

Algoritmos e Programação II

Recursividade

Profª Yorah Bosse

yorah.bosse@gmail.com

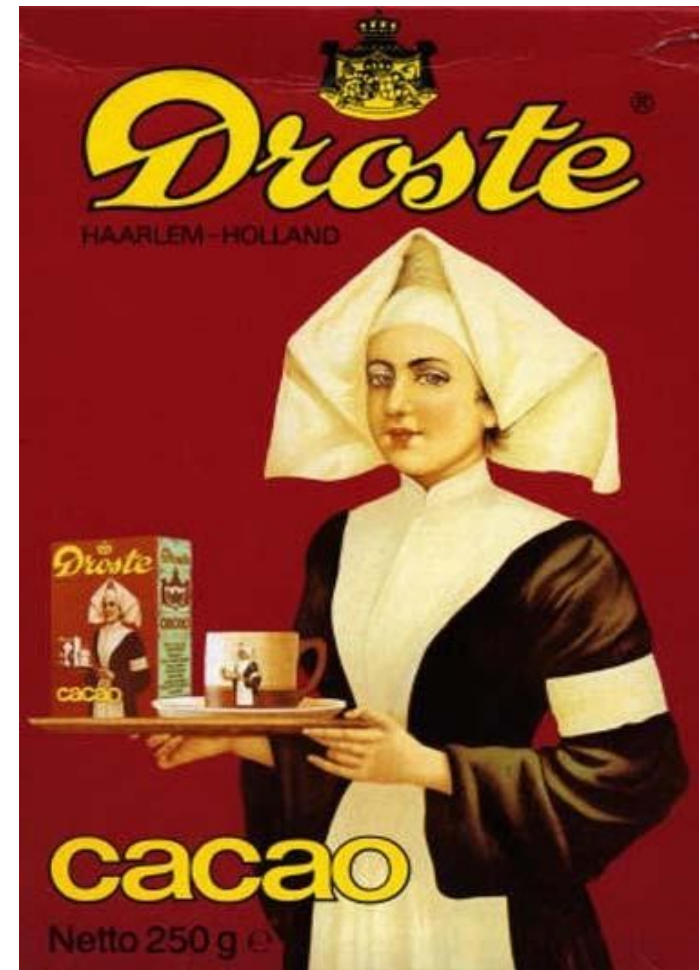
The logo of the Universidade Federal do Mato Grosso do Sul (UFMS) is located in the bottom left corner. It features a stylized graphic of vertical black lines of varying heights on the left, and a circular emblem with horizontal black and white stripes on the right. Below these elements, the letters 'UFMS' are written in a bold, black, sans-serif font, set against a light blue rectangular background.

UFMS

- É o processo de resolução de um problema, reduzindo-o em um ou mais subproblemas com as seguintes características:
 - São idênticos aos problemas originais;
 - São mais simples de resolver.
- Uma vez realizada a primeira subdivisão, a mesma técnica de decomposição é usada para dividir cada subproblema.
- Eventualmente, os subproblemas tornam-se tão simples que é possível resolvê-los sem efetuar novas subdivisões.

- Recursão é usado em arte (em figuras, telas, etc.), em matemática, em programação e em muitas situações do nosso cotidiano
- Exemplo de uma definição recursiva sobre ancestralidade:
 - (caso base) Os pais de uma pessoa são seus antepassados
 - (passo recursivo) Os pais de qualquer antepassado de uma pessoa são também antepassados desta pessoa

- Em figuras, é usado quando a figura contém ela mesma. Isto gera um efeito chamado de efeito “Droste”
- O nome veio de um produto holandês (cacau em pó), cuja embalagem possui figura recursiva

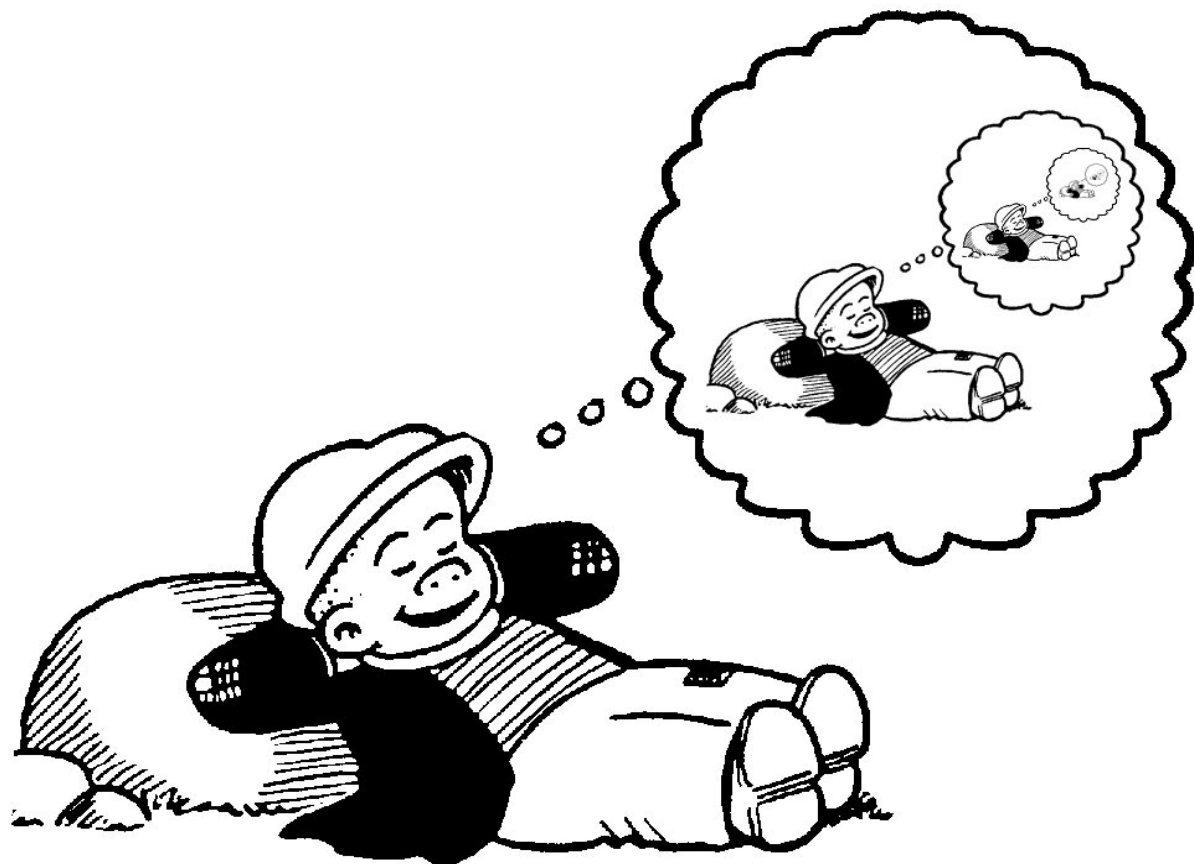


- Este tipo de efeito é frequentemente usado em fotos e álbuns como o Ummagumma (Pink Floyd)



Observe essa imagem

- Outros exemplos



- Na programação: surgiu meados dos anos 60
- Linguagem: Algol 60
- Definição:

“Algoritmo recursivo é aquele que usa a si mesmo, só que com parâmetros diferentes...”

Prof.Sérgio V.A.Campos - UFMG

- Exemplo clássico:

FATORIAL DE “n”

Sendo “n” um número inteiro positivo

Cálculo do fatorial :

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

- Exemplo clássico:

FATORIAL DE “5”

$$5! = 5 * 4!$$

$$4! = 4 * 3!$$

$$3! = 3 * 2!$$

$$2! = 2 * 1!$$

$$1! = 1 * 0!$$

$$0! = 1$$

$\text{fat}(n) = n * \text{fat}(n-1) \rightarrow \text{até que } n = 1$

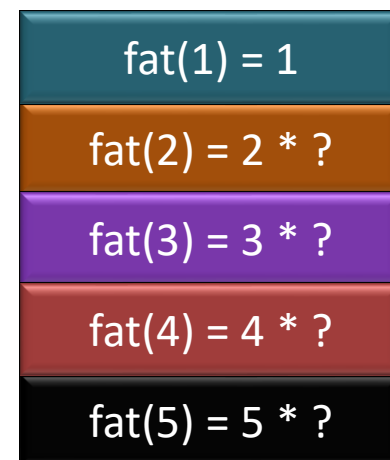
$$\text{fat}(5) = 5 * \text{fat}(4)$$

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

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

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

$$\text{fat}(1) = 1$$



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

Empilhando

$\text{fat}(n) = n * \text{fat}(n-1) \rightarrow \text{até que } n = 1$

$$\text{fat}(5) = 5 * \cancel{\text{fat}(4)}^{24} = 120$$

$$\text{fat}(4) = 4 * \cancel{\text{fat}(3)}^6 = 24$$

$$\text{fat}(3) = 3 * \cancel{\text{fat}(2)}^2 = 6$$

$$\text{fat}(2) = 2 * \cancel{\text{fat}(1)}^1 = 2$$

$$\cancel{\text{fat}(1) = 1}$$

$\text{fat}(1) = 1$
$\text{fat}(2) = 2 * 1$
$\text{fat}(3) = 3 * 2$
$\text{fat}(4) = 4 * 6$
$\text{fat}(5) = 5 * 24$

Desempilhando

```
#include <stdio.h>
```

```
int fatorial (int n) {  
    if (n <= 1)  
        return 1; // caso base  
    else // passo recursivo  
        return n * fatorial (n-1);  
}
```

Representação matemática:

$$n! = \begin{cases} 1 & \text{se } n \leq 1 \\ n \cdot (n-1)! & \text{se } n > 1 \end{cases}$$

```
int main(){  
    int num;  
    do{  
        printf("Digite um numero inteiro positivo ou zero: ");  
        scanf("%d",&num);  
    }while(num<0);  
    printf("\nO fatorial de %d eh %d\n\n",num,fatorial(num));  
    return 0;  
}
```

Recursivo

```
int fatorial (int n) {  
    if (n <= 1)  
        return 1; // caso base  
    else // passo recursivo  
        return n * fatorial (n-1);  
}
```

Iterativo

```
int fatorial (int n) {  
    int fat = 1; i;  
    for( i = n; i > 1; i-- )  
        fat *= i;  
    return fat;  
}
```

**Mas, se dá para resolver sem recursão,
por que devo aprender essa técnica?**

- A solução iterativa é, em geral, mais rápida do que a recursiva.
- A solução recursiva pode também ocupar muito mais espaço na memória, pois precisa armazenar em uma pilha cada resultado antes de resolvê-lo.
- Em alguns problemas a solução recursiva é muito mais prática, e em algumas linguagens de programação, só é possível fazer repetição por meio de recursividade. Por exemplo, a linguagem LISP, do paradigma funcional.

- Uma função recursiva deve obrigatoriamente ter um **critério de parada**
- A parada da recursividade se dá pelo caso base (que não possui recursão)

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

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

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

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

$$\text{fat}(1) = 1$$

- Exemplo 2: (cálculo do somatório)

Qual o somatório de $[2, 5]$?

$$\text{somatorio}(2,5) = 2 + 3 + 4 + 5$$


```
#include <stdio.h>
#include <stdlib.h>

int somatorio (int m, int n) {
    // caso base
    if (n == m) // caso base
        return (m);
    else // passo recursivo
        return (m + somatorio (m+1, n));
}
```

```
int main(){
    int num1,num2,aux;
    do{
        printf("Digite o 1o numero: ");
        scanf("%d",&num1);
    }while(num1<0);
    do{
        printf("Digite o 2o numero: ");
        scanf("%d",&num2);
    }while(num2<0);
    if (num2 < num1){
        aux = num1;
        num1 = num2;
        num2 = aux;
    }
    printf("\nA soma eh %d\n\n",
           somatorio(num1,num2));
    system("pause");
    return 0;
}
```

```
int somatorio (int m, int n) {  
    // caso base  
    if (n == m)  
        return (m);  
    else // passo recursivo  
        return (m + somatorio (m+1, n));  
}
```

Representação matemática:

$$\sum_{k=m}^n = \begin{cases} m & \text{se } n = m \text{ e} \\ m + \sum_{k=m+1}^n & \text{se } n > m. \end{cases}$$

Dado $S(m,n)$, onde $n > m$, calcule $S(2, 5)$:

The diagram illustrates the recursive calculation of $S(2, 5)$. It shows a series of equations where the recursive call is crossed out and its value is substituted. Arrows indicate the flow of the calculation from the base case back to the initial call.

$$\begin{aligned} S(2, 5) &= 2 + \cancel{S(3, 5)}^{12} \\ S(3, 5) &= 3 + \cancel{S(4, 5)}^9 \\ S(4, 5) &= 4 + \cancel{S(5, 5)}^5 \\ S(5, 5) &= 5 \end{aligned}$$

The final result, 14, is shown in a black box next to the first equation.

Recursivo

```
int somatorio (int m, int n) {  
  
    if (n == m) // caso base  
        return (m);  
    else // passo recursivo  
        return (m + somatorio (m+1, n));  
}
```

Iterativo

```
int somalt(int m, int n) {  
    int soma = 0,i;  
    for (i=m;i<=n;i++)  
        soma += i;  
    return soma;  
}
```

- Exemplo 3 (cálculo da potência)

Dado que:

$$\begin{aligned}
 3^0 &= 1 \\
 3^1 &= 3 \\
 3^2 &= 3 * 3 \\
 3^3 &= 3 * 3 * 3 \\
 3^4 &= 3 * 3 * 3 * 3 \\
 3^5 &= 3 * 3 * 3 * 3 * 3
 \end{aligned}$$

$$\begin{aligned}
 3^0 &= 1 \\
 3^1 &= 3 \\
 3^2 &= 3 * 3^1 \\
 3^3 &= 3 * 3^2 \\
 3^4 &= 3 * 3^3 \\
 3^5 &= 3 * 3^4
 \end{aligned}$$

```
#include <stdio.h>
#include <stdlib.h>

int pot (int m, int n) {
    if (n == 0)
        return 1;
    else
        // caso base
        if (n == 1)
            return m;
        else // passo recursivo
            return (m * pot(m, n-1));
}
```

```
int main(){
    int num1,num2;
    do{
        printf("Base: ");
        scanf("%d",&num1);
    }while(num1<0);
    do{
        printf("Expoente: ");
        scanf("%d",&num2);
    }while(num2<0);
    printf("\n%d elevado a %d = %d\n\n",
        num2,num1,pot(num1,num2));
    system("pause");
    return 0;
}
```

```
int pot (int m, int n) {
    if (n == 0)
        return 1;
    else
        if (n == 1) // caso base
            return m;
        else // passo recursivo
            return (m * pot(m, n-1));
}
```

Representação matemática:

$$x^n = \begin{cases} 1/x^{-n} & \text{se } n < 0, \\ 1 & \text{se } n = 0 \text{ e} \\ x \times x^{n-1} & \text{se } n > 0. \end{cases}$$

Dado $P(m,n)$, onde $n \geq 0$, calcule $P(2, 4)$:

$$\begin{array}{lcl}
 P(2,4) = & 2 * P(2,3) & \leftarrow \\
 \mathbf{16} & & \leftarrow \\
 & P(2,3) = 2 * P(2,2) & \leftarrow \\
 & & \leftarrow \\
 & P(2,2) = 2 * P(2,1) & \leftarrow \\
 & & \leftarrow \\
 & P(2,1) = 2 &
 \end{array}$$

Recursivo

```
int pot (int m, int n) {  
    if (n == 0)  
        return 1;  
    else  
        if (n == 1) // caso base  
            return m;  
        else // passo recursivo  
            return (m * pot(m, n-1));  
}
```

Iterativo

```
int potIt(int m, int n) {  
    int pl = 1, i;  
    for (i=n; i>=1; i--)  
        pl *= m;  
    return pl;  
}
```

- Encontrar o maior elemento em um vetor por um algoritmo recursivo


```
int maiorRec1(int v[], int length)
{
    if ( length == 1 ) // caso base
        return v[0];
    else // passo recursivo
    {
        int temp;
        temp = maiorRec1( v, length - 1 );
        if (temp > v[ length - 1 ] )
            return temp;
        else
            return v[ length - 1 ];
    } // fim do método maximoRecursivo
}
```

OU

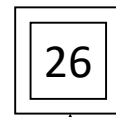
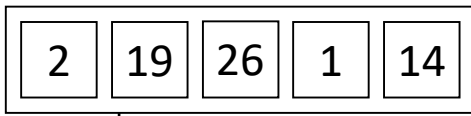
```
int maiorRec2(int v[], int pos, int maior)
{
    if (pos == 0)
        return v[pos]>maior?v[pos]:maior;
    else
        return maiorRec2(v,pos-1,
                        v[pos]>maior?v[pos]:maior);
}
```

Primeira chamada para um vetor vet de max posições:

`int x = maiorRec1(vet,max)`

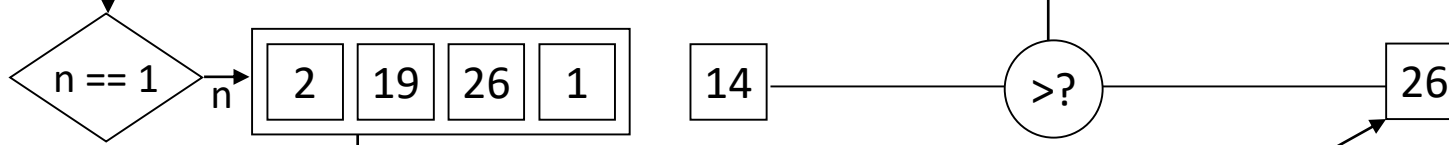
`int x = maiorRec2(vet,max-2,vet[max-1])`

Teste de mesa para a solução 1

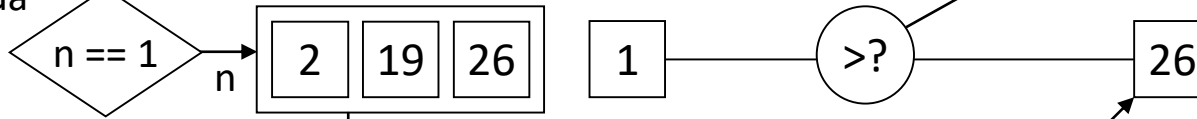


Elemento Máximo

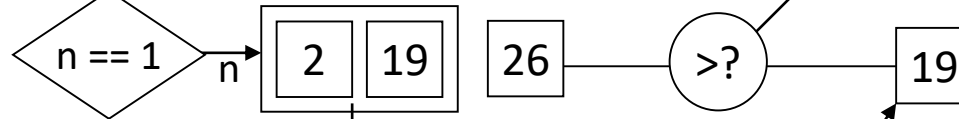
1a. chamada



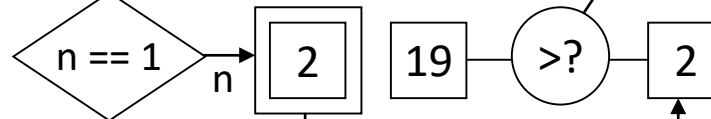
2a. chamada



3a. chamada



4a. chamada



5a. chamada

