Teoria dos Números

Funções Multiplicativas

Prof. Edson Alves

Faculdade UnB Gama

Funções Multiplicativas

Funções aritméticas

Definição de função arimética

Uma função é denominada função **aritmética** (ou **número-teórica**) se ela tem como domínio o conjunto dos inteiros positivos e, como contradomínio, qualquer subconjunto dos números complexos.

Funções multiplicativas

Definição de função multiplicativa

Uma função f aritmética é denominada função **multiplicativa** se

1. $f(1) = 1$
2. $f(mn) = f(m)f(n)$ se $(m, n) = 1$

Número de divisores

Definição de função $\tau(n)$

Seja n um inteiro positivo. A função $\tau(n)$ computa o número de divisores positivos de n.

Cálculo do valor de $\tau(n)$

- Segue diretamente da definição que $\tau(1) = 1$
- Suponha que $(a, b) = 1$
- Se d divide ab então ele pode ser escrito como $d = mn$, com $(m, n) = 1$, onde m divide a e n divide b
- Desde modo, qualquer divisor do produto ab será o produto de um divisor de a por um divisor de b
- Logo, $\tau(ab) = \tau(a)\tau(b)$, ou seja, $\tau(n)$ é uma função multiplicativa

Cálculo do valor de $\tau(n)$

- Considere a fatoração
$$n = p_1^{\alpha_1} p_2^{\alpha_2} \dots p_k^{\alpha_k}$$
- Se $n = p^k$, para algum primo p e um inteiro positivo k, d será um divisor de n se, e somente se, $d = p^i$, com $i \in [0, k]$
- Assim, $\tau(p^k) = k + 1$
- Portanto,
$$\tau(n) = \prod_{i=1}^{k} \tau(p_i^{\alpha_i}) = (\alpha_1 + 1)(\alpha_2 + 1) \dots (\alpha_k + 1)$$

Implementação da função $\tau(n)$ em C++

```
long long number_of_divisors(int n, const vector<int>& primes)
{
    auto fs = factorization(n, primes);
    long long res = 1;

    for (auto [p, k] : fs)
        res *= (k + 1);

    return res;
}
```

Cálculo de $\tau(n)$ em competições

- Em competições, é possível computar $\tau(n)$ em $O(\sqrt{n})$ diretamente, sem recorrer à fatoração de n
- Isto porque, se d divide n, então $n = dk$ e ou $d \leq \sqrt{n}$ ou $k \leq \sqrt{k}$
- Assim só é necessário procurar por divisores de n até $\sqrt{n}$
- Caso um divisor d seja encontrado, é preciso considerar também o divisor $k = n/d$
- Esta abordagem tem implementação simples e direta, sendo mais adequada em um contexto de competição

Implementação $O(\sqrt{n})$ de $\tau(n)$

```
long long number_of_divisors(long long n)
{
    long long res = 0;

    for (long long d = 1; d * d <= n; ++d)
    {
        if (n % d == 0)
            res += (d == n/d ? 1 : 2);
    }

    return res;
}
```

Soma dos divisores

Definição de função $\sigma(n)$

Seja n um inteiro positivo. A função $\sigma(n)$ retorna a soma dos divisores positivos de n.

Caracterização dos divisores de $n = ab$

- Sejam a e b dois inteiros positivos tais que $(a, b) = 1$ e $n = ab$
- Se c e d são divisores positivos de a e b, respectivamente, então cd divide n
- Por outro lado, se k divide n e $d = (k, a)$, então

$$k = d\left(\frac{k}{d}\right)$$

- Como $d = (k, a)$, em particular d divide a
- Uma vez que $(k/d, a) = 1$ e k divide $n = ab$, então k/d divide b
- Isso mostra que qualquer divisor $c = d_i e_j$ de n será o produto de um divisor d_i de a por um divisor e_j de b

Cálculo de $\sigma(n)$

- Da caracterização anterior segue que

$$\sigma(n) = \sum_{i=1}^{r} \sum_{j=1}^{s} d_i e_j$$

- Daí,

$$\sigma(n) = d_1e_1 + \ldots + d_1e_s + d_2e_1 + \ldots + d_2e_s + \ldots + d_re_1 + \ldots + d_re_s$$

Cálculo de $\sigma(n)$

- Esta expressão pode ser reescrita como

$$\sigma(n) = (d_1 + d_2 + \ldots + d_r)(e_1 + e_2 + \ldots + e_s)$$

- Portanto

$$\sigma(n) = \sigma(ab) = \sigma(a)\sigma(b)$$

- Como $\sigma(1) = 1$, a função $\sigma(n)$ é multiplicativa

Cálculo de $\sigma(n)$

- Deste modo, para se computar $\sigma(n)$ basta saber o valor de $\sigma(p^k)$ para um primo k e um inteiro positivo k
- O divisores de p^k são as potências p^i, para $i \in [0, k]$
- Logo
$$\sigma(p^k) = 1 + p + p^2 + \ldots + p^k = \left(\frac{p^{k+1} - 1}{p - 1}\right)$$

Implementação da função $\sigma(n)$ em C++

```cpp
long long sum_of_divisors(int n, const vector<int>& primes)
{
    auto fs = factorization(n, primes);
    long long res = 1;

    for (auto [p, k] : fs)
    {
        long long pk = p;

        while (k--)
            pk *= p;

        res *= (pk - 1)/(p - 1);
    }

    return res;
}
```

Cálculo de $\sigma(n)$ em competições

- De forma semelhante à função $\tau(n)$, é possível computar $\sigma(n)$ sem necessariamente fatorar n
- A estratégia é a mesma: listar os divisores de n, por meio de uma busca completa até $\sqrt{n}$, e totalizar os divisores encontrados
- Esta rotina tem complexidade $O(\sqrt{n})$

Implementação da função $\sigma(n)$ em $O(\sqrt{n})$

```
long long sum_of_divisors(long long n)
{
    long long res = 0;

    for (long long d = 1; d * d <= n; ++d)
    {
        if (n % d == 0)
        {
            long long k = n / d;

            res += (d == k ? d : d + k);
        }
    }

    return res;
}
```

Função φ de Euler

Definição de função $\varphi(n)$

A função $\varphi(n)$ de Euler retorna o número de inteiros positivos menores ou iguais a n que são coprimos com n.

Cálculo de $\varphi(n)$

- É fácil ver que $\varphi(1) = 1$ e que $\varphi(p) = p - 1$, se p é primo
- A prova que $\varphi(ab) = \varphi(a)\varphi(b)$ se $(a, b) = 1$ não é trivial (uma demonstração possível utiliza os conceitos de sistemas reduzidos de resíduos)
- Assim, $\varphi(n)$ é uma função multiplicativa
- Para p primo e k inteiro positivo, no intervalo $[1, p^k]$ apenas os múltiplos de p não são coprimos com p
- Os múltiplos de p são
$$p, 2p, 3p, \ldots, p^k$$
- Observe que $p^k = p \times p^{k-1}$

Cálculo de $\varphi(n)$

- Assim são p^{k-1} múltiplos de p em $[1, p^k]$ e portanto

$$\varphi(p^k) = p^k - p^{k-1} = p^{k-1}(p-1)$$

- Seja n um inteiro positivo tal que

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \ldots p_k^{\alpha_k}$$

- O valor de $\varphi(n)$ será dado por

$$\varphi(n) = \prod_{i=1}^{k} \varphi(p_i^{\alpha_i}) = \prod_{i=1}^{k} p_i^{\alpha_i - 1}(p_i - 1) = n \prod_{i=1}^{k} \left(1 - \frac{1}{p_i}\right)$$

Implementação de $\varphi(n)$ em C++

```cpp
int phi(int n, const vector<int>& primes)
{
    if (n == 1)
        return 1;

    auto fs = factorization(n, primes);
    auto res = n;

    for (auto [p, k] : fs)
    {
        res /= p;
        res *= (p - 1);
    }

    return res;
}
```

Cálculo de φ em $[1, n]$

- É possível computar $\varphi(k)$ para todos inteiros k no intervalo $[1, n]$ em $O(n \log n)$
- Para tal, basta utilizar uma versão modificada do crivo de Erastótenes
- Inicialmente, `phi[k] = k` para todos $k \in [1, n]$
- Para todos os primos p, os múltiplos i de p devem ser atualizados de duas formas:
    1. `phi[i] /= p`
    2. `phi[i] *= (p - 1)`

Cálculo de φ em $[1, n]$ com complexidade $O(n \log n)$

```
vector<int> range_phi(int n)
{
    bitset<MAX> sieve;
    vector<int> phi(n + 1);

    iota(phi.begin(), phi.end(), 0);
    sieve.set();

    for (int p = 2; p <= n; p += 2)
        phi[p] /= 2;
```

Cálculo de φ em $[1, n]$ com complexidade $O(n \log n)$

```
    for (int p = 3; p <= n; p += 2) {
        if (sieve[p]) {
            for (int j = p; j <= n; j += p) {
                sieve[j] = false;
                phi[j] /= p;
                phi[j] *= (p - 1);
            }
        }
    }

    return phi;
}
```

Referências

1. Mathematics LibreTexts. 4.2 Multiplicative Number Theoretic Functions. Acesso em 10/01/2021.
2. Wikipédia. Arithmetic function. Acesso em 10/01/2021.
3. Wikipédia. Multiplicative function. Acesso em 10/01/2021.