Nome: José Luis Bressan Ruas
Matrícula: 10202969

Trabalho Individual 2 - Geração de Números Pseudo-Aleatórios

1. Algoritmos Escolhidos

Blum Blum Shub (BBS): Gera uma sequência de bits z1, ..., zl de tamanho l.
1. Configuração: Gera 2 números secretos aleatórios, distintos e primos p e q, devem ser congruente a 3 módulo 4 e calcula n = p*q.
2. Seleciona um número inteiro aleatório s (a semente) no intervalo [1, n-1] tal que mdc(s, n)=1 e calcula x0 ←s^2 mod n.
3. Para i de 1 até l faça o seguinte:
    3.1 xi←xi-1^2 mod n.
    3.2 zi←o bit menos significativo de xi.
4. A sequência da saída é z1, z2,... ,zl.

Decisões de implementação: A geração de números primos é feito com a função next_prime da biblioteca gmpy.

Referências: 1 e 4.

Linear congruential generator (LCG): A sequência de números aleatórios {Xn} é obtida por uma equação iterativa: Xn+1 = (a*Xn + c) mod m, tal que,

m é o módulo, m > 0
a é um fator multiplicador, 0 < a < m
c é um incremento, 0 <= c <= m
X0 é uma semente, 0 <= X0 < m

A sequência produzida serão números inteiros entre o intervalo 0 <= Xn < m. A escolha dos valores das variáveis a, c e m é muito importante, pois dependendo dos valores das variáveis o número de valores gerados dentro do intervalo não será satisfatório, existem valores apropriados para essas variáveis descritas na Wikipedia que passam pelos 3 testes que avaliam um gerador de número aleatório proposto por [PARK88a].

Note que a semente de um número gerado com o LCG é o último número pseudo-aleatório gerado, para Xi, i != 0.

Decisões de implementação: Os valores escolhidos para a, c e m são para números de grandeza 2^32.

Referências: 1, 3 e 5.

Xorshift 32bits : Produz uma sequência de 2^32 - 1 inteiros x com uma sequência de deslocamentos (shifts) para direita ou esquerda. O algoritmo proposto por Marsaglia é o seguinte:

```
semente s, s != 0
s ^= (s<<13)
s ^= (y>>17)
s ^= (y<<5)
return s;
```

Note que a semente de um número gerado com o Xorshift 32bits é o último número pseudo-aleatório gerado, para Si, i != 0.

Decisões de implementação para os 3 algoritmos: Algoritmos PRNG precisam de uma semente, as sementes nas implementações foram geradas atráves da biblioteca random, função getrandbits(grandeza do número).

Referências: 2 e 6.

2. Análise de tempo e de tamanho do número por algoritmo

Sequência de número de bits para a geração de números pseudo-aleatórios:
BITS_NUMBER = (32, 40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096)

XorShift 32bits: Só funciona para 32bits, os deslocamentos de bits são específicos, otimizados e testados para este tamanho, conforme pode ser visto no artigo do Marsaglia nas referências.

BBS: Funciona para qualquer quantidade de bits, visto que a escolha do módulo é um número grande tal que p e q satisfaçam as regras explicadas anteriormente.

LCG: Existem valores otimizados para m, c e a, na Wikipedia e no artigo encontrado na internet “TABLES OF LINEAR CONGRUENTIAL GENERATORS OF DIFFERENT SIZES AND GOOD LATTICE STRUCTURE” tem uma tabela com tais valores, os valores descritos nesta tabela são para poucos valores de m que é o módulo, estes valores vão de 2^8 até 2^128, ou seja, no máximo grandeza de 128bits.

Segue abaixo uma tabela com o tempo de execução para cada algoritmo com um tamanho de número.

Algoritmo	**Tamanho do Número**	**Tempo (microsegundos)**
XorShift	32	1.39
BBS	32	55.7
LCG	32	0.87
BBS	40	62.57
BBS	56	80.96

BBS	80	98.17
BBS	128	175.82
BBS	168	274.17
BBS	224	341.62
BBS	256	443.72
BBS	512	2079.97
BBS	1024	15916.2
BBS	2048	164809.82
BBS	4096	2018222.2
LCG**	40	1.08
LCG**	56	0.83
LCG**	80	0.89
LCG**	128	0.98
LCG**	168	1.03
LCG**	224	1.13
LCG**	256	0.92
LCG**	512	1.15
LCG**	1024	1.19
LCG**	2048	1.7
LCG**	4096	2.05

**O LCG só foi testado com valores de a, c e m ideais para 32 bits, os testes para outras grandezas utilizaram esses mesmos valores, caso exista valores ideias para essas grandezas a única operação a ser executada continuará sendo um módulo.

3. Código dos algoritmos geradores de números pseudo-aleatórios

XorShift:

```
class XorShift():
   #seta os valores do algoritmo xorshift32bits
```

```
    #mod = modulo e seed = semente
    def __init__(self, mod, seed):
        self.seed = seed
        self.mod = mod
        pass
    #algoritmo xorshift32 bits do artigo Xorshift RNGs de George Marsaglia
    def generate_random(self):
        x = self.seed
        x ^= x << 13
        x ^= x >> 17
        x ^= x << 5
        self.seed = x % self.mod
        return self.seed
```

LCG:

```
class LCG:
    #seta os valores do algoritmo congruencia linear
    #m = modulo; a = multiplicador; c = incremento; seed = x0
    def __init__(self, a, c, m, seed):
        self.a = a
        self.c = c
        self.m = m
        self.seed = seed

    #gera o numero aleatorio Xi = (a*xi-1 + c) % modulo
    def generate_random(self):
        self.seed = (self.a * self.seed + self.c) % (self.m)
        return self.seed
```

BBS:

```
import random
import gmpy
from fractions import gcd

class BBS:
    #inicializa com a grandeza/numero de bits que deseja gerar os numeros aleatorios
    #: a pseudorandom bit sequence z1, z2,... ,zl of length l is generated.
    def __init__(self, n_bits):
        self.n_bits = n_bits
        self.mod = 0
        self.seed = 0
        self.p = 0
        self.q = 0

    def get_p(self):
```

```
        return self.p

    def get_q(self):
        return self.q

    def get_mod(self):
        return self.mod

    def get_seed(self):
        return self.seed

    # Gera os numeros primos p e q e retorna o valor de n = mod
    #p x q = n = mod
    # p = q = 3 (mod 4)
    #nem sempre essa funcao faz o mod ter o n_bits desejado
    #1. Setup. Generate two large secret random (and distinct) primes p and q (cf. Note 8.8),
each congruent to 3 modulo 4, and compute n = pq.
    def generate_mod(self):
        n_bits_primes = int(self.n_bits/2)
        while self.p == 0:
            #gera um numero aleatorio com uma grandeza de n_bits/2 e pega o proximo numero
primo provavel
            p = gmpy.next_prime(random.getrandbits(n_bits_primes))
            if p % 4 == 3:
                self.p = p
        while self.q == 0 or self.q == self.p:
            q = gmpy.next_prime(random.getrandbits(n_bits_primes))
            if q % 4 == 3:
                self.q = q
        self.mod = self.p * self.q
        return self.mod

    #gera o valor da semente, a semente eh um numero aleatorio que seja relativamente
primo de n = mod e retorna o valor da semente
    #nem p nem q podem ser fatores de seed
    #2. Select a random integer s (the seed) in the interval [1, n−1]such that gcd(s, n)=1, and
compute x0←s2 mod n
    def generate_seed(self):
        while self.seed == 0:
            seed = random.getrandbits(self.mod.bit_length())
            if gcd(seed, self.mod) == 1 and self.seed < self.mod:
                self.seed = seed
        return self.seed

    #For i from 1 to l do the following:
    # 3.1 xi←x2i−1 mod n=mod.
```

```
    # 3.2 zi← the least significant bit of xi.
    def generate_random(self):
        self.generate_mod()
        self.generate_seed()
        self.seed = (self.seed**2) % self.mod
        z = 0
        if self.seed != 0 and self.mod != 0:
            for i in range(self.n_bits):
                self.seed = (self.seed**2) % self.mod
                z = (z << 1) | self.seed % 2
            return z
        return 0
```

Testes de tempo de execução e geração de um número pseudo-aleatório:

```
from lcg import LCG
from xorshift import XorShift
from bbs import BBS
import time
import random

BITS_NUMBER = (32, 40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096)
#calcula uma media de tempo de execução para cada tamanho de número de
BITS_NUMBER
def test_time_bbs():
        for n_bits in BITS_NUMBER:
            #print de um numero aleatorio gerado
            bbs = BBS(n_bits)
            print("BBS " + str(n_bits) + "bits : ", bbs.generate_random())

            #gera 100 numeros aleatorios com o bbs e pega a media do tempo
            sum_time = 0
            for i in range(100):
                time_before = int(round(time.time() * 1000000))
                bbs = BBS(n_bits)
                bbs.generate_random()
                time_after = int(round(time.time() * 1000000))
                sum_time += time_after - time_before
            print("tempo BBS " + str(n_bits) + "bits :", sum_time/100, " microsegundos")
#calcula uma media de tempo de execução para cada tamanho de número de
BITS_NUMBER
def test_time_lcg():
        for n_bits in BITS_NUMBER:
            lcg = LCG(1664525, 1013904223, 2**n_bits, random.getrandbits(n_bits))
            #print de um numero aleatorio gerado
            print("LCG " + str(n_bits) + "bits : ", lcg.generate_random())
```

```
        #gera 100 numeros aleatorios com o lcg e pega a media do tempo
        sum_time = 0
        for i in range(100):
            time_before = int(round(time.time() * 1000000))
            lcg.generate_random()
            time_after = int(round(time.time() * 1000000))
            sum_time += time_after - time_before
        print("tempo LCG " + str(n_bits) + "bits :", sum_time/100, " microsegundos")
#calcula uma media de tempo de execução para números de 32bits
def test_time_xs():
    #32bit XorShift
    xs = XorShift(2**32, random.getrandbits(32))
    #print de um numero aleatorio gerado
    print("XorShift 32bits : ", xs.generate_random())

    #gera 100 numeros aleatorios com o XorShift e pega a media do tempo
    sum_time = 0
    for i in range(100):
        time_before = int(round(time.time() * 1000000))
        xs_rn1 = xs.generate_random()
        time_after = int(round(time.time() * 1000000))
        sum_time += time_after - time_before
    print("tempo XS 32bits : ", sum_time/100, " microsegundos")


if __name__ == '__main__':
    test_time_bbs()
    test_time_lcg()
    test_time_xs()
```

Saída do código de testes:

```
BBS 32bits :  3593513922
tempo BBS 32bits : 48.46  microsegundos
BBS 40bits :  65446631822
tempo BBS 40bits : 59.34  microsegundos
BBS 56bits :  13527612334237944
tempo BBS 56bits : 72.16  microsegundos
BBS 80bits :  216557275106747287298627
tempo BBS 80bits : 96.42  microsegundos
BBS 128bits :  208839645244930691954590453431671368379
tempo BBS 128bits : 169.6  microsegundos
BBS 168bits :  656158278494212703640546895293848714555179819467
tempo BBS 168bits : 288.8  microsegundos
BBS 224bits :
23535497291043272728911328602699340266422780180171321297157686876955
```

tempo BBS 224bits : 358.24 microsegundos
BBS 256bits :
47893108733698070181751944569687337127892168478596206008679519134862390590574
tempo BBS 256bits : 460.79 microsegundos
BBS 512bits :
499557418122738697220993365033959083750023040375416535550251802109036341392373679230881479907599969108573734795569075983625430316818303657293369226746
0061
tempo BBS 512bits : 2141.73 microsegundos
BBS 1024bits :
293874484591194035883050806719560784732218671103895817467886305459984553911353590373251002455641604549419808079377357518620849553873616736635676986186847984766982865846247415239314103647063181072142169034479047645958620444907643190682882389441608784809130507596205985775921998055076245473459227460097880101
13
tempo BBS 1024bits : 14209.99 microsegundos
BBS 2048bits :
848986267904666502078732894112692246969370369110857521114606415209835568594954486004893073093942817857902956087136743030442734123025227564755768962692487188776081235980492527996977880189942797958219551635744563798048541576492221124326870082167724727078668874426331626036397986410979630462405088484945172015027878780797961290404252147426130630740556031622413215121343301274103654122892421648896082306318167352553448156362290976907504063795220809192331148047689542383749634784989928276087790391879326448035246574718979099802666996496768373350624101941754668777415122937756574653141126830605990099189482777647395043587
3
tempo BBS 2048bits : 161729.34 microsegundos
BBS 4096bits :
6784946209076892304761107959052954949867821071232226635812396790370141847275559108490412903483431053518613133844973322934226186047409263824427912004127160476633367167747146470061169641714933104253525239311674895555117307885496754234147715029587634091411159295780480829200335219006258639420383816528091968592370650244222570678953715700614978472365175511744507439399229437970294964510697539007499814842483143834437902883907758149743094066898094223406927573685507555652807313879528126413686194403887698750217132603232131818101856442834900119373021222367362726239915197389567120333360507486638946080942765469834023010329690786980350854746910248325769148020791925977782276970477727861714249418633561773250527044886835577471510540572513255510389073553470778265916540196218968996004019987557328651016034619670530563668084890391851134117412359881317152402934466628295280298907362597678206524971065187163663404503946886794211549013717451683998272531910341923852502980898984665512611354148612369941273712704614645663108768462180356101319604914388342648257747607229925918076025156616472060112277695528110705709557128820552293250235741674437780327238400152759216666219462640312337199940648724250067073384948518014422113719947666718195786040453
4

tempo BBS 4096bits : 1685573.55 microsegundos
LCG 32bits : 3701785372
tempo LCG 32bits : 0.84 microsegundos
LCG 40bits : 827174327342
tempo LCG 40bits : 1.03 microsegundos
LCG 56bits : 37127357958766606
tempo LCG 56bits : 1.02 microsegundos
LCG 80bits : 623910474906486923448258
tempo LCG 80bits : 0.87 microsegundos
LCG 128bits : 249032628299517602161109774198922933571
tempo LCG 128bits : 0.82 microsegundos
LCG 168bits : 371114045004126989322636857474350015974510711522574
tempo LCG 168bits : 0.83 microsegundos
LCG 224bits :
16597288093196406370679184901741412698047265977168224821895671684612
tempo LCG 224bits : 0.88 microsegundos
LCG 256bits :
6212611674005897310863925808022066335386058326654958504332275771208054508
1496
tempo LCG 256bits : 1.0 microsegundos
LCG 512bits :
8976042432854903973231682290822467237598932720481333803786215179940123219
1180917000638127849283843552591119089887519112292395808184257739642876805
93169640
tempo LCG 512bits : 0.91 microsegundos
LCG 1024bits :
8926572784198942273898091021414794756507727846699491498830845925144837681
6928218714881802486401466072620049749883532950949846114920674171868124847
8795379478370562128346072234899895893172460195277745946216975935800630400
5178880012232006973944628004395756868246828698963486780132419626451357090
5260441101511186
tempo LCG 1024bits : 1.38 microsegundos
LCG 2048bits :
1549419454101297613965832840687361396758543426245153484708008965699714624
3778771845527984096942747340965232871195125444430416556031827822433917770
1238889663099911982705591057692440731208634879991592583490376806179103967
2445768886893494549468134436952870819513211933442546778299795428481427080
9011113679840602105837640494859830259269145554202902075060962927767063890
8290273608704647645805037634774703967168435069160382698250567189364753929
5055406988080518651838302766191005264138980170170689930226140891939749856
3584381819347997681968544684069412532301814856867447735684893712871224407
357387487394692181063642519811566
tempo LCG 2048bits : 1.35 microsegundos
LCG 4096bits :
8697404757494845072565017552570778707039245876377660120817686699273455839
7522897853085996020950966522998384343185872241262963292455906867505535812

0205828502285310991722776518273556234395823648272453895259979287797289077
2394072616341911106026343354785464453369580140374590076426464927977496903
3412132907364904198521661662602876494387520502274692482744026844313805470
1065798972154310553608889556661079751165508442175761237225185230974032890
6545318513866582458138904792159420224491550054055021021760675322904886732
5306257742692175170803274800263231695574169955850659125041685297819012961
2338773483554714403579136099498313048055916005482390955969882526942962790
4162152242138923787870217715557535605720948152912477950173413229286416426
8026288599999390764643607725398564722763093488882649093936611015153371526
0432574688830296521597932165510385628875229726570967830405715817753066258
6645928938013488372690454495721588362826188452468806518998941988099474686
3662037464436024102268051933063080850441410482995423722946638838302810852
6005326226076633191125106619692497861706122859335276807244218637123543526
0469395705114626631685402290299178540554793373419526196987126957242249632
23042907343080036060019418346423669643280604234836739099881284134
tempo LCG 4096bits : 1.99 microsegundos
XorShift 32bits : 4259151135
tempo XS 32bits : 1.06 microsegundos

4. Comparação entre os algoritmos

Com relação ao tempo de execução o BBS foi aproximadamente 50 vezes mais lento que o LCG e o XorShift para números de 32 bits.
Com relação a aleatoriedade segundo Marsaglia o XorShift 32 bits passa na maioria dos testes de verificação de aleatoriedade, exceto no binary rank test.
Com relação a aleatoriedade segundo Stallings o LCG não possui nada de aleatório além da escolha de X0, a sequência de números seguem de uma forma determinística permitindo criptoanálise.
O BBS segundo Stallings talvez tem a prova pública mais forte em termos de força criptográfica dos algoritmos de geração de números pseudo-aleatórios, é conhecido como um gerador de bits pseudo-aleatórios criptograficamente seguro.

5. Análise de Complexidade

A operação de módulo para números grandes no python tem complexidade de O(n^2)
A operação de deslocamento de bits tem a complexidade de O(1)
A função gmpy.next_prime() tem uma complexidade muito maior que as 2 outras operações.

O LCG utiliza somente uma operação de módulo para gerar um número pseudo-aleatório, ou seja, O(n^2).
O XorShift utiliza 3 operações de shift mantendo a sua complexidade em O(1), porém nos testes de tempo médio o XorShift obteve um valor maior que o LCG, ou seja, eles devem ter a mesma complexidade para números grandes.
O BBS utiliza uma operação de módulo para gerar cada bit do número aleatório, ou seja,

O(n_bits * n^2), também é necessário fazer uma operação de deslocamento para armazenar o resultado em z, calcular 2 números primos e verificar o mdc entre n=mod e a semente.
Estimar a complexidade exata do BBS na linguagem python na minha implementação seria uma tarefa bem árdua, a quantidade de operações utilizadas neste algoritmo é muito maior que no LCG e XorShift, os testes de tempo de execução apontaram que o BBS é muito mais demorado que os outros 2 algoritmos, ou seja, muito mais complexo.

6. Medição de aleatoriedade

Retirado do livro "Criptografia e Segurança de Redes: princípios e práticas - 6ª Ed. 2014 - William Stallings" existem 3 requisitos de aleatoriedade para um Pseudo-random number generator (PRNG):

1. **Uniformidade:** O número esperado de 0s e 1s é n/2, n é o tamanho da sequência.
2. **Escalabilidade:** Qualquer subsequência extraída do número deverá passar em qualquer teste de aleatoriedade.
3. **Consistência:** Inadequado testar um PRNG com base em uma única semente.

Também retirado do livro 3 testes para medir aleatoriedade são listados abaixo:
1. **Teste de frequência:** O número de 1s e 0s em uma sequência é mais ou menos o mesmo que seria esperado para uma sequência verdadeiramente aleatória.
2. **Teste de rodadas:** O número de rodadas de 1's e 0's de vários tamanhos é aquele que se espera para uma sequência aleatória.
3. **Teste da estatística universal de Maurer:** Detectar se uma sequẽncia pode ou não ser comprimida sem perder informação.

Para aplicar o teste de frequência nos algoritmos implementados neste trabalho em python seria necessário de um True Random Number Generator (TRNG), gerar um número com a mesma grandeza com o TRNG e um para os 3 algoritmos PRNG implementados e verificar se a frequência de 1s e 0s é mais ou menos o mesmo entre o número gerado com TRNG e o outro gerado com PRNG.

Referências

1 - Criptografia e Segurança de Redes: princípios e práticas - 6ª Ed. 2014 - William Stallings
2 - https://en.wikipedia.org/wiki/Xorshift
3 - https://en.wikipedia.org/wiki/Linear_congruential_generator
4 - http://cacr.uwaterloo.ca/hac/about/chap5.pdf
5 - https://pdfs.semanticscholar.org/8284/542deb19d556c8818e0456cce771a50ed0ff.pdf
6 - Xorshift RNGs, Journal of Statistical Software, Julho 2003 - Marsaglia, George. Link para acesso: http://www.jstatsoft.org/v08/i14/paper