



UNIVERSIDADE FEDERAL DE ITAJUBÁ

Algoritmos e Estrutura de Dados I

COM 111

Recursão

Vanessa Cristina Oliveira de Souza



Recursão





Recursão

- ▶ Método de **resolução de problemas** que envolve **quebrar um problema em subproblemas menores** até chegar a um problema pequeno o suficiente para que ele possa ser resolvido **trivialmente**.
- ▶ Um método que chama a si mesmo direta, ou indiretamente, é dito **recursivo**.





Recursão



Matrioska





Recursão

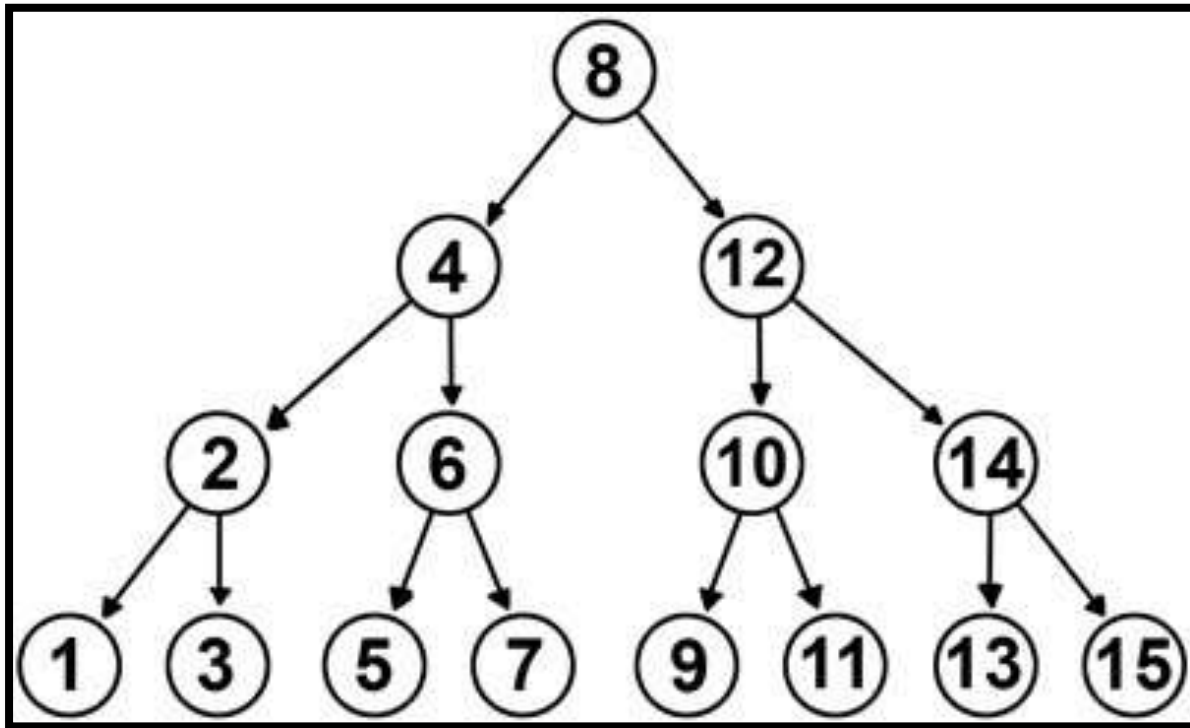


Fractais Geométricos





Recursão



Árvore Binária de Busca



Leis da Recursão

- ▶ Todo algoritmo recursivo deve obedecer a 3 leis:
 1. O algoritmo recursivo deve possuir um **caso base**;
 2. O algoritmo recursivo deve alterar o seu estado de maneira a **se aproximar do caso base**;
 3. O algoritmo recursivo deve ter uma **chamada a si mesmo** (direta ou indiretamente).





Como o compilador implementa a recursão?

- ▶ Um compilador implementa um método recursivo por meio de uma **pilha**, na qual são armazenados os dados usados em cada chamada de um método que ainda não terminou de processar.
- ▶ Todos os dados não globais vão para a pilha, pois o **estado corrente da computação deve ser registrado** para que possa ser recuperado posteriormente.





Como funciona a Recursão?

- ▶ Pode-se dizer que os algoritmos recursivos apresentam duas fases:
 - ▶ Fase de expansão
 - ▶ Fase de encolhimento





Como funciona a Recursão?

- ▶ Pode-se dizer que os algoritmos recursivos apresentam duas fases:
 - ▶ **Fase de expansão**
 - ▶ As chamadas da função são **empilhadas**, mas ainda não apresentam resultado algum
 - ▶ As chamadas são empilhadas até que se chegue ao **caso base**
 - ▶ O 'empilhamento de chamadas' acontece quando o método chama a ele mesmo





Como funciona a Recursão?

- ▶ Pode-se dizer que os algoritmos recursivos apresentam duas fases:
 - ▶ **Fase de encolhimento**
 - ▶ Chegando no caso base, é possível começar a realizar a computação
 - ▶ As chamadas começam a ser desempilhadas
 - ▶ A computação termina quando a pilha estiver vazia





Como funciona a Recursão?

- ▶ Pode-se dizer que os algoritmos recursivos apresentam duas fases:
 - ▶ **Fase de encolhimento**
 - ▶ Chegando no caso base, é possível começar a realizar a computação
 - ▶ As chamadas começam a ser desempilhadas
 - ▶ A computação termina quando a pilha estiver vazia

LEMBRETE:
UMA FUNÇÃO SEMPRE RETORNA O VALOR CALCULADO PARA
AQUELA FUNÇÃO QUE A CHAMOU!







Exemplo 1

- Calcular a soma de elementos em um vetor de inteiros

```
int somaIt(int vet[], int tam)
{
    int i;
    int soma = 0;

    for (i=0; i<tam; i++)
    {
        soma = soma + vet[i];
    }

    return soma;
}
```

```
int main()
{
    int vet[10] = {1, 2, 3, 4, 5, 5, 6, 7, 10, 0};
    int res = somaIt(vet, 10);

    printf("O valor da soma iterativa eh : %d\n", res);

    return 0;
}
```





Exemplo 1

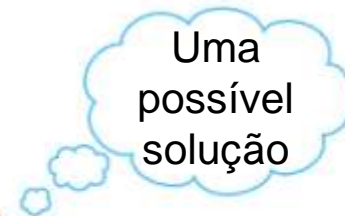
- ▶ Calcular a soma de elementos em um vetor de inteiros
 - ▶ É possível pensar numa solução recursiva para esse problema?
 - ▶ Qual é meu caso base?
 - ▶ Como eu posso me aproximar dele?






Exemplo 1

- ▶ Calcular a soma de elementos em um vetor de inteiros
- ▶ É possível pensar numa solução recursiva para esse problema?
 - ▶ $v[0] + \text{soma}(v[1:9])$
 - $v[1] + \text{soma}(v[2:9])$
 - $v[2] + \text{soma}(v[3:9])$
 - ▶ ...
 - $v[9]$





Exemplo 1

- ▶ Calcular a soma de elementos em um vetor de inteiros
 - ▶ É possível pensar numa solução recursiva para esse problema?
 - ▶ $v[0] + \text{soma}(v[1:9])$
 - $v[1] + \text{soma}(v[2:9])$
 - $v[2] + \text{soma}(v[3:9])$
 - ▶ ...
 - $v[9]$  **Caso base**

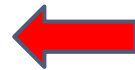
O algoritmo recursivo deve possuir um **caso base**





Exemplo 1

- ▶ Calcular a soma de elementos em um vetor de inteiros
- ▶ É possível pensar numa solução recursiva para esse problema?
 - ▶ $v[0] + \text{soma}(v[1:9])$
 - $v[1] + \text{soma}(v[2:9])$
 - $v[2] + \text{soma}(v[3:9])$
 - ▶ ...
 - $v[9]$



Atualização das Posições

O algoritmo recursivo deve alterar o seu estado de maneira a **se aproximar do caso base**





Exemplo 1

- ▶ Calcular a soma de elementos em um vetor de inteiros
 - ▶ É possível pensar numa solução recursiva para esse problema?
 - ▶ $v[0] + \text{soma}(v[1:9])$
 - $v[1] + \text{soma}(v[2:9])$
 - $v[2] + \text{soma}(v[3:9])$
 - ▶ ...
 - $v[9]$

 **soma**

O algoritmo recursivo deve ter uma **chamada a si mesmo** (direta ou indiretamente).





Exemplo 1

- Calcular a soma de elementos em um vetor de inteiros

```
int somaRec(int vet[], int ini, int fim)
{
    int soma;
    if (ini == fim)
        soma = vet[fim];
    else
        soma = vet[ini] + somaRec(vet, ini+1, fim);
    return soma;
}
```





Exemplo 1

- Calcular a soma de elementos em um vetor de inteiros

```
int somaRec(int vet[], int ini, int fim)
{
    int soma;
    if (ini == fim)
        soma = vet[fim];
    else
        soma = vet[ini] + somaRec(vet, ini+1, fim);
    return soma;
}
```

```
int main()
{
    int vet[10] = {1, 2, 3, 4, 5, 5, 6, 7, 10, 0};

    int res2 = somaRec(vet, 0, 9);

    printf("O valor da soma recursiva eh : %d\n", res2);
    return 0;
}
```

```
int somaIt(int vet[], int tam)
{
    int i;
    int soma = 0;

    for (i=0; i<tam; i++)
    {
        soma = soma + vet[i];
    }

    return soma;
}
```



Exemplo 1

- ▶ Calcular a soma de elementos em um vetor de inteiros

```
int somaRec(int vet[], int ini, int fim)
{
    int soma;
    if (ini == fim)
        soma = vet[fim];
    else
        soma = vet[ini] + somaRec(vet, ini+1, fim);
    return soma;
}
```

- ▶ Muitas linguagens de programação permitem o acesso a sub-vetores. Em C isso não é possível, por isso é necessário o acesso pelas posições inicial e final.





Exemplo 1

- Calcular a soma de elementos em um vetor de inteiros

```
int somaRec(int vet[], int ini, int fim)
{
    int soma;
    if (ini == fim)
        soma = vet[fim];
    else
        soma = vet[ini] + somaRec(vet, ini+1, fim);
    return soma;
}
```

```
int main()
{
    int vet[10] = {1, 2, 3, 4, 5, 5, 6, 7, 10, 0};

    int res2 = somaRec(vet, 0, 9);

    printf("O valor da soma recursiva eh : %d\n", res2);
    return 0;
}
```

**TESTE
DE
MESA**



Exemplo 2

- ▶ Calcular o fatorial de um número
 - ▶ $n!$ = Produto de todos os inteiros positivos menores ou iguais a n

$$n! = \prod_{k=1}^n k$$

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





Exemplo 2

- ▶ Calcular o fatorial de um número
 - ▶ $n!$ = Produto de todos os inteiros positivos menores ou iguais a n

$$0! = 1$$





Exemplo 2

- ▶ Calcular o fatorial de um número
 - ▶ Definição Recursiva

$$n! = \begin{cases} 1, & \text{se } n = 0 \\ n * (n-1)!, & \text{se } n > 0 \end{cases}$$





Exemplo 2

- ▶ Calcular o fatorial de um número
 - ▶ Solução Iterativa

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





Exemplo 2

- ▶ Calcular o fatorial de um número
 - ▶ Solução Recursiva

$$n! = \begin{cases} 1, & \text{se } n = 0 \\ n * (n-1)!, & \text{se } n > 0 \end{cases}$$





Exemplo 2

- ▶ Calcular o fatorial de um número
- ▶ Solução Recursiva

```
int fatRec(int valor)
{
    int fat;
    if (valor == 0)
    {
        fat = 1;
    }
    else
    {
        fat = valor * fatRec(valor-1);
    }
    return fat;
}
```

res = fatRec(5);

**TESTE
DE
MESA**





Exemplo 2

- ▶ Calcular o fatorial de um número
 - ▶ Solução Recursiva

```
int fatRec(int valor)
{
    if (valor == 0)
        return 1;
    else
        return valor * fatRec(valor-1);
}
```

Não é necessário declarar a variável local





Exercício

1. Dado um vetor de números inteiros, implemente:
 - a) Função iterativa que encontre o maior elemento do vetor
 - b) Função recursiva que encontre o maior elemento do vetor

2. Depure o código e acompanhe a pilha de execução:
 - a) Na função iterativa
 - b) Na função recursiva

