Conteúdo

1. Introdução
2. Listas
3. Pilhas e Filas
4. Árvores
5. Árvores de Pesquisa
   - Árvore Binária e Árvore AVL
   - Árvore N-ária e Árvore B
6. Tabelas de Dispersão (Hashing)
7. Métodos de Acesso a Arquivos
8. Métodos de Ordenação de Dados

Métodos de Ordenação de Dados

Ordenação de Dados

Objetivo: Manter os dados ordenados para facilitar a pesquisa

⇨ Dados ordenados garantem uma melhor performance de pesquisa

- busca seqüencial
  - evita a varredura completa de uma lista de dados
- busca binária
  - só é possível se os dados estão ordenados
  - apresenta baixa complexidade

Ordenação de Dados

"A complexidade da ordenação da Estrutura de Dados não deve exceder a complexidade da computação a ser feita na Estrutura de Dados sem o processo de ordenação"

Exemplo: deseja-se realizar uma única pesquisa a um vetor

- busca seqüencial $\Rightarrow$ *O(n)*
- ordenação $\Rightarrow$ *O(n log n)*
  - $\hookrightarrow$ não vale a pena ordenar!

Ordenação de Dados

Nível Lógico: os algoritmos recebem uma lista não ordenada e a convertem em uma lista ordenada.

Nível Físico (Implementação): os algoritmos recebem um array e organizam seus valores para que eles fiquem ordenados pelas chaves.

As estruturas de dados que serão ordenadas possuem:

- dados que estão mantidos em um array
- elementos que são objetos que podem ser comparáveis

Métodos de Ordenação de Dados

Métodos de Ordenação mais conhecidos

- Selection Sort
- Bubble Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Heap Sort

Revisão do Somatório

Propriedade 1 (P1)

$$\sum_{i=1}^{n} i = \frac{n\,(n+1)}{2}$$

Propriedade 2 (P2)

$$\sum_{i=1}^{n} k\,i = k \sum_{i=1}^{n} i$$

Métodos de Ordenação Simples

Métodos de Ordenação Simples

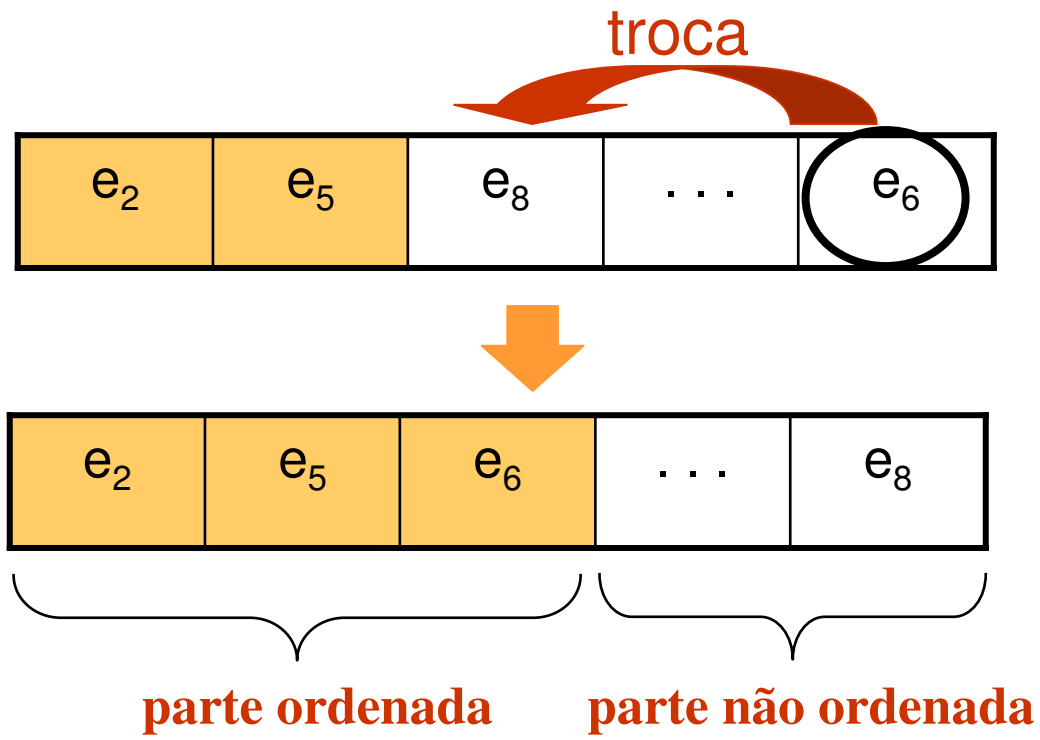
- Selection Sort
- Bubble Sort
- Insertion Sort

Características:

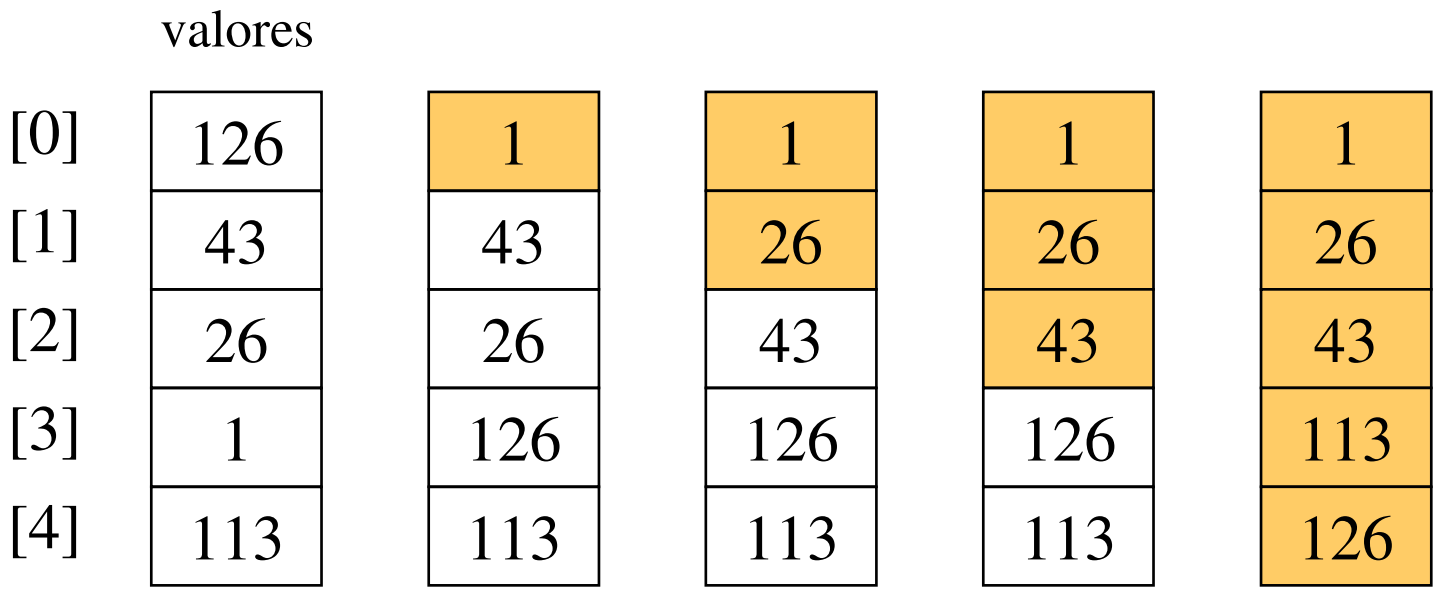
- fáceis de implementar e de entender
- não são muito eficientes

Selection Sort - (Método da Seleção)

Ordena através de sucessivas buscas pelo elemento de menor valor em uma **parte não-ordenada** e seu posicionamento no final de uma **parte ordenada**.



Selection Sort - Exemplo



Selection Sort - Algoritmo

posicao-corrente ← índice do primeiro item do array

enquanto posicao-corrente < fim // existem mais elementos na parte não ordenada do array

percorre da posicao-corrente até o fim procurando a posição do menor elemento // parte não ordenada

troca (array[posição_corrente], array[posição do menor elemento na parte não ordenada)

posição_corrente ++ // diminui a parte não ordenada

Selection Sort - Exemplo

Exemplo 1

[0]	[1]	[2]	[3]	[4]	[5]	[6]
70	55	45	50	30	40	20

Exemplo 2

[0]	[1]	[2]	[3]	[4]	[5]	[6]
20	30	40	45	50	55	60

Selection Sort - Complexidade

- Para qualquer caso

Comparações em cada varredura

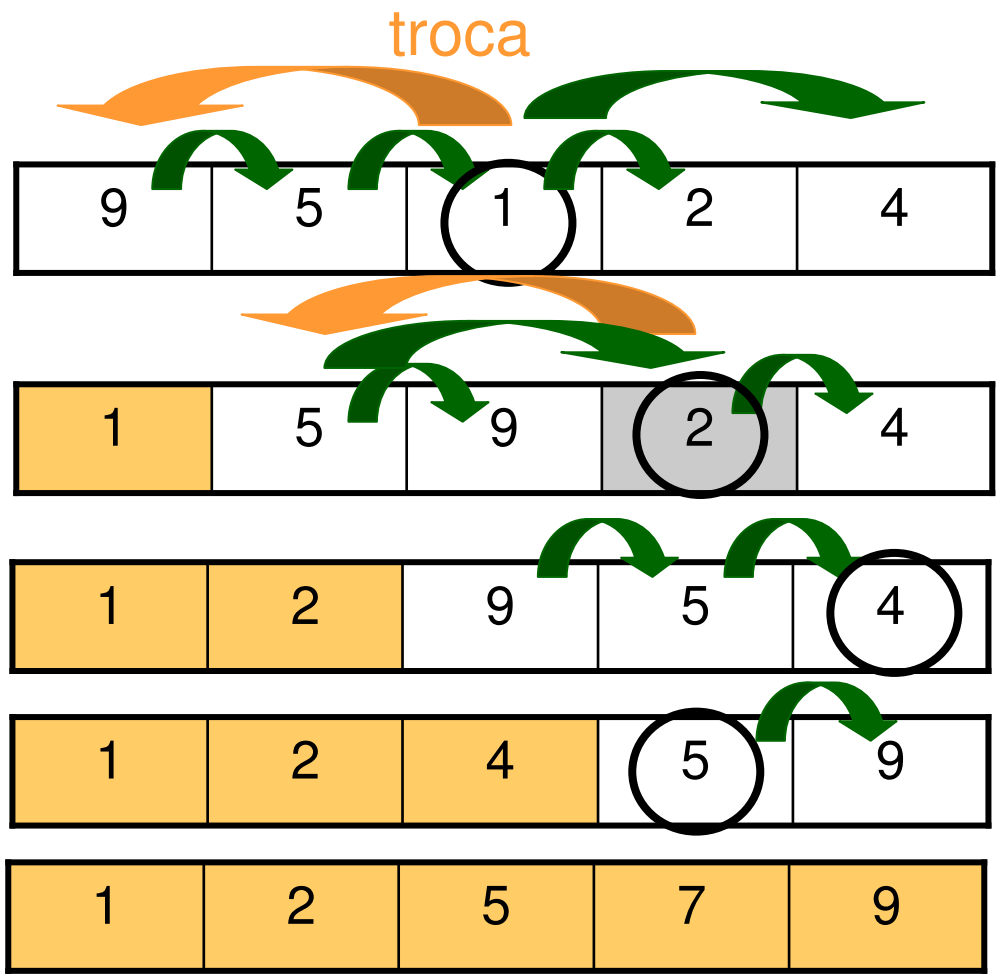
1ª V: n – 1

2ª V: n – 2

. . .

(n-2)ª V: 2

(n-1)ª V: 1



Selection Sort - Complexidade

- Definida pelo número de comparações envolvendo a quantidade de elementos do array
- Número de comparações:

$$(n - 1) + (n - 2) + \ldots + 2 + 1$$

- Complexidade (para qualquer caso):

$$\sum_{i=1}^{n-1} i = \frac{(n - 1)\, n}{2} \Rightarrow \boldsymbol{O(n^2)}$$

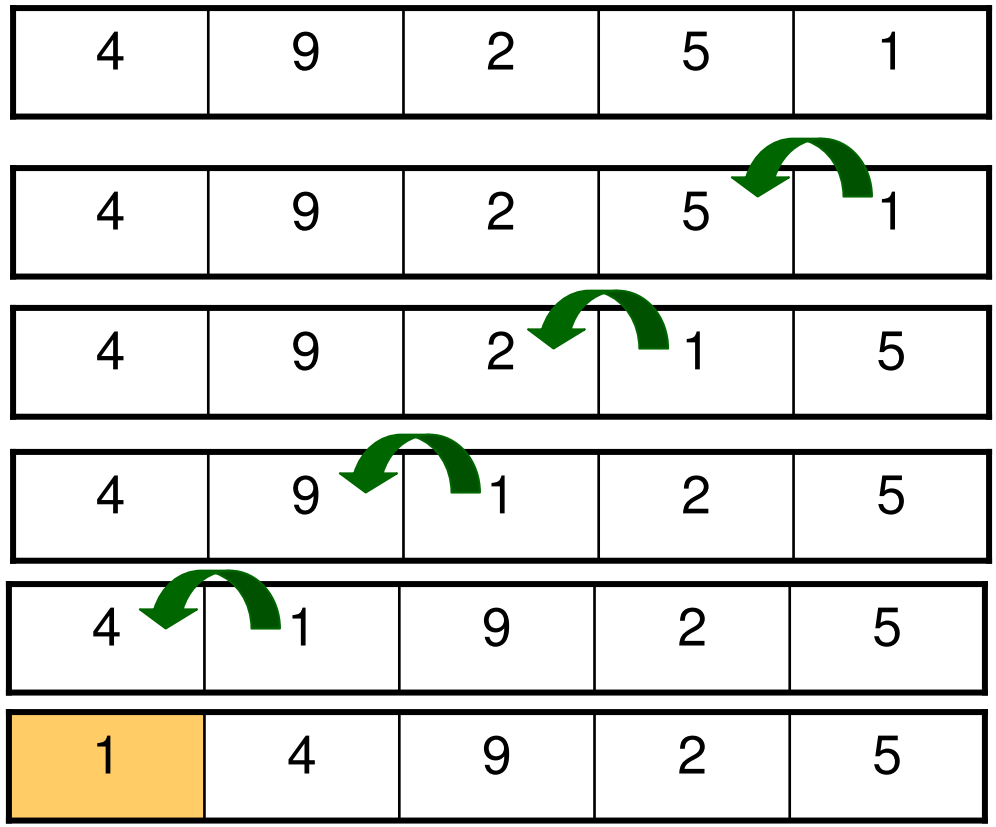
⇨ Mesmo que o array já esteja ordenado, o método faz N(N-1)/2 comparações

Selection Sort - Complexidade

Número de Itens	Número de Comparações
10	45
20	190
100	4.950
1.000	499.500
10.000	49.995.000

Bubble Sort (Método da Bolha)

Ordena através de sucessivas trocas entre pares de elementos do array sempre que um elemento for menor do que o anterior.



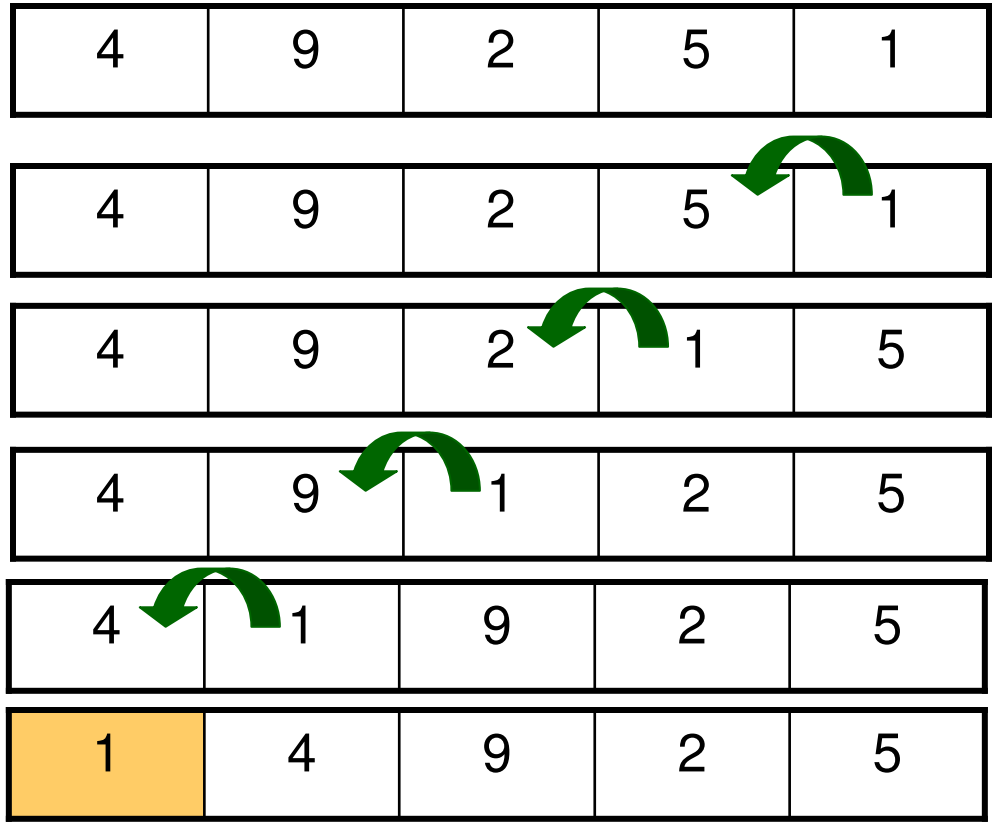
Bubble Sort (Método da Bolha)

Cada iteração coloca o menor elemento da parte não ordenada na sua posição correta, mas também muda as posições dos outros elementos do array.

- desta maneira, o menor elemento “bubbles up” para o topo do array

Bubble Sort

1ª iteração



→ Enquanto o Selection Sort tem somente uma troca por iteração, o Bubble Sort pode fazer várias trocas por iteração.

Bubble Sort

Exemplo

	valores				
[0]	36	6	6	6	6
[1]	24	36	10	10	10
[2]	10	24	36	12	12
[3]	6	10	24	36	24
[4]	12	12	12	24	36

Bubble Sort - Algoritmo

```
posicao-corrente ← índice do primeiro item do array

enquanto posicao-corrente < fim   // existem mais elementos
                                     na parte não ordenada do array

    pos  ← fim

    enquanto pos > posicao-corrente  // "bubble up" o menor
                                        elemento da parte não ordenada

        se array[pos] < array[pos-1]

                troca (array[pos], array[pos-1])

        pos--

    posição-corrente ++  // diminui a parte não ordenada
```

Bubble Sort - Exemplo

Exemplo 1

[0]	[1]	[2]	[3]	[4]	[5]	[6]
70	55	45	50	30	40	20

Exemplo 2

[0]	[1]	[2]	[3]	[4]	[5]	[6]
20	30	40	45	50	55	60

Bubble Sort - Complexidade

- Para qualquer caso

n = 5

Comparações em cada varredura

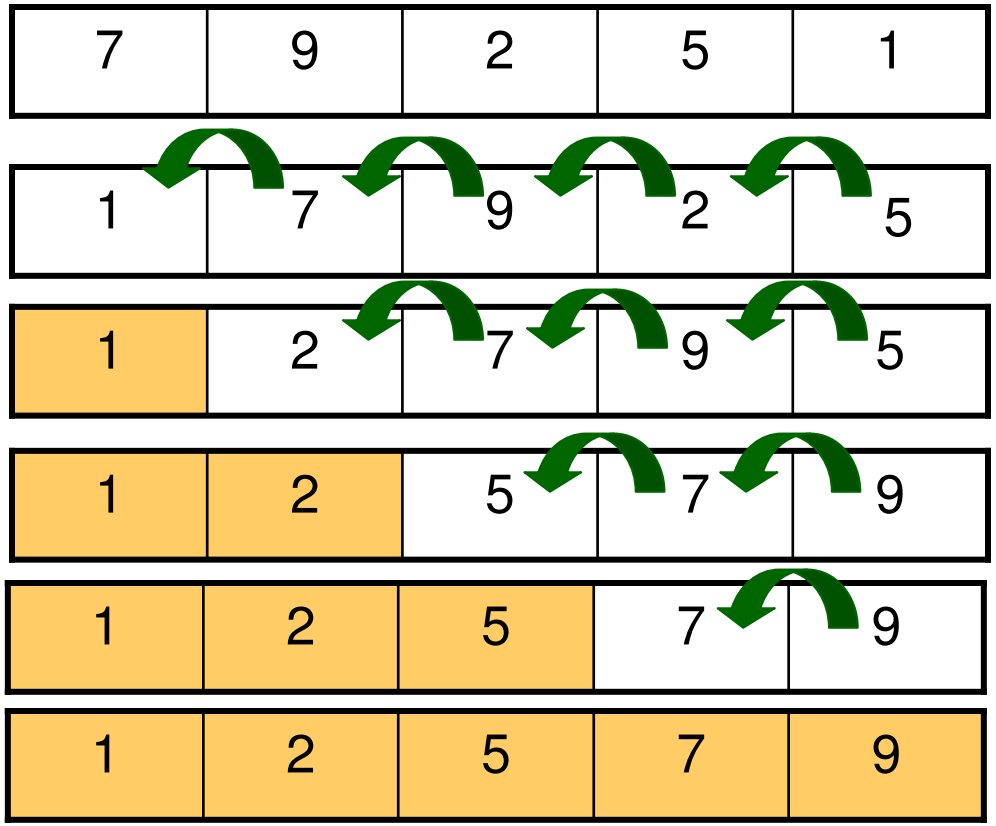
1ª V: n – 1

2ª V: n – 2

⋮

(n-2)ª V: 2

(n-1)ª V: 1



Bubble Sort - Complexidade

- Definida pelo número de comparações envolvendo a quantidade de elementos do array
- Número de comparações:

$$(n - 1) + (n - 2) + \ldots + 2 + 1$$

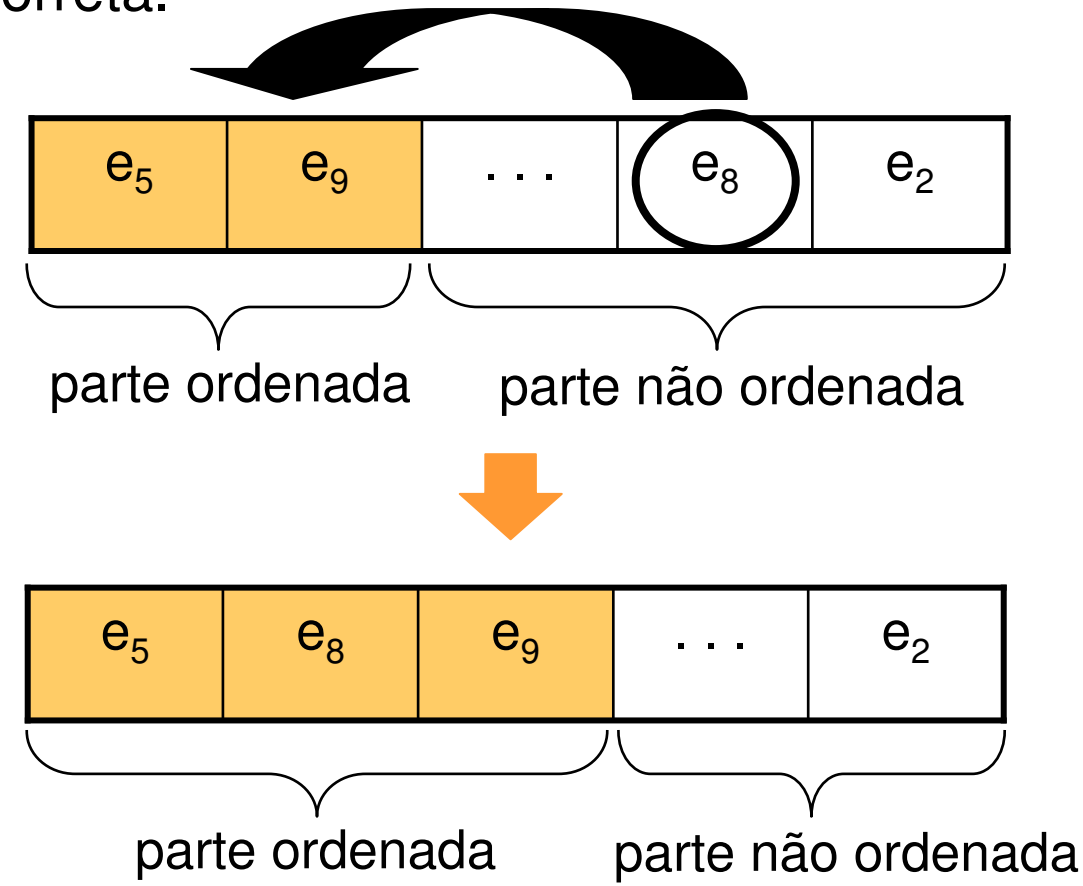
- Complexidade (para qualquer caso):

$$\sum_{i=1}^{n-1} i = \frac{(n-1)\,n}{2} \Rightarrow \boldsymbol{O(n^2)}$$

⇨ Mesmo que o array já esteja ordenado, o método faz N(N-1)/2 comparações, mesmo que nenhuma troca seja feita

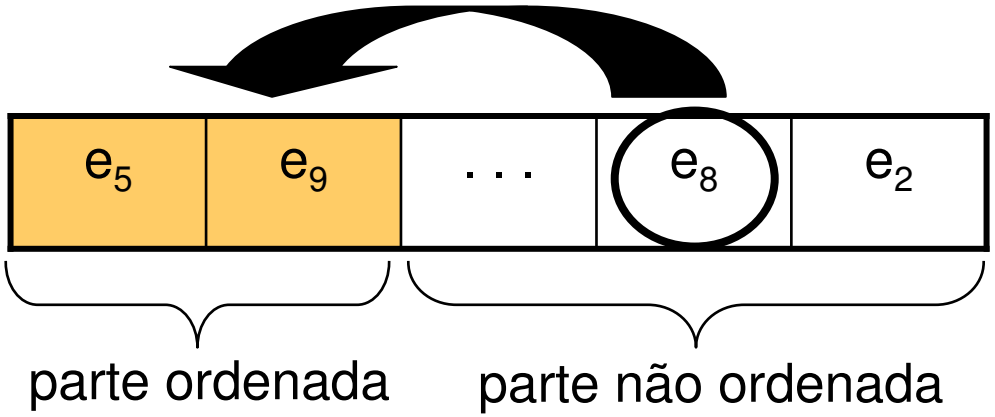
Insertion Sort (Método da Inserção)

Ordena através da inserção de um elemento por vez (primeiro elemento) da parte **não-ordenada** na parte **ordenada**, na sua posição correta.



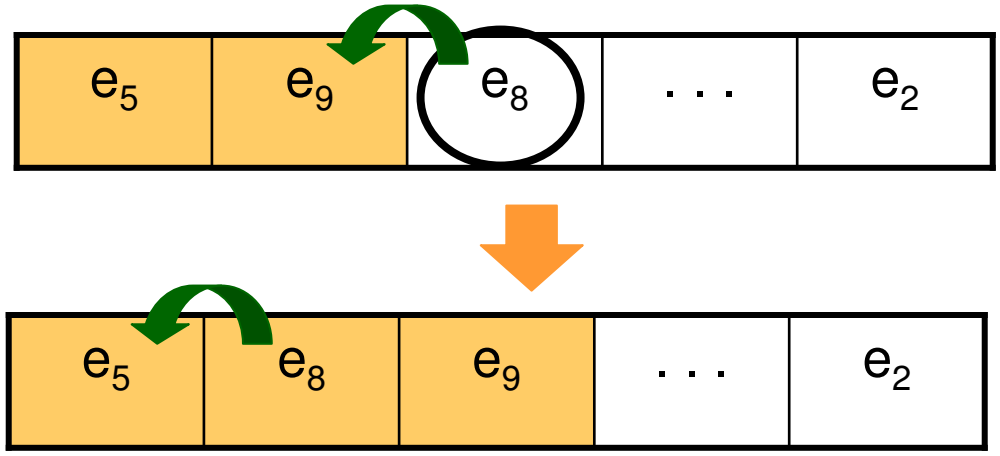
Insertion Sort

- Inicialmente, a parte ordenada contém apenas o primeiro elemento do vetor



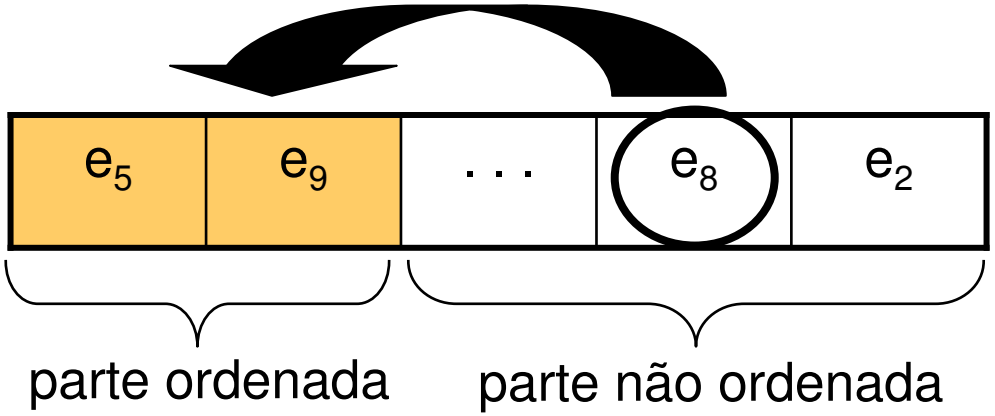
Insertion Sort

- Realiza uma busca seqüencial na parte ordenada para inserir corretamente um elemento da parte não-ordenada
- Nesta busca, realiza trocas entre elementos adjacentes para ir acertando a posição do elemento a ser inserido



Insertion Sort

→ Diferente do Selection Sort e Bubble Sorte, no Insertion Sort podem existir na parte não ordenada elementos menores do que os elementos da parte ordenada.



Insertion Sort

Funcionamento

	valores				
[0]	36	24	10	6	6
[1]	24	36	24	10	10
[2]	10	10	36	24	12
[3]	6	6	6	36	24
[4]	12	12	12	12	36

Insertion Sort - Algoritmo

```
para contador de 1 até tamanho - 1
    posicao-corrente ← contador
    ordenou ← false
   enquanto tem-elemento-para-analisar E NÃO ordenou
        se array [posição_corrente] < array [posição_corrente-1]
            troca (array[posição-corrente], array [posição-corrente-1])
            posição-corrente --
            tem-elemento-para-analisar ← posição-corrente não é
         igual a 0
        senão
            ordenou ← true
```

Insertion Sort - Exemplo

Exemplo 1

[0]	[1]	[2]	[3]	[4]	[5]	[6]
70	55	45	50	30	40	20

Exemplo 2

[0]	[1]	[2]	[3]	[4]	[5]	[6]
20	30	40	45	50	55	60

Insertion Sort - Complexidade

Pior caso: vetor totalmente desordenado

n = 5

Comparações em cada varredura

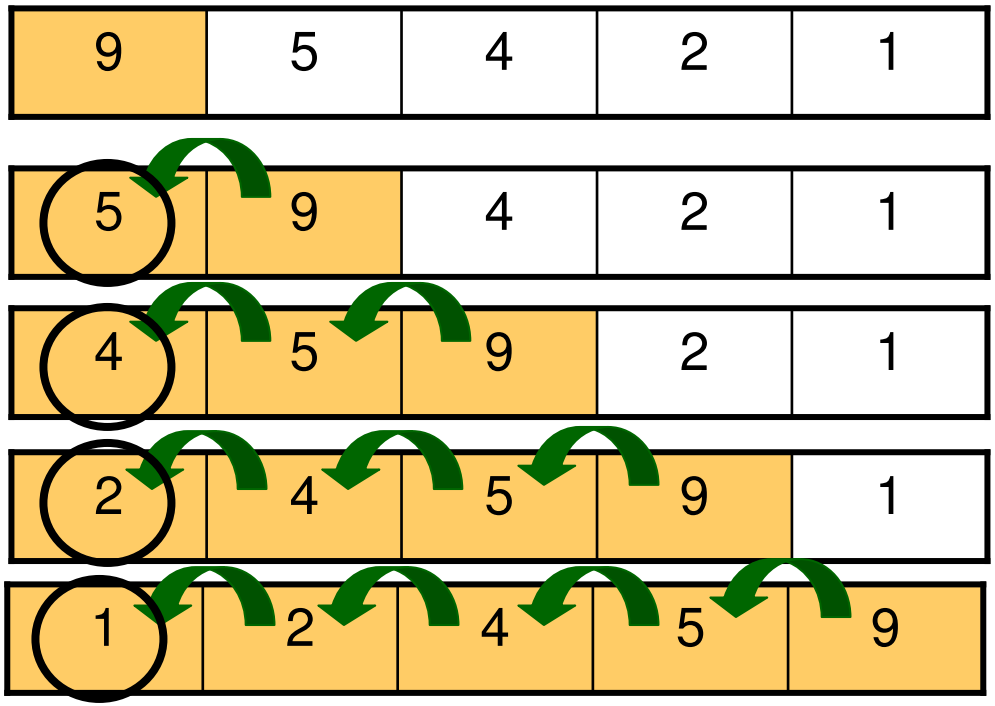
1ª V: 1

2ª V: 2

. . .

(n-2)ª V: n - 2

(n-1)ª V: n - 1



Insertion Sort - Complexidade

Melhor caso: vetor já ordenado

n = 5

Comparações em cada varredura

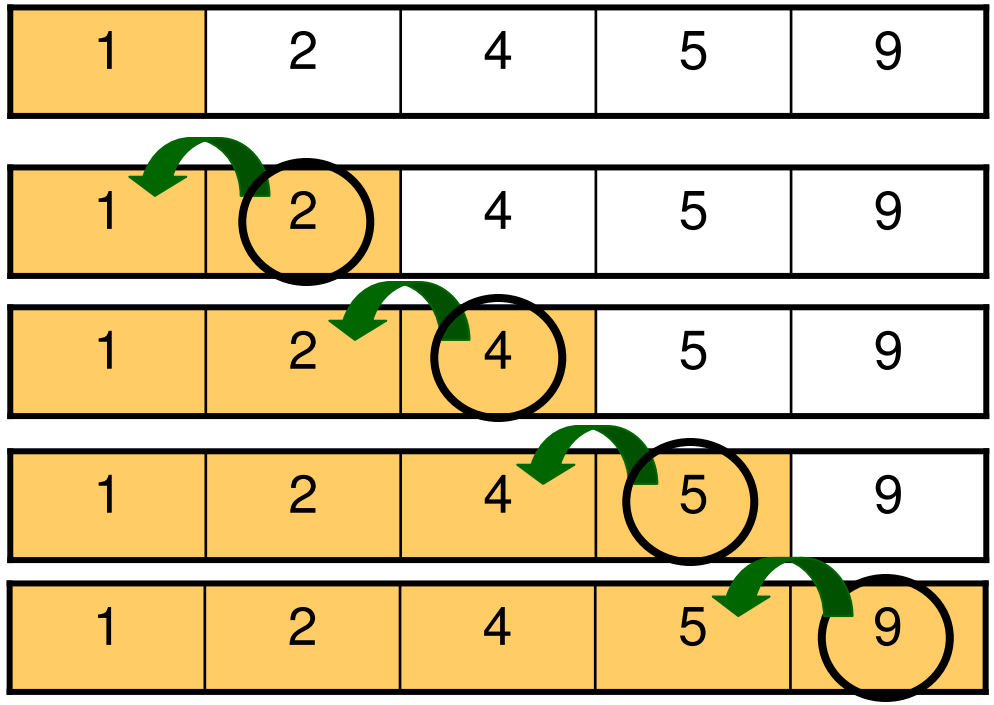
1ª V: 1

2ª V: 1

...

(n-2)ª V: 1

(n-1)ª V: 1



Insertion Sort - Complexidade

- Definida pelo número de comparações envolvendo a quantidade de elementos do array
- Número de comparações e Complexidade:

pior caso (desordenado): (n - 1) + (n – 2) + ... + 2 + 1

$$\sum_{i=1}^{n-1} i = \frac{(n-1)\,n}{2} \Rightarrow \boldsymbol{O(n^2)}$$

melhor caso (ordenado): 1 + 1 + ... + 1 + 1

n - 1 $\Rightarrow$ ***O(n)***

Comparação

	Melhor caso	Pior caso
Selection Sort	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$

Métodos de Ordenação

Simulação de funcionamento

http://math.hws.edu/TMCM/java/xSortLab

Implementação

- Implementar os seguintes algoritmos de acordo com a hierarquia apresentada a seguir:
  - Selection Sort
  - Bubble Sort
  - Insertion Sort

Implementação

Hierarquia das Classes de Ordenação