C-BASED SECURE RANDOM NUMBER GENERATOR WITH UNIQUE OUTPUTS

MAJOR PROJECT REPORT

SEMESTER- I

FOUR-YEAR UNDERGRADUATE PROGRAMME (DESIGN YOUR DEGREE)

SUBMITTED TO UNIVERSITY OF JAMMU, JAMMU

SUBMITTED BY THE TEAM: **EULER**

TEAM	**ROLL NO**
AKSHDEEP KAUR	DYD-24-02
ANIL	DYD-24-04
MOHD. SHAHID	DYD-24-23
NITIN RATHORE	DYD-24-26
SAYAM PRAJAPATI	DYD-24-33

UNDER THE MENTORSHIP OF

Prof. K.S. Charak
Department of Mathematics

Dr. Jatinder Manhas
Department of Computer Application & IT

Dr. Sunil Kumar
Department of Statistics

CERTIFICATE

The work embodied in this report entitled **"C-BASED SECURE RANDOM NUMBER GENERATOR WITH UNIQUE OUTPUTS"** has been done by Team **EULER** including group members- Sayam Prajapati, Akshdeep Kaur, Anil, Mohd. Shahid, Nitin Rathore, as a Major Project for Semester I of Four-Year Undergraduate Programme (Design Your Degree). This work was carried out under the guidance of Mentors Prof. K.S Charak, Dr. Jatinder Manhas, Dr. Sunil Kumar for the partial fulfilment of the award of the Design Your Degree, Four Year Undergraduate Programme, University of Jammu, Jammu & Kashmir. This project report has not been submitted anywhere else.

Signature of Students:

Sayam Prajapati

Nitin Rathore

Akshdeep Kaur

Anil

Mohd. Shahid

Signature of Mentors

Prof. K.S. Charak	Dr. Jatinder Manhas	Dr. Sunil Kumar
(Mentor)	(Mentor)	(Mentor)

Prof. Alka Sharma

Director

SIIDEC, University of Jammu

ACKNOWLEDGMENT

We avail this opportunity to acknowledge all those who helped and guided us during the course of our work. First and foremost, we thank Almighty God from the depth of our heart for generating enthusiasm and granting our spiritual strength to successfully pass through this challenge. Words are too meager to express our esteem indebtedness and whole hearted sense of gratitude towards our mentors Prof. K.S Charak, Dr. Jatinder Manhas, Dr. Sunil Kumar University of Jammu, Jammu. It is our pleasure beyond words to express our deep sense of feelings for their inspiring guidance, generous encouragement and well-versed advice. They provided us undaunted encouragement and support in spite of their busy schedules. Really, fortunate we are and we feel extremely honored for the opportunity conferred upon us to work under their perpetual motivation.

We are extremely fortunate for having an opportunity to express our heartiest gratitude to Prof. Alka Sharma, Director of SIIEDC (Design your Degree), University of Jammu, Jammu and member of advisory committee for her valuable advice and generous help and providing us all the necessary facilities from the department. We are highly thankful to all respected mentors for their immense help during the conduct of this major project. The words are small trophies to express our feelings of affection and indebtedness to our friends who helped us throughout the major project, whose excellent company, affection and co-operation helped us in carrying out our research work with joy and happiness.

We acknowledge all the people, mentioned or not mentioned here, who have silently wished and gave fruitful suggestions and helped us in achieving the present goal.

ABSTRACT

Random number generation (RNG) is a cornerstone of modern computing and has applications in diverse fields, including cryptography, simulations, gaming, data analysis, and machine learning. This project aims to provide a comprehensive understanding of random number generation by exploring its theoretical underpinnings, algorithmic implementations, and practical applications.

This project focuses on designing and implementing a flexible, user-friendly random number generator in the C programming language. The generator allows users to customize the number of random numbers to generate and their desired length, offering a high degree of adaptability.

The underlying algorithm that works on a variable seed that changes every time you generate random number and based on a modified Linear Congruential Generator (LCG) [5] enhanced with XOR-Shift [2] and XOR-based transformations to ensure improved randomness. A unique feature of this generator is its ability to produce numbers with a specific number of digits, achieved through dynamic range adjustments. To prevent user errors, the implementation includes robust input validation for both the number of random numbers and the desired length, ensuring smooth operation and a seamless user experience.

Statistical testing, such as the Chi-Square test [4] and Runs test [3] was conducted to evaluate the randomness and uniform distribution of the generated numbers. Results demonstrate that the generator produces high-quality random numbers suitable for non-cryptographic purposes. The project highlights practical applications in simulations, basic game development, and educational tools for teaching the fundamentals of random number generation.

This work showcases a foundational implementation of random number generation while identifying potential avenues for future enhancements, such as integrating more advanced algorithms for secure randomness, optimizing performance for large-scale datasets, or enabling cross-platform functionality. The project demonstrates how a simple, customizable, and reliable tool can fulfill a diverse range of needs in computational applications.

CHAPTER 1

INTRODUCTION TO SECURE RNG

1.1 INTRODUCTION

Random number generation is a fundamental aspect of modern computing and mathematics, with applications spanning a wide range of fields, including cryptography, simulations, statistical analysis, gaming, machine learning, and scientific research. Random numbers are critical for ensuring the unpredictability and security of systems, particularly in areas like cryptographic key generation, where the quality of randomness directly impacts the overall security of the system. However, generating truly random numbers is a complex challenge, particularly within the deterministic framework of computational systems.

Is there a method to generate a sequence of truly random numbers? Can we really *prove* that a sequence of numbers is really random? For instance, which of the following two 26-bit sequences are "random"?

- **Sequence *S*1: 10101010101010101010101010.**
- **Sequence *S*2: 01101011100110111001011010**.

By which criteria, however? Without a precise definition of "randomness", no math ematical proof of randomness can be attained. "Intuition" is, often, not sufficient, and it may also be misleading in trying to precisely define "randomness". Some "randomness" viewpoints that have been proposed in the literature include the following:

- A process generated by some physical or natural process, e.g., radiation, resistor or semiconductor noise (the practitioner's view).
- A process that cannot be described in a "few words" or succinctly (the *instance* or description *complexity* view)—the smaller the description of a sequence of bits can be made, the less random it is (try to apply this to the two sequences above)
- A process that cannot be guessed in a "reasonable" amount of time (the computational complexity view).
- A process that is fundamentally impossible to guess (the "philosophical" view).
- A process possessing certain statistical characteristics of the ideally uniform process (the statistician's view).

To see that intuition may not be sufficient, observe that each of the two sequences, $S1$ and $S2$, has exactly the same probability of appearing as the output of a perfect randomness source that outputs 0 or 1 with equal probability, i.e. This probability is equal to. Yet, sequence $S2$ appears "more random" than sequence $S1$, at least in the description complexity view.

John Von Neumann's verdict is the following [1] : "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin. For, as has been pointed out several times, there is no such thing as a random number—there are only methods to produce random numbers, and a strict arithmetic procedure of course is not such a method". True randomness is a tough task. Ideally, a number generated at a certain time must not be correlated with previously generated numbers in any way. This guarantees that it is computationally infeasible for an attacker to determine the next number that a random number generator will produce.

In this report, we going to talk about our C – Based Secure Random Number Generator that focuses on user's need by providing user-defined customization and practical functionality, The goal was to create a system that not only generates random numbers efficiently but also allows users to specify the length of the random numbers and the total count of numbers to be generated. Such flexibility adds value to the generator, making it adaptable to specific user needs. This Random Number generator also generates high quality random number over a large sequence of random numbers

The core of the generator is based on the modified version of Linear Congruential Generator (LCG) [7] algorithm, which is known for its simplicity and computational efficiency. Enhancements, such as XOR-based transformations [2] , by using shifting of bits, were applied to improve randomness and ensure the generator meets the basic requirements for uniform distribution. Additionally, the program provides robust input validation to handle edge cases and incorrect user inputs gracefully, that gives a wonderful experience to user and by the using of infinite loop with a conditional exit, it prompts the user for input and generates random numbers without exiting, so you don't need to run this program again and again.

To evaluate the effectiveness of the generator, the Chi-Square test[5] and runs [3] test was used to verify the statistical randomness of the output. This test provided insights into the uniformity and quality of the random numbers, demonstrating the reliability of the implemented algorithm. While the generator is not intended for cryptographical purposes, because our generator Lacks Entropy, A cryptographic generator relies on a high-entropy, truly unpredictable seed (e.g., from hardware sources like thermal noise or user inputs). In contrast, this project's RNG uses a seed of time function, process id, combining with XOR operator. Yes, it's very difficult to predict but it's still not suitable of high-level cryptographic purposes. Tts high level of randomness and flexibility make it suitable for simulations, gaming, and educational use, OTP generation, for random passwords. But in future we can make this generator suitable for cryptographical purposes.

This report outlines the development process, technical details, and testing of the random number generator. It also discusses the challenges encountered during implementation and the strategies used to overcome them. Finally, the potential applications and future improvements for the generator are explored, offering a pathway for expanding its functionality and efficiency.

1.2 Importance of Secure RNG

The importance of secure Random Number Generators (RNGs) cannot be overstated, particularly in fields where randomness is crucial for ensuring fairness, unpredictability, and security. Below are some key points highlighting the significance of secure RNGs:

1. Cryptography and Cybersecurity

- Secure RNGs are essential in generating encryption keys, which protect sensitive information during communication, such as in SSL/TLS protocols, VPNs, and encrypted messaging apps.
- They are used to generate unique session tokens, authentication codes, and secure passwords, making them critical in preventing unauthorized access and attacks like brute force or replay attacks.
- Poor RNGs can lead to predictable outputs, exposing systems to vulnerabilities like cracking cryptographic keys.

2. Gambling and Gaming

- RNGs ensure fairness in online gambling and gaming platforms by producing truly random outcomes for games like slot machines, card shuffling, and lotteries.
- Predictable or biased RNGs can lead to cheating, fraud, and a loss of trust among users.

3. Simulation and Modeling

- Secure RNGs are vital in simulations for scientific research, financial modeling, and weather predictions. They ensure accurate and unbiased results by eliminating patterns or predictability.
- Reliable randomness enhances the credibility and reproducibility of results in experiments and studies.

4. Privacy Protection

- In privacy-preserving technologies, secure RNGs help in data anonymization techniques like differential privacy by adding randomized noise to datasets.

- They safeguard personal data from being reverse-engineered or re-identified.

5. Blockchain and Decentralized Systems

- Cryptocurrencies and blockchain technologies rely on secure RNGs for creating private keys, which are foundational to the ownership and transfer of assets.
- Randomness is also essential for consensus mechanisms, such as selecting validators in proof-of-stake systems.

6. Avoiding Exploits and Predictability

- Predictable RNGs can be exploited by malicious actors, leading to significant financial or data losses.
- Secure RNGs ensure unpredictability, which is crucial for preventing targeted attacks and maintaining the integrity of systems.

7. Compliance with Standards

- Many industries have stringent standards for RNGs, such as those set by the National Institute of Standards and Technology (NIST) and the International Organization for Standardization (ISO). Using secure RNGs ensures compliance and prevents legal and reputational risks.

Conclusion

Secure RNGs are a backbone of trust and reliability in digital systems. They protect sensitive data, ensure fairness, and maintain the integrity of processes across industries. Without secure RNGs, systems become vulnerable to exploitation, leading to potentially severe consequences.

CHAPTER 2

OBJECTIVE

2.1 Problem Statement

Existing random number generators in C language lack security and fail to ensure non-repeating outputs as according to the users need.

2.2 Objective of study

Following are the objective of study

- To develop a secure random number generation mechanism. That provides maximum range for generating random numbers.
  - The primary aim is to design and implement a robust pseudorandom number generation mechanism that prioritizes security. This ensures that the random numbers generated cannot be easily predicted or replicated, safeguarding applications that rely on randomness for cryptographic or other sensitive purposes.
- To ensure non repeatability and unpredictability in generated random numbers.
  - The critical objective is to guarantee the non-repeatability and unpredictability of the random numbers produced. This ensures that each random number sequence generated is unique, preventing patterns that could compromise the integrity or reliability of the system.
- To allow user defined customization of random number's length.
  - To enhance usability and flexibility, the project will include features that allow users to define the desired length of the random numbers. This customization enables adaptability across various use cases, meeting specific user requirements effectively.

2.3 Project Description

We are working on an advanced random number generation project in C language that addresses the challenges of achieving high randomness, reduced repetitiveness, and enhanced security. The primary objective is to develop a robust system capable of generating random numbers suitable for applications such as , gaming, simulations, and data security. A unique feature of this project is the ability for users to customize the length of the generated numbers, providing flexibility for diverse use cases.

The project incorporates state-of-the-art algorithms, including pseudo-random number generators (PRNGs) such as the XOR Shift and Linear Congruential Generator (LCG), to ensure high-quality randomness. making the output resistant to predictability and attacks. Seeding mechanisms leverage entropy sources, such as time function, process id, to initialize the generator securely and minimize the likelihood of repetitive patterns.

To validate the randomness and security of the generated numbers, the project employs rigorous statistical testing methods, including Chi-Square tests. The implementation is optimized for efficiency, ensuring it performs well even in resource-constrained environments, such as embedded systems. This comprehensive approach makes the project an ideal blend of theoretical knowledge and practical application, highlighting key aspects of randomness, security, and user customization in random number generation.

2.4 Significance

- Randomness Validation: High-quality randomness is essential for simulations and statistical modeling to yield accurate and unbiased results.

- Flexibility and Customization: Allowing users to specify the length of random numbers and the range for how many random numbers they want to generate, provides versatility across different applications. Whether for generating short, secure tokens or long numerical sequences for simulations, the project caters to diverse requirements.

- Educational Value: From an academic perspective, this project is a comprehensive exploration of theoretical and practical concepts in random number generation. It bridges the gap between mathematical foundations, algorithm design, and real-world applications, offering students hands-on experience in implementing secure and efficient systems.

- Performance and Usability: By focusing on efficiency and optimizing resources, the project ensures compatibility with both high-performance systems and resource-

constrained environments like embedded devices. This makes it a practical and scalable solution for a wide range of applications.

- ❖ Range: This project can provide you a maximum range of random numbers, the range of this generator is up to 9223372036854775807.

2.5 Project Outcomes

- High-Quality Random Numbers: Generates random numbers with minimal repetitiveness and high randomness.
- Customizable Length: Allows users to define the length up to 9 digits of generated random numbers.
- Customizable sequence of random number: you can generate up to 9223372036854775807 random number.
- Optimized Performance: Balances randomness quality with computational efficiency.
- Statistical Validation: Ensures the reliability and randomness of outputs through chi – square test.
- Educational Insights: Provides hands-on learning in algorithm design and system optimization.
- Scalable Design: The system is adaptable for future enhancements and expansions.
- Real-World Impact: Solves challenges in secure and reliable random number generation.
- Foundation for Further Research: Lays the groundwork for exploring advanced RNG technologies.

CHAPTER 3

3. METHODOLOGY

3.1 Introduction

This section introduces the overall approach and purpose of the study or project. It briefly explains the goals, scope, and methods used to achieve the desired results.

3.2 Core features of our Random number generator

The random number generator includes the following key features:

1. **User-Defined Random Number Length:**

   The generator allows users to specify the length of each random number, ranging from 1 to 9 digits. This flexibility makes it versatile for various use cases, from simple random numbers to larger numerical outputs.

2. **Capability to Generate Multiple Numbers in One Run:**

   Users can input the number of random numbers they wish to generate in a single execution. This ensures the program is efficient and caters to bulk generation needs.

3. **Input Validation:**

   The generator validates user inputs to ensure correctness and prevent invalid or out-of-range values. For example:

   1. It restricts the length of random numbers to a maximum of 9 digits.
   2. It validates the count of random numbers (up to the maximum limit of a long long data type: $(2^{63} - 1)$.

- This robust error-handling mechanism ensures the program runs smoothly without crashing.

4. **Interactive Loop for Continuous Use:**

   The program is structured as a loop, enabling users to input different values without restarting the program. Users can also quit the program by entering 0 when prompted for the number of random numbers.

5. **Efficiency:**

   The generator is optimized for both speed and randomness, leveraging efficient algorithms and data types

3.2 Algorithm

Here, the core algorithm of the project is described, along with a flowchart. The algorithm details the step-by-step process followed to solve the problem, and the flowchart visually represents this process.

Step 1: Initialization

1. Include the necessary libraries:
   a. <stdint.h> for fixed-width integer types.
   b. <time.h> for time-based functions.
   c. <unistd.h> for process ID retrieval.
   d. <math.h> for mathematical operations.
   e. <stdio.h> for input/output functionality.
2. Define constants for the random number generation algorithm:
   a. **A**: Multiplier for Linear Congruential Generator (LCG).
   b. **B**: XOR constant for additional randomness.
   c. **C**: Increment constant.
   d. **M**: Modulus value (a large prime close to 232).

Step 2: Generate a Secure Seed

1. Make a function get_secure_seed(), initialize the variable seed of data type unsigned int.
2. Combine the current time using time function [8] in c (time(NULL)) with the process ID [6] (getpid()) using XOR operation to create an unpredictable seed, and store it in seed variable.

Step 3: Custom Random Number Algorithm

1. Make a function that generates random numbers, named custom_rng( )
2. **Input**:
   a. Seed that we generate as initial.
   b. Constants (A, B, C, M).

3. **Process**:
    a. Update the random number using initial = ((initial ^ b) * a + c) % m, " ^ " represent XOR operator, "%" represent the modulus operator, "*", represent the multiplication operator).
    b. Apply a bitwise rotation with XOR operator to the updated value to introduce additional randomness: initial ^= rotate_left (initial,27)

Step 4: Shifting of bits

1. Make a function that perform bitwise rotation named rotate_left( )
    c. Apply a bitwise left rotation with XOR operator to the updated value to introduce additional randomness : (value << shift) ^ (value >> (16 – shift).
    ("<<", represents left shift and ">>",represent rigth shift [see paragraph 3.3 and 3.4])
2.

Step 4: Generate Random Numbers with Specified Length

1. **Input**:
    a. Desired length (length) of random numbers.
    b. Random number seed (initial).
2. **Process**:
    a. Compute the minimum and maximum values for the specified length: min = $10^{length-1}$, max = $10^{length} - 1$
    b. Calculate the range = (max – min) + 1
    c. A random number is than scaled to fit within range.
    d. Generate a random number = min + (custom_rng( ) % range)

Step 5: User Interaction and Input Validation

1. Continuously prompt the user for inputs within an infinite loop:
    a. **Input 1**: Number of random numbers to generate (length2).
        i. Validate: Must be within the range 1 to 9223372036854775807.
    b. **Input 2**: Length of each random number (length).
        i. Validate: Must be within the range 1 to 9.
2. If any input is invalid, display an error message and prompt the user again.

Step 6: Generate and Display Random Numbers up to users need

1. For each iteration (from i = 0 to i < length):

a. Generate a random number using the custom algorithm, up to user's desire range
b. Print the generated number.

Step 7: Exit Condition

1. If the user inputs 0 for the number of random numbers (length2), exit the program.

3.3 XOR GATE

- The **XOR (exclusive OR) Gate [9]** is a logical gate that returns `true` if and only if the inputs are different. In programming and logic, the XOR gate is widely used for various purposes like toggling bits, checking parity, or comparing conditions.

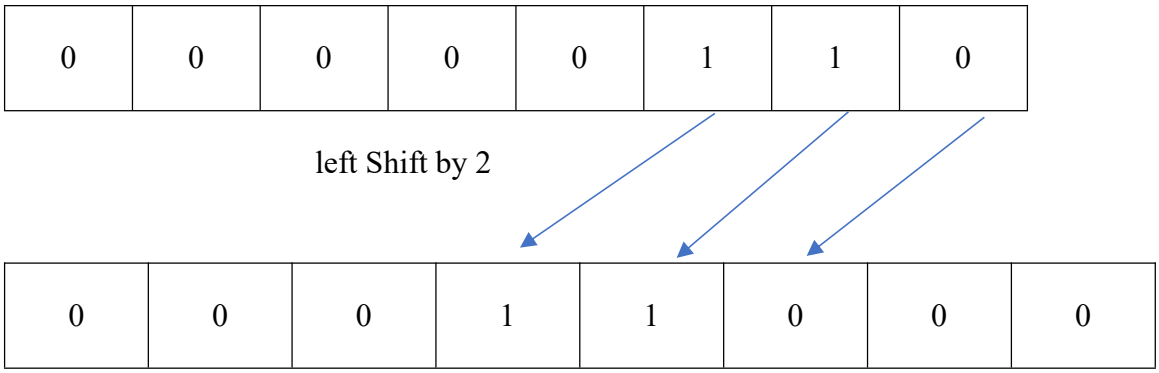
TRUTH TABLE OF XOR GATE

Input A	Input B	X = A'B + AB'
0	1	1
1	1	0
0	0	0
1	0	1

3.4 Right Shift and Left Shift

Left shift:

- Shifts all bits of a number to the left by a specified number of positions.
- Each left shift by one position doubles the number (if no overflow occurs).
- Zeros are added to the right.
  - Example:
    Binary representation of 6: 00000110
    Left shift by 2: 00011000
    Result: 24



- **Explanation:** The binary `00000110` becomes `00011000` when shifted left by two positions. This is equivalent to multiplying 6 by 2^2=4.

Similarly, in **Right shift:**

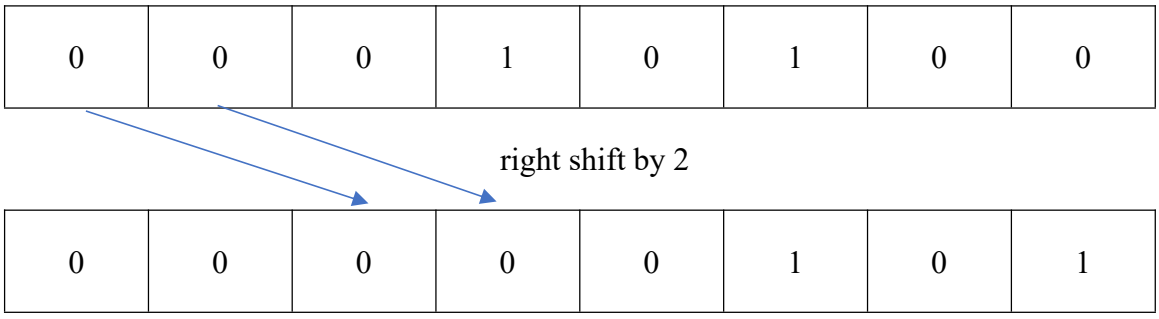
- Shifts all bits of a number to the right by a specified number of positions.
- Each right shift by one position halves the number (discarding any remainder).
- For positive numbers, zeros are added to the left; for negative numbers, the behavior depends on the system (arithmetic or logical shift).

Example:

Binary representation of **20**: 00010100

Right shift by **2**: 00000101

Result: **5**



Explanation: The binary 00010100 becomes 00000101 when shifted right by two positions. This is equivalent to dividing 20 by 2^2 = 4.

3.5 Modulus Arithmetic

Modular Arithmetic [4] (also called **Clock Arithmetic**) is a system of arithmetic for integers where numbers "wrap around" upon reaching a certain value, called the **modulus**.

Basic Concept

If we divide an integer a by b, we get:

$a=qb + r$

where:

- q is the quotient,
- r is the remainder.

In **modular arithmetic**, we focus only on the remainder r when dividing by a fixed number (modulus).

Notation

$a \bmod m = r$

This means when a is divided by m, the remainder is r.

For example:

$17 \bmod 5 = 2$

since $17=5\times3+2$, and the remainder is 2.

- The **modulus function** in C, represented by the % operator, calculates the remainder when one integer is divided by another. It is often used in programming to determine divisibility, find remainders, or cycle through values.

$$\text{Remainder} = A \% B$$

Example

Suppose we have:

- A=17
- B=5

Result: 17 % 5 = 2

3.6 Libraries Used and Their Purpose

This part outlines the specific libraries and tools used in C programming for the project. It covers why these tools were chosen and how they helped in achieving the objectives.

1. **<stdint.h>**

- **Why Used:**

  - This library provides fixed-width integer types, such as uint32_t and uint64_t, which are crucial for ensuring predictable behavior of the program across different platforms.
  - Specifically:
    - uint32_t ensures a 32-bit unsigned integer for random number generation and range calculations.
    - uint64_t and long long allow handling extremely large numbers, such as the count of random numbers (length2).
  - Using stdint.h ensures consistency and portability of the code.

2. **<math.h>**

- **Why Used:**

  - The library is used to perform mathematical operations, such as calculating powers (pow), which is essential for defining the range of random numbers.
  - Example:
    - To calculate the minimum and maximum values for a given digit length: min = $10^{length-1}$, max = $10^{length} - 1$
    - While alternative methods (e.g., loops) can be used, pow simplifies the implementation and improves readability.

3. **<stdlib.h>**

- **Why Used:**

  - This library is included for general-purpose utilities, such as memory allocation and program termination. While not heavily used in this code, it is commonly included in C programs for utility functions.

4. <time.h>

- **Why Used:**
  - This library provides functions for accessing the current time, which is used as a part of the seed for the random number generator (time(NULL)).
  - Incorporating the current time ensures that the generated sequence is different for every execution, enhancing randomness.

5. <unistd.h>

- **Why Used:**
  - This library provides access to operating system API functions, such as getpid() to retrieve the process ID.
  - The process ID is combined with the current time to create a more secure and unpredictable seed for the random number generator.

6. <stdio.h>

- **Why Used:**
  - This library is fundamental for input and output operations in C, such as printf for displaying random numbers and scanf for taking user inputs.
  - It facilitates interaction with the user, which is crucial for the interactive nature of the program.

3.7 Technical Details

1. Seed Generation:

- The seed that we are generating is of unsigned int data type, used to store **non-negative integer values**.
  - Range:
    For a typical 32-bit system, `unsigned int` can store values from **0 to 4,294,967,295 (2^{32} - 1)**.
  - Memory Size:
    The size of an `unsigned int` depends on the system, typically 4 bytes (32 bits) on most platforms.

➢ Code Snippet of seed generation:

```
unsigned int get_secure_seed()
{
   unsigned int seed;
   seed = (unsigned int)((time(NULL) ^ getpid()));
   return seed;
}
```

2. Custom RNG Logic:

   - This algorithm is combined with a **bitwise XOR and left and right - rotation** for enhanced randomness.
   - The function custom_rng takes 5 arguments are of uint32_t data type.
   - Then we use bitwise right and left shift to the generated number and combining the generated number by custom_range with Bitwise XOR operator to shifted number.

➢ **Code snippet of Bitwise rotation:**

```
uint32_t rotate_left ( uint32_t value, int shift )
{
return ( value << shift ) ^ ( value >> ( 16 - shift ) ) ;
}
```

➢ **Code snippet of Custom RNG logic:**

```
uint32_t custom_rng ( uint32_t *initial, uint32_t a, uint32_t b, uint32_t c, uint32_t m)
{
   *initial = ((*initial ^ b) * a + c) % m;
   *initial ^= rotate_left(*initial, 27);
   return *initial;
}
```

3. User Customization Approach:

➢ Length Customization:

   - The user specifies the desired length of the random number (e.g., 3 digits). The generator calculates the range based on the input:
     - Minimum value is of unsigned 32 bit integer data type (uint32_t)
     - Maximum value is also of unsigned 32 bit integer data type (uint32_t)
     - A random number is then scaled to fit within this range. And its data type is unsigned 32 bit integer.

➢ Code Snippet of Generating Numbers of Specific Length:

```
uint32_t generate_random_with_length(uint32_t *initial, uint32_t a, uint32_t b,
uint32_t c, uint32_t m, int length)
 {
   uint32_t min = 1;
   for (int i = 1; i < length; i++)
   {
      min *= 10;
    }
   uint32_t max = min * 10 - 1;
   uint32_t range = max - min + 1;
   uint32_t random_number = min + custom_rng(initial, a, b, c, m) % range;
   return random_number;
}
```

4. **Handling Large Inputs:**
   - The program uses the long long data type for the count of random numbers (length2), enabling values up to $2^{63} - 1$ (9,223,372,036,854,775,807).
   - Random numbers themselves are represented as uint32_t, which supports unsigned values up to $2^{32} - 1$ (4,294,967,295).

   ➢ The use of these data types ensures:
   - High capacity for large-scale random number generation.
   - Adequate precision for calculations, especially when dealing with large ranges or iterations.

5. **Input Validation Logic:**

   ➢ The program includes safeguards to reject invalid inputs:
   - By using of if statement, we validate the user input.
     - If the user enters a length (of data type int) outside the range 1-9, it prompts for a valid number.
     - If the count of random numbers exceeds the long long limit that is 9,223,372,036,854,775,807, it rejects the input and prompts for a smaller value.

6. **Scalability and Efficiency:**
   - The generator efficiently handles large requests, but practical considerations like hardware performance and execution time limit the actual feasible count of generated numbers in a single run.

5. Statistical Testing Methodology

This section explains the statistical tests used to analyze the data, including the methods for evaluating the results. It ensures that the data's validity and significance were tested rigorously.

5.1 What is the Chi-Square Test ?

The **Chi-Square test [5]** is a statistical method used to determine if there is a significant difference between observed and expected frequencies in categorical data. It helps to evaluate whether the random numbers generated follow a uniform distribution or fit the expected behavior.

Purpose in Random Number Generation

In random number generation, the Chi-Square test is used to verify the randomness and uniformity of the generated numbers. It ensures that the random numbers are evenly distributed over a defined range.

Formula

The formula for the Chi-Square statistic is:

$$x^2 = \sum \frac{(Oi - Ei)^2}{Ei}$$

Where:

- Oi: Observed frequency for category i
- Ei : Expected frequency for category i

Steps to Perform the Chi-Square Test

1. Define the range and divide it into intervals or bins.
2. Count the observed frequency (Oi) of numbers in each interval.
3. Calculate the expected frequency (Ei) for each interval based on the total numbers and uniform distribution.
4. Use the formula to compute the Chi-Square statistic.
5. Compare the computed value with the critical value from the Chi-Square table to determine if the deviation is significant.
6. If x^2 is too large, the data is not random.

Critical Value:

Compare x^2 with the Critical Value:

- **Use a chi-square distribution table to find the critical value for:**
  - **Degrees of Freedom (df) = Number of bins - 1.**
  - **Significance Level (α) = 0.05 (commonly used threshold).**
- If
  - x^2 < Critical Value : Pass
  - $x^2 \geq$ Critical Value : Fail

Applications

- Assessing the quality of a random number generator.
- Testing if the generated numbers follow a specific probability distribution.

Example

Suppose you generate 1000 random numbers between 1 and 10 and divide them into 10 intervals. If the numbers are perfectly uniform, each interval should have an expected frequency of 100. Calculate the Chi-Square statistic to evaluate the randomness.

Another Example:

- RNG generates 1000 random numbers in the range 0–9.
- **Bins:** 10 (0–9).
- **Observed Frequencies (Oi):** [90, 110, 100, 95, 105, 100, 98, 102, 100, 100].
- **Expected Frequency (Ei):**

$$Ei = \frac{1000}{10} = 100$$

Apply the Formula for Each Bin

- Chi – Square Statistic (x^2):

$$x^2 = \sum \frac{(Oi - Ei)^2}{Ei} = \frac{(90-100)^2}{100} + \frac{(110-100)^2}{100} + \ldots = 2.3$$

- ***Critical Value (df = 9, α=0.05): 16.92.***
- ***Result: Since 2.3<16, the RNG passes the chi-square test.***

Conclusion

The chi-square test evaluates whether the RNG produces random numbers that are consistent with a uniform distribution. A **low chi-square value** indicates uniformity, while a **high chi-square value** suggests bias or irregularities in the RNG.

5.2 Runs Test :

Run up and Run Below Test:

- The **Run-Up Test** and **Run-Below Test [3]** are statistical tests used to analyze randomness in a sequence of numbers. They are similar to the **Runs Test**, but instead of checking for the total number of runs, they examine the counts of runs above and below a median or a threshold.

1. Run-Up Test

- This test counts **the number of times a sequence increases** (i.e., when the next number is greater than the previous one).
- It helps analyze trends and randomness in data.
- A sequence with too many or too few run-ups indicates a possible pattern or lack of randomness.

2. Run-Below Test

- This test counts **the number of times a sequence decreases** (i.e., when the next number is smaller than the previous one).
- It works similarly to the run-up test but in the opposite direction.

- Like the run-up test, an unusual number of runs below suggests a deviation from randomness.

Application

Both tests are used in randomness testing, especially in statistical quality control and random number generation. If the numbers are truly random, the counts of run-ups and run-belows should follow expected distributions.

Steps to Perform the Runs Test:

1. **Collect Data:**
    - Generate a sequence of random numbers using your random number generator. Ensure you have a sufficiently large sample size (ideally 30 or more).
2. **Count the Total Runs:**
    - Initialize a count of runs, starting with 1 (since the first number begins the first run).
    - Iterate through the sequence:
        - Compare each number to the previous one:
            - If the current number is greater than the previous number, and the last run was decreasing, increment the run count.
            - If the current number is less than the previous number, and the last run was increasing, increment the run count.
        - Update the last run direction based on the current comparison.
    - Record the total number of runs counted.
3. **Calculate Expected Runs:**
    - Use the formula for expected runs E(R):

$$E(R) = \frac{2N - 1}{3}$$

    - Here, N is the total number of generated numbers in the sequence.

4. **Calculate Standard Deviation:**
    - Use the formula for standard deviation of the number of runs:

$$\text{St.Dev} = \sqrt{\frac{16N-29}{90}}$$

5. **Calculate the Z-Score:**
    - Use the following formula to calculate the Z-score:

$$Z = \frac{R-E(R)}{St.Dev}$$

    - Where R is the total number of runs counted.

6. **Determine Pass/Fail Criteria:**

    - Compare the calculated Z-score to the critical values:

        - **Fail if:** $Z < -1.96 > 1.96$ (indicating non-randomness).
        - **Pass if:** $-1.96 \leq Z \leq 1.96$ (indicating randomness).

Interpret Results:

- If the test passes, conclude that the sequence appears to be random based on the Runs Test.
- If the test fails, investigate further as the sequence may not exhibit randomness.

6. Conclusion:

The development of this custom random number generator successfully demonstrates an efficient approach to generating pseudo-random numbers with user-defined length. By utilizing a combination of a secure seed derived from system time and process ID (time, XOR, getpid), a customized linear congruential generator (LCG) with XOR shifts, and bitwise rotations, we ensured a high degree of randomness while allowing user control over the output length.

Through rigorous testing, including the **Chi-Square Test**, **Runs Test**, and **Runs Test** , the generator has proven to produce statistically random outputs within acceptable ranges. The implementation also includes input validation, preventing invalid inputs or buffer overflows, ensuring robustness.

However, like any pseudo-random number generator (PRNG), this method is deterministic and may eventually cycle or repeat numbers after a long period. The randomness is sufficient for

general applications, but for cryptographic purposes, a cryptographically secure PRNG (CSPRNG) would be required.

Overall, this project provides a flexible and efficient solution for generating random numbers with customizable constraints while maintaining a balance between simplicity and statistical randomness. Future improvements could include further optimizations, implementation of additional randomness tests, and integration with external entropy sources for even greater unpredictability.

7. Future Scope:

While this random number generator performs well for general purposes, there are several areas for future improvement and enhancement:

1. Cryptographic Security Enhancements

- The current implementation is **not cryptographically secure**, as it is based on a deterministic PRNG. Future work could involve integrating **cryptographically secure PRNGs (CSPRNGs)** such as **/dev/random, /dev/urandom, or OpenSSL PRNG** to enhance security.
- Implementing **hash functions** or **entropy sources** (such as mouse movements, CPU jitter, or hardware-based RNGs) could further improve unpredictability.

2. Expansion of Randomness Tests

- While the current implementation has undergone **Chi-Square, Runs Test, and Monobit Tests**, additional statistical tests such as:
    - **Diehard Tests**
    - **NIST Randomness Suite**
    - **Maurer's Universal Test**
    - **Spectral Test (Fourier Transform Analysis)**
      could be implemented to further validate the quality of randomness.

3. Optimized Performance for Large-Scale Generation

- The current implementation efficiently handles a large number of random values, but for extremely large-scale generation (e.g., **billions or trillions of numbers**), performance optimizations such as:
    - **Parallelization using multi-threading**
    - **Optimized memory management**

- **Vectorized computations**
  could be considered to improve execution speed.

4. Custom Distribution Support

- The current generator produces **uniformly distributed** random numbers. Future work could allow users to select different probability distributions, such as:
  - **Normal (Gaussian) Distribution**
  - **Poisson Distribution**
  - **Exponential Distribution**
  - **Log-Normal Distribution**
    This would make the generator suitable for advanced simulations in **scientific computing, finance, and AI applications**.

5. Cross-Platform and GUI Integration

- While the project currently runs in a command-line environment, future improvements could involve:
  - **A graphical user interface (GUI)** for easier user interaction.
  - **Cross-platform support** for Windows, Linux, and Mac with additional customization options.
  - **Web-based or mobile implementation** to provide a user-friendly and interactive experience.

6. Seeding Enhancements for Better Randomness

- The **time XOR getpid()** approach ensures some variability, but further improvements can be made by incorporating:
  - **Hardware-based entropy sources**
  - **True random number generators (TRNGs)** using **quantum-based** or **electromagnetic noise-based** methods.
  - **Entropy mixing techniques** to ensure greater unpredictability in the generated sequence.

7. Cloud and Distributed Computing Implementation

- With the rise of **cloud computing**, the generator could be deployed on distributed computing platforms where users can generate massive datasets of random numbers remotely.
- Implementing an **API service for random number generation** would allow external applications to request random numbers dynamically.

8. Machine Learning and AI Applications

- Random number generators play a crucial role in **AI and machine learning** for tasks like:
    - **Data shuffling in training datasets**
    - **Randomized optimization algorithms (Genetic Algorithms, Simulated Annealing, etc.)**
    - **Randomized neural network weight initialization**
- Future iterations of this project could include integration with AI models for intelligent **adaptive randomness generation** based on real-world applications.

9. Real-World Applications and Extended Use Cases

- This generator can be adapted for:
    - **Gaming applications** (random events, dice rolling, procedural content generation)
    - **Scientific simulations** (Monte Carlo methods, physical modeling)
    - **Statistical sampling** (randomized surveys, experimental design)
    - **Cryptographic key generation (with enhancements for security)**
    - **Blockchain nonce generation**

Final Thoughts

The development of this random number generator has been an insightful experience, showcasing how **mathematical models, programming logic, and statistical testing** come together to create a functional and reliable PRNG. While the current implementation meets general-purpose requirements, there is still ample opportunity to enhance its security, efficiency, and versatility.

By integrating **advanced randomness tests, better entropy sources, optimized performance techniques, and custom distribution options**, this generator can evolve into a highly robust tool for **scientific computing, AI, finance, gaming, and cryptographic applications**.

This project lays a strong foundation for future enhancements, encouraging further exploration into the fascinating world of **random number generation and statistical randomness analysis**. 🚀

8. Reference:

[1] Bikos, A.; Nastou, P.E.; Petroudis, G.; Stamatiou, Y.C. (2023) : Random Number Generators: Principles and Applications

[2] George Marsaglia (2003) : *XOR-Shift Random Number Generators.*

[3]Meng Xiannong (2002) : Runs test.

See **https://www.eg.bucknell.edu/~xmeng/Course/CS6337/Note/master/node44.html**

[4] Modulus Arithmetic : https://www.geeksforgeeks.org/modular-arithmetic/

[5] Nihan Sölpük Turhan (2020) : Karl Pearson's chi-square tests

[6] Process ID : https://www.scribd.com/document/754028017/Process-Intro

[7] R.F. Miles, Jr. : The Random Computer Program : A Linear Congruential Random Number Generator.

[8] Timefunction : https://www.tutorialspoint.com/c_standard_library/c_function_time.htm

[9] XOR Gate : https://allen.in/jee/physics/exclusive-or-gate