Linear Congruential Generator

$$X_{n+1} = (a X_n + c) \bmod m$$

with X_0: the starting value or the **seed**

a : multiplier

c : increment

m : the modulus

Note: The parameters a, c, and m are predefined by each implementation of the LCG algorithm and may be subject to certain criteria. To learn more about these parameters and how they are used, please refer to this article on Wikipedia.

For this example, we will use the same parameter values as in the Numerical Recipes book.

Now, let's write a simple code to generate a random sequence with N numbers.

```
def LCG(seed, N, min, max): # min and max are used to scale the random number to the desired range
    Seq = []
    a = 1664525 #The predefined parameters
    c = 1013904223
    m = 2**32
    for i in range(N):
        seed = (a * seed + c) % m  # Update the seed using the LCG formula
        random_num = min + (seed % (max - min))  # Scale the output to the desired range (+1 to make the max value included)
        Seq.append(random_num)

    return Seq

# Generate the sequence, setting the seed to be 2004
LCG(5, 10, 0, 9)
```

```
[8, 7, 4, 6, 3, 3, 8, 6, 3, 7]
```

The Seed Problem:

The LCG algorithm we used above generates the sequence of random numbers through a deterministic process, starting from an initial value called the seed X_0 so, what happens if we generate another random sequence starting from the same seed? Will the sequences be different? Let’s see:

We will fix the seed to 1954, set the sequence length to 20, and choose the minimum value to be 0 and the maximum value to be 100, so if the two sequences were truly randomly generated, they would be different each time.

What is the probability of them being the same if they are truly randomly generated?

- We assume that we have two randomly generated sequences, both generated independently.
- Each value in each sequence has 100 possible values, so the probability for each value to appear is $\frac{1}{100}$
- The sequence length is 20 for each.

So the probability of having an exact match between two truly randomly generated sequences is:

$$P_{match} = \left(\frac{1}{100}\right)^{20} = 10^{-40}$$

which is literally: 0.0000000000000000000000000000000000000001

This is effectively 0, meaning we would need an enormous amount of luck to get two sequences with exactly the same 20 values.

```
#Let's generate the first sequence with 1954 as  our fixed seed in both sequences:

LCG(1954, 20, 0, 100)

[73, 40, 47, 46, 29, 0, 75, 54, 13, 36, 19, 18, 69, 64, 83, 58, 61, 88, 35, 62]

# Now the second one with the same parameters:

LCG(1954, 20, 0, 100)

[73, 40, 47, 46, 29, 0, 75, 54, 13, 36, 19, 18, 69, 64, 83, 58, 61, 88, 35, 62]
```

The Risks of Seed-Based Encryption:

To understand how dangerous it is to have numbers that are not truly random and can be reproduced if we know the algorithm used, let us have a look at an amazing challenge called "Enigma" in the AlphaCTF 2024 edition (an annual cybersecurity competition (Capture The Flag) organized by the AlphaBit club at ESI-SBA University) You can find the challenge and its solution here

Basically, the challenge provides you with a seed-based algorithm to encode a secret message (the Flag). When you try to request their server, you receive an encrypted text that seems meaningless. Knowing that the seed used for the encryption is the timestamp of the encryption, you can reverse the algorithm and decrypt the secret message.

Here's a simplified version of the challenge:

```
import time

def encrypt(message, seed):

    encrypted_message = ""

    random_numbers = LCG(seed, len(message), 0, 256) #Generate a
random sequence to use it for the encryption
    print("This is the random sequence used for encryption" ,
random_numbers, "\n")

    for i, char in enumerate(message):
        encrypted_message += chr(ord(char) ^ random_numbers[i])  # XOR
each character with the random number

    return encrypted_message


time_stamp_of_the_encryption = int(time.time())

print("The time stamp of the encryption:" ,
time_stamp_of_the_encryption, "\n")

the_secret_msg = "THIS IS A VERY SECRET INFORMATION, PLEASE DON'T
SHARE: 1954-1962"

encrypted_msg = encrypt(the_secret_msg, time_stamp_of_the_encryption)

print("The Encrypted message is:", encrypted_msg)

The time stamp of the encryption: 1736220828

This is the random sequence used for encryption [75, 46, 181, 144,
175, 66, 185, 196, 83, 150, 253, 56, 55, 42, 129, 236, 91, 254, 69,
224, 191, 18, 73, 20, 99, 102, 141, 136, 71, 250, 17, 60, 107, 206,
213, 48, 207, 226, 217, 100, 115, 54, 29, 216, 87, 202, 161, 140, 123,
158, 101, 128, 223, 178, 105, 180, 131, 6, 173, 40, 103, 154, 49, 220]


The Encrypted message is: füÃÊêä¶«}es¡¿½¥ë2Z%)ßÅ®Xs%âõ`Ê§Ϯ6Y□íõ¬(Ö$Ò□□Ī□
º3□V£î

#Now let's reverse the encryption algorithm

def decrypt(encrypted_message, seed):
```

```
    decrypted_message = ""

    random_numbers = LCG(seed, len(encrypted_message), 0, 256)  # 
Generate the same sequence if the seed is the same
    print("This is the random sequence used for decryption " ,
random_numbers, "\n")

    for i, char in enumerate(encrypted_message):
        decrypted_message += chr(ord(char) ^ random_numbers[i])  # XOR 
with the same numbers to decrypt (XOR is the inverse of itself)

    return decrypted_message


decrypted_msg = decrypt(encrypted_msg, time_stamp_of_the_encryption)


print("The Decrypted message is:", decrypted_msg)
```

```
This is the random sequence used for decryption  [75, 46, 181, 144, 
175, 66, 185, 196, 83, 150, 253, 56, 55, 42, 129, 236, 91, 254, 69, 
224, 191, 18, 73, 20, 99, 102, 141, 136, 71, 250, 17, 60, 107, 206, 
213, 48, 207, 226, 217, 100, 115, 54, 29, 216, 87, 202, 161, 140, 123, 
158, 101, 128, 223, 178, 105, 180, 131, 6, 173, 40, 103, 154, 49, 220]


The Decrypted message is: THIS IS A VERY SECRET INFORMATION, PLEASE 
DON'T SHARE: 1954-1962
```

Luckily, encryption algorithms are much more complex, such as RSA, and the choice of the seed is also more complicated. In addition to the confidentiality of the algorithms used, the keys, and the security of data transformation and data centers, there are many layers of security involved.

However, sometimes information like what we've mentioned above can be leaked, and hackers may find themselves only facing the encryption algorithm. In such a scenario, encoding confidential materials requires strong foundational principles, and since these algorithms rely on randomness, the random number generation used must be truly random.

In all the following we assume that the Bitstream is a Bits List

Autocorrelation Calculation Function

```
def calculate_autocorrelation(bitlist, lag):

    n = len(bitlist)
    if lag >= n:
        raise ValueError("Lag must be less than the length of the 
bitlist.")
```

```
    bitlist = np.array(bitlist)

    # Calculate the mean of the bitlist
    mean = np.mean(bitlist)

    # Calculate the autocovariance and variance
    autocovariance = np.sum((bitlist[:n-lag] - mean) * (bitlist[lag:]
- mean)) / (n - lag)
    variance = np.var(bitlist)

    # Calculate autocorrelation
    autocorrelation = autocovariance / variance

    return autocorrelation
```

Shanon Entropy Calculation -Normalized-

```
import numpy as np
from collections import Counter

def calculate_shanon_entropy(bitstring, n):


    # Ensure bitstring length is divisible by n
    bitstring = bitstring[:len(bitstring) - (len(bitstring) % n)]

    # Group bits into n-bit chunks
    chunks = [tuple(bitstring[i:i + n]) for i in range(0,
len(bitstring), n)]

    # Count occurrences of each chunk
    chunk_counts = Counter(chunks)

    # Calculate probabilities
    total_chunks = len(chunks)
    probabilities = np.array([count / total_chunks for count in
chunk_counts.values()])

    # Calculate entropy
    entropy = -np.sum(probabilities * np.log2(probabilities))

    # Normalize the entropy
    normalized_entropy = entropy / n

    return normalized_entropy
```

Extraction Efficiency Calculation (ExE)

```
def calculate_ExE(input, output):
    return output / input
```

Von Neumann Post Processor

```
class VonNeumannPP:
    def __init__(self):
        pass

    def postprocess(self, input, input2 = None):
        if input2 is None:
            sample_1, sample_2 = input[:(len(input)//2)],
input[(len(input)//2):]

        else:
            sample_1, sample_2 = input, input2

        # Ensure both samples are of equal length by truncating to the
length of the smaller sample
        min_length = min(len(sample_1), len(sample_2))
        sample_1 = sample_1[:min_length]
        sample_2 = sample_2[:min_length]

        bits_1 = np.ravel(np.array(sample_1) == 0).astype(np.int8)
        bits_2 = np.ravel(np.array(sample_2) == 0).astype(np.int8)

        arr = np.where(bits_1 > bits_2, np.zeros_like(bits_1),
np.ones_like(bits_1))
        arr = np.where(bits_1 == bits_2, np.nan *
np.ones_like(bits_1), arr)

        return arr[~np.isnan(arr)].astype(np.int8)

list(VonNeumannPP().postprocess([0,1,1,0,0,1,0,1,0,1,0,]))
```

```
[0, 1, 0]
```

Markov Chain with 1-bit histroy Post Processor

```
import numpy as np
from collections import deque

class MKV1:
    def postprocess(self, bitstring):
```

```
        if len(bitstring) < 2:
            raise ValueError("Bitstring must have at least 2 bits for
decorrelation.")

        # Initialize queues for previous bit history
        queue0, queue1 = deque(), deque()
        de_correlated_bits = [bitstring[0]]  # First bit is taken as
is

        # Process bitstring starting from the second bit
        for i in range(1, len(bitstring)):
            current_bit = bitstring[i]
            previous_bit = bitstring[i - 1]

            # Route current bit to the appropriate queue
            if previous_bit == 0:
                queue0.append(current_bit)
            else:
                queue1.append(current_bit)

        # Merge queues into the output
        de_correlated_bits.extend(queue0)
        de_correlated_bits.extend(queue1)

        return np.array(de_correlated_bits, dtype=np.int8)

list(MKV1().postprocess([0,1,1,0,0,1,0,1,0,1,0,]))

[0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0]
```

Markov Chain with 2-bits histroy Post Processor

```
import numpy as np
from collections import deque

class MKV2:
    def postprocess(self, bitstring):
        if len(bitstring) < 3:
            raise ValueError("Bitstring must have at least 3 bits for
2-bit history decorrelation.")

        # Initialize 4 queues for 2-bit history
        queues = {
            "00": deque(),
            "01": deque(),
            "10": deque(),
            "11": deque()
        }
```

```
        de_correlated_bits = [bitstring[0], bitstring[1]]  # First two
bits are taken as is

        # Process bitstring starting from the third bit
        for i in range(2, len(bitstring)):
            current_bit = bitstring[i]
            previous_bits = f"{bitstring[i - 2]}{bitstring[i - 1]}"

            # Route current bit to the appropriate queue
            queues[previous_bits].append(current_bit)

        # Merge queues into the output
        for key in ["00", "01", "10", "11"]:
            de_correlated_bits.extend(queues[key])

        return np.array(de_correlated_bits, dtype=np.int8)

list(MKV1().postprocess([0,1,1,0,0,1,0,1,0,1,0,]))
```

```
[0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0]
```

Generate Markov Model with 1-bit Memory Correlated Bitlist

```
import numpy as np

def generate_markov1_bitlist(length, P1=0.5, phi1=0.0):
    """
    Generates a bitlist using a 1-bit Markov chain model (MKV1).

    Args:
        length (int): Length of the generated bitlist.
        P1 (float): Probability of state to be 1.
        phi1 (float): Autocorrelation at lag 1.

    Returns:
        list: Generated bitlist.
    """
    # Calculate transition probabilities using the MKV1 model
    T01 = P1 * (1 - phi1)
    T11 = phi1 + T01
    T00 = 1 - T01
    T10 = 1 - T11

    # Validate probabilities
    if not (0 <= T01 <= 1 and 0 <= T11 <= 1):
        raise ValueError("Transition probabilities must be between 0
and 1.")
```

```
    # Initial state probabilities
    P0 = 1 - P1

    # Generate the bitlist
    bitlist = [np.random.choice([0, 1], p=[P0, P1])]

    for _ in range(1, length):
        current_state = bitlist[-1]
        if current_state == 0:
            next_bit = np.random.choice([0, 1], p=[T00, T01])
        else:
            next_bit = np.random.choice([0, 1], p=[T10, T11])
        bitlist.append(next_bit)

    return bitlist

calculate_autocorrelation(generate_markov_bitlist(10000, 0.5, 0.6),1)
#The result converge to 0.6 to phi1 when p is 0.5
```

```
0.5982346566021407
```

The CQTPost Processor

```
class CQTPP:
    """Implementation of the CQTPP

    Parameters:
        dep_seq_len (int): The length of the dependent sequences.
    """

    def __init__(self, **kwargs):
        self.__dep_seq_len = kwargs.get("dep_seq_len")
        if self.__dep_seq_len is None:
            self.__dep_seq_len = 1 #It becomes Von Neuman Post
Processor


    def postprocess(self, input):
        ouput = np.array([], dtype=np.int8)
        sample_1, sample_2 = split_bitstring(input)
        for i in range(len(sample_1) // self.__dep_seq_len):
            sub_s1 = sample_1[i * self.__dep_seq_len : (i + 1) *
self.__dep_seq_len]
            sub_s2 = sample_2[i * self.__dep_seq_len : (i + 1) *
self.__dep_seq_len]
            postprocess_output = VonNeumannPP().postprocess(sub_s1,
sub_s2)
            if np.size(postprocess_output):
```

```
                ouput = np.append(ouput,
np.array([postprocess_output[0]]))
        return ouput

CQTPP(dep_seq_len = 2).postprocess(generate_markov_bitlist(1000, 0.5,
0.6))
```

```
array([1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0,
1,
       1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1,
1,
       1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1,
0,
       0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1,
0,
       0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0,
0,
       0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
1,
       1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1,
1,
       1, 0, 0, 1, 1], dtype=int8)
```

The Itterative CQTPostProcessor

```
class IterCQTPP(PostProcessor):
    """Implementation of IterCQTPP

    Parameters:
        dep_seq_len (int): The length of the dependent sequences.
        it (int): The number of iterations for the post-processing.

    """

    def __init__(self, **kwargs):
        self.__dep_seq_len = kwargs.get("dep_seq_len")
        self.__iterations = kwargs.get("it")


        if self.__dep_seq_len is None:
            self.__dep_seq_len = 1
        if self.__iterations is None:
            self.__iterations = 1


    def postprocess(self, sample_1, sample_2 = None) -> np.ndarray:

        ouput = np.array([], dtype=np.int8)
```

```
        if sample_2 is None:
            sample_1, sample_2 = sample_1[:(len(sample_1)//2)], 
sample_1[(len(sample_1)//2):]

        oui, dii = 
CQTPP(dep_seq_len=self.__dpq).postprocess2(sample_1)
        out_final = oui
        for _ in range(self.__iterations - 1):
            oui, dii = CQTPP(dep_seq_len=self.__dpq).postprocess2(dii)
            out_final = np.append(out_final, oui)

        return out_final
```

Generationg all possible combinations of a bitstream with length N,and post porcess them using CQTPP and VNPP

```
import itertools
import numpy as np
from tqdm import tqdm

def Combinations_Cqtpp_Vnpp(N):


    # Generate all combinations of N-bit binary numbers
    combinations = list(itertools.product([0, 1], repeat=N))

    # Initialize the postprocessors
    post_processor_1 = CQTPP(dep_seq_len=2)
    post_processor_2 = VonNeumannPP()

    # Process each combination using both post-processors and store 
results
    results = []
    for bits in combinations:
        bits_list = list(bits)  # Ensure input is a list
        processed_1 = post_processor_1.postprocess(bits_list)  # 
Process as a bit list
        processed_2 = post_processor_2.postprocess(bits_list)  # 
Process as a bit list
        results.append((bits, processed_1, processed_2))

    # Output the table
    print(f"{'Bits':>{N*2}} | CQTPP          | VonNeuPP")
    print("-" * (N * 2 + 30))
    for bits, result_1, result_2 in results:
```

```
        bits_str = ''.join(map(str, bits))
        result_1_str = ''.join(map(str, result_1))
        result_2_str = ''.join(map(str, result_2))

        result_1_str = 'X' if result_1_str == '' else result_1_str
        result_2_str = 'X' if result_2_str == '' else result_2_str
        print(f"{bits_str:>{N*2}} | {result_1_str:13} | 
{result_2_str:13}")

Combinations_Cqtpp_Vnpp(4)
```

```
    Bits | CQTPP          | VonNeuPP
--------------------------------------
    0000 | X             | X
    0001 | 0             | 0
    0010 | 0             | 0
    0011 | 0             | 00
    0100 | 1             | 1
    0101 | X             | X
    0110 | 0             | 01
    0111 | 0             | 0
    1000 | 1             | 1
    1001 | 1             | 10
    1010 | X             | X
    1011 | 0             | 0
    1100 | 1             | 11
    1101 | 1             | 1
    1110 | 1             | 1
    1111 | X             | X
```

VonNeuman_4bits Post Processor

```
def vn4_post_processing(bit_list):
    def count_ones(bits):
        return sum(bits)

    def get_output_bits(group, num_bits):
        # Generate bit assignments for the group
        output = []
        max_value = 2 ** num_bits
        for i in range(len(group)):
            output.append(f"{i % max_value:0{num_bits}b}")
        return output

    # VN_4 mapping: divide bit sequences into groups based on the 
number of 1s
    n = 4
    groups = {k: [] for k in range(n + 1)}
```

```python
    for i in range(0, len(bit_list), n):
        sub_seq = bit_list[i:i + n]
        if len(sub_seq) < n:
            break
        k = count_ones(sub_seq)
        groups[k].append(sub_seq)

    # Assign output bits without bias
    result = []
    for k, group in groups.items():
        if k in (0, n):
            continue  # No output bits for groups S0 or S4
        num_bits = 2 if k in (1, 3) else 1
        output_bits = get_output_bits(group, num_bits)
        result.extend(output_bits)

    return [int(bit) for string in result for bit in string]
```