Números aleatórios e primalidade

Vinicius Macelai

September 2019

1 Números aleatórios

Foram escolhidos os algoritmos Linear congruential generator e Xorshift devido a sua facilidade de implementação e consequentemente são os mais populares.

1.1 Linear congruential generator

```
def lcg(m: int, seed: int = None) -> int:
    a = 3917736999557981949897212880467659694119851
    1312402174107900118507767888787985319889078 49876
    8125726538781142699985438519849879849879867 91338
    7985078149715117956575598705232859627482107 67876
    5877225323948753141
    c = 0
    num = seed or 1
    while True:
        num = (a * num + c) % m
        yield num
```

É um algoritmo que produz números pseudo-aleatórios calculados em uma função linear em trecho. O gerador é definido pela seguinte fórmula:

$$X_{(n+1)} = (aX_n + c)mod_m$$

O benefício do LCG é que, com a escolha apropriada dos parâmetros, o período é conhecido e longo. Foi escolhido usar m como potência de 2 e c=*0*. Escolher m como potência de 2, produz um LCG particularmente eficiente, porque isso permite que a operação do módulo seja calculada simplesmente truncando a representação binária. De fato, os bits mais significativos geralmente não são computados. Existem, no entanto, desvantagen.

Desta forma há um período máximo $m/4$, alcançado se 3 ou 5 (mod 8). O estado inicial X_n deve ser ímpar e os três bits baixos de X se alternam entre dois estados e não são úteis[1]. Utilizando dessa congruência que cheguei no valor calculado a para ser grande o suficiente para um período de 4096.

Conclui-se que é um gerador de números pseudo-aleatórios não suficiente para geração de chaves ou casos mais complexos, funciona bem para números pequenos de 32 a 64 bits, mas ao aumentar esse período há diversas falhas.

1.2 Xorshift

```python
def xorshift32(seed: int, n: int) -> int:
    xor_parts = list()

    while n != 0:
        x = seed & 0x7fffffff  # 32 bits
        x ^= x << 13
        x ^= x >> 17
        x ^= x << 5
        x = x & 0x7fffffff  # 32 bits novamente
        xor_parts.append(x)

        n >>= 32

    x = 0
    for i, part in enumerate(xor_parts):
        x |= part << (32 * i)

    return x
```

É um algoritmo que produz números pseudo-aleatórios calculados a partir do deslocamento de registradores que são realimentados em uma função linear(LFSRs) que permitem uma implementação particulamente eficiente sem o uso de polinômios. Como todos os LFSRs, os parâmetros devem ser escolhidos com cuidado para alcançar um período longo.

Encontrei na literatura paramêtros para o algoritmo para números de até 128 bits, porém, optou-se nessa implementação de se utilizar o Xorshift padrão de 32 bits e realizando diversas rodadas para se obter o número de tamanho desejado, isso revela um grande problema para uso em determinados casos. Pois a entropia do algoritmo é de apenas 32 bits e não por exemplo 4096.

A conclusão é a mesma para o LCG.

1.3 Comparação de resultados

Foram realizados 100000 chamadas a função e retirado a sua média aritmética.

Bits	LCG	Xorshift
40	5.32e-07	8.09e-07
56	5.37e-07	9.06e-07
80	5.43e-07	7.99e-07
128	5.36e-07	8.57e-07
168	5.72e-07	8.16e-07
224	5.55e-07	7.98e-07
256	5.72e-07	8.17e-07
512	6.54e-07	8.11e-07
1024	8.61e-07	7.93e-07
2048	1.56e-06	8.10e-07
4096	4.03e-06	8.65e-07

2 Números primos

2.1 Miller-rabin

```python
def miller_rabin(n: int, k: int) -> bool:
    if n == 2:
        return True

    if n % 2 == 0:
        return False

    r, s = 0, n - 1
    while s % 2 == 0:
        r += 1
        s //= 2
    for _ in range(k):
        a = random.randrange(2, n - 1)
        x = pow(a, s, n)
        if x == 1 or x == n - 1:
            continue
        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False
    return True
```

2.2 Fermart

```python
def fermat_little_theorem(n: int) -> bool:
    return (2 << (n - 2)) % n == 1
```

P é um número primo se para qualuqer a o número $a^p - a$ é inteiro múltiplo de p:

$$a^p \equiv a(mod_p)$$

A escolha do Teorema de Fermat foi pela sua simplicidade e facilidade de implementação, além do mais, o teste de primalidade de Miller-Rabin usa uma extensão do teorama de Fermat.

O teste prova que um número não é primo ou afirma que ele é primo com uma probabilidade de erro que pode ser escolhida tão baixa quanto desejado. O teste é muito simples de implementar e computacionalmente mais eficiente do que todos os testes determinísticos conhecidos. Portanto, geralmente é usado antes de iniciar uma prova de primalidade.

2.3 Comparação de resultados

Foram realizadas apenas uma chamada devido a demora para os números maiores.

Bits	Segundos
40	0.000442
56	0.000592
80	0.001174
128	0.004918
168	0.010128
224	0.01733
256	0.0499
512	0.4354
1024	3.06
2048	31.90
4096	479.14

References

[1] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.