Aula 7

Professores:

Dante Corbucci Filho
Leandro A. F. Fernandes

Conteúdo:

- Moda
- Algoritmos de Ordenação
  - Método da Seleção (*SelectionSort*)
  - Método da Bolha (*BubbleSort*)
  - Método da Partição (*QuickSort)*
- Noções de Complexidade de Algoritmos

Busca pela Moda de um Vetor

- A moda de um vetor é o elemento que aparece com mais frequência neste vetor.

Busca pela Moda de um Vetor

- A moda de um vetor é o elemento que aparece com mais frequência neste vetor.

- O problema da busca pela moda consiste em encontrar este elemento no vetor.

Busca pela Moda de um Vetor

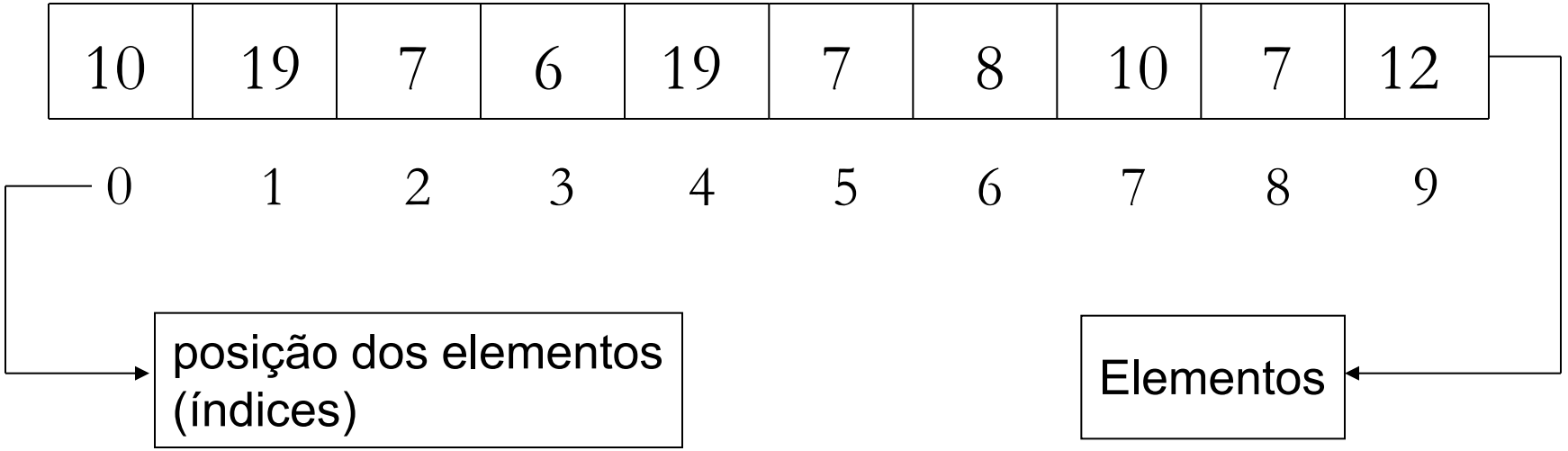
- A moda de um vetor é o elemento que aparece com mais frequência neste vetor.

- O problema da busca pela moda consiste em encontrar este elemento no vetor.

- Sem perda de generalidade, o problema será atacado considerando-se que:
  - Os elementos do vetor são numéricos;
  - Mais de um elemento pode aparecer o mesmo número de vezes e ser moda. Nesse caso, o que aparecer primeiro no vetor será apresentado como resposta.

Busca pela Moda de um Vetor

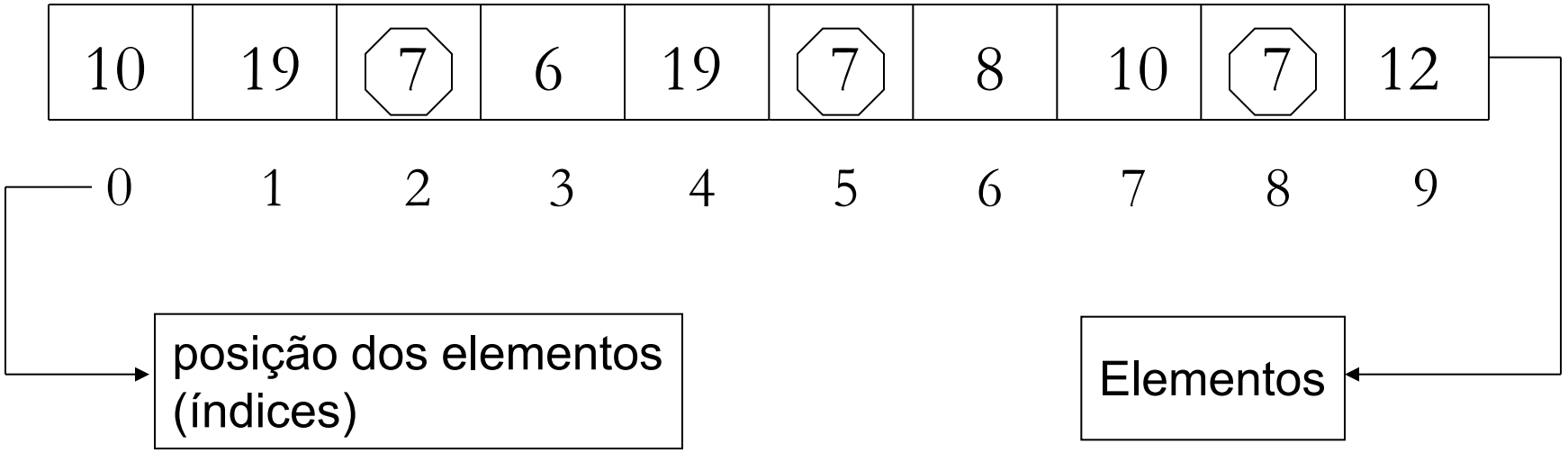
valores: contém um grupo de inteiros



Qual é a Moda deste vetor **valores**?

Busca pela Moda de um Vetor

valores: contém um grupo de inteiros



Moda deste vetor **valores**: elemento 7 (que ocorre 3 vezes).

Busca pela Moda de um Vetor (Exemplo)

```
# Subprogramas

def preenche(valores):
    for ind in range(len(valores)):
        valores[ind] = int(input("Elemento["+str(ind)+"]="))
    return None

def buscaModa(valores):
    ...

# Programa Principal de Busca o Elemento da Moda
numeros = [0]*10
preenche(numeros)
moda = buscaModa(numeros)

print("A moda do vetor  elemento da é:", moda)
```

Busca Simples pela Moda

Todas as posições do vetor são comparadas com os elementos seguintes do vetor.

valores

10	19	7	6	19	7	8	10	7	12
0	1	2	3	4	5	6	7	8	9

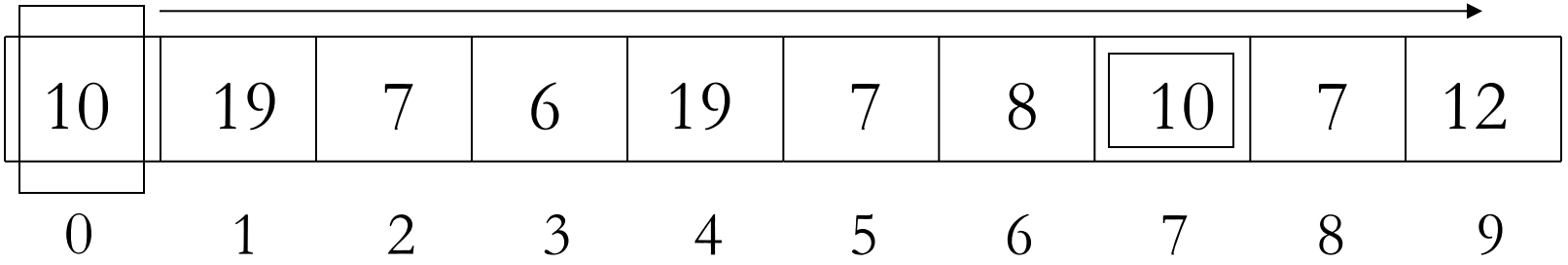
auxiliar (contém a frequência de cada elemento do valores)

0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9

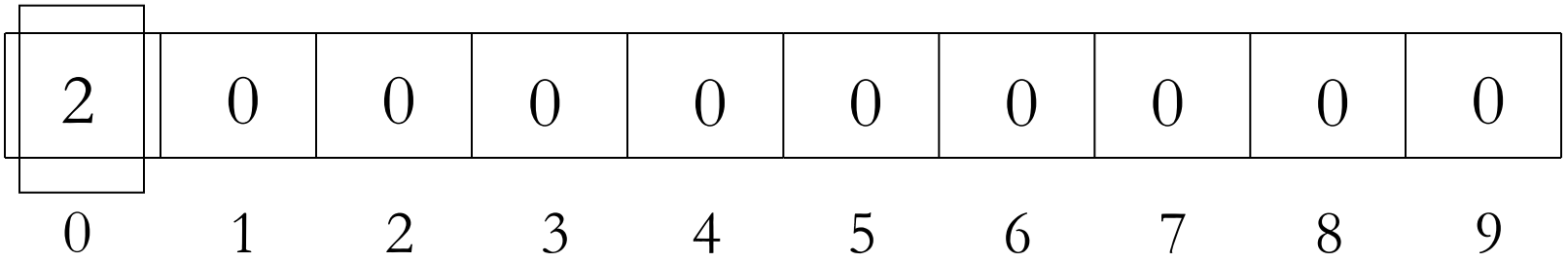
Busca Simples pela Moda

Todas as posições do vetor são comparadas com os elementos seguintes do vetor.

valores



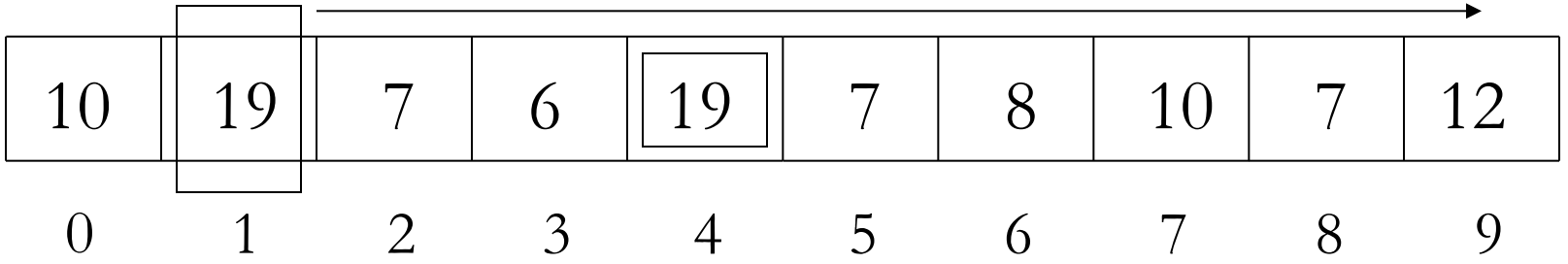
auxiliar (contém a frequência de cada elemento do valores)



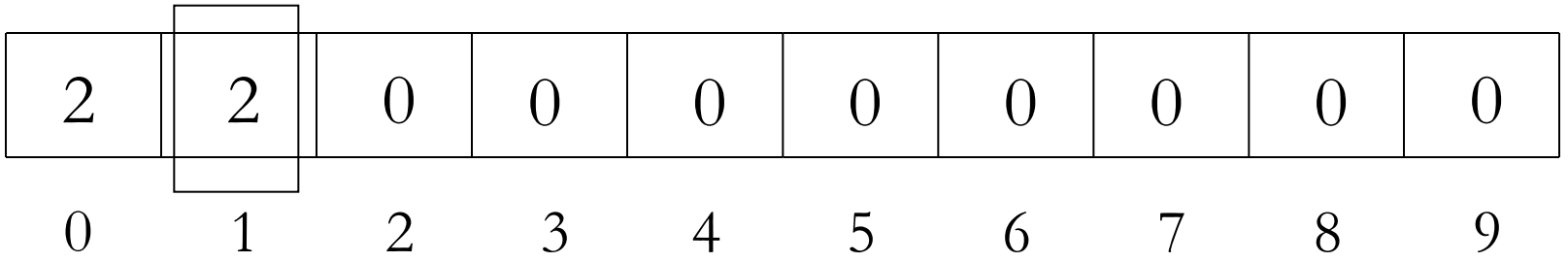
Busca Simples pela Moda

Todas as posições do vetor são comparadas com os elementos seguintes do vetor.

valores



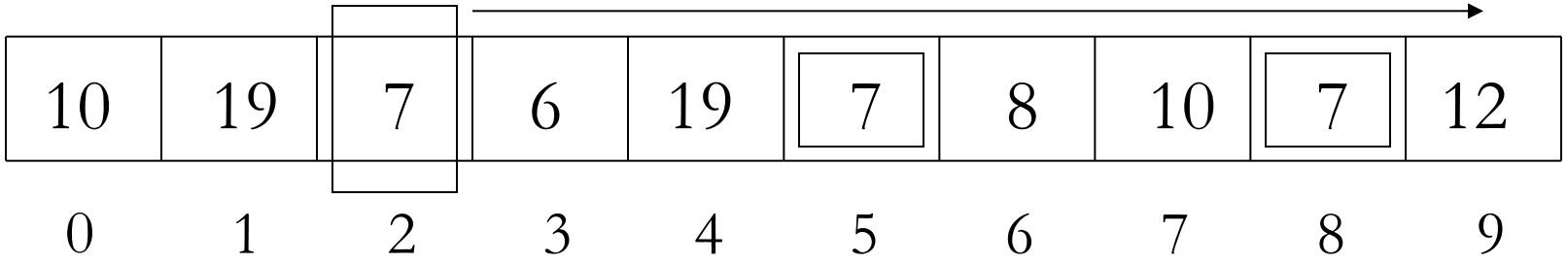
auxiliar (contém a frequência de cada elemento do valores)



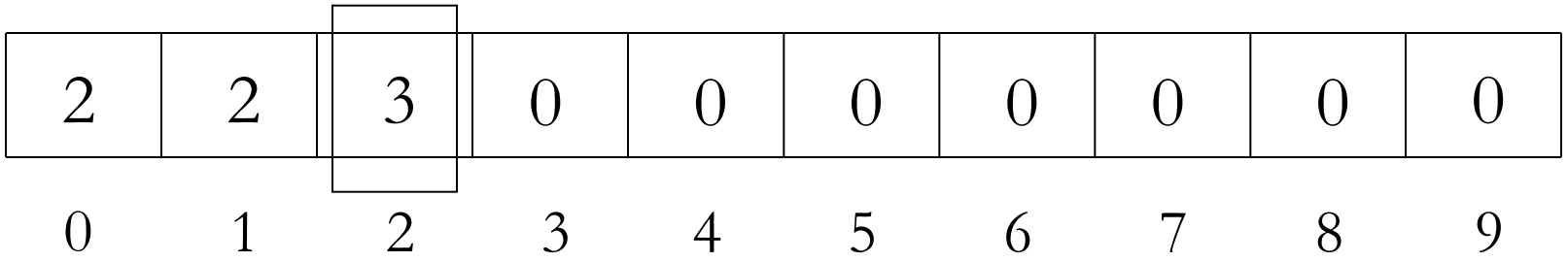
Busca Simples pela Moda

Todas as posições do vetor são comparadas com os elementos seguintes do vetor.

valores



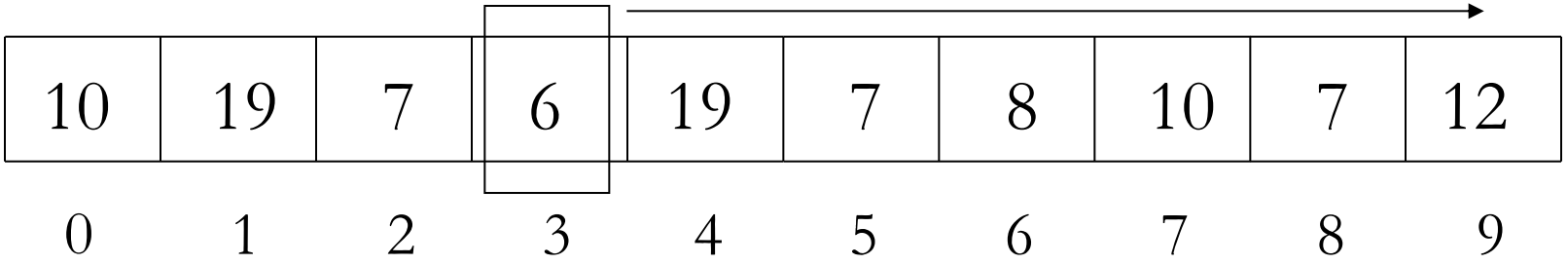
auxiliar (contém a frequência de cada elemento do valores)



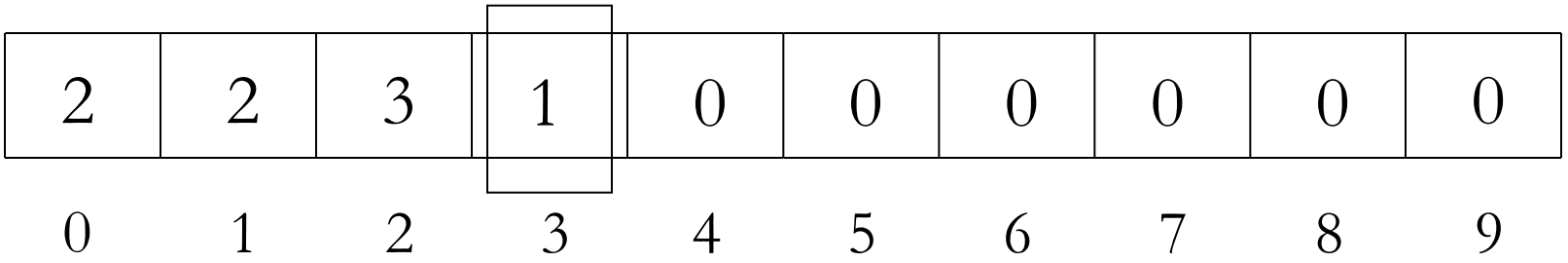
Busca Simples pela Moda

Todas as posições do vetor são comparadas com os elementos seguintes do vetor.

valores



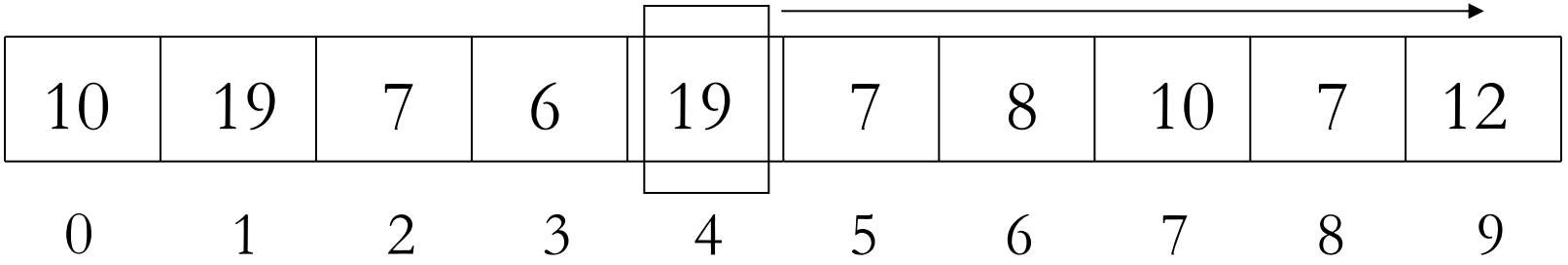
auxiliar (contém a frequência de cada elemento do valores)



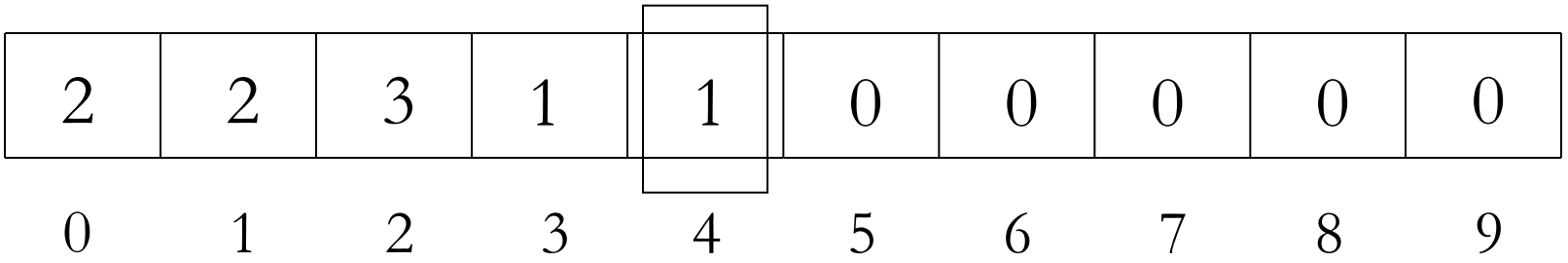
Busca Simples pela Moda

Todas as posições do vetor são comparadas com os elementos seguintes do vetor.

valores



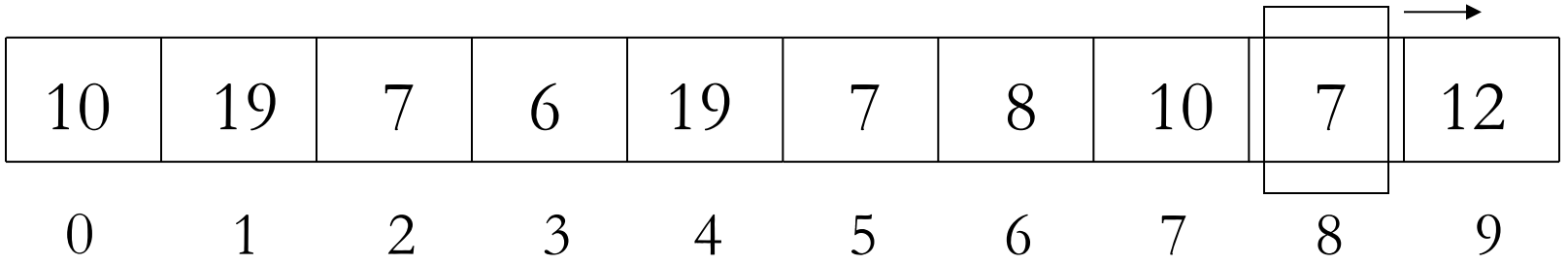
auxiliar (contém a frequência de cada elemento do valores)



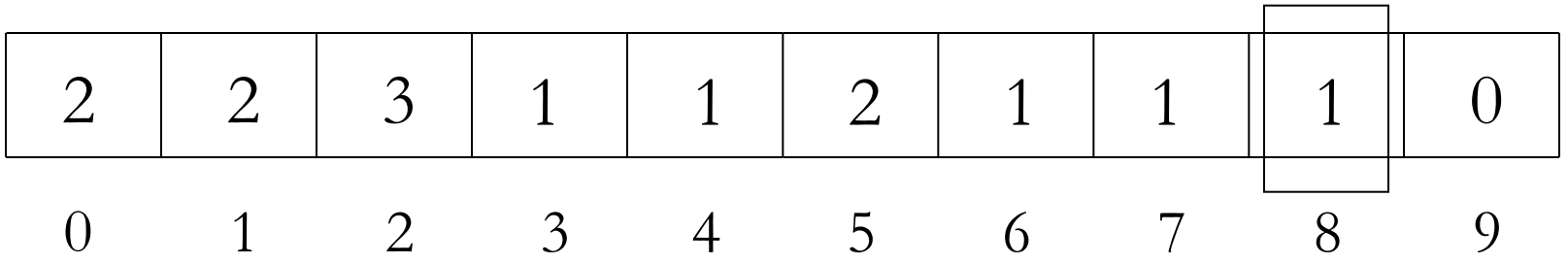
Busca Simples pela Moda

Todas as posições do vetor são comparadas com os elementos seguintes do vetor.

valores



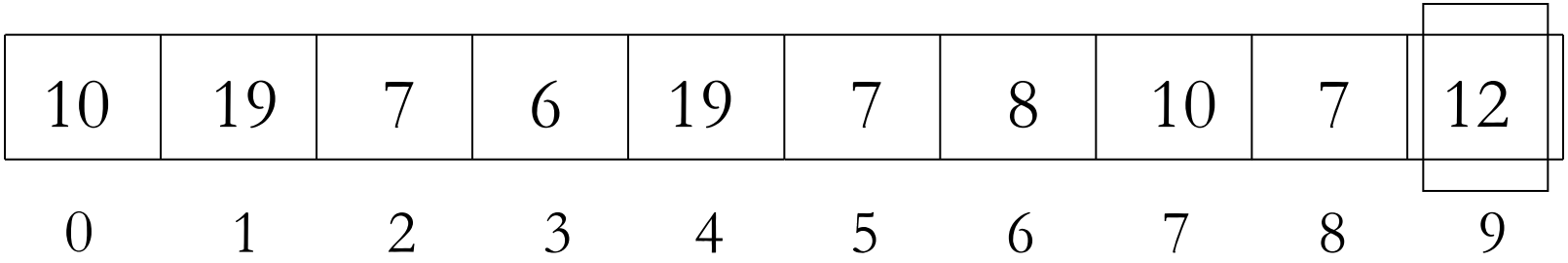
auxiliar



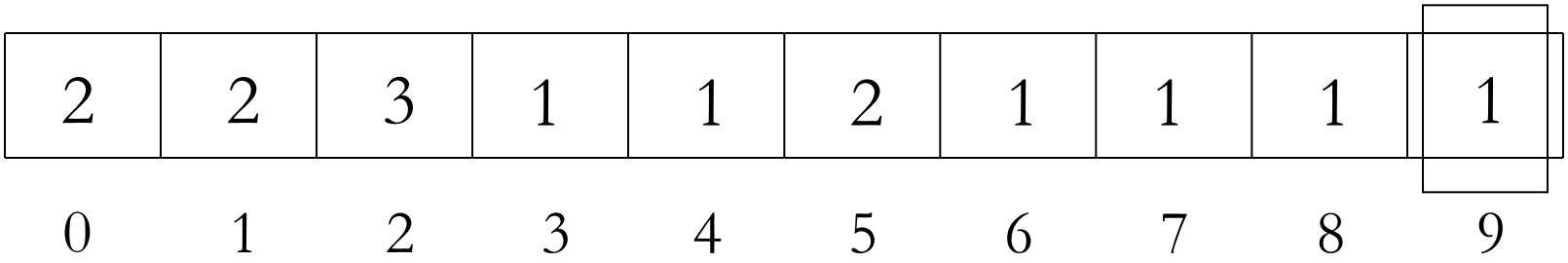
Busca Simples pela Moda

Todas as posições do vetor são comparadas com os elementos seguintes do vetor.

valores

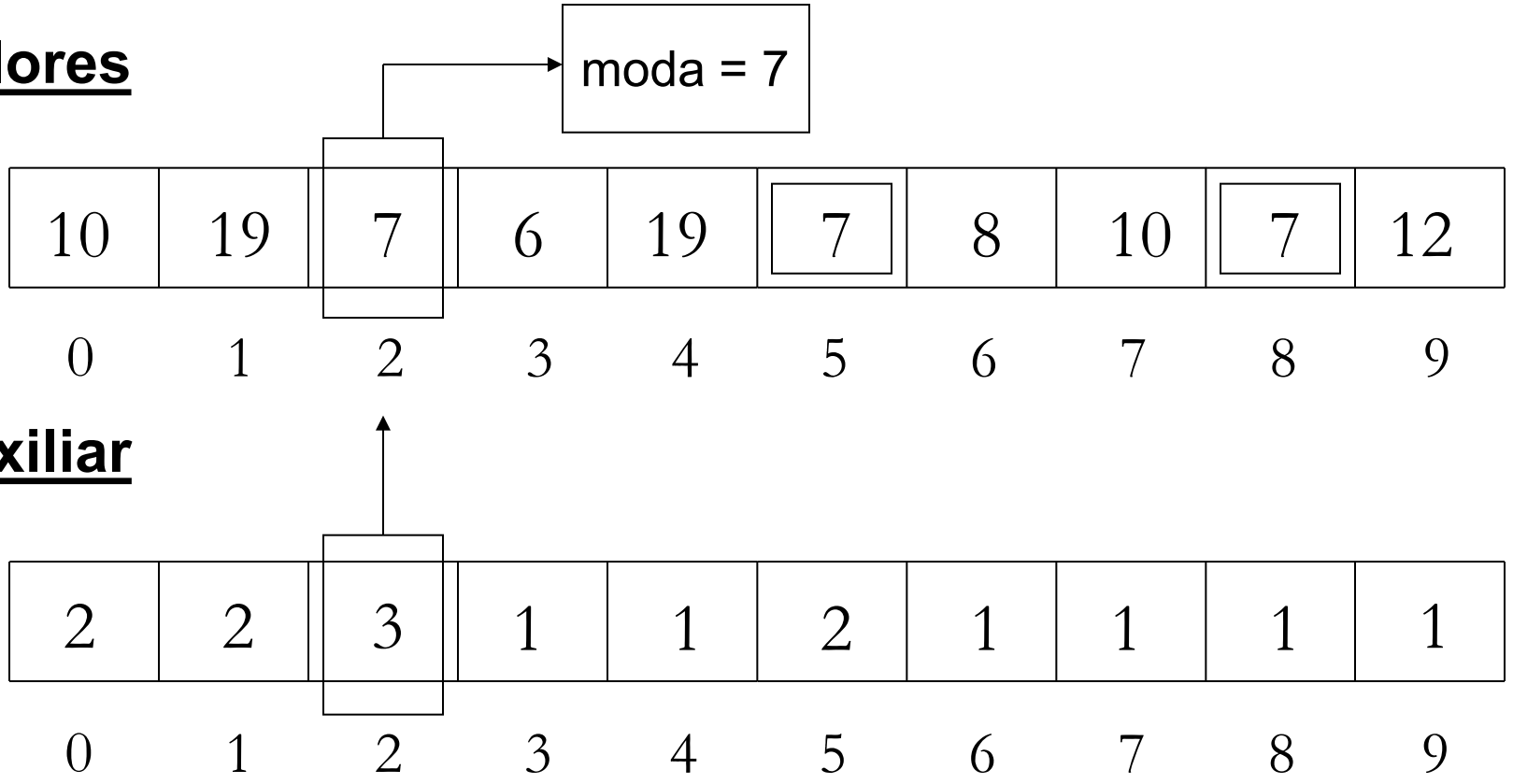


auxiliar



Busca Simples pela Moda

Todas as posições do vetor são comparadas com os elementos seguintes do vetor.



```
# Subprogramas
def preenche(valores):
    for ind in range(len(valores)):
        valores[ind] = int(input("Elemento["+str(ind)+"]="))
    return None
def buscaModa(valores):
    auxiliar = [0]*len(valores)
    for indice in range(len(valores)):          # calcula as frequências
        auxiliar[indice] = 1
        for varre in range(indice+1, len(valores)):
            if valores[varre]==valores[indice]:
                auxiliar[indice] += 1
    ondeModa = 0
    for i in range(1, len(auxiliar)):           # localiza a maior frequência
        if auxiliar[i] > auxiliar[ondeModa]:
            ondeModa = i
    return valores[ondeModa]                    # retorna o valor da moda
# Programa Principal de Busca o Elemento da Moda
numeros = [0]*10
preenche(numeros)
moda = buscaModa(numeros)
print("A moda do vetor  elemento da é:", moda)
```

```
# Subprogramas
def preenche(valores):
    for ind in range(len(valores)):
        valores[ind] = int(input("Elemento["+str(ind)+"]="))
    return None
def buscaModa(valores):
    auxiliar = [0]*len(valores)
    for indice in range(len(valores)):          # calcula as frequências
        auxiliar[indice] = 1
        for varre in range(indice+1, len(valores)):
            if valores[varre]==valores[indice]:
                auxiliar[indice] += 1
    ondeModa = 0
    for i in range(1, len(auxiliar)):           # localiza a maior frequência
        if auxiliar[i] > auxiliar[ondeModa]:
            ondeModa = i
    return valores[ondeModa]                    # retorna o valor da moda
# Programa Principal de Busca o Elemento da Moda
numeros = [0]*10
preenche(numeros)
moda = buscaModa(numeros)
print("A moda do vetor  elemento da é:", moda)
```

```
# Subprogramas
def preenche(valores):
    for ind in range(len(valores)):
        valores[ind] = int(input("Elemento["+str(ind)+"]="))
    return None
def buscaModa(valores):
    auxiliar = [0]*len(valores)
    for indice in range(len(valores)):          # calcula as frequências
        auxiliar[indice] = 1
        for varre in range(indice+1, len(valores)):
            if valores[varre]==valores[indice]:
                auxiliar[indice] += 1
    ondeModa = 0
    for i in range(1, len(auxiliar)):           # localiza a maior frequência
        if auxiliar[i] > auxiliar[ondeModa]:
            ondeModa = i
    return valores[ondeModa]                    # retorna o valor da moda
# Programa Principal de Busca o Elemento da Moda
numeros = [0]*10
preenche(numeros)
moda = buscaModa(numeros)
print("A moda do vetor  elemento da é:", moda)
```

Busca Simples pela Moda

- No algoritmo anterior, há basicamente duas estruturas de controle **for** em sequência.
  - Neste caso, o primeiro **for**, que possui o maior custo computacional, determina a complexidade do algoritmo.

Busca Simples pela Moda

- No algoritmo anterior, há basicamente duas estruturas de controle **for** em sequência.
  - Neste caso, o primeiro **for**, que possui o maior custo computacional, determina a complexidade do algoritmo.

- O primeiro **for** executa, para cada elemento do vetor, comparações com os elementos seguintes.
  - Desta forma, aproximadamente **n** (**n**-1)/2 elementos são acessados.
  - Ou seja, o número de elementos avaliados é da ordem de $\mathbf{n}^2$. Portanto sua complexidade é $O(\mathbf{n}^2)$.

Busca pela Moda de um Vetor Ordenado

Neste caso, considera-se que os elementos encontra-se ordenados de forma crescente no vetor.

valores (ordenado)

6	7	7	7	8	10	10	12	19	19
0	1	2	3	4	5	6	7	8	9

Moda deste vetor: elemento 7 (que ocorre 3 vezes).

Busca pela Moda de um Vetor Ordenado

```
def buscaModa(valores):
    moda = valores[0]
    ind = 0
    frequencia = 1
    while ind < len(valores) - 1:
        ind = ind + 1
        if valores[ind] == valores[ind-frequencia]:
            moda = valores[ind]
            frequencia = frequencia + 1
    return moda
```

No algoritmo acima, o vetor é percorrido apenas uma vez. Desta forma, o número de elementos avaliados é da ordem de **n**. Portanto a complexidade do algoritmo é O(**n**).

Algoritmos de Ordenação

- O problema da ordenação é caracterizado pela organização de um conjunto de elemento do mesmo tipo segundo um critério de ordenação.

Algoritmos de Ordenação

- O problema da ordenação é caracterizado pela organização de um conjunto de elemento do mesmo tipo segundo um critério de ordenação.

- Sem perda de generalidade, o problema será atacado considerando-se que:
  - Os elementos a serem ordenados são numéricos e estão armazenados em um vetor;
  - Deseja-se ordenar os elementos não decrescentemente:

se i < j, então valores[i] <= valores[j].

Algoritmos de Ordenação

Entrada:

Vetor **valores** que contém um conjunto de elementos.

7	19	4	10	18	6	8	1	3	12
0	1	2	3	4	5	6	7	8	9

Saída:

Vetor **valores** com o conjunto de elementos ordenados.

1	3	4	6	7	8	10	12	18	19
0	1	2	3	4	5	6	7	8	9

Algoritmos de Ordenação

```
# Subprogramas

def preenche(valores):
    for ind in range(len(valores)):
        valores[ind] = int(input("Elemento["+str(ind)+"]="))
    return None

def ordena(valores):
    ...
    return None

# Programa Principal para Ordenar Vetor

numeros = [0]*10
preenche(numeros)
print("Vetor Lido:", numeros)
ordena(numeros)
print("Vetor Ordenado:", numeros)
```

Ordenação pelo Método da Seleção (*SelectionSort*)

Para cada posição **i** do vetor **valores**, o algoritmo procura pelo **i**-ésimo menor elemento e o coloca na posição **i**.

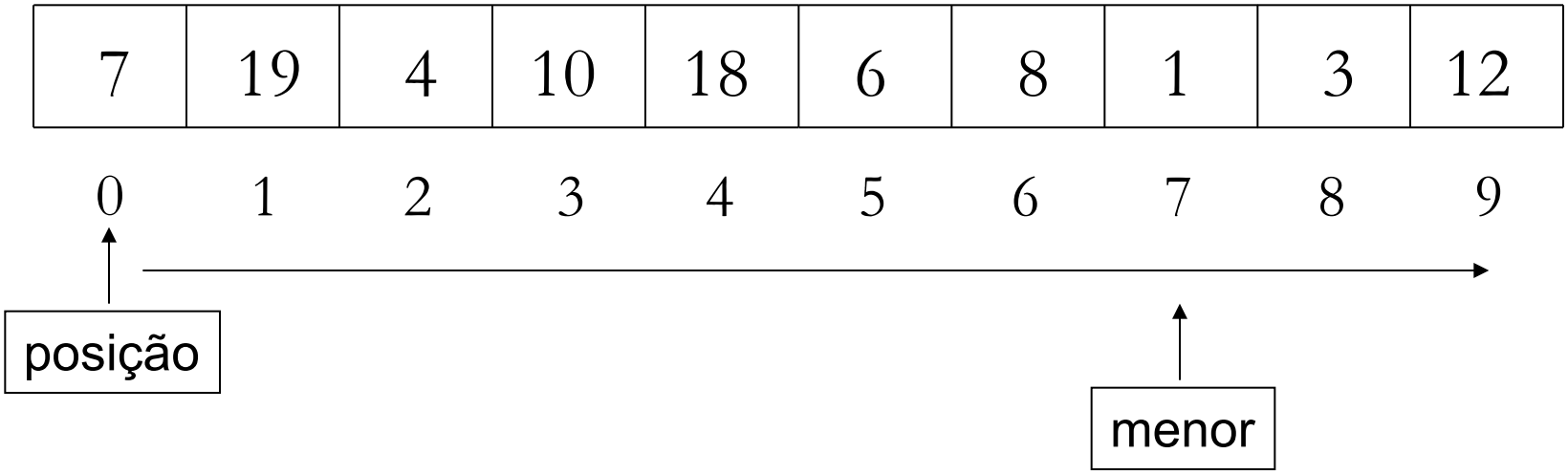
valores:

7	19	4	10	18	6	8	1	3	12
0	1	2	3	4	5	6	7	8	9

Ordenação pelo Método da Seleção (*SelectionSort*)

Para cada posição **i** do vetor **valores**, o algoritmo procura pelo **i**-ésimo menor elemento e o coloca na posição **i**.

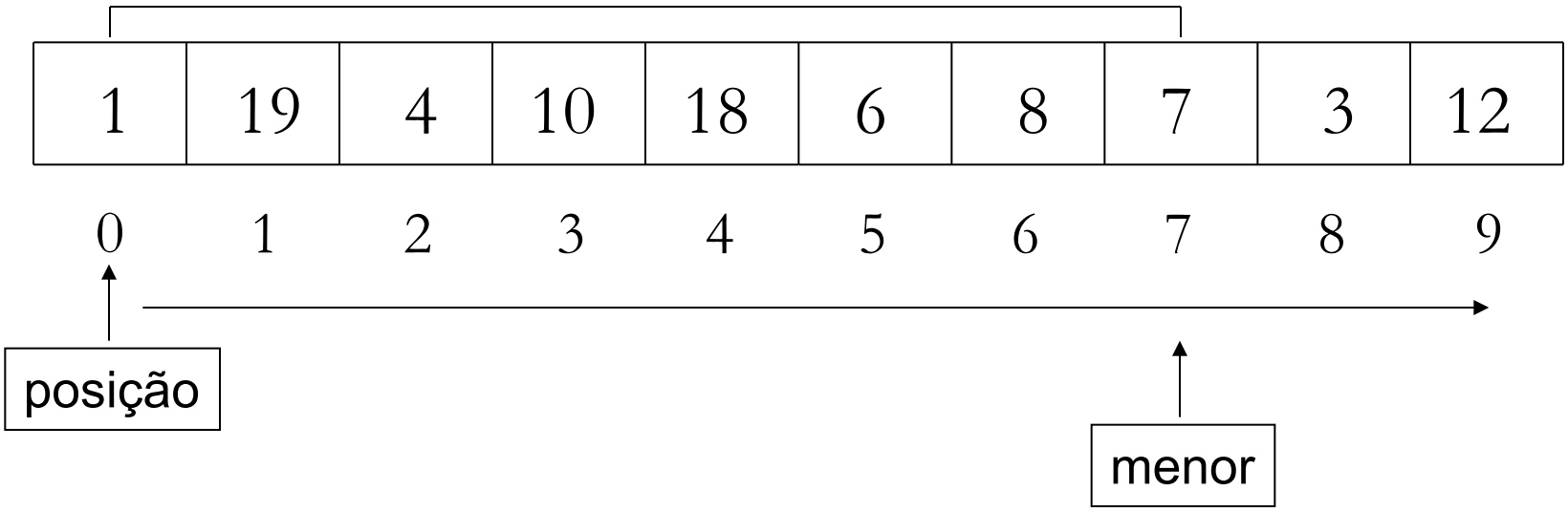
valores:



Ordenação pelo Método da Seleção (*SelectionSort*)

Para cada posição **i** do vetor **valores**, o algoritmo procura pelo **i**-ésimo menor elemento e o coloca na posição **i**.

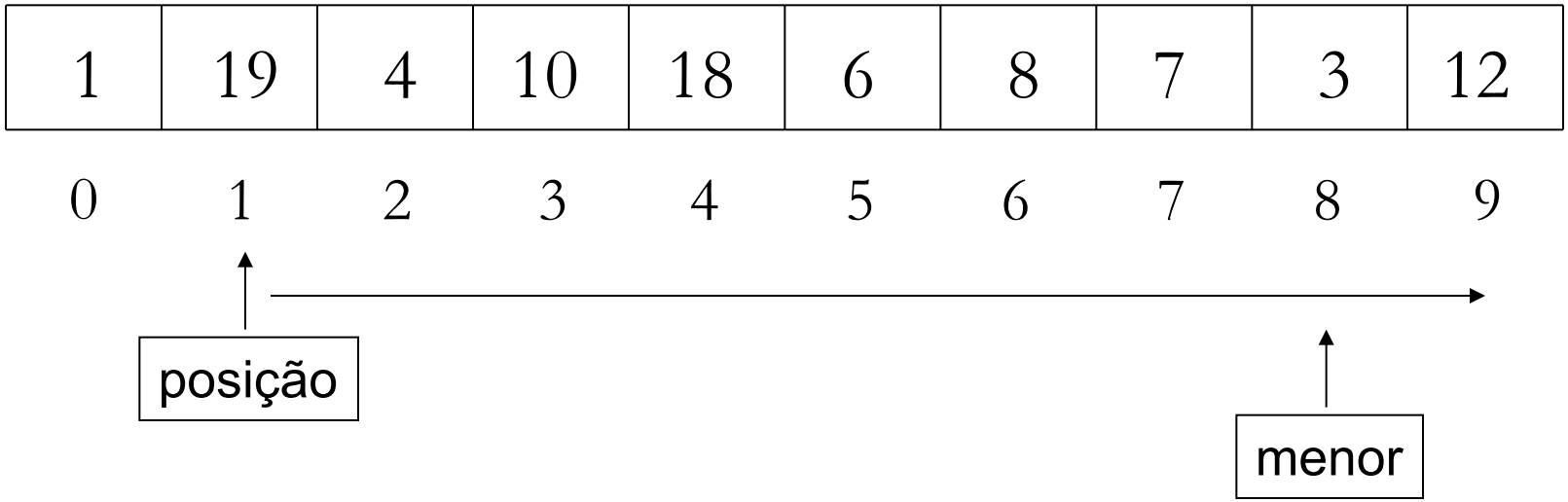
valores:



Ordenação pelo Método da Seleção (*SelectionSort*)

Para cada posição **i** do vetor **valores**, o algoritmo procura pelo **i**-ésimo menor elemento e o coloca na posição **i**.

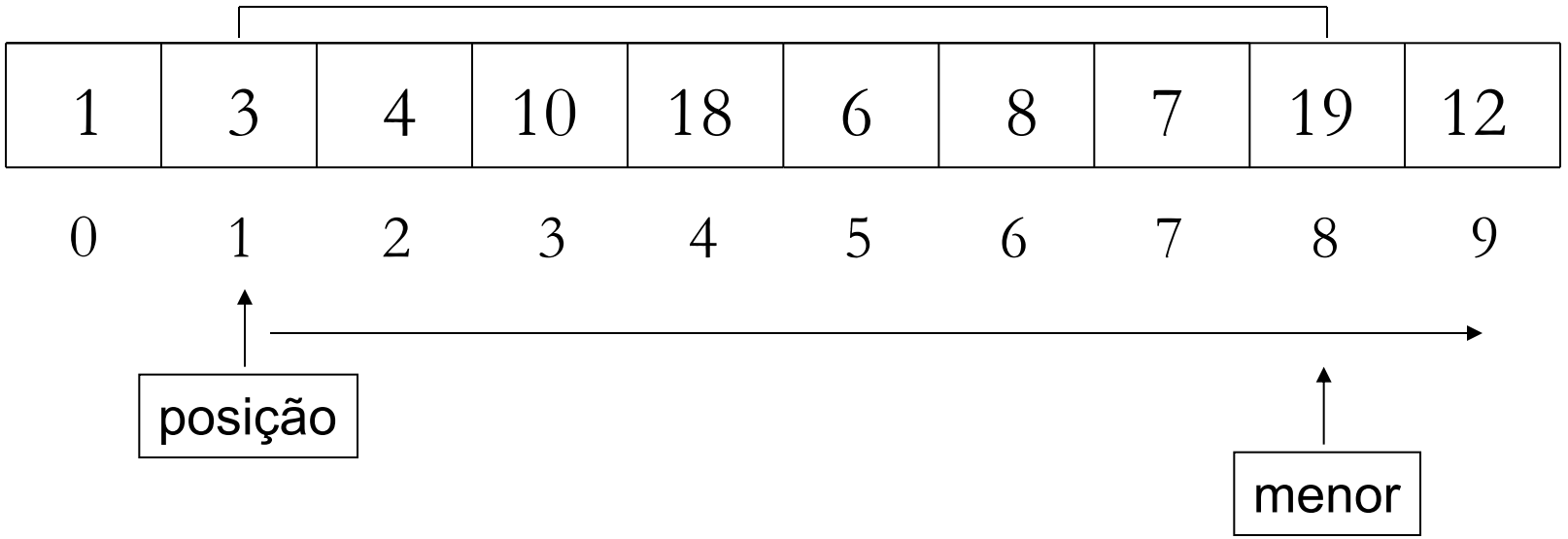
valores:



Ordenação pelo Método da Seleção (*SelectionSort*)

Para cada posição **i** do vetor **valores**, o algoritmo procura pelo **i**-ésimo menor elemento e o coloca na posição **i**.

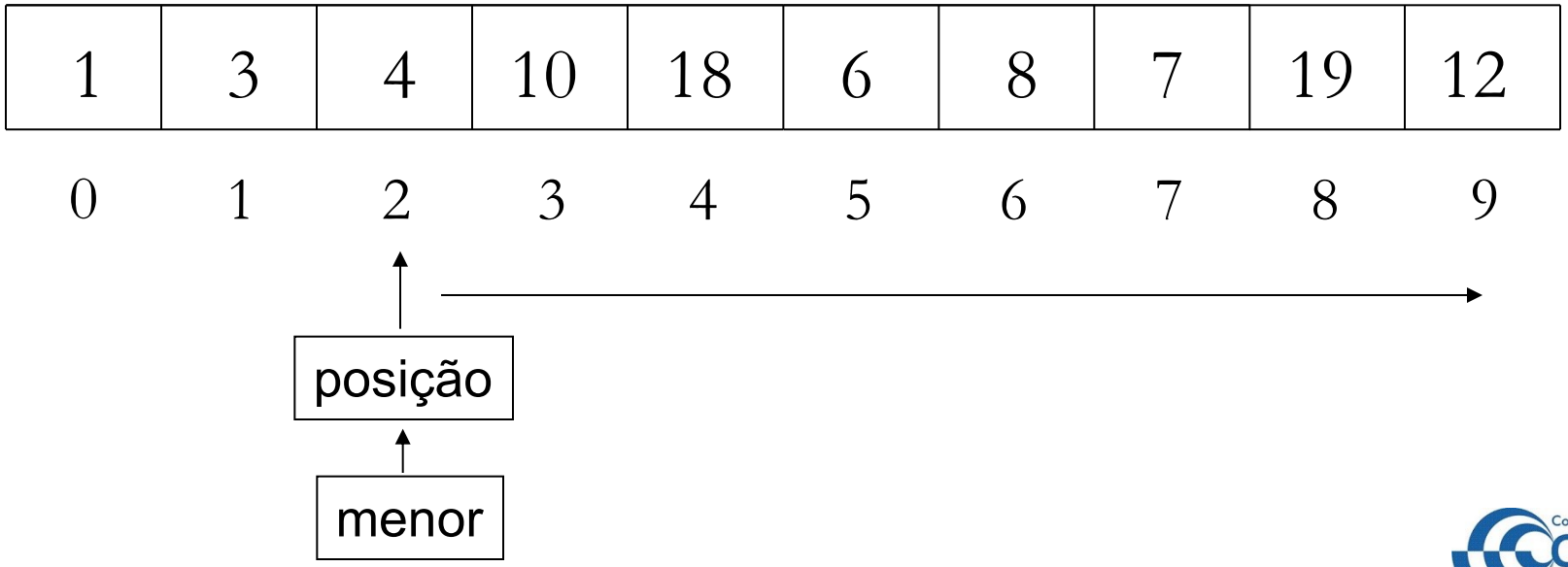
valores:



Ordenação pelo Método da Seleção (*SelectionSort*)

Para cada posição **i** do vetor **valores**, o algoritmo procura pelo **i**-ésimo menor elemento e o coloca na posição **i**.

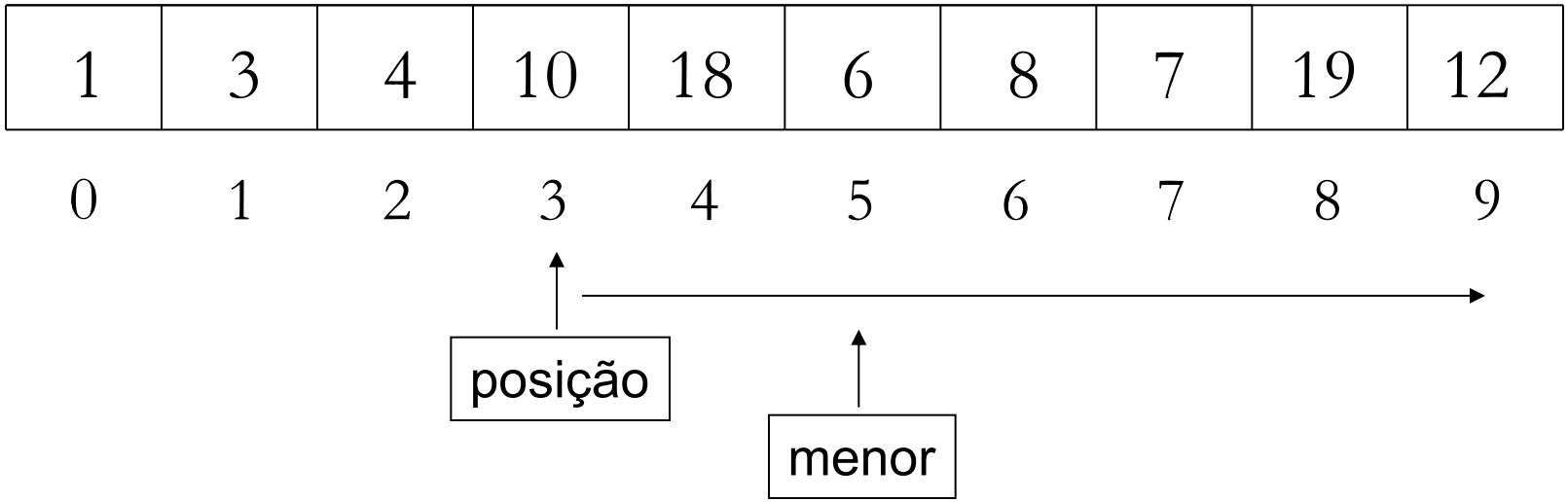
valores:



Ordenação pelo Método da Seleção (*SelectionSort*)

Para cada posição **i** do vetor **valores**, o algoritmo procura pelo **i**-ésimo menor elemento e o coloca na posição **i**.

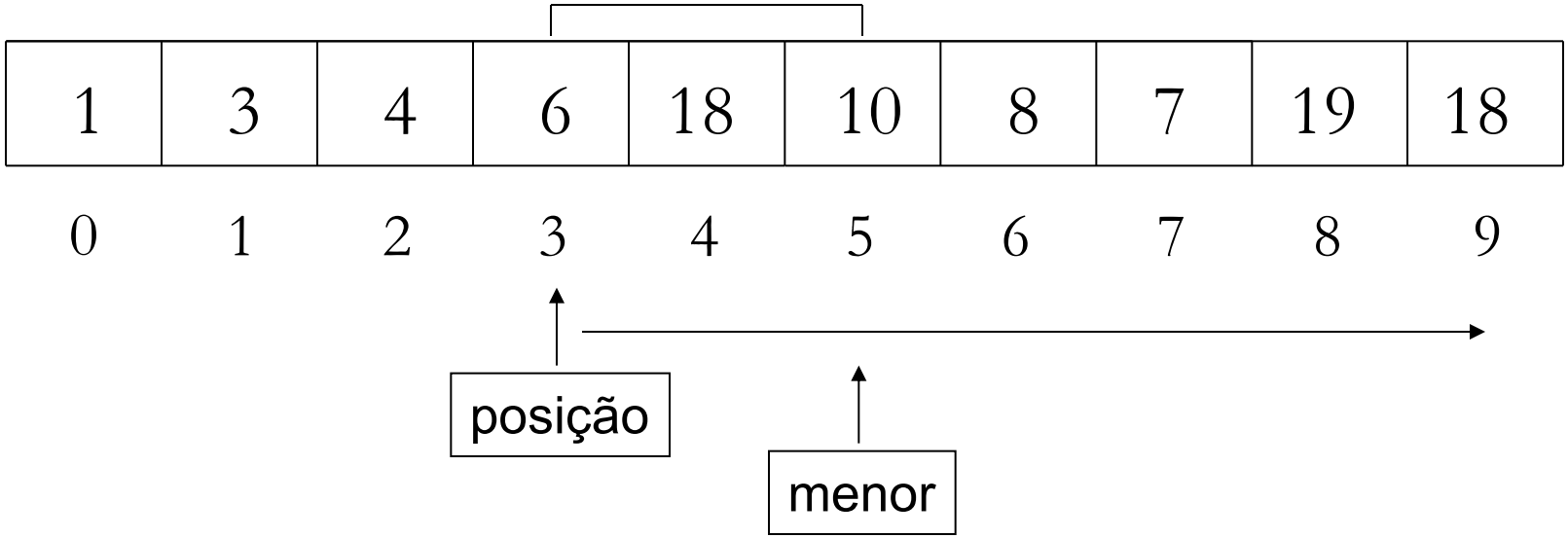
valores:



Ordenação pelo Método da Seleção (*SelectionSort*)

Para cada posição **i** do vetor **valores**, o algoritmo procura pelo **i**-ésimo menor elemento e o coloca na posição **i**.

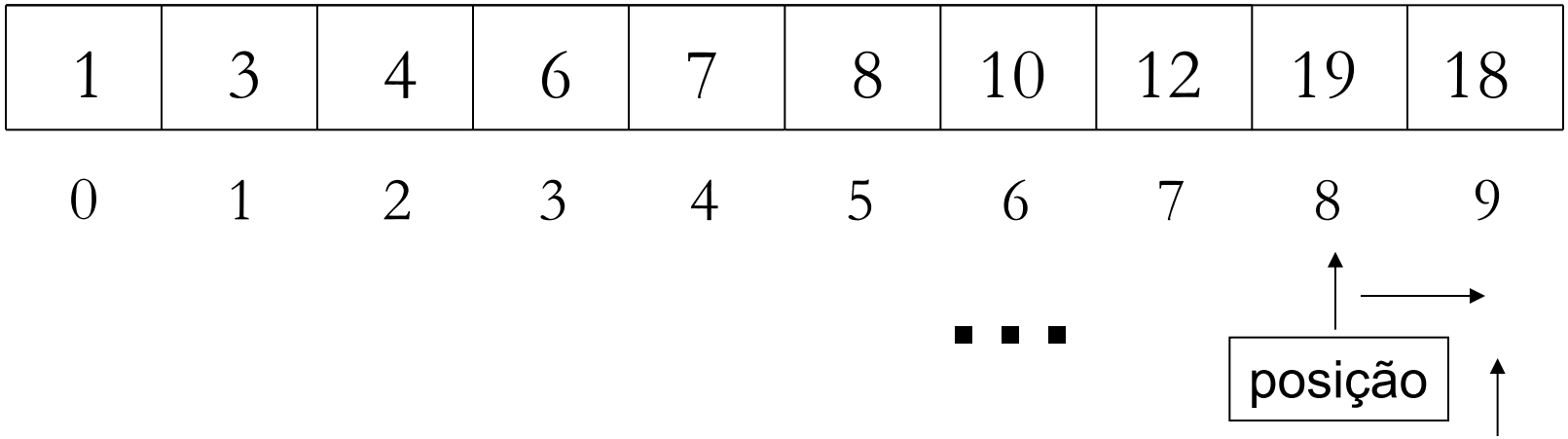
valores:



Ordenação pelo Método da Seleção (*SelectionSort*)

Para cada posição **i** do vetor **valores**, o algoritmo procura pelo **i**-ésimo menor elemento e o coloca na posição **i**.

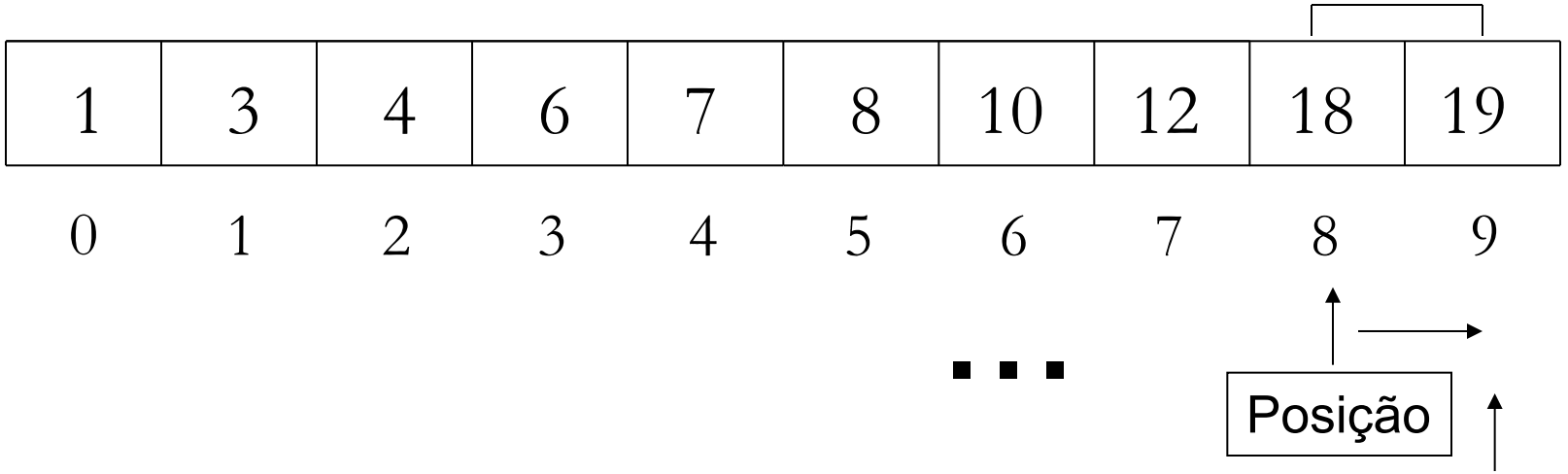
valores:



Ordenação pelo Método da Seleção (*SelectionSort*)

Para cada posição **i** do vetor **valores**, o algoritmo procura pelo **i**-ésimo menor elemento e o coloca na posição **i**.

valores:



```
# Operação que troca o conteúdo de duas células do vetor.
def trocar(vals, posX, posY):
    temp = vals[posX]
    vals[posX] = vals[posY]
    vals[posY] = temp
    return None
# Operação que encontra e retorna o local do menor elemento do vetor,
# considerando as células a partir de um dado início.
def selecionarMenor(vals, inicio):
    localMenor = inicio
    for pos in range(inicio+1, len(vals)):
        if vals[pos]<vals[localMenor]:
            localMenor = pos
    return localMenor

# Método da Seleção
def ordenar(valores):
    for ind in range(len(valores)-1):
        menor = selecionarMenor(valores, ind)
        trocar(valores, ind, menor)
    return None
```

```
# Operação que troca o conteúdo de duas células do vetor.
def trocar(vals, posX, posY):
    temp = vals[posX]
    vals[posX] = vals[posY]
    vals[posY] = temp
    return None
# Operação que encontra e retorna o local do menor elemento do vetor,
# considerando as células a partir de um dado início.
def selecionarMenor(vals, inicio):
    localMenor = inicio
    for pos in range(inicio+1, len(vals)):
        if vals[pos]<vals[localMenor]:
            localMenor = pos
    return localMenor

# Método da Seleção
def ordenar(valores):
    for ind in range(len(valores)-1):
        menor = selecionarMenor(valores, ind)
        trocar(valores, ind, menor)
    return None
```

```
# Operação que troca o conteúdo de duas células do vetor.
def trocar(vals, posX, posY):
    temp = vals[posX]
    vals[posX] = vals[posY]
    vals[posY] = temp
    return None
# Operação que encontra e retorna o local do menor elemento do vetor,
# considerando as células a partir de um dado início.
def selecionarMenor(vals, inicio):
    localMenor = inicio
    for pos in range(inicio+1, len(vals)):
        if vals[pos]<vals[localMenor]:
            localMenor = pos
    return localMenor

# Método da Seleção
def ordenar(valores):
    for ind in range(len(valores)-1):
        menor = selecionarMenor(valores, ind)
        trocar(valores, ind, menor)
    return None
```

Ordenação pelo Método da Seleção (*SelectionSort*)

- No algoritmo anterior, há basicamente duas estruturas de repetição **for** aninhadas.

Ordenação pelo Método da Seleção (*SelectionSort*)

- No algoritmo anterior, há basicamente duas estruturas de repetição **for** aninhadas.

- A mais externa executa, para cada elemento do vetor, comparações com os elementos seguintes e em seguida uma troca.
  - Desta forma, aproximadamente **n** (**n**-1)/2 elementos são acessados.

Ordenação pelo Método da Seleção (*SelectionSort*)

- No algoritmo anterior, há basicamente duas estruturas de repetição **for** aninhadas.

- A mais externa executa, para cada elemento do vetor, comparações com os elementos seguintes e em seguida uma troca.
  - Desta forma, aproximadamente **n** (**n**-1)/2 elementos são acessados.

- Ou seja, o número de elementos avaliados é da ordem de $\mathbf{n}^2$.
  - Portanto sua complexidade é O($\mathbf{n}^2$).

Ordenação pelo Método da Bolha (*BubbleSort*)

- Esta estratégia executa **n**-1 iterações, controlada por uma repetição mais externa.

Ordenação pelo Método da Bolha (*BubbleSort*)

- Esta estratégia executa **n**-1 iterações, controlada por uma repetição mais externa.
- Em cada uma delas, via repetição interna,
  - percorre-se todo o vetor comparando cada par de elementos valores[i] e valores[i+1] e
  - trocando-os de posição caso valores[i] > valores[i+1].

Ordenação pelo Método da Bolha (*BubbleSort*)

- Esta estratégia executa **n**-1 iterações, controlada por uma repetição mais externa.
- Em cada uma delas, via repetição interna,
  - percorre-se todo o vetor comparando cada par de elementos valores[i] e valores[i+1] e
  - trocando-os de posição caso valores[i] > valores[i+1].

valores:

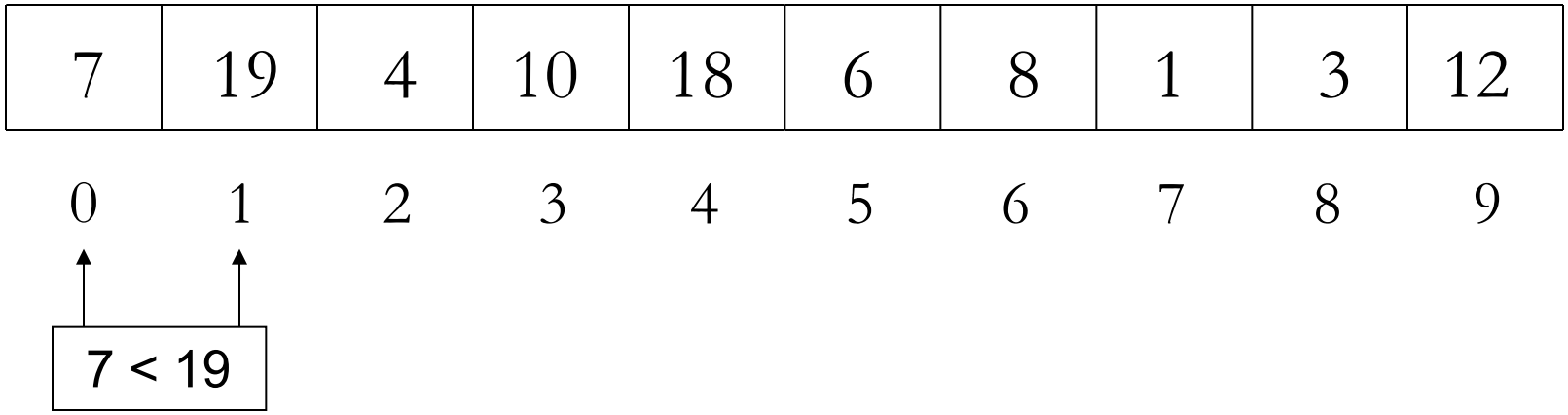
7	19	4	10	18	6	8	1	3	12
0	1	2	3	4	5	6	7	8	9

Ordenação pelo Método da Bolha (*BubbleSort*)

- Esta estratégia executa **n**-1 iterações, controlada por uma repetição mais externa.
- Em cada uma delas, via repetição interna,
  - percorre-se todo o vetor comparando cada par de elementos valores[i] e valores[i+1] e
  - trocando-os de posição caso valores[i] > valores[i+1].

valores:

Primeira Iteração Externa

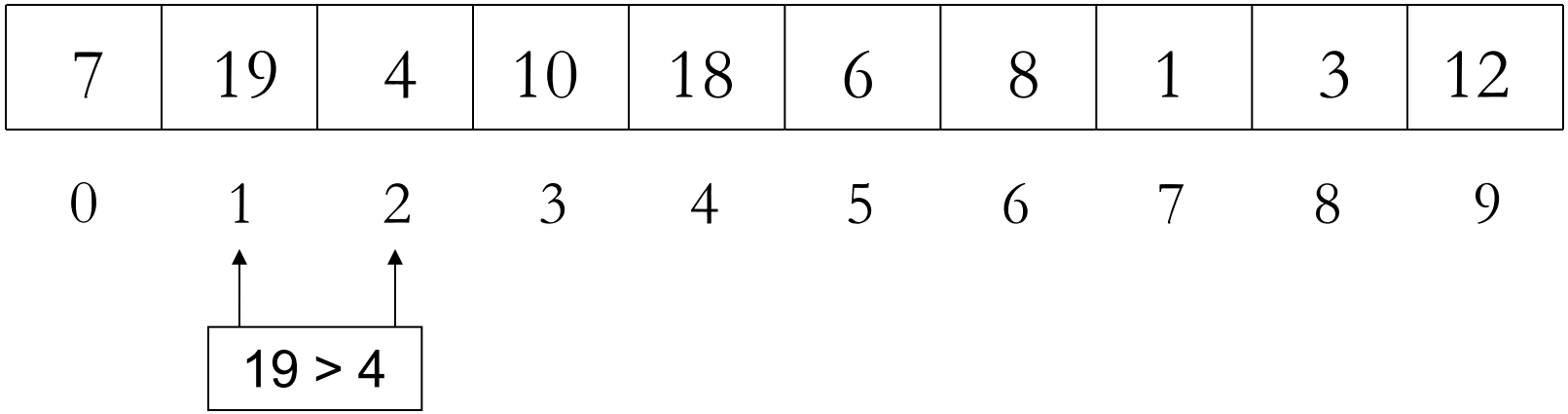


Ordenação pelo Método da Bolha (*BubbleSort*)

- Esta estratégia executa **n**-1 iterações, controlada por uma repetição mais externa.
- Em cada uma delas, via repetição interna,
  - percorre-se todo o vetor comparando cada par de elementos valores[i] e valores[i+1] e
  - trocando-os de posição caso valores[i] > valores[i+1].

valores:

Primeira Iteração Externa

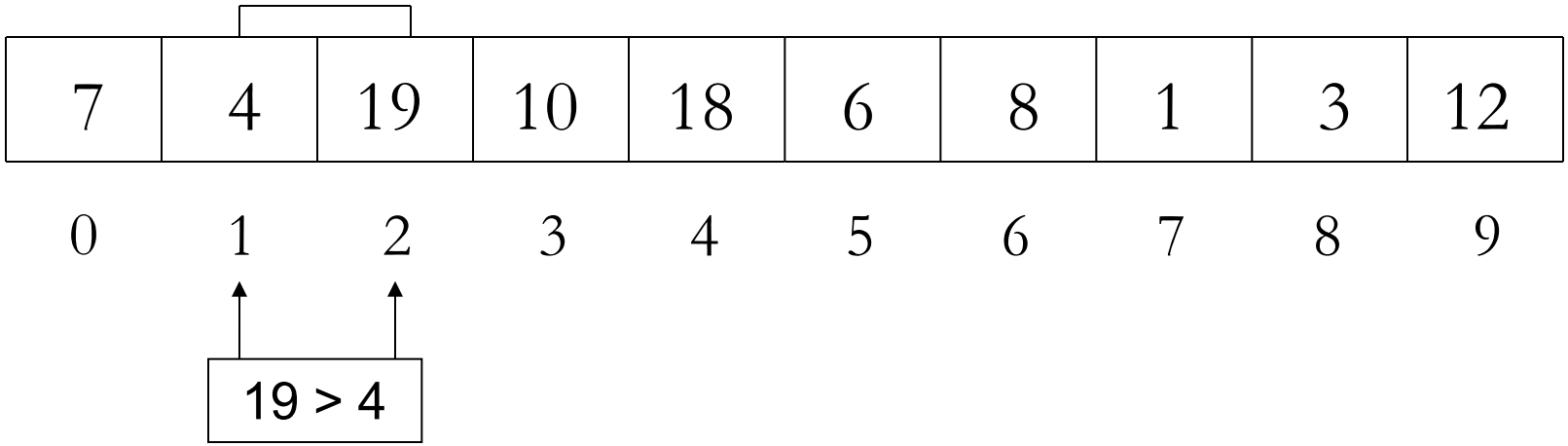


Ordenação pelo Método da Bolha (*BubbleSort*)

- Esta estratégia executa **n**-1 iterações, controlada por uma repetição mais externa.
- Em cada uma delas, via repetição interna,
  - percorre-se todo o vetor comparando cada par de elementos valores[i] e valores[i+1] e
  - trocando-os de posição caso valores[i] > valores[i+1].

valores:

Primeira Iteração Externa

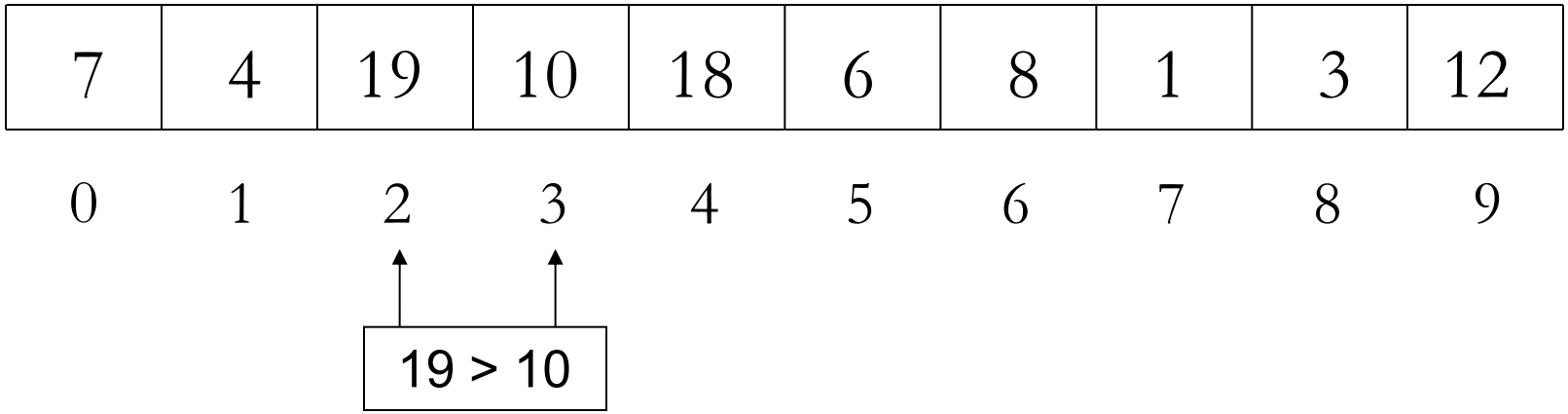


Ordenação pelo Método da Bolha (*BubbleSort*)

- Esta estratégia executa **n**-1 iterações, controlada por uma repetição mais externa.
- Em cada uma delas, via repetição interna,
  - percorre-se todo o vetor comparando cada par de elementos valores[i] e valores[i+1] e
  - trocando-os de posição caso valores[i] > valores[i+1].

valores:

Primeira Iteração Externa

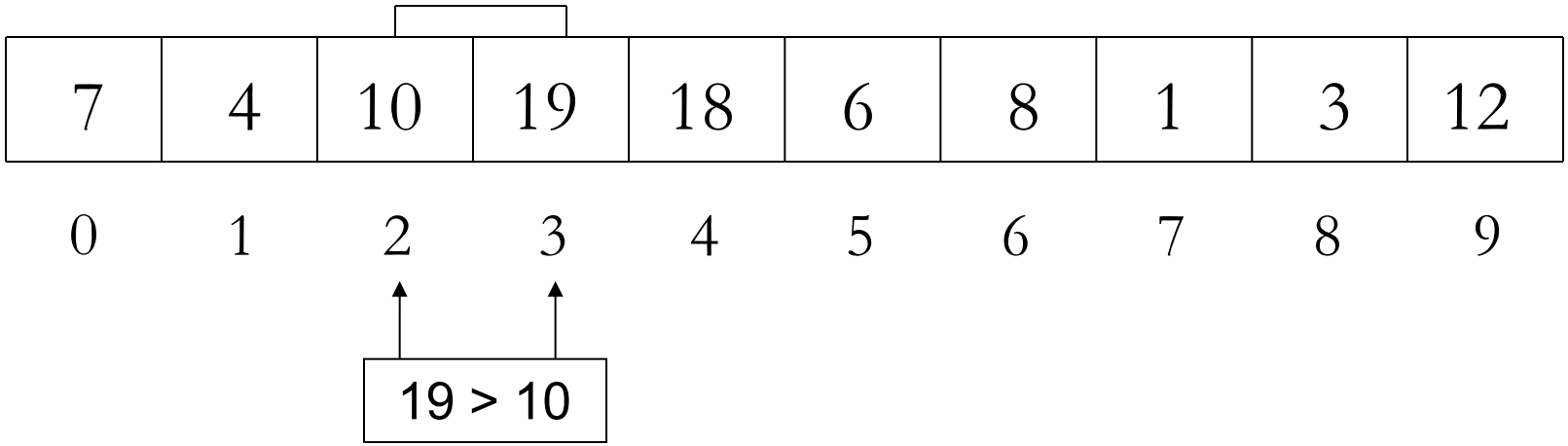


Ordenação pelo Método da Bolha (*BubbleSort*)

- Esta estratégia executa **n**-1 iterações, controlada por uma repetição mais externa.
- Em cada uma delas, via repetição interna,
  - percorre-se todo o vetor comparando cada par de elementos valores[i] e valores[i+1] e
  - trocando-os de posição caso valores[i] > valores[i+1].

valores:

Primeira Iteração Externa

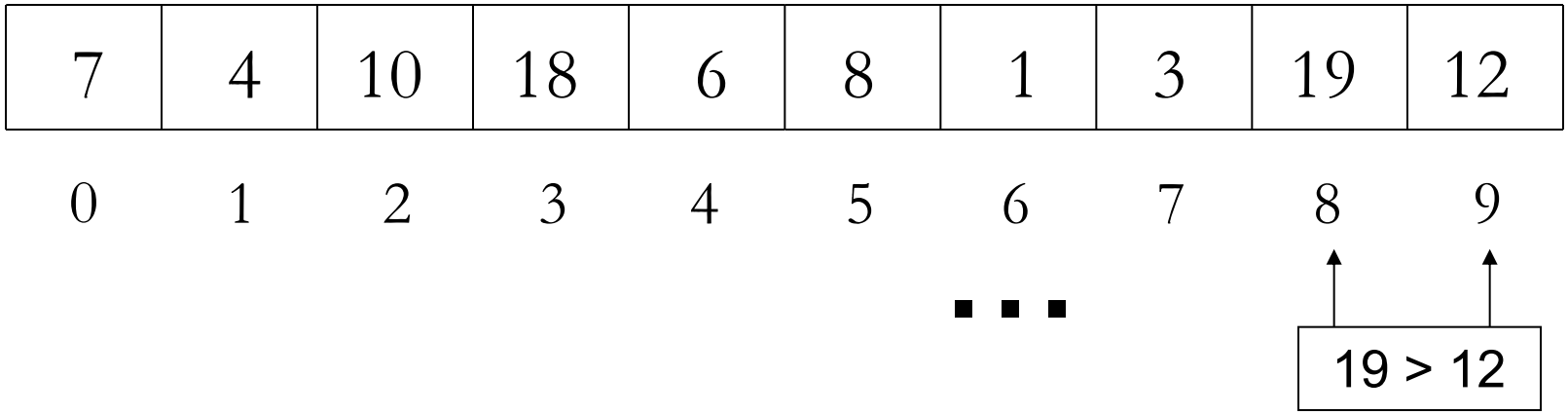


Ordenação pelo Método da Bolha (*BubbleSort*)

- Esta estratégia executa **n**-1 iterações, controlada por uma repetição mais externa.
- Em cada uma delas, via repetição interna,
  - percorre-se todo o vetor comparando cada par de elementos valores[i] e valores[i+1] e
  - trocando-os de posição caso valores[i] > valores[i+1].

valores:

Primeira Iteração Externa

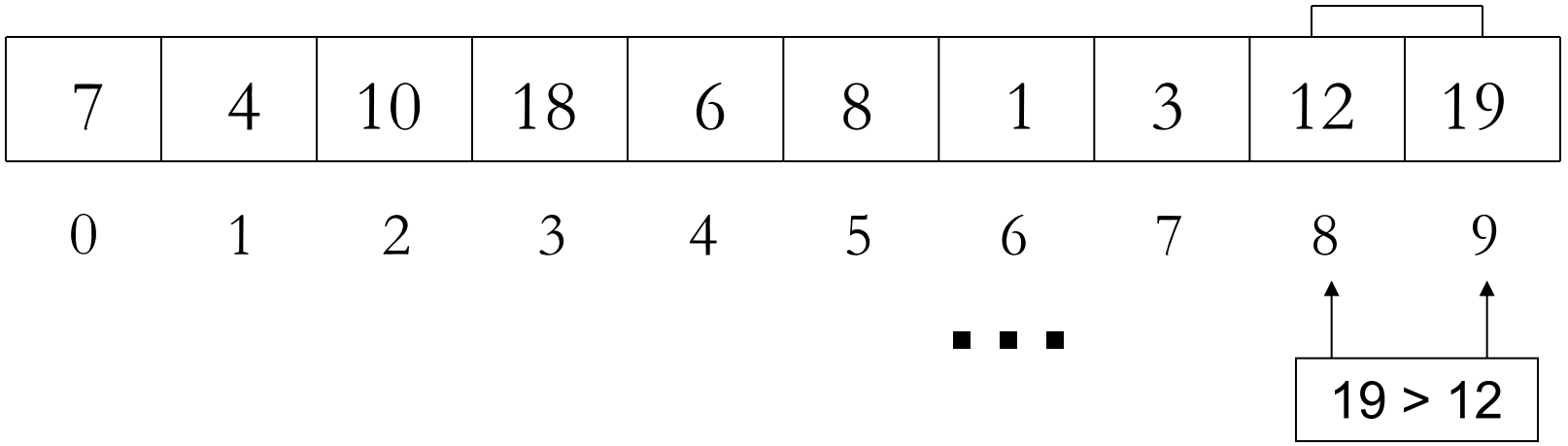


Ordenação pelo Método da Bolha (*BubbleSort*)

- Esta estratégia executa **n**-1 iterações, controlada por uma repetição mais externa.
- Em cada uma delas, via repetição interna,
  - percorre-se todo o vetor comparando cada par de elementos valores[i] e valores[i+1] e
  - trocando-os de posição caso valores[i] > valores[i+1].

valores:

Primeira Iteração Externa

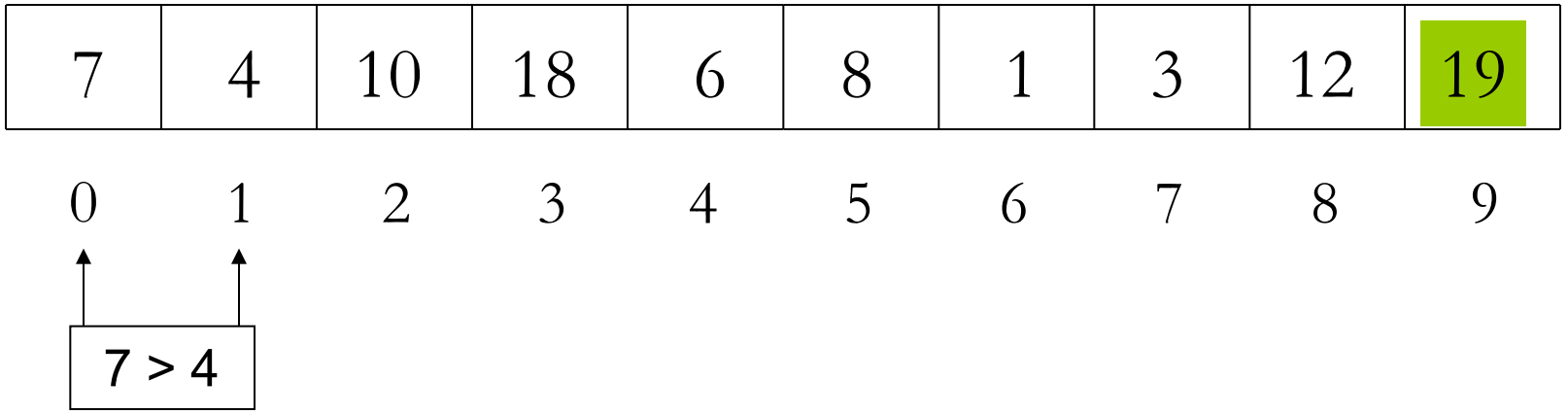


Ordenação pelo Método da Bolha (*BubbleSort*)

- Esta estratégia executa **n**-1 iterações, controlada por uma repetição mais externa.
- Em cada uma delas, via repetição interna,
  - percorre-se todo o vetor comparando cada par de elementos valores[i] e valores[i+1] e
  - trocando-os de posição caso valores[i] > valores[i+1].

valores:

Segunda Iteração Externa

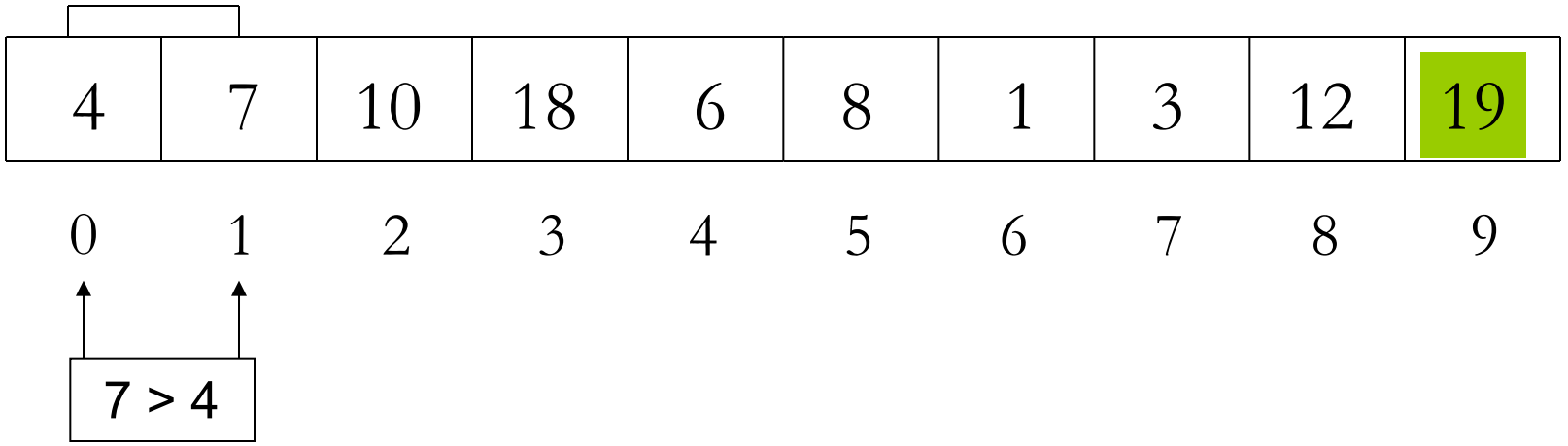


Ordenação pelo Método da Bolha (*BubbleSort*)

- Esta estratégia executa **n**-1 iterações, controlada por uma repetição mais externa.
- Em cada uma delas, via repetição interna,
  - percorre-se todo o vetor comparando cada par de elementos valores[i] e valores[i+1] e
  - trocando-os de posição caso valores[i] > valores[i+1].

valores:

Segunda Iteração Externa

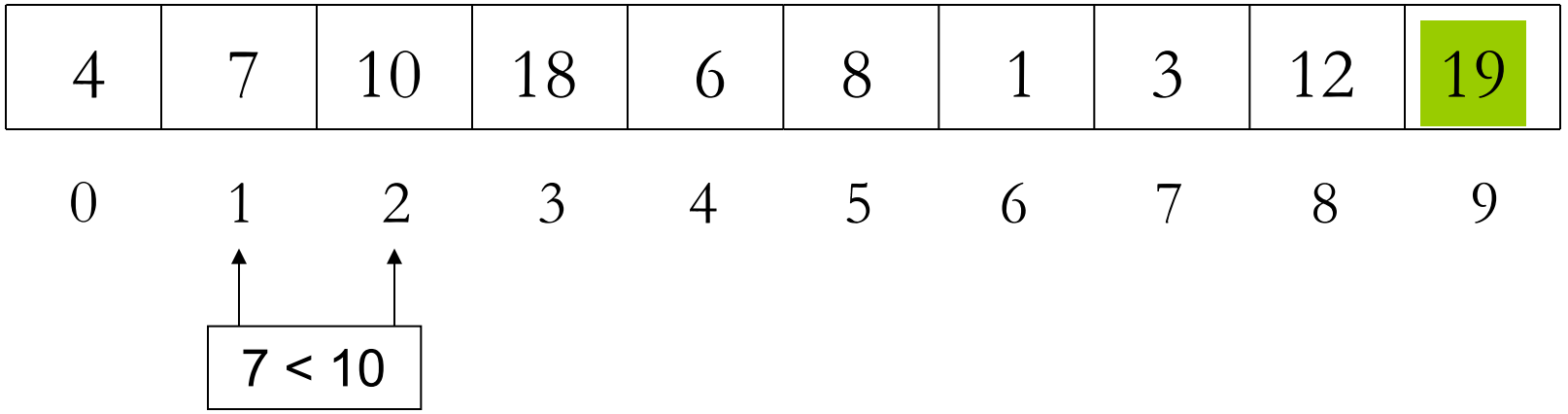


Ordenação pelo Método da Bolha (*BubbleSort*)

- Esta estratégia executa **n**-1 iterações, controlada por uma repetição mais externa.
- Em cada uma delas, via repetição interna,
  - percorre-se todo o vetor comparando cada par de elementos valores[i] e valores[i+1] e
  - trocando-os de posição caso valores[i] > valores[i+1].

valores:

Segunda Iteração Externa

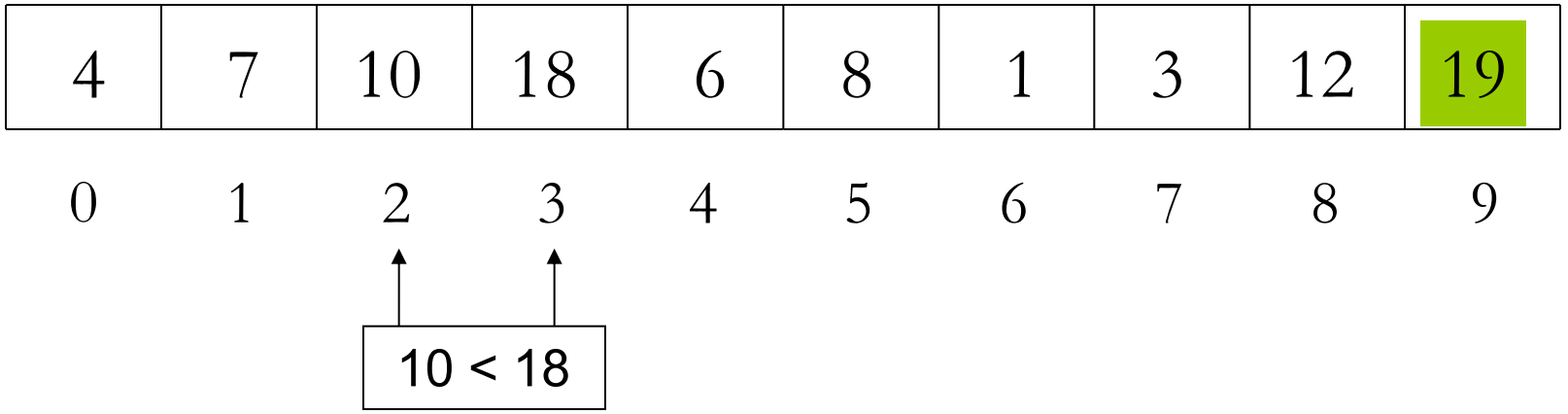


Ordenação pelo Método da Bolha (*BubbleSort*)

- Esta estratégia executa **n**-1 iterações, controlada por uma repetição mais externa.
- Em cada uma delas, via repetição interna,
  - percorre-se todo o vetor comparando cada par de elementos valores[i] e valores[i+1] e
  - trocando-os de posição caso valores[i] > valores[i+1].

valores:

Segunda Iteração Externa

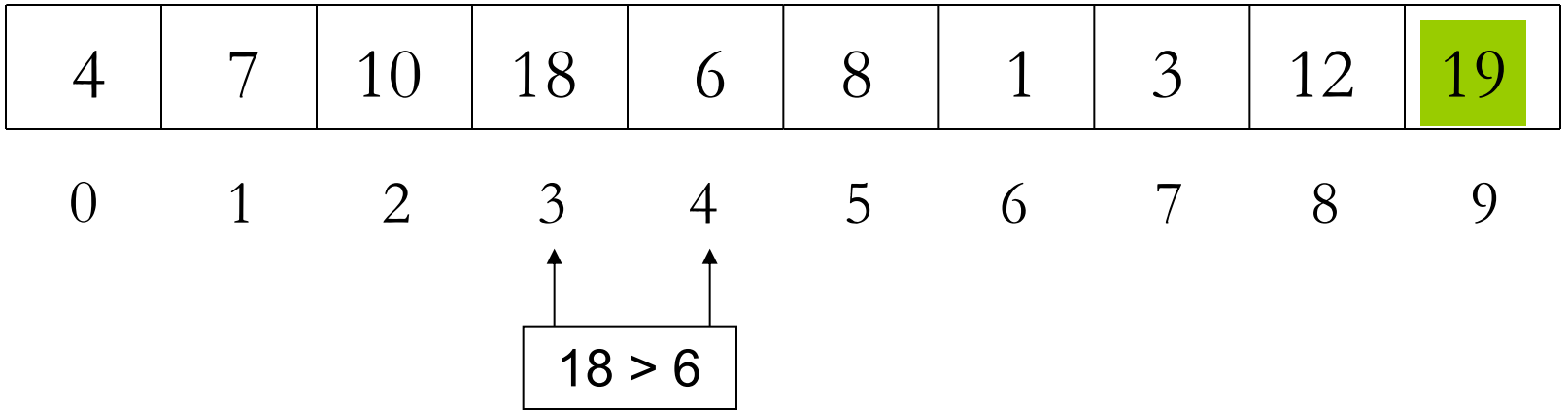


Ordenação pelo Método da Bolha (*BubbleSort*)

- Esta estratégia executa **n**-1 iterações, controlada por uma repetição mais externa.
- Em cada uma delas, via repetição interna,
  - percorre-se todo o vetor comparando cada par de elementos valores[i] e valores[i+1] e
  - trocando-os de posição caso valores[i] > valores[i+1].

valores:

Segunda Iteração Externa

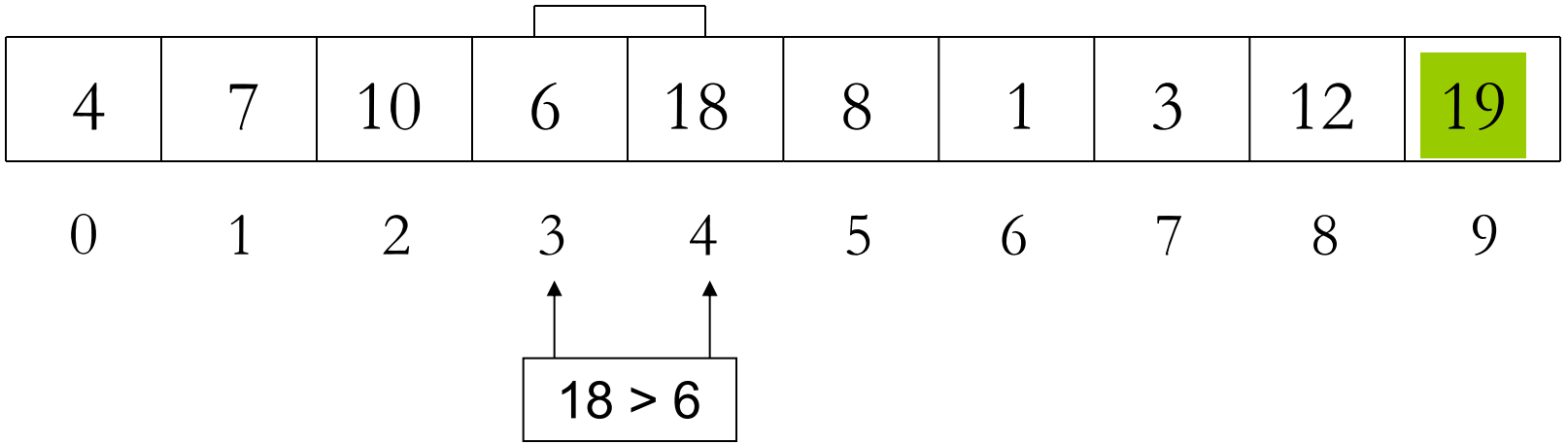


Ordenação pelo Método da Bolha (*BubbleSort*)

- Esta estratégia executa **n**-1 iterações, controlada por uma repetição mais externa.
- Em cada uma delas, via repetição interna,
  - percorre-se todo o vetor comparando cada par de elementos valores[i] e valores[i+1] e
  - trocando-os de posição caso valores[i] > valores[i+1].

valores:

Segunda Iteração Externa

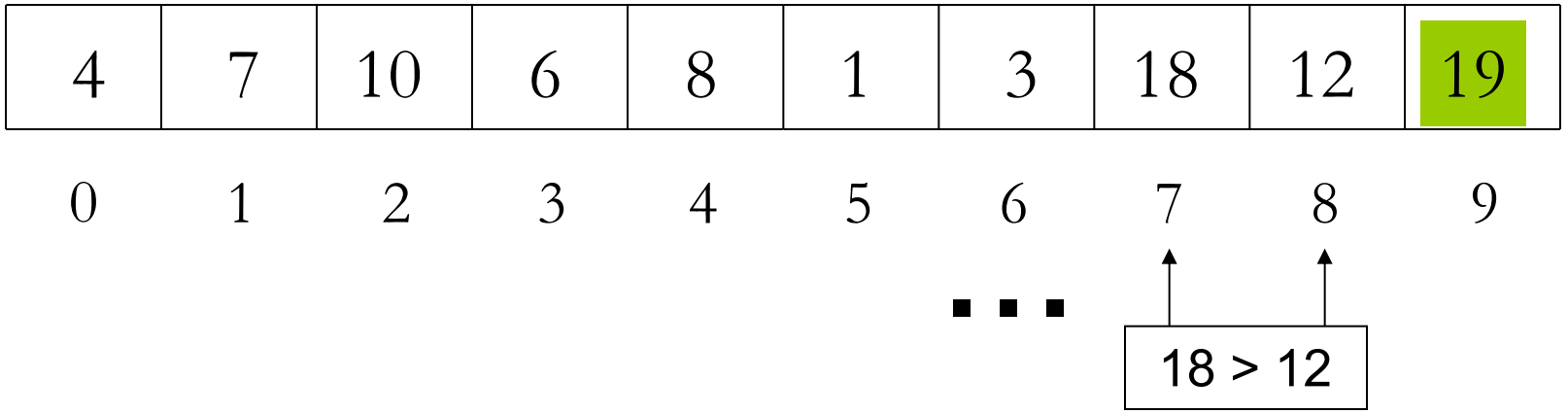


Ordenação pelo Método da Bolha (*BubbleSort*)

- Esta estratégia executa **n**-1 iterações, controlada por uma repetição mais externa.
- Em cada uma delas, via repetição interna,
  - percorre-se todo o vetor comparando cada par de elementos valores[i] e valores[i+1] e
  - trocando-os de posição caso valores[i] > valores[i+1].

valores:

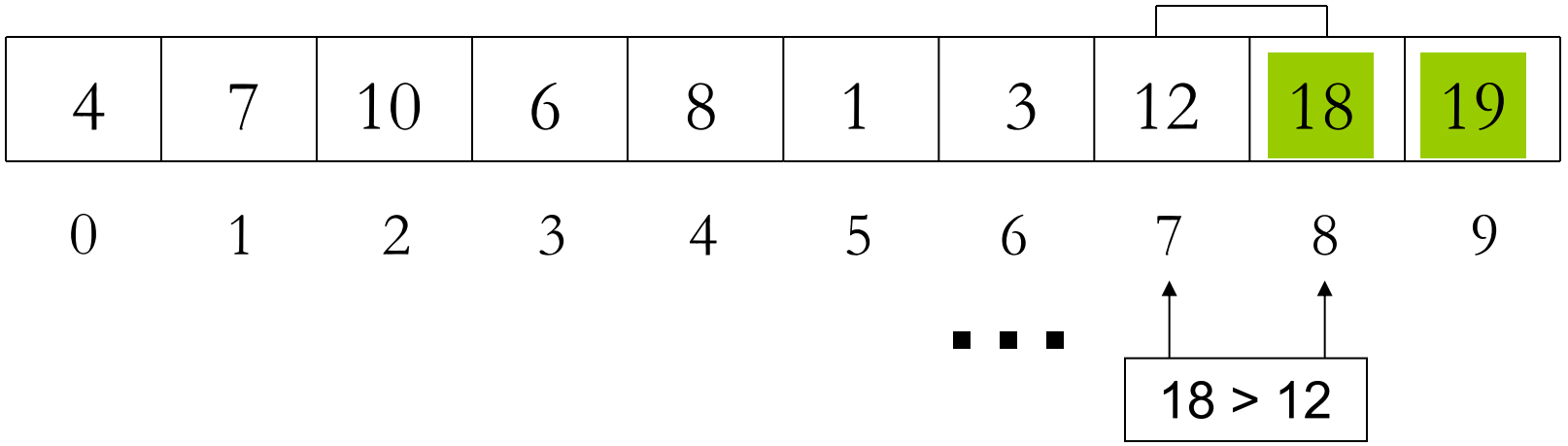
Segunda Iteração Externa



Ordenação pelo Método da Bolha (*BubbleSort*)

- Esta estratégia executa **n**-1 iterações, controlada por uma repetição mais externa.
- Em cada uma delas, via repetição interna,
  - percorre-se todo o vetor comparando cada par de elementos valores[i] e valores[i+1] e
  - trocando-os de posição caso valores[i] > valores[i+1].

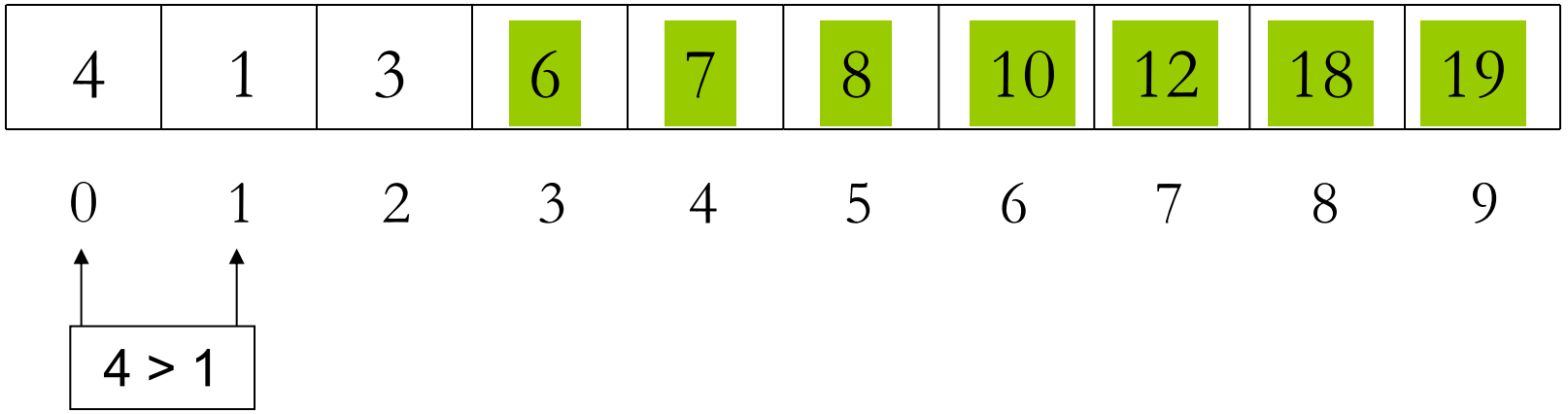
valores:



Ordenação pelo Método da Bolha (*BubbleSort*)

- Esta estratégia executa **n**-1 iterações, controlada por uma repetição mais externa.
- Em cada uma delas, via repetição interna,
  - percorre-se todo o vetor comparando cada par de elementos valores[i] e valores[i+1] e
  - trocando-os de posição caso valores[i] > valores[i+1].

valores: ▪ ▪ ▪ Oitava Iteração Externa

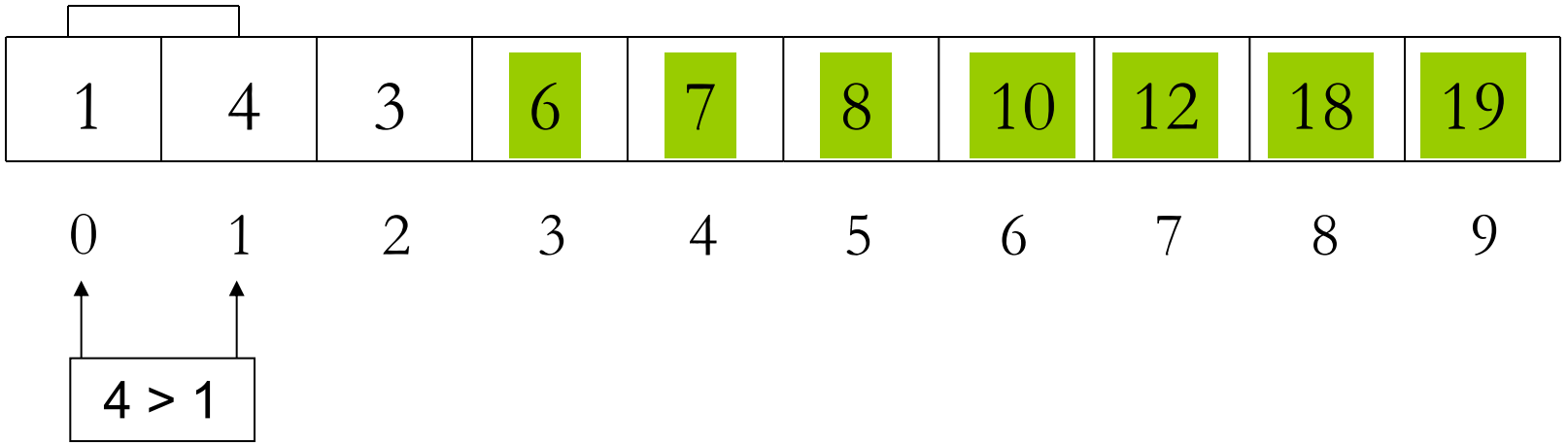


Ordenação pelo Método da Bolha (*BubbleSort*)

- Esta estratégia executa **n**-1 iterações, controlada por uma repetição mais externa.
- Em cada uma delas, via repetição interna,
  - percorre-se todo o vetor comparando cada par de elementos valores[i] e valores[i+1] e
  - trocando-os de posição caso valores[i] > valores[i+1].

valores:

▪ ▪ ▪ Oitava Iteração Externa

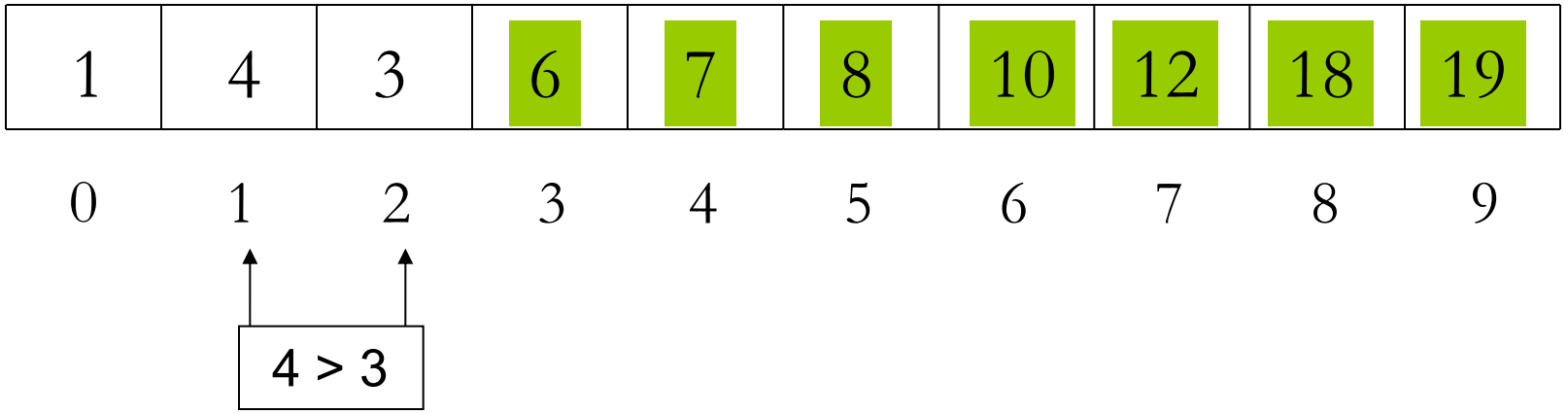


Ordenação pelo Método da Bolha (*BubbleSort*)

- Esta estratégia executa **n**-1 iterações, controlada por uma repetição mais externa.
- Em cada uma delas, via repetição interna,
  - percorre-se todo o vetor comparando cada par de elementos valores[i] e valores[i+1] e
  - trocando-os de posição caso valores[i] > valores[i+1].

valores:

▪ ▪ ▪ Oitava Iteração Externa

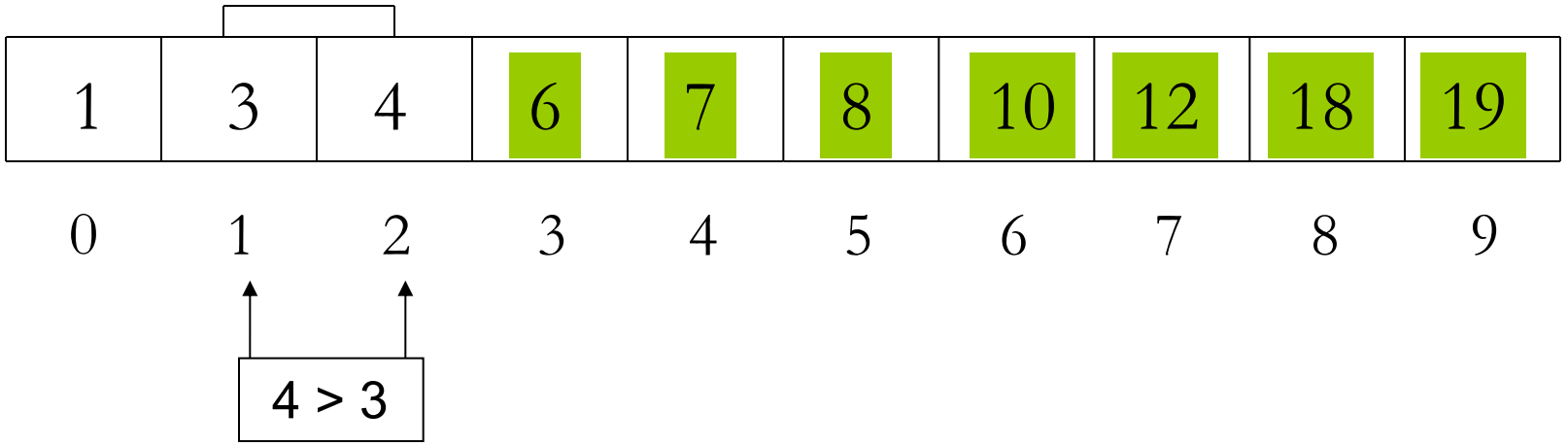


Ordenação pelo Método da Bolha (*BubbleSort*)

- Esta estratégia executa **n**-1 iterações, controlada por uma repetição mais externa.
- Em cada uma delas, via repetição interna,
  - percorre-se todo o vetor comparando cada par de elementos valores[i] e valores[i+1] e
  - trocando-os de posição caso valores[i] > valores[i+1].

valores:

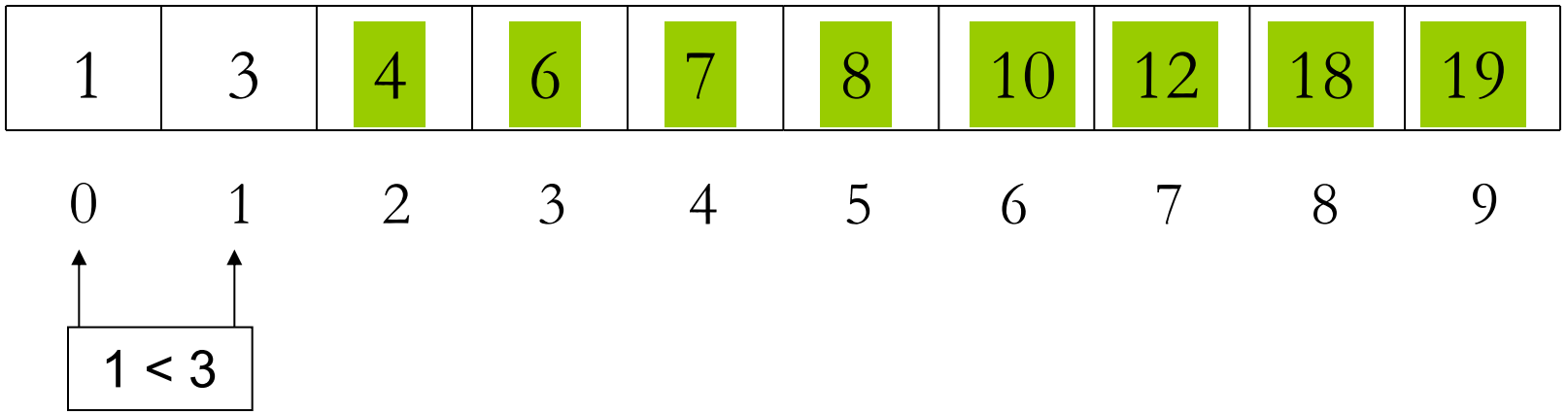
▪ ▪ ▪ Oitava Iteração Externa



Ordenação pelo Método da Bolha (*BubbleSort*)

- Esta estratégia executa **n**-1 iterações, controlada por uma repetição mais externa.
- Em cada uma delas, via repetição interna,
  - percorre-se todo o vetor comparando cada par de elementos valores[i] e valores[i+1] e
  - trocando-os de posição caso valores[i] > valores[i+1].

valores: Nona (última) Iteração Externa



Ordenação pelo Método da Bolha (*BubbleSort*)

```
# Operação que troca o conteúdo de duas células do vetor.
def trocar(vals, posX, posY):
        temp = vals[posX]
        vals[posX] = vals[posY]
        vals[posY] = temp
        return None

# Método da Bolha
def ordenar(valores):
    for tamanho in range(len(valores)-1,0,-1):
        for i in range(tamanho):
            if valores[i]>valores[i+1]:
                trocar(valores, i, i+1)
    return None
```

Ordenação pelo Método da Bolha (*BubbleSort*)

- Novamente, neste algoritmo, há basicamente duas estruturas de repetição **for** aninhadas.

Ordenação pelo Método da Bolha (*BubbleSort*)

- Novamente, neste algoritmo, há basicamente duas estruturas de repetição **for** aninhadas.

- A mais externa, executado **n**-1 vezes, representa as iterações. Na i-ésima iteração, **n**-i comparações são feitas.
  - Como no algoritmo anterior, aproximadamente **n** (**n**-1)/2 comparações são realizadas.

Ordenação pelo Método da Bolha (*BubbleSort*)

- Novamente, neste algoritmo, há basicamente duas estruturas de repetição **for** aninhadas.

- A mais externa, executado **n**-1 vezes, representa as iterações. Na i-ésima iteração, **n**-i comparações são feitas.
  - Como no algoritmo anterior, aproximadamente **n** (**n**-1)/2 comparações são realizadas.

- Desta forma, o custo computacional do método da bolha também é da ordem de $\mathbf{n}^2$.
  - Portanto sua complexidade é O($\mathbf{n}^2$).

Ordenação pelo Método da Bolha (*BubbleSort*)

- Nesta implementação, o algoritmo para na primeira iteração em que não houver nenhuma troca.

Ordenação pelo Método da Bolha (*BubbleSort*)

- Nesta implementação, o algoritmo para na primeira iteração em que não houver nenhuma troca.

- No pior caso, **n** iterações serão realizadas, o que não muda a complexidade de pior caso do algoritmo.

Ordenação pelo Método da Bolha (*BubbleSort*)

```
# Operação que troca o conteúdo de duas células do vetor.
def trocar(vals, posX, posY):
        temp = vals[posX]
        vals[posX] = vals[posY]
        vals[posY] = temp
        return None

# Outra forma do Método da Bolha – com saída rápida
def ordenar(valores):
    tamanho = len(valores)-1
    troquei = True
    while troquei:
        troquei = False
        for i in range(tamanho):
            if valores[i]>valores[i+1]:
                trocar(valores, i, i+1)
                troquei = True
        tamanho -= 1
    return None
```

Ordenação pelo Método da Partição (*QuickSort*)

- Trata-se de um método de ordenação eficiente e de natureza recursiva.

Ordenação pelo Método da Partição (*QuickSort*)

- Trata-se de um método de ordenação eficiente e de natureza recursiva.

- Em cada ativação do algoritmo:

  1. Um elemento do vetor é escolhido e é denominado pivô;

  2. Todos os elementos do vetor são re-arrumados, ocupando as primeiras células do vetor os elementos menores que o pivô, o pivô ocupa sua posição definitiva, seguido pelos valores maiores ou iguais a ele.

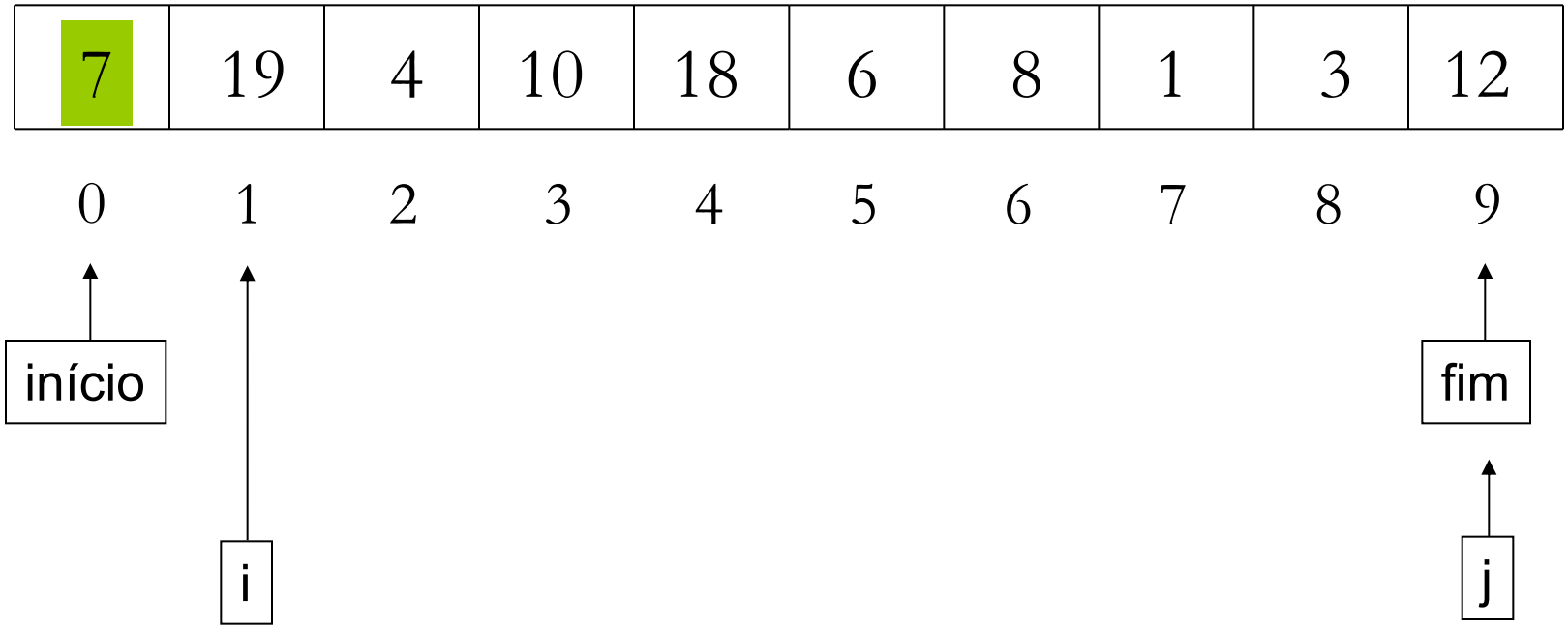
  3. Aplicamos a mesma ideia, com chamadas recursivas, aos subvetores, também chamados de partições, com tamanho maiores que 1, contendo elementos menores que o pivô, e com valores maiores ou iguais ao pivô, respectivamente.

  4. Quando todos os subvetores tiverem tamanhos menores ou iguais a um, o vetor original estará ordenado.

Ordenação pelo Método da Partição (*QuickSort*)

valores:

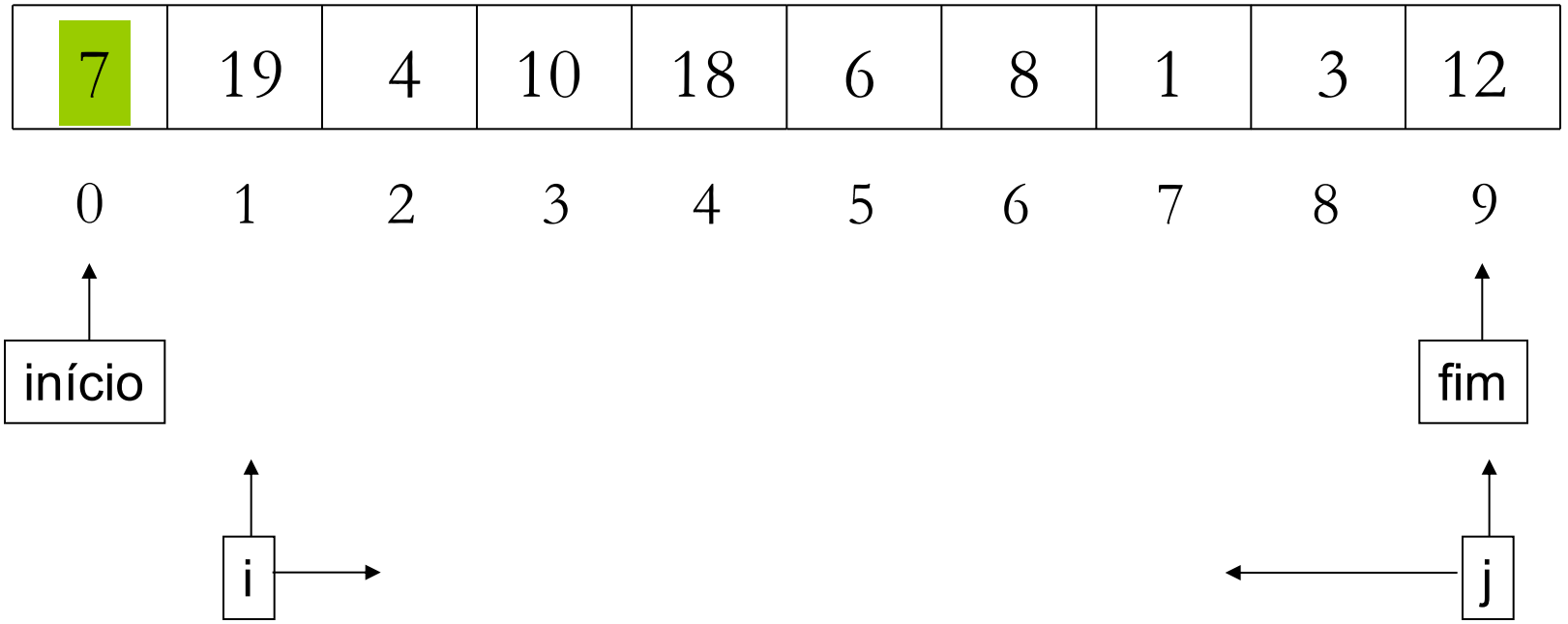
pivo = valores[início] = 7



Ordenação pelo Método da Partição (*QuickSort*)

valores:

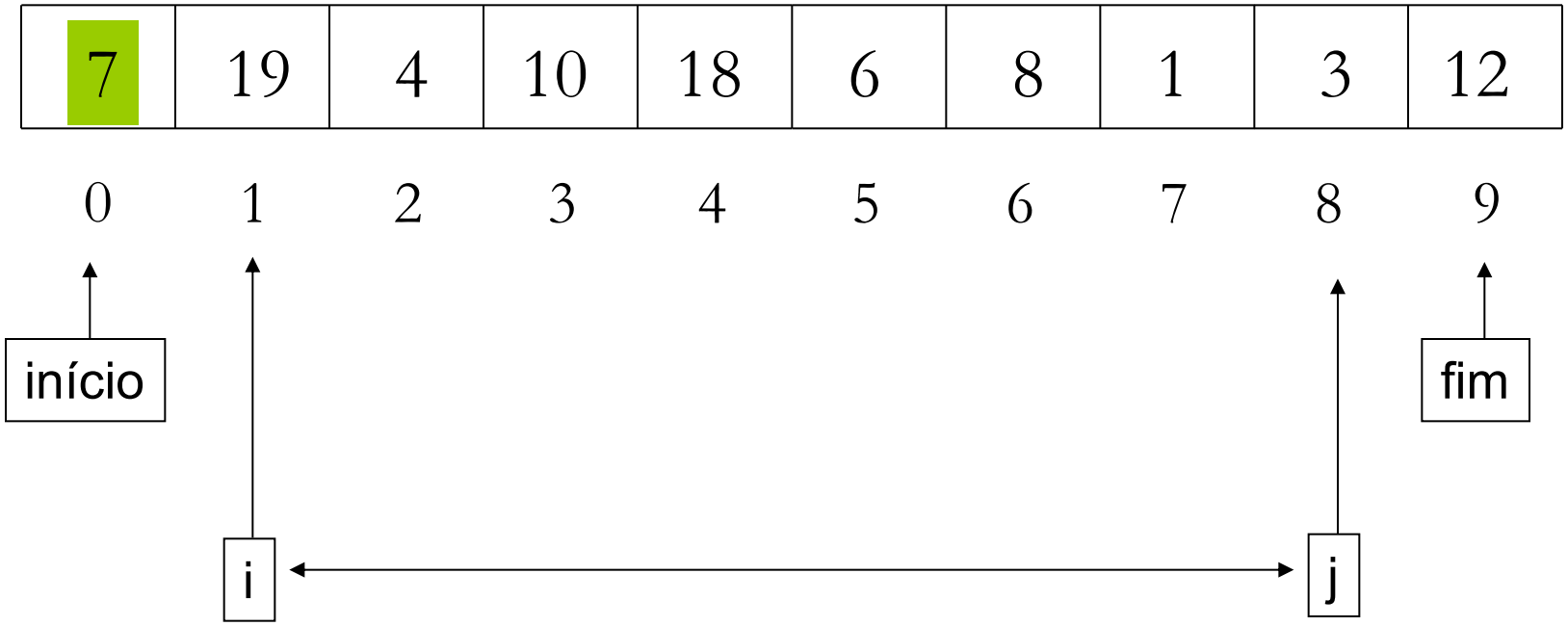
pivo = valores[início] = 7



Ordenação pelo Método da Partição (*QuickSort*)

valores:

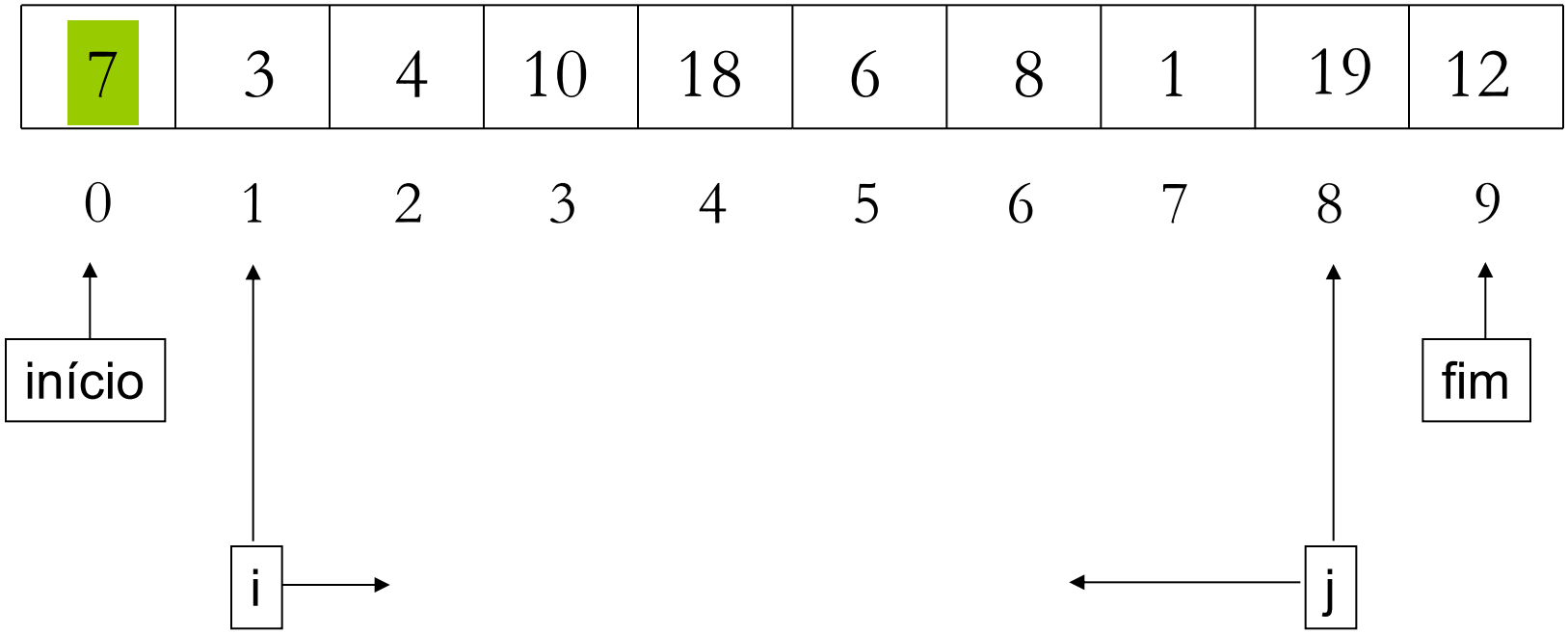
pivo = valores[início] = 7



Ordenação pelo Método da Partição (*QuickSort*)

valores:

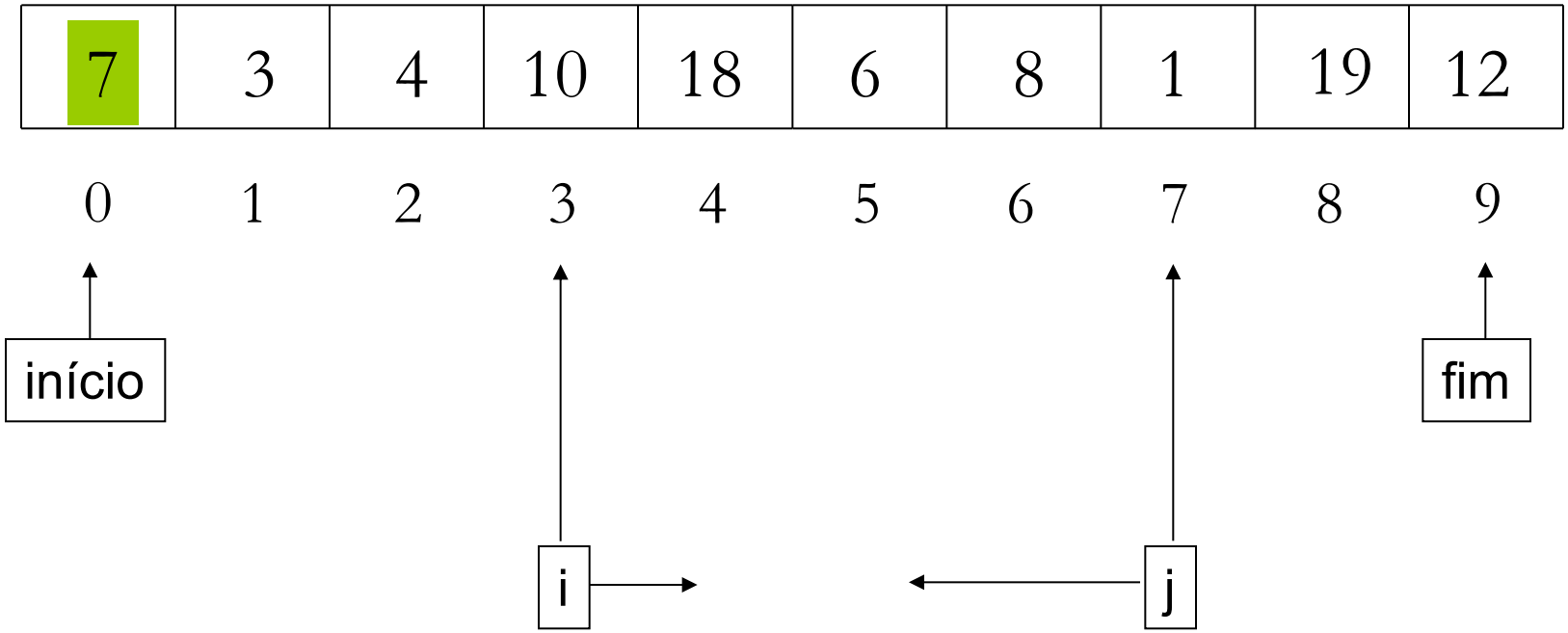
pivo = valores[início] = 7



Ordenação pelo Método da Partição (*QuickSort*)

valores:

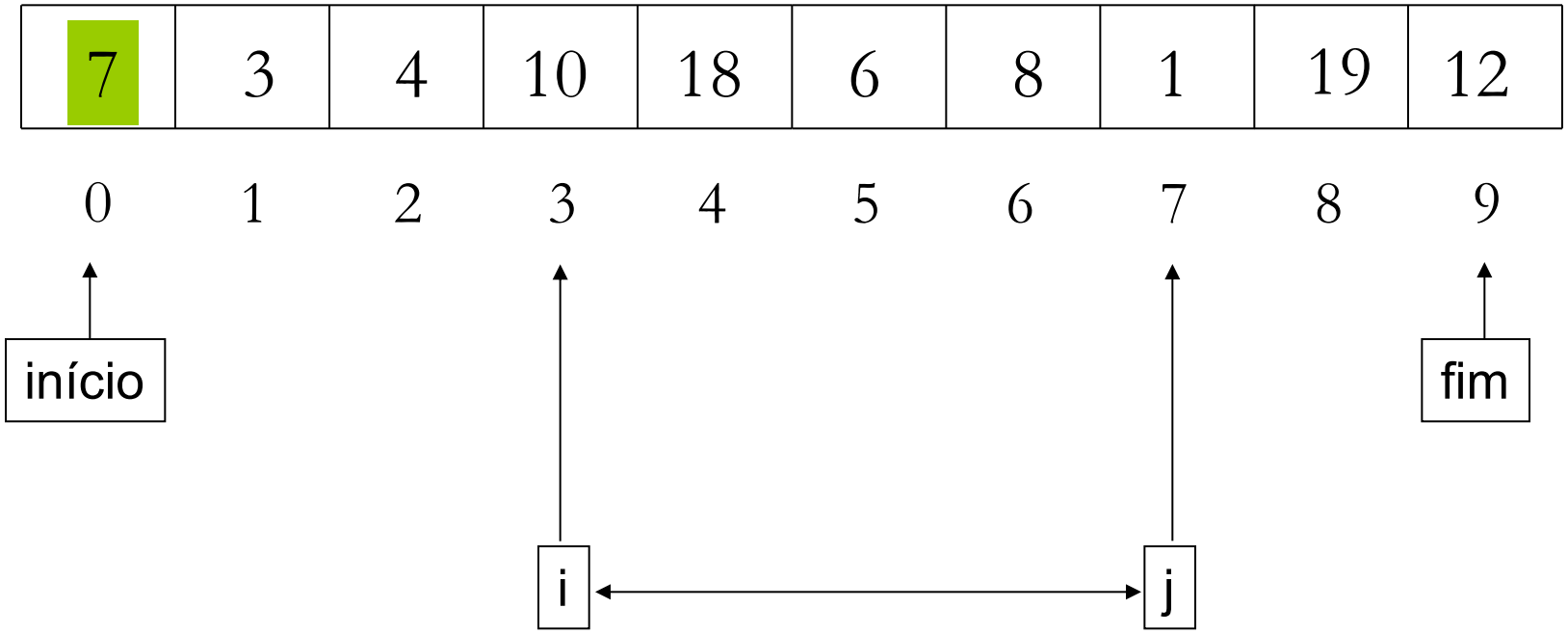
pivo = valores[início] = 7



Ordenação pelo Método da Partição (*QuickSort*)

valores:

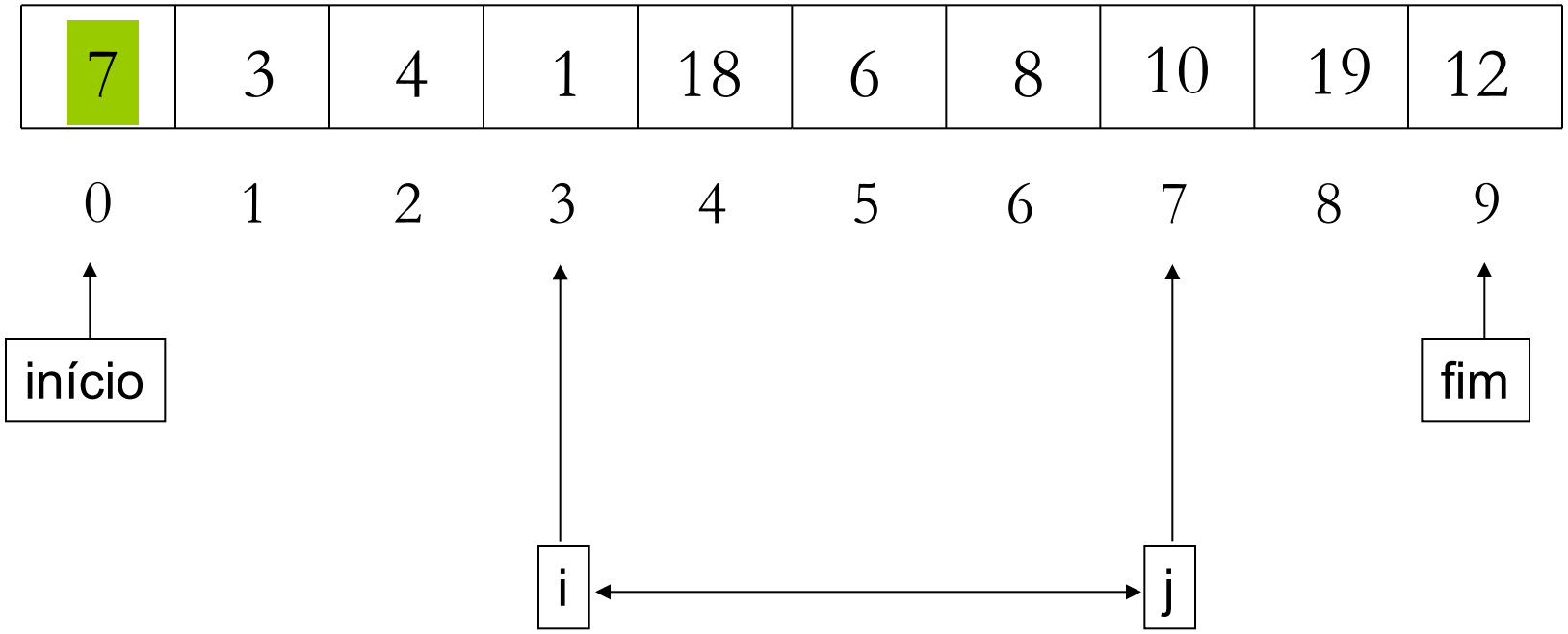
pivo = valores[início] = 7



Ordenação pelo Método da Partição (*QuickSort*)

valores:

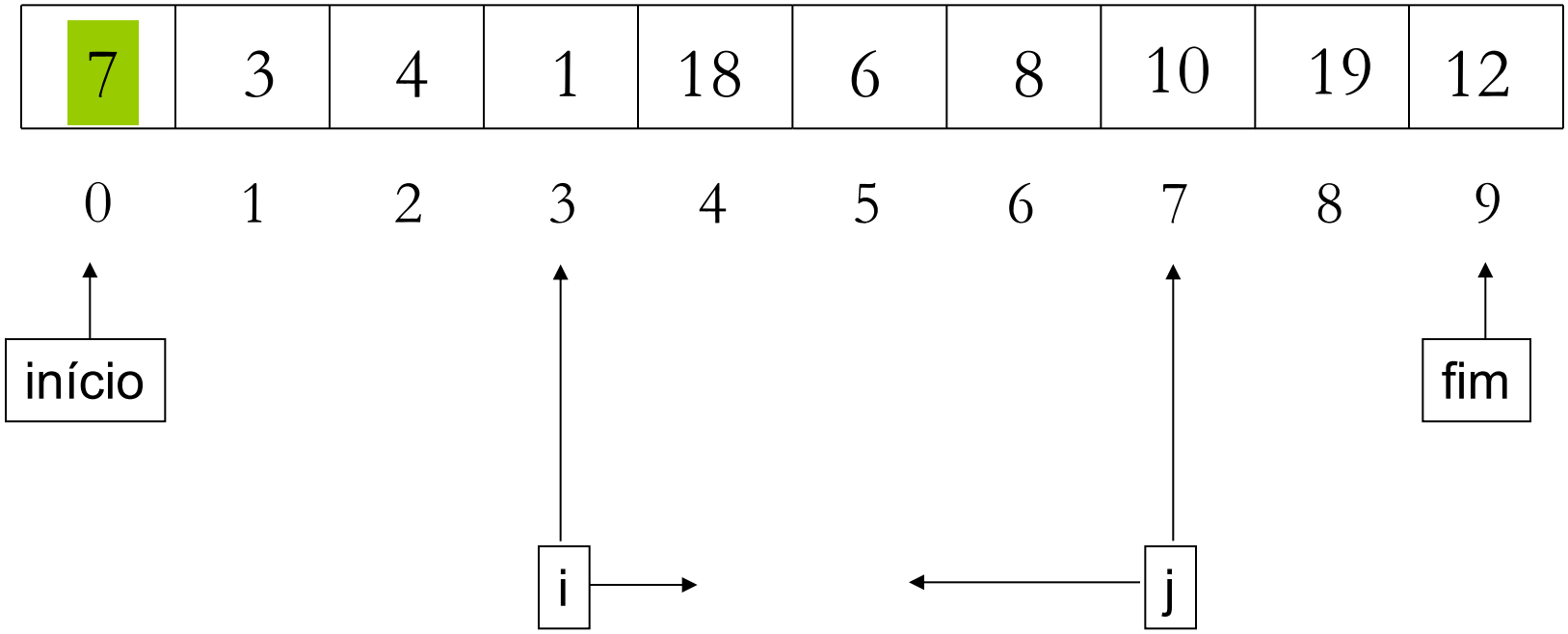
pivo = valores[início] = 7



Ordenação pelo Método da Partição (*QuickSort*)

valores:

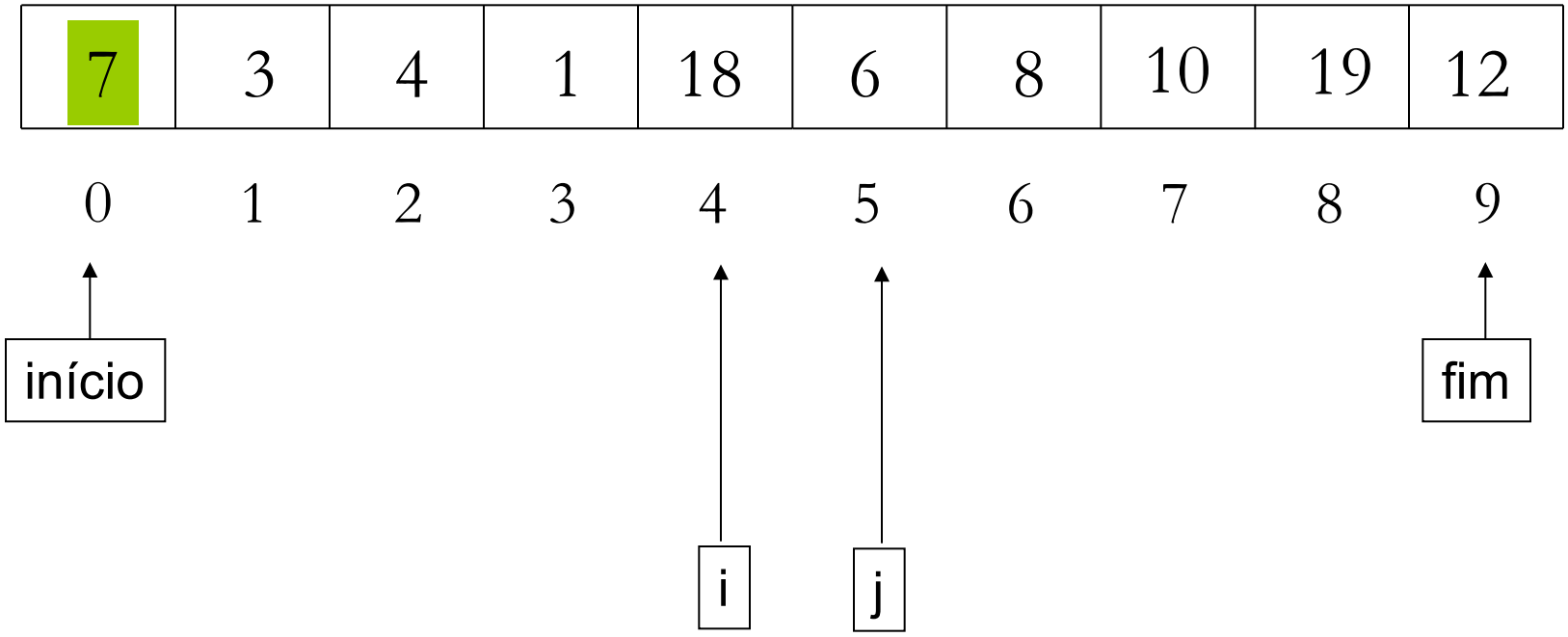
pivo = valores[início] = 7



Ordenação pelo Método da Partição (*QuickSort*)

valores:

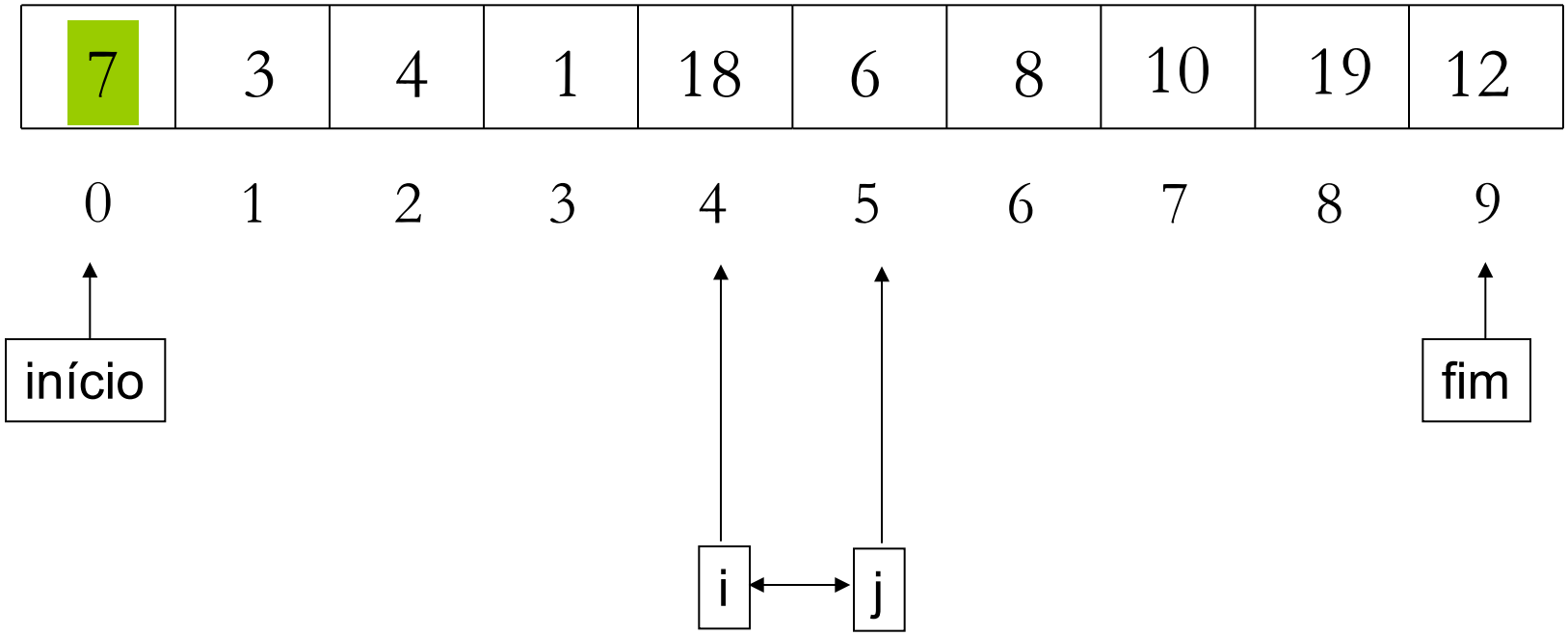
pivo = valores[início] = 7



Ordenação pelo Método da Partição (*QuickSort*)

valores:

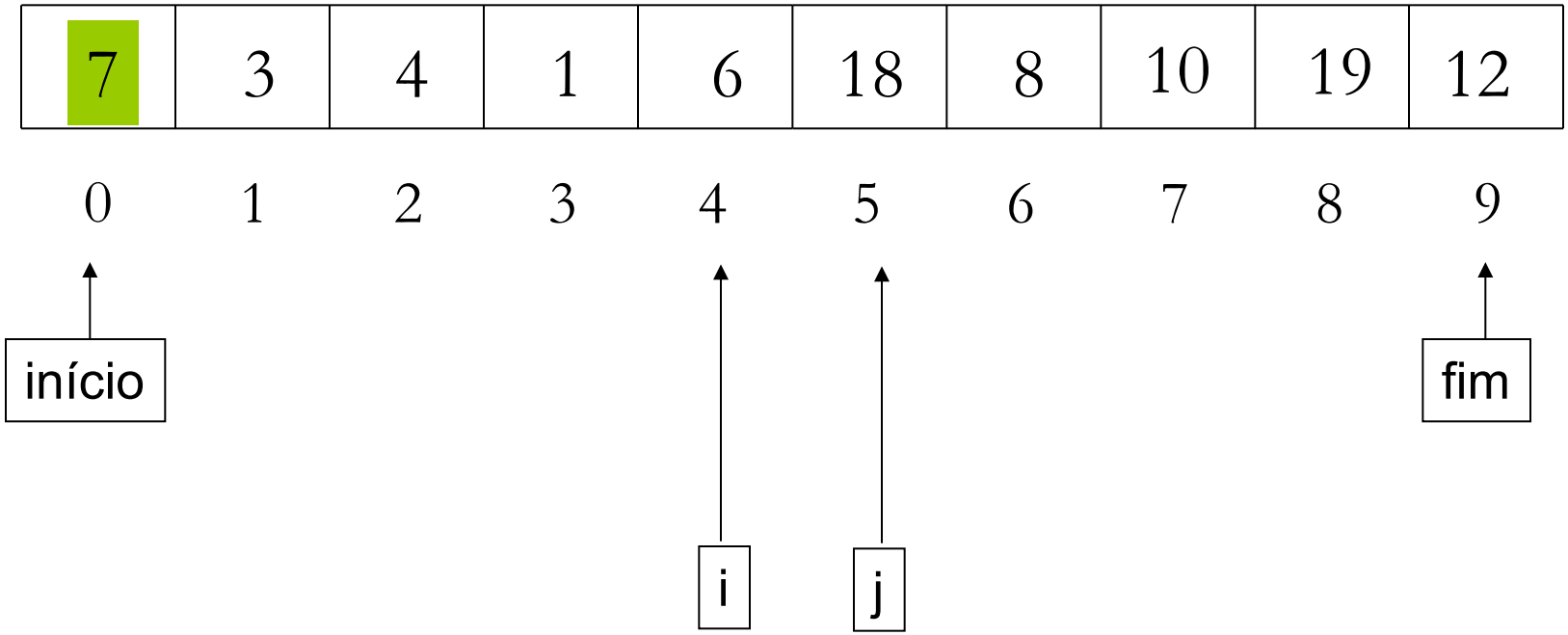
pivo = valores[início] = 7



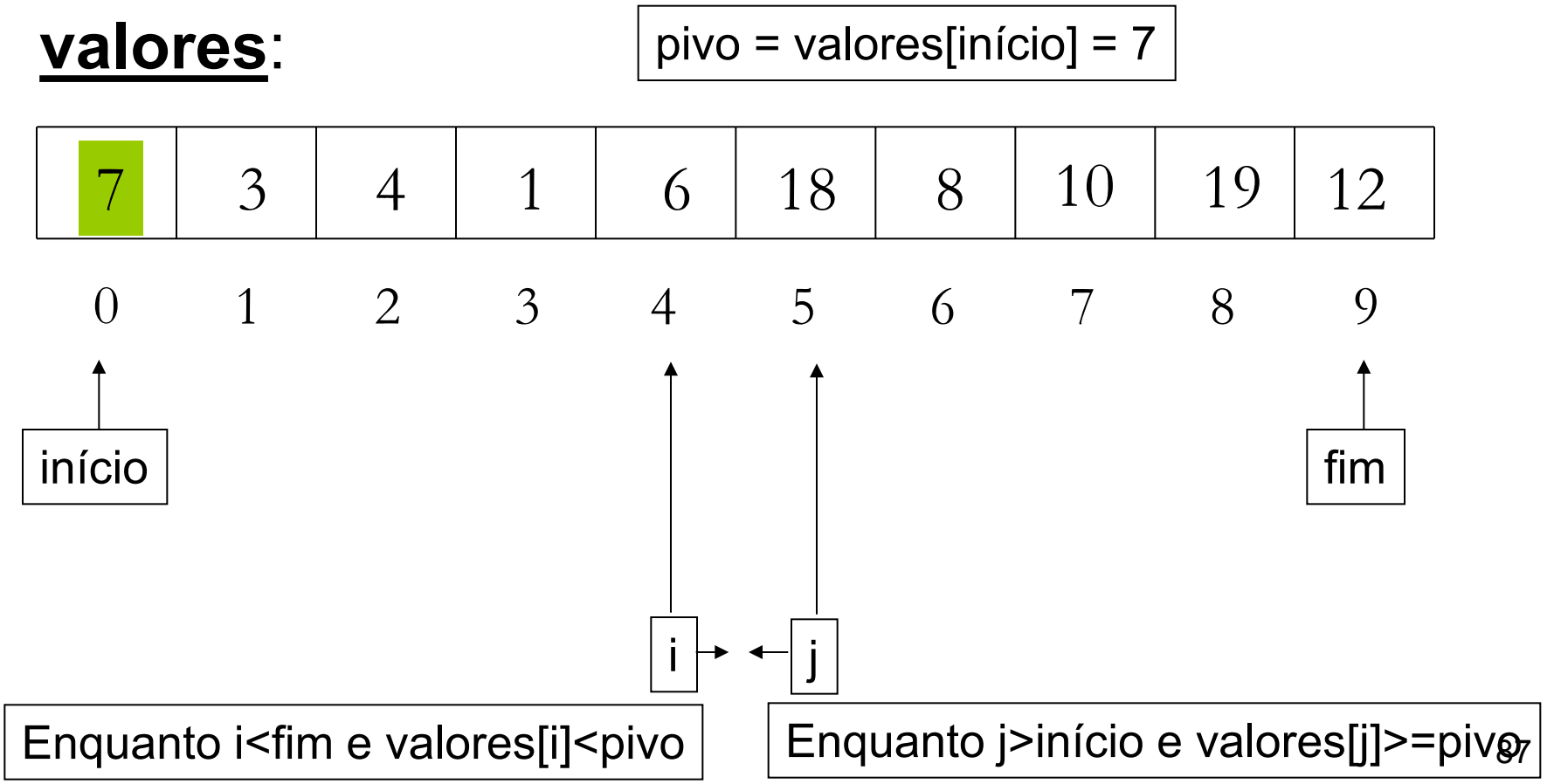
Ordenação pelo Método da Partição (*QuickSort*)

valores:

pivo = valores[início] = 7



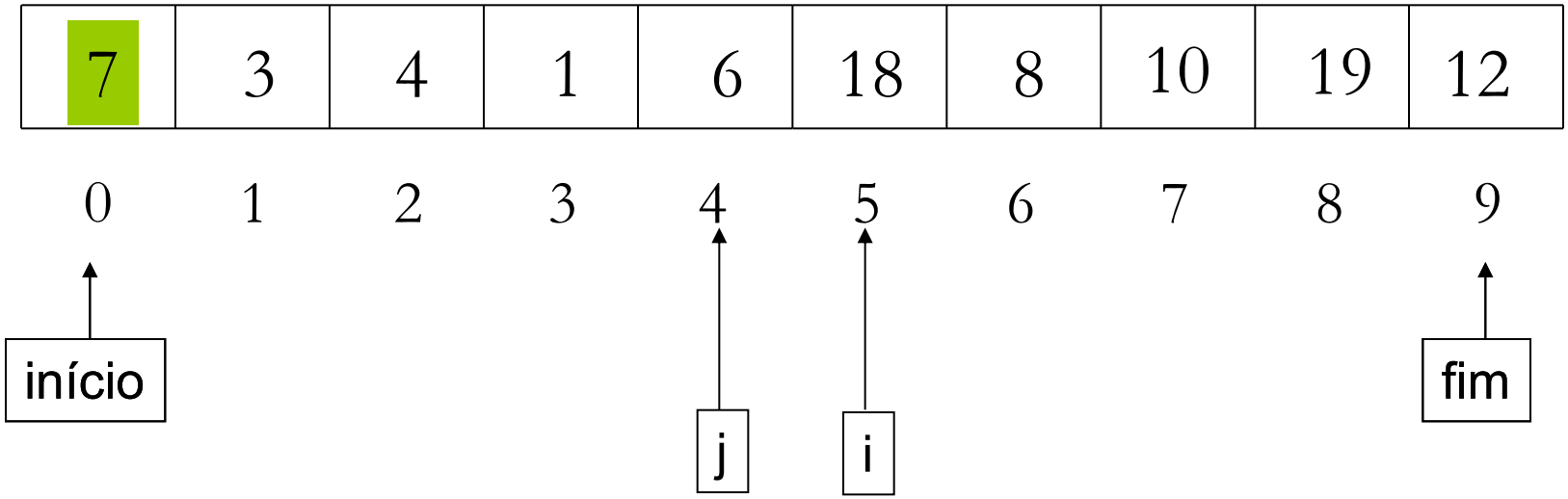
Ordenação pelo Método da Partição (*QuickSort*)



Ordenação pelo Método da Partição (*QuickSort*)

valores:

pivo = valores[início] = 7

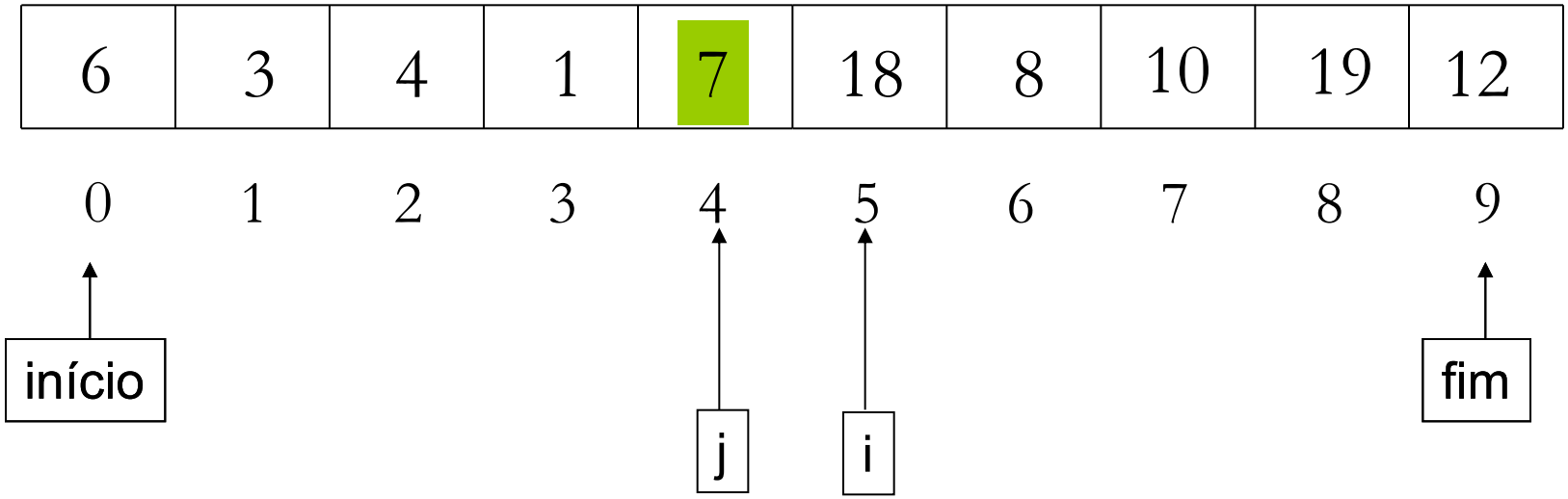


Dado que j < i, encerra-se a repetição. Em seguida coloca-se o pivô na sua posição definitiva e ativa-se recursivamente o algoritmo para os sub-vetores V[0,j-1] e V[j+1,9].

Ordenação pelo Método da Partição (*QuickSort*)

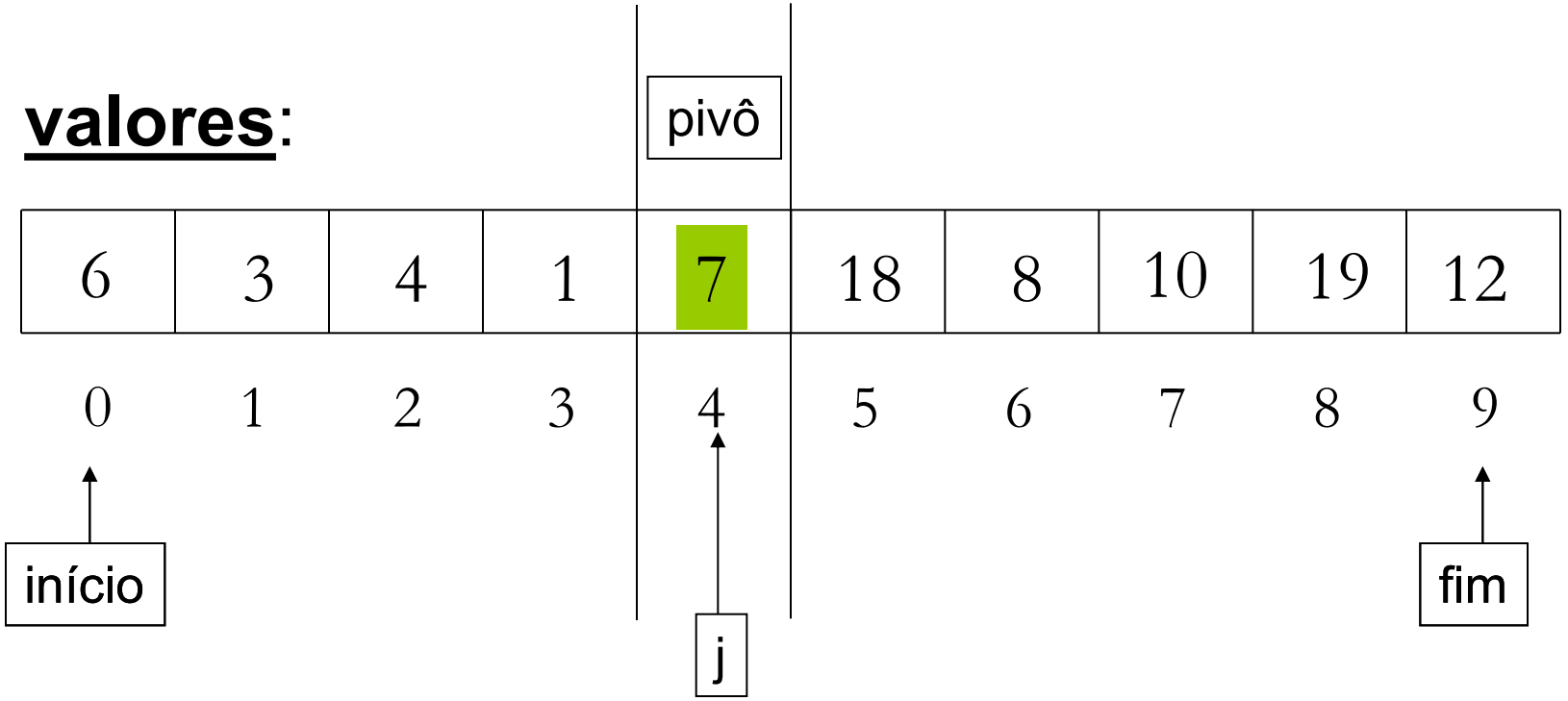
valores:

pivo = valores[início] = 7



Dado que j < i, encerra-se a repetição. Em seguida coloca-se o pivô na sua posição definitiva e ativa-se recursivamente o algoritmo para os sub-vetores V[0,j-1] e V[j+1,9].

Ordenação pelo Método da Partição (*QuickSort*)



Observe que todos os elementos em valores[0..j-1] são menores que o pivô e os elementos em valores[j+1..9] são maiores ou iguais ao pivô.

```
def particiona(vals, inicio, fim):
    pivo = vals[inicio]
    i  = inicio+1
    j  = fim
    while i < j:
        while i<fim and vals[i] < pivo:
            i += 1
        while j>inicio and vals[j] >= pivo:
            j -= 1
        if i < j:
            trocar(vals, i, j)
    if pivo>vals[j]:
        trocar(vals, inicio, j)
    return j
def quickSort(vals, inicio, fim):
    if inicio < fim:
        posPivo = particiona(vals,inicio,fim)
        quickSort(vals,inicio,posPivo-1)
        quickSort(vals,posPivo+1,fim)
    return None
def  ordena (valores):
    quickSort(valores, 0, len(valores)-1)
    return None
```

```
def particiona(vals, inicio, fim):
    pivo = vals[inicio]
    i  = inicio+1
    j  = fim
    while i < j:
        while i<fim and vals[i] < pivo:
            i += 1
        while j>inicio and vals[j] >= pivo:
            j -= 1
        if i < j:
            trocar(vals, i, j)
    if pivo>vals[j]:
        trocar(vals, inicio, j)
    return j
def quickSort(vals, inicio, fim):
    if inicio < fim:
        posPivo = particiona(vals,inicio,fim)
        quickSort(vals,inicio,posPivo-1)
        quickSort(vals,posPivo+1,fim)
    return None
def  ordena (valores):
    quickSort(valores, 0, len(valores)-1)
    return None
```

```
def particiona(vals, inicio, fim):
    pivo = vals[inicio]
    i  = inicio+1
    j  = fim
    while i < j:
        while i<fim and vals[i] < pivo:
            i += 1
        while j>inicio and vals[j] >= pivo:
            j -= 1
        if i < j:
            trocar(vals, i, j)
    if pivo>vals[j]:
        trocar(vals, inicio, j)
    return j
def quickSort(vals, inicio, fim):
    if inicio < fim:
        posPivo = particiona(vals,inicio,fim)
        quickSort(vals,inicio,posPivo-1)
        quickSort(vals,posPivo+1,fim)
    return None
def  ordena (valores):
    quickSort(valores, 0, len(valores)-1)
    return None
```

Ordenação pelo Método da Partição (*QuickSort*)

- No pior caso, este algoritmo executará aproximadamente **n** (**n**-1)/2 comparações.
  - Isto acontecerá quando o valor pivô gerar sempre dois subvetores, sendo um deles vazio e o outro subvetor com os demais elementos a menos do pivô, com **n**-1 elementos.

Ordenação pelo Método da Partição (*QuickSort*)

- No pior caso, este algoritmo executará aproximadamente **n** (**n**-1)/2 comparações.
  - Isto acontecerá quando o valor pivô gerar sempre dois subvetores, sendo um deles vazio e o outro subvetor com os demais elementos a menos do pivô, com **n**-1 elementos.

- Portanto, também se trata de um algoritmo O($\mathbf{n}^2$).

Ordenação pelo Método da Partição (*QuickSort*)

- No pior caso, este algoritmo executará aproximadamente **n** (**n**-1)/2 comparações.
  - Isto acontecerá quando o valor pivô gerar sempre dois subvetores, sendo um deles vazio e o outro subvetor com os demais elementos a menos do pivô, com **n**-1 elementos.

- Portanto, também se trata de um algoritmo $O(\mathbf{n}^2)$.

- Porém, é um algoritmo mais eficiente do que os demais apresentados, pois, na média, executará $\mathbf{n} \log_2(\mathbf{n})$, onde o fator $\log_2(\mathbf{n})$ se deve a divisão recursiva do vetor original.

Faça os Exercícios Relacionados a essa Aula

Clique no botão para visualizar os enunciados:

Aula 7

Professores:

Dante Corbucci Filho
Leandro A. F. Fernandes

Conteúdo:

- Moda
- Algoritmos de Ordenação
  - Método da Seleção (*SelectionSort*)
  - Método da Bolha (*BubbleSort*)
  - Método da Partição (*QuickSort)*
- Noções de Complexidade de Algoritmos