Algoritmos de Ordenação

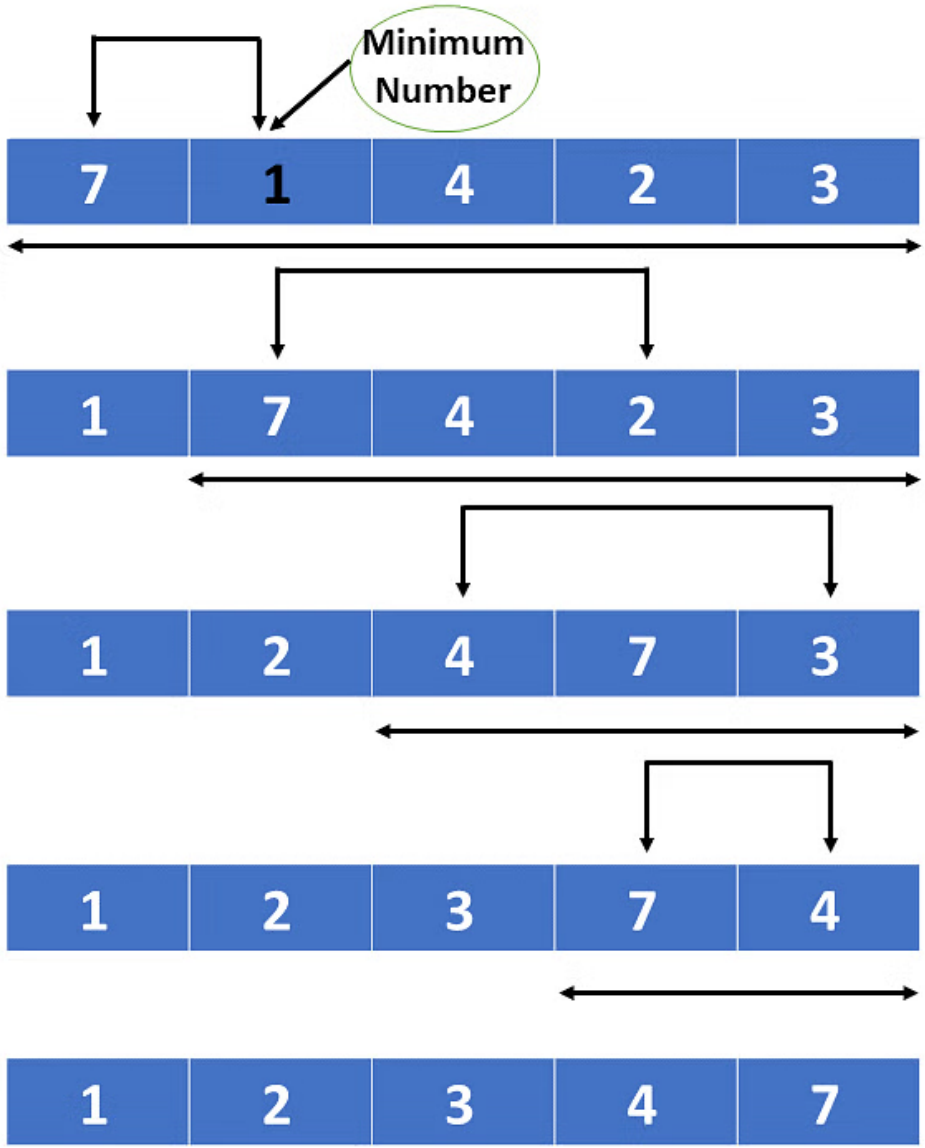
Prof. Gabriel Sobral

FIAP

18 de agosto de 2024

profgabriel.sobral@fiap.com.br

Tópicos

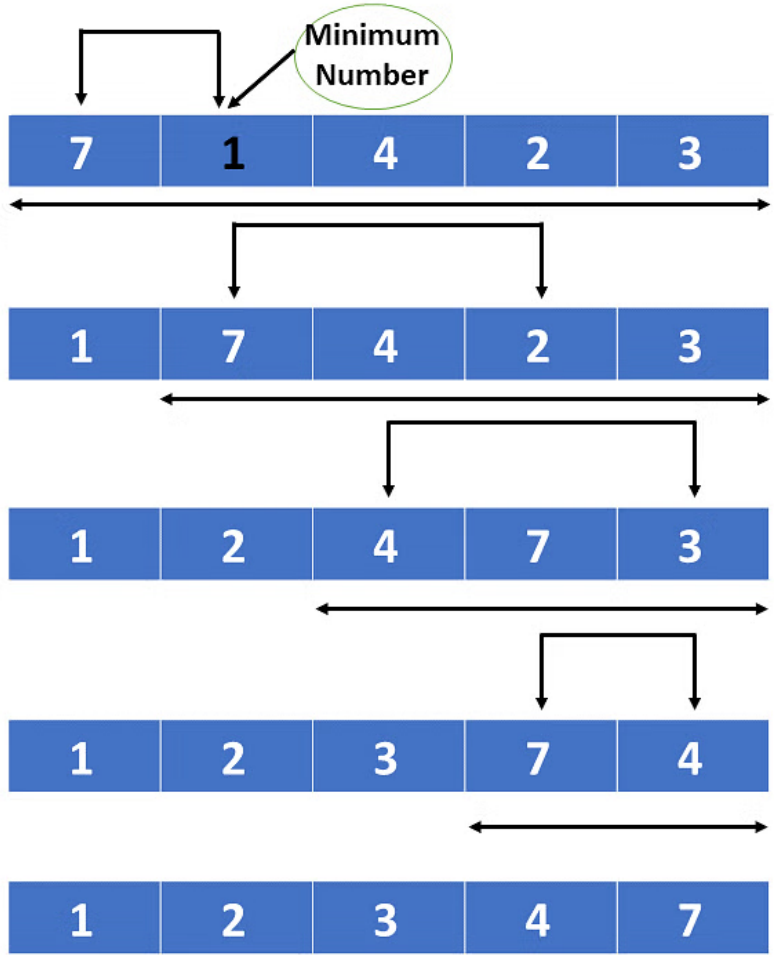

Minimum
Number
7 1 4 2 3
1 7 4 2 3
1 2 4 7 3
1 2 3 7 4
1 2 3 4 7

Algoritmo 1: Selection Sort(V, n)

Entrada: Um vetor V com n elementos

```
para i ← 0 até n − 1 faça
    id_min ← i
    para j ← i + 1 até n − 1 faça
        se V[j] < V[id_min] então
            id_min ← j
        troca V[i] com V[j]
    fim
fim
```

```
def selection_sort(vetor):
  for i in range(len(vetor)):
    id_min = i
    for j in range(i +1, len(vetor)):
      if vetor[j] < vetor[id_min]:
        id_min = j
    vetor[i], vetor[id_min] = vetor[id_min], vetor[i]
```



$$(n-1+1).\frac{n-1}{2} = \frac{n(n-1)}{2} = O(n^2)$$

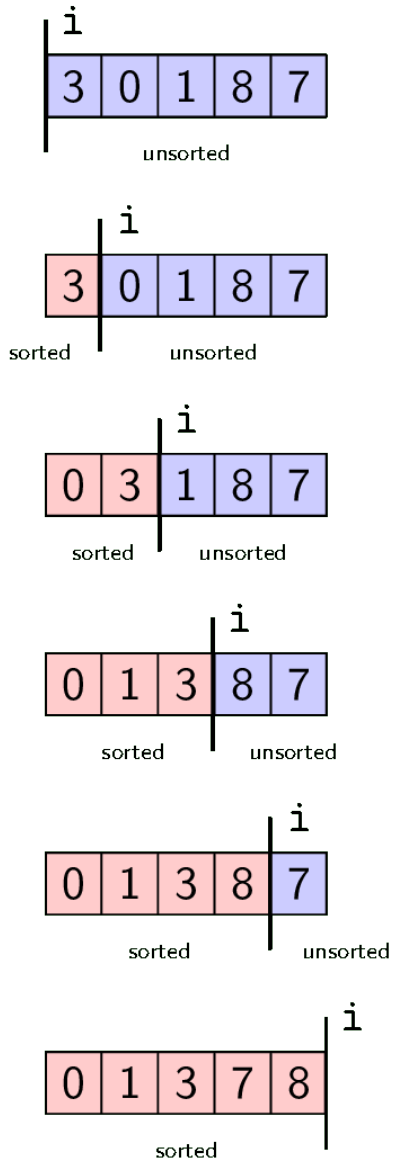
Selection Sort

Vantagens

- implementação simples
- não precisa estrutura de dados auxiliar
- rápido em vetores pequenos

Desvantagens

- lento para vetores grandes
- complexidade de tempo $O(n^2)$ (sempre)


i
3 0 1 8 7
unsorted
i
3 0 1 8 7
sorted
unsorted
i
0 3 1 8 7
sorted
unsorted
i
0 1 3 8 7
sorted
unsorted
i
0 1 3 8 7
sorted
unsorted
i
0 1 3 7 8
sorted

Algoritmo 2: Insertion Sort(V, n)

Entrada: Um vetor V com n elementos

para $i \leftarrow 1$ *até* $n-1$ **faça**

    $chave \leftarrow V[i]$

    $j \leftarrow i-1$

    enquanto $j \geq 0$ *e* $V[j] > chave$ **faça**

        $V[j+1] \leftarrow V[j]$

        $j \leftarrow j-1$

    fim

    $V[j] \leftarrow chave$

fim

```
def insertion_sort(vetor):
  for i in range(1, len(vetor)):
    chave = vetor[i]
    j = i -1
    while j >= 0 and vetor[j] > chave:
      vetor[j+1] = vetor[j]
      j -= 1
    vetor[j+1] = chave
```

37	29	14	10	8
0	1	2	3	4

$$(n-1+1).\frac{n-1}{2} = \frac{n(n-1)}{2} = O(n^2)$$

8	10	14	29	37
0	1	2	3	4

$$O(n)$$

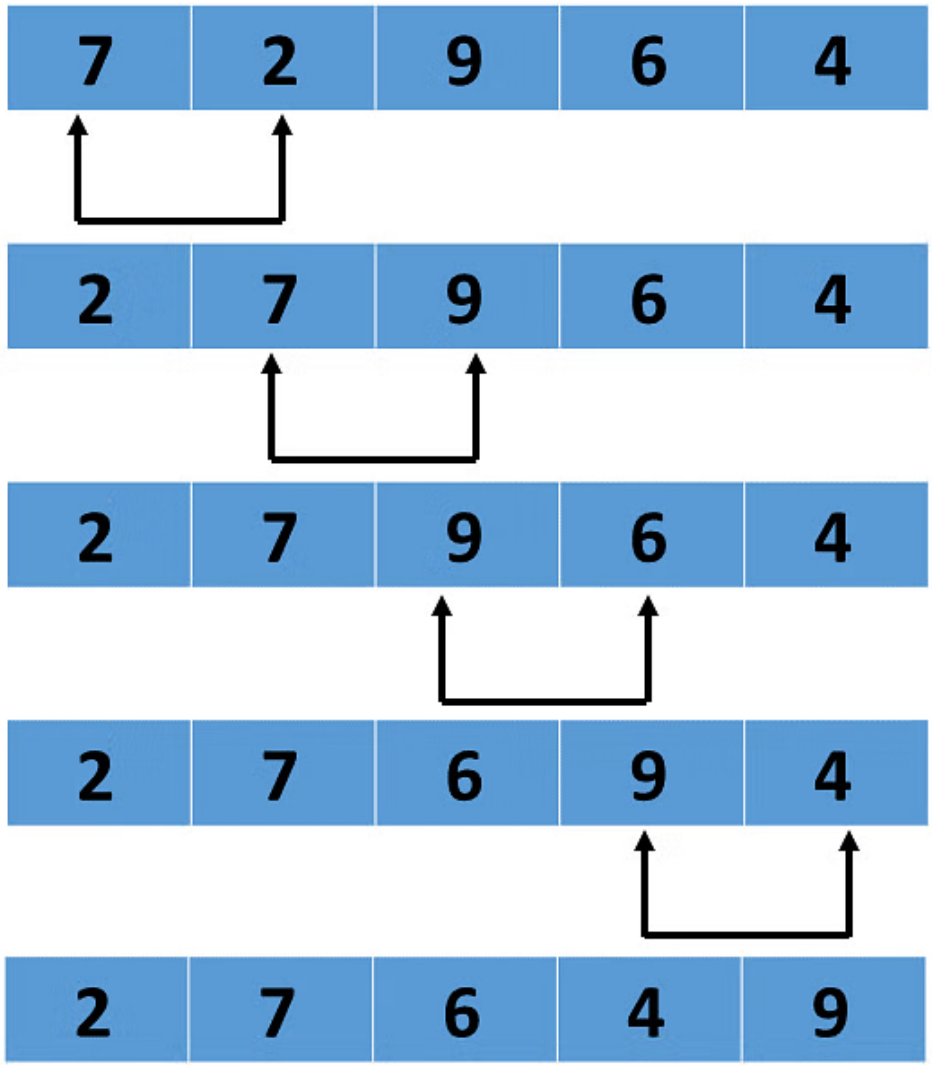
Insertion Sort

Vantagens

- implementação simples
- trocas são feitas no próprio vetor
- rápido em vetores pequenos
- $O(n)$ se o vetor está ordenado

Desvantagens

- ineficiente para vetores grandes
- não é tão eficiente quanto outros algoritmos de ordenação


7 2 9 6 4
2 7 9 6 4
2 7 9 6 4
2 7 6 9 4
2 7 6 4 9

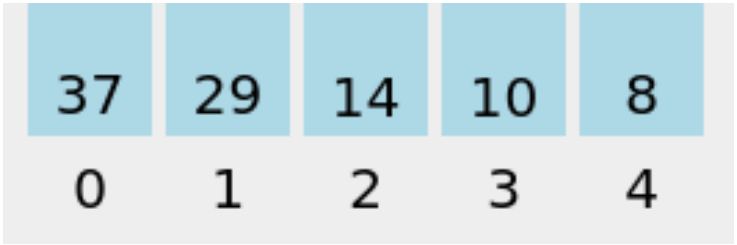
Algoritmo 3: Bubble Sort(V, n)

Entrada: Um vetor V com n elementos

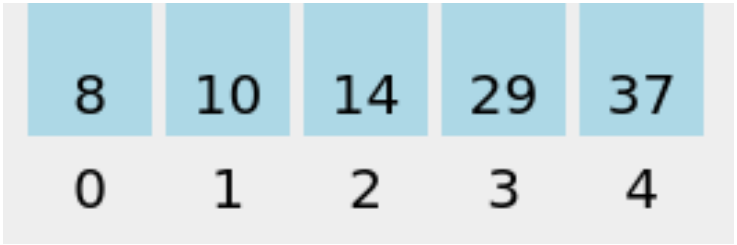
para $i \leftarrow 1$ *até* $n-1$ **faça**
  para $j \leftarrow n-1$ *até* $i+1$ **faça**
    se $V[j] < V[j-1]$ **então**
      troca $V[j]$ com $V[j-1]$
  fim
fim

```
def Bubble_sort(vetor):
  for i in range(len(vetor)):
    for j in range(0, len(vetor) -i -1):
      if vetor[j] > vetor[j + 1]:
        vetor[j], vetor[j+1] = vetor[j+1], vetor[j]
```

```
def Bubble_Sort(vetor):
  ordenado = False
  while ordenado is False:
    ordenado = True
    for i in range(len(vetor) -1):
      if vetor[i] > vetor[i+1]:
        vetor[i], vetor[i+1] = vetor[i+1], vetor[i]
        ordenado = False
```



$$(n-1+1).\frac{n-1}{2} = \frac{n(n-1)}{2} = O(n^2)$$



$$O(n)$$

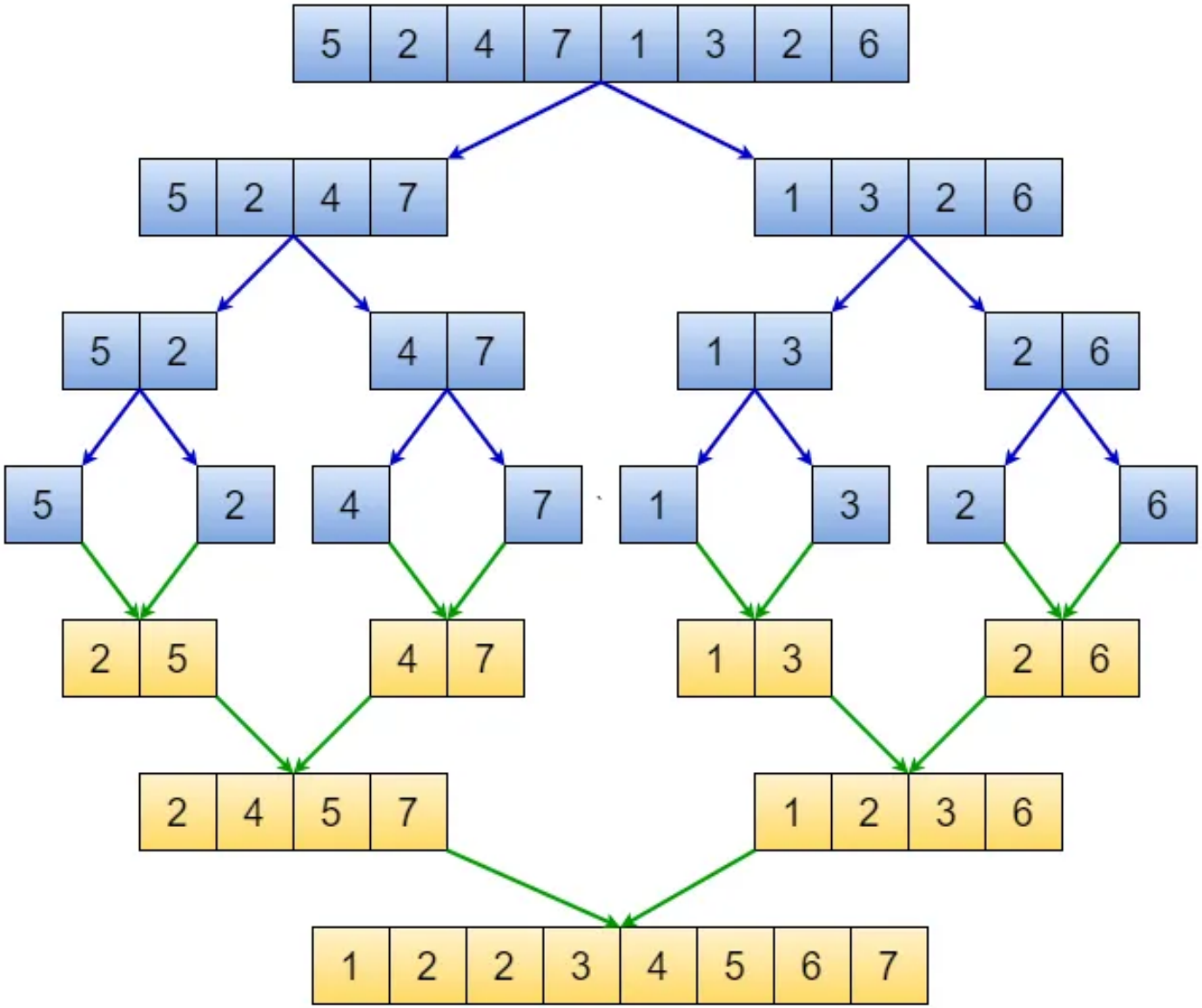
Bubble Sort

Vantagens

- implementação simples
- trocas dos elementos *in-place*
- $O(n)$ se o vetor está ordenado

Desvantagens

- ineficiente para vetores grandes
- pior e médio caso são $O(n^2)$


5 2 4 7 1 3 2 6
5 2 4 7
1 3 2 6
5 2
4 7
1 3
2 6
5
2
4
7
1
3
2
6
2 5
4 7
1 3
2 6
2 4 5 7
1 2 3 6
1 2 2 3 4 5 6 7

V: um vetor
e: índice que marca o início de V
d: índice que marca o fim de V

Algoritmo 4: Merge Sort(V, e, d)

se $e < d$ **então**
    $meio \leftarrow \lfloor (d + e)/2 \rfloor$
    Merge Sort($V, e, meio$)
    Merge Sort($V, meio + 1, d$)
    Combina($V, e, meio, d$)

Algoritmo 5: Combina($V, e, meio, d$)

$n_1 \leftarrow d - e + 1$

$n_2 \leftarrow d - meio$

Sejam $E[1, \ldots, n_1 + 1]$ e $D[1, \ldots, n_2 + 1]$ dois vetores

para $i \leftarrow 1$ *até* n_1 **faça**

$\quad E[i] = V[e + i - 1]$

fim

para $i \leftarrow 1$ *até* n_2 **faça**

$\quad D[i] \leftarrow V[d + j]$

fim

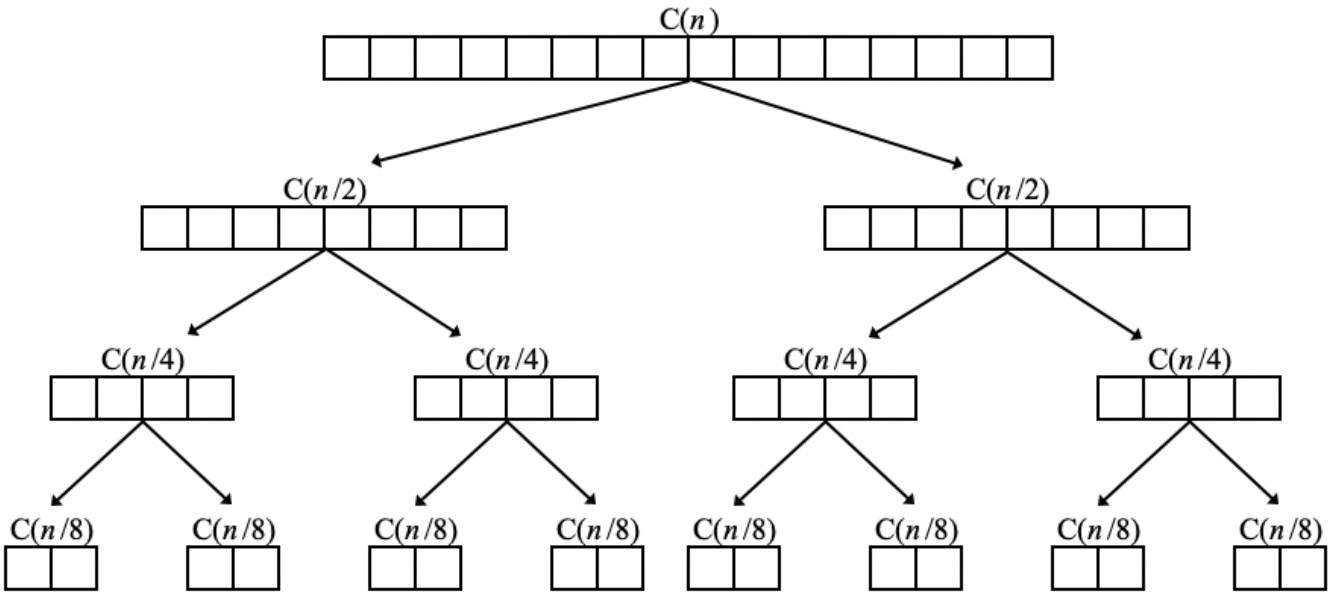
```
i ← 1
j ← 1
para k ← e até d faça
    se E[i] ≤ D[j] então
        V[k] ← E[i]
        i ← i + 1
    fim
    V[k] ← D[j]
    j ← j + 1
fim
```

```
def merge_sort(vetor: list[any],
               esquerda: int,
               direita: int) -> None:
    if esquerda < direita:
        meio = esquerda + (direita - esquerda) // 2
        merge_sort(vetor, esquerda, meio)
        merge_sort(vetor, meio +1, direita)
        combina(vetor, esquerda, meio, direita)
```

```
def combina(vetor: list[any],
            esquerda: int,
            meio: int,
            direita: int) -> None:
    n1 = meio - esquerda +1
    n2 = direita - meio

    subvetor_e = [vetor[esquerda +i] for i in range(n1)]
    subvetor_d = [vetor[meio +1 +j]  for j in range(n2)]
    i = j = 0
    k = esquerda
```

```
while i < n1 and  j < n2:
    if subvetor_e[i] <= subvetor_d[j]:
        vetor[k] = subvetor_e[i]
        i += 1

    else:
        vetor[k] = subvetor_d[j]
        j += 1

    k += 1
```

```
while i < n1:
    vetor[k] = subvetor_e[i]
    i += 1
    k += 1

while j < n2:
    vetor[k] = subvetor_d[j]
    j += 1
    k += 1
```



$\log_2 n$ levels

Merging Costs
n
$2(n/2) = n$
$4(n/4) = n$
$8(n/8) = n$

Total = $n \log_2 n$

$$\Theta(n \lg n)$$

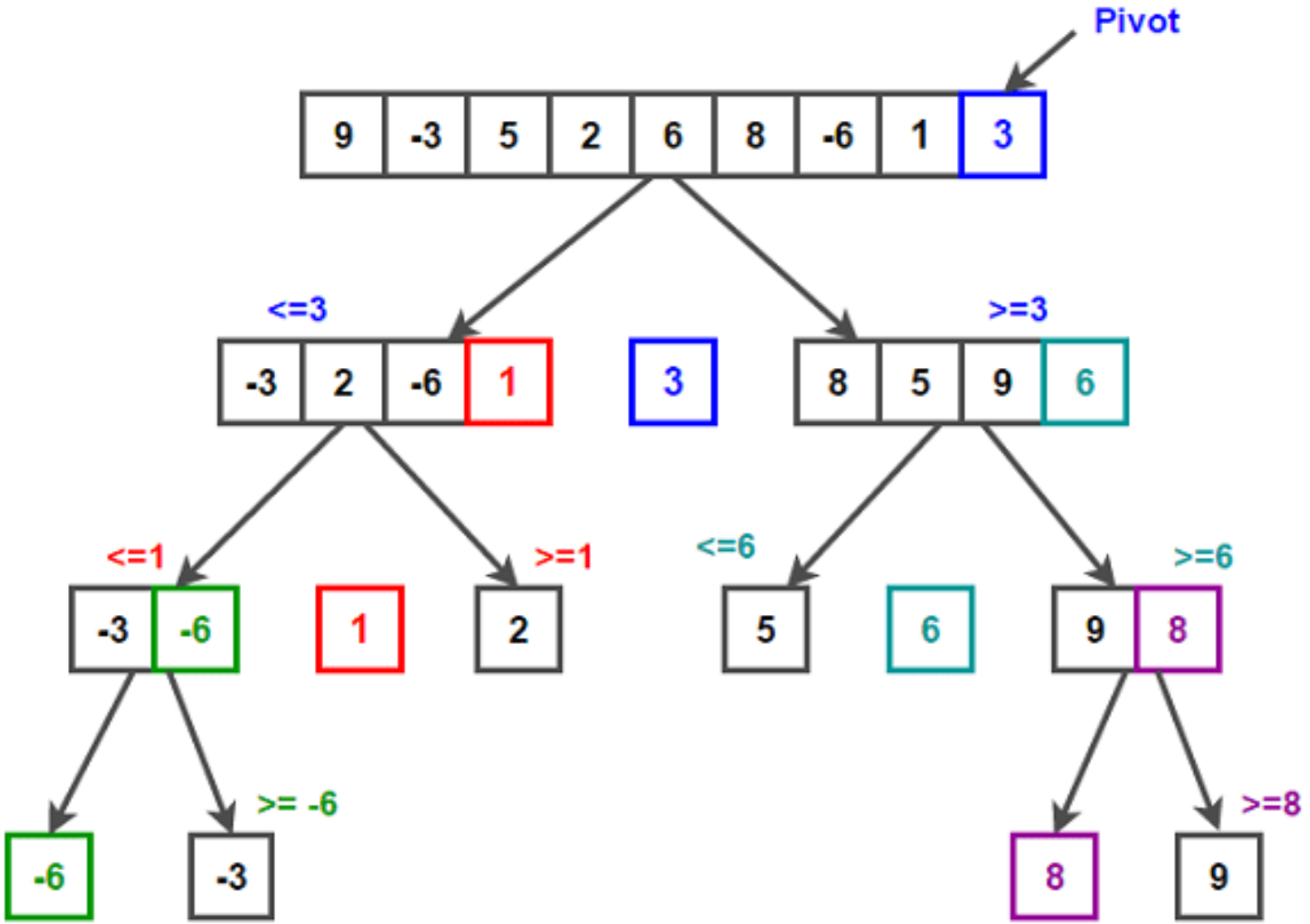
Merge Sort

Vantagens

- eficiente, custa $\Theta(n \lg n)$

Desvantagens

- precisa de estrutura de dados auxiliar
- ineficiente para vetores pequenos


Pivot
9 -3 5 2 6 8 -6 1 3
<=3
-3 2 -6 1
3
8 5 9 6
>=3
<=1
-3 -6
1
2
>=1
<=6
5
6
9 8
>=6
-6
-3
>= -6
8
9
>=8

V: um vetor
e: índice que marca o início de V
d: índice que marca o fim de V

Algoritmo 6: Quick Sort(V, e, d)

se $e < q$ **então**
    $pivot \leftarrow$ Particiona(V, e, q)
    Quick Sort(V, e, $meio - 1$)
    Quick Sort(V, $meio + 1$, d)

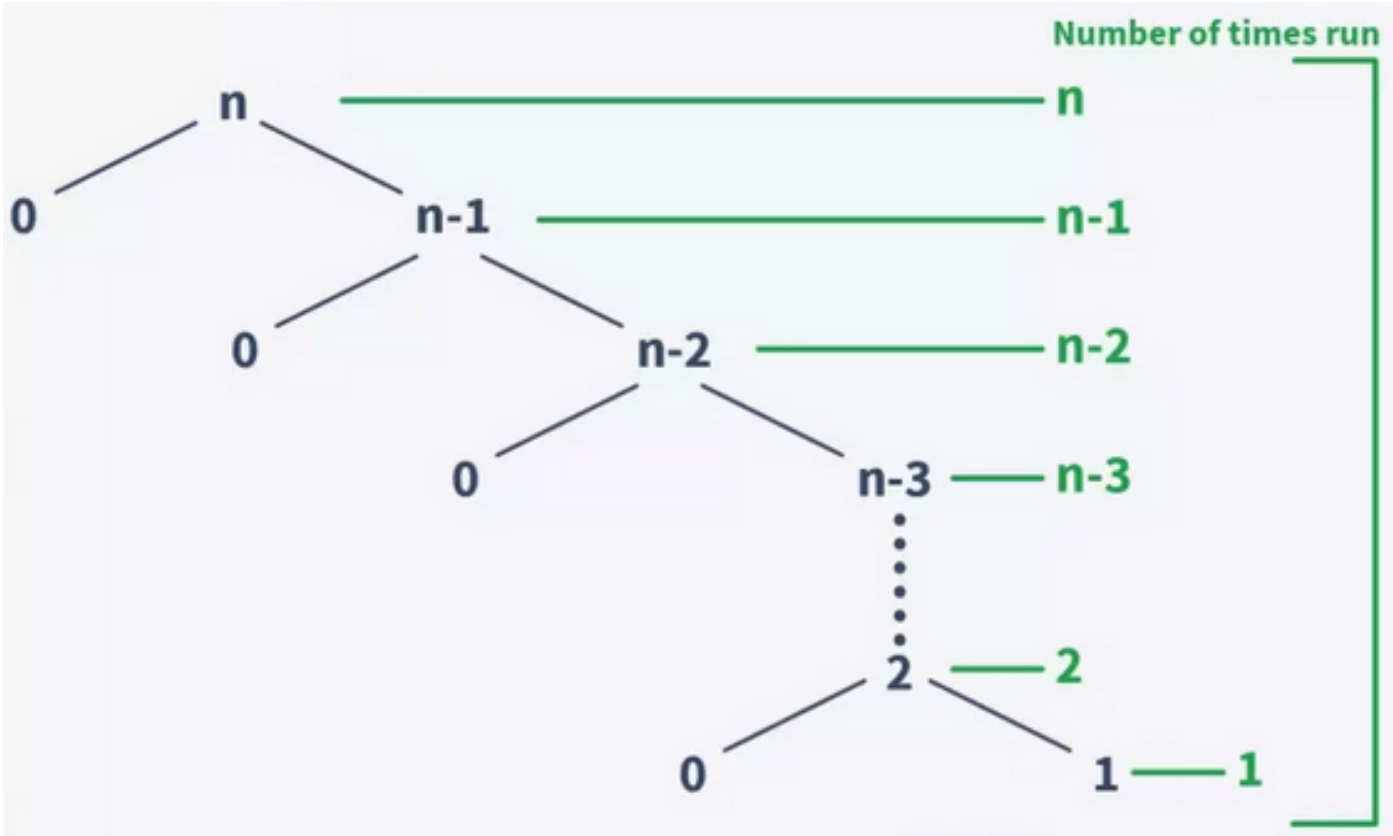
Algoritmo 7: Particiona(V, e, d)

$x \leftarrow V[d]$
$i \leftarrow e - 1$
para $j \leftarrow e$ *até* $d - 1$ **faça**
    se $V[j] \leq x$ **então**
        $i \leftarrow i + 1$
        troca $V[i]$ com $V[j]$
fim
troca $V[i + 1]$ com $V[r]$
retorna $i + 1$

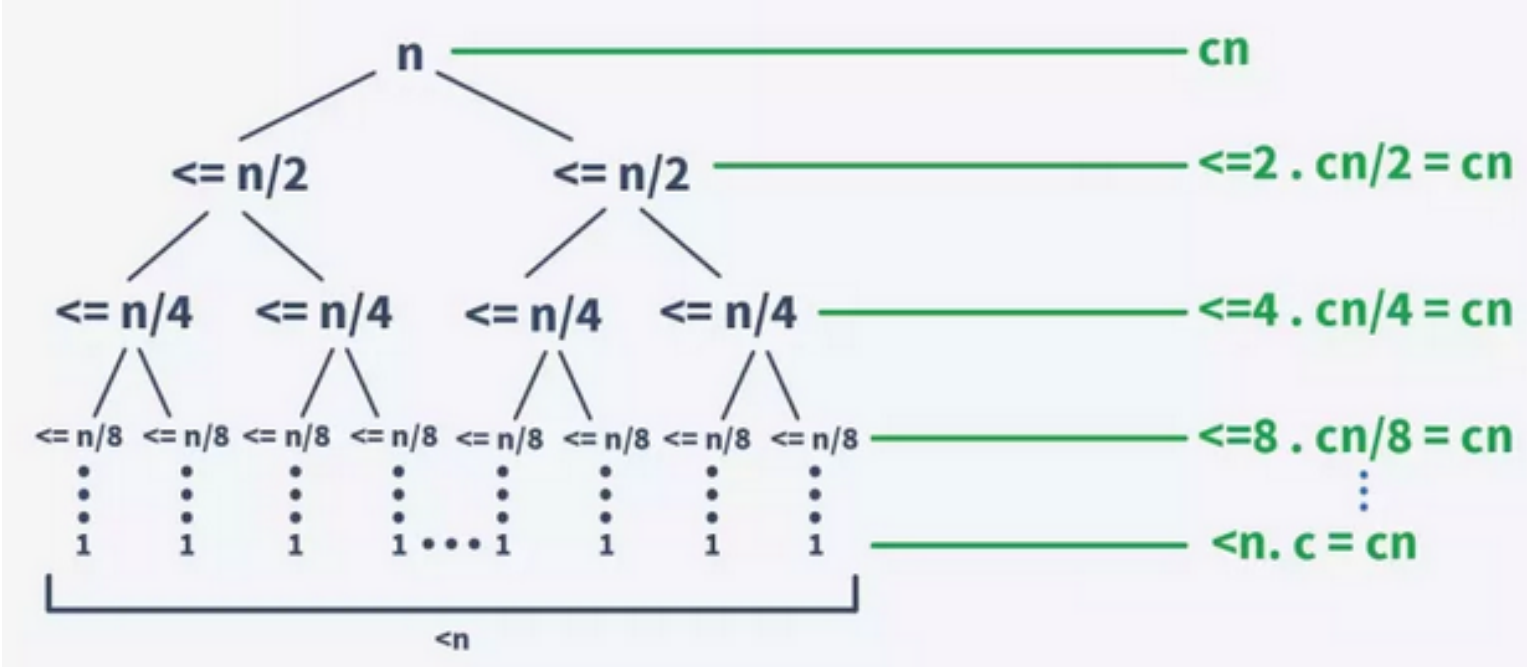
```
def quick_sort(vetor: list[any],
               esquerda: int,
               direita: int) -> None:
    if esquerda < direita:
        pivot = particiona(vetor, esquerda, direita)
        quick_sort(vetor, esquerda, pivot -1)
        quick_sort(vetor, pivot +1, direita)
```

```
def particiona(vetor: list[any],
               esquerda: int,
               direita: int) -> int :
    pivot = vetor[direita]
    i = esquerda -1

    for j in range(esquerda, direita):
        if vetor[j] <= pivot:
            i += 1
            vetor[i], vetor[j] = vetor[j], vetor[i]

    vetor[i +1], vetor[direita] = vetor[direita],
                                  vetor[i +1]

    return i +1
```

$$[1, 2, 3, 4, 5]$$



$$O(n^2)$$


n
cn
<= n/2
<= n/2
<=2 . cn/2 = cn
<= n/4
<= n/4
<= n/4
<= n/4
<=4 . cn/4 = cn
<= n/8
<= n/8
<= n/8
<= n/8
<= n/8
<= n/8
<= n/8
<= n/8
<=8 . cn/8 = cn
1
1
1
1
1
1
1
1
<n. c = cn
<n

$$\Theta(n \lg n)$$

Quick Sort

Vantagens

- $\Omega(n \lg n)$ no melhor e no médio caso
- não usa estrutura auxiliar

Desvantagens

- $O(n^2)$ no pior caso