

BandTec
DIGITAL SCHOOL



ED

Estrutura de Dados e Armazenamento

Ordenação

(Selection Sort, Bubble Sort, Insertion Sort,
MergeSort, QuickSort)

Pesquisa Binária

© Profa. Célia Taniwaki

Ordenação

- Muitas vezes, é preciso ordenar os dados para poder manipular esses dados de uma forma organizada.
 - Por exemplo:
 - Nomes em ordem alfabética
 - Alunos por ordem do RA ou por Nota
- A busca de um elemento em uma lista ordenada é mais eficiente (Pesquisa binária)

Algoritmos de ordenação

- Existem inúmeros algoritmos de ordenação
- Em inglês a palavra sort significa ordenar ou classificar
- O link abaixo é de um vídeo que compara 9 algoritmos de ordenação:

<https://www.youtube.com/watch?v=ZZuD6iUe3Pc>

1. SelectionSort

2. Shell Sort

3. Insertion Sort

4. Merge Sort

5. Quick Sort

6. Heap Sort

7. Bubble Sort

8. Comb Sort

9. Cocktail Sort

- Os algoritmos realçados serão os que veremos com detalhes.
- Esse vídeo utiliza 4 tipos de dados de entrada:
 - Random (aleatório)
 - Few unique (muitos dados duplicados)
 - Reversed (ordem inversa)
 - Almost sorted (praticamente ordenado)
- Assista o vídeo para ver quais algoritmos são mais eficientes e quais são menos

Métodos de ordenação simples

- Entre os 5 que vamos estudar, esses 3 são os algoritmos simples e intuitivos:
 - Selection sort – ordenação por seleção
 - Bubble sort – ordenação por troca
 - Insertion sort – ordenação por inserção
- Características:
 - Fácil implementação

Selection Sort

- Método simples de seleção
 - Percorre uma vez o vetor, procurando o menor valor . Troca o 1º elemento com o menor valor selecionado.
 - Percorre do 2º elemento em diante para procurar o 2º menor valor. Troca o 2º elemento com o 2º menor valor selecionado.
 - E assim sucessivamente.
- Características
 - Ordena através de sucessivas seleções do elemento de menor valor (ou de maior valor) em um segmento não ordenado do vetor e seu posicionamento no final de um segmento ordenado
 - Realiza uma busca sequencial pelo menor valor (ou maior valor) no segmento não ordenado a cada iteração

Selection Sort - Exemplo

Sejam os dados (ordenados / não ordenados):

4 7 5 2 8 1 6 3

⇒ menor: 1 – troca com o 1º 1 7 5 2 8 4 6 3

⇒ menor: 2 – troca com o 2º 1 2 5 7 8 4 6 3

⇒ menor: 3 – troca com o 3º 1 2 3 7 8 4 6 5

⇒ menor: 4 – troca com o 4º 1 2 3 4 8 7 6 5

⇒ menor: 5 – troca com o 5º 1 2 3 4 5 7 6 8

⇒ menor: 6 – troca com o 6º 1 2 3 4 5 6 7 8

⇒ menor: 7 – já está ok 1 2 3 4 5 6 7 8

Selection Sort - Algoritmo

```
selectionSort (int[] v)
início
    inteiro i, j, min;
    para i de 0 até v.length-2 (inclusive) faça
        início
            min ← i;
            para j de i+1 até v.length-1 (inclusive) faça
                se v[j] < v[min]
                    então min ← j;
                troca (v[i], v[min]);
            fim
        fim
    // onde está troca(v[i],v[min]) significa que os valores
    // de v[i] e v[min] devem ser trocados (isso pode ser
    // feito com 3 instruções e uma variável auxiliar)
```

Bubble Sort

- Método simples de troca
 - Compara elementos vizinhos do vetor. Se o anterior for maior do que o próximo, troca-os de lugar
- Características
 - Realiza varreduras no vetor, trocando pares adjacentes (vizinhos) de elementos, sempre que o próximo elemento for menor que o anterior
 - Após uma varredura, o maior elemento está corretamente posicionado no vetor e não precisa mais ser comparado. Ordena através de sucessivas trocas entre pares de elementos do vetor

Bubble Sort - Exemplo

Sejam os dados (**desordenados** / **ordenados**):

$\Rightarrow 4 > 7 ?$ – não, não troca

$\Rightarrow 7 > 5 ?$ – sim, troca

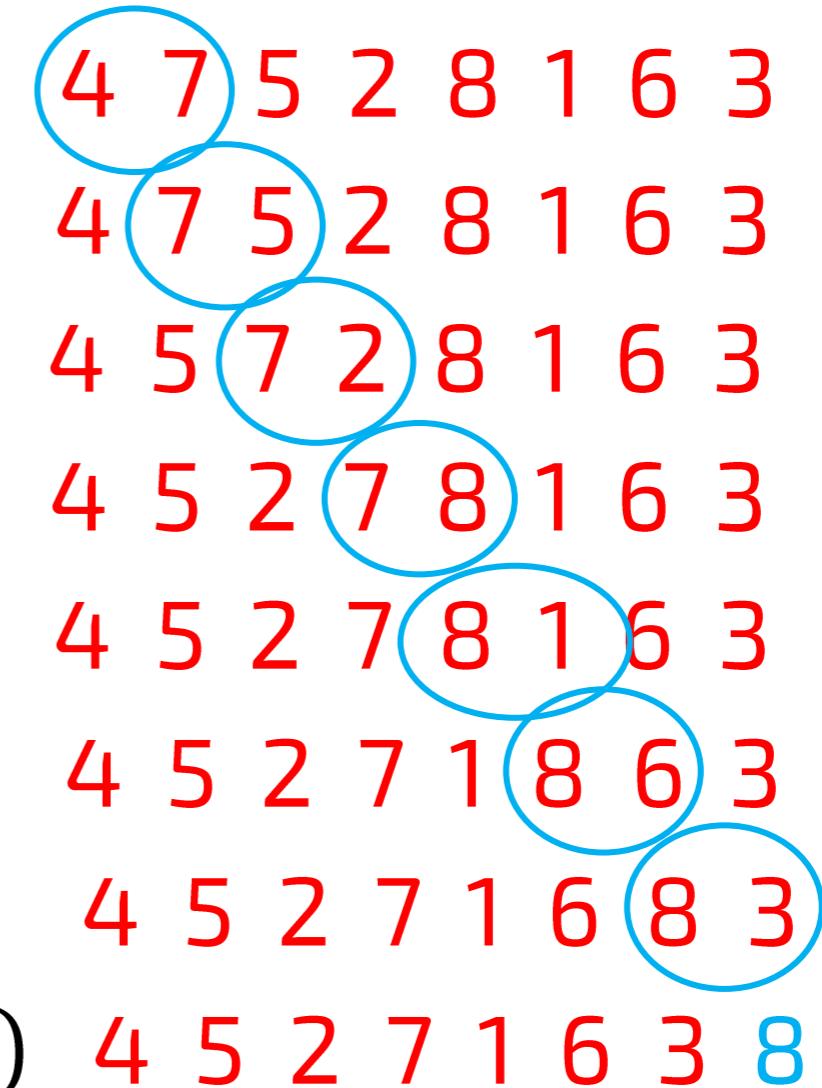
$\Rightarrow 7 > 2 ?$ – sim, troca

$\Rightarrow 7 > 8 ?$ – não, não troca

$\Rightarrow 8 > 1 ?$ – sim, troca

$\Rightarrow 8 > 6 ?$ – sim, troca

$\Rightarrow 8 > 3 ?$ – sim, troca (8 fica certo)



O maior valor “sobe” como se fosse uma bolha (bubble)

Bubble Sort - Exemplo

Continuando (desordenados / ordenados):

$\Rightarrow 4 > 5 ?$ – não, não troca

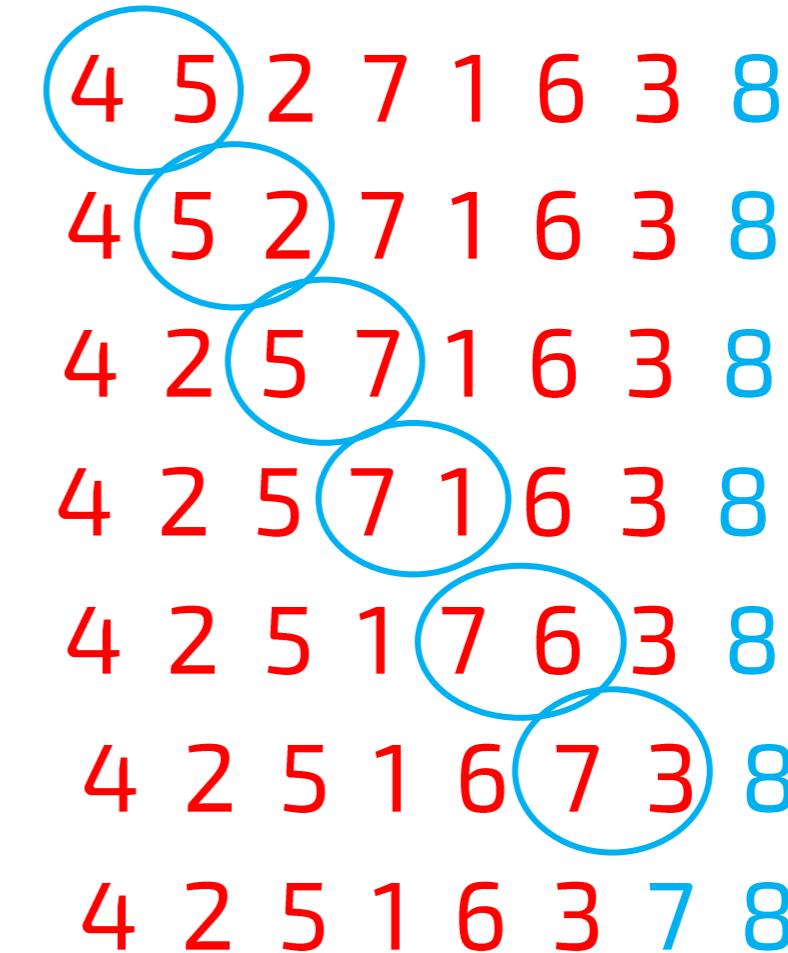
$\Rightarrow 5 > 2 ?$ – sim, troca

$\Rightarrow 5 > 7 ?$ – não, não troca

$\Rightarrow 7 > 1 ?$ – sim, troca

$\Rightarrow 7 > 6 ?$ – sim, troca

$\Rightarrow 7 > 3 ?$ – sim, troca



7 fica no lugar correto ... E assim sucessivamente...

Bubble Sort - Algoritmo

```
bubbleSort (int[] v)
início
    inteiro i, j;
    para i de 0 até v.length-2 (inclusive) faça
        para j de 1 até v.length-1-i (inclusive) faça
            se v[j-1] > v[j]
                então troca (v[j], v[j-1]);
fim
```

Insertion Sort

- Método simples de inserção
- Características
 - Considera 2 segmentos do vetor: ordenado (aumenta a cada varredura) e não ordenado (diminui)
 - Ordena através da inserção de um elemento por vez do segmento não ordenado no segmento ordenado, na sua posição correta
 - Inicialmente, o segmento ordenado contém apenas o primeiro elemento do vetor

Insertion Sort - Exemplo

Sejam os dados (ordenados / não ordenados):

- ⇒ Considera o 1º ordenado
- ⇒ Insere o 7 na parte ordenada
- ⇒ Insere o 5 na parte ordenada
- ⇒ Insere o 2 na parte ordenada
- ⇒ Insere o 8 na parte ordenada
- ⇒ Insere o 1 na parte ordenada
- ⇒ Insere o 6 na parte ordenada
- ⇒ Insere o 3 na parte ordenada

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

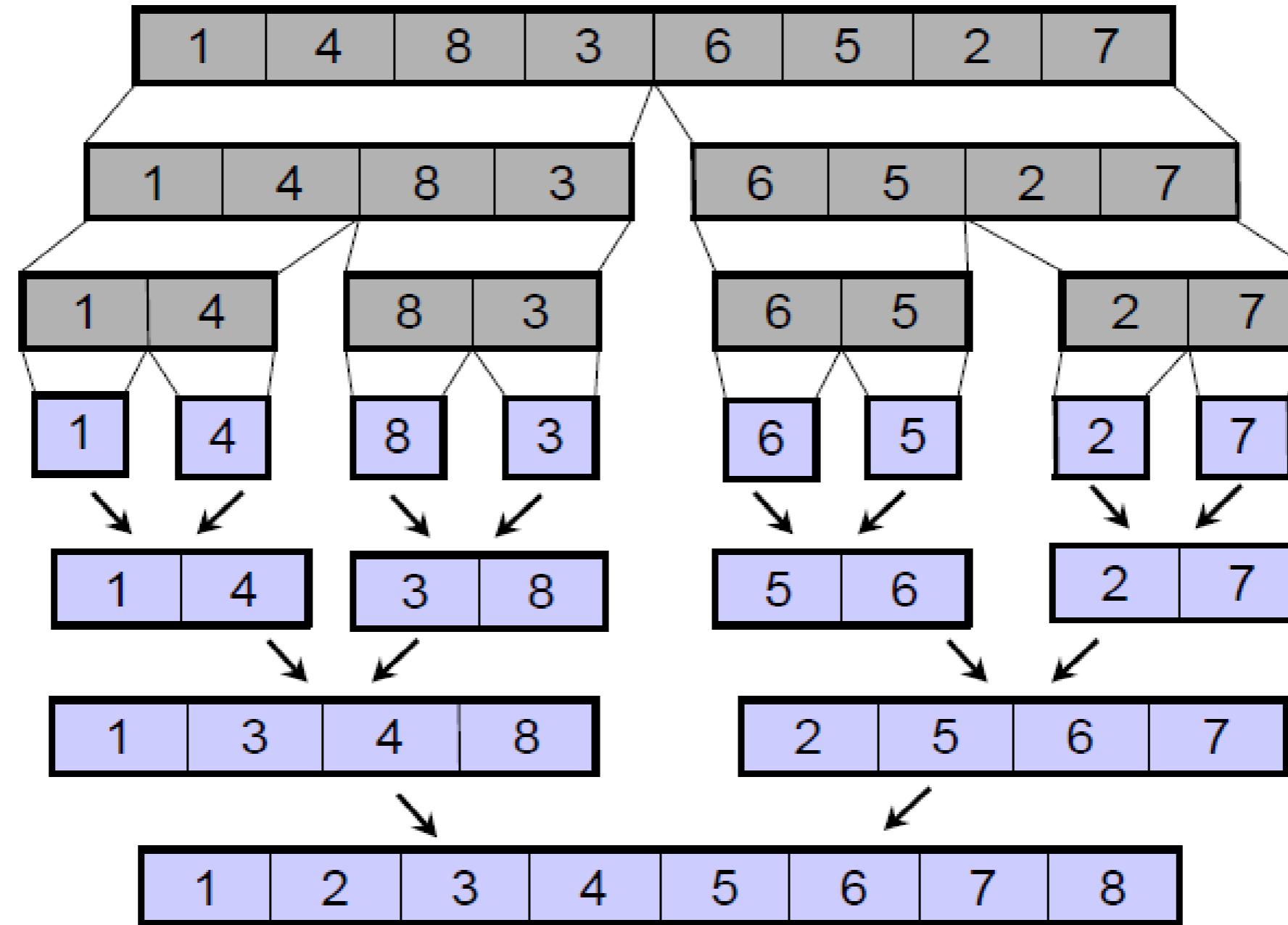
Insertion Sort - Algoritmo

```
insertionSort (int[] v)
início
    inteiro i, j, x;
    para i de 1 até v.length-1 (inclusive) faça
        início
            x ← v[i];
            j ← i - 1;
            enquanto (j >= 0) e (v[j] > x) faça
                início
                    v[j+1] ← v[j];
                    j ← j - 1;
                fim
                v[j+1] ← x;
            fim
        fim
    fim
```

Merge Sort

- Método particular de ordenação
 - Baseia-se em intercalações sucessivas de 2 sequências ordenadas em uma única sequência ordenada
- Aplica o método “dividir para conquistar”
 - Divide o vetor de n elementos em 2 segmentos de comprimento $n/2$
 - Ordena recursivamente cada segmento
 - Intercala os 2 segmentos ordenados para obter o vetor ordenado completo

Merge Sort - Exemplo



Merge Sort – Algoritmo recursivo

```
mergeSort (int p, int r, int[] v)
/* p = índice inicial do vetor a ser ordenado */
/* r = índice final + 1 do vetor a ser ordenado */
/* vetor v [p ..... r-1] */
início
  se (p < r-1)
    início
      int q ← (p + r) / 2;      /* q = índice do meio */
      mergeSort(p, q, v);       /* ordena 1a metade */
      mergeSort(q, r, v);       /* ordena 2a metade */
      intercala(p, q, r, v);   /* intercala 2 metades */
    fim
  fim
```

Merge Sort – Algoritmo Intercala

```
intercala (int p, int q, int r, int[] v)
/* 1a metade do vetor v[ p ... q-1 ] */
/* 2a metade do vetor v[ q ... r-1 ] */
início
    inteiro i, j, k, w[];
    /* deve alocar vetor w de r-p elementos */

    i ← p; j ← q; k ← 0;
    enquanto (i < q) e (j < r) faça
        se (v[i] ≤ v[j])
            então w[k++] ← v[i++];
        senão w[k++] ← v[j++];
    enquanto (i < q) faça w[k++] ← v[i++];
    enquanto (j < r) faça w[k++] ← v[j++];
    para i de p até r-1 faça v[i] ← w[i - p];
fim
```

QuickSort

- Proposto por Hoare em 1960 e publicado em 1962.
- Em geral, o algoritmo é muito mais rápido que os algoritmos elementares.
- Também é um algoritmo de troca: ordena através de sucessivas trocas entre pares de elementos do vetor.
- Aplica o método “dividir para conquistar”:
 - A idéia básica é dividir o problema de ordenar um conjunto com n itens em dois problemas menores.
 - Os problemas menores são ordenados independentemente.
 - Os resultados são combinados para produzir a solução final.

QuickSort

- A parte mais delicada do método é o processo de partição.
- O vetor A [Esq..Dir] é rearranjado por meio da escolha arbitrária de um **pivô** x .
- O vetor A é particionado em duas partes:
 - Parte esquerda: elementos $\leq x$.
 - Parte direita: elementos $\geq x$.



QuickSort – Escolha do pivô

- Particionamento pode ser feito de diferentes formas.
- Principal decisão é escolher o pivô
 - Primeiro elemento do vetor
 - Último elemento do vetor
 - Elemento do meio do vetor
 - Elemento que mais ocorre no vetor
 - Elemento mais próximo da média aritmética dos elementos do vetor
- Logicamente, o algoritmo é diferente, dependendo da forma de escolher o pivô

QuickSort – Explo (pivô = elemento do meio)

- Seja o vetor abaixo.
- Considerando que 4 seja o pivô, o primeiro passo do Quicksort rearranjaria o vetor da forma abaixo:

| | | | | | |
|-------------|---|---|---|---|---|
| 6 | 5 | 4 | 1 | 3 | 2 |
| <i>pivô</i> | | | | | |

| | | | | | |
|-------------|---|---|---|---|---|
| 2 | 3 | 1 | 4 | 5 | 6 |
| <i>pivô</i> | | | | | |

QuickSort – Explô (pivô = elemento do meio)

1º Passo

| | | | | | |
|----------|---|---|---|---|----------|
| 6 | 5 | 4 | 1 | 3 | 2 |
| <i>i</i> | | | | | <i>j</i> |

$6 > 4$ e $2 < 4$, então, troca 6 com 2

Incrementa i e decrementa j

Pivô
4

| | | | | | |
|----------|---|---|---|---|----------|
| 2 | 5 | 4 | 1 | 3 | 6 |
| <i>i</i> | | | | | <i>j</i> |

$5 > 4$ e $3 < 4$, então, troca 5 com 3

Incrementa i e decrementa j

| | | | | | |
|---|----------|---|----------|---|---|
| 2 | 3 | 4 | 1 | 5 | 6 |
| | <i>i</i> | | <i>j</i> | | |

$4 = 4$ e $1 < 4$, então, troca 4 com 1

Incrementa i e decrementa j

| | | | | | |
|---|---|----------|----------|---|---|
| 2 | 3 | 1 | 4 | 5 | 6 |
| | | <i>j</i> | <i>i</i> | | |

Partição 1

Partição 2

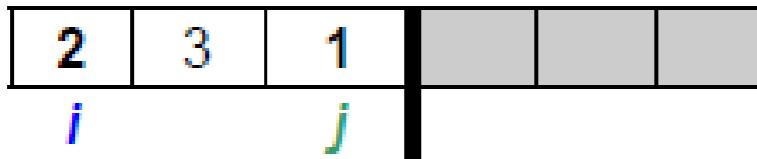
$i > j \rightarrow PARTICIONA$

i é diferente de final e j é diferente de início.

Faz a chamada recursiva para os dois casos.

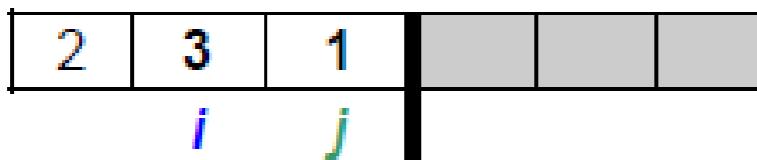
QuickSort – Explor (continuação)

2º Passo



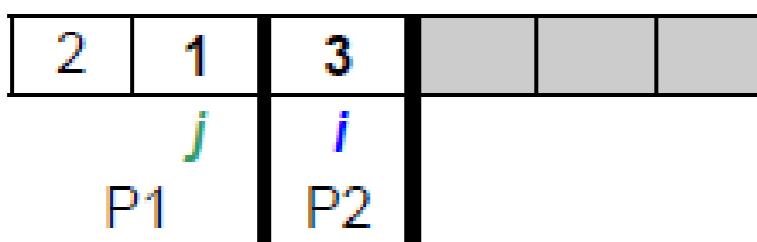
$2 < 3 \rightarrow$ incrementa i

Pivô
3



$3 = 3$ e $1 < 3$, então, troca 3 com 1

Incrementa i e decrementa j

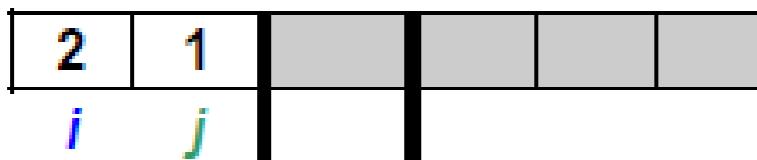


$i > j \rightarrow$ PARTICIONA

Chama a recursão apenas para a Partição 1

(P1), pois P2 é de tamanho igual a 1 ($i = \text{fim}$)

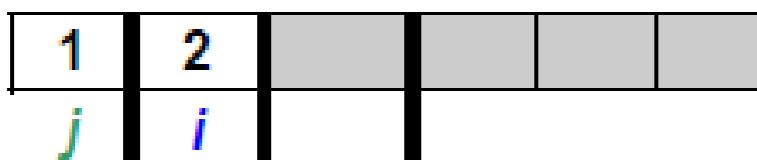
3º Passo



$2 = 2$ e $1 < 2$, então, troca 2 com 1

Pivô
2

Incrementa i e decrementa j



$i > j \rightarrow j = \text{inicio}$ e $i = \text{fim} \rightarrow$ Finaliza

Não particiona mais

QuickSort – Algoritmo (pivô = elem do meio)

```
particiona (int[] v, int indInicio, int indFim)
início
    inteiro i, j, pivo;
    i ← indInicio; j ← indFim;
    pivo ← v[(indInicio+indFim)/2];
    enquanto i <= j faça
        início
            enquanto i < indFim e v[i] < pivo
                i ← i + 1;
            enquanto j > indInicio e v[j] > pivo
                j ← j - 1;
            se i <= j
                entao inicio
                    troca (v[i], v[j]);
                    i ← i + 1;
                    j ← j - 1;
                fim
            fim
            se indInicio < j então particiona (v, indInicio, j);
            se i < indFim então particiona (v, i, indFim);
        fim
```

QuickSort – Algoritmo (pivô = elem do meio)

```
quickSort  
    particiona ( v, 0, v.length-1 );
```

Quick Sort - Exemplo (pivô= elem. do meio)

Sejam os dados (simulação do algoritmo anterior):

$\Rightarrow i = \text{indInic} = \text{zero}, j = \text{indFim} = 7, \text{ pivo} = v[(\text{indInic} + \text{indFim})/2] = 2$

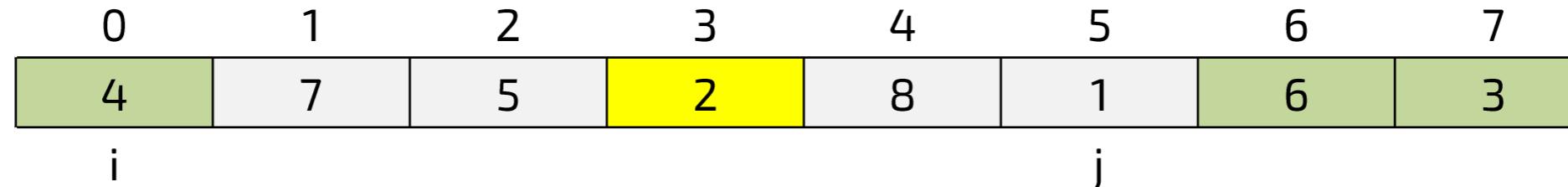
$\Rightarrow v[i] < pivo$? Não (elemento maior que o pivô – verde)

$\Rightarrow v[j] > pivo$? Sim, então decremente j (verde)

A horizontal array of 8 elements. The elements are: 4, 7, 5, 2, 8, 1, 6, 3. The segment containing 2 is highlighted in yellow. Index i is at the start of the segment containing 4. Index j is at the end of the segment containing 6.

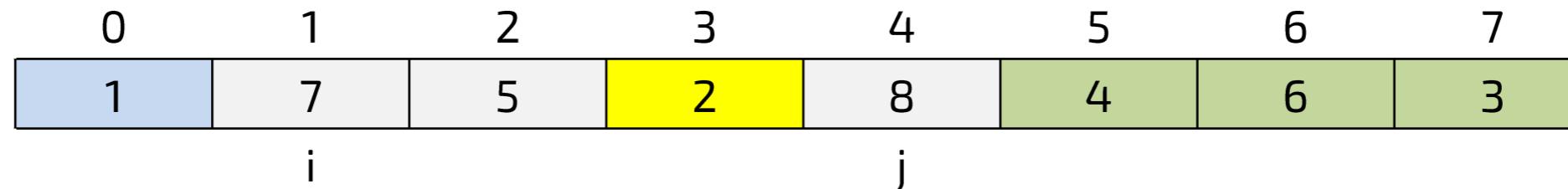
Quick Sort - Exemplo (pivô= elem. do meio)

$\Rightarrow v[j] > \text{pivô}$? Sim, então decrementa j (verde)



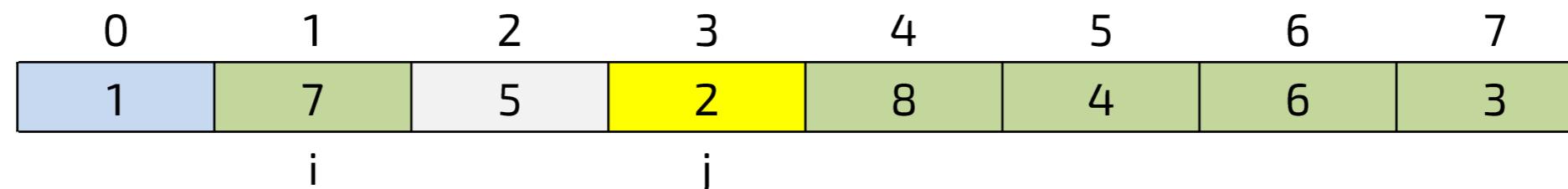
$\Rightarrow v[j] > \text{pivô? Não (azul)}$

$\Rightarrow i \leq j$? Sim, então troca $v[i]$ com $v[j]$, incrementa i , decrementa j



$\Rightarrow v[i] < \text{pivô} ? \text{Não (verde)}$

$\Rightarrow v[j] > \text{pivô}$? Sim, então decrementa



Quick Sort - Exemplo (pivô= elem. do meio)

A horizontal array of 8 cells, indexed from 0 to 7 above each cell. The cells contain the following values: 1, 7, 5, 2, 8, 4, 6, 3. The cell at index 3 is highlighted in yellow. The cell at index 1 is shaded blue and contains the label 'i' below it. The cell at index 3 is shaded yellow and contains the label 'j' below it.

$\Rightarrow v[j] > \text{pivô} ? \text{N}\ddot{\text{a}}\text{o}$

$\Rightarrow i \leq j$? Sim, então troca $v[i]$ com $v[j]$, incrementa i , decrementa j

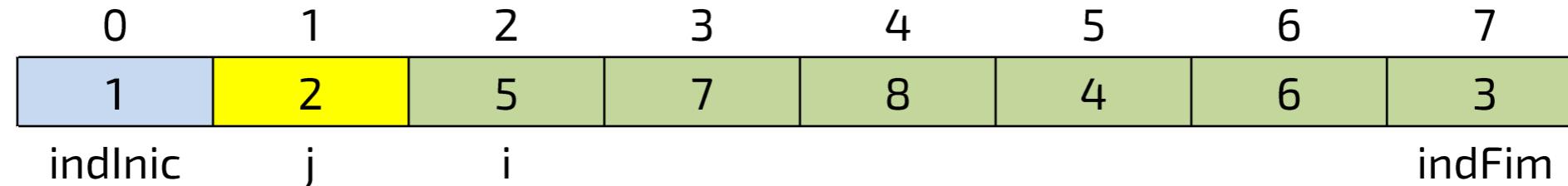
A horizontal array of 8 cells, indexed from 0 to 7 above each cell. The values in the cells are: 1, 2, 5, 7, 8, 4, 6, 3. The cell at index 1 contains the value 2, which is highlighted with a yellow background. The cell at index 5 contains the value 4, which is highlighted with a green background.

$\Rightarrow v[i] < \text{pivô} ? \text{N}\ddot{\text{a}}\text{o}$

$\Rightarrow v[j] > \text{pivô}$? Sim, então decrementa

A diagram illustrating a 1D array of size 8. The indices are labeled from 0 to 7 above the array. The array elements are: 1, 2, 5, 7, 8, 4, 6, 3. A pointer 'j' is positioned below index 1, pointing to the value 2. A pointer 'i' is positioned below index 3, pointing to the value 7.

Quick Sort – Exemplo (pivô= elem. do meio)

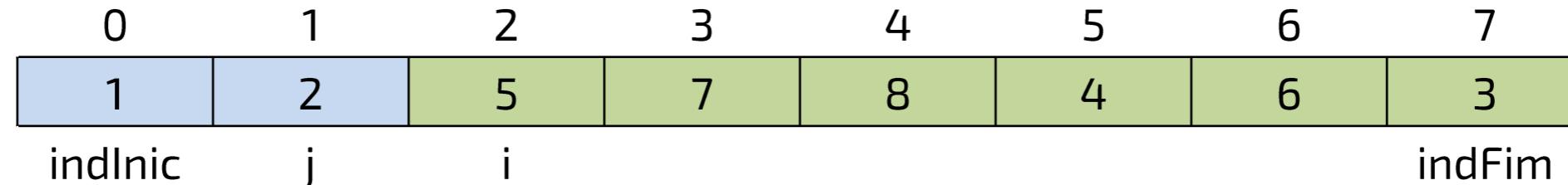


⇒ $v[j] > \text{pivô}$? Não

⇒ $i \leq j$? Não

⇒ Como i não é mais $\leq j$, acaba essa iteração: o pivô está posicionado no lugar correto, os números à direita do pivô são maiores do que ele, e número à esquerda do pivô é menor do que ele.

⇒ Aplica recursivamente o algoritmo na partição [indInic até j] e na partição [j até indFim]



QuickSort – Algoritmo (pivô = último elemento)

```
particiona (int[] v, int indInicio, int indFim)
início
    inteiro i, j, pivo;
    pivo ← v[indFim];
    i ← indFim;
    para j de indFim - 1 até indInicio faça
        início
            se v[j] > pivo
            então inicio
                i ← i - 1;
                troca (v[i], v[j]);
            fim
        fim
    troca (v[indFim], v[i]);
    se indInicio < i então particiona (v, indInicio, i-1);
    se i < indFim então particiona (v, i+1, indFim);
fim
```

Quick Sort - Exemplo (pivô= último elem.)

Sejam os dados (simulação do algoritmo anterior):

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---------|---|---|---|---|---|---|--------|
| indInic | | | | | | | indFim |
| 3 | 7 | 5 | 6 | 8 | 2 | 1 | 4 |

$\Rightarrow \text{pivo} = v[\text{indFim}] = 4, \quad i = \text{indFim} = 7, \quad j = \text{indFim}-1 = 6$

The diagram shows an array of 8 elements represented by white boxes. Above the array, indices 0 through 7 are listed. The element at index 7, which contains the value 4, is highlighted with a yellow background.

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

j i

$$\Rightarrow j = \text{indFim} - 1 = 6$$

$\Rightarrow v[j] > \text{pivo}$? Não, então decrementa j (azul = menor que pivô)

A horizontal array of 8 cells, indexed from 0 to 7 above each cell. The cells contain the values 3, 7, 5, 6, 8, 2, 1, and 4 respectively. Below the array, the index j points to the cell containing 2, and the index i points to the cell containing 4.

Quick Sort - Exemplo (pivô= último elem.)

$\Rightarrow v[j] > \text{pivô}$? Não, então decrementa j

A horizontal array of 8 cells, each containing a number from 0 to 7. The numbers are: 3, 7, 5, 6, 8, 2, 1, 4. The cell at index 5 contains the value 2, which is highlighted with a light blue background. The cell at index 7 contains the value 4, which is highlighted with a yellow background. Below the array, the index 5 is labeled 'j' and the index 7 is labeled 'i'.

$\Rightarrow v[j] > \text{pivô}$? Sim, então decrementa i, troca $v[i]$ com $v[j]$, decrementa j

A horizontal array of 8 elements: 3, 7, 5, 6, 8, 2, 1, 4. The element at index 4 is shaded green, and the element at index 5 is shaded blue. An arrow labeled *j* points to the green element, and another arrow labeled *i* points to the blue element.

A horizontal array of 8 boxes labeled 0 through 7. The boxes are colored: 0 (light gray), 1 (light gray), 2 (light gray), 3 (light gray), 4 (blue), 5 (blue), 6 (green), and 7 (yellow). Below the array, 'j' is under box 4 and 'i' is under box 6.

Quick Sort - Exemplo (pivô= último elem.)

A horizontal array of 8 elements with indices 0 through 7 above them. The elements are: 3, 7, 5, 6, 1, 2, 8, 4. The element at index 4 (value 1) is highlighted in blue, and the element at index 6 (value 8) is highlighted in green.

$\Rightarrow v[j] > \text{pivô}$? Sim, então decrementa i, troca $v[i]$ com $v[j]$, decrementa j

A horizontal array of 8 cells, indexed from 0 to 7. The elements are: 3, 7, 5, 6, 1, 2, 8, 4. The cell at index 3 contains the value 6, which is highlighted with a green background. The cell at index 5 contains the value 2, which is highlighted with a light blue background. The indices are labeled above the array: 0, 1, 2, 3, 4, 5, 6, 7.

A horizontal array of 8 cells, indexed from 0 to 7 above each cell. The cells contain the values 3, 7, 5, 2, 1, 6, 8, and 4 respectively. The cell at index 3 contains the value 2, and the cell at index 5 contains the value 6. The indices are labeled below the array.

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

j i

The diagram shows an array of 8 elements with indices 0 to 7 above them. The elements are: 3, 7, 5, 2, 1, 6, 8, 4. Indices 3 and 5 are highlighted in blue, while the others are grey. A pointer labeled 'j' points to index 3, and a pointer labeled 'i' points to index 5.

Quick Sort - Exemplo (pivô= último elem.)

A horizontal array of 8 elements with indices 0 through 7 above them. The elements are: 3, 7, 5, 6, 1, 2, 8, 4. The element at index 4 (value 1) is highlighted with a light blue background. The element at index 6 (value 8) is highlighted with a light green background.

$\Rightarrow v[j] > \text{pivô}$? Sim, então decrementa i, troca $v[i]$ com $v[j]$, decrementa j

A horizontal array of 8 cells, indexed from 0 to 7 above each cell. The values are: 3, 7, 5, 6, 1, 2, 8, 4. The cell containing 6 is highlighted in green and labeled *j* below it. The cell containing 1 is highlighted in light blue and labeled *i* below it.

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

j *i*

A horizontal array of 8 cells, indexed from 0 to 7 above each cell. The cells contain the values 3, 7, 5, 2, 1, 6, 8, and 4 respectively. Below the array, the index j is centered under the cell containing 2, and the index i is centered under the cell containing 6.

A horizontal array of 8 elements, indexed from 0 to 7. The elements are: 3, 7, 5, 2, 1, 6, 8, 4. Index j is at position 2 (value 5). Index i is at position 5 (value 6).

Quick Sort - Exemplo (pivô= último elem.)

$\Rightarrow v[j] > \text{pivô}$? Sim, então decrementa i, troca $v[i]$ com $v[j]$, decrementa j

A horizontal array of 8 cells, indexed from 0 to 7. The values are: 3, 7, 5, 2, 1, 6, 8, 4. Cell 5 (index 2) is highlighted in green. Cell 1 (index 4) is highlighted in light blue. Below the array, 'j' is centered under index 2, and 'i' is centered under index 4.

A horizontal array of 8 cells, indexed from 0 to 7 above each cell. The cells contain the following values: 3, 7, 1, 2, 5, 6, 8, and 4. The cell at index 2 contains 1, and the cell at index 4 contains 5. The indices are labeled below the array as j and i.

A horizontal array of 8 cells, indexed from 0 to 7 above each cell. The cells contain the following values:
Cell 0: 3
Cell 1: 7 (shaded gray)
Cell 2: 1 (shaded blue)
Cell 3: 2 (shaded blue)
Cell 4: 5 (shaded green)
Cell 5: 6 (shaded green)
Cell 6: 8 (shaded green)
Cell 7: 4 (shaded yellow)
Below the array, 'j' is positioned under the 7th cell, and 'i' is positioned under the 4th cell.

Quick Sort - Exemplo (pivô= último elem.)

A horizontal array of 8 cells, each containing a value. The indices (0 to 7) are shown above the array. The values are: 3, 7, 1, 2, 5, 6, 8, and 4. Index j is at index 1, and index i is at index 5.

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

j

i

$\Rightarrow v[j] > \text{pivô}$? Sim, então decrementa i, troca $v[i]$ com $v[j]$, decrementa j

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 3 | 7 | 1 | 2 | 5 | 6 | 8 | 4 |
| j | | i | | | | | |

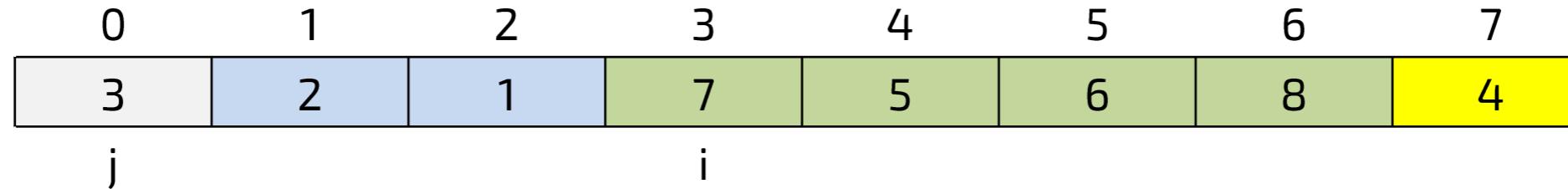
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 3 | 2 | 1 | 7 | 5 | 6 | 8 | 4 |
| j | | i | | | | | |

Diagram illustrating the state of an array A with 8 elements. The indices j , i , and k are marked as follows:

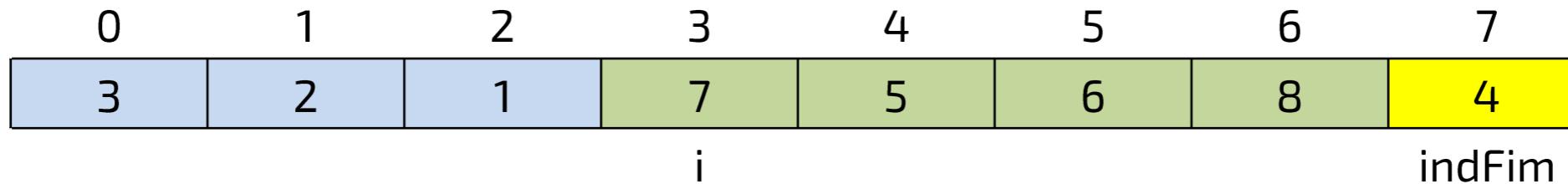
- $j = 0$: Points to the element $A[0] = 3$.
- $i = 3$: Points to the element $A[3] = 7$.
- $k = 6$: Points to the element $A[6] = 8$.

The array elements are: $A[0] = 3$, $A[1] = 2$, $A[2] = 1$, $A[3] = 7$, $A[4] = 5$, $A[5] = 6$, $A[6] = 8$, $A[7] = 4$.

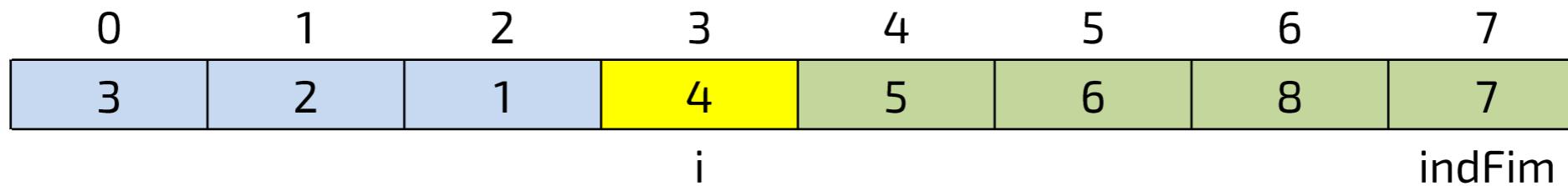
Quick Sort – Exemplo (pivô= último elem.)



$\Rightarrow v[j] > \text{pivô}$? Não, então decrementa j \Rightarrow acabou o vetor



\Rightarrow Troca $v[i]$ com $v[\text{indFim}]$



\Rightarrow Acabou a 1^a iteração: pivô está posicionado no lugar

\Rightarrow Repete recursivamente o algoritmo na partição da esq e na da direita

Pesquisa Sequencial vs Pesquisa Binária

- Quando é preciso verificar se um valor está em um vetor, normalmente executamos uma Pesquisa Sequencial.
- Na **Pesquisa Sequencial**, percorre-se o vetor, desde o início, seguindo na sequência do vetor, do índice zero até o último elemento do vetor, até encontrar o valor procurado. Se o valor não for encontrado, percorrerá o vetor inteiro.
- A Pesquisa Sequencial funciona tanto se os dados do vetor estiverem desordenados como se estiverem ordenados.
- Existe outro método de busca ou pesquisa, chamada de **Busca ou Pesquisa Binária**
- Esse método funciona apenas quando os dados estão ordenados.

Pesquisa Binária

- Na Pesquisa Binária, ao invés de começarmos a procurar no início do vetor, verificamos se o elemento que está sendo procurado está no meio do vetor.
- Se estiver, então já encontramos.
- Se não estiver: verificamos: Valor procurado > elemento do meio ?
 - Sim, então procuramos na metade à direita do elemento do meio
 - Não, então procuramos na metade à esquerda do elemento do meio
- A pesquisa continua até encontrar o elemento procurado, ou até o algoritmo detectar que o elemento não está no vetor.

Pesquisa Binária - Algoritmo

```
inteiro pesqBin (inteiro[] vetor, inteiro x)    // x é o valor procurado
início
    inteiro indinf, indsуп, meio;
    indinf = 0;                      /* índice inferior */
    indsуп = vetor.length - 1; /* índice superior */
    enquanto (indinf <= indsуп)
        início
            meio = (indinf + indsуп) / 2; /* calcula o índice do meio */
            se (vetor[meio] == x)
                retorna meio;          // retorna o índice do elemento encontrado
            senão se (x < vetor[meio]) // x < elemento do meio ?
                indsуп = meio - 1; // sim, então indsуп passa a ser meio-1
            senão
                indinf = meio + 1; // não, então indinf passa a ser meio+1
        fim
    retorna -1; /* quando indinf > indsуп, o elemento não está no vetor */
fim
```

Pesquisa Binária – Exemplo (simulação)

Sejam os dados (simulação do algoritmo anterior):

A horizontal scale from 0 to 7 with tick marks at 10, 20, 30, 40, 50, 60, 70, and 80. The label "indinf" is at the left end and "indsup" is at the right end.

⇒ vamos supor que o elemento procurado $x = 50$

$$\Rightarrow \text{meio} = (\text{indinf} + \text{indsup})/2 = \exists$$

A horizontal scale from 0 to 7 with tick marks at 10, 20, 30, 40, 50, 60, 70, and 80. The labels "indinf", "meio", and "indsup" are positioned below the scale.

$\Rightarrow x = \text{vetor}[\text{meio}]?$ Não

$\Rightarrow x < \text{vetor}[\text{meio}]?$ Não

\Rightarrow Então $x > \text{vetor}[\text{meio}] \Rightarrow \text{indinf} = \text{meio} + 1 = 4$

Pesquisa Binária – Exemplo (simulação)

| | | | | | | | |
|--------|----|----|----|----|----|--------|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 |
| indinf | | | | | | indsup | |

⇒ A busca agora terá foco na parte do vetor, dos índices 4 a 7

$$\Rightarrow \text{meio} = (\text{indinf} + \text{indsup})/2 = 5$$

| | | | | | | | |
|--------|----|----|----|----|----|--------|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 |
| indinf | | | | | | indsup | |

⇒ $x = \text{vetor}[\text{meio}]$? Não

⇒ $x < \text{vetor}[\text{meio}]$? Sim ⇒ $\text{indsup} = \text{meio}-1 = 4$

| | | | | | | | |
|--------|----|----|----|----|----|--------|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 |
| indinf | | | | | | indsup | |

Pesquisa Binária – Exemplo (simulação)

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 |

indinf
indsup

- ⇒ A busca agora terá foco na parte do vetor, dos índices 4 a 4
- ⇒ meio = (indinf + indsуп) / 2 = 4
- ⇒ x = vetor[meio]? Sim ⇒ Encontramos!!!

Dessa forma, em 3 iterações, encontramos o elemento procurado.

Para um vetor de 8 posições, em 3 iterações podemos encontrar o valor ou concluir que o valor não está no vetor.

Se fosse a busca sequencial, precisaríamos de 8 iterações para concluir que o valor não está no vetor.

Notação O-grande ou Big-O

- Em computação, utiliza-se a notação O-grande ou Big-O para representar a complexidade de tempo de execução de um algoritmo
- Leva-se em conta a quantidade de operações que o algoritmo executa em função da quantidade de dados que ele manipula
- A pesquisa sequencial tem complexidade $O(n)$, sendo n o tamanho do vetor.
- Num vetor de 8 posições, são necessários 8 iterações para concluir que o elemento não está no vetor
- A pesquisa binária tem complexidade $O(\log_2 n)$
- Quando $n=8$, $\log_2 n = 3$. Vimos que em 3 iterações, encontramos o elemento ou concluímos que ele não está no vetor.

Complexidade dos algoritmos de ordenação

- Selection Sort, Bubble Sort e Insertion Sort tem **complexidade $O(n^2)$** , pois seus algoritmos tem um for dentro de outro for.
- Nesses algoritmos, para posicionar um elemento na posição correta, é preciso percorrer todo o vetor. Assim, como para cada elemento do vetor, precisa fazer aproximadamente n iterações, a quantidade total de iterações é aproximadamente $n * n = n^2$.
- Para um vetor com $n = 8$, isso dá aproximadamente 64 iterações.
- O Merge Sort tem **complexidade $O(n \log_2 n)$** , porque ele tem um comportamento semelhante ao da Pesquisa Binária, ao ir dividindo o vetor ao meio, e depois ir juntando fazendo as intercalações.
- Quando $n=8$, $n \log_2 n = 8 * 3 = 24$.
- O Quick Sort também tem, no geral, a mesma complexidade do Merge.

Complexidade dos algoritmos de ordenação

- Por isso, para quantidades muito grande de dados a serem ordenados, é recomendável utilizar o MergeSort ou o QuickSort, ao invés dos 3 primeiros algoritmos vistos neste material.
- Assista o vídeo que há no início do material para ver quais algoritmos são mais eficientes e quais são menos eficientes.

Outras animações interessantes

- MergeSort vs QuickSort:

<https://www.youtube.com/watch?v=es2T6KY45cA>

- InsertionSort vs BubbleSort:

<https://www.youtube.com/watch?v=TZRWRjq2CAg>

- Selection Sort:

- Dança: <https://www.youtube.com/watch?v=Ns4TPTC8whw>

- Bubble Sort:

- Lego: https://www.youtube.com/watch?v=MtcrEhrt_K0

- Dança: <https://www.youtube.com/watch?v=lyZQPjUT5B4>

- Explicação sobre HeapSort

<https://www.youtube.com/watch?v=H5kAcmGOn4Q>

Exercícios

1. Sejam os valores: 5, 3, 4, 2, 8, 1, 6, 7

Faça a simulação (a mão) dos 5 algoritmos dessa aula para a ordenação dos valores acima, exibindo os dados após cada iteração do algoritmo.

2. Implementar os 5 algoritmos de ordenação dessa aula em Java. Acrescentar nos algoritmos a exibição dos elementos do vetor, após cada iteração da ordenação, e observe como é o mecanismo de cada algoritmo.
3. Implementar o algoritmo de pesquisa binária e testar com valores que existem no vetor e com valores que não existem. Implementar também a versão recursiva do algoritmo de pesquisa binária e testar.

Obrigada!

BandTec
DIGITAL SCHOOL

Em caso de dúvidas, entre em contato com:
celia.taniwaki@bandtec.com.br