TITLE PAGE

Comparative Analysis of Random Number Generators Using Monte Carlo Algorithm

By

Yusuf Auwal Sani

(U/CS/13/219)

A PROJECT REPORT SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE, YOBE STATE UNIVERSITY DAMATURU

IN PARTIAL FULFILMENT OF THE REQUIREMENT FOR THE AWARD OF BSc. IN COMPUTER SCIENCE

November, 2017

CERTIFICATION

This is to certify that this project work, Comparative Analysis of Random Number Generators Using Monte Carlo Algorithms, was carried out under the supervision of Engineer Adamu Abdullahi Garba of Computer Science Department, Yobe State University, Damaturu.

Student Name	**Signature and Date**
Yusuf Auwal Sani	**…….**

APPROVAL PAGE

This project work has read and approved by undersigned people on behalf of the Department of Computer Science, Yobe State University Damaturu, as meeting the requirement for the award of Bachelor of Science in Computer Science.

Engr. Adamu Abdullahi Garba ……

Supervisor Signature and Date

Dr. Mahdi Alhaji Musa ……..

Head of Department Signature and Date

.....

External Examiner Signature and Date

DEDICATION

This research work is dedicated to all people that helped me in my life.

ACKNOWLEDMENT

First of all, I am grateful to Almighty ALLAH for giving me life, health; knowledge and wisdom complete this project. Next greeting goes to my thesis supervisor in person of Engineer Adamu Abdullahi Garba for supervising me, support, encouragement, patience and suggestion for the completion of my thesis, from the beginning to the end.

I would also like to thank Dr. Mahdi Alhaji Musa as My head of Department; Computer Science, my profound gratitude also goes to the external examiner for taking his time to certify and evaluate my work despite his tight schedule.

Special appreciation and thanks to all my lecturers that thought me from level one to graduation. I wish to extend my thanks to all friends and colleagues for their valuable help in the development of this thesis. Especially, I will like to thanks Usman Yunusa, Abu-Bakr Suleiman, Bukar Babagana, Salisu Alhaji Haruna, Ali Babale Amshi Bukar Babagana, Idriss Disa, Falmata Modu, Fatima Zanna Bukar, Ahmed Abu-Bakr, Sadiq Yawale, Ali Suleiman, who have been on my side during my happy and difficult days in these four years.

There are no words to express my gratitude to my family and relatives. All can be said is that I am the most fortunate person to have such parents.

ABSTRACT

Random number generation is the art and science of deterministically generating sequence of random number that is difficult to distinguish from true random sequence. This thesis introduced the field of random number generation, where three random number generators implemented in c language are compared in terms of their uniform distribution, independence and speed. Each generator is described and its code presented. The tests used to evaluate the generators were simple ones. First, streams of random numbers generated were checked for uniform distribution by means of chi-square (goodness of fit) test. Next the numbers were checked for independence by means of chi-square independent test (with contingency table). The code was then timed for a certain number of iterations. Finally, the results of the generators are analyzed for used to categorize the generators based on suitable areas of application.

The significance of this research is to address the problems that usually arise when it comes to choosing right random number generators for a particular domain, hence trying to reduce risk. This field has gone through number of trends, however, Mersenne twister generator is now consider as current technology for used for generating pseudo random numbers. Notwithstanding, there is no best pseudo random number generator to this day, shown by many journals and researches.

List of Abbreviations

RNGs	Random Number Generators
TRNGs	True Random Number Generators
PRNGs	Pseudo Random Number Generators
KS	Kolmogorov-Smirnov
NIST	National Institute of Standard and Technology
LCG	Linear Congruential Generator
LFG	Lagged Fibonacci Generator
FSR	Feedback Shift Register

List of Figures

List of Tables

CHAPTER ONE

1.1: INTRODUCTION

"*Everything we do to achieve privacy and security in computer age depends on random numbers*" – Simon Cooper (1951)

Informally, an algorithm is any well-defined computational procedure that take value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into output (Thomas H. Cormen *et al*.**,** .2009).

Monte Carlo methods (or Monte Carlo experiment) are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. Their essential idea is using randomness to solve problem that might be deterministic in principle (Wikipedia, 2017). Before Monte Carlo was developed, simulations tested a previously understood deterministic problem and statistical sampling was used to estimate uncertainties in the simulations. Monte Carlo simulations invert this approach, solving deterministic problem using a probabilistic analog (simulated annealing).An early variant of Monte Carlo method can be seen in the Buffon's needles experiment, in which π can be estimated by dropping needles on a floor made of parallel and equidistance strips. In the (1930s), Enrico Fermi first experimented with the Monte Carlo method while studying neuron diffusion, but did not publish anything on it.

The new version of Markov chain Monte Carlo method was invented in the late 1940s by Stanislaw Ulam, while working on nuclear weapons projects at the Los Alamos National Laboratory. Immediately after Ulam breakthrough, John von Neumann understood it importance and programmed in the ENIAC computer to carry out Monte Carlo calculations. In (1946), physicists at Los Alamos scientific Laboratory were investigating radiation shielding and the distance that neurons would likely travel through various materials. Despite having most of the necessary data, such as average distance a neuron would travel in a substance before it collide with neutron was likely to give off following a collision, the los Alamos physicist were unable to solve the problem using conventional, deterministic mathematical methods. Stanislaw Ulam had the idea of using random experiments.

He recounts his inspiration as follows:

The first thoughts attempts I made to practice the Monte Carlo method were suggested by a question on which occurred to me in 1946 as I was convalescing from an illness and playing solitaire. The question was what are the chances that a canfield solitaire laid out with 52 cards will come out successfully? After spending a lot of time trying to estimate them by pure combinatorial calculations, I wondered whether a more practical method than abstract thinking might not be lay out say one hundred times and simply observed already possible to envisage with the beginning of the new era of fast computers, and I immediately thought of problems of neutron diffusion and other questions of mathematical physics, and more generally how to change processes describe by certain differential equations into an equivalent form interpretable as a succession of random operations. Later (in 1946), I described the idea to John Von Neumann, and we began to plan actual calculations – Stanislaw Ulam. Being secret, the work of Von Neumann and Ulam required a code name. A colleague of Von Neumann and Ulam, Nicholas Metropolis, suggested using the name Monte Carlo, which refers to the Monte Carlo Casino in Monaco where Ulam's uncle would borrow money from relative to gamble. Using list of "truly random" random numbers was extremely slow, but Von Neumann developed a way to calculate pseudorandom numbers using the middle square method. Thought this method has been criticized as crude, von Neumann was aware of this: he justified it as being faster than any other method at his disposal, and also noted that when it went awry it did so obviously, unlike methods that could be subtly incorrect.

Monte Carlo methods were central to the simulations required for the Manhattan project, though severely limited by computational tool at the time. In the (1950), they were used at Los Alamos for early work relating to the development of the hydrogen bomb, and become popularized in the fields of physics, physical chemistry and operations research. The Rand Corporation and U.S Air force were two of the major organizations responsible for funding disseminating information on Monte Carlo methods during this time and they began to find a wide application in many different fields.

Monte Carlo methods are employed in solving complex problems of probability, modeling and sampling. However, this research work intent to carry out the comparative analysis of some commonly used Monte Carlo algorithms, whereupon, statistical testing would be performed to check whether the numbers generated are uniformly distributed, and subsequently ensuring that the algorithms satisfied other requirements; longer period,

and efficiency, which are necessary for a typical random number generator, and finally draw out conclusion, recommendations and suggest area for further research where possible.

1.2: Background of the Study

Monte Carlo simulation is an important numerical technique for studying a wide range of problems in the physical sciences. Being a probabilistic technique, it is relies heavily on the use of pseudo-random number generators. Many statistical tests have been developed to check for randomness, and in most cases the period of the generator can be calculated at least approximately (P.D. Coddington, 1994).

Quality of generators; what are good random numbers? A practical answer is the requirements that the numbers should meet "all" aims or rather pass as many tests as possible. The requirements on good numbers generators can roughly be divided in three (3) groups.

The first requirement is that of a large period. In view of linear congruential generator (LCG), the number M must be as large as possible, because small set of numbers makes the outcome easier to predict-contrast to randomness this leads to select M close to the largest integer machine number. But a period p close to M is only achieved if a and b are chosen to properly, criteria for relations among M, p, a, b have been derived by number theoretic arguments (Seydel, R. 2012). For 32- bit computers, a common choice has been $M = 2^{31} - 1$, a = 16807, b=0.

A second group of requirement is the statistical tests that check whether the numbers are distributed as intended. Another slightly more involved test checks how well the probability distribution is approximated.

Random number generators (RNGs) are divided in two classes: True random number generators (TRNGs) and Pseudo-Random Number Generators (PRNGs).

1.3: Statement of Problem

A reliable random number generator is critical for the success of Monte Carlo problem. This is particularly important for Monte Carlo simulations which typically

involve the use of literally millions of random numbers. If the numbers are poorly chosen, that is if they show nonrandom behavior over relatively short interval, the integrity of the method is severely compromised. In real life, it is very easy to generate truly random number; in contrast, it is impossible to conceive an algorithm that results in purely random numbers because by definition, an algorithm and the computer that executes it is deterministic.

For instance, flipping coins and rolling dice are ways entropy could be obtained for a true random number generator (TRNGs), although the rate at which random numbers could be produced is restricted. Low production rate is a problem that plagues most TRNGs. Another major problem of these generators is that they are mostly hardware driven, this can make them more expensive to implement especially if necessary device is not commonly used.

It is also means that the generators are vulnerable to physical attacks that can bias the number sequences. Finally, even when there are no attackers present, physical devices can typically wear out overtime and error in their construction can be naturally bias the sequence produced (Sunar, Martin and Stinson, 2006).

1.4: Aim of the study

This research work is aimed at testing the randomness of commonly used Monte Carlo algorithms; linear congruential generator, lagged Fibonacci generator and feedback shift register, as well as conducting performance analysis and categorized of these algorithms based on their suitable areas of applications.

1.5: Objective of the Study

The generation of random numbers on a computer is a notoriously difficult problem. An ideal random number generator would provide numbers that are uniformly distributed, uncorrelated, satisfy any statistical test of randomness, have a large period of repetition, can be change by adjusting an initial “seed ”value, are repeatable, portable and can be generated rapidly using minimal computer memory.

Random number generators provided by computer vendors or recommended in computer science texts often have been (an unfortunately continue to be) of poor quality. Even generators that perform well in standard statistical tests for randomness may be unreliable in certain application. The motives of this project work are:

i. To test the randomness of some commonly used PRNGs
ii. To conduct a performance analysis on PRNGs algorithms.
iii. To classify PRNGs based on suitable areas of applications.

1.6: Significance of the study

The idea of constructing a system that produces randomness can seems like contradiction, decade of research has refine the art. On the other end of the spectrum, mathematicians and cryptographers have developed many algorithms that are unpredictable under certain circumstances.

In this context it is desirable to developed means to categories suitable generators for a domain or application in question. For instance, random number needed for a cryptographic system, must ensure that the sequence can neither be discovered nor repeated; otherwise, attackers would break in the system. The major importance of high performance and high quality randomness generators in cryptography attracts an increasing interest from the research community, but unfortunately, this tendency is only vaguely reflected in the community of security application developers, thus many cryptographic systems, in lack of a thorough analysis of the randomness source and of quality generators (Kingo Marton, 2011).

1.7: Scope of the study

This project work is meant to carry out a thorough test and performance analysis of pseudo-random number (PRNGs) algorithms such as:

i. Linear congruential generator (LCG)
ii. Lagged Fibonacci generator (LFG)
iii. Feedback shift register (Mersenne Twister) etc.

Nevertheless, C/C++ programming language would be used to implement those algorithms because of it fast execution and timing feature.

1.8: Definition of Terms

i. Null Hypothesis: This is a research hypothesis about the parameter (s) that we wish to support and is denoted by $H_{0.}$

ii. Alternative Hypothesis: This is a research that converse the null hypothesis and it is denoted by H_1.

iii. Period: is the time taken for Monte Carlo algorithm to repeat the previous sequence produced.

iv. Deterministic: is an event that is predictable.

v. Non-deterministic: is an event that is not predictable.

vi. Seed: value chosen as initial value to the algorithm, it is often denoted by X_0.

vii. Recursive: a procedure that invoke itself to return a value.

viii. Uniform distribution number: a set of number is said to be uniformly distributed if and only if each element is equally likely probable.

ix. Test Statistics: The decision to reject or accept the null hypothesis is based on information contained in a sample drawn from the population of interest.

x. Rejection region: is a region on a chi-square curve, if a test statistics assumes a value in the rejection region, then the null hypothesis is rejected.

xi. Acceptance region: is a region on a chi-square curve, if a test statistics assumes a value in the acceptance region alternative hypothesis is considered.

CHAPTER TWO

REVIEW OF RELATED LITERATURE

2.1: INTRODUCTION

"……*Random numbers should not be generated with a method chosen at random, some theory should."*-Donald E. Knuth (1997)

This chapter would review some literature concern that are related to comparative analysis of random number generators and explain in details the techniques involved. Randomness is a crucial resource for cryptography, and random numbers generators are critical building blocks of almost all cryptographic systems. Therefore, random number generation is one of the key parts of secure communication.

Problematic random number generation process may result in breaking the encrypted communication channel, because the encryption keys are obtained by using random numbers. For computers and smart devices, generation of random numbers is handled by operating systems because of the fact that obtaining truly random numbers from physical source is a costly method. Digital devices are fully deterministic machines but unpredictability is still required for cryptography, security, randomized algorithms, scheduling and networking; therefore some, modifications and additions are needed in other to construct random number generators using those machines (SerkenSaritas, 2013).

2.2: Desirable features of random numbers

The following are desirable attributes of random numbers:

1. The random number should be uniformly distributed.
2. They should be statistically independent.
3. The period of the random number should be sufficiently larger than the desired length for a particular application.
4. The generation of random number should be faster.

2.3: Methods of Random numbers

2.3.1: Middle-Square Method

The middle square method was proposed by John Von Neumann and Nicholas metropolis in 1946. In this method of random number generation, an initial seed is assumed and that number is squared. The middle four digits of the squared value are taken as the first random number. Next, the random number which is generated most recently is again squared and the middle most digits of this squared value are assumed as the next random number and so on.

This method is demonstrated as shown in the table below assuming the initial seed as 8297:

Table 2.1: Example Random Numbers Using Middle Square Method

S/N	n (Four digits)	n^2
1	8297	68840209
2	8402	70593604
3	5936	35236096
4	2360	05569600
5	5696	32444416
6	4444`	19749136
7	7491	56115081
8	1150	01322500
9	3225	10400625
10	4006	16048036

The recipe for the middle square method can be described below:

Step 1: Input a four digit number, n and set the value of counter C to 1

Step 2: Square the four digit number n

Step 3: Store the square of n into a variable P.

Step 4: Is the number of digit in n equals to 8? Else add necessary number of Zeros to the left of P so as to have a total of eight characters in P

Step 5: Select the middle four characters in P and store these four characters in the variable n, Treat the value of n as the first counter variable.

Step 6: C = C + 1

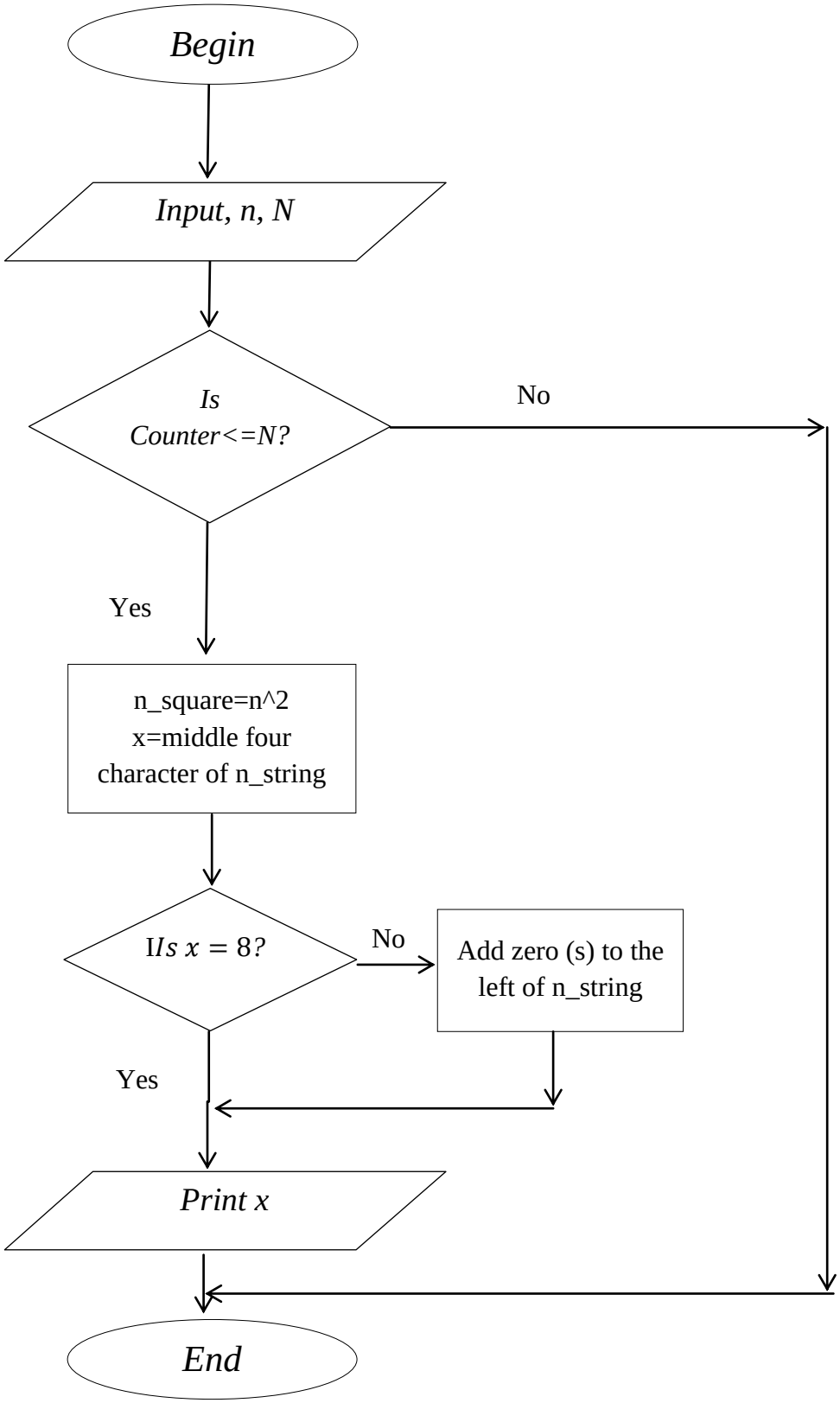
Step 7: If C is less than or equal to the required number of random numbers, then go to step 2, else go to step 8.

Step 8: Stop

Pseudo code for middle square method:

```
Input n /*enter four  digit number*/
Input N /* enter the number of random number needed to produced*/
For Counter= 1 to n do
{
   n_square = n^2
   n_string=n_square /*convert n_square in n_string */
   Count the number of characters (x) in string
   If x<8 then do
{
   Add (8 – x) Zeros to the left of n_string
 }
  x = middle four character on n_string
  n = x
 Print I, n
}
Stop
```

Flow charting representation of the middle square method:



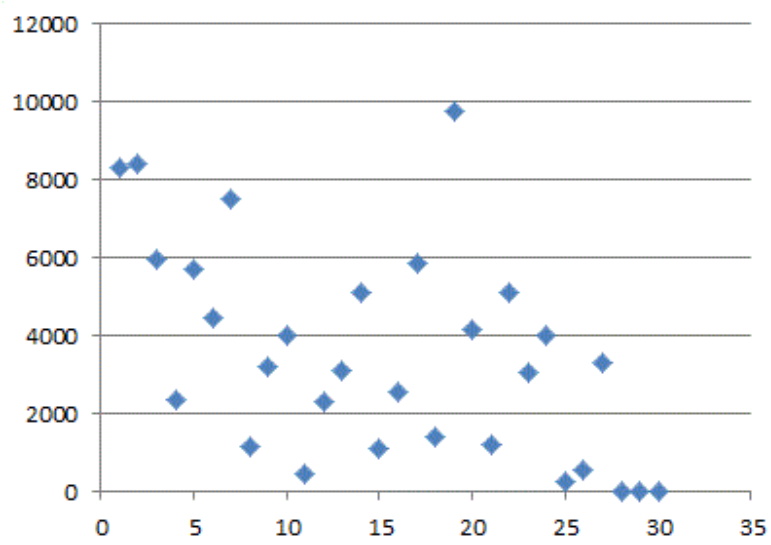


Fig 2.1: Middle Square Method

Limitations of Middle Square Method:

1. It is relatively slow
2. Statistically unsatisfactory
3. Sample of random numbers may be too short
4. There is no relationship between the initial seed and the length of the sequence numbers.

2.3.2: The Linear Congruential Method

By far the most popular random number generators in use today are special cases of the following scheme, introduced by D.H. Lehmer in (1949). We choice four magic integers:

N, the modulus; $0 \leq N$

r, the multiplier; $0 \leq r < N$

s, the increment ; $0 \leq s < N$

x_0, the starting value (seed); $0 \leq x_0 < N$

The desired sequence of random numbers (x_n) is obtained by setting

x_{n+1} = (rx_n + s) $mod\ N\ n \geq 0$ …… (1)

This is called a linear congruential sequence. Taking the remainder mod m is somewhat like determining where a ball will land in a spinning roulette wheel.

The notation modN means that the expression on the right of the equation is divided by and replace with the remainder. To understand the mechanics let consider the following simple example:

The sequence obtained when N=10 and x_0 = r = s = 7 is

$$x_1 = (7 * 7 + 7) mod 10 = 56\ mod 10 = 6$$

$$x_2 = (6 * 7 + 7) mod 10 = 49 mod 10 = 9$$

$$x_3 = (9 * 7 + 7) mod 10 = 70 mod 10 = 0$$

$$x_4 = (0 * 7 + 7) mod 10 = 7\ mod 10 = 7$$

We have the sequence 7, 6, 9, 0, 7, 6, 9, 0… …… (2)

As this example shows, the sequence is not always "random" for all choices of N, x_0 , r , and s ; the principles of choosing the magic numbers appropriately will be investigated carefully in later part of this chapter.

The above examples (2) illustrate the fact that the linear congruential sequence always get into a loop; there is ultimately a cycle of numbers that is repeated endlessly; this property is common to all sequences having the general form $x_{n+1} = f(x_n)$, when f transforms a finite set into itself. The repeating cycle is called the period; sequence generated above has a period of length 4.

A useful sequence will of course have a relatively long period. The special case when the increment s = 0 deserves explicit mention, since the number generation process is a little faster when s=0 than it is when s ≠ 0. We shall see later that the restriction s = 0 cut down the length of the period of the sequence, but it is still possible to make the period reasonable long.

2.3.2.1: Properties of Linear Congruential Generators

All of the pseudorandom generators examined in this project are congruential generators where each term is defined recursively in terms of k immediately preceding terms. We call this type of generator a recursive congruential generator, and it is expressed mathematically as

$$x_n = f(x_{n-1,}x_{n-2,} \dots \dots x_{n-k,})modN$$

Note the f need not be linear and need not used all k terms. We do assume that k is a small as possible, so x_n depend on $x_{n-k.}$ For example, in an LCG, f is defined by a$x_{n-k.}$ for example in an LCG, f is defined by ax_{n-1} + c, and K=1. However, f can be an arbitrary function, an x_n = ax_{n-1} + $(x_{n-7})^2$ is a perfectly good recursive congruential generator. This theorem provides an upper bound on the period, which is formalized in the subsequent corollary.

2.3.2.2: Merit of Linear Congruential Generators

i. Linear congruential generators are simple to implement.
ii. Linear congruential generators are faster and efficient.
iii. Linear congruential generators require minimal memory

2.3.2.3: Limitations of Linear Congruential Generators

i. Linear congruential generators should not be used for applications where high quality randomness is critical.
ii. Linear congruential generators produces pattern that are predictable.
iii. A further problem of the linear congruential generators is that of shorter period.
iv. Linear congruential generators cannot be used for cryptographic applications.

2.3.2.4: Applications of Linear Congruential Generators

i. Linear congruential generators are widely used simulation and Monte Carlo calculation because they are very fast.
ii. Because linear congruential generators have state space, they remain attractive for use in parallel computing environment.
iii. Nevertheless, linear congruential generators may be good option. For instance, in an embedded system because the amount of memory available is often limited

2.3.3: Additive Fibonacci Generator

The simplest sequence in which x_{n-1} depends on more than one of the preceding values of the Fibonacci sequence;

$x_{n+1} = (x_n + x_{n-1}) \ mod \ N$ ..... (2)

This generator was considered in the early 1950s, and it usually gives a period length greater than m; but tests have shown that the numbers produced by the Fibonacci recurrence (2) are definitely not satisfactorily random.

2.3.3.1: Merit of Lagged Fibonacci Generator

The primary aimed of this generator is an improvement of on the standard linear congruential generators, hence minimizing their shortcomings.

2.3.3.2: Problem with Lagged Fibonacci Generator

i. The initialization of Fibonacci generator is very complex problem. The output of the Fibonacci generators is very sensitive to initial conditions, and statistical defects may appear initially but also periodically in the output sequence unless extreme is taken.
ii. Another potential problem with Fibonacci generators is that the mathematical theory behind them is incomplete, making it necessary to rely on statistical tests rather than theoretical performance. (G. Prasada Rao, 2010).

2.3.3.3: Application of Lagged Fibonacci Generators

i. The oracle Database implement lagged Fibonacci generator in it DBMS_Random Package (Available in oracle 8 and newer version)

ii. The "Pocket Dungeon" on www.BoardGameGeek.com uses it as an alternative ("Stealth) dice generator.

iii. Freecive uses a lagged Fibonacci generator for its random number generator.

2.3.4: Feedback Shift Register

At times it can be easier to describe pseudo random number generators in terms of hardware instead of their mathematical form. This is the case with feedback shift registers; these are best visualized as string of n bits sitting inside a register in hardware. An even number of positions are selected; such as indexes five, seven, nine and n. these generators operate by performing XOR on the bits at these positions, taking the result as the new leftmost bit, and shifting the rest of the string by the right (Dichtl, 2003). The bit that gets shifted out is the next random bit in the generator's output. Mathematically, this algorithm can be written as:

$$x^p = x_1^{p-t} + \cdots + x_n^{p-t} + x^0 \ldots\ldots\ldots\ldots\ldots\ldots (3)$$

Where x is the bit string of length n, and t decides the index positions (Sunar, Martin, & Stinson, 2006). An advantage of these types of generators is that entropy can be easily added into the system, simply by including the new information in the XOR operation.

Without entropy being introduced, shift registers have a period of 2^n-1, because zero will never be the result.

The Mersenne twister is a very popular and widely used example of feedback shift register in simulation and modeling (Nishimora, 2000). It is a good first choice when picking a pseudo random number generator. The Mersenne twister can be classified as a twisted generalized feedback shift registers (TGFSR), which has algorithms more tightly to matrices than string. Adaptations exist both for making the generator faster

and making secure enough for cryptography. The benefits of these generators are rapid number generation, highly random sequence and large period.

2.3.4.1: Merit of Feedback Shift Register

i. It has a very long period of 2^{19937}-1
ii. It is more accurate
iii. It passes numerous tests for statistical randomness, including the Die-hard test.

2.3.4.2: Shortcoming of Feedback Shift Register

i. It is very complex to implement
ii. It is not elegant
iii. This generator is not very sensitive to initialization and can take long time to recover from a zero excess initial state.

2.3.4.3: Applications of Feedback Shift Register

The feedback shift register is a default PRNG is for the following software system:

i. Microsoft Excel
ii. Microsoft visual C++
iii. MatLab
iv. Mathematica etc.

CHAPTER THREE
RESEARCH METHODOLOGY

3.1: INTRODUTION

"…….Anyone who considers arithmetical methods of producing random digit is, of course, in a state of sin." John Von Neumann (1951).

In this chapter a brief but thorough description of the activities or event in respect of which data will be gathered will be provided. Assurance of statistical quality of a generator can only come from testing output against theoretical results. These can be either specific result of the problem being studied, or more generally results for the generator itself (Donald E. Knuth, 1997). Specifically, this project work would use data collected from a primary source collected from the program output for linear congruential generator, lagged Fibonacci generator and feedback shift register respectively.

3.2: Statistical Test for Random Number Generator

One of the approaches to testing random number generators is leveraging the wide array of statistical formulas. With this approach, each test examines a different quality that a random number should have. For example, random number generator would go through a chi-squared test to ensure a uniform distribution, and then reverse arrangements test to see if the sequence contained any trends.

Confidence in the generator's randomness is only gained after it passes on entire suite of tests which comes at it passes an entire suite of tests which come from different directions. These tests should be run on more than just a single sequence to ensure that the test results are accurate. Making the act of testing more difficult is the fact that failing a test does not indicate that a generator is not random. When output are truly random, then there will be some isolated sequences produced that appears non-random (Haahr, 2011). Tests need to be picked carefully and tailored to context generators are needed in. the chi-squared, runs, next bit , and matrix based tests will be examined because of their popularity.

3.2.1: Types of Statistical Test

There are two types of statistical test for random numbers, namely; theoretical test and empirical test.

Theoretical Tests: evaluate the choices of m, a, and c without actually generating any number. The availability of theoretical of proof statistical properties is the most desirable situation. We are then left only with concern about the actual implementation of the generator. Most generators have proof of the uniformity of the output of the full sequence and generally limited subsets before they are even considered. If not worth even starting with the generator, Paired correlations and low order independent proofs are not available for many of the better studied generators. Only congruential generator and generators with lattice structure have higher order theoretical tests available in the form of the spectral test and discrepancy test respectively.

Empirical Tests: applied to actual sequence of numbers produced. (Our emphasis), the limited availability of theoretical tests makes empirical tests necessary. Random number generators also must be tested empirically. Good discussion of empirical tests can be found both in Knuth [49] and in Kennedy and Gentle [47]. There is no limit to the possibilities for testing a sequence. Effectively, every computation using random number generation with theoretical results available is an empirical test.

Knuth describes in [49] 12 separate empirical tests. Some are common statistic tests such as the chi-squared or Kolmogorov-Smirnov tests for uniformity. Other such as Serial, Run, Gaps, Poker and Birthday spacing are based on specific scenarios aimed at detecting patterns in the numbers generated discrete probability theory available from which to construct testing framework and decision rule (David Zeitler, 2001).

3.2.2: Chi-Squared Test

The chi-square (χ^2) test was initially published by Karl Pearson in 1900. Pearson's original notation in explaining the theory behind the test included use of the x^2 symbol; hence the name. This test can be used in many situations and basically, when given an outcome of an experiment, can give an approximate probability as to how likely that outcome is (Donald E. Knuth, 1997).

The chi-squared test is used to ensure that the available numbers are uniformly utilized in a sequence. This test is easy to understand and set up, so it is commonly used (Foley, 2001). The formula for the test is:

$$\chi^2 = \sum_{i=1}^{n} \frac{(O-E)^2}{E}$$

Where:

$$\chi^2 = chi - square\ value$$
$$O = Observed\ frequency$$
$$E = Expected\ frequency$$
n= the number of classes
$$\sum = Summation$$

The expected frequency is given by $E = \frac{N}{n}$, where N is the total number of observation.

For example, if the random numbers were scattered one through six, then there would be six numbers of Expected frequencies, and number of times each sequence appears in the sequence becomes the value of O. When the resulting value is above the chosen significance level, then it can be said that the values in the sequence are uniformly distributed. Because this test is simple, it can be run many times on difference sequences with relative ease to increase the chance of its accuracy (Foley, 2001).

Let apply the chi square testing to find out whether a die used to obtain the result of the experiment below is unbiased:

Table 3.1: Result of an experiment for rolling a die

#	Observed Frequency(O)	Expected Frequency (E)
1	44	34
2	46	34
3	30	34
s	38	34
5	24	34
6	22	34
Total	204	

$E(1) = 205\ (^1/_6) = 34 E(2) = 205\ (^1/_6) = 34 \quad E(3) = 205\ (^1/_6) = 34$

$E(4) = 205\left({}^{1}/_{6}\right) = 34 \quad E(5) = 205\left({}^{1}/_{6}\right) = 34 \quad E(6) = 205\left({}^{1}/_{6}\right) = 34$

Chi square testing steps:

i. State null (H_o) and alternative (H_1) hypothesis
ii. Choose level of significance (α)
iii. Find the critical values
iv. Find test statistic
v. Draw your conclusion

Let apply these steps to test the fairness of the die used to obtain the data in the table 3.1.

i. H_o : The die is fair , H_1 : The die is not fair
ii. The level of significance

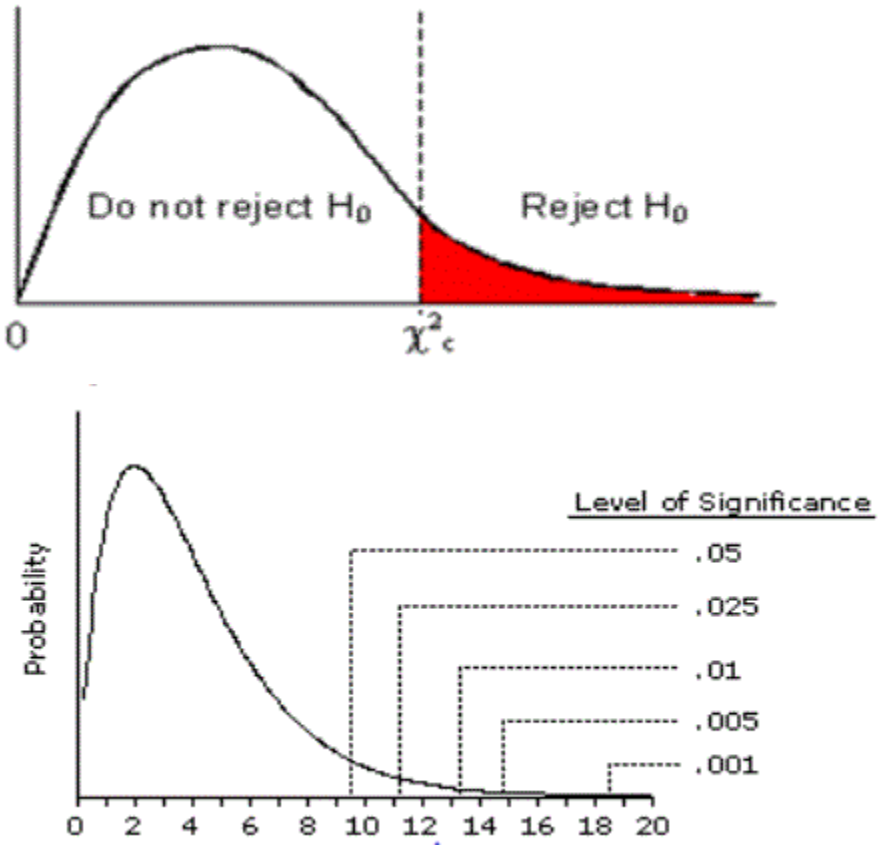
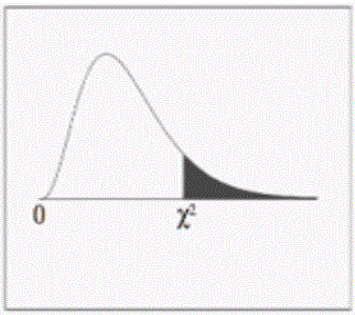


Figure 3.1: Chi-Square Distribution curve (Source: Statistical testing for randomness, Masaryk university in Brno, faculty of informatics, by Jan Krhovjak.)

In this case we will choose the level of significance to be .01, therefore α= .01

iii. The critical value (CV) is the point that separates rejection region and other region in the curve. It is denoted by χ^2_c

Table 3.2: Table of Chi-Square (χ^2) distribution (Source: YouTube – Chi-square test explained)



The shaded area is equal to α for $\chi^2 = \chi^2_\alpha$.

df	$\chi^2_{.995}$	$\chi^2_{.990}$	$\chi^2_{.975}$	$\chi^2_{.950}$	$\chi^2_{.900}$	$\chi^2_{.100}$	$\chi^2_{.050}$	$\chi^2_{.025}$	$\chi^2_{.010}$	$\chi^2_{.005}$
1	0.000	0.000	0.001	0.004	0.016	2.706	3.841	5.024	6.635	7.879
2	0.010	0.020	0.051	0.103	0.211	4.605	5.991	7.378	9.210	10.597
3	0.072	0.115	0.216	0.352	0.584	6.251	7.815	9.348	11.345	12.838
4	0.207	0.297	0.484	0.711	1.064	7.779	9.488	11.143	13.277	14.860
5	0.412	0.554	0.831	1.145	1.610	9.236	11.070	12.833	15.086	16.750

Note: the degree of frequency ***(df)*** is always less than the possible outcome

From table 3.2 have the critical value (CV) to be equals to 15.086

iv. Find test statistics, using ;

$$\chi^2 = \sum_{i=1}^{n} \frac{(O-E)^2}{E}$$

We have:

$$\chi^2 = \sum_{i=1}^{6} \frac{(O-E)^2}{E}$$

$$\chi^2 = \frac{(44-34)^2}{34} + \frac{(46-34)^2}{34} + \frac{(30-34)^2}{34} + \frac{(38-34)^2}{34} + \frac{(24-34)^2}{34} + \frac{(22-34)^2}{34} = 15.29$$

v. **Conclusion**:

Since value of χ^2 is greater than that of critical value i.e. 15.29 > 15.086

This indicates the die is not fair, hence the alternative hypothesis (H_1) is accepted.

3.2.2.1: Application of Chi-Square

Other than probability theories, in cryptanalysis, chi-squared test is used to compare the distribution of plain text and (possibly) decrypted cipher text. The lowest value of the test means that the decryption was successful. With high probability, this method can be generalized for solving modern cryptographic problems.

3.2.3: The Kolmogorov-Smirnov Test

The Kolmogorov-Smirnov (KS) test has its origins in a 1933 paper by A.N Kolmogorov, and N.V Smirnov suggested some improvements in 1939, leading to the joint name of test [Knuth 54]. The KS test is useful in areas where the chi-square test is not, and can also be used in conjunction with the chi-Square test. Because of this, it too is a foundational test for many of the empirical tests to follow, and will be examined here.

We first introduce a common function in probability theory. Given a random variable$x$, the *cumulative distribution function (cdf)* F (x), is defined as

F (x) = Probability that (X $\leq x$)

Note that any F (x) has a range of [0.1] (sometimes asymptotically) and will always be increasing (or remain constant over some intervals) as x increases from -∞ to +∞ [Knuth 47].

Kolmogorov-Smirnov test compares the continuous *(cdf),* F (x), of the uniform distribution with the empirical *(cdf),* S_N (x), of the N sample observations.

We know: F (x) = x $0 \leq x \leq 1$, if the sample from the Random number generator (RNG) is$R_1, R_2, \ldots\ldots\ldots R_N,$ then the empirical *CDF,* $S_N(x)$is:

$$S_N(x) = \frac{Number\ of\ R_i\ where\ R_i \leq x}{N}$$

Based on the statistic:

D= max|F (x) - S_N (x), the sampling distribution of D is known.

Kolmogorov-Smirnov test consist of the following steps:

Step 1: Rank the data from smallest to largest $R_1 \leq R_2 \leq \cdots \leq R_N$

Step 2: Compute

$$D^+ = \max_{1 \leq i \leq N} \{ \frac{i}{N} - R_{(i)} \}$$

$$D^- = \max_{1 \leq i \leq N} \{ R_{(i)} - \frac{i-1}{N} \}$$

Step 3: Get Dα for the significant level α.

Step 4: if D≤ Dα accept, otherwise reject H_0

Example: Suppose N=5 numbers; 0.44, 0.81, 0.14, 0.05, 0.93.

	i	1	2	3	4	5
Step 1:	$R_{(i)}$	0.05	0.14	0.44	0.81	0.93
	i/N	0.20	0.40	0.60	0.80	1.00
Step 2:	$i/N - R_{(i)}$	0.15	0.26	0.16	-	0.07
	$R_{(i)} - (i\text{-}1)/N$	0.05	-	0.04	0.21	0.13

Arrange $R_{(i)}$ from smallest to largest

$D^+ = max\{i/N - R_{(i)}\}$

$D^- = max\{R_{(i)} - (i\text{-}1)/N\}$

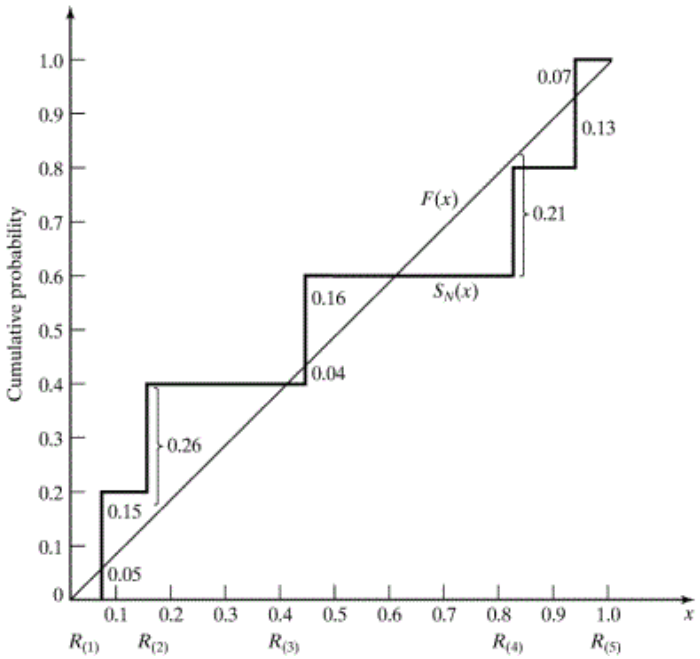


Figure 3.2: Demonstration for Kolmogorov Smirnov Test

Step 3: $D = \max (D^+, D^-) = 0.26$

Step 4: for α=0.05, $D\alpha$= 0.565>D=0.26

Hence, H_0 is not rejected.

Table 3.3: Difference between Kolmogorov-Smirnov and Chi-Square Test

K-S Test	**Chi-square Test**
1. Uses small samples	Uses large samples
2. Continues Distribution	Discrete distribution
3. Differentiate between observed and expected frequencies	Differences between observed and hypothesized probabilities
4. Uses each observation in sample without any grouping	Group observation into small number of cell
5. Produced exact values	Produced approximate values

3.2.4: The Runs Test

An important trait for a random sequence is that it does not contain patterns. The test of runs above and below the median can be used to verify this property (Foley, 2001). In this test, the

number of runs, or streaks of numbers, above or below the median value are counted. If the random sequence has an upward or downward trend, or some kind of cyclical pattern, the test of runs will pick up on it. The total number of runs and the number of values above and below the median are recorded from the sequence.

Then, these values are used to compute z-score to determine if numbers are appearing in a random order. The formula for the score is:

$Z= ((u \pm 0.5) - MEAN_U) / \delta_U$

Where δ denote the standard deviation of the sequence (Foley, 2001), just like the chi-squared test, a test for runs is easy to implement, and can be run frequently.

3.2.5: Next Bit Test

When testing pseudo random number generators for cryptography applications, the next bit test is a staple. In its theoretical form, the next bit test declares that a generator is not random if given every number is the generated sequence up to that point there is an algorithm that can predict the next bit produced with significantly greater than 50% accuracy (Lavasani & Eghlidos, 2009). This definition makes the next bit test virtually impossible to implement, because it would require trying every conceivable algorithm to predict the next bit. Instead, it can be used after a pattern is discovered to cement the fact that a generator is insecure. Several attempts have been made to alter the next bit test so that it can be used as an actual test.

The universal next bit test developed in 1996 was the first to allow the next bit test to be administered, but it was shown that this test would pass non-random generators. Later, the practical next bit test was developed and was shown to be as accurate as the National Institute of Standard and Technology (NIST) test suite at the time; it was if not more so (Lavasani & Eghlidos, 2009). However, this test required a large amount of resources to run, limiting its usefulness. The next bit test remains relevant in cryptography because it was been proven that if a generator can pass the theoretical next bit test, then it will pass every other statistical test for randomness.

3.2.6: National Institute of Standard and Technology (NIST) test

Of the available suite for testing random number generators, the NIST suite reigns as the industry standard (Kenny, 2005). The NIST suite was designed to test next bit sequences, with the idea

that passing all NIST test means that the generator is fit for cryptographic purposes. Even new true random number generators have their preliminary results run through the NIST battery to demonstrate their potential (Li, Wang, & Zhang, 2010). The NIST suite contains fifteen well documented statistical tests (NIST.gov, 2017). Because cryptography has the most stringent requirements for randomness out of all the categories, a generator that passes the NIST suite is also random enough for all other applications. However, when a generator fails the NIST suite, it could still be random enough to serve in areas such as gaming and simulation, since the consequences of using less than perfectly random information is small. NIST does not look at factors such as rate of production, so passing the NIST suite should not be the only factor when determining a generator's quality.

3.2.7: Diehard Test

Another widely used suite of random number tests is known as Diehard. This suite was invented by George Marsaglia in 1995 (Kenney, 2005). It was made to be an update for the original random number test suite, Knuth. Knuth is named after Donald Knuth and was published in the 1969 book *The Art of Computer Programming, Volume 2.* Knuth's tests were designed before cryptography become major industry, and the suite was later considered to be too easy to pass for situations where vast quantities of random numbers were needed. Diehard was designed to be more difficult to pass than Knuth's suite, fulfilling the role of a general-purpose battery for detecting non-randomness. All of the tests are available free online, so they can be easily used to test any number sequence (Marsaglia, 2005). The Diehard suite has not been updated since its inception in 1995, but it stills a widely used test suite (Kenny, 2005).

3.3: Complexity Analysis

Analyzing algorithm has come to mean predicting the resources that the algorithm requires occasionally, resources such as memory, communication bandwidth or computer hardware is primary concern. But most often it is computational time that we want to measure. Generally, by analyzing several candidate algorithms for a problem, we can identify a most efficient one. Such analysis may be indicating more than one viable candidate, but we can often discard several inferior algorithms in the process (Thomas H. Cormen *et al.*, 2009).

In order to use probabilistic analysis, we need to know something about the distribution. Even if we do know something about the distribution, we may not be able to model this knowledge computationally. Yet we often can use probability and randomness as a tool for algorithm design

and analysis, by making the behavior of part of the algorithm random. More generally we call algorithm randomized if its behavior is determined not only by its input but also by values produced by a random number generator. We shall assume that we have at our disposal a random number generator RANDOM. A call to RANDOM (a, b) returns an integer between a and b, inclusive, with each such integer being equally likely. For example, RANDOM (0, 1) produces 0 with probability 1/2, and it produces 1 with probability 1/2. A call to RANDOM (3,7) return either 3,4,5,6,or 7, each with probability 1/5. Each integer returned by RANDOM is independent of the integers returned on previous calls.

When analyzing the running time of a randomized algorithm, we take the expectation of the running time over the distribution of values returned by the random number generator. We distinguish thus algorithms from those in which the input is random by referring to the running time of randomized algorithm as an expected running time. In general, we discuss the average-case running time when the probability distribution is over the inputs to the algorithm and we discuss the expected running time when the algorithm itself make random choices.

3.3.1: Asymptotic notation

Big-Oh (O) notation: $f(n) = O(g(n))$: is read $f(n)$ is upper-bounded by $g(n)$.

Definition: given two non-negative functions $f(n)$and $g(n)$, we say that $f(n)$ is $O(g(n))$ iff there exist positive constant$C\ and\ n_0$, such that $f(n) \leq C.g(n)\ for\ n \geq n_0$. therefore, big-oh (O) notation is used to expressed upper-bound on running time. Graphically we can have:

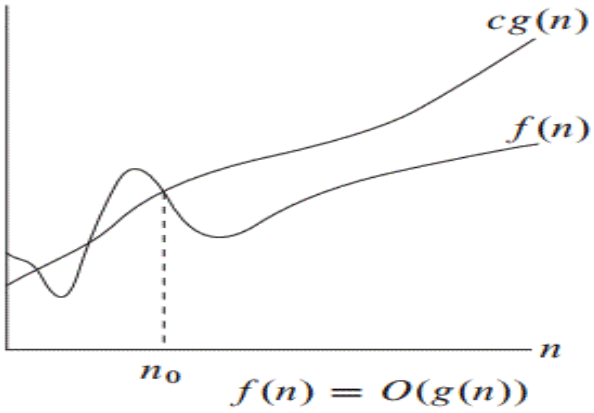


Fig 3.3: Big-O Notation Graph.

Example:

$f(n) = 6n^3 + 3n^2 + 3$, find O notation.

Solution: By dropping lower order terms and ignoring leading coefficient we have $f(n) = O(n^3)$

Big Omega (Ω): $f(n) = \Omega(g(n))$: is read $f(n)$ is lower bounded by$g(n)$.

Definition: suppose $f(n)\ and\ g(n)$ are two non-negative functions of size(n), say that $f(n)$ is $\Omega(g(n))$ iff there exist positive constants $C\ and\ n_0$ such that $f(n) \leq C.g(n)\ for\ n \geq n_0$. Big theta (Ω)is used to expressed lower bound on running time. Graphically we have:

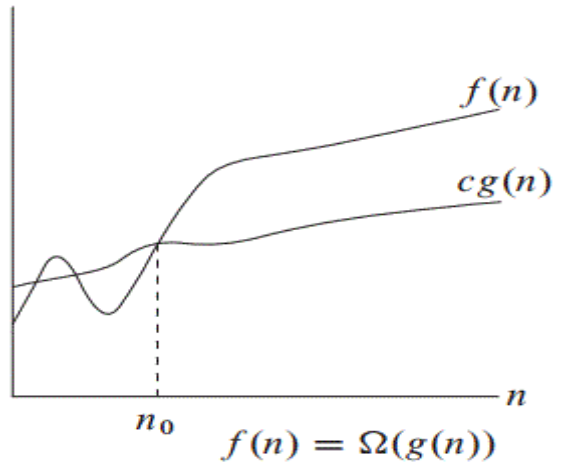


Fig 3.4: Big Omega Notation Graph

Example:

Let $(n) = 2n + 3\ and\ g(n) = n$, check whether $f(n) = \Omega(g(n))$

Solution:

$$c = 1, \qquad n = 1$$

$$2n + 3 \geq 1 \times n$$

$$2(1) + 3 \geq 1 \times 1$$

$$\therefore \quad f(n) = \Omega(g(n))$$

Big Theta - ($\boldsymbol{\theta}$): $f(n) = \theta(g(n))$ is read $f(n)$ is tight bounded by $g(n)$.

Definition: suppose $f(n)$ and $g(n)$ are two non-negative function of size (n), say that $f(n)$ is $\theta(g(n))$ iff there exist constants c_1, c_2 and n_0 such that $c_1.g(n) \leq f(n) \leq c_2.g(n)$ for $n \geq n_0$. Graphically:

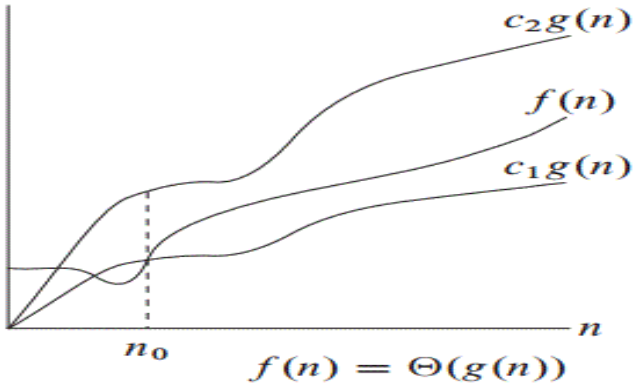


Fig 3.5: Big theta Notation Graph.

CHAPTER FOUR

DATA PRESENTATION AND ANALYSIS

4.1: INTROCTION

"…Probably… Cannot be justified, but should merely be judged by their results. Some statistical study of the digits generated by a given recipe should be made, but exhaustive tests are impractical." – John Von Neumann (1951).

This chapter deals with data presentation, analysis and interpretation of the results based on proper collation, analysis and synthesis, including test of hypotheses and discussion of results so achieved (Adefila, J.J, 2008).

4.2: Simulation Result

Table 4.1: Sample sequence from three random number generators

Numbers generated by each generator after fifty (50) iterations			
S/N	**LCG**	**LFG**	**FSR**
1	029833	000000	18718
2	304563	000001	30956
3	067763	633571	30273
4	646878	879214	1780
5	225968	527550	28148
6	272643	421529	5151
7	921033	949079	23755
8	954278	385373	11053
9	147383	349217	30980
10	477878	734590	3579
11	222203	098572	5966
12	140868	833162	6045
13	249848	931734	8972

14	123208	779661	30747
15	616983	726160	32139
16	164878	520586	24398
17	101953	261511	9215
18	858293	782097	6000
19	728848	058373	14912
20	139213	840470	21849
21	191923	898843	18592
22	066303	754078	29573
23	595778	667689	11607
24	407938	436529	13902
25	730183	118980	1616
26	185938	555509	27399
27	842063	674489	15966
28	166798	244763	214
29	947398	919253	292
30	891818	178780	29060
31	916988	112797	13111
32	812703	291577	16791
33	118433	404374	14395
34	449858	695951	23155
35	226913	115090	24337
36	305718	811041	35337
37	108188	926131	22104
38	091283	751937	19535
39	484843	692833	21877
40	466153	459535	15767
41	797238	167133	27750
42	562393	626668	5257
43	228193	783801	32730
44	350518	435234	20070

45	690953	243800	15926
46	783358	679034	12135
47	076593	922834	19605
48	955928	616633	21579
49	205133	554232	12039
50	528653	185630	23126
Seed	$a = 35$ $x = 387928$ $m = 985235$ $c = 245643$	# of terms=50 $x = 387928$ $m = 985235$ $y = 245643$	System Time (srand function)

4.3: Code Timing

Each random number generator was timed for fifty iterations. First, the default code timer from the standard c compiler was used to determine the start and end time for each generator. The generator was called repeatedly within the FOR loop, finally the time period was taking and recorded.

Table 4.2: Relative time taken by each generator to run

Time for, 50 Iterations in seconds		
Linear congruential generator	Lagged Fibonacci generator	Linear feedback shift register
55.942	64.428	8.053

`A generator will run at different speeds when it is run on different computers or different operating systems. A generator will event run at different speed on the same computer and same operating system. However, when all of the generators are run on the same system, the relative speeds of the generators can be estimated. These timings in table 4.1 were conducted on e machine E510 which contained a Celeron 32 bit processor, with 560 @ 2.13 GHz, 2128 MHz clock. The machine was running windows 7 operating system. The random number generators were compiled with Borland 5.5 C compiler.

The feedback shift register was by far the fastest of the three generators. The linear congruential generator was second which 42% slower than feedback shift register. The lagged Fibonacci generator come in third and was 53% slower than it two counter-parts.

4.4: Chi-square Test

This test involves producing sequence of 50 random integers between 0 and 50. This is accomplished by generating real numbers greater than 0.5 but less than 49.5 and then rounding the fractional part. These number streams are grouped into five difference categories each consist of ten integer numbers for easy computation. If this generator produces uniformly distributed numbers, one would expect to have about 10 occurrences for each integer. The actual frequency of each integer produced by the generator is the observed frequency for that integer. The difference between the observed and the expected frequency of each integer can be used to compute the chi- square statistic as follows:

$$\chi^2 = \sum_{i=1}^{R} \frac{(O_i - E_i)^2}{E_i}$$

Where R is the number of different random integers possible, O_i is the observed frequency of occurrence for the random integer i and $E_i$is the expected frequency of occurrence for the random number *i.* because the distribution of integers is expected to be uniform, the expected frequencies of occurrence for each random integer are equal. If N is the total number of observations, then the following equation can be used to computeE_i.

$$E_i = \frac{N}{R}$$

The E_i can then be replaced by the constant N/R, producing the following chi-square equation form:

$$\chi^2 = \frac{R}{N}\left(\sum_{i=1}^{R} \mathrm{O}_\mathrm{i}^2\right) - N$$

This equation provides a more efficient method of computing the chi-square. A large chi-square statistic indicates the random numbers are not uniformly distributed. In general, a smaller chi-square is considered preferable to a large chi-square. The one exception to this is when a chi-square statistic is very close to zero (0). This case may indicate that the random numbers are artificially too uniformly distributed.

Five chi-square statistics were produced for each of the three random number generators. Each of the 5 chi-square statistics were computed from difference sample of random numbers generated.

4.4.1: Chi-square Results

Table 4.3: Table of chi square results

CHI-SQUARE for 50 observations and 5 categories			
# of category	Linear Congruential Generator	Lagged Fibonacci Generator	Feedback Shift Register
1	74.32	134.37	93.75
2	230.64	43.57	55.85
3	238.37	41.42	94.24
4	58.58	83.18	114.31
5	167.76	42.68	97.73
Average	153.934	69.044	91.176

All of the generators seemed to produce acceptable chi-square statistics. The three chi-square value for the linear congruential and one for lagged Fibonacci generator looked a little high. This seemed to confirm the visual examination of number streams in the linear congruential generator and lagged Fibonacci generator were there appeared to be some initial clustering of random numbers. Clustering occurs when several consecutive random numbers in the generated stream are very close in value. Nevertheless, the chi-squares improve for the two generators in other categories of each sequence.

4.5: Chi-Square Independent Test Using Contingency Table

The chi-square independent test can be used to test the independent of two variables. For example, a random number generator may be check ensured that the digits do not appear to follow any regular pattern. Since there are so many possible patterns which could occur, it is impossible to test for all pattern. However, it is possible to test for some of the more obvious ones (Mark D. Meyerson, 1984). Samples of number generated by feedback shift register generator are used as an example below:

Table 4.4: Sample of random numbers produced by feedback shift register

# of category	Column 1	Column 2	Column 3	Column 4	Column 5
1	1	29	10	3	27
2	19	35	5	23	41
3	34	30	38	42	9

At $\alpha = 0.05$, we can conclude whether the numbers generated are independent to one another.

Step 1: State the hypothesis and identify the claim

H_0: The numbers generated are dependent on previous number so generated.

H_1: Number generated is independent to previous one.

If the null hypothesis is not rejected, the test means that number generated are not independent on previous one, otherwise, the number are independent. In other to test the null hypothesis using the chi-square independent test, one must compute the expected frequency, assuming that the null hypothesis is true. These frequencies are computed by using the observed frequencies.

4.5.1: The Contingency Table

When data are arranged in table form for the chi-square independence test, the table is called a contingency table. The table is made up of R rows and C columns.

The table 4.1 above has 3 rows and 5 columns. The contingency table is designated as a RxC (rows time's columns) table. In this case, R=3 and C=5; hence, this table is a 3x5 contingency table. Each block in the table is called a cell and is designated by its rows and column position. For example, the cell with number 29 is designated as $C_{1,2}$or Row1, and Column2.

The degrees of freedom for any contingency table are (rows –1) times (column – 1). That is; d.f=(R – 1) (C–1), in this case (5–1) (3–1) = $4x2$ = 8. The reason for this formula for d.f is that all expected values for each block (or cell) as shown next.

a. Find the sum of each row and each column, and find the grand total, as shown.

Table 4.5: Row and Column Sum of observed sample values

# of category	Column 1	Column 2	Column 3	Column 4	Column 5	Row Sums
1	1	29	10	3	27	Row 1 Sum=70
2	19	35	5	23	41	Row 2 Sum=123
3	34	30	38	42	9	Row 3 Sum=153
Column Sum	54	94	53	68	77	**Grand Total for Row sum= 346**

b. For each cell, multiply the corresponding row sum by column sum and divide by the grand total to get the expected value.

Expected value $= \frac{row\ sum\ x\ column\ sum}{grand\ total}$

For example, for $C_{1,2,}$ the expected value denoted by $E_{1,2}$is:

$E_{1,2} = \frac{94x70}{346} = 19.02$

≈ 19

For each cell the expected value is compute using the above procedure, therefore;

$E_{1,1} = \frac{54x70}{346} = 10.92$ ≈ 11

$E_{1,2} = \frac{94x70}{346} = 19.02$ ≈ 19

$E_{1,3} = \frac{53x70}{346} = 10.72$ ≈ 11

$E_{1,4} = \frac{68x70}{346} = 13.75$ ≈ 14

$E_{1,5} = \frac{77x70}{346} = 15.57$ ≈ 16

$E_{2,1} = \frac{54x123}{346} = 19.19$ ≈ 19

$E_{2,2} = \frac{35x123}{346} = 12.44$ ≈ 12

$E_{2,3} = \frac{53x123}{346} = 18.84$ ≈ 19

$E_{2,4} = \frac{68x123}{346} = 24.17$ $E_{2,5} = \frac{77x123}{346} = 27.37$

≈ 24 ≈ 27

$E_{3,1} = \frac{54x153}{346} = 23.87$ $E_{3,2} = \frac{94x153}{346} = 41.56$

≈ 24 ≈ 42

$E_{3,3} = \frac{53x153}{346} = 23.50$ $E_{3,4} = \frac{68x153}{346} = 30.06$

≈ 24 ≈ 30

$E_{3,5} = \frac{77x153}{346} = 34.04$

The expected values can now be placed in the corresponding cells along with the observed values as shown:

Table 4.6: Sample of observed values with associated expected values

# of category	Column 1	Column 2	Column 3	Column 4	Column 5
1	1(11)	29(19)	10(11)	3(14)	27(16)
2	19(19)	35(33)	5(19)	23(24)	41(27)
3	34(24)	30(41)	38(23)	42(30)	9(34)

Statistical Test: the formula for the test value for the independence is the same as the one used for the goodness of fit test.

$$\chi^2 = \sum_{i=1}^{n} \frac{(O-E)^2}{E}$$

$$\chi^2 = \frac{(1-11)^2}{11} + \frac{(29-19)^2}{19} + \frac{(10-11)^2}{11} + \frac{(3-14)^2}{14} + \frac{(27-16)^2}{16} + \frac{(19-19)^2}{19} + \frac{(35-33)^2}{33} + \frac{(5-19)^2}{19} + \frac{(23-24)^2}{24} + \frac{(41-27)^2}{27} + \frac{(34-24)^2}{24} + \frac{(30-41)^2}{41} + \frac{(38-23)^2}{23} + \frac{(42-30)^2}{30} + \frac{(9-34)^2}{34}$$

= 9.09 + 5.26 + 0.09 + 8.64 +7.56 + 0.00+0.12+10.31+ 0.04+7.25+4.16+ 2.95 +9.78+4.80+18.38

= 47.36

Step 3: Find the critical value, the critical value is 7.34, since the degree of freedom are $(5-1)(3-1) = 8$

Step 4: Make the decision, the decision is to accept the null hypothesis since 47.36> 7.34.

4.5.2: Independence Test Result

Below is Chi-square values for test of independence from random number generators under consideration. There are obtaining from 3X3 contingency table with .001 degree of freedom:

Table 4.7: Independence Test Result

# of category	Linear congruential generator	lagged Fibonacci generator	Linear feedback shift register
1	54.32	20.37	12.55
2	155.60	36.93	48.81
3	165.76	266.41	96.97
4	56.30	173.19	100.66
5	93.57	28.39	53.36

For the independence test, only the linear feedback shift register generators produce one acceptable chi-square statistics.

4.6: Application Areas of Random Numbers

4.6.1: Simulation

When a computer is being used to simulate natural phenomena, random numbers are required to make things realistic. Simulation covers many fields, from the study of nuclear physics (where particles are subject to random collisions) to operation research (where people come into, say, an airport at random intervals).

4.6.2: Gambling

Suppose we wish to simulate rolling a pair of dice and we have random number generator producing numbers x between 0 and 1 (Not including 1). We simulate the roll of one die with INT (6*X) + 1. We repeat the process for the second die and add the numbers for both "rolls" to get our result.

4.6.3: Approximation

As another application of random numbers, let's approximate the area of the unit circle, $x^2 + y^2 \leq 1$. one fourth of the circle lies in the unit area square, $0 \leq x \leq 1, (0 \leq y \leq 1)$.So if we generate the points in the square at random by generating x and y at random, (the number of points in the circle) / (the number of point generated) should approximate (one-fourth the area of the circle)/ (the area of the square). Let A be the area of the circle. Then using the linear congruential generator we can get A=3.1648. Note that we have obtained a pair approximation of π.

4.6.4: Sampling

Sampling refers to method of deducing properties of a large set of elements by studying only a small, random subset, thus average value of $f(x)$ over an interval may be estimated from its average over a finite, random subset of points in the interval. Since the average of $f(x)$ is actually an integral, this amount of a Monte Carlo method for approximation integration.

4.6.5: Computer Programming

Random values make a good source of data for testing the effectiveness of computer algorithm. More importantly, they are crucial to the operation of randomized algorithms, which are often far superior to their deterministic counter-parts.

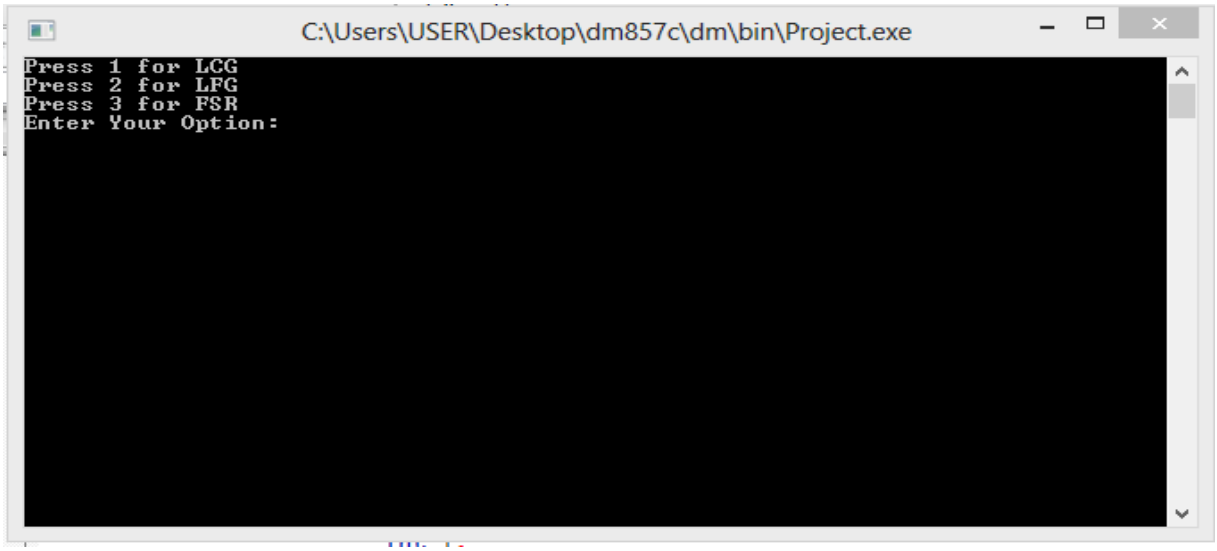
4.7: Categorization of Random Number Generators

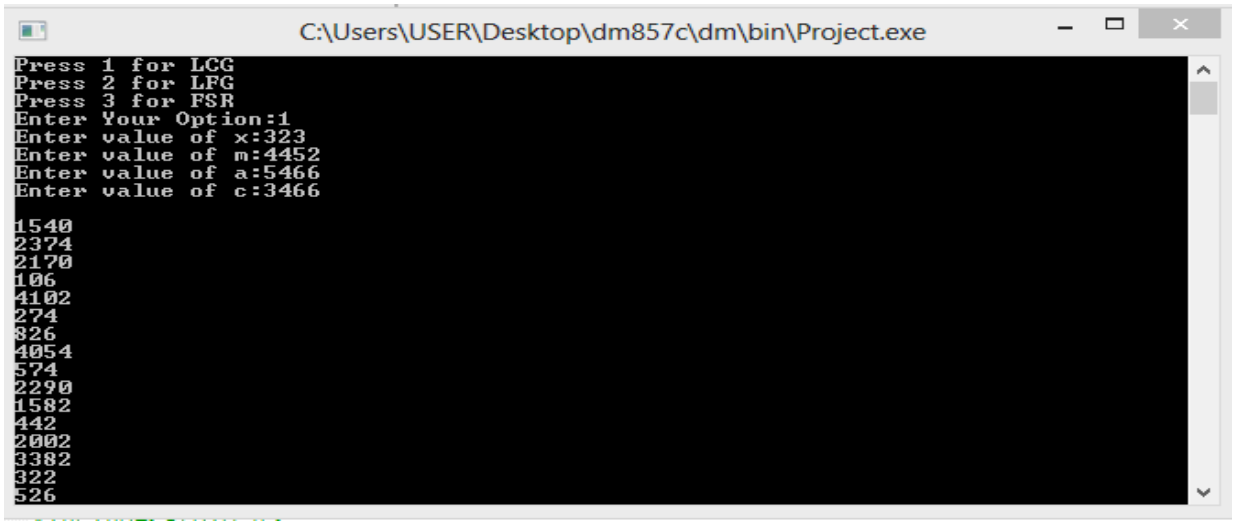
Based on the result analysis obtained from random number generators under student, it has come to our conclusion that each of them is worthy to accepted and placed in an area that based suit users need.

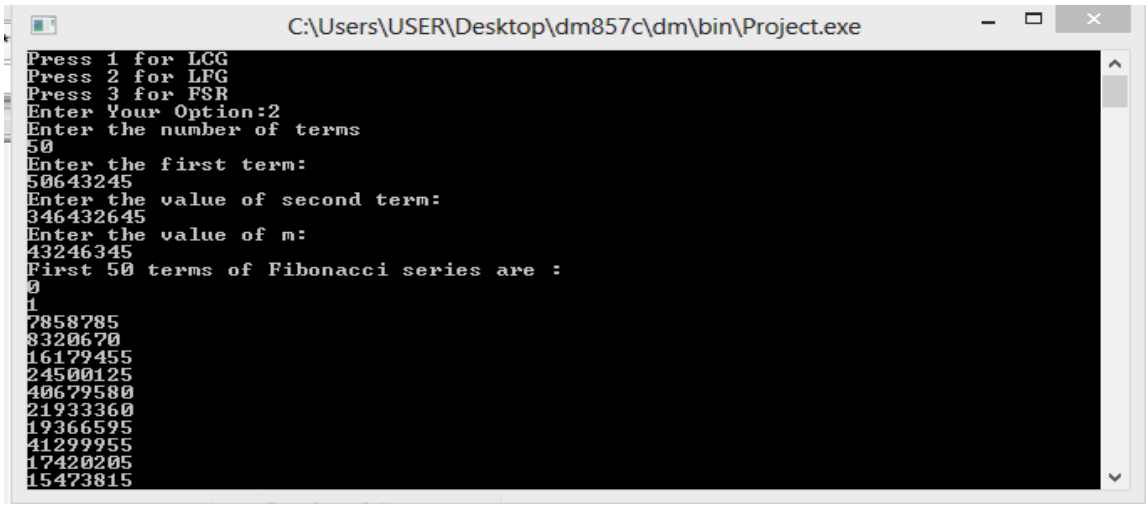
Table 4.8: Random Number Generators with their Suitable Application Domain

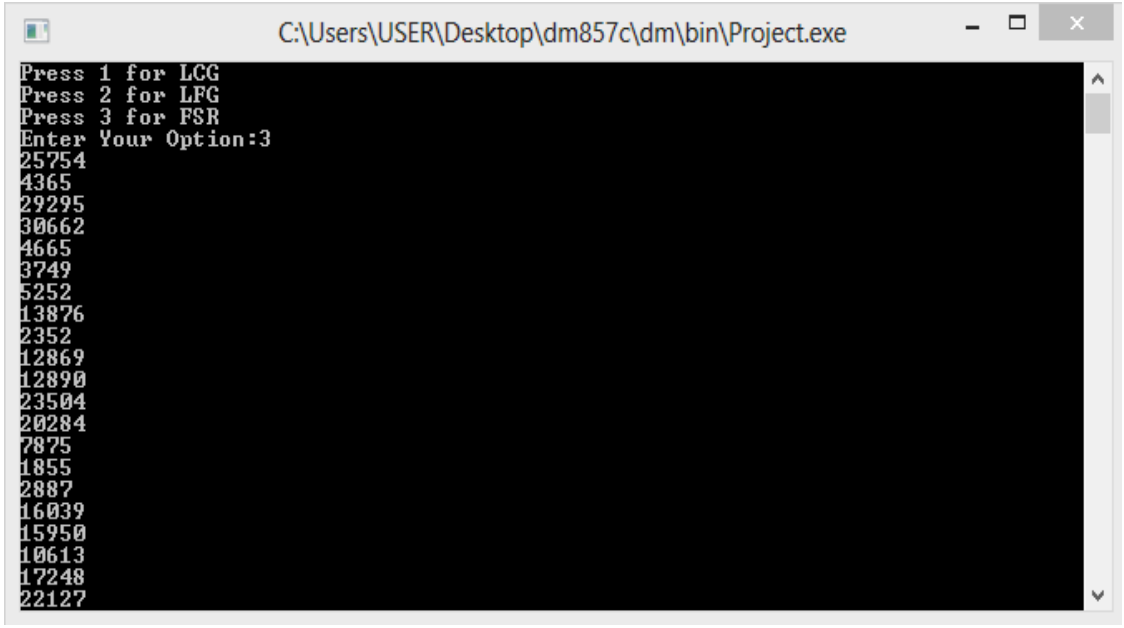
Application Area	Random Number Generator
Simulation	Linear Congruential Generator
Computer Programming	Feedback Shift Register
Approximation (Numerical Analysis)	Lagged Fibonacci Generator
Sampling	Lagged Fibonacci Generator

4.8: Screen Shots






C:\Users\USER\Desktop\dm857c\dm\bin\Project.exe
Press 1 for LCG
Press 2 for LFG
Press 3 for FSR
Enter Your Option:2
Enter the number of terms
50
Enter the first term:
50643245
Enter the value of second term:
346432645
Enter the value of m:
43246345
First 50 terms of Fibonacci series are :
0
1
7858785
8320670
16179455
24500125
40679580
21933360
19366595
41299955
17420205
15473815


C:\Users\USER\Desktop\dm857c\dm\bin\Project.exe
Press 1 for LCG
Press 2 for LFG
Press 3 for FSR
Enter Your Option:3
25754
4365
29295
30662
4665
3749
5252
13876
2352
12869
12890
23504
20284
7875
1855
2887
16039
15950
10613
17248
22127

CHAPTER FIVE
SUMMARY, CONCLUSION AND FUTURE WORK

5.1: SUMMARY

We have presented the history of random number generation and Monte Carlo method in chapter one, while an in depth explanations of mathematical techniques and methods for generating random number drawn from related literatures was presented in chapter two. Three pseudo random number generators implemented in c language are compared and each of them is tested using empirical testing technique to rate their degree of randomness. The chi-square test demonstration in chapter three was used for uniform distribution test and a special case of chi-square that uses a contingency table was used for independence test.

Finally, these test results for uniform distribution, independence and speed obtained in chapter four was used to place each random number generator into category that might suit user application.

5.2: CONCLUSION

In this thesis, we analyzed random number generators statistically and dynamically in other to diagnose its weaknesses. Initially, historical examples of random number generators were given in order to emphasize the importance of the topic. Then the works on some commonly generator were examined generally. After the preparation phase, empirical tests of these random numbers were done. Sources codes in c language for these algorithms were also analysis for performance using asymptotic notations. While investigating the chi-square result for uniformity and independence, it was found that all the generators produced accepted result for uniformity and rejected results for independence. After this discovery, each generation was placed into a category that might suit user's application.

5.3: FUTURE WORK

In the future, to search for good generators of this form, one should employ more sophisticated testing algorithm. Using the metropolis, swedsen- wang, and Wolff Monte Carlo algorithms is our recommendation. Many generators that perform well in standard statistical tests are shown to fail these Monte Carlo tests.

REFERENCES

Adefila, J.J. (2008) *Research Methodology in Behavioral Science*; Kaduna: Apani Publication.

Avantika Y. (2013*) Design and Analysis of True Random Number Generator,* M.Sc. Thesis, Allahabad, Motilal Nehru National Institute of Technology, India.

Benneth P. and Deborah J. (1998).*Randomness,* Cambridge: Harvard University Press.

Christophe D. & Diethelm W. (2009) *A note on Random Number Generation.* Retrieve September,04,2017from http://cr.yp.to/streamciphers.com

Coddington, Paul D. (1993) *Analysis of random number generators using Monte Carlo simulation,* M.Sc. Thesis, Syracuse, Syracuse University, U.S.A.

David D. (2012). *Random Number Generation: Types and Techniques,* M.Sc. Thesis, New Jersey, Liberty University, U.S.A.

David Z. (2001) *Empirical Spectral Analysis of Random Number Generators,* Ph.D Thesis, Michigan, Western University of Michigan, U.S.A.

Francis S. (2001). *Numerical Analysis*: (2nd Edition); Schaum's outline.

Haahr M. (2017).*Random.org*: Introduction to Randomness and Random Numbers, Retrieved May, 05, 2017 from: https://www.random.org

Knuth, Donald E. (1997) *The Art of Computer Programming*, volume 2; Semi Numerical Algorithm (Second Edition): Addison Wesley.

Mark D.M. (1984). *Random Numbers*: Module 590; Department of Mathematics, U.S. Naval Academy, Annapolis, U.S.A.

Nicholas P.D. (2006). *Elementary Statistics.* (Second Edition).Kaduna: Apani Publications.

Prasada Rao G. (2010) *Random number generation and its better technique.* M.Sc Thesis, Partiala, Thapar University, India.

Raj J. (2008) *Testing Random Number Generators.* Retrieved September, 04, 2017 From http://www.cse.wustl.edu./jain/cse574-08

Rustagi J.S. (1981) *Some Test of Random Numbers with Application*: Technical-Report No. 222, Ohio, Ohio State University, USA.

Sadus R.J. (1999) Molecular Simulation of Fluid: Theory, Algorithms and Object Orientation: Retrieved May, 05, 2017 from http:// www.fourmilab.ch/hotbits

Serkan S. (2013) *Analysis of Android Random Number Generator,* M.Sc. Thesis, Istanbul,

Bilkent University, Turkey.

Thomas, H. C. *et al.*,,,. (2009) Introduction to Algorithms; (3rd Edition) Massachusetts: The MIT Press.

Wikipedia (2017) *Monte Carlo Method.* Retrieved September, 04, 2017 from the free encyclopedia: https://en.wikipedia.org/wiki/Web_development.

Appendix I: Source Code

```
/*=========================================================*/
/* Test of the random number generators for uniformly distribution */
/* Three types of random generators can be simulated:
/*     - modulo generator
/*     - lagged Fibonacci
/*     - shift register generator
/* PROGRAM NAME: Random Number Generators
/* (c) Yusuf Auwal Sani, Department of Computer Science Yobe State University.
/*=========================================================*/
#include <stdio.h>
#include <conio.h>
#include <time.h>
#include <stdlib.h>
    void main(){
    int a,c,m,x,i;// time O(0)
    int option;// time O (0)
    printf("Comparative Analysis of Random Number Generators");
    printf("\n");
    printf("Using Monte Carlo Algorithms");
    printf("\n");
    printf("By","");
    printf("Yusuf Auwal Sani");
    printf("\n");
    printf("Press 1 for LCG");// prompt user to enter a choose for LCG
    printf("\n");
    printf("Press 2 for LFG");// prompt user to enter a choose for LFG
    printf("\n");
    printf ("Press 3 for FSR");
    printf("\n");
    printf("Enter Your Option:");// prompt user to enter an option
    scanf("%d",&option);// read a console input for option
    switch (option)// initialize switch
    {
       case 1:
            printf("Enter value of x:");
            scanf("%d",&x);
            printf("Enter value of m:");
            scanf("%d",&m);
```

```
        printf("Enter value of a:");
        scanf("%d",&a);
        printf("Enter value of c:");
        scanf("%d",&c);
        for (i=0; i<50; i++)//time (n +1)
        {
        x=(a*x+c)%m;// time O(1)
        printf("\n");
        printf("%d",x); //time O(1xn)
        }
        break;
 case 2:
          {
             int n, first, second, next, c,m;
             printf("Enter the number of terms\n");
             scanf("%d",&n);
             printf("Enter the first term:\n");
             scanf("%d",&first);
             printf("Enter the value of second term:\n");
             scanf("%d",&second);
             printf("Enter the value of m:\n");
             scanf("%d",&m);
             printf("First %d terms of Fibonacci series are :\n",n);
             for ( c = 0 ; c < n ; c++ )// time O(n+1)
          {
                if ( c <= 1 ) // time O (1)
                next = c;// time O(1)
                else {
                next = (first + second)%m;// time O(1)
                first = second;// time O(1)
                second = next; // time O(1)
             }
                printf("%d\n",next);//time O( 1xn)
             }
                return 0;
           }
 case 3:
          {
             time_t t;
             srand((unsigned)time(&t));
```

```
                int i; // time O(0)
                int array[50]; // time O(0)
                for(i=0; i<50; i++) // time O(n+1)
                printf("%d\n",rand()%30000); // time O(n x 1)
               }
    default: printf("You have enter a wrong choice!");
  }
}
```

Note:

Time Complexity of Linear Congruential Generator can be expressed as:

$$\sum_{n=1}^{\infty} = n + 1 + 1 + n \times 1$$

$$= 2n + 2$$

By ignoring the low order terms and removing the leading constant, we have:

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

Time Complexity of Lagged Fibonacci Generator can be expressed as:

$$2n+5= O(n)$$

Time Complexity of Feedback Shift Register Generator can be expressed as:

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