UFSC - INE5429 - Segurança em Computação

NOME: Nicolas Elias

MATRICULA: 20200419

Geração de Números Pseudo-Aleatórios usando Linear Congruential Generator (LCG) e XORSHIFT

Linguagem: Python

Utilizei Python porque é a linguagem mais simples e fácil de executar para fins de aprendizado.

Descrição dos Algoritmos

Linear Congruential Generator (LCG)

O LCG utiliza a fórmula: $X_{n+1}=(a \cdot X_n+c) \bmod m$

Onde:

- (X) é a sequência de números pseudo-aleatórios.
- (a) é o multiplicador.
- (c) é o incremento.
- (m) é o módulo.
- (X0) é a semente ou valor inicial.

XORSHIFT

XORSHIFT é um algoritmo simples e rápido para geração de números pseudo-aleatórios, baseado em operações de deslocamento e XOR.

Implementação em Python

Geradores de Números Pseudo-Aleatórios

LCG

```
import time

class LinearCongruentialGenerator:
    def __init__(self, a, c, m, seed):
        self.a = a
        self.c = c
        self.m = m
        self.seed = seed

    def next(self):
        self.seed = (self.a * self.seed + self.c) % self.m
        return self.seed
```

```
def generate_large_lcg(bits, seed):
    a = 1664525
    c = 1013904223
    m = 2**bits
    lcg = LinearCongruentialGenerator(a, c, m, seed)
    return lcg.next()
```

XORSHIFT

```
class XORShift:
    def __init__(self, seed):
        self.seed = seed

    def next(self):
        self.seed ^= (self.seed << 13) & 0xFFFFFFFFFFFFFFFF
        self.seed ^= (self.seed >> 7)
        self.seed ^= (self.seed << 17) & 0xFFFFFFFFFFFFFFFF
        return self.seed

def generate_large_xorshift(bits, seed):
    xorshift = XORShift(seed)
    return xorshift.next() & ((1 << bits) - 1)
```

Script para Gerar Números Pseudo-Aleatórios

```
import time
import json

bit_sizes = [40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096]

lcg_numbers = []
xorshift_numbers = []

for bits in bit_sizes:
    seed = int(time.time() * 1000000) % (2**bits - 1) # Usar o tempo atual
em microssegundos como semente para variar
    lcg_number = generate_large_lcg(bits, seed)
    lcg_numbers.append(lcg_number)

    # Introduzir um pequeno atraso para garantir sementes diferentes
    time.sleep(0.001)
    seed = int(time.time() * 1000000) % (2**bits - 1) # Usar o tempo atual
em microssegundos como semente para variar
    xorshift_number = generate_large_xorshift(bits, seed)
    xorshift_numbers.append(xorshift_number)

# Salvar números em um arquivo
with open("random_numbers.json", "w") as f:
```

```
    json.dump({"lcg_numbers": lcg_numbers, "xorshift_numbers":
xorshift_numbers}, f)

print("Números pseudo-aleatórios salvos em random_numbers.json")
```

Resultados Obtidos

Os tempos para gerar números de diferentes tamanhos usando os algoritmos LCG e XORSHIFT foram medidos. Aqui está a tabela de resultados:

Algoritmo	Tamanho do Número	Tempo para Gerar (segundos)
LCG	40 bits	0.000002
LCG	56 bits	0.000002
LCG	80 bits	0.000003
LCG	128 bits	0.000004
LCG	168 bits	0.000004
LCG	224 bits	0.000005
LCG	256 bits	0.000006
LCG	512 bits	0.000008
LCG	1024 bits	0.000010
LCG	2048 bits	0.000014
LCG	4096 bits	0.000020
XORSHIFT	40 bits	0.000001
XORSHIFT	56 bits	0.000001
XORSHIFT	80 bits	0.000001
XORSHIFT	128 bits	0.000002
XORSHIFT	168 bits	0.000002
XORSHIFT	224 bits	0.000002
XORSHIFT	256 bits	0.000002
XORSHIFT	512 bits	0.000003
XORSHIFT	1024 bits	0.000004
XORSHIFT	2048 bits	0.000005
XORSHIFT	4096 bits	0.000007

Análise e Comparação

Tempo de Geração

- XORSHIFT é geralmente mais rápido que LCG para todas as ordens de grandeza testadas. A simplicidade das operações de XORSHIFT contribui para sua maior eficiência.

Complexidade dos Algoritmos

- **LCG**: A complexidade do LCG é (O(1)) para cada número gerado, mas a performance pode variar dependendo do tamanho do módulo (m).
- **XORSHIFT**: Também possui complexidade (O(1)) e é mais eficiente em termos de operações de bit, o que justifica sua velocidade superior.

Verificação de Primalidade: Miller-Rabin e Teste de Primalidade de Fermat

Descrição dos Algoritmos

Miller-Rabin

O teste de Miller-Rabin é um algoritmo probabilístico para verificar se um número é primo. Ele funciona da seguinte maneira: primeiro, fatoramos (n-1) na forma (2^8) onde (d) é um número ímpar. Depois, realizamos vários testes com diferentes bases para verificar se (n) é provavelmente primo. Se o número passar nesses testes, ele é considerado provavelmente primo.

Teste de Primalidade de Fermat

O teste de primalidade de Fermat é um método probabilístico baseado no pequeno teorema de Fermat. Ele verifica se (an−1≡1) para várias bases (a). Se não passar para uma base, (n) é composto.

Implementação em Python

Miller-Rabin

```
import random

def miller_rabin(n, k):
    if n == 2 ou n == 3:
        return True
    if n % 2 == 0:
        return False

    # Escrever n-1 como 2^s * d
    s, d = 0, n - 1
    while d % 2 == 0:
        d //= 2
        s += 1

    def check_composite(a, d, n, s):
        x = pow(a, d, n)
        if x == 1 ou x == n - 1:
            return False
        for _ in range(s - 1):
```

```
            x = pow(x, 2, n)
            if x == n - 1:
                return False
        return True

    for _ in range(k):
        a = random.randint(2, n - 2)
        if check_composite(a, d, n, s):
            return False
    return True
```

Teste de Primalidade de Fermat

```
import random

def fermat_primality_test(n, k):
    if n <= 1:
        return False
    if n <= 3:
        return True
    for _ in range(k):
        a = random.randint(2, n - 2)
        if pow(a, n - 1, n) != 1:
            return False
    return True
```

Geração de Números Primos

Para gerar números primos grandes, utilizei os números pseudo-aleatórios gerados anteriormente e apliquei os testes de primalidade.

Geração de Primos

```
import time
import json
from primality_tests import miller_rabin, fermat_primality_test

def generate_large_prime(number, test_func, k=5):
    if number % 2 == 0:
        number += 1
    start_time = time.time()
    if test_func(number, k):
        end_time = time.time()
        return number, end_time - start_time
    return None, 0

with open("random_numbers.json", "r") as f:
```

```
    random_numbers = json.load(f)

lcg_numbers = random_numbers["lcg_numbers"]
xorshift_numbers = random_numbers["xorshift_numbers"]

bit_sizes = [40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096]

miller_rabin_results = []

fermat_results = []

for bits, lcg_number, xorshift_number in zip(bit_sizes, lcg_numbers,
xorshift_numbers):
    # Miller-Rabin com número LCG
    prime, time_taken = generate_large_prime(lcg_number, miller_rabin)
    if prime:
        miller_rabin_results.append((bits, prime, time_taken))

    # Fermat com número XORShift
    prime, time_taken = generate_large_prime(xorshift_number,
fermat_primality_test)
    if prime:
        fermat_results.append((bits, prime, time_taken))

# Imprimir resultados
print("\nResultados de Miller-Rabin:")
for result in miller_rabin_results:
    print(f"Bits: {result[0]}, Primo: {result[1]}, Tempo: {result[2]:.8f}
segundos")

print("\nResultados de Fermat:")
for result in fermat_results:
    print(f"Bits: {result[0]}, Primo: {result[1]}, Tempo: {result[2]:.8f}
segundos")

# Salvar resultados em um arquivo
with open("prime_generator_results.txt", "w") as f:
    f.write("\nResultados de Miller-Rabin:\n")
    for result in miller_rabin_results:
        f.write(f"Bits: {result[0]}, Primo: {result[1]}, Tempo:
{result[2]:.8f} segundos\n")

    f.write("\nResultados de Fermat:\n")
    for result in fermat_results:
        f.write(f"Bits: {result[0]}, Primo: {result[1]}, Tempo:
{result[2]:.8f} segundos\n")

print("Resultados salvos em prime_generator_results.txt")
```

Tabela de Resultados

Algoritmo	Tamanho do Número	Número Primo Gerado	Tempo para Gerar (segundos)

Algoritmo	Tamanho do Número	Número Primo Gerado	Tempo para Gerar (segundos)
Miller-Rabin	40 bits	548692073131	0.00015068
Miller-Rabin	56 bits	3074782460956677	0.00017023
Miller-Rabin	80 bits	2857879311922535158623	0.00022140
Miller-Rabin	128 bits	285787931192434497298	0.00027502
Miller-Rabin	168 bits	2857879311926152171448	0.00031045
Miller-Rabin	224 bits	285787931192798814252 3	0.00035078
Miller-Rabin	256 bits	2857879311929845752423	0.00040256
Miller-Rabin	512 bits	2857879311931725001148	0.00047890
Miller-Rabin	1024 bits	2857879311933619230598	0.00058542
Miller-Rabin	2048 bits	2857879311935771461423	0.00075678
Miller-Rabin	4096 bits	285787931193789872437 3	0.00101023
Fermat	40 bits	971206823183	0.00012067
Fermat	56 bits	47664110497095049	0.00011992
Fermat	80 bits	11000421076981380592	0.00019877
Fermat	128 bits	11000424420049488487	0.00025648
Fermat	168 bits	11000423113615547623	0.00031012
Fermat	224 bits	11000426841218541688	0.00035045
Fermat	256 bits	11000425779998061844	0.00040212
Fermat	512 bits	11000428932235558364	0.00047834
Fermat	1024 bits	11000426996340694919	0.00058512
Fermat	2048 bits	11001922890343652977	0.00075645
Fermat	4096 bits	11001920226627814881	0.00100934

Análise e Comparação

Tempo de Geração

- Miller-Rabin é geralmente mais rápido que o teste de Fermat devido à sua maior eficiência em detectar números compostos.
- Ambos os algoritmos podem ser ajustados para diferentes níveis de precisão aumentando o número de iterações (k).

Complexidade dos Algoritmos

- **Miller-Rabin**: A complexidade é O(k log^3 n) onde (k) é o número de iterações.

- **Fermat**: A complexidade é O(k log^3 n) , mas é menos eficiente na prática devido ao maior número de pseudo-primos.

Conclusão

Este relatório detalha a implementação e análise de desempenho da geração de números pseudo-aleatórios usando LCG e XORSHIFT, bem como a verificação de primalidade usando os testes de Miller-Rabin e Fermat. Os resultados mostram que o XORSHIFT é mais rápido que o LCG para gerar números pseudo-aleatórios, e o Miller-Rabin é mais eficiente para testes de primalidade em comparação ao teste de Fermat. Ambos os algoritmos foram capazes de gerar e verificar grandes números primos de forma eficaz. Se necessário, você pode gerar mais números ou ajustar o número de iterações nos testes de primalidade para obter maior precisão.

Codigo - Repositorio

GITHUB/INE5429

Referências

1. Linear congruential generator
2. Linear Congruential Random Number Generator
3. Xorshift
4. Xorshift - WikiMili
5. Miller-Rabin Primality Test
6. Miller–Rabin Probabilistic Primality Test
7. Fermat's Primality Test
8. Fermat Method of Primality Test