AULA 1

# RECURSÃO

Recursão é um conceito fundamental em computação. Sua compreensão nos permite construir programas elegantes, curtos e poderosos. Muitas vezes, o equivalente não-recursivo é muito mais difícil de escrever, ler e compreender que a solução recursiva, devido em especial à estrutura recursiva intrínseca do problema. Por outro lado, uma solução recursiva tem como principal desvantagem maior consumo de memória, na maioria das vezes muito maior que uma função não-recursiva equivalente.

Na linguagem C, uma função recursiva é aquela que contém em seu corpo uma ou mais chamadas a si mesma. Naturalmente, uma função deve possuir pelo menos uma chamada proveniente de uma outra função externa. Uma função não-recursiva, em contrapartida, é aquela para qual todas as suas chamadas são externas. Nesta aula construiremos funções recursivas para soluções de problemas.

Este texto é baseado nas referências [2, 7].

## 1.1 Definição

De acordo com [2], em muitos problemas computacionais encontramos a seguinte propriedade: cada entrada do problema contém uma entrada menor do mesmo problema. Dessa forma, dizemos que esses problemas têm uma **estrutura recursiva**. Para resolver um problema como esse, usamos em geral a seguinte estratégia:

se a entrada do problema é pequena então
  resolva-a diretamente;
senão,
  reduza-a a uma entrada menor do mesmo problema,
  aplique este método à entrada menor
  e volte à entrada original.

A aplicação dessa estratégia produz um **algoritmo recursivo**, que implementado na linguagem C torna-se um **programa recursivo** ou um programa que contém uma ou mais **funções recursivas**.

Uma **função recursiva** é aquela que possui, em seu corpo, uma ou mais chamadas a si mesma. Uma chamada de uma função a si mesma é dita uma **chamada recursiva**. Como sabemos, uma função deve possuir ao menos uma chamada proveniente de uma outra função, externa a ela. Se a função só possui chamadas externas a si, então é chamada de função não-recursiva. No semestre anterior, construimos apenas funções não-recursivas.

Em geral, a toda função recursiva corresponde uma outra não-recursiva que executa exatamente a mesma computação. Como alguns problemas possuem estrutura recursiva natural, funções recursivas são facilmente construídas a partir de suas definições. Além disso, a demonstração da correção de um algoritmo recursivo é facilitada pela relação direta entre sua estrutura e a indução matemática. Outras vezes, no entanto, a implementação de um algoritmo recursivo demanda um gasto maior de memória, já que durante seu processo de execução muitas informações devem ser guardadas na sua pilha de execução.

## 1.2 Exemplos

Um exemplo clássico de uma função recursiva é aquela que computa o fatorial de um número inteiro $n \geqslant 0$. A idéia da solução através de uma função recursiva é baseada na fórmula mostrada a seguir:

$$n! = \begin{cases} 1\,, & \text{se } n \leqslant 1\,, \\ n \times (n-1)!\,, & \text{caso contrário}\,. \end{cases}$$

A função recursiva `fat`, que calcula o fatorial de um dado número inteiro não negativo $n$, é mostrada a seguir:

```
/* Recebe um número inteiro n >= 0 e devolve o fatorial de n */
int fat(int n)
{
   int result;

   if (n <= 1)
      result = 1;
   else
      result = n * fat(n-1);

   return result;
}
```

A sentença `return` pode aparecer em qualquer ponto do corpo de uma função e por isso podemos escrever a função `fat` como a seguir:

```
/* Recebe um número inteiro n >= 0 e devolve o fatorial de n */
int fat(int n)
{
   if (n <= 1)
      return 1;
   else
      return n * fat(n-1);
}
```

Geralmente, preferimos a primeira versão implementada acima, onde existe apenas uma sentença com a palavra reservada `return` posicionada no final do corpo da função `fat`. Essa maneira de escrever funções recursivas evita confusão e nos permite seguir o fluxo de execução dessas funções mais naturalmente. No entanto, a segunda versão da função `fat` apresentada acima é equivalente à primeira e isso significa que também é válida. Além disso, essa segunda solução é mais compacta e usa menos memória, já que evita o uso de uma variável. Por tudo isso, essa segunda forma de escrever funções recursivas é muito usada por programadores(as) mais experientes.

A execução da função `fat` se dá da seguinte forma. Imagine que uma chamada `fat(3)` foi realizada. Então, temos ilustrativamente a seguinte situação:

```
fat(3)
┌
  fat(2)
  ┌
    fat(1)
    ┌
      devolve 1
    └
    devolve 2 × 1 = 2 × fat(1)
  └
  devolve 3 × 2 = 3 × fat(2)
└
```

Repare nas indentações que ilustram as chamadas recursivas à função.

Vamos ver um próximo exemplo. Considere o problema de determinar um valor máximo de um vetor $v$ com $n$ elementos. O tamanho de uma entrada do problema é $n \geqslant 1$. Se $n = 1$ então $v[0]$ é o único elemento do vetor e portanto $v[0]$ é máximo. Se $n > 1$ então o valor que procuramos é o maior dentre o máximo do vetor $v[0..n-2]$ e o valor armazenado em $v[n-1]$. Dessa forma, a entrada $v[0..n-1]$ do problema fica reduzida à entrada $v[0..n-2]$. A função `maximo` a seguir implementa essa idéia.

```
/* Recebe um número inteiro n > 0 e um vetor v de números in-
   teiros com n elementos e devolve um elemento máximo de v */
int maximo(int n, int v[MAX])
{
   int aux;

   if (n == 1)
      return v[0];
   else {
      aux = maximo(n-1, v);
      if (aux > v[n-1])
         return aux;
      else
         return v[n-1];
   }
}
```

Segundo P. Feofiloff [2], para verificar que uma função recursiva está correta, devemos seguir o seguinte roteiro, que significa mostrar por indução a correção de um algoritmo ou programa:

**Passo 1:** escreva *o que* a função deve fazer;

**Passo 2:** verifique se a função de fato faz o que deveria fazer quando a entrada é pequena;

**Passo 3:** imagine que a entrada é grande e suponha que a função fará a coisa certa para entradas menores; sob essa hipótese, verifique que a função faz o que dela se espera.

Isso posto, vamos provar então a seguinte propriedade sobre a função `maximo` descrita acima.

**Proposição 1.1.** *A função* `maximo` *encontra um maior elemento em um vetor* $v$ *com* $n \geqslant 1$ *números inteiros.*

*Demonstração.* Vamos provar a afirmação usando indução na quantidade $n$ de elementos do vetor $v$.

Se $n = 1$ o vetor $v$ contém exatamente um elemento, um número inteiro, armazenado em $v[0]$. A função `maximo` devolve, neste caso, o valor armazenado em $v[0]$, que é o maior elemento no vetor $v$.

Suponha que para qualquer valor inteiro positivo $m$ menor que $n$, a chamada externa à função `maximo(m, v)` devolva corretamente o valor de um maior elemento no vetor $v$ contendo $m$ elementos.

Suponha agora que temos um vetor $v$ contendo $n > 1$ números inteiros. Suponha que fazemos a chamada externa `maximo(n, v)`. Como $n > 1$, o programa executa a sentença descrita abaixo:

```
aux = maximo(n-1, v);
```

Então, por hipótese de indução, sabemos que a função `maximo` devolve um maior elemento no vetor $v$ contendo $n - 1$ elementos. Esse elemento, por conta da sentença acima, é armazenado então na variável `aux`. A estrutura condicional que se segue na função `maximo` compara o valor armazenado em `aux` com o valor armazenado em `v[n-1]`, o último elemento do vetor $v$. Um maior valor entre esses dois valores será então devolvido pela função. Dessa forma, a função `maximo` devolve corretamente o valor de um maior elemento em um vetor com $n$ números inteiros. $\square$

## Exercícios

1.1 A $n$-ésima potência de um número $x$, denotada por $x^n$, pode ser computada recursivamente observando a seguinte a fórmula:

$$x^n = \begin{cases} 1\,, & \text{se } n = 0\,,\\ x \cdot x^{n-1}\,, & \text{se } n > 1\,. \end{cases}$$

Considere neste exercício que $x$ e $n$ são números inteiros.

(a) Escreva uma função não-recursiva com a seguinte interface:

```
int pot(int x, int n)
```

que receba dois números inteiros $x$ e $n$ e calcule e devolva $x^n$.

(b) Escreva uma função recursiva com a seguinte interface:

```
int potR(int x, int n)
```

que receba dois números inteiros $x$ e $n$ e calcule e devolva $x^n$.

(c) Escreva um programa que receba dois números inteiros $x$ e $n$, com $n \geqslant 0$, e devolva $x^n$. Use as funções em (a) e (b) para mostrar os dois resultados.

Programa 1.1: Solução do exercício 1.1.

```
#include <stdio.h>

/* Recebe um dois números inteiros x e n e devolve x a n-ésima potência */
int pot(int x, int n)
{
   int i, result;

   result = 1;
   for (i = 1; i <= n; i++)
      result = result * x;
   return result;
}

/* Recebe um dois números inteiros x e n e devolve x a n-ésima potência */
int potR(int x, int n)
{
   if (n == 0)
      return 1;
   else
      return x * potR(x, n-1);
}

/* Recebe dois números inteiros x e n e imprime x a n-ésima potên-
   cia chamando duas funções: uma não-recursiva e uma recursiva */
int main(void)
{
   int x, n;

   scanf("%d%d", &x, &n);
   printf("Não-resursiva: %d^%d = %d\n", x, n, pot(x, n));
   printf("Resursiva    : %d^%d = %d\n", x, n, potR(x, n));

   return 0;
}
```

1.2 O que faz a função abaixo?

```
void imprime_alguma_coisa(int n)
{
   if (n != 0) {
      imprime_alguma_coisa(n / 2);
      printf("%c", '0' + n % 2);
   }
}
```

Escreva um programa para testar a função `imprime_alguma_coisa`.

1.3 (a) Escreva uma função recursiva que receba dois números inteiros positivos e devolva o máximo divisor comum entre eles usando o algoritmo de Euclides.

(b) Escreva um programa que receba dois números inteiros e calcule o máximo divisor comum entre eles. Use a função do item (a).

1.4 (a) Escreva uma função recursiva com a seguinte interface:

```
float soma(int n, float v[MAX])
```

que receba um número inteiro $n > 0$ e um vetor $v$ de números com ponto flutuante com $n$ elementos, e calcule e devolva a soma desses números.

(b) Usando a função do item anterior, escreva um programa que receba um número inteiro $n$, com $n \geqslant 1$, e mais $n$ números reais e calcule a soma desses números.

1.5 (a) Escreva uma função recursiva com a seguinte interface:

```
int soma_digitos(int n)
```

que receba um número inteiro positivo $n$ e devolva a soma de seus dígitos.

(b) Escreva um programa que receba um número inteiro $n$ e imprima a soma de seus dígitos. Use a função do item (a).

1.6 A **seqüência de Fibonacci** é uma seqüência de números inteiros positivos dada pela seguinte fórmula:

$$\begin{cases} F_1 & = & 1 \\ F_2 & = & 1 \\ F_i & = & F_{i-1} + F_{i-2}\,, \qquad \text{para } i \geqslant 3. \end{cases}$$

(a) Escreva uma função recursiva com a seguinte interface:

```
int Fib(int i)
```

que receba um número inteiro positivo $i$ e devolva o $i$-ésimo termo da seqüência de Fibonacci, isto é, $F_i$.

(b) Escreva um programa que receba um número inteiro $i \geqslant 1$ e imprima o termo $F_i$ da seqüência de Fibonacci. Use a função do item (a).

1.7 O **piso** de um número inteiro positivo $x$ é o único inteiro $i$ tal que $i \leqslant x < i + 1$. O piso de $x$ é denotado por $\lfloor x \rfloor$.

Segue uma amostra de valores da função $\lfloor \log_2 n \rfloor$:

| $n$ | 15 | 16 | 31 | 32 | 63 | 64 | 127 | 128 | 255 | 256 | 511 | 512 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\lfloor \log_2 n \rfloor$ | 3 | 4 | 4 | 5 | 5 | 6 | 6 | 7 | 7 | 8 | 8 | 9 |

(a) Escreva uma função recursiva com a seguinte interface:

```
int piso_log2(int n)
```

que receba um número inteiro positivo $n$ e devolva $\lfloor \log_2 n \rfloor$.

(b) Escreva um programa que receba um número inteiro $n \geqslant 1$ e imprima $\lfloor \log_2 n \rfloor$. Use a função do item (a).

1.8 Considere o seguinte processo para gerar uma seqüência de números. Comece com um inteiro $n$. Se $n$ é par, divida por 2. Se $n$ é ímpar, multiplique por 3 e some 1. Repita esse processo com o novo valor de $n$, terminando quando $n = 1$. Por exemplo, a seqüência de números a seguir é gerada para $n = 22$:

$$22 \quad 11 \quad 34 \quad 17 \quad 52 \quad 26 \quad 13 \quad 40 \quad 20 \quad 10 \quad 5 \quad 16 \quad 8 \quad 4 \quad 2 \quad 1$$

É conjecturado que esse processo termina com $n = 1$ para todo inteiro $n > 0$. Para uma entrada $n$, o **comprimento do ciclo de** $n$ é o número de elementos gerados na seqüência. No exemplo acima, o comprimento do ciclo de 22 é 16.

(a) Escreva uma função não-recursiva com a seguinte interface:

```
int ciclo(int n)
```

que receba um número inteiro positivo $n$, mostre a seqüência gerada pelo processo descrito acima na saída e devolva o comprimento do ciclo de $n$.

(b) Escreva uma versão recursiva da função do item (a) com a seguinte interface:

```
int cicloR(int n)
```

que receba um número inteiro positivo $n$, mostre a seqüência gerada pelo processo descrito acima na saída e devolva o comprimento do ciclo de $n$.

(c) Escreva um programa que receba um número inteiro $n \geqslant 1$ e determine a seqüência gerada por esse processo e também o comprimento do ciclo de $n$. Use as funções em (a) e (b) para testar.

1.9 Podemos calcular a potência $x^n$ de uma maneira mais eficiente. Observe primeiro que se $n$ é uma potência de 2 então $x^n$ pode ser computada usando seqüências de quadrados. Por exemplo, $x^4$ é o quadrado de $x^2$ e assim $x^4$ pode ser computado usando somente duas multiplicações ao invés de três. Esta técnica pode ser usada mesmo quando $n$ não é uma potência de 2, usando a seguinte fórmula:

$$x^n = \begin{cases} 1\,, & \text{se } n = 0, \\ (x^{n/2})^2\,, & \text{se } n \text{ é par,} \\ x \cdot x^{n-1}\,, & \text{se } n \text{ é ímpar.} \end{cases} \tag{1.1}$$

(a) Escreva uma função com interface

```
int potencia(int x, int n)
```

que receba dois números inteiros $x$ e $n$ e calcule e devolva $x^n$ usando a fórmula (1.1).

(b) Escreva um programa que receba dois números inteiros $a$ e $b$ e imprima o valor de $a^b$.