

Estruturas de Dados

Prof. Rodrigo Martins

rodrigo.martins@francomontoro.com.br

Cronograma da Aula

- Tipos de Ordenação
 - Insertion Sort
 - Selection Sort
 - Merge Sort
 - Quick Sort
- Exemplos
- Exercício

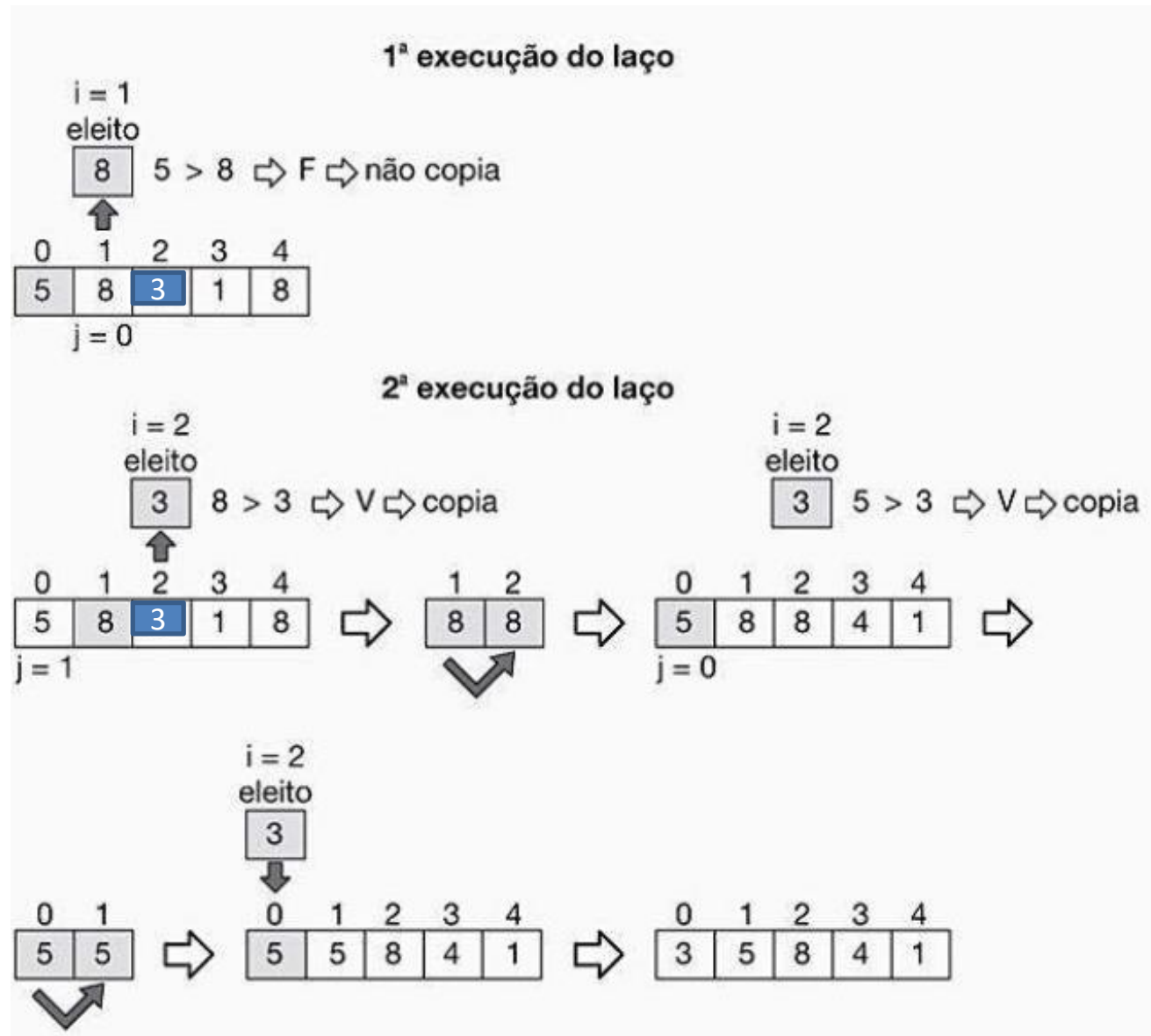
Insertion Sort

- A ordenação por inserção é um algoritmo de ordenação simples e eficiente, que percorre uma lista de elementos e insere cada elemento em sua posição correta, mantendo a lista ordenada em cada etapa.
- É um algoritmo que funciona bem com pequenas quantidades de dados e é relativamente fácil de implementar.

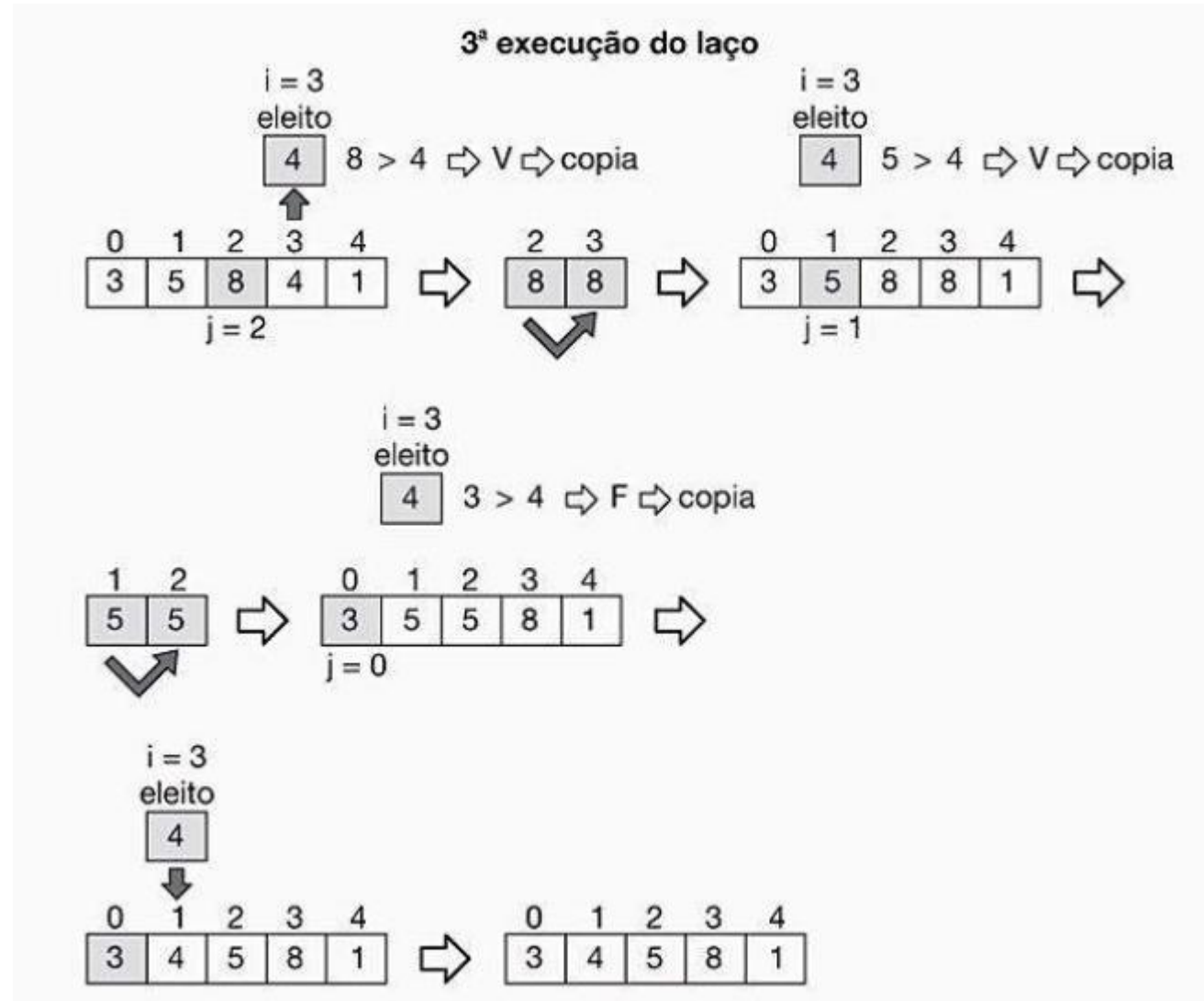
Insertion Sort

- O algoritmo funciona da seguinte forma:
 1. Percorre a lista, começando do segundo elemento até o último.
 2. Compara o elemento atual com o elemento anterior.
 3. Se o elemento atual for menor que o elemento anterior, os elementos são trocados.
 4. Repete-se o processo de comparação e troca com os elementos anteriores até que o elemento atual seja maior que o elemento anterior ou até chegar ao início da lista.
 5. Insere o elemento atual na posição correta na lista ordenada.
 6. Repete o processo para o próximo elemento na lista, até que toda a lista esteja ordenada.

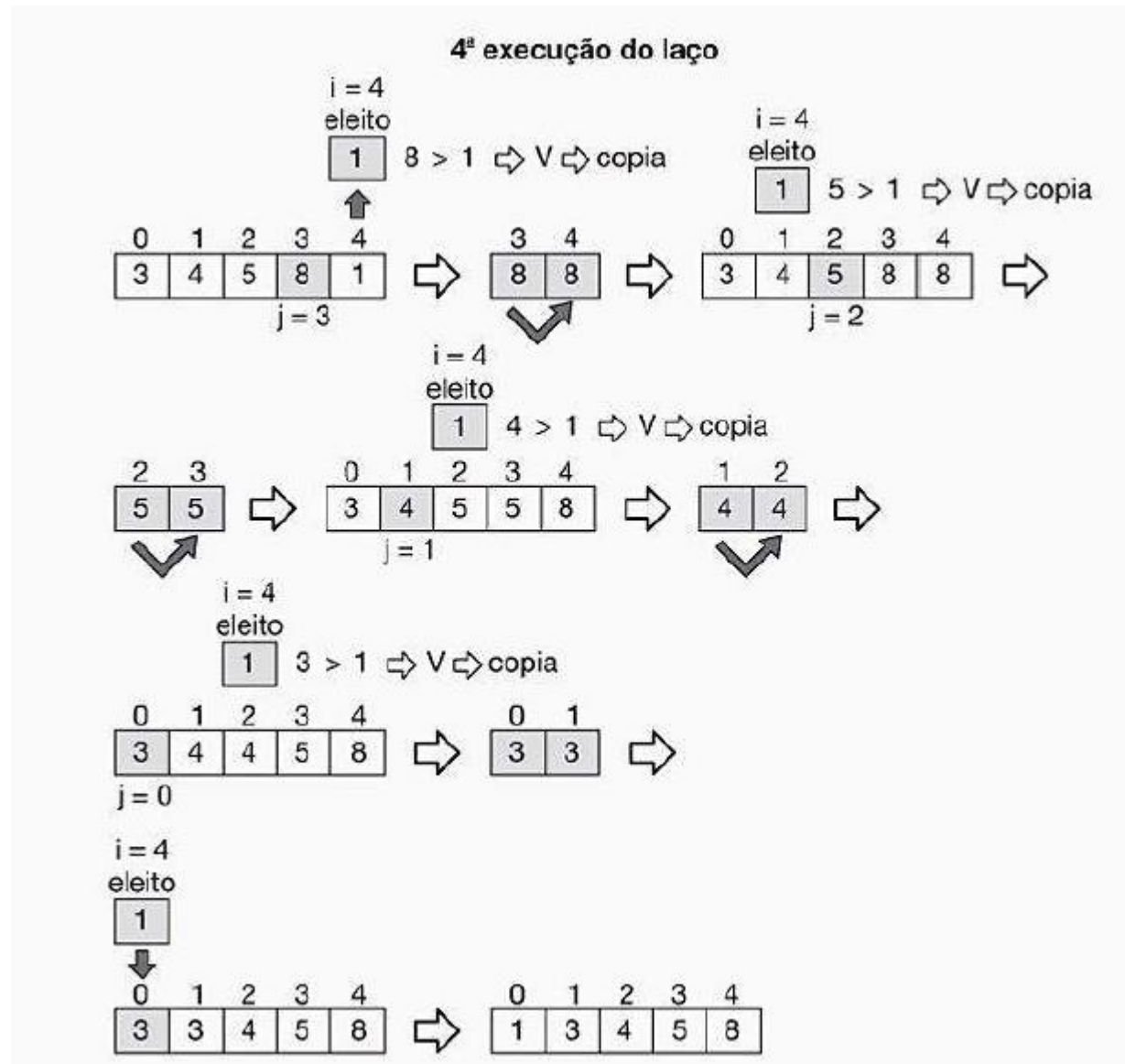
Insertion Sort



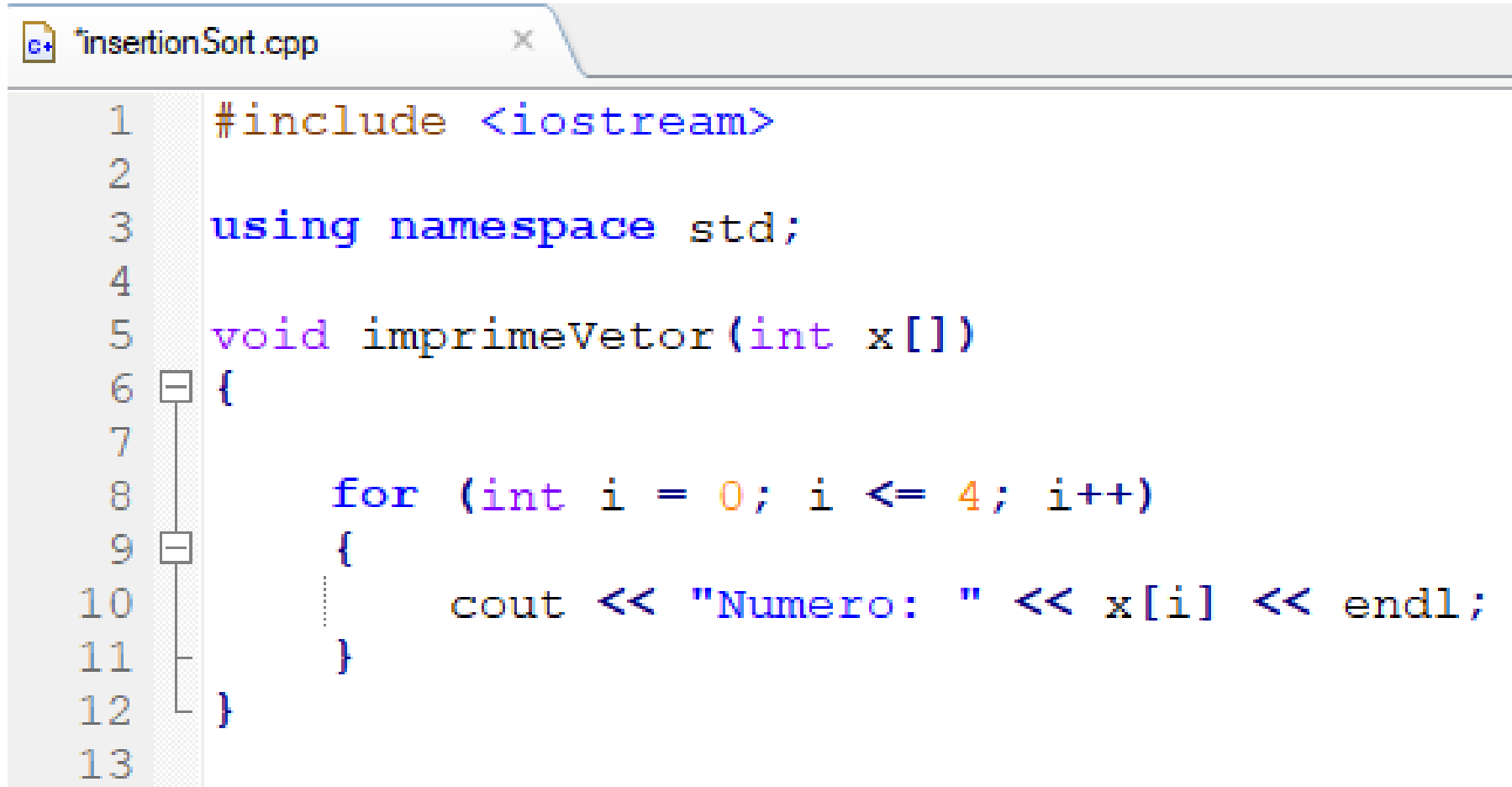
Insertion Sort



Insertion Sort



Insertion Sort



```
1  #include <iostream>
2
3  using namespace std;
4
5  void imprimeVetor(int x[])
6  {
7
8      for (int i = 0; i <= 4; i++)
9      {
10         cout << "Numero: " << x[i] << endl;
11     }
12 }
13
```


Insertion Sort

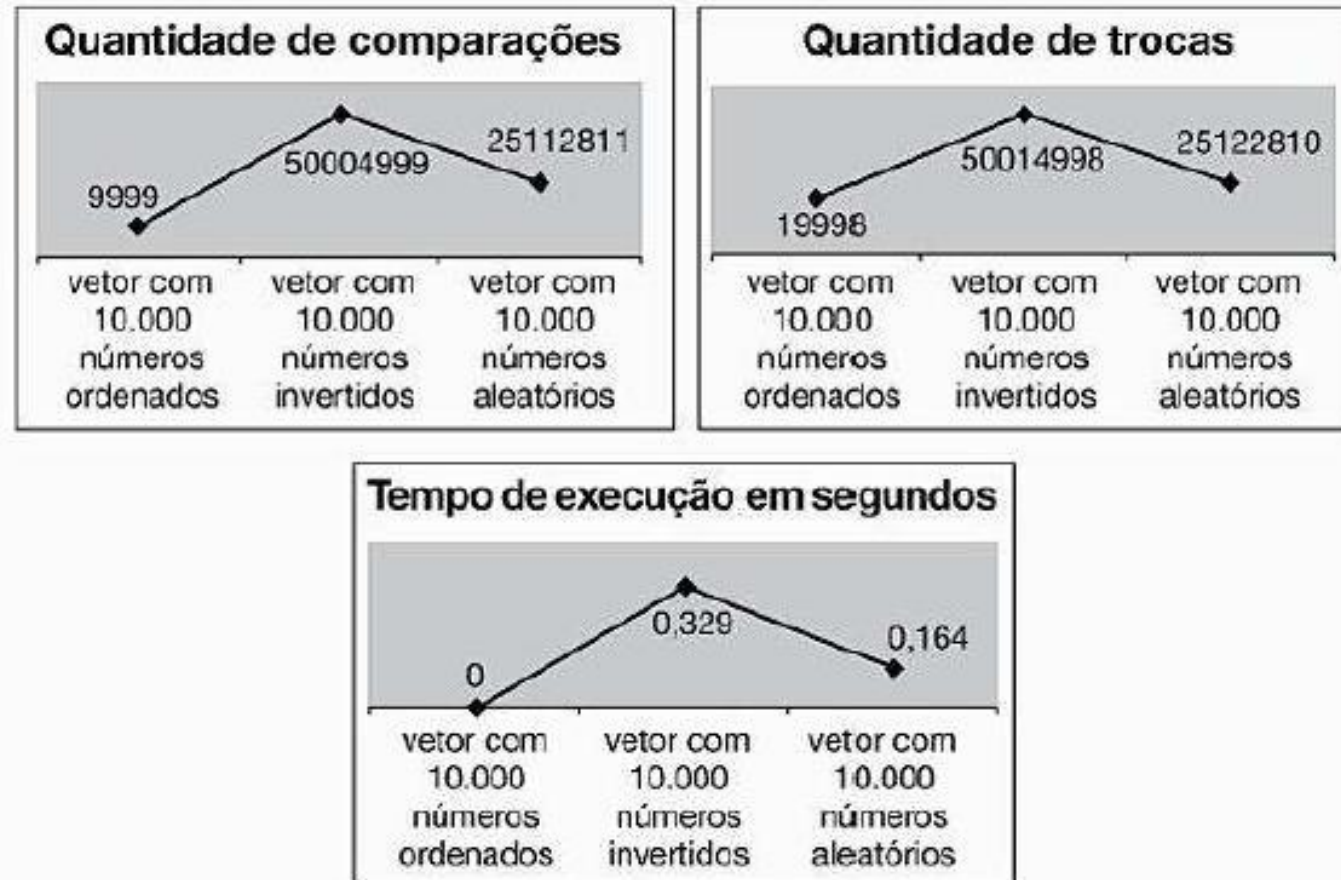
```
14 void insertionSort(int vetor[])
15 {
16
17     int j = 0, eleito = 0;
18
19     //ordenando de forma crescente
20     //laço com a quantidade de elementos do vetor - 1
21     for (int i = 1; i <= 4; i++)
22     {
23         eleito = vetor[i];
24         j = i - 1;
25         //laço que percorre os elementos a esquerda do numero eleito
26         //ou até encontrar a posição para recolocação do número eleito
27         //respeitando a ordenação procurada
28         while (j >= 0 && vetor[j] > eleito)
29         {
30             vetor[j + 1] = vetor[j];
31             j--;
32         }
33         vetor[j + 1] = eleito;
34     }
35
36 }
```

Insertion Sort

```
37
38 int main(int argc, char** argv)
39 {
40     int vetor[5];
41
42     //carregando os números no vetor
43     for (int i = 0; i <= 4; i++)
44     {
45         cout << "Digite o numero" << endl;
46         cin >> vetor[i];
47     }
48
49     imprimeVetor(vetor);
50     insertionSort(vetor);
51     cout << endl;
52     imprimeVetor(vetor);
53     return 0;
54 }
```

Insertion Sort

- Gráfico de desempenho:



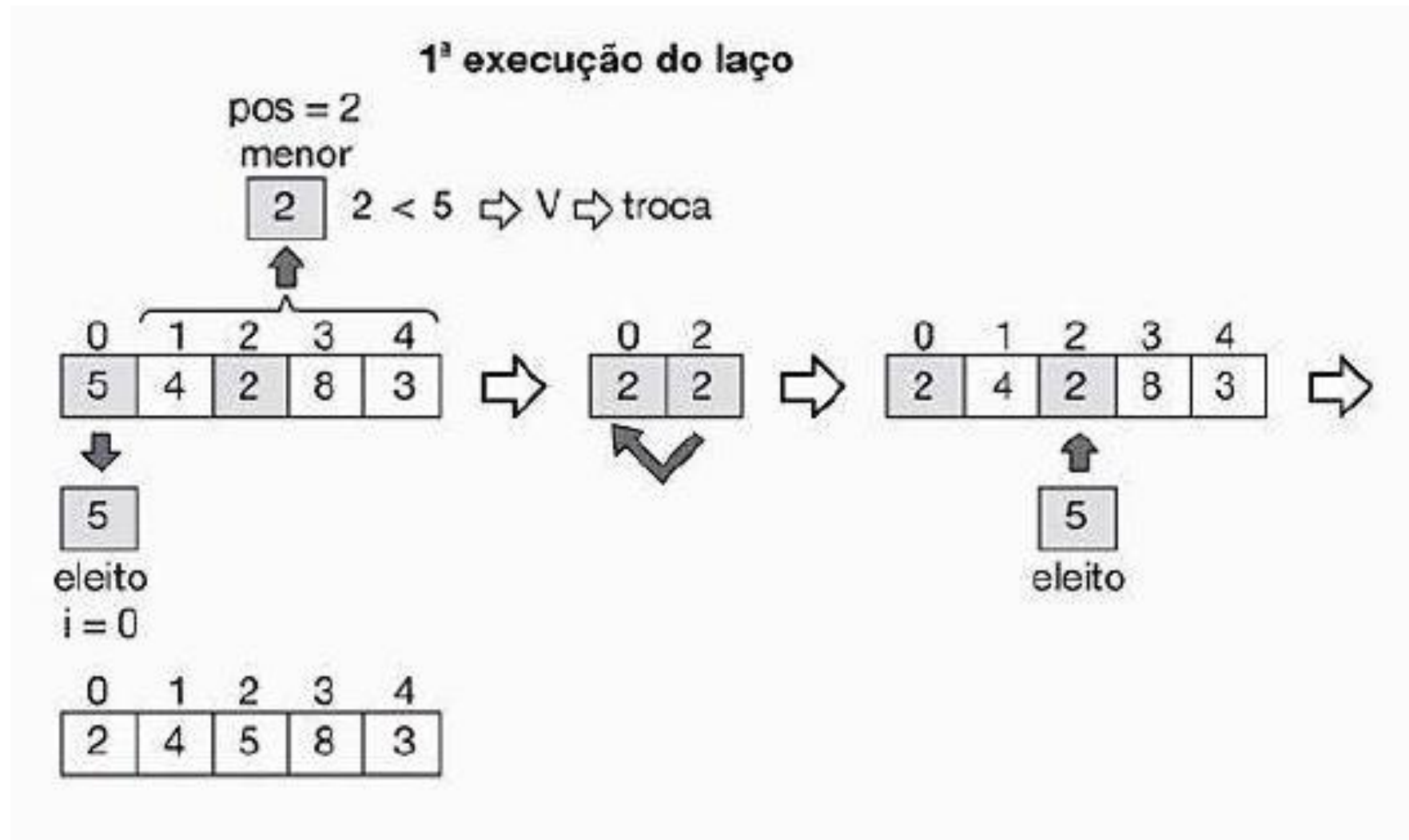
Selection Sort

- Este algoritmo é baseado em se passar sempre o menor valor do vetor para a primeira posição (ou o maior dependendo da ordem requerida), depois o segundo menor valor para a segunda posição e assim sucessivamente, até os últimos dois elementos.
- Neste algoritmo de ordenação é escolhido um número a partir do primeiro, este número escolhido é comparado com os números a partir da sua direita, quando encontrado um número menor, o número escolhido ocupa a posição do menor número encontrado.

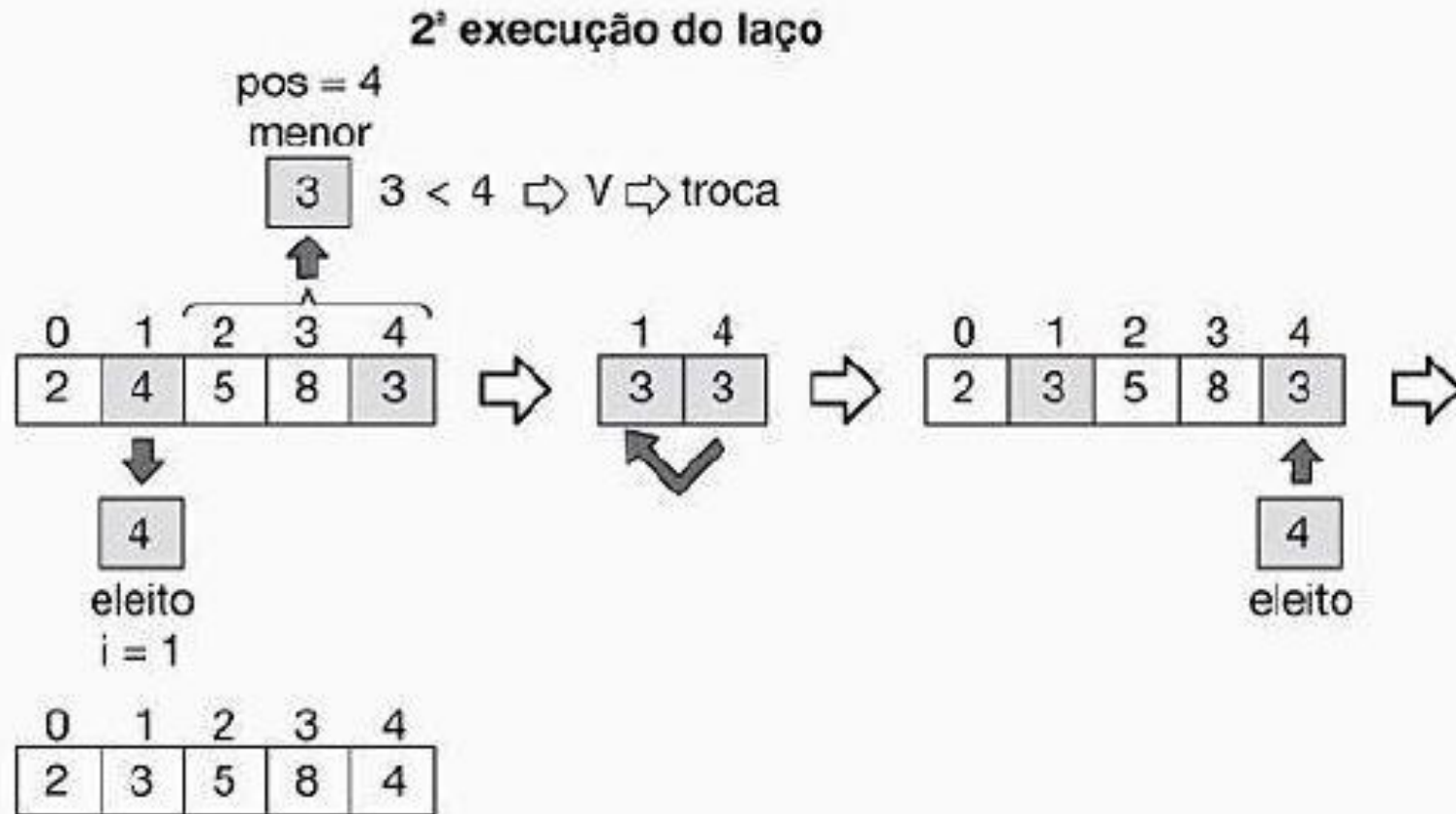
Selection Sort

- Este número encontrado será o próximo número escolhido, caso não for encontrado nenhum número menor que este escolhido, ele é colocado na posição do primeiro número escolhido, e o próximo número à sua direita vai ser o escolhido para fazer as comparações.
- É repetido esse processo até que a lista esteja ordenada.

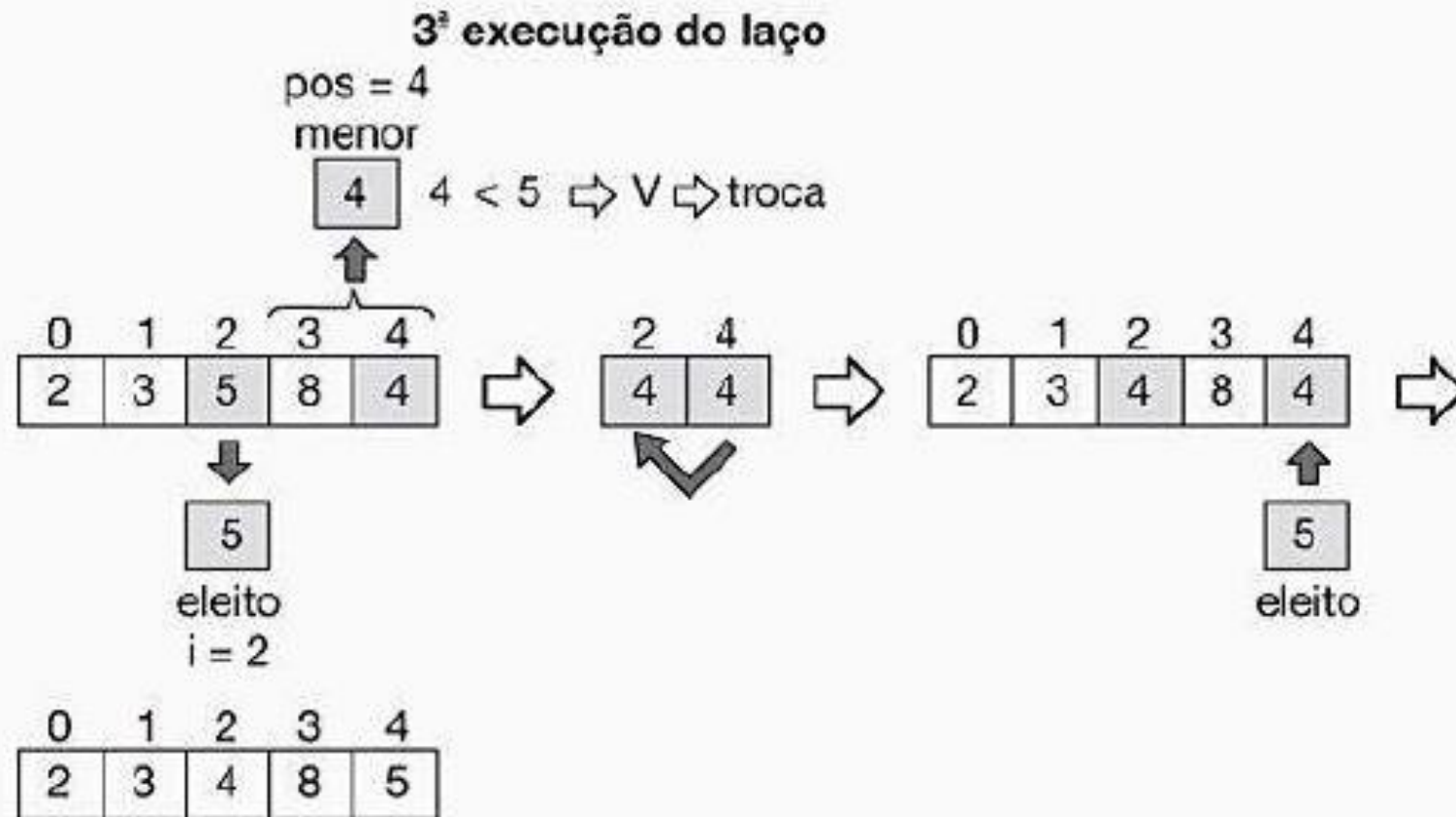
Selection Sort



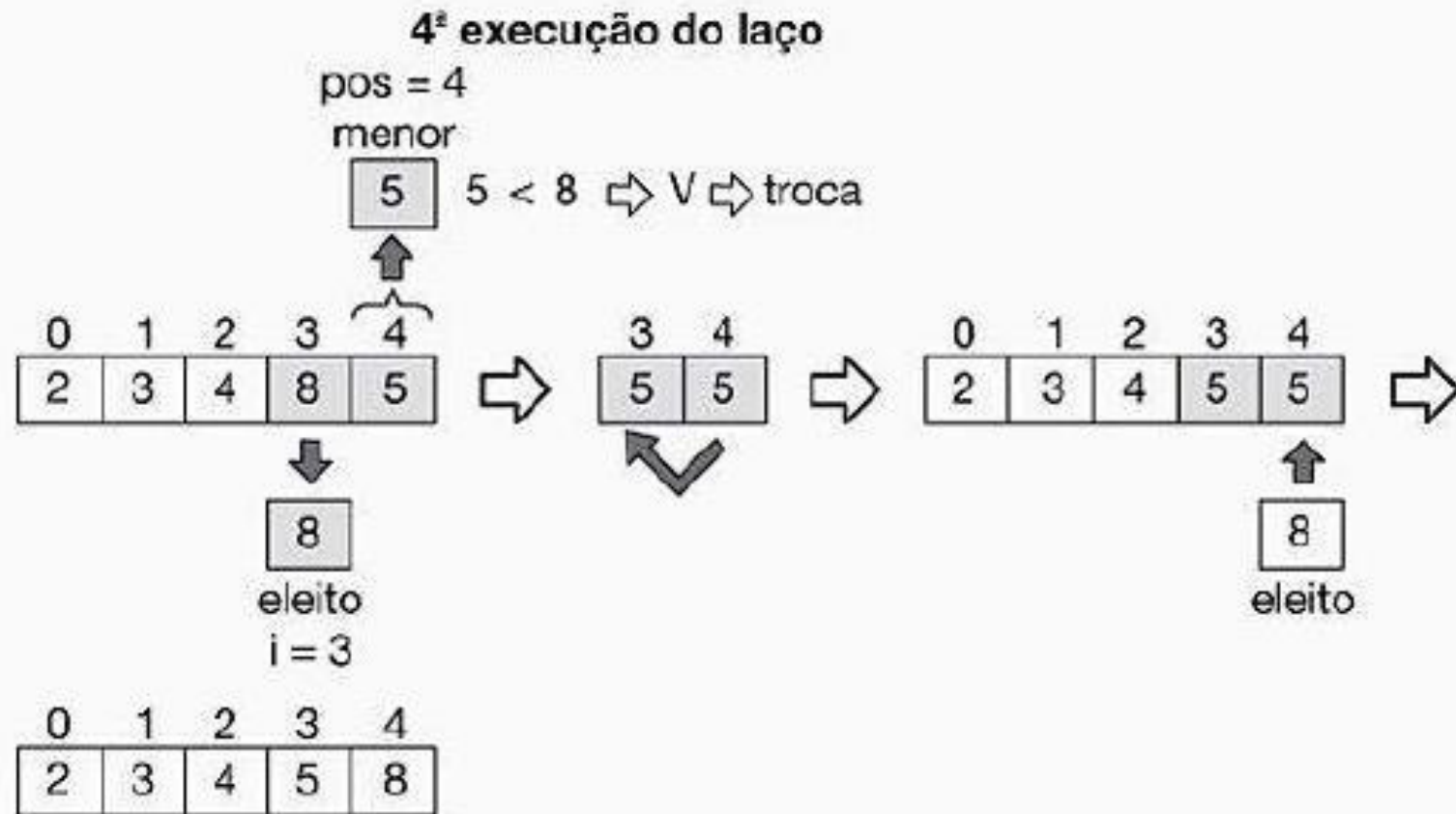
Selection Sort



Selection Sort



Selection Sort



Selection Sort

```
*selectionSort.cpp
1  #include <iostream>
2  #include <stdlib.h>
3  #include <string>
4
5  #define TAM 10
6
7  using namespace std;
8
9  void imprimeVetor(int vetor[]){
10     int i;
11     cout << endl;
12     for (i = 0; i < TAM; i++){
13         cout << " |" << vetor[i] << "| ";
14     }
15 }
16
```

Selection Sort

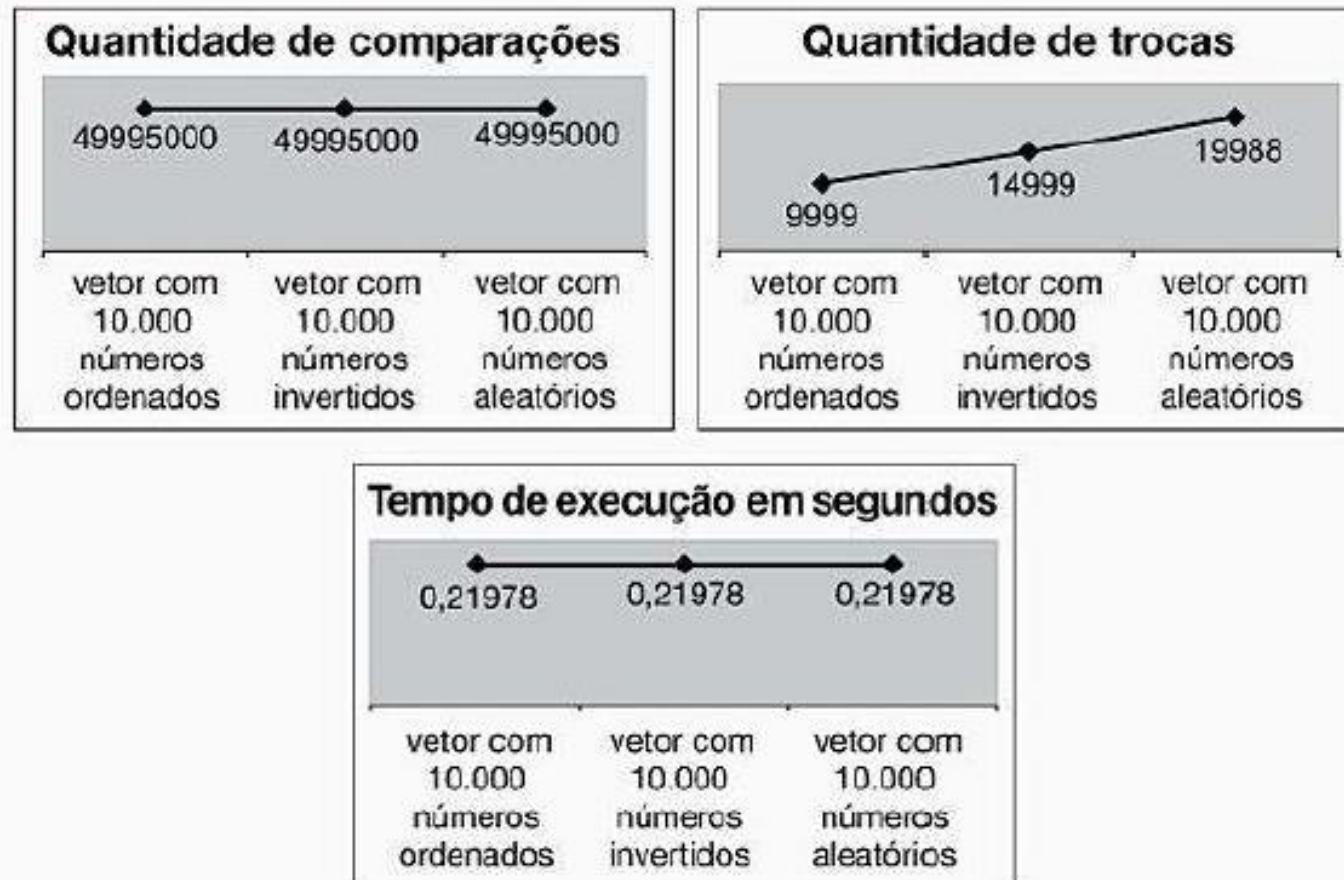
```
17 void selectionSort(int vetor[TAM]){
18     int posicaoMenorValor, aux, i, j;
19
20     for(i = 0; i < TAM; i++){
21         //recebe a posição inicial para o menor valor
22         posicaoMenorValor = i;
23
24         for(j = i + 1; j < TAM; j++){
25             //caso encontre um valor menor na frente dos analisados
26             if(vetor[j] < vetor[posicaoMenorValor]){
27                 posicaoMenorValor = j;
28             }
29         }
30
31         //verifica se houve mudança e troca os valores
32         if (posicaoMenorValor != i){
33             aux = vetor[i];
34             vetor[i] = vetor[posicaoMenorValor];
35             vetor[posicaoMenorValor] = aux;
36         }
37     }
38 }
```

Selection Sort

```
39
40 int main() {
41
42     int vetor[TAM] = {10,9,8,7,6,5,4,3,2,1};
43
44     imprimeVetor(vetor);
45     cout << endl;
46
47     selectionSort(vetor);
48     imprimeVetor(vetor);
49
50 }
```

Selection Sort

- Gráfico de desempenho:

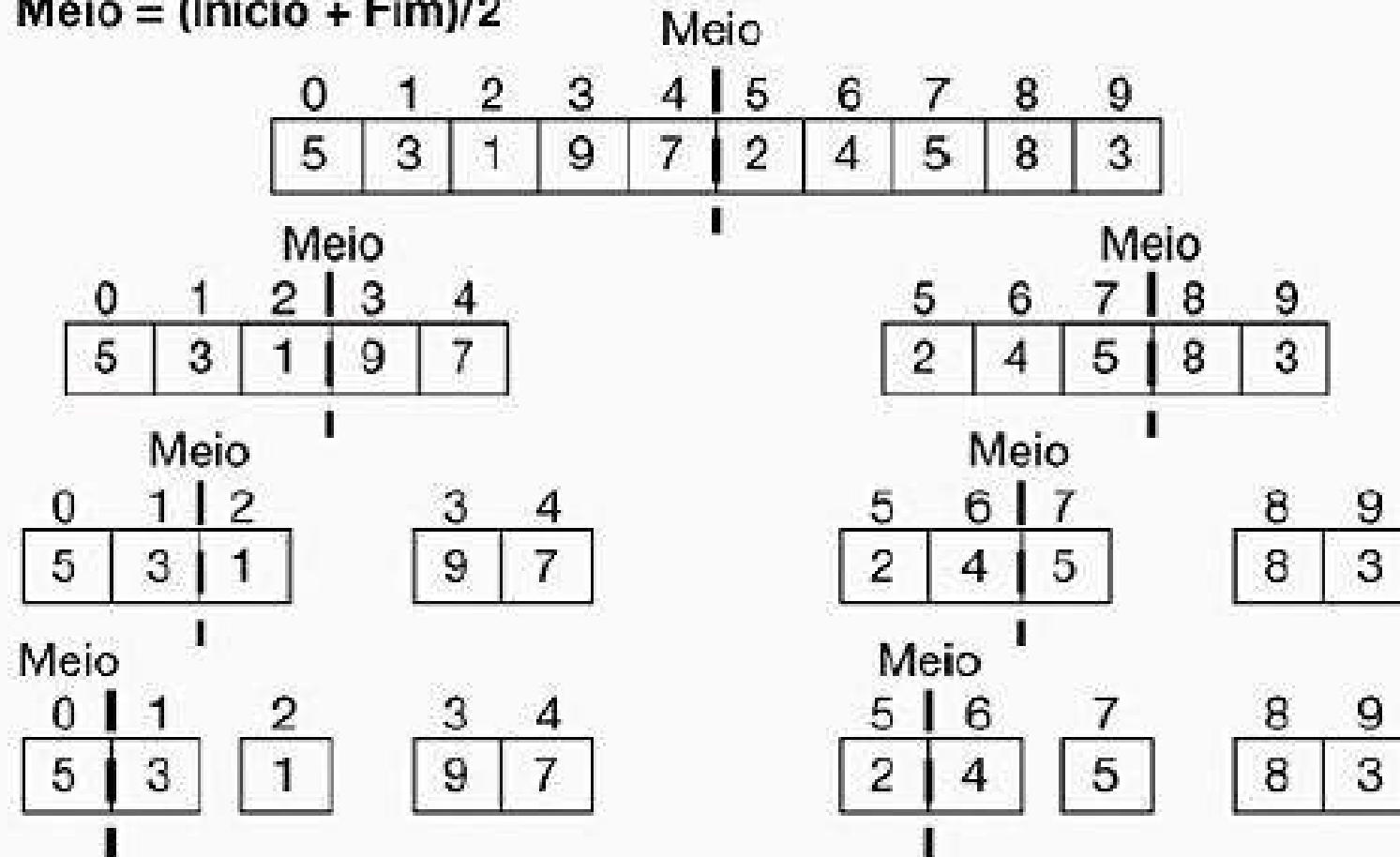


Merge Sort

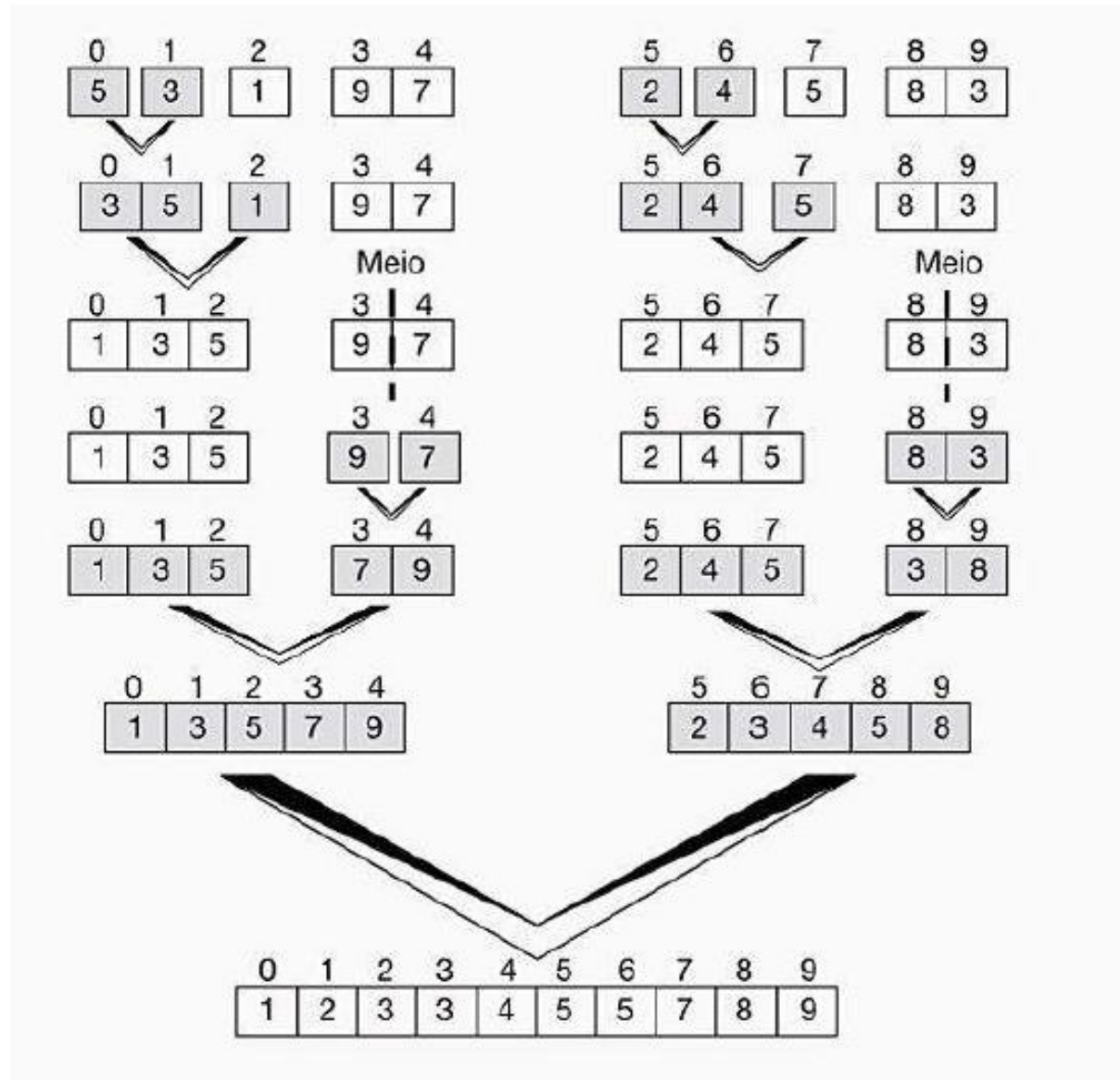
- O **Merge Sort**, ou ordenação por mistura, é um exemplo de algoritmo de ordenação por comparação do tipo dividir-para-conquistar.
- Sua ideia básica consiste em Dividir (o problema em vários subproblemas e resolver esses subproblemas através da recursividade) e Conquistar (após todos os subproblemas terem sido resolvidos ocorre a conquista que é a união das resoluções dos subproblemas).
- Como o algoritmo Merge Sort usa a recursividade, há um alto consumo de memória e tempo de execução, tornando esta técnica não muito eficiente em alguns problemas.

Merge Sort

$$\text{Meio} = (\text{Início} + \text{Fim})/2$$



Merge Sort



Merge Sort

```
mergeSort.cpp x
1  #include <iostream>
2  #include <stdlib.h>
3  #include <string>
4
5  #define TAM 10
6
7  using namespace std;
8
9  void imprimeVetor(int vetor[]){
10     int i;
11     cout << endl;
12     for (i = 0; i < TAM; i++){
13         cout << " |" << vetor[i] << "| ";
14     }
15 }
16
```

Merge Sort

```
17 //junta os dois subarrays criados ao dividir o vetor principal
18 void merge(int vetor[TAM], int indiceEsquerdo, int meio, int indiceDireito){
19     int i, j, k;
20
21     int n1 = meio - indiceEsquerdo + 1; //tamanho do primeiro vetor auxiliar
22     int n2 = indiceDireito - meio; //tamanho do segundo vetor auxiliar
23
24     //cria dois arrays temporários
25     int vetorEsquerdo[n1], vetorDireito[n2];
26
27     //passa os elementos do vetor principal para o primeiro vetor auxiliar(esquerda)
28     for(i = 0; i < n1; i++){
29         vetorEsquerdo[i] = vetor[indiceEsquerdo + i];
30     }
31
32     //passa os elementos do vetor principal para o segundo vetor auxiliar(direita)
33     for(j = 0; j < n2; j++){
34         vetorDireito[j] = vetor[meio + 1 + j];
35     }
36
37     //reseta as variáveis
38     i = 0;
39     j = 0;
40     k = indiceEsquerdo;
```

Merge Sort

```
41
42 while(i < n1 && j < n2){
43     //caso o valor da esquerda seja menor
44     if(vetorEsquerdo[i] <= vetorDireito[j]){
45         //passo para o meu vetor principal o valor menor
46         vetor[k] = vetorEsquerdo[i];
47
48         //incrementa o auxiliar para passar a análise para os próximos valores
49         //do vetor auxiliar
50         i++;
51     }
52     else{
53         //passo para o meu vetor principal o valor menor
54         vetor[k] = vetorDireito[j];
55
56         //incrementa o auxiliar para passar a análise para os próximos valores
57         //do vetor auxiliar
58         j++;
59     }
60
61     //aumenta o indice de posicionamento no vetor
62     k++;
63 }
64
65 //se faltarem alguns elementos do array esquerdo, passa eles para o array principal
66 while(i < n1){
67     vetor[k] = vetorEsquerdo[i];
68     i++;
69     k++;
70 }
```

Merge Sort

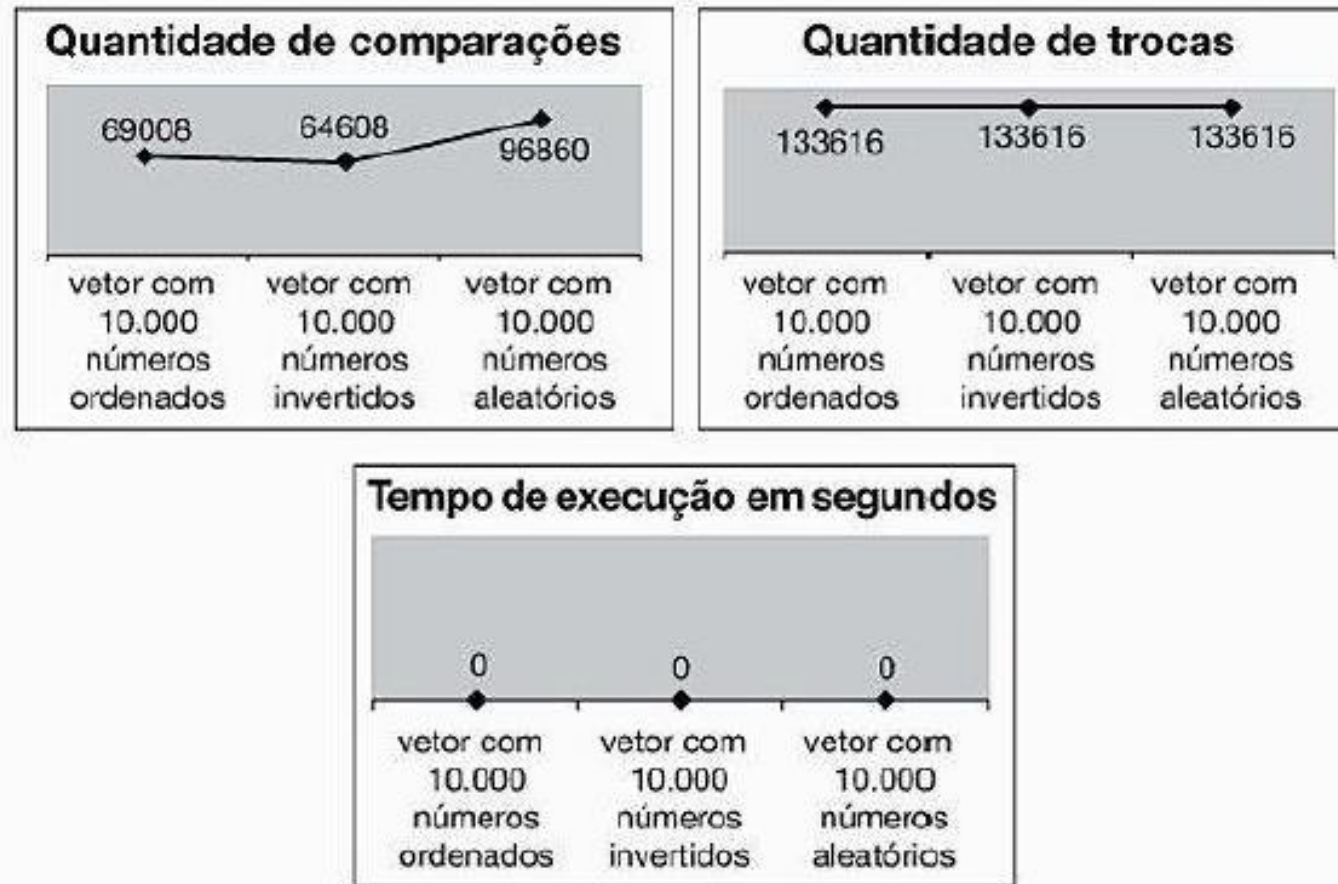
```
71
72 //se faltarem alguns elementos do array direito, passa eles para o array principal
73 while(j < n2){
74     vetor[k] = vetorDireito[j];
75     j++;
76     k++;
77 }
78
79 }
80
81 void mergeSort(int vetor[TAM], int indiceEsquerdo, int indiceDireito){
82     if(indiceEsquerdo < indiceDireito){
83         //encontra o meio
84         int meio = indiceEsquerdo + (indiceDireito - indiceEsquerdo) / 2;
85
86         //da metade para trás
87         mergeSort(vetor, indiceEsquerdo, meio);
88
89         //da metade para frente
90         mergeSort(vetor, meio + 1, indiceDireito);
91
92         //une os dois subarrays que foram criados
93         merge(vetor, indiceEsquerdo, meio, indiceDireito);
94     }
95
96
97 }
```

Merge Sort

```
98
99 □ int main() {
100
101     int vetor[TAM] = {10,9,8,7,6,5,4,3,2,1};
102
103     imprimeVetor(vetor);
104     cout << endl;
105
106     mergeSort(vetor, 0, TAM - 1);
107     imprimeVetor(vetor);
108
109 }
```

Merge Sort

- Gráfico de desempenho:



Quick Sort

- O **Quicksort** é o algoritmo mais eficiente na ordenação por comparação.
- Nele se escolhe um elemento chamado de pivô, a partir disto é organizada a lista para que todos os números anteriores a ele sejam menores que ele, e todos os números posteriores a ele sejam maiores que ele.
- Ao final desse processo o número pivô já está em sua posição final. Os dois grupos desordenados recursivamente sofreram o mesmo processo até que a lista esteja ordenada.

Quick Sort

1ª execução do laço

Vetor de 0 a 9

Posição do pivô = parte inteira $[(0+9)/2] = 4$

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

↑
pivô

$5 \leq 6 \Rightarrow V \Rightarrow$ para
↓
 $j = 9$

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

↑
pivô
pivô
↓

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

↑ $i = 0$
 $5 \geq 6 \Rightarrow F \Rightarrow$ continua

pivô
↓

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

↑ $i = 1$
 $8 \geq 6 \Rightarrow V \Rightarrow$ para

pivô
↓

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

$i < j \Rightarrow 1 < 9 \Rightarrow V \Rightarrow$ troca

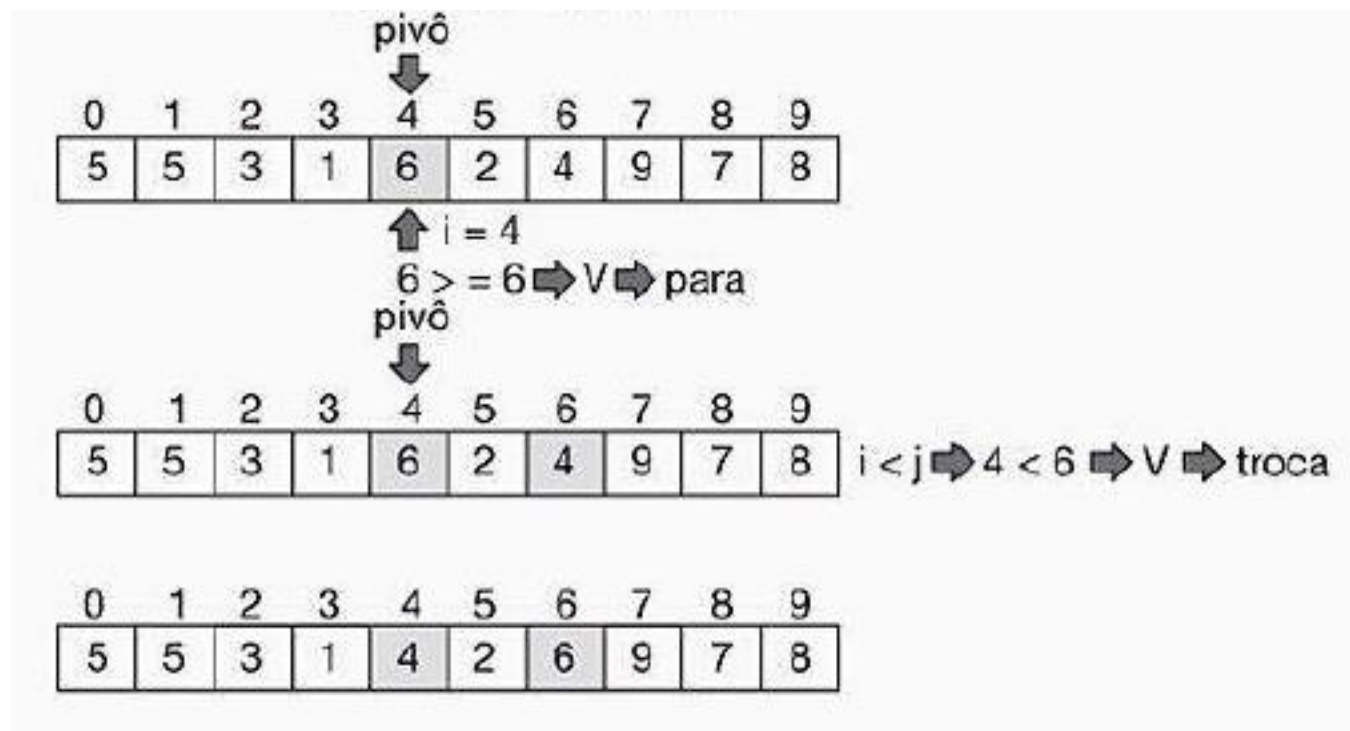
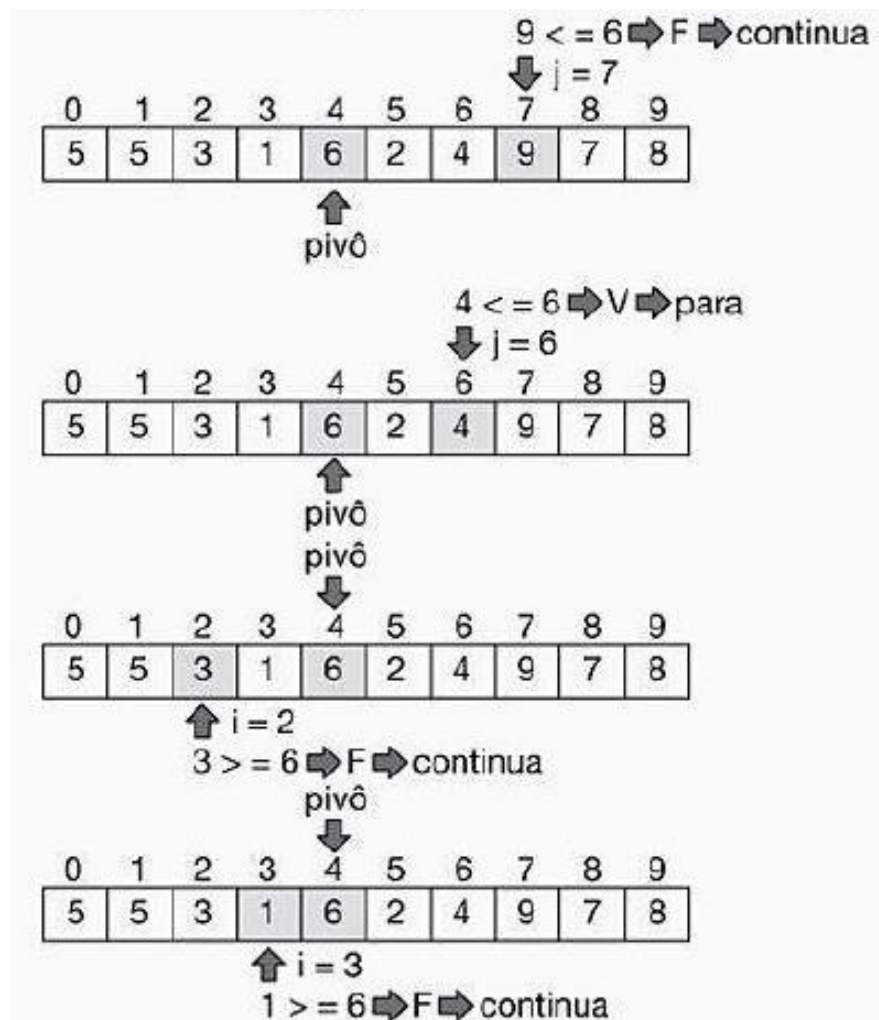
0	1	2	3	4	5	6	7	8	9
5	5	3	1	6	2	4	9	7	8

$7 \leq 6 \Rightarrow F \Rightarrow$ continua
↓
 $j = 8$

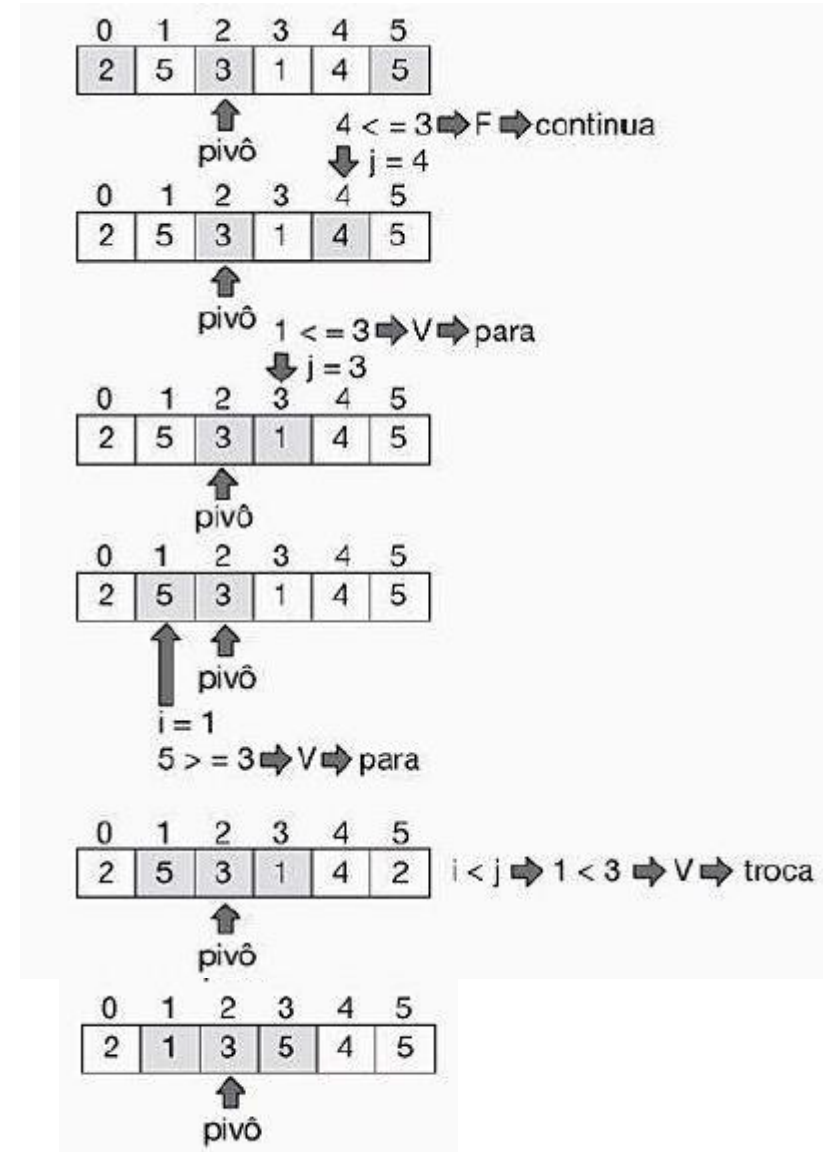
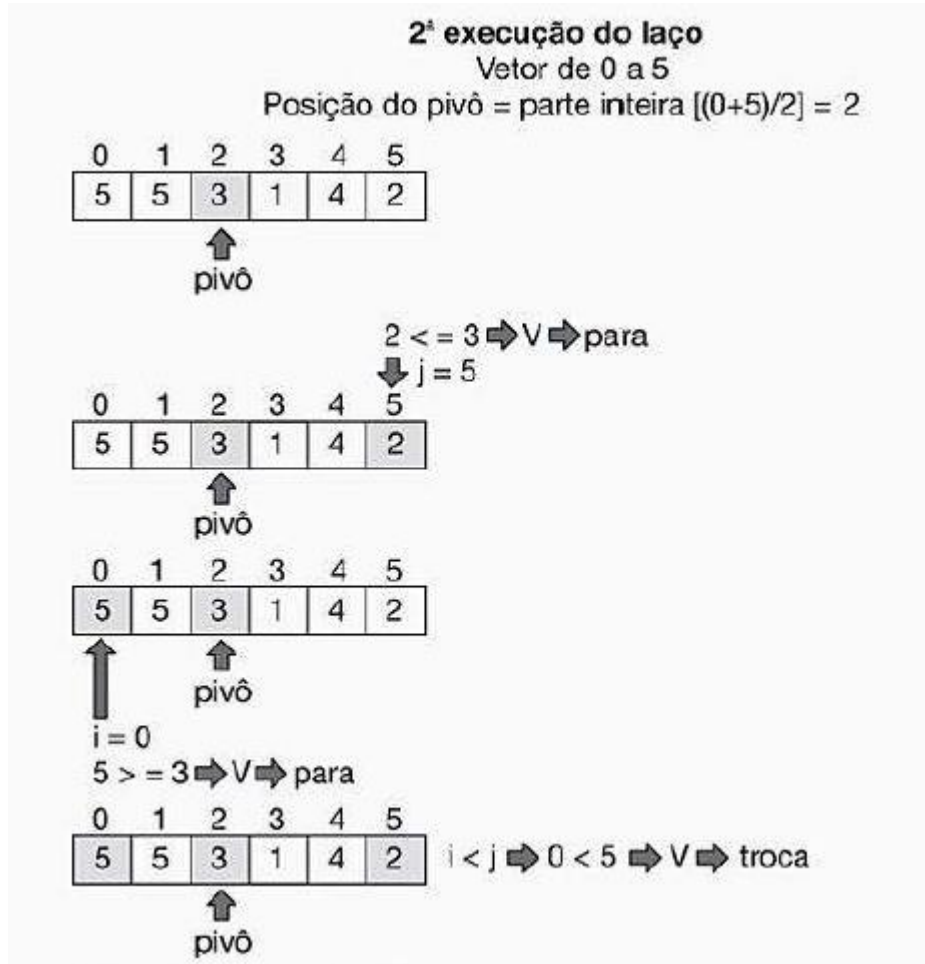
0	1	2	3	4	5	6	7	8	9
5	5	3	1	6	2	4	9	7	8

↑
pivô

Quick Sort



Quick Sort



Quick Sort

3ª execução do laço

Vetor de 0 a 2

Posição do pivô = parte inteira $\lfloor (0+2)/2 \rfloor = 1$

0	1	2
2	1	3

↑
pivô

$3 \leq 1 \Rightarrow F \Rightarrow \text{continua}$

↓ $j = 2$

0	1	2
2	1	3

↑
pivô

$1 \leq 1 \Rightarrow V \Rightarrow \text{para}$

↓ $j = 1$

0	1	2
2	1	3

↑
pivô

0	1	2
2	1	3

↑
pivô

$i = 0$

$2 \geq 1 \Rightarrow V \Rightarrow \text{para}$

0	1	2
2	1	3

$i < j \Rightarrow 0 < 1 \Rightarrow V \Rightarrow \text{troca}$

↑
pivô

0	1	2
1	2	3

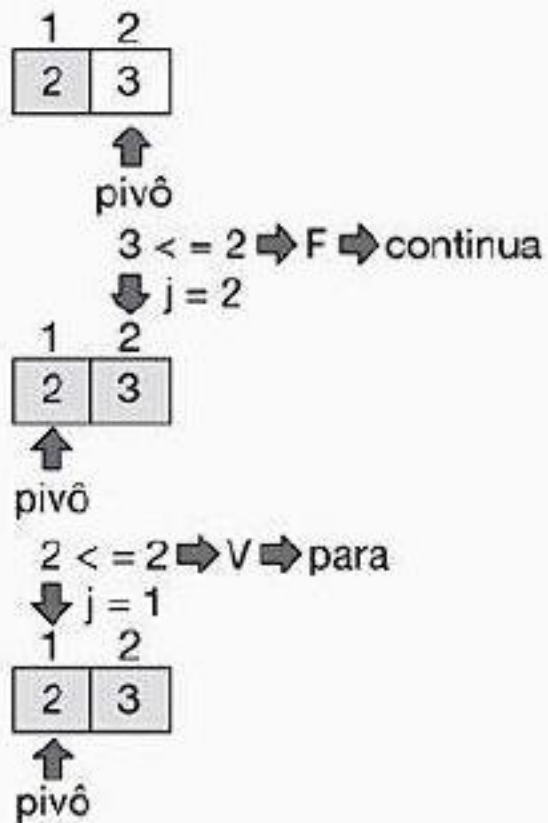
↑
pivô

Quick Sort

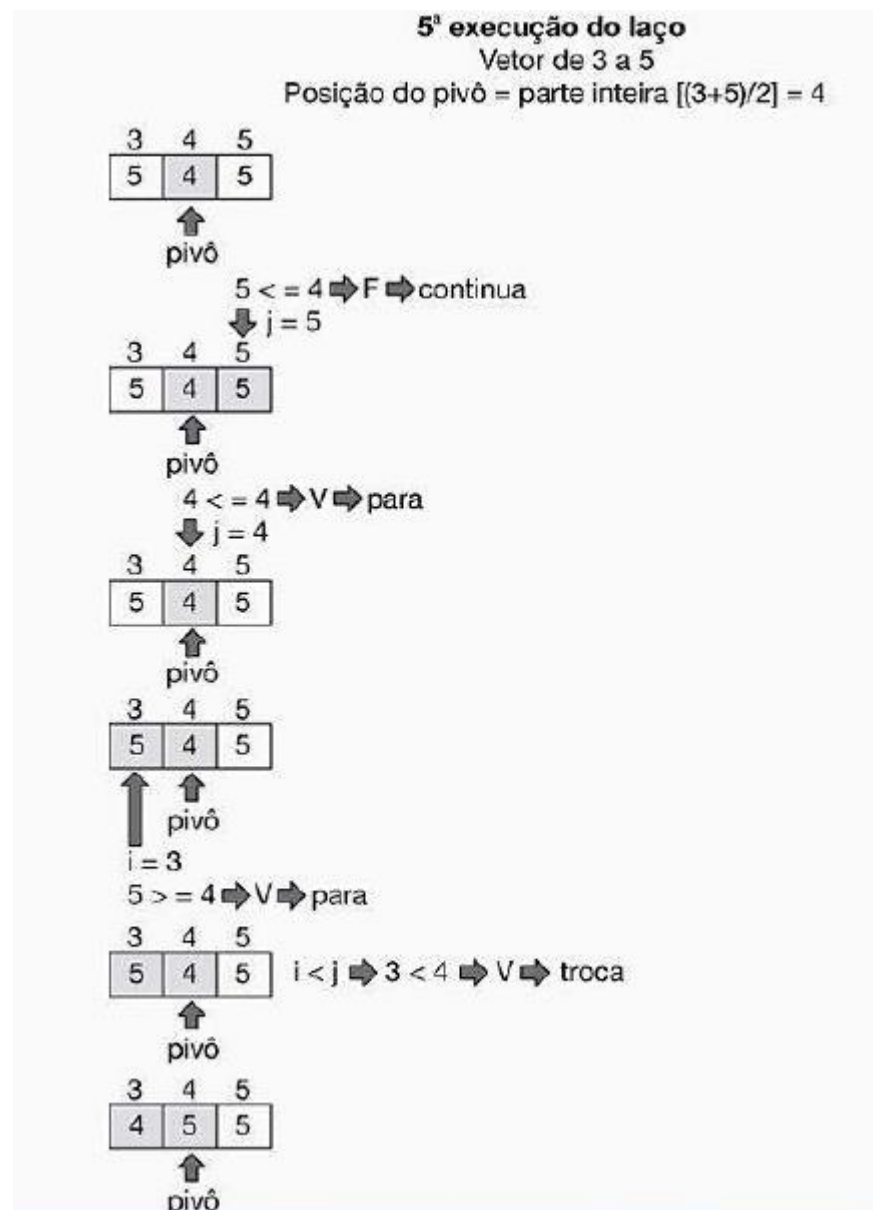
4ª execução do laço

Vetor de 1 a 2

Posição do pivô = parte inteira $[(1+2)/2] = 1$



Quick Sort

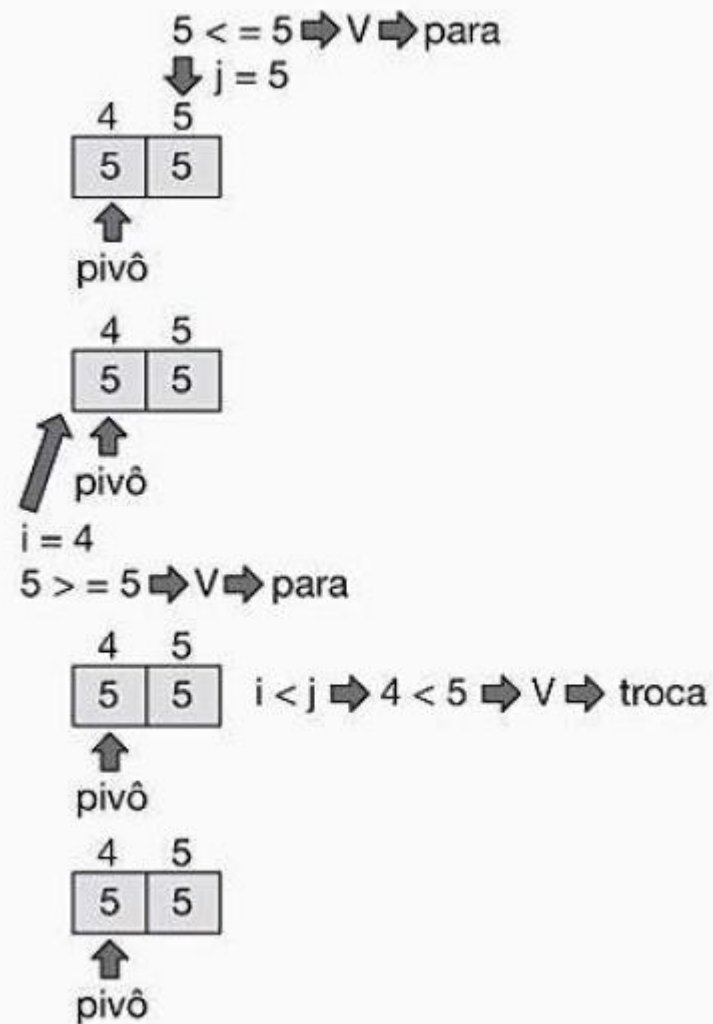
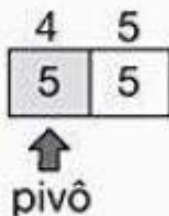


Quick Sort

6ª execução do laço

Vetor de 4 a 5

Posição do pivô = parte inteira $[(4+5)/2] = 4$



Quick Sort

7ª execução do laço

Vetor de 6 a 9

Posição do pivô = parte inteira $[(6+9)/2] = 7$

6	7	8	9
6	9	7	8

↑
pivô

$8 \leq 9 \Rightarrow V \Rightarrow \text{para}$
↓
 $j = 9$

6	7	8	9
6	9	7	8

↑
pivô

6	7	8	9
6	9	7	8

↑
↑
pivô

$i = 6$

$6 > 9 \Rightarrow F \Rightarrow \text{continua}$

6	7	8	9
6	9	7	8

↑
↑
pivô

$i = 7$

$9 > 9 \Rightarrow V \Rightarrow \text{para}$

6	7	8	9
6	9	7	8

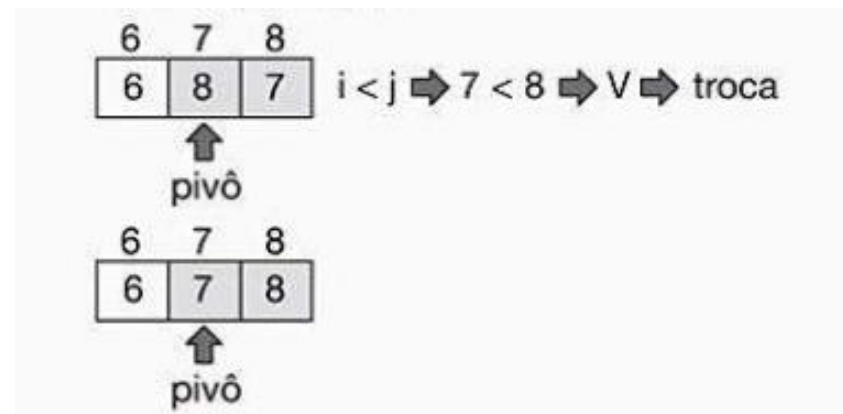
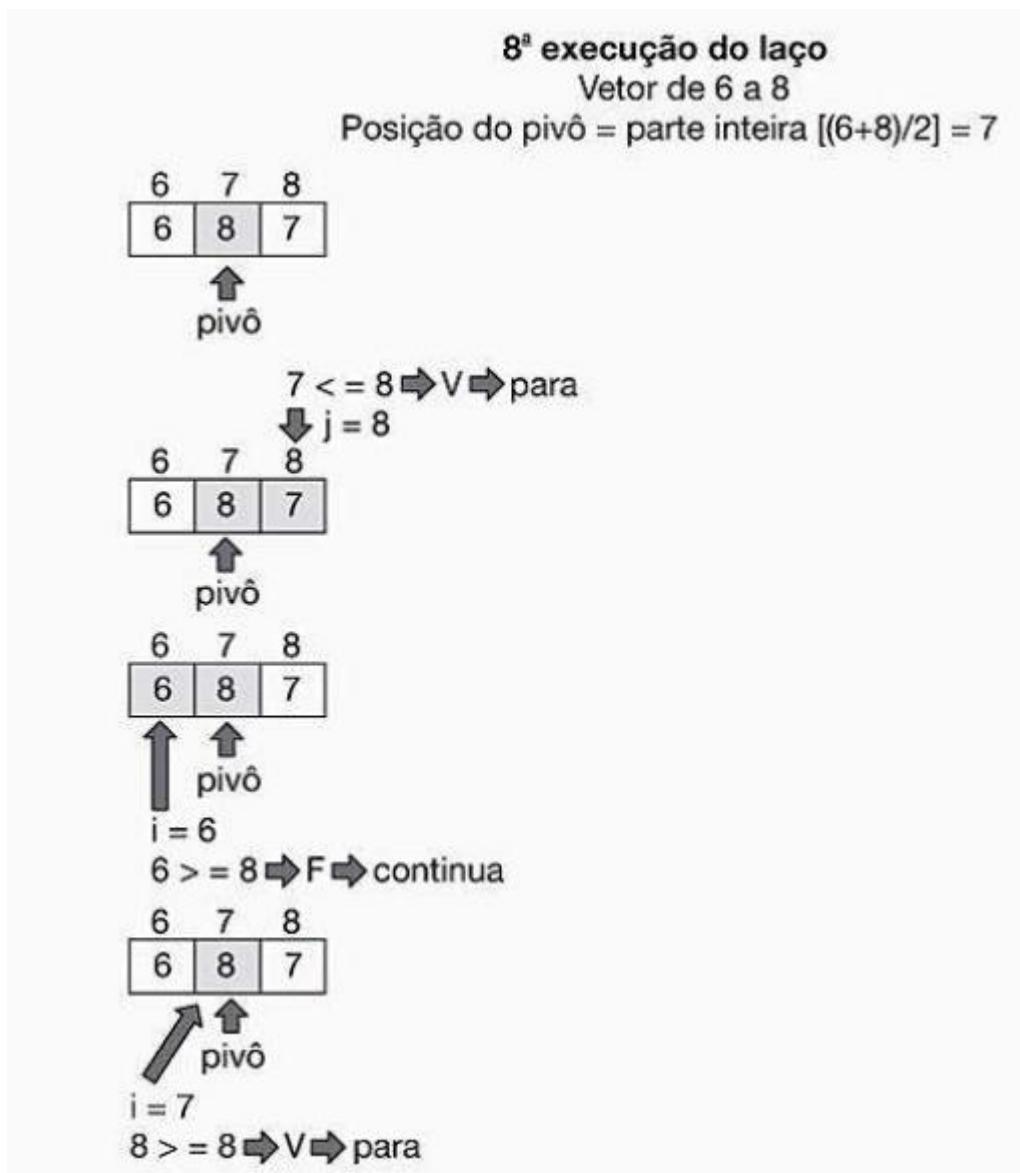
↑
pivô

6	7	8	9
6	8	7	9

↑
pivô

$i < j \Rightarrow 7 < 9 \Rightarrow V \Rightarrow \text{troca}$

Quick Sort



Quick Sort

9ª execução do laço

Vetor de 6 a 7

Posição do pivô = parte inteira $[(6+7)/2] = 7$

6	7
6	7

↑
pivô

$7 \leq 6 \Rightarrow F \Rightarrow \text{continua}$

↓ $j = 7$

6	7
6	7

↑
pivô

$6 \leq 6 \Rightarrow V \Rightarrow \text{para}$

↓ $j = 6$

6	7
6	7

↑
pivô

6	7
6	7

↑
pivô

$i = 6$

$6 \geq 6 \Rightarrow V \Rightarrow \text{para}$

6	7
6	7

↑
pivô

$i < j \Rightarrow 6 < 6 \Rightarrow F \Rightarrow \text{não troca}$

Vetor ordenado

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

Quick Sort

```
quickSort.cpp x
1  #include <iostream>
2  #include <stdlib.h>
3  #include <string>
4
5  #define TAM 10
6
7  using namespace std;
8
9  void imprimeVetor(int vetor[]){
10     int i;
11     cout << endl;
12     for (i = 0; i < TAM; i++){
13         cout << " |" << vetor[i] << "| ";
14     }
15 }
16
```

Quick Sort

```
17 void quickSort(int vetor[TAM], int inicio, int fim){
18     int pivo, esq, dir, meio, aux;
19
20     //limites da esquerda e direita da região analisada
21     esq = inicio;
22     dir = fim;
23
24     //ajustando auxiliares do centro
25     meio = (int) ((esq + dir) / 2);
26     pivo = vetor[meio];
27
28     while(dir > esq){
29         while(vetor[esq] < pivo){
30             esq = esq + 1;
31         }
32
33         while (vetor[dir] > pivo){
34             dir = dir - 1;
35         }
36     }
```

Quick Sort

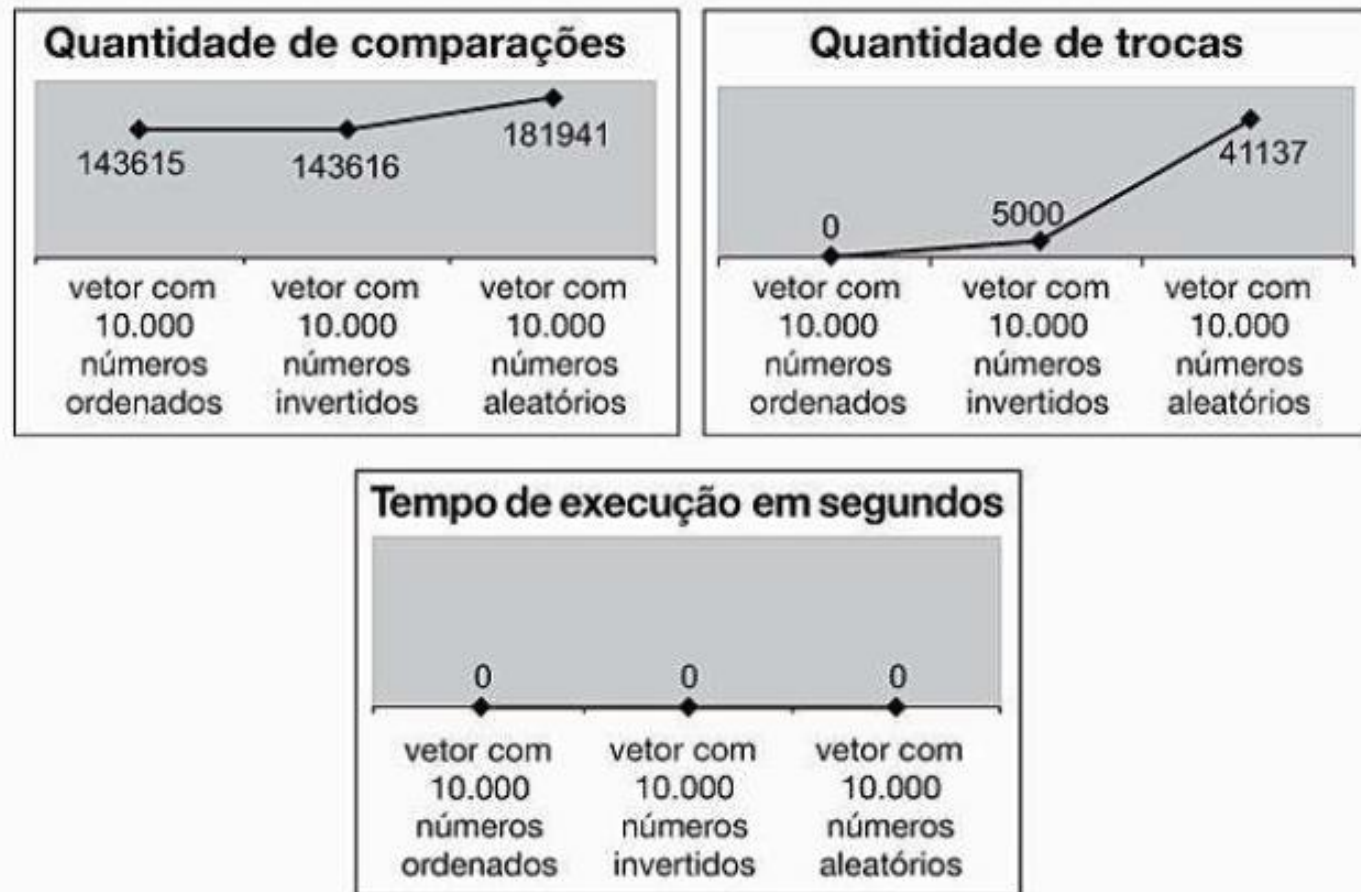
```
37  if (esq <= dir){
38      //realiza uma troca
39      aux = vetor[esq];
40      vetor[esq] = vetor[dir];
41      vetor[dir] = aux;
42
43      //faz os limites laterais caminharem para o centro
44      esq = esq + 1;
45      dir = dir - 1;
46  }
47  }
48
49  //recursão para continuar ordenando
50  if (inicio < dir){
51      quickSort(vetor, inicio, dir);
52  }
53
54  if (esq < fim){
55      quickSort(vetor, esq, fim);
56  }
57
58  }
59
```

Quick Sort

```
60 int main() {  
61  
62     int vetor[TAM] = {10,9,8,7,6,5,4,3,2,1};  
63  
64     imprimeVetor(vetor);  
65     cout << endl;  
66  
67     quickSort(vetor, 0, TAM);  
68     imprimeVetor(vetor);  
69  
70 }
```

Quick Sort

- Gráfico de desempenho:



Exercício

- Faça um programa em c++ que cadastre 10 números, ordene-os e em seguida encontre e mostre:
 - O menor número e quantas vezes ele aparece no vetor;
 - O maior número e quantas vezes ele aparece no vetor.

Referência desta aula

- <https://www.devmedia.com.br/algoritmos-de-ordenacao-analise-e-comparacao/28261>
- <https://www.devmedia.com.br/algoritmos-de-ordenacao/2622>
- <https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao/>
- <http://www.cplusplus.com/reference/>
- Ascencio, A. F. G. (2010). Estruturas De Dados: ALGORITMOS, ANÁLISE DA COMPLEXIDADE E IMPLEMENTAÇÃO. Brasil: PEARSON BRASIL.

Obrigado

Rodrigo