

Estruturas Avançadas de Dados I

(Métodos de Classificação)

Prof. Gilberto Irajá Müller

Introdução

- Processo de organizar itens em ordem (de)crescente, segundo algum critério;
- Também chamado de ordenação. As ordens mais usadas são a numérica e a lexicográfica (quando ordenamos palavras ou textos).
- Aplicações
 - Preparação de dados para facilitar pesquisas futuras
 - Exemplo: dicionários e listas telefônicas;
 - Agrupar itens que apresentam mesmos valores
 - Para eliminação de elementos repetidos.
- Ordem numérica: 1 2 3 4 5 6
- Ordem lexicográfica: A... B... C... D... E... F...
- Mas podemos ter outras ordens. Ex.: 2 3 5 7 4 9 6

Introdução

- Sejam R_1, R_2, \dots, R_N , N itens (chamados registros);
- Cada registro R_i é formado por uma chave C_i e por informações ditas satélites;
- A ordenação dos registros é feita definindo-se uma relação de ordem "<" sobre os valores das **chaves**;
- O objetivo da ordenação é determinar uma permutação dos índices $1 \leq i_1, i_2, \dots, i_N \leq N$ das chaves, tal que $C_{i1} \leq C_{i2} \leq \dots \leq C_{iN}$.
- Um conjunto de registros é chamado de arquivo.

Introdução

- Uma relação de ordem " $<$ " (leia-se precede) deve satisfazer as seguintes condições para quaisquer valores a , b e c :
 - (i) Uma e somente uma das seguintes possibilidades é verdadeira: $a < b$, $a = b$ ou $b < a$ (lei da tricotomia)
 - (ii) Se $a < b$ e $b < c$, então $a < c$ (transitividade)
- As propriedades (i) e (ii) definem o conceito de ordem linear ou ordem total.

Classificação Quanto à Estabilidade

- **Métodos Instáveis:** a ordem relativa dos itens com chaves iguais é alterada durante o processo de ordenação;
- **Métodos Estáveis:** se a ordem relativa dos itens com chaves iguais mantém-se inalterada durante o processo
 - Um algoritmo somente é estável se:
 - $i < j$ e $a[i] == a[j]$, implica que $p(i) < p(j)$. Onde p é o movimento de permutação (move $a[i]$ para a posição $p[i]$).
- Alguns dos métodos de ordenação mais eficientes não são estáveis.

Classificação Quanto à Estabilidade – Exemplo Ordenação Dinheiro

1	2	3	4	5	6	7	8	9
R\$ 100,00	R\$ 100,00	R\$ 200,00	R\$ 400,00	R\$ 500,00	R\$ 600,00	R\$ 600,00	R\$ 500,00	R\$ 400,00

Estável:

1	2	3	4	9	5	8	6	7
R\$ 100,00	R\$ 100,00	R\$ 200,00	R\$ 400,00	R\$ 400,00	R\$ 500,00	R\$ 500,00	R\$ 600,00	R\$ 600,00

Instável:

2	1	3	9	4	5	8	7	6
R\$ 100,00	R\$ 100,00	R\$ 200,00	R\$ 400,00	R\$ 400,00	R\$ 500,00	R\$ 500,00	R\$ 600,00	R\$ 600,00

Classificação Quanto ao Conjunto Registros

- **Ordenação Interna:** o conjunto de registros cabe todo na memória principal.
- **Ordenação Externa:** o conjunto de registros não cabe completamente em memória principal e deve ser armazenado em disco ou fita.
 - Alguns autores utilizam ordenação de arrays (ordenação interna) e ordenação de registros (ordenação externa);
 - Principal diferença: na ordenação interna o registro pode ser acessado diretamente, enquanto na ordenação externa, os registros são acessados sequencialmente ou em blocos.

Ordenação Interna

- Medidas de complexidade levam em conta:
 - O número de comparação entre as chaves;
 - O número de trocas entre os itens.
- São classificados em dois tipos:
 - **Métodos Simples (elementares):** mais recomendados para conjuntos pequenos de dados. Usam mais comparações, mas produzem códigos menores e mais simples;
 - **Métodos Eficientes (sofisticados):** adequados para conjuntos maiores de dados. Usam menos comparações, porém, produzem códigos mais complexos e com muitos detalhes.

Alguns Exemplos de Ordenação Interna

- Métodos Simples
 - Ex.: Bubble Sort, Insertion Sort e Selection Sort;
 - Adequados para um volume pequeno de dados;
 - Normalmente, requerem $O(n^2)$ comparações;
 - Porém, são simples e produzem pequenos programas.
- Métodos Eficientes
 - Ex.: Mergesort, Quicksort e Heapsort;
 - Adequados para grandes volumes de dados;
 - Normalmente, requerem $O(n \log(n))$ comparações;
 - Usam menos comparações. Porém, são mais complexos;
 - Métodos sofisticados possuem mais eficiência.

Formas de Representação do Resultado

- Reorganização Física
- Encadeamento
- Vetor Indireto de Ordenação (VIO)

Reorganização Física

	chave	
1	10	
2	19	
3	13	
4	12	
5	7	

(a) antes da classificação

	chave	
1	7	
2	10	
3	12	
4	13	
5	19	

(b) após a classificação

Encadeamento

cabeça da lista

5

chave

1	10	
2	19	
3	13	
4	12	
5	7	

(a) antes da classificação

chave

1	10	4
2	19	0
3	13	2
4	12	3
5	7	1

(b) após a classificação

Vetor Indireto de Ordenação

Chave

1	10	
2	19	
3	13	
4	12	
5	7	

VIO

1	5
2	1
3	4
4	3
5	2

Outras Classificações

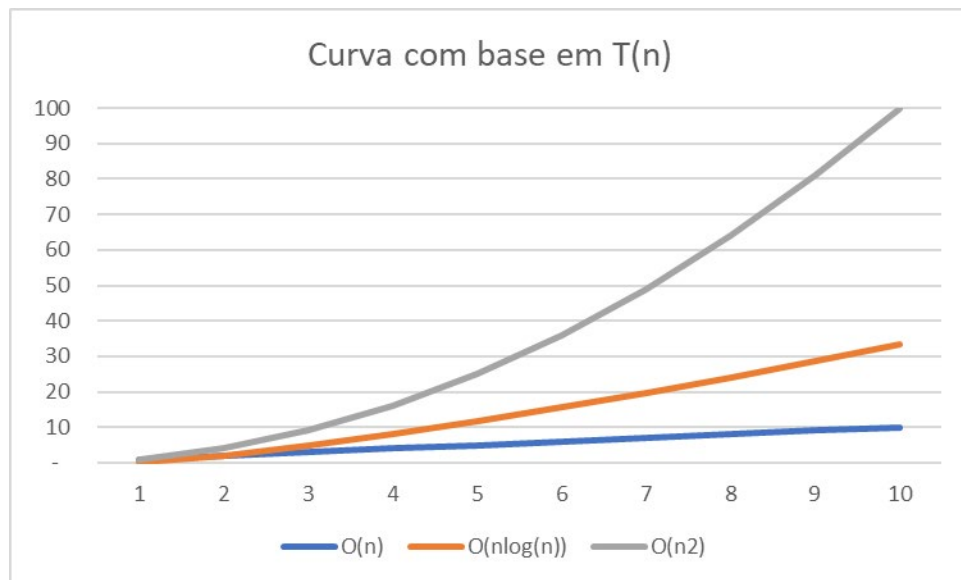
- **Complexidade computacional:** usa-se o pior caso como base. Um algoritmo de ordenação com bom desempenho está na ordem $O(n \log(n))$ e com desempenho ruim está em $O(n^2)$;
- **Complexidade de espaço:** há dois tipos de padrões: (a) “in-place”, algoritmos que utilizam tamanho constante de memória para o processo de ordenação $O(1)$; (b) “out-place”, tamanho de memória adicional de acordo com a entrada (Ex.: array auxiliar).

Outras Classificações (cont.)

- **Recursivo e não recursivo:** normalmente, o problema pode ser resolvido utilizando soluções para variantes menores do mesmo problema (divisão e conquista). Quicksort e Mergesort são considerados recursivos e Insertion e Selection Sort são considerados não recursivos;
- **Baseado em Comparação:** são aqueles algoritmos que comparam uma chave com a outra de forma a realizar o processo de permutação. Ex.: Insertion Sort, HeapSort, entre outros. Há métodos, porém, que ordenam sem a necessidade de comparação como o caso do Radix Sort, Bucket Sort, Counting Sort, entre outros.

Antes de Analisarmos os Algoritmos

$T(n)$	$O(n)$	$O(n\log(n))$	$O(n^2)$
1	1	-	1
2	2	2	4
3	3	5	9
4	4	8	16
5	5	12	25
6	6	16	36
7	7	20	49
8	8	24	64
9	9	29	81
10	10	33	100



- Três casos (pior, médio e melhor) serão analisados;
- Quatro classes serão utilizadas: $O(1)$ constante, $O(n)$ = linear, $O(n\log(n))$ = linearítmico e $O(n^2)$ = quadrática.

Bubble Sort

- Por que Bolha?
 - Se o array a ser ordenado for colocado na vertical, com $\text{Item}[n]$ em cima e $\text{Item}[1]$ embaixo, durante cada passo o menor elemento “sobe” até encontrar um elemento maior ainda, como se uma bolha subisse dentro de um tubo de acordo com sua densidade.
- **Funcionamento:** Neste algoritmo são efetuadas comparações entre os dados armazenados em um array de tamanho “n”. Cada elemento de posição “i” será comparado com o elemento de posição $i + 1$, e quando a ordenação procurada é encontrada, uma troca de posições entre os elementos é feita. A execução finalizará quando não ocorrer mais trocas.

Bubble Sort (cont.)

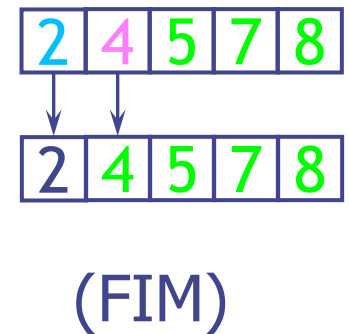
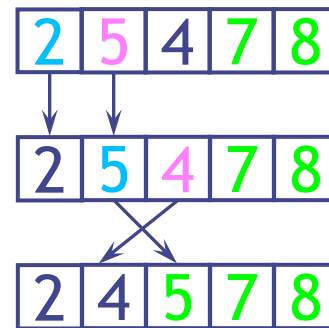
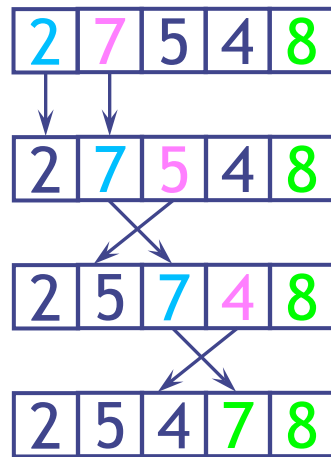
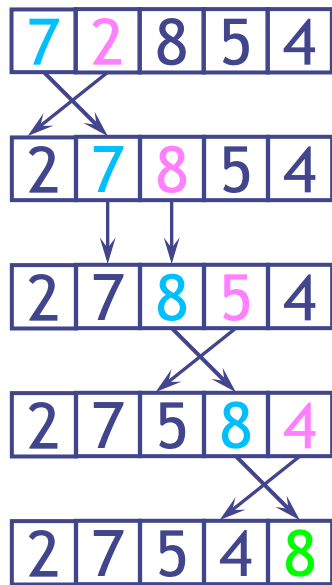
```
public static <T extends Comparable<? super T>> void bubbleSort(T[] a) {
    boolean exchange;
    do {
        exchange = false;
        for (int i = 0; i < a.length - 1; i++) {
            if (a[i].compareTo(a[i + 1]) > 0) {
                exchange(a, i, i + 1);
                exchange = true;
            }
        }
    } while (exchange);
}
```

```
private static <T extends Comparable<? super T>> void exchange(T[] a, int i, int j) {
    T tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
}
```

<https://www.toptal.com/developers/sorting-algorithms>

<http://www.caseyrule.com/projects/sounds-of-sorting>

Bubble Sort (cont.)



Bubble Sort (cont.)

- ***Melhor caso:*** → **$O(n)$**
 - Array está em ordem crescente;
 - Número de trocas: 0 → $O(1)$
 - Número de comparações: $(n - 1)$ → $O(n)$
- ***Pior caso:*** → **$O(n^2)$**
 - Array está em ordem decrescente;
 - Primeiro laço executará $n - 1$ vezes,
 - Número de trocas: → $O(n^2)$
 - Número de comparações: → $O(n^2)$
- **Caso médio:** → **$O(n^2)$**

A complexidade do Bubble Sort é $O(n^2)$

Insertion Sort

- A classificação é obtida porque os elementos são inseridos na sua posição correta;
- Normalmente utilizado para um conjunto pequeno de dados, pois apresenta baixa eficiência;
- Divide-se o array em 2 segmentos:
 - o primeiro contendo os elementos já ordenados;
 - o segundo contendo os elementos ainda não ordenados.
- **Funcionamento:** Utiliza o primeiro elemento do segmento não ordenado e procura seu lugar no segmento ordenado.
- No início: o 1º segmento terá apenas 1 elemento.

Insertion Sort (cont.)

```
public static <T extends Comparable<? super T>> void insertionSort(T[] a) {  
    for (int i = 1; i < a.length; i++) {  
        for (int j = i; j > 0 && a[j - 1].compareTo(a[j]) > 0; j--) {  
            exchange(a, j - 1, j);  
        }  
    }  
}
```

Insertion Sort (cont.)

Chave 8 está na posição

2	7	8	5	4
---	---	---	---	---

Segmento Ordenado

7	2	8	5	4
---	---	---	---	---

Segmento **não** Ordenado

Variável tmp

2	7	8	5	4
---	---	---	---	---

Chave 5

2		7	8	4
---	--	---	---	---

2	5	7	8	4
---	---	---	---	---

Variável tmp

2	5	7	8	4
---	---	---	---	---

Chave 4

2		5	7	8
---	--	---	---	---

Segmento Ordenado

2	4	5	7	8
---	---	---	---	---

(FIM)

7	2	8	5	4
---	---	---	---	---

Variável tmp

Desloca os elementos até encontrar a posição para o 2

	7	8	5	4
--	---	---	---	---

Segmento Ordenado

2	7	8	5	4
---	---	---	---	---

Segmento não Ordenado

Chave 2

Insertion Sort (cont.)

- O pior caso ocorre quando o array possui os elementos na ordem decrescente. Neste caso, cada elemento é comparado com cada elemento no subarray $a[0..j - 1]$, e então executa-se a linha das comparações. Logo, o tempo de execução, que é o número de comparações, é dado pela fórmula:

$$T(n) = 2 + 3 + 4 + \dots + n$$

$$T(n) = \left(\sum_{i=1}^n i \right) - 1$$

$$T(n) = \frac{(1+n)n}{2} - 1$$

$$T(n) = \frac{n^2 + n}{2} - 1$$

$$T(n) = O(n^2)$$

Insertion Sort (cont.)

- No melhor caso ocorre quando o array possui os elementos ordenados crescente.

$$T(n) = O(n)$$

- Para cada $j = 0, 1, \dots, n - 2$, tem-se que a condição $a[j - 1] > \text{tmp}$ é falsa. Então, o custo de executar é 1 para cada valor de j .
- No caso médio $O(n^2)$.

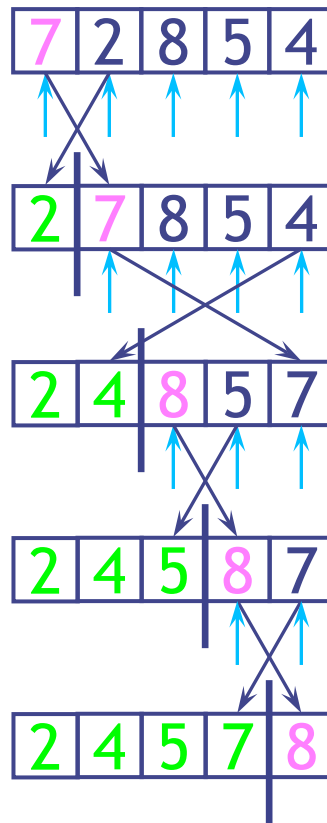
Selection Sort

- O array é dividido em duas listas (ordenada e não ordenada), através de uma “parede imaginária”;
- Busca-se o menor elemento da lista não ordenada trocando-o com o primeiro elemento da lista não ordenada;
- Depois de cada troca, a parede imaginária aumenta em uma posição, incrementando o número de elementos ordenados e diminuindo o número de elementos não ordenados.

Selection Sort (cont.)

```
public static <T extends Comparable<? super T>> void selectionSort(T[] a) {  
    for (int min, i = 0; i < a.length; i++) {  
        min = i;  
        for (int j = i + 1; j < a.length; j++) {  
            if (a[j].compareTo(a[min]) < 0) {  
                min = j;  
            }  
        }  
        exchange(a, min, i);  
    }  
}
```

Selection Sort (cont.)



Selection Sort (cont.)

- O caso médio, pior e melhor têm o mesmo desempenho $O(n^2)$;
- Conclui-se que, ao usar tal método, não importará como os elementos do array estarão distribuídos;
- Útil somente para n pequeno;
- O interessante deste algoritmo está relacionado com o número de trocas $O(n)$. Então, em um aplicação que necessite pouca troca, ele seria útil.

Complexidade

Método	Caso médio	Melhor caso	Pior caso	Complexidade de Espaço	Estável	Interno	Recursivo	Comparação
Bubble Sort	$O(n^2)$	$O(n)$	$O(n^2)$	In-place = $O(1)$	Sim	Sim	Não	Sim
Insertion Sort	$O(n^2)$	$O(n)$	$O(n^2)$	In-place = $O(1)$	Sim	Sim	Não	Sim
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	In-place = $O(1)$	Sim	Sim	Não	Sim

Exercícios Teóricos

Exercício 1. Baseado no array abaixo, desenvolva o seguinte:

0	1	2	3	4
2	7	4	9	3

Somente as alterações no array:

- Passo-a-passo do Bubble Sort
- Passo-a-passo do Insertion Sort
- Passo-a-passo do Selection Sort

Exercícios Teóricos

- **Exercício 2.** Baseado no array abaixo (em negrito) e o processo de ordenação, qual algoritmo produziria tais permutações?

[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

[1, 9, 8, 7, 6, 5, 4, 3, 2, 10]

[1, 2, 8, 7, 6, 5, 4, 3, 9, 10]

[1, 2, 3, 7, 6, 5, 4, 8, 9, 10]

[1, 2, 3, 4, 6, 5, 7, 8, 9, 10]

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Exercícios Teóricos

- **Exercício 3.** Dado um array com elementos aleatórios, ao utilizar o algoritmo Selection Sort, seria executado em tempo linear, quadrático ou algo entre os dois anteriores?

Exercícios Teóricos

- **Exercício 4.** Qual seria o resultado produzido pelo algoritmo de ordenação abaixo. Mostre o estado do array somente quando existir uma troca.

[5, 4, 3, 2, 1]

```
public static <T extends Comparable<T>> void oddEvenSort(T[] a) {  
    for (int i = 0; i < a.length; i++) {  
        if (i % 2 != 0) {  
            for (int j = 2; j < a.length; j += 2) {  
                if (a[j].compareTo(a[j - 1]) < 0) {  
                    T tmp = a[j];  
                    a[j] = a[j - 1];  
                    a[j - 1] = tmp;  
                }  
            }  
        } else {  
            for (int j = 1; j < a.length; j += 2) {  
                if (a[j].compareTo(a[j - 1]) < 0) {  
                    T tmp = a[j];  
                    a[j] = a[j - 1];  
                    a[j - 1] = tmp;  
                }  
            }  
        }  
    }  
}
```

Referências Bibliográficas

- CORMEN, Thomas H. et al. **Introduction to algorithms**. 3. ed. Cambridge: MIT, 2009. xix. 1292 p.
- Algorithms 4a. Edição. <http://algs4.cs.princeton.edu/21elementary/>. Acessado em 11/10/2017.
- Quadro Complexidade. <http://bigocheatsheet.com/>. Acessado em 11/10/2017.
- LUZZARDI, P. R. G. **Lâminas sobre Trabalho da Disciplina de Estruturas de Dados**. UCPEL.