Algoritmos e Estruturas de Dados

Ordenação (parte I) – Capítulo 12
2019/20

Ordenação

Dada uma sequência de registos

$R_1, R_2, \ldots, R_n$, com as respetivas chaves $k_1, k_2, \ldots, k_n$,

pretende-se uma permutação dos registos

$R_{i1}, R_{i2}, \ldots, R_{in}$ tal que $k_{i1} \leq k_{i2} \leq \ldots \leq k_{in}$.

Um algoritmo de ordenação diz-se **estável** se preserva a ordem original dos registos com a mesma chave.

Exemplo

(3, “Joana”), (7, “Antonio”), (3, “Francisco”), (5, “Teresa”), (5, “Ana”)

Permutação estável

(3, “Joana”), (3, “Francisco”), (5, “Teresa”), (5, “Ana”), (7, “Antonio”)

TAD ***Comparator***(1)

```
package dataStructures;

public interface Comparator<E>{

    // Compares its two arguments for order.
    // Returns a negative integer, zero, or a positive integer
    // as the first argument is less than, equal to, or greater
    // than the second.
    int compare( E element1, E element2 );

}
```

Permite ter vários critérios de ordenação.

As variáveis na calculadora (problema das práticas) podem ser ordenadas pelo seu identificador ou pelo seu valor.

Como fazer?

TAD ***Comparator*** (2)
Exemplo

Ter uma classe por cada critério de comparação. Neste caso:

- Ter duas classes que implementam o **Comparator<Variavel>**

Por identificador

```
package calculadoraMemoria;

import dataStructures.Comparator;

public class IdComparator implements Comparator<Variavel>{

    @Override
    public int compare(Variavel v1, Variavel v2) {
        return v1.getName().compareTo(v2.getName());
    }

}
```

TAD ***Comparator*** (3)
Exemplo

Por valor e em caso de empate por identificador

```
package calculadoraMemoria;

import dataStructures.Comparator;

public class ValueComparator implements Comparator<Variavel>{

    @Override
    public int compare(Variavel v1, Variavel v2) {
        if (v1.getValue() < v2.getValue())
            return -1;
        if (v1.getValue() > v2.getValue())
            return 1;
        Comparator<Variavel> c= new IdComparator();
        return c.compare(v1,v2);
    }

}
```

Algoritmos de ordenação


```
public class Array<E> implements List<E>{

...

public static <E> void xSort( E[] vec, int vecSize, Comparator<E> c){
        ..... c.compare(vec[i],vec[j])...
}

public static <E> void ySort( E[] vec, int vecSize, Comparator<E> c){
        ..... c.compare(vec[i],vec[j])...
}

…
}
```

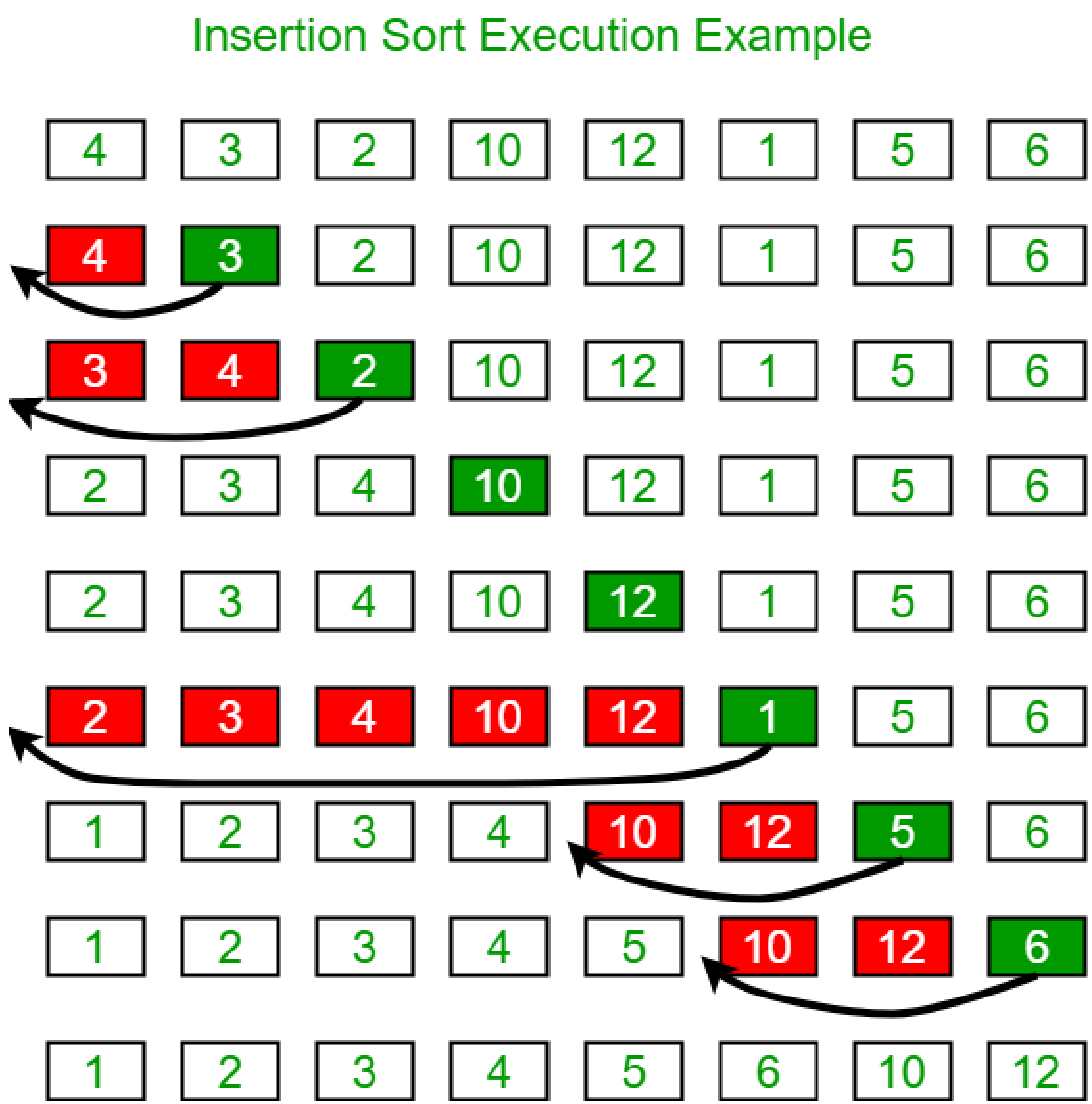
Algoritmos de ordenação
exemplo

```
private Variavel[] memoria;
private int variavelContador;

public void ordenarPorIdentificador( ) {
 Array.xSort(memoria, variavelContador, new IdComparator());
}

public void ordenarPorValor( ){
 Array.xSort(memoria, variavelContador, new ValueComparator());
}
```

Insertion Sort (n elementos)



Em cada **iteração i**:
- vector ordenado de 0 a i-1;
- vector desordenado de i a n-1

https://visualgo.net/

Insertion Sort (n elementos)

Algoritmo estável

```java
public static <E> void insertionSort( E[] vec, int vecSize,
                                      Comparator<E> c ){

    for ( int pos = 1; pos < vecSize; pos++ ) {
        E element = vec[pos];
        int hole = pos;
        while ( hole > 0 && c.compare(element, vec[hole - 1]) < 0 ) {
                vec[hole] = vec[hole - 1];
                hole--;
        }
        vec[hole] = element;
    }
}
```

Insertion Sort (n elementos)

Complexidade Temporal (1)

1	2	3	4	5	6	10	12

No melhor caso

(quando a sequência está ordenada pela ordem pretendida)

$$\sum_{i=1}^{n-1} 1 = n - 1 = O(n).$$

```
public static <E> void insertionSort( E[] vec, int vecSize,
                                              Comparator<E> c ){

    for ( int pos = 1; pos < vecSize; pos++ ) {
        E element = vec[pos];
        int hole = pos;
        while ( hole > 0 && c.compare(element, vec[hole - 1]) < 0 ) {
                vec[hole] = vec[hole - 1];
                hole--;
        }
        vec[hole] = element;
    }
}
```

Insertion Sort (n elementos)
Complexidade Temporal (2)

12	10	6	5	4	3	2	1

No pior caso

(quando a sequência está ordenada por ordem inversa da pretendida)

$$\sum_{i=1}^{n-1} i = \frac{(n-1)\,n}{2} = O(n^2).$$

```
public static <E> void insertionSort( E[] vec, int vecSize,
                                       Comparator<E> c ){

    for ( int pos = 1; pos < vecSize; pos++ ) {
        E element = vec[pos];
        int hole = pos;
        while ( hole > 0 && c.compare(element, vec[hole - 1]) < 0 ) {
                vec[hole] = vec[hole - 1];
                hole--;
        }
        vec[hole] = element;
    }
}
```

Insertion Sort (n elementos)
Complexidade Temporal (3)

No caso esperado $O(n^2)$.

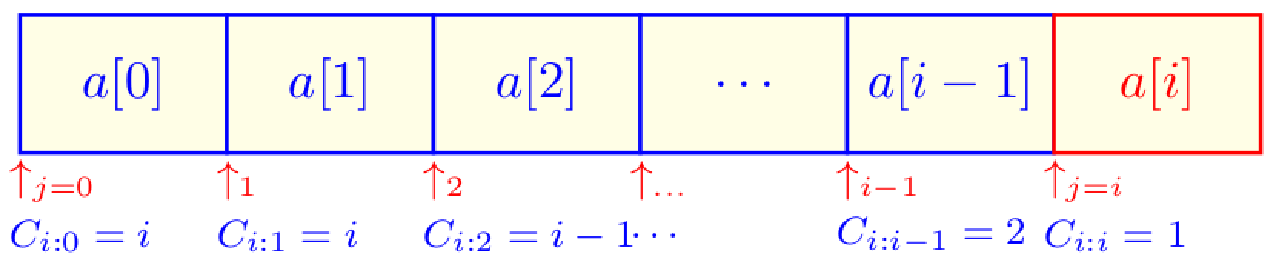
Prova:

- Assumindo que todas as entradas são equiprováveis.

- Seja $\overline{C}_i$ o número medio (esperado) de comparacões em cada iteração i.

- Logo o número total médio de comparações é $\overline{C} = \sum_{i=1}^{n-1} \overline{C}_i$.

- Na iteração i, temos da posição 0 à i-1 já ordenadas, e vamos inserir o elemento na posição i.



$$\overline{C}_i = \frac{1}{i+1} \sum_{j=0}^{i} C_{i:j},$$

Insertion Sort (n elementos)
Complexidade Temporal (4)

No caso esperado $O(n^2)$.

$$\overline{C} = \sum_{i=1}^{n-1} \overline{C}_i .$$

Prova (continuação):

$a[0]$	$a[1]$	$a[2]$	$\cdots$	$a[i-1]$	$a[i]$
$\uparrow_{j=0}$	$\uparrow_1$	$\uparrow_2$	$\uparrow_{\ldots}$	$\uparrow_{i-1}$	$\uparrow_{j=i}$
$C_{i:0} = i$	$C_{i:1} = i$	$C_{i:2} = i-1 \cdots$		$C_{i:i-1} = 2$	$C_{i:i} = 1$

$$\overline{C}_i = \frac{1}{i+1} \sum_{j=0}^{i} C_{i:j} ,$$

$$\overline{C}_i = \frac{1+2+\ldots+i+i}{i+1} = \frac{\frac{i(i+1)}{2}+i}{i+1} = \frac{i}{2} + \frac{i}{i+1} \equiv \frac{i}{2} + \left(1 - \frac{1}{i+1}\right)$$

Insertion Sort (n elementos)
Complexidade Temporal (4)

No caso esperado $O(n^2)$.

Prova (continuação): $\overline{C}_i = \frac{1+2+\ldots+i+i}{i+1} = \frac{\frac{i(i+1)}{2}+i}{i+1} = \frac{i}{2} + \frac{i}{i+1} \equiv \frac{i}{2} + \left(1 - \frac{1}{i+1}\right)$

$$\overline{C} = \overbrace{\left(\frac{1}{2} + \left(1 - \frac{1}{2}\right)\right)}^{\overline{C}_1} + \overbrace{\left(\frac{2}{2} + \left(1 - \frac{1}{3}\right)\right)}^{\overline{C}_2} + \ldots + \overbrace{\left(\frac{n-1}{2} + \left(1 - \frac{1}{n}\right)\right)}^{\overline{C}_{n-1}}$$

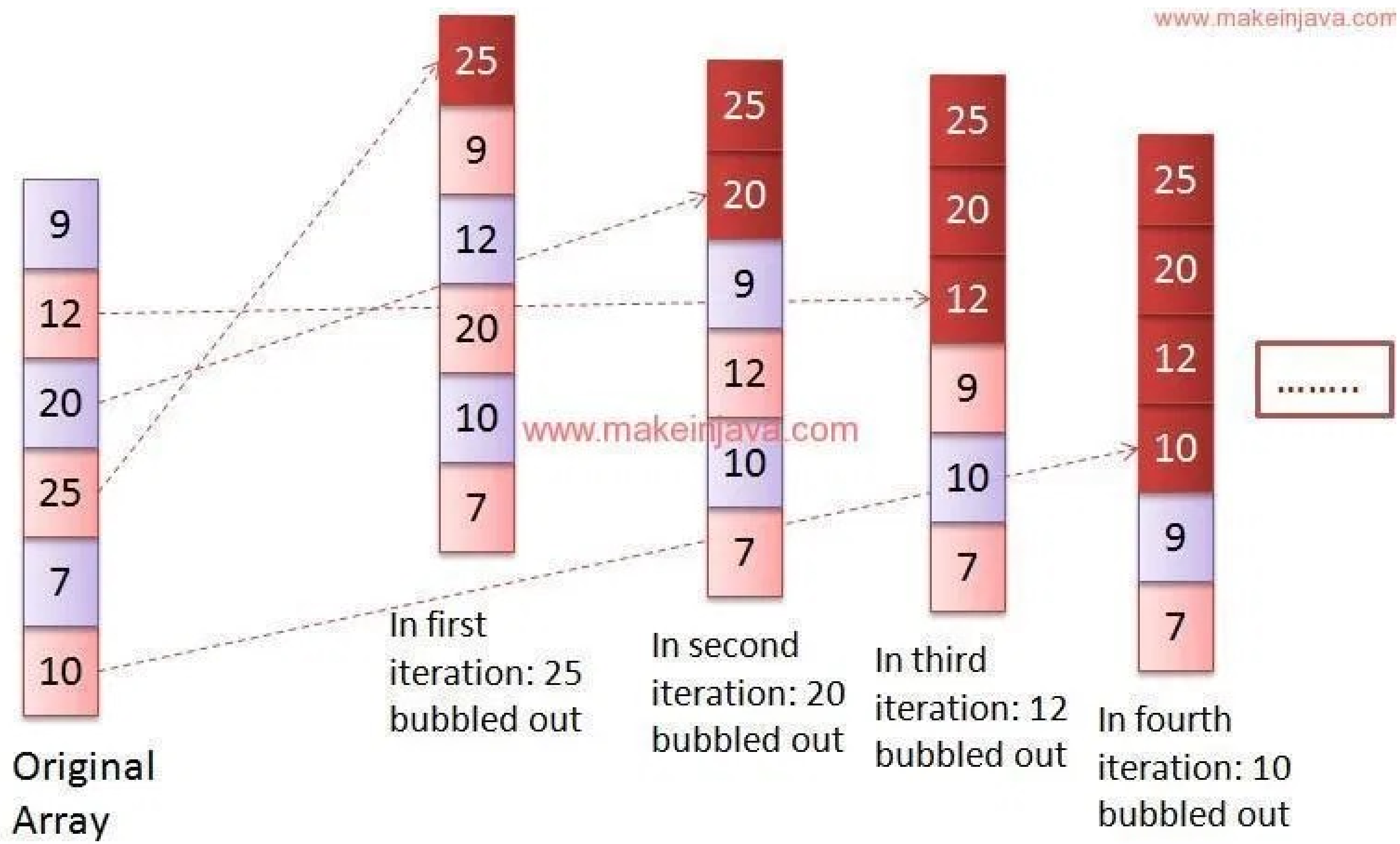
$$= \frac{1}{2}\underbrace{(1+2+\ldots+(n-1))}_{\frac{(n-1)n}{2}} +$$

$$\underbrace{\left(1-\frac{1}{2}\right) + \left(1-\frac{1}{3}\right) + \ldots + \left(1-\frac{1}{n}\right)}_{(n-1)-(H_n-1)=n-H_n}$$

$$= \frac{(n-1)n}{4} + n - H_n \in \Theta(n^2)$$

$H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \ln n$ when $n \to \infty$ is the n-th harmonic number.

Bubble Sort (n elementos)



Em cada **iteração i**:
- garantidamente temos i elementos no lugar certo

https://visualgo.net/

Bubble Sort (n elementos)

Complexidade temporal

Algoritmo estável

```
public static <E> void bubbleSort( E[] vec, int vecSize,
                                               Comparator<E> c ) {
      for ( int i = vecSize - 1; i > 0; i-- )
         for ( int j = 0; j < i; j++ )
            if ( c.compare(vec[j], vec[j + 1]) > 0 )
               swapElements(vec, j, j + 1);
}
```

Em todos os casos

$$\sum_{i=1}^{n-1} i = \frac{(n-1)\,n}{2} = O(n^2).$$

Selection Sort (n elementos)

Em cada **iteração i**:
- procura mínimo entre os desordenados e coloca na posição i

https://visualgo.net/

Selection Sort (n elementos)
Complexidade temporal

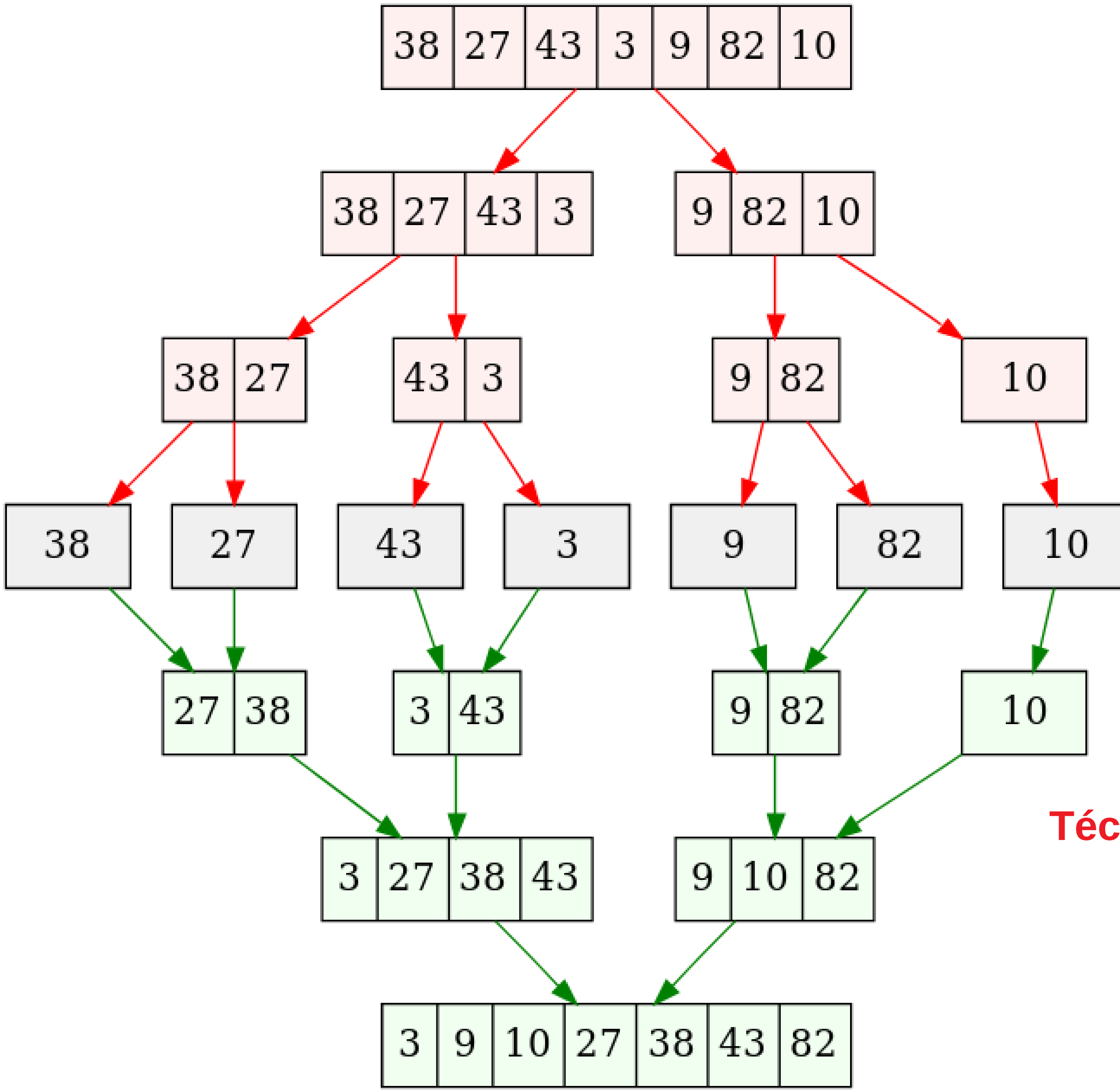
Algoritmo não estável

```
public static <E> void selectionSort( E[] vec, int vecSize,
                                          Comparator<E> c ) {
      for ( int i = 0; i < vecSize; i++ ) {
         int posMinElem = i;
         for ( int j = vecSize-1; j < i; j++ )
            if ( c.compare(vec[posMinElem], vec[j]) > 0 )
               posMinElem = j;
         swapElements(vec, i, posMinElem);
      }
}
```

Em todos os casos

$$\sum_{i=1}^{n-1} i = \frac{(n-1)\,n}{2} = O(n^2).$$

Merge Sort (n elementos)



Consiste em:
- Partição;
- Ordenação;
- Reunião (fusão)

Técnica de divisão e conquista

https://visualgo.net/

Merge Sort (n elementos)

Implementação recursiva (1)

Algoritmo estável

```
@SuppressWarnings("unchecked")
   public static <E> void mergeSortR( E[] vec, int vecSize,
                                          Comparator<E> c ){
      // Variable auxVec declared here to speed up the algorithm.
      // Compiler gives a warning.
      E[] auxVec = (E[]) new Object[vecSize];
      mergeSortR(vec, auxVec, 0, vecSize-1, c);
   }

   protected static <E> void mergeSortR( E[] vec, E[] auxVec,
      int firstPos, int lastPos, Comparator<E> c ){
      if ( firstPos < lastPos ){
         int centre = ( firstPos + lastPos ) / 2;
         mergeSortR(vec, auxVec, firstPos, centre, c);
         mergeSortR(vec, auxVec, centre + 1, lastPos, c);
         mergeR(vec, auxVec, firstPos, centre, lastPos, c);
      }
   }
```

Divisão

Fusão

Merge Sort (n elementos)

Implementação recursiva (2)

```java
protected static <E> void mergeR( E[] vec, E[] auxVec, int firstLeft,
                        int lastLeft, int lastRight, Comparator<E> c ){
    int left = firstLeft;
    int right = lastLeft + 1;
    int result = firstLeft;
    while ( left <= lastLeft && right <= lastRight )
        if ( c.compare(vec[left], vec[right]) <= 0 )
            auxVec[ result++ ] = vec[ left++ ];
        else
            auxVec[ result++ ] = vec[ right++ ];
    // Copy rest of left sequence.
    while ( left <= lastLeft )
        auxVec[ result++ ] = vec[ left++ ];
    // Rest of right sequence in right place.
    // Copy from auxVec to vec.
    // Number of elements to be copied: (result-1) - firstLeft + 1.
    System.arraycopy(auxVec,firstLeft,vec,firstLeft,result-firstLeft);
}
```

Posição a usar em auxVec para inserir o menor elemento

Se os maiores estiverem na sequência do lado direito, essas posição não precisam de ser copiadas

Merge Sort (n elementos)

Como funciona?

60 40 12 55 15 95 23 30 23 81 11 39 71 25 18 47

60 40 12 55 15 95 23 30   23 81 11 39 71 25 18 47

60 40 12 55   15 95 23 30   23 81 11 39   71 25 18 47

60 40   12 55   15 95   23 30   23 81   11 39   71 25   18 47

60 40 12 55 15 95 23 30 23 81 11 39 71 25 18 47

40 60   12 55   15 95   23 30   23 81   11 39   25 71   18 47

12 40 55 60   15 23 30 95   11 23 39 81   18 25 47 71

12 15 23 30 40 55 60 95   11 18 23 25 39 47 71 81

11 12 15 18 23 23 25 30 39 40 47 55 60 71 81 95

Merge Sort (n elementos)

Quantas comparações?

$$C(n) = \begin{cases} 0 & n = 0{,}1 \\ 2\,C\left(\frac{n}{2}\right) + F(n) & n \geq 2 \end{cases} \qquad F(n) = \begin{cases} n-1 & \textit{no pior caso} \\ \frac{n}{2} & \textit{no melhor caso} \end{cases}$$

Divisão

Fusão

Recorrência 2(b)

$$T(n) = \begin{cases} a & n = 0 \quad \textbf{ou} \quad n = 1 \\ bT(\frac{n}{c}) + O(n) & n \geq 1 \qquad\quad n \geq 2 \end{cases}$$

com $a \geq 0, \quad b \geq 1, \quad c > 1$ **constantes**

$$T(n) = \begin{cases} O(n) & b < c \\ O(n \log_c n) & b = c \\ O(n^{\log_c b}) & b > c \end{cases}$$

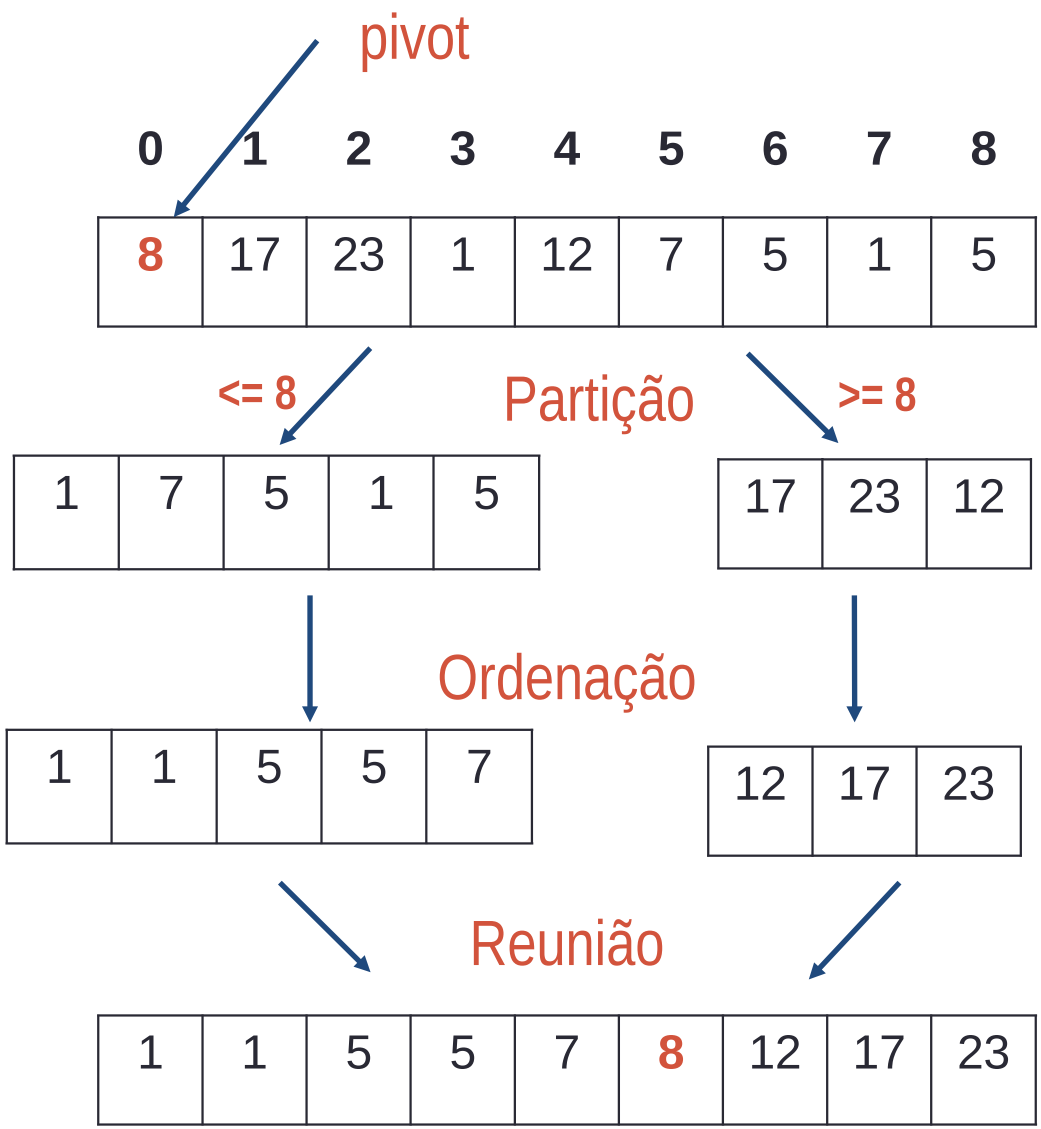
Aplicando a recorrência 2(b) ficamos com

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

Logo

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

Quick Sort (n elementos)



- O algoritmo é constituído por:
  1. Escolha de **pivot**
  2. O vetor é particionado, separando os valores menores que o pivot dos superiores ao pivot (***partição***)
  3. O algoritmo é chamado recursivamente para as partições

Técnica de divisão e conquista

https://visualgo.net/

Quick Sort (n elementos)
Base da recursividade

pivot

0	1	2	3	4	5	6	7	8
8	17	23	1	12	7	5	1	5

<= 8   Partição   >= 8

1	7	5	1	5

17	23	12

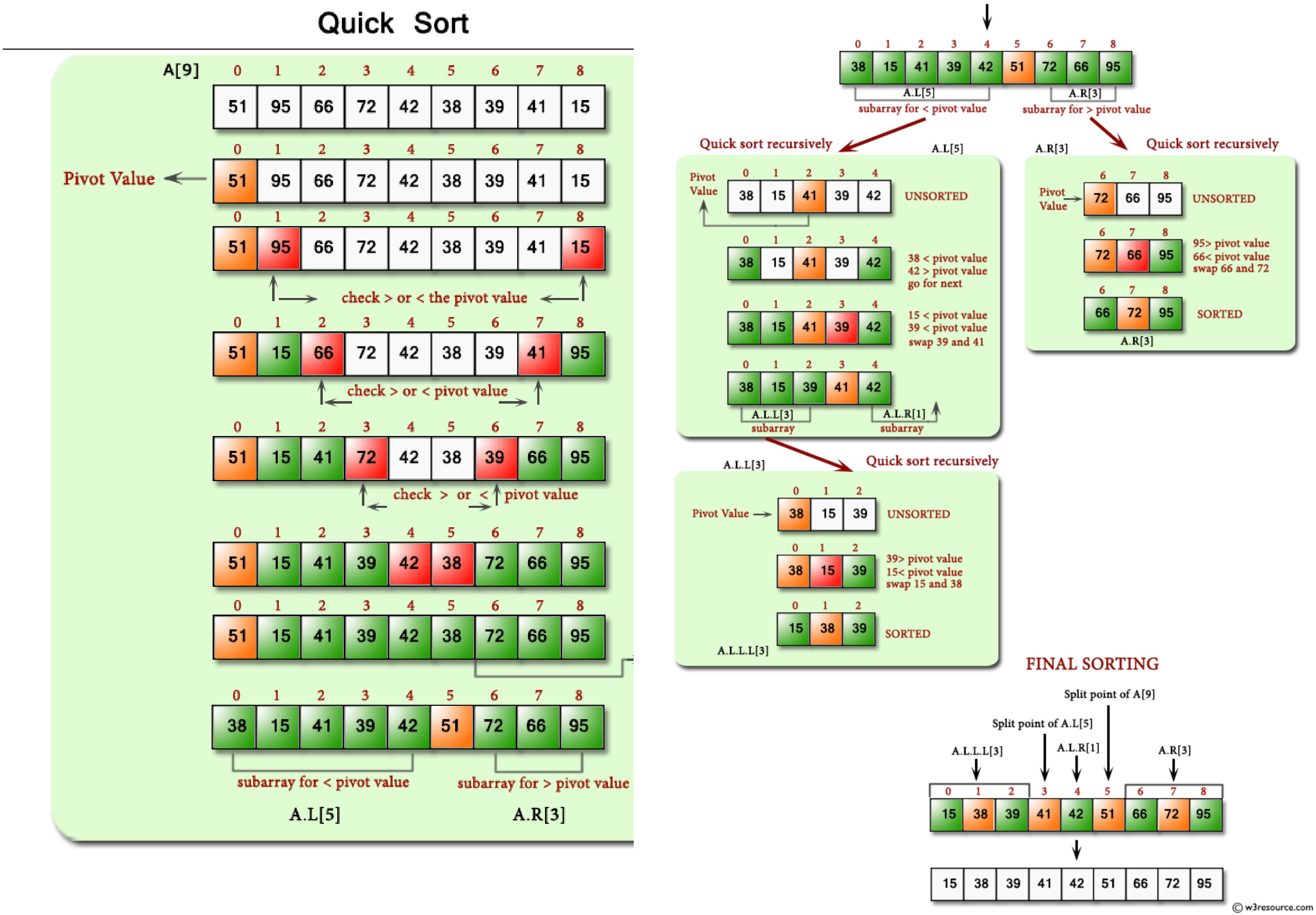
Ordenação

Não ordenar sequências de pequena dimensão com o Quick Sort.
No final, ordenar a sequência total com o ***Insertion Sort***

Reunião

1	1	5	5	7	**8**	12	17	23

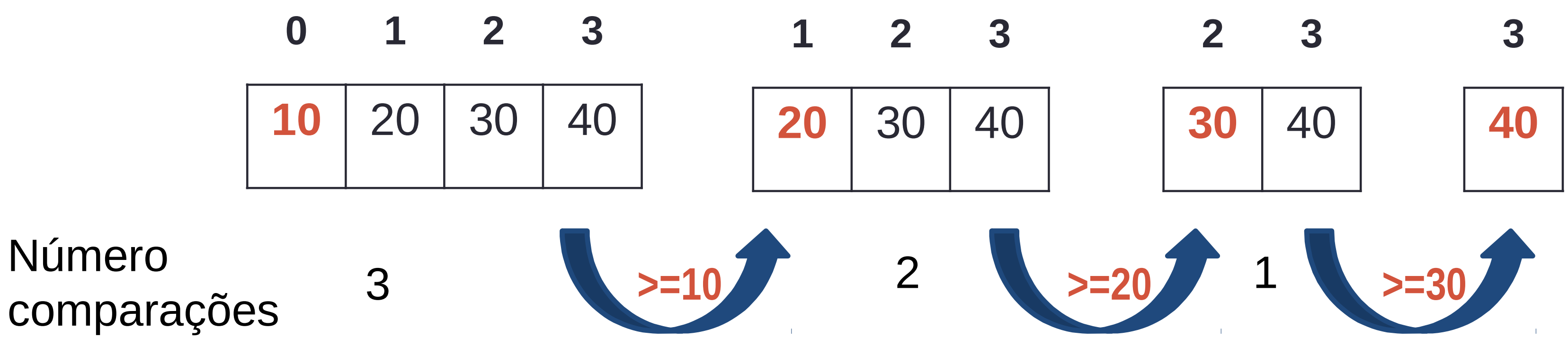
Quick Sort (n elementos)



Quick Sort (n elementos)
Escolha do pivot (1)

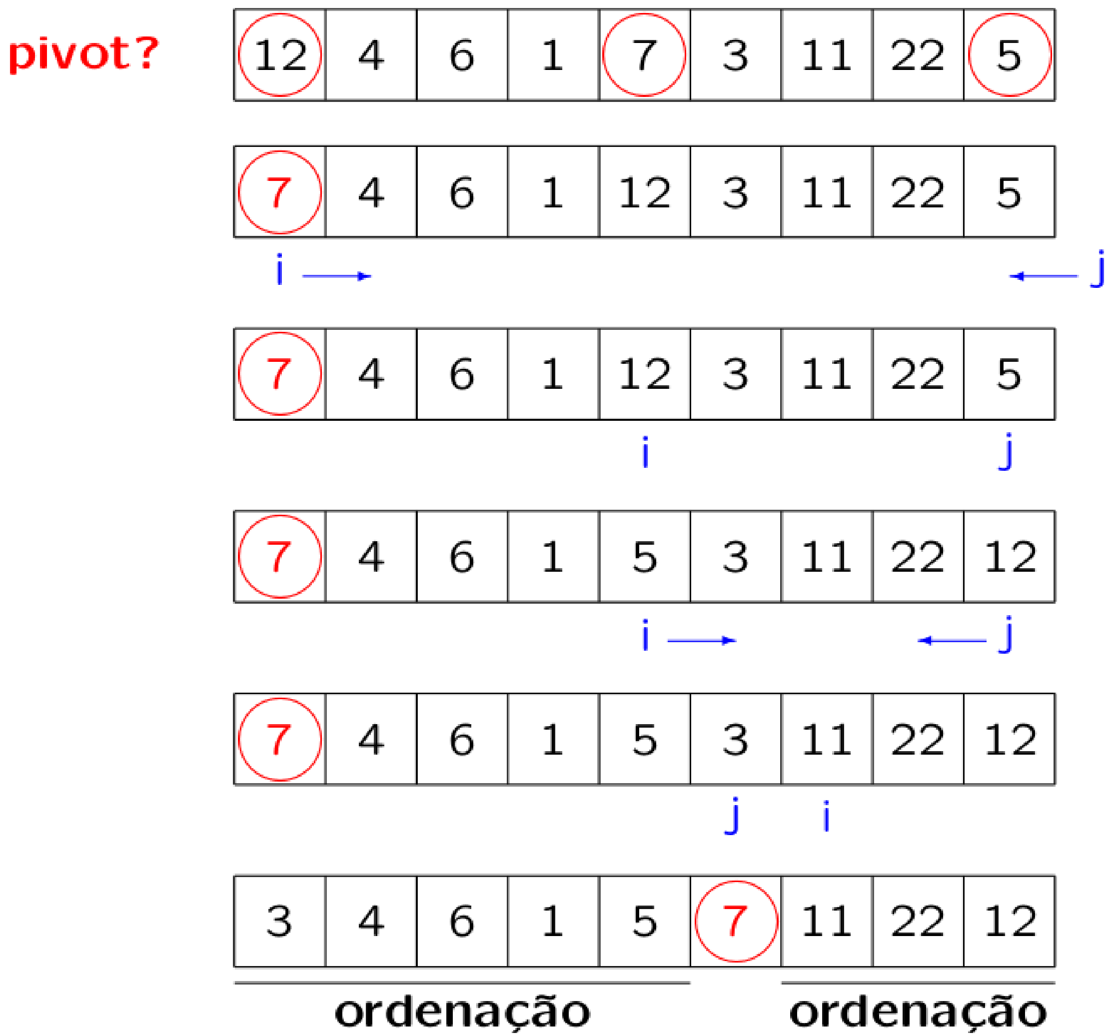
- Se o pivot for sempre o mínimo ou o máximo, o número de comparações para ordenar n elementos é:

$$\sum_{i=1}^{n-1} i = \frac{(n-1)\,n}{2} = O(n^2)$$



Para evitar este caso, o pivot será a mediana entre três elementos: o primeiro, o último e o do meio

Quick Sort (n elementos)
Escolha do pivot (2)



$1^o \longleftrightarrow$ pivot

O **pivot** será a mediana entre três elementos: o primeiro, o último e o do meio.
O pivot fica na 1ª posição

i-ésimo $\longleftrightarrow$ j-ésimo

1. i avança até encontrar
2. elemento >= pivot (ou chegar ao fim)
3. j recua até encontrar
   elemento <= pivot (ou chegar ao início)

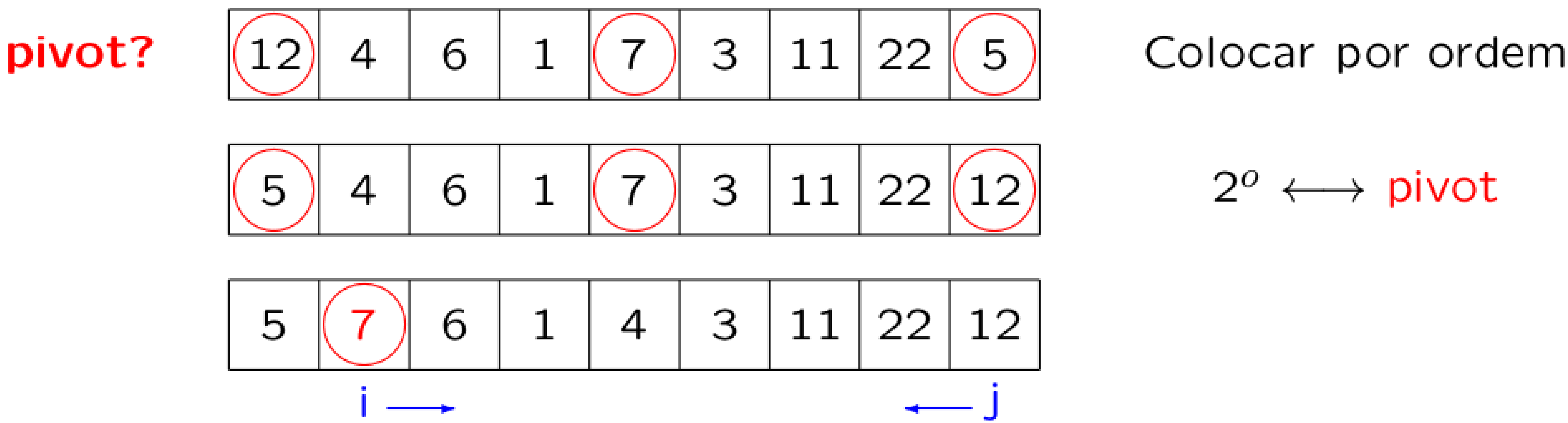
j-ésimo $\longleftrightarrow$ pivot

4. Se i < j trocar i com j
5. Senão troca-se pivot com j
6.

Quick Sort (n elementos)
Escolha do pivot (3)

Garantir, sem testar, que as variáveis (i e j) que controlam a partição não ultrapassam os limites do vector.

O **pivot** será a mediana entre três elementos: o primeiro, o último e o do meio. Mas este elementos ficam já ordenados no vector. O pivot é colocado na 2ª posição.



Consequência: O Algoritmo não é estável

Quick Sort (n elementos)

Escolha do pivot (4)

12	4	6	1	7	3	11	22	5

5	4	6	1	7	3	11	22	12

5	7	6	1	4	3	11	22	12

i → ← j

5	7	6	1	4	3	11	22	12

j i

5	3	6	1	4	7	11	22	12

5	3	6	1	4	7	11	22	12

Quick Sort (n elementos)

Implementação recursiva

Algoritmo estável

```
protected static <E> void quickSortR( E[] vec, int firstPos,
                                        int lastPos, Comparator<E> c ){
    if ( lastPos-firstPos >= 16 ){
        E pivot = median3(vec, firstPos, lastPos, c);
        int i = firstPos + 1;
        int j = lastPos;
        while ( true ){
            do { i++; } while ( c.compare(vec[i], pivot) < 0 );
            do { j--; } while ( c.compare(vec[j], pivot) > 0 );
            if ( i < j )
                swapElements(vec, i, j);
            else break;
        }
        swapElements(vec, firstPos + 1, j); // Restore pivot.
        quickSortR(vec, firstPos, j-1, c);
        quickSortR(vec, j + 1, lastPos, c);
    }
    else
        insertionSort(vec,firstPos,lastPos,c);
}
```

Quick Sort (n elementos)
Complexidade temporal (1)

Número de comparações

$$C(n) = \begin{cases} 0 & n = 0,1 \\ C(k) + C(n - k - 1) + (n-1) & n \geq 2 \end{cases}$$

Os piores casos é o elemento pivot ser o maior ou o menor elemento. Logo k=0 ou k=n-1.

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

Pior caso

$$\boxed{C(n) = O(n^2)}$$

Quick Sort (n elementos)
Complexidade temporal (2)

Número de comparações

$$C(n) = \begin{cases} 0 & n = 0{,}1 \\ C(k) + C(n - k - 1) + (n-1) & n \geq 2 \end{cases}$$

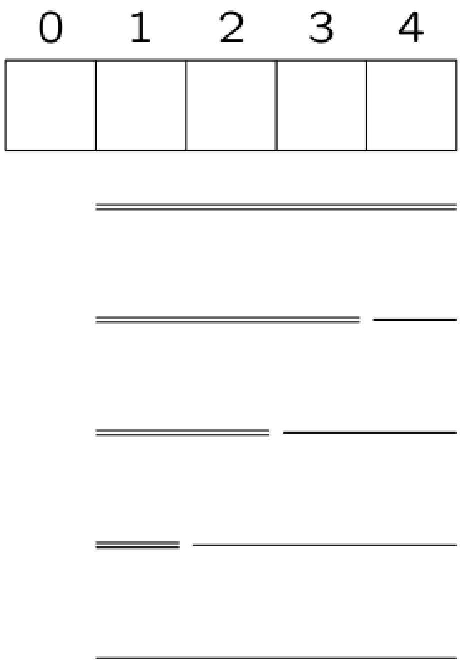
O melhor caso é quando a sequência for dividida ao meio. Logo k=n/2.

$$C(n) = \begin{cases} 0 & n = 0{,}1 \\ 2\,C\left(\frac{n}{2}\right) + (n-1) & n \geq 2 \end{cases}$$

Melhor caso

$$\boxed{C(n) = O(n \log n)}$$

Quick Sort (n elementos)
Complexidade temporal (3)



Número de comparações

$$\frac{1}{5}\left(C(4) + C(0) + 4 \right)$$

$$\frac{1}{5}\left(C(3) + C(1) + 4 \right)$$

$$\frac{1}{5}\left(C(2) + C(2) + 4 \right)$$

$$\frac{1}{5}\left(C(1) + C(3) + 4 \right)$$

$$\frac{1}{5}\left(C(0) + C(4) + 4 \right)$$

$$C(n) = \begin{cases} 0 & n = 0, 1 \\ \left(\frac{2}{n}\sum_{k=0}^{n-1} C(k)\right) + (n-1) & n \geq 2 \end{cases}$$

$$\frac{1}{5}\left(2C(0) + 2C(1) + \cdots + 2C(4)\right) + 4 \overset{(n=5)}{=} \left(\frac{2}{n}\sum_{k=0}^{n-1} C(k)\right) + (n-1)$$

Caso médio

$$\boxed{C(n) = O(n \log n)}$$

Algoritmos de ordenação

	Melhor Caso	Pior Caso	Caso Esperado	Estabilidade
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	Sim
Bubble	$O(n^2)$	$O(n^2)$	$O(n^2)$	Sim
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	Não
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Sim
Quick	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	Não