



Aula 6 - Recursividade

David Menotti

Algoritmos e Estruturas de Dados I

DECOM – UFOP

Conceito de Recursividade

- Fundamental em Matemática e Ciência da Computação
 - Um programa recursivo é um programa que chama a si mesmo
 - Uma função recursiva é definida em termos dela mesma
- Exemplos
 - Números naturais, Função fatorial, Árvore
- Conceito poderoso
 - Define conjuntos infinitos com *comandos finitos*

$$N! = N \times (N-1)! \quad N > 0$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1!$$

$$1! = 1 \times 0!$$

$$0! = 1$$

} passo base

} recursivo

} passo

Recursividade

- A recursividade é uma estratégia que pode ser utilizada sempre que o cálculo de uma função para o valor n , pode ser descrita a partir do cálculo desta mesma função para o termo anterior ($n-1$).

Exemplo – Função fatorial:

$$n! = n * (n-1) * (n-2) * (n-3) \\ * \dots * 1$$

$$(n-1)! = (n-1) * (n-2) * (n-3) \\ * \dots * 1$$

logo:

$$n! = n * (n-1)!$$

Algoritmos e Estrutura
de Dados I

Recursividade

- Definição: dentro do corpo de uma função, chamar novamente a própria função
 - recursão direta: a função A chama a própria função A
 - recursão indireta: a função A chama uma função B que, por sua vez, chama A

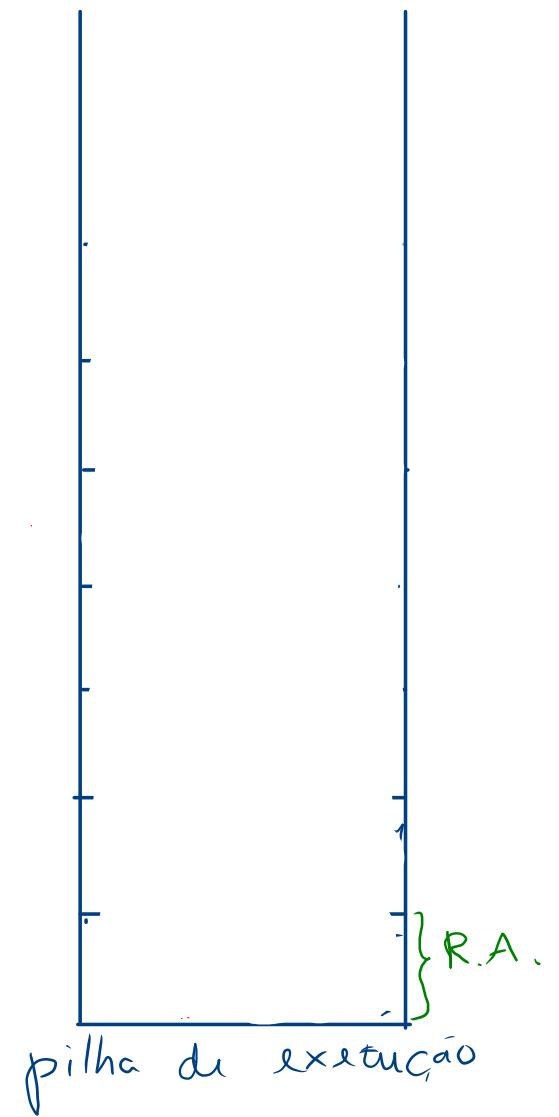
Condição de parada

- Nenhum programa nem função pode ser exclusivamente definido por si
 - Um programa seria um loop infinito
 - Uma função teria definição circular
- Condição de parada (*passo base, ponto de parada*)
 - Permite que o procedimento pare de se executar
 - $F(x) > 0$ onde x é decrescente
- Objetivo
 - Estudar recursividade como ferramenta *prática!*

Recursividade

- Para cada chamada de uma função, recursiva ou não, os parâmetros e as variáveis locais são empilhados na pilha de execução.

```
... main (...) {  
    fatn(5);  
}  
;
```



Execução

- Internamente, quando qualquer chamada de função é feita dentro de um programa, é criado um **Registro de Ativação na Pilha de Execução** do programa
- O registro de ativação armazena os parâmetros e variáveis locais da função bem como o “ponto de retorno” no programa ou subprograma que chamou essa função.
- Ao final da execução dessa função, o registro é desempilhado e a execução volta ao subprograma que chamou a função



UFOP

Exemplo

```
Fat (int n) {
    if (n<=0)
        return 1;    CP
    else
        return n * Fat(n-1); Passo recursivo
}
```

```
Main() {
    int f;
    f = fat(5);
    printf("%d", f);
}
```

Algoritmos e Estrutura
de Dados I

Complexidade

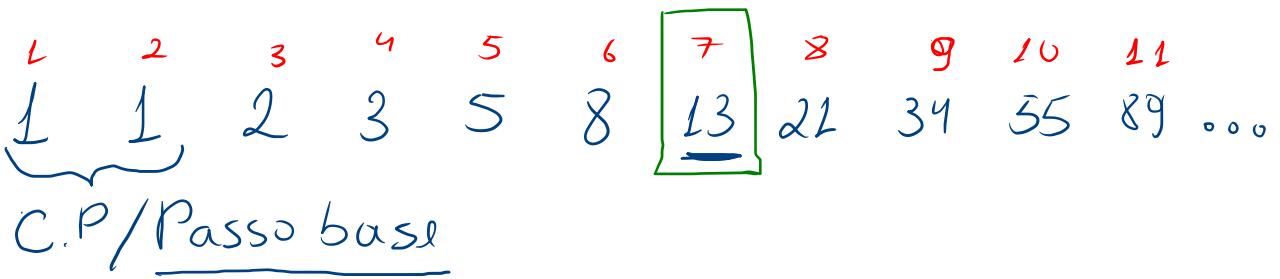
- A complexidade de tempo do fatorial recursivo é $O(n)$.
(Em breve iremos ver a maneira de calcular isso usando **equações de recorrência**)
- Mas a complexidade de espaço também é $O(n)$, devido a pilha de execução
- Já no fatorial não recursivo a complexidade de espaço é $O(1)$

```
fat (int n) {
    int f;
    f = 1;
    while (n > 0) {
        f = f * n;
        n = n - 1;
    }
    return f;
}
```

Algoritmos e Estrutura
de Dados I

Recursividade

- Portanto, a recursividade nem sempre é a melhor solução, mesmo quando a definição matemática do problema é feita em termos recursivos

Fibonacci: 

$$\text{Fibo}(7) = 13$$

$\hookrightarrow \text{Fibo}(5) + \text{Fibo}(6)$

$$\text{Fibo}(N) =$$

$\hookrightarrow \text{Fibo}(N-1) + \text{Fibo}(N-2)$

$$\text{Fibo}(347) =$$

$\hookrightarrow \text{Fibo}(346) + \text{Fibo}(345)$

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

> passo base

$$\text{Fibo}(2) = 1$$

> passo base

$$\begin{array}{c}
 \text{Fib}_0(-3) \\
 \swarrow \quad \searrow \\
 \text{Fib}_0(-4) + \text{Fib}_0(-5) \\
 \swarrow \quad \searrow \\
 \text{Fib}_0(-5) + \text{Fib}_0(-6)
 \end{array}$$

$$\begin{array}{c}
 \text{Fib}_0(5) = 5 \\
 \swarrow \quad \searrow \\
 \text{Fib}_0(4) + \text{Fib}_0(3) = 2 \\
 \swarrow \quad \searrow \\
 \text{Fib}_0(3) + \text{Fib}_0(2) = 1 \quad \text{Fib}_0(2) + \text{Fib}_0(1) = 1 \\
 \swarrow \quad \searrow \quad \swarrow \quad \searrow \\
 \text{Fib}_0(2) + \text{Fib}_0(1) = 1 \quad = 1
 \end{array}$$

$\text{Fib}_0(5)$ $\text{f}(4) + \text{f}(3)$ $\text{main}()$ $\text{fib}(5)$



UFOP

Fibonacci

- Outro exemplo: **Série de Fibonacci:**

- $\square F_n = F_{n-1} + F_{n-2} \quad n > 2,$
- $\square F_0 = 0 \quad F_1 = 1$
- $\square 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89\dots$

```
Fib(int n) {  
    if (n<2)  
        return 1;  
    else  
        return Fib(n-1) + Fib(n-2);  
}
```

Algoritmos e Estrutura
de Dados I

Análise da função Fibonacci

■ Ineficiência em Fibonacci

- Termos F_{n-1} e F_{n-2} são computados independentemente
- Número de chamadas recursivas = número de Fibonacci!
- Custo para cálculo de F_n
 - $O(\varphi^n)$ onde $\varphi = (1 + \sqrt{5})/2 = 1,61803\dots$
 - *Golden ratio*
 - Exponencial!!!

Análise da função Fibonacci

■ Ineficiência em Fibonacci

- Termos F_{n-1} e F_{n-2} são computados independentemente
- Número de chamadas recursivas = número de Fibonacci!
- Custo para cálculo de F_n
 - $O(\varphi^n)$ onde $\varphi = (1 + \sqrt{5})/2 = 1,61803\dots$
 - *Golden ratio*
 - Exponencial!!!

Fibonacci não recursivo

```
int FibIter(int n) {  
    int i, k, F;  
  
    i = 1; F = 0;  
    for (k = 1; k <= n; k++) {  
        F += i;  
        i = F - i;  
    }  
    return F;  
}
```

- Complexidade: $O(n)$
- Conclusão: não usar recursividade cegamente!

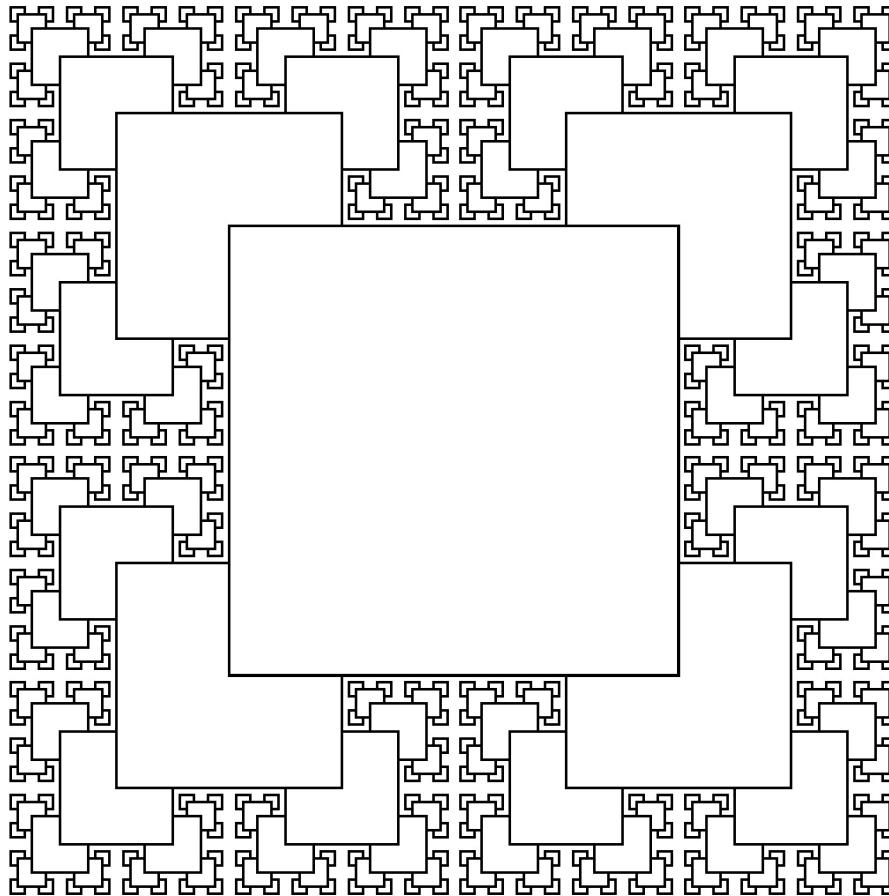
Quando vale a pena usar recursividade

- Recursividade vale a pena para Algoritmos complexos, cuja a implementação iterativa é complexa e normalmente requer o uso explícito de uma pilha
 - Dividir para Conquistar (Ex. Quicksort)
 - Caminhamento em Árvores (pesquisa, backtracking)

Dividir para Conquistar

- Duas chamadas recursivas
 - Cada uma resolvendo a metade do problema
- Muito usado na prática
 - Solução eficiente de problemas
 - Decomposição
- Não se reduz trivialmente como fatorial
 - Duas chamadas recursivas
- Não produz recomputação excessiva como fibonacci
 - Porções diferentes do problema

Outros exemplos de recursividade

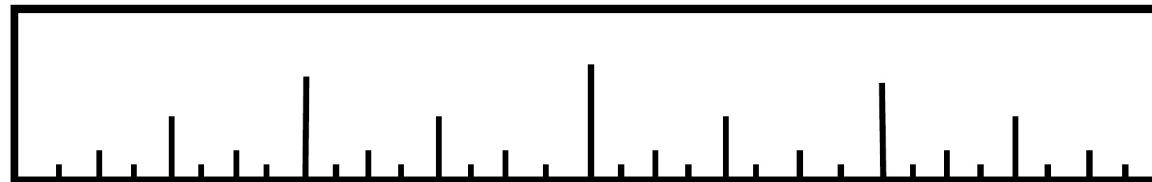


```
void estrela(int x,int y, int r)
{
    if ( r > 0 )
    {
        estrela(x-r, y+r, r div 2);
        estrela(x+r, y+r, r div 2);
        estrela(x-r, y-r, r div 2);
        estrela(x+r, y-r, r div 2);
        box(x, y, r);
    }
}
```

Algoritmos e Estrutura
de Dados I

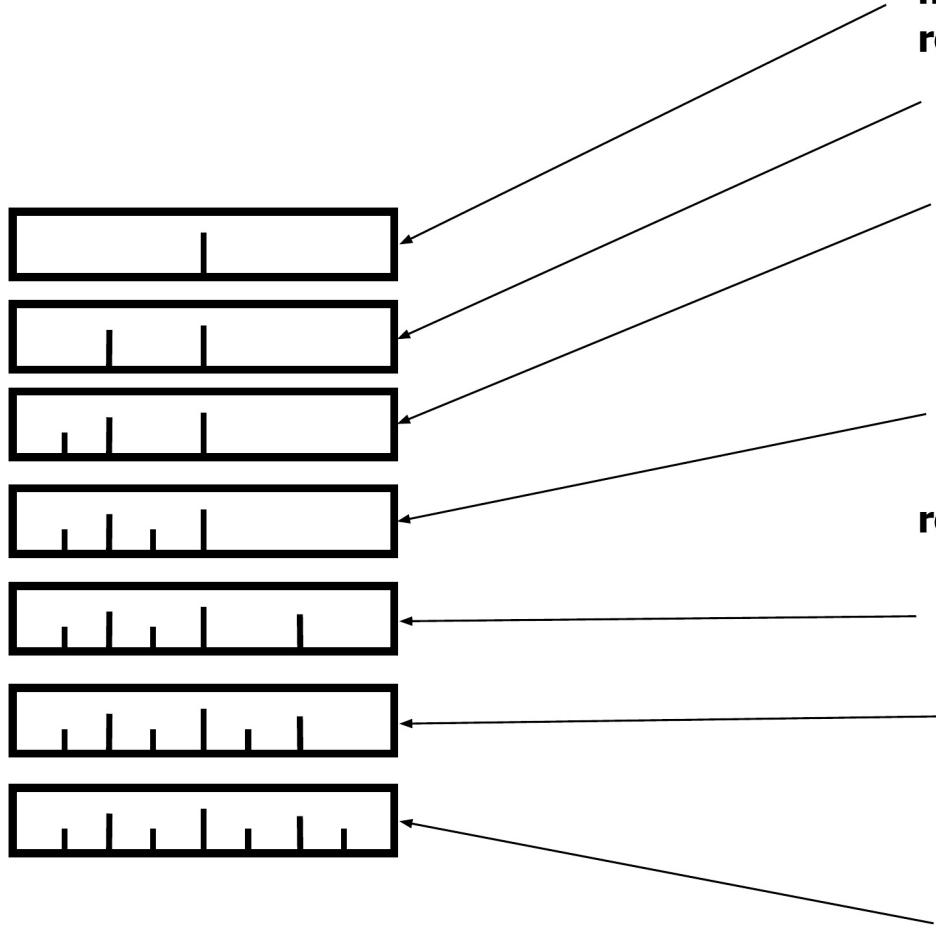
Exemplo simples: régua

```
int regua(int l,int r,int h)
{
    int m;
    if ( h > 0 )
    {
        m = (l + r) / 2;
        marca(m, h);
        regua(l, m, h - 1);
        regua(m, r, h - 1);
    }
}
```



Algoritmos e Estrutura
de Dados I

Execução: régua



Algoritmos e Estrutura
de Dados I