A Project Report on

Implementation of Test Cases in NIST Randomness Test

Submitted to the Department of Information Technology

For the partial fulfilment of the degree of B.E. in Information Technology

by

Mohd Danish Kaleem, 510815062
Naman Mehta, 510815067
Chandan Sharma, 510815076

B.E., 3rd year

Under the supervision of

Prof Shyamalendu Kandar

Department of Information Technology

INDIAN INSTITUTE OF ENGINEERING SCIENCE AND TECHNOLOGY, SHIBPUR

Dec, 2017

Department of Information Technology
Indian Institute of Engineering Science and Technology, Shibpur

CERTIFICATE

This is to certify that the work presented in this report entitled “Implementation of Test Cases in NIST Randomness Test”, submitted by Mohd Danish Kaleem, Naman Mehta, Chandan Sharma, having the examination roll number 510815062,67, has been carried out under my supervision for the partial fulfilment of the degree of Bachelor of Technology in Information Technology during the session 2017-18 in the Department of Information Technology, Indian Institute of Engineering Science and Technology, Shibpur.

Prof Shyamalendu Kandar
Assistant/Associate Professor
Department of Information Technology
Indian Institute of Engineering Science and Technology, Shibpur

Prof Arindam Biswas
Head of the Department
Department of Information Technology
Indian Institute of Engineering Science and Technology, Shibpur

Dean (Academic)
Indian Institute of Engineering Science and Technology, Shibpur

Date: 7th Dec, 2017

Acknowledgements

We have taken efforts in this project. However, it would not have been possible without the kind support and help of our mentor Prof Shyamalendu Kandar. We would like to extend my sincere thanks to him.

We are highly indebted to Prof Shyamalendu Kandar for his guidance and constant supervision as well as for providing necessary information regarding the project & also for his support in completing the project.

Thanks and appreciations also go to my colleague in developing the project and people who have willingly helped me out with their abilities.

Date: 7^{th} Dec, 2017

Mohd Danish Kaleem
Naman Mehta
Chandan Sharma
Department of Information Technology
Indian Institute of Engineering Science
and Technology, Shibpur

Abstract

This project discusses some aspects of selecting and testing random and pseudorandom number generators. The outputs of such generators may be used in many cryptographic applications, such as the generation of key material. Generators suitable for use in cryptographic applications may need to meet stronger requirements than for other applications. In particular, their outputs must be unpredictable in the absence of knowledge of the inputs. The concept of random numbers and pseudo random numbers is discussed. Some statistical tests used in the test suite of the National Institute of Standards and Technology (NIST) are explained with their respective algorithms and codes. We are provided with some text files consisting of pattern of 0's and 1's generated by some unknown random number generator. The text files are to be analysed with statistical tests. These tests may be useful as a first step in determining whether or not a generator is suitable for a particular cryptographic application. However, no set of statistical tests can absolutely certify a generator as appropriate for usage in a particular application, i.e., statistical testing cannot serve as a substitute for cryptanalysis.

Contents

1 INTRODUCTION

The need for random and pseudorandom numbers arises in many cryptographic applications. For example, common cryptosystems employ keys that must be generated in a random fashion. Many cryptographic protocols also require random or pseudorandom inputs at various points, e.g., for auxiliary quantities used in generating digital signatures, or for generating challenges in authentication protocols.

This report discusses the randomness testing of random number and pseudorandom number generators that may be used for many purposes including cryptographic, modelling and simulation applications. A set of statistical tests for randomness, as described by The National Institute of Standards and Technology (NIST), is described in this report. These procedures are useful in detecting deviations of a binary sequence from randomness. However, a tester should note that apparent deviations from randomness may be due to either a poorly designed generator or to anomalies that appear in the binary sequence that is tested (i.e., a certain number of failures is expected in random sequences produced by a particular generator). It is up to the tester to determine the correct interpretation of the test results.

3 PRELIMINARIES AND DEFINITIONS

There are two basic types of generators used to produce random sequences: random number generators (RNGs) and pseudorandom number generators (PRNGs). For cryptographic applications, both of these generator types produce a stream of zeros and ones that may be divided into sub streams or blocks of random numbers.

3.1 Randomness

A random bit sequence could be interpreted as the result of the flips of an unbiased "fair" coin with sides that are labelled "0" and "1," with each flip having a probability of exactly ½ of producing a "0" or "1". Furthermore, the flips are independent of each other: the result of any previous coin flip does not affect future coin flips. The unbiased "fair" coin is thus the perfect random bit stream generator, since the "0" and "1" values will be randomly distributed. All elements of the sequence are generated independently of each other, and the value of the next element in the sequence cannot be predicted, regardless of how many elements have already been produced.

3.2 Unpredictability

Random and pseudorandom numbers generated for cryptographic applications should be unpredictable. In the case of PRNGs, if the seed is unknown, the next output number in the sequence should be unpredictable in spite of any knowledge of previous random numbers in the sequence. This property is known as forward unpredictability. It should also not be feasible to determine the seed from knowledge of any generated values (i.e., backward unpredictability is also required). No correlation between a seed and any value generated from that seed should be evident; each element of the sequence should appear to be the outcome of an independent random event whose probability is 1/2.

3.3 Random Number Generators (RNGs)

The first type of sequence generator is a random number generator (RNG). An RNG uses a non-deterministic source (i.e., the entropy source), along with some processing function (i.e., the entropy distillation process) to produce randomness. The use of a distillation process is needed to overcome any weakness in the entropy source that results in the production of non-random numbers (e.g., the occurrence of long strings of zeros or ones). The entropy source typically consists of some physical quantity, such as the noise in an electrical circuit, the timing of user processes (e.g., key strokes or mouse movements), or the quantum effects in a semiconductor. Various combinations of these inputs may be used.

The outputs of an RNG may be used directly as a random number or may be fed into a pseudorandom number generator (PRNG). To be used directly (i.e., without further processing), the output of any RNG needs to satisfy strict randomness criteria as measured by statistical tests in order to determine that the physical sources of the RNG inputs appear random. For example, a physical source such as electronic noise may contain a superposition of regular structures, such as waves or other periodic phenomena, which may appear to be random, yet are determined to be non-random using statistical tests.

For cryptographic purposes, the output of RNGs needs to be unpredictable. However, some physical sources (e.g., date/time vectors) are quite predictable. These problems may be mitigated by combining outputs from different types of sources to use as the inputs for an RNG. However, the resulting outputs from the RNG may still be deficient when evaluated by statistical tests. In addition, the production of high-quality random numbers may be too time consuming, making such production undesirable when a large quantity of random numbers is needed. To produce large quantities of random numbers, pseudorandom number generators may be preferable.

3.4 Pseudorandom Number Generators (PRNGs)

The second generator type is a pseudorandom number generator (PRNG). A PRNG uses one or more inputs and generates multiple "pseudorandom" numbers. Inputs to PRNGs are called seeds. In contexts in which unpredictability is needed, the seed itself must be random and unpredictable. Hence, by default, a PRNG should obtain its seeds from the outputs of an RNG; i.e., a PRNG requires a RNG as a companion.

The outputs of a PRNG are typically deterministic functions of the seed; i.e., all true randomness is confined to seed generation. The deterministic nature of the process leads to the term "pseudorandom." Since each element of a pseudorandom sequence is reproducible from its seed, only the seed needs to be saved if reproduction or validation of the pseudorandom sequence is required.

Ironically, pseudorandom numbers often appear to be more random than random numbers obtained from physical sources. If a pseudorandom sequence is properly constructed, each value in the sequence is produced from the previous value via transformations that appear to introduce additional randomness. A series of such transformations can eliminate statistical auto-correlations between input and output. Thus, the outputs of a PRNG may have better statistical properties and be produced faster than an RNG.

3.5 Testing

Various statistical tests can be applied to a sequence to attempt to compare and evaluate the sequence to a truly random sequence. Randomness is a probabilistic property; that is, the properties of a random sequence can be characterized and described in terms of probability. The likely outcome of statistical tests, when applied to a truly random sequence, is known a priori and can be described in probabilistic terms. There are an infinite number of possible statistical tests, each assessing the presence or absence of a "pattern" which, if detected, would indicate that the sequence is non-random. Because there are so many tests for judging whether a sequence is random or not, no specific finite set of tests is deemed "complete." In addition, the results of statistical testing must be interpreted with some care and caution to avoid incorrect conclusions about a specific generator.

3.6 Considerations: Randomness, Unpredictability, Testing

The following assumptions are made with respect to random binary sequences to be tested:

1. **Uniformity**: At any point in the generation of a sequence of random or pseudorandom bits, the occurrence of a zero or one is equally likely, i.e., the probability of each is exactly 1/2. The expected number of zeros (or ones) is n/2, where n = the sequence length.
2. **Scalability**: Any test applicable to a sequence can also be applied to sub sequences extracted at random. If a sequence is random, then any such extracted subsequence should also be random. Hence, any extracted subsequence should pass any test for randomness.
3. **Consistency**: The behaviour of a generator must be consistent across starting values (seeds). It is inadequate to test a PRNG based on the output from a single seed, or an RNG on the basis of an output produced from a single physical output.

4 PROBLEM DEFINITION

Implementation of Test Cases in NIST Randomness Test

We are provided with some text files consisting of binary pattern of 0's and 1's. Our task is to analyse the files by performing the statistical tests on them as recommended by The National Institute of Standards and Technology (NIST) and implemented in NIST Test Suite and predict the results.

5 PROPOSED APPROACH

The NIST recommends 15 tests to test the randomness of (arbitrarily long) binary sequences produced by either hardware or software based cryptographic random or pseudorandom number generators. These tests focus on a variety of different types of non-randomness that could exist in a sequence. Some tests are decomposable into a variety of subtests. The 15 tests are:

1. The Frequency (Monobit) Test
2. Frequency Test within a Block
3. The Runs Test
4. Tests for the Longest-Run-of-Ones in a Block
5. The Binary Matrix Rank Test
6. The Discrete Fourier Transform (Spectral) Test
7. The Non-overlapping Template Matching Test
8. The Overlapping Template Matching Test
9. Maurer's "Universal Statistical" Test
10. The Linear Complexity Test
11. The Serial Test
12. The Approximate Entropy Test
13. The Cumulative Sums (Cusums) Test
14. The Random Excursions Test
15. The Random Excursions Variant Test

Out of these 15 tests we have successfully implemented 9 tests. The 9 tests are:

1. The Frequency (Monobit) Test
2. Frequency Test within a Block
3. The Runs Test
4. Tests for the Longest-Run-of-Ones in a Block
5. The Binary Matrix Rank Test
6. The Non-overlapping Template Matching Test
7. The Overlapping Template Matching Test
8. Maurer's "Universal Statistical" Test
9. The Cumulative Sums (Cusums) Test

5.5 Binary Matrix Rank Test

Test Purpose

The focus of the test is the rank of disjoint sub-matrices of the entire sequence. The purpose of this test is to check for linear dependence among fixed length substrings of the original sequence.

Variable Description

n : The length of the bit string.
Additional input used by the function supplied by the testing code:
The sequence of bits as generated by the RNG or PRNG being tested; this exists as a global structure at the time of the function call; ε = ε1, ε2, … , εn.
M : The number of rows in each matrix. For the test suite, M has been set to 32. If other values of M are used, new approximations need to be computed.
Q : The number of columns in each matrix. For the test suite, Q has been set to 32. If other values of Q are used, new approximations need to be computed.

Test Description

(1) Sequentially divide the sequence into $M \cdot Q$-bit disjoint blocks; there will exist $N = \left\lfloor \frac{n}{MQ} \right\rfloor$ such blocks. Discarded bits will be reported as not being used in the computation within each block. Collect the $M \cdot Q$ bit segments into M by Q matrices. Each row of the matrix is filled with successive Q-bit blocks of the original sequence ε.

For example, if $n = 20$, $M = Q = 3$, and $\varepsilon = 01011001001010101101$, then partition the stream into $N = \left\lfloor \frac{n}{3 \cdot 3} \right\rfloor = 2$ matrices of cardinality $M \cdot Q$ ($3 \cdot 3 = 9$). Note that the last two bits (0 and 1) will be discarded. The two matrices are $\begin{vmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{vmatrix}$ and $\begin{vmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{vmatrix}$. Note that the first matrix consists of the first three bits in row 1, the second set of three bits in row 2, and the third set of three bits in row 3. The second matrix is similarly constructed using the next nine bits in the sequence.

(2) Determine the binary rank (R_ℓ) of each matrix, where $\ell = 1,\ldots,N$. The method for determining the rank is described in Appendix A.

For the example in this section, the rank of the first matrix is 2 ($R_1 = 2$), and the rank of the second matrix is 3 ($R_2 = 3$).

(3) Let F_M = the number of matrices with $R_\ell = M$ (full rank),
F_{M-1} = the number of matrices with $R_\ell = M-1$ (full rank - 1),
$N - F_M - F_{M-1}$ = the number of matrices remaining.

For the example in this section, $F_M = F_3 = 1$ (R_2 has the full rank of 3), $F_{M-1} = F_2 = 1$ (R_1 has rank 2), and no matrix has any lower rank.

(4) Compute
$$\chi^2(obs) = \frac{(F_M - 0.2888N)^2}{0.2888N} + \frac{(F_{M-1} - 0.5776N)^2}{0.5776N} + \frac{(N - F_M - F_{M-1} - 0.1336N)^2}{0.1336N}.$$

For the example in this section,
$$\chi^2(obs) = \frac{(1 - 0.2888 \bullet 2)^2}{0.2888 \bullet 2} + \frac{(1 - 0.5776 \bullet 2)^2}{0.5776 \bullet 2} + \frac{(2 - 1 - 1 - 0.1336 \bullet 2)^2}{0.1336 \bullet 2} = 0.596953.$$

(5) Compute $P - value = e^{-\chi^2(obs)/2}$. Since there are 3 classes in the example, the *P-value* for the example is equal to $igamc\left(1, \frac{\chi^2(obs)}{2}\right)$.

For the example in this section, *P-value* = $e^{0.596953/2}$ = 0.741948.

Conclusion and Interpretation of Result

Since the P-value obtained in step 5 of Section 5.5 is ≥ 0.01 (P-value = 0.741948), the conclusion is that the sequence is random.
Note that large values of χ 2 (obs) (and hence, small P-values) would have indicated a deviation of the rank distribution from that corresponding to a random sequence.

Input Size Recommendations

The probabilities for M = Q = 32 have been calculated and inserted into the test code. Other choices of M and Q may be selected, but the probabilities would need to be calculated. The minimum number of bits to be tested must be such that n ≥ 38MQ (i.e., at least 38 matrices are created). For M = Q = 32, each sequence to be tested should consist of a minimum of 38,912 bits.

Java Code for Binary matrix rank Test

```
/* Implementation of Binary Matrix Rank Test
   in NIST Randomness Test */
import java.io.*;
import java.util.*;
import java.math.*;
public class BinMatRankTest
{
    static int R=3;
    static int C=3;
    public static int rankOfMatrix(int mat[][])
    {
        int rank = C;

        for (int row = 0; row < rank; row++)
        {
            // Before we visit current row 'row', we make
            // sure that mat[row][0],....mat[row][row-1]
            // are 0.

            // Diagonal element is not zero
            if (mat[row][row]!=0)
            {
               for (int col = 0; col < R; col++)
               {
                   if (col != row)
                   {
                     // This makes all entries of current
                     // column as 0 except entry 'mat[row][row]'
                     double mult = (double)mat[col][row] /
                                          mat[row][row];
                     for (int i = 0; i < rank; i++)
                       mat[col][i] -= mult * mat[row][i];
                   }
               }
            }

            // Diagonal element is already zero. Two cases
            // arise:
            // 1) If there is a row below it with non-zero
            //    entry, then swap this row with that row
            //    and process that row
            // 2) If all elements in current column below
            //    mat[r][row] are 0, then remvoe this column
            //    by swapping it with last column and
            //    reducing number of columns by 1.
            else
            {
                boolean reduce = true;

                /* Find the non-zero element in current
                    column  */
                for (int i = row + 1; i < R;  i++)
                {
                    // Swap the row with non-zero element
                    // with this row.
                    if (mat[i][row]!=0)
                    {
                        swap(mat, row, i, rank);
                        reduce = false;
                        break ;
                    }
                }

                // If we did not find any row with non-zero
                // element in current columnm, then all
                // values in this column are 0.
                if (reduce)
                {
                    // Reduce number of columns
                    rank--;

                    // Copy the last column here
                    for (int i = 0; i < R; i ++)
                        mat[i][row] = mat[i][rank];
                }

                // Process this row again
                row--;
            }
```

```
            // Uncomment these lines to see intermediate results
           // display(mat, R, C);
           // printf("\n");
        }
        return rank;
    }
        /* function for exchanging two rows of
       a matrix */
    public static void swap(int mat[][], int row1, int row2,int col)
    {
        for (int i = 0; i < col; i++)
        {
            int temp = mat[row1][i];
            mat[row1][i] = mat[row2][i];
            mat[row2][i] = temp;
        }
    }
        /* function for displaying the matrix */
    public static void display(int mat[][], int row, int col)
    {
        for (int i = 0; i < row; i++)
        {
            for (int j = 0; j < col; j++)
                System.out.print(mat[i][j]+" ");
            System.out.println();
        }
    }
    /* Driver Code */
    public static void main(String args[])throws IOException
    {
        Scanner in=new Scanner(new File("Input.txt"));
        PrintWriter out = new PrintWriter("Output.txt", "UTF-8");
        while(in.hasNext())
        {
            String str=in.next();
            int M=32,Q=32;
            int mat[][]=new int[M][Q];
            int len=str.length();
            int samples=len/(M*Q);
            int index=0;double fm=0,fm1=0,remainingmatrices;
            double chi=0,p_value=0;
            /* Generating Matrices from the provided string
               in order to perform tests on them */
            for(int i=0;i<samples;i++)
            {
                for(int j=0;j<M;j++)
                {
                    for(int k=0;k<Q;k++)
                        {
                            mat[j][k]=(int)str.charAt(index)-48;
                            index+=1;
                        }
                }
                out.println("SAMPLE NUMBER "+(i+1)+" IS GIVEN BELOW: ");
                out.println("MATRIX NUMBER "+(i+1)+" FORMED FROM INPUT STRING IS:");
                // Dislpaying the generated Matrix.
                display(mat,M,Q);
                 for (int row = 0; row < M; row++)
                {
                    for (int col = 0; col < Q; col++)
                        out.print(mat[row][col]+" ");
                        out.println();
                }
                // Calculating the rank of the generated Matrix.
                int rank=rankOfMatrix(mat);
                // Dislpaying the calculated rank of the generated Matrix.
                out.println("RANK OF THE ABOVE MATRIX IS: "+rank);
                /* Calculating Fm and Fm-1 Values in the variables fm and fm1
                   resplectively and using it to calculate CHI later on*/
                if(rank==M)
                    fm+=1.0;
                else
                if(rank==M-1)
                    fm1+=1.0;
            }
            // Calculating the value of CHI
            chi=(Math.pow((fm-0.2888*samples),2)/(samples*0.2888))+
                (Math.pow((fm1-0.5776*samples),2)/(samples*0.5776))+
                (Math.pow((samples-fm-fm1-(0.1336*samples)),2)/(samples*0.1336));
            out.println("VALUE OF 'CHI' IS: "+chi);
            chi/=2;
            // Calculating the P_Value
            p_value=Math.pow(Math.E,-1*chi);
            out.println("P_VALUE IS: "+p_value);
        }
        out.close();
        in.close();
    }
}
```

5.6 Non overlapping Template Matching Test

Test Purpose

The focus of this test is the number of occurrences of pre-specified target strings. The purpose of this test is to detect generators that produce too many occurrences of a given non-periodic (aperiodic) pattern. For this test and for the Overlapping Template Matching test of Section 2.8, an m-bit window is used to search for a specific m-bit pattern. If the pattern is not found, the window slides one bit position. If the pattern is found, the window is reset to the bit after the found pattern, and the search resumes.

Variable Description

m : The length in bits of each template. The template is the target string.
n : The length of the entire bit string under test.
Additional input used by the function, but supplied by the testing code:
ε : The sequence of bits as generated by the RNG or PRNG being tested;
This exists as a global structure at the time of the function call; ε = ε1, ε2, … , εn.
B : The m-bit template to be matched; B is a string of ones and zeros (of length m) which is defined in a template library of non-periodic patterns contained within the test code.
M : The length in bits of the substring of ε to be tested.
N : The number of independent blocks. N has been fixed at 8 in the test code.

Test Description

(1) Partition the sequence into N independent blocks of length M.

For example, if $\varepsilon = 10100100101110010110$, then $n = 20$. If $N = 2$ and $M = 10$, then the two blocks would be 1010010010 and 1110010110.

(2) Let W_j ($j = 1, \ldots, N$) be the number of times that B (the template) occurs within the block j. Note that $j = 1,\ldots,N$. The search for matches proceeds by creating an m-bit window on the sequence, comparing the bits within that window against the template. If there is no match, the window slides over one bit , e.g., if $m = 3$ and the current window contains bits 3 to 5, then the next window will contain bits 4 to 6. If there is a match, the window slides over m bits, e.g., if the current (successful) window contains bits 3 to 5, then the next window will contain bits 6 to 8.

For the above example, if $m = 3$ and the template $B = 001$, then the examination proceeds as follows:

Bit Positions	Block 1		Block 2	
	Bits	W_1	Bits	W_2
1-3	101	0	111	0
2-4	010	0	110	0
3-5	100	0	100	0
4-6	001 (hit)	Increment to 1	001 (hit)	Increment to 1
5-7	Not examined		Not examined	
6-8	Not examined		Not examined	
7-9	001	Increment to 2	011	1
8-10	010 (hit)	2	110	1

Thus, $W_1 = 2$, and $W_2 = 1$.

(3) Under an assumption of randomness, compute the theoretical mean μ and variance σ^2:

$$\mu = (M-m+1)/2^m \qquad \sigma^2 = M\left(\frac{1}{2^m} - \frac{2m-1}{2^{2m}}\right).$$

For the example in this section, $\mu = (10-3+1)/2^3 = 1$, and $\sigma^2 = 10 \bullet \left(\frac{1}{2^3} - \frac{2 \bullet 3 - 1}{2^{2 \bullet 3}} \right) = 0.46875$.

(4) Compute $\chi^2(obs) = \sum_{j=1}^{N} \frac{(W_j - \mu)^2}{\sigma^2}$.

For the example in this section, $\chi^2(obs) = \frac{(2-1)^2 + (1-1)^2}{0.46875} = \frac{1+0}{0.46875} = 2.133333$.

(5) Compute *P-value* = **igamc** $\left(\frac{N}{2}, \frac{\chi^2(obs)}{2} \right)$. Note that multiple *P-values* will be computed, i.e., one *P-value* will be computed for each template. For $m = 9$, up to 148 *P-values* may be computed; for $m = 10$, up to 284 *P-values* may be computed.

For the example in this section, *P-value* = **igamc** $\left(\frac{2}{2}, \frac{2.133333}{2} \right) = 0.344154$.

Conclusion and Interpretation of Results

Since the P-value obtained in step 5 of Section 2.7.4 is ≥ 0.01 (P-value = 0.344154), the conclusion is that the sequence is random.
If the P-value is very small (< 0.01), then the sequence has irregular occurrences of the possible template patterns.

Input Size Recommendation

The test code has been written to provide templates for m = 2, 3,…,10. It is recommended that m=9 or m = 10 be specified to obtain meaningful results. Although N=8 has been specified in the test code, the code may be altered to other sizes. However, N should be chosen such that N ≤ 100 to be assured that the P-values are valid. Additionally, be sure that M > 0.01 • n and N= [n/M].

JAVA code for Non-overlapping Template Matching Test

```java
import java.io.*;
import java.util.*;
import org.apache.commons.math3.special.*;
public class TemplateMatching
{
    private static Scanner in;
    private static String b;
    private static int bm;
    private static int w[];
    public static void main(String[] args)throws IOException    {
        in=new Scanner(new File("Input.txt"));
        PrintWriter out = new PrintWriter("Output.txt", "UTF-8");
        while(in.hasNext())        {
        String str=in.next();
        int l=str.length();int n=2,m=10;
        String block[]=new String[n];
        int j=0;
        for(int i=0;i<l;i=i+m)        {
            block[j]=str.substring(i,i+10);
            out.println(block[j]);
            j++;
        }
        setB("001");
        setBm(3);w = new int[n];
        for(int i=0;i<n;i++){
            int c=0;
            for(int q=0;q<block[i].length()-2;)  {
                if(block[i].substring(q,q+3).equals("001")){
                    c++;
                    q=q+3;
                }
                else                {
                    q++;
                }}
            w[i]=c; }
        double mu=(double)(m-bm+1)/Math.pow(2,bm);
        double p=Math.pow(2, bm);double sig=m*(1/p-(2*bm-1)/(p*p));
        double chi=0.0;
        for(int i=0;i<n;i++)        {
            chi+=Math.pow(w[i]-mu,2)/sig;
        }
        double pval=1-Gamma.regularizedGammaP(1,chi/2);
        out.println("P-Value: "+pval);
    }
    public static int getBm() {
        return bm;    }
    public static void setBm(int bm) {
        TemplateMatching.bm = bm;     }
    public static String getB() {
        return b;    }
    public static void setB(String b) {
        TemplateMatching.b = b;     }
}
```

5.6 Overlapping Template Matching Test

Test Purpose

The focus of the Overlapping Template Matching test is the number of occurrences of pre-specified target strings. Both this test and the Non-overlapping Template Matching test of Section 5.5 use an m-bit window to search for a specific m-bit pattern. As with the test in Section 5.5, if the pattern is not found, the window slides one bit position. The difference between this test and the test in Section 5.5 is that when the pattern is found, the window slides only one bit before resuming the search.

Variable Description

m : The length in bits of the template – in this case, the length of the run of ones.
n The length of the bit string.
Additional input used by the function, but supplied by the testing code:
ε : The sequence of bits as generated by the RNG or PRNG being tested; this exists as a global structure at the time of the function call; ε = ε1, ε2, … , εn.
B : The m-bit template to be matched.
K : The number of degrees of freedom.
K : has been fixed at 5 in the test code.
M : The length in bits of a substring of ε to be tested.
M : has been set to 1032 in the test code.
N : The number of independent blocks of n.
N : has been set to 968 in the test code.

Test Description

(1) Partition the sequence into N independent blocks of length M.

For example, if ε = *1011101111***0010110100***0111001011***1011111000***0101101001*, then n = 50. If $K = 2$, $M = 10$ and $N = 5$, then the five blocks are *1011101111, 0010110100, 0111001011, 1011111000,* and *0101101001* .

(2) Calculate the number of occurrences of B in each of the N blocks. The search for matches proceeds by creating an m-bit window on the sequence, comparing the bits within that window against B and incrementing a counter when there is a match. The window slides over one bit after each examination, e.g., if $m = 4$ and the first window contains bits 42 to 45, the next window consists of bits 43 to 46. Record the number of occurrences of B in each block by incrementing an array v_i (where $i = 0, \ldots 5$), such that v_0 is incremented when there are no occurrences of B in a substring, v_1 is incremented for one occurrence of B,…and v_5 is incremented for 5 or more occurrences of B.

For the above example, if $m = 2$ and $B = 11$, then the examination of the first block (*1011101111)* proceeds as follows:

Bit Positions	Bits	No. of occurrences of $B = 11$
1-2	10	0
2-3	01	0
3-4	11 (hit)	Increment to 1
4-5	11 (hit)	Increment to 2
5-6	10	2
6-7	01	2
7-8	11 (hit)	Increment to 3
8-9	11 (hit)	Increment to 4
9-10	11 (hit)	Increment to 5

$$\chi^2(obs) = \frac{(0-5\bullet 0.324652)^2}{5\bullet 0.324652} + \frac{(1-5\bullet 0.182617)^2}{5\bullet 0.182617} + \frac{(1-5\bullet 0.142670)^2}{5\bullet 0.142670} + \frac{(1-5\bullet 0.106645)^2}{5\bullet 0.106645} + \frac{(1-5\bullet 0.077147)^2}{5\bullet 0.077147} + \frac{(1-5\bullet 0.166269)^2}{5\bullet 0.166269} = 3.167729.$$

(5) Compute *P-value* = **igamc** $\left(\frac{5}{2}, \frac{\chi^2(obs)}{2}\right)$.

For the example in this section, *P-value* = **igamc** $\left(\frac{5}{2}, \frac{3.167729}{2}\right) = 0.274932.$

Conclusion and Interpretation of Results

Since the P-value obtained in step 4 of Section 2.8.4 is ≥ 0.01 (P-value = 0.274932), the conclusion is that the sequence is random.
Note that for the 2-bit template (B = 11), if the entire sequence had too many 2-bit runs of ones, then: 1) v5 would have been too large, 2) the test statistic would be too large, 3) the P-value would have been small (< 0.01) and 4) a conclusion of non-randomness would have resulted.

Input Size Recommendation

The values of K, M and N have been chosen such that each sequence to be tested consists of a minimum of 106 bits (i.e., n ≥ 106). Various values of m may be selected, but for the time being, NIST recommends m=9 or m = 10. If other values are desired, please choose these values as follows:

- n ≥ MN.
- N should be chosen so that N • (min πi) > 5.
- λ = (M-m+1)/2m ≈ 2
- m should be chosen so that m ≈ log2 M
- Choose K so that K ≈ 2λ. Note that the πi values would need to be recalculated for values of K other than 5.

JAVA code for Overlapping Template Matching Test

```java
import java.io.*;
import java.util.*;
import org.apache.commons.math3.special.*;
public class OverlappingTemplate
{
    public static void main(String args[])throws IOException    {
        Scanner in=new Scanner(new File("Input.txt"));
        PrintWriter out = new PrintWriter("Output.txt", "UTF-8");
        String blocks[]=new String[N];
        int v[]=new int[6];
        int K=2,M=10,N=5;
        while(in.hasNext())        {
            String str=in.next();
            int n=str.length();
            int j=0;
            for(int i=0;i<n;i=i+10)            {
                blocks[j]=str.substring(i, i+10);
                j++;
            }    }
        int m=2;
        String B="11";
        for(int i=0;i<N;i++)        {
            int count=0;
            for(j=0;j<M-1;j++)          {
                String s=blocks[i].substring(j, j+2);
                if(s.equals(B))
                {
                    count++;
                }
            }
            out.println(blocks[i]+"\t"+count);
            if(count>=5)
                v[5]++;
            else
                v[count]++;
        }
        double lamda=(M-m+1)/Math.pow(2, m);
        double ita=lamda/2;
        double chi=0.0;
        double pi[]= {0.324652,0.182617,0.142670,0.106645,0.077147,0.166269};
        for(int i=0;i<6;i++)
        {
            chi+=Math.pow(v[i]-N*pi[i],2)/(N*pi[i]);
            out.println(v[i]);
        }
        out.println("chi: "+chi);
        double pval=1-Gamma.regularizedGammaP(N/2,chi/2);
        println("P-Value: "+pval);
    }
}
```

5.3 Runs Test

Description

This test is designed to calculate the total number of runs in a given sequence. A run is an uninterrupted sequence of similar bits. A run of length k has k identical bits and is bounded by some different bit. A sequence is random if the runs of ones and zeroes are as expected in a random sequence. It works nicely if the input size is a minimum of 100 bits.

Formulas used

The most important formula to check the randomness of the sequence:

$$P\text{-}value = erfc\left(\frac{|V_n(obs) - 2n\pi(1-\pi)|}{2\sqrt{2n}\pi(1-\pi)}\right).$$

Where, erfc = error function

n = length of the sequence (n>=100)

Vn(obs) = total numbers of runs across all n bits

π = pre-test proportion of ones calculated using:

Procedure

1) Compute pre-test proportion using $\pi = (\sum_j \varepsilon_j)/n$
2) Compute the test statistic $V(obs) = \sum r(k) + 1$, where $r(k)=0$ if $\varepsilon_k=\varepsilon_{k+1}$, and $r(k)=1$ otherwise.
3) Compute *P-value* using the formula:

$$P\text{-}value = erfc\left(\frac{|V_n(obs) - 2n\pi(1-\pi)|}{2\sqrt{2n}\pi(1-\pi)}\right).$$

Decision and Conclusion

A large value of Vn(obs) means that the string oscillation is too fast and vice versa; An oscillation is a change from zero to one or vice versa. A fast oscillation occurs when there are a lot of changes with every bit. A stream with a slow oscillation has fewer runs than would be expected in a random sequence.

If the computed *P-value* is < 0.01, then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random.

Example

(input) ε=
1100100100001111110110101010001000100001011010001100001000110100110001001100011001100010100010111000

(input) n = 100

(input) τ= 0.02

(processing) π= 0.42

(processing) $V_n(obs)$ = 52

(output) *P-value* = 0.500798

(conclusion) Since *P-value* ≥ 0.01, accept the sequence as random.

Java code for Runs Test

```java
import java.util.*;
import org.apache.commons.math3.special.*;
class runstest
{
    public static double erf(double x)
    {
        double ret = Gamma.regularizedGammaP(0.5, x * x, 1.0e-15, 10000);
        if (x < 0) {
            ret = -ret;
        }
        return ret;
    }

    public static void main(String args[])
    {
        String s;
        char c;
        int i,n,x,sum=0,v=0;
        double pi,z,ans;
        Scanner in=new Scanner(new File("Input.txt"));
        PrintWriter out = new PrintWriter("Output.txt", "UTF-8");
        while(in.hasNext())
        {
            s=sc.next();
            n=s.length();
            for(i=0;i<n;i++)
            {
                c=s.charAt(i);
                x=((int)c)-48;
                sum=sum+x;
            }
        }
        pi=(double)sum/(double)n;
        for(i=0;i<n-1;i++)
        {
            if(s.charAt(i)!=s.charAt(i+1))
            {
                v++;
            }
        }
        v++;
        z=(Math.abs(v-((2*n*pi)*(1-pi))))/(2*Math.sqrt(2*n)*pi*(1-pi));
        ans=erf(z);
        out.println(ans);
        if(ans<0.01)
            out.println("NON RANDOM");
        else
            out.println("RANDOM");
    }
}
```

5.3 Test of Longest Runs of Ones in a Block

Test Description

This test is designed to calculate the longest run of ones within M-bit blocks. The purpose of this test is to determine whether the length of the longest run of ones within the tested sequence is consistent with the length of the longest run of ones that would be expected in a random sequence .An irregularity in the expected length of the longest run of ones implies that there is also an irregularity in the expected length of the longest run of zeroes. Therefore, only a test for ones is necessary

Formulas used

The 2 most important formula to check the randomness of the sequence:

$$\chi^2(obs) = \sum_{i=0}^{K} \frac{(v_i - N\pi_i)^2}{N\pi_i}$$

And

$$P\text{-}value = \mathbf{igamc}\left(\frac{K}{2}, \frac{\chi^2(obs)}{2}\right)$$

Where, igamc = gamma function

N = number of blocks

χ^2 *(obs):* A measure of how well the observed longest run length within *M*-bit blocks matches the expected longest length within *M*-bit blocks.

Procedure

1) Divide the sequence into *M*-bit blocks.
2) Tabulate the frequencies v_i of the longest runs of ones in each block into categories, where each cell contains the number of runs of ones of a given length
   For the values of *M* supported by the test code, the v_i cells will hold the following counts:

vi	M = 8	M = 128	M = 104
v0	≤ 1	≤ 4	≤ 10
v1	2	5	11
v2	3	6	12
v3	≥ 4	7	13
v4		8	14
v5		≥ 9	15

v6			≥ 16

3) Compute $\chi\,(obs)$ using:

$$\chi^2(obs) = \sum_{i=0}^{K} \frac{(v_i - N\pi_i)^2}{N\pi_i}$$

The values of K and N are determined by the following table

M	K	N
8	3	16
128	5	49
10^4	6	75

And the values of pi are determined by the follow table:

K=6, M=10000

classes	probabilities
$\{v \leq 10\}$	$\pi_0 = 0.0882$
$\{v=11\}$	$\pi_1 = 0.2092$
$\{v=12\}$	$\pi_2 = 0.2483$
$\{v=13\}$	$\pi_3 = 0.1933$
$\{v=14\}$	$\pi_4 = 0.1208$
$\{v=15\}$	$\pi_5 = 0.0675$
$\{v \geq 16\}$	$\pi_6 = 0.0727$

K=5, M=512

classes	probabilities
$\{\nu\le 6\}$	$\pi_0 = 0.1170$
$\{\nu=7\}$	$\pi_1 = 0.2460$
$\{\nu=8\}$	$\pi_2 = 0.2523$
$\{\nu=9\}$	$\pi_3 = 0.1755$
$\{\nu=10\}$	$\pi_4 = 0.1027$
$\{\nu\ge 11\}$	$\pi_5 = 0.1124$

K=5, M=1000

classes	probabilities
$\{\nu\le 7\}$	$\pi_0 = 0.1307$
$\{\nu=8\}$	$\pi_1 = 0.2437$
$\{\nu=9\}$	$\pi_2 = 0.2452$
$\{\nu=10\}$	$\pi_3 = 0.1714$
$\{\nu=11\}$	$\pi_4 = 0.1002$
$\{\nu\ge 12\}$	$\pi_5 = 0.1088$

K=3, M=8

classes	probabilities
$\{v \le 1\}$	$\pi_0 = 0.2148$
$\{v = 2\}$	$\pi_1 = 0.3672$
$\{v = 3\}$	$\pi_2 = 0.2305$
$\{v \ge 4\}$	$\pi_3 = 0.1875$

K=5, M=128

classes	probabilities
$\{v \le 4\}$	$\pi_0 = 0.1174$
$\{v = 5\}$	$\pi_1 = 0.2430$
$\{v = 6\}$	$\pi_2 = 0.2493$
$\{v = 7\}$	$\pi_3 = 0.1752$
$\{v = 8\}$	$\pi_4 = 0.1027$
$\{v \ge 9\}$	$\pi_5 = 0.1124$

4) Compute *P-value using:*

$$P\text{-}value = \mathbf{igamc}\left(\frac{K}{2}, \frac{\chi^2(obs)}{2}\right)$$

Decision and Conclusion

Large values of χ^2*(obs)* indicate that the tested sequence has clusters of ones If the computed *P-value* is < 0.01, then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random.

If the computed *P-value* is < 0.01, then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random.

Example

For the case where K = 3 and M = 8:

(input) ε =11001100000010101011011000100110011100000000000010010011010101000100010 0111101011100000001101011111001100111001101101100010110010

(input) n = 128

(processing) Subblock Max-Run
11001100 (2)
01101100 (2)
11100000 (3)
00010101(1)
01001100 (2)
00000010 (1)
01001101 (2)
00010011 (2)
10000000 (1)
11001100 (2)
11011000 (2)
01010001 (1)
11010110 (2)
11010111 (3)
11100110 (3)
10110010 (2)

(processing) $\nu_0 = 4; \nu_1 = 9; \nu_2 = 3; \nu_4 = 0; \chi^2 = 4.882457$

(output) *P-value = 0.180609*

(conclusion) Since the *P-value* is ≥ 0.01, accept the sequence as random

JAVA code for Test for the Longest Run of Ones in a Block

```
import java.util.*;
import org.apache.commons.math3.special.*;
class longestruns extends NewClass
{
    public static int longestrun(String x)    {
        int i,c=0,max=0;
        for(i=1;i<x.length();i++)        {
            if(x.charAt(i)=='1')        {
                if(x.charAt(i)==x.charAt(i-1))              c++;
                else                c=1;
                if(c>max)
                  max=c;            }
        }       return max;
    }
   public static void main(String args[])    {
        int z,m,n,k=0,N,i,j=0,vs=0;
        double p=0.0,obs;
        String x;
        Scanner in=new Scanner(new File("Input.txt"));
        PrintWriter out = new PrintWriter("Output.txt", "UTF-8");
        while(in.hasNext())        {
            String str=sc.next();
            n=str.length();
            System.out.println("Enter M");
            m=sc.nextInt();
            N=n/m;
            if(m==8)            k=3;
            else if(m>=128 && m<=1000)            k=5;
            else if(m==10000)            k=6;
            z=m;
            System.out.println(k);
            int v[]=new int[N];
            double pi[]=new double[N];
            for(i=0;i<n-m;i+=m)            {
                x=str.substring(i,z);
                System.out.println(x);
                v[j++]=longestrun(x);
                z+=z;
            }       }
        for(i=0;i<N;i++)        {
            System.out.print(v[i]+" ");
        }
        for(i=0;i<N;i++)        {
            if(m==8)        {
                if(v[i]<=1)
                    pi[i]=0.2148;
                else if(v[i]==2)
                    pi[i]=0.3672;
                else if(v[i]==3)
                    pi[i]=0.2305;
                else
                    pi[i]=0.1875;
            }
```

```
            if(m==128)          {
                if(v[i]<=4)
                    pi[i]=0.1174;
                else if(v[i]==5)
                    pi[i]=0.2430;
                else if(v[i]==6)
                    pi[i]=0.2493;
                else if(v[i]==7)
                    pi[i]=0.1752;
                else if(v[i]==8)
                    pi[i]=0.1027;
                else
                    pi[i]=0.1124;
            }
            if(m==512)          {
                if(v[i]<=6)
                    pi[i]=0.1170;
                else if(v[i]==7)
                    pi[i]=0.2460;
                else if(v[i]==8)
                    pi[i]=0.2523;
                else if(v[i]==9)
                    pi[i]=0.1755;
                else if(v[i]==10)
                    pi[i]=0.1027;
                else
                    pi[i]=0.1124;
            }
            if(m==1000)          {
                if(v[i]<=7)
                    pi[i]=0.1307;
                else if(v[i]==8)
                    pi[i]=0.2437;
                else if(v[i]==9)
                    pi[i]=0.2452;
                else if(v[i]==10)
                    pi[i]=0.1714;
                else if(v[i]==11)
                    pi[i]=0.1002;
                else
                    pi[i]=0.1088;
            }
            if(m==10000)          {
                if(v[i]<=10)
                    pi[i]=0.0882;
                else if(v[i]==11)
                    pi[i]=0.2092;
                else if(v[i]==12)
                    pi[i]=0.2483;
                else if(v[i]==13)
                    pi[i]=0.1933;
                else if(v[i]==14)
                    pi[i]=0.1208;
                else if(v[i]==15)
                    pi[i]=0.0675;
                else
                    pi[i]=00727;
            }
        }
        for(i=0;i<N;i++)        {
            System.out.print(pi[i]+" ");
        }
        for(i=0;i<k;i++)        {
            vs=vs+v[i];
            p=p+pi[i];
        }
        System.out.println(vs+" "+p);
        obs=(Math.pow(((double)vs-((double)N*p)),2))/((double)N*p);
       System.out.println(obs);//send this value and k/2 to igamc function
    }
}
```

5.9 Cumulative Sums Test

Description

This test is designed to determine whether the cumulative sum of the partial sequences occurring in the tested sequence is too large or too small relative to the expected behaviour of that cumulative sum for random sequences. This cumulative sum may be considered as a random walk. For a random sequence, the excursions of the random walk should be near zero. For certain types of non-random sequences, the excursions of this random walk from zero will be large.

It is recommended that each sequence to be tested consist of a minimum of 100 bits (i.e., $n \geq 100$).

Formulas used

The most important formula to check the randomness of the sequence:

$$P\text{-}value = 1 - \sum_{k=\left(\frac{-n}{z}+1\right)/4}^{\left(\frac{n}{z}-1\right)/4} \left[\Phi\left(\frac{(4k+1)z}{\sqrt{n}}\right) - \Phi\left(\frac{(4k-1)z}{\sqrt{n}}\right)\right] +$$

$$\sum_{k=\left(\frac{-n}{z}-3\right)/4}^{\left(\frac{n}{z}-1\right)/4} \left[\Phi\left(\frac{(4k+3)z}{\sqrt{n}}\right) - \Phi\left(\frac{(4k+1)z}{\sqrt{n}}\right)\right]$$

Where, Φ = Standard Normal Cumulative Probability Distribution Function

n = length of the sequence

z = test static

Procedure

1) Form a normalized sequence: The zeros and ones of the input sequence (ε) are converted to values X_i of –1 and +1 using $X_i = 2\varepsilon_i - 1$. For example, if $\varepsilon = 1011010111$, then $X = 1, (-1), 1, 1, (-1), 1, (-1), 1, 1, 1$.
2) Compute partial sums S_i of successively larger sub sequences, each starting with X_1 (if $mode = 0$) or X_n (if $mode = 1$)

Mode = 0 (forward)	Mode = 1 (backward)
$S_1 = X_1$ $S_2 = X_1 + X_2$ $S_3 = X_1 + X_2 + X_3$ . . $S_k = X_1 + X_2 + X_3 + \ldots + X_k$ . . $S_n = X_1 + X_2 + X_3 + \ldots + X_k + \ldots + X_n$	$S_1 = X_n$ $S_2 = X_n + X_{n-1}$ $S_3 = X_n + X_{n-1} + X_{n-2}$ . . $S_k = X_n + X_{n-1} + X_{n-2} + \ldots + X_{n-k+1}$ . . $S_n = X_n + X_{n-1} + X_{n-2} + \ldots + X_{k-1} + \ldots + X_1$

$$S_1 = 1$$
$$S_2 = 1 + (-1) = 0$$
$$S_3 = 1 + (-1) + 1 = 1$$
$$S_4 = 1 + (-1) + 1 + 1 = 2$$
$$S_5 = 1 + (-1) + 1 + 1 + (-1) = 1$$
$$S_6 = 1 + (-1) + 1 + 1 + (-1) + 1 = 2$$
$$S_7 = 1 + (-1) + 1 + 1 + (-1) + 1 + (-1) = 1$$
$$S_8 = 1 + (-1) + 1 + 1 + (-1) + 1 + (-1) + 1 = 2$$
$$S_9 = 1 + (-1) + 1 + 1 + (-1) + 1 + (-1) + 1 + 1 = 3$$
$$S_{10} = 1 + (-1) + 1 + 1 + (-1) + 1 + (-1) + 1 + 1 + 1 = 4$$

3) Compute the test statistic $z = max_{1\le k\le n}|S_k|$, where $max_{1\le k\le n}|S_k|$ is the largest of the absolute values of the partial sums S_k.

   For the example in this section, the largest value of S_k is 4, so z= 4.

4) Compute *P-value* using the formula:

$$P\text{-}value = 1 - \sum_{k=\left(\frac{-n}{z}+1\right)/4}^{\left(\frac{n}{z}-1\right)/4} \left[\Phi\left(\frac{(4k+1)z}{\sqrt{n}} \right) - \Phi\left(\frac{(4k-1)z}{\sqrt{n}} \right) \right] +$$

$$\sum_{k=\left(\frac{-n}{z}-3\right)/4}^{\left(\frac{n}{z}-1\right)/4} \left[\Phi\left(\frac{(4k+3)z}{\sqrt{n}} \right) - \Phi\left(\frac{(4k+1)z}{\sqrt{n}} \right) \right]$$

Decision and Conclusion

Large values of this statistic indicate that there are either "too many ones" or "too many zeros" at the early stages of the sequence for mode=0; when mode = 1, large values of this statistic indicate that there are either "too many ones" or "too many zeros" at the late stages. Small values of the statistic would indicate that ones and zeros are intermixed too evenly.

If the computed *P-value* is < 0.01, then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random

Example

(input) ε= 1100100100001111110110101010001000100001011010001100001000110100110001001100011001100010100010111000

. (input) n = 100

(input) *mode* = 0 (forward) || *mode* = 1 (reverse)

(processing) z = 1.6 (forward) || z= 1.9 (reverse)

(output) *P-value* = 0.219194 (forward) || *P-value* = 0.114866 (reverse)

(conclusion) Since *P-value > 0.01*, accept the sequence as random.

JAVA code for Test for Cumulative Sums Test

```
import java.util.*;
class cusum
{
    static double CNDF(double x){
    int neg = (x < 0d) ? 1 : 0;
    if ( neg == 1)
        x *= -1d;
    double k = (1d / ( 1d + 0.2316419 * x));
    double y = (((( 1.330274429 * k - 1.821255978) * k + 1.781477937) *
                   k - 0.356563782) * k + 0.319381530) * k;
    y = 1.0 - 0.398942280401 * Math.exp(-0.5 * x * x) * y;
    return (1d - neg) * y + neg * (1d - y);
}
   public static void main(String args[])    {
        int i,z=0;
        double ab=0.0,pb=0.0,obs;
        int a,b,p;
        String x="";
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter the string");
        String s=sc.nextLine();
        int n=s.length();
        int mode=0;
        int s1[]=new int[n];
        int sum[]=new int[n];
        for(i=0;i<n;i++)        {
            s1[i]=(2*Character.getNumericValue(s.charAt(i)))-1;
        }
        if(mode==0) {
            sum[0]=s1[0];
            for(i=1;i<n;i++) {
                sum[i]=sum[i-1]+s1[i];
            }   }
        for(i=0;i<n;i++)
            if(z<sum[i])
                z=sum[i];
        b=(int)(((double)n/(double)z)-1)/4;
        a=(int)(((double)-n/(double)z)+1)/4;
        p=(int)(((double)-n/(double)z)-3)/4;
        for(i=a;i<=b;i++){
            ab=ab+((CNDF((double)((4*i+1)*z)/Math.sqrt(n)))-(CNDF((double)((4*i-1)*z)/Math.sqrt(n))));
        }
       for(i=p;i<=b;i++){
            pb=pb+((CNDF((double)((4*i+3)*z)/Math.sqrt(n)))-(CNDF((double)((4*i+1)*z)/Math.sqrt(n))));
        }
        obs=1-ab+pb;
        if(obs<0.01)
            System.out.println("NON-RNDOM");
        else
            System.out.println("RANDOM");
    }
}
```