Algoritmos fundamentais

Pedro Vasconcelos, DCC/FCUP

Outubro 2019

Algoritmos

O que é um algoritmo?

- Um método para resolver uma *classe de problemas*; e.g.:
  - calcular o máximo divisor comum de dois inteiros;
  - colocar uma sequência de valores por ordem crescente;
  - encontrar o caminho mais curto entre duas cidades num mapa
- Especificado de forma *não-ambigua*
  - uma sequência de instruções precisas
- Efetivo:
  - executável num *número finito de passos* com papel e lápis (em princípio)

Nesta aula

Dois algoritmos clássicos sobre números inteiros:

- testar se um número é primo

- calcular o máximo divisor comum de dois números

Vamos ainda ver a *recursão*: uma forma alternativa aos ciclos para especificar repetição.

Testar números primos

Números primos

Um número inteiro positivo n é *primo* se for divisível apenas por 1 e por n:

2, 3, 5, 7, 11, 13 ...

Vamos especificar um algoritmo para testar se um número é primo.

Algoritmo

Dado: n inteiro.

Se $n \leq 1$ então **não é primo** e terminamos imediatamente.

Se $n > 1$ tentamos para $d = 2, 3, \ldots, n - 1$:

- se d divide n terminamos (**não é primo**)
- caso contrário, continuamos a procurar

Se chegamos ao final sem encontrar um divisor: concluimos que n **é primo**.

É um algoritmo

- Testa primalidade para n qualquer
- Sequência de operações não-ambíguas
    - operações aritméticas elementares
- Termina sempre
    - número finito de possíveis divisores

Implementação

```
#define FALSE 0
#define TRUE  1

/* Testar se um número inteiro é primo */

int primo(int n) {
   int d;
   if(n <= 1) return FALSE;
   for (d = 2; d < n; d++) {
     if (n%d == 0)  // d divide n?
       return FALSE;
    }
   return TRUE;
}
```

Observações

- Se n tem um divisor maior do que $\sqrt{n}$, então também tem um divisor inferior a $\sqrt{n}$.
- Logo podemos limitar a procura de divisores de 2 até $\sqrt{n}$
- Evitamos computação desnecessária quando o número é primo
- Em vez da testar a condição

$$d \leq \sqrt{n}$$

  podemos testar

$$d^2 \leq n$$

e evitamos o cálculo da raíz quadrada

Implementação (2)

```
/* Testar se um número é primo;
   versão mais eficiente */

int primo(int n)
{
   int d;
   if(n <= 1) return FALSE;
   for (d = 2; d*d <= n; d++) {
     if (n%d == 0)   // d divide n
       return FALSE;
    }
   return TRUE;
}
```

Máximo divisor comum

Exemplo

O *máximo divisor comum* (mdc) de dois inteiros a, b é o maior número inteiro que divide a e b.

Exemplo:

$$252 = 21 \times 12$$
$$105 = 21 \times 5$$

- 21 é divisor de 252 e 105
- 21 é o *maior* divisor destes dois números (porquê?)
- Logo, 21 é o mdc de 252 e 105

Algoritmo de Euclides

Dados: a, b dois números **inteiros positivos**.

1. se $a = b$ então terminamos; o mdc é a e b (são iguais).
2. se $a > b$ então:
   $a \leftarrow a - b$ e repetimos o passo 1.
3. se $a < b$ então:
   $b \leftarrow b - a$ e repetimos o passo 1.

Exemplo de execução

Calcular o mdc de 252 e 105.

iter	a	b
0	252	105
1	147	105
2	42	105
3	42	63
4	42	21
5	21	21

R: 21

Implementação

```
/* Calcular o mdc de dois inteiros positivos
   pelo Algoritmo de Euclides (1ª versão)
   */
int mdc(int a, int b)
{
   while (a != b) {
     if(a > b)
         a = a - b;
      else
         b = b - a;
   }
   return a; // a, b são iguais
}
```

Será um algoritmo?

OK:

- sequência de instruções ✓
- operações não-ambíguas ✓

Questões:

1. **Correção**: porque obtemos o mdc no final?
2. **Terminação**: será que o ciclo termina sempre?

Correção

Propriedade da divisão inteira:

Se d divide a e b, então d divide $a - b$ e d divide $b - a$.

- Cada iteração do ciclo preserva *todos* os divisores dos números iniciais
- Logo: preserva também o *maior divisor*

$$\mathrm{mdc}(a,\, b) = \mathrm{mdc}(a-b,\, b)$$
$$\mathrm{mdc}(a,\, b) = \mathrm{mdc}(a,\, b-a)$$

- No fim: se $a = b$ então $\mathrm{mdc}(a,\, b) = a = b$.

Terminação

Em cada iteração:

$$\begin{aligned} &\text{se } a > b: \quad (a,b) \longrightarrow (a-b,b) \\ &\text{se } a < b: \quad (a,b) \longrightarrow (a,b-a) \end{aligned}$$

- No 1º caso diminuimos o valor de a
- No 2º caso diminuimos o valor de b
- Como a, b são sempre inteiros positivos, este processo não pode continuar infinitamente

Observações

- Se a é muito maior do b vamos repetidamente subtrair b a a até chegar a um *resto* inferior a b
- Podemos obter esse resto num só passo efetuando uma *divisão inteira*
- Por isso a formulação moderna do algoritmo de Euclides usa divisões em vez de subtrações
- Bónus: o algoritmo funciona todos os inteiros não-negativos (incluido zero)

Algoritmo de Euclides (2)

(Versão usando resto da divisão.)

Dados: a, b dois inteiros não-negativos.

1. se $b = 0$ então terminamos; o mdc é a.

2. caso contrário:
    - $r \leftarrow$ resto da divisão de a por b
    - $a \leftarrow b$
    - $b \leftarrow r$
    - repetimos o passo 1.

Exemplo

Calcular o mdc de 252 e 105.

iter	**a**	**b**	**resto a ÷ b**
0	252	105	42
1	105	42	21
2	42	21	0
3	21	0	

R: 21

Implementação

```
/* Calcular o mdc de dois inteiros usando
   o algoritmo de Euclides (2ª versão)
   */
int mdc(int a, int b)
{
    int r;
    while(b != 0) {
        r = a%b;
        a = b;
        b = r;
    }
    return a;
}
```

Recursão

Definições recursivas

- A definição de uma função diz-se *recursiva* se usamos a função na sua própria definição
- Exemplo, a função *factorial* definida recursivamente:

$$\begin{aligned} \mathrm{fact}(0) &= 1 \\ \mathrm{fact}(n) &= n \times \mathrm{fact}(n-1)\,, \quad \text{se } n > 0 \end{aligned}$$

Algoritmos recursivos

A definição anterior define um *algoritmo*: permite calcular o factorial de qualquer inteiro não negativo.

Exemplo:

$$
\begin{aligned}
\text{fact}(4) &= 4 \times \text{fact}(3) \\
&= 4 \times (3 \times \text{fact}(2)) \\
&= 4 \times (3 \times (2 \times \text{fact}(1))) \\
&= 4 \times (3 \times (2 \times (1 \times \text{fact}(0)))) \\
&= 4 \times (3 \times (2 \times (1 \times 1))) \\
&= 24
\end{aligned}
$$

Recursão em linguagem C

Podemos implementar este processo defindo a função recursiva em C:

```
int fact(int n)
{
   if (n == 0)
      return 1;              // caso base
   else
      return n * fact(n-1);  // caso recursivo
}
```

Caso base e recursivo

Há dois casos na definição anterior:

caso base ($n = 0$)

o factorial de zero é 1 (sem mais chamadas recursivas)

caso recursivo ($n > 0$)

calculamos o factorial do natural anterior e multiplicamos o resultado por n

Terminação de definições recursivas

Para que uma definição recursiva termine é suficiente que:

- tenha (pelo menos) um caso base;
- as chamadas recursivas usem argumentos *estritamente inferiores*

Exemplo: na função de `fact`

- caso base é `n == 0`
- a chamada recursiva tem argumento `n-1`
  (que é menor do que `n`)

Logo: `fact` termina para qualquer `n` maior ou igual a 0.

Observações finais

- Podemos definir o fatorial usando um *ciclo* em vez de recursão:

```
int fact(int n) {
   int r = 1;    // resultado
   for(int i = 1; i<=n; i++)
       r = r*i;
   return r;
}
```

- Para problema simples como o fatorial o ciclo é mais simples e eficiente do que a recursão
- Mais à frente vamos ver problemas em que a solução recursiva é mais simples e eficiente