SCC-210
Algoritmos Avançados

Capítulo 8
Teoria dos Números

Adaptado por João Luís G. Rosa

Introdução

- A Teoria dos Números é uma das mais bonitas e interessantes áreas da matemática.
- É o ramo da matemática que se preocupa com as propriedades dos números inteiros:
  - Envolve muitos problemas práticos que são facilmente compreendidos mesmo por não-matemáticos;
  - Muitos desses problemas podem ser encontrados em situações práticas, científicas e problemas de competição.
- Existe uma coleção de algoritmos interessantes relacionados que solucionam problemas de forma inteligente e eficiente.

Números Primos

- Um **número primo** é um inteiro $p > 1$ que somente é divisível por 1 e por ele mesmo:
  - Se p é um número primo, então $p = a \times b$ para inteiros $a \leq b$ implica que $a = 1$ e $b = p$;
  - Os 10 primeiros primos são: 2, 3, 5, 7, 11, 13, 17, 19, 23 e 27.
- Qualquer número não-primo é chamado de **composto**.

Encontrando Primos

- A forma mais simples de testar se um número n é primo é por divisões sucessivas:
  - Comece pelo menor divisor candidato e tente todos os possíveis divisores maiores;
  - Como 2 é o único primo par (por que?), uma vez que for verificado que n não é par, pode-se verificar somente números ímpares como possíveis fatores;
  - Por fim, pode-se considerar n como primo tão logo não se consiga encontrar fatores primos menores que $\sqrt{n}$ (por que?).

Encontrando Primos

```
int is_prime (unsigned int x) {
     unsigned int i;

     if (x > 2 && x % 2 == 0)
          return 0;

     i = 3;
     while (i <= sqrt(x)+1) {
          if (x % i == 0)
               return 0;
          i += 2;
     }
     return 1;
}
```

sqrt(c) pode ser pré-calculado fora do laço. **sqrt(c)+1** para evitar problemas de precisão.

Teorema Fundamental da Aritmética

- Números primos são importantes por causa do **teorema fundamental da aritmética**.
  - Qualquer número inteiro *n* pode ser expresso somente de uma forma como um produto de primos.
    - Por exemplo, 105 é unicamente expresso como $3 \times 5 \times 7$, e 32 como $2 \times 2 \times 2 \times 2 \times 2$.
- Esse coleção de números que quando multiplicada resulta em *n* é chamada de **fatoração em primos** de *n*.

Fatoração em Primos

- A ordem não importa em uma fatoração em primos:
  - Pode-se canonicamente listar os números em ordem crescente;
  - Mas a multiplicidade de cada primo importa: é o que diferencia a fatoração em primos de 4 da de 8.
- Um número primo p é um **fator** de x se ele ocorre na fatoração em primos de x.

Fatoração em Primos

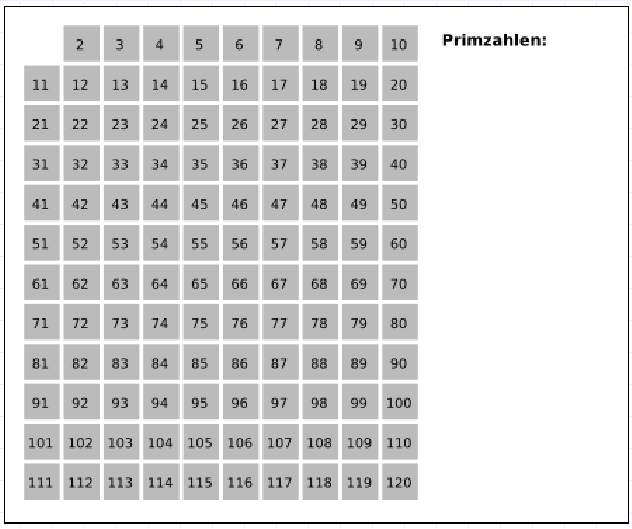
```
void prime_factorization(long x) {
    long i;     /* counter */
    long c;     /* remaining product to factor */

    c = x;
    while ((c % 2) == 0) {
        printf("2\n");
        c /= 2;
    }
    i = 3;
    while (i <= (sqrt(c)+1)) {
        if ((c % i) == 0) {
            printf("%ld\n", i);
            c /= i;
        } else
            i += 2;
    }
    if (c > 1) printf("%ld\n", c);
}
```

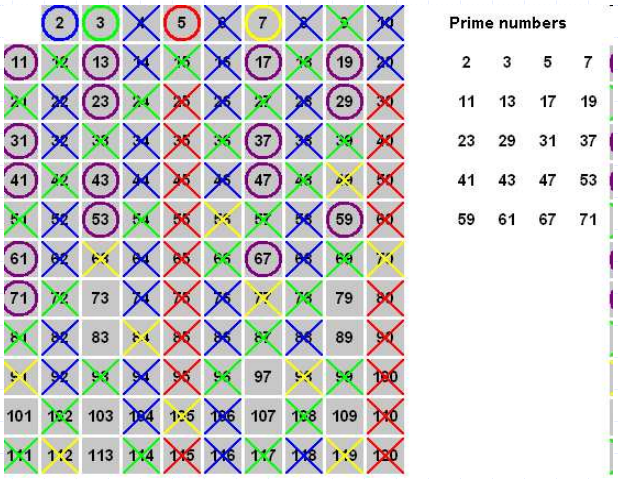
sqrt(c) pode ser calculado somente quando o valor de **c** é alterado. **sqrt(c)+1** para evitar problemas de precisão.

i é incrementado em duas unidades, portanto segue a seqüência de números ímpares. Uma abordagem mais rápida seria ter uma lista de primos.

Crivo de Eratóstenes



Crivo de Eratóstenes



Crivo de Eratóstenes

- A complexidade do algoritmo é $O((n \log n)(\log \log n))$.
- E possui um requisito de memória de $O(n)$.

Divisibilidade

- Um inteiro b divide a, denotado por $b|a$, se $a = bk$ para algum inteiro k:
  - Equivalentemente, b é um divisor de a ou a é um múltiplo de b, se $b|a$;
  - Como conseqüência, o menor divisor natural de qualquer número é 1.
- Como se pode encontrar todos os divisores de um número inteiro?
  - Pode-se utilizar os fatores primos.

Divisibilidade

- A partir do teorema fundamental da aritmética sabe-se que um inteiro *n* é unicamente representado pelo produto de seus fatores primos:
  - Cada divisor é o produto de algum subconjunto dos fatores primos;
  - Tais subconjuntos podem ser construídos utilizando *backtracking*;
  - Mas cuidado com fatores primos repetidos. Por exemplo, 12 (2, 2 e 3) e divisores (1,2,3,4,6,12).

Factors and Factorials (Uva 160)

The factorial of a number *N* (written *N*!) is defined as the product of all the integers from 1 to *N*. It is often defined recursively as follows:

1! = 1

N! = N * (N-1)!

Factorials grow very rapidly--5! = 120, 10! = 3,628,800. One way of specifying such large numbers is by specifying the number of times each prime number occurs in it, thus 825 could be specified as (0 1 2 0 1) meaning no twos, 1 three, 2 fives, no sevens and 1 eleven.

Factors and Factorials (Uva 160)

Write a program that will read in a number *N* ($2 \leq N \leq 100$) and write out its factorial in terms of the numbers of the primes it contains.

Input

Input will consist of a series of lines, each line containing a single integer *N*. The file will be terminated by a line consisting of a single 0.

Factors and Factorials (Uva 160)

Output

Output will consist of a series of blocks of lines, one block for each line of the input. Each block will start with the number N, right justified in a field of width 3, and the characters `!', space, and `='. This will be followed by a list of the number of times each prime number occurs in *M*!.

These should be right justified in fields of width 3 and each line (except the last of a block, which may be shorter) should contain fifteen numbers. Any lines after the first should be indented. Follow the layout of the example shown below exactly.

Factors and Factorials (Uva 160)

Sample input

```
5
53
0
```

Sample output

```
  5! =  3  1  1
 53! = 49 23 12  8  4  4  3  2  2  1  1
  1  1  1  1  1
```

Factovisors (UVa 10139)

PC/Uva IDs: 110704/10139, Popularity: A, Sucess rate: ave, Level: 2

The factorial function, *n*! is defined as follows for all non-negative integers *n*:

```
0! = 1
n! = n * (n-1)!    (n > 0)
```

We say that *a* divides *b* if there exists an integer *k* such that

```
k*a = b
```

The input to your program consists of several lines, each containing two non-negative integers, *n* and *m*, both less than 2^31. For each input line, output a line stating whether or not *m* divides *n*!, in the format shown below.

Factovisors

Sample Input	Output for Sample Input
6 9	9 divides 6!
6 27	27 does not divide 6!
20 10000	10000 divides 20!
20 100000	100000 does not divide 20!
1000 1009	1009 does not divide 1000!

Máximo Divisor Comum

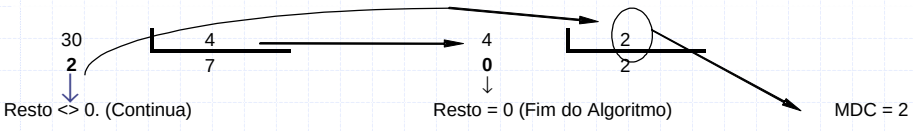
- Como 1 divide qualquer inteiro, então o mínimo divisor de qualquer par de inteiros é 1.
- Mais interessante é o **máximo divisor comum**, ou **mdc**, ou seja, o maior divisor comum compartilhado por um par de inteiros.
- Diz-se que dois inteiros *a* e *b* são **relativamente primos** se o *mdc*(*a*, *b*) = 1.

Máximo Divisor Comum

- O algoritmo de Euclides para encontrar o mdc de dois inteiros é considerado o primeiro algoritmo interessante da história.
  - Outras formas seriam testar todos os divisores de *a* em *b*;
  - Ou encontrar os fatores primos de *a* e *b* e calcular o produto de todos os fatores comuns;
  - Ambas as abordagens são computacionalmente intensivas.

Algoritmo de Euclides

Ex: a = 30 e b = 4:



Máximo Divisor Comum

- O algoritmo de Euclides se baseia em duas observações:
  - Se $b|a$, então $mdc(a, b) = b$;
  - Se $a = bt + r$, então $mdc(a, b) = mdc(b, r)$.
- O algoritmo de Euclides pode ser aplicado recursivamente, substituindo $max(a, b)$ pelo resto da divisão de $max(a, b)$ por $min(a, b)$.
- O algoritmo de Euclides realiza um número logarítmico de iterações.

Máximo Divisor Comum

- Por exemplo, se $a = 34398$ e $b = 2132$:

$mdc(34398, 2132) = mdc(34398 \% 2132, 2132) = mdc(2132, 286)$
$mdc(2132, 286) = mdc(2132 \% 286, 286) = mdc(286, 130)$
$mdc(286, 130) = mdc(286 \% 130, 130) = mdc(130, 26)$
$mdc(130, 26) = mdc(130 \% 26,26) = mdc(26, 0)$

Portanto, $mdc(34398, 2132) = 26$.

Máximo Divisor Comum

```
long mdc(long a, long b) {
    if (b > a)
        return mdc(b, a);
    if (b == 0)
        return a;
    return mdc(b, a%b);
}
```


Mínimo Múltiplo Comum

- Outra função interessante entre dois inteiro *a* e *b* é o **mínimo múltiplo comum**, **mmc**:
  - É o menor inteiro divisível por *a* e *b*;
  - Por exemplo, mmc(24, 36) = 72.
- Uma aplicação de mmc é o cálculo da periodicidade entre dois eventos periódicos distintos:
  - Qual é o próximo ano (após 2000) que a eleição presidencial (4 anos) coincidirá com o censo (10 anos)?
  - Os eventos coincidem a cada 20 anos, pois mmc(4,10) = 20.
  - Logo, como a última vez (antes de 2000) em que a eleição e o censo coincidiram foi em 1990, haverá coincidência novamente em 2010 (este ano!).

Mínimo Múltiplo Comum

- É evidente que $mmc(a, b) \geq max(a, b)$. De forma similar, como $a \times b$ é múltiplo de ambos a e b, então $mmc(a, b) \leq ab$.
- O algoritmo de Euclides provê uma forma eficiente de calcular mmc, uma vez que $mmc(a, b) = ab/mdc(a, b)$.
  - Entretanto, é necessário ter cuidado com a possibilidade de *overflow* na multiplicação de a por b.
  - Existe um algoritmo em (Dijkstra, 76) que não utiliza a multiplicação.

Aritmética Modular

- Em diversos problemas, está-se interessado em conhecer o resto de divisões de inteiros:
  - Por exemplo, dado que o seu aniversário é em uma quarta-feira, quando será o seu aniversário no próximo ano?
  - Basta saber o resto da divisão de 365 (ou 366) por 7, ou seja, 365 % 7 = 1 ou 366 % 7 = 2;
  - Portanto, o aniversário pode cair em uma quinta ou em uma sexta-feira, dependendo se o ano atual é bissexto ou não.

Aritmética Modular

- **Aritmética modular** permite que diversos cálculos similares sejam feitos de forma eficiente, ou seja, sem o uso de aritmética de grandes números *(big numbers)*.
- O número dividido é chamado de **módulo** e o resto de **resíduo**.
- As operações aritméticas podem ser realizadas da seguinte maneira...

Aritmética Modular

- Adição
  - $(x + y)\ \%\ n = ((x\ \%\ n) + (y\ \%\ n))\ \%\ n$
  - Quantos centavos eu tenho se receber R$123,45 da minha mãe e R$94,67 do meu pai?
  - ((12.345 % 100) + (9.467 % 100)) % 100 = (45 + 67) % 100 = 12 % 100.
- Subtração
  - Pode-se considerar uma adição com números negativos.
  - Quantos centavos eu tenho após gastar R$52,53?
  - (12 % 100) – (53 % 100) = -41 % 100 = 59 % 100.

Aritmética Modular

- Multiplicação:
  - Pode-se considerar uma adição repetida.
  - $xy \% n = (x \% n)\,(y \% n) \% n$.
  - Quantos centavos você terá se receber R$17,28 por hora com 2.143 horas trabalhadas?
  - $(1.728 \times 2.143) \% 100 = (1.728 \% 100) \times (2.143 \% 100) \% 100 = 1.204 \% 100 = 4 \% 100$.
- Exponenciação:
  - $x^y \% n = (x \% n)^y \% n$

Aritmética Modular

- Aritmética modular possui diversas aplicações interessantes:
  - Encontrar os últimos dígitos.
  - Algoritmo de criptografia RSA.
  - Cálculos de Calendário.

Aritmética Modular

- Encontrar os últimos dígitos
  - Qual é o último dígito de 2^{100}? (ou seja, 2^{100} % 10):
    - 2^3 % 10 = 8
    - 2^6 % 10 = 8 × 8 % 10 = 4
    - 2^{12} % 10 = 4 × 4 % 10 = 6
    - 2^{24} % 10 = 6 × 6 % 10 = 6
    - 2^{48} % 10 = 6 × 6 % 10 = 6
    - 2^{96} % 10 = 6 × 6 % 10 = 6
    - 2^{100} % 10 = 2^{96} × 2^3 × 2^1 % 10 = **6**

Just the Facts (UVa 568 Cap.5*)

- The expression *N!*, read as "*N* factorial," denotes the product of the first *N* positive integers, where *N* is nonnegative. So, for example,

N	*N*!
0	1
1	1
2	2
3	6
4	24
5	120
10	3628800

* Agradecimentos a Alan Evangelista e Raphael Ferras

Just the Facts (UVa 568)

- For this problem, you are to write a program that can compute the last non-zero digit of any factorial for (0 <= *N* <= 10000). For example, if your program is asked to compute the last nonzero digit of 5!, your program should produce "2" because 5! = 120, and 2 is the last nonzero digit of 120.
- **Input**
  - Input to the program is a series of nonnegative integers not exceeding 10000, each on its own line with no other letters, digits or spaces. For each integer *N*, you should read the value and compute the last nonzero digit of *N*!.

Just the Facts (UVa 568)

- **Output**
  - For each integer input, the program should print exactly one line of output. Each line of output should contain the value *N*, right-justified in columns 1 through 5 with leading blanks, not leading zeroes. Columns 6 - 9 must contain " -> " (space hyphen greater space). Column 10 must contain the single last non-zero digit of *N*!.

Just the Facts (UVa 568)

Sample Input

```
1
2
26
125
3125
9999
```

Sample Output

```
   1 -> 1
   2 -> 2
  26 -> 4
 125 -> 8
3125 -> 2
9999 -> 8
```


Just the Facts (UVa 568)

```
long long res = 1;
int lastDigit = 0;
for (int i=2, i <= N; i++)
{
    res *= i;
    while (res%10==0)
        res/=10;
    res%=1000000;
}
lastDigit = res%10;
```

Carmichael Numbers (UVa 10139)

PC/Uva IDs: 110702/10006, Popularity: A, Sucess rate: ave, Level: 2

Certain cryptographic algorithms make use of big prime numbers. However, checking whether a big number is prime is not so easy.

Randomized primality tests exist that offer a high degree of confidence of accurate determination at low cost, such as the Fermat test. Let *a* be a random number between 2 and *n* - 1, where *n* is the number whose primality we are testing. Then, *n* is *probably* prime if the following equation holds:

$$a^n \bmod n = a$$

Carmichael Numbers (UVa 10139)

If a number passes the Fermat test several times, then it is prime with a high probability. Unfortunately, there is bad news. Certain composite numbers (non-primes) still pass the Fermat test with every number smaller than themselves. These numbers are called Carmichael numbers.

Write a program to test whether a given integer is a Carmichael number.

Carmichael Numbers (UVa 10139)

Input

The input will consist of a series of lines, each containing a small positive number n ($2 < n < 65{,}000$). A number $n = 0$ will mark the end of the input, and must not be processed.

Output

For each number in the input, print whether it is a Carmichael number or not as shown in the sample output.

Carmichael Numbers (UVa 10139)

Sample Input

```
1729
17
561
1109
431
0
```

Output for Sample Input

```
The number 1729 is a Carmichael
  number.
17 is normal.
The number 561 is a Carmichael
  number.
1109 is normal.
431 is normal.
```

Referências

- Batista, G. & Campello, R.
  - Slides disciplina *Algoritmos Avançados*, ICMC-USP, 2007.
- Skiena, S. S. & Revilla, M. A.
  - *Programming Challenges – The Programming Contest Training Manual*. Springer, 2003.
- Wikipedia:
  - **`http://pt.wikipedia.org/wiki/Crivo_de_Erat%C3%B3stenes`**