



# Estruturas de Dados 2

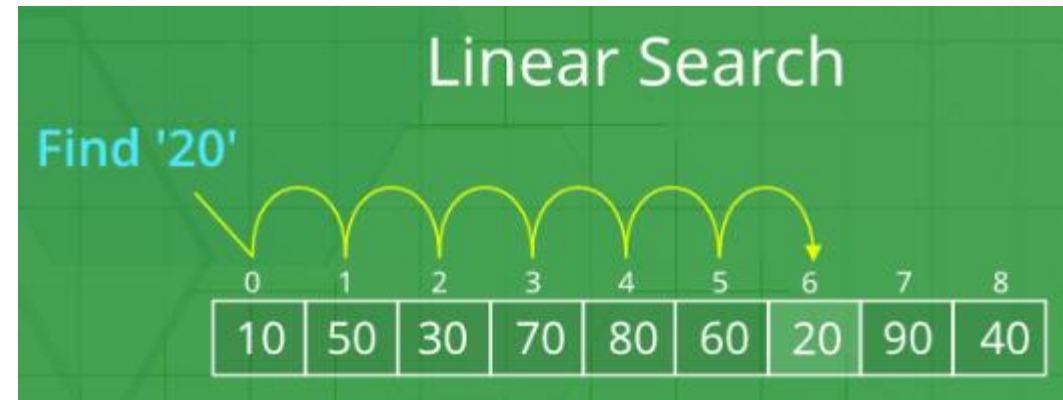
## 01 – Buscas e ordenação em vetores

Antonio Angelo de Souza Tartaglia  
angelot@ifsp.edu.br

# Estrutura de Dados 2

## Busca e Ordenação

- No mundo da computação, talvez nenhuma outra tarefa seja mais **fundamental** ou tão **exaustivamente** analisada como estas de “**Busca e Ordenação**”.
- Estas rotinas são utilizadas em praticamente todos os programas de Banco de Dados, bem como compiladores, interpretadores e sistemas operacionais



# Estrutura de Dados 2

## Buscas em Vetores



- Busca – Definição:

- Recuperação de dados armazenados em um repositório ou “Base de Dados”;

Vetores, Lista  
Ligada, Árvore, etc.

- O tipo de Busca depende do tipo de dados armazenados:

- Dados estão estruturados (Vetor, Lista ou Árvore);
- Dados ordenados, ou não ordenados;
- Valores duplicados.

O tipo de Busca a ser  
realizada sempre depende  
do tipo de dado.

Uma Busca pode encontrar itens  
duplicados, então o que fazer?  
Recuperar o 1º, 2º, ambos?

# Estrutura de Dados 2

## Busca

- Métodos de Busca:
  - Encontrar informações em um vetor desordenado requer uma **Busca Linear**, começando pelo primeiro elemento, e terminando quando o elemento é encontrado, ou quando o fim do vetor é alcançado sem que o elemento procurado seja encontrado, ou seja, ele não existe.
  - Este método deve ser usado em dados desordenados, mas também pode ser aplicado para dados ordenados. Porém, ainda mantém a baixa eficiência, mesmo quando os dados estão ordenados
  - Mas, se os dados estiverem ordenados, então os tipos de buscas “Ordenada” ou “Binária” (que possui um desempenho superior à Busca Ordenada), podem ser utilizadas, já que apresentam uma melhor performance em relação a Busca Linear.



# Estrutura de Dados 2

## Busca em vetores

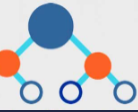
- Tipos de busca abordados:
  - Dados armazenados em um Vetor;
  - Dados Ordenados ou não.

- Métodos:

- Busca Linear;
- Busca Ordenada;
- Busca Binária.

Para este tipo de busca, os dados não necessitam de ordenação.

Funcionam somente para dados previamente ordenados.





## Busca em vetores

- Busca Linear: é a mais fácil de ser codificada. A função a seguir faz uma busca em um vetor de inteiros de comprimento conhecido “n”, até encontrar o elemento procurado “elem”:

```
int buscaLinear(int *vetor, int n, int elem){  
    int i;  
    for(i = 0; i < n; i++){  
        if(elem == vetor[i]){  
            return i;  
        }  
    }  
    return -1;  
}
```

- Esta função retorna o **índice** do elemento no vetor, caso ele seja encontrado, ou **-1** se o elemento não existir no vetor.
- É fácil notar que uma busca linear testará em média  $\frac{1}{2}n$  elementos. No melhor caso testará somente 1 elemento e no pior caso “n” elementos.

# Estrutura de Dados 2

## Busca em vetores

- Busca Linear:

```
int buscaLinear(int *vetor, int n, int elem){  
    int i;  
    for(i = 0; i < n; i++){  
        if(elem == vetor[i]){  
            return i;  
        }  
    }  
    return -1;  
}
```

Vetor: 

120	150	110	130	100	160	140	190	170	180
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

elem: 

100
-----

 Elemento à ser procurado

i = 0	120	150	110	130	100	160	140	190	170	180	Diferente: continua a busca...
i = 1	120	150	110	130	100	160	140	190	170	180	Diferente: continua a busca...
i = 2	120	150	110	130	100	160	140	190	170	180	Diferente: continua a busca...
i = 3	120	150	110	130	100	160	140	190	170	180	Diferente: continua a busca...
i = 4	120	150	110	130	100	160	140	190	170	180	Igual: Busca terminada.

Retorna índice 4





## Busca em vetores

- Busca Ordenada: igualmente fácil de ser implementada como a Busca Linear, apenas devemos acrescentar, **a cada iteração onde o elemento não foi encontrado**, uma verificação com a finalidade de detectar se o elemento verificado no vetor é maior do que o elemento que se procura “**elem**” naquele momento.

```
int buscaOrdenada(int *vetor, int n, int elem){
    int i;
    for(i = 0; i < n; i++){
        if(elem == vetor[i]){
            return i;
        }else{
            if(elem < vetor[i]){
                return -1;
            }
        }
    }
    return -1;
}
```

- A partir do momento que **o elemento no vetor for maior que o elemento procurado**, a Busca é encerrada, pois o elemento não existe no vetor (já que está ordenado). Nesse caso a função retorna **-1**. O último return é atingido quando o elemento buscado é maior do que todos que estão no vetor, e também não foi encontrado.



# Estrutura de Dados 2

## Busca em vetores

- Busca Ordenada:

```
int buscaOrdenada(int *vetor, int n, int elem){
    int i;
    for(i = 0; i < n; i++){
        if(elem == vetor[i]){
            return i;
        }else{
            if(elem < vetor[i]){
                return -1;
            }
        }
    }
    return -1;
}
```

A tarefa de busca é otimizada se o vetor estiver ordenado. Ainda assim não é muito eficiente, pois é necessário percorrer todo o vetor em alguns casos.

Busca é encerrada.  
Deste elemento para frente, só existem elementos maiores.

Vetor: 

100	110	120	130	140	150	160	170	180	190
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

elem: 

125
-----

 Elemento à ser procurado

i = 0	100	110	120	130	140	150	160	170	180	190	Diferente: continua a busca...
i = 1	100	110	120	130	140	150	160	170	180	190	Diferente: continua a busca...
i = 2	100	110	120	130	140	150	160	170	180	190	Diferente: continua a busca...
i = 3	100	110	120	130	140	150	160	170	180	190	Valor é maior: elem não existe.
i = 4	100	110	120	130	140	150	160	170	180	190	

Retorna -1, não existe o elemento procurado.



# Estrutura de Dados 2

## Busca em vetores

- Busca Binária: Este método é superior aos dois anteriores. Utiliza o conceito de “Divisão e Conferência”.
  - A estratégia é baseada na ideia de **dividir para conquistar**. A cada passo, esse algoritmo analisa o valor do meio do vetor. Caso o valor seja igual ao elemento procurado, a busca é encerrada. Do contrário, baseado na comparação anterior (elemento procurado é maior ou menor), a busca continua na metade do vetor em que o elemento pode se encontrar .
  - Este procedimento é repetido até que o elemento procurado seja encontrado, ou até que não haja mais elementos a testar, caso em que retornará **-1**.





## Busca em vetores

- Busca Binária:

```
int buscaBinaria(int *vetor, int n, int elem){
    int i, inicio, meio, fim;
    inicio = 0;
    fim = n - 1;

    while(inicio <= fim){
        meio = (inicio + fim)/2;
        if(elem < vetor[meio]){
            fim = meio - 1; //busca na metade esquerda
        }else{
            if(elem > vetor[meio]){
                inicio = meio + 1; // busca na metade direita
            }else{
                return meio;
            }
        }
    }
    return -1; // elemento não encontrado
}
```

- Neste tipo de Busca, o número de comparações no pior caso, é  $\log_2 n$ .

# Estrutura de Dados 2

## Busca em vetores

- Busca Binária:

```
while(inicio <= fim){
    meio = (inicio + fim)/2;
    if(elem < vetor[meio]){
        fim = meio - 1; //busca na metade esquerda
    }else{
        if(elem > vetor[meio]){
            inicio = meio + 1; // busca na metade direita
        }else{
            return meio;
        }
    }
}
```

Vetor: 

100	110	120	130	140	150	160	170	180	190
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

elem: 

130
-----

 Elemento à ser procurado

fim - 1

i = 0	100	110	120	130	<del>140</del>	<del>150</del>	<del>160</del>	<del>170</del>	<del>180</del>	<del>190</del>	Elemento é menor: Busca no início	<table border="1"><tr><td>elem &lt; 140</td></tr></table>	elem < 140
elem < 140													
i = 1	<del>100</del>	<del>110</del>	120	130	Elemento é maior: Busca no final...								
i = 2			<del>120</del>	130	Elemento é maior: Busca no final...								
i = 3				130	Elemento é igual: Busca encerrada.								

elem > 110

elem > 120

Retorna índice 3

elem = 130



# Estrutura de Dados 2

## Busca em vetores

- É importante lembrar que toda busca é feita utilizando como base uma chave específica. Esta chave é o “**campo**” utilizado para comparação.
- No caso de uma estrutura, para a chave é utilizado um campo da **struct** para comparação, normalmente um identificador único (ID ou código).



# Estrutura de Dados 2

## Busca em Vetor de Estruturas – struct

```
struct funcionario{  
    int codigo;  
    char nome[30];  
    float salario;  
};  
  
struct funcionario func[6];  
  
int buscaLinearCodigo(struct funcionario *vetor, int n, int cod){  
    int i;  
    for(i = 0; i < n; i++){  
        if(cod == vetor[i].codigo){  
            return i; //elemento encontrado  
        }  
    }  
    return -1; //elemento não encontrado  
}
```

Na prática trabalhamos com dados mais complexos, estruturas com mais informações

Codigo; Nome[30]; Salario;	Codigo; Nome[30]; Salario;	Codigo; Nome[30]; Salario;	Codigo; Nome[30]; Salario;	Codigo; Nome[30]; Salario;	Codigo; Nome[30]; Salario;
Func[0]	Func[1]	Func[2]	Func[3]	Func[4]	Func[5]

- Em um vetor de estruturas não basta mais comparar a posição do vetor, é necessário comparar alguns campos da estrutura que está em cada posição dentro do vetor.



# Estrutura de Dados 2

## Busca em Vetor de Estruturas – struct

- Busca por Código:

```
int buscaLinearCodigo(int *vetor, int n, int cod){
    int i;
    for(i = 0; i < n; i++){
        if(cod == vetor[i].codigo){
            return i;//elemento encontrado
        }
    }
    return -1;//elemento não encontrado
}
```

*Diagrama: Uma seta tracejada vermelha aponta do campo `codigo` na estrutura `funcionario` para o acesso `vetor[i].codigo` no código.*

```
struct funcionario{
    int codigo;
    char nome[30];
    float salario;
};
```

- Busca por outro campo qualquer (*string*):

```
int buscaLinearNome(int *vetor, int n, char *nome){
    int i;
    for(i = 0; i < n; i++){
        //strstr() retorna NULL se string1 não contém string2
        if(strstr(vetor[i].nome, nome){
            return i;//elemento encontrado
        }
    }
    return -1;//elemento não encontrado
}
```



# Estrutura de Dados 2

## Atividade 1

- Elabore um programa que execute os três tipos de busca:
  - Busca Linear;
  - Busca Ordenada;
  - Busca Binária.
- Para isso, seu programa deverá contar com dois vetores de inteiros, um desordenado para a Busca Linear, e outro ordenado para as Buscas Ordenada e Binária.
- Os resultados devem ser apresentados em tela juntamente com o respectivo vetor para a conferência do **índice** retornado.
- Entregue no Moodle como Atividade 1 – Buscas em Vetores

```
"C:\Users\angelot\Documents\Aulas\EDA2\Aulas\01\material apoio\buscaVetor.exe"
Vetor: 120 150 110 130 100 160 140 190 180 170
Elemento a ser procurado por busca linear: 100
A posicao do elemento no vetor e: 4

Vetor Ordenado: 100 110 120 130 140 150 160 170 180 190
Elemento a ser procurado por busca ordenada: 170
A posicao do elemento no vetor e: 7

Elemento a ser procurado por busca binaria: 130

inicio: 0    fim: 9
inicio: 0    fim: 3
inicio: 2    fim: 3
inicio: 3    fim: 3

A posicao do elemento no vetor e: 3
```

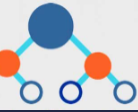




# Estrutura de Dados 2

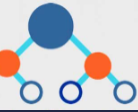
## Atividade 2

- Utilizando o programa da atividade 1, escreva uma função para a inserção de um novo valor no vetor ordenado em sua posição correta. Desloque os outros números, se necessário. Disponibilize espaço vago no vetor para a movimentação dos elementos.
- Entregue na plataforma Moodle, como atividade 2.



## Ordenação

- Existem muitos algoritmos de ordenação. Cada um deles têm seus méritos, mas, de uma forma geral, a avaliação de um algoritmo de ordenação está baseada nas respostas as seguintes perguntas:
  - Em que velocidade ele pode ordenar as informações no caso médio?
  - Qual a velocidade de seu melhor e pior casos?
  - Este algoritmo apresenta comportamento **natural** ou **não-natural**?
  - Ele rearranja elementos iguais?



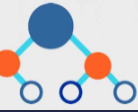
## Ordenação

- Com que velocidade um algoritmo ordena para ter um ótimo desempenho? A velocidade na qual um algoritmo ordena um vetor é diretamente relacionada ao número de comparações e ao número de trocas exigidas, **com trocas exigindo mais tempo**. Alguns algoritmos variam o tempo de ordenação de um elemento de forma **exponencial**, e outros de forma **logarítmica**.
- Os tempos de processamento no **pior** e no **melhor** caso são importantes se você desejar saber regularmente quais situações de melhor e pior caso. Frequentemente uma ordenação terá um bom caso médio mas um terrível pior caso, ou vice e versa.



## Ordenação

- Dizemos que um algoritmo de ordenação tem um comportamento ***natural*** se ele **trabalha o menos possível** quando a lista já está ordenada, e quanto mais desordenada estiver a lista, mais trabalho terá o algoritmo, e trabalhará o maior tempo ainda, quando a lista estiver em **ordem inversa**.
- O maior trabalho de um algoritmo de ordenação é o número de comparações e movimentos (trocas), que ele deve executar.



## Ordenação

- Definição:

- Processo de organizar um conjunto de informações semelhantes, em uma ordem, *crescente* ou *decrescente*. Especificamente, dada uma lista ordenada  $i$  de  $n$  elementos, então:

$$i_1 \leq i_2 \leq \dots \leq i_n$$

- A ordenação permite que o acesso aos dados seja feita de forma mais eficiente.

- Algoritmo de Ordenação:

- É o algoritmo que organiza os elementos de uma determinada sequência, em uma certa ordem.

- Exemplo:

- 130, 150, 120, 100, 110, 140 – Fora de ordem;
- 100, 110, 120, 130, 140, 150 – Ordenado.

A operação de busca se torna muito mais rápida e eficiente se os dados estiverem ordenados.



# Estrutura de Dados 2

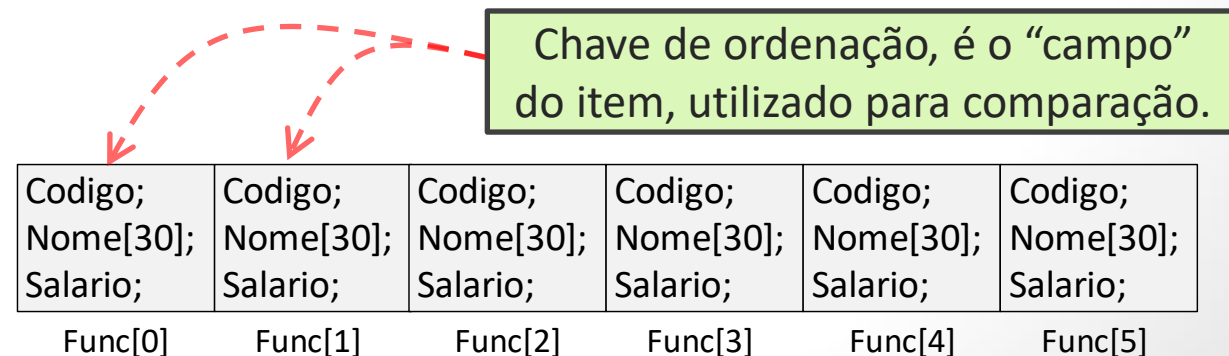
## Ordenação

- A **ordenação** de um conjunto de dados é feita utilizando como base uma chave específica.

```
struct funcionario{  
    int codigo; <-----  
    char nome[30];  
    float salario;  
};
```

- Ela é utilizada nas comparações, mas quando uma troca se torna necessária, **toda a estrutura de dados** é transferida.
- Para realizar a **ordenação**, podemos utilizar qualquer tipo de **chave**, desde que exista uma regra de ordenação bem definida.
- Tipos de ordenação mais comuns:

- Numérica:
  - 1, 2, 3, 4, 5...
- Lexicográfica (ordem alfabética):
  - Ana, Antônio, Claudio, Marcelo...



## Ordenação

- Independente do tipo, a Ordenação pode ser:
  - Crescente:
    - 1, 2, 3, 4, 5...
    - Ana, Antônio, Claudio, Marcelo...
  - Decrescente:
    - ... 5, 4, 3, 2, 1.
    - ... Marcelo, Claudio, Antônio, Ana.

Algoritmo de ordenação é aquele que rearranja os elementos de uma determinada sequencia fornecida, em uma ordem predefinida.



## Ordenação

- Classificação dos algoritmos de Ordenação:

- Ordenação Interna:

- O bloco a ser ordenado está na sua totalidade na **memória** do computador;
    - Qualquer registro pode ser imediatamente acessado.

Todo o conjunto de dados a ser ordenado cabe na memória disponível.

Vantagem!  
Todos os registros estão na memória.

- Ordenação Externa:

- O bloco a ser ordenado não cabe na memória principal disponível, e neste caso, trata-se de um arquivo;
    - Os registros são acessados sequencialmente ou em grandes blocos.

Um algoritmo de ordenação é considerado **estável**, se a ordem dos elementos com chaves iguais não muda durante a ordenação.





## Métodos de Ordenação

- Existem três métodos básicos gerais (que possuem baixo rendimento), para a ordenação de matrizes:
  - Por troca;
  - Por seleção;
  - Por inserção.
- Para entender melhor, imagine as cartas de um baralho. Para ordenar as cartas, utilizando **troca**, espalhe-as em uma mesa, então troque sequencialmente as cartas fora de ordem. Repita a operação até que todo o baralho esteja ordenado.
- Utilizando a **seleção**, espalhe as cartas na mesa e selecione a carta de menor valor, retire-a do baralho e segure-a na mão. Esse processo continua até que todas as cartas estejam em sua mão. As cartas na sua mão estarão ordenadas quando o processo estiver terminado.
- Para ordenar por **inserção**, segure todas as cartas em sua mão. Ponha uma carta por vez na mesa, sempre inserindo-a na posição correta. O baralho estará ordenado quando não restarem mais cartas em sua mão.



## Métodos de Ordenação

- Métodos de Ordenação

- Básicos:

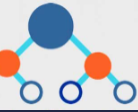
Complexidade:  $O(n^2)$

- Fácil entendimento;
    - Auxiliam o entendimento dos algoritmos complexos.

- Sofisticados

Complexidade:  $O(n \log n)$

- Em geral são mais eficientes e possuem melhor desempenho.



## Métodos de Ordenação

- $O(n^2)$  - Em um conjunto de “ $n$ ” elementos a ser ordenado, um algoritmo com este nível de complexidade, gasta “ $n^2$ ” operações para sua ordenação. Exemplo:
  - Em um vetor com **100** elementos, o algoritmo gasta  **$100^2$**  operações para sua ordenação, ou seja **10.000** operações.
- $O(n \log n)$  – Em um conjunto de “ $n$ ” elementos a ser ordenado, um algoritmo mais sofisticado que utiliza este nível de complexidade, gasta “ $n \log n$ ” operações para a ordenação destes dados. Exemplo:
  - Em um vetor com 100 elementos, o algoritmo gasta  **$100 \log 100$**  operações para sua ordenação, ou seja  **$\log 100 = 2$** , logo  **$100 * 2 = 200$** . O algoritmo gasta cerca de **200** operações para a ordenação do mesmo vetor.

**$O(n \log n)$  é muito mais rápido do que  $O(n^2)$**





## Ordenação BubbleSort ou Ordenação por Bolha

- É uma ordenação por **trocas**. Compara pares de elementos adjacentes (2 elementos sequenciais dentro de um vetor), e os troca de lugar se estiverem na ordem errada.
- Este processo se repete até que mais nenhuma troca seja necessária, ou seja os elementos já estarão ordenados.

$n$  operações são necessárias para a ordenação, ou seja, todos estão já ordenados, e o vetor é varrido uma só vez verificando se algum elemento está desordenado.

- Performance:

- Melhor caso:  $O(n)$ ;

- Pior caso:  $O(n^2)$ ;

- Não recomendado para grandes conjuntos de dados

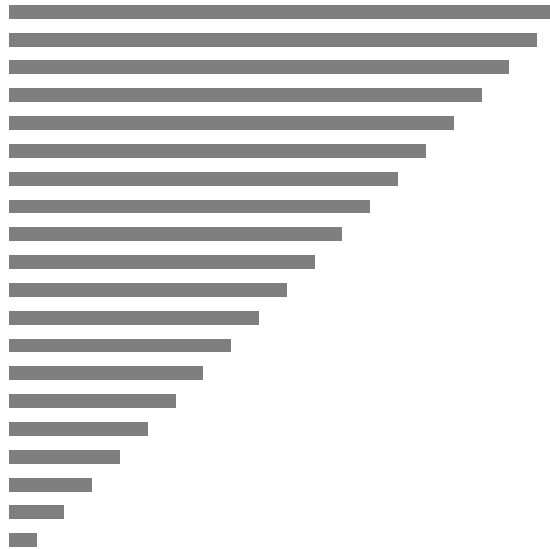
Todos os seus elementos estão fora de ordem (inverso)!

Este algoritmo possui baixo desempenho!

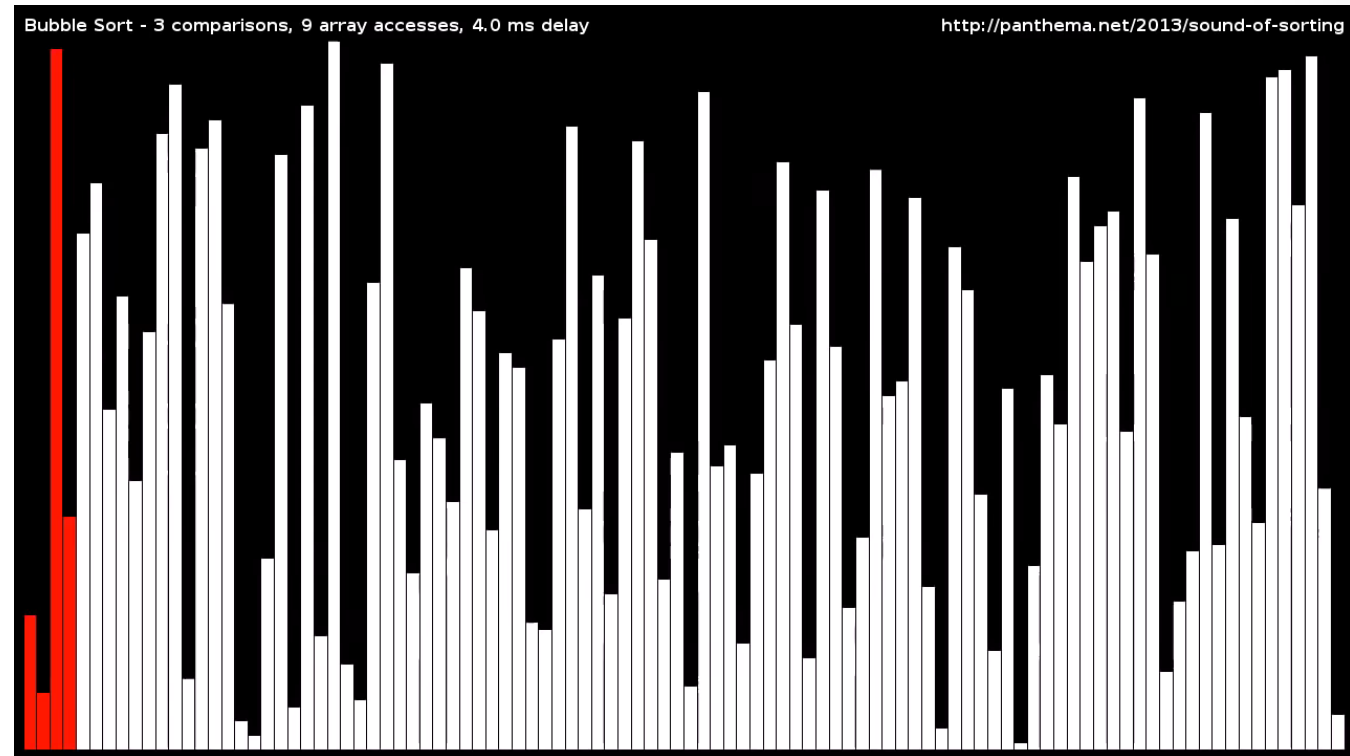
# Estrutura de Dados 2

## Ordenação BubbleSort ou Ordenação por Bolha

- Funcionamento do algoritmo BubbleSort:



<http://www.sorting-algorithms.com/>



<https://www.youtube.com/watch?v=Cq7SMsQBEUw>



## Ordenação BubbleSort ou Ordenação por Bolha

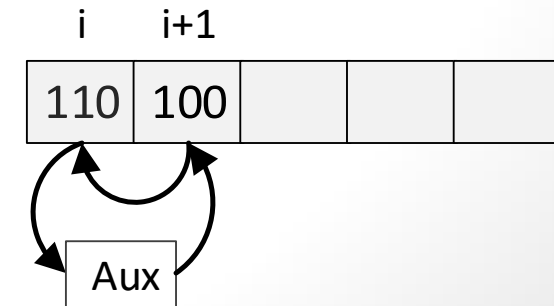
```
void Ordena_bubbleSort(int *v, int n) {  
    int i, continua, aux, fim = n;  
    do {  
        continua = 0;  
        for(i = 0; i < fim - 1; i++) {  
            if(v[i] > v[i+1]) {  
                aux = v[i];  
                v[i] = v[i+1];  
                v[i+1] = aux;  
                continua = i;  
            }  
        }  
        fim--;  
    } while (continua != 0);  
}
```

Durante toda a iteração do laço **for**, serão feitas comparações com as posições vizinhas.

Recebe "i" atual para continuar a busca

Percorre o vetor até a penúltima posição

Troca dois valores consecutivos no vetor



## Ordenação BubbleSort ou Ordenação por Bolha

130	110	150	100	160	140	120
-----	-----	-----	-----	-----	-----	-----

```
for(i = 0; i < fim - 1; i++){  
    if(v[i] > v[i+1]){  
        aux = v[i];  
        v[i] = v[i+1];  
        v[i+1] = aux;  
        continua = i;  
    }  
}
```

O laço **for** simplesmente compara 2 à 2, os elementos do vetor

i = 0	130	110	150	100	160	140	120
i = 1	110	130	150	100	160	140	120
i = 2	110	130	150	100	160	140	120
i = 3	110	130	100	150	160	140	120
i = 4	110	130	100	150	160	140	120
i = 5	110	130	100	150	140	160	120
final	110	130	100	150	140	120	160

Trocar

ok

Trocar

ok

Trocar

Trocar

Fim

O laço **for** posicionará o maior elemento no final do vetor.



## Ordenação BubbleSort ou Ordenação por Bolha

```
void Ordena_bubbleSort(int *v, int n){  
    int i, continua, aux, fim = n;  
    do{  
        continua = 0;  
        for(i = 0; i < fim - 1; i++){  
            if(v[i] > v[i+1]){  
                aux = v[i];  
                v[i] = v[i+1];  
                v[i+1] = aux;  
                continua = i;  
            }  
        }  
        fim--;  
    }while(continua != 0);  
}
```

Só uma passagem pelo **for**,  
não é suficiente para  
ordenação total. Então,  
continuamos...

Já que na primeira passagem pelo  
**for**, o maior elemento foi colocado  
no final, então diminuimos o  
tamanho do vetor



# Estrutura de Dados 2

## Ordenação BubbleSort ou Ordenação por Bolha



1ª iteração do/while

i = 0	130	110	150	100	160	140	120	Trocar
i = 1	110	130	150	100	160	140	120	ok
i = 2	110	130	150	100	160	140	120	Trocar
i = 3	110	130	100	150	160	140	120	ok
i = 4	110	130	100	150	160	140	120	Trocar
i = 5	110	130	100	150	140	160	120	Trocar
final	110	130	100	150	140	120	160	Fim

2ª iteração do/while

i = 0	110	130	100	150	140	120	160	ok
i = 1	110	130	100	150	140	120	160	Trocar
i = 2	110	100	130	150	140	120	160	ok
i = 3	110	100	130	150	140	120	160	ok
i = 4	110	100	130	140	150	120	160	Trocar
final	110	100	130	140	120	150	160	Fim

# Estrutura de Dados 2

## Ordenação BubbleSort ou Ordenação por Bolha



3ª iteração do/while

i = 0	110	100	130	140	120	150	160	Trocar
i = 1	100	110	130	140	120	150	160	ok
i = 2	100	110	130	140	120	150	160	ok
i = 3	100	110	130	140	120	150	160	Trocar
final	100	110	130	120	140	150	160	Fim

4ª iteração do/while

i = 0	100	110	130	120	140	150	160	ok
i = 1	100	110	130	120	140	150	160	ok
i = 2	100	110	130	120	140	150	160	Trocar
final	100	110	120	130	140	150	160	Fim

4ª iteração do/while

i = 0	100	110	120	130	140	150	160	ok
i = 1	100	110	120	130	140	150	160	ok

Não houve mudanças, ordenação concluída



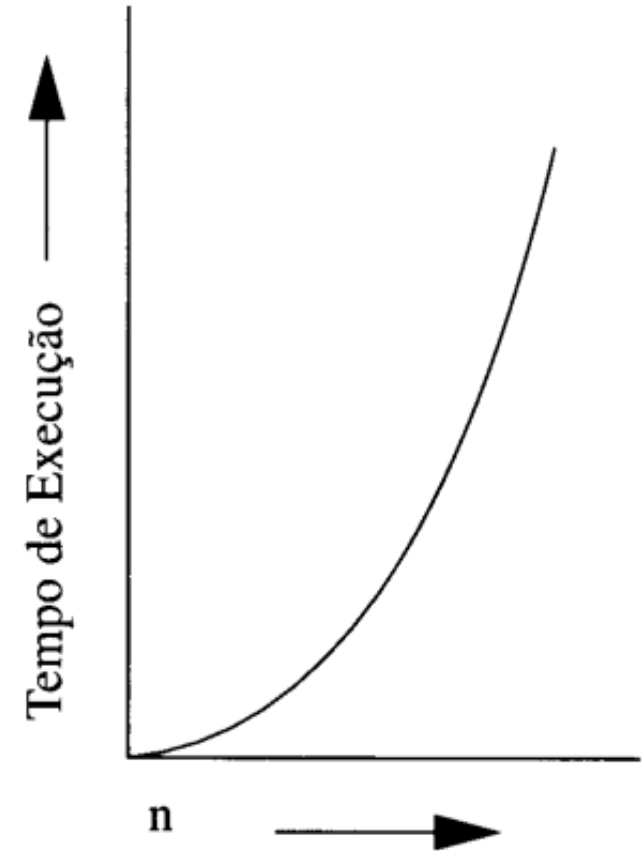
## Ordenação BubbleSort ou Ordenação por Bolha

- Ao analisar qualquer ordenação, deve-se determinar quantas comparações e trocas serão realizadas para o melhor, médio e pior casos. Com a ordenação **BubbleSort**, o número de comparações é sempre o mesmo, porque os dois laços (do/while e for) se repetem um número específico de vezes, estando a lista inicialmente ordenada ou não.
- Nos casos em que a lista esteja menos ordenada, o número de elementos fora de ordem se aproxima do número de comparações, lembrando que para este método existem três trocas para cada elemento fora de ordem.
- Essa é uma ordenação ***n-quadrado***, pois seu tempo de execução é um múltiplo do quadrado do número de elementos. Esse tipo de algoritmo é muito **ineficiente**, quando aplicado em um grande número de elementos, porque o tempo de execução está diretamente relacionado com o número de comparações e trocas.



## Ordenação BubbleSort ou Ordenação por Bolha

- Ignorando o tempo que leva para trocar qualquer elemento fora da posição, assumiremos que cada **comparação** leva 0,001 segundos;
- Para ordenar 10 elementos (comparação + troca), são gastos 0,05 segundos;
- Para ordenar 100 elementos, serão gastos 5 segundos;
- Uma ordenação de 100.000 elementos, o tamanho de uma pequena lista telefônica, levaria em torno de 5.000.000 de segundos, ou 1.388,89 horas, ou ainda, um pouco mais de 57 dias de ordenação contínua.
- O gráfico ao lado mostra como o tempo de execução aumenta com relação ao tamanho da quantidade de elementos a serem ordenados.



Tempo de execução de uma ordenação  $n^2$  em relação ao tamanho da matriz



## Ordenação BubbleSort ou Ordenação por Bolha

- É possível fazer pequenas melhorias na ordenação **Bubble** para eu ela fique mais rápida.
- Ela tem uma peculiaridade. Observe o vetor abaixo:

160	110	150	140	130	100	120
-----	-----	-----	-----	-----	-----	-----

- Um elemento fora de ordem, como o **160**, subirá vagarosamente para sua posição apropriada. Isso sugere uma melhoria na ordenação **BubbleSort**:
  - Em vez de sempre ler o vetor na mesma direção, pode-se inverter a direção em passos subsequentes. Dessa forma, elementos muito fora do lugar irão mais rapidamente para suas posições corretas. Essa versão da ordenação **BubbleSort** é chamada de **ordenação oscilante**, devido ao seu movimento de vaivém sobre a matriz.
  - Embora seja uma melhoria, ela ainda é executada na ordem de um algoritmo ***n-quadrado***, porque o número de comparações não foi alterado, e o número de trocas foi reduzido em uma constante relativamente pequena.



## Ordenação SelectionSort

- SelectionSort ou Ordenação por Seleção

Este algoritmo seleciona o menor elemento do vetor, depois o 2º menor, depois o 3º menor e assim por diante.

- A cada iteração, procura-se o menor valor do vetor e o coloca na primeira posição do vetor;
- Diminui-se então o tamanho do vetor descartando-se a primeira posição, e em seguida, o processo se repete para a segunda posição;
- O processo se repete para todas as posições seguintes do vetor

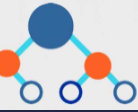
- Performance:

- Melhor caso:  $O(n^2)$ ;
- Pior caso:  $O(n^2)$ ;
- Ineficiente para grandes conjuntos de dados ;
- Estável: não altera a ordem dos dados iguais

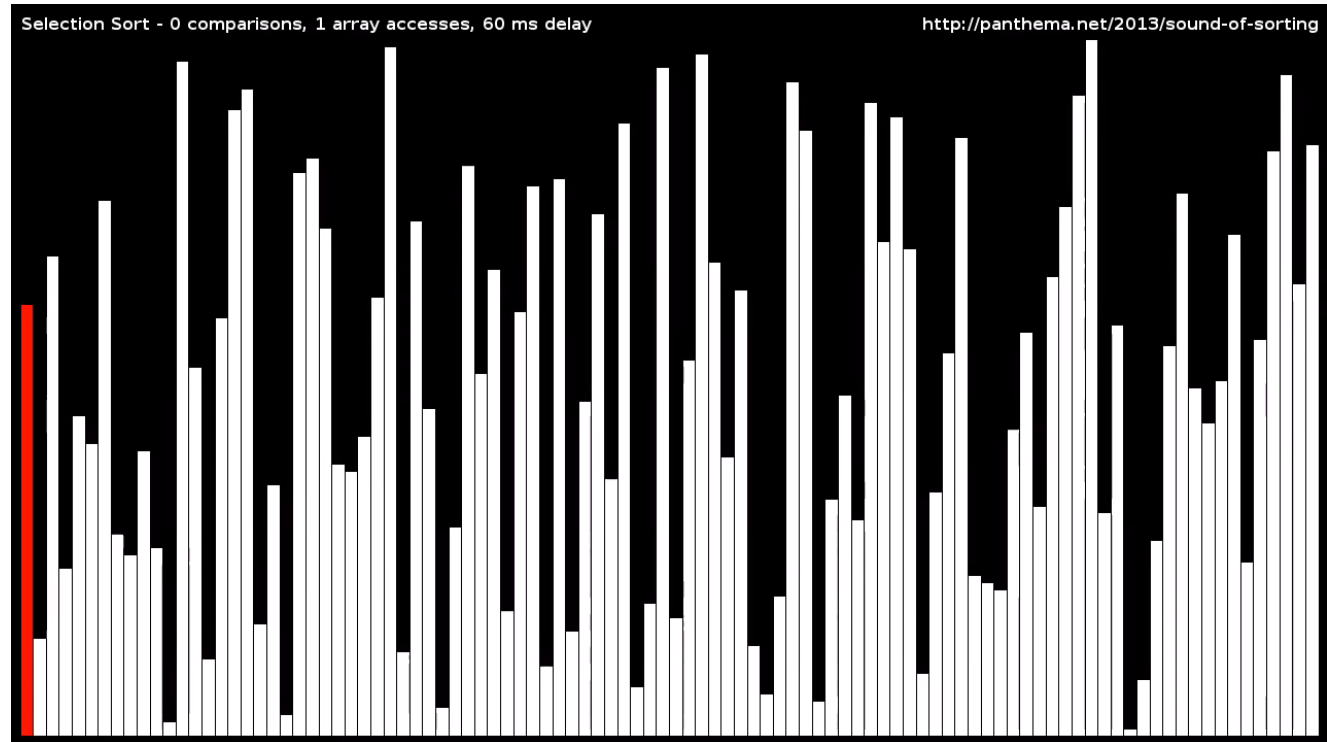
A cada iteração é calculado o menor valor dos elementos que ainda faltam ordenar. Repete-se o processo até que todos os elementos estejam ordenados.

# Estrutura de Dados 2

## Ordenação SelectionSort



- <http://www.sorting-algorithms.com/>



<https://www.youtube.com/watch?v=92BfuxHn2XE>

## Ordenação SelectionSort

```
void Ordena_selectionSort(int *v, int n){  
    int i, j, menor, troca;  
    for(i = 0; i < n - 1; i++){  
        menor = i;  
        for(j = i + 1; j < n; j++){  
            if (v[j] < v[menor]){  
                menor = j;  
            }  
        }  
        if(i != menor){  
            troca = v[i];  
            v[i] = v[menor];  
            v[menor] = troca;  
        }  
    }  
}
```

Procura o menor elemento em relação a **i**

Troca os valores da posição atual com a “menor”

```
"C:\Users\angelot\Desktop\Aulas 1\ semestre 2016\ED2D3\Aulas\Aula ...  
Vetor desordenado:  120 150 110 130 100 160 140 190 180 170  
Ordenando por metodo selectionSort:  
Vetor ja ordenado:  100 110 120 130 140 150 160 170 180 190  
  
Process returned 0 (0x0)   execution time : 0.787 s  
Press any key to continue.
```





# Estrutura de Dados 2

## Ordenação SelectionSort

Sem Ordenar

130	110	150	100	160	140	120
-----	-----	-----	-----	-----	-----	-----

$l = 0$

130	110	150	100	160	140	120
100	110	150	130	160	140	120

$l = 1$

100	110	150	130	160	140	120
100	110	150	130	160	140	120

$l = 2$

100	110	150	130	160	140	120
100	110	120	130	160	140	150

$l = 3$

100	110	120	130	160	140	150
100	110	120	130	160	140	150

$l = 4$

100	110	120	130	160	140	150
100	110	120	130	140	160	150

$l = 5$

100	110	120	130	140	160	150
100	110	120	130	140	150	160

100	110	120	130	140	150	160
-----	-----	-----	-----	-----	-----	-----

Vetor Ordenado





## Ordenação SelectionSort

- Infelizmente como na ordenação **BubbleSort**, o laço mais externo é executado  $n - 1$  vezes e o laço interno  $\frac{1}{2}(n)$  vezes. Como resultado, a ordenação por seleção requer  $\frac{1}{2}(n^2 - n)$  comparações, o que a torna muito lenta para um número grande de itens.
- Para o melhor caso, quando a lista está inicialmente ordenada, apenas  $n - 1$  elementos precisam ser movimentados, e cada movimento requer três trocas.
- O seu pior caso, aproxima-se do número de comparações.
- Embora o número de comparações para a ordenação **BubbleSort** e para a ordenação por seleção seja o mesmo, o número de trocas no caso médio, é muito menor para a ordenação por seleção.



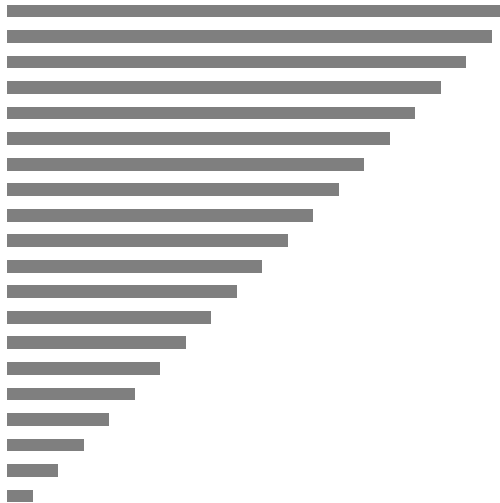
## Ordenação InsertionSort ou Ordenação por Inserção

- Tem este nome pois se assemelha ao processo de ordenação de um conjunto de cartas de baralho com as mãos;
  - Pega-se uma carta de cada vez e a coloca em seu devido lugar, sempre deixando as cartas da mão em ordem.
- 
- Performance:
    - Melhor caso:  $O(n)$ ;
    - Pior caso:  $O(n^2)$ ;
    - Eficiente para conjunto pequenos de dados;
    - **Estável**: não altera a ordem de dados iguais;
    - Capaz de ordenar os dados a medida em que os recebe (tempo real).

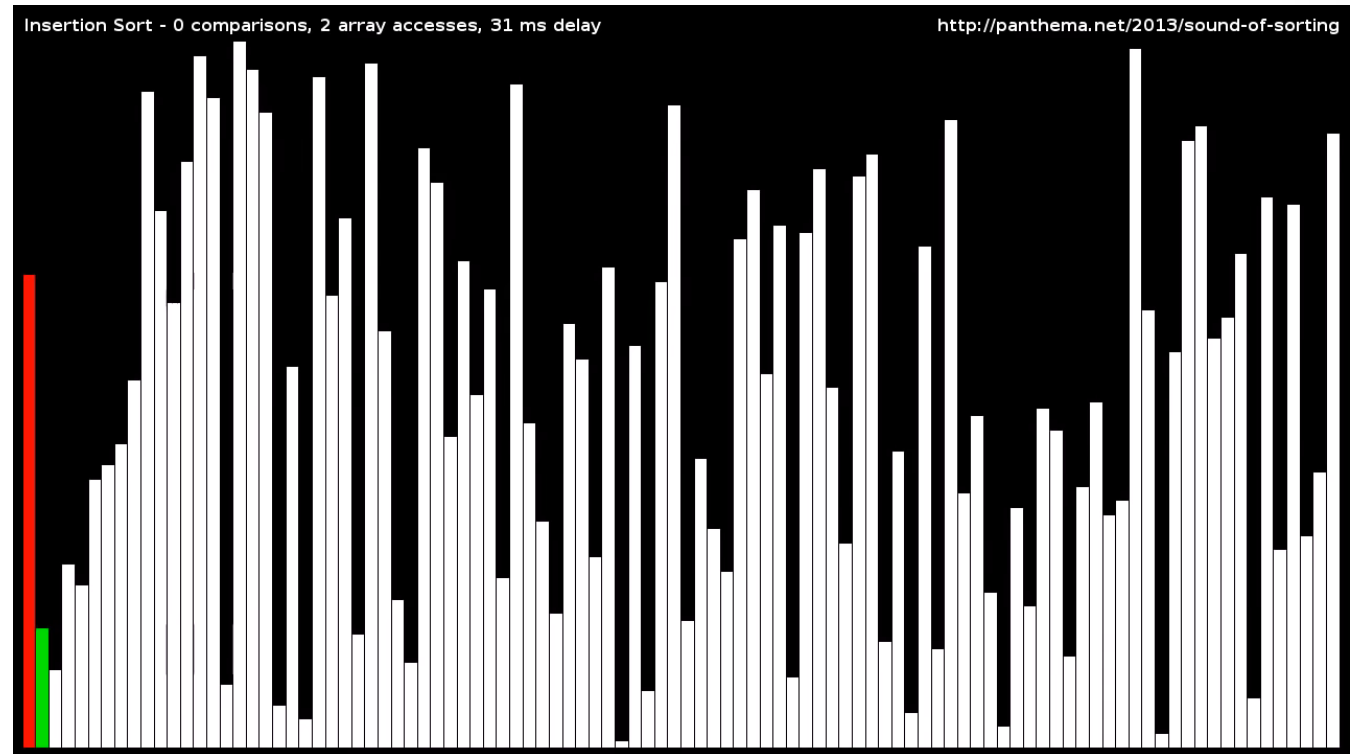


## Ordenação InsertionSort ou Ordenação por Inserção

- Funcionamento do algoritmo InsertionSort:



- <http://www.sorting-algorithms.com/>



<https://www.youtube.com/watch?v=8oJS1BMKE64&t=4s>

## Ordenação *InsertionSort* ou Ordenação por Inserção

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

Move os elementos  
menores encontrados  
para a frente



100	110	130	140	150	160	120
100	110	120	130	140	150	160

Diagram illustrating the insertion sort process. A red arrow points from the element 120 in the first row to its new position at the beginning of the second row. The elements 130, 140, 150, and 160 are shifted one position to the right in the second row. The indices *j* and *i* are marked above the last two columns of the first row.

## Ordenação *InsertionSort* ou Ordenação por Inserção

Sem Ordenar

130	110	150	100	160	140	120
-----	-----	-----	-----	-----	-----	-----

$l = 1$

130	110	150	100	160	140	120
110	130	150	100	160	140	120

Trocar

$l = 2$

110	130	150	100	160	140	120
110	130	150	100	160	140	120

ok

$l = 3$

110	130	150	100	160	140	120
100	110	130	150	160	140	120

Trocar

$l = 4$

100	110	130	150	160	140	120
100	110	130	150	160	140	120

ok

$l = 5$

100	110	130	150	160	140	120
100	110	130	140	150	160	120

Trocar

$l = 6$

100	110	130	140	150	160	120
100	110	120	130	140	150	160

Trocar

Ordenado

100	110	120	130	140	150	160
-----	-----	-----	-----	-----	-----	-----

## Ordenação *InsertionSort* ou Ordenação por Inserção

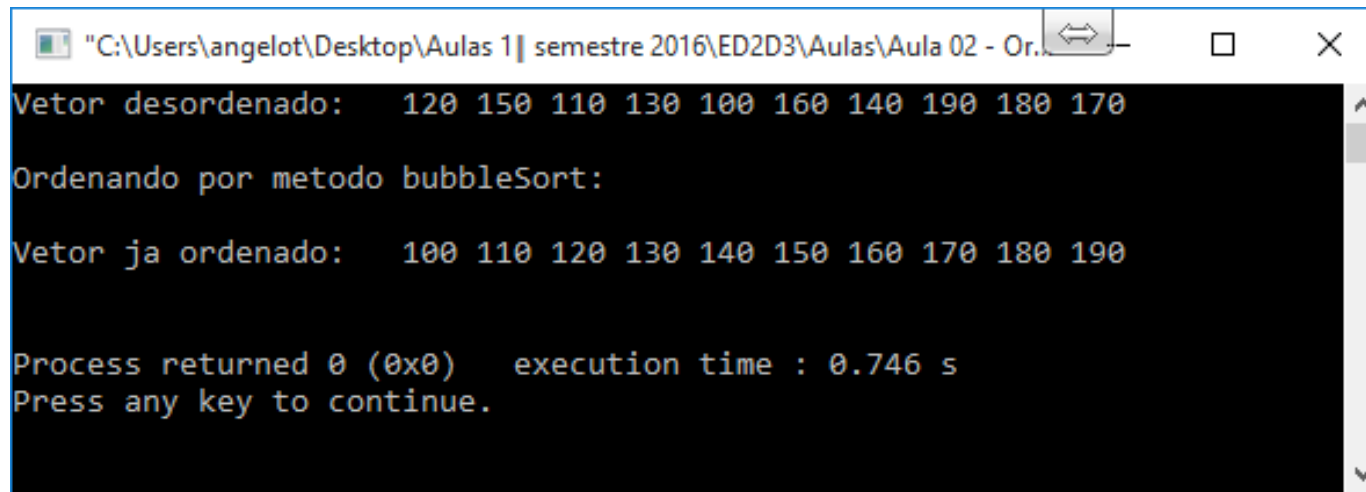
- Ao contrario da ordenação **BubbleSort**, e da ordenação por seleção, o número de comparações que ocorrem durante a ordenação por inserção depende de como a lista está inicialmente ordenada. Se a lista estiver em ordem, o número de comparações será  $n - 1$ . Se estiver fora de ordem, o número de comparações será  $\frac{1}{2}(n^2 + n)$ .
- Portanto, para o pior caso, a ordenação por inserção é tão ruim quanto a ordenação bolha e a ordenação por seleção e, para o caso médio, é somente um pouco melhor. No entanto, a ordenação por inserção tem duas vantagens:
  - Ela se comporta **naturalmente**, isto é, trabalha menos quando a matriz já está ordenada e o máximo quando a matriz está ordenada no sentido inverso. Isso torna a ordenação por inserção excelente para dados que estão quase em ordem.
  - **Ela não rearranja elementos de mesma chave**. Isso significa que dados ordenados por duas chaves, permanecem ordenados para ambas as chaves após uma ordenação por inserção.
- Muito embora o número de comparações possa ser razoavelmente baixo para certos conjuntos de dados, a matriz precisa ser deslocada cada vez que um elemento é colocado em sua posição correta. Como resultado o número de movimentações pode ser significativo.



# Estrutura de Dados 2

## Atividade 3

- Faça um programa para ordenação de um vetor de inteiros, utilizando o algoritmo de ordenação **BubbleSort**, que apresente sua saída como o exemplo abaixo:



```
"C:\Users\angelot\Desktop\Aulas 1\ semestre 2016\ED2D3\Aulas\Aula 02 - Or...
Vetor desordenado:  120 150 110 130 100 160 140 190 180 170

Ordenando por metodo bubbleSort:

Vetor ja ordenado:  100 110 120 130 140 150 160 170 180 190

Process returned 0 (0x0)   execution time : 0.746 s
Press any key to continue.
```

- Entregue no Moodle como: Atividade 3.





# Estrutura de Dados 2

## Atividade 4

- Implemente um programa para ordenação de um vetor de inteiros, utilizando o conceito de Ordenação Oscilante, visto anteriormente, para o algoritmo de ordenação **BubbleSort**.
- Entregue na plataforma Moodle como Atividade 4.

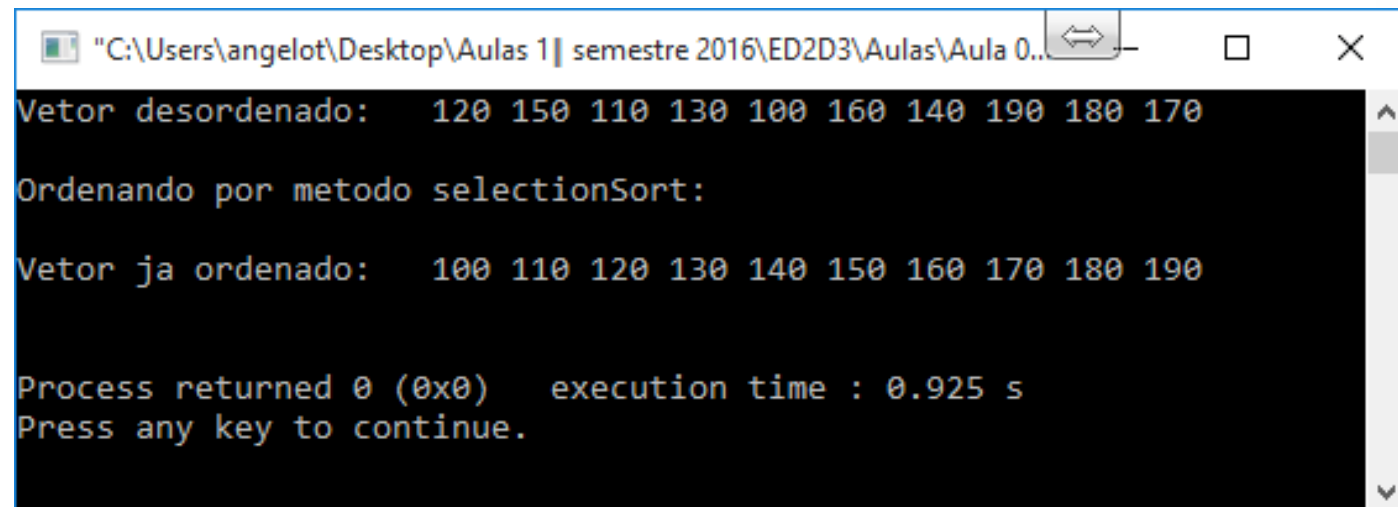
```
"C:\Users\angelot\Documents\Aulas\EDA2\Aulas\01\material apoio\CocktailSort.exe"
Vetor desordenado:  120 150 110 130 100 160 140 190 180 170
Ordenando por metodo bubbleSort Oscilante - CocktailSort:
Vetor ja ordenado:  100 110 120 130 140 150 160 170 180 190
Process returned 0 (0x0)   execution time : 0.072 s
Press any key to continue.
```



# Estrutura de Dados 2

## Atividade 5

- Faça um programa para ordenação de um vetor de inteiros, utilizando o algoritmo de ordenação ***SelectionSort***, que apresente sua saída como o exemplo abaixo:
- Entregue no Moodle como: Atividade 5.



```
"C:\Users\angelot\Desktop\Aulas 1 | semestre 2016\ED2D3\Aulas\Aula 0...
Vetor desordenado:  120 150 110 130 100 160 140 190 180 170
Ordenando por metodo selectionSort:
Vetor ja ordenado:  100 110 120 130 140 150 160 170 180 190

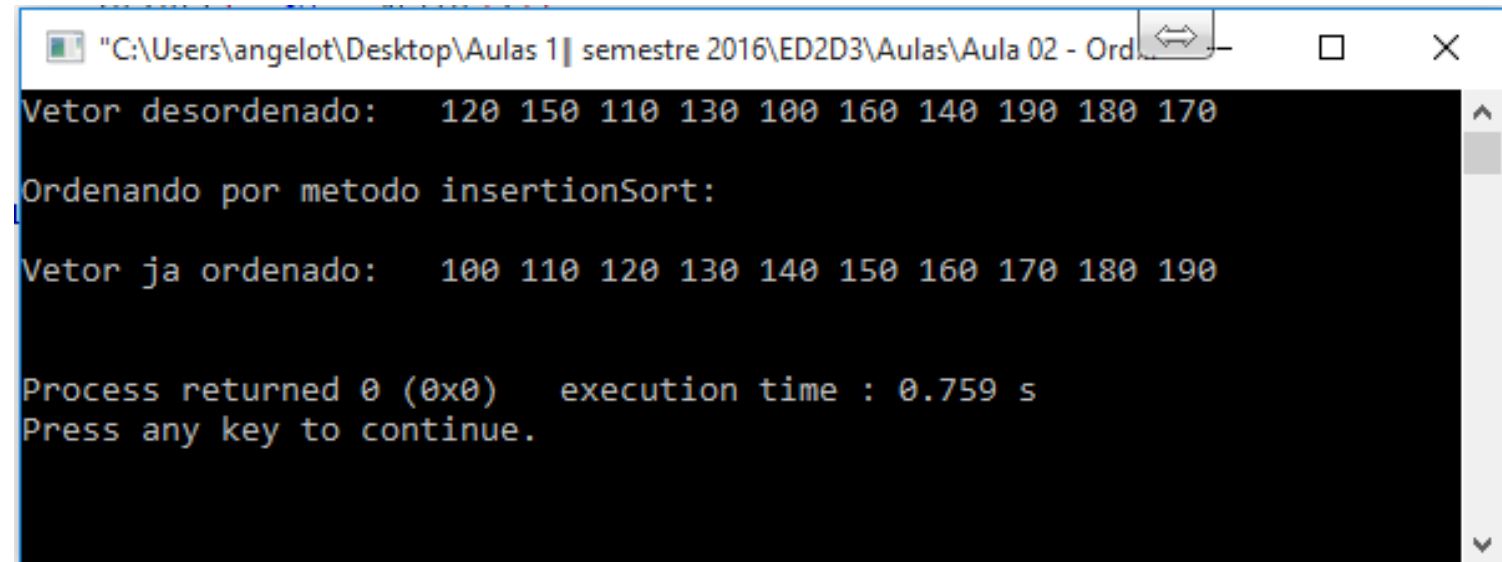
Process returned 0 (0x0)   execution time : 0.925 s
Press any key to continue.
```



# Estrutura de Dados 2

## Atividade 6

- Faça um programa para ordenação de um vetor de inteiros, utilizando o algoritmo de ordenação ***InsertionSort***, que apresente sua saída como o exemplo abaixo:
- Entregue no Moodle como: Atividade 6.



```
"C:\Users\angelot\Desktop\Aulas 1\ semestre 2016\ED2D3\Aulas\Aula 02 - Ord...
Vetor desordenado:  120 150 110 130 100 160 140 190 180 170
Ordenando por metodo insertionSort:
Vetor ja ordenado:  100 110 120 130 140 150 160 170 180 190

Process returned 0 (0x0)   execution time : 0.759 s
Press any key to continue.
```

