Número Primos

Trabalho 2 - Segurança em Computação (INE5429)

Universidade Federal de Santa Catarina

Número Primos

Trabalho 2 - Segurança em Computação (INE5429)

Universidade Federal de Santa Catarina

Felipe de Campos Santos

17200441

09/07/2021

# Objetivo

Este trabalho teve como objetivo a geração de números primos, de maneira pseudo-aleatória, nos tamanhos de 40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096 (expresso em bits). Para isso, utilizamos de geradores de números pseudo-aleatórios com os algoritmos:

* [Blum Blum Shub](https://pt.wikipedia.org/wiki/Blum_Blum_Shub);
* [Lehmer](https://en.wikipedia.org/wiki/Lehmer_random_number_generator) (também conhecido como Park-Miller)

Depois de gerado o número, fazemos um teste de primalidade nele, utilizando os algoritmos:

* Teste de primalidade de [Miller-Rabin](https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test);
* Teste de primalidade de [Fermat](https://en.wikipedia.org/wiki/Fermat_primality_test);

Foram escolhidos estes algoritmos pela facilidade de entendimento, e consequentemente de implementação, visto que seus algoritmos são mais sucintos e simples. As outras opções vistas pelo aluno envolviam algoritmos com mais passos, o que tornava o entendimento mais difícil e o código mais propício a erros.

Como a finalidade deste trabalho é conhecer alguns dos algoritmos e técnicas de geração de números pseudo-aleatórios e testes de primalidade e não a implementação do melhor e mais rápido gerador, a escolha foi baseada no que seria mais simples de entender e replicar.

Além disso, a implementação foi feita em python pela familiaridade do aluno e simplicidade da linguagem.

# Código

Os códigos e arquivos de resultados de testes estão disponibilizados no [github](https://github.com/felipecampossantos/INE5429/tree/master/t2);

# Geração de Números Pseudo-Aleatórios

Os dados de geração são exportados para um arquivo .csv, para facilitar a análise dos mesmos.

O arquivo usado para o teste pode ser encontrado no [github](https://github.com/felipecampossantos/INE5429/blob/master/t2/src/prgn_test.py)

Abaixo, a tabela resultado de uma rodada de geração no computador do aluno:

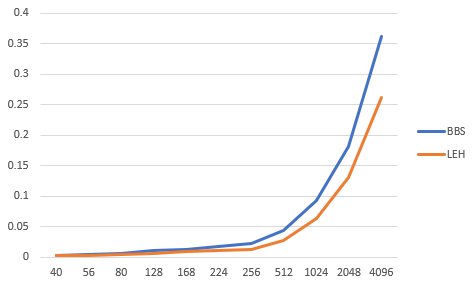
Blum Blum Shub

| BITS | TEMPO (s) |
| --- | --- |
| 40 | 0.002997 |
| 56 | 0.005001 |
| 80 | 0.006000 |
| 128 | 0.010999 |
| 168 | 0.013000 |
| 224 | 0.017037 |
| 256 | 0.023000 |
| 512 | 0.043995 |
| 1024 | 0.091991 |
| 2048 | 0.181040 |
| 4096 | 0.361042 |

Lehmer

| BITS | TEMPO (s) |
| --- | --- |
| 40 | 0.002000 |
| 56 | 0.002000 |
| 80 | 0.004000 |
| 128 | 0.006000 |
| 168 | 0.009002 |
| 224 | 0.009998 |
| 256 | 0.012000 |
| 512 | 0.027016 |
| 1024 | 0.063958 |
| 2048 | 0.129961 |
| 4096 | 0.261001 |

Colocando os resultados em um gráfico, podemos fazer uma comparação:



fonte: autor

Assim fica fácil ver que, apesar de pequena a diferença, o algoritmo Park-Miller (no gráfico representado com o nome LEH, de Lehmer) é mais rápido.

Vale lembrar que esses resultados podem variar, visto que o tempo levado para a geração é influenciado pelo computador em que está rodando.

Visto que o algoritmo de Blum Blum Shub pode ser resumido em

x(n+1) = x(n)² mod M, até que n+1 = k, onde k é o número de bits desejado

podemos dizer que sua complexidade é de O(k), onde k é o número de bits desejado. Olhando o código, fica mais claro:

| def algorithm(self):  self.num = pow(self.num, 2) % self.m  return self.num   def calculate(self):  result = ""  for \_ in range(self.size):  rnd = self.algorithm()  b = rnd % 2  result += str(b)  self.num = int(result, 2)  self.result = result  return result |
| --- |

O algoritmo roda “size” vezes.

O mesmo pode ser replicado para o algoritmo de Park-Miller

| def algorithm(self):  self.num = (self.a \* self.num) % self.m  return self.num   def calculate(self):  result = ""  for \_ in range(self.size):  rnd = self.algorithm()  b = rnd % 2  result += str(b)  self.num = int(result, 2)  self.result = result  return result |
| --- |

Como ambos tem complexidade linear e semelhante, podemos dizer que os resultados de tempo obtidos fazem sentido.

Vale ressaltar que, para Park-Miller, os parâmetros iniciais usados são os propostos pelos autores, segundo a [página na Wikipedia](https://en.wikipedia.org/wiki/Lehmer_random_number_generator#Parameters_in_common_use).

# Teste de Primalidade

Para os testes de primalidade, foram usados uma sequência pré-definida de números primos e não primos, com os mesmos tamanhos dos números gerados pseudo-aleatoriamente.

O arquivo usado para o teste pode ser encontrado no [github](https://github.com/felipecampossantos/INE5429/blob/master/t2/src/primes_test.py)

Os resultados de tempo, junto com o resultado esperado e obtido do teste, foram exportados para um arquivo .csv para facilitar a análise. Abaixo, os resultados médios de tempo para testes esperados verdadeiros e testes esperados falsos para os dois algoritmos

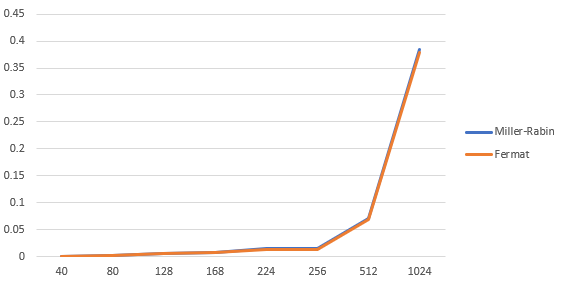
## Testes esperados verdadeiros

Miller-Rabin

| BITS | TEMPO (s) |
| --- | --- |
| 40 | 0.000501 |
| 80 | 0.001999 |
| 128 | 0.004999 |
| 168 | 0.007501 |
| 224 | 0.014500 |
| 256 | 0.014499 |
| 512 | 0.071000 |
| 1024 | 0.383999 |

Fermat

| BITS | TEMPO (s) |
| --- | --- |
| 40 | 0.000498 |
| 80 | 0.002004 |
| 128 | 0.004976 |
| 168 | 0.007519 |
| 224 | 0.013483 |
| 256 | 0.014024 |
| 512 | 0.068486 |
| 1024 | 0.378479 |



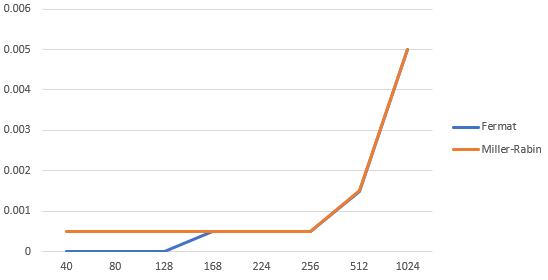
## Testes esperados falsos

Miller-Rabin

| BITS | TEMPO (s) |
| --- | --- |
| 40 | 0.0005 |
| 80 | 0.0005 |
| 128 | 0.0005 |
| 168 | 0.0005 |
| 224 | 0.0005 |
| 256 | 0.0005 |
| 512 | 0.0015 |
| 1024 | 0.005 |

Fermat

| BITS | TEMPO (s) |
| --- | --- |
| 40 | 0 |
| 80 | 0 |
| 128 | 0 |
| 168 | 0.000499 |
| 224 | 0.000499 |
| 256 | 0.000499 |
| 512 | 0.001499 |
| 1024 | 0.004999 |



fonte: autor

Apesar de extremamente próximos, para números grandes o algoritmo de Fermat se mostrou um pouco mais rápido, o que faz sentido analisando a complexidade dos algoritmos, apresentados na própria página da wikipedia, como segue:

* Fermat: O(k log2n), sendo *n* o número testado e *k* o número de vezes que o algoritmo é rodado para um valor ‘a’ aleatório.
* Miller-Rabin: O(k log3n), sendo *n* o número testado e *k* o número de vezes que o algoritmo é rodado.

# Resultados finais

Para os testes finais, o aluno escolheu os algoritmos:

* Lehmer / Park-Miller, para geração de números pseudo-aleatorios
* Fermat, para o teste de primalidade

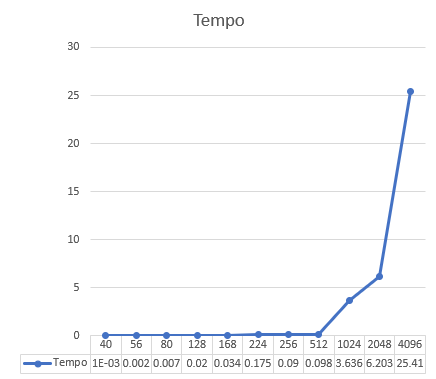
devido aos resultados de testes anteriores.

O arquivo usado para o teste final pode ser encontrado no [github](https://github.com/felipecampossantos/INE5429/blob/master/t2/src/generate_primes.py)

O teste feito retornou todos os primos no período de 35.6s aproximadamente, nos arquivos “generated\_primes.csv” e “generated\_primes.txt” do github é possível ver os tempos e os números que foram retornados do teste.

No tabela abaixo temos os tempos necessários para se achar um primo com “bits” digitos binários, e no gráfico abaixo podemos ver a curva dos tempos necessários para gerar os números com seus respectivos bits de tamanho.

| BITS | TEMPO (s) |
| --- | --- |
| 40 | 0.000974 |
| 56 | 0.002 |
| 80 | 0.007034 |
| 128 | 0.020005 |
| 168 | 0.033997 |
| 224 | 0.175004 |
| 256 | 0.090036 |
| 512 | 0.097998 |
| 1024 | 3.636036 |
| 2048 | 6.202995 |
| 4096 | 25.40896 |
| **TOTAL** | **35.66504** |



Com isso, é finalizado o trabalho sobre geração de números pseudo-aleatorios e de verificação de primalidade, concluindo que a geração de números primos de até 512bits de tamanho podem levar tempos insignificantes em computadores atuais, mas a partir de números de 1024 o tempo já passa a crescer significantemente.

# Referências

<https://asecuritysite.com/encryption/blum#:~:text=Blum%20Blum%20Shub%20uses%20the,the%20difficulty%20in%20factorizing%20M>

<https://en.wikipedia.org/wiki/Fermat_primality_test>

<https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test>

<https://pt.wikipedia.org/wiki/Blum_Blum_Shub>

<https://en.wikipedia.org/wiki/Lehmer_random_number_generator>

<https://jeremykun.com/2016/07/11/the-blum-blum-shub-pseudorandom-generator/>