 'q' be distinct primes and let $e \geq 1$ satisfy

$$\gcd\left(e,(p-1)(q-1)\right) = 1$$

According to Proposition 1 'e' has an inverse modulo $(p-1)(q-1)$, say

$$de \equiv 1 \pmod{(p-1)(q-1)}$$

Then the congruence

$$x^e \equiv c \pmod{pq}$$

has the unique solution $x \equiv c^d \pmod{pq}$

This proposition results in an efficient algorithm to solve the congruence

$x^e \equiv c \pmod{pq}$ that involves first solving $de \equiv 1 \pmod{(p-1)(q-1)}$ and then computing $c^d \bmod pq$. We can often make the computation faster by using a smaller value of 'd'. Let $g = \gcd(p-1,q-1)$ and suppose that we solve the following congruence for 'd'.

$$de \equiv 1 \pmod{\frac{(p-1)(q-1)}{g}}$$

Euler's formula says that $a^{(p-1)(q-1)/g} \equiv 1 \pmod{pq}$. Hence we write

$d \equiv 1 + k\left(\frac{(p-1)(q-1)}{g}\right)$ then

$$\left(c^d\right)^e = c^{de} = c^{1+k(p-1)(q-1)/g} = c\left(c^{(p-1)(q-1)/g}\right)^k \equiv c \pmod{pq}$$

Thus using this smaller value of d, we still find that $c^d \bmod pq$ is a solution to $x^e \equiv c \pmod{pq}$

DEFINITION A group consists of a set G and a rule, which can be denoted by $*$ for combining two elements $a,b \in G$ to obtain another element $c \in G$ where $c = a*b$. The composition operation is required to have an identity element such that $a*e = e*a = a$, each element should have an inverse such that $a*a^{-1} = e$ and follow the associative law. If G has finite elements then the order of G is the number of elements in G, which is denoted by $|G|$.

DEFINITION Let G be a group and let $a \in G$, then if $a^d = e$ then the smallest such d is called the order of a.

THEOREM (Lagrange's Theorem). Let G be a finite group and let $a \in G$. Then the order of a divides the order of G.

More precisely, let $n = |G|$ be the order of G and let d be the order of , i.e. a^d is the smallest power of a that is equal to e. Then

$$a^d = e \quad \text{and} \quad d|n$$

CONJECTURE 1 It has been conjectured that there are infinitely many Sophie Germain Primes, but like the twin prime conjecture this too has not been proven.[1]

However if it is assumed that the Hardy-Littlewood estimate is true and approximates the value of the number of Sophie Germain primes correctly without a significant error as n, the number of Sophie Germain primes approaches infinity then a rough proof of the conjecture above has been outlined below.

$$N(n) = \frac{C_2 n}{(\ln n)^2}$$

$$\text{then, } \lim_{n\to\infty} N(n) = \lim_{n\to\infty} \frac{C_2 n}{(\ln n)^2} = C_2 \lim_{n\to\infty} \frac{n}{2(\ln n)}$$

$$= \frac{C_2}{2} \lim_{n\to\infty} n = \infty \text{ (applying L'Hopital's rule twice)}$$

this proves that there are infinitely number of Sophie Germain primes and therefore infinitely many safe primes taking the assumption above to be correct.

[1]Taken from an article on www.princeton.edu

3 The Pseudorandom Number Generator

DEFINTION Let $N = pq$ be a product of two safe primes p and q such that they are each equal to $2m+1$ and $2n+1$ respectively. Let k be any number such that $\gcd(k,pq)=1$, in other words $k \in F_N^*$ and such that $k^2 \neq 1 \pmod{N}$. Here k is called the 'seed' of the PRNG and N is called the modulus. The pseudorandom sequence generated by the generator with input (k,N) is the sequence of bits $b_1b_2b_3b_4b_5\ldots\ldots$ obtained by extracting the bit $b_i = parity(x_i)$ in other words $b_i = 1$ if x_i is odd and $b_i = 0$ if x_i is even, where $x_i = k^{f(i)} \bmod N$. Where $f(i)$ is any arbitrary function which successfully shuffles the bit sequence, without changing the period of the generator with respect to k. For the purposes of this paper $f(i)$ is taken to be the identity function such that $f(i) = i$. The period of the sequence so generated is always equal to or greater than the smaller of the two primes m or n.

For example let $N = 23 \times 47$ be the modulus and $k = 21$ be the 'seed' of the Pseudorandom Number Generator. Therefore with the input (21,1081), The sequence $x_i = (21,441,613,982,83,662,930,72,431,403.....)$ and its corresponding bit sequence $b_i = (1110100011....)$. Here the period of the sequence is equal to 506 which in this case is equal to $2mn$ if $m = 11$ and $n = 23$, which are the Sophie Germain equivalent primes of 23 and 47 that factor N.

THEOREM (period of the PRNG) Let $N = pq$ be a product of two relatively large safe primes p and q such that they are each equal to $2m+1$ and $2n+1$ respectively. Then the order of any element $a \in (Z/NZ)^*$ and $a^2 \neq 1 \pmod{N}$ has order greater than or equal to the smaller of the two primes m and n such that it is a factor of $4mn$.

Proof. According to the Euler's Formula for pq on page 4 for an element $a \in F_N^*$ $a^{\frac{(p-1)(q-1)}{g}} \equiv 1 \pmod{N}$, where $g = \gcd(p-1,q-1)$, however g in this case will always be equal to 2 as $\gcd(p-1,q-1) = \gcd(2m,2n) = 2$, since m and n are also prime, and as a consequence the term $\frac{(p-1)(q-1)}{g} = \frac{(2m)(2n)}{2} = 2mn$. Also the order of the group F_N^* is equal to $\Phi(pq) = \Phi(p)\Phi(q) = (p-1)(q-1) = 4mn$. Therefore according to Lagrange's Theorem any element $a \in F_N^*$ has an order that is a factor of $4mn$. Further assuming that a has an order of 4, then $\left(a^4\right)^x = a^{2mn} \bmod N$, which implies that $4x = 2mn$ which implies $x = \frac{mn}{2}$, which is not possible as m and n are prime, it therefore contradicts the assumption. Therefore the least order possible for $a \in F_N^*$ smaller than both m and n is 2.

REMARK The Hardy-Littlewood estimate and Conjecture 1 lead me to believe that it is relatively easy to find safe primes. Which can be found using the Miller-Rabin Test for Primality[2]. The algorithm I used to find a large value of N is simply to check each value for being a safe prime and then multiplying two such primes together.

CONJECTURE Although the idea of simply modifying an existing generator in order to get an 'even more random' sequence is often false[Knuth], I believe that taking an appropriate $f(i)$ function mentioned in the definition of the PRNG may result in an 'even more random sequence'

as compared to when $f(i) = i$, for example by taking it to be $f(i) = k_2^{\,i}$ where k_2 is a preset value of a constant which has followed the same rules for the value of the 'seed'(unexplored idea).

I will be referring to the following number as number (1) throughout this paper

1057689550064397758323064492852433663725447492742849950844129234061235094397443556084932321563691193523097380668936993032921 7

which is a product of 2 safe primes

3213876088517980551083924184682325205044405987565585670568399

and

3291009114642412084309938365114701009965471731267159726697207983

3.1 Python Code for finding these Primes[2]

```
def gcd(a,b):
    """ Calculates Greatest Common Divisor of 'a' and 'b'."""
    r = 1
    if b > a:
        a,b = b,a
    while r != 0:
        r = a%b
        a = b
        b = r
    return a

def FastPower(g,A,N):[3]
    """ Calculates (g^A) mod N."""
    a = g
    b = 1
    while A > 0:
        if A%2 == 1:
            b = (b*a) % N
        a = (a*a) % N
        A = A/2
    return b




def isWitness(a,n,k,q):
    """Checks to see if 'a' is a witness of 'n' where (n-1) = 2^k*q."""
    if gcd(a,n) > 1 and gcd(a,n) < n:
        return 1
    a = FastPower(a,q,n)
    if a%n == 1:
        return 0
    i = 0
    while i < k:
        if a%n == (n-1):
            return 0
        a = (a*a)%n
        i = i+1
    return 1
```

[2]The algorithm for the Miller-Rabin Test for Primality taken from [1]

[3]The name for this function has been borrowed from the Fast-Powering Algorithm which is described in [1]

```
def isDivby2357(n):
    """Checks to see if 'n' is divisible by 2,3,5 or 7."""
    if n%2 == 0 or n%5 == 0 or n%3 == 0 or n%7 == 0:
        return 1
    return 0

def isPrime(n):
    """Check to see if 'n' is prime (Checks for 30 witnesses,
       ie. probability of being wrong is 10^-19)."""
    if n == 1:
        return 0
    if n == 2:
        return 1
    if n == 3:
        return 1

    q = n-1
    k = 0
    while q%2 == 0:
        k = k+1
        q = q//2

    i = 0
    while i < 30:
        a = random.randint(2, n-1)
        if isWitness(a,n,k,q) == 1:
            return 0
        i = i+1
    return 1

def PUT(count):
    """Prints Primes up to 'count'."""
    counter = 0
    i = 2
    while i < count:
        if isPrime(i) == 1:
            print i,
            counter = counter + 1
        i = i+1
    return counter

def FPLTOET(number):
    """Returns First Prime Less Than Or Equal To 'number'."""
    while 1:
        if isPrime(number) == 1:
            return number
        number = number - 1


def PrimeForPsi(number):
    """Returns Prime number with approximately p/2 primitive roots, where p is the
prime."""
    while 1:
        if isPrime(number) == 1 and isPrime(2*number + 1):
            return (2*number + 1)
        number = number - 1
```

4 Statistical Tests

In this paper, I will use a couple of simple random number tests of Knuth [2] to test the PRNG described earlier. The first 'Chi-Square Test' test simply incorporates the 'number' of outcomes into its result whereas the second 'poker test' also takes into account the even distribution of those elements.

4.1 The Chi-Square Test (χ^2 Test) [2]

In this test each possible event s is assigned a probability p_s, Then the number of actual occurrences of that event Y_s are counted. After all the events are counted each Y_s is compared to the expected number of occurrences np_s, where n is the expected number of events in the test. If we sum the squares of the difference between Y_s and np_s with equal weight given to each possible event s we determine a measure of the difference between the expected and the actual results. This quantity V, can be expressed mathematically as

$$V = \sum_{s=1}^{\alpha} \frac{(Y_s - np_s)^2}{np_s}$$

and since

$$\sum_{s=1}^{\alpha} Y_s = n \quad \text{and} \quad \sum_{s=1}^{\alpha} p_s = 1,$$

$$V = \sum_{s=1}^{\alpha} \frac{(Y_s - np_s)^2}{np_s} = \sum_{s=1}^{\alpha} \frac{Y_s^2 - 2np_sY_s + n^2p_s^2}{np_s}$$
$$= \sum_{s=1}^{\alpha} \frac{Y_s^2}{np_s} - \sum_{s=1}^{\alpha} 2Y_s + \sum_{s=1}^{\alpha} np_s = \frac{1}{n}\sum_{s=1}^{\alpha} \frac{Y_s^2}{p_s} - 2n + n$$
$$= \frac{1}{n}\sum_{s=1}^{\alpha} \frac{Y_s^2}{p_s} - n$$

with the rule of thumb being that n should be taken such that each $np_s \geq 5$. Table 1 [4] gives us the values of V such that with ν degrees of freedom the probability p denotes the probability of the value obtained being lesser than the table entry in the ν row and under the p column. Here 'degrees of freedom' are the number of statistics that are free to vary. They are therefore for the purpose of this paper $\alpha - 1$.

Therefore, in the bit sequence $\langle b_i \rangle$ produced with the PRNG described above, there are 2 possibilities with $\alpha = 2$, $N = pq = 23 \times 47$ and the 'seed' $k = 21$, i.e. the sequence $\langle b_i \rangle$ produced with the input (21,1081) contains 258 zeroes and 248 ones.

[4] Table 1 is present in [2] chapter 3, page 44

$$V = \frac{(258 - ½506)^2}{½506} + \frac{(248 - ½506)^2}{½506}$$
$$= 0.19762844$$

similarly, with the seeds 62, 69, 73 and the modulus as the number (1), the values of V are the following : 0.8, 0.9604 and 0.162, each of which fall between the 25% and 75% range in Table 1 mentioned earlier. So, the pseudorandom number generator is neither 'Reject, 'Suspect or 'Almost Suspect'.

4.2 The Poker test (Partition test) [2]

The classical Poker test considers 'n' groups of five successive integers, $\{Y_{5n}, Y_{5n+1}, ..., Y_{5n+4}\}$ and observes which of the seven following patterns is matched by each (order-less) quintuple:

All different:	*abcde*
One Pair:	*aabcd*
Two Pairs:	*aabbc*
Three of a Kind:	*aaabc*
Four of a Kind:	*aaaab*
Five of a Kind:	*aaaaa*

A chi-square test is done based on the number of appearances of these categories with respect to their respective probabilities. For a simpler version of this test we can simply count the number of 1s appearing in a group of 5 bits of the pseudorandom sequence and then perform a chi-square test on them.

Taking our modulus in the PRNG to be the number (1) and the key to be '32138760885179805510839241846823252050444059875655856705683 98' the calculation for the value of V is as follows:

Y_0 = The number of quintuples with 0 ones = 18
Y_1 = The number of quintuples with 1 one = 74
Y_2 = The number of quintuples with 2 ones = 158
Y_3 = The number of quintuples with 3 ones = 142
Y_4 = The number of quintuples with 4 ones = 87
Y_5 = The number of quintuples with 5 ones = 21

Here in this test $\alpha = 5$ as there are 5 different possible outcomes, $n = 500$ as the number of bits generated by the PRNG is 2500,

$$p_0 = {}^5C_0\left(\frac{1}{2}\right)^5\left(\frac{1}{2}\right)^0 = \frac{1}{32},\ p_1 = {}^5C_1\left(\frac{1}{2}\right)^4\left(\frac{1}{2}\right)^1 = \frac{5}{32},\ p_2 = {}^5C_2\left(\frac{1}{2}\right)^3\left(\frac{1}{2}\right)^2 = \frac{10}{32},$$

$$p_3 = {}^5C_3\left(\frac{1}{2}\right)^3\left(\frac{1}{2}\right)^2 = \frac{10}{32},\ p_4 = {}^5C_4\left(\frac{1}{2}\right)^4\left(\frac{1}{2}\right)^1 = \frac{5}{32},\ p_5 = {}^5C_5\left(\frac{1}{2}\right)^0\left(\frac{1}{2}\right)^5 = \frac{1}{32},$$

where p_i represents the probability of the i^{th} possibility

$$V = \frac{1}{n}\sum_{s=1}^{\alpha}\frac{Y_s^2}{p_s} - n$$

$$= \frac{1}{500} \bullet \left[\left(\frac{18^2}{1/32}\right) + \left(\frac{74^2}{5/32}\right) + \left(\frac{158^2}{10/32}\right) + \left(\frac{142^2}{10/32}\right) + \left(\frac{87^2}{5/32}\right) + \left(\frac{21^2}{1/32}\right)\right] - 500$$

$$= 4.7552$$

in Table 1[Knuth] mentioned earlier, for 5 degrees of freedom, this value lies within the acceptable range of $p = 50\%$ and $p = 75\%$

4.3 Python Code for this test

```
def FastPower(g, A, N):
    a = g
    b = 1
    while A > 0:
        if A%2 == 1:
            b = (b*a) % N
        a = (a*a) % N
        A = A/2
    return b


def genSequences(k, N, num=500):
    resArr = []
    oneCount = 0
    for i in range(0, num*5):
        res = FastPower(k, i, N)
        elem = res%2
        resArr.append(elem)
        if(elem ==1):
            oneCount = oneCount + 1
        if len(resArr) == 5:
            yield(resArr, oneCount)
            resArr = []
            oneCount = 0


if __name__ == "__main__":
    fd = open("sequences.dat", "w")
    k = 32138760885179805510839241846823252050444059875655856705683 98
    N = 10576895500643977583230644928524336637254474927428499508441292340612350943974435560849323215636911935230973806689369930329217
    countArr = [0, 0, 0, 0, 0, 0]

    for tuple in genSequences(k,N):
        countArr[tuple[1]] = countArr[tuple[1]] + 1
        fd.write(str(tuple) + '\n')
    fd.write(str(countArr))
    fd.close()
```

5 Application in Public Key Cryptography (One-Time Pad)

An Army Signal Corp officer Joseph Mauborgne, proposed an improvement to an already existing cipher called the Vernam cipher that yielded the ultimate security in mathematical cryptography. He suggested using a key as long as the message itself in the encryption and decryption functions. Such a scheme is therefore called a One-Time Pad.

Let the key bit be k_i, the message bit m_i and the cipher text bit be c_i, then

$$c_i = m_i \oplus k_i$$
$$m_i = c_i \oplus k_i$$

In this paper an application the PRNG described in the previous sections has been proposed to be used in a public key One-Time Pad. The security lies in the problem of factoring large numbers and in the problem of solving $x^a \equiv c \pmod{N}$ where N is the product of two large primes[5]. Supposing a situation in which 'Bob' wants to send a message over to 'Alice' with 'Eve' trying to break the cryptosystem.

THE ALGORITHM

Alice : Alice secretly chooses two relatively large safe primes p and q, where $p = 2m+1$ and $q = 2n+1$ then computes $N = pq$ and publishes this N.

Bob : If Bob wants to send an n-bit message $m = (m_1 \ldots\ldots\ldots m_n)$, he picks any k such that $\gcd(k,N) = 1$ and $k^2 \neq 2 \pmod{N}$. He generates the bit sequence $b_i = (b_1 \ldots\ldots\ldots b_n)$ from the sequence $x_n = (x_1 \ldots\ldots\ldots x_n)$ obtained using the PRNG with the input (k,N) and then computes the cipher text $c_i = (m_1 \oplus b_1 \ldots\ldots\ldots m_n \oplus b_n)$ where $c_i = m_i \oplus b_i$ and then publishes c and x_{n+1} if n is even OR x_{n+2} if n is odd.

Alice : Since Alice knows the factors of N, i.e. p and q. She can efficiently use Proposition 2 to solve the congruence $k^{n+1} \equiv x_{n+1} \pmod{N}$ if n is even OR $k^{n+2} \equiv x_{n+2}$ if n is odd (although the parity of the power is inconsequential to Eve). She can then reconstruct the message by simply generating the pseudorandom bit sequence $b = (b_1 \ldots\ldots\ldots b_n)$ using the PRNG with the input (k,N) and then computing $m = (c_1 \oplus b_1 \ldots\ldots\ldots c_n \oplus b_n)$.

This algorithm can be said to be secure because Eve will have a hard time recovering the 'seed' k from the data sent to Alice because of the problem Eve will face in factoring the large number N and therefore in solving the congruence $k^{n+1} \equiv x_{n+1} \pmod{N}$ if n is even OR $k^{n+2} \equiv x_{n+2} \pmod{N}$ if n is odd.

NOTE x_{n+1} OR x_{n+2} are sent because in order to use Proposition 2 to solve the congruencies the exponent of k need to have the highest common factor with $(p-1)(q-1) = (2m+1-1)(2n+1-1) = 4mn$ equal to 1 but all even numbers will share 2 as a common factor with it, hence the parity of the exponent makes a difference. If a different $f(i)$ is used then the parity of the exponent will have to be checked accordingly.

6 Scope for further investigations

The ideas for the function $f(i)$ mentioned earlier can be further investigated and the concept of using a pseudorandom number generator as a hash function can be further investigated. An abstract model of this concept implemented in C++ is given below.

```
char* HashDigest(char* arr, int lambda, int ClassSize = 0) {
     /* This Hash function converts the required value, i.e. the object of any misc data type, or a string
by utilizing a pointer to the first byte of the data. It then starts converting each byte into a binary
string of length lambda and then to hexadecimal form using the pseudorandom number generator and keeps
```

```
modding the value to a FinalHash string which the function then returns. IntoBinary(x, y, z) converts y to a
pseudorandom string of length z and stores it in x. */

      char* ptr = arr;
      int phi;
      int fordifference = FastPower(lambda,2,PRIME);
      if (ClassSize == 0)
        phi = stringLength(arr);
      else
        phi = ClassSize;

      char* BinaryString = new char[4*lambda + 1];
      char* Hexed;
      char* FinalHash = new char[lambda + 1];

//Gives an initial value to FinalHash based on the length of the object or string being hashed
      for (int i = 0; i < lambda; i++) FinalHash[i] = '0';
      intoBinary(BinaryString, phi, 4*lambda);
      Hexed = Hex(BinaryString); // Converts BinaryString to its hexadecimal equivalent
      Mod(FinalHash, Hexed, lambda);
      delete[] Hexed;

      for(int i = 0; i < phi; i++) {
      // Converts the byte to binary of length 4*lambda using the byte as the seed to the PRNG
intoBinary(BinaryString, *ptr + i + fordifference,4*lambda); // so thqt each ele gets a //diff value on the
virtue of its position and lambda

        //SpecificPermute(BinaryString, strlen(BinaryString), arr, strlen(arr));
        Hexed = Hex(BinaryString);

        //SpecificPermute(Hexed, strlen(Hexed), arr, strlen(arr));
        Mod(FinalHash, Hexed, lambda);
        ptr++;
        delete[] Hexed;
      }

      FinalHash[lambda] = '\0'; //To make it a readable string
      delete[] BinaryString;
      return FinalHash;
    }
```

where the `intoBinary()` function can be

```
void intoBinary(char arr[], int byte, int LENGTH) {
char temp[5];
for(int i = 0; i < LENGTH; i++) {
byte = FastPower(byte, i, PRIME); // PRNG
    }
    arr[LENGTH] = '\0';
    }
```

References

[1] An Introduction to Mathematical Cryptography by Jeffrey Hoffstein, Jill Pipher and Joseph H. Silverman.

[2] The Art of Computer Programming, Seminumerical Algorithms (Volume 2) by Donald E. Knuth

[3] A Comparison of Two Pseudorandom Number Generators by Lenore Blum, Manuel Blum and Michael Shub

[4] www.wikipedia.org

[5] www.princeton.edu