UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO

LUAN DINIZ MORAES

Trabalho Individual sobre Números Primos

Florianópolis, 2024

1. Introdução

A implementação deste trabalho está disponível no Github:

https://github.com/Luan-Diniz/RandomAndPrimality

O trabalho foi desenvolvido utilizando a linguagem de programação Python. Essa decisão foi feita visando a simplicidade e a maior facilidade para implementar algoritmos. Naturalmente, essa decisão impacta a performance dos algoritmos. Durante a execução do trabalho ficou claro que a escolha de uma linguagem como Julia seria mais adequada.

2. Algoritmos

2.1 Linear Congruential Generator

É um gerador de números pseudo aleatórios, definido pela relação de recorrência:

$$X_{n+1} = (aX_n + c) \bmod m$$

Os parâmetros escolhidos para os testes mais à frente foram:

m = 2^{32},

a = 1664525,

c = 1013904223

Note que com esse *m*, a saída será na ordem de 32 bits.

Quanto ao cálculo do período, eles diferem em relação ao caso. O aqui exposto se enquadra em *m potência de 2 e c* ≠ 0 [11]. Nesse caso, podemos afirmar que o período é de 2^{32} devido ao Teorema de Hull-Dobell [11][12], o qual diz, sobre o caso acima, que, se *c* e *a* são coprimos, *a - 1* é divisível por todos os fatores de *m* e que *a - 1* tem que ser divisível por 4 caso *m* também o seja, então o período desse gerador é igual a *m*.

Os parâmetros escolhidos respeitam todos esses casos, note que *m* é divisível por 4, seu único fator é 2 e o maior divisor comum entre *a* e *c* é 1, ou seja, são coprimos.

```
def generateNumber(self) -> int:
    self.__Xn = (self.__a * self.__Xn + self.__c) % self.__m

    return self.__Xn
```

O método implementado que gera o número aleatório.

2.2 Blum Blum Shub

Também é um gerador de números pseudo aleatórios, é baseado em uma função *oneway*[13], que é basicamente uma função que o resultado é fácil de calcular, porém difícil de calcular o caminho reverso a partir do resultado. A relação de recorrência desse método é definida por:

$$x_{n+1} = x^2_n \bmod M,$$

Seus parâmetros são x_0 (a seed) e *p* e *q*, dois números primos congruentes a *3 mod 4* (isso garante que cada resíduo quadrático tenha uma raiz que também é um resíduo quadrático). M = pq.

"Nós dizemos que *a* é um resíduo quadrático se existe algum x tal que $x^2 = a$" [14]. Esse conceito é uma das bases do Blum Blum Shub. Além disso, os primos *p* e *q* devem ser *safe primes* [13]. Um *safe prime* nada mais é que um número primo associado a um subconjunto de primos que, se multiplicados por 2 e somados por 1, resultam em um primo (esse primo resultante é o *safe prime*) [15].

De forma mais simplificada, um número primo *k* é chamado de um *Sophie Germain prime* se *2k + 1* também for um primo. Seja *w = 2k+1* anteriormente citado, *w* é chamado de *safe prime.*

Além disso, é possível calcular x_i arbitrário com a seguinte fórmula[13]:

$$x_i = \left(x_0^{2^i \bmod \lambda(M)}\right) \bmod M$$

Sendo que λ se refere à função de Carmichael e nesse caso, terá o valor de *(p-1)(q-1)*.

```
def generateNumber(self) -> int:
    self.__Xn = self.__Xn**2 % self.__M

    return self.__Xn
```

O método implementado que gera o número aleatório.

2.3 Fermat Primality Test

Esse algoritmo é baseado no Pequeno Teorema de Fermat, onde:

- Se *n* e *a* pertencem aos naturais,
- Se *n* é primo e *mdc(a,n) = 1*

então:

$$a^{n-1} \equiv 1 \bmod n \qquad (1)$$

Se a sentença acima for verdadeira, é muito provável de *n* ser um número primo, embora não necessariamente, e é nisso em que o Teste de Primalidade de Fermat se baseia (na conversão do Pequeno Teorema de Fermat).

Portanto, estamos falando de um algoritmo probabilístico, que pode falsamente identificar um número composto como primo.

Bases *a* nas quais

$$a^n \not\equiv a \bmod n \qquad (2)$$

são chamadas de *witnesses* (testemunhas) para *n*. Pois, ao verificar o cálculo acima com uma *witness* de *n,* provou-se que *n* trata-se de um número composto.

A maioria dos números compostos possui várias *witnesses*, pseudoprimos possuem menos, sendo que alguns, como os *Carmichael Numbers*, não possuem nenhuma *witness*. Números primos também não possuem *witnesses*. [1]

A implementação desse algoritmo segue na imagem abaixo:

```
if (n % 2 == 0):
    return "Composite"

for _ in range(0, number_of_rounds):
    a = randint(2, n-2)
    if(pow(a,(n-1),n) != 1):        # (a**(n-1) % n != 1)
        return "Composite"
return "Probably prime"
```

Basicamente são várias rodadas em que escolhe-se uma base *a* aleatória, calcula-se *(2)* na esperança de achar uma *witness* para *n* e assim comprovar que ele é composto.

Naturalmente, há uma chance de falha, visto que *a* é escolhida aleatoriamente. Os *Carmichael Numbers*, já citados, tendem a passar esse teste uma grande quantidade de vezes, embora compostos, pois eles só falham a menos que o *a* escolhido seja um dos fatores de *n*. [2]

No algoritmo é verificado $a^{n-1} \not\equiv 1 \bmod n$ e o conceito de *witness* varia um pouco (A referência [2] a utiliza essa forma para as definições por exemplo). Se por acaso no teste fosse feito *if(pow(a,n,n) != a)*, ou seja, verificando *(2)* de fato, então os *Carmichael Numbers* enganariam o algoritmo em absolutamente todas as vezes.

```
Fermat Pseudoprimes : times it passed the Fermat Primality Test test.
{561: 1000,
 1105: 1000,
 2465: 1000,
 6601: 1000,
 8911: 1000,
 10585: 1000,
 11972017: 1000,
 67902031: 1000,
 1208361237478669: 1000,
 844154128953833755776750022681: 1000,
 365376903642671522645639268043801: 1000,
 349407515342287435050603204719587201: 1000,
 135130930814893808401886512466750320670111400792O1: 1000,
 260849323075371835669784094383812120359260783810157225730623388382401: 1000}
```

Nota: teste realizado com *Carmichael Numbers,* cada número foi testado 1000 vezes no Teste de Primalidade de Fermat verificando $a^n \not\equiv a \bmod n$ ao invés de $a^{n-1} \not\equiv 1 \bmod n$, com 13 rodadas cada iteração. Em absolutamente todas as iterações o algoritmo respondeu que os números em questão eram provavelmente primos.

2.4 Miller-Rabin

O algoritmo de Miller-Rabin é baseado no seguinte teorema [2]:

"Supondo que n é um primo ímpar, e $n-1 = 2^s t$, sendo t ímpar. Se a não é divisível por n então:

ou $a^t \equiv 1 \pmod n$

ou $a^{2^i t} \equiv -1 \pmod n$ para algum i tal que $0 \leq i \leq s - 1$"

E também se baseia no seguinte resultado : *"se n é primo, então as únicas raízes quadradas de 1 mod n são 1 e −1".* [9]

As bases *a* que, com *n*, falham os resultados acima são chamadas *witnesses.* Aqui, diferentemente do algoritmo anterior, existem resultados mostrando que a quantidade de *witnesses* é de pelo menos $¾(n - 1)$. A partir disso é possível calcular a precisão com que Miller-Rabin identifica um número composto $(1 - ¼)^k$. [2]

Ou seja, nesse contexto não existem números como os *Carmichael Numbers* , os quais são um problema no Teste de Primalidade de Fermat.

A implementação seguiu a mostrada na Wikipédia [9]:

```
s, d = MillerRabin.__findSandD(n)

for _ in range(0, number_of_rounds):
    a = randint(2, n-2)
    x = pow(a,d,n)
    for _ in range(0, s):
        y = (x ** 2) % n
        if y == 1 and x != 1 and x != n-1:
            return "Composite"
        x = y
    if y != 1:
        return "Composite"
return "Probably prime"
```

No Teste de Primalidade de Fermat, um número par poderia ser testado e, caso não houver uma verificação no começo da execução para retornar *Composite* se esse for o caso, ele poderia passar no teste caso as bases escolhidas não forem *witnesses.* O enunciado do algoritmo de Miller-Rabin por outro lado, simplifica essa questão, dizendo que somente números ímpares servem de entrada para o algoritmo.

3. Estrutura do Projeto

No diretório *cryptolibrary/* estão os módulos com a implementação dos algoritmos já citados e com os testes de unidade.

Em *outputs/* estão os arquivos com o resultado dos testes da *main.py* e também o probability_carmichael_numbers.py, arquivo feito para investigar o resultado de alguns resultados que serão discutidos mais para frente.

Já no diretório raiz do trabalho, está a *main.py* com os testes realizados com os algoritmos, o *utils.py*, que cria as funções necessárias para encontrar um número aleatório com quantidade específica de bits e também para encontrar números primos.

4. Testes

Foram criados testes unitários simples para os 4 algoritmos implementados. Além disso, no *main.py,* encontra-se o teste executado para medir tempo de execução da geração de números aleatórios e de números primos, cujos resultados foram organizados em tabelas que serão apresentadas mais adiante.

Para executar testes unitários, basta executar o arquivo .py relativo ao teste dos algoritmos. Já para os outros testes, basta executar a main.py, porém a execução deles pode demorar bastante tempo (horas).

4.1 Carmichael Numbers

Alguns *Carmichael Numbers* foram escolhidos[3][4][5][6] para os testes unitários do Miller-Rabin e do Teste de Primalidade de Fermat. A ideia inicialmente era apenas comparar as taxas com que eles enganavam os testes.

Porém, a princípio parecia que quanto maior o *Carmichael Number* era, maior a facilidade ele teria em passar no Teste de Primalidade de Fermat. Conjecturou-se então que o crescimento do número de divisores de um número não acompanhava o crescimento do mesmo (como já citado, os *Carmichael Numbers* não passam no teste quando a base escolhida aleatoriamente é um fator deles), ou seja, um limite de crescimento assintótico menor que o linear.

Procurou-se então resultados que fortalecessem ou comprovassem essa conjectura com algum sucesso[7][8].

Decidiu-se então procurar mais *Carmichael Numbers* e adicioná-los ao teste, porém o resultado não foi o esperado:

```
Fermat Pseudoprimes : times it passed the Fermat Primality Test test.
{561: 585,
 1105: 699,
 2465: 727,
 6601: 803,
 8911: 796,
 10585: 795,
 11972017: 978,
 67902031: 973,
 1208361237478669: 978,
 844154128953833755776750022681: 978,
 365376903642671522645639268043801: 981,
 392000251605356793349050844538065236557716721692385776886401: 865,
 270644058193296027005955632086513529954302748834156406194893727505922295661037223068979868653329911238895996329920l: 788}
```

Resultados após aplicar o algoritmo 1000 vezes para cada *Carmichael Number*, com 1 rodada..

```
Fermat Pseudoprimes : times it passed the Fermat Primality Test test.
{561: 56,
 1105: 169,
 2465: 217,
 6601: 324,
 8911: 309,
 10585: 280,
 11972017: 847,
 67902031: 882,
 1208361237478669: 905,
 844154128953833755776750022681: 929,
 365376903642671522645639268043801: 929,
 392000251605356793349050844538065236557716721692385776886401: 490,
 270644058193296027005955632086513529954302748834156406194893727505922295661037223068979868653329911238895996329920l: 258}
```

Resultados após aplicar o algoritmo 1000 vezes para cada *Carmichael Number*, com 5 rodadas.

No início dos testes a ideia de quanto maior o *Carmichael Number* maior a chance dele passar pelo teste parecia se concretizar, porém em específico os últimos dois números foram contra exemplos bem impactantes.

Decidiu-se então tentar calcular a probabilidade com que cada um desses números passaria pelo teste.

Primeiro, é necessário calcular o número de divisores deles.

- Se $n = p^{a}_{1} \ldots p^{a}_{k}$ (Considere que os expoente são a_1 até a_k),

então, o número de divisores de n (d(n)) é :

- $d(n) = (a_1 + 1) * \ldots * (a_k + 1)$ [8]

Porém, os *Carmichael Numbers*, são produtos de primos distintos e são *square-free* (inteiro sem fator quadrático) [1], portanto:

- $d(n) = (2)^{NF(n)}$, sendo *NF(n)* o número de fatores de n.

Criou-se um novo arquivo com um teste que possui uma lista com todos os *Carmichael Numbers* usados fatorados. A ideia é dividir o número de divisores de n pela quantidade de números na faixa de 2 a n - 2, que é usada para gerar a base aleatoriamente para os testes.

```python
def number_dividers_carmichael(list_factor):
    return (1 + 1)**len(list_factor)

for carmichael_factors in carmichael_factors_list:
    carmichael = 1
    for factor in carmichael_factors:
        carmichael *= factor

    number_dividers = number_dividers_carmichael(carmichael_factors)
    print(f"Number: {carmichael}")

    # number_dividers - 2 because we don't want 1 neither n.
    # ((carmichael - 2) - 2) is the number of numbers in randint(2, carmichael-2)
    print(f"Probability of not passing the test: {(number_dividers - 2)/((carmichael - 2) - 2)}", end = "")
    print(f"\t (Number of dividers: {number_dividers})")
```

O teste criado.

```
Number: 561
Probability of not passing the test: 0.010771992818671455       (Number of dividers: 8)
Number: 1105
Probability of not passing the test: 0.005449591280653951       (Number of dividers: 8)
Number: 2465
Probability of not passing the test: 0.002438033319788704       (Number of dividers: 8)
Number: 6601
Probability of not passing the test: 0.0009095043201455207      (Number of dividers: 8)
Number: 8911
Probability of not passing the test: 0.0006736274840013472      (Number of dividers: 8)
Number: 10585
Probability of not passing the test: 0.0005670541536716756      (Number of dividers: 8)
Number: 11972017
Probability of not passing the test: 5.01168851052868e-07       (Number of dividers: 8)
Number: 67902031
Probability of not passing the test: 8.836260513990253e-08      (Number of dividers: 8)
Number: 1208361237478669
Probability of not passing the test: 1.1585939341460533e-14     (Number of dividers: 16)
Number: 844154128953833755776750022681
Probability of not passing the test: 3.5538533747597984e-29     (Number of dividers: 32)
Number: 365376903642671522645639268043801
Probability of not passing the test: 8.21069960933797e-32       (Number of dividers: 32)
Number: 392000251605356793349050844538065236557716721692385776886401
Probability of not passing the test: 2.1399491366768725e-53     (Number of dividers: 8388608)
Number: 270644058193296027005955632086513529954302748834156406194893727505922295661037223068979686853329911238895963299201
Probability of not passing the test: 1.6250297680508774e-102    (Number of dividers: 4398046511104)
```

O output do teste.

O output deste teste também não condiz com o que foi pensado e nem com a quantidade de vezes que esses números passaram pelo teste. Ou seja, definitivamente algo está incorreto na linha de raciocínio construída até aqui..

A investigação a respeito desses resultados poderia continuar, modificando o algoritmo de primalidade de Fermat para printar as bases com que ele avalia o número de entrada por exemplo. Porém, creio que neste ponto já ultrapassei o escopo do trabalho proposto.

4.2 Comparação Carmichael Numbers entre Fermat e Miller-Rabin

Nos testes unitários são rodadas 1000 iterações com cada *Carmichael Number* listado, para ver quantas vezes os números passam pelos testes como prováveis primos. Em ambos os casos, foram rodados com 1 rodada nos algoritmos.

```
Fermat Pseudoprimes : times it passed the Fermat Primality Test test.
{561: 585,
 1105: 690,
 2465: 731,
 6601: 808,
 8911: 807,
 10585: 746,
 11972017: 967,
 67902031: 967,
 1208361237478669: 976,
 844154128953833755776750022681: 986,
 365376903642671522645639268043801: 989,
 392000251605356793349050844538065236557716721692385776886401: 861,
 270644058193296027005955632086513529954302748834156406194893727505922295661037223068979868653329911238895996329920 1: 748}
```

Output do test_fermat_primality_test.py

```
Carmichael number : times it passed the Miller-Rabin test.
{561: 14,
 1105: 29,
 2465: 33,
 6601: 52,
 8911: 179,
 10585: 64,
 11972017: 30,
 67902031: 256,
 1208361237478669: 18,
 844154128953833755776750022681: 4,
 365376903642671522645639268043801: 6,
 392000251605356793349050844538065236557716721692385776886401: 0,
 270644058193296027005955632086513529954302748834156406194893727505922295661037223068979868653329911238895996329920 1: 0}
```

Output do test_ miller_rabin.py

É perceptível como o Miller-Rabin possui mais precisão, isso é realçado pois não existe um conjunto de números no Miller-Rabin que seja análogo aos *Carmichael Numbers* no Teste de Primalidade de Fermat.

4.3 Tabelas Geradas

Na *main.py*, estão os testes que geraram esse resultado. Foram feitas tentativas de paralelização utilizando o módulo *multiprocessing* do Python, porém com insucesso.

De seed para o Blum Blum Shub e para o Linear Congruential Generator foi usado um número aleatório inteiro entre 0 e 1000000 (inclusive). Quanto aos parâmetros, para o primeiro foram utilizados dois números primos congruentes a *3 mod 4*, os quais são 3263052707 e 5847777359 [10]. Já para o segundo, utilizou-se m = 2^{32}, a = 1664525 e c = 1013904223 [11].

A *main.py* utiliza as funções definidas em *utils.py* para calcular um número aleatório com tamanho específico de bits (para isso considerou-se o bit mais significativo sendo um) e para encontrar um primo com número específico de bits (que vai gerando números aleatórios com o tamanho específico até que ele passe em um dos testes de primalidade implementado, com 20 rodadas por padrão.).

Nas tabelas a seguir as seguintes siglas foram adotadas:

- LCG = Linear Congruential Generator
- BBS = Blum Blum Shub
- FPT = Fermat Primality Test
- MR = Miller-Rabin
- A = Ambiente de testes 1
- B = Ambiente de testes 2
- TMA = Tempo médio medido no ambiente A
- TMB = Tempo médio medido no ambiente B

Os testes foram realizados em dois ambientes diferentes:

- A = Ambiente com processador Intel i39100f, Windows10 e Python 3.10.10.
- B = Ambiente com processador Intel i511300H, Linux Mint e Python 3.10.12.

No ambiente A, o teste demorou entre 4 e 4.5 horas para concluir, enquanto no ambiente B a execução levou um pouco menos de 2.5 horas.

A tabela a seguir possui alguns números pseudo aleatórios gerados:

Nota: O processo foi tão rápido que a função *time.time()* do Python não conseguiu registrar nada, portanto também não é relevante comparar o resultado entre processadores.

Algoritmo	**Tamanho (bits)**	**Tempo (s)**	**Número Gerado**
LCG	40	0.0	553636930982
LCG	56	0.0	64628866870957814
LCG	80	0.0	1004592810389924125524972
LCG	128	0.0	298911456437518989244913902290238171252
LCG	168	0.0	3162152112569147799341597047984518810068993400309 87
LCG	224	0.0	17004652790190763581310561488807650263620105081421385859468389832228
LCG	256	0.0	937819473332855253538060868944539631181883513312883457079834994259719059 44959
LCG	512	0.0	671349101026586672465454990651185136473635550672613317470929366574615708436169047917119471242994769217848912373156601218525631455413136057688682431206186
LCG	1024	0.0	101696212319806699726757301809509025165615707553321020700111624688593126159824065733546878395591812156743079059613856267 68

			5159343007426375337794345220417224313612405324044797281887933 9085469980820864430871319811937726662507584124782056865483276 4029724393790253010623474654876530532534105071819405333222665 4604
LCG	2048	0.0	3076445082354625477758241207870090905433498453972521760224922 6996821496520443021992395798654841908013809959496401942522708 5350624866498384607684527224733917364994908819442917345024099 4823400995496758991640665395376086579711559823613335044926578 4266049726124305917801793853846393426824881926250377184004598 2874098895557022576981624947904786355652638736108944797077626 8935477615418291441288755441990535964906931733936638730915167 4300889254550622661465790393746151906945811368157762661025904 4054278323954772409802640520451208608730374617722834001013209 4762040692916628716697690330802441843968167651947058036660530 9541769
LCG	4096	0.0	9188853351196462989781186964418826409948218581110187250457997 1001978616528847748341181265682066976756320965224274270000578 4009127587393049592720402070462257928312755958688392506113456 1148040764816971861929066119269373860504343293168171800698676 2087877308257167982232333120533401121346904423357888996406046 7892892880631393936847987558212568486116700495867740312183505 9642580048912833608998833814916630107769618636361819962224093 4349450626684486583053865778779870578578292491863912108391570 2395158579587910164527913134685739668961387071660996517898353 8748080424105546102088421174727032460685847042896695151847272 3391468349481452271829213755897924758934945369856412002267594 8844665379973988300098270529358619346908396936885135556840536 7564145333200268236097645637039820327204637417865188239028091 0391813789322823752175941695262076188401392826644019808855650 0902110523359632929894545597515071230732528212446912040212196 9105470623161436867292097190811127514282731705883857514091425 1714875077997839838208554450147830962141919938770778929726818 6750265476784404206948226491644000920816185250343468587117542 9175654369856384927702987261259794685239111626974920053051795 9383183197540332062048115037455055147294297818236537653754868 9439572423467
BBS	40	0.0	549755817635
BBS	56	0.0	36028797346157748
BBS	80	0.0	604462910133351698176236
BBS	128	0.0	170141183460901257576947682806473793664
BBS	168	0.0	187072216349818326276998352094950920882280755552169
BBS	224	0.0	134799739297819351826191771000858377189135740333910880774139013 11358
BBS	256	0.0	57896044700622001433457496169116927349554419394790444334476044 242622083164332
BBS	512	0.0	6703904039195347080162057385514897991482490983542391096692250 2976440397176038657732354707747831455491387211046258506627665 3053490261726682204640624350643 7
BBS	1024	0.0	8988465740947960034422542591661824076279045999457472093109811 2094631943290925537753804654807019469345214135873438494293272 6747346274939721101232615114119486050894489518917569703242350 5367989529607164164650002002380562293122174222459072365784713 9922999803664462781812680984984573517511152124143823856852715

			153
BBS	2048	0.0	323170076935210857939959049352774641951021106681344153100209121345217789870953493334458809425804428533685705006869093442348148315862771026549046697440471972141361854499384117203331805522176259515299086435209280132140185927382298011939171921562142911676932470880001596248937884540875280561815153849664612552566957412758883632969573007376260677879907078451704541274757753662108259792146089409159866735364654021708479331663128136757252326884591143350218367391561990876723350490139801477964217338936676219338439640948318268149040304580616525333224896140795006726340460072291074302956577074109582478852166789172400969814195007769352108579399590493527746419510211066813441531002091213452177898709534933344588094258044285336857050068690934423481483158627710265490466974404719721413618544993841172033318055221762595152990864352092801321401859273822980119391719215621429116769324708800015962489378845408752805618151538496646125525669574127588836329695730073762606778799070784517045412742757753662108259792146089409159866735364654021708479331663128136757252326884591143350218367391561990876723350490139801477964217338936676219338439640948318268149040304580616525333224896140795006726340460072291074302956577074109582478852166789172400969814195007769352108579399590493527746419510211066813441531002091213452177898709534933344588094258044285336857050068690934423481483158627710265490466974404719721413618544993841172033318055221762595152990864352092801321401859273822980119391719215621429116769324708800015962489378845408752805618151538496646125525669574127588836329695730073762606778799070784517045412742757753662108259792146089409159866735364654021708479331663128136757252326884591143350218367391561990876723350490139801477964217338936676219338439640948318268149040304580616525333224896140795006726340460072291074302956577074109582478852166789172400969814188888777812469272242777233623801678657820649037840090653162449436966399331148161135142731047045367064614258741713070726265956944649006213197085640344892150321101373456509226358929850439347351093828092943690193542555424156755899019205503610432194059671229074322054353152001367026576604900354021617038337900516131629696306360976172308287241571764150455855629119502575442822639281562584178646924375553569753389552064933301050855514421007154611851012187604893772478800566045370763741857276922688656550001703726826377256351465865020686666628102193690075092753766085139890640673574680355701185058670043177056758119792396705805241077194354570340952499612313484165936177473656288757821145922708523736210811473365291764911006623563129412155592098861545804586801708531466586115525172042715761309652838508625792707351999874870151359651867582005718931840191051458190929580335908131143833626186077779755093871072898667856758119848684024697086348403108025102954367728125722518061876545008024578714425598801969390126316001499922248374500721937242766462679501269368934434308096756573556532370834356015877277499238907658445823307302137456105106191799590227656111128816140490269197231354367683045028867888589474318596672782759714700461423001134711548771
BBS	4096	0.0	208877781246927224277723362380167865782064903784009065316244943696639933114816113514273104704536706461425874171307072626595694464900621319708564034489215032110137345650922635892985043934735109382809294369019354255542415675589901920550361043219405967122907432205435315200136702657660490035402161703833790051613162969630636097617230828724157176415045585562911950257544282263928156258417864692437555356975338955206493330105085551442100715461185101218760489377247880056604537076374185727692268865655000170372682637725635146586502068666662810219369007509275376608513989064067357468035570118505867004317705675811979239670580524107719435457034095249961231348416593617747365628875782114592270852373621081147336529176491100662356312941215559209886154580458680170853146658611552517204271576130965283850862579270735199987487015135965186758200571893184019105145819092958033590813114383362618607777975509387107289866785675811984868402469708634840310802510295436772812572251806187654500802457871442559880196939012631600149992224837450072193724276646267950126936893443430809675657355653237083435601587727749923890765844582330730213745610510619179959022765611112881614049026919723135436768304502886788858947431859667278275971470046142300113471154877

A seguir, estão dispostos um número primo gerado para cada tamanho:

Tamanho (bits)	Primo Gerado
40	675547841851
56	43310412621672257
80	766305076827328597097837
128	245229093866957615730514748706106737211
168	31459500470162287437959087014099408435322871902650 7
224	241048402048582573773505041217163320245979201711561107075702019128 17
256	771499606569849868442830008124938611781568570914978909455110093575318863890 99
512	111556396799214685773181836940263475203013084729165758584454600195395345790 109974456001773890279355611470094826572893473734793316490970320113632766341 39209
1024	138828206594885133587233244729252154819208455356635120551608780419211071619 374480408407085414365526769516515218429869942241698581576807295681243959851 323830015291587295217891920163912667021235583435901351043391861590983686434 5865237260764846199378585219127301831580034145610180143825953387893207779 4672111003
2048	217103150812017687798229475739181789151937125707312197352203588936001377625 165544761327226218062208981335489418052341686427057159764729682949501862634 234180338192903543488076354912798635844204474743734971004030958470973159732 933485566985296430276253711089708778883860275553977393695097938478155864448 406013235383265422576657947926087278713377515166734956976125464718989389557 505040348642226914137977186331410947560420326549924211151111476034936839305 896659005386837741900853116345538435550927302314918306898305040704520286688 461662248203905042896587667018433617476909720984331318599559003119430470586 309691867924571
4096	873282623580606909590356034856176057468809609690447308303138778869197879629 387438871324712660808847218626965972489358917485066120598160449573953246529 191659611481176528138954173089252737058323513012868192808204727188076972044 7 149700674809710632632096715893392007919618128075607017515229137198268697559 674559035860484522334197732523634691161858663639573129486771003051472892052 8 156381617769817859750268768891095682706936251888613797301001562074592914686 290603798582265615709689986968799606573337519882023201249708229187369926067 016946961112490251696294378603253444043058335451778434229060713597357092933 4 003255255036728576068967667520269081405694294546482518527354950894703775861 212643754361809867341714899439215864175686097951548307612521823425193626577 252735325249845886953396688059134327131192873210090020810908234346721176060 3 407665909635964005264149359351978873900728451702616824855654436787859646707 241452501189593454477147899439851831283385380708647710399791441468934914431 0 101076246415831798039285681363113468421896993206972506100801256607608564470 0 705571263590126719407081347667613719200645804482021609057725372102179303473 704973287400810387561558390819080973583003323745270439105827622142329273278 382977645550023161776821851

A próxima tabela compara o desempenho dos algoritmos (usou-se 20 rodadas em ambos por padrão). Para cada tamanho foram gerados 25 números primos.

Nota: Para a geração de números aleatórios utilizou-se o Linear Congruential Generator já criado anteriormente para outros testes. Como a diferença no tempo de execução é baixa, considerou-se que o gerador tinha pouca influência, logo não sendo necessário repetir os testes com o Blum Blum Shub.

Tamanho (bits)	TMA(s) com FPT	TMB(s) com FPT	TMA(s) com MR	TMB(s) com MR
1024	1.5497776317596434	0.8814673614501953	1.3986224842071533	1.0814462947845458
2048	16.057759704589845	9.88320538520813	19.497605276107787	16.081907243728637
4096	243.59869854927064	114.67833729743957	266.3586693382263	199.33737391471863

É perceptível o melhor desempenho no ambiente B, devido à diferença de processador. Além disso, em quase todos os casos o tempo médio do Teste de Primalidade de Fermat foi mais eficiente do que o tempo médio do Miller-Rabin.

Esse resultado faz sentido, já que o Miller-Rabin é mais robusto (e possui melhor precisão) do que o Teste de Primalidade de Fermat.

5. Complexidade

A complexidade para gerar um número com o LCG ou BBS é de $O(1)$.

Considerando que w é o número de rodadas escolhido para um algoritmo rodar, o Teste de Primalidade de Fermat tem complexidade $O(w)$, enquanto o Miller-Rabin é $O(ws)$. Lembrando que o s vem da exponenciação da forma $n - 1 = 2^s d$, sendo n o número que está sendo testado no algoritmo.

Nenhuma tentativa de melhorar essas complexidades foi feita.

Referências

[1]: Riemer, Emily. "Pseudoprimes and Carmichael Numbers." *MATH0420* (2016).

[2]: Pun, Nicholas. "Survey of Primality Testing Methods."

[3]: Pinch, RICHARD GE. "The Carmichael numbers up to 10^{15}." *Mathematics of Computation* 61.203 (1993): 381-391.

[4]: Pinch, Richard GE. "The Carmichael numbers up to $10^{17} $." *arXiv preprint math/0504119* (2005).

[5]: Howe, Everett. "Higher-order Carmichael numbers." *Mathematics of computation* 69.232 (2000): 1711-1719.

[6]: Pinch, Richard GE. "Carmichael numbers with small index."

[7]: Hakobyan, Tigran. "On an asymptotic behavior of the divisor function $\tau (n) $." *arXiv preprint arXiv:1406.3698* (2014).

[8]: https://terrytao.wordpress.com/2008/09/23/the-divisor-bound/ Acesso em 25/05/2024.

[9]: https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test Acesso em 26/05/24.

[10]: https://www.gkbrk.com/blum-blum-shub Acesso em 24/05/2024.

[11]: https://en.wikipedia.org/wiki/Linear_congruential_generator Acesso em 23/05/2024.

[12]: https://www.howardrudd.net/mathematics/hull-and-dobells-first-theorem/ Acesso em 24/05/2024

[13]: https://en.wikipedia.org/wiki/Blum_Blum_Shub Acesso em 24/05/2024

[14]: https://crypto.stanford.edu/pbc/notes/numbertheory/qr.html Acesso em 25//05/2024

[15]: Zeolla, Gabriel Martin. "Argentest, primality test for Sophie Germain's prime numbers and safe prime." *Annals of Mathematics* 160.2: 781-793.