

RECURSÃO

Conceito

2

- Conceito básico: sub-rotina ou função que chama a si mesma.
- Definição: 'é o processo de **definir algo em termos de si mesmo**, também chamado de *definição circular*.'
- Uso: descrever um procedimento de forma generalizada, para solucionar **partições** do problema.
- Em outras palavras: resolver problemas menores em cada chamada de função.

Função recursiva

3

- Resolução recursiva:
 - ▣ Definir o conjunto de parâmetros necessário;
 - ▣ Definir um critério de parada (quando a função deve parar de chamar a si mesma);
 - ▣ Definir os retornos para cada caso (próximas instruções);
- Funções recursivas utilizam o retorno como um “aviso”.
 - ▣ Cada função chamada resolve um problema parcial, e retorna o seu resultado para quem a chamou.

Fatorial

4

- Produto dos números inteiros consecutivos de 1 até n , definido por $n!$:
 - $3! : 3*2*1 = 6;$
 - $5! : 5*4*3*2*1 = 120;$
 - $10! : 10*9*8*7*6*5*4*3*2*1 = 3628800;$
- Em um processo *iterativo*:

```
int fatorial(int n)
{
    int i, fat = 1;
    for(i = 1; i <= n; i++)
    {
        fat = fat * i;
    }
    return fat;
}
```

Fatorial

5

- Para calcular 5!:

fat = 1

i

```
int fatorial(int n)
{
    int i, fat = 1;
    for(i = 1; i <= n; i++)
    {
        fat = fat * i;
    }
    return fat;
}
```

Fatorial

6

- Para calcular 5!:

fat = (1*1)

i = 1

```
int fatorial(int n)
{
    int i, fat = 1;
    for(i = 1; i <= n; i++)
    {
        fat = fat * i;
    }
    return fat;
}
```

Fatorial

7

- Para calcular 5!:

fat = ((1*1) * 2)
i = 2

```
int fatorial(int n)
{
    int i, fat = 1;
    for(i = 1; i <= n; i++)
    {
        fat = fat * i;
    }
    return fat;
}
```

Fatorial

8

- Para calcular 5!:

$$\text{fat} = (((1 * 1) * 2) * 3) \\ i = 3$$

```
int fatorial(int n)
{
    int i, fat = 1;
    for(i = 1; i <= n; i++)
    {
        fat = fat * i;
    }
    return fat;
}
```


Fatorial

9

- Para calcular 5!:

fat = (((1*1) * 2) * 3) * 4)
i = 4

```
int fatorial(int n)
{
    int i, fat = 1;
    for(i = 1; i <= n; i++)
    {
        fat = fat * i;
    }
    return fat;
}
```

Fatorial

10

- Para calcular 5!:

$$\text{fat} = (((((1 * 1) * 2) * 3) * 4) * 5)$$

$i = 5$

```
int fatorial(int n)
{
    int i, fat = 1;
    for(i = 1; i <= n; i++)
    {
        fat = fat * i;
    }
    return fat;
}
```

Fatorial

11

- Para calcular 5!:

$$5! = (((((1 * 1) * 2) * 3) * 4) * 5)$$

```
int fatorial(int n)
{
    int i, fat = 1;
    for(i = 1; i <= n; i++)
    {
        fat = fat * i;
    }
    return fat;
}
```

Fatorial

12


□ Para calcular 5!:

$$5! = (((((1 * 1) * 2) * 3) * 4) * 5)$$

Fatorial

13

□ Para calcular 5!:

$$5! = (((((1 * 1) * 2) * 3) * 4) * 5)$$


Fatorial

14

□ Para calcular 5!:

$$5! = \overbrace{(((1 * 1) * 2) * 3) * 4}^{4!} * 5$$

Fatorial

15

□ Para calcular 5!:

$$5! = (((((1 * 1) * 2) * 3) * 4) * 5)$$

3!

Fatorial

16

□ Para calcular 5!:

$$5! = (((((1 * 1) * 2) * 3) * 4) * 5)$$

Fatorial

17

□ Para calcular 5!:

$$5! = (((((1 * 1) * 2) * 3) * 4) * 5)$$



Fatorial

18

- Para calcular $5!$:

$$5! = (((((1 * 1) * 2) * 3) * 4) * 5)$$

The diagram illustrates the recursive calculation of $5!$ using nested parentheses and colored brackets. The expression is $5! = (((((1 * 1) * 2) * 3) * 4) * 5)$. Brackets and factorial symbols above the expression show the recursive steps: $1!$ for $(1 * 1)$, $2!$ for $((1 * 1) * 2)$, $3!$ for $((((1 * 1) * 2) * 3)$, $4!$ for $(((((1 * 1) * 2) * 3) * 4)$, and $5!$ for $((((((1 * 1) * 2) * 3) * 4) * 5)$. The colors of the brackets and numbers correspond to the factorial being calculated at each step: $1!$ is red, $2!$ is blue, $3!$ is pink, $4!$ is yellow, and $5!$ is green.

O fatorial de um número está definido em função dos fatoriais de outros números.

Fatorial

19


- Para calcular 5!:

$$5! = (((((1 * 1) * 2) * 3) * 4) * 5)$$

Fatorial

20

□ Para calcular 5!:

$$5! = (((((1 * 1) * 2) * 3) * 4) * 5)$$



A green bracket is drawn under the expression $(((((1 * 1) * 2) * 3) * 4) * 5)$. The bracket starts under the first closing parenthesis and ends under the last closing parenthesis. Below the center of the bracket is the label $5!$.

$$5! = 5 * 4!;$$

Fatorial

21

□ Para calcular 5!:

$$5! = (((((1 * 1) * 2) * 3) * 4) * 5)$$



$$5! = 5 * 4!;$$

$$4! = 4 * 3!;$$

Fatorial

22

□ Para calcular 5!:

$$5! = (((((1 * 1) * 2) * 3) * 4) * 5)$$


3!

$$5! = 5 * 4!;$$


$$4! = 4 * 3!;$$

$$3! = 3 * 2!;$$

Fatorial

23

□ Para calcular 5!:

$$5! = (((((1 * 1) * 2) * 3) * 4) * 5)$$


$$5! = 5 * 4!;$$

$$4! = 4 * 3!;$$

$$3! = 3 * 2!;$$


$$2! = 2 * 1!;$$

Fatorial

24

□ Para calcular 5!:

$$5! = (((((1 * 1) * 2) * 3) * 4) * 5)$$



$$5! = 5 * 4!;$$

$$4! = 4 * 3!;$$

$$3! = 3 * 2!;$$

$$2! = 2 * 1!;$$

$$1! = 1;$$

Fatorial

25

□ Para calcular 5!:

$$5! = (((((1 * 1) * 2) * 3) * 4) * 5)$$

$$5! = 5 * 4!;$$

$$4! = 4 * 3!;$$

$$3! = 3 * 2!;$$

$$2! = 2 * 1!;$$

Critério de parada $\rightarrow 1! = 1;$

Fatorial

26

□ Para calcular 5!:

$$5! = (((((1 * 1) * 2) * 3) * 4) * 5)$$

$$5! = 5 * 4!;$$

$$4! = 4 * 3!;$$

$$3! = 3 * 2!;$$

$$2! = 2 * 1!;$$

Árvore de recursão:
expansão das chamadas
recursivas.

Critério de parada $\rightarrow 1! = 1;$

Fatorial

27

- Em um processo *recursivo*:

```
int fatorialR(int n)
{
    if (n == 1) return 1;

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

Fatorial

28

□ Em um processo *recursivo*:

```
int fatorialR(int n) → Definição dos parâmetros
{
    if (n == 1) return 1; → Critério de parada

    return n * fatorialR(n - 1); → Retorno de cada caso
}
```

Pilha da memória

29

- A recursão consiste em funções aninhadas.
- Suponha que a função fatorial $R()$ seja $F(n)$.
 - ▣ $F(4) = 4 * F(3)$.
- A função $F(4)$ fica em pausa esperando o retorno de $F(3)$.
- $F(3)$ cria novas variáveis, chama $F(2)$ e entra em pausa esperando $F(2)$ retornar...
- Nesse procedimento, o sistema está **empilhando** funções e suas variáveis na memória.

Pilha da memória

30

- Ao atingir o critério de parada, é realizado um retorno sem novas chamadas de função.
- Quando $F(1)$ retorna ela é desalocada da pilha do sistema.
- $F(2)$ volta a executar e recebe o retorno.
- $F(2)$ realiza sua operação e retorna um valor para $F(3)$...
- Sequencialmente, na ordem que foram chamadas, acontece o “desempilhamento” das funções, até atingir a chamada inicial de $F(4)$.

Torre de Hanoi

31

- Problema consiste em três pinos e n discos;
- Cada disco possui um tamanho diferente;
- Os discos iniciam empilhados do maior para o menor.



Torre de Hanoi

32

- O objetivo é transportar a pilha de um pino para outro, seguindo algumas regras:
 - ▣ Só é possível mover um disco por vez;
 - ▣ Só é possível mover discos do topo de uma pilha;
 - ▣ O disco só pode ser colocado acima de um disco maior que ele, ou em um pino vazio.

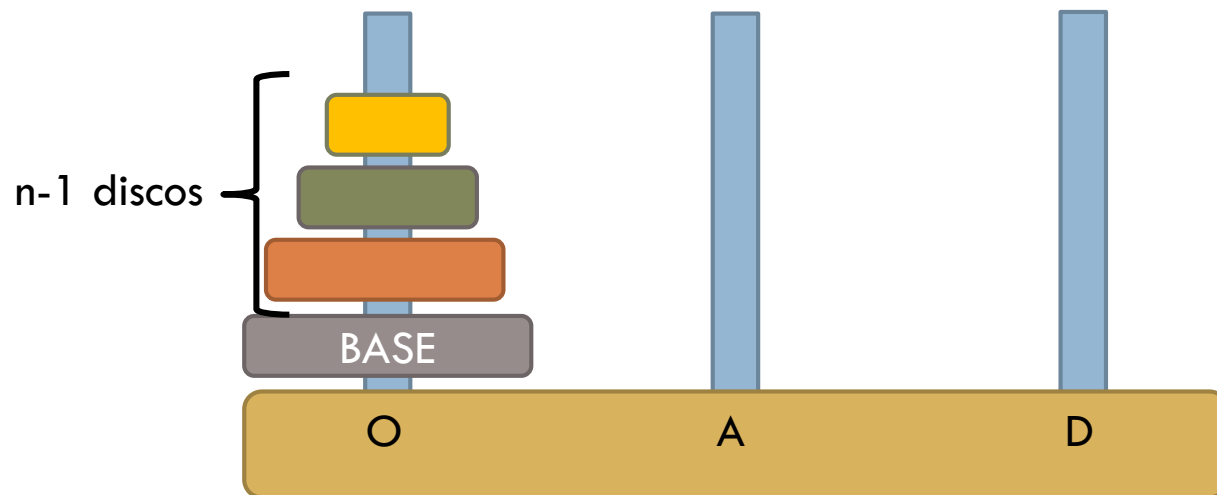
TORRE DE HANOI:

<https://www.ufrgs.br/psicoeduc/hanoi/>

Torre de Hanoi

33

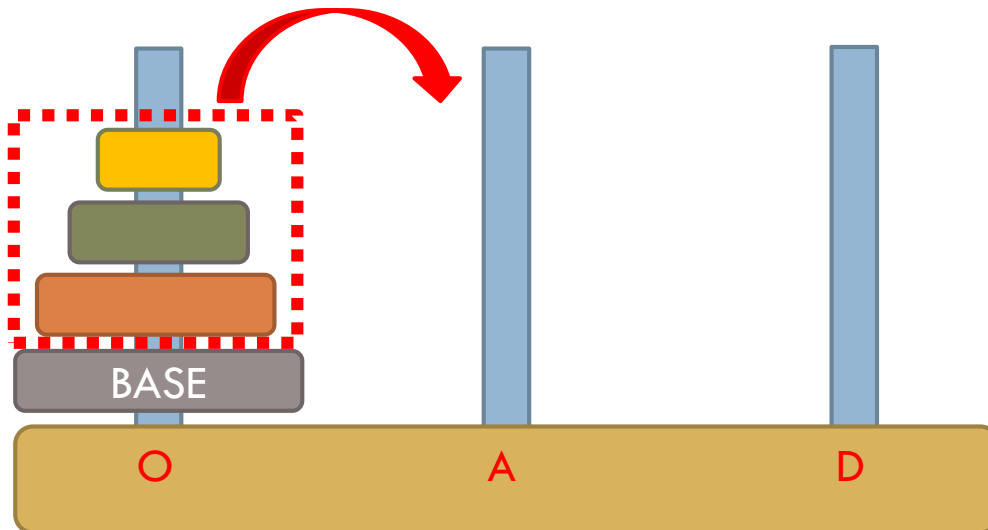
- Simplificando o problema:
 - ▣ Resolver Hanoi para n discos pode ser visto como:
 - Resolver Hanoi para $n-1$ discos da Origem para Auxiliar;
 - Mover base da Origem para Destino;
 - Resolver Hanoi para $n-1$ discos do Auxiliar para Destino.



Torre de Hanoi

34

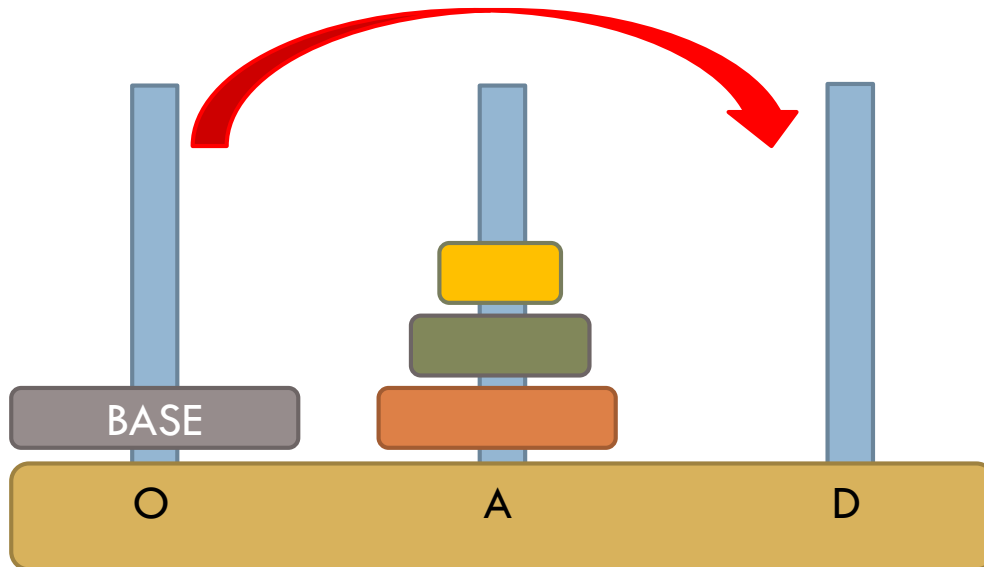
- Simplificando o problema:
 - ▣ Resolver Hanoi para n discos pode ser visto como:
 - Resolver Hanoi para $n-1$ discos da **Origem** para **Auxiliar**;
 - Mover base da Origem para Destino;
 - Resolver Hanoi para $n-1$ discos do Auxiliar para Destino.



Torre de Hanoi

35

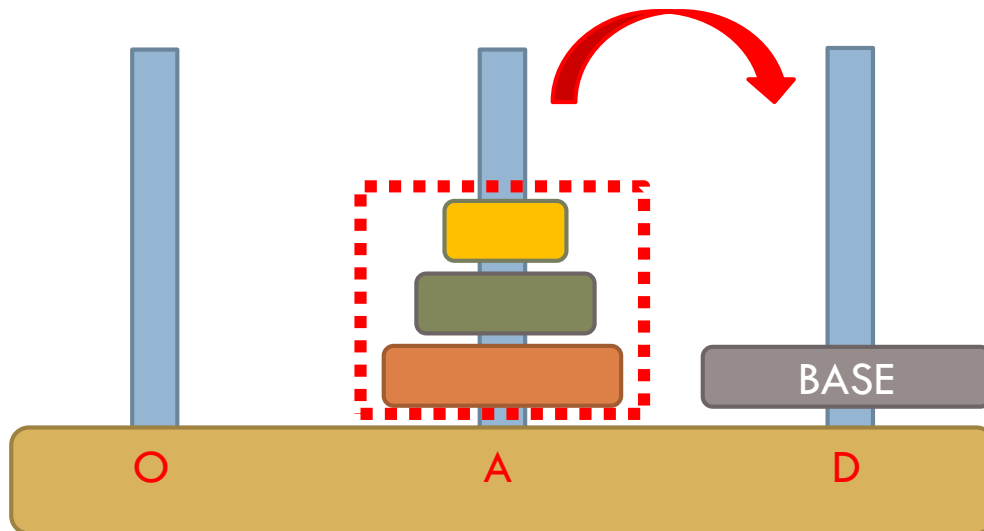
- Simplificando o problema:
 - ▣ Resolver Hanoi para n discos pode ser visto como:
 - Resolver Hanoi para $n-1$ discos da Origem para Auxiliar;
 - **Mover base da Origem para Destino;**
 - Resolver Hanoi para $n-1$ discos do Auxiliar para Destino.



Torre de Hanoi

36

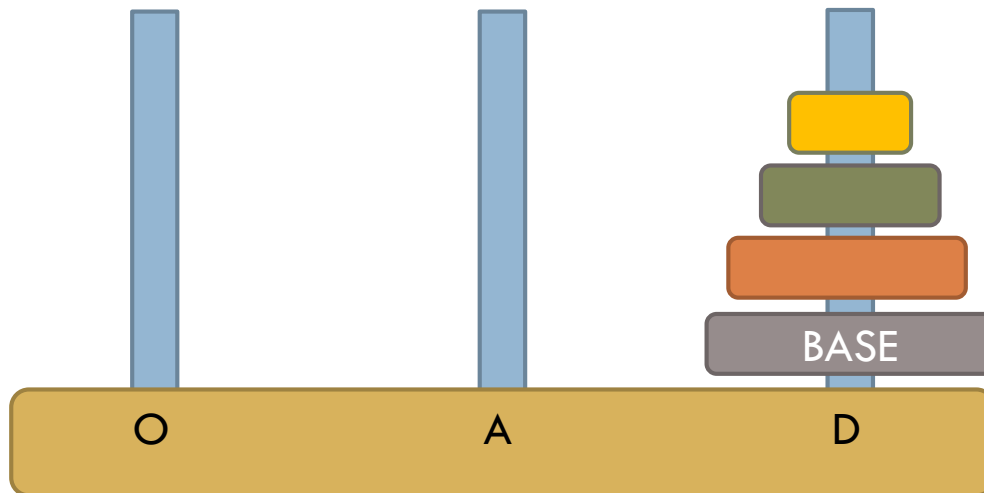
- Simplificando o problema:
 - ▣ Resolver Hanoi para n discos pode ser visto como:
 - Resolver Hanoi para $n-1$ discos da Origem para Auxiliar;
 - Mover base da Origem para Destino;
 - Resolver Hanoi para $n-1$ discos do Auxiliar para Destino.



Torre de Hanoi

37

- Simplificando o problema:
 - ▣ Resolver Hanoi para n discos pode ser visto como:
 - Resolver Hanoi para $n-1$ discos da Origem para Auxiliar;
 - Mover base da Origem para Destino;
 - Resolver Hanoi para $n-1$ discos do Auxiliar para Destino.



Torre de Hanoi

38

- Parâmetros necessários:
- Critério de parada:
- Outros casos:

Torre de Hanoi

39

- Parâmetros necessários:
 - ▣ n , origem, destino, auxiliar;
- Critério de parada:
 - ▣ Mover base da Origem para Destino;
 - ▣ Hanoi com apenas 1 disco ($n == 1$);
- Outros casos:
 - ▣ Resolver Hanoi para $n-1$ discos, da Origem para Auxiliar;
 - ▣ Resolver Hanoi para $n-1$ discos do Auxiliar para Destino.

Torre de Hanoi

40

```
void Hanoi(int n, char origem, char destino, char auxiliar)
{
    if (n == 1)
    {
        printf("\n Mover disco 1 do pino %c para o pino %c", origem, destino);
        return;
    }
    Hanoi(n-1, origem, auxiliar, destino);
    printf("\n Mover disco %d do pino %c para o pino %c", n, origem, destino);
    Hanoi(n-1, auxiliar, destino, origem);
}
```


Torre de Hanoi

41

Definição dos parâmetros

```
void Hanoi(int n, char origem, char destino, char auxiliar)
{
    if (n == 1) ← Critério de parada
    {
        printf("\n Mover disco 1 do pino %c para o pino %c", origem, destino);
        return;
    }
    Hanoi(n-1, origem, auxiliar, destino);
    printf("\n Mover disco %d do pino %c para o pino %c", n, origem, destino);
    Hanoi(n-1, auxiliar, destino, origem);
}
```

Outros casos

Série de Fibonacci

42

- 1 1 2 3 5 8 13 ...
- Identifique o padrão recursivo.
- **Informe o que é necessário para desenvolver uma função recursiva** que retorne o valor em uma determinada posição da sequência.
- Desenhe a árvore de recursão para esse caso.

Série de Fibonacci

43

□ 1 1 2 3 5 8 13 ...

```
int fibonacci(int n)
{
    if (n == 0) return 0;
    else if (n == 1) return 1;
    return fibonacci(n-2) + fibonacci(n-1);
}
```

□ Desenhe o árvore de recursão para esse caso.