Projeto e Análise de Algoritmos II

Sequências e Conjuntos

Prof. Dr. 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

Desenvolvendo o Insertion Sort

I) Interface

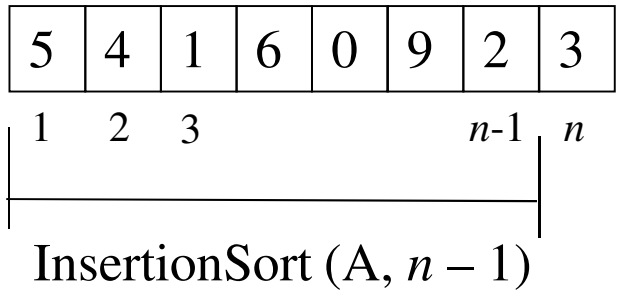
InsertionSort (A, *n*)

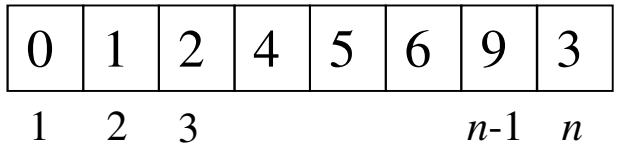
II) Significado

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

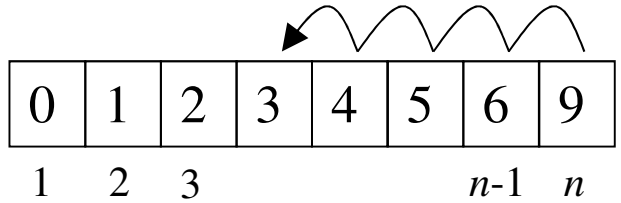
Desenvolvendo o Insertion Sort

III) Redução





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



Desenvolvendo o Insertion Sort

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.

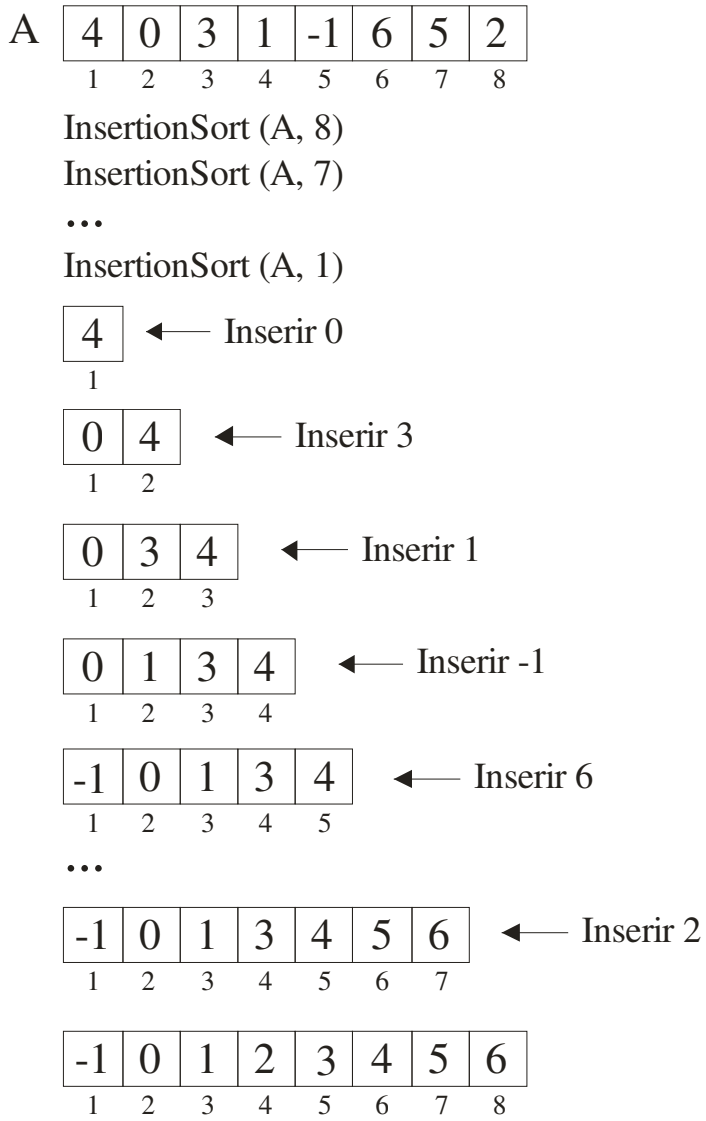
Desenvolvendo o Insertion Sort

V) 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
         }

    }
}
```

Ilustrando o funcionamento do Insertion Sort



Complexidade do Insertion Sort

```
Algoritmo InsertionSort (A, n)
Entrada: vetor A de n elementos, n ≥ 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
        }

    }
}
```

T(*n*)

T(*n* – 1)

n - 1

T(1) = 1

Complexidade do Insertion Sort

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

Resolvendo:

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

Desenvolvendo o Selection Sort

I) Interface

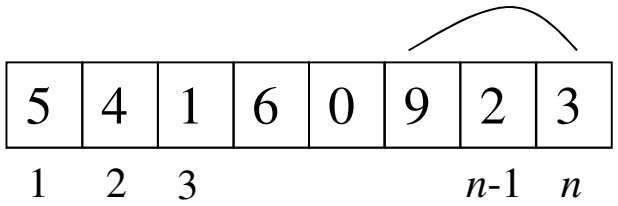
SelectionSort (A, *n*)

II) Significado

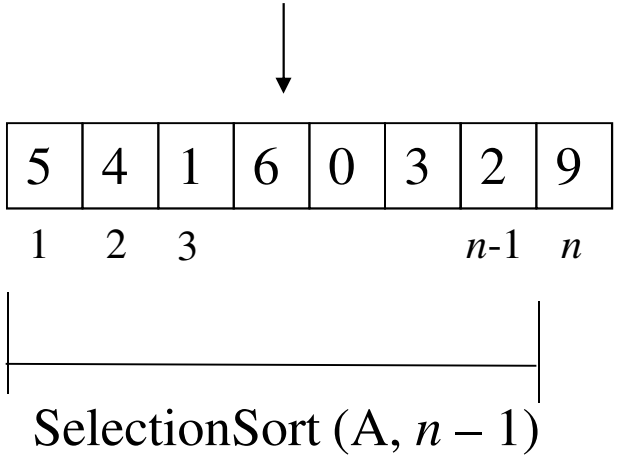
Ordena “in-loco” o vetor A de *n* elementos.

Desenvolvendo o Selection Sort

III) Redução (seleção do maior)



Localizar o maior elemento e trocar ele de posição com A[n] .



Desenvolvendo o Selection Sort

Comandos:

```
im := Maior (A, n);   // O algoritmo Maior retorna o índice do maior.
troca := A[im];   A [im] := A[n];   A[n] := troca;
SelectionSort (A, n – 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.

Desenvolvendo o Selection Sort

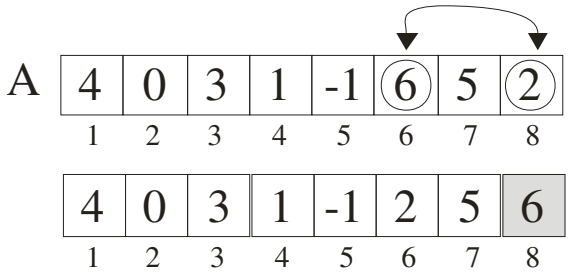
V) O Algoritmo

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

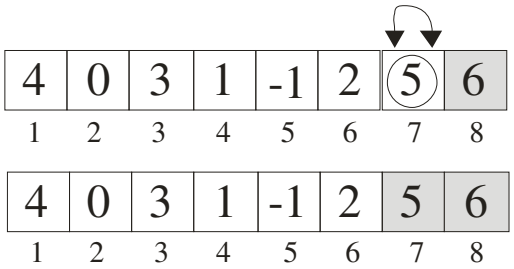
```
{
    se ( n > 1 )
    {
       im := Maior (A, n);   // O algoritmo Maior retorna o índice do maior.
       troca := A[im];   A [im] := A[n];   A[n] := troca;
       SelectionSort (A, n – 1)
     }
}
```

Ilustrando o funcionamento do Selection Sort

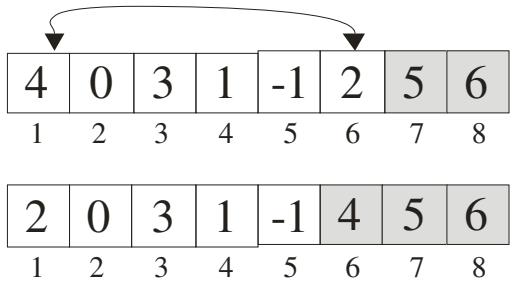
SelectionSort (A, 8)



SelectionSort (A, 7)



SelectionSort (A, 6)



...

Complexidade do Selection Sort

T(n)

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

```
{
    se ( n > 1 )
    {
       im := Maior (A, n);   // O algoritmo Maior retorna o índice do maior.
       troca := A[im];   A [im] := A[n];   A[n] := troca;
       SelectionSort (A, n – 1)
    }
}
```

n - 1

T(n – 1)

T(1) = 1

Complexidade do Selection Sort

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

$$T(1) = 1$$

Resolvendo:

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

Desenvolvendo o Bubble Sort

I) Interface

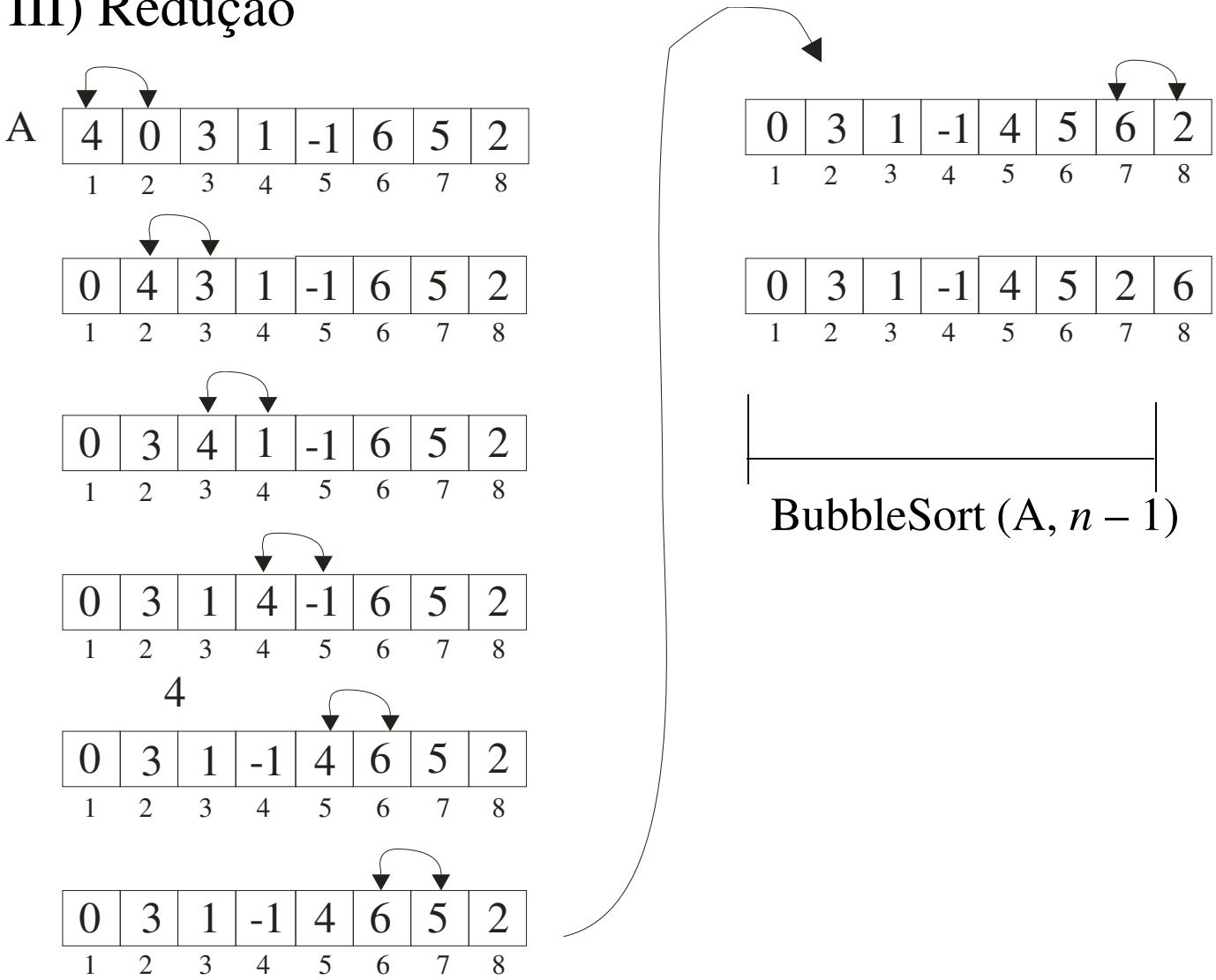
BubbleSort (A, *n*)

II) Significado

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

Desenvolvendo o Bubble Sort

III) Redução



Desenvolvendo o Bubble Sort

Comandos:

```
para i := 1 até n – 1 faça
    se ( A[i] < A[i+1] )
    {
        troca := A[i];  A [i] := A[i+1];  A[i+1] := troca
     }
```

BubbleSort (A , n – 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.

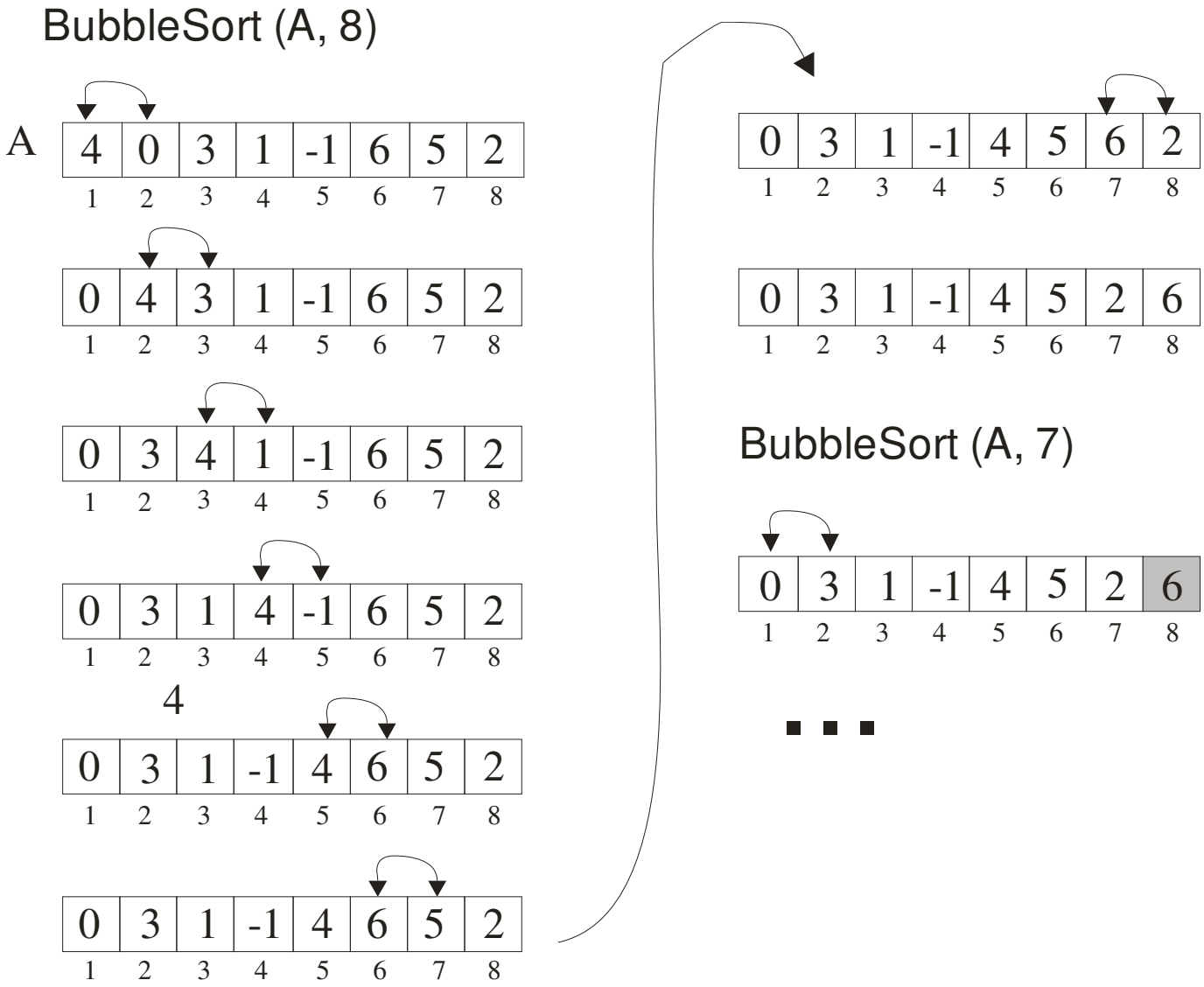
Desenvolvendo o Bubble Sort

V) O Algoritmo

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

```
{
    se ( n > 1 )
    {
       para i := 1 até n – 1 faça
          se ( A[i] < A[i+1] )
          {
              troca := A[i];  A [i] := A[i+1];  A[i+1] := troca
          }
        BubbleSort (A , n – 1)
    }
}
```

Ilustrando o funcionamento do Bubble Sort



Complexidade do BubbleSort

```
Algoritmo BubbleSort (A, n)                                   T(n)
Entrada: vetor A de n elementos, n ≥ 1.
Saída: o vetor A, ordenado "in-loco".
{
    se ( n > 1 )
    {                                                         n - 1
       para i := 1 até n – 1 faça
          se ( A[i] < A[i+1] )
          {
              troca := A[i];  A [i] := A[i+1];  A[i+1] := troca
          }
        BubbleSort (A , n – 1)                                T(n – 1)
    }
}                                                             T(1) = 1
```

Complexidade do Bubble Sort

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

$$T(1) = 1$$

Resolvendo:

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

Desenvolvendo o Merge Sort

I) Interface

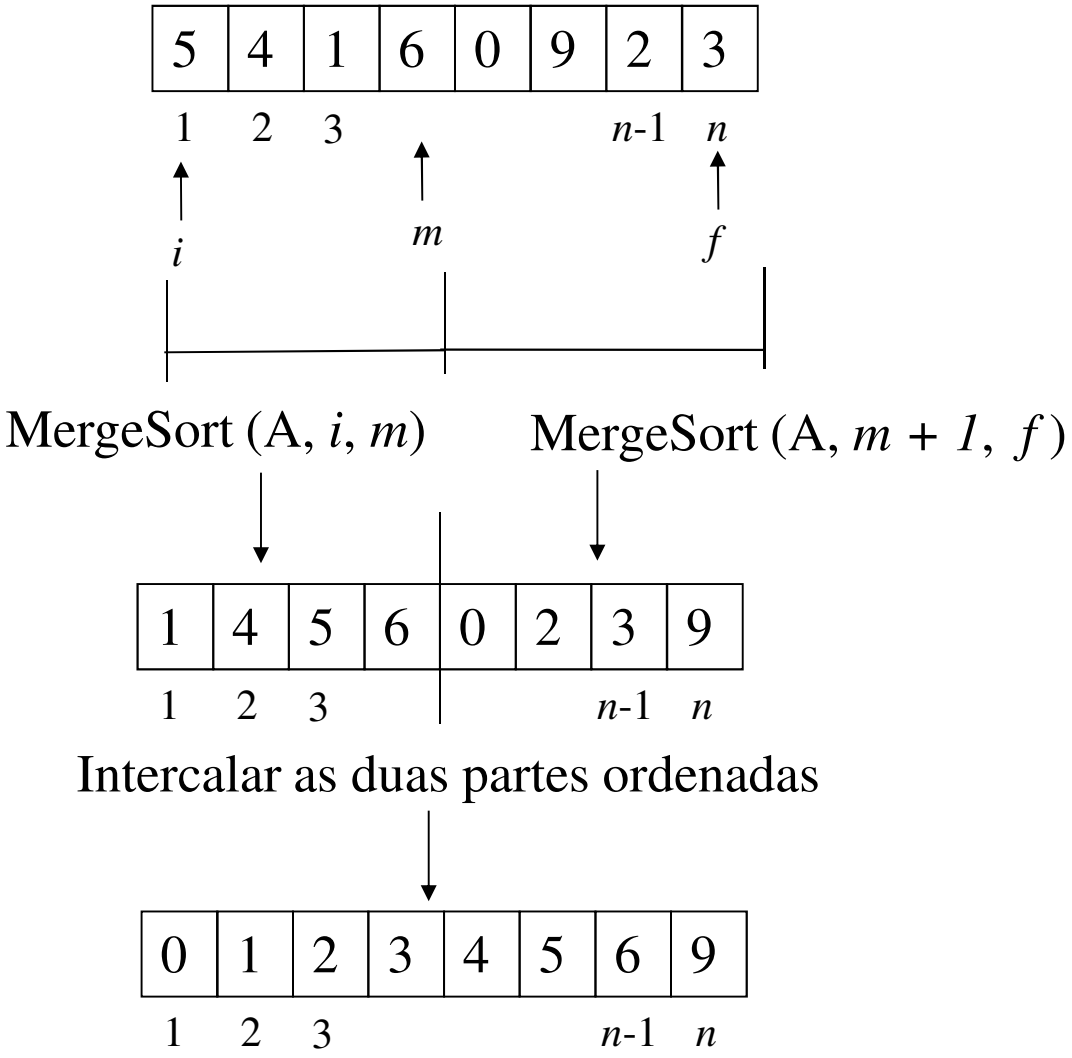
MergeSort (A, *i*, *f*)

II) Significado

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

Desenvolvendo o Merge Sort

III) Redução



Desenvolvendo o Merge Sort

Comandos:

m := $\lfloor$ (i + f) / 2$\rfloor$;

MergeSort (A, i, m);

MergeSort (A, m + 1, f)

Intercalar (A, i, m, f)

IV) Base

É caracterizada por i = f (veja reduções “dividir para conquistar”).

Comandos (para ordenar uma faixa de vetor com 1 elemento):

Nenhum comando é necessário.

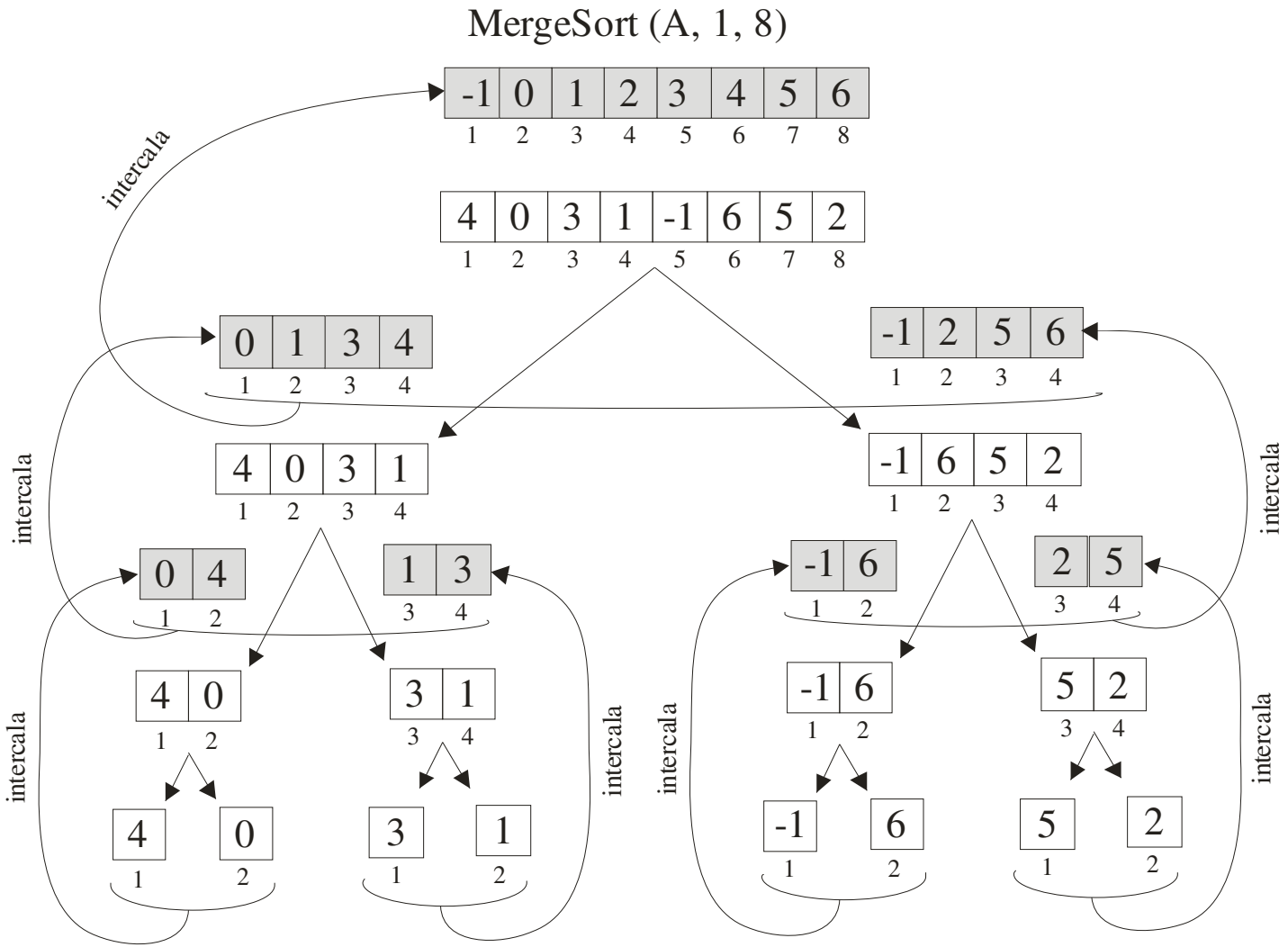
Desenvolvendo o Merge Sort

V) O Algoritmo

Algoritmo MergeSort (A, i, f)
Entrada: vetor ‘A’ e índices ‘i’ e ‘f’ do vetor (i ≤ f).
Saída: o vetor A, ordenado “in-loco” do índice ‘i’ até o índice ‘f’.

```
{
    se ( i < f )   // Ou i ≠ f .
    {
        m := ⌊ (i + f ) / 2⌋;
        MergeSort (A, i, m);
        MergeSort (A, m + 1, f );
        Intercalar (A, i, m, f )
    }
}
```

Ilustrando o funcionamento



Complexidade

Seja a quantidade de elementos $n = f - i + 1$

```
Algoritmo MergeSort (A, i, f )                                   T(n)
Entrada: vetor 'A' e índices 'i' e 'f' do vetor (i ≤ f ).
Saída: o vetor A, ordenado "in-loco" do índice 'i' até o índice 'f'.
{
    se ( i < f )   // Ou i ≠ f .
    {
        m := ⌊ (i + f ) / 2 ⌋;                                   T(n/2)
        MergeSort (A, i, m);                                     T(n/2)
        MergeSort (A, m + 1, f );
        Intercalar (A, i, m, f )                                 n - 1
    }
}                                                                T(1) = 0
```

Algoritmos "Dividir para Conquistar" dão origem a relações de recorrência "Dividir para Conquistar"

$$T(n) = 2T(n/2) + n - 1$$

$$T(1) = 0$$

Solução geral de relações de recorrência do tipo "dividir para conquistar" (Método da "Divisão e Conquista")

Teorema:
A solução da relação de recorrência $T(n) = aT(n/b) + cn^k$, onde a e b são constantes inteiras, $a \geq 1$, $b \geq 2$ e c e k constantes positivas, é

$$T(n) = \begin{cases} O(n^{\log_b a}) & se\ a > b^k \\ O(n^k \log n) & se\ a = b^k \\ O(n^k) & se\ a < b^k \end{cases}.$$

Desenvolvendo o Quick Sort

I) Interface

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

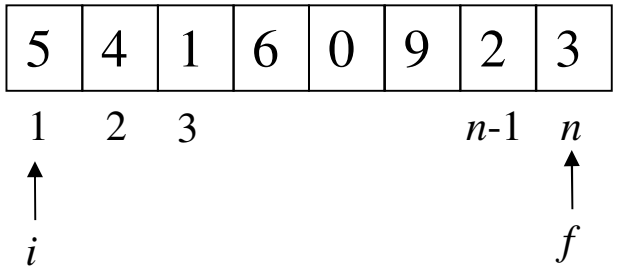
II) Significado

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

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

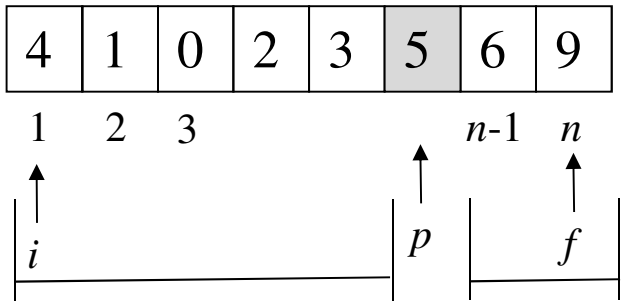
Desenvolvendo o Quick Sort

III) Redução



Escolher um pivô (qualquer elemento). Elegemos A[*i*].

Particionar o vetor colocando os elementos menores do que o pivô antes dele e os maiores após ele. Seja *p* a posição do pivô após o particionamento.



QuickSort (A, *i*, *p* - *1*) QuickSort (A, *p* + 1, *f*)

Desenvolvendo o Quick Sort

Comandos:

```
p := Partição (A, i, f, i);  // O quarto argumento indica a posição do elemento
                             // escolhido para pivô, antes do particionamento.  O
                             // algoritmo retorna a posição p do pivô após o
                             // particionamento.
QuickSort (A, i, p - 1);
QuickSort (A, p + 1, f )
```

IV) Base

É caracterizada por i = f (um elemento) e i > f (zero elemento). Verificar.

Comandos (para ordenar uma faixa de vetor com 0 ou 1 elemento):

Nenhum comando é necessário.

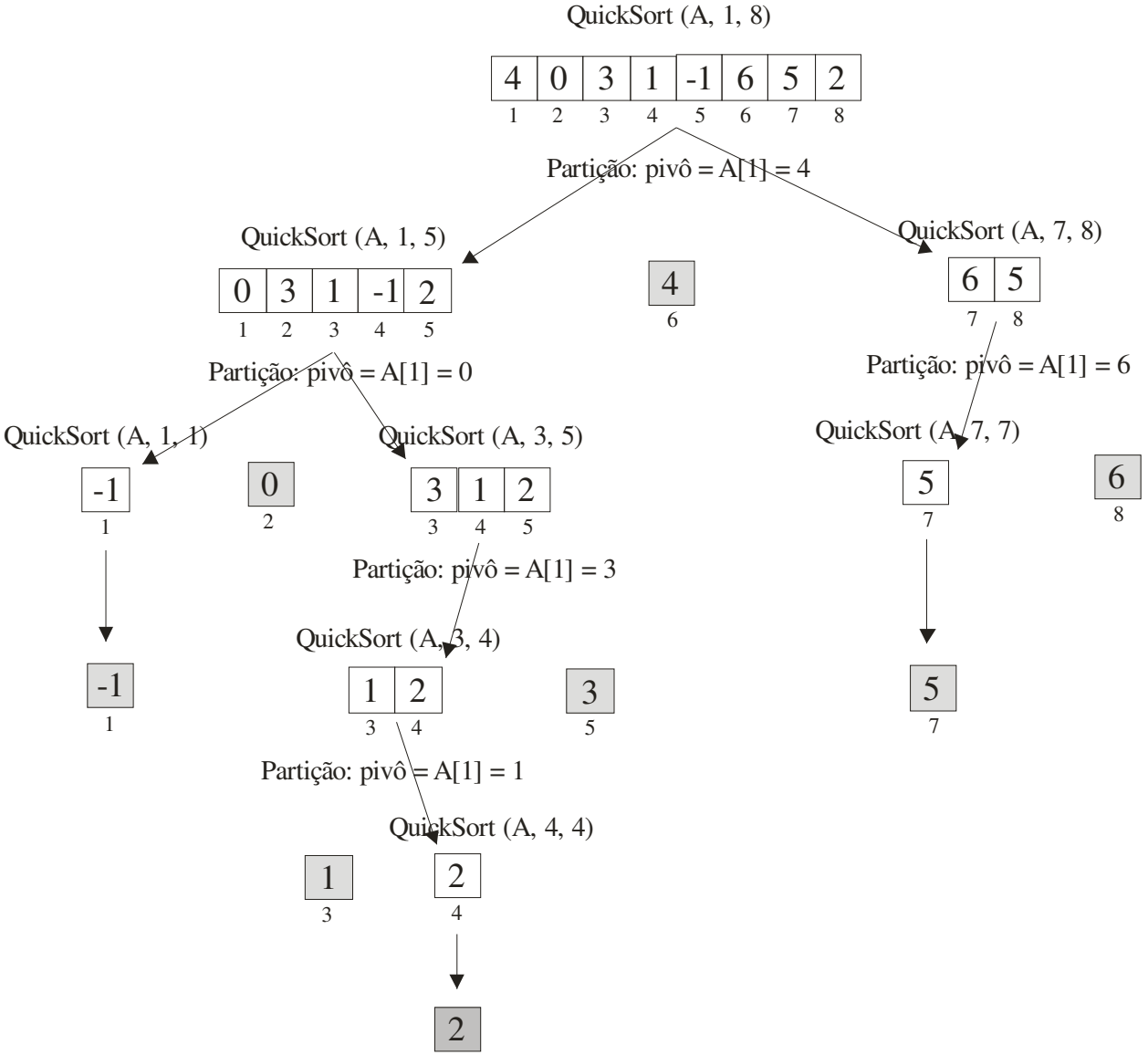
Desenvolvendo o Quick Sort

V) O Algoritmo

Algoritmo QuickSort (A, i, f)
Entrada: vetor ‘A’ e índices ‘i’ e ‘f’ do vetor .
Saída: o vetor A, ordenado “in-loco” do índice ‘i’ até o índice ‘f’.

```
{
    se ( i < f )
    {
        p := Partição (A, i, f, i);  // O quarto argumento indica a posição do elemento escolhido para pivô,
                                     // antes do particionamento.  O algoritmo retorna a posição p do pivô após
                                     // o particionamento.
        QuickSort (A, i, p - 1);
        QuickSort (A, p + 1, f )
     }
}
```

Ilustrando o funcionamento do Quick Sort



Complexidades do Quick Sort

Seja a quantidade de elementos $n = f - i + 1$

Pior caso

Ocorre quando o pivô divide o vetor em dois subvetores, um com zero elemento e outro com $n - 1$ elementos

Melhor caso

Ocorre quando o pivô escolhido divide o vetor em dois subvetores de tamanho iguais a $n/2$.

Caso médio

É a média de todos os possíveis casos de posicionamento do pivô após a partição.

Complexidade (pior caso)

```
Algoritmo QuickSort (A, i, f )
Entrada: vetor 'A' e índices 'i' e 'f' do vetor .
Saída: o vetor A, ordenado "in-loco" do índice 'i' até o índice 'f'.
{
    se ( i < f )
    {
        p := Partição (A, i, f, i);
        QuickSort (A, i, p - 1);
        QuickSort (A, p + 1, f )
    }
}
```

T(*n*)

n - 1

T(0), digamos.

T(*n* - 1), digamos.

T(1) = T(0) = 1

Complexidade (pior caso)

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

$$T(0) = 1$$

Resolvendo:

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

Complexidade (melhor caso)

```
Algoritmo QuickSort (A, i, f )
Entrada: vetor 'A' e índices 'i' e 'f' do vetor .
Saída: o vetor A, ordenado "in-loco" do índice 'i' até o índice 'f'.
{
    se ( i < f )
    {
        p := Partição (A, i, f, i);
        QuickSort (A, i, p - 1);
        QuickSort (A, p + 1, f )
    }
}
```

T(*n*)

n - 1

T(*n*/2)

T(*n*/2).

T(1) = T(0) = 1

Complexidade (melhor caso)

$$T(n) = 2T(n/2) + n - 1$$

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

Resolvendo (r. r. “dividir para conquistar”):

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

Complexidade (caso médio)

Seja q a quantidade de elementos antes da posição do pivô.

```
Algoritmo QuickSort (A, i, f )
Entrada: vetor 'A' e índices 'i' e 'f' do vetor .
Saída: o vetor A, ordenado "in-loco" do índice 'i' até o índice 'f'.
{
    se ( i < f )
    {
        p := Partição (A, i, f, i);
        QuickSort (A, i, p - 1);
        QuickSort (A, p + 1, f )
    }
}
```

T(n)

n - 1

T(q)

T($n - q - 1$).

T(1) = T(0) = 0

Complexidade (caso médio)

q	Complexidade do caso	Probabilidade
0	$T(0) + T(n-1) + n - 1$	$1/n$
1	$T(1) + T(n-2) + n - 1$	$1/n$
2	$T(2) + T(n-3) + n - 1$	$1/n$
...	...	$1/n$
n - 3	$T(n-1) + T(0) + n - 1$	$1/n$
n - 2	$T(n-2) + T(1) + n - 1$	$1/n$
n - 1	$T(n-1) + T(0) + n - 1$	$1/n$

Complexidade (caso médio)

$$T(n) = \frac{n(n-1) + 2T(0) + 2T(1) + \ldots + 2T(n-2) + 2T(n-1)}{n}$$

$$T(n) = n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} T(i)$$

A Solução desta relação de recorrência é
T(n) = O (n log (n)).

Partição

I) Interface

Partição (A, i, f, p)

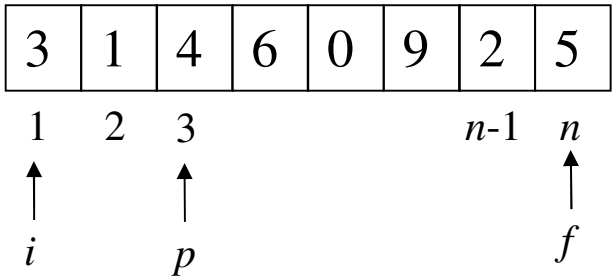
II) Significado

Particiona “in-loco” o vetor A tendo como pivô o elemento da posição p. A partição será realizada do índice i até o índice f, sendo que $i \leq p \leq f$. Retorna a posição final do pivô.

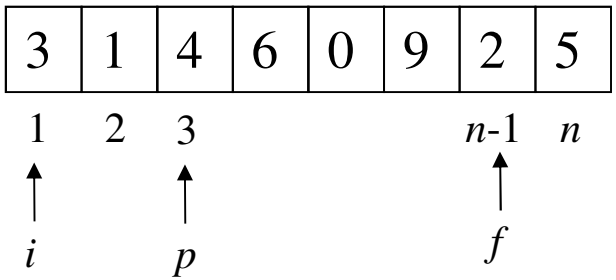
Obs.: o algoritmo QuickSort foi originalmente proposto por:
HOARE, C. A. R. Quicksort, Computer Journal 5 (1), 1962 10-15.

Partição

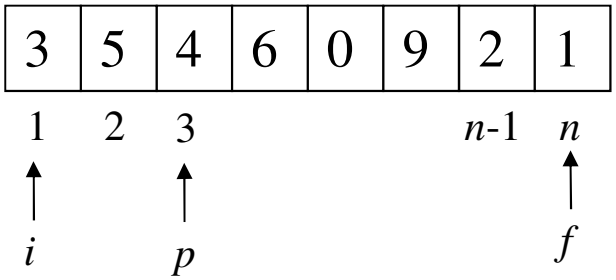
III) Redução



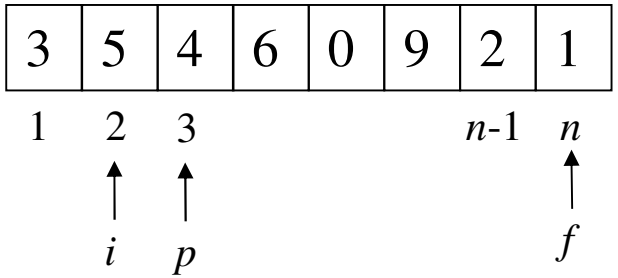
Se A[*f*] > A[*p*], então A[*f*] está na posição correta em relação ao pivô e o tamanho do problema pode ser diminuído em 1.



Partição

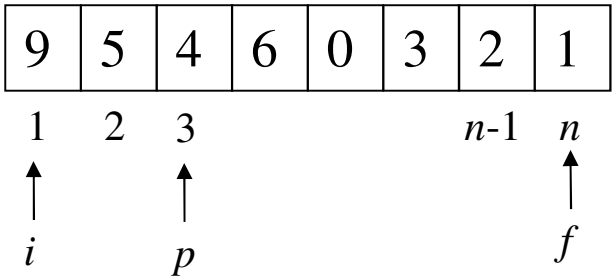


Se A[i] < A[p], então A[i] está na posição correta em relação ao pivô e o tamanho do problema também pode ser diminuído em 1.

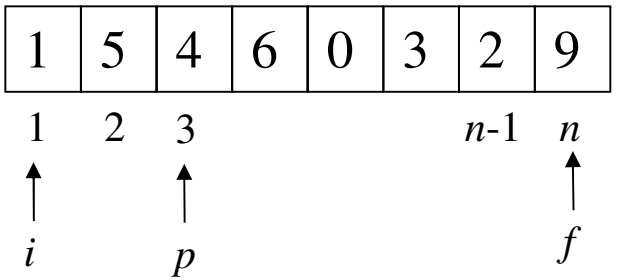


Partição

Neste caso não ocorre de A[i] < A[p] e de A[f] > A[p].



Troca-se A[i] de posição com A[f] (poder-se-ia diminuir o tamanho do problema, mas não faremos isto nesta versão)



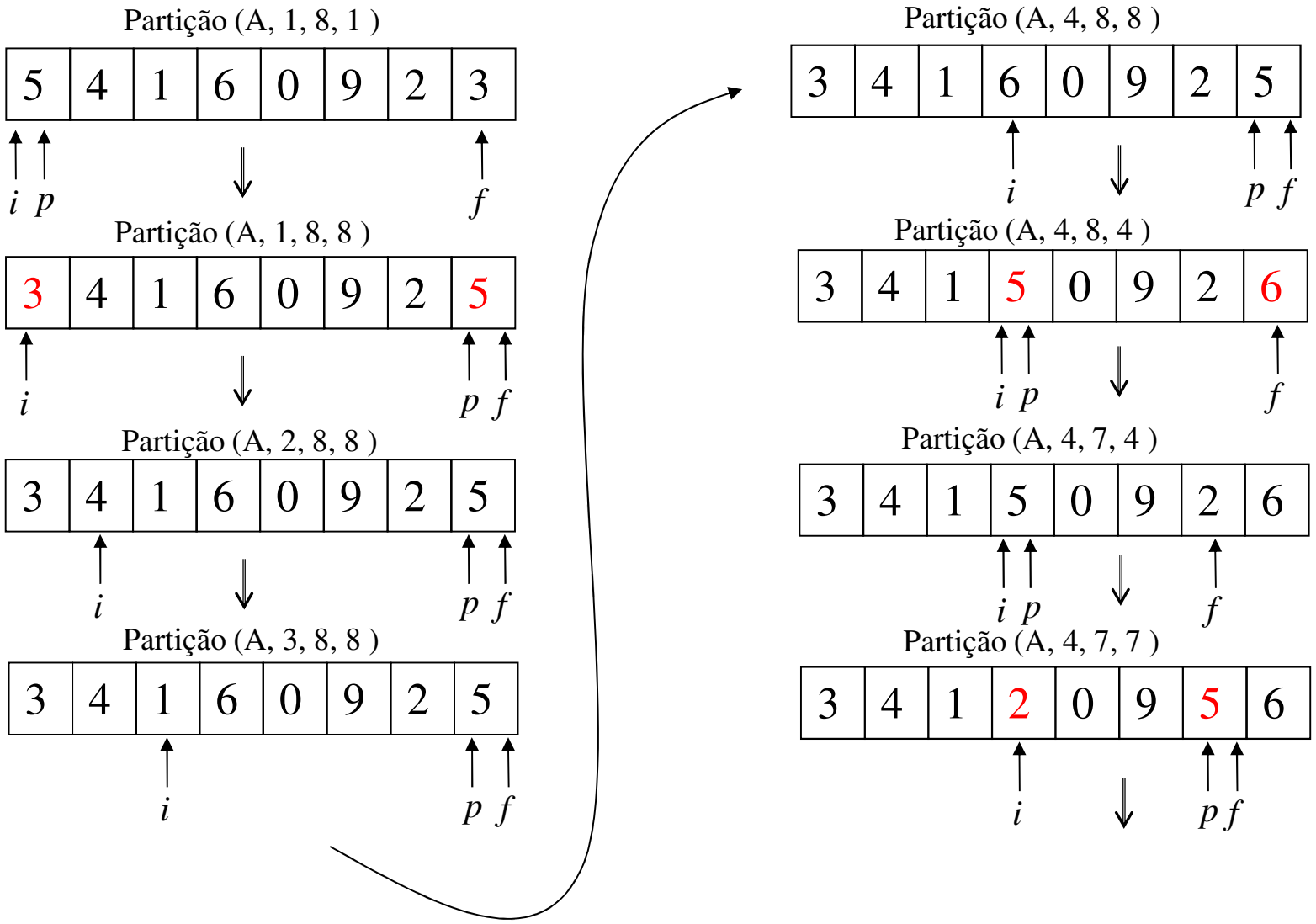
Partição

Comandos:

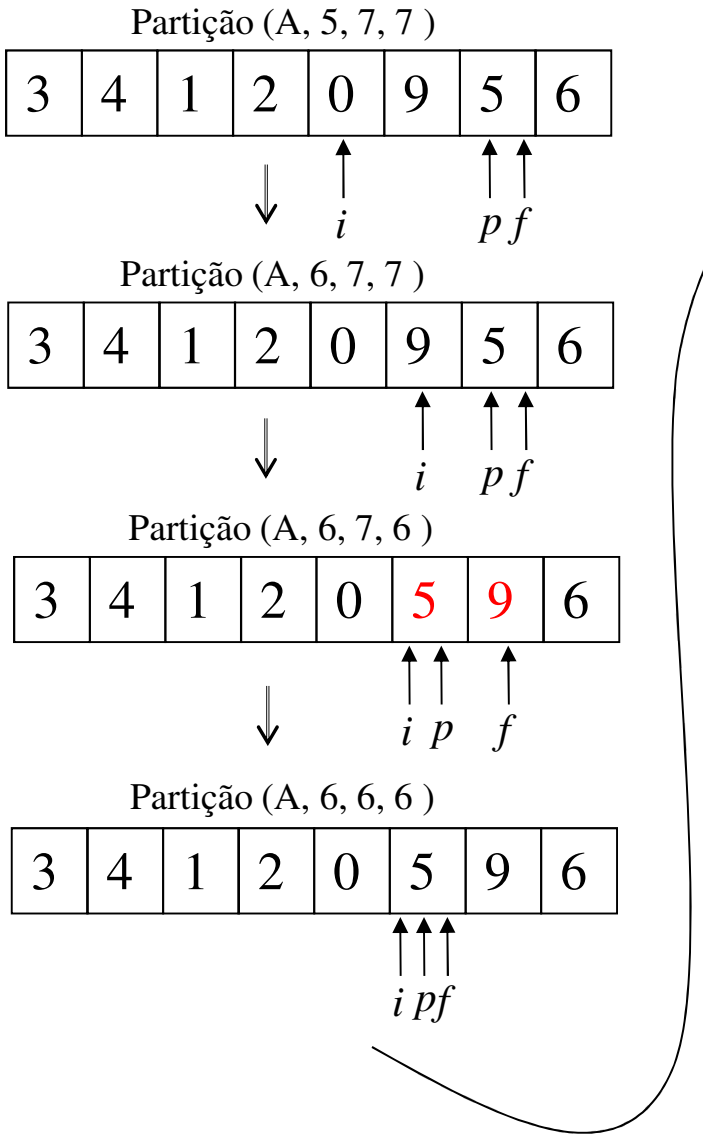
```
se ( A[f ] > A[p] ) f := f – 1
senão
      se ( A [i] < A [p] ) i := i + 1
      senão {
           troca := A[i]; A [i] := A[f]; A[f] := troca;
          // troca o ponteiro p do pivô se o pivô for movimentado.
          se (p = i) p := f senão se (p = f) p := i
       }

retornar Partição (A, i, f, p)
```

Partição

IV) Base - Descobrindo qual é a base:



Partição



A base é caracterizada por i = f (a faixa compreende 1 elemento.

Pode-se dizer, neste caso que o vetor está particionado.

Comandos:

retornar p

Partição

V) O Algoritmo

Algoritmo Partição (A, i, f, p)

Entrada: vetor ‘A’, índices ‘i’ e ‘f’ do vetor e ‘p’, a posição do pivô .

Saída: Particiona “in-loco” o vetor A em torno do pivô da posição dada por p. Retorna a posição do pivô após o particionamento.

```
{
    se (i = f )  retornar p
    senão
    {
        se ( A[f ] > A[p] ) f := f – 1
        senão
            se ( A [i] < A [p] ) i := i + 1
            senão {
                troca := A[i]; A [i] := A[f]; A[f] := troca;
                // troca o ponteiro p do pivô se o pivô for movimentado.
                se (p = i) p := f senão se (p = f) p := i
            }
        retornar Partição (A, i, f, p)
    }
}
```

Para criar novos algoritmos de ordenação, proponha outras reduções.

Seja criativo!

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 $n = f - i + 1$.

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

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

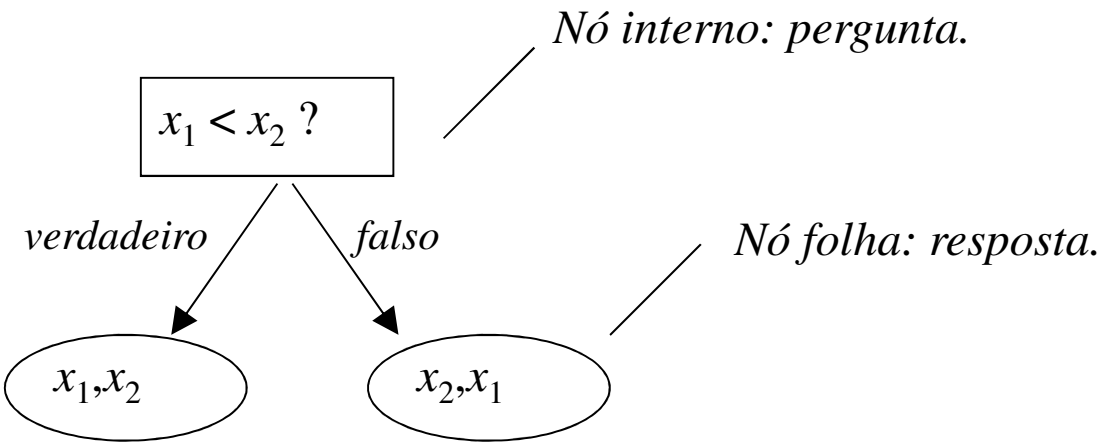
Logo:

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

Cota inferior para ordenação

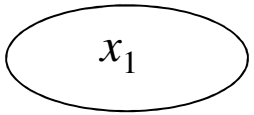
- Algoritmos baseados em comparação.
- Prove que qualquer algoritmo de ordenação baseado em comparação tem complexidade mínima de $\Omega(n \log n)$.

Modelo de computação: árvore de decisão

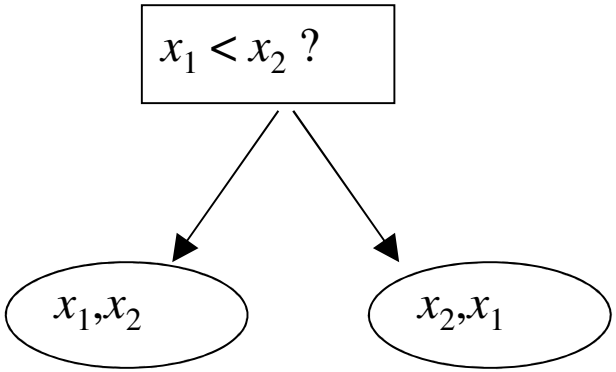


$n = 1$ e $n = 2$

$n = 1$: { x_1 }

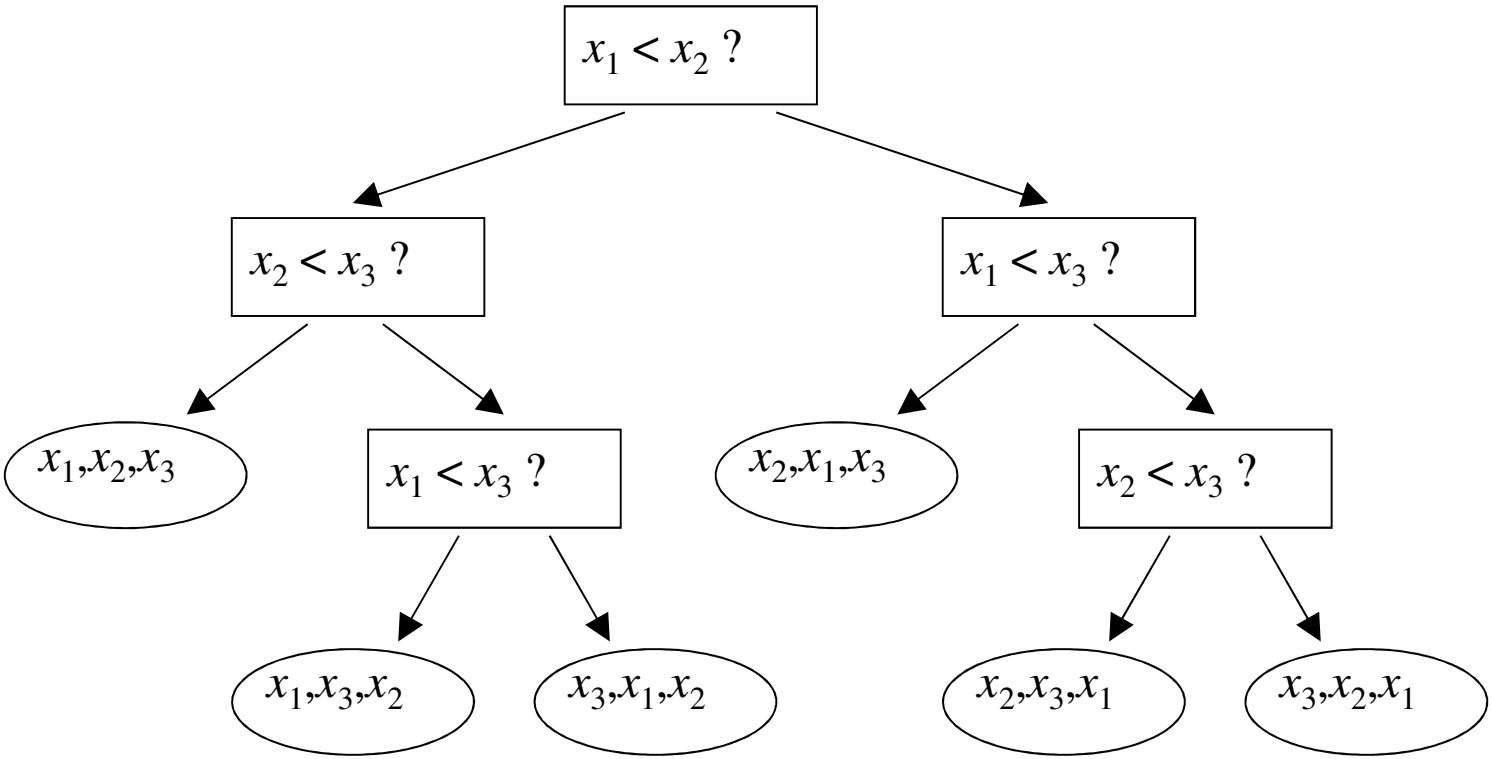


$n = 2$: { x_1, x_2 }



$n = 3$

$n = 3$: { 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: $log\ (n!)$.
- $log\ (n!) = \Omega\ (n \log n)$.

Aproximação de Stirling.

- 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*.

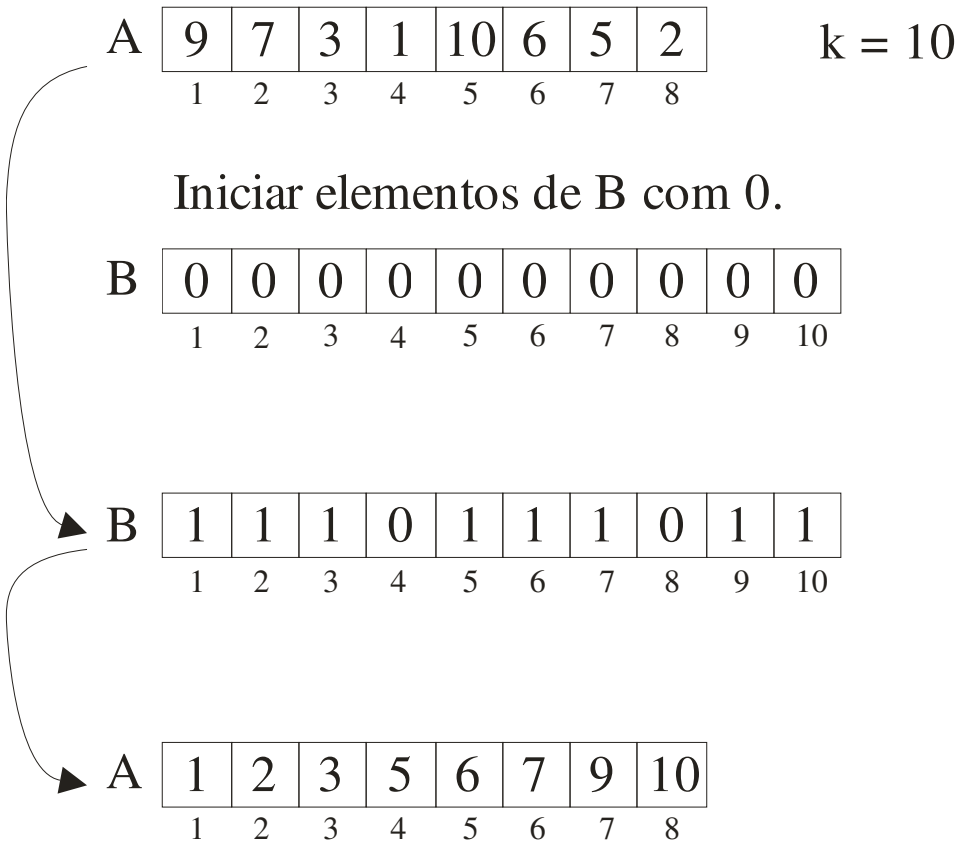
Bucket Sort

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 do Bucket Sort

```
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
     }
```

$O(k)$

$O(n)$

$O(k)$

Concluindo

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

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

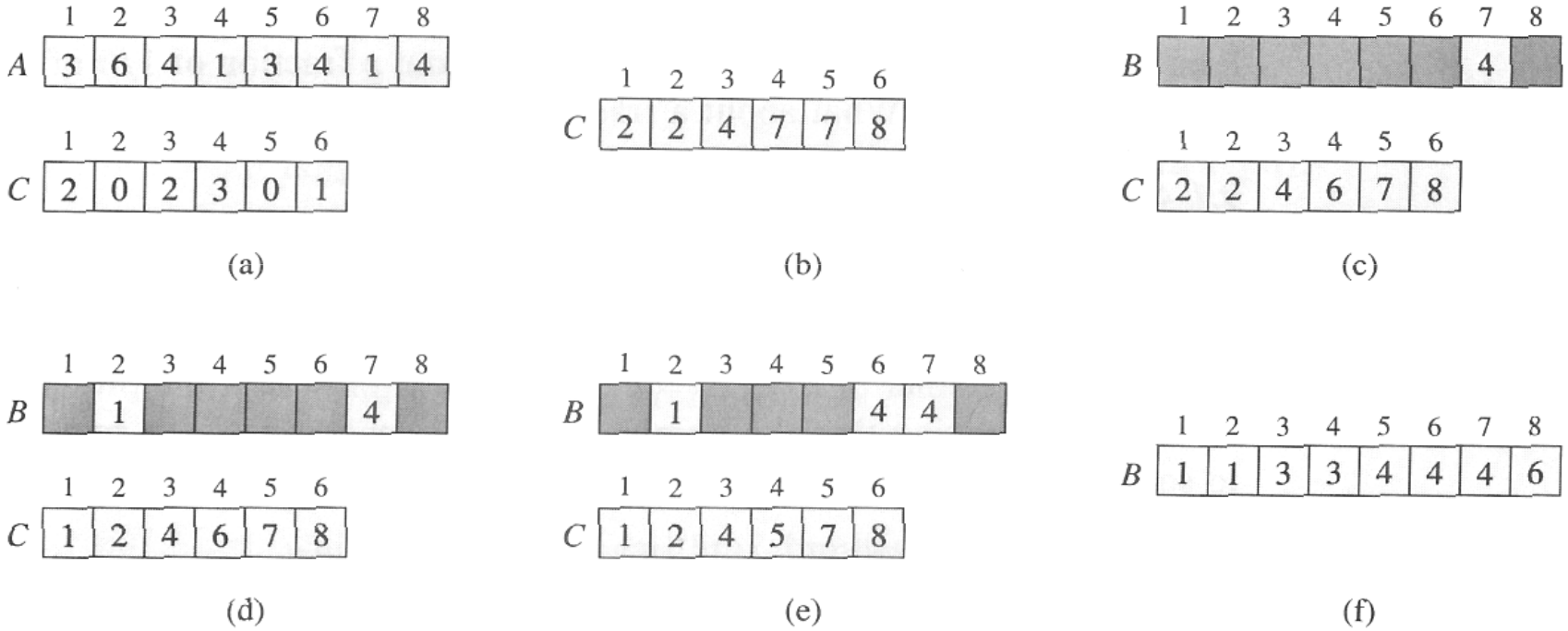
Se $k >>>> n$ então as complexidades de tempo e de espaço do algoritmo podem ser grandes.

Counting Sort

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

Ideia

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



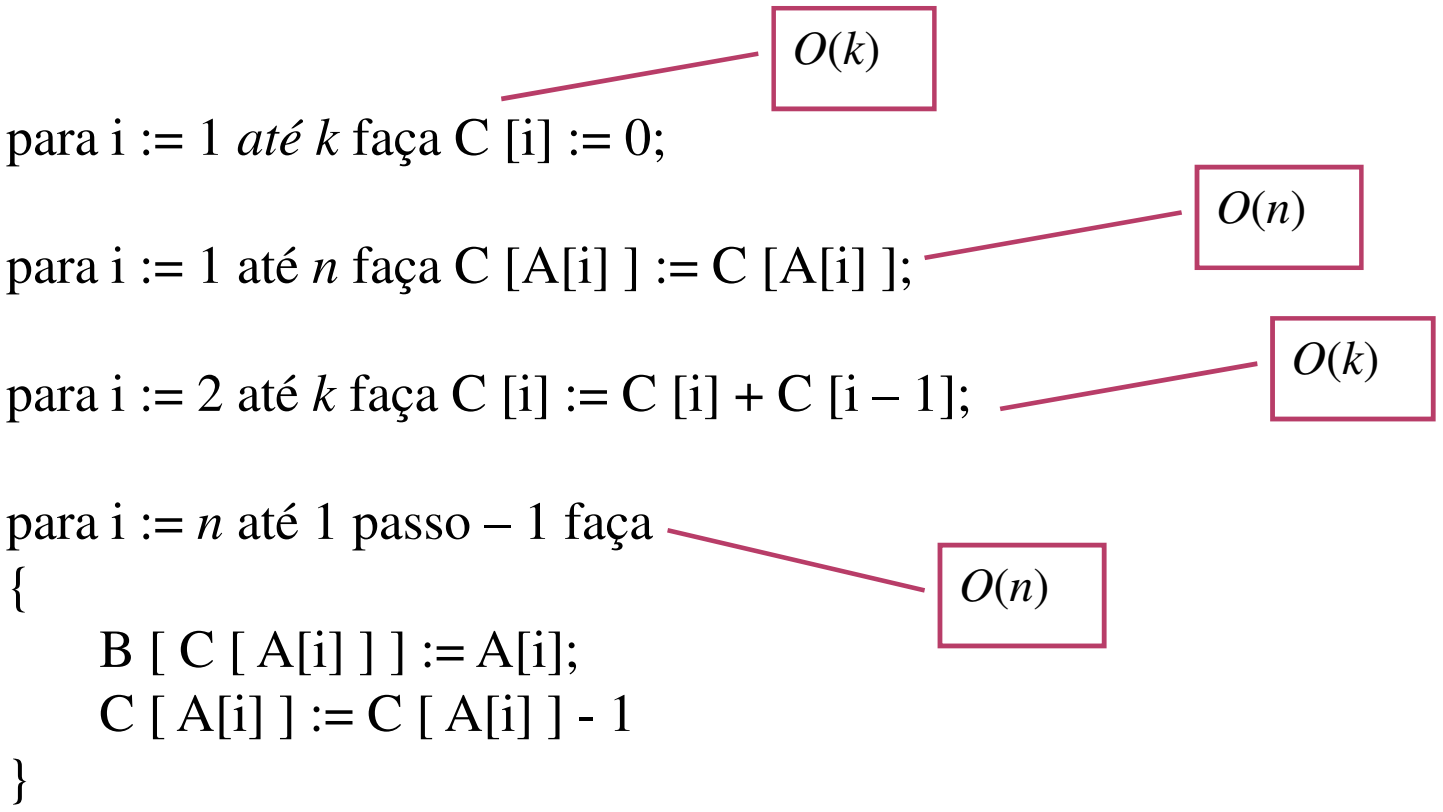
Fonte: CORMEN, T.; LEISERSON, C.; RIVEST, R.; STEIN, C. Introduction to Algorithms. New York: MIT Press, 2004.

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] ] + 1;
    // 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

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

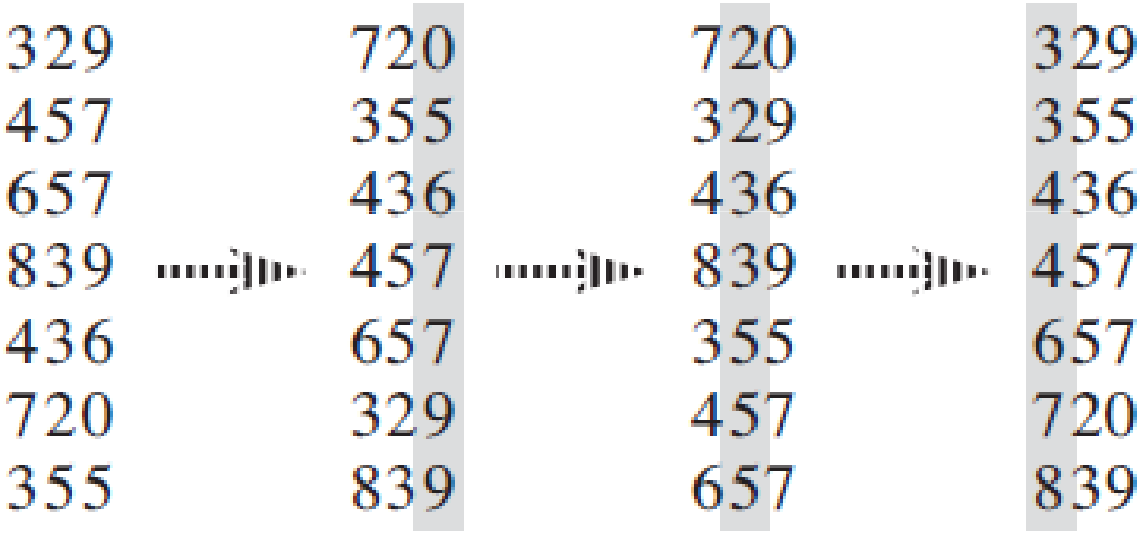
Se $k = O(n)$ então $T(n) = O(n)$.

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 Counting Sort.

  - 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): $O\ (log\ n)$

Obs.: A discussão destes algoritmos já foi feita na disciplina “Projeto e Análise de Algoritmos I.

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.