

# CP63B-DPGR3A

# COMPUTAÇÃO 2

APNP 07 - Recursividade

Prof. Rafael Gomes Mantovani

# Licença

Este trabalho está licenciado com uma Licença CC BY-NC-ND 4.0:



maiores informações:

[https://creativecommons.org/licenses/by-nc-nd/4.0/deed.pt\\_BR](https://creativecommons.org/licenses/by-nc-nd/4.0/deed.pt_BR)

# Roteiro



- 1** Introdução
- 2** Recursão
- 3** Exemplo
- 4** Exercícios
- 5** Referências

# Roteiro

- 1** Introdução
- 2** Recursão
- 3** Exemplo
- 4** Exercícios
- 5** Referências

# Introdução

- O que é uma função **recursiva**?

# Introdução

- função é dita **recursiva** quando dentro do seu código existe uma **chamada para si mesma**.

```
1  int fatorial(int n){  
2      int fat;  
3      if(n <= 1)  
4          return 1;  
5      else{  
6          fat = n * fatorial(n-1);  
7          return fat;  
8      }  
9  }  
10  
11 int main(){  
12     int n=3, resultado;  
13     resultado = fatorial(n);  
14     printf("%d", resultado);  
15  
16     return 0;  
17 }
```

# Introdução

- função é dita **recursiva** quando dentro do seu código existe uma **chamada para si mesma**.

```
1  int fatorial(int n){  
2      int fat;  
3      if(n <= 1)
```

Mas ...

- Não pode entrar em loop?
- Isso pode ser vantajoso?
- **Vamos entender melhor.**

```
12      int n=3, resultado;  
13      resultado = fatorial(n);  
14      printf("%d", resultado);  
15  
16      return 0;  
17  }
```

# Roteiro



- 1 Introdução
- 2 Recursão
- 3 Exemplo
- 4 Exercícios
- 5 Referências



# Recursão

## □ Anteriormente ...

- Problemas resolvidos de maneira iterativa (por laços de repetição)

```
void exhibeNumeros(int valor){  
    for(int i = valor ; i >= 1; i--){  
        printf("Valor de i: %d", i);  
    }//for  
    return;  
}//exibNumeros
```

- Podemos resolver de outra maneira?

# Recursão

- Podemos pensar que para exibir os números de 5 até 1 é possível dividir o problema em problemas (partes) menores, da seguinte maneira:

Exibir o **5** e depois exibir os números de **4 até 1**

Exibir o **4** e depois exibir os números de **3 até 1**

Exibir o **3** e depois exibir os números de **2 até 1**

Exibir o **2** e depois exibir os números de **1 até 1**

Exibir o **1**

# Recursão

## □ Como ficaria o código

```
void exibeNumeros(int valor){  
    if(valor == 1){ //Caso base  
        printf("Valor de i: %d\n", valor);  
    } else { //Caso recursivo  
        printf("Valor de i: %d\n", valor);  
        exibeNumeros(valor - 1); //Chamada recursiva  
    } //else  
    return;  
} //exibeNumeros
```

## □ Partes principais

- **caso base**: caso em que não há uma nova chamada recursiva e garante que a função termine
- **caso recursivo**: chama a função recursivamente mudando os parâmetros para que se aproxime do resultado

# Recursão

- **recursão** é uma técnica que define um problema em termos de um ou mais versões menores deste mesmo problema;
- portanto, pode ser utilizada sempre que for possível expressão a solução de um problema em função do próprio problema.

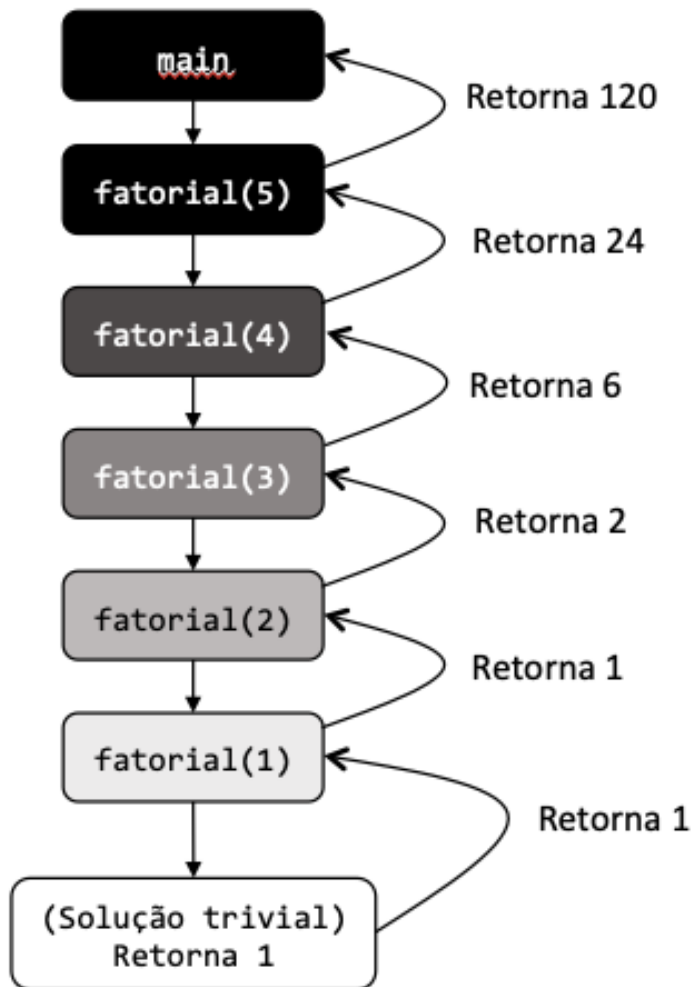


# Recursão

```
1  int fatorial(int n){
2      if(n == 0)
3          return 1;
4      else if(n < 0){
5          exit(0);
6      }
7      return n * fatorial(n-1);
8  }
9
10 int main(){
11     int n=3, resultado;
12     resultado = fatorial(n);
13     printf("%d", resultado);
14
15     return 0;
16 }
```

```
fatorial(5)
=> return 5 * fatorial(4)
=> return 4 * fatorial(3)
    => return 3 * fatorial(2)
        => return 2 * fatorial(1)
            => return 1 * fatorial(0)
                => 0 == 0
                    <= return 1
                        <= return 1 * 1 → (1)
                            <= return 2 * 1 → (2)
                                <= return 3 * 2 → (6)
                                    <= return 4 * 6 → (24)
                                        <= return 5 * 24 → (120)
```

# Recursão



```
fatorial(5)
=> return 5 * fatorial(4)
=> return 4 * fatorial(3)
=> return 3 * fatorial(2)
=> return 2 * fatorial(1)
=> return 1 * fatorial(0)
=> 0 == 0
<= return 1
<= return 1 * 1 → (1)
<= return 2 * 1 → (2)
<= return 3 * 2 → (6)
<= return 4 * 6 → (24)
<= return 5 * 24 → (120)
```

# Recursão

- Em uma função recursiva pode ocorrer um problema de terminação do programa, como um *loop* infinito;
- Para determinar a terminação das repetições, deve-se:
  - Definir uma função que implica em uma condição de terminação (solução trivial);
  - Provar que a função decresce a cada iteração, permitindo que, eventualmente, esta solução trivial seja atingida;

Um exemplo de recursividade. Dada uma função que recebe um parâmetro **par**:

funcao(**par**)

- Teste de término de recursão utilizando **par**
  - Se teste for verdadeiro, retornar a solução final
- Processamento
  - Aqui a função processa as informações baseado em **par**
- Chamada recursiva utilizando **par**
  - **par** deve ser modificado para fazer a recursão chegar a um término

# Vantagens e Desvantagens

## □ Vantagens

- A solução recursiva é mais elegante e menor que a sua versão iterativa
- Exibe com maior clareza a solução do problema
- Elimina a necessidade de controle manual de diversas variáveis comuns em processos iterativos

## □ Desvantagens

- Tende a exigir mais espaço de memória
- Pode ser mais lento pois vai enchendo a pilha com várias chamadas de função
- Pode ser mais difícil de depurar (e dar manutenção)



# Roteiro



- 1 Introdução
- 2 Recursão
- 3 Exemplo
- 4 Exercícios
- 5 Referências

# Exemplo: Soma

- uma função recursiva pode retornar valores e esses valores acumulados para compor um único resultado

```
int soma(int valor){  
    if(valor == 1){ //Caso base  
        return 1;  
    } else { //Caso recursivo  
        return valor + somaNumeros(valor - 1);  
    } //else  
} //soma
```



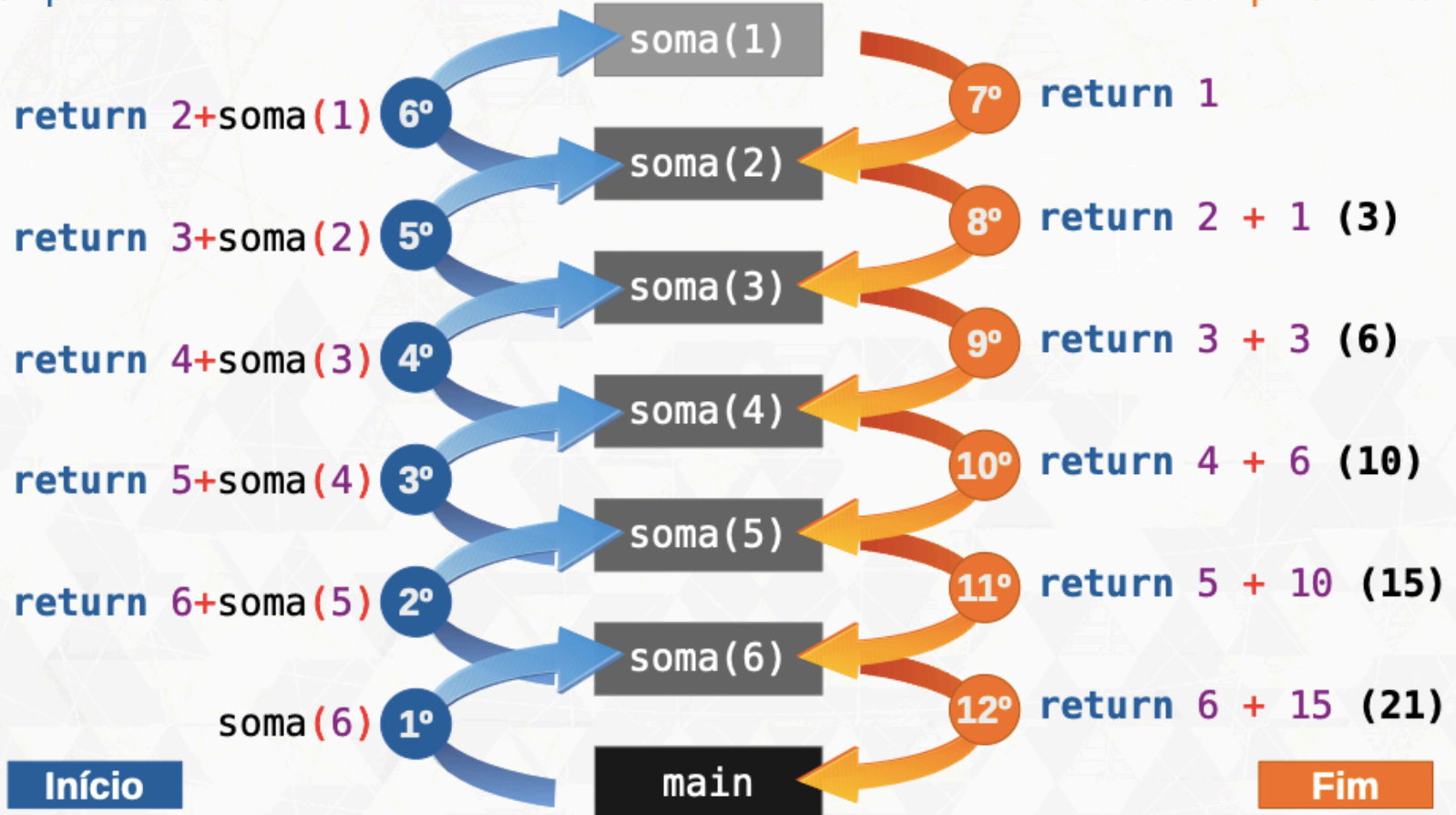
Quando a última coisa feita em uma função é a chamada recursiva, ela usa menos memória e é chamada de **recursão de cauda**

# Exemplo: Soma

empilhamento

caso base

desempilhamento



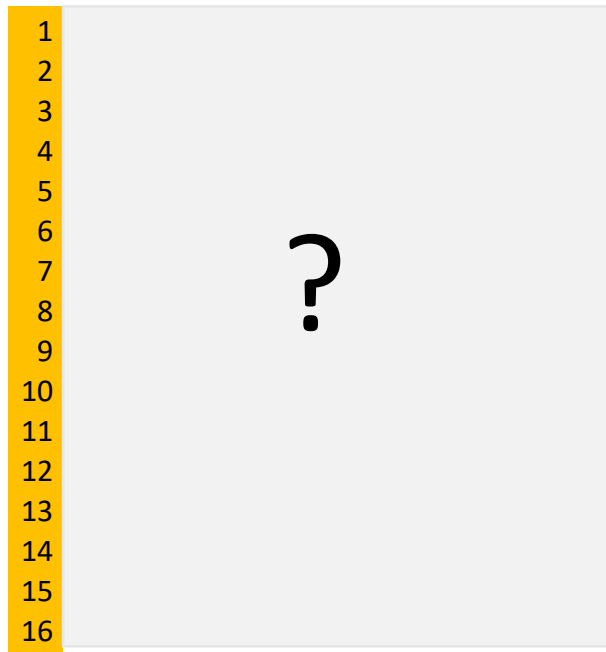
# Exemplo: Fibonacci

- Exemplo: Fibonacci

$$S(N) = \begin{cases} 1, & \text{se } N = 1 \\ 1, & \text{se } N = 2 \\ \text{fib}(N-1) + \text{fib}(N-2), & \text{se } N > 2 \end{cases}$$

$\rightarrow$  Soluções triviais

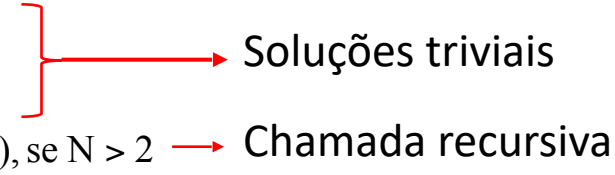
$\rightarrow$  Chamada recursiva



# Exemplo: Fibonacci

- Exemplo: Fibonacci

$$S(N) = \begin{cases} 1, \text{ se } N = 1 \\ 1, \text{ se } N = 2 \\ \text{fib}(N-1) + \text{fib}(N-2), \text{ se } N > 2 \end{cases}$$



Soluções triviais

Chamada recursiva

```
1 int fib(int n){
2     if(n == 1)
3         return 1;
4     if(n == 2)
5         return 1;
6     return fib(n-1)+fib(n-2);
7 }
8
9 int main(){
10     int n;
11
12     printf("Forneca o n: ");
13     scanf("%d", &n);
14     printf("Resultado: %d", fib(n));
15     return 0;
16 }
```

# Vantagens e Desvantagens

## □ Vantagens

- A solução recursiva é mais elegante e menor que a sua versão iterativa
- Exibe com maior clareza a solução do problema
- Elimina a necessidade de controle manual de diversas variáveis comuns em processos iterativos

## □ Desvantagens

- Tende a exigir mais espaço de memória
- Pode ser mais lento pois vai enchendo a pilha com várias chamadas de função
- Pode ser mais difícil de depurar (e dar manutenção)

# Roteiro



- 1** Introdução
- 2** Recursão
- 3** Exemplo
- 4** Exercícios
- 5** Referências

# Exercícios

Universidade Tecnológica Federal do Paraná – Câmpus Apucarana  
Computação 2 (CP63B) - DPGR3A – Recursividade  
Prof. Dr. Rafael Gomes Mantovani

---

## Instruções:

- Antes de codificar, esboce em um papel a sequência de passos necessários para criar o seu programa. Isso ajuda a programar a solução;
- Crie um arquivo .c para cada um dos exercícios. Por exemplo, na resolução do exercício 01, crie um arquivo chamado 'ex01.c'.
- em todos os exercícios faça uma função `main` para testar sua função.


## Exercícios sobre Recursividade

**Exercício 1.** Crie uma função que retorne  $x * y$  através de operação de soma. A função recebe  $x$  e  $y$  por parâmetro.

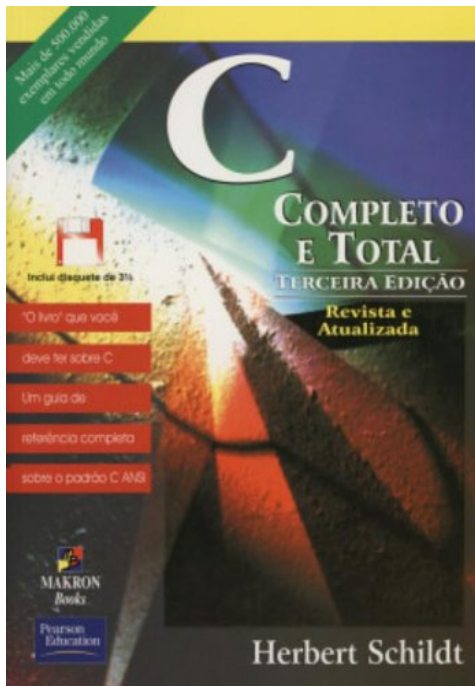


# Roteiro



- 1** Introdução
  - 2** Recursão
  - 3** Exemplo
  - 4** Exercícios
  - 5** Referências
- 

# Referências



[Schildt, 1997]



[de Souza et al, 2011]

# Referências



- [Schildt, 1997] SCHILDT, H. **C Completo e Total**. 3. ed. São Paulo: Pearson, 1997.
- [de Souza et al, 2011] DE SOUZA, M. A. F. et al. **Algoritmos e lógica de programação**. 2. ed. São Paulo: Cengage Learning, 2011.

# Perguntas?

Prof. Rafael G. **Mantovani**

[rafaelmantovani@utfpr.edu.br](mailto:rafaelmantovani@utfpr.edu.br)