Curso Estruturas de Dados e Algoritmos Expert

Prof. Nelio Alves

Recursividade

https://devsuperior.com.br

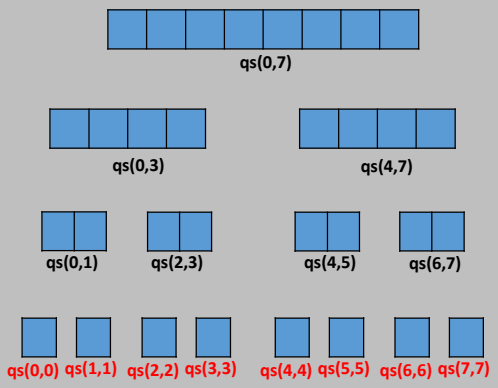
Recursividade

Recursividade é um conceito fundamental na programação e matemática, onde uma função chama a si mesma diretamente ou indiretamente para resolver um problema.

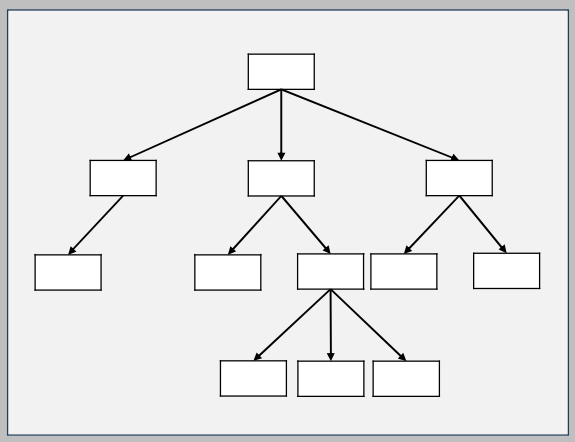
Função recursiva: uma função é dita recursiva se, durante a execução, ela chama a si mesma.

Exemplo de aplicação: algoritmos eficientes de ordenação (QuickSort e MergeSort)

Complexidade caso médio: *O(n log n)*



Exemplo de aplicação: estruturas de dados árvores e grafos



Exemplo de aplicação: problemas matemáticos

- Fatorial
- Fibonacci
- Máximo divisor comum

Exemplo de aplicação: programação funcional

Linguagens "puramente" funcionais (ou quase), tais como Haskell e Erlang, não possuem laços como *for* e *while*. Assim, algoritmos repetitivos precisam ser implementados com funções de alta ordem ou recursividade.

```
fatorial 0 = 1
fatorial n = n * fatorial (n - 1)
```

Resolvido 1: soma-naturais

Faça uma função que, dado um número natural N, retorne a soma dos números de 0 até N. Exemplos:

Exemplo 1

Entrada	Saída
0	0

Exemplo 2

Entrada	Saída
2	3

Exemplo 3

Entrada	Saída
4	10

Solução imperativa

```
function sum(n) {
    let total = 0;
    for (let i = 0; i <= n; i++) {
        total = total + i;
    }
    return total;
}

let result = sum(4);
console.log("RESULTADO: " + result);
```

Solução recursiva

ANÁLISE:

sum(0) = 0

sum(1) = 0+1 = 1 → **sum(1) = 1 + sum(0)**

sum(2) = 0+1+2 = 3 → **sum(2) = 2 + sum(1)**

sum(3) = 0+1+2+3 = 6 → **sum(3) = 3 + sum(2)**

sum(4) = 0+1+2+3+4 = 10 → **sum(4) = 4 + sum(3)**

sum(5) = 0+1+2+3+4+5 = 15 → **sum(5) = 5 + sum(4)**

...

sum(N) = N + sum(N - 1)

Solução recursiva

```
function sum(n) {
    if (n == 0) {
        return 0;
    }
    return n + sum(n - 1);
}

let result = sum(4);
console.log("RESULTADO: " + result);
```

Resolvido 2: fatorial

O **fatorial** de um número natural N é a multiplicação de 1 até N, exceto para o valor 0 (zero), cujo fatorial por definição é 1. Faça uma função para retornar o fatorial de um dado número.

Exemplo 1

Entrada	**Saída**
0	1

Exemplo 2

Entrada	**Saída**
3	6

Exemplo 3

Entrada	**Saída**
4	24

Solução imperativa

```
function factorial(n) {
    total = 1;
    for (let i = 1; i <= n; i++) {
        total = total * i;
    }
    return total;
}

let result = factorial(5);
console.log("RESULTADO: " + result);
```

Solução recursiva

ANÁLISE:

factorial(0) = 1

factorial(1) = 1 → **factorial(1) = 1 * factorial(0)**

factorial(2) = 1*2 = 2 → **factorial(2) = 2 * factorial(1)**

factorial(3) = 1*2*3 = 6 → **factorial(3) = 3 * factorial(2)**

factorial(4) = 1*2*3*4 = 24 → **factorial(4) = 4 * factorial(3)**

...

factorial(N) = N * factorial(N - 1)

Solução recursiva

```
function factorial(n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}

let result = factorial(4);
console.log("RESULTADO: " + result);
```

Vantagens da Recursividade

Adequação à natureza do problema:

Alguns problemas são naturalmente recursivos, tais como navegação em árvores e algumas funções matemáticas. Soluções recursivas nesses casos são geralmente claras e diretas.

Redução de código:

Algumas soluções recursivas podem ser bem mais concisas que as soluções equivalentes imperativas.

Vantagens da Recursividade

Soluções declarativas:

Soluções recursivas muitas vezes correspondem à definição declarativa da solução, descrevendo "o quê" é a solução, em oposição a descrever "como" executar os passos de um algoritmo.

Solução declarativas muitas vezes são elegantes e fáceis de entender.

```
fatorial 0 = 1
fatorial n = n * fatorial (n - 1)
```

Desvantagens da Recursividade

Consumo de memória:

Funções recursivas frequentemente usam mais memória devido à pilha de chamadas. Se a recursividade for muito profunda e o número de chamadas recursivas exceder a capacidade da pilha, pode ocorrer um erro de stack overflow.

Inadequação à natureza do problema:

Alguns problemas são naturalmente imperativos, e para eles geralmente soluções imperativas são mais fáceis de elaborar.

Casos base e casos recursivos

Caso base:

É a condição que interrompe a recursão.

É o ponto de parada para resolver diretamente o menor fragmento do problema.

Sem um caso base apropriado, a função recursiva continuará chamando a si mesma indefinidamente, levando a um loop infinito ou a um erro de stack overflow.

```
fatorial 0 = 1
fatorial n = n * fatorial (n - 1)
```

Casos base e casos recursivos

Caso recursivo:

É a parte da função que inclui uma ou mais chamadas para a própria função, mas com argumentos que se aproximam do caso base.

Cada chamada recursiva deve alterar os argumentos de tal forma que eles se aproximem do caso base.

```
fatorial 0 = 1
fatorial n = n * fatorial (n - 1)
```


Pilha de chamadas

A pilha de chamadas é uma estrutura de dados usada pelo sistema operacional e pelo ambiente de execução de linguagens de programação para gerenciar a execução de funções.

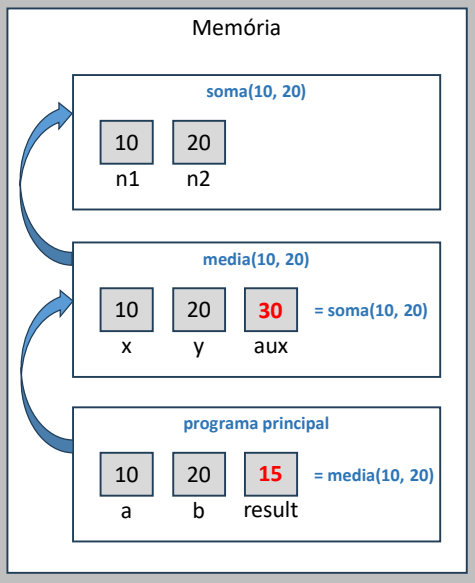
Pilha de chamadas

```
let a = 10;
let b = 20;

let result = media(a, b);
console.log("RESULTADO: " + result);

function media(x, y) {
    let aux = soma(x, y);
    return aux / 2;
}

function soma(n1, n2) {
    return n1 + n2;
}
```

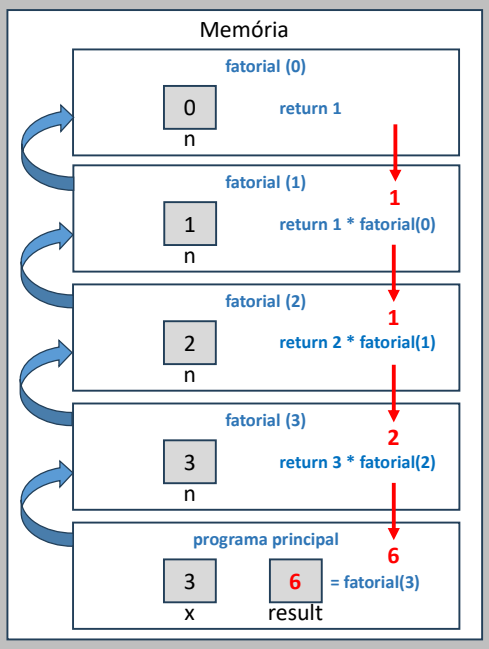


Pilha de chamadas

```
let x = 3;

let result = fatorial(x);
console.log("RESULTADO: " + result);

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



Mas atenção!

Entender o funcionamento da pilha de chamadas é importante para que o programador tenha cuidado com o consumo de memória ao elaborar uma solução recursiva.

Porém, para elaborar uma solução recursiva, **não** foque seu raciocínio na sequência de passos de execução!

Foque em descobrir os **casos base** e os **casos recursivos**.

```
fatorial(0) = 1
fatorial(n) = n * fatorial(n - 1)
```


Recursividade de cauda

Recursividade de cauda é uma técnica que consiste em deixar a chamada recursiva como a **última operação** realizada antes da função retornar um resultado.

Em outras palavras: nenhum processamento deve ser feito na função depois da chamada recursiva.

Recursividade de cauda

Se o compilador/interpretador da linguagem for devidamente preparado para recursividade de cauda, a execução do programa será otimizada, sem a necessidade de manter informações sobre as chamadas anteriores na pilha de chamadas.

Exemplos: Haskell, Skala

Compiladores de linguagens não "puramente" funcionais geralmente não oferecem essa otimização por padrão.

Recursividade de cauda

Outra vantagem:

Soluções alternativas e mais otimizadas também podem ser construídas com a recursividade de cauda.

Exemplo: Fibonacci exponencial vs. Fibonacci linear usando função auxiliar com parâmetros adicionais.

(veremos em breve)

Solução recursiva sem recursividade de cauda

factorial(0) = 1

factorial(1) = 1 → **factorial(1) = 1 * factorial(0)**

factorial(2) = 1*2 = 2 → **factorial(1) = 2 * factorial(1)**

factorial(3) = 1*2*3 = 6 → **factorial(1) = 3 * factorial(2)**

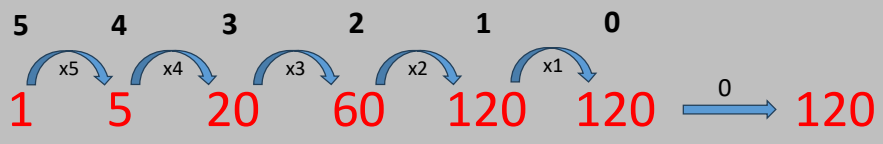
factorial(4) = 1*2*3*4 = 24 → **factorial(4) = 4 * factorial(3)**

...

```
function factorial(n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

27

Ideia da solução com recursividade de cauda: manter o cálculo intermediário como um **parâmetro adicional** da função.




```
factorial(5, 1)
= factorial(4, 5)
= factorial(3, 20)
= factorial(2, 60)
= factorial(1, 120)
= factorial(0, 120)
= 120
```

```
factorial(4, 1)
= factorial(3, 4)
= factorial(2, 12)
= factorial(1, 24)
= factorial(0, 24)
= 24
```

```
factorial(0, 1)
= 1
```

28

Solução do problema fatorial com recursividade de cauda

```
function factorial(n) {
  return factorialTailRecursive(n, 1);
}

function factorialTailRecursive(n, total) {
  if (n === 0) {
    return total;
  }
  return factorialTailRecursive(n - 1, n * total);
}
```


Resolvido 3: fibonacci

A sequência de Fibonacci começa com 0, 1, e depois cada número é a soma de seus dois antecessores: 0 1 1 2 3 5 8 13...

Faça uma função para retornar o valor de uma dada posição da sequência de Fibonacci. Exemplos:

Exemplo 1

Entrada	**Saída**
0	0

Exemplo 2

Entrada	**Saída**
1	1

Exemplo 3

Entrada	**Saída**
6	8

Solução recursiva (spoiler: solução ineficiente)

Sequência Fibonacci: **0** **1** **1** **2** **3** **5** **8** **13** **...**

0 1 2 3 4 5 6 7

fib(0) = 0

fib(1) = 1

fib(2) = 0+1 = 1 → **fib(2) = fib(1) + fib(0)**

fib(3) = 1+1 = 2 → **fib(3) = fib(2) + fib(1)**

fib(4) = 1+2 = 3 → **fib(4) = fib(3) + fib(2)**

fib(5) = 2+3 = 5 → **fib(5) = fib(4) + fib(3)**

fib(6) = 3+5 = 8 → **fib(6) = fib(5) + fib(4)**

...

fib(N) = fib(N - 1) + fib(N - 2)

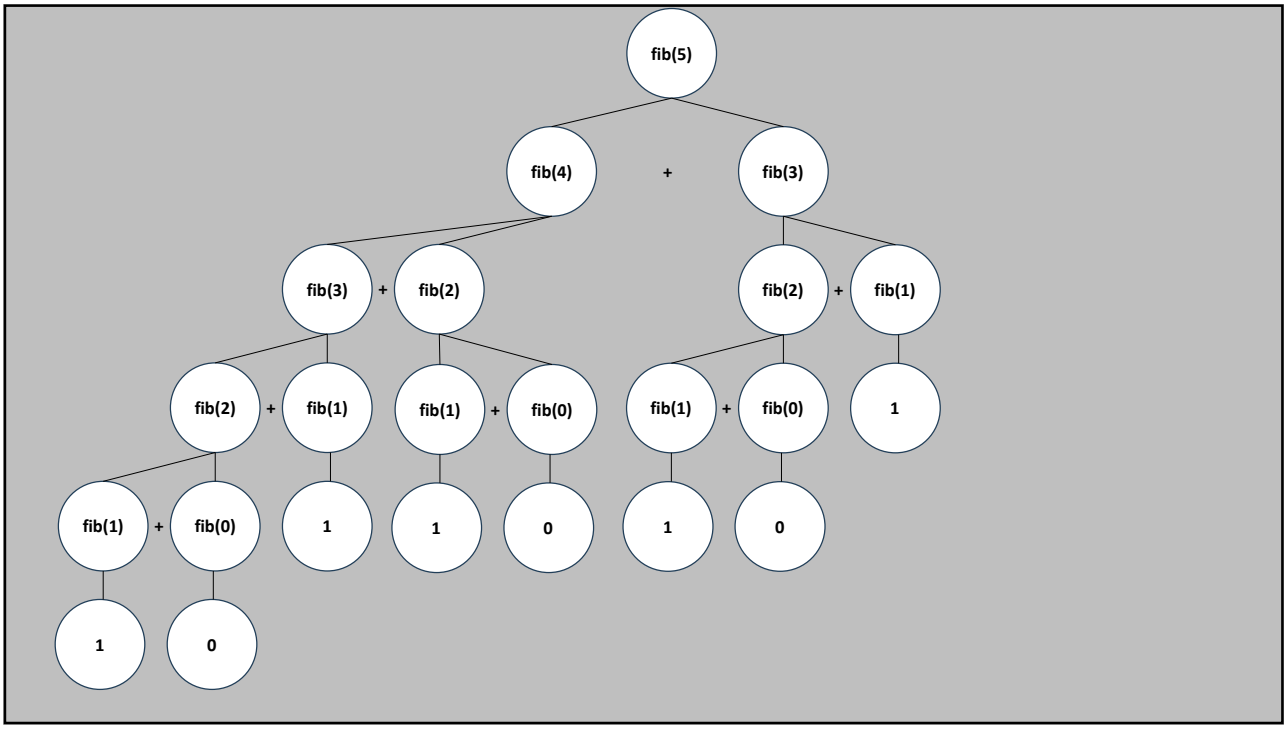
31

Solução recursiva (spoiler: solução ineficiente)

```
function fib(n) {
  if (n == 0) {
    return 0;
  }
  if (n == 1) {
    return 1;
  }
  return fib(n - 1) + fib(n - 2);
}

const result = fib(6);
console.log("RESULTADO = " + result);
```

32



Recursividade de cauda com **parâmetros adicionais**

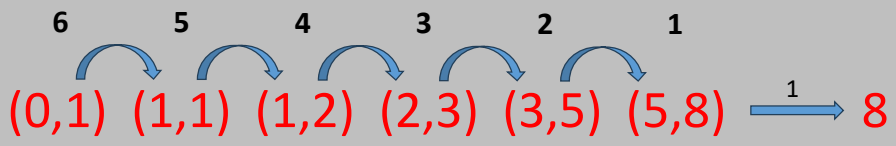
6 5 4 3 2 1

(0,1) (1,1) (1,2) (2,3) (3,5) (5,8) —1→ 8

0 1 1 2 3 5 8 13 ...

0 1 2 3 4 5 6 7 ...

Recursividade de cauda com **parâmetros adicionais**




```
fib(6, 0, 1)
= fib(5, 1, 1)
= fib(4, 1, 2)
= fib(3, 2, 3)
= fib(2, 3, 5)
= fib(1, 5, 8)
= 8
```

```
fib(7, 0, 1)
= fib(6, 1, 1)
= fib(5, 1, 2)
= fib(4, 2, 3)
= fib(3, 3, 5)
= fib(2, 5, 8)
= fib(1, 8, 13)
= 13
```

```
fib(0, 0, 1)
= 0
```

```
fib(1, 0, 1)
= 1
```


Solução com recursividade de cauda

```
function fib(n) {
  return fibTailRecursive(n, 0, 1);
}

function fibTailRecursive(n, a, b) {
  if (n == 0) {
    return a;
  }
  if (n == 1) {
    return b;
  }
  return fibTailRecursive(n - 1, b, a + b);
}

const result = fib(6);
console.log("RESULTADO = " + result);
```

Conceitos de cabeça e cauda de uma lista:

$[e_1, e_2, e_3, \ldots, e_N]$

Cabeça = e_1

Cauda = $[e_2, e_3, \ldots, e_N]$

```
Exemplo 1: ["azul", "verde", "amarelo"]
Cabeça = "azul"
Cauda = ["verde", "amarelo"]
```

```
Exemplo 2: ["azul"]
Cabeça = "azul"
Cauda = []
```

```
Exemplo 3: []
Cabeça = indefinido
Cauda = indefinido
```


Resolvido 4: reverse

Faça uma função que receba uma lista (de qualquer tipo) e retorne a lista reversa.

Exemplos:

Exemplo 1

Entrada	Saída
[ ]	[ ]

Exemplo 2

Entrada	Saída
["azul"]	["azul"]

Exemplo 3

Entrada	Saída
["azul", "verde", "preto", "rosa"]	["rosa", "preto, "verde", "azul"]

Solução imperativa

```
function reverse(list) {
    let newList = [];
    for (let i = list.length - 1; i >= 0 ; i--) {
        newList.push(list[i]);
    }
    return newList;
}

let result = reverse(["azul", "verde", "preto", "rosa"]);
console.log(result);
```


Solução recursiva

ANÁLISE:

```
reverse([]) = []

reverse(["rosa"]) = ["rosa"]

reverse(["preto", "rosa"]) = ["rosa", "preto"]

reverse(["verde", "preto", "rosa"]) = ["rosa", "preto", "verde"]

reverse(["azul", "verde", "preto", "rosa"]) = ["rosa", "preto", "verde", "azul"]
...
```

$$reverse([e_1, e_2, e_3, \ldots, e_N]) = reverse([e_2, e_3, \ldots, e_N]) + [e_1]$$

Solução recursiva (versão 1)

```
function reverse(list) {
    if (list.length <= 1) {
        return list;
    }

    const head = list[0];
    const tail = list.slice(1);

    return reverse(tail).concat([head]);
}

let result = reverse(["azul", "verde", "preto", "rosa"]);
console.log(result);
```


Solução recursiva (versão 2)

```
function reverse(list) {
    if (list.length <= 1) {
        return list;
    }

    const head = list[0];
    const tail = list.slice(1);

    const newList = reverse(tail);
    newList.push(head);

    return newList;
}

let result = reverse(["azul", "verde", "preto", "rosa"]);
console.log(result);
```

Exercícios propostos

Acesse a lista de exercícios do curso para treinar recursividade com os exercícios propostos.