Trabalho Individual sobre Números Primos

Fernando Paladini, Segurança em Computação (INE5429) 23/08/2016

Introdução

Devido à necessidade de geração de números aleatórias para diversas áreas da computação e à nossa incapacidade de gerar números verdadeiramente aleatórios em computadores determinísticos, criaram-se muitas técnicas algorítimicas para geração de números pseudo-aleatórios (PRNG - Pseudo-random number generator). Esses algoritmos evoluíram tanto ao longo do tempo que hoje podemos gerar números pseudo-aleatórios que tem uma qualidade muito similar aos números verdadeiramente aleatórios gerados com auxílio da entropia do meio.

Com a finalidade de conhecer e entender o funcionamento de alguns desses algoritmos, esse trabalho se faz presente. Para ele, a linguagem de programação Python foi escolhida devido ao seu grande poder de expressão, alta legibilidade, facilidade de uso e também ao fato interessante de trabalhar com números de precisão arbitrária, o que fornece muita flexibilidade ao programador.

1) Gerar números pseudo-aleatórios

Os dois algoritmos escolhidos para geração de números pseudo-aleatórios foram: LCG (Linear Congruential Generator) e BBS (Blum Blum Shub). A escolha do **Linear Congruential Generator** se deu pelos seguintes motivos:

- Usado em muitas bibliotecas e compiladores, tais como glibc, C99, C11, Turbo Pascal, Borland C/C++, Microsoft Visual Basic, etc.
- Recomendado para sistemas embarcados.
- Um dos algoritmos mais clássicos e mais conhecidos para PRNG.
- Fácil compreensão.
- Velocidade elevada e baixo consumo de memória.

A escolha do **Blum Blum Shub** se deu pelos seguintes motivos:

- Existe uma prova (controversa) que prova sua segurança.
- Fácil compreensão.
- Implementação elegante.
- Nome peculiar.

A complexidade do algoritmo LCG é extremamente baixa, pois não depende de nada além de um cálculo para obter uma resposta. Assim, podemos afirmar que a complexidade de LCG é O(1), mas há um agravante muito específico criado pelo fato de o número gerado possuir o número de bits desejado pessoa, o que faz com que a existência de um laço de repetição seja necessária. A complexidade do BBS também é variável de acordo com a quantidade de bits desejada, sendo, de maneira bem genérica, O(1).
Abaixo é possível ver diversos números pseudo-aleatórios gerados pelo algoritmo LCG (Linear Congruential Generator):

40 bits:
1084561151355 (0m0.008s)
654045254620 (0m0.012s)
1080517984963 (0m0.009s)
1058554509804 (0m0.020s)

56 bits:
65037325448226801 (0m0.015s)
54305578582889695 (0m0.012s)
54324973087219070 (0m0.016s)
54341596948072820 (0m0.013s)

80 bits:
825478553948280999142574 (0m0.012s)
823657620336969256720675 (0m0.010s)
741754578878573068025056 (0m0.012s)
883462854729154844725206 (0m0.008s)

128 bits:
268853856020913286940901910622370278949 (0m0.008s)
245119205298605232443540886800341667921 (0m0.020s)

168 bits:
246618470779997295880953988703001426232298520142150 (0m0.028s)
277140253240266825630062297621100181156036692087498 (0m0.016s)

224 bits:
13755011397324812115354082314941958941498927444247465331799606790003 (0m0.008s)
13731769903020160639743065215475186406147341550513132410782221346592 (0m0.012s)

256 bits:
75934965606474312794107659343314611582390986107840384428494084559422279448

656 (0m0.020s)

512 bits:
6708081316904432149589009221475343374137366139605697590093702902702289181675254180018049296440144460803354073244657365732248731248019343976199148101
119158 (0m0.008s)

1024 bits:
14869026400888196802679431423197642419989611218196268730818338113634592965014049273831277518792690492431749626153796149689548097286434895822614632603452134816660189033513823271149722119994609645779408071548391107540303021571354096661519485053981835671173945694918831660400573440069809300541662885
8798924051303 (0m0.008s)

2048 bits:
20761933335176962018661925159721784773668150332034900388175899345935649908
46290215972680991380161784817136721606149706594451670491724372896246414173
06952700256995512912730234410006463725506923319063127612911416120010167502
08750692441372185323615992393573325950294736159343116280858602193508629271
30082059864871733913476842543936636643916898566330848272180687385038741269
30601485180780941662125753841570749713128475904506974772850572958882788935
86398101090481632300278733543460637370170073952290507161396186258692719603
46513961864381094418846852935902257651832277806685329409874409459675617944
0675217933541737886437964 (0m0.020s)

4096 bits:
77664718511829621341309675674439283950086935498468088382362265895197180678
57458162289285085285274112320182583157502411574925001117950066739688382901
29232506476863869841365438698892679312626636723254689403202717726925631041
64739664776962161499573775175702493889862525742216824123236857036599531828
04694414474293074724228148121272334721344098643163252521794744390327795182
92877797576492582090940140436587077449623348296569173416761371393524010115
89405301927825702194723519285583190547045447838262326389289708477429164641
76592167472948591735160107776632545159169585110804679162232917681917172798
35283122010724463391830294351658699420968473546183160288731991956493742652
95689484545122939973086514000137738119929885319742608932326202743658646144
19095427534676037299340238889475741089302139835445723023608937617520459564
00640423872448530095011206883943292499825699347080391997432376255518312620
38472603039115949671126033735878033739643459078488234177706151880573189402
34185365703775698526373070369996257562423740145541867409256100020713912279
17128938186234474890590016770884086948801410235003075260153936795544028263

466222018721146918322249342498436981582585260081322888770286600121389056169082050643750980741760094212338419271992329688298 (0m0.012s)

Abaixo é possível ver diversos números pseudo-aleatórios gerados pelo algoritmo BBS (Blum Blum Shub):

40 bits:
881150597781 (0m0.016s)
695308320892 (0m0.020s)
101137878464 (0m0.044s)
777879788419 (0m0.020s)

56 bits:
29170280903880792 (0m0.016s)
58720041018058963 (0m0.020s)
18850276677706401 (0m0.024s)
1822408514450231 (0m0.016s)

80 bits:
898823120399127663989487 (0m0.020s)
767331397759652107947207 (0m0.024s)
60959099700994804801500 (0m0.032s)
434174257068244438676183 (0m0.028s)

128 bits:
12002746307558928577133413317202998450 (0m0.064s)
119841567296394090473295001181692520687 (0m0.044s)

168 bits:
120090224487931614653387891896756808363453866142009 (0m0.060s)
201176655275174844338948044499909035039311621705284 (0m0.080s)

224 bits:
11195511721587131003114485058722927740020457087309531472989241399807 (0m0.100s)
13625850567232881799057368302204199730304398036338223025712609887503 (0m0.148s)

256 bits:
97757971878987503400941450006837324247693788830712525723281635255687359441896 (0m0.164s)

512 bits:
884071943027848293037745087454980490271135701796763504253664980634870582572576054352891332155439267936514962307310365314449424218856358802739094483
1772194 (0m2.632s)

1024 bits:
119998270273247773446130448725542557927650494207820163528955492871080796960773522231016892974274996238781634908420437039680170123349655839923472254803781476788334970217141505982483219650112873950441423492132239660181470492411923776895696620968100315864596043215646719732419826052509847917569710490998329050580 (0m2.832s)

2048 bits:
150561825534468995248970252874945895869335538252264386973001507078126901182116544386501692864888827081280627580315992831220939503288366767576102622374492878360061977209268606280628651375677868330838041238444145590562859086011579770618290594284082963333106836658272363613541980879367623099540777956310116608292996697982249165656557668012193752025211674613540349762893181922771912037327259994702443539700962178806532899265978884399084907217550840845465259543614264696073164495907191854388145687329321268511187611279447058107071835563994875526799268955740167617130733683622937873199694978920114693461352590045258756072 71 (1m0.716s)

4096 bits:
12070648260053143028154460404325515572753803549960811033401607170303684900018294507904131168290282156538996749605408346444592970910203306522994816389948166319990752803014874242041960712270710723890834951268430266480531910849555176870273601404582354628985731686802945771823687783873369552254581729346839081133698743903830309757101042608189368800972205719315467751484533139524053087276980643012779826785181451141456737161155923271183744628086520232246081565517026881888163628023679768064048404327253949930571677657312201371436994004294055173400030577115094069834130033932968274646023190296077371974680975147006383161275918509018434150456468619793464981062030265073769007071426307515374000685316487948076859438900780632711937807366517999381925156722651629590833113563975362679146092386328683650698075851279550453882115758263829935488951587685222038972364181073168206646222827455485874429191917319379302842401537213337909568782114526923563411803978047395471028780465354168378056233967324069850867484311243720562966378936364154890973327560358805365158684684669907964720592584213307706004307396636879818980589326732367807057150268137450536354740966819178459554694373141926094740822231347437

1059055317620904963430413621481360206627847666620 (16m32.380s)

A implementação dos algoritmos escolhidos (LCG e BBS) devidamente comentada pode ser encontrada logo abaixo. Observação: não foi possível renderizar os caracteres do código/documentação em UTF-8, por isso eles foram substituídos por caracteres sem acentos e similares.

Listing 1: Implementação do algoritmo LCG - Arquivo lcg.py.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import sys
import time

"""

Esta classe gera numeros pseudo-aleatorios utilizando o algoritmo LCG (Linear Congruential Generator).
Para chamar esta classe a partir da linha de comando basta digitar:

    $ python lcg.py <qtd_de_bits>

Onde "<qtd_de_bits>" eh a quantidade de bits que o numero gerado deve possuir. Exemplo:

    $ python lcg.py 32
    [LCG] Gerando numero de 32 bits...
    [LCG] N mero: 4174021489

Referencias:
    https://en.wikipedia.org/wiki/Linear_congruential_generator
    https://en.wikipedia.org/wiki/Combined_Linear_Congruential_Generator
    http://www.eternallyconfuzzled.com/tuts/algorithms/jsw_tut_rand.aspx
    https://rosettacode.org/wiki/Linear_congruential_generator

"""
class lcg(object):

    def __init__(self, seed = int(time.time()), m = 2**32, a = 1664525, c = 1013904223, size = None):
        """
            O construtor da classe do gerador de numeros pseudo-aleatorios eh altamente customizavel e
```

```
            recebe alguns parametros com valores padr o baseados no livro "Numerical Recipes: The Art
            of Scientific Computing" (Press, WH; Teukolsky, SA; Vetterling, WT; Flannery, BP).

            Por padrao vai gerar um n mero de at  32 bits, mas caso um "size" seja fornecido, vai gerar
            somente numeros com "size" bits.

            Args:
                seed: valor de semente para iniciar o gerador de n meros pseudo-aleatorios. Se nao
                      informado, utiliza o Unix Timestamp (Epoch) de acordo com as informacoes do sistema.
                m: o modulo, cujo valor padrao eh 2^32 / 4294967296.
                a: o multiplicador, cujo valor padrao eh 1664525.
                c: o incremento, cujo valor eh 1013904223.
                size: o tamanho em bits do n mero a ser gerado, cujo valor padrao eh None (na pratica eh 32).

            Returns:
                Esse metodo  n o retorna nada.

        """
        self.a = a
        self.c = c
        self.seed = seed
        self.size = size
        if self.size:
            self.m = 2**size
        else:
            self.m = m

    def rand(self):
        """
        Gera um numero aleatorio utilizando o algoritmo LCG, que eh descrito pela relacao
        de recorrencia expressa a seguir:

            Xn+1 = (a * Xn + c) mod m

        Onde:
            m: o modulo (0 < m)
            a: o multiplicador (0 < a < m)
            c: o incremento (0 <= c < m)
            X: sequencia de valores pseudo-aleatorios.
            X0: o "seed" ou valor de comeco.
```

```
            Xn+1: o proximo numero a ser gerado.

        Se uma instancia dessa classe possuir o atributo "size" definido no momento da construcao do objeto
        ou definido posteriormente em uma chamada ao metodo seed(self, new_seed), entao o numero gerado de
        forma pseudo-aleatoria possuira "size" bits (ficara dentro de um loop enquanto nao atingir essa
        quantidade de bits estipulada). Caso a instancia n o possua o atributo "size" definido, entao o
        numero pseudo-aleatorio gerado possuira ate 32 bits.

        Args:
            Este metodo nao recebe nenhum argumento.

        Returns:
            Um valor numerico pseudo-aleatorio de "size" bits.

        """
        self.seed = self.seed * self.a + self.c
        num = self.seed % self.m
        if self.size:
            while (num.bit_length() < self.size):
                self.seed = self.seed * self.a + self.c
                num = self.seed % self.m
        return num

    def randint(self, a, b):
        """
        Gera um numero aleatorio que esta entre os intervalos "a" e "b".

        O numero a ser gerado, denominado "num", sera maior ou igual a "a" e menor ou igual
        a "b". Em outras palavras, a <= num <= b.

        Args:
            a: valor numerico de limite inferior para o numero a ser gerado.
            b: valor numerico de limite superior para o numero a ser gerado.

        Returns:
            Um valor numerico pseudo-aleatorio que esta entre os valores a e b.

        """
        self.seed = self.seed * self.a + self.c
```

```
        num = self.seed % self.m
        while (not (a <= num <= b)):
            self.seed = self.seed * self.a + self.c
            num = self.seed % self.m
        return num

    def seed(self, new_seed):
        """
        Metodo para mudar o valor do seed para algum valor desejado.

        Args:
            new_seed: um valor numerico para indicar o novo valor de seed.

        Returns:
            Esse metodo nao retorna nada.

        """
        self.seed = new_seed

    def size(self, new_size):
        """
        Metodo para mudar a quantidade de bits que o numero gerado tera.

        Args:
            new_size: um valor numerico para indicar a nova quantidade de bits do numero gerado.

        Returns:
            Esse metodo nao retorna nada.

        """
        if self.size != new_size:
            self.size = new_size
            self.m = 2**new_size


if (__name__ == "__main__"):

    bits = int(sys.argv[1])

    print("[LCG] Gerando numero de {} bits...".format(bits))
    print("[LCG] Numero: {}".format(lcg(size=bits).rand()))
```

Listing 2: Implementação do algoritmo BBS - Arquivo blum_blum_shub.py.

```
#!/usr/bin/env python3
```

```python
# -*- coding: utf-8 -*-

import sys
import time
from lcg import lcg
from primality import MillerRabin

"""

Esta classe gera numeros pseudo-aleatorios utilizando o algoritmo BBS (Blum Blum Shub).
Para chamar esta classe a partir da linha de comando basta digitar:

    $ python blum_blum_shub.py <qtd_de_bits>

Onde "<qtd_de_bits>" eh a quantidade de bits que o numero gerado deve possuir. Exemplo:

    $ python blum_blum_shub.py 32
    [Blum Blum Shub] Gerando n mero de 32 bits...
    [Blum Blum Shub] N mero: 3234751506

Referencias:
    https://en.wikipedia.org/wiki/Blum_Blum_Shub
    https://pt.wikipedia.org/wiki/Blum_Blum_Shub
    https://crypto.stackexchange.com/questions/3454/blum-blum-shub-vs-aes-ctr-or-other-csprngs
    https://jeremykun.com/2016/07/11/the-blum-blum-shub-pseudorandom-generator/
    http://cs.ucsb.edu/~koc/cren/project/pp/gawande-mundle.pdf

"""
class BlumBlumShub(object):

    def __init__(self, seed = None, size = None):
        """
        Construtor da classe do gerador de numeros pseudo-aleatorios.

        Por padrao vai gerar um numero pseudo-aleatorio de ate 32 bits, mas caso um "size" seja fornecido,
        vai gerar somente numeros com "size" bits.

        Args:
            seed: valor de semente para iniciar o gerador de n meros pseudo-aleatorios. Se nao
                informado, utiliza um valor numerico pseudo-aleatorio entre 2 e (m - 1).
```

```
            size: o tamanho em bits do numero a ser gerado, cujo valor padrao eh None (na pratica eh 32).

        Returns:
            Esse metodo nao retorna nada.

        """
        self.size = size
        self.m = MillerRabin.encontrar_primo(self.size) * MillerRabin.encontrar_primo(self.size)
        if (seed):
            self.state = seed % self.m
        else:
            self.state = lcg(size=self.size).randint(2, self.m - 1) % self.m

    def rand(self):
        """
        Gera um n umero aleatorio utilizando o algoritmo BBS, que eh descrito da seguinte forma:

            Xn+1 = (Xn)   mod M

        Onde:
            M: eh o produto de dois numeros primos muito grandes (comumente denominados p e q),
               ambos congruentes a 3 (mod 4) e com mdc (maximo divisor comum) pequeno (fazendo
               o tamanho do ciclo ser grande).
            X0: o seed (X0) precisa ser um inteiro co-primo a M e nao pode ser 1 ou 0.
            Xn+1: o proximo n mero a ser gerado.

        Args:
            Este metodo n o recebe nenhum argumento.

        Returns:
            Um valor numerico pseudo-aleatorio de "size" bits. No BBS a saida costuma ser o bit de paridade
            ou um ou mais dos bits menos significantes (vide metodo bitstram(self)).

        """
        output_bits = ''
        for bit in self.bitstream():
            output_bits += str(bit)
            if (len(output_bits) == self.size):
```

```
                break

        return int(output_bits, 2)

    def seed(self, new_seed):
        """
        Metodo para mudar o valor do seed para algum valor desejado.

        Args:
            new_seed: um valor numerico para indicar o novo valor de seed.

        Returns:
            Esse metodo n o retorna nada.

        """
        self.state = new_seed

    def size(self, new_size):
        """
        Meodo para mudar a quantidade de bits que o numero gerado tera.

        Args:
            new_size: um valor numerico para indicar a nova quantidade de bits do numero gerado.

        Returns:
            Esse metodo nao retorna nada.

        """
        if self.size != new_size:
            self.size = new_size
            self.m = 2**new_size

    def bitstream(self):
        """
            Metodo auxiliar (e necessario) para o calculo do n mero pseudo-aleatorio utilizando o algoritmo BBS.

            Returns:
                Bit que vai compor um conjunto de bits pseudo-aleatorios que depois serao convertidos
                para um valor numerico pseudo-aleatorio.
        """
        while (True):
            yield (sum(int(x) for x in bin(self.state)[2:]) % 2)
            self.state = pow(self.state, 2, self.m)

```

```
if (__name__ == "__main__"):

    bits = int(sys.argv[1])

    print("[Blum Blum Shub] Gerando numero de {} bits...".format(bits))
    print("[Blum Blum Shub] Numero: {}".format(BlumBlumShub(size=bits).rand()))
```

2) Verificação de primalidade

Os números primos são essenciais para muitas aplicações da computação, com a mais conhecida sendo a criptografia, em particular a criptografia assimétrica. Chaves RSA são geradas a partir de números primos com uma grande quantidade de bits, de forma que sejam suficientemente seguras - se tratando de segurança, isso significa que precisam ser praticamente inquebráveis. Para que isso seja possível, testes simples e eficientes devem existir para determinar a primalidade de números. Entre os testes probabilísticos mais conhecidos estão o teste de primalidade de Fermat e o teste de Miller-Rabin. Devido à grande importância do teste de Fermat para iniciar os estudos na área de primalidade (e também devido a base fornecida para o teste de Miller-Rabin), resolvi escolher ele para tratar aqui.

O teste de primalidade de Fermat, também conhecido como o "Pequeno Teorema de Fermat", afirma que se p é primo, $0 < a < p$, então:

$$a^{p-1} \equiv 1 \pmod{p} \tag{1}$$

Para testar se p é primo, basta escolher inteiros a aleatórios no intervalo possível e verificar se a congruência (expressa acima) é válida. Se isso for verdade pra muitos valores, então p é muito possivelmente um primo. Entretanto, se um inteiro a gera uma incongruência da forma

$$a^{p-1} \not\equiv 1 \pmod{p} \tag{2}$$

, então podemos dizer que é a é uma testemunha de que p é composto - e portanto não é primo.

Embora a complexidade do teste de Miller-Rabin seja maior do que do teste de primalidade de Fermat (portanto, possui um tempo de execução maior), ele possui maior precisão independente do número testado, de forma que acaba garantindo com um nível maior de segurança que um número é primo ou composto.

Abaixo é possível ver diversos números primos gerados através do algoritmo de LCG e verificados pelo teste de primalidade de Fermat e teste de Miller-Rabin:

40 bits:
Fermat: 951159012547 (0m0.008s)

Miller-Rabin: 813765973871 (0m0.012s)

56 bits:
Fermat: 56914756873663081 (0m0.004s)
Miller-Rabin: 56753366274499183 (0m0.024s)

80 bits:
Fermat: 929945705236430685794213 (0m0.012s)
Miller-Rabin: 721584187574854472469109 (0m0.012s)

128 bits:
Fermat: 251144094554926686005408286496966864861 (0m0.020s)
Miller-Rabin: 310221192693605632789209171515817452831 (0m0.036s)

168 bits:
Fermat: 349050850241449722612777471186888866498433331556159 (0m0.024s)
Miller-Rabin: 330574843746154037512847946249138708551260727604581 (0m0.032s)

224 bits:
Fermat: 24701610941290092073600617507245648431874818097677454260412190079909 (0m0.036s)
Miller-Rabin: 141533904041570637570744591556392749569205098313360066498088998 54069 (0m0.092s)

256 bits:
Fermat: 849644049206950665508436868262423021964619872673597959971214908555500 25244891 (0m0.040s)
Miller-Rabin: 790023474772544328750046637268678788420949100196251629805700653 94251951770463 (0m0.080s)

512 bits:
Fermat: 101312636165095236250600730742464447640465739011727812697632208917133 146207310008255839066036243221879501091150669324651427472908035060795008 21445707485127 (0m0.820s)
Miller-Rabin: 790516987879378390965020343239431590735078943302778706255972564017 411869546087781272933280567129316630119678980536078064401484488073668551 8570879839799773 (0m0.892s)

1024 bits:
Fermat: 145705102039952137418475509314546364205101802258842912632043631197757 443223217931618143386179881147902812502187004137952225214403992109750444 1

45857588554486059685575566978006219428522471607641661426188275627255455626295676067440108431777390742137180421631506271170715986707249831468443712285645717067352039269 (0m1.192s)
Miller-Rabin: 153203101678922435973792299558747852522965891157025105191264941926310598044277342287968814352436198623535696078822904876826364789958221829656625510785685305180700799552606876279669908115331545874647708231733832302475412744252216927406928221056623611299201310211367575164379562291293045482394238774509212512431 (0m2.780s)

2048 bits:
Fermat: 22938966653774785416754486611272032381511122765045352327253061776694001762793835777876105880323163733280461323575188699600540824242639941507484167068162921997812607061335178463406801896282950761332095588078303255269139549873938484569156476205250620648142895897423703521355109896207812475300366054760813063580638398062029345556633489349922647306748309637175016723811998151702095423802642753452354237575281706266550533321561013303856091406434428632537018568223716907330920291587084836207800047840770203450769454798244954529983789859684769634622300033899229981441481483751849185263586554055488572641435747321172351664737 (0m33.384s)
Miller-Rabin: 20284410440733597505836134517744494592748921392227084888355694426593706895306799866751471778523475573632853526505644244521279608377088227492969699793373121374816436760443643478092552322679081735076496388189919862611685226391230976410822702023023634251824647715268906605354689376387986942646048516083122415688826811234103618953741453802750949396108413856378664166420252310028603610617778226620410743614758195421928317274394297274547023025699503361502747385979256180051289356791342414877016440258756335259525400208886709908809958478188126501107971413084041303321948946158806736777577505665678564056312622473620637868789 (4m55.212s)

4096 bits:
Fermat: 9903739818881791501961910026282606980214816630237494692359811636754699244599761866530509605126300888949313510407599256393494974304483130812345611512840176640755840724831758687944866259679971748986660325877929777426129601902279786745205911832572497758599635386344718764543842269096487990266285337208422491936085867948857604309477037835713303297637902904930241141313414211843234757151450878330210648749197408987937249922352748533058503610412056881351605537351097781646973237911050559136157449753781321505867851174836735883813755280141025916671167893141597153705748801036522619221423734924844944513886721407888138884528530231413291710349500908394357010350477624856645173191149932590627312068177444990177014550913278868038529322168024264506142558806881447541698470882784686636403635981434519033338498143946781763

0329893812174804718448791632037384459130895836041138204547671521710263773093052734006745520222701961441256486964233070174564545863912854490558855995235297659916701614210064628662147334215828646482043139818308701691010199273958452136085686928918375287677002822368228529013477143216082350163047196212591284074143611288869081107924769376789191690284407992802625275857272733926329933544402450217592333716656506894225017811860649 1 (0m25.808s)
Miller-Rabin: 8245257712039196254626962383255614338593538271638698838890457585928355100646418642765599499053621136633185653813311522281062585877729698172556303323796162723705979408822549407634301818335383689475414765195505055715846610608627101845070788431042390795160200439234631494212153904353966448952400096985615578848129713991508268570886346115050635690323096383448949398620382077038849550250158645819426498724932227329109918421519633832481225429949376391384401350132955848467742241029170595399166012324097776478174614328192540503564558486352868709904484430235293720380133146108028275382972156491749545159581105964626636177158514102571095399926321346934630153511677968028941819340632220221864531333734982130652326300536996740948424180000184018023061543636086441112953953328856360778473607283022309893809964426275553159792207650210964816941757380754590111332975416658885917778162247364862292352980053840274048274744864578674437869150843340960311176366051598300961249422065940595921196652840514579825874605998073873479547137940937303479909752538210357062321723116057797329331191181150546099289334065241844130693929108623751610123651490420349144506897223724515326677322473039766193110116826691855192004294739082881384701834057728920109738297789 61 (2m52.932s)

É notável a diferença de tempo de execução entre os testes de primalidade de Fermat e Miller-Rabin, de forma que tive bastante dificuldade para gerar números primos com este último. Ainda assim, o gerador de número pseudo-aleatórios utilizado foi o LCG, que é extremamente mais rápido do que o BBS. Realizei alguns testes e pude notar que é praticamente inviável a geração de números primos no meu computador utilizando o teste de Miller-Rabin com a geração de números pseudo-aleatórios através de BBS (Blum Blum Shub).

A implementação dos verificadores de primalidade devidamente comentados podem ser encontrados logo abaixo. Observação: não foi possível renderizar os caracteres do código/documentação em UTF-8, por isso eles foram substituídos por caracteres sem acentos e similares.

Listing 3: Implementação dos algoritmos de teste de primalidade de Fermat e Miller-Rabin - Arquivo primality.py

```
#!/usr/bin/env python3
```

```python
# -*- coding: utf-8 -*-

import sys
import time
from lcg import lcg

"""

As classes deste arquivo verificam se os numeros fornecidos sao primos e geram numeros primos de tamanho variavel de bits de forma pseudo-aleatoria.

Para executar esse arquivo a partir da linha de comando basta digitar:

    $ python primality.py <qtd_de_bits>

Onde "<qtd_de_bits>" eh a quantidade de bits que o numero gerado deve possuir. Exemplo:

    $ python primality.py 128
    [MillerRabin] Procurando primo...
    [MillerRabin] 304159568226184448912103696911866306307 eh primo!
    [Fermat] Procurando primo...
    [Fermat] 301970148924118150955226359108149582211 eh primo!

Referencias:
    https://en.wikipedia.org/wiki/Fermat%27s_little_theorem
    https://en.wikipedia.org/wiki/Pseudoprime
    https://pt.wikipedia.org/wiki/Teste_de_primalidade_de_Miller-Rabin
    https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test
    https://pt.wikipedia.org/wiki/Teste_de_primalidade_de_Fermat
    https://jeremykun.com/2013/06/16/miller-rabin-primality-test/
    http://mathworld.wolfram.com/Rabin-MillerStrongPseudoprimeTest.html
    https://www.youtube.com/watch?v=qfgYfyyBRcY

"""
class MillerRabin(object):

    @staticmethod
    def verificar_testemunha(possivel_testemunha, p, exp, resto):
        """
        Verifica se a possivel testemunha de que um numero n o eh primo eh
        realmente testemunha. Se for, significa que o numero que esta
        sendo testado tem uma testemunha da sua nao primalidade, de forma
        que a hipotese de que o numero testado eh primo pode ser descartada.
```

```
        Args:
            possivel_testemunha: a possivel testemunha de que "p" nao eh primo.
                                  Sera testemunha caso a^d     1 (mod n) e
                                  a^((2^r)d)     -1 (mod n) para todo 0 <= r <= s - 1.
            p: o numero para o qual a primalidade esta sendo testada.
            exp, resto: numeros inteiros, onde 'resto' eh um numero impar.

        Returns:
            True se a possivel testemunha for uma testemunha de que "p" nao eh primo.
            False se a possivel testemunha nao for uma testemunha de verdade.

        """
        possivel_testemunha = pow(possivel_testemunha, resto, p)
        if ((possivel_testemunha == 1) or (possivel_testemunha == p - 1)):
            return False

        for _ in range(exp):
            possivel_testemunha = pow(possivel_testemunha, 2, p)
            if (possivel_testemunha == (p - 1)):
                return False

        return True

    @staticmethod
    def verificar_primalidade(p, certeza=100):
        """
        O teste de Miller-Rabin eh um important ssimo teste probabilistico da primitividade de um n mero dado.
        Se um numero passa nesse teste significa que ele tem uma probabilidade >= 75% de ser um numero primo,
        mas ate este numero ser provado como sendo um numero primo ele eh considerado apenas um "pseudoprimo".

        Ao aplicar o mesmo teste varias vezes, a margem de erro pode ser diminuida aleatoriamente, de forma
        que a margem de erro final seja consideravelmente baixa. Ele eh baseado no "Pequeno Teorema de Fermat",
        que consiste do "Teste de primalidade de Fermat".

        Args:
            p: o numero a ser testado.
            certeza: o grau de "certeza" de que este n mero seja de fato
```

```
                um n mero primo. Valor padrao eh 100, o que significa que o teste sera aplicado 100 vezes.

        Returns:
            True se o numero eh um (pseudo-)primo.
            False se o numero n o eh primo.

        """
        if (p == 2 or p == 3):
            return True
        elif (p < 2):
            return False

        resto = p - 1
        exp = 0
        while (resto % 2 == 0):
            resto = resto/2
            exp += 1

        for _ in range(certeza):
            possivel_testemunha = lcg().randint(2, p - 2)
            if (MillerRabin.verificar_testemunha(possivel_testemunha, p, exp, resto)):
                return False

        return True

    @staticmethod
    def encontrar_primo(bits=None):
        """
        Encontra um numero primo que possui 'bits' bits utilizando uma busca com numeros gerados de forma pseudo-aleatoria.

        Args:
            bits: quantidade de bits que o numero primo deve possuir. Se nenhum valor for informado, sera usado 32 bits.

        Returns:
            Um valor numerico com 'bits' bits e que eh primo.

        """
        bits = bits or 32
        random = lcg(size=bits)

        while (True):
            primo = random.rand()
            if (MillerRabin.verificar_primalidade(primo)):
```

```
                return primo

class FermatPrimality(object):

    @staticmethod
    def verificar_primalidade(p):
        """
        Verifica se o numero dado 'p' eh um numero primo atraves do m todo de Fermat, tambem conhecido
        como "Teste de Primalidade de Fermat". Este eh um dos metodos mais simples para verificar se um
        numero eh primo ou nao (e provavelmente um dos mais elegantes tambem). Neste metodo a composicao
        do numero dado eh verificada e os numeros que falham no teste nao sao primos.

        Esta implementa  o nao leva em consideracao os numeros de Carmichael, que sao infinitos e
        passam pelo teste, mas n o sao primos. Portanto, a partir deste teste podemos obter apenas
        numeros que sao considerados "pseudoprimos".

        Args:
            p: o numero que sera testado a primalidade.

        Returns:
            True significa que o numero eh pseudoprimo, ou seja, provavelmente eh primo (existem falso-positivos).
            False significa que o numero eh composto, ou seja, nao eh primo.

        """
        if (p == 2):
            return True
        if (not p & 1):
            return False

        if (pow(2, p-1, p) == 1):
            return True
        else:
            return False

    @staticmethod
    def encontrar_primo(bits=None):
        """
        Encontra um numero primo que possui 'bits' bits utilizando uma busca com numeros gerados de forma pseudo-aleatoria.
```

```

        Args:
            bits: quantidade de bits que o numero primo deve possuir. Se nenhum valor for informado, sera usado 32 bits.

        Returns:
            Um valor numerico com 'bits' bits e que eh primo.

        """
        bits = bits or 32
        random = lcg(size=bits)

        while (True):
            primo = random.rand()
            if (FermatPrimality.verificar_primalidade(primo)):
                return primo

if (__name__ == "__main__"):

    bits = int(sys.argv[1])

    print("[MillerRabin] Procurando primo...")
    print("[MillerRabin] {} eh primo!".format(MillerRabin.encontrar_primo(bits)))

    time.sleep(2)

    print("[Fermat] Procurando primo...")
    print("[Fermat] {} eh primo!".format(FermatPrimality.encontrar_primo(bits)))
```