Relatório do segundo trabalho

Maria Eduarda de Melo Hang (17202304)

20 de julho de 2021

1 Introdução

Este trabalho consiste na geração de número primos pseudo-aleatórios de 40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048 e 4096 bits. A linguagem utilizada para a implementação foi o Python 3.8.10 (Python3) devido à familiaridade da aluna com a linguagem.

2 Geração de números pseudo-aleatórios

Os algoritmos utilizados para a geração de números pseudo-aleatórios foram *Linear Congruential Generator* (LCG) e *Blum Blum Shub* (BBS), ambos foram escolhidos devido à simplicidade de implementação.

2.1 LCG

Ao pesquisar sobre o algoritmo na internet, foi possível encontrar uma implementação em Python [Wik21c]. Foram necessárias as seguintes adaptações para atender às exigências do trabalho: alterar o parâmetro m para sempre seguir a forma 2^b, sendo b a quantidade de bits necessária, verificar se o número resultante contém a quantidade de bits solicitada e, caso tenham pedido um número primo, o *least significant bit* (LSB) sempre receberá o valor 1 para evitar a geração de números pares, que é um desperdício de processamento, já que números pares não são primos para números com mais de 2 bits (apenas o 2 seria considerado um número primo nesse caso). Os parâmetros a e c foram retirados da tabela presente na seção *Parameters in common use* [Wik21c]. O trecho de código 1 contém a implementação realizada.

```python
class LCG:
    def __init__(self, seed: int = 1, num_bits: int = 32, a: int = 1664525, c : int = 1013904223):
        self.num_bits = num_bits
        self.m = 2 ** num_bits
        self.a = a
        self.c = c
        # X_0 = seed
        self.state = seed

    # Atribui o valor do parâmetro m como 2 ** num_bits e atribui o valor de bits solicitado
    def __m__(self, num_bits: int):
        self.num_bits = num_bits
        self.m = 2 ** num_bits

    # Gera um numero pseudo-aleatorio
    def next(self, prime: bool = False):
        # X_n+1 = (a * X_n - c) mod m
        self.state = (self.a * self.state + self.c) % self.m

        # Enquanto o número não tiver o bit mais significativo (MSB) igual a 1,
        # esse numero não contém a quantidade de bits necessária, então continue gerando
        # Exemplo:
        # Se queremos um número com 40 bits, pelo menos o bit 40 deve ser 1,
        # já que apenas os zeros à direita são contabilizados na quantidade de bits de um número
        while self.state < (2**(self.num_bits - 1)):
            self.state = (self.a * self.state + self.c) % self.m

        # Não retornar números pares caso estejam pedindo um primo, é perda de tempo
        # Coloque o LSB como 1
        if prime:
            return self.state | 1

        # Retorne o valor obtido no cálculo
        return self.state
```

Listing 1: Gerando os números através do LCG

Em relação à complexidade do algoritmo, foi necessário realizar um teste simples para verificar quantas vezes o *while* presente na linha 25 do código 1 seria executado, uma vez que caso se provasse com complexidade linear, todo o algoritmo se tornaria $O(n)$, já que todas as outras operações são $O(1)$. Com isso, o algoritmo foi executado para gerar 1000 números para cada valor de bits, como pode ser visto no código 2 e o método *next* do LCG foi adaptado para contabilizar as execuções, como pode ser visto no código 3.

```python
import time
from typing import Dict, List
from lcg import LCG

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

seed =  int(time.time())

lcg = LCG(seed=seed)

# Quantidade de numeros a serem gerados para cada valor de bits
quantity_of_numbers = 1000

# Dicionario para guardar quantas vezes o while foi executado para cada valor de bits
dict : Dict[int, List[int]] = {}

# Inicializando as listas dentro do dicionário
for b in solicited_bits:
    dict[b] = []

# Abrindo um arquivo no modo escrita
with open("./generate_random.txt", "w") as writer:
    writer.write("===== LCG =====\n")

    for numBits in solicited_bits:
        # Colocando o valor do parametro m
        lcg.__m__(numBits)
        writer.write(f"Começando a gerar números de {numBits} bits\n")

        for _ in range(quantity_of_numbers):
            lcg.next(dict=dict)

    for b in dict.keys():
        writer.write(f'Para números de {b} bits, foi necessário na média {sum(dict[b])/len(dict[b])} tentativas\n')
```

Listing 2: Arquivo de teste para avaliar a quantidade de execuções do while

```python
# ...
# Método adaptado da classe LCG para contabilizar as tentativas
    def next(self, prime: bool = False, dict: Dict[int, List[int]]=None):
        # X_n+1 = (a * X_n - c) mod m
        self.state = (self.a * self.state + self.c) % self.m

        i = 0
        # Enquanto o número não tiver o bit mais significativo igual a 1, esse numero nao contem a quantidade de bits necessar
        while self.state < (2**(self.num_bits - 1)):
            self.state = (self.a * self.state + self.c) % self.m
            i += 1
        dict[self.num_bits].append(i)

        # Nao retornar numeros pares caso estejam pedindo um primo, eh perda de tempo
        if prime:
            return self.state | 1

        return self.state
```

Listing 3: Método next do LCG adaptado

Os valores obtidos podem ser vistos na Tabela 1. A partir desse resultado, torna-se claro que a complexidade do algoritmo é $O(1)$.

Quantidade de bits	Média de execuções do while
40	0,978
56	0,941
80	0,999
128	0,992
168	0,95
224	0,978
256	1,045
512	1,043
1024	0,998
2048	1,105
4096	1,106

Tabela 1: Resultados obtidos após a execução do arquivo teste

2.2 BBS

A implementação do BBS foi baseada no algoritmo original [Wik21a] com a estratégia de pegar o LSB do resultado. Assim como no LCG, algumas adaptações foram necessárias, como criar um método para gerar um número com a quantidade de bits necessárias, uma vez que o BBS gera apenas o próximo bit, verificar se o número tem a quantidade de bits necessárias e alterar o LSB para 1, caso seja solicitado um número primo. Os parâmetros p e q, como deveriam ser números primos grandes e atender a alguns critérios (que podem ser vistos no código abaixo), foram encontrados em outras implementações na internet. O código 4 contém a implementação realizada.

```python
from math import gcd

class BBS:
    def __init__(self, seed: int = 1, p: int = 3141592653589771, q: int = 2718281828459051) -> None:
        if p == q:
            raise ValueError("Os valores de p e q não podem ser iguais")
        if p % 4 != 3 or q % 4 != 3:
            raise ValueError("Os valores de p e q devem ser congruentes a 3 (mod 4)")
        if gcd(p,q) != 1:
            raise ValueError("Os valores p e q devem ser relativamente primos, ou seja, gcd(p,q) = 1")
        self.m = p * q
        self.state = seed

    # Gera o próximo bit
    def next(self):
        # X_n+1 = ( X_n ** 2 ) mod m
        self.state = pow(self.state, 2, self.m)

        # B_n = X_n mod 2 (pegando o LSB)
        return self.state % 2

    # Gera um número com uma quantidade de bits
    def generate(self, num_bits: int, prime: bool = False) -> int:
        number = 0
        for _ in range(num_bits):
            number = (number << 1) | self.next()

        # Uma solução proposta, mas que não foi utilizada porque não parecia adequada
        # Se o valor não atender ao número de bits, coloque o valor 1 no MSB
        # if number < (2**(num_bits - 1)):
        #     number = 1 << num_bits - 1 | number

        # Se o número não atender à quantidade de bits (MSB=0), gere novamente
        if number < (2**(num_bits - 1)):
            return self.generate(num_bits, prime)

        # Não retornar números pares caso estejam pedindo um primo, é perda de tempo
        if prime:
            return number | 1

        return number
```

Listing 4: Método BBS

Se considerarmos apenas o método *next*, a complexidade do algoritmo também seria $O(1)$. Todavia, se considerarmos o *generate* teríamos uma complexidade $O(n^2)$, sendo n a quantidade de bits solicitada. Caso não existisse o if da linha 34, a complexidade seria $O(n)$.

2.3 Gerando os números pseudo-aleatórios

Para avaliar o desempenho de cada algoritmo na geração dos números, foi necessário criar um arquivo de teste, podendo ser visto nos códigos 5 e 6.

```python
import time
from util import find_number_of_bits
from lcg import LCG
from bbs import BBS

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

seed =  int(time.time())

lcg = LCG(seed=seed)

# Como o SO pode escalonar os processos,
# eh importante considerar mais de um numero gerado para encontrar um valor mais proximo da realidade
# Quantidade de números que serão gerados para cada quantidade de bits
quantity_of_numbers = 1000

with open("./generate_random.txt", "w") as writer:
    writer.write("===== LCG =====\n")
    for numBits in solicited_bits:
        # Colocando o valor do parâmetro m
        lcg.__m__(numBits)

        # Tempo médio para gerar um número com numBits bits
        avg_time = 0.0

        writer.write(f"Começando a gerar números de {numBits} bits\n")
        for _ in range(quantity_of_numbers):
            # Tempo inicial
            start = time.time()

            value = lcg.next()

            # Tempo final ao gerar o valor
            final = time.time()

            # Pega a quantidade de bits que a saida gerou
            bits = find_number_of_bits(value)

            # Verifica se a quantidade de bits foi realmente atendida
            assert(numBits == bits)

            time_to_generate = final - start

            avg_time += (time_to_generate / quantity_of_numbers)

            #writer.write(f"bits = {bits} => {time_to_generate} segundos\n")
        # Escreve no arquivo o tempo médio obtido
        writer.write(f"O tempo médio para gerar um número com {numBits} bits foi {avg_time} segundos\n")
```

Listing 5: Gerando os números pseudo-aleatórios - parte 1

```
# Continuacao ...
bbs = BBS(seed=seed)

    writer.write("===== BBS =====\n")
    for numBits in solicited_bits:

        avg_time = 0.0

        writer.write(f"Começando a gerar números de {numBits} bits\n")
        for _ in range(quantity_of_numbers):
            # Tempo inicial
            start = time.time()

            value = bbs.generate(num_bits=numBits)

            # Tempo final ao gerar o valor
            final = time.time()

            # Pega a quantidade de bits que a saida gerou
            bits = find_number_of_bits(value)

            # Verifica se a quantidade de bits foi realmente atendida
            assert(numBits == bits)

            time_to_generate = final - start

            avg_time += (time_to_generate / quantity_of_numbers)

            # writer.write(f"bits = {bits} => {time_to_generate} segundos\n")
        # Escreve no arquivo o tempo médio obtido
        writer.write(f"O tempo médio para gerar um número com {numBits} bits foi {avg_time} segundos\n")
```

Listing 6: Gerando os números pseudo-aleatórios - parte 2

Os resultados podem ser vistos nas Tabelas 2 e 3 para os algoritmos de LCG e BBS respectivamente. A partir desses dados, torna-se perceptível que o LCG foi mais rápido que o BBS para gerar os números. A diferença se torna mais gritante a partir dos 1024 bits em diante, chegando a ter uma redução de no mínimo duas ordens de grandeza. Caso fosse utilizada a solução comentada nas linhas 28 a 31 do código 4, haveria uma redução nessa diferença, uma vez que o BBS teria uma complexidade $O(n)$. Além disso, como era o esperado, os algoritmos consumiram mais tempo para gerar números com quantidades de bits maiores.

Quantidade de bits	Tempo média para gerar (seg)
40	2,59E-06
56	1,81E-06
80	1,77E-06
128	1,84E-06
168	1,76E-06
224	1,97E-06
256	1,95E-06
512	2,41E-06
1024	3,39E-06
2048	5,32E-06
4096	1,23E-05

Tabela 2: Resultados para LCG

Quantidade de bits	Tempo média para gerar (seg)
40	7,19E-05
56	9,67E-05
80	1,4E-04
128	2,1E-04
168	2,93E-04
224	3,89E-04
256	4,37E-04
512	8,78E-04
1024	1,85E-03
2048	3,56E-03
4096	7,37E-03

Tabela 3: Resultados para BBS

3 Números primos

Nesta seção serão descritos os algoritmos utilizados para o teste de primalidade e os resultados obtidos na geração de número primos pseudo-aleatórios.

3.1 Testes de primalidade

Os algoritmos de teste de primalidade implementados foram o de Miller-Rabin e de Fermat. O algoritmo de Fermat foi escolhido devido à simplicidade e abundância de materiais (códigos e textos) na internet para o algoritmo.

3.1.1 Miller-Rabin

A implementação pode ser encontrada no código 7, seguindo o algoritmo [Wik21d] e a implementação de base desenvolvida em Python [Gee21b]. As alterações feitas foram mais apenas para melhorar a legibilidade do código. A complexidade do algoritmo é $O(k * log^3 n)$ [Wik21d], sendo k o número de iterações e n o número testado.

```python
from random import randint

def miller_rabin_test(n: int, k: int = 100) -> bool:
    # Nenhum numero menor que 2 eh primo
    if n < 2:
        return False

    # Casos base
    if n in [2,3]:
        return True

    # Numeros pares nao sao primos
    if n % 2 == 0:
        return False

    # Eh necessario decompor o n como 2^r * d + 1, sendo d um numero impar
    r, d = decompose(n-1)

    for _ in range(k):
        # Pegue um valor aleatorio entre [2, n-2]
        a = randint(2, n - 2)

        # x = a^d mod n
        x = pow(a, d, n)

        # Se x = 1 ou x = -1, continue o loop
        if x in (1, n - 1):
            continue

        continue_loop = False
        for _ in range(r - 1):
            # x = x^2 mod n
            x = pow(x, 2, n)

            # Se x == n -1, continue o loop
            if x == n - 1:
                continue_loop = True

        if continue_loop: continue

        # Eh um numero composto
        return False

    # Eh um numero primo
    return True

def decompose(n: int):
    # Inicializando a maior potência de 2 que n pode ter como zero
    r = 0
    # Enquanto n tiver potências de 2
    while n % 2 ==0:
        # Incremente o valor de maior potência
        r += 1
        # Pegue o resto da divisão por 2
        n = n / 2
    return r, n
```

Listing 7: Teste de Miller-Rabin

3.1.2 Fermat

Assim como no teste de Miller-Rabin, a implementação se baseou no algoritmo [Wik21b] e em um código base [Gee21a], podendo ser vista no código 8. A complexidade do algoritmo é $O(k * log^2 n)$ [Wik21b], sendo k a quantidade de iterações que serão feitas e n o número a ser testado.

```python
from random import randint

def fermat_test(n: int, k: int = 100) -> bool:
    # Nenhum numero menor que 2 eh primo
    if n < 2:
        return False

    # Casos base
    if n in [2,3]:
        return True

    # Numeros pares nao sao primos
    if n % 2 == 0:
        return False

    for _ in range(k):
        # Pegue um valor entre [2,...,n-2]
        a = randint(2, n - 2)

        # a^(n-1) % n != 1 => Eh composto
        if pow(a, n - 1, n) != 1:
            return False

    # Provavelmente eh um primo
    return True
```

Listing 8: Teste de Fermat

3.2 Geração de números primos

Os códigos 9 e 10 mostram o arquivo utilizado para gerar os números primos. O algoritmo de Fermat foi usado para testar a primalidade dos números por ser mais eficiente que o de Miller Rabin. Os resultados para o LCG e BBS podem ser vistos nas Tabelas 4 e 5 respectivamente.

```python
from typing import Dict, List
from fermat import fermat_test
import time
from util import find_number_of_bits
from lcg import LCG
from bbs import BBS

solicited_bits = [40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096]
seed =  int(time.time())
lcg = LCG(seed=seed)

# Quantidade de numeros primos a serem gerados para cada quantidade de bits
quantity_of_numbers = 10

# Dicionario contendo os numeros gerados
possible_primes : Dict[int, List[int]] = {}
# Inicializando a lista
for numBits in solicited_bits:
    possible_primes[numBits] = []

with open("./generate_prime.txt", "w") as writer:
    writer.write("===== LCG =====\n")
    for numBits in solicited_bits:
        # Colocando o valor do parametro m
        lcg.__m__(numBits)

        avg_time = 0.0

        writer.write(f"Começando a gerar números primos de {numBits} bits\n")
        for _ in range(quantity_of_numbers):

            start = time.time()

            value = lcg.next()

            # Enquanto o resultado obtido nao for primo, continue gerando
            while not fermat_test(value):
                value = lcg.next(prime=True)

            final = time.time()

            # Pega a quantidade de bits que a saida gerou
            bits = find_number_of_bits(value)

            # Verifica se a quantidade de bits foi realmente atendida
            assert(numBits == bits)

            time_to_generate = final - start

            avg_time += (time_to_generate / quantity_of_numbers)

            # Adiciona o valor a lista
            possible_primes[numBits].append(value)

            # writer.write(f"bits = {bits} => {time_to_generate} segundos\n")
        writer.write(f"O tempo médio para gerar um número primo com {numBits} bits foi {avg_time} segundos\n")
```

Listing 9: Gerando números primos - parte 1

```python
# Continuacao...
    bbs = BBS(seed=seed)

    writer.write("===== BBS =====\n")
    for numBits in solicited_bits:

        avg_time = 0.0

        writer.write(f"Começando a gerar números primos de {numBits} bits\n")
        for _ in range(quantity_of_numbers):

            start = time.time()

            value = bbs.generate(num_bits=numBits)

            # Enquanto o resultado obtido nao for primo, continue gerando
            while not fermat_test(value):
                value = bbs.generate(num_bits=numBits, prime=True)

            final = time.time()

            # Pega a quantidade de bits que a saida gerou
            bits = find_number_of_bits(value)

            # Verifica se a quantidade de bits foi realmente atendida
            assert(numBits == bits)

            time_to_generate = final - start

            avg_time += (time_to_generate / quantity_of_numbers)

            # Adiciona o valor a lista
            possible_primes[numBits].append(value)

            # writer.write(f"bits = {bits} => {time_to_generate} segundos\n")
        writer.write(f"O tempo médio para gerar um número primo com {numBits} bits foi {avg_time} segundos\n")

    # Escreve no arquivo todos os valores gerados
    writer.write(possible_primes.__str__())
```

Listing 10: Gerando números primos - parte 2

Quantidade de bits	Tempo média para gerar (seg)
40	9,66E-04
56	1,41E-03
80	2,52E-03
128	5,85E-03
168	1,15E-02
224	1,91E-02
256	2,97E-02
512	1,8E-01
1024	1,55
2048	1,57E01
4096	2,24E02

Tabela 4: Tempo médio para gerar números primos no LCG

Quantidade de bits	Tempo média para gerar (seg)
40	1,47E-03
56	3,33E-03
80	7,71E-03
128	1,54E-02
168	1,58E-02
224	4,52E-02
256	6,05E-02
512	5,15E-01
1024	1,88
2048	1,69E01
4096	3,16E02

Tabela 5: Tempo médio para gerar números primos no BBS

A partir desses resultados torna-se perceptível novamente que o LCG é mais rápido que o BBS na geração dos números primos, sendo que essa diferença pode ter sido causada pelo if que chama o *generate* do BBS novamente. Além disso, vale ressaltar que o tempo gasto para gerar os números primos foi superior ao dos números, como era esperado. A Figura 1 mostra a variação percentual entre o tempo médio para gerar um número e um primo para os algoritmos LCG e BBS. Por fim, a Figura 2 mostra um possível primo gerado para cada quantidade de bits, os outros valores foram omitidos porque não caberiam na imagem.

	LCG		Variação LCG	BBS		Variação BBS
Quantidade de bits	Número Aleatório	Primo	Variação percentual	Número Aleatório	Primo	Variação percentual
40	2,59E-06	9,66E-04	3709730%	7,19E-05	1,47E-03	184451%
56	1,81E-06	1,41E-03	7770055%	9,67E-05	3,33E-03	324364%
80	1,77E-06	2,52E-03	14217288%	1,40E-04	7,71E-03	530714%
128	1,84E-06	5,85E-03	31773478%	2,10E-04	1,54E-02	713333%
168	1,76E-06	1,15E-02	65320909%	2,93E-04	1,58E-02	519249%
224	1,97E-06	1,91E-02	96934315%	3,89E-04	4,52E-02	1141954%
256	1,95E-06	2,97E-02	152287692%	4,37E-04	6,05E-02	1364439%
512	2,41E-06	1,80E-01	746867967%	8,78E-04	5,15E-01	5845604%
1024	3,39E-06	1,55E+00	4572251386%	1,85E-03	1,88E+00	10142162%
2048	5,32E-06	1,57E+01	29511258195%	3,56E-03	1,69E+01	47451910%
4096	1,23E-05	2,24E+02	182113801138%	7,37E-03	3,16E+02	428745265%

Figura 1: Variação do tempo médio para os métodos LCG e BBS ao produzir números primos

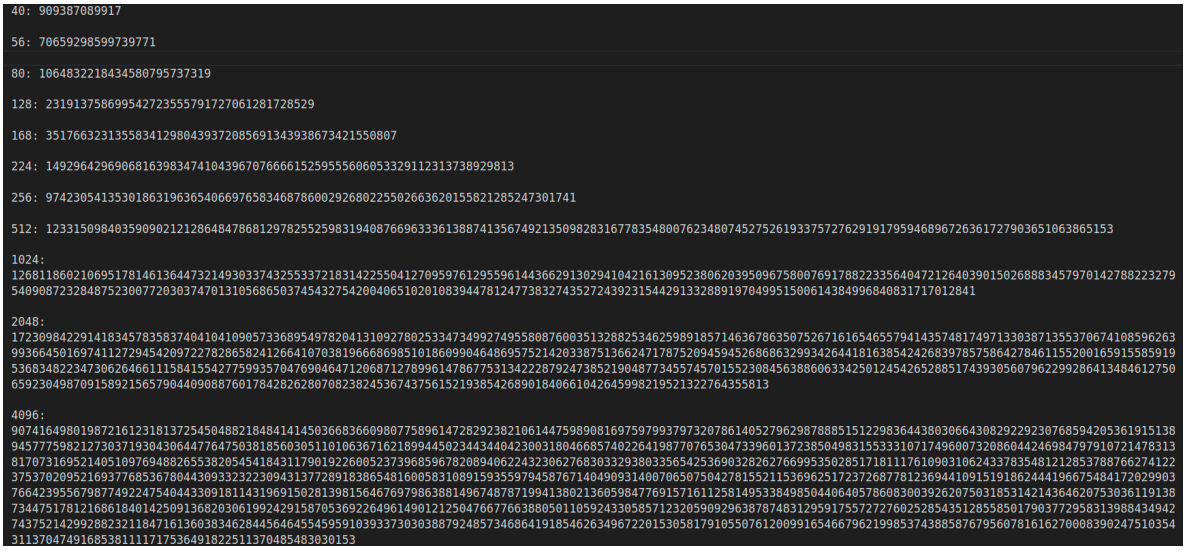

Figura 2: Um possível primo gerado para cada quantidade de bits

4 Dificuldades encontradas

As principais dificuldades foram encontrar parâmetros para os métodos LCG e BBS, porque ambos os algoritmos são muito sensíveis aos parâmetros escolhidos (além de precisar seguir algumas regras, como foi explicado anteriormente), logo foi necessária uma busca considerável para achá-los. Além disso, o tempo para gerar 10 primos grandes para cada algoritmo foi também uma dificuldade, uma vez que custou um tempo considerável (*eu deixei rodando enquanto dormia porque sabia que ia demorar*).

Referências

[Gee21a] Geek for Geeks contributors. Primality test — set 2 (fermat method), 2021. [Online; accessed 20-July-2021].

[Gee21b] Geek for Geeks contributors. Primality test — set 3 (miller–rabin), 2021. [Online; accessed 20-July-2021].

[Wik21a] Wikipedia contributors. Blum blum shub — Wikipedia, the free encyclopedia, 2021. [Online; accessed 20-July-2021].

[Wik21b] Wikipedia contributors. Fermat primality test — Wikipedia, the free encyclopedia, 2021. [Online; accessed 20-July-2021].

[Wik21c] Wikipedia contributors. Linear congruential generator — Wikipedia, the free encyclopedia, 2021. [Online; accessed 19-July-2021].

[Wik21d] Wikipedia contributors. Miller–rabin primality test — Wikipedia, the free encyclopedia, 2021. [Online; accessed 20-July-2021].