Digital root study and its use in the generation of random number sequences

Tatiana Montes Celinski, Lucas Lourival Alves

State University of Ponta Grossa (UEPG) – Ponta Grossa, Paraná, Brazil

tmontesc@uepg.br, lucalves7@outlook.com

Abstract. *This article presentes an overview of the digital root, demonstrating its functions, methods and applications. This article also presentes a study about the application of the digital root in the generation of random sequences of numbers. For this, an algorithm for generation of random sequence of numbers is proposed, whose results as to its usability, using a battery of tests, are presented in this article.*

Keywords: *Random sequence generation. Digital root characteristic. Digital root applications.*

1. Introduction

In the digital world, the use of mathematical methods and formulas is a common practice for solving problems related to the speed or performance of computer programs as well as other purposes. In this sense, digital root presents a mathematical formulation not yet applied to the present moment. That said, this work presents a study on digital root, its functionalities and applications, as well as the development of an application for the generation of random sequences.

Digital root, also known as seed number, is a value obtained from a non-negative integer by adding the digits of that number repeatedly until a single digit is obtained. Although digital root has few applications studies currently, its base, digital sum, which is the simple sum of the digits of a number, is used in algorithms for checking messages in Internet protocols (IP) (BARR, 1999).

The study on the use of digital root in the area of random number generation is justified because the basic property of digital root favors the creation of simple numbers (from a single digit) from complex numbers, further modifying these sequences of numbers .

While there are very good random number generators, such as SIMD-oriented Fast Mersenne Twister (SFMT), Permuted Congruential Generator (PCG) and Xorshift, there are other bad generators, such as RANDU and Fibonacci sequence, that when passed by batterie tests to determine their usability, get negative results most of the time. Thus, it is justified to use digital root in these latter algorithms, to make them viable and usable to generate random numbers.

Therefore, this work has the general objective to present what is digital root, its characteristics, properties and applications. More specifically, an algorithm is presented using digital root and also its evaluation in the creation of a random sequence of digits, from the application of a set of tests. In addition, the algorithm developed is also tested on bad generators to compare with the results of the original generator.

2. Theoretical review

This section presents the main concepts about digital root and random numbers, as well as related work on the subject, including digital root applications.

2.1. *Digital Root*

Digital root, according to Averbach and Chain (2000), is defined as "one-digit number obtained by summing all digits of the original number to obtain a new number, then adding all the digits of the new number to obtain a third number, and so on until it results in a one-digit number".

To illustrate, digital root of a number *n*, which is the concatenation of *m* digits that it contains, will be the sum of those *m* digits that are concatenated, as long as the result has a single digit. For *n* equal to 318, with *m* equal to 3, we have 3+1+8 resulting in 12, which is not yet the digital root of 318. Thus, considering the previous result, 12, a new *n*, for which repeats the process. Therefore, 1+2 equals 3, this being the digital root of 318 and consequently of 12 and itself (3).

Thus, it is possible to verify that the numbers resulting from digital root are in an infinite rotation, that is, if *n* is a number between 1 and 9, then *n*+1 will be the next count, where *n* is equal to 9, *n*+1 will have the value of 1, restarting the rotation.

Among some applications, it is possible to use digital root as a real proof of a value. So, for example, digital root is used to know if 2758+1475, whose result is 4233, is correct. Thus, if *dr* (2758) = 4 and *dr* (1475) = 8, *dr* (4+8) = 3, and if *dr* (4233) = 3, the result is correct (MENTAL MATH: DIGITAL ..., 2008).

Aprajita and Kumar (2015, p. 19) have developed a study on digital root where they bring their concept, fundamentals and possible applications. As a proposition, they define the following.

> If *N* be the set of natural numbers and *D* = {1,2,3,4,5,6,7,8,9}, then the function *dr: N* → *D* is well defined.
> **Proof:** let *a*,*b*∈*N* and *a* = *b*, then obviously digits of *a* and *b* are the same. Therefore, sum of the digits of "*a*"= sum of the digits of "*b*", and sum of digits of "*a*" till a single digit is obtained = sum of digits of "*b*" till a single digit is obtained, that is *dr(a)* = *dr(b)*. Therefore, *dr:N*→*D* is well defined.

Based on this, the authors formulated some results as digital root partition the set of non-negative integers. (APRAJITA; KUMAR, 2015, p. 19)

> Let the relation is defined as *aRb* ↔ *a*~*b*,e sabendo que a~b ↔ (dr(a)= dr(b))tem-se que:
> - R is reflexive: *dr(a) = dr(b)* ↔ *a~b* → *aRb;*
> - R is symmetric: *aRb*→*dr(a) = dr(b)* →*dr(b) = dr(a)*→ *b~a* → *bRa;*
> - R is transitive: *aRb* e *bRc*→*dr(a) = dr(b) e dr(b) = dr(c)* →*dr(a) = dr(b) = dr(c)* →*dr(a) = dr(c)* →*a~c* → *aRc*
>
> Therefore it is a equivalence relation and it partition the set of non-negative integers.

Izmirli (2014, p. 301) has published an article in which he defines some properties of the digital root and consequently it demonstrates an application using the digital root, detailed below.

> Suppose we have a five-digit number. We are given that this number is divisible by 72. Starting with the first one, how many digits of this number must be disclosed before we can uniquely determine it?

Assume we are given the first digit, say 4. Obviously, more information will be needed before a unique solution is found. For exemple, 46,800 = 650 x 72, 48,600 = 675 x 72, ... all fit the bill. So, assume now the second digit is also given, say 8. Again, we can not find a unique solution based on this information: 48,321=671 x 72, 48,600 = 675 x 72, ..., are all possible solutions. So, assume one more digit is given, say 9. We claim this would be enough to solve the problem.
If a number is divisible by 72, it must be divisible by both 8 and 9. But a number is divisible by 8 only if one of the two conditions holds: The hundreds digit is even and the last two digits are a multiple of 8 or the hundreds digit is odd and the last two digits are a multiple of 4 but not 8. Since in our example the hundreds digit is odd, the last two digits of the number we are looking for must be a multiple of 4 but not 8, that is, the last two digits must be one of 04 12 20 28 36 44 52 60 68 76 84 92. On the other hand, to be divisible by 9, the digital root of the number must be 9. So the answer will be 48,960, since dr (48,960) = 9 and 680 x 72 = 48,960.

2.2. Random Number Sequences

According to Casquilho (2007), "algorithms of the genre [random number generation] that can be implemented in a computer program are, by definition, completely deterministic and can never, therefore, produce a random effect of any kind". In this sense, the algorithms used to generate numbers never generate really random numbers, but pseudorandom numbers, which "simulate" the behavior of random numbers. There are already in the market equipment that, through physical phenomena, seek to generate "more random" numbers. However, these equipments are still expensive (COMO OS..., 2010).

Thus, knowing that computers generate pseudorandom numbers, it is necessary to understand theoretically what this means. In this regard, Katz and Lindell (2008) state: "A pseudo-random chain is a chain that looks like a uniformly distributed chain, as long as the "observing" entity executes it in polynomial time. Just as indistinguishability can be visualized as a computational loosening of perfect stealth, pseudorandomness is the computational loosening of true randomness".

Pseudo-randomness refers to the distribution in strings, so a distribution D over a chain of length L is pseudorandom if D is indistinguishable from the uniform distribution over the strings of length L. Thus, it is impracticable for any polynomial time algorithm to determine whether it is given a chain that is in accordance with the D distribution or if a randomly chosen L chain is given.

Thus, a pseudorandom generator is a deterministic algorithm that receives a small initial and really random value, called seed, and extends it into a long chain that is pseudorandom. Otherwise, a pseudorandom generator uses a small amount of real randomness to generate a large amount of pseudorandomness. Thus, with n being the seed length which is the input of the generator and *L (n)* the output length.

Since L (.) is a polynomial and G is a deterministic polynomial time algorithm, such that any input $s \in \{0,1\}^n$, and the algorithm G will output a chain of length L (n). G will be a pseudo-random generator provided that it follows the following two conditions:

1) (Expansion) For each n, $L(n) > n$;

2) (Pseudo-randomness) For all the probabilistic polynomial time differentiators D, there is a negligible *negl* function such that:

$$\left| \Pr[D(r) = 1] - \Pr[D(G(s)) = 1] \right| \leq \mathsf{negl}(n),$$

where r is chosen uniformly at random from {0,1} $^{L(n)}$, the seed is chosen uniformly at random from {0,1} ^{n}, and the probabilities are taken on the random currencies used by D and the choice of *r* and *s*. The function L (.) is called the expansion factor of G. (KATZ, J; LINDELL, Y, 2008).

2.3. Pseudo-random generators

There are several random computational generators, which generate numbers in a pseudo-random chain. Among them, the congruent generator and the Fibonacci sequence, which are two bad generators and are used in composition to the digital root for the creation of a generator better than the originals, are approached in this work.

The first one, the congruent generator (cong ()), is a generator that uses the multiplier: x (n) = 69069x (n-1) + 1234567, initially x = 123456789. The generator has period 2^{32}. When the sequence is created, the final half of it is usually very regular, which leads to being a bad generator.

The other generator uses the classical Fibonacci sequence (fib ()), where x (n) = x (n-1) + x (n-2), with modulo 2^{32}. Its period is 3 * 2^{31} if one of the two seeds are odd and not 1 mod 8. As a pseudorandom generator, it is not very functional, but provides a simple and fast function to be used in other generators.

3. Methodology

For the development of the algorithm, we used the C programming language, which is easy to use and understand. In addition, the program used for programming the algorithm is the Code :: Blocks, open source IDE that is mainly focused on C, C ++ and Fortran.

In the proposed model, the digital roots of random values obtained using the rand function of language C are determined. Then a simple frequency test is performed to verify that the determined values occur at the same frequency, with a very small percentage variation, and can proceed with the methodology if this variation is less than 0.1%.

Subsequently digital roots are transformed into 3-bit values, and for each value other than digital root, a value in the range of 1 byte is related. Thus, the value 1 of the digital root is bit 0 of the 8 bits of 1 byte, with 2 being bit 1, value 3 is bit 2 and so on, being 8 bit 7. In this case, it is necessary to ignore the value 9 of the digital root, because it does not fit in the byte and is one of the limits of digital root, besides being a null variable. Then one can say that the value 1 of the digital root is 000, the value 2 is 001, ..., and 8 is 111. Finally these values are summed 11 times in a single variable forming a random value of 33 bits, with the last bit being ignored, which leads to a value of 32 bits.

The values must be 32 bits, since the tests used, provided by the Center for Information Security & Cryptography (CISC) of the University of Honk Kong (HKU) (TSANG, W; 2002a), require this format. The tests used are: GCD, Birthday Spacings, Maurer, Collision, Frequency and Gorilla.

From the results of the tests, the evaluation of the use of digital root to improve random number generators considered bad is carried out. The tests are detailed below.

3.1. GCD test

The GCD (Greatest common divisor) is the maximum common divisor test. It is used to analyze the dependency and optimization of computational loops. The test is based on verifying the dependence of 2 random numbers using the Euclid algorithm, also checking the number of steps. The test result is the general value *p*. This value, for a positive result, must be less than 1, and greater than 0 for the generator to pass the test. (MARSAGLIA, G., 2002)

3.2. Birthday Spacings test

Birthday Spacings é um teste baseado no paradoxo do aniversário, da teoria das probabilidades. Os aniversários são escolhidos aleatoriamente de um ano e então são ordenados. Os espaçamentos entre os aniversários são gravados, checando a distribuição desses espaçamentos com um valor geral *p*. Esse valor, assim como o GCD, deve ser menor que 1 e maior que 0 para o gerador passar no teste (MARSAGLIA, G., 2002).

3.3. Maurer test

Maurer's universal test focuses on the number of bits between corresponding patterns (a measure related to the length of a compressed sequence). The purpose of the test is to detect whether or not the sequence can be significantly compressed without loss of information. If it is significantly compressible it can not be considered random. The result, *p*, must be greater than 0 and less than 1 for the generator to pass the test (TSANG, W; 2002e).

3.4. Collision test

The collision test is a statistical test that simulates balls being played in urns at random. The number of polls *m* is a power of 2, while the number of poles is *n*. The target of a ball must be determined by $\log_2 m$ bits produced by the random number generator to be tested. When a ball falls into an urn that is already occupied, a collision occurs. The test counts the number of collisions *c*. The generator fails the test if the number *c* is outside the predefined range. The result, *p*, must be greater than 0 and less than 1 for the generator to pass the test (TSANG, W, HUI, L., CHOW, K; CHONG, C., 1999) (TSANG, W; 2002b).

3.5. Frequency test

The frequency test focuses on the ratio of zeros and ones to a complete sequence. Its purpose is to determine whether the number of zeros and ones in a sequence is approximately the same as would be expected from a truly random sequence. The test evaluates the proximity of the fraction of ones to ½, that is, the number of ones and zeros in sequence should be approximately the same. There is no evidence to indicate that the sequence tested is non-random. The resulting *p* value must not be equal to or greater than 1 or less than or equal to 0 for the generator to have a positive test result (TSANG, W; 2002c).

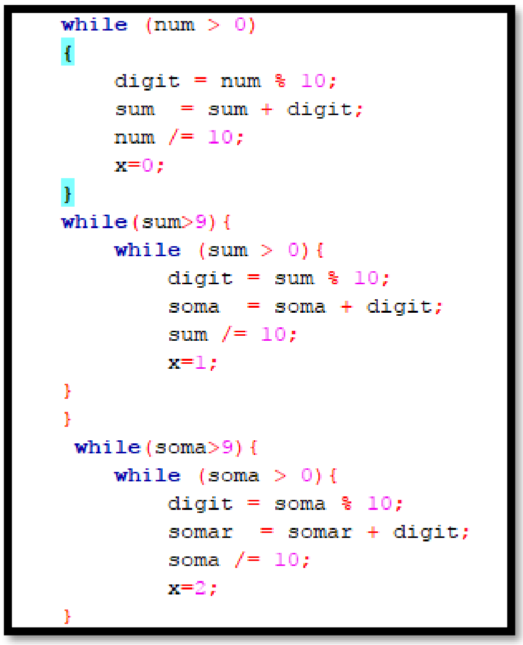
3.5. Gorilla test

The Gorilla test is based on the infinite monkey theorem, where a monkey typing randomly on a keyboard for an infinite amount of time will surely create any text. So for 32-bit integers produced by the generator, one of the 32-bit possible positions is specified, with the bit numbered from 0 to 31 from least to most significant. A 2^{26}+25 sequence of these bits is obtained, consisting of the designated bit of each of the $2^{26}+25$ integers of the generator. If *x* is the number of 26-bit words that do not appear in the sequence, then *x* must be approximately normally distributed with the mean 24687971 and standard deviation 4170, so that f((x-24687971)/4170) must be normally distributed in [0,1], where f() is the standard normal distribution function. The value of *p* must beless than 1 and greater than 0 for the result to be positive for the generator. (TSANG, W; 2002d).

5. Results and discussion

Initially, the code of transformation of random numbers obtained from the rand () function was done for their respective digital roots. The developed code is shown in Table 1.

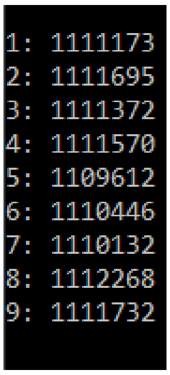
Table 1 - Code to obtain the digital root of a number



```
while (num > 0)
{
    digit = num % 10;
    sum  = sum + digit;
    num /= 10;
    x=0;
}
while(sum>9){
    while (sum > 0){
        digit = sum % 10;
        soma  = soma + digit;
        sum /= 10;
        x=1;
}
}
 while(soma>9){
    while (soma > 0){
        digit = soma % 10;
        somar  = somar + digit;
        soma /= 10;
        x=2;
}
```

The code in Table 1 has, in addition to the initial *while*, two more *whiles* for the two possible iterations required to obtain a single digit of the random value, being a maximum value of 999999999. Based on the obtained values, it is now necessary to test if they have a frequency of values that is approximately the same, with a minimum percentage change. Thus, using a simple code to visualize the frequency, the results are presented in Table 2.

Table 2 - Results of frequency of values obtained from digital root

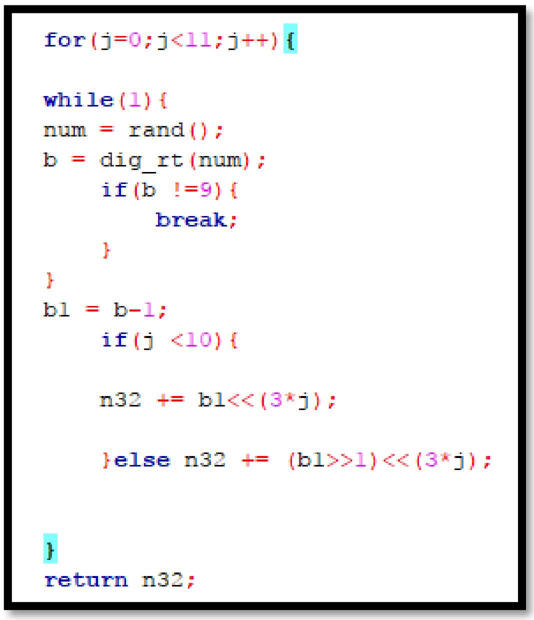


```
1: 1111173
2: 1111695
3: 1111372
4: 1111570
5: 1109612
6: 1110446
7: 1110132
8: 1112268
9: 1111732
```

According to Table 2, each value next to the numbers 1 through 9 represents the number of times this number appeared in the random number sequence. Thus, knowing that the total number was 10000000 (10 million), all values correspond to approximately 11%, which leads to an equal frequency for all values, demonstrating a positive result.

The next step is to transform the obtained digital roots into 3-bit values, ignoring 9, according to the code shown in Table 3.

Table 3 - Digital roots transformation code for 3-bit values

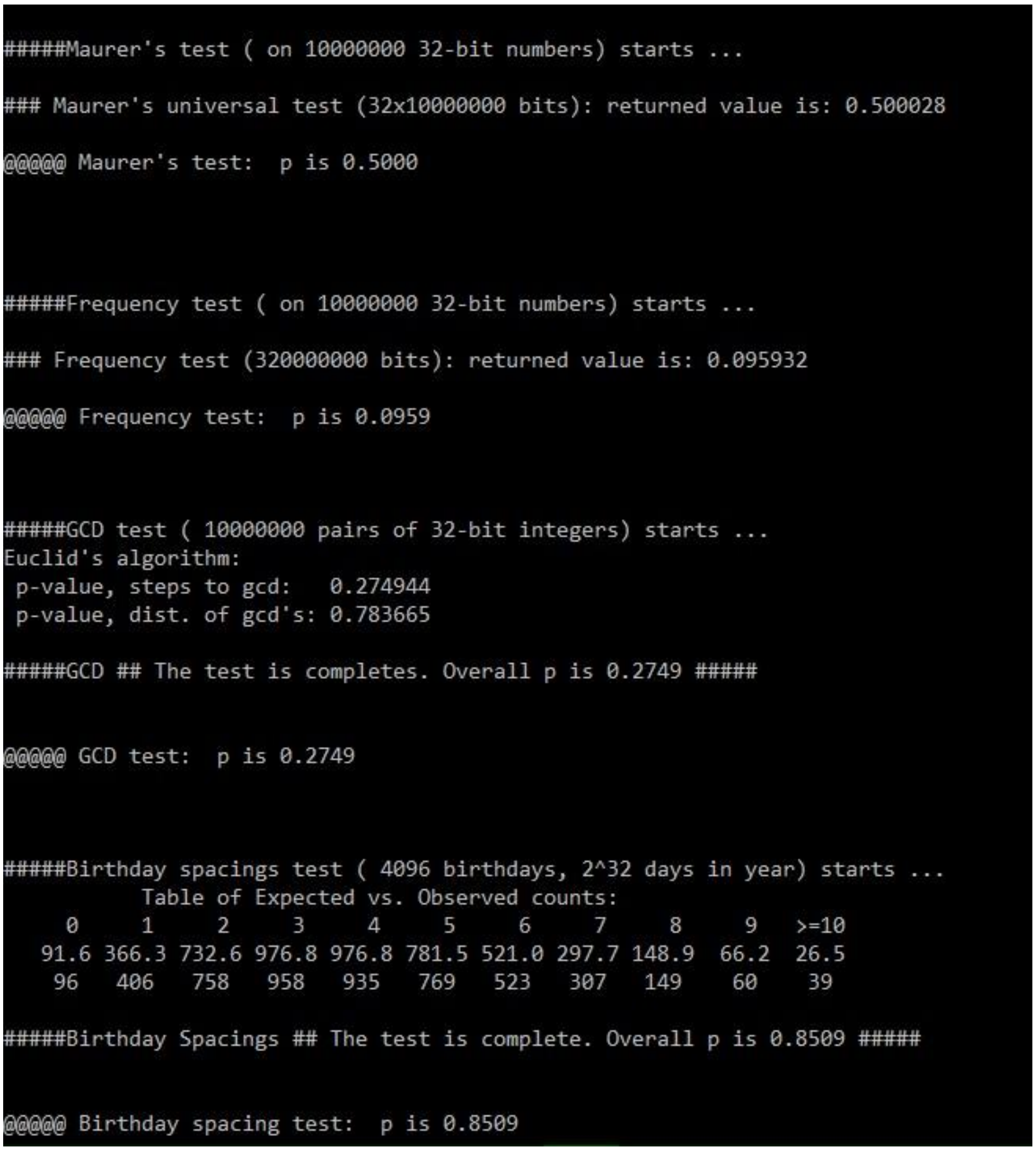


```
for(j=0;j<11;j++){

while(1){
num = rand();
b = dig_rt(num);
    if(b !=9){
        break;
    }
}
b1 = b-1;
    if(j <10){

    n32 += b1<<(3*j);

    }else n32 += (b1>>1)<<(3*j);


}
return n32;
```

To obtain a 32-bit sequence, it is necessary to repeat the routine of obtaining 3 bits of digital roots 11 times, ignoring the last bit in the last iteration. These bits are then summed and transformed into a possibly random value, which will be input from functions that will test whether the generator actually generates random numbers.

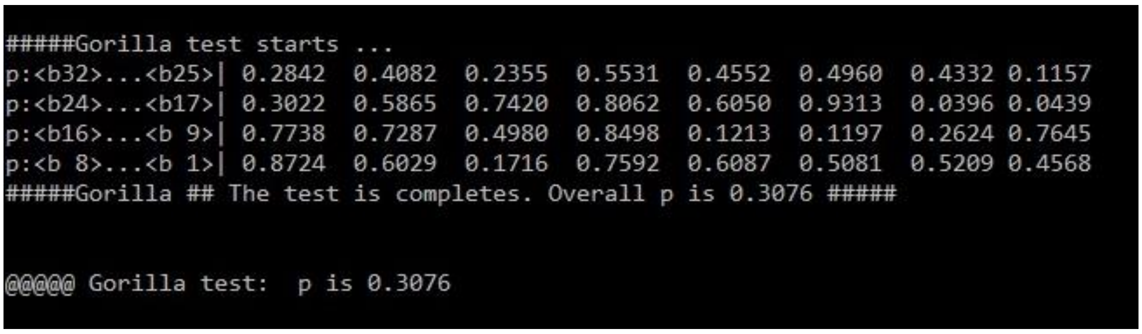
Then the GCD, Birthday Spacings, Maurer, Collision, Frequency and Gorilla tests are applied to the data obtained from the function shown in Table 3. The test results are presented in Tables 4 and 5.

Table 4 - Results of the Maurer, Frequency, GCD and Birthday Spacings tests for the 32-bit sequence

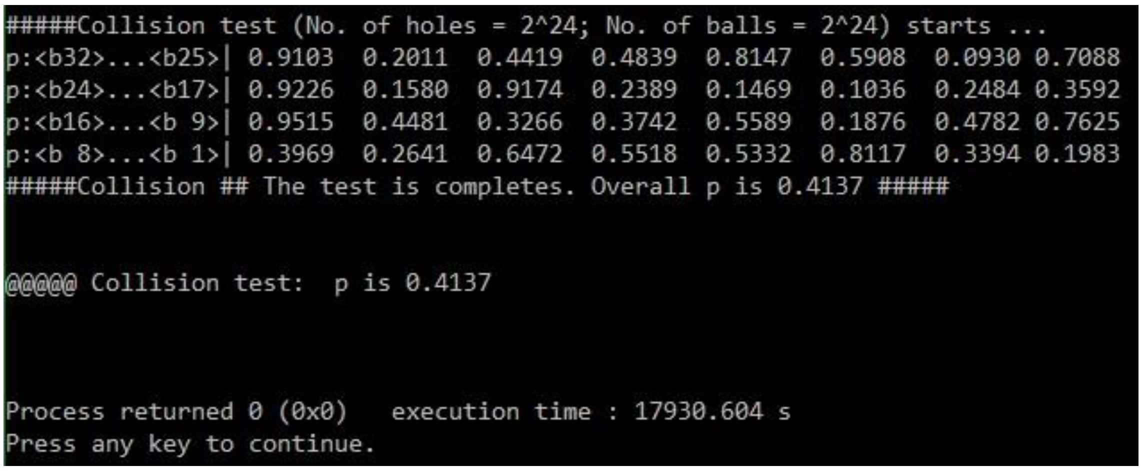


```
#####Maurer's test ( on 10000000 32-bit numbers) starts ...

### Maurer's universal test (32x10000000 bits): returned value is: 0.500028

@@@@@ Maurer's test:  p is 0.5000



#####Frequency test ( on 10000000 32-bit numbers) starts ...

### Frequency test (320000000 bits): returned value is: 0.095932

@@@@@ Frequency test:  p is 0.0959


#####GCD test ( 10000000 pairs of 32-bit integers) starts ...
Euclid's algorithm:
 p-value, steps to gcd:   0.274944
 p-value, dist. of gcd's: 0.783665

#####GCD ## The test is completes. Overall p is 0.2749 #####


@@@@@ GCD test:  p is 0.2749



#####Birthday spacings test ( 4096 birthdays, 2^32 days in year) starts ...
           Table of Expected vs. Observed counts:
     0     1     2     3     4     5     6     7     8     9   >=10
   91.6 366.3 732.6 976.8 976.8 781.5 521.0 297.7 148.9  66.2  26.5
    96   406   758   958   935   769   523   307   149    60    39

#####Birthday Spacings ## The test is complete. Overall p is 0.8509 #####


@@@@@ Birthday spacing test:  p is 0.8509
```

Table 5 - Gorilla and Collision tests for the 32-bit sequence



```
#####Gorilla test starts ...
p:<b32>...<b25>| 0.2842  0.4082  0.2355  0.5531  0.4552  0.4960  0.4332 0.1157
p:<b24>...<b17>| 0.3022  0.5865  0.7420  0.8062  0.6050  0.9313  0.0396 0.0439
p:<b16>...<b 9>| 0.7738  0.7287  0.4980  0.8498  0.1213  0.1197  0.2624 0.7645
p:<b 8>...<b 1>| 0.8724  0.6029  0.1716  0.7592  0.6087  0.5081  0.5209 0.4568
#####Gorilla ## The test is completes. Overall p is 0.3076 #####


@@@@@ Gorilla test:  p is 0.3076
```

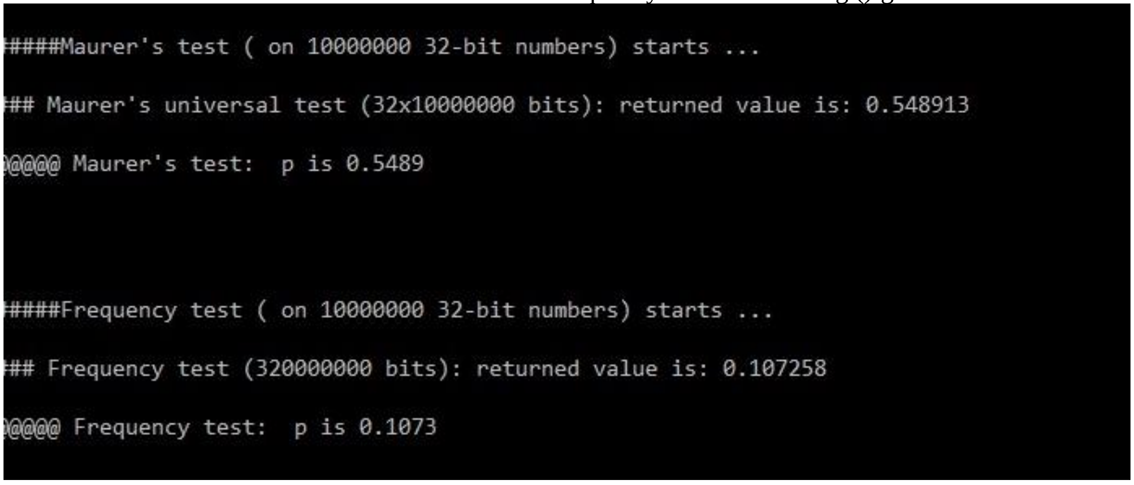


```
#####Collision test (No. of holes = 2^24; No. of balls = 2^24) starts ...
p:<b32>...<b25>| 0.9103  0.2011  0.4419  0.4839  0.8147  0.5908  0.0930 0.7088
p:<b24>...<b17>| 0.9226  0.1580  0.9174  0.2389  0.1469  0.1036  0.2484 0.3592
p:<b16>...<b 9>| 0.9515  0.4481  0.3266  0.3742  0.5589  0.1876  0.4782 0.7625
p:<b 8>...<b 1>| 0.3969  0.2641  0.6472  0.5518  0.5332  0.8117  0.3394 0.1983
#####Collision ## The test is completes. Overall p is 0.4137 #####


@@@@@ Collision test:  p is 0.4137


Process returned 0 (0x0)   execution time : 17930.604 s
Press any key to continue.
```

As can be seen in Tables 4 and 5, the values obtained using digital root pass in all tests, having a worse result in Birthday Spacings. The best results were obtained in the Frequency and Gorilla tests, the first differing from the initial frequency test because it uses the chi-square and zeros and ones for their calculations, and the second one being known since many generators do not pass this test.

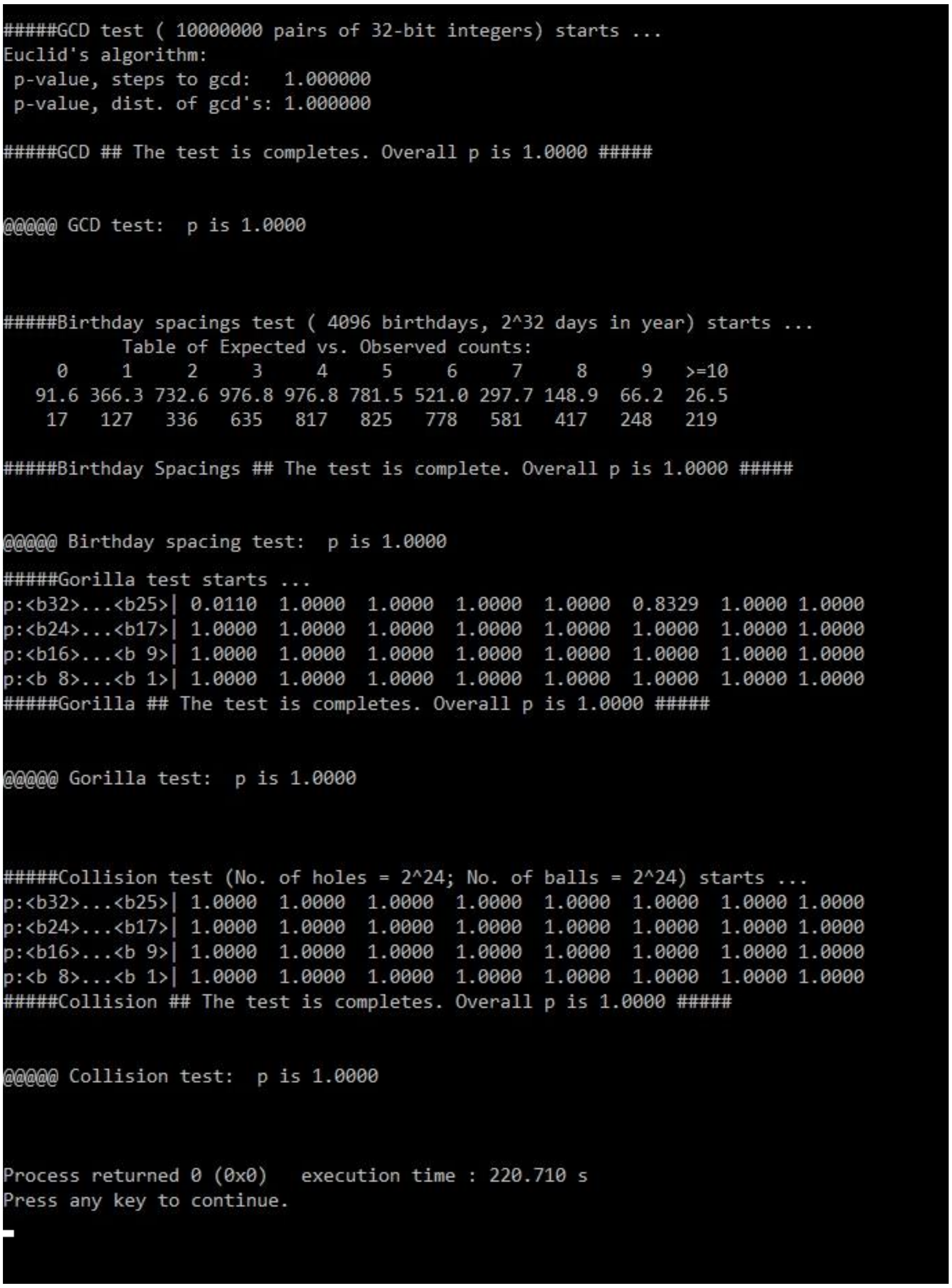
Then it was tested if the values obtained from digital root improve bad random number generators, in this case cong () and fib (), previously defined. The results for cong () are presented in Tables 6 and 7, and for fib () in Tables 8 and 9.

Table 6 - Results of the Maurer and Frequency tests for the cong () generator



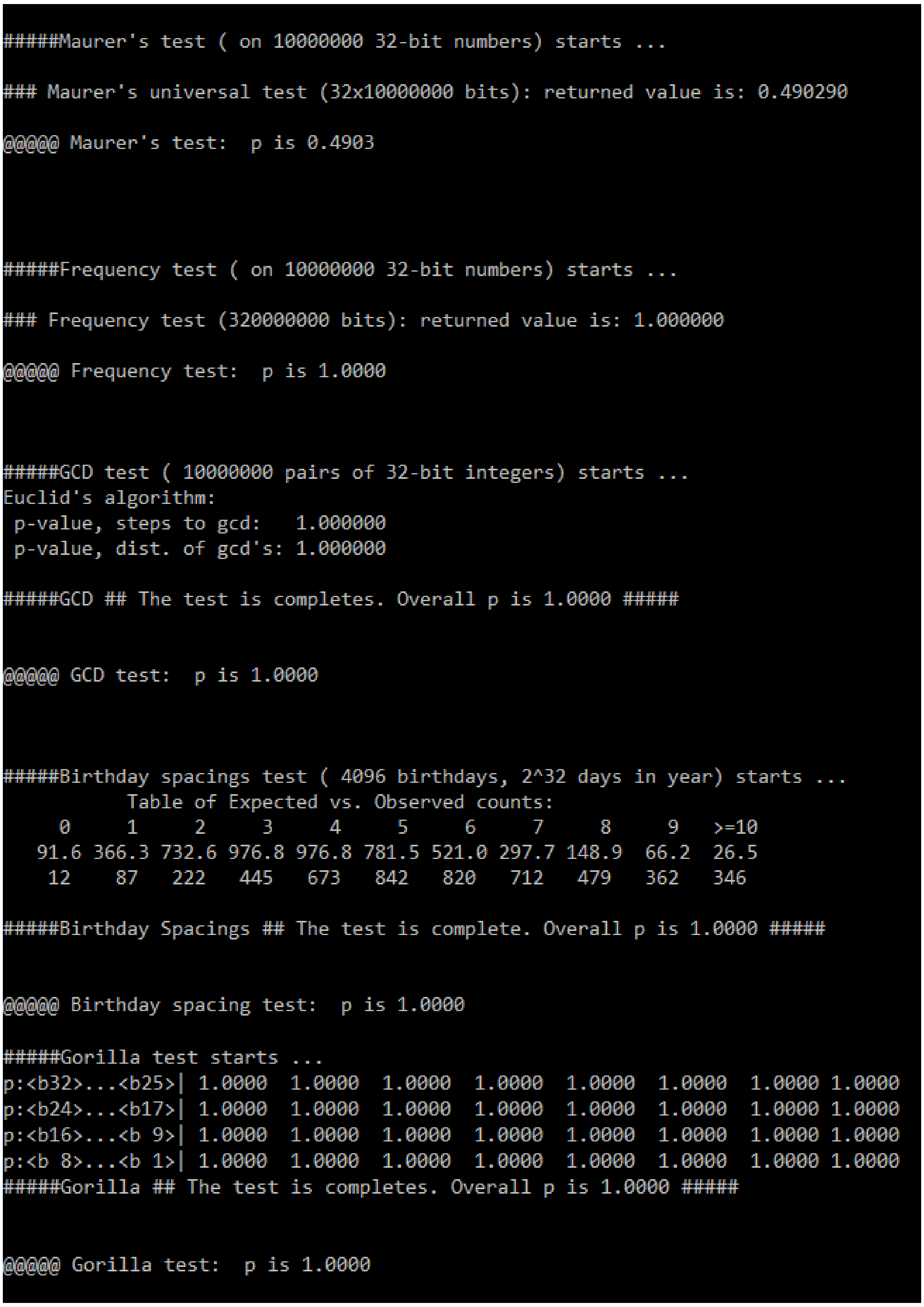
```
####Maurer's test ( on 10000000 32-bit numbers) starts ...

## Maurer's universal test (32x10000000 bits): returned value is: 0.548913

@@@@ Maurer's test:  p is 0.5489



####Frequency test ( on 10000000 32-bit numbers) starts ...

## Frequency test (320000000 bits): returned value is: 0.107258

@@@@ Frequency test:  p is 0.1073
```

Table 7 - Results of the GCD, Birthday Spacings, Gorilla and Collision tests for the cong() generator



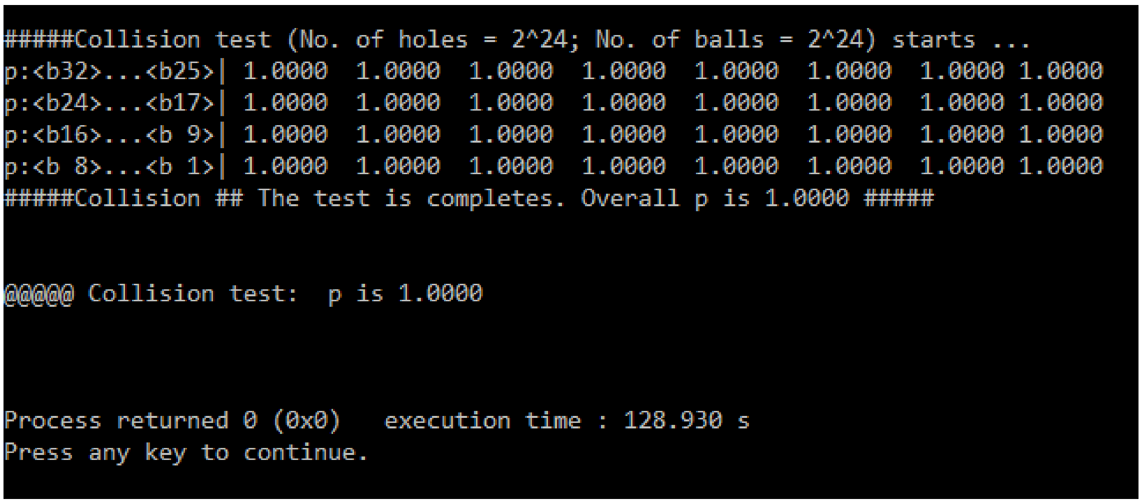
```
#####GCD test ( 10000000 pairs of 32-bit integers) starts ...
Euclid's algorithm:
 p-value, steps to gcd:   1.000000
 p-value, dist. of gcd's: 1.000000

#####GCD ## The test is completes. Overall p is 1.0000 #####


@@@@@ GCD test:  p is 1.0000



#####Birthday spacings test ( 4096 birthdays, 2^32 days in year) starts ...
           Table of Expected vs. Observed counts:
     0     1     2     3     4     5     6     7     8     9   >=10
   91.6 366.3 732.6 976.8 976.8 781.5 521.0 297.7 148.9  66.2  26.5
    17   127   336   635   817   825   778   581   417   248   219

#####Birthday Spacings ## The test is complete. Overall p is 1.0000 #####


@@@@@ Birthday spacing test:  p is 1.0000

#####Gorilla test starts ...
p:<b32>...<b25>| 0.0110  1.0000  1.0000  1.0000  1.0000  0.8329  1.0000 1.0000
p:<b24>...<b17>| 1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000 1.0000
p:<b16>...<b 9>| 1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000 1.0000
p:<b 8>...<b 1>| 1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000 1.0000
#####Gorilla ## The test is completes. Overall p is 1.0000 #####


@@@@@ Gorilla test:  p is 1.0000



#####Collision test (No. of holes = 2^24; No. of balls = 2^24) starts ...
p:<b32>...<b25>| 1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000 1.0000
p:<b24>...<b17>| 1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000 1.0000
p:<b16>...<b 9>| 1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000 1.0000
p:<b 8>...<b 1>| 1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000 1.0000
#####Collision ## The test is completes. Overall p is 1.0000 #####


@@@@@ Collision test:  p is 1.0000



Process returned 0 (0x0)   execution time : 220.710 s
Press any key to continue.
```

Table 8 - Results of the Maurer, Frequency, GCD, Birthday Spacings and Gorilla tests for the fib () generator



```
#####Maurer's test ( on 10000000 32-bit numbers) starts ...

### Maurer's universal test (32x10000000 bits): returned value is: 0.490290

@@@@@ Maurer's test:  p is 0.4903



#####Frequency test ( on 10000000 32-bit numbers) starts ...

### Frequency test (320000000 bits): returned value is: 1.000000

@@@@@ Frequency test:  p is 1.0000


#####GCD test ( 10000000 pairs of 32-bit integers) starts ...
Euclid's algorithm:
 p-value, steps to gcd:   1.000000
 p-value, dist. of gcd's: 1.000000

#####GCD ## The test is completes. Overall p is 1.0000 #####


@@@@@ GCD test:  p is 1.0000



#####Birthday spacings test ( 4096 birthdays, 2^32 days in year) starts ...
           Table of Expected vs. Observed counts:
     0     1     2     3     4     5     6     7     8     9    >=10
   91.6 366.3 732.6 976.8 976.8 781.5 521.0 297.7 148.9  66.2  26.5
    12    87   222   445   673   842   820   712   479   362   346

#####Birthday Spacings ## The test is complete. Overall p is 1.0000 #####


@@@@@ Birthday spacing test:  p is 1.0000

#####Gorilla test starts ...
p:<b32>...<b25>| 1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000 1.0000
p:<b24>...<b17>| 1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000 1.0000
p:<b16>...<b 9>| 1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000 1.0000
p:<b 8>...<b 1>| 1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000 1.0000
#####Gorilla ## The test is completes. Overall p is 1.0000 #####


@@@@@ Gorilla test:  p is 1.0000
```

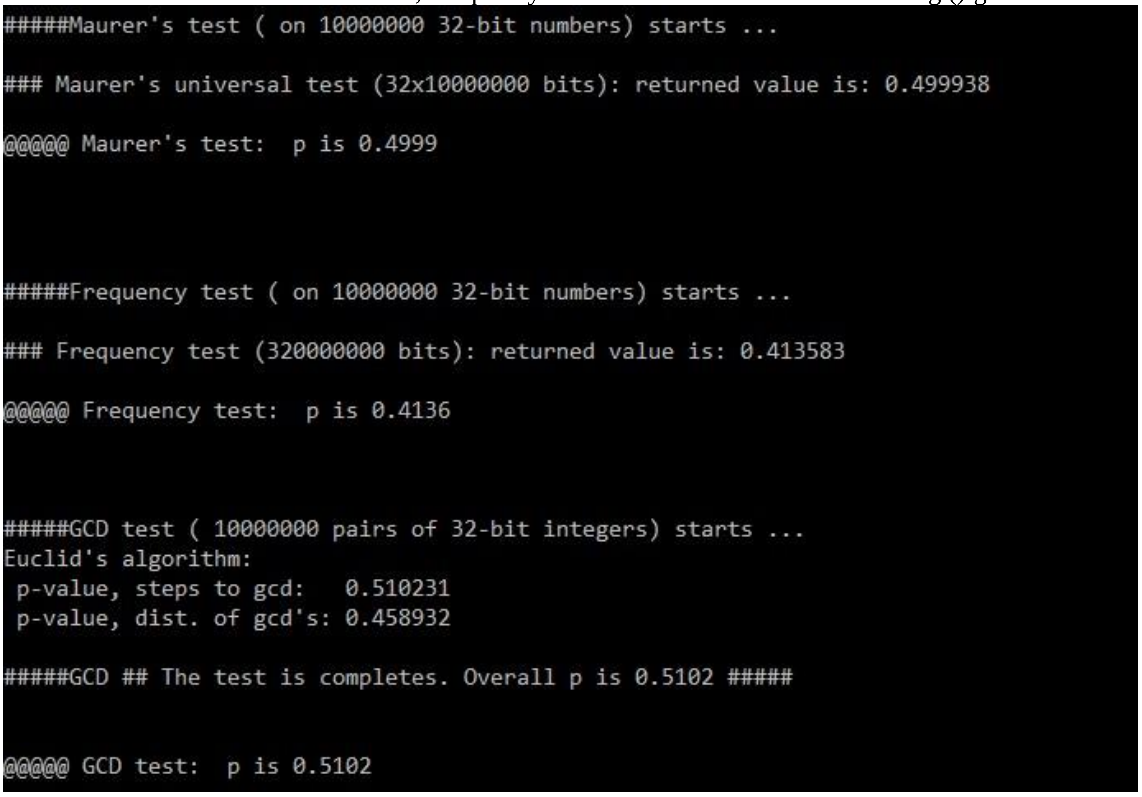
Table 9 - Collision test for the fib () generator



```
#####Collision test (No. of holes = 2^24; No. of balls = 2^24) starts ...
p:<b32>...<b25>| 1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000 1.0000
p:<b24>...<b17>| 1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000 1.0000
p:<b16>...<b 9>| 1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000 1.0000
p:<b 8>...<b 1>| 1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000 1.0000
#####Collision ## The test is completes. Overall p is 1.0000 #####


@@@@@ Collision test:  p is 1.0000



Process returned 0 (0x0)   execution time : 128.930 s
Press any key to continue.
```

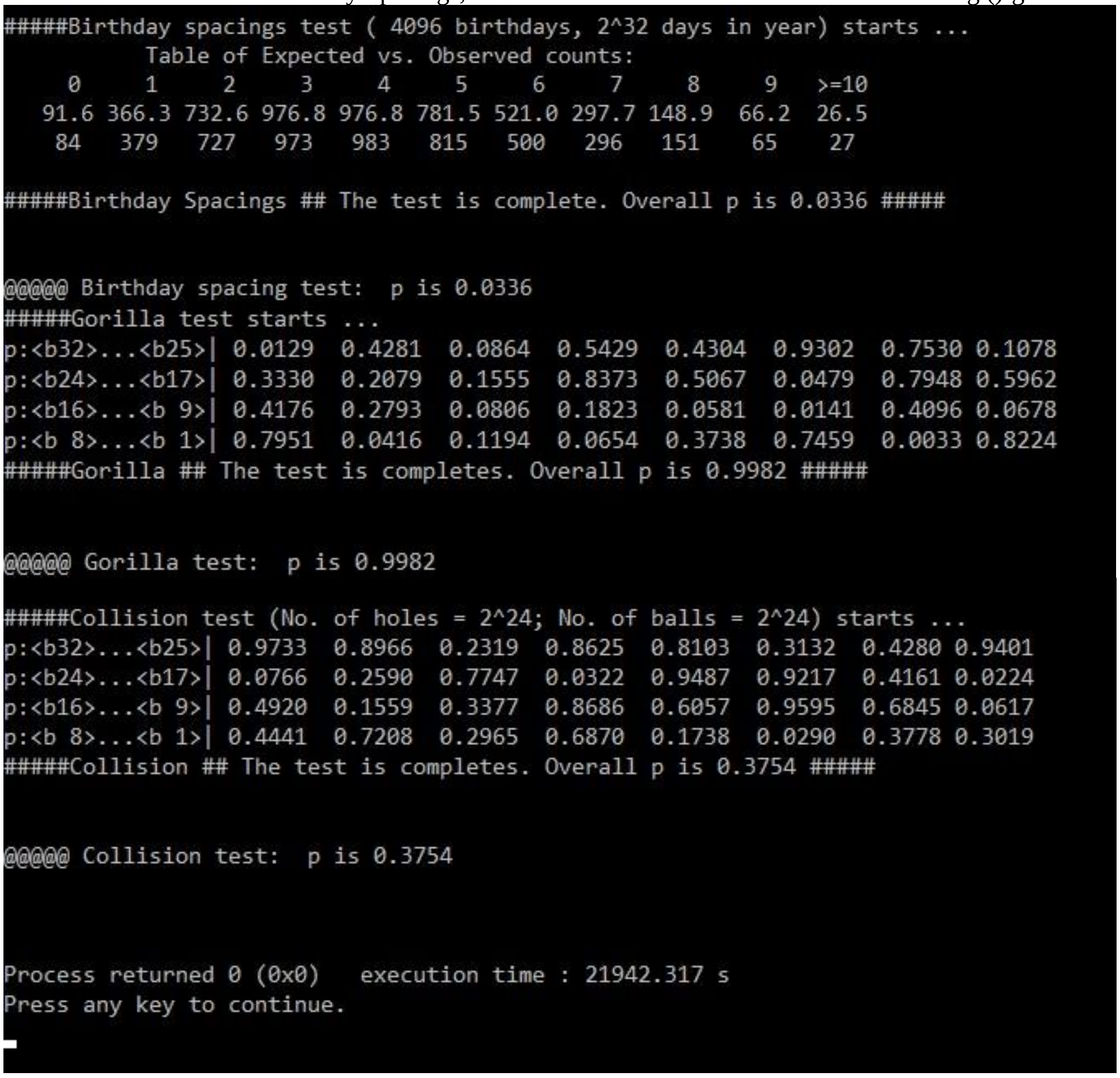
It is observed that the generator cong () only passes in the tests Frequency and Maurer, the latter being the easiest to have a positive result, being that the generator fib () only passes this test. This demonstrates that these generators are bad enough not to pass these tests, different from the generator developed by digital root, which passes in all. Next, the results of the cong () and fib () tests are presented from the algorithm that uses the digital root transformed into 3 bits, in Tables 10, 11, 12 and 13.

Table 10 - Results of the Maurer, Frequency and GCD tests for the modified cong () generator



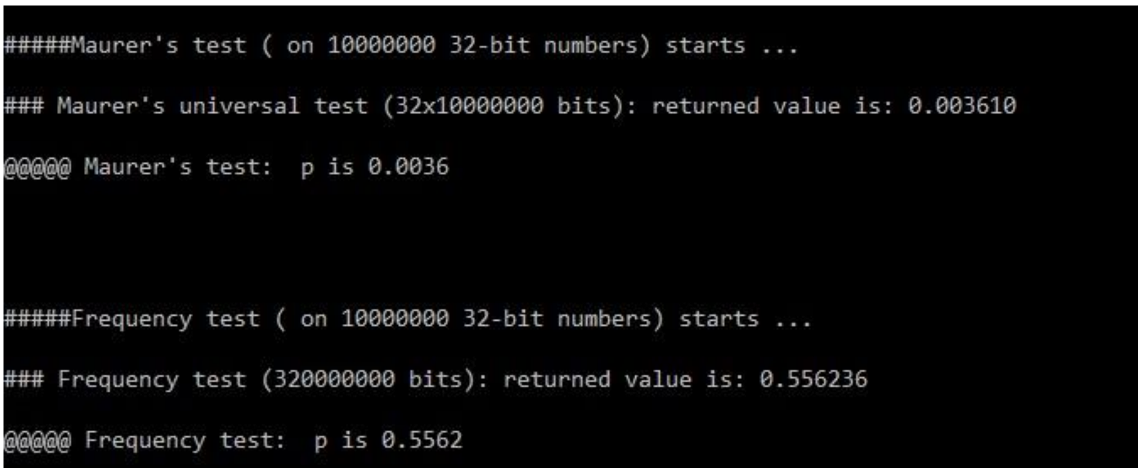
```
#####Maurer's test ( on 10000000 32-bit numbers) starts ...

### Maurer's universal test (32x10000000 bits): returned value is: 0.499938

@@@@@ Maurer's test:  p is 0.4999



#####Frequency test ( on 10000000 32-bit numbers) starts ...

### Frequency test (320000000 bits): returned value is: 0.413583

@@@@@ Frequency test:  p is 0.4136


#####GCD test ( 10000000 pairs of 32-bit integers) starts ...
Euclid's algorithm:
 p-value, steps to gcd:   0.510231
 p-value, dist. of gcd's: 0.458932

#####GCD ## The test is completes. Overall p is 0.5102 #####


@@@@@ GCD test:  p is 0.5102
```

Table 11 -Results of the Birthday Spacings, Gorilla and Collision tests for the modified cong () generator



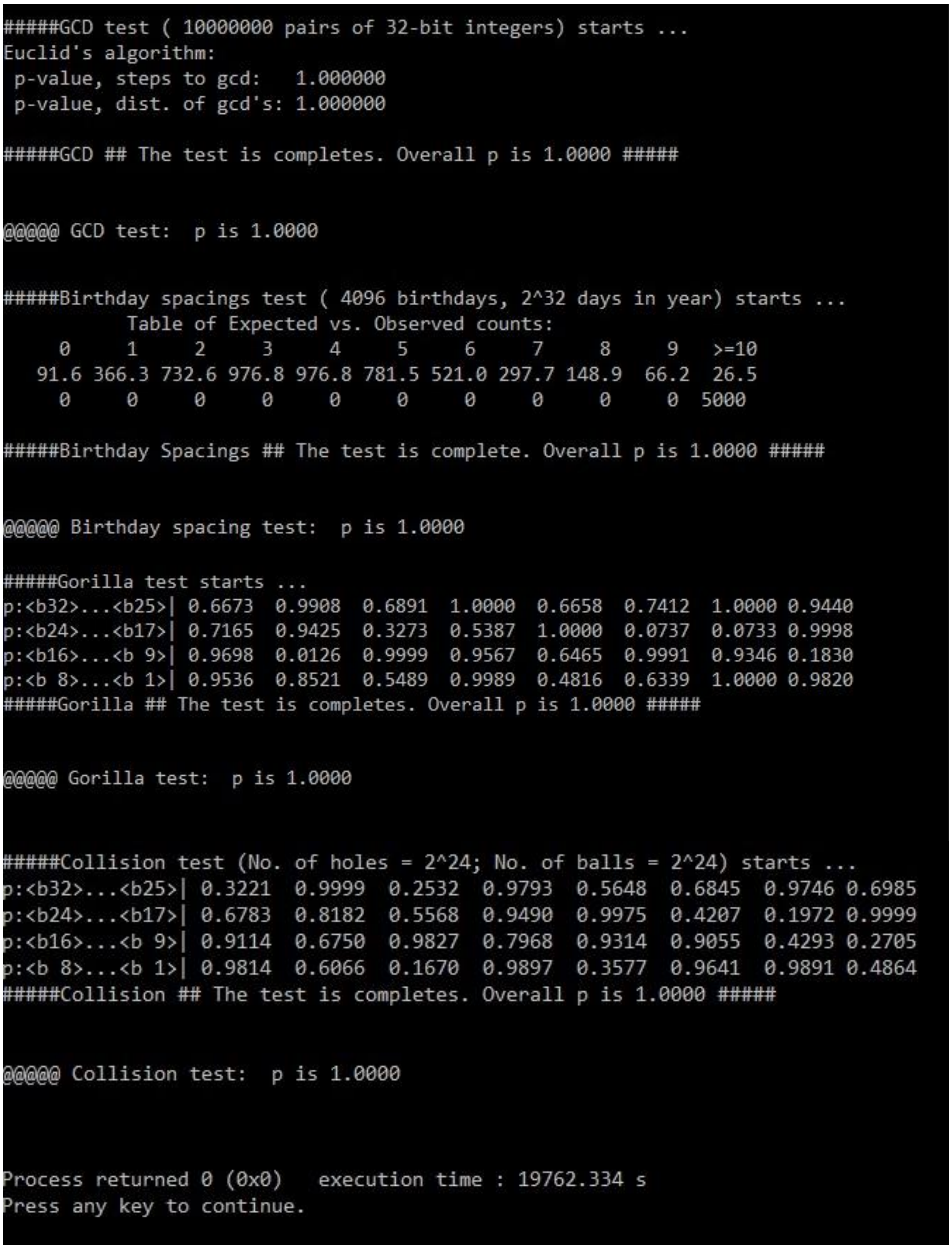
```
#####Birthday spacings test ( 4096 birthdays, 2^32 days in year) starts ...
          Table of Expected vs. Observed counts:
     0     1     2     3     4     5     6     7     8     9   >=10
  91.6 366.3 732.6 976.8 976.8 781.5 521.0 297.7 148.9  66.2  26.5
    84   379   727   973   983   815   500   296   151    65    27

#####Birthday Spacings ## The test is complete. Overall p is 0.0336 #####


@@@@@ Birthday spacing test:  p is 0.0336
#####Gorilla test starts ...
p:<b32>...<b25>| 0.0129  0.4281  0.0864  0.5429  0.4304  0.9302  0.7530 0.1078
p:<b24>...<b17>| 0.3330  0.2079  0.1555  0.8373  0.5067  0.0479  0.7948 0.5962
p:<b16>...<b 9>| 0.4176  0.2793  0.0806  0.1823  0.0581  0.0141  0.4096 0.0678
p:<b 8>...<b 1>| 0.7951  0.0416  0.1194  0.0654  0.3738  0.7459  0.0033 0.8224
#####Gorilla ## The test is completes. Overall p is 0.9982 #####


@@@@@ Gorilla test:  p is 0.9982

#####Collision test (No. of holes = 2^24; No. of balls = 2^24) starts ...
p:<b32>...<b25>| 0.9733  0.8966  0.2319  0.8625  0.8103  0.3132  0.4280 0.9401
p:<b24>...<b17>| 0.0766  0.2590  0.7747  0.0322  0.9487  0.9217  0.4161 0.0224
p:<b16>...<b 9>| 0.4920  0.1559  0.3377  0.8686  0.6057  0.9595  0.6845 0.0617
p:<b 8>...<b 1>| 0.4441  0.7208  0.2965  0.6870  0.1738  0.0290  0.3778 0.3019
#####Collision ## The test is completes. Overall p is 0.3754 #####


@@@@@ Collision test:  p is 0.3754



Process returned 0 (0x0)   execution time : 21942.317 s
Press any key to continue.
```

Table 12 - Results of the Maurer and Frequency tests for the modified fib () generator



```
#####Maurer's test ( on 10000000 32-bit numbers) starts ...

### Maurer's universal test (32x10000000 bits): returned value is: 0.003610

@@@@@ Maurer's test:  p is 0.0036




#####Frequency test ( on 10000000 32-bit numbers) starts ...

### Frequency test (320000000 bits): returned value is: 0.556236

@@@@@ Frequency test:  p is 0.5562
```

Table 13 - Results of the GCD, Birthday Spacings, Gorilla and Collision tests for the modified fib () generator



```
#####GCD test ( 10000000 pairs of 32-bit integers) starts ...
Euclid's algorithm:
 p-value, steps to gcd:   1.000000
 p-value, dist. of gcd's: 1.000000

#####GCD ## The test is completes. Overall p is 1.0000 #####


@@@@@ GCD test:  p is 1.0000


#####Birthday spacings test ( 4096 birthdays, 2^32 days in year) starts ...
          Table of Expected vs. Observed counts:
     0     1     2     3     4     5     6     7     8     9   >=10
   91.6 366.3 732.6 976.8 976.8 781.5 521.0 297.7 148.9  66.2  26.5
     0     0     0     0     0     0     0     0     0     0  5000

#####Birthday Spacings ## The test is complete. Overall p is 1.0000 #####


@@@@@ Birthday spacing test:  p is 1.0000

#####Gorilla test starts ...
p:<b32>...<b25>| 0.6673  0.9908  0.6891  1.0000  0.6658  0.7412  1.0000 0.9440
p:<b24>...<b17>| 0.7165  0.9425  0.3273  0.5387  1.0000  0.0737  0.0733 0.9998
p:<b16>...<b 9>| 0.9698  0.0126  0.9999  0.9567  0.6465  0.9991  0.9346 0.1830
p:<b 8>...<b 1>| 0.9536  0.8521  0.5489  0.9989  0.4816  0.6339  1.0000 0.9820
#####Gorilla ## The test is completes. Overall p is 1.0000 #####


@@@@@ Gorilla test:  p is 1.0000


#####Collision test (No. of holes = 2^24; No. of balls = 2^24) starts ...
p:<b32>...<b25>| 0.3221  0.9999  0.2532  0.9793  0.5648  0.6845  0.9746 0.6985
p:<b24>...<b17>| 0.6783  0.8182  0.5568  0.9490  0.9975  0.4207  0.1972 0.9999
p:<b16>...<b 9>| 0.9114  0.6750  0.9827  0.7968  0.9314  0.9055  0.4293 0.2705
p:<b 8>...<b 1>| 0.9814  0.6066  0.1670  0.9897  0.3577  0.9641  0.9891 0.4864
#####Collision ## The test is completes. Overall p is 1.0000 #####


@@@@@ Collision test:  p is 1.0000



Process returned 0 (0x0)   execution time : 19762.334 s
Press any key to continue.
```

The results obtained demonstrate the viability of the digital root for random number generation, even if the speed has not been evaluated in this case.

6. Final considerations

The results obtained in the tests for generators of random sequences were satisfactory for the method developed from digital root. However, it is important to note that speed tests have not been performed, which can be done in future work.

Collision and Gorilla tests, which take longer to run completely, ranging from 3 minutes to 5 hours in total, are the most complex tests and are the most difficult to apply. The algorithm developed based on the digital root was able to pass in these tests, which demonstrates an excellent performance, compared to the weak generators, fib () and cong (), that were improved by the use of the digital root algorithm with 32 bits developed.

Finally, it is important to note that the tests used do not prove the total usability of the digital root, since a number of its limits were removed (9), which results in a subset of the digital root, ranging from 1 to 8. In addition, this test battery does not prove usability in cryptography, for example, it needs its own generators with more restricted dependencies than those included here.

7. References

APRAJITA; KUMAR, A. **Digital Roots and Their Properties.** 2015. Disponível em: <https://www.academia.edu/10228439/Digital_Roots_and_their_Properties>. Acesso em: 24 set. 2017.

AVERBACH, B.; CHEIN, O. **Problem Solving Through Recreational Mathematics.** P. 125, 2000. Disponível em: <https://books.google.com.br/books?id=qtMoAwAAQBAJ&pg=PA125&redir_esc=y#v=onepage&q&f=false>. Acesso em: 7 abr. 2017.

BARR, M. **CRC Series, Part 1: Additive Checksums**. 1999. Disponível em: <https://barrgroup.com/Embedded-Systems/How-To/Additive-Checksums>. Acesso em: 15 abr. 2017.

CASQUILHO, M. **Simulação: geração de números pseudo-aleatórios.** 2008. Disponível em: <http://web.tecnico.ulisboa.pt/mcasquilho/acad/or/simul/GerNumAleat.pdf>. Acesso em: 26 set. 2017.

INSTRUCTABLES. **Mental Math: Digital Root Extraction.** 2008. Disponível em: <http://www.instructables.com/id/Mental-Math-Digital-Root-Extraction/>. Acesso em: 15 abr. 2017.

IZMIRLI, I. **On Some Propertiesof Digital Roots.** 2014. Disponível em: <http://file.scirp.org/pdf/APM_2014062516180597.pdf>. Acesso em: 07 set. 2017.

KATZ, J.; LINDELL, Y. **Introduction To Modern Cryptography.** 2008, pg. 69-70. Disponível em:<https://murdercube.com/files/Computers/Computer%20Security/Introduction%20to%20Modern%20Cryptography.pdf>. Acesso em: 25 nov. 2017.

MARSAGLIA, G.**Some Difficult-to-pass Tests of Randomness.** 2002. Disponível em:<https://www.jstatsoft.org/article/view/v007i03/tuftests.pdf>. Acesso em: 15 nov. 2017.

TSANG, W. **CISC Library of Tests for Random Number Generators.** 2002. Disponível em: <http://www.csis.hku.hk/cisc/download/idetect/>. Acesso em: 14 nov. 2017

TSANG, W. **Collision Test.** 2002. Disponível em: <http://i.cs.hku.hk/~cisc/projects/va/details/collision.html>. Acesso em: 20 nov. 2017.

TSANG, W. **Frequency Test.** 2002. Disponível em: <http://i.cs.hku.hk/cisc/projects/va/details/freq_monobit.html>. Acesso em: 20 nov. 2017.

TSANG, W. **Gorilla Test.** 2002. Disponível em: <http://i.cs.hku.hk/cisc/projects/va/details/gorilla.html>. Acesso em: 20 nov. 2017.

TSANG, W. **Maurer's "Universal Statistical" Test.** 2002. Disponível em: <http://i.cs.hku.hk/cisc/projects/va/details/universal.html>. Acesso em: 20 nov. 2017.

TSANG, W; HUI, L.; CHOW, K; CHONG, C. **Tuning the Collision Test for Stringency.** 1999. Disponível em <http://i.cs.hku.hk/cisc/projects/va/tuning.pdf>. Acesso em: 15 nov. 2017.

UNIVERSIDADE FEDERAL FLUMINENSE. **Como os computadores geram números aleatórios?** 2010. Disponível em: <http://www.uff.br/cdme/rdf/rdf-html/rdf-g-br.html>. Acesso em: 24 set. 2017.