Geradores de números pseudo-aleatórios:
Inversive congruential, Lagged Fibonacci e Linear congruential

Lucas João Martins

1 Códigos das implementações

icg.py

```
# -*- coding: utf-8 -*-
"""Implementacao do 'Inversive congruential generator'"""


class icg:
    """Classe que constroi o gerador de numeros
       pseudo-aleatorios

    Atributos:
        seed: semente do gerador
        q: numero que e feito o modulo da formula
        a: numero multiplicador da formula
        c: numero que faz a adicao da formula, ou, pode ser
           utilizado como um
           resultado
    """

    def __init__(self, seed, q, a, c):
        """Construtor com todos os atributos obrigatorios

        Args:
            ja especificado na classe
        """
        self.seed = seed
        self.q = q
        self.a = a
        self.c = c

    def generate(self, n):
        """Gerador de numeros pseudo-aleatorios

        Gera uma lista com n-1 numeros pseudo-aleatorios
           conforme o algoritmo
        icg ou seja, utiliza da seguinte relacao:
```

```
            x_0 = seed
            x_{i+1} = (a * x^{-1}_i + c) mod q  if x_i != 0
            x_{i+1} = c                         if x_i == 0
                Onde x^{-1}_i e a funcao modular inversa

        Args:
            n: valor representa quantidade de numeros que serao gerados - 1

        Returns:
            lista com os n + 1 numeros gerados
        """
        result = [self.seed]
        for i in range(n):
            number = 0
            if result[i] == 0:
                number = self.c
            else:
                x = (self.a * self.modular_inverse(result[i]) + self.c)
                number = x % self.q

            result.append(number)

        return result

    def modular_inverse(self, x):
        """Funcao que calcula a modular inversa de um numero

        Forma 'naive' de se calcular a modular inversa de um numero x mod q,
        onde q e informado na construcao do gerador. Considerada 'naive' por nao
        usar do algoritmo de euclides estendido. Funciona da seguinte maneira:
            Calcula x * i mod q para todo i entre 0 e q - 1, o i que gerar o
            resultado 1 sera a modular inversa

        Args:
            x: numero do qual se quer saber a modular inversa com q

        Returns:
            a modular inversa de um numero x mod q

        Raises:
            Exception: quando nao existe modular inversa
        """
```

```
        for i in range(self.q):
            result = (x * i) % self.q
            if result == 1:
                return i

        raise Exception("Nao foi possivel calcular a
           'inversa modular'")

if __name__ == '__main__':
    # nao usar sys.argv porque demora para gerar numeros com
       mais de 30 bits
    psng = icg(109951162775, 100110021, 10, 1099511627775)
    print(psng.generate(5))
```

lfg.py

```
# -*- coding: utf-8 -*-
"""Implementacao do 'Lagged Fibonacci generator'"""

import random
import sys


class lfg:
    """Classe que constroi o gerador de numeros
       pseudo-aleatorios

    Atributos:
        lags: tupla de dois valores que irao representar os
           'lags' da formula
        exponent: expoente que ira na potencia de 2 que e
           feita de modulo da
        formula
    """

    def __init__(self, lags, exponent):
        """Construtor com todos os atributos obrigatorios

        Args:
            seed: lista vazia que ira conter a semente do
               gerador
            random: valor utilizado na montagem da semente
            alem dos ja especificados na classe
        """
        self.j = lags[0]
        self.k = lags[1]
        self.m = 2**exponent
        self.seed = []
        self.random = 10000

```

```
    def make_seed(self):
        """Monta a lista de numeros que sera utilizada como semente no gerador

        Para o funcionamento correto do algoritmo e necessario que k valores
        aleatorios sejam utilizados como semente. Nessa implementacao isso e
        feito da seguinte forma:
            Utiliza-se o modulo random do python para gerar um numero entre 0 e
            1, para assim multiplicar esse valor por 10000
            (um valor arbitrario) e por fim pegar a parte inteira desse
            resultado e adicionar na lista de sementes
        """
        i = 0
        while i < self.k:
            self.seed.append((int(random.random() * self.random)))
            i += 1

    def valid_amount(self, n):
        """Verifica a validade dos argumentos utilizados na geracao dos numeros

        Argumentos serao validos quando:
            j < k
            n - j < k
            n - k < k

        Args:
            n: quantidade de numeros que serao gerados

        Returns:
            True se os argumentos sao validos, senao False
        """
        return True if n - self.j < self.k and self.j < self.k else False

    def alfg_generate(self, n):
        """Gerador de numeros pseudo-aleatorios com a operacao de adicao

        Gera uma lista com n numeros pseudo-aleatorios conforme o algoritmo
        alfg (additive lagged fibonacci generator), ou seja, ha outras versoes
        que ao inves da adicao para montar o p do algoritmo
```

```
            utilizam da:
            subtracao
            multiplicacao
            xor
        O algoritmo utiliza da seguinte relacao:
            Para cada numero que sera gerado a lista de sementes e iterada:
                Quando e o primeiro elemento da lista de sementes:
                    Calcula-se o pseudo-aleatorio p:
                        p  = (seed_{n - j} + seed_{n - k}) mod m
                            Onde seed e a lista de sementes
                Quando e o ultimo elemento da lista de sementes:
                    A posicao recebe o pseudo-aleatorio calculado
                Nos outros casos:
                    O elemento a direita da lista vem para a posicao atual

        Args:
            n: quantidade de numeros que serao gerados

        Returns:
            lista com os n numeros gerados

        Raises:
            Exception: quando os argumentos sao considerados invalidos
        """
        if not self.valid_amount(n):
            raise Exception("Valores invalidos!")

        result = []
        self.make_seed()
        i = 0
        while i < n:
            for j in range(self.k):
                if j == 0:
                    x = self.seed[n - self.j] + self.seed[n - self.k]
                    number = x % self.m
                elif 0 < j < self.k - 1:
                    self.seed[j] = self.seed[j + 1]
                else:
                    self.seed[j] = number
                    result.append(number)
            i += 1
```

```python

        return result

if __name__ == '__main__':
    psng = lfg((70, 100), int(sys.argv[2]))
    psng.random = int(sys.argv[1])
    print(psng.alfg_generate(5))
```

lcg.py

```python
# -*- coding: utf-8 -*-
"""Implementacao do 'Linear congruential generator'"""

import sys


class lcg:
    """Classe que constroi o gerador de numeros pseudo-aleatorios

    Atributos:
        multiplier: numero multiplicador da formula
        increment: numero que faz a adicao da formula
        seed: semente do gerador
        modulus: numero que e feito o modulo da formula
    """

    def __init__(self, multiplier, increment, seed, modulus):
        """Construtor com todos os atributos obrigatorios

        So constroi o gerador se os atributos sao considerados validos para a
        geracao dos numeros

        Args:
            ja especificado na classe

        Raises:
            Exception: quando os argumentos sao considerados invalidos
        """
        if modulus > 0 and 0 < multiplier < modulus and \
           0 <= increment < modulus and 0 <= seed < modulus:
            self.a = multiplier
            self.c = increment
            self.r0 = seed
            self.m = modulus
        else:
            raise Exception("Argumentos invalidos!")

```

```
    def generate(self, n):
        """Gerador de numeros pseudo-aleatorios

        Gera uma lista com n numeros pseudo-aleatorios conforme o algoritmo lcg,
        ou seja, utiliza da seguinte relacao:
            X_{n+1} = (a * X_n + c) mod m
                Quando n = 0, entao X_n = semente do gerador

        Args:
            n: quantidade de numeros que serao gerados

        Returns:
            lista com os n numeros gerados
        """
        result = []
        for i in range(n):
            if i == 0:
                previous = self.r0
            else:
                previous = result[i-1]

            result.append((self.a * previous + self.c) % self.m)

        return result

if __name__ == '__main__':
    psng = lcg(1664525, 1013904223, int(sys.argv[1]), 2**int(sys.argv[2]))
    print(psng.generate(5))
```

2 Explicação dos algoritmos

O inversive congruential generator, também conhecido como ICG, foi desenvolvido por Eichenauer e Lehn em 1986 e trata-se de um gerador de números pseudo-aleatórios que é baseado em recursão não linear. A sua congruência é definida por:

$$x_{i+1} \equiv a x_i^{-1} + c \pmod{q} \tag{1}$$

Onde x_i^{-1} é a função modular inversa entre x e q. Os outros componentes dessa relação já foram explicados na documentação do código. Importante citar que para um correto funcionamento da relação x e q precisam ser coprimos, e, que essa verificação não é realizada no código implementado. O máximo período que o ICG pode alcançar é de q unidades.

O lagged fibonacci generator, também conhecido como LFG, é um gerador de números pseudo-aleatórios que é baseado em uma generalização da sequência de

Fibonacci. A sua concruência é definida por:

$$p_n \equiv p_{n-j} \star p_{n-k} \pmod{m} \tag{2}$$

Onde $\star$ denota uma operação binária qualquer, ou seja, pode ser adição, subtração, multiplicação ou um xor. Os outros atributos dessa relação já foram explicados na documentação do código. A implementação do gerador nesse trabalho faz uso do additive lagged fibonacci generator, também chamado de ALFG, ou seja, a operação binária escolhida foi a adição. O máximo período que o ALFG pode alcançar é de $(2^k - 1)2^{p-1}$ unidades.

O linear congruential generator, também conhecido como LCG, foi desenvolvido por D. H. Lehmer em 1948 e trata-se de um popular gerador de números pseudo-aleatórios que é baseado em uma equação linear. A sua relação de recorrência é definida por:

$$X_{n+1} = (aX_n + c) \ \ mod \ \ m \tag{3}$$

Os atributos apresentados já foram explicados na documentação do código. O máximo período que o LCG pode alcançar é de m unidades

Todos os três geradores possuem regras associadas as suas variáveis e também as suas formas de operações que podem ser classificadas, de maneira informal, em dois tipos:

1. Essenciais para o funcionamento do gerador;

2. Responsáveis por maximizar o desempenho do gerador.

Optou-se por não implementar regras do segundo tipo, mas mais informações sobre elas podem ser obtidas nas referências citadas. Já com relação às regras do primeiro tipo, experimentou-se utilizar três estratégias diferentes:

- Não permitir a instanciação do gerador, o que foi feito no LCG;
- Não fazer nada em nível de código e esperar que quem o utilize saiba das regras, aplicado no ICG;
- Permitir a instanciação do gerador, mas não permitir a geração dos números, utilizado no LFG.

3 Comparação entre os algoritmos

Ao comparar o LCG com o ICG, percebe-se o seguinte:

- O ICG é a versão não linear do LCG;
- O LCG apresenta diversas regularidades não desejadas devido a sua linearidade;
- O resultado do LCG é reticulado, enquanto que o do ICG não é,

Já ao comparar o LFG contra o LCG e o ICG, pode-se destacar:

Tabela 1: 5 números pseudo-aleatórios grandes

Gerador	Tamanho do número em bits	Tempo gasto
ICG	27	1m15,687s
ICG	mais que 30	Muito tempo, impraticável
LCG	40	0m0,023s
LFG	40	0m0,037s
LCG	56	0m0,027s
LFG	56	0m0,043s
LCG	80	0m0,030s
LFG	80	0m0,035s
LCG	128	0m0,023s
LFG	128	0m0,033s
LCG	168	0m0,027s
LFG	168	0m0,040s
LCG	512	0m0,027s
LFG	512	0m0,043s
LCG	2048	0m0,023s
LFG	2048	Overflow
LCG	4096	0m0,023s

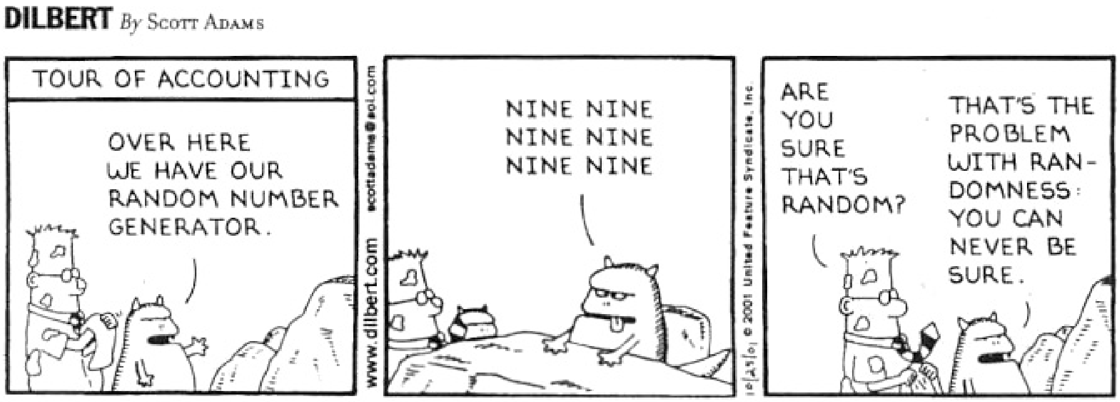
- O LFG requer que k unidades sejam armazenadas para poder gerar um único número, enquanto que os outros dois só necessitam do armazenamento de uma unidade
- O período máximo do LCG e do ICG é limitado ao valor do seu módulo, enquanto que o LFG não é, e, por isso pode atingir um valor bem maior do que o dos outros dois já citados

4 Complexidade dos algoritmos

- Inversive congruential generator: $O(n * q)$
  - Devido aos laços nas linhas 44 e 74 do código.
- Lagged Fibonacci generator: $O(n * k)$
  - Devido aos laços nas linhas 94 e 95 do código.
- Linear congruential generator: $O(n)$
  - Devido ao laço na linha 51 do código.

5 Os números são aleatórios?

Uma resposta curta: você não consegue afirmar se algo é aleatório ou não, conforme Dilbert já falou um dia.



Uma resposta longa: há diferentes maneiras de interpretar o significado da aleatoriedade, pois isso pode ser feito do ponto de vista estatístico, ou do ponto de vista da complexidade de Kolmogorov, ou até mesmo do ponto de vista criptográfico. Entretanto, nenhum desses pontos de vista conseguem afirmar se algo é aleatório ou não.

Uma possível solução: se o ponto de vista estatístico for o escolhido, então há diversos testes que podem mensurar a qualidade de números aleatórios. Testes de Diehard, desenvolvido em 1995 por George Marsaglia, e os testes de Charmaine Kenny, desenvolido em 2005, são alguns exemplos de conhecidas baterias de testes que realizam essa verificação de qualidade.

6 Referências

Inversive congruential generator:

- Wikipedia
- Khan Academy
- The Mathematical-Function Computation Handbook, Nelson H.F. Beebe
- pLab
- Construction of inversive congruential pseudorandom number generators with maximal period length, Jürgen Eichenauer-Herrmann, 2002
- Parallel inversive congruential generators: Software and field-programmable gate array implementations, Michael Mascagni e Shahram Rahimi, 2001

Lagged Fibonacci generator:

- Lagged Fibonacci Random Number Generators for Distributed Memory Parallel Computers, Srinivas Aluru, 1997
- Aaron Toponce
- Wikipedia

- Bernie Pope
- Oak Ridge National Laboratory

Linear congruential generator:

- Wikipedia
- Eternally Confuzzled
- Rosetta Code

Aleatorieadade:

- Jeremy Kun
- Wikipedia
- Stack overflow