# Complexidade de Algoritmos

## Algoritmos Envolvendo Seqüências e Conjuntos

Prof. Osvaldo Luiz de Oliveira

Estas anotações devem ser complementadas por apontamentos em aula.

# Seqüências e Conjuntos

- Conjunto: coleção finita de elementos distintos.
- Seqüência: coleção finita e ordenada de elementos.

Obs.: se não especificarmos nada em contrário, seqüências serão de elementos distintos.

# Ordenação

KNUTH, D. E.. **The Art of  Computer Programming. Vol 3, Sorting and Searching**.
Reading: Addison-Wesley, 1997, é uma “enciclopédia” sobre algoritmos de ordenação e busca.

# Diferentes reduções, diferentes algoritmos

# *Insertion Sort*

I) Interface

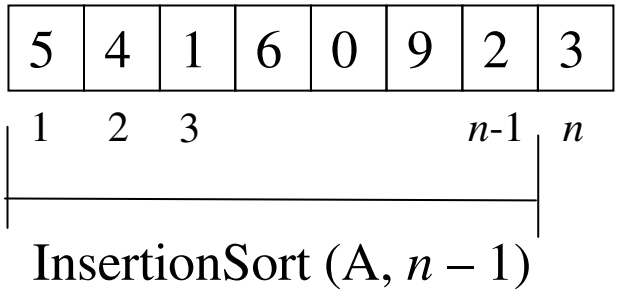
InsertionSort (A, *n*)

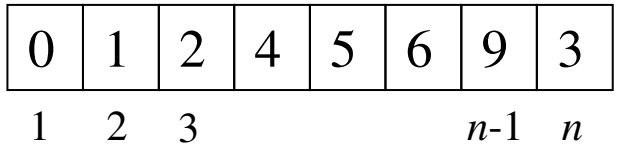
II) Significado

Ordena “in-loco” o vetor A de *n* elementos.

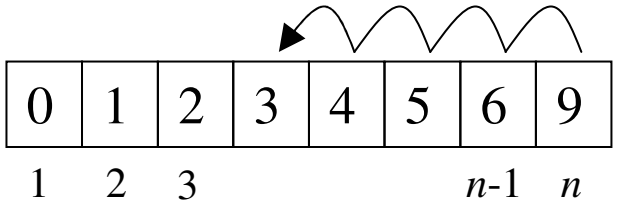
# *Insertion Sort*

## III) Redução




Inserir elemento A[*n*] na posição dele.



# *InsertionSort*

Comandos:

InsertionSort (A, $n - 1$);

i := $n$;

**enquanto** ( i ≥ 2 e A [i] > A [i - 1] )

{

troca := A[i]; A [i] := A [i – 1]; A [i – 1] := troca;

i := i - 1

}

## IV) Base

A redução é de 1 em 1. Escolhemos base para $n = 1$.

Comandos (para ordenar um vetor de 1 elemento):

Nenhum comando é necessário.

# O algoritmo

**Algoritmo** InsertionSort (A, *n*)
**Entrada**: vetor A de *n* elementos, $n \geq 1$.
**Saída**: o vetor A, ordenado "in-loco".

```
{
    se ( n > 1 )
    {
        InsertionSort (A, n – 1);

        i := n;
        enquanto ( i ≥ 2 e A [i] > A [i - 1] )
        {
            troca := A[i]; A [i] := A [i – 1]; A [i – 1] := troca;
            i := i - 1
         }

    }
}
```

# Complexidade

**Algoritmo** InsertionSort (A, $n$) — $T(n)$

**Entrada**: vetor A de $n$ elementos, $n \geq 1$.

**Saída**: o vetor A, ordenado "in-loco".

```
{
    se ( n > 1 )
    {
        InsertionSort (A, n – 1);          T(n – 1)

        i := n;                            n - 1
        enquanto ( i ≥ 2 e A [i] > A [i - 1] )
        {
            troca := A[i]; A [i] := A [i – 1]; A [i – 1] := troca;
            i := i - 1
        }
    }
}                                          T(1) = 0
```

# Concluindo

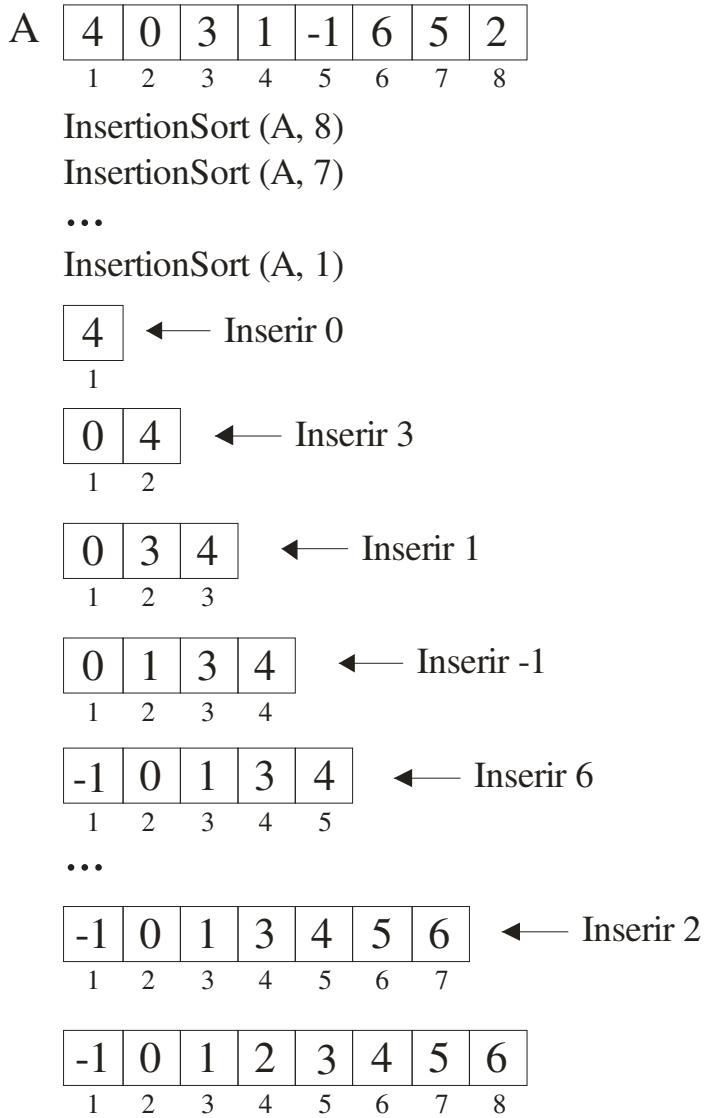
$T(n) = T(n - 1) + n - 1$

$T(1) = 0$

Resolvendo:

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

# Ilustrando




SC

# *Selection Sort*

I) Interface

SelectionSort (A, *n*)

II) Significado

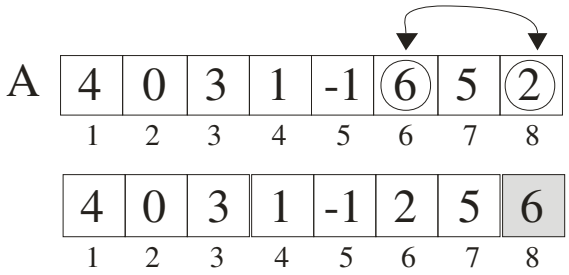
Ordena “in-loco” o vetor A de *n* elementos.

III) Redução

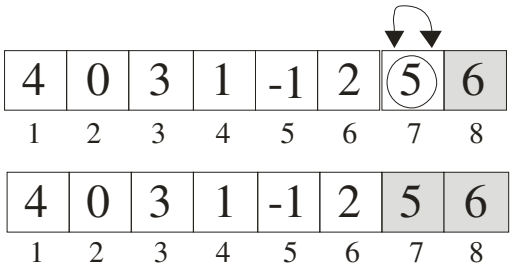
*Continuar ...*

# Ilustrando

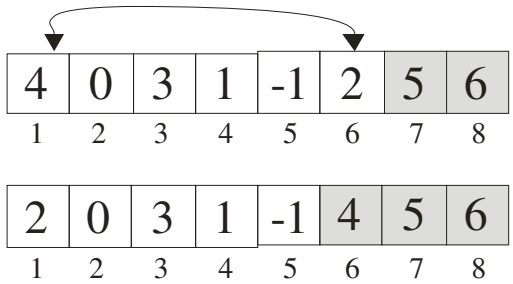
SelectionSort (A, 8)




SelectionSort (A, 7)




SelectionSort (A, 6)



# *Bubble Sort*

I) Interface

BubbleSort (A, *n*)

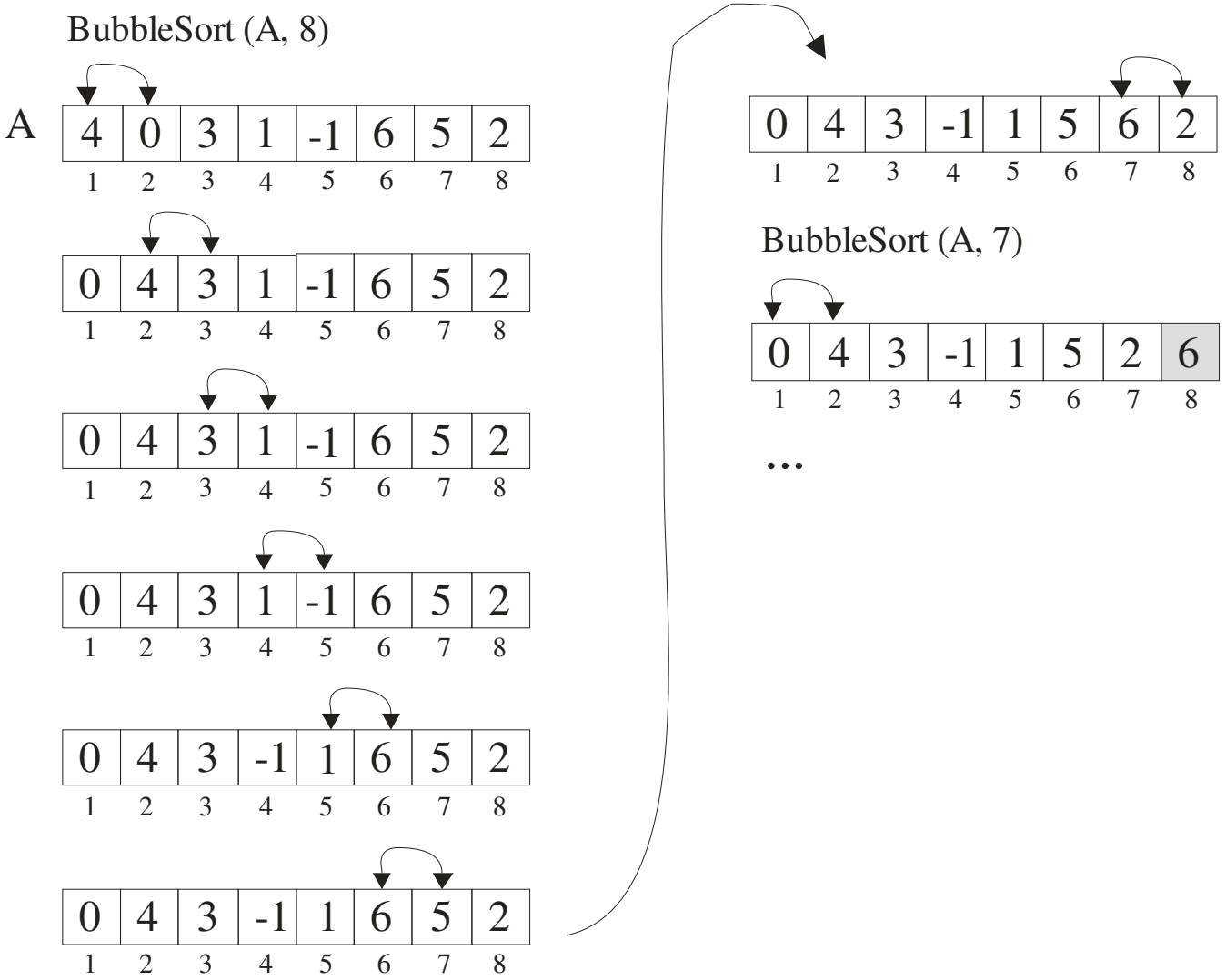
II) Significado

Ordena “in-loco” o vetor A de *n* elementos.

III) Redução

*Continuar ...*

# Ilustrando


BubbleSort (A, 8)
A
4 0 3 1 -1 6 5 2
1 2 3 4 5 6 7 8
0 4 3 1 -1 6 5 2
0 4 3 1 -1 6 5 2
0 4 3 1 -1 6 5 2
0 4 3 -1 1 6 5 2
0 4 3 -1 1 6 5 2
0 4 3 -1 1 5 6 2
BubbleSort (A, 7)
0 4 3 -1 1 5 2 6
...

# *Merge Sort*

I) Interface

MergeSort (A, *i*, *f*)

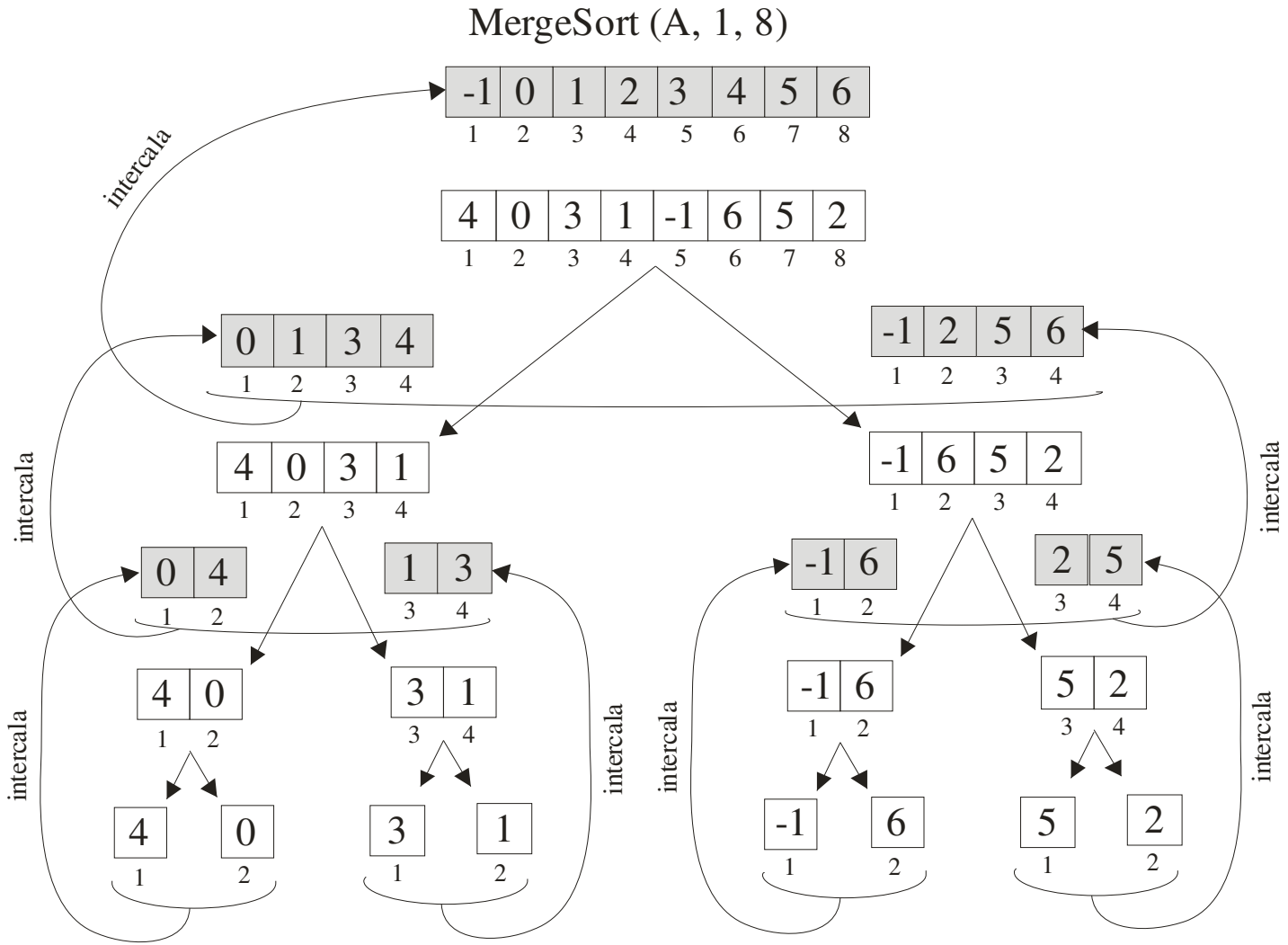
II) Significado

Ordena “in-loco” o vetor A do índice *i* até o índice *f*.

III) Redução

*Continuar ...*

# Ilustrando


MergeSort (A, 1, 8)
-1 0 1 2 3 4 5 6
1 2 3 4 5 6 7 8
intercala
4 0 3 1 -1 6 5 2
1 2 3 4 5 6 7 8
0 1 3 4
1 2 3 4
-1 2 5 6
1 2 3 4
4 0 3 1
1 2 3 4
-1 6 5 2
1 2 3 4
intercala
intercala
0 4
1 2
1 3
3 4
-1 6
1 2
2 5
3 4
4 0
1 2
3 1
3 4
-1 6
1 2
5 2
3 4
intercala
intercala
intercala
intercala
4
1
0
2
3
1
1
2
-1
1
6
2
5
1
2
2

# *Quick Sort*

I) Interface

QuickSort (A, *i*, *f*)

II) Significado

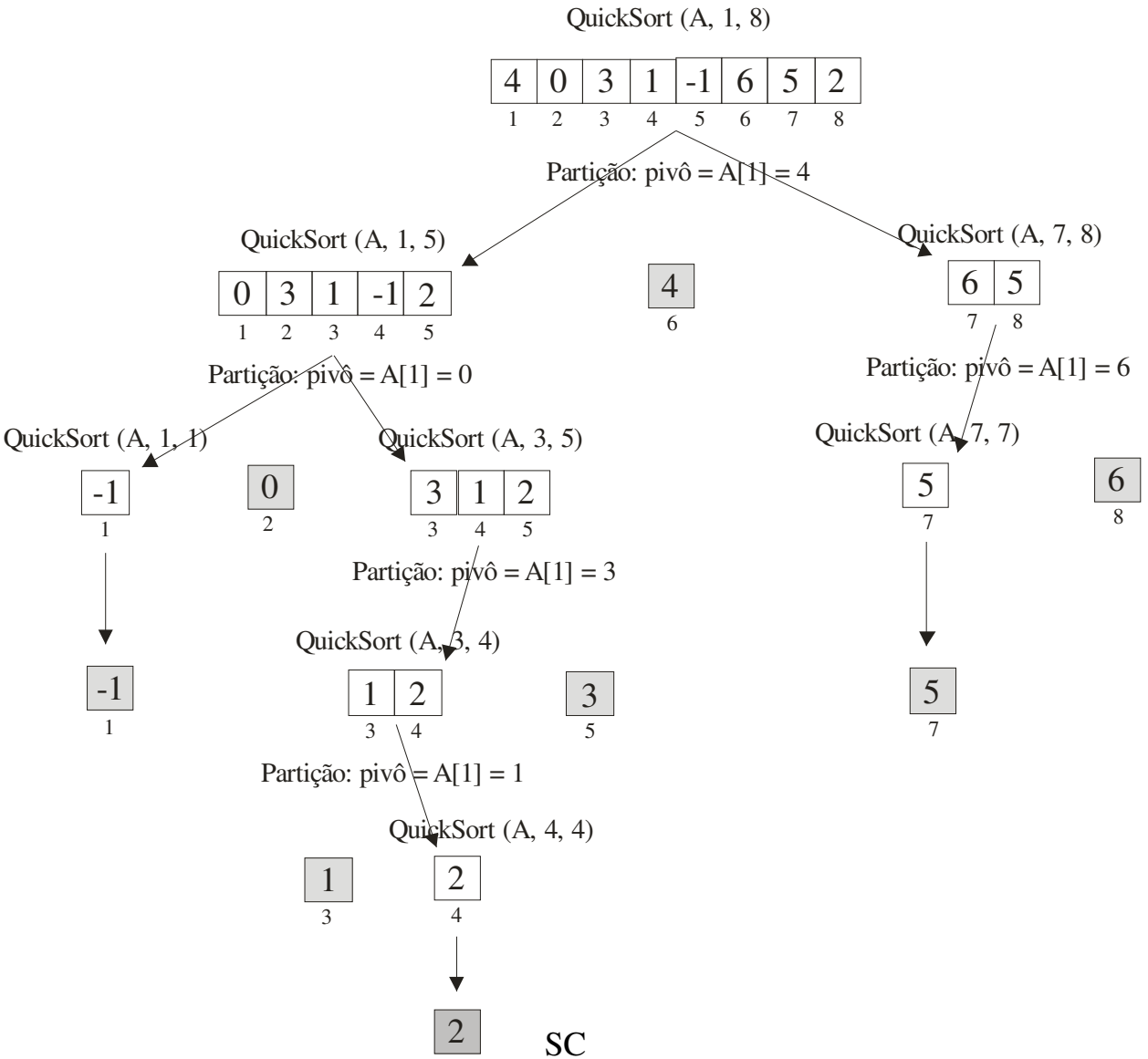
Ordena “in-loco” o vetor A do índice *i* até o índice *f*.

III) Redução

*Continuar ...*

Obs.: o algoritmo QuickSort foi originalmente proposto por:
HOARE, C. A. R. Quicksort, **Computer Journal** 5 (1), 1962 10-15.

# Ilustrando


QuickSort (A, 1, 8)
4 0 3 1 -1 6 5 2
1 2 3 4 5 6 7 8
Partição: pivô = A[1] = 4
QuickSort (A, 1, 5)
0 3 1 -1 2
1 2 3 4 5
4
6
QuickSort (A, 7, 8)
6 5
7 8
Partição: pivô = A[1] = 0
Partição: pivô = A[1] = 6
QuickSort (A, 1, 1)
-1
1
0
2
QuickSort (A, 3, 5)
3 1 2
3 4 5
QuickSort (A, 7, 7)
5
7
6
8
Partição: pivô = A[1] = 3
-1
1
QuickSort (A, 3, 4)
1 2
3 4
3
5
5
7
Partição: pivô = A[1] = 1
QuickSort (A, 4, 4)
1
3
2
4
2

# Partição

# Outros: proponha outras reduções

# Resumo

- Insertion, Selection

  - pior, melhor e médio: $O\ (n^2)$.

- Bubble

  - pior: $O\ (n^2)$.

  - melhor: $O\ (n)$.

  - médio: $O\ (n^2)$.

# Resumo

Merge

- pior, melhor e médio: $O$ ($n \log n$).

- Quick
  - pior: $O$ ($n^2$).
  - melhor: $O$ ($n \log n$).
  - médio: $O$ ($n \log n$).

# *Quick Sort* com mediana para pivô

**Algoritmo** QuickSort (A, $i, f$)
**Entrada**: vetor A e inteiros $i \geq 1, f \geq 1$.
**Saída**: o vetor A, ordenado "in-loco".

```
{
    se (i < f )
    {
        pivô := Mediana (A, i, f ); // a variável "pivô" recebe o índice do elemento pivô.
        pivô := Partição (A, i, f, pivô);  // "pivô" recebe o índice do pivô após a partição.
        QuickSort (A, i, pivô - 1);
        QuickSort (A, pivô + 1,  f  )
     }
}
```

# Complexidade (pior, melhor e média)

Seja $n = f - i + 1$.

$T(n) = 2\,T(n/2) + O(n)$

$T(0) = T(1) = 0$

Logo:

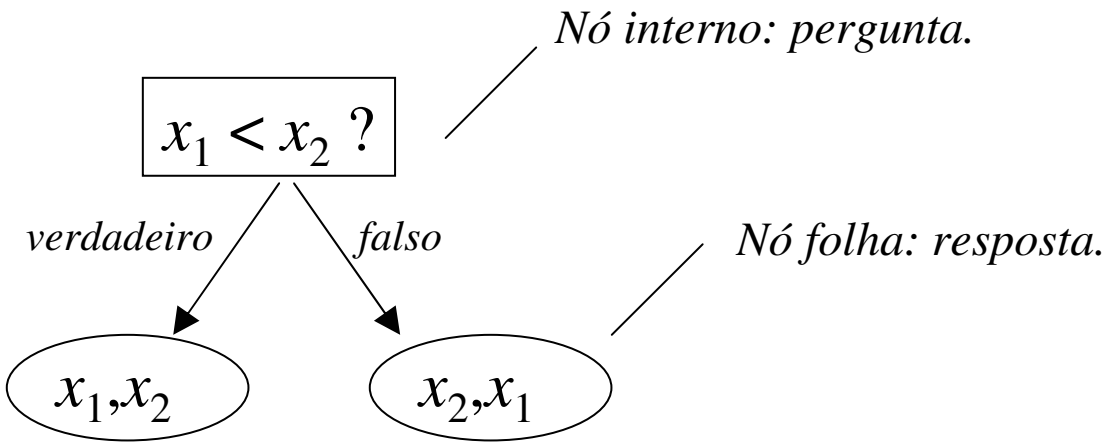
$T(n) = O\,(n \log n)$.

# Cota inferior para ordenação

- Algoritmos baseados em comparação.

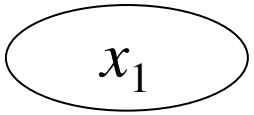
- Prove que qualquer algoritmo de ordenação baseado em comparação tem complexidade mínima de $\Omega$ ($n$ log $n$).

# Modelo de computação: árvore de decisão

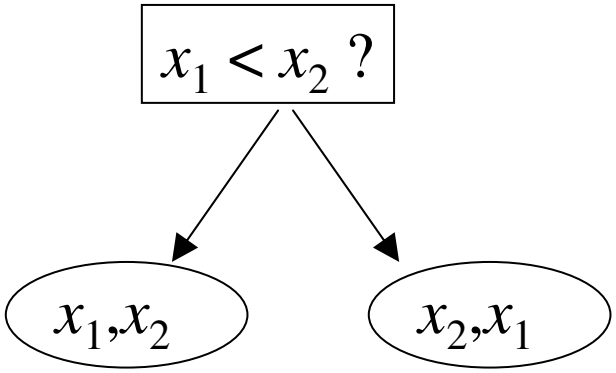


# $n = 1$ e $n = 2$

$n = 1$: { $x_1$ }

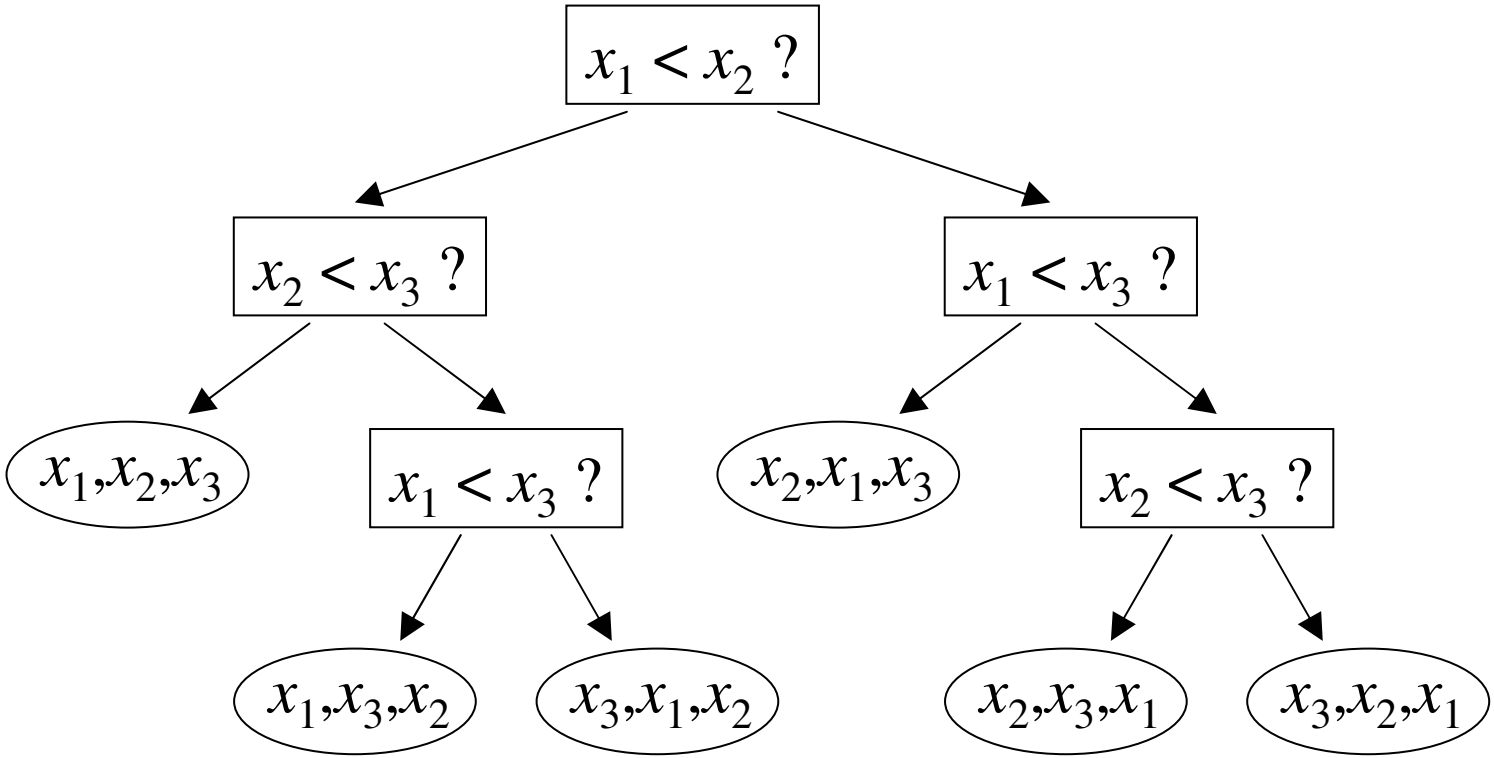



$n = 2$: { $x_1, x_2$ }



# $n = 3$

$n = 3$: { $x_1, x_2, x_3$ }


$x_1 < x_2$ ?
$x_2 < x_3$ ?
$x_1 < x_3$ ?
$x_1,x_2,x_3$
$x_1 < x_3$ ?
$x_2,x_1,x_3$
$x_2 < x_3$ ?
$x_1,x_3,x_2$
$x_3,x_1,x_2$
$x_2,x_3,x_1$
$x_3,x_2,x_1$

# Quantidade de folhas

| $n$ | Quantidade |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 6 |
| ... | |
| $n$ | $n!$ |

Permutação de $n$ elementos.

# Concluindo

- Altura mínima da árvore de decisão: $log\ (n!)$.

Aproximação de Stirling.

- $log\ (n!) = \Omega\ (n \log n)$.

- Logo a cota inferior é $\Omega\ (n \log n)$.

# Ordenação em tempo linear

- Algoritmos baseados em propriedades especiais dos elementos a ordenar.

- *Bucket Sort, Counting Sort e Radix Sort*.

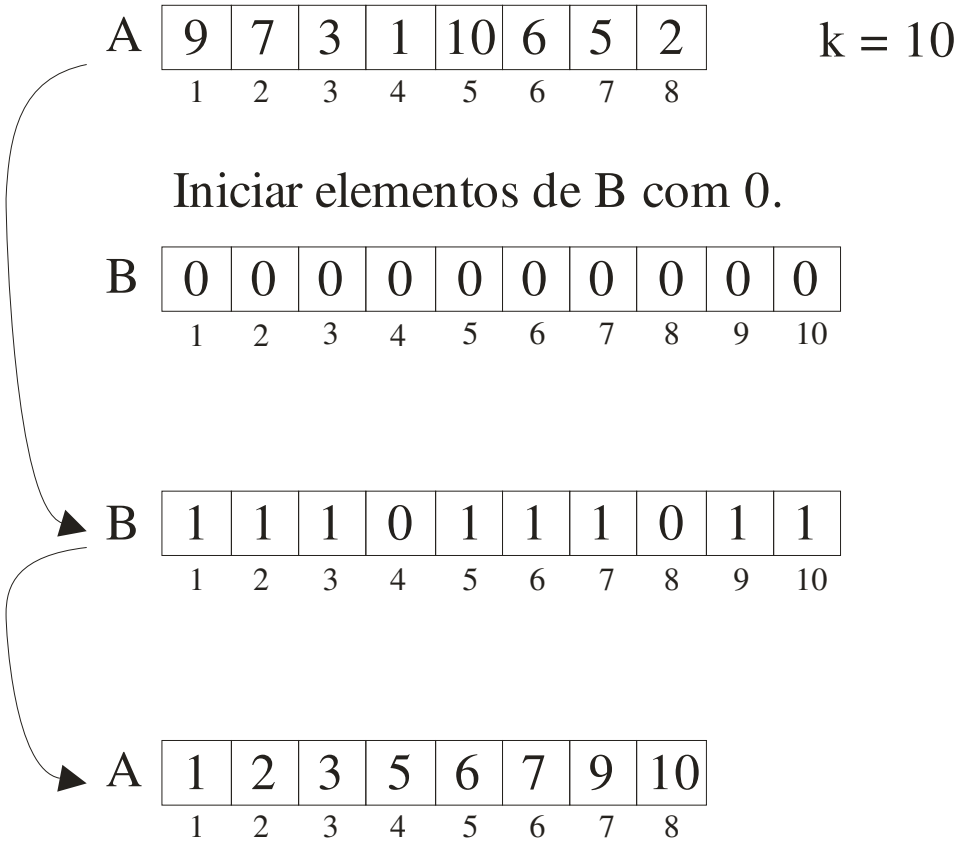
# Bucket Sort

**Pressuposição**: elementos a ordenar são inteiros no intervalo de 1 a $k$.

**Obs.:** não há repetição de elementos.

# Idéia

Alocar um *bucket* (vetor B) de tamanho igual a *k*.



# O algoritmo

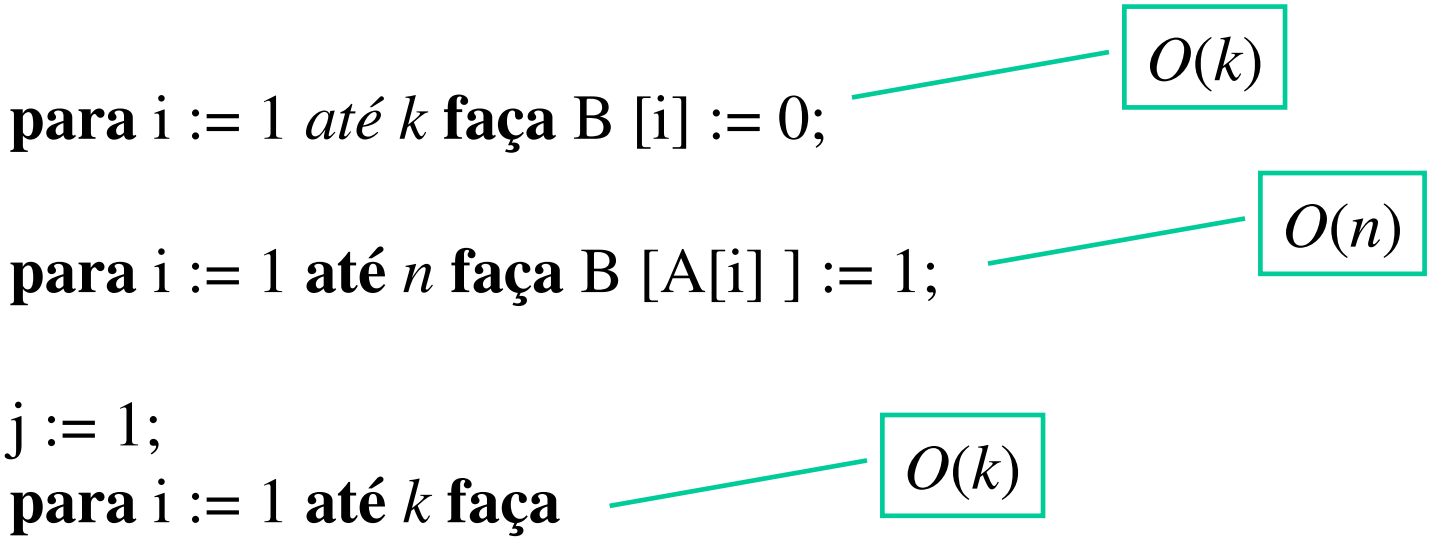
**Algoritmo** BucketSort (A, *n*, *k*)
**Entrada**: vetor A de *n* elementos inteiros situados no intervalo de 1 até *k*.
**Saída**: o vetor A ordenado.
**Usa**: vetor auxiliar B (bucket) de *k* elementos.

```
{
    para i := 1 até k faça B [i] := 0;

    para i := 1 até n faça B [A[i] ] := 1;

    j := 1;
    para i := 1 até k faça
        se ( B [ i ] = 1 )
        {
            A[ j ] := i;  j := j + 1
        }
}
```

# Complexidade



# Concluindo

$T(n, k) = O(n + 2k) = O(n + k)$.

Se $k = O(n)$ então $T(n) = O(2n) = O(n)$.

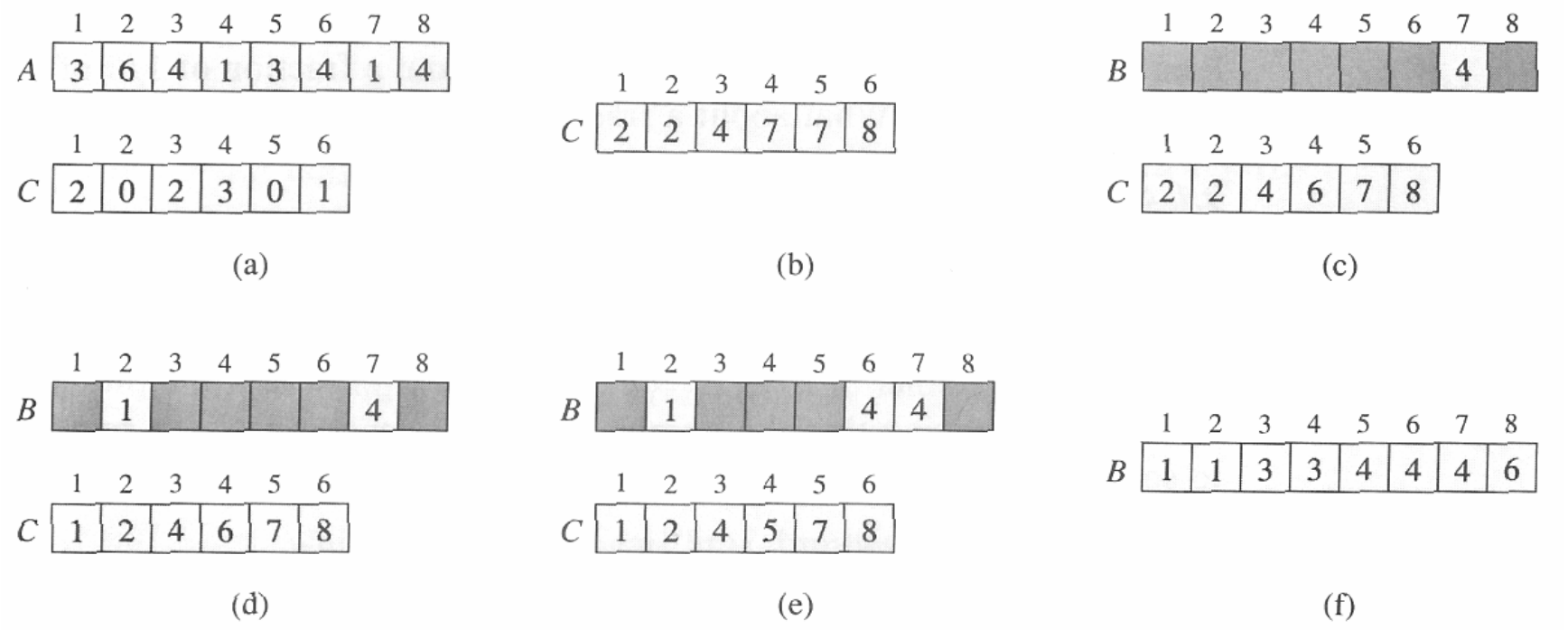
Se $k >>>> n$ então as complexidades de tempo e de espaço do algoritmo podem ser grandes.

# Counting Sort

**Pressuposição**: elementos a ordenar são inteiros, possivelmente repetidos, no intervalo de 1 a $k$.

# Idéia

Determinar, para cada elemento *x*, a quantidade de elementos que é menor ou igual a *x*.




Fonte: CORMEN, T.; LEISERSON, C.; RIVEST, R.; STEIN, C. **Introduction to Algorithms**. New York: MIT Press, 2004.

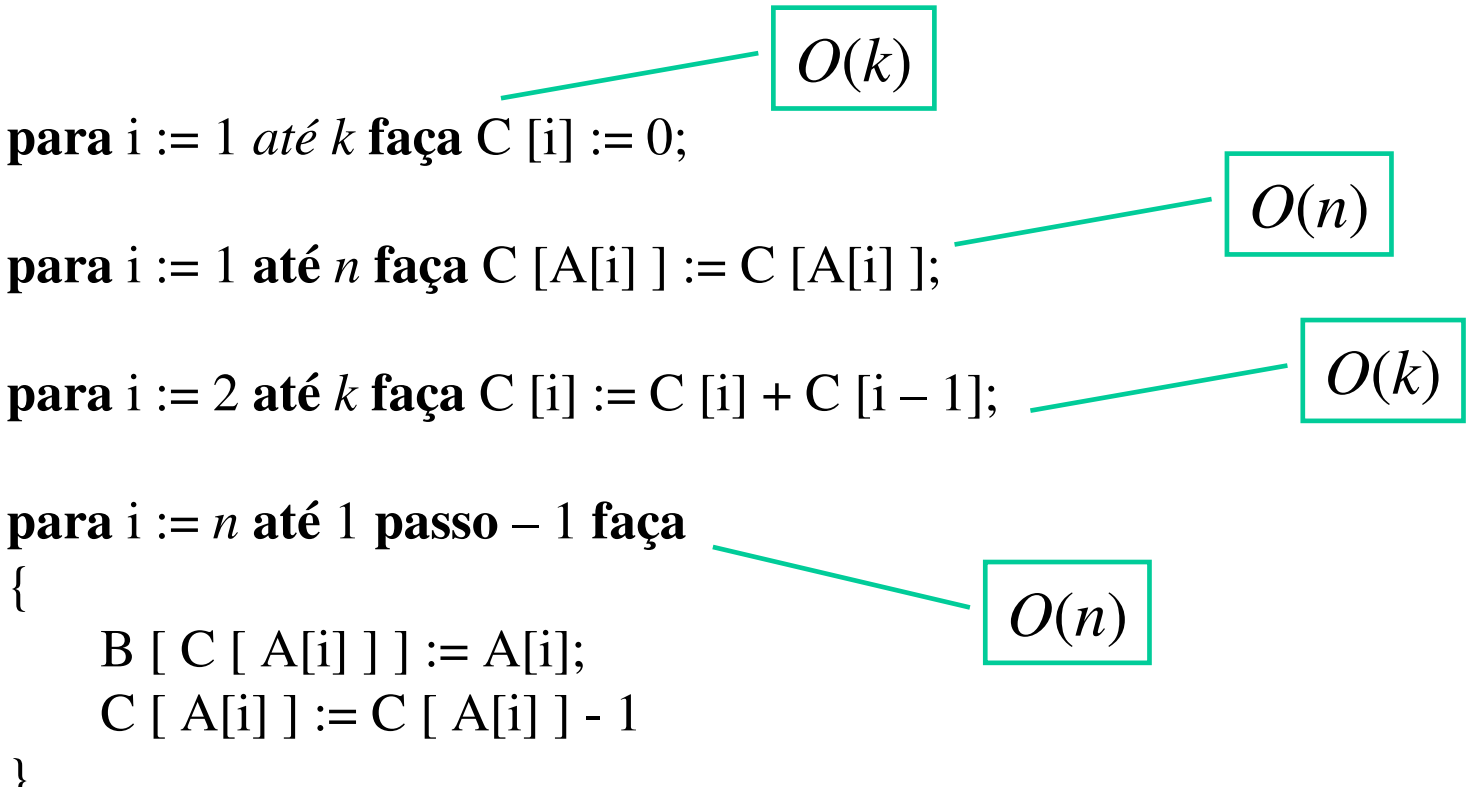
# O algoritmo

**Algoritmo** CountingSort (A, B, *n*, *k*)
**Entrada**: vetor A de *n* elementos inteiros situados no intervalo de 1 até *k*.
**Saída**: vetor B de *n* elementos.
**Usa**: vetor auxiliar C de *k* elementos.

```
{
    para i := 1 até k faça C [i] := 0;
    para i := 1 até n faça C [A[i] ] := C [A[i] ];
    // Neste ponto cada C [i] contém a quantidade de elementos igual a i.
    para i := 2 até k faça C [i] := C [i] + C [i – 1];
    // Neste ponto cada C [i] contém a quantidade de elementos menor ou igual a i.

    para i := n até 1 passo – 1 faça
    {
        B [ C [ A[i] ] ] := A[i];
        C [ A[i] ] := C [ A[i] ] - 1
    }
}
```

# Complexidade



# Concluindo

T($n$, $k$) = $O$ ($n$ + $k$).

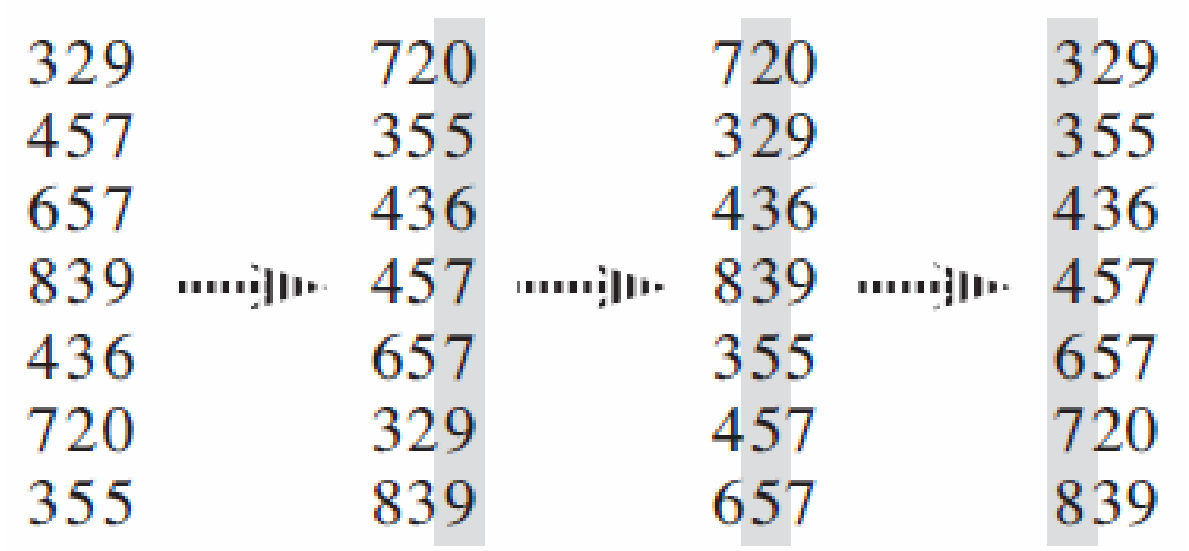
Se $k$ = $O$ ($n$) então T($n$) = $O$ ($n$).

Se $k$ >>>> $n$ então as complexidades de tempo e de espaço do algoritmo podem ser grandes.

Este algoritmo é **estável**: elementos com o mesmo valor aparecerão na saída na mesma ordem em que estavam na entrada.

# Radix Sort

Idéia: ordenar o conjunto dígito por dígito, do menos significativo ao mais significativo.




Fonte: CORMEN, T.; LEISERSON, C.; RIVEST, R.; STEIN, C. **Introduction to Algorithms**. New York: MIT Press, 2004.

# O algoritmo

**Algoritmo** RadixSort (A, *n*, *d*)
**Entrada**: vetor A de *n* elementos inteiros com *d* dígitos.
**Saída**: vetor A ordenado.
{
    **para** i := 1 **até** *d* **faça**
        Usar um algoritmo de ordenação estável para
        ordenar o vetor A pelo dígito i
}

# Complexidade

- Depende do algoritmo estável usado na ordenação.

- Suponhamos usar o CountingSort.

  - Se cada dígito está no intervalo de 1 até $k$ então a ordenação do i-ésimo dígito é igial a $O\ (n + k)$.

  - Logo $T(n, d, k) = (d\ n + d\ k)$.

  - Se $d$ for constante ($d <<<< n$) e $k = O\ (n)$ então $T(n) = O\ (n)$.

# Busca

- Linear (em um vetor não ordenado - visto): $O\ (n)$.

- Binária (em um vetor ordenado - visto): $O\ (log\ n)$

# Variações de busca binária
## (ver lista de exercícios)

- Busca em uma seqüência cíclica.
- Busca de um índice i tal que i = A[i].
- Busca em uma seqüência de tamanho não conhecido.
- Cálculo de raízes de equações (método de Bolzano).

# Estatísticas de ordem

# Máximo e mínimo

- Máximo: $\Theta(n)$.
- Mínimo: $\Theta(n)$.
- Máximo e mínimo simultaneamente (ver lista de exercícios): aprox. $3n/2$ comparações em vez de $2n$ comparações.

# Seleção do $k$-ésimo menor

(caso médio linear)

# Seleção do *k*-ésimo menor
## (pior caso linear)

- S: uma coleção de *n* elementos, possivelmente repetidos.

- *k*: um inteiro $1 \leq k \leq n$.

- A idéia é encontrar um elemento *m* que particione S em:
  - $S_1$: coleção de elementos menores do que *m*;
  - $S_2$: coleção de elementos iguais a *m*;
  - $S_3$: coleção de elementos maiores do que *m*.

Obs.: este algoritmo foi originalmente proposto por:

BLUM, M.; FLOYD, R.W.; PRATT, V.; RIVEST, R. and TARJAN, R. Time bounds for selection, **J. Comput. System Sci.** 7 (1973) 448-461.

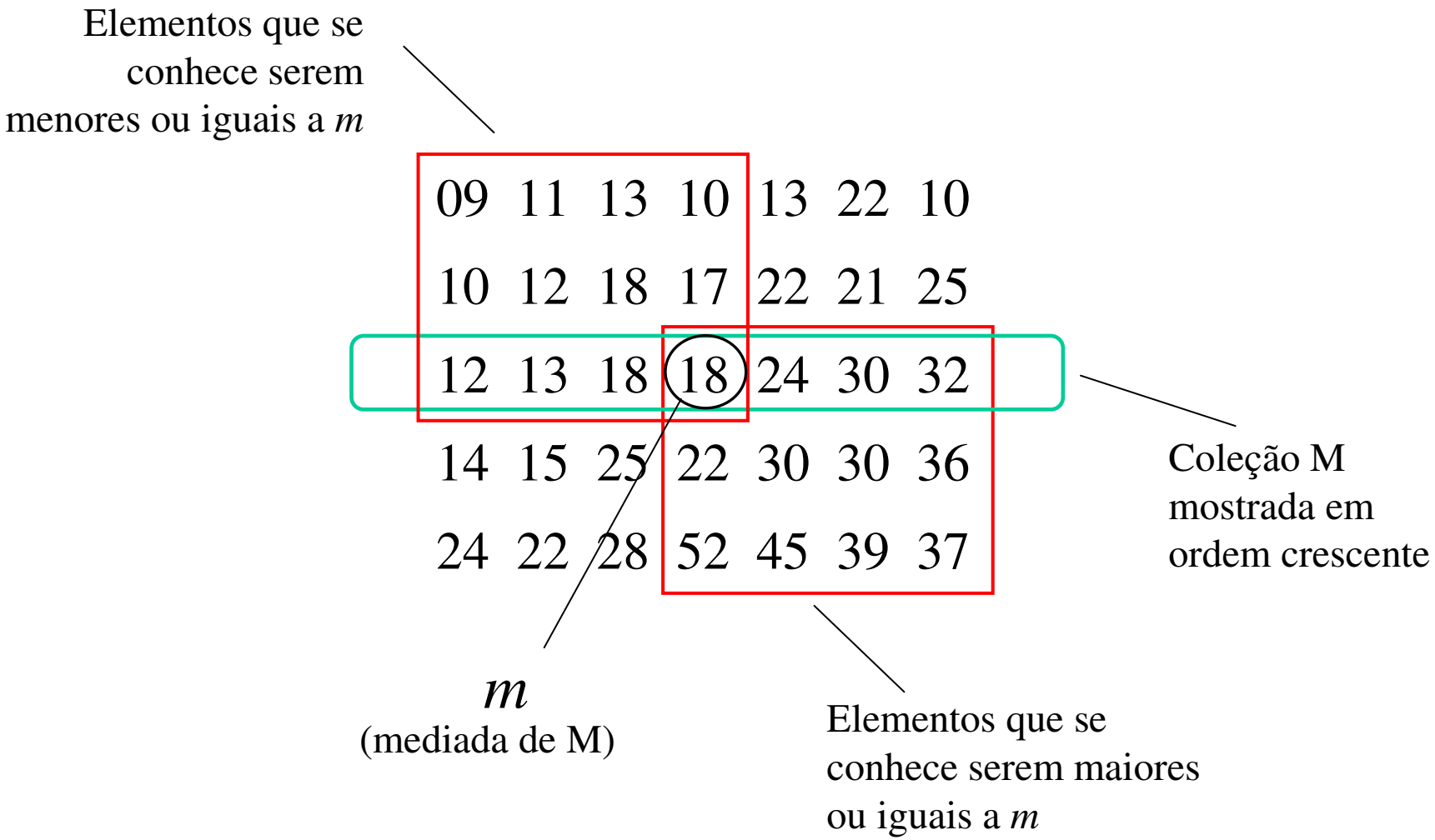
# Como achar *m*?

• Dividir S em blocos de 5 elementos cada.

• Cada bloco de 5 elementos é ordenado e a mediana de cada bloco é utilizada para formar uma coleção M.

• Agora M contém $\lfloor n/5 \rfloor$ elementos e nós podemos achar a mediana *m* de M cinco vezes mais rápido.

# Como achar *m*?


Elementos que se conhece serem menores ou iguais a *m*
09 11 13 10 13 22 10
10 12 18 17 22 21 25
12 13 18 18 24 30 32
14 15 25 22 30 30 36
24 22 28 52 45 39 37
Coleção M mostrada em ordem crescente
*m*
(mediada de M)
Elementos que se conhece serem maiores ou iguais a *m*

# Particionar S em subcoleções $S_1$, $S_2$ e $S_3$ tendo *m* como pivô

$S_1$ = { 09, 11, 13, 10, 13, 10, 10, 12, 17, 12, 13, 14, 15 }

$S_2$ = { 18, 18, 18 }

$S_3$ = { 22, 22, 21, 25, 24, 30, 32, 25, 22, 30, 30, 36, 24, 22, 28,
52, 45, 39, 37 }

$$n_1 = | S_1 | = 13$$
$$n_2 = | S_2 | = 3$$
$$n_3 = | S_3 | = 19$$

# O Algoritmo

**Algoritmo** Seleção (S, $n$, $k$)
**Entrada**: S, coleção de $n$ elementos, possivelmente repetidos e um inteiro $1 \leq k \leq n$.
**Saída**: retorna o $k$-ésimo menor elemento da coleção S.

```
{
    se ( n < 50 )
    {
        Ordenar S (qualquer algoritmo visto ou na "força bruta").
        Retornar o k-ésimo menor elemento em S.
     }
     senão
     {
```

Dividir S em $\lfloor n/5 \rfloor$ blocos de 5 elementos (o último pode ter menos do que 5 elementos).
Ordenar cada bloco de 5 elementos (qualquer algoritmo).
Seja M o conjunto das medianas de cada bloco de 5 elementos.

$m$ := Seleção (M, $\lfloor n/5 \rfloor$, $\lfloor n/10 \rfloor$ ). ($m$ é a mediana de M).

Particionar S usando $m$ como pivô em:
- $S_1$: coleção de elementos menores do que $m$;
- $S_2$: coleção de elementos iguais a $m$;
- $S_3$: coleção de elementos maiores do que $m$.

Sejam $n_1$, $n_2$ e $n_3$ as quantidades de elementos em $S_1$, $S_2$ e $S_3$.

**se** ( $k \leq n_1$ ) **retornar** Seleção ($S_1$, $n_1$, $k$)
**senão**
**se** ( $k \leq n_1 + n_2$) **retornar** $m$
**senão retornar** Seleção ($S_3$, $n_3$, k - $n_1$ - $n_2$)

}

}

# Complexidade

se ( $n < 50$ )
{
Ordenar S (qualquer algoritmo ou na "força bruta").
Retornar o $k$-ésimo menor elemento em S.
}
senão
{
Dividir S em $\lfloor n/5 \rfloor$ blocos de 5 elementos (o último pode ter menos do que 5 elementos).
Ordenar cada bloco de 5 elementos (qualquer algoritmo).
Seja M o conjunto das medianas de cada bloco de 5 elementos.

$m$ := Seleção (M, $\lfloor n/5 \rfloor$, $\lfloor n/10 \rfloor$). ($m$ é a mediana de M).

Particionar S usando $m$ como pivô em:
- S1: coleção de elementos menores do que $m$;
- S2: coleção de elementos iguais a $m$;
- S3: coleção de elementos maiores do que $m$.

Sejam $n1$, $n2$ e $n3$ as quantidades de elementos em S1, S2 e S3.

**se** ( $k \leq n1$ ) **retornar** Seleção (S1, $n1$, $k$)
**senão**
**se** ( $k \leq n1 + n2$) **retornar** $m$
**senão retornar** Seleção (S3, $n3$, k - $n1$ - $n2$)
}
}

$O(1)$

$O(n)$

$O(n)$

$T(n/5)$

$O(n)$

$T(3n/4)$

$T(3n/4)$

# Complexidade

$T(n) = T(n/5) + T(3n/4) + O(n)$, se $n \geq 50$
$T(n) = O(1)$, se $n < 50$

Resolvendo (método da substituição)

Teorema
$T(n) = O(n)$, ou seja, $T(n) \leq a\, n$, para alguma constante $a > 0$ e $n \geq N$.

Bases:
Para $n < 50$, $T(n) = c\, n$, para alguma constante $c > 0$.
Para satisfazer o teorema, $T(n) = c\, n \leq a\, n$. *Logo,* $a \geq c$ (primeira restrição). Esta restrição é plenamente factível.

Hipóteses de indução:
$T(n/5) \leq a\, n/5$ e que $T(3n/4) \leq a\, 3n/4$.

Prova de que a validade das hipóteses implicam na validade do teorema.
$T(n) = T(n/5) + T(3n/4) + c\, n \leq a\, n/5 + a\, 3n/4 + c\, n = a\, n\, 19/20 + c\, n$.
Para que provemos temos que chegar à conclusão de que $T(n) \leq a\, n$.
Ou seja, $T(n) \leq a\, n\, 19/20 + c\, n \leq a\, n$.
Isto é verdade para $a \geq 20\, c$ (segunda restrição, que também é factível e não conflita com a primeira).

# Por que divisão em blocos de 5 elementos?

- $1/5 + 3/4 = 19/20 \leq 1$.

  Assim: $T(n/5) + T(3n/4) \leq T(n)$.

- Você poderia propor outras divisões?

# Por que $n < 50$ para a base do algoritmo?

- A quantidade máxima de elementos em $S_1$ é $n - 3\lfloor n/10 \rfloor$ .
- Para $n \geq 50$ esta quantidade é menor que $3n/4$.

| $n$ | $n - 3\lfloor n/10 \rfloor$ | $3n/4$ |
|---|---|---|
| 49 | 37 | 36.7 |
| 50 | 35 | 37.5 |
| 59 | 44 | 44.25 |