Estruturas de Dados Lineares

Métodos de Ordenação

Prof. Vinicius Bischoff

v.03/2024

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 $R_1, R_2, \ldots, R_N$, 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 $1 \leq i_1, i_2, \ldots, i_N \leq N$ das chaves, tal que $C_{i1} \leq C_{i2} \leq \ldots \leq C_{iN}$.
- 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(nlog(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(nlog(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 conquita). 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.

Métodos de Ordenação

- Por inserção
  - **Inserção Direta (*Insertion Sort*)**
- Por seleção
  - **Seleção Direta (*Selection Sort*)**
- Por troca
  - ***Bubble Sort***

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

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

Insertion Sort

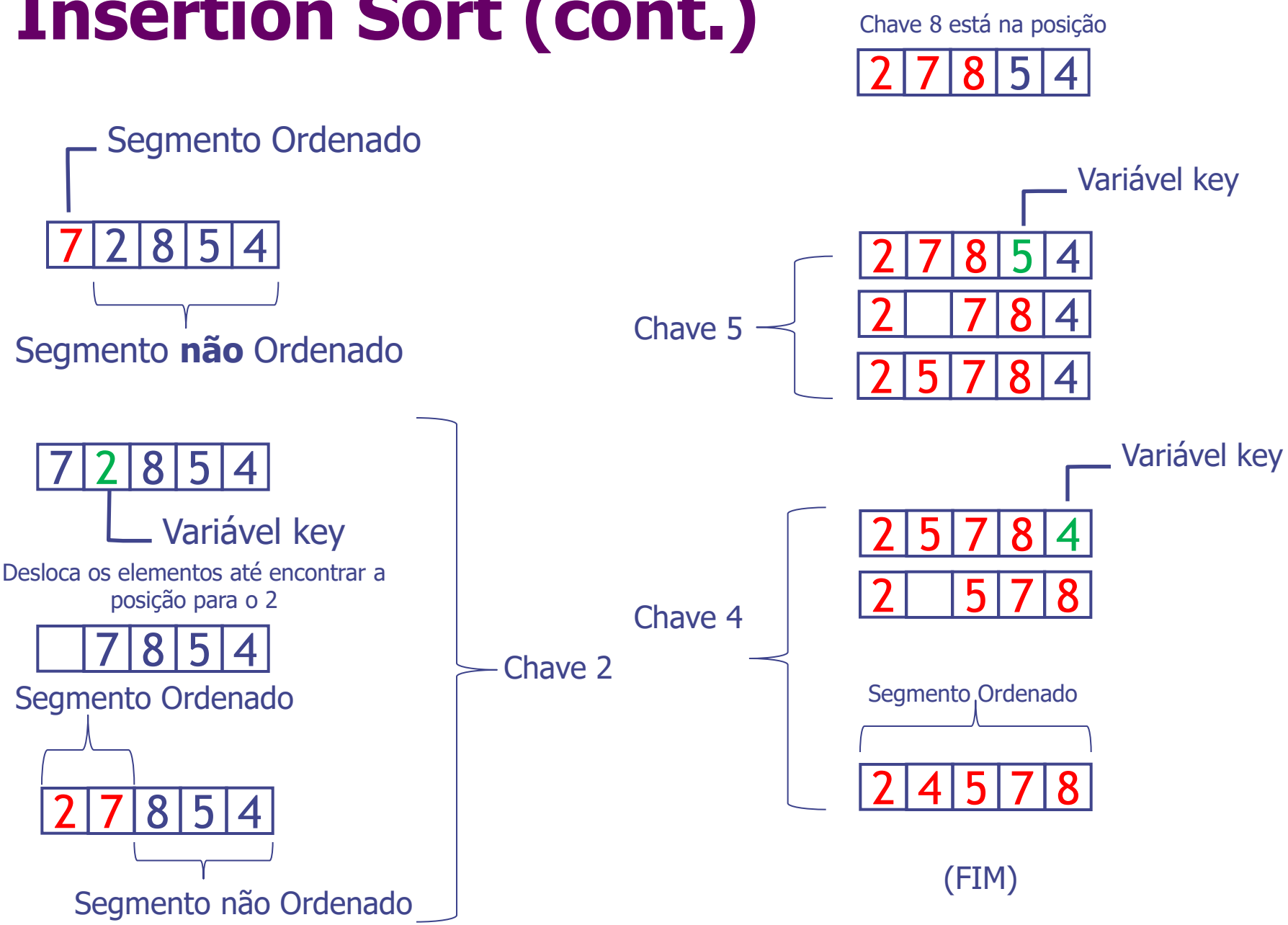
- A classificação é obtida porque os elementos são inseridos na sua posição correta;

- 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.)



Insertion Sort (cont.)

```
def insertion_sort(a: np.array) -> None:
    for i in range(1, len(a)):
        j: int = i # pos do 1º elemento no seg. não ord.
        key: any = a[i] # 1º elemento no seg. não ord.
        while j > 0 and a[j - 1] > key:
            a[j] = a[j - 1]
            j -= 1
        a[j] = key
```

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:

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

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

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

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

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

Insertion Sort (cont.)

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

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

- Para cada j = 0, 1, ..., n – 2, tem-se que a condição a[j - 1] > tmp é falsa. Então, o custo de executar é 1 para cada valor de j.

- No caso médio $O(n^2)$.

Selection Sort

- **Funcionamento**: 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.)

- Como funciona?

9	25	10	18	5	7	15	3

Iteração	Array	Chave	Permutação
1	9 25 10 18 5 7 15 3	3	9 e 3
2	3 25 10 18 5 7 15 9	5	25 e 5
3	3 5 10 18 25 7 15 9	7	10 e 7
4	3 5 7 18 25 10 15 9	9	18 e 9
5	3 5 7 9 25 10 15 18	10	25 e 10
6	3 5 7 9 10 25 15 18	15	25 e 15
7	3 5 7 9 10 15 25 18	18	25 e 18
8	3 5 7 9 10 15 18 25		

Selection Sort (cont.)

```
def selection_sort(a: np.array) -> None:
    for i in range(len(a) - 1):
      min: int = i # mínimo inicial
      for j in range(i + 1, len(a)):
         if a[j] < a[min]:
            min = j # acha o novo mínimo
      a[i], a[min] = a[min], a[i]
```

Selection Sort (cont.)

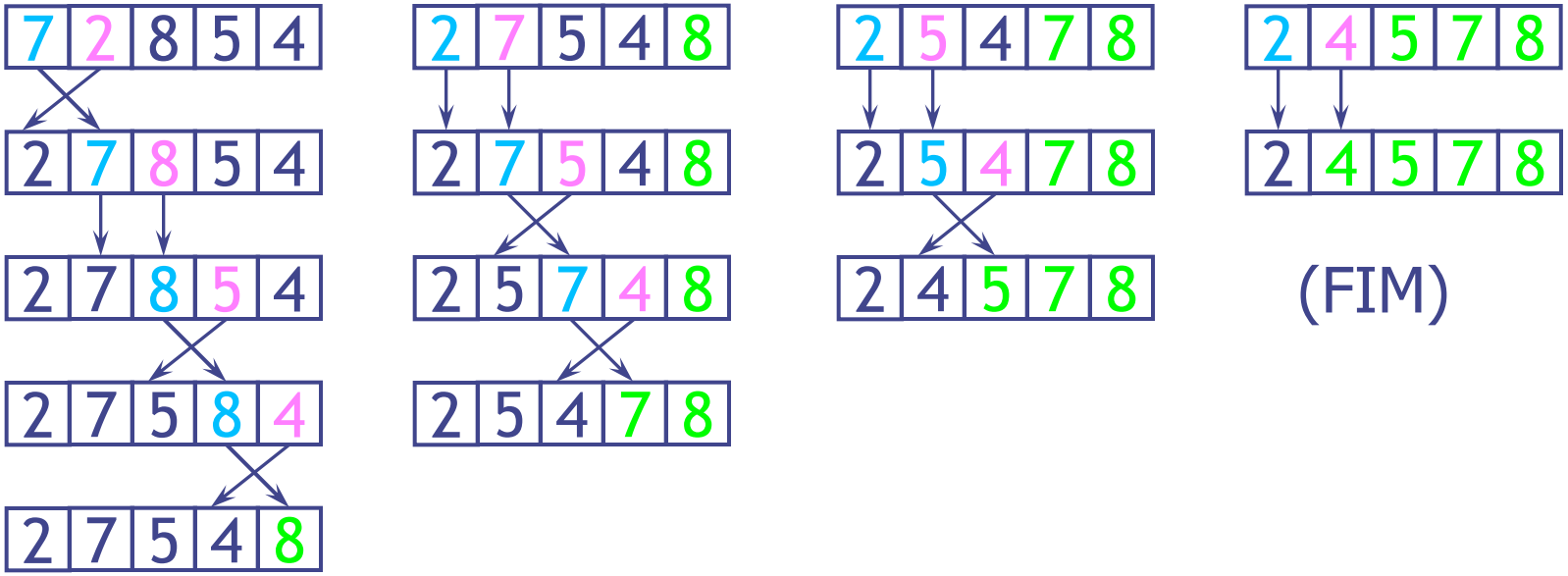
- O caso médio, pior e melhor têm o mesmo desempenho $O(n^2)$;
- Conclui-se que, ao usar tal método, não importará como os elementos do array estarão distribuídos;
- Útil somente para *n* pequeno;
- O interessante deste algoritmo está relacionado com o número de trocas O(n). Então, em um aplicação que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

Bubble Sort

- Por que Bolha?
  - Se o array a ser ordenado for colocado na vertical, com Item[n] em cima e Item[0] 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.)



Bubble Sort (cont.)

```
def bubble_sort(a: np.array) -> None:
    exchange: bool = True
    while exchange:
        exchange = False
        for i in range(len(a) - 1):
            if a[i] > a[i + 1]:
                a[i], a[i + 1] = a[i + 1], a[i]
                exchange = True
```

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)

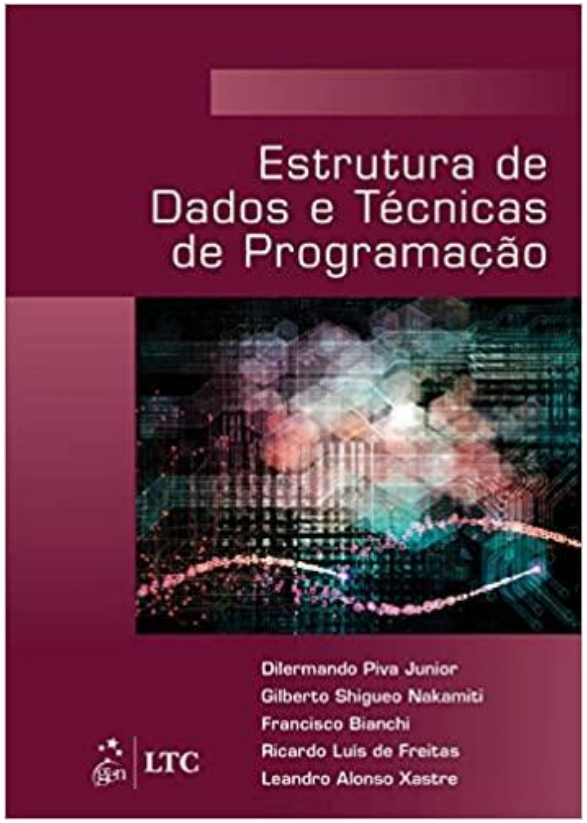
- **Caso médio:** ➔ **O(n^2)**

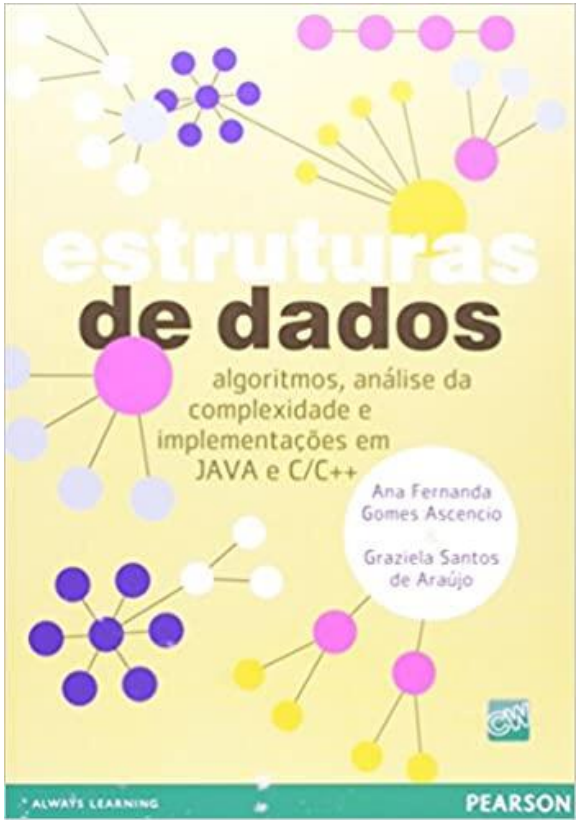
A complexidade do Bubble Sort é O(n^2)

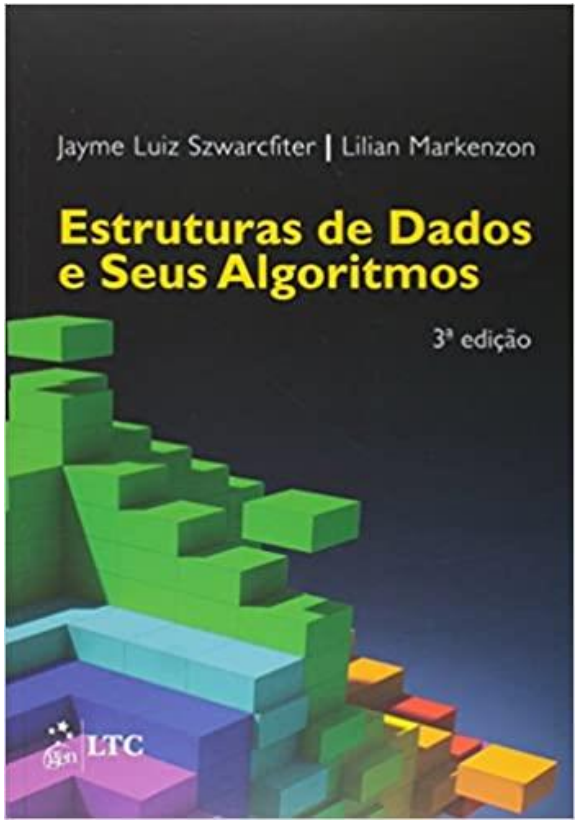
Complexidade

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 = $O(1)$	Não	Sim	Não	Sim

Referências Bibliográficas







Material de Programação II. Professores de Programação II e Laboratório II. Acessado em 01/03/2022.