INE5429 - Segurança em computação

Professores: Jean Everson Martina e Ricardo Felipe Custódio

Aluno: Anthony Bernardo Kamers | 25 de novembro de 2021

Pseudo-Random Number Generators e identificadores de números primos

O trabalho foi inteiramente construído usando a linguagem de programação python (versão 3.8.10). Os resultados obtidos foram testados várias vezes e com diversas "seeds", mas no final, foi escolhido uma seed fixa (para geração de mesmo número todas as vezes, a fim de analisar os resultados). Além disso, todas as informações referentes a cada algoritmo estão comentados no código do mesmo (assim como suas respectivas referências).

1. Números aleatórios

Para geração de números pseudo-aleatórios, foram escolhidos os algoritmos de Lagged Fibonacci Generator e de Blum-Blum-Shub (BBS). Vale considerar que, para cada algoritmo, foi feito um método para gerar um número com o algoritmo com um determinado número de bits (visto que o trabalho exigia isso em algumas etapas). Tendo isso em vista, alguns trechos do código foram alterados de seu algoritmo original, para que se pudesse obter um número com determinado tamanho com maior facilidade.

Os algoritmos foram testados para gerarem números com tamanhos de 40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048 e 4096 bits.

Lagged Fibonacci Generator

Com conceitos relacionados ao algoritmo clássico recursivo de Fibonacci, foi fácil fazer uma implementação desse algoritmo usando as informações obtidas com o pseudo-código no Wikipedia.

A implementação pode ser vista abaixo:

```
#############################################################
# Fibonacci é a sequência gerada por
# Sn = S(n-1) + S(n-2)
# Generalizando para gerar números aleatórios,
# podemos gerar (com offsets j e k) de forma cíclica
# (com mod m), pela seguinte fórmula
# Sn = S(n-j) ★ S(n-k) (mod m), 0 < j < k
# onde o operador ★ pode ser adição, subtração,
# multiplicação ou XOR binário
# fontes:
    # https://en.wikipedia.org/wiki/Lagged_Fibonacci_generator
    # https://medium.com/asecuritysite-when-bob-met-alice/for-the-love-of-
computing-the-lagged-fibonacci-generator-where-nature-meet-random-numbers-
f9fb5bd6c237
#############################################################
```

```python
class FibonacciGenerator:
    # aqui vamos usar uma determinada seed,
    # tal como j e k fixos em 3 e 7, respectivamente
    # os valores podem ser mudados na instanciação
    # para fornecer maior randomicidade
    def __init__(self, seed = 8675319, j = 3, k = 7, sizeBit = 32):
        self.seed = toList(seed)
        self.j = j
        self.k = k
        self.m = 2 ** sizeBit # para fazer números de n tamanho, vamos fazer módulo de 2^quantidade de bits
        self.sizeBit = sizeBit # guardar tamanho de bits que quer fazer o número aleatório

    def next(self):
        num1 = self.seed[self.j - 1] # pega número da posição j-1
        num2 = self.seed[self.k - 1] # pega número da posição k-1

        # usar operador como sendo multiplicação (*)
        numGenerated = (num1*num2) % self.m # faz a conta de fibonacci usando operador * e módulo m
        self.seed.append(numGenerated) # adiciona o número gerado na última posição (para continuar sequência Fib)
        self.seed.pop(0) # remove o primeiro número, para continuar randomicidade

        return numGenerated

    # gera números aleatórios até alcançar a quantidade de bits solicitada
    def generateNumOfBits(self):
        while True:
            randomNumber = self.next()
            if (len(bin(randomNumber)[2::])) == self.sizeBit:
                return randomNumber
                break

# função auxiliar para transformar número em lista de números
def toList(number):
    return list(map(int, str(number)))
```

Com as referências, assim como análise do resultado do código, comprovou-se que a complexidade do algoritmo é linear (maior do que metade do seu período (escolhido)). Tal complexidade depende também do operador usado para fazer a implementação (no caso aqui foi usada a multiplicação).

Blum-Blum-Shub

Este algoritmo feito em 1968 precisa de alguns requisitos para a implementação do mesmo, como dois valores "p" e "q" sendo primos. Como precisamos gerar um número com determinado números de bits, foi feita uma implementação utilizando "p" e "q" sendo ímpares para teste da matemática envolvida. Depois, com auxílio do artigo da Medium (antes da implementação do algoritmo próprio), foram gerados números de n bits (especificados na instanciação do mesmo). Com isso, temos garantias de que será gerado, no mínimo, um número com n bits especificados. Segue a implementação do algoritmo:

```
#############################################################
# Fazer um PRNG usando o altoritmo de Blum Blum Shub (BBS)
# Proposto em 1968, usa o a função de um-caminho de Michael Rabin
# da função de primalidade Miller-Rabin. A fórmula do algoritmo é:
# X(n+1) = X(n)^2 mod M
    # X(0) = seed (co-primo de M)
    # M = p * q
        # p e q são primos congruentes a 3 (mod 4)
# fontes:
    # https://en.wikipedia.org/wiki/Blum_Blum_Shub
    # https://asecuritysite.com/encryption/blum
    #
https://github.com/VSpike/BBS/tree/9be96e30acd072db61ed2c05ba4c1a5044ea554e
#############################################################

from .RandomGenerator import generate_prime_number

class BBSGenerator:
    def __init__(self, seed = 3, sizeBit = 2048, sizeBit1 = None):
        # como precisamos de 2 números primos para serem multiplicados
        # entre si (pelo algoritmo) e precisamos também que seja gerado um
número
        # com determinado número de bits, vamos gerar, com a ajuda
        # do código retirado de https://medium.com/@prudywsh/how-to-
generate-big-prime-numbers-miller-rabin-49e6e6af32fb,
        # somente para ter parâmetros necessários para o funcionamento
completo do algoritmo.
        # Dessa maneira, quando forem multiplicados,
        # com certeza gerarão um número maior que sizeBit bits
        self.p = generate_prime_number(sizeBit1 if sizeBit1 is not None
else sizeBit)
        self.q = generate_prime_number(sizeBit1 if sizeBit1 is not None
else sizeBit)
        self.m = self.p * self.q # multiplicação de p por q
        self.seed = seed
        self.sizeBit = sizeBit

    def next(self):
        self.seed = (self.seed ** 2) % self.m
        return self.seed

    # gera números aleatórios até alcançar a quantidade de bits solicitada
    def generateNumOfBits(self):
        while True:
            # para forçar ser do tamanho de bits que queremos,
            # vamos fazer o número gerado mod 2^self.sizeBit
            randomNumber = self.next() % (2 ** self.sizeBit)

            if (len(bin(randomNumber)[2::])) == self.sizeBit:
```

```
                return randomNumber
                break
```

Devido à geração de um número primo aleatório de n bits, para usar de base para "p" e "q", a instanciação do algoritmo é bem demorada (quando comparado com outros algoritmos). Como isso é arbitrário e não é relacionado à geração do número aleatório em si, a instanciação dos algoritmos não foram usados para o cômputo do processamento das informações (foi usado somente o método `generateNumOfBits()`). Com o código acima, pode-se inferir que a complexidade do código é O(n^2).

Resultados obtidos dos algoritmos

Com os dois algoritmos utilizados, foi possível obter números com tamanho específico de bits. Foi usado então um algoritmo usando as duas implementações e calculando o tempo de processamento usando um "iterator" do Python. Segue código e tabela de dados identificados para análise:

```
import time
from PRNG.FibonacciGenerator import FibonacciGenerator
from PRNG.BBSGenerator import BBSGenerator

# calcular o tempo de uma função
# https://stackoverflow.com/questions/1557571/how-do-i-get-time-of-a-python-programs-execution
def profile(fct):
  def wrapper(*args, **kw):
    start_time = time.time()
    ret = fct(*args, **kw)
    print("{} {} {} return {} in {} seconds".format(args[0].__class__.__name__,
                                                     args[0].__class__.__module__,
                                                     fct.__name__,
                                                     ret,
                                                     time.time() - start_time))
    return ret
  return wrapper

@profile
def calcularProcessamento(generator, tamanho):
    print(tamanho)
    print(generator.generateNumOfBits())

def main():
    tamanhoBits = [40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096]

    # para cada tamanho em bits, rodar os algoritmos geradores de números
    # pseudo-aleatórios e calcular o tempo de processamento para fazer
    # a planilha com comparações de resultado
    for tamanho in tamanhoBits:
        fib = FibonacciGenerator(sizeBit=tamanho)
```

```
        calcularProcessamento(fib, tamanho)

    for tamanho in tamanhoBits:
        # fazer dessa forma, pois o gerador de primos com
        # 4096 bits é muito demorado ou não alcança um resultado
        # satisfatório e, como ao usar 2 de 2048 bits, pode-se fazer
        # um número com 4096 bits, vamos fazer dessa forma
        if tamanho == 4096:
            bbs = BBSGenerator(sizeBit=tamanho, sizeBit1=tamanho/2)
        else:
            bbs = BBSGenerator(sizeBit=tamanho)
        calcularProcessamento(bbs, tamanho)

if __name__ == "__main__":
    main()
```

Algoritmo	**Tamanho (bits)**	**Tempo processamento (s)**
Lagged Fibonacci	40	5,44E-05
Blum Blum Shub	40	6,60E-05
Lagged Fibonacci	56	2,79E-05
Blum Blum Shub	56	4,22E-05
Lagged Fibonacci	80	2,00E-05
Blum Blum Shub	80	3,48E-05
Lagged Fibonacci	128	2,29E-05
Blum Blum Shub	128	4,15E-05
Lagged Fibonacci	168	2,53E-05
Blum Blum Shub	168	5,29E-05
Lagged Fibonacci	224	2,38E-05
Blum Blum Shub	224	7,06E-05
Lagged Fibonacci	256	3,86E-05
Blum Blum Shub	256	5,17E-05
Lagged Fibonacci	512	4,36E-05
Blum Blum Shub	512	5,84E-05
Lagged Fibonacci	1024	7,51E-05
Blum Blum Shub	1024	0,000109910964966
Lagged Fibonacci	2048	0,000208377838135
Blum Blum Shub	2048	0,000115394592285

Algoritmo	Tamanho (bits)	Tempo processamento (s)
Lagged Fibonacci	4096	0,000154495239258
Blum Blum Shub	4096	0,000690698623657

Pode-se observar que os algoritmos são muito parecidos para geração de números pseudo-aleatórios, entretanto, o algoritmo de Lagged Fibonacci Generator é melhor para a grande maioria dos casos.

2. Números primos

Para fazer a verificação se um determinado é primo ou não, foram usados os métodos de Miller-Rabin (obrigatória a implementação) e de Solovay-Strassen (escolhido por ter melhor performance comparado com os outros).

Miller-Rabin

Baseando-se no pseudo-código dispobilizado na Wikipedia, podemos obter o seguinte código:

```
#############################################################
# Teste de primalidade de Miller-Rabin
# Teste probabilístico em tempo polinomial
# Segundo o wikipedia, o algoritmo (pseudo-código) é o seguinte:
'''
write n as 2r·d + 1 with d odd (by factoring out powers of 2 from n − 1)
WitnessLoop: repeat k times:
    pick a random integer a in the range [2, n − 2]
    x ← a^d mod n
    if x = 1 or x = n − 1 then
        continue WitnessLoop
    repeat r − 1 times:
        x ← x^2 mod n
        if x = n − 1 then
            continue WitnessLoop
    return “composite”
return “probably prime”
'''
# Vamos fazer algumas adaptações, para que fique mais simples e mais
"pythônico"
# Fontes:
    # https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test
    # https://www.geeksforgeeks.org/primality-test-set-3-miller-rabin/
#############################################################

import random # para pegar números aleatórios dentro do "range" de números

# n = valor de teste de primalidade
# k = parâmetro que determina a precisão do teste
def MillerRabin(n, k):
    # se n for <= 3, é fixamente primo
    if n <= 3:
        return True
```

```
    # qualquer múltiplo de 2 (par), não é primo por definição
    if n % 2 == 0:
        return False

    # identificar variáveis auxiliares do algoritmo
    r = 0
    d = n-1
    while d % 2 == 0:
        r += 1
        d //= 2

    # continuação do algoritmo
    for _ in range(k):
        a = random.randint(2, n-1)
        x = pow(a, d, n) # x = (a**d) % n # trocado por ser mais eficiente

        if x == 1 or x == n-1:
            continue

        for _ in range(r-1):
            x = pow(x, 2, n) # x = (x**2) % n # trocado por ser mais
eficiente

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

Foi feito um teste com números de 1 a 100 e outros números grandes para testar o algoritmo.

Solovay-Stressen

Para a implementação desse código foi um pouco mais difícil, pois é preciso implementar também o algoritmo de Legendre. Com identificações de pseudo-códigos e outras implementações na internet, pôde-se chegar no seguinte código (utilizado Fibonacci e Miller-Rabin) e seguintes resultados:

```
#############################################################
# Teste de primalidade de Solovay-Stressen
# Teste probabilístico de ordem O(k * log3 n)
# Segundo o wikipedia, o algoritmo (pseudo-código) é o seguinte:
'''
repeat k times:
    choose a randomly in the range [2,n − 1]
    x = legendre(a, n)
    if x = 0 or a^((n-1)/2) != x
        return composite
return probably prime
'''
# Vamos fazer algumas adaptações, para que fique mais simples e mais
```

```
"pythônico"
# Fontes:
    # https://en.wikipedia.org/wiki/Solovay%E2%80%93Strassen_primality_test
    # https://www.geeksforgeeks.org/primality-test-set-4-solovay-strassen/
##############################################################

import random # para pegar números aleatórios dentro do "range" de números

# função auxiliar usada para resolver primalidade
# de Euler, no algoritmo de Solovay-Stressen,
# usando Legendre
def legendre(a, n):
    if a == 0 or a == 1:
        return a

    if a % 2 == 0:
        r = legendre(a//2, n)

        if pow(n, 2) - 1 & 8 != 0:
            r *= -1
    else:
        r = legendre(n%a, a)
        if (a-1) * (n-1) & 4 != 0:
            r *= -1

    return r

# n = valor de teste de primalidade
# k = parâmetro que determina a precisão do teste
def SolovayStressen(n, k):
    # se n for <= 3, é fixamente primo
    if n <= 3:
        return True

    # qualquer múltiplo de 2 (par), não é primo por definição
    if n % 2 == 0:
        return False

    # continuação do algoritmo
    for _ in range(k):
        a = random.randint(2, n-1)
        x = legendre(a, n)
        aux = (n-1)//2

        if x == 0 or (pow(a, aux) % n) != (x % n):
            return False

    return True
```

Uso para geração de primos grandes

Ao testar os algoritmos, foi então gerado números aleatórios primos com o seguinte código:

```
import time
from PrimeNumbers.MillerRabin import MillerRabin
from PrimeNumbers.SolovayStrassen import SolovayStressen
from PRNG.FibonacciGenerator import FibonacciGenerator
from PRNG.BBSGenerator import BBSGenerator

# calcular o tempo de uma função
# https://stackoverflow.com/questions/1557571/how-do-i-get-time-of-a-
python-programs-execution
def profile(fct):
  def wrapper(*args, **kw):
    start_time = time.time()
    ret = fct(*args, **kw)
    print("{} {} {} return {} in {}
seconds".format(args[0].__class__.__name__,

args[0].__class__.__module__,
                                                    fct.__name__,
                                                    ret,
                                                    time.time() -
start_time))
    return ret
  return wrapper

@profile
def calcularProcessamento(generator, tamanho):
    print(tamanho)
    k = 2
    while True:
        randomNumber = generator.generateNumOfBits()
        if MillerRabin(randomNumber, k) == True:
            print(randomNumber)
            break

def main():
    tamanhoBits = [40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096]

    for tamanho in tamanhoBits:
        generator = FibonacciGenerator(sizeBit=tamanho)
        calcularProcessamento(generator, tamanho)

if __name__ == "__main__":
    main()
```

Tamanho (bits)	Tempo de processamento (s)
40	0,000487804412842
56	0,000592708587646
80	0,00013279914856

Tamanho (bits)	Tempo de processamento (s)
128	0,00293231010437
168	0,009145498275757
224	0,006147861480713
256	0,014803171157837
512	0,008777856826782
1024	1,72275733947754
2048	21,1480839252472
4096	58,5916748046875

Foi possível chegar nos seguintes números primos:

- 40 bits: 639959461061
- 56 bits: 41302803960498653
- 80 bits: 1110464721812197119711419
- 128 bits: 256077653534100943071996657463146132179
- 168 bits: 317498160766395487349710983714406224440972216131603
- 224 bits: 23813261452471940964967936515063140055070444710553864209691171239709
- 256 bits: 114627241809324025826964415894827240558345761906494063884934186852869439070671
- 512 bits: 10196985030404639646526193552698375638599823572364076001251686821614875961860656386579909369670866958897071772389798132295349021099849371613672845065100311
- 1024 bits: 124841135873087571025134458208024600806513819708040693752171810277252135558643198642662603424937209097871519977637065732492177834078747540252162856394837331375766618468478883468362325529921575519859656970760219581403658216095248561943027394634173680229836679864353157777766887862582486567893730031156542733313
- 2048 bits: 27588121827707265114401733330075021308212629465440494674321227413159462311094408019112666321525714871378761289874416922093786507771830925417143018971421994724166241966660310568696160602018749350754656467057684776306775308818581112959020196926413359745814049979501137591451453556403541706250472812117768964352288061289530322145431245787393520341285822222109945030895808451344037237556593872359284968586803316521136525257160519122169081666445419668617595240004400986039643983205994241151585235860687829272892527569029209124178779680439536517471314476375569578526535029643454164827279532794676993491308931470554917227799
- 4096 bits: 581264768892650102889863285375976671974311901407700391098903164015305611030480796169710633923722018644143424385881774566270297085756073321759314662659030854522088273491632404341318427488737420108852940644178675765715971449235271522510019163240035130236298885695332996767174730659147259104318047909220915575309873538801802893146258720582064405716215743726964189385957384005021402050424082025067433481226080

613463566457541305142927919181959311156003383255170006815037271500937087856764219
352343276737414599266608671518608578938355138525279920228836619346713968286782367
524361464788917804055162244619485159817343165530336903703196475521320963736043346
342622957324735171836624496737803442353025143178601405752682968344633489950366855
279052465175921074326217712413137156339943193467362305520405528461381538127270589
056926104817333943241107903791752532232590801949782146668279002071056291382666109
726626310464945695632507579396397890608615888476642836080085007132578112473875899
879594011964565960669207969335347832788921111811789782874420183097143213894422415
108576475810077327805090492500666138640652003758809905103078171631047072734287187
750193051688778324702015850084573847798885135462659397952381748021882086544277877
048473691705055433

Dificuldades encontradas

Como para testar os algoritmos para números grandes, foi preciso bastante tempo e então, foi difícil saber se a veracidade dos dados eram satisfatórios. Mas após várias tentativas e entendimento dos algoritmos, bastou esperar para o computador processar os códigos solicitados.