Introdução à Complexidade Computacional

Algoritmos e Programação 2

Prof. Dr. Anderson Bessa da Costa
Universidade Federal de Mato Grosso do Sul

O que é um Algoritmo?

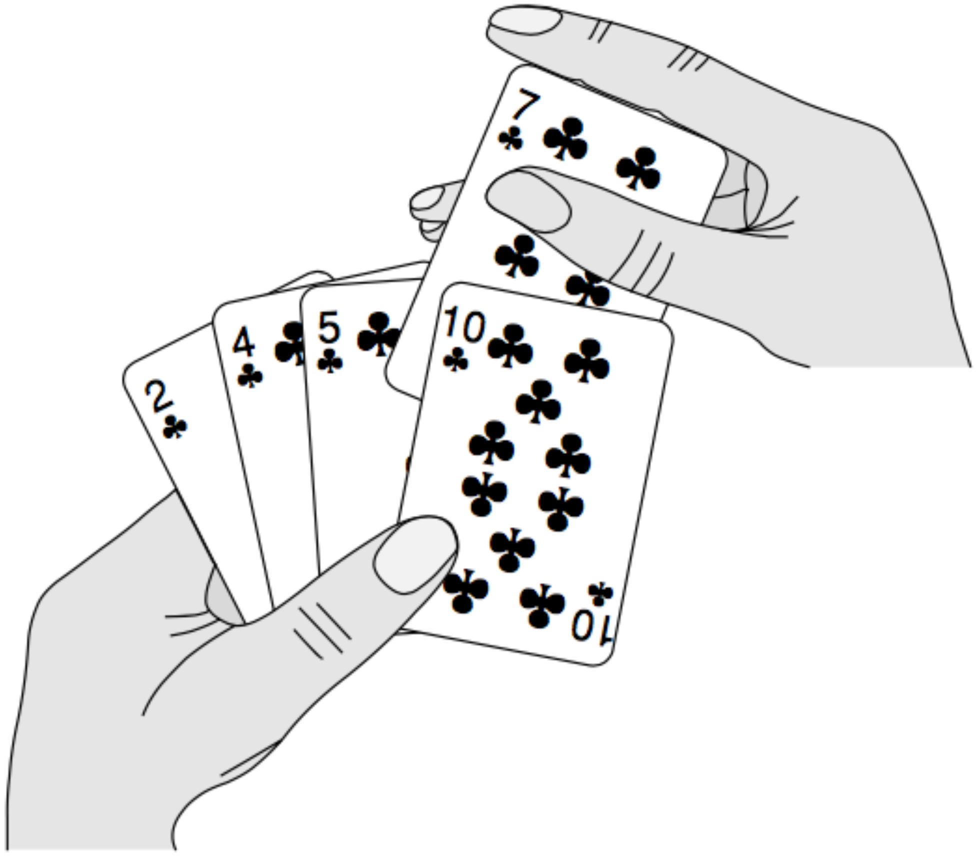
Um **algoritmo** é qualquer procedimento computacional bem definido que recebe algum valor, ou conjunto de valores, como **entrada** e produz algum valor, ou conjunto de valores como **saída**

Problema de Ordenação

- **Entrada:** Uma sequência de n números $\langle a_1, a_2, \ldots, a_n \rangle$
- **Saída:** Uma permutação $\langle a'_1, a'_2, \ldots, a'_n \rangle$ da sequência de entrada tal que $a'_1 \leq a'_2 \leq \ldots \leq a'_n$
- Por exemplo, dado a sequência de entrada: $\langle 31, 41, 59, 26, 41, 58 \rangle$
- Um algoritmo de ordenação retorna como saída a sequência $\langle 26, 31, 41, 41, 58, 59 \rangle$

Ordenação por Inserção

Ordenação por inserção (***insertion sort***) resolve o problema de ordenação apresentado anteriormente



Ordenando uma mão de cartas usando ordenação por inserção.

Ordenação por Inserção: Algoritmo (1)

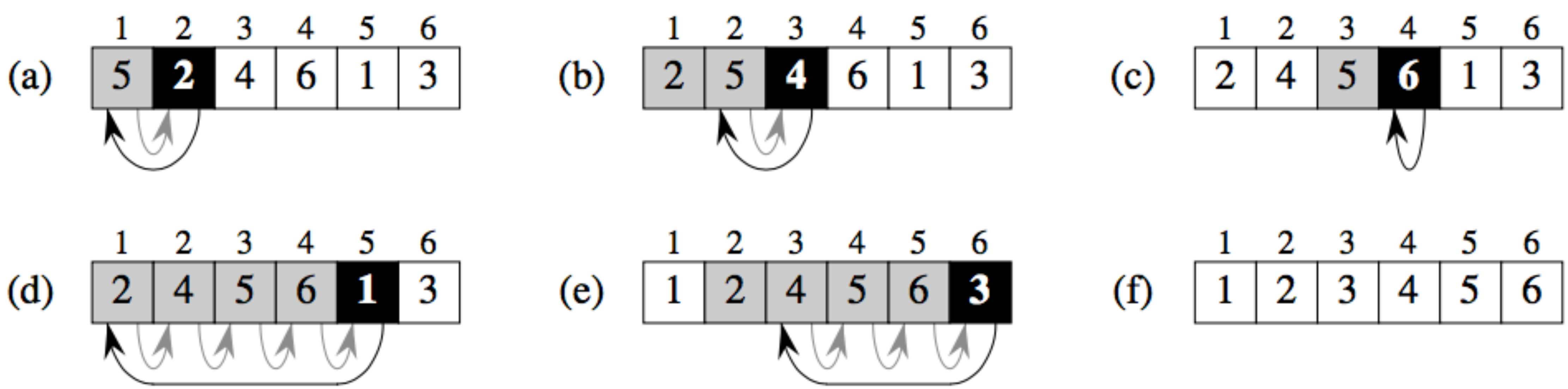
```c
void insertion_sort(int v[], int n) {
    int i, j, chave;

    for(j = 1; j < n; j++) {
        chave = v[j];
        i = j - 1;
        // insere A[j] na sequência ordenada A[0..j-1].
        while (i >= 0 && v[i] > chave) {
            v[i+1] = v[i];
            i = i - 1;
        }
        v[i+1] = chave;
    }
}
```

Ordenação por Inserção: Algoritmo (2)

```
função ordenacao_por_insercao (A: vetor[] de inteiro, n: inteiro)
var
  i, j, chave: inteiro
início
  para j de 2 até n faça
    chave <- A[j]
    // Insere A[j] na sequência ordenada A[1..j-1].
    i <- j - 1
    enquanto i > 0 e A[i] > chave faça
      A[i+1] <- A[i]
      i <- i - 1
    fim_enquanto
    A[i+1] <- chave
  fim_para
fim_função
```

Ordenação por Inserção: Passo a Passo



Ordenação por inserção em um vetor $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Os índices aparecem acima do retângulo, e os valores armazenados aparecem dentro dos retângulos.

Estudo de Algoritmos

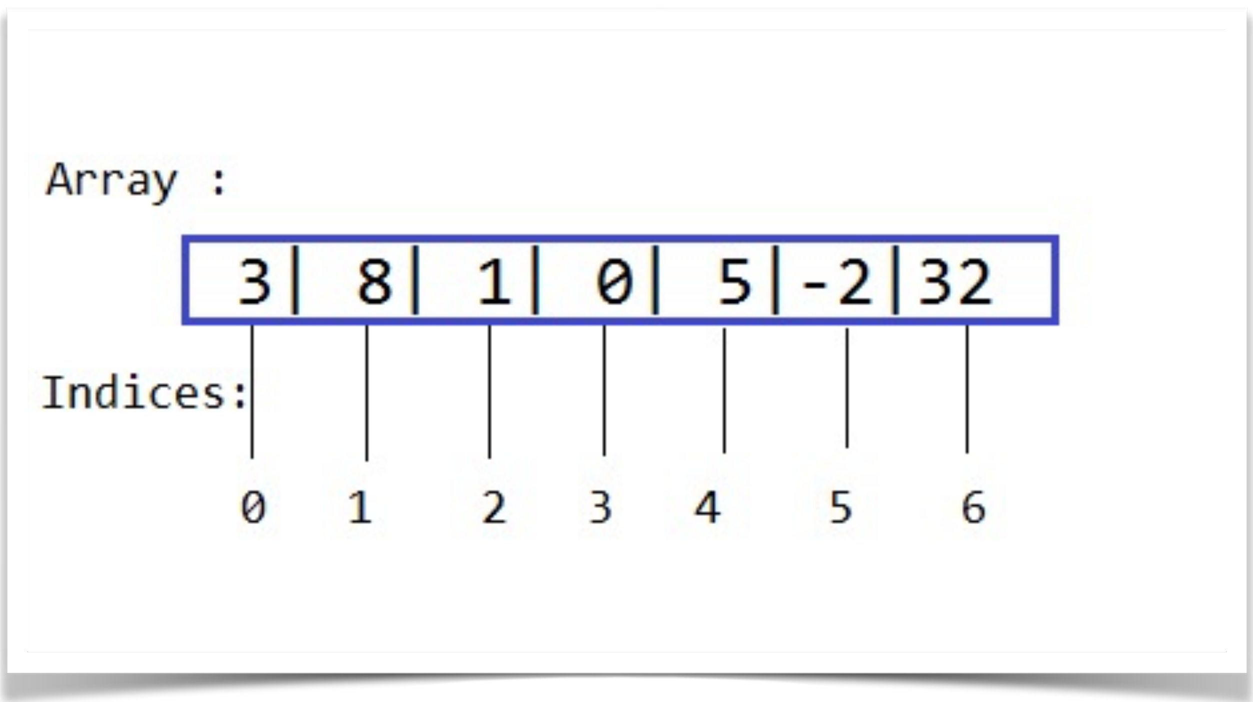
- Dois aspectos estudo de algoritmos: **correção** e **análise**
- **Correção** verifica exatidão do método, o que é realizado através de uma prova matemática
- **Análise** visa à obtenção de parâmetros para avaliar eficiência do algoritmo em termos de tempo de execução e memória ocupada

Análise Ordenação por Inserção

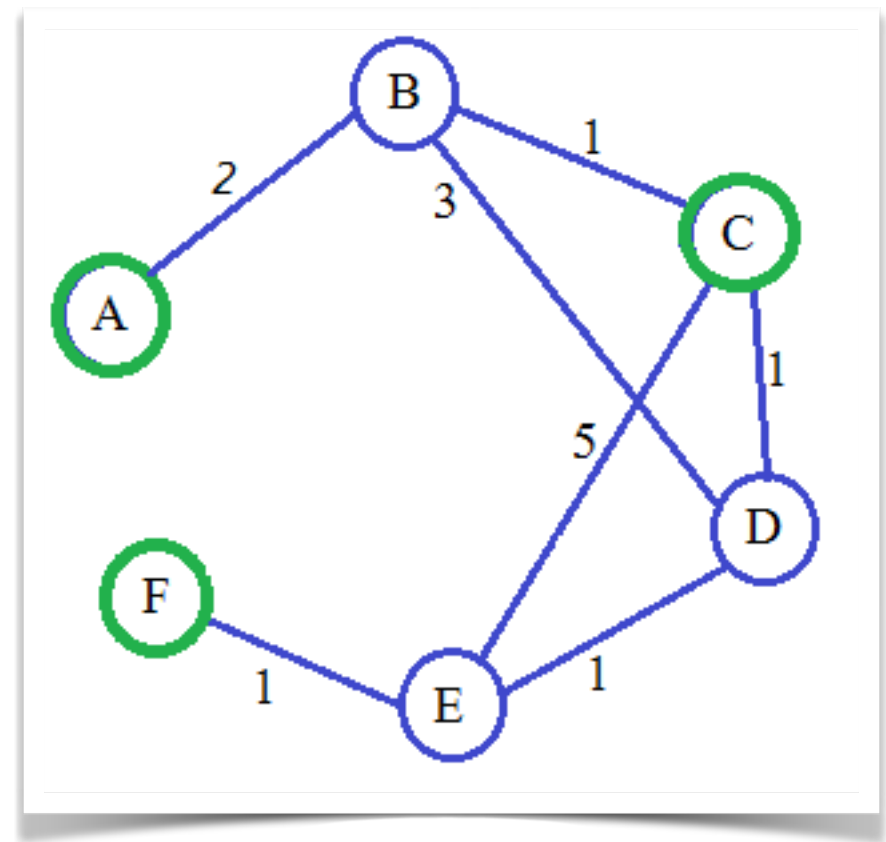
Tempo Gasto do Algoritmo

- Depende da entrada
    - Ordenar milhares de números leva mais tempo que ordenar três números
- Pode-se levar diferentes quantidades de tempo para entradas do mesmo tamanho
- Em geral, o **tempo gasto por um algoritmo cresce com o tamanho da entrada**, então tradicionalmente descrevemos o tempo de execução de um programa como uma função do tamanho da entrada

Tempo Proporcional à Entrada



Em um algoritmo de ordenação, o tempo do algoritmo em geral é proporcional a quantidade de elementos que se deseja ordenar.



Em um algoritmo de busca em um grafo, o tempo do algoritmo pode ser proporcional ao número de vértices ou número de arestas.

Tempo de Execução

- É o número de operações primitivas ou "passos" executados
  - É conveniente definir a noção de passo como sendo independente de máquina quanto possível
  - Por conveniência, definiremos que uma quantidade de tempo constante é necessária para executar cada linha do pseudocódigo

Ordenação por Inserção: Tempo de Execução

```
função ordenacao_por_insercao (A: vetor[] de inteiro, n : inteiro)
var
    i, j, chave : inteiro
início
    para j de 2 até n faça
     chave <- A[j]
     // Insere A[j] na sequência ordenada A[1..j-1].
     i <- j - 1
     enquanto i > 0 e A[i] > chave faça
          A[i+1] <- A[i]
         i <- i - 1
     fim_enquanto
     A[i+1] <- chave
  fim_para
fim_função
```

Ordenação por Inserção: Tempo de Execução

```
1. função ordenacao_por_insercao (A: vetor[]
   de inteiro, n : inteiro)
2. var
3.     i, j, chave : inteiro
4. início
5.     para j de 2 até n faça
6.      chave <- A[j]
7.      // Insere A[j] na sequência ordena
   A[1..j-1].
8.      i <- j - 1
9.      enquanto i > 0 e A[i] > chave faça
10.          A[i+1] <- A[i]
11.         i <- i - 1
12.      fim_enquanto
13.      A[i+1] <- chave
14.  fim_para
15.fim_função
```

Linha	**Custo**	**#**
5	c_1	n
6	c_2	$n-1$
7	0	$n-1$
8	c_4	$n-1$
9	c_5	$\sum_{j=1}^{n} t_j$
10	c_6	$\sum_{j=2}^{n}(t_j-1)$
11	c_7	$\sum_{j=2}^{n}(t_j-1)$
13	c_8	$n-1$

Estimando Tempo de Execução

- Tempo de execução é a soma de todos os tempos de cada instrução
- Expressão de custo c_j executada n vezes contribuirá $c_j \times n$
- Depende de qual entrada daquele tamanho é fornecida

Tempo Execução Ordenação por Inserção

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$

$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)\,.$$

Entrada

- Mesmo para entradas do mesmo tamanho, o tempo de execução do algoritmo pode depender de **qual** entrada
- O **melhor caso** ocorre se o vetor já está ordenado
    - Para cada $j = 2, 3, \ldots, n$, nós encontramos que $A[i] \leq chave$ …

Pior, Melhor e Médio

- Seja um algoritmo A, $\{E_1, \ldots, E_m\}$, o conjunto de todas as entradas possíveis de A. Denote por t_i, o número de passos efetuados por A, quando a entrada for E_i. Definem-se
  - Complexidade do pior caso = $\max\limits_{E_i \in E}\{t_i\}$
  - Complexidade do melhor caso = $min_{E_i \in E}\{t_i\}$
  - Complexidade do caso médio $\sum\limits_{1 \leq i \leq m} p_i t_i$

  onde p_i é a probabilidade de ocorrência da entrada E_i.

Melhor Caso

- Ocorre se o vetor já está ordenado
- Para cada $j = 2,3,\ldots,n$, nós temos que $A[i] \leq chave$ na linha 9, quando i tem valor inicial de $j-1$. Assim, $t_j = 1$, para $j = 2,3,\ldots,n$ e o melhor caso é:

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\
&= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)
\end{aligned}
$$

Melhor Caso (cont.)

- Este tempo de execução pode ser expresso como $an + b$ para constantes a e b que dependem dos custos c_i
- Assim, dizemos que é uma **função linear** de n

Pior Caso

- Ocorre se vetor estiver ordenado na ordem contrária
- Nós devemos então comparar cada elemento de $A[j]$ com cada elemento do sub vetor ordenado $A[1..j-1]$, e então $t_j = j$ para $j = 2,3,...,n$.
- Note que:

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \qquad \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

Pior Caso (cont.)

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(\frac{n(n+1)}{2} - 1)$$
$$+c_6(\frac{n(n-1)}{2}) + c_7(\frac{n(n-1)}{2}) + c_8(n-1)$$
$$= (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2})n^2 + (c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8)n$$
$$-(c_2 + c_4 + c_5 + c_8)\,.$$

- Temos $an^2 + bn + c$ para constantes a, b e c que, novamente, dependem dos custos de c_i
- Dizemos que esta é uma **função quadrática** de n

Por que Pior Caso?

- De maneira geral estaremos interessado no **pior caso**:

1. É o limite superior do tempo de execução;
2. Ocorre com bastante frequência;
3. O caso médio é frequentemente tão ruim quanto o pior caso.

Abstrações na Análise

- Nós fizemos algumas abstrações para simplificar a nossa análise do *insertion sort*:
    - Ignoramos o custo real de cada instrução, por meio do uso de constantes c_i para representar estes custos
    - Ignoramos não apenas o custo real das instruções, mas também o custo abstrato c_i
        - Dissemos que o tempo de execução no pior caso é $an^2 + bn + c$ para alguma constante a, b e c

Ordem de Crescimento

- É a **taxa de crescimento**, ou **ordem de crescimento**, do tempo de execução que nos interessa
- Estamos interessados apenas no termo dominante da fórmula (e.g., an^2)
  - Os termos de ordem mais baixa são relativamente insignificantes para n grande
- Nós também ignoramos as constantes que multiplicam os termos dominantes
  - Fatores constantes são menos importantes que a taxa de crescimento para determinar eficiência computacional para entradas grandes

Notação Θ (Theta)

- Podemos escrever que o algoritmo *insertion sort*, tem o tempo de execução no pior caso de $\Theta(n^2)$ ("**theta de n ao quadrado**")
- Consideraremos um algoritmo ser mais **eficiente** que outro se o tempo de execução no pior caso tem uma ordem de crescimento menor
    - Algoritmo $\Theta(n^2)$ é mais eficiente que um algoritmo $\Theta(n^3)$
    - Devido a constantes (aditivas e multiplicativas), esta avaliação pode ser falha para pequenas entradas
    - Porém para grandes entradas é válida

Eficiência

- Considere ***insertion sort*** com complexidade $\Theta(n^2)$
- Considere ***merge sort*** com complexidade $\Theta(n \lg n)$
- Suponha que ambos tenham que ordenar um vetor de 1 milhão de números
- Computador A executa 1 bilhão de instruções por segundo
- Computador B executa apenas 10 milhões de instruções por segundo (100x mais rápido que o A)
- Suponha insertion sort $2n^2$ $(c_1 = 2)$ instruções
- Suponha merge sort $50n \lg n$ $(c_2 = 50)$ instruções

Ordenar 1 Milhão de Elementos

- $A = \dfrac{2 \cdot (10^6)^2 \text{ instruções}}{10^9 \text{ instruções/segundo}} = 2000 \text{ segundos}$

- $B = \dfrac{50 \cdot 10^6 \lg 10^6 \text{ instruções}}{10^7 \text{ instruções/segundo}} \approx 100 \text{ segundos}$

Tempo de Execução

- Utilizando um algoritmo que cresce mais devagar, mesmo com uma constante mais alta e um computador pior, executa mais rápido
- A vantagem do ***merge sort*** seria maior ainda considerando 10 milhões de números
  - ***Insertion sort*** levaria 2.3 dias
  - ***Merge sort*** levaria 20 minutos

Comparativo Tempo de Execução

- Para cada função $f(n)$ e tempo t, determine o maior tamanho n de um problema que pode ser resolvido em tempo t, assumindo que o algoritmo para resolver o problema leva $f(n)$ microsegundos

We assume a 30 day month and 365 day year.

	1 Second	1 Minute	1 Hour	1 Day	1 Month	1 Year	1 Century
$\lg n$	$2^{1\times 10^6}$	$2^{6\times 10^7}$	$2^{3.6\times 10^9}$	$2^{8.64\times 10^{10}}$	$2^{2.592\times 10^{12}}$	$2^{3.1536\times 10^{13}}$	$2^{3.15576\times 10^{15}}$
$\sqrt{n}$	1×10^{12}	3.6×10^{15}	1.29×10^{19}	7.46×10^{21}	6.72×10^{24}	9.95×10^{26}	9.96×10^{30}
n	1×10^6	6×10^7	3.6×10^9	8.64×10^{10}	2.59×10^{12}	3.15×10^{13}	3.16×10^{15}
$n \lg n$	62746	2801417	133378058	2755147513	71870856404	797633893349	6.86×10^{13}
n^2	1000	7745	60000	293938	1609968	5615692	56176151
n^3	100	391	1532	4420	13736	31593	146679
2^n	19	25	31	36	41	44	51
$n!$	9	11	12	13	15	16	17

Exercício 1

Expresse a função $n^3/1000 - 100n^2 - 100n + 3$ em termos de notação Θ.

Exercício 2

Ordenação por flutuação (*bubble sort*) é um algoritmo de ordenação que repetidamente percorre o vetor, compara pares adjacentes e troca-os se eles estão na ordem errada. O passo de percorrer o vetor é repetido até o que o mesmo seja ordenado. Faça a mesma análise de complexidade do melhor e pior caso para o algoritmo ordenação por flutuação (*bubble sort*).

Ordenação por Flutuação: Algoritmo

```
função ordenacao_por_flutuacao (A: vetor[] de inteiro, n: inteiro)
var
  i, j, aux: inteiro
início
  para i de 1 até n−1 faça
    para j de 1 até n − i
      // se verdadeiro, troque valores
      se A[j] > A[j+1] faça
        aux = A[j]
        A[j] = A[j+1]
        A[j+1] = aux
      fim_se
    fim_para
  fim_para
fim_função
```

Referências

- CORMEN, T. H.[et al]. Algoritmos: teoria e prática. 3ª ed. Rio de Janeiro: Elsevier, 2012.
- SZWARCFITER, Jayme Luiz; MARKENZON, Lilian. Estruturas de dados e seus algoritmos. 3. ed. Rio de Janeiro, RJ: LTC, 2010. 302p.