Teoria dos Números

Teorema Fundamental da Aritmética

Prof. Edson Alves

Faculdade UnB Gama

Sumário

Teorema Fundamental da Aritmética

Funções multiplicativas e números primos

- Uma função f é multiplicativa se $f(1) = 1$ e $f(ab) = f(a)f(b)$ se $(a, b) = 1$
- Uma consequência destas propriedades é que f pode ser determinada se conhecidos os valores que ela assume nas potências p^k de qualquer primo p
- Isto porque, se o inteiro positivo n é um produto de potências de primos, isto é

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

então

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

- O Teorema Fundamental da Aritmética nos garante que, se $n > 1$, então n poderá ser escrito como este produto de primos

Teorema Fundamental da Arimética

Teorema Fundamental da Aritmética

Seja n um inteiro positivo maior do que 1. Então n pode ser escrito de forma única, exceto pela ordem dos fatores, como o produto de números primos.

Uma notação canônica para esta fatoração de n é a seguinte: se p_i é o i-ésimo número primo e $\alpha_i \geq 0$, para todo $i \in [1, k]$, então

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

Fatoração de inteiros

- O Teorema Fundamental da Aritmética apresenta a relação fundamental dos números primos com todos os números naturais
- O conhecimento da fatoração (ou decomposição) de um natural n como produto de primos permite o cálculo de várias funções importantes, como MDC, MMC ou qualquer função multiplicativa
- A fatoração serve como alternativa para a representação do número, principalmente quando o número é muito grande (maior do que a capacidade de um `long long`, por exemplo)

Implementação da fatoração de inteiros

- Uma maneira de se representar a fatoração de um inteiro positivo $n > 1$ é por meio de um dicionário, onde chave é um primo p divisor de n, e o valor k é o maior expoente tal que p^k divide n
- É possível implementar a fatoração de duas formas
- Sem o conhecimento prévio da lista dos primos, a implementação terá complexidade $O(\sqrt{n})$
- O algoritmo será semelhante ao que lista todos os divisores de n, com duas diferenças

Implementação da fatoração de inteiros

- A primeira delas é que, ao encontrar um divisor p de n, o valor de n será atualizado para n/p^k
- A segunda é que, se após a verificação de todos os candidatos a divisores de n o valor de n permanecer maior do que 1, este valor remanescente será primo
- A fatoração pode ser implementada em $O(\pi(\sqrt{n}))$, se forem conhecida a lista dos primeiros primos
- O algoritmo será o mesmo, com a diferença de que os candidatos à divisores serão todos primos

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

```
map<long long, long long> factorization(long long n) {
    map<long long, long long> fs;

    for (long long d = 2, k = 0; d * d <= n; ++d, k = 0) {
        while (n % d == 0) {
            n /= d;
            ++k;
        }

        if (k) fs[d] = k;
    }

    if (n > 1) fs[n] = 1;

    return fs;
}
```

Implementação da fatoração em $O(\pi(\sqrt{n}))$

```cpp
map<long long, long long> factorization(long long n, vector<long long>& primes)
{
    map<long long, long long> fs;

    for (auto p : primes)
    {
        if (p * p > n)
            break;

        long long k = 0;

        while (n % p == 0) {
            n /= p;
            ++k;
        }

```

Implementação da fatoração em $O(\pi(\sqrt{n}))$

```
        if (k)
            fs[p] = k;
    }

    if (n > 1)
        fs[n] = 1;

    return fs;
}
```

MDC, MMC e fatoração

Sejam a e b dois inteiros positivos tais que $a = p_1^{\alpha_1} p_2^{\alpha_2} \dots p_k^{\alpha_k}$ e $b = p_1^{\beta_1} p_2^{\beta_2} \dots p_k^{\beta_k}$, com $\alpha_i, \beta_j \geq 0$ para todos $i, j \in [1, k]$. Então

$$(a, b) = p_1^{\min\{\alpha_1,\beta_1\}} p_2^{\min\{\alpha_2,\beta_2\}} \dots p_k^{\min\{\alpha_k,\beta_k\}}$$

e

$$[a, b] = p_1^{\max\{\alpha_1,\beta_1\}} p_2^{\max\{\alpha_2,\beta_2\}} \dots p_k^{\max\{\alpha_k,\beta_k\}}$$

Implemetação do MDC usando a fatoração de n

```
int gcd(int a, int b, const vector<int>& primes)
{
    auto ps = factorization(a, primes);
    auto qs = factorization(b, primes);

    int res = 1;

    for (auto p : ps) {
        int k = min(ps.count(p) ? ps[p] : 0, qs.count(p) ? qs[p] : 0);

        while (k--)
            res *= p;
    }

    return res;
}
```

Implemetação do MMC usando a fatoração de n

```cpp
int lcm(int a, int b, const vector<int>& primes)
{
    auto ps = factorization(a, primes);
    auto qs = factorization(b, primes);

    int res = 1;

    for (auto p : ps) {
        int k = max(ps.count(p) ? ps[p] : 0, qs.count(p) ? qs[p] : 0);

        while (k--)
            res *= p;
    }

    return res;
}
```

Fatoriais

Fatoração de Fatoriais

- Uma aplicação importante da fatoração é a fatoração de fatoriais
- Os fatoriais crescem rapidamente, e mesmo para valores relativamente pequenos de n, o número $n!$ pode ser computacionalmente intratável
- A fatoração de $n!$ permite trabalhar com tais números e realizar algumas operações com os mesmos (multiplicação, divisão, MMC, MDC, etc)
- A função $E(n, p)$ retorna um inteiro k tal que p^k é a maior potência do primo p que divide $n!$

Exemplo de cálculo de $E(n,p)$ para $n = 12$ e $p = 2$

Para ilustrar o cálculo de $E(n,p)$ considere $n = 12$ e $p = 2$. A expansão de $12!$ é

$$1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 \times 8 \times 9 \times 10 \times 11 \times 12$$

É fácil observar que todos os múltiplos de 2 contribuem com um fator 2. Cancelando estes fatores obtém-se

$$1 \times {\color{red}1} \times 3 \times {\color{red}2} \times 5 \times {\color{red}3} \times 7 \times {\color{red}4} \times 9 \times {\color{red}5} \times 11 \times {\color{red}6}$$

Exemplo de cálculo de $E(n, p)$ para $n = 12$ e $p = 2$

Ainda restam ainda fatores 2 no produto, onde haviam originalmente os números $4, 8$ e 12. Isto acontece por, além de serem múltiplos de 2, os números $4, 8$ e 12 também são múltiplos de 2^2.

Eliminando os fatores 2 associados a 2^2 resulta em

$$1 \times 1 \times 3 \times 1 \times 5 \times 3 \times 7 \times 2 \times 9 \times 5 \times 11 \times 3$$

Mais 3 fatores foram eliminados, e sobrou ainda um fator, onde estava o 8. Isto acontece também porque 8 é múltiplo de 2^3. Eliminando este último fator, foi removido um total de $6 + 3 + 1 = 10$ fatores do produto. Portanto $E(12, 2) = 9$.

Cálculo de $E(n, p)$

O exemplo anterior permite a deducação da expressão para o cálculo de $E(n, p)$:

$$E(n, p) = \left\lfloor \frac{n}{p} \right\rfloor + \left\lfloor \frac{n}{p^2} \right\rfloor + \ldots + \left\lfloor \frac{n}{p^r} \right\rfloor,$$

onde $\left\lfloor \frac{a}{b} \right\rfloor$ é a divisão inteira de a por b e p^r é a maior potência de p que é menor ou igual a n.

Implementação de $E(n, p)$ em C++ em $O(\log n)$

```cpp
int E(int n, int p)
{
    int k = 0, base = p;

    while (base <= n)
    {
        k += n / base;
        base *= p;
    }

    return k;
}
```

Implementação da fatoração de $n!$ em $O(\pi(n)\log n)$

```cpp
map<int, int> factorial_factorization(int n, const vector<int>& primes)
{
    map<int, int> fs;

    for (const auto& p : primes)
    {
        if (p > n)
            break;

        fs[p] = E(n, p);
    }

    return fs;
}
```

Problemas propostos

- AtCoder Beginner Contest 120B – K-th Common Divisor
- AtCoder Beginner Contest 148E – Double Factorial
- Codeforces 515C – Drazil and Factorial
- OJ 10061 – How many zero's and how many digits?
- OJ 10527 – Persistent Numbers

Referências

1. **HALIM**, Felix; **HALIM**, Steve. *Competitive Programming 3*, 2010.
2. **HEFEZ**, Abramo. Arimética, Coleção PROFMAT, SBM, 2016.
3. **LAAKSONEN**, Antti. *Competitive Programmer's Handbook*, 2018.
4. **SKIENA**, Steven S.; **REVILLA**, Miguel A. *Programming Challenges*, 2003.

Soluções dos problemas propostos

OJ 10527 – Persistent Numbers

Versão resumida do problema: compute o menor número x tal que n é igual ao produto de seus dígitos

Restrições: o número n tem até 1000 dígitos decimais.

Solução $O(\log n)$

- Se n é o produto dos dígitos de x, então a fatoração prima de n só pode conter primos cuja representação decimal só tem um dígito, a saber: 2, 3, 5 e 7
- Assim, se a fatoração de n tem qualquer outro primo o problema não terá solução
- Nos demais casos, a fatoração prima de n seria uma solução, embora nem sempre seja a mínima
- Para minimizar a solução, é preciso agrupar os fatores primos em dígitos compostos
- Antes de fazer esta redução, tratemos primeiro de um caso especial

Solução $O(\log n)$

- Mesmo não estando explícito no texto do problema, espera-se que x tenha, no mínimo, dois dígitos, conforme se observa nos exemplos de entrada e saída
- Assim, se n tiver um único digo, a solução mínima seria o número $10 + n$
- Nos demais casos, para minimizar x agruparemos os fatores primos nos compostos 9, 8, 6 e 4, nesta ordem, de forma gulosa
- Feito este agrupamento, x é formado por estes agrupamentos, ordenados do menor para o maior

Solução $O(\log n)$

```python
def factorization_2357(x):

    fs = [0]*10

    for p in [2, 3, 5, 7]:
        while x % p == 0:
            fs[p] += 1
            x //= p

    if x > 1:
        fs = []

    return fs
```

Solução $O(\log n)$

```python
def solve(n):

    if len(n) == 1:
        return '1' + n

    fs = factorization_2357(int(n))

    if not fs:
        return 'There is no such number.'

    fs[9] = fs[3] // 2
    fs[3] %= 2

    fs[8] = fs[2] // 3
    fs[2] %= 3

```

Solução $O(\log n)$

```python
    if fs[3] > 0 and fs[2] > 0:
        fs[6] = 1
        fs[2] -= 1
        fs[3] -= 1

    fs[4] = fs[2] // 2
    fs[2] %= 2

    x = 0

    for d in range(2, 10):
        while fs[d] > 0:
            x = 10*x + d
            fs[d] -= 1

    return str(x)
```

Solução $O(\log n)$

```python
if __name__ == '__main__':

    ns = [x.strip() for x in sys.stdin.readlines()]
    ns = takewhile(lambda x: x != '-1', ns)
    xs = map(solve, ns)

    print('\n'.join(xs))
```

AtCoder Beginner Contest 148E – Double Factorial

Versão resumida do problema: Determine o número de zeros à direita na representação decimal de $f(N)$, onde $f(1) = 1$ e

$$f(n) = nf(n-2), \quad \text{se } n \geq 2$$

Restrição: $0 \leq N \leq 10^{18}$

Solução com complexidade $O(\log N)$

- Observe que a função $f(n)$ é uma variante do fatorial, que computa o produto dos pares ou dos ímpares menores ou iguais a n, dependendo da paridade de n
- Se n for ímpar, então $f(n)$ será também ímpar, e portanto não terá nenhum zero à direita
- Se for um positivo n par, então $f(n)$ pode ser reescrita como

$$f(n) = 2^{n/2} \times \left(\frac{n}{2}\right)!$$

- Neste caso, $f(n) = 2^r 5^s m$, onde $(2, m) = 1 = (5, m)$, $s = E(n/2, 5)$ e $r = n/2 + E(n/2, 2)$

Solução com complexidade $O(\log N)$

- A representação decimal de $f(n)$ terá um zero à direita para cada par de fatores 2 e 5
- Assim, a solução S do problema será dada por $S = \min(r, s)$
- Como $s \leq r$, pois $E(n/2, 2) \geq E(n/2, 5)$, então S de fato corresponde a $E(n/2, 5)$
- Esta solução tem complexidade $O(\log n)$

Solução com complexidade $O(\log N)$

```
ll solve(ll N)
{
    if (N % 2)
        return 0;

    N /= 2;

    ll ans = 0, base = 5;;

    while (N >= base)
    {
        ans += (N / base);
        base *= 5;
    }

    return ans;
}
```

Codeforces 515C – Drazil and Factorial

Versão resumida do problema: determinar o maior positivo x tal que x não contenha nem zeros nem uns em sua representação, e que $F(x) = F(a)$, onde $F(n)$ é o produto dos fatoriais dos dígitos de n.

Restrição: o número a tem entre 1 e 15 dígitos decimais.

Solução $O(n \log n)$

- Para determinar o valor de x é preciso, inicialmente, determinar a fatoração prima de $F(a)$
- Como $F(a)$ é o produto do fatorial de cada dígito de a, esta fatoração conterá, no máximo, 4 primos distintos: 2, 3, 5 e 7
- Esta fatoração será composta pelo produto das fatorações de cada dígito de a
- Uma vez que há apenas 10 dígitos decimais e alguns deles podem se repetir em a, podemos usar um histograma para evitar o cálculo de uma mesma fatoração repetidas vezes

Solução $O(n \log n)$

- Observe que o menor fatorial que contém o primo p em sua fatoração é $p!$
- Como desejamos o maior x possível, podemos escolher, gulosamente, o maior dentre os fatoriais $2!, 3!, 5!$ e $7!$ que ainda pode ser formado com os fatores disponíveis
- A cada fatorial escolhido é preciso atualizar a lista dos fatores disponíveis
- Eventualmente o resultado pode exceder os limites de um `long long`, então utilize uma string para armazenar o resultado, evitando assim o *overflow*

Solução $O(n \log n)$

```
map<int, int> fact_factorization(int n)
{
    map<int, int> fs;

    for (auto p : { 2, 3, 5, 7 })
        fs[p] = E(n, p);

    return fs;
}
```

Solução $O(n \log n)$

```
vector<int> histogram(long long n)
{
    vector<int> hs(10, 0);

    while (n)
    {
        ++hs[n % 10];
        n /= 10;
    }

    return hs;
}
```

Solução $O(n \log n)$

```
string solve(long long a)
{
    auto hs = histogram(a);
    map<int, int> fs;

    for (int i = 2; i <= 9; ++i)
        for (auto [p, k] : fact_factorization(i))
            fs[p] += k*hs[i];

    string ans;

    for (auto p : { 7, 5, 3, 2 })
    {
        auto qs = fact_factorization(p);
        auto n = fs[p];

```

Solução $O(n \log n)$

```
            for (auto q : { 2, 3, 5, 7 })
                if (q <= p)
                    n = min(n, fs[q] / qs[q]);

        for (int i = 0; i < n; ++i)
            ans.push_back((char) ('0' + p));

        for (auto q : { 2, 3, 5, 7 })
            fs[q] -= (n*qs[q]);
    }

    return ans;
}
```