

# Algoritmos de Ordenação Quadráticos

Pesquisa, Ordenação e Técnicas de Armazenamento

*Prof. Me. Orlando da Silva Junior*

# Agenda

- ❑ Ordenação interna e externa
- ❑ *Bubble sort*
- ❑ *Selection sort*
- ❑ *Insertion sort*
- ❑ Análise de performance
- ❑ Aplicabilidade dos algoritmos

*Assim como o problema da busca, o problema da ordenação também é um problema clássico da computação.*

- O problema da ordenação trata de arranjar uma sequência de elementos em alguma **ordem**.
- Com dados ordenados, o acesso a esses dados pode ser feito de maneira mais eficiente.



Para escolher um bom algoritmo, é preciso considerar:

- ☐ **Complexidade computacional:** *em quanto tempo ele ordena?*
- ☐ **Complexidade espacial:** quanto de memória gasta?
- ☐ **Estabilidade das ordenações:** o algoritmo mantém a ordem original para repetições?

# Métodos de ordenação

- **Quadráticos  $O(n^2)$** 
  - Bubble Sort
  - Insertion Sort
  - Selection Sort
- **Recursivos e n-logarítmicos  $O(n \log_2 n)$** 
  - Merge Sort
  - Quick Sort
  - Heap Sort
  - Shell Sort
- **Lineares  $O(n)$** 
  - Radix Sort
  - Count Sort
  - Bucket Sort

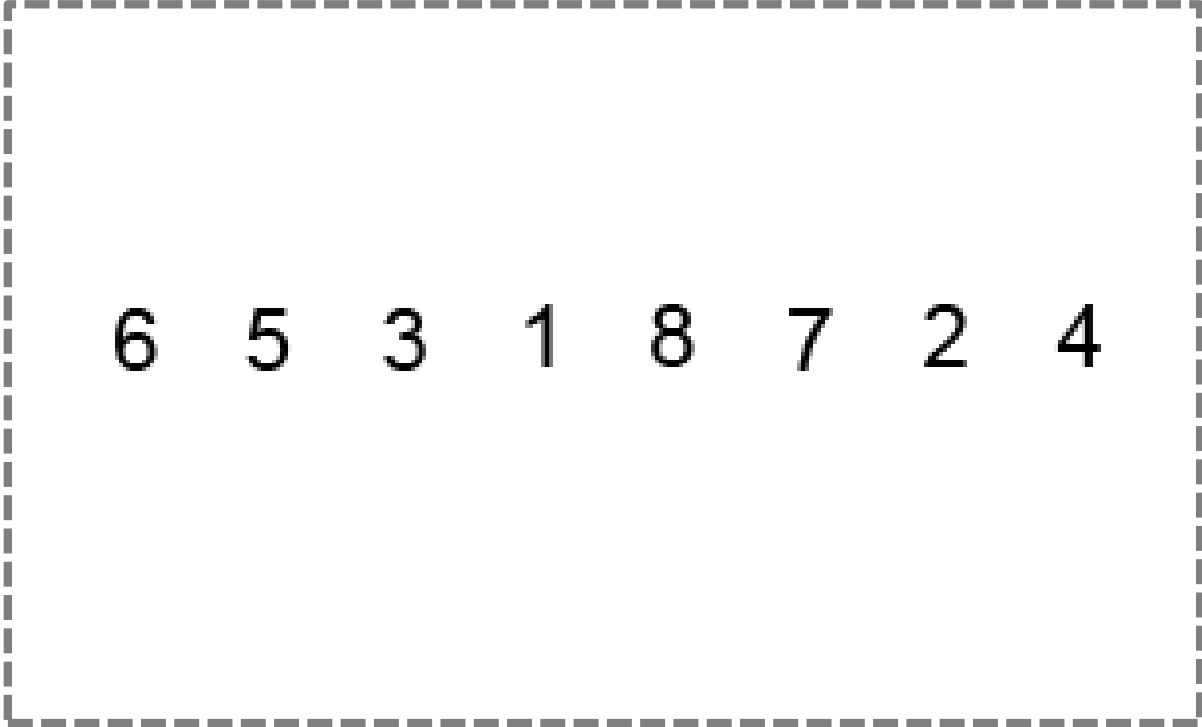


# Bubble sort

**Ideia geral:** fazer com que os elementos “borbulhem” a cada iteração até que estejam em sua posição correta.

**Algoritmo:** percorra o vetor desde o início comparando os elementos adjacentes, dois a dois. Troque as posições dos elementos se eles estiverem fora de ordem. Repita os dois passos anteriores com os primeiros  $n-1$  elementos, depois com os primeiros  $n-2$  itens, ..., até que reste apenas um elemento.

- **Vantagens:** é o algoritmo de ordenação popular mais simples e fácil de ser implementado; estável.
- **Desvantagens:** como os elementos são trocados frequentemente, há um alto custo.



6 5 3 1 8 7 2 4

# Bubble sort – análise de complexidade

		custo	melhor caso	pior caso
	<b>Procedimento</b> bubbleSort(V[0..N]: vetor, N: inteiro)			
	<b>Início</b>			
1	AUX: inteiro			
2	Para i de 0 até N passo 1 Faça	c1	N	N
3	Para j de i até N passo 1 Faça	c2	N	N
4	Se V[j] < V[i] Então	c3	N	N
5	AUX ← V[j]	c4	1	N
6	V[j] ← V[i]	c5	1	N
7	V[i] ← AUX	c6	1	N
8	Fim-se			
9	Fim-para			
10	Fim-para			
	<b>Fim</b>			

# Bubble sort – análise de complexidade

## Perguntas importantes:

1. Quantas comparações são feitas?
2. Quantas trocas são feitas?

**Implemente e faça a contagem para um vetor de 100 elementos.**

## Melhor caso:

Já está ordenado  
Comparações:  $O(n^2)$   
Trocas: nenhuma

## Pior caso:

Está completamente desordenado  
Comparações:  $O(n^2)$   
Trocas:  $O(n)$

```
Procedimento bubbleSort (V[0..N]: vetor, N: inteiro)
Início
    AUX: inteiro
    Para i de 0 até N passo 1 Faça
        Para j de i até N passo 1 Faça
            Se V[j] < V[i] Então
                AUX ← V[j]
                V[j] ← V[i]
                V[i] ← AUX
            Fim-se
        Fim-para
    Fim-para
Fim
```

*A análise pode melhorar conforme a implementação.*

# Bubble sort – como melhorar?

```
public void bubbleSort2(int[] v, int n) {  
    boolean trocou = true;  
    for (int i = n - 1; i > 0 && trocou; i--) {  
        trocou = false;  
        for (int j = 0; j < i; j++) {  
            if (v[j] > v[j + 1]) {  
                troca(v, j, j + 1);  
                trocou = true;  
            }  
        }  
    }  
}
```

*A implementação ao lado  
permite que a execução  
abandone o ciclo quando  
a lista de elementos  
estiver ordenada.*



# Selection sort

**Ideia geral:** fazer com que os elementos menores estejam no início da lista.

**Algoritmo:** percorra o vetor desde o início, selecione o menor elemento e coloque-o na primeira posição. Repita para os próximos  $n-1$  elementos até que o vetor esteja ordenado.

- **Vantagens:** é fácil de ser implementado e não utiliza vetores auxiliares.
- **Desvantagens:** não é estável e sempre faz  $(n^2 - n)/2$  comparações.

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

# Selection sort – algoritmo

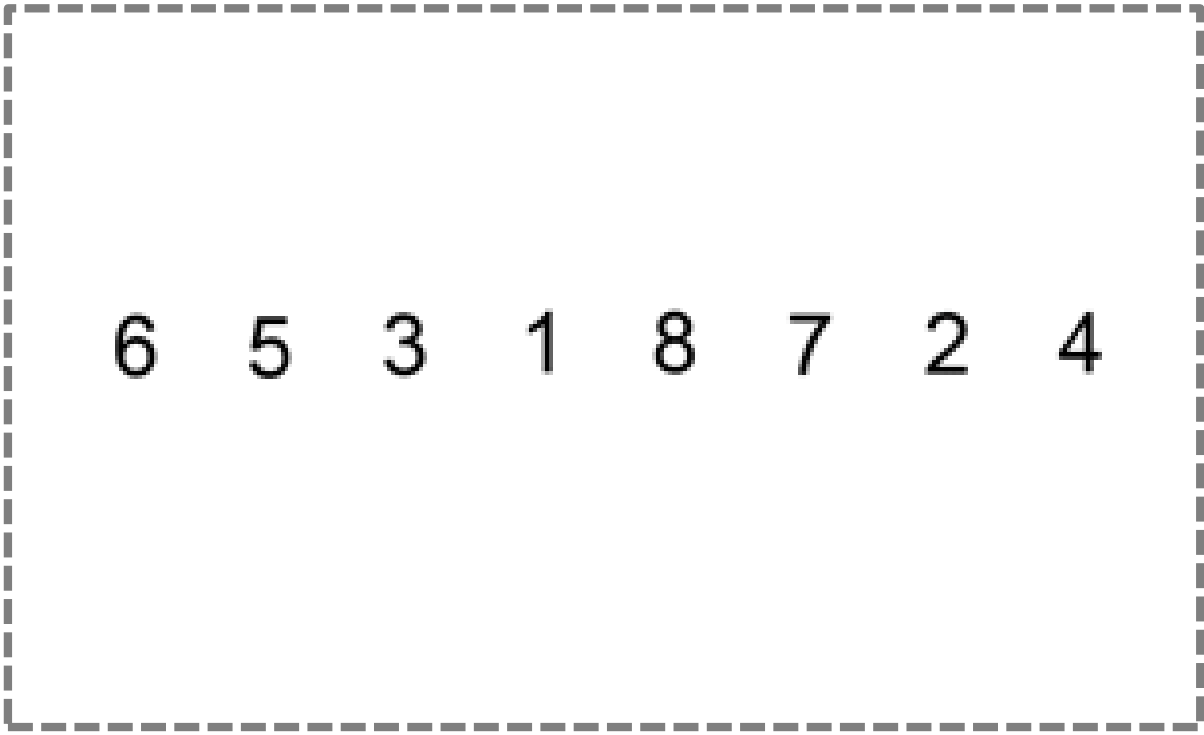
```
public void selectionSort(int[] v, int n) {  
    for (int i = 0; i < n - 1; i++) {  
        int menor = i;  
        for (int j = i + 1; j < n; j++) {  
            if (v[menor] > v[j]) {  
                menor = j;  
            }  
        }  
        if (menor != i) {  
            int temp = v[i];  
            v[i] = v[menor];  
            v[menor] = temp;  
        }  
    }  
}
```

# Insertion sort

**Ideia geral:** fazer com que os elementos sejam inseridos em sua posição correta dentro da sequência final.

**Algoritmo:** percorra o vetor desde o início, da esquerda para a direita, e movimente o elemento à esquerda a fim de deixá-lo ordenado.

- **Vantagens:** fácil implementação.
- **Desvantagens:** como os elementos são trocados frequentemente, há um alto custo.



6 5 3 1 8 7 2 4

# Insertion sort – algoritmo

```
public void insertionSort(int[] v, int n) {  
    int i, j;  
    int aux;  
    for (i = 1; i < n; i++) {  
        aux = v[i];  
        j = i - 1;  
        while ((j >= 0) && (v[j] > aux)) {  
            v[j + 1] = v[j];  
            j--;  
        }  
        v[j + 1] = aux;  
    }  
}
```

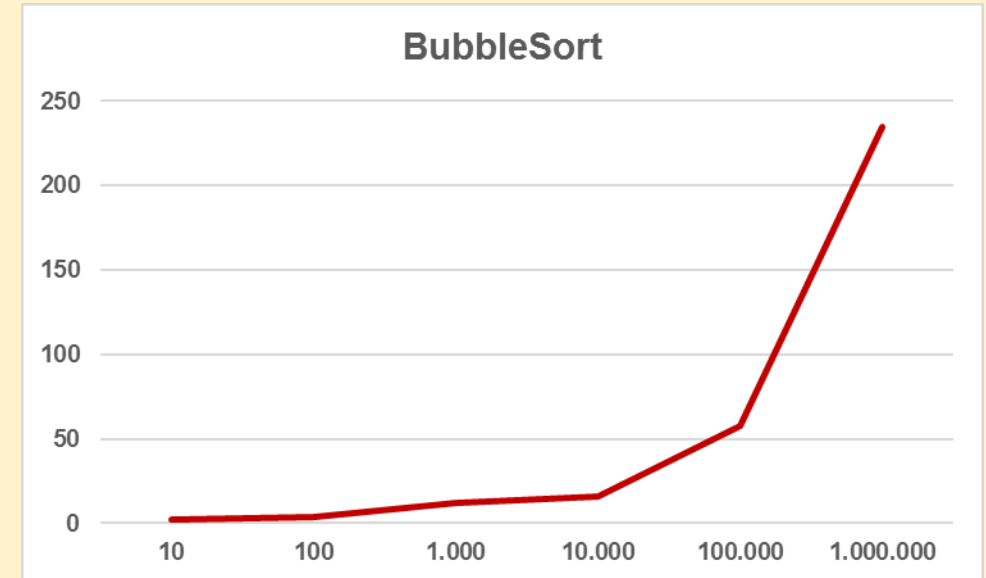
Algoritmo	Melhor Caso	Pior Caso	Complexidade Espacial	Estável?
Bubble sort	$O(n)$	$O(n^2)$	$O(1)$	Sim
Selection sort	$O(n^2)$	$O(n^2)$	$O(1)$	Não
Insertion sort	$O(n)$	$O(n^2)$	$O(1)$	Sim

# Exercício

Implemente as duas versões do algoritmo Bubble sort e compare o tempo de execução (em milissegundos) para os seguintes tamanhos de entrada de vetores: 10, 100, 1.000, 10.000, 100.000 e 1.000.000.

Coloque os resultados em um gráfico.

**Dica:** gere vetores com números aleatórios, mas utilize o mesmo vetor gerado para cada algoritmo.



# Exercício

1. Implemente os algoritmos Selection sort e Insertion sort e compare o tempo de execução (em milissegundos) para os seguintes tamanhos de entrada de vetores: 10, 100, 1.000, 10.000, 100.000 e 1.000.000.
2. Coloque os resultados em um gráfico.
3. Faça a análise de complexidade dos algoritmos Selection sort e Insertion sort.