Teoria dos números +

André Gustavo dos Santos

Departamento de Informática
Universidade Federal de Viçosa

INF 492 - 2012/1

Todos os códigos foram retirados do livro *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 $\sqrt{n}$

Implementação do teste + rápido

- Pré-processamento
  - Gerar primos < 10.000 pelo Crivo de Eratóstenes
- Verificação
  - Se $n < 10.000$ usar marcação do crivo
  - Senão testar divisibilidade com os primos do crivo

Note que isso só funciona se $n \leq 100.000.000$. 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
  ...
}
```

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

```
#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) ) ); }
```

Note que no mmc a divisão é feita antes. Por quê?

Java BigInteger Class

UVa 10925 - Krakovia

link pro enunciado

- Simplesmente somar e dividir
- Mas os números não cabem num `long long`
- + fácil com BigInteger

Java BigInteger Class

```
import java.io.*;
import java.util.*;
import java.math.*;

class Main{ /* UVa 10925 - Krakovia */
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int caseNo = 1;
    while (true) {
      int N = sc.nextInt(), F = sc.nextInt(); // N bills, F friends
      of (N == 0 && F == 0) break;

      BigInteger sum = BigInteger.valueOf(0); // use valueOf to initialize
      for (int i = 0; i < N; i++) { // sum the N large bills
        BigInteger V = sc.nextBigInteger(); // for reading next BigInteger
        sum = sum.add(V); // this is the BigInteger addition
      }

      System.out.println("Bill #" + (caseNo++) + " costs " + sum +
         ": each friend should pay " + sum.divide(BigInteger.valueOf(F)));
      System.out.println(); // the line above is BigInteger division
                            // divide the large sum to F friends
} } }
```

UVa 10814 - Simplifying Fractions

link pro enunciado

- Simplesmente dividir pelo MDC
- Mas os números não cabem num `long long`
- + fácil com BigInteger, ainda mais com certa função pronta...

Java BigInteger Class

```
import java.io.*;
import java.util.*;
import java.math.*;

class Main{ /* UVa 10814 - Simplifying Fractions */
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int N = sc.nextInt();
    while (N-- >0) {
      BigInteger p = sc.nextBigInteger();
      String ch = sc.next(); //ignore this one
      BigInteger q = sc.nextBigInteger();
      BigInteger qcd_pq = p.gcd(q); // wow :)
      System.out.println(p.divide(gdc_pq) + " / "
                         + q.divide(gcd_pq));
} } }
```

Java BigInteger Class

LA 4104 - MODEX

link pro enunciado

- Simplesmente calcular $x^y \mod n$
- Para evitar *overflow* deveria usar mod durante o cálculo
- Para não exceder tempo limite deveria usar o fato que $x^y = x^{y/2} \times x^{y/2} \times x^{y\%2}$
- + fácil com BigInteger, que tem certa função pronta...

Java BigInteger Class

```
import java.io.*;
import java.util.*;
import java.math.*;

class Main{ /* LA 4104 - MODEX */
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int nTC = sc.nextInt();
    while (nTC-- >0) {
      BigInteger x = BigInteger.valueOf(scan.nextInt());
      BigInteger y = BigInteger.valueOf(scan.nextInt());
      BigInteger n = BigInteger.valueOf(scan.nextInt());
      System.out.println(x.modPow(y, n)); //it's in the library:)
} } }
```

UVa 10551 - Basic Remains

link pro enunciado

- Calcular $p \bmod m$ (numa base b)
- Como p pode ter 1.000 dígitos, + fácil com BigInteger
- + fácil ainda com conversão de base pronta...

Java BigInteger Class

```
import java.io.*;
import java.util.*;
import java.math.*;

class Main{ /* UVa 10551 - Basic Remains */
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int b = sc.nextInt();
    if (b == 0) break;
    String p_str = sc.next();
    BigInteger p = new BigInteger(p_str, b);  //special constructor!
    String m_str = sc.next();
    BigInteger m = new BigInteger(m_str, b);  //2nd parameter is the base
    System.out.println((p.mod(m)).toString(b));  //can output in any base
} } }
```

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 $f : S \to S$ 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 (tamanho do ciclo) o menor valor tal que $x_\mu = x_{\mu+\lambda}$
- 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 $O(\mu + \lambda)$

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);
}
```