

Estruturas de Informação

Análise de Complexidade

Departamento de Engenharia Informática (DEI/ISEP)

Fátima Rodrigues

mfc@isep.ipp.pt

Análise de Complexidade

"Ao verificar que um dado programa está muito lento, um programador menos responsável, normalmente culpabiliza as capacidades de processamento da máquina e provavelmente pede nova máquina [...]"

Entretanto, o ganho potencial que uma máquina mais rápida pode proporcionar é tipicamente limitado por um factor de 10, por razões técnicas e/ou económicas.

Para obter um maior ganho de desempenho, é preciso desenhar ***melhores algoritmos – algoritmos eficientes***.

Um algoritmo rápido em execução numa máquina lenta terá *sempre um melhor desempenho* para grandes instâncias do problema. *Sempre.*"

S. S. Skiena, *The Algorithm Design Manual*

Introdução

- **Algoritmo:** conjunto claramente específico de instruções a seguir para resolver um problema
 - o mesmo algoritmo pode ser "implementado" de diferentes formas
- Descrição de algoritmos:
 - em linguagem natural, pseudo-código, numa linguagem de programação...
- Um algoritmo deve apresentar as seguintes características:
 - ter entrada e saída
 - ser finito
 - ser definido
 - ser correcto
 - ser eficiente

Características Desejáveis dos Algoritmos

- **Correcção** significa trabalhar correctamente quaisquer que sejam os dados de entrada dentro do domínio de valores admissíveis pelas suas variáveis
- **Eficiência** significa que os algoritmos devem ser rápidos e nunca usar recursos do computador superiores ao necessário

Estes objectivos serão alcançados se a implementação dos algoritmos obedecer aos seguintes objectivos:

- **Robustez**

- ♦ é a capacidade do algoritmo dar resposta à manipulação de entradas não esperadas

- **Adaptabilidade**

- ♦ é a capacidade do algoritmo conseguir dar resposta a alterações

- **Reutilização**

- ♦ significa o mesmo código seja um componente de diferentes sistemas em vários domínios de aplicação

Critérios de Escolha

Dado um problema podem existir vários algoritmos possíveis

Critérios de escolha:

- Extensibilidade
- Modularidade
- Portabilidade
- **Eficiência** (em função da dimensão do problema)
 - ♦ **em tempo** Complexidade Temporal
 - ♦ **em espaço** Complexidade Espacial

Complexidade Espacial e Temporal

- **Complexidade Espacial $S(n)$** de um programa ou algoritmo: é o espaço de memória que necessita para executar até ao fim em função do tamanho (n) da entrada
- **Complexidade Temporal $T(n)$** de um programa ou algoritmo: é o tempo que demora a executar (tempo de execução) em função do tamanho (n) da entrada

Complexidade \uparrow versus Eficiência \downarrow

- **Análise de Complexidade** de um algoritmo envolve determinar os recursos (tempo, espaço) exigidos pelo algoritmo
- Para diferentes algoritmos que resolvem o mesmo problema, um algoritmo é mais eficiente, se exige menos recursos para resolver o mesmo problema

Análise de Complexidade

O valor exacto do tempo de execução de um algoritmo depende também:

- da linguagem de programação
- da máquina utilizada
- A análise de Complexidade não considera estes factores **apenas a ordem de grandeza Tempo** em função da quantidade dos dados de entrada

Exemplo: ordenar um vector com 10 elementos não leva o mesmo tempo que ordenar um vector com 1000 elementos

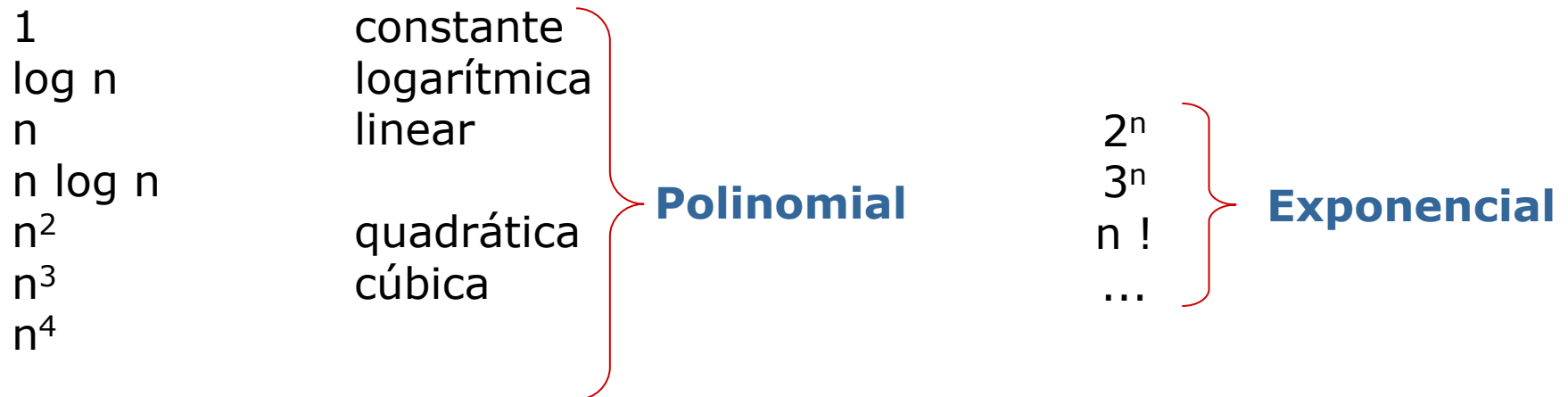
- **O que interessa é relacionar a variação que existe no tempo de ordenação com a variação do número de elementos**
- prever o crescimento dos recursos exigidos pelo algoritmo à medida que o tamanho dos dados de entrada cresce

Algumas Funções Típicas

Basicamente existem dois tipos de algoritmos:

- Os que têm tempo de execução limitável por um polinómio dependente do tamanho da entrada
→ **Algoritmos eficientes**
- E os que não são limitáveis por um polinómio – têm normalmente uma evolução exponencial
→ **Algoritmos não eficientes**

$T(n)$ – ordem de grandeza do tempo/espaco requerido para um problema de dimensão n



Ordens de Complexidade mais comuns

Considerando $n = 10^5$ elementos e assumindo como tempo de execução de cada passo $k = 10^{-5}$ segundos = $10 \mu\text{s}$

		Tempo de Execução $10^{-5} \text{ g } (10^5)$
$O(3^n)$	Tempo exponencial	50 000 horas
$O(n^2)$	Tempo quadrático	28 horas
$O(n \log n)$	Tempo $n \log n$	17 seg
$O(n)$	Tempo linear	1 seg
$O(\log n)$	Tempo logarítmico	$170 \mu\text{s}$
$O(1)$	Tempo constante	$10 \mu\text{s}$

Ordens de Complexidade mais comuns

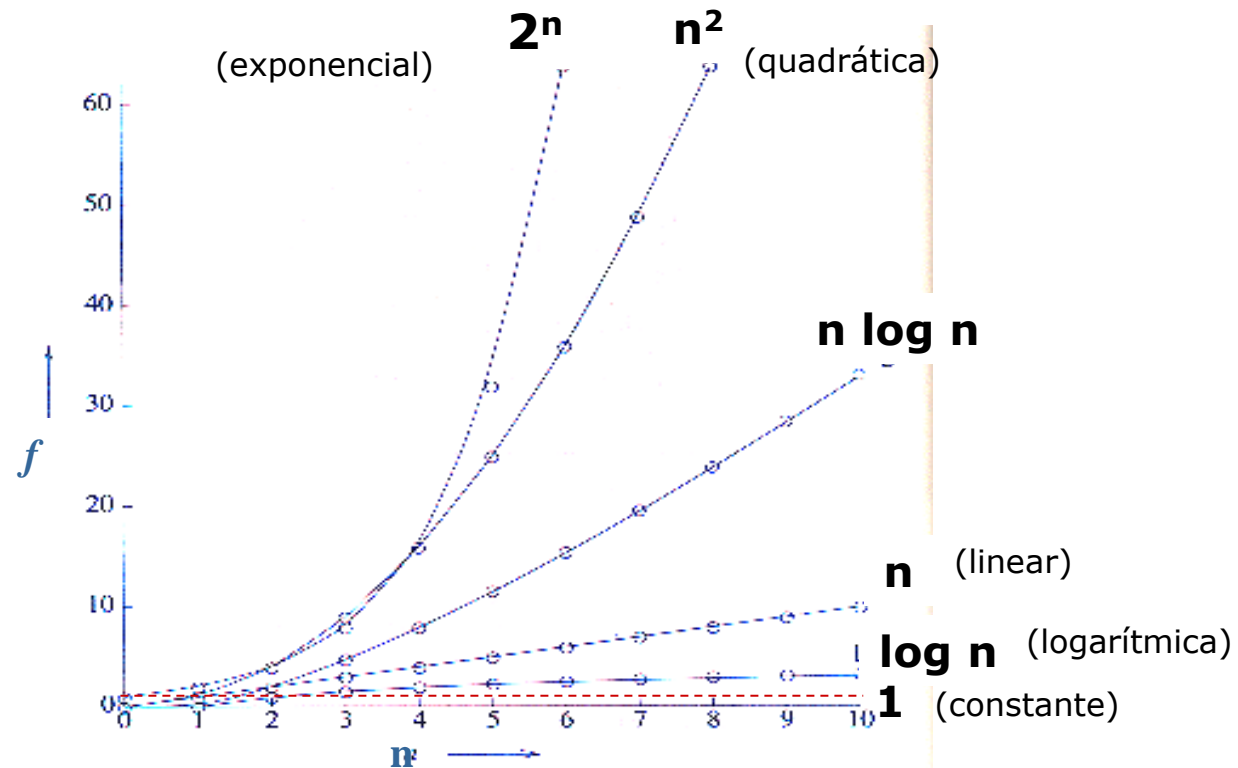


Figure 2.24 Plot of various functions

Notações O , Ω e Θ

Definição (notação O) = Limite superior para o tempo de execução

Definição (notação Ω) = Limite inferior para o tempo de execução

Definição (notação Θ) = Tempo de execução exacto

Notação Big-Oh (O Grande)

Num algoritmo que toma como domínio um conjunto de n elementos, a notação Big-Oh exprime o tipo de proporcionalidade existente entre n e o tempo $t(n)$ que demora a sua execução

Este tipo de notação é **assimptótica** porque vai ser definida para um comportamento limite quando aumenta o tamanho do problema

Definição:

$$T(n) = O(f(n)) \quad (\text{diz-se que } T(n) \text{ é de ordem } f(n))$$

se e só se existem constantes positivas c e n_0 tal que

$$T(n) \leq c f(n) \quad \text{para todo o } n > n_0, n_0 > 0, c > 0$$

Com a notação Big-Oh garantimos que a complexidade da função $T(n)$ não é superior a $f(n)$; **$f(n)$ é um majorante de $T(n)$**

Exemplos:

$$f(n) = c_k n^k + c_{k-1} n^{k-1} + \dots + c_0 \rightarrow O(n^k) \quad (c_i - \text{constantes})$$

$$f(n) = \log_2 n \rightarrow O(\log n) \quad (\text{não se indica a base - a base é uma const.})$$

$$f(n) = 4 \rightarrow O(1) \quad (\text{usa-se 1 para ordem constante})$$

Propriedades

Se $F(n) = O(f(n))$ e $G(n) = O(g(n))$
então $F(n) + G(n) = O(\max(f(n), g(n)))$

Demonstração

Por hipótese existem n_1, n_2, c_1, c_2 tais que:

$$n \geq n_1 \rightarrow F(n) \leq c_1 f(n)$$

$$n \geq n_2 \rightarrow G(n) \leq c_2 g(n)$$

Sejam:

$$n_3 = \max(n_1, n_2)$$

$$c_3 = \max(c_1, c_2)$$

Então, para qualquer $n \geq n_3$:

$$\begin{aligned} F(n) + G(n) &\leq c_1 f(n) + c_2 g(n) \\ &\leq c_3 (f(n) + g(n)) \\ &\leq c_3 \max(f(n), g(n)) \end{aligned}$$

Propriedades

1. $O(f) + O(g) = O(f + g) = O(\max(f, g))$

Ex: $O(n^2) + O(\log n) = O(n^2)$

2. $O(f) \times O(g) = O(f \times g)$

Ex: $O(n^2) \times O(\log n) = O(n^2 \log n)$

3. $O(cf) = O(f)$ com c constante

Ex: $O(3n^2) = O(n^2)$

4. $F = O(f)$

Ex: $3n^2 + \log n = O(3n^2 + \log n)$

5. $O(n^g) < O(n^{g+k})$

Exemplo

$$\begin{aligned} 3n^2 + \log n &= O(3n^2 + \log n) && \text{por 4.} \\ &= O(3n^2) && \text{por 1.} \\ &= O(n^2) && \text{por 3.} \end{aligned}$$

Análise de Complexidade

Exemplo: Função que executa o somatório $\rightarrow \sum_{i=0}^n i^3$

		Unidades de Tempo
1	<code>int soma_cubos (int n)</code>	
2	<code>{ int somaparcial ;</code>	
3	<code>somaparcial = 0 ;</code>	1
4	<code>for (int i = 1 ; i <= n ; i++)</code>	$1 + (n+1) + n$
5	<code>somaparcial += i * i * i ;</code>	$4n$
6	<code>return somaparcial ;</code>	1
	<code>}</code>	$6n + 4$

$T(n) = O(n)$ Linear

Análise de Complexidade

Na prática, é difícil (senão impossível) prever com rigor o tempo de execução de um algoritmo ou programa

- obtém-se o tempo a menos de constantes multiplicativas e de parcelas menos significativas (pelo menos para n grande)
- identificam-se uma ou mais operações-chave (operações mais frequentes ou muito mais demoradas) e determina-se o nº de vezes que são executadas

Exemplo: num **algoritmo de ordenação**, uma operação fundamental, natural é a **comparação entre elementos** quanto à ordem

Análise de Complexidade

Exemplo: Determinar a quantidade de 1s existentes na representação binária de um número n

1	1
2	10
...	
4	100
...	
7	111
...	
16	10000
...	
32	100000

1	<i>ler(n)</i>	1	t1
2	<i>conta ← 0</i>	1	
3	<u>Enq</u> ^{to} <i>n > 0</i> <u>fazer</u>	k+1	t2
4	<i>conta ← conta + n mod 2</i>	k	t3
5	<i>n ← n / 2</i>	k	

Tempo total para execução do algoritmo:

$$T(n) = t1 + (k + 1) \times t2 + k \times t3$$

$$T(n) = (t1 + t2) + (t2 + t3) \times k$$

em que k é o número de iterações do ciclo

Análise de Complexidade

É necessário relacionar k (nº de iterações) com n

n		k	Função
1	1	1	$1 + \log_2 1$
2	10	2	$1 + \log_2 2$
4	100	4	
8	1000	8	
16	10000	16	
32	100000	32	$1 + \log_2 32$

A operação divisões por 2 é inversa da operação potência de base 2:

função inversa potência de base 2
 $\rightarrow \log_2 n$

Substituindo $k = 1 + \log_2 n$

$$T(n) = (t_1 + t_2) + (t_2 + t_3) \times k$$

$$T(n) < (t_1 + t_2) + (t_2 + t_3) \times (1 + \log_2 n)$$

$$T(n) < C \times (1 + \log_2 n)$$

$T(n) = O(\log n)$ Logarítmica

Na Prática

Todas as operações simples (teste, afectação, etc.) custam 1 unidade de tempo

Para calcular a ordem de grandeza da complexidade, ignoram-se as constantes

Afectações

Comparações

Operações aritméticas

Leituras e Escritas

Chamadas de funções

Retornos de funções

Acessos a campos de vectores

Acessos a campos de registos

$O(1)$

switch E

case V1 : S1

...

case Vj : Sj

$O(\max(T_E, T_{S1}, \dots, T_{Sj}))$

Na Prática

if C then S1 else S2 $O(\max(TC, TS1, TS2))$
for do S $O(n T(S))$
while C do S $O(n \max(T_C, T_S))$
do S while C $O(n \max(T_C, T_S))$ n é o nº iterações do ciclo

Ciclos aninhados

Para $i = 1$ até n
 Para $j = 1$ até n
 $k++$

} $n \times n$ ou $O(n^2)$ Os ciclos exteriores têm efeito multiplicativo sobre as operações no ciclo interior

Ciclos paralelos

Para $i = 1$ até n $O(n)$
 $A[i] = 0$;
Para $i = 1$ até n $O(n^2)$
 Para $j = 1$ até n
 $k++$

} pela Propriedade 1 $O(n^2)$

Na Prática

$h = 1;$

Enq^{to} $h \leq n$

{ $s;$
 $h = h * 2;$ }

h toma os valores 1, 2, 4, ..., n

$\hookrightarrow 1 + \log_2 n \rightarrow \mathbf{O(\log n)}$

Quando o 2º ciclo depende do ciclo exterior

Para $i = 1$ até n

Para $j = 1$ até i
 $k++$

O ciclo interno é executado i vezes, pelo que o tempo total é

$$\sum_{i=1}^n i = \frac{n(n+1)}{2} \approx \frac{n^2}{2}$$

$\hookrightarrow \mathbf{O(n^2)}$

$h = n;$

Enq^{to} $h > 0$

{

Para $i = 1$ até n

$k++;$

$h = h / 2;$ }

o ciclo externo é executado $\log_2 n$ vezes

o ciclo interno n vezes

$\hookrightarrow \mathbf{O(n \log n)}$

Algoritmos determinísticos vs. não determinísticos

Os **algoritmos determinísticos**, são aqueles para os quais não é preciso atender à forma de distribuição dos dados para obter o seu tempo de execução.

Exemplos:

- média dos elementos de um vector
- máximo de um vector
- multiplicação de duas matrizes
- ...

Estes algoritmos independentemente dos valores contidos no vector apresentarão sempre a mesma complexidade

Os **algoritmos não determinísticos**, a sua complexidade depende da forma de distribuição dos dados

Exemplos:

- pesquisa de um valor num vector
- ordenação dos valores de um vector
- ...

Os principais critérios para a determinação da complexidade dos algoritmos **não determinísticos** são: o **melhor caso**, o **caso médio** e o **pior caso**

Algoritmos de Pesquisa

Pesquisa Sequencial

Problema (*pesquisa de valor num vector*): verificar se um valor existe num vector e, no caso de existir, indicar a sua posição

int	<i>Pesq_Sequencial</i> (T v[], int n, T x)
1	$i \leftarrow 0$; $enc \leftarrow False$
2	<u>Enq</u> ^{to} $i < n$ <u>fazer</u>
3	<u>se</u> (v[i] = x) <u>então</u> devolve i <u>senão</u> $i \leftarrow i + 1$
4	devolve -1

Pesquisa Sequencial

Operação fundamental:

- comparação (instr. se linha 3)

A complexidade do algoritmo vai depender da posição em que x se encontrar:

1ª posição → 1 comparação

2ª posição → 2 comparações

.....

nª posição → n comparações

Complexidade Pior caso → $O(n)$

Complexidade Média : Se **x** existir no vector, o teste é realizado aproximadamente **$n/2$** vezes em média (1 vez no melhor caso)

$$\frac{1}{2} \times (1 + 2 + \dots + n) = \frac{(1 + n)}{2} \rightarrow O(n)$$

$T(n) = O(n)$ Linear no pior caso e no caso médio

Complexidade Espacial

Complexidade espacial de um programa é dada pela função que define o comportamento do programa relacionando o espaço de memória ocupado com o volume de dados a manipular

Intervêm neste espaço os seguintes componentes:

Espaço de Instruções: espaço necessário para guardar a versão compilada das instruções do programa. Este espaço é constante para um dado programa referente a um dado compilador, com determinadas opções de compilação → **Este espaço não interessa analisar**

Espaços que variam com o algoritmo usado:

- **Espaço de Dados**: espaço onde estão definidas as variáveis globais e que varia de acordo com os dados manipulados pelo problema
- **Espaço de Stack**: usado para guardar a informação necessária para resumir a execução de funções.

Na stack são guardados os endereços:

- ♦ de retorno das funções
- ♦ variáveis locais
- ♦ parâmetros formais

Complexidade Espacial

Pesquisa Sequencial

Espaço total :

endereço de retorno	2 bytes
apontador v	2 bytes
apontador para parâmetro actual de x	2 bytes
valor do parâmetro formal n	2 bytes
variável local i	2 bytes
variável local enc	2 bytes
	<hr/>
	12 bytes

Assim esta função tem uma Complexidade espacial constante

$S(n) = O(1)$ em qualquer caso

Eficiência da Pesquisa Sequencial

- Eficiência temporal da ***Pesq_Sequencial***

- A operação realizada mais vezes é o teste da condição de continuação do ciclo **for**, no máximo **$n+1$** vezes (no caso de não encontrar **x**).
- Se **x** existir no vector, o teste é realizado aproximadamente **$n/2$** vezes em média (1 vez no melhor caso)

⇒ **$T(n) = O(n)$ (linear) no pior caso e no caso médio**

- Eficiência espacial da ***Pesq_Sequencial***

- Gasta o espaço das variáveis locais (incluindo argumentos)
- Como o array é passado "por referência" (de facto o que é passado é o endereço do array), o espaço gasto pelas variáveis locais é constante e independente do tamanho do array

⇒ **$S(n) = O(1)$ (constante) em qualquer caso**

O que analisar ?

O recurso mais importante a analisar é o **tempo de execução**

Vários factores afectam o **tempo de execução**:

- Computador
 - Compilador
 - Algoritmo usado
 - Tamanho dos dados de entrada do algoritmo (N)
- fora do âmbito de qualquer modelo teórico

Em algoritmos não determinísticos:

- $T_{\text{med}}(N)$ – representa um comportamento típico
- $T_{\text{pior}}(N)$ – garante-nos o desempenho do algoritmo para qualquer input

$T_{\text{med}}(N) \leq T_{\text{pior}}(N)$ na maioria dos casos segue o pior caso

Uma vez que estamos interessados num limite superior – **Complexidade Big-Oh** – basta apenas calcular a complexidade para o **Pior Caso** e **Melhor Caso**

Pesquisa Binária

Verificar se um valor (x) existe num vector (v) previamente ordenado e, no caso de existir, indicar a sua posição

Algoritmo

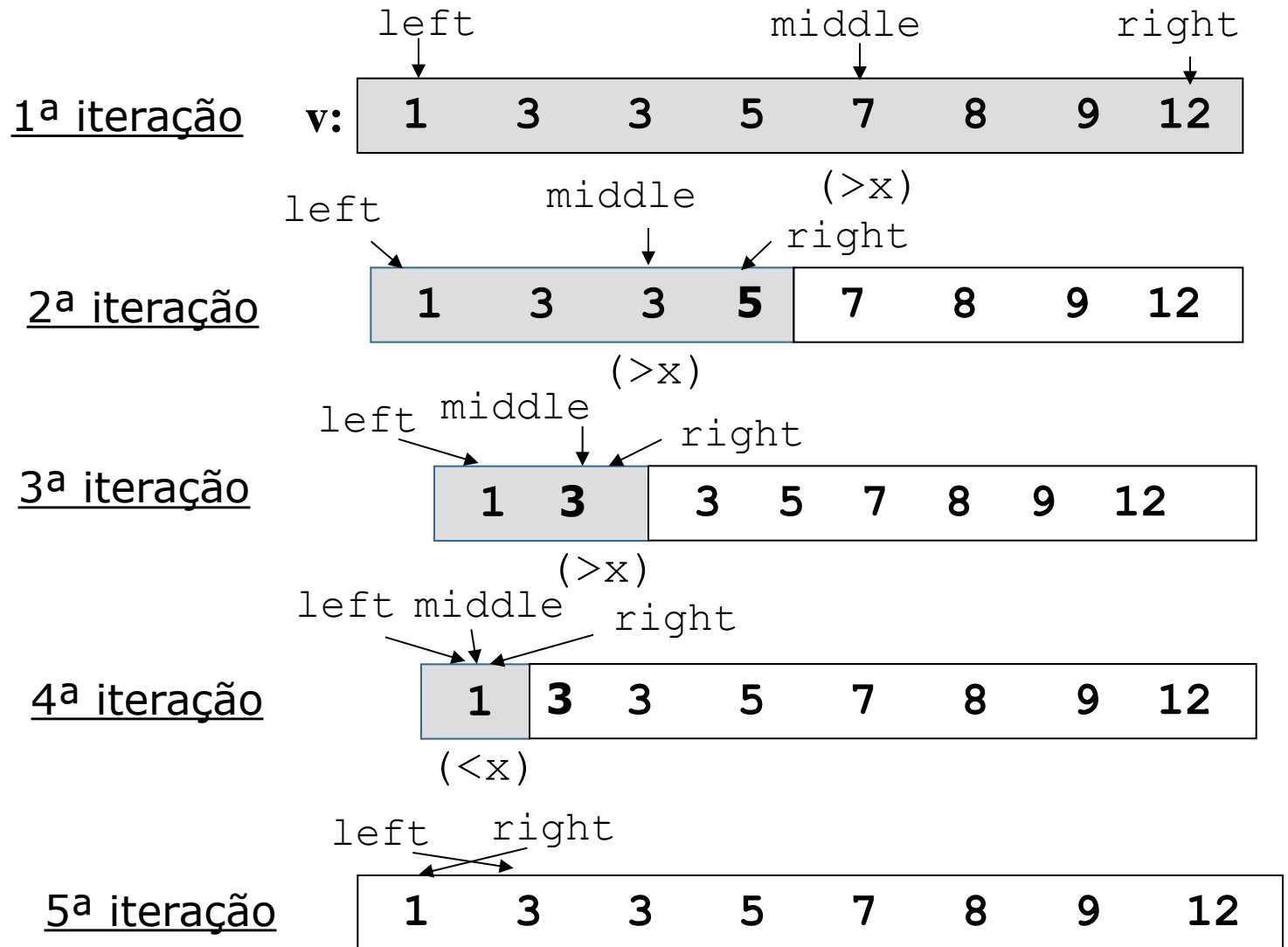
```
inf ← 1
sup ← n
Enqto inf <= sup fazer
    meio ← (inf+sup)/2
    se v[meio] = x então
        devolve meio
    senão
        se v[meio] < x então
            inf ← meio+1
        senão
            sup ← meio-1
devolve -1
```

Codificação em C++

```
template <class T>
int BinarySearch(const T v[], int n, T x)
{
    int left = 0, right = n - 1;
    while (left <= right)
    {
        int middle = (left+right)/2;
        if (x == v[middle])
            return middle;
        else if (x > v[middle])
            left = middle + 1;
        else
            right = middle - 1;
    }
    return -1;
}
```

Exemplo de Pesquisa Binária

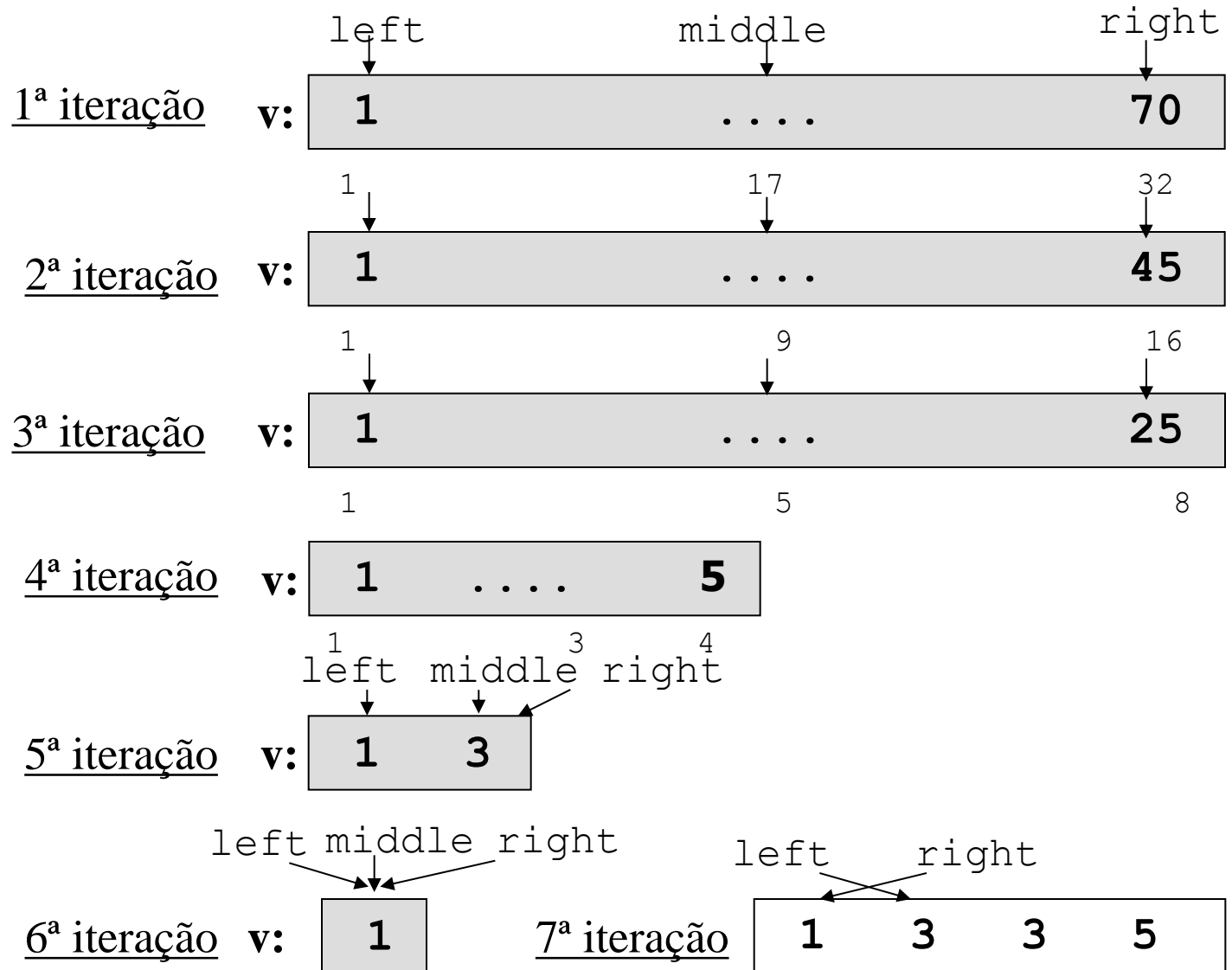
x: 2



vector a inspeccionar vazio \Rightarrow o valor 2 não existe no vector inicial!

Exemplo de Pesquisa Binária

x: 2



Eficiência Temporal da Pesquisa Binária

Operação fundamental: $\text{meio} \leftarrow (\text{inf} + \text{sup})/2$ (instr. linha 3)

A complexidade do algoritmo vai depender do número de elementos do vector (n) e da posição onde x se encontra

Pior Caso: x não se encontra no vector

n	k
32	7
16	6
8	5
4	4
2	3
...	...

Em cada iteração, o tamanho do sub-vector a analisar é dividido por um factor de aproximadamente 2

Ao fim de k iterações, o tamanho do sub-vector a analisar é aproximadamente $n / 2^k$

Se não existir no vector o valor procurado, o ciclo só termina quando:

$$n / 2^k \approx 1 \Leftrightarrow \log_2 n - k \approx 0 \Leftrightarrow k \approx \log_2 n$$

$$2 + \log_2 n$$

Pior caso, o nº de iterações é aproximadamente **$\log_2 n$**

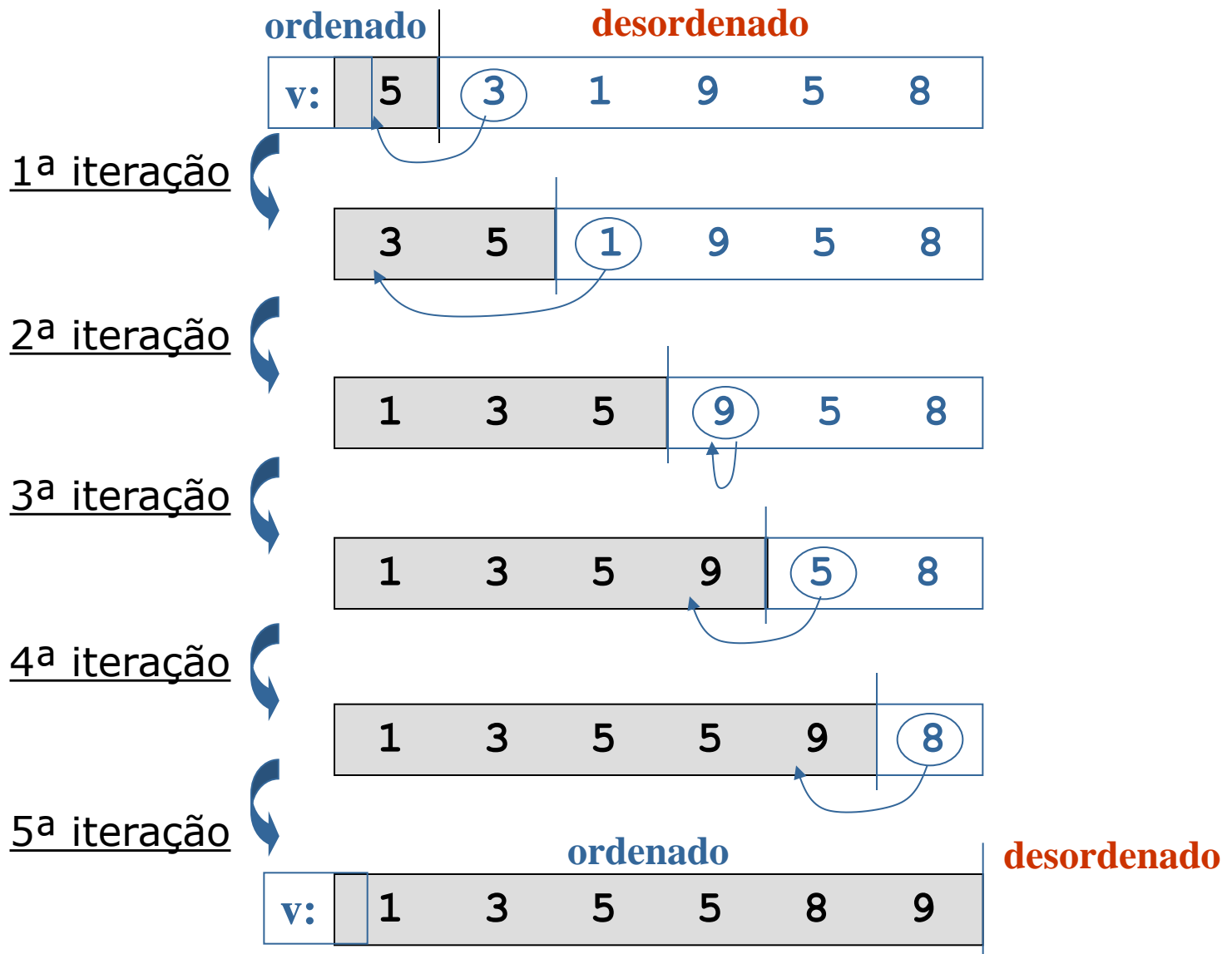
$\Rightarrow T(n) = \mathbf{O(\log n)}$ **Logarítmico**

Algoritmo de Ordenação

Ordenação por Inserção

- Problema (*ordenação de vector*): reorganizar os n elementos de um *vector* (v) por ordem crescente
 - ou melhor, por ordem não decrescente, porque podem existir valores repetidos
- Algoritmo (*ordenação por inserção*):
 - Considera-se o vector dividido em dois sub-vectores (esquerdo e direito), com o da esquerda ordenado e o da direita desordenado
 - Começa-se com um elemento apenas no sub-vector da esquerda
 - Move-se um elemento de cada vez do sub-vector da direita para o sub-vector da esquerda, inserindo-o na posição correcta por forma a manter o sub-vector da esquerda ordenado
 - Termina-se quando o sub-vector da direita fica vazio

Exemplo de Ordenação por Inserção



Ordenação por Inserção

Algoritmo (*ordenação por inserção*):

$i \leftarrow 1$

Enq^{to} $i < n$ fazer

$j \leftarrow i$

$x \leftarrow v[i]$

 Enq^{to} $j > 0 \wedge x < v[j-1]$ fazer

$v[j] \leftarrow v[j-1]$

$j \leftarrow j-1$

$v[j] \leftarrow x$

$i \leftarrow i + 1$

Sub-problema:

Inserção de um valor num vector ordenado mantendo-o ordenado

Implementação da Ordenação por Inserção em C++

```
template <class T>
void InsertSorted(T v[], int n, T x)
{
    for (int j = n; j > 0 && x < v[j-1]; j--)
        v[j] = v[j-1];
    v[j] = x;
}

// Ordena vector v de n elementos, ficando v[0] ≤ ... ≤ v[n-1]
template <class T>
void InsertionSort(T v[], int n)
{
    for (int i = 1; i < n; i++)
        InsertSorted(v, i, v[i]);
}
```

OU

```
template <class T>
void InsertionOrd(T v[], int n)
{
    for (int i = 1; i < n; i++)
    {
        T x = v[i];
        for (int j = i; j > 0 && x < v[j-1]; j--)
            v[j] = v[j-1];
        v[j] = x;
    }
}
```

Eficiência da Ordenação por Inserção

O nº de iterações do **InsertSorted** (**v**, **n**, **x**) é:

- **Pior caso:** **n** (inserir um valor menor do que todos os que existem no vector)
- **Melhor caso:** 1 (inserir um valor maior do que todos os que existem no vector)
- **Média:** **n/2**

O nº de iterações do **InsertionSort** (**v**, **n**):

- faz InsertSorted (,1,), InsertSorted (,2,), ..., InsertSorted (,n-1,)
- o nº total de iterações do ciclo for de InsertSorted é:
 - ♦ **Melhor caso:** $1 + 1 + \dots + 1$ ($n-1$ vezes) $= n-1 \approx \mathbf{n}$
 - ♦ **Pior caso:** $1 + 2 + \dots + n-1 = (n-1)(1+ n-1)/2 = n(n-1)/2 \approx \mathbf{n^2/2}$
 - ♦ **Média,** metade do anterior, isto é, aproximadamente **n²/4**

Melhor Caso: **T(n) = O(n)** Linear

Pior Caso e Caso Médio: **T(n) = O(n²)** Quadrático

Análise de Complexidade

Funções Recursivas

Funções Recursivas

Os problemas que podem ser resolvidos recursivamente têm normalmente as seguintes características:

- um ou mais casos de paragem, em que a solução é não recursiva e conhecida
- casos em que o problema pode ser diminuído recursivamente até se atingirem os casos de paragem

Estrutura de um Algoritmo Recursivo

se caso de paragem atingido então

resolver o problema

senão

fazer uma ou mais invocações recursivas

Custo da Recursividade


A invocação de uma função recursiva produz um **desperdício de tempo e de memória** devido:

- à criação de uma cópia local dos parâmetros de entrada que são passados por valor
- à recolha do endereço dos parâmetros passados por referência
- espaço para guardar variáveis locais
- bem como a salvaguarda do estado do programa na altura da invocação - memória stack - para que o programa possa retomar a execução na instrução seguinte à invocação da função, quando a execução da função terminar

Complexidade Funções Recursivas

Em funções recursivas a complexidade é determinada:

- pelo número de chamadas recursivas
- pela complexidade das operações que acarretam cada chamada

```
void metodo_recursivo (...)  
{  
      
    metodo_recursivo (...);  
}
```

Exemplo Factorial

```
long factiter (long num)
{ long res=1 ;
  for (int i = 1; i <= num; i++)
    res *= i ;
  return res ; }
```

$T(n) = O(n)$ Linear

$S(n) = 1$

```
long factrecurs (long &num)
{ if (num == 1)
  return 1 ;
else
  return num * factrecurs(num-1) ; }
```

$$\text{numCR} = \begin{cases} 0 & n \leq 1 \\ 1 + \text{numCR}(n-1) & n \geq 2 \end{cases}$$

$T(n) = O(n)$ Linear

$S(n) = O(n)$

Fibonnaci Interactivo

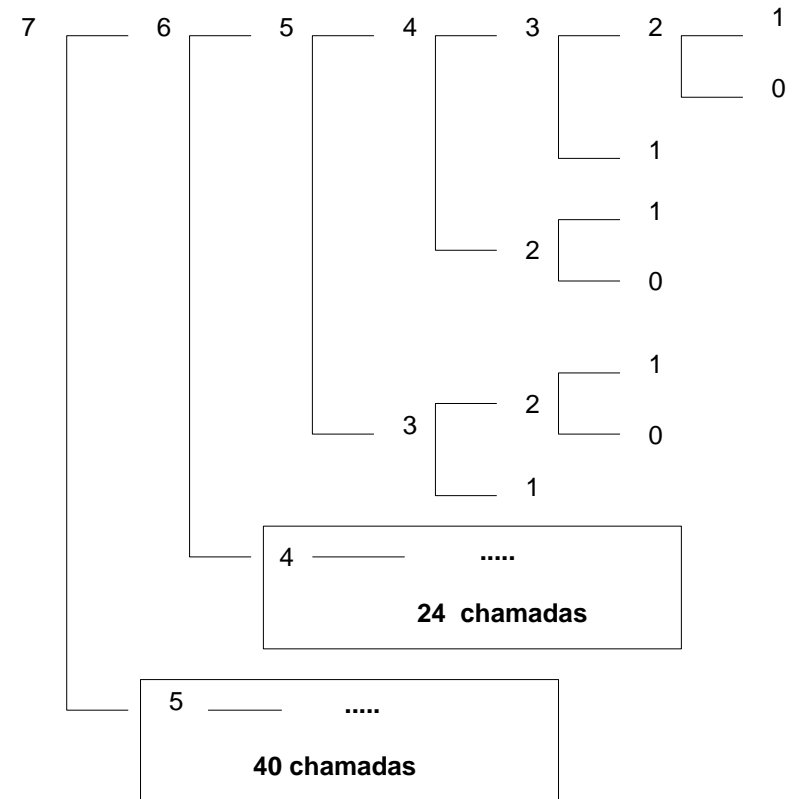
```
int fib_iter (int n)
{
    if (n == 0 || n == 1)
        return n;

    segano=0;
    ano=1;
    for (int i = 2; i <= n; i++)
    {
        corrente = segano + ano;
        segano = ano;
        ano = corrente;
    }
    return corrente;
}
```

$T(n) = O(n)$ Linear

Fibonnaci recursivo

```
int fib (int n)
{
    if (n ≤ 1)
        return n ;
    else
        return fib(n-1) + fib(n-2);
}
```



Número de Chamadas recursivas:

$$\text{numCR}(n) = \begin{cases} 0 & n = 0 \\ 0 & n = 1 \\ \text{numCR}(n-1) + \text{numCR}(n-2) + 1 & n \geq 2 \end{cases}$$

Fibonnaci recursivo

n	k	Função
2	3	$\leq 2^{n-1} + 1$
3	5	$\leq 2^{n-1} + 1$
4	9	$\leq 2^{n-1} + 1$
5	15	$\leq 2^{n-1} + 1$
6	24	$\leq 2^{n-1} + 1$
....	...	$\leq O(2^{n-1})$

$T(n) = O(2^n)$ Exponencial

Potência x^n (Versão iterativa)

```
double potencia (double x, int n)
{
    double pot = 1.0;

    for ( ; n > 0; n--)
        pot *= x;

    for ( ; n < 0; n++)
        pot /= x;

    return pot;
}
```

$T(n) = O(n)$ Linear

Potência x^n (versão recursiva)

```
double potenc (const double x, const int& n)
{
    if (n == 0)
        return 1 ;
    else
        return x * potenc(x,n-1) ;
}
```

Número de Chamadas Recursivas

n	k
0	0
1	1
2	2
...	
15	14
n	n-1

$T(n) = O(n)$ Linear

Potência x^n (outra versão recursiva)

```
double potenc (const double x, const int& n)
{ if (n == 0)
    return 1 ;
  if (n == 1)
    return x ;
  if (n % 2 == 0)
    return potenc (x * x, n/2) ;
  else
    return x * potenc (x * x, n/2) ; }
```

$$2^{16} = (2*2)^8 = (4*4)^4 = (16 * 16)^2 = (256*256)^1$$

↪ **5 chamadas recursivas**

$$2^{15} = 2*(2*2)^7 = 2*4*(4*4)^3 = 2*4*16*(16*16)^1$$

↪ **4 chamadas recursivas**

Potência x^n (outra versão recursiva)

Número de Chamadas Recursivas

n (par)	k
0	1
1	1
2	2
4	3
8	4
16	5
64	7
...	...
	$1 + \log_2 n$

n (ímpar)	k
3	2
5	3
7	3
9	4
...	...
	$1 + \log_2 n$

$$T(n) = O(\log n)$$

Logarítmica

Torres de Hanói

Torres-Hanoi (N, TorreA, TorreB, TorreC)

Se (N = 1)

Mover disco TorreA → TorreB

Senão

Torres-Hanoi (N-1, TorreA, TorreC, TorreB)

Torres-Hanoi (1, TorreA, TorreB, TorreC)

Torres-Hanoi (N-1, TorreC, TorreB, TorreA)

n (nº discos)	K (nº movimentos)
1	1
2	3
4	15
8	63
16	...
...	...
	$2^n - 1$

$T(n) = O(2^n)$ Exponencial

Algoritmos de Ordenação

Dividir-e-Conquistar

- Dividir-e-Conquistar é um paradigma genérico de desenho de algoritmos:
 - **Dividir**: dividir os dados de entrada S em dois subconjuntos disjuntos $S1$ e $S2$
 - **Recorrência**: resolver os subproblemas associados com $S1$ e $S2$
 - **Conquistar**: combinar as soluções $S1$ e $S2$ numa solução final S
- O caso base para a recursividade são subproblemas de tamanho 0 ou 1

Algoritmo MergeSort

MergeSort é um algoritmo de ordenação baseado no paradigma Dividir-e-Conquistar

MergeSort recebe uma sequência de entrada S com n elementos e consiste em três etapas:

- **Divide**: divide S em duas sequências $S1$ e $S2$ de cerca de $n/2$ elementos cada
- **Recorrência**: recursivamente ordena $S1$ e $S2$
- **Conquistar**: une $S1$ e $S2$ numa sequência ordenada única

Algoritmo MergeSort

Input: Sequência S com n elementos

Output: Sequência S ordenada

```
mergeSort(S)
```

```
  Se S.size() > 1
```

```
    S1 ← partition (S, n/2) //subvector esq.
```

```
    S2 ← partition (S, n/2) //subvector dir.
```

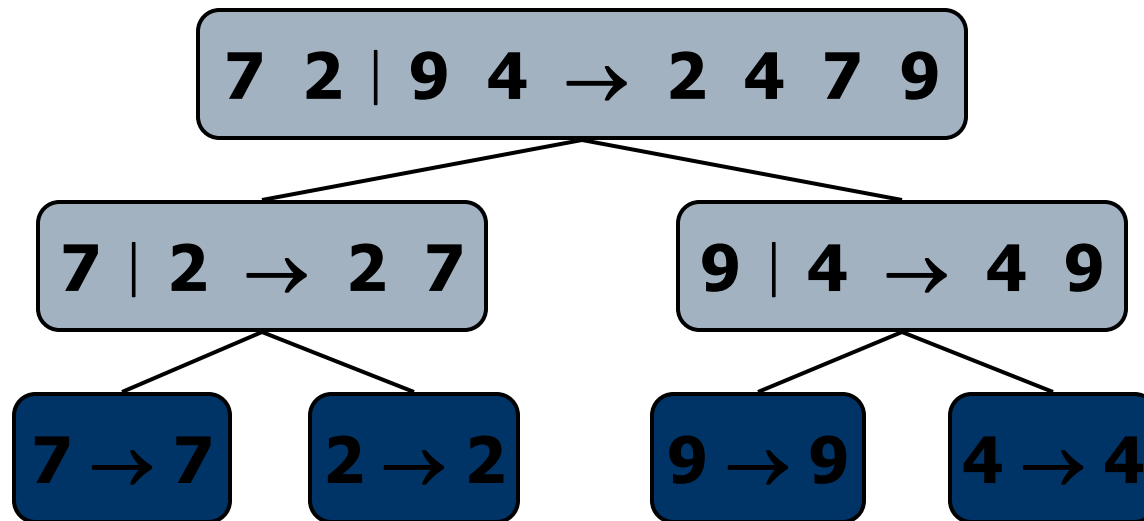
```
    mergeSort(S1)
```

```
    mergeSort(S2)
```

```
    S ← merge(S1, S2)
```

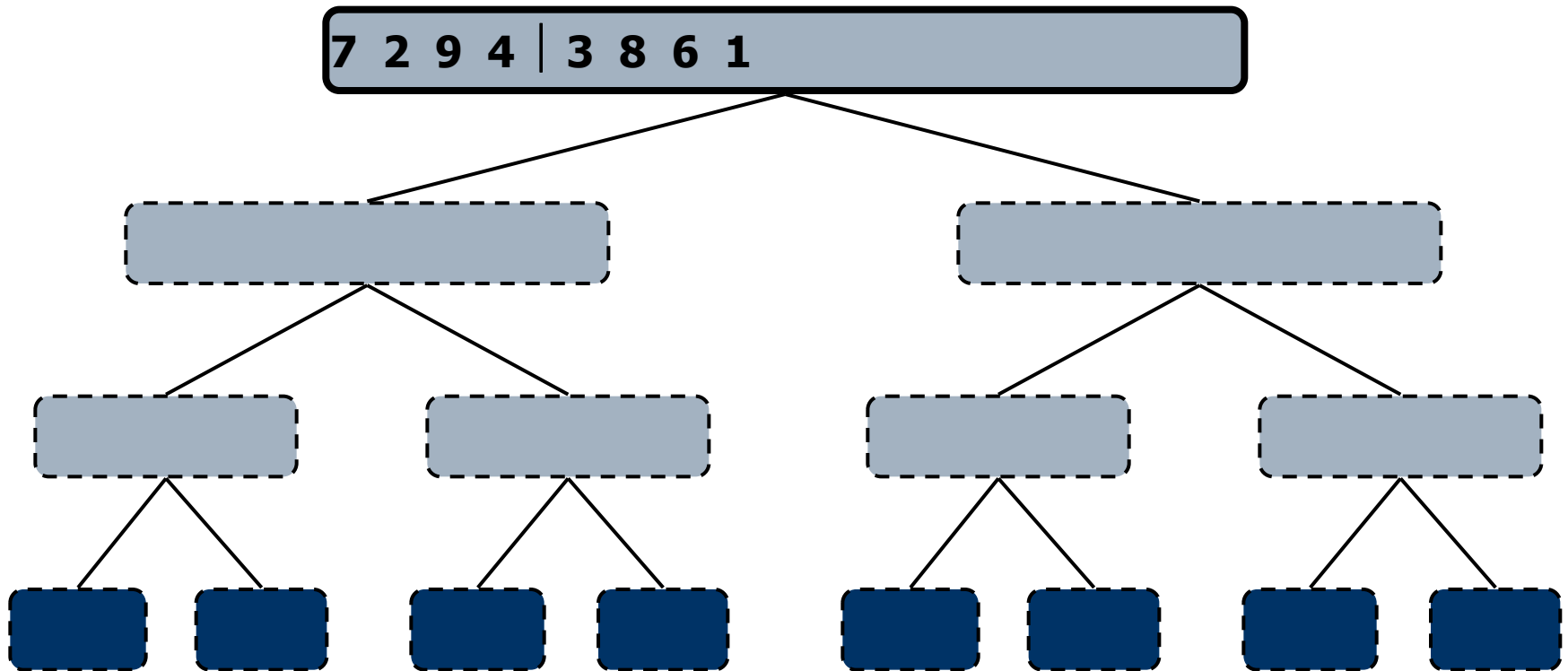

Árvore MergeSort

- A execução do mergeSort pode ser descrita através de uma árvore binária
- Cada nó representa uma chamada recursiva do mergeSort e apresenta a sequência não ordenada antes da execução e a respectiva partição ordenada no final da execução
- a raiz é a chamada inicial
- as folhas são chamadas de subsequências de tamanho 0 ou 1



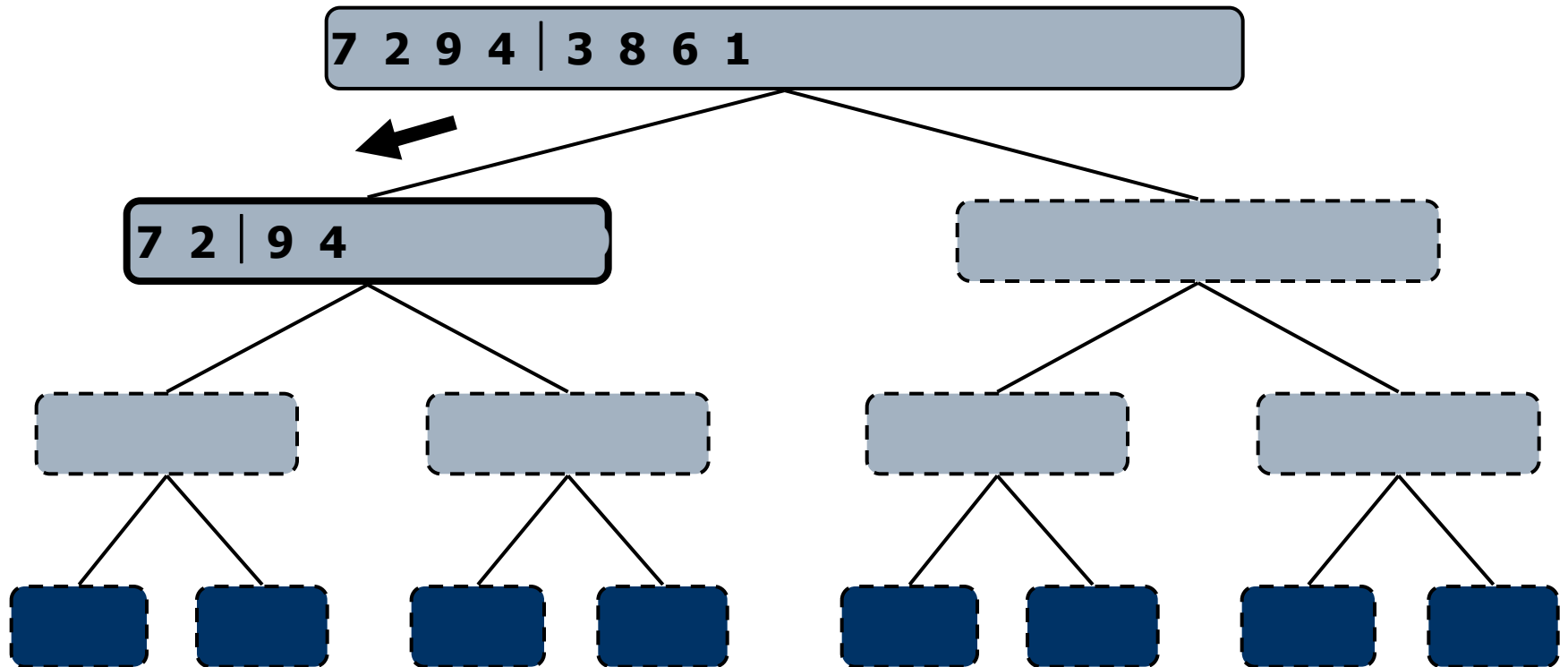
Exemplo

- Partição



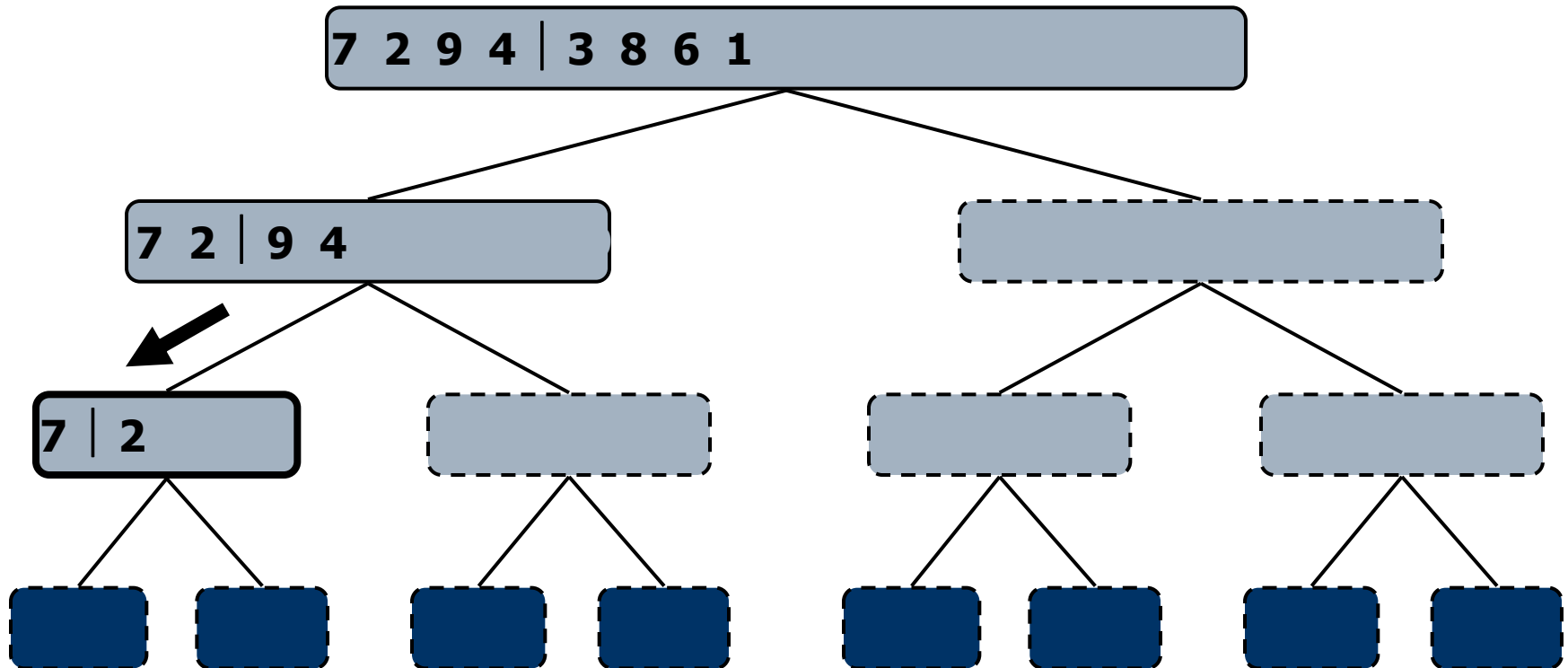
Exemplo (cont.)

- Chamada recursiva, partição



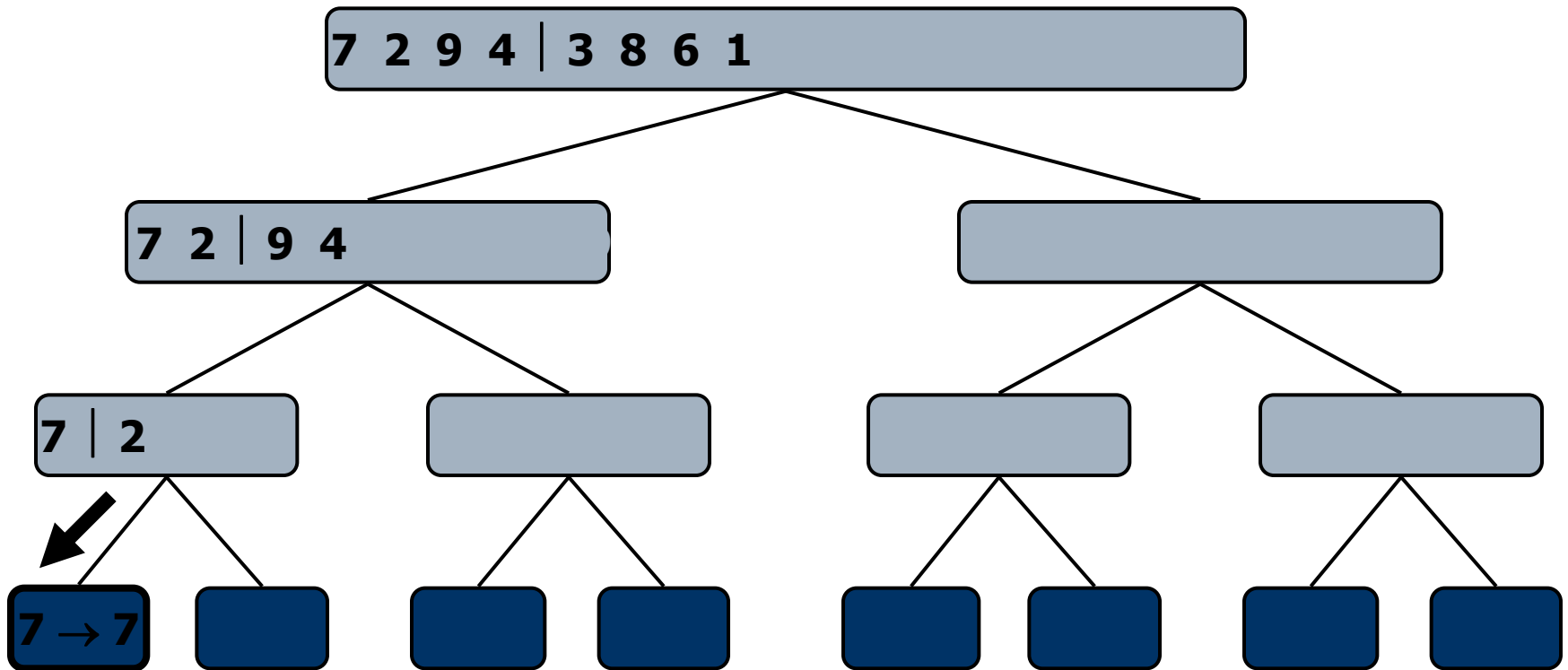
Exemplo (cont.)

- Chamada Recursiva, partição



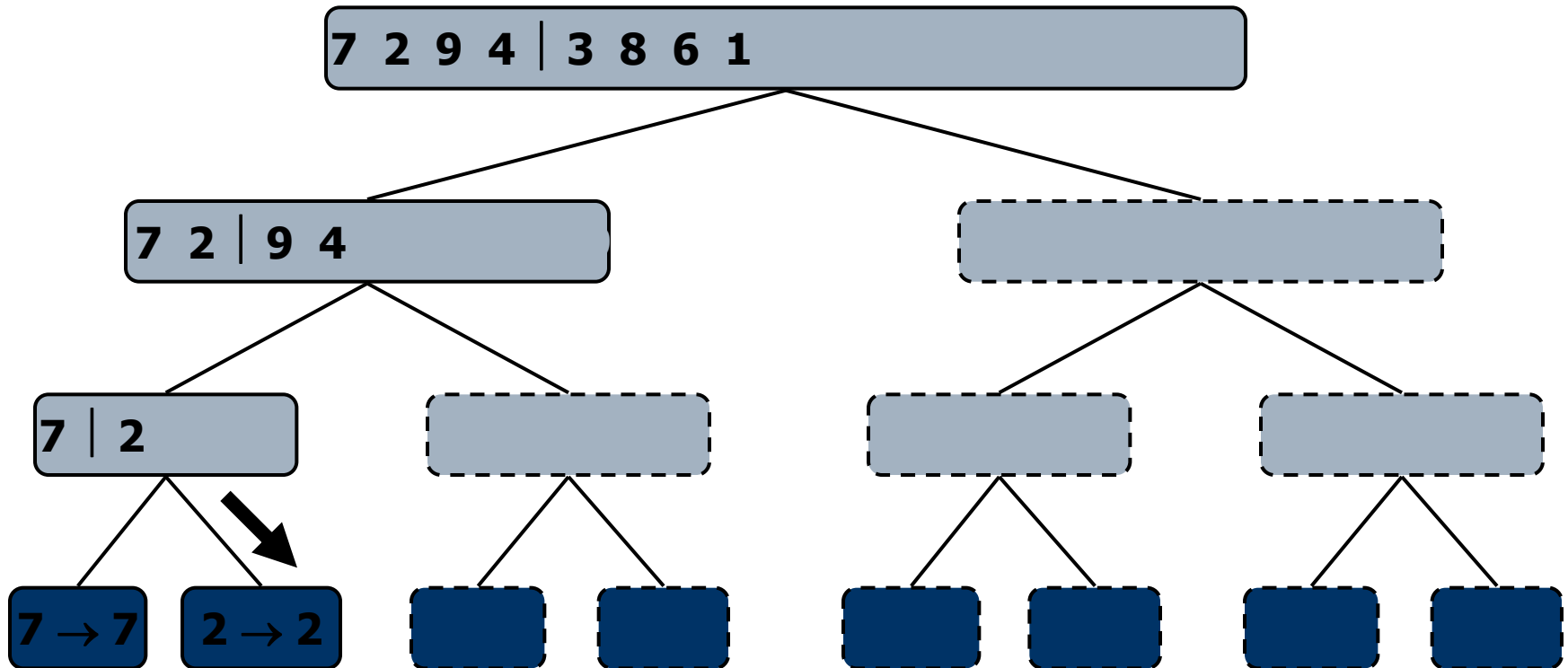
Exemplo (cont.)

- Chamada Recursiva, caso base



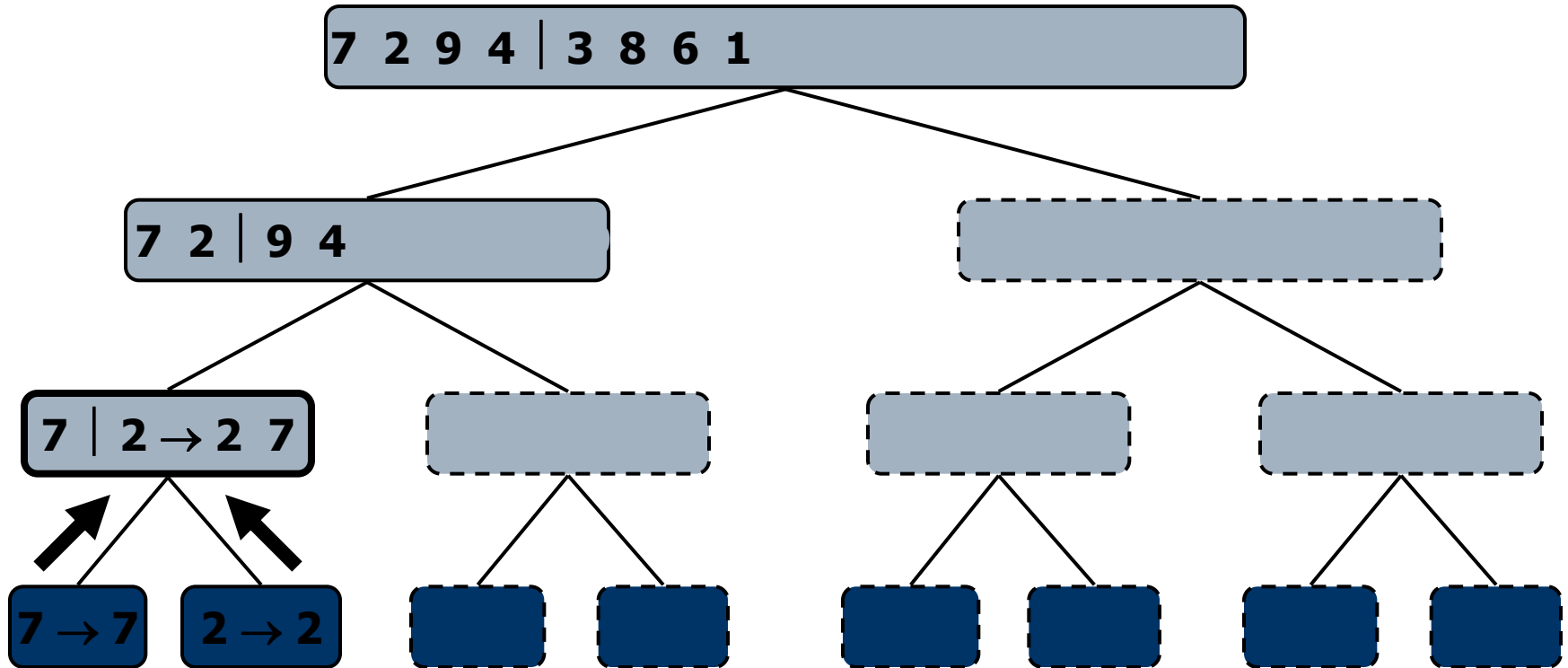
Exemplo (cont.)

- Chamada Recursiva, caso base



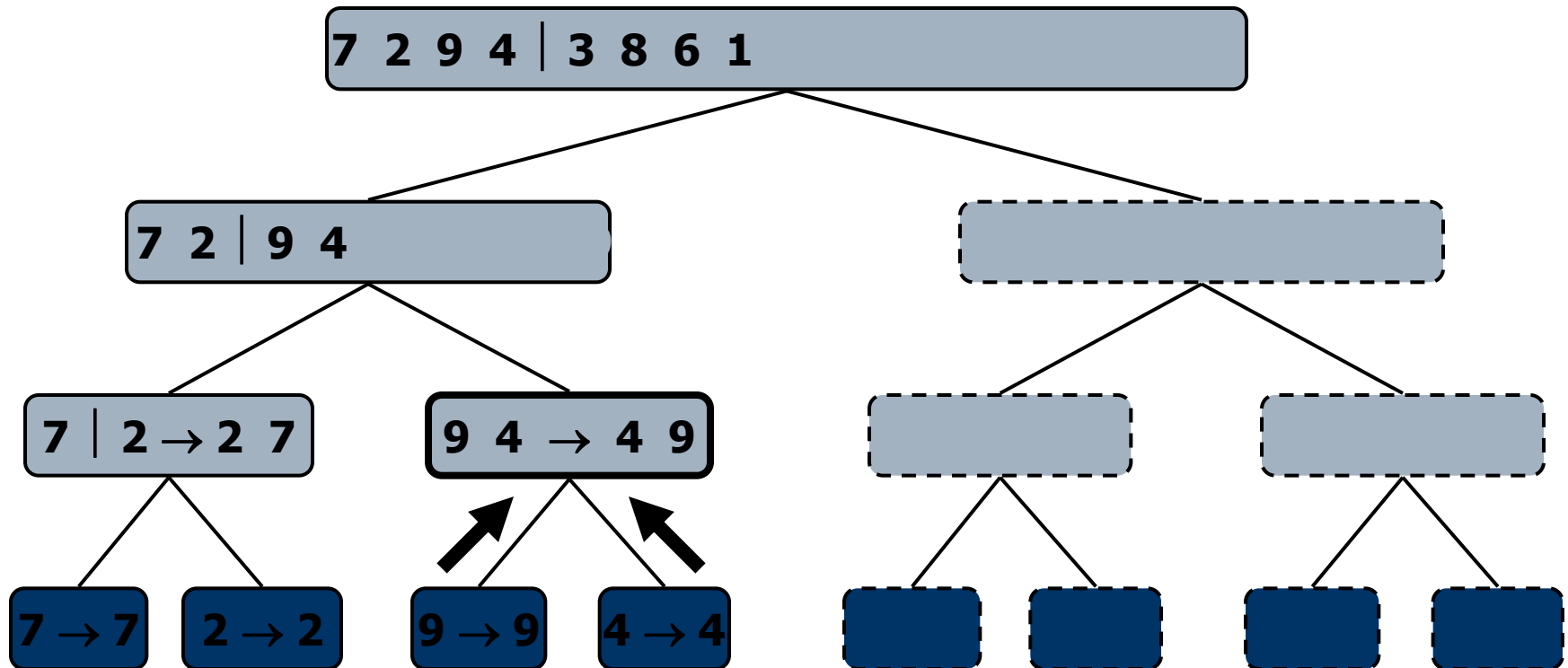
Exemplo (cont.)

- Une



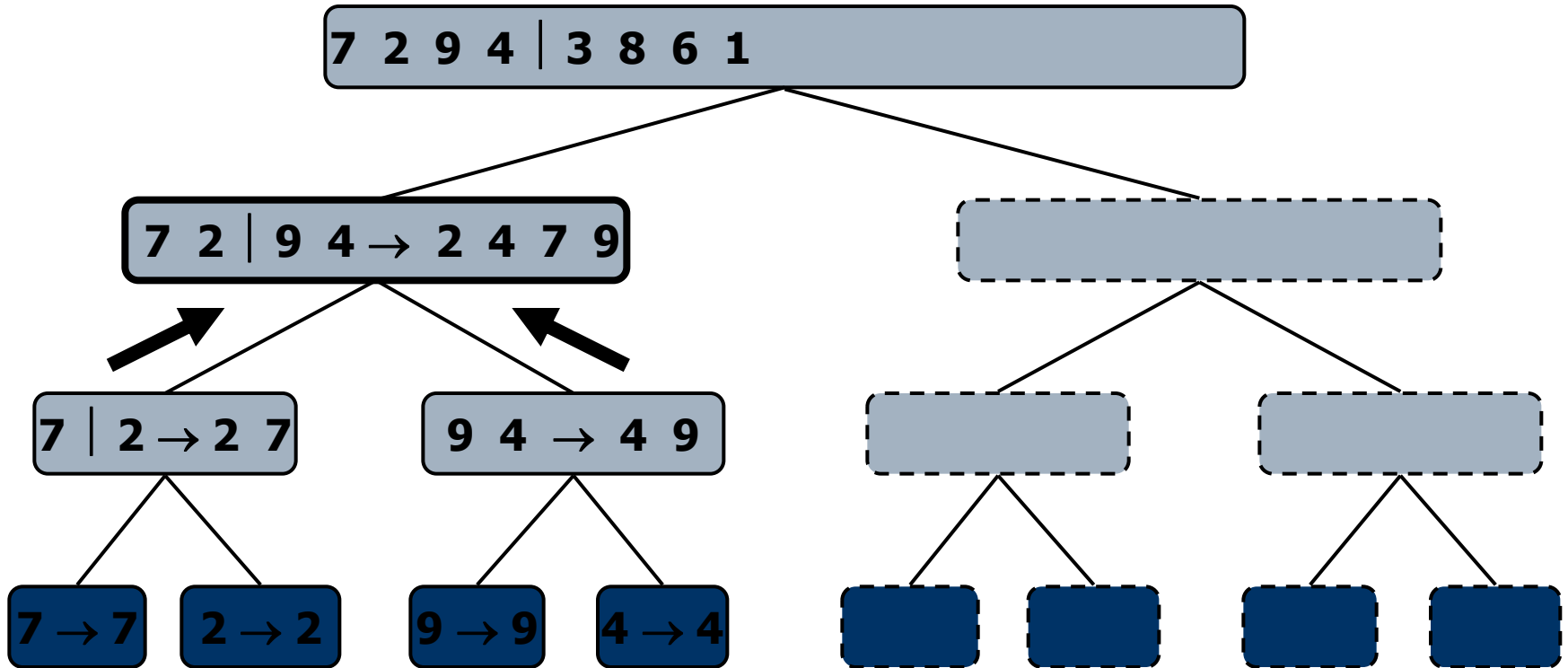
Exemplo (cont.)

- Chamada Recursiva, ..., caso base, une



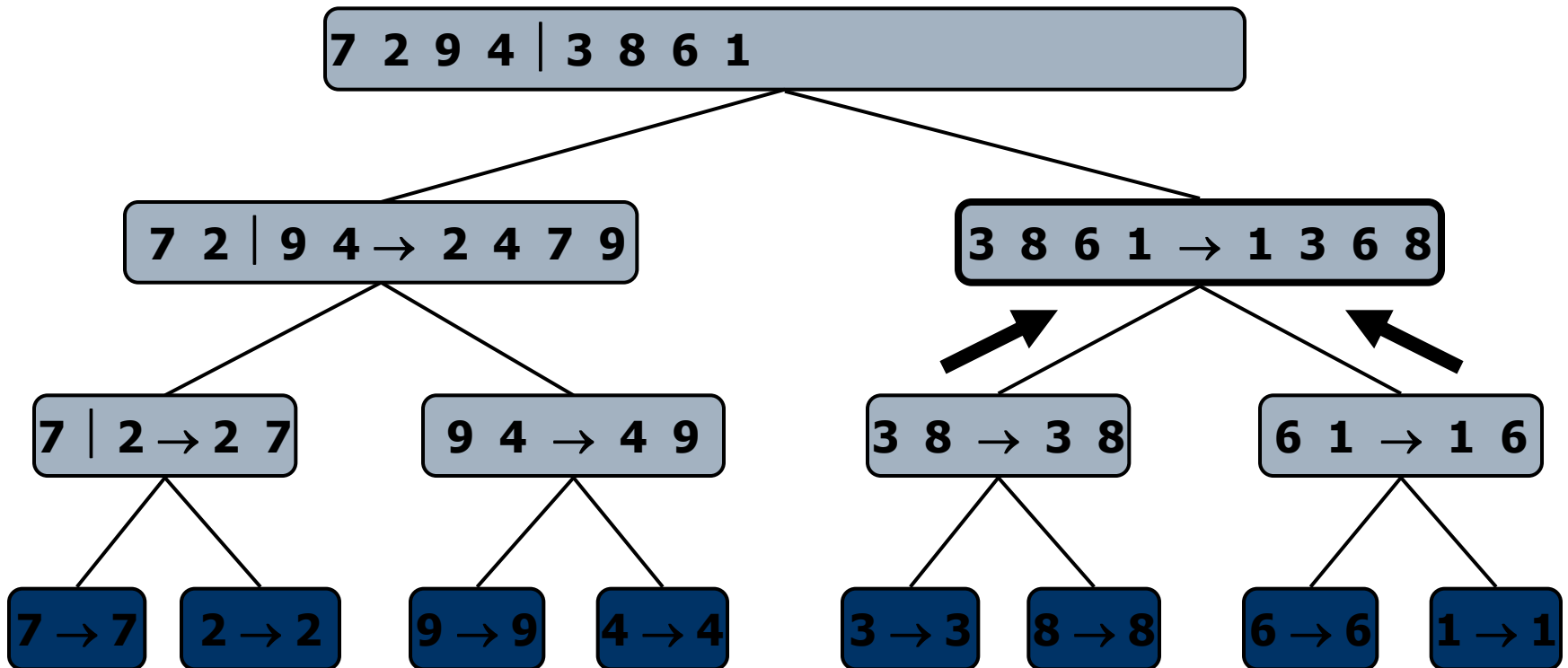
Exemplo (cont.)

- Une



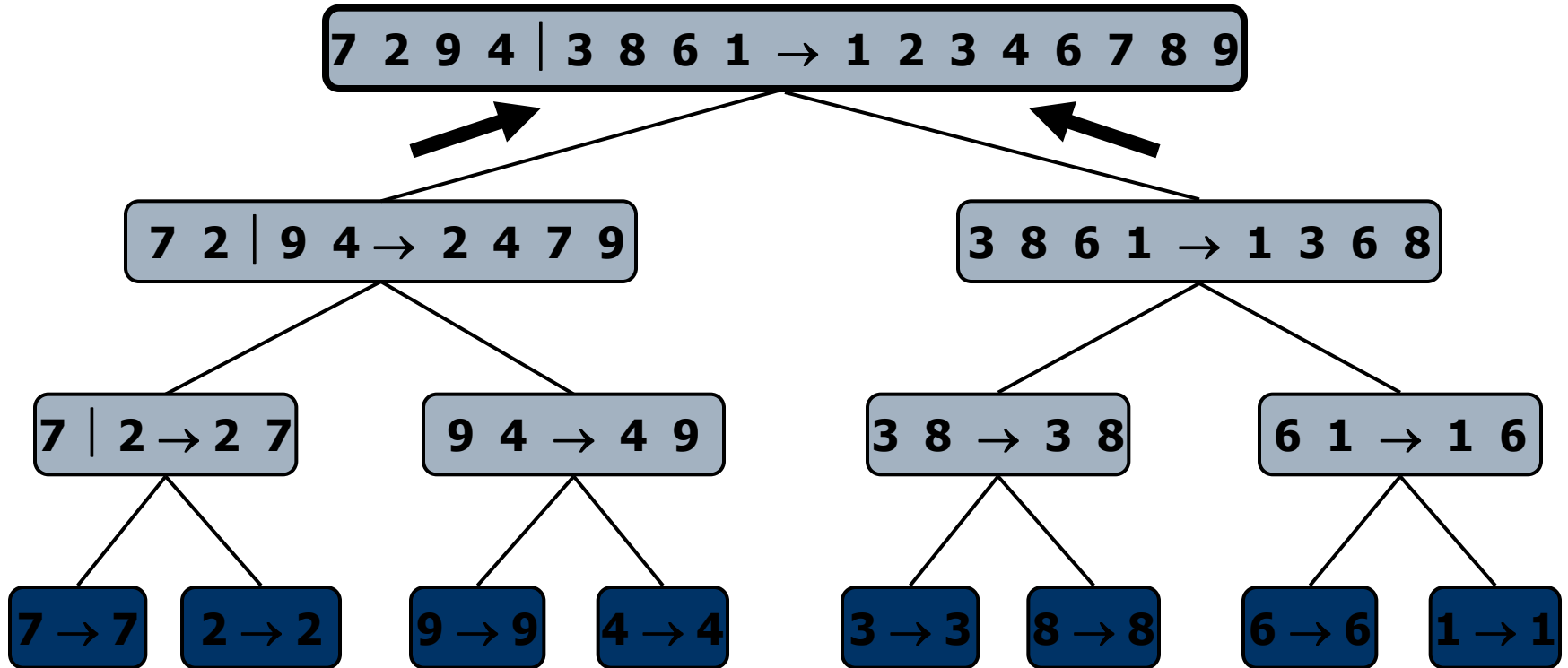
Exemplo (cont.)

- Chamada Recursiva, ..., une, une, une



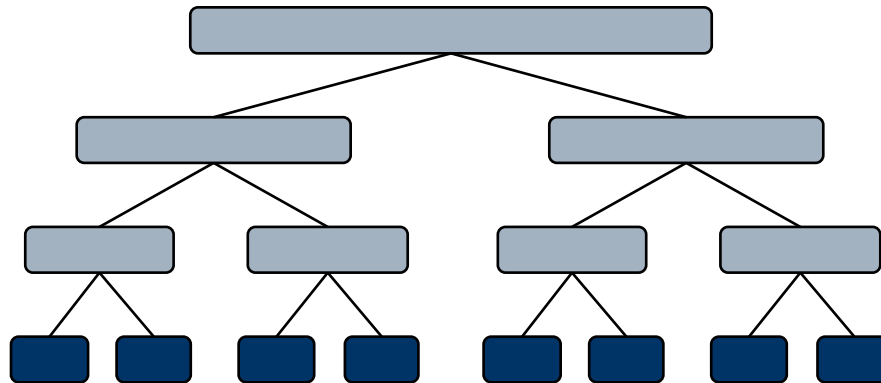
Exemplo (cont.)

- Une



Análise MergeSort

- A altura h da árvore mergeSort é $O(\log n)$
- Em cada chamada recursiva a sequência é dividida ao meio
- O trabalho feito nos nós de profundidade é $O(n)$
- Assim, o tempo total de merge-sort é $O(n \log n)$



Algoritmo QuickSort

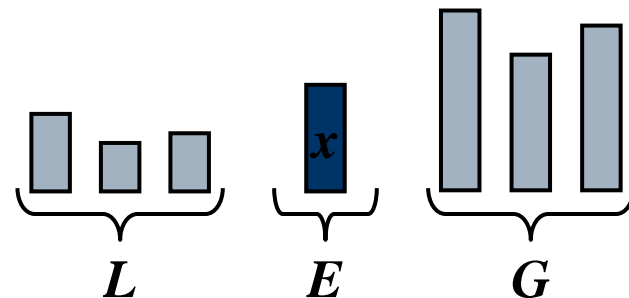
QuickSort é um algoritmo de ordenação baseado no paradigma Dividir-e-Conquistar

QuickSort recebe uma sequência de entrada S com n elementos e consiste em três etapas:

Divide: Escolhe um elemento x (meio do vetor, chamado pivô)

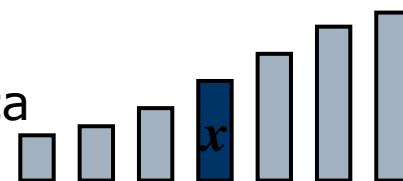
Particiona S em:

- L elementos menores que x
- E elementos iguais a x
- G elementos superiores a x



Recorrência: recursivamente ordena L e G

Conquista: une L , E e G numa sequência ordenada única



Algoritmo Ordenação por Partição *Quick Sort*

1. Caso básico:

Se o número (n) de elementos do vector (v) a ordenar for 0 ou 1, não é preciso fazer nada

2. Passo de partição:

2.1. Escolher um elemento arbitrário (x) do vector (chamado *pivot*)

2.2. Partir o vector inicial em dois sub-vectores (esquerdo e direito):

- ♦ valores $\leq x$ no sub-vector esquerdo
- ♦ valores $\geq x$ no sub-array direito
- ♦ pode existir um 3º sub-array central com valores $=x$

3. Passo recursivo:

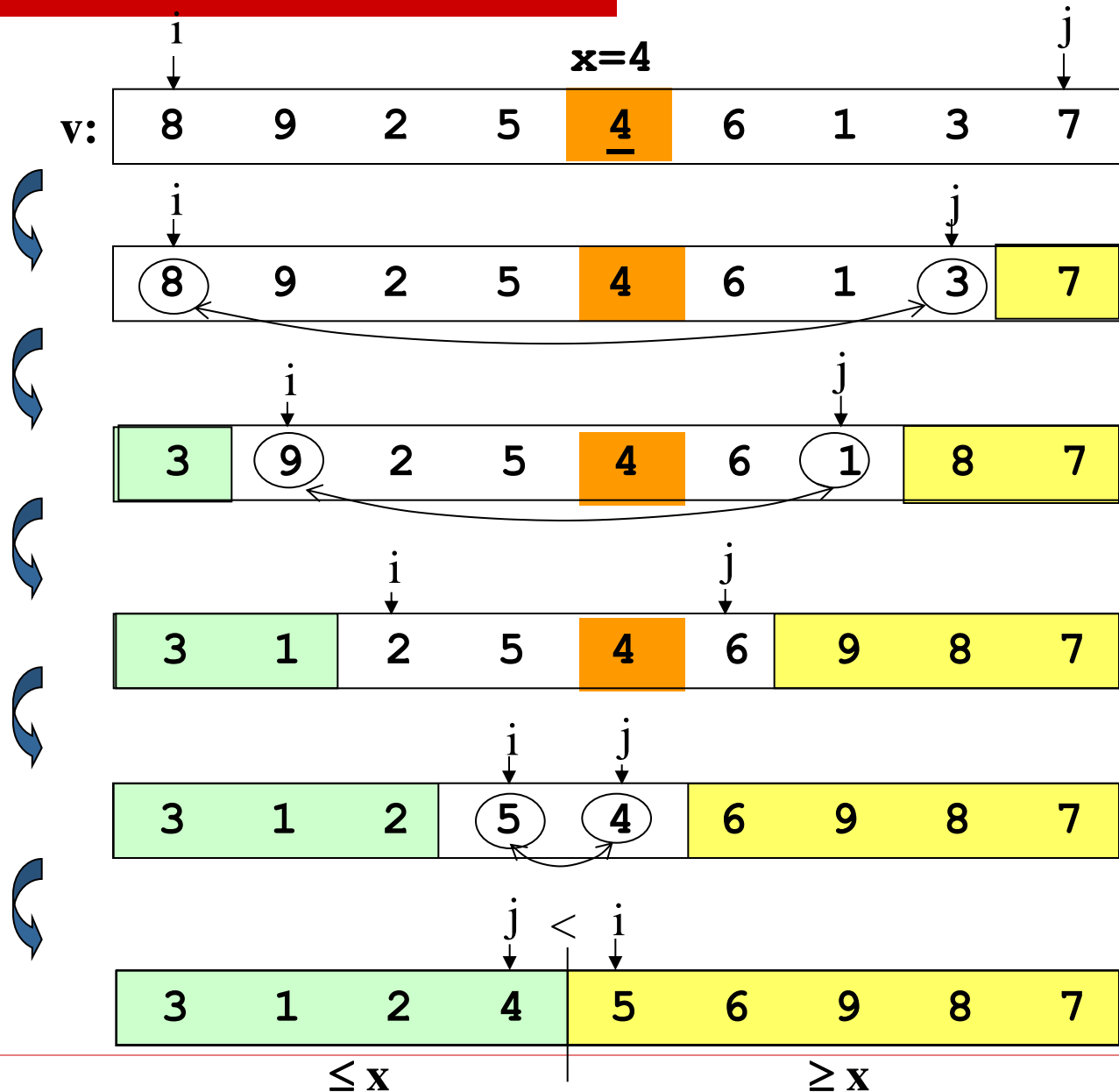
Ordenar os sub-vectores esquerdo e direito, usando o mesmo método recursivamente

Algoritmo recursivo baseado na técnica *divide and conquer*!

Algoritmo Ordenação por Partição *Quick Sort*

```
quicksort (v, inf, sup)
    pivot ← v[(inf+sup)/2]           //elemento do meio do array
    i ← inf                          //índice da 1ª posição do vector
    j ← sup                          //índice da última posição do vector
    Enqto i ≤ j  fazer
        Enqto v[i] < pivot
            i++
        Enqto v[j] > pivot
            j--
        se i ≤ j
            v[i] ↔ v[j]
            i++
            j--
    se (inf < j)
        quicksort (v, inf, j)
    se (sup > i)
        quicksort (v, i, sup)
```

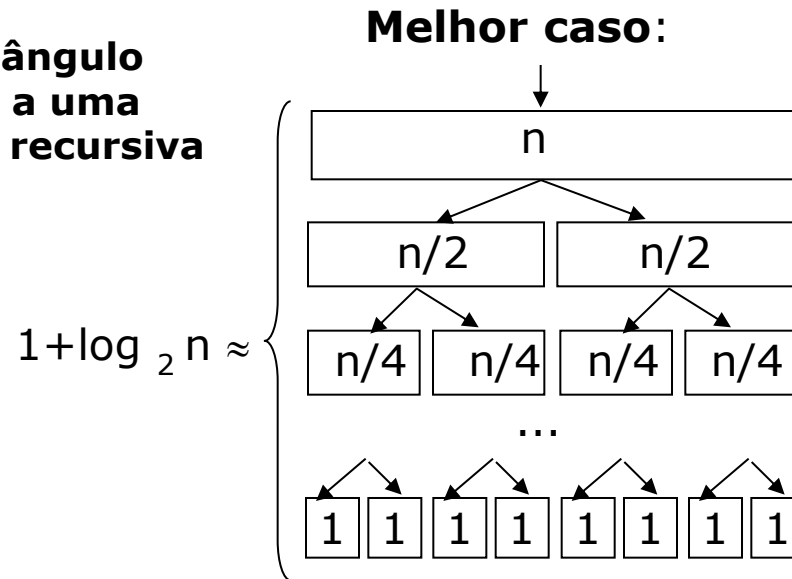
Exemplo do Passo de Partição – Melhor Caso



Eficiência da Ordenação por Partição

Melhor caso: Ocorre quando a tabela está sempre dividida precisamente ao meio

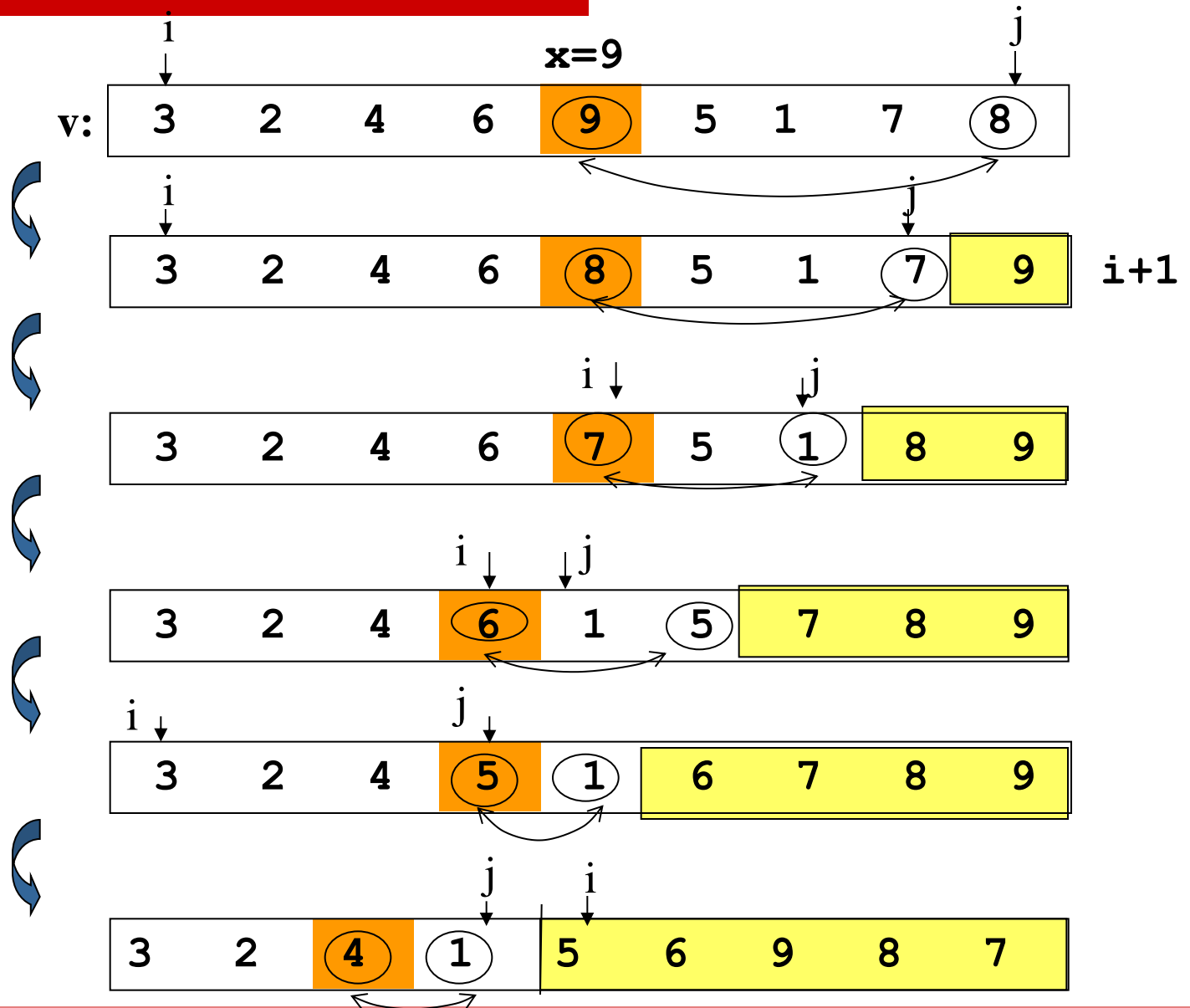
cada rectângulo
refere-se a uma
chamada recursiva



Melhor caso:

- profundidade de recursão: $\approx 1 + \log_2 n$
- Tempo de execução total : **$T(n) = O[(1 + \log_2 n) \times n] = O(n \log n)$**

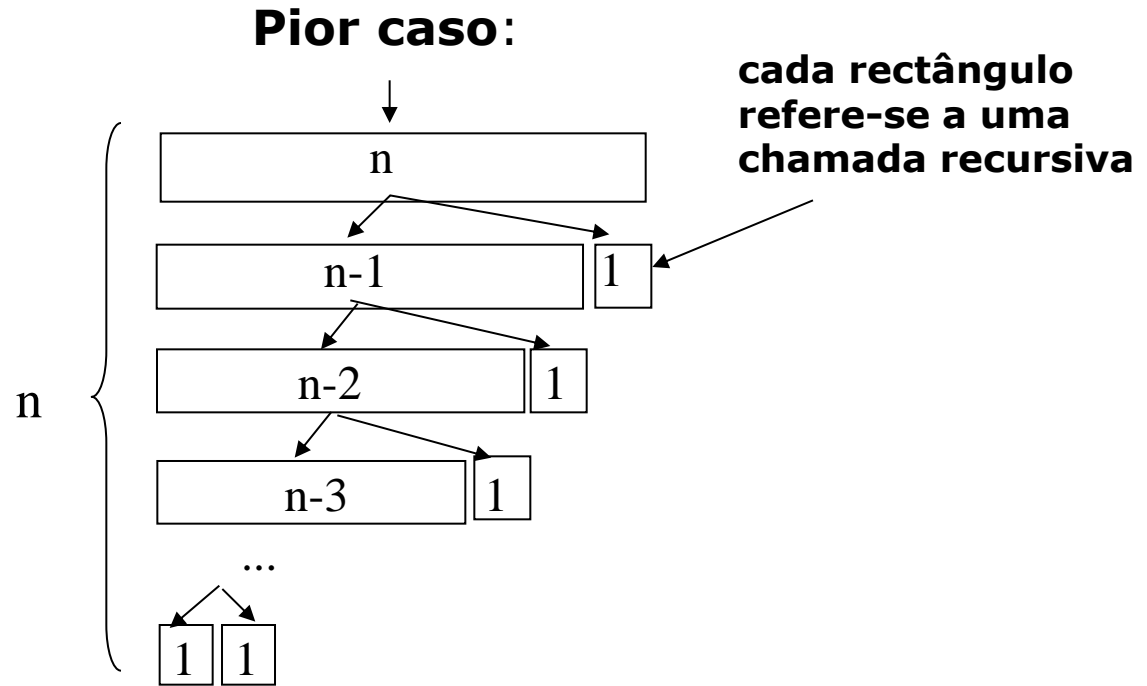
Exemplo do Passo de Partição – Pior Caso



...

Eficiência da Ordenação por Partição

Pior caso: Uma das tabelas é vazia



Pior caso:

- profundidade de recursão: n
- tempo de execução total (somando totais de linhas):

$$T(n) = O[n + (n-1) + (n-2) + \dots + 2]$$

$$T(n) = O\left[\frac{n + (n-1)(n+2)}{2}\right] = \mathbf{O(n^2)}$$

Algumas considerações

- Prova-se que no caso médio (na hipótese de os valores estarem aleatoriamente distribuídos pelo array), o tempo de execução é da mesma ordem que no melhor caso → **$T(n) = O(n \log n)$**
- Para um “input” pequeno ($n \leq 20$) o “QuickSort” não se comporta tão bem como o método Ordenação Linear, **é mais adequado para ordenar sequências de grande dimensão**
- O critério seguido para a escolha do *pivot* destina-se a tratar eficientemente os casos em que o vetor está inicialmente ordenado
 - se se escolher sempre o *pivot* como o primeiro elemento do vector e se este estiver ordenado, este algoritmo toma um tempo da ordem **n^2** para não fazer nada na ordenação da tabela
 - Em regra, o *pivot* escolhido não deve ser nem o primeiro nem o último elemento do vector

Complexidade Espacial QuickSort

- O espaço de memória exigido por cada chamada de **QuickSort**, sem contar com chamadas recursivas, é independente do tamanho (n) do vector
- O espaço de memória total exigido pela chamada de **QuickSort**, incluindo as chamadas recursivas, é pois proporcional à profundidade de recursão
- Assim, a complexidade espacial de **QuickSort** é:
 - **$O(\log n)$** no melhor caso (e no caso médio)
 - **$O(n)$** no pior caso