

# Algoritmos e Programação

**1º Ano - 1º Semestre**

## ***3. Recursividade***

Escola Superior de Tecnologia e Gestão de Viseu

Agradecimentos a Francisco Morgado, Carlos Simões e Jorge Loureiro

## ***3. Recursividade***

### ***3.1 Conceito***

### ***3.2 Execução de um programa recursivo***

### ***3.3 Objetos locais***

### ***3.4 Análise da eficiência do processo recursivo***

### ***3.5 Recursão direta e indireta***

## 3.1 Conceito

Muitos problemas são descritos através de relações de recorrência.

Diz-se que algo é recursivo quando se define em função de si próprio.

**Em programação, entende-se por recursividade a capacidade que uma linguagem tem de permitir que uma função se invoque a si própria.**

A utilização de um subprograma recursivo permite desencadear um número arbitrariamente grande de repetições de instruções sem, contudo, usar explicitamente estruturas de controlo repetitivo: aplicando a relação de recorrência.

*Os algoritmos recursivos são apropriados para resolver problemas que por natureza são recursivos. Alguns têm soluções recursivas simples, concisas, elegantes e para os quais é difícil desenhar soluções não recursivas com tanta clareza e simplicidade.*

## Exemplo mais comum de recursão: função Fatorial

$$0! = 1$$

Caso base

$$1! = 1 \cdot 0! = 1$$

$$2! = 2 \cdot 1! = 2 \cdot 1$$

$$3! = 3 \cdot 2! = 3 \cdot 2 \cdot 1$$

$$4! = 4 \cdot 3! = 4 \cdot 3 \cdot 2 \cdot 1$$

Regra Geral:

$$n! = n * (n-1)!$$

$$\text{fact}(n) = n * \text{fact}(n-1)$$

### Ex. Fatorial de 4

Diagram illustrating the recursive calculation of  $4!$  (factorial of 4). The diagram shows a sequence of steps:

- $4! = 4 \cdot 3!$
- $3! = 3 \cdot 2!$
- $2! = 2 \cdot 1!$
- $1! = 1 \cdot 0!$
- $0! = 1$  (Caso base)

The final result is  $4! = 24$ .

## *Caso Base ou Condição de Paragem em Recursão*

- ◆ Como uma função recursiva pode chamar-se a si mesma **indefinidamente**, é essencial a existência do caso base, ou condição de paragem.
- ◆ No caso do factorial, o caso base é o zero, cujo valor do Fatorial é 1. A partir dele, são encontrados todos os outros valores.

```
// versão recursiva do programa fatorial
long FatorialRec(int n)
{
    if (n == 0) return 1; // caso base, onde a recursão acaba
    else
        return n * FatorialRec(n - 1);
}
```

A técnica de recursão pode ser usada também na descrição de processos

## Exemplo

Dado um inteiro não negativo, escrever por ordem inverta os dígitos que o representam.

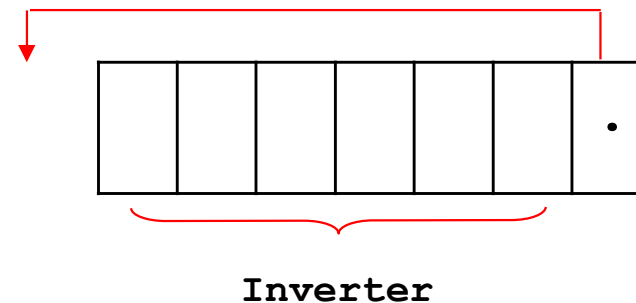
*Princípio:* Dividir o problema em subproblema(s) semelhante(s).

Inverter

*Escrever último dígito;*

*Se sobram dígitos Então*

*inverte-os;*



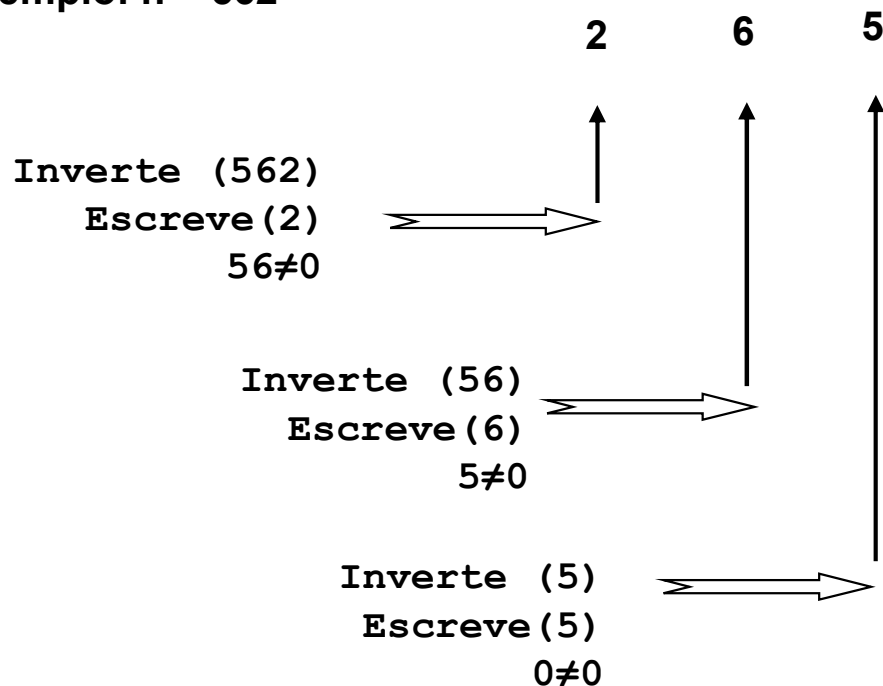
# Recursão Usada na Descrição de Processos

Em C, um subprograma (procedimento ou função) pode chamar não só outro subprograma como ainda chamar-se a si próprio. Tal chamada diz-se RECURSIVA.

Para o problema de inversão dos dígitos de um número  $n$ , tem-se:

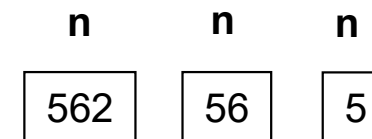
## Simulação

Exemplo:  $n = 562$



```
void inverte(int n)
{
    printf("%d", n % 10);
    if (n / 10 != 0)
        inverte(n / 10);
}
```

## Utilização de memória



A utilização de diferentes registos para as diversas chamadas de um subprograma recursivo torna-o por vezes menos eficiente, em termos do espaço de memória, que o correspondente iterativo.



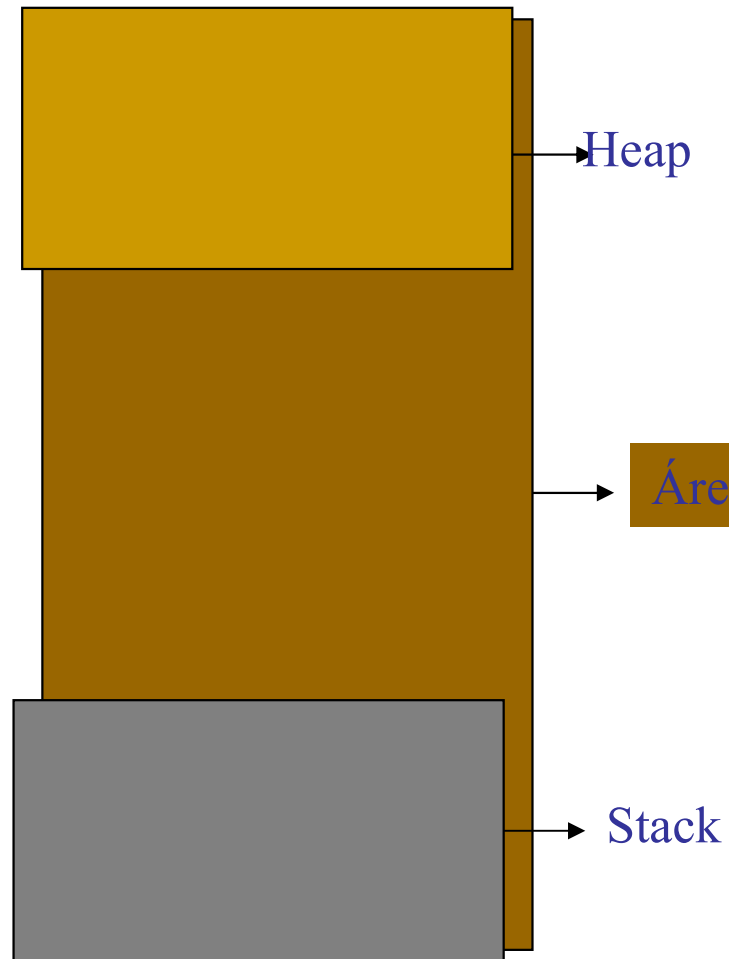
## 3.2 Execução de um programa recursivo

### Como é executado um programa

◆ O sistema operativo aloca para ele uma área na memória do sistema

◆ Internamente, tal segmento de memória é dividido em duas partes:

- ◆ Stack (pilha)
- ◆ Heap



- Área dinâmica da memória do programa
- Responsável pelos objectos manipulados pelo programa
- O seu propósito é suportar a informação (não gerir a execução).
- Qualquer coisa na Heap pode ser acedida em qualquer momento.

- Responsável pelo controlo do que está a ser executado (ou pelo que é chamado).
- Chamada de métodos/funções
- Pense-se no stack como uma série de caixas colocadas umas por cima das outras.

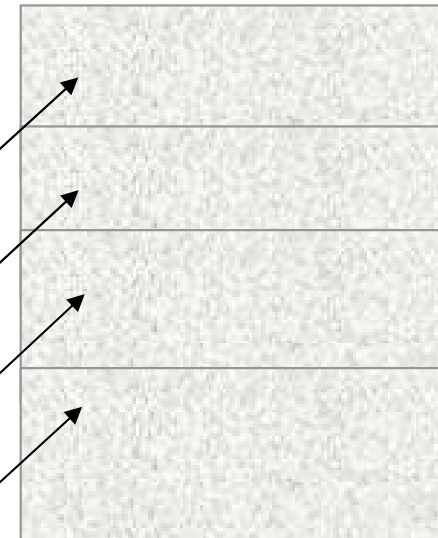
## Chamar funções

- ◆ Quando uma função é chamada, esta é inserida na pilha e executada.
- ◆ Por isso, chamar uma função torna-se mais lento do que escrever o código diretamente.

## Chamar funções recursivamente

- ◆ A cada chamada de uma função recursiva, ela é inserida na pilha e novas variáveis são alocadas.

Stack (pilha)



## 3.3 Objetos Locais

**Exemplo** - Ler uma sequência de caracteres, terminando com um '.', e escrevê-la por ordem inversa.

**Simulação** - Exemplo para a palavra ESTGV

```
Ler ('E')
'E' != '.'
  inverte_sequencia
    Ler ('S')
    'S' != '.'
      inverte_sequencia
        Ler ('T')
        'T' != '.'
          inverte_sequencia
            Ler ('G')
            'G' != '.'
              inverte_sequencia
                Ler ('V')
                'V' != '.'
                  inverte_sequencia
                    Ler ('.')
                    Escrever ('.')
                  Escrever ('V ')
                Escrever ('G ')
              Escrever ('T ')
            Escrever ('S ')
          Escrever ('E ')
        
```

```
void InverterSeq()
{
    char letra;
    scanf("%c",&letra);
    if (letra != '.')
        InverterSeq();
    printf("%c", letra);
}
```

```
[ E [ S [ T [ G [ V [ ' ' ] ] ] ] ]
```

## 3.4 Análise da eficiência do processo recursivo

### Algumas considerações

- Cada vez que o procedimento é chamado, é criado um novo objecto local (variável do tipo char), sem relação com os anteriores, embora com o mesmo identificador.
- Quanto mais profunda for a recursão, mais objectos são criados.
- O seu alcance e vida obedecem às regras conhecidas para variáveis locais.

### Consequentemente ...



*A recursão consome muito espaço*



*Leva a um acréscimo de tempo devido às salvaguardas de contexto*

### VANTAGENS

- ✓ Modo natural e transparente de descrever estruturas ou processos recursivos
- ✓ Dispensa certas variáveis auxiliares

## QUANDO UTILIZAR?

### SEMPRE ... mas

Depois de transformar o processo recursivo no seu correspondente iterativo, quando tal for possível e desejável.

Embora as soluções recursivas sejam mais elegantes,

- ➔ necessitam de mais espaço (os objetos locais devem ser guardados em cada chamada)
- ➔ são mais lentas (devido às operações auxiliares de entrada e saída de um subprograma) do que as não recursivas.

**Exemplo:** Cálculo do termo de ordem  $n$  ( $F_n$ ) de uma sucessão de Fibonacci, sendo

**Fibonacci (0) = 0**

**Fibonacci (1) = 1**

**Fibonacci ( $n$ ) = Fibonacci ( $n-1$ ) + Fibonacci ( $n-2$ )**

0      1      1      2      3      5      8      13      ...

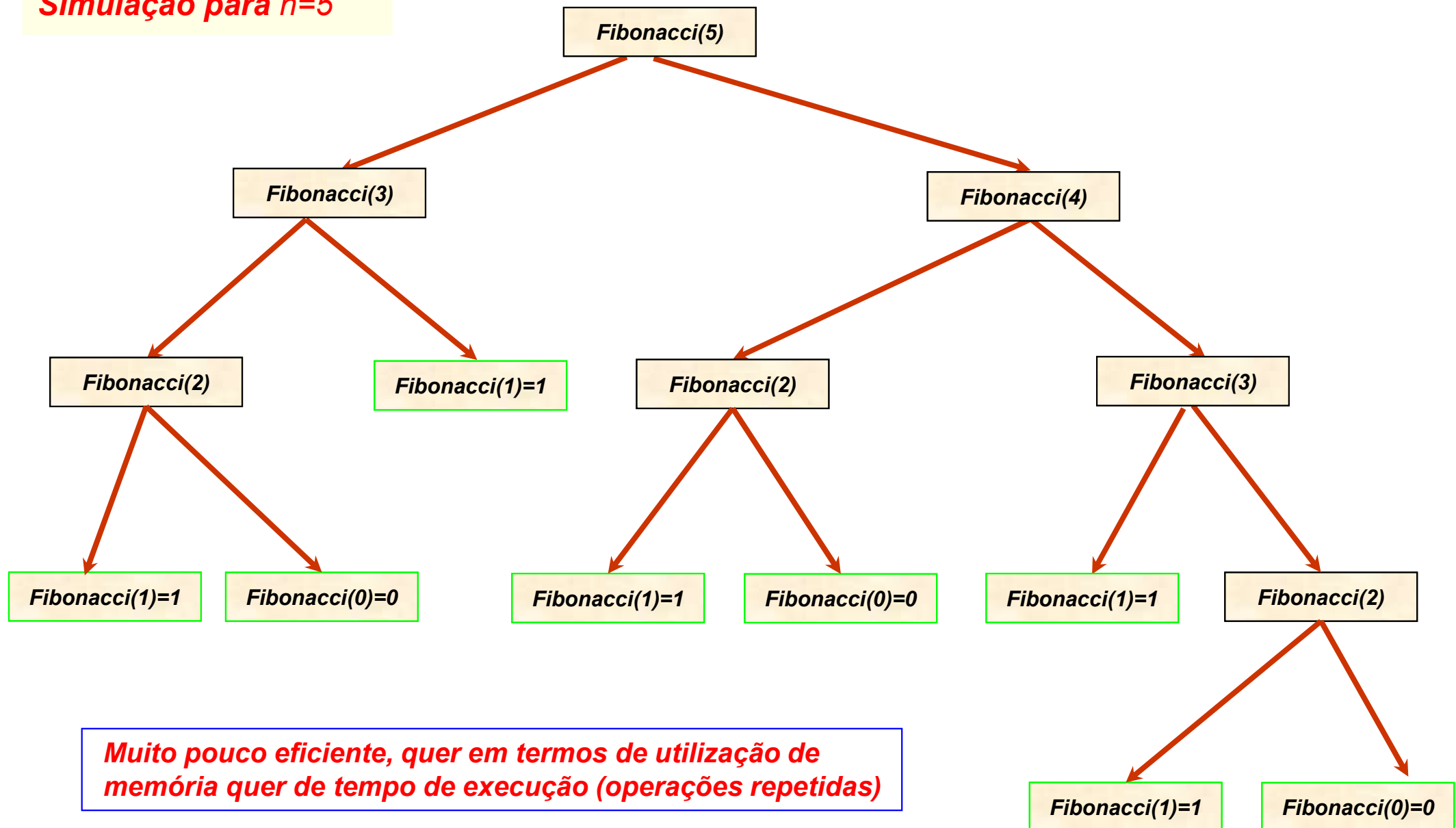
### Solução recursiva

```
int fibonacci (int n)
{
    if (n==0 || n == 1)
        return n;
}
```

```
else return fibonacci(n-1) + fibonacci(n-2);
```

*Fibonacci ( $n$ ) = Fibonacci ( $n-1$ ) + Fibonacci ( $n-2$ )*

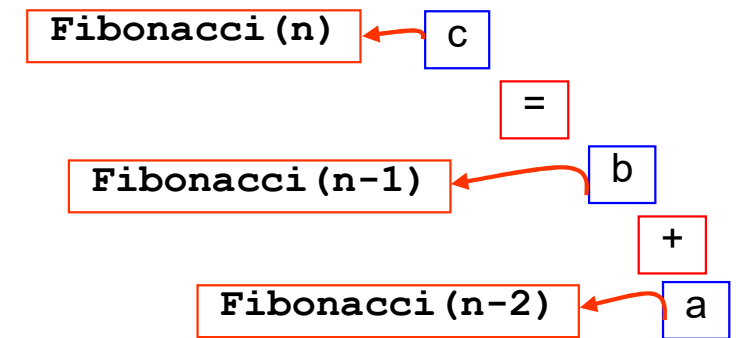
## Simulação para $n=5$



## Solução não recursiva

```
int Fibonacci (int n)
{
    int i, a, b, c;

    if (n==1)
        c = 0;
    else if (n==2)
        c = 1;
    else
    {
        a = 0;
        b = 1;
        for (i=3; i<=n; i++)
        {
            c = a+b;
            // preparar os valores para o próximo ciclo
            a = b;
            b = c;
        }
    }
    return c;
}
```



### Fibonacci (6)

i	a	b	c
	0	1	
3	1	1	1
4	1	2	2
5	2	3	3
6	3	5	5

## 3.5 Recursão direta e indireta

### TIPOS DE RECURSÃO

- Direta: um subprograma chama-se a si próprio
- Indireta: um subprograma chama outro que, por sua vez, chama o primeiro (um chama o outro)

Em C não é possível a chamada de um subprograma antes de ele ter sido declarado

### SOLUÇÃO

```
char B(int x);  
char A(int Y)  
{  
    ... B(i) ...  
}  
char B(int x)  
{  
    ... A(i) ...  
}
```



## Exemplo 1. Cálculo do fatorial versão recursiva (atrás apresentado) e respetiva chamada

```
#include <stdio.h>
int fatorial( int n)
{ if (n==1)
    return 1;
  else
    return n*fatorial(n-1);
}


void main()
{
  int k;
  printf("\nCalculo do fatorial de números lidos. Para terminar, inserir 0 ou negativo\n");
  do
  { printf("\nEscreva um numero inteiro positivo ");
    scanf("%d", &k);
    if (k>0)
      printf("\n%d!= %d",k, fatorial(k));
  }while (k>0);
}
```

## Exemplo 2. Somatório simples

Calcular  $\sum_{i=1}^n i$

```
void main()
{
    int num=0;
    printf("Calcular o Somatório de 1 a: ");
    scanf("%d", &num);
    printf("Solução Recursiva - O somatório de 1 a %d é %d", num, RecursiveSomat(num));
    printf("Solução Iterativa - O somatório de 1 a %d é %d", num, IterativeSomat(num));
}
```

```
// Somatório: versão recursiva
int RecursiveSomat(int n)
{
    if ( n == 1 ) // caso base (n=1)
        return 1;
    else
        return n + RecursiveSomat(n - 1);
}
```



```
// Somatório: versão iterativa
int IterativeSomat(int n)
{
    int soma=0;
    for (int i = 1; i <= n; i++)
    {
        soma = soma + i;
    }
    return soma;
}
```

A recursividade é uma forma de implementar um ciclo através de sucessivas chamadas à mesma função.

### Regras para escrita de funções recursivas

- A primeira instrução de uma função recursiva deve ser a implementação do critério de paragem: a condição ou condições que se devem verificar para que a função deixe de chamar-se a si própria. Estes casos chamam-se **casos base**.
- A chamada recursiva da função deve ser sempre relativa a um subconjunto e só deve vir depois de escrito o critério de paragem.
- A recursividade permite ganhos quanto ao código escrito e, proporciona, por isso, maior legibilidade, mas resulta em perdas (geralmente pouco significativas) ao nível do desempenho.

Implementar, de forma recursiva, a função *strlen* que devolve o número de caracteres existentes numa *string*.

```
#include <stdio.h>
int strlen (char *s)
{
    if (*s == '\0')
        return 0;
    else return (1+ strlen(s+1));
}
```

```
void main()
{
    char texto[50];
    fflush(stdin);
    printf("\nEscreva um pequeno texto:");
    gets(texto);
    printf("\nTexto lido: %s\n", texto);
    printf("\nO texto lido tem %d caracteres\n", strlen(texto));
}
```

**EXERCÍCIO: Determinar o resultado da chamada das funções seguintes.**

```
int somatorioI2N(int i, int n)
{
    if (i==n)
        return n;
    else
        return i+somatorioI2N(i+1, n);
}
```

**printf("\n Resultado da primeira: %d\n", somatorioI2N(1,5));**

```
int downFrom(int i)
{
    if (i==1)
        return 1;
    else
        return i + downFrom(i-1);
}
```

**printf("\n Resultado da segunda: %d\n", downFrom(5));**