ENGENHARIA INFORMÁTICA E DE COMPUTADORES

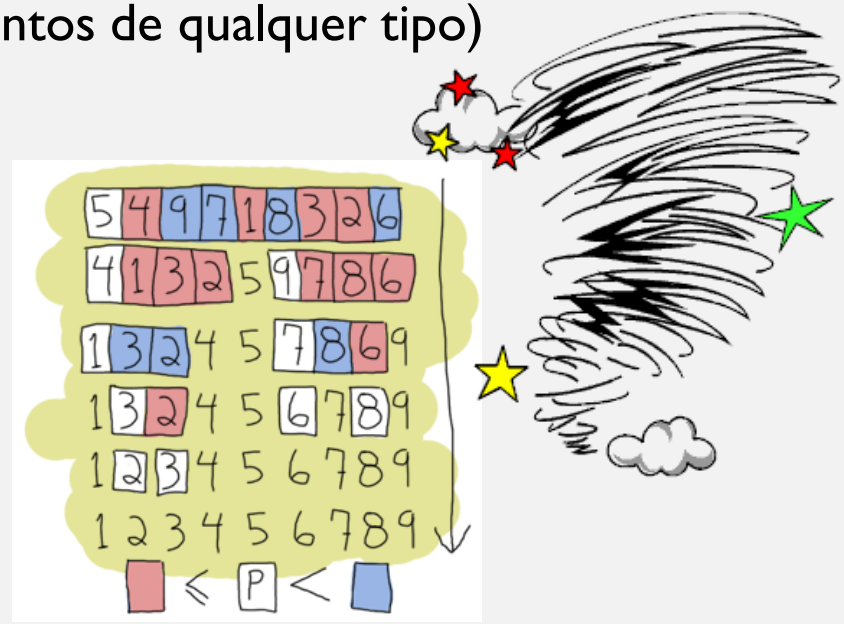
Algoritmos e Estruturas de Dados

(parte 8 – Quick Sort)

2° Semestre 2022/2023
Instituto Superior de Engenharia de Lisboa
Paula Graça

QUICK SORT

- O *quick sort* é o algoritmo mais rápido de ordenação conhecido (considerando uma lista de elementos de qualquer tipo)
  - O algoritmo base foi inventado por em 1960 por C.A.R. Hoare
  - É do tipo "dividir para conquistar"



  - É mais rápido e consome menos recursos do que qualquer outro algoritmo de ordenação

QUICK SORT

- Ideia do algoritmo: efectuar partição dos elementos e ordenar as várias partes independentemente (de forma recursiva)
  - A posição de partição a efectuar depende da ordenação inicial dos elementos
  - O processo de partição é crítico
  - Uma vez efectuada a partição, cada uma das partes pode por sua vez ser ordenada pelo mesmo algoritmo

QUICK SORT - DESCRIÇÃO

- A ordenação é feita através dos seguintes passos
  - partição do *array* em dois *sub-arrays*

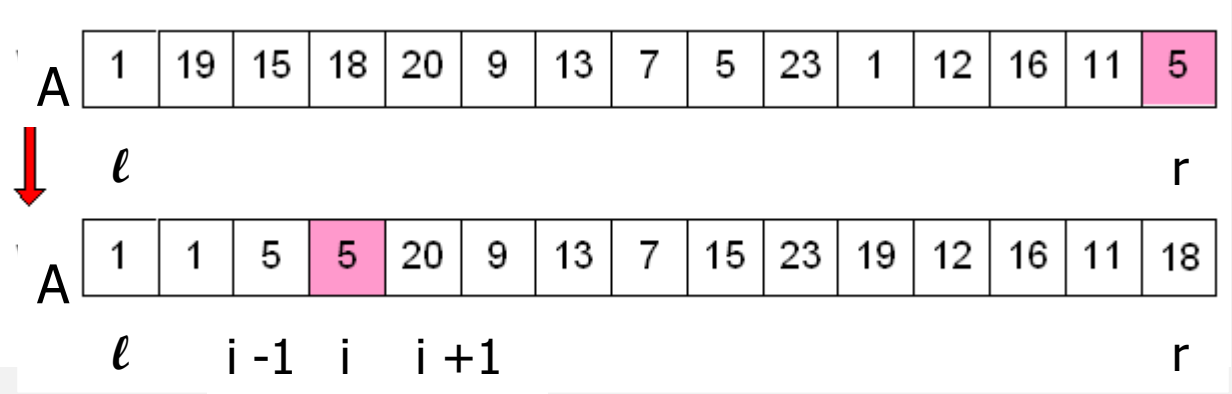
    +
  - aplicação recursiva do algoritmo aos dois *sub-arrays* resultantes

- Em cada processo de partição, pelo menos um elemento fica na sua posição final (*pivot*)

- Após partição, o *array* fica subdividido em dois *sub-arrays* que podem ser ordenadas separadamente

QUICK SORT - PARTIÇÃO

- A partição do *array* inicial rearruma-o em dois *sub-arrays* da seguinte forma:
  - Escolhe-se A[r] arbitrariamente para ser o elemento de partição (*pivot*)
  - Arrumam-se todos os elementos de tal forma que
    - O *pivot* fica na sua posição final em A[i]
    - Todos os elementos em A[ℓ .. i -1] são menores ou iguais ao *pivot*
    - Todos os elementos em A[i +1 .. r] são maiores que o *pivot*



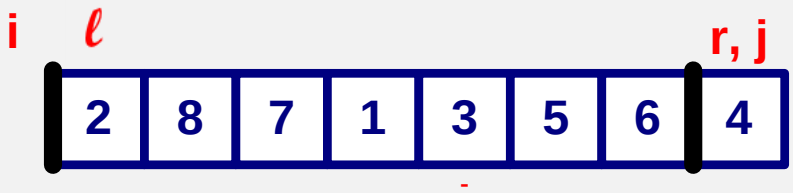
- Repete-se a ordenação para os dois *sub-arrays* A[ℓ .. i -1] e A[i +1 .. r], chamando recursivamente o *quicksort*

QUICK SORT (**HOARE**)

- Estratégia de partição
  - Escolhe-se A[r] arbitrariamente para ser o elemento de partição (*pivot*)
  - Percorre-se o *array* a partir da esquerda (i) até encontrar um elemento A[i] maior que o *pivot*
  - Percorre-se o *array* a partir da direita (j) até encontrar um elemento A[j] menor que o *pivot*
  - Trocam-se os elementos A[i] com A[j]
  - O procedimento continua para todos os elementos ainda não examinados
  - Completa-se trocando o *pivot* A[r] com i-ésimo elemento A[i], ficando o *pivot* na posição final de ordenação

QUICK SORT (HOARE)

a)



b)

c)

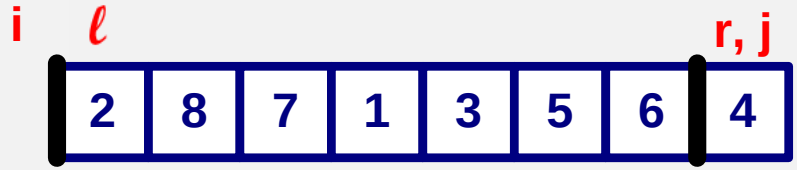
d)

e)

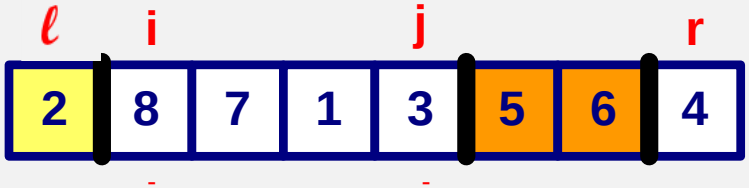
f)

QUICK SORT (HOARE)

a)



b)



c)

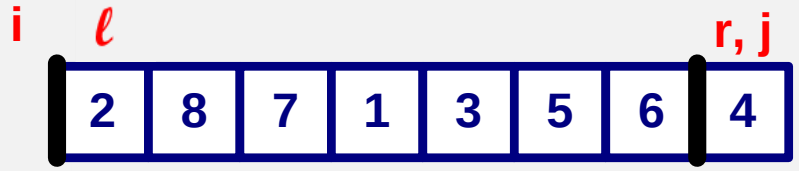
d)

e)

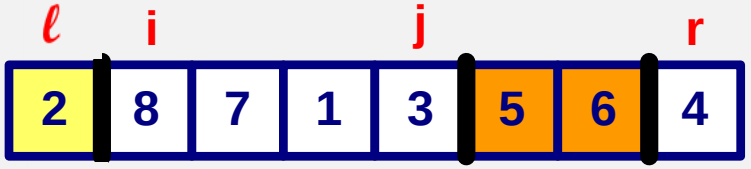
f)

QUICK SORT (HOARE)

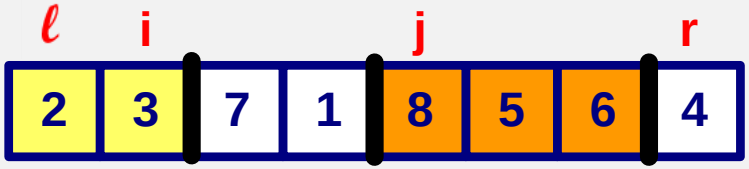
a)



b)



c)



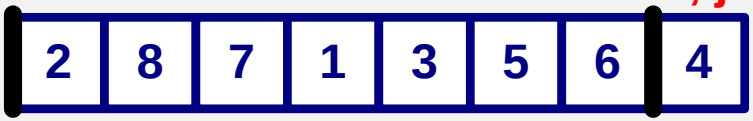
d)

e)

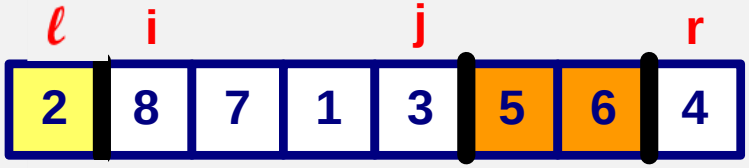
f)

QUICK SORT (HOARE)

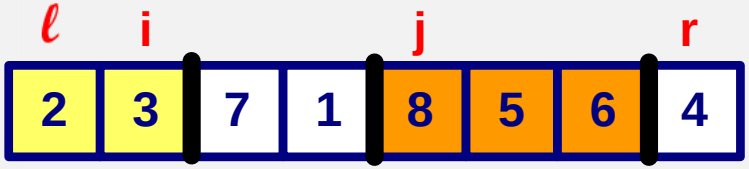
a)



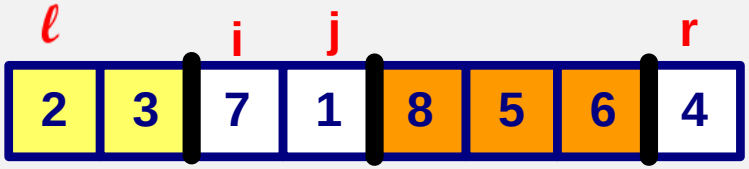
b)



c)



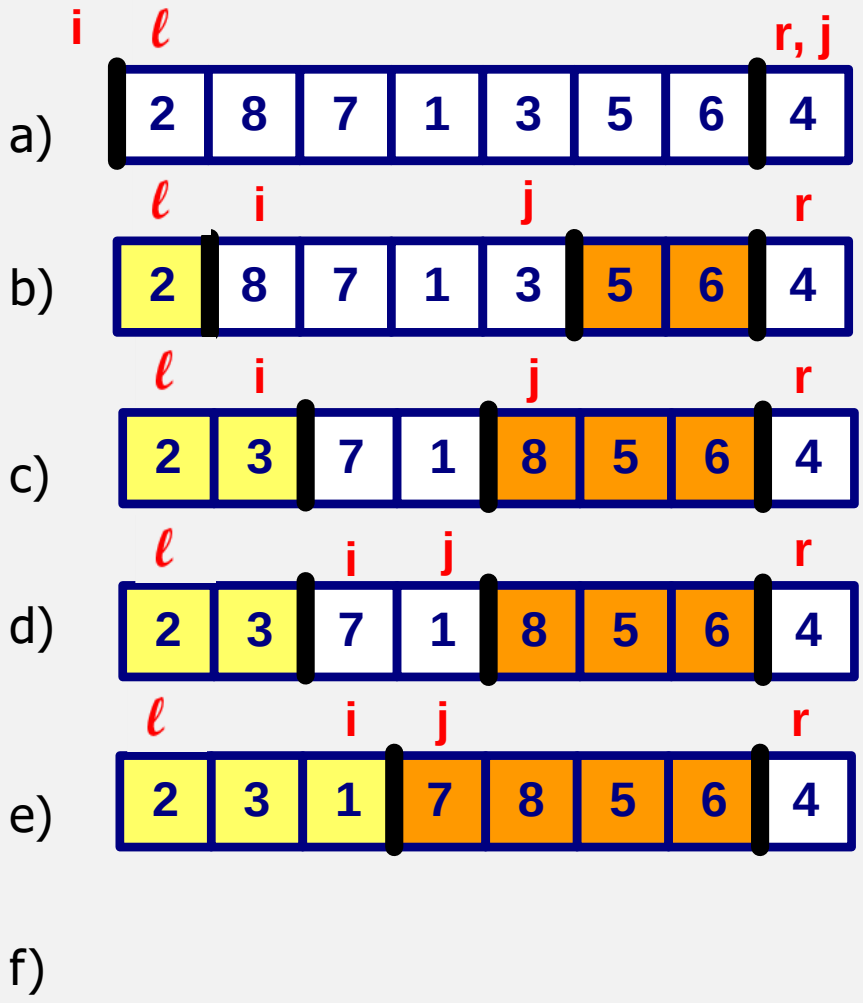
d)



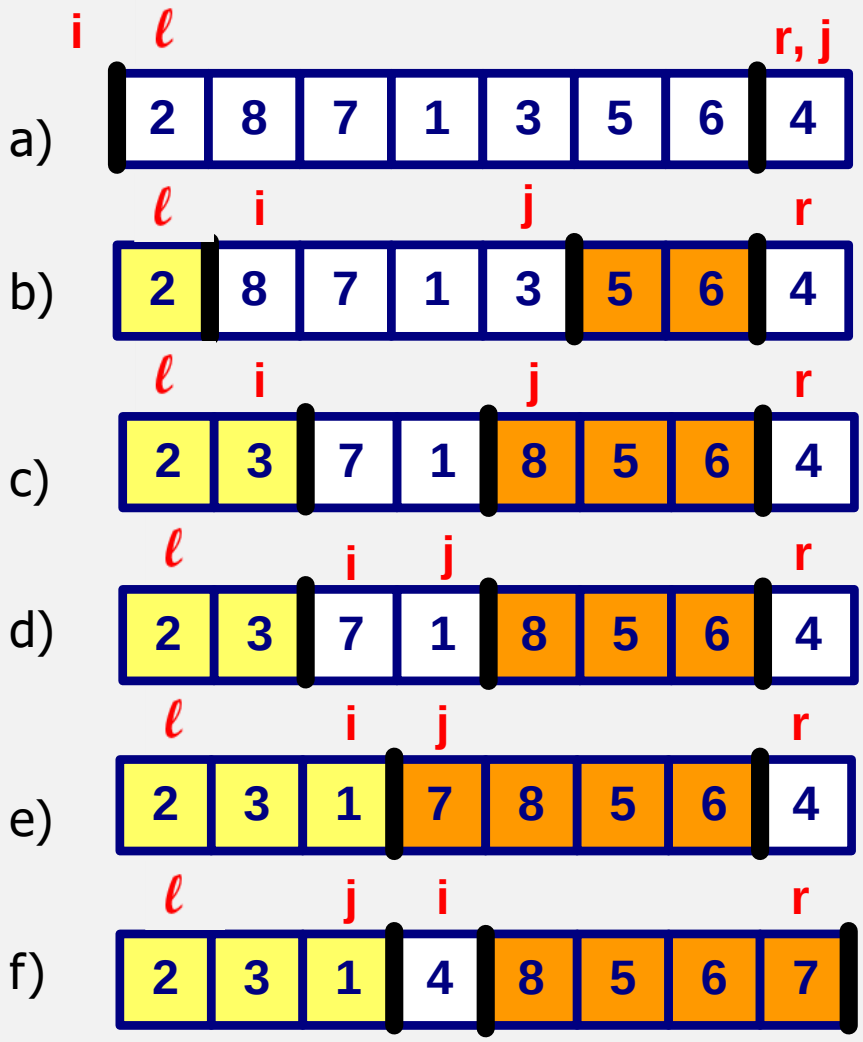
e)

f)

QUICK SORT (HOARE)



QUICK SORT (HOARE)



QUICK SORT – IMPLEMENTAÇÃO (HOARE)

```
fun partition(a: IntArray, left: Int, right: Int): Int {
    var i =  left – 1          // i inicia antes do índice da esquerda
    var j = right              // j inicia após o índice da direita
    val pivot =  a[right]      // o pivot está no indice da direita
    while (true) {
        while (i < right && a[++i] < pivot);
        while (j > left && a[--j] > pivot);
        if (i >= j) break
        exchange(a, i, j)
    }
    exchange(a, i, right)
    return i
}
```

QUICK SORT – IMPLEMENTAÇÃO (HOARE)

```
fun quickSort(a: IntArray, left: Int, right: Int) {
    if (left < right) {
        val i = partition(a, left, right)
        quickSort(a, left, i - 1)
        quickSort(a, i + 1, right)
    }
}
```

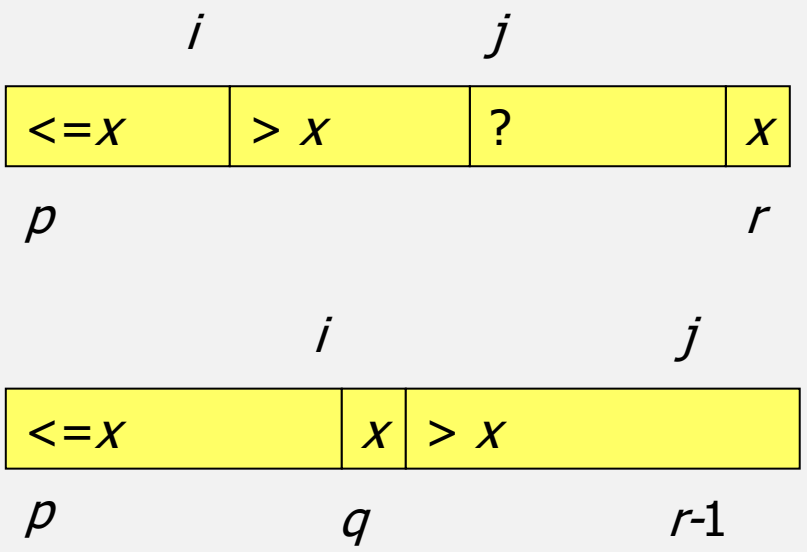
```
fun  exchange(a: IntArray, i: Int, j: Int) {
    val x = a[i]
    a[i] = a[j]
    a[j] = x
}
```

QUICK SORT – ALGORITMO (**LOMUTO**)

```
QuickSort (A, p, r)
    if p < r
        q = Partition(A, p, r)
        QuickSort(A, p, q – 1)
        QuickSort(A, q + 1, r)
```

Para ordenar o *array*, a chamada inicial é

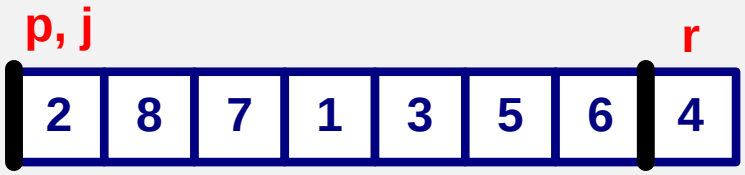
QuickSort(A, 1, A.length)




```
Partition (A, p, r)
    x = A[r]
    i = p – 1
    for j = p to r – 1
        if A[j] ≤ x
            i = i + 1
            exchange A[i] with A[j]
    exchange A[i+1] with A[r]
    return i + 1
```

QUICK SORT (LOMUTO)

a)



b)

c)

d)

e)

f)

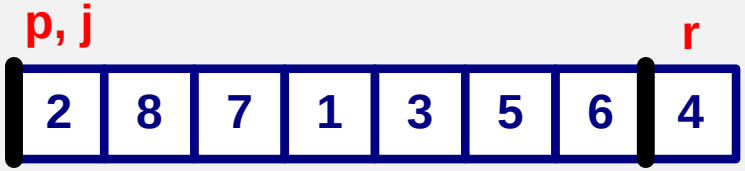
g)

h)

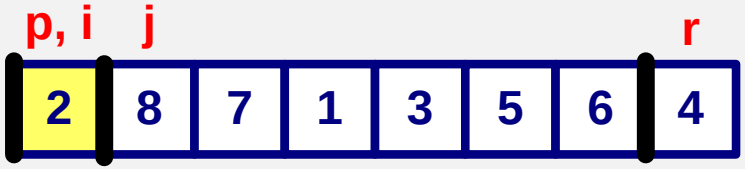
i)

QUICK SORT (LOMUTO)

a)



b)



c)

d)

e)

f)

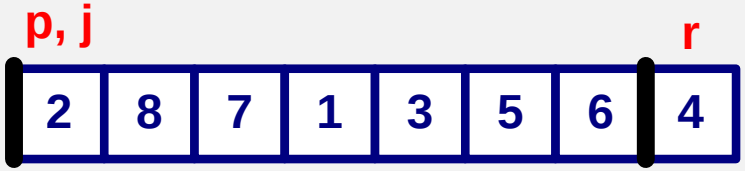
g)

h)

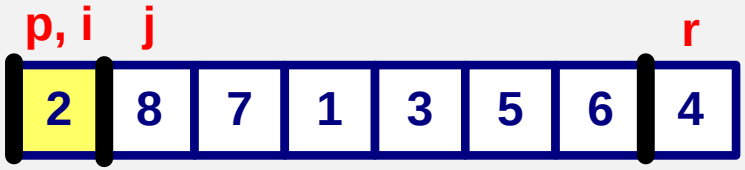
i)

QUICK SORT (LOMUTO)

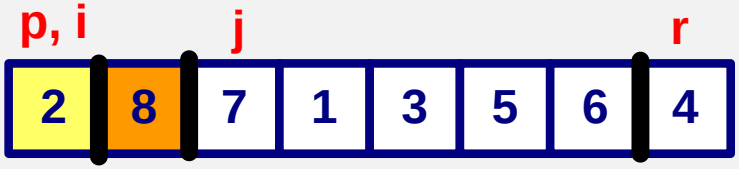
a)



b)



c)



d)

e)

f)

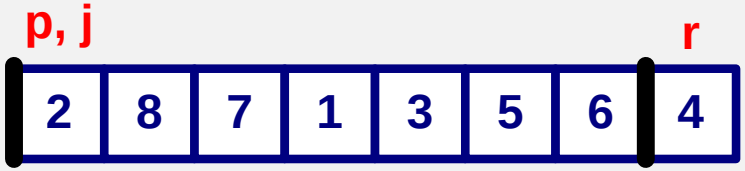
g)

h)

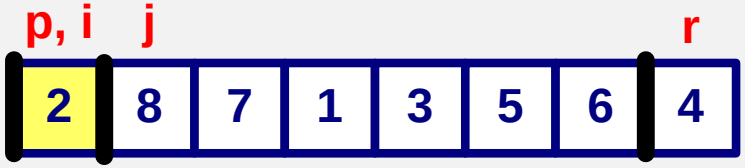
i)

QUICK SORT (LOMUTO)

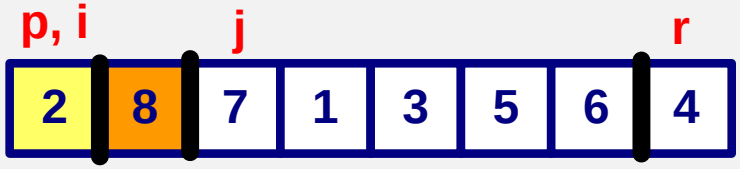
a)



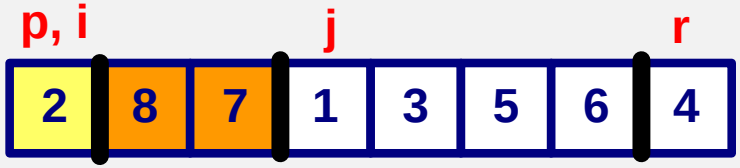
b)



c)



d)



e)

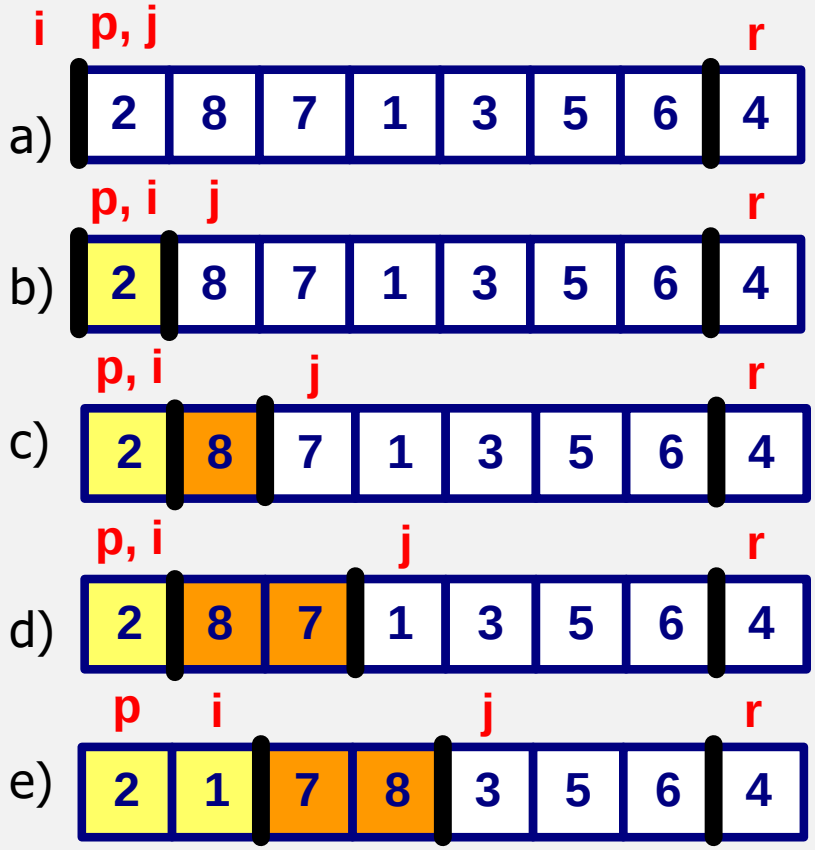
f)

g)

h)

i)

QUICK SORT (LOMUTO)



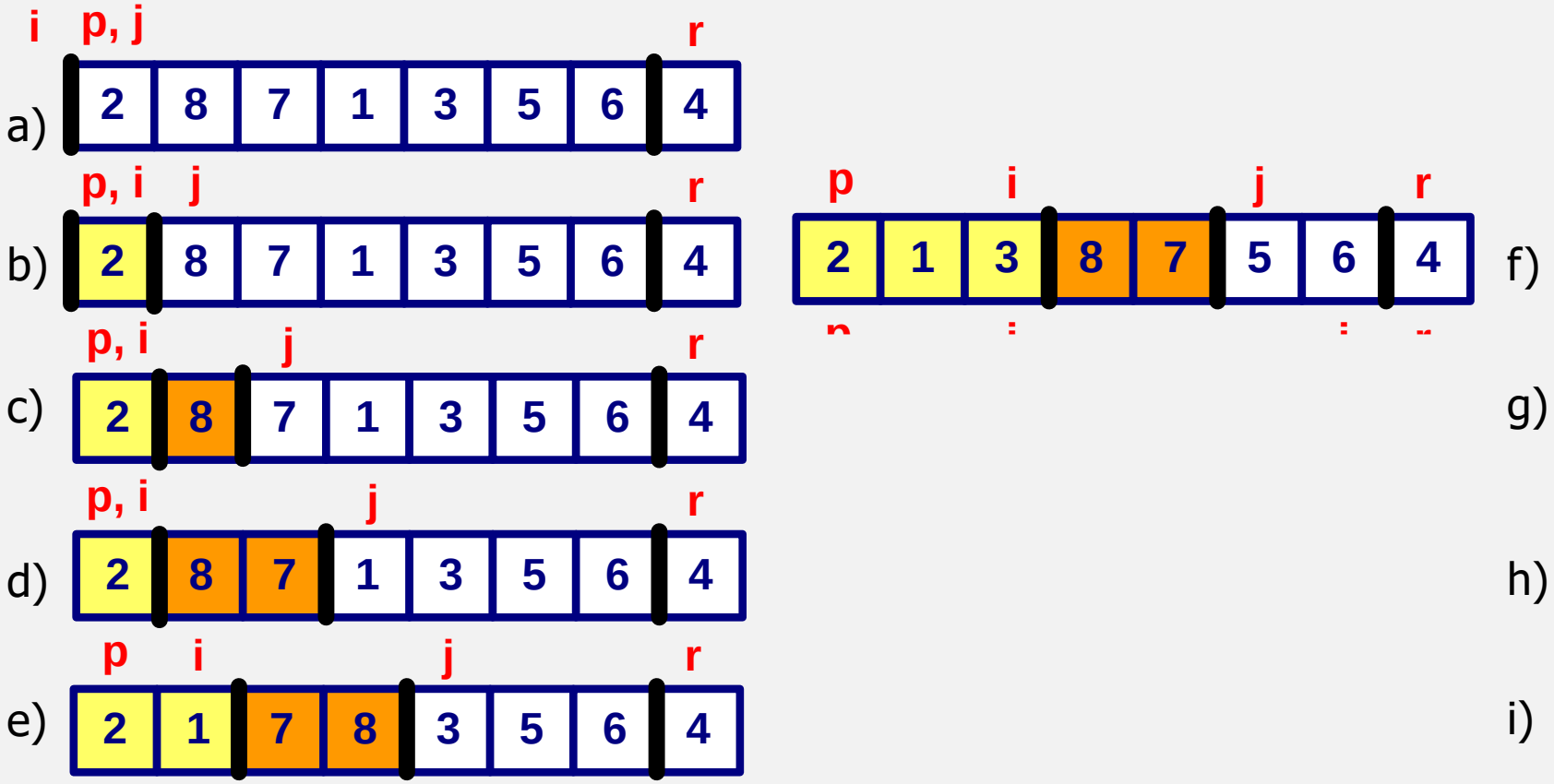
f)

g)

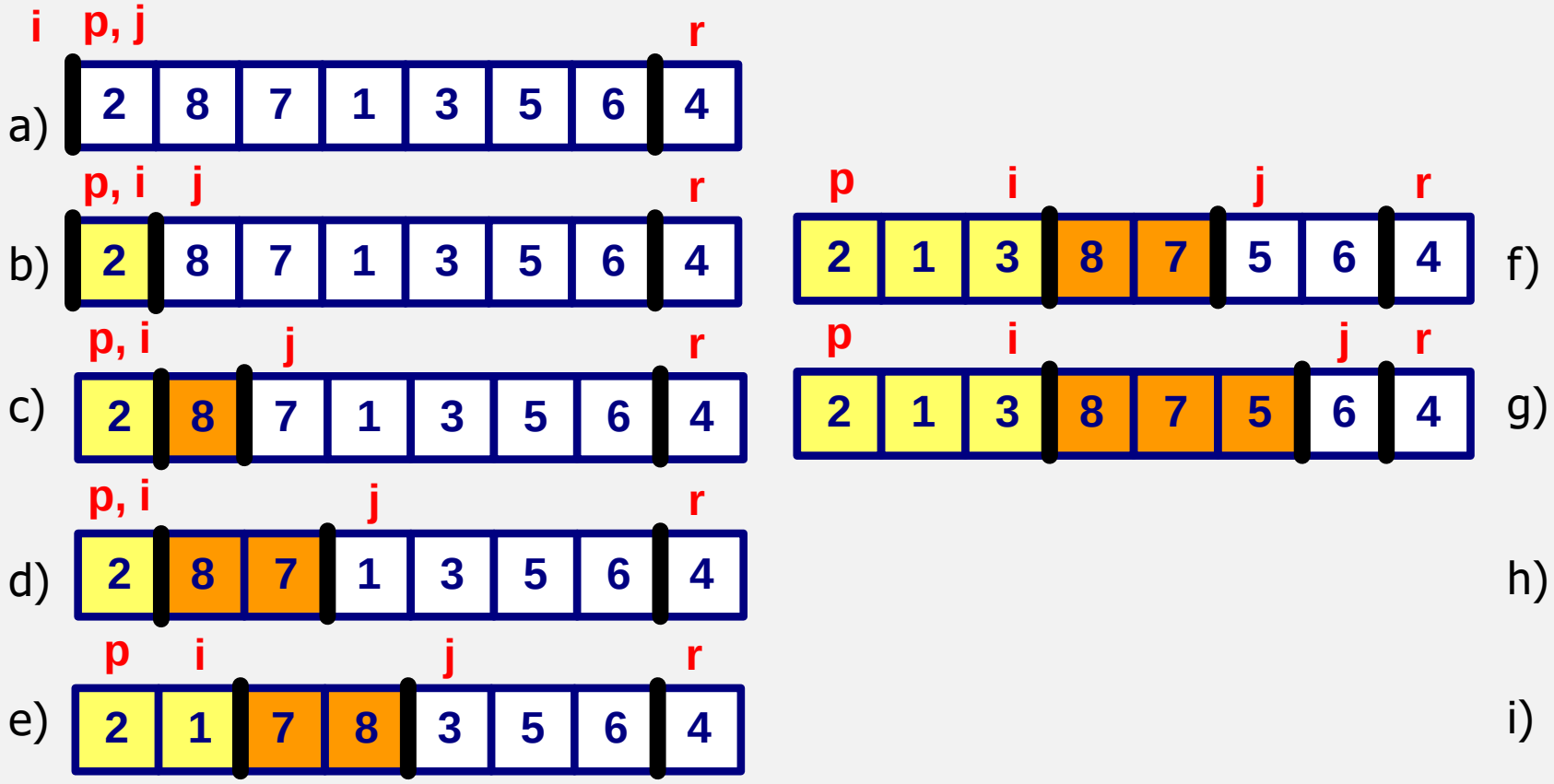
h)

i)

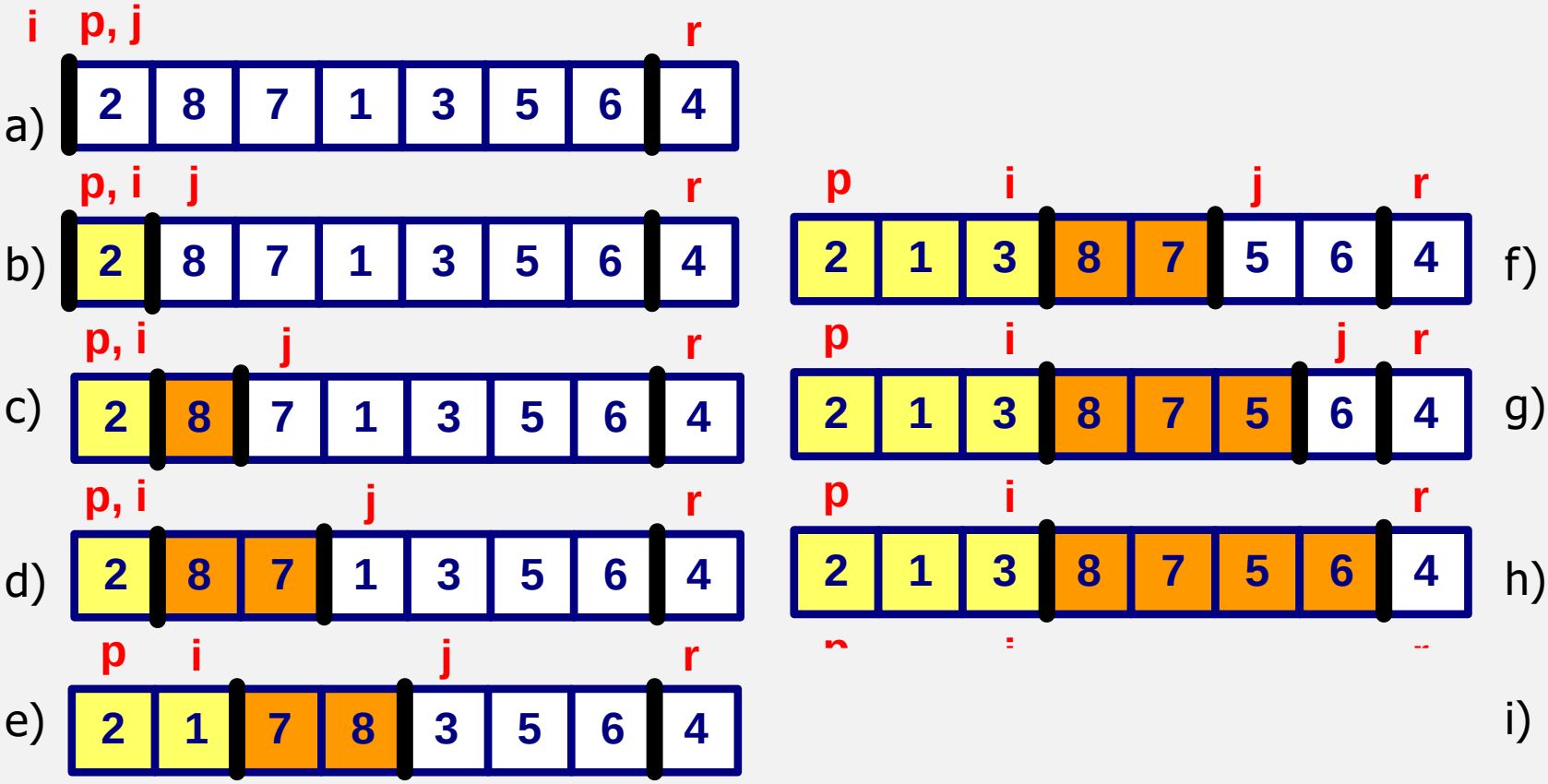
QUICK SORT (LOMUTO)



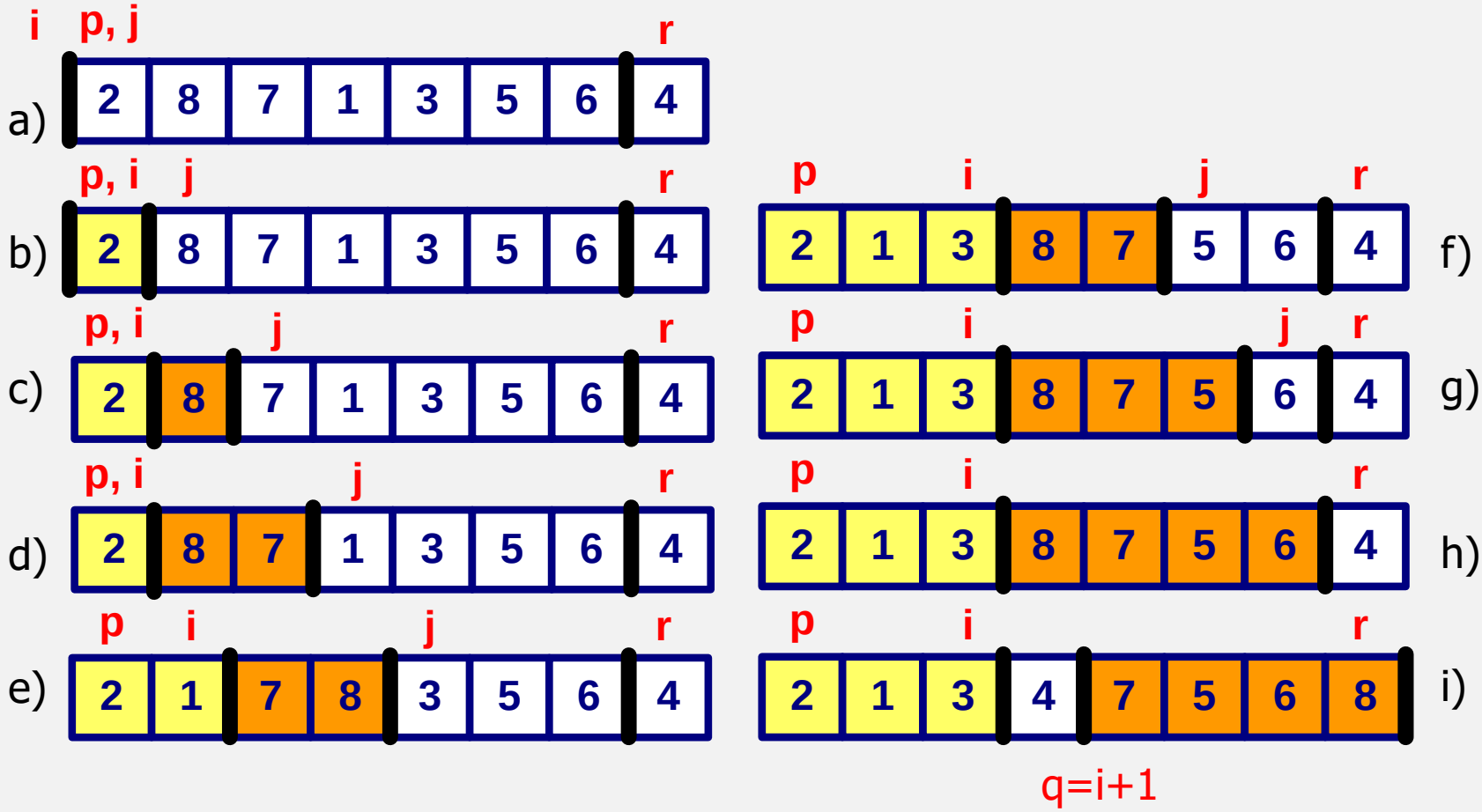
QUICK SORT (LOMUTO)



QUICK SORT (LOMUTO)



QUICK SORT (LOMUTO)



ANÁLISE DO QUICK SORT

- A eficiência do processo de ordenação depende de como a partição divide o *array*, ou seja, depende do elemento de partição
- A partição será tanto mais equilibrada quanto mais perto o elemento de partição, estiver no centro
- A escolha do *pivot* é crítica
- Existem várias estratégias para o *pivot*
  - É sempre escolhido o último elemento (solução implementada)
  - É escolhido um elemento aleatoriamente (boa solução)
  - É escolhida a mediana de entre 3 elementos escolhidos aleatoriamente

ANÁLISE DO QUICK SORT

- Melhor caso 
  - É quando cada partição divide o *array* exatamente ao meio
  - Trata-se de um algoritmo "dividir para conquistar"

Recorrência:

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

O custo do *partition* é $O(n)$ com $n = r - p + 1$

Duas chamadas recursivas do *quicksort* , para cada uma das metades resultantes da partição

Solução: $T(n) = O(n \lg n)$

ANÁLISE DO QUICK SORT

- Pior caso 
  - Ocorre quando a função de partição produz um *sub-array* com n -1 elementos e outro com zero
  - Assume-se que esta partição desequilibrada existe em cada chamada recursiva

Caso base com zero ou 1 elemento, termina em $O(1)$

Sub-array com n-1 elementos

Sub-array com zero elementos

Custo do *partition*

Recorrência:

$$\begin{cases} T(0) = T(1) = 1 \\ T(n) = T(n-1) + T(0) + n \end{cases}$$

Solução: $T(n) \approx n^2/2 = O(n^2)$

ANÁLISE DO QUICK SORT

- Caso médio 
  - É o caso mais frequente, quando os elementos estão distribuídos aleatoriamente
  - Sendo i = 0, 1, 2, ..., n-1, cada partição contém entre i e n-1-i elementos, respetivamente (o i'ésimo é o pivot):

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

    Custo do *partition*
  - Cada partição tem uma probabilidade de 1/n

Recorrência:
$$T(n) = \frac{1}{n}\sum_{i=0}^{n-1} T(i) + T(n-1-i) + n =$$
$$= \frac{2}{n}\left[T(0) + T(1) + \cdots + T(n-2) + T(n-1)\right] + n$$

Desenvolvendo:
$$\frac{T(n)}{n+1} - \frac{T(n-1)}{n} = \frac{2}{n+1}$$

ANÁLISE DO QUICK SORT

- Caso médio (cont.)

Continuando a desenvolver:

$$\frac{T(n)}{n+1} = \frac{T(0)}{1} + 2\left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} + \frac{1}{n+1}\right)$$

$$\frac{T(n)}{n+1} \approx 2(H_{n+1} - 1)$$

$$T(n) \approx n.\,2(H_{n+1} - 1)$$

$$H_n = \sum_{k=1}^{n} \frac{1}{k} \approx \ln n + \gamma = O(\lg n)$$

Solução: T(n) ≈ 2n O(lg n) = O (n lg n)

QUICK SORT – DOIS *PIVOTS*

1. Para a partição, escolhem-se dois elementos como pivots **P1** e **P2**
   - **A[left]** para **P1** e **A[right]** para **P2**
2. O *pivot* **P1** tem que ser menor que **P2**, senão trocam-se. O *array* fica então dividido em 4 partes
   - **parte I** (índices de **left+1** até **L–1**) com os elementos menores que **P1**
   - **parte II** (índices de **L** a **K–1**) com os elementos maiores ou iguais a **P1** e menores ou iguais a **P2**
   - **parte III** (índices de **G+1** a **right–1**) com os elementos maiores que **P2**
   - **parte IV** (índices de **K** a **G**) contém os restantes elementos ainda não visitados
3. O próximo elemento **A[K]** da parte **IV** é comparado com os dois *pivots* **P1** e **P2**, sendo colocado na parte respetiva **I**, **II**, ou **III**
4. Os apontadores **L**, **K**, e **G** vão sendo modificados nas direções correspondentes
5. Os passos 3 - 4 são repetidos enquanto **K** ≤ **G**
6. O *pivot* **P1** é trocado com o último elemento da parte **I**, e o *pivot* **P2** é trocado com o primeiro elemento da parte **III**
7. Os passos 1 - 6 são repetidos recursivamente para as três partes **I**, **II**, e **III**