

Ordenação - Métodos eficientes

Quick Sort e Merge Sort

Prof. Joaquim Quinteiro Uchôa
Profa. Juliana Galvani Gregghi
Profa. Marluce Rodrigues Pereira
Profa. Paula Christina Cardoso
Prof. Renato Ramos da Silva



Roteiro

- Problema do Particionamento
- Quick Sort
- Particionamento de Lomuto

Problema do Particionamento



Quick Sort - Visão Geral



O quick sort é um algoritmo de ordenação eficiente bastante popular, sendo possivelmente a implementação mais utilizada de ordenação em bibliotecas de uso geral.

É um algoritmo recursivo, baseado na ideia de dividir para conquistar, cujas bases estão no problema do particionamento (ou separação):

rearranjar um vetor $v[p..r]$ de modo que todos os elementos menores que um pivô fiquem na parte esquerda do vetor e todos os elementos maiores fiquem na parte direita.

Problema do Particionamento - i



O problema da particionamento admite diferentes formulações, por exemplo:

Rearranjar $v[p..r]$ de modo que $v[p..j-1] \leq v[j] < v[j+1..r]$, para algum j em $p...r$. ($v[j]$ é o pivô.)

Exemplo com pivô explícito (21):

0	7	1	2	21	59	48	24
---	---	---	---	----	----	----	----

Problema do Particionamento - ii



Uma outra reformulação é dada por:

Rearranjar $v[p..r]$ de modo a obter
 $v[p..j] \leq v[j+1..r]$ para algum j em $p \dots r-1$
(pivô implícito)

Exemplo com pivô implícito:

24	0	7	21	1	2	59	48
----	---	---	----	---	---	----	----

Particionamento - Exemplo:

Considere o vetor [24, 36, 0, 7, 48, 1, 59, 2, 4, 78, 21, 12, 6, 50, 5] e que o pivô escolhido seja a primeira posição, tem-se então, como possível resultado do particionamento:

[0, 7, 1, 2, 4, 21, 12, 6, 5, 24, 36, 48, 59, 78, 50]

Note que não há ordenamento nas partições, apenas a separação dos maiores e menores.

Algoritmo de Particionamento

```
int particiona(int v[], int c, int f) { // c = começo, f = fim
    int pivo = v[c], i = c+1, j = f;
    while (i <= j) {
        if (v[i] <= pivo) i++;
        else if (pivo <= v[j]) j--;
        else { // (v[i] > pivo) e (v[j] < pivo)
            swap (v[i],v[j]);
            i++;
            j--;
        }
    } // agora i == j+1
    v[c] = v[j];
    v[j] = pivo;
    return j; // retorna posição do pivô
}
```



Algoritmo de Particionamento

```
int particiona(int v[], int c, int f) { // c = começo, f = fim
    int pivo = v[c], i = c+1, j = f;
    while (i <= j) {
        if (v[i] <= pivo) i++;
        else if (pivo <= v[j]) j--;
        else { // (v[i] > pivo) e (v[j] < pivo)
            swap (v[i],v[j]);
            i++;
            j--;
        }
    } // agora i == j+1
    v[c] = v[j];
    v[j] = pivo;
    return j; // retorna posição do pivô
}
```

Enquanto $v[i]$ for menor que o pivô, avança na parte inicial do vetor

Algoritmo de Particionamento

```
int particiona(int v[], int c, int f) { // c = começo, f = fim
    int pivo = v[c], i = c+1, j = f;
    while (i <= j) {
        if (v[i] <= pivo) i++;
        else if (pivo <= v[j]) j--;
        else { // (v[i] > pivo) e (v[j] < pivo)
            swap (v[i],v[j]);
            i++;
            j--;
        }
    } // agora i == j+1
    v[c] = v[j];
    v[j] = pivo;
    return j; // retorna posição do pivô
}
```

Caso $v[i]$ seja maior que o pivô, mas $v[j]$ também é maior, retrocede na parte final do vetor

Algoritmo de Particionamento

```
int particiona(int v[], int c, int f) { // c = começo, f = fim
    int pivo = v[c], i = c+1, j = f;
    while (i <= j) {
        if (v[i] <= pivo) i++;
        else if (pivo <= v[j]) j--;
        else { // (v[i] > pivo) e (v[j] < pivo)
            swap (v[i],v[j]);
            i++;
            j--;
        }
    } // agora i == j+1
    v[c] = v[j];
    v[j] = pivo;
    return j; // retorna posição do pivô
}
```

Caso $v[i]$ seja maior que o pivô, mas $v[j]$ seja menor, temos uma situação invertida e necessário trocar os valores nas posições i e j , avançando i e retrocedendo j em seguida.

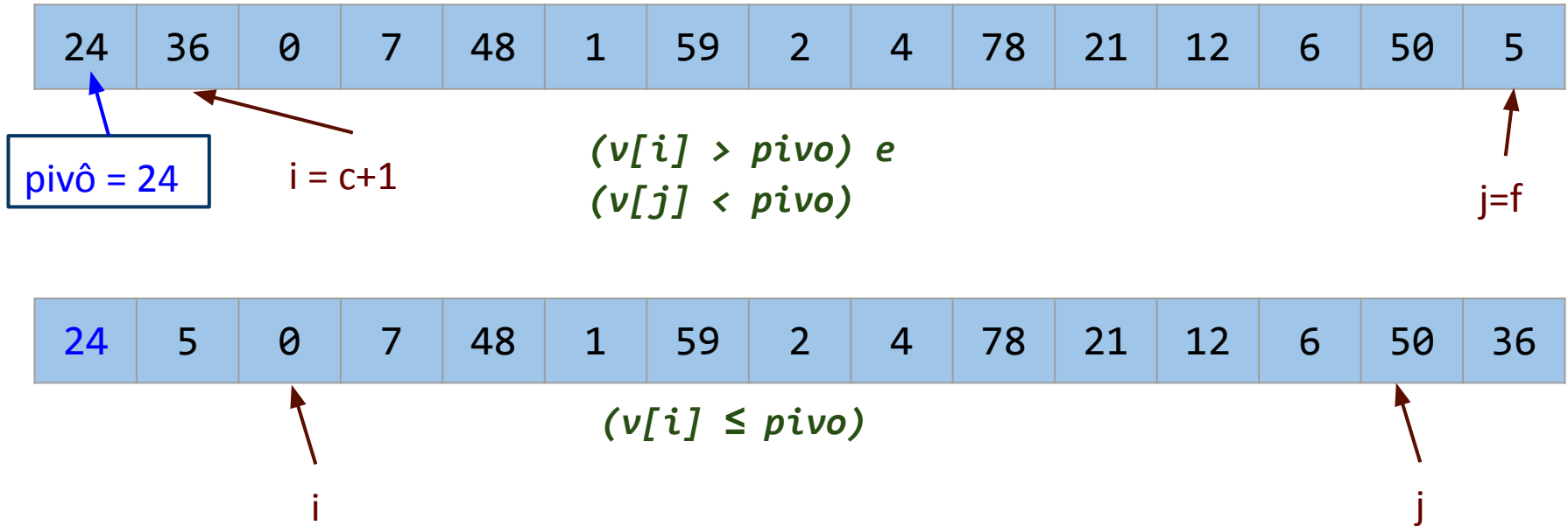
Algoritmo de Particionamento

```
int particiona(int v[], int c, int f) { // c = começo, f = fim
    int pivo = v[c], i = c+1, j = f;
    while (i <= j) {
        if (v[i] <= pivo) i++;
        else if (pivo <= v[j]) j--;
        else { // (v[i] > pivo) e (v[j] < pivo)
            swap (v[i],v[j]);
            i++;
            j--;
        }
    } // agora i == j+1
    v[c] = v[j];
    v[j] = pivo;
    return j; // retorna posição do pivô
}
```

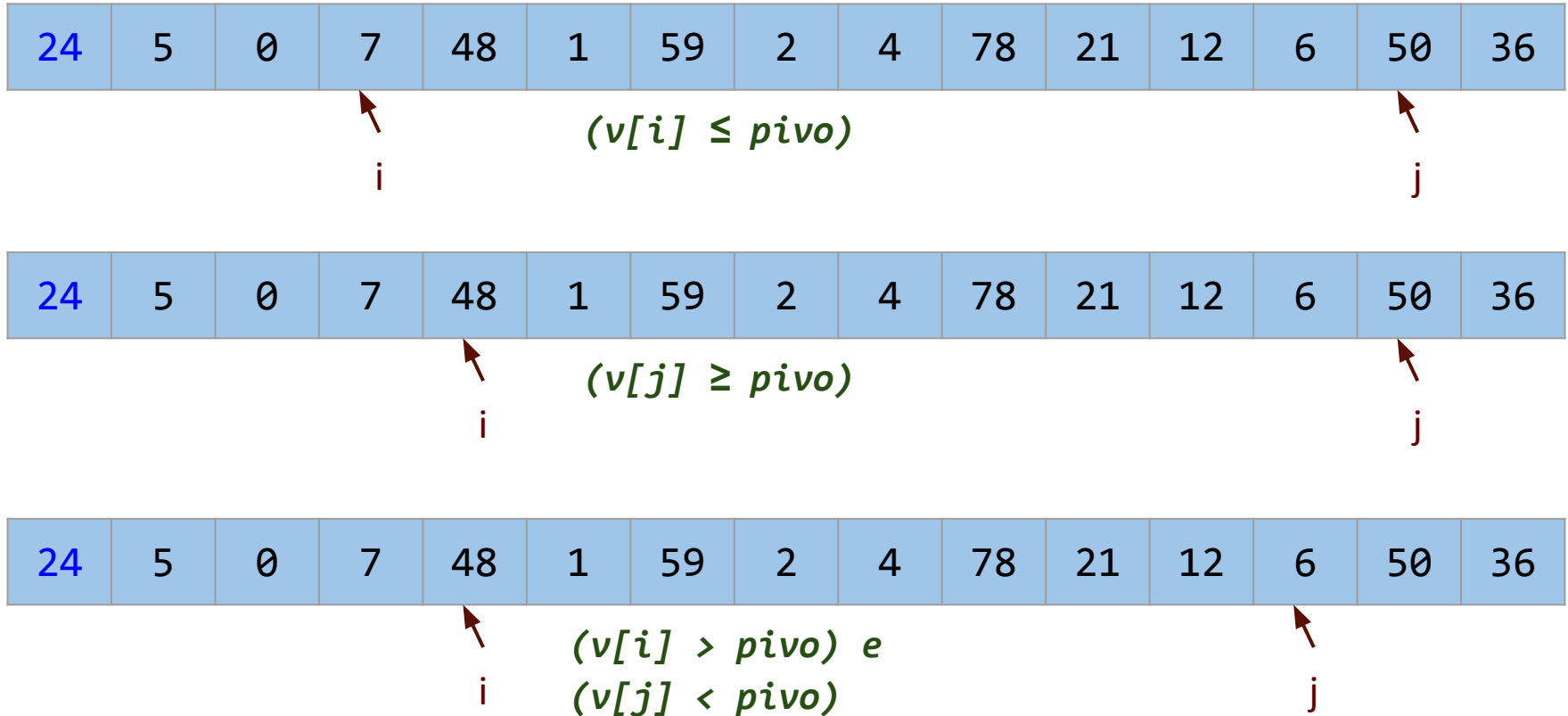
Ao término das comparações, basta trocar o pivô, na posição inicial, com aquele na posição v[j] e j será justamente a posição do pivô ao término do processo.

Aplicando Particionamento - i

Seja $V = [24, 36, 0, 7, 48, 1, 59, 2, 4, 78, 21, 12, 6, 50, 5]$,
tem-se após chamada $\text{particiona}(V, 0, 14)$:



Aplicando Particionamento - ii



Aplicando Particionamento - iii

24	5	0	7	6	1	59	2	4	78	21	12	48	50	36
----	---	---	---	---	---	----	---	---	----	----	----	----	----	----

i

$(v[i] \leq pivo)$

j

24	5	0	7	6	1	59	2	4	78	21	12	48	50	36
----	---	---	---	---	---	----	---	---	----	----	----	----	----	----

i

$(v[i] > pivo)$ e
 $(v[j] < pivo)$

j

24	5	0	7	6	1	12	2	4	78	21	59	48	50	36
----	---	---	---	---	---	----	---	---	----	----	----	----	----	----

$(v[i] \leq pivo)$

i

j

Aplicando Particionamento - iv

24	5	0	7	6	1	12	2	4	78	21	59	48	50	36
----	---	---	---	---	---	----	---	---	----	----	----	----	----	----

$(v[i] \leq \text{pivo})$

i

j

24	5	0	7	6	1	12	2	4	78	21	59	48	50	36
----	---	---	---	---	---	----	---	---	----	----	----	----	----	----

$(v[i] > \text{pivo})$ e
 $(v[j] < \text{pivo})$

i

j

24	5	0	7	6	1	12	2	4	21	78	59	48	50	36
----	---	---	---	---	---	----	---	---	----	----	----	----	----	----

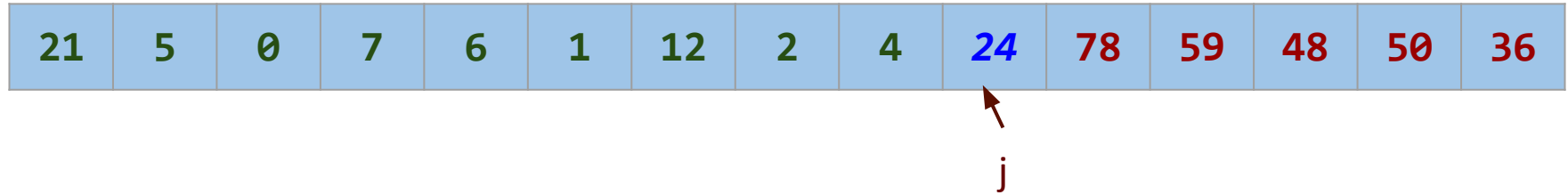
$v[c] \leftarrow v[j]$
 $v[j] \leftarrow \text{pivo}$

j

i

Aplicando Particionamento - v

21	5	0	7	6	1	12	2	4	24	78	59	48	50	36
----	---	---	---	---	---	----	---	---	----	----	----	----	----	----

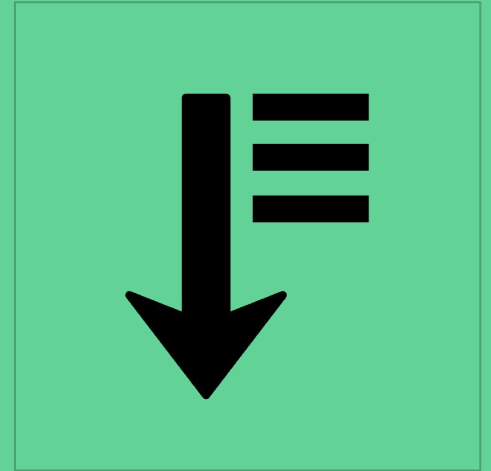


O vetor V foi particionado com sucesso, sendo que o retorno (j) indica exatamente em que posição do vetor ficou o pivô.

Observe que a ordem dos elementos nas partições não é a mesma do exemplo anterior com o mesmo vetor.

Além disso, existem vários algoritmos de particionamento que podem ser utilizados e que produzirão diferentes organizações.

Quick Sort



Quick Sort

O quick sort consiste em aplicar, recursivamente, o algoritmo de particionamento, até que as partições estejam vazias:

```
void quicksort(int a[], int pos_pivo, int fim) {  
    int pos_novo_pivo;  
    if (pos_pivo < fim) {  
        pos_novo_pivo = particiona(a, pos_pivo, fim);  
        quicksort(a, pos_pivo, pos_novo_pivo - 1);  
        quicksort(a, pos_novo_pivo + 1, fim);  
    }  
}
```



Quick Sort

O quick sort consiste em aplicar, recursivamente, o algoritmo de particionamento, até que as partições estejam vazias:

```
void quicksort(int a[], int pos_pivo, int fim) {  
    int  
    if  
        Procedimento é chamado inicialmente  
        passando o vetor, sua posição inicial (posição  
        do primeiro pivô) e sua posição final.  
        );  
    quicksort(a, pos_novo_pivo + 1, fim);  
}  
}
```



Exemplo: ordenar o vetor $a = [12, 4, 6, 25, 2, 8, 9]$

Chama quicksort (a, 0, tamanho -1)

$\text{pos_pivo} = 0$
 $\text{fim} = 6$

quicksort (a,0,6)
 $\text{pos_pivo} = 0$
 $\text{fim} = 6$

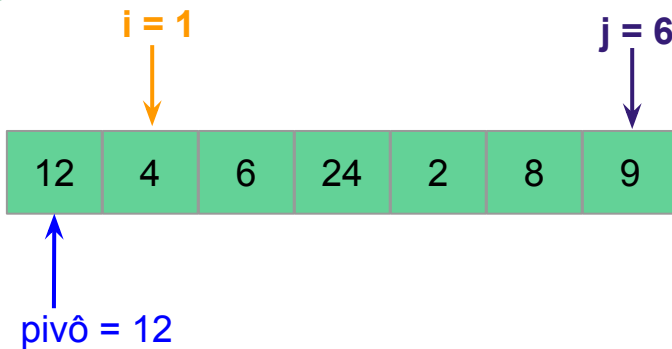
12	4	6	24	2	8	9
----	---	---	----	---	---	---

↑
pivô = 12

```
quicksort (a,0,6)  
pos_pivo = 0  
fim = 6
```

pos_pivo < fim ? Sim

Chama particiona (a, 0, 6)



Resultado do particionamento

retorna j (5)

j = 5



8	4	6	9	2	12	24
---	---	---	---	---	----	----

8	4	6	9	2	12	24
---	---	---	---	---	----	----

```
quicksort (a,0,6)  
pos_pivo = 0  
fim = 6  
pos_novo_pivo = 5
```

```
quicksort (a,0,4)  
pos_pivo = 0  
fim = 4
```

```
quicksort (a,6,6)  
pos_pivo = 6  
fim = 6
```


Chama quicksort (a, 0, 4)

pos_pivo = 0
fim = 4

pos_pivo < fim ? Sim

recursão à
esquerda

Chama particiona (a, 0, 4)

i = 1

j = 4

8	4	6	9	2	12	24
---	---	---	---	---	----	----

pivô = 8

Resultado do particionamento

$j = 3$



2	4	6	8	9	12	24
---	---	---	---	---	----	----

retorna j (3)

2	4	6	8	9	12	24
---	---	---	---	---	----	----

```
quicksort (a,0,6)  
pos_pivo = 0  
fim = 6  
pos_novo_pivo = 5
```

```
quicksort (a,0,4)  
pos_pivo = 0  
fim = 4  
pos_novo_pivo = 3
```

```
quicksort (a,6,6)  
pos_pivo = 6  
fim = 6
```

```
quicksort (a,0,2)  
pos_pivo = 0  
fim = 2
```

```
quicksort (a,4,4)  
pos_pivo = 4  
fim = 4
```

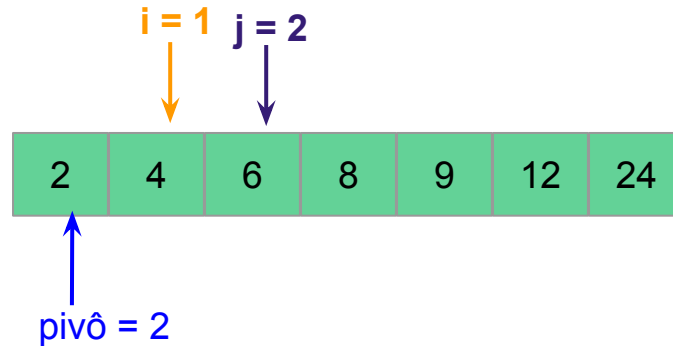
Chama quicksort (a, 0, 2)

pos_pivo = 0
fim = 2

pos_pivo < fim ? Sim

recursão à
esquerda

Chama particiona (a, pivo, fim)



Resultado do particionamento

$j = 0$



2	4	6	8	9	12	24
---	---	---	---	---	----	----

retorna j (0)

2	4	6	8	9	12	24
---	---	---	---	---	----	----

```
quicksort (a,0,6)  
pos_pivo = 0  
fim = 6  
pos_novo_pivo = 5
```

```
quicksort (a,0,4)  
pos_pivo = 0  
fim = 4  
pos_novo_pivo = 3
```

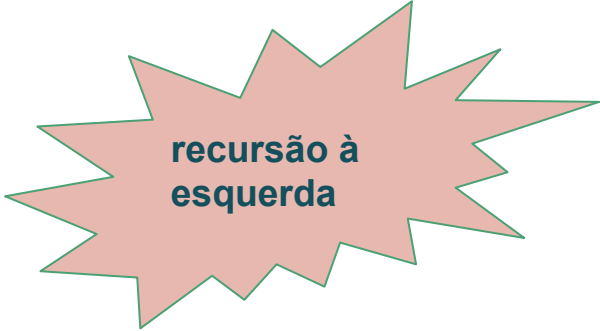
```
quicksort (a,6,6)  
pos_pivo = 6  
fim = 6
```

```
quicksort (a,0,2)  
pos_pivo = 0  
fim = 2  
pos_novo_pivo = 0
```

```
quicksort (a,4,4)  
pos_pivo = 4  
fim = 4
```

```
quicksort (a,0,-1)  
pos_pivo = 0  
fim = -1
```

```
quicksort (a,1,2)  
pos_pivo = 1  
fim = 2
```



**recursão à
esquerda**



Chama quicksort (a, 0, -1)

**pos_pivo = 0
fim = -1**

pos_pivo < fim ? Não!

2	4	6	8	9	12	24
---	---	---	---	---	----	----

```
quicksort (a,0,6)  
pos_pivo = 0  
fim = 6  
pos_novo_pivo = 5
```

```
quicksort (a,0,4)  
pos_pivo = 0  
fim = 4  
pos_novo_pivo = 3
```

```
quicksort (a,6,6)  
pos_pivo = 6  
fim = 6
```

```
quicksort (a,0,2)  
pos_pivo = 0  
fim = 2  
pos_novo_pivo = 0
```

```
quicksort (a,4,4)  
pos_pivo = 4  
fim = 4
```

```
quicksort (a,0,-1)  
pos_pivo = 0  
fim = -1
```

```
quicksort (a,1,2)  
pos_pivo = 1  
fim = 2
```

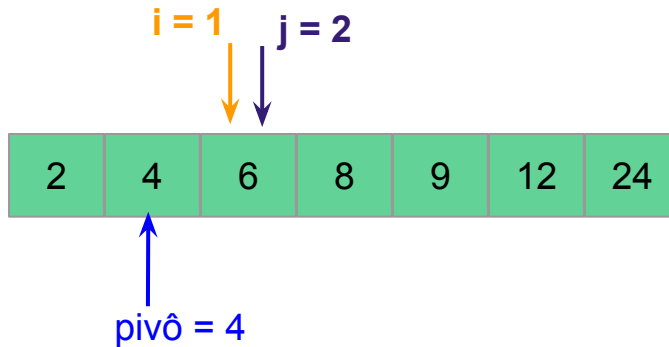

Chama quicksort (a, 1, 2)

pos_pivo = 1
fim = 2

pos_pivo < fim ? Sim

Chama particiona (a, 1, 2)

**recursão à
direita**



Resultado do particionamento

$j = 1$



retorna j (1)

2	4	6	8	9	12	24
---	---	---	---	---	----	----

2	4	6	8	9	12	24
---	---	---	---	---	----	----

quicksort (a,0,6)

quicksort (a,0,4)

quicksort (a,6,6)

quicksort (a,0,2)

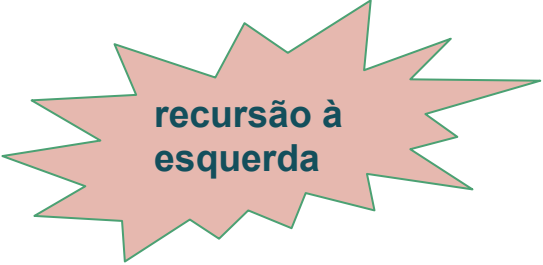
quicksort (a,4,4)

quicksort (a,0,-1)
pos_pivo = 0
fim = -1

quicksort (a,1,2)
pos_pivo = 1
fim = 2
pos_novo_pivo = 1

quicksort (a,1,0)
pos_pivo = 1
fim = 0

quicksort (a,2,2)
pivo = 2
fim = 2



**recursão à
esquerda**



Chama quicksort (a, 1, 0)

pos_pivo = 1
fim = 0

pos_pivo < fim ? Não

2	4	6	8	9	12	24
---	---	---	---	---	----	----

quicksort (a,0,6)

quicksort (a,0,4)

quicksort (a,6,6)

quicksort (a,0,2)

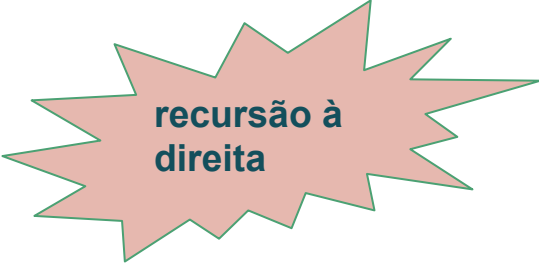
quicksort (a,4,4)

quicksort (a,0,-1)
pos_pivo = 0
fim = -1

quicksort (a,1,2)
pos_pivo = 1
fim = 2
pos_novo_pivo = 1

quicksort (a,1,0)
pos_pivo = 1
fim = 0

quicksort (a,2,2)
pivo = 2
fim = 2



**recursão à
direita**



Chama quicksort (a, 2, 2)

pos_pivo = 2
fim = 2

pos_pivo < fim ? Não

2	4	6	8	9	12	24
---	---	---	---	---	----	----

quicksort (a,0,6)

quicksort (a,0,4)

quicksort (a,6,6)

quicksort (a,0,2)

quicksort (a,4,4)

quicksort (a,0,-1)
pos_pivo = 0
fim = -1

quicksort (a,1,2)
pos_pivo = 1
fim = 2
pos_novo_pivo = 1

quicksort (a,1,0)
pos_pivo = 1
fim = 0

quicksort (a,2,2)
pivo = 2
fim = 2

2	4	6	8	9	12	24
---	---	---	---	---	----	----

quicksort (a,0,6)

quicksort (a,0,4)

quicksort (a,6,6)

quicksort (a,0,2)

quicksort (a,4,4)

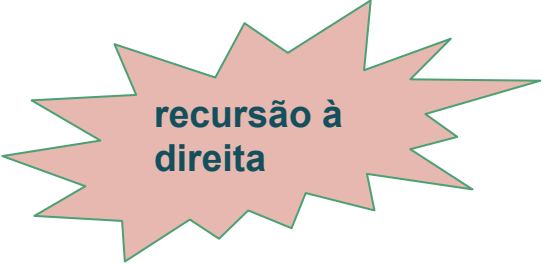


quicksort (a,0,-1)
pos_pivo = 0
fim = -1

quicksort (a,1,2)
pos_pivo = 1
fim = 2
pos_novo_pivo = 1

quicksort (a,1,0)
pos_pivo = 1
fim = 0

quicksort (a,2,2)
pivo = 2
fim = 2



**recursão à
direita**



Chama quicksort (a, 4, 4)

pos_pivo = 4
fim = 4

pos_pivo < fim ? Não

2	4	6	8	9	12	24
---	---	---	---	---	----	----

quicksort (a,0,6)

quicksort (a,0,4)

quicksort (a,6,6)

quicksort (a,0,2)

quicksort (a,4,4)

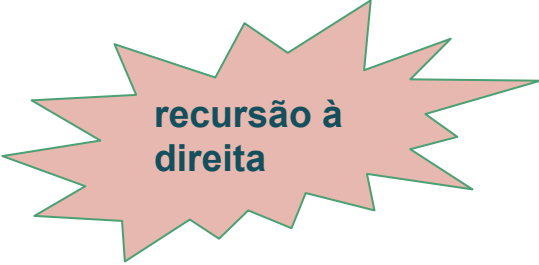
quicksort (a,0,-1)
pos_pivo = 0
fim = -1

quicksort (a,1,2)
pos_pivo = 1
fim = 2
pos_novo_pivo = 1

quicksort (a,1,0)
pos_pivo = 1
fim = 0

quicksort (a,2,2)
pivo = 2
fim = 2





**recursão à
direita**



Chama quicksort (a, 6, 6)

pos_pivo = 6
fim = 6

pos_pivo < fim ? Não

2	4	6	8	9	12	24
---	---	---	---	---	----	----

quicksort(a,0,6) ✓

quicksort(a,0,4) ✓

quicksort(a,6,6) ✓

quicksort(a,0,2) ✓

quicksort(a,4,4) ✓

quicksort(a,0,-1) ✓
pos_pivo = 0
fim = -1

quicksort(a,1,2) ✓
pos_pivo = 1
fim = 2
pos_novo_pivo = 1

quicksort(a,1,0) ✓
pos_pivo = 1
fim = 0

quicksort(a,2,2) ✓
pivo = 2
fim = 2

Análise do Quick Sort

O quick sort é um algoritmo de ordenação eficiente, bastante utilizado em problemas práticos, uma vez que, quando bem implementado, consegue ser mais rápido, na média, que seus principais competidores, o merge sort e o heap sort.

Seu pior caso é muito ruim, equivalente aos algoritmos simples, mas esse caso raramente ocorre em casos práticos.



Particionamento de Lomuto e Hoare



Algoritmos de Particionamento

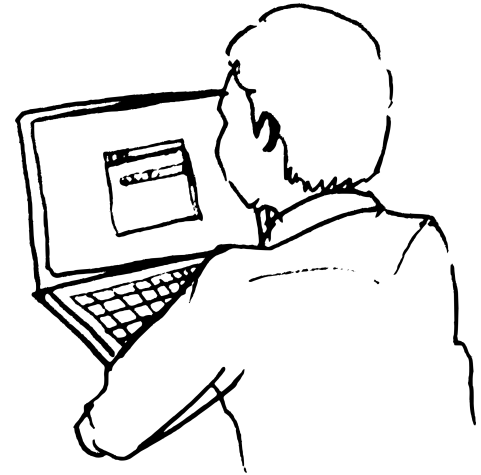
A eficiência do quick sort depende da estratégia de particionamento utilizada.

Existem várias formas de particionar um vetor, uma bastante popular é a proposta por Lomuto.

Outra popular é o particionamento de Hoare, o criador do quick sort.

Algoritmo de Lomuto

```
int particiona_L (int v[], int p, int r) { // v[p..r]
    int pivo = v[r];
    int j = p;
    int k;
    for (k = p; k < r; k++) {
        if (v[k] <= pivo) {
            swap(v[j],v[k]);
            j++;
        }
    }
    swap(v[j],v[r]);
    return j;
}
```



Aplicando o Algoritmo de Lomuto - i

Seja $V = [24, 0, 7, 48, 1, 59, 2, 21]$, tem-se após chamada $\text{particiona_L}(V, 0, 7)$:

24	0	7	48	1	59	2	21
----	---	---	----	---	----	---	----



$j = 0$



$\text{pivô} = 21$

Iniciando
o for

24	0	7	48	1	59	2	21
----	---	---	----	---	----	---	----

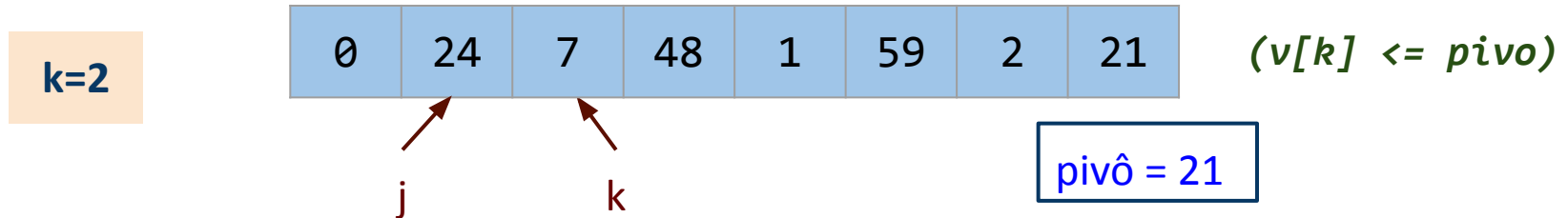
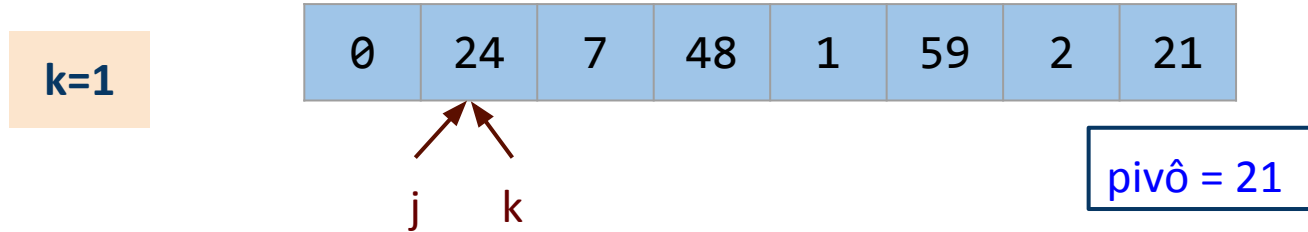
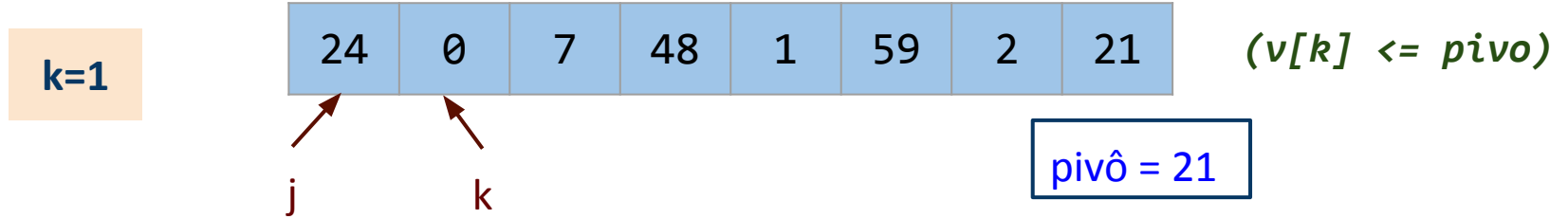


j k

$(v[k] > \text{pivô})$

$\text{pivô} = 21$

Aplicando o Algoritmo de Lomuto - ii



Aplicando o Algoritmo de Lomuto - iii

k=2

0	7	24	48	1	59	2	21
---	---	----	----	---	----	---	----

j k

pivô = 21

k=3

0	7	24	48	1	59	2	21
---	---	----	----	---	----	---	----

j k

pivô = 21

$(v[k] > \text{pivo})$

k=4

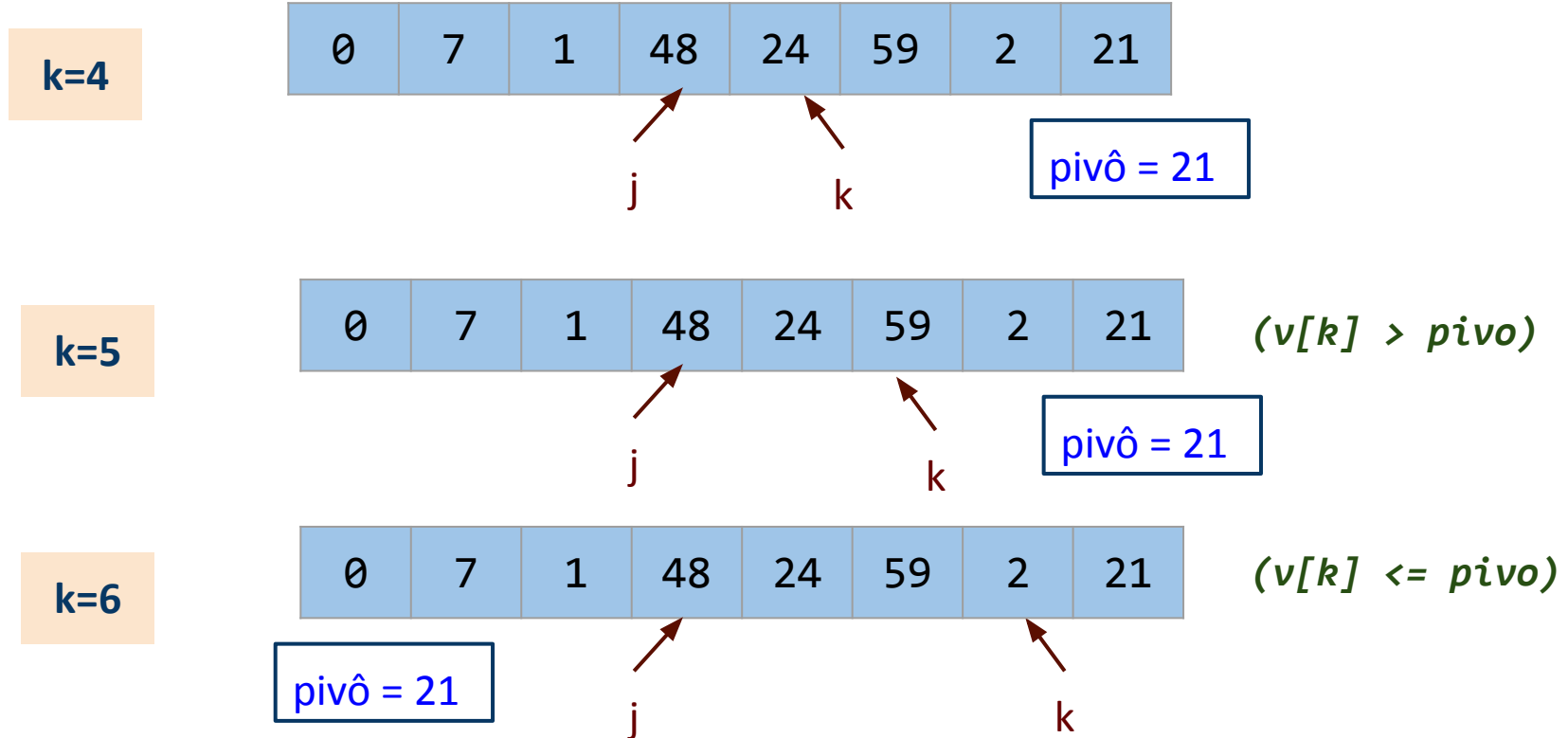
0	7	24	48	1	59	2	21
---	---	----	----	---	----	---	----

j k

pivô = 21

$(v[k] \leq \text{pivo})$

Aplicando o Algoritmo de Lomuto - iv

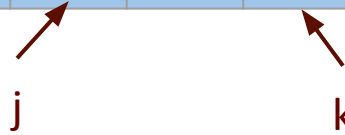


Aplicando o Algoritmo de Lomuto - v

k=6

0	7	1	2	24	59	48	21
---	---	---	---	----	----	----	----

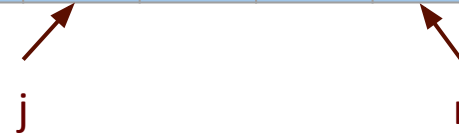
pivô = 21



encerrando
o for

0	7	1	2	24	59	48	21
---	---	---	---	----	----	----	----

pivô = 21

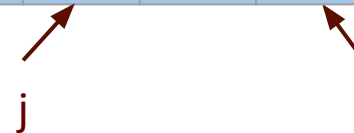


swap(v[j], v[r])

k=6

0	7	1	2	21	59	48	24
---	---	---	---	----	----	----	----

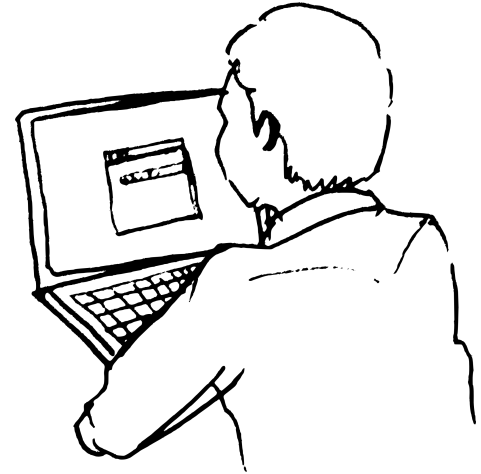
pivô = 21



return 4

Algoritmo de Hoare

```
int part_Hoare (int A[], int lo, int hi) { // A[lo..hi]
    int pivo = A[lo + (hi - lo) / 2];
    int i = lo - 1;
    int j = hi + 1;
    while (true) { // laço infinito
        do {
            i++;
        } while (A[i] < pivo);
        do {
            j--;
        } while (A[j] > pivo);
        if (i >= j) return j;
        swap(A[i],A[j]);
    } // término do while
}
```



Algoritmo de Hoare

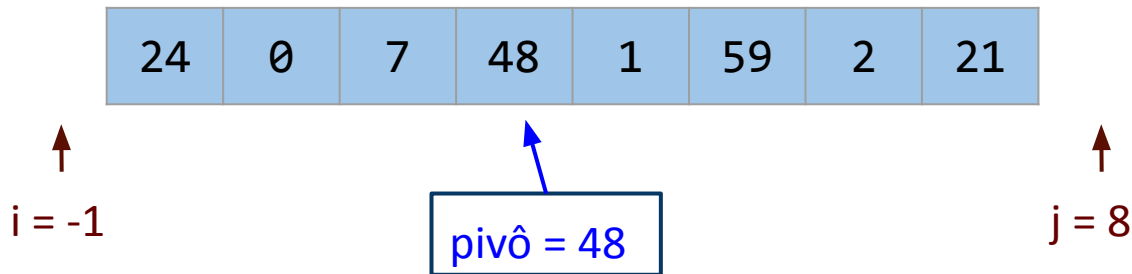
```
int part_Hoare (int A[], int lo, int hi) { // A[lo..hi]
    int pivo = A[lo + (hi - lo) / 2];
    int i = lo - 1;
    int j = hi + 1;
    while (true) { // laço infinito
        do {
            i++;
        } while (A[i] < pivo);
        do {
            j--;
        } while (A[j] > pivo);
        if (i >= j) return j;
        swap(A[i], A[j]);
    } // término do while
}
```

Término da função irá ocorrer ao retornar valor da posição do pivô.



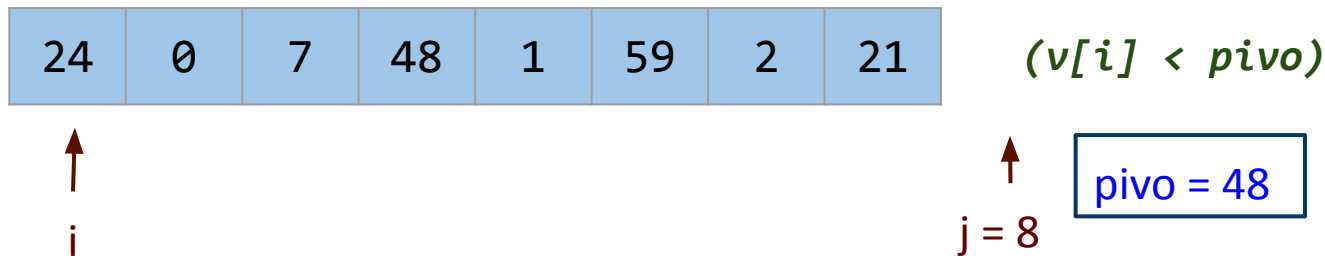
Aplicando o Algoritmo de Hoare - i

Seja $V = [24, 0, 7, 48, 1, 59, 2, 21]$, tem-se após chamada $\text{part_Hoare}(V, 0, 7)$:



Executando
o while

Primeiro
do...while



Aplicando o Algoritmo de Hoare - ii

Primeiro
do...while

24	0	7	48	1	59	2	21
----	---	---	----	---	----	---	----

↑
i

$(v[i] < pivo)$

↑
j = 8

pivo = 48

24	0	7	48	1	59	2	21
----	---	---	----	---	----	---	----

↑
i

$(v[i] < pivo)$

↑
j = 8

pivo = 48

Aplicando o Algoritmo de Hoare - iii

Primeiro
do...while

24	0	7	48	1	59	2	21
----	---	---	----	---	----	---	----

↑
i

$(v[i] \geq \text{pivo})$

↑
j = 8

pivo = 48

Segundo
do...while

24	0	7	48	1	59	2	21
----	---	---	----	---	----	---	----

↑
i

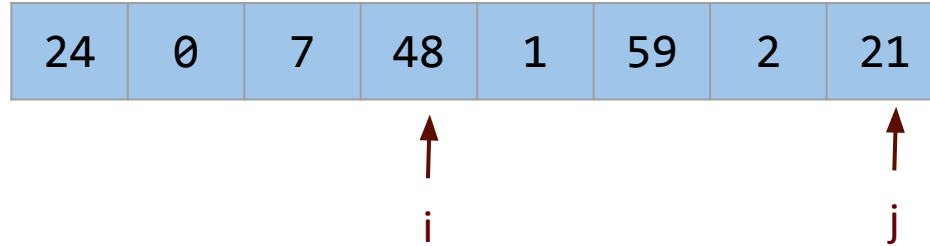
↑
j

$(v[j] \leq \text{pivo})$

pivo = 48

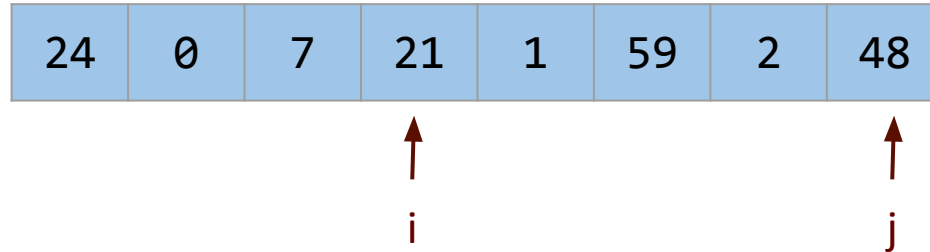
Aplicando o Algoritmo de Hoare - iv

$i < j$:
Não encerra



pivo = 48

swap(A[i],A[j])



pivo = 48

Aplicando o Algoritmo de Hoare - v

Executando
o while

Primeiro
do...while

24	0	7	21	1	59	2	48
----	---	---	----	---	----	---	----

↑
i

↑
j

$(v[i] < pivo)$

pivo = 48

24	0	7	21	1	59	2	48
----	---	---	----	---	----	---	----

↑
i

↑
j

$(v[i] \geq pivo)$

pivo = 48

Aplicando o Algoritmo de Hoare - vi

Segundo
do...while

24	0	7	21	1	59	2	48
----	---	---	----	---	----	---	----

$(v[j] \leq \text{pivo})$

pivo = 48

↑
i ↑
 j

i < j:
Não encerra

24	0	7	21	1	59	2	48
----	---	---	----	---	----	---	----

pivo = 48

↑
i ↑
 j

Aplicando o Algoritmo de Hoare - vii

`swap(A[i],A[j])`

24	0	7	21	1	2	59	48
----	---	---	----	---	---	----	----

↑ ↑
i j

pivo = 48

Executando
o while

24	0	7	21	1	2	59	48
----	---	---	----	---	---	----	----

↖ ↗
i j

$(v[i] \geq \text{pivo})$

pivo = 48

Primeiro
do...while

Aplicando o Algoritmo de Hoare - viii

Segundo
do...while

24	0	7	21	1	2	59	48
----	---	---	----	---	---	----	----

↑ ↑
j i

$(v[j] \leq pivo)$

pivo = 48

$i \geq j$:
Encerra função,
retorna 5,
valor de j

24	0	7	21	1	2	59	48
----	---	---	----	---	---	----	----

$(v[i] \geq pivo)$

pivo = 48

Quick Sort para Algoritmo de Hoare

Como o particionamento de Hoare é implícito, é necessário chamar o quicksort de forma levemente diferenciada:

```
void quicksort(int a[], int pos_pivo, int fim) {  
    int pos_novo_pivo;  
    if (pos_pivo < fim) {  
        pos_novo_pivo = particiona(a, pos_pivo, fim);  
        quicksort(a, pos_pivo, pos_novo_pivo);  
        quicksort(a, pos_novo_pivo + 1, fim);  
    }  
}
```



Quick Sort para Algoritmo de Hoare

Como o particionamento de Hoare é implícito, é necessário chamar o quicksort de forma levemente diferenciada:

```
void quicksort(int a[], int pos_pivo, int fim)
    int pos_novo_pivo;
    if (pos_pivo < fim) {
        pos_novo_pivo = particiona(a, pos_pivo, fim);
        quicksort(a, pos_pivo, pos_novo_pivo);
        quicksort(a, pos_novo_pivo + 1, fim);
    }
}
```

Chamada é feita com pos_novo_pivo, e não pos_novo_pivo-1



Sobre o Material



Sobre este material

Material produzido coletivamente, principalmente pelos seguintes professores do DCC/UFLA:

- Janderson Rodrigo de Oliveira
- Joaquim Quinteiro Uchôa
- Juliana Galvani Greggi
- Renato Ramos da Silva

Inclui contribuições de outros professores do setor de Fundamentos de Programação do DCC/UFLA.

Esta obra está licenciado com uma Licença [Creative Commons Atribuição 4.0 Internacional](https://creativecommons.org/licenses/by/4.0/).