



CENTRO DE CIÊNCIA E TECNOLOGIA
LABORATÓRIO DE CIÊNCIAS MATEMÁTICAS
UNIVERSIDADE ESTADUAL DO NORTE FLUMINENSE

Recursão

Disciplina: Estruturas de Dados I

Prof. Fermín Alfredo Tang Montané

Curso: Ciência da Computação

Algoritmos Recursivos vs Iterativos

- Interessa compreender as diferenças entre algoritmos recursivos e iterativos.
- Começaremos ilustrando com ambas versões para o problema de cálculo do fatorial de um número.

Fatorial

Definição Iterativa

- Considere o caso do cálculo do fatorial. O cálculo do fatorial de um número positivo n é o produto dos valores inteiros de 1 a n .
- Esta definição é mostrada na figura.

Factorial (n) =
$$\begin{cases} 1 & \text{if } n = 0 \\ n \times (n - 1) \times (n - 2) \times \dots \times 3 \times 2 \times 1 & \text{if } n > 0 \end{cases}$$

Inicialização
Iteração

Iterative Factorial Algorithm Definition

- Note que esta definição é **iterativa**. Um algoritmo é definido de maneira iterativa quando a sua definição somente envolve parâmetros do algoritmo. Neste caso a definição do fatorial depende somente do parâmetro n . Calculamos o Fatorial(4) :

$$\text{factorial}(4) = 4 \times 3 \times 2 \times 1 = 24$$

Fatorial

Definição Recursiva

- Um algoritmo usa **recursão** quando o próprio algoritmo aparece dentro da sua definição.
- Considere o caso do cálculo do fatorial. O algoritmo para o cálculo do fatorial pode ser definido de maneira recursiva como mostrado a seguir:

$$\text{Factorial } (n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (\text{Factorial } (n - 1)) & \text{if } n > 0 \end{cases}$$

Recursive Factorial Algorithm Definition

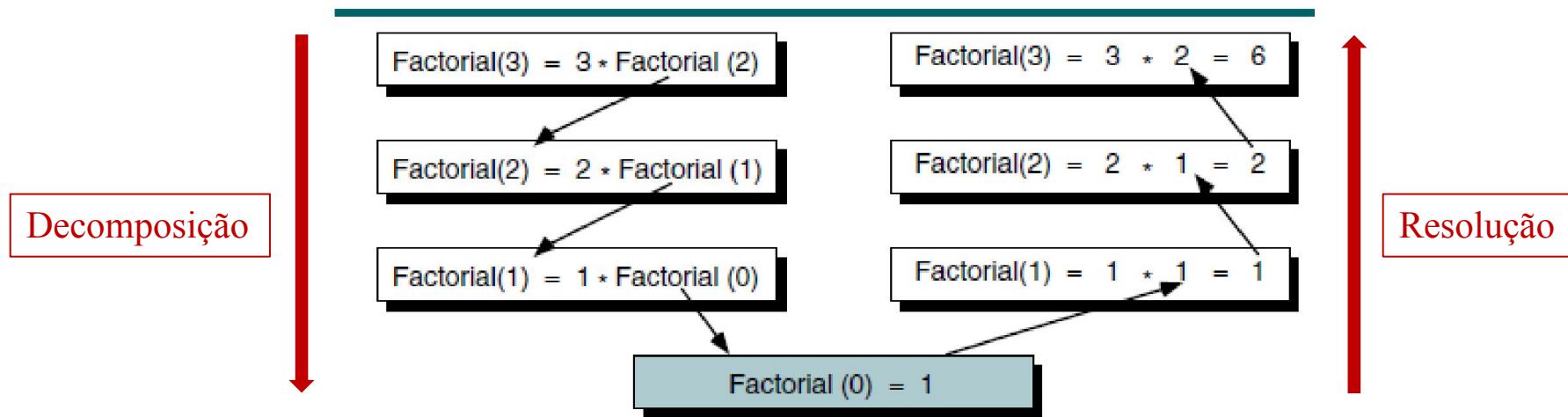
if $n = 0$ Caso Base
if $n > 0$ Relação de Recorrência

- Todo algoritmo recursivo possui duas partes:
 - **Caso Base:** É a condição de parada para as chamadas recursivas.
 - **Caso Geral:** É a relação de recorrência, define de que maneira o problema é subdividido em instâncias menores dele próprio.

Fatorial

Definição Recursiva

- A característica fundamental de algoritmo recursivo é a decomposição de um problema em um ou mais subproblemas de tamanho menor.
- A figura ilustra a decomposição do Fatorial(3).



- Vale observar, que a solução recursiva de um problema de fato envolve um percurso em **duas direções**:
 - A decomposição do problema, de cima para baixo;
 - A resolução dos subproblemas, e o retorno de baixo para cima.

Fatorial

Definição Recursiva

- Embora a solução recursiva possa parecer mais longa e mais difícil de acompanhar, ela com frequência costuma ser uma solução **mais simples de formular** e até mais elegante por causa disso.

Recursividade é um processo repetitivo em que um algoritmo chama a se próprio.

Cada chamada recursiva deve resolver parte do problema ou reduzir o tamanho do problema.

Fatorial

Algoritmo Iterativo

- Considere o seguinte **algoritmo iterativo** para resolver o problema do calculo do fatorial de um número. Este tipo de solução envolve o uso de um ou mais loops.

Iterative Factorial Algorithm

```
Algorithm iterativeFactorial (n)
Calculates the factorial of a number using a loop.
Pre n  is the number to be raised factorially
Post n! is returned
1 set i to 1
2 set factN to 1
3 loop (i <= n)
    1 set factN to factN * i
    2 increment i
4 end loop
5 return factN
end iterativeFactorial
```

Inicialização

Iteração

Fatorial

Algoritmo Recursivo

- Considere agora o seguinte algoritmo recursivo para resolver o problema de calculo do fatorial de um número. Neste tipo de solução não existem loops, a própria recursão é em si uma repetição.
- O algoritmo factorial se chama a si próprio, cada vez com um conjunto diferente de parâmetros.

Recursive Factorial

```
Algorithm recursiveFactorial (n)
Calculates factorial of a number using recursion.
Pre    n  is the number being raised factorially
Post   n! is returned
1  if (n equals 0)
1    return 1
2  else
1    return (n * recursiveFactorial (n - 1))
3  end if
end recursiveFactorial
```

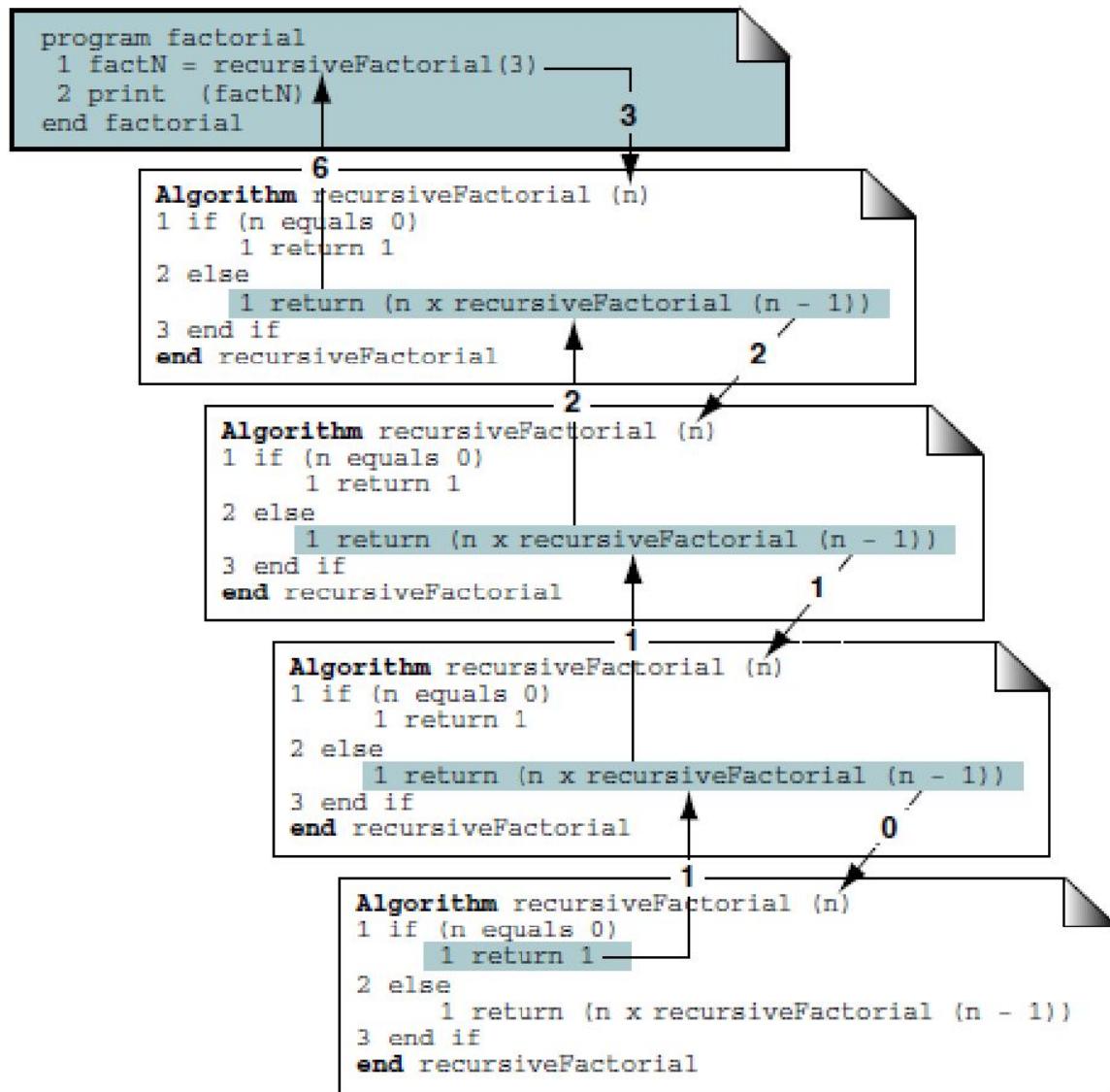
Caso Base

Caso Geral:
Relação de
Recorrência

Chamada
Recursiva

Decomposição do Fatorial Recursivo

- A figura ilustra a geração de funções para cada subproblema do fatorial.
- Cada chamada recursiva se sobrepõe a função de origem criando uma pilha de chamadas recursivas.
- Observe os caminhos de ida e volta:
 - No caminho de ida, etapa de decomposição, observe a passagem de parâmetros.
 - No caminho de volta, resolução dos subproblemas, observe o retorno dos resultados



Limitações da Recursividade

- Não deveríamos utilizar recursividade **se a resposta a qualquer uma das seguintes questões é NÃO:**
 - O algoritmo ou a estrutura de dados é apropriada de forma natural para a recursão ?;
 - A solução recursiva é menor e mais comprehensível ?;
 - A recursão executa dentro de padrões aceitáveis de limites de tempo e espaço ?.

Inversão da entrada do teclado

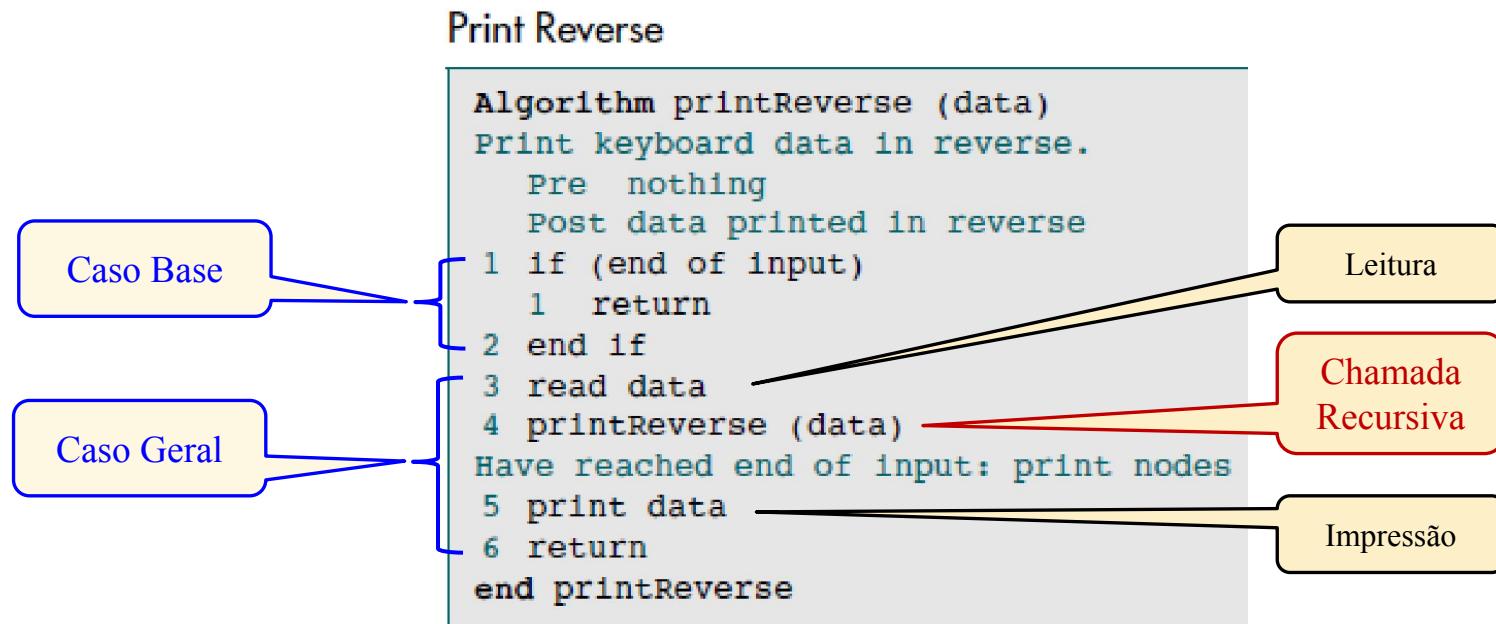
Algoritmo Recursivo

- Considere que estamos lendo a entrada do teclado e que precisamos imprimir os dados em ordem inversa.
- Uma maneira de imprimir os dados em ordem inversa consiste em utilizar um algoritmo recursivo.
- Uma questão óbvia é que para imprimir uma lista em ordem inversa devemos, primeiro ler todos os dados. Assim:
 - O **caso base**, só acontece quando acabamos a leitura do último dado.
 - O **caso geral** consiste em ler o próximo dado.
- A questão que fica é quando imprimir?
- Somente podemos imprimir após ler o último dado, ou seja na volta da chamada recursiva. Caso contrário, imprimiremos na ordem de leitura.

Inversão da entrada do teclado

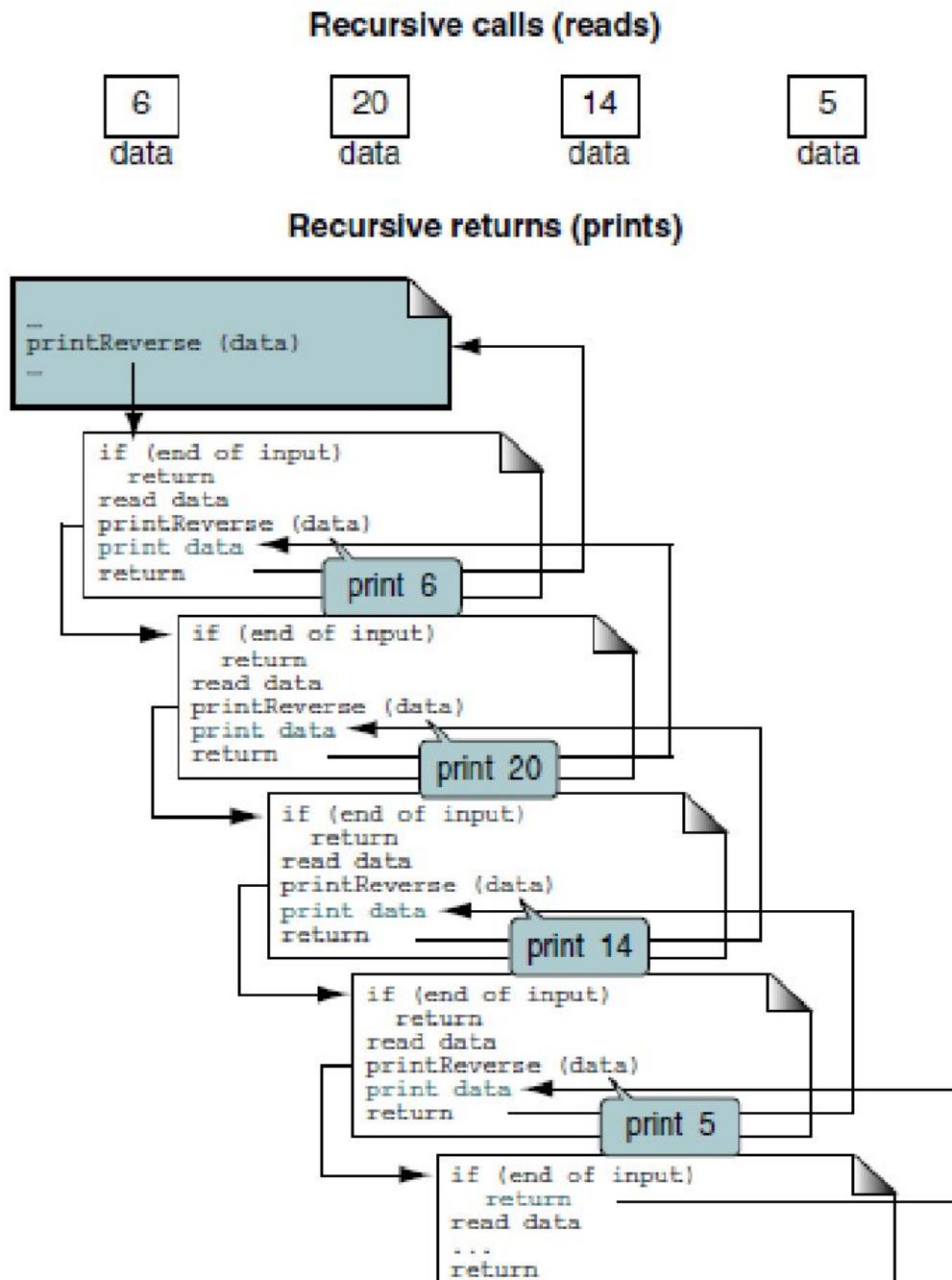
Algoritmo Recursivo

- Somente podemos imprimir após ler o último dado, ou seja na volta da chamada recursiva. Caso contrário, imprimiremos na ordem de leitura.
- Estude com atenção o seguinte algoritmo recursivo.



Decomposição da Inversão da entrada do teclado

- A figura ilustra a sequência de chamadas recursivas geradas por uma chamada individual da função recursiva PrintReverse.
- Cria-se uma pilha de funções recursivas.
- Observe os caminhos de ida e volta:
 - No caminho de ida (etapa de decomposição), se verifica a condição de parada. Caso não seja verificada essa condição, se faz a leitura de um novo dado.
 - No caminho de volta (etapa de resolução), após atingir a condição de parada, cada função recursiva retorna a função de chamada e imprime o dado lido localmente.



Recursão

Mais Exemplos

- Discutimos 4 exemplos de algoritmos recursivos:
 - Máximo divisor comum;
 - Números de Fibonacci;
 - Torres de Hanoi;
 - Conversão Posfixa.

Máximo divisor comum

Exemplo 1 - Descrição

- Dados dois números a e b , queremos determinar o maior divisor comum (*Greatest Common Divisor*) entre eles. Para isso temos a seguinte definição de função recursiva:

$$\text{gcd}(a, b) = \begin{cases} a & \text{if } b = 0 \\ b & \text{if } a = 0 \\ \text{gcd}(b, a \bmod b) & \text{otherwise} \end{cases}$$

Greatest Common Divisor Recursive Definition

if $b = 0$
if $a = 0$
otherwise

Caso Base

Relação de Recorrência

- A função recursiva usa a operação módulo, $a \bmod b$, que calcula o resto da divisão de a entre b . Considera-se a maior que b e vice-versa.
- Se $a > b$, temos duas possibilidades: i) $(a \bmod b) = 0$, o que indica que b é maior divisor comum entre a e b , obtendo-se $\text{gdc}(a, b) = \text{gdc}(b, 0)$. ii) $(a \bmod b) = c > 0$, obtendo-se $\text{gdc}(a, b) = \text{gdc}(b, c)$ que indica que a recursão deve continuar.
- Se $b > a$, temos: $(a \bmod b) = a$ causando a inversão na ordem dos parâmetros de $\text{gdc}(a, b) = \text{gdc}(b, a)$.

Máximo divisor comum

Exemplo 1 - Algoritmo

- Definição do algoritmo euclidiano para o máximo divisor comum.
- O algoritmo reproduz fielmente a definição de função recursiva.
- Observe que temos dois casos base.

Euclidean Algorithm for Greatest Common Divisor

```
Algorithm gcd (a, b)
Calculates greatest common divisor using the Euclidean algorithm.

Pre a and b are positive integers greater than 0
Post greatest common divisor returned

1 if (b equals 0)
  1 return a
2 end if
3 if (a equals 0)
  2 return b
4 end if
5 return gcd (b, a mod b)
end gcd
```

Casos Base

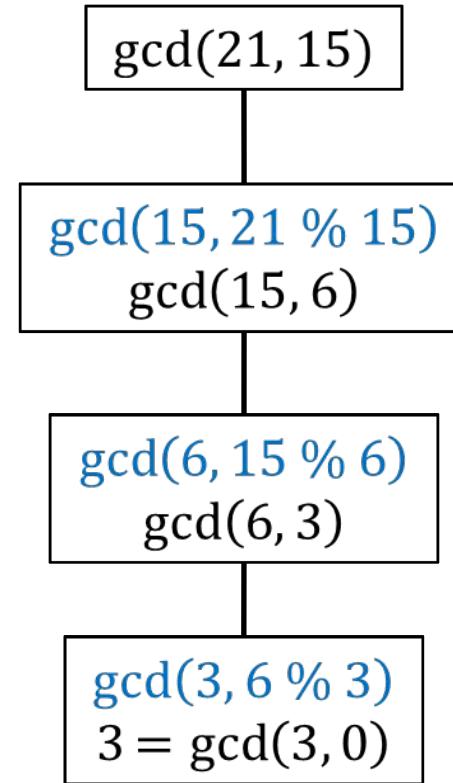
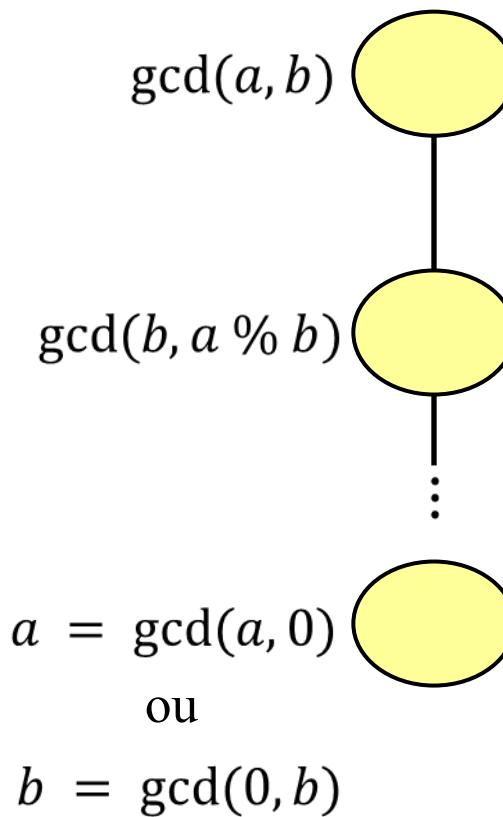
Caso Geral

Máximo divisor comum

Exemplo 1 - Decomposições

- O algoritmo recursivo $\text{gdc}(a, b)$ reduz o tamanho do problema ao diminuir o tamanho dos parâmetros a e b nos subproblemas resultantes até achar um divisor comum.

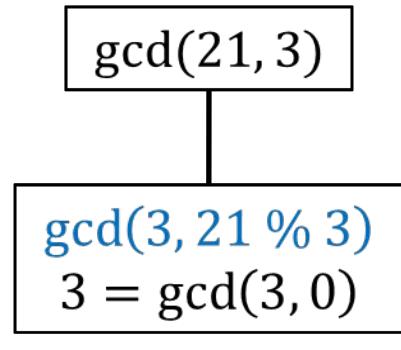
Ex. (i)



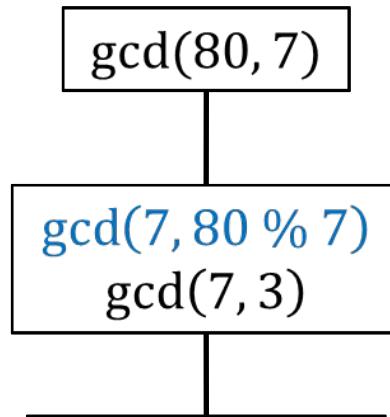
Máximo divisor comum

Exemplo 1 - Decomposições

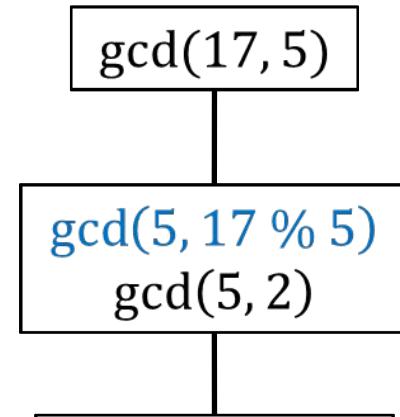
Ex. (ii)



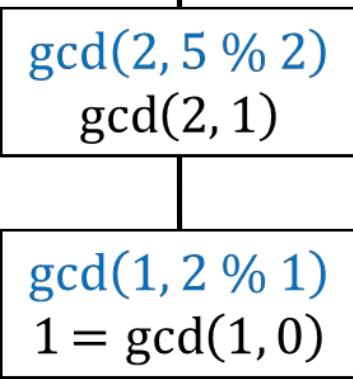
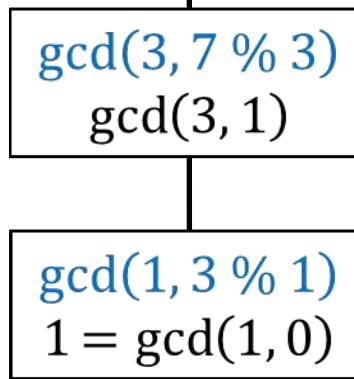
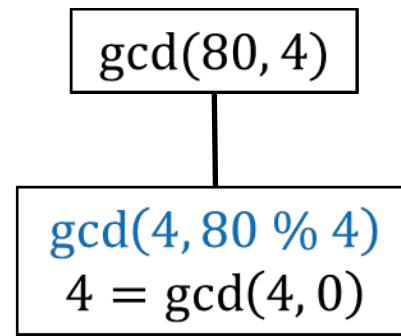
Ex. (iv)



Ex. (v)



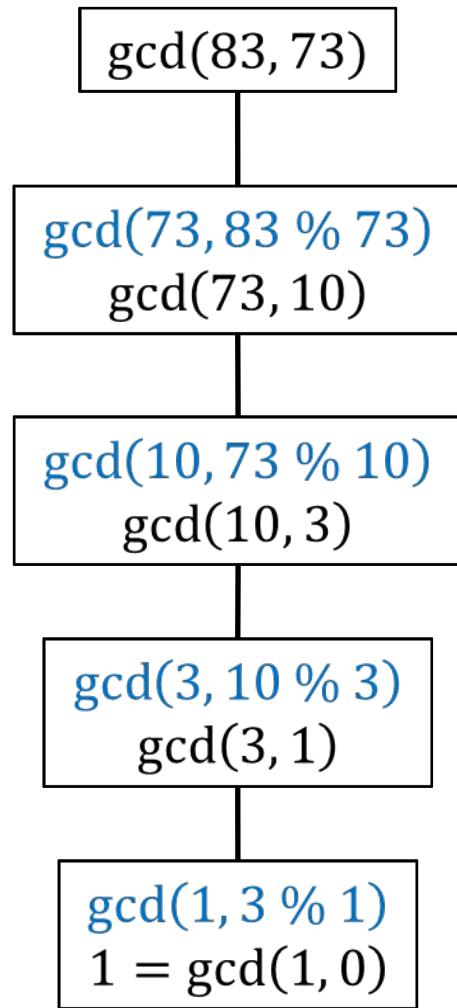
Ex. (iii)



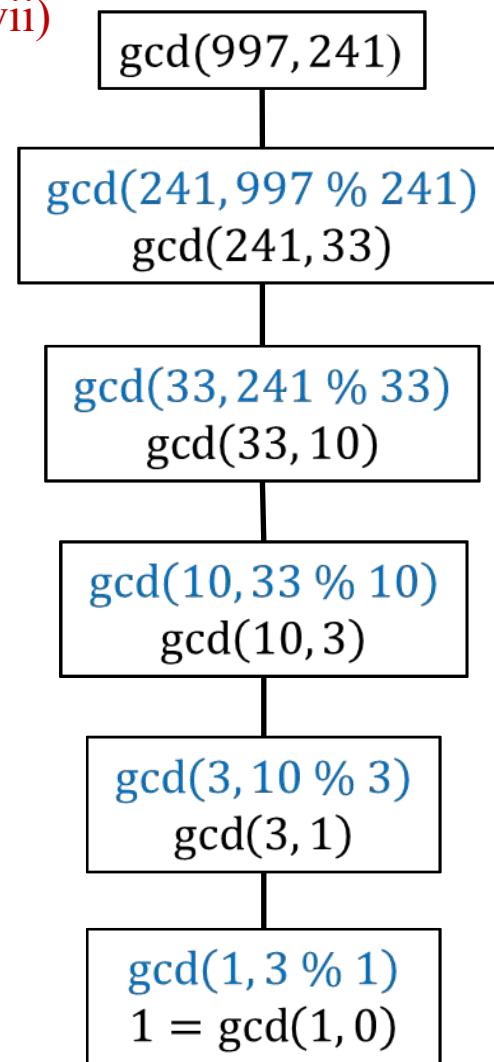
Máximo divisor comum

Exemplo 1 - Decomposições

Ex. (vi)



Ex. (vii)



Máximo divisor comum

Implementação

P2-01.c

- Esta parte do código ilustra o programa principal `main()` e o uso da função recursiva `gdc(a, b)`.

```
1  /* This program determines the greatest common divisor
2   of two numbers.
3
4   Written by:
5   Date:
6 */
7 #include <stdio.h>
8 #include <ctype.h>
9
10 // Prototype Statements
11 int gcd (int a, int b);
12
13 int main (void)
14 {
15 // Local Declarations
16     int gcdResult;
17
18 // Statements
19     printf("Test GCD Algorithm\n");
20
21     gcdResult = gcd (10, 25);
22     printf("GCD of 10 & 25 is %d", gcdResult);
23     printf("\nEnd of Test\n");
24
25     return 0;
```

Uso da Função
Recursiva

Máximo divisor comum

Implementação

P2-01.c (Continuação...)

- Esta parte do código ilustra a implementação da função recursiva $\text{gdc}(a, b)$.

```
24 } // main
25 /* ===== gcd =====
26     Calculates greatest common divisor using the
27     Euclidean algorithm.
28     Pre a and b are positive integers greater than 0
29     Post greatest common divisor returned
30 */
31 int gcd (int a, int b)
32 {
33     // Statements
34     if (b == 0)
35         return a;
36     if (a == 0)
37         return b;
38     return gcd (b, a % b);
39 } // gcd
```

Função
Recursiva

Chamada
Recursiva

Results:
Test GCD Algorithm
GCD of 10 & 25 is 5
End of Test

Números de Fibonacci

Exemplo 2 - Descrição

- A série de Fibonacci é uma sequência de números em que cada novo número é calculado como a soma dos dois números precedentes. Os dois primeiros números da série são zero e um. Temos assim, a seguinte definição de função recursiva:

$$\text{Fibonacci}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{Fibonacci}(n - 1) + \text{Fibonacci}(n - 2) & \text{otherwise} \end{cases}$$

Fibonacci Numbers Recursive Definition

Caso Base

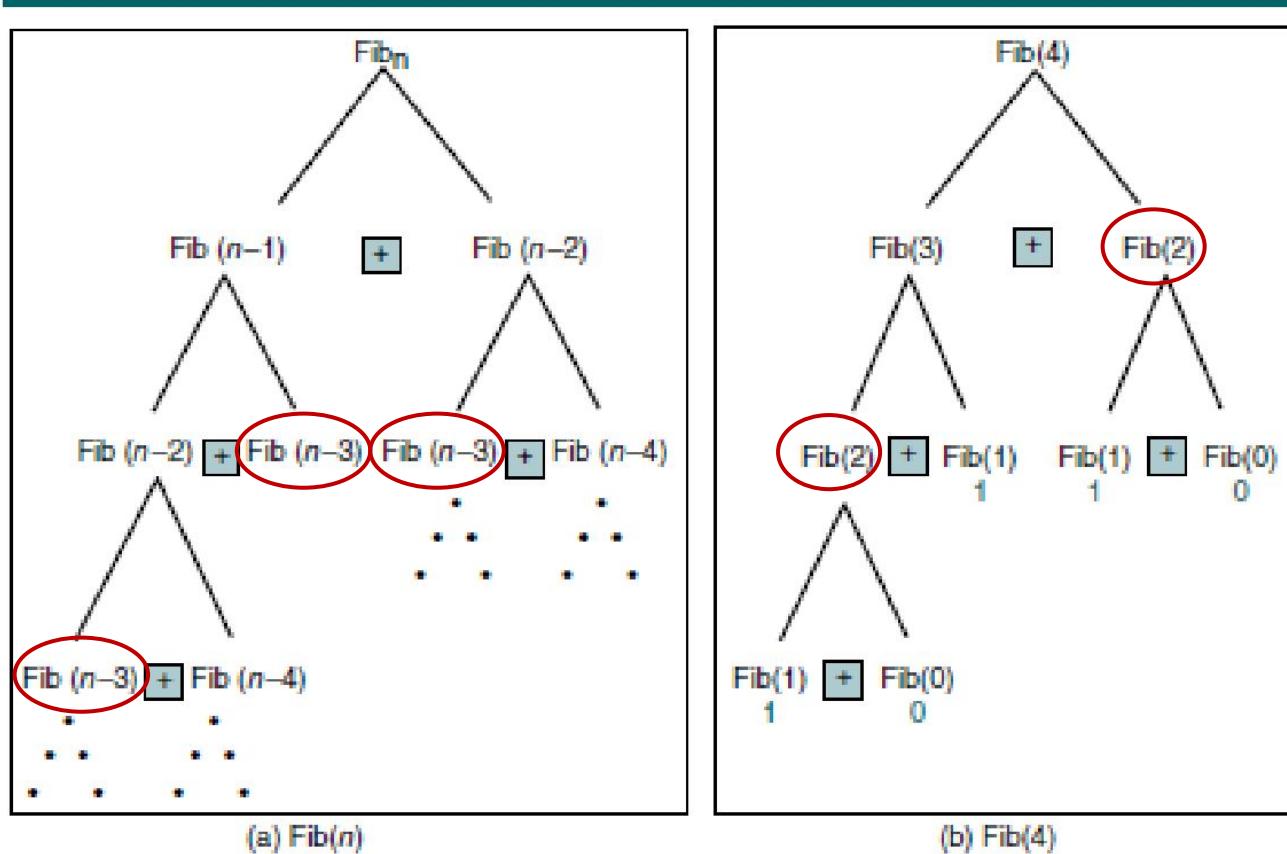
Relação de Recorrência

- Observe que a relação de recorrência faz duas chamadas recursivas, ou decompõe o problema de tamanho n em dois subproblemas de tamanhos $n-1$ e $n-2$.

Números de Fibonacci

Exemplo 2 - Decomposição

- A figura ilustra o processo de decomposição da função Fibonacci recursiva.
- Cada chamada recursiva gera dois subproblemas.
- Observe que a redução do tamanho não é significativa.
- Este algoritmo é bastante ineficiente devido ao fato de que o mesmo subproblema é gerado e resolvido várias vezes.



Fibonacci Numbers

Números de Fibonacci

Exemplo 2 - Implementação

P2-02.c

- Esta parte do código ilustra o programa principal `main()` e o uso da função recursiva `fib(n)`.
- Os números da série são impressos 5 em cada linha. Utiliza-se o operador módulo para controlar o salto de linha.

```
1  /* This program prints out a Fibonacci series.  
2   Written by:  
3   Date:  
4 */  
5 #include <stdio.h>  
6  
7 // Prototype Statements  
8 long fib (long num);  
9  
10 int main (void)  
11 {  
12 // Local Declarations  
13     int seriesSize = 10;  
14  
15 // Statements  
16     printf("Print a Fibonacci series.\n");  
17  
18     for (int looper = 0; looper < seriesSize; looper++)  
19     {  
20         if (looper % 5)  
21             printf(", %ld", fib(looper));  
22         else  
23             printf("\n%ld", fib(looper));  
24     } // for  
25     printf("\n");  
26     return 0;
```

Necessário para
armazenar números
muito grandes.

Valor do n

Uso da Função
Recursiva

Uso da Função
Recursiva

Números de Fibonacci

Exemplo 2 – Implementação

P2-02.c (Continuação...)

- Esta parte do código ilustra a implementação da função recursiva $\text{fib}(n)$.
- Observe a relação de recorrência com duas chamadas recursivas.

```
27 } // main
28
29 /* ===== fib =====
30     Calculates the nth Fibonacci number
31     Pre num identifies Fibonacci number
32     Post returns nth Fibonacci number
33 */
34 long fib (long num)
35 {
36     // Statements
37     if (num == 0 || num == 1)
38         // Base Case
39         return num;
40     return (fib (num - 1) + fib (num - 2));
41 } // fib
```

Função Recursiva

1^a. Chamada Recursiva

2^a. Chamada Recursiva

Results:
Print a Fibonacci series.

0, 1, 1, 2, 3
5, 8, 13, 21, 34

Números de Fibonacci

Exemplo 2 - Desempenho

- A tabela mostra o número de subproblemas (ou chamadas recursivas) geradas pelo algoritmo recursivo de Fibonacci. Observe que este número explode rapidamente, mesmo para tamanhos de n pequenos.

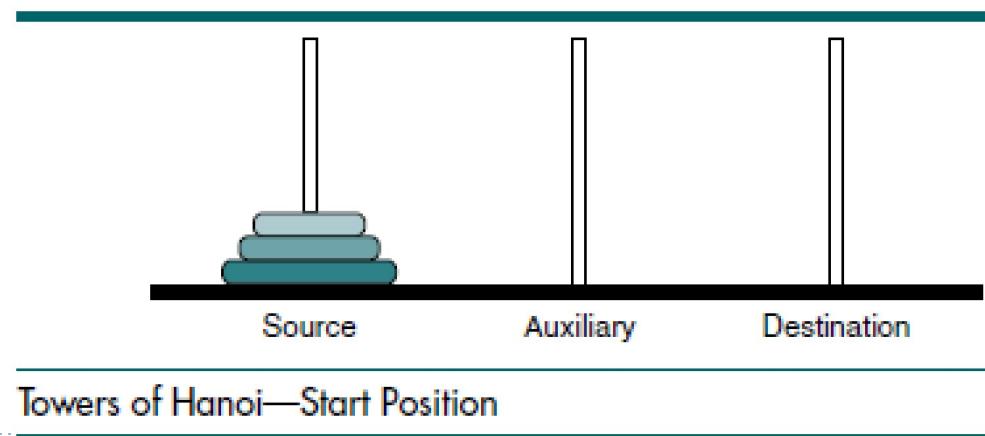
fib(n)	Calls	fib(n)	Calls
1	1	11	287
2	3	12	465
3	5	13	753
4	9	14	1219
5	15	15	1973
6	25	20	21,891
7	41	25	242,785
8	67	30	2,692,573
9	109	35	29,860,703
10	177	40	331,160,281

- Observação: A vírgula é o separador de milhar em países de fala inglesa; Substituir a vírgula por ponto.

Torres de Hanoi

Exemplo 3 - Descrição

- “No inicio dos tempos, Deus criou as Torres de Brahma com 64 discos de ouro puro, dispostos em agulhas de diamante. Quando a hora chegou, ordenou a um grupo de monges de um Monastério no Tibet que dessem inicio à movimentação dos discos segundo regras determinadas e alertou que concluído o trabalho as torres desmoronariam dando inicio ao final dos tempos”.
- O problema das Torres de Hanoi é um problema altamente complexo, que consiste na movimentação de discos de diâmetros diferentes de um pino para outro, observando-se a regra de que um disco de diâmetro maior não pode ser colocado sobre um disco de diâmetro menor.

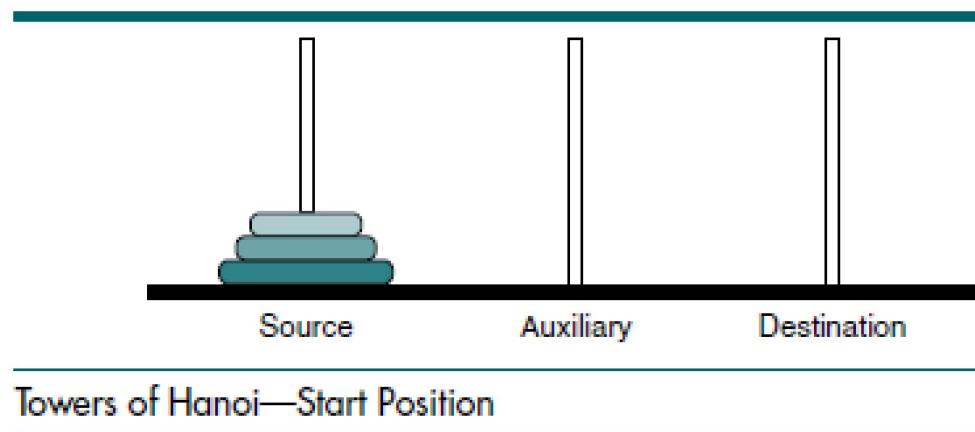


- A figura ilustra o caso com 3 discos.

Torres de Hanoi

Exemplo 3 - Descrição

- O problema das Torres de Hanoi consiste na movimentação de discos de diâmetros diferentes de um **pino origem** para um **pino destino**. Existe ainda um **pino auxiliar** que serve para facilitar a movimentação dos discos.
- Somente podemos movimentar um disco de cada vez.
- Na movimentação, um disco de diâmetro maior não pode ser colocado sobre um disco de diâmetro menor.
- Na medida que número de discos (tamanho do problema n) aumenta, o número de movimentações necessárias explode.

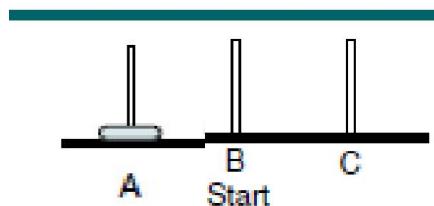


Torres de Hanoi

Exemplo 3 - Descrição

- **Caso com um disco:**
- Este é o caso trivial, que corresponde ao **Caso Base**, e consiste na movimentação de um único disco. Neste caso, movimentamos o disco diretamente do pino origem (A) ao pino destino (C).

Case 1: Move one disk from source to destination needle.



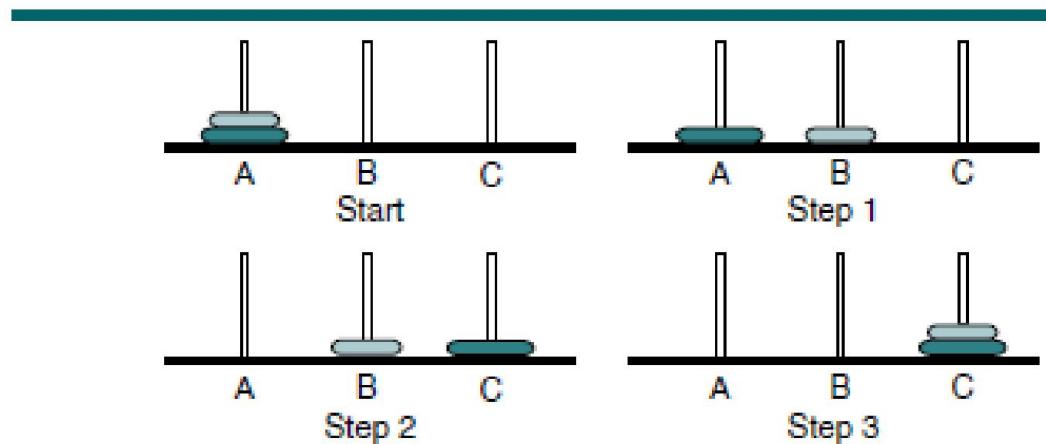
Torres de Hanoi

Exemplo 3 - Descrição

- **Caso com dois discos:**
- Queremos movimentar dois discos do pino A para o pino C.
- Este caso já exige o uso do pino auxiliar (B).
- Movimentamos primeiro o disco menor de A para B. O disco maior fica assim liberado, e é movimentado diretamente do pino origem (A) para o pino destino (C). Finamente, o disco menor é movimentado de B para C.
- Observe que o problema com dois discos é decomposto em dois subproblemas com um disco (movimentar o disco menor).

Case 2:

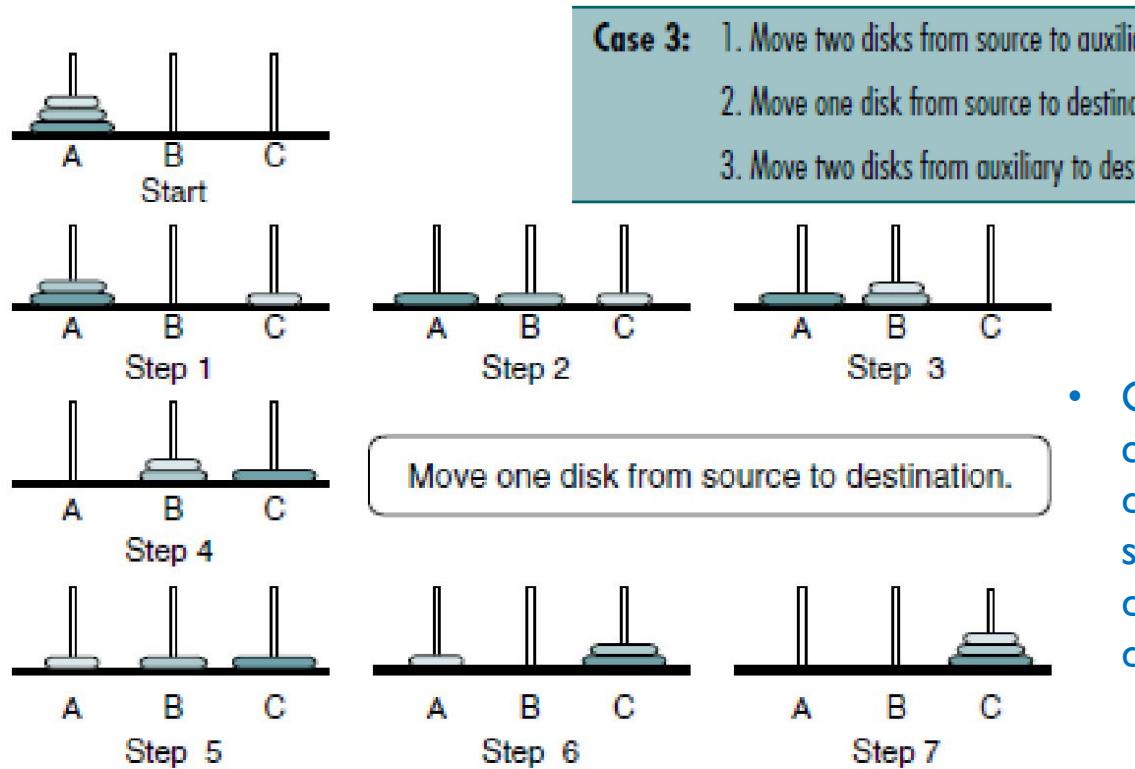
1. Move one disk to auxiliary needle.
2. Move one disk to destination needle.
3. Move one disk from auxiliary to destination needle.



Torres de Hanoi

Exemplo 3 - Descrição

- **Caso com três discos:**
- Queremos movimentar três discos do pino A para o pino C.
- Movimentamos primeiro os dois discos menores de A para B. O disco maior fica assim liberado, e é movimentado diretamente do pino origem (A) para o pino destino (C). Finalmente, os dois discos menores são movimentados de B para C.

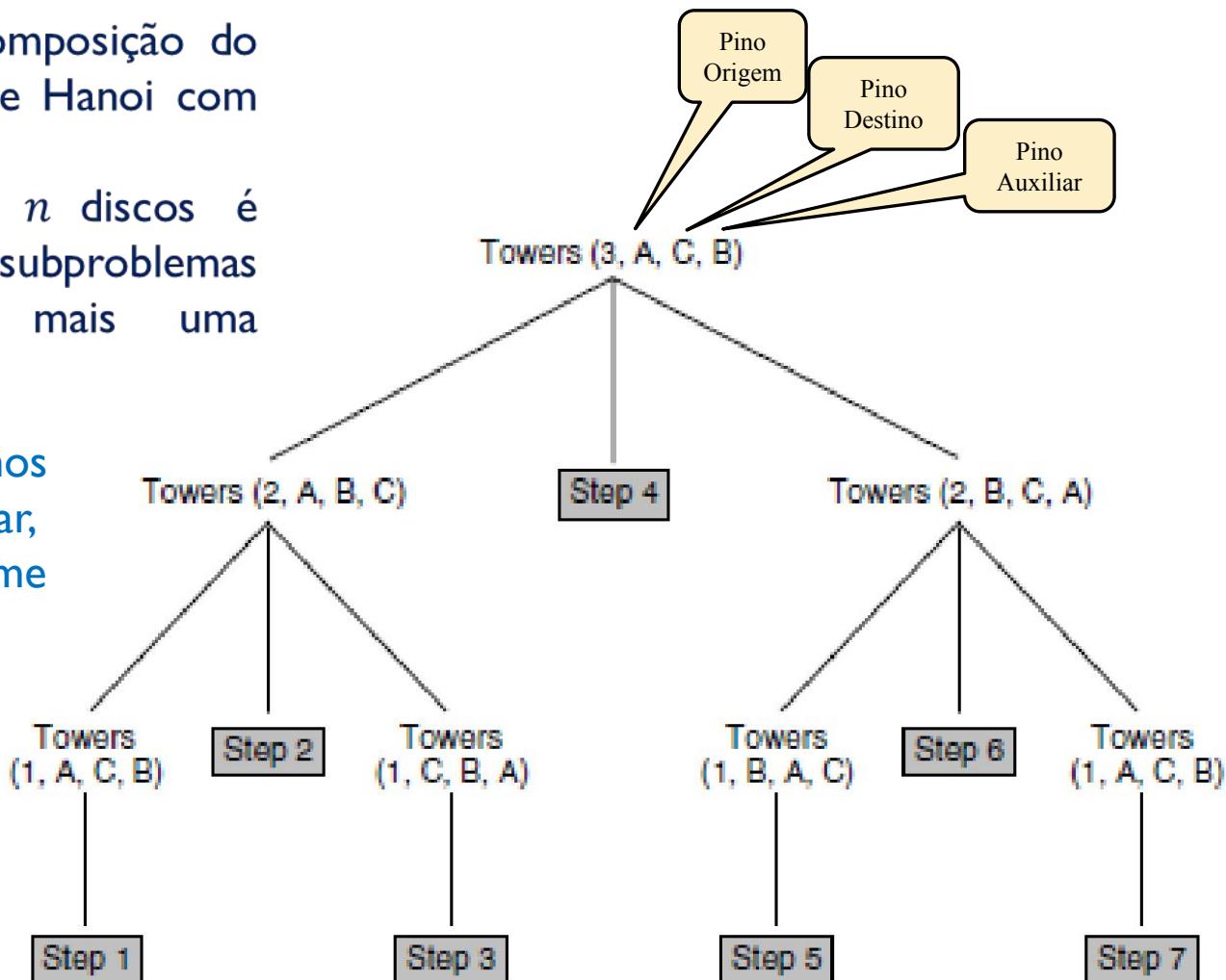


- Observe que o problema com três discos é decomposto em dois subproblemas com dois discos (movimentar os dois discos menores).

Torres de Hanoi

Exemplo 3 - Decomposição

- A figura ilustra a decomposição do problema das Torres de Hanoi com três discos.
- Cada problema com n discos é decomposto em dois subproblemas com $n - 1$ discos, mais uma movimentação direta.
- Observe que os pinos origem, destino e auxiliar, variam conforme necessidade.



Torres de Hanoi

Exemplo 3 - Algoritmo

- **Caso Geral:**
- Podemos assim, generalizar o algoritmo recursivo das Torres de Hanoi para n discos.
- O problema da movimentação de n discos é decomposto em dois subproblemas de movimentação de $n - 1$ discos. Assim que os $n - 1$ discos menores são movimentados do pino origem para o pino auxiliar, o disco maior é movido diretamente do pino origem para o pino destino. Finalmente, os $n - 1$ discos menores são movimentados para do pino auxiliar para o pino destino.

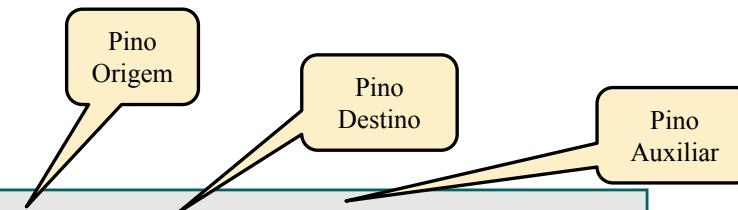
1. Move $n - 1$ disks from source to auxiliary.	General case
2. Move one disk from source to destination.	Base case
3. Move $n - 1$ disks from auxiliary to destination.	General case

```
1. Call Towers (n - 1, source, auxiliary, destination)
2. Move one disk from source to destination
3. Call Towers (n - 1, auxiliary, destination, source)
```

Torres de Hanoi

Exemplo 3 - Algoritmo

- A figura ilustra o algoritmo recursivo para o Problema das Torres de Hanoi com n discos.
- Observe que a movimentação de discos é simulada mediante uma impressão.



```
Algorithm towers (numDisks, source, dest, auxiliary)
  Recursively move disks from source to destination.
  Pre numDisks is number of disks to be moved
    source, destination, and auxiliary towers given
  Post steps for moves printed
1 print("Towers: ", numDisks, source, dest, auxiliary)
2 if (numDisks is 1)
  1 print ("Move from ", source, " to ", dest)
3 else
  1 towers (numDisks - 1, source, auxiliary, dest, step)
  2 print ("Move from " source " to " dest)
  3 towers (numDisks - 1, auxiliary, dest, source, step)
4 end if
end towers
```

Caso Base

Caso Geral

Torres de Hanoi

Exemplo 3 - Implementação

P2-04.c

- Esta parte do código ilustra o programa principal *main()* e o uso da função recursiva *towers(n, ori, des, aux)*.
- O programa começa com a leitura do número de discos.
- Mostra como resultado a sequência de movimentos a serem realizados.

```
1  /* Test Towers of Hanoi
2      Written by:
3      Date:
4  */
5 #include <stdio.h>
6
7 // Prototype Statements
8 void towers (int n,      char source,
9                 char dest,   char auxiliary);
10
11 int main (void)
12 {
13 // Local Declarations
14     int numDisks;
15
16 // Statements
17     printf("Please enter number of disks: ");
18     scanf ("%d", &numDisks);
19
20     printf("Start Towers of Hanoi.\n\n");
21
22     towers (numDisks, 'A', 'C', 'B');
23
24     printf("\nI Hope you didn't select 64 "
25           "and end the world!\n");
26     return 0;
27 } // main
```

Uso da Função Recursiva

Torres de Hanoi

Exemplo 3 - Implementação

P2-04.c (Continuação...)

- Esta parte do código ilustra a implementação da função recursiva `towers(n, ori, des, aux)`.
- Observe a relação de recorrência com duas chamadas recursivas.

```
29  /* ===== towers =====
30   Move one disk from source to destination through
31   the use of recursion.
32   Pre  The tower consists of n disks
33   Source, destination, & auxiliary towers
34   Post Steps for moves printed
35 */
36 void towers (int n, char source,
37             char dest, char auxiliary)
38 {
39 // Local Declarations
40     static int step = 0;
41
42 // Statements
43 printf("Towers (%d, %c, %c, %c)\n",
44           n, source, dest, auxiliary);
45 if (n == 1)
46     printf("\t\t\tStep %3d: Move from %c to %c\n",
47           ++step, source, dest);
48 else
49 {
50     towers (n - 1, source, auxiliary, dest);
51     printf("\t\t\tStep %3d: Move from %c to %c\n",
52           ++step, source, dest);
53     towers (n - 1, auxiliary, dest, source);
54 } // if ... else
55 return;
56 } // towers
```

Função
Recursiva

Impressão

1ª. Chamada
Recursiva

Impressão

2ª. Chamada
Recursiva

Conversão Posfixa

Exemplo 4 - Descrição

- Diferentes notações são usadas para representar expressões aritméticas.
- Na notação pré-fixa o operador precede aos dois operandos.
- Enquanto, na notação infixa o operador se encontra no meio dos dois operandos.
- Já na notação pós-fixa o operador sucede aos dois operandos.

Prefix:	+ A B
Infix:	A + B
Postfix:	A B +

In *prefix* notation the operator comes *before* the operands.

In *infix* notation the operator comes *between* the operands.

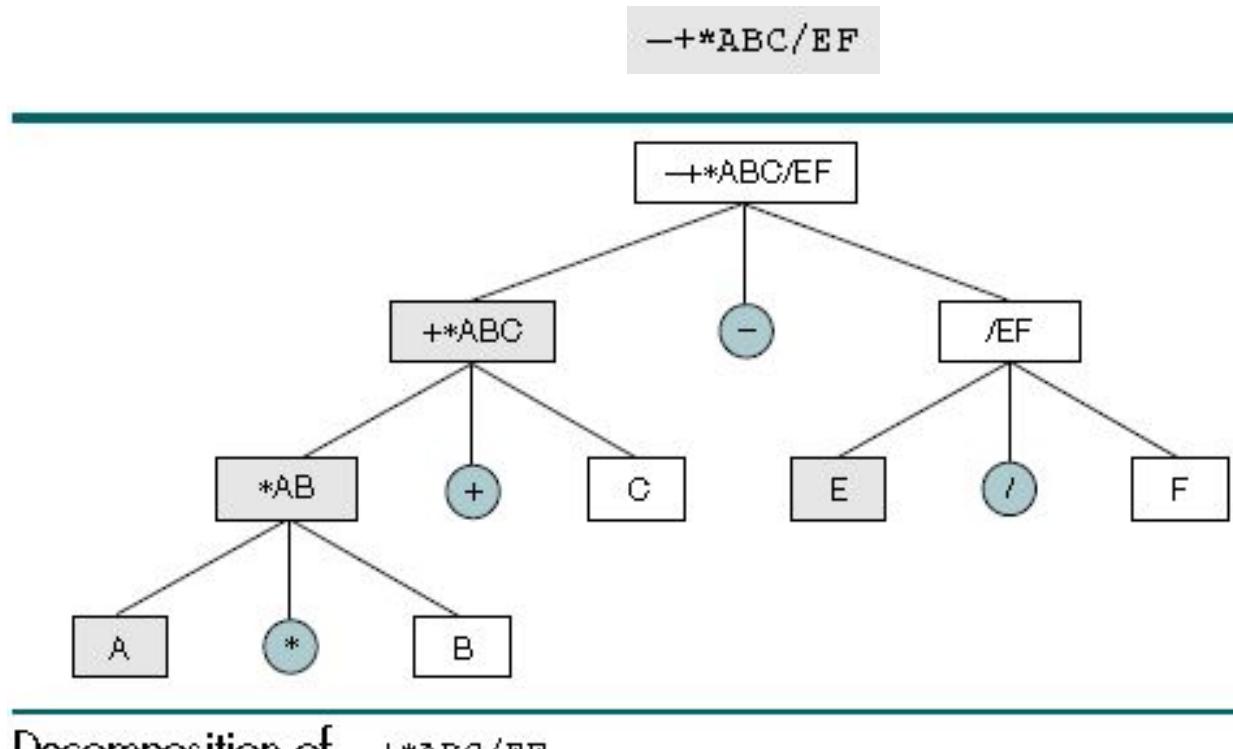
In *postfix* notation the operator comes *after* the operands.

- Estudaremos um algoritmo recursivo para converter uma expressão pré-fixa na notação pós-fixa.

Conversão Posfixa

Exemplo 4 - Decomposição

- Como em todos os casos os operadores são binários, as expressões podem ser decompostas em duas partes. A figura ilustra a decomposição de uma expressão pré-fixa em duas componentes mais o operador correspondente.
- Esse processo se repete até ter um termo de comprimento um (Caso Base).



Conversão Posfixa

Exemplo 4 - Implementação

P2-03.c

- Esta parte do código ilustra o programa principal *main()* e o uso da função recursiva *preToPostFix(pre, pos)*.
- O programa começa inicializando a expressão pré-fixa.
- Mostra como resultado a expressão pós-fixa.

```
1  /* Convert prefix to postfix expression.
2   Written by:
3   Date:
4   */
5  #include <stdio.h>
6  #include <string.h>
7
8  #define OPERATORS "+-*"
9
10 // Prototype Declarations
11 void preToPostFix (char* preFixIn, char* exprout);
12 int findExprLen (char* exprIn);
13
14 int main (void)
15 {
16 // Local Definitions
17     char preFixExpr[256] = "-+*ABC/EF";
18     char postFixExpr[256] = "";
19
20 // Statements
21     printf("Begin prefix to postfix conversion\n\n");
22
23     preToPostFix (preFixExpr, postFixExpr);
24     printf("Prefix expr: %s\n", preFixExpr);
25     printf("Postfix expr: %s\n", postFixExpr);
26
27     printf("\nEnd prefix to postfix conversion\n");
28     return 0;
29 } // main
```

Uso da Função
Recursiva

Conversão Posfixa

Ex.4-Implementação

P2-03.c (Continuação...)

- Esta parte do código ilustra a implementação da função recursiva:
`preToPostFix(pre, pos).`
- O algoritmo identifica o comprimento da primeira expressão. Identificando assim as duas componentes.
- Decompõe assim o problema em dois subproblemas.

```
38 void preToPostFix (char* preFixIn, char* postFix)
39 {
40     // Local Definitions
41     char operator [2];
42     char postFix1[256];
43     char postFix2[256];
44     char temp [256];
45     int lenPreFix;
46
47     // statements
48     if (strlen(preFixIn) == 1)
49     {
50         *postFix = *preFixIn;
51         *(postFix + 1) = '\0';
52         return;
53     } // if only operand
54
55     *operator = *preFixIn;
56     *(operator + 1) = '\0';
57
58     // Find first expression
59     lenPreFix = findExprLen (preFixIn + 1);
60     strncpy (temp, preFixIn + 1, lenPreFix);
61     *(temp + lenPreFix) = '\0';
62     preToPostFix (temp, postFix1);
63
64     // Find second expression
65     strcpy (temp, preFixIn + 1 + lenPreFix);
66     preToPostFix (temp, postFix2);
67
68     // Concatenate to postFix
69     strcpy (postFix, postFix1);
70     strcat (postFix, postFix2);
71     strcat (postFix, operator);
72
73     return;
74 } // preToPostFix
```

Função Recursiva

Caso Base

Comprimento da primeira expressão

1ª. Chamada Recursiva

2ª. Chamada Recursiva

Concatena as duas expressões postfixas e o operador

Conversão Posfixa

Ex.4-Implementação

P2-03.c (Continuação...)

- Esta parte do código ilustra a implementação da função recursiva:
`findExprLen(pre)`.
- Esta função identifica o comprimento da expressão de forma recursiva.
- Decompõe o problema em dois subproblemas.

```
76 /* ===== findExprLen =====
77   Determine size of first substring in an expression.
78   Pre exprIn contains prefix expression
79   Post size of expression is returned
80 */
81 int findExprLen (char* exprIn)
82 {
83 // Local Definitions
84     int len1;
85     int len2;
86
87 // Statements
88     if (strcspn (exprIn, OPERATORS) == 0)
89         // General Case: First character is operator
90         // Find length of first expression
91     {
92         len1 = findExprLen(exprIn + 1);
93
94         // Find length of second expression
95         len2 = findExprLen(exprIn + 1 + len1);
96     } // if
97     else
98         // Base case--first char is operand
99         len1 = len2 = 0;
100    return len1 + len2 + 1;
101 } // findExprLen
```

Expressão
Prefixa

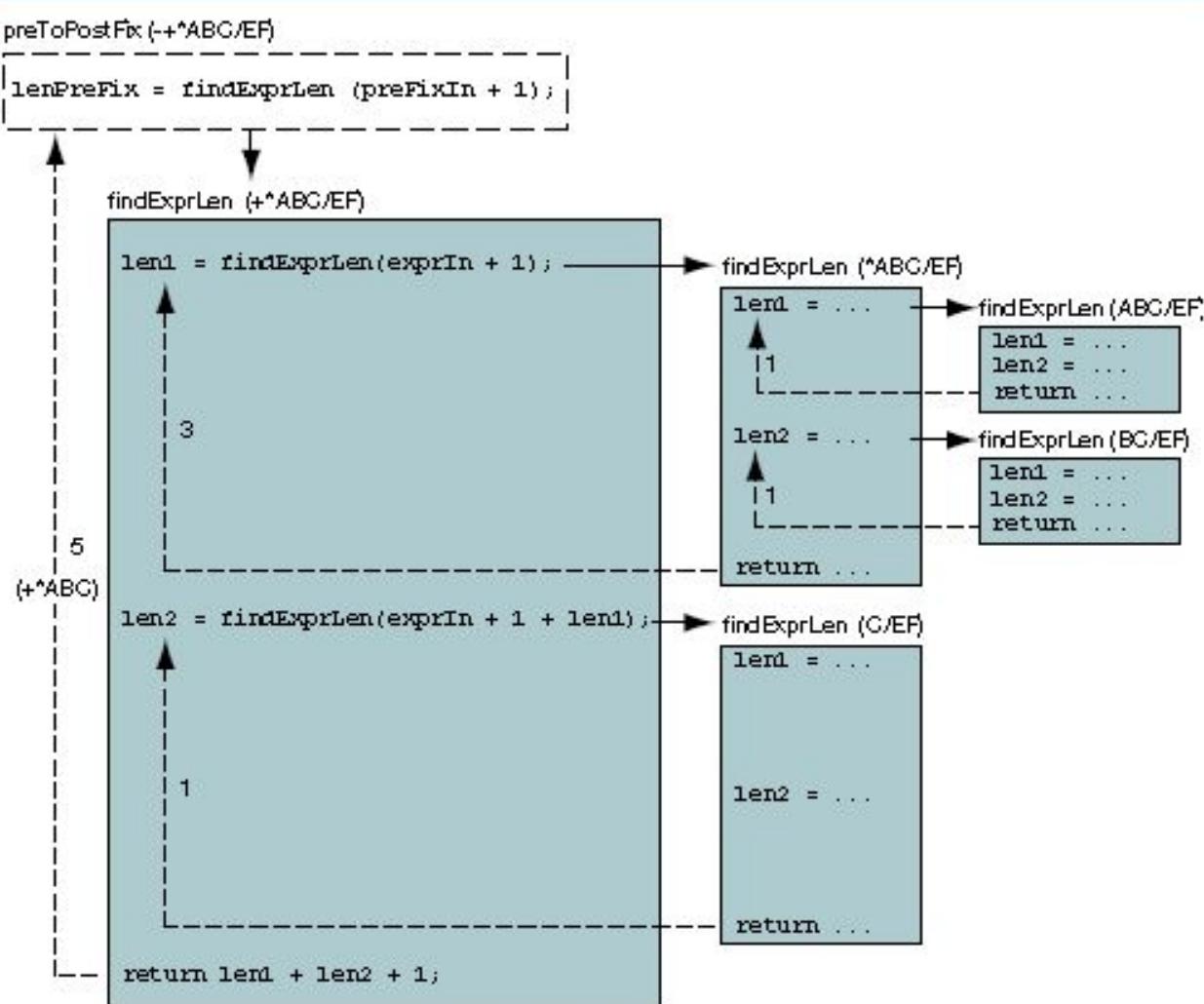
1ª. Chamada
Recursiva

2ª. Chamada
Recursiva

Conversão Posfixa

Exemplo 4 - Função *findExprLen()*

- A figura ilustra a decomposição do algoritmo recursivo: *findExprLen(pre)* em dois subproblemas.



Conversão Posfixa

Ex.4 - Resultados

- A Figura ilustra a expressão posfixa correspondente.

```
Results:  
Begin prefix to postfix conversion  
  
Prefix expr: -+*ABC/EF  
Postfix expr: AB*C+EF/-  
  
End prefix to postfix conversion
```

Referências

- Gilberg, R.F. e Forouzan, B.A. Data Structures_A Pseudocode Approach with C. Capítulo 2. Recursion. Segunda Edição. Editora Cengage, Thomson Learning, 2005.