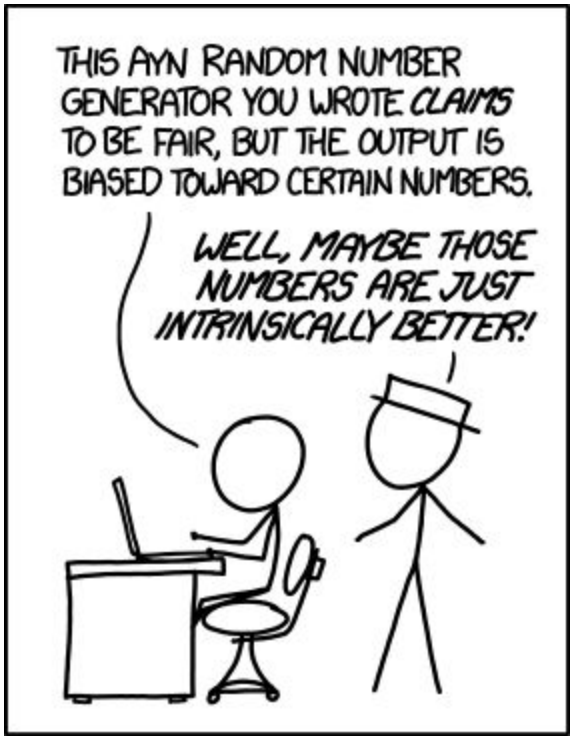
Comparing Pseudo-Random Number Generators

Michael Dubé
5243845
COSC 4P03
April 23rd 2018

Comedy First:




```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

Okay, let's get down to business!

Tables of Contents

Introduction

Pseudo-Random Number Generators have been an active component of research since the dawn of modern computing. With applications in cryptography and simulation the need for truly random numbers has been large and will only grow as the processing power of computers continues to expand. The focus of this paper is on implementing 4 random number generators, as well as Java's implementation in the Random class, and testing these generators using two different testing suites which were also implemented by me. The generators chosen were not chosen for any particular reason nor were there particular properties that were desired by the generators I chose to implement. An important note is that the testing suites used within this paper do not constitute a definitive analysis of the generators presented though does provide an introduction to testing the randomness of different random number generators.

I will begin by introducing what a random number generator is followed by explaining the different generators explored within this paper. Example outputs and analysis of a generators will be included within its respective section. I will then outline different ways to test generators and go over the testing suites I used to compare the generators in this paper. The results for a test will be included within the its respective section.

Random Number Generators

A random number generator is characterized by its ability to generate a seemingly random stream of numbers. Different random number generators output different data. The default output of a random number generator can be one of the following number types: boolean, integer, or floating point. From there, the other number types can be created through converting the output of a generator to another type. For example, given a generator which outputs integers in the range [0,1000000) one can divide the output by 1000000 to get an output of floating points in the range [0, 1). This presents a possible issue for the implementation of random number generators as faulty logic or implementation of the conversion from one number type to another can impact the randomness of the generator for certain number types.

Random number generators can also have different probability distributions such as uniform and nonuniform [1]. A uniform random number generator has equal probability of outputting any value from its range (ie. a single die roll). Whereas, a nonuniform generator has different probabilities for different range values. For example, given a generator with range [2,12] the centre value, 7, may be most likely than the edge values, 2 and 12, are the least likely (ie. the sum of two dice). This paper will focus on uniform random number generators as it is

often the case that nonuniform generators can be created by utilizing an existing uniform generator as shown above through the use of dice.

Java Generator

The Java generator is simply a wrapping of the existing random number generator provided within Java Development Kit 8. The documentation indicates that Java uses a linear congruential generator [2] which will be further described within that section. This generator was included as a benchmark to compare against other implementations of random number generators. More information on the Java Generator can be found within Table 1.

Table 1: Java Generator Characteristics

Generator Characteristic	Characteristic Value	Explanation/Comment
Default Output	Unknown.	As this was not personally implemented no information is present.
Conversion from Default → Boolean	Unknown.	As this was not personally implemented no information is present.
Conversion from Default → Integer	Unknown.	As this was not personally implemented no information is present.
Boolean Output of length 100	0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1	No abnormalities in output stand out for this generator.
Integer Output in Range [0, 100) of length 100	87, 92, 74, 24, 6, 5, 54, 91, 22, 21, 31, 3, 47, 60, 28, 90, 78, 15, 17, 17, 8, 65, 8, 56, 61, 2, 80, 47, 15, 20, 89, 92, 57, 29, 7, 30, 46, 43, 69, 33, 84, 51, 70, 57, 41, 98, 52, 42, 53, 95, 25, 5, 12, 37, 3, 12, 59, 0, 47, 31, 99, 51, 7, 26, 98, 65, 33, 97, 44, 18, 49, 36, 58, 30, 55, 57, 69, 60, 13, 99, 65, 73, 84, 63, 56, 91, 88, 6, 91, 37, 43, 11, 0, 1, 0, 85, 27, 69, 54, 60	No abnormalities in output stand out for this generator.

Blum Blum Shub Generator

The Blum Blum Shub generator was first introduced in 1986 and is explained in detail within [3]. A unique attribute of the generator is that it does not output the integer generated within the algorithm and instead outputs a boolean generated by looking at the least significant bit of the generated number. See Algorithm 1-3 and Table 2 for more information on the Blum Blum Shub generator.

Algorithm 1: Blum Blum Shub Initializer

Description	Creates a Blum Blum Shub random number generator
Input	• p - large prime number such that $p \equiv 3 \ (mod\ 4)$ • q - large prime number such that $q \equiv 3 \ (mod\ 4)$ and $\lvert p \rvert = \lvert q \rvert$ where $\lvert p \rvert = \ length\ of\ p\ in\ bits$ • seed - initial integer value of the generator such that $gcd(p * q,\ seed) = 1$ and $seed > 1$
Output	Blum Blum Shub generator
Method	1. Assign parameters to generator's variables

Algorithm 2: Blum Blum Shub Next Boolean

Description	Returns the next boolean from the generator
Input	None
Output	Boolean value
Knowns	• x - previous value generated by the algorithm • M - product of primes p, q used to initialize the generator
Method	1. $x = x^2 \ mod\ M$ 2. Return $x \ mod\ 2 == 0$

Algorithm 3: Blum Blum Shub Next Integer

Description	Returns the next integer from the generator
Input	• min - minimum value returned • lessThan - upper bound (exclusive) of value returned
Output	Integer value in range [min, lessThan)
Knowns	• x - previous value generated by the algorithm • M - product of primes p, q used to initialize the generator

Method	1. $diff = lessThan - min - 1$ 2. $numBits = number\ of\ bits\ in\ value\ diff$ 3. $boolList = list\ of\ booleans\ of\ length\ numBits\ from\ Algorithm\ 2$ 4. $num = convert\ boolList\ to\ integer$ 5. If $num > diff$ go to 3 6. Return $min + num$

Table 2: Blum Blum Shub Generator Characteristics

Generator Characteristic	Characteristic Value	Explanation/Comment
Default Output	Least Significant Bit of generated integer.	Instead of outputting the value calculated, this generator returns the least significant bit of the value calculated.
Conversion from Default → Boolean	Not applicable.	No conversion necessary.
Conversion from Default → Integer	Use a list of bits outputted by the generator to convert to an integer within the desired range.	See Algorithm 3.
Boolean Output of length 100	0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0	No abnormalities in output stand out for this generator.
Integer Output in Range [0, 100) of length 100	95, 97, 60, 33, 11, 44, 65, 24, 28, 6, 70, 60, 23, 96, 3, 45, 48, 66, 76, 69, 59, 94, 13, 58, 73, 79, 9, 56, 87, 31, 32, 98, 42, 71, 18, 79, 26, 52, 17, 71, 33, 96, 83, 42, 53, 50, 57, 68, 9, 93, 5, 84, 9, 20, 58, 20, 77, 21, 20, 69, 34, 66, 62, 47, 14, 69, 82, 4, 15, 79, 47, 98, 38, 48, 37, 89, 87, 51, 75, 76, 29, 36, 65, 16, 73, 83, 47, 42, 34, 78, 44, 32, 71, 71, 2, 74, 58, 74, 89, 75	No abnormalities in output stand out for this generator.

Linear Congruential Generator

The Linear Congruential Generator was first introduced in 1962 and is explained in detail within [4]. This generator will eventually repeat and the length of its unique steam of output is known as the generator's period. There are various conditions on the generator's parameters which impact the length of the period. My implementation will use the conditions indicated within [4] which guarantees the maximal period for any given generator. Another important fact is that numerous variations and parameter restrictions can alter the randomness of a Linear Congruential Generator and therefore, the results within this paper do not define the effectiveness of all Linear Congruential Generators. See Algorithm 4-6 and Table 3 below for more information on the Linear Congruential Generator.

Algorithm 4: Linear Congruential Generator Initializer

Description	Creates a Linear Congruential random number generator with maximal period
Input	• a - integer such that $a > 0$ and $a \equiv 1 \ (mod\ 4)$ • b - odd integer such that $b \geq 0$ and • seed - initial integer value of generator such that $seed \geq 0$ • m - power of 2 such that $m > seed$, $m > a$, $m > b$, and $m > 0$
Output	Linear Congruential Generator with maximal period m
Method	1. Assign parameters to generator's variables

Algorithm 5: Linear Congruential Generator Next Integer

Description	Returns the next integer from the generator
Input	• min - minimum value returned • lessThan - upper bound (exclusive) of value returned
Output	Integer value in range [min, lessThan)
Knowns	• x - previous value generated by the algorithm • a - generator variable from initialization • b - generator variable from initialization • m - generator variable from initialization
Method	1. $diff = lessThan - min$ 2. $quotient = m/diff$ 3. $x = (a * x + b)\ mod\ m$ 4. If $x \geq quotient * diff$ go to 3 5. Return $min + (x\ mod\ diff)$

Algorithm 6: Linear Congruential Generator Next Boolean

Description	Returns the next boolean from the generator
Input	None
Output	Boolean value
Knowns	• x - previous value generated by the algorithm • a - generator variable from initialization • b - generator variable from initialization • m - generator variable from initialization
Method	1. $x = Algorithm\ 5\ (0,\ m)$ // Any number in range 2. If $m\ mod\ 2 == 0$ return $x\ mod\ 2 == 1$ 3. If $x \neq m - 1$ return $x\ mod\ 2 == 1$ 4. Go to 1 // Only happens to ensure true and false equally likely

Table 3: Linear Congruential Generator Characteristics

Generator Characteristic	Characteristic Value	Explanation/Comment
Default Output	Integer in range [0, m).	The generator outputs the next value calculated.
Conversion from Default → Boolean	Returns whether or not the value calculated is odd.	See Algorithm 6.
Conversion from Default → Integer	Not applicable.	No conversion necessary.
Boolean Output of length 100	1, 0, 1, 0	The output goes back and forth between 1 and 0 without compromise. This is to be expected as a works out to be odd and b is defined to be odd and therefore, the value will switch back and forth between even and odd.
Integer Output in Range [0, 100) of length 100	57, 10, 67, 96, 49, 82, 27, 36, 45, 78, 11, 20, 37, 94, 55, 36, 1, 38, 43, 72, 21, 90, 51, 20, 25, 70, 87, 28, 1, 70, 91, 8, 69, 10, 3, 84, 21, 62, 7, 32, 1, 34, 47, 92, 41, 34, 63, 32, 65, 98, 91, 88, 93, 14, 63, 80, 5, 62, 63, 96, 65, 90, 59, 44, 37,	No abnormalities in output stand out for this generator.

	66, 55, 88, 45, 18, 23, 48, 61, 18, 79, 80, 17, 10, 23, 96, 69, 86, 35, 28, 69, 2, 99, 44, 73, 66, 55, 60, 25, 30, 79, 16, 49, 14, 59, 12	

Inversive Congruential Generator

The Inversive Congruential Generator was first introduced in 1986 and is explained in detail in [5]. As with the Linear Congruential Generator this generator also has a stream of random numbers which repeats after a certain number of outputs. The length of the unique stream is known as the period length of the generator. There are numerous combinations of parameters which result in an Inversive Congruential Generator of maximal period, though there is no simple way to guarantee that a period will be maximal. Therefore, the constructor used within my implementation continually generates new generators until one is generated with a maximal period. With this being said there are certain limitations on the parameters which are more likely to create a generator of maximal period, as shown in [6], and those will be used within the program and the algorithms below. See Algorithm 7-9 and Table 4 for more information on the Inversive Congruential Generator.

Algorithm 7: Inversive Congruential Generator Initializer

Description	Creates a Inversive Congruential random number generator of maximal period
Input	• a - positive integer such that $a > 0$ • b - positive integer such that $b > 0$ • p - prime number such that $p \geq 5$ • seed - initial integer value of generator such that $0 \leq seed < p$
Output	Inversive Congruential Generator with maximal period p
Method	1. Assign parameters to generator's variables

Algorithm 8: Inversive Congruential Generator Next Integer

Description	Returns the next integer from the generator
Input	• min - minimum value returned • lessThan - upper bound (exclusive) of value returned
Output	Integer value in range [min, lessThan)
Knowns	• x - previous value generated by the algorithm • a - generator variable from initialization • b - generator variable from initialization

	• p - generator variable from initialization
Method	1. $diff = lessThan - min$ 2. $quotient = p/diff$ 3. If $x == 0$ then $x = b$ and go to 6 4. $x^{-1} = modular\ multiplicative\ inverse\ of\ x$ // $x * x^{-1} \equiv 1\ mod\ p$ 5. $x = (a * x^{-1} + b)\ mod\ p$ 6. If $x \geq quotient * diff$ go to 3 7. Return $min + (x\ mod\ diff)$

Algorithm 9: Inversive Congruential Generator Next Boolean

Description	Returns the next boolean from the generator
Input	None
Output	Boolean value
Knowns	• x - previous value generated by the algorithm • a - generator variable from initialization • b - generator variable from initialization • p - generator variable from initialization
Method	1. $x = Algorithm\ 8\ (0,\ p)$ // Any number in range 2. If $x == p - 1$ go to 1 // Ensure equal likelihood of true or false 3. Return $x\ mod\ 2 == 1$

Table 4: Inversive Congruential Generator Characteristics

Generator Characteristic	Characteristic Value	Explanation/Comment
Default Output	Integer in range [0, p).	The generator outputs the next value calculated.
Conversion from Default → Boolean	Not applicable.	No conversion necessary.
Conversion from Default → Integer	Returns whether or not the value calculated is odd.	See Algorithm 9.
Boolean Output of length 100	1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1	The sequence of booleans has several lengthy continuous runs of the same value. This is important to note should this generator be used in certain applications.

Integer Output in Range [0, 100) of length 100	22, 42, 65, 34, 54, 21, 18, 41, 25, 77, 65, 87, 31, 51, 19, 37, 80, 71, 44, 60, 96, 54, 10, 4, 65, 10, 89, 40, 37, 31, 38, 96, 67, 54, 60, 29, 5, 4, 66, 30, 56, 70, 52, 36, 98, 28, 71, 44, 90, 52, 74, 16, 55, 66, 83, 37, 8, 28, 7, 28, 74, 60, 8, 9, 23, 49, 69, 24, 34, 70, 62, 55, 49, 38, 18, 53, 37, 74, 61, 72, 99, 8, 4, 6, 1, 84, 84, 68, 58, 56, 91, 92, 84, 25, 7, 88, 37, 52, 52, 28	The sequence of integers has several lengthy continuous runs of even values and odd values. This is important to note should this generator be used in certain applications.

Compound Inversive Generator

The Compound Congruential Generator was introduced in 1986 and is explained in detail in [6]. This generator is a combination of several Inversive Congruential Generators and due to this fact the stream of values generated will have a period equal to the product of the periods of the Inversive Congruential Generators used to create this generator. Only Inversive Congruential Generators of maximal period are used and therefore the generated Compound Inversive Generator will also have a maximal period. See Algorithm 10-12 and Table 5 for more information on the Compound Inversive Generator.

Algorithm 10: Compound Inversive Generator Initializer

Description	Creates a Compound Inversive random number generator of maximal period
Input	• n - positive integer such that $n > 1$ • $a_1, a_2, \ldots, a_n$ - positive integer such that $a_i > 0$ for all $1 \le i \le n$ • $b_1, b_2, \ldots, b_n$ - positive integer such that $b_i > 0$ for all $1 \le i \le n$ • $p_1, p_2, \ldots, p_n$ - prime number such that $p_i \ge 5$ for all $1 \le i \le n$
Output	Compound Inversive Generator with maximal period T (from Method)
Method	1. $generators = list\ of\ size\ n\ with\ generators\ from\ Algorithm\ 7\ (a_i, b_i, p_i)$ 2. $T = p_1 * p_2 * \ldots * p_n$

Algorithm 11: Compound Inversive Generator Next Integer

Description	Returns the next integer from the generator
Input	• min - minimum value returned • lessThan - upper bound (exclusive) of value returned

Output	Integer value in range [min, lessThan)
Knowns	• n - number of Inversive Congruential Generators • $g_1, g_2, \ldots, g_n$ - Inversive Congruential Generators • T - product of primes used to make $g_1, g_2, \ldots, g_n$
Method	1. $diff = lessThan - min$ 2. $quotient = p / diff$ 3. For $i = 1$ *to* n a. $v_i = g_i$*'s next integer with Algorithm* 8 $(0, p)$ // p from g_i 4. $sum = v_1 + v_2 + \ldots + v_n$ 5. If $sum \geq quotient * diff$ go to 3 6. Return $min + (sum \bmod diff)$

Algorithm 12: Compound Inversive Generator Next Boolean

Description	Returns the next boolean from the generator
Input	None
Output	Boolean value
Knowns	• n - number of Inversive Congruential Generators • $g_1, g_2, \ldots, g_n$ - Inversive Congruential Generators • T - product of primes used to make $g_1, g_2, \ldots, g_n$
Method	1. $x = Algorithm$ 11 $(0, T)$ // Any number in range 2. If $x == T - 1$ go to 1 // Ensure equal likelihood of true or false 3. Return $x \bmod 2 == 1$

Table 5: Compound Inversive Generator Characteristics

Generator Characteristic	Characteristic Value	Explanation/Comment
Default Output	Integer in range [0, T).	The generator outputs the next value calculated.
Conversion from Default → Boolean	Not applicable.	No conversion necessary.
Conversion from Default → Integer	Returns whether or not the value calculated is odd.	See Algorithm 12.
Boolean Output of length 100	0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1,	The sequence of booleans has a few lengthy continuous runs of the same value. This is less obvious than that of the Inversive

	1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0	Congruential Generator. This is important to note should this generator be used in certain applications.
Integer Output in Range [0, 100) of length 100	91, 60, 51, 73, 67, 7, 26, 1, 11, 46, 33, 8, 21, 81, 27, 18, 26, 8, 20, 91, 42, 63, 29, 46, 2, 96, 21, 60, 59, 79, 10, 20, 18, 49, 73, 10, 85, 77, 43, 22, 76, 58, 73, 98, 90, 5, 76, 8, 95, 18, 47, 42, 43, 18, 35, 56, 6, 71, 41, 75, 24, 28, 48, 79, 5, 44, 58, 46, 71, 43, 93, 7, 49, 11, 63, 52, 42, 62, 10, 42, 10, 2, 53, 1, 98, 9, 8, 90, 8, 46, 46, 66, 36, 3, 77, 73, 95, 63, 96, 63	The sequence of integers has a few lengthy continuous runs of even values and odd values. This is less obvious than that of the Inversive Congruential Generator. This is important to note should this generator be used in certain applications.

Testing Random Number Generators

There are numerous ways to test random number generators as there are several requirements that random number generators must satisfy. Some key ways to test random number generators are introduced within [1]. We will focus on testing uniform random number generators as it is expected that the generators focused on within this paper all output values at the same rate regardless of value. The two tests that were implemented were the mean test and the chi-squared test.

Mean Test

The mean test was taken from [1] as it allowed for the testing of the integer values natively generated by several of the generators discussed within this report. The mean test was not able to be done on the boolean outputs from the random number generators. The basis of a mean test is to determine if the random number generator outputs a stream of values that, when averaged, equals the expected mean, within a set confidence interval. This test utilizes the central limit theorem. The central limit theorem states that when you average a sequence of n integers that come from a uniform distribution with mean μ that the average tends towards μ as $n \to \infty$. See Algorithm 13 for how the mean test is implemented and see Figure 1-3 to see the results from the mean test.

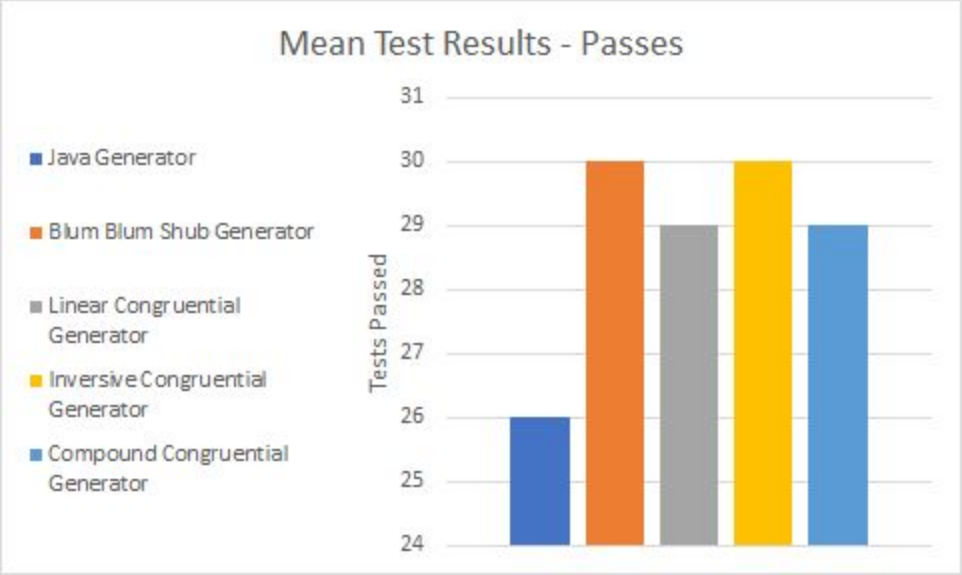
Algorithm 13: Mean Test for Integers

Description	Conducts a mean test for integers on a generator
Input	• gen - the random number generator being tested • s - integer value such that $s > 0$; total number of samples

	• t - integer value such that $t > 0$; number of tests • min - minimum value generated by gen • lessThan - upper bound (exclusive) of value generated by gen
Output	Boolean value where true means the test passed
Method	1. $l = s / t$ 2. $means = empty\ list$ 3. For $i = 1$ *to* t a. $vals = list\ of\ length\ l\ of\ integers\ from\ gen$ b. $add\ mean\ of\ vals\ to\ means$ 4. $mean = mean\ of\ means$ 5. $variance = variance\ of\ means$ 6. $interval = 1.960 * sqrt(variance) / sqrt(t)$ // Confidence interval 7. $expected = (lessThan - 1 - min) / 2$ // Expected mean 8. Return $expected \geq mean - interval\ \&\ expected \leq mean + interval$

Figure 1: Passes of The Mean Test

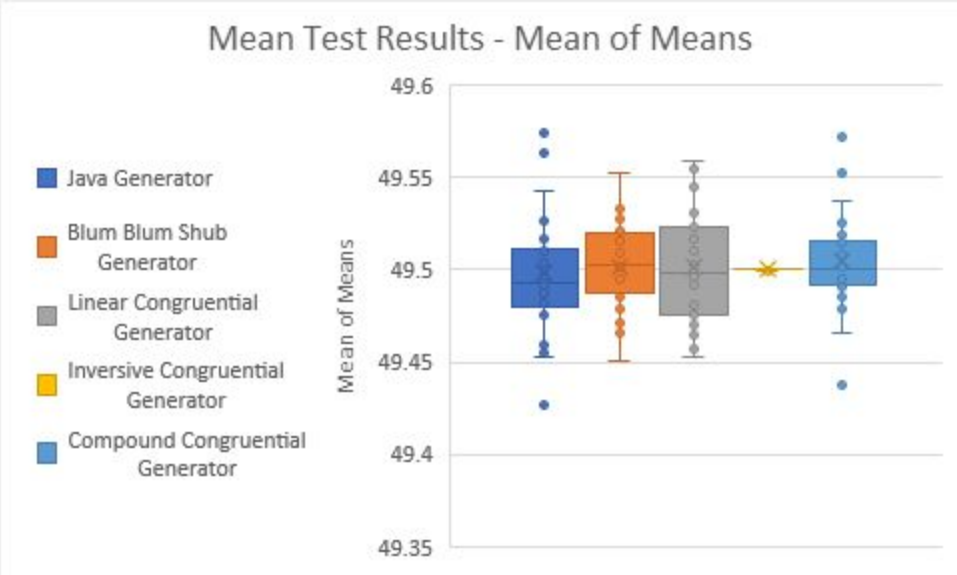
The mean test was completed on each of the generators 30 times with 1000000 samples and 1000 tests each. The range of the values generated by the generators was [0,100) yielding an expected mean of 49.5.



By analyzing the results in Figure 1 it is obvious that all of the generators passed the majority of the mean tests. Since the test utilizes a 95% confidence interval it is expected that these tests should fail once in awhile regardless of the generator being used. With that being said the Java generator did do worse than the other generators and since the test was run 30 times it would be unexpected that a single generator would fail 4 times.

Figure 2: Mean of Means

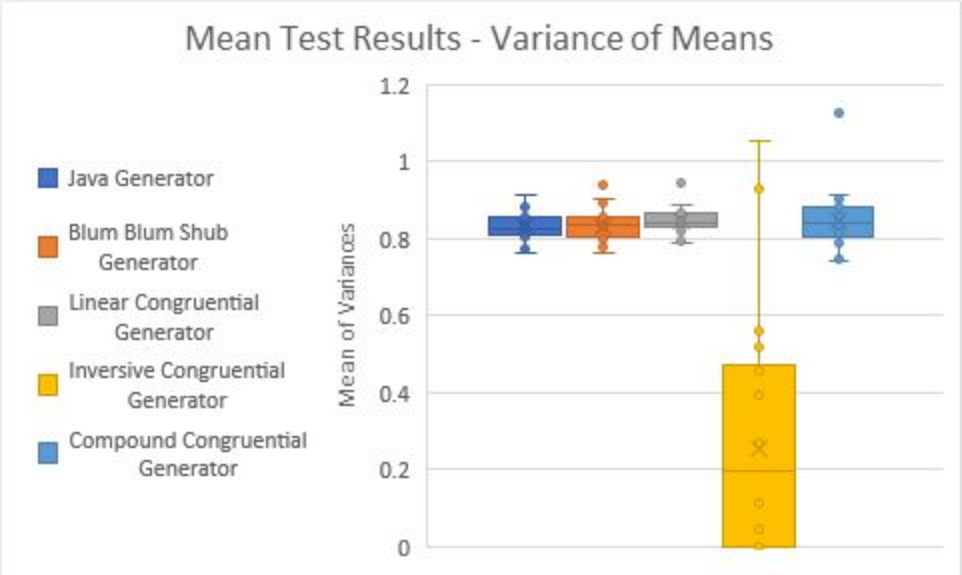
The mean test was completed on each of the generators 30 times with 1000000 samples and 1000 tests each. The range of the values generated by the generators was [0,100) yielding an expected mean of 49.5. The mean from these tests was plotted using a box and whisker chart.



By analyzing the results in Figure 2 it is obvious that all the generators have the expected mean of 49.5 within their boxes and that the whiskers for all the generators are very short, as desired. Also, the Inversive Congruential Generator definitely takes the lead when it comes to its ability to randomly generate numbers in a prescribed range.

Figure 3: Variance of Means

The mean test was completed on each of the generators 30 times with 1000000 samples and 1000 tests each. The range of the values generated by the generators was [0,100) yielding an expected mean of 49.5. The variance from these tests was plotted using a box and whisker chart.



By analyzing the results in Figure 3 it is apparent that all the generators but the Inversive Congruential Generator provided similar results when it comes to their variance. For the Inversive Congruential Generator the variance was normally much lower than that of the other generators, as would be expected due to the small box for the Inversive Congruential Generator in Figure 2.

Chi-Squared Test

The chi-squared test was taken from [1] as it presented another test that would provide a definitive answer as to whether or not the random number generator is considered to have passed the test. Additionally, the chi-squared test is able to be tested against both the integer and boolean implementations of all the generators. The basis of the mean test is to determine if the list of numbers generated by a random number generator appropriately represents what should be expected by a random number generator. This is accomplished by comparing the expected number of times a particular output is expected by the random number generator and the observed number of times a particular output is generated. This test yields a single value which is then compared to the value of chi-squared at various values for α. Each test that yields a value less than the chi-squared value the test is considered to have been passed. See Algorithm 14 for how the chi-squared test is implemented and see Figure 4-7 for the results to the chi-squared test.

Algorithm 14: Chi-Squared Test for Integers

Description	Conducts a chi-squared test for integers on a generator
Input	• gen - the random number generator being tested • s - integer value such that $s > 0$; total number of samples • min - minimum value generated by gen

	• lessThan - upper bound (exclusive) of value generated by gen
Output	Integer value representing the number of tests passed
Method	1. $vals = list\ of\ length\ s\ of\ integers\ from\ gen$ 2. $count = list\ of\ zeros\ of\ length\ (lessThan - min)$ 3. For each $val\ in\ vals$ a. $count[val]$ += 1 4. $expected = s\ /\ (lessThan - min)$ // Expected for each value 5. $chi = 0$ // The chi-squared value 6. For each $c\ in\ count$ a. chi += $(c - expected)^2\ /\ expected$ 7. $passes = 0$ 8. If $chi < 77.046$ then $passes$ += 1 // $\alpha = 95\%$ 9. If $chi < 81.449$ then $passes$ += 1 // $\alpha = 90\%$ 10. If $chi < 89.181$ then $passes$ += 1 // $\alpha = 75\%$ 11. If $chi < 98.334$ then $passes$ += 1 // $\alpha = 50\%$ 12. If $chi < 108.09$ then $passes$ += 1 // $\alpha = 25\%$ 13. Return $passes$

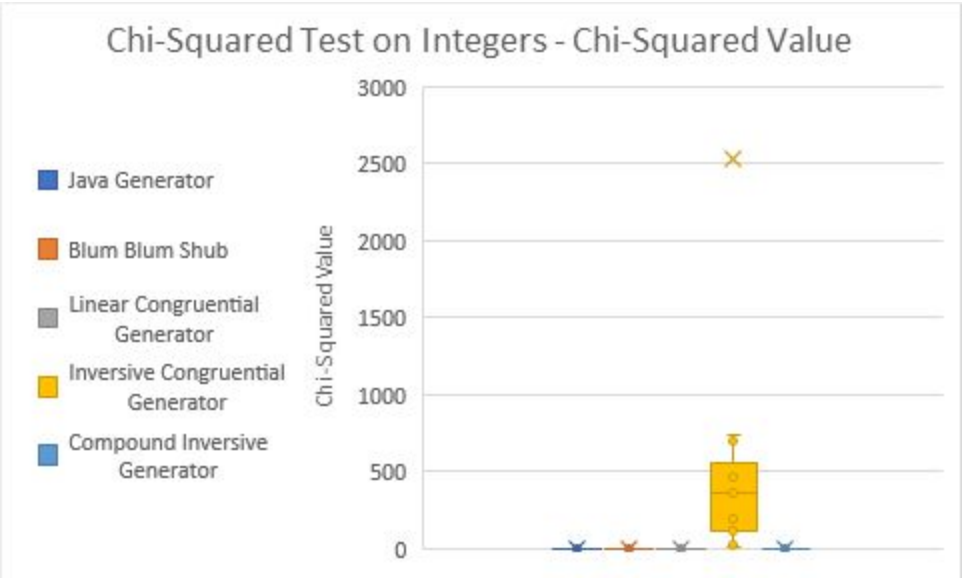
Algorithm 15: Chi-Squared Test for Integers

Description	Conducts a chi-squared test for integers on a generator
Input	• gen - the random number generator being tested • s - integer value such that $s > 0$; total number of samples • min - minimum value generated by gen • lessThan - upper bound (exclusive) of value generated by gen
Output	Integer value representing the number of tests passed
Method	1. $vals = list\ of\ length\ s\ of\ booleans\ from\ gen$ 2. $count = list\ of\ zeros\ of\ length\ 2$ // True and False 3. For each $val\ in\ vals$ a. If $val == true$ then $count[1]$ += 1 b. If $val == false$ then $count[0]$ += 1 4. $expected = s\ /\ 2$ // Expected for each value 5. $chi = 0$ // The chi-squared value 6. For each $c\ in\ count$ a. chi += $(c - expected)^2\ /\ expected$ 7. $passes = 0$ 8. If $chi < 0.00393$ then $passes$ += 1 // $\alpha = 95\%$ 9. If $chi < 0.0158$ then $passes$ += 1 // $\alpha = 90\%$ 10. If $chi < 0.102$ then $passes$ += 1 // $\alpha = 75\%$ 11. If $chi < 0.455$ then $passes$ += 1 // $\alpha = 50\%$ 12. If $chi < 1.323$ then $passes$ += 1 // $\alpha = 25\%$

	13.Return *passes*

Figure 4: Chi-Squared Value for Integers

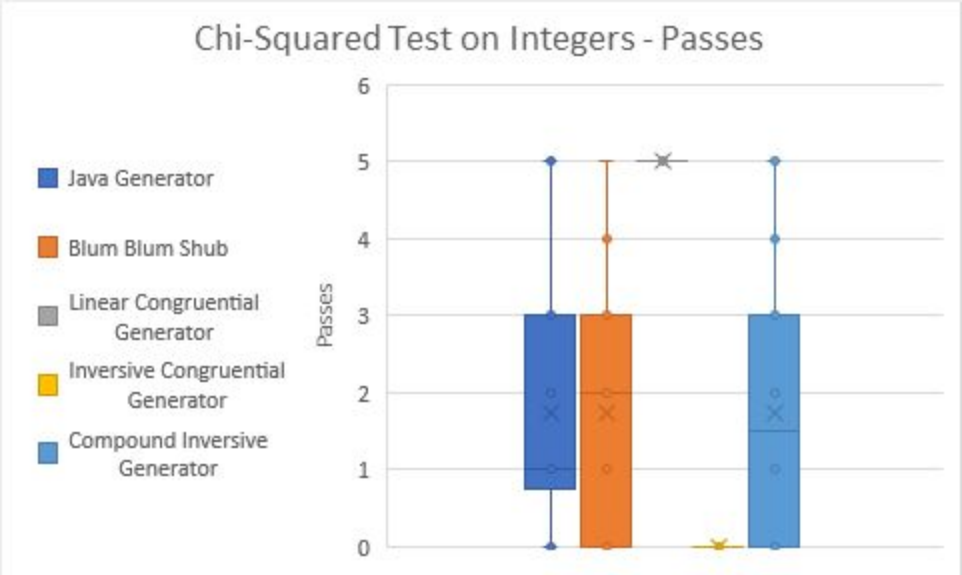
The chi-squared test was run on each of the generators 30 times with 1000000 samples. The range of the values generated by the generators was [0,100) meaning each value should be returned 10000 times. The chi-squared value from these runs was plotted using a box and whisker chart.



By analyzing the results in Figure 4 it is apparent that all of the generators but the Inversive Congruential Generator have a low and similar chi-squared value. The mean of the chi-squared value for all the generators, in the order they appear in the graph, are as follows: 0.927, 1.178, 0.0, 2535.832, 0.986. Therefore, the best generator for the chi-squared test on integers is obviously the Linear Congruential Generator.

Figure 5: Chi-Squared Passes for Integers

The chi-squared test was run on each of the generators 30 times with 1000000 samples. The range of the values generated by the generators was [0,100) meaning each value should be returned 10000 times. The number of passed tests from each run was plotted using a box and whisker chart.



By analyzing Figure 5 the results for the Java, Blum Blum Shub and Compound Inversive generator have similar results. Whereas, the obvious worst is the Inversive Congruential Generator and the obvious best is the Linear Congruential Generator.

Figure 6: Chi-Squared Value for Booleans

The chi-squared test was run on each of the generators 30 times with 1000000 samples. This means that both true and false should be returned 500000 times. The chi-squared value from these runs was plotted using a box and whisker chart.

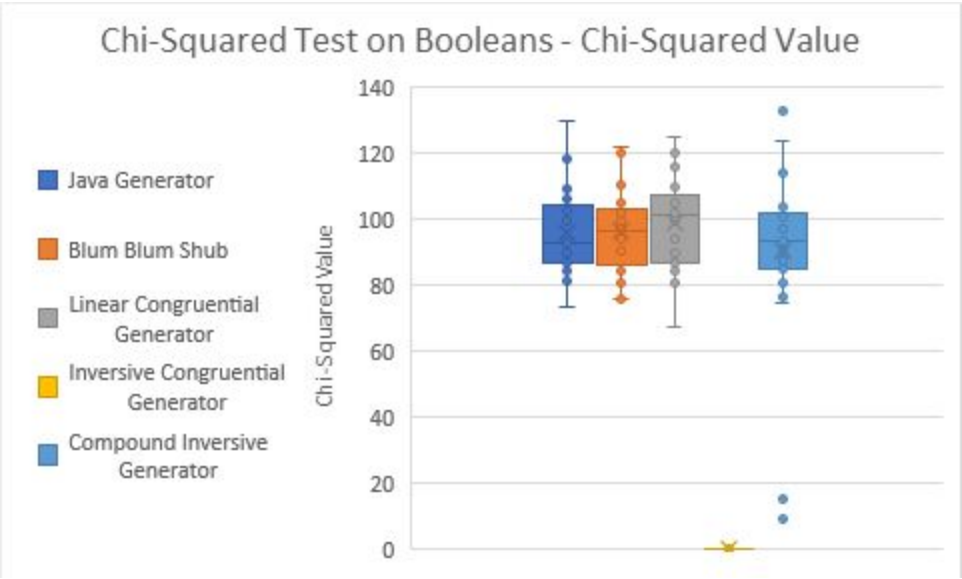


Figure 6 demonstrates that when it comes to boolean values the Inversive Congruential Generator provides superior results to that of the other generators. While, the other generators experience similar results.

Figure 7: Chi-Squared Passes for Booleans

The chi-squared test was run on each of the generators 30 times with 1000000 samples. This means that both true and false should be returned 500000 times. The number of passed tests from each run was plotted using a box and whisker chart.

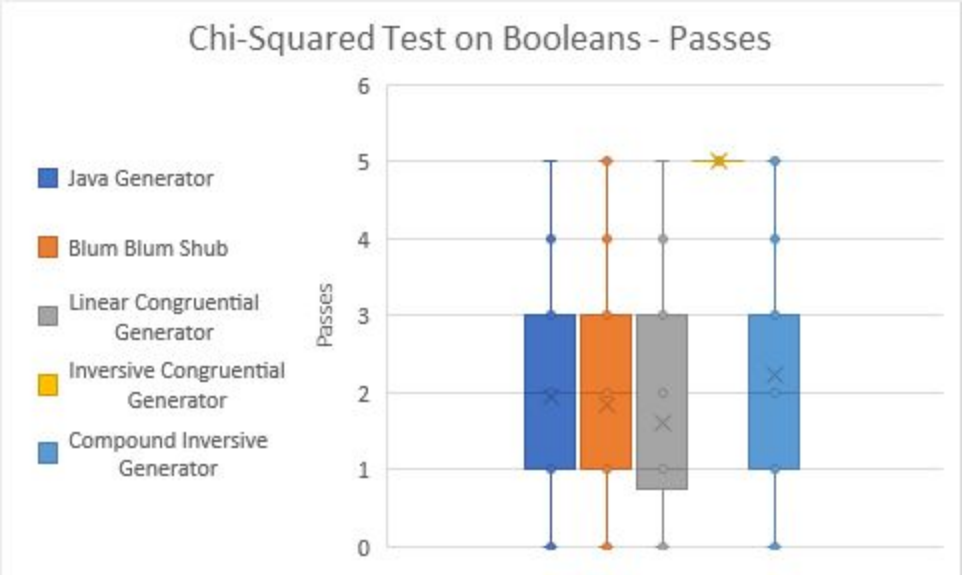


Figure 7 adds more evidence to say that the Inversive Congruential Generator is the superior generator when it comes to producing boolean values in a manner that satisfies the chi-squared test.

Conclusions

The most interesting random number generator implemented within this project was the Inversive Congruential Generator due to the fact that it has demonstrated the ability to be the best and worst random number generator in the tests implemented within this project. This provides evidence towards the assertion made in [1] that claims that in order to adequately test a random number generator several tests must be completed.

This project does not yield a single best random number generator and instead provides three generators which provide more or less the same results from all tests. These generators are: Blum Blum Shub, Linear Congruential Generator and Compound Inversive Generator as they did not compare any worse than the other generators in the majority of the tests. One outlier here would be from Figure 5 where the Linear Congruential Generator performed significantly better than all the other generators. Although, the boolean output from the Linear Congruential Generator switches back and forth between true and false and this would defy the idea that the next output from a generator should be unknown given the current value.

This project only implements the vanilla version of each of the generators. Therefore, there are several other implementations that have the ability of providing better or worse results for each of the tests presented here. The references included feature many additional tweaks to the generators which should result in better generators overall.

When comparing the output from the Inversive Congruential and Compound Inversive generators it is important to note that the strings of the same boolean value is less apparent within the Compound Inversive Generator than that of the Inversive Congruential Generator. This is likely because the foundation of the Compound generator is the Inversive generator and therefore, it follows that a shortfall of the Inversive generator would remain in the Compound generator. Though since the inversive generator produces 1s and 0s at appropriate rates and since several Inversive generators are used in a Compound generator the strings of the same value are shorter and less frequent than the Inversive generator.

When looking at the results of the chi-squared test for the Inversive Congruential Generator it is interesting that this generator was best for the boolean chi-squared test and the worst for the integer chi-squared test. This leads to the idea that a generator may need to be chosen based on the type of output required and the characteristics of said output.

This project was an introduction to various random number generators and ways of testing said generators. There are numerous other generators and tests that have yet to be explored and I will likely continue to implement and test these generators. Thank you for the great term!

References

[1] A. Goucher and T. Riley, *Beautiful Testing: Leading Professionals Reveal How They Improve Software*. "O'Reilly Media, Inc.," 2009.

[2] "Random (Java Platform SE 7)," 27-Feb-2018. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/Random.html. [Accessed: 23-Apr-2018].

[3] L. Blum, M. Blum, and M. Shub, "A Simple Unpredictable Pseudo-Random Number Generator," *SIAM J. Comput.*, vol. 15, no. 2, pp. 364–383, 1986.

[4] T. E. Hull and A. R. Dobell, "Random Number Generators," *SIAM Rev.*, vol. 4, no. 3, pp. 230–254, 1962.

[5] J. Eichenauer and J. Lehn, "A non-linear congruential pseudo random number generator," *Statistische Hefte*, vol. 27, no. 1, pp. 315–326, 1986.

[6] J. Bubicz and J. Stoklosa, "Compound Inversive Congruential Generator Design Algorithm," 2018.