

# 04 – Recursão

## SCC201/501 - Introdução à Ciência de Computação II

Prof. Moacir Ponti Jr.  
[www.icmc.usp.br/~moacir](http://www.icmc.usp.br/~moacir)

Instituto de Ciências Matemáticas e de Computação – USP

2010/2



## 1 Recursividade

- Definições
- Recursão e computação

## 2 Exemplos e Implementação

- Exemplo: cálculo do fatorial
- Implementação de Recursividade
- Exemplo: sequência de Fibonacci

## 3 Questões importantes

- Quando não usar recursividade
- Recursão indireta
- Recursão infinita
- Ordem da chamada recursiva

# Préviews: notícias e páginas interessantes a visitar

- Foi provado que qualquer posição do Cubo Mágico pode ser resolvida com 20 movimentos. <http://www.reddit.com/tb/cz011>
- “Martin e o Dragão”, série de três contos sobre recursão (Prof. Ruiter – UFAM),
  - <http://www.dcc.ufam.edu.br/~ruiter/icc/martin0.html>
  - <http://www.dcc.ufam.edu.br/~ruiter/icc/martin1.html>
  - <http://www.dcc.ufam.edu.br/~ruiter/icc/martin2.html>
- Uncyclopedia sobre recursão finita.  
[http://uncyclopedia.wikia.com/wiki/Finite\\_recursion](http://uncyclopedia.wikia.com/wiki/Finite_recursion)



# Definições

- Em *matemática*, pode ser definida como: **o ato de definir um objeto (geralmente uma função), em termos do próprio objeto.**
- Em *computação*, ocorre quando: **um dos passos de um determinado algoritmo envolve a repetição desse mesmo algoritmo**
- Um procedimento que se utiliza da recursão é dito recursivo.
- Também é dito recursivo qualquer objeto que seja resultado de um procedimento recursivo.



# Definições

- É possível, por meio de recursão, obter um objeto ou sequências infinitas a partir de um componente finito.
- O conjunto dos números naturais, por exemplo, pode ser definido formalmente (de maneira resumida), por:
  - Seja 0 um número natural. Cada número natural,  $n$  tem um sucessor  $n + 1$ , que é também um número natural.



## Caso base

- Parte não recursiva, também chamada de âncora, ocorre quando a resposta para o problema é trivial.

## Passo indutivo

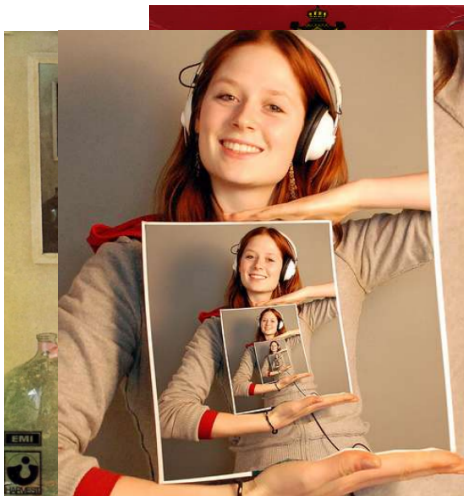
- Parte da definição que especifica como cada elemento (solução) é gerado a partir do precedente.
- A função fatorial  $n!$  pode ser definida como, dado um número inteiro positivo  $n$ :

$$n! = \begin{cases} 1, & \text{se } n = 0 \text{ (caso base),} \\ n \cdot (n-1)!, & \text{se } n > 0 \text{ (passo de indução).} \end{cases} \quad (1)$$



# Aplicação visual

- Efeito “Droste”



## Definição de: Recursão

- Se você ainda não entendeu; ver “Recursão”.



About 2,460,000 results (0.29 seconds) [Advanced search](#)

Everything

☒ More

Any time

Past 2 days

All results

Related searches

Wonder wheel

Timeline

☒ More search tools

Did you mean: [Recursion](#)

[Recursion](#) - Wikipedia, the free encyclopedia ☆

**Recursion**, in mathematics and computer science, is a method of defining functions in which the function being defined is applied within its own definition; ...  
[en.wikipedia.org/wiki/Recursion](#) - [Cached](#) - [Similar](#)

[Recursion \(computer science\)](#) - Wikipedia, the free encyclopedia ☆

**Recursion** in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem. ...  
[en.wikipedia.org/wiki/Recursion\\_\(computer\\_science\)](#) - [Cached](#) - [Similar](#)  
[+ Show more results from en.wikipedia.org](#)

[Recursion -- from Wolfram MathWorld](#) ☆

A **recursive** process is one in which objects are defined in terms of other objects of the same type. Using some sort of recurrence relation, the entire class ...  
[mathworld.wolfram.com](#) > ... > [Algorithms](#) > [Recursion](#) - [Cached](#) - [Similar](#)

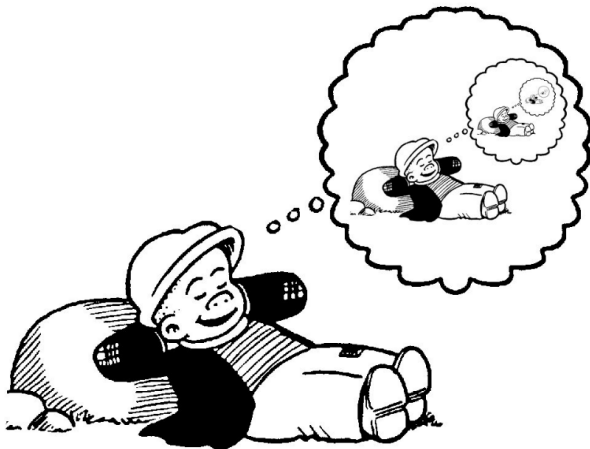
[recursion](#) ☆

Definition of **recursion**, possibly with links to more information and





# Humor recursivo



## Gramáticas de linguagens de programação

- Na especificação de gramáticas de linguagens de programação se utiliza recursão para modelar a estrutura de expressões e declarações.

$$\begin{aligned}\langle \text{expr} \rangle &::= \langle \text{numero} \rangle \\ &\quad | (\langle \text{expr} \rangle * \langle \text{expr} \rangle) \\ &\quad | (\langle \text{expr} \rangle + \langle \text{expr} \rangle)\end{aligned}$$

- O exemplo acima mostra que a expressão pode ser um número, o produto de duas expressões ou a soma de duas expressões.
- A referência recursiva à  $\langle \text{expr} \rangle$  permite expressões arbitrariamente complexas com mais de um produto ou soma em uma única expressão, como:  $(5 * ((3 * 6) + 8))$
- Qual é o caso base? E o que ele representa?



## Problemas com estrutura recursiva

- Propriedade: cada instância do problema contém uma instância menor do mesmo problema.

## Resolução de problemas recursivos

- se a instância é pequena, resolva-a diretamente (caso base)
- senão
  - 1 reduza-a a uma instância menor do mesmo problema,
  - 2 aplique o método à instância menor, e
  - 3 volte à instância original.

## Algoritmo recursivo

- O uso da estratégia acima produz um algoritmo recursivo que é **caracterizado por possuir uma chamada a si mesmo.**

# Recursão e computação: as três regras

## 1 – Saber quando parar.

- qualquer função recursiva deve verificar se a jornada já terminou (caso base) antes da nova chamada recursiva.

## 2 – Decidir como fazer o primeiro passo

- pensar em como quebrar um problema em subproblemas que possam ser resolvidos instantaneamente.

## 3 – Analisar a jornada de forma que possa ser dividida em jornadas menores

- encontrar uma maneira da função chamar a si mesma (recursivamente), passando por parâmetro um problema menor resultante da segunda regra.

Fonte: <http://www.dcc.ufam.edu.br/~ruiter/icc/martin4.html>



# Exemplo: cálculo do fatorial

## Função recursiva para cálculo do fatorial

```
unsigned long fatorial(int n)
{
    unsigned long resultado = 1; // caso base
    if (n > 1) {
        resultado = n * fatorial(n-1); // passo indutivo
    }
    return resultado;
}
```

- Na função acima, é possível ver que o caso base é tomado como sendo o padrão.
- A seguir, se  $n > 1$ , então não estamos no caso base e assim o resultado será o produto de  $n$  por  $(n - 1)!$ .



# Implementação de Recursividade

- A forma de um compilador implementar um procedimento recursivo é por meio de uma **pilha**.
- Nessa pilha são armazenados os dados usados em cada chamada de uma função que ainda não terminou de processar.

fatorial(4)



# Implementação de Recursividade

- A forma de um compilador implementar um procedimento recursivo é por meio de uma **pilha**.
- Nessa pilha são armazenados os dados usados em cada chamada de uma função que ainda não terminou de processar.

```
fatorial(4)
```

```
[ 4 * fatorial(3) ]
```



# Implementação de Recursividade

- A forma de um compilador implementar um procedimento recursivo é por meio de uma **pilha**.
- Nessa pilha são armazenados os dados usados em cada chamada de uma função que ainda não terminou de processar.

```
fatorial(4)
```

```
[ 4 * fatorial(3) ]
```

```
[ 4 * < 3 * fatorial(2) > ]
```





# Implementação de Recursividade

- A forma de um compilador implementar um procedimento recursivo é por meio de uma **pilha**.
- Nessa pilha são armazenados os dados usados em cada chamada de uma função que ainda não terminou de processar.

```
fatorial(4)
```

```
[ 4 * fatorial(3) ]
```

```
[ 4 * < 3 * fatorial(2) > ]
```

```
[ 4 * < 3 * { 2 * fatorial (1) } > ]
```



# Implementação de Recursividade

- A forma de um compilador implementar um procedimento recursivo é por meio de uma **pilha**.
- Nessa pilha são armazenados os dados usados em cada chamada de uma função que ainda não terminou de processar.

```
fatorial(4)
```

```
[ 4 * fatorial(3) ]
```

```
[ 4 * < 3 * fatorial(2) > ]
```

```
[ 4 * < 3 * { 2 * fatorial (1) } > ]
```

```
[ 4 * < 3 * { 2 * 1 } > ]
```



# Implementação de Recursividade

- A forma de um compilador implementar um procedimento recursivo é por meio de uma **pilha**.
- Nessa pilha são armazenados os dados usados em cada chamada de uma função que ainda não terminou de processar.

```
fatorial(4)
```

```
[ 4 * fatorial(3) ]
```

```
[ 4 * < 3 * fatorial(2) > ]
```

```
[ 4 * < 3 * { 2 * fatorial (1) } > ]
```

```
[ 4 * < 3 * { 2 * 1 } > ]
```

```
[ 4 * < 3 * 2 > ]
```



# Implementação de Recursividade

- A forma de um compilador implementar um procedimento recursivo é por meio de uma **pilha**.
- Nessa pilha são armazenados os dados usados em cada chamada de uma função que ainda não terminou de processar.

```
fatorial(4)
[ 4 * fatorial(3) ]
[ 4 * < 3 * fatorial(2) > ]
[ 4 * < 3 * { 2 * fatorial (1) } > ]
[ 4 * < 3 * { 2 * 1 } > ]
[ 4 * < 3 * 2 > ]
[ 4 * 6 ]
```



# Implementação de Recursividade

- A forma de um compilador implementar um procedimento recursivo é por meio de uma **pilha**.
- Nessa pilha são armazenados os dados usados em cada chamada de uma função que ainda não terminou de processar.

```
fatorial(4)
[ 4 * fatorial(3) ]
[ 4 * < 3 * fatorial(2) > ]
[ 4 * < 3 * { 2 * fatorial (1) } > ]
[ 4 * < 3 * { 2 * 1 } > ]
[ 4 * < 3 * 2 > ]
[ 4 * 6 ]
[ 24 ]
```



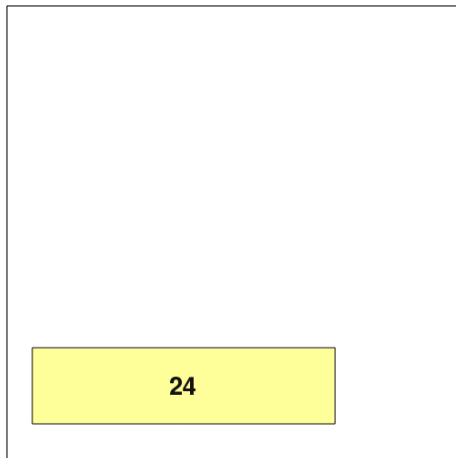
# Implementação de Recursividade

- A função começa a execução do seu primeiro comando cada vez que é chamada
- **Novas e distintas** cópias dos parâmetros passados por valor e variáveis locais são criadas
- A posição que chama a função é colocada em estado de espera — o nível gerado recursivamente é executado

```
fatorial(4)
[ 4 * fatorial(3) ]
[ 4 * < 3 * fatorial(2) > ]
[ 4 * < 3 * { 2 * fatorial (1) } > ]
[ 4 * < 3 * { 2 * 1 } > ]
[ 4 * < 3 * 2 > ]
[ 4 * 6 ]
[ 24 ]
```



# Implementação de Recursividade



# Sequência de Fibonacci

- Sequência numérica obtida de forma recursiva:

$$f(n) = \begin{cases} 0, & \text{se } n = 0, \\ 1, & \text{se } n = 1, \\ f(n-1) + f(n-2), & \text{se } n > 1. \end{cases} \quad (2)$$

- Inicialmente relacionado à velocidade de reprodução de coelhos e observado como sendo o modelo de muitos fenômenos biológicos, possui inúmeras aplicações na computação, matemática, teoria dos jogos, artes e música.
- Os primeiros dez termos da sequência são:  
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55





# Sequência de Fibonacci

- Um uso interessante é na conversão de milhas para quilômetros.
  - Para saber converter 5 mil. em km. aproximadamente, olha-se para o número seguinte ao número de Fibonacci correspondendo ao número de milhas: 5 mil. são aproximadamente 8 km.

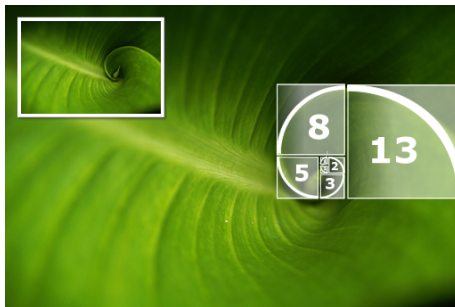


Figura: A evolução da espiral da folha da bromélia

# Sequência de Fibonacci

- Uma função que calcule o número de Fibonacci para qualquer valor de  $n$  pode ser construída diretamente usando uma estratégia recursiva:

$$f(n) = \begin{cases} 0, & \text{se } n = 0, \\ 1, & \text{se } n = 1, \\ f(n-1) + f(n-2), & \text{se } n > 1. \end{cases} \quad (3)$$

```
unsigned int Fib(unsigned int n)
{
    if (n<=1) return n; // caso base
    else
        return (Fib(n-1) + Fib(n-2)); // passo indutivo
}
```



# Sequência de Fibonacci

- Para valores de entrada pequenos, o programa é rápido. Já para  $n > 40$  o computador demora um certo tempo para processar.
- Vamos inserir mensagens de monitoramento na função para verificar.

```
unsigned int Fib(unsigned int n)
{
    printf("> Entrando em Fib(%d)\n", n);
    unsigned int F;

    if (n <= 1) F = n;
    else
        F = (Fib(n-1) + Fib(n-2));

    printf("<< Saindo de Fib(%d), retorno=%d\n", n, F);
    return F;
}
```



# Sequência de Fibonacci

- A saída do monitoramento, para  $n = 4$  é:

```
> Entrando em Fib(4)
> Entrando em Fib(3)
> Entrando em Fib(2)
> Entrando em Fib(1)
<< Saindo de Fib(1), retorno=1
> Entrando em Fib(0)
<< Saindo de Fib(0), retorno=0
<< Saindo de Fib(2), retorno=1
> Entrando em Fib(1)
<< Saindo de Fib(1), retorno=1
<< Saindo de Fib(3), retorno=2
```

```
> Entrando em Fib(2)
> Entrando em Fib(1)
<< Saindo de Fib(1), retorno=1
> Entrando em Fib(0)
<< Saindo de Fib(0), retorno=0
<< Saindo de Fib(2), retorno=1
<< Saindo de Fib(4), retorno=3
```



# Sequência de Fibonacci

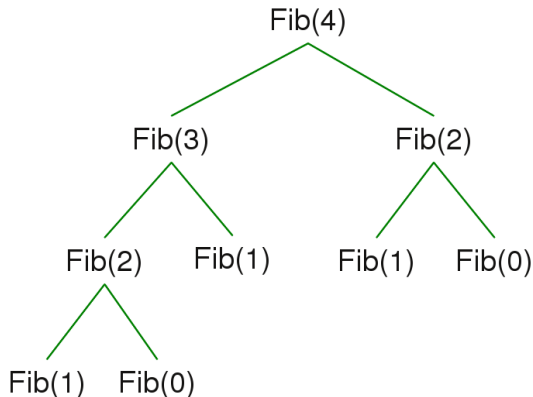


Figura: A árvore de chamadas recursivas à função  $\text{Fib}()$

# Sequência de Fibonacci: algoritmo iterativo

```
unsigned int Fib_iter(unsigned int n)
{
    unsigned int k, i = 1, F = 0;

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



# Quando não usar recursividade

- No exemplo anterior, a primeira implementação seguia a estrutura “natural” da sequência de Fibonacci. Mas, como vimos, nem sempre a estratégia recursiva é a melhor.
- Os problemas para os quais algoritmos recursivos devem ser evitados são ditos terem **recursividade de cauda**, e devem substituídos por uma versão iterativa.
  - nesse tipo de função, a chamada recursiva é a última instrução a ser executada.
- Há, no entanto, problemas para os quais é difícil ou impossível implementar uma solução não recursiva.



# Recursão indireta

- Funções podem ser recursivas indiretamente, fazendo isto através de outras funções: assim, P pode chamar Q que chama R e assim por diante, até que P seja novamente invocada.
- Um exemplo é a análise de expressões (como no exemplo anterior): um analisador gramatical para cada tipo de sub-expressão, uma expressão " $3 + (2 * (4 + 4))$ " é resolvida da seguinte forma:
  - 1 A função que processa expressões "+" chama uma segunda função que processa expressões "\*",
  - 2 A função de multiplicação, por sua vez, chama novamente a função de soma.





# Recursão infinita

- Repetição infinita causada por chamada recursiva.
- Ocorre quando o caso base não é definido (ou não é definido corretamente).
- Na prática, o programa não irá executar infinitamente, pois em algum momento alcançará o limite da **pilha**, e haverá um estouro de memória, causando um erro.
- Exemplo:

```
long int fatorial(int n)
{
    return n * fatorial(n-1);
}
```



# Ordem da chamada recursiva

```
void recursiva1(int num){  
    if (num <= 4) {  
        printf("%d\n", num);  
        recursiva1(num+1);  
    }  
}
```

```
recursiva1(1)  
printf(1)  
    recursiva1(1+1)  
printf(2)  
    recursiva1(2+1)  
printf(3)  
        recursiva1(3+1)  
printf(4)
```

```
void recursiva2(int num){  
    if (num <= 4) {  
        recursiva1(num+1);  
        printf("%d\n", num);  
    }  
}
```

```
recursiva2(1)  
    recursiva2(1+1)  
        recursiva2(2+1)  
            recursiva2(3+1)  
                printf(4)  
            printf(3)  
        printf(2)  
    printf(1)
```



# Exercícios

- (1) Implemente as versões recursiva e iterativa da função para obter números de Fibonacci. Utilize a biblioteca `time.h` para medir e observar o tempo necessário para calcular  $n = 15, 30, 45$  e  $60$ , utilizando as duas versões.
- (2) Implemente uma função recursiva para encontrar o maior elemento em um arranjo. *Dica:* encontre o maior elemento no subconjunto que contém todos, exceto o último elemento, então compare aquele máximo com o valor do último elemento.
- (3) Implemente uma função que exhibe todas as *substrings* de uma *string*. *Dica:* primeiro enumere todos os *substrings* que começam com o primeiro caractere. Existem  $n$  deles se o string tem tamanho  $n$ . Então, enumere as *substrings* da *string* após remover o primeiro caractere. Exemplo: substrings de rum:  
r, ru, rum, u, um, m

- ZIVIANI, N. **Projeto de algoritmos**: com implementações em Pascal e C (Seção 2.2). 2.ed. Thomson, 2004.
- CORMEN, T.H. et al. **Algoritmos: Teoria e Prática** (Seção 2.3.1). Campus. 2002.
- FEOFILOFF, P. **Recursão e algoritmos recursivos**. Disponível em: <http://www.ime.usp.br/~pf/algoritmos/aulas/recu.html>.
- CALDAS, R.B.. **Introdução a Computação**. Disponível em: <http://www.dcc.ufam.edu.br/~ruiter/icc/>.