



# INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SÃO PAULO

CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E  
DESENVOLVIMENTO DE SISTEMAS

## ESTRUTURA DE DADOS APLICADA

BUSCA/PESQUISA E ORDENAÇÃO EM VETORES

**PROFESSORA RESPONSÁVEL:**

Gislaine Cristina Micheloti Rosales

**CONTATOS:** [gimicheloti@gmail.com](mailto:gimicheloti@gmail.com) || [gislaine@ifsp.edu.br](mailto:gislaine@ifsp.edu.br)

Gislaine Cristina Micheloti Rosales 

# BUSCA/PESQUISA EM VETOR

Busca consiste na recuperação de dados armazenados em **memória principal/interna** (conjunto de dados pequeno) ou **memória secundária** (para conjunto de dados maiores). Os dados podem estar organizados em uma base de dados estruturada ou não, um arquivo etc.

Há vários métodos para realizar busca por dados, dependendo da forma como eles estão organizados e de características, como:

- Se os dados são estruturados (vetor, lista, árvore) ou não;
- Se os dados estão ordenados ou não;
- Se há valores duplicados ou não (chave/identificador único).

# MÉTODOS DE BUSCA

## Exemplos de Métodos de Busca:

- Busca Sequência/Linear;
- Busca Ordenada;
- Busca Binária;
- Árvore de Busca Binária (já vimos);
- Tabelas Hash
- Etc.

# BUSCA LINEAR/SEQUENCIAL

Percorre todo o vetor até encontrar o elemento procurado.

```
62 int buscaLinearNome(struct aluno *vet, int n, char* valor){  
63     int i;  
64     for(i = 0; i<n; i++){  
65         if(strcmp(valor, vet[i].nome)==0)  
66             return i; //elemento encontrado  
67     }  
68     return -1; //elemento não encontrado  
69 }
```

# BUSCA ORDENADA

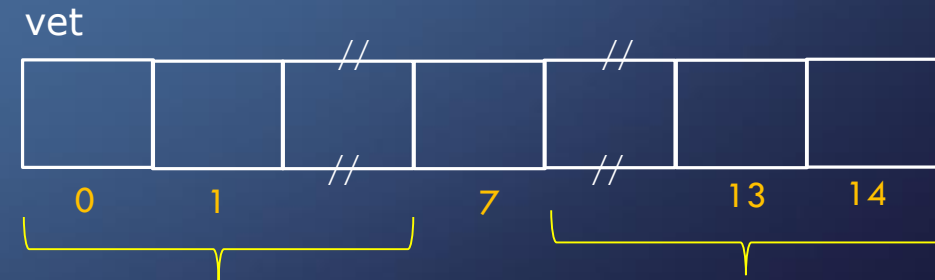
Este algoritmo é semelhante ao de Busca Linear, porém como o vetor está ordenado, posso ter **ganho no custo da busca**, caso não tenha que percorrer até a última posição para encontrar o elemento desejado.

```
23 int buscaOrdenada(int *vet, int n, int valor){
24     int i;
25     for(i = 0; i < n; i++){
26         if(valor == vet[i])
27             return i; //elemento encontrado
28         else
29             if(valor < vet[i])
30                 return -1; //para a busca
31     }
32     return -1; //elemento não encontrado
33 }
```

# BUSCA BINÁRIA

Este algoritmo considera um **vetor ordenado** e consiste em **encontrar o "meio"** do vetor e identificar em qual metade (da esquerda ou da direita) deve realizar a busca pelo elemento desejado. Essa lógica é aplicada sucessivamente até encontrar o elemento pesquisado, ou chegar ao fim sem encontrá-lo.

```
36 int buscaBinaria(int *vet, int n, int valor){
37     int i, inicio, meio, fim;
38     inicio = 0; //início do vetor
39     fim = n-1; //fim do vetor
40     while(inicio <= fim){
41         meio = (inicio + fim)/2; //meio do vetor
42         if(valor < vet[meio])
43             fim = meio-1; //reduz busca à metade da esquerda
44         else
45             if(valor > vet[meio])
46                 inicio = meio+1; //reduz a busca à metade da direita
47             else
48                 return meio; //valor está no meio
49     }
50     return -1; //elemento não encontrado
51 }
```



**inicio** = 0  
**fim** = 14

**meio** = 7 (será  
inicio ou fim)

# Ordenação de Dados...

# ORDENAÇÃO DE DADOS

Ordenar dados consiste em **organizá-los em determinada ordem** com o objetivo de possibilitar **busca mais eficiente e com menor custo**.

## Tipos mais comuns de ordenação:

- Numérica: 1, 2, 3, ..., n (ou decrescente)
- Lexicográfica/alfabética: Ana, Beatriz, Gustavo, Ivo, Marisa, Pedro, Washington (ou inversa Z...A)

## Classificação métodos de ordenação:

- **Ordenação Interna**
  - Arquivo que sofrerá ordenação é levado para Memória Principal (MP)
  - Registros são acessados imediatamente
- **Ordenação Externa**
  - Arquivo que sofrerá ordenação não cabe na MP
  - Registros são acessados sequencialmente ou blocos



# EXEMPLOS DE MÉTODOS DE ORDENAÇÃO EM VETORES

## Métodos Simples:

- Insertion Sort
- Selection Sort
- Bubble Sort
- Comb Sort

## Métodos Sofisticados:

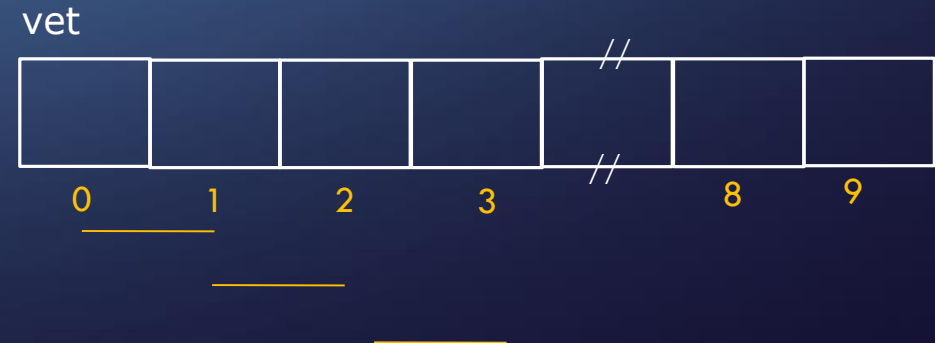
- Quick Sort
- Merge Sort
- Heap Sort
- Shell Sort
- Radix Sort
- Gnome Sort

# ORDENAÇÃO POR FLUTUAÇÃO/BOLHA - BUBBLESORT

BubbleSort compara dois **elementos vizinhos** e, caso estejam desordenados, os **inverte (troca-os de posição)**. A comparação é feita até que não haja mais nenhuma troca a ser realizada.

## Performance:

- Melhor Caso:  $O(n)$  – Dados Ordenados = 9
- Pior Caso:  $O(n^2)$  – Dados Invertidos = 81
- Eficiente para poucos dados



# ORDENAÇÃO POR BOLHA - BUBBLESORT

```
23 void bubbleSort(int *v , int n){
24     int i, continua, aux;
25     do{
26         continua = 0;
27         for(i = 0; i < n-1; i++){
28             if (v[i] > v[i+1]){
29                 aux = v[i];
30                 v[i] = v[i+1];
31                 v[i+1] = aux;
32                 continua = 1;
33             }
34         }
35     }while(continua);
36 }
```

```
38 void bubbleSortOtimizado(int *v , int n){
39     int i, continua, aux, fim = N;
40     do{
41         continua = 0;
42         for(i = 0; i < fim-1; i++){
43             if (v[i] > v[i+1]){
44                 aux = v[i];
45                 v[i] = v[i+1];
46                 v[i+1] = aux;
47                 continua = i;
48             }
49         }
50         fim--;
51     }while(continua != 0);
52 }
```

# ORDENAÇÃO POR INSERÇÃO - INSERTIONSORT

No InsertionSort, a partir de um determinado elemento (iniciando pelo 2º), verifica quais elementos anteriores são maiores que o atual e “abre espaço” para que este elemento ocupe seu lugar corretamente. “Abrir espaço”, significa deslocar todos os elementos maiores que o atual para inseri-lo na posição correta.

## Performance:

- Melhor Caso:  $O(n)$  – dados ordenados = 9
- Pior Caso:  $O(n^2) = 81$
- Eficiente para poucos dados

6 5 3 1 8 7 2 4

vet

5	2	-3	9	12	14	23
0	1	2	3	4	5	6


# ORDENAÇÃO POR INSERÇÃO - INSERTIONSORT

```
54 void insertionSort(int *v, int n){  
55     int i, j, atual;  
56     for(i = 1; i < n; i++){  
57         atual = v[i];  
58         for(j = i; (j > 0) && (atual < v[j - 1]); j--){  
59             v[j] = v[j - 1];  
60             v[j] = atual;  
61         }  
62     }
```

vet



5	2	-3	9	12	14	23
0	1	2	3	4	5	6




5	2	-3	9	12	14	23
0	1	2	3	4	5	6




2	5	-3	9	12	14	23
0	1	2	3	4	5	6

```
122 void insertionSortMatricula(struct aluno *v, int n){  
123     int i, j;  
124     struct aluno aux;  
125     for(i = 1; i < n; i++){  
126         aux = v[i];  
127         for(j=i; (j>0) && (aux.matricula<v[j-1].matricula);j--){  
128             v[j] = v[j - 1];  
129             v[j] = aux;  
130         }  
131     }
```



-3	2	5	9	12	14	23
0	1	2	3	4	5	6



-3	2	5	9	12	14	23
0	1	2	3	4	5	6



-3	2	5	9	12	14	23
0	1	2	3	4	5	6



-3	2	5	9	12	14	23
0	1	2	3	4	5	6

# ORDENAÇÃO POR SELEÇÃO - SELECTIONSORT

O método de SelectionSort busca o menor valor de um conjunto de dados e o desloca para a primeira posição, busca o segundo menor valor e o desloca para a segunda posição e assim sucessivamente. Repete-se a lógica para cada elemento do conjunto até a ordenação de todos os dados.

## Performance:

- Melhor Caso:  $O(n^2)$  – dados ordenados = 81
- Pior Caso:  $O(n^2) = 81$
- Pouco eficiente se comparado aos demais métodos vistos

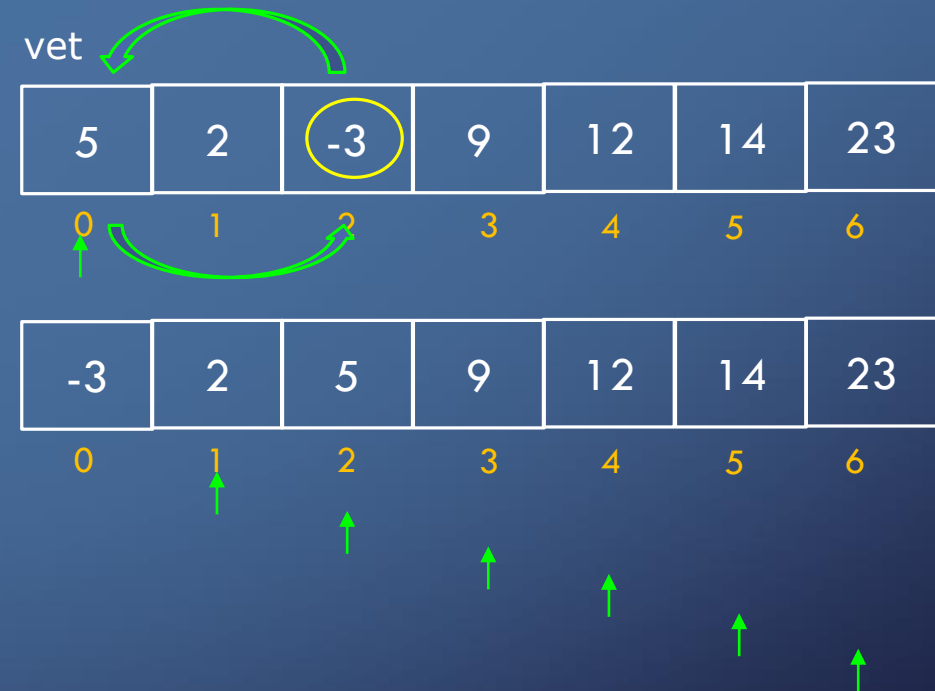
vet

5	2	-3	9	12	14	23
0	1	2	3	4	5	6

8
5
2
6
9
3
1
4
0
7

# ORDENAÇÃO POR SELEÇÃO - SELECTIONSORT

```
64 void selectionSort(int *v, int n){  
65     int i, j, posMenor, troca;  
66     for(i = 0; i < n-1; i++){  
67         posMenor = i;  
68         for(j = i+1; j < n; j++){  
69             if(v[j] < v[posMenor]){  
70                 posMenor = j;  
71             }  
72         if(i != posMenor){  
73             troca = v[i];  
74             v[i] = v[posMenor];  
75             v[posMenor] = troca;  
76         }  
77     }  
78 }
```



# ORDENAÇÃO POR MISTURA – MERGESORT – DIVIDIR E CONQUISTAR

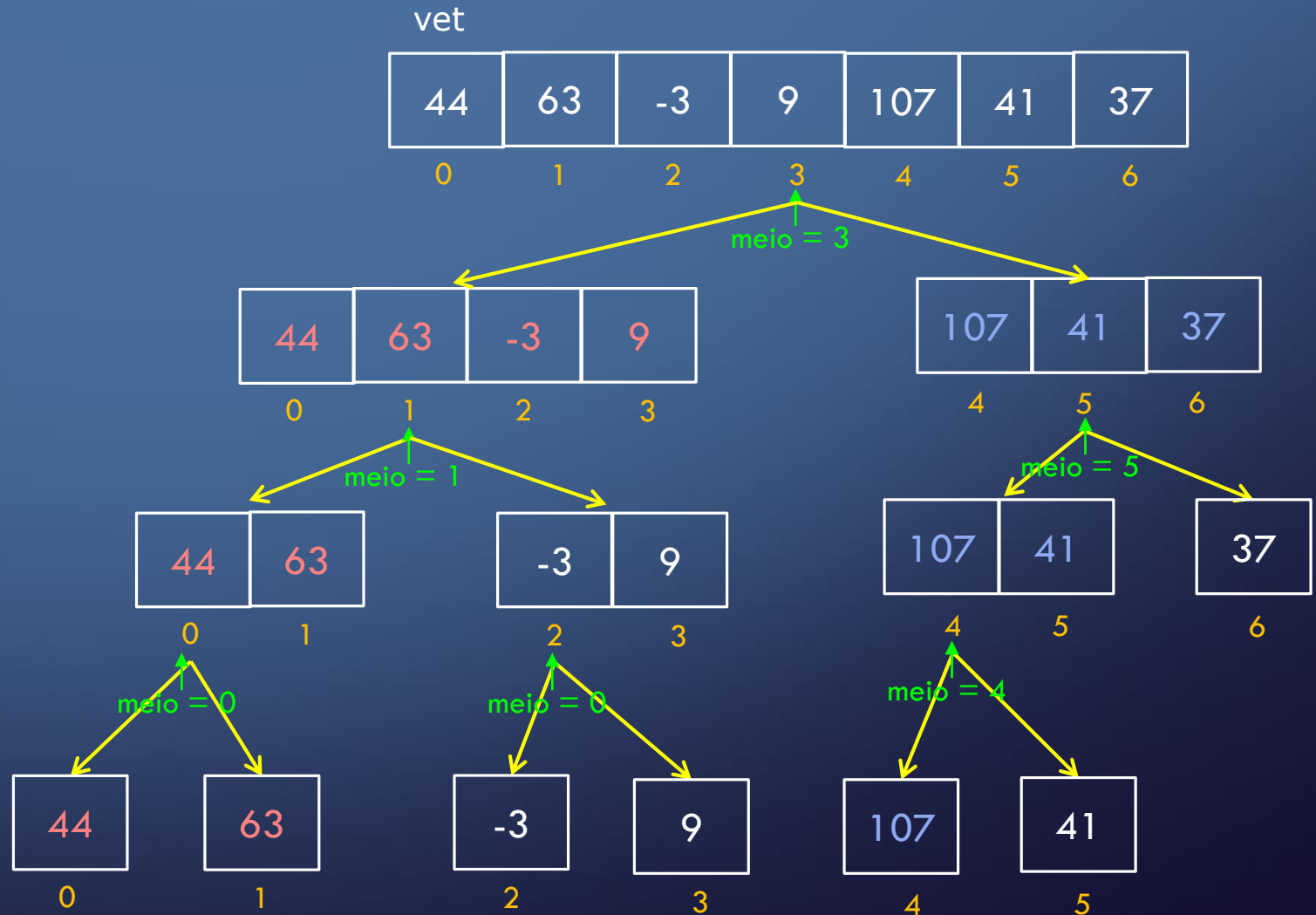
MergeSort é implementado como algoritmo recursivo em que um conjunto original, desordenado, é dividido em subconjuntos até obter subconjuntos com 1 elemento. Compara-se dois subconjuntos e decide-se qual deles ocorre primeiro na estrutura final de classificação.

## Performance:

- Melhor Caso:  $O(n \log n)$
- Pior Caso:  $O(n \log n)$
- Usa vetor auxiliar durante a ordenação (usa mais memória que os demais algoritmos de ordenação).



# ORDENAÇÃO POR MISTURA – MERGESORT



# ORDENAÇÃO POR MISTURA – MERGESORT

fim = 7 (tamanho do vetor) = n

```
110 void mergeSort(int *v, int inicio, int fim){  
111     int meio; //para achar o meio do vetor  
112     if(inicio < fim){  
113         meio = floor((inicio+fim)/2);  
114         mergeSort(v, inicio, meio);  
115         mergeSort(v, meio+1, fim);  
116         merge(v, inicio, meio, fim);  
117     }  
118 }
```



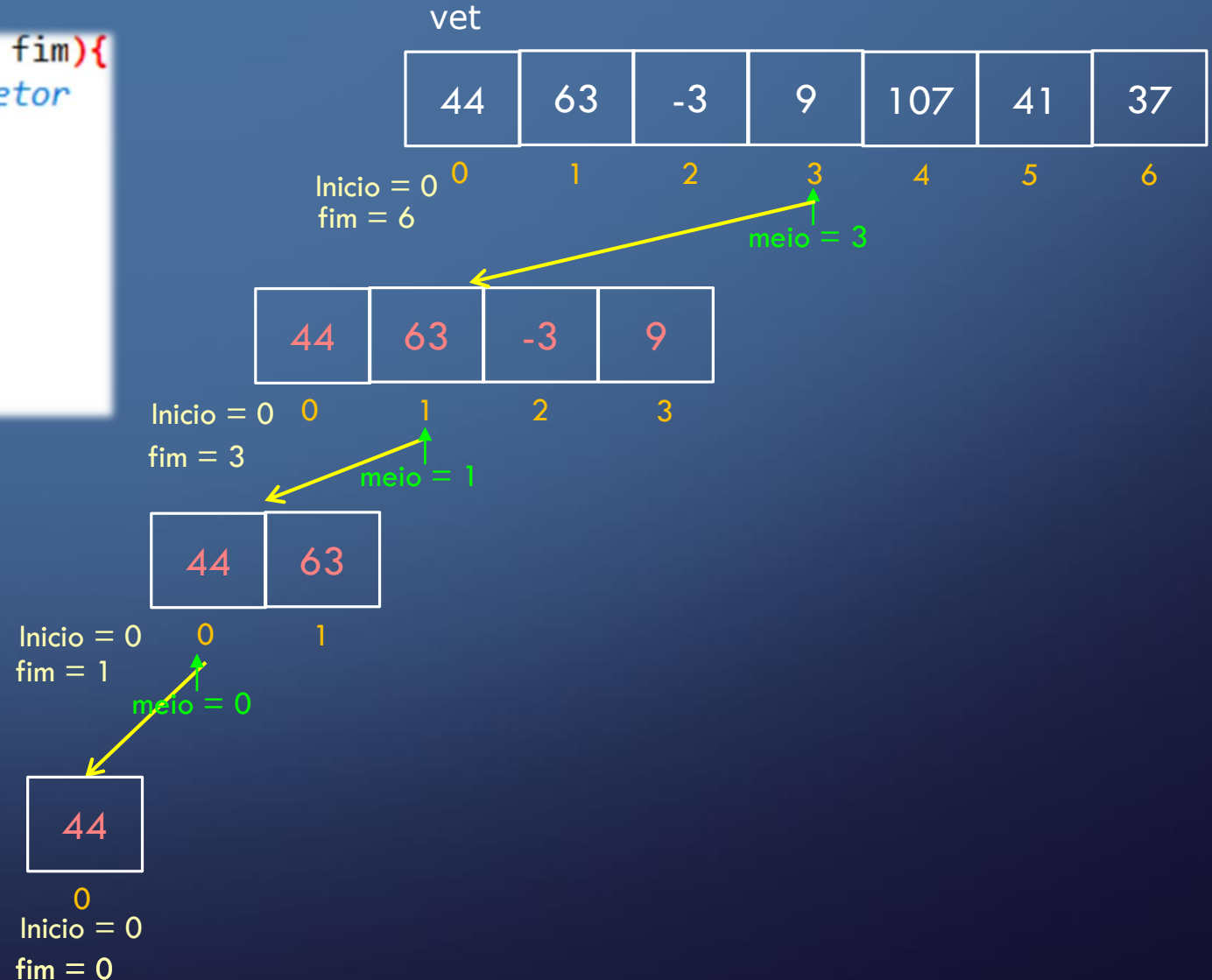
# ORDENAÇÃO POR MISTURA – MERGESORT

```
110 void mergeSort(int *v, int inicio, int fim){
111     int meio; //para achar o meio do vetor
112     if(inicio < fim){
113         meio = floor((inicio+fim)/2);
114         mergeSort(v, inicio, meio);
115         mergeSort(v, meio+1, fim);
116         merge(v, inicio, meio, fim);
117     }
118 }
```



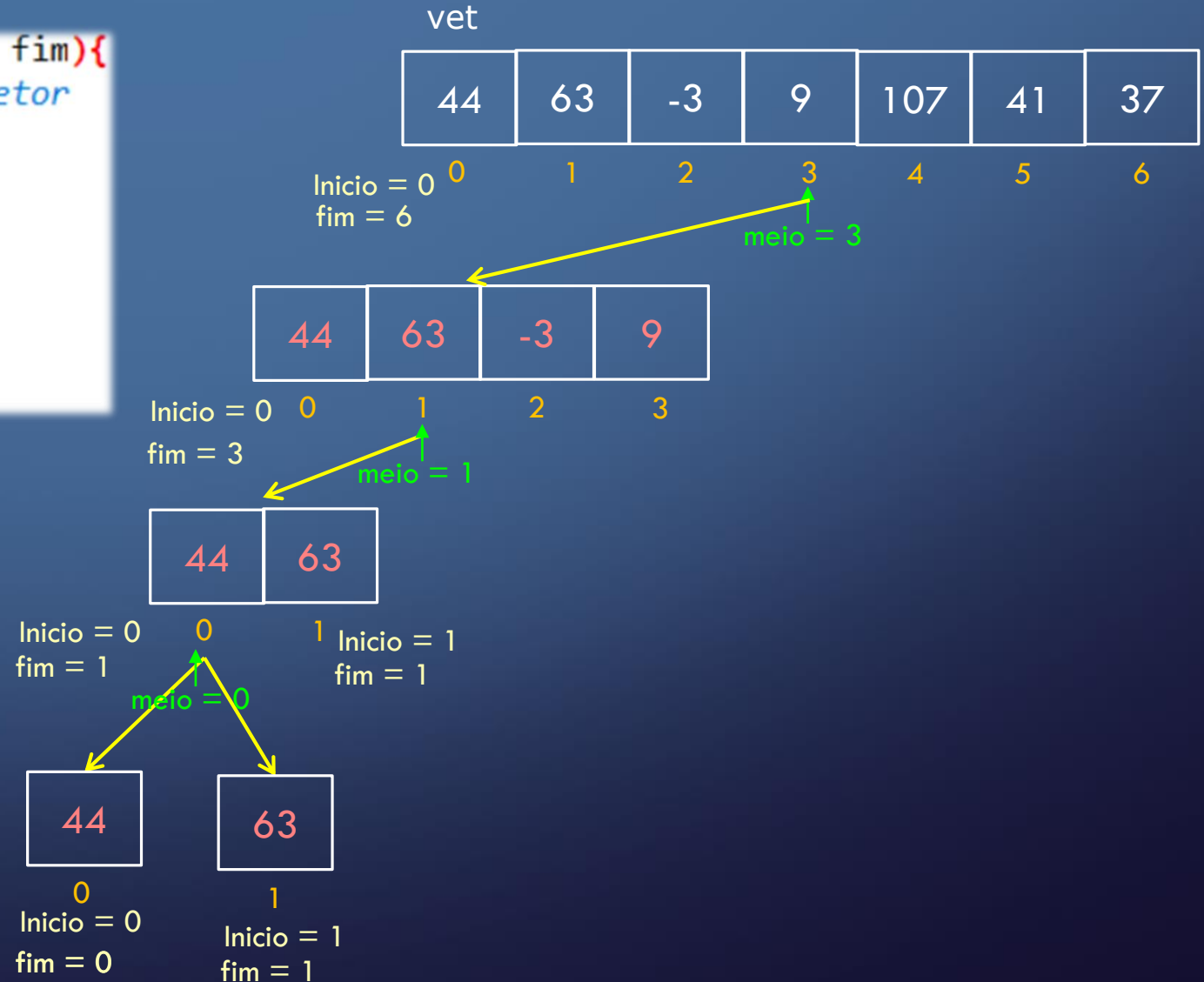
# ORDENAÇÃO POR MISTURA – MERGESORT

```
110 void mergeSort(int *v, int inicio, int fim){
111     int meio; //para achar o meio do vetor
112     if(inicio < fim){
113         meio = floor((inicio+fim)/2);
114         mergeSort(v, inicio, meio);
115         mergeSort(v, meio+1, fim);
116         merge(v, inicio, meio, fim);
117     }
118 }
```



# ORDENAÇÃO POR MISTURA – MERGESORT

```
110 void mergeSort(int *v, int inicio, int fim){
111     int meio; //para achar o meio do vetor
112     if(inicio < fim){
113         meio = floor((inicio+fim)/2);
114         mergeSort(v, inicio, meio);
115         mergeSort(v, meio+1, fim);
116         merge(v, inicio, meio, fim);
117     }
118 }
```



# ORDENAÇÃO POR MISTURA – MERGESORT

```
110 void mergeSort(int *v, int inicio, int fim){
111     int meio; //para achar o meio do vetor
112     if(inicio < fim){
113         meio = floor((inicio+fim)/2);
114         mergeSort(v, inicio, meio);
115         mergeSort(v, meio+1, fim);
116         merge(v, inicio, meio, fim);
117     }
118 }
```

void merge(int \*v, int inicio, int meio, int fim){

inicio = 0; meio = 0; fim = 1

vet

44	63	-3	9	107	41	37
----	----	----	---	-----	----	----

0 1 2 3 4 5 6

vAux

44	63
----	----

0 1

vet

44	63	-3	9	107	41	37
----	----	----	---	-----	----	----

Inicio = 0  
fim = 6

meio = 3

44	63	-3	9
----	----	----	---

Inicio = 0  
fim = 3

meio = 1

44	63
----	----

Inicio = 0  
fim = 1

meio = 0

44
----

0  
Inicio = 0  
fim = 0

Inicio = 1  
fim = 1

63
----

1  
Inicio = 1  
fim = 1

# ORDENAÇÃO POR MISTURA – MERGESORT

```
110 void mergeSort(int *v, int inicio, int fim){
111     int meio; //para achar o meio do vetor
112     if(inicio < fim){
113         meio = floor((inicio+fim)/2);
114         mergeSort(v, inicio, meio);
115         mergeSort(v, meio+1, fim);
116         merge(v, inicio, meio, fim);
117     }
118 }
```

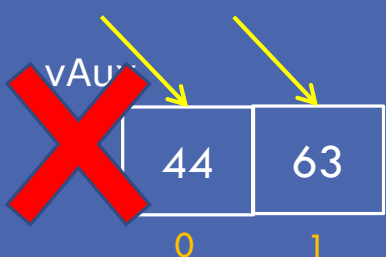
void merge(int \*v, int inicio, int meio, int fim){

inicio = 0; meio = 0; fim = 1

vet

44	63	-3	9	107	41	37
----	----	----	---	-----	----	----

0 1 2 3 4 5 6



vet

44	63	-3	9	107	41	37
----	----	----	---	-----	----	----

Início = 0 0 1 2 3 4 5 6  
fim = 6

meio = 3

44	63	-3	9
----	----	----	---

Início = 0 0 1 2 3  
fim = 3

meio = 1

44	63
----	----

Início = 0 0 1 Início = 1  
fim = 1 fim = 1

meio = 0

44	63
----	----

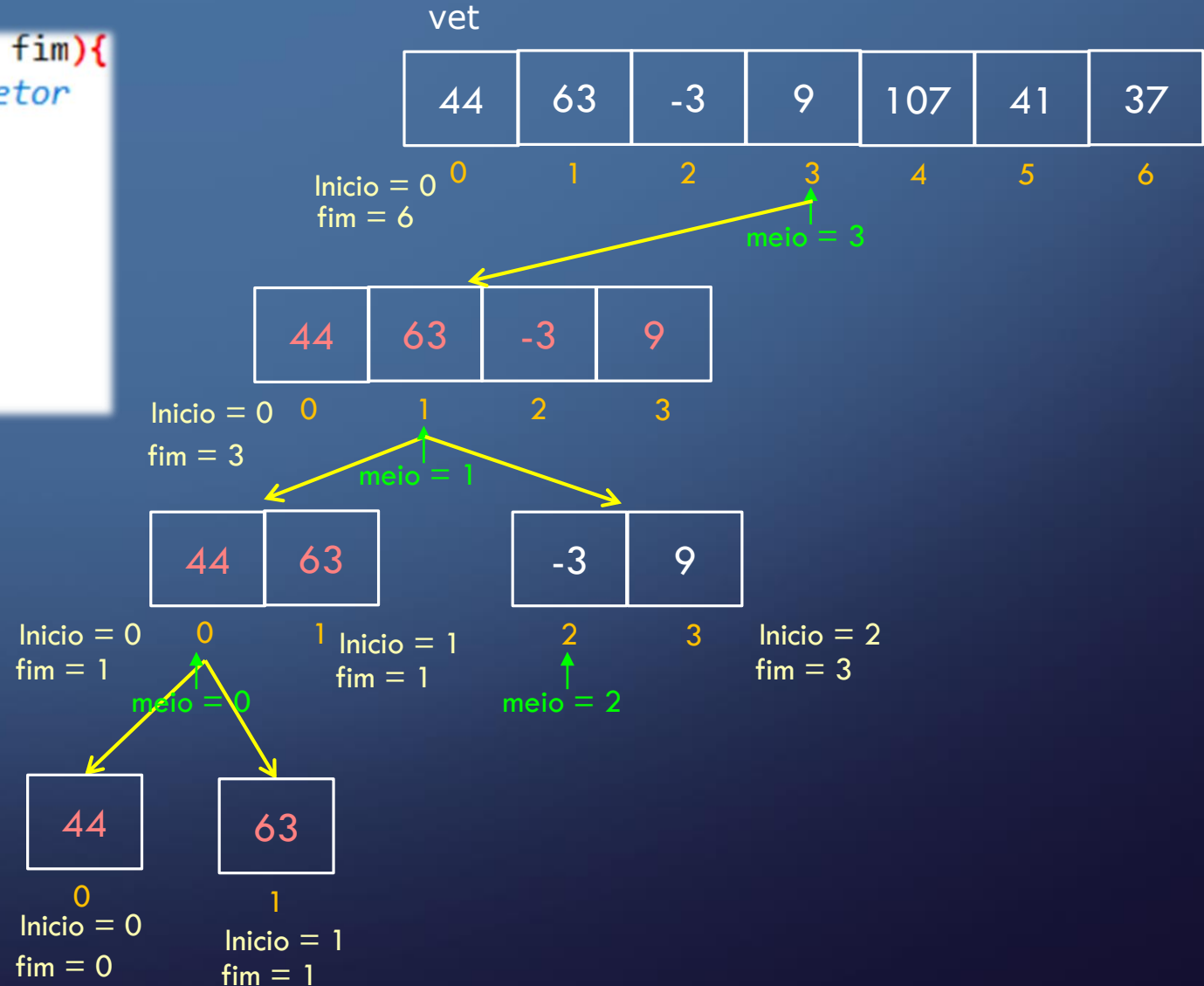
0  
Início = 0  
fim = 0

1  
Início = 1  
fim = 1



# ORDENAÇÃO POR MISTURA – MERGESORT

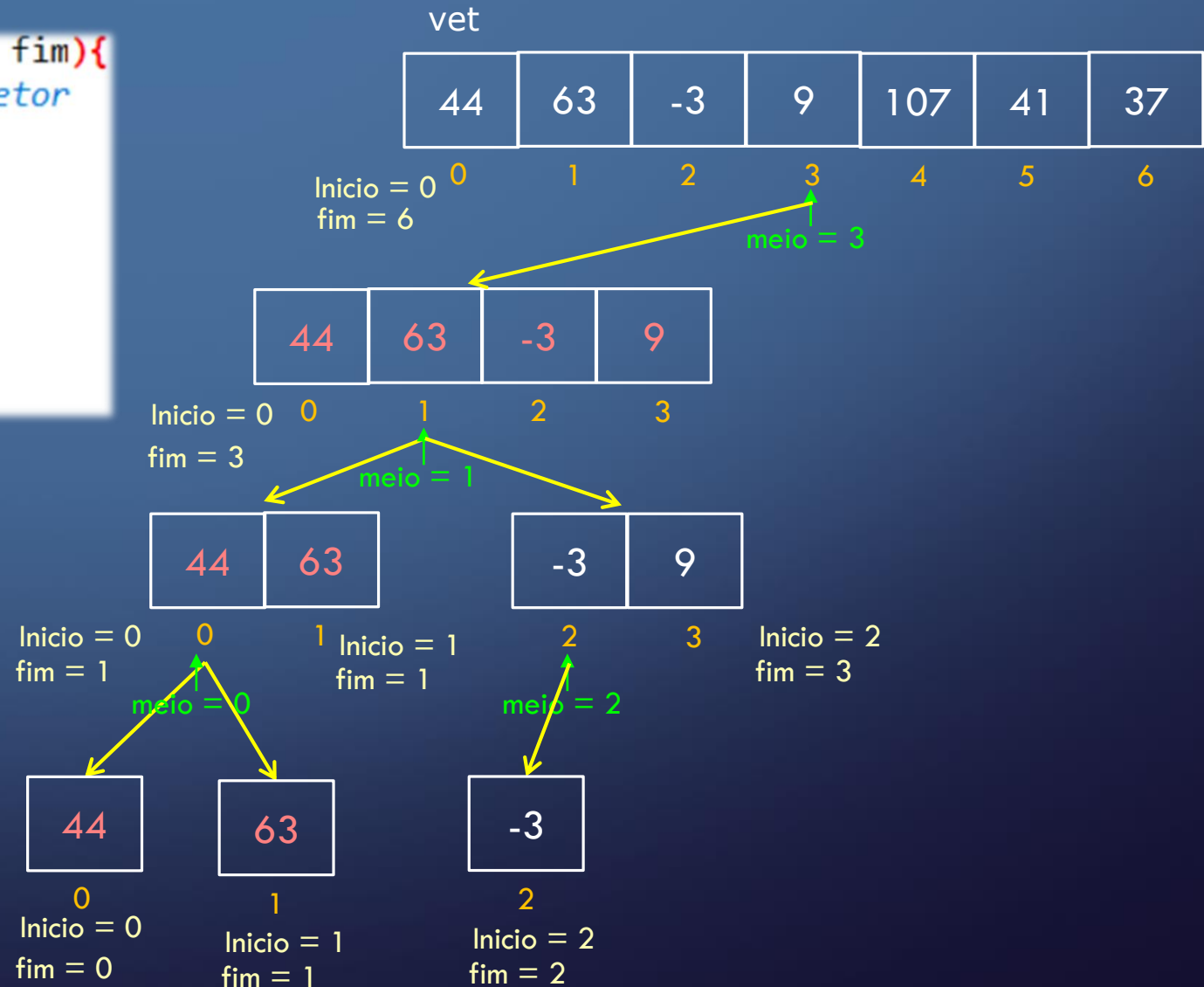
```
110 void mergeSort(int *v, int inicio, int fim){
111     int meio; //para achar o meio do vetor
112     if(inicio < fim){
113         meio = floor((inicio+fim)/2);
114         mergeSort(v, inicio, meio);
115         mergeSort(v, meio+1, fim);
116         merge(v, inicio, meio, fim);
117     }
118 }
```





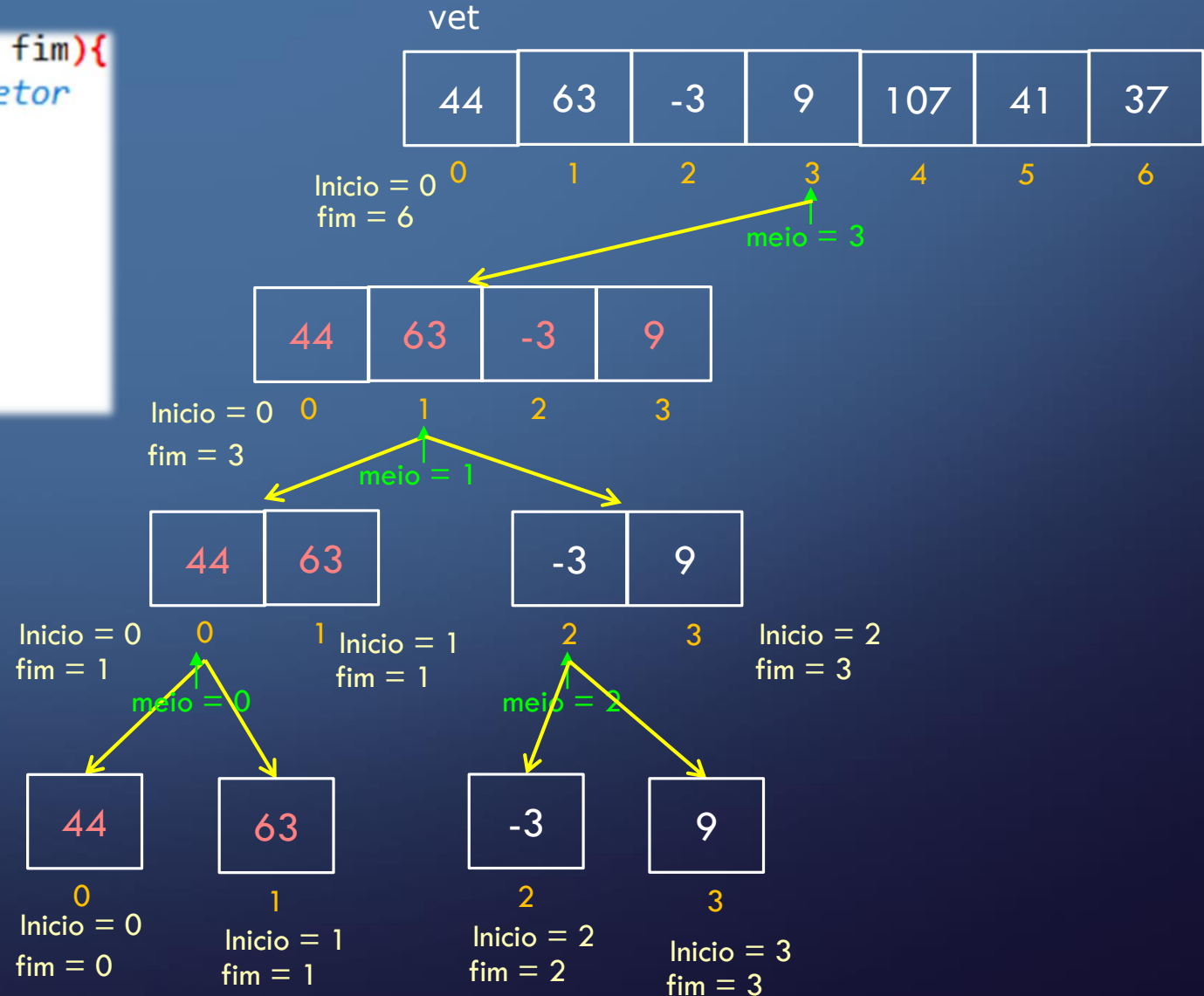
# ORDENAÇÃO POR MISTURA – MERGESORT

```
110 void mergeSort(int *v, int inicio, int fim){
111     int meio; //para achar o meio do vetor
112     if(inicio < fim){
113         meio = floor((inicio+fim)/2);
114         mergeSort(v, inicio, meio);
115         mergeSort(v, meio+1, fim);
116         merge(v, inicio, meio, fim);
117     }
118 }
```



# ORDENAÇÃO POR MISTURA – MERGESORT

```
110 void mergeSort(int *v, int inicio, int fim){
111     int meio; //para achar o meio do vetor
112     if(inicio < fim){
113         meio = floor((inicio+fim)/2);
114         mergeSort(v, inicio, meio);
115         mergeSort(v, meio+1, fim);
116         merge(v, inicio, meio, fim);
117     }
118 }
```



# ORDENAÇÃO POR MISTURA – MERGESORT

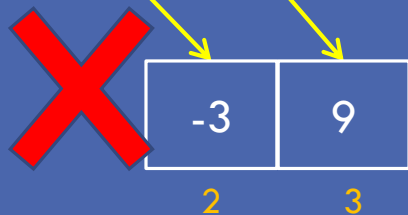
```
110 void mergeSort(int *v, int inicio, int fim){
111     int meio; //para achar o meio do vetor
112     if(inicio < fim){
113         meio = floor((inicio+fim)/2);
114         mergeSort(v, inicio, meio);
115         mergeSort(v, meio+1, fim);
116         merge(v, inicio, meio, fim);
117     }
118 }
```

```
void merge(int *v, int inicio, int meio, int fim){
    inicio = 2; meio = 2; fim = 3
```

vet

44	63	-3	9	107	41	37
0	1	2	3	4	5	6

vAux



vet

44	63	-3	9	107	41	37
----	----	----	---	-----	----	----

Inicio = 0  
fim = 6

meio = 3

44	63	-3	9
----	----	----	---

Inicio = 0  
fim = 3

meio = 1

44	63
----	----

-3	9
----	---

Inicio = 0  
fim = 1

meio = 0

44
----

Inicio = 0  
fim = 0

63
----

Inicio = 1  
fim = 1

Inicio = 1  
fim = 1

Inicio = 2  
fim = 2

meio = 2

-3
----

Inicio = 2  
fim = 2

9
---

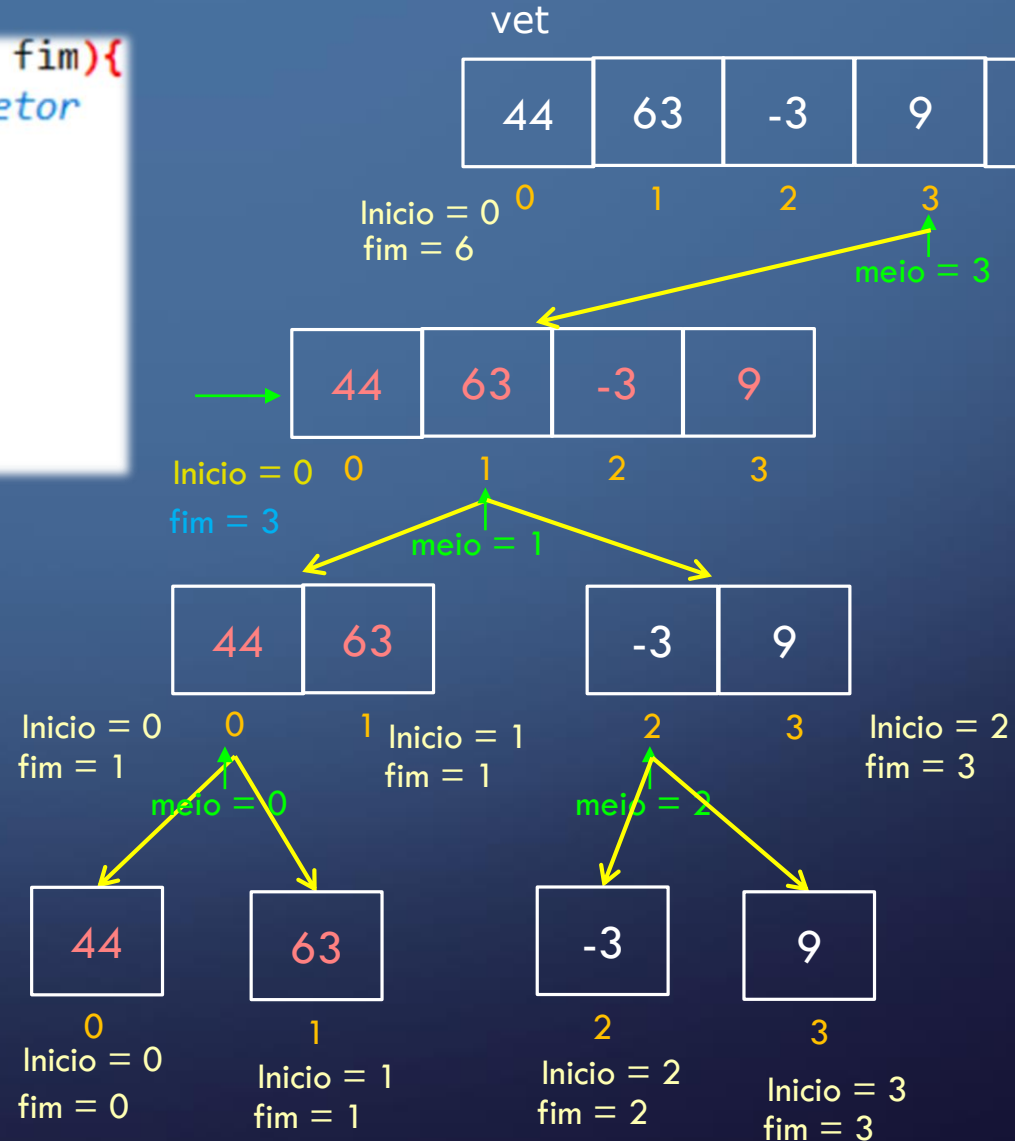
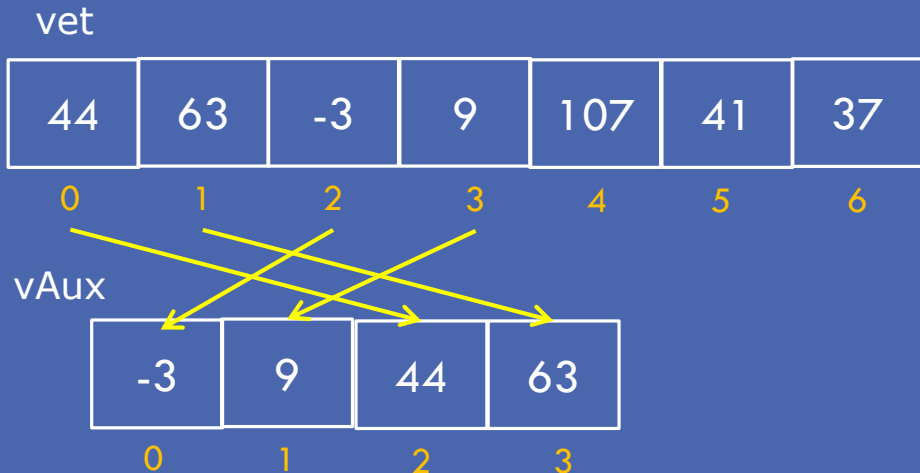
Inicio = 3  
fim = 3

Inicio = 2  
fim = 3

# ORDENAÇÃO POR MISTURA – MERGESORT

```
110 void mergeSort(int *v, int inicio, int fim){
111     int meio; //para achar o meio do vetor
112     if(inicio < fim){
113         meio = floor((inicio+fim)/2);
114         mergeSort(v, inicio, meio);
115         mergeSort(v, meio+1, fim);
116         merge(v, inicio, meio, fim);
117     }
118 }
```

```
void merge(int *v, int inicio, int meio, int fim){
    inicio = 0; meio = 1; fim = 3
```



# ORDENAÇÃO POR MISTURA – MERGESORT

```
110 void mergeSort(int *v, int inicio, int fim){
111     int meio; //para achar o meio do vetor
112     if(inicio < fim){
113         meio = floor((inicio+fim)/2);
114         mergeSort(v, inicio, meio);
115         mergeSort(v, meio+1, fim);
116         merge(v, inicio, meio, fim);
117     }
118 }
```

void merge(int \*v, int inicio, int meio, int fim){

inicio = 0; meio = 1; fim = 3

vet

-3	9	44	63	107	41	37
0	1	2	3	4	5	6

**X**

-3	9	44	63
0	1	2	3

vet

44	63	-3	9	107	41	37
----	----	----	---	-----	----	----

Inicio = 0  
fim = 6

meio = 3

44	63	-3	9
----	----	----	---

Inicio = 0  
fim = 3

meio = 1

44	63
----	----

-3	9
----	---

Inicio = 0  
fim = 1

meio = 0

44
----

Inicio = 0  
fim = 0

63
----

Inicio = 1  
fim = 1

Inicio = 1  
fim = 1

Inicio = 2  
fim = 2

meio = 2

-3
----

Inicio = 2  
fim = 2

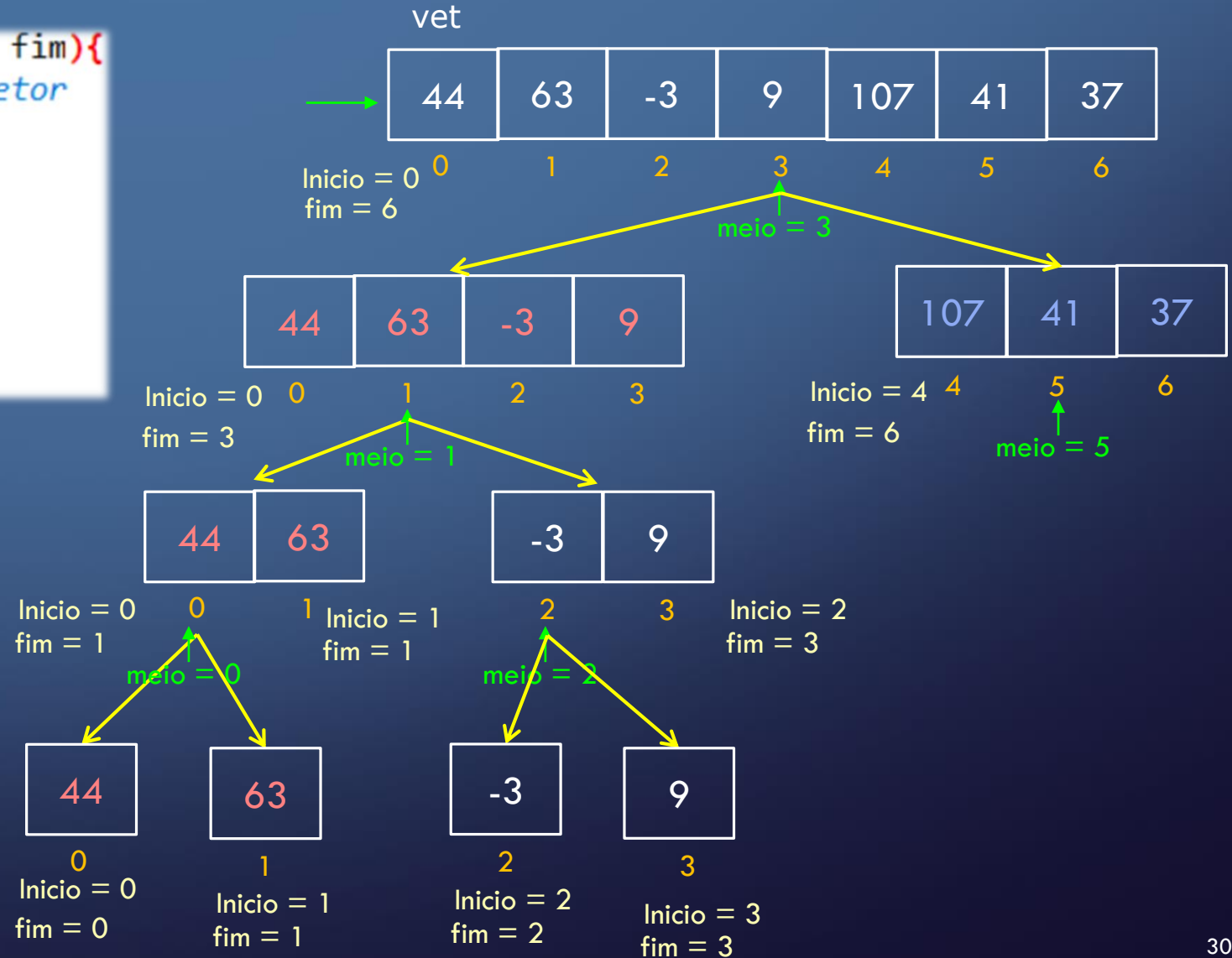
9
---

Inicio = 3  
fim = 3

Inicio = 2  
fim = 3

# ORDENAÇÃO POR MISTURA – MERGESORT

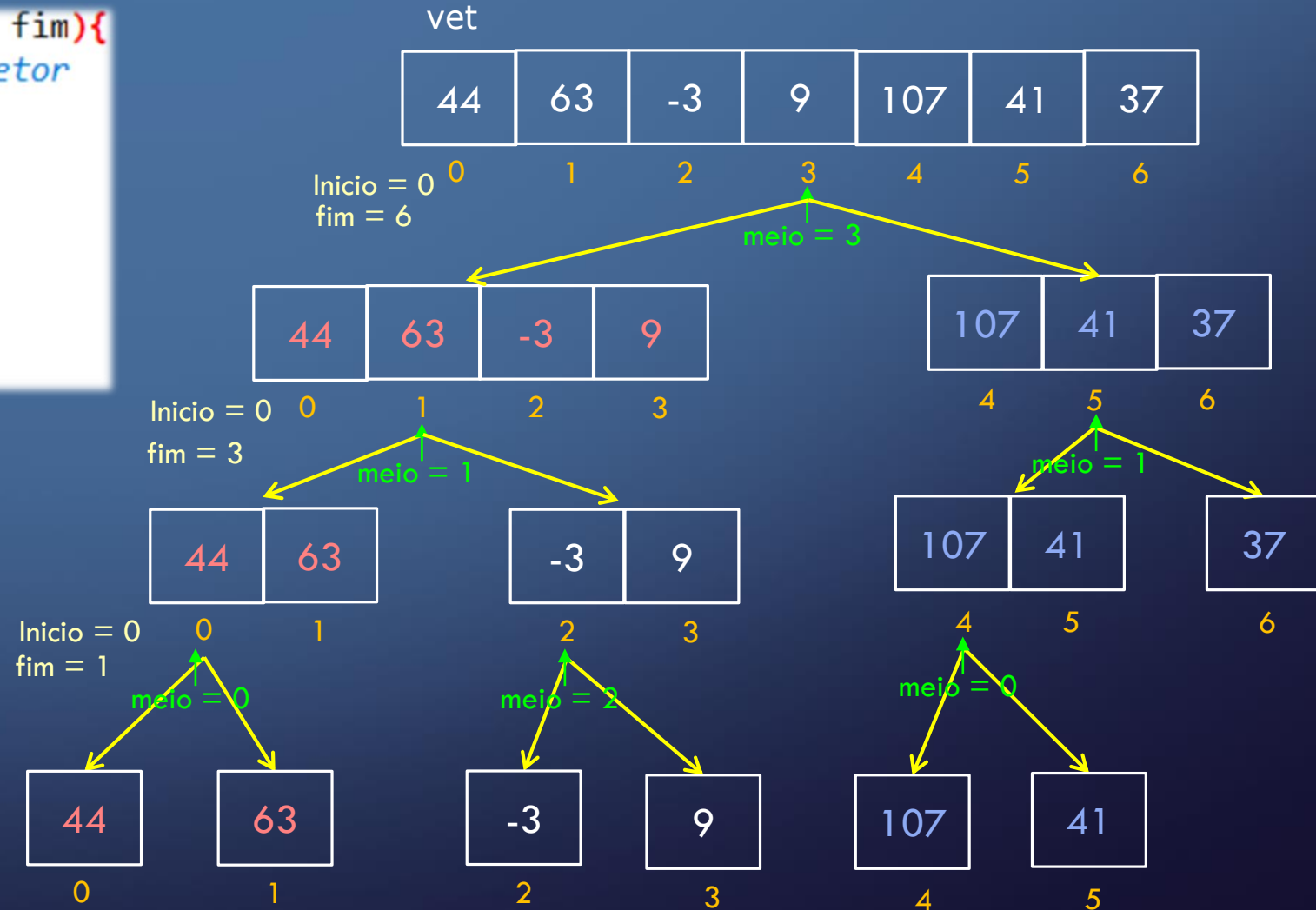
```
110 void mergeSort(int *v, int inicio, int fim){  
111     int meio; //para achar o meio do vetor  
112     if(inicio < fim){  
113         meio = floor((inicio+fim)/2);  
114         mergeSort(v, inicio, meio);  
115         mergeSort(v, meio+1, fim);  
116         merge(v, inicio, meio, fim);  
117     }  
118 }
```





# ORDENAÇÃO POR MISTURA – MERGESORT

```
110 void mergeSort(int *v, int inicio, int fim){  
111     int meio; //para achar o meio do vetor  
112     if(inicio < fim){  
113         meio = floor((inicio+fim)/2);  
114         mergeSort(v, inicio, meio);  
115         mergeSort(v, meio+1, fim);  
116         merge(v, inicio, meio, fim);  
117     }  
118 }
```



# ORDENAÇÃO POR MISTURA – MERGESORT

```
80 void merge(int *v, int inicio, int meio, int fim){
81     int *temp, p1, p2, tamanho, i, j, k;
82     int fim1 = 0, fim2 = 0;
83     tamanho = fim-inicio+1;
84     p1 = inicio;
85     p2 = meio+1;
86     temp = (int *) malloc(tamanho*sizeof(int));
87     if(temp != NULL){
88         for(i=0; i<tamanho; i++){
89             if(!fim1 && !fim2){
90                 if(v[p1] < v[p2])
91                     temp[i]=v[p1++];
92                 else
93                     temp[i]=v[p2++];
94
95                 if(p1>meio) fim1=1;
96                 if(p2>fim) fim2=1;
97             }else{
98                 if(!fim1)
99                     temp[i]=v[p1++];
100                 else
101                     temp[i]=v[p2++];
102             }
103         }
104         for(j=0, k=inicio; j<tamanho; j++, k++)
105             v[k]=temp[j];
106     }
107     free(temp);
108 }
```



# ORDENAÇÃO POR PARTIÇÃO – QUICKSORT - DIVIDIR E CONQUISTAR

No método **QuickSort**, um elemento do conjunto de dados é escolhido (**pivô**); os demais **elementos menores** que o pivô são **rearranjados à sua esquerda**; os **elementos maiores à sua direita**. Por fim, ordena-se as duas partições.

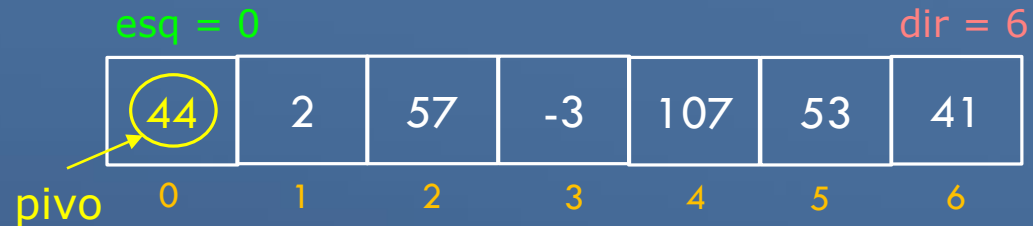
## **Performance:**

- Melhor Caso:  $O(n \log n)$
- Pior Caso (raro):  $O(n^2)$
- Dificuldade: escolher o pivô.

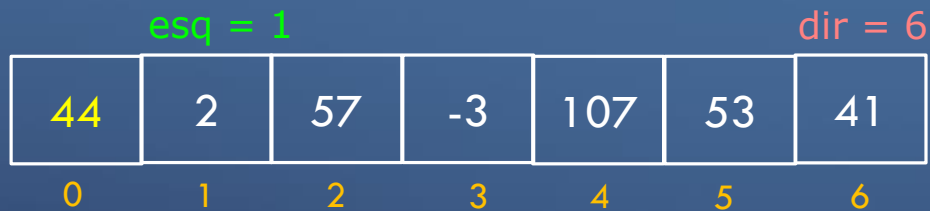
# ORDENAÇÃO POR PARTIÇÃO – QUICKSORT

particiona(v,0,6);

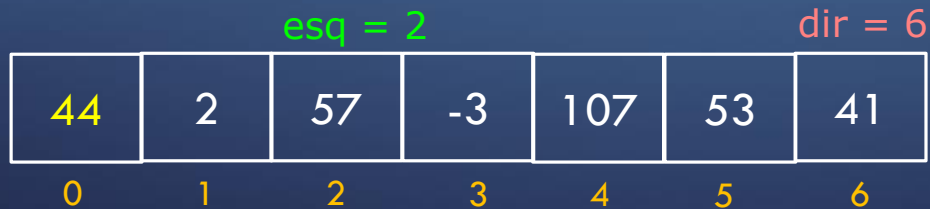
```
175 int particiona(int *v, int posInicio, int posFim ){
```



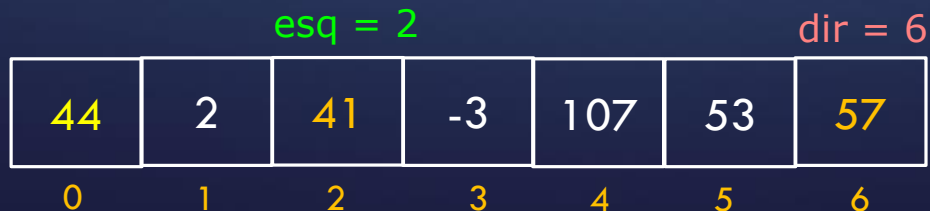
esq = posInicio;  
dir = posFim;  
pivo = v[posInicio] = 44  
v[esq] <= pivo: esq++;



v[esq] <= pivo: esq++;



v[esq] > pivo: COMPARA DIR;  
v[dir] < pivo: trocam de lugar;



v[esq] <= pivo: esq++;

# ORDENAÇÃO POR PARTIÇÃO – QUICKSORT

esq = 3      dir = 6

44	2	41	-3	107	53	57
0	1	2	3	4	5	6

v[esq] <= pivo: esq++;

esq = 4      dir = 6

44	2	41	-3	107	53	57
0	1	2	3	4	5	6

v[esq] > pivo: COMPARA DIR;  
v[dir] > pivo : dir--;

esq = 4      dir = 5

44	2	41	-3	107	53	57
0	1	2	3	4	5	6

v[dir] > pivo : dir--;

dir = 4  
esq = 4

44	2	41	-3	107	53	57
0	1	2	3	4	5	6

v[dir] > pivo : dir--;

# ORDENAÇÃO POR PARTIÇÃO – QUICKSORT

dir = 3 esq = 4

44	2	41	-3	107	53	57
0	1	2	3	4	5	6

$v[\text{dir}] < \text{pivo}$  : trocam de lugar;

dir = 3 esq = 4

-3	2	41	44	107	53	57
0	1	2	3	4	5	6

Menores que pivo

Maiores que pivo

pivo

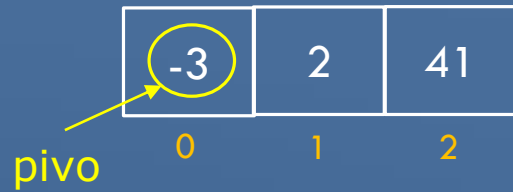
$v[\text{dir}] < \text{pivo}$  : trocam de lugar;



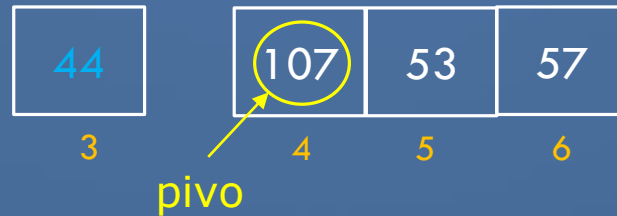
Aplica-se quickSort para cada uma das metades/partições

# ORDENAÇÃO POR PARTIÇÃO – QUICKSORT

particiona(v,0,2);



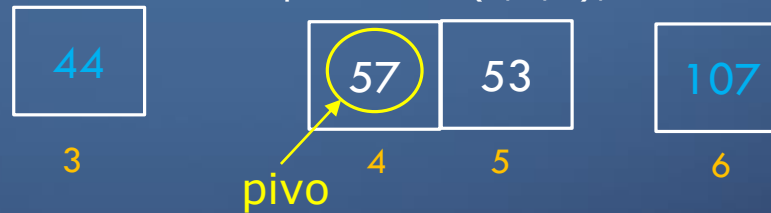
particiona(v,4,6);



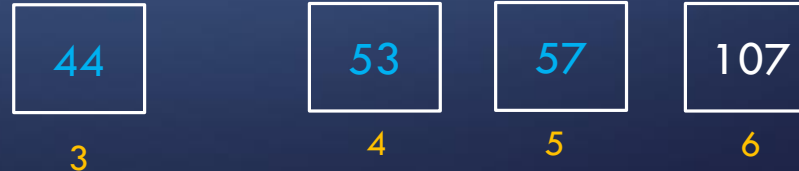
particiona(v,0,2);



particiona(v,4,6);



particiona(v,0,2);



# DÚVIDAS?



# EXERCÍCIOS DE FIXAÇÃO

1. Implemente todos os métodos de busca *Sequência/Linear*; *Ordenada* e *Busca Binária*.
2. Implemente os métodos de ordenação simples vistos: *Insertion Sort*; *Selection Sort* e *Bubble Sort*.
3. Implemente o método *mergeSort* para vetor de alunos.
4. Implemente o método *quickSort* para vetor de alunos. Fique a vontade para consultar livros e apostilas.
5. Pesquise sobre a função *qsort()*; e crie um código para ordenar os dados do vetor alunos.