


# Ciência da Computação

## Algoritmos e Estrutura de Dados 1

# Ordenação



## Parte 1 - Estratégias simples

Prof. Rafael Liberato  
liberato@utfpr.edu.br

# Objetivos

# Roteiro

⊗ **Contexto**

⊗ **Algoritmos**

⇒ Bubble sort

⇒ Selection sort

⇒ Insertion sort

⊗ **Resumo**

Contexto



# Contexto

⊗ Qual a vantagem de se ter um conjunto de elementos ordenados?

⇒ Desempenho

Lista Telefônica



Como seria encontrar o telefone do João se a lista telefônica não estivesse em ordem alfabética?

# Como ordenar?

## ⊗ Temos duas alternativas:

- ⇒ Inserimos os elementos na ordem correta. A ordenação é garantida por construção
- ⇒ Aplicamos um algoritmo para ordenar os elementos de um conjunto já criado

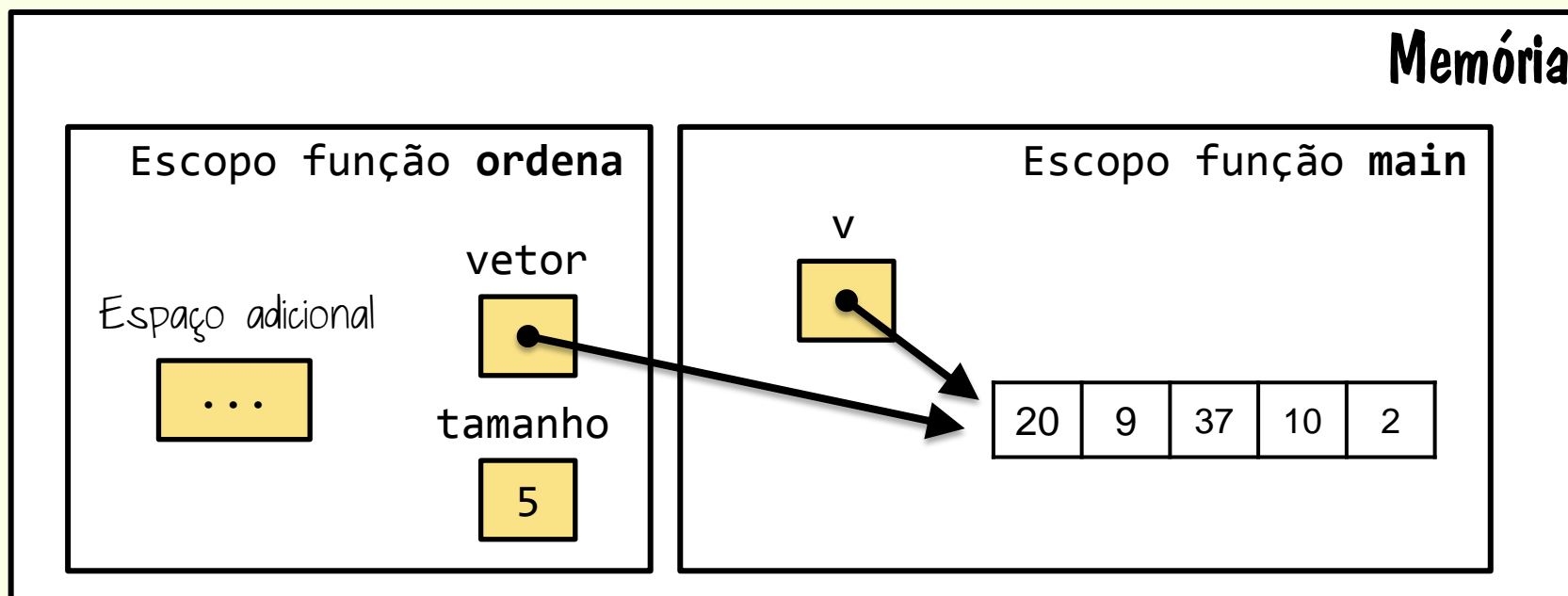
## ⊗ Preocupações

- ⇒ Eficiência em termos de tempo (rápido)
- ⇒ Eficiência em termos de espaço (ocupar pouca memória)

# Como ordenar?

⊛ **Protótipo** void ordena(int\* vetor, int tamanho)

```
int v[5] = {20,9,37,10,2};
ordena(v,5);
```

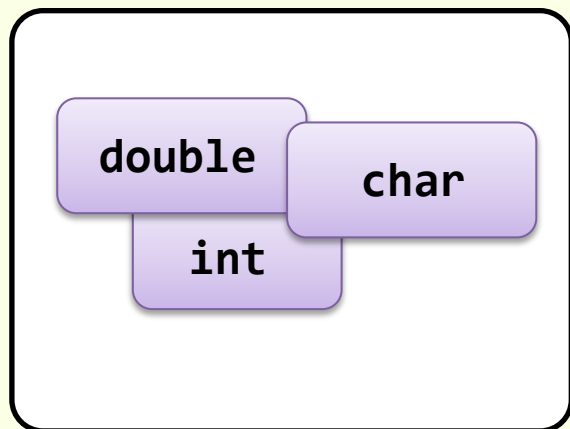


# Quais elementos?

⊛ Os algoritmos de ordenação podem ser aplicados a qualquer informação

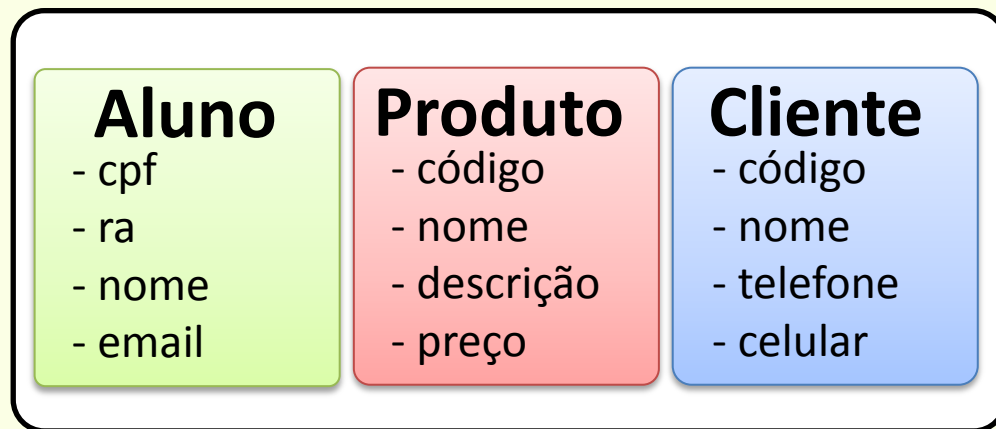
⇒ Desde que exista uma ordem bem definida entre os elementos

## Tipos primitivos



Ordem natural

## Tipos compostos



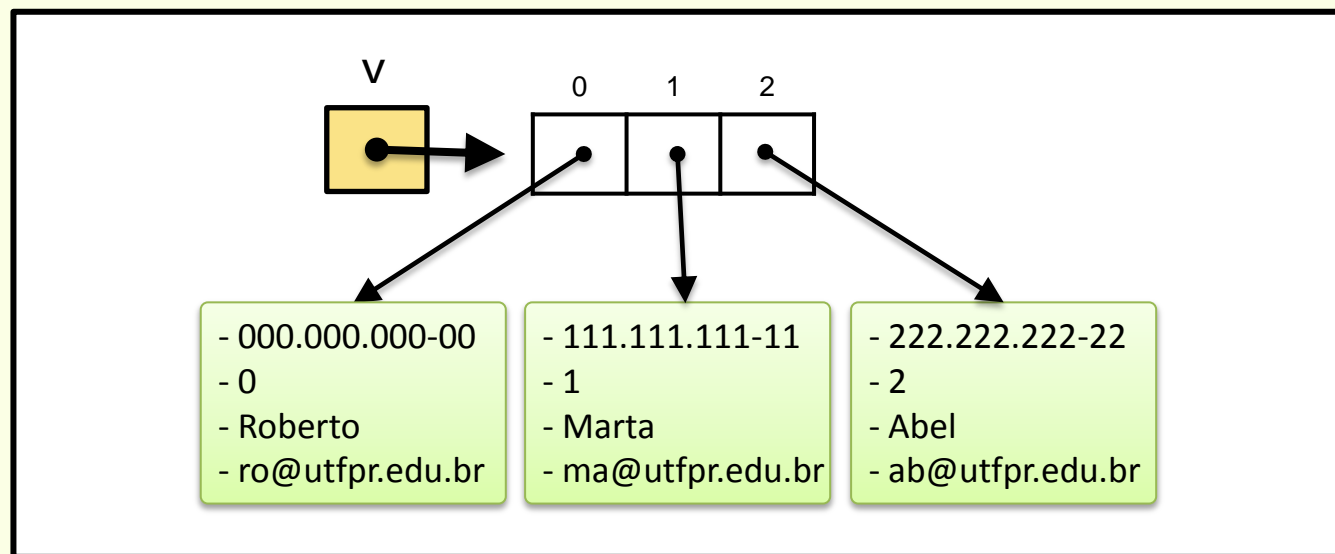
Devemos definir quem será a chave da ordenação



# Quais elementos?

⊗ Para ordenação de tipos compostos, normalmente o vetor armazena somente o endereço de onde a informação se encontra

⇒ Isso evita a movimentação de grandes quantidades de informações



## Aluno

- cpf
- ra
- nome
- email

# Algoritmos



# Algoritmos

- ⊗ Vamos analisar alguns algoritmos e suas características
  - ⇒ Bubble sort
  - ⇒ Selection sort
  - ⇒ Insertion sort

# Bubble Sort



# Bubble Sort

## ⊛ Origem do nome

⇒ Elementos maiores sobem como bolhas até suas posições corretas

## ⊛ Processo básico

- ⇒ Quando dois elementos estão fora de ordem, troque-os de posição até que o  $i$ -ésimo elemento de maior valor do vetor seja levado para as posições finais do vetor.
- ⇒ Continue o processo até que todo o vetor esteja ordenado

# Bubble Sort

## 1ª Iteração

<b>25</b>	<b>48</b>	99	12	57	86	33	89	25 x 48
25	<b>48</b>	<b>99</b>	12	57	86	33	89	48 x 99
25	48	<b>99</b>	<b>12</b>	57	86	33	89	99 X 12 TROCA
25	48	12	<b>99</b>	<b>57</b>	86	33	89	99 X 57 TROCA
25	48	12	57	<b>99</b>	<b>86</b>	33	89	99 X 86 TROCA
25	48	12	57	86	<b>99</b>	<b>33</b>	89	99 X 33 TROCA
25	48	12	57	86	33	<b>99</b>	<b>89</b>	99 X 89 TROCA
25	48	12	57	86	33	89	<u><b>99</b></u>	Final da primeira iteração. Faltam N-1



Maior elemento do vetor, ele já está na sua posição final.

# Bubble Sort

2ª Iteração

25	48	12	57	86	33	89	<u>99</u>	25 x 48
25	48	12	57	86	33	89	<u>99</u>	48 x 12 TROCA
25	12	48	57	86	33	89	<u>99</u>	48 X 57
25	12	48	57	86	33	89	<u>99</u>	57 X 86
25	12	48	57	86	33	89	<u>99</u>	86 X 33 TROCA
25	12	48	57	33	86	89	<u>99</u>	86 X 89
25	12	48	57	33	86	<u>89</u>	<u>99</u>	Final da segunda iteração

# Bubble Sort

3ª Iteração

25	12	48	57	33	86	89	99	25 x 12 TROCA
12	25	48	57	33	86	89	99	25 x 48
12	25	48	57	33	86	89	99	48 X 57
12	25	48	57	33	86	89	99	57 X 33 TROCA
12	25	48	33	57	86	89	99	57 X 86
12	25	48	33	57	86	89	99	Final da terceira Iteração



# Bubble Sort

4ª Iteração

12	25	48	33	57	86	89	99	12 x 25
12	25	48	33	57	86	89	99	25 x 48
12	25	48	33	57	86	89	99	48 X 33 TROCA
12	25	33	48	57	86	89	99	48 X 57
12	25	33	48	57	86	89	99	Final da quarta iteração

# Bubble Sort

5ª Iteração

12	25	33	48	57	86	89	99	12 x 25
12	25	33	48	57	86	89	99	25 x 33
12	25	33	48	57	86	89	99	33 X 48
12	25	33	48	57	86	89	99	Final da quinta iteração



Como não houve nenhuma troca, não precisamos verificar mais. O vetor está ordenado

# Bubble Sort

## ⊗ Versão Iterativa (1)

```
void bubble_sort_v1 (int* v, int n){
    int fim,i;
    for (fim=n-1; fim>0; fim--){
        for (i=0; i<fim; i++){

            if (v[i]>v[i+1]) {
                troca(&v[i], &v[i+1]);
            }
        }
    }
}
```

```
void troca(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

# Bubble Sort

## ⊛ Versão Iterativa (2)

```
void bubble_sort_v2 (int* v, int n){
    int i, fim;
    for (fim=n-1; fim>0; fim--) {
        int houve_troca = 0;
        for (i=0; i<fim; i++){
            if (v[i]>v[i+1]) {
                troca(&v[i], &v[i+1]);
                houve_troca = 1;
            }
        }
        if (houve_troca == 0) return;
    }
}
```

```
void troca(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Finaliza quando encontra uma passagem inteira sem trocas

# Bubble Sort

## \* Análise para um vetor com 10 elementos

### Comparações

1 passada: 9 comparações (N-1)  
 2 passada: 8 comparações (N-2)  
 3 passada: 7 comparações (N-3)  
 ...

### Progressão Aritmética

$$9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 45$$

$$(N-1) + (N-2) + (N-3) + (N-4) \dots + 1 = \mathbf{N*(N-1)/2}$$

$$N*(N-1)/2 = 10*9/2 = 45$$

\* Assim o algoritmo faz cerca de  **$N^2/2$**  comparações  
 podemos ignorar o -1 porque ele não faz muita diferença,  
 especialmente se N for grande

# Bubble Sort

## ⊗ Análise para um vetor com 10 elementos

### Trocas

Depende da disposição dos elementos no vetor

#### Pior caso

4	3	2	1
---	---	---	---

Número de trocas é igual ao número de comparações

**$N^2/2$  trocas**

#### Melhor caso

1	2	3	4
---	---	---	---

Não faz nenhuma troca

#### Caso Médio

Vetor aleatório

Normalmente o número de trocas é menor que o número de comparações

**$N^2/4$  trocas**

# Bubble Sort

## ⊗ Análise para um vetor com 10 elementos

### Resumo

$$\frac{N^2}{2} \text{ (comparações)} + \frac{N^2}{4} \text{ (trocas)}$$

- ⊗ Tanto o número de comparações quanto o número de trocas são proporcionais a  $n^2$
- ⊗ Na notação  $O$  podemos ignorar as constantes 2 e 4 e dizer que o algoritmo executa em tempo  $O(n^2)$

# Bubble Sort

## \* Resumo

### Pior Caso

4	3	2	1
---	---	---	---

**$N^2/2$  comparações**

**$N^2/2$  trocas**

### Melhor Caso

1	2	3	4
---	---	---	---

**$N^2/2$  comparações**

**$N^2/2$  trocas**

### Caso Médio

Vetor aleatório

**$N^2/2$  comparações**

**$N^2/4$  trocas**



# Bubble Sort

## ⊛ Versão Recursiva

```
void bubble_sort_rec (int* v, int n){
    int i;
    int houve_troca = 0;
    for (i=0; i<n-1; i++){
        if (v[i]>v[i+1]) {
            troca(&v[i], &v[i+1]);
            houve_troca = 1;
        }
    }
    if (houve_troca != 0 && n > 1){
        bubble_sort_rec (v, n-1);
    }
}
```

```
void troca(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

# Selection Sort




# Selection Sort

- ⊗ Melhora a ordenação do método bubble reduzindo o número de **trocas** de  $O(N^2)$  para  $O(N)$ .
- ⊗ O número de comparações permanece o mesmo  $O(N^2)$
- ⊗ Porém, ainda apresenta uma melhora significativa
- ⊗ Ideia do algoritmo


- Ache o menor elemento a partir da posição 0 e troque este elemento com o elemento da posição 0.
- Ache o menor elemento a partir da posição 1 e troque este elemento com o elemento da posição 1.
- Ache o menor elemento a partir da posição 2 e troque este elemento com o elemento da posição 2.
- E assim sucessivamente..

# Selection Sort


1ª Iteração

25	48	99		57	86	33	89	Encontra o menor
<b>12</b>	48	99	<b>25</b>	57	86	33	89	troca

2ª Iteração

<u>12</u>	<b>48</b>	99		57	86	33	89	Encontra o menor
<u>12</u>	<b>25</b>	99	<b>48</b>	57	86	33	89	troca

3ª Iteração

<u>12</u>	<u>25</u>	<b>99</b>	48	57	86		89	Encontra o menor
<u>12</u>	<u>25</u>	<b>33</b>	48	57	86	<b>99</b>	89	troca

# Selection Sort

4ª Iteração				menor ↓					
	12	25	33	48	57	86	99	89	Encontra o menor
	12	25	33	48	57	86	99	89	troca

5ª Iteração					menor ↓				
	12	25	33	48	57	86	99	89	Encontra o menor
	12	25	33	48	57	86	99	89	troca

6ª Iteração						menor ↓			
	12	25	33	48	57	86	99	89	Encontra o menor
	12	25	33	48	57	86	99	89	troca

# Selection Sort

6ª Iteração

12	25	33	48	57	86	99	89	menor ↓	Encontra o menor
12	25	33	48	57	86	89	99		troca
12	25	33	48	57	86	89	99		

# Selection Sort

```
void selection_sort (int* v, int n){
    int i, min;
    count_troca = 0;
    for (i=0; i<n-1; i++){
        min = menorElemento(v, n, i);
        troca(&v[i], &v[min]);
    }
}
```

```
void troca(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
int menorElemento(int v[], int n, int indice){
    int i, min = indice;
    for(i=indice+1; i<n; i++){
        if(v[i] < v[min]){
            min = i;
        }
    }
    return min;
}
```

# Selection Sort

## ⊗ **Análise para um vetor com 10 elementos**

Resumo  $\frac{N^2}{2}$  (comparações) +  $N - 1$  (trocas)

- ⊗ Executa o mesmo número de comparações, proporcional a  $n^2$ .  
Porém, o número de trocas é proporcional a  $n$ .
- ⊗ Para grandes valores de  $N$  o tempo de comparação predominará, mas é inquestionavelmente mais rápida porque há menos trocas.



# Selection Sort

## \* Resumo

### Pior Caso

4	3	2	1
---	---	---	---

**$N^2/2$  comparações**

**$N-1$  trocas**

### Melhor Caso

1	2	3	4
---	---	---	---

**$N^2/2$  comparações**

**$N-1$  trocas**

\*pode ser menor dependendo da versão do algoritmo

### Caso Médio

Vetor aleatório

**$N^2/2$  comparações**

**$N-1$  trocas**

\*pode ser menor dependendo da versão do algoritmo

# InSertion Sort



# Insertion Sort

⊗ Na maioria dos casos, a ordenação por inserção é a melhor das ordenações elementares.

⊗ É um pouco mais rápida que a ordenação por seleção em situações normais

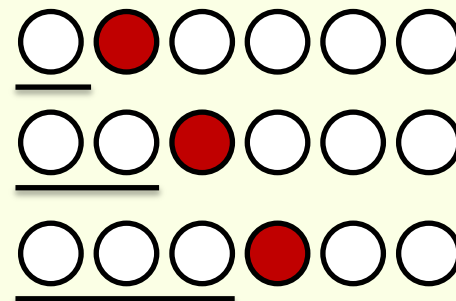
⊗ Ideia do algoritmo

⇒ Separar o vetor em duas partes

⇒ Uma parte ordenada e o restante do vetor

- Percorrer todos os elementos da parte desordenada, inserindo-os na posição correta da parte ordenada

— Parte ordenada  
● Elemento a ser inserido na parte ordenada



# Insertion Sort

1ª Iteração	<u>25</u>	↓ 48	99	12	57	86	33	89	Inserir o 48 na parte ordenada
2ª Iteração	<u>25</u>	<u>48</u>	↓ 99	12	57	86	33	89	Inserir o 99 na parte ordenada
3ª Iteração	<u>25</u>	<u>48</u>	<u>99</u>	↓ 12	57	86	33	89	Inserir o 12 na parte ordenada
	<u>25</u>	<u>48</u>	<u>99</u>	57	86	33	89		
	<u>12</u>	<u>25</u>	<u>48</u>	<u>99</u>	57	86	33	89	

# Insertion Sort

4ª Iteração

12	25	48	99	57	86	33	89
12	25	48		99	86	33	89
12	25	48	57	99	86	33	89

Inserir o 57 na parte ordenada

5ª Iteração

12	25	48	57	99	86	33	89
12	25	48	57		99	33	89
12	25	48	57	86	99	33	89

Inserir o 86 na parte ordenada

# Insertion Sort

5ª Iteração

12	25	48	57	86	99	33	89
		Deslocar				↓	
12	25		48	57	86	99	89
12	25	33	48	57	86	99	89

Inserir o 33 na parte ordenada

6ª Iteração

12	25	33	48	57	86	99	89
						Deslocar	↓
12	25	33	48	57	86		99
12	25	33	48	57	86	89	99

Inserir o 86 na parte ordenada

# Insertion Sort

## \* Análise para um vetor com 10 elementos

Quantas **comparações** o algoritmo requer?

$$1 + 2 + 3 + \dots + N-1 = \frac{N*(N-1)}{2}$$

- \* Porém, em cada iteração não é realizada todas as comparações. Isso somente acontece no pior caso.
- \* Em média, apenas **metade** do número máximo de itens é de fato comparada antes do ponto de inserção ser encontrado

$$\frac{N*(N-1)}{4}$$

# Insertion Sort

## \* Análise para um vetor com 10 elementos

Quantas **cópias** o algoritmo requer?

- \* O número de cópias é aproximadamente o mesmo que o número de comparações. Em média:

$$N*(N-1) / 4$$

- \* Contudo, uma cópia não é tão demorada quanto uma troca.



# Insertion Sort

## \* Resumo

### Comparações/Cópias

#### Pior caso

4	3	2	1
---	---	---	---

Todas as cópias possíveis

**$N^2/2$  comparações**

**$N^2/2$  cópias**

#### Melhor caso

1	2	3	4
---	---	---	---

Não faz nenhuma cópia

**$N$  comparações**

**$0$  cópias**

#### Caso Médio

Vetor aleatório

**$N^2/4$  comparações**

**$N^2/4$  cópias**

ReSumo



# Resumo

- ⊗ Todos os algoritmos apresentados executam em tempo  $O(N^2)$ .

Entretanto, alguns podem ser substancialmente mais rápidos que outros.

- ⊗ A ordenação pelo método da bolha é a menos eficiente

- ⊗ A ordenação por seleção minimiza o número de trocas, mas o número de comparações ainda é alto

Essa ordenação poderá ser útil quando a quantidade de dados for pequena.

- ⊗ A ordenação por inserção é a mais versátil das três

# Referências

- Waldemar Celes, Renato Cerqueira, José Lucas Rangel, Introdução a Estruturas de Dados, Editora Campus (2004). Capítulo 16 – Ordenação
- LAFORE