

**ACH2002**

## **Aula 14**

# **Algoritmos de Ordenação: estabilidade e alg. inserção, seleção direta e bolha**

Profa. Arianne Machado Lima

# Motivação

- Grande parte das operações de Sistemas de Informações é constituída por buscas em bases de dados.
- Exemplos?

# Motivação

- Grande parte das operações de Sistemas de Informações é constituída por buscas em bases de dados.
- Exemplos?
  - consultar saldo bancário, fornecendo número da conta corrente;
  - consultar nota no sistema Júpiter, fornecendo número USP;
  - consultar preço de um livro em uma loja, fornecendo seu código.

# Motivação

Para essas operações, os dados devem estar ordenados.

- Algoritmos de ordenação constituem uma classe muito estudada de algoritmos.
- Por quê?

# Motivação

Para essas operações, os dados devem estar ordenados.

- Algoritmos de ordenação constituem uma classe muito estudada de algoritmos.
  - é impossível pensar em buscas sem ordenação;
  - buscas exigem que os dados estejam organizados;
  - volume de dados geralmente é grande.

# O problema da ordenação

Um vetor  $v[0..n-1]$  é **crescente** se  $v[0] \leq v[1] \leq \dots \leq v[n-1]$ . O problema da ordenação de um vetor consiste no seguinte:

Rearranjar (ou seja, permutar) os elementos de um vetor  $v[0..n-1]$  de tal modo que ele se torne crescente.

Mas obviamente os mesmos algoritmos podem ser facilmente adaptados para realizar uma ordenação **decrescente**

# Algoritmos de ordenação

- Terceira parte da disciplina
- Já vimos em aulas anteriores:
  - **InsertionSort** (inserção direta)
  - **MergeSort** (ordenação por intercalação)
- Vamos revisar hoje outros que já devem ter visto:
  - **SelectionSort** (ordenação por seleção direta)
  - **BubbleSort** (ordenação pelo método da bolha)
- Vamos comparar os quatro levando em consideração:
  - Quanto usa de espaço
  - Complexidade de tempo (total, comparações e movimentações)
  - **Estabilidade**

Veremos outros até o final da disciplina, mas mesmo assim ainda não veremos todos!!!

# Algoritmos de ordenação

- Terceira parte da disciplina
- Já vimos em aulas anteriores:
  - InsertionSort (inserção direta)
  - MergeSort (ordenação por intercalação)
- Vamos revisar hoje outros que já devem ter visto:
  - SelectionSort (ordenação por seleção direta)
  - BubbleSort (ordenação pelo método da bolha)
- Vamos comparar os quatro levando em consideração:
  - Quanto usa de espaço
  - Complexidade de tempo (total, comparações e movimentações)
  - Estabilidade

Vamos agora falar um pouquinho disso



# Ordenação interna x externa

- Ordenação **interna**:
  - o arquivo/vetor cabe inteiramente na memória principal
  - **tema desta disciplina**
- Ordenação **externa**:
  - o arquivo/vetor NÃO cabe inteiramente na memória principal, e portanto a memória secundária (disco/SSD) precisa ser usada como apoio
  - Tema da disciplina ACH2024 (AED 2)


# Uso de espaço

- O uso econômico da memória disponível é um requisito primordial na ordenação interna.
- Métodos de ordenação *in situ* ou *in loco* (que ordenam no próprio vetor, usando no máximo algumas poucas variáveis a mais) são os preferidos.
- Métodos que utilizam listas encadeadas não são muito utilizados (ponteiro gasta espaço).

# Algoritmos de ordenação

- Terceira parte da disciplina
- Já vimos em aulas anteriores:
  - InsertionSort (inserção direta)
  - MergeSort (ordenação por intercalação)
- Vamos revisar hoje outros que já devem ter visto:
  - SelectionSort (ordenação por seleção direta)
  - BubbleSort (ordenação pelo método da bolha)
- Vamos levar em consideração nas comparações:
  - Quanto usa de espaço
  - Complexidade de tempo (total, comparações e movimentações)
  - Estabilidade

Vamos agora falar um  
pouquinho disso



# Algoritmos de ordenação

- Ordenação utilizando um determinado campo chave

```
typedef long TipoChave;  
typedef struct Tipoltem {  
    TipoChave Chave;  
    /* outros componentes */  
} Tipoltem;
```

- Normalmente usaremos vetores com apenas um valor (ao invés de uma estrutura) para simplificar o problema e focar apenas na lógica de ordenação
- Mas na prática, quando as estruturas (registros) são muito grandes, minimizar movimentações é interessante. Por isso....

# Complexidade de tempo de algoritmos de ordenação

Para um vetor de tamanho  $n$ , a complexidade de tempo pode ser analisada pela:

- Complexidade total -  $T(n)$ : número de operações (como fizemos até agora)
- Número de comparações (entre as chaves) –  $C(n)$
- Número de movimentações de registros -  $M(n)$

# Algoritmos de ordenação

- Terceira parte da disciplina
- Já vimos em aulas anteriores:
  - InsertionSort (inserção direta)
  - MergeSort (ordenação por intercalação)
- Vamos revisar hoje outros que já devem ter visto:
  - SelectionSort (ordenação por seleção direta)
  - BubbleSort (ordenação pelo método da bolha)
- Vamos levar em consideração nas comparações:
  - Quanto usa de espaço
  - Complexidade de tempo (total, comparações e movimentações)
  - Estabilidade ← Vamos agora falar um pouquinho disso

# Estabilidade: motivação

Suponha que você quer ordenar uma planilha por um determinado campo, ex. Nome:

Nome	Idade	Endereço	...
Fulano da Silva	22	aaaa	
Beltrano Siqueira	22	bbbb	
Jurubeba Leão do Norte	34	cccc	
Lilica da Cunha	22	dddd	
Ciclano da Fonseca	34	eeee	

# Estabilidade: motivação

Suponha que você quer ordenar uma planilha por um determinado campo, ex. Nome:

Nome	Idade	Endereço	...	Nome	Idade	Endereço	...
Fulano da Silva	22	aaaa		Beltrano Siqueira	22	bbbb	
Beltrano Siqueira	22	bbbb		Ciclano da Fonseca	34	eeee	
Jurubeba Leão do Norte	34	cccc		Fulano da Silva	22	aaaa	
Lilica da Cunha	22	dddd		Jurubeba Leão do Norte	34	cccc	
Ciclano da Fonseca	34	eeee		Lilica da Cunha	22	dddd	



# Estabilidade: motivação

Suponha que você quer ordenar uma planilha por um determinado campo, ex. Nome:

Nome	Idade	Endereço	...	Nome	Idade	Endereço	...
Fulano da Silva	22	aaaa		Beltrano Siqueira	22	bbbb	
Beltrano Siqueira	22	bbbb		Ciclano da Fonseca	34	eeee	
Jurubeba Leão do Norte	34	cccc		Fulano da Silva	22	aaaa	
Lilica da Cunha	22	dddd		Jurubeba Leão do Norte	34	cccc	
Ciclano da Fonseca	34	eeee		Lilica da Cunha	22	dddd	

Agora você quer ordenar por **outro** campo (ex. Idade) mantendo a ordenação por nome no caso de empate de idade (ou seja, **sem alterar a ordem relativa entre os empates**)

Nome	Idade	Endereço	...
Beltrano Siqueira	22	bbbb	
Fulano da Silva	22	aaaa	
Lilica da Cunha	22	dddd	
Ciclano da Fonseca	34	eeee	
Jurubeba Leão do Norte	34	cccc	

# Estabilidade: motivação

Suponha que você quer ordenar uma planilha por um determinado campo, ex. Nome:

Nome	Idade	Endereço	...	Nome	Idade	Endereço	...
Fulano da Silva	22	aaaa		Beltrano Siqueira	22	bbbb	
Beltrano Siqueira	22	bbbb		Ciclano da Fonseca	34	eeee	
Jurubeba Leão do Norte	34	cccc		Fulano da Silva	22	aaaa	
Lilica da Cunha	22	dddd		Jurubeba Leão do Norte	34	cccc	
Ciclano da Fonseca	34	eeee		Lilica da Cunha	22	dddd	

Agora você quer ordenar por **outro** campo (ex. Idade) mantendo a ordenação por nome no caso de empate de idade (ou seja, **sem alterar a ordem relativa entre os empates**)

Nome	Idade	Endereço	...
Beltrano Siqueira	22	bbbb	
Fulano da Silva	22	aaaa	
Lilica da Cunha	22	dddd	
Ciclano da Fonseca	34	eeee	
Jurubeba Leão do Norte	34	cccc	

Ou seja, você quer um algoritmo de ordenação **estável** !

# Estabilidade

**Definição:** Um algoritmo de ordenação é **estável** se não altera a posição relativa de elementos que têm um mesmo valor.

Ex.:

vetor original:	444	555	666	777	333	222 <sub>1</sub>	111	222 <sub>2</sub>	888
vetor ordenado:	111	222 <sub>1</sub>	222 <sub>2</sub>	333	444	555	666	777	888

Figura 8.3: Ordenação estável. O vetor original tem dois elementos com valor 222 (índices <sub>1</sub> e <sub>2</sub> são usados para distinguir o primeiro do segundo). No vetor ordenado, o primeiro destes elementos continua à frente do segundo.

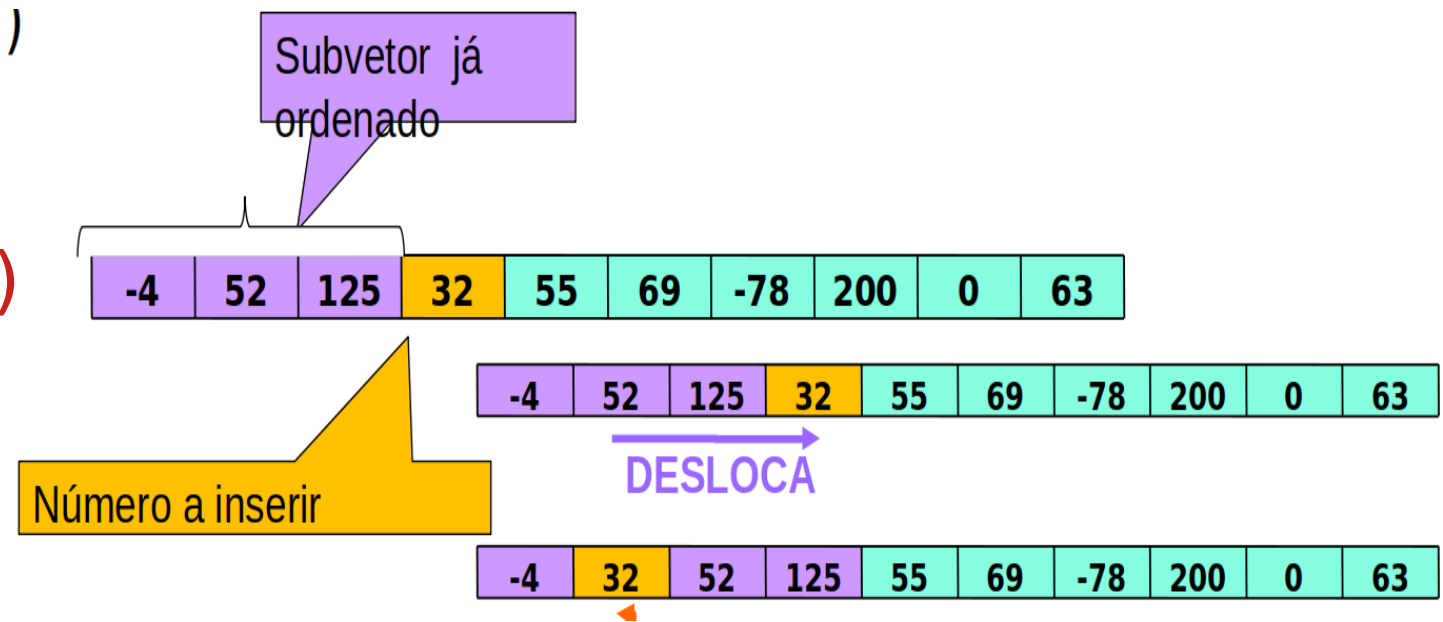
# Estabilidade

- Alguns dos métodos de ordenação mais eficientes não são estáveis.

# Algoritmos de ordenação

- Algoritmos elementares (simples)
  - Inserção Direta (InsertionSort)
  - Seleção Direta (SelectionSort)
  - Método da Bolha (BubbleSort)

# Insertion sort (inserção direta)



insercao (n, v)

**para**  $j \leftarrow 2$  **até** tamanho de v **faça**

chave  $\leftarrow v[j]$ ;

// ordenando elementos à esquerda

$i \leftarrow j - 1$

**enquanto**  $i > 0$  **e**  $v[i] > \text{chave}$  **faça**

$v[i+1] \leftarrow v[i]$

$i \leftarrow i - 1$

**fim enquanto**

$v[i+1] \leftarrow \text{chave}$

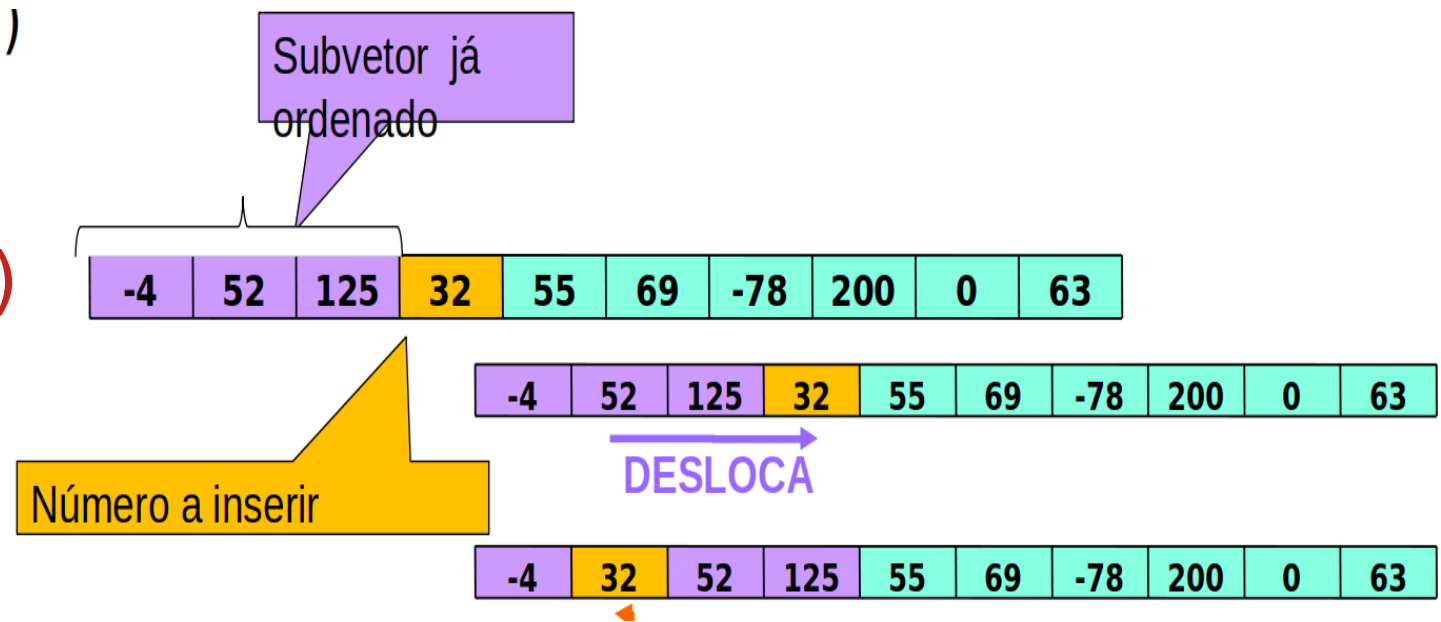
**fim para**

**Complexidade de tempo:**

*In loco?*:

**Estável:**

# Insertion sort (inserção direta)



insercao (n, v)

**para** j ← 2 **até** tamanho de v **faça**

chave ← v[j];

// ordenando elementos à esquerda

i ← j - 1

**enquanto** i > 0 **e** v[i] > chave **faça**

v[i+1] ← v[i]

i ← i - 1

**fim enquanto**

v[i+1] ← chave

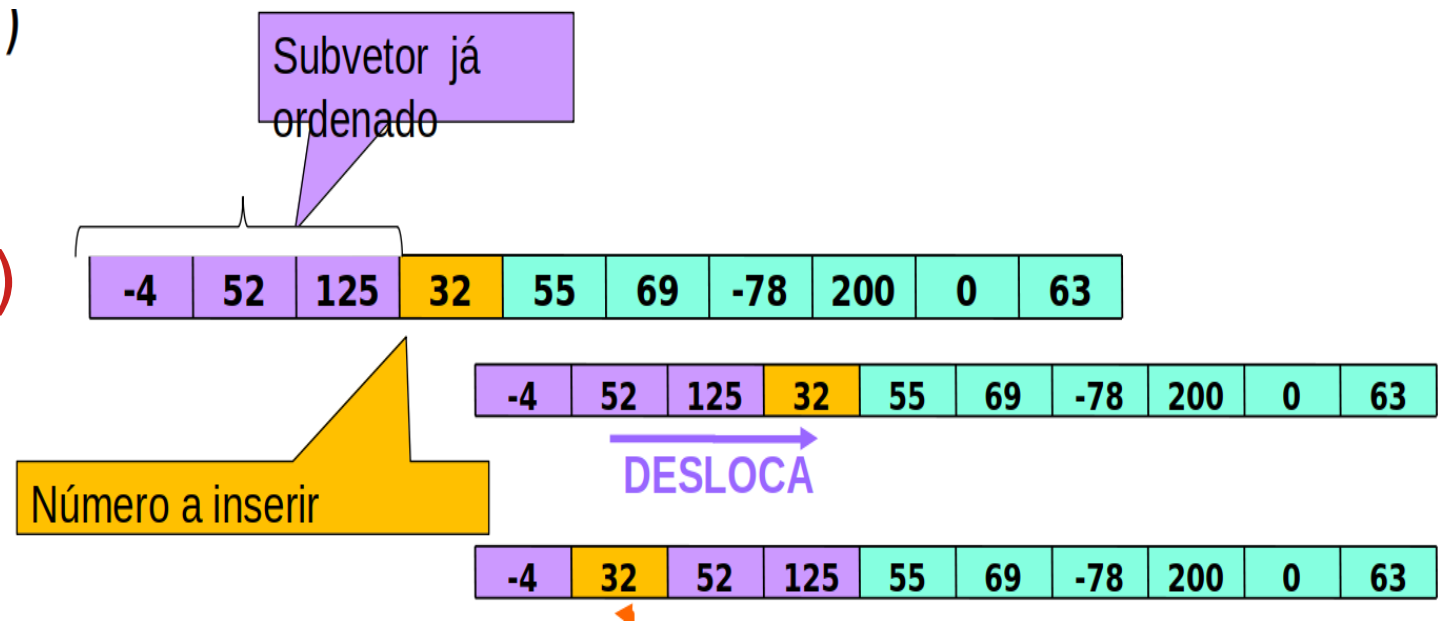
**fim para**

Complexidade de tempo:  $O(n^2)$

*In loco?*:

Estável:

# Insertion sort (inserção direta)



insercao (n, v)

**para** j ← 2 **até** tamanho de v **faça**

chave ← v[j];

// ordenando elementos à esquerda

i ← j - 1

**enquanto** i > 0 **e** v[i] > chave **faça**

v[i+1] ← v[i]

i ← i - 1

**fim enquanto**

v[i+1] ← chave

**fim para**

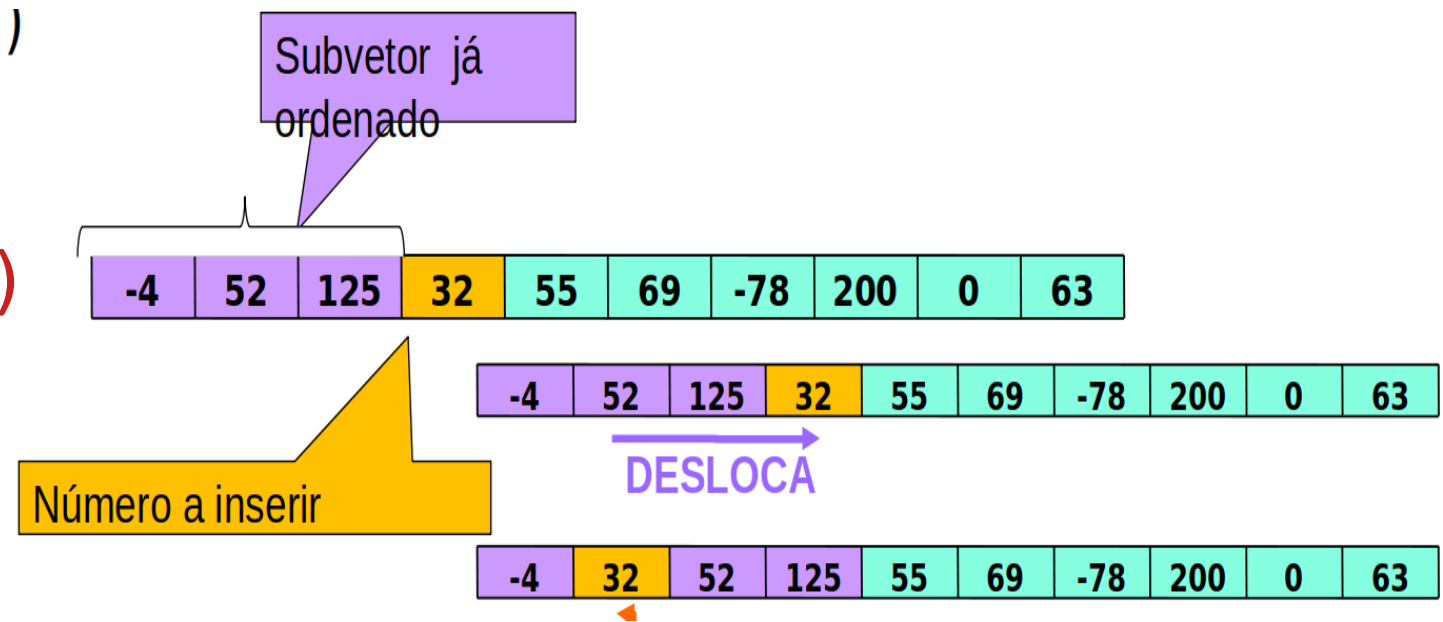
Complexidade de tempo:  $O(n^2)$

*In loco?*: **SIM!**

Estável:



# Insertion sort (inserção direta)



insercao (n, v)

**para**  $j \leftarrow 2$  **até** tamanho de v **faça**

chave  $\leftarrow v[j]$ ;

// ordenando elementos à esquerda

$i \leftarrow j - 1$

**enquanto**  $i > 0$  **e**  $v[i] > \text{chave}$  **faça**

$v[i+1] \leftarrow v[i]$

$i \leftarrow i - 1$

**fim enquanto**

$v[i+1] \leftarrow \text{chave}$

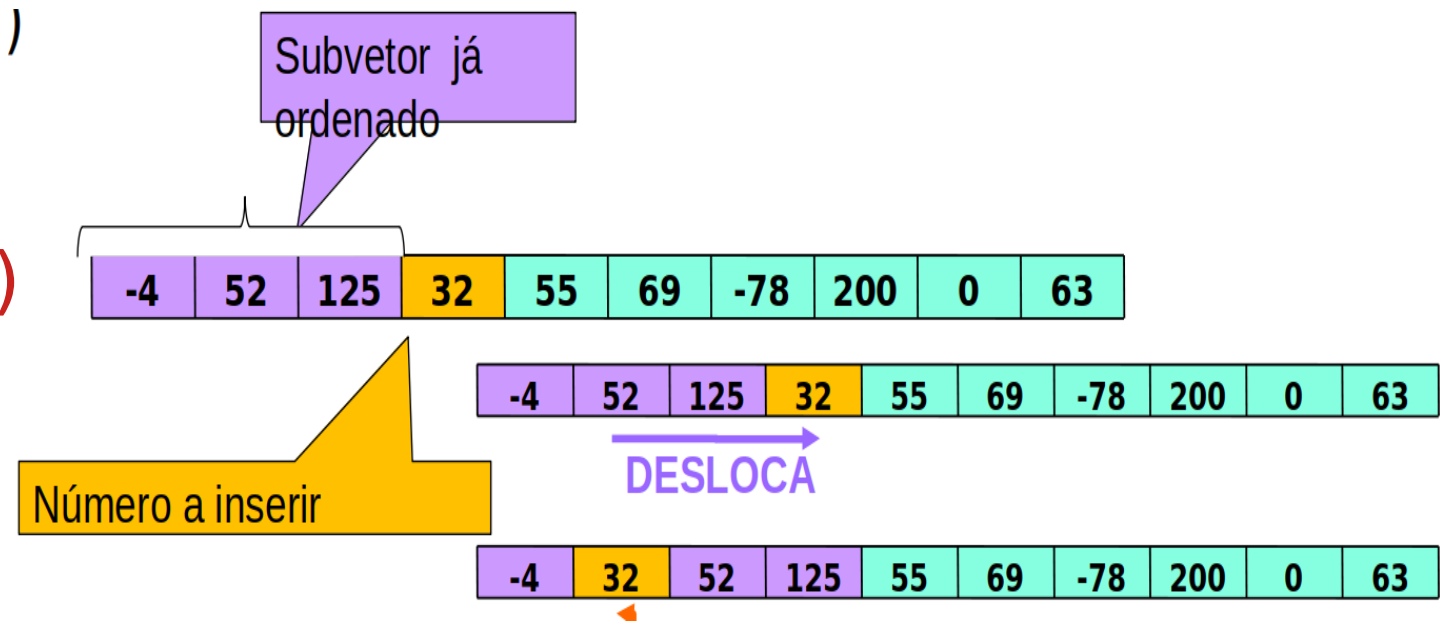
**fim para**

Complexidade de tempo:  $O(n^2)$

*In loco?*: **SIM!**

Estável: **Sim!** O que o torna estável?

# Insertion sort (inserção direta)



insercao (n, v)

**para** j ← 2 **até** tamanho de v **faça**

chave ← v[j];

// ordenando elementos à esquerda

i ← j - 1

**enquanto** i > 0 **e** v[i] > chave **faça**

v[i+1] ← v[i]

i ← i - 1

**fim enquanto**

v[i+1] ← chave

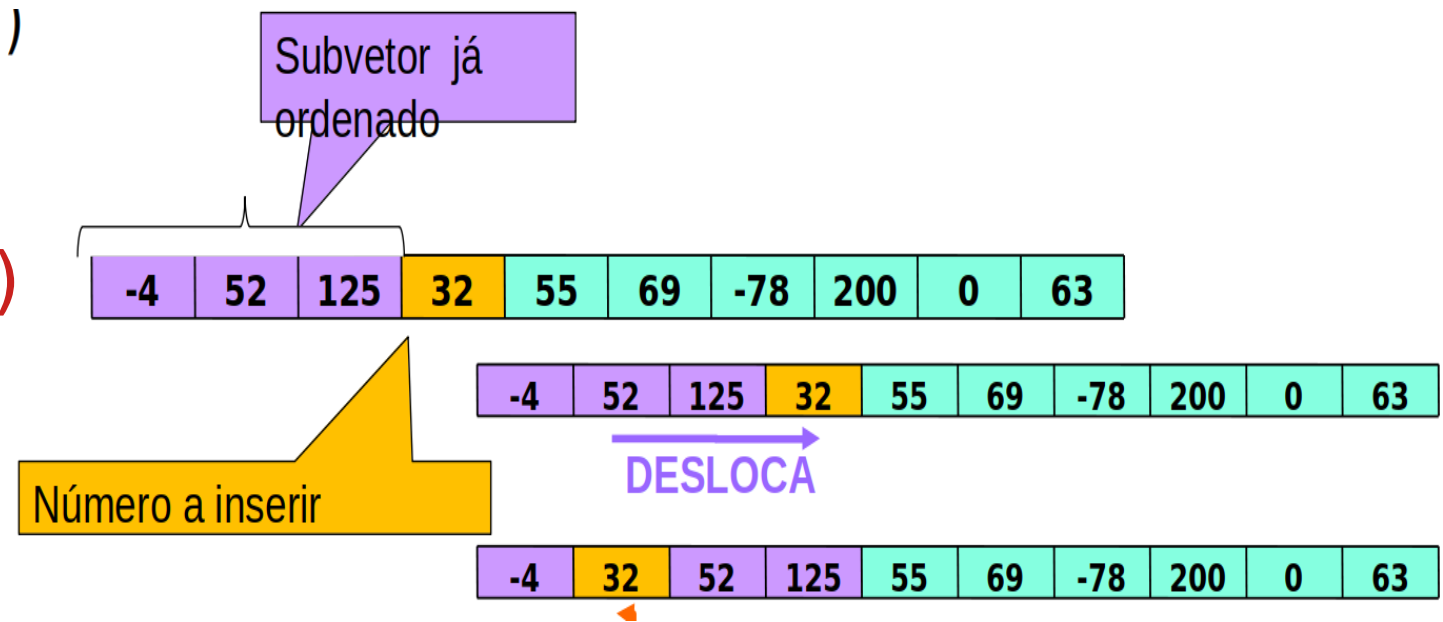
**fim para**

**Complexidade de tempo:**  $O(n^2)$

**In loco?:** **SIM!**

**Estável:** **Sim!** O que o torna estável? O menor elemento do subvetor à direita é inserido APÓS os elementos que já foram ordenados, e o processo é realizado da esquerda para a direita.

# Insertion sort (inserção direta)



insercao (n, v)

**para** j ← 2 **até** tamanho de v **faça**

chave ← v[j];

// ordenando elementos à esquerda

i ← j - 1

**enquanto** i > 0 **e** v[i] > chave **faça**

v[i+1] ← v[i]

i ← i - 1

**fim enquanto**

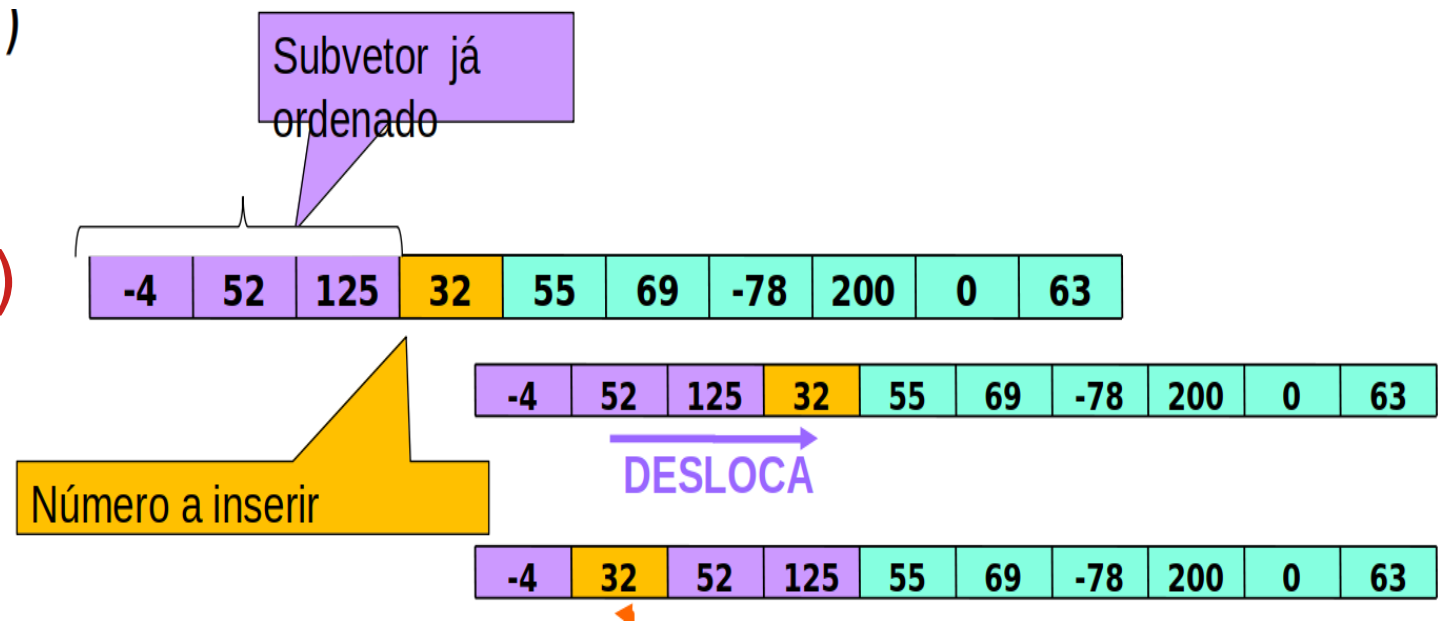
v[i+1] ← chave

**fim para**

Complexidade de tempo:  $O(n^2)$

Complexidade de tempo no melhor caso:

# Insertion sort (inserção direta)



insercao (n, v)

**para**  $j \leftarrow 2$  **até** tamanho de v **faça**

chave  $\leftarrow v[j]$ ;

// ordenando elementos à esquerda

$i \leftarrow j - 1$

**enquanto**  $i > 0$  e  $v[i] > \text{chave}$  **faça**

$v[i+1] \leftarrow v[i]$

$i \leftarrow i - 1$

**fim enquanto**

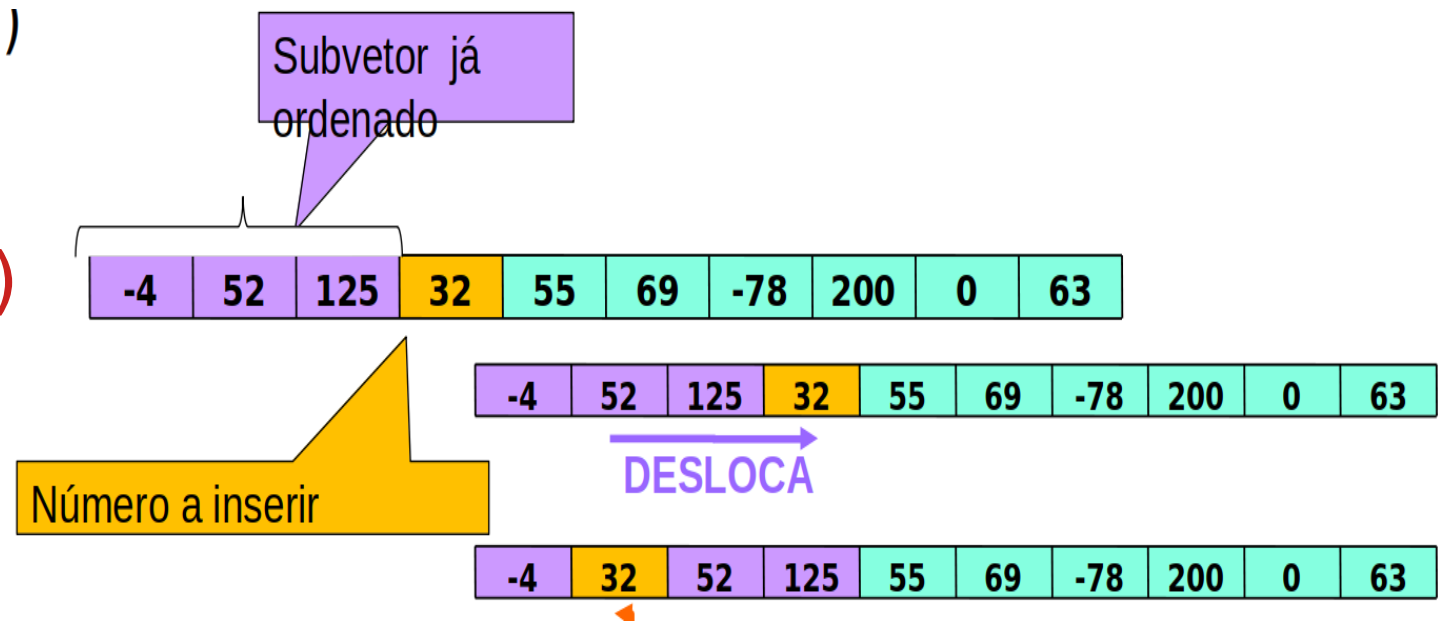
$v[i+1] \leftarrow \text{chave}$

**fim para**

Complexidade de tempo:  $O(n^2)$

Complexidade de tempo no melhor caso:  
 $O(n)$  quando o vetor já está ordenado

# Insertion sort (inserção direta)



insercao (n, v)

**para**  $j \leftarrow 2$  **até** tamanho de v **faça**

chave  $\leftarrow v[j]$ ;

// ordenando elementos à esquerda

$i \leftarrow j - 1$

**enquanto**  $i > 0$  **e**  $v[i] > \text{chave}$  **faça**

$v[i+1] \leftarrow v[i]$

$i \leftarrow i - 1$

**fim enquanto**

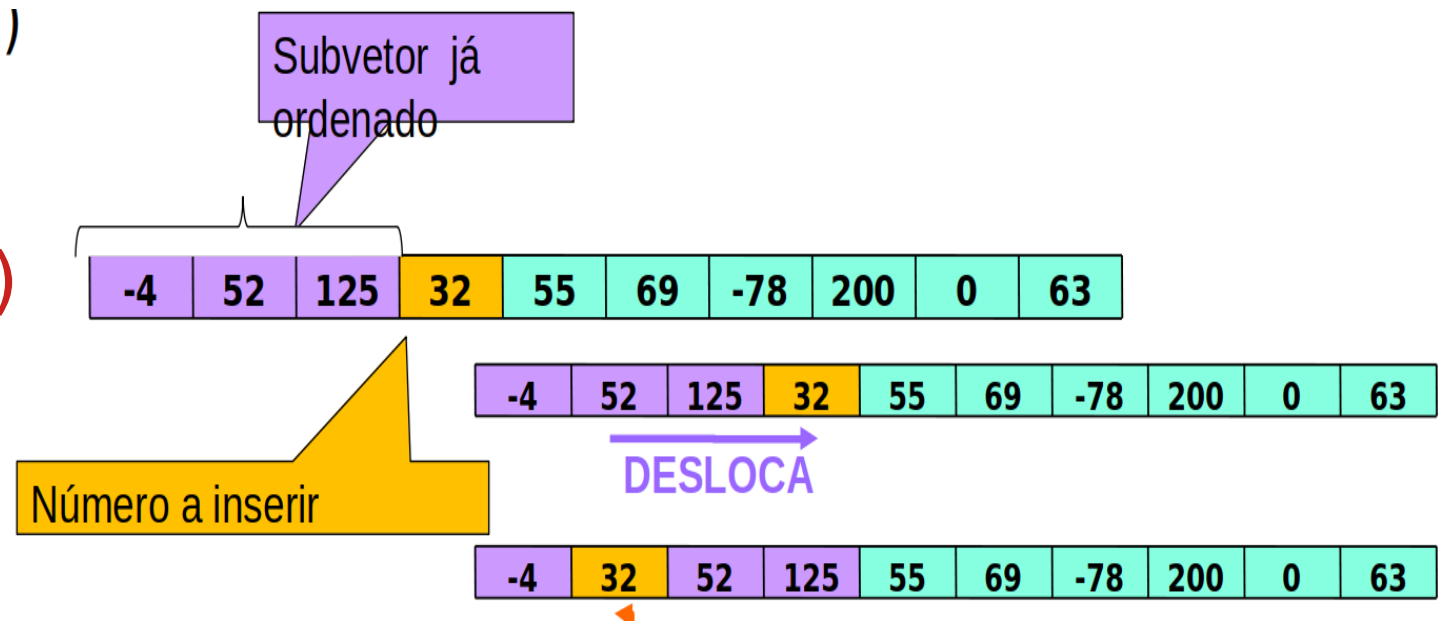
$v[i+1] \leftarrow \text{chave}$

**fim para**

Número de comparações entre chaves -  $C(n)$ :  
?

Note que além das comparações entre chaves  
temos a comparação  $i > 0$   
Dá para eliminá-la... como?

# Insertion sort (inserção direta)



insercao (n, v)

**para**  $j \leftarrow 2$  **até** tamanho de v **faça**

chave  $\leftarrow v[j]$ ;

// ordenando elementos à esquerda

$i \leftarrow j - 1$

**enquanto**  $i > 0$  **e**  $v[i] > \text{chave}$  **faça**

$v[i+1] \leftarrow v[i]$

$i \leftarrow i - 1$

**fim enquanto**

$v[i+1] \leftarrow \text{chave}$

**fim para**

Número de comparações entre chaves -  $C(n)$ :  
?

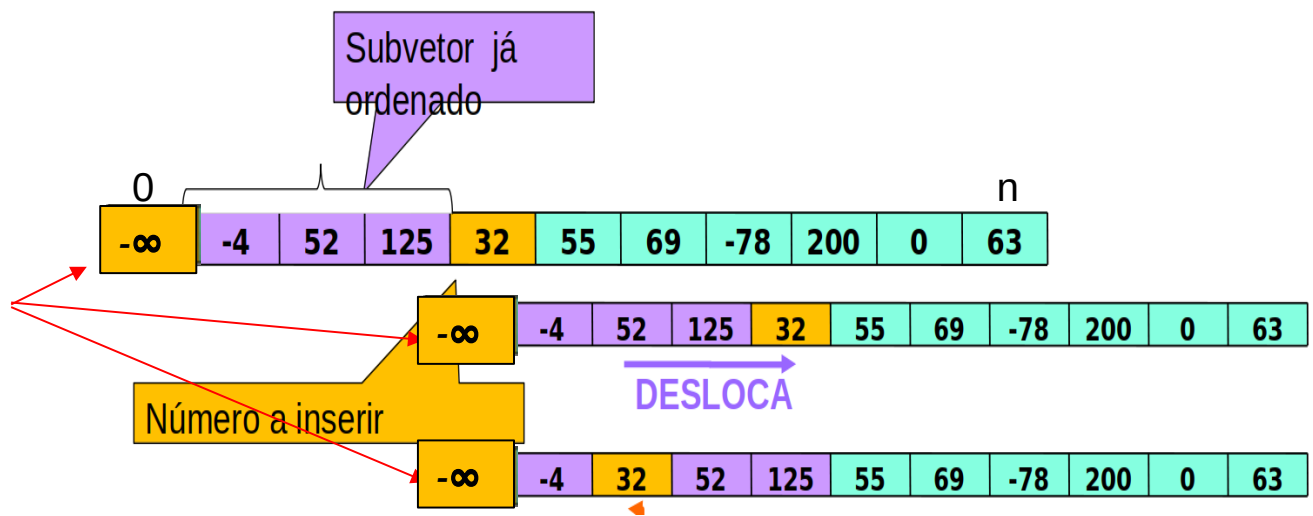
Note que além das comparações entre chaves temos a comparação  $i > 0$

Dá para eliminá-la... como?

Usando uma **sentinela**! - colocando na primeira posição (extra) um valor “-infinito” (ex: INT\_MIN)

O código do slide a seguir está em C, por isso assumiremos que o vetor começa em 0 (mas terá uma posição a mais por conta da sentinela).

# Insertion sort (inserção direta)



```
#include <limits.h>
```

```
void insercao(int n, int [] v) {
```

```
    int i, j, chave;
```

```
    v[0] = INT_MIN; //sentinela!!!
```

```
    for (j = 2; j <= n; j++) {
```

```
        chave = v[j];
```

```
        // ordenando elementos à esquerda
```

```
        i = j - 1;
```

```
        while (v[i] > chave){
```

```
            v[i+1] = v[i];
```

```
            i = i - 1;
```

```
        }
```

```
        v[i+1] = chave;
```

```
    }
```

## Número de comparações entre chaves - $C(n)$ :

Para análise de caso médio é útil entender o que ocorre em **cada** iteração:

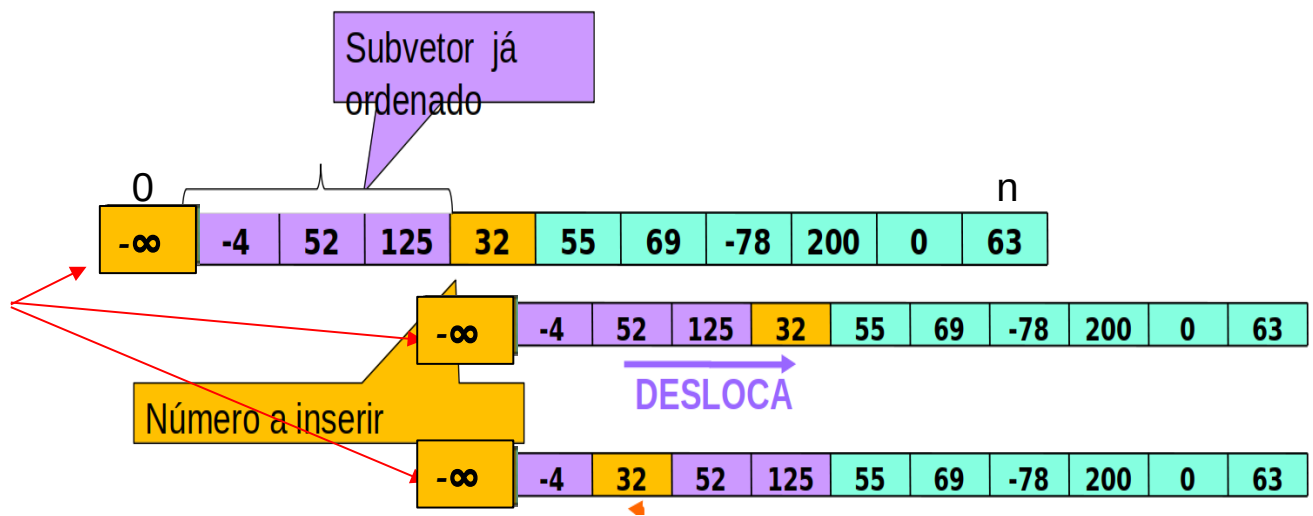
Na iteração de um dado valor  $j$ :

Melhor caso :  $C_j(n) =$

Pior caso :  $C_j(n) =$

Caso médio :  $C_j(n) =$

# Insertion sort (inserção direta)



```
#include <limits.h>
```

```
void insercao(int n, int [] v) {
```

```
    int i, j, chave;
```

```
    v[0] = INT_MIN; //sentinela!!!
```

```
    for (j = 2; j <= n; j++) {
```

```
        chave = v[j];
```

```
        // ordenando elementos à esquerda
```

```
        i = j - 1;
```

```
        while (v[i] > chave){
```

```
            v[i+1] = v[i];
```

```
            i = i - 1;
```

```
        }
```

```
        v[i+1] = chave;
```

```
    }
```

## Número de comparações entre chaves - $C(n)$ :

Para análise de caso médio é útil entender o que ocorre em **cada** iteração:

Na iteração de um dado valor  $j$ :

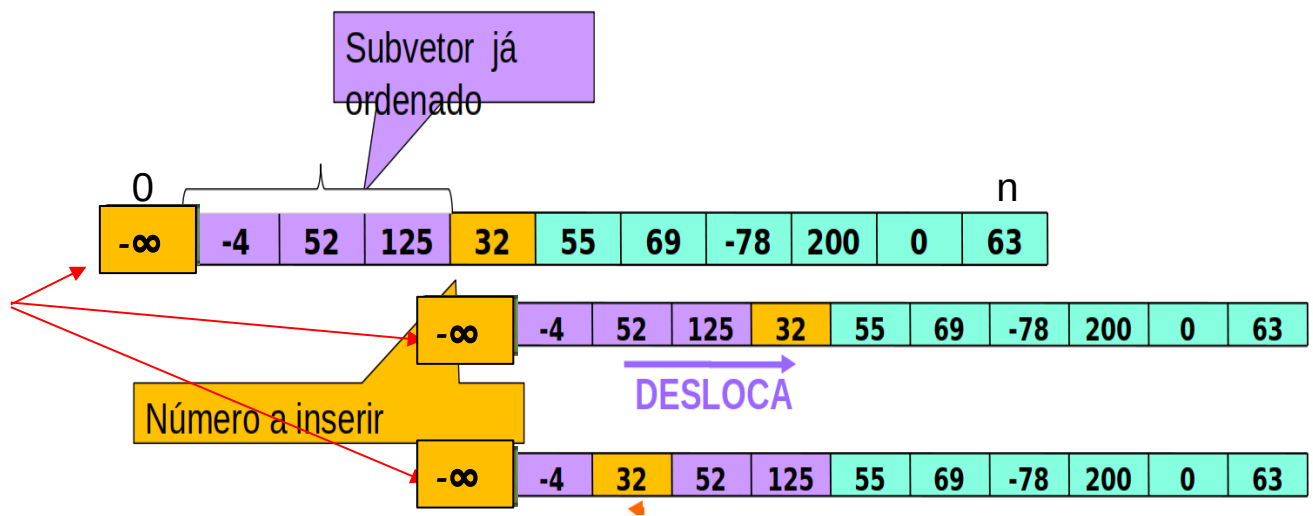
Melhor caso :  $C_j(n) = 1$

Pior caso :  $C_j(n) =$

Caso médio :  $C_j(n) =$



# Insertion sort (inserção direta)



```
#include <limits.h>
```

```
void insercao(int n, int [] v) {
```

```
    int i, j, chave;
```

```
    v[0] = INT_MIN; //sentinela!!!
```

```
    for (j = 2; j <= n; j++) {
```

```
        chave = v[j];
```

```
        // ordenando elementos à esquerda
```

```
        i = j - 1;
```

```
        while (v[i] > chave){
```

```
            v[i+1] = v[i];
```

```
            i = i - 1;
```

```
        }
```

```
        v[i+1] = chave;
```

```
    }
```

## Número de comparações entre chaves - $C(n)$ :

Para análise de caso médio é útil entender o que ocorre em **cada** iteração:

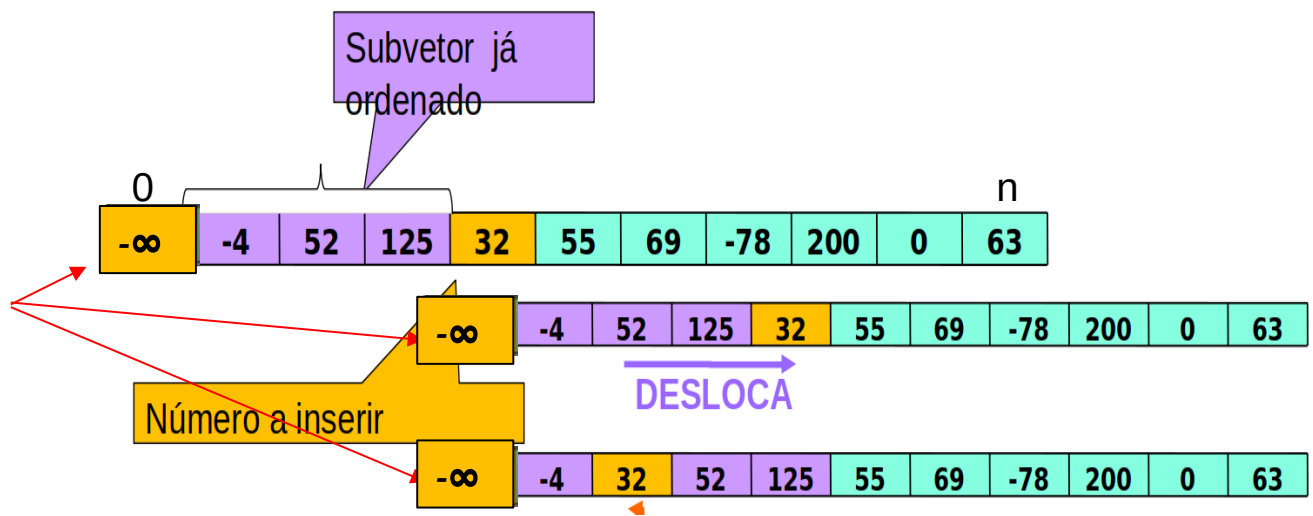
Na iteração de um dado valor  $j$ :

Melhor caso :  $C_j(n) = 1$

Pior caso :  $C_j(n) = j$  (j-1 números + sentinela)

Caso médio :  $C_j(n) =$

# Insertion sort (inserção direta)



```
#include <limits.h>
```

```
void insercao(int n, int [] v) {
```

```
    int i, j, chave;
```

```
    v[0] = INT_MIN; //sentinela!!!
```

```
    for (j = 2; j <= n; j++) {
```

```
        chave = v[j];
```

```
        // ordenando elementos à esquerda
```

```
        i = j - 1;
```

```
        while (v[i] > chave){
```

```
            v[i+1] = v[i];
```

```
            i = i - 1;
```

```
        }
```

```
        v[i+1] = chave;
```

```
    }
```

## Número de comparações entre chaves - $C(n)$ :

Para análise de caso médio é útil entender o que ocorre em **cada** iteração:

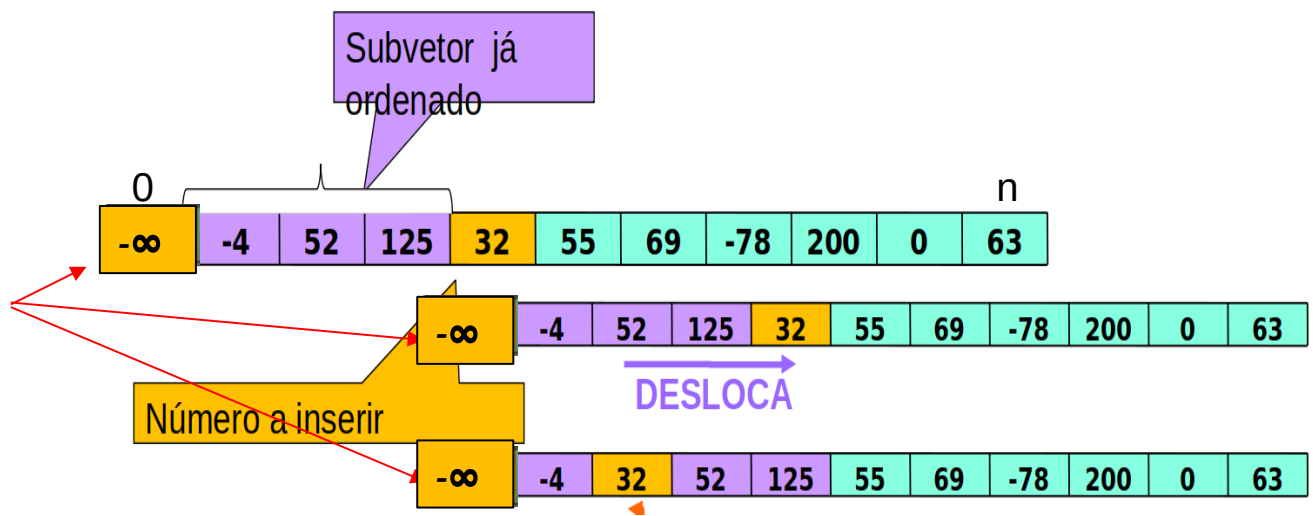
Na iteração de um dado valor  $j$ :

Melhor caso :  $C_j(n) = 1$

Pior caso :  $C_j(n) = j$  ( $j-1$  números + sentinela)

Caso médio :  $C_j(n) = \frac{1}{j}(1 + 2 + \dots + j) = \frac{j+1}{2}$

# Insertion sort (inserção direta)



```
#include <limits.h>
```

```
void insercao(int n, int [] v) {
```

```
    int i, j, chave;
```

```
    v[0] = INT_MIN; //sentinela!!!
```

```
    for (j = 2; j <= n; j++) {
```

```
        chave = v[j];
```

```
        // ordenando elementos à esquerda
```

```
        i = j - 1;
```

```
        while (v[i] > chave){
```

```
            v[i+1] = v[i];
```

```
            i = i - 1;
```

```
        }
```

```
        v[i+1] = chave;
```

```
    }
```

## Número de comparações entre chaves - $C(n)$ :

Para análise de caso médio é útil entender o que ocorre em cada iteração:

Na iteração de um dado valor  $j$ :

Melhor caso :  $C_j(n) = 1$

Pior caso :  $C_j(n) = j$  ( $j-1$  números + sentinela)

Caso médio :  $C_j(n) = \frac{1}{j}(1 + 2 + \dots + j) = \frac{j+1}{2}$

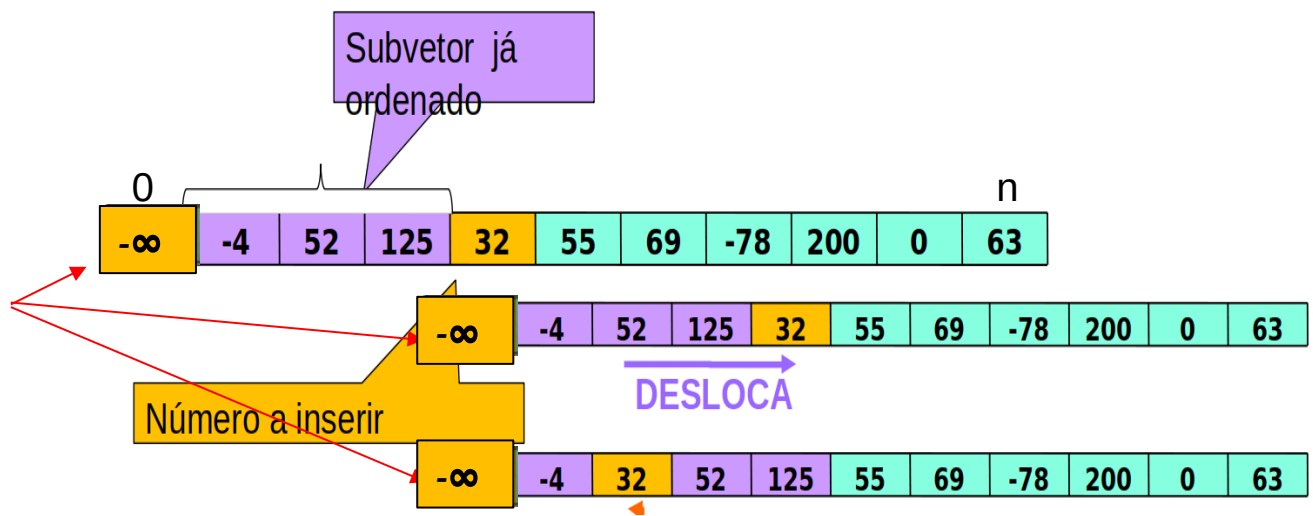
No total

Melhor caso :  $C(n) =$

Pior caso :  $C(n) =$

Caso médio :  $C(n) =$

# Insertion sort (inserção direta)



```
#include <limits.h>
```

```
void insercao(int n, int [] v) {
```

```
    int i, j, chave;
```

```
    v[0] = INT_MIN; //sentinela!!!
```

```
    for (j = 2; j <= n; j++) {
```

```
        chave = v[j];
```

```
        // ordenando elementos à esquerda
```

```
        i = j - 1;
```

```
        while (v[i] > chave){
```

```
            v[i+1] = v[i];
```

```
            i = i - 1;
```

```
        }
```

```
        v[i+1] = chave;
```

```
    }
```

## Número de comparações entre chaves - $C(n)$ :

Para análise de caso médio é útil entender o que ocorre em cada iteração:

Na iteração de um dado valor  $j$ :

Melhor caso :  $C_j(n) = 1$

Pior caso :  $C_j(n) = j$  ( $j-1$  números + sentinela)

Caso médio :  $C_j(n) = \frac{1}{j}(1 + 2 + \dots + j) = \frac{j+1}{2}$

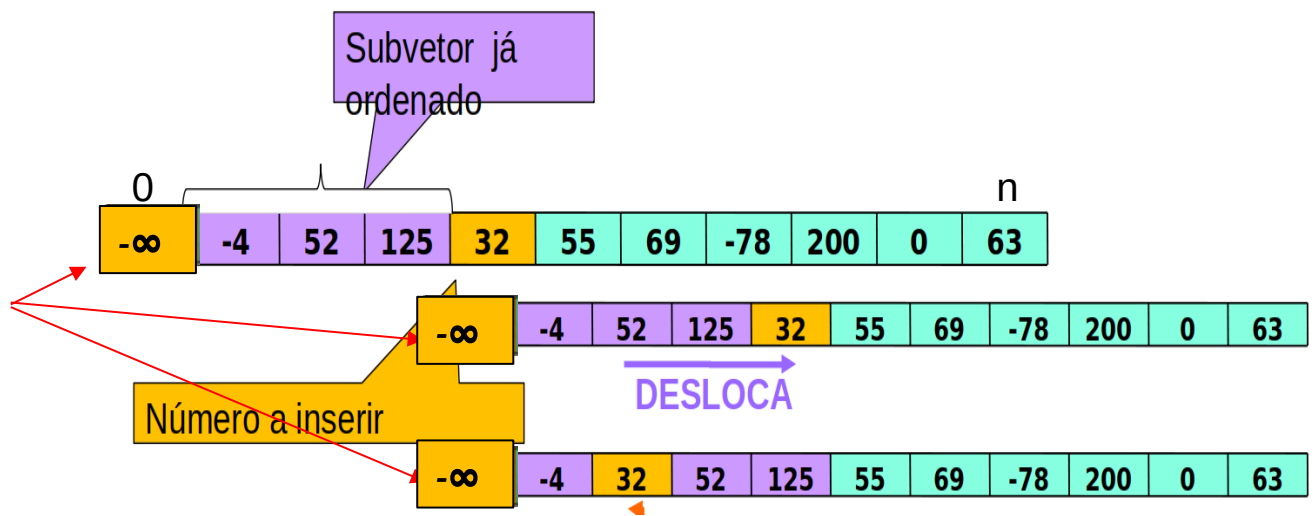
No total

Pois  $j$  começa em 2 Melhor caso :  $C(n) = (1 + 1 + \dots + 1) = n - 1$

Pior caso :  $C(n) =$

Caso médio :  $C(n) =$

# Insertion sort (inserção direta)



```
#include <limits.h>
```

```
void insercao(int n, int [] v) {
```

```
    int i, j, chave;
```

```
    v[0] = INT_MIN; //sentinela!!!
```

```
    for (j = 2; j <= n; j++) {
```

```
        chave = v[j];
```

```
        // ordenando elementos à esquerda
```

```
        i = j - 1;
```

```
        while (v[i] > chave){
```

```
            v[i+1] = v[i];
```

```
            i = i - 1;
```

```
        }
```

```
        v[i+1] = chave;
```

```
    }
```

## Número de comparações entre chaves - $C(n)$ :

Para análise de caso médio é útil entender o que ocorre em cada iteração:

Na iteração de um dado valor  $j$ :

Melhor caso :  $C_j(n) = 1$

Pior caso :  $C_j(n) = j$  ( $j-1$  números + sentinela)

Caso médio :  $C_j(n) = \frac{1}{j}(1 + 2 + \dots + j) = \frac{j+1}{2}$

No total

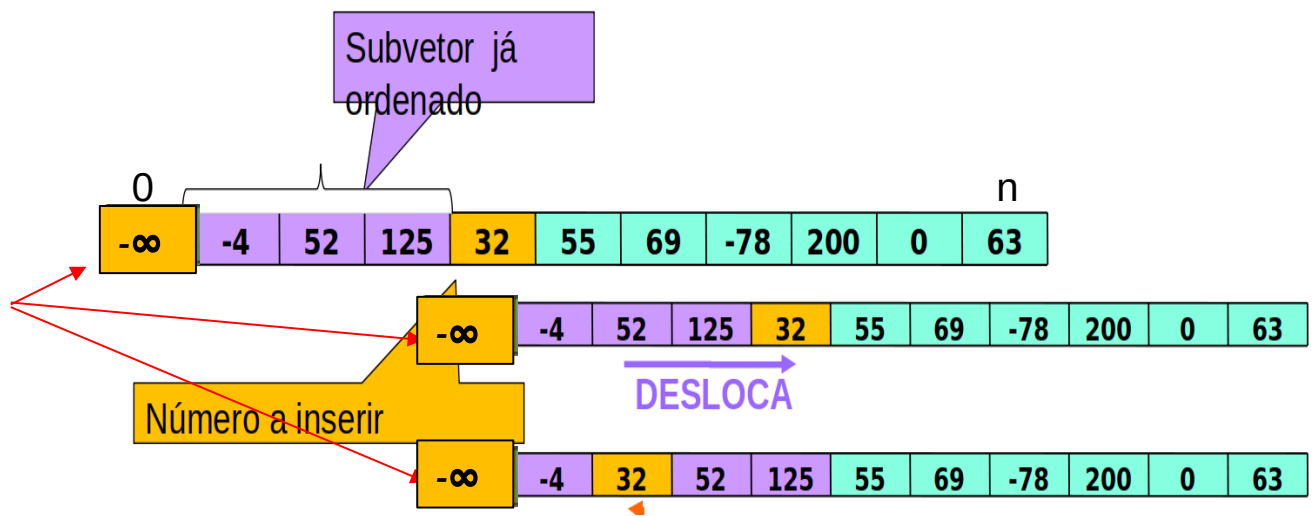
Melhor caso :  $C(n) = (1 + 1 + \dots + 1) = n - 1$

Pior caso :  $C(n) = (2 + 3 + \dots + n) = \frac{n^2}{2} + \frac{n}{2} - 1$

Caso médio :  $C(n) =$

Pois  $j$  começa em 2

# Insertion sort (inserção direta)



```
#include <limits.h>
```

```
void insercao(int n, int [] v) {
```

```
    int i, j, chave;
```

```
    v[0] = INT_MIN; //sentinela!!!
```

```
    for (j = 2; j <= n; j++) {
```

```
        chave = v[j];
```

```
        // ordenando elementos à esquerda
```

```
        i = j - 1;
```

```
        while (v[i] > chave){
```

```
            v[i+1] = v[i];
```

```
            i = i - 1;
```

```
        }
```

```
        v[i+1] = chave;
```

```
    }
```

## Número de comparações entre chaves - $C(n)$ :

Para análise de caso médio é útil entender o que ocorre em cada iteração:

Na iteração de um dado valor  $j$ :

Melhor caso :  $C_j(n) = 1$

Pior caso :  $C_j(n) = j$  ( $j-1$  números + sentinela)

Caso médio :  $C_j(n) = \text{média}(1, j) = \frac{j+1}{2}$

**No total** (para  $j$  de 2 a  $n$ ):

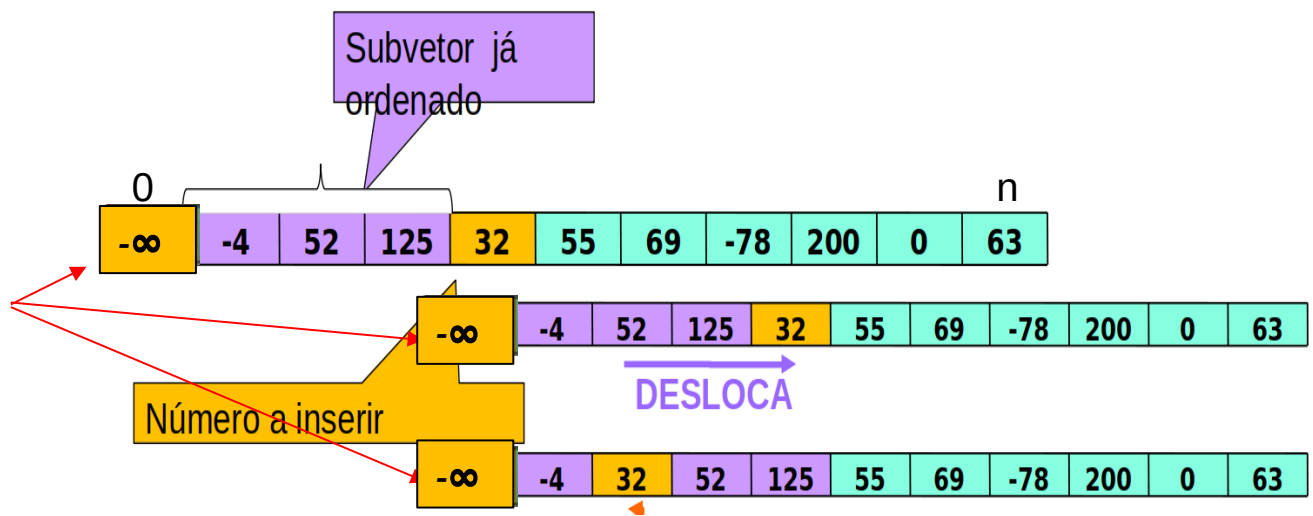
Melhor caso :  $C(n) = (1 + 1 + \dots + 1) = n - 1$

Pior caso :  $C(n) = (2 + 3 + \dots + n) = \frac{n^2}{2} + \frac{n}{2} - 1$

Caso médio :  $C(n) = \frac{1}{2}(3 + 4 + \dots + n + 1) = \frac{n^2}{4} + \frac{3n}{4} - 1$

$$\sum_{j=2}^n \frac{j+1}{2} = \frac{1}{2} \sum_{j=2}^n j+1$$

# Insertion sort (inserção direta)



```
#include <limits.h>
```

```
void insercao(int n, int [] v) {
```

```
    int i, j, chave;
```

```
    v[0] = INT_MIN; //sentinela!!!
```

```
    for (j = 2; j <= n; j++) {
```

```
        chave = v[j];
```

```
        // ordenando elementos à esquerda
```

```
        i = j - 1;
```

```
        while (v[i] > chave){
```

```
            v[i+1] = v[i];
```

```
            i = i - 1;
```

```
        }
```

```
        v[i+1] = chave;
```

```
    }
```

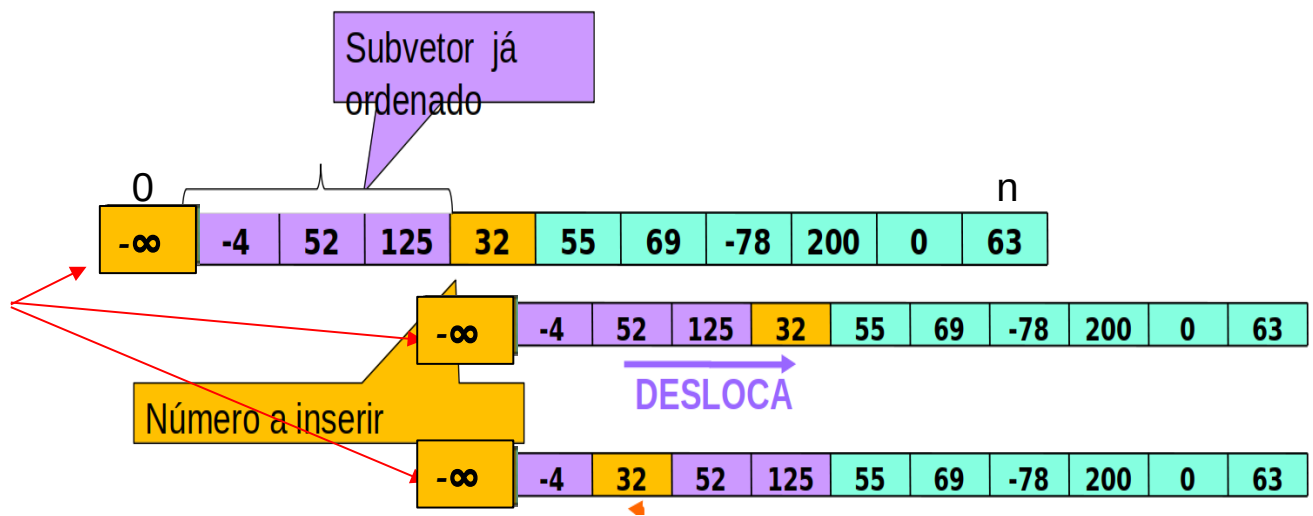
**Número de movimentações de registros -  $M(n)$ :**

Para análise de caso médio é útil entender o que ocorre em cada iteração:

Na iteração de um dado valor  $j$ :

**Melhor caso:  $M_j(n) = 2$**

# Insertion sort (inserção direta)



```
#include <limits.h>
void insercao(int n, int [] v) {
    int i, j, chave;
    v[0] = INT_MIN; //sentinela!!!
    for (j = 2; j <= n; j++) {
        chave = v[j];
        // ordenando elementos à esquerda
        i = j - 1;
        while (v[i] > chave){
            v[i+1] = v[i];
            i = i - 1;
        }
        v[i+1] = chave;
    }
}
```

## Número de movimentações de **registros** - $M(n)$ :

Para análise de caso médio é útil entender o que ocorre em cada iteração:

Na iteração de um dado valor  $j$ :

Melhor caso:  $M_j(n) = 2$

Pior caso:  $M_j(n) = (j-1)+2$

Caso médio:  $M_j(n) = \text{média}(2, j+1) = (j+3)/2$

**No total** (para  $j$  de 2 a  $n$ ):

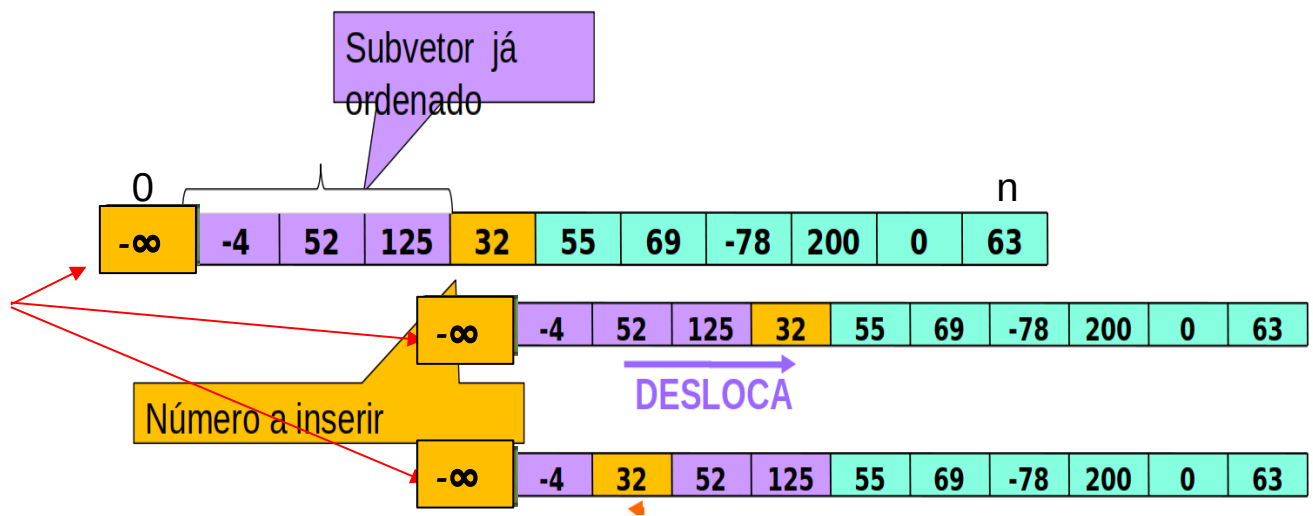
Melhor caso:  $M(n) = 2(n-1)$

Pior caso :  $M(n) = (4 + 5 + \dots + n + 2) = \frac{n^2}{2} + \frac{5n}{2} - 3$

Caso médio :  $M(n) = \frac{1}{2}(5 + 6 + \dots + n + 3) = \frac{n^2}{4} + \frac{11n}{4} - 3$



# Insertion sort (inserção direta)



```
#include <limits.h>
```

```
void insercao(int n, int [] v) {
```

```
    int i, j, chave;
```

```
    v[0] = INT_MIN; //sentinela!!!
```

```
    for (j = 2; j <= n; j++) {
```

```
        chave = v[j];
```

```
        // ordenando elementos à esquerda
```

```
        i = j - 1;
```

```
        while (v[i] > chave){
```

```
            v[i+1] = v[i];
```

```
            i = i - 1;
```

```
        }
```

```
        v[i+1] = chave;
```

```
    }
```

**Número de movimentações de registros -  $M(n)$ :**

Para análise de caso médio é útil entender o que ocorre em cada iteração:

Na iteração de um dado valor  $j$ :

Melhor caso:  $M_j(n) = 2$

Pior caso:  $M_j(n) = (j-1)+2$

Caso médio:  $M_j(n) = \text{média}(2, j+1) = (j+3)/2$

**No total** (para  $j$  de 2 a  $n$ ):

Melhor caso:  $M(n) = 2(n-1)$

No livro do Ziviani ele coloca a inicialização da sentinela dentro do **for** ( $v[0] = \text{chave}$ ). É desnecessário, e aumenta em 1 o nr de movimentações (que ele só considerou no melhor caso, por isso lá ele coloca  $M_j(n) = 3$ ). No pior caso e caso médio ficou igual porque acho que ele considerou usar o  $v[0]$  no lugar da *chave*.

# Tabela comparativa dos algoritmos de ordenação

	T(n)			C(n)			M(n)			in loco?	estável?
Algoritmo	Melhor caso	Pior caso	Caso médio	Melhor caso	Pior caso	Caso médio	Melhor caso	Pior caso	Caso médio		
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n^2)$	sim	sim

# Seleção Direta

- Primeiro passo:
  - Encontrar o menor elemento do array
  - Levar este menor elemento para o início do array (troca com o primeiro)
- Segundo passo:
  - Encontrar o segundo menor elemento do array
  - Levar este menor elemento para a segunda posição do array (troca de posição)
- E assim por diante...

52	125	-4	32	55	69	-78	200	0	63
----	-----	----	----	----	----	-----	-----	---	----

# Seleção Direta

1)

52	125	-4	32	55	69	-78	200	0	63
----	-----	----	----	----	----	-----	-----	---	----

Prox.  
Posição a ser preenchida

Subvetor a ser pesquisado  
pelo menor elemento

# Seleção Direta

1)

52	125	-4	32	55	69	-78	200	0	63
----	-----	----	----	----	----	-----	-----	---	----

Prox.  
Posição a ser preenchida

Subvetor a ser pesquisado  
pelo menor elemento

// código para achar o menor elemento

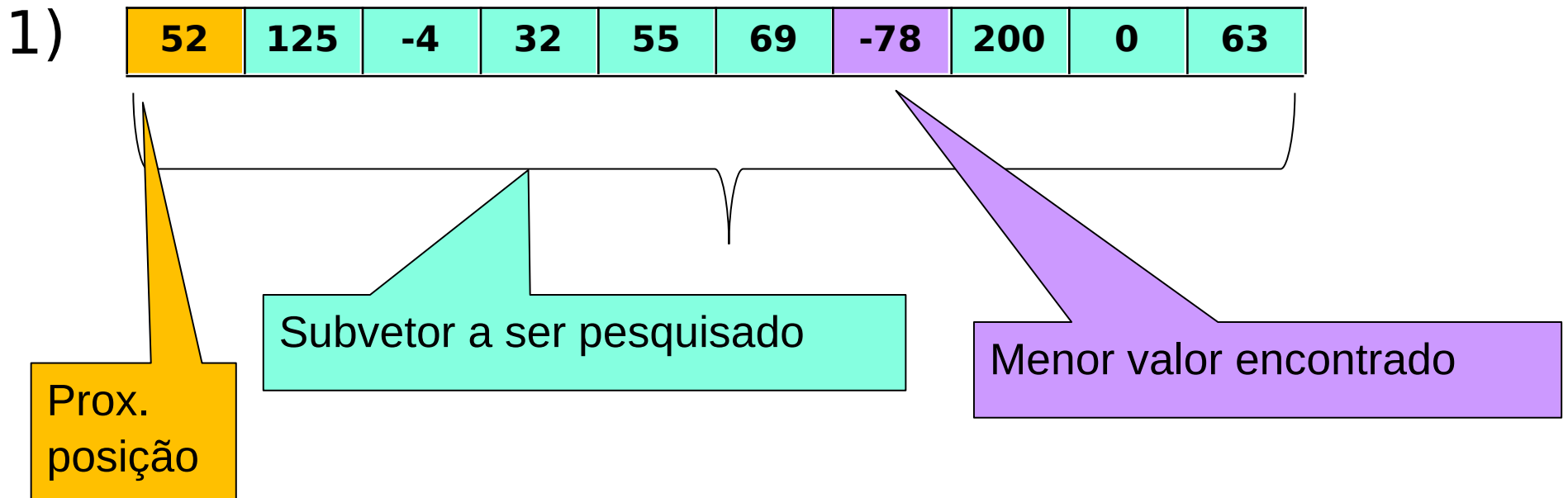
// i é a prox posição a ser preenchida, min é o índice do menor elemento

min = i;

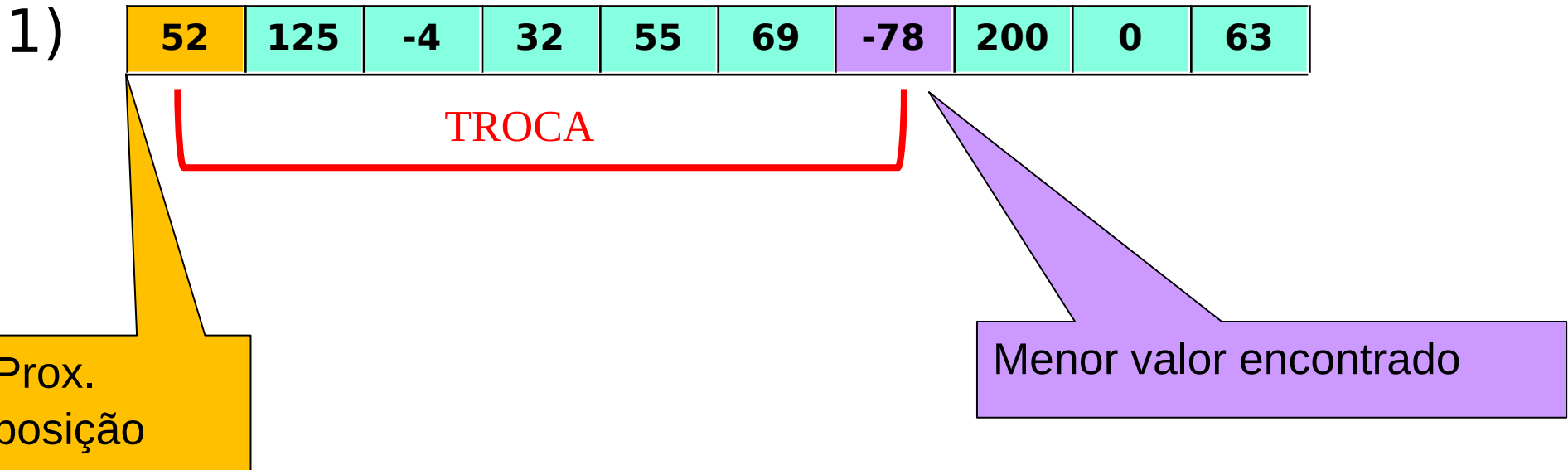
for (j = i+1; j < n; j++)

if (v[j] < v[min]) min = j;

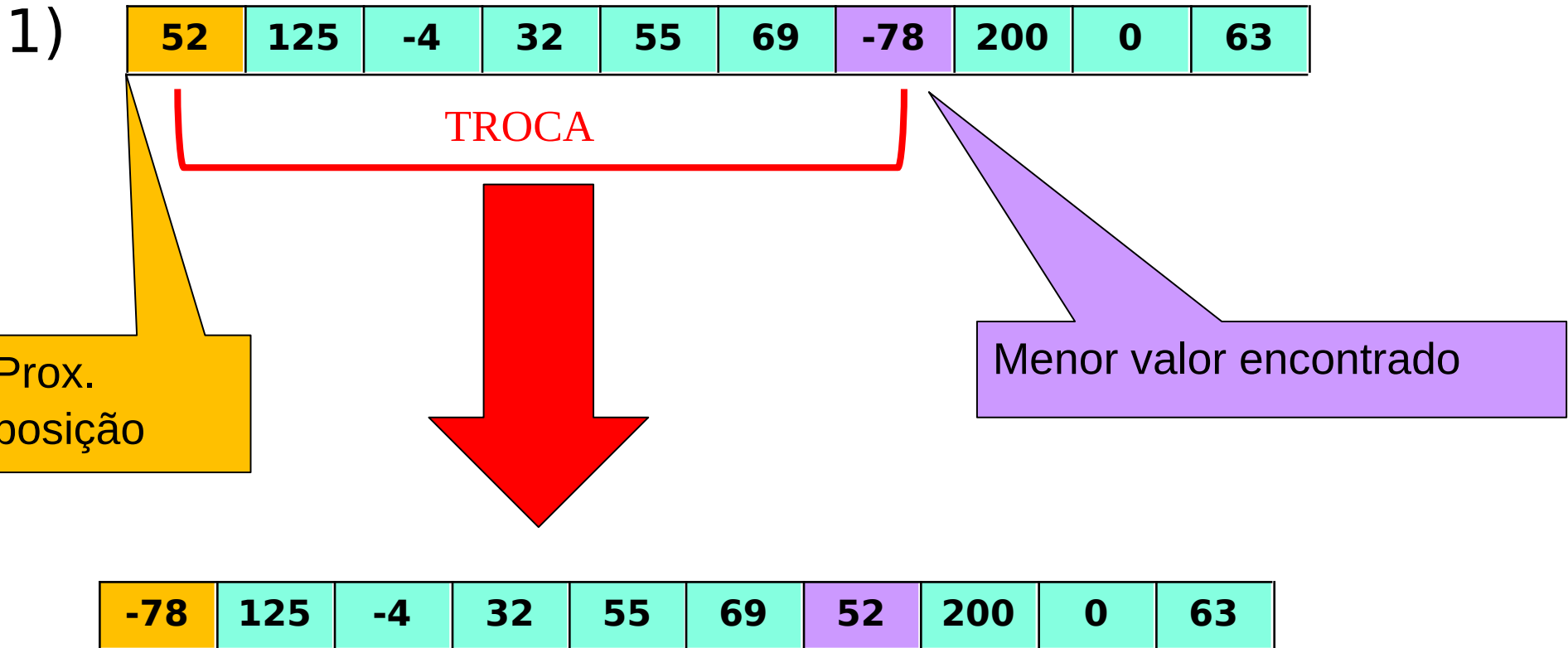
# Seleção Direta



# Seleção Direta

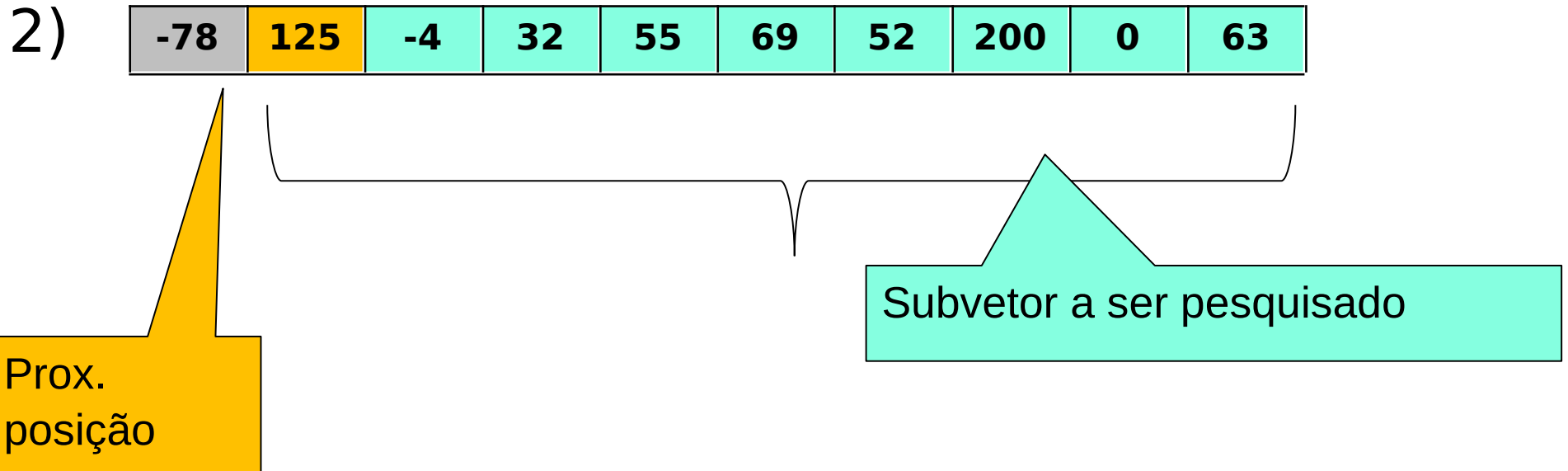


# Seleção Direta

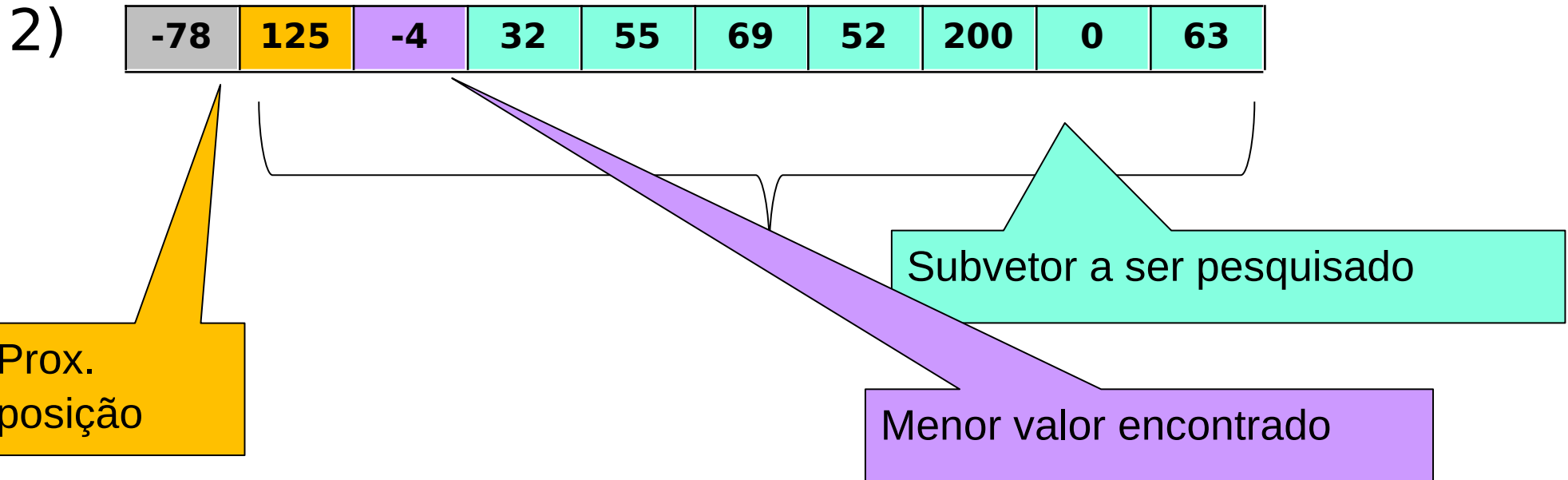




# Seleção Direta



# Seleção Direta



# Seleção Direta

2)

-78	125	-4	32	55	69	52	200	0	63
-----	-----	----	----	----	----	----	-----	---	----

TROCA

Prox.  
posição

Menor valor encontrado

-78	-4	125	32	55	69	52	200	0	63
-----	----	-----	----	----	----	----	-----	---	----

# Seleção Direta

3)

-78	-4	125	32	55	69	52	200	0	63
-----	----	-----	----	----	----	----	-----	---	----

Prox.  
posição

Subvetor a ser pesquisado

# Seleção Direta

3)

-78	-4	125	32	55	69	52	200	0	63
-----	----	-----	----	----	----	----	-----	---	----

Prox.  
posição

Menor valor encontrado

# Seleção Direta

3)

-78	-4	125	32	55	69	52	200	0	63
-----	----	-----	----	----	----	----	-----	---	----

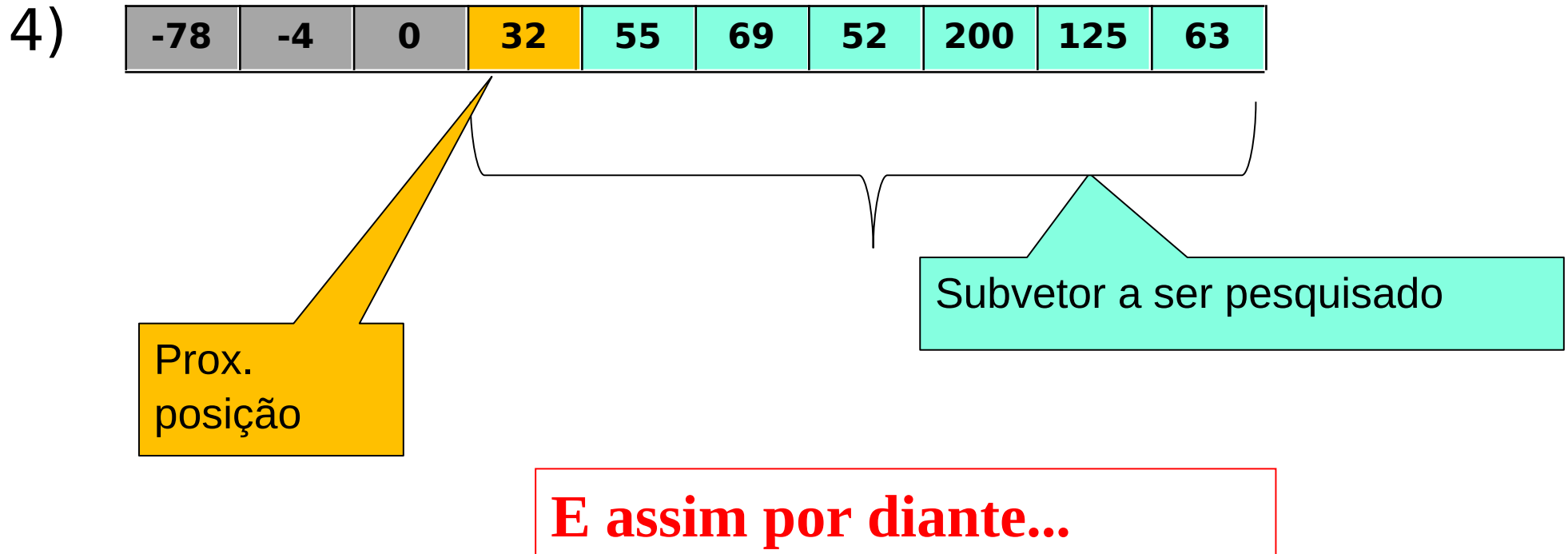
Prox.  
posição

TROCA

Menor valor encontrado

-78	-4	0	32	55	69	52	200	125	63
-----	----	---	----	----	----	----	-----	-----	----

# Seleção Direta



# Seleção Direta

4)

-78	-4	0	32	55	69	52	200	125	63
-----	----	---	----	----	----	----	-----	-----	----

Prox.  
posição

Subvetor a ser pesquisado

**E assim por diante...  
ATÉ QUANDO?**



# Seleção Direta

4)

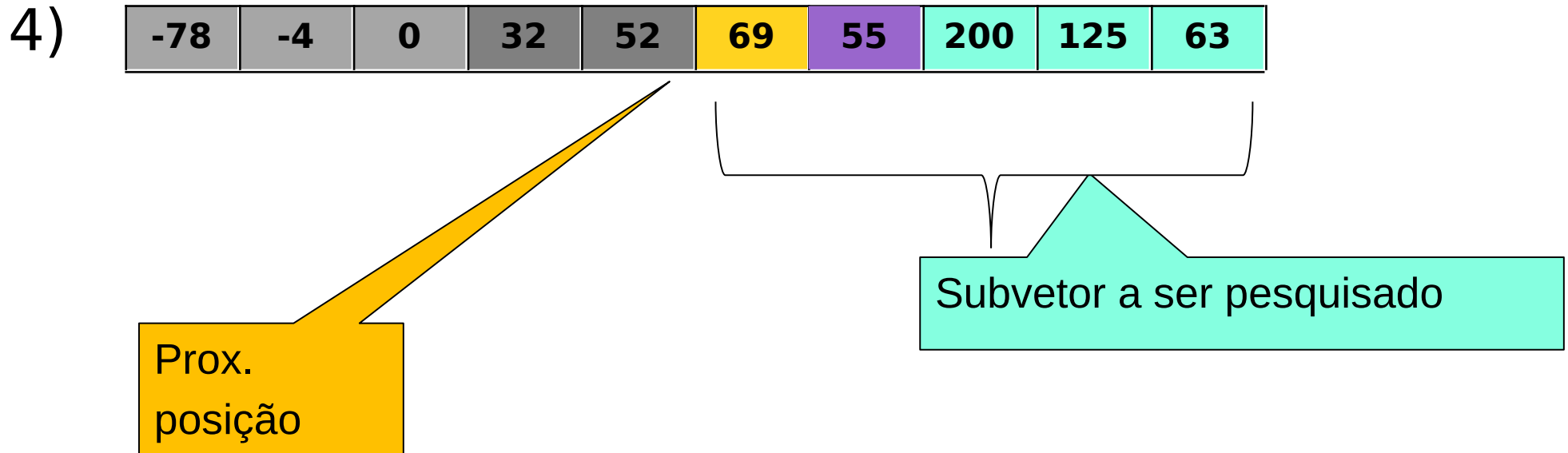
-78	-4	0	32	55	69	52	200	125	63
-----	----	---	----	----	----	----	-----	-----	----

Prox.  
posição

Subvetor a ser pesquisado

**E assim por diante...  
ATÉ QUANDO?**

# Seleção Direta



**E assim por diante...  
ATÉ QUANDO?**

# Seleção Direta

4)

-78	-4	0	32	52	55	69	200	125	63
-----	----	---	----	----	----	----	-----	-----	----

Prox.  
posição

Subvetor a ser pesquisado

**E assim por diante...  
ATÉ QUANDO?**

# Seleção Direta

4)

-78	-4	0	32	52	55	63	200	125	69
-----	----	---	----	----	----	----	-----	-----	----

Prox.  
posição

Subvetor a ser pesquisado

**E assim por diante...  
ATÉ QUANDO?**

# Seleção Direta

4)

-78	-4	0	32	52	55	63	69	125	200
-----	----	---	----	----	----	----	----	-----	-----

Prox.  
posição

Subvetor a ser pesquisado

**E assim por diante...  
ATÉ QUANDO?**

# Seleção Direta

4)

-78	-4	0	32	52	55	63	69	125	200
-----	----	---	----	----	----	----	----	-----	-----

Prox.  
posição

?

Subvetor a ser pesquisado

**E assim por diante...  
ATÉ QUANDO?**

# Seleção Direta

4)

-78	-4	0	32	52	55	63	69	125	200
-----	----	---	----	----	----	----	----	-----	-----

Prox.  
posição

?

Subvetor a ser pesquisado

E assim por diante...

**PÁRO QUANDO PRÓXIMA  
POSIÇÃO = TAM-1**

**(a última rodada é para pos = tam - 2)**

# Seleção Direta

## Algoritmo (vetor com início em 0):

- para cada posição do vetor original, onde a próxima posição a ser preenchida vai de 0 a tam-2
  - Percorre subvetor (da próxima posição a ser preenchida até tamanho do vetor -1) para encontrar menor valor no subvetor
  - Troca com o elemento da próxima posição a ser preenchida



# Seleção Direta

```
void Seleção (int n, int v[]) {  
    int i, j, min, x;  
    for (i = 0; /*A*/ i < n-1; i++) {  
        min = i;  
        for (j = i+1; j < n; j++)  
            if (v[j] < v[min]) min = j;  
        x = v[i]; v[i] = v[min]; v[min] = x;  
    }  
}
```

# Seleção Direta

```
void Seleção (int n, int v[]) {  
    int i, j, min, x;  
    for (i = 0; /*A*/ i < n-1; i++) {  
        min = i;  
        for (j = i+1; j < n; j++)  
            if (v[j] < v[min]) min = j;  
        x = v[i]; v[i] = v[min]; v[min] = x;  
    }  
}
```

*In loco?:*

# Seleção Direta

```
void Seleção (int n, int v[]) {  
    int i, j, min, x;  
    for (i = 0; /*A*/ i < n-1; i++) {  
        min = i;  
        for (j = i+1; j < n; j++)  
            if (v[j] < v[min]) min = j;  
        x = v[i]; v[i] = v[min]; v[min] = x;  
    }  
}
```

*In loco?*: **SIM!**

*Estável*:

# Seleção Direta

```
void Seleção (int n, int v[]) {  
    int i, j, min, x;  
    for (i = 0; /*A*/ i < n-1; i++) {  
        min = i;  
        for (j = i+1; j < n; j++)  
            if (v[j] < v[min]) min = j;  
        x = v[i]; v[i] = v[min]; v[min] = x;  
    }  
}
```

*In loco?*: **SIM!**

*Estável*: **Não!** Por quê?

# Seleção Direta

```
void Seleção (int n, int v[]) {  
    int i, j, min, x;  
    for (i = 0; /*A*/ i < n-1; i++) {  
        min = i;  
        for (j = i+1; j < n; j++)  
            if (v[j] < v[min]) min = j;  
        x = v[i]; v[i] = v[min]; v[min] = x;  
    }  
}
```

**In loco?: SIM!**

**Estável: Não!** Por quê?

A troca pode fazer com que um elemento da posição  $j$  vá para a posição  $min$ , passando “por cima” de outros elementos de igual valor

# Seleção Direta

```
void Seleção (int n, int v[]) {  
    int i, j, min, x;  
    for (i = 0; /*A*/ i < n-1; i++) {  
        min = i;  
        for (j = i+1; j < n; j++)  
            if (v[j] < v[min]) min = j;  
        x = v[i]; v[i] = v[min]; v[min] = x;  
    }  
}
```

Complexidade de tempo:

# Seleção Direta

```
void Seleção (int n, int v[]) {  
    int i, j, min, x;  
    for (i = 0; /*A*/ i < n-1; i++) {  
        min = i;  
        for (j = i+1; j < n; j++)  
            if (v[j] < v[min]) min = j;  
        x = v[i]; v[i] = v[min]; v[min] = x;  
    }  
}
```

Complexidade de tempo:  $O(n^2)$

Complexidade de tempo no melhor caso:

# Seleção Direta

```
void Seleção (int n, int v[]) {  
    int i, j, min, x;  
    for (i = 0; /*A*/ i < n-1; i++) {  
        min = i;  
        for (j = i+1; j < n; j++)  
            if (v[j] < v[min]) min = j;  
        x = v[i]; v[i] = v[min]; v[min] = x;  
    }  
}
```

Complexidade de tempo:  $O(n^2)$

Complexidade de tempo no melhor caso:

Não tem melhor caso – faz  $O(n^2)$  sempre!



# Seleção Direta

```
void Seleção (int n, int v[]) {  
    int i, j, min, x;  
    for (i = 0; /*A*/ i < n-1; i++) {  
        min = i;  
        for (j = i+1; j < n; j++)  
            if (v[j] < v[min]) min = j;  
        x = v[i]; v[i] = v[min]; v[min] = x;  
    }  
}
```

## Pseudocódigo:

(vetor começando em 1 facilita as contas corretas)

```
para i ← 1 até n - 1 faça  
    min ← i  
    para j ← i + 1 até n faça  
        se v[j] < v[min] min ← j  
    x ← v[i]  
    v[i] ← v[min]  
    v[min] ← x
```

$$C(n) =$$

$$M(n) =$$

# Seleção Direta

```
void Seleção (int n, int v[]) {  
    int i, j, min, x;  
    for (i = 0; /*A*/ i < n-1; i++) {  
        min = i;  
        for (j = i+1; j < n; j++)  
            if (v[j] < v[min]) min = j;  
        x = v[i]; v[i] = v[min]; v[min] = x;  
    }  
}
```

## Pseudocódigo:

(vetor começando em 1 facilita as contas corretas)

```
para i ← 1 até n - 1 faça  
    min ← i  
    para j ← i + 1 até n faça  
        se v[j] < v[min] min ← j  
    x ← v[i]  
    v[i] ← v[min]  
    v[min] ← x
```

$$\sum_{i=1}^{n-1} (n-i) = n-1 + n-2 + \dots + 1$$

$$C(n) = \frac{n^2}{2} - \frac{n}{2}$$

$$M(n) =$$

# Seleção Direta

```
void Seleção (int n, int v[]) {  
    int i, j, min, x;  
    for (i = 0; /*A*/ i < n-1; i++) {  
        min = i;  
        for (j = i+1; j < n; j++)  
            if (v[j] < v[min]) min = j;  
        x = v[i]; v[i] = v[min]; v[min] = x;  
    }  
}
```

## Pseudocódigo:

(vetor começando em 1 facilita as contas corretas)

```
para i ← 1 até n - 1 faça  
    min ← i  
    para j ← i + 1 até n faça  
        se v[j] < v[min] min ← j  
    x ← v[i]  
    v[i] ← v[min]  
    v[min] ← x
```

$$\sum_{i=1}^{n-1} (n-i) = n-1 + n-2 + \dots + 1 \quad \longleftarrow \quad C(n) = \frac{n^2}{2} - \frac{n}{2}$$

$$M(n) = 3(n - 1)$$

Linear no número de movimentações!!!

Muito atraente quando os registros são grandes!!!



# Comparando InsertionSort com SelectionSort

	T(n)			C(n)			M(n)			in loco?	estável?
Algoritmo	Melhor caso	Pior caso	Caso médio	Melhor caso	Pior caso	Caso médio	Melhor caso	Pior caso	Caso médio		
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n^2)$	sim	sim
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n)$	$O(n)$	sim	não

- InsertionSort: quando é uma boa escolha?
- SelectionSort:

# Comparando InsertionSort com SelectionSort

Algoritmo	T(n)			C(n)			M(n)			in loco?	estável?
	Melhor caso	Pior caso	Caso médio	Melhor caso	Pior caso	Caso médio	Melhor caso	Pior caso	Caso médio		
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n^2)$	sim	sim
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n)$	$O(n)$	sim	não

- InsertionSort:
  - Boa escolha para arquivos quase ordenados, ou quando precisa inserir alguns poucos registros em um arquivo que já estava ordenado
- SelectionSort: quando é uma boa escolha?

# Comparando InsertionSort com SelectionSort

	T(n)			C(n)			M(n)			in loco?	estável?
Algoritmo	Melhor caso	Pior caso	Caso médio	Melhor caso	Pior caso	Caso médio	Melhor caso	Pior caso	Caso médio		
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n^2)$	sim	sim
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n)$	$O(n)$	sim	não

- InsertionSort:
  - Boa escolha para arquivos quase ordenados, ou quando precisa inserir alguns poucos registros em um arquivo que já estava ordenado
- SelectionSort:
  - Boa escolha para quando os arquivos não estão quase ordenados e os registros são grandes – e estabilidade não é um problema (M(n) linear é um grande diferencial...)

# Seleção Direta - Exercício

```
void Seleção (int n, int v[]) {  
    int i, j, min, x;  
    for (i = 0; /*A*/ i < n-1; i++) {  
        min = i;  
        for (j = i+1; j < n; j++)  
            if (v[j] < v[min]) min = j;  
        x = v[i]; v[i] = v[min]; v[min] = x;  
    }  
}
```

Prove por indução que esse algoritmo está correto.

Dica: Qual é a invariante A no início do laço que pode ser usada nesta prova?

# Método da bolha (BubbleSort)

- Passa pelo array uma bolha que ordena duas posições consecutivas (**leva o maior elemento para o final do vetor**)
- Percorre o array e, em cada passo:
  - Testa se dois vizinhos estão ordenados
  - Troca o par que não estiver ordenado



1)

52	125	-4	32	55	69	-78	200	0	63
----	-----	----	----	----	----	-----	-----	---	----

não troca

52	-4	125	32	55	69	-78	200	0	63
----	----	-----	----	----	----	-----	-----	---	----

troca

52	-4	32	125	55	69	-78	200	0	63
----	----	----	-----	----	----	-----	-----	---	----

troca

52	-4	32	55	125	69	-78	200	0	63
----	----	----	----	-----	----	-----	-----	---	----

troca

52	-4	32	55	69	125	-78	200	0	63
----	----	----	----	----	-----	-----	-----	---	----

troca

52	-4	32	55	69	-78	125	200	0	63
----	----	----	----	----	-----	-----	-----	---	----

não troca

52	-4	32	55	69	-78	125	0	200	63
----	----	----	----	----	-----	-----	---	-----	----

troca

52	-4	32	55	69	-78	125	0	63	200
----	----	----	----	----	-----	-----	---	----	-----

Posição  
correta



2) 

52	-4	32	55	69	-78	125	0	63	200
----	----	----	----	----	-----	-----	---	----	-----

 troca

-4	52	32	55	69	-78	125	0	63	200
----	----	----	----	----	-----	-----	---	----	-----

 troca

-4	32	52	55	69	-78	125	0	63	200
----	----	----	----	----	-----	-----	---	----	-----

 não troca

...

-4	32	52	55	-78	69	0	63	125	200
----	----	----	----	-----	----	---	----	-----	-----

Posição  
correta

2) 

52	-4	32	55	69	-78	125	0	63	200
----	----	----	----	----	-----	-----	---	----	-----

 troca

-4	52	32	55	69	-78	125	0	63	200
----	----	----	----	----	-----	-----	---	----	-----

 troca

-4	32	52	55	69	-78	125	0	63	200
----	----	----	----	----	-----	-----	---	----	-----

 não troca

...

-4	32	52	55	-78	69	0	63	125	200
----	----	----	----	-----	----	---	----	-----	-----

Posição  
correta

E assim por diante...

# Método da bolha (BubbleSort)

## Algoritmo:

- Quantas vezes?
  - Percorre o vetor desde o início até onde já estiver ordenado (**ivet**) e vai trocando o par de lugar quando posição  $[j-1] > \text{posição } [j]$

# Método da bolha (BubbleSort)

```
void bolha(int n, int [] v) {  
    int ivet, j, temp;  
    // ivet é o "final" do vetor para onde deve ir o maior elemento (do fim para o início)  
    for (ivet = n - 1; ivet > 0; ivet--)  
    {  
        // a cada passagem o maior elemento é descolado para a sua posição correta  
        for (j = 0; j < ivet; j++)  
        // se ordem está errada para dois números consecutivos, troca os números  
        if (v[j] > v[j+1])  
        {  
            temp = v[j];  
            v[j] = v[j+1];  
            v[j+1] = temp;  
        }  
    }  
}
```

# Método da bolha (BubbleSort)

```
void bolha(int n, int [] v) {  
    int ivet, j, temp;  
    // ivet é o “final” do vetor para onde deve ir o maior elemento (do fim para o início)  
    for (ivet = n - 1; ivet > 0; ivet--)  
    {  
        // a cada passagem o maior elemento é descolado para a sua posição correta  
        for (j = 0; j < ivet; j++)  
        // se ordem está errada para dois números consecutivos, troca os números  
        if (v[j] > v[j+1])  
        {  
            temp = v[j];  
            v[j] = v[j+1];  
            v[j+1] = temp;  
        }  
    }  
}
```

***In loco?:***

# Método da bolha (BubbleSort)

```
void bolha(int n, int [] v) {  
    int ivet, j, temp;  
    // ivet é o “final” do vetor para onde deve ir o maior elemento (do fim para o início)  
    for (ivet = n - 1; ivet > 0; ivet--)  
    {  
        // a cada passagem o maior elemento é descolado para a sua posição correta  
        for (j = 0; j < ivet; j++)  
        // se ordem está errada para dois números consecutivos, troca os números  
        if (v[j] > v[j+1])  
        {  
            temp = v[j];  
            v[j] = v[j+1];  
            v[j+1] = temp;  
        }  
    }  
}
```

*In loco?:* **SIM!**

# Método da bolha (BubbleSort)

```
void bolha(int n, int [] v) {  
    int ivet, j, temp;  
    // ivet é o “final” do vetor para onde deve ir o maior elemento (do fim para o início)  
    for (ivet = n - 1; ivet > 0; ivet--)  
    {  
        // a cada passagem o maior elemento é descolado para a sua posição correta  
        for (j = 0; j < ivet; j++)  
        // se ordem está errada para dois números consecutivos, troca os números  
        if (v[j] > v[j+1])  
        {  
            temp = v[j];  
            v[j] = v[j+1];  
            v[j+1] = temp;  
        }  
    }  
}
```

**In loco?: SIM!**

**Estável: Exercício!** Se sim, o que o torna estável? Se não, dá para fazer alguma alteraçãozinha para torná-lo estável?

**Complexidade de tempo:**



# Método da bolha (BubbleSort)

```
void bolha(int n, int [] v) {  
    int ivet, j, temp;  
    // ivet é o "final" do vetor para onde deve ir o maior elemento (do fim para o início)  
    for (ivet = n - 1; ivet > 0; ivet--)  
    {  
        // a cada passagem o maior elemento é descolado para a sua posição correta  
        for (j = 0; j < ivet; j++)  
        // se ordem está errada para dois números consecutivos, troca os números  
        if (v[j] > v[j+1])  
        {  
            temp = v[j];  
            v[j] = v[j+1];  
            v[j+1] = temp;  
        }  
    }  
}
```

**In loco?: SIM!**

**Estável: Exercício!** Se sim, o que o torna estável? Se não, dá para fazer alguma alteraçãozinha para torná-lo estável?

**Complexidade de tempo:  $O(n^2)$**

**Complexidade de tempo no melhor caso:**

# Método da bolha (BubbleSort)

```
void bolha(int n, int [] v) {  
    int ivet, j, temp;  
    // ivet é o “final” do vetor para onde deve ir o maior elemento (do fim para o início)  
    for (ivet = n - 1; ivet > 0; ivet--)  
    {  
        // a cada passagem o maior elemento é descolado para a sua posição correta  
        for (j = 0; j < ivet; j++)  
        // se ordem está errada para dois números consecutivos, troca os números  
        if (v[j] > v[j+1])  
        {  
            temp = v[j];  
            v[j] = v[j+1];  
            v[j+1] = temp;  
        }  
    }  
}
```

**In loco?: SIM!**

**Estável: Exercício!** Se sim, o que o torna estável? Se não, dá para fazer alguma alteraçãozinha para torná-lo estável?

**Complexidade de tempo:  $O(n^2)$**

**Complexidade de tempo no melhor caso:**

Não tem melhor caso – faz  **$O(n^2)$  sempre!**

# Método da bolha (BubbleSort)

```
void bolha(int n, int [] v) {  
    int ivet, j, temp;  
    // ivet é o "final" do vetor para onde deve ir o maior elemento (do fim para o início)  
    for (ivet = n - 1; ivet > 0; ivet--)  
    {  
        // a cada passagem o maior elemento é descolado para a sua posição correta  
        for (j = 0; j < ivet; j++)  
        // se ordem está errada para dois números consecutivos, troca os números  
        if (v[j] > v[j+1])  
        {  
            temp = v[j];  
            v[j] = v[j+1];  
            v[j+1] = temp;  
        }  
    }  
}
```

## Exercício:

Faça as análises de nr de comparações e  
Nr de movimentações para o melhor/pior  
Caso e caso médio.

# Algoritmos elementares x eficientes

- InsertionSort, SelectionSort e BubbleSort são exemplos de **algoritmos elementares** de ordenação
  - Complexidade de tempo (pior caso):  $O(n^2)$
  - Porém na prática são bem rápidos para **arquivos pequenos** (preferíveis)
  - Algoritmos mais simples
- MergeSort e outros que veremos nas próximas aulas são exemplos de **algoritmos eficientes** –  $O(n \lg n)$ 
  - Vamos fazer a análise do MergeSort

# MergeSort

**mergeSort (A, i, f)**

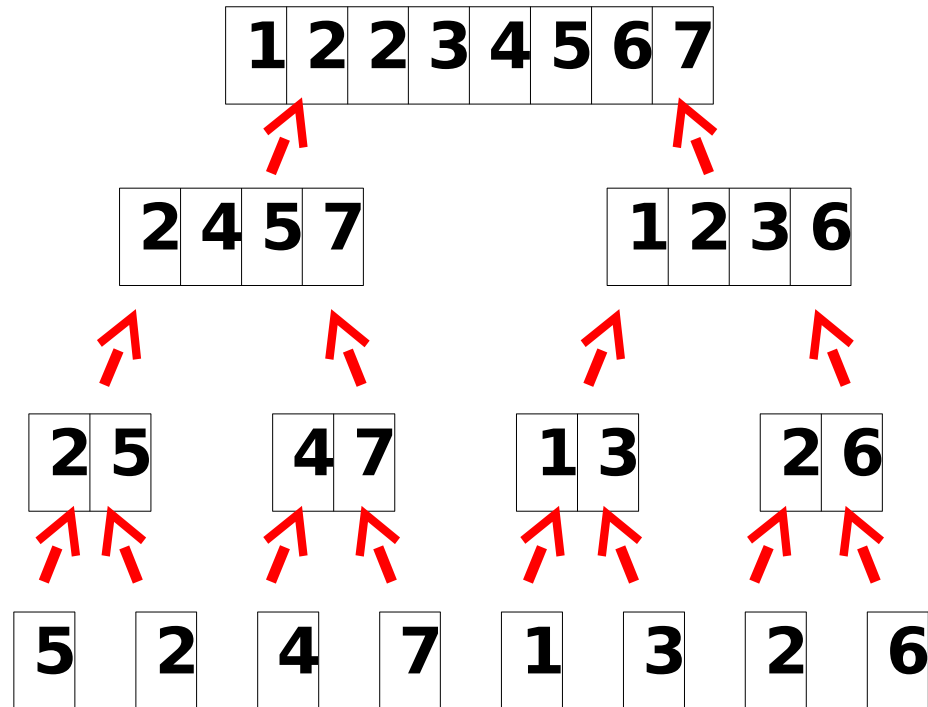
se  $i < f$

$m \leftarrow \lfloor (i+f)/2 \rfloor$

mergeSort(A, i, m)

mergeSort(A, m+1, f)

merge(A, i, m, f)



Arranjo inicial



Divide até obter subarranjos com tamanho 1.  
Então, começa a mesclar...

# MergeSort (ordenação por intercalação)

```
mergeSort (A, i, f)
```

```
se i < f
```

```
  m ←  $\lfloor (i+f)/2 \rfloor$ 
```

```
  mergeSort(A, i, m)
```

```
  mergeSort(A, m+1, f)
```

```
  merge(A, i, m, f)
```

```
merge (A, i, m, f)
```

```
  n1 ← m-i+1 // define subarranjos
```

```
  n2 ← f-m
```

```
  criar arranjos L[1..n1+1] e R[1..n2+1]
```

```
  L[n1+1] ←  $\infty$ ; R[n2+1] ←  $\infty$  //sentinela
```

```
  para j ← 1 até n1  L[j] ← A[i+j-1]
```

```
  para j ← 1 até n2  R[j] ← A[m+j]
```

```
  // mesclar subarranjos
```

```
  k1 ← 1 // k1 é o índice que percorre L
```

```
  k2 ← 1 // k2 é o índice que percorre R
```

```
  para k ← i até f // k percorre A
```

```
    se L[k1] ≤ R[k2]
```

```
      A[k] ← L[k1]
```

```
      k1 ← k1 + 1
```

```
    senão
```

```
      A[k] ← R[k2]
```

```
      k2 ← k2 + 1
```

```
  fim se
```

```
  fim para
```

# Complexidade de tempo do MergeSort (cálculo “por intuição”)

$$T(n) = \begin{cases} c & \text{if } n = 1, \\ 2T(n/2) + cn & \text{if } n > 1, \end{cases}$$

Nr de subproblemas

Tamanho de cada subproblema

Tempo para divisão e conquista

mergeSort (A, p, r)

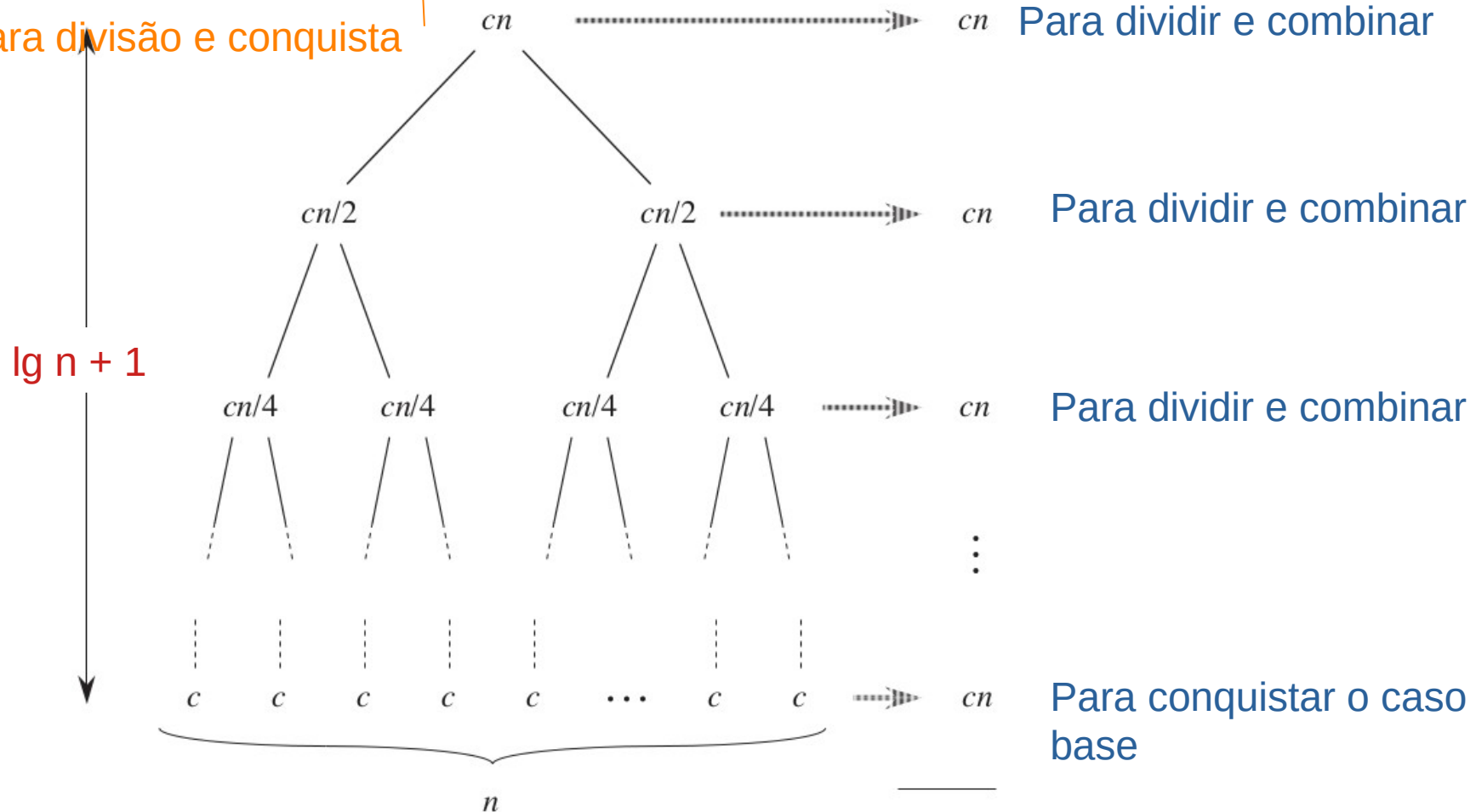
se  $p < r$

$q \leftarrow \lfloor (p+r)/2 \rfloor$

mergeSort(A, p, q)

mergeSort(A, q+1, r)

merge(A, p, q, r)



Total:  $cn (\lg n + 1) = O(n \lg n)$

# MergeSort (ordenação por intercalação)

**mergeSort (A, i, f)**

se  $i < f$

$m \leftarrow \lfloor (i+f)/2 \rfloor$

mergeSort(A, i, m)

mergeSort(A, m+1, f)

merge(A, i, m, f)

merge (A, i, m, f)

$n1 \leftarrow m-i+1$  // define subarranjos

$n2 \leftarrow f-m$

criar arranjos L[1..n1+1] e R[1..n2+1]

$L[n1+1] \leftarrow \infty$ ;  $R[n2+1] \leftarrow \infty$  //sentinela

para  $j \leftarrow 1$  até  $n1$   $L[j] \leftarrow A[i+j-1]$

para  $j \leftarrow 1$  até  $n2$   $R[j] \leftarrow A[m+j]$

// mesclar subarranjos

$k1 \leftarrow 1$  //  $k1$  é o índice que percorre L

$k2 \leftarrow 1$  //  $k2$  é o índice que percorre R

para  $k \leftarrow i$  até  $f$  //  $k$  percorre A

se  $L[k1] \leq R[k2]$

$A[k] \leftarrow L[k1]$

$k1 \leftarrow k1 + 1$

senão

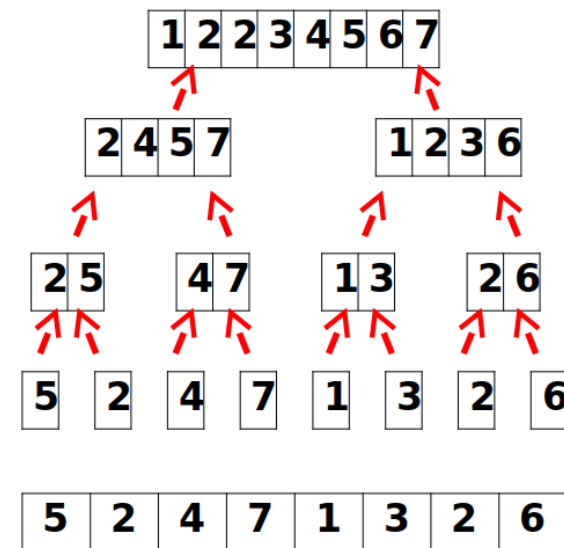
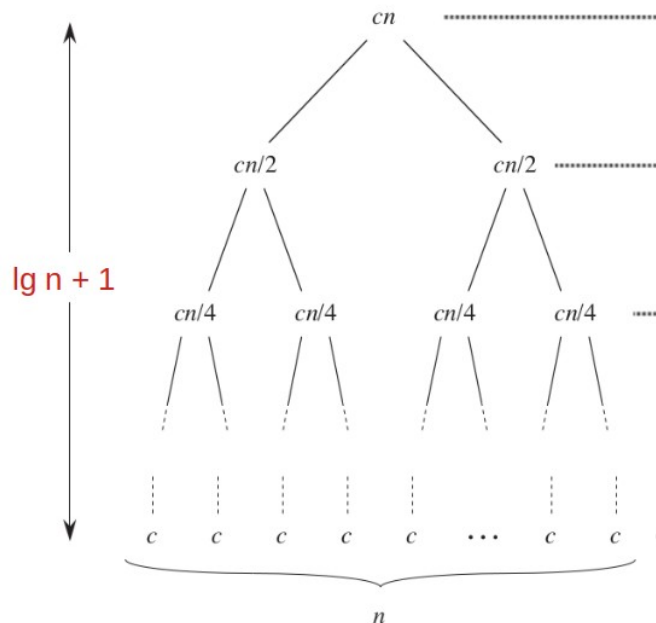
$A[k] \leftarrow R[k2]$

$k2 \leftarrow k2 + 1$

fim se

fim para

**C(n): ?**





# MergeSort (ordenação por intercalação)

**mergeSort (A, i, f)**

se  $i < f$

$m \leftarrow \lfloor (i+f)/2 \rfloor$

mergeSort(A, i, m)

mergeSort(A, m+1, f)

merge(A, i, m, f)

merge (A, i, m, f)

$n1 \leftarrow m-i+1$  // define subarranjos

$n2 \leftarrow f-m$

criar arranjos L[1..n1+1] e R[1..n2+1]

$L[n1+1] \leftarrow \infty$ ;  $R[n2+1] \leftarrow \infty$  //sentinela

para  $j \leftarrow 1$  até  $n1$   $L[j] \leftarrow A[i+j-1]$

para  $j \leftarrow 1$  até  $n2$   $R[j] \leftarrow A[m+j]$

// mesclar subarranjos

$k1 \leftarrow 1$  //  $k1$  é o índice que percorre L

$k2 \leftarrow 1$  //  $k2$  é o índice que percorre R

para  $k \leftarrow i$  até  $f$  //  $k$  percorre A

se  $L[k1] \leq R[k2]$

$A[k] \leftarrow L[k1]$

$k1 \leftarrow k1 + 1$

senão

$A[k] \leftarrow R[k2]$

$k2 \leftarrow k2 + 1$

fim se

fim para

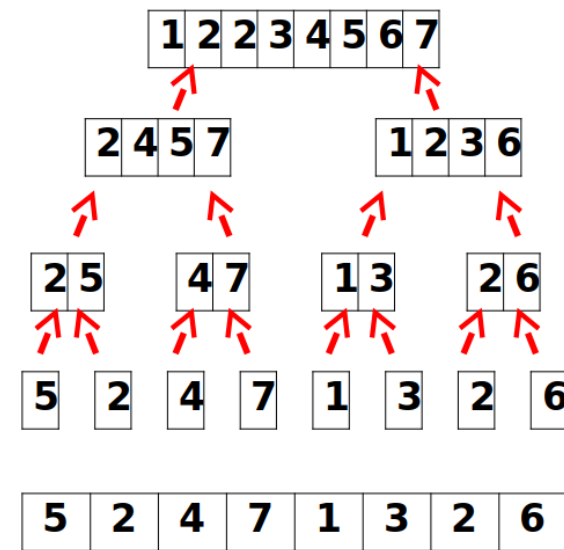
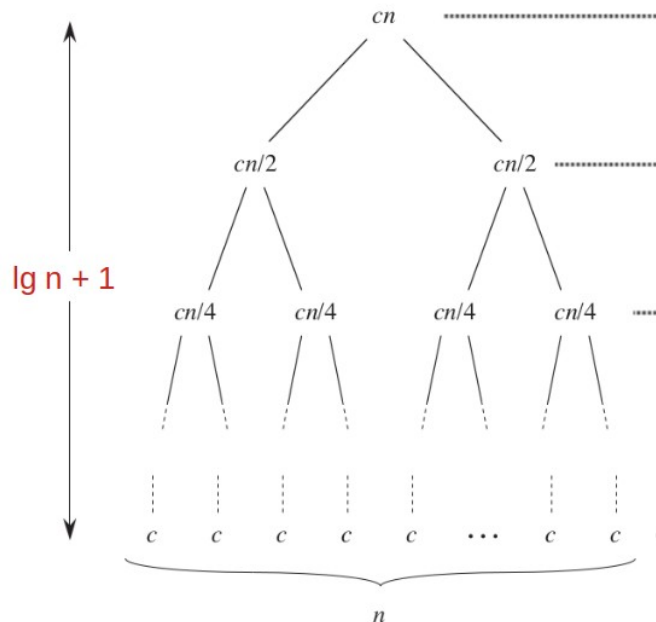
$f-i+1$  comparações  
(n')

$C(n): ?$

$$C(n) = \begin{cases} 0 & \text{se } n = 1 \\ C(\lfloor n/2 \rfloor) + C(\lfloor n/2 \rfloor) + n & \text{se } n > 1 \end{cases}$$

se  $n = 1$

se  $n > 1$



# MergeSort (ordenação por intercalação)

**mergeSort (A, i, f)**

se  $i < f$

$m \leftarrow \lfloor (i+f)/2 \rfloor$

mergeSort(A, i, m)

mergeSort(A, m+1, f)

merge(A, i, m, f)

merge (A, i, m, f)

$n1 \leftarrow m-i+1$  // define subarranjos

$n2 \leftarrow f-m$

criar arranjos L[1..n1+1] e R[1..n2+1]

$L[n1+1] \leftarrow \infty$ ;  $R[n2+1] \leftarrow \infty$  //sentinela

para  $j \leftarrow 1$  até  $n1$   $L[j] \leftarrow A[i+j-1]$

para  $j \leftarrow 1$  até  $n2$   $R[j] \leftarrow A[m+j]$

// mesclar subarranjos

$k1 \leftarrow 1$  //  $k1$  é o índice que percorre L

$k2 \leftarrow 1$  //  $k2$  é o índice que percorre R

para  $k \leftarrow i$  até  $f$  //  $k$  percorre A

se  $L[k1] \leq R[k2]$

$A[k] \leftarrow L[k1]$

$k1 \leftarrow k1 + 1$

senão

$A[k] \leftarrow R[k2]$

$k2 \leftarrow k2 + 1$

fim se

fim para

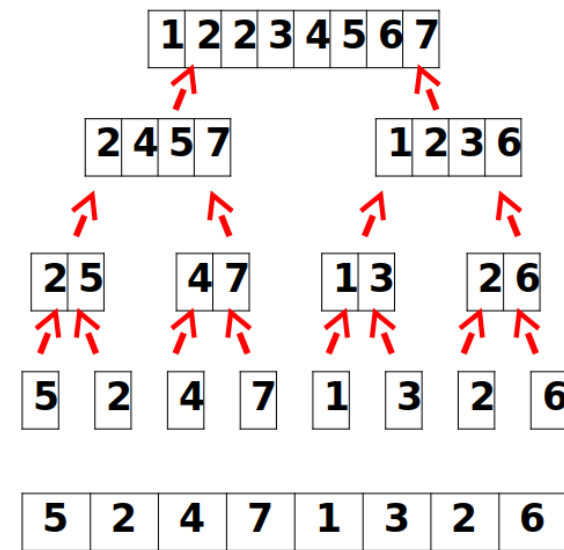
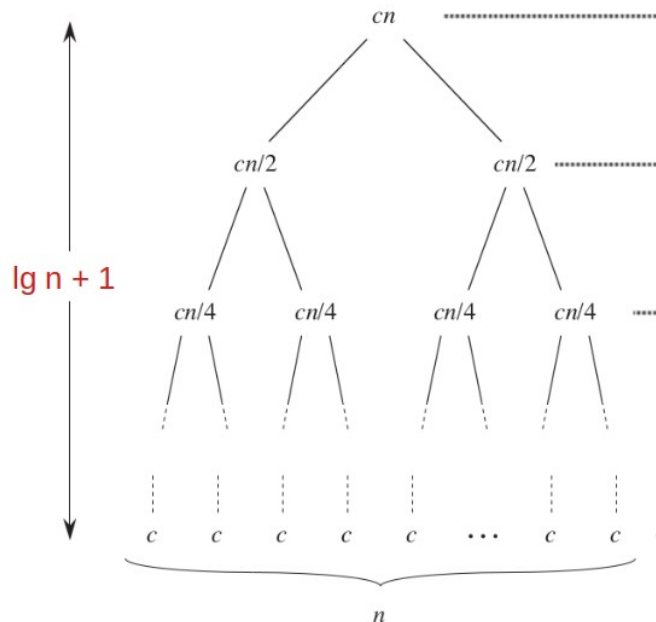
**C(n):  $O(n \lg n)$**

$$C(n) = \begin{cases} 0 & \text{se } n = 1 \\ C(\lfloor n/2 \rfloor) + C(\lfloor n/2 \rfloor) + n & \text{se } n > 1 \end{cases}$$

se  $n = 1$

se  $n > 1$

$f-i+1$  comparações  
(n')



# MergeSort (ordenação por intercalação)

**mergeSort (A, i, f)**

se  $i < f$

$m \leftarrow \lfloor (i+f)/2 \rfloor$

mergeSort(A, i, m)

mergeSort(A, m+1, f)

merge(A, i, m, f)

merge (A, i, m, f)

$n1 \leftarrow m-i+1$  // define subarranjos

$n2 \leftarrow f-m$

criar arranjos L[1..n1+1] e R[1..n2+1]

$L[n1+1] \leftarrow \infty$ ;  $R[n2+1] \leftarrow \infty$  //sentinela

para  $j \leftarrow 1$  até  $n1$   $L[j] \leftarrow A[i+j-1]$

para  $j \leftarrow 1$  até  $n2$   $R[j] \leftarrow A[m+j]$

// mesclar subarranjos

$k1 \leftarrow 1$  //  $k1$  é o índice que percorre L

$k2 \leftarrow 1$  //  $k2$  é o índice que percorre R

para  $k \leftarrow i$  até  $f$  //  $k$  percorre A

se  $L[k1] \leq R[k2]$

$A[k] \leftarrow L[k1]$

$k1 \leftarrow k1 + 1$

senão

$A[k] \leftarrow R[k2]$

$k2 \leftarrow k2 + 1$

fim se

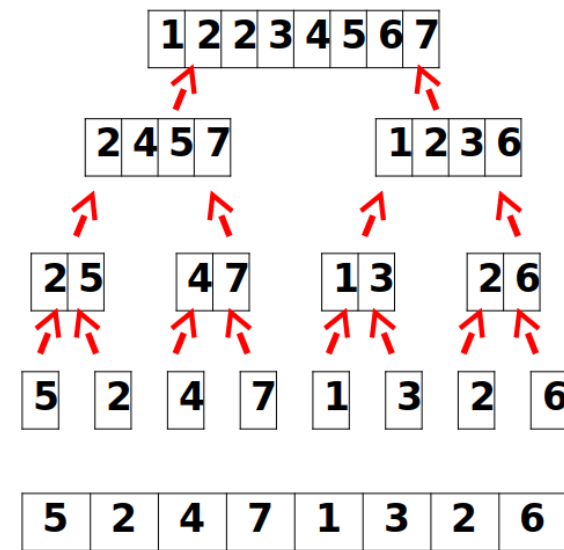
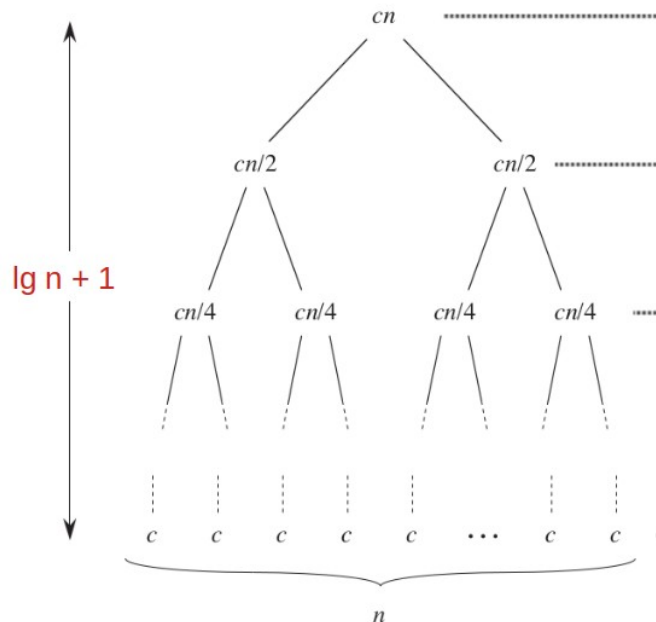
fim para

**C(n):  $O(n \lg n)$**

$$C(n) = \begin{cases} 0 & \text{se } n = 1 \\ C(\lfloor n/2 \rfloor) + C(\lfloor n/2 \rfloor) + n & \text{se } n > 1 \end{cases}$$

Tem melhor caso?

$f-i+1$  comparações  
(n')



# MergeSort (ordenação por intercalação)

**mergeSort (A, i, f)**

se  $i < f$

$m \leftarrow \lfloor (i+f)/2 \rfloor$

mergeSort(A, i, m)

mergeSort(A, m+1, f)

merge(A, i, m, f)

merge (A, i, m, f)

$n1 \leftarrow m-i+1$  // define subarranjos

$n2 \leftarrow f-m$

criar arranjos L[1..n1+1] e R[1..n2+1]

$L[n1+1] \leftarrow \infty$ ;  $R[n2+1] \leftarrow \infty$  //sentinela

para  $j \leftarrow 1$  até  $n1$   $L[j] \leftarrow A[i+j-1]$

para  $j \leftarrow 1$  até  $n2$   $R[j] \leftarrow A[m+j]$

// mesclar subarranjos

$k1 \leftarrow 1$  //  $k1$  é o índice que percorre L

$k2 \leftarrow 1$  //  $k2$  é o índice que percorre R

para  $k \leftarrow i$  até  $f$  //  $k$  percorre A

se  $L[k1] \leq R[k2]$

$A[k] \leftarrow L[k1]$

$k1 \leftarrow k1 + 1$

senão

$A[k] \leftarrow R[k2]$

$k2 \leftarrow k2 + 1$

fim se

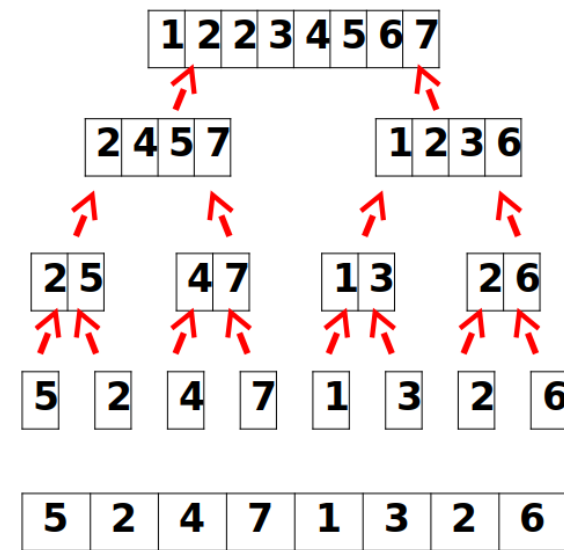
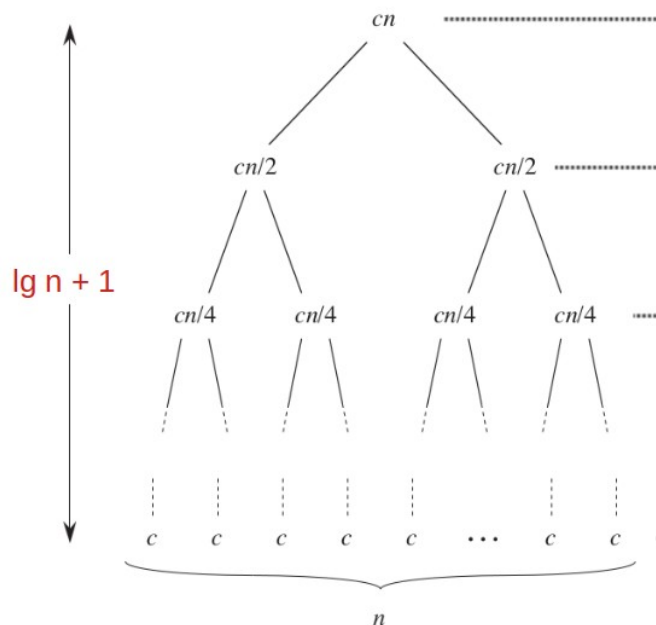
fim para

**C(n):  $O(n \lg n)$**

$$C(n) = \begin{cases} 0 & \text{se } n = 1 \\ C(\lfloor n/2 \rfloor) + C(\lfloor n/2 \rfloor) + n & \text{se } n > 1 \end{cases}$$

Tem melhor caso? **Não!**

$f-i+1$  comparações  
(n')



# MergeSort (ordenação por intercalação)

**mergeSort (A, i, f)**

se  $i < f$

$m \leftarrow \lfloor (i+f)/2 \rfloor$

mergeSort(A, i, m)

mergeSort(A, m+1, f)

merge(A, i, m, f)

merge (A, i, m, f)

$n1 \leftarrow m-i+1$  // define subarranjos

$n2 \leftarrow f-m$

criar arranjos L[1..n1+1] e R[1..n2+1]

$L[n1+1] \leftarrow \infty$ ;  $R[n2+1] \leftarrow \infty$  //sentinela

para  $j \leftarrow 1$  até  $n1$   $L[j] \leftarrow A[i+j-1]$

para  $j \leftarrow 1$  até  $n2$   $R[j] \leftarrow A[m+j]$

// mesclar subarranjos

$k1 \leftarrow 1$  //  $k1$  é o índice que percorre L

$k2 \leftarrow 1$  //  $k2$  é o índice que percorre R

para  $k \leftarrow i$  até  $f$  //  $k$  percorre A

se  $L[k1] \leq R[k2]$

$A[k] \leftarrow L[k1]$

$k1 \leftarrow k1 + 1$

senão

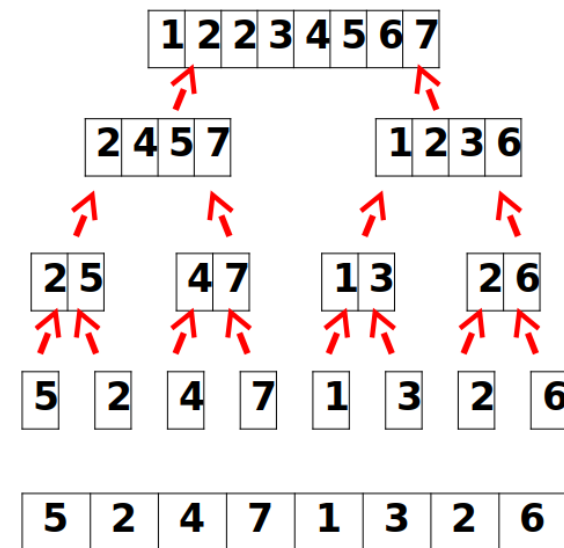
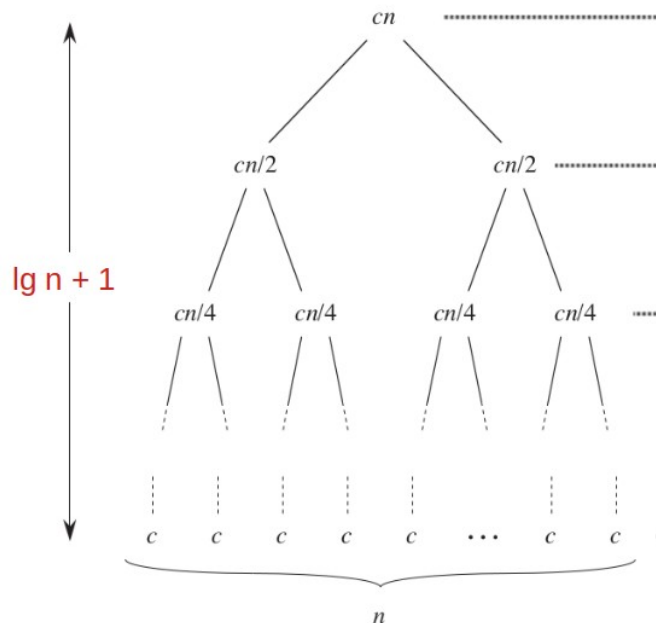
$A[k] \leftarrow R[k2]$

$k2 \leftarrow k2 + 1$

fim se

fim para

**M(n): ?**



# MergeSort (ordenação por intercalação)

**mergeSort (A, i, f)**

se  $i < f$

$m \leftarrow \lfloor (i+f)/2 \rfloor$

mergeSort(A, i, m)

mergeSort(A, m+1, f)

merge(A, i, m, f)

merge (A, i, m, f)

$n1 \leftarrow m-i+1$  // define subarranjos

$n2 \leftarrow f-m$

criar arranjos L[1..n1+1] e R[1..n2+1]

$L[n1+1] \leftarrow \infty$ ;  $R[n2+1] \leftarrow \infty$  //sentinela

para  $j \leftarrow 1$  até  $n1$   $L[j] \leftarrow A[i+j-1]$

para  $j \leftarrow 1$  até  $n2$   $R[j] \leftarrow A[m+j]$

f-i+1  
movimentações  
(n')

// mesclar subarranjos

$k1 \leftarrow 1$  //  $k1$  é o índice que percorre L

$k2 \leftarrow 1$  //  $k2$  é o índice que percorre R

para  $k \leftarrow i$  até  $f$  // k percorre A

se  $L[k1] \leq R[k2]$

$A[k] \leftarrow L[k1]$

$k1 \leftarrow k1 + 1$

senão

$A[k] \leftarrow R[k2]$

$k2 \leftarrow k2 + 1$

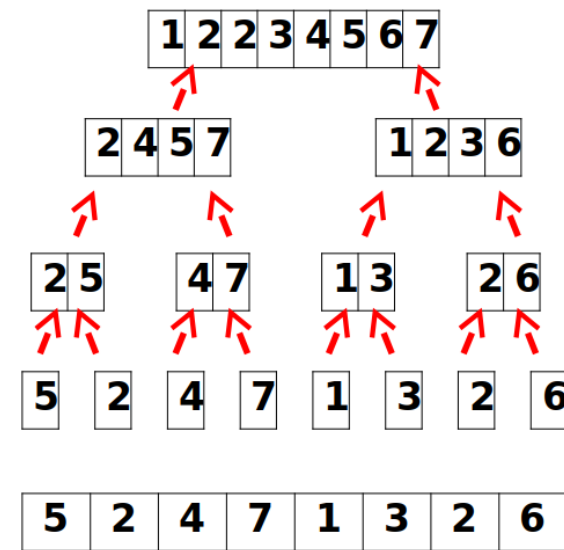
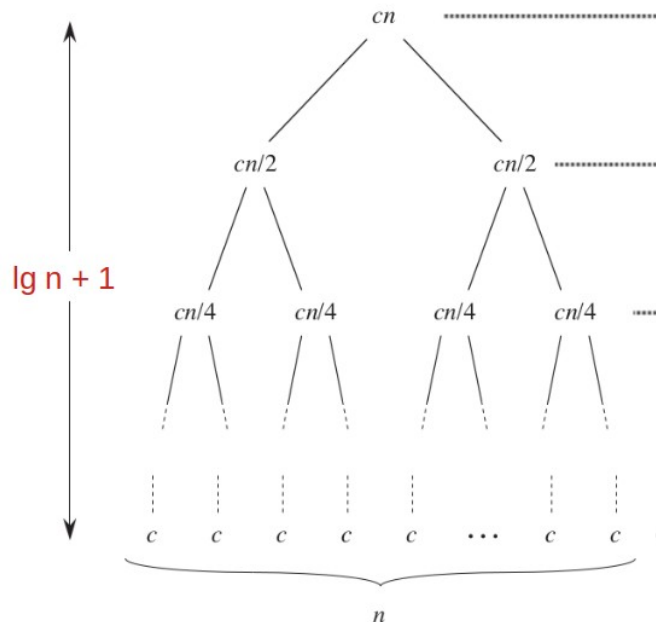
f-i+1  
movimentações  
(n')

fim se

fim para

**M(n):  $O(n \lg n)$**

$$M(n) = \begin{cases} 0 & \text{se } n = 1 \\ M(\lfloor n/2 \rfloor) + M(\lfloor n/2 \rfloor) + 2n & \text{se } n > 1 \end{cases}$$



# MergeSort (ordenação por intercalação)

**mergeSort (A, i, f)**

se  $i < f$

$m \leftarrow \lfloor (i+f)/2 \rfloor$

mergeSort(A, i, m)

mergeSort(A, m+1, f)

merge(A, i, m, f)

merge (A, i, m, f)

$n1 \leftarrow m-i+1$  // define subarranjos

$n2 \leftarrow f-m$

criar arranjos L[1..n1+1] e R[1..n2+1]

$L[n1+1] \leftarrow \infty$ ;  $R[n2+1] \leftarrow \infty$  //sentinela

para  $j \leftarrow 1$  até  $n1$   $L[j] \leftarrow A[i+j-1]$

para  $j \leftarrow 1$  até  $n2$   $R[j] \leftarrow A[m+j]$

f-i+1  
movimentações  
(n')

// mesclar subarranjos

$k1 \leftarrow 1$  // k1 é o índice que percorre L

$k2 \leftarrow 1$  // k2 é o índice que percorre R

para  $k \leftarrow i$  até  $f$  // k percorre A

se  $L[k1] \leq R[k2]$

$A[k] \leftarrow L[k1]$

$k1 \leftarrow k1 + 1$

senão

$A[k] \leftarrow R[k2]$

$k2 \leftarrow k2 + 1$

f-i+1  
movimentações  
(n')

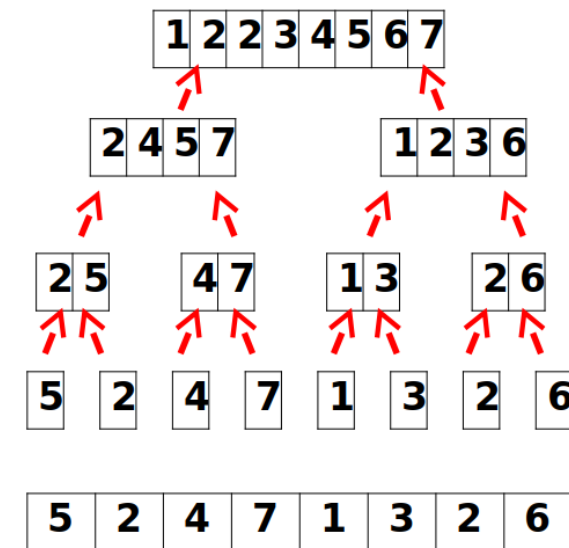
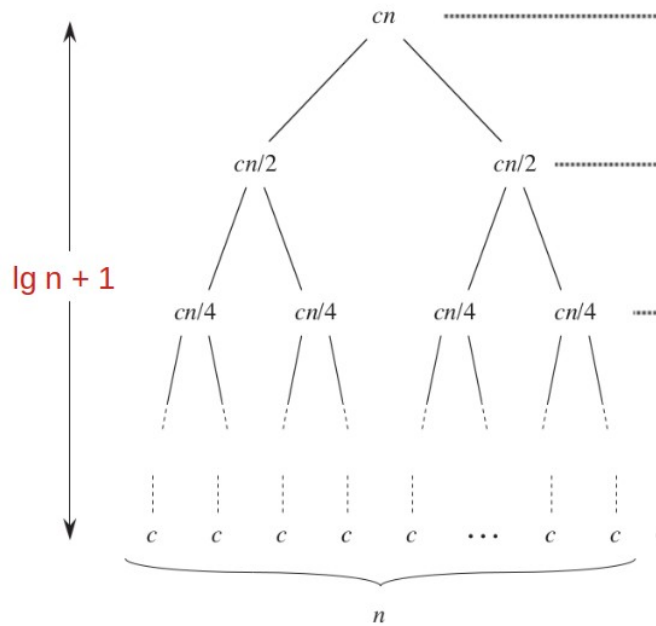
fim se

fim para

**M(n):  $O(n \lg n)$**

$$M(n) = \begin{cases} 0 & \text{se } n = 1 \\ M(\lfloor n/2 \rfloor) + M(\lfloor n/2 \rfloor) + 2n & \text{se } n > 1 \end{cases}$$

Tem melhor caso?



# MergeSort (ordenação por intercalação)

**mergeSort (A, i, f)**

se  $i < f$

$m \leftarrow \lfloor (i+f)/2 \rfloor$

mergeSort(A, i, m)

mergeSort(A, m+1, f)

merge(A, i, m, f)

merge (A, i, m, f)

$n1 \leftarrow m-i+1$  // define subarranjos

$n2 \leftarrow f-m$

criar arranjos L[1..n1+1] e R[1..n2+1]

$L[n1+1] \leftarrow \infty$ ;  $R[n2+1] \leftarrow \infty$  //sentinela

para  $j \leftarrow 1$  até  $n1$   $L[j] \leftarrow A[i+j-1]$

para  $j \leftarrow 1$  até  $n2$   $R[j] \leftarrow A[m+j]$

f-i+1  
movimentações  
(n')

// mesclar subarranjos

$k1 \leftarrow 1$  //  $k1$  é o índice que percorre L

$k2 \leftarrow 1$  //  $k2$  é o índice que percorre R

para  $k \leftarrow i$  até  $f$  //  $k$  percorre A

se  $L[k1] \leq R[k2]$

$A[k] \leftarrow L[k1]$

$k1 \leftarrow k1 + 1$

senão

$A[k] \leftarrow R[k2]$

$k2 \leftarrow k2 + 1$

f-i+1  
movimentações  
(n')

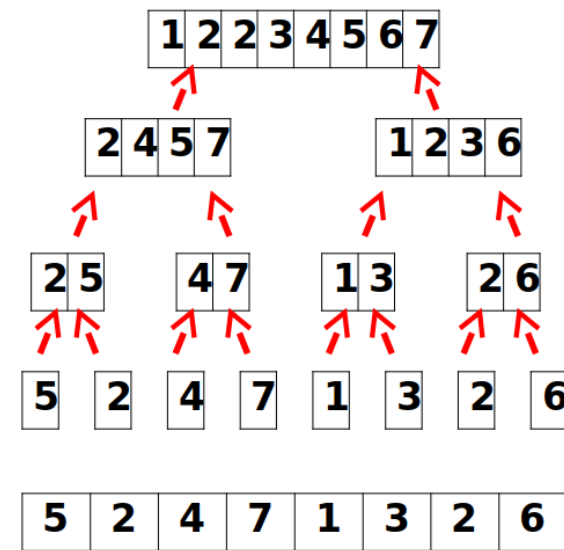
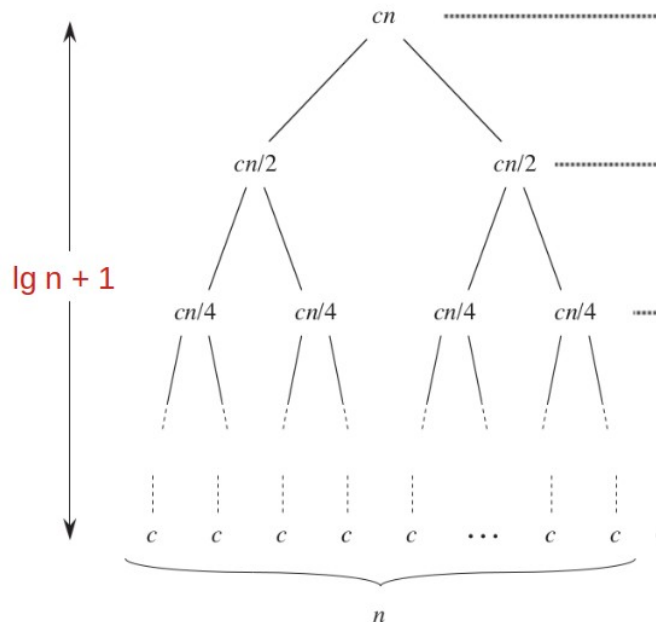
fim se

fim para

**M(n):  $O(n \lg n)$**

$$M(n) = \begin{cases} 0 & \text{se } n = 1 \\ M(\lfloor n/2 \rfloor) + M(\lfloor n/2 \rfloor) + 2n & \text{se } n > 1 \end{cases}$$

Tem melhor caso? **Não!**





# MergeSort (ordenação por intercalação)

**mergeSort (A, i, f)**

se  $i < f$

$m \leftarrow \lfloor (i+f)/2 \rfloor$

mergeSort(A, i, m)

mergeSort(A, m+1, f)

**merge(A, i, m, f)**

**merge** (A, i, m, f)

$n1 \leftarrow m-i+1$  // define subarranjos

$n2 \leftarrow f-m$

criar arranjos L[1.. $n1+1$ ] e R[1.. $n2+1$ ]

$L[n1+1] \leftarrow \infty$ ;  $R[n2+1] \leftarrow \infty$  //sentinela

para  $j \leftarrow 1$  até  $n1$   $L[j] \leftarrow A[i+j-1]$

para  $j \leftarrow 1$  até  $n2$   $R[j] \leftarrow A[m+j]$

// mesclar subarranjos

$k1 \leftarrow 1$  //  $k1$  é o índice que percorre L

$k2 \leftarrow 1$  //  $k2$  é o índice que percorre R

para  $k \leftarrow i$  até  $f$  //  $k$  percorre A

se  $L[k1] \leq R[k2]$

$A[k] \leftarrow L[k1]$

$k1 \leftarrow k1 + 1$

senão

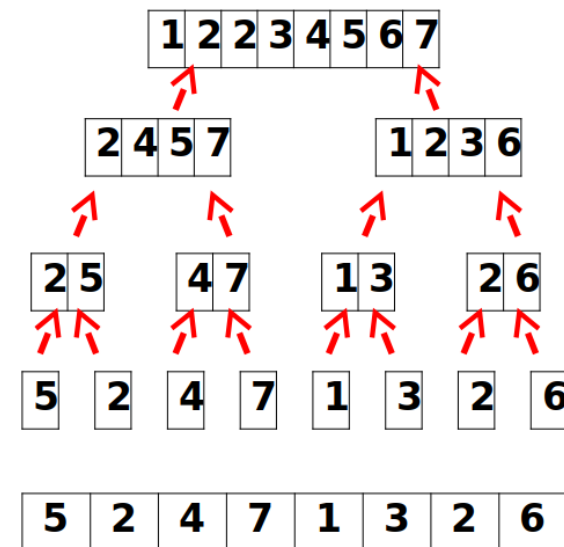
$A[k] \leftarrow R[k2]$

$k2 \leftarrow k2 + 1$

fim se

fim para

In loco?: ?



# MergeSort (ordenação por intercalação)

**mergeSort (A, i, f)**

se  $i < f$

$m \leftarrow \lfloor (i+f)/2 \rfloor$

mergeSort(A, i, m)

mergeSort(A, m+1, f)

**merge(A, i, m, f)**

**merge** (A, i, m, f)

$n1 \leftarrow m-i+1$  // define subarranjos

$n2 \leftarrow f-m$

criar arranjos L[1..n1+1] e R[1..n2+1]

$L[n1+1] \leftarrow \infty$ ;  $R[n2+1] \leftarrow \infty$  //sentinela

para  $j \leftarrow 1$  até  $n1$   $L[j] \leftarrow A[i+j-1]$

para  $j \leftarrow 1$  até  $n2$   $R[j] \leftarrow A[m+j]$

// mesclar subarranjos

$k1 \leftarrow 1$  //  $k1$  é o índice que percorre L

$k2 \leftarrow 1$  //  $k2$  é o índice que percorre R

para  $k \leftarrow i$  até  $f$  //  $k$  percorre A

se  $L[k1] \leq R[k2]$

$A[k] \leftarrow L[k1]$

$k1 \leftarrow k1 + 1$

senão

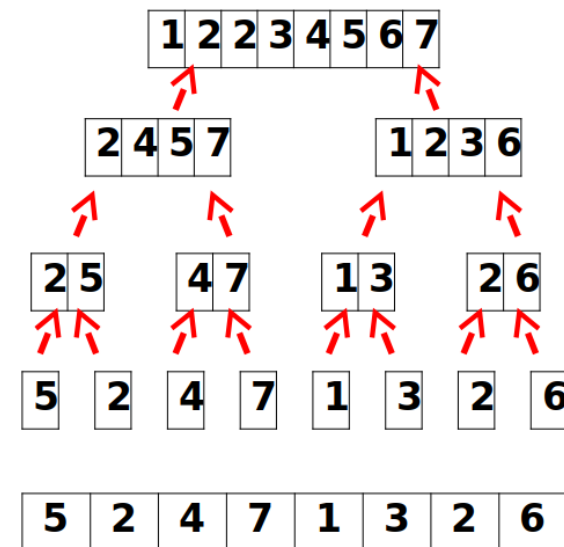
$A[k] \leftarrow R[k2]$

$k2 \leftarrow k2 + 1$

fim se

fim para

In loco?: Não...



# MergeSort (ordenação por intercalação)

**mergeSort (A, i, f)**

se  $i < f$

$m \leftarrow \lfloor (i+f)/2 \rfloor$

mergeSort(A, i, m)

mergeSort(A, m+1, f)

**merge(A, i, m, f)**

**merge** (A, i, m, f)

$n1 \leftarrow m-i+1$  // define subarranjos

$n2 \leftarrow f-m$

criar arranjos L[1..n1+1] e R[1..n2+1]

$L[n1+1] \leftarrow \infty$ ;  $R[n2+1] \leftarrow \infty$  //sentinela

para  $j \leftarrow 1$  até  $n1$   $L[j] \leftarrow A[i+j-1]$

para  $j \leftarrow 1$  até  $n2$   $R[j] \leftarrow A[m+j]$

// mesclar subarranjos

$k1 \leftarrow 1$  //  $k1$  é o índice que percorre L

$k2 \leftarrow 1$  //  $k2$  é o índice que percorre R

para  $k \leftarrow i$  até  $f$  //  $k$  percorre A

se  $L[k1] \leq R[k2]$

$A[k] \leftarrow L[k1]$

$k1 \leftarrow k1 + 1$

senão

$A[k] \leftarrow R[k2]$

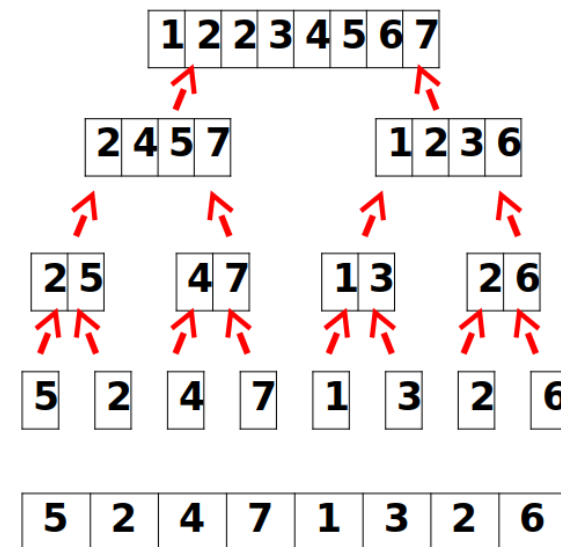
$k2 \leftarrow k2 + 1$

fim se

fim para

In loco?: Não...

Estável?:



# MergeSort (ordenação por intercalação)

```
mergeSort (A, i, f)
```

```
se i < f
```

```
    m ←  $\lfloor (i+f)/2 \rfloor$ 
```

```
    mergeSort(A, i, m)
```

```
    mergeSort(A, m+1, f)
```

```
    merge(A, i, m, f)
```

```
merge (A, i, m, f)
```

```
    n1 ← m-i+1 // define subarranjos
```

```
    n2 ← f-m
```

```
    criar arranjos L[1..n1+1] e R[1..n2+1]
```

```
    L[n1+1] ←  $\infty$ ; R[n2+1] ←  $\infty$  //sentinela
```

```
    para j ← 1 até n1  L[j] ← A[i+j-1]
```

```
    para j ← 1 até n2  R[j] ← A[m+j]
```

```
    // mesclar subarranjos
```

```
    k1 ← 1 // k1 é o índice que percorre L
```

```
    k2 ← 1 // k2 é o índice que percorre R
```

```
    para k ← i até f // k percorre A
```

```
        se L[k1] ≤ R[k2]
```

```
            A[k] ← L[k1]
```

```
            k1 ← k1 + 1
```

```
        senão
```

```
            A[k] ← R[k2]
```

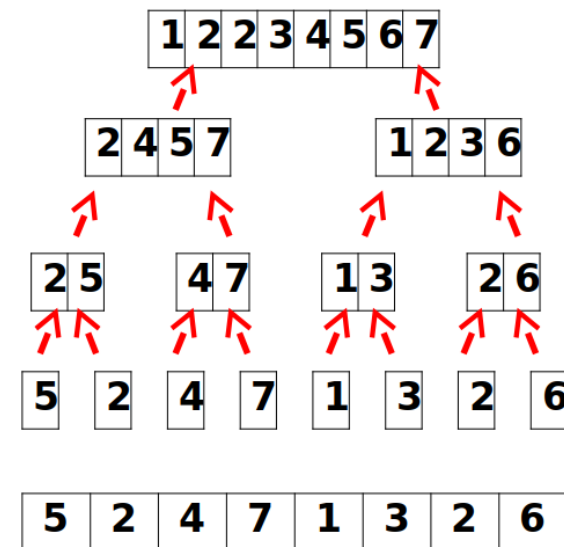
```
            k2 ← k2 + 1
```

```
    fim se
```

```
    fim para
```

In loco?: Não...

Estável?: SIM! Por quê?



# MergeSort (ordenação por intercalação)

**mergeSort (A, i, f)**

se  $i < f$

$m \leftarrow \lfloor (i+f)/2 \rfloor$

mergeSort(A, i, m)

mergeSort(A, m+1, f)

**merge(A, i, m, f)**

**merge (A, i, m, f)**

$n1 \leftarrow m-i+1$  // define subarranjos

$n2 \leftarrow f-m$

criar arranjos L[1..n1+1] e R[1..n2+1]

$L[n1+1] \leftarrow \infty$ ;  $R[n2+1] \leftarrow \infty$  //sentinela

para  $j \leftarrow 1$  até  $n1$   $L[j] \leftarrow A[i+j-1]$

para  $j \leftarrow 1$  até  $n2$   $R[j] \leftarrow A[m+j]$

// mesclar subarranjos

$k1 \leftarrow 1$  //  $k1$  é o índice que percorre L

$k2 \leftarrow 1$  //  $k2$  é o índice que percorre R

para  $k \leftarrow i$  até  $f$  //  $k$  percorre A

se  $L[k1] \leq R[k2]$

$A[k] \leftarrow L[k1]$

$k1 \leftarrow k1 + 1$

senão

$A[k] \leftarrow R[k2]$

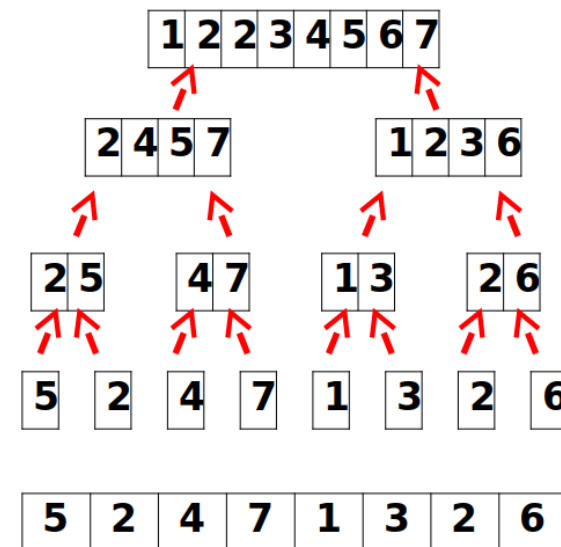
$k2 \leftarrow k2 + 1$

fim se

fim para

In loco?: Não...

Estável?: SIM! Por quê? Duas coisas...



# MergeSort (ordenação por intercalação)

**mergeSort (A, i, f)**

se  $i < f$

$m \leftarrow \lfloor (i+f)/2 \rfloor$

mergeSort(A, i, m)

mergeSort(A, m+1, f)

merge(A, i, m, f)

merge (A, i, m, f)

$n1 \leftarrow m-i+1$  // define subarranjos

$n2 \leftarrow f-m$

criar arranjos L[1..n1+1] e R[1..n2+1]

$L[n1+1] \leftarrow \infty$ ;  $R[n2+1] \leftarrow \infty$  //sentinela

para  $j \leftarrow 1$  até  $n1$   $L[j] \leftarrow A[i+j-1]$

para  $j \leftarrow 1$  até  $n2$   $R[j] \leftarrow A[m+j]$

// mesclar subarranjos

$k1 \leftarrow 1$  //  $k1$  é o índice que percorre L

$k2 \leftarrow 1$  //  $k2$  é o índice que percorre R

para  $k \leftarrow i$  até  $f$  //  $k$  percorre A

se  $L[k1] \leq R[k2]$

$A[k] \leftarrow L[k1]$

$k1 \leftarrow k1 + 1$

senão

$A[k] \leftarrow R[k2]$

$k2 \leftarrow k2 + 1$

fim se

fim para

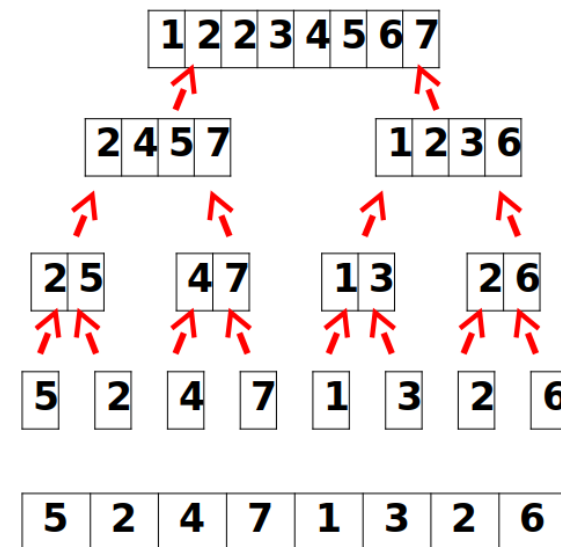
In loco?: Não...

Estável?: SIM! Por quê? Duas coisas...

1) L e R são vetores vizinhos

2) Na hora de intercalar, L tem preferência sobre R

(quando  $L[k1] = R[k2]$ )



# Comparando...

	T(n)			C(n)			M(n)			in loco?	estável?
Algoritmo	Melhor caso	Pior caso	Caso médio	Melhor caso	Pior caso	Caso médio	Melhor caso	Pior caso	Caso médio		
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n^2)$	sim	sim
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n)$	$O(n)$	sim	não
BubbleSort											
MergeSort	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	não	sim

- O que podem falar agora do MergeSort (em comparação com os demais?)

# Comparando...

Algoritmo	T(n)			C(n)			M(n)			in loco?	estável?
	Melhor caso	Pior caso	Caso médio	Melhor caso	Pior caso	Caso médio	Melhor caso	Pior caso	Caso médio		
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n^2)$	sim	sim
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n)$	$O(n)$	sim	não
BubbleSort											
MergeSort	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	não	sim

- O que podem falar agora do MergeSort (em comparação com os demais)?
  - Boa escolha se o arquivo é grande e desordenado, desde que tenha memória para isso (se for pequeno o selection na prática deve valer a pena)



# Obs: ShellSort

O mais eficiente dos algoritmos de complexidade quadrática

- Proposto por Shell em 1959.
- É uma extensão do algoritmo de ordenação por inserção.
- Problema com o algoritmo de ordenação por inserção:
  - Troca itens adjacentes para determinar o ponto de inserção.
  - São efetuadas  $n - 1$  comparações e movimentações quando o menor item está na posição mais à direita no vetor.
- O método de Shell contorna este problema permitindo trocas de registros distantes um do outro.

# ShellSort

- Os itens separados de  $h$  posições são rearranjados.
- Todo  $h$ -ésimo item leva a uma seqüência ordenada.
- Tal seqüência é dita estar  $h$ -ordenada.
- Exemplo de utilização:

	1	2	3	4	5	6
Chaves iniciais:	<i>O</i>	<i>R</i>	<i>D</i>	<i>E</i>	<i>N</i>	<i>A</i>
$h = 4$	<i>N</i>	<i>A</i>	<i>D</i>	<i>E</i>	<i>O</i>	<i>R</i>
$h = 2$	<i>D</i>	<i>A</i>	<i>N</i>	<i>E</i>	<i>O</i>	<i>R</i>
$h = 1$	<i>A</i>	<i>D</i>	<i>E</i>	<i>N</i>	<i>O</i>	<i>R</i>

- Quando  $h = 1$  Shellsort corresponde ao algoritmo de inserção.

# ShellSort

- Como escolher o valor de  $h$ : O cálculo é iterativo, partindo de  $h(1) = 1$

- Seqüência para  $h$ :

$h(s)$  é o  $s$ -ésimo cálculo de  $h$

$$h(s) = 3h(s - 1) + 1, \quad \text{para } s > 1$$

$$h(s) = 1, \quad \text{para } s = 1.$$

- Knuth (1973, p. 95) mostrou experimentalmente que esta seqüência é difícil de ser batida por mais de 20% em eficiência.
- A seqüência para  $h$  corresponde a 1, 4, 13, 40, 121, 364, 1.093, 3.280, ... Continua enquanto  $h(s) < n$ , e daí aplica-se esse maior  $h$ , e depois os inferiores ( $h(s)$ ,  $h(s-1)$ ,  $h(s-2)$ , ...,  $h(1)$ )

# Obs: ShellSort

```
void shellSort(int n, int[] v) {  
    int i, j, h, chave;  
    for (h = 1; h < n; h = 3*h+1); // calcula-se o máximo h  
    while (h > 0) { // faz um insertionSort para cada h da série  
        h = (h-1)/3;  
        for(j = h; j < n; j++) {  
            chave = v[j];  
            i = j; // vou olhar a posição i - h  
            while (i >= h && v[i - h] > chave) {  
                v[i] = v[i - h];  
                i = i - h;  
            }  
            v[i] = chave;  
        }  
    }  
}
```

para garantir  
que  $i-h \geq 0$

**InsertionSort – versão  
Sem sentinela**

```
for (j = 1; j < n; j++) {  
    chave = v[j];  
    i = j - 1;  
    while (i >= 0 &&  
           v[i] > chave){  
        v[i+1] = v[i];  
        i = i - 1;  
    }  
    v[i+1] = chave;  
}
```

# ShellSort

- A implementação do Shellsort não utiliza registros **sentinelas**.
- Seriam necessários  $h$  registros sentinelas, uma para cada  $h$ -ordenação.

# ShellSort

- A razão da eficiência do algoritmo ainda não é conhecida.
- Ninguém ainda foi capaz de analisar o algoritmo.
- A sua análise contém alguns problemas matemáticos muito difíceis.
- A começar pela própria seqüência de incrementos.
- O que se sabe é que cada incremento não deve ser múltiplo do anterior.
- Conjecturas referente ao número de comparações para a seqüência de Knuth:

$$\text{Conjetura 1} : C(n) = O(n^{1,25})$$

$$\text{Conjetura 2} : C(n) = O(n(\ln n)^2)$$

# ShellSort

- Vantagens:
  - Shellsort é uma ótima opção para arquivos de tamanho moderado.
  - Sua implementação é simples e requer uma quantidade de código pequena.
- Desvantagens:
  - O tempo de execução do algoritmo é sensível à ordem inicial do arquivo.
  - O método não é **estável**,

## Tempo de execução:

# Comparação (Ziviani)

- Observação: O método que levou menos tempo real para executar recebeu o valor 1 e os outros receberam valores relativos a ele.
- Registros na ordem aleatória:

	5.00	5.000	10.000	30.000
Inserção	11,3	87	161	—
Seleção	16,2	124	228	—
Shellsort	1,2	1,6	1,7	2

- Registros na ordem ascendente:

	500	5.000	10.000	30.000
Inserção	1	1	1	1
Seleção	128	1.524	3.066	—
Shellsort	3,9	6,8	7,3	8,1

- Registros na ordem descendente:

	500	5.000	10.000	30.000
Inserção	40,3	305	575	—
Seleção	29,3	221	417	—
Shellsort	1,5	1,5	1,6	1,6



# Referências

- Paulo Feofiloff. Algoritmos em C. Cap 8 e 9 (**tem exercícios!!!**) <https://www.ime.usp.br/~pf/algoritmos-livro/>
- Nívio Ziviani. Projeto de Algoritmos com implementações em C e Pascal. Editora Thomson, 3a. Edição, 2004. Cap 4.1.1 a 4.1.3 - <http://www2.dcc.ufmg.br/livros/algoritmos/implementacoes.php>
- Apostila Prof. Marcio – cap 6.1 a 6.3