Trabalho Individual – Números Primos

Aluna: Jéssica Regina dos Santos
Matrícula: 22100626
Data de entrega: 28/03/25

Objetivos. O presente relatório tem como objetivo principal o estudo e a análise de geradores de números pseudo-aleatórios e testes de primalidade, com foco em sua aplicabilidade na área de criptografia e segurança da informação.

Sumário.

To see a world in a grain of sand
Or a heaven in a wild flower,
Hold infinity in the palm of your hand
And eternity in an hour.
WILLIAM BLAKE (1757-1827)

1. Números Pseudo-aleatórios

1.1 Blum Blum Shub

Criado por Lenore Blum, Manuel Blum e Michael Shub em 1968, o gerador de números pseudo aleatórios Blum Blum Shub (BBS; $x^2 \, mod \, M$) é um algoritmo determinístico que produz sequências de bits pseudo aleatórios a partir de um valor inicial (semente) e de um módulo N (Blum et al., 1986).

O módulo N deve ser o produto de dois primos distintos p e q, ambos congruentes a 3 módulo 4, ou seja, $p \equiv q \equiv 3 \, mod \, 4$. Essa condição é muito importante, pois garante que a função de raiz quadrada módulo M seja sempre uma permutação sobre o conjunto dos resíduos quadráticos, o que é essencial para a segurança e a reversibilidade controlada do algoritmo BBS (Blum et al., 1986).

Dado um valor inicial $x_0 \in QR$ (o conjunto dos resíduos quadráticos módulo M), a sequência é construída iterativamente pela fórmula $x_{i+1} = x_i^2 \, mod \, M$. A cada passo da iteração, é extraído um bit da sequência, o que fornece uma sequência binária (Blum et al., 1986). Este procedimento é eficiente, pois as operações de exponenciação modular e de cálculo de paridade são computacionalmente rápidas.

Uma das características mais relevantes do gerador é sua unidirecionalidade ("só tem um sentido de deslocamento"). Conhecendo M e a semente x0, é fácil calcular a sequência x1,x2,…, mas é computacionalmente inviável recuperar os valores anteriores, como x_{-1}, sem conhecer os fatores primos p e q. Isso decorre do fato de que calcular raízes quadradas módulo N é um problema computacionalmente difícil, desde que a fatoração de N seja desconhecida (Blum et al., 1986).

Além disso, os autores demonstram que o gerador é imprevisível em tempo polinomial. Isso significa que, dado um prefixo da sequência gerada, nenhum algoritmo probabilístico de tempo polinomial consegue prever o próximo bit com probabilidade

significativamente maior que ½. Formalmente, esse resultado está demonstrado no Teorema 4 do artigo (Blum et al., 1986).

Adicionalmente, o gerador também satisfaz um importante critério estatístico de aleatoriedade. Segundo o Teorema 5 do artigo, as sequências geradas passam em todos os testes estatísticos probabilísticos de tempo polinomial, de acordo com a definição dada por Yao. Ou seja, essas sequências não podem ser distinguidas de sequências verdadeiramente aleatórias por nenhum algoritmo de tempo polinomial (Blum et al., 1986).

Outra propriedade relevante é a possibilidade de reverter a sequência quando se conhece a fatoração de N. Nesse caso, é possível computar a raiz quadrada única que também é um resíduo quadrático, utilizando o Teorema Chinês do Resto e propriedades dos primos congruentes a 3 mod 4 (Blum et al., 1986).

Por fim, o artigo analisa o período das sequências geradas, mostrando que ele é um divisor da função de Carmichael. Em muitos casos, esse período é suficientemente longo para assegurar uma boa distribuição estatística dos bits gerados, o que reforça o uso do gerador em contextos onde a imprevisibilidade é essencial, como criptografia de chave pública e geração de senhas seguras (Blum et al., 1986).

1.1.2 Exemplo

Um gerador de números pseudo aleatórios bastante interessante e com forte base teórica é o baseado na fórmula recursiva $x_{i+1} = x_i^2 \, mod \, M$.

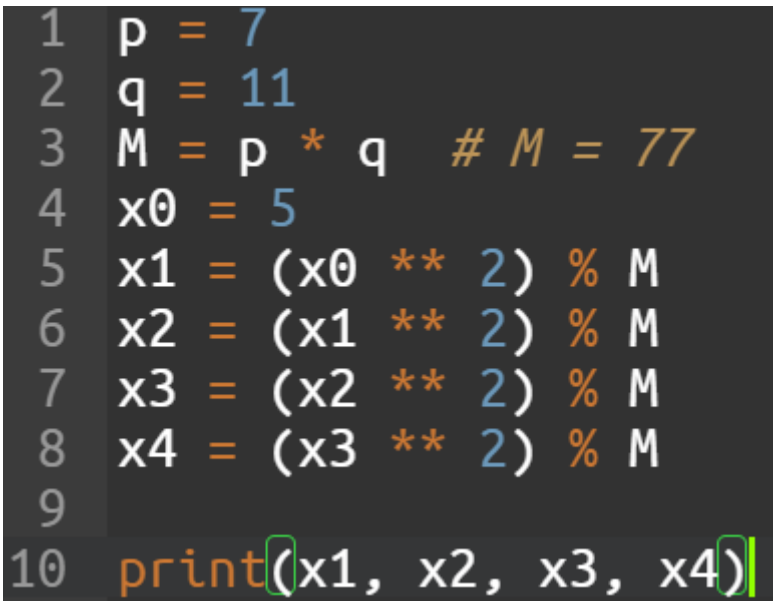
Neste caso, escolhemos um valor inicial x_0 (conhecido como semente ou *seed*), e o valor de M é dado por M = p * q, onde p e q são números primos.

Um requisito muito importante é que tanto p quanto q devem ser congruentes a 3 módulo 4, ou seja, $p \equiv 3 \, mod \, 4 \; e \; q \equiv 3 \, mod \, 4$.

Isso significa que, ao dividirmos p ou q por 4, o resto da divisão deve ser 3. Por exemplo:

- p = 7 → 7 ÷ 4 = 1 com resto 3 → válido.
- p = 11 → 11 ÷ 4 = 2 com resto 3 → válido.
- p = 13 → 13 ÷ 4 = 3 com resto 1 → inválido.

Abaixo, segue uma implementação simples em Python, com p = 7, q = 11, x_0= 5, gerando uma sequência de números pseudo aleatórios:



```
p = 7
q = 11
M = p * q  # M = 77
x0 = 5
x1 = (x0 ** 2) % M
x2 = (x1 ** 2) % M
x3 = (x2 ** 2) % M
x4 = (x3 ** 2) % M

print(x1, x2, x3, x4)
```

Figura 1. Output: 25 9 4 16.

Cada valor é calculado como o quadrado do valor anterior, módulo M. Em aplicações práticas, normalmente se extrai apenas o bit menos significativo (LSB) de cada número para formar a sequência binária pseudo aleatória. No exemplo acima, os valores correspondem aos bits:

- 25 → 11001b → 1
- 09 → 01001b → 1
- 04 → 00100b → 0
- 16 → 10000b → 0

Ou seja, a saída binária seria: '1100'.

Esse método é conhecido como o Gerador de Números Pseudo-aleatórios Blum Blum Shub (BBS) e sua segurança está fundamentada na dificuldade do problema de fatoração de inteiros, ou seja, recuperar p e q a partir de M é computacionalmente inviável para valores suficientemente grandes.

Embora esse método seja relativamente lento, é considerado um dos geradores de números aleatórios mais seguros já formalmente provados. Esse tipo de gerador é utilizado principalmente em processos de geração de chaves criptográficas.

Agradecimentos especiais ao Professor Bill Buchanan da Escola de Computação da Universidade de Edinburgh Napier pelo exemplo extremamente didático e de fácil compreensão, que serviu como base para esta explicação adaptada.

1.1.3 Análise Gráfica dos Resultados

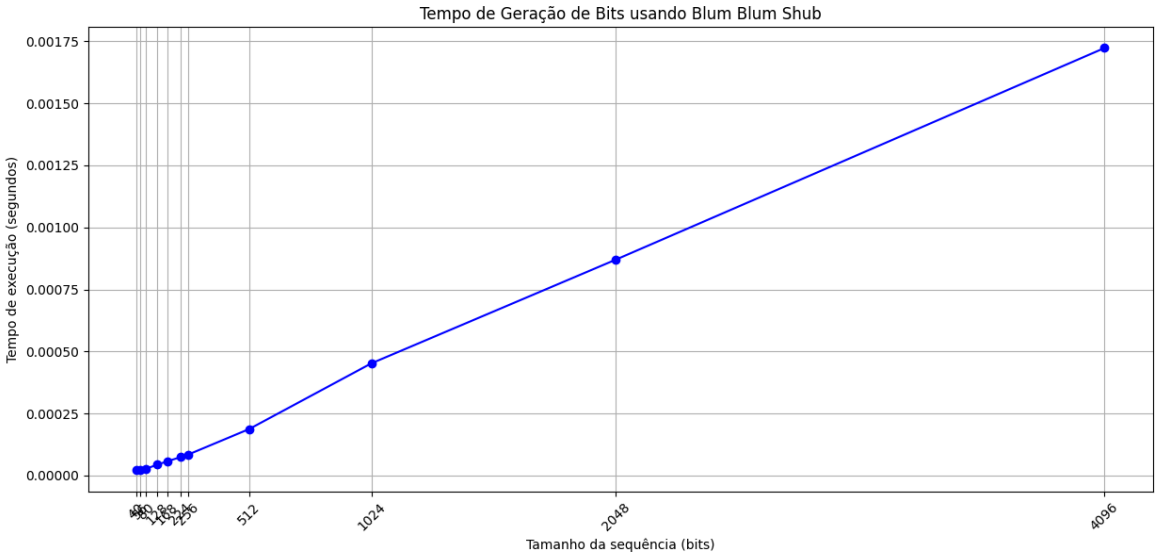


Figura 2. Tempo de geração de bits usando BBS.

Por curiosidade, gerei um gráfico que mostra a relação entre o tamanho do número (em bits) e o tempo de geração (em segundos) usando o algoritmo Blum Blum Shub. Ele evidencia o crescimento gradual do tempo de execução conforme o tamanho dos bits aumenta, especialmente a partir de 1024 bits.

Tendência observada: Crescimento quase linear no tempo de execução em relação ao tamanho dos bits até 1024 bits. A partir de 1024 bits, há um crescimento mais acentuado, porém ainda dentro de uma ordem de grandeza bem controlada para os padrões do BBS. O tempo para gerar 4096 bits ainda é inferior a 2 milissegundos, o que demonstra que o BBS, mesmo sendo criptograficamente seguro e pesado, é viável para aplicações que não exigem geração em tempo real em altíssima velocidade.

1.1.4 Conclusões

O Blum Blum Shub se mostrou eficaz para gerar sequências de até 4096 bits em menos de 1,5 milissegundos, o que é um ótimo resultado considerando a segurança criptográfica oferecida. Logo, para aplicações que exigem segurança, o BBS continua sendo uma escolha sólida, especialmente quando o desempenho é aceitável e quando combinado com outras técnicas que compensam suas limitações.

1.2 Linear Feedback Shift Registers (LFSRs)

Os Registradores de Deslocamento com Realimentação Lineares (ou Linear Feedback Shift Registers – LFSRs) são estruturas amplamente utilizadas em sistemas digitais, especialmente em áreas como criptografia, geração de números pseudo aleatórios, detecção e correção de erros. Sua popularidade deve-se à simplicidade de implementação, alta eficiência e capacidade de gerar sequências periódicas e determinísticas (GeeksforGeeks, 2024).

Um LFSR é composto por um conjunto de flip-flops (registradores de deslocamento) e uma função de realimentação linear baseada na operação XOR. O valor de entrada de cada ciclo de clock é calculado a partir de determinados bits do registrador (chamados *taps*) e é

realimentado ao início do registrador. A posição desses taps é determinada por um polinômio característico que define o comportamento e a periodicidade da sequência gerada (GeeksforGeeks, 2024).

As principais características dos LFSRs incluem a produção de sequências pseudo aleatórias com ciclos definidos, alta eficiência computacional (baseada em operações de deslocamento e XOR) e facilidade de implementação. Quando o polinômio característico é primitivo, o LFSR pode atingir um período máximo de $2^n - 1$, onde n é o número de bits do registrador (GeeksforGeeks, 2024).

Existem diferentes tipos de LFSRs, entre eles:

- **Fibonacci LFSR**: A forma clássica, onde a realimentação é aplicada apenas ao primeiro flip-flop com base na saída dos taps. É simples de implementar e bastante usada em criptografia básica (GeeksforGeeks, 2024).

- **Galois LFSR**: A realimentação afeta múltiplos pontos simultaneamente, o que pode tornar a implementação mais eficiente em hardware (GeeksforGeeks, 2024).

- **LFSRs não-lineares (NLFSRs)**: Usam funções não-lineares (como AND ou OR) na realimentação, proporcionando sequências menos previsíveis, porém mais difíceis de analisar (GeeksforGeeks, 2024).

- **LFSRs truncados**: Usam apenas parte dos bits do LFSR original, o que reduz o período da sequência mas simplifica a implementação (GeeksforGeeks, 2024).

- **LFSRs programáveis**: Permitem a alteração dinâmica dos taps, adequando-se a diferentes polinômios conforme a necessidade da aplicação (GeeksforGeeks, 2024).

Apesar de suas vantagens (como simplicidade, alta velocidade e baixo custo de hardware), os LFSRs apresentam limitações importantes, sobretudo no contexto da segurança. Sua natureza determinística e previsível os torna vulneráveis a ataques criptográficos, como ataques por correlação ou baseados em complexidade linear (GeeksforGeeks, 2024).

Entretanto, os LFSRs continuam relevantes nas áreas de engenharia digital e criptografia, especialmente quando combinados com outras técnicas que compensam suas limitações. Sua eficiência e versatilidade ainda os tornam úteis em ampla gama de aplicações tecnológicas.

1.2.1 Exemplo Didático: Aplicação do Fibonacci LFSR

Leve em consideração que 0’ = 1 e 1’ = 0.
Leve em consideração que + = ⊕.

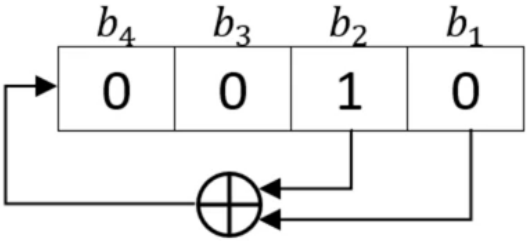
Exemplo 1:

$b_1 \leftarrow b'_2$

$b_2 \leftarrow b'_3$

$b_3 \leftarrow b'_4$

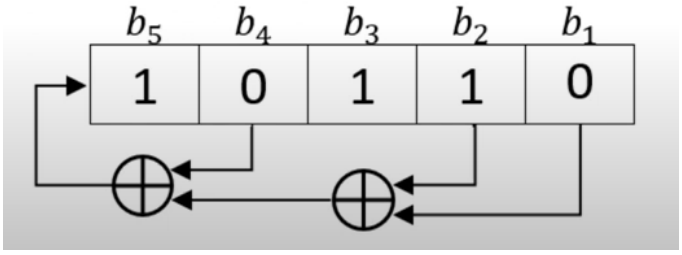
$b_4 \leftarrow b'_1 + b'_2$



Exemplo 2:

$b_5 \leftarrow b'_1 + b'_2 + b'_4$

Semente: 10110



	b_5	b_4	b_3	b_2	b_1
1	1	0	1	1	0
2	1	1	0	1	1
3	1	1	1	0	1
4	0	1	1	1	0
5	0	0	1	1	1
6	0	0	0	1	1
7	0	0	0	0	1
8	1	0	0	0	0
...	...	...	...	...	...

1.2.2 Registrador de Deslocamento com Realimentação Linear (LFSR) na Configuração de Galois

Nomeado em homenagem ao matemático francês Évariste Galois, o LFSR na configuração de Galois (também conhecido como modular, com XORs internos ou one-to-many LFSR) representa uma estrutura alternativa ao LFSR convencional (PRESS et al., 2007).

Na configuração de Galois os bits que não estão nas posições de realimentação (*taps*) são simplesmente deslocados uma posição para a direita, sem alterações. Em seguida, os bits nas posições de *tap* são "XORados" com o bit de saída antes de serem armazenados na próxima posição. O novo bit de saída é o próximo bit de entrada.

Se o bit de saída for 0, todo o registrador é deslocado para a direita sem alterações. Se o bit de saída for 1, os bits nas posições de *tap* são invertidos (via XOR) e o registrador é então deslocado, com o bit de entrada tornando-se 1.

1.2.3 Análise Gráfica dos Resultados

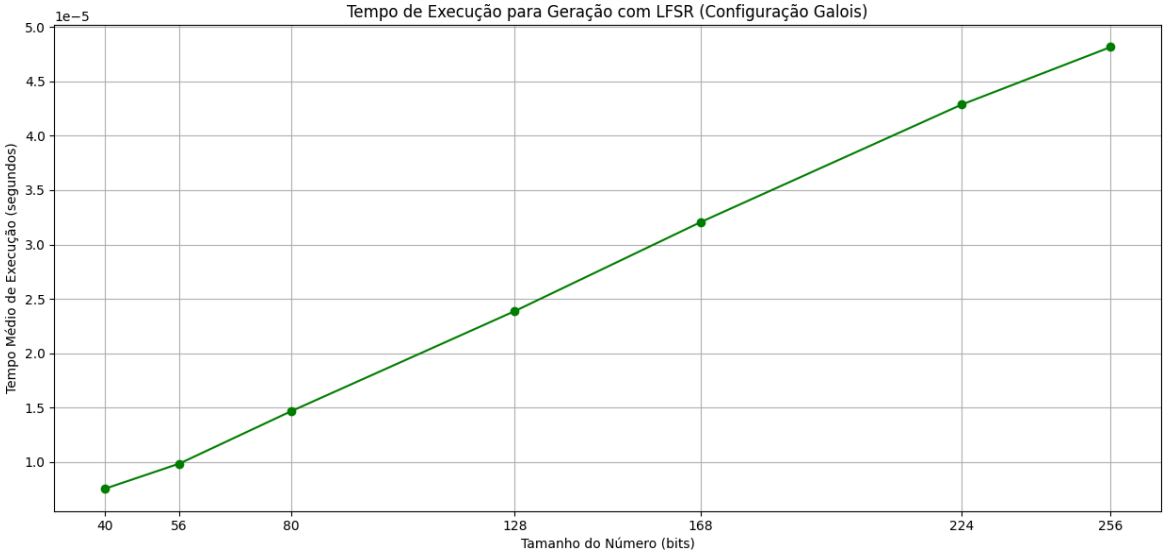


Figura 3. Tempo de geração de bits usando LFSR (configuração Galois).

Também gerei um gráfico de tempo médio para gerar números utilizando Galois LFSR para diferentes tamanhos de números, variando de 40 a 256 bits. A partir dessa visualização, podemos fazer algumas observações e análises, por exemplo:

O tempo médio de execução aumenta à medida que o número de bits desejado cresce. Isso é esperado, pois a cada bit adicional gerado, o algoritmo precisa realizar mais operações de deslocamento e realimentação no registrador, o que aumenta o custo computacional.

A curva do gráfico parece ser aproximadamente linear. Ou seja, o tempo para gerar um número de 256 bits é um múltiplo maior do tempo para gerar um número de 40 bits, mas o aumento não é tão agressivo quanto o crescimento exponencial. Isso é indicativo de que o algoritmo tem uma complexidade razoavelmente eficiente para os tamanhos testados.

Embora a curva continue a subir conforme o tamanho aumenta, se o gráfico fosse estendido para tamanhos ainda maiores (como 512 ou 1024 bits), o tempo de execução poderia aumentar. Isso acontece porque, no caso de LFSR, o número de iterações e operações

cresce linearmente com o número de bits, podendo resultar em limitações práticas à medida que os números desejados aumentam ainda mais.

1.2.4 Conclusões

Embora o Galois LFSR seja eficiente para tamanhos menores, ele não é tão adequado para números muito maiores, onde outros algoritmos de geração de números pseudo-aleatórios podem ser mais apropriados.

1.3 Análise Comparativa

O Blum Blum Shub apresentou um tempo de geração proporcional ao tamanho da sequência de bits, conforme evidenciado pelo gráfico da Figura 2. Para sequências pequenas (40 a 256 bits), o tempo se manteve na ordem de 0,00025 a 0,0015 segundos, enquanto para tamanhos maiores (como 4096 bits), houve um aumento significativo, ainda que dentro de limites aceitáveis. Essa escalabilidade se deve à natureza computacionalmente intensa do BBS, que realiza operações de exponenciação modular ($x_{i+1} = x_i^2 \, mod \, M$) a cada bit gerado. Quanto maiores os primos p e q (que geram M), maior o custo computacional, justificando o crescimento no tempo de execução.

Em contraste, o LFSR na configuração Galois demonstrou um desempenho muito mais rápido, com tempos na ordem de $1e^{-5}$segundos para todas as sequências testadas (40 a 256 bits), conforme ilustrado na Figura 3. Essa eficiência se deve à simplicidade das operações realizadas: deslocamento de bits, XOR e extração do bit menos significativo (LSB), todas executadas em tempo constante. Como o registrador utilizado possui apenas 16 bits, o algoritmo não sofre um aumento significativo no tempo mesmo para sequências maiores, embora sua utilidade seja limitada devido ao período curto das sequências geradas.

A análise de complexidade também explica essa diferença de desempenho. Enquanto o BBS possui complexidade $\simeq O(N \times k^2)$ (onde N é o número de bits e k o tamanho dos primos), o LFSR opera em O(N), sendo muito mais eficiente.

No entanto, essa eficiência tem um custo: o LFSR é linear e previsível. Já o BBS, apesar de mais lento, é considerado seguro para geração de chaves e criptografia devido à sua imprevisibilidade.

Logo, se a prioridade for velocidade, o LFSR é a melhor opção.

Se a segurança for essencial (como em criptografia), o BBS, apesar de mais lento, é a escolha adequada.

2. Números Primos

2.1 Testes de Primalidade

O volume de dados vem crescendo a uma velocidade extraordinária, e, para proteger essa quantidade crescente de informações, são necessárias técnicas de segurança da informação cada vez mais sofisticadas. A melhoria na proteção de dados demanda métodos criptográficos mais avançados, o que, por sua vez, exige o uso de números semiprimos maiores, difíceis de serem fatorados. Assim, a verificação eficiente da primalidade de grandes números primos torna-se fundamental para aprimorar a segurança da informação (Narayanan, 2014).

Um teste de primalidade é um algoritmo cujo objetivo é determinar se um número dado é primo. Alguns testes de primalidade são determinísticos, ou seja, fornecem uma resposta correta para todo número de entrada, distinguindo de forma inequívoca números primos de compostos. O teste determinístico mais rápido conhecido foi desenvolvido em 2004 por Agrawal, Kayal e Saxena, conhecido como teste AKS, com complexidade $O(log^6 n)$, onde $O(f(n))$ é definido como $O(f(n)\log(f(n))^k)$ para algum inteiro k (Agrawal et al, 2004).

Testes probabilísticos de primalidade, por outro lado, são geralmente mais rápidos, embora não garantam precisão absoluta. Esses testes verificam se o número de entrada n satisfaz certas condições que todo número primo deve cumprir. Se n não satisfizer essas condições, ele é certamente composto; se satisfizer, é considerado provavelmente primo.

2.3 Teste de Primalidade de Fermat

O Teste de Primalidade de Fermat baseia-se no Pequeno Teorema de Fermat, que afirma que, se n é primo, então $a^n - 1 \equiv 1 \, mod \, n$ para todo $a < n$. Para verificar se um número n é primo, escolhe-se um $a < n$ e testa-se se $a^n - 1 \equiv 1 \, mod \, n$. Se a congruência não for satisfeita, n é composto; caso contrário, n é provavelmente primo. Contudo, o Teste de Fermat apresenta uma taxa de erro elevada, sendo comum que números compostos sejam classificados incorretamente como provavelmente primos. Para contornar essas limitações, utiliza-se o Teste de Primalidade de Miller-Rabin (Narayanan, 2014).

2.3 Teste de Primalidade de Miller-Rabin

O Teste de Primalidade de Miller-Rabin é uma extensão do Teste de Fermat, oferecendo maior confiabilidade.

O teste opera da seguinte maneira: dado um inteiro ímpar n, decompõe-se $n - 1$ como $d \times 2^e$, onde d é ímpar. Escolhe-se um inteiro positivo $a < n$. Se $a^d \equiv 1 \, mod \, n$ ou $a^{2rd} \equiv -1 \, mod \, n$ para algum $r < e$, então n é provavelmente primo. Caso contrário, n é composto.

Se n for composto, mas passar no teste, diz-se que a é um não-testemunho de n e que n é um pseudoprimo forte na base a (Miller, 1976) (Rabin, 1980).

2.2.1 Exemplo 1

Considere $n = 65$.

Decompomos $65 - 1 = 64 = 2^6 \times 1$, portanto $d = 1$ e $e = 6$.

Tomando $a = 8$, temos $8^1 \equiv 8 \bmod 65 \neq 1$, mas $8^{2^1 x 1} = 8^2 \equiv -1 \bmod 65$.

Assim, segundo o teste, 65 seria classificado como provavelmente primo. Contudo, 65 é composto ($65 = 5 \times 13$).

Escolhendo $a = 2$, o teste revela a compostidade de 65, mostrando que 2 é uma testemunha e 8 é um não-testemunho.

O Teste de Miller-Rabin é consideravelmente mais preciso que o Teste de Fermat. Apesar da existência de uma infinidade de números compostos — os chamados números de Carmichael — que satisfazem $a^{n-1} \equiv 1 \bmod n$ para todo a coprimo a n (Alford et al., 1994), Michael O. Rabin demonstrou que, para qualquer inteiro ímpar composto n, o número de não-testemunhos é, no máximo, $n \div 4$, podendo ser reduzido a $\varphi(n) \div 4$ para $n \geq 25$ (Rabin, 1980).

2.2.2 Exemplo 2

Verificando $n = 9$ na base 3, o Teste de Fermat mostra $3^{90} \equiv 1\ mod\ 91$, sugerindo incorretamente que 91 é primo. No entanto, aplicando o Teste de Miller-Rabin, observamos que $3^{45} \equiv 27\ mod\ 91$, evidenciando que 3 é uma testemunha e que 91 é composto.

2.3 Análise Comparativa

O Teste de Fermat demonstrou ser mais rápido para números pequenos (40 a 256 bits), com tempos na ordem de 10^{-3} a 10^{-1}segundos, conforme ilustrado na Figura 4.2. No entanto, para números maiores (como 4096 bits), seu tempo de execução aumenta significativamente, chegando a dezenas ou centenas de segundos.

Essa escalabilidade se deve à operação de exponenciação modular ($a^n - 1\ mod\ n$), cujo custo cresce com o tamanho do número.

Apesar de sua eficiência, o teste de Fermat possui uma limitação crítica: ele pode erroneamente classificar números de Carmichael (compostos que satisfazem a condição de Fermat) como primos, reduzindo sua confiabilidade em aplicações que exigem alta precisão.

Por outro lado, o teste de Miller-Rabin, representado nas figuras 5.1 e 5.2, apresentou um tempo de execução ligeiramente superior para números pequenos, mas com uma diferença mais acentuada para tamanhos maiores (acima de 1024 bits). Essa diferença ocorre porque o Miller-Rabin realiza etapas adicionais, como a decomposição de $n - 1$ em fatores e verificações iterativas, aumentando seu custo computacional.

Contudo, esse método é mais confiável, com uma probabilidade de erro mais baixa, tornando-o padrão em sistemas criptográficos como RSA.

Em termos de complexidade computacional, ambos os algoritmos possuem uma ordem de grandeza semelhante, $O(k \times log^3 n)$, onde k é o número de iterações e n o número

testado. A diferença prática está na velocidade: o Miller-Rabin é mais lento devido às operações extras, enquanto o Fermat é mais rápido, porém menos seguro.

A escolha entre os dois métodos dependerá então do contexto de aplicação: se a velocidade for prioritária (e eventuais falsos positivos forem toleráveis, como em testes preliminares), o teste de Fermat é uma opção viável.

Porém, para aplicações preocupadas com segurança, onde a confiabilidade é essencial, o Miller-Rabin é a escolha mais adequada.

Tabela de Números Primos Gerados:

	Algoritmo	Tamanho do Número (bits)	Número Primo Gerado	Tempo para Gerar (s)
0	Fermat	40	874044314243	0.000194
1	Fermat	56	11815934883334877	0.000116
2	Fermat	80	643187509791524926280861	0.005589
3	Fermat	128	102931863063580187750672101538551141249	0.004879
4	Fermat	168	168954462757022695302784896046445734889369026...	0.005407
5	Fermat	224	206093681651418649195475713515232218012023044...	0.018186
6	Fermat	256	459860096351635228205907801163108599648258170 4...	0.013081
7	Fermat	512	892777602258888708793983337668912971104720558 5...	0.260057
8	Fermat	1024	164975531557893065518140519802589927799293354 9...	0.734602
9	Fermat	2048	179390311340858908790323226172807279690769518 3...	12.712863
10	Fermat	4096	180398052425008861946503235386718038186144369 6...	140.994728

Figura 4.1 Tempo para gerar números primos usando Teste de Fermat (tabela).

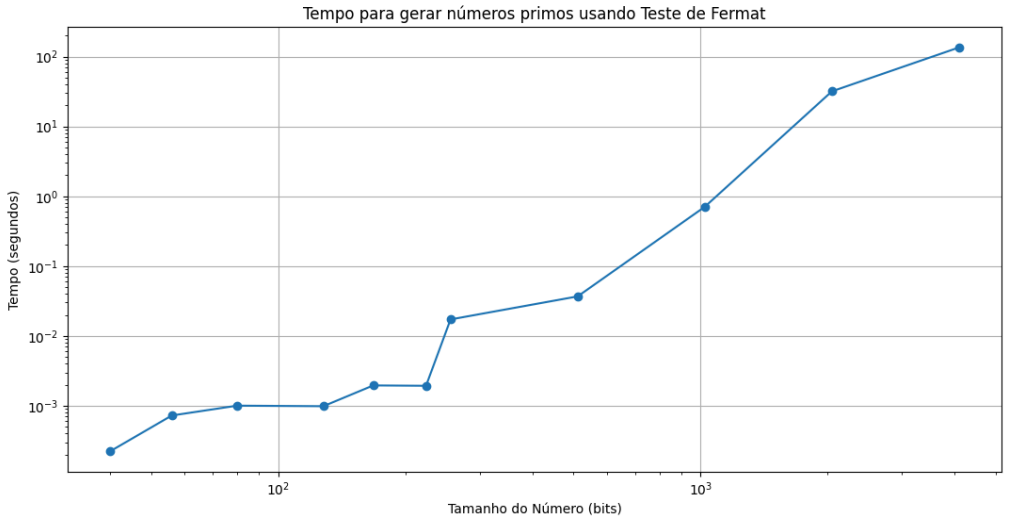


Figura 4.2 Tempo para gerar números primos usando Teste de Fermat (gráfico equivalente).

	Algoritmo	Tamanho do Número (bits)	Número Primo Gerado	Tempo para Gerar (s)
0	Miller-Rabin	40	997935124411	0.000553
1	Miller-Rabin	56	61416515261357677	0.000504
2	Miller-Rabin	80	106760326032689322091934 3	0.000948
3	Miller-Rabin	128	311816980637365528824348574735179769973	0.001957
4	Miller-Rabin	168	339504708758720662173379193903519239363678924 8...	0.005254
5	Miller-Rabin	224	189748638683776451164497971093643586995411968 1...	0.014418
6	Miller-Rabin	256	598185377136424413160946801805483563881428214 7...	0.003869
7	Miller-Rabin	512	860015495118579370566062871582731945408860690 0...	0.350892
8	Miller-Rabin	1024	108690281135115849263652422805644251073736730 5...	5.097350
9	Miller-Rabin	2048	233659135593329290937224788091215013268632748 7...	0.379537
10	Miller-Rabin	4096	734656416377139614239797084676049269946757572 4...	402.471444

Figura 5.1 Tempo para gerar números primos usando Miller-Rabin (tabela).

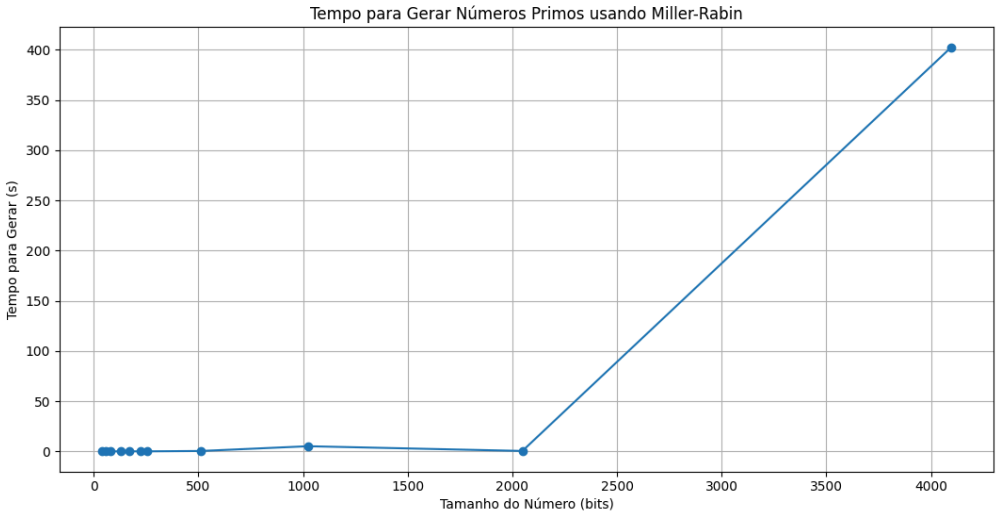


Figura 5.2 Tempo para gerar números primos usando Miller-Rabin (gráfico equivalente).

Referências

AGRAWAL, M.; KAYAL, N.; SAXENA, N. **PRIMES is in P**. *Annals of Mathematics*, v. 160, n. 2, p. 781–793, 2004.

ALFORD, W. R.; GRANVILLE, A.; POMERANCE, C. **There are Infinitely Many Carmichael Numbers**. *Annals of Mathematics*, v. 139, p. 703–722, 1994.

BLUM, L.; BLUM, M.; SHUB, M. **A Simple Unpredictable Pseudo-Random Number Generator**. *SIAM Journal on Computing*, v. 15, n. 2, 1986. Disponível em: <https://people.tamu.edu/~rojas//bbs.pdf>. Acesso em: 25 abr. 2025.

BLUM, L.; BLUM, M.; SHUB, M. **Comparison of two pseudo-random number generators**. In: *Advances in Cryptology*. Springer, Boston, MA, 1983. Acesso em: 25 abr. 2025.

BUCHANAN, William J. **Blum Blum Shub**. *Asecuritysite.com*, 2025. Disponível em: <https://asecuritysite.com/encryption/blum>. Acesso em: 25 abr. 2025.

Fermat Method of Primality Test. *GeeksforGeeks*. Disponível em: <https://www.geeksforgeeks.org/fermat-method-of-primality-test/>. Acesso em: 26 abr. 2025.

Linear Feedback Shift Registers (LFSR). *GeeksforGeeks*, 2024. Disponível em: <https://www.geeksforgeeks.org/linear-feedback-shift-registers-lfsr/>. Acesso em: 25 abr. 2025.

Linear-feedback shift register. *Wikipedia*. Disponível em: <https://en.wikipedia.org/wiki/Linear-feedback_shift_register>. Acesso em: 25 abr. 2025.

MILLER, G. **Riemann's Hypothesis and Tests for Primality**. *Journal of Computer and System Sciences*, v. 13, n. 3, p. 300–317, 1976.

NARAYANAN, A. **Improving the Speed and Accuracy of the Miller-Rabin**. *MIT PRIMES*, 2014. Disponível em: https://math.mit.edu/research/highschool/primes/materials/2014/Narayanan.pdf. Acesso em: 26 abr. 2025.

NORTH CAROLINA SCHOOL OF SCIENCE AND MATHEMATICS. **Cryptography: Linear Feedback Shift Register**. *YouTube*, 14 out. 2019. Disponível em: <https://www.youtube.com/watch?v=1UCaZjdRC_c>. Acesso em: 25 abr. 2025.

Primality Test | Set 3 (Miller–Rabin). *GeeksforGeeks*. Disponível em: <https://www.geeksforgeeks.org/primality-test-set-3-miller-rabin/>. Acesso em: 26 abr. 2025.

PRESS, W.; TEUKOLSKY, S.; VETTERLING, W.; FLANNERY, B. **Numerical Recipes: The Art of Scientific Computing**. 3. ed. Cambridge University Press, 2007. p. 386. ISBN: 978-0-521-88407-5. Acesso em: 25 abr. 2025.

RABIN, M. O. **Probabilistic algorithm for testing primality**. *Journal of Number Theory*, v. 12, n. 1, p. 128–138, 1980.