INE5429 - Relatório do Trabalho 1

Rodrigo Pedro Marques

05 de Abril de 2017

1 Introdução

O objetivo deste trabalho é implementar dois algoritmos geradores de números pseudo-aleatórios e em seguida analisar a primalidade dos mesmos através dos método "Miller-Rabin" e um outro método escolhido pelo aluno.

A linguagem escolhida para implementar estes algoritmos foi **Python**. Isto se deve ao fato de ela ser uma linguagem bastante simples, eficiente e de rápida implementação. Em questão dos algoritmos geradores de números pseudo-aleatórios, foram escolhidos o "*Linear Congruential Generator*" e o "*Blum Blum Shub*". Não houve nenhuma medida especial para escolhe-los.

2 Algoritmos

2.1 *Linear Congruential Generator*

O primeiro algoritmo gerador de números pseudo-aleatórios escolhido foi o "*Linear Congruential Generator*" (LCG). Este algoritmo utiliza uma função linear em trechos para gerar os números e é definido pela seguinte relação:

$X_{n+1} = (aX_n + c) mod(m)$

tal que:

módulo **m**: $0 < m$

multiplicador **a**: $0 < a < m$

incrementador **c**: $0 \leq c < m$

a semente ou valor inicial X_0: $0 \leq X_0 < m$

É importante ressaltar que o incrementador tem um papel importante pois se $c = 0$, o gerador é chamado de "*Multiplicative Congruential Generator*" (MCG), ou *Lehmer RNG*, caso $c \neq 0$, o gerador é chamado de "*Mixed Congruential Generator*". Outra curiosidade é que, os LCGs mais eficientes possuem um valor de m na potência de dois, como $m = 2^{32}$ ou $m = 2^{64}$, assim, é possível computar o módulo apenas truncando todos os bits exceto os 32 ou 64 mais significativos. Este algoritmo não é recomendado para criptografia pois ele é fácil de se prever uma vez que a semente é conhecida.

2.2 *Blum Blum Shub*

O segundo algoritmo gerador de números pseudo-aleatórios escolhido foi o “Blum Blum Shub” (BBS). Este algoritmo utiliza a seguinte relação:

$x_{n+1} = ((x_n)^2 mod(M))$

tal que:

$M = pq$, onde p e q é o produto de dois números primos.

A semente deve ser um número inteiro co-primo de M e diferente de 1 ou 0.

p e q devem ser congruentes à $3mod4$, garantindo que o resto quadrático tenha apenas uma raiz quadrada, e o maior divisor comum deve ser pequeno.

2.3 Miller-Rabin

Miller-Rabin é um algoritmo que testa a primalidade de um número, isto é, se um dado número é primo ou não. Basicamente, este algoritmo faz testes sobre uma igualdade ou um conjunto de igualdades verdadeiras para números primos, assim, ele verifica quando estas igualdades possuem um número que querem testar a primalidade.

O método consiste em realizar por um número de iteração i a verificação de primalidade sobre o número p, realizando operações modulares sobre a decomposição do valor $p - 1$. É necessário verificar se o número $p - 1$ pode ser decomposto na forma $2^s * d$ e usar estes valores e a seleção de um valor aleatório entre $1 < a < n - 1$ e computar i vezes. Segue a fórmula:

$a^{2^i * d}\ mod\ p = x$

Caso o valor da resposta $x\ mod\ p$ com $i = 0$ seja congruente a 1, ou para $i > 0$ a resposta $x\ mod\ p$ seja congruente a -1 temos que o número p tem fortes indícios de ser um primo, ou, de que o valor de a é um valor “falso positivo” de p é primo.

2.4 Fermat

O algoritmo de “Fermat” é outro algoritmo para testar a primalidade de um número. O seu teorema declara que se p é primo e $1 < a < p$, então, $a^{p-1} \equiv 1(modp)$. Assim, para testar se um número é primo ou não, deve-se escolher um número a aleatório dentro do intervalo indicado e verificar quando a igualdade se mantém. Se a igualdade não se manter, então p é composto, porém, caso a igualdade se mantenha para diversos números distintos a, então p provavelmente é primo.

3 Implementação

3.1 *Linear Congruential Generator* e *Blum Blum Shub*

A implementação do algoritmo *Linear Congruential Generator* se encontra no arquivo “lcg.py” e a implementação do algoritmo *Blum Blum Shub* se encontra no arquivo “blumblumshub.py” (ambos os arquivos se encontram no mesmo diretório deste relatório).

Nos apêndices A e **??** é possível encontrar a implementação como foi pedido no enunciado. Como é possível observar, ambas as implementações necessitaram de poucas linhas e são fácil de se entender quando conhecemos como cada algoritmo funciona na prática. É importante ressaltar que a lista “listaDeTamanhos” armazena os valores

$40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096$ que são os tamanhos em bits binários dos números resultantes dos algoritmos. Na tabela 1 é possível observar os números gerados pelos algoritmos. Caso precisar observar o real tamanho deles, consulte os arquivos *lcg_output.txt* e *bbs_output.txt*

Tabela 1 – Números gerados pelos algoritmos

LCG	**BBS**
928581922306	7861836889
$4,5320.10^{16}$	$8,8849.10^{23}$
$4,4565.10^{23}$	$5,5121.10^{16}$
$4,9179.10^{32}$	$2,8442.10^{50}$
$5,4269.10^{41}$	$4.6572.10^{66}$
$5,9887.10^{50}$	$1,7696.10^{76}$
$6,6086.10^{59}$	$3,1315.10^{152}$
$7,2927.10^{68}$	$9,8067.10^{199}$
$8,0476.10^{77}$	$9,6173.10^{199}$
$9,8000.10^{95}$	$9,2492.10^{199}$

Como é possível analisar pela tabela 1, os *Blum Blum Shub* acaba gerando números de maiores grandezas no mesmo número de iterações sobre ambos os algoritmos. O tempo médio de cinco execuções do algoritmo *Linear Congruential Generator* de foi de 0.000365734100342 segundos contra 0.0010203361511242 segundos do *Blum Blum Shub*, onde o LCG gerou o resultado em cerca de $2.7x$ mais rápido. Já em questão de complexidade, ambos tem uma complexidade linear, visto que apenas uma função é executada.

3.2 Miller-Rabin e Fermat

Tabela 2 – Possíveis números primos gerados com LCG + Miller-Rabin

LCG + Miller-Rabin
999935289161
30698845911109471
314447169166764106865243
682990814815986408979692412892648321 77
118665923101618319259820048495220453036239551129689
2109170631572549970795645948364284696753342088991766737859271 7663299
139730992993915559946635326430792028438897653561752245950721121479175673858 33
$7,720830.10^{153}$
$5,3277402936.10^{199}$
$3,207224146.10^{199}$
$9,28350516360118.10^{199}$

A implementação do algoritmo “Miller-Rabin” e “Fermat” podem ser encontradas nos arquivos “millerrabin.py” e “fermat.py”, respectivamente, também é possível encontra-los no apêndice A. A escolha de de implementar o algoritmo de “Fermat” se deu ao fato de sua simplicidade de implementação e compreensão de como ele funciona. O algoritmo de “Fermat” é menos utilizado pois o seu desempenho é menor que o de “Miller”.

O meu computador demorou cerca de alguns segundos para gerar números primos de grandezas de 40 à 1024 bits. Porém, para calcular números primos de grandeza de 4096 bits, o computador chegou a levar de 15 minutos à horas. No arquivo "numeroPrimos.txt" e na tabela 2 é possível encontrar alguns possíveis números primos gerados através dos algoritmos "LCG" e "Miller".

Em relação à complexidade do algoritmo de "Miller-Rabin" é de $O(log^3 n)$, onde k é o número de diferentes valores que a assume durante o teste, contra a complexidade de $O(kn^p)$ de "Fermat". Isto é, o algoritmo de "Miller-Rabin" possui complexidade polinomial, contra a complexidade exponencial de "Fermat".

4 Conclusão

Os algoritmos de geração de números pseudo-aleatórios possuem um tempo de execução muito rápido em relação aos de verificação de primalidade de um número. Porém, ambos escolhidos aqui não são práticos para criptografia pois são previsíveis. Números primos grandes atualmente são importantes para segurança em computação pois são difíceis de fatorar e também ajudam nas funções de *hash* na criptografia (MD5, SHA1, etc). Como podemos perceber através deste trabalho prático, números na grandeza de 4096 bits ou mais, podem levar horas, semanas, anos para serem fatorados, o que ajuda na segurança.

A Códigos das Implementações

```python
##    @package main
#        ================================================
#            FEDERAL UNIVERSITY OF SANTA CATARINA
#        ================================================
#
#    File: ~/codigo/main.py
#    Created on 4 de abr de 2017
#    @author:  Rodrigo Pedro Marques
#    GitHub: https://github.com/rodrigo93/INE5429-Trabalho1
#    Professor: Renato Felipe Custodio
#
#    This file is part of a college project for the INE5429 Computer Security
#    course lectured in Federal University of Santa Catarina.
import time
from lcg import LCG
from blumblumshub import BBS
from millerrabin import Millerrabin
from fermat import Fermat

##
#    Funcao main que ira gerar os numeros primos
if __name__ == '__main__':
    tamanhos = [40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096] #Tamanhos dos numeros que serao gerados pelos geradores

    lcg = LCG(74573)            #semente do LCG
    bbs = BBS(88667)            #semente do BBS
    miller = Millerrabin(50)    #50 iteracoes
    fermat = Fermat(50)         #50 iteracoes

    outFile = open("numeroPrimos.txt", "wb")

    #Testando LCG com Miller
    print "Gerando possiveis numeros primos com LCG e Miller.\n"
    start_lcg_miller_time = time.time()
    outFile.write("Possiveis numeros primos gerados por LCG e Miller: \n")
    for m in tamanhos:
        while True:
            numeroPrimo = lcg.gerador(m, 1103515245, 12345)
            if miller.teste(numeroPrimo):
                outFile.write(str(numeroPrimo) + "\n")
                print "Um possivel numero primo de tamanho ", m, " bits foi gerado em ", (time.time() - start_lcg_miller_time), " segundos."
                break

    end_lcg_miller_time = time.time() - start_lcg_miller_time
    tempoDeExecucao = "--- Tempo de execucao: %s segundos. --- \n" % end_lcg_miller_time
    outFile.write(tempoDeExecucao)

        #Testando LCG com Fermat
    start_lcg_fermat_time = time.time()
```

```
    outFile.write("Possiveis numeros primos gerados por LCG e Fermat: \n")
    for m in tamanhos:
        while True:
            numeroPrimo = lcg.gerador(m, 1103515245, 12345)
            if fermat.teste(numeroPrimo):
                outFile.write(str(numeroPrimo) + "\n")
                print "Um possivel numero primo de tamanho ", m, " bits foi gerado em ", (time.time() - start_lcg_fermat_time), " segundos."
                break

    end_lcg_fermat_time = time.time() - start_lcg_fermat_time
    tempoDeExecucao = "--- Tempo de execucao: %s segundos. --- \n" % end_lcg_fermat_time
    outFile.write(tempoDeExecucao)

    #Testando BBS com Miller
    start_bbs_miller_time = time.time()
    outFile.write("Possiveis numeros primos gerados por BBS e Miller: \n")
    for m in tamanhos:
        while True:
            numeroPrimo = bbs.gerador(m)
            if miller.teste(numeroPrimo):
                outFile.write(str(numeroPrimo) + "\n")
                print "Um possivel numero primo de tamanho ", m, " bits foi gerado em ", (time.time() - start_bbs_miller_time), " segundos."
                break

    end_bbs_miller_time = time.time() - start_bbs_miller_time
    tempoDeExecucao = "--- Tempo de execucao: %s segundos. --- \n" % end_bbs_miller_time
    outFile.write(tempoDeExecucao)


    #Testando BBS com Fermat
    start_bbs_fermat_time = time.time()
    outFile.write("Possiveis numeros primos gerados por BBS e Fermat: \n")
    for m in tamanhos:
        while True:
            numeroPrimo = bbs.gerador(m)
            if fermat.teste(numeroPrimo):
                outFile.write(str(numeroPrimo) + "\n")
                print "Um possivel numero primo de tamanho ", m, " bits foi gerado em ", (time.time() - start_bbs_fermat_time), " segundos."
                break

    end_bbs_fermat_time = time.time() - start_bbs_fermat_time
    tempoDeExecucao = "--- Tempo de execucao: %s segundos. --- \n" % end_bbs_fermat_time
    outFile.write(tempoDeExecucao)

    outFile.close()

    print "PROGRAMA FINALIZADO COM EXITO"
```

```
##  @package lcg
#        ================================================
#            FEDERAL UNIVERSITY OF SANTA CATARINA
#        ================================================
#
#    File: ~/codigo/lcg.py
#    Created on 4 de abr de 2017
#    @author:  Rodrigo Pedro Marques
#    GitHub: https://github.com/rodrigo93/INE5429-Trabalho1
#    Professor: Renato Felipe Custodio
#
#    This file is part of a college project for the INE5429 Computer Security
#    course lectured in Federal University of Santa Catarina.
import time


##
#   Linear Congruential Generator e um algoritmo gerador de numeros pseudo-aleatorios.
class LCG(object):

    ##
    #   Construtor da classe.
    #   @param self ponteiro do objeto.
    #   @param semente valor inicial atribuido a semente do algoritmo
    def __init__(self, semente):
        self.semente = semente
        return

    ##
    #   Formula do algoritmo LCG para gerar os numeros pseudo-aleatorios.
    #   @param self ponteiro do objeto.
    #   @param m valor do modulo que ira limitar o tamanho do numero gerado.
    #   @param a multiplicador.
    #   @param c incrementador.
    #   @return valor pseudo-aleatorio gerado.
    def gerador(self, m, a, c):
        self.semente = (a*self.semente + c) % (2**m)
        return self.semente

    ##
    #   Metodo utilizado para testar o algoritmo LCG.
    #   Como exemplo, utilizei os valores:
    #   semente = 74573,
    #   m = aos tamanhos especificados no enunciado
    #   a = 1103515245
    #   c = 12345
    def teste(self):
        outFile = open("lgc_output.txt", "wb")

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

```
        tabelaDeResultado = []
        indice = 0;
        for m in tamanhos:
            indice += 1
            tabelaDeResultado.append(self.gerador(m, 1103515245, 12345))
            print "Para o tamanho m = ", m, " gerou-se o numero ",
tabelaDeResultado[indice-1]
            outFile.write(str(tabelaDeResultado[indice-1]) + "\n")

        outFile.close()
        return
```

```python
##  @package blumblumshub
#        ================================================
#            FEDERAL UNIVERSITY OF SANTA CATARINA
#        ================================================
#
#    File: ~/codigo/blumblumshub.py
#    Created on 4 de abr de 2017
#    @author:  Rodrigo Pedro Marques
#    GitHub: https://github.com/rodrigo93/INE5429-Trabalho1
#    Professor: Renato Felipe Custodio
#
#    This file is part of a college project for the INE5429 Computer Security
#    course lectured in Federal University of Santa Catarina.
import time

##
#   Blum Blum Shub e um algoritmo gerador de numeros pseudo-aleatorios.
class BBS(object):


    ##
    #   Construtor da classe.
    #   @param self ponteiro para o objeto.
    #   @param semente valor que sera atribuido a semente do algoritmo.
    def __init__(self, semente):
        self.seed = semente
        return

    ##
    #   Formula do algoritmo BBS para gerar os numeros pseudo-aleatorios.
    #   @param self ponteiro do objeto
    #   @param m valor do modulo que ira limitar o tamanho do numero gerado
    #   @return numero pseudo-aleatorio gerado
    def gerador(self, m):
        self.seed = (self.seed**2) % (2**m)
        return self.seed

    ##
    #   Metodo utilizado para testar o algoritmo Blum Blum Shub.
    #   @param self ponteiro para o objeto
    def teste(self):
        outFile = open("bbs_output.txt", "wb")

        tamanhos = [40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096]
        tabelaDeResultado = []
        indice = 0;
        for m in tamanhos:
            indice += 1
            tabelaDeResultado.append(self.gerador(m))
            print "Para o tamanho m = ", m, " gerou-se o numero ", tabelaDeResultado[indice-1]
            outFile.write(str(tabelaDeResultado[indice-1]) + "\n")
```

```
    outFile.close()
    return
```

```python
##  @package millerrabin
#        =============================================
#            FEDERAL UNIVERSITY OF SANTA CATARINA
#        =============================================
#
#    File: ~/codigo/millerrabin.py
#    Created on 4 de abr de 2017
#    @author:  Rodrigo Pedro Marques
#    GitHub: https://github.com/rodrigo93/INE5429-Trabalho1
#    Professor: Renato Felipe Custodio
#
#    This file is part of a college project for the INE5429 Computer Security
#    course lectured in Federal University of Santa Catarina.
import time
import random

##
#    Miller Rabin e um algoritmo para realizar a verificacao da primalidade de um dado numero.
class Millerrabin(object):


    ##
    #    Construtor da classe.
    #    @param self ponteiro para o objeto
    #    @param it iteracoes que o algoritmo ira realizar
    def __init__(self, it):
        self.iteracoes = it
        return

    ##
    #   Testa a base 'a' para verificar se 'a' eh um candidato para a composicao de 'numero'
    #   @param self ponteiro para o objeto
    #   @param a numero aleatorio dentro do intervalor 1 <= a <= n-1
    #   @param d valor de d
    #   @param s valor de s
    #   @param numero numero que ira tentar ser decomposto
    def decompoe(self, a, d, s, numero):
        if pow(a, d, numero) == 1:
            return False
        for i in range(s):
            if pow(a, 2**i * d, numero) == numero-1:
                return False
        return True

    ##
    #   Metodo utilizado para testar a primalidade de um numero.
    #   @param self ponteiro para o objeto.
    #   @param numero numero que tera sua primalidade testada.
    def teste(self, numero):
        # Verifica se o numero e par
```

```
        if numero % 2 == 0:
            if numero == 2:
                return True
            return False

        # caso num nao seja primo
        # descoberta dos valores de s e d
        s = 0
        d = numero-1
        while True:
            quociente, resto = divmod(d, 2)
            if resto == 1:
                break
            s += 1
            d = quociente
        assert(2**s * d == numero-1)

        for it in range(self.iteracoes):
            a = random.randrange(2, numero)
            if self.decompoe(a, d, s, numero):
                return False

        return True
        return
```

```python
## @package fermat
#        ================================================
#            FEDERAL UNIVERSITY OF SANTA CATARINA
#        ================================================
#
#    File: ~/codigo/fermat.py
#    Created on 4 de abr de 2017
#    @author:  Rodrigo Pedro Marques
#    GitHub: https://github.com/rodrigo93/INE5429-Trabalho1
#    Professor: Renato Felipe Custodio
#
#    This file is part of a college project for the INE5429 Computer Security
#    course lectured in Federal University of Santa Catarina.
import random

##
#    Fermat e um algoritmo para realizar a verificacao da primalidade de um dado numero.
#
class Fermat(object):

    ##
    #    Construtor da classe.
    #    @param self ponteiro do objeto
    #    @param it numero de iteracoes que serao realizadas
    def __init__(self, it):
        self.iteracoes = it
        return

    ##
    #   Formula do algoritmo BBS para gerar os numeros pseudo-aleatorios.
    #
    def teste(self, numero):
        if numero == 2:
            return True

        if numero % 2 == 0:
            return False

        for it in xrange(self.iteracoes):
            a = random.randint(1, numero-1)
            return pow(a, numero-1, numero) == 1

```