Campus de Florianópolis

Relatório do Trabalho Individual sobre Números Primos

Segurança em Computação (INE5429)

João Paulo A. Bonomo (21100133)

Florianópolis - SC

Abril, 2025

Relatório do Trabalho Individual sobre Números Primos

Segurança em Computação (INE5429)

Professor Jean Everson Martina

Professora Thaís Bardini Idalino

Sumário

1 Introdução

Essa Capítulo aborda a descrição do problema proposto, bem como sumariza a abordagem escolhida pelo estudante para desenvolver o trabalho, além de introduzir os capítulos seguintes.

1.1 Descrição do Problema Proposto

A descrição do problema proposto foi disponibilizada via Moodle para os estudantes e tem o seguinte texto introdutório, que é acompanhado por uma lista de passos a serem seguidos para cumprir o trabalho adequadamente:

"Números inteiros maiores que um são ditos primos se seus únicos divisores são 1 e o próprio número. Em segurança computacional utilizamos números primos em várias algoritmos e protocolos. Para isso, é necessário manter-se uma tabela de números primos (uma lista pré computada) ou, gerar tais números quando necessário.

"Não é simples a geração de números primos para uso em sistemas de segurança computacional. Normalmente, estamos interessados em números grandes, da ordem de grandeza de centenas de dígitos. No Brasil, por exemplo, para assinar documentos eletrônicos, você vai precisar ter chaves criptográficas geradas a partir de números primos de 2048 bits.

"Uma forma de se gerar números primos é primeiro gerar um número aleatório ímpar (grande) e depois testá-lo para saber se é primo. Caso não seja, gera-se outro número aleatório até que seja primo.

"Portanto, temos duas tarefas para fazer:

- Gerar números aleatórios;
- Testar se esse número gerado é primo.

"Neste trabalho individual, vamos explorar técnicas para se gerar números pseudo-aleatórios e para se verificar a primalidade desses números".

O aluno tinha a liberdade de escolher dois dos algoritmos da lista abaixo para gerar números pseudo-aleatórios:

- Blum Blum Shub;
- Complementary-multiply-with-carry;

- Inversive congruential generator;
- ISAAC (cipher);
- Lagged Fibonacci generator;
- Linear congruential generator;
- Linear feedback shift register;
- Maximal periodic reciprocals;
- Mersenne twister;
- Multiply-with-carry;
- Naor-Reingold Pseudorandom Function;
- Park–Miller random number generator;
- Well Equidistributed Long-period Linear e;
- Xorshift.

O aluno deveria utilizar os dois algoritmos escolhidos para gerar números pseudo-aleatórios na ordem de grandeza de 40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048 e 4096 bits binários e comparar o tempo de geração obtido em cada um deles e fazer uma análise de complexidade dos algoritmos para justificar o tempo.

Além disso, o estudante deveria implementar dois métodos para verificar a primalidade dos números gerados, sendo um deles o Teste de Primalidade de Miller-Rabin e o segundo ficando à sua escolha, porém com três sugestões:

- Teste de Primalidade de Fermat;
- Solovay-Strassem e;
- Frobenius.

O estudante deveria gerar uma tabela com números primos de 40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048 e 4096 bits, usando como entrada os números aleatórios gerados no passo anterior. Se necessário, gerando mais números ou usando a estratégia de incrementar o número ímpar gerado.

Por fim, o aluno também deveria discutir o tempo de geração obtido e fazer uma análise de complexidade dos dois algoritmos, assim como feito para os geradores de números pseudo-aleatórios.

1.2 Abordagem

Para o problema proposto, foram escolhidos os algoritmos **Blum Blum Shub (BBS)** e **Lagged Fibonacci Generator (LFB)** para a geração de números pseudo-aleatórios e, em adição ao Miller-Rabin, foi escolhido o **Teste de Primalidade de Fermat** para a verificação de primalidade dos números obtidos através dos dois geradores escolhidos. A escolha dos dois geradores se justifica por querer utilizar duas soluções diferentes, sendo a primeira delas um algoritmo criptograficamente seguro, o BBS, e a segunda não, o LFB. Outro critério de escolha foi a facilidade em encontrar informações a respeito dos algoritmos na internet e a simplicidade de sua aplicação. O BBS, além de ser um algoritmo criptograficamente seguro, também destaca-se por sua fácil implementação (BLUM; BLUM; SHUB, 1986). Enquanto o LFB tem como ponto positivo o seu uso em simulações (LAW; KELTON, 2006). Por fim, a escolha do Teste de Primalidade de Femart para acompanhar o Miller-Rabin se deu pelo interesse do aluno em conhecê-lo melhor, visto que já tinha ouvido falar sobre o teste, porém nunca de forma profunda e experimental.

As implementações foram desenvolvidas com a linguagem de programação **Go**, visando um tempo de execução baixo, porém sem deixar de utilizar uma linguagem moderna. O aluno optou por não incluir o código fonte no documento do relatório, porém o repositório público no GitHub está "*linkado*"no Apêndice A, seguindo as recomendações de estar devidamente documentado utlizando comentários no corpo do código. Além disso, os números pseudo-aleatórios e primos usados para a experimentação e análises deste relatório também estão inclusos no repositório.

O restante do documento está organizado da seguinte forma:

- Capítulo 2: aborda a descrição dos geradores de números pseudo-aleatórios escolhidos, a experimentação e a análise de seu uso;
- Capítulo 3: traz a descrição dos geradores de números pseudo-aleatórios escolhidos, a experimentação e a análise de seu uso;
- Capítulo 4: descreve cas onsiderações finais e aconclusão do trabalho;
- Referências: lista as citações e referências utilizadas como base para o desenvolvimento teórico e prático da solução e;
- Apêndice A: traz o código fonte da solução.

2 Números Aleatórios

Este Capítulo descreve os dois algoritmos geradores de números pseudo-aleatórios utilizados no trabalho, bem como os experimentos e a análise de seus usos.

2.1 Blum Blum Shub

Blum Blum Shub (BBS) é um algoritmo utilizado para gerar números pseudoaleatórios — já que números que aparentam ser aleatórios, mas que são determinados a partir de uma semente inicial. O método foi desenvolvido em 1986 por Lenore Blum, Manuel Blum e Michael Shub (daí vem o nome do algoritmo, do sobrenome dos autores).

O funcionamento do BBS baseia-se na fórmula abaixo:

$$x_{n+1} = x_n^2 \bmod M$$

onde x_0 é a semente aleatória. O valor de M é calculado como o produto de dois números primos p e q, ou seja, $M = p \times q$, sendo que ambos satisfazem a congruência:

$$p \equiv q \equiv 3 \pmod 4$$

A segurança do BBS está relacionada à dificuldade de fatorar M. Em cada iteração, é comum extrair apenas parte do número gerado, geralmente o bit menos significativo de x_{n+1} (BLUM; BLUM; SHUB, 1986).

Embora o algoritmo seja relativamente lento em comparação a outros geradores de números pseudoaleatórios, é considerado um dos mais seguros do ponto de vista teórico. Além disso, embora o BBS não seja recomendado para uso direto em cifras, ele é adequado para a geração de chaves criptográficas (BUCHANAN, 2025b).

2.2 Lagged Fibonacci Generator

O **Lagged Fibonacci Generator (LFG)** é um gerador de números pseudoaleatórios que generaliza a sequência de Fibonacci tradicional. Em vez de somar apenas os dois termos anteriores, o LFG utiliza dois termos anteriores com atrasos específicos, aplicando uma operação definida e um módulo (BUCHANAN, 2025a).

A fórmula geral do LFG é:

$$S_n = (S_{n-j} \star S_{n-k}) \bmod m$$

onde:

- S_n é o novo número gerado,
- j e k são os atrasos (lags), com $0 < j < k$,
- $\star$ representa uma operação binária, como adição ($+$), subtração ($-$), multiplicação ($\times$) ou XOR ($\oplus$),
- m é o módulo utilizado para limitar o valor gerado.

Por exemplo, com $j = 3$, $k = 7$, uma semente inicial de 6421893 e $m = 10$, o LFG gera uma sequência pseudoaleatória ao aplicar a operação de adição entre os termos atrasados e o módulo 10. A sequência resultante seria:

$$5, 6, 4, 3, 6, 1, 7, 1, 4, 0, 1, 8, 9, 3, 3, 4, 2, 1, 4, 7$$

Um exemplo de utilização do LFG são as aplicações em jogos, como o jogo *Freeciv*, com parâmetros $j = 24$ e $k = 55$ (BUCHANAN, 2025a).

É importante destacar que, embora o LFG seja eficiente para gerar números pseudoaleatórios, ele não é adequado para aplicações criptográficas, pois não oferece segurança suficiente contra ataques que visam prever ou reconstruir a sequência gerada.

Curiosamente, o gerador de números pseudoaleatórios padrão da linguagem de programação Go é baseado no Lagged Fibonacci Generator. Como dito anteriormente, o LFG é inadequado para aplicações de segurança, pois seu estado interno pode ser reconstruído a partir de poucas saídas observadas. Embora a própria documentação do Go alerte sobre isso, alguns sistemas ainda o utilizam de forma incorreta, expondo-se a ataques. A vulnerabilidade permite prever futuros números gerados, comprometendo a segurança de sistemas que dependem dele (KATZ, 2024).

2.3 Experimentos

Descritos os dois algoritmos selecionados, passamos para a seção de experimentos realizados com os geradores.

Os dois algoritmos foram utilizados para gerarem 10 números para cada uma das ordens especificadas na descrição do problema (40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048 e 4096 bits). Os resultados detalhados das gerações, como os números em base

N bits	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8	i=9	i=10	AVG
40	0.11	0.32	0.26	0.32	0.14	0.31	0.11	0.18	0.11	0.19	0.205
56	0.14	0.72	0.50	0.14	0.18	0.44	0.25	0.26	0.18	0.37	0.318
80	0.18	0.66	0.11	1.05	0.61	0.93	0.42	0.27	0.40	0.51	0.514
128	0.40	0.72	0.73	0.87	0.12	0.89	0.63	0.25	0.42	0.59	0.562
168	0.96	1.80	0.85	3.74	0.71	0.84	0.89	0.98	1.27	1.88	1.492
224	3.17	0.45	1.83	0.84	0.71	3.95	2.59	1.56	0.97	0.34	1.641
256	0.83	1.13	1.12	1.78	0.68	3.22	1.06	0.97	1.32	1.14	1.325
512	3.44	4.04	3.87	2.58	2.09	1.26	4.53	3.37	1.93	7.85	3.696
1024	10.8	21.7	9.04	32.13	4.77	23.7	17.07	48.4	56.5	8.56	23.06
2048	49.2	126.7	84.8	379	209.8	136.6	80.03	128.8	100.6	48.2	134.573
4096	1446.7	1516.3	814.8	2012.4	3885.1	2067.9	2607.9	2489.5	3960.2	1936.3	2283.71

Tabela 1 – Tempo para gerar números pseudo-aleatórios com o **Blum Blum Shub** em **microssegundos**. OBS: "i"indica apenas o número da instância, "AVG"indica o tempo médio de geração e "N bits"é o número de bits.

N bits	i=1	i=2	i=3	i=4	i=5	i=6	i=7	i=8	i=9	i=10	AVG
40	267	227	234	308	2145	212	349	232	206	429	461.9
56	196	207	208	187	197	180	205	202	220	212	201.4
80	335	447	240	206	249	333	175	201	280	251	271.7
128	170	233	198	189	185	181	365	212	108	219	206.0
168	339	407	362	345	188	356	194	360	226	202	297.9
224	175	436	195	363	379	376	211	365	113	211	282.4
256	350	386	193	186	188	192	189	203	193	214	219.4
512	432	197	469	234	176	187	190	475	197	424	338.1
1024	169	581	197	636	623	180	613	606	118	460	418.3
2048	892	938	944	189	228	942	231	206	129	251	495.0
4096	1543	312	269	1684	278	1582	298	2821	1313	1208	1330.8

Tabela 2 – Tempo para gerar números pseudo-aleatórios com o **Lagged Fibonacci Generator** em **nanossegundos**. OBS: "i"indica apenas o número da instância, "AVG"indica o tempo médio de geração e "N bits"é o número de bits.

decimal e binária, podem ser vistos no repositório público no GitHub (Apêndice A), porém estão sumarizados nas Tabelas 2.3 e 2.3 com as informações solicitadas na descrição do problema.

2.4 Análise dos Resultados

Ao compararmos os tempos de geração dos números pseudo-aleatórios obtidos para o LFG e para o BBS, observa-se uma diferença significativa no desempenho de ambos os algoritmos. A primeira tabela apresenta os resultados do LFG, com tempos de geração medidos em nanossegundos, enquanto a segunda tabela mostra os tempos do BBS, medidos em microssegundos, lembrando que 1 microssegundo equivale a 1.000 nanossegundos.

Analisando os dados, é evidente que o LFG é consideravelmente mais rápido. Por exemplo, para 128 bits, o tempo médio de geração usando o LFG é da ordem de 166 nanossegundos, enquanto o BBS leva aproximadamente 590 microssegundos, o que equivale

a 590.000 nanossegundos — ou seja, o BBS é cerca de 3.550 vezes mais lento neste caso. A diferença de desempenho se acentua conforme o número de bits aumenta: para 512 bits, o LFG apresenta um tempo médio de cerca de 660 nanossegundos, enquanto o BBS leva 3.280 microssegundos (3.280.000 nanossegundos), sendo aproximadamente 5.000 vezes mais lento. Para 4096 bits, a diferença é ainda mais expressiva, com o LFG em torno de 6.112 nanossegundos e o BBS alcançando cerca de 2.190.000.000 nanossegundos.

Esses resultados são coerentes com a complexidade dos algoritmos envolvidos. O Lagged Fibonacci Generator possui uma complexidade computacional relativamente baixa, próxima de $O(1)$ por número gerado, pois depende apenas de operações aritméticas simples (adição e deslocamento de elementos em um buffer de números previamente gerados). Isso o torna extremamente eficiente para gerar números pseudo-aleatórios rapidamente. Esse é um dos motivos para que linguagens de programação, como Go, usam o LFG como padrão para a geração de números aleatórios (KATZ, 2024).

Por outro lado, o Blum Blum Shub é baseado em operações de aritmética modular envolvendo números grandes, incluindo multiplicações e operações de módulo com números primos de grandes dimensões. Essas operações possuem complexidade bem maior, tipicamente $O(n^2)$ ou até mais dependendo da implementação (considerando n como o número de bits), especialmente para operações de módulo e exponenciação modular. Como consequência, o BBS é muito mais lento, mas, em compensação, ele oferece propriedades de segurança criptográfica muito superiores ao LFG, sendo considerado um gerador de número pseudo-aleatório criptograficamente seguro - CSPRNG, na sigla em inglês (BUCHANAN, 2025b).

Portanto, a diferença de tempo observada entre os dois geradores se explica tanto pelo tipo de operações realizadas quanto pela complexidade dos algoritmos: enquanto o LFG prioriza velocidade, o BBS prioriza segurança. Sendo assim, em amplicações que visam velocidade e a segurança do número gerado não é tão importante, como simulações, o LFG certamente se sobressai (LAW; KELTON, 2006). Porém, em aplicações de segurança, como criptografia, o BBS é a escolha ideal entre os dois, mesmo com um tempo de geração muito superior ao LFG.

3 Números Primos

Este Capítulo é destinado a descrever os dois testes utilizados para verificar a primalidade dos números gerados pelos algoritmos descritos e analisados no Capítulo 2. Ambos os testes são baseados no Teorema de Fermat, embora tenham suas particularidades. A seguir, abordaremos uma descrição mais formal de cada um dos dois testes, os experimentos realizados com eles e uma análise dos resultados.

3.1 Teste de Primalidade de Fermat

O Teste de Primalidade de Fermat é um teste probabilístico usado para identificar se um número é ou não primo. Baseia-se no Teorema de Fermat, que afirma que se p é um número primo e a é um número inteiro tal que $1 < a < p - 1$, então:

$$a^{p-1} \equiv 1 \pmod{p}$$

O teste funciona da seguinte maneira:

1. Escolha um número inteiro aleatório a no intervalo $[2, p-2]$.
2. Calcule $a^{p-1} \mod p$.
3. Se o resultado for diferente de 1, então p é composto.
4. Caso contrário, repita o teste para diferentes valores de a para aumentar a confiança de que p é primo.

Este teste, porém, pode identificar erroneamente alguns números compostos como primos. Esses números são conhecidos como **números de Carmichael**, que satisfazem a congruência para todos os a coprimos com p. Portanto, o Teste de Primalidade Fermat é um teste de primalidade eficiente, embora não seja completamente seguro (GeeksforGeeks, 2023).

3.2 Teste de Miller-Rabin

O teste de Miller-Rabin é um algoritmo probabilístico para verificar a primalidade de um número. Ele, assim como o Teste de Primalidade de Fermat descrito acima, é baseado no Teorema de Fermat e é utilizado para verificar se um número é *provavelmente primo* ou *composto* (GeeksforGeeks, 2018).

Dado um número ímpar $n > 3$, o procedimento do teste é o seguinte:

1. Representar $n - 1$ na forma:
$$n - 1 = 2^r \times d$$
onde d é um número ímpar e $r \geq 1$. Para isso, divide-se sucessivamente $n - 1$ por 2 até obter um valor ímpar.

2. Escolher aleatoriamente uma base a tal que:
$$2 \leq a \leq n - 2$$

3. Calcular:
$$x = a^d \mod n$$
   - Se $x = 1$ ou $x = n - 1$, o número n passa neste teste para a base a.
   - Caso contrário, repetir até $r - 1$ vezes:
     - Calcular:
     $$x = x^2 \mod n$$
     - Se em algum momento $x = n - 1$, o número n passa no teste para esta base.
   - Se x nunca se tornar $n - 1$, então n é declarado composto.

4. Repetir o processo para diferentes valores de a para aumentar a confiança de que n é primo.

O Teste de Miller-Rabin não garante com certeza a primalidade de n em uma única execução, mas quanto mais iterações forem feitas com diferentes bases, maior a probabilidade de identificação correta (GeeksforGeeks, 2018).

3.3 Experimentos

Para essa experimentação, os números iniciais gerados pelos algoritmos geradores de números pseudoaleatórios (descritos no Capítulo 2) servem como candidato base para o Teste de Primalidade de Fermat e para o Teste de Miller-Rabin. Este candidato é então ajustado para garantir que possua o número exato de bits requeridos e que seja ímpar (condição necessária para primalidade, exceto para o número 2). O algoritmo então verifica se este candidato é provavelmente primo através de múltiplas iterações, onde cada iteração seleciona aleatoriamente um valor de teste 'a' e realiza a verificação. Caso o candidato falhe em qualquer iteração, o algoritmo incrementa o valor em 2 (preservando a condição de imparidade) e inicia novamente o processo de teste. O número de iterações do teste é

adaptado dinamicamente conforme o tamanho do número, sendo 20 para números menores que 256 bits, 30 para números entre 256 e 1024 bits, e 40 para números maiores que 1024 bits, fornecendo assim maior confiabilidade para números de maior magnitude. Mais detalhes da implementação pode ser vistos no código fonte, disponível no Apêndice A.

Número de Bits	**Miller-Rabin**	**Tempo Fermat**
40	0.07	0.04
56	0.08	0.05
80	0.25	0.08
128	0.48	0.09
168	0.71	0.16
224	1.34	0.27
256	2.37	0.32
512	10.51	1.88
1024	149.43	9.81
2048	2340.19	88.37
4096	5171.98	753.1

Tabela 3 – Comparação de Tempo Médio de Execução entre o Teste Miller-Rabin e o Teste de Primalidade Fermat para Geração de Números Primos com os números gerados pelo BBS como candidatos iniciais em milissegundos (ms).

A Tabela 3.3 traz a comparação do tempo de execução médio entre os dois testes aplicados aos números gerados no Capítulo 2. É válido ressaltar que o tempo médio anotado na tabela também considera o tempo levado por aquele algoritmo para gerar um número primo com base no número pseudo-aleatório fornecido como candidato. Os resultados completos podem ser encontrados no repositório do trabalho, no Apêndice A.

3.4 Análise dos Resultados

Do ponto de vista teórico, essa diferença de desempenho é esperada devido às características computacionais de cada teste. O Teste de Primalidade de Fermat, para cada base a, realiza essencialmente **uma única exponenciação modular**, cuja complexidade é $O(\log p)$, onde p é o número candidato (KUMAR, 2024).

Já o Teste de Miller-Rabin, além da exponenciação inicial, pode executar até $r-1$ quadraturas modulares adicionais (no pior caso), onde r é tal que $n-1 = 2^r \times d$. Assim, a complexidade de uma rodada de Miller-Rabin fica em torno de $O(r \log p)$, que pode ser interpretada aproximadamente como $O(\log^2 p)$, considerando o tamanho dos números grandes e a profundidade dos testes (GeeksforGeeks, 2018; GeeksforGeeks, 2023).

Além disso, Miller-Rabin é inerentemente mais robusto, pois busca detecções compostas em múltiplos níveis internos (a partir das sucessivas quadraturas), enquanto Fermat faz uma verificação direta baseada em uma única condição. Essa maior robustez

do MR tem um custo de tempo computacional que se torna mais pronunciado conforme o número de bits dos candidatos cresce, como pode ser observado para candidatos de 1024 bits ou mais, onde o tempo de execução do Miller-Rabin supera em mais de 10 vezes o tempo do teste de Fermat (GeeksforGeeks, 2018; GeeksforGeeks, 2023).

Curiosamente, embora nenhum dos candidatos, ou seja, os números gerados com o BBS e com o LFG, tenha sido considerado primo, Fermat e Miller-Rabin convergiram para os mesmo números primos usando cada um de nossos candidatos. Tendo isso em consideração e analisando a , o simples Teste de Primalidade de Fermat se mostra mais eficiente que o Miller-Rabin (MR) na geração de um número primo para um mesmo candidato. Embora o MR geralmente tenha uma resposta mais segura quanto à primalidade um número, ambos chegando em uma mesma resposta, a decisão de um dos dois algoritmos para uma aplicação poderia basear-se quase que exclusivamente pela eficiência.

4 Conclusão

Ao longo deste trabalho, foram analisados e comparados dois algoritmos voltados à geração de números pseudo-aleatórios e dois voltados à verificação de primalidade, ambos fundamentais para aplicações criptográficas e sistemas que exigem segurança e imprevisibilidade.

Na comparação entre os geradores de números pseudoaleatórios, o algoritmo **Blum Blum Shub** (BBS) demonstrou-se mais seguro do ponto de vista criptográfico, sendo baseado em problemas de difícil resolução, como a fatoração de grandes números compostos. Essa segurança, no entanto, vem acompanhada de um custo computacional elevado, tornando o BBS consideravelmente mais lento em relação ao gerador **Lagged Fibonacci Generator** (LFG). O LFG, por sua vez, apresentou uma performance significativamente superior em termos de velocidade, sendo apropriado para aplicações que demandam grande volume de números aleatórios, mas onde a segurança não precisa ser a principal preocupação.

Quanto aos testes de primalidade, observou-se que o **Teste de Primalidade de Fermat** é mais eficiente que o **Teste de Miller-Rabin** (MR) na geração de primos a partir de candidatos aleatórios, principalmente em termos de tempo de execução. Do ponto de vista computacional, essa diferença é esperada devido à maior quantidade de operações internas no algoritmo de MR. Embora o Teste de Fermat apresente maior vulnerabilidade a falsos positivos, sua simplicidade e velocidade podem ser vantajosas em cenários de pré-filtragem de candidatos.

Portanto, enquanto para aplicações críticas de segurança criptográfica a robustez do BSS e do Teste de Miller-Rabin justificam seus custos adicionais, em contextos onde uma pequena margem de erro pode ser tolerada, ou onde a eficiência é prioritária — como na geração massiva de números candidatos para posterior validação adicional — a combinação de LFG e Teste de Fermat pode se mostrar uma escolha mais apropriada.

Referências

BLUM, L.; BLUM, M.; SHUB, M. A simple unpredictable pseudo-random number generator. *SIAM Journal on Computing*, v. 15, n. 2, p. 364–383, 1986.

BUCHANAN, W. J. *Blum Blum Shub*. 2025. <https://asecuritysite.com/encryption/blum>. Acesso em: 26 abr. 2025.

BUCHANAN, W. J. *Lagged Fibonacci Generator*. 2025. <https://asecuritysite.com/encryption/fab>. Acesso em: 27 abr. 2025.

GeeksforGeeks. *Primality Test | Set 3 (Miller–Rabin)*. 2018. <https://www.geeksforgeeks.org/primality-test-set-3-miller-rabin/>. Acesso em: 26 abr. 2025.

GeeksforGeeks. *Fermat Method of Primality Test*. 2023. <https://www.geeksforgeeks.org/fermat-method-of-primality-test/>. Acesso em: 27 abr. 2025.

KATZ, D. *Attacking Go's Lagged Fibonacci Generator*. 2024. <https://www.leviathansecurity.com/blog/attacking-gos-lagged-fibonacci-generator>. Acesso em: 26 abr. 2025.

KUMAR, A. *Primality Test*. 2024. <https://www.scaler.in/primality-test/>. Acesso em: 28 abr. 2025.

LAW, A. M.; KELTON, W. D. *Simulation Modeling and Analysis*. 4. ed. [S.l.]: McGraw-Hill, 2006.

APÊNDICE A – Código Fonte

O código fonte do trabalho, bem como o *script* de execução dos testes e os resultados obtidos (salvos em arquivos *.txt*) podem ser encontrados em um repositório público no perfil do aluno no GitHub, acessível em: <https://github.com/BonomoJoaoPaulo/PrimeNumGenerator>. O *README.md* do repositório descreve melhor a organização do projeto e como executar o código.