Practical Assignment 4 (PA4)

Name:	Ashish Kumar
Roll No:	18075068
Dept:	CSE B.Tech. 4th Year
Course:	Network Security (CSE-537)

Practical Assignment 4

Implement any two PRNGs in a language of your choice. Run any two randomness tests over them and report the result.

As usual, the Practical Assignment submission must include a report (pdf) consisting of basic theory, explanation of interesting code snippets, screen shots and a Github link where your code / data is deposited.

GitHub link (Source code): https://github.com/krashish8/netsec-assignments

(Exact Link of this Assignment)

PRNG:

A PRNG creates several "pseudorandom" integers from one or more inputs. Seeds are the inputs to PRNGs. The seed itself must be random and unexpected in circumstances where unpredictability is required. As a result, by default, a PRNG should get its seeds from a RNG's outputs; in other words, a PRNG needs the presence of a RNG.

PRNG implemented in this assignment:

(a) Blum Blum Shub Generator (BBSG):

- The Blum Blum Shub generator takes two primes p and q as an input, which must leave the remainder 3 when divided by 4.
- A random number (seed) is selected in the interval [1, pq-1], such that the gcd(s, pq) = 1. For the purpose of this assignment, I have given seed as an input to the generator, so that consistent results are obtained.

- The subsequent bits are generated using the algorithm:

```
$X_0 = s^2 mod\, n$
$for\ i\ =\ 1\ to\ N$:
    $X_i = (X_{i-1})^2\ mod\ n$
    $B_i = X_i\ mod\ 2$
```

Since the parameters p, q, s vary, the original sequence can't be reconstructed by anyone not knowing these values.

(b) Exclusive OR Generator (XORG):

- The Exclusive OR Generator takes only a seed as an input, which must be of 127-bits.
- Let the initial bit sequence (from the seed) be $x_0,\ x_1,\ x_2,\ \dots,\ x_{126}$
- The subsequent bits are generated using the rule:
    - $x_i = x_{i-1} \oplus x_{i-127};\ for\ i\ \geq\ 127$

Since the initial seed varies, anyone not knowing the seed can't guess the sequence of bytes produced.

Randomness Tests:

The Randomness Tests determine whether the sequence generated by a PRNG is random or not. A good random number generator must pass these tests.

The Randomness Tests I have implemented in this assignment are:

(a) Frequency Test:

The goal of this test is to see if the count of ones and zeros in a sequence are about the same as they would be in a really random sequence. It works as follows:

- The test takes a sequence $s_n = c_0 c_1 \dots c_{n-1}$ as input, where c_i are bits.

- Conversion to $\pm$ 1: The difference between the number of 1s and 0s is calculated. Mathematically, the 0s in the sequence are converted to -1, and then all the bits in the sequence are added. ($S = \Sigma X_i$, where $X_i = 2c_i - 1$)
- Computing the test statistic, $s_{obs} = \frac{|S|}{\sqrt{n}}$
- Computing the P-value, $Pvalue = erfc\left(\frac{s_{obs}}{\sqrt{2}}\right)$, where erfc(.) is the complementary error function.
- Decision: If P-value < 0.01, then the sequence is non-random, otherwise the sequence is considered to be random.

(b) Runs Test:

The runs test is used to see if the number of runs of ones and zeros of varied lengths matches what one would anticipate from a random sequence. It works as follows:

- The test takes a sequence $s_n = c_0 c_1 ... c_{n-1}$ as input, where c_i are bits.
- Computing proportion of 1s: The proportion of 1s in the sequence is calculated as $\omega = \frac{Number\ of\ ones}{n}$
- Computing the test statistic, $s_{obs} = \sum_{k=1}^{n-1} r(k) + 1$, where r(k) = 0, if $c_k = c_{k+1}$, else r(k) = 1.
- Computing the P-value, $Pvalue = erfc\left(\frac{|s_{obs} - 2n\omega(1-\omega)|}{2\sqrt{2n}\,\omega(1-\omega)}\right)$, where erfc(.) is the complementary error function.
- Decision: If P-value < 0.01, then the sequence is non-random, otherwise the sequence is considered to be random.

Source Code:

The well-commented source code is shown below:

For complete source code, please refer to the repository link.

Blum Blum Shub Generator (blum_blum_shub.py)

```python
class BlumBlumShubGenerator:
    def __init__(self, p, q, seed):
        """
        Blum Blum Shub generator
        Takes p and q (prime numbers) and seed (integer) as an input
        The prime numbers must leave remainder 3 when divided by 4
        """
        self.p = p
        self.q = q
        self.n = p * q
        self.seed = seed
        self._check_seed()
        self.x = (self.seed * self.seed) % self.n

    def next(self):
        """
        Get the next bit
        """
        self.x = (self.x * self.x) % self.n
        return self.x % 2

    def _check_seed(self):
        """
        Check whether the seed is valid (relatively prime to n)
        and in the proper range
        """
        if self.seed < 0 or self.seed > self.n:
            raise ValueError("Seed must be in range [0, n]")
        if self._gcd(self.seed, self.n) != 1:
            raise ValueError("Seed must be relatively prime to n")
    def _gcd(self, a, b):
        """
        Euclidean algorithm to find the greatest common divisor
        """
        while b != 0:
            a, b = b, a % b
        return a
```

Exclusive OR Generator (exclusive_or.py)

```
class ExclusiveOrGenerator:
    def __init__(self, seed):
        """
        Exclusive OR generator
        """
        self.seed = seed
        self._check_seed()

    def next(self):
        """
        Get the next bit
        """
        l = len(self.seed)
        # xor of x[i-1] and x[i-127] is the next character
        c = str(int(self.seed[l-1]) ^ int(self.seed[l-127]))
        self.seed += c
        return int(c)

    def _check_seed(self):
        """
        Check that the seed is 127-bit string
        """
        if len(self.seed) != 127:
            raise ValueError("Seed must be of 127-bits")
        for i in self.seed:
            if i != '0' and i != '1':
                raise ValueError("Seed must be binary (0 or 1)")
```

Frequency Test (frequency_test.py)

```
import math

class FrequencyTest:
    def __init__(self, data):
        """
        Frequency Test, data is a string of 0 and 1
        """
        self.data = data

    def run(self):
        """
        Run the frequency test
        """
        p_value = self._p_value()
        print("P-value (Frequency Test):", p_value)
        if p_value < 0.01:
            print("FAILED Frequency Test: The sequence is not random")
        else:
            print("PASSED Frequency Test: The sequence is random")


    def _s_n(self):
        """
        if char in data is '1', add 1, else add -1
        """
        return sum(1 if char == '1' else -1 for char in self.data)

    def _s_obs(self):
        """
        abs(s_n) / sqrt(n), where n is length of the data
        """
        s_n = self._s_n()
        return abs(s_n) / math.sqrt(len(self.data))

    def _p_value(self):
        """
        p_value = erfc(s_obs / sqrt(2))
        """
        s_obs = self._s_obs()
        return math.erfc(s_obs / math.sqrt(2))
```

Runs Test (runs_test.py)

```
import math

class RunsTest:
    def __init__(self, data):
        """
        Runs Test, data is a string of 0 and 1
        """
        self.data = data

    def run(self):
        """
        Run the Runs Test
        """
        p_value = self._p_value()
        print("P-value (Runs Test):", p_value)
        if p_value < 0.01:
            print("FAILED Runs Test: The sequence is not random")
        else:
            print("PASSED Runs Test: The sequence is random")

    def _ones_proportion(self):
        """
        prop = number of '1' in data / length of data
        """
        return self.data.count('1') / len(self.data)
    def _v_n_obs(self):
        """
        sum of r(k) + 1, where r(k) = 0 if data[k] = data[k+1] are same,
        else 1
        """
        return 1 + sum(0 if self.data[k] == self.data[k+1]
                                 else 1 for k in range(len(self.data) - 1))

    def _p_value(self):
        """
        p_value = erfc(abs(v_n_obs - 2*n*prop*(1-prop)) /
                        (2*sqrt(2*n)*prop*(1-prop)))
        """
        n = len(self.data)
        prop = self._ones_proportion()
        v_n_obs = self._v_n_obs()
        return math.erfc(abs(v_n_obs - 2*n*prop*(1-prop))
                    / (2*math.sqrt(2*n)*prop*(1-prop)))
```

The main code for importing and running the above codes:

Main (main.py)

```
from frequency_test import FrequencyTest
from runs_test import RunsTest

from exclusive_or import ExclusiveOrGenerator
from blum_blum_shub import BlumBlumShubGenerator


def num_to_127_bits(num):
    """
    Convert a number to a 127 bit binary string
    """
    return bin(num)[2:].zfill(127)[:127]
```

```python
print("=========================")
print("BLUM BLUM SHUB GENERATOR")
print("=========================")


gen = BlumBlumShubGenerator(p=383, q=503, seed=101355)

rand = ''
for i in range(100):
   rand += str(gen.next())
print("Random stream of bits:", rand)


test = FrequencyTest(rand)
test.run()

test = RunsTest(rand)
test.run()


print("\n=========================")
print("EXCLUSIVE OR GENERATOR")
print("=========================")

gen = ExclusiveOrGenerator(
   num_to_127_bits(270817515257177748616637665753355960446) # random num
)
rand = ''
for i in range(100):
   rand += str(gen.next())
print("Random stream of bits:", rand)


test = FrequencyTest(rand)
test.run()

test = RunsTest(rand)
test.run()
```

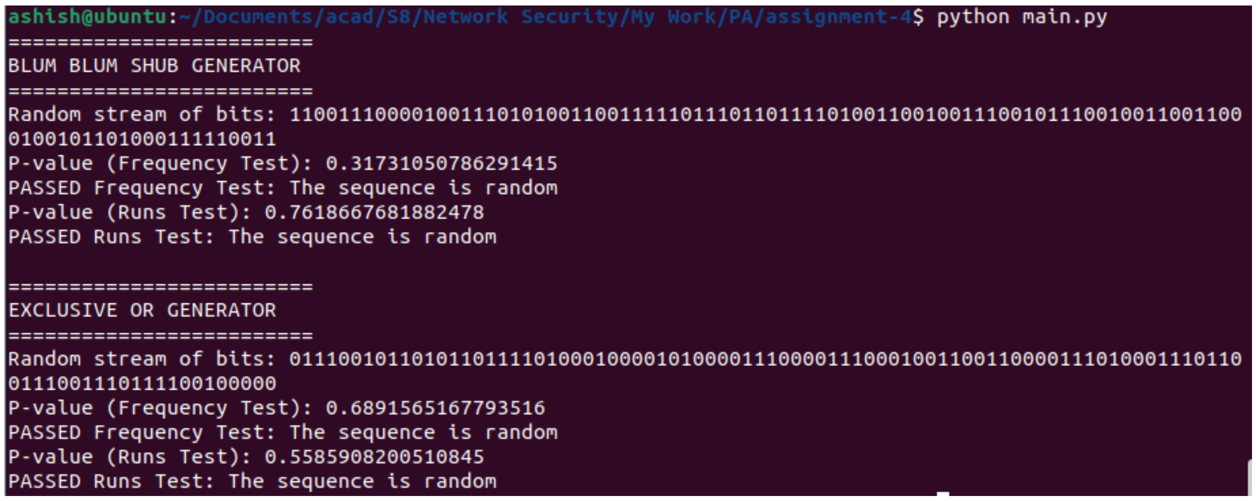
Output:

```
$ python main.py

=========================
BLUM BLUM SHUB GENERATOR
=========================
Random stream of bits:
110011100001001110101001100111110111011011110100110010011100101110010011
00110001001011010001111110011
P-value (Frequency Test): 0.31731050786291415
PASSED Frequency Test: The sequence is random
P-value (Runs Test): 0.7618667681882478
PASSED Runs Test: The sequence is random

=========================
EXCLUSIVE OR GENERATOR
=========================
Random stream of bits:
011100101101011011110100010000101000011100001110001001100110000111010001
11011001110011101111100100000
P-value (Frequency Test): 0.6891565167793516
PASSED Frequency Test: The sequence is random
P-value (Runs Test): 0.5585908200510845
PASSED Runs Test: The sequence is random
```

Screenshot:



```
ashish@ubuntu:~/Documents/acad/S8/Network Security/My Work/PA/assignment-4$ python main.py
=========================
BLUM BLUM SHUB GENERATOR
=========================
Random stream of bits: 110011100001001110101001100111110111011011110100110010011100101110010011001100
01001011010001111110011
P-value (Frequency Test): 0.31731050786291415
PASSED Frequency Test: The sequence is random
P-value (Runs Test): 0.7618667681882478
PASSED Runs Test: The sequence is random

=========================
EXCLUSIVE OR GENERATOR
=========================
Random stream of bits: 011100101101011011110100010000101000011100001110001001100110000111010001110110
01110011101111100100000
P-value (Frequency Test): 0.6891565167793516
PASSED Frequency Test: The sequence is random
P-value (Runs Test): 0.5585908200510845
PASSED Runs Test: The sequence is random
```