Geração de números primos

Gustavo Zambonin*
Segurança em Computação (UFSC – INE5429)

- Diversos métodos para geração de números aleatórios estão disponíveis como alternativas à necessidade do usuário: por exemplo, um gerador pode focar em desempenho, enquanto outros podem gerar números com uma maior quantidade de bits a partir de processos matemáticos mais complexos. Um *gerador de números pseudoaleatórios* (PRNG, também chamado de gerador de bits determinístico) é um algoritmo que tem como função gerar sequências de números aproximadamente aleatórios, dependentes apenas de um pequeno conjunto de valores iniciais, chamados de semente.

  É relevante apontar que, embora tais algoritmos sejam em grande parte complexos no aspecto teórico, ainda são baseados em uma série de transformações lineares que podem ser relacionadas com a semente inicial, tornando a saída dos algoritmos previsível e insegura. Assim sendo, um gerador de números pseudoaleatórios *criptograficamente seguro* (CSPRNG) é muito mais recomendável para uso em aplicações sensíveis.

  O PRNG discutido é o *Mersenne Twister* (MT, [1]), o mais difundido e presente em várias linguagens de programação como o gerador padrão de números pseudoaleatórios. Seu nome é derivado do período[1] de $2^{19937}-1$, um primo de Mersenne[2]. Formalmente, o algoritmo é baseado numa relação de recorrência linear matricial sobre $\mathbb{F}_2$[3]. Definindo uma série x_i através de uma relação de recorrência simples, obtêm-se números na forma x_iA, onde A é uma matriz com elementos em $\mathbb{F}_2$, assim ''temperando'' os elementos de modo recorrente. Este passo pode ser facilmente revertido através de uma transformação linear, e o padrão dos números gerados revelado com suficiente observação, assim fundamentando o fato de que este gerador não é criptograficamente seguro.

- Uma linguagem com números de precisão arbitrária mostra-se útil para que exista flexibilidade caso exista a necessidade de geração de números muito grandes. Assim sendo, a linguagem Python foi escolhida, pois além de sua alta legibilidade e grande número de recursos embutidos, é possível trabalhar com números de tamanho indefinido, dado poder computacional existente para tal. Uma possível implementação genérica para o MT está localizada em `mt19937.py`, e pode ser executada da seguinte maneira (`seed` deve ser um número inteiro, e os outros parâmetros utilizados são fornecidos pelos autores do PRNG, podendo divergir dado o tamanho do inteiro que deseja ser gerado):

```
$ python
>>> from mt19937 import MT19937
>>> mt19937_32 = (32, 624, 397, 31, 0x9908b0df, 11, 0xffffffff, 7,
                  0x9d2c5680, 15, 0xefc60000, 18, 1812433253)
>>> seed = 13104307
>>> MT19937(seed, *mt19937_32).generate()
2594696978
```

- A utilização de números primos com fins criptográficos é bem conhecida; um uso bastante comum é na modalidade assimétrica – chaves RSA são geradas a partir de números primos com um número de bits suficiente para que sejam seguras e praticamente inquebráveis. Assim sendo, devem existir testes de primalidade que sejam razoavelmente simples de modo a facilitar estes processos e viabilizar a criptografia. Os dois testes probabilísticos discutidos são o teste de *Fermat* e o teste de *Miller-Rabin*.

  O Pequeno Teorema de Fermat diz que, se p é primo e $0 < a < p$, então

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

  Deseja-se testar se p é primo, então é possível escolher inteiros a aleatórios no intervalo possível e verificar se a congruência é válida. Se isto acontecer para muitos valores de a, então p *provavelmente* é um primo. De modo contrário, se um inteiro a gera uma incongruência da forma

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

  então a é uma *testemunha* do fato de que p é composto.

*`gustavo.zambonin@grad.ufsc.br` — todos os algoritmos utilizados podem ser encontrados também neste repositório.
[1] a quantidade de números gerados antes da sequência começar a se repetir.
[2] número primo na forma $2^n - 1, n \in \mathbb{Z}$.
[3] o corpo de Galois de dois elementos, também representado por $GF(2)$.

O teste de Miller-Rabin adiciona facetas a este teorema: seja um primo p onde $p > 2$. $p - 1$ é um número par, e pode ser escrito como $2^s d$ (s, d inteiros e d ímpar). É possível verificar que um número não é primo se

$$a^d \not\equiv 1 \pmod{p} \text{ e } a^{2^r d} \not\equiv -1 \pmod{p} \; \forall \; (0 \leq r \leq s-1)$$

de modo que a novamente é uma testemunha. Procede-se da mesma maneira, escolhendo inteiros a aleatoriamente. Se qualquer uma das congruências acima proceder, então p não é primo. Similarmente, caso o método retorne ''verdadeiro'', então p provavelmente é primo. Uma demonstração sobre a origem das congruências acima pode ser encontrada em [2] e em diversos outros artigos posteriores que simplificam esta prova. Naturalmente, este teste tem uma maior complexidade e portanto um tempo de execução maior, porém isso é relevado pela maior precisão, pois independente do número testado, existe uma probabilidade de no mínimo $\frac{1}{2}$ de que este seja detectado como composto, o que não acontece no teste de Fermat, onde existem números mais e menos facilmente detectados.

O código relevante está localizado em `primality_test.py` e pode ser executado da seguinte maneira:

```
$ python
>>> from primality_test import fermat, miller_rabin
>>> n, k = 253559837810710172535057072944137070561, 10
>>> miller_rabin(n, k)
True
```

- Por fim, o código localizado em `find_primes.py` é um simples *script* para encontrar alguns primos de até 4000 bits. Sua saída mostra o tempo necessário para obter tal número, assim como o inteiro em si. É possível notar que o processo torna-se extremamente demorado com o número de bits > 1800 por conta das várias operações exponenciais e modulares nos testes de primalidade.

```
$ python find_primes.py
Time: 0:00:00.009934    Bits: 100
Number: 1107641301581031329462575872343
Time: 0:00:00.595348    Bits: 200
Number: 347025374246034602632061254452315635868455417461108122309671
...
```

Referências

[1] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, January 1998.

[2] Gary L. Miller. Riemann's hypothesis and tests for primality. *J. Comput. Syst. Sci.*, 13(3):300–317, December 1976.