INE5429 - Segurança em Computação
Trabalho Individual II

16100725 - Fabíola Maria Kretzer

16 de abril de 2019

1 Introdução

O presente trabalho apresenta a implementação de algoritmos para geração de números pseudo-aleatórios e para testar se esse número gerado é primo. A linguagem *Java* foi escolhida por possuir o pacote *BigInteger*, no qual fornece análogos para todos os operadores de números primitivos do *Java*, facilitando a escrita, leitura e compreensão de algoritmos que necessitam números grandes.

2 Gerador de Números Aleatórios

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

2.1 Blum Blum Shub

Um algoritmo popular para gerar números pseudo-aleatórios seguros é conhecido como gerador *Blum Blum Shub* (Stallings, 1999). Esse gerador produz um sequência de *bits* de acordo com a implementação a seguir:

```
/*
    Implementacao 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
    calculo do x inicial, no qual sera o numero que dara origem aos demais numeros da
    sequencia e tambem um vetor de bytes eh inicializado com a quantidade de bytes que
    o numero pseudo-aleatorio deve possuir. Os proximos valores de x sao calculados e
    realizado o modulo para ver se x eh par ou impar, determinando se o respectivo bit
    eh 0 ou 1. Os bits sao agrupados em bytes, atraves da soma dos bits que tem o valor
    1, utilizando a logica binaria, para entao 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_key (int): numero de bits do numero desejado.

    @return
        number (BigInteger): numero pseudo-aleatorio grande, com numero de bits igual a
            size_key.
*/

public BigInteger blum_blum_shub (int p, int q, int s, int size_key) {

```

```
    if (p % 4 != 3 || 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;
        }
    }

    BigInteger number = new BigInteger(data);

    return number;
}
```

Código 1: implementação do algoritmo de geração de números pseudo-aleatórios Blum Blum Shub

2.2 Linear Congruential

Outro algoritmo amplamente utilizada para geração de números pseudo-aleatórios é *Linear Congruential* (Stallings, 1999). 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 BigInteger linear_congruential (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;
    }

    BigInteger number = new BigInteger(data);

    return number;
}
```

Código 2: implementação do algoritmo de geração de números pseudo-aleatórios Linear Congruential

2.3 Análise dos algoritmos

O gerador *Blum Blum Shub* é um gerador de bits pseudo-aleatório criptograficamente seguro, pelo motivo de passar no teste de *bit* (Stallings, 1999). Dado que não existe um algoritmo viável que possa permitir que você afirme que o próximo bit será 1 (ou 0) com probabilidade maior que 1/2, sendo a sequência de *bits* imprevísivel (Stallings, 1999). 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 reproduzível (Stallings, 1999). Mas a seleção de valores de entrada para o algoritmo é uma etapa crítica, pois poucos valores podem produzir sequências satisfatórias (Stallings, 1999). 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 possível 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. Enquanto no gerador *Linear Congruential* (Tabela 1) não é viável produzir alguns tamanhos de números, pois esse algoritmo é implementado eficientemente com aritmética de 32 bits, ocorrendo a concatenação de números de 32 bits.

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

Tabela 1: Tempo para gerar um número pseudo-aleatório de diferentes tamanhos usando o algoritmo *Blum Blum Shub*

	Tempo para gerar (em nanosegundos)	
Tamanho do número (em bits)	Blum Blum Shub	Linear Congruential
40	715606	-
56	723018	-
80	757903	-
128	742202	737182
168	771242	-
224	726315	779814
256	788319	733901
512	797538	763406
1024	813868	757934
2048	886454	762185
4096	1089005	738951

3 Teste de primalidade

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 probabilístico, ou seja, se o número não passar pelo teste, com certeza não é primo (Stallings, 1999). Mas se o número passar no teste há grande probablilidade de ser primo (Stallings, 1999). O teste pode ser implementado da seguinte forma:

```
/*
    Implementacao do o 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 inteiracoes 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 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);
```

```java

    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;

}
```

Código 3: implementação do teste de primalidade de Miller-Rabin

```java
/*
    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 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;
}
```

Código 4: implementação do cálculo para determinar o número ímpar e quantas divisões por 2 são realizadas

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 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;

}
```

Código 5: implementação do teste de primalidade de Fermat

3.3 Análise do 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.

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) apresenta muitas dificuldades.

Referências

Stallings, William. Cryptografhy and Network Security: Priciples and Practice. Prentice Hall, 1999. 569p.