Segurança em Computação
Trabalho Individual II

Adriano Tosetto - 15104099

16 de abril de 2019

1 Introdução

O trabalho foi feito em Python com a única justificativa de ser tremendamente mais fácil lidar com números gigantes nesta linguagem. Os algoritmos de geração de números aleatórios implementados foram: **Lagged Fibonacci Generator**, **Linear Congruential Generator** e **Multiply With Carry**. Para a verificação de números aleatórios foram utilizados os algoritmos: **Miller-Rabin** e **Fermat**.

O fonte está aqui

2 Geração de números aleatórios

2.1 Números pseudorandômicos

Dados			
Algoritmo	Bits	Tempo para geração(em segundos)	Número (apenas dimensão do número gerado)
Lagged Fibonacci Generator	40	5.7220458984375e-06	4.98E+11
Lagged Fibonacci Generator	56	5.960464475390625e-06	1.03E+16
Lagged Fibonacci Generator	80	4.0531158447265625e-06	4.77E+23
Lagged Fibonacci Generator	128	7.62939453125e-06	1.10E+38
Lagged Fibonacci Generator	168	6.9141387939453125e-06	2.39E+50
Lagged Fibonacci Generator	224	7.3909759521484375e-06	8.76E+66
Lagged Fibonacci Generator	256	1.0251998901367188e-05	6.82E+76
Lagged Fibonacci Generator	512	1.4543533325195312e-05	6.59E+153
Lagged Fibonacci Generator	1024	8.106231689453125e-06	1.51E+308
Lagged Fibonacci Generator	2048	9.059906005859375e-06	1.97E+616
Lagged Fibonacci Generator	4096	9.5367431640625e-06	3.84E+1232
Linear Congruential Generator	40	6.4373016357421875e-06	9.97E+11
Linear Congruential Generator	56	6.9141387939453125e-06	6.81E+16
Linear Congruential Generator	80	5.960464475390625e-06	1.03E+24
Linear Congruential Generator	128	8.344650268554688e-06	9.33E+37
Linear Congruential Generator	168	1.049041748046875e-05	8.69E+49
Linear Congruential Generator	224	1.0013580322265625e-05	9.77E+66
Linear Congruential Generator	256	1.5497207641601562e-05	1.12E+77
Linear Congruential Generator	512	9.775161743164062e-06	3.27E+153
Linear Congruential Generator	1024	1.5735626220703125e-05	1.42E+308
Linear Congruential Generator	2048	1.430511474609375e-05	2.16E+616
Linear Congruential Generator	4096	1.2874603271484375e-05	3.61E+1232
Multiply With Carry	40	5.245208740234375e-06	5.81E+11
Multiply With Carry	56	5.4836273193359375e-06	3.81E+16
Multiply With Carry	80	4.76837158203125e-06	7.87E+23
Multiply With Carry	128	6.67572021484375e-06	2.34E+38
Multiply With Carry	224	6.198883056640625e-06	1.94E+67
Multiply With Carry	256	1.0251998901367188e-05	6.82E+76
Multiply With Carry	512	1.2636184692382812e-05	8.81E+153
Multiply With Carry	1024	7.867813110351562e-06	1.19E+308
Multiply With Carry	2048	9.059906005859375e-06	1.64E+616
Multiply With Carry	4096	3.600120544433594e-05	5.73E+1232

2.2 Complexidade dos Algoritmos RNG

2.3 Linear Congruential Generator

```
class LinearCongruentialGenerator:
    def __init__(self, seed=None, bits=40, a=1664525, c=1013904223):
        lower_bound = upper_bound = 0
        for i in range(0, bits):
            upper_bound = upper_bound + pow(2, i)
        if seed is None:
            self._seed = int(time.time())
        self._state = int(time.time()) if seed is None else seed
        self._a = randint(lower_bound, upper_bound)
        self._c = randint(lower_bound, upper_bound)
        self._m = 2**(bits+1)
        self._bits = bits
    def next(self):
        _next = (self._a*self._state + self._c) % self._m
        if (_next.bit_length() > self._bits):
            diff_bits = (_next.bit_length() - self._bits)
            _next = _next >> diff_bits
        self._state = _next
        return self._state
```

Como mostrado no código, a complexidade para gerar um número aleatório é O(1), visto que é apenas realizar uma única vez cálculos de soma, multiplicação e módulo.

2.4 Multiply With Carry

```
class MultiplyWithCarry:
    def __init__(self, bits):
        self._r = 0xFFFFFFFE
        self._CMWC_CYCLE = 4096
        self._CMWC_C_MAX = 809430660
        self._i = self._CMWC_CYCLE
        self._Q = [0]*4096
        self._c = randint(2, 809430660)
        self._bits = bits

        for i in range(self._CMWC_CYCLE):
            self._Q[i] = randint(2, int(2**(bits-1)))
    def next(self):
        a = 18782
        self._i = (self._i+1) & 4095
        t = a * (self._Q[self._i]) + self._c
        self._c = t >> 32
        x = t + self._c
        if self._c > x:
            x = x+1
            self._c = self._c+1
        aux = (x - self._r)
        if (aux.bit_length() > self._bits):
            diff_bits = (aux.bit_length() - self._bits)
            aux = aux >> diff_bits
        self._Q[self._i] = aux
```

```
        return self._Q[self._i]
```

Como o outro algoritmo, o "gargalo do algoritmo"está no preenchimento do vetor $\_Q$ que é usado para calcular o próximo número. Para o cálculo do próximo número não há cálculos depedendo de alguma entrada. Se a variável **self._CMWC_CYCLE** fosse uma entrada do algoritmo, a complexidade seria *O(self._CMWC_CYCLE)*, pois ela é que determina o tamanho do vetor a ser preenchido e a responsável pelo único for do algoritmo.

2.5 LaggedFibonacciGenerator

```
class LaggedFibonacciGenerator:
    def __init__(self, seed = None, j = 3, k = 7, bits = 32):
        self._seed = [0] * k
        if seed is None:
            upper_bound = 0
            lower_bound = 2**(bits-1)
            for i in range(0, bits):
                upper_bound = upper_bound + pow(2, i)
            for i in range(0, k):
                self._seed[i] = randint(lower_bound, upper_bound)
        else:
            self._seed = seed
        self._m = 2**bits
        self._j = j - 1
        self._k = k - 1
        self._bits = bits
        if (len(self._seed) < k):
            print("error, len _seed shoud be, at least, the value of k")
    def next(self):
        _next = (self._seed[self._j] + self._seed[self._k]) % self._m
        if (_next.bit_length() > self._bits):
            diff_bits = (_next.bit_length() - self._bits)
            _next = _next >> diff_bits
        self._seed = self._seed[1:7] + [_next]
        return _next
```

O Lagged Fibonacci Generator é O(k), onde k é o tamanho da lista usada para fazer a soma dos elementos que gerarão o próximo número. Como visto no código, é necessário recopiar essa lista a cada número gerado.

Complexidade	
Algoritmo	Complexidade
Lagged Fibonacci Generator	(k)
Linear Congruential Generator	(1)
Multiply With Carry	(1)

2.6 Comparação entre os algoritmos

O algoritmo Linear Congruential Generator é o único dos três que não possui uma "memória", ou seja, um vetor para calcular o próximo número. Ele possui apenas um estado. O Linear Congruential Generator e o Lagged Fibonacci Generator confiam em operações de multiplicação, adição e módulo para gerar o próximo número randômico. Além disso, eles têm uma "memória"em que se beseiam para calcular o próximo número. Comparando a velocidade de geração de números, o Multiply With Carry ganha, isso pode se dever ao fato que ele faz menos operações e boa parte dessas operações são de bitwise, que são muito mais rápidas que adição/multiplicação/módulo.

3 Números primos

3.1 Justificativa da escolha

O teste de fermat foi o segundo algoritmo escolhido pois era o primeiro da lista de sugestões e funcionou bem.

3.2 Números primos gerados

Foram gerados números primos para o algoritmo Multiply With Carry, agged Fibonacci Generator e inear Congruential Generator. O output dos três métodos está no arquivo **primos_gerados.txt** junto com os outputs gerados pelos outros dois algoritmos. Não inclui na tabela os números primos gerados pelo Multiply With Carry, no entanto, como já dito, eles se encontram no txt do repositório. A tabela no final mostra os primos gerados.

3.3 Dificuldades encontradas

Não foram encontradas dificuldades muito grandes, além da espera para encontrar números primos com muitos bits. Algumas tentativas geravam números primos até uma certa quantidade de bits, depois disso havia uma demora muito grande. Executei diversas vezes, em algumas delas, os algoritmos pareciam acertar o passo para gerar primos mais rapidamento. Quase todas as dificuldades que poderiam acontecer foram sanadas pelo próprio Python, que possui um suporte para BigInt poderoso, além de funções como *pow(a, b, c)* (exponenciação modular) que rodam muito rapidamente.

3.4 Complexidade dos Algoritmos

3.4.1 Implementação

```
def miller_rabbin(n, k):
    if (n % 2 == 0):
        return False
    t = n - 1
    e = 0
    while(t % 2 == 0):
        e = e + 1
        t = t // 2
    m = (n-1) // (2**e)
    #print("t == " + str(t))
    #print("m == " + str(m))
    #print("e == " + str(e))

    for _ in range(k):
        lgc = LinearCongruentialGenerator(bits=n.bit_length()-1)
        a = 0
        if PYTHON_RAND:
            a = randint(2, n - 1)
        else:
            a = lgc.next()
        if a >= n:
            raise("a nao pode ser igual ou maior que o suposto primo N")
        x = pow(a, m, n)
        if x == 1 or x == n - 1:
            continue
        for _ in range(e-1):
            x = pow(x, 2, n)
```

```
            if x == 1:
                return False
            if x == n - 1:
                break
        else:
            return False
    return True

def fermat_primality_test(p, k):
    lgc = LinearCongruentialGenerator(bits=p.bit_length()-1)
    if p == 2:
        return True
    if p % 2 == 0:
        return False
    for _ in range(k):
        a = 0
        if PYTHON_RAND:
            a = randint(2, p - 1)
        else:
            a = lgc.next()
        if a >= p:
            raise("a nao pode ser igual ou maior que o suposto primo P")
        if pow(a, p - 1, p) != 1:
            return False
    return True
```

3.4.2 Miller Rabbin & Fermat Primality Test

Ambos dependem da entrada k do usuário (número de testemunhas para verificar se um número P é primo) para gerar as testemunhas. O miller-rabin tem duas partes dentro do for que gera as testemunhas: a parte de fazer exponenciação: pow(a, d, n), logo a complexidade dessa parte k * O(pow). Logo abaixo existe um loop até e - 1 vezes onde e é da seguinte fórmula:

$$n - 1 = m * 2^e \tag{1}$$

assim, esse loop vai executar **e-1** vezes e **e**. O log de **n-1** é:

$$log((2^e) * m) = log(2^e) + log(m) \tag{2}$$

desconsiderando o **log(m)**, e - 1 pode ser aproximado por **log(n)**. Assim, a complexidade para Miller-Rabin fica:

$$k * (O(pow) + log(n) * O(pow)) \tag{3}$$

Onde **O(pow)** é a complexidade da função pow(a,b,c) do Python. Não consegui achar nenhuma fonte que falasse de fato qual era a complexidade de pow(a,b,c) do python. O código de **pow()** do Python é complexo e não consegui calcular o Big-O dele.
Para o test de fermat, a análise é semelhante: São k testemunhas geradas onde é preciso chamar a função **pow** k vezes. Assim, a complexidade para Fermat é:

$$k * (O(pow) + log(n) * O(pow)) \tag{4}$$

Onde pow é a função pow(a,b,c) do Python. Em ambos os algoritmos, a complexidade para a geração de forma aleatória das testemunhas foi desconsiderada visto que para o **randint** do Python e **LaggedFibonacciGenerator** que eu fiz são O(1)

Complexidade	
Algoritmo	Complexidade
Miller Rabin	$k * (O(pow) + log(n) * O(pow))$ (5)
Fermat Primality Test	k * (O(pow) + log(n) * O(pow))

3.5 Tempo necessário para gerar os números primos para cada método

Está presente na tabela apresentada no fim do PDF.

3.6 Números pseudorandômicos

LFG = Lagged Fibonacci Generator
LCG = Linear Congruential Generator

Dados				
Algoritmo de geração	Algoritmo de Verificação	Bits	Tempo para geração (em segundos)	Tempo para testar primalidade (em segundos)
LFG	Miller-Rabin	40	0.0003616809844	0.0002799034118
LFG	Fermat	40	0.00019049644470	0.00012612342834
LFG	Miller-Rabin	56	0.0033750534057	0.0003781318664
LFG	Fermat	56	0.0005850791931	0.00015306472778
LFG	Miller-Rabin	80	0.0009388923645	.0005881786346
LFG	Fermat	80	0.0021889209747	0.0003237724304
LFG	Miller-Rabin	128	0.006439208	0.0012166500091
LFG	Fermat	128	0.013254404067	0.00110101699829
LFG	Miler-Rabin	168	0.01657557487	0.0.0018815994262
LFG	Fermat	168	0.004686832427	0.001121520996
LFG	Miller-Rabin	224	0.05834627151	0.0027666091918
LFG	Fermat	224	0.006192922592	0.0019910335540
LFG	Miller-Rabin	256	0.025435924530	0.003098726272
LFG	Fermat	256	0.012595176696	0.002302885055
LFG	Miller-Rabin	512	0.232330322	0.011030912399
LFG	Fermat	512	0.15907716751	0.010582685470
LFG	Miller-Rabin	1024	0.5817663669	0.05894970893
LFG	Fermat	1024	0.45347094535	0.053189992904
LFG	Miller-Rabin	2048	20.75140690803528	0.6315298080444336
LFG	Fermat	2048	21.9278724	0.5591094493
LFG	Miller-Rabin	4096	488.83546471	4.116994142
LFG	Fermat	4096	274.5899832248688	4.159802436828613
LCG	Miller-Rabin	40	0.00143337249755	0.0003037452697
LCG	Fermat	40	0.00017309188842	0.0001323223114
LCG	Miller-Rabin	56	0.00087332725524	0.0004069805145
LCG	Fermat	56	0.00022006034851	0.0001630783081
LCG	Miller-Rabin	80	0.001957416534	0.0006330013275
LCG	Fermat	80	0.0006330013275	0.00029659271240
LCG	Miller-Rabin	128	0.011278629302	0.001222848892
LCG	Fermat	128	0.018824338912	0.0006694793701
LCG	Miler-Rabin	168	0.0043039321899	0.0019781589508
LCG	Fermat	168	0.001088857650	0.0009996891021
LCG	Miller-Rabin	224	0.045264244079	0.0036072731018
LCG	Fermat	224	0.0519692897796	0.0019066333770
LCG	Miller-Rabin	256	0.0639791488	0.0033431053161
LCG	Fermat	256	0.02133464813	0.002092123031
LCG	Miller-Rabin	512	0.16827654838	0.013980388641
LCG	Fermat	512	0.1833810806	0.010114431381
LCG	Miller-Rabin	1024	3.3544223308	0.06139492988
LCG	Fermat	1024	2.102313756942749	0.0670781135559082
LCG	Miller-Rabin	2048	3.907580852	0.386580228805542
LCG	Fermat	2048	8.401597261428833	0.3506994247436
LCG	Miller-Rabin	4096	845.8991656303406	2.64555692672
LCG	Fermat	4096	51.649994134	2.5893337726

4 Primos de Mersenne

Foram testados alguns primos conhecidos de Mersenne. Segue o resultado:

Dados		
Algoritmo de Verificação	Primo de Mersenne	Tempo para testar primalidade (em segundos)
Miller-Rabin	$2^{89}) - 1$	0.0008168220520019531
Teste de fermat	$(2^{89} - 1)$	0.00045561790466308594
Teste miller-rabin	$2^{107} - 1$	0.0011477470397949219
Teste de fermat	$2^{107} - 1$	0.0006520748138427734
Teste miller-rabin	$2^{127} - 1$	0.0014646053314208984
Teste de fermat	$2^{127} - 1$	0.0009024143218994141
Teste miller-rabin	$2^{521} - 1$	0.012618064880371094
Teste de fermat	$2^{521} - 1$	0.01132345199584961
Teste miller-rabin	$2^{607} - 1$	0.01987743377685547
Teste de fermat	$2^{607} - 1$	0.014545917510986328
Teste miller-rabin	$2^{1279} - 1$	0.10896921157836914
Teste de fermat	$2^{1279} - 1$	0.09621596336364746
Teste miller-rabin	$2^{9941} - 1$	32.62743091583252
Teste de fermat	$2^{9941} - 1$	31.419555187225342
Teste miller-rabin	$2^{21701} - 1$	309.1169214248657
Teste de fermat	$2^{21701} - 1$	301.7051751613617
Teste miller-rabin	$2^{110503} - 1$	Só Deus sabe
Teste de fermat	$2^{110503} - 1$	Só Deus sabe

5 Referências

https://medium.com/asecuritysite-when-bob-met-alice/for-the-love-of-computing-the-lagged-fibonacci-generator-where-nature-meet-random-numbers-f9fb5bd6c237
https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test
https://en.wikipedia.org/wiki/Fermat_primality_test https://en.wikipedia.org/wiki/Multiply-with-carry_pseudorandom_num
https://pt.wikipedia.org/wiki/Geradores_congruentes_lineares