Análise de Algoritmos e Complexidade da Computação

Ordenação e Busca

Prof. Dr. Osvaldo Luiz de Oliveira

Estas anotações devem ser complementadas por apontamentos em aula.

Diferentes reduções, diferentes algoritmos

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.

Desenvolvendo o Insertion Sort

I) Interface

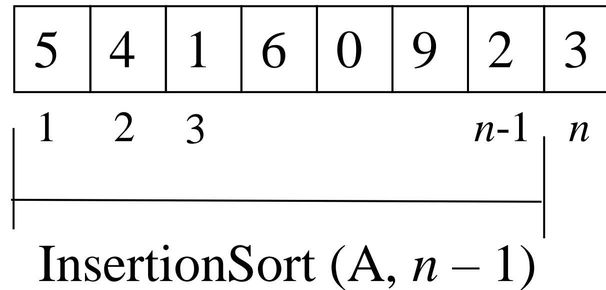
InsertionSort (A, n)

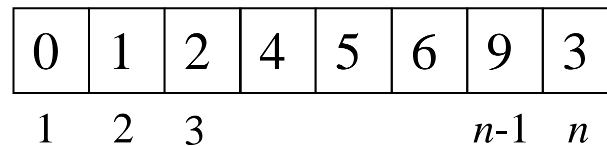
II) Significado

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

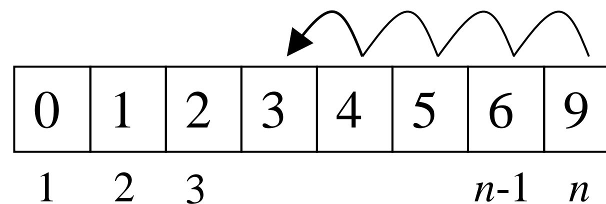
Desenvolvendo o Insertion Sort

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

A

4	0	3	1	-1	6	5	2
1	2	3	4	5	6	7	8

InsertionSort (A, 8)
InsertionSort (A, 7)
...
InsertionSort (A, 1)

4
1

← Inserir 0

0	4
1	2

← Inserir 3

0	3	4
1	2	3

← Inserir 1

0	1	3	4
1	2	3	4

← Inserir -1

-1	0	1	3	4
1	2	3	4	5

← Inserir 6

...

-1	0	1	3	4	5	6
1	2	3	4	5	6	7

← Inserir 2

-1	0	1	2	3	4	5	6
1	2	3	4	5	6	7	8

Complexidade do Insertion Sort

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
        }
    }
}
```

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

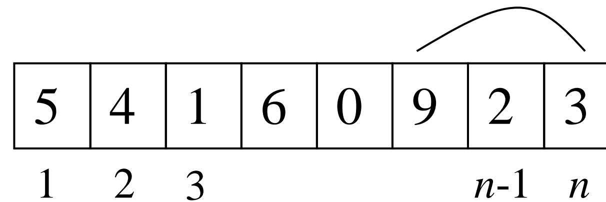
SelectionSort (A, n)

II) Significado

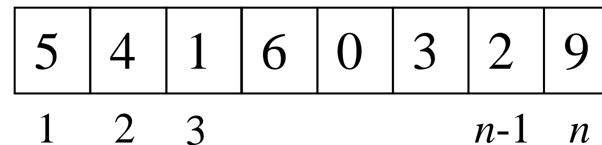
Ordena “in-loco” o vetor A de n elementos.

Desenvolvendo o Selection Sort

III) Redução (seleção do maior)



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



SelectionSort (A, $n - 1$)

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

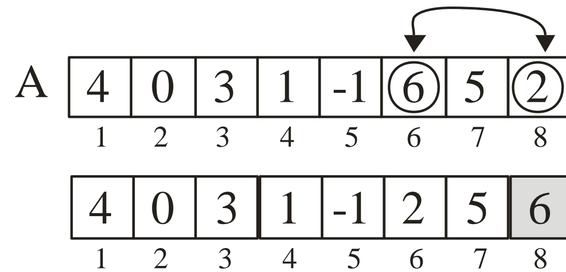
Algoritmo SelectionSort (A, n)

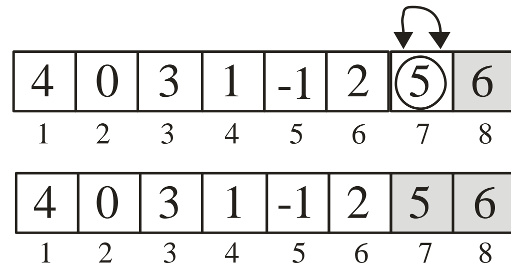
Entrada: vetor A de n elementos, $n \geq 1$.

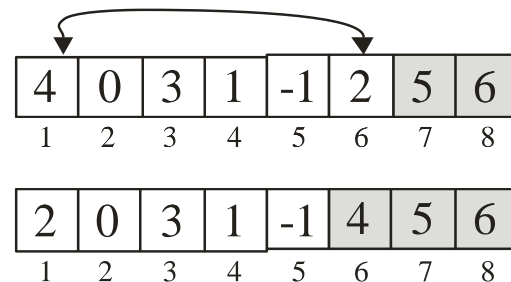
Saída: o vetor A, ordenado "in-loco".

```
{
    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







...

Complexidade do Selection Sort

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)
    }
}
```

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

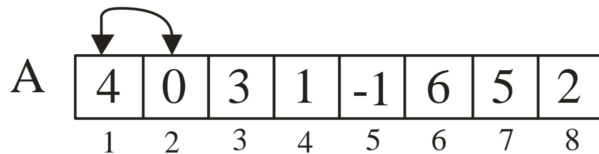
BubbleSort (A, *n*)

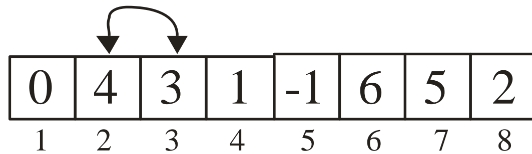
II) Significado

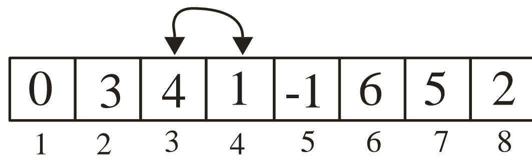
Ordena "in-loco" o vetor A de *n* elementos.

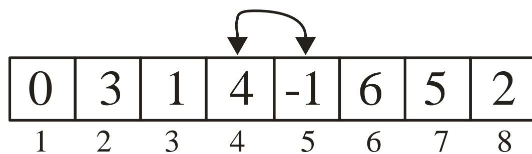
Desenvolvendo o Bubble Sort

III) Redução

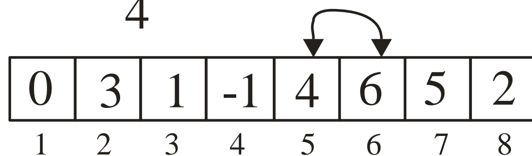


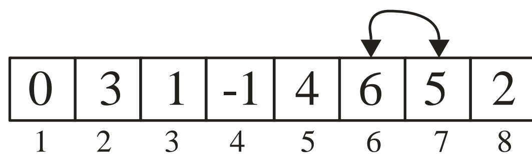


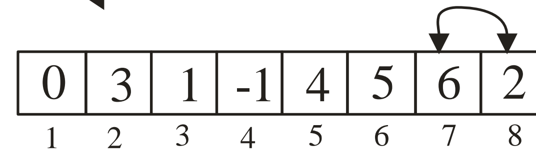


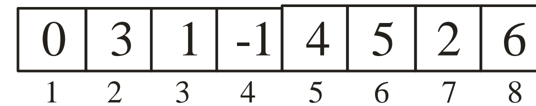


4









BubbleSort (A, $n-1$)

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)

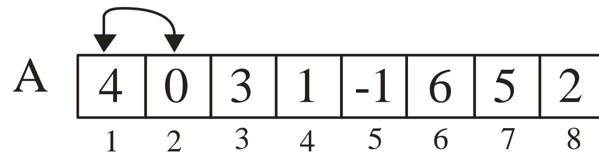
Entrada: vetor A de n elementos, $n \geq 1$.

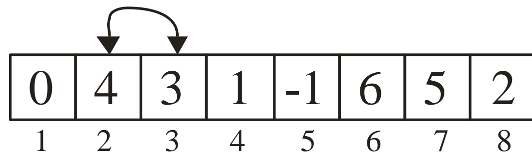
Saída: o vetor A, ordenado “in-loco”.

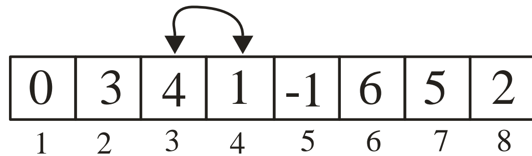
```
{
    se ( n > 1 )
    {
        para i := 1 até n – 1 faça
            se ( A[i] < A[i+1] )
            {
                troca := A[i];  A [i] := A[i+1];  A[i+1] := troca
            }
        BubbleSort (A , n – 1)
    }
}
```

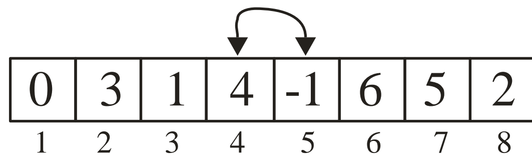
Ilustrando o funcionamento do Bubble Sort

BubbleSort (A, 8)

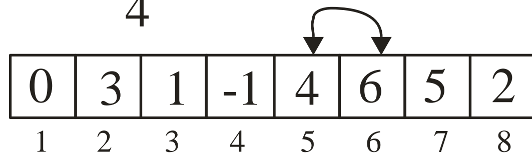


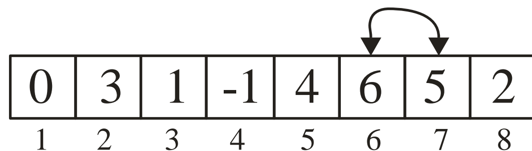


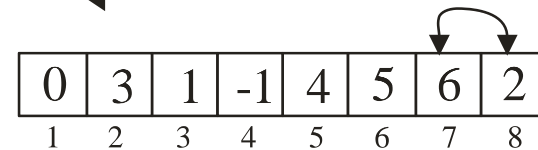


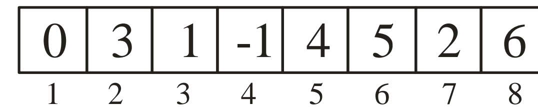


4

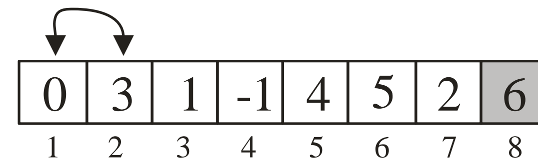








BubbleSort (A, 7)



...

Complexidade do BubbleSort

T(n)

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)
    }
}
```

n - 1

T(n – 1)

T(1) = 1

Complexidade do Bubble Sort

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

$T(1) = 1$

Resolvendo:

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

Merge Sort

Ver slides em
“Design e Análise de Algoritmos por indução”.

Desenvolvendo o Quick Sort

I) Interface

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

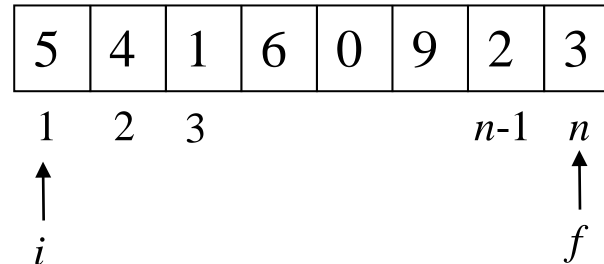
II) Significado

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.

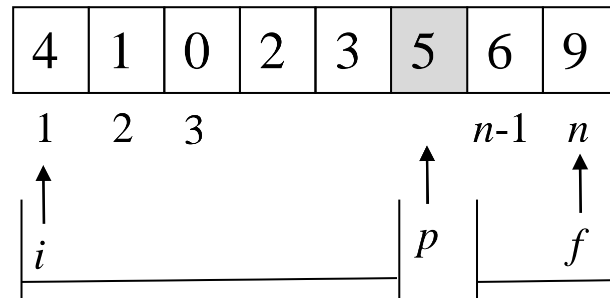
Desenvolvendo o Quick Sort

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):

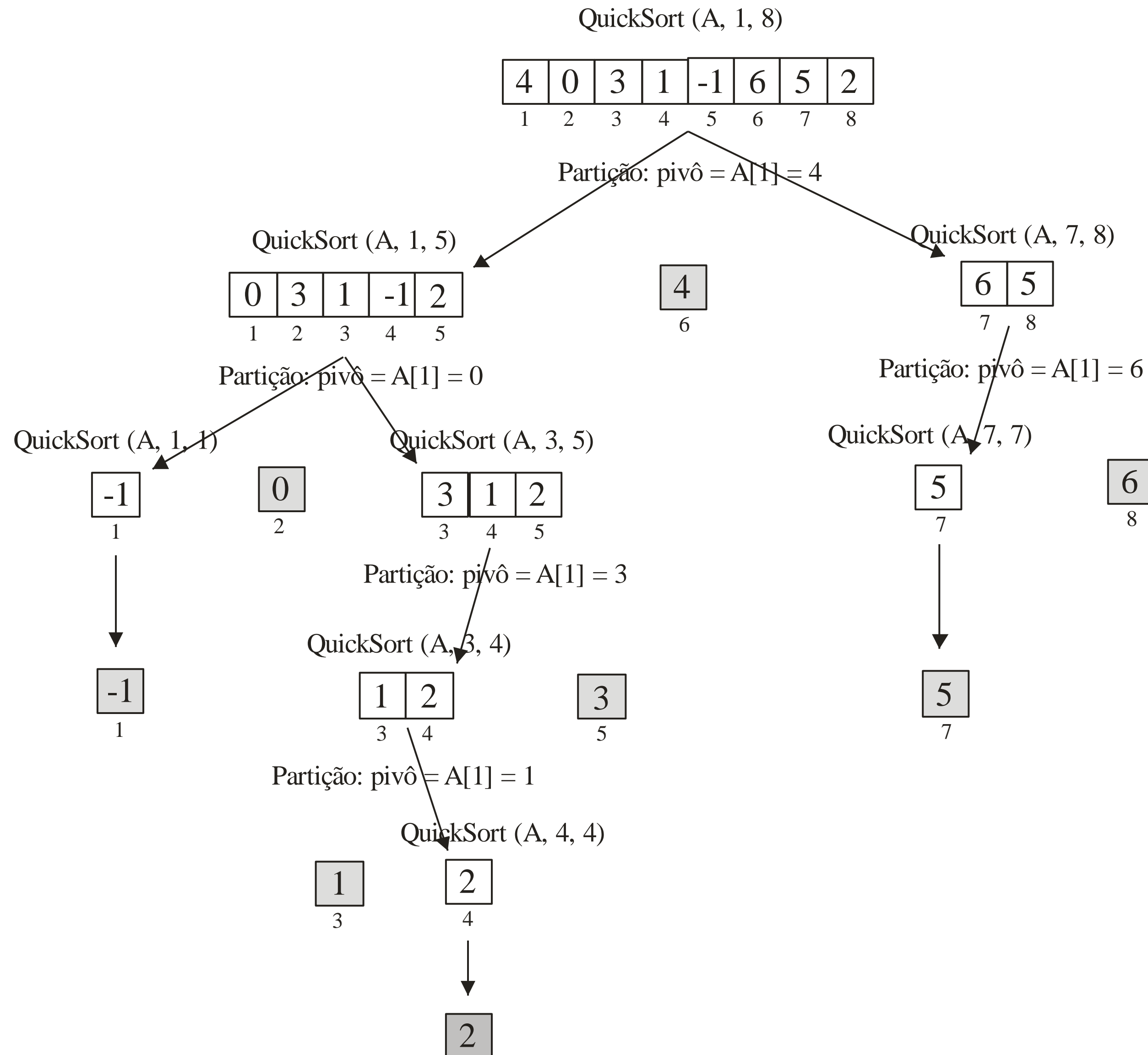
Nenhum comando é necessário.

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 )
    }
}
```

- Algoritmo QuickSort (A, i, f): $T(n)$
- p := Partição (A, i, f, i): $n - 1$
- QuickSort (A, i, p - 1): $T(0)$, digamos.
- QuickSort (A, p + 1, f): $T(n - 1)$, 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 )
    }
}
```

Anotações:

- Algoritmo QuickSort (A, i, f): $T(n)$
- p := Partição (A, i, f, i): $n - 1$
- QuickSort (A, i, p - 1): $T(n/2)$
- QuickSort (A, p + 1, f): $T(n/2)$.
- }: $T(1) = T(0) = 1$

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	$T(n-3) + T(2) + n - 1$	$1/n$
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 pode ser feita pelo Método da Substituição (veja slides finais de “Design de Análise de Algoritmos por Indução”).

Solução: T(n) = O (n log (n)).

Partição

I) Interface

Partição (A, i, f, p)

II) Significado

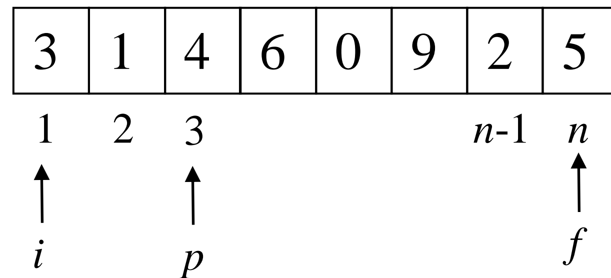
Particiona "in-loco" o vetor A tendo como pivô o elemento da posição p. A partição será realizada do índice i até o índice f, sendo que $i \leq p \leq f$. Retorna a posição final do pivô.

Obs.: o algoritmo QuickSort foi originalmente proposto por:

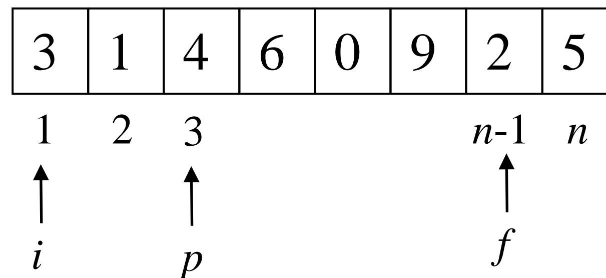
HOARE, C. A. R. Quicksort, Computer Journal 5 (1), 1962 10-15.

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

3	5	4	6	0	9	2	1
1	2	3				n-1	n
↑ i		↑ p					↑ f

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.

3	5	4	6	0	9	2	1
1	2	3				n-1	n
	↑ i	↑ p					↑ f

Partição

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

9	5	4	6	0	3	2	1
1	2	3				n-1	n
↑ i		↑ p					↑ f

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

1	5	4	6	0	3	2	9
1	2	3				n-1	n
↑ i		↑ p					↑ f

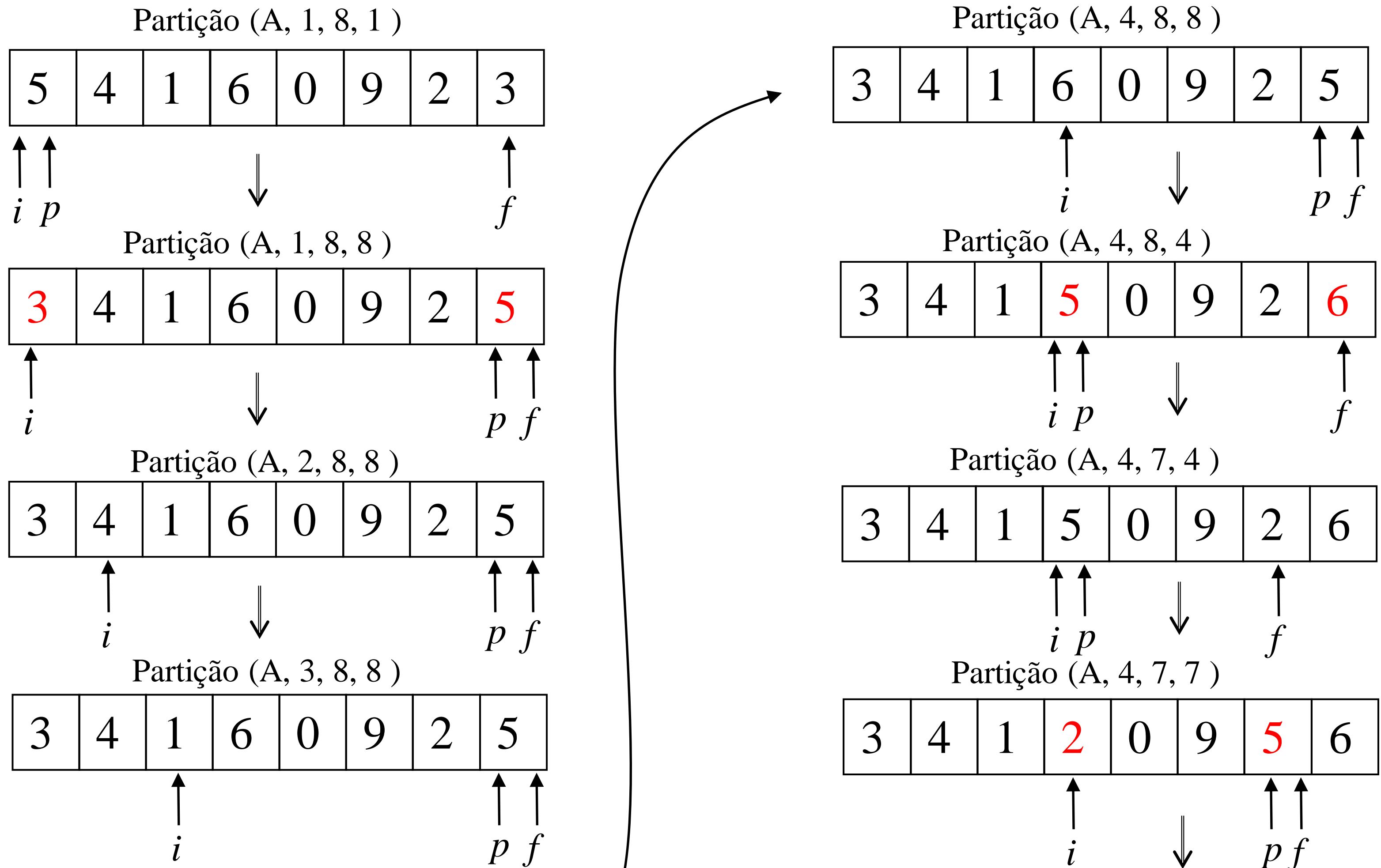
Partição

Comandos:

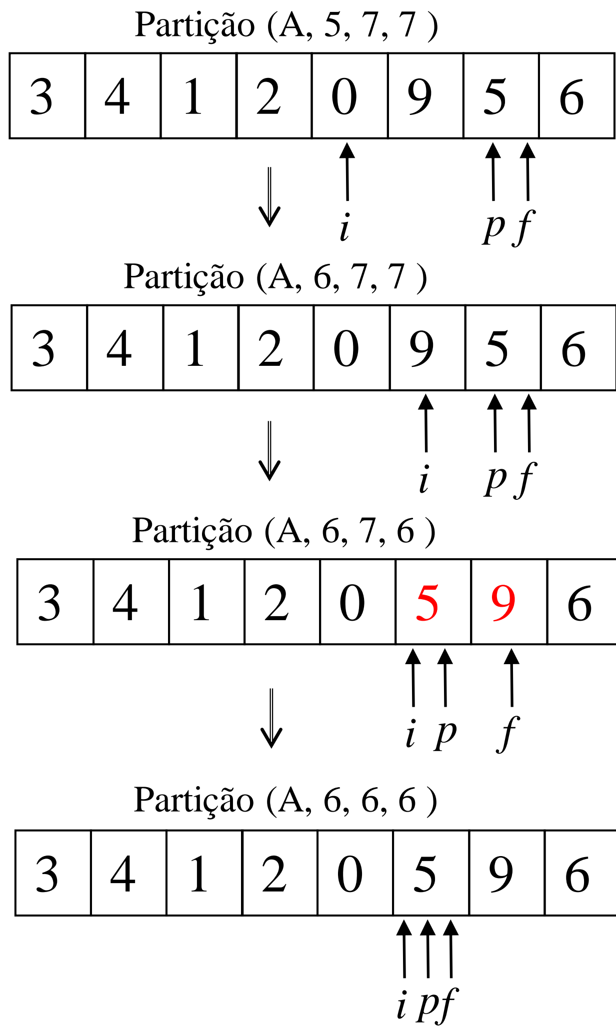
```
se ( A[f ] > A[p] ) f := f – 1
senão
    se ( A [i] < A [p] ) i := i + 1
    senão {
        troca := A[i]; A [i] := A[f]; A[f] := troca;
        // troca o ponteiro p do pivô se o pivô for movimentado.
        se (p = i) p := f senão se (p = f) p := i
    }

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 ≥ 1, f ≥ 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:

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

Busca

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.

Busca

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

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

Obs.: A discussão destes algoritmos foi feita em “Design e Análise de Algoritmos por Indução”.

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).