

# Linguagem C: Funções recursivas

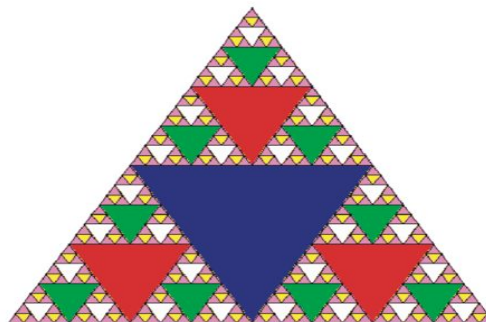
Joice Otsuka

# Função recursiva

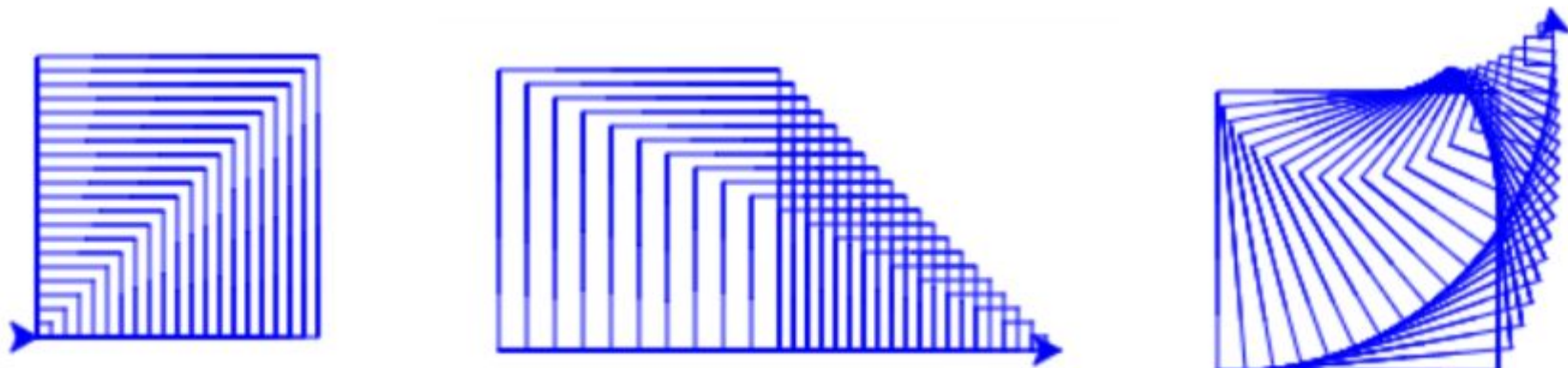
- Uma função recursiva é uma função que chama a si mesma, reduzindo o problema a cada chamada
- É um método de resolução de problemas
  - O problema é sucessivamente “quebrado” em subproblemas menores, até chegar a um problema muito pequeno, de solução trivial.

# Função recursiva

- A recursão também é chamada de definição circular.
  - Um objeto é dito recursivo se pode ser definido em termos de si próprio



# Visualizando a recursão



# Para que?

- Alguns problemas complexos possuem soluções recursivas mais simples e elegantes
  - Fatorial, Sequência de Fibonacci, Fractais, Algoritmos de grafos, Algoritmos com listas e strings

# Exemplo: Fatorial

- Um exemplo clássico de função que usa recursão é o cálculo do fatorial de um número:
  - $3! = 3 * 2!$
  - $4! = 4 * 3!$
  - $n! = n * (n - 1)!$

# 3 leis da recursão

- Um algoritmo recursivo deve ter um caso base
  - caso com solução de trivial, quando a função pára de chamar a si mesma e retorna solucionando.
- Um algoritmo recursivo deve mudar o seu estado e se aproximar do caso base.
- Um algoritmo recursivo deve chamar a si mesmo, recursivamente.

# Recursão

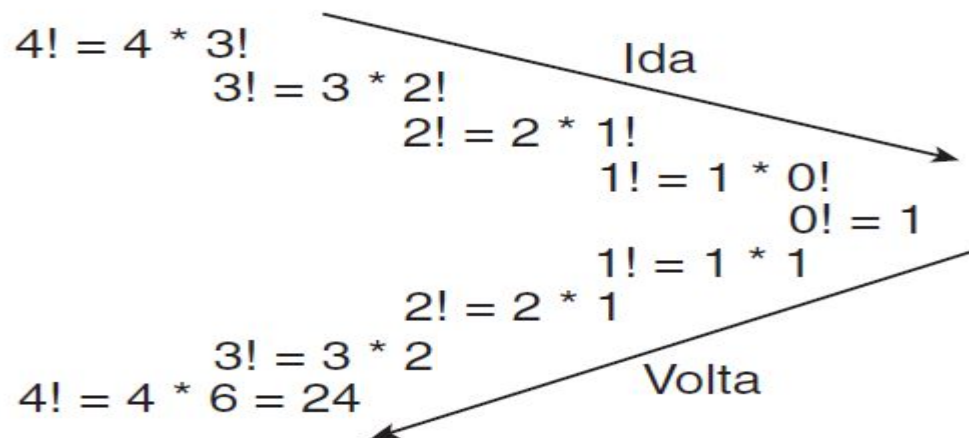
$$0! = 1$$

$$1! = 1 * 0!$$

$$2! = 2 * 1!$$

$$3! = 3 * 2!$$

$$4! = 4 * 3!$$



$$n! = n * (n - 1)! : \text{fórmula geral}$$

$$0! = 1 : \text{caso-base}$$



# Recursão

## Com Recursão

```
int fatorial(int n) {  
    if (n == 0) { // critério de  
        return 1; // parada  
    } else {  
        return n * fatorial(n-1);  
    }  
}
```

## Sem Recursão

```
int fatorial(int n) {  
    int i, f;  
    if (n == 0) {  
        return 1;  
    } else {  
        f = 1;  
        for (i = 1; i <= n; i++) {  
            f = f * i;  
        }  
        return f;  
    }  
}
```

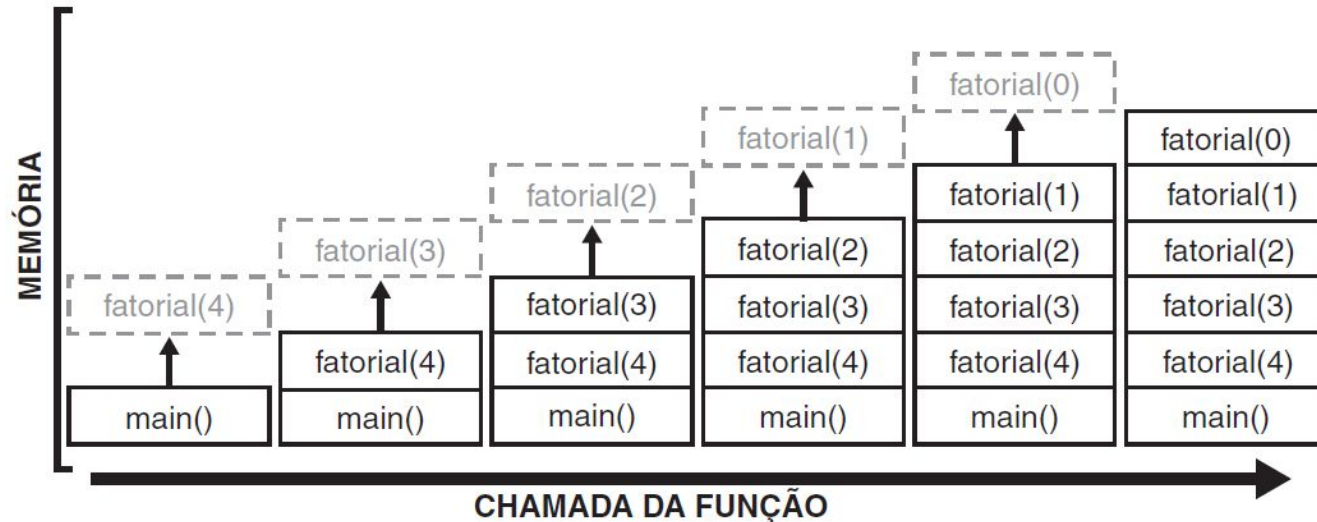
# Recursão

- Em geral, formulações recursivas de algoritmos são frequentemente consideradas "mais enxutas" ou "mais elegantes" do que formulações iterativas.
- Porém, algoritmos recursivos tendem a necessitar de mais espaço em memória do que algoritmos iterativos.

# Recursão

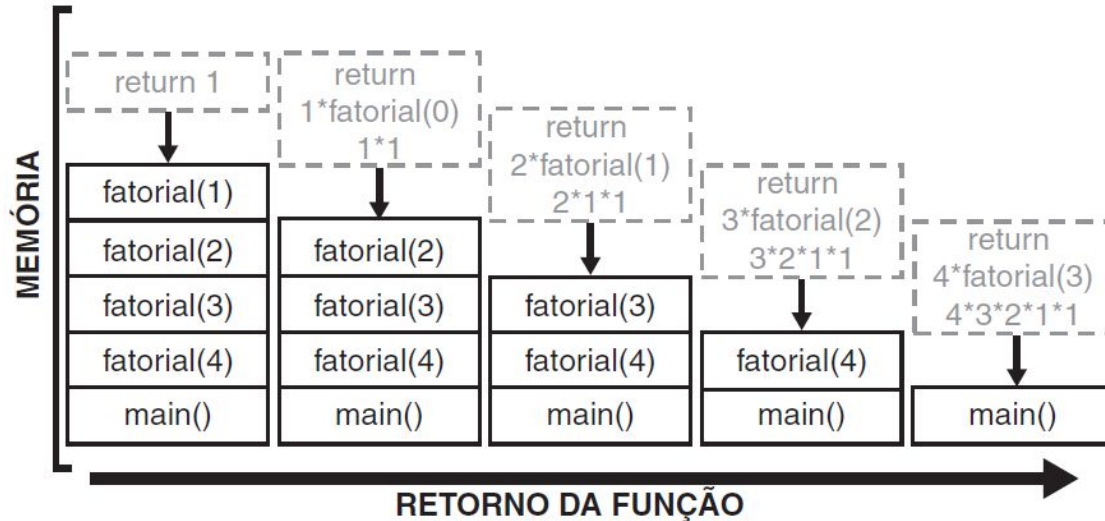
- O que acontece na chamada da função `fatorial` com um valor como `n = 4`?

```
int x = fatorial(4);
```



# Recursão

- Uma vez que chegamos ao caso-base, é hora de fazer o caminho de volta da recursão.



# Recursão

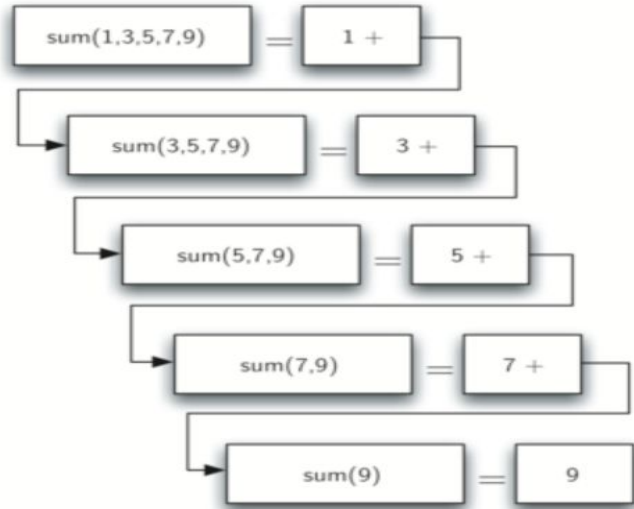
- Todo cuidado é pouco ao se fazer funções recursivas.
  - **Critério de parada(caso base):** determina quando a função deverá parar de chamar a si mesma.
  - **O parâmetro da chamada recursiva deve ser sempre modificado,** de forma que a recursão chegue a um término.

```
int fatorial(int n) {  
    if (n == 0) { // critério de parada  
        return 1;  
    } else { // parâmetro de fatorial sempre muda  
        return n * fatorial(n-1);  
    }  
}
```

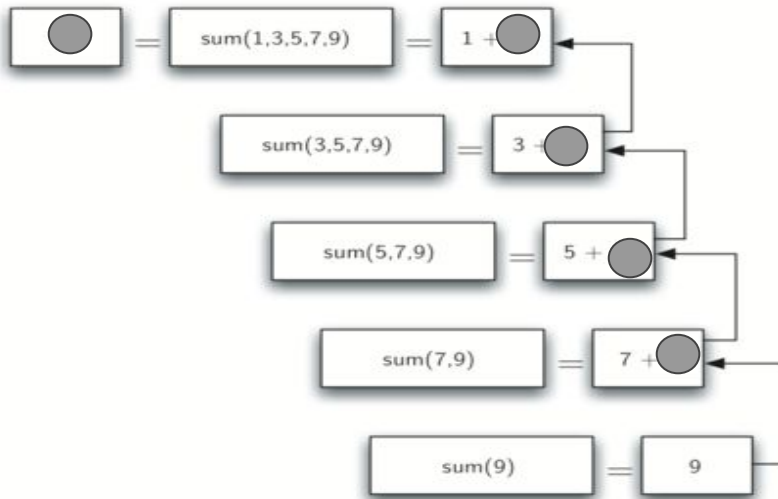
# Exemplo

- Soma dos elementos de uma lista
  - Como podemos implementar uma função que soma os elementos de uma lista recursivamente, sem usar estrutura de repetição?

# Soma recursiva



# Solução recursiva



Série de Retornos Recursivos da Adição de uma Lista de Números



# Soma recursiva

**Caso base:** vetor com 1 elemento, logo a soma é o próprio elemento.

**Caso geral:**

- retira o primeiro elemento e chama a função recursivamente para o resto do conjunto.
- retorna a soma do elemento retirado com a soma retornada na chamada recursiva

```
int soma (int v[], int tam)
{
    if (tam == 1)
    {
        return v[0];
    }
    else
    {
        return v[0] + soma (&v[1], tam - 1);
    }
}
```

# Fibonacci

- Essa seqüência é um clássico da recursão
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- A sequência de Fibonacci é definida como uma função recursiva utilizando a fórmula a seguir

$$F(n) = \begin{cases} 0, & \text{se } n = 0 \\ 1, & \text{se } n = 1 \\ F(n-1) + F(n-2), & \text{outros casos} \end{cases}$$

# Recursão

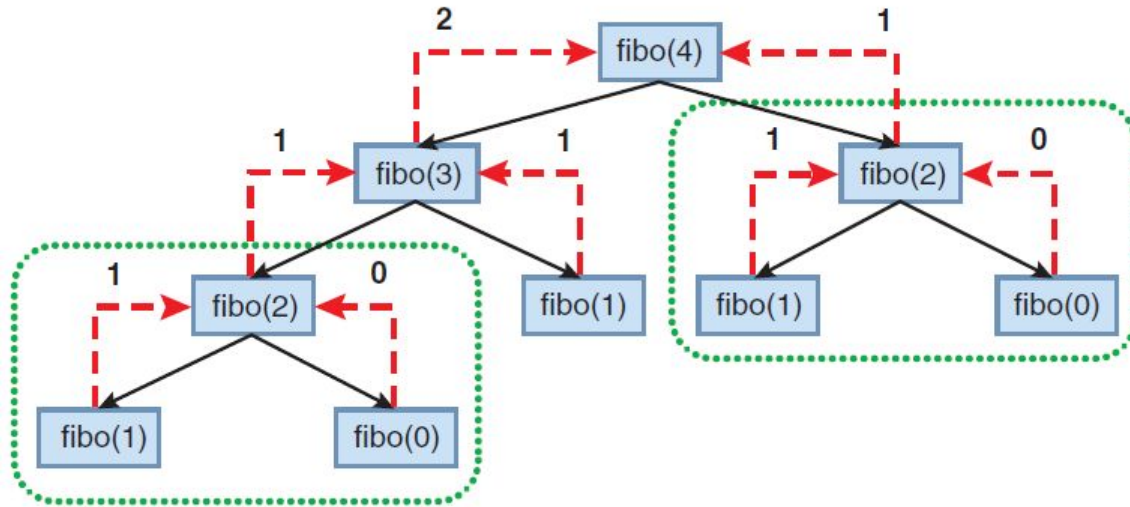
## Sem Recursão

```
int fibo(int n){  
    int i,a=0,b=1,c;  
    for (i=0;i<n;i++){  
        c=a+b;  
        a=b;  
        b=c;  
    }  
    return a;  
}
```

## Com Recursão

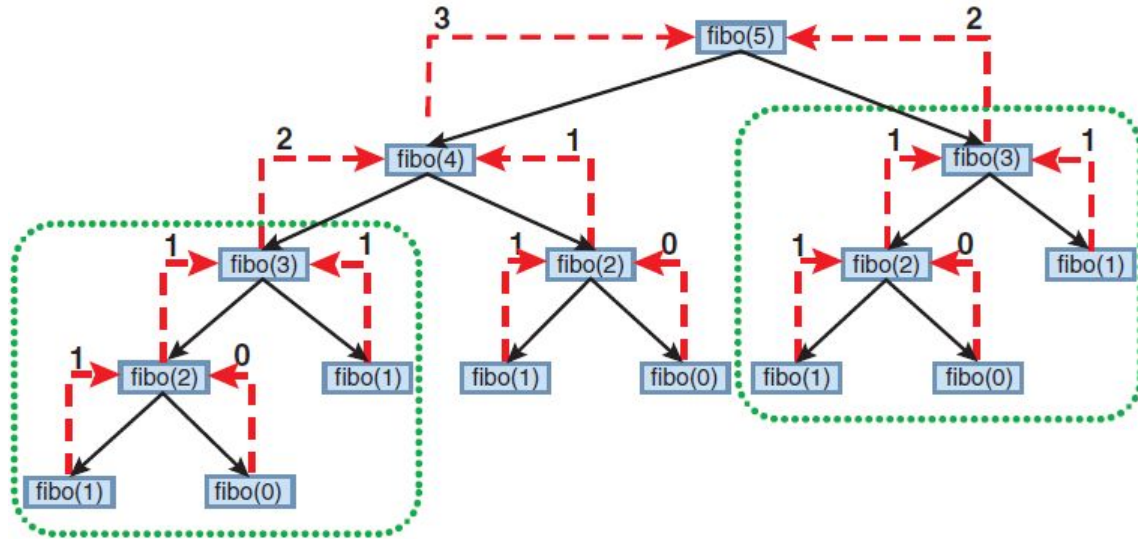
```
int fiboR(int n){  
    if((n==0)||(n==1)){  
        return n;  
    }  
    else{  
        return fiboR(n-1) + fiboR(n-2);  
    }  
}
```

# Fibonacci



# Fibonacci

- Aumentando para **fibo(5)**



# Exercício

Escreva uma função recursiva que procure um valor em um vetor, não importando a ordem em que os elementos estão distribuídos. A função deverá retornar 1 se o valor for encontrado e 0, caso contrário.

# Busca recursiva 1

**Caso base:** vetor de tamanho 0 retorna 0 (falso).

**Caso geral:** caso  $v[0]$  seja o elemento procurado, retorna 1 (verdadeiro). Caso não seja, chama a função recursivamente para o vetor com 1 elemento a menos (remove o elemento  $v[0]$ , que acabou de ser verificado)

```
int busca (int v[], int tam, int x)
{
    if (tam == 0)
    {
        return 0;
    }
    else
    {
        if (v[0]==x)
        {
            return 1;
        }
        else
        {
            return busca (&v[1], tam - 1,x);
        }
    }
}
```

```
int busca(int v[], int tam,int x){  
    if (tam == 0){  
        return 0;  
    }  
    else  
        return (v[0] == x)?1:busca(&v[1],tam-1,x);  
}
```

Soluções  
equivalentes

```
int busca(int v[], int tam,int x){  
    if (tam == 0){  
        return 0;  
    }  
    else  
        return (v[tam-1] == x)?1:busca(v,tam-1,x);  
}
```



# Exercício

Escreva uma função recursiva que inverta os elementos de um vetor.

# Inverte recursivo

**Caso base:** vetor com 0 ou 1 elemento, não faz nada.

**Caso geral:** troca o primeiro com o último e chama recursivamente para os demais elementos do conjunto (retirando o primeiro e último)

```
void inverte (int v[], int tam)
{
    int aux;

    if (tam > 1)
    {
        aux = v[0];
        v[0] = v[tam - 1];
        v[tam - 1] = aux;
        inverte (&v[1], tam - 2);
    }
}
```

# Inverte recursivo - outra solução

**Caso base:** índice do elemento inicial é maior ou igual ao índice do elemento final, não faz nada.

**Caso geral:** troca o primeiro com o último e chama recursivamente para os demais elementos do conjunto (retirando o primeiro e último)

```
void inverte (int v[], int ini, int fim)
{
    int aux;

    if (ini < fim)
    {
        aux = v[ini];
        v[ini] = v[fim];
        v[fim] = aux;
        inverte (v, ini + 1, fim - 1);
    }
}
```

# Exercício

Escreva uma função recursiva que retorne o menor elemento de um vetor.

# Menor recursivo

**Caso base:** vetor com 1 elemento, logo o menor.

**Caso geral:**

compara o primeiro elemento com o último. Chama a função recursivamente retirando do vetor o maior dos dois elementos comparados.

```
int menor(int v[], int tam){  
    if (tam == 1){  
        return v[0];  
    }  
    else  
        if (v[0] < v[tam-1])  
            return menor(v,tam-1);  
        else  
            return menor(&v[1],tam-1);  
}
```

# Exercícios

1. Defina uma função recursiva que receba um vetor  $V$  de inteiros, o tamanho de  $V$  e um inteiro  $x$  como parâmetros e conta quantas vezes  $x$  ocorre em  $V$ .
2. Defina uma função recursiva que receba uma string como parâmetro e retorne 1 caso a string seja um palíndromo, e 0 caso contrário.