Capítulo 3

# Algoritmos de ordenação simples

Entrar em um local desorganizado com o desejo de encontrar algo específico pode ser uma missão quase impossível. Muitas vezes, horas são perdidas para encontrar o que se procura. Por outro lado, quando existe uma determinada organização no ambiente, essa mesma tarefa acaba sendo prática, rápida e eficiente.

Em computação, o cenário descrito também ocorre. Quando os dados se mantêm ordenados, encontrar um determinado objeto é mais rápido, pois, para isso, podem ser utilizados algoritmos como o da busca binária, que encontra um elemento em, aproximadamente, metade do tempo dos demais algoritmos.

O problema de ordenação, em computação, não é só estudado por questões de organização e ordenação de dados. As possibilidades de pesquisas sobre esses algoritmos vão muito além disso. As funções apresentadas na construção dos algoritmos de ordenação e as técnicas utilizadas tornam-se referência para a solução de diversos outros problemas, principalmente em áreas como inteligência artificial. Outro ponto interessante é o seu uso didático na solução de problemas. Apesar de ser uma ciência exata, solucionar problemas computacionais pode apresentar muitas alternativas. Algoritmos de ordenação oferecem múltiplas soluções para um mesmo problema (WIRTH, 1989).

Graças às diversas possibilidades de solução, o uso dos algoritmos de ordenação vai além do simples fato de ordenar. Esse tipo de algoritmo tornou-se um *benchmark* (teste realizado para classificar os melhores métodos de acordo com um parâmetro) para comparações e análises de complexidade e de desempenho dos algoritmos (WIRTH, 1989). Existem duas classificações de aplicação para os algoritmos de ordenação: memória interna e memória externa.

Algoritmos que trabalham com memória interna geralmente ordenam estruturas de dados alocadas nas memórias do computador, como a RAM e a cache. Tendem a ser algoritmos mais rápidos, já que o acesso a esse tipo de memória é aleatório e a velocidade de leitura e de escrita é bem maior do que a de um disco rígido, por exemplo. Já os algoritmos de memória externa são mais lentos, pois são utilizados na organização de arquivos que, por sua vez, estão armazenados em dispositivos físicos como discos rígidos, SSDs e mídias. Outro aspecto ao qual se atribui a lentidão dos algoritmos de memória externa é que o

acesso a esse tipo de dispositivo é feito de maneira sequencial, impossibilitando algumas técnicas de programação em memórias de acesso aleatório. Além disso, dispositivos físicos possuem uma limitação de tamanho muito maior que as memórias RAM, por exemplo, o que torna o volume de dados muito grande, mesmo para uma ordenação simples (WIRTH, 1989).

Neste capítulo, trataremos apenas dos algoritmos de memória interna. Entre os mais comuns, existem diversas técnicas que são utilizadas para atingir o objetivo de ordenação de dados ou informações. Os algoritmos que apresentam as técnicas mais simples são o bubble sort, ou ordenação por bolhas, o insertion sort e o selection sort. Eles serão apresentados e discutidos nos tópicos a seguir.

# 1 Bubble sort

Dentre os métodos de ordenação existentes, o mais simples de ser implementado é o de ordenação por bolhas. É possível encontrar também, na literatura, esse método denominado como classificação por bolhas ou bubble sort, nome em inglês. O método leva esse nome pois o processo todo do algoritmo se assemelha a bolhas de gás de um refrigerante (TENENBAUM; LANGSAM; AUGENSTEIN, 2004). Quando uma bolha está leve, ela vai trocando de posição com as bolhas mais pesadas até alcançar a borda do copo.

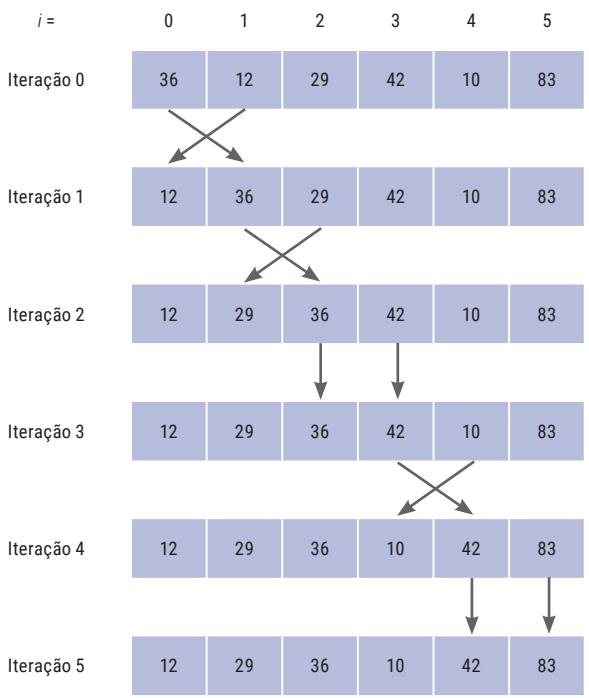
Por ter implementação e compreensão simples, geralmente, ele é o primeiro tipo a ser estudado pelos iniciantes em algoritmos. Contudo, a facilidade de implementação faz com que ele seja, provavelmente, o algoritmo de ordenação menos eficiente (TENENBAUM; LANGSAM; AUGENSTEIN, 2004). O algoritmo funciona da seguinte maneira: ele percorre o vetor ou o arquivo sequencialmente por várias iterações. A cada iteração, ele compara o elemento em evidência com o seu sucessor. Em termos formais, temos $v[i]$ com $v[i + 1]$, em que $v$ é o vetor ou o

arquivo em processo de ordenação. Após a comparação, se o sucessor for menor que o elemento *v*[*i*], eles são trocados de posição. Quando a primeira passagem pelo vetor é concluída, o elemento *v*[*n* – 1], sendo *n* a quantidade de elementos armazenados no vetor, encontra-se na posição correta (TENENBAUM; LANGSAM; AUGENSTEIN, 2004).

A figura 1, a seguir, apresenta a primeira passagem do algoritmo bubble sort por um vetor de inteiros com 6 elementos. O vetor apresentado possui 6 números inteiros na seguinte ordem [36, 12, 29, 42, 10, 83]. No seu início, é estabelecida a variável *i*, que controlará o índice atual da iteração do algoritmo. Os possíveis valores de *i* são apresentados acima da ilustração de cada posição do vetor correspondente. Logo abaixo está o primeiro passo da iteração, em que o valor de *i* é igual a 0. Sendo assim, existe uma comparação entre os valores de *v*[0] e *v*[1]. Como *v*[0] é maior que *v*[1], é realizada uma troca entre esses dois elementos. Inicia-se então o valor de iteração 1. Agora, a comparação é entre *v*[1] e *v*[2], ocorrendo novamente a troca, pois o valor de *v*[1] é maior. Imediatamente depois, é iniciada a iteração de número 2. Nessa iteração, os valores se mantêm em suas posições, porque o valor posterior *v*[3] é maior que o atual *v*[2]. A próxima iteração é a de número 3, e nessa também ocorre uma troca. Na iteração 4, os valores são mantidos, finalizando o processo na iteração 5, na qual o maior valor entre os elementos é posicionado no final do vetor.

Nesse momento, pode-se considerar que o último elemento do vetor está posicionado no local correto de ordenação. Contudo, o algoritmo deve seguir as demais passagens até que todo o vetor esteja ordenado.

**Figura 1 – Primeira passagem do bubble sort**




Agora, entenda como é feita a implementação do bubble sort:

```
1  import java.util.Arrays;
2
3  public class BubbleSort {
4
5      public static void bubbleSort(int[] v) {
6             for (int i = 0; i < v.length; i++) {
7                    for (int j = 0; j < v.length - 1;
j++)
8                           if (v[j] > v[j + 1])
9                                  troca(v, j, j + 1);
10                   System.out.printf("Passagem %d -> %s
\n",
11                                 i, Arrays.toString(v));
12            }
```

```
13      }
14
15      public static void troca(int[] v, int a, int b) {
16              int aux = v[a];
17              v[a] = v[b];
18              v[b] = aux;
19      }
20
21      public static void main(String[] args) {
22              int[] v = {36, 12, 29, 42, 10, 83};
23              System.out.printf("Início -> %s \n",
24                              Arrays.toString(v));
25              bubbleSort(v);
26              System.out.printf("Fim -> %s \n",
27                              Arrays.toString(v));
28      }
29 }
```

Algumas considerações sobre o código-fonte apresentado:

- O método troca apresentado entre as linhas 15 e 19 foi criado para auxiliar no processo de troca dos valores de posição dentro do vetor apenas para facilitar e deixar o código mais legível.
- No método bubble sort, nas linhas de 5 a 13, foram necessários dois laços de repetição do tipo *for*. O primeiro, com a variável de controle *i*, representa a passagem pelo vetor. O *for* com a variável de controle *j* representa as iterações ilustradas na figura 1.
- Na linha 10, foi inserida uma saída do programa para verificar o resultado de cada passagem do algoritmo. Entretanto, no algoritmo original, não existe essa necessidade, ela foi utilizada apenas para fins didáticos.

Outro ponto importante sobre essa implementação é que, mesmo que o vetor esteja completamente ordenado, as passagens são executadas até o final. Realizar essas passagens com o vetor já ordenado apenas deixa o algoritmo mais ineficiente. Veja a saída da implementação e observe o resultado e o efeito mencionado:

```
Início     -> [36, 12, 29, 42, 10, 83]
Passagem 0 -> [12, 29, 36, 10, 42, 83]
Passagem 1 -> [12, 29, 10, 36, 42, 83]
Passagem 2 -> [12, 10, 29, 36, 42, 83]
Passagem 3 -> [10, 12, 29, 36, 42, 83]
Passagem 4 -> [10, 12, 29, 36, 42, 83]
Passagem 5 -> [10, 12, 29, 36, 42, 83]
Fim        -> [10, 12, 29, 36, 42, 83]
```

Na saída obtida, é possível observar que, a partir da passagem 3, o algoritmo já ordenou por completo o vetor. Nesse ponto, não há a necessidade de mais passagens. Outro ponto que pode ser melhorado é o número de iterações a cada passagem. Para isso, é preciso subtrair o valor da passagem do tamanho do vetor no segundo *for*. Esse procedimento fará com que os valores já ordenados, as últimas posições do vetor, não sejam mais comparadas. Todas as medidas tomadas a partir desse momento são para aumentar o desempenho do algoritmo.

A seguir, apresentamos a nova versão do algoritmo bubble sort:

```
1   public class BubbleSort {
2
3      public static void bubbleSort(int[] v) {
4             boolean troca = true;
5             for (int i = 0; i < v.length && troca; i++)
{
6                    troca = false;
7                    for (int j = 0; j < v.length - i - 1;
j++)  {
8                           if (v[j] > v[j + 1]) {
9                                  trocar(v, j, j + 1);
10                                 troca = true;
11                          }12                  }
13                   System.out.printf("Passagem %d -> %s
\n",
```

```
14                                  i, Arrays.toString(v));
15            }
16      }
17
18      public static void trocar(int[] v, int a, int b) {
19            int aux = v[a];
20            v[a] = v[b];
21            v[b] = aux;
22      }
23
24      public static void main(String[] args) {
25            int[] v = {36, 12, 29, 42, 10, 83};
26            System.out.printf("Início     -> %s \n",
27                        Arrays.toString(v));
28            bubbleSort(v);
29            System.out.printf("Fim        -> %s \n",
30                        Arrays.toString(v));
31      }
32  }
```

Perceba que a implementação realizada apresentou uma variável a mais de controle (troca) e a iteração não é executada até o final do vetor, impedindo a comparação com as posições já ordenadas. Como o vetor do exemplo é pequeno e a quantidade de passagens também não é significativa, não é possível ver uma melhora expressiva. Contudo, quando o vetor estiver totalmente ordenado, ele fará apenas uma passagem pelo vetor e não mais *n* vezes *n* passagens como na versão anterior. Veja a saída da nova versão do método bubble sort:

```
Início     -> [36, 12, 29, 42, 10, 83]
Passagem 0 -> [12, 29, 36, 10, 42, 83]
Passagem 1 -> [12, 29, 10, 36, 42, 83]
Passagem 2 -> [12, 10, 29, 36, 42, 83]
Passagem 3 -> [10, 12, 29, 36, 42, 83]
Passagem 4 -> [10, 12, 29, 36, 42, 83]
Fim        -> [10, 12, 29, 36, 42, 83]
```

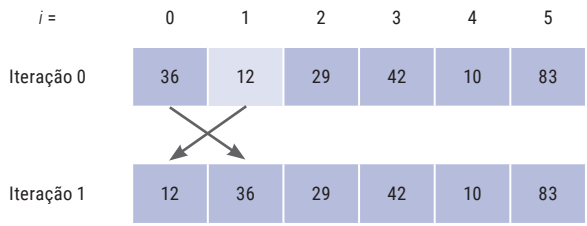
Perceba que, nesse caso, apenas uma passagem da versão inicial do algoritmo foi subtraída. Isso ocorre porque, na passagem 3, o algoritmo finaliza a ordenação efetivamente. Na passagem 4 não ocorre nenhuma troca, o que finaliza o processo de ordenação. Assim, a passagem 5, existente na versão inicial, não ocorre nessa nova versão. Em vetores com tamanhos maiores, é possível ter um ganho mais significativo.

## 2 Insertion sort

Imagine um jogo de pôquer: os jogadores recebem, cada um, um conjunto de cartas e as colocam nas mãos. A entrega das cartas é aleatória, portanto, sem nenhuma ordem específica. Para facilitar e agilizar suas jogadas, o jogador as organiza em ordem crescente da esquerda para a direita. Para ordenar as cartas, o jogador, geralmente, segura as cartas com uma das mãos e com a outra tira as cartas menores do final e as coloca na ordem correta a partir da carta menor até a carta maior. Esse processo é chamado de ordenação por inserção, ou insertion sort, em inglês (CORMEN *et al*., 2002).

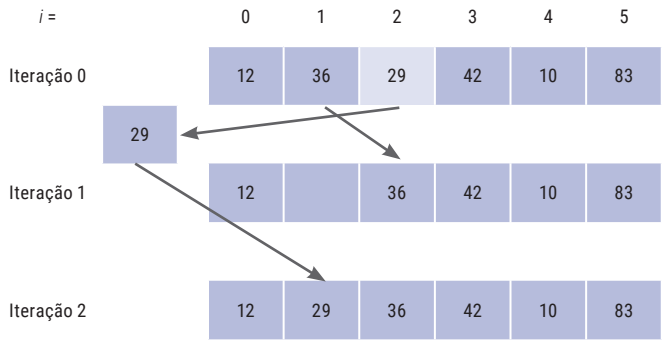
O insertion sort é um algoritmo de ordenação que efetua esse processo removendo os elementos menores do final do vetor e os reinserindo no início, já em sua posição final. O passo a passo do algoritmo começa com a seleção a partir da segunda posição do vetor. Ele copia esse elemento para uma variável auxiliar, depois ele o compara com todos os elementos à sua esquerda. Se ele for menor que o elemento à esquerda, ele continua procurando a posição onde deverá ser inserido. O critério para a posição é: elemento à esquerda menor; à direita, maior (CORMEN *et al*., 2002; ASCENCIO; ARAÚJO, 2010). Veja na figura 2 o primeiro passo do algoritmo insertion sort:

**Figura 2 – Primeira passagem do algoritmo insertion sort no vetor**




Na figura 2, é apresentada a primeira passagem do algoritmo insertion sort. O elemento da segunda posição é selecionado (em cor mais clara). Nesse momento, ele é armazenado em uma variável auxiliar. A partir disso, ele será comparado com todos os elementos à esquerda e será posicionado no local de ordenação do vetor. Como nessa primeira iteração é realizada apenas uma comparação, a troca ocorre entre a primeira posição e a segunda, já que o elemento 12 (segunda posição) é menor que o elemento 36 (primeira posição). Para continuar o processo de compreensão do algoritmo, analise a figura 3. Ela apresenta a segunda passagem do algoritmo.

**Figura 3 – Segunda passagem do algoritmo insertion sort no vetor**




Na figura 3, é possível observar o passo a passo da segunda passagem do algoritmo insertion sort. Nesse momento, o elemento selecionado é o 29 (em cor mais clara). Ele é armazenado em uma variável separada, representada pelo quadrado flutuante entre as iterações 0 e 1, e, em seguida, é comparado com o primeiro elemento à sua esquerda, no caso o 36, que ocupa a posição de índice 1 no vetor. Como 36 é maior

que 29, ele é copiado para a posição de índice 2, ocupada antes pelo 29. Agora, o algoritmo compara a posição de índice 0 que tem o valor 12, com o 29. Como 12 é menor que 29, as comparações se encerram e o valor selecionado é copiado na posição vazia do vetor, que, nesse caso, é a 1. Veja a implementação do código em Java:

```
1  public class InsertionSort {
2
3      public static void insertionSort (int[] v) {
4              for (int i = 1; i < v.length; i++) {
5                      int x = v[i];
6                      for (int j = i - 1; j >= 0 && v[j] >
x; j--) {
7                              v[j + 1] = v[j];
8                              v[j] = x;
9                      }
10                     System.out.printf("Iteração do nro %d
-> %s \n",
11                                    x, Arrays.toString(v));
12             }
13     }
14
15     public static void main(String[] args) {
16             int[] v = {36, 12, 29, 42, 10, 83};
17             System.out.printf("Início             -> %s
\n",
18                             Arrays.toString(v));
19             insertionSort(v);
20             System.out.printf("Fim                -> %s
\n",
21                             Arrays.toString(v));
22     }
23 }
```

O método de ordenação insertion sort é apresentado entre as linhas 3 e 13. Perceba que a variável *x* é a variável responsável por armazenar o elemento que será ordenado. Se fosse no jogo de pôquer, seria a carta que o jogador segurou em sua mão antes de decidir onde colocá-la. Na linha 6, existe um laço de repetição do tipo *for*, que é inicializado com o

primeiro elemento anterior a *x*, ou seja, à sua esquerda. Sendo decrescente esse passo, isso garante que apenas os elementos já ordenados serão percorridos. Para fins didáticos, na linha 10 foi inserido o passo a passo do algoritmo. Veja o resultado a seguir:

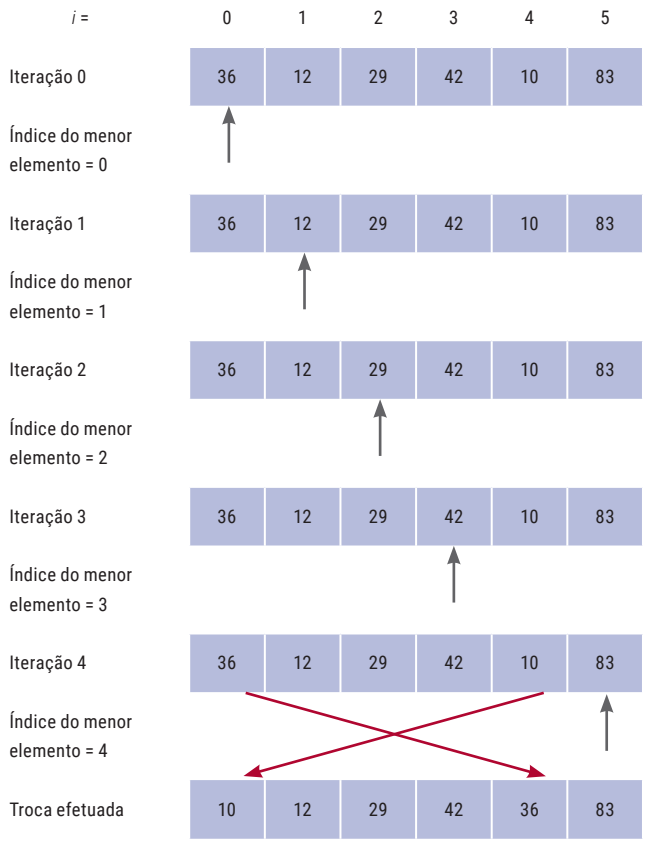
```
Início             -> [36, 12, 29, 42, 10, 83]
Iteração do nro 12 -> [12, 36, 29, 42, 10, 83]
Iteração do nro 29 -> [12, 29, 36, 42, 10, 83]
Iteração do nro 42 -> [12, 29, 36, 42, 10, 83]
Iteração do nro 10 -> [10, 12, 29, 36, 42, 83]
Iteração do nro 83 -> [10, 12, 29, 36, 42, 83]
Fim                -> [10, 12, 29, 36, 42, 83]
```

Perceba como os números em destaque na saída vão ficando ordenados a cada passagem do algoritmo, pois ele se preocupa apenas com os elementos à esquerda. No caso do número 42, por exemplo, ele já é maior que o 36 na primeira iteração. Isso faz com que ele seja mantido na posição e o algoritmo não tenha a necessidade de percorrer o vetor. Esse ponto é o que traz um diferencial em comparação com o bubble sort. O insertion sort, apesar de poder percorrer o vetor duas vezes, como o bubble sort, mantém-se com uma média de desempenho melhor (CORMEN *et al*., 2002; ASCENCIO; ARAÚJO, 2010).

# 3 Selection sort

O algoritmo de ordenação por seleção, ou selection sort, em inglês, apresenta uma terceira estratégia para realizar o processo de ordenação de um vetor. Em linhas gerais, o selection sort percorre todo o vetor em busca do menor elemento não ordenado. Quando ele o encontra, ele posiciona o valor mínimo na primeira posição não ordenada do vetor (CORMEN *et al*., 2002; ASCENCIO; ARAÚJO, 2010). Veja a primeira passagem do algoritmo para o vetor *v*, representado pela figura 4.

**Figura 4 – Primeira passagem do algoritmo selection sort**




O passo a passo da primeira passagem do algoritmo selection sort demonstra todo o funcionamento dele. Na primeira iteração, o índice do menor elemento é o próprio 0, pois é o primeiro elemento a ser comparado. Para a próxima iteração, o elemento que está na posição 1 é menor que o da posição 0, então o índice de menor elemento passa a ser 1. Na iteração 2, o valor 29 na posição 2 não é menor que o 12 na posição de menor elemento. O algoritmo mantém o índice escolhido. Na próxima iteração, de número 3, os valores também se mantêm, pois 42 é um número maior que 12. Seguindo para a iteração de número 4, existe uma atualização do índice de menor valor, pois 10 é menor que 12.

Sendo assim, o índice, nesse momento, passa a ser o de valor 4. Na última iteração, de número 5, não há nenhuma troca ou atualização, pois 83 é o maior valor do vetor do exemplo. Após esse passo, é realizada a troca do valor na posição 0 (36) com o da posição 4 (10). O algoritmo prossegue nesse passo a passo até que o vetor esteja completamente ordenado.

Agora que o passo a passo da execução do algoritmo foi estabelecido, deve-se implementar o método de ordenação. O exemplo a seguir é implementado utilizando a linguagem de programação Java:

```
1   public class SelectionSort {
2
3      public static void selectionSort(int[] v) {
4        for (int i = 0; i < v.length; i++) {
5                    int sindex = i;
6                    for (int j = i + 1; j < v.length;
j++) {
7                           if (v[j] < v[sindex]) sindex =
j;
8                    }
9                    trocar(v, i, sindex);
10                   System.out.printf("Iteração %d -> %s
\n",
11                                  i, Arrays.toString(v));
12       }
13    }
14
15     public static void trocar(int[] v, int a, int b) {
16            int aux = v[a];
17            v[a] = v[b];
18            v[b] = aux;
19     }
20
21     public static void main(String[] args) {
22            int[] v = {36, 12, 29, 42, 10, 83};
23            System.out.printf("Início     -> %s \n",
24                         Arrays.toString(v));
25            selectionSort(v);
```

```
26            System.out.printf("Fim        -> %s \n",
27                              Arrays.toString(v));
28      }
29  }
```

Na implementação realizada, é necessário que haja novamente a existência do método *trocar* para auxiliar no momento da permuta dos valores para o menor índice, veja as linhas de 15 a 19. Entre as linhas 3 e 13, ocorre a implementação do algoritmo selection sort. No laço de repetição principal, é possível verificar que o vetor é percorrido de ponta a ponta. O índice do menor elemento é inicializado na linha 5, com o valor de *i* dado no laço de repetição. Na linha 6, inicia-se o segundo laço de repetição que percorre o vetor da posição *i* + 1 até o seu fim. Ao final, na linha 9, o método *trocar* é acionado. Na linha 10, para fins didáticos, é exibido o passo a passo do algoritmo após as trocas do vetor. O resultado é este:

```
Início     -> [36, 12, 29, 42, 10, 83]
Iteração 0 -> [10, 12, 29, 42, 36, 83]
Iteração 1 -> [10, 12, 29, 42, 36, 83]
Iteração 2 -> [10, 12, 29, 42, 36, 83]
Iteração 3 -> [10, 12, 29, 36, 42, 83]
Iteração 4 -> [10, 12, 29, 36, 42, 83]
Iteração 5 -> [10, 12, 29, 36, 42, 83]
Fim        -> [10, 12, 29, 36, 42, 83]
```

No final, o selection sort apresentou o pior desempenho entre os três algoritmos apresentados, pois ele não possibilita nenhuma modificação para garantir um melhor desempenho de maneira trivial.

# 4 Exercícios de fixação

Para praticar, segue uma lista de exercícios:

1. Execute o algoritmo bubble sort para os vetores abaixo e compare as diferenças entre os resultados:
   a. [36, 15, 75, 2, 16, 48, 42, 51]
   b. [13, 45, 38, 20, 23, 7, 6, 5, 10, 11]
   c. [9, 8, 7, 6, 5, 4, 3, 2, 1]
2. Execute o algoritmo insertion sort para os vetores abaixo e compare as diferenças entre os resultados:
   a. [36, 15, 75, 2, 16, 48, 42, 51]
   b. [13, 45, 38, 20, 23, 7, 6, 5, 10, 11]
   c. [9, 8, 7, 6, 5, 4, 3, 2, 1]
3. Execute o algoritmo selection sort para os vetores abaixo e compare as diferenças entre os resultados:
   a. [36, 15, 75, 2, 16, 48, 42, 51]
   b. [13, 45, 38, 20, 23, 7, 6, 5, 10, 11]
   c. [9, 8, 7, 6, 5, 4, 3, 2, 1]
4. Compare os resultados dos três algoritmos.

# Considerações finais

Neste capítulo, foi possível perceber como existem diversas soluções para um mesmo problema. Os algoritmos demonstrados apresentam, em média, o mesmo desempenho computacional. Alguns deles podem ser otimizados por meio de truques em sua implementação e

até mesmo pelas próprias linguagens de programação. Esses algoritmos são os mais comuns e os mais fáceis de se implementar. Existem outros mais avançados, com técnicas que permitem que sejam muito mais rápidos que os apresentados aqui, mas eles serão abordados no próximo capítulo.

# Referências


ASCENCIO, Ana Fernanda Gomes; ARAÚJO, Graziela Santos de. **Estruturas de dados**: algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Pearson Prentice Hall, 2010.

CORMEN, Thomas H. *et al*. **Algoritmos**: teoria e prática. Rio de Janeiro: Editora Campus, 2002.

TENENBAUM, Aaron M.; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. **Estruturas de dados usando C**. São Paulo: MAKRON Books, 2004.

WIRTH, Niklaus. **Algoritmos e estruturas de dados**. Rio de Janeiro: Prentice Hall do Brasil, 1989.