Segurança em Computação: Trabalho Números Primos

Rafael Begnini de Castilhos (20205642)

06 de junho de 2022

Resumo

O presente trabalho possui como objetivo analisar o tempo e complexidade de algoritmos para gerar e validar números primos.

Sumário

1 Introdução

O presente trabalho apresenta a implementação de algoritmos para geração de números pseudo-aleatórios, além de realizar testes e análises se esse número gerado é primo. A linguagem *Java* foi escolhida por maior afinidade e também pelo motivo de possuir o pacote *BigInteger*, assim pode fornecer suporte para todos os operadores de número positivos do *Java*, facilitando de modo geral a escrita e compreensão dos algoritmos.

2 Gerar Números Pseudo-aleatórios

Os dois algoritmos de geração de números pseudo-aleátórios implementados foram *Blum Blum Shub* e *Linear Congruential*. Os códigos fonte correspondentes a implementação de cada algoritmo estão abaixo:

2.1 Blum Blum Shub

Blum Blum Shub (BBS) é um gerador de números pseudoaleatórios proposto por Lenore Blum, Manuel Blum e Michael Shub em 1986. Esse gerador produz um sequência de bits de acordo com a implementação a seguir:

```
/*
 Implementaao do algoritmo Blum Blum Shub para gerar numeros
    pseudo-aleatorio.
Calcula o valor do n, que eh utilizado como modulo.
Em seguida, eh realizado o clculo do x inicial, no qual ser o
    numero que dara origem aos demais numeros da sequncia e tambehm
    um vetor de bytes eh inicializado com a quantidade de bytes que
    o numero pseudo-aleatorio deve possuir.
Os prximos valores de x so calculados e realizado o modulo para
    ver se x eh par ou mpar, determinando se o respectivo bit eh 0
    ou 1.
Os bits so agrupados em bytes, atravehs da soma dos bits que tem o
    valor 1, utilizando a logica binaria, para ento criar um numero
    pseudo-aleatorio grande.
@param
p(int) e q(int): numeros primos grandes, com resto 3 quando
    divididos por 3;
```

```
s(int): eh relativamente primo para n, ou seja, nem p nem q eh um
    fator de s;
size_k e y(int): numero de bits do numero desejado .
@return
number(BigInteger) numero pseudo-aleatorio grande, com numero de
    bits igual a size_key.
*/
public static BigInteger blumBlumShub(int p, int q, int s, int
    size_key) {
    if (p % 4 != 3)
        return null;
    if (s % p == 0 || s % q == 0)
        return null;
    int aux;
    int sum = 0;
    int position = 0;
    int n = p * q;
    int x = (int) Math.pow(s , 2) % n;
    byte data [] = new byte [size_key / 8];
    for (int i = 0; i < size_key; i++) {
        x = (int) Math.pow(x, 2) % n;
        aux = i % 8;
        if (aux != 7){
            if (x % 2 == 1)
                sum += (int) Math.pow(2 , aux);
        } else {
            data [position] = (byte) sum;
            position ++;
            if (x % 2 == 1)
                sum = 1;
            else
                sum = 0;
        }
    }
    return new BigInteger(data);
}
```

2.2 Linear Congruential

Um gerador congruencial linear (LCG) é um algoritmo que produz uma sequência de números pseudo-aleatórios calculados com uma equação linear descontínua por partes. O método representa um dos mais antigos e conhecidos algoritmos geradores de números pseudoaleatórios. O número aleatório é obtido através da seguinte implementação:

```
/*
Implementacao do algoritmo Linear Congruential Generators para
    gerar numeros pseudo-aleatorios.
Inicialmente eh calculado o numero de valores de x que precisa
    calcular e eh inicializado com a quantidade de bytes que o
    numero pseudo-aleatorio deve possuir.
Em seguida, cada inteiro x eh calculado e dividido em bytes para
    entao criar um numero pseudo-aleatorio grande.
@param
modulus(int): tipicamente um valor primo proximo a 2^31;
multiplies(int): fator de multiplicacao a cada interacao;
increment(int): utilizado para incrementar o valor de x, a cada
    interacao;
seed(int): valor inicial do algoritmo;
size_key(int): numero de bits do numero desejado.
@return
number(BigInteger): numero pseudo-aleatorio grande, com numero de
    bits igual a size_key.
*/
public static BigInteger linearCongruential(int modulus, int
    multiplier, int increment, int seed, int size_key) {
    if (modulus <= 0)
        return null;
    if (!(0 < multiplier && multiplier < modulus))
        return null;
    if (!(0 <= increment && increment < modulus))
        return null;
    if (!(0 <= seed && seed < modulus))
        return null;
    if (size_key % 32 != 0)
        return null;
    int x = seed;
```

```
    int aux = Math.floorDiv (size_key , 32);
    byte [] data = new byte [aux * 4];
    int position = 0;
    for (int i = 0; i < aux; i++) {
        x = (multiplier * x + increment) % modulus;
        data[position] = (byte) ((i& 0x000000FF) >> 24);
        data[position + 1] = (byte) ((i & 0x0000FF ) >> 16);
        data[position + 2] = (byte) ((i & 0x00FF) >> 8);
        data[position + 3] = (byte) (i & 0xFF);
        position = position + 4;
    }
    return new BigInteger(data);
}
```

2.3 Análise de tempo dos algoritmos

O gerador *Blum Blum Shub* é um gerador de bits pseudo-aleatório criptograficamente seguro, devido ao motivo de passar no teste de bit. Dado que não existe um algoritmo viável que possa permitir a afirmação que o próximo bit será 1 ou 0, com probabilidade maior que 50 por cento, sendo a sequência de bits imprevisível. A segurança desse gerador é baseada na dificuldade de fatoração n, já que é necessário determinar seus dois fatores primos p e q (Stallings, 1999).

No gerador *Linear Congruential* é desejável que a sequência real usada não seja reproduzıvel. Mas a seleção de valores de entrada para o algoritmo é uma etapa crıtica, pois poucos valores podem produzir sequências satisfatórias. A possível reprodução da sequência pode ser evitada um relógio interno do sistema para modificar o fluxo de números aleatórios(Stallings, 1999).

Após a implementação dos algoritmos, se fez necessária a geração de valores pseudo-aleatórios na quantidade de bits requisitadas: 40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096. No algoritmo *Blum Blum Shub* (Tabela 1) é possıvel gerar números para todas as quantidades de bits citadas, uma vez que apenas o último bit de cada valor intermediário é utilizado para formar o número pseudo-aleatório. Entretanto no gerador *Linear Congruential* (Tabela 1) não é viável produzir alguns tamanhos de números, devido à implementação eficiente com aritmética de 32 bits, ocorrendo a concatenação de números de 32 bits.

2.3.1 Tabela de comparação de tempo na geração de números pseudo-aleatórios

Tamanho do número em bits	Tempo em ns BBS	Tempo em ns LC
40	12631665	-
56	72934	-
80	109263	-
128	98222	679057
168	121165	-
224	137705	73734
256	166010	100409
512	279512	163106
1024	652492	458447
2048	1062432	1019763
4096	2273131	1843146

2.4 Análise de complexidade dos algoritmos

A execução do algoritmo *Blum Blum Shub* possui uma complexidade linear, proporcional ao tamanho do número pseudo-aleatório que se deseja gerar. E o algoritmo *Linear Congruential* também possui tempo linear, mas é proporcional ao tamanho do número dividido por 32, levando menos tempo para executar.

3 Verificação de Primalidade

"O problema de distinguir números primos de compostos e de decompor esses últimos em seus fatores primos é conhecido como sendo um dos mais importantes e úteis na aritmética... A dignidade da própria ciência parece requerer que todos os meios possíveis sejam explorados para a solução de um problema tão elegante e tão celebrado."(Gauss, 1801).

Um teste de primalidade refere-se a um algoritmo que tendo como entrada um inteiro positivo n, determina se n ´e ou n˜ao primo. H´a basicamente dois grandes grupos de algoritmos: os determinísticos e os não determinísticos. Os algoritmos do primeiro grupo determinam com exatidão a primalidade de um número, porém, na maioria dos casos, seu custo aumenta muito conforme aumentamos o número estudado. Em paralelo, estão os do segundo grupo,

que, embora sejam menos custosos, não determinam exatamente se o número é primo.

Os códigos fonte correspondentes a implementação do algoritmo de Miller-Rabin e do Teorema de Fermat, quanto a primalidade de números estão abaixo

3.1 Miller-Rabin

O teste de primalidade de Miller-Rabin é probabilıstico, ou seja, se o número não passar pelo teste, com certeza não é primo. Mas se o número passar no teste há grande probabilidade de ser primo. O teste pode ser implementado de acordo com o código abaixo, também é necessário a implementação do cálculo para determinar o número ímpar e quantas divisões por 2 são realizadas:

```
/*
Implementacao do algoritmo de Miller-Rabin para verificar se um
    numero n eh primo.
Calcula seus valores k, q e um numero aleatorio para verificar a
    primalidade de n.
Entao, executa um loop com k interacoes e verificando a sua
    primalidade.
@param
number(BigInteger): numero que vai ser verificado se eh primo.
@return
(boolean): se falso o numero nao eh primo, se verdadeiro
    provavelmente eh primo.
*/
public static boolean miller_rabin (BigInteger number) {
    if
        (number.mod(BigInteger.valueOf(2)).equals(BigInteger.valueOf(0))
        || number.equals(
            BigInteger.valueOf(0)) ||
                number.equals(BigInteger.valueOf(1)))
        return false;
    BigInteger data[] = calculate_k_q(number);
    BigInteger k = data [0];
    BigInteger q = data [1];
    Random rand = new Random();
    BigInteger a;
```

```
    do {
        a = new BigInteger (number.bitLength (), rand);
    } while (a.compareTo (BigInteger.valueOf(2)) == -1 ||
        number.subtract(BigInteger.valueOf(1)).compareTo(a) == -1);
    if (a.modPow(q, number).equals(BigInteger.valueOf(1)))
        return true;
    BigInteger aux;
    BigInteger i = BigInteger.valueOf(0);
    while (i.compareTo(k) == -1) {
        aux = BigInteger.valueOf(2).modPow(i, number).multiply(q);
        if (a.modPow(aux,
            number).equals(number.subtract(BigInteger.valueOf(1))))
            return true;
        i = i.add(BigInteger.valueOf(1));
    }
    return false;
}

/*
Note que n -1 eh um inteiro par. Em seguida, divida (n -1) por 2
    ate que o resultado
seja um numero impar q, para um total de k divisoes.
@param
number (BigInteger): numero que vai ser verificado se eh primo.
@return
data (BigInteger[2]): data [0] eh a quantidade de divisoes que foi
    realizado e data [1] eh o valor de quando o resultado das
    sucessivas divisoes nao eh inteiro .
*/
public static BigInteger [] calculate_k_q (BigInteger number) {
    BigInteger k = BigInteger.valueOf(0);
    BigInteger q = number.subtract(BigInteger.valueOf(1));
    while
        (q.mod(BigInteger.valueOf(2)).equals(BigInteger.valueOf(0)))
        {
        k = k.add(BigInteger.valueOf(1));
        q = q.shiftRight(1);
    }
    BigInteger [] data = new BigInteger [2];
    data [0] = k;
```

```
    data [1] = q;
    return data;
}
```

3.2 Fermat

O teste de primalidade de Fermat foi escolhido por sua simplicidade e eficiência. Nesse teorema se o número não passar pelo teste, com certeza não é primo. O teste pode ser implementado da seguinte forma:

```
/*
Implementacao do teorema de Fermat.
Se o numero for menor que 2 ou divisivel por dois, entao o numero
    nao eh primo.
Segundo o teorema, se um numero n eh primo, qualquer numero entre
    1 e n-1 elevado a n -1 modulo n, o resultado sera 1.
Se existir um numero que nao atende a esta condicao , entao , com
    certeza , n nao e um numero primo.
Caso contrario, n eh, com uma grande chance primo.
@param
number (BigInteger): numero que vai ser verificado se eh primo.
@return
(boolean): se falso o numero nao eh primo, se verdadeiro
    provavelmente eh primo.
*/
public static boolean fermat (BigInteger number) {
    if (number.mod
        (BigInteger.valueOf(2)).equals(BigInteger.valueOf(0)) ||
        number.equals(BigInteger.valueOf(0)) ||
        number.equals(BigInteger.valueOf(1)))
        return false;
    Random rand = new Random();
    BigInteger a, result;
    for (int i = 0; i < number.bitLength(); i++) {
        do {
            a = new BigInteger (number.bitLength(), rand);
        } while (a.compareTo(BigInteger.valueOf(2)) == -1 ||
            number.subtract(BigInteger.valueOf(1)).compareTo(a) ==
            -1);
```

```
            result = a.modPow(number.subtract(BigInteger.valueOf(1)),
                number);
            if (!result.equals(BigInteger.valueOf(1)))
                return false;
            else
                continue;
        }
        return true;
    }
```

3.3 Análise de tempo dos algoritmos

Os geradores de números pseudo-aleatórios necessitam que a sequência real usada não seja reproduzível, caso esse efeito ocorrer o algoritmo não possui segurança. Isso somado ao fato da necessidade de números primos para sistemas seguros, faz com que esse processo exija considerável poder computacional. Muitas vezes o número gerado não é primo, tendo que gerar outros números até que um primo seja gerado. Assim, gerar e testar a primalidade desses números primos grandes (40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048 e 4096 bits) pode apresentar dificuldades.

3.3.1 Tabela de comparação de tempo na validação de primalidade

Tamanho do número em bits	Tempo em ns Miller-Rabin	Tempo em ns Fermat
40	286008	443712
56	11763	8513
80	6468	4090
128	6891	4157
168	4313684	2107393
224	10582	7649
256	7569	11972
512	8151	8259
1024	10655	10259
2048	14975	15291
4096	23195	18302

3.4 Análise de complexidade dos algoritmos

A execução do algoritmo *Miller-Rabin* possui uma complexidade linear, proporcional ao número de divisões por 2 realizadas até encontrar um número ímpar. E algoritmo de Fermat também possui tempo linear, mas é proporcional número, mas como este pode ser muito grande, é comum escolher alguns números menores para testar. Ambos os algoritmos podem demorar em achar um número randômico que satisfaça as condições.

4 Conclusão

Realizando esse trabalho foi possível colocar em prática os tópicos apresentado em sala de aula sobre números primos e verificação de primalidade, fazendo referência as possíveis abordagens na área da segurança da computação. Em suma, o trabalho cumpriu seus requisitos pedagógicos e didáticos para a formação de um profissional da ciência da computação.

Referências

[1] C. F. Gauss. Disquisitiones arithmeticae. Springer, 1801.

[2] W. Stallings. Cryptografhy and network security: Priciples and practice. Prentice Hall, 1999.