ASSIGNMENT NO: BD2

TITLE:

Write a program to generate a pseudorandom number generator for generating the long-term private key and the ephemeral keys used for each signing based on SHA-1 using Python/Java/C++. Disregard the use of existing pseudorandom number generators available.

PREREQUISITES

- 64-bit Fedora or equivalent OS with 64-bit Intel-i5/i7
- Python 2.7

OBJECTIVES:

1. To develop problem solving abilities using Mathematical Modeling.
2. To understand the use and working of Pseudorandom number generator.

MATHEMATICAL MODEL:

Let P be the solution perspective.

Let, S be the System Such that,
A= {S, E, I, O, F, DD, NDD, success, failure}
Where,
S= Start state,
E= End State,
I= Set of Input
O= Set of Out put
F =Set of Function
DD=Deterministic Data
NDD=Non Deterministic Data
Success Case: It is the case a pseudorandom number is generated.
Failure Case: It is the case when some exception occurs and pseudorandom number is not generated.

THEORY:

For the purpose of generating pseudorandom number we are using Mersenne Twister Algorithm.

The **Mersenne Twister** is a pseudorandom number generator (PRNG). It is by far the most widely used general-purpose PRNG.[1] Its name derives from the fact that its period length is chosen to be a Mersenne prime.

For a *w*-bit word length, the Mersenne Twister generates integers in the range $[0, 2^w-1]$.

The Mersenne Twister algorithm is based on a matrix linear recurrence over a finite binary field F_2. The algorithm is a twistedgeneralised feedback shift register[41] (twisted GFSR, or TGFSR) of rational normal form (TGFSR(R)), with state bit reflection and tempering. The basic idea is to define a series x_i through a simple recurrence relation, and then output numbers of the form $x_i T$, where T is an invertible F_2 matrix called a tempering matrix.

The general algorithm is characterized by the following quantities (some of these explanations make sense only after reading the rest of the algorithm):

- w: word size (in number of bits)
- n: degree of recurrence
- m: middle word, an offset used in the recurrence relation defining the series $\boldsymbol{x}$, $1 \le m < n$
- r: separation point of one word, or the number of bits of the lower bitmask, $0 \le r \le w - 1$
- a: coefficients of the rational normal form twist matrix
- b, c: TGFSR(R) tempering bitmasks
- s, t: TGFSR(R) tempering bit shifts
- u, d, l: additional Mersenne Twister tempering bit shifts/masks

with the restriction that $2^{nw - r} - 1$ is a Mersenne prime. This choice simplifies the primitivity test and k-distribution test that are needed in the parameter search.

The series $\boldsymbol{x}$ is defined as a series of w-bit quantities with the recurrence relation:

$$x_{k+n} := x_{k+m} \oplus ({x_k}^u \mid {x_{k+1}}^l)A \qquad\qquad k = 0, 1, \ldots$$

where $|$ denotes the bitwise or, $\oplus$ the bitwise exclusive or (XOR), ${x_k}^u$ means the upper $w - r$ bits of x_k, and x_{k+1}^l means the lower r bits of x_{k+1}. The twist transformation $A$is defined in rational normal form as:

$$A = \begin{pmatrix} 0 & I_{w-1} \\ a_{w-1} & (a_{w-2}, \ldots, a_0) \end{pmatrix}$$

with I_{n-1} as the $(n - 1) \times (n - 1)$ identity matrix. The rational normal form has the benefit that multiplication by A can be efficiently expressed as: (remember that here matrix multiplication is being done in F_2, and therefore bitwise XOR takes the place of addition)

$$\boldsymbol{x}A = \begin{cases} \boldsymbol{x} \gg 1 & x_0 = 0 \\ (\boldsymbol{x} \gg 1) \oplus \boldsymbol{a} & x_0 = 1 \end{cases}$$

where x_0 is the lowest order bit of x.

As like TGFSR(R), the Mersenne Twister is cascaded with a tempering transform to compensate for the reduced dimensionality of equidistribution (because of the choice of *A*being in the rational normal form). Note that this is equivalent to using the matrix *A* ***where*** $\mathbf{A} = T^{-1}AT$ for *T* an invertible matrix, and therefore the analysis of characteristic polynomial mentioned below still holds.

As with *A*, we choose a tempering transform to be easily computable, and so do not actually construct *T* itself. The tempering is defined in the case of Mersenne Twister as

$\boldsymbol{y} := \boldsymbol{x} \oplus ((\boldsymbol{x} >> u)\ \&\ \boldsymbol{d})$

$\boldsymbol{y} := \boldsymbol{y} \oplus ((\boldsymbol{y} << s)\ \&\ \boldsymbol{b})$

$\boldsymbol{y} := \boldsymbol{y} \oplus ((\boldsymbol{y} << t)\ \&\ \boldsymbol{c})$

$\boldsymbol{z} := \boldsymbol{y} \oplus (\boldsymbol{y} >> l)$

where ***x*** is the next value from the series, ***y*** a temporary intermediate value, ***z*** the value returned from the algorithm, with <<, >> as the bitwise left and right shifts, and & as the bitwise and. The first and last transforms are added in order to improve lower-bit equidistribution. From the property of TGFSR, $s + t \geq \lfloor w/2 \rfloor - 1$ is required to reach the upper bound of equidistribution for the upper bits.

The coefficients for MT19937 are:

- $(w, n, m, r) = (32, 624, 397, 31)$
- $a = \mathrm{9908B0DF}_{16}$
- $(u, d) = (11, \mathrm{FFFFFFFF}_{16})$
- $(s, b) = (7, \mathrm{9D2C5680}_{16})$
- $(t, c) = (15, \mathrm{EFC60000}_{16})$
- $l = 18$

Note that 32-bit implementations of the Mersenne Twister generally have $d = \mathrm{FFFFFFFF}_{16}$. As a result, the *d* is occasionally omitted from the algorithm description, since the bitwise and with *d* in that case has no effect.

The coefficients for MT19937-64 are:[42]

- $(w, n, m, r) = (64, 312, 156, 31)$
- $a = \mathrm{B5026F5AA96619E9}_{16}$
- $(u, d) = (29, \mathrm{5555555555555555}_{16})$
- $(s, b) = (17, \mathrm{71D67FFFEDA60000}_{16})$
- $(t, c) = (37, \mathrm{FFF7EEE000000000}_{16})$
- $l = 43$

CONCLUSION:

Hence, we have written program which is capable of generating pseudorandom number without using existing pseudorandom number generator available.

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
"""
Based on the pseudocode in https://en.wikipedia.org/wiki/Mersenne_Twister.
Generates uniformly distributed 32-bit integers in the range [0, 232 − 1] with the MT19937
algorithm

Yaşar Arabacı <yasar11732 et gmail nokta com>
"""
# Create a length 624 list to store the state of the generator
MT = [0 for i in xrange(624)]
index = 0

# To get last 32 bits
bitmask_1 = (2 ** 32) - 1

# To get 32. bit
bitmask_2 = 2 ** 31

# To get last 31 bits
bitmask_3 = (2 ** 31) - 1

def initialize_generator(seed):
    "Initialize the generator from a seed"
    global MT
    global bitmask_1
    MT[0] = seed
    for i in xrange(1,624):
        MT[i] = ((1812433253 * MT[i-1]) ^ ((MT[i-1] >> 30) + i)) & bitmask_1


def extract_number():
    """
    Extract a tempered pseudorandom number based on the index-th value,
    calling generate_numbers() every 624 numbers
    """
    global index
    global MT
    if index == 0:
        generate_numbers()
    y = MT[index]
    y ^= y >> 11
    y ^= (y << 7) & 2636928640
    y ^= (y << 15) & 4022730752
```

```
    y ^= y >> 18

    index = (index + 1) % 624
    return y

def generate_numbers():
    "Generate an array of 624 untempered numbers"
    global MT
    for i in xrange(624):
        y = (MT[i] & bitmask_2) + (MT[(i + 1 ) % 624] & bitmask_3)
        MT[i] = MT[(i + 397) % 624] ^ (y >> 1)
        if y % 2 != 0:
            MT[i] ^= 2567483615

if __name__ == "__main__":
    from datetime import datetime
    now = datetime.now()
    initialize_generator(now.microsecond)
    for i in xrange(5):
        "Print 100 random numbers as an example"
        print extract_number()
'''
C:\Users\neera\Documents\be-2\BD2(no writeup)>python rngmt.py
2830386514
514528569
2208694548
302490786
331860162
'''
```