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 $x_0 \in QR\square$ (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:

```python
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

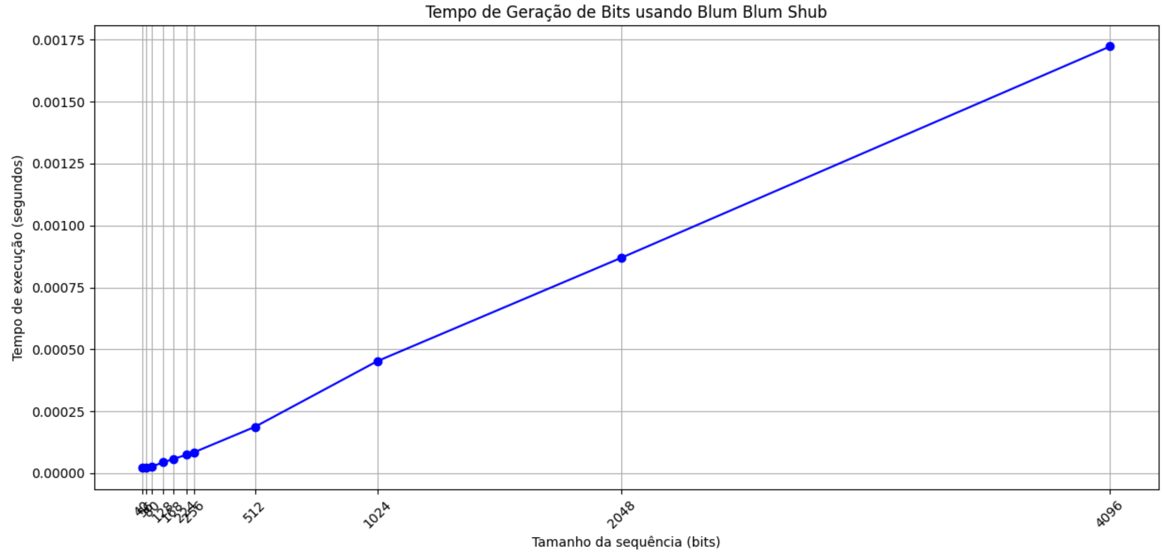


Figura 2. Tempo de geração de bits usando BBS.

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 + = ⊕.

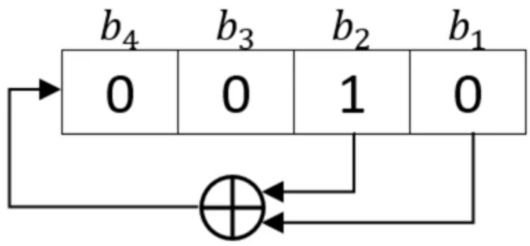
Exemplo 1:

$b_1 \leftarrow b'_2$

$b_2 \leftarrow b'_3$

$b_3 \leftarrow b'_4$

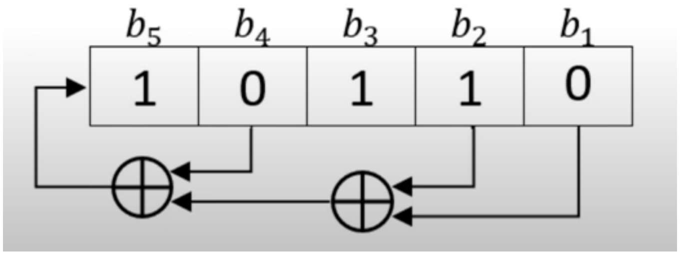
$b_4 \leftarrow b'_1 + b'_2$



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

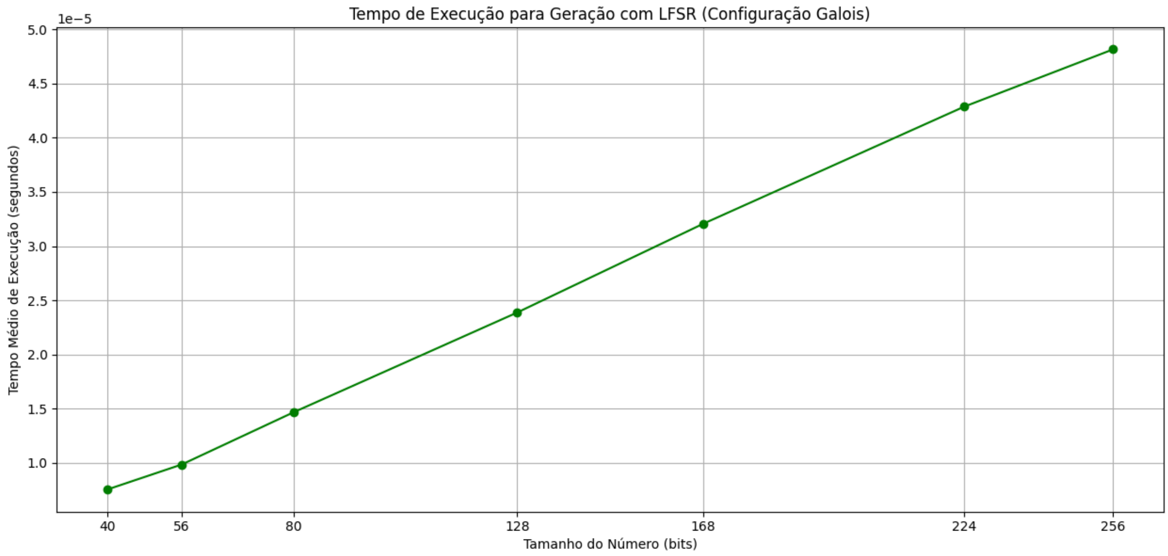


Figura 3. Tempo de geração de bits usando LFSR (configuração Galois).

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 $8^1 \equiv 8 \mod 65 \neq 1$, mas $8^{2^1 x 1} = 8^2 \equiv -1 \mod 65$.

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 $a^{n-1} \equiv 1 \mod n$ 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	16895446275702269530278489604644573488933369026...	0.005407
5	Fermat	224	2060936816514186491954757135152322180120230244...	0.018186
6	Fermat	256	4598600963516352282059078011631085996482581704...	0.013081
7	Fermat	512	8927776022588887087939833376689129711047205585...	0.260057
8	Fermat	1024	1649755315578930655181405198025899277992933549...	0.734602
9	Fermat	2048	1793903113408589087903232261728072796907695183...	12.712863
10	Fermat	4096	1803980524250088619465032353867180381861443696...	140.994728

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

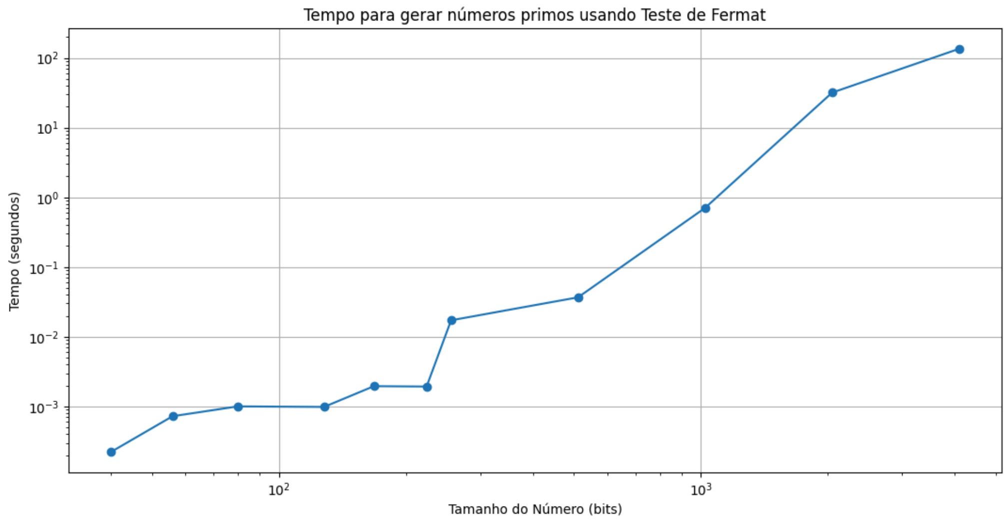


Figura 4.2 Tempo para gerar números primos usando Teste de Fermat (gráfico equivalente).

	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	1067603260326893220919343	0.000948
3	Miller-Rabin	128	311816980637365528824348574735179769973	0.001957
4	Miller-Rabin	168	3395047087587206621733791939035192393636789248...	0.005254
5	Miller-Rabin	224	1897486386837764511644979710936435869954119681...	0.014418
6	Miller-Rabin	256	5981853771364244131609468018054835638814282147...	0.003869
7	Miller-Rabin	512	8600154951185793705660628715827319454088606900...	0.350892
8	Miller-Rabin	1024	1086902811351158492636524228056442510737367305...	5.097350
9	Miller-Rabin	2048	2336591355933292909372247880912150132686327487...	0.379537
10	Miller-Rabin	4096	7346564163771396142397970846760492699467575724...	402.471444

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

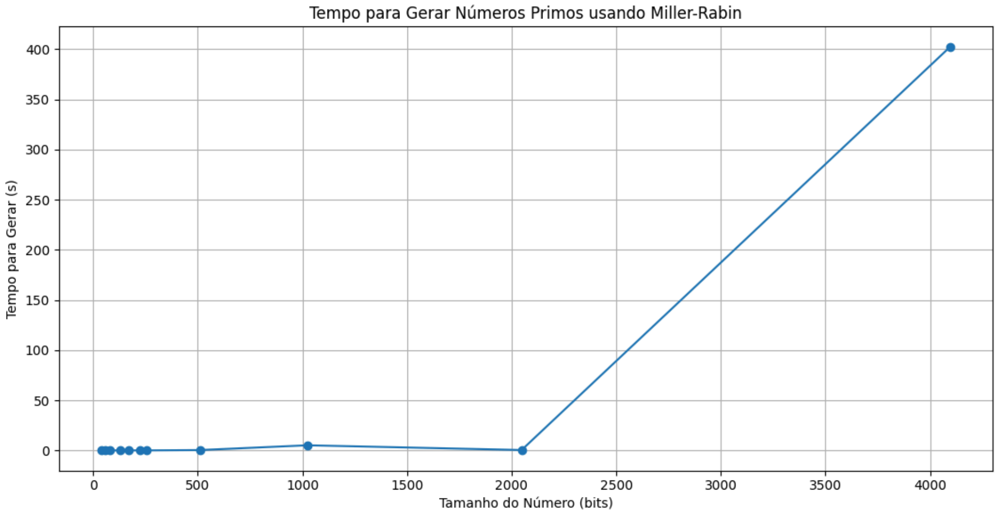


Figura 5.2 Tempo para gerar números primos usando Miller-Rabin (gráfico equivalente).

3. Códigos

3.1 Blum Blum Shub

```python
import sympy
import random
import time
import matplotlib.pyplot as plt

# Função que encontra o próximo primo válido para o BBS (p ≡ 3 mod 4)
def proximo_primo_valido(x):
    p = sympy.nextprime(x)
    while p % 4 != 3:
        p = sympy.nextprime(p)
    return p

# Função para gerar bits pseudo-aleatórios e retornar detalhes
def gera_bits_pa_bbs(semente, p, q, N):
    M = p * q
    x = semente
    bit_output = ""
    for _ in range(N):
        x = (x * x) % M
        b = x % 2
        bit_output += str(b)
    num_zeros = bit_output.count("0")
    num_uns = bit_output.count("1")
    return bit_output, num_zeros, num_uns, M

# Função principal que testa diversos tamanhos e exibe os dados estendidos
def testa_tamanhos(tamanhos):
    resultados = []
    for tamanho in tamanhos:
        x = random.randint(1, 10**10)
        y = random.randint(1, 10**10)
        p = proximo_primo_valido(x)
        q = proximo_primo_valido(y)
        semente = random.randint(1, 10**10)
        N = tamanho

        tempo_inicio = time.time()
        bits, zeros, uns, M = gera_bits_pa_bbs(semente, p, q, N)
        tempo_fim = time.time()
        tempo_total = tempo_fim - tempo_inicio

        # Adiciona informações completas no resultado
        resultados.append({
            "tamanho": tamanho,
            "time": tempo_total,
            "p": p,
            "q": q,
            "M": M,
            "semente": semente,
            "bits": bits,
            "zeros": zeros,
            "uns": uns
        })

    return resultados

# Tamanhos dos números a serem gerados (em bits)
tamanhos = [40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096]

# Executa o teste
```

```python
resultados = testa_tamanhos(tamanhos)

# Exibe o tempo de execução
print("Algoritmo: Blum Blum Shub")
print(f"{'Tamanho (bits)':<15}{'Tempo (segundos)'}")
for r in resultados:
    print(f"{r['tamanho']:<15}{r['time']:.6f}")

print("\nDetalhes para cada tamanho testado:")
for r in resultados:
    print(f"\np:  {r['p']}")
    print(f"q:  {r['q']}")
    print(f"M:  {r['M']}")
    print(f"Semente:  {r['semente']}")
    print(f"{r['bits']}")
    print(f"Número de zeros:    {r['zeros']}")
    print(f"Número de uns:     {r['uns']}")

###### Explicação do Código ######

### Funções auxiliares ###

# proximo_primo_valido: Encontra o próximo número primo válido para o algoritmo BBS. Isto é, um número primo congruente a 3 módulo 4.

# gera_bits_pa_bbs: Gera uma sequência de bits pseudo-aleatórios usando o BBS. Para cada bit gerado, o valor de x é elevado ao quadrado e reduzido módulo M (produto de dois primos p e q), e o bit é o valor de x % 2.

# testa_tamanhos: Testa a geração de números de vários tamanhos (em bits), medindo o tempo necessário para gerar cada sequência de bits. O código gera números primos p e q para cada execução, além de uma semente aleatória. A função retorna uma lista com os tempos de geração.

# Tabela de Resultados: checar output do programa no terminal.

###### Possíveis Limitações ######
# Tempo de Geração: Para tamanhos de número muito grandes, como 2048 ou 4096 bits, o tempo de execução pode aumentar significativamente devido à natureza computacionalmente intensa do algoritmo.
# Limitações de Memória: Gerar números com tamanhos muito grandes pode exigir uma quantidade significativa de memória, especialmente para valores próximos a 4096 bits.
# Talvez o Algoritmo Não Funcione: Para tamanhos extremamente grandes, como 4096 bits, a execução do algoritmo pode ser inviável em sistemas com recursos limitados ou em contextos que exigem um tempo de resposta muito rápido. Isso ocorre devido à complexidade do algoritmo, que cresce conforme o tamanho do número aumenta. Em tais casos, pode ser necessário utilizar outras abordagens, como algoritmos de geração de números pseudo-aleatórios mais rápidos ou usar bibliotecas dedicadas a números grandes (que são otimizadas para eficiência).

###### Interpretação técnica ######
# O tempo de geração é proporcional ao número de iterações do laço (for _ in range(N)) em gera_bits_pa_bbs(). Cada iteração envolve uma operação de exponenciação modular x = (x * x) % M, que é computacionalmente "cara".
# O custo dessas operações aumenta conforme o tamanho de M = p * q, que é diretamente influenciado pelos primos escolhidos. Mesmo assim, o uso de primos relativamente pequenos (~10 dígitos) ajuda a manter o desempenho aceitável.
# O resultado demonstra boa escalabilidade para tamanhos de chave utilizados em segurança (ex: 1024, 2048 bits).
```

```

###### Referências ######
# Adapted from:
# https://medium.com/asecuritysite-when-bob-met-alice/cryptography-in-the-family-blum-blum-and-blum-6277590f0c94
# https://asecuritysite.com/encryption/blum
# All credits to the author.

###### Geração de Gráfico ######
# Visualização da relação entre o tamanho da sequência de bits e o tempo de execução.

# Extraindo dados para o gráfico
tamanhos_bits = [r['tamanho'] for r in resultados]
tempos_execucao = [r['time'] for r in resultados]

# Criando o gráfico
plt.figure(figsize=(10,6))
plt.plot(tamanhos_bits, tempos_execucao, marker='o', linestyle='-', color='b')
plt.title('Tempo de Geração de Bits usando Blum Blum Shub')
plt.xlabel('Tamanho da sequência (bits)')
plt.ylabel('Tempo de execução (segundos)')
plt.grid(True)
plt.xticks(tamanhos_bits, rotation=45)
plt.tight_layout()
plt.show()

```

3.1.1 Saída

Algoritmo: Blum Blum Shub

Tamanho (bits)	Tempo (segundos)
40	0.000040
56	0.000042
80	0.000062
128	0.000093
168	0.000120
224	0.000161
256	0.000179
512	0.000376
1024	0.000886
2048	0.001920
4096	0.003361

Detalhes para cada tamanho testado:

p: 5881868467
q: 2923680223
M: 17196702511255228141
Semente: 1233782001
1011101111011001110101111100000000001011
Número de zeros: 17
Número de uns: 23

p: 4006660187
q: 7165866179
M: 28711190724769115473
Semente: 6434422838
10111101100000100011110000010111111110011010100000001001
Número de zeros: 29
Número de uns: 27

p: 1092994027
q: 9010962967
M: 9848928700449198109
Semente: 6817188724
01011010011011110011100010101010110010001011100011001000001011111010111010100001
Número de zeros: 40
Número de uns: 40

p: 8845834031
q: 1740138691
M: 15392978051507593421
Semente: 5564860313
11010110110010011110100100101101001001101101111111100101100110111001010101011001110011010001101011100001001111001100000101001101
Número de zeros: 59
Número de uns: 69

p: 6813415571
q: 8006507279
M: 54551661364063441309
Semente: 1472554859
111110100010111110011011111011100110011100000001110000000101111000101011011110101000000000110000101110111011011011000111010010001001011111001011110001000100010101110110100
Número de zeros: 81
Número de uns: 87

p: 793316591
q: 8782279787
M: 6967128261831046117
Semente: 8736926877
100011100100000101000101000110011010100011101001100001011100001100001100010010001011110000000110011010100111001000000100101011110000000101010010010010000010011011101110111011000011101000011000101010111001111101100001101001010

Número de zeros: 127
Número de uns: 97

p: 494345903
q: 9066844987
M: 4482157672459538261
Semente: 8135528838
0111111010101010010001111010101100001001000111000101110001110001101111100101
0111010001001011010101101011000000100001110001110011000110000101110000110011
1101011010000111100101011001001010100001110001011101111110110001011010010111
0011000100100010011111001011
Número de zeros: 128
Número de uns: 128

p: 1189078411
q: 7468339687
M: 8880441487826197357
Semente: 7830862782
0000001001010110000111011110011101101011000110011001010101101111001111011100
0010100111011000100000010011000100110111111110000111111100000000110100011011
1100110111000111011011011000111100001001111011010101000100100010101110001111
0010010100100011000010011101110010101111111110111110001001110000000100110011
0001111010111010000100000001000010111111000001011100010000010111111010000010
0111110011010110000101110111011011000010011011111101100101010101010001001101
010001011010101011001001100011100010010011101100000000011
Número de zeros: 259
Número de uns: 253

p: 6549526987
q: 4829527907
M: 31631123361366126209
Semente: 9208279488
0100010000111001101001010011110101110001111000100101000010001001011101100101
0101110001011001110100010011011100011011111101101011000000101110010100010110
1111011010111111110111110110101111011100001110100000000010001001100111011010 0
1000000000111001000011110100010001001011100001000111000000000110000001000110
1011101100100011001111011100001011101100100011011110001111111100010000111010 1
0011100011110101010000111110100111100110010101001011001001010001101101111110
1001011110001111100101110010010101110000010100111000110101000010111100100110
1010111111100001000011111110100110000101010010101111011010100111001000011000
0001101100111101010001111011011010100010101010100100001010111110101000000110
1011001010110000010111011100110100111010100010001010101000101100010101101000
0001111001001111001111011001110011000100010010110101101001001010111011000110
1101000110110100010111111011100110001100110101110011110111000100111101110000

0001000100011110110001000001101101100000001000001010001000010000111110100111
00111101011010011110010101000011 1
Número de zeros: 517
Número de uns: 507

p: 1453068283
q: 1428638459
M: 2075909232646895897
Semente: 9439271592
0101010001011110111100101000101100101011001101101100111000101001010110110111
1111010110100111111010011100111111000111110001010000111101111010001001001110
0000011111100101010010101001111010111010100011100110010010101110110100000100
1100100111110111111000010011001000101100000110111111011111111100110100001000 0
1010010111010001001001011001101110101001001011110001010100111010101111101011
0111010111111100010000010010010101011101011010110010111101000100110100100000
1100011110000010010001000000111100011000100010001110001110010000111010100001
010010001000110000010100110101100100111100000001011101010100101010010110000 1
0001010111101001011100011010000101001010011111100101010010100100010010001000
00100101110100010000101011111111000110011000100011000001001110111111001000011
1011110000000111110010010011001001110100010111010001101001101101001111100101 0
1000111011101011100011000100100000000011111101011001101001111110111000011000
0110001110101011111011001101010101000101100100110110010101100011000000100001
0110001101001001101111110101100111000001101101011100100011010011100000101011
1111010000000111111111110100101000011001011000100101110111100001111110000011 0
1110000000011001100101001100011100111100100111011001101001011000011100010000
0111101010001001010001010111101011010011000010010101110000000001010001001 01
0010110000000011011011111000111010000001111111101011101000011001111010110001
0111100010101101101101111001011001101110111000110101011101000101011010100000
111110110001000110111100111000111111100101000110011010100001011101101000001 1
0111011110000111000000110010111111010010010110010111111000000111111011100100 0
1111011101000111001110011010111100100010101010001110000000111010001000000100 1
1010011110101011110101111010110101111010001010011001011001100101000110001001
0001110011111110110101000101001101001001011110100110000010001110010100000111
0010100011000101110111000010101110010011100011100100011010111110111101001100
0011111011101001010111001100001001000110110100011010000100010000110111111110
11111101000010010111001101000000100111111000011001011100001110011101 10
Número de zeros: 1023
Número de uns: 1025

p: 8676055919
q: 3371629559
M: 29252446592037309721
Semente: 5573267178

1100111011000101111001000101111110100111011000010010001010111010001100111011
1000001010101110110100011000001100110001001011110011100011100001101010110000
0000010011101111000000110111110110000101010010010110001011111101101010111101
0001010000110110101101101010000100101000000100110101011110011111000011101110
0100100110001100110001101101011100011011001110100011011111110100011101101010
1011011011101111011110001000000001000011000010001010001011100101100100011010
1000000111011001001010000000100000101000101000000000000000001000000111011011010
1110010101010011110101110010100010101110000010101000100100010101001100011101
0110101010101001101010000111110100011010110101011101101011001010110010010101
0011101000000001001101111100100000110011101110001000111000101111101000010001 1
0001110001000111101010100010101111001101111110100100111100011100110100010010
1011111000011111011101111110111110011000011101110000100101100100110011001110 10
0101000101100101011101101101110111100001101110011010100001011101011100000101
0111110010001011011101100001000100100000111000001101111101001100010001001010
0011101001110101100101011101110001101010100110010110000000001010011111100100
0010011001101110000001011001100000100000110000001001100101011010101101100000 0
0100000001101110000010110011111111110000101001110100110000101000011001111100
0100100110100001000111101110111101000100111000100001000011110100001101100000
1111111110010101110100011110001011111101101110011000000100110111101101010001 1
0001100111001010110001000101001110101100000000110101011000110001111100000010
1111100100011011101101001110101000001111011110110101010010100110001111001110
1100110100101101101101110011110100110010100111001010101100110100101000000011
1011110010111100100011000110100100010100101111010100100111110110111110110100
1101011110110001000001100101000000101111101111100111010000000000011100110000
1001101100000110110101001011111110101100110100101111011001100010010110110110
0110001010101101101010110011101010010010001111101011110111110011010010101110
0110001101000001011110111001110001000110001011001001010011010101000110111101
0000011100110000001101111110111011100110101100110111100000000100100011111000
1111010101100011011110000000100111010111001010000111010111111000010000010010 0
1100111000110011011011100100111101111100000010110000010011010000011001101000
0011101110010100111011010000010111010011111101100100101000110110100100011110
0001010100001000110101110110001010101111010101000010001101011000010010110 10
1111100101001110111100000101110010011111001010100110111001001011100110001001 0
0111001100011000100001000010100100010010111111110011010110000010100010001101
1111011010110010110001001001011100001000111000000000100000111001101110100100
1101010100010000101110100110011110100000111000010010000011000000010011001011
0001100110000000101000011101101011011111010111000111111110111010011001011111 11
1000000110000100100110001101010111010001010010110100010110011110011110001000
1010010001010010100101101000101110100110100100100011000011101011011011101000
0110001110001010011110011110011111001101100011110010001010000011101110110011
0100111010001001100011100100001101111010000000100011111011101111101100111010 1
0100110101111111101101110101101100011111010100111101100101010011011101000100
0100101011010011110011111010000000101000010111010010100001101001100011000000
0111001101011101011100111000000101100100111000100000010111101101111001010111

0000110100011011001001011000111100101111100010101011010000001100010001100011
0001111011011100001011101011000101010110011110010100111111111110101110110111 0
1000100111000111010001010111100101111001100101011011010010111101100011110111
0001010100110111110100111100101111101010010011010000011101001101011001101110
1000011110010001001001100011011110100001101111011001101000101000100110101110
0011101011100100111011110100101110001010110011111001100001010001010011010101
0110001111110011111010110110111001001001100101110110101110111010110001111010 0
0111011000100111011010001011110100100101001011111101001000010011000000010111
1100000011111001100100100100101101100010110111000101011010100010010011110110
0100010100111011110011111001000100111110011110100111011000000010
Número de zeros: 2055
Número de uns: 2041

3.2 Linear Feedback Shift Registers (LFSRs)

```python
import time
import matplotlib.pyplot as plt  # Adicionado para gerar o gráfico

def lfsr_galois(bits):
    # Estado inicial com um número grande (dependendo do número de bits)
    estado_inicial = 0xACE1
    lfsr = estado_inicial
    periodo = 0
    numero_gerado = 0

    # Gerar o número pseudo-aleatório com o tamanho desejado
    for i in range(bits):
        lsb = lfsr & 1  # Obtém o bit menos significativo (bit de saída)
        lfsr >>= 1      # Desloca o registrador para a direita

        if lsb == 1:    # Se o bit de saída for 1
            lfsr ^= 0xB400  # Aplica a máscara de realimentação (tap positions)

        # Adiciona o bit gerado ao número final
        numero_gerado = (numero_gerado << 1) | lsb

    return numero_gerado

# Função para medir o tempo de execução
def medir_tempo_lfsr(bits, tentativas=10):
    tempos = []
    for _ in range(tentativas):
        inicio = time.time()
        lfsr_galois(bits)
        fim = time.time()
        tempos.append(fim - inicio)

    tempo_medio = sum(tempos) / len(tempos)
    return tempo_medio

# Testando para diferentes tamanhos de números
tamanhos = [40, 56, 80, 128, 168, 224, 256]
resultados = []

for tamanho in tamanhos:
    tempo_medio = medir_tempo_lfsr(tamanho)
    resultados.append((tamanho, tempo_medio))

# Exibir os resultados em uma tabela
print("Tabela de Resultados LFSR (configuração Galois)")
print("Tamanho do Número (bits) | Tempo Médio (segundos)")
for tamanho, tempo in resultados:
    print(f"{tamanho} | {tempo:.6f} segundos")

###### Explicação do Código ######

# Este código implementa um Registrador de Deslocamento com Realimentação Linear (LFSR) utilizando a configuração Galois, capaz de gerar números pseudo-aleatórios de tamanhos variados.

### Valor inicial (estado_inicial = 0xACE1) ###
# O registrador é inicializado com o valor hexadecimal 0xACE1, equivalente a 44257 em decimal e 1010110011100001 em binário. Este valor deve ser diferente de zero para garantir que o LFSR não entre em um estado estático.

```

```python
### Máscara de realimentação (feedback mask = 0xB400) ###
# A máscara 0xB400 determina as posições dos bits (taps) que serão utilizadas no processo de realimentação. Essa escolha de máscara é típica para LFSRs de 16 bits, garantindo boas propriedades pseudo-aleatórias.

### Processo de geração ###
# A cada iteração, o bit menos significativo (LSB) é extraído.
# O registrador é deslocado uma posição para a direita.
# Caso o bit de saída seja 1, a máscara de realimentação é aplicada via operação XOR.
# O bit gerado é adicionado à esquerda do número pseudo-aleatório final (numero_gerado).

### Função de medição de tempo ###
# A função medir_tempo_lfsr avalia o tempo médio necessário para gerar números de diferentes tamanhos em bits. Para cada tamanho, são realizadas 10 execuções para calcular o tempo médio de geração.

### Tamanhos testados ###
# O algoritmo foi testado para números de 40, 56, 80, 128, 168, 224 e 256 bits. Tamanhos superiores (como 512, 1024, 2048 e 4096 bits) não foram incluídos devido à limitação da largura do estado do LFSR implementado (16 bits), o que impossibilita gerar sequências suficientemente longas sem repetição ou colapsos.

###### Referências Algoritmo ######
# Adapted from:
# https://en.wikipedia.org/wiki/Linear-feedback_shift_register
# All credits to the authors.

###### Geração de Gráfico ######
# Visualização da relação entre o tamanho do número (bits) e o tempo médio de execução.

# Extraindo dados para o gráfico
tamanhos_bits = [r[0] for r in resultados]
tempos_execucao = [r[1] for r in resultados]

# Criando o gráfico
plt.figure(figsize=(10,6))
plt.plot(tamanhos_bits, tempos_execucao, marker='o', linestyle='-', color='g')
plt.title('Tempo de Execução para Geração com LFSR (Configuração Galois)')
plt.xlabel('Tamanho do Número (bits)')
plt.ylabel('Tempo Médio de Execução (segundos)')
plt.grid(True)
plt.xticks(tamanhos_bits)
plt.tight_layout()
plt.show()
```

3.2.1 Saída

Tabela de Resultados LFSR (configuração Galois)

Tamanho do Número (bits) | Tempo Médio (segundos)

40 | 0.000016 segundos

56 | 0.000021 segundos

80 | 0.000031 segundos

128 | 0.000050 segundos

168 | 0.000066 segundos

224 | 0.000088 segundos

256 | 0.000101 segundos

3.3 Teste de Primalidade de Fermat

```python
import random
import time
import pandas as pd
import matplotlib.pyplot as plt

# Função iterativa para calcular (a^n) mod p em tempo O(log n).
# Essa operação é fundamental para testar a primalidade com base no Pequeno Teorema de Fermat.
def potencia(a, n, p):
    resultado = 1
    a = a % p  # Atualiza 'a' para o seu valor módulo 'p', se necessário.

    while n > 0:
        if n % 2:
            # Se n for ímpar, multiplica 'resultado' por 'a' e reduz 'n'
            resultado = (resultado * a) % p
            n = n - 1
        else:
            # Se n for par, eleva 'a' ao quadrado e divide 'n' por 2
            a = (a ** 2) % p
            n = n // 2

    return resultado % p

# Função para testar se um número n é primo utilizando o Pequeno Teorema de Fermat.
# Parâmetros:
# - n: número a ser testado.
# - k: número de iterações para aumentar a confiabilidade do teste.
def eh_primo(n, k):
    # Casos triviais: 1 e 4 são compostos; 2 e 3 são primos.
    if n == 1 or n == 4:
        return False
    elif n == 2 or n == 3:
        return True
    else:
        # Repete o teste 'k' vezes para reduzir a probabilidade de erro.
        for _ in range(k):
            # Escolhe aleatoriamente um inteiro 'a' no intervalo [2, n-2].
            a = random.randint(2, n - 2)
            # Se 'a^(n-1) mod n' for diferente de 1, então 'n' é composto.
            if potencia(a, n - 1, n) != 1:
                return False
        # Se passou em todos os testes, retorna True (provavelmente primo).
        return True

# Função para gerar um número primo com aproximadamente 'bits' bits de tamanho.
# Utiliza o teste de Fermat para validar a primalidade.
def gerar_primo(bits, k=5):
    inicio = time.time()  # Marca o tempo inicial de execução.
    while True:
        # Gera um número aleatório de 'bits' bits e garante que seja ímpar (bit menos significativo igual a 1).
        candidato = random.getrandbits(bits) | 1

        # Testa se o número gerado é primo.
        if eh_primo(candidato, k):
            tempo = time.time() - inicio  # Calcula o tempo decorrido.
            return candidato, tempo

# Lista com os tamanhos de bits desejados para geração dos números primos.
```

```python
tamanhos_bits = [40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096]

# Lista para armazenar os resultados da geração de primos.
resultados = []

# Loop para gerar primos para cada tamanho especificado.
for bits in tamanhos_bits:
    print(f"Gerando número primo de {bits} bits...")
    primo, tempo = gerar_primo(bits)
    resultados.append({
        "Algoritmo": "Fermat",
        "Tamanho do Número (bits)": bits,
        "Número Primo Gerado": primo,
        "Tempo para Gerar (s)": tempo
    })

# Criação da tabela de resultados utilizando a biblioteca pandas.
tabela = pd.DataFrame(resultados)
print("\nTabela de Números Primos Gerados:")
print(tabela)

# Geração de gráfico utilizando a biblioteca matplotlib.
# O gráfico ilustra o tempo de geração em função do tamanho do número (em escala logarítmica).
plt.figure(figsize=(10, 6))
plt.plot(tabela["Tamanho do Número (bits)"], tabela["Tempo para Gerar (s)"], marker='o')
plt.title("Tempo para gerar números primos usando Teste de Fermat")
plt.xlabel("Tamanho do Número (bits)")
plt.ylabel("Tempo (segundos)")
plt.grid(True)
plt.xscale('log')
plt.yscale('log')
plt.show()
# ---------------------------------------------------------------------------
# Pequeno Teorema de Fermat:
# Se n é um número primo, então para todo a, com 1 < a < n-1,
# temos que:
#     a^(n-1) ≡ 1 (mod n)
# ou seja,
#     a^(n-1) % n = 1

# Exemplos:
# - Como 5 é primo, temos que:
#       2^4 ≡ 1 (mod 5),
#       3^4 ≡ 1 (mod 5),
#       4^4 ≡ 1 (mod 5).

# - Como 7 é primo, temos que:
#       2^6 ≡ 1 (mod 7),
#       3^6 ≡ 1 (mod 7),
#       4^6 ≡ 1 (mod 7),
#       5^6 ≡ 1 (mod 7),
#       6^6 ≡ 1 (mod 7).

# Procedimento para testar a primalidade usando o teste de Fermat:
# 1) Repetir k vezes:
#    a) Escolher aleatoriamente um número a no intervalo [2, n-2].
#    b) Se o máximo divisor comum (mdc) de (a, n) for diferente de 1, retornar falso.
```

```
#    c) Se a^(n-1) não for congruente a 1 módulo n, retornar falso.
# 2) Se passar por todos os testes, retornar verdadeiro (provavelmente primo).

### Observações ###:
# - Um valor mais alto de k reduz a probabilidade de erro em números compostos.
# - Para números primos verdadeiros, o teste sempre retorna verdadeiro.
# ---------------------------------------------------------------------------

# This code is contributed by Aanchal Tiwari https://www.geeksforgeeks.org/fermat-method-of-primality-test/

```

3.3.1 Saída

```
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  168954462757022695302784896046445734889336902...              0.005407
5     Fermat                       224  206093681651418649195475713515232218012023024...              0.018186
6     Fermat                       256  459860096351635228205907801163108599648258170...              0.013081
7     Fermat                       512  892777602258888708793983337668912971104720558...              0.260057
8     Fermat                      1024  164975531557893065518140519802589927799293354...              0.734602
9     Fermat                      2048  179390311340858908790323226172807279690769518...             12.712863
10    Fermat                      4096  180398052425008861946503235386718038186144369...            140.994728
```

3.4 Teste de Primalidade de Miller-Rabin

```python
import random
import time
import pandas as pd
import matplotlib.pyplot as plt

# Função para fazer a exponenciação modular
# Retorna (x^y) % p
def potenciacao_modular(x, y, p):
    resultado = 1
    x = x % p  # Atualiza x se for maior que p
    while y > 0:
        if (y & 1):  # Se y for ímpar, multiplica x pelo resultado
            resultado = (resultado * x) % p
        y = y >> 1  # y = y // 2
        x = (x * x) % p
    return resultado

# Função que executa o teste de Miller
# Retorna False se n for composto e True se n for provavelmente primo
def teste_miller(d, n):
    a = 2 + random.randint(1, n - 4)  # Escolhe um 'a' aleatório no intervalo [2, n-2]
    x = potenciacao_modular(a, d, n)
    if x == 1 or x == n - 1:
        return True
    while d != n - 1:
        x = (x * x) % n
        d *= 2
        if x == 1:
            return False
        if x == n - 1:
            return True
    return False

# Função principal que aplica o Teste de Miller-Rabin
# Retorna False se n é composto, True se n é provavelmente primo
def eh_primo(n, k):
    if n <= 1 or n == 4:
        return False
    if n <= 3:
        return True
    d = n - 1
    while d % 2 == 0:
        d //= 2
    for _ in range(k):
        if not teste_miller(d, n):
            return False
    return True

# Função para gerar um número primo de 'bits' bits
def gerar_primo(bits, k=10):
    while True:
        # Gera número ímpar aleatório
        numero = random.getrandbits(bits) | 1 | (1 << (bits - 1))
        if eh_primo(numero, k):
            return numero

# Gerar a tabela de primos e tempos
bits_teste = [40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096]
tabela_resultados = []
```

```
for bits in bits_teste:
    inicio = time.time()
    try:
        primo = gerar_primo(bits)
        fim = time.time()
        tempo = fim - inicio
    except Exception as e:
        primo = None
        tempo = None
    tabela_resultados.append({
        "Algoritmo": "Miller-Rabin",
        "Tamanho do Número (bits)": bits,
        "Número Primo Gerado": primo,
        "Tempo para Gerar (s)": tempo
    })

# Mostrar tabela
df = pd.DataFrame(tabela_resultados)
print(df)

# Plotar gráfico
plt.figure(figsize=(10,6))
plt.plot(df["Tamanho do Número (bits)"], df["Tempo para Gerar (s)"], marker='o')
plt.title("Tempo para Gerar Números Primos usando Miller-Rabin")
plt.xlabel("Tamanho do Número (bits)")
plt.ylabel("Tempo para Gerar (s)")
plt.grid(True)
plt.show()

# ----------------------------------------------------------------------------------
####### Explicação do Algoritmo de Miller-Rabin: #######

### Função eh_primo(n, k): ###
# 1) Trata os casos base para n < 3.
# 2) Se n é par, retorna False.
# 3) Encontra um número ímpar d tal que n-1 possa ser escrito como d*2^r.
#     Como n é ímpar, n-1 é par e r deve ser maior que 0.
# 4) Executa k iterações:
#     Se teste_miller(d, n) retornar False, retorna False imediatamente.
# 5) Se todas as iterações passarem, retorna True.

#Função teste_miller(d, n):
# 1) Escolhe um número aleatório 'a' no intervalo [2, n-2].
# 2) Calcula x = a^d % n.
# 3) Se x == 1 ou x == n-1, retorna True.
# 4) Caso contrário:
#     Enquanto d não chegar a n-1:
#      a) Atualiza x = (x*x) % n.
#      b) Se x == 1, retorna False.
#      c) Se x == n-1, retorna True.
# 5) Se o laço terminar, retorna False.

"""
Exemplo de Execução (n = 13, k = 2):
- Encontramos d = 3, r = 2, pois 13-1 = 12 = 3*2^2.
- Primeira iteração:
    Escolhemos a = 4, calculamos x = 4^3 % 13 = 12.
    Como x = n-1, retornamos True.
```

```
- Segunda iteração:
    Escolhemos a = 5, calculamos x = 5^3 % 13 = 8.
    Como x não é 1 nem n-1:
    - Calculamos x = (8*8) % 13 = 12.
    - x agora é n-1, então retornamos True.
- Como ambas as iterações retornaram True, concluímos que 13 é provavelmente primo.
"""
# ------------------------------------------------------------------------------------

### Observações ###:
# - A função eh_primo aplica o teste múltiplas vezes para aumentar a confiança.
# - A geração de números primos maiores (especialmente acima de 1024 bits) pode demorar bastante (provavelmente devido à baixa "densidade" de primos nessa faixa).
# - Para números de 2048 e 4096 bits, a geração levou vários minutos.
# - Nem sempre o primeiro número aleatório gerado é primo, então vários testes foram necessários.

# ------------------------------------------------------------------------------------

# This code is contributed by mits https://www.geeksforgeeks.org/fermat-method-of-primality-test/

```

3.4.1 Saída

	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	1067603260326893220919343	0.000948
3	Miller-Rabin	128	311816980637365528824348574735179769973	0.001957
4	Miller-Rabin	168	3395047087587206621733791939035192393636789248...	0.005254
5	Miller-Rabin	224	1897486386837764511644979710936435869954119681...	0.014418
6	Miller-Rabin	256	5981853771364244131609468018054835638814282147...	0.003869
7	Miller-Rabin	512	8600154951185793705660628715827319454088606900...	0.350892
8	Miller-Rabin	1024	1086902811351158492636524228056442510737367305...	5.097350
9	Miller-Rabin	2048	2336591355933292909372247880912150132686327487...	0.379537
10	Miller-Rabin	4096	7346564163771396142397970846760492699467575724...	402.471444

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.