

**UNIVERSIDADE FEDERAL DO PARÁ  
INSTITUTO DE CIÊNCIAS EXATAS E NATURAIS  
FACULDADE DE COMPUTAÇÃO**

# **Estrutura de Dados I / Projeto de Algoritmos I**

**Prof. Denis Rosário  
Email: [denis@ufpa.br](mailto:denis@ufpa.br)**



# Agenda

## Unidade 5: Algoritmo de Ordenação

1. Bubble sort
2. Insertion sort
3. Selection sort
4. Merge sort
5. Quick sort
6. Heap sort
7. Ordenação em tempo linear: radixsort, bucket sort e counting sort
8. Implementações



# Introdução

- Ordenar: corresponde ao processo de reorganizar um conjunto de objetos em uma ordem ascendente ou descendente
- A ordenação visa facilitar a recuperação posterior de itens do conjunto ordenado.
  - Dificuldade de se utilizar um catálogo telefônico se os nomes das pessoas não estivessem listados em ordem alfabética
- Principais critérios
  - Comparação numérica: um número  $x$  é menor do que um número  $y$  se a expressão  $x-y$  for negativa
  - Comparação alfabética: um caracter é menor do que outro se precede esse outro na ordem do alfabeto. Ex: “a” precede “b” que precede “c”
  - Comparação cronológica: data (AAAA/MM/DD)



# Introdução

- Classificação dos métodos de ordenação:
  - Interna: arquivo a ser ordenado cabe todo na memória principal.
  - Externa: arquivo a ser ordenado não cabe na memória principal.
- Diferenças entre os métodos:
  - Em um método de ordenação interna, qualquer registro pode ser imediatamente acessado.
  - Em um método de ordenação externa, os registros são acessados sequencialmente ou em grandes blocos.
- A maioria dos métodos de ordenação é baseada em comparações
- Existem métodos de ordenação que utilizam o princípio da distribuição.

# Introdução

- Exemplo de ordenação por distribuição: considere o problema de ordenar um baralho com 52 cartas na ordem:

$A < 2 < 3 < \dots < 10 < J < Q < K$  e



- Algoritmo:

1. Distribuir as cartas em treze montes: ases, dois, três, . . . , reis.
2. Colete os montes na ordem especificada.
3. Distribua novamente as cartas em quatro montes: paus, ouros, copas e espadas.
4. Colete os montes na ordem especificada

# Introdução

- Exemplo de ordenação por distribuição: considere o problema de ordenar um baralho com 52 cartas na ordem:

$A < 2 < 3 < \dots < 10 < J < Q < K$  e



- Algoritmo:

1. Distribuir as cartas em treze montes: ases, dois, três, . . . , reis.
2. Colete os montes na ordem especificada.
3. Distribua novamente as cartas em quatro montes: paus, ouros, copas e espadas.
4. Colete os montes na ordem especificada

- Métodos como o ilustrado são também conhecidos como ordenação digital, radixsort ou bucketsort.



# Ordenação Interna

- Algoritmos que ordenam utilizando apenas a memória interna
- Descrição: lista armazenada em um vetor indexado a partir da posição zero
- Obs: a lista pode ser constituída de registro e pode-se ordená-la por um de seus campos (chave de ordenação)
  - A comparação entre os elementos da lista é feita comparando-se os valores das chaves

# Ordenação Interna

- Na escolha de um algoritmo de ordenação interna deve ser considerado o tempo gasto pela ordenação.
- Sendo  $n$  o número registros no arquivo, as medidas de complexidade relevantes são:
  - Número de comparações  $C(n)$  entre chaves.
  - Número de movimentações  $M(n)$  de itens do arquivo.



# Ordenação Interna

- Classificação dos métodos de ordenação interna:
  - Métodos simples:
    - Adequados para pequenos arquivos.
    - Requerem  $O(n^2)$  comparações.
    - Produzem programas pequenos.
  - Métodos eficientes:
    - Adequados para arquivos maiores.
    - Requerem  $O(n \log n)$  comparações.
    - Usam menos comparações.
    - As comparações são mais complexas nos detalhes.
  - Métodos simples são mais eficientes para pequenos arquivos.



# Agenda

## Unidade 5: Algoritmo de Ordenação

1. **Bubble sort**
2. Insertion sort
3. Selection sort
4. Merge sort
5. Quick sort
6. Heap sort
7. Ordenação em tempo linear: radixsort, bucket sort e counting sort
8. Implementações

# Ordenação por método Bolha


## *(Bubble Sort)*

- O *bubble sort*, ou ordenação por flutuação (literalmente "por bolha"), é um algoritmo de ordenação dos mais simples.
- A idéia é percorrer o vetor diversas vezes, a cada passagem fazendo flutuar para o topo o menor elemento da seqüência. Os valores maiores afundam para o fundo (fim do vetor)
- Essa movimentação lembra a forma como as bolhas em um tanque de água procuram seu próprio nível, e disso vem o nome do algoritmo.



# Ordenação por método Bolha

- A técnica consiste em comparar sequencialmente cada um dos elementos do vetor com o seu vizinho. Caso estejam fora de ordem, os mesmos trocam de posição (swap).
- Procede-se assim até o final do vetor. Na primeira “varredura” o último elemento do vetor estará no seu devido lugar (caso a ordenação seja crescente, ele é o maior de todos).



# Ordenação por método Bolha (Bubble Sort)

- A segunda “varredura” é análoga à primeira e vai até o penúltimo elemento.
- O processo é repetido até que tenham sido feitas tantas “varreduras” quanto o número de elementos a serem classificados menos um.
- Ao final o vetor estará classificado segundo o processo escolhido.

# Método da Bolha (*Bubble Sort*)

ALGORITMO BOLHA

ENTRADA: UM VETOR L COM N POSIÇÕES

SAÍDA: O VETOR L EM ORDEM CRESCENTE

PARA  $i = 1$  até  $n - 1$

PARA  $j = 0$  até  $n - 1 - i$

SE  $L[j] > L[j+1]$

AUX = L[j] // SWAP

L[j] = L[j+1]

L[j+1] = AUX

FIM {BOLHA}

# Método da Bolha (*Bubble Sort*)

## ■ Analise de Complexidade

- A instrução crítica do Algoritmo Bolha é a comparação (se)
- Em cada iteração do laço mais interno é feita uma comparação
- O número de comparações realizadas pelo algoritmo é

i	Comparações
0	$n - 1$
1	$n - 2$
...	...
$n - 1$	1
Total	$(n^2 - n) / 2$

# Método da Bolha (*Bubble Sort*)

## ■ Analise de Complexidade

- Neste caso a complexidade temporal do Algoritmo Bolha pertence à  $\Theta(n^2)$
- Além dos parâmetros ( $L$  e  $n$ ), o algoritmo utiliza apenas três variáveis escalares ( $i$ ,  $j$  e  $aux$ )
- Assim a quantidade extra de memória utilizada pelo algoritmo é constante
- Complexidade de espaço  $O(1)$



# Método da Bolha (*Bubble Sort*)

## ■ Estabilidade

- Um método de ordenação é estável se ele preserva a ordem relativa existente entre os elementos repetidos
  - Ex: (4, 7, 6, 7, 2) resulta em (2, 4, 6, 7, 7)
  - O sete sublinhado estava à direita do outro sete e ao final da ordenação ele continuou à direita
- Se todas as ordenações produzidas por um método de ordenação são estáveis, dizemos que o método é estável

# Método da Bolha (*Bubble Sort*)

■ Exemplo: 9 4 5 10 5 8

## ALGORITMO BOLHA

ENTRADA: UM VETOR L COM N POSIÇÕES  
SAÍDA: O VETOR L EM ORDEM CRESCENTE

```
PARA i = 1 até n - 1
  PARA j = 0 até n - 1 - i
    SE L[j] > L[j+1]
      AUX = L[j]          // SWAP
      L[j] = L[j+1]
      L[j+1] = AUX
```

FIM {BOLHA}

## Simulação

Lista Original	0	1	2	3	4	5
	9	4	5	10	<u>5</u>	8

# Método da Bolha (*Bubble Sort*)

■ Exemplo: 9 4 5 10 5 8

## ALGORITMO BOLHA

ENTRADA: UM VETOR L COM N POSIÇÕES  
SAÍDA: O VETOR L EM ORDEM CRESCENTE

```
PARA i = 1 até n - 1
  PARA j = 0 até n - 1 - i
    SE L[j] > L[j+1]
      AUX = L[j]          // SWAP
      L[j] = L[j+1]
      L[j+1] = AUX
```

FIM {BOLHA}

## Simulação

Lista Original	0	1	2	3	4	5
	9	4	5	10	<u>5</u>	8
	4	9	5	10	<u>5</u>	8

# Método da Bolha (*Bubble Sort*)

■ Exemplo: 9 4 5 10 5 8

## ALGORITMO BOLHA

ENTRADA: UM VETOR L COM N POSIÇÕES  
SAÍDA: O VETOR L EM ORDEM CRESCENTE

```
PARA i = 1 até n - 1
  PARA j = 0 até n - 1 - i
    SE L[j] > L[j+1]
      AUX = L[j]          // SWAP
      L[j] = L[j+1]
      L[j+1] = AUX
```

FIM {BOLHA}

## Simulação

Lista Original	0	1	2	3	4	5
	9	4	5	10	<u>5</u>	8
	4	9	5	10	<u>5</u>	8
	4	5	9	10	<u>5</u>	8

# Método da Bolha (*Bubble Sort*)

■ Exemplo: 9 4 5 10 5 8

## ALGORITMO BOLHA

ENTRADA: UM VETOR L COM N POSIÇÕES  
SAÍDA: O VETOR L EM ORDEM CRESCENTE

```
PARA i = 1 até n - 1
  PARA j = 0 até n - 1 - i
    SE L[j] > L[j+1]
      AUX = L[j]          // SWAP
      L[j] = L[j+1]
      L[j+1] = AUX
```

FIM {BOLHA}

## Simulação

Lista Original	0	1	2	3	4	5
	9	4	5	10	<u>5</u>	8
	4	9	5	10	<u>5</u>	8
	4	5	9	10	<u>5</u>	8
	4	5	9	10	<u>5</u>	8

# Método da Bolha (*Bubble Sort*)

■ Exemplo: 9 4 5 10 5 8

## ALGORITMO BOLHA

ENTRADA: UM VETOR L COM N POSIÇÕES  
SAÍDA: O VETOR L EM ORDEM CRESCENTE

```
PARA i = 1 até n - 1
  PARA j = 0 até n - 1 - i
    SE L[j] > L[j+1]
      AUX = L[j]          // SWAP
      L[j] = L[j+1]
      L[j+1] = AUX
```

FIM {BOLHA}

## Simulação

Lista Original

0	1	2	3	4	5
9	4	5	10	<u>5</u>	8
4	9	5	10	<u>5</u>	8
4	5	9	10	<u>5</u>	8
4	5	9	10	<u>5</u>	8
4	5	9	<u>5</u>	10	8

# Método da Bolha (*Bubble Sort*)

■ Exemplo: 9 4 5 10 5 8

## ALGORITMO BOLHA

ENTRADA: UM VETOR L COM N POSIÇÕES  
SAÍDA: O VETOR L EM ORDEM CRESCENTE

```
PARA i = 1 até n - 1
  PARA j = 0 até n - 1 - i
    SE L[j] > L[j+1]
      AUX = L[j]          // SWAP
      L[j] = L[j+1]
      L[j+1] = AUX
```

FIM {BOLHA}

## Simulação

Lista Original

0	1	2	3	4	5
9	4	5	10	<u>5</u>	8
4	9	5	10	<u>5</u>	8
4	5	9	10	<u>5</u>	8
4	5	9	10	<u>5</u>	8
4	5	9	<u>5</u>	10	8
4	5	9	<u>5</u>	8	10

# Método da Bolha (*Bubble Sort*)

■ Exemplo: 9 4 5 10 5 8

## ALGORITMO BOLHA

ENTRADA: UM VETOR L COM N POSIÇÕES  
SAÍDA: O VETOR L EM ORDEM CRESCENTE

```
PARA i = 1 até n - 1
  PARA j = 0 até n - 1 - i
    SE L[j] > L[j+1]
      AUX = L[j]           // SWAP
      L[j] = L[j+1]
      L[j+1] = AUX
```

FIM {BOLHA}

## Simulação

	0	1	2	3	4	5
Lista Original	9	4	5	10	<u>5</u>	8
1ª Iteração	4	5	9	<u>5</u>	8	10



# Método da Bolha (*Bubble Sort*)

■ Exemplo: 9 4 5 10 5 8

## ALGORITMO BOLHA

ENTRADA: UM VETOR L COM N POSIÇÕES  
SAÍDA: O VETOR L EM ORDEM CRESCENTE

```
PARA i = 1 até n - 1
  PARA j = 0 até n - 1 - i
    SE L[j] > L[j+1]
      AUX = L[j]          // SWAP
      L[j] = L[j+1]
      L[j+1] = AUX
```

FIM {BOLHA}

## Simulação

	0	1	2	3	4	5
Lista Original	9	4	5	10	<u>5</u>	8
1ª Iteração	4	5	9	<u>5</u>	8	10
2ª Iteração	4	5	<u>5</u>	8	9	10

# Método da Bolha (*Bubble Sort*)

■ Exemplo: 9 4 5 10 5 8

## ALGORITMO BOLHA

ENTRADA: UM VETOR L COM N POSIÇÕES  
SAÍDA: O VETOR L EM ORDEM CRESCENTE

```
PARA i = 1 até n - 1
  PARA j = 0 até n - 1 - i
    SE L[j] > L[j+1]
      AUX = L[j]          // SWAP
      L[j] = L[j+1]
      L[j+1] = AUX
FIM {BOLHA}
```

## Simulação

	0	1	2	3	4	5
Lista Original	9	4	5	10	<u>5</u>	8
1ª Iteração	4	5	9	<u>5</u>	8	10
2ª Iteração	4	5	<u>5</u>	8	9	10
3ª Iteração	4	5	<u>5</u>	8	9	10
4ª Iteração	4	5	<u>5</u>	8	9	10
5ª Iteração	4	5	<u>5</u>	8	9	10

OBS: No final da segunda iteração a lista estava ordenada  
Então, o que podemos fazer para o método ficar mais “esperto”?

# Método da Bolha (*Bubble Sort*)

- Incluir uma variável (*flag*) para sinalizar se foi realizada alguma troca de elementos numa iteração do método.
- Caso nenhuma troca tenha sido realizada, a lista já estará ordenada e, portanto, o método deve parar

```
ALGORITMO BOLHA_COM_FLAG
  ENTRADA: UM VETOR L COM N POSIÇÕES
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

  i = 0
  FAÇA
    FLAG = FALSO
    PARA j = 0 até n - 1 - i
      SE L[j] > L[j+1]
        TROCA(L[j], L[j+1])
        FLAG = VERDADEIRO
    i = i + 1
  ENQUANTO FLAG = VERDADEIRO
FIM {BOLHA_COM_FLAG}
```

OBS: Enquanto que o tempo requerido pelo Bolha é sempre quadrático em relação ao tamanho da lista, o comportamento do Bolha com flag é sensível ao tipo de lista. No melhor caso (lista ordenada) o algoritmo Bolha\_Com\_Flag tem complexidade linear  $\Theta(n)$

# Método Bolha

## ■ Em C:

```
typedef int ChaveTipo;
```

```
typedef struct
```

```
{
```

```
    ChaveTipo Chave;
```

```
    /* outros componentes */
```

```
} Item;
```

# Método Bolha

```
void Bolha (Item* v, int n )
{
    int i, j;
    Item aux;
    for( i = 0 ; i < n-1 ; i++ )
    {
        for( j = 1 ; j < n-i ; j++ )
            if ( v[j].Chave < v[j-1].Chave )
            {
                aux = v[j];
                v[j] = v[j-1];
                v[j-1] = aux;
            } // if
    }
}
```

# Método Bolha

```
void Bolha (Item* v, int n )
{
    int i, j, troca;
    Item aux;

    for( i = 0 ; i < n-1 ; i++ )
    {
        troca = 0;
        for( j = 1 ; j < n-i ; j++ )
            if ( v[j].Chave < v[j-1].Chave )
            {
                aux = v[j];
                v[j] = v[j-1];
                v[j-1] = aux;
                troca = 1;
            } // if
        if (troca == 0)
            break;
    }
}
```

# Método Bolha

- Para um vetor de  $n$  elementos,  $n - 1$  varreduras são feitas para acertar todos os elementos



# Método Bolha

- Definida pelo número de comparações envolvendo a quantidade de dados do vetor
- Número de comparações:
  - $(n - 1) + (n - 2) + \dots + 2 + 1$
- Complexidade (para qualquer caso):

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



# Exercício

- Faça um teste de mesa com cada método de ordenação estudado até o momento, utilizando as seguintes sequências de dados de entrada:
  - $S1 = \{2, 4, 6, 8, 10, 12\}$
  - $S2 = \{11, 9, 7, 5, 3, 1\}$
  - $S3 = \{5, 7, 2, 8, 1, 6\}$
  - $S4 = \{2, 4, 6, 8, 10, 12, 11, 9, 7, 5, 3, 1\}$
  - $S5 = \{2, 4, 6, 8, 10, 12, 1, 3, 5, 7, 9, 11\}$
  - $S6 = \{8, 9, 7, 9, 3, 2, 3, 8, 4, 6\}$
  - $S7 = \{89, 79, 32, 38, 46, 26, 43, 38, 32, 79\}$
- Em cada caso, mostre o número de comparações e trocas que realizam na ordenação de sequências.



# Agenda

## Unidade 5: Algoritmo de Ordenação

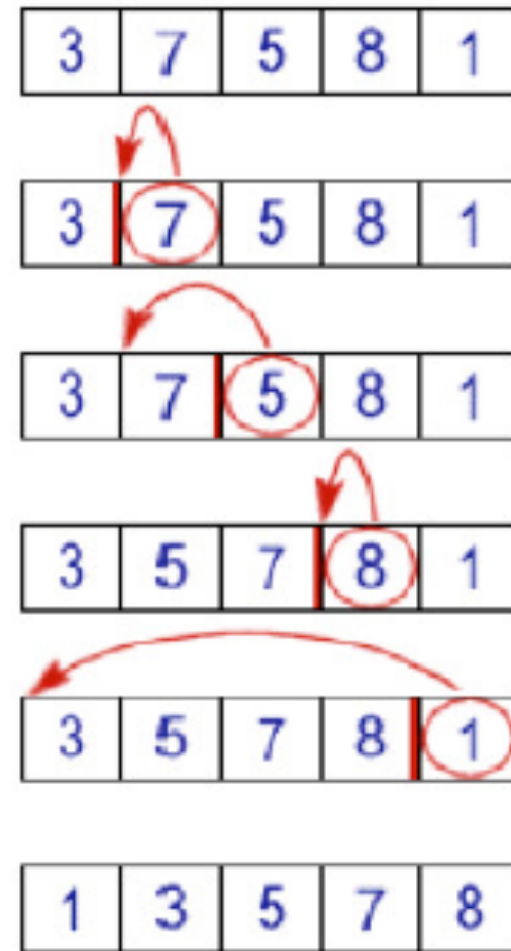
1. Bubble sort
2. **Insertion sort**
3. Selection sort
4. Merge sort
5. Quick sort
6. Heap sort
7. Ordenação em tempo linear: radixsort, bucket sort e counting sort
8. Implementações

# Ordenação por Inserção (insertion sort)

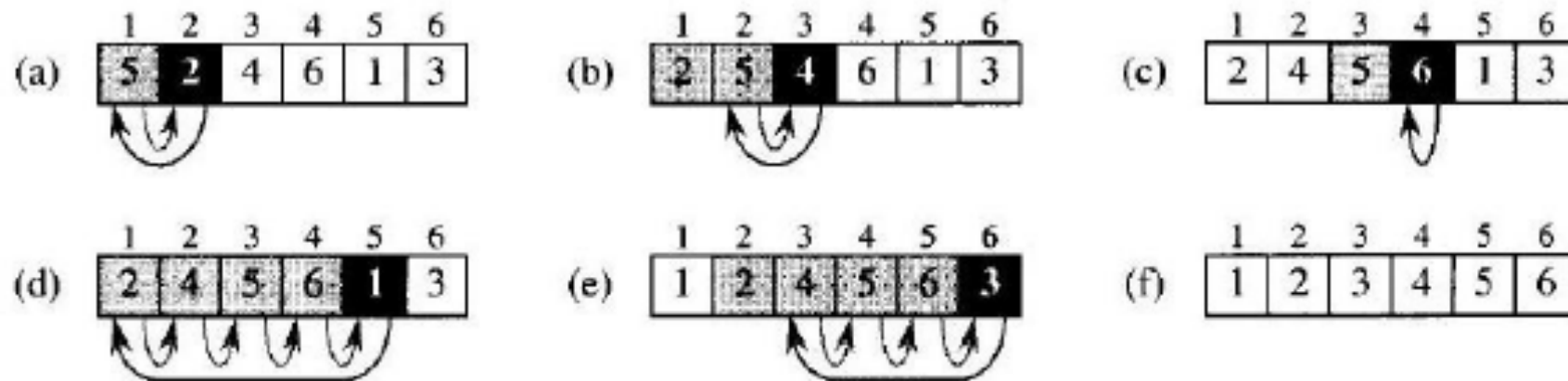
- é um simples algoritmo de ordenação, eficiente quando aplicado a um pequeno número de elementos.
- Em termos gerais, ele percorre um vetor de elementos da esquerda para a direita e à medida que avança vai deixando os elementos mais à esquerda ordenados.
- O algoritmo de inserção funciona da mesma maneira com que muitas pessoas ordenam cartas em um jogo de baralho como o pôquer.
  - Sempre temos dois grupos de elementos: os já ordenados no início do vetor e os não ordenados no final.

# Ordenação por inserção

- Insere cada elemento do vetor, na sua posição correta em uma subsequência ordenada de elementos, de modo a mantê-la ordenada.
- Esta forma de ordenação seleciona cada um dos elementos de uma seqüência e os atribui uma posição relativa a seu valor, de acordo com a comparação com os elementos já ordenados da mesma seqüência.



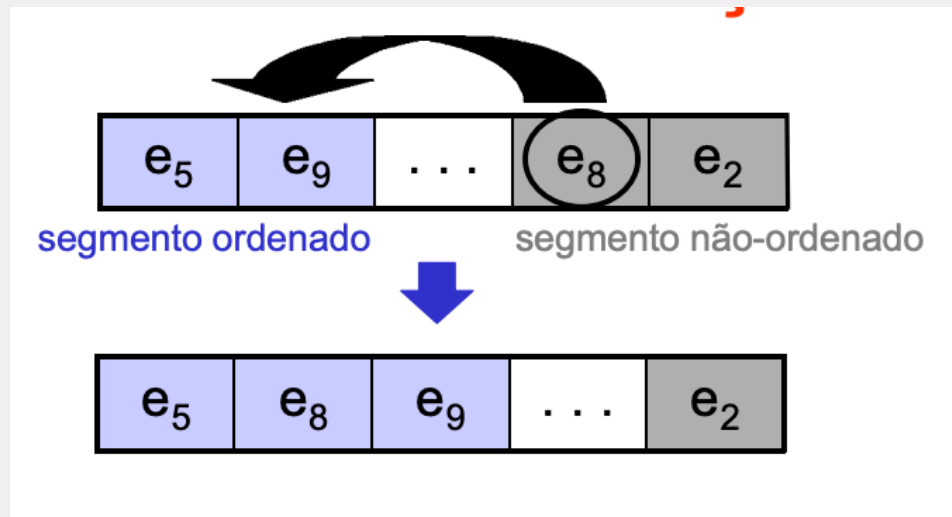
# Ordenação por inserção



Realizamos  $N - 1$  rodadas. Em cada rodada, um elemento não ordenado é deslocado para a esquerda (através de trocas com seus vizinhos) até que chegue na sua posição correta dentro do vetor ordenado.

# Ordenação por inserção

- Lista dividida:
  - Parte esquerda: já ordenada
  - Parte direita: em possível desordem



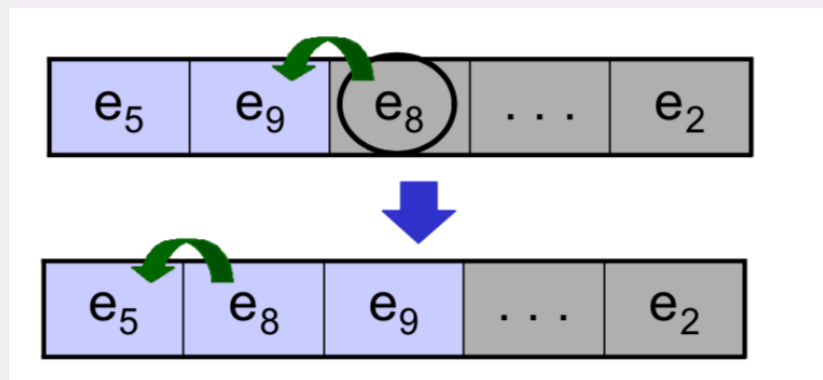


# Ordenação por inserção

- Inicialmente a parte esquerda contém apenas o primeiro elemento da lista
- Cada iteração consiste em inserir o primeiro elemento da parte direita (pivô) na posição adequada da parte esquerda, de modo que a parte esquerda continue ordenada
- É fácil perceber que se a lista possui  $n$  elementos, após  $n-1$  inserções ela estará ordenada
- Para inserir o pivô percorremos a parte esquerda, da direita para esquerda, deslocando os elementos maiores que o pivô uma posição para a direita

# Ordenação por inserção

- realiza uma busca sequencial no segmento ordenado para inserir corretamente um elemento do segmento não-ordenado
- nesta busca, realiza trocas entre elementos adjacentes para ir acertando a posição do elemento a ser inserido





# Ordenação por inserção

ALGORITMO INSERÇÃO

*ENTRADA: UM VETOR L COM N POSIÇÕES*

*SAÍDA: O VETOR L EM ORDEM CRESCENTE*

PARA  $i = 1$  até  $n - 1$

    PIVO =  $L[i]$

$j = i - 1$

    ENQUANTO  $j \geq 0$  e  $L[i] > \text{PIVO}$

$L[j+1] = L[j]$

$j = j - 1$

$L[j+1] = \text{PIVO}$

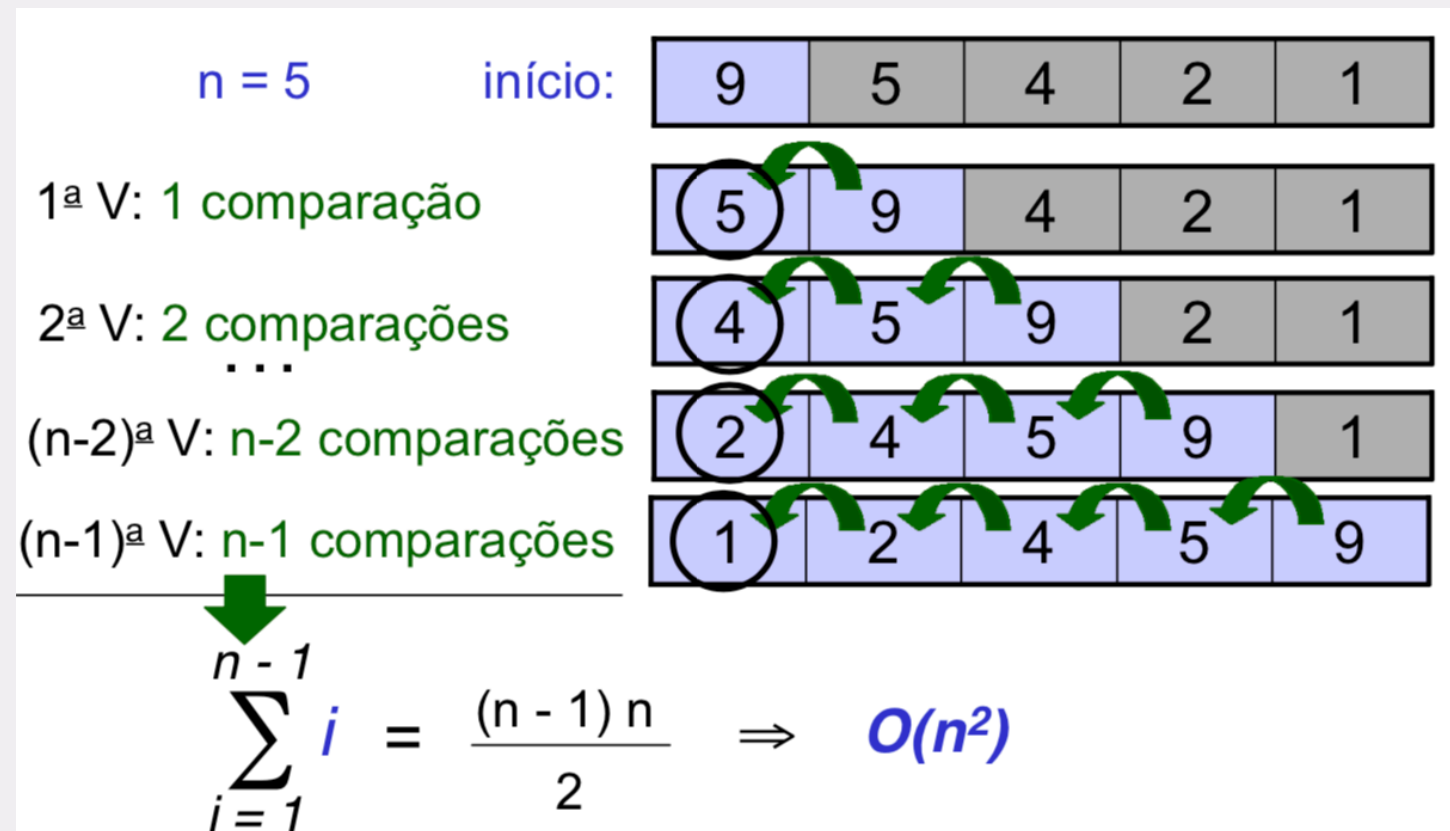
FIM {INSERÇÃO}

# Ordenação por inserção

- No melhor caso (lista ordenada), cada inserção é feita em tempo constante, pois o pivô é maior ou igual a todos os elementos da parte esquerda e a condição do laço interno nunca é verdadeira
  - Complexidade temporal é  $\Theta(n)$
- No pior caso (lista em total desordem), cada inserção requer que todos os elementos da parte esquerda sejam movidos para a direita
  - Nesse caso, a quantidade total de iterações do laço interno será  $(n^2-n)/2$
  - Complexidade temporal é  $\Theta(n^2)$
  - No caso médio também é  $\Theta(n^2)$

# Ordenação por inserção

- Pior caso: vetor totalmente desordenado



# Ordenação por inserção

## ■ Melhor caso: vetor já ordenado



# Ordenação por inserção

- O Algoritmo de inserção requer o uso de apenas três variáveis escalares.
- Assim sua complexidade espacial é  $O(1)$
- O algoritmo de inserção é **estável**: um elemento da parte esquerda somente é movido para a direita se ele for estritamente maior do que o pivô

# Ordenação por inserção

- Exercício: 9, 4, 5, 10, 5, 8

	0	1	2	3	4	5
Lista Original	9	4	5	10	<u>5</u>	8
1ª Iteração	4	9	5	10	<u>5</u>	8
2ª Iteração	4	5	9	10	<u>5</u>	8
3ª Iteração	4	5	9	10	<u>5</u>	8
4ª Iteração	4	5	<u>5</u>	9	10	8
5ª Iteração	4	5	<u>5</u>	8	9	10



# Método Inserção

```
void Insercao (Item* v, int n )
{
    int i,j;
    Item aux;
    for (i = 1; i < n; i++)
    {
        aux = v[i];
        j = i - 1;
        while ( ( j >= 0 ) && ( aux.Chave < v[j].Chave ) )
        {
            v[j + 1] = v[j];
            j--;
        }
        v[j + 1] = aux;
    }
}
```

# InsertionSort X BubbleSort

	Melhor caso	Pior caso
<i>InsertionSort</i>	$O(n)$	$O(n^2)$
<i>BubbleSort</i>	$O(n^2)$	$O(n^2)$





# Agenda

## Unidade 5: Algoritmo de Ordenação

1. Bubble sort
2. Insertion sort
3. **Selection sort**
4. Merge sort
5. Quick sort
6. Heap sort
7. Ordenação em tempo linear: radixsort, bucket sort e counting sort
8. Implementações

# Ordenação por Seleção (selection sort)

- Tem como ponto forte o fato de que ele realiza poucas operações de *swap*
- Seu desempenho, em termos absolutos, costuma ser superior ao do método Bolha, mas inferior ao do Método Inserção
- A lista está dividida em duas partes
  - Esquerda: ordenada
  - Direita: em possível desordem
  - Os elementos da parte esquerda são todos menores ou iguais aos elementos da parte direita
- Cada iteração consiste em selecionar o menor elemento da parte direita (pivô) e trocá-lo com o primeiro elemento da direita

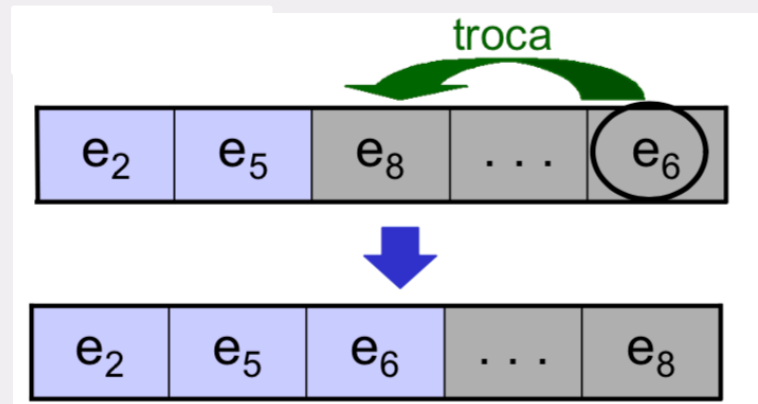


# Ordenação por seleção

- A idéia básica do Método de Seleção é, a cada passagem pelo vetor, selecionar o menor elemento e colocar este elemento o mais a esquerda possível. Para isto deve-se trocar as posições dos elementos do vetor.

# Ordenação por seleção

- Selection Sort é um método simples de seleção
  - ordena através de sucessivas seleções do elemento de menor valor em um segmento não-ordenado e seu posicionamento no final de um segmento ordenado





# Ordenação por seleção

- Na primeira passagem troca-se o menor elemento com o que está na primeira posição;
- Na segunda passagem troca-se o segundo menor elemento com o que está na segunda posição . Assim por diante...

# Ordenação por Seleção

ALGORITMO SELEÇÃO

*ENTRADA: UM VETOR L COM N POSIÇÕES*

*SAÍDA: O VETOR L EM ORDEM CRESCENTE*

PARA  $i = 0$  ate  $n - 2$

MIN =  $i$

PARA  $j = i + 1$  até  $n - 1$

SE  $L[j] < L[MIN]$

MIN =  $j$

TROCA( $L[i]$ ,  $L[MIN]$ )

FIM {SELEÇÃO}

# Ordenação por Seleção

- Análise semelhante à análise do Algoritmo Bolha
- Os dois algoritmos são constituídos de dois laços contados encaixados que realizam as mesmas quantidades de iterações
- Logo a complexidade temporal do algoritmo Seleção é  $\Theta(n^2)$
- O algoritmo Seleção precisa apenas de 4 variáveis escalares. Assim sua complexidade de espaço é constante

# Ordenação por seleção

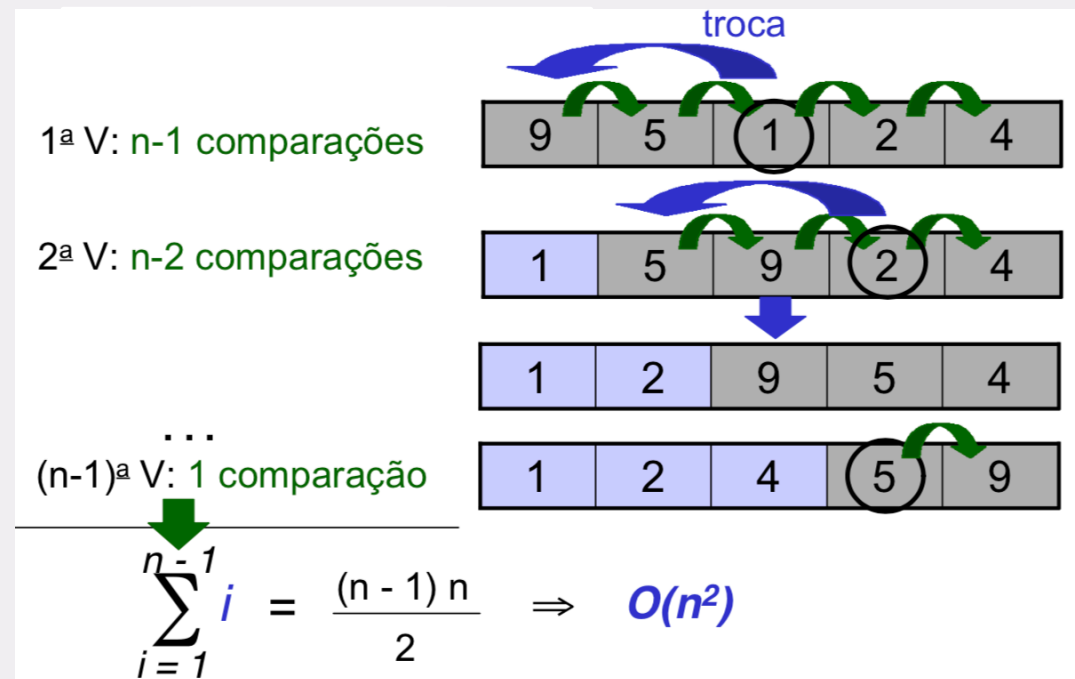
■ Exemplo: 10, 4, 5, 3, 10, 2

	0	1	2	3	4	5
Lista original	10	4	5	3	<u>10</u>	2
1ª iteração	2	4	5	3	<u>10</u>	10
2ª iteração	2	3	5	4	<u>10</u>	10
3ª iteração	2	3	4	5	<u>10</u>	10
4ª iteração	2	3	4	5	<u>10</u>	10
5ª iteração	2	3	4	5	<u>10</u>	10




# Ordenação por seleção

## ■ Para qualquer caso



# Comparação

	Melhor caso	Pior caso
<i>InsertionSort</i>	$O(n)$	$O(n^2)$
<i>BubbleSort</i>	$O(n^2)$	$O(n^2)$
<i>SelectionSort</i>	$O(n^2)$	$O(n^2)$



# Inserção × Seleção

- Arquivos já ordenados:
  - Inserção: algoritmo descobre imediatamente que cada item já está no seu lugar (custo linear).
  - Seleção: ordem no arquivo não ajuda (custo quadrático).
- Adicionar alguns itens a um arquivo já ordenado:
  - Método da inserção é o método a ser usado em arquivos “quase ordenados”.

# Inserção × Seleção

## ■ Comparações:

- Inserção tem um número médio de comparações que é aproximadamente a metade da Seleção

## ■ Movimentações:

- Seleção tem um número médio de comparações que cresce linearmente com  $n$ , enquanto que a média de movimentações na Inserção cresce com o quadrado de  $n$ .



# **Simulação de funcionamento dos algoritmos de ordenação**

- [www.cs.usfca.edu/~galles/visualization/ComparisonSort.html](http://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html)



# Links interessante

<http://www.cs.oswego.edu/~mohammad/classes/csc241/samples/sort/Sort2-E.html>

<http://pt.wikipedia.org/wiki/Ficheiro:Selection-Sort-Animation.gif>

# Exercício

- Faça um teste de mesa com cada método de ordenação estudado até o momento, utilizando as seguintes sequências de dados de entrada:
  - $S1 = \{2, 4, 6, 8, 10, 12\}$
  - $S2 = \{11, 9, 7, 5, 3, 1\}$
  - $S3 = \{5, 7, 2, 8, 1, 6\}$
  - $S4 = \{2, 4, 6, 8, 10, 12, 11, 9, 7, 5, 3, 1\}$
  - $S5 = \{2, 4, 6, 8, 10, 12, 1, 3, 5, 7, 9, 11\}$
  - $S6 = \{8, 9, 7, 9, 3, 2, 3, 8, 4, 6\}$
  - $S7 = \{89, 79, 32, 38, 46, 26, 43, 38, 32, 79\}$
- Em cada caso, mostre o número de comparações e trocas que realizam na ordenação de sequências.