Aluno: Douglas Pereira Luiz
Matrícula: 18203343

Trabalho Individual - Número Primos

Um número primo (ou um primo) é um número natural maior que 1 que não é produto de dois números naturais menores. Um número natural maior que 1 que não é primo é chamado de número composto. Por exemplo, 5 é primo porque as únicas maneiras de escrevê-lo como um produto, 1 × 5 ou 5 × 1, envolvem o próprio 5. No entanto, 4 é composto porque é um produto (2 × 2) em que ambos os números são menores que 4. Os primos são centrais na teoria dos números por causa do teorema fundamental da aritmética: todo número natural maior que 1 é um primo em si ou pode ser fatorado como um produto de primos que é único até sua ordem.

Sumário

Geração de números pseudoaleatórios.

Um gerador de números pseudoaleatórios (Pseudorandom number generator, **PRNG**) é um algoritmo para gerar uma sequência de números cujas propriedades se aproximam das propriedades de sequências de números aleatórios. A sequência gerada não é verdadeiramente aleatória, pois é completamente determinada por um valor inicial, chamado *seed* (que pode incluir valores verdadeiramente aleatórios). Os geradores de números pseudoaleatórios são importantes na prática por sua velocidade na geração de números e sua reprodutibilidade.

Podem ser aplicados em criptografia, sendo que, um PRNG adequado para aplicação criptográfica é chamado de PRNG criptograficamente seguro (*cryptographically-secure* PRNG, **CSPRNG**). Um

requisito para um CSPRNG é que um adversário que não conheça a *seed* tenha apenas uma vantagem insignificante ao distinguir a sequência de saída do gerador de uma sequência aleatória. O projeto de um CSPRNG é extremamente difícil porque eles devem atender a critérios adicionais.

Uma propriedade importante dos geradores de números pseudoaleatórios é o **período**, que indica quantas gerações levarão até que o gerador comece à se repetir. É desejável que o gerador tenha um período grande, para que a periodicidade não seja percebida.

Gerador Congruencial Linear

Um gerador congruencial linear (Linear Congruential Generator, **LCG**), ou gerador congruente linear, é um algoritmo que produz uma sequência de números pseudoaleatórios calculados com uma equação linear descontínua por partes. O método representa um dos mais antigos e conhecidos algoritmos geradores de números pseudoaleatórios. A teoria por trás deles é relativamente fácil de entender, e eles são facilmente implementados e rápidos.

O gerador é definido pela relação de recorrência:

$X_{n+1} = (aX_n + c) \bmod m$,

sendo que X é a sequência de valores pseudoaleatórios, e

$m, 0 < m$ − o "módulo"

$a, 0 < a < m$ − o "multiplicador"

$c, 0 \leq c < m$ − o "incremento"

$X_0, 0 \leq X_0 < m$ − o "valor inicial" ou "seed"

são inteiros constantes que especificam o gerador.

Período

Uma característica do LCG é que uma escolha apropriada de parâmetros resulta em um período longo e conhecido.

Quando $c \neq 0$, os parâmetros escolhidos corretamente permitem um período igual a m, para qualquer valor inicial. Isso ocorrerá se e somente se:

1. m e c são relativamente primos;
2. $a - 1$ é divisível por todos os primos fatores de m ;
3. $a - 1$ é divisível por 4 se m é divisível por 4.

Esses três requisitos são definidos no Teorema de Hull-Dobell, e funcionam bem quando m tem muitos fatores primos repetidos. Se m é potência de 2, $m \geq 4$, c é primo, a é ímpar e $a > 4$ satisfazemos as condições.

Complexidade

A complexidade da geração de um valor é definida em função da execução de uma operação de multiplicação, uma de soma e uma de resto da divisão, sendo então, nas implementações mais diretas, $O(N^2+N+\sqrt{N})$, sendo n o tamanho dos valores em bits.

BlumBlumShub

Blum Blum Shub (BBS) é um gerador de números pseudoaleatórios proposto em 1986 por Lenore Blum, Manuel Blum e Michael Shub derivado da função unidirecional de Michael O. Rabin. O gerador é definido por:

$X_{n+1}=(X_n^2) \bmod M$,

sendo que X é a sequência de valores pseudoaleatórios, e

$X_0, 0 \leq X_0 < m$ – o "valor inicial" ou "seed"

$M = pq$, sendo p e q primos grandes

M é coprimo de X_0

Normalmente é utilizado somente o bit de paridade de X_n , ao contrário da implementação apresentada daqui.

Período

O período do BBS é um fator de $\lambda(\lambda(M))$ onde $\lambda(n)$ é a função de Carmichael, que é o menor inteiro m que satisfaz $a^m \equiv 1 (\bmod n)$.

Complexidade

A complexidade é definida pelas operações de exponenciação com módulo, que pode ser calculado em $O(N^N+\sqrt{N})$, com N o tamanho dos valores em bits.

Comparação

O LCG e o BlumBlumShub são PRNGS com aplicações diferentes. O LCG é um dos geradores mais antigos e mais conhecidos, utilizado para gerar números que não serão aplicados em criptografia. É muito rápido e com a escolha certa dos parâmetros tem longo período. O algoritmo Blum Blum Shub tem sua segurança baseada na dificuldade do problema de resíduo quadrático. Como a única maneira conhecida de resolver esse problema é fatorar o módulo, geralmente considera-se que a dificuldade da fatoração inteira fornece uma segurança condicional para o algoritmo Blum Blum Shub. Porém o algoritmo é muito ineficiente e, portanto, impraticável, a menos que seja necessária uma segurança extrema.

Testes

Para compilar executar os testes, na pasta raiz:

javac main/Main.java

java main.Main <n> <p> <k>

sendo “n” o número de números aleatórios gerados para cada tamanho e algorítimo, “p” o número de primos gerados e “k” o número de *rounds* dos algorítimos de teste de primalidade.

Para os testes, foram gerados para cada tamanho, 40, 56, 80, 128, 224, 256, 512, 1024, 2048, 4096, 10.000 números. Em *Main.java* são realizados os testes para cada tamanho de valor através de um laço que itera sobre um vetor com os tamanhos desejados. Em uma iteração de testes:

1. É carregado os valores X_0 , p e q através da classe *PRNGConstants*, de acordo com o tamanho que a iteração se refere. Os valores são definidos nos arquivos *prng/constants/40*, …, *prng/constants/4096*;
2. Para o LCG é definido a, c, e m . m é definido em função do tamanho de valor desejado, sendo $m=2^{tamanho}$. Com esses valores e X_0 é instanciado um *LCG*.
3. Para o BlumBlumShub é calculado M com o produto de p e q e instanciado um *BBS* com os valores. Os valores de M e X_0 nos arquivos já são garantidos coprimos.
4. Em seguida é iniciado o teste, definido em *Analysis*.`generationAnalysis`, que faz chamadas n sucessivas da função `public BigInteger next()` do *PRNG* passado como parâmetro, guardando os valores gerados e os tempos para geração, retornando uma instância que guarda os resultados do teste. A instância é colocada em um vetor, `generationAnalyses,` que armazena todos os testes.

No fim dos testes é apresentado uma tabela com todos os valores gerados e seus tempos de geração em **microsegundos**. Além disso são apresentadas as médias, desvios padrão, mínimos e máximos dos tempos de geração para cada tamanho e algorítimo.

Geração de números pseudo aleatórios

Algoritmo	rraTamanho	Número	Tempo para gerar
LCG	40	414241498940	273.406
LCG	40	695981972631	15.242
LCG	40	273611871830	11.104
LCG	40	1038854227625	9.361
LCG	40	635040050176	11.064
LCG	40	671612451851	8.623
LCG	40	787727905466	8.377
LCG	40	1074176685373	14.595
LCG	40	766900287748	8.519
LCG	40	220170903231	8.279
LCG	40	333043265886	8.197
LCG	40	451213080849	7.984
LCG	40	877985960520	8.293
LCG	40	123653126323	9.655
LCG	40	149404956226	7.508
LCG	40	75943568933	7.351
LCG	40	876863372236	7.274
LCG	40	392569011687	8.165
LCG	40	664036271462	8.169
LCG	40	72151322745	8.133

Geração de números pseudo aleatórios

Algoritmo	Tamanho	Quantidade	Média	Mínimo	Máximo	Desvio padrão
LCG	40	10000	0.80612812...	0.405	2.851	0.54949573...
BBS	40	10000	1.36456569...	0.888	3.569	0.55946045...
LCG	56	10000	0.41353217...	0.262	0.646	0.07974759...
BBS	56	10000	0.99504894...	0.844	1.539	0.12475640...
LCG	80	10000	0.31983646...	0.273	0.382	0.02008525...
BBS	80	10000	1.30906310...	1.138	1.88	0.10517579...
LCG	128	10000	0.52786383...	0.382	0.685	0.05473037...
BBS	128	10000	1.12254231...	0.997	1.348	0.06365581...
LCG	224	10000	0.32363050...	0.277	0.383	0.01788726...
BBS	224	10000	1.64727209...	1.132	2.878	0.34128689...
LCG	256	10000	0.47925846...	0.355	0.606	0.04653504...
BBS	256	10000	1.30217374...	0.981	1.628	0.12499988...
LCG	512	10000	0.34454705...	0.311	0.4	0.01392561...
BBS	512	10000	1.87729208...	1.395	3.743	0.49684307...
LCG	1024	10000	0.51144675...	0.472	0.547	0.01255637...
BBS	1024	10000	4.26895607...	4.077	4.466	0.07070263...
LCG	2048	10000	0.88419664...	0.764	1.043	0.05188729...
BBS	2048	10000	13.7086235...	13.115	14.553	0.25350790...
LCG	4096	10000	1.88526913...	1.538	2.386	0.17069892...
BBS	4096	10000	46.3985577...	41.26	52.269	1.79358212...

Verificação de primalidade

Um teste de primalidade é um algoritmo para determinar se um número de entrada é primo. Entre outros campos da matemática, é usado para criptografia. A fatoração é considerado um problema computacionalmente difícil, enquanto o teste de primalidade deve ser comparativamente fácil. Alguns testes de primalidade provam que um número é primo, enquanto outros, como Miller-Rabin, provam que um número é composto.

Miller-Rabin

O teste de primalidade de Miller-Rabin é um teste de primalidade probabilística: um algoritmo que determina se um determinado número é provavelmente primo, semelhante ao teste de primalidade de Fermat e ao teste de primalidade de Solovay-Strassen. É de importância histórica na busca de um teste de primalidade determinístico em tempo polinomial. Sua variante probabilística continua sendo amplamente utilizada na prática, como um dos testes mais simples e rápidos conhecidos.

O algoritmo pode ser escrito em pseudocódigo da seguinte forma. O parâmetro k determina a precisão do teste. Quanto maior o número de rodadas, mais preciso será o resultado.

```
Input #1: n > 3, an odd integer to be tested for primality
Input #2: k, the number of rounds of testing to perform
Output: "composite" if n is found to be composite, "probably prime" otherwise
write n as 2^s·d + 1 with d odd (by factoring out powers of 2 from n − 1)
WitnessLoop: repeat k times:
    pick a random integer a in the range [2, n − 2]
    x ← a^d mod n
    if x = 1 or x = n − 1 then
        continue WitnessLoop
    repeat s − 1 times:
        x ← x^2 mod n
        if x = n − 1 then
            continue WitnessLoop
    return "composite"
return "probably prime"
```

Teste de Primalidade de Fermat

O teste de primalidade de Fermat é um teste probabilístico para determinar se um número é um provável primo. O pequeno teorema de Fermat afirma que se p é primo e a não é divisível por p, então $a^{p-1} \equiv 1 (mod\ p)$

Escolhi esse algorítimo pela simplicidade de aplicar diretamente o pequeno teorema de Fermat.

Para testar se p é primo, podemos escolher inteiros aleatórios a não divisíveis por p e ver se a igualdade é válida. Se a igualdade não vale para um valor de a, então p é composto. Esta congruência não é válida para um a aleatório se p for composto.[1] Portanto, se a igualdade vale para um ou mais valores de a, então dizemos que p é provavelmente primo. O algorítimo é pode ser descrito como à seguir.

Inputs: *n*: a value to test for primality, *n*>3; *k*: a parameter that determines the number of times to test for primality

Output: *composite* if *n* is composite, otherwise *probably prime*

Repeat *k* times:
  Pick *a* randomly in the range [2, *n* − 2]
  If $a^{n-1} \equiv 1 (mod\, n)$ is false, then return *composite*
If composite is never returned: return *probably prime*

Complexidade

Em ambos a complexidade é definida em função de k rounds e a operação de exponenciação, resultando para o teste de Fermat complexidade computacional de $O(k N^N)$, devido à

computação da exponencial. Para o teste de Miller-Rabin, ainda há as operaçãos para decomposição de n em $2^s d+1$, com no máximo $O(N)$ e as s repetições dentro de k. Apesar do laço interior, as operações nesse algorítimo são feitas sobre os valores decompostos da entrada, o que resulta em $O(N+kN^N)$ ou $O(kN^N)$. Considerando N o tamanho dos valores e implementação ingênua das operações sobre inteiros.

Comparação

Para o teste de Fermat, se um número composto n não for um número de Carmichael, que é um número inteiro positivo composto tal que, para todo inteiro positivo a coprimo com N, a^N é congruente com a módulo N, então a probabilidade de o teste não detectar a composição de um número composto é de 2^{-k}. No entanto, o teste falhará em todos os números de Carmichael.

Apesar de mais lento, o teste de Rabin-Miller apresenta 4^{-k} de chance de um número ser composto não ser detectado. Isso significa que a probabilidade de acerto é independente da entrada. Isto torna este algoritmo mais forte.

Testes

Para compilar executar os testes, na pasta raiz:

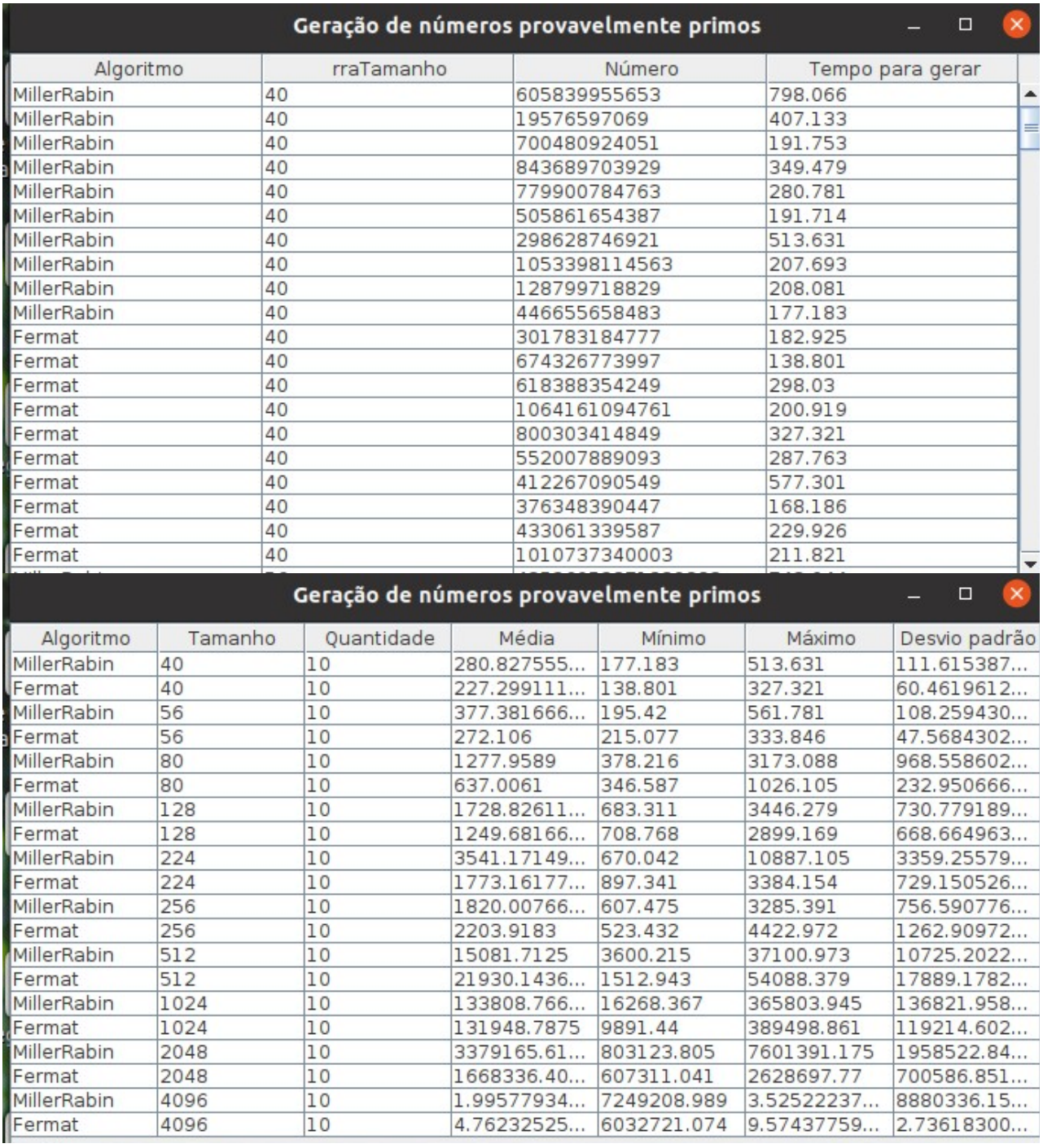
> javac main/Main.java
>
> java main.Main <n> <p> <k>

sendo “n” o número de números aleatórios gerados para cada tamanho e algorítimo, “p” o número de primos gerados e “k” o número de *rounds* dos algorítimos de teste de primalidade.

Para os testes, foram gerados para cada tamanho, 40, 56, 80, 128, 224, 256, 512, 1024, 2048, 4096, 10 números provavelmente primos, com 16 *rounds*. Em *Main.java* são realizados os testes para cada tamanho de valor através de um laço que itera sobre um vetor com os tamanhos desejados. Em uma iteração de testes:

1. É criado uma instância de *MillerRabin* que realiza em k *rounds* o teste e utiliza a instância de *LCG* (a mesma utilizada no teste de geração para o mesmo tamanho de valor) para geração dos números aleatórios, escolhido por ser mais rápido que o *BBS*.
2. É criado uma instância de *Fermat* que realiza em k *rounds* o teste e utiliza a instância de *LCG* (a mesma utilizada no teste de geração para o mesmo tamanho de valor) para geração dos números aleatórios.
3. Em seguida é iniciado o teste, definido em *Analysis*.`primalityTestAnalysis`, que faz chamadas sucessivas da função `public BigInteger next()` do *PRNG* passado como parâmetro, e testa os valores ímpares gerados com o *PrimalityTester* até que n valores satisfaçam o testador, guardando os n valores gerados e os tempos para obtenção, retornando uma instância que guarda os resultados do teste. A instância é colocada em um vetor, `primalityAnalyses,` que armazena todos os testes.

No fim dos testes é apresentado uma tabela com todos os n valores provavelmente primos e seus tempos de obtenção em **microsegundos**. Além disso são apresentadas as médias, desvios padrão, mínimos e máximos dos tempos de geração para cada tamanho e algorítimo.



Geração de números provavelmente primos

Algoritmo	rraTamanho	Número	Tempo para gerar
MillerRabin	40	605839955653	798.066
MillerRabin	40	19576597069	407.133
MillerRabin	40	700480924051	191.753
MillerRabin	40	843689703929	349.479
MillerRabin	40	779900784763	280.781
MillerRabin	40	505861654387	191.714
MillerRabin	40	298628746921	513.631
MillerRabin	40	1053398114563	207.693
MillerRabin	40	128799718829	208.081
MillerRabin	40	446655658483	177.183
Fermat	40	301783184777	182.925
Fermat	40	674326773997	138.801
Fermat	40	618388354249	298.03
Fermat	40	1064161094761	200.919
Fermat	40	800303414849	327.321
Fermat	40	552007889093	287.763
Fermat	40	412267090549	577.301
Fermat	40	376348390447	168.186
Fermat	40	433061339587	229.926
Fermat	40	1010737340003	211.821

Geração de números provavelmente primos

Algoritmo	Tamanho	Quantidade	Média	Mínimo	Máximo	Desvio padrão
MillerRabin	40	10	280.827555...	177.183	513.631	111.615387...
Fermat	40	10	227.299111...	138.801	327.321	60.4619612...
MillerRabin	56	10	377.381666...	195.42	561.781	108.259430...
Fermat	56	10	272.106	215.077	333.846	47.5684302...
MillerRabin	80	10	1277.9589	378.216	3173.088	968.558602...
Fermat	80	10	637.0061	346.587	1026.105	232.950666...
MillerRabin	128	10	1728.82611...	683.311	3446.279	730.779189...
Fermat	128	10	1249.68166...	708.768	2899.169	668.664963...
MillerRabin	224	10	3541.17149...	670.042	10887.105	3359.25579...
Fermat	224	10	1773.16177...	897.341	3384.154	729.150526...
MillerRabin	256	10	1820.00766...	607.475	3285.391	756.590776...
Fermat	256	10	2203.9183	523.432	4422.972	1262.90972...
MillerRabin	512	10	15081.7125	3600.215	37100.973	10725.2022...
Fermat	512	10	21930.1436...	1512.943	54088.379	17889.1782...
MillerRabin	1024	10	133808.766...	16268.367	365803.945	136821.958...
Fermat	1024	10	131948.7875	9891.44	389498.861	119214.602...
MillerRabin	2048	10	3379165.61...	803123.805	7601391.175	1958522.84...
Fermat	2048	10	1668336.40...	607311.041	2628697.77	700586.851...
MillerRabin	4096	10	1.99577934...	7249208.989	3.52522237...	8880336.15...
Fermat	4096	10	4.76232525...	6032721.074	9.57437759...	2.73618300...

Dificuldade na geração

Os testes foram feitos com k rounds. Para geração de números provavelmente primos de até 1024 bits, a média do tempo levado ainda é menor do que 1 segundo, utilizando o gerador de números pseudoaleatórios LCG. A partir desse tamanho é extremamente custoso, chegando à média

de 20 segundos para o teste de MillerRabin, e 50 segundos para o teste de Fermat quando queremos obter valores de 4096bits.

Código

O código também pode ser obtido em https://github.com/douglaspereira04/primes

main/Main.java

```java
package main;

import java.io.FileNotFoundException;
import java.math.BigInteger;

import analysis.Analysis;
import display.Table;
import primalitytester.Fermat;
import primalitytester.MillerRabin;
import primalitytester.PrimalityTester;
import prng.BBS;
import prng.LCG;
import prng.PRNG;
import prng.PRNGConstants;

public class Main {

	/**
	 * Tamanhos dos valores gerados
	 */
	public static int[] SIZE = { 40, 56, 80, 128, 224, 256, 512, 1024, 2048, 4096 };


	/**
	 * Executa os testes
	 * @param args vetor de {@link String} que na posição 0 deve ter
	 * o número de valores que devem ser gerados por cada gerador
	 * de números pseudo-aleatórios, na posição 1 deve ter o número
	 * de prováveis primos aprovados por cada teste de primalidade
	 * e na posição 2 tem o valor de estágios para os testes de
	 * primalidade
	 *
	 * @throws FileNotFoundException
	 */
	public static void main(String[] args) throws FileNotFoundException {

		int generationAmount = Integer.valueOf(args[0]);
		int primeAmount = Integer.valueOf(args[1]);
		int primalityPrecision = Integer.valueOf(args[2]);

		BigInteger x0;
		BigInteger a, c, m;
		BigInteger M, p, q;

		PRNG lcg;
		PRNG bbs;
		PrimalityTester millerRabin;
		PrimalityTester fermat;

		/*
		 * Mesmos parâmetros usados em java.util.Random, exceto por m, que varia de
		 * acordo com o tamanho do número que queremos gerar, fazendo-o sempre potência
		 * de 2 para não ferir os requisitos do teorema de Hull–Dobell
		 */
		a = new BigInteger("25214903917");
		c = new BigInteger("11");

		Analysis[] generationAnalyses = new Analysis[(SIZE.length)*2];
		Analysis[] primalityAnalyses = new Analysis[(SIZE.length)*2];
```

```
                for (int i = 0; i < SIZE.length; i++) {
                        System.out.println("Generation test: " + SIZE[i]);

                        BigInteger[] x0pq = PRNGConstants.getX0PQ(SIZE[i]);
                        x0 = x0pq[0];
                        p = x0pq[1];
                        q = x0pq[2];
                        M = p.multiply(q);
                        bbs = new BBS(x0, M, SIZE[i]);

                        m = new BigInteger("2").pow(SIZE[i]);
                        lcg = new LCG(x0, a, c, m);

                        millerRabin = new MillerRabin(primalityPrecision, lcg);
                        fermat = new Fermat(primalityPrecision, lcg);

                        Analysis lcgAnalysis =
Analysis.generationAnalysis(lcg.getClass().getSimpleName(), SIZE[i], lcg, generationAmount);
                        Analysis bbsAnalysis =
Analysis.generationAnalysis(bbs.getClass().getSimpleName(), SIZE[i], bbs, generationAmount);
                        Analysis millerRabinAnalysis =
Analysis.primalityTestAnalysis(millerRabin.getClass().getSimpleName(), SIZE[i], millerRabin, lcg,
primeAmount);
                        Analysis fermatAnalysis =
Analysis.primalityTestAnalysis(fermat.getClass().getSimpleName(), SIZE[i], fermat, lcg,
primeAmount);

                        System.out.println(lcg.getClass().getSimpleName());
                        System.out.println(lcgAnalysis.toString());
                        System.out.println(bbs.getClass().getSimpleName());
                        System.out.println(bbsAnalysis.toString());
                        System.out.println(millerRabin.getClass().getSimpleName());
                        System.out.println(millerRabinAnalysis.toString());
                        System.out.println(fermat.getClass().getSimpleName());
                        System.out.println(fermatAnalysis.toString());
                        System.out.println("-------------------------------------------");

                        generationAnalyses[2*i] = lcgAnalysis;
                        generationAnalyses[(2*i)+1] = bbsAnalysis;
                        primalityAnalyses[2*i] = millerRabinAnalysis;
                        primalityAnalyses[(2*i)+1] = fermatAnalysis;

                }

                Table.generalStatisticsTable(generationAnalyses, "Geração de números pseudo
aleatórios");
                Table.generalStatisticsTable(primalityAnalyses, "Geração de números provavelmente
primos");

                Table.valuesTable(generationAnalyses, "Geração de números pseudo aleatórios");
                Table.valuesTable(primalityAnalyses, "Geração de números provavelmente primos");
        }
}
```

primalitytester/PrimalityTester.java

```
package primalitytester;

import java.math.BigInteger;

/**
 * Interface para classes que testam
 * a primalidade de números
 * @author douglas
 *
 */
public interface PrimalityTester {
        public static BigInteger ZERO = BigInteger.valueOf(0);
        public static BigInteger ONE = BigInteger.valueOf(1);
        public static BigInteger TWO = BigInteger.valueOf(2);
        public static BigInteger FOUR = BigInteger.valueOf(4);
        /**
```

```
         * Testa se um valor é (ou provavelmente é) um primo
         * @param value {@link BigInteger} com o valor as ser testado
         * @return boolean indicando se é (ou provavelmente é) primo
         */
        public boolean isPrime(BigInteger value);
}
```

primalitytester/MillerRabin.java

```
package primalitytester;

import java.math.BigInteger;
import prng.PRNG;

public class MillerRabin implements PrimalityTester{

        protected int k;
        protected PRNG prg;

        /**
         * Testador de primalidade MillerRabin com k rounds
         * @param k inteiro que indica o número de bases à
         * à testar (proporcional à precisão dos testes)
         * @param prg {@link PRNG} gerador de números aleatórios
         * para gerar as bases a
         */
        public MillerRabin(int k, PRNG prg) {
                this.k = k;
                this.prg = prg;
        }


        /**
         * Testa se um valor provavelmente é um primo
         * @param value {@link BigInteger} com o valor as ser testado
         * @return boolean indicando se provavelmente é primo
         */
        @Override
        public boolean isPrime(BigInteger value) {
                BigInteger d, valueMinusOne,valueMinusFour;
                int s = 1;
                d = PrimalityTester.ZERO;
                valueMinusOne = value.subtract(PrimalityTester.ONE);
                valueMinusFour = value.subtract(PrimalityTester.FOUR);

                //complexidade
                while(!d.testBit(0)) {
                        d = value.subtract(PrimalityTester.ONE).divide(PrimalityTester.TWO.pow(s));
            s++;
                }

                //k
                WitnessLoop: for (int i = 0; i < k; i++) {
                        BigInteger a = prg.next().mod(valueMinusFour).add(PrimalityTester.TWO);
                        BigInteger x = a.modPow(d, value);


                        if(x.equals(PrimalityTester.ONE) || x.equals(valueMinusOne)) {
                                continue WitnessLoop;

                        }
                        for (int j = 0; j < s-1; j++) {
                                x = x.modPow(PrimalityTester.TWO, value);
                                if(x.equals(valueMinusOne)) {
                                        continue WitnessLoop;
                                }
                        }
                        return false;
                }
                return true;
        }
}
```

primalitytester/Fermat.java

```java
package primalitytester;

import java.math.BigInteger;
import prng.PRNG;

public class Fermat implements PrimalityTester {
	protected int k;
	protected PRNG prg;


	/**
	 * Testador de primalidade de Fermat com k rounds
	 * @param k inteiro que indica o número de bases à
	 * à testar (proporcional à precisão dos testes)
	 * @param prg {@link PRNG} gerador de números aleatórios
	 * para gerar as bases a
	 */
	public Fermat(int k, PRNG prg) {
		this.k = k;
		this.prg = prg;

	}

	/**
	 * Testa se um valor provavelmente é um primo
	 * @param value {@link BigInteger} com o valor as ser testado
	 * @return boolean indicando se provavelmente é primo
	 */
	@Override
	public boolean isPrime(BigInteger value) {
		BigInteger valueMinusOne,valueMinusFour;
		valueMinusOne = value.subtract(PrimalityTester.ONE);
		valueMinusFour = value.subtract(PrimalityTester.FOUR);

		if(value.compareTo(PrimalityTester.FOUR) == -1) {
			if(value.compareTo(PrimalityTester.ONE) == 1) {
				return true;
			}
			return false;
		}

		for (int i = 0; i < k; i++) {
			BigInteger a =
prg.next().abs().mod(valueMinusFour).add(PrimalityTester.TWO);

			if (!a.modPow(valueMinusOne, value).equals(PrimalityTester.ONE))
				return false;
		}
		return true;
	}
}
```

prng/PRNG.java

```java
package prng;

import java.math.BigInteger;

/**
 * Interface para geradores de números pseudo-aleatórios
 * @author douglas
 *
 */
public interface PRNG {

	public static BigInteger TWO = BigInteger.valueOf(2);
	/**
	 * Obtém o próximo valor da sequência pseudo-aleatória.
	 * @return {@link BigInteger} com o próximo valor da sequência pseudo-aleatória.
```

```
         */
        public BigInteger next();
}
```

prng/LCG.java

```
package prng;

import java.math.BigInteger;

/**
 * Classe que implementa o gerador de números
 * pseudo-aleatórios LCG
 * @author douglas
 *
 */
public class LCG implements PRNG{

        protected BigInteger xn, a, c, m;

        /**
         * Construtor.
         * @param x0 {@link BigInteger} que define o número usado como seed.
         * @param a {@link BigInteger} que define a constante multiplicadora.
         * @param c {@link BigInteger} que define a constante de incremento.
         * @param m {@link BigInteger} que define o módulo.
         */
        public LCG(BigInteger x0, BigInteger a, BigInteger c, BigInteger m) {
                this.xn = x0;
                this.a = a;
                this.c = c;
                this.m = m;
        }

        @Override
        public BigInteger next() {
                xn = xn.multiply(a).add(c).mod(m);
                return xn;
        }
}
```

prng/BBS.java

```
package prng;

import java.math.BigInteger;

/**
 * Classe que implementa o gerador de números
 * pseudo-aleatórios BlumBlumShub
 * @author douglas
 *
 */
public class BBS implements PRNG{

        protected BigInteger xn, M;
        protected int size;

        /**
         * Construtor.
         * @param x0 {@link BigInteger} que define o número usado como estado inicial.
         * @param M {@link BigInteger} inteiro que define o módulo.
         */
        public BBS(BigInteger x0, BigInteger M, int size) {
                this.xn = x0;
                this.M = M;
                this.size = size;
        }

        @Override
        public BigInteger next() {
                xn = xn.modPow(PRNG.TWO, M);
```

```
                /*
                 * corrige a quantidade de bits,
                 * mantendo os bits mais significativos
                 */
                int diff = xn.bitLength() - size;
                if(diff > 0) {
                        return xn.shiftRight(diff);
                }
                return xn;
        }
}
```

prng/PRNGConstants.java

```
package prng;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.math.BigInteger;
import java.net.URL;
import java.nio.file.FileSystems;

public class PRNGConstants {

        /**
         * https://primes.utm.edu/curios/index.php?start=143&stop=700
         * Retorna valor para x0, p e q.
         * Os valores estão salvos no arquivos em prng/constants/
         * com nome de acordo com o tamanho de geração.
         * x0 é primo ou um primo vezes uma potencia de 2;
         * p e q são primos congruentes à 3 mod 4;
         * Para um valor de size, p*q é coprimo à x0.
         * É claro que eu poderia ter salvo M diretamente,
         * mas quis salvar p e q para apresentar M exatamente
         * como o produto de primos congruentes à 3 (mod 4)
         * @return vetor de dois BigIntegers contendo p e q
         */
        public static BigInteger[] getX0PQ(int size) {
                BigInteger x0,p,q;

                File file = null;
                BufferedReader br = null;
                URL url = null;

                try {
                        url =
PRNGConstants.class.getResource("constants"+FileSystems.getDefault().getSeparator()+size);
                        file = new File(url.getPath());
                        br = new BufferedReader(new FileReader(file));
                        x0 = new BigInteger(br.readLine());
                        p = new BigInteger(br.readLine());
                        q = new BigInteger(br.readLine());

                        br.close();
                        return new BigInteger[] {x0,p,q};
                } catch (Exception e) {
                  e.printStackTrace();
                } finally {
                        if(br != null) {
                                try {
                                        br.close();
                                } catch (IOException e) {
                                        e.printStackTrace();
                                }
                        }
                }
                return null;
        }
}
```

analysis/Analysis.java

```java
package analysis;

import java.math.BigInteger;
import java.util.Arrays;
import primalitytester.PrimalityTester;
import prng.PRNG;

/**
 * Classe para realizar análise sobre
 * os resultados de geração de número pseudo-aleatórios
 * e números provavelmente primos
 * @author douglas
 *
 */
public class Analysis {

	protected String algorithm;
	protected int size;
	protected double mean, stddev, min, max;
	protected double[] timeElapsed;
	protected BigInteger[] generatedValues;

	/**
	 * Carrega média, desvio padrão, valor mínimo,
	 * valor máximo de um conjunto de valores
	 * @param mean média
	 * @param stddev desvio padrão
	 * @param min valor mínimo
	 * @param max valor máximo
	 * @param values conjunto de valores
	 */
	public Analysis(String algorithm, int size, double mean, double stddev, double min, double
max, double[] timeElapsed, BigInteger[] generatedValues) {
		super();
		this.algorithm = algorithm;
		this.size = size;
		this.mean = mean;
		this.stddev = stddev;
		this.min = min;
		this.max = max;
		this.timeElapsed = timeElapsed;
		this.generatedValues = generatedValues;
	}

	/**
	 * Dada uma lista de valores em ponto flutuante,
	 * obtém e média, desvio padrão, menor e maior valor
	 * em um objeto do tipo analysis
	 * @param values
	 * @return
	 */
	public static Analysis analyse(String algorithm, int size, double[] timeElapsed,
BigInteger[] generatedValues) {

		double[] values = Arrays.copyOf(timeElapsed, timeElapsed.length);
		double sum = 0;
		double mean, variance, stddev, min = Double.MAX_VALUE, max = 0;
		double q1, q3, iqr, lowerFence, upperFence;
		int samples = values.length, medianIndex;

		Arrays.sort(values);

		/*
		 * Calcula Q1, Q2 e IQR definindo limites superior e inferior para detecção de
		 * outliers
		 */
		medianIndex = midIndex(0, values.length);
		q1 = values[midIndex(0, medianIndex)];
		q3 = values[midIndex(medianIndex, values.length)];
		iqr = q3 - q1;
		lowerFence = q1 - 1.5 * iqr;
```

```
                upperFence = q3 + 1.5 * iqr;

                /*
                 * Média, mínimo e máximo de valores não outliers "Retira" outiliers da lista
                 * marcando-os com -1 já que são valores de tempo decorrido sempre positivos
                 */
                for (int i = 0; i < values.length; i++) {
                        if (values[i] >= lowerFence && values[i] <= upperFence) {
                                sum += values[i];
                                if (min > values[i]) {
                                        min = values[i];
                                }
                                if (max < values[i]) {
                                        max = values[i];
                                }
                        } else {
                                values[i] = -1;
                                samples--;
                        }
                }
                mean = (double) sum / samples;

                sum = 0;
                /*
                 * Variância e desvio padrão desconsiderando outliers
                 */
                for (int i = 0; i < values.length; i++) {
                        if (values[i] > -1) {
                                sum += Math.pow(values[i] - mean, 2);
                        }
                }
                variance = sum / samples;
                stddev = Math.sqrt(variance);

                return new Analysis(algorithm, size, mean, stddev, min, max, timeElapsed,
generatedValues);
        }

        /**
         * Gera n números pseudo-aleatórios com o gerador prng, contabilizando e
         * retornando os tempos de execução para geração
         * @param {@link PRNG} prng para geração
         * @param n      inteiro que indica o número de valores à gerar
         */
        public static Analysis generationAnalysis(String algorithm, int size, PRNG prng, int n) {
                long start;
                double[] timeElapsed = new double[n];
                BigInteger[] randomNumbers = new BigInteger[n];

                for (int i = 0; i < n; i++) {
                        start = System.nanoTime();

                        randomNumbers[i] = prng.next();

                        timeElapsed[i] = (System.nanoTime() - start) / 1000.0;
                }

                return Analysis.analyse(algorithm, size, timeElapsed, randomNumbers);
        }



        /**
         * Por n vezes,
         * gera números pseudo-aleatórios com o gerador prng
         * até que se satisfaça a condição de testador pt
         * contabilizando e retornando os n tempos de execução
         * para encontrar os n prováveis primos
         *
         * @param {@link PrimalityTester} pt para teste dos números gerados
         * @param {@link PRNG} prng para geração
         * @param n inteiro que indica o número de prováveis primos que devem
         * ser encontrados
         */
```

```
	public static Analysis primalityTestAnalysis(String algorithm, int size, PrimalityTester
pt, PRNG prng, int n) {

		long start;
		double[] timeElapsed = new double[n];
		BigInteger[] randomNumbers = new BigInteger[n];

		for (int i = 0; i < n; i++) {
			start = System.nanoTime();

			boolean notPrime = true;
			while (notPrime) {
				randomNumbers[i] = prng.next();
				if (randomNumbers[i].testBit(0)) {
					notPrime = !pt.isPrime(randomNumbers[i]);
				}
			}

			timeElapsed[i] = (System.nanoTime() - start) / 1000.0;
		}

		return Analysis.analyse(algorithm, size, timeElapsed, randomNumbers);
	}

	@Override
	public String toString() {
		return "Média: " + mean + "; Desvio padrão: " + stddev + "; Min: " + min + "; Max: "
+ max;
	}

	public String getAlgorithm() {
		return algorithm;
	}

	public int getSize() {
		return size;
	}

	public double getMean() {
		return mean;
	}

	public double getStddev() {
		return stddev;
	}

	public double getMin() {
		return min;
	}

	public double getMax() {
		return max;
	}

	public double[] getTimeElapsed() {
		return timeElapsed;
	}

	public BigInteger[] getGeneratedValues() {
		return generatedValues;
	}

	protected static int midIndex(int l, int r) {
		int n = r - l + 1;
		n = (n + 1) / 2 - 1;
		return n + l;
	}
}
```

display/Table.java

```java
package display;

import java.awt.Dimension;
import java.awt.GridLayout;
import java.awt.Toolkit;
import java.util.ArrayList;
import java.util.List;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;

import analysis.Analysis;

/**
 * Classe para apresentar tabelas de valores
 * @author douglas
 *
 */
public class Table {


	public static void generalStatisticsTable(Analysis[] analyses, String title) {
		String[] columns = { "Algoritmo", "Tamanho", "Quantidade", "Média", "Mínimo",
"Máximo", "Desvio padrão"};
		Object[][] data = new Object[analyses.length][columns.length];

		JFrame frame = new JFrame(title);
		JPanel panel;
		JTable table;
		JScrollPane scroll;

		for (int i = 0; i < analyses.length; i++) {
			data[i][0] = analyses[i].getAlgorithm();
			data[i][1] = analyses[i].getSize();
			data[i][2] = analyses[i].getTimeElapsed().length;
			data[i][3] = analyses[i].getMean();
			data[i][4] = analyses[i].getMin();
			data[i][5] = analyses[i].getMax();
			data[i][6] = analyses[i].getStddev();
		}

		panel = new JPanel();
		panel.setLayout(new GridLayout(1, 1));
		table = new JTable(data, columns);
		scroll = new JScrollPane(table);
		panel.add(scroll);

		Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
		Dimension size = new Dimension(screenSize.width/2, screenSize.height/2);
		frame.getContentPane().add(panel);
		frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
		frame.setSize(size);
		frame.setLocationRelativeTo(null);
		frame.setVisible(true);
	}


	public static void valuesTable(Analysis[] analyses, String title) {
		String[] columns = { "Algoritmo", "Tamanho", "Número", "Tempo para gerar" };
		List<Object[]> data = new ArrayList<>();

		JFrame frame = new JFrame(title);
		JPanel panel;
		JTable table;
		JScrollPane scroll;

		for (int i = 0; i < analyses.length; i++) {
			for (int j = 0; j < analyses[i].getTimeElapsed().length; j++) {
				Object[] row = new Object[columns.length];
				row[0] = analyses[i].getAlgorithm();
				row[1] = analyses[i].getSize();
				row[2] = analyses[i].getGeneratedValues()[j];
```

```
                                row[3] = analyses[i].getTimeElapsed()[j];
                                data.add(row);
                        }
                }

                panel = new JPanel();
                panel.setLayout(new GridLayout(1, 1));
                table = new JTable(data.toArray(new Object[data.size()][]), columns);
                scroll = new JScrollPane(table);
                panel.add(scroll);

                Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
                Dimension size = new Dimension(screenSize.width/2, screenSize.height/2);
                frame.getContentPane().add(panel);
                frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
                frame.setSize(size);
                frame.setLocationRelativeTo(null);
                frame.setVisible(true);
        }
}
```

prng/constants/40

```
553559562581
9187
87613571
```

prng/constants/56

```
69668002914515347
23010067
1879048183
```

prng/constants/80

```
1015402101225201202154011
1879048183
347811194367163
```

prng/constants/128

```
236918336221672442100866320173263025989
3868168229228618683
71322723161814151019
```

prng/constants/224

```
252729313335373941434547495153555759616365676971737577798183858789 91
151322281731701525311159232490 71
136398358416995989861656309247978008 7
```

prng/constants/256

```
799999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
768231807465763655682670928358014951
915212527333539454951555763656975778185 87
```

prng/constants/512

```
799441209771611054812721173333160052293377675704670764996367396268620083843295023910398107072836959981631464648272070682601836018119684315422474838221101913169525310647365859726838724295606890549323807888004534353641360687318060281490199180612328166730772686396383698676545930088884461843637361053498018365439
```

prng/constants/1024

```
11592217955149597338341017634264372233455725568287960586483880629365961962500430320625038439254685
50638441069651562879517493876341125510892845955411036927165287748763116417009299869880231972422245
81099872580798960693521778607396791006450968430359009613295725905514216842343121690916290236558767
890728449777
76543234567654323456765432345676543234567654323456765432345676543234567654323456765432345676543234
5676543234567654323456765432345676543234567
15772837059492543173572130144338799397358769266508888881444826001811896226765682529114078607640990
57228234707758276963354904515920197804734678270581258722822164851844205868348711
```

prng/constants/2048

```
23588765696000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000010301210624
10405118339442999672731295480463911385769358680837048371637289970216805381845901924659360422432111
2479056098651
16813234007896699147923484304021416381400887913100505019998268605227519286257928181220084249817684
42434803335415398044412696568557957902467185360056613902009619180847049207160143515075314062539482
35628729161481417569772046047249313751887076455262444135248318465774602119797030821262907700716028
32977491720976012880743032497708424779403739757905156000351023751229621421498238328162929719744526
77133016906339873142680869331594034032404538402135764261055493104239782418100659487707956316397030
34173006193345651
```

prng/constants/4096

```
74263032744074110814878611467026973537502214463504924168934987943322533190530044517046760494675224
47969978875372007846633955130215290743885014195451055435155442722142454074444882314730742425461770
85577157775846856965377145043346769635215640257856211488697319113469977520112233151214172250732216
58212523944011815939385927735803608856223601423098284949026300302201971643174732718008208818365023
45563202908757202579109278717756794067287734634047698107433003015147748525249010918826206160149517
33424786123498606625811490619558336667239380128164722559473859598362138074113777371618035471993339
79381288958047612884640229968033712204501318878113047306711935512870312823060647702596098929495239
46053596005009104799348363708537960773608620309558267493213735423130156453455087039091657303716402
88418733351679387288498265597147791231485059395051523869331426779021445339108807183631327036620661
84299941958250075963566996547991752303149883200510368910258809441389930819117031176749921231943774
88626555395172339737470404571974903324037714790950872799318381939917824694643318972939365897130642
44256439693030258773699842170359962161668010567734874593776266029669597127594756153686034090081630
211842216075733091627369619313618939118371931558632256448
7666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666667
10609807089687956886618331228024772126304617570231850599197815334721347178200702497579840810826492
28451705998731862074176182853518645443624918601695671024343710458664503076901280899702585841576989
25788106177354317191297297775317430007078143561240470823958484382353873640014489493443753964136843
52277092172904315923103574877854240526487133602110426847732804928059329224325097411740676621392095
39802498386450953729247751019564655842415524896797759694700841143525426394840882255289891782735720
82160810131842291046394903591691405065851279680939714825701540334748124469905644112793580463203720
75195179686844596277640063103137409884981390576116988534386703128128432915134033483962743207159436
84146881864345283522961655407281022066761066500206378251940770365450740175793931208006782183204910
06627215320889105076519834088971029185585352120636074204066397443411386876921029225141763851748009
72177673413562655119174387153187801414043554652769946611845491564611539497820925939355344123769431
06484277863547306264788424982767007591646193008832028665831788958278794605333848516232771596581889
02856952424548318228910390912389681630744571669218427082451
```

Referências

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

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

https://en.wikipedia.org/wiki/Primality_test

https://en.wikipedia.org/wiki/Pseudorandom_number_generator

https://en.wikipedia.org/wiki/Quadratic_residue

https://en.wikipedia.org/wiki/Linear_congruential_generator

https://en.wikipedia.org/wiki/Blum_Blum_Shub