Cap 4. RECURSIVIDADE

- **Recursividade** (ou **recursão**) ⇒ capacidade que uma linguagem tem de permitir que uma função possa invocar a si mesma.

- **Função recursiva** é aquela que chama a si própria.

- Recursividade pode ser **direta** ou **indireta**:

  → **Recursividade Direta** ⇒ quando uma função invoca a si mesma no seu corpo de função.

  → **Recursividade Indireta** ⇒ quando uma função ***f*** invoca uma outra função ***g*** que, por sua vez, volta a invocar a função ***f***.

- Funções recursivas ⇒ são, em sua maioria, **soluções mais elegantes e simples**.

  ⇒ Essa elegância e simplicidade têm um preço → requer muita atenção em sua implementação.

- **Exemplo cotidiano**: ação de contar um saco de moedas.

→ A cada ato de retirar uma moeda do saco, precisa-se "contar dinheiro" ≡ identificar qual é o valor da moeda e somá-la à quantia que ainda está no saco.

→ Para identificar a quantia que ainda está no saco ⇒ basta chamar a função "contar dinheiro" novamente, porém, dessa vez, já considerando que essa moeda não está mais lá.

→ Processo de:

1. retirar uma moeda,
2. identificar seu valor,
3. somar com o restante do saco

repete-se até que o saco esteja vazio, quando atingiremos o **ponto de parada** e a função retornará o valor zero, indicando que não há mais moedas no saco.

→ Função "contar dinheiro" ⇒ foi chamada um número de vezes igual a quantidade de moedas no saco.

→ **Última chamada** ⇒ começa a devolver os valores de retorno de cada instância da função, iniciando por zero (saco vazio), somado ao valor da última moeda, da penúltima, etc, até retornar à primeira chamada referente a primeira moeda.

→ Aqui se encerra a função, trazendo como valor de retorno a soma dos valores de todas as moedas que estavam no saco.

- **Cada vez que uma função é chamada de forma recursiva** ⇒ são alojados e armazenados uma cópia dos seus parâmetros, de modo a não perder os valores dos parâmetros das chamadas anteriores.

- Em **cada instância da função** ⇒ só são diretamente acessíveis os parâmetros criados para esta instância, não sendo possível acessar os parâmetros das outras instâncias.

- A informação armazenada na chamada de uma função ⇒ é designada <u>estrutura de invocação</u> ou <u>registro de ativação</u> e contém:

  → Endereço de retorno
  → Estado dos registros e flags da CPU
  → Variáveis passadas como argumentos para a função
  → Variável de retorno

- Chamada a uma função recursiva **=** chamada de uma função não recursiva ⇒ é necessário guardar uma "estrutura de invocação", sendo esta estrutura liberada depois do fim da execução da função e atualização do valor de retorno.

- Funções recursivas contem <u>duas partes fundamentais</u>:

  → **Ponto de Parada ou Condição de Parada**: que é o ponto onde a função será encerrada. Geralmente, é um limite superior ou inferior da regra geral.

  → **Regra Geral**: é o método que reduz a resolução do problema através da invocação recursiva de casos menores, que por sua vez são resolvidos pela resolução de casos ainda menores pela própria função, assim sucessivamente até atingir o "ponto de parada" que finaliza o método.

- **Para criar um algoritmo recursivo:**

1º) Procurar encontrar uma solução de como o problema pode ser dividido em passos menores.

2º) Definir um ponto de parada.

3º) Definir uma regra geral que seja válida para todos os demais casos.

Deve-se, ainda, verificar se o algoritmo termina $\equiv$ se o ponto de parada é atingido.

Para auxiliar nessa verificação $\rightarrow$ recomenda-se criar uma árvore de execução do programa, mostrando o desenvolvimento do processo.

Vantagens da recursão:

$\rightarrow$ Redução do tamanho do código fonte

$\rightarrow$ Maior clareza do algoritmo para problemas de definição naturalmente recursiva.

Desvantagens da recursão:

$\rightarrow$ Baixo desempenho na execução devido ao tempo para gerenciamento das chamadas $\Rightarrow$ processo recursivo requer recursos da máquina, tanto de tempo quando de espaço de memória.

$\rightarrow$ Dificuldade de depuração dos subprogramas recursivos, principalmente se a recursão for muito profunda.

Quando utilizar a recursividade?

$\Rightarrow$ Quando esta for o modo mais simples e intuitivo de implementar uma solução para a resolução de um determinado problema.

Se não for simples e intuitiva $\rightarrow$ melhor empregar métodos não recursivos $\rightarrow$ “métodos iterativos”.

$\Rightarrow$ A **recursividade** é uma forma de implementar um laço através de chamadas sucessivas à mesma função.

Aplicações práticas de funções recursivas na linguagem C

EX1) Como **primeiro exemplo** de função recursiva, vamos ver o **cálculo de fatorial**.

Primeiro, imaginemos um algoritmo simples para calcular o fatorial de um número.

```
int fatorial (int x)
{
    int i, p;
    p = 1;
    for ( i=2; i<=n; i++ )
            p = p * i;
    return p;
}
```

Vamos pensar como se calcula o fatorial de um número:

- O fatorial de zero é, por definição, 1.
- O fatorial de 1 é 1.
- O fatorial de 5 é 5x4x3x2x1.

Agora: **qual é o fatorial de N?**

N! = N * (N-1) * (N-2) * .... * 2 * 1

é equivalente a:

N! = N * (N-1) !

pois **(N-1) !** = (N-1) * (N-2) * .... * 2 * 1

Olhando a definição ⇒ podemos dizer que **o fatorial de N é realizado à custa do próprio fatorial**:

0! = 1
N! = N * (N-1)!

⇒ **Viram a recursividade ?**

∴ **Versão recursiva** da função fatorial:

```
int fatorial (int x)
{
    if ( x == 0 )    //critério para término
        return 1;
    return  x * fatorial(x-1);      //chamada recursiva
}
```

Viram como a função ficou bem mais simples?

⇒ **Caso da função <u>não recursiva:</u>** duas variáveis, *i* e *p*, são necessárias para armazenar o estado da computação.

<u>*Ex*</u>: ao calcular o fatorial de 6, o computador vai passar sucessivamente por

i	p
===	===
	1
2	2
3	6
4	24
5	120
6	720

⇒ **Caso recursivo**:

Para calcular o fatorial de 6 → computador tem que calcular primeiro o fatorial de 5 e só depois é que faz a multiplicação de 6 pelo resultado (120).

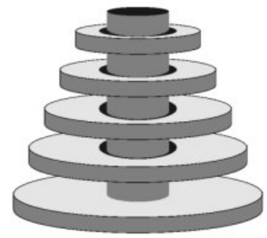
Para calcular o fatorial de 5 → tem que calcular o fatorial de 4.

Esse processo continua até chegarmos ao caso base.

<u>Resumindo:</u> internamente, ocorre uma expansão seguida de uma contração.

```
factorial(6)
6 * factorial(5)
6 * 5 * factorial(4)
6 * 5 * 4 * factorial(3)
6 * 5 * 4 * 3 * factorial(2)
6 * 5 * 4 * 3 * 2 * factorial(1)
6 * 5 * 4 * 3 * 2 * 1 * factorial(0)
6 * 5 * 4 * 3 * 2 * 1 * 1
6 * 5 * 4 * 3 * 2 * 1
6 * 5 * 4 * 3 * 2
6 * 5 * 4 * 6
6 * 5 * 24
6 * 120
720
```

EX2) Problema clássico: **<u>as Torres de Hanoi</u>**



→ O problema consiste em:

- uma pilha de *n* discos, numeradas de 1 a *n;*
- 3 postes: origem (**O**) , destino (**D**) e auxiliar (**A**);

→ O jogo funciona assim:

- um disco de número menor não pode estar debaixo de um disco maior;
- podemos retirar apenas o disco superior de uma determinada pilha.

→ Entradas e saídas:

- o número de **entradas** é simplesmente o número *n* de discos;
- as **saídas** poderão ser as mais variadas. Por exemplo, uma tela de animação. Vamos adotar a <u>saída em modo texto</u>, com instruções no seguinte formato:

"Mova o disco **x** do poste **L** para o poste **M**."

→ **<u>Solução Recursiva</u>**:

- Parte da identificação de que a solução problema, quando temos **apenas 1 disco**, é trivial ⇒ consiste na movimentação deste disco do poste origem para o poste destino;

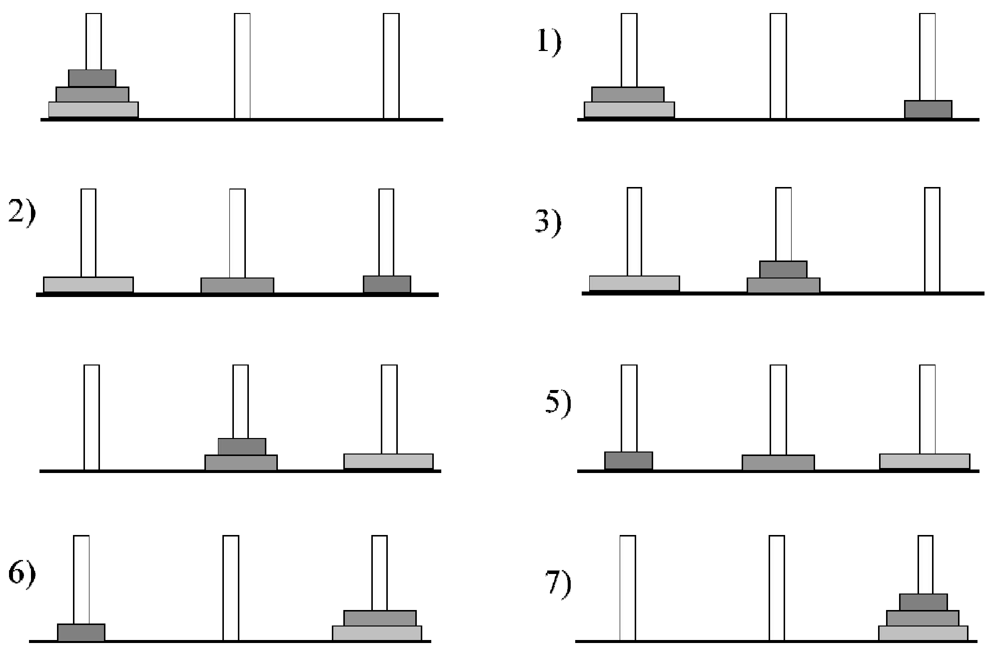
- Deve ser então observado que **a solução para uma quantidade *n*** qualquer de discos ⇒ obtida considerando que temos que mover *n-1* discos do poste <u>origem</u> para o poste <u>auxiliar</u>, chegando então ao problema trivial de um disco, e depois movimentando os mesmos n-1 discos do poste <u>auxiliar</u> para o <u>destino</u>.

<u>Torre</u>

- Se 1 disco ⇒ mover do poste **atual** para seu **destino**;
- Se *n* discos ⇒
  - resolver torre com *n-1* discos do poste **atual** para **auxiliar**
  - resolver torre com 1 disco
  - resolver torre com *n-1* discos do **auxiliar** para **destino**

→ **<u>Algoritmo</u>**:

```
void   hanoi  (int  n,  int o,  int  a,  int d)
{
    if ( n ==1 )    //solução trivial
            printf("\t  Mova o disco %d  de  %d   para  %d  \n", n, o, d);
    else
    {        hanoi(n-1,o,d,a);
            printf("\t  Mova o disco %d  de  %d   para  %d  \n", n, o, d);
            hanoi(n-1,a,o,d);
     }
 }
int main (void)
{
    int total_discos;
    printf("\t Introduza o número de discos: ");
    scanf("%d",&total_discos);
    hanoi(total_discos,1,2,3);
 }
```




```
Introduza o n·mero de discos: 3

   Mova o disco 1  de  1   para  3
   Mova o disco 2  de  1   para  2
   Mova o disco 1  de  3   para  2
   Mova o disco 3  de  1   para  3
   Mova o disco 1  de  2   para  1
   Mova o disco 2  de  2   para  3
   Mova o disco 1  de  1   para  3
```