Comparison of Pseudorandom Number Generation Algorithms

Bradley Allen
Department of Computer Science
University of South Carolina Upstate
Spartanburg, South Carolina, USA
btallen@email.uscupstate.edu

ABSTRACT

Random numbers are used in many different aspects of the modern world. A couple examples of the uses of a random number include security encryption to prevent unauthorized access to private data and a game that will deal damage based on the generated number. While one example is used for recreational use, the other is required for use in keeping private data private. In this paper, pseudorandom number generation will be discussed and two generation algorithms will be compared to determine which algorithm is overall better.

Keywords

Pseudorandom Number Generator, PRNG, RNG, Cryptography, Linear Congruential Generator, LCG, XOR Shift, Algorithm, Gap Test, Eclipse IDE, Eclipse, IDE

1. INTRODUCTION

Computers have been built like any other machine, just more complex. At the core of computing there are electrical signals, signals that operate at a fraction of a second and are either there or not. These signals are used to determine whether the processor of the computer needs to compute arithmetic, load information from a memory storage device, or even render a three-dimensional simulated space. Binary is a base-2 number system often used to understand the instructions of the computer. Since all instructions are deterministic in nature by use of these binary electrical signals, instructions are impossible to be completely random.

A pseudorandom number generator is used to generate seemingly random numbers while using deterministic computer algorithms. The purpose of this study is to determine which generation algorithm is superior, whether a linear congruential generator algorithm (LCG) (which uses the following equation, where S is the seed, M is the multiplier, N is the increment term, and R is the pseudorandom number)

$$R = \big((S * M) + N\big) \,\%\, mod$$

or an XOR shift algorithm (which raises the seed to itself multiple times after shifting the bits left or right by a determined amount [1][2]) operates in less time and with more "random" numbers generated. In this study, the dataset generated by the PRNGs will be 1,000 randomly generated numbers and will be analyzed using various statistical methods to determine which generator is more random and which executes at a faster time.

2. LITERATURE REVIEW

While the two algorithms are used in pseudorandom number generation, studies done on generations use these generators primarily for cryptography. [3]

A study done by Ahmad Gaeini, et al. shows a comparison of multiple algorithms, one being a standard LCG. According to the study's findings, a relatively simple generator such as the LCG still passes tests for randomness in the generated numbers. In addition to passing randomness tests, the generator also proved to be resistant against cryptography attack attempts. For a test of the law of iterated algorithms, however, this simple generator failed [4].

For the approach used in the study, it has been concluded that a simple LCG would not be used as a means of securely encrypting data. In this study, the PRNGs will not be tested in secureness directly, but the randomness/gap test will be giving an approximate secureness. Referencing the study, LCG's generated numbers are pseudorandom and the algorithm itself is simple, but the numbers repeat quickly due to a low range; this means to achieve better performance the LCG should be used with a different seed each run to generate the number sets.

3. METHODOLOGY

The linear congruential algorithm, as previously stated, will be using the following equation, where S is the seed, M is the multiplier, N is the increment term, and R is the pseudorandom number.

$$R = \big((S * M) + N\big) \,\%\, mod$$

The XOR shift algorithm will consist of raising the seed to itself multiple times after shifting the bits left or right.

The advantages of the linear congruential algorithm is that it is simple and fast. The equation used in the algorithm to generate a pseudorandom number is a simple mathematical equation with four variables. Since the equation is not too complicated, the computer can also do this generation relatively quickly because it requires a few ALU cycles to solve. The disadvantages, however, makes this algorithm much less useful in modern applications. The algorithm itself has a low range, meaning the numbers generated will repeat quickly. The algorithm, depending mainly on the modulus used, can only make about 10 random numbers before repeating if the modulus is below 100. The repeating itself is not random and will follow the same pattern regardless how many numbers are generated.

The main advantage over the XOR shift algorithm is that it is more random. This advantage will be tested in the study, but it is proven to have a higher range since the numbers are not generated based on a mathematical function, but instead shifting bits and raising the seed. The disadvantage of this algorithm is that it is more complex than the simple equation of the LCG algorithm. While this means that the algorithm takes more time to understand in human terms, the computer can still process the information at the same speed as the LCG (which will also be tested to see how close the two speeds are).

4. IMPLEMENTATION

This study will be conducted using the Java programming language. For simplicity of running the code, the code will be written, compiled, and executed using the Eclipse IDE made by the Eclipse Foundation [5]. Methods used will be built-in to Eclipse and in the Java System class and Math class. To create the seed, System.nanoTime() will be used. This will provide a more seemingly random input and thus a relatively random output. To avoid any negatives that will skew results, the Math class's absolute value method will be used (abs).

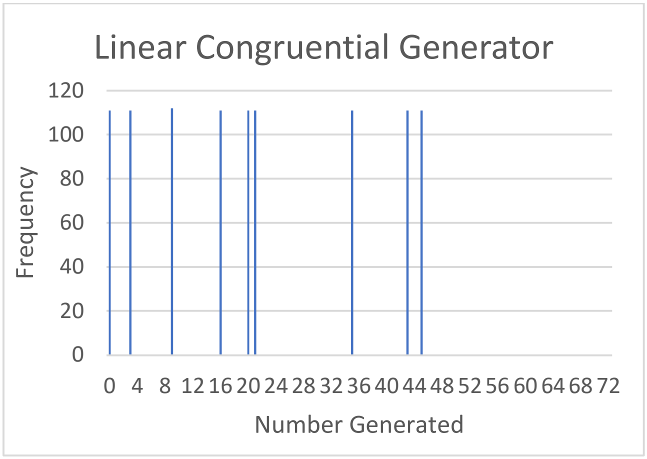

Figure 1: LCG generated numbers (showing low range)

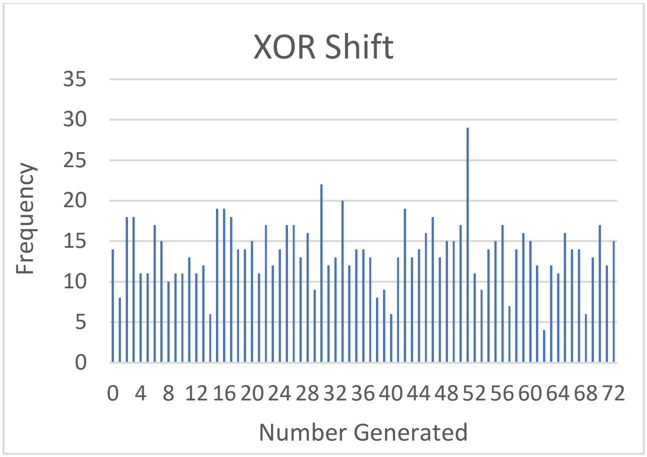

Figure 2: XOR Shift generated numbers

5. EXPERIMENTAL SETUP

In the study, the seed, modulus, and amount of numbers generated will remain as constant variables. The seed, as mentioned previously, will be defined to be System.nanoTime() for both implementations. The modulus, used to limit the output to a range of numbers to compare the two algorithms, will be set to 73 for both algorithms. 73 is prime and a large enough number to compare the two algorithms. The numbers generated will be set to 1,000,000 to be enough to compare and equal out any possible outliers in the data.

The experiment will compare the overall randomness of the output data, or the repetition of numbers generated. To standardize comparison, the gap test will be used. The gap test measures the distance between successive occurrences of a digit [6].

In addition to the randomness of the numbers, the performance of each algorithm will also be compared. This includes the speed of execution and will be an average of the runs.

6. RESULTS ANALYSIS

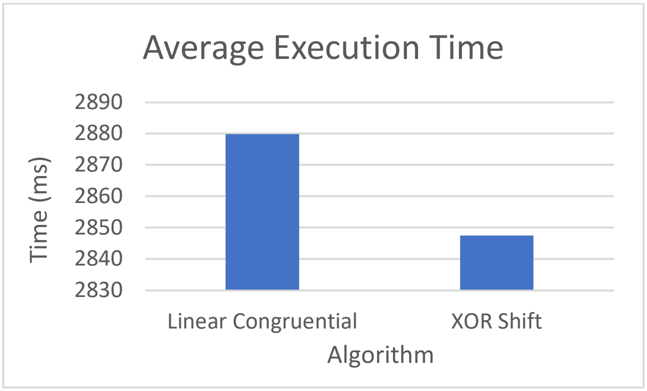

Figure 3: LCG generated numbers (showing low range)

Figure 3 shows the average execution time of the program with 1,000,000 numbers generated where the program was ran 6 times to minimize discrepancies. In the figure, the linear congruential algorithm is shown to take around 2,880ms (2.88 seconds) and the XOR shift algorithm is shown to take around 2,850ms (2.85 seconds).

A visual comparison for randomness can be done using figure 1 and 2. The more "random" an algorithm is, the less frequent each number in the range would be generated. As previously stated, the LCG implementation has a low range but the numbers themselves are random, meaning if only four numbers were generated, it would be deterministically randomized. Figure 1 shows that, with 1,000 numbers generated, the frequency for each number is extremely high (111 average) compared to figure 2's XOR shift frequencies. Since XOR shift does not rely on accumulation, the range is much higher, leading to a lower frequency of each number (29 being the highest frequency). [3]

In addition to comparing frequencies, comparing any gaps the data has would also be useful in comparing randomness. While this is typically done through mathematical equations, for this specific data this can be done through visual observation. Figure 1 shows the low range of LCG and thus, large gaps between each number generated throughout the entire range. For comparison, figure 2 shows no gaps where no number was generated, but does have less frequency around 14, 40, and 62. While this could prove to be a discrepancy due to only 1,000 numbers being generated, this result would fail cryptography tests as it shows gaps in frequencies around certain parts of the range that could be exploited. [1]

7. CONCLUSION

While this study goes over two common, primitive methods of pseudorandom number generation, the direct comparison of the two is unique. The linear congruential generation is simple to learn, as the operations are algebraic while the XOR shift is more difficult to learn as the operations are bitwise manipulations. Simple to learn, however, does not necessarily mean better as it was shown that the execution time of the LCG was higher than the XOR shift. [7][8]

Stated previously, the direct comparison of execution time proves that, when generating a large amount of numbers, the XOR shift implementation finishes execution sooner than the linear congruential implementation.

To further compare, it is shown that XOR shift is more random than the linear congruential generator due to a lower frequency of generated numbers within the range and no gaps where no number was generated.

For the comparison of these two implementations, with all three metrics considered, it is shown that the XOR shift implementation for pseudorandom number generation is faster and more random than the linear congruential generation.

8. REFERENCES

[1] Stallings, William. “Entitled Cryptography and Network Security: Principles and Practice,” 7th Edition, London EC1N 8TS: Pearson Education, 2017.

[2] “Pseudorandom Number Generators.” Oracle Help Center, 14 July 2022, https://docs.oracle.com/en/java/javase/17/core/pseudorandom-number-generators.html.

[3] “XORShift Random Number Generators.” Javamex. https://www.javamex.com/tutorials/random_numbers/xorshift.shtml.

[4] Gaeini, Ahmad, et al. "Comparing some pseudo-random number generators and cryptography algorithms using a general evaluation pattern." IJ Information Technology and Computer Science 9 (2016): 25-31.

[5] “The Eclipse Foundation.” https://www.eclipse.org/org/foundation/.

[6] “Generating Random Data.” Generating Pseudorandom Numbers - MATLAB & Simulink, https://www.mathworks.com/help/stats/generating-random-data.html.

[7] Arobelidze, Alexander. “Random Number Generator: How Do Computers Generate Random Numbers?” FreeCodeCamp.org, FreeCodeCamp.org, 8 June 2021, https://www.freecodecamp.org/news/random-number-generator/.

[8] Chapter 3 Pseudo-Random Numbers Generators - University of Arizona. https://www.math.arizona.edu/~tgk/mc/book_chap3.pdf