



# Métodos de Ordenação

## Parte 1

---

**ICC2**

**Prof Alneu de Andrade Lopes**

**Prof Thiago Pardo**



# 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 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.

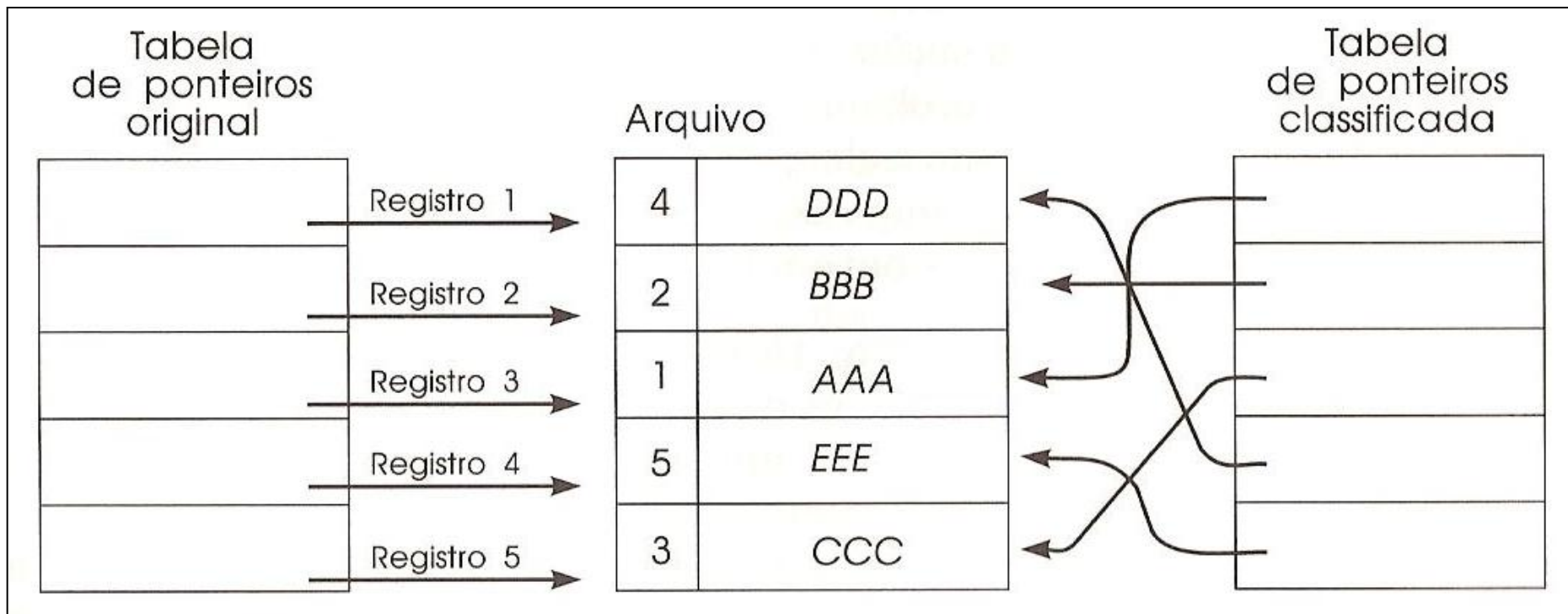
1	AAA
2	BBB
3	CCC
4	DDD
5	EEE

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**?
  - 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 ( $\text{vetor}[i]$  com  $\text{vetor}[i+1]$ ) 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 em grupos de 4 (valendo nota)
  - Implementar bubble-sort
  - Calcular complexidade do algoritmo



# 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
■ <b>passo 5</b>	<b>12</b>	<b>25</b>	<b>33</b>	<b>37</b>	<b>48</b>	<b>57</b>	<b>86</b>	<b>92</b>
■ 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
- O elemento  $\text{vetor}[n-j]$  estará na sua posição após o passo  $j$ 
  - Para um vetor de  $n$  elementos são necessárias  $n-j$  iterações



# Bubble-sort aprimorado

---

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



# Bubble-sort aprimorado

---

- Num. de comparações na iteração  $j$  é  $n - j$ :
  - $(n-1) + (n-2) + (n-3) + \dots (n-k) = (2kn - k^2 - k) / 2$
  - Número médio de iterações ( $k$ ) é  $O(n)$ :  $(2kn - k^2 - k) / 2 = (2n^2 - n^2 - n) / 2 = \frac{1}{2}(n^2 - n) = O(n^2)$
- $T(n)$  continua  $O(n^2)$ , mas a constante multiplicativa é menor
- E se o vetor já estiver ordenado?
- E a complexidade de espaço?



# Bubble-sort aprimorado

---

- Num. de comparações na iteração  $j$  é  $n - j$ :
  - $(n-1) + (n-2) + (n-3) + \dots (n-k) = (2kn - k^2 - k) / 2$
  - Número médio de iterações ( $k$ ) é  $O(n)$ :  $(2kn - k^2 - k) / 2 = (2n^2 - n^2 - n) / 2 = \frac{1}{2}(n^2 - n) = O(n^2)$
- $T(n)$  continua  $O(n^2)$ , mas a constante multiplicativa é menor
- E se o vetor já estiver ordenado?  $O(n)$
- E a complexidade de espaço?  $O(n)$



# 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

---

- Primeiro passo

- Elemento pivô:  $v$

- Colocar  $v$  em sua posição correta
    - Particionar de forma que os elementos à esquerda do pivô sejam menores que o mesmo e os elementos à direita maiores
    - Percorrer o vetor  $X$  da esquerda para a direita até  $X[i] > v$ ; e da direita para a esquerda até  $X[j] < v$ 
      - Troca  $X[i]$  com  $X[j]$
    - Quando  $i$  e  $j$  cruzarem, a iteração finaliza e  $v$  troca de lugar com  $i$

- Segundo passo

- Ordenar sub-vetores abaixo e acima do elemento pivô





# Quick-sort

25 57 48 37 12 86 92 33



# Quick-sort

**Pivô = 25**

→  
down  
25   57   48   37   12   86   92   ←  
up  
33   ponteiros inicializados



# Quick-sort

**Pivô = 25**

→  
down  
25 57 48 37 12 86 92 33

←  
up  
33

ponteiros inicializados

down

25 57 48 37 12 86 92

up

33

procura-se down > que pivô



# Quick-sort

**Pivô = 25**

→							←	
down							up	
25	57	48	37	12	86	92	33	ponteiros inicializados
	down						up	
25	57	48	37	12	86	92	33	procura-se down > que pivô
	down			up				
25	57	48	37	12	86	92	33	procura-se up < que pivô



# Quick-sort

**Pivô = 25**

↓							↑		
down	25	57	48	37	12	86	92	33	ponteiros inicializados
down	25	57	48	37	12	86	92	33	procura-se down > que pivô
down	25	57	48	37	12	86	92	33	procura-se up < que pivô
down	25	12	48	37	57	86	92	33	<b>*troca*</b>



# Quick-sort

**Pivô = 25**

→							←				
down	25	57	48	37	12	86	92	33	ponteiros inicializados		
	down	25	57	48	37	12	86	92	33	procura-se down > que pivô	
	down	25	57	48	37	12	86	92	33	procura-se up < que pivô	
	down	25	12	48	37	57	86	92	33	<b>*troca*</b>	
		down	25	12	48	37	57	86	92	33	procura-se down > que pivô



# Quick-sort

**Pivô = 25**

→ down	25	57	48	37	12	86	92	← up	33	ponteiros inicializados
down	25	57	48	37	12	86	92	up	33	procura-se down > que pivô
down	25	57	48	37	12	86	92	up	33	procura-se up < que pivô
down	25	12	48	37	57	86	92	up	33	<b>*troca*</b>
down	25	12	48	37	57	86	92	up	33	procura-se down > que pivô
up	25	12	48	37	57	86	92	down	33	procura-se up < que pivô



# Quick-sort

**Pivô = 25**

→ down	25	57	48	37	12	86	92	← up	33	ponteiros inicializados
down	25	57	48	37	12	86	92	up	33	procura-se down > que pivô
down	25	57	48	37	12	86	92	up	33	procura-se up < que pivô
down	25	12	48	37	57	86	92	up	33	<b>*troca*</b>
down	25	12	48	37	57	86	92	up	33	procura-se down > que pivô
up	25	12	48	37	57	86	92	up	33	procura-se up < que pivô
up	12	25	48	37	57	86	92	up	33	<b>*troca*</b> (down e up cruzam)





# Quick-sort

---

- Todo elemento à esquerda de 25 é  $\leq 25$
- Todo elemento à direita de 25 é  $\geq 25$
- Ordenar os dois subvetores (12) e (48 37 57 86 92 33)



# Quick-sort

---

**Exercício: fazer a ordenação do vetor abaixo**

- 25 57 48 37 12 86 92 33
- (12) 25 (48 37 57 86 92 33)
- 12 25 (48 37 57 86 92 33)
- 12 25 (33 37) 48 (86 92 57)
- 12 25 33 (37) 48 (86 92 57)
- 12 25 33 37 48 (86 92 57)
- 12 25 33 37 48 (57) 86 (92)
- 12 25 33 37 48 57 86 (92)
- 12 25 33 37 48 57 86 92



# Quick-sort: algoritmo

---

```
1 void Quicksort (int X[], int p, int r)
2 {
3     if (p < r) {
4         q = Partição(X, p, r);
5         Quicksort(X, p, q-1);
6         Quicksort(X, q+1, r);
7     }
8 }
```

```
1 int Partição (int X[], int p, int r) {
2     x = X[p];   up = r;   down = p;
3     while (down < up) {
4         while (X[down] <= x) {
5             down = down+1;
6         }
7         while(X[up] > x) {
8             up = up - 1;
9         }
10        if (down < up)
11            troca(X[down], X[up]);
12    }
13    X[p]= X[up];
14    X[up] = x;
15    return(up);
16 }
```



# Quick-sort

---

- Custo com pivô cujo posição correta (final) seja o meio do vetor
  - O vetor de tamanho  $n$  é dividido ao meio, cada metade é dividida ao meio, ...,  $m$  vezes  $\Rightarrow n = 2^m$ , logo,  $m = \log_2 n$ .
  - Cada parte do vetor realiza  $n$  comparações (com  $n$  = partição atual do vetor)
  - Cada metade =  $n * m$
  - Custo do vetor =  $2 * (n * m) = O(n \log_2 n)$



# Quick-sort

---

- Custo com vetor ordenado
  - Cada partição produz um sub-vetor com 0 elementos e outro com  $n-1$  elementos
    - A sub-rotina Partição será chamada  $n$  vezes
    - Cada partição fará  $n$  comparações
    - Custo =  $O(n^2)$
  - Mesmo custo do Bubble-sort