



# Métodos de Ordenação

## Parte 1

---

### **Introdução à Ciência da Computação II**

Prof. Diego Raphael Amancio



# O Problema da Ordenação

---

- Ordenação (ou classificação) é largamente utilizada
  - Listas telefônicas e dicionários
  - Grandes sistemas de BD e processamento de dados
    - 25% da computação em ordenação
  - Algoritmos de ordenação são ilustrativos
    - Como resolver problemas computacionais
    - Como lidar com estruturas de dados
    - Como desenvolver algoritmos elegantes e como analisar e comparar seus desempenhos



# O Problema da Ordenação

---

- Ordenar (ou classificar)
  - *Definição: **organizar uma seqüência de elementos** de modo que os mesmos estabeleçam alguma relação de ordem*
    - *Diz-se que os elementos  $k_1, \dots, k_n$  estarão dispostos de modo que  $k_1 \leq k_2 \leq \dots \leq k_n$*
  - Facilita a busca/localização/recuperação de um elemento dentro do conjunto a que pertence
    - **Será?**



# O Problema da Ordenação

---

- **Ocasionalmente**, dá menos trabalho buscar um elemento em um conjunto desordenado do que ordenar primeiro e depois buscar
- Por outro lado, **se a busca for uma operação freqüente**, vale a pena ordenar
  - A classificação pode ser feita somente uma vez!
- Depende das circunstâncias!



# O Problema da Ordenação

---

- Terminologia/conceitos
  - Ordenação de **registros** (em um “arquivo”), em que cada registro é ordenado por sua **chave**
  - Ordenação **interna** vs. **externa**
  - Ordenação **estável**: ordenação original de registros com mesma chave é preservada após a ordenação dos registros



# O Problema da Ordenação

---

- Terminologia/conceitos
  - Ordenação sobre os próprios registros
    - Os registros são trocados de posição
  - Ordenação por endereços
    - Mantém-se uma tabela de ponteiros para os registros e alteram-se somente os ponteiros durante a ordenação



# O Problema da Ordenação

- Exemplo: ordenação sobre os próprios registros

	Chave	Outros campos
Registro 1	4	DDD
Registro 2	2	BBB
Registro 3	1	AAA
Registro 4	5	EEE
Registro 5	3	CCC

Arquivo

(a) Arquivo original.

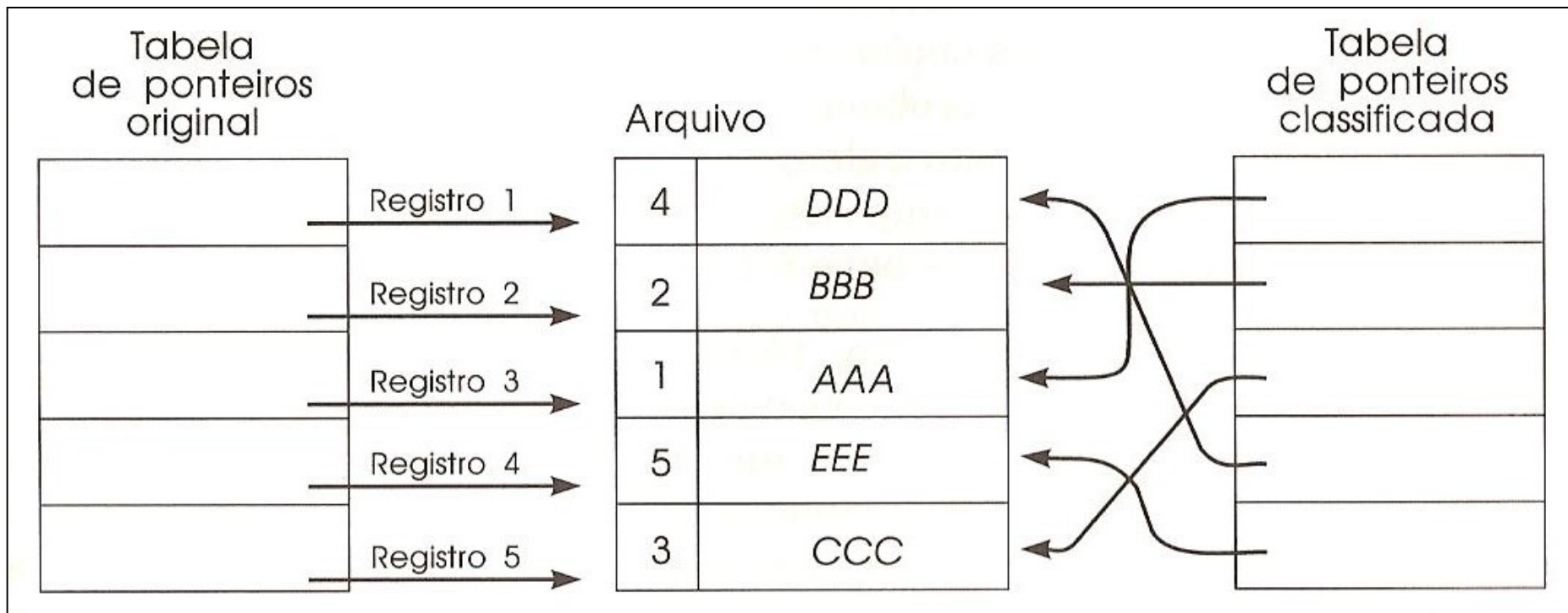
1	AAA
2	BBB
3	CCC
4	DDD
5	EEE

Arquivo

(b) Arquivo classificado.

# O Problema da Ordenação

- Exemplo: ordenação por endereços







# O Problema da Ordenação

---

- Terminologia/conceitos
  - **Registros** a serem ordenados podem ser **complexos** ou **não**
    - Exemplos
      - Dados de empregados de uma empresa, sendo que a ordenação deve ser pelo RG do empregado
      - Números inteiros
  - Métodos de ordenação **independem** desse fator!



# O Problema da Ordenação

---

- Existem vários meios de implementar ordenação
- Dependendo do problema, um algoritmo apresenta **vantagens** e **desvantagens** sobre outro
- Como comparar?



# O Problema da Ordenação

---

- Devemos comparar as complexidades dos algoritmos
- Qual a **operação dominante**?



# O Problema da Ordenação

---

- Devemos comparar as complexidades dos algoritmos
- Qual a **operação dominante**?
  - **Número de comparações** entre elementos, na maioria dos casos
    - Somente as comparações que podem resultar em trocas



# Algoritmos de Ordenação

---

- Tradicionalmente, nos estudos dos métodos de ordenação, assume-se que a entrada dos algoritmos é um **vetor de números inteiros**
  - Procura-se **ordem crescente**



# Algoritmos de Ordenação Baseados em Troca

---

- Mais conhecidos algoritmos baseados em troca
  - **Bubble-sort**, também chamado método da bolha
  - **Quick-sort**, ou ordenação rápida ou, ainda, ordenação por troca de partição



# Bubble-sort

---

- É um dos métodos mais conhecidos e intuitivos
- Idéia básica
  - Percorrer o vetor várias vezes
  - A cada iteração, **comparar cada elemento com seu sucessor** (vetor[i] com vetor[i+1]) e trocá-los de lugar caso estejam na ordem incorreta



# Bubble-sort: um passo

---

- $X = (25, 57, 48, 37, 12, 92, 86, 33)$ 
  - $X[0]$  com  $X[1]$  (25 com 57) não ocorre permutação
  - $X[1]$  com  $X[2]$  (57 com 48) ocorre permutação
  - $X[2]$  com  $X[3]$  (57 com 37) ocorre permutação
  - $X[3]$  com  $X[4]$  (57 com 12) ocorre permutação
  - $X[4]$  com  $X[5]$  (57 com 92) não ocorre permutação
  - $X[5]$  com  $X[6]$  (92 com 86) ocorre permutação
  - $X[6]$  com  $X[7]$  (92 com 33) ocorre permutação





# Bubble-sort

---

- Depois do primeiro passo
  - vetor = (24 , 48 , 37 , 12 , 57 , 86 , 33 , 92)
  - O maior elemento (92) está na posição correta
- Para um vetor de  $n$  elementos, são necessárias  $n-1$  iterações
- A cada iteração, os elementos vão assumindo suas posições corretas
  - Por que se chama método das bolhas?



# Bubble-sort

---

- Exercício para entregar
  - Implementar bubble-sort
  - Calcular complexidade do algoritmo



# Implementação

---

```
for(int i=tamanho-1; i >= 1; i--) {  
    for( int j=0; j < i ; j++) {  
        if(vetor[j]>vetor[j+1]) {  
            aux = vetor[j];  
            vetor[j] = vetor[j+1];           vetor[j+1] = aux;  
        }  
    }  
}
```



# Bubble-sort

---

- Que melhorias podem ser feitas?
- passo 0 (vetor original)    25   57   48   37   12   92   86   33
- passo 1                    25   48   37   12   57   86   33   92
- passo 2                    25   37   12   48   57   33   86   92
- passo 3                    25   12   37   48   33   57   86   92
- passo 4                    12   25   37   33   48   57   86   92
- **passo 5                    12   25   33   37   48   57   86   92**
- passo 6                    12   25   33   37   48   57   86   92
- passo 7                    12   25   33   37   48   57   86   92



# Bubble-sort aprimorado

---

- Detectar quando o vetor já está ordenado
  - Isso ocorre quando, em um determinado passo, nenhuma troca é realizada
- Após o passo  $j$ , garante-se que o elemento  $\text{vetor}[n-j]$  está em sua posição correta
  - Para um vetor de  $n$  elementos são necessárias  $n-j$  iterações



# Bubble-sort: exercício

---

- Implementação do bubble-sort aprimorado



# BB Aprimorado

---

```
troca=1;
for (i=0; (i<n-1) && troca; i++) {
    troca=0;
    for (j=0; j<n-i-1; j++)
        if (v[j]>v[j+1]) {
            troca=1;
            aux=v[j];
            v[j]=v[j+1];
            v[j+1]=aux;
        }
    }
}
```



# Bubble-sort aprimorado

---

- Número de comparações na iteração  $j$  é  $n-j$ :
  - $(n-1) + (n-2) + (n-3) + \dots (n-k) = (2kn - k^2 - k) / 2$
  - Número de iterações para  $k=n$ :  $(2kn - k^2 - k) / 2 = (2n^2 - n^2 - n)/2 = \frac{1}{2}(n^2 - n) = O(n^2)$ , pior caso
- E se o vetor já estiver ordenado?
- E a complexidade de espaço?





# Bubble-sort aprimorado

---

- Número de comparações na iteração  $j$  é  $n-j$ :
  - $(n-1) + (n-2) + (n-3) + \dots (n-k) = (2kn - k^2 - k) / 2$
  - Número de iterações para  $k=n$ :  $(2kn - k^2 - k) / 2 = (2n^2 - n^2 - n)/2 = \frac{1}{2}(n^2 - n) = O(n^2)$ , pior caso
- E se o vetor já estiver ordenado?  $O(n)$ , para melhor caso
- E a complexidade de espaço?  $O(n)$



# Exercício

---

- Implementar o bubble sort recursivo



# Implementação

---

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



# Quick-sort

---

- Melhoramento do bubble-sort
  - Troca de elementos distantes são mais efetivas
- Idéia básica: **dividir para conquistar**
  - Dividir o vetor em dois vetores menores que serão ordenados independentemente e combinados para produzir o resultado final



# Quick-sort

---

- Considere um vetor  $v$  de  $n$  posições
- Primeiro passo
  - Elemento pivô:  $x$  (escolha do pivô é importantíssima)
    - Colocar  $x$  em sua posição correta
    - Ordenar de forma que os elementos à esquerda do pivô são menores ou iguais a ele e os elementos à direita são maiores ou iguais a ele
      - Percorrer o vetor  $v$  da esquerda para a direita até  $v[i] \geq x$ ; e da direita para a esquerda até  $v[j] \leq x$ 
        - Troca  $v[i]$  com  $v[j]$
      - Quando  $i$  e  $j$  se cruzarem, a iteração finaliza, de forma que  $v[0] \dots v[j]$  são menores ou iguais a  $x$  e  $v[i] \dots v[n-1]$  são maiores ou iguais a  $x$
- Segundo passo
  - Ordenar sub-vetores abaixo e acima do elemento pivô



# Quick-sort: exemplo

---

25 57 35 37 12 86 92 33



# Quick-sort: exemplo

---

$\text{pivô} = v[(0+7)/2] = 37$

25 57 35 37 12 86 92 33    ponteiros inicializados

$i \rightarrow$                        $\leftarrow j$



# Quick-sort: exemplo

---

$$\text{pivô} = v[(0+7)/2] = 37$$

25 57 35 37 12 86 92 33    ponteiros inicializados

i  
→

j  
←

25 57 35 37 12 86 92 33    procura-se  $i \geq \text{pivô}$

i

j





# Quick-sort: exemplo

---

$$\text{pivô} = v[(0+7)/2] = 37$$

25 57 35 **37** 12 86 92 33    ponteiros inicializados

i  
→

←  
j

25 57 35 **37** 12 86 92 33    procura-se  $i \geq \text{pivô}$

i

j

25 57 35 **37** 12 86 92 33    procura-se  $j \leq \text{pivô}$

i

j



# Quick-sort: exemplo

$\text{pivô} = v[(0+7)/2] = 37$

25 57 35 37 12 86 92 33    ponteiros inicializados

i  
→

←  
j

25 57 35 37 12 86 92 33    procura-se  $i \geq \text{pivô}$

i

j

25 57 35 37 12 86 92 33    procura-se  $j \leq \text{pivô}$

i

j

25 33 35 37 12 86 92 57    \*\*\*troca\*\*\*

i

j



# Quick-sort: exemplo

$\text{pivô} = v[(0+7)/2] = 37$

25 57 35 **37** 12 86 92 33    ponteiros inicializados

i  
↓

j  
↑

25 57 35 **37** 12 86 92 33    procura-se  $i \geq \text{pivô}$

i

j

25 57 35 **37** 12 86 92 33    procura-se  $j \leq \text{pivô}$

i

j

25 33 35 **37** 12 86 92 57    \*\*\*troca\*\*\*

i

j

25 33 35 **37** 12 86 92 57    procura-se  $i \geq \text{pivô}$

i

j



# Quick-sort: exemplo

$$\text{pivô} = v[(0+7)/2] = 37$$

25 57 35 37 12 86 92 33    ponteiros inicializados



25 57 35 37 12 86 92 33    procura-se  $i \geq \text{pivô}$

i

j

25 57 35 37 12 86 92 33    procura-se  $j \leq \text{pivô}$

i

j

25 33 35 37 12 86 92 57    \*\*\*troca\*\*\*

i

j

25 33 35 37 12 86 92 57    procura-se  $i \geq \text{pivô}$

i

j

25 33 35 37 12 86 92 57    procura-se  $j \leq \text{pivô}$

i

j



# ponteiros inicializados

```
procura-se i >= pivô
```

```
procura-se j<=pivô
```

\*\*\*troca\*\*\*

```
procura-se i >= pivô
```

procura-se  $j \leq \text{pivô}$

\*\*\*troca\*\*\*



# Quick-sort: exemplo

---

$$\text{pivô} = v[(0+7)/2] = 37$$

25 33 35 12 **37** 86 92 57    procura-se  $i \geq \text{pivô}$   
          ij



# Quick-sort: exemplo

---

$$\text{pivô} = v[(0+7)/2] = 37$$

25 33 35 12 **37** 86 92 57    procura-se  $i \geq \text{pivô}$   
                                 ij

25 33 35 12 **37** 86 92 57    procura-se  $j \leq \text{pivô}$   
                                 ij  
                                 → como i e j se cruzaram, fim do processo



# Quick-sort: exemplo

---

$$\text{pivô} = v[(0+7)/2] = 37$$

25 33 35 12 **37** 86 92 57    procura-se  $i \geq \text{pivô}$   
                  *ij*

25 33 35 12 **37** 86 92 57    procura-se  $j \leq \text{pivô}$   
                  *ij*                     $\rightarrow$  como  $i$  e  $j$  se cruzaram, fim do processo

Todos à esquerda do pivô são menores ou iguais a ele  
 $\rightarrow v[0] \dots v[j] \leq \text{pivô}$

Todos à direita do pivô são maiores ou iguais a ele  
 $\rightarrow v[i] \dots v[n-1] \geq \text{pivô}$





# Quick-sort: exercício

---

- Fazer as ordenações dos subvetores, repetindo o processo

25 33 35 12 (37) 86 92 57



# Quick-sort: exercício

---

- Exercício para entregar
  - Implementar quicksort
  - Calcular complexidade do pior e melhor caso