Xorshift PRNG

Xorshift pseudo random number generators (PRNG) Proposed by George Marsaglia in his paper ‘Xorshift RNGs’ published in 2003.

“a class of simple, extremely fast random number generators (RNGs) with periods 2k −1 for k = 32, 64, 96, 128, 160, 192.”. (Marsaglia, Xorshift RNGs, 2003)

A pseudo random number generator is a function that outputs a random number created using some form of deterministic process. In Xorshift’s simplest form this is achieved by applying two bitwise operations on a given 32-bit unsigned integer seed, for this you can use any available source of entropy. First a shift left, or right operation is applied to the seed followed by a xor operation between the original seed and the shifted value. This is then repeated two more times with shifts of differing lengths. The value that is returned is used as the seed for the next iteration. The set of results produced will have a period of 2^{32}-1. An implementation with a period of 2^{64}-1 using certain shift values is capable of passing Marsaglia’s own "Diehard battery" of statistical tests for randomness.

Random number are useful for many different requirements such as generating random numbers for gambling or creating unpredictable results in a computer game, but more importantly randomness is major tool in cryptography. Most cryptographical security systems will rely on random numbers, if a number can be guessed the system will become vulnerable to attacks. A good example of this is HTTPS where the use of randomly generated encryption keys is used to scramble and unscramble data. Without the ability to produce reliably random numbers the ability to assure any form of security on the internet would not be possible.

Pseudorandomness is the measurement of how unpredictable a sequence of numbers is when the sequence is generated through a deterministic process.
For an algorithm to be considered a PRNG it must meet the criteria for quality of deterministic random number generators put forward by The German Federal Office for Information Security (*Bundesamt für Sicherheit in der Informationstechnik*, BSI).
When generating random numbers, the “period” measures the number of output bits that will be generated before the output begins to repeat itself. The validity of a random number generator for cryptographical means is not really dependent on the period produced, however any random number generator that has a period over 2^{64}-1 is considered nontrivial as it can no longer be realistically explored.
Although Xorshift cannot stand up to the same level of scrutiny as a cryptographically secure pseudorandom number generator (CSPRNG) it is among the fastest non-CSPRNG. Another benefit of Xorshift is that its weaknesses can be easily amended with the addition of a nonlinear function such as Xorshift+ and Xorshift* allowing it to pass all of the tests in the BigCrush suite.

There are several suites that have been created to evaluate the quality of PRNG’s. The suites are usually composed of multiple randomness quality tests. Two of the most notable test suites are George Marsaglia’s "Diehard battery" of statistical tests for randomness, and the TestU01 suite that includes “small crush”, “crush” and “big crush”. TestU01 is the current standard for testing the randomness of RNG’s, having supplanted the "Diehard battery".

The Xorshift algorithm uses two bitwise operations a logical shift and an xor comparison. The logical shift has two variations: the shift left and shift right. Both shift the bits of the operand left and right respectively. A logical shot uses the notation << for a shift left and >> for a shift right with the bit string on the left and the number of bits to be shifted on the right. The then vacant bit positions are filled usually with zeroes.

Logical shifts are also an efficient way of performing multiplication and division by powers of 2. This is because moving a 1 bit from position k to position k+1 means it is now worth 2^{k+1} so for example a 1 bit at position 1 is worth $2^1 = 2$ if it is shifted by 2 to position 3 it is worth $2^3 = 8$. If you shift a bit string to the left by n you are multiplying it by 2^n for both signed and unsigned integers. You can also divide by 2^n in this manner by shifting right this is only possible with unsigned integers as the most significant bit in a signed integer is reserved for distinguishing between positive and negative.

The xor operation or the exclusive or is a logic gate that compares two Boolean inputs and outputs true if and only if one of the inputs are true. This means that the xor gate is able to distinguish inequalities between inputs. An xor gate is not a basic logic gate in that it is created from the combination of basic logic gates. We can see this if we express it algebraically as $A \oplus B$ that is equal to $(A \wedge \neg B) \vee (\neg A \wedge B)$ or $(A \vee B) \wedge (\neg A \vee \neg B)$.

This is xor expressed as a truth table

Xor is also a useful tool for binary addition as when adding two single binary bits xor is equal to the sum of the two bits modulo 2. As you can see from this table.

An example of this is a half adder that sums two single binary digits by combining an XOR operation that outputs the sum $S = A \oplus B$ as well an AND operation that outputs the carry $C = A \wedge B$.

The problem is that a half adder can only output a carry bit and cannot handle an input carry bit this can be solved with a full adder that takes in a carry bit and adds it to the next calculation.
An implementation of which is:
$S = A \oplus B \oplus C_{in}$ and $C_{out} = (A \wedge B) \vee (C_{in} \wedge (A \oplus B))$

The following is pseudo code for implementations of Xorshift 32, 64 and 128:

```
    Key:
    ^ = Xor operation
    << = Logical shift left
    >> = Logical shift right

32-bit unsigned int seed
Function Xorshift_32()
    32-bit unsigned int x
    x = seed
    x = (x ^ (x << a))
    x = (x ^ (x >> b))
    x = (x ^ (x << c))
    return seed = x
end function
```

This implementation of Xorshift will produces sequence of 32-bit words with a period of $2^{32}-1$. marsaglia suggests 81 triples (a, b, c), a < c. he also states that you can use any of the 8 possible combinations of the left and right shift creating 8x81=648 possible choices that produce a period of $2^{32}-1$.

The 64-bit implementation of Xorshift is functionally the same as the 32-bit implementation. Though increases the state of the algorithm by simply increasing the size of the bit string 'seed' producing a sequence of 64-bit words with a period of $2^{64}-1$. marsaglia suggests 275 triples that will produces a sequence with a period of $2^{64}-1$ meaning 8x275 = 2200 choices.

```
32-bit unsigned int w,x,y,z
Function Xorshift_128()
        32-bit unsigned int t,s
        t = z
        s = w

        z = y
        y = x
        x = s

        t = (t ^ (t << a))
        t = (t ^ (t >> b))
        return w = t ^ (s ^ (s >> c))
end function
```

This implementation of Xorshift will produces a sequence of 32-bit words with a period of $2^{128}-1$. This is accomplished by increasing the state by using four 32-bit unsigned int w,x,y,z as seeds. The values w,x,y,z are used as a queue where the head 32-bit word 'z' is used for the first two Xorshift operations and the tail 32-bit word 'w' is used for the final Xorshift operation that is xor'd again with the head 'z' to produce the returned 32-bit word. This becomes the new tail 'w' while all values are shifted up one position.

As previously mentioned, although Xorshift is an effective PRNG it will fail to pass all of the tests of "big crush" in the TestU01 battery. There have been several variations of Xorshift that have been suggested to improve its performance. One of the largest improvements and most widely used variations is Xorshift128+. This variation uses addition as a non-linear transformation to further scramble the output.

The following is pseudo code for Xorshift128+:

```
32-bit unsigned int x,y
Function Xorshift_128+()
        32-bit unsigned int t,s
        t = x
        s = y
        x = s
        t = (t ^ (t << a))
        t = (t ^ (t >> b))
        t = (s ^ (s >> c))
        y = t
        return t + s
end function
```

This variation will now pass all of the test of “big crush”.

One of the benefits of Xorshift is how fast it performs. Marsaglia states that “Xorshift on an 1800MHz PC, xor128() takes 4.4 nanoseconds (> 220 million numbers/sec)”.
(Marsaglia, Xorshift RNGs, 2003)
We can see the efficiency of Xorshift by calculating the run time:

```
32-bit unsigned int seed        constant time           times executed
Function Xorshift_32()
        32-bit unsigned int x
        x = seed                  =>    1                       1
        x = (x ^ (x << a))        =>    3                       1
        x = (x ^ (x >> b))        =>    3                       1
        x = (x ^ (x << c))        =>    3                       1
        return seed = x           =>    1                       1
end function
```

$T(n) = C_1 + C_2 + C_3 + C_4 + C_5$ where C_i is the constant time at the line i.

From this we can see that T(n) is not bounded by an input so it has a constant time this means that the time complexity of Xorshift is O(1).

There are many options when deciding on an algorithm for pseudo random number generation. When evaluating PRNG’s you will have to make a decision between the cost of the algorithm and the quality of randomness produced. A cryptographically secure pseudorandom number generator’s (CSPRNG) are a class of PRNG that produce words that are considered to be of a quality for use in cryptology.
A CSPRNG has to meet all of the requirements of a PRNG. It will also have to be able to pass a next bit test where given the first k bits of a random sequence, there is no polynomial-time algorithm that can predict the (k+1)th bit with probability of success better than 50%. It also has to withstand "state compromise extensions" this is a test to prove that it is impossible to reconstruct the stream of random numbers produced even if part or all of its state have been revealed. (Cryptographically secure pseudorandom number generator, n.d.)

Although CPRNG's produce a higher quality randomness the cost of the algorithms is generally a lot higher especially when compared to PRNG's such as Xorshift this makes them unviable in certain scenarios.
A more important comparison would be between Xorshift and other PRNG family's such as the Mersenne Twister.

Mersenne Twister is considered the most widely used PRNG "The basic idea is to define a series $x_i$through a simple recurrence relation, and then output numbers of the form x_iT, where T is an invertible F_2 matrix called a tempering matrix." (Mersenne Twister, n.d.). The algorithm passes many tests for statistical randomness including the Diehard tests and most, but not all of the TestU01 tests. This very close to the performance of Xorshift128+ it similarly has a time complexity of O(1). Although the two algorithms are similar in performance Mersenne Twister has some drawbacks its state buffer is larger than Xorshift as it requires keeping a table of the k most recent values rather than the different sets of 32-bit seed variable required for Xorshift variations. However, the Mersenne Twister produces much large periods such as 2^{131102} rather that the 2^{32}-1, 2^{64}-1, 2^{128}-1… produced by Xorshift. it can also take a long time to start generating outputs that pass randomness tests meaning that two instances of the algorithm with similar states will output sequences that are very similar for many iterations. It seems that if you are comfortable having the smaller periods produced by Xorshift it becomes a viable choice.

Xorshift is clearly a very competent PRNG performing extremely well because of it only relying on simple, fast operations and having a very small state. Because variations of Xorshift are capable of passing "Big Crush" without a noticeable loss of performance it a viable choice of PRNG especially when performance is a major factor and cryptographic level of security is not an issue. Although it is not nearly as widely used as the Mersenne Twister its Xorshift128+ variation has become a replacement for JavaScript's math.random() in all of the more commonly used browsers such as Google Chrome, Firefox and Safari.

I believe that George Marsaglia was correct in his evaluation of Xorshift.
"(Xorshift) is worth considering as a workhorse RNG. Particularly when such xorshift RNGs are so simple, so fast and do so well on tests of randomness." (Marsaglia, Xorshift RNGs, 2003)

Bibliography

Bitwise operation. (n.d.). Retrieved from En.wikipedia.org: https://en.wikipedia.org/wiki/Bitwise_operation#XOR

Cryptographically secure pseudorandom number generator. (n.d.). Retrieved from en.wikipedia.org: https://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_number_generator#Requirements

Logical shift. (n.d.). Retrieved from En.wikipedia.org: https://en.wikipedia.org/wiki/Logical_shift

Marsaglia, G. (1995). The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness.

Marsaglia, G. (2003). Random Number Generators. *Journal of Modern Applied Statistical Methods, 2*(1), 1-13.

Marsaglia, G. (2003). Xorshift RNGs. *Journal of Statistical Software, Articles, 8*(14), 1-6.

New pseudorandom number generator in all major browser. (2016). Retrieved from Complexitybiosystems.it: https://complexitybiosystems.it/en/news/new-pseudorandom-number-generator-in-all-major-browser-designed-by-sebastiano-vigna

Pseudorandom number generator. (n.d.). Retrieved from En.wikipedia.org: https://en.wikipedia.org/wiki/Pseudorandom_number_generator

Pseudorandomness. (n.d.). Retrieved from en.wikipedia.org: https://en.wikipedia.org/wiki/Pseudorandomness#cite_note-RandomArticle_Phys.NYT2001-1

XOR gate. (n.d.). Retrieved from En.wikipedia.org: https://en.wikipedia.org/wiki/XOR_gate#:~:text=XOR%20gate%20(sometimes%20EOR%2C%20or,to%20the%20gate%20is%20true.

Xorshift. (n.d.). Retrieved from En.wikipedia.org: https://en.wikipedia.org/wiki/Xorshift#xoshiro256+