

# Engenharia de Computação

# **Fundamentos de Programação**

## **Aula 18 – Recursividade**

**Prof. Muriel de Souza Godoi**  
[muriel@utfpr.edu.br](mailto:muriel@utfpr.edu.br)

# Recursividade

- Um processo que é definido **a partir de si próprio**.

- **Exemplo:** Como **esvaziar um vaso** com 3 flores?



- Retirar uma flor e **esvaziar um vaso** com 2 flores



# Recursividade

- Mas como **esvaziar um vaso** com 2 flores?



- Retirar uma flor e **esvaziar um vaso** com 1 flor



# Recursividade

- Mas como **esvaziar um vaso** com 1 flor?



- Retirar uma flor e **esvaziar um vaso** com 0 flores





# Recursividade

- Mas como **esvaziar um vaso** com 0 flores?
  - Por definição um vaso com 0 flores já está vazio
  - Chamamos de **CASO BASE DA RECURSÃO**



- **Generalizando a solução recursiva:**

- Para **esvaziar um vaso** com N flores:
- SE vaso está vazio:
  - Não há nada a ser feito – **Caso base**
- Caso contrário
  - Retirar uma flor
  - **Esvaziar um vaso** com N-1 flores

# Recursividade

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

```
void recursao(int valor){  
    ...  
    recursao(valor - 1);  
    ...  
    return;  
} //recursao
```

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



# Anteriormente...

- Vimos problemas sendo resolvidos de maneira **iterativa** (por laços de repetição)

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

- Podemos resolver de outra maneira?  
sem repetição?

# Recursividade

- Podemos resolver de outra maneira sem repetição?
- Podemos pensar que para exibir os números de 5 até 1 é possível dividir o problema em partes menores, da seguinte maneira:

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

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

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

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

Exibir o **1**



# Solução recursiva

- Como ficaria no código?

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

- Quais são as partes principais de uma função recursiva?
  - **Caso base:** Caso em que não há nova chamada recursiva e garante que a função termine
  - **Chamada recursiva:** Chama função recursivamente mudando os parâmetros para que se aproximem do resultado

# • 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 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 as chamadas
- Pode ser mais **difícil para depurar**

# Simulando...

- O algoritmo **recursivo** abaixo exibe todos os números de **n até 1** e, em seguida, todos os números de **1 até n**

```
void recursao(int a) {
    if (a > 0) {
        printf("%i ", a);
        recursao(a-1); // Chamada recursiva
        printf("%i ", a);
    } else { // Trata caso base
        printf("\n");
    } //else
} //recursao

int main() {
    recursao(4);
    return 0;
} // main
```

# Simulando empilhamento...

## Caminho Ida

1º 

```
if (a > 0) {  
    printf("%i ", a);  
    recursao(a-1);  
    printf("%i ", a);  
}
```

a=4




4

# Simulando empilhamento...

## Caminho Ida

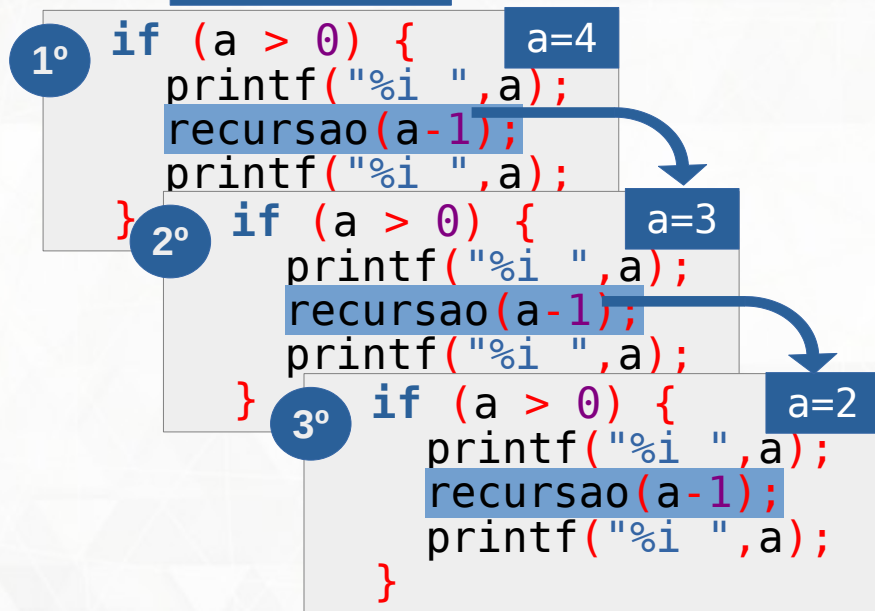
```
1º if (a > 0) { a=4
    printf("%i ", a);
    recursao(a-1);
    printf("%i ", a);
} 2º if (a > 0) { a=3
    printf("%i ", a);
    recursao(a-1);
    printf("%i ", a);
}
```





# Simulando empilhamento...

## Caminho Ida



4



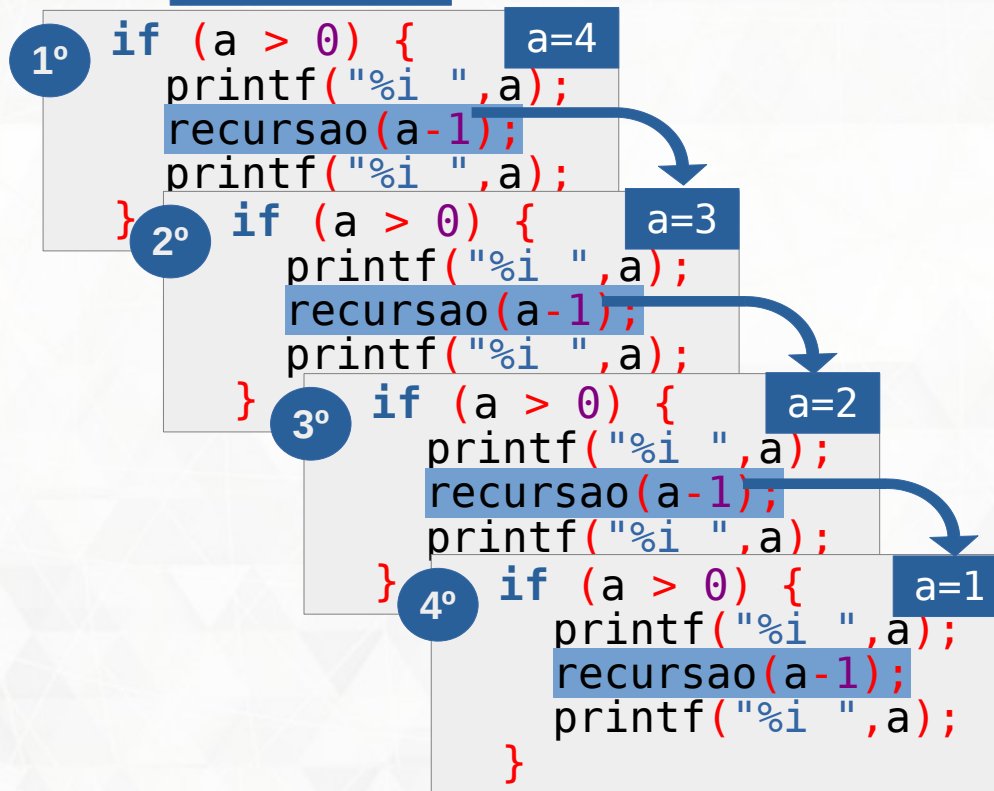
4 3



4 3 2

# Simulando empilhamento...

## Caminho Ida



4



4 3



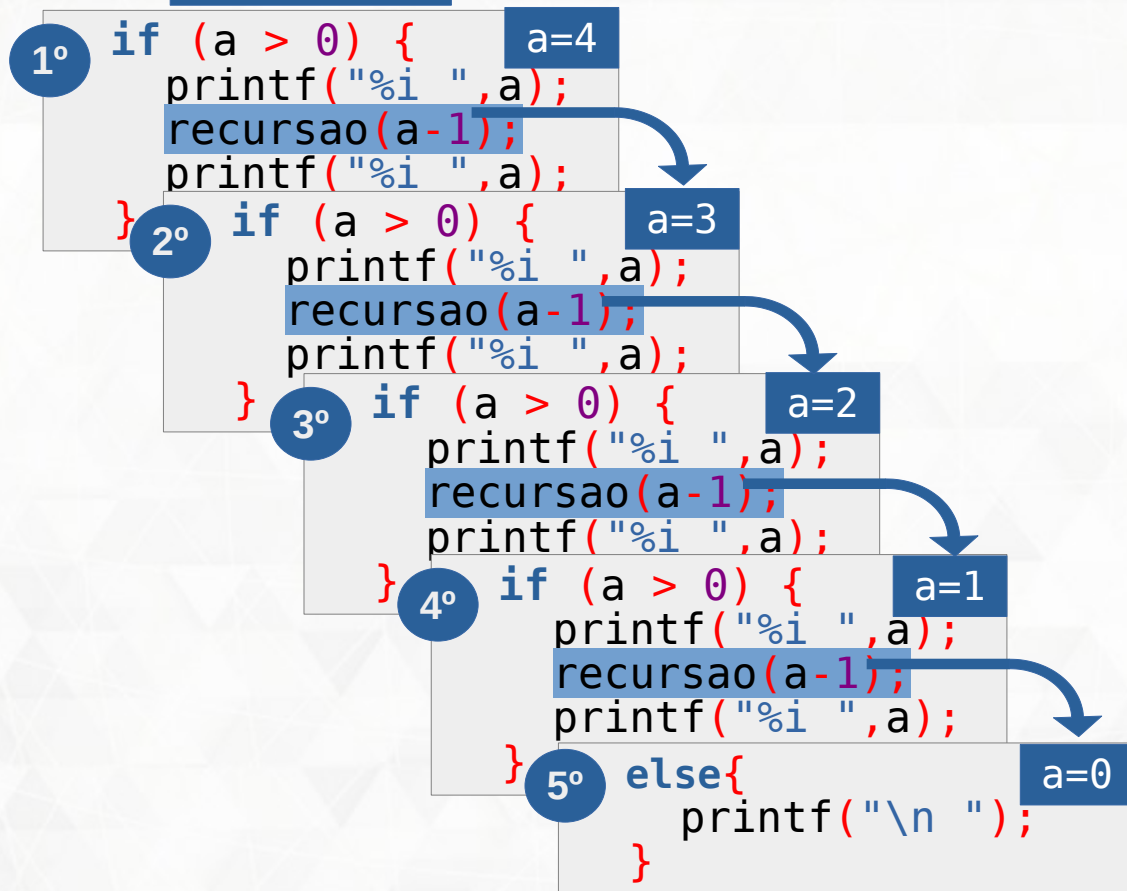
4 3 2



4 3 2 1

# Simulando empilhamento...


## Caminho Ida




```
4
```




```
4 3
```



```
4 3 2
```



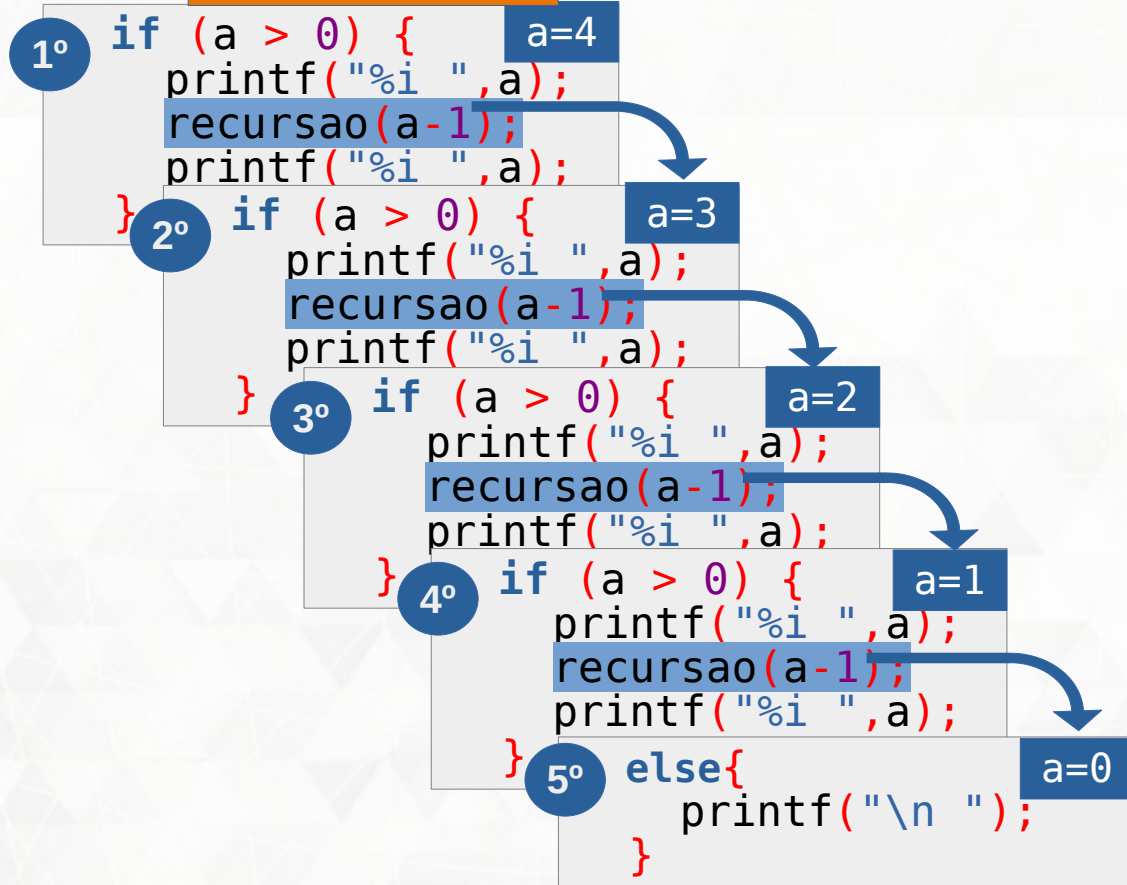
```
4 3 2 1
```



```
4 3 2 1 \n
```

# Simulando desempenho...

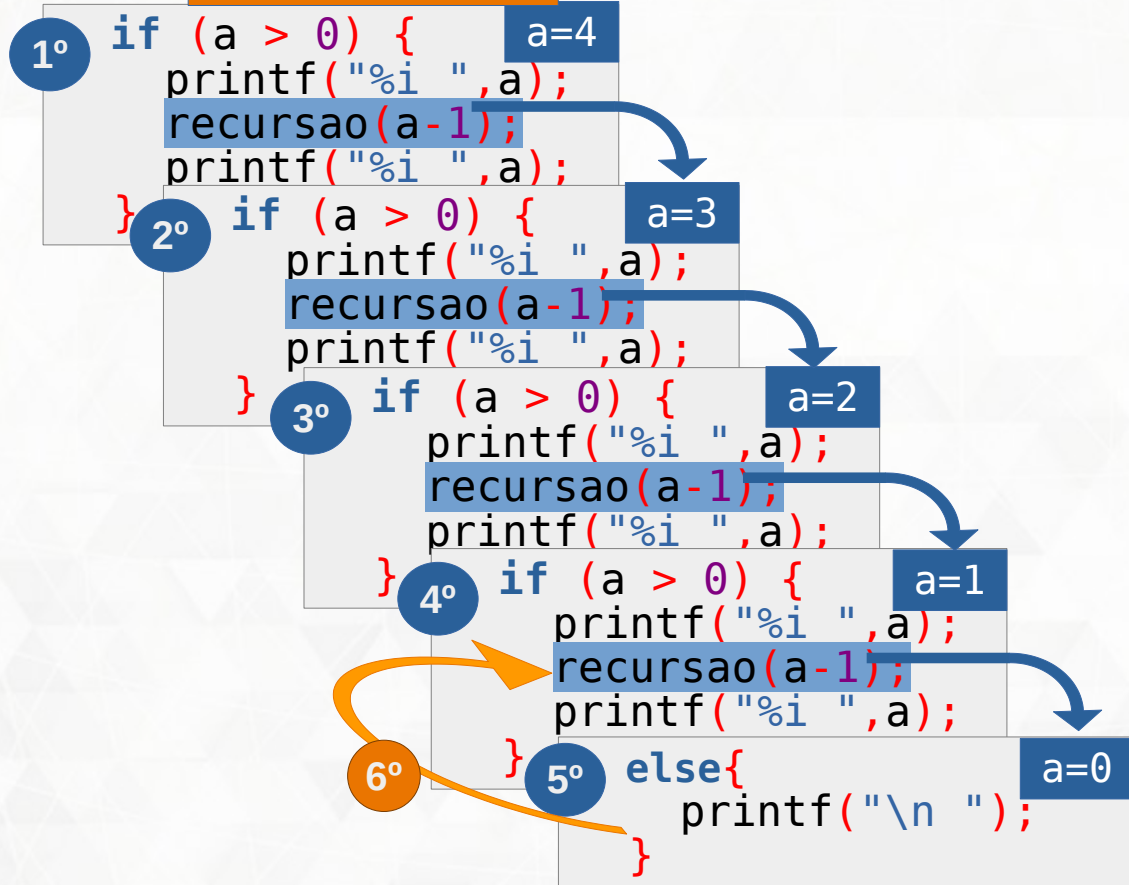
## Caminho Volta



4 3 2 1 \n

# Simulando desempenho...

## Caminho Volta



```
4 3 2 1
1
```


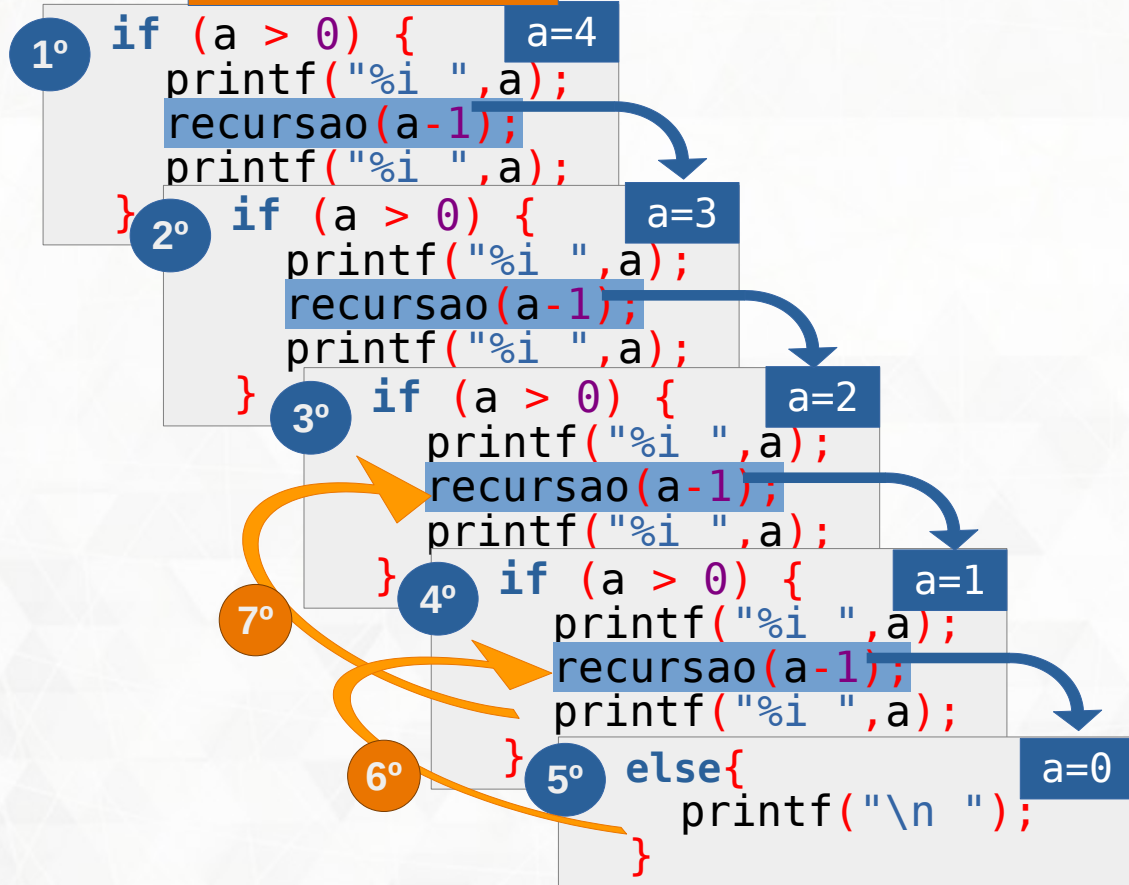


```
4 3 2 1 \n
```




# Simulando desempenho...


## Caminho Volta



```
4 3 2 1  
1 2
```



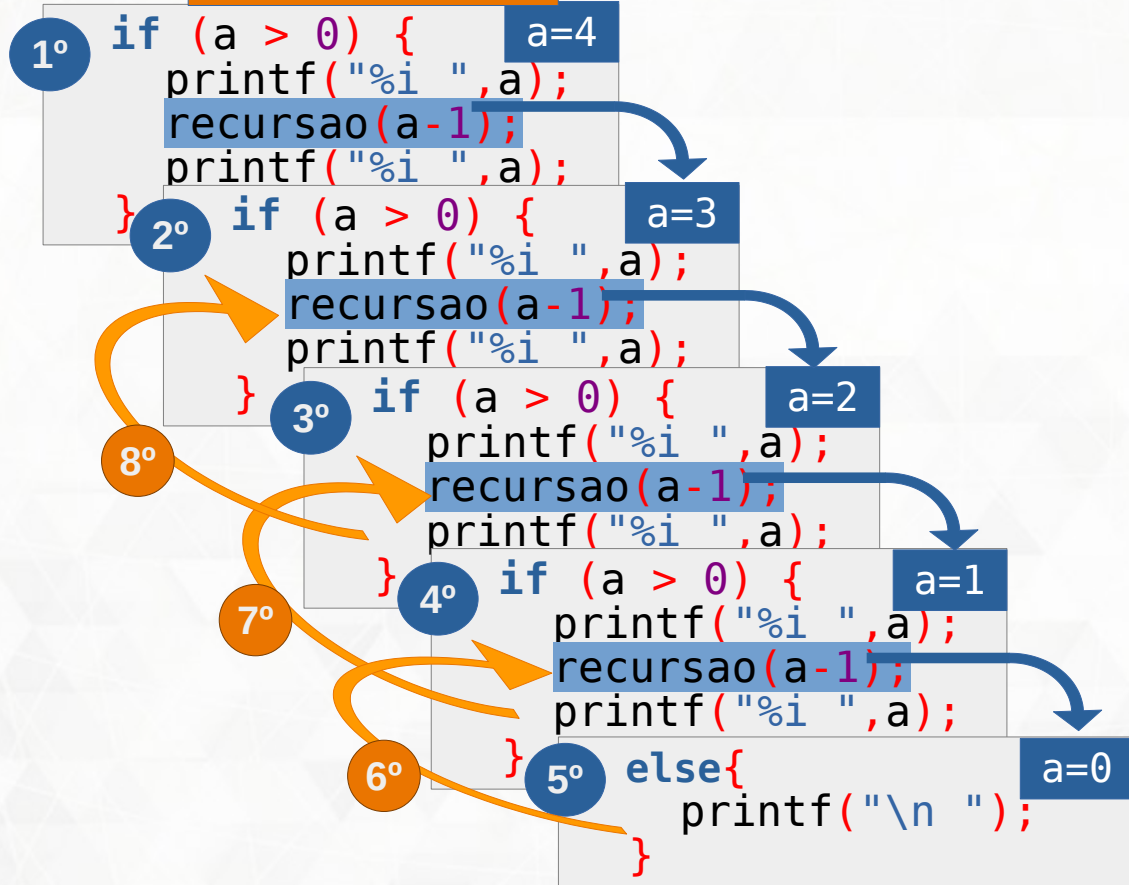
```
4 3 2 1  
1
```



```
4 3 2 1 \n
```

# Simulando desempenho...

## Caminho Volta



```
4 3 2 1  
1 2 3
```



```
4 3 2 1  
1 2
```




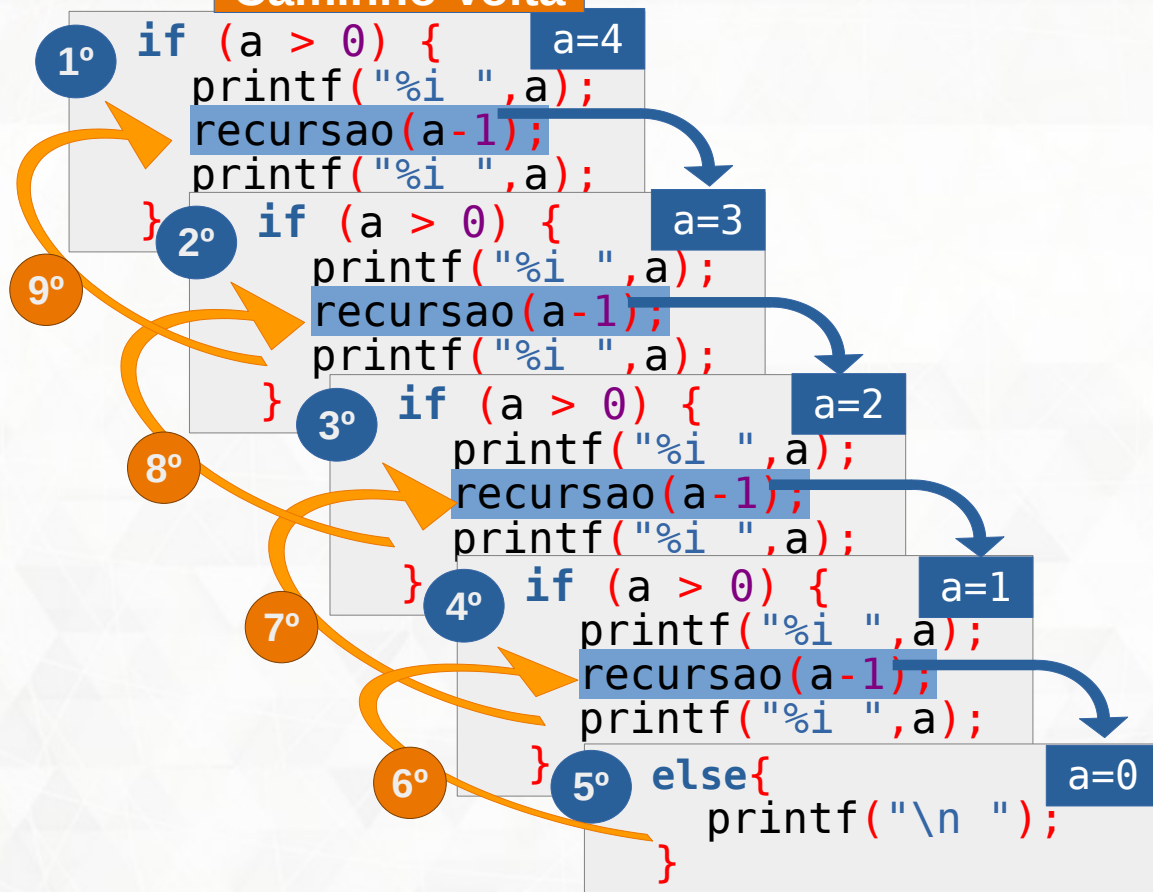
```
4 3 2 1  
1
```




```
4 3 2 1 \n
```

# Simulando desempenho...


## Caminho Volta




```
4 3 2 1
1 2 3 4
```




```
4 3 2 1
1 2 3
```



```
4 3 2 1
1 2
```



```
4 3 2 1
1
```



```
4 3 2 1 \n
```

# Recursão com retorno

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

```
int somaNumeros(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**

# Simulando

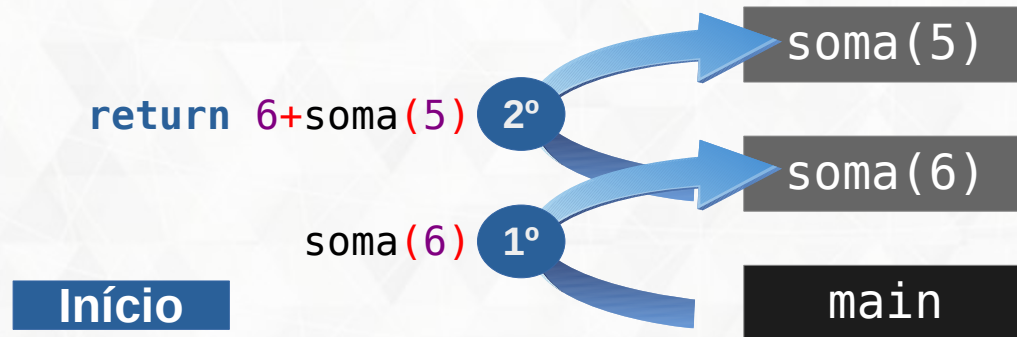
empilhamento





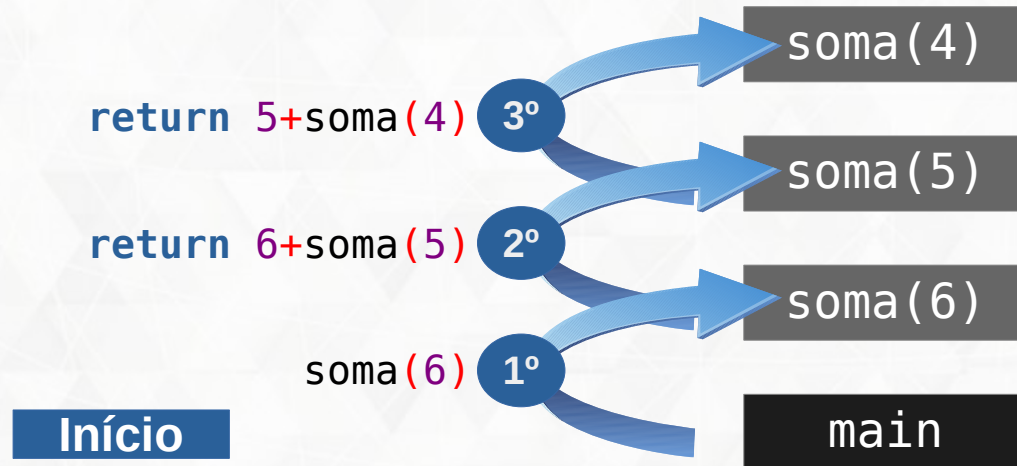
# Simulando

empilhamento



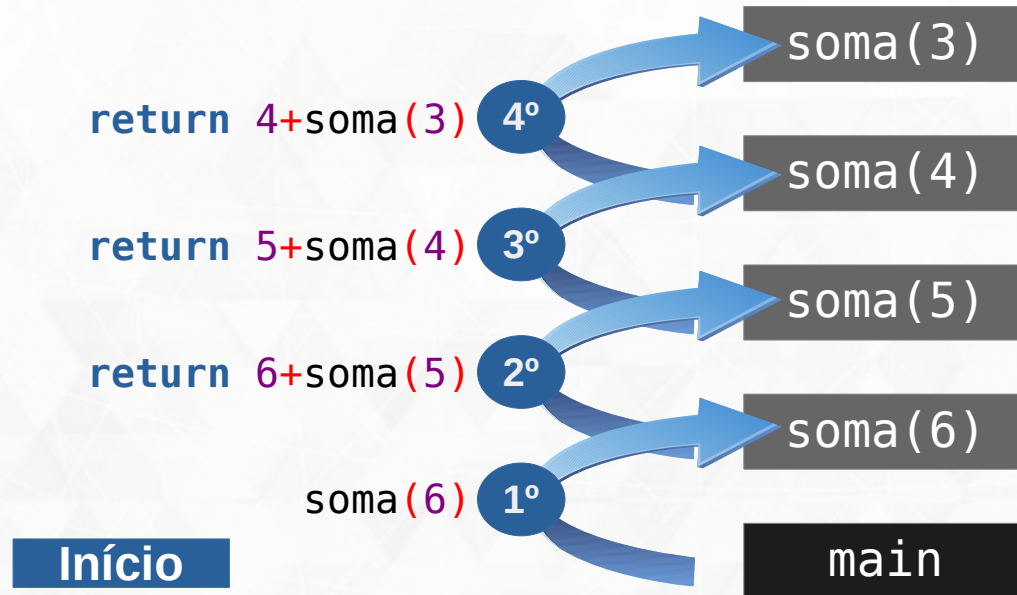
# Simulando

empilhamento



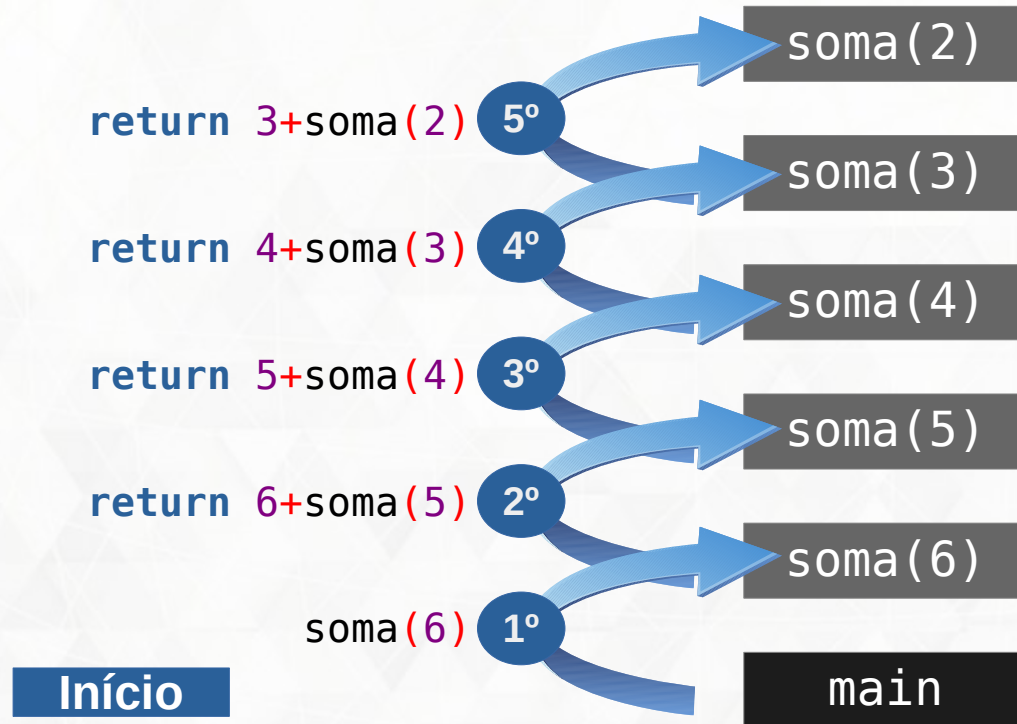
# Simulando

empilhamento



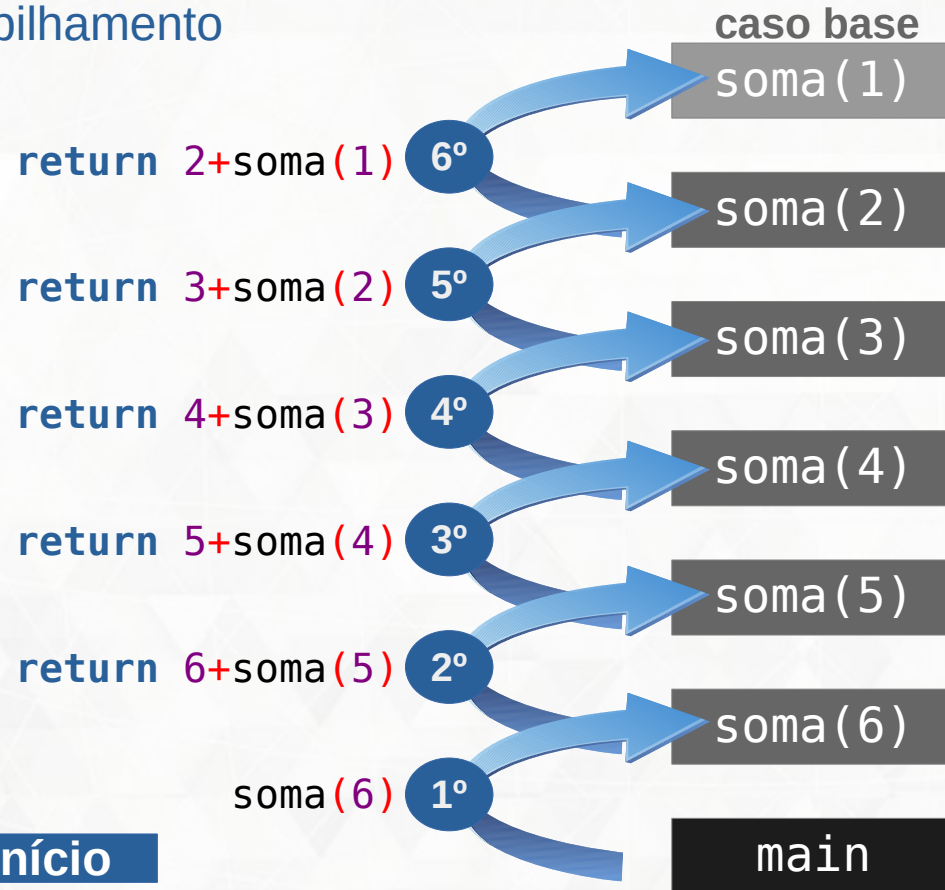
# Simulando

empilhamento



# Simulando

empilhamento

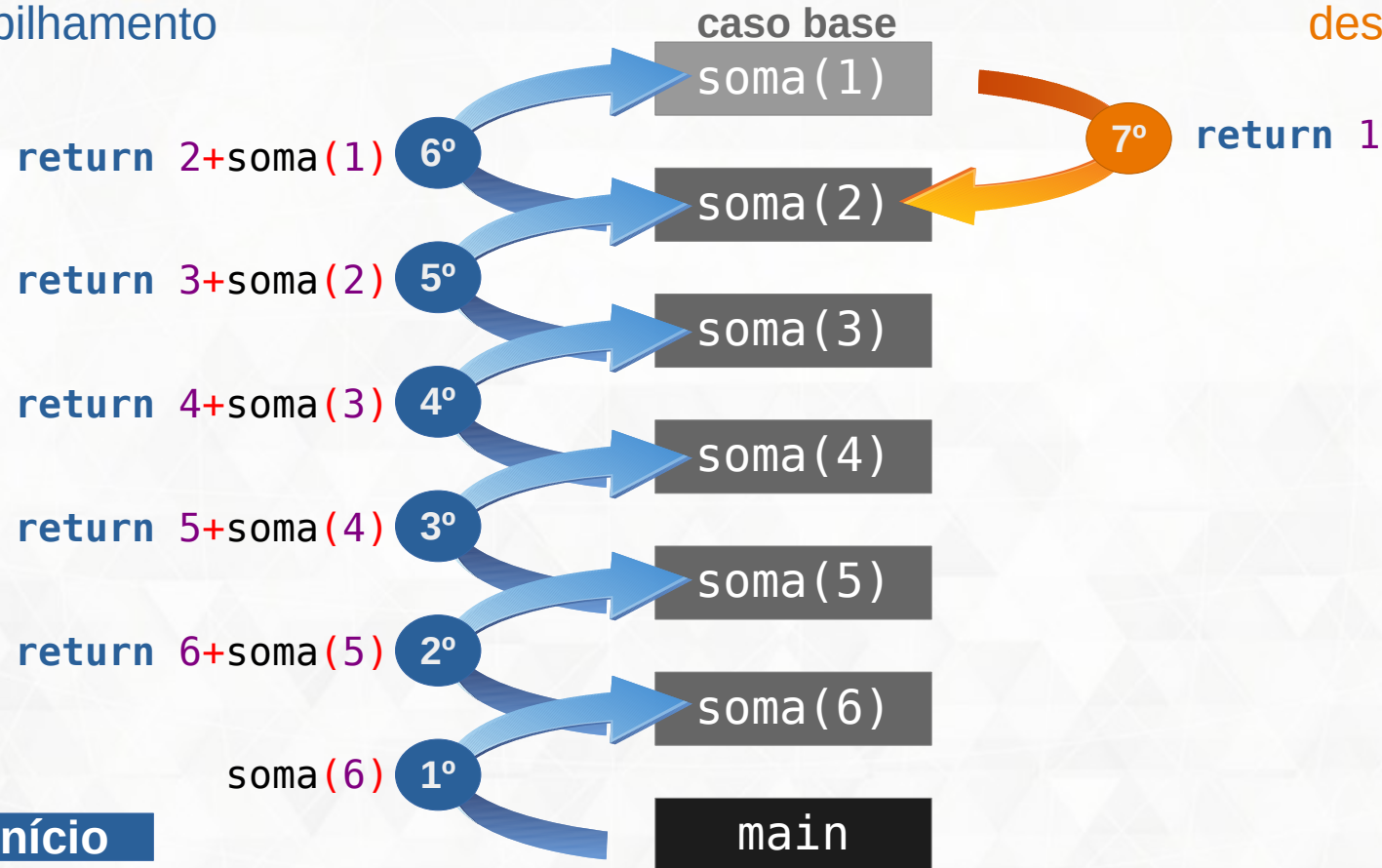




# Simulando

empilhamento

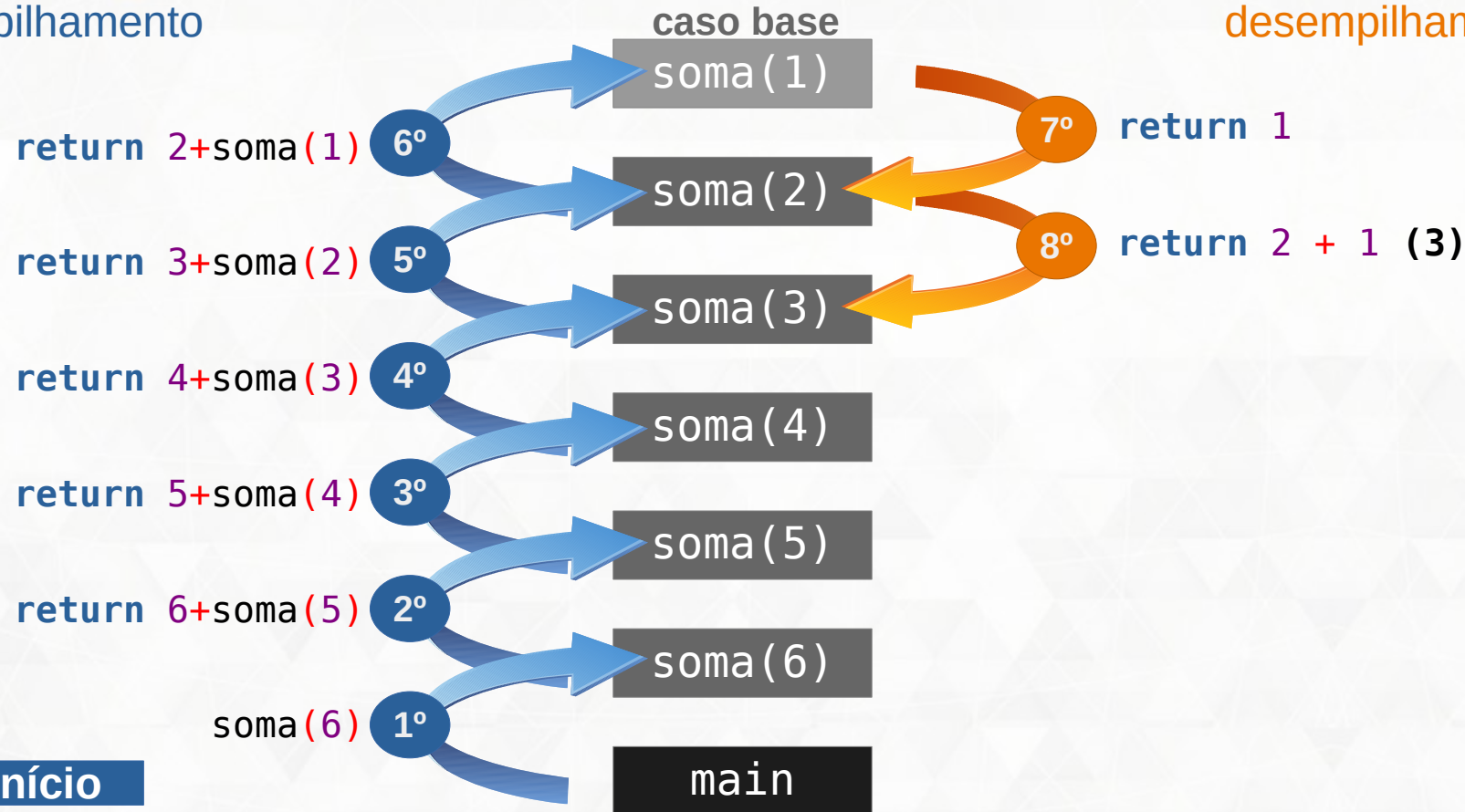
desempilhamento



# Simulando

empilhamento

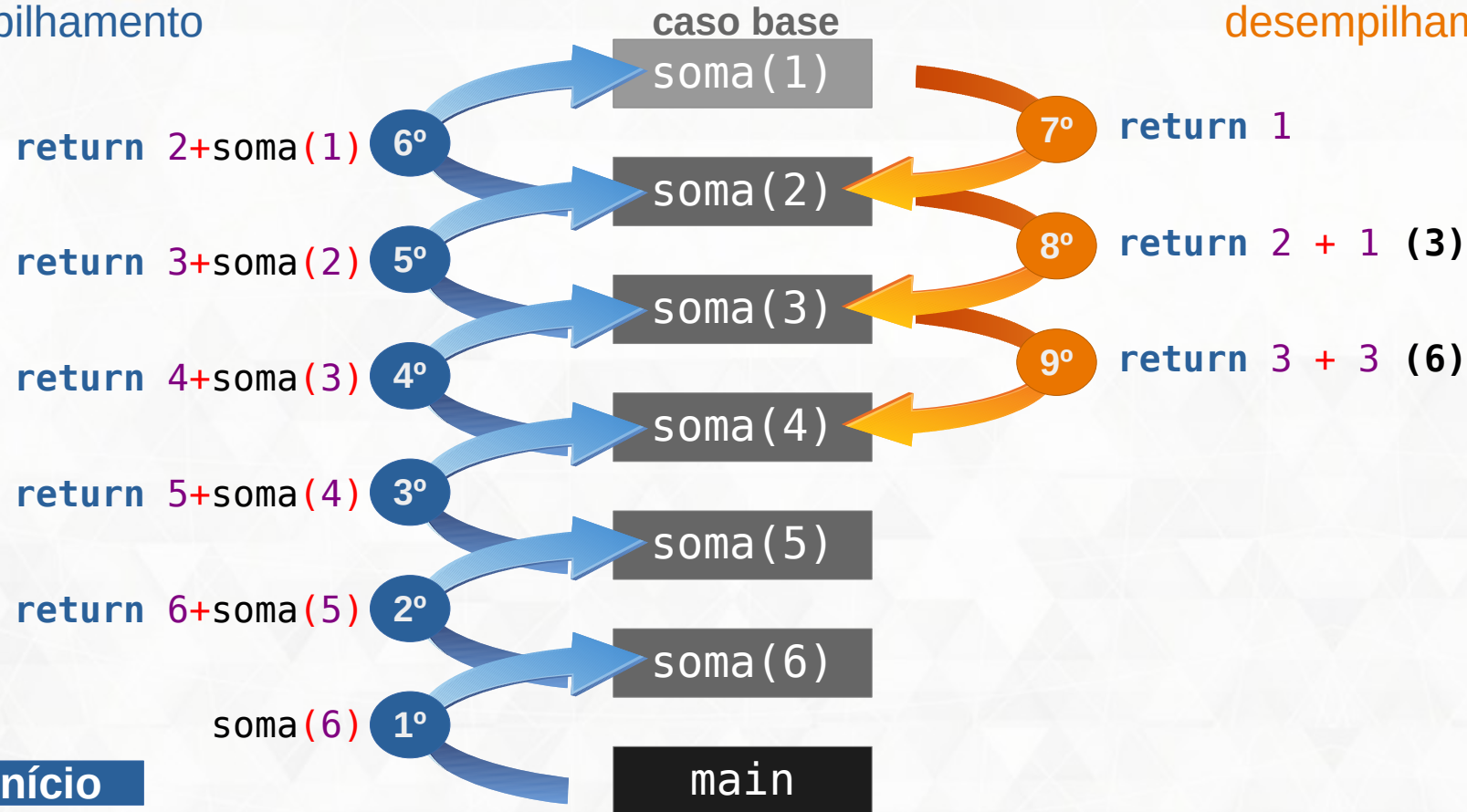
desempilhamento



# Simulando

empilhamento

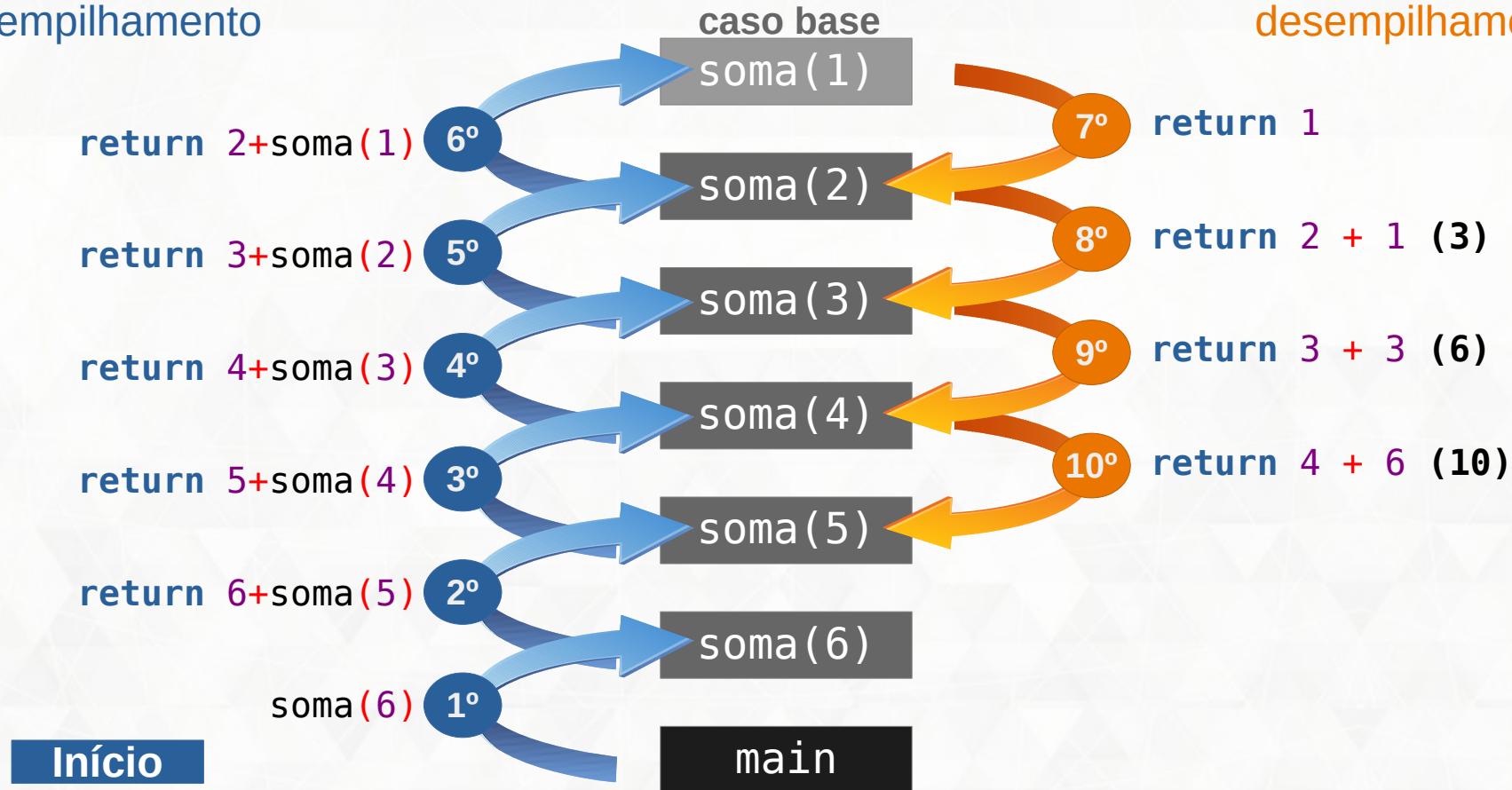
desempilhamento



# Simulando

empilhamento

desempilhamento

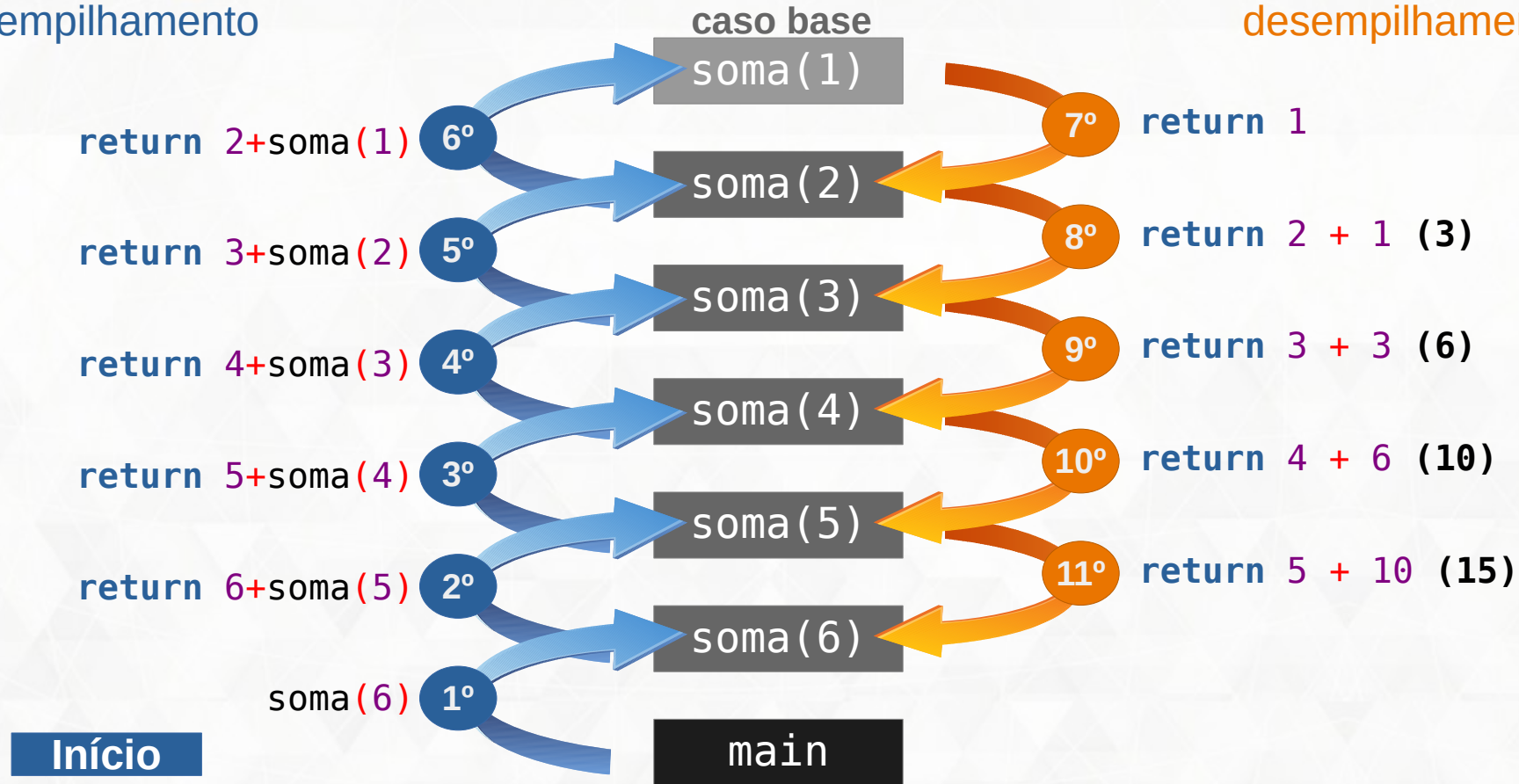




# Simulando

empilhamento

desempilhamento

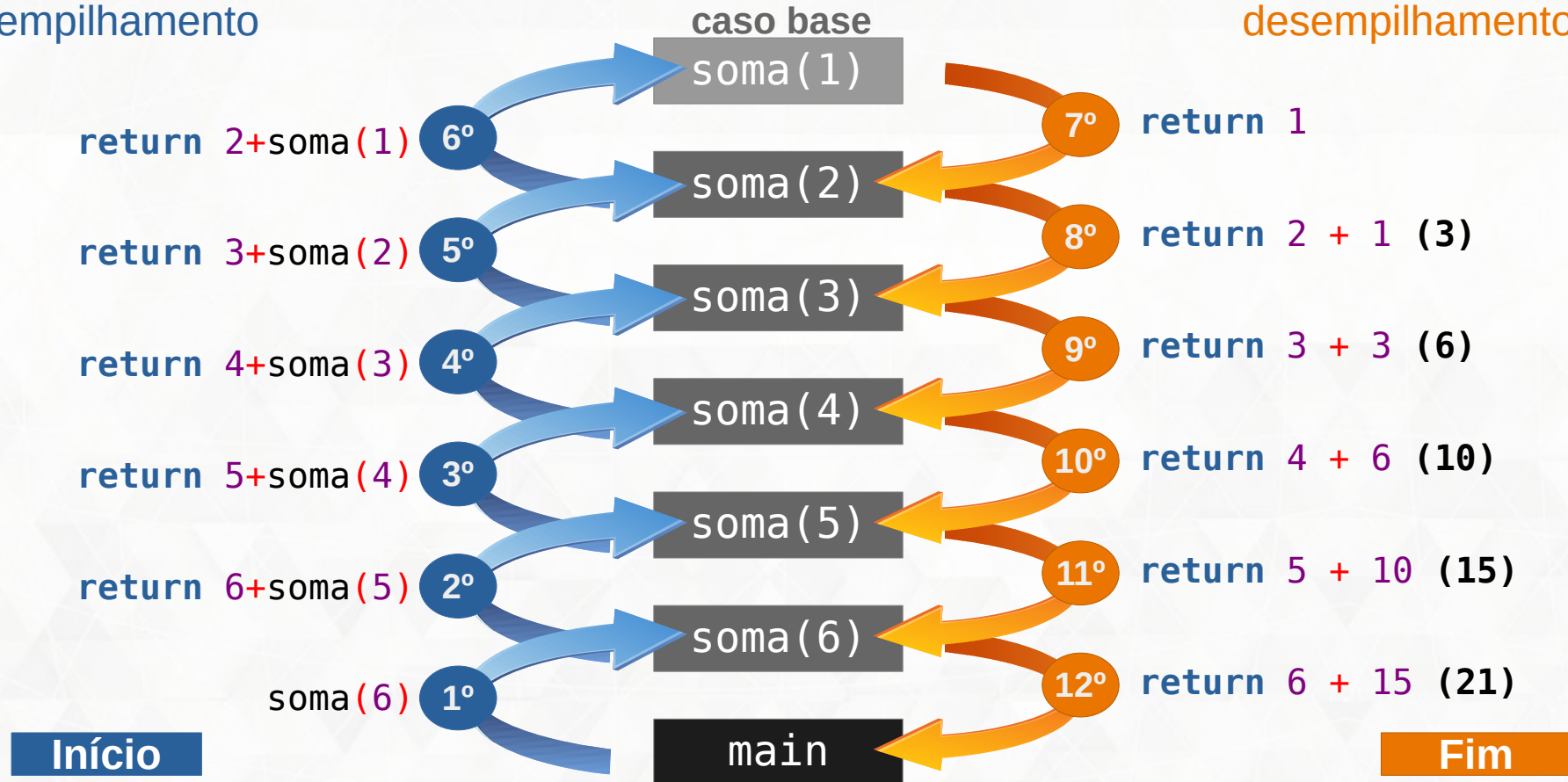




# Simulando

empilhamento

desempilhamento



# Exercícios

- **1)** Crie uma função que retorne o fatorial de um número passado por parâmetro. A ideia do fatorial está abaixo:

$$fatorial(n) = \begin{cases} 1 & \text{se } n = 0 \\ n \times fatorial(n - 1) & \text{se } n > 0 \end{cases}$$

- **2)** Crie uma função que retorne  $x*y$  através de operação de soma. A função recebe  $x$  e  $y$  por parâmetro
- **3)** Crie uma função que retorne  $x$  elevado a  $y$  através de operação de multiplicação. A função recebe  $x$  e  $y$  por parâmetro
- **4)** Faça uma função recursiva que retorne o  $n$ -ésimo termo da sequência de Fibonacci, sendo que  $n$  é recebido por parâmetro. Utilize essa função para desenvolver um programa que mostre no `main()` os  $n$  termos dessa sequência na tela, a partir do valor de  $n$  recebido pelo teclado. Sabe-se que o 1º termo é 0 e o 2º termo é 1.

# Exercícios

- **5)** Um problema típico em ciência da computação consiste em converter um número da sua forma decimal para binária. Crie um algoritmo recursivo para resolver esse problema.
  - **Solução trivial:**  $x=0$  quando o número inteiro já foi convertido para binário
  - **Passo da recursão:** saber como  $x/2$  é convertido. Depois, imprimir um dígito (0 ou 1) dado o sucesso da divisão.
- **6)** Considere a funcao abaixo. O que essa função faz? Escreva uma função não-recursiva que resolve o mesmo problema

```
int funcao(int a){  
    if(a <= 0) return 0;  
    else return a + funcao(a-1);  
}  
//funcao
```