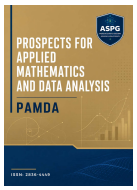
Prospects for Applied Mathematics and Data Analysis

♦ ISSN: 2836-4449 Vol: 04, No. 01, 2024 ♦ PP. 09–17

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



FULL LENGTH ARTICLE

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

Emails: sawsan.almokabaa@damascusuniversity.edu.sy · maissam.jdid66@damascusuniversity.edu.sy

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

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, 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. We present a reference study of some of the algorithms used to generate random numbers, highlighting their advantages, disadvantages, and most important areas of use. We then calculate and compare these algorithms. The algorithms discussed are the Middle Square Method, Middle multi-Method, Fibonacci Methods, and 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 concerned with algorithms for generating random numbers [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17], we reached the following observations.

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 that the concept 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 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

© 2024 The Author(s). Published by ASPG. This article is licensed under CC BY 4.0.

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 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. Other methods, such as "mean product" and the "Fibonacci series," rely on addition and multiplication in numerical series to produce seemingly unpredictable random numbers.

Each of these methods has advantages and disadvantages, and they differ in their ability to generate long, non-repeating sequences. This research analyzes and studies several algorithms, focusing on how they work. Before presenting the algorithms, the types of random numbers that can be obtained must be mentioned. True random numbers depend on natural phenomena such as electrical noise or particle movement and provide high randomness, but they are difficult to obtain and measure. Pseudo-random numbers are produced by mathematical algorithms that use initial values (seeds) to generate sequences that appear random. Although these numbers are not completely random, they are sufficient for many applications, provided that they have the following properties:

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

2. DISCUSSION

This research focuses on the following methods in terms of their advantages, disadvantages, improvements, and most important areas of use:

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

2.1 Middle Square Method

The Middle Square Method is a simple and easy algorithm for generating random numbers, introduced by John von Neumann in 1949 [1, 2, 3, 4, 5, 6]. The method is given by

$$R_{i+1} = \text{Mid}\left[R_i^2\right], \qquad i = 0,1,2,3,\ldots, \tag{1}$$

where Mid denotes the middle digits of the squaring result. We choose R_i as a random fractional number consisting of four digits and containing no zero among those digits. In this case, Mid refers to the middle four digits of R_i^2.

The method is summarized as follows:

- ✓ Choose an initial value (seed).
- ✓ Square the initial value.
- ✓ Extract the middle digits.
- ✓ Use the extracted digits as the new random number and repeat until the required number of values has been obtained.

The number of extracted digits must equal the length of the initial number. If the initial number consists of six digits, for example, the six middle digits are extracted.

2.1.1 Example 1

Let $seed = R_0 = 0.1234$. Squaring the initial value gives

$$(R_0)^2 = (0.1234)^2 = 0.01522756. \tag{2}$$

The middle four digits yield the new random number $R_1 = 0.5227$. Repeating the process gives

$$(0.5227)^2 = 0.27321529, \tag{3}$$

so the next four middle digits give $R_2 = 0.3215$. The process continues until enough random numbers have been generated.

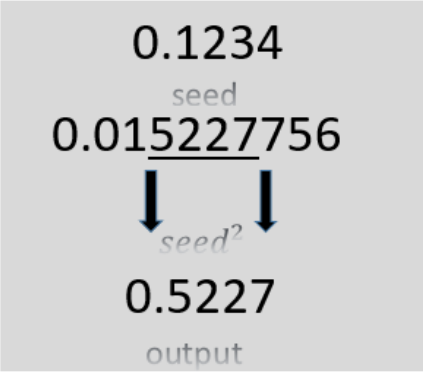


Figure 1. Illustration of the Middle Square Method for the seed 0.1234.

2.1.2 Features of the Middle Square Method

- ✓ **Easy to implement:** It 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:** Because of its simplicity, random numbers can be calculated quickly.
- ✓ The required operations—squaring and taking the middle digits—can be performed quickly even on devices with limited computing capabilities.
- ✓ **Ease of programming:** The method can be programmed in most programming languages, making it useful for teaching and experimentation when the goal is to understand the basics of random-number generators.

2.1.3 Disadvantages of the Middle Square Method

- ✓ **Short cycles:** The method produces short cycles that may repeat quickly. Once the sequence reaches a certain number, it can enter a closed loop and the values begin repeating periodically. This reduces the quality of randomness and makes the generated values unusable in applications requiring long random sequences.

Example 2. Let $seed = R_0 = 0.2499$. The sequence is

$$R_1 = \text{Mid}\left[(0.2499)^2\right] = 0.2450, \quad (4)$$
$$R_2 = \text{Mid}\left[(0.2450)^2\right] = 0.0025, \quad (5)$$
$$R_3 = \text{Mid}\left[(0.0025)^2\right] = 0.0000. \quad (6)$$

Thus,

$$R_0 = 0.2499, \quad R_1 = 0.2450, \quad R_2 = 0.0025, \quad R_3 = 0.0000. \quad (7)$$

This is a very short cycle in which the numbers repeat or vanish quickly.

- ✓ **Convergence to zero:** If the middle number has too many zeros at the ends, continued squaring may cause the generated values to vanish toward zero, ending the production of useful random numbers.
- ✓ **Lack of statistical randomness:** The generated numbers may not perform well in statistical tests of randomness and may exhibit a non-random pattern or distribution.
- ✓ **Computational dependencies:** The method relies heavily on precise numerical calculations. A small error in squaring or extracting the middle digits can lead to incorrect results.
- ✓ **Slow technology:** Many multiplications and divisions are required to obtain the middle digits in a binary computer with a fixed process. Despite its simplicity, the method is unsuitable for modern applications requiring high levels of randomness and quality. It can be used for educational or experimental purposes, but not for sensitive uses such as cryptography or scientific simulations.

2.1.4 Improving the Middle Square Method

The Middle Square Method is a classical pseudo-random-number algorithm, but it suffers from fundamental drawbacks such as short cycle length and a tendency to decay toward zero. Research has therefore combined it with other mathematical techniques, most notably the Weyl Sequence and Logistic Map [7, 8].

Weyl Sequence. Weyl sequences are a mathematical mechanism for improving random-number generators, especially when used with the Middle Square Method. They provide a simple linear addition that increases randomness and avoids redundancy [7]. The mechanism adds a constant S to a variable W at each step:

$$W \mathrel{+}= S. \quad (8)$$

Here, W is a 64-bit variable updated at each step, and S is a carefully chosen odd, non-repeating unsigned 64-bit integer, contributing to an extremely long lifetime.

At every generation step, the following operations are performed:

- ✓ **Squaring:** The 64-bit variable X is squared.
- ✓ **Adding a Weyl sequence:** The Weyl value W is added to the squaring result.
- ✓ **Extracting the middle digits:** The middle parts are extracted using a 32-bit circular shift.

Example 3. Consider the 16-digit hexadecimal value

$$X = \texttt{E3296D171EC4A36F}. \quad (9)$$

Its square is represented as

$$\texttt{C9927E2B2075471D31C2914AAE4E8A21}, \quad (10)$$

whose middle digits are `2075471D31C2914A`. Since the computer stores only 64 bits, X contains the lower part `31C2914AAE4E8A21`. A 32-bit circular shift converts it to `AE4E8A2131C2914A`, and the function returns the lower 32 bits, `31C2914A`.

The Weyl sequence improves the Middle Square Method in several ways:

- ✓ **Overcoming the zero mechanism:** If the middle digits reach zero in the traditional method, the generator enters an infinite loop of zeros; the Weyl addition prevents this behavior.
- ✓ **Long-period guarantee:** The Weyl sequence prevents repeating cycles, increasing the generator period to 2^{64} numbers before repetition.
- ✓ **Efficiency:** Implementation requires simple instructions such as `imulq`, `iaddq`, and `rorq`, making it suitable for high-performance applications.
- ✓ **Uniformity:** The Weyl sequence produces a uniform distribution, helping the final output remain uniform even when the squared value is irregular.

Logistic Chaos Map. The Middle Square Method can also be improved by combining it with a chaotic Logistic Map, addressing some of the traditional problems of the method [8]. The source study gives the Logistic Map as

$$x_{n+1} = (x_n - 1)x_n r, \quad (11)$$

where x_n is the current value and r is a control parameter, usually chosen between 3.4 and 4 to achieve chaos. The result is described as a real number between zero and one and can be converted to a suitable integer. Values derived from the chaotic map replace some Middle Square Method values when repetition or short cycles appear. The improved method is reported to provide higher-quality random numbers for cryptographic and security applications.

2.1.5 Java Implementation

The following Java code implements the Middle Square Method [9].

Listing 1. Java implementation of the Middle Square Method.

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

final class MiddleSquare {
    public MiddleSquare(int seed) {
        final int length = String.valueOf(seed).length();
        if (length % 2 == 1) {
            throw new IllegalArgumentException(
```

```
                "Seed must have an even number of digits");
        }
        state = seed;
        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;
}
```

The source analyzes the code complexity as follows: division of the domain by two has complexity $\mathcal{O}(\log_2 n)$; for `nextInt()`, multiplication of a number by itself is called n times and is reported as $\mathcal{O}(2n)$. Therefore, the complexity of the entire program is reported as $\mathcal{O}(2n)$.

2.2 Middle Multi-Method

The Middle multi-Method is another classic method for generating random numbers, introduced by John von Neumann in 1949 [1, 3]. It is similar to the Middle Square Method, except that the new number is obtained from

$$R_{i+1} = \text{Mid}[D(R_i R_{i-1})], \qquad i = 0, 1, 2, \ldots, \tag{12}$$

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

2.2.1 Example 4

Assume that a series is generated from the two initial values

$$R_0 = 1234, \qquad R_1 = 5678. \tag{13}$$

First,

$$R_0 R_1 = 1234 \times 5678 = 7006652. \tag{14}$$

Taking the middle digits gives $R_2 = 066$. Repeating with R_1 and R_2 gives

$$R_1 R_2 = 5678 \times 066 = 374748, \tag{15}$$

and the middle digits give $R_3 = 47$. The resulting sequence begins

$$R_0 = 1234, \quad R_1 = 5678, \quad R_2 = 066, \quad R_3 = 47. \tag{16}$$

The work is continued until enough random numbers are obtained.

2.2.2 Advantages

- ✓ It has a longer period than the Middle Square technique.
- ✓ The numbers are distributed more uniformly than with the Middle Square technique.
- ✓ It is better than the Middle Square Method.

2.2.3 Disadvantages

- ✓ The 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 with some other methods.

2.2.4 Java Implementation

The following Java code implements the Middle multi-Method as presented in the source article.

Listing 2. Java implementation of the Middle multi-Method.

```
public class MiddleProductRandom {
    private long seed1;
    private long seed2;
    private int 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;
    }

    public long nextInt() {
        long product = seed1 * seed2;
        String productStr = String.format(
            "%0" + (digits * 2) + "d", product);
        int start = (productStr.length() - digits) / 2;
        String middleDigits = productStr.substring(
            start, start + digits);
        seed1 = Long.parseLong(middleDigits);
        seed2 = (seed2 + 1) %
                (long) Math.pow(10, digits);
        return seed1;
    }

    public static void main(String[] args) {
        MiddleProductRandom randomGenerator =
            new MiddleProductRandom(1234, 5678, 4);
        System.out.println(
            "Random numbers generated using " +
            "Middle Product Method:");
        for (int i = 0; i < 10; i++) {
            System.out.println(randomGenerator.nextInt());
        }
    }
}
```

The code-complexity calculation in the source lists product complexity as $\mathcal{O}(2n)\mathcal{O}(n^2)\mathcal{O}(2n)$; text formatting, middle-digit extraction, and conversion from text to number as $\mathcal{O}(n)$; and constant operations as $\mathcal{O}(1)$. The reported total complexity per call is

$$\mathcal{O}(2n) + \mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(1) = \mathcal{O}(2n). \tag{17}$$

2.3 Fibonacci Methods

Random numbers can also be generated from the original Fibonacci sequence, starting with two numbers, usually zero and one, with each subsequent number equal to the sum of the preceding two [3, 5]. In random-number generation, a similar concept is used with modifications to meet randomness requirements. The Fibonacci series begins

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots, \tag{18}$$

and the next value in this example is $21 + 34 = 55$.

2.3.1 Modified Fibonacci Method

The Fibonacci Method for generating random numbers uses the series in a slightly modified way:

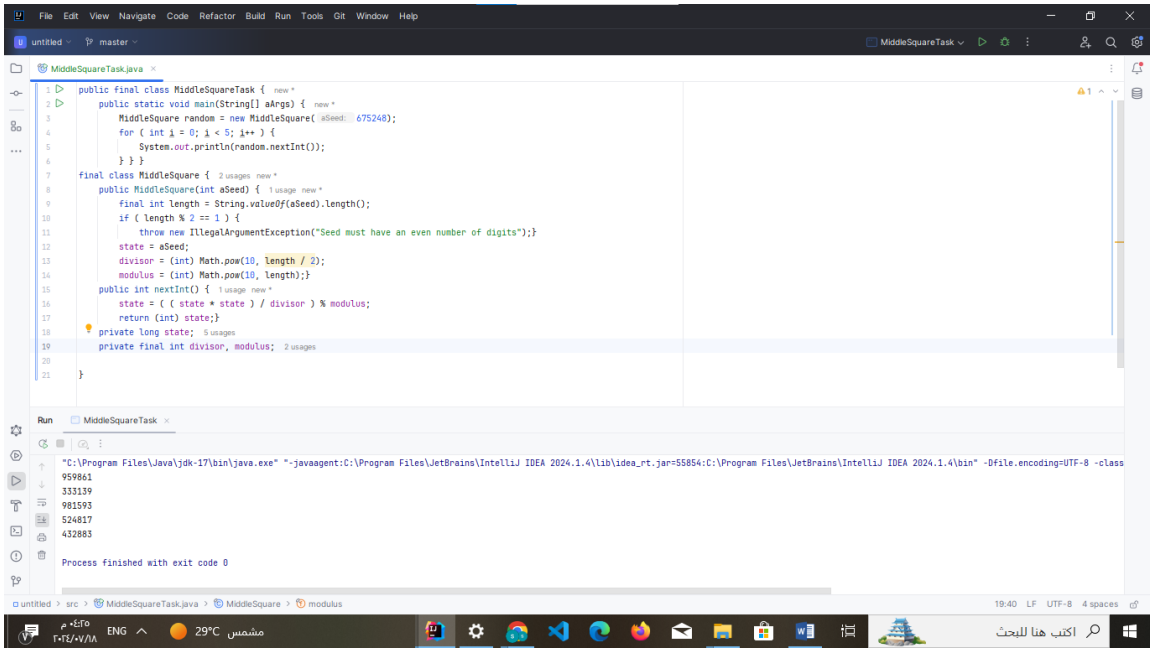


Figure 2. Execution screen for the Java implementation of the Middle Square Method.

$$R_{i+1} = (R_i + R_{i-1}) \bmod M. \tag{19}$$

Features.

✓ It is easy to implement and understand without complex programming skills.

✓ This method produces a greater role for M.

Disadvantages.

✓ The Fibonacci series is easy to predict; attackers can predict upcoming values.

✓ It fails to pass randomness.

✓ It does not give appropriate results.

2.3.2 Lagged Fibonacci Generator

The classical Fibonacci-series method faces significant challenges. To address these limitations, the Lagged Fibonacci Generator re-engineers the relationship between sequence terms by introducing a time delay, greatly increasing the complexity and length of the cycle. The generator is based on

$$R_i = (R_{i-L} + R_{i-K}) \bmod M, \qquad 0 < L < K, \tag{20}$$

where L and K are lag periods and M is the scale, usually a power of two such as 2^{32} or 2^{64}. The example in the source is

$$R_i = (R_{i-24} + R_{i-55}) \bmod e^2, \tag{21}$$

with $K = 55$, $L = 24$, and $M = e^2$.

The delayed Fibonacci generator offers the following advantages:

✓ **Long period:** When L and K are chosen appropriately, a very long period can be achieved. For example, choosing $L = 24$ and $K = 55$ can produce a period of 2^{55} or more, reducing repetition.

✓ **Computational efficiency:** The generator relies on simple operations such as addition and subtraction, without complex multiplication, making it fast and efficient when large quantities of random numbers are required.

✓ **Good distribution:** It provides a homogeneous distribution within the specified range, making it suitable for applications requiring a regular distribution.

Despite these advantages, the Lagged Fibonacci Generator is not statistically perfect:

✓ **Lack of statistical randomness:** The generated sequence is not completely random statistically, limiting its use where high-quality randomness is required.

✓ **Weakness in distribution tests:** The generator fails some basic tests, such as the gap test and run test, indicating repetitive patterns or unwanted correlations.

✓ **Reliance on initial values:** If $R_0, R_1, \ldots, R_{K-1}$ are not varied or carefully chosen, repetitive patterns may appear quickly.

The Lagged Fibonacci Generator is effective in terms of long duration and computational efficiency, but more sophisticated generators are preferred for sensitive applications such as cryptography or accurate statistical simulation.

2.3.3 Java Implementation

The following Java code generates Fibonacci numbers [10].

Listing 3. Java implementation of the iterative Fibonacci series.

```
public class IterativeFibonacci {
    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;
        }
```

```java
public class Main {

    private long seed1;
    private long seed2;
    private int digits;

    // Constructor to initialize the seeds and validate the digits
    public Main(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;}

    // 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
        Main randomGenerator = new Main(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());
        }
    }
}
```

```
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.
```

Figure 3. Three execution screens for the Java implementation of the Middle multi-Method.

```
    }

    public static void main(String[] args) {
        System.out.println(
            "Print Fibonacci Series Using Iterative Method");
        int max = 15;
        fibonacci(max);
    }
}
```

The loop repeats $n-2$ times, while print and assignment instructions have constant cost. Therefore, the source gives the complexity as

$$\mathcal{O}(n-2) = \mathcal{O}(n). \tag{22}$$

2.4 Liner Congruential Methods

The Linear Congruential Generator (LCG) is one of the oldest and simplest methods for generating random numbers [1, 2, 3, 4, 5]. It relies on a linear function with fixed coefficients to generate a sequence that appears random:

$$R_{i+1} = (AR_i + C) \bmod M, \qquad i = 0, 1, 2, 3, \ldots, \tag{23}$$

```java
public class fibclass {
    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 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
```

Figure 4. Execution screen for the Java Fibonacci-series implementation.

where

$$0 \le R_{i+1} \le M-1, \qquad A < M, \quad C < M. \tag{24}$$

For example, let $R_0 = 7$, $A = 5$, $C = 3$, and $M = 16$. Then

$$R_1 = (5 \times 7 + 3) \bmod 16 = 38 \bmod 16 = 6, \tag{25}$$
$$R_2 = (5 \times 6 + 3) \bmod 16 = 33 \bmod 16 = 1. \tag{26}$$

The process continues until the required number of values has been generated.

2.4.1 Advantages

- ✓ **Ease of implementation:** The method requires only simple calculations.
- ✓ Linear congruential generators are simple and fast and produce almost uniform outputs.
- ✓ They achieve the properties of equal distribution, independence, and maximum density.

2.4.2 Disadvantages

- ✓ **Predictability:** If some generated values are known, the coefficients A, C, and M can be deduced and the entire sequence broken.
- ✓ **Dependence on coefficients:** The maximum period M is achieved only when A, C, and M satisfy particular conditions.
- ✓ **Knuth's warning:** Random selection of A, C, and M inevitably leads to short periods.
- ✓ **Unsafe:** The method is not a secure random-number generator and can have inaccurate statistical properties.
- ✓ **Sensitivity to coefficients:** A small change in A or C can destroy the randomness.

2.4.3 Improving the Linear Congruence Method

The LCG remains a cornerstone because of its ease of implementation and computational efficiency. However, its reliance on simple linear equations motivates the incorporation of nonlinear mechanisms such as quadratic congruence and inverse congruence.

Quadratic Congruential Method. This method adds a quadratic term to improve random distribution and extend the repetition period [5]:

$$R_{i+1} = (DR_i^2 + AR_i + C) \bmod M, \qquad i = 0,1,2,3,\ldots, \tag{27}$$

where D is a nonzero quadratic constant, A is a linear constant, C is an increment constant, and M is usually a prime number or a power of two.

Its roles in improving the linear method are:

- ✓ **Better distribution:** Carefully selected A, D, and C can achieve a better random distribution than the linear method.
- ✓ **Longer period:** When M is prime and the parameters are chosen appropriately, a long sequence period can be achieved.

The method also has disadvantages. Random parameter selection can lead to short periods or unwanted patterns, and the quadratic operation R_i^2 increases computational complexity. It is recommended for applications requiring high-quality random distribution, provided that parameters are selected carefully.

Inversive Congruential Generator. The Inversive Congruential Generator is an advanced method based on the multiplicative inverse rather than traditional linear operations [5]. The source gives the formula

$$R_{i+1} = (AR_{i-1}^{-1} + C) \bmod P, \qquad i = 0,1,2,\ldots, \tag{28}$$

where P is prime, A is a nonzero multiplicative constant, C is an increment constant, and R_{i-1}^{-1} is the multiplicative inverse modulo P.

To achieve a maximum period, P must be prime; A and C are generally nonzero and chosen so that a complete cycle is produced before repetition. Its advantages include excellent statistical distribution without linear patterns, a potentially long period approaching P, and resistance to tests such as Chi-Square and the Spectral Test.

The disadvantages include the high computational cost of the multiplicative inverse, especially for large P; the difficulty of choosing parameters; and unsuitability for applications requiring very fast generation. Nevertheless, the method remains a strong option where a high degree of randomness and statistical accuracy is required.

2.4.4 Java Implementation

The following Java program implements a Linear Congruential Generator [11].

Listing 4. Java implementation of the Linear Congruential Generator.

```java
import java.util.*;

class GFG {
    static void lcm(int seed, int mod, int multiplier,
                    int inc, int[] randomNums,
                    int noOfRandomNum) {
        randomNums[0] = seed;
        for (int i = 1; i < noOfRandomNum; i++) {
            randomNums[i] =
                ((randomNums[i - 1] * multiplier) + inc)
                % mod;
        }
    }

    public static void main(String[] args) {
        int seed = 5;
        int mod = 7;
        int multiplier = 3;
        int inc = 3;
        int noOfRandomNum = 10;
        int[] randomNums = new int[noOfRandomNum];

        lcm(seed, mod, multiplier, inc,
            randomNums, noOfRandomNum);

        for (int i = 0; i < noOfRandomNum; i++) {
            System.out.print(randomNums[i] + " ");
        }
    }
}
```

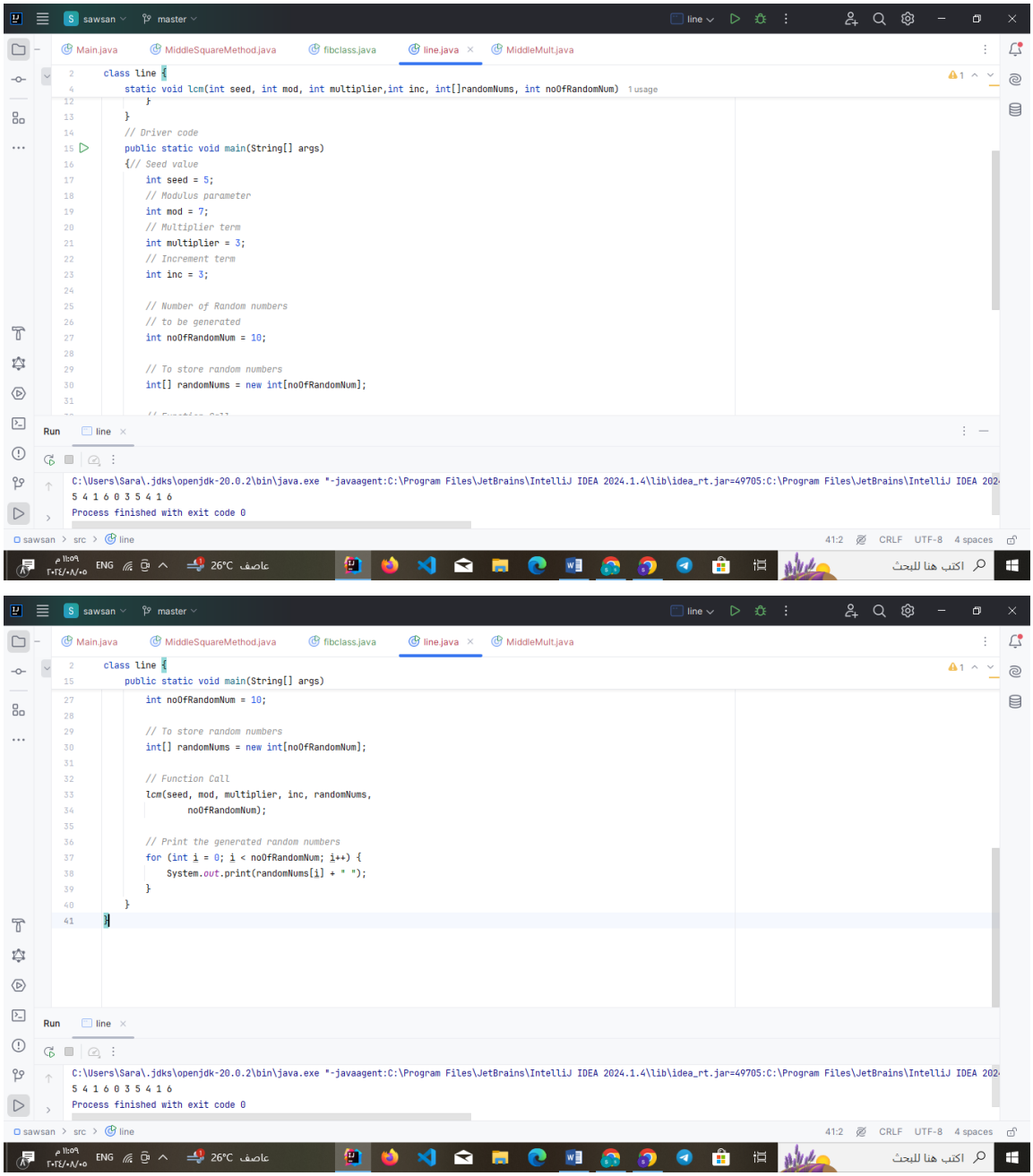


Figure 5. Two execution screens for the Java Linear Congruential Generator.

The cost of defining or assigning a variable is $\mathcal{O}(1)$. The loop costs $\mathcal{O}(n)$; therefore, the reported cost of the linear congruential method is $\mathcal{O}(n)$.

3. CONCLUSION AND RESULTS

Random numbers are an essential part of many scientific and technical applications, from computer simulation to cryptography. Four main generation methods have been discussed: the Middle Square Method, Middle multi-Method, Fibonacci Methods, and Liner congruential Methods.

- **Middle Square Method:** This method squares the previous number and extracts its middle digits. Although easy to apply, it may lack randomness over long periods and can lead to duplication.
- **Middle multi-Method:** This method multiplies two random numbers to generate a new one. It provides a good distribution, but requires care to avoid duplication.
- **Fibonacci Methods:** Each number is the sum of the previous two. Although simple, the Fibonacci series is not completely random, which may affect applications that require strong randomness.
- **Liner congruential Methods:** These common and effective methods use a linear equation and provide fast performance and good random results when the parameters are chosen correctly.

3.1 Which Method Is Best?

There is no single answer suitable for all applications because the appropriate method depends on the requirements of the specific application. However, the Liner congruential Method can be considered one of the best general methods due to its speed and efficiency in producing high-quality random numbers, provided that the parameters are chosen carefully.

The mean product method also provides good results, but may require fine-tuning to avoid duplicates. Ultimately, using any of these methods requires a deep understanding of its properties and limitations.

The complexity reported for each algorithm is summarized in Table 1.

Table 1. Reported algorithmic complexity.

Algorithm	Algorithmic complexity
Fibonacci	$\mathcal{O}(n) = \mathcal{O}(n-1) + \mathcal{O}(n-2)$
Middle Square	$\mathcal{O}(2n)$
Middle multi	$\mathcal{O}(2n)$
Linear congruential	$\mathcal{O}(n)$

According to this table, the source concludes that the Fibonacci algorithm is the best in terms of complexity, followed by the Linear congruential method.

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.

[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] B. Widynski, "Squares: A Fast Counter-Based RNG," arXiv preprint arXiv:2004.06278, 2020.

[10] R. Cybulski, "Pseudo-random number generator based on linear congruence and delayed Fibonacci method," *Technical Sciences*, vol. 24, no. 1, pp. 331–349, 2021.

[11] G. L. Steele Jr. and S. Vigna, "Computationally easy, spectrally good multipliers for congruential pseudorandom number generators," *Software: Practice and Experience*, vol. 52, no. 2, pp. 443–458, 2022.

[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," in *Monte Carlo Method*, National Bureau of Standards Applied Mathematics Series, vol. 12, pp. 36–38, 1951.

[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, 1988.