A Study of Some Important Algorithms Used in the Process of Generating Random Numbers

Sawsan Rateb almokabaa[1,*], Maissam Ahamad Jdid[2]

[1]Master's Student - Faculty of Science, Damascus University – Damascus, Syria

[2]Faculty of Science, Damascus University, Damascus, Syria

Email: sawsan.almokabaa@damascusuniversity.edu.sy; maissam.jdid66@damascusuniversity.edu.sy

Abstract

The efforts of many researchers and scholars have focused on providing appropriate algorithms for generating random numbers and developing them in a manner that suits the need for them, but these algorithms still have advantages and disadvantages, so they are suitable for a specific study and not suitable for another study. The reason for the interest of researchers and scholars in the process of generating random numbers is that random numbers have many scientific and technical applications, starting with generating a series of semi-random numbers, starting from computer simulation to encryption, games of chance, and random samples for statistics and security. In simulation, which is one of the important methods provided by the new science of operations research, the primary reliance is on generating a series of random numbers that follow the regular distribution in the range [0,1], and then converting these random numbers into random variables that follow the probability distribution according to which the system to be simulated works, as the accuracy of the results we obtain from the simulation process depends on the numbers we generate using one of the algorithms. In other words, the appropriate algorithm for the field of study must be chosen from among the algorithms used, which prompted us to prepare this research, through which we will present a reference study of some of the algorithms used to generate random numbers. Where we will highlight the advantages and disadvantages of these algorithms and the most important areas of their use. Then we will calculate the number of these algorithms and compare them. The algorithms that we will discuss in this research are:

- Middle Square Method.
- Middle multi-Method.
- Fibonacci Methods.
- Linear congruential Methods.

Keywords: Random number generation; Center square method; Center product method; Fibonacci series; Linear congruence method; Random number series

1. Introduction

After reviewing a number of references that were interested in presenting algorithms for generating random numbers such as [1-17], we reached the following:

Random numbers are considered essential and important elements in a wide range of scientific and technical applications, ranging from computer simulation to encryption, games of chance, and random samples. It is worth noting that random numbers have been used for thousands of years and the concept of random numbers is not new. From the lottery in ancient Babylon, to the roulette tables in Monte Carlo, to the dice games in Vegas, the goal has always been to leave the outcome to random chance. Regardless of gambling, randomness has many uses in science, statistics, cryptography, and more. However, the use of dice, coins, or similar media as a random device has its limitations due to their mechanical nature. Generating large amounts of random numbers requires a great deal of time and work. Thanks to the efforts of researchers and scholars in this field, we have more powerful algorithms, each of which relies on a specific set of mathematical operations to achieve randomness. These

DOI: https://doi.org/10.54216/PAMDA.040102

Received: January 23, 2024 Revised: March 19, 2024 Accepted: June 28, 2024

algorithms range from the very simple, such as the “mean square method,” which relies on squaring numbers and extracting the middle values, to more complex algorithms such as “linear matching,” which relies on mathematical functions. There are other methods such as “mean product” and “Fibonacci series,” which rely on the principles of addition and multiplication in numerical series to produce seemingly unpredictable random numbers. As we mentioned earlier, each of these methods has advantages and disadvantages, and they differ in their ability to generate long, non-repeating sequences of random numbers. This research aims to analyze and study some algorithms, focusing on how they work. Before starting to present these algorithms, we must mention the types of random numbers that we can obtain. We have

True random numbers: which depend on natural phenomena such as electrical noise or particle movement and provide numbers with high randomness but are difficult to obtain and measure, and pseudo-random numbers that are produced by mathematical algorithms that use initial values? (Seeds) to generate sequences of numbers that appear random. Although these numbers are not completely random, they are sufficient for many applications, but they must have the following properties:

- ✓ The generated random numbers must follow a regular distribution in the range [0,1].
- ✓ The cycle of random numbers must be statistically independent.
- ✓ The cycle of random numbers must be long.
- ✓ The speed of the random number generation process.

2. Discussion

In this research, we focus on presenting the following methods:

- ➢ Middle Square Method.
- ➢ Middle multi-Method.
- ➢ Fibonacci Methods.
- ➢ Linear congruential Methods.

In terms of advantages, disadvantages, improvements, and most important areas of use:

➢ **Middle Square Method:** $[1-6]$

It is a simple and easy algorithm for generating random numbers introduced by the scientist John von Neumann in 1949.

The following relation gives the Middle Square Method:

$$R_{i+1} = Mid[R_i^2]; i = 0,1,2,3 \dots$$

Where Mid denotes the middle numbers of the squaring result

We choose R_i as a random fractional number consisting of four digits and does not contain zero in any of the four digits. In this case, Mid refers to the middle four digits of R_i^2.

We summarize this method in the following steps:

- ✓ Choose an initial value (seed).
- ✓ Square the initial value.
- ✓ Extract the middle numbers.

Note that the number of numbers must be equal to the length of the initial number. If the initial number consists of 6 numbers, we extract the six middle numbers.

We get the new random number (the extracted middle numbers represent the new random number).

We repeat the process until we get the number required for the simulation process.

We illustrate the above through the following example:

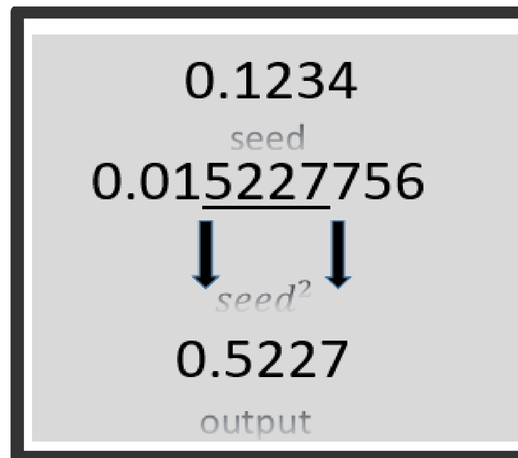
Example1:

- ✓ Let us take $seed = R_0 = 0.1234$ which is the initial value.
- ✓ Square the initial value $seed = R_0 = 0.1234$

$$(R_0)(0.1234)^2 = 0.01522756$$

- ✓ We take the middle four numbers from the result of the square and we get the new random number which is $R_1 = 0.5227$.

DOI: https://doi.org/10.54216/PAMDA.040102
Received: January 23, 2024 Revised: March 19, 2024 Accepted: June 28, 2024



- ✓ We repeat the work using 0.5227 and repeat the same steps:

$$(0.5227)^2 = 0.27321529$$

The middle four numbers are $R_2 = 0.3215$.

We continue working until we get enough random numbers needed for the simulation process (the series of random numbers $R_1, R_2, \ldots$

❖ **Features of the middle square method:**

- ✓ Easy to implement: The center square method is one of the simplest methods for generating random numbers. It does not require complex equations or advanced software, making it easy to understand and implement even for beginners.
- ✓ Calculation speed: Due to its simplicity, random numbers can be calculated quickly.
- ✓ The basic operations it requires (squaring and taking the middle numbers) can be performed very quickly even on devices with limited computing capabilities.
- ✓ Ease of programming: The middle square method can be easily programmed in most programming languages. This makes it a good choice for teaching and experimentation when the goal is to understand the basics behind random number generators.

❖ **Disadvantages of the middle square method:**

- ✓ Produces short cycles: The middle square method produces short cycles that can be repeated quickly. Once the string reaches a certain number, it can enter a closed loop and the numbers start repeating periodically. This reduces the quality of the randomness and makes the generated numbers unusable in applications that require long sequences of random numbers.

Example2:

That shows that the middle square method gives short cycles. Let us take $seed = R_0 = 0.2499$ as the initial value.

$$seed = R_0 = 0.2499$$

$$R_1 = Mid[(R_0)^2]$$

$$R_1 = Mid[(0.2499)^2]$$

$$R_1 = 0.2450$$

$$R_2 = Mid[(R_1)^2]$$

$$R_2 = Mid[(0.2450)^2]$$

$$R_2 = 0.0025$$

$$R_3 = Mid[(R_2)^2]$$

$$R_3 = Mid[(0.0025)^2]$$

$$R_3 = 0.0000$$

$$R_0 = 0.2499, R1 = 0.2450, R2 = 0.0025, R3 = 0.0000$$

We notice that we entered a very short cycle where the numbers were repeated very quickly.

DOI: https://doi.org/10.54216/PAMDA.040102
Received: January 23, 2024 Revised: March 19, 2024 Accepted: June 28, 2024

- ✓ Convergence to zero**:** Sometimes, using the middle square method may cause the series to converge to zero. For example, if the middle number has too many zeros at the ends, continued squaring may cause the generated numbers to vanish towards zero, causing efficient random numbers to cease to be produced.
- ✓ Lack of statistical randomness**:** Numbers generated using the middle square method may not perform well on statistical tests of randomness. This means that the numbers generated may exhibit a non-random pattern or distribution, which reduces its usefulness in applications that require high-quality random numbers.
- ✓ Computational dependencies: The method relies heavily on precise numerical calculations. A small error in squaring or extracting the middle numbers can lead to incorrect results. This makes it unstable for applications that require high accuracy.
- ✓ Slow technology: Many multiplications and divisions are required to arrive at the middle numbers in a binary computer with a fixed process. Despite its simplicity and ease of implementation, the mean square method is not suitable for modern applications that require high levels of randomness and quality. It can be used for educational or experimental purposes, but it is not suitable for sensitive uses such as cryptography or scientific simulations.

❖ **Improving the middle square method:** [7,8]

The middle square method is one of the classical algorithms for generating pseudo-random numbers, but it suffers from fundamental drawbacks such as short cycle length and tendency to decay towards zero, which limits its effectiveness in modern applications. To address these limitations, research trends have emerged to combine it with other mathematical techniques to enhance its randomness and stability, most notably the use of Weyl Sequence and Logistic Map.

❖ **Weyl Sequence: [7]**

Weyl sequences are a powerful mathematical mechanism for improving the performance of random number generators, especially when used with algorithms such as the middle square method proposed by John von Neumann. Weyl sequences act as simple linear addition used to increase randomness and avoid redundancy in random number generators.

This mechanism depends on adding a constant S to a variable Wat each step according to the following relationship:

$$W += S$$

Where:

W: Is a 64-bit variable updated at each step.

S : An unsigned 64-bit integer that is carefully chosen to be odd and non-repeating, which contributes to an extremely long lifetime.

❖ **Implementation mechanism:**

In each step of generating random numbers, the following steps are followed:

- ✓ Squaring: The variable X (64-bit number) is squared to obtain a new value.
- ✓ Adding a Weyl sequence: The Weyl sequence $\underline{W}$ is added to the result of the squaring.
- ✓ Extracting the middle numbers: The middle parts of the result are extracted using a circular shift (32 bits).

Example3:

Squaring a 16-digit hexadecimal number like:

$$X = E3296D17IEC4A36F$$

We square this number and get the result:

$$C9927E2B2075471D31C2914AAE4E8A21$$

The middle digits are:

$$2075471D31C2914A$$

However, the computer only stores 64 bits (16 hexadecimal digits). After squaring, x stores the lower parts:

$$31C2914AAE4E8A21$$

Using a circular shift (32 bits), this part is converted to:

DOI: https://doi.org/10.54216/PAMDA.040102
Received: January 23, 2024 Revised: March 19, 2024 Accepted: June 28, 2024

AE4E8A2131C2914A

The function returns the lower parts (32 bits):

$$31C2914A$$

These are the middle digits of the original square.

❖ **Its role in improving the middle square method:**

✓ Overcoming the zero mechanism: In the traditional middle square method, if the middle numbers reach zero, the generator enters an infinite loop of zeros.
✓ Long period guarantee The Weyl sequence prevents repeating cycles, which increases the generator period to 2^{64} numbers before it repeats itself.
✓ Efficiency: The sequence implementation requires simple code (such as imulq, iaddq, rorq,) making it fast and suitable for high-performance applications.
✓ Uniformity: The Weyl sequence produces a uniform distribution of numbers, which ensures that the final generator output is uniform even if the square x is irregular.

❖ **Improving the Middle Square Method using Logistic Chaos Map [8]:**

The researchers improved the Middle Square Method (MSM) by combining it with a mathematical chaotic map (Logistic Map), which helped address some of the traditional problems of the method.

❖ **Logistic Map:**

It is a mathematical model used in studying chaotic systems, and its equation is:

$$x_{n+1} = (x_n - 1).x_n.r$$

Where:

x_n: Is the current value in the series.

r: Is the control parameter (usually chosen between 3.4 and 4 to achieve chaos).

The result is a real number between 0 and 1, and can be converted to a suitable integer.

How to combine MSM with Chaos Map: The researchers indicated that they replaced some values in MSM with values derived from the Chaos Map when the problem of repetition or short cycles appeared.

The researchers concluded that the improved method provides higher quality random numbers and can be used in cryptography and security fields.

❖ **Code: Java code to generate random numbers using the middle square method [9].**

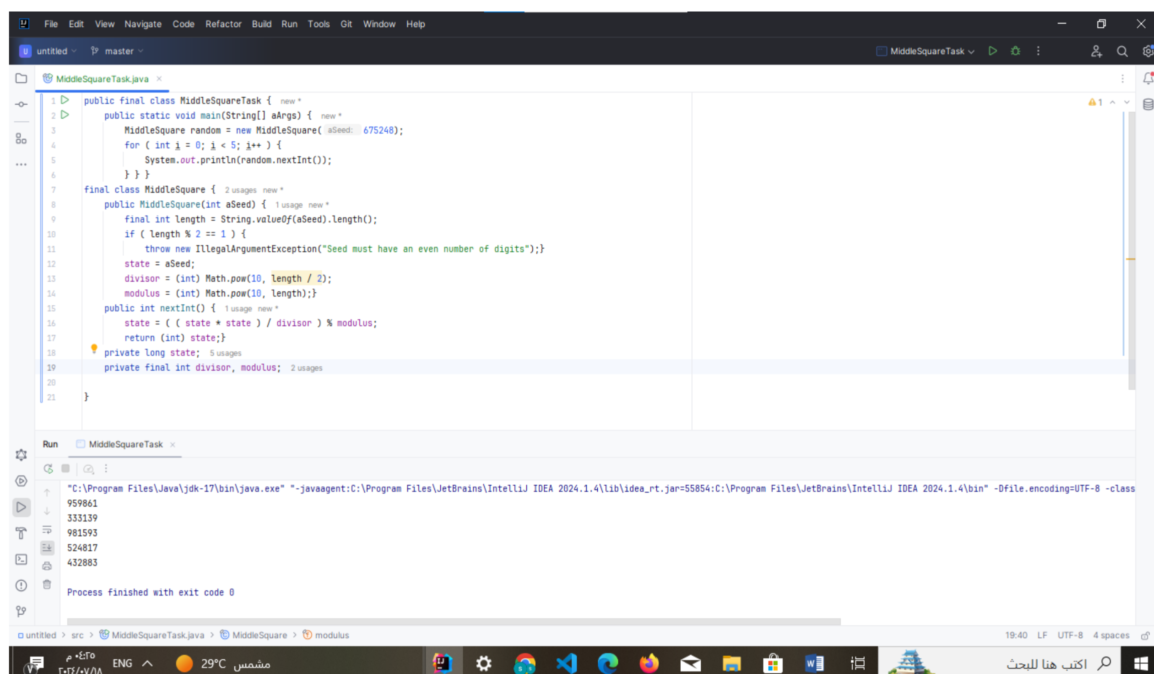
```
public final class MiddleSquareTask {
        public static void main(String [] aArgs) {
                MiddleSquare random = new MiddleSquare(675248);
                for ( int i = 0; i < 5; i++ ) {
            System.out.println(random.nextInt());
                }
        }


}
final class MiddleSquare {
        public MiddleSquare(int aSeed) {
                final int length = String.valueOf(aSeed).length();
                if ( length % 2 == 1 ) {
                        throw new IllegalArgumentException("Seed must have an even number of digits");
                }
```


DOI: https://doi.org/10.54216/PAMDA.040102
Received: January 23, 2024 Revised: March 19, 2024 Accepted: June 28, 2024

```
            state = aSeed;
            divisor = (int) Math.pow(10, length / 2);
            modulus = (int) Math.pow(10, length);
        }
        public int nextInt() {
            state = ( ( state * state ) / divisor ) % modulus;
            return (int) state;
        }
        private long state;
        private final int divisor, modulus;
}
```

❖ **Screen of executing the code to generate random numbers using the middle square method:**



❖ **Calculating the complexity of the code for the middle square method:**

✓ There is a division of the domain by 2, so the complexity is $OLog2(n)$

✓ For the function $nextInt()$

✓ Since there is a multiplication of a number by itself and it is called n times, the complexity is $O(2n)$

Therefore, the complexity of the entire program is $O(2n)$.

➢ **Middle multi-Method:**[1,3].

The middle multi-method is one of the classic methods for generating random numbers introduced by the scientist John von Neumann in 1949. The middle multi-method is a mathematical method that generates random numbers.

This method is similar to the middle square method with one difference that the new generated number is obtained from the following formula:

$$R_{i+1} = Mid[D(R_i . R_{i-1})] \quad ; i = 0,1,2, \dots$$

D represents the fractional part of the product $R_i . R_{i-1}$

We illustrate the above through the following example:

DOI: https://doi.org/10.54216/PAMDA.040102
Received: January 23, 2024 Revised: March 19, 2024 Accepted: June 28, 2024

Example4:

Let us assume that we will generate a series of random numbers using the middle multi-method

We take the two initial peaks:

$$R_0 = 1234 \quad , R_1 = 5678$$

First, we multiplying $R_0 . R_1$:

$$R_0 . R_1 = 1234 * 5678 = 7006652$$

$$R_0 . R_1 = 7006652$$

Then we take the middle numbers:

$$R_2 = 066$$

We repeat the work using R_1 and R_2:

$$R_1 . R_2 = 5678 * 066 = 374748$$

Then we take the middle numbers of the product and we get a new random number

$$R_3 = 47$$

The series we got is:

$$R_0 = 1234 , R_1 = 5678, R_2 = 066, R_3 = 47$$

We continue working until we get enough random numbers needed for the simulation process (the series of random numbers $R_1, R_2, \ldots$)

❖ **Advantages of the middle multi-method:**

- ✓ It has a longer period than the square center technique.
- ✓ The numbers are distributed more uniformly than the square center technique.
- ✓ Better than the square center method.

❖ **Disadvantages of the middle multi-method:**

- ✓ This method tends to deteriorate.
- ✓ It is not effective in distributing numbers in a balanced and random manner.
- ✓ It is less efficient in generating random numbers.

The mean product method for generating random numbers is considered weak compared to some other methods.

❖ **Code: Java code to generate random numbers using the middle multi-method:**[10]

```
public class MiddleProductRandom {
   private long seed1;
   private long seed2;
   private int digits;
   // Constructor to initialize the seeds and validate the digits
   public MiddleProductRandom(long seed1, long seed2, int digits) {
      if (String.valueOf(seed1).length() != digits || String.valueOf(seed2).length() != digits || digits % 2 != 0) {
         throw new IllegalArgumentException("Seeds must have an even number of digits matching the specified
digits.");
      }
      this.seed1 = seed1;
      this.seed2 = seed2;
      this.digits = digits;
   }
```


DOI: https://doi.org/10.54216/PAMDA.040102
Received: January 23, 2024 Revised: March 19, 2024 Accepted: June 28, 2024

```
    // Method to generate the next random number
    public long nextInt() {
        // Multiply the two seeds
        long product = seed1 * seed2;
        // Convert product to string and pad with leading zeros if necessary
        String productStr = String.format("%0" + (digits * 2) + "d", product);
        // Extract the middle digits
        int start = (productStr.length() - digits) / 2;
        String middleDigits = productStr.substring(start, start + digits);
        // Update the seeds for the next iteration
        seed1 = Long.parseLong(middleDigits);
        seed2 = (seed2 + 1) % (long) Math.pow(10, digits); // Increment seed2 for variability
        return seed1;
    }
    public static void main(String[] args) {
        // Initialize the random generator with two seeds and number of digits
        MiddleProductRandom randomGenerator = new MiddleProductRandom(1234, 5678, 4);
        // Generate and print 10 random numbers
        System.out.println("Random numbers generated using Middle Product Method:");
        for (int i = 0; i < 10; i++) {
            System.out.println(randomGenerator.nextInt());
        }
    }
}
```

❖ **Three screen execution of the code to generate random numbers using the middle multi-method**.

```
1  public class Main {
2
3      private long seed1;
4      private long seed2;
5      private int digits;
6
7      // Constructor to initialize the seeds and validate the digits
8      public Main(long seed1, long seed2, int digits) {
9          if (String.valueOf(seed1).length() != digits || String.valueOf(seed2).length() != digits || digits % 2 != 0) {
10             throw new IllegalArgumentException("Seeds must have an even number of digits matching the specified digits.");}
11         this.seed1 = seed1;
12         this.seed2 = seed2;
13         this.digits = digits;}
14
15     // Method to generate the next random number
16     public long nextInt() {
17         // Multiply the two seeds
18         long product = seed1 * seed2;
19
20         // Convert product to string and pad with leading zeros if necessary
21         String productStr = String.format("%0" + (digits * 2) + "d", product);
22
23         // Extract the middle digits
24         int start = (productStr.length() - digits) / 2;
25         String middleDigits = productStr.substring(start, start + digits);
26
27         // Update the seeds for the next iteration
28         seed1 = Long.parseLong(middleDigits);
29         seed2 = (seed2 + 1) % (long) Math.pow(10, digits); // Increment seed2 for variability
30
31         return seed1;
32     }
33
34     public static void main(String[] args) {
35         // Initialize the random generator with two seeds and number of digits
```


DOI: https://doi.org/10.54216/PAMDA.040102
Received: January 23, 2024 Revised: March 19, 2024 Accepted: June 28, 2024

```
35      // Initialize the random generator with two seeds and number of digits
36      Main randomGenerator = new Main(1234, 5678, 4);
37
38      // Generate and print 10 random numbers
39      System.out.println("Random numbers generated using Middle Product Method:");
40      for (int i = 0; i < 10; i++) {
41          System.out.println(randomGenerator.nextInt());
42      }
43    }
44  }
45
```

```
Random numbers generated using Middle Product Method:
66
3748
2886
3953
4609
1929
9644
8261
9720
2776


...Program finished with exit code 0
Press ENTER to exit console.
```

- **Calculate the code complexity of the Middle multi-Method:**

- ✓ Product $O(2n)O(n^2)O(2n)$
- ✓ Text format $O(n)O(n)O(n)$.
- ✓ Extract middle numbers $O(n)O(n)O(n)$
- ✓ Convert text to number $O(n)O(n)O(n)$
- ✓ Constant operations$O(1)O(1)O(1)$.

Total complexity per call:

$$O(2n) + O(n) + O(n) + O(n) + O(1) = O(2n)O(n^2) + O(n) + O(n) + O(n) + O(n) + O(1) = O(n^2)O(2n) + O(n) + O(n) + O(n) + O(1) = O(2n)$$

➢ **Fibonacci Methods:** [3,5]

Random numbers can also be generated from the original Fibonacci sequence starting with two numbers (usually 0 and 1) and each subsequent number being the sum of the two preceding numbers.

In the random number generation method, a similar concept is used but with modifications to meet the requirements of randomness.

"The Fibonacci series is formed when we add the last two consecutive numbers of the sequence starting with 0 and 1."

In other words, we can say that, in the Fibonacci sequence, the next number is equal to the sum of the last two numbers.

For example, 0, 1, 1, 2, 3, 5, 8, 13, 21, 34... If we look at the above series, it seems very easy to calculate in mathematics.

All you have to do is take the last two numbers of the Fibonacci numbers, add them and voila. The result is the latest number in the series.

Therefore, the next number in the Fibonacci sequence will be 21 + 34 = 55.

DOI: https://doi.org/10.54216/PAMDA.040102
Received: January 23, 2024 Revised: March 19, 2024 Accepted: June 28, 2024

- **Modified Fibonacci method:**

The Fibonacci method for generating random numbers is based on the famous Fibonacci series, but it is used in a slightly modified way to generate random numbers as follows:

$$R_{i+1} = (R_i + R_{i-1}) Modulo\ M$$

- **Features of the Fibonacci method:**

✓ Ease of implementation: It is easy to implement and understand, without the need for complex programming skills

✓ This method produces a greater role for M.

- **Disadvantages of Fibonacci method:**

✓ We notice that the Fibonacci series is easy to predict, it is easy for attackers to predict the upcoming numbers in the Fibonacci series

✓ It fails to pass randomness.

✓ It does not give appropriate results.

- **Improving the Fibonacci Series Method:**

The classical Fibonacci series method for generating random numbers faces significant challenges. To address these limitations, we used the Lagged Fibonacci Generator, which re-engineers the relationship between the sequence terms by introducing a time delay between them, which greatly increases the complexity of the sequence and lengthens its cycle.

- **Lagged Fibonacci Generator:**

A method of generating random numbers based on a modification of the classic Fibonacci sequence concept, where lagged periods are used to increase the lag of the random period and avoid repeating patterns. It is usually used with modular arithmetic operations to ensure that the numbers remain within a specified range.

Mathematical formula:

$$R_i = (R_{i-L} + R_{i-K}) Modulo\ M \quad ; \ 0 < L < K$$

Where:

L, K: lag periods (specifies the number of steps back)

M: scale (usually a power of (2) such as $(2^{32}, 2^{64})$

Example5:

$$R_i = (R_{i-24} + R_{i-55}) Modulo e^2$$

Where:

$$K = 55, L = 24, M = e^2$$

- **Its role in improving the Fibonacci series:**

The delayed Fibonacci generator is an effective tool for improving the process of generating random numbers, through several main advantages:

✓ Long period: When the delayed periods L, K are chosen appropriately, a very long period can be achieved for the random sequence. For example, if $L = 24, K = 55$ is chosen, the period can reach very large values (such as 2^{55} or more), which reduces the probability of the generated numbers being repeated.

✓ Computational efficiency: The generator relies on simple arithmetic operations such as addition and subtraction, without the need for complex multiplication operations. This makes it fast and efficient in terms of computational resources, especially in applications that require generating random numbers in large quantities.

✓ Good distribution: The generator is characterized by a homogeneous distribution of random numbers within the specified range, making it suitable for applications that require random numbers with a regular distribution.

- **Disadvantages of the Lag Fibonacci Series Method:**

DOI: https://doi.org/10.54216/PAMDA.040102
Received: January 23, 2024 Revised: March 19, 2024 Accepted: June 28, 2024

Despite the above advantages, the Lag Fibonacci Generator is not statistically perfect, as it suffers from some limitations that must be taken into account:

- ✓ Lack of statistical randomness: The sequence generated by the Lag Fibonacci Generator is not completely random statistically, which limits its use in applications that require high-quality randomness.
- ✓ Weakness in distribution tests: The generator fails some basic statistical tests, such as the gap test and the run test, indicating the presence of repetitive patterns or unwanted correlations in the resulting sequence.
- ✓ Reliance on initial values: If the initial values $R_0, R_1, \dots \dots, R_{k-1}$ are not varied or not carefully chosen, repetitive patterns may quickly appear which affects the quality of the generated numbers.

The Lag Fibonacci Generator is an effective tool for improving the process of generating random numbers, especially in terms of long duration and computational efficiency. However, it suffers from some statistical limitations, such as the lack of complete randomness and poor performance in statistical tests. Therefore, this generator is recommended for applications that do not require high-quality statistical randomness, while generators that are more sophisticated are preferred for sensitive applications such as cryptography or accurate statistical simulation.

❖ **Java code to generate random numbers using Fibonacci series: [10]**

```
public class Iterative Fibonacci {     public static void fibonacci (int MAX) {
        int firstNumber = 0;
        int secondNumber = 1;
        int fibonacci = '\0';
        System.out.print(firstNumber + " ");
        System.out.print(secondNumber + " ");
        for (int i = 2; i < MAX; i++) {
        fibonacci = firstNumber + secondNumber;
        System.out.print(fibonacci + " ");
        firstNumber = secondNumber;
        secondNumber = fibonacci;
                }
        }
        public static void main(String [] args) {
        System .out . println("Print Fibonacci Series Using Iterative Method");
        int MAX = 15;
        fibonacci(MAX);
        }
}
```

❖ **Fibonacci random number generation code execution screen in Java language:**

DOI: https://doi.org/10.54216/PAMDA.040102
Received: January 23, 2024 Revised: March 19, 2024 Accepted: June 28, 2024

```
public class fibclass {  new *
    public static void fibonacci(int MAX) {  1 usage  new *
        int firstNumber = 0;
        int secondNumber = 1;
        int fibonacci = '\0';
        System.out.print(firstNumber + " ");
        System.out.print(secondNumber + " ");
        for (int i = 2; i < MAX; i++) {
            fibonacci = firstNumber + secondNumber;
            System.out.print(fibonacci + " ");
            firstNumber = secondNumber;
            secondNumber = fibonacci;}}
    public static void main(String[] args) {  new *
        System.out.println("Print Fibonacci Series Using Iterat
        int MAX = 15;
        fibonacci(MAX);

    }
}
```

```
line ×

C:\Users\Sara\.jdks\openjdk-20.0.2\bin\java.exe "-javaagent:C:\Progra
5 4 1 6 0 3 5 4 1 6
Process finished with exit code 0
```

❖ **Calculating the complexity of the Fibonacci sequence method code:**

We have a for loop that repeats $n-2$ times

The rest of the instructions from print and assign cost 1

Therefore, the complexity is $O(n-2)$

So, $O(n)$

Liner congruential Methods: $[1-5]$

The (Linear Congruential Generator - LCG) is one of the oldest and simplest methods for generating random numbers.

This method relies on a linear function that uses certain coefficients to generate a sequence of numbers that appear to be random.

It can be defined by the following formulas:

$$R_{i+1} = (AR_i + C) Modulo\ M; i = 0,1,2,3 \dots$$

Thus:

$$0 \leq R_{i+1} \leq M - 1$$

Where:

$$A < M, C < M$$

We illustrate the above through the following example:

DOI: https://doi.org/10.54216/PAMDA.040102
Received: January 23, 2024 Revised: March 19, 2024 Accepted: June 28, 2024

Use the linear matching method to generate random numbers where we have

$$R_0 = 7\ , A = 5, C = 3, M = 16$$

$$R_1 = (AR_0 + C)/Modulo\ M$$

$$R_1 = (5 * 7 + 3)/Modulo\ 16\ = 38/Modulo\ 16 = 6$$

$$R_1 = 6$$

$$R_2 = (AR_1 + C)/Modulo\ M$$

$$R_2 = (5 * 6 + 3)/Modulo16 = 33/Modulo16\ = 1$$

$$R_2 = 1$$

We continue working until we get the sufficient number of numbers Randomness required for the simulation process (series of random numbers $R_1, R_2, \ldots$).

❖ **Advantages of the liner congruential Methods:**

✓ Ease of implementation: The method is easy to implement and requires only simple calculations.
✓ Linear matching generators are simple and fast generators that produce almost uniform outputs.
✓ They achieve the properties of equal distribution, independence, and the maximum density property.

❖ **Disadvantages of the liner congruential Methods:**

✓ Predictability: If some of the generated numbers are known, the coefficients A, C, M can be deduced and the entire sequence broken.
✓ Dependence of the period on the coefficients: The maximum period (M) is achieved only if the coefficients A, C, M are chosen according to conditions.
✓ Knut is warning: Random selection of A, C, M will inevitably lead to short periods.
✓ Unsafe: It is by no means safe and should not be used as a secure random number generator. It also suffers from having very inaccurate statistical properties.
✓ Sensitivity of the coefficients: Any small change in $A\ or\ C$ can destroy the randomness.

❖ **Improving the linear congruence method:**

In the context of the rapid development of random number generation techniques, the linear congruence method (LCG) remains a cornerstone due to its ease of implementation and computational efficiency. However, its exclusive reliance on simple linear equations exposes it to serious criticisms,

Therefore, it has become an urgent necessity to develop this method by incorporating non-linear mechanisms, such as quadratic congruence and inverse congruence.

❖ **Quadratic Congruential Method [5]:**

The Quadratic Congruential Method is considered an improvement of the linear congruence method, where a quadratic term is added to improve the random distribution and extend the repetition period. This method is based on the following equation:

$$R_{i+1} = (DR_i^2 + AR_i + C)Modulo\ M\ ; i = 0,1,2,3 \ldots$$

Where:

D: Quadratic constant (must be $D \neq 0$).

A: Linear constant.

C: Increasing constant.

M: Scale (usually a prime number or a power of 2).

❖ **Its role in improving the linear congruence method:**

✓ Better distribution: When the parameters A, D, C are chosen carefully, a better random distribution can be achieved compared to the linear method.
✓ Longer period: If M is a prime number and the parameters A, D, C are chosen appropriately, a long period of random sequence can be achieved.

DOI: https://doi.org/10.54216/PAMDA.040102
Received: January 23, 2024 Revised: March 19, 2024 Accepted: June 28, 2024

❖ **Disadvantages of Quadratic Congruential Method:**

The Quadratic Congruential Method is an effective tool for improving the performance of random number generators, especially in terms of random distribution and repetition period. However, it suffers from some limitations, including:

✓ Sensitivity to parameters: Random selection of A, D, C may lead to short periods or unwanted patterns.
✓ Computational complexity: Due to quadratic operations (R_i^2).

The Quadratic Congruential Method is an effective tool for improving the performance of random number generators, especially in terms of random distribution and repetition period. However, it suffers from some limitations, such as sensitivity to parameters and computational complexity. Therefore, it is recommended to use this method in applications that require high quality in random distribution, taking into account the careful selection of parameters to achieve the best results.

❖ **Inversive Congruential Generator**: [5]

Inversive Congruential Generator are one of the advanced methods for generating random numbers, as they rely on using the multiplicative inverse instead of traditional linear operations. This approach provides a better statistical distribution of the generated numbers, making them suitable for applications that require a high degree of randomness and statistics.

Mathematical formula:

Inversive Congruential Generator are based on the following formula:

$$R_{i+1} = (AR_{i-1}^{-1} + C) Modulo\ P; i = 0,1,2 \dots$$

Where**:**

P: Prime number (determines the range of generated numbers).

A: Multiplicative constant (must be $A \neq 0$).

C: Increment constant (may be zero or any integer).

R_{i-1}: Multiplicative inverse of R_i, modulo P.

❖ **Maximum Period Conditions:**

To achieve a maximum period (i.e., the sequence of numbers repeats after the maximum number of steps), it must:

P : prime number.

$A\ and\ C$: are non-zero (in most cases).

$A\ and\ C$: are chosen so that a complete cycle is produced before iteration.

❖ **Its role in improving the linear congruence method:**

✓ Excellent statistical distribution: No linear patterns or correlations appear between the generated numbers, making it suitable for statistical applications.
✓ Long period: If P is chosen as a large prime number, the period may reach P
✓ Resistant to statistical tests: These generators pass tests such as Chi-Square and Spectral Test with ease.

❖ **Disadvantages of the Inversive Congruential Generator:**

Despite the many advantages offered by the inverse **Inversive Congruential Generator**, such as excellent statistical distribution and long periods of the generated numbers, some challenges and disadvantages may limit their effectiveness in some applications. These disadvantages include:

✓ High computational cost: Calculating the multiplicative inverse R_(i-1) is a computationally expensive process, especially when p is large.
✓ Difficulty in choosing parameters: p must be a prime number, which determines the range of numbers and makes choosing parameters more complex.
✓ Unsuitability for fast applications: Due to the high computational cost, this method may not be suitable for applications that require generating random numbers very quickly.

Although inverse matching generators provide excellent statistical distribution and long periods of the generated numbers, their high computational cost and difficulty in choosing parameters may limit their use

DOI: https://doi.org/10.54216/PAMDA.040102
Received: January 23, 2024 Revised: March 19, 2024 Accepted: June 28, 2024

in some fast applications. However, this method remains a strong option for applications that require a high degree of randomness and statistical accuracy.

- **Java code to generate random numbers using liner congruential Methods:** [11]

```
// Java implementation of the above approach
import java.util.*;
class GFG {
	// Function to generate random numbers
	static void lcm(int seed, int mod, int multiplier,
	int inc, int[] randomNums,
	int noOfRandomNum)
	{
	// Initialize the seed state
	randomNums[0] = seed;
	// Traverse to generate required
	// numbers of random numbers
	for (int i = 1; i < noOfRandomNum; i++) {

	// Follow the linear congruential method			)randomNums[i] = ((randomNums[i - 1] * multiplier)+inc)%m
		}
	}
	// Driver code
	public static void main(String[] args)
	{
	// Seed value
	int seed = 5;
	// Modulus parameter
	int mod = 7;
	// Multiplier term
	int multiplier = 3;
	// Increment term
	int inc = 3;
	// Number of Random numbers
	// to be generated
	int noOfRandomNum = 10;
	// To store random numbers
	int[] randomNums = new int[noOfRandomNum];
	// Function Call
	lcm(seed, mod, multiplier, inc, randomNums,
	noOfRandomNum);
```


DOI: https://doi.org/10.54216/PAMDA.040102
Received: January 23, 2024 Revised: March 19, 2024 Accepted: June 28, 2024

```
            // Print the generated random numbers
for (int i = 0; i < noOfRandomNum; i++) {
            System.out.print(randomNums[i] + " ");
                        }
            }
}
```

- **Two screens to execute the code to generate random numbers using the liner congruential Methods in Java:**

- **Calculating the complexity of the liner congruential Methods code:**

DOI: https://doi.org/10.54216/PAMDA.040102
Received: January 23, 2024 Revised: March 19, 2024 Accepted: June 28, 2024

The cost of defining a variable or assigning it is equal to 1, i.e., $O(1)$,

There is a for loop, the cost of calling it is $O(n)$.

Therefore, the cost of linear matching is $O(n)$.

3. Conclusion and results

Random numbers are an essential part of many scientific and technical applications, from computer simulation to cryptography. In this context, four main methods for generating random numbers are discussed:

The middle square method, the middle multi method, the Fibonacci methods, and liner congruential methods.

- Middle square method: This method depends on taking the square of the previous number and extracting the middle numbers from it. Although it is easy to apply, it may suffer from a lack of randomness, especially when used for long periods, as it can lead to duplications.
- Middle multi method: This method is used to multiply two random numbers to generate a new number. It provides a good distribution of random numbers, but requires care in choosing the prime numbers to avoid duplications.
- Fibonacci methods: This method depends on the mathematical relationship between numbers, where each number is the sum of the previous two numbers. Although simple, the Fibonacci series is not completely random, which may affect the quality of the resulting numbers in applications that require strong randomness.
- Liner congruential methods: It is one of the most common and effective methods, as it uses a linear equation to generate numbers. This method provides fast performance and good random results when the parameters are chosen correctly.

Which method is best?

There is no one answer that fits all applications, as the choice of the appropriate method depends on the requirements of the specific application. However, it can be said that liner congruential Methods one of the best methods in general, due to its speed and efficiency in producing high-quality random numbers, provided that the parameters are chosen carefully.

While the mean product method also provides good results, it may require fine-tuning to avoid duplicates. Ultimately, using any of these methods requires a deep understanding of their properties and limitations, which helps in choosing the method that is most suitable for the specific needs by calculating the complexity of each of the four algorithms, we obtain the following table:

Algorithmic	Algorithmic complexity
Fibonacci	$(O(n) = O(n-1) + O(n-2)$
Middle Square	$O(2n)$
Middle multi	$O(2n)$
Linear congruential	$O(n)$

According to the complexity table for each algorithm, we find that the Fibonacci algorithm is the best in terms of complexity, followed by Linear congruential.

References

[1] A. K. I. Sheet, *Introduction to Random Number Generators and Simulation*, University of Mosul, 2009.

[2] I. M. Alali, *Operations Research*, Tishreen University Publications, 2004.

[3] S. Ahmed, *Operations Research*, Arab Center for Arabization, Translation, Authorship, and Publishing, Damascus, 1998.

[4] A. M. A. R. Bari, *Modeling and Simulation*, Saudi Arabia, 2002.

[5] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, Addison-Wesley, 1969.

DOI: https://doi.org/10.54216/PAMDA.040102
Received: January 23, 2024 Revised: March 19, 2024 Accepted: June 28, 2024

[6] V. Padanyi and T. Herendi, "Generalized Middle Square Method," 2022.

[7] B. Widynski, "Middle-Square Weyl Sequence RNG," 2022.

[8] H. Rahimov, M. Babaie, and H. Hassanabadi, "Improving Middle Square Method RNG Using Chaotic Map," 2011.

[9] "Pseudo-random numbers/Middle-square method," *Rosetta Code*, [Online]. Available: https://rosettacode.org/wiki/Pseudo-random_numbers/Middle-square_me. [Accessed: Feb. 2025].

[10] "Fibonacci Series in Java," *Javatpoint*, [Online]. Available: https://www.javatpoint.com/fibonacci-series-in-java. [Accessed: Feb. 2025].

[11] "Java Program to Implement the Linear Congruential Generator for Pseudo Random Number Generation," *GeeksforGeeks*, [Online]. Available: https://www.geeksforgeeks.org/java-program-to-implement-the-linear-congruential-generator-for-pseudo-random-number-generation/. [Accessed: Feb. 2025].

[12] D. Johnston, *Random Number Generations - Principles and Practices*, 2018.

[13] Austrian Academy of Sciences, *Random Number Generation and Quasi Monte Carlo Methods*, Philadelphia, PA, 1992.

[14] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed., Addison-Wesley, 1997.

[15] J. E. Gentle, *Random Number Generation and Monte Carlo Methods*, Springer, 2003.

[16] J. Von Neumann, "Various Techniques Used in Connection with Random Digits," *Monte Carlo Method, Applied Mathematics Series*, vol. 12, National Bureau of Standards, 1949.

[17] S. K. Park and K. W. Miller, "Random Number Generators: Good Ones Are Hard to Find," *Communications of the ACM*, vol. 31, no. 10, pp. 1192-1201, 1989.

DOI: https://doi.org/10.54216/PAMDA.040102

Received: January 23, 2024 Revised: March 19, 2024 Accepted: June 28, 2024