UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

José Daniel Alves do Prado

Título: Números Primos

Florianópolis/SC
2024

José Daniel Alves do Prado

Título: Números Primos

Trabalho individual 03 Números Primos submetido ao curso Graduação em Ciências da Computação da Universidade Federal de Santa Catarina como requisito parcial para avaliação da disciplina INE5429 – Segurança em Computação.

Orientador(a): Prof. Dr. Jean Everson Martina e Profa. Dra. Thaís Bardini Idalino.

Florianópolis/SC

2024

SUMÁRIO

IDENTIFICAÇÃO

NOME

1. José Daniel Alves do Prado

MATRÍCULA

1. 20103689

TURMA

1. 07208

ENDEREÇO ELETRÔNICO

1. daniel.prado@grad.ufsc.br

CONFIGURAÇÃO DO NOTEBOOK

Modelo do hardware: Positivo Bahia - VAIO VJFE49F11X-B0111H

Memória: 32,0 GiB

Processador: AMD® Ryzen 5 5500u with radeon graphics × 12

Gráficos: RENOIR (renoir, LLVM 15.0.7, DRM 3.54, 6.5.0-35-generic)

Capacidade de disco: SSD 256,1 GB

Nome do SO: Ubuntu 22.04.4 LTS

Tipo do SO: 64 bits

Versão do GNOME: 42.9

Sistema de janelas: Wayland

LINK DO GITHUB

https://github.com/jdanprad0/INE5429-Seguranca-em-Computacao/tree/trabalho-individual-3

1 DESCRIÇÃO E IMPLEMENTAÇÃO DOS ALGORITMOS ESCOLHIDOS

Para este relatório, foi escolhido dois algoritmos geradores de números pseudo-aleatórios: Blum Blum Shub (BBS) e Linear Congruential Generator (LCG). Foi utilizado Python para a implementação devido à sua simplicidade e familiaridade.

1.1 ALGORITMO BLUM BLUM SHUB (BBS)

O Blum Blum Shub é um gerador de números pseudo-aleatórios criptograficamente seguro, introduzido por Lenore Blum, Manuel Blum e Michael Shub em 1986. Ele se baseia em propriedades de números primos e quadrados, e sua segurança se deriva da dificuldade de fatoração de grandes números [1].

$$X_{n+1} = (X_n)^2 \ mod\ m \ [2]$$

Passos do Algoritmo [3]:

1. Escolha dois grandes números primos p e q onde $p \equiv q \equiv 3 \ mod\ 4$.
2. Calcule $m = p\ x\ q$.
3. Escolha um inteiro s (semente) inicial, em que $s \in QR(m)$ onde $QR(m) = \{\forall a \in Z, \exists x \in Z \mid x^2 \equiv a \ mod\ m\}$.
4. Calcule $X_0 = s^2 \ mod\ m$.
5. Para gerar um bit, calcule $X_{n+1} = (X_n)^2 \ mod\ m$ e o bit gerado é $x_{n+1} \ mod\ 2$. O resultado da operação é composto pelo quadrado do resultado anterior já calculado.

Implementação em Python:

```python
class BlumBlumShub:
    """
    Implementação do algoritmo Blum Blum Shub (BBS) para geração de números pseudo-aleatórios.

    Parâmetros:
    p (int): Um número primo onde p ≡ 3 (mod 4).
    q (int): Um número primo onde q ≡ 3 (mod 4).
    s (int): Semente inicial.
    """
    def __init__(self, p: int, q: int, s: int):
        self.p = p
        self.q = q
        self.m = p * q # Calcular m = p * q
        self.s = s


    def generate(self, num_bits: int) → int:
        """
        Gera um número pseudo-aleatório de tamanho num_bits.

        Parâmetros:
        num_bits (int): Número de bits do número pseudo-aleatório gerado.

        Retorna:
        int: Um número pseudo-aleatório de tamanho num_bits.
        """
        x = (self.s * self.s) % self.m # Calcular x0 = s^2 mod m
        result = 0
        for i in range(num_bits):
            x = (x * x) % self.m # Calcular xi+1 = xi^2 mod m
            bit = x % 2 # Extrair o bit menos significativo de x
            result = (result << 1) | bit # Construir o número resultante bit a bit
        return result
```

1.2 ALGORITMO LINEAR CONGRUENTIAL GENERATOR (LCG)

O LCG é um dos métodos mais simples e populares para gerar sequências de números pseudo-aleatórios. A fórmula básica é:

$$X_{n+1} = (a \, x \, X_n + c) \, mod \, m \, [4]$$

Passos do Algoritmo [4]:

1. Escolha o módulo m | $m > 0$.
2. Escolha o multiplicador a | $0 < a < m$.
3. Escolha o incremento c | $0 \leq c < m$.
4. Escolha um valor para a semente inicial X_0 | $0 \leq X_0 < m$.
5. Para gerar um bit, calcule $X_{n+1} = (X_n)^2 \, mod \, m$ e o bit gerado é $x_{n+1} \, mod \, 2$. O resultado da operação é composto pelo resultado anterior já calculado, multiplicado pelo seu respectivo multiplicador adicionado ao incremento.

Implementação em Python:

```python
class LinearCongruential:
    """
    Implementação do gerador de números pseudo-aleatórios de congruência linear (LCG).

    Parâmetros:
    a (int): Multiplicador.
    c (int): Incremento.
    m (int): Módulo.
    seed (int): Semente inicial.
    """
    def __init__(self, a: int, c: int, m: int, seed: int):
        self.a = a
        self.c = c
        self.m = m
        self.seed = seed


    def generate(self, num_bits: int) → int:
        """
        Gera um número pseudo-aleatório de tamanho num_bits.

        Parâmetros:
        num_bits (int): Número de bits do número pseudo-aleatório gerado.

        Retorna:
        int: Um número pseudo-aleatório de tamanho num_bits.
        """
        x = self.seed
        result = 0
        for i in range(num_bits):
            x = (self.a * x + self.c) % self.m # Calcular Xn+1 = (a * Xn + c) mod m
            bit = x % 2 # Extrair o bit menos significativo de x
            result = (result << 1) | bit # Construir o número resultante bit a bit
        return result
```

Previsibilidade do LCG:

Embora o LCG seja fácil de implementar e rápido, ele tem algumas limitações significativas em termos de segurança e previsibilidade. As principais razões para sua previsibilidade são:

- **Linearidade:** A fórmula básica do LCG é linear, o que significa que a sequência de números gerada não possui uma complexidade suficiente para evitar a previsão. Dado um número suficiente de valores da sequência, é possível resolver o sistema de equações lineares para determinar os parâmetros a, c e m.

- **Período Limitado:** O LCG pode gerar no máximo m valores distintos antes de começar a repetir a sequência. Para muitos LCGs práticos, o valor de m é uma potência de 2 (por exemplo, $m = 2^{32}$), o que limita o período máximo e pode levar a ciclos curtos.

2 GERAÇÃO DE NÚMEROS PSEUDO-ALEATÓRIOS

A seguir, implementamos a geração de números de diferentes tamanhos (em bits) e medimos o tempo necessário para cada um dos algoritmos.

Parâmetros para BBS [3]:

$p = 383$

$q = 503$

$s_0 = 101355$

Parâmetros para LCG [10]:

$\mathrm{a} = 15005$

$c = 8371$

$m = 19993$

$seed = 135$

Tamanhos de bits: 40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096

Implementação em Python:

```python
# Parâmetros para os geradores
p = 383
q = 503
s = 101355

a = 15005
c = 8371
m = 19993
seed = 135

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

random_generator = RandomNumberGenerator(p, q, s, a, c, m, seed)

# Gerando números aleatórios com Blum Blum Shub e Congruência Linear
random_numbers = random_generator.generate_numbers(tamanhos_bits)

print("Números gerados com Blum Blum Shub:")
for i, (num, duration) in enumerate(random_numbers["blum_blum_shub"]):
    print(f"Número de {tamanhos_bits[i]} bits: {num} (Tempo: {duration:.6f} segundos)")

print("\nNúmeros gerados com Congruência Linear:")
for i, (num, duration) in enumerate(random_numbers["linear_congruential"]):
    print(f"Número de {tamanhos_bits[i]} bits: {num} (Tempo: {duration:.6f} segundos)")
```

```python
import time
from typing import Dict, Tuple, List

from generators.blum_blum_shub import BlumBlumShub
from generators.linear_congruential import LinearCongruential


class RandomNumberGenerator:
    """
    Classe para gerar números aleatórios usando Blum Blum Shub e Congruência Linear.

    Parâmetros:
    p (int): Um número primo onde p ≡ 3 (mod 4) para Blum Blum Shub.
    q (int): Um número primo onde q ≡ 3 (mod 4) para Blum Blum Shub.
    s (int): Semente inicial para Blum Blum Shub.
    a (int): Multiplicador para Congruência Linear.
    c (int): Incremento para Congruência Linear.
    m (int): Módulo para Congruência Linear.
    seed (int): Semente inicial para Congruência Linear.
    """
    def __init__(self, p: int, q: int, s: int, a: int, c: int, m: int, seed: int):
        self.blum_blum_shub = BlumBlumShub(p, q, s)
        self.linear_congruential = LinearCongruential(a, c, m, seed)
```

```python
    def generate_numbers(self, tamanhos_bits: List[int]) → Dict[str, Tuple[int, float]]:
        """
        Gera números aleatórios para cada tamanho de bits especificado.

        Parâmetros:
        tamanhos_bits (int): Lista de tamanhos de bits.

        Retorna:
        dict[str, (int, float)]: dict[nome do algoritmo, (número gerado, tempo de geração em segundos)].
        """
        results = {
            "blum_blum_shub": [],
            "linear_congruential": []
        }

        for tamanho in tamanhos_bits:
            start_time = time.time()
            random_number = self.blum_blum_shub.generate(tamanho)
            duration = time.time() - start_time
            results["blum_blum_shub"].append((random_number, duration))

            start_time = time.time()
            random_number = self.linear_congruential.generate(tamanho)
            duration = time.time() - start_time
            results["linear_congruential"].append((random_number, duration))

        return results
```

Resultados:

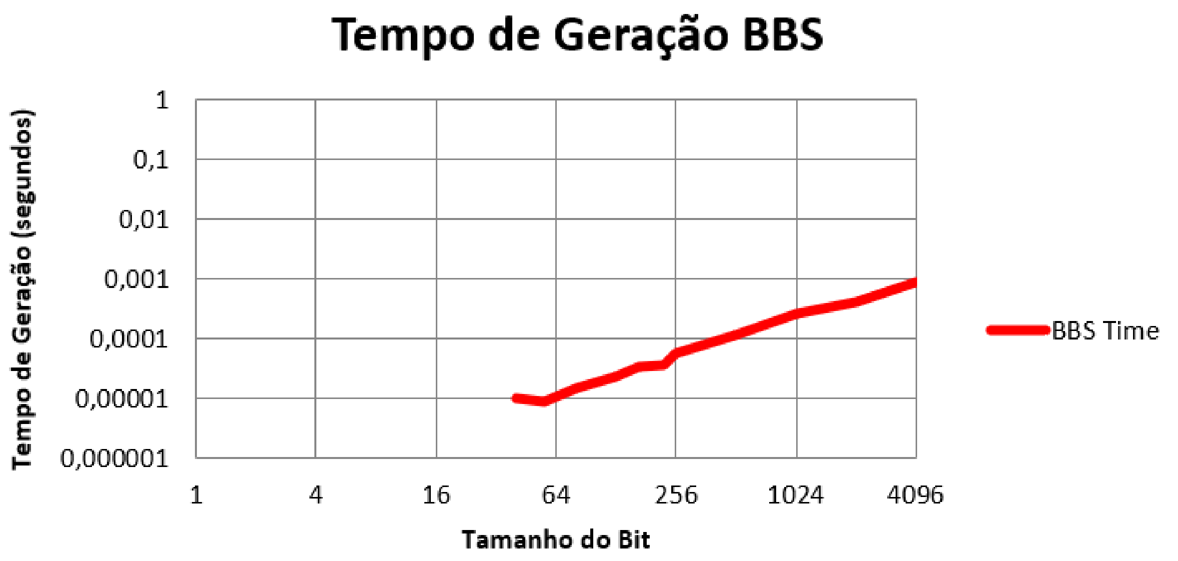
Algoritmo	Tamanho do Número (bits)	Tempo pra Gerar (segundos)
Blum Blum Shub	40	0.000010
Blum Blum Shub	56	0.000009
Blum Blum Shub	80	0.000015

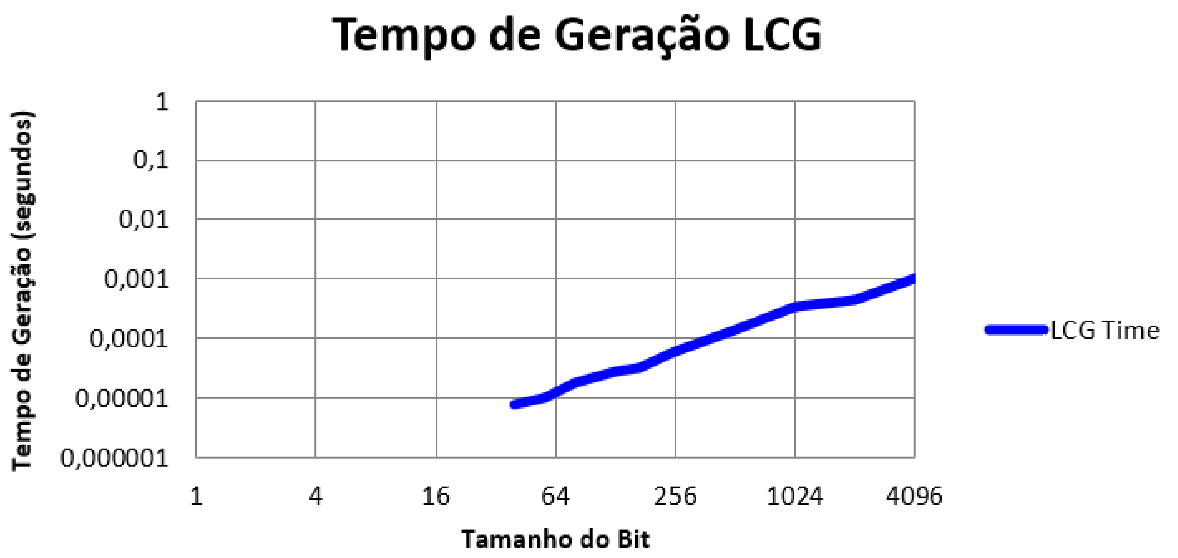
Blum Blum Shub	128	0.000023
Blum Blum Shub	168	0.000033
Blum Blum Shub	224	0.000037
Blum Blum Shub	256	0.000055
Blum Blum Shub	512	0.000113
Blum Blum Shub	1024	0.000256
Blum Blum Shub	2048	0.000406
Blum Blum Shub	4096	0.000891
LCG	40	0.000008
LCG	56	0.000010
LCG	80	0.000018
LCG	128	0.000029
LCG	168	0.000031
LCG	224	0.000050
LCG	256	0.000062
LCG	512	0.000136
LCG	1024	0.000346
LCG	2048	0.000436
LCG	4096	0.001013

3 COMPARAÇÃO DOS ALGORITMOS

3.1 TEMPO DE GERAÇÃO

Ambos os algoritmos geraram números pseudo-aleatórios de todos os tamanhos especificados com sucesso. Os tempos de geração foram bastante rápidos para ambos, mas o Blum Blum Shub apresentou tempos ligeiramente menores nos casos apresentados. No entanto, são valores muito baixos, a diferença não chega a ser destoante. O que nos leva a induzir que essa diferença está interligada ao tamanho dos parâmetros escolhidos e as operações entre si.





3.2 ANÁLISE DE COMPLEXIDADE

- Blum Blum Shub (BBS): A complexidade do BBS é $O\ (num\_bits)$ pois cada bit adicional requer uma operação de módulo que depende do tamanho dos números primos p e q.
- Linear Congruential Generator (LCG): A complexidade do LCG também é $O\ (num\_bits)$, já que cada iteração para gerar um bit envolve uma operação aritmética modular simples. O LCG não é adequado para aplicações criptográficas devido à sua previsibilidade, portanto, sua qualidade é baseada na escolha de seus parâmetros, e para teste, foram utilizados valores conforme sugeridos por PRESS, William H [5] após estudos.

3.3 CONCLUSÃO

Neste relatório, foi implementado e comparado dois algoritmos geradores de números pseudo-aleatórios: Blum Blum Shub e Linear Congruential Generator. O BBS apresentou tempos de geração ligeiramente melhores e oferece maior segurança criptográfica em comparação ao LCG. Dependendo da aplicação e dos requisitos de segurança, um algoritmo pode ser mais adequado do que o outro.

4 ALGORITMO MILLER-RABIN

O algoritmo Miller-Rabin é um teste probabilístico para verificar se um número é primo. A ideia central é que, para um número ser primo, ele deve satisfazer certas propriedades aritméticas. O teste baseia-se em propriedades dos números em relação a uma base escolhida aleatoriamente.

Passos do Algoritmo [6]:

1. **Decomposição do Número:** Dado um número ímpar n, escreva $n - 1$ na forma $2^t\ x\ u$, onde u é um número ímpar.
2. **Escolha de Bases:** Escolha s bases aleatórias para gerar a onde $1 < a < n - 1$.
3. **Verificação:** Para cada base a:
   a. Calcule $x = a^u\ mod\ n$.
   b. Para i de 0 a t:
      i. Calcule $x = x^2\ mod\ n$. Se $x = 1$ e $x_{i-1} \neq 1$ e $x_{i-1} \neq n - 1$, n é composto.
   c. Se $x_t \neq 1, n$ é composto.
   d. Se nenhuma das condições acimas forem satisfeitas, n pode ser primo.
4. **Conclusão:** Se n passar em todas as iterações, é provavelmente primo; caso contrário, é composto.

Código em Python:

```python
from typing import List
import random
import time


class MillerRabinTest:
    """
    Classe para realizar o teste de primalidade de Miller-Rabin.

    Parâmetros:
    s (int): O número de iterações do teste.
    """
    def __init__(self, s: int):
        self.s = s


    def miller_rabin(self, n: int) -> bool:
        """
        Teste de primalidade de Miller-Rabin.

        Parâmetros:
        n (int): O número a ser testado.

        Retorna:
        bool: True se n é provavelmente primo, False se n é composto.
        """

        # Casos base
        if n == 2 or n == 3:
            return True
        if n <= 1 or n % 2 == 0:
            return False

        # Escreva n-1 na forma 2^t * u
        t, u = 0, n - 1
        while u % 2 == 0:
            t += 1
            u //= 2
```

```python
        # Realiza o teste s vezes
        for j in range(self.s):
            a = random.randint(2, n - 2)
            if self.check_composite(a, t, u, n):
                return False

        return True


    def check_composite(self, a: int, t: int, u: int, n: int) -> bool:
        """
        Verifica se o número é composto para uma base a.

        Parâmetros:
        a (int): A base para o teste.
        t (int): O número de fatores de 2 em n-1.
        u (int): O fator ímpar em n-1.
        n (int): O número sendo testado.

        Retorna:
        bool: True se n é composto, False se provavelmente primo.
        """
        x = pow(a, u, n)
        for i in range(t):
            new_x = pow(x, 2, n)
            if new_x == 1 and x != 1 and x != n - 1:
                return True
            x = new_x
        if new_x != 1:
            return True
        return False
```

```python
    def test(self, numbers: List[int]):
        """
        Testa a primalidade de uma lista de números.

        Parâmetros:
        numbers (list): Lista de números a serem testados.
        """
        for n in numbers:
            start_time = time.time()
            result = self.miller_rabin(n)
            duration = time.time() - start_time

            print(f"Teste de {n}:")
            print(f"  Miller Rabin: {'Primo' if result else 'Composto'}, Tempo: {duration:.6f} segundos")
            print()
```

4.1 ANÁLISE DE COMPLEXIDADE DO ALGORITMO MILLER-RABIN

O algoritmo Miller-Rabin é um teste probabilístico para verificar a primalidade de um número. A análise de complexidade leva em consideração tanto o tempo de execução quanto a eficiência do algoritmo em termos de operações necessárias para concluir o teste.

1. **Decomposição do Número $\boldsymbol{n-1}$ na Forma $\boldsymbol{2^t\ x\ u}$:**
    a. Este passo envolve fatorar $n-1$ para encontrar t e u.
    b. A fatoração $n-1$ em $2^t\ x\ u$ é feita dividindo respectivamente $n-1$ por 2 até que o resultado seja ímpar.
    c. No pior caso, isso requer $O(\log n)$ operações de divisão, já que o número de divisões é proporcional ao número.
2. **Escolha das Bases e Verificações:**
    a. Para cada uma das s iterações, escolhemos uma base a aleatória no intervalo $1 < a < n-1$.
    b. O cálculo de $x = a^u\ mod\ n$ usa exponenciação modular, que pode ser feita em $O(\log u)$ usando o método de exponenciação rápida [7].
    c. A exponenciação requer $O(\log u)$ multiplicações modulares. Cada multiplicação modular $p_1\ x\ p_2\ mod\ n$ pode ser realizado em $O(\log n)\ x\ O(\log n) = O(\log^2 n)$.
    d. Portando, a complexidade de $a^u\ mod\ n = O(\log u\ \ x\ \log^2 n)$.
    e. Como $u < n, \log u$ é no máximo $\log n$, então a complexidade é $O(\log^3 n)$.
3. **Verificação de $\boldsymbol{x = x^2\ mod\ n}$**
    a. Este passo é repetido t vezes. Cada atribuição é uma multiplicação modular, que tem complexidade $O(\log^2 n)$.
    b. No pior caso, este passo é realizado t vezes, onde $t = O(\log n)$.
4. **Iteração Total:**
    a. Como realizamos s iterações, a complexidade total para as verificações é $s\ x\ O(\log^3 n + \log^2 n\ x \log n)$.
    b. Simplificando, a complexidade por iteração é $O(\log^3 n),$ então a complexidade total é $O(s\ x\ \log^3 n)$.
5. **Conclusão:**
    a. A complexidade se dá pelas exponenciações modulares que são feitas e pelo número de iterações escolhido.
    b. [6] Cormen menciona que a complexidade do algoritmo Miller-Rabin, com n sendo um número de β bits, exige $O(s\ x\ \beta)$ operações aritméticas

e $O(s\ x\ \beta^3)$ operações de bits, pois não exige assintoticamente mais trabalho que s operações modulares.

5 ALGORITMO DE PRIMALIDADE DE FERMAT

O Teste de Primalidade de Fermat é um método probabilístico para verificar se um número é primo. Baseia-se no pequeno teorema de Fermat, que afirma que se p é um número primo, então $a^{p-1} \equiv 1 \ (mod\ p) \ \forall a \in Z_p^*$ $[6]$.

O Teste de Primalidade de Fermat foi escolhido pois é um dos algoritmos mais simples para verificar a primalidade de um número. De tal modo, é interessante ver como ele se comporta comparado com um algoritmo mais complexo como é o de Miller-Rabin.

Passos do algoritmo [9]:

1. **Escolha de bases:** escolha k bases aleatórias a onde $1 < a < n$ [8].
2. **Verificação:** Para cada base a:
    a. Calcule $a^{n-1}\ mod\ n$. Se $a^{n-1} \equiv 1 \ (mod\ n)$, então n é primo, do contrário, é composto.
3. **Conclusões:** Se n passar em todas as iterações, é provavelmente primo; caso contrário, é composto.

Código em Python:

```python
from typing import List
import random
import time

class FermatTest:
    """
    Classe para realizar o teste de primalidade de Fermat.

    Parâmetros:
    s (int): O número de iterações do teste.
    """
    def __init__(self, s: int):
        self.s = s


    def fermat(self, n: int) -> bool:
        """
        Teste de primalidade de Fermat.

        Parâmetros:
        n (int): O número a ser testado.

        Retorna:
        bool: True se n é provavelmente primo, False se n é composto.
        """

        # Casos base: verifica números pequenos e pares
        if n == 2 or n == 3:
            return True
        if n <= 1 or n % 2 == 0:
            return False

        # Realiza o teste s vezes
        for i in range(self.s):
            a = random.randint(2, n - 2)
            if pow(a, n - 1, n) != 1:
                return False
        return True
```

```python
def test(self, numbers: List[int]):
    """
    Testa a primalidade de uma lista de números.

    Parâmetros:
    numbers (list): Lista de números a serem testados.
    """
    for n in numbers:
        start_time = time.time()
        result = self.fermat(n)
        duration = time.time() - start_time

        print(f"Teste de {n}:")
        print(f"  Fermat: {'Primo' if result else 'Composto'}, Tempo: {duration:.6f} segundos")
        print()
```

5.1 ANÁLISE DE COMPLEXIDADE DO ALGORITMO DE FERMAT

1. **Escolha das Bases e Verificações:**
    a. Para cada uma das s iterações, escolhemos uma base a no intervalo $1 < a < n$ [8].
    b. O cálculo de $a^{n-1}\ mod\ n$ usa exponenciação modular, que pode ser feita em $O(\log(n-1))$ usando o método de exponenciação rápida [7].
    c. A exponenciação requer $O(\log(n-1))$ multiplicações modulares. Cada multiplicação modular $p_1\ x\ p_2\ mod\ n$ pode ser realizado em $O(\log n)\ x\ O(\log n) = O(\log^2 n)$.
    d. Portando a complexidade de $a^{n-1}\ mod\ n$ é $O(\log(n-1))\ x\ O(\log^2 n)$.
    e. Como $n - 1 < n, \log(n-1)$ é no máximo $\log n$, então a complexidade é $O(\log^3 n)$.
2. **Iteração Total:**
    a. Como realizamos s iterações, a complexidade total para as verificações é de $O(s\ x\ \log^3 n)$.
3. **Conclusão:**
    a. A complexidade se dá pelas exponenciações modulares que são feitas e pelo número de iterações escolhido.

6 RESULTADO DOS TESTES DE PRIMALIDADE

Todos os números obtidos foram validados por ambos os algoritmos, ou seja, os dois algoritmos de teste de primalidade respondiam “Primo” para o número gerado em todas as situações. Mais detalhes podem ser encontrados no arquivo output.txt no GitHub.

Algoritmo Gerador	Algoritmo Verificador	Tamanho do Número (bits)	Tempo Para Gerar (s)	Número Gerado
Blum Blum Shub	Miller-Rabin	40	0.000604	227877182887
Blum Blum Shub	Miller-Rabin	56	0.000750	30613074601800277
Blum Blum Shub	Miller-Rabin	80	0.001014	651432684564531651056533
Blum Blum Shub	Miller-Rabin	128	0.005069	25500585582584652074852757638440668733 1
Blum Blum Shub	Miller-Rabin	168	0.008927	216359074388476251830217325082949384885798347894 77
Blum Blum Shub	Miller-Rabin	224	0.021098	21829146326494738484793623778111811550763244173127943046489196798937
Blum Blum Shub	Miller-Rabin	256	0.022905	49488524816903844853526848168458758216625260329202956597982717682969560125529
Blum Blum Shub	Miller-Rabin	512	0.125956	845928106860802245706893721830974669406676060072286258929312555195362155407935916409243150664221501237468634461185608677430347366448237703657341 6673829251
Blum Blum Shub	Miller-Rabin	1024	0.350132	999144848725936462086326320916329953833851576325016071506556728593461939669457239188896999188801

				3950853684388641 0128925638720694 7319135000655856 0628009020327465 0624397240161092 4176969963968911 0487183990261014 0634847777073478 5082120599396655 4125204299114228 6155721108640022 0095717245698560 2277967741087002 7077
Blum Blum Shub	Miller-Rabin	2048	13.229811	2847110650259421 2738199680286732 0926254602830743 7213871750977932 8703378766687193 6635177634965399 3006794927831544 4020541504660054 9096897706891333 1680652832800750 1962052512108420 0596710062212003 9420403533327578 9303961004772657 8362907002141000 9189765783322881 6810504071307506 9653620042037379 8265526596635338 8579292194373753 0688137295582884 3016635746341641 5229695501005497 9506697269193606 8420671494941710 6061862479534144 2137649625052584 9874781512494175 7004118458894754 7910102228084434 1036131485124051 4043078446605082 9480117515642567 9687744396333848 4630439006513125 3163674296158199

				6197903348564003 1287582080924716 670071393
Blum Blum Shub	Miller-Rabin	4096	327.818829	3198103855054007 6337590058376230 9931672242425636 4914736105150264 3814282852001489 2911123505915414 6350694574598128 4210659906436296 2341864735257018 6454148504509548 3182624618668020 1534739404992863 6994874064710436 2508433843061922 1343384333300642 7156362013799384 6782918180066314 4596564685337985 5143350943871740 7869585323245903 2631114708970358 2355399758539104 7434209104218130 4823813732329458 4246785273946945 6904476903671814 0431830996637767 7695836101639034 5518446708072753 1979547625500839 8131683313545042 0766664484913591 7440253493261025 0479051356964816 8444474153863546 3551348438053488 0250040063692888 8947867156835311 5291084176041321 0674400566950618 4073473267576779 2536658756244772 3202649815062037 4892445620988917 6298850191879724 1662039796696655 7077138234276741

				2413876355087523 0820083163206903 7373956977722748 7160147100208178 3487714812355226 7941892067700697 2738422173827759 5905297789169001 8881409607290204 0788618994432516 1733066179908934 4199044995025752 9350485846558356 5790818783491766 6412321065598051 1307188400808597 4957783747415707 0543037337443763 2465845628944487 8404794559789536 8992547769459096 5267642196312533 8637159475275220 5138749096835717 2502170469200566 1352078237229882 6128293789582483 2047592429909958 5035775977010237 7569742107704087 9
Blum Blum Shub	Fermat	40	0.000541	227877182887
Blum Blum Shub	Fermat	56	0.000673	3061307460180027 7
Blum Blum Shub	Fermat	80	0.000945	6514326845645316 51056533
Blum Blum Shub	Fermat	128	0.005650	2550058558258465 2074852757638440 6687331
Blum Blum Shub	Fermat	168	0.009552	2163590743884762 5183021732508294 9384885798347894 77
Blum Blum Shub	Fermat	224	0.021561	2182914632649473 8484793623778111 8115507632441731 2794304648919679 8937

Blum Blum Shub	Fermat	256	0.025817	49488524816903844853526848168458758216625260329202956597982717682969560125529
Blum Blum Shub	Fermat	512	0.124287	845928106860802245706893721830974669406676060072286258929312555195362155407935916409243150664221501237468634461185608677430347366448237703657341667382925 1
Blum Blum Shub	Fermat	1024	0.349625	99914484872593646208632632091632995383385157632501607150655672859346193966945723918889699918880139508536843886410128925638720694731913500065585606280090203274650624397240161092417696996396891104871839902610140634847777073478508212059939665541252042991142286155721108640022009571724569856022779677410870027077
Blum Blum Shub	Fermat	2048	13.232701	284711065025942127381996802867320926254602830743721387175097793287033787666871936635177634965399300679492783154440205415046600549096897706891333168065283280075019620525121084200596710062212003942040353332757893039610047726578362907002141000

				9189765783322881 6810504071307506 9653620042037379 8265526596635338 8579292194373753 0688137295582884 3016635746341641 5229695501005497 9506697269193606 8420671494941710 6061862479534144 2137649625052584 9874781512494175 7004118458894754 7910102228084434 1036131485124051 4043078446605082 9480117515642567 9687744396333848 4630439006513125 3163674296158199 6197903348564003 1287582080924716 670071393
Blum Blum Shub	Fermat	4096	327.880555	3198103855054007 6337590058376230 9931672242425636 4914736105150264 3814282852001489 2911123505915414 6350694574598128 4210659906436296 2341864735257018 6454148504509548 3182624618668020 1534739404992863 6994874064710436 2508433843061922 1343384333300642 7156362013799384 6782918180066314 4596564685337985 5143350943871740 7869585323245903 2631114708970358 2355399758539104 7434209104218130 4823813732329458 4246785273946945 6904476903671814

				0431830996637767 7695836101639034 5518446708072753 1979547625500839 8131683313545042 0766664484913591 7440253493261025 0479051356964816 8444474153863546 3551348438053488 0250040063692888 8947867156835311 5291084176041321 0674400566950618 4073473267576779 2536658756244772 3202649815062037 4892445620988917 6298850191879724 1662039796696655 7077138234276741 2413876355087523 0820083163206903 7373956977722748 7160147100208178 3487714812355226 7941892067700697 2738422173827759 5905297789169001 8881409607290204 0788618994432516 1733066179908934 4199044995025752 9350485846558356 5790818783491766 6412321065598051 1307188400808597 4957783747415707 0543037337443763 2465845628944487 8404794559789536 8992547769459096 5267642196312533 8637159475275220 5138749096835717 2502170469200566 1352078237229882 6128293789582483 2047592429909958 5035775977010237

				75697421077040879
Linear Congruential	Miller-Rabin	40	0.001323	346028905727
Linear Congruential	Miller-Rabin	56	0.002710	3949291432650937
Linear Congruential	Miller-Rabin	80	0.002890	418045946593530905965129
Linear Congruential	Miller-Rabin	128	0.003456	19763175459022166019713931851200437010 3
Linear Congruential	Miller-Rabin	168	0.015765	1061146404307599871358800571967694341335539738458 61
Linear Congruential	Miller-Rabin	224	0.019281	10520265214584507099285208134265786403433700225124378856914842408043
Linear Congruential	Miller-Rabin	256	0.035845	25781976137606568767524762621046834646435149573681735932906611543975970177031
Linear Congruential	Miller-Rabin	512	0.073406	84035217972708454263652954236313091825240218993673461076877154421303681672281456861568173629418691781225968021730070358163616933651049080819571726575306 9
Linear Congruential	Miller-Rabin	1024	1.059104	434263980177776115922346045203312591456716137660528134760944204354927218863564418079539904075436060112258227435600320066244599791691560127147466805628739270891393982196551331160976995466309275668849402597219845576595331551738375881047688501

				83872363308024426964530067687737562670006941790447213047207198119271
Linear Congruential	Miller-Rabin	2048	17.540953	2234951209082311184682158736080742651826200205932526402892725085094686908176708041441753141815660545499145041120377695265027278343323195289015979688660166009564643364628238215835052587619737318532173724090129295709655278126995246821815647914126792119691247820275388420870836003214694686682278548272766351535130623984014047445877416888075014670325892371517438565951755454940847121297678520827687665816571099525329565612723759415597762101937975333514094870444218870946205637974291988168239253332799485104570337193811200804624056304591013415347801632531577150913294743660234062228030074979246714768409956810182794200017 7
Linear Congruential	Miller-Rabin	4096	382.107086	465963050408088428948853807295774144625553332264856755959355910156433816045541148463785657766247

				9393130331444377 0262322657658563 0502842797264746 5625605439779002 8236744394006173 2214596125350210 3480367220276930 4300730353183095 5213164275565678 6222286493806533 3507996356853704 2040500850234886 5114337354596946 7940156308487508 7499339116689052 3896103005232281 3044132732631062 9695382823562916 9364981047231765 7301179299983679 0368185666605597 3366523950728358 4561296341169804 4037924250071433 1174032502143319 1486800588417042 1896806633591569 7761627065281513 3348015540465229 2668431234379515 6603105246171445 1461250011368629 8780465021489008 5406929620266675 6880786697817238 1991697202071364 2058452365470157 4348716470339728 5855183998197716 3823529687626948 6664553557907967 0002199521669167 6948520606916283 7220000462654830 8831324136043981 0001299591810035 4235044921285774 5812556804506610 8300229890293470 4025446634839547

				33646290647085358645932084733307530608972402506201714607057444793473379482282033918435033498483459459362307994704439547515593744783279101003885533951386785499197612413044700323565854292694013719069825661836940343926530349688717654077081633618043460000000495255391143451835934056509530803977627986855561692422467991287041893185410719467290
Linear Congruential	Fermat	40	0.001156	346028905727
Linear Congruential	Fermat	56	0.002379	3949291432650937
Linear Congruential	Fermat	80	0.002622	418045946593530905965129
Linear Congruential	Fermat	128	0.003284	197631754590221660197139318512004370103
Linear Congruential	Fermat	168	0.014839	1061146404307599871358800571967694341335539738458 61
Linear Congruential	Fermat	224	0.018411	10520265214584507099285208134265786403433700225124378856914842408043
Linear Congruential	Fermat	256	0.034730	25781976137606568767524762621046834646435149573681735932906611543975970177031
Linear Congruential	Fermat	512	0.073168	84035217972708454263652954236313091825240218993673461076877154421303681672281456

				8615681736294186 9178122596802173 0070358163616933 6510490808195717 265753069
Linear Congruential	Fermat	1024	1.060179	4342639801777761 1592234604520331 2591456716137660 5281347609442043 5492721886356441 8079539904075436 0601122582274356 0032006624459979 1691560127147466 8056287392708913 9398219655133116 0976995466309275 6688494025972198 4557659533155173 8375881047688501 8387236330802442 6964530067687737 5626700069417904 4721304720719811 9271
Linear Congruential	Fermat	2048	17.528493	2234951209082311 1846821587360807 4265182620020593 2526402892725085 0946869081767080 4144175314181566 0545499145041120 3776952650272783 4332319528901597 9688660166009564 6433646282382158 3505258761973731 8532173724090129 2957096552781269 9524682181564791 4126792119691247 8202753884208708 3600321469468668 2278548272766351 5351306239840140 4744587741688807 5014670325892371 5174385659517554 5494084712129767 8520827687665816

				5710995253295656 1272375941559776 2101937975333514 0948704442188709 4620563797429198 8168239253332799 4851045703371938 1120080462405630 4591013415347801 6325315771509132 9474366023406222 8030074979246714 7684099568101827 942000177
Linear Congruential	Fermat	4096	382.418324	4659630504080884 2894885380729577 4144625553332264 8567559593559101 5643381604554114 8463785657766247 9393130331444377 0262322657658563 0502842797264746 5625605439779002 8236744394006173 2214596125350210 3480367220276930 4300730353183095 5213164275565678 6222286493806533 3507996356853704 2040500850234886 5114337354596946 7940156308487508 7499339116689052 3896103005232281 3044132732631062 9695382823562916 9364981047231765 7301179299983679 0368185666605597 3366523950728358 4561296341169804 4037924250071433 1174032502143319 1486800588417042 1896806633591569 7761627065281513 3348015540465229 2668431234379515

				6603105246171445 1461250011368629 8780465021489008 5406929620266675 6880786697817238 1991697202071364 2058452365470157 4348716470339728 5855183998197716 3823529687626948 6664553557907967 0002199521669167 6948520606916283 7220000462654830 8831324136043981 0001299591810035 4235044921285774 5812556804506610 8300229890293470 4025446634839547 3364629064708535 8645932084733307 5306089724025062 0171460705744479 3473379482282033 9184350334984834 5945936230799470 4439547515593744 7832791010038855 3395138678549919 7612413044700323 5658542926940137 1906982566183694 0343926530349688 7176540770816336 1804346000000049 5255391143451835 9340565095308039 7762798685556169 2422467991287041 8931854107194672 9

7 ANÁLISE DO RESULTADO PARA OS ALGORITMOS DE PRIMALIDADE

7.1 TEMPOS DE GERAÇÃO

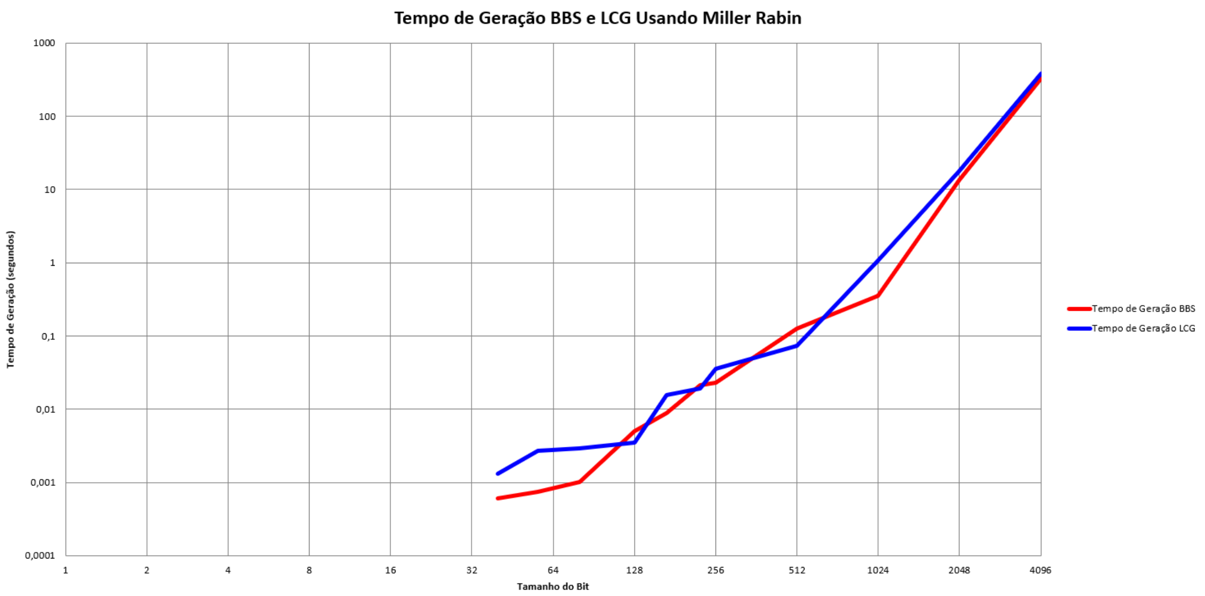
- Blum Blum Shub (BBS): O tempo de geração aumenta de acordo com o aumento do tamanho do bit. Por exemplo, a geração de números primos de 40 bits leva aproximadamente 0.000604 segundos, enquanto para 4096 bits leva cerca de 327.818829 segundos.
- Linear Congruential (LCG): Similar ao BBS, o tempo de geração para LC também aumenta com o tamanho do bit, chegando a 382.107086 segundos com um número de 4096 bits.

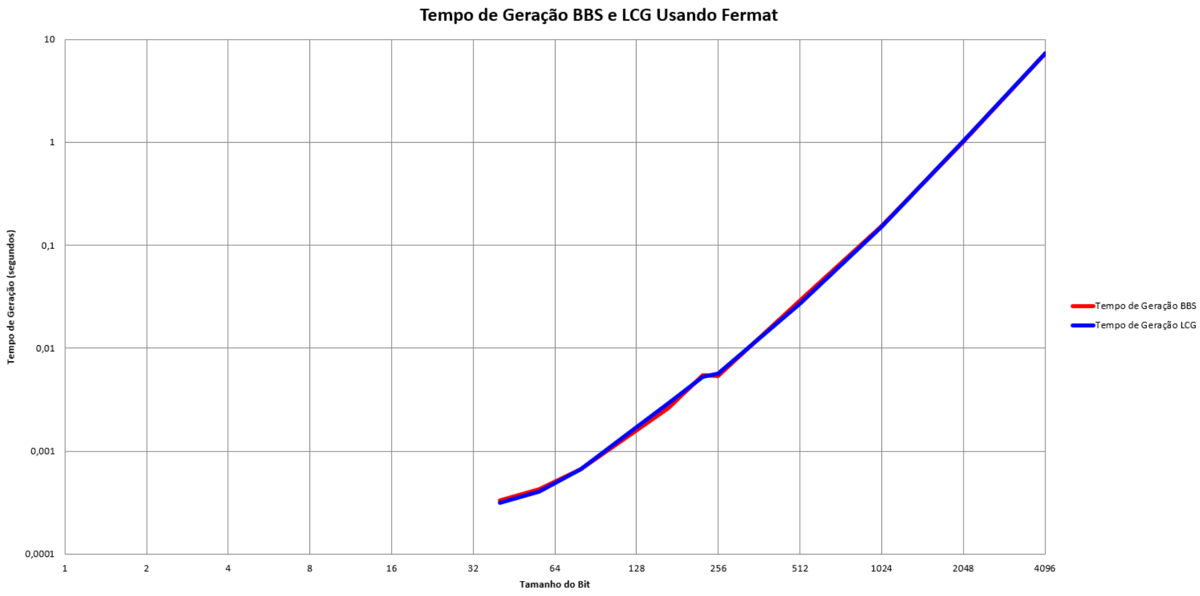
7.2 GRÁFICOS

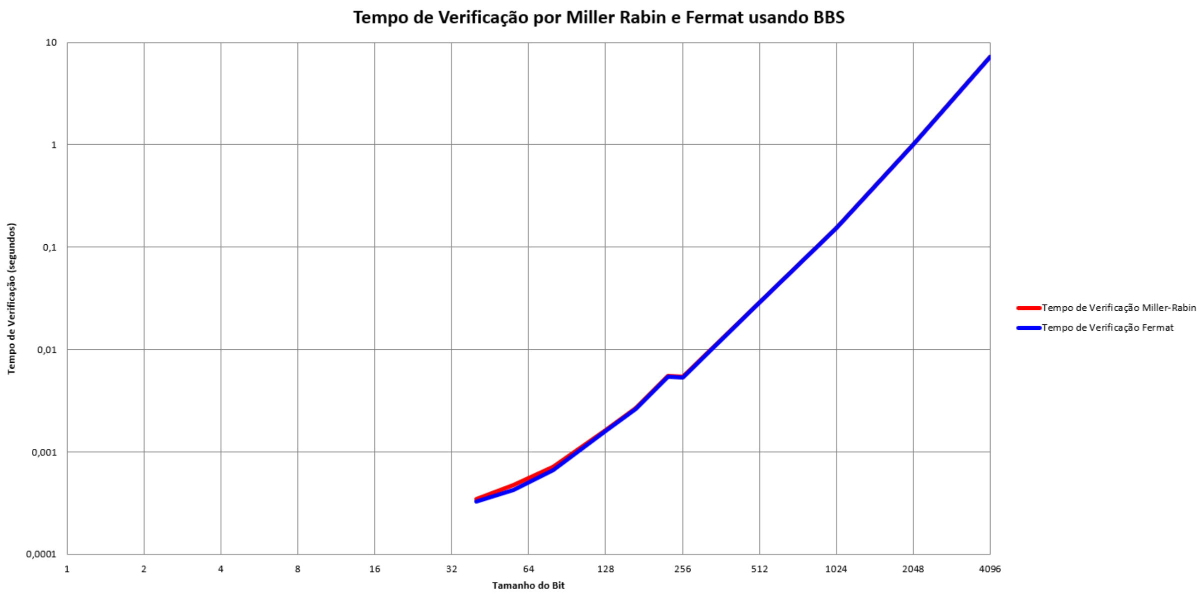
Os gráficos a seguir mostram visualmente os tempos de geração e verificação:

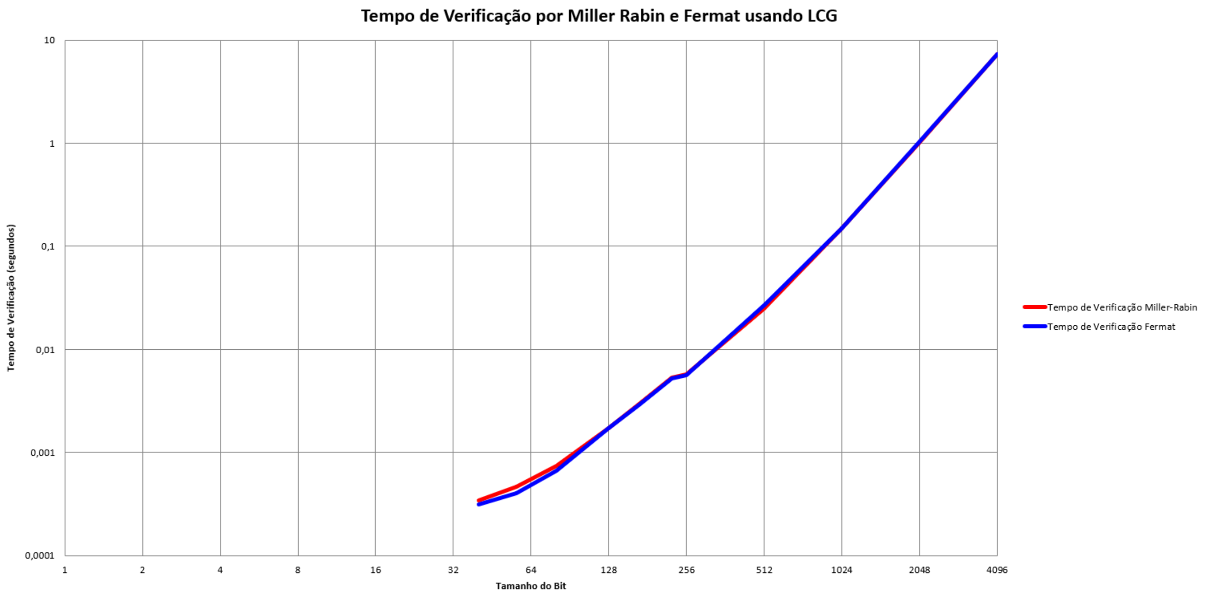
- Tempo de Geração: Comparação dos tempos de geração de número possivelmente primo entre BBS e LCG usando cada um dos algoritmos de verificação de primalidade implementado.

- Tempo de Verificação de Primalidade: Comparação dos tempos de verificação de primalidade de Fermat e Miller-Rabin após número possivelmente primo ser gerado com BBS e LCG.




Tempo de Geração BBS e LCG Usando Fermat
10
1
0,1
0,01
0,001
0,0001
Tempo de Geração (segundos)
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
Tamanho do Bit
Tempo de Geração BBS
Tempo de Geração LCG


Tempo de Verificação por Miller Rabin e Fermat usando BBS
10
1
0,1
0,01
0,001
0,0001
Tempo de Verificação (segundos)
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
Tamanho do Bit
Tempo de Verificação Miller-Rabin
Tempo de Verificação Fermat



7.3 CONCLUSÃO

Ao gerar números possivelmente primos utilizando os algoritmos Blum Blum Shub (BBS) e Linear Congruential Generator (LCG) e verificar sua primalidade com os métodos de Fermat e Miller-Rabin, observamos comportamentos interessantes. Quando utilizamos Miller-Rabin para a verificação de primalidade, o algoritmo BBS se destaca na geração de números pequenos e grandes, enquanto para números intermediários, há uma oscilação entre os dois algoritmos. Já ao utilizarmos o método de Fermat para verificar a primalidade, os tempos de geração dos números pseudo-aleatórios por ambos os algoritmos permanecem semelhantes.

Por outro lado, quando analisamos a eficácia dos métodos de Fermat e Miller-Rabin para verificar a primalidade dos números possivelmente primos gerados por BBS ou LCG, os tempos de validação são muito próximos. Isso reflete a eficiência de ambos os algoritmos de validação de primalidade, independentemente do método utilizado. Tanto Fermat quanto Miller-Rabin apresentam a mesma complexidade computacional, como analisado anteriormente. Isso implica que o tempo necessário para verificar se um número é possivelmente primo é similar para ambos os métodos, o que é confirmado pelos resultados e pelos gráficos apresentados.

8 REFERÊNCIAS

1. BLUM, Lenore; BLUM, Manuel; SHUB, Michael. A simple unpredictable pseudo-random number generator. SIAM Journal on Computing, v. 15, n. 2, p. 364-383, 1986.
2. BLUM Blum Shub. Wikipédia, a enciclopédia livre. Disponível em: https://pt.wikipedia.org/wiki/Blum_Blum_Shub. Acesso em: 20 de maio de 2024.
3. STINSON, Douglas; PATERSON, Maura. Cryptography: Theory and Practice. 4ª ed. Boca Raton: CRC Press, 2019.
4. STALLINGS, William. Cryptography and Network Security: Principles and Practice. 7ª ed. Pearson, 2017. Capítulo 8.
5. PRESS, William H.; TEUKOLSKY, Saul A.; VETTERLING, William T.; FLANNERY, Brian P. Numerical Recipes in C: The Art of Scientific Computing. 2ª ed. Cambridge: Cambridge University Press, 1992. Capítulo 7.
6. CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. Introduction to Algorithms. 4. ed. MIT Press, 2022.
7. Exponentiation by squaring. Wikipédia, a enciclopédia livre. Disponível em: https://en.wikipedia.org/wiki/Exponentiation_by_squaring. Acesso em: 20 maio 2024.
8. RABIN, M. O. Probabilistic Algorithm for Testing Primality. Journal of Number Theory, v. 12, n. 1, p. 128-138, 1980.
9. Fermat Primality Test. Disponível em: https://en.wikipedia.org/wiki/Fermat_primality_test. Acesso em: 28 May 2024.
10. J. C. Hernndez, A. Ribagorda, P. Isasi, and J. M. Sierra. 2001. Finding near optimal parameters for Linear Congruential pseudorandom number generators by means of evolutionary computation. In Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation (GECCO'01). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1292–1298.