

# Recursão

Prof. Paulo Henrique Pisani

outubro/2021

# Tópicos

- Recursividade
- Iteração vs recursão
- Exemplos adicionais

# Recursividade

# Recursão

**re•cur•são**

Ação de recorrer ou de reajustar (linhas, colunas, páginas);  
Se você ainda não entendeu, ver: **recursão**

<https://michaelis.uol.com.br/moderno-portugues/busca/portugues-brasileiro/recursão/>

Slide do Prof. Jesús P. Mena-Chalco:

<http://professor.ufabc.edu.br/~jesus.mena/courses/mcta028-3q-2017/>

# Recursão

- Recursão é um conceito fundamental em computação!
- Muitos problemas computacionais podem ser solucionados **combinando soluções de instâncias menores** desses mesmos problemas.

# Recursão

- Em Matemática, muitas **funções** são definidas de forma recursiva:

$$n! = \begin{cases} 1, & n = 0 \\ (n - 1)! \cdot n, & n > 0 \end{cases}$$

# Recursão (factorial)

- $4! = ?$

$$n! = \begin{cases} 1, & n = 0 \\ (n - 1)! \cdot n, & n > 0 \end{cases}$$

$$4! = 4 \cdot 3!$$


```
graph TD; A["4! = 4 . 3!"] --> B["3! = 3 . 2!"]; B --> C["2! = 2 . 1!"]; C --> D["1! = 1 . 0!"]; D --> E["0! = 1"];
```

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

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

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

$$0! = 1$$

# Recursão (factorial)

- $4! = ?$

$$n! = \begin{cases} 1, & n = 0 \\ (n - 1)! \cdot n, & n > 0 \end{cases}$$

$$4! = 4 \cdot 3!$$


```
graph TD; A["4! = 4 · 3!"] --> B["3! = 3 · 2!"]; B --> C["2! = 2 · 1!"]; C --> D["1! = 1 · 1"]; D --> E["0! = 1"];
```

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

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

$$1! = 1 \cdot 1$$

$$0! = 1$$



# Recursão (factorial)

- $4! = ?$

$$n! = \begin{cases} 1, & n = 0 \\ (n - 1)! \cdot n, & n > 0 \end{cases}$$

$$4! = 4 \cdot 3!$$


```
graph TD; A["4! = 4 · 3!"] --> B["3! = 3 · 2!"]; B --> C["2! = 2 · 1"]; C --> D["1 = 1 · 1"]; D --> E["0! = 1"];
```

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

$$2! = 2 \cdot 1$$

$$1 = 1 \cdot 1$$

$$0! = 1$$

# Recursão (factorial)

- $4! = ?$

$$n! = \begin{cases} 1, & n = 0 \\ (n - 1)! \cdot n, & n > 0 \end{cases}$$

$$4! = 4 \cdot 3!$$


```
graph TD; A["4! = 4 · 3!"] --> B["3! = 3 · 2"]; B --> C["2! = 2 · 1"]; C --> D["1 = 1 · 1"]; D --> E["0! = 1"];
```

$$3! = 3 \cdot 2$$

$$2! = 2 \cdot 1$$

$$1 = 1 \cdot 1$$

$$0! = 1$$

# Recursão (factorial)

- $4! = ?$

$$n! = \begin{cases} 1, & n = 0 \\ (n - 1)! \cdot n, & n > 0 \end{cases}$$

$$4! = 4 \cdot 3!$$

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

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

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

$$0! = 1$$

# Recursão (factorial)

- $4! = ?$

$$n! = \begin{cases} 1, & n = 0 \\ (n - 1)! \cdot n, & n > 0 \end{cases}$$

$$4! = 4 \cdot 3! = 24$$

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

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

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

$$0! = 1$$

# Recursão

- Em C, a recursão ocorre por meio da definição de **funções** também!
- Escreveremos **funções recursivas**, ou seja, funções que chamam a si mesmas.

```
int fatorial(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fatorial(n - 1);  
}
```

# Recursão

- Uma função recursiva deve possuir duas partes básicas:
  - **Caso base**
  - **Chamada recursiva**: pelo menos uma chamada a si mesma

```
int fatorial(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fatorial(n - 1);  
}
```

# Vendo de outra forma...

```
int f4(int n) {  
    if (n == 0)  
        return 1;  
}
```

```
int f3(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * f4(n - 1);  
}
```

```
int f2(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * f3(n - 1);  
}
```

```
int fatorial(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * f2(n - 1);  
}
```

Qual o valor de  
fatorial(3) ?

# Vendo de outra forma...

```
int f4(int n) {
    if (n == 0)
        return 1;
}

int f3(int n) {
    if (n == 0)
        return 1;
    else
        return n * f4(n - 1);
}

int f2(int n) {
    if (n == 0)
        return 1;
    else
        return n * f3(n - 1);
}

int fatorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * f2(n - 1);
}
```

**Mas esse código está muito ruim!**

**Não funciona para  $n > 3$  e também há muito código copiado!**



# Vendo de outra forma...

```
int f4(int n) {  
    if (n == 0)  
        return 1;  
}  
  
int f3(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * f4(n - 1);  
}  
  
int f2(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * f3(n - 1);  
}  
  
int fatorial(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * f2(n - 1);  
}
```



```
int fatorial(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fatorial(n - 1);  
}
```

Esta versão funciona para  $n > 3$

## Qual o valor de fatorial(2) ?

```
int fatorial(int n =2 ) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fatorial(n - 1);  
}
```

## Qual o valor de fatorial(2) ?

```
int fatorial(int n =2 ) {  
    if (n == 0)  
        return 1;  
    else  
        return n *
```

```
int fatorial(int n =1 ) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fatorial(n - 1);  
}
```

```
}
```

## Qual o valor de fatorial(2) ?

```
int fatorial(int n =2 ) {  
    if (n == 0)  
        return 1;  
    else  
        return n *  
            int fatorial(int n =1 ) {  
                if (n == 0)  
                    return 1;  
                else  
                    return n *  
                        int fatorial(int n =0 ) {  
                            if (n == 0)  
                                return 1;  
                            else  
                                return n * fatorial(n - 1);  
                        }  
                    }  
            }  
}
```

## Qual o valor de fatorial(2) ?

```
int fatorial(int n =2 ) {  
    if (n == 0)  
        return 1;  
    else  
        return n *  
            int fatorial(int n =1 ) {  
                if (n == 0)  
                    return 1;  
                else  
                    return n *  
                        int fatorial(int n =0 ) {  
                            if (n == 0)  
                                return 1; ←  
                            else  
                                return n * fatorial(n - 1);  
                        }  
                    }  
            }  
}
```

## Qual o valor de fatorial(2) ?

```
int fatorial(int n =2 ) {  
    if (n == 0)  
        return 1;  
    else  
        return n *   
        int fatorial(int n =1 ) {  
            if (n == 0)  
                return 1;  
            else  
                return n * 1  
        }  
}
```

## Qual o valor de fatorial(2) ?

```
int fatorial(int n =2 ) {  
    if (n == 0)  
        return 1;  
    else  
        return n *  
}  
}
```

```
int fatorial(int n =1 ) {  
    if (n == 0)  
        return 1;  
    else  
        return n * 1  
}
```

## Qual o valor de fatorial(2) ?

```
int fatorial(int n =2 ) {  
    if (n == 0)  
        return 1;  
    else  
        return n * 1  
}
```



## Qual o valor de fatorial(2) ?

```
int fatorial(int n =2 ) {  
    if (n == 0)  
        return 1;  
    else  
        return n * 1 ←  
}
```

Qual o valor de fatorial(2) ?

2

```
int fatorial(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fatorial(n - 1);  
}
```

Veremos a  
recursão de outra  
forma ainda!!!

## Qual o valor de fatorial(2) ?

```
int fatorial(int n =2 ) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fatorial(n - 1);  
}
```

Pilha de chamadas (call stack)

fatorial(2)

## Qual o valor de fatorial(2) ?

```
int fatorial(int n =2 ) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fatorial(n - 1);  
}
```

```
int fatorial(int n =1 ) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fatorial(n - 1);  
}
```

### Pilha de chamadas (call stack)

fatorial(1)

fatorial(2)

# Qual o valor de fatorial(2) ?

```
int fatorial(int n =2 ) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fatorial(n - 1);  
}
```

```
int fatorial(int n =1 ) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fatorial(n - 1);  
}
```

```
int fatorial(int n =0 ) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fatorial(n - 1);  
}
```

## Pilha de chamadas (call stack)

fatorial(0)

fatorial(1)

fatorial(2)

# Qual o valor de fatorial(2) ?

```
int fatorial(int n =2 ) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fatorial(n - 1);  
}
```

```
int fatorial(int n =1 ) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fatorial(n - 1);  
}
```

```
int fatorial(int n =0 ) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fatorial(n - 1);  
}
```

## Pilha de chamadas (call stack)

fatorial(0)

fatorial(1)

fatorial(2)

## Qual o valor de fatorial(2) ?

```
int fatorial(int n =2 ) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fatorial(n - 1);  
}
```

```
int fatorial(int n =1 ) {  
    if (n == 0)  
        return 1;  
    else  
        return n * 1;  
}
```

### Pilha de chamadas (call stack)

fatorial(1)

fatorial(2)



## Qual o valor de fatorial(2) ?

```
int fatorial(int n =2 ) {  
    if (n == 0)  
        return 1;  
    else  
        return n * 1;  
}
```

Pilha de chamadas (call stack)

fatorial(2)

Qual o valor de fatorial(2) ?



2

Pilha de chamadas (call stack)



# Retomando...

- Uma função recursiva deve possuir duas partes básicas:
  - **Caso base**
  - **Chamada recursiva**: pelo menos uma chamada a si mesma

```
int fatorial(int n) {  
    if (n == 0)  
        return 1; Caso base  
    else  
        return n * fatorial(n - 1);  
} Chamada recursiva
```

# Retomando...

É importante ter cuidado  
ao definir o caso base!  
Caso contrário poderá  
ocorrer infinitas chamadas  
recursivas

- Uma função recursiva deve possuir duas partes básicas:
  - **Caso base**
  - **Chamada recursiva**: pelo menos uma chamada a si mesma

```
int fatorial(int n) {  
    if (n == 0)  
        return 1; Caso base  
    else  
        return n * fatorial(n - 1);  
    }  
    Chamada recursiva
```

# Caso base

- É importante ter cuidado ao definir o caso base!
  - Caso contrário, poderá ocorrer infinitas chamadas recursivas!
  - Na verdade, o que ocorre nesse é o chamado estouro da pilha de chamadas: **stack overflow**
  - O controle das chamadas a outros métodos é realizado por meio de uma **pilha**, como vimos no exemplo do fatorial.

# Caso base e chamada recursiva

- Se a instância é pequena:
  - Resolva-a diretamente e retorne o resultado.
- Senão:
  - Reduza a uma instância menor e aplique o mesmo método nesta instância menor.

**As chamadas recursivas devem ir reduzindo a instância do problema de forma a ir em direção a um caso base.**

# Teste A

- O que será impresso neste programa?

```
#include <stdio.h>

void imprimir(int i) {

    if (i > 0) {
        imprimir(i - 1);
        int j;
        for (j = 1; j <= i; j++)
            printf("*");
        printf("\n");
    }
}

int main() {
    imprimir(5);
    return 0;
}
```

## Teste B

```
#include <stdio.h>

void imprime_elemento(int v[], int n) {
    if (n == 0) {
        printf("\n");
        return;
    }
    imprime_elemento(v, n-1);
    printf("%d ", v[n-1]);
}

int main() {
    int vetor[6] = {10, 20, 30, 40, 50, 60};

    imprime_elemento(vetor, 6);

    return 0;
}
```

O que será impresso  
neste programa?




# Há diferença entre as duas funções?

```
void imprime_elemento(int v[], int n) {  
    if (n == 0) {  
        printf("\n");  
        return;  
    }  
    imprime_elemento(v, n-1);  
    printf("%d ", v[n-1]);  
}
```

```
void imprime_elemento2(int v[], int n) {  
    if (n == 0) {  
        printf("\n");  
        return;  
    }  
    printf("%d ", v[n-1]);  
    imprime_elemento2(v, n-1);  
}
```

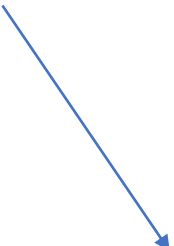
# Há diferença entre as duas funções?

```
void imprime_elemento(int v[], int n) {  
    if (n == 0) {  
        printf("\n");  
        return;  
    }  
    imprime_elemento(v, n-1);  
    printf("%d ", v[n-1]);  
}
```



Inverte a ordem de  
impressão dos  
números do vetor!

```
void imprime_elemento2(int v[], int n) {  
    if (n == 0) {  
        printf("\n");  
        return;  
    }  
    printf("%d ", v[n-1]);  
    imprime_elemento2(v, n-1);  
}
```



# Iteração vs Recursão

# Iteração vs recursão

- Toda função pode ser escrita usando recursão (sem o uso de iteração);
- A recíproca também é verdadeira: toda função pode ser escrita usando iteração (sem o uso de recursão).
- Muitas vezes a implementação recursiva é mais simples, mas é menos eficiente computacionalmente que a versão iterativa.

# Somatório

- Faça uma função para somar os  $n$  primeiros elementos da seguinte série:

$$1^k + 2^k + 3^k + \dots + n^k$$

Assuma que  $k$  é inteiro e  $\geq 0$ .

# Somatório

- Faça uma função para somar os  $n$  primeiros elementos da seguinte série:

$$1^k + 2^k + 3^k + \dots + n^k$$

```
#include<math.h>
```

```
int somatorio(int k, int n) {  
    int soma = 0;  
    int i;  
    for (i = 1; i <= n; i++)  
        soma += lround(pow(i, k));  
    return soma;  
}
```

Versão  
iterativa

# Somatório


- Faça uma função para somar os n primeiros elementos da seguinte série:

$$1^k + 2^k + 3^k + \dots + n^k$$

```
#include<math.h>
```

```
int somatorio(int k, int n) {  
    int soma = 0;  
    int i;  
    for (i = 1; i <= n; i++)  
        soma += lround(pow(i, k));  
    return soma;  
}
```

Observe que usamos o lround para converter o double retornado por pow para int (long int)



# Somatório

- E como seria a forma **recursiva**?

$$1^k + 2^k + 3^k + \cdots + n^k$$

Assuma que  $k$  é inteiro e  $\geq 0$ .



# Somatório

- E como seria a forma **recursiva**?

$$1^k + 2^k + 3^k + \cdots + n^k$$

- Primeiro, vamos tentar formular qual seria o **caso base** e qual a **chamada recursiva**:

$$s(k, n) = \begin{cases} 1, & n = 1 \\ n^k + s(k, n - 1), & n > 1 \end{cases}$$

$$s(k, n) = \begin{cases} 1, & n = 1 \\ n^k + s(k, n - 1), & n > 1 \end{cases}$$

$$s(2, 3) = ?$$

$$s(k, n) = \begin{cases} 1, & n = 1 \\ n^k + s(k, n - 1), & n > 1 \end{cases}$$

$$s(2, 3) = 3^2 + s(2, 2)$$


$$s(2, 2) = 2^2 + s(2, 1)$$

$$s(k, n) = \begin{cases} 1, & n = 1 \\ n^k + s(k, n - 1), & n > 1 \end{cases}$$

$$s(2, 3) = 3^2 + s(2, 2)$$


$$s(2, 2) = 2^2 + s(2, 1)$$


$$s(2, 1) = 1$$

$$s(k, n) = \begin{cases} 1, & n = 1 \\ n^k + s(k, n - 1), & n > 1 \end{cases}$$

$$s(2, 3) = 3^2 + s(2, 2)$$



$$s(2, 2) = 2^2 + 1$$

$$s(k, n) = \begin{cases} 1, & n = 1 \\ n^k + s(k, n - 1), & n > 1 \end{cases}$$

$$s(2, 3) = 3^2 + 5$$

$$s(k, n) = \begin{cases} 1, & n = 1 \\ n^k + s(k, n - 1), & n > 1 \end{cases}$$

$$s(2, 3) = 3^2 + 5 = 14$$

Ótimo, agora vamos escrever a função recursiva em C!

$$s(k, n) = \begin{cases} 1, & n = 1 \\ n^k + s(k, n - 1), & n > 1 \end{cases}$$

?



# Ótimo, agora vamos escrever a função recursiva em C!

$$s(k, n) = \begin{cases} 1, & n = 1 \\ n^k + s(k, n - 1), & n > 1 \end{cases}$$

```
#include<math.h>

int somatorio_recursivo(int k, int n) {
    if (n == 1)
        return 1;
    else
        return lround(pow(n, k)) + somatorio_recursivo(k, n-1);
}
```

## Versão Iterativa

```
#include<math.h>

int somatorio(int k, int n) {
    int soma = 0;
    int i;
    for (i = 1; i <= n; i++)
        soma += lround(pow(i, k));
    return soma;
}
```

## Versão Recursiva

```
#include<math.h>

int somatorio_recursivo(int k, int n) {
    if (n == 1)
        return 1;
    else
        return lround(pow(n, k)) + somatorio_recursivo(k, n-1);
}
```

## Versão Iterativa

```
#include<math.h>

int somatorio(int k, int n) {
    int soma = 0;
    int i;
    for (i = 1; i <= n; i++)
        soma += lround(pow(i, k));
    return soma;
}
```

Para pensar: veja que na versão iterativa, o índice **aumenta** enquanto que na versão recursiva ele **diminui**.

Mesmo assim, na prática, a soma é feita na ordem do menor n (n=1) até o maior nos dois casos.

## Versão Recursiva

```
#include<math.h>

int somatorio_recursivo(int k, int n) {
    if (n == 1)
        return 1;
    else
        return lround(pow(n, k)) + somatorio_recursivo(k, n-1);
}
```

# Exemplo

- Como seria uma função recursiva para calcular o valor de  $x^n$ ? Assuma que  $n$  é inteiro e  $\geq 0$ .

# Exemplo

- Como seria uma função recursiva para calcular o valor de  $x^n$ ? Assuma que  $n$  é inteiro e  $\geq 0$ .

```
int potencia(int b, int e) {  
    if (e == 0) return 1;  
    return b * potencia(b, e-1);  
}
```

**Exemplos adicionais**

# Outro exemplo: Fibonacci

- A série de Fibonacci é definida da seguinte forma:

$$fib(i) = \begin{cases} 0, & i = 0 \\ 1, & i = 1 \\ fib(i - 1) + fib(i - 2), & i > 1 \end{cases}$$



0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

# Outro exemplo: Fibonacci

- A série de Fibonacci é definida da seguinte forma:

Casos base

$$fib(i) = \begin{cases} 0, & i = 0 \\ 1, & i = 1 \\ fib(i - 1) + fib(i - 2), & i > 1 \end{cases}$$

Chamadas recursivas



0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...



# Versão recursiva

Casos base

$$fib(i) = \begin{cases} 0, & i = 0 \\ 1, & i = 1 \\ fib(i - 1) + fib(i - 2), & i > 1 \end{cases}$$

Chamadas recursivas

```
int numero_fibonacci(int i) {  
    if (i == 0)  
        return 0;  
    else if (i == 1)  
        return 1;  
    else  
        return numero_fibonacci(i - 1) + numero_fibonacci(i - 2);  
}
```

# Versão **iterativa**

$$fib(i) = \begin{cases} 0, & i = 0 \\ 1, & i = 1 \\ fib(i - 1) + fib(i - 2), & i > 1 \end{cases}$$



Qual versão é mais eficiente? A iterativa ou a recursiva?

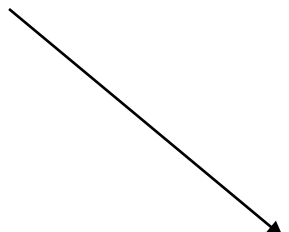
# Mais um exemplo: somar elementos de um vetor

Essa a versão iterativa!



```
#include<stdio.h>
```

```
double soma_vetor(double vetor[], int n) {  
    double soma = 0.0;  
    int i;  
    for (i = 0; i < n; i++)  
        soma += vetor[i];  
    return soma;  
}
```



```
int main() {  
    double v[] = {1, 2, 3, 4.5, 5.5};  
    printf("%.2lf\n", soma_vetor(v, 5));  
  
    return 0;  
}
```

**Como seria a  
versão recursiva?**

# Mais um exemplo: somar elementos de um vetor

Essa a versão recursiva!

```
#include<stdio.h>
```

```
double soma_vetor_rec(double vetor[], int n) {  
    if (n == 0)  
        return 0;  
    else  
        return vetor[n-1] + soma_vetor_rec(vetor, n-1);  
}
```

```
int main() {  
    double v[] = {1, 2, 3, 4.5, 5.5};  
    printf("%.2lf\n", soma_vetor_rec(v, 5));  
  
    return 0;  
}
```

# Exemplo

1. Como seria uma função recursiva para verificar se um número  $x$  está presente no vetor?
2. E para encontrar o menor valor em um vetor?

# Exemplo (1)

Comprimento  
do vetor

Número a ser  
localizado

```
int encontra_numero(int v[], int n, int numero) {  
    if (n==0) return 0;  
    if (v[n-1] == numero)  
        return 1;  
    else  
        return encontra_numero(v, n-1, numero);  
}
```



Comprimento  
do vetor

Número a ser  
localizado

```
int encontra_numero(int v[], int n, int numero) {  
    if (n==0) return 0;  
    return (v[n-1] == numero) || encontra_numero(v, n-1, numero);  
}
```

# Exemplo (2)

Comprimento  
do vetor



```
int menor_numero(int v[], int n) {  
    if (n==1)  
        return v[0];  
    else {  
        int menor = menor_numero(v, n-1);  
        int atual = v[n-1];  
        return (atual < menor ? atual : menor);  
    }  
}
```

# Exercícios



# Exercício 1

- Escreva uma função recursiva que retorne a soma dos  $n$  primeiros números ímpares. Por exemplo:  $\text{soma\_impares}(3) = 1 + 3 + 5 = 9$ .


```
int soma_impares(int n)
```

# Exercício 2

- Faça uma função recursiva que calcule o valor de PI usando a série Gregory:

$$\frac{\pi}{4} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots$$

**double calcula\_pi(int n)**



n é o número de elementos a considerar da série de Gregory

# Exercício 2

- Faça uma função recursiva que calcule o valor de  $\pi$  usando a série Gregory:

```
double calcula_pi(int n) {  
    if (n == 1)  
        return 4.0;  
    else  
        return 4*(n%2 == 0 ? -1 : 1) * (1.0/(2.0*n-1)) + calcula_pi(n-1);  
}
```

# Exercício 3

- *O professor ABC escreveu uma função, mas esqueceu para que ela servia... Você pode ajuda-lo a descobrir a função **misterio** faz?*
- Faça também uma versão iterativa dela.
- Há risco dessa função recursiva executar indefinidamente?

```
#include<stdio.h>
```

```
int misterio(char t[], int c) {  
    if (t[c] == '\0')  
        return 0;  
    else  
        return 1 + misterio(t, c + 1);  
}
```

```
int main() {  
    char vetor[50];  
  
    ...  
  
    printf("%d\n", misterio(vetor, 0));  
  
    return 0;  
}
```

# Referências

- Slides do Prof. Jesús P. Mena-Chalco:
  - <http://professor.ufabc.edu.br/~jesus.mena/courses/mcta028-3q-2017/>
- Slides do Prof. Fabrício Olivetti:
  - <http://folivetti.github.io/courses/ProgramacaoEstruturada/>
- Slide do Prof. Monael Pinheiro Ribeiro:
  - <https://sites.google.com/site/aed2018q1/>
- CELES, W.; CERQUEIRA, R.; RANGEL, J. L. Introdução a Estruturas de Dados. Elsevier/Campus, 2004.

# Bibliografia básica

- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: teoria e prática. 2. ed. Rio de Janeiro, RJ: Campus, 2002.
- FORBELLONE, A. L. V.; EBERSPACHER, H. F. Lógica de programação: a construção de algoritmos e estruturas de dados. 3. ed. São Paulo, SP: Prentice Hall, 2005.
- PINHEIRO, F. A. C. Elementos de programação em C. Porto Alegre, RS: Bookman, 2012.

# Bibliografia complementar

- AGUILAR, L. J. Programação em C++: algoritmos, estruturas de dados e objetos. São Paulo, SP: McGraw-Hill, 2008.
- DROZDEK, A. Estrutura de dados e algoritmos em C++. São Paulo, SP: Cengage Learning, 2009.
- KNUTH D. E. The art of computer programming. Upper Saddle River, USA: Addison- Wesley, 2005.
- SEDGEWICK, R. Algorithms in C++: parts 1-4: fundamentals, data structures, sorting, searching. Reading, USA: Addison-Wesley, 1998.
- SZWARCFITER, J. L.; MARKENZON, L. Estruturas de dados e seus algoritmos. 3. ed. Rio de Janeiro, RJ: LTC, 1994.
- TENENBAUM, A. M.; LANGSAM, Y.; AUGENSTEIN, M. J. Estruturas de dados usando C. São Paulo, SP: Pearson Makron Books, 1995.