Aula 7 · Teoria dos Números: Fatoração, Crivos e Algoritmo de Euclides

Desafios de Programação

Fernando Kiotheka Vinicius Tikara Date

UFPR

23/10/2024

Apresentação de Teoria dos Números

Maratonista só pensa em número primo...

- Teoria dos números é uma área da matemática que estuda os números inteiros e funções com números inteiros.
- Cientistas da computação são bem fãs de números inteiros, porque é o que conseguimos representar nos computadores sem problemas (a história é diferente com os números reais).
- O registro mais antigo desse estudo pode ser traçado à Mesopotamia, 1800 BCE, em uma tablete (de argila) com uma lista de trios pitagóricos ($a^2 + b^2 = c^2$).
- Um dos mais importantes problemas não resolvidos na matemática é a Hipótese de Riemann que implicaria em resultados sobre a distribuição de números primos.
- **Sempre** cai na maratona.

O Teorema Fundamental da Aritmética

O Teorema Fundamental da Aritmética afirma que todo inteiro $N > 1$ pode ser representado como um produto único de números primos não necessariamente distintos:

$$50 = 2 \cdot 5 \cdot 5 = 2^1 \cdot 5^2$$

Nomeamos isso a fatoração prima de um número.

Fatoração por divisão por tentativa

- Método que testa os números $\leq \sqrt{N}$ para ver se é divisível.
- Ao encontrar um fator, divide o N quantas vezes for possível.
- No pior caso (por exemplo se o número for um primo gigante), a complexidade é de $\mathcal{O}(\sqrt{N})$.

Implementação de fatoração por divisão por tentativa

Código `factorize.cpp`

```cpp
#include <bits/stdc++.h>
using namespace std; using ll = long long;
void factorize /* O(sqrt(n)) */ (ll n) {
    for (ll i = 2; i*i <= n; i++)
        while (n % i == 0) { n /= i; cout << i << " "; }
    if (n > 1) { cout << n << " "; }
    cout << "\n";
}
int main() { int n; while (cin >> n) { factorize(n); } }
```

Entrada	Saída
50	2 5 5
1000000007	1000000007
1000000009	1000000009
1000696969	1000696969
998244353	998244353
720720	2 2 2 2 3 3 5 7 11 13

Primos e coprimos

- Números primos são aqueles cuja fatoração prima é composta apenas por ele mesmo (ou alternativamente, só podem ser divididos por 1 ou por ele mesmo).
- 1 não é primo.
- São infinitos e são numerosos. A prova é deixada para matemáticos do século 3 BCE.
- Sua densidade pode ser aproximada por

$$\pi(n) \approx \frac{n}{\ln n}$$

- $\pi(10^6) = \frac{10^6}{\ln 10^6} \approx 72382$ (o número real é 78496)
- **Coprimos:** dois números com fatoração completamente distinta ($15 = 3 \cdot 5$ e $26 = 2 \cdot 13$, por exemplo, são coprimos)

Somas harmônicas e complexidades estranhas

Por conta da densidade dos primos, essa aula tem alguma complexidades "esquisitas". Em particular:

$$\lim_{n\to\infty}\left(\sum_{\substack{p \text{ primo}\\ p\leq n}}\frac{1}{p}-\ln(\ln n)\right)=M\approx 0{,}2614972\ldots$$

A constante de Meissel-Mertens acima nos deixa concluir que um laço sobre números primos é $\mathcal{O}(\lg\lg n)$.
Ao mesmo tempo, a constante de Euler-Mascheroni abaixo leva à ideia de que um laço com passos de tamanho incremental é $\mathcal{O}(\lg n)$:

$$\lim_{n\to\infty}\left(\sum_{k=1}^{n}\frac{1}{k}-\ln n\right)=\gamma\approx 0{,}5772156$$

Crivos e geradores de números

Crivos

- Vêm da ideia de, a partir de uma lista de números, filtrar os que não satisfizerem alguma condição.
- **Crivo de Eratóstenes:** obter primos em $\mathcal{O}(n \lg \lg n)$ e com menor uso de memória (suporta algo na magnitude de 10^9 números)
- **Crivo de Euler:** obter primos em $\mathcal{O}(n)$ (potencialmente mais lento que o crivo de Eratóstenes) sob um custo maior de memória (mas que permite uma fatoração mais rápida)
- **Crivos segmentados:** construir primos maiores usando menos memória (não veremos aqui)
- **Extensões do crivo:** `https://codeforces.com/blog/entry/22229` (veremos aqui)
- Mais coisas assim no capítulo 5 do livro dos Halim.

Crivo de Erastótenes

Código `sieve.cpp`

```
#include <bits/stdc++.h>
using namespace std;
vector<bool> sieve (1e7+15, true);
void eratosthenes /* O(n lg lg n) */ (int n) {
    for (int i = 2; i * i <= n; i++) if (sieve[i])
        for (int j = i * i; j <= n; j += i)
            sieve[j] = false;
}
int main() {
    int n; cin >> n;
    eratosthenes(n);
    for (int i = 2; i <= n; i++)
        if (sieve[i]) { cout << i << " "; }
    cout << "\n";
}
```

Entrada	Saída
30	2 3 5 7 11 13 17 19 23 29

Crivo de Euler

Código `linearsieve.cpp`

```
#include <bits/stdc++.h>
using namespace std;
vector<int> pr, lp (1e7+15);
void eulersieve /* O(n) */ (int n) {
    for (int i = 2; i <= n; i++) {
        if (lp[i] == 0) { lp[i] = i; pr.push_back(i); }
        for (int j = 0; j < int(pr.size())
                && pr[j] <= lp[i] && i*pr[j] <= n; j++)
            lp[i * pr[j]] = pr[j];
    }
}
int main() {
    int n; cin >> n; eulersieve(n);
    for (int p : pr) { cout << p << " "; } cout << "\n";
}
```

Entrada	Saída
30	2 3 5 7 11 13 17 19 23 29

Maior divisor primo em $\mathcal{O}(n \lg \lg n)$

Código `bigpsieve.cpp`

```
int big[n + 1] = {1, 1};
for (int i = 1; i*i <= n; ++i)
    if (big[i] == 1)
        for (int j = i; j <= n; j += i)
            big[j] = i;
```

Número de divisores

Sabendo a fatoração de um número inteiro, podemos calcular quantos divisores (primos e não primos) ele tem. Se

$$n = p_1^{e_1} p_2^{e_2} \dots p_k^{e_k}$$

Então n tem

$$d(n) = (e_1 + 1)(e_2 + 1) \dots (e_k + 1)$$

divisores.

Intuição: qualquer divisor de n vai ter em sua fatoração um subconjunto da fatoração de n, então o i-ésimo primo na fatoração de n vai ter expoente $0 \leq f \leq e_i$.

Implementação de um gerador de número de divisores

Código `divisors.cpp`

```
vector<int> divisors (n + 1);

void number_of_divisors /* O(n lg n) */ (int n) {
  for (int i = 1; i <= n; ++i)
    for (int j = i; j <= n; j += i)
      divisors[j]++;
}
```

Soma dos divisores

Numa linha de raciocínio similar, a soma dos divisores de n será somar todas as possibilidades para cada primo que divide n:

$$(p_1^0 + p_1^1 + p_1^2 + ... + p_1^{e_1})(p_2^0 + p_2^1 + p_2^2 + ... + p_2^{e_2})...$$

Da matemática discreta, sabemos que

$$\sum_{i=0}^{n} p_1^i = \frac{p_1^{e_1+1} - 1}{p_1 - 1}$$

Então, a soma dos divisores de n será

$$\prod_{i=1}^{k} \frac{p_i^{e_i+1} - 1}{p_i - 1}$$

Implementação de um gerador de soma de divisores

Código `sumdivsieve.cpp`

```
void sumdiv /* O(n lg n) */ (int n) {
    int sumdiv[n+1];
    for (int i = 1; i <= n; ++i)
        for (int j = i; j <= n; j += i)
            sumdiv[j] += i;
}
```

Função totiente de Euler

- A função totiente de euler conta quantos números menores ou iguais a n que são coprimos com n.
- Ela tem propriedades multiplicativas: $\phi(mn) = \phi(m)\phi(n)$.
- Para um p primo, $\phi(p) = p - 1$
- Podemos implementá-la em $O(n \lg \lg n)$.

$$\phi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right).$$

+	1	2	3	4	5	6	7	8	9	10
0	1	1	2	2	4	2	6	4	6	4
10	10	4	12	6	8	8	16	6	18	8
20	12	10	22	8	20	12	18	12	28	8
30	30	16	20	16	24	12	36	18	24	16
40	40	12	42	20	24	22	46	16	42	20
50	32	24	52	18	40	24	36	28	58	16

Implementação da função totiente de Euler

Código totient.cpp

```
#include <bits/stdc++.h>
using namespace std; using ll = long long;
const int N = 1e6+15;
vector<ll> phi (N);
void euler_totient /* O(n lg lg n) */ (int n) {
    for (int i = 1; i <= n; ++i)
        phi[i] = i;
    for (int i = 2; i <= n; ++i)
        if (phi[i] == i)
            for (int j = i; j <= n; j += i)
                phi[j] -= phi[j] / i;
}
int main() {
    euler_totient(1e6);
    int n; while (cin >> n) { cout << phi[n] << "\n"; }
}
```

Quero contar, mas não cabe no meu inteiro!

- Muitos problemas pedem contagens, mas geralmente o número pode ser gigantesco e não cabe no inteiro.
- Para resolver esse problema, quem cria os problemas geralmente usa aritmética modular usando um módulo **primo**.
- Isso significa que a resposta serve somente para provar que você sabe contar, mas não é o número exato.

Invariantes do módulo

Invariante é uma propriedade de um objeto matemático que permanece inalterada depois de operações ou transformações de algum tipo. Dentro da aritmética modular, temos várias operações onde as classes dos números são invariantes:

- $a + m \pmod{m} = a \pmod{m}$.
- $a + b \pmod{m} = (a \pmod{m}) + (b \pmod{m}) \pmod{m}$.
- $a * b \pmod{m} = (a \pmod{m}) * (b \pmod{m}) \pmod{m}$.
- $a - b \pmod{m} = (a \pmod{m}) - (b \pmod{m}) + m \pmod{m}$.
- $a^b \pmod{m} = (a \pmod{m})^b \pmod{m}$.

Cuidado com a subtração!

No C e no C++, o módulo de um número negativo com módulo positivo é negativo, então some o módulo para garantir números positivos. O Python não compartilha do mesmo comportamento.

Código `negmod.cpp`

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    int a = ((5 - 8) % 10 - 14) % 10;
    cout << a << " " << (a + 10) % 10 << "\n";
}
```

Entrada	Saída
	`-7 3`

Código `negmod.py`

```python
print(a := ((5 - 8) % 10 - 14) % 10, a + 10 % 10)
```

Entrada	Saída
	`3 3`

Inverso multiplicativo

Resultado do pequeno teorema de Fermat:

$$a^{p-1} = 1 \pmod{p},$$

onde p é **primo**.
Isso é bem útil para podermos dividir em módulo. Afinal,

$$\begin{aligned}
aa^{-1} &= 1 && \pmod{p} \\
aa^{-1} &= a^{p-1} && \pmod{p} \\
a^{-1} &= a^{p-1}a^{-1} && \pmod{p} \\
a^{-1} &= a^{p-2}. && \pmod{p}
\end{aligned}$$

Então usando a exponenciação, conseguimos o inverso multiplicativo em módulo!

Exponenciação binária

- Multiplicar pode ser caro: a^b é feito em $\mathcal{O}(b)$, e em problemas de contagem b pode ser grande.
- Podemos fazer exponenciação em $\lg b$ se aproveitando do fato que $a^b = (a^{\frac{b}{2}})^2$.
- Para o caso discreto:
  - $a^b = (a^{b/2})^2$ se b é par.
  - $a^b = a(a^{\lfloor b/2 \rfloor})^2$ caso contrário.
- Podemos fazer a conta $\pmod n$.

Implementação da exponenciação binária

Código `pow.cpp`

```
#include <bits/stdc++.h>
using namespace std; using ll = long long;
const int M = 1e9;
ll fast_pow(ll a, ll b) {
    if (b == 0) { return 1; } if (b == 1) { return a; }
    ll res = fast_pow(a, b/2);
    res = (res * res) % M;
    if (b % 2) res = (res * a) % M;
    return res;
}
int main() {
    int a, b;
    while (cin >> a >> b) cout << fast_pow(a, b) << "\n";
}
```

Entrada	Saída
2 100000	883109376
3 100000	522000001

Triângulo de Pascal

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, \text{para todo } (n \geq 0) \text{ e } (0 \leq k \leq n).$$

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n.$$

Implementação do triângulo de Pascal

Código pascal.cpp

```
#include <bits/stdc++.h>
using namespace std;
vector<vector<int>> binom (5, vector<int>(5));
int main() {
    for (int n = 0; n <= 4; n++) {
        cout << (binom[n][0] = 1);
        for (int k = 1; k <= n; k++) {
            binom[n][k] = binom[n-1][k-1] + binom[n-1][k];
            cout << " " << binom[n][k];
        } cout << "\n"; }
}
```

Entrada	Saída
	1 1 1 1 2 1 1 3 3 1 1 4 6 4 1

Cuidado com a memória!

Essa implementação é útil, mas pode ser muito custosa em memória. Ao invés disso você pode usar a outra definição de combinação:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

E pré-calcular o fatorial (usando um vetor como acima) de todos os números necessários.

Teorema de Euler (da teoria dos números)

Já vimos na Aula 2, o pequeno teorema de Fermat:

$$a^{p-1} = 1 \pmod{p},$$

onde p é **primo**.
Porém Euler o generalizou, sendo que

$$a^{\phi(n)} = 1 \pmod{n}$$

onde n é qualquer módulo, e a é coprimo com n, e $\phi(n)$ é a função totiente de Euler. É claro que $\phi(p) = p - 1$, onde p é primo.

Outro corolário do Teorema de Euler

Para todo inteiro positivo n, se a é coprimo com n então

$$x = y \pmod{\phi(n)} \implies a^x = a^y \pmod{n}.$$

Isso significa que

$$a^k = a^{(k \bmod \phi(n))} \pmod{n}.$$

MDC e MMC

Máximo Divisor Comum e Mínimo Múltiplo Comum

O **Máximo Divisor Comum** (MDC) entre dois números a e b é o maior inteiro g que divide tanto a quanto b.
Já o **Mínimo Múltiplo Comum** (MMC) entre a e b é o menor inteiro divisível por a e b.
Como obtê-los?

- **MDC:** Algoritmo de Euclides
- **MMC:** $\frac{ab}{\mathrm{MDC}(a,b)}$

Importante: Desde o C++17, `gcd` e `lcm` já são funções prontas do C++, mas conhecer a sua definição pode ajudar bastante!

Implementação do Algoritmo de Euclides

Código `euclides.cpp`

```
ll gcd /* O(lg min(a, b)) */ (ll a, ll b) {
        if (b == 0) { return a; }
        return gcd(b, a%b);
}

ll lcm /* O(lg min(a, b)) */ (ll a, ll b) {
        return a*(b/gcd(a, b));
}
```

Estendendo o Algoritmo de Euclides

- A complexidade do algoritmo de Euclides é $\mathcal{O}(\lg \min\{a, b\})$.
- Podemos calcular o inverso multiplicativo mais rápido do que $\mathcal{O}(\lg P)$ (exponenciação binária de primos), mas pra isso precisamos saber do x.

$$ax = g \pmod{b}, g = 1$$

$$\implies ax = 1 \pmod{b}$$

- Queremos resolver $ax + by = gcd(a, b)$. Faremos isso com o algoritmo de Euclides, salvando os valores de x e y.
- Quais valores x e y assumirão sabendo que na chamada subsequente obtivemos x_1 e y_1?

Estendendo o Algoritmo de Euclides (continuado)

Na chamada recursiva, obtivemos:

$$bx_1 + (a \bmod b)y_1 = g$$

Nesta chamada, estamos tratando de:

$$ax + by = g$$

Sabendo que $a \pmod b = a - \left\lfloor \frac{a}{b} \right\rfloor b$, manipulamos as equações para obter:

$$g = ay_1 + b\left(x_1 - y_1 \left\lfloor \frac{a}{b} \right\rfloor\right)$$

Isto é,

$$x = y_1$$

e

$$y = x_1 - y_1 \left\lfloor \frac{a}{b} \right\rfloor$$

Implementação do Euclides estendido

Código `extgcd.h`

```
ll extgcd /* O(lg min(a, b)) */ (ll a, ll b, ll& x, ll& y) {
    if (b == 0) { x = 1; y = 0; return a; }
    ll g = extgcd(b, a%b, x, y);
    tie(x, y) = make_tuple(y, x - (a/b)*y);
    return g;
}

ll inv(ll a, ll m) {
    ll x, y; extgcd(a, m, x, y);
    return ((x % m) + m) % m;
}
```