Teoria dos Números

Funções Multiplicativas

Prof. Edson Alves

Faculdade UnB Gama

Sumário

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

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

            res += (i == j ? i : i + j);
        }
    }

    return res;
}
```

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

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

- 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;
}
```

Problemas propostos

- AtCoder Beginner Contest 170D – Not Divisible
- AtCoder Beginner Contest 172D – Sum of Divisors
- Codeforces 1033D – Divisors
- OJ 10299 – Relatives
- OJ 12043 – Divisors

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.

Soluções dos problemas propostos

AtCoder Beginner Contest 172D – Sum of Divisors

Versão resumida do problema: calcule o valor S dado pela soma

$$S = \sum_{K=1}^{N} K \times \tau(K)$$

Restrição: $1 \leq N \leq 10^7$

Solução com complexidade $O(N \log N)$

- Assim como fizemos no caso da função φ de Euler, é preciso computar o valor de τ para todos inteiros no intervalo $[1, N]$ de forma eficiente
- Uma vez que τ é uma função multiplicativa, isto pode ser feito por meio de uma variante do crivo de Erastótenes
- O crivo permite a identificação de um dos fatores primos de n, para $n \in [2, N]$
- No código da solução será mantido, para cada n, o maior primo p que o divide
- Uma vez identificados estes fatores primos, o valor de $\tau(n)$ é computado em ordem crescente

Solução com complexidade $O(N \log N)$

- Lembre que $\tau(1) = 1$, de modo que esta computação inicia em $n = 2$
- Para cada n, utilizamos o fator p para escrever $n = p^k \times m$, com $(p^k, m) = 1$
- Daí
$$\tau(n) = \tau(p^k)\tau(m) = (k+1)\tau(m)$$
- Como $m < n$, pois p é primo, quando $\tau(n)$ estiver sendo computado o valor de $\tau(m)$ já estará disponível
- Como a fatoração parcial de n tem complexidade $O(\log n)$, a solução terá complexidade $O(N \log N)$, por conta do crivo modificado

Solução com complexidade $O(N \log N)$

```
vector<ll> factors(ll N)
{
    bitset<MAX> sieve;
    vector<ll> fs(N + 1, 1);

    sieve.set();

    for (ll i = 2; i <= N; i++)
        if (sieve[i])
            for (ll j = i; j <= N; j += i)
            {
                sieve[j] = false;
                fs[j] = max(fs[j], i);
            }

    return fs;
}
```

Solução com complexidade $O(N \log N)$

```
vector<ll> divisors(ll N, const vector<ll>& fs)
{
    vector<ll> tau(N + 1, 1);

    for (ll n = 2; n <= N; ++n)
    {
        ll k = 0, m = n;

        for (auto p = fs[i]; m % p == 0; ++k, m /= p);

        tau[n] = (k + 1)*tau[m];
    }

    return tau;
}
```

Solução com complexidade $O(N \log N)$

```
ll solve(ll N)
{
    auto fs = factors(N);
    auto tau = divisors(N, fs);
    ll ans = 0;

    for (ll K = 1; K <= N; ++K)
        ans += (K * tau[K]);

    return ans;
}
```

OJ 12043 – Divisors

Versão resumida do problema: compute as somas

$$g(a, b, k) = \sum_i \tau(i)$$
$$h(a, b, k) = \sum_i \sigma(i)$$

onde $a \leq i \leq b$ e i é divisível por k.

Restrições:

- $0 < a \leq b \leq 10^5$
- $0 < k < 2000$

Solução em $O(B \log B)$

- Para resolver este problema dentro do limite de tempo estabelecido, é preciso computar, de forma eficiente, as funções $\tau(m)$ e $\sigma(m)$ para todos os inteiros m de 1 a N
- Isto pode ser feito por meio de uma variante do crivo de Erastótenes
- Para os demais inteiros positivos tem, no mínimo, dois divisores: 1 e o próprio número
- A ideia portanto é iniciar os valores $\tau(m) = 2$ e $\sigma(m) = m + 1$
- Após esta inicialização, para cada inteiro positivo d no intervalo de 2 a N, devemos identificar quais inteiros m são divisíveis por d

Solução em $O(B \log B)$

- Para cada um destes inteiros os valores de $\tau(m)$ e $\phi(m)$ devem ser atualizados, de acordo com o valor de $k = m/d$
- Se $d \neq k$, então $\tau(m)$ deve ser acrescido em duas unidades, pois são dois novos divisores de m encontrados, e $\sigma(m)$ deve aumentar em $d + k$ unidades
- Nos casos em que $d = k$, $\tau(m)$ deve ser incrementado em uma única unidade, e o valor de $\sigma(m)$ deve ser acrescido em d unidades
- É preciso tomar cuidado para que nenhum divisor seja contabilizado mais de uma veze
- Assim, os múltiplos de d começarão a ser considerados a partir de d^2

Solução em $O(B \log B)$

- Conforme comentado no vídeo que apresentou o crivo de Erastótenes, ao proceder desta maneira os múltiplos de d menores que d^2 já foram processados anteriormente
- De posse dos valores pré-computados de $\tau(n)$ e de $\sigma(n)$, a soma pode ser feita de forma linear
- Para evitar iterar sobre valores que não são múltiplos de k, o laço deve iniciar no primeiro múltiplo m de k que é maior ou igual a a, e o incremento deve ser feito em passos de tamanho k
- Este múltiplo m pode ser obtido por meio da expressão $m = kt$, onde

$$t = \left\lceil \frac{a}{k} \right\rceil$$

Solução $O(B \log B)$

```
pair<vector<ll>, vector<ll>> tau_and_sigma(ll N)
{
    vector<ll> tau(N + 1, 1), sigma(N + 1, 1);

    for (ll m = 2; m <= N; ++m)
    {
        tau[m] = 2;
        sigma[m] = m + 1;
    }

```

Solução $O(B \log B)$

```
    for (ll d = 2; d <= N; ++d)
    {
        for (ll m = d*d; m <= N; m += d)
        {
            ll k = m / d;

            tau[m] += (d == k ? 1 : 2);
            sigma[m] += (d == k ? d : d + k);
        }
    }

    return { tau, sigma };
}
```

Solução $O(B \log B)$

```
pair<ll, ll>
solve(int a, int b, int k, const vector<ll>& tau, const vector<ll>& sigma)
{
    int t = (a + k - 1)/k, m = k*t;
    ll x = 0, y = 0;

    for (int i = m; i <= b; i += k)
    {
        x += tau[i];
        y += sigma[i];
    }

    return { x, y };
}
```

Codeforces 1033D – Divisors

Versão resumida do problema: compute o resto da divisão do produto

$$\prod_{i=1}^{n} a_i$$

por 998244353.

Restrição: cada elemento a_i tem no mínimo 3 e no máximo 5 divisores.

Solução $O(n^2)$

- A restrição do número de divisores de um elemento a_i implica em apenas 4 cenários distintos, onde p e q são primos distintos
  1. $a_i = p^2$
  2. $a_i = p^3$
  3. $a_i = pq$
  4. $a_i = p^4$
- No primeiro caso, $\tau(p^2) = 2 + 1 = 3$, ou seja, a_i tem 3 divisores
- Nos casos 2 e 3 temos 4 divisores, pois $\tau(p^3) = 3 + 1 = 4$ e $\tau(pq) = (1 + 1)(1 + 1) = 4$
- No último caso $\tau(p^4) = 5$
- A solução portanto, depende da identificação destes casos e do devido tratamento dado a eles

Solução $O(n^2)$

- É possível determinar se um número n é um quadrado perfeito por meio de uma rotina baseada em busca binária com complexidade $O(\log n)$
- Esta rotina pode identificar os casos 1 e 4
- Uma rotina semelhante identifica se n é um cubo perfeito também em $O(\log n)$, e pode identificar o caso 2
- Nos casos 1, 2 e 4 os primos identificados pelas rotinas acima devem ser acumulados em um histograma que contém a fatoração do produto de todos os termos, de acordo com o expoente em questão

Solução $O(n^2)$

- O caso 3 é o mais difícil e que merece mais atenção e cuidado
- Uma vez que $a_i \leq 2 \times 10^{18}$, a fatoração de tais termos não pode ser feita por meio de um algoritmo naive
- Uma forma de obter esta fatoração é computar o maior divisor comum d entre todos os pares de números da forma $a = pq$
- Caso d seja um divisor próprio destes números e d não for uma chave do histograma da fatoração, ele deve ser registrado no histograma, inicialmente associado ao expoente zero

Solução $O(n^2)$

- Após este processamento, as chaves primas do histograma podem ser usadas numa tentativa de fatoração destes números
- Caso um número possa ser fatorado, os dois fatores devem atualizar o histograma
- Se o número não for fatorado, ele não compartilha primos com os demais número, exceto possivelmente com cópias idênticas de si mesmo
- Assim, tais números devem ser guardados em um segundo histograma

Solução $O(n^2)$

- Finalizado todos estes passos, a resposta pode ser computada a partir da entrada de ambos histogramas
- A resposta inicialmente é igual a 1
- Para todo par (p, k) do primeiro histograma, a resposta deve ser atualizada por meio de seu produto por $(k + 1)$
- Para todo par (x, c) do segundo histograma, a atualização deve ser por meio do produto por $(c + 1)^2$
- Ou seja, para cada conjunto de c repetições do número $x = p_j q_j$, a fatoração do produto dos a_i conterá os fatores p_j^c e q_j^c, e cada um contribui com um fator $(c + 1)$ no cálculo do número de divisores deste produto

Solução $O(n^2)$

```
ll is_square(ll n)
{
    ll a = 1, b = n;

    while (a <= b) {
        auto m = a + (b - a)/2;

        if (n/m == m and m*m == n)
            return m;
        else if (m < n/m)
            a = m + 1;
        else
            b = m - 1;
    }

    return -1;
}
```

Solução $O(n^2)$

```
ll is_cube(ll n)
{
    ll a = 1, b = n;

    while (a <= b) {
        auto m = a + (b - a)/2;

        if ((n/m)/m == m and m*m*m == n)
            return m;
        else if (m < (n/m)/m)
            a = m + 1;
        else
            b = m - 1;
    }

    return -1;
}
```

Solução $O(n^2)$

```
ll solve(const vector<ll>& as)
{
    map<ll, ll> fs, uniques;
    vector<ll> pqs;

    for (auto a : as)
    {
        auto s = is_square(a);

        if (s > 0)
        {
            auto p = is_square(s);

            // a = p^4 ou a = p^2
            p > 0 ?  fs[p] += 4 : fs[s] += 2;
        }
```

Solução $O(n^2)$

```
        else
        {
            auto c = is_cube(a);

            // a = p^3 ou a = pq
            c > 0 ?  (void) (fs[c] += 3) : pqs.push_back(a);
        }
    }

    for (auto x : pqs)
        for (auto y : pqs)
        {
            auto d = gcd(x, y);

            if (d > 1 and d < x and fs.count(d) == 0)
                fs[d] = 0;
        }
```

Solução $O(n^2)$

```
    for (auto x : pqs)
    {
        bool ok = false;

        for (auto [p, k] : fs)
            if (x % p == 0)
            {
                ++fs[p];
                ++fs[x / p];
                ok = true;
                break;
            }

        if (not ok)
            uniques[x]++;
    }
```

Solução $O(n^2)$

```cpp
    ll ans = 1;

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

    for (auto [x, k] : uniques)
        ans = (ans * (k + 1)*(k + 1)) % MOD;

    return ans;
}
```