Raízes Primitivas Módulo n

Ranieri S. Althoff[1]

[1]Universidade Federal de Santa Catarina
Departamento de Informática e Estatística
Segurança em Computação

1. Introdução

Em aritmética modular, uma raiz primitiva módulo n é um número g tal que todos os números $m \in \{1, 2, \ldots, n\}$ coprimos de n podem ser expressado na forma $g^x \equiv m \pmod{n}$. Por exemplo, 3 é uma raiz primitiva de 7, porque:

$$
\begin{aligned}
&3^1 = 3 \equiv 3 \pmod{7}\\
&3^2 = 9 \equiv 2 \pmod{7}\\
&3^3 = 27 \equiv 6 \pmod{7}\\
&3^4 = 81 \equiv 4 \pmod{7}\\
&3^5 = 243 \equiv 5 \pmod{7}\\
&3^6 = 729 \equiv 1 \pmod{7}
\end{aligned}
$$

O período de $3^k \pmod{7}$ é 6 e os restos nesse período são uma permutação do conjunto de coprimos de 7, implicando que 3 é uma raiz primitiva. Neste caso, como 7 é primo, todos os números anteriores são seus coprimos.

2. Definição

Se n é um inteiro positivo, os inteiros entre 1 e $n - 1$ que são coprimos de n formam um conjunto chamado **grupo multiplicativo módulo** n e é denotado por $\mathbb{Z}_n{}^x$, e esse grupo é cíclico se e somente se $n \in \{2, 4, p^k, 2p^k\}$ onde p^k é uma exponenciação de um número primo ímpar p.

Um número gerador g desse grupo cíclico é chamado de **raiz primitiva módulo n**, ou na teoria dos grupos multiplicativos, um **elemento primitivo de** $\mathbb{Z}_n{}^x$.

A quantidade de elementos em $\mathbb{Z}_n{}^x$ é $\phi(n)$, onde ϕ é a função totiente de Euler. O teorema de Euler diz que $a^{\phi(n)} \equiv 1 \pmod{n}$ para todo a coprimo de n, e o menor expoente de a que é congruente a 1 módulo n é chamado de **ordem multiplicativa** de a módulo n.

Se a ordem multiplicativa de um número a módulo n é $\phi(n)$, ou seja, o menor número x tal que $a^x \equiv 1 \pmod{n}$ é $\phi(n)$, então a é uma raiz primitiva módulo n.

3. Encontrando raízes primitivas

O algoritmo de encontrar raízes primitivas módulo n consiste em explorar as propriedades descritas, pois não há uma fórmula genérica para computar as raízes de números arbitrários. No entanto, é possível encontrar uma raiz primitiva de forma mais rápida do que força bruta.

Utilizando a inversa da última propriedade descrita, se a é uma raiz primitiva módulo n, então a ordem multiplicativa de a é $\phi(n)$, e isso pode ser usado para encontrar raízes multiplicativas.

3.1. Calculando $\phi(n)$

O primeiro passo é encontrar $\phi(n)$, para então poder testar possíveis valores de a. Para isso, basta calcular quantos números $k \in 2, 3, ..., n-1$ são coprimos de n, utilizando o seguinte código:

```
def phi(n):
    # soma 1 para cada i relativamente primo de n
    return sum(gcd(i, n) == 1 for i in range(1, n))
```

O que este código faz, descontando o açúcar sintático da linguagem, é somar 1 para cada número i que seja coprimo de n, ou seja, é análogo a seguinte expressão matemática:

$$\sum_{i=1}^{n-1} f(i), \text{ tal que } f(i) = \begin{cases} 1, \text{gcd(i, n)} = 1 \\ 0, \text{gcd(i, n)} \neq 1 \end{cases}$$

Se, por exemplo, utilizamos 9 como n, fazemos o somatório dos seguintes termos:

- $f(1) = 1$, pois gcd(1, 9) $= 1$
- $f(2) = 1$, pois gcd(2, 9) $= 1$
- $f(3) = 0$, pois gcd(3, 9) $= 3$
- $f(4) = 1$, pois gcd(4, 9) $= 1$
- $f(5) = 1$, pois gcd(5, 9) $= 1$
- $f(6) = 0$, pois gcd(6, 9) $= 3$
- $f(7) = 1$, pois gcd(7, 9) $= 1$
- $f(8) = 1$, pois gcd(8, 9) $= 1$

Portanto, $\phi(9) = 6$.

3.2. Fatores primos de $\phi(n)$

Depois de encontrado $\phi(n)$, é necessário determinar os fatores primos de $\phi(n)$, nomeados $p_1, \ldots, p_n$. Para encontrar os fatores, basta percorrer $i = 2, \ldots, n-1$ e encontrar um i que divida n. Para verificar sua primalidade, usa-se fatoração integral ou um teste probabilístico como o de Fermat ou de Miller-Rabin.

```
# gerador de fatores primos de n
def prime\_factors(n):
    # se n for par, gera 2
    if n&1 == 0:
        yield 2

    # gera p para cada p primo em [3, n/2] que divide n
    for p in range(3, n//2 + 1, 2):
        if n%p == 0 and prime(p):
            yield p

    # gera n se n for primo
    if prime(n):
        yield n
```

Este algoritmo utiliza um conceito de Python chamado **função geradora**, que é uma função que produz mais de um resultado com a *keyword* `yield`. Em termos simplificados, a cada `yield` a função retorna um valor para quem está a executando.

O uso de geradores ficará mais explícito na continuação deste código, mas a vantagem é que não é necessário usar tanta memória para guardar todos os fatores primos de n caso seja um número muito grande quando se retorna um por um.

O primeiro teste verifica o bit menos significativo de n, já que este bit representa a paridade e todo número par (ou seja, cujo bit menos significativo é 0), é divisível por 2. Como 2 é o único primo par, isso possibilita diminuir nosso espaço de busca de fatores pela metade, pois não é necessário buscar fatores pares (eles não são primos).

Além disso, só é necessário verificar fatores até a metade de n, porque a menor composição possível de um número é $2 \times k$, onde k é um inteiro qualquer e, portanto, igual a $n/2$. Em Python, $//$ representa divisão inteira, ou seja, com arredondamento para baixo.

Utilizando o mesmo $n = 9$, esse algoritmo retornaria apenas 3, porque é o único fator primo de 9. Na primeira execução do laço `for`, se verifica que $9 = 0 \pmod 3$, ou seja, 3 divide 9 exatamente, e 3 é primo, portanto é fator primo de 9. Nenhum outro valor p entre 3 e $9/2 = 4$ cumpre estas condições.

3.3. Utilizando o teorema de Fermat

É necessário verificar o teorema de Fermat $a^x \equiv 1 \pmod n$ para $a \in 2, \ldots, n-1$ e $x \in prime\_factors(n)$. Esse passo é simples, mas como requer laços aninhados para todos os valores de a e x, é um passo de alta complexidade.

É possível reduzir o espaço de busca considerando que, para que a seja uma raiz primitiva módulo n, a e n devem ser coprimos, ou seja, gcd(a, n) $= 1$.

Um código que satisfaça as constatações acima segue:

```
def primitive\_roots(n):
    phi\_n = phi(n)

    # testa para todos os valores de a (2 ate n-1)
    for a in range(2, n):
        # para m ser raiz primitiva de n devem ser relativamente primos
        if gcd(a, n) != 1:
            continue

        # verifica a\^(phi(n) / p) mod n para cada fator primo p de phi(n)
        if all(pow(a, phi\_n//p, n) != 1 for p in prime\_factors(phi\_n)):
            yield a
```

Novamente fazendo uso de geradores para deixar o código mais leve e limpo. Sendo bastante auto-explicativo, esse código encontra $\phi(n)$ e, para cada $a \in [2, n)$, executa o teorema de Fermat em todos os x fatores primos de $\phi(n)$.