What is the most appropriate random number algorithm for procedural content generation within game development?

**Name:** Samuel Tebbs

**Subject:** Computer Science

**Supervisor:** Mr Coetzee

**School:** Dartford Grammar School

**Word count:** 3982

# 

# Abstract

This essay discusses the requirements of a random number algorithm/generator for use in procedural content generation, specifically within the field of video game development, and how the limitations of modern computers impact the feasibility of each algorithm. The individual technical details, flaws and merits of four random number algorithms are explored and are then compared through the use of various statistical tests, in order to determine their suitability. Based on the merits of each algorithm and the analysis of the results produced, the conclusion drawn was that the *Mersenne twister* algorithm proved to the most appropriate for use in procedural generation, however the *Fibonacci LFSR* provides a substitute where time is of the essence and a smaller period is acceptable.

**Word count:** 120

Table of Contents

Introduction 4

Procedural generation 4

Random number generator/algorithm 4

Video game 4

Capabilities and limitations 5

Procedural generation 5

Random number algorithms 6

The Algorithms 7

Criteria 7

Selected algorithms 7

Linear Congruential Generator 8

Xorshift 10

Fibonacci Linear Feedback Shift Register 11

Mersenne Twister 13

Statistical Tests 14

Runs 14

Monkey test 15

Method 15

Results 16

Analysis 17

Conclusion 17

Bibliography 19

Books 19

Journals 19

Web pages 19

Appendix 21

A: Raw statistical data 21

B: Statistical test program 24

# Introduction

Since the conception of games such as *Pong* and *Spacewar!*, video games have been, along with business and education, a significant driving force behind the success of the personal computer market, providing a huge audience of people fascinated by the new technology’s applications within entertainment. However, due to the very limited hardware and software capabilities of the day, developers were forced to come up with unique and innovative techniques in order to overcome technological limitations and to provide an interactive experience to their users. One such technique is procedural generation, a technique that is becoming increasingly more relevant to the video games industry as the demands become more advanced, sometimes requiring development teams the size of film studios.

## Procedural generation

Procedural content generation (PCG) is the programmatic generation of game content using a random or pseudo-random process that results in an unpredictable range of possible game play spaces[[1]](#footnote-1). An algorithm that implements procedural generation takes random numbers as its input and outputs any type of contents that the developer desires.

## Random number generator/algorithm

Random number generation is the generation of an unpredictable sequence of numbers that appears to be randomly distributed. A random number algorithm is a mathematical or computational algorithm that produces such a sequence of numbers.

## Video game

An game played on an electronic device that receives user input, processes it and then sends some form of feedback (video, sound etc.) to the user.

# Capabilities and limitations

## Procedural generation

In most cases of its use, procedural generation takes place while the software is running, generating and creating any type of content that the developers wish. It is implemented in software through the use of one or more algorithms that take random sequences of numbers as the input and output limitless types of data. For example, if the developer’s aim is to create a collection of planets, each with its own set of data, such as economy and population size, an input value of one may assign it an “Agricultural” economy type while a value of two may assign it an “Industrial” economy type. This number then multiplied by the second number in the sequence could give the population size in billions. All of this data would then be stored in a pre-determined data structure for later retrieval.

A video game that used a similar, albeit more complex approach is *Elite*, a game published by Acornsoft in 1984 and developed by David Braben and Ian Bell. The game contained 8 galaxies, each of which contained 256 (the quantity of numbers that can be stored in 8 bits) planets, which were created and then stored using procedural generation, an impressive feat considering the limited amount of memory available and slow processors (compared to modern standards) used.

One benefit of procedural generation is that it aids in alleviating the workload from people working on a project such as graphic designers[[2]](#footnote-2) and hands it over to the software at run-time. A clear effect of this is that it gives teams of only a few developers a chance at achieving the same level of immersive complexity as entire companies are able to.

Yet another benefit of implementing procedural generation is that of replayability. If the bulk of the content in a game is procedurally generated and the random number generator’s seed (discussed later) is different every time, the user will experience drastically different content each time the generation algorithm is put to use, hugely increasing the replayability factor of a game.

However, a limitation of the use of procedural generation is the abilities of the computer on which the software runs. Since the data and content is usually generated as the software is running, competent systems are required to generate rich content without disrupting the user’s experience of the software.

## Random number algorithms

Random number algorithms have a number as an input (the seed), process the number one or more times and then output a random sequence of numbers. The sequence is the result of a function in the form of where the previous element of the sequence or in the case of 0, the seed.

There are two types of random numbers that the sequences can consist of.

* **True random numbers:** These are numbers that are not solely generated using any concrete formula, algorithm or equation and within the realm of computing, often depend on readings from equipment such as Geiger-counters (detect the random emission of particles by radioactive substances) or atmospheric noise detectors.
* **Pseudo-random numbers:** These are numbers that depend on an explicit formula, algorithm or equation but when in a sequence, have no obvious relation to each other, and therefore appear random. Although they do exhibit repeatable patterns and re-occurrence of numbers[[3]](#footnote-3).

Since computers depend on an explicitly stated program, algorithm and instruction set, they are unable to produce truly random sequences of numbers without the use of external equipment, therefore the term *pseudo-random* is more appropriate and will define the further usage of the word *random*. Because each element of a randomly generated sequence of numbers is the direct result of a function of its previous element (we say that it shows determinism), it is inevitable and clear that the sequence will return to the seed value and repeat itself. The quantity of numbers that can successfully be generated by the algorithm up until this point is called the *period.* A random number algorithm of high quality will have a very long period, most usually something similar to , where the integer length in bits.

Many random number algorithms use a bitwise operation called *exclusive-or* (xor) that outputs true whenever the two inputs (A and B) are unequal (see truth table below).

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **Output** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

For example, the result of an xor operation on the binary integers 01100101 and 10101001 would be 11001100.

# The Algorithms

## Criteria

In order to effectively compare several different algorithms, certain criteria must be put in place so that they can be tested on equal grounds without unnecessarily stressing the computer. The criteria chosen are as follows:

* **32-bit integer compatibility:** The algorithms must be compatible with 32-bit integers so that the standard *int* variable data type common to most modern programming languages can be used. This requirement was chosen to ensure compatibility and reliability among a wide range of modern systems, since 32-bit computing has become the standard.
* **Independency of external equipment:** Excluding standard personal computing hardware, no extra equipment should be required in order to write, run and test the algorithm, so that it can be utilised by a wide range of systems and users. External equipment includes tools such as cameras and sensors.
* **Implementation in C++:** The C++ programming language was chosen as the standard language for testing the algorithms due to its bitwise operation capabilities, therefore all algorithms must exist or be trivial to implement in C++ or C form. It is also acceptable for the algorithm to exist in pseudocode form for further translation into C++.

## Selected algorithms

After reviewing the above criteria, the following algorithms were selected for use in the statistical tests.

* Linear Congruential Generator
* Xorshift
* Fibonacci Linear Feedback Shift Register
* Mersenne Twister

Three of the selected algorithms are part of a family of random number algorithms called *Linear feedback shift registers* (LFSR), which when implemented in hardware, consist of an array of flip-flops (a circuit with two stable states that is used to store one bit of data) that share the same clock and whose state is directly dependent on a linear function of its previous state. Some linear feedback registers use the term *tap*, which is a certain bit in a byte or word that is used in a linear function to affect the next state.

## Linear Congruential Generator

The linear congruential generator is a random number algorithm that was introduced by D. H. Lehmer in 1949 and forms the basis of the most popular random number algorithms in use today[[4]](#footnote-4). It uses three constant integers and one varying to generate a sequence of random numbers with the following equation5.

Where

* constant modulus,
* constant multiplier,
* constant increment,
* element of the sequence
* the modulus function (remainder after a cyclic division)

Fig. 1

input

Increment

The algorithm starts by calculating and uses the given seed as the value which is then multiplied by , added to and used as a parameter to the modulus function along with . The second value is then calculated in the same way but with as the value, this continues until the desired quantity of values for the sequence has been calculated.

The most efficient linear congruential generators use powers of two as the value of , since they allow the modulus function to be computed faster (by truncating the lower bits) than non-powers of two[[5]](#footnote-5), and since 32-bit integers will be used, a value of will be used as . The period of a linear congruential generator can at most be if the following conditions are met[[6]](#footnote-6).

* is relatively prime to
* is divisible by all prime factors of
* is a multiple of 4 if is a multiple of 4

When considering the above conditions, the following integer constants that are used in the ANSI C library[[7]](#footnote-7) prove to be the most effective.

* 12345
* 1103515245

## 

## Xorshift

Xorshift is a random number algorithm introduced by George Marsaglia in 2003[[8]](#footnote-8) that belongs to a group of random number algorithms called *Linear feedback shift registers.* It generates random numbers by using multiple bit shift and exclusive-or (xor) operations on the seed. One implementation is as follows.

Where

* the next value in the sequence to be calculated, or the seed
* the first tap
* the second tap
* the third tap
* the left shift operation
* the right shift operation
* the exclusive-or operation (xor)

Fig. 2

input

output input

This starts by setting to the seed, applying the bit shifts and exclusive-or operations and then setting the next element of the sequence to . The further iterations of the algorithm set to the previously calculated value. If the taps are chosen correctly, the period of an xorshift algorithm will equal where is the chosen integer length. One limitation of the xorshift algorithm is that a seed value of 0 will produce a sequence of all zeros. This is due to the fact that the seed is not incremented by a constant value.

When using 32 bit integers, the following taps are used to provide a maximum period of (9.

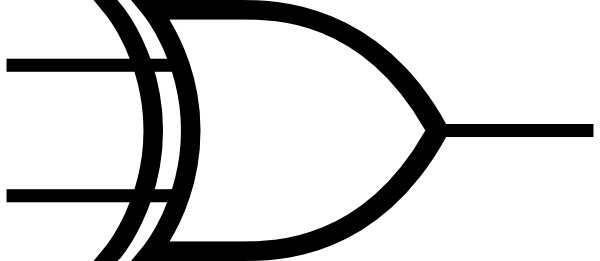
* 13
* 17
* 5

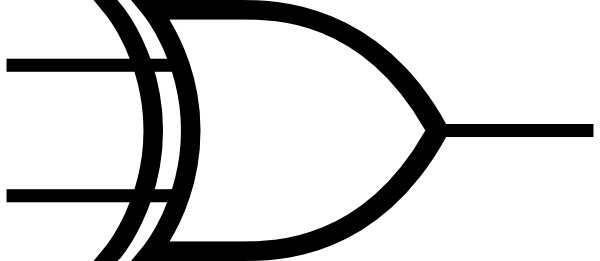
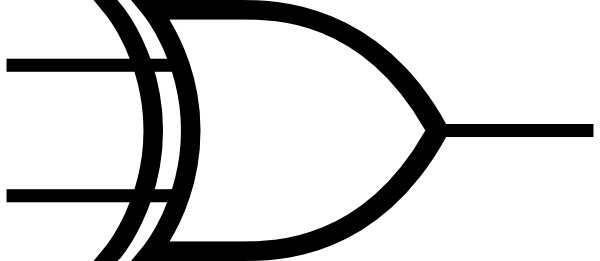
## 

## Fibonacci Linear Feedback Shift Register

This algorithm belongs to the *Linear feedback shift register* family of random number algorithms, due to its linear function. Figure 3 illustrates the structure and linear function of a hardware implementation of the Fibonacci LFSR.

Fig. 3





Where

* the integer’s most significant bit
* the taps

As shown in figure 3, the algorithm works by xor-ing the first two taps, xor-ing the result with the third tap, xor-ing the result of that with the fourth tap, setting the most significant bit to the result and then shifting all bits right by one.

When the correct taps are chosen, the *Fibonacci LFSR* has a period of , but when incorrect taps are chosen, the period depends on the seed[[9]](#footnote-9). In order to determine the correct taps, the polynomial of the sequence where the expansion is equal to (, where the tap, must be considered. If the expansion cannot be factorised but is a factor of ( where , the taps will produce a period of maximum length9. For example, the following taps produce such an expansion.

(

Due to their simplicity and speed, *Fibonacci LFSR*s are commonly implemented in hardware systems where the rapid generation of random numbers is essential, often replacing a software implementation altogether. For example, they are used to generate white noise in programmable sound generators11, where they are created by using a series of D-type flip flops, the quantity of which will depend on the integer word length desired. Hardware implementations share the same period as the software versions since the flip-flops can only cycle through a finite number of steps until they repeat themselves. Figure 4 is an example of an 8 bit hardware implementation that has been modified to set the most significant bit to instead of , after shifting the integer’s bits,

Fig. 4[[10]](#footnote-10)

## 

## 

## Mersenne Twister

The Mersenne twister is a variant of random number algorithms called *Twisted generalized feedback shift register* (TGFSR)[[11]](#footnote-11) proposed by M. Matsumoto and T. Nishimura, with a very long period of (. The fact that the period is a *Mersenne prime*[[12]](#footnote-12) gives the algorithm its name.

It generates a sequence of uniformly pseudorandom integers between 0 and , where the integer word size. Since the period is much larger than the possible range of integers that it can generate, each integer will be produced approximately times () due its uniform distribution.

The algorithm is initialised by populating an array with 624 random integers. The algorithm then loops times and processes the element of the array and repopulates the array once 0. The fact that the array is re-populated with distinct values every 624 iterations is what allows the huge period. Figure 5 describes the algorithm that generates each random number, where is the array of 624 integers.

Fig. 5

if, repopulate

# Statistical Tests

The selected algorithms will be subject to several statistical tests, each one evaluating distinct characteristics and properties of the random number sequences that the algorithms produce.

One test that will be conducted will be the time taken by each algorithm to generate a sequence of random numbers of a fixed length. This will be tested due to the fact that procedural generation usually takes place while the software is running, and the generation of random numbers must be fast enough so that is doesn’t occupy the computer for too long and interrupt the user’s experience. The period will also be considered when deciding upon the most appropriate algorithm.

The tests will be conducted via the *Xcode* IDE on a computer running Mac OS X 10.9.3 with the following specs; 2.4GHz Dual Core i5 Intel Processor, 8GB DDR3 RAM and a 500GB hard drive. Although more or less capable computers could produce different results, the above specifications adhere to those demanded by several games that implement procedural generation during run-time[[13]](#footnote-13).

In order for a test to be selected for use, they must adhere to the same criteria that the random number algorithms adhere to, in order to maximise compatibility and consistency between the test and the algorithms. The following tests were chosen from a variety that falls under the suite of *Diehard tests* first developed by George Marsaglia[[14]](#footnote-14).

* Runs
* Monkey test

## Runs

A run is a streak of either decreasing or increasing values in a sequence of random numbers, a successive increase in values is called a *run up* and a successive decrease in values is called a *run down*. For example, the following sequence (3, 8, 16, 4, 2) consists of one run-up from 3 to 16 and one run-down from 16 to 2. The test itself will measure the longest and average run length as well as the quantity of runs. A good quality random number algorithm will produce a sequence consisting of an almost equal quantity of run-downs and run-ups and will have a smaller likelihood of producing longer runs compared to that of shorter runs[[15]](#footnote-15). For an algorithm to not fail the test, the average and maximum run lengths cannot be equal to each other and must be greater than 1. This test is relevant to procedural generation since successive decreasing or increasing values could produce a visible and obvious pattern in the content generated.

## Monkey test

The monkey test is based on the *Infinite monkey theorem*, where the premise is that a monkey pressing typewriter keys at random for an infinite amount of time, will at some point have typed out the entirety of Shakespeare’s works[[16]](#footnote-16), even if it were just entering random strings of characters. In the case of the monkey test, the monkey is represented by the random number algorithm and the keystrokes are represented by the random numbers produced by said algorithm. The test itself measures the occurrences of certain sequences of numbers or characters (converted from the numbers) and the fewer times the sequence appears (as long as it appears at least once), the better the algorithm is that produced it. This is because a good quality random number algorithm will have a greater variance in the numbers it produces, hence having fewer occurrences of the same character sequences. An example of a popular sequence of characters that is used in monkey tests is “CAT”[[17]](#footnote-17), however, the selected algorithms will be tested for all possible 3 letter ( when using the English alphabet) character sequences in order to give a more representative and encompassing result. The algorithm that has the smallest maximum occurrences of a word sequence, is missing as few word sequences as possible and has an average occurrence as close to 1 as possible will pass the test. The monkey test is relevant to procedural generation, since reoccurring sequences of numbers could produce predictable and obvious patterns among the content generated.

## Method

A seed based on the amount of processor time that has passed since the start of the program will be generated and fed into 500 iterations of each algorithm. The time taken by each iteration will contribute towards an average and after the iterations are complete, the monkey and runs tests will be conducted on the generated sequence. This is then repeated for 5 different seeds.

The seeds are generated using the system time since this is a commonly used method of generating a pseudo-random seed for an algorithm, in addition, the program waits for user-input before generating a new seed in order to create a delay between each. The ideal sequence length of could not be used since this would use 16GB of RAM (twice as much as was present on the computer), therefore an arbitrary length of was chosen, which would only occupy 262KB of RAM.

Figure 6 shows the method.

Generate a sequence 500 times

Repeat for each algorithm. Follow the blue path once every algorithm has been tested

Generate a new seed

Fig. 6

Compute the average time

Repeat 5 times. Exit program afterwards

Conduct the runs test

Conduct the monkey test

Wait for user input

## Results

The mean results calculated from the 5 seeds for each test are presented in figure 7, 8 and 9. Please see the appendix for all data collected.

Fig. 7: The mean values from the *Runs test*.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Algorithms** | **Runs Up** | | | **Runs Down** | | |
| **Quantity** | **Average** | **Maximum** | **Quantity** | **Average** | **Maximum** |
| Fibonacci LFSR | 16391.1 | 1.4 | 19.6 | 16391.1 | 1.6 | 17.2 |
| Xorshift | 16476.4 | 1.8 | 11.2 | 16476.8 | 1.2 | 6 |
| LCG | 32767 | 1 | 1 | 32767 | 1 | 1 |
| Mersenne Twister | 21925 | 1 | 6 | 21924 | 1 | 8 |

Fig. 8: The mean values from the *Monkey test.*

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithms** | **Missing sequences** | **Maximum occurrences** | **Average occurrences** |
| Fibonacci LFSR | 17448 | 208.2 | 1 |
| Xorshift | 17242.8 | 196 | 1 |
| LCG | 17573 | 10922 | 1 |
| Mersenne Twister | 17064 | 65 | 1 |

Fig. 9: The mean values from the time test.

|  |  |
| --- | --- |
| **Algorithm** | **Time (ms)** |
| Fibonacci LFSR | 466.6 |
| Xorshift | 539 |
| LCG | 930.8 |
| Mersenne Twister | 2624.2 |

## Analysis

As can be seen by the results from the *Runs* test, each algorithm produced a near equal quantity of runs down and runs up. However, both the average and maximum run lengths of the *Linear congruential generator* was 1, which means that each successive number in the sequence alternates between being greater than the previous and less than the previous. Ultimately, this would produce a sequence with a clear pattern. The other algorithms fulfill the requirement that the average and maximum lengths should not being equal to 1 or each other, therefore all algorithms but the *Linear congruential generator* pass this test.

The algorithm that had the most missing sequences in the *Monkey test* was the *Linear congruential generator* which only produced 3 of the 17576 possible word sequences, in addition, the algorithm had a maximum occurrence of 10922, which means that it produced one of three word sequences 10922 times. This is a very inadequate result, as it could produce a clear pattern among the content generated. The algorithm that performed the best in the test was the *Mersenne twister*,since it produced the fewest missing word sequences and had the smallest maximum occurrences of any one sequence.

The algorithm that took the least amount of time to generate a sequence of numbers was the *Fibonacci Linear Feedback Shift Register* with an average time of 466.6 milliseconds. The algorithm that performed the worst was the *Mersenne twister* with an average time of 2624.2 milliseconds.

## Conclusion

A random number generator that is to be used in procedural generation must show a balance between speed, randomness and a lengthy period in order to produce adequate quality content that doesn’t consist of clear patterns. These requirements are represented by the statistical tests performed on all four algorithms. Figure 10 was constructed to show each algorithm’s qualities and performance in the tests, with a tick indicating that the algorithm passed the test.

Fig. 10

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Runs test** | **Monkey test** | **Time** | **Period** |
| Fibonacci LFSR | 🗸 |  | 🗸 |  |
| Xorshift | 🗸 |  |  |  |
| LCG |  |  |  |  |
| Mersenne Twister | 🗸 | 🗸 |  | 🗸 |

As can be seen by figure 10, the *Mersenne Twister* algorithm proved to be the most appropriate. Except for during the time test, the *Mersenne Twister* constantly outperformed the other algorithms, which proves it worth as an algorithm for use in procedural generation. Since the *Fibonacci LFSR* excelled where the *Mersenne Twister* failed in the time test, the *Fibonacci LFSR* can be used as an alternative for applications when time is of the essence.

The factors that contributed to the *Mersenne Twister*’s success as the most appropriate algorithm was its long period , its ability to produce the most varied sub-sequences of numbers during the *Monkey test* (17064 sequences) and that during the *Runs test*, it produced runs of excellent length (mean between 1 and 6), of which it produced the most (mean 21925) out of all the algorithms that passed said test. The implications of these factors are that it has the potential capability of generating a sequence suitable for procedural generation within a video game and as a result it has a large user-base, ensuring that help with the algorithm is not far away. It’s suitability for procedural generation is reinforced by the requirements of each algorithm selected; compatibility with C++, usage of 32-bit integers and independency of unusual external equipment, all of which the *Mersenne Twister* passes, ensuring that it will function as intended on the vast majority of modern computers represented by the computer on which the tests took place. In conclusion, the *Mersenne Twister* poses as a suitable and appropriate random number algorithm for use within procedural content generation in video games.

# Bibliography

## Books

**Handbook of Computational Statistics**

L’Ecuyer, P. Montréal: Springer-Verlag, 2004

**The Art Of Computer Programming**

Knuth, Donald E. Reading, Mass.: Addison-Wesley, 1968

**Procedural Content Generation In Games: A Textbook and an Overview of Current Research**

Shaker, Noor and Togelius, Julian and Nelson, Mark J. Springer, 2014

## Journals

**Monkey Tests for Random Number Generators.**

Marsaglia, G. And Zaman, A. *Computers and Mathematics with Applications*, 1993, 26(9), 1-10

## Web pages

**Characteristics of random number generators**

Johnson, W. <http://anomaly.org/ThinAir/charactr.html>, 1997 [27/05/14]

**Digital noise generator**

[http://saroselectronics.com/blog-3/page/4/](http://saroselectronics.com/blog-3/page/4), 28/02/13 [20/05/14]

**How Computers Generate Random Numbers**

Deley, D. <http://daviddeley.com/random/random4.htm> [03/06/14]

**Infinite-monkey theorem**

<http://www.catb.org/jargon/html/I/Infinite-Monkey-Theorem.html> [27/05/14]

**Linear congruential generators**

<https://www.princeton.edu/~achaney/tmve/wiki100k/docs/Linear_congruential_generator.html> [10/05/14]

**Linear Feedback Shift Registers**

[http://www.newwaveinstruments.com/resources/articles/m\_sequence\_linear\_feedback\_shift\_register\_lfsr.htm](http://www.newwaveinstruments.com/resources/articles/m_sequen) [15/05/14]

**Minecraft system requirements**

[https://help.mojang.com/customer/portal/articles/325948-minecraft-system-requirements](https://help.mojang.com/customer/portal/articles/325948-minecraft-), 20/02/14 [30/05/14]

**Procedural Content Generation Wiki**

<http://pcg.wikidot.com>, 18/10/13. [06/05/14]

**Random (Java Platform SE 7)**

<http://docs.oracle.com/javase/7/docs/api/java/util/Random.html#next(int>) [03/06/14]

**Random Number Generators**

Johnson, D. and Capela, M. [http://courses.physics.illinois.edu/phys466/sp2013/lnotes/random\_numbers.html](http://courses.physics.il) [09/05/14]

**Space Engine**

Available at: <http://en.spaceengine.org>[30/05/14]

**Statistical testing of random number generators**

Soto, J. <http://csrc.nist.gov/groups/ST/toolkit/rng/documents/nissc-paper.pdf> [23/05/14]

**The Mersenne Twister**

<http://www.quadibloc.com/crypto/co4814.htm> [20/05/14]

**Xorshift RNGs**

Marsaglia, G. <http://www.jstatsoft.org/v08/i14/paper> [11/05/14]

# 

# Appendix

All material in this appendix can be downloaded from: <https://github.com/VivaDaylight3/ExtendedEssay>

## A: Raw statistical data

**Runs test**

Seed: 3014

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Algorithms** | **Runs Up** | | | **Runs Down** | | |
| **Quantity** | **Average** | **Maximum** | **Quantity** | **Average** | **Maximum** |
| Fibonacci LFSR | 16350 | 2 | 25 | 16351 | 2 | 18 |
| Xorshift | 16464 | 2 | 14 | 16465 | 1 | 6 |
| LCG | 32767 | 1 | 1 | 32766 | 1 | 2 |
| Mersenne Twister | 21925 | 1 | 6 | 21924 | 1 | 8 |

Seed: 2295388

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Algorithms** | **Runs Up** | | | **Runs Down** | | |
| **Quantity** | **Average** | **Maximum** | **Quantity** | **Average** | **Maximum** |
| Fibonacci LFSR | 16447 | 1 | 18 | 16446 | 1 | 17 |
| Xorshift | 15985 | 2 | 12 | 15986 | 2 | 6 |
| LCG | 32767 | 1 | 1 | 32766 | 1 | 2 |
| Mersenne Twister | 21925 | 1 | 6 | 21924 | 1 | 8 |

Seed: 4579353

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Algorithms** | **Runs Up** | | | **Runs Down** | | |
| **Quantity** | **Average** | **Maximum** | **Quantity** | **Average** | **Maximum** |
| Fibonacci LFSR | 16390 | 1 | 17 | 16390 | 2 | 15 |
| Xorshift | 16760 | 2 | 9 | 16760 | 1 | 6 |
| LCG | 32767 | 1 | 1 | 32767 | 1 | 1 |
| Mersenne Twister | 21925 | 1 | 6 | 21924 | 1 | 8 |

Seed: 6879503

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Algorithms** | **Runs Up** | | | **Runs Down** | | |
| **Quantity** | **Average** | **Maximum** | **Quantity** | **Average** | **Maximum** |
| Fibonacci LFSR | 16354 | 2 | 19 | 16353 | 2 | 18 |
| Xorshift | 16458 | 1 | 9 | 16458 | 1 | 6 |
| LCG | 32767 | 1 | 1 | 32767 | 1 | 1 |
| Mersenne Twister | 21925 | 1 | 6 | 21924 | 1 | 8 |

Seed: 9150701

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Algorithms** | **Runs Up** | | | **Runs Down** | | |
| **Quantity** | **Average** | **Maximum** | **Quantity** | **Average** | **Maximum** |
| Fibonacci LFSR | 16415 | 1 | 19 | 16416 | 1 | 18 |
| Xorshift | 16715 | 2 | 12 | 16715 | 1 | 6 |
| LCG | 32767 | 1 | 1 | 32767 | 1 | 1 |
| Mersenne Twister | 21925 | 1 | 6 | 21924 | 1 | 8 |

**Monkey test**

Seed: 3014

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithms** | **Missing sequences** | **Maximum occurrences** | **Average occurrences** |
| Fibonacci LFSR | 17448 | 198 | 1 |
| Xorshift | 17239 | 211 | 1 |
| LCG | 17574 | 10923 | 1 |
| Mersenne Twister | 17064 | 65 | 1 |

Seed: 2295388

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithms** | **Missing sequences** | **Maximum occurrences** | **Average occurrences** |
| Fibonacci LFSR | 17448 | 215 | 1 |
| Xorshift | 17252 | 213 | 1 |
| LCG | 17573 | 10922 | 1 |
| Mersenne Twister | 17064 | 65 | 1 |

Seed: 4579353

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithms** | **Missing sequences** | **Maximum occurrences** | **Average occurrences** |
| Fibonacci LFSR | 17448 | 221 | 1 |
| Xorshift | 17248 | 172 | 1 |
| LCG | 17573 | 10922 | 1 |
| Mersenne Twister | 17064 | 65 | 1 |

Seed: 6879503

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithms** | **Missing sequences** | **Maximum occurrences** | **Average occurrences** |
| Fibonacci LFSR | 17488 | 203 | 1 |
| Xorshift | 17243 | 170 | 1 |
| LCG | 17573 | 10922 | 1 |
| Mersenne Twister | 17064 | 65 | 1 |

Seed: 9150701

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithms** | **Missing sequences** | **Maximum occurrences** | **Average occurrences** |
| Fibonacci LFSR | 17448 | 204 | 1 |
| Xorshift | 17232 | 214 | 1 |
| LCG | 17573 | 10922 | 1 |
| Mersenne Twister | 17064 | 65 | 1 |

**Time test**

Seed: 3014

|  |  |
| --- | --- |
| **Algorithm** | **Time (ms)** |
| Fibonacci LFSR | 478 |
| Xorshift | 531 |
| LCG | 924 |
| Mersenne Twister | 2633 |

Seed: 2295388

|  |  |
| --- | --- |
| **Algorithm** | **Time (ms)** |
| Fibonacci LFSR | 472 |
| Xorshift | 540 |
| LCG | 927 |
| Mersenne Twister | 2609 |

Seed: 4579353

|  |  |
| --- | --- |
| **Algorithm** | **Time (ms)** |
| Fibonacci LFSR | 447 |
| Xorshift | 540 |
| LCG | 911 |
| Mersenne Twister | 2683 |

Seed:6879503

|  |  |
| --- | --- |
| **Algorithm** | **Time (ms)** |
| Fibonacci LFSR | 476 |
| Xorshift | 520 |
| LCG | 950 |
| Mersenne Twister | 2577 |

Seed: 9150701

|  |  |
| --- | --- |
| **Algorithm** | **Time (ms)** |
| Fibonacci LFSR | 460 |
| Xorshift | 564 |
| LCG | 942 |
| Mersenne Twister | 2619 |

## 

## c

## B: Statistical test program

include <iostream>

#include <cmath> // pow() function

#include <time.h> // clock() function

const uint32\_t arrayLength = 65535; // Arbitrary (although feasible memory usage) number

uint32\_t sequence[arrayLength]; // The sequence in which the numbers will be placed

int indexMT = 0, MT[623]; // Used with the Mersenne Twister algorithm

using namespace std;

// Fibonnaci Linear Feedback Shift Register

void fibonnaciLFSR(uint32\_t seed){

for(int c = 0; c < arrayLength; c++){

int xOr = (seed ^ (seed >> 28)) ^ (seed >> 30) ^ (seed >> 31) & 1;

seed = (seed >> 1) | (xOr << 31);

sequence[c] = seed;

}

}

// Xorshift

void xorShift(uint32\_t seed){

for(int c = 0; c < arrayLength; c++){

seed ^= seed << 13;

seed = seed >> 17;

seed ^= seed << 5;

sequence[c] = seed;

}

}

// Linear Congruential Generator

void linearCongruentialGenerator(int seed){

int m = UINT32\_MAX - 1; // Maximum value of a 32-bit integer - 1, defines longest period possible

int a = 1103515245;

int c = 12345;

sequence[0] = seed;

for(int k = 1; k < arrayLength; k++){

sequence[k] = (uint16\_t)(sequence[k-1] \* a + c) % m;

}

}

// (Mersenne Twister) Generates a new set of 624 integers

void genMT(){

for (int i = 0; i < 624; i++) {

int y = (MT[i] & 0x80000000) + (MT[(i+1) % 624] & 0x7fffffff);

MT[i] = MT[(i + 397) % 624] xor (y >> 1);

if ((y % 2) != 0) {

MT[i] = MT[i] xor (2567483615);

}

}

}

// (Mersenne Twister) Main number generation function, dependent on genMT()

uint32\_t randMT(){

if (indexMT == 0) {

genMT();

}

int y = MT[indexMT];

y = y xor (y >> 11);

y = y xor ((y << 7) & 2636928640);

y = y xor ((y << 15) & 4022730752);

y = y xor (y >> 18);

indexMT = (indexMT + 1) % 624;

return y;

}

// (Mersenne Twister) Initialises first array of 624 integers

void initMT2(uint32\_t seed){

indexMT = 0;

MT[0] = seed;

for (int i = 0; i < 624; i++){

MT[i] = (1812433253 \* (MT[i-1] xor (MT[i-1] >> 30)) + i) & 0xffff;

}

}

// Mersenne Twister

void mersenneTwister(uint32\_t seed){

initMT2(seed);

for(int c = 0; c < arrayLength; c++){

sequence[c] = randMT();

}

}

// Runs test function

void testRuns(){

int runsUpQuantity = 0, runsDownQuantity = 0, runsUpLongest = 0, runsDownLongest = 0, runsUpCurrent = 1, runsDownCurrent = 1;

double runsUpAverage = 0, runsDownAverage = 0;

bool runUp = false, runDown = false;

int runsDown[arrayLength];

int runsUp[arrayLength];

for(int c = 1; c < arrayLength; c++){

if(sequence[c] > sequence[c-1]){

if(runUp){

runsUpCurrent++;

}else{

runsDown[runsDownQuantity] = runsDownCurrent;

runsDownQuantity++;

runsDownCurrent = 1;

runsUpCurrent = 1;

runDown = false;

runUp = true;

}

}else if(sequence[c] < sequence[c-1]){

if(runDown){

runsDownCurrent++;

}else{

runsUp[runsUpQuantity] = runsUpCurrent;

runsUpQuantity++;

runsUpCurrent = 1;

runsDownCurrent = 1;

runUp = false;

runDown = true;

}

}

}

int runsUpSum = 0, runsDownSum = 0;

for(int c = 0; c < max(runsUpQuantity, runsDownQuantity); c++){

if(c <= runsUpQuantity){

runsUpLongest = max(runsUpLongest, runsUp[c]);

runsUpSum += runsUp[c];

}

if(c <= runsDownQuantity){

runsDownLongest = max(runsDownLongest, runsDown[c]);

runsDownSum += runsDown[c];

}

}

runsUpAverage = runsUpSum / runsUpQuantity;

runsDownAverage = runsDownSum / runsDownQuantity;

cout << "Runs Up" << endl;

cout << " - Run quantity: " + to\_string(runsUpQuantity) << endl;

cout << " - Average length: " + to\_string(runsUpAverage) << endl;

cout << " - Maximum length: " + to\_string(runsUpLongest) << endl;

cout << "Runs Down" << endl;

cout << " - Run quantity: " + to\_string(runsDownQuantity) << endl;

cout << " - Average length: " + to\_string(runsDownAverage) << endl;

cout << " - Maximum length: " + to\_string(runsDownLongest) << endl;

}

// Monkey test function

void testMonkey(){

int words[26][26][26];

// Initialises words array, invalid results occur without this.

for(int a = 0; a < 26; a++){

for(int b = 0; b < 26; b++){

for(int c = 0; c < 26; c++){

words[a][b][c] = 0;

}

}

}

for(int c = 2; c < arrayLength; c+=3){

words[sequence[c-2] & 25][sequence[c-1] & 25][sequence[c] & 25]++;

}

int missingC = 0, total = 0, max = 0;

for(int a = 0; a < 26; a++){

for(int b = 0; b < 26; b++){

for(int c = 0; c < 26; c++){

if(words[a][b][c] == 0){

missingC++;

}else if(words[a][b][c] > max){

max = words[a][b][c];

}

total += words[a][b][c];

}

}

}

double average = total / 17576;

cout << "Monkey test" << endl;

cout << " - Missing sequences: " << missingC << endl;

cout << " - Maximum occurence: " << max << endl;

cout << " - Average occurences: " << average << endl;

}

// Main function

int main(int argc, const char \* argv[])

{

int rounds = 500;

for(int k = 0; k < 5; k++){

int seed = clock();

cout << "Seed: " << seed << endl;

cout << "Fibonacci LFSR" << endl;

int totalTime = 0;

for(int c = 0; c < rounds; c++){

clock\_t t = clock();

fibonnaciLFSR(seed);

totalTime += (clock() - t);

}

cout << "Average time: " + to\_string(totalTime / rounds) + " ms" << endl;

testRuns();

testMonkey();

cout << endl;

cout << "Xorshift" << endl;

totalTime = 0;

for(int c = 0; c < rounds; c++){

clock\_t t = clock();

xorShift(seed);

totalTime += (clock() - t);

}

cout << "Average time: " + to\_string(totalTime / rounds) + " ms" << endl;

testRuns();

testMonkey();

cout << endl;

cout << "Linear Congruential Generator" << endl;

totalTime = 0;

for(int c = 0; c < rounds; c++){

clock\_t t = clock();

linearCongruentialGenerator(seed);

totalTime += (clock() - t);

}

cout << "Average time: " + to\_string(totalTime / rounds) + " ms" << endl;

testRuns();

testMonkey();

cout << endl;

cout << "Mersenne Twister" << endl;

totalTime = 0;

for(int c = 0; c < rounds; c++){

clock\_t t = clock();

mersenneTwister(seed);

totalTime += (clock() - t);

}

cout << "Average time: " + to\_string(totalTime / rounds) + " ms" << endl;

testRuns();

testMonkey();

cout << endl << "### END OF SEED ###" << endl << endl;

string str;

cin >> str; // Waits for user input in order to form a delay between each seed

}

cout << endl << "### END OF PROGRAM ###" << endl;

return 0;

}

1. Definition from *Procedural Content Generation Wiki*. Available at: <http://pcg.wikidot.com> 18/10/13. [06/05/14] [↑](#footnote-ref-1)
2. Shaker, Noor and Togelius, Julian and Nelson, Mark J. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2014. Chapter 1, page 3. [↑](#footnote-ref-2)
3. Johnson D.D. and Caperly M., *Random Number Generators*. Available at: <http://courses.physics.illinois.edu/phys466/sp2013/lnotes/random_numbers.html> [09/05/14] [↑](#footnote-ref-3)
4. Knuth, Donald E. *The Art Of Computer Programming*. Reading, Mass.: Addison-Wesley, 1968. Volume 2, page 10 [↑](#footnote-ref-4)
5. *Linear congruential generators*. Available at: <https://www.princeton.edu/~achaney/tmve/wiki100k/docs/Linear_congruential_generator.html> [10/05/14] [↑](#footnote-ref-5)
6. Knuth, Donald E. *The Art Of Computer Programming*. Reading, Mass.: Addison-Wesley, 1968. Volume 2, page 17 [↑](#footnote-ref-6)
7. Deley, D. *How Computers Generate Random Numbers.* Available at: <http://daviddeley.com/random/random4.htm> [03/06/14] [↑](#footnote-ref-7)
8. Marsaglia, G. *Xorshift RNGs*. Available at: <http://www.jstatsoft.org/v08/i14/paper> [11/05/14] [↑](#footnote-ref-8)
9. *Linear Feedback Shift Registers*. Available at: <http://www.newwaveinstruments.com/resources/articles/m_sequence_linear_feedback_shift_register_lfsr.htm> [15/05/14] [↑](#footnote-ref-9)
10. *Digital noise generator*. Available at: <http://saroselectronics.com/blog-3/page/4/> [20/05/14] [↑](#footnote-ref-10)
11. L’Ecuyer, P. *Handbook of Computational Statistics*. Montréal: Springer-Verlag, 2004 [↑](#footnote-ref-11)
12. *The Mersenne Twister*. Available at: <http://www.quadibloc.com/crypto/co4814.htm> [20/05/14] [↑](#footnote-ref-12)
13. *Space Engine.* Available at: <http://en.spaceengine.org>[30/05/14], *Minecraft system requirements*. Available at: <https://help.mojang.com/customer/portal/articles/325948-minecraft-system-requirements> [30/05/14] [↑](#footnote-ref-13)
14. Soto, J. *Statistical testing of random number generators.* Available at: <http://csrc.nist.gov/groups/ST/toolkit/rng/documents/nissc-paper.pdf> [23/05/14] [↑](#footnote-ref-14)
15. Johnson, W. *Characteristics of random number generators.* Available at: <http://anomaly.org/ThinAir/charactr.html> [27/05/14] [↑](#footnote-ref-15)
16. *Infinite-monkey theorem*. Available at: <http://www.catb.org/jargon/html/I/Infinite-Monkey-Theorem.html> [27/05/14] [↑](#footnote-ref-16)
17. Marsaglia, G. And Zaman, A. Monkey Tests for Random Number Generators. *Computers and Mathematics with Applications,* 1993, 26(9), 1-10 [↑](#footnote-ref-17)