

# Introdução à Programação

## Capítulo 6

### Recursividade

**Desenvolvimento para Web e Dispositivos Móveis**

**Joana Fialho**

E-mail: [jfialho@estgv.ipv.pt](mailto:jfialho@estgv.ipv.pt)

**Escola Superior de Tecnologia e Gestão de Viseu**

# Capítulo 6 – Recursividade

## Definição de programa recursivo

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

Recursividade é a propriedade que uma função (ou procedimento) tem de chamar a si própria, diretamente ou não. Trata-se de um processo usado para simplificar problemas dotados de certas características.

### Exemplo mais comum de recursão: função Fatorial

$0! = 1$  ————— Caso base

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

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

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

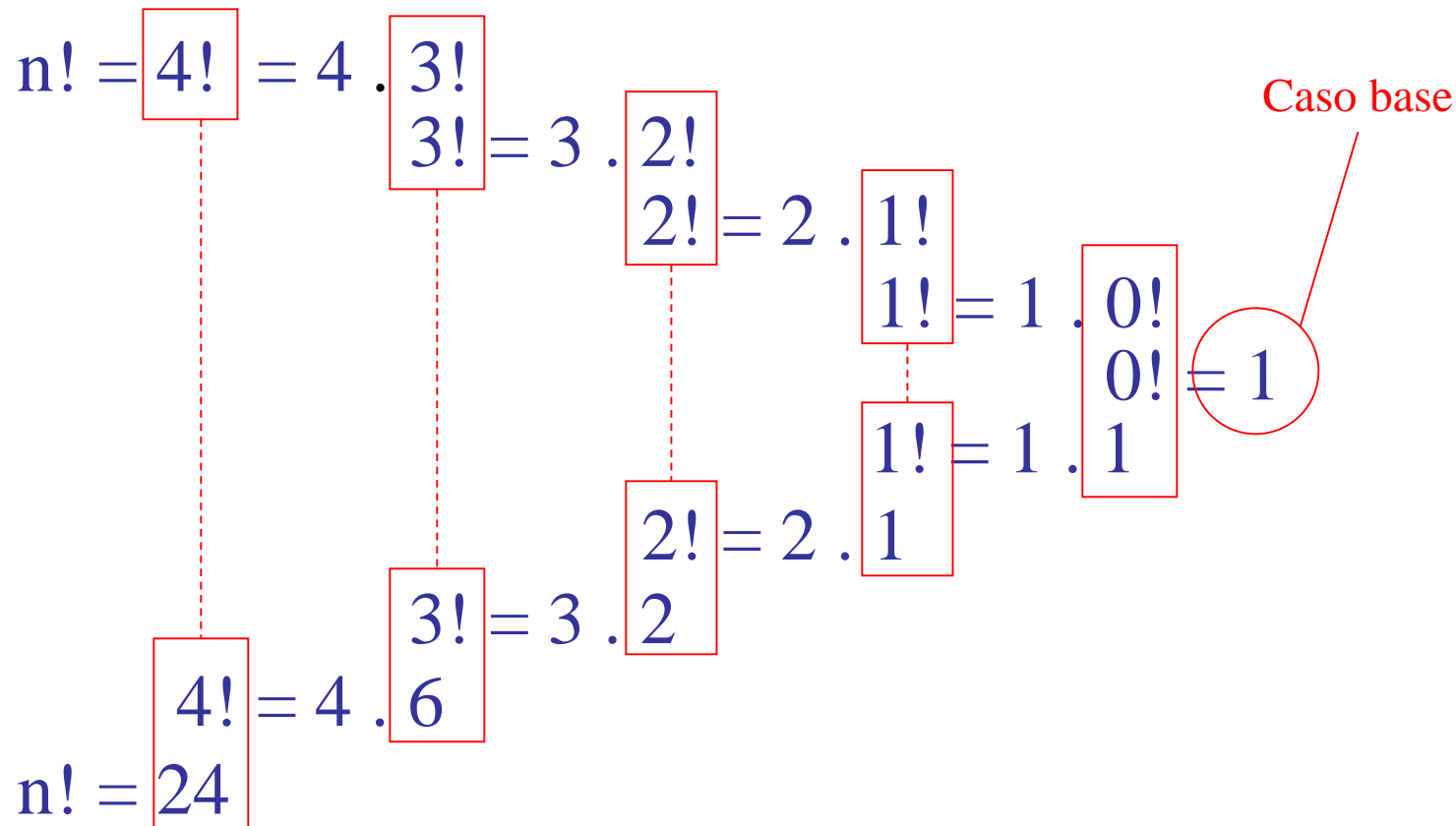
$$4! = 4 \cdot 3! = 4 \cdot 3 \cdot 2 \cdot 1$$

Regra Geral:  
 $n! = n * (n-1)!$   
 $\text{fat}(n) = n * \text{fat}(n-1)$

# Capítulo 6 – Recursividade

## Definição de programa recursivo

### Ex. Fatorial de 4



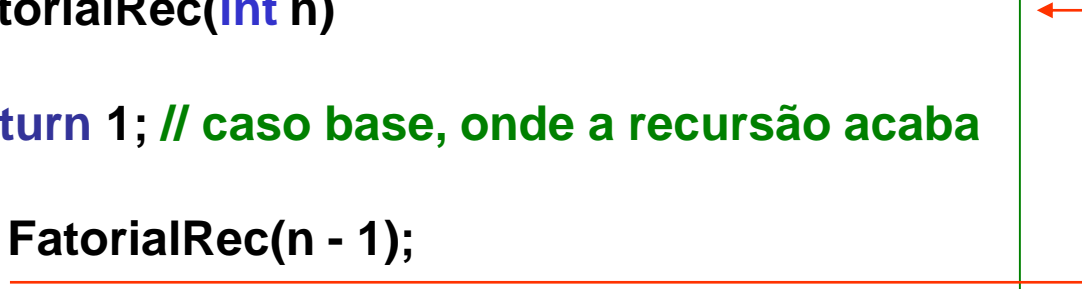
# Capítulo 6 – Recursividade

## Definição de programa recursivo

### 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 fatorial, o caso base é o zero, cujo valor do Fatorial, por definição, é 1. A partir dele, são encontrados todos os outros valores.

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



# Capítulo 6 – Recursividade

## Definição de programa recursivo

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

**Simulação** - Exemplo para a palavra ESTV

```
Ler('E')
'E' ≠ ' '
  inverte_sequencia
    Ler('S')
    'S' ≠ ' '
      inverte_sequencia
        Ler('T')
        'T' ≠ ' '
          inverte_sequencia
            Ler('V')
            'V' ≠ ' '
              inverte_sequencia
                Ler(' ')
                Escrever(' ')
              Escrever('V ')
            Escrever('T ')
          Escrever('S ')
        Escrever('E ')
      Escrever('E ')
```

```
static void InverterSeq()
{
    char letra;
    letra=Char.Parse(Console.ReadLine());
    if (letra != ' ')
        InverterSeq();
    Console.Write(letra);
}
```

E S T V "

# Capítulo 6 – Recursividade

## Vantagens e inconvenientes da recursão

### Algumas considerações

- Cada vez que o procedimento é chamado, é criado um novo objeto local (variável do tipo char), sem relação com os anteriores, embora com o mesmo identificador.
- Quanto mais profunda for a recursão, mais objetos 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, pois são feitas muitas chamadas consecutivas.**



**Erros de implementação podem implicar problemas em tempo de execução pois a memória alocada pelo programa cresce sucessivamente. Ex.: caso onde não seja indicada uma condição de paragem, ou se esta condição nunca for satisfeita.**

### VANTAGENS

- ✓ Modo natural e transparente de descrever estruturas ou processos recursivos, materializado em códigos mais concisos
- ✓ Dispensa certas variáveis auxiliares

# Capítulo 6 – Recursividade

## Vantagens e inconvenientes da recursão

### Quando utilizar a recursividade?

#### SEMPRE ... mas

Quando for possível, é desejável que se transforme o processo recursivo no seu correspondente iterativo.

Embora as soluções recursivas sejam mais elegantes,

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

**Exemplo:** Cálculo do termo de ordem  $n$  ( $F_n$ ) de uma **sucessão de Fibonacci**, sendo  $F_1=0$  e  $F_2=1$ :

0      1      1      2      3      5      8      13      ...

// Definição recursiva do método Fibonacci

```
static long Fibonacci(int n)
```

```
{
```

```
    if ( n == 1)
```

```
        return 0; // caso base 1
```

```
    else if (n == 2 ) // caso base 2
```

```
        return 1;
```

```
    else
```

```
        return Fibonacci( n - 1 ) + Fibonacci( n - 2 );
```

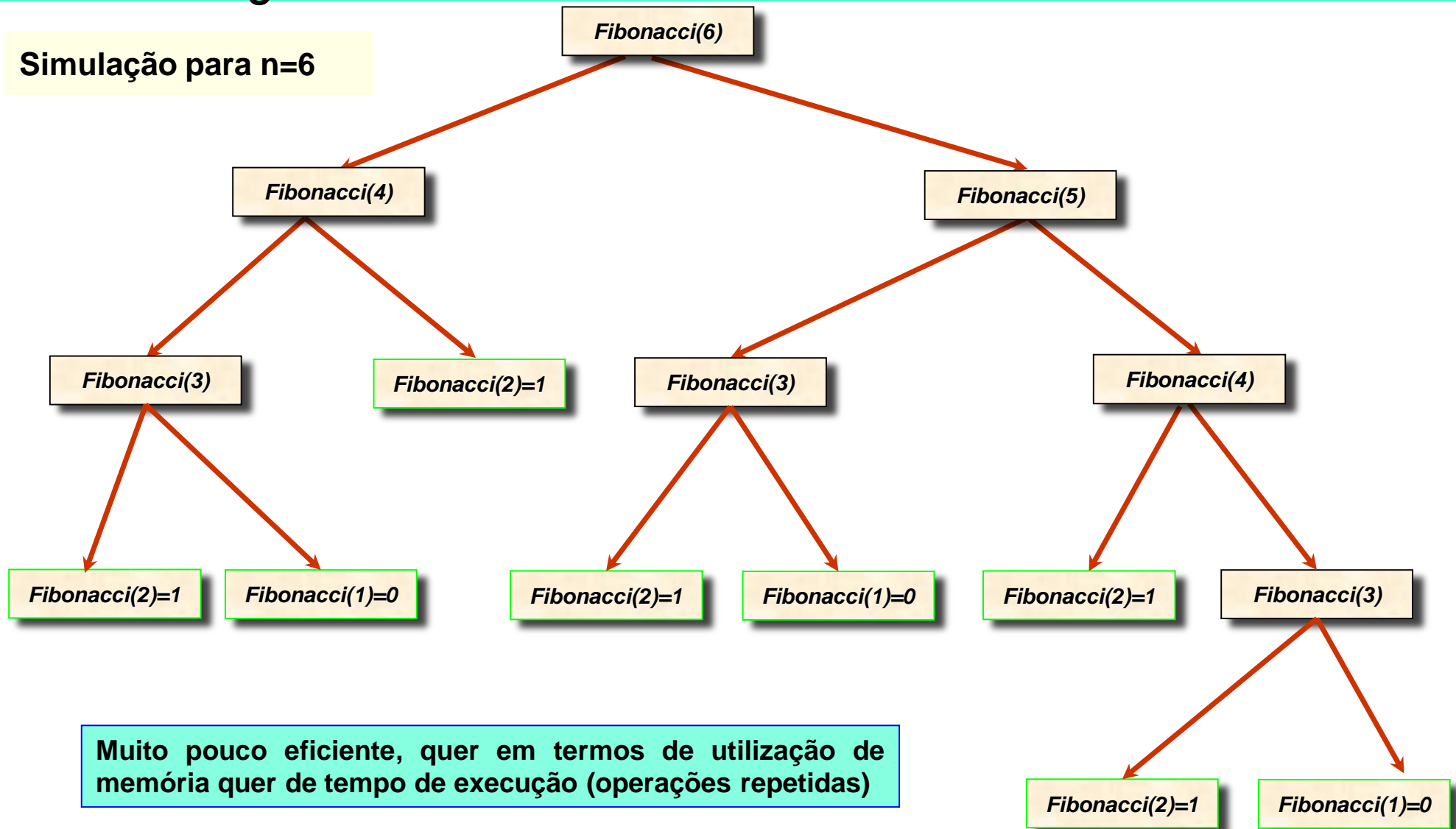
```
}
```

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

# Capítulo 6 – Recursividade

## Vantagens e inconvenientes da recursão

Simulação para n=6

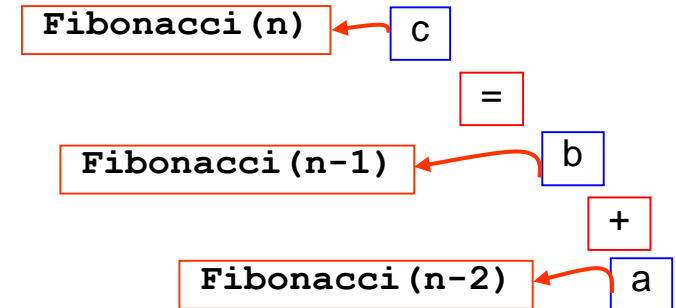




# Capítulo 6 – Recursividade

## Vantagens e inconvenientes da recursão

```
// Definição não recursiva do método Fibonacci
static long Fibonacciterativa(int n)
{
    int a, b, c;
    if (n == 1) // termo 1
        return 0;
    else if (n==2) // termo 2
        return 1;
    else
    {
        a = 0; // n-2
        b = 1; // n-1
        c = 0; // n
        for (int i = 3; i <= n; i++) // gerar os termos n>=3
        {
            c = b+a; // Fibonacci(n)=Fibonacci(n-1)+Fibonacci(n-2)
            a = b; // desloca n-2 para n-1
            b = c; // o n-1 passa a n
        }
        return c; // retorna o termo ordem n
    }
}
```



# Capítulo 6 – Recursividade

## Vantagens e inconvenientes da recursão

Calcular  $\sum_{i=0}^n i$

```
static void Main(string[] args)
{
    int num=0;
    Console.Write("Calcular o Somatório de 1 a: ");
    num = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("Solução Recursiva - O somatório de 1 a {0} é “+
        “{1}”,num,RecursiveSomat(num));
    Console.WriteLine("Solução Iterativa - O somatório de 1 a {0} é “+
        “{1}”,num,IterativeSomat(num));
}
```

// Somatório: versão recursiva

```
static long RecursiveSomat(int n)
{
    if ( n == 1 ) // caso base (n=1)
        return 1;
    else
        return n + RecursiveSomat(n - 1);
}
```

// Somatório: versão iterativa

```
static long IterativeSomat(int n)
{
    int soma=0;
    for (int i = 1; i <= n; i++)
    {
        soma = soma + i;
    }
    return soma;
}
```