



# Recursão

*Algoritmos e Programação II*

*2º semestre de 2023*

Prof. André Kishimoto

Prof. Gustavo Scalabrini Sampaio

Prof. Leandro Carlos Fernandes

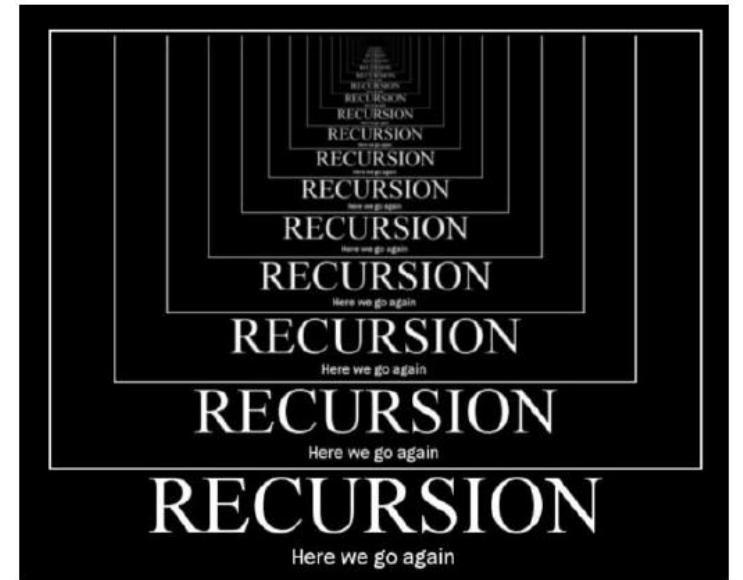
*(Conteúdo adaptado do material elaborado e gentilmente cedido pela  
profa. Ana Grasielle Dionisio Correa e prof. Tomaz Mikio Sasaki)*



# Recursão

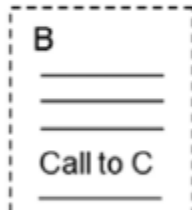
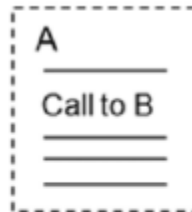
---

- A maioria dos problemas em computação envolve a repetição de passos.
- As declarações de controle **iterativo**, tais como *for* e *while*, é uma forma de controlar a execução repetida de instruções.
- Outro caminho para a solução desses problemas é por meio de **recursão**.
- Na solução de problemas recursivos, um problema é repetidamente quebrado em subproblemas até que o **subproblema** possa ser resolvido diretamente.

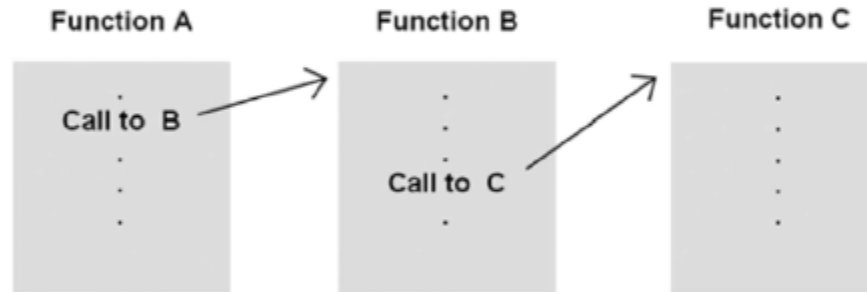


# Funções Não-Recursivas

## Function Definitions



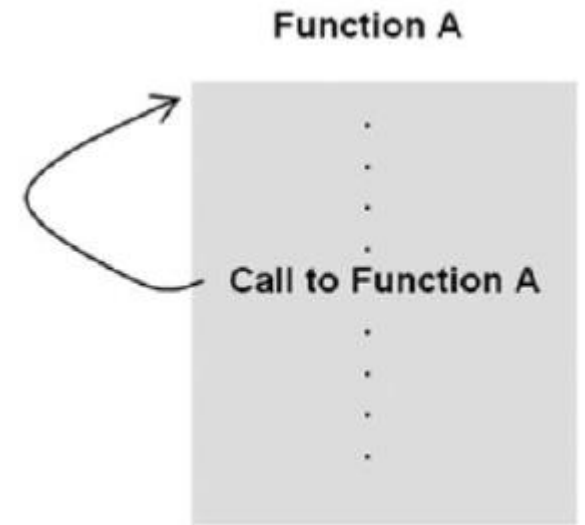
## Function Instances



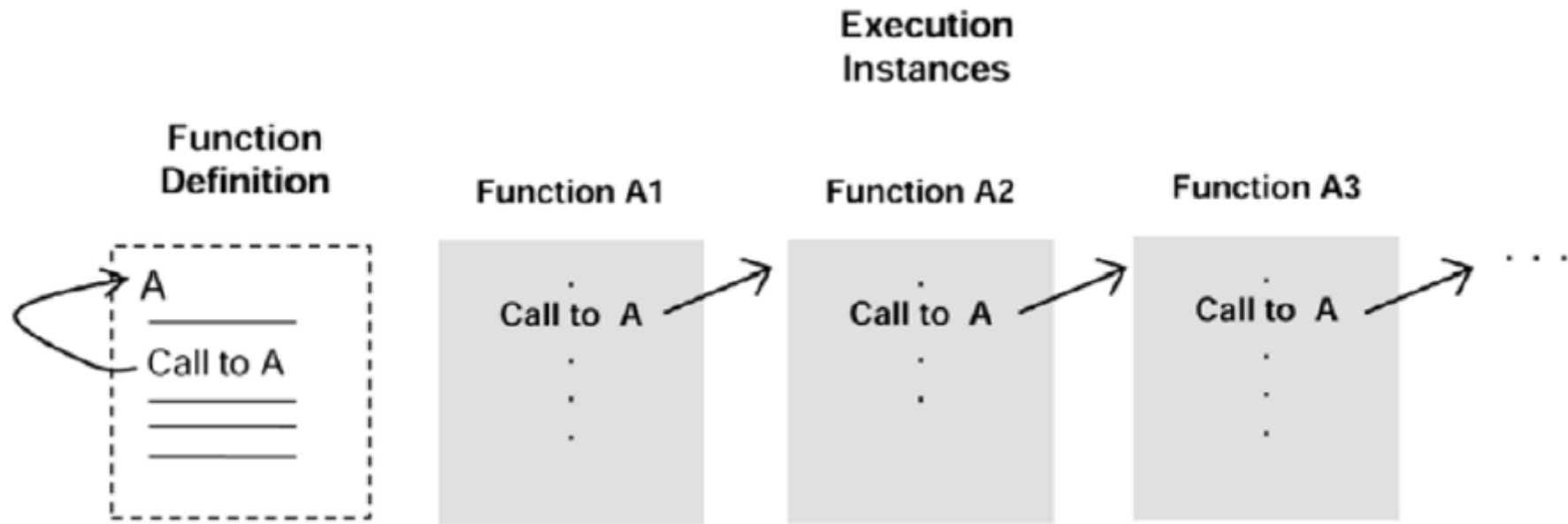
# Função Recursiva

---

- Uma **função recursiva** é definida como uma função que condicionalmente chama a si mesma.
- Na figura ao lado, temos uma função de nome *A* que é definida em algum ponto para chamar a função *A* (ela própria).
- Existe dois tipos de entidades relacionadas com qualquer função: a **definição da função** e quaisquer **instâncias de execução**. Uma função que chama a si mesmo é uma função instância de execução que chama outra instância de execução da mesma função.



# Função Recursiva



Cada instância de execução atual da função A gerará uma nova instância de execução da função A.



# Função Recursiva

---

- A execução de uma série de instâncias da função recursiva é similar à execução de uma série de instâncias não-recursivas, exceto que as instâncias são clones uma da outra. Assim, todas as instâncias são idênticas, e a chamada da função ocorre exatamente no mesmo lugar.
- Por outro lado, toda função recursiva deve contemplar, no processo de diminuir sucessivamente o problema em um problema menor, uma chamada de instância de execução que permita resolver o problema de forma direta, sem recorrer a si mesmo.
- Quando isso ocorre, diz-se que o algoritmo atingiu uma **condição de parada ou o seu caso base**, a qual deve estar presente em pelo menos um local dentro da função. Sem esta condição o algoritmo não para de chamar a si mesmo, até estourar a capacidade da pilha, o que geralmente causa efeitos colaterais e até mesmo o término indesejável do programa.



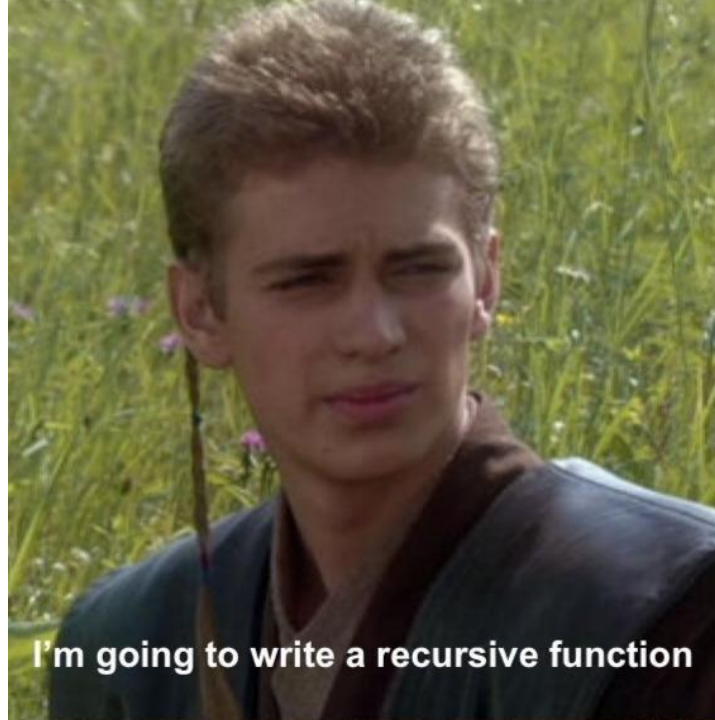
# Função Recursiva

---

Portanto:

- Uma função pode chamar a si mesma.
- Dois pontos importantes para funções recursivas:
  1. Cada vez que a função chama a si mesma, uma nova cópia da função é empilhada na memória do computador (*call stack*).
  2. Toda função recursiva precisa ter uma condição de parada. Caso contrário, a função seria chamada infinitas vezes e “nunca” terminaria. O resultado de uma recursividade infinita é o estouro de pilha (*stack overflow*).

# Função recursiva + caso base





# Recursão

O que será exibido no console ao chamar a função `rfunc()` com valor inteiro 5?

```
1  #include <stdio.h>
2
3  int rfunc(int n)
4  {
5      if (n == 1)
6          return 1;
7      else
8          return n + rfunc(n - 1);
9  }
10
11 int main()
12 {
13     int x = rfunc(5);
14     printf("Saída %d\n", x);
15 }
```

Pilha de chamada ( <i>call stack</i> )	Retorno
main() --> x = rfunc(5);	



# Recursão

O que será exibido no console ao chamar a função `rfunc()` com valor inteiro 5?

```
1  #include <stdio.h>
2
3  int rfunc(int n)
4  {
5      if (n == 1)
6          return 1;
7      else
8          return n + rfunc(n - 1);
9  }
10
11 int main()
12 {
13     int x = rfunc(5);
14     printf("Saída %d\n", x);
15 }
```

Pilha de chamada ( <i>call stack</i> )	Retorno
rfunc(n=5) --> 5 + rfunc(4);	5 + ?
main() --> x = rfunc(5);	



# Recursão

O que será exibido no console ao chamar a função `rfunc()` com valor inteiro 5?

```
1  #include <stdio.h>
2
3  int rfunc(int n)
4  {
5      if (n == 1)
6          return 1;
7      else
8          return n + rfunc(n - 1);
9  }
10
11 int main()
12 {
13     int x = rfunc(5);
14     printf("Saída %d\n", x);
15 }
```

Pilha de chamada ( <i>call stack</i> )	Retorno
rfunc(n=4) --> 4 + rfunc(3);	4 + ?
rfunc(n=5) --> 5 + rfunc(4);	5 + ?
main() --> x = rfunc(5);	



# Recursão

O que será exibido no console ao chamar a função rfunc() com valor inteiro 5?

```
1  #include <stdio.h>
2
3  int rfunc(int n)
4  {
5      if (n == 1)
6          return 1;
7      else
8          return n + rfunc(n - 1);
9  }
10
11 int main()
12 {
13     int x = rfunc(5);
14     printf("Saída %d\n", x);
15 }
```

Pilha de chamada ( <i>call stack</i> )	Retorno
rfunc(n=3) --> 3 + rfunc(2);	3 + ?
rfunc(n=4) --> 4 + rfunc(3);	4 + ?
rfunc(n=5) --> 5 + rfunc(4);	5 + ?
main() --> x = rfunc(5);	



# Recursão

O que será exibido no console ao chamar a função rfunc() com valor inteiro 5?

```
1  #include <stdio.h>
2
3  int rfunc(int n)
4  {
5      if (n == 1)
6          return 1;
7      else
8          return n + rfunc(n - 1);
9  }
10
11 int main()
12 {
13     int x = rfunc(5);
14     printf("Saída %d\n", x);
15 }
```

Pilha de chamada ( <i>call stack</i> )	Retorno
rfunc(n=2) --> 2 + rfunc(1);	2 + ?
rfunc(n=3) --> 3 + rfunc(2);	3 + ?
rfunc(n=4) --> 4 + rfunc(3);	4 + ?
rfunc(n=5) --> 5 + rfunc(4);	5 + ?
main() --> x = rfunc(5);	



# Recursão

O que será exibido no console ao chamar a função `rfunc()` com valor inteiro 5?

```
1  #include <stdio.h>
2
3  int rfunc(int n)
4  {
5      if (n == 1)
6          return 1;
7      else
8          return n + rfunc(n - 1);
9  }
10
11 int main()
12 {
13     int x = rfunc(5);
14     printf("Saída %d\n", x);
15 }
```

Pilha de chamada ( <i>call stack</i> )	Retorno
rfunc(n=1)	1
rfunc(n=2) --> 2 + rfunc(1);	2 + ?
rfunc(n=3) --> 3 + rfunc(2);	3 + ?
rfunc(n=4) --> 4 + rfunc(3);	4 + ?
rfunc(n=5) --> 5 + rfunc(4);	5 + ?
main() --> x = rfunc(5);	



# Recursão

O que será exibido no console ao chamar a função rfunc() com valor inteiro 5?

```
1  #include <stdio.h>
2
3  int rfunc(int n)
4  {
5      if (n == 1)
6          return 1;
7      else
8          return n + rfunc(n - 1);
9  }
10
11 int main()
12 {
13     int x = rfunc(5);
14     printf("Saída %d\n", x);
15 }
```

Pilha de chamada ( <i>call stack</i> )	Retorno
rfunc(n=1)	1
rfunc(n=2) --> 2 + rfunc(1);	2 + ?
rfunc(n=3) --> 3 + rfunc(2);	3 + ?
rfunc(n=4) --> 4 + rfunc(3);	4 + ?
rfunc(n=5) --> 5 + rfunc(4);	5 + ?
main() --> x = rfunc(5);	



# Recursão

O que será exibido no console ao chamar a função rfunc() com valor inteiro 5?

```
1  #include <stdio.h>
2
3  int rfunc(int n)
4  {
5      if (n == 1)
6          return 1;
7      else
8          return n + rfunc(n - 1);
9  }
10
11 int main()
12 {
13     int x = rfunc(5);
14     printf("Saída %d\n", x);
15 }
```

Pilha de chamada ( <i>call stack</i> )	Retorno
rfunc(n=1)	1
rfunc(n=2) --> 2 + rfunc(1);	2 + 1 ←
rfunc(n=3) --> 3 + rfunc(2);	3 + ?
rfunc(n=4) --> 4 + rfunc(3);	4 + ?
rfunc(n=5) --> 5 + rfunc(4);	5 + ?
main() --> x = rfunc(5);	





# Recursão

O que será exibido no console ao chamar a função rfunc() com valor inteiro 5?

```
1  #include <stdio.h>
2
3  int rfunc(int n)
4  {
5      if (n == 1)
6          return 1;
7      else
8          return n + rfunc(n - 1);
9  }
10
11 int main()
12 {
13     int x = rfunc(5);
14     printf("Saída %d\n", x);
15 }
```

Pilha de chamada ( <i>call stack</i> )	Retorno
<del>rfunc(n=1)</del>	<del>1</del>
<del>rfunc(n=2) --&gt; 2 + rfunc(1);</del>	<del>2 + 1 = 3</del>
rfunc(n=3) --> 3 + rfunc(2);	3 + 3 ←
rfunc(n=4) --> 4 + rfunc(3);	4 + ?
rfunc(n=5) --> 5 + rfunc(4);	5 + ?
main() --> x = rfunc(5);	



# Recursão

O que será exibido no console ao chamar a função rfunc() com valor inteiro 5?

```
1  #include <stdio.h>
2
3  int rfunc(int n)
4  {
5      if (n == 1)
6          return 1;
7      else
8          return n + rfunc(n - 1);
9  }
10
11 int main()
12 {
13     int x = rfunc(5);
14     printf("Saída %d\n", x);
15 }
```

Pilha de chamada ( <i>call stack</i> )	Retorno
<del>rfunc(n=1)</del>	<del>1</del>
<del>rfunc(n=2) --&gt; 2 + rfunc(1);</del>	<del>3</del>
<del>rfunc(n=3) --&gt; 3 + rfunc(2);</del>	<del>3 + 3 = 6</del>
rfunc(n=4) --> 4 + rfunc(3);	4 + 6 ←
rfunc(n=5) --> 5 + rfunc(4);	5 + ?
main() --> x = rfunc(5);	



# Recursão

O que será exibido no console ao chamar a função rfunc() com valor inteiro 5?

```
1  #include <stdio.h>
2
3  int rfunc(int n)
4  {
5      if (n == 1)
6          return 1;
7      else
8          return n + rfunc(n - 1);
9  }
10
11 int main()
12 {
13     int x = rfunc(5);
14     printf("Saída %d\n", x);
15 }
```

Pilha de chamada ( <i>call stack</i> )	Retorno
<del>rfunc(n=1)</del>	<del>1</del>
<del>rfunc(n=2) --&gt; 2 + rfunc(1);</del>	<del>3</del>
<del>rfunc(n=3) --&gt; 3 + rfunc(2);</del>	<del>6</del>
<del>rfunc(n=4) --&gt; 4 + rfunc(3);</del>	<del>4 + 6 = 10</del>
rfunc(n=5) --> 5 + rfunc(4);	5 + 10 ←
main() --> x = rfunc(5);	



# Recursão

O que será exibido no console ao chamar a função rfunc() com valor inteiro 5?

```
1  #include <stdio.h>
2
3  int rfunc(int n)
4  {
5      if (n == 1)
6          return 1;
7      else
8          return n + rfunc(n - 1);
9  }
10
11 int main()
12 {
13     int x = rfunc(5);
14     printf("Saída %d\n", x);
15 }
```

Pilha de chamada ( <i>call stack</i> )	Retorno
rfunc(n=1)	1
rfunc(n=2) --> 2 + rfunc(1);	3
rfunc(n=3) --> 3 + rfunc(2);	6
rfunc(n=4) --> 4 + rfunc(3);	10
rfunc(n=5) --> 5 + rfunc(4);	5 + 10 = 15
main() --> x = rfunc(5);	15



# Recursão

O que será exibido no console ao chamar a função rfunc() com valor inteiro 5?

```
1  #include <stdio.h>
2
3  int rfunc(int n)
4  {
5      if (n == 1)
6          return 1;
7      else
8          return n + rfunc(n - 1);
9  }
10
11 int main()
12 {
13     int x = rfunc(5);
14     printf("Saída %d\n", x);
15 }
```

Pilha de chamada ( <i>call stack</i> )	Retorno
rfunc(n=1)	1
rfunc(n=2) --> 2 + <del>rfunc(1)</del> ;	3
rfunc(n=3) --> 3 + <del>rfunc(2)</del> ;	6
rfunc(n=4) --> 4 + <del>rfunc(3)</del> ;	10
rfunc(n=5) --> 5 + <del>rfunc(4)</del> ;	15
main() --> x = <del>rfunc(5)</del> ;	15



# Recursão

O que será exibido no console ao chamar a função rfunc() com valor inteiro 5?

```
1  #include <stdio.h>
2
3  int rfunc(int n)
4  {
5      if (n == 1)
6          return 1;
7      else
8          return n + rfunc(n - 1);
9  }
10
11 int main()
12 {
13     int x = rfunc(5);
14     printf("Saída %d\n", x);
15 }
```

Pilha de chamada ( <i>call stack</i> )	Retorno
rfunc(n=1)	1
rfunc(n=2) --> 2 + <del>rfunc(1)</del> ;	3
rfunc(n=3) --> 3 + <del>rfunc(2)</del> ;	6
rfunc(n=4) --> 4 + <del>rfunc(3)</del> ;	10
rfunc(n=5) --> 5 + <del>rfunc(4)</del> ;	15
main() --> x = 15	



# Recursão

O que será exibido no console ao chamar a função rfunc() com valor inteiro 5?

```
1  #include <stdio.h>
2
3  int rfunc(int n)
4  {
5      if (n == 1)
6          return 1;
7      else
8          return n + rfunc(n - 1);
9  }
10
11 int main()
12 {
13     int x = rfunc(5);
14     printf("Saída %d\n", x);
15 }
```

Pilha de chamada ( <i>call stack</i> )	Retorno
rfunc(n=1)	1
rfunc(n=2) --> 2 + rfunc(1);	3
rfunc(n=3) --> 3 + rfunc(2);	6
rfunc(n=4) --> 4 + rfunc(3);	10
rfunc(n=5) --> 5 + rfunc(4);	15
main() --> x = 15	

```
~/dev/recursao$ ./rec3
Saída 15
```



# Fatorial iterativo x fatorial recursivo

---

- Mecanismos de recursão são **adequados** para resolução de **problemas que já possuem uma estrutura recursiva**.
- Por exemplo, vamos considerar o problema de se calcular o fatorial  $n$  de um número natural  $n \geq 0$ .
- Se considerarmos a definição de fatorial como:

$$n! = n \times (n - 1) \times (n - 2) \times (n - 3) \times \cdots \times 1$$





# Fatorial iterativo x fatorial recursivo

---

## Fatorial iterativo

```
3  int fatorial_iterativo(int n)
4  {
5      int fatorial = 1;
6      int contador = 1;
7
8      while (contador <= n)
9      {
10         fatorial *= contador;
11         contador++;
12     }
13
14     return fatorial;
15 }
```

## Fatorial recursivo



# Fatorial iterativo x fatorial recursivo

---

## Fatorial iterativo

```
3  int fatorial_iterativo(int n)
4  {
5      int fatorial = 1;
6      int contador = 1;
7
8      while (contador <= n)
9      {
10         fatorial *= contador;
11         contador++;
12     }
13
14     return fatorial;
15 }
```

## Fatorial recursivo

```
17 int fatorial_recursivo(int n)
18 {
19     if (n == 1)
20         return 1;
21
22     return n * fatorial_recursivo(n - 1);
23 }
```



# Fatorial recursivo

---

```
17 int fatorial_recursivo(int n)
18 {
19     if (n == 1)
20         return 1;
21
22     return n * fatorial_recursivo(n - 1);
23 }
```

resultado = fatorial\_recursivo(5)



# Fatorial recursivo

```
17 int fatorial_recursivo(int n)
18 {
19     if (n == 1)
20         return 1;
21
22     return n * fatorial_recursivo(n - 1);
23 }
```

**n = 5**

resultado = fatorial\_recursivo(5)



# Fatorial recursivo



```
17 int fatorial_recursivo(int n)
18 {
19     if (n == 1) ←
20         return 1;
21
22     return n * fatorial_recursivo(n - 1);
23 }
```

**n = 5**


resultado = fatorial\_recursivo(5)



# Fatorial recursivo



```
17 int fatorial_recursivo(int n)
18 {
19     if (n == 1)
20         return 1;
21
22     return n * fatorial_recursivo(n - 1);
23 }
```



**n = 5**

resultado = fatorial\_recursivo(5)  
              5 \* fatorial\_recursivo(4)



# Fatorial recursivo

```
17 int fatorial_recursivo(int n)
18 {
19     if (n == 1)
20         return 1;
21
22     return n * fatorial_recursivo(n - 1);
23 }
```

**n = 5 4**

resultado = fatorial\_recursivo(5)  
              5 \* fatorial\_recursivo(4)



# Fatorial recursivo

---

```
17 int fatorial_recursivo(int n)
18 {
19     if (n == 1) ←
20         return 1;
21
22     return n * fatorial_recursivo(n - 1);
23 }
```

**n = 5 4**

resultado = fatorial\_recursivo(5)  
              5 \* fatorial\_recursivo(4)





# Fatorial recursivo

```
17 int fatorial_recursivo(int n)
18 {
19     if (n == 1)
20         return 1;
21
22     return n * fatorial_recursivo(n - 1);
23 }
```

**n = 5 4**

resultado = fatorial\_recursivo(5)  
            5 \* fatorial\_recursivo(4)  
            4 \* fatorial\_recursivo(3)



# Fatorial recursivo

```
17 int fatorial_recursivo(int n)
18 {
19     if (n == 1)
20         return 1;
21
22     return n * fatorial_recursivo(n - 1);
23 }
```

**n = 5 4 3**

resultado = fatorial\_recursivo(5)  
              5 \* fatorial\_recursivo(4)  
              4 \* fatorial\_recursivo(3)



# Fatorial recursivo

```
17 int fatorial_recursivo(int n)
18 {
19     if (n == 1) ←
20         return 1;
21
22     return n * fatorial_recursivo(n - 1);
23 }
```

**n = 5 4 3**


resultado = fatorial\_recursivo(5)  
            5 \* fatorial\_recursivo(4)  
            4 \* fatorial\_recursivo(3)



# Fatorial recursivo



```
17 int fatorial_recursivo(int n)
18 {
19     if (n == 1)
20         return 1;
21
22     return n * fatorial_recursivo(n - 1);
23 }
```



**n = 5 4 3**

```
resultado = fatorial_recursivo(5)
             5 * fatorial_recursivo(4)
               4 * fatorial_recursivo(3)
                 3 * fatorial_recursivo(2)
```



# Fatorial recursivo

```
17 int fatorial_recursivo(int n)
18 {
19     if (n == 1)
20         return 1;
21
22     return n * fatorial_recursivo(n - 1);
23 }
```

**n = 5 4 3 2**

resultado = fatorial\_recursivo(5)  
              5 \* fatorial\_recursivo(4)  
              4 \* fatorial\_recursivo(3)  
              3 \* fatorial\_recursivo(2)



# Fatorial recursivo

```
17 int fatorial_recursivo(int n)
18 {
19     if (n == 1) ←
20         return 1;
21
22     return n * fatorial_recursivo(n - 1);
23 }
```

**n** = 5 4 3 2


resultado = fatorial\_recursivo(5)  
          5 \* fatorial\_recursivo(4)  
          4 \* fatorial\_recursivo(3)  
          3 \* fatorial\_recursivo(2)



# Fatorial recursivo



```
17 int fatorial_recursivo(int n)
18 {
19     if (n == 1)
20         return 1;
21
22     return n * fatorial_recursivo(n - 1);
23 }
```



**n** = 5 4 3 2

resultado = fatorial\_recursivo(5)  
          5 \* fatorial\_recursivo(4)  
          4 \* fatorial\_recursivo(3)  
          3 \* fatorial\_recursivo(2)  
          2 \* fatorial\_recursivo(1)



# Fatorial recursivo

```
17 int fatorial_recursivo(int n)
18 {
19     if (n == 1)
20         return 1;
21
22     return n * fatorial_recursivo(n - 1);
23 }
```

**n** = 5 4 3 2 **1**

resultado = fatorial\_recursivo(5)  
            5 \* fatorial\_recursivo(4)  
                4 \* fatorial\_recursivo(3)  
                    3 \* fatorial\_recursivo(2)  
                        2 \* fatorial\_recursivo(**1**)





# Fatorial recursivo

```
17 int fatorial_recursivo(int n)
18 {
19     if (n == 1) ←
20         return 1;
21
22     return n * fatorial_recursivo(n - 1);
23 }
```

**n** = 5 4 3 2 **1**

```
resultado = fatorial_recursivo(5)
            5 * fatorial_recursivo(4)
              4 * fatorial_recursivo(3)
                3 * fatorial_recursivo(2)
                  2 * fatorial_recursivo(1)
```



# Fatorial recursivo

```
17 int fatorial_recursivo(int n)
18 {
19     if (n == 1)
20         return 1;
21
22     return n * fatorial_recursivo(n - 1);
23 }
```

**n** = 5 4 3 2 **1**

resultado = fatorial\_recursivo(5)  
          5 \* fatorial\_recursivo(4)  
          4 \* fatorial\_recursivo(3)  
          3 \* fatorial\_recursivo(2)  
          2 \* fatorial\_recursivo(1)  
          **1**



# Fatorial recursivo

```
17 int fatorial_recursivo(int n)
18 {
19     if (n == 1)
20         return 1;
21
22     return n * fatorial_recursivo(n - 1);
23 }
```

**n** = 5 4 3 2 1

resultado = fatorial\_recursivo(5)  
          5 \* fatorial\_recursivo(4)  
          4 \* fatorial\_recursivo(3)  
          3 \* fatorial\_recursivo(2)  
          2 \* 1 = 2  
              1



# Fatorial recursivo

```
17 int fatorial_recursivo(int n)
18 {
19     if (n == 1)
20         return 1;
21
22     return n * fatorial_recursivo(n - 1);
23 }
```

**n** = 5 4 3 2 1

resultado = fatorial\_recursivo(5)  
          5 \* fatorial\_recursivo(4)  
          4 \* fatorial\_recursivo(3)  
          3 \* 2 = 6  
          2 \* 1 = 2  
          1



# Fatorial recursivo

```
17 int fatorial_recursivo(int n)
18 {
19     if (n == 1)
20         return 1;
21
22     return n * fatorial_recursivo(n - 1);
23 }
```

**n** = 5 4 3 2 1

resultado = fatorial\_recursivo(5)  
5 \* fatorial\_recursivo(4)  
4 \* 6 = 24  
3 \* 2 = 6  
2 \* 1 = 2  
1



# Fatorial recursivo

```
17 int fatorial_recursivo(int n)
18 {
19     if (n == 1)
20         return 1;
21
22     return n * fatorial_recursivo(n - 1);
23 }
```

**n = 5** 4 3 2 1

resultado = fatorial\_recursivo(5)

**5 \* 24 = 120**

4 \* 6 = 24

3 \* 2 = 6

2 \* 1 = 2

1



# Fatorial recursivo

```
17 int fatorial_recursivo(int n)
18 {
19     if (n == 1)
20         return 1;
21
22     return n * fatorial_recursivo(n - 1);
23 }
```

n = 5 4 3 2 1

resultado = 120 ←

5 \* 24 = 120

4 \* 6 = 24

3 \* 2 = 6

2 \* 1 = 2

1



# Para não esquecer!

- Toda função recursiva precisa ter uma condição de parada!
- Toda chamada recursiva de uma função deve ter valores de argumentos diferentes!

```
int fatorial_recursivo(int n)
{
    // Caso base???
    return n * fatorial_recursivo(n - 1);
}
```

```
int fatorial_recursivo(int n)
{
    if (n == 1)
        return 1;
    return n * fatorial_recursivo(n);
}
```

Exemplos de recursividades infinitas





# Problema

---

Implemente uma função [em C] que resolva a equação abaixo.  
Assuma que  $x$  é o valor retornado pela função.

$$x = \sum_{i=a}^b i$$

Alguns exemplos de valores e resultados:

$$a = 0; b = 9 \therefore x = 45$$

$$a = 2; b = 4 \therefore x = 9$$



## Possível solução (versão iterativa)

```
1  #include <stdio.h>
2
3  int somatoria(int a, int b)
4  {
5      int x = 0;
6      for (int i = a; i <= b; ++i)
7          x += i;
8      return x;
9  }
10
11 int main()
12 {
13     int a, b, x;
14
15     a = 0;
16     b = 9;
17     x = somatoria(a, b);
18     printf("somatoria(%d, %d) = %d\n", a, b, x);
19
20     a = 2;
21     b = 4;
22     x = somatoria(a, b);
23     printf("somatoria(%d, %d) = %d\n", a, b, x);
24 }
```

$$x = \sum_{i=a}^b i$$

```
somatoria(0, 9) = 45
somatoria(2, 4) = 9
```



# Problema

---

Como implementar uma função que resolve o mesmo problema, mas sem usar loops?

É possível!

Por exemplo, linguagens de programação puramente funcionais, como Haskell, não possuem estruturas de repetição.

Imagine... Programar sem loops!



## Possível solução (versão recursiva)

```
1  #include <stdio.h>
2
3  int somatoria(int a, int b)
4  {
5      if (a > b)
6          return 0;
7
8      return a + somatoria(a + 1, b);
9  }
10
11 int main()
12 {
13     int a, b, x;
14
15     a = 0;
16     b = 9;
17     x = somatoria(a, b);
18     printf("somatoria(%d, %d) = %d\n", a, b, x);
19
20     a = 2;
21     b = 4;
22     x = somatoria(a, b);
23     printf("somatoria(%d, %d) = %d\n", a, b, x);
24 }
```

$$x = \sum_{i=a}^b i$$

```
somatoria(0, 9) = 45
somatoria(2, 4) = 9
```



# Desafio

---

Escreva uma função recursiva **bool is\_palindrome(char str[])** que recebe uma string e retorna **true** se a string informada é um palíndromo(\*) ou **false** caso contrário.

- A verificação do palíndromo deve ser feita, obrigatoriamente, de forma recursiva.
- O código pode assumir que a string passada como argumento não tem acentos, sinais de pontuação e outros símbolos/caracteres especiais. Assim, "Olá, galo!" não precisa ser considerada palíndromo pelo seu código, mas "Ola galo", sim.

(\*) *“Palíndromo é uma palavra, número ou frase que ao ser feita a leitura da esquerda para a direita ou da direita para a esquerda permanece com o mesmo sentido.”* (referência:

<https://www.educamaisbrasil.com.br/enem/lingua-portuguesa/palindromo>).

Por exemplo, "Mussum" e "A sacada da casa" são palíndromos, mas "Bolton" e "Hello World" não são.



# Referências

MENOTTI, D.; OLIVEIRA, L. **CI-1002: Programação 2**. Disponível em: <<https://wiki.inf.ufpr.br/maziero/doku.php?id=prog2:start>>. Acesso em: 03 de janeiro de 2023.

DEITEL, P.; DEITEL, H. **C: Como programar**. 6ª edição. Editora Pearson, 2011. (*disponível na Biblioteca Virtual Pearson*)

KISHIMOTO, A. **Programação de computadores: desenvolvimento de jogos digitais com GameMaker: Studio**. 1ª edição. Edição do autor, 2016.



