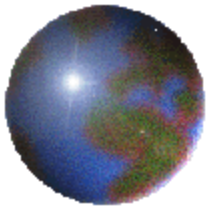
ESTRUTURAS DE DADOS I

Recursividade

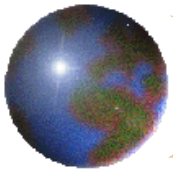


Professores

Leandro Luiz de Almeida

Mário Augusto Pazoti

Robson Augusto Siscoutto



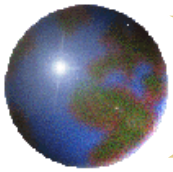
Recursividade

- Um objeto é dito recursivo se ele consistir parcialmente ou for definido em termos de si mesmo;
- Em termos de programação: a recursividade é uma técnica em que uma rotina, no processo de execução de suas tarefas, chama a si mesma.

Recursividade

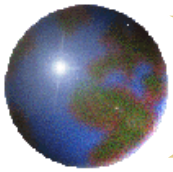
Uma Imagem Recursiva

(Fonte: WIRTH, N., 1993)



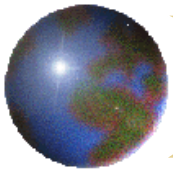
Recursividade

- A recursão é uma técnica poderosa em definições matemáticas;
- A recursão em programação se dá por meio de procedimentos ou funções (sub-rotinas);
- Assim pode-se dar um nome à um comando, comando esse que pode chamar a si próprio por meio deste nome.



Recursividade

- A recursão é uma técnica apropriada se o problema a ser resolvido tem as seguintes características:
  - A resolução dos casos maiores do problema envolve a resolução de um ou mais casos menores;
  - Os menores casos possíveis do problema podem ser resolvidos diretamente;
  - A solução iterativa do problema (usando enquanto, para ou repita) é complexa.



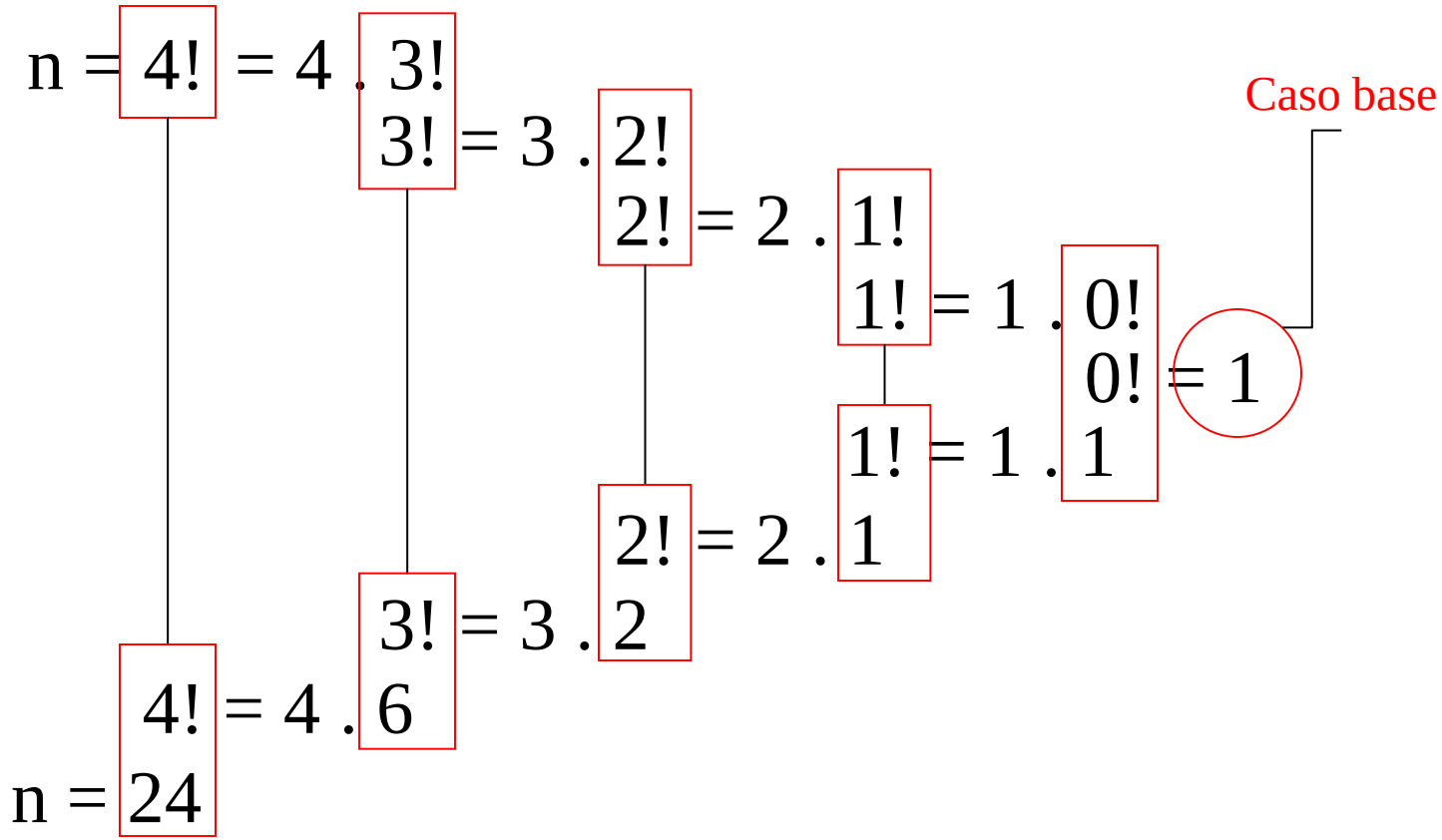
Recursividade

Como uma função recursiva pode chamar a si mesma indefinidamente, é essencial a existência do *caso base*, ou *condição de parada*. No caso do fatorial, o caso base é o zero, cujo valor do fatorial é 1. A partir dele, são encontrados todos os outros valores.

```
inteiro Fatorial(inteiro n)

Início
  Se (n == 0)     {Caso base, onde a recursão termina}
    Retorna (1);
  Senão           {Caso indutivo}
    Retorna (n * fatorial(n-1));
Fim
```

Recursividade

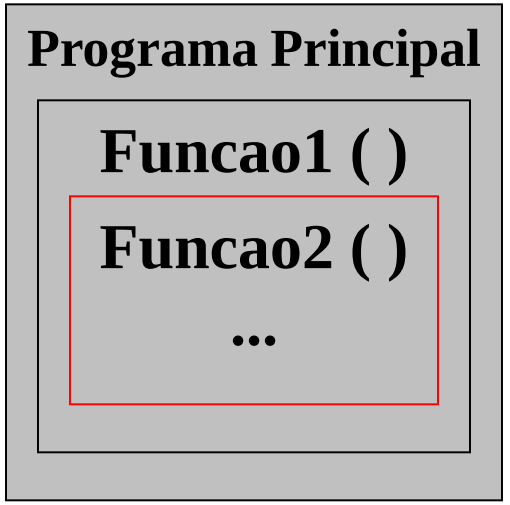
Ex: Fatorial de 4



Chamada de Funções

Quando uma função é chamada, esta é inserida na pilha e executada. Por isso, chamar uma função é torna-se mais lento do que escrever o código diretamente.

```
Funcao1()

Início
   Funcao2();
Fim;


Programa Principal

Início

   Funcao1();
Fim.
```

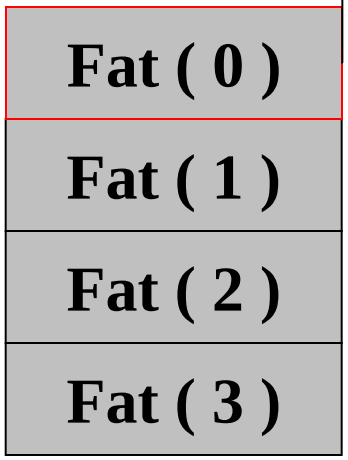


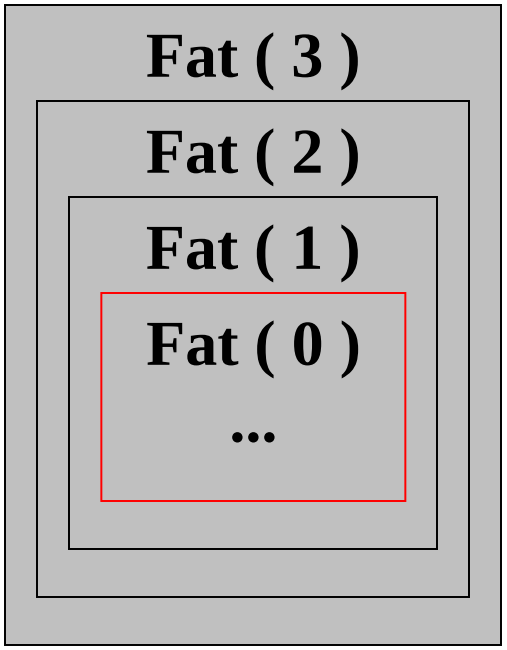
Funções Recursivas

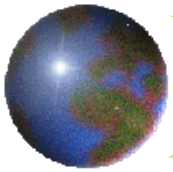
A cada chamada de uma função recursiva, ela é inserida na pilha e novas variáveis são alocadas.

```
Inteiro Fat(inteiro n)

Início

 Se (n == 0)
  Retorna(1);

 Senão
  Retorna(n*Fat(n-1));

Fim;
```

Topo:
Função atual







Recursividade

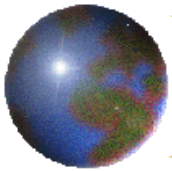
- Como uma rotina pode chamar a si mesma?
- Exemplo 1: cálculo do fatorial de um número;
  - 0! = 1
  - n>0: n! = n * (n-1)!

```
int Fat (int x)
{
    if (x==0 || x==1)
return 1;
    else
return x*Fat(x-1);
}
```

Recursiva

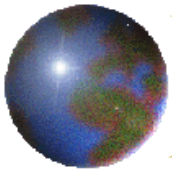
```
Int FAT (int X)
{
    int  i, ResFat;
    ResFat =1;
    for (i=X; i>=2; i--)
        ResFat:=ResFat*I;
   return ResFat;
}
```

Não Recursiva



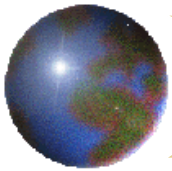
Recursividade

- Cuidado: rotinas recursivas requerem área para armazenamento dos valores dos objetos a cada chamada;
- Isto acarreta num risco de exceder o espaço disponível e ocorrer uma falha no programa.



Tipos de Recursão

- Se um procedimento P contiver uma referência explícita a si próprio, este procedimento é dito diretamente recursivo;
- Se P contiver uma referência a outro procedimento Q, que por sua vez contém uma referência direta ou indireta a P, então P é dito indiretamente recursivo.



Recursividade

- Um requisito fundamental é que, evidentemente, chamadas recursivas de um procedimento P estejam sujeitas à uma condição B, a qual, em algum instante, se torne falsa.

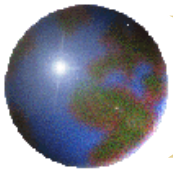
"Finitude dinâmica"

Recursividade

- Outro exemplo:definição de caixas para a construção de listas;

```
struct TipoRegistro
{
    char Nome[20], Endereco;
}

struct Caixa
{
    TipoRegistro Reg;
    Caixa *Prox;
}
```

Indiretamente Recursiva



Recursividade

Recursividade é a propriedade que uma função tem de chamar a si própria, diretamente ou não. Isto é usado para simplificar um problema.

Exemplo mais comum de recursão: Função Fatorial

---

0 ! = 1 — Caso base

1 ! = 1 . 0 ! = 1

2 ! = 2 . 1 ! = 2 . 1

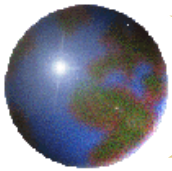
3 ! = 3 . 2 ! = 3 . 2 . 1

4 ! = 4 . 3 ! = 4 . 3 . 2 . 1

Regra Geral:

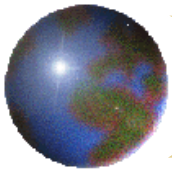
n ! = n . (n-1) !

Fat(n) = n * Fat(n-1)



Vantagens da Recursão

- Simplifica a solução de alguns problemas;
- Geralmente, um código com recursão é mais conciso;
- Caso não seja usada, em alguns problemas, é necessário manter o controle das variáveis manualmente (book keeping).



Desvantagens da Recursão

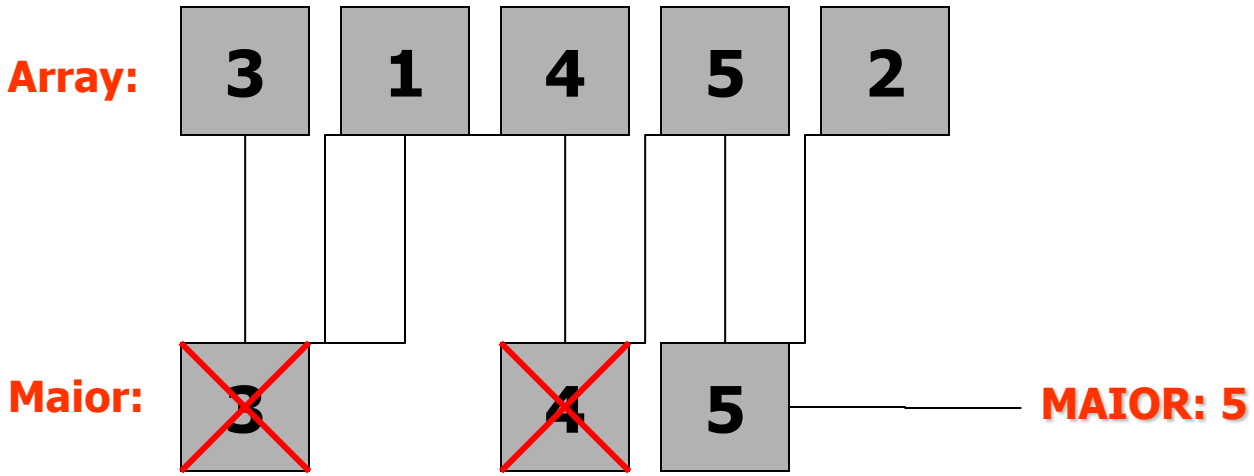
- Funções recursivas são mais lentas que funções iterativas, pois muitas chamadas consecutivas a funções são feitas;
- Erros de implementação podem levar a estouro de pilha. Isto é, caso não seja indicada uma condição de parada, ou se esta condição nunca for satisfeita, entre outros. Ex: fatorial de um número negativo.

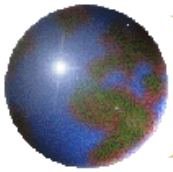
Recursividade Direta

Exemplo 2: encontrar o maior elemento em um array

Modo iterativo:

Basta percorrer o *array* com um laço (*for*, *while* ou *repeat*), e comparar cada elemento com o *maior* já encontrado:





Recursividade Direta

Exemplo 2: encontrar o maior elemento em um array.

A - Modo iterativo:

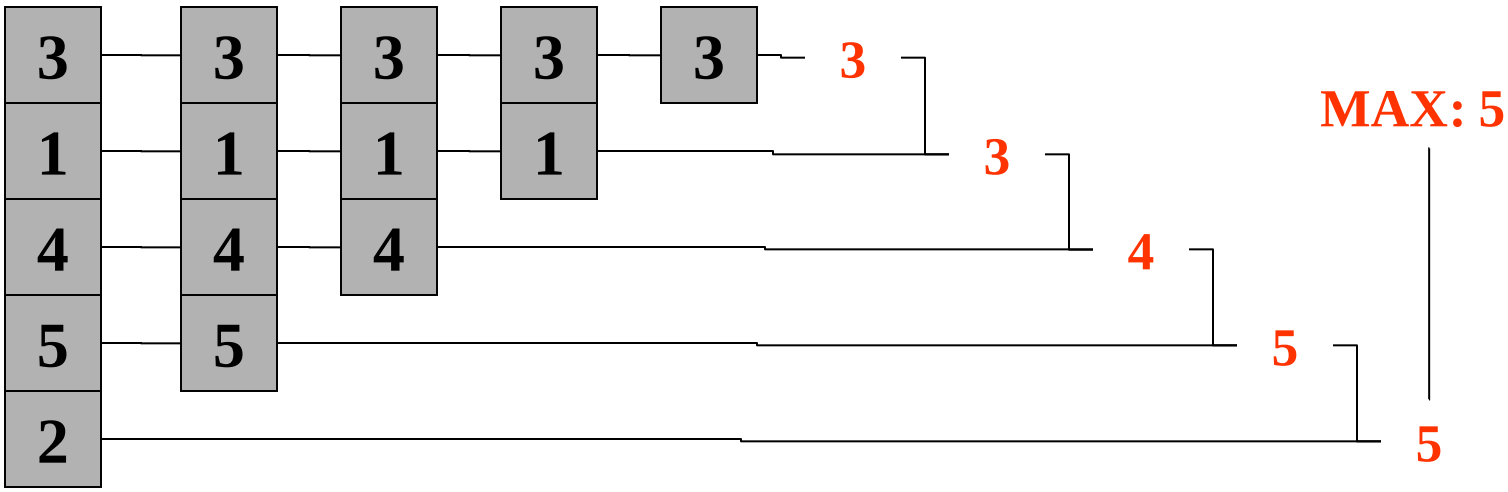
```
int MaiorElemento(TpVetor Vetor[100], int TL)
{
   int Maior,i;
    Maior = Vetor[0];
    for (i=1; i<TL; i++)
          if (Maior < Vetor[i])
                Maior = Vetor[i];
    return Maior;
}
```

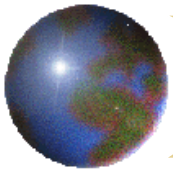
Recursividade Direta

Exemplo 2: encontrar o maior elemento em um array.

B - Modo recursivo:

O *maior* de um array é o maior entre o último elemento e o *maior* entre os elementos restantes. Quando houver apenas um elemento, este é o *maior*.





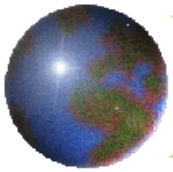
Recursividade Direta

Exemplo 2: encontrar o maior elemento em um array.

B - Modo recursivo:

```
int MaiorElemento(TpVetor Vetor[100], int TL)
{
    if (TL == 0)
     return Vetor[0];
    else return Maior(Vetor[TL-1],MaiorElemento(Vetor, TL-1));
}
```

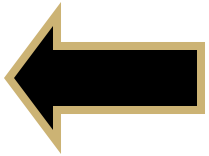
Obs.: A Função ***MAIOR*** retorna o maior elemento entre 2 números.

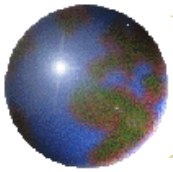


Recursividade Direta

Função para determinar o maior elemento entre dois valores.

```
int Maior(int Elem1, int Elem2)
{
    If (Elem1 > Elem2)
     return Elem1;
     else  return Elem2;
}
```

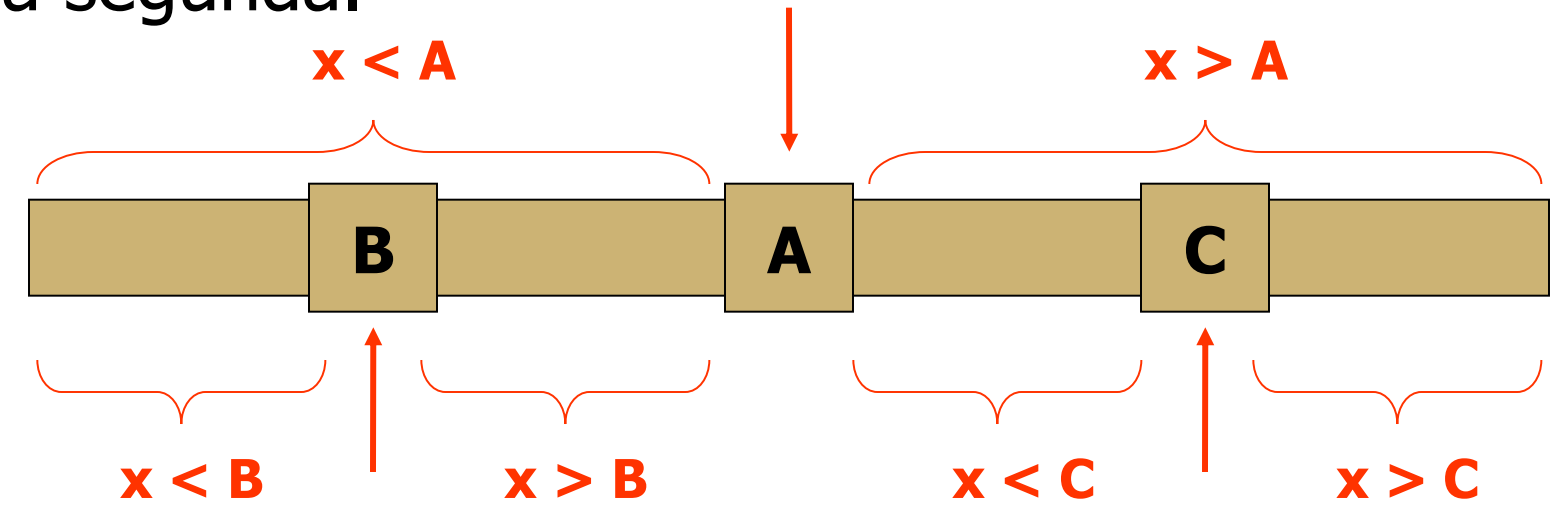


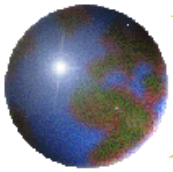


Recursividade Direta

Exemplo 3: Busca Binária em um array ordenado.

A busca é iniciada pelo elemento central. Se o elemento procurado for menor, procura-se novamente na primeira metade, caso contrário, na segunda.



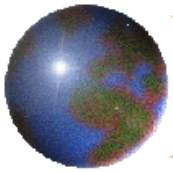


Recursividade Direta

Exemplo 3: Busca Binária em um array ordenado.

- Modo recursivo:

```
int BB(TpVetor Vetor[100], int Inicio, int Fim, int Elemento)
{
    int  Meio;
    Meio= (Inicio+Fim) / 2;
    if (Elemento==Vetor[Meio])
          return Meio;
    else    if (Inicio==Fim)
             return -1;      {Elemento nao encontrado}
            else If (Elemento < Vetor[Meio])
                  return BB(Vetor, Inicio, Meio-1, Elemento);
                 else return BB(Vetor, Meio+1, Fim, Elemento);
}
```



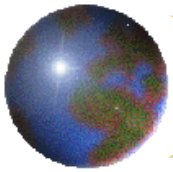
Recursividade Indireta

Além da recursividade direta estudada anteriormente, existe a *recursividade indireta*.

Uma função com tal propriedade não chama *diretamente* a si própria, mas sim *indiretamente*. Ou seja, por meio de outras funções (igualmente indiretamente recursivas).

Exemplo: f(x) = 2 * g(x - 1), f(0) = 1, f(1) = 2
g(x) = 2 * f(x - 1), g(1) = 1, g(0) = 2

f(4) = 2*g(3) = 2*2*f(2) = 2*2*2*g(1) = 2*2*2*1 = 8

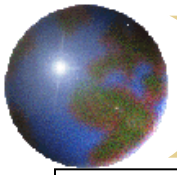


Recursividade Indireta

Exemplos 4-5: um número positivo é Par ou Ímpar

```
int Par(int Numero)
{
   if (Numero==0)                    {Caso Base}
    return 1;
    else  return Impar(Numero-1);   {Chama a Função Ímpar}
}
```

```
int  Impar(int Numero)
{
   if (Numero==0)                    {Caso Base}
      return 0;
    else return Par(Numero-1);   {Chama a Função Par}
}
```

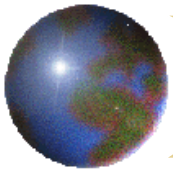


Recursividade Indireta

```
int F (int n)
{
   if (n==0)          {Caso base}
    return F= 1;
    else  return G(n-1);    {Chama a Função G}
}
```

```
int G (int n)
{
   if (n==0)          {Caso base}
    return 2;
    else  return H(n-1);    {Chama a Função H}
}
```

```
int H (int n)
{
   if (n==0)          {Caso base}
    return 1
    else return F(n-1);    {Chama a Função F}
}
```



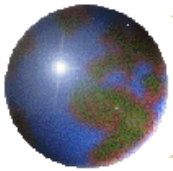
Recursividade

- Exercício: Série de Fibonacci
  - $Fib_0 = 1$
  - $Fib_1 = 1$
  - $Fib_n = Fib_{n-1} + Fib_{n-2}$, para n>1

Faça uma função recursiva para calcular o valor da série de Fibonacci para um dado número inteiro.

Por exemplo:

- 1º elemento da série: 1;
- 5º elemento da série: 8;
- 10º elemento da série: 89.

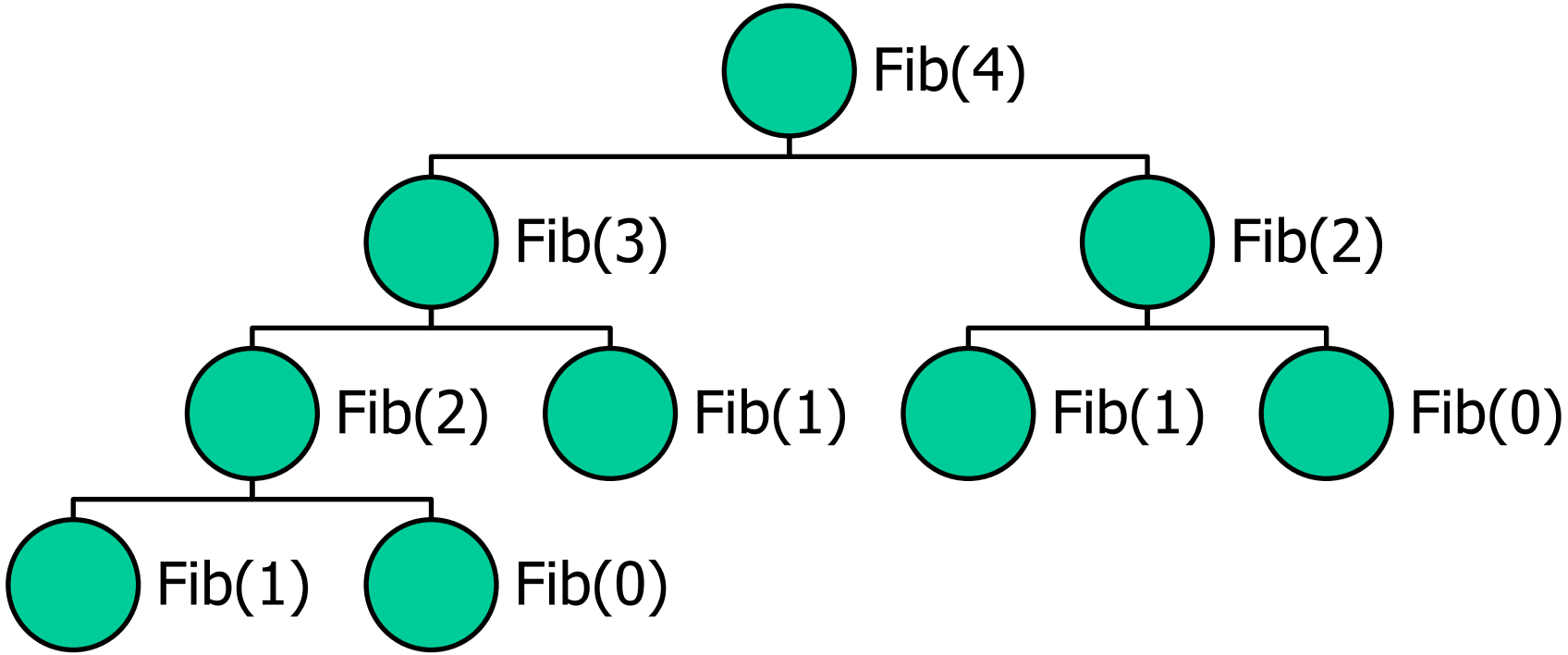


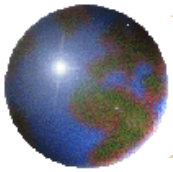
Recursividade

- Solução:

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

Função: int Fibonacci(int x)





Concluindo

- A recursividade é uma técnica elegante e quem a domina, geralmente, demonstra experiência, porém possui um preço: a movimentação de dados na PILHA (controle interno da linguagem para possibilitar ativações recursivas). Essa movimentação de dados de controle, impõe à solução um tempo adicional que pode torná-la ineficiente;
- Devido a esse tempo extra utilizado pelas soluções recursivas, deve-se dar preferência às soluções iterativas, deixando a utilização da recursividade para os casos apropriados (atentar para características básicas de um problema recursivo).

Bibliografia

PEREIRA, Silvio do Lago, "Estruturas de Dados Fundamentais: conceitos e aplicações", Editora Érica, 2002.

TENENBAUM, Aaron M., "Estrutura de dados Usando C", São Paulo, Makron Books, 1995.

WIRTH, N., "Algoritmos e estruturas de dados", Editora Prentice Hall, 1993.

ZIVIANI, Nivio, "Projeto de Algoritmos com implementação em Pascal e C", Editora Pioneira Thomson Learning, 2004. 2ª Edição.