

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.

```

1  class LCG:
2      def __init__(self, seed: int = 1, num_bits: int = 32, a: int = 1664525, c : int = 1013904223):
3          self.num_bits = num_bits
4          self.m = 2 ** num_bits
5          self.a = a
6          self.c = c
7          #  $X_0 = seed$ 
8          self.state = seed
9
10         # Atribui o valor do parâmetro m como 2 ** num_bits e atribui o valor de bits solicitado
11     def __m__(self, num_bits: int):
12         self.num_bits = num_bits
13         self.m = 2 ** num_bits
14
15         # Gera um numero pseudo-aleatorio
16     def next(self, prime: bool = False):
17         #  $X_{n+1} = (a * X_n - c) \bmod m$ 
18         self.state = (self.a * self.state + self.c) % self.m
19
20         # Enquanto o número não tiver o bit mais significativo (MSB) igual a 1,
21         # esse numero não contém a quantidade de bits necessária, então continue gerando
22         # Exemplo:
23         # Se queremos um número com 40 bits, pelo menos o bit 40 deve ser 1,
24         # já que apenas os zeros à direita são contabilizados na quantidade de bits de um número
25         while self.state < (2**(self.num_bits - 1)):
26             self.state = (self.a * self.state + self.c) % self.m
27
28         # Não retornar números pares caso estejam pedindo um primo, é perda de tempo
29         # Coloque o LSB como 1
30         if prime:
31             return self.state | 1
32
33         # Retorne o valor obtido no cálculo
34         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.

```

1  import time
2  from typing import Dict, List
3  from lcg import LCG
4
5  solicited_bits = [40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096]
6
7  seed = int(time.time())
8
9  lcg = LCG(seed=seed)
10
11  # Quantidade de numeros a serem gerados para cada valor de bits
12  quantity_of_numbers = 1000
13
14  # Dicionario para guardar quantas vezes o while foi executado para cada valor de bits
15  dict : Dict[int, List[int]] = {}
16
17  # Inicializando as listas dentro do dicionário
18  for b in solicited_bits:
19      dict[b] = []
20
21  # Abrindo um arquivo no modo escrita
22  with open("./generate_random.txt", "w") as writer:
23      writer.write("==== LCG ==== \n")
24
25      for numBits in solicited_bits:
26          # Colocando o valor do parametro m
27          lcg.__m__(numBits)
28          writer.write(f"Começando a gerar números de {numBits} bits \n")
29
30          for _ in range(quantity_of_numbers):
31              lcg.next(dict=dict)
32
33      for b in dict.keys():
34          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

```

1  # ...
2  # Método adaptado da classe LCG para contabilizar as tentativas
3  def next(self, prime: bool = False, dict: Dict[int, List[int]]=None):
4      #  $X_{n+1} = (a * X_n - c) \bmod m$ 
5      self.state = (self.a * self.state + self.c) % self.m
6
7      i = 0
8      # Enquanto o número não tiver o bit mais significativo igual a 1, esse numero nao contem a quantidade de bits necessar
9      while self.state < (2**(self.num_bits - 1)):
10         self.state = (self.a * self.state + self.c) % self.m
11         i += 1
12     dict[self.num_bits].append(i)
13
14     # Nao retornar numeros pares caso estejam pedindo um primo, eh perda de tempo
15     if prime:
16         return self.state | 1
17
18     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.

```

1  from math import gcd
2
3  class BBS:
4      def __init__(self, seed: int = 1, p: int = 3141592653589771, q: int = 2718281828459051) -> None:
5          if p == q:
6              raise ValueError("Os valores de p e q não podem ser iguais")
7          if p % 4 != 3 or q % 4 != 3:
8              raise ValueError("Os valores de p e q devem ser congruentes a 3 (mod 4)")
9          if gcd(p,q) != 1:
10             raise ValueError("Os valores p e q devem ser relativamente primos, ou seja, gcd(p,q) = 1")
11             self.m = p * q
12             self.state = seed
13
14         # Gera o próximo bit
15         def next(self):
16             #  $X_{n+1} = (X_n ** 2) \bmod m$ 
17             self.state = pow(self.state, 2, self.m)
18
19             #  $B_n = X_n \bmod 2$  (pegando o LSB)
20             return self.state % 2
21
22         # Gera um número com uma quantidade de bits
23         def generate(self, num_bits: int, prime: bool = False) -> int:
24             number = 0
25             for _ in range(num_bits):
26                 number = (number << 1) | self.next()
27
28             # Uma solução proposta, mas que não foi utilizada porque não parecia adequada
29             # Se o valor não atender ao número de bits, coloque o valor 1 no MSB
30             # if number < (2**(num_bits - 1)):
31             #     number = 1 << num_bits - 1 | number
32
33             # Se o número não atender à quantidade de bits (MSB=0), gere novamente
34             if number < (2**(num_bits - 1)):
35                 return self.generate(num_bits, prime)
36
37             # Não retornar números pares caso estejam pedindo um primo, é perda de tempo
38             if prime:
39                 return number | 1
40
41             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.

```

1  import time
2  from util import find_number_of_bits
3  from lcg import LCG
4  from bbs import BBS
5
6  solicited_bits = [40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096]
7
8  seed = int(time.time())
9
10 lcg = LCG(seed=seed)
11
12 # Como o SO pode escalonar os processos,
13 # eh importante considerar mais de um numero gerado para encontrar um valor mais proximo da realidade
14 # Quantidade de números que serão gerados para cada quantidade de bits
15 quantity_of_numbers = 1000
16
17 with open("./generate_random.txt", "w") as writer:
18     writer.write("==== LCG ==== \n")
19     for numBits in solicited_bits:
20         # Colocando o valor do parâmetro m
21         lcg._m__(numBits)
22
23         # Tempo médio para gerar um número com numBits bits
24         avg_time = 0.0
25
26         writer.write(f"Começando a gerar números de {numBits} bits \n")
27         for _ in range(quantity_of_numbers):
28             # Tempo inicial
29             start = time.time()
30
31             value = lcg.next()
32
33             # Tempo final ao gerar o valor
34             final = time.time()
35
36             # Pega a quantidade de bits que a saída gerou
37             bits = find_number_of_bits(value)
38
39             # Verifica se a quantidade de bits foi realmente atendida
40             assert(numBits == bits)
41
42             time_to_generate = final - start
43
44             avg_time += (time_to_generate / quantity_of_numbers)
45
46             #writer.write(f"bits = {bits} => {time_to_generate} segundos \n")
47             # Escreve no arquivo o tempo médio obtido
48             writer.write(f"0 tempo médio para gerar um número com {numBits} bits foi {avg_time} segundos \n")

```

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

```

1  # Continuacao ...
2  bbs = BBS(seed=seed)
3
4  writer.write("==== BBS ====\\n")
5  for numBits in solicited_bits:
6
7      avg_time = 0.0
8
9      writer.write(f"Começando a gerar números de {numBits} bits\\n")
10     for _ in range(quantity_of_numbers):
11         # Tempo inicial
12         start = time.time()
13
14         value = bbs.generate(num_bits=numBits)
15
16         # Tempo final ao gerar o valor
17         final = time.time()
18
19         # Pega a quantidade de bits que a saída gerou
20         bits = find_number_of_bits(value)
21
22         # Verifica se a quantidade de bits foi realmente atendida
23         assert(numBits == bits)
24
25         time_to_generate = final - start
26
27         avg_time += (time_to_generate / quantity_of_numbers)
28
29         # writer.write(f"bits = {bits} => {time_to_generate} segundos\\n")
30     # Escreve no arquivo o tempo médio obtido
31     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.

```

1  from random import randint
2
3  def miller_rabin_test(n: int, k: int = 100) -> bool:
4      # Nenhum numero menor que 2 eh primo
5      if n < 2:
6          return False
7
8      # Casos base
9      if n in [2,3]:
10         return True
11
12     # Numeros pares nao sao primos
13     if n % 2 == 0:
14         return False
15
16     # Eh necessario decompor o n como 2^r * d + 1, sendo d um numero impar
17     r, d = decompose(n-1)
18
19     for _ in range(k):
20         # Pegue um valor aleatorio entre [2, n-2]
21         a = randint(2, n - 2)
22
23         # x = a^d mod n
24         x = pow(a, d, n)
25
26         # Se x = 1 ou x = -1, continue o loop
27         if x in (1, n - 1):
28             continue
29
30         continue_loop = False
31         for _ in range(r - 1):
32             # x = x^2 mod n
33             x = pow(x, 2, n)
34
35             # Se x == n - 1, continue o loop
36             if x == n - 1:
37                 continue_loop = True
38
39         if continue_loop: continue
40
41         # Eh um numero composto
42         return False
43
44     # Eh um numero primo
45     return True
46
47 def decompose(n: int):
48     # Inicializando a maior potência de 2 que n pode ter como zero
49     r = 0
50     # Enquanto n tiver potências de 2
51     while n % 2 == 0:
52         # Incremente o valor de maior potência
53         r += 1
54         # Pegue o resto da divisão por 2
55         n = n / 2
56     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.

```
1  from random import randint
2
3  def fermat_test(n: int, k: int = 100) -> bool:
4      # Nenhum numero menor que 2 eh primo
5      if n < 2:
6          return False
7
8      # Casos base
9      if n in [2,3]:
10         return True
11
12     # Numeros pares nao sao primos
13     if n % 2 == 0:
14         return False
15
16     for _ in range(k):
17         # Pegue um valor entre [2,...,n-2]
18         a = randint(2, n - 2)
19
20         # a^(n-1) % n != 1 => Eh composto
21         if pow(a, n - 1, n) != 1:
22             return False
23
24     # Provavelmente eh um primo
25     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.

```

1  from typing import Dict, List
2  from fermat import fermat_test
3  import time
4  from util import find_number_of_bits
5  from lcg import LCG
6  from bbs import BBS
7
8  solicited_bits = [40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096]
9  seed = int(time.time())
10 lcg = LCG(seed=seed)
11
12 # Quantidade de numeros primos a serem gerados para cada quantidade de bits
13 quantity_of_numbers = 10
14
15 # Dicionario contendo os numeros gerados
16 possible_primes : Dict[int, List[int]] = {}
17 # Inicializando a lista
18 for numBits in solicited_bits:
19     possible_primes[numBits] = []
20
21 with open("./generate_prime.txt", "w") as writer:
22     writer.write("==== LCG ==== \n")
23     for numBits in solicited_bits:
24         # Colocando o valor do parametro m
25         lcg._m_(numBits)
26
27         avg_time = 0.0
28
29         writer.write(f"Começando a gerar números primos de {numBits} bits \n")
30         for _ in range(quantity_of_numbers):
31
32             start = time.time()
33
34             value = lcg.next()
35
36             # Enquanto o resultado obtido nao for primo, continue gerando
37             while not fermat_test(value):
38                 value = lcg.next(prime=True)
39
40             final = time.time()
41
42             # Pega a quantidade de bits que a saida gerou
43             bits = find_number_of_bits(value)
44
45             # Verifica se a quantidade de bits foi realmente atendida
46             assert(numBits == bits)
47
48             time_to_generate = final - start
49
50             avg_time += (time_to_generate / quantity_of_numbers)
51
52             # Adiciona o valor a lista
53             possible_primes[numBits].append(value)
54
55             # writer.write(f"bits = {bits} => {time_to_generate} segundos \n")
56 writer.write(f"0 tempo médio para gerar um número primo com {numBits} bits foi {avg_time} segundos \n")

```

Listing 9: Gerando números primos - parte 1

```

1  # Continuacao...
2      bbs = BBS(seed=seed)
3
4      writer.write("==== BBS ====\\n")
5      for numBits in solicited_bits:
6
7          avg_time = 0.0
8
9          writer.write(f"Começando a gerar números primos de {numBits} bits\\n")
10         for _ in range(quantity_of_numbers):
11
12             start = time.time()
13
14             value = bbs.generate(num_bits=numBits)
15
16             # Enquanto o resultado obtido nao for primo, continue gerando
17             while not fermat_test(value):
18                 value = bbs.generate(num_bits=numBits, prime=True)
19
20             final = time.time()
21
22             # Pega a quantidade de bits que a saida gerou
23             bits = find_number_of_bits(value)
24
25             # Verifica se a quantidade de bits foi realmente atendida
26             assert(numBits == bits)
27
28             time_to_generate = final - start
29
30             avg_time += (time_to_generate / quantity_of_numbers)
31
32             # Adiciona o valor a lista
33             possible_primes[numBits].append(value)
34
35             # writer.write(f"bits = {bits} => {time_to_generate} segundos\\n")
36         writer.write(f"0 tempo médio para gerar um número primo com {numBits} bits foi {avg_time} segundos\\n")
37
38     # Escreve no arquivo todos os valores gerados
39     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.

Quantidade de bits	LCG		Variação LCG	BBS		Variação BBS
	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

```

40: 909387089917
56: 70659298599739771
80: 106483221843458079573719
128: 231913758699542723555791727061281728529
168: 351766323135583412980439372085691343938673421550807
224: 149296429696816398347104396707666615259555606053329112313738929813
256: 97423054135301863196365406697658346878600292680225502663620155821285247301741
512: 12331509840835909021212864847868129782525983194087669633361388741356749213509828316778354800762348074527526193375727629191795946896726361727903651063865153
1024:
126811860210695178146136447321493033743255337218314225504127095976129559614436629130294104216130952380620395096758007691788223564047212640390150268883457970142788223279
5409087238487523007720303747013105686503745432754200406510201083944781247738327435272439231544291332889197049951500614384996840831717012841
2048:
1723098422914183457835837404104109057336895497820413109278025334734992749558087600351328825346259891857146367863507526716165465579414357481749713303871355370674108596263
9936645616974112729454209722782865824126641070381966686985101860990464869575214203387513662471787520945945268686329934264418163854242683978575864278461155200165915585919
536834822347306264661115841554277599357947690464712068712789961478677531342228792473852190487734557457015523084563886063425012454265288517439305607962299286413484612750
65923049870915892156579044098087601784282628070823824536743756152193854268901840661042645998219521322764355813
4096:
90741649801987216123181372545048821848414145036683660980775896147202923021061447598908169759793797272078614052796298708851512298036443803066430829229230760594205361915138
9457775902127303719304306447764750381856030511010636716210994450234434404230031804668574022641987707653047339601372385049831553331071749600732086044246984797910721478313
8170721695214051007694802655382054541843117901922600523739685967820804062243230627683033203803356542536903282627669953502851718111761090310624337835481212853788766274122
3757020952169377685367804430933222309431377289183665481600583108915935597045876714049093140076507504270155211536962517237268778123694410915191862444196675484172029903
7664239556798774922475404433091811431969150281398156467697986380149674878719941380213605984776915716112581495338498504406405786083003926207503185314214364620753036119138
7344751781216861840142509136820306199242915870536922649614901212504766776638805011059243305857123205909296387874831295917557272760252854351285585017903772958313988434942
74375214299288232118471613603834628445646455459591039373030388792485734686419185462634967220153058179105507612009916546679621998537438858767956078161627008390247510354
31137047491685381111717536491822511370485483080153

```

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].