Quiz 5

a. Write a Python/C++ program to generate 1M bytes of cryptographically secure random numbers.

Code:

```
import secrets
random_bytes = secrets.token_bytes(10048675)
with open("random.bin", "wb") as file:
    file.write(random_bytes)
```

Explain:

`secret` package use class `random.SystemRandom`, a class for generating random numbers using the highest-quality sources provided by the operating system.

How to Run:

Just execute the following command, and them move `random.bin` file to `sts-2.1.2/data/` for next step.

```
python RNG.py
```

b. Run the NIST SP 800-22 statistical test on your 1M bytes of binary cryptographically secure random numbers and analyze the test results to identify any deviations from the expected statistical properties of random numbers.

After download the NIST SP 800-22 statistical test file, run the command `make` to generate `assess` file. Then run the following command to start testing.

```
./assess 8388608
```

After setting all parameters, we can find our testing result in `./experiments/AlgorithmTesting`.

- **Frequency Test**

  This test focus on the proportion of zeroes and ones for the entire sequence. The purpose is to determine whether the number of ones and zeros in a sequence are approximately the same.

- **Frequency Test within a Block**

  To determine whether the frequency of 1s and 0s in an M-bit block is approximately $\left|\frac{M}{2}\right|$.

- **Runs Test**

  To determine whether the number of identical bits of 1s and 0s of various lengths is as expected for a random sequence.

- **Longest Run Test**

  To determine whether the length of the longest identical bits of 1s is consistent with the length of the longest run of 1s that would be expected in a random sequence.

- **Binary Matrix Rank Test**

  To check for linear dependence among fixed length substrings of the original sequence.

- **Discrete Fourier Transform (Spectral) Test**

  To detect whether the number of peaks in the Discrete Fourier Transform of the sequence exceeding the 95 % threshold is significantly different than 5 %.

- **Non-overlapping Template Matching Test**

  Using a sliding window to detect whether generators produce too many occurrences of a specific pattern, when the pattern is found, the window is reset to the bit after the found pattern.

- **Overlapping Template Matching Test**

  The same with **Non-overlapping Template Matching Test** but the difference is that when the pattern is found, the window slides only one bit.

- **Maurer's "Universal Statistical" Test**

  To detect the number of bits between matching patterns that can be compressed without loss of information.

- **Linear Complexity Test**

  To determine whether the sequence is complex enough to be considered random

- **Serial Test**

  To determine whether the number of occurrences of the 2^m m-bit overlapping patterns is approximately the same.

- **Approximate Entropy Test**

To compare the frequency of overlapping blocks of two consecutive/adjacent lengths (m and m+1) against the expected result for a random sequence.

- **Cumulative Sums Test**

  To determine whether the cumulative sum of the partial sequences is as expected.

- **Random Excursions Test**

  To determine if the number of visits to a particular state within a cycle deviates from what one would expect for a random sequence.

- **Random Excursions Variant Test**

  Count the total number of times that a particular state is visited in a cumulative sum random walk, then detect deviations from the expected number of visits to various states in the random walk

c. Extra credit: Find out a non-cryptographically secure random number generator, such as `random()`, to demonstrate its lack of safety. Then, propose modifications to enhance its security to generate cryptographically secure random numbers that meet the highest standards of security and reliability.

   First we can show that the original random function generate the same random number if we use the same seed, indicating that this is not a cyptographically secure random number generator.

```
import random

for _ in range(10):
    random.seed(1000000)
    print(random.random())
```

   Result:

```
0.9241408041681706
0.9241408041681706
0.9241408041681706
0.9241408041681706
0.9241408041681706
0.9241408041681706
0.9241408041681706
0.9241408041681706
0.9241408041681706
0.9241408041681706
```

We can use `os.urandom` to can integrate a high-entropy source for seeding from the operating system. Then the result show that this is a cyptographically secure random number generator.

```
import os
import random

def get_high_entropy_seed(size=32):
    """Reads a specified number of bytes from /dev/urandom."""
    return os.urandom(size)

for _ in range(10):
    seed_bytes = get_high_entropy_seed()
    int_value = int.from_bytes(seed_bytes, byteorder='big')
    random.seed(int_value)
    print(random.random())
```

Result:

```
0.7876424089418691
0.8054323338091726
0.4271770156291276
0.03280196158599347
0.5068605667475528
0.5749424588588495
0.3362573548247043
0.4094005160829157
0.9498209197368858
0.4657533327657828
```