

Métodos de Ordenação

Parte 1

SCC-201 Introdução à Ciência da Computação II

Rosane Minghim
2010

Baseado no material dos Professores Rudinei Goularte e Thiago Pardo

1

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

2

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?

3

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!

4

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!

9

O Problema da Ordenação

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

10

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

11

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**

12

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

13

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

14

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

15

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?

16

Bubble-sort

Exercício

- Implementar bubble-sort
- Calcular complexidade do algoritmo

17

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

18

Bubble-sort aprimorado

- Detectar quando o vetor já está ordenado
 - Isso ocorre quando, em um determinado passo, nenhuma troca é realizada
- O elemento vetor[n-j] estará na sua posição após o passo j
 - Para um vetor de n elementos são necessárias n-j iterações

19

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;
        }
}
```

20

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?

21

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? $O(n)$

22

Quick-sort


- Melhoria 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

23


Quick-sort

- Primeiro passo
 - Elemento pivô: v
 - Colocar v em sua posição correta
 - Ordenar de forma que os elementos à esquerda do pivô são menores que o mesmo e os elementos à direita são 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

24


 Quick-sort

25 57 48 37 12 86 92 33

 Quick-sort


Pivô = 25

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

 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

 Quick-sort

Pivô = 25

→ down
 25 57 48 37 12 86 92 33 ponteiros inicializados
↓ down
↓ down ↑ up
 25 57 48 37 12 86 92 33 procura-se up < 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
	down							up		
25	12	48	37	57	86	92	33			*troca*

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
	down							up		
25	12	48	37	57	86	92	33			*troca*
		down		up						
25	12	48	37	57	86	92	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
	down							up		
25	12	48	37	57	86	92	33			*troca*
		down		up						
25	12	48	37	57	86	92	33			procura-se down > que pivô
	up	down								
25	12	48	37	57	86	92	33			procura-se up < 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
	down							up		
25	12	48	37	57	86	92	33			*troca*
		down		up						
25	12	48	37	57	86	92	33			procura-se down > que pivô
	up	down								
25	12	48	37	57	86	92	33			procura-se up < que pivô
		up	down							
12	25	48	37	57	86	92	33			*troca* (down e up cruzam)

Quick-sort

- Todo elemento à esquerda de 25 é ≤ 25
- Todo elemento à direita de 25 é ≥ 25
- Ordenar os dois subvetores (12) e (48 37 57 86 92 33)

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

34

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 }
9
10 int Partição (int X[], int p, int r) {
11     x = X[p]; up = r; down = p;
12     while (down < up) {
13         while (X[down] <= x) {
14             down = down+1;
15         }
16         while (X[up] > x) {
17             up = up - 1;
18         }
19         if (down < up)
20             troca(X[down], X[up]);
21     }
22     X[p] = X[up];
23     X[up] = x;
24     return(up);
25 }
```

35

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 = $2 * (n * m) = O(n \log_2 n)$

36



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