Técnicas de Programação 1

9ª parte

Recursividade

Prof. Jobson Massollar

jobson@uniriotec.br

Recursividade

Existem problemas cuja definição é elaborada a partir dele mesmo.

Exemplo: fatorial

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

Exemplo:

$$fat(4) = 4 \times 3 \times 2 \times 1$$

$$fat(5) = 5 \times 4 \times 3 \times 2 \times 1$$

$$fat(6) = 6 \times 5 \times 4 \times 3 \times 2 \times 1$$

Exemplo:

$$fat(4) = 4 \times 3 \times 2 \times 1$$

$$fat(5) = 5 \times 4 \times 3 \times 2 \times 1$$

$$fat(6) = 6 \times 5 \times 4 \times 3 \times 2 \times 1$$

Exemplo:

$$fat(4) = 4 \times 3 \times 2 \times 1$$

$$fat(5) = 5 \times 4 \times 3 \times 2 \times 1 \qquad \Rightarrow fat(5) = 5 \times fat(4)$$

$$fat(6) = 6 \times 5 \times 4 \times 3 \times 2 \times 1 \qquad \Rightarrow fat(6) = 6 \times fat(5)$$

Exemplo:

$$fat(4) = 4 \times 3 \times 2 \times 1$$

$$fat(5) = 5 \times 4 \times 3 \times 2 \times 1 \qquad \Rightarrow fat(5) = 5 \times fat(4)$$

$$fat(6) = 6 \times 5 \times 4 \times 3 \times 2 \times 1 \qquad \Rightarrow fat(6) = 6 \times fat(5)$$

Generalizando temos:

$$fat(n) = n \times fat(n-1)$$

Note que essa é uma definição **recursiva**, ou seja, a definição do fatorial é feita em função do próprio fatorial!

Recursividade

Toda definição recursiva possui pelo menos um caso-base.

Caso-base é o caso de **solução trivial** do problema, onde não é mais necessário utilizar a definição recursiva.

Qual o caso-base do fatorial?

Foto criada por katemangostar - br.freepik.com

Caso-base do fatorial:

$$fat(5) = 5 \times fat(4)$$

$$fat(4) = 4 \times fat(3)$$

$$fat(3) = 3 \times fat(2)$$

$$fat(2) = 2 \times fat(1)$$

$$fat(1) = 1 \times fat(0)$$

$$fat(0) = 1$$

Note que o fatorial de **ZERO** é **UM** por definição, ou seja, a definição de fat(0) não usa mais a recursão.

Assim, a definição do fatorial pode ser descrita como:

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

Generalizando, podemos dizer que toda definição recursiva é composta de:

a) Um ou mais casos-base (situações onde o problema possui uma solução trivial não recursiva).

b) Uma ou mais definições recursivas (definições da solução do problema em função dele mesmo).

Iteração x Recursão

Quando precisamos repetir um conjunto de comandos temos, geralmente, duas formas de construir a solução:

Algoritmo Iterativo	Algoritmo Recursivo
É aquele que repete um conjunto de comandos usando estruturas de repetição implementadas nas linguagens de programação como for, while, do..while, etc.	É aquele que repete um conjunto de comandos usando chamadas recursivas, ou seja, um procedimento ou função que chama ele mesmo.

Recursividade no C/C++

O C/C++ permite que uma função chame **ela mesma**.

Quando usamos esse tipo de recurso dizemos que a nossa função é **recursiva**.

Funções recursivas podem ser úteis quando a definição do problema também é recursiva.

Recursividade no C/C++

Função para cálculo do fatorial de n:

```
long long fat(long n) {
  if (n == 0)
    return 1;

  return n * fat(n-1);
}
```

Caso-base

Definição recursiva

Recursividade no C/C++

Simulação do algoritmo para n = 5:

fat(5)

Recursividade no C/C++

Simulação do algoritmo para n = 5:

fat(5)

5 * fat(4)

Recursividade no C/C++

Simulação do algoritmo para n = 5:

fat(5)

5 * fat(4)

4 * fat(3)

Recursividade no C/C++

Simulação do algoritmo para n = 5:

fat(5)

5 * fat(4)

4 * fat(3)

3 * fat(2)

Recursividade no C/C++

Simulação do algoritmo para n = 5:

fat(5)

5 * fat(4)

4 * fat(3)

3 * fat(2)

2 * fat(1)

Recursividade no C/C++

Simulação do algoritmo para n = 5:

fat(5)

5 * fat(4)

4 * fat(3)

3 * fat(2)

2 * fat(1)

1 * <u>fat(0)</u>

Simulação do algoritmo para n = 5:

fat(5)

5 * fat(4)

4 * fat(3)

3 * fat(2)

2 * fat(1) 1

1 * fat(0)

Simulação do algoritmo para n = 5:

fat(5)

5 * fat(4)

4 * fat(3)

3 * fat(2) 2

2 * fat(1) 1

1 * <u>fat(0)</u>

Recursividade no C/C++

Simulação do algoritmo para n = 5:

fat(5)

5 * fat(4)

4 * fat(3) 6

3 * fat(2) 2

2 * fat(1) 1

1 * <u>fat(0)</u>

Recursividade no C/C++

Simulação do algoritmo para n = 5:

fat(5)

5 * fat(4) 24

4 * fat(3) 6

3 * fat(2) 2

2 * fat(1) 1

1 * fat(0)

Recursividade no C/C++

Simulação do algoritmo para n = 5:

fat(5) 120

5 * fat(4) 24

4 * fat(3) 6

3 * fat(2) 2

2 * fat(1) 1

1 * <u>fat(0)</u>

Recursividade no C/C++

A série de Fibonacci é constituída de uma sequência de números inteiros, na qual:

a) O primeiro número é sempre 0.

b) O segundo número é sempre 1.

c) Os demais números são formados pela soma dos seus dois imediatamente antecessores.

0 1

A série de Fibonacci é constituída de uma sequência de números inteiros, na qual:

a) O primeiro número é sempre 0.

b) O segundo número é sempre 1.

c) Os demais números são formados pela soma dos seus dois imediatamente antecessores.

$0 + 1 = 1$

A série de Fibonacci é constituída de uma sequência de números inteiros, na qual:

a) O primeiro número é sempre 0.

b) O segundo número é sempre 1.

c) Os demais números são formados pela soma dos seus dois imediatamente antecessores.

0 1 + 1 = 2

Recursividade no C/C++

A série de Fibonacci é constituída de uma sequência de números inteiros, na qual:

a) O primeiro número é sempre 0.

b) O segundo número é sempre 1.

c) Os demais números são formados pela soma dos seus dois imediatamente antecessores.

0 1 1 + 2 = 3

A série de Fibonacci é constituída de uma sequência de números inteiros, na qual:

a) O primeiro número é sempre 0.

b) O segundo número é sempre 1.

c) Os demais números são formados pela soma dos seus dois imediatamente antecessores.

0 1 1 2 + 3 = 5

A série de Fibonacci é constituída de uma sequência de números inteiros, na qual:

a) O primeiro número é sempre 0.

b) O segundo número é sempre 1.

c) Os demais números são formados pela soma dos seus dois imediatamente antecessores.

0 1 1 2 3 + 5 = 8

A série de Fibonacci é constituída de uma sequência de números inteiros, na qual:

a) O primeiro número é sempre 0.

b) O segundo número é sempre 1.

c) Os demais números são formados pela soma dos seus dois imediatamente antecessores.

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

Assim, o n-ésimo número da série de Fibonacci pode ser calculado como:

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

Recursividade no C/C++

Função para cálculo do n-ésimo número de Fibonacci:

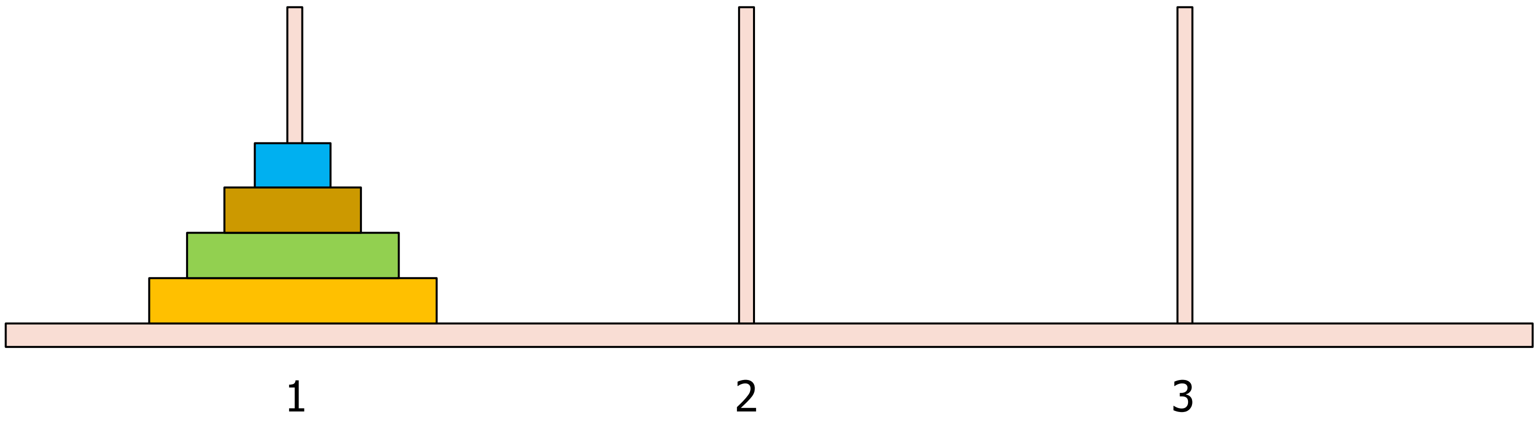
```
int fib(int n) {
  if (n == 1)
    return 0;

  if (n == 2)
    return 1;

  return fib(n-1) + fib(n-2);
}
```

Casos-base

Definição recursiva

Recursividade no C/C++

Torre de Hanoi: levar todos os discos da origem (pino 1) para o destino (pino 3) usando o pino auxiliar (pino 2).

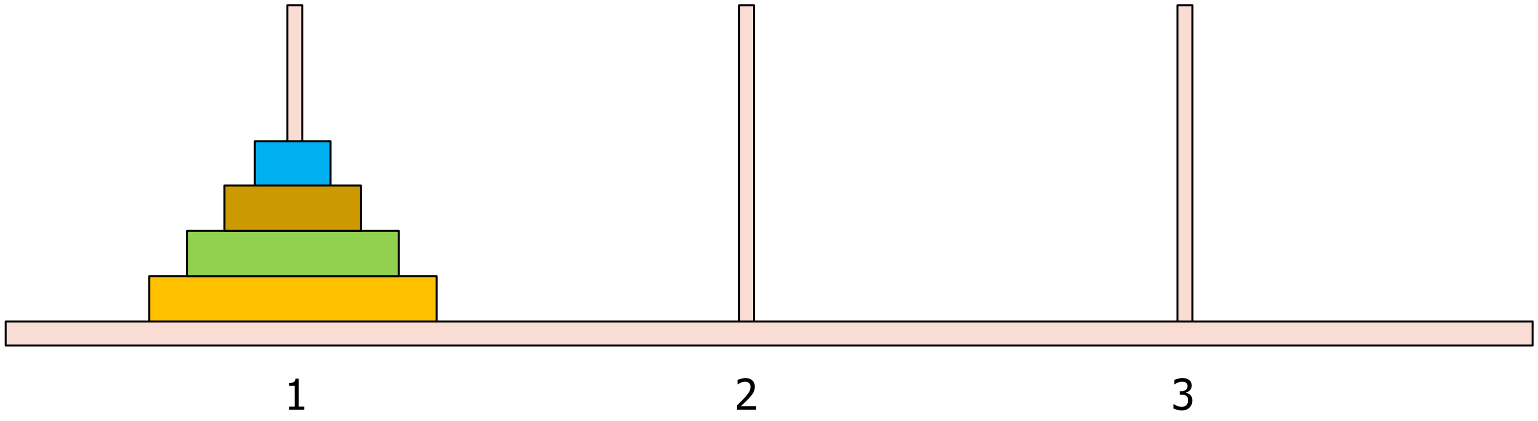


Regras:

- ✓ Só pode mover um disco por vez.
- ✓ Não pode colocar um disco maior sobre um disco menor.

A solução da Torre de Hanoi é um algoritmo recursivo:

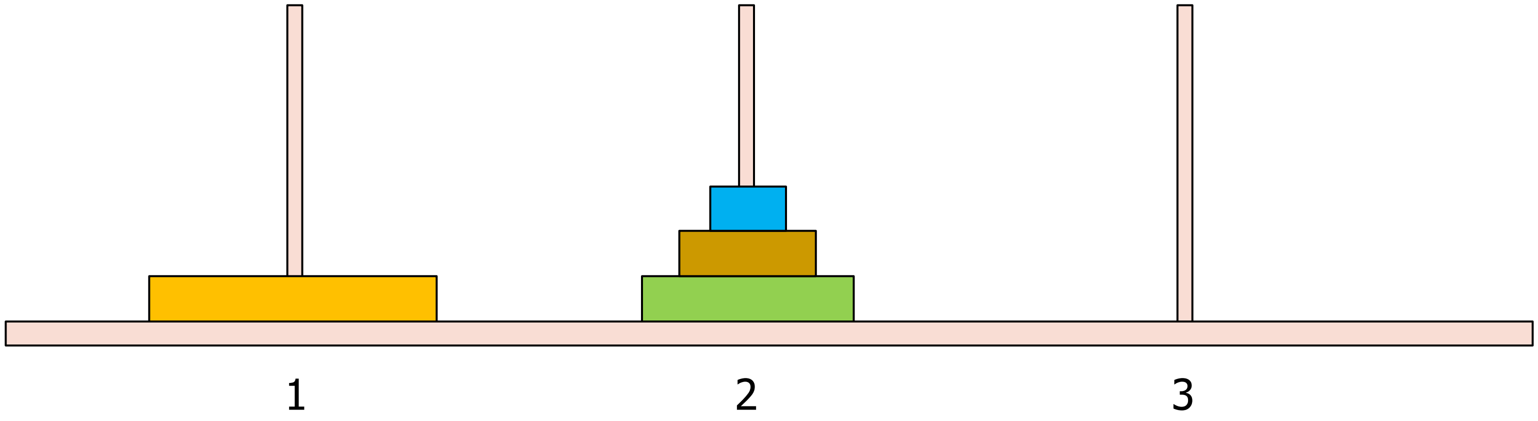
```
hanoi(número de discos, origem, destino, auxiliar)
    se número de discos = 1 então
      move o disco da origem para o destino
    senão
      hanoi(número de discos-1, origem, auxiliar, destino)
      move o disco da origem para o destino
      hanoi(número de discos-1, auxiliar, destino, origem)
    fim-se
```



Recursividade no C/C++

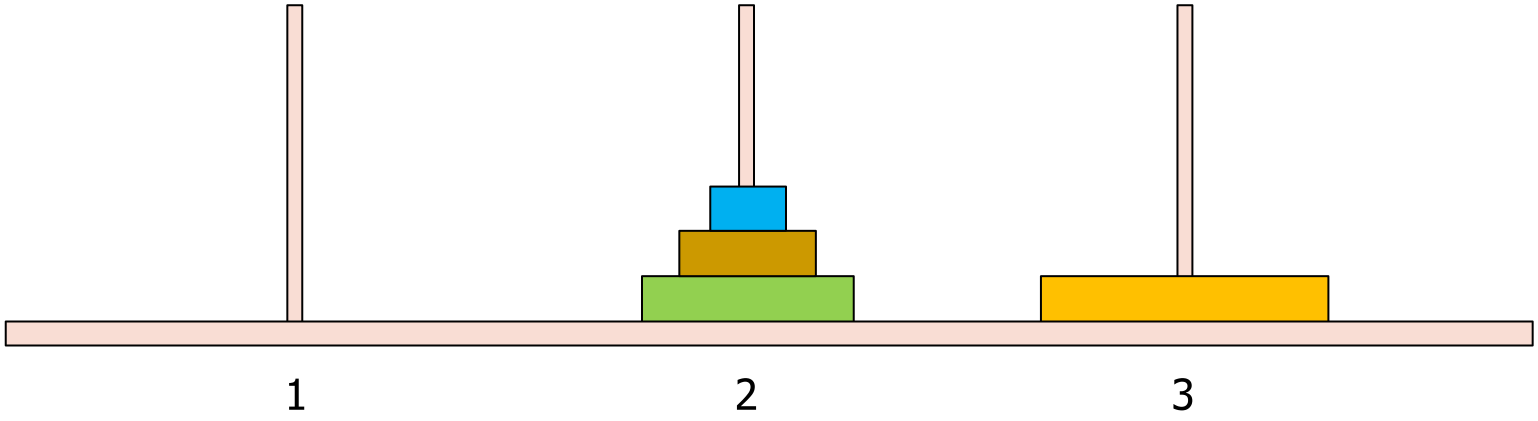
A solução da Torre de Hanoi é um algoritmo recursivo:

```
hanoi(número de discos, origem, destino, auxiliar)
    se número de discos = 1 então
      move o disco da origem para o destino
    senão
      hanoi(número de discos-1, origem, auxiliar, destino)
      move o disco da origem para o destino
      hanoi(número de discos-1, auxiliar, destino, origem)
    fim-se
```



A solução da Torre de Hanoi é um algoritmo recursivo:

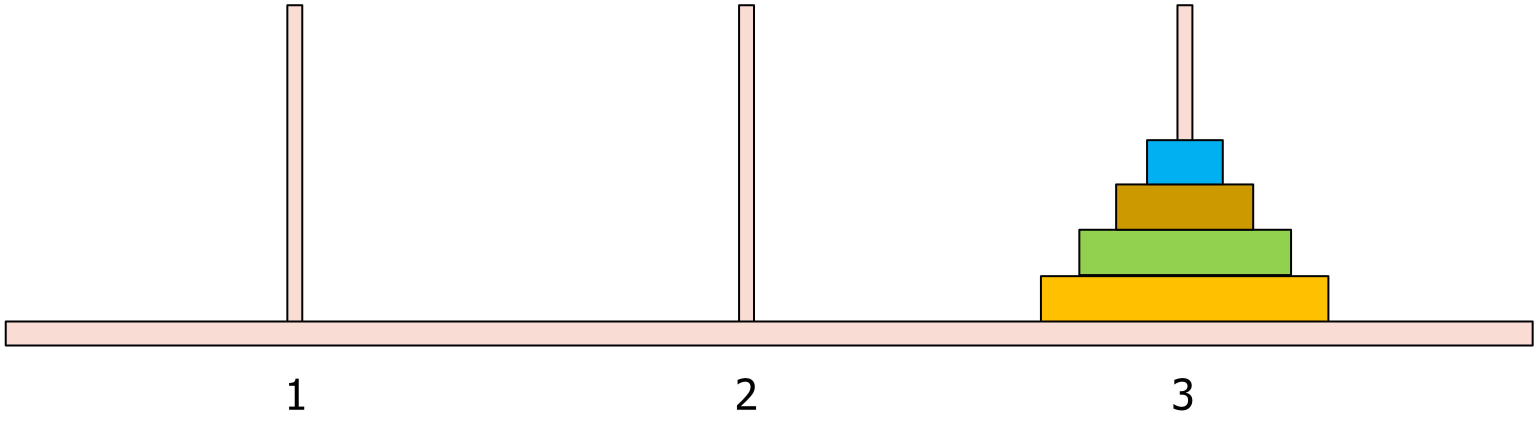
```
hanoi(número de discos, origem, destino, auxiliar)
    se número de discos = 1 então
      move o disco da origem para o destino
    senão
      hanoi(número de discos-1, origem, auxiliar, destino)
      move o disco da origem para o destino
      hanoi(número de discos-1, auxiliar, destino, origem)
    fim-se
```



Recursividade no C/C++

A solução da Torre de Hanoi é um algoritmo recursivo:

```
hanoi(número de discos, origem, destino, auxiliar)
    se número de discos = 1 então
      move o disco da origem para o destino
    senão
      hanoi(número de discos-1, origem, auxiliar, destino)
      move o disco da origem para o destino
      hanoi(número de discos-1, auxiliar, destino, origem)
    fim-se
```



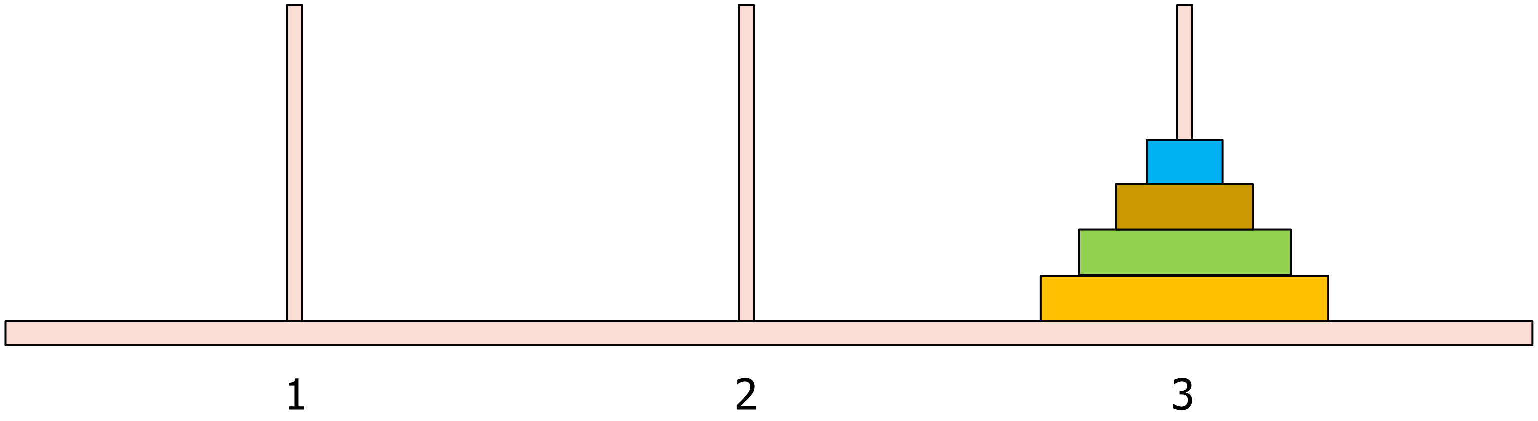
Recursividade no C/C++

A solução da Torre de Hanoi é um algoritmo recursivo:

```
hanoi(número de discos, origem, destino, auxiliar)
    se número de discos = 1 então
      move o disco da origem para o destino
    senão
      hanoi(número de discos-1, origem, auxiliar, destino)
      move o disco da origem para o destino
      hanoi(número de discos-1, auxiliar, destino, origem)
    fim-se
```

Caso-base

Definição recursiva



Quando o último comando de uma função é uma chamada recursiva dizemos que temos uma recursão de cauda.

```
long long fat(long n) {
  if (n == 0)
    return 1;

  return n * fat(n-1);
}
```

Recursividade

Toda recursão de cauda pode ser facilmente reescrita como um algoritmo iterativo.

```
long long fat(long n) {
  if (n == 0)
    return 1;

  return n * fat(n-1);
}
```

```
long long fat(long n) {
  long f = 1;

  while (n > 0) {
    f = f * n;
    n--;
  }

  return f;
}
```

Importante!

A recursividade é um recurso importante, mas deve ser usada com cuidado:

- ✓ Algoritmos iterativos tendem a ter melhor desempenho, pois chamadas de função demandam mais tempo da CPU e alocam memória para controlar o retorno da função.
- ✓ Alguns problemas são mais fáceis de serem implementados na sua forma recursiva (Torre de Hanoi, algoritmos de árvores, dentre outros).

9.1) O cálculo de x^n pode ser definido como:

$$x^n = \begin{cases} 1 & , n = 0 \\ x \, . \, x^{n-1} & , n > 0 \end{cases}$$

Implemente a função recursiva potencia que calcula x^n. Caso $n < 0$ retorne -1.

9.2) Implemente a função recursiva soma que soma dois números inteiros a e b. Atente para o fato de que a ou b podem ser negativos. Dica: escreva primeiramente a definição recursiva da soma e depois implemente a função.