Introdução à Computação II

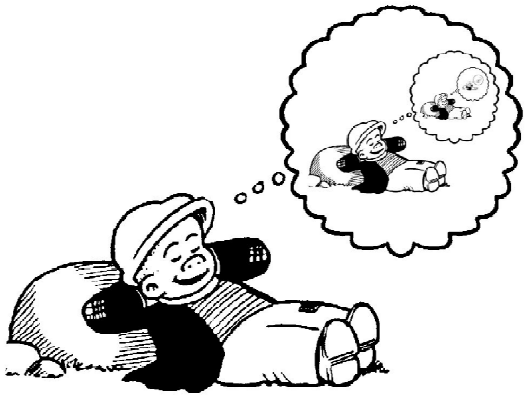
Recursividade

Profa.: *Adriane Beatriz de Souza Serapião*

adriane@rc.unesp.br

Recursividade

- **Um problema é dito recursivo se é definido em termos de si mesmo.**
- **Desta forma é possível definir um problema similar, porém de alguma forma mais simples.**



Recursividade

- Algoritmos recursivos são principalmente apropriados quando o problema a ser resolvido, a função a ser computada, ou os dados já estão definidos em termos recursivos ou por indução.
- Mas não significa que tal definição recursiva garanta a melhor solução do problema em termos de eficiência.
- A única ferramenta necessária para expressar operações recursivamente é o próprio procedimento ou a função, que tem a capacidade de invocar a si próprio.

Recursividade

- A recursão é uma forma interessante de resolver problemas, pois o divide em problemas menores de mesma natureza.
- Um processo recursivo consiste de duas partes:
  - O caso trivial, cuja solução é conhecida.
  - Um método geral que reduz o problema a um ou mais problemas menores de mesma natureza.

Função recursiva

- As funções e procedimentos podem ser recursivos, i.e., podem ativar-se a si próprios. Para que essa ativação não se repita indefinidamente, é necessário que haja uma estrutura condicional.

- Exemplo: Fatorial de N

N! = (N-1)! x N,	se N>0
N! = 1,	se N=1

Função recursiva

- Exemplo:

```
fatorial de um número
Se N = 1
   então
       Fatorial de N = 1
    senão
       Fatorial de N = N * fatorial de N-1
```

- Ou seja:

  Fatorial de 5 = 5 * fatorial de (4 * fatorial de
  (3 * fatorial de (2 * fatorial de (1 * fatorial(0))

Função recursiva

- **Função fatorial**

Solução Iterativa

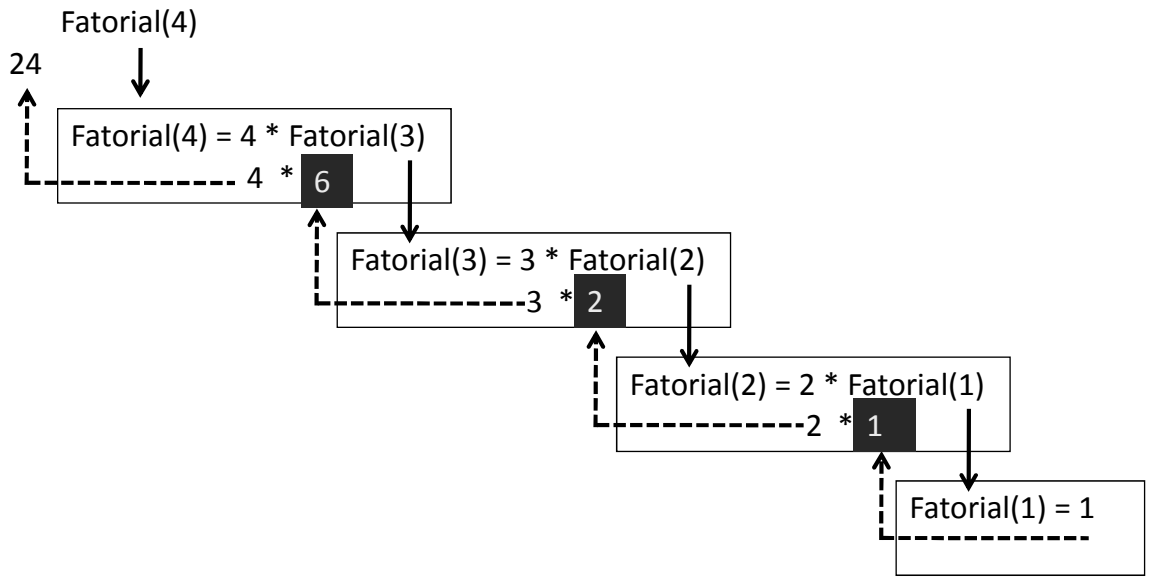
```
long int Fat (int N)
{
   long int result = 1;
   for (i=1; i <= N; i++)
      result *= i;
   return result;
};
```

Solução Recursiva

```
long int Fat (int N)
{
   if (N>1) return Fat (N-1) * N;
   else return 1;
};
```


Função recursiva

- **Função fatorial**



Função recursiva

- **Característica:**
  - **Uma condição de parada, isto é, algum evento que encerre a auto-chamada consecutiva. No caso do fatorial, isso ocorre quando a função é chamada com parâmetro (*n*) igual a 1. Um algoritmo recursivo precisa garantir que esta condição será alcançada.**
  - **Uma mudança de "estado" a cada chamada, isto é, alguma "diferença" entre uma chamada e a próxima. No caso do fatorial, o parâmetro *n* é decrementado a cada chamada.**

Implementação de recursividade em C

- **Do ponto de vista sintático, desde que o escopo de um identificador de subprograma estende-se do seu cabeçalho até o final do bloco em que é declarado, um subprograma pode ser chamado de dentro de outro subprograma ou até mesmo de dentro de si mesmo.**
- **Do ponto de vista de implementação, as considerações sobre chamadas de subprogramas e passagem de parâmetros valem, isto é, cada chamada a um subprograma recursivo gera uma nova alocação de parâmetros e variáveis locais e uma chamada independente da anterior.**
- **Múltiplas chamadas sobrecarregam a pilha de passagem de parâmetros e variáveis.**

Criação e destruição de variáveis locais

- A cada ativação de um procedimento ou função, os locais (variáveis, constantes, parâmetros por valor) aos mesmos são criados,

- e tão logo o procedimento ou função conclui (o end final é atingido) são destruídos.

Problema de implementação de um código recursivo

- Um código recursivo implica em execuções interrompidas que em um momento futuro devem ser retomadas e concluídas.

- Como garantir que uma execução interrompida possa ser retomada do ponto em que ocorreu a interrupção, com garantia de que ela seguirá sendo executada com os recursos locais idênticos àqueles do momento da interrupção?

Solução

- A cada interrupção de uma execução a ser retomada no futuro, salvar em uma pilha:
  - as variáveis locais e os parâmetros por valor; mais
  - o endereço de retorno.

- Quando uma execução interrompida tiver que ser retomada:
  - restaurar os valores das variáveis locais e dos parâmetros por valor, a partir da pilha e;
  - seguir a execução normalmente, a partir do ponto indicado pelo endereço de retorno.

Recursividade

- Detalhamento da execução de uma função FATORIAL, com implementação recursiva.

  - Atenção: os endereços de retorno não estão indicados na pilha, embora sejam também nela armazenados.

Chamada interrompida - Empilhamento

```
long int Fatorial(long int Valor)
{
if (Valor == 1)
    return 1;
  else
    return Valor * Fatorial (Valor - 1);
}
```

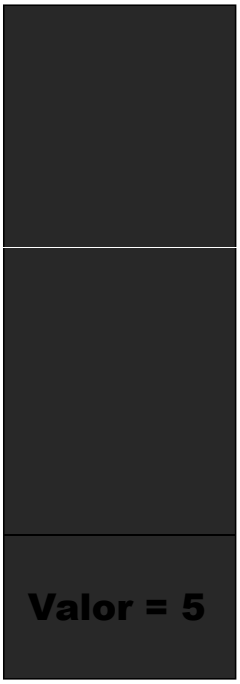
Primeira Chamada: Valor = 5

Executado o Else do if, que dispara nova execução da função, com Valor - 1 de parâmetro, ou seja, 4.

Atribuição de Fatorial inconclusa!

Atenção: o end; da função não foi atingido, logo a execução com Valor = 5 não concluiu!

Situação da pilha, ao ser interrompida a primeira execução de Fatorial



Chamada interrompida - Empilhamento

```
long int Fatorial(long int Valor)
{
if (Valor == 1)
    return 1;
  else
    return Valor * Fatorial (Valor - 1);
}
```

Segunda Chamada: Valor = 4

Executado o Else do if, que dispara nova execução da função, com Valor - 1 de parâmetro, ou seja, 3.

Atribuição de Fatorial inconclusa!

Atenção: o end; da função não foi atingido, logo a execução com Valor = 4 não concluiu!

Situação da pilha, ao ser interrompida a segunda execução de Fatorial

Chamada interrompida - Empilhamento

```
long int Fatorial(long int Valor)
{
if (Valor == 1)
    return 1;
  else
    return Valor * Fatorial (Valor - 1);
}
```

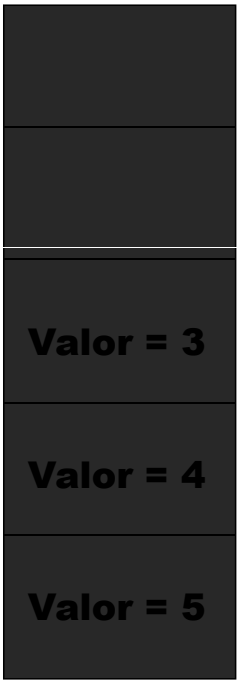
Terceira Chamada: Valor = 3

Executado o Else do if, que dispara nova execução da função, com Valor - 1 de parâmetro, ou seja, 2.

Atribuição de Fatorial inconclusa!

Atenção: o end; da função não foi atingido, logo a execução com Valor = 3 não concluiu!

Situação da pilha, ao ser interrompida a terceira execução de Fatorial



Chamada interrompida - Empilhamento

```
long int Fatorial(long int Valor)
{
if (Valor == 1)
    return 1;
  else
    return Valor * Fatorial (Valor - 1);
}
```

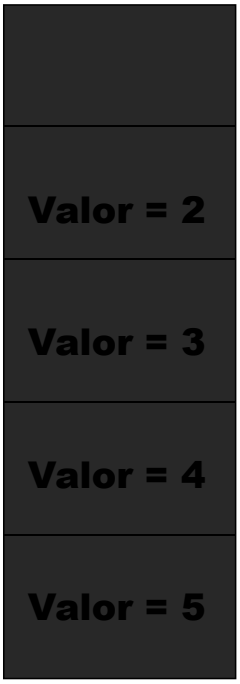
Quarta Chamada: Valor = 2

Executado o Else do if, que dispara nova execução da função, com Valor - 1 de parâmetro, ou seja, 1.

Atribuição de Fatorial inconclusa!

Atenção: o end; da função não foi atingido, logo a execução com Valor = 2 não concluiu!

Situação da pilha, ao ser interrompida a quarta execução de Fatorial



Chamada interrompida - Empilhamento

```
long int Fatorial(long int Valor)
{
if (Valor == 1)
    return 1;
  else
    return Valor * Fatorial (Valor - 1);
}
```

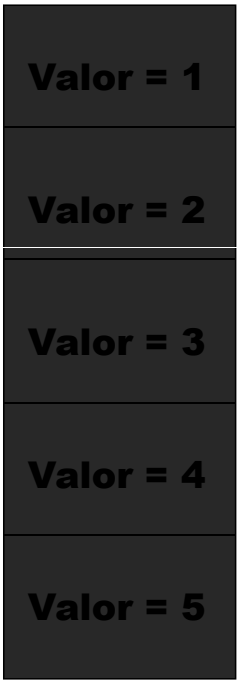
Quinta Chamada: Valor = 1

Executado o Else do if, que dispara nova execução da função, com Valor - 1 de parâmetro, ou seja, 0.

Atribuição de Fatorial inconclusa!

Atenção: o end; da função não foi atingido, logo a execução com Valor = 1 não concluiu!

Situação da pilha, ao ser interrompida a quinta execução de Fatorial



Chamada concluída

```
long int Fatorial(long int Valor)
{
if (Valor == 1)
    return 1;
  else
    return Valor * Fatorial (Valor - 1);
}
```

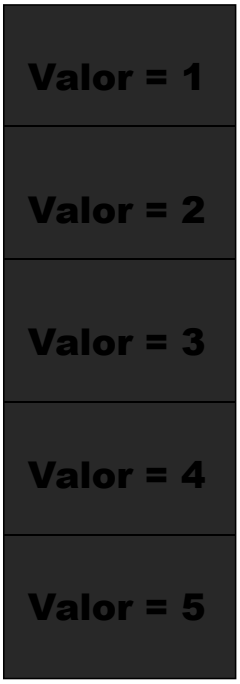
Sexta Chamada: Valor = 0

Executado o Then do if.

Fatorial recebe 1.

Esta chamada de Fatorial conclui ao ser executado o end; da função!

Situação da pilha, ao concluir a sexta execução de Fatorial



Conclusão de chamada de interrompida - Desempilhamento

```
long int Fatorial(long int Valor)
{
if (Valor == 1)
    return 1;
  else
    return Valor * Fatorial (Valor - 1);
}
```

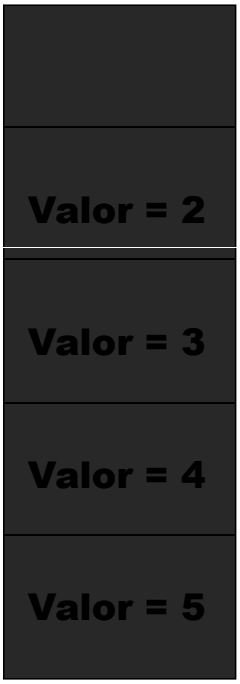
Retomada da Quinta Chamada: Valor recebe novamente o valor 1.

Atribuição de Fatorial no Else finalmente conclui.

Fatorial := 1 * 1 (valor retornado pela chamada anterior)

Esta chamada de Fatorial conclui ao ser executado o end; da função.

Situação da pilha, ao concluir a quinta execução de Fatorial



Conclusão de chamada de interrompida - Desempilhamento

```
long int Fatorial(long int Valor)
{
if (Valor == 1)
    return 1;
  else
    return Valor * Fatorial (Valor - 1);
}
```

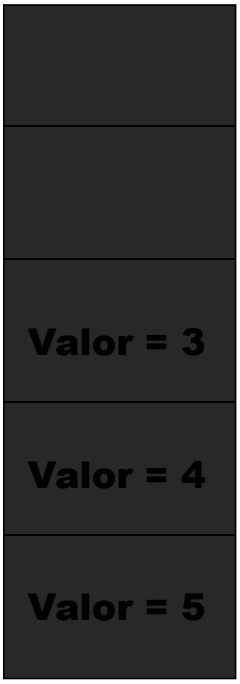
Retomada da Quarta Chamada: Valor recebe novamente o valor 2.

Atribuição de Fatorial no Else finalmente conclui.

Fatorial := 2 * 1 (valor retornado pela chamada anterior).

Esta chamada de Fatorial conclui ao ser executado o end; da função!

Situação da pilha, ao concluir a quarta execução de Fatorial



Conclusão de chamada de interrompida - Desempilhamento

```
long int Fatorial(long int Valor)
{
if (Valor == 1)
    return 1;
  else
    return Valor * Fatorial (Valor - 1);
}
```

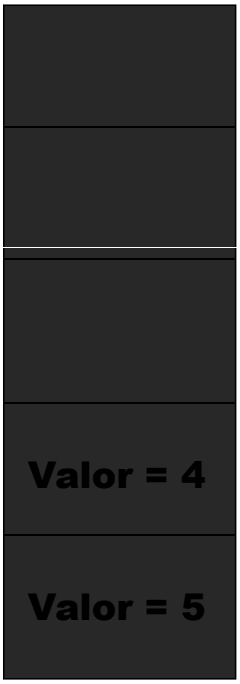
Retomada da Terceira Chamada: Valor recebe novamente o valor 3.

Atribuição de Fatorial no Else finalmente conclui.

Fatorial := 3 * 2 (valor retornado pela chamada anterior).

Esta chamada de Fatorial conclui ao ser executado o end; da função!

Situação da pilha, ao concluir a terceira execução de Fatorial



Conclusão de chamada de interrompida - Desempilhamento

```
long int Fatorial(long int Valor)
{
if (Valor == 1)
    return 1;
  else
    return Valor * Fatorial (Valor - 1);
}
```

Retomada da Segunda Chamada: Valor recebe novamente o valor 4.

Atribuição de Fatorial no Else finalmente conclui.

Fatorial := 4 * 6 (valor retornado pela chamada anterior).

Esta chamada de Fatorial conclui ao ser executado o end; da função!

Situação da pilha, ao concluir a segunda execução de Fatorial

Conclusão de chamada de interrompida - Desempilhamento

```
long int Fatorial(long int Valor)
{
if (Valor == 1)
    return 1;
  else
    return Valor * Fatorial (Valor - 1);
}
```

Retomada da Primeira Chamada: Valor recebe novamente o valor 5.

Atribuição de Fatorial no Else finalmente conclui.

Fatorial := 5 * 24 (valor retornado pela chamada anterior).

Esta chamada de Fatorial conclui ao ser executado o end; da função!

Situação da pilha, ao concluir a primeira execução de Fatorial

Execução de um subprograma recursivo

- Início da execução:
  - criação de locais:
    - variáveis, constantes, etc. locais e
    - parâmetros por valor.

Execução de um subprograma recursivo

- **Processamento:**
  - **Tentativa de execução do código até o *end* final.**
  - **Se a execução for interrompida por nova chamada recursiva, empilhamento dos locais (variáveis, constantes, etc. locais e parâmetros por valor) e do endereço de retorno.**

Execução de um subprograma recursivo

- **Retomada de uma execução interrompida:**
  - **Retomada da execução a partir do ponto onde havia sido interrompida, usando o endereço de retorno.**
  - **Desempilhamento dos locais com restauração de seu conteúdo conforme encontravam-se ao dar-se a interrupção da execução.**

Execução de um subprograma recursivo

- **Fim de uma execução:**
  - **destruição dos locais e retorno ao ponto de chamada do trecho recursivo (procedimento ou função).**

Atenção

- **Parâmetros por referência NÃO sofrem empilhamento!!!**
- **Eles NÃO SÃO LOCAIS aos procedimentos ou funções!!!**

Função recursiva

- **Escreva uma função recursiva que calcule A^B, onde A é um número real e B é um inteiro não negativo. Essa função pode ser definida da seguinte forma:**

$$A^B = \begin{cases} 1 & , \text{se } B = 0 \\ A * A^{B-1} & , \text{se } B > 0 \end{cases}$$

Função recursiva

- **Função potência**

```
// programa CalculaAelevadoB;
#include <stdio.h>
float Potencia (float base, int expoente);

void main()
{
   float B; int E;
   // Programa Principal
   printf("Entre com a base e o expoente (A elevado a B): ");
   scanf("%f %d",&B, &E);
   printf("\n A elevado a B vale: %f\n", Potencia(B,E));
}

// Funcao Potencia
float Potencia (float base, int expoente)
{
   if (expoente==0) return 1;
   else
      return base * Potencia(base, expoente-1);
}
```

Função recursiva

- **Função potência – considerando agora a possibilidade de B ser um inteiro negativo:**

```
// programa CalculaAelevadoB;

// Funcao Potencia
float Potencia (float base, int expoente)
{
   if (expoente == 0) return 1;
   else if (expoente < 0)
      return 1/Potencia(base, abs(expoente));
   else
      return base*Potencia(base, expoente-1);
}
```


Função recursiva

- **Função potência melhorada**
  - **O algoritmo da função anterior pode ser melhorado, obtendo-se significativa melhoria no seu tempo de execução (o número de ativações recursivas decresce de N para log2(N):**

```
// programa CalculaAelevadoB;

// Funcao Potencia
float Potencia ( float base, int expoente )
{
   if (expoente == 0) return 1;
   else if (expoente < 0)
         return 1/Potencia(base, abs(expoente));
   else if (expoente%2 != 0)
         return base*sqrt(Potencia(base,(expoente-1)/2));
   else
         return sqrt(Potencia(base, expoente/2));
}
```

Recursividade – Torre de Hanói

- São dados *n* discos de diâmetro 1, 2, 3, *n* dispostos por ordem decrescente no primeiro poste (A).

- Pretende-se transferir todos os discos para o terceiro poste (C), utilizando o menor número de movimentos, de tal modo que as seguintes restrições sejam satisfeitas:
  1. Apenas um disco pode ser movido de cada vez.
  2. Apenas os discos do topo podem ser movidos.
  3. Um disco não pode ser colocado sobre outro menor.

Recursividade – Torre de Hanói

- A Origem:
  - Edouard Lucas – 1883.
  - Matemático francês.

- A lenda:
  - Num templo hindu existe uma versão do jogo com 64 peças douradas.
  - Os monges lá estão movendo as peças.
  - Quando o jogo terminar o mundo termina.
  - À taxa de um movimento por segundo, serão necessários 585 bilhões de anos.

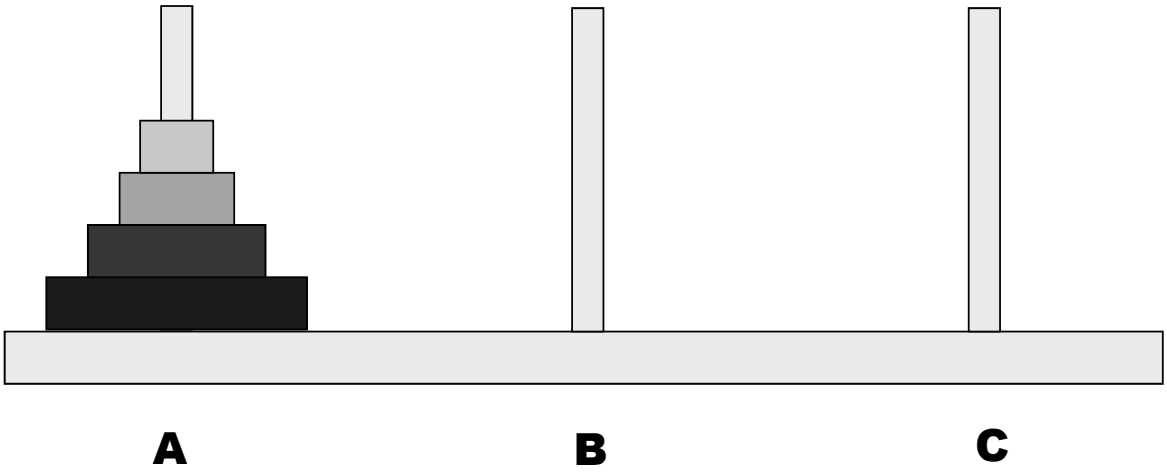
Recursividade – Torre de Hanói

- A solução deste problema advém da redução do problema original em problemas menores. Devem ser feitas reduções sucessivas do problema até que se chegue ao problema trivial de mover um disco de uma haste para outra.

- Explicando: em nosso exemplo temos 4 discos, então vamos reduzi-lo ao problema imediatamente menor que é o de mover 3 discos e assim sucessivamente até que se tenha o problema de mover apenas um disco.

Recursividade – Torre de Hanói

Solução



Recursividade – Torre de Hanói

- **Seguindo esse raciocínio, pode-se estabelecer uma estratégia para resolver o problema de mover um número N qualquer de discos da haste A para a haste C:**

Se N = 1 então
mover o disco 1 da haste A para C
Senão
1. Movemos N-1 discos da haste A para a haste B, usando a haste C para armazenamento temporário;
2. Em seguida, movemos o disco N da haste A para a haste C;
3. Por fim, movemos os N-1 discos da haste B para a haste C, usando a haste A para armazenamento temporário.

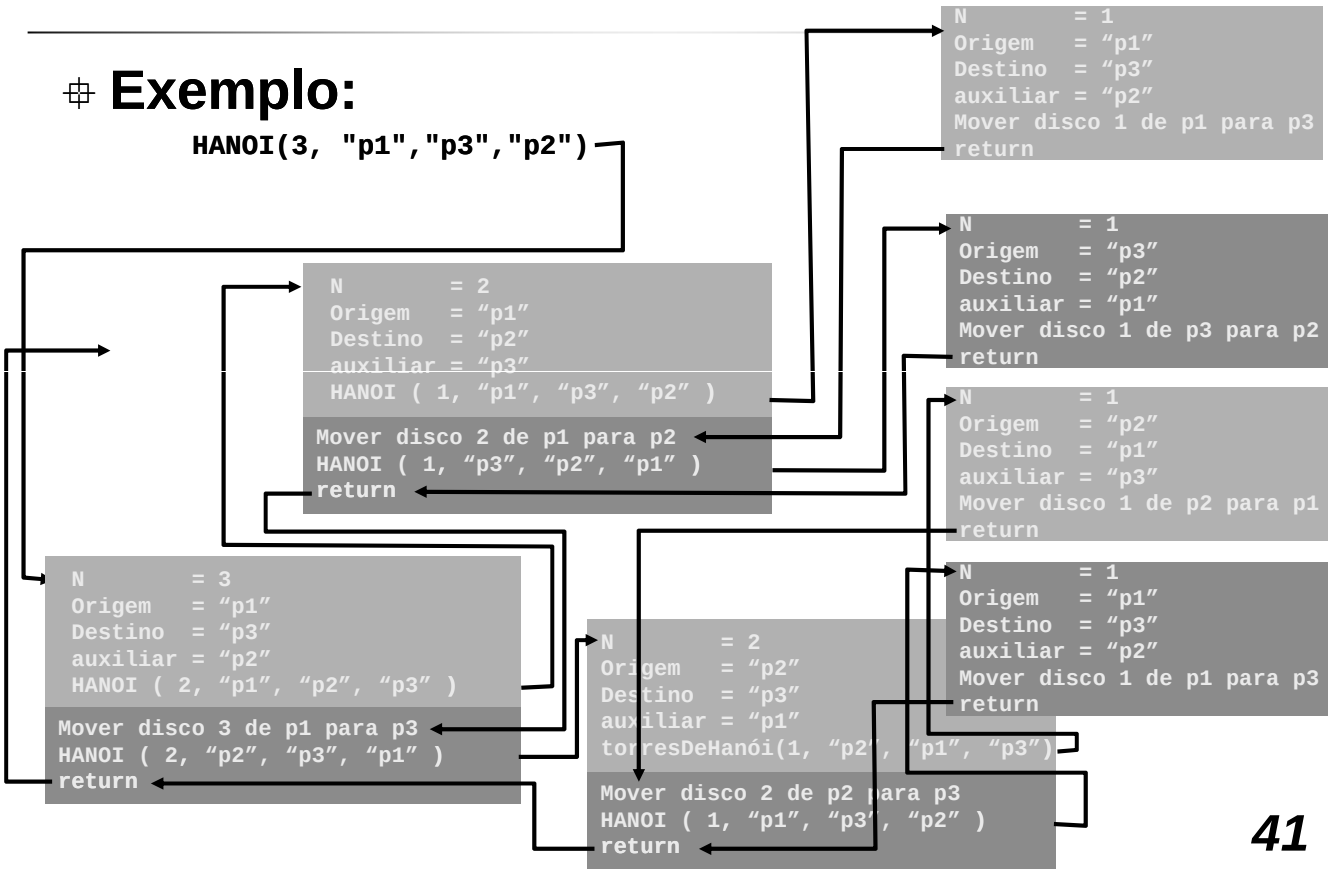
Recursividade – Torre de Hanói

- **O procedimento recursivo para implementar esta estratégia ficaria, na linguagem C, da seguinte forma:**

```
void HANOI (int N, char ORIGEM, char DESTINO, char TEMPORARIO);
{
   if (N == 1)
        printf(“Mover disco 1 da haste %c para haste %c\n” ,
                ORIGEM, DESTINO;
   else
   {
        HANOI(N-1,ORIGEM,TEMPORARIO,DESTINO);
        printf(“Mover disco %d da haste %c p/ haste %c\n“, N,
                ORIGEM, DESTINO);
        HANOI(N-1,TEMPORARIO,DESTINO,ORIGEM);
   };
}
```

Recursividade – Torre de Hanói

- **Exemplo:**



Recursão de cauda

```
1 Subprograma impvet(v,i,n)
2
3 e: v:vetor
4 i:inteiro {posição atual dentro do vetor}
5 n:inteiro {número de elementos do vetor}
6
7 {subprograma recursivo p/ impressão do
8 vetor – imprime elementos de i a n}
9
10 início
11    Se i <= n então
12        escreva(vet[i])
13        impvet(v,i+1,n)
14    fim se
15 fim
```

```
1 Subprograma impvet(v,i,n)
2
3 e: v:vetor
4 i:inteiro {posição atual dentro do vetor}
5 n:inteiro {número de elementos do vetor}
6
7 {subprograma recursivo p/ impressão do
8 vetor – imprime elementos de i a n}
9
10 início
11    Se i <= n então
12        impvet(v,i+1,n)
13        escreva(vet[i])
14    fim se
15 fim
```

Qual a diferença entre os dois subprogramas?

Recursão de cauda

- **Os subprogramas acima têm definições que seriam idênticas não fosse pelas linhas 12 e 13, que estão invertidas.**
- **No primeiro algoritmo, o i-ésimo valor é escrito e então o subprograma é chamado para os elementos subseqüentes do vetor.**
- **No segundo algoritmo a chamada ocorre antes da impressão. Como conseqüência, o primeiro algoritmo tem como resultado a impressão dos elementos do vetor na ordem em que se encontram armazenados no vetor, e o segundo imprime os valores na ordem inversa, isto é, o primeiro valor a ser impresso é v[n].**
- **A recursão do primeiro algoritmo é chamada de Recursão de Cauda.**

Recursão indireta

- **Como verificar se um número inteiro (*n*) é par ou ímpar usando recursividade?**
- **Quais são os casos triviais?**
  - **Se *n* = 0 então *n* é par**
  - **Se *n* = 1 então *n* é ímpar**

Recursão indireta

Subprograma par(x):lógico

```
e:x:inteiro
r:verdadeiro se x é par, falso caso
    contrário
início
  Se x=0 então
    retorne(verdadeiro)
  senão
    Se x=1 então
      retorne(falso)
    senão
      retorne (impar(x-1))
    fim se
  fim se
fim
```

Subprograma impar(x):lógico

```
e:x:inteiro
r:verdadeiro se x é ímpar, falso
    caso contrário
início
  Se x=0 então
    retorne(falso)
  senão
    Se x=1 então
      retorne(verdadeiro)
    senão
      retorne (par(x-1))
    fim se
  fim se
fim
```

Dados dois subprogramas a e b, uma recursão indireta é quando a chama b que chama a.

Recursão indireta

```
//programa ProgramaParesImpares;
bool impar(int x);
bool par(int x);
void main()
{
  int n;
  printf("Entre com um inteiro positivo: ");
  scanf("%d", &n);
  if (par(n)) printf("\n Número %d é par", n);
  else printf("\n Número %d é ímpar", n;
}

bool par(int x)
{
  if (x == 0) return true;
  else if (x == 1) return false;
  else return impar(x-1);
}
bool impar(int x)
{
  if (x == 0) return false;
  else if (x == 1) return true;
  else return par(x-1);
}
```

Exemplo de recursão indireta: jogo resta um

```
// Este programa simula o jogo de nim com
// dois jogadores (um deles é o computador).
// O jogo envolve duas jogadas alternadas de
// jogadores. O jogo começa com uma pilha de
// InitialCoins (usualmente 13) moedas na mesa.
// Cada jogada consiste em retirar entre 1
// e MaxTake (usualmente 3) moedas da pilha.
// O jogador que retirar a última moeda perde.
// Neste programa, o jogador humano joga contra
// o computador; o humano é sempre o primeiro
// a jogar.

include <string.h>

typedef enum {FALSE, TRUE} bool;
typedef enum {Human, Computer} playerT;

// Protótipo de funções
// --------------------

#include <stdio.h>
#include <string.h>

#define InitialCoins 13
#define MaxTake 3
#define NoGoodMove -1

int FindGoodMove( int);
bool IsBadPosition( int);
int GetUserMove( int);
int ChooseComputerMove( int);
```

```
int main( void)
{
   int nCoins, nTaken;
   playerT whoseTurn;

   nCoins = InitialCoins;
   whoseTurn = Human;
   while (nCoins > 1) {
      printf("Há %d moedas na pilha.\n“, nCoins);
      switch (whoseTurn) {
        case Human:
           nTaken = GetUserMove( nCoins);
           whoseTurn = Computer;
           break;
        case Computer:
           nTaken = ChooseComputerMove( nCoins);
           cout << "Eu pego " << nTaken <<
              “moedas.\n",< (nTaken == 1) ? "" :
              "s");
           whoseTurn = Human;
           break;
      }
      nCoins -= nTaken;
   }
   if (nCoins == 0) {
      printf("Você pegou a última moeda e
     perdeu.\n“);
   } else {
      printf("Sobrou uma só moeda.\n“);
      switch (whoseTurn) {
         case Human: printf("Eu ganhei.\n“);
              break;
         case Computer: printf("Eu perdi.\n“);
              break;
      }
   }
   return 0;
}
```


Exemplo de recursão indireta: jogo resta um

```
// A função abaixo recebe um número nCoins >= 2
// de moedas e devolve uma jogada (isto é,
// número de moedas a retirar) que seja boa.
// Uma jogada é boa se deixa o adversário em
// uma posição ruim. Se tal jogada não existir,
// a função devolve a constante NoGoodMove.

int FindGoodMove (int nCoins)
{
   int nTaken;

   for (nTaken = 1; nTaken <= MaxTake; nTaken++)
       if  (IsBadPosition(  nCoins  -  nTaken))
     return nTaken;
   return NoGoodMove;
}

// Esta função recebe um número nCoins >= 1
// e devolve TRUE se nCoins é uma posição ruim.
// Uma posição é ruim se não existe uma boa
// jogada a partir dela. Se nCoins é 1 então
// a posição é obviamente ruim.

bool IsBadPosition( int nCoins)
{
   if (nCoins == 1) return TRUE;
   return FindGoodMove( nCoins) == NoGoodMove;
}
```

```
/ A função abaixo recebe o número nCoins
// de moedas na pilha e devolve o número de
// moedas que o jogador humano decidiu retirar.

int GetUserMove( int nCoins)
{
   int nTaken, limit;

   while (TRUE) {
      printf("\nQuantas moedas você quer? “);
      scanf(“%d”, &nTaken);
      if (nTaken > 0 && nTaken <= MaxTake
                    && nTaken <= nCoins) break;
      limit  =  (nCoins  <  MaxTake)  ?  nCoins  :
     MaxTake;
      printf("Escolha número entre 1 e %d\n”,
     limit);
      printf("Há  %d  moedas  na  pilha.\n",
     nCoins);
   }
   return nTaken;
}


// Esta função recebe o número nCoins de
// moedas na pilha e devolve uma boa jogada.
// Se não houver uma boa jogada boa,
// a função devolve 1 (para que o adversário
// tenha mais oportunidades de cometer um erro).

int ChooseComputerMove( int nCoins)
{
   int nTaken;

   nTaken = FindGoodMove( nCoins);
   if (nTaken == NoGoodMove) nTaken = 1;
   return nTaken;
}
```

Esquemas não apropriados à recursão

- Quando existe uma única chamada do procedimento recursivo no fim ou no começo da rotina, o procedimento é facilmente transformado numa iteração simples. É boa política não usar recursão quando existe um algoritmo iterativo igualmente claro que resolva o problema. É o caso do fatorial e da soma de vetores.

- Quando o uso de recursão acarreta num número maior de cálculos.

Desenvolvimento de um código usando recursividade

CUIDADO BÁSICO

- Não esquecer que deve haver uma condição de interrupção das sucessivas chamadas recursivas que, uma vez ocorrendo, conclua finalmente uma execução e permita a conclusão de todas as outras execuções pendentes.

Prós e contras da recursividade

PRÓS:

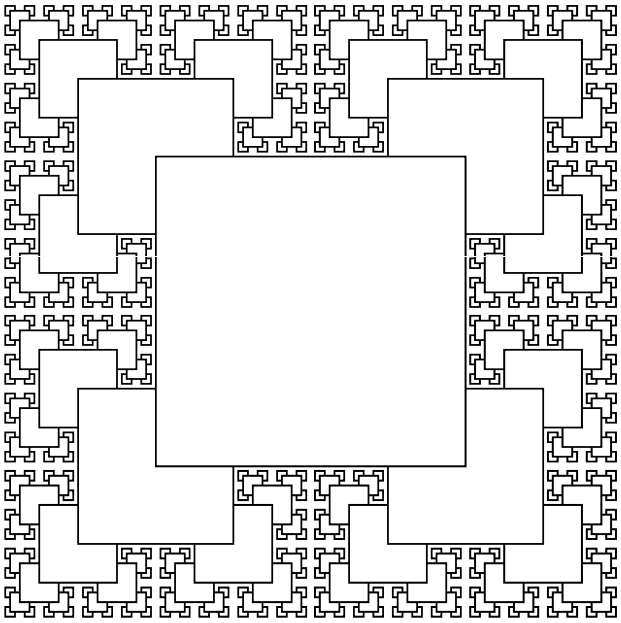
- **Código mais compacto.**
- **Especialmente conveniente para estruturas de dados definidas recursivamente, tais como árvores.**
- **Eventualmente código pode ser mais fácil de entender.**

Prós e contras da recursividade

CONTRAS:

- **Em geral não oferece economia de memória. Os valores locais sendo processados têm que ser empilhados, o quê consome espaço.**
- **Uma solução recursiva nem sempre será mais rápida que uma correspondente iterativa, por exemplo.**

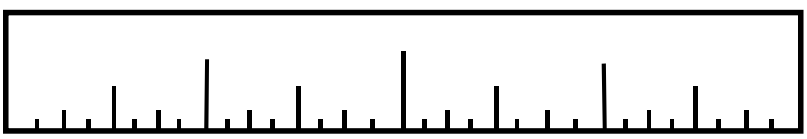
Outros exemplos de recursividade



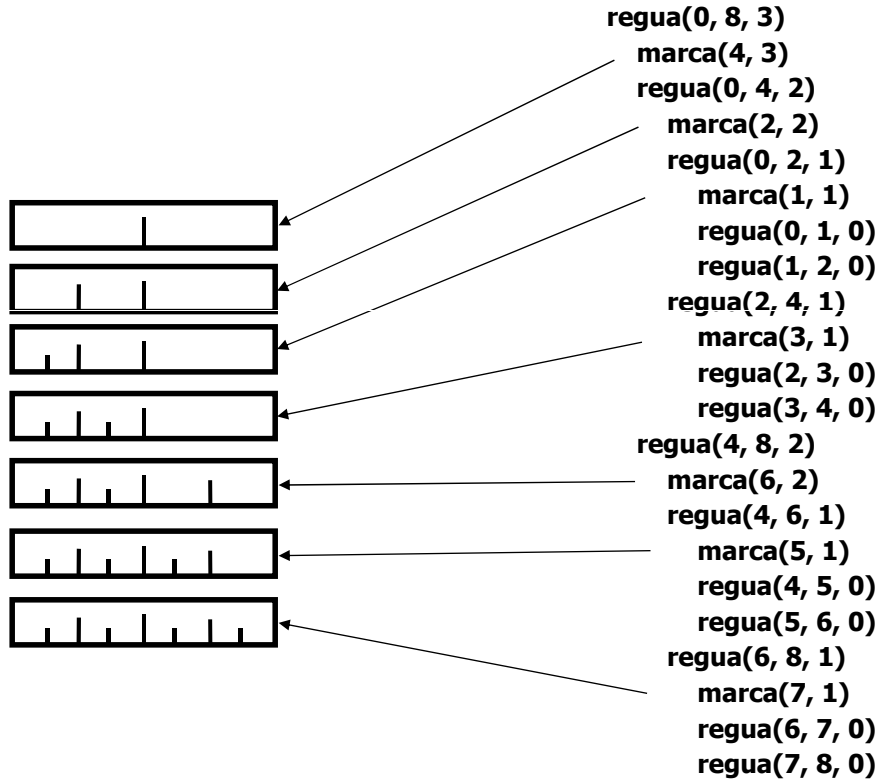
```
void estrela(int x,int y, int r)
{
  if ( r > 0 )
  {
    estrela(x-r, y+r, r div 2);
    estrela(x+r, y+r, r div 2);
    estrela(x-r, y-r, r div 2);
    estrela(x+r, y-r, r div 2);
    box(x, y, r);
  }
}
```

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);
  }
}
```



Execução: régua



Usar recursão quando

- O problema é naturalmente recursivo (clareza) e a versão recursiva do algoritmo não gera ineficiência evidente se comparado com a versão iterativa do mesmo algoritmo.
- O algoritmo se torna compacto sem perda de clareza ou generalidade.
- É possível prever que o número de chamadas ou a carga sobre a pilha de passagem de parâmetros não irá causar interrupção do processo.

Não usar recursão quando

- A solução recursiva causa ineficiência se comparada com uma versão iterativa.
- A recursão é de cauda.
- Parâmetros consideravelmente grandes têm que ser passados por valor.
- Não é possível prever se o número de chamadas recursivas irá causar sobrecarga da pilha de passagem de parâmetros.

Revisão: Pontos importantes da recursividade

1. Embora a recursão seja um recurso muito poderoso, ela é cara no sentido de que ocupa muita memória para ser executada.
2. Todo módulo recursivo, para ser recursivo, deve se auto-invocar.
3. Toda recursão deve apresentar, obrigatoriamente, um caso-base, que é uma condição em que a função não se auto-invoca novamente.
4. Toda recursão atua em pelo menos uma das características do problema sendo por ela resolvido.
5. Quando um módulo qualquer se invoca (fazendo, portanto, uma recursão), as duas instâncias desse módulo possuem variáveis distintas, tal como se fossem módulos diferentes (na verdade, são instâncias diferentes do mesmo módulo).

Exercícios

- 1) Verifique e explique o que acontece com os códigos abaixo:

```
void chama1(int n)
{
  if (n==10)
    printf(“Atingi o máximo.
     \n”);
  else
  {
    chama1(n+1);
        //Chamada recursiva
    printf(“%d ...\n”, n);
        //Comandos      após
   chamada       recursiva
  };
}
```

```
void chama2(int n)
{
  if (n==10)
    printf(“Atingi o máximo.
     \n”);
  else
  {
    printf(“%d ...\n”, n);
       //Comandos antes da
        chamada recursiva
    chama2(n+1);
       //Chamada recursiva
  };
}
```


Exercícios

- 1) Continuando o exercício anterior, faça ainda o mesmo com os códigos abaixo:

```
void chama3(int n)
{
  if (n==10)
    printf( “ Executando  chama3(10)  -
    Atingi o máximo. \n”);
  else
  {
    //Comandos    antes    da    chamada
    recursiva
    printf(“Executando
            chama3(%d)...\n”, n);
    printf(“%d“...”, n);
    //Chamada recursiva
    chama3(n+1);
    //Comandos após chamada recursiva
    printf(“Voltando para
            chama3(%d “)...\n”,n);
  };
}
```

```
void chama4(int n)
{
  if (n==10)
   printf( “ Executando  chama4(10)  -
    Atingi o máximo. \n”);
  else
  {
    //Comandos    antes    da    chamada
    recursiva
    printf(“Executando
            chama4(%d)...\n”, n);
    //Chamadas recursivas
    chama4(n+1);
    //Comandos    após    1a.    chamada
    recursiva
    printf(“Voltando para
            chama4(%d “)...\n”,n);
    chama4(n+1);
    //Comandos    após    2a.    chamada
    recursiva
    printf(“Voltando novamente para
            chama4(%d “)...\n”,n);
  };
}
```

Exercícios

- **2) Faça uma função recursiva para resolver a seguinte função matemática:**

$$P_n(x) = \begin{cases} 1 & , \text{se } n = 0 \\ x & , \text{se } n = 1 \\ \dfrac{(2n-1) * x * P_{n-1}(x) - (n-1) * P_{n-2}(x)}{n} & , \text{se } n > 1 \end{cases}$$

- **3) Um algoritmo muito conhecido para determinar o maior divisor comum de dois inteiros é o algoritmo de Euclides. A função maior divisor comum (MDC) é definida como segue. Crie uma função recursiva para implementar este algoritmo.**

$$MDC(X,Y) = \begin{cases} MDC(Y,X) & , \text{se } Y > X \\ X & , \text{se } Y = 0 \\ MDC(Y, X \bmod Y) & , \text{se } Y > 0 \end{cases}$$

Exercício 2) – Solução $P_n(x)$

```
float p(int n, float x)
{
  IF (n == 0)
     return 1;
  ELSE
    IF (n == 1)
       return x;
    ELSE
       return        ((2*n-1)*x*p(n-1,x)-(n-1)*p(n-
   2,x))/n;
}
```

Exercício 3) – Solução *mdc(x,y)*

```
#include <stdio.h>

int mdc(int x,int y)
{
 if(y == 0)
  return x;
 elseif (y > x)
        return mdc(y, x)
 else return mdc(y,x%y);
}

int main()
{
 printf("%d",mdc(25,20));
}
```