Teoria dos números +

André Gustavo dos Santos

Departamento de Informática
Universidade Federal de Viçosa

INF 333 - 2024/1

Fonte

Todos os códigos foram retirados do livro texto da disciplina: *Competitive Programming: Increasing the Lower Bound of Programming Contests*, de Steven Halim e Felix Halim

Teste primo

Teste

Verificar divisibilidade de n com valores 2 a $n - 1$

Teste rápido

Verificar divisibilidade de n com valores 2 a $\sqrt{n}$

Teste + rápido

Verificar divisibilidade de n com valores primos de 2 a $\sqrt{n}$

Teste primo

Implementação do teste + rápido

- Pré-processamento
  - Gerar primos $\leq 10.000.000$ pelo Crivo de Eratóstenes
- Verificação
  - Se $n \leq 10.000.000$ usar marcação do crivo
  - Senão testar divisibilidade com os primos do crivo

Note que isso só funciona se $n \leq 100.000.000.000.000$*. Por quê?*

Lista de primos

```
#include <bitset>     // mais eficiente que vector<bool>!
ll _tam_crivo;        // ll definido com: typedef long long ll;
bitset<10000010> bs; // 10^7 + extra bits, suficiente para maioria
vector<int> primos;  // lista de primos

void crivo(ll limite) { // cria lista de primos em [0..limite]
  _tam_crivo = limite + 1; // + 1 para incluir limite
  bs.reset(); bs.flip(); // todos valendo true
  bs.set(0, false); bs.set(1,false); // exceto indices 0 e 1
  for (ll i = 2; i <= _tam_crivo; i++)
    if (bs.test((size_t)i)) {
      //corta todos os multiplos de i comecando de i*i
      for (ll j = i*i; j <= _tam_crivo; j += i)
        bs.set((size_t)j, false);
      primos.push_back((int)i); // adiciona na lista de primos
  }
} // OBS: chamar esse metodo na funcao main!
...
```

Lista de primos

```
...
bool eh_primo(ll N) {  // metodo rapido para teste de primalidade
  if (N < _tam_crivo)
    return bs.test(N); // O(1) para primos pequenos
  for (int i=0; i<primos.size(); i++)
    if (N % primos[i] == 0)
      return false;
  return true;         // demora mais quando N e' primo
} // OBS: so funciona se N <= (ultimo primo do vector primos)^2

int main() {
  ...
  crivo(10000000);
  cout << eh_primo(5915587277);
  ...
}
```

Fatoração

```
vector<int> primeFactors{int N) {
  vector<int> factors;
  int PF_idx = 0, PF = primos[PF_idx]; //primos gerado pelo crivo

  while (N!=1 && (PF*PF <= N) { //ate sqrt(N), mas N vai diminuind
    while (N%PF == 0) {
      N /= PF;                  //retira esse fator do N
      factors.push_back(PF);    //e o adiciona na lista
    }
    PF = primos[++PF_idx];      //considera somente primos
  }
  if (N!=1) factors.push_back(N); //caso especial, se N for primo
  return factors;
}
```

Usar primos do crivo é opcional, também funciona com `PF` *= 2, 3, 4,...*

Fatoração

```
int main {
  crivo(100); // prepara lista de primos [0..100]
              // com isso, pode fatorar ate 100^2 = 10.000

  vector<int> result= primeFactors(10000); //fatora 10.000
  vector<int>::iterator last =
    unique(result.begin(), result.end());  //remove duplicados
  for(vector<int>>iterator it = result.begin(); it != last; it++)
    cout << *it << endl;                   //escreve 2 e 5
  ...
}
```

Para 10.000, primeFactors retorna 2 2 2 2 5 5 5 5 e o código acima escreve 2 5

Funções envolvendo fatores primos

Número de fatores primos

Como no código de fatoração, mas contando em vez de adicionar na lista

Ex.: $60 = 2 \times 2 \times 3 \times 5$*, são 4 fatores*

Número de divisores

Se $N = a^i \times b^j \times \cdots \times c^k$ é a fatoração,
então N tem $(i+1) \times (j+1) \times \cdots \times (k+1)$ divisores.

Ex.: $60 = 2^2 \times 3^1 \times 5^1$*, são* $(2+1)(1+1)(1+1) = 12$ *divisores.*

(a título de informação, os 12 fatores são $1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60$*)*

Número de inteiros $< N$ relativamente primos com N

Totiente de Euler, a seguir

Totiente de Euler (Phi)

Produto de Euler para cálculo de $\varphi(n)$

$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right)$ *Obs.: p|n significa p é divisor de n*

Quantos números $<$ n são relativamente primos com n

```
int EulerPhi(int N) {
  vector<int> factors = primeFactors(N);
  vector<int>::iterator new_end = unique(factors.begin(),
                                         factors.end()); // get unique
  int result = N;
  for (vector<int>::iterator it = factors.begin(); it!=new_end; it++)
    result = result - result / *it;
  return result;
}
```

MDC e MMC - versões compactas

```
#define ll long long

ll mdc(ll a, ll b) {return (b==0 ? a : mdc(b, a%b) ); }
ll mmc(ll a, ll b) {return (a * (b / mdc(a, b) ) ); }
```

Busca-ciclos

O problema

Busca-ciclos (*cycle-finding*, ou *cycle-detection*) é o problema de detectar ciclos em sequências de valores gerados por funções iteradas

Definição

- Para toda função $f : S \leftarrow S$ e um valor inicial $x_0 \in S$, sendo S um conjunto finito, a sequência de valores iterados da função
  $x_0, x_1 = f(x_0), x_2 = f(x_1), \ldots, x_i = f(x_{i-1}), \ldots$
  deve em algum momento repetir um valor (ciclo), isto é, $\exists i \neq j | x_i = x_j$.
- Depois, a sequência segue repetindo o ciclo de valores de x_i a x_{j-1}.
- Seja μ o menor índice e seja λ o menor valor tal que $x_\mu = x_{\mu+\lambda}$.
  (μ é o início e λ é o tamanho do ciclo)
- O problema busca-ciclos consiste em determinar μ e λ dados f e x_0.

Busca-ciclos

Algoritmo

- Considere um array de booleanos de tamanho $|S|$
- "Seguir" a sequência, marcando os valores x_i visitados
- Para cada x_j gerado, se o valor já foi marcado, tem-se o ciclo, com $\mu = i, \lambda = j - i$
- O algoritmo gasta tempo $O(\mu + \lambda))$ e espaço $O(|S|)$

Para gastar menos espaço

- Usar `set`
- Assim gasta espaço $O(\mu + \lambda))$
  (mas aumenta o tempo para verificar se o elemento já ocorreu)

Busca-ciclos

Algoritmo de Floyd (lebre e tartaruga)

O algoritmo também gasta tempo $O(\mu + \lambda)$ mas espaço $O(1)$

A ideia central é usar dois índices

- tartaruga: segue a sequência normalmente
- lebre: segue a sequência com o dobro da velocidade
- Após entrarem no ciclo, eventualmente os dois se encontram

Busca-ciclos (lebre e tartaruga)

```
pair<int, int> floyd_cycle_finding (int (*f)(int), int x0) {

  //Fase principal, encontrar uma repeticao x_i = x_2i
  //lebre com velocidade o dobro da tartaruga
  int tart = f(x0), lebr = f(f(x0));
  while (lebr != tart) { tart = f(tart); lebr = f(f(lebr)); }

  //Encontrar mu, inicio do ciclo
  //lebre e tartaruga na mesma velocidade
  int mu = 0; lebr = tart; tart = x0;
  while (lebr != tart) { tart = f(tart); lebr = f(lebr); mu+; }

  //Encontrar o tamanho do ciclo comecando de x_mu
  //lebre se move, tartaruga parada
  int lamb = 1; lebr = f(tart);
  while (lebr != tart) { lebr = f(lebr); lamb++; }

  return make_pair(mu,lamb);
}
```